
COMPOSER

简介

Composer 是 PHP 的一个依赖管理工具。它允许你申明项目所依赖的代码库，它会在你的项目中为你安装他们。

依赖管理

Composer 不是一个包管理器。是的，它涉及 "packages" 和 "libraries"，但它在每个项目的基础上进行管理，在你项目的某个目录中（例如 `vendor`）进行安装。默认情况下它不会在全局安装任何东西。因此，这仅仅是一个依赖管理。

这种想法并不新鲜，Composer 受到了 node's npm 和 ruby's bundler 的强烈启发。而当时 PHP 下并没有类似的工具。

Composer 将这样为你解决问题：

a) 你有一个项目依赖于若干个库。

b) 其中一些库依赖于其他库。

c) 你声明你所依赖的东西。

d) 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）。

声明依赖关系

比方说，你正在创建一个项目，你需要一个库来做日志记录。你决定使用 monolog。为了将它添加到你的项目中，你所需要做的就是创建一个 `composer.json` 文件，其中描述了项目的依赖关系。

```
{
    "require": {
        "monolog/monolog": "1.2.*"
    }
}
```

我们只要指出我们的项目需要一些 `monolog/monolog` 的包，从 `1.2` 开始的任何版本。

系统要求

运行 Composer 需要 PHP 5.3.2+ 以上版本。一些敏感的 PHP 设置和编译标志也是必须的，但对于任何不兼容项安装程序都会抛出警告。

我们将从包的来源直接安装，而不是简单的下载 zip 文件，你需要 git 、 svn 或者 hg ，这取决于你载入的包所使用的版本管理系统。

Composer 是多平台的，我们努力使它在 Windows 、 Linux 以及 OSX 平台上运行的同样出色。

安装 - *nix

下载 Composer 的可执行文件

局部安装

要真正获取 Composer，我们需要做两件事。首先安装 Composer （同样的，这意味着它将下载到你的项目中）：

```
curl -sS https://getcomposer.org/installer | php
```

> **注意：** 如果上述方法由于某些原因失败了，你还可以通过 `php` >下载安装器：

```
php -r "readfile('https://getcomposer.org/installer');" | php
```

这将检查一些 PHP 的设置，然后下载 `composer.phar` 到你的工作目录中。这是 Composer 的二进制文件。这是一个 PHAR 包（PHP 的归档），这是 PHP 的归档格式可以帮助用户在命令行中执行一些操作。

你可以通过 `--install-dir` 选项指定 Composer 的安装目录（它可以是一个绝对或相对路径）：

```
curl -sS https://getcomposer.org/installer | php -- --install-dir=bin
```

全局安装

你可以将此文件放在任何地方。如果你把它放在系统的 `PATH` 目录中，你就能在全局访问它。 在类Unix系统中，你甚至可以在使用时不加 `php` 前缀。

你可以执行这些命令让 `composer` 在你的系统中进行全局调用：

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

> **注意：** 如果上诉命令因为权限执行失败， 请使用 sudo 再次尝试运行 `mv` 那行命令。

现在只需要运行 `composer` 命令就可以使用 Composer 而不需要输入 `php composer.phar` 。

全局安装 (on OSX via homebrew)

Composer 是 homebrew-php 项目的一部分。

```
brew update
brew tap josegonzalez/homebrew-php
brew tap homebrew/versions
brew install php55-intl
brew install josegonzalez/php/composer
```

安装 - Windows

使用安装程序

这是将 Composer 安装在你机器上的最简单的方法。

下载并且运行 Composer-Setup.exe，它将安装最新版本的 Composer ，并设置好系统的环境变量，因此你可以在任何目录下直接使用 `composer` 命令。

手动安装

设置系统的环境变量 `PATH` 并运行安装命令下载 composer.phar 文件：

```
C:\Users\username>cd C:\bin
C:\bin>php -r "readfile('https://getcomposer.org/installer');" | php
```

> **注意：** 如果收到 readfile 错误提示，请使用 `http` 链接或者在 php.ini 中开启 php_openssl.dll 。

在 `composer.phar` 同级目录下新建文件 `composer.bat` ：

```
C:\bin>echo @php "%~dp0composer.phar" %*>composer.bat
```

关闭当前的命令行窗口，打开新的命令行窗口进行测试：

```
C:\Users\username>composer -V
Composer version 27d8904
```

使用 Composer

现在我们将使用 Composer 来安装项目的依赖。如果在当前目录下没有一个 `composer.json` 文件，请查看基本用法章节。

要解决和下载依赖，请执行 `install` 命令：

```
php composer.phar install
```

如果你进行了全局安装，并且没有 phar 文件在当前目录，请使用下面的命令代替：

```
composer install
```

继续 上面的例子，这里将下载 monolog 到 `vendor/monolog/monolog` 目录。

自动加载

除了库的下载，Composer 还准备了一个自动加载文件，它可以加载 Composer 下载的库中所有的类文件。使用它，你只需要将下面这行代码添加到你项目的引导文件中：

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

现在我们就可以使用 monolog 了！想要学习更多关于 Composer 的知识，请查看“基本用法”章节。

基本用法

安装

安装 Composer，你只需要下载 `composer.phar` 可执行文件。

```
curl -sS https://getcomposer.org/installer | php
```

详细请查看 简介 章节。

要检查 Composer 是否正常工作，只需要通过 `php` 来执行 PHAR：

```
php composer.phar
```

这将返回给你一个可执行的命令列表。

> **注意：** 你也可以仅执行 `--check` 选项而无需下载 Composer。 要获取更多的信息请使用 `--help`。
>
> ```
> curl -sS https://getcomposer.org/installer | php -- --help
> ```

`composer.json`：项目安装

要开始在你的项目中使用 Composer，你只需要一个 `composer.json` 文件。该文件包含了项目的依赖和其它的一些元数据。

这个 JSON format 是很容易编写的。它允许你定义嵌套结构。

关于 `require` Key

第一件事情（并且往往只需要做这一件事），你需要在 `composer.json` 文件中指定 `require` key 的值。你只需要简单的告诉 Composer 你的项目需要依赖哪些包。

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

你可以看到，`require` 需要一个 **包名称**（例如 `monolog/monolog`）映射到 **包版本**（例如 `1.0.*`）的对象。

包名称

包名称由供应商名称和其项目名称构成。通常容易产生相同的项目名称，而供应商名称的存在则很好的解决了命名冲突的问题。它允许两个不同的人创建同样名为 `json` 的库，而之后它们将被命名为 `igorw/json` 和 `seldaek/json`。

这里我们需要引入 `monolog/monolog`，供应商名称与项目的名称相同，对于一个具有唯一名称的项目，我们推荐这么做。它还允许以后在同一个命名空间添加更多的相关项目。如果你维护着一个库，这将使你可以很容易的把它分离成更小的部分。

包版本

在前面的例子中，我们引入的 monolog 版本指定为 `1.0.*`。这表示任何从 `1.0` 开始的开发分支，它将会匹配 `1.0.0`、`1.0.2` 或者 `1.0.20`。

版本约束可以用几个不同的方法来指定。

名称	实例	描述
确切的版本号	`1.0.2`	你可以指定包的确切版本。

范围	`>=1.0`
	
	`>=1.0,=1.0,=1.2`
		通过使用比较操作符可以指定有效的版本范围。 有效的运算符：`>`、`>=`、`<`、`<=`、`!=`。 你可以定义多个范围，用逗号隔开，这将被视为一个**逻辑AND**处理。一个管道符号 `|` 将作为**逻辑OR**处理。 AND 的优先级高于 OR。

通配符	`1.0.*`	你可以使用通配符 `*` 来指定一种模式。 `1.0.*` 与 `>=1.0,<1.1` 是等效的。
赋值运算符	`~1.2`	这对于遵循语义化版本号的项目非常有用。 `~1.2` 相当于 `>=1.2,<2.0`。想要了解更多，请阅读下一小节。

下一个重要版本（波浪号运算符）

`~` 最好用例子来解释： `~1.2` 相当于 `>=1.2,=1.2.3,<1.3`。正如你所看到的这对于遵循 语义化版本号 的项目最有用。一个常见的用法是标记你所依赖的最低版本，像 `~1.2` （允许1.2以上的任何版本，但不包括2.0）。由于理论上直到2.0应该都没有向后兼容性问题，所以效果很好。你还会看到它的另一种用法，使用 `~` 指定最低版本，但允许版本号的最后一位数字上升。

> **注意：** 虽然 `2.0-beta.1` 严格地说是早于 `2.0`，但是，根据版本约束条件， 例如 `~1.2` 却不会安装这个版本。就像前面所讲的 `~1.2` 只意味着 `.2` 部分可以改变，但是 `1.` 部分是固定的。

稳定性

默认情况下只有稳定的发行版才会被考虑在内。如果你也想获得 RC、beta、alpha 或 dev 版本，你可以使用 稳定标志。你可以对所有的包做 最小稳定性 设置，而不是每个依赖逐一设置。

安装依赖包

获取定义的依赖到你的本地项目，只需要调用 `composer.phar` 运行 `install` 命令。

```
php composer.phar install
```

接着前面的例子，这将会找到 `monolog/monolog` 的最新版本，并将它下载到 `vendor` 目录。 这是一个惯例把第三方的代码到一个指定的目录 `vendor`。如果是 monolog 将会创建 `vendor/monolog/monolog` 目录。

> **小技巧：** 如果你正在使用Git来管理你的项目， 你可能要添加 `vendor` 到你的 `.gitignore` 文件中。你不会希望将所有的代码都添加到你的版本库中。

另一件事是 `install` 命令将创建一个 `composer.lock` 文件到你项目的根目录中。

`composer.lock` - 锁文件

在安装依赖后，Composer 将把安装时确切的版本号列表写入 `composer.lock` 文件。这将锁定改项目的特定版本。

请提交你应用程序的 `composer.lock` （包括 `composer.json`）到你的版本库中

这是非常重要的，因为 `install` 命令将会检查锁文件是否存在，如果存在，它将下载指定的版本（忽略 `composer.json` 文件中的定义）。

这意味着，任何人建立项目都将下载与指定版本完全相同的依赖。你的持续集成服务器、生产环境、你团队中的其他开发人员、每件事、每个人都使用相同的依赖，从而减轻潜在的错误对部署的影响。即使你独自开发项目，在六个月内重新安装项目时，你也可以放心的继续工作，即使从那时起你的依赖已经发布了许多新的版本。

如果不存在 composer.lock 文件，Composer 将读取 composer.json 并创建锁文件。

这意味着如果你的依赖更新了新的版本，你将不会获得任何更新。此时要更新你的依赖版本请使用 update 命令。这将获取最新匹配的版本（根据你的 composer.json 文件）并将新版本更新进锁文件。

```
php composer.phar update
```

如果只想安装或更新一个依赖，你可以白名单它们：

```
php composer.phar update monolog/monolog [...]
```

> **注意：** 对于库，并不一定建议提交锁文件 请参考：库的锁文件.

Packagist

packagist 是 Composer 的主要资源库。 一个 Composer 的库基本上是一个包的源：记录了可以得到包的地方。Packagist 的目标是成为大家使用库资源的中央存储平台。这意味着你可以 require 那里的任何包。

当你访问 packagist website (packagist.org)，你可以浏览和搜索资源包。

任何支持 Composer 的开源项目应该发布自己的包在 packagist 上。虽然并不一定要发布在 packagist 上来使用 Composer，但它使我们的编程生活更加轻松。

自动加载

对于库的自动加载信息，Composer 生成了一个 vendor/autoload.php 文件。你可以简单的引入这个文件，你会得到一个免费的自动加载支持。

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

这使得你可以很容易的使用第三方代码。例如：如果你的项目依赖 monolog，你就可以像这样开始使用这个类库，并且他们将被自动加载。

```
$log = new Monolog\Logger('name');
$log->pushHandler(new Monolog\Handler\StreamHandler('app.log', Monolog\Logger::WAR
NING));

$log->addWarning('Foo');
```

你可以在 composer.json 的 autoload 字段中增加自己的 autoloader。

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

Composer 将注册一个 PSR-4 autoloader 到 `Acme` 命名空间。

你可以定义一个从命名空间到目录的映射。此时 `src` 会在你项目的根目录，与 `vendor` 文件夹同级。例如 `src/Foo.php` 文件应该包含 `Acme\Foo` 类。

添加 `autoload` 字段后，你应该再次运行 `install` 命令来生成 `vendor/autoload.php` 文件。

引用这个文件也将返回 autoloader 的实例，你可以将包含调用的返回值存储在变量中，并添加更多的命名空间。这对于在一个测试套件中自动加载类文件是非常有用的，例如。

```
$loader = require 'vendor/autoload.php';
$loader->add('Acme\\Test\\', __DIR__);
```

除了 PSR-4 自动加载，classmap 也是支持的。这允许类被自动加载，即使不符合 PSR-0 规范。详细请查看 自动加载-参考。

> **注意：** Composer 提供了自己的 autoloader。如果你不想使用它，你可以仅仅引入 `vendor/composer/autoload_*.php` 文件，它返回一个关联数组，你可以通过这个关联数组配置自己的 autoloader。

库（资源包）

本章将告诉你如何通过 Composer 来安装你的库。

每一个项目都是一个包

只要你有一个 `composer.json` 文件在目录中，那么整个目录就是一个包。当你添加一个 `require` 到项目中，你就是在创建一个依赖于其它库的包。你的项目和库之间唯一的区别是，你的项目是一个没有名字的包。

为了使它成为一个可安装的包，你需要给它一个名称。你可以通过 `composer.json` 中的 `name` 来定义：

```
{
    "name": "acme/hello-world",
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

在这种情况下项目的名称为 `acme/hello-world`，其中 `acme` 是供应商的名称。供应商的名称是必须填写的。

> **注意：** 如果你不知道拿什么作为供应商的名称， 那么使用你 github 上的用户名通常是不错的选择。虽然包名不区分大小写，但惯例是使用小写字母，并用连字符作为单词的分隔。

平台软件包

Composer 将那些已经安装在系统上，但并不是由 Composer 安装的包视为一个虚拟的平台软件包。这包括PHP本身，PHP扩展和一些系统库。

- `php` 表示用户的 PHP 版本要求，你可以对其做出限制。例如 `>=5.4.0`。如果需要64位版本的 PHP，你可以使用 `php-64bit` 进行限制。
- `hhvm` 代表的是 HHVM（也就是 HipHop Virtual Machine）运行环境的版本，并且允许你设置一个版本限制，例如，'>=2.3.3'。
- `ext-` 可以帮你指定需要的 PHP 扩展（包括核心扩展）。通常 PHP 拓展的版本可以是不一致的，将它们的版本约束为 `*` 是一个不错的主意。一个 PHP 扩展包的例子：包名可以写成 `ext-gd`。
- `lib-` 允许对 PHP 库的版本进行限制。

以下是可供使用的名称：`curl`、`iconv`、`icu`、`libxml`、`openssl`、`pcre`、`uuid`、`xsl`。

你可以使用 `composer show --platform` 命令来获取可用的平台软件包的列表。

指明版本

你需要一些方法来指明自己开发的包的版本，当你在 Packagist 上发布自己的包，它能够从 VCS (git, svn, hg) 的信息推断出包的版本，因此你不必手动指明版本号，并且也不建议这样做。请查看 标签 和 分支 来了解版本号是如何被提取的。

如果你想要手动创建并且真的要明确指定它，你只需要添加一个 `version` 字段：

```
{
    "version": "1.0.0"
}
```

> **注意：** 你应该尽量避免手动设置版本号，因为标签的值必须与标签名相匹配。

标签

对于每一个看起来像版本号的标签，都会相应的创建一个包的版本。它应该符合 'X.Y.Z' 或者 'vX.Y.Z' 的形式，`-patch`、`-alpha`、`-beta` 或 `-RC` 这些后缀是可选的。在后缀之后也可以再跟上一个数字。

下面是有效的标签名称的几个例子：

- 1.0.0
- v1.0.0
- 1.10.5-RC1
- v4.4.4beta2
- v2.0.0-alpha
- v2.0.4-p1

> **注意：** 即使你的标签带有前缀 `v`，由于在需要 `require` 一个版本的约束时是不允许这种前缀的，因此 `v` 将被省略（例如标签 `V1.0.0` 将创建 `1.0.0` 版本）。

分支

对于每一个分支，都会相应的创建一个包的开发版本。如果分支名看起来像一个版本号，那么将创建一个如同 `{分支名}-dev` 的包版本号。例如一个分支 `2.0` 将产生一个 `2.0.x-dev` 包版本（加入了 `.x` 是出于技术的原因，以确保它被识别为一个分支，而 `2.0.x` 的分支名称也是允许的，它同样会被转换为 `2.0.x-dev`）。如果分支名看起来不像一个版本号，它将会创建 `dev-{分支名}` 形式的版本号。例如 `master` 将产生一个 `dev-master` 的版本号。

下面是版本分支名称的一些示例：

- 1.x
- 1.0 (equals 1.0.x)
- 1.1.x

> **注意：** 当你安装一个新的版本时，将会自动从它 `source` 中拉取。 详细请查看 `install` 命令。

别名

它表示一个包版本的别名。例如，你可以为 `dev-master` 设置别名 `1.0.x-dev` ，这样就可以通过 require `1.0.x-dev` 来得到 `dev-master` 版本的包。

详细请查看 “别名” 。

锁文件

如果你愿意，可以在你的项目中提交 `composer.lock` 文件。他将帮助你的团队始终针对同一个依赖版本进行测试。任何时候，这个锁文件都只对于你的项目产生影响。

如果你不想提交锁文件，并且你正在使用 Git，那么请将它添加到 `.gitignore` 文件中。

发布到 VCS（线上版本控制系统）

一旦你有一个包含 `composer.json` 文件的库存储在线上版本控制系统（例如：Git），你的库就可以被 Composer 所安装。在这个例子中，我们将 `acme/hello-world` 库发布在 GitHub 上的 `github.com/username/hello-world` 中。

现在测试这个 `acme/hello-world` 包，我们在本地创建一个新的项目。我们将它命名为 `acme/blog` 。此博客将依赖 `acme/hello-world` ，而后者又依赖 `monolog/monolog` 。我们可以在某处创建一个新的 `blog` 文件夹来完成它，并且需要包含 `composer.json` 文件：

```
{
    "name": "acme/blog",
    "require": {
        "acme/hello-world": "dev-master"
    }
}
```

在这个例子中 `name` 不是必须的，因为我们并不想将它发布为一个库。在这里为 `composer.json` 文件添加描述。

现在我们需要告诉我们的应用，在哪里可以找到 `hello-world` 的依赖。为此我们需要在 `composer.json`

中添加 repositories 来源申明：

```
{
    "name": "acme/blog",
    "repositories": [
        {
            "type": "vcs",
            "url": "https://github.com/username/hello-world"
        }
    ],
    "require": {
        "acme/hello-world": "dev-master"
    }
}
```

更多关于包的来源是如何工作的，以及还有什么其他的类型可供选择，请查看资源库。

这就是全部了。你现在可以使用 Composer 的 install 命令来安装你的依赖包了！

小结： 任何含有 composer.json 的 GIT、SVN、HG 存储库，都可以通过 require 字段指定“包来源”和“声明依赖”来添加到你的项目中。

发布到 packagist

好的，你现在可以发布你的包了，但你不会希望你的用户每次都这样繁琐的指定包的来源。

你可能注意到了另一件事，我们并没有指定 monolog/monolog 的来源。它是怎么工作的？答案是 packagist。

Packagist 是 Composer 主要的一个包信息存储库，它默认是启用的。任何在 packagist 上发布的包都可以直接被 Composer 使用。就像 monolog 它被 发布在 packagist 上，我们可以直接使用它，而不必指定任何额外的来源信息。

如果我们想与世界分享我们的 hello-world，我们最好将它发布到 packagist 上。这样做是很容易的。

你只需要点击那个大大的 "Submit Package" 按钮并注册。接着提交你库的来源地址，此时 packagist 就开始了抓取。一旦完成，你的包将可以提供给任何人使用。

命令行

你已经学会了如何使用命令行界面做一些事情。本章将向你介绍所有可用的命令。

为了从命令行获得帮助信息，请运行 `composer` 或者 `composer list` 命令，然后结合 `--help` 命令来获得更多的帮助信息。

全局参数

下列参数可与每一个命令结合使用：

- **--verbose (-v):** 增加反馈信息的详细度。
    - -v 表示正常输出。
    - -vv 表示更详细的输出。
    - -vvv 则是为了 debug。
- **--help (-h):** 显示帮助信息。
- **--quiet (-q):** 禁止输出任何信息。
- **--no-interaction (-n):** 不要询问任何交互问题。
- **--working-dir (-d):** 如果指定的话，使用给定的目录作为工作目录。
- **--profile:** 显示时间和内存使用信息。
- **--ansi:** 强制 ANSI 输出。
- **--no-ansi:** 关闭 ANSI 输出。
- **--version (-V):** 显示当前应用程序的版本信息。

进程退出代码

- **0:** 正常
- **1:** 通用/未知错误
- **2:** 依赖关系处理错误

初始化 `init`

在 “库” 那一章我们看到了如何手动创建 `composer.json` 文件。实际上还有一个 `init` 命令可以更容易的做到这一点。

当您运行该命令，它会以交互方式要求您填写一些信息，同时聪明的使用一些默认值。

```
php composer.phar init
```

初始化-参数

- **--name:** 包的名称。
- **--description:** 包的描述。
- **--author:** 包的作者。
- **--homepage:** 包的主页。
- **--require:** 需要依赖的其它包，必须要有一个版本约束。并且应该遵循 `foo/bar:1.0.0` 这样的格式。
- **--require-dev:** 开发版的依赖包，内容格式与 **--require** 相同。
- **--stability (-s):** `minimum-stability` 字段的值。

安装 install

`install` 命令从当前目录读取 `composer.json` 文件，处理了依赖关系，并把其安装到 `vendor` 目录下。

```
php composer.phar install
```

如果当前目录下存在 `composer.lock` 文件，它会从此文件读取依赖版本，而不是根据 `composer.json` 文件去获取依赖。这确保了该库的每个使用者都能得到相同的依赖版本。

如果没有 `composer.lock` 文件，composer 将在处理完依赖关系后创建它。

安装-参数

- **--prefer-source:** 下载包的方式有两种： `source` 和 `dist`。对于稳定版本 composer 将默认使用 `dist` 方式。而 `source` 表示版本控制源。如果 `--prefer-source` 是被启用的，composer 将从 `source` 安装（如果有的话）。如果想要使用一个 bugfix 到你的项目，这是非常有用的。并且可以直接从本地的版本库直接获取依赖关系。
- **--prefer-dist:** 与 `--prefer-source` 相反，composer 将尽可能的从 `dist` 获取，这将大幅度的加快在 build servers 上的安装。这也是一个回避 git 问题的途径，如果你不清楚如何正确的设置。
- **--dry-run:** 如果你只是想演示而并非实际安装一个包，你可以运行 `--dry-run` 命令，它将模拟安装并显示将会发生什么。
- **--dev:** 安装 `require-dev` 字段中列出的包（这是一个默认值）。
- **--no-dev:** 跳过 `require-dev` 字段中列出的包。
- **--no-scripts:** 跳过 `composer.json` 文件中定义的脚本。
- **--no-plugins:** 关闭 plugins。
- **--no-progress:** 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- **--optimize-autoloader (-o):** 转换 PSR-0/4 autoloading 到 classmap 可以获得更快的加载支持。特

别是在生产环境下建议这么做，但由于运行需要一些时间，因此并没有作为默认值。

更新 update

为了获取依赖的最新版本，并且升级 composer.lock 文件，你应该使用 update 命令。

```
php composer.phar update
```

这将解决项目的所有依赖，并将确切的版本号写入 composer.lock。

如果你只是想更新几个包，你可以像这样分别列出它们：

```
php composer.phar update vendor/package vendor/package2
```

你还可以使用通配符进行批量更新：

```
php composer.phar update vendor/*
```

更新-参数

- **--prefer-source:** 当有可用的包时，从 source 安装。
- **--prefer-dist:** 当有可用的包时，从 dist 安装。
- **--dry-run:** 模拟命令，并没有做实际的操作。
- **--dev:** 安装 require-dev 字段中列出的包（这是一个默认值）。
- **--no-dev:** 跳过 require-dev 字段中列出的包。
- **--no-scripts:** 跳过 composer.json 文件中定义的脚本。
- **--no-plugins:** 关闭 plugins。
- **--no-progress:** 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- **--optimize-autoloader (-o):** 转换 PSR-0/4 autoloading 到 classmap 可以获得更快的加载支持。特别是在生产环境下建议这么做，但由于运行需要一些时间，因此并没有作为默认值。
- **--lock:** 仅更新 lock 文件的 hash，取消有关 lock 文件过时的警告。
- **--with-dependencies** 同时更新白名单内包的依赖关系，这将进行递归更新。

申明依赖 require

require 命令增加新的依赖包到当前目录的 composer.json 文件中。

```
php composer.phar require
```

在添加或改变依赖时， 修改后的依赖关系将被安装或者更新。

如果你不希望通过交互来指定依赖包，你可以在这条令中直接指明依赖包。

```
php composer.phar require vendor/package:2.* vendor/package2:dev-master
```

申明依赖-参数

- **--prefer-source:** 当有可用的包时，从 `source` 安装。
- **--prefer-dist:** 当有可用的包时，从 `dist` 安装。
- **--dev:** 安装 `require-dev` 字段中列出的包。
- **--no-update:** 禁用依赖关系的自动更新。
- **--no-progress:** 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- **--update-with-dependencies** 一并更新新装包的依赖。

全局执行 `global`

`global` 命令允许你在 COMPOSER_HOME 目录下执行其它命令，像 `install`、`require` 或 `update`。

并且如果你将 `$COMPOSER_HOME/vendor/bin` 加入到了 `$PATH` 环境变量中，你就可以用它在命令行中安装全局应用，下面是一个例子：

```
php composer.phar global require fabpot/php-cs-fixer:dev-master
```

现在 `php-cs-fixer` 就可以在全局范围使用了（假设你已经设置了你的 PATH）。如果稍后你想更新它，你只需要运行 `global update`：

```
php composer.phar global update
```

搜索 `search`

`search` 命令允许你为当前项目搜索依赖包，通常它只搜索 packagist.org 上的包，你可以简单的输入你的搜索条件。

```
php composer.phar search monolog
```

您也可以通过传递多个参数来进行多条件搜索。

搜索-参数

- **--only-name (-N):** 仅针对指定的名称搜索（完全匹配）。

展示 `show`

列出所有可用的软件包，你可以使用 `show` 命令。

```
php composer.phar show
```

如果你想看到一个包的详细信息，你可以输入一个包名称。

```
php composer.phar show monolog/monolog

name     : monolog/monolog
versions : master-dev, 1.0.2, 1.0.1, 1.0.0, 1.0.0-RC1
type     : library
names    : monolog/monolog
source   : [git] http://github.com/Seldaek/monolog.git 3d4e60d0cbc4b888fe5ad223d77964428b1978da
dist     : [zip] http://github.com/Seldaek/monolog/zipball/3d4e60d0cbc4b888fe5ad223d77964428b1978da 3d4e60d0cbc4b888fe5ad223d77964428b1978da
license  : MIT

autoload
psr-0
Monolog : src/

requires
php >=5.3.0
```

你甚至可以输入一个软件包的版本号，来显示该版本的详细信息。

```
php composer.phar show monolog/monolog 1.0.2
```

展示-参数

- **--installed (-i):** 列出已安装的依赖包。
- **--platform (-p):** 仅列出平台软件包（PHP 与它的扩展）。
- **--self (-s):** 仅列出当前项目信息。

依赖性检测 depends

`depends` 命令可以查出已安装在你项目中的某个包，是否正在被其它的包所依赖，并列出他们。

```
php composer.phar depends --link-type=require monolog/monolog

nrk/monolog-fluent
poc/poc
propel/propel
symfony/monolog-bridge
symfony/symfony
```

依赖性检测-参数

- **--link-type:** 检测的类型，默认为 `require` 也可以是 `require-dev`。

有效性检测 validate

在提交 `composer.json` 文件，和创建 tag 前，你应该始终运行 `validate` 命令。它将检测你的 `composer.json` 文件是否是有效的

```
php composer.phar validate
```

有效性检测参数

- **--no-check-all:** Composer 是否进行完整的校验。

依赖包状态检测 status

如果你经常修改依赖包里的代码，并且它们是从 source（自定义源）进行安装的，那么 `status` 命令允许你进行检查，如果你有任何本地的更改它将会给予提示。

```
php composer.phar status
```

你可以使用 `--verbose` 系列参数（-v|vv|vvv）来获取更详细的详细：

```
php composer.phar status -v

You have changes in the following dependencies:
vendor/seld/jsonlint:
    M README.mdown
```

自我更新 self-update

将 Composer 自身升级到最新版本，只需要运行 `self-update` 命令。它将替换你的 `composer.phar` 文件到最新版本。

```
php composer.phar self-update
```

如果你想要升级到一个特定的版本，可以这样简单的指定它：

```
php composer.phar self-update 1.0.0-alpha7
```

如果你已经为整个系统安装 Composer（参见 全局安装），你可能需要在 root 权限下运行它：

```
sudo composer self-update
```

自我更新-参数

- **--rollback (-r):** 回滚到你已经安装的最后一个版本。
- **--clean-backups:** 在更新过程中删除旧的备份，这使得更新过后的当前版本是唯一可用的备份。

更改配置 config

config 命令允许你编辑 Composer 的一些基本设置，无论是本地的 composer.json 或者全局的 config.json 文件。

```
php composer.phar config --list
```

更改配置-使用方法

config [options] [setting-key] [setting-value1] ... [setting-valueN]

setting-key 是一个配置选项的名称， setting-value1 是一个配置的值。可以使用数组作为配置的值（像 github-protocols ），多个 setting-value 是允许的。

有效的配置选项，请查看“架构”章节的 config 。

更改配置-参数

- **--global (-g):** 操作位于 $COMPOSER_HOME/config.json 的全局配置文件。如果不指定该参数，此命令将影响当前项目的 composer.json 文件，或 --file 参数所指向的文件。
- **--editor (-e):** 使用文本编辑器打开 composer.json 文件。默认情况下始终是打开当前项目的文件。当存在 --global 参数时，将会打开全局 composer.json 文件。
- **--unset:** 移除由 setting-key 指定名称的配置选项。
- **--list (-l):** 显示当前配置选项的列表。当存在 --global 参数时，将会显示全局配置选项的列表。
- **--file="..." (-f):** 在一个指定的文件上操作，而不是 composer.json。注意：不能与 --global 参数一起使用。

修改包来源

除了修改配置选项， `config` 命令还支持通过以下方法修改来源信息：

```
php composer.phar config repositories.foo vcs http://github.com/foo/bar
```

创建项目 create-project

你可以使用 Composer 从现有的包中创建一个新的项目。这相当于执行了一个 `git clone` 或 `svn checkout` 命令后将这个包的依赖安装到它自己的 vendor 目录。

此命令有几个常见的用途：

1. 你可以快速的部署你的应用。
2. 你可以检出任何资源包，并开发它的补丁。
3. 多人开发项目，可以用它来加快应用的初始化。

要创建基于 Composer 的新项目，你可以使用 "create-project" 命令。传递一个包名，它会为你创建项目的目录。你也可以在第三个参数中指定版本号，否则将获取最新的版本。

如果该目录目前不存在，则会在安装过程中自动创建。

```
php composer.phar create-project doctrine/orm path 2.2.*
```

此外，你也可以无需使用这个命令，而是通过现有的 `composer.json` 文件来启动这个项目。

默认情况下，这个命令会在 packagist.org 上查找你指定的包。

创建项目-参数

- **--repository-url:** 提供一个自定义的储存库来搜索包，这将被用来代替 packagist.org。可以是一个指向 `composer` 资源库的 HTTP URL，或者是指向某个 `packages.json` 文件的本地路径。
- **--stability (-s):** 资源包的最低稳定版本，默认为 `stable`。
- **--prefer-source:** 当有可用的包时，从 `source` 安装。
- **--prefer-dist:** 当有可用的包时，从 `dist` 安装。
- **--dev:** 安装 `require-dev` 字段中列出的包。
- **--no-install:** 禁止安装包的依赖。
- **--no-plugins:** 禁用 plugins。
- **--no-scripts:** 禁止在根资源包中定义的脚本执行。
- **--no-progress:** 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- **--keep-vcs:** 创建时跳过缺失的 VCS 。如果你在非交互模式下运行创建命令，这将是非常有用的。

打印自动加载索引 dump-autoload

某些情况下你需要更新 autoloader，例如在你的包中加入了一个新的类。你可以使用 dump-autoload 来完成，而不必执行 install 或 update 命令。

此外，它可以打印一个优化过的，符合 PSR-0/4 规范的类的索引，这也是出于对性能的可考虑。在大型的应用中会有许多类文件，而 autoloader 会占用每个请求的很大一部分时间，使用 classmaps 或许在开发时不太方便，但它在保证性能的前提下，仍然可以获得 PSR-0/4 规范带来的便利。

打印自动加载索引-参数

- **--optimize (-o):** 转换 PSR-0/4 autoloading 到 classmap 获得更快的载入速度。这特别适用于生产环境，但可能需要一些时间来运行，因此它目前不是默认设置。
- **--no-dev:** 禁用 autoload-dev 规则。

查看许可协议 licenses

列出已安装的每个包的名称、版本、许可协议。可以使用 --format=json 参数来获取 JSON 格式的输出。

执行脚本 run-script

你可以运行此命令来手动执行 脚本，只需要指定脚本的名称，可选的 --no-dev 参数允许你禁用开发者模式。

诊断 diagnose

如果你觉得发现了一个 bug 或是程序行为变得怪异，你可能需要运行 diagnose 命令，来帮助你检测一些常见的问题。

```
php composer.phar diagnose
```

归档 archive

此命令用来对指定包的指定版本进行 zip/tar 归档。它也可以用来归档你的整个项目，不包括 excluded/ignored（排除/忽略）的文件。

```
php composer.phar archive vendor/package 2.0.21 --format=zip
```

归档-参数

- **--format (-f):** 指定归档格式：tar 或 zip（默认为 tar）。

- **--dir:** 指定归档存放的目录（默认为当前目录）。

获取帮助信息 help

使用 help 可以获取指定命令的帮助信息。

```
php composer.phar help install
```

环境变量

你可以设置一些环境变量来覆盖默认的配置。建议尽可能的在 composer.json 的 config 字段中设置这些值，而不是通过命令行设置环境变量。值得注意的是环境变量中的值，将始终优先于 composer.json 中所指定的值。

COMPOSER

环境变量 COMPOSER 可以为 composer.json 文件指定其它的文件名。

例如：

```
COMPOSER=composer-other.json php composer.phar install
```

COMPOSER_ROOT_VERSION

通过设置这个环境变量，你可以指定 root 包的版本，如果程序不能从 VCS 上猜测出版本号，并且未在 composer.json 文件中申明。

COMPOSER_VENDOR_DIR

通过设置这个环境变量，你可以指定 composer 将依赖安装在 vendor 以外的其它目录中。

COMPOSER_BIN_DIR

通过设置这个环境变量，你可以指定 bin（Vendor Binaries）目录到 vendor/bin 以外的其它目录。

http_proxy or HTTP_PROXY

如果你是通过 HTTP 代理来使用 Composer，你可以使用 http_proxy 或 HTTP_PROXY 环境变量。只要简单的将它设置为代理服务器的 URL。许多操作系统已经为你的服务设置了此变量。

建议使用 http_proxy（小写）或者两者都进行定义。因为某些工具，像 git 或 curl 将使用 http_proxy 小写的版本。另外，你还可以使用 git config --global http.proxy 来单独设置 git 的代理。

no_proxy

如果你是使用代理服务器，并且想要对某些域名禁用代理，就可以使用 no_proxy 环境变量。只需要输入

一个逗号相隔的域名 *排除* 列表。

此环境变量接受域名、IP 以及 CIDR地址块。你可以将它限制到一个端口（例如：`:80`）。你还可以把它设置为 `*` 来忽略所有的 HTTP 代理请求。

HTTP_PROXY_REQUEST_FULLURI

如果你使用了 HTTP 代理，但它不支持 `request_fulluri` 标签，那么你应该设置这个环境变量为 `false` 或 `0`，来防止 composer 从 `request_fulluri` 读取配置。

HTTPS_PROXY_REQUEST_FULLURI

如果你使用了 HTTPS 代理，但它不支持 `request_fulluri` 标签，那么你应该设置这个环境变量为 `false` 或 `0`，来防止 composer 从 `request_fulluri` 读取配置。

COMPOSER_HOME

`COMPOSER_HOME` 环境变量允许你改变 Composer 的主目录。这是一个隐藏的、所有项目共享的全局目录（对本机的所有用户都可用）。

它在各个系统上的默认值分别为：

- *nix `/home//.composer`。
- OSX `/Users//.composer`。
- Windows `C:\Users\\AppData\Roaming\Composer`。

COMPOSER_HOME/config.json

你可以在 `COMPOSER_HOME` 目录中放置一个 `config.json` 文件。在你执行 `install` 和 `update` 命令时，Composer 会将它与你项目中的 `composer.json` 文件进行合并。

该文件允许你为用户的项目设置 配置信息 和 资源库。

若 *全局* 和 *项目* 存在相同配置项，那么项目中的 `composer.json` 文件拥有更高的优先级。

COMPOSER_CACHE_DIR

`COMPOSER_CACHE_DIR` 环境变量允许你设置 Composer 的缓存目录，这也可以通过 `cache-dir` 进行配置。

它在各个系统上的默认值分别为：

- *nix and OSX `$COMPOSER_HOME/cache`。
- Windows `C:\Users\\AppData\Local\Composer` 或 `%LOCALAPPDATA%/Composer`。

COMPOSER_PROCESS_TIMEOUT

这个环境变量控制着 Composer 执行命令的等待时间（例如：git 命令）。默认值为300秒（5分钟）。

COMPOSER_DISCARD_CHANGES

这个环境变量控制着 discard-changes config option。

COMPOSER_NO_INTERACTION

如果设置为1，这个环境变量将使 Composer 在执行每一个命令时都放弃交互，相当于对所有命令都使用了 `--no-interaction`。可以在搭建 *虚拟机/持续集成服务器* 时这样设置。

架构

本章将解释所有在 `composer.json` 中可用的字段。

JSON schema

我们有一个 JSON schema 格式化文档，它也可以被用来验证你的 `composer.json` 文件。事实上，它已经被 `validate` 命令所使用。 你可以在这里找到它： `res/composer-schema.json`.

Root 包

“root 包”是指由 `composer.json` 定义的在你项目根目录的包。这是 `composer.json` 定义你项目所需的主要条件。（简单的说，你自己的项目就是一个 root 包）

某些字段仅适用于“root 包”上下文。 `config` 字段就是其中一个例子。只有“root 包”可以定义。在依赖包中定义的 `config` 字段将被忽略，这使得 `config` 字段只有“root 包”可用（ `root-only` ）。

如果你克隆了其中的一个依赖包，直接在其上开始工作，那么它就变成了“root 包”。与作为他人的依赖包时使用相同的 `composer.json` 文件，但上下文发生了变化。

> **注意：** 一个资源包是不是“root 包”，取决于它的上下文。 例：如果你的项目依赖 `monolog` 库，那么你的项目就是“root 包”。 但是，如果你从 GitHub 上克隆了 `monolog` 为它修复 bug， 那么此时 `monolog` 就是“root 包”。

属性

包名 `name`

包的名称，它包括供应商名称和项目名称，使用 `/` 分隔。

例：

- monolog/monolog
- igorw/event-source

对于需要发布的包（库），这是必须填写的。

描述 `description`

一个包的简短描述。通常这个最长只有一行。

对于需要发布的包（库），这是必须填写的。

版本 version

version 不是必须的，并且建议忽略（见下文）。

它应该符合 'X.Y.Z' 或者 'vX.Y.Z' 的形式， -dev 、 -patch 、 -alpha 、 -beta 或 -RC 这些后缀是可选的。在后缀之后也可以再跟上一个数字。

例：

- 1.0.0
- 1.0.2
- 1.1.0
- 0.2.5
- 1.0.0-dev
- 1.0.0-alpha3
- 1.0.0-beta2
- 1.0.0-RC5

通常，我们能够从 VCS (git, svn, hg) 的信息推断出包的版本号，在这种情况下，我们建议忽略 version 。

> **注意：** Packagist 使用 VCS 仓库， 因此 version 定义的版本号必须是真实准确的。 自己手动指定的 version ，最终有可能在某个时候因为人为错误造成问题。

安装类型 type

包的安装类型，默认为 library 。

包的安装类型，用来定义安装逻辑。如果你有一个包需要一个特殊的逻辑，你可以设定一个自定义的类型。这可以是一个 symfony-bundle ，一个 wordpress-plugin 或者一个 typo3-module 。这些类型都将是具体到某一个项目，而对应的项目将要提供一种能够安装该类型包的安装程序。

composer 原生支持以下4种类型：

- **library:** 这是默认类型，它会简单的将文件复制到 vendor 目录。
- **project:** 这表示当前包是一个项目，而不是一个库。例：框架应用程序 Symfony standard edition ，内容管理系统 SilverStripe installer 或者完全成熟的分布式应用程序。使用 IDE 创建一个新的工作区时，这可以为其提供项目列表的初始化。
- **metapackage:** 当一个空的包，包含依赖并且需要触发依赖的安装，这将不会对系统写入额外的文件。因此这种安装类型并不需要一个 dist 或 source。
- **composer-plugin:** 一个安装类型为 composer-plugin 的包，它有一个自定义安装类型，可以为其它包提供一个 installler。详细请查看 自定义安装类型。

仅在你需要一个自定义的安装逻辑时才使用它。建议忽略这个属性，采用默认的 `library`。

关键字 keywords

该包相关的关键词的数组。这些可用于搜索和过滤。

实例：

- logging
- events
- database
- redis
- templating

可选。

项目主页 homepage

该项目网站的 URL 地址。

可选。

版本发布时间 time

版本发布时间。

必须符合 `YYYY-MM-DD` 或 `YYYY-MM-DD HH:MM:SS` 格式。

可选。

许可协议 license

包的许可协议，它可以是一个字符串或者字符串数组。

最常见的许可协议的推荐写法（按字母排序）：

- Apache-2.0
- BSD-2-Clause
- BSD-3-Clause
- BSD-4-Clause
- GPL-2.0
- GPL-2.0+
- GPL-3.0
- GPL-3.0+
- LGPL-2.1
- LGPL-2.1+

- LGPL-3.0
- LGPL-3.0+
- MIT

可选，但强烈建议提供此内容。更多许可协议的标识符请参见 SPDX Open Source License Registry。

对于闭源软件，你必须使用 `"proprietary"` 协议标识符。

一个例：

```
{
    "license": "MIT"
}
```

对于一个包，当允许在多个许可协议间进行选择时（"disjunctive license"），这些协议标识符可以被指定为数组。

多协议的一个例：

```
{
    "license": [
      "LGPL-2.1",
      "GPL-3.0+"
    ]
}
```

另外它们也可以由 "or" 分隔，并写在括号中：

```
{
    "license": "(LGPL-2.1 or GPL-3.0+)"
}
```

同样，当有多个许可协议需要结合使用时（"conjunctive license"），它们应该被 "and" 分隔，并写在括号中。

作者 `authors`

包的作者。这是一个对象数组。

这个对象必须包含以下属性：

- **name:** 作者的姓名，通常使用真名。
- **email:** 作者的 email 地址。
- **homepage:** 作者主页的 URL 地址。

- **role:** 该作者在此项目中担任的角色（例：开发人员 或 翻译）。

一个实例：

```
{
    "authors": [
        {
            "name": "Nils Adermann",
            "email": "naderman@naderman.de",
            "homepage": "http://www.naderman.de",
            "role": "Developer"
        },
        {
            "name": "Jordi Boggiano",
            "email": "j.boggiano@seld.be",
            "homepage": "http://seld.be",
            "role": "Developer"
        }
    ]
}
```

可选，但强烈建议提供此内容。

支持 support

获取项目支持的向相关信息对象。

这个对象必须包含以下属性：

- **email:** 项目支持 email 地址。
- **issues:** 跟踪问题的 URL 地址。
- **forum:** 论坛地址。
- **wiki:** Wiki 地址。
- **irc:** IRC 聊天频道地址，类似于 irc://server/channel。
- **source:** 网址浏览或下载源。

一个实例：

```
{
    "support": {
        "email": "support@example.org",
        "irc": "irc://irc.freenode.org/composer"
    }
}
```

可选。

Package links

下面提到的所有对象，都应该是 包名 到 版本 的映射对象。

实例：

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

所有的这些都是可选的。

require 和 require-dev 还支持稳定性标签（@，仅针对“root 包”）。这允许你在 minimum-stability 设定的范围外做进一步的限制或扩展。例：如果你想允许依赖一个不稳定的包，你可以在一个包的版本约束后使用它，或者是一个空的版本约束内使用它。

实例：

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

如果你的依赖之一，有对另一个不稳定包的依赖，你最好在 require 中显示的定义它，并带上足够详细的稳定性标识。

实例：

```
{
    "require": {
        "doctrine/doctrine-fixtures-bundle": "dev-master",
        "doctrine/data-fixtures": "@dev"
    }
}
```

require 和 require-dev 还支持对 dev（开发）版本的明确引用（即：版本控制系统中的提交编号 commit），以确保它们被锁定到一个给定的状态，即使你运行了更新命令。你只需要明确一个开发版本号，并带上诸如 # 的标识。

实例：

```
{
    "require": {
        "monolog/monolog": "dev-master#2eb0c0978d290a1c45346a1955188929cb4e5db7",
        "acme/foo": "1.0.x-dev#abc123"
    }
}
```

> **注意：** 虽然这有时很方便，但不应该长期在你的包中使用，因为它有一个技术上的限制。composer.json 将仍然在哈希值之前指定的分支名称读取元数据， 正因为如此，在某些情况下，它不会是一个实用的解决方法， 如果可能，你应该总是尝试切换到拥有标签的版本。

它也可以应用于行内别名，这样它将匹配一个约束，否则不会。更多信息请参考 别名。

require

必须的软件包列表，除非这些依赖被满足，否则不会完成安装。

require-dev (root-only)

这个列表是为开发或测试等目的，额外列出的依赖。 “root 包” 的 require-dev 默认是会被安装的。然而 `install` 或 `update` 支持使用 `--no-dev` 参数来跳过 `require-dev` 字段中列出的包。

conflict

此列表中的包与当前包的这个版本冲突。它们将不允许同时被安装。

请注意，在 `conflict` 中指定类似于 `< 1.1` 的版本范围时，这表示它与小于1.0 *并且* 同时大等于1.1的版本冲突，这很可能不是你想要的。在这种情况下你可能想要表达的是 `>= 1.1` 。

replace

这个列表中的包将被当前包取代。这使你可以 fork 一个包，以不同的名称和版本号发布，同时要求依赖于原包的其它包，在这之后依赖于你 fork 的这个包，因为它取代了原来的包。

这对于创建一个内部包含子包的主包也非常的有用。例如 symfony/symfony 这个主包，包含了所有 Symfony 的组件，而这些组件又可以作为单独的包进行发布。如果你 require 了主包，那么它就会自动完成其下各个组件的任务，因为主包取代了子包。

注意，在使用上述方法取代子包时，通常你应该只对子包使用 `self.version` 这一个版本约束，以确保主包仅替换掉子包的准确版本，而不是任何其他版本。

provide

List of other packages that are provided by this package. This is mostly useful for common interfaces. A package could depend on some virtual `logger` package, any library that implements this logger interface would simply list it in `provide`.

suggest

建议安装的包，它们增强或能够与当前包良好的工作。这些只是信息，并显示在依赖包安装完成之后，给你的用户一个建议，他们可以添加更多的包。

格式如下，版本约束变成了描述信息。

实例：

```
{
    "suggest": {
        "monolog/monolog": "Allows more advanced logging of the application flow"
    }
}
```

autoload

PHP autoloader 的自动加载映射。

Currently `PSR-0` autoloading, `PSR-4` autoloading, `classmap` generation and `files` includes are supported. PSR-4 is the recommended way though since it offers greater ease of use (no need to regenerate the autoloader when you add classes).

PSR-4

Under the `psr-4` key you define a mapping from namespaces to paths, relative to the package root. When autoloading a class like `Foo\\Bar\\Baz` a namespace prefix `Foo\\` pointing to a directory `src/` means that the autoloader will look for a file named `src/Bar/Baz.php` and include it if present. Note that as opposed to the older PSR-0 style, the prefix (`Foo\\`) is **not** present in the file path.

Namespace prefixes must end in `\\` to avoid conflicts between similar prefixes. For example `Foo` would match classes in the `FooBar` namespace so the trailing backslashes solve the problem: `Foo\\` and `FooBar\\` are distinct.

The PSR-4 references are all combined, during install/update, into a single key => value array which may be found in the generated file `vendor/composer/autoload_psr4.php`.

Example:

```
{
    "autoload": {
        "psr-4": {
            "Monolog\\": "src/",
            "Vendor\\Namespace\\": ""
        }
    }
}
```

If you need to search for a same prefix in multiple directories, you can specify them as an array as such:

```
{
    "autoload": {
        "psr-4": { "Monolog\\": ["src/", "lib/"] }
    }
}
```

If you want to have a fallback directory where any namespace will be looked for, you can use an empty prefix like:

```
{
    "autoload": {
        "psr-4": { "": "src/" }
    }
}
```

PSR-0

在 `psr-0` key 下你定义了一个命名空间到实际路径的映射（相对于包的根目录）。注意，这里同样支持 PEAR-style 方式的约定（与命名空间不同，PEAR 类库在类名上采用了下划线分隔）。

请注意，命名空间的申明应该以 `\\` 结束，以确保 autoloader 能够准确响应。例： `Foo` 将会与 `FooBar` 匹配，然而以反斜杠结束就可以解决这样的问题， `Foo\\` 和 `FooBar\\` 将会被区分开来。

在 install/update 过程中，PSR-0 引用都将被结合为一个单一的键值对数组，存储至 `vendor/composer/autoload_namespaces.php` 文件中。

实例：

```
{
    "autoload": {
        "psr-0": {
            "Monolog\\": "src/",
            "Vendor\\Namespace\\": "src/",
            "Vendor_Namespace_": "src/"
        }
    }
}
```

如果你需要搜索多个目录中一个相同的前缀，你可以将它们指定为一个数组，例：

```
{
    "autoload": {
        "psr-0": { "Monolog\\": ["src/", "lib/"] }
    }
}
```

PSR-0 方式并不仅限于申明命名空间，也可以是精确到类级别的指定。这对于只有一个类在全局命名空间的类库是非常有用的（如果 php 源文件也位于包的根目录）。例如，可以这样申明：

```
{
    "autoload": {
        "psr-0": { "UniqueGlobalClass": "" }
    }
}
```

如果你想设置一个目录作为任何命名空间的备用目录，你可以使用空的前缀，像这样：

```
{
    "autoload": {
        "psr-0": { "": "src/" }
    }
}
```

Classmap

`classmap` 引用的所有组合，都会在 install/update 过程中生成，并存储到 `vendor/composer/autoload_classmap.php` 文件中。这个 map 是经过扫描指定目录（同样支持直接精确到文件）中所有的 `.php` 和 `.inc` 文件里内置的类而得到的。

你可以用 classmap 生成支持支持自定义加载的不遵循 PSR-0/4 规范的类库。要配置它指向需要的目录，以便能够准确搜索到类文件。

实例：

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

Files

如果你想要明确的指定，在每次请求时都要载入某些文件，那么你可以使用 'files' autoloading。通常作为函数库的载入方式（而非类库）。

实例：

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

autoload-dev (root-only)

This section allows to define autoload rules for development purposes.

Classes needed to run the test suite should not be included in the main autoload rules to avoid polluting the autoloader in production and when other people use your package as a dependency.

Therefore, it is a good idea to rely on a dedicated path for your unit tests and to add it within the autoload-dev section.

Example:

```
{
    "autoload": {
        "psr-4": { "MyLibrary\\": "src/" }
    },
    "autoload-dev": {
        "psr-4": { "MyLibrary\\Tests\\": "tests/" }
    }
}
```

include-path

> **不建议**：这是目前唯一支持传统项目的做法，所有新的代码都建议使用自动加载。 这是一个过时的做法，但 Composer 将仍然保留这个功能。

一个追加到 PHP `include_path` 中的列表。

实例：

```
{
    "include-path": ["lib/"]
}
```

可选。

target-dir

> **DEPRECATED**: This is only present to support legacy PSR-0 style autoloading, and all new code should preferably use PSR-4 without target-dir and projects using PSR-0 with PHP namespaces are encouraged to migrate to PSR-4 instead.

定义当前包安装的目标文件夹。

若某个包的根目录，在它申明的命名空间之下，将不能正确的使用自动加载。而 `target-dir` 解决了这个问题。

Symfony 就是一个例子。它有一些独立的包作为组件。Yaml 组件就放在 `Symfony\Component\Yaml` 目录下，然而这个包的根目录实际上是 `Yaml`。为了使自动加载成为可能，我们需要确保它不会被安装到 `vendor/symfony/yaml`，而是安装到 `vendor/symfony/yaml/Symfony/Component/Yaml`，从而使 Symfony 定义的 autoloader 可以从 `vendor/symfony/yaml` 加载它。

要做到这一点 `autoload` 和 `target-dir` 应该定义如下：

```
{
    "autoload": {
        "psr-0": { "Symfony\\Component\\Yaml\\": "" }
    },
    "target-dir": "Symfony/Component/Yaml"
}
```

可选。

minimum-stability (root-only)

这定义了通过稳定性过滤包的默认行为。默认为 `stable`（稳定）。因此如果你依赖于一个 `dev`（开发）包，你应该明确的进行定义。

对每个包的所有版本都会进行稳定性检查，而低于 `minimum-stability` 所设定的最低稳定性的版本，将在解决依赖关系时被忽略。对于个别包的特殊稳定性要求，可以在 `require` 或 `require-dev` 中设定（请参考Package links）。

可用的稳定性标识（按字母排序）：`dev`、`alpha`、`beta`、`RC`、`stable`。

prefer-stable (root-only)

当此选项被激活时，Composer 将优先使用更稳定的包版本。

使用 `"prefer-stable": true` 来激活它。

repositories (root-only)

使用自定义的包资源库。

默认情况下 composer 只使用 packagist 作为包的资源库。通过指定资源库，你可以从其他地方获取资源包。

Repositories 并不是递归调用的，只能在“Root包”的 `composer.json` 中定义。附属包中的 `composer.json` 将被忽略。

支持以下类型的包资源库：

- **composer:** 一个 composer 类型的资源库，是一个简单的网络服务器（HTTP、FTP、SSH）上的 `packages.json` 文件，它包含一个 `composer.json` 对象的列表，有额外的 `dist` 和/或 `source` 信息。这个 `packages.json` 文件是用一个 PHP 流加载的。你可以使用 `options` 参数来设定额外的流信息。
- **vcs:** 从 git、svn 和 hg 取得资源。
- **pear:** 从 pear 获取资源。
- **package:** 如果你依赖于一个项目，它不提供任何对 composer 的支持，你就可以使用这种类型。你基本上就只需要内联一个 `composer.json` 对象。

更多相关内容，请查看 资源库。

实例：

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "http://packages.example.com"
        },
        {
            "type": "composer",
            "url": "https://packages.example.com",
            "options": {
                "ssl": {
                    "verify_peer": "true"
                }
            }
        },
        {
            "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/files/Smarty-3.1.7.zip",
                    "type": "zip"
                },
                "source": {
                    "url": "http://smarty-php.googlecode.com/svn/",
                    "type": "svn",
                    "reference": "tags/Smarty_3_1_7/distribution/"
                }
            }
        }
    ]
}
```

> **注意：** 顺序是非常重要的，当 Composer 查找资源包时，它会按照顺序进行。默认情况下 Packagist 是最后加入的，因此自定义设置将可以覆盖 Packagist 上的包。

config (root-only)

下面的这一组选项，仅用于项目。

支持以下选项：

- **process-timeout:** 默认为 `300`。处理进程结束时间，例如：git 克隆的时间。Composer 将放弃超时的任务。如果你的网络缓慢或者正在使用一个巨大的包，你可能要将这个值设置的更高一些。
- **use-include-path:** 默认为 `false`。如果为 true，Composer autoloader 还将在 PHP include path 中继续查找类文件。
- **preferred-install:** 默认为 `auto`。它的值可以是 `source`、`dist` 或 `auto`。这个选项允许你设置 Composer 的默认安装方法。
- **github-protocols:** 默认为 `["git", "https", "ssh"]`。从 github.com 克隆时使用的协议优先级清单，因此默认情况下将优先使用 git 协议进行克隆。你可以重新排列它们的次序，例如，如果你的网络有代理服务器或 git 协议的效率很低，你就可以提升 https 协议的优先级。
- **github-oauth:** 一个域名和 oauth keys 的列表。 例如：使用 `{"github.com": "oauthtoken"}` 作为此选项的值， 将使用 `oauthtoken` 来访问 github 上的私人仓库，并绕过 low IP-based rate 的 API 限制。 关联知识 关于如何获取 GitHub 的 OAuth token。
- **vendor-dir:** 默认为 `vendor`。通过设置你可以安装依赖到不同的目录。
- **bin-dir:** 默认为 `vendor/bin`。如果一个项目包含二进制文件，它们将被连接到这个目录。
- **cache-dir:** unix 下默认为 `$home/cache`，Windows 下默认为 `C:\Users\\AppData\Local\Composer`。用于存储 composer 所有的缓存文件。相关信息请查看COMPOSER_HOME。
- **cache-files-dir:** 默认为 `$cache-dir/files`。存储包 zip 存档的目录。
- **cache-repo-dir:** 默认为 `$cache-dir/repo`。存储 `composer` 类型的 VCS（`svn`、`github`、`bitbucket`）repos 目录。
- **cache-vcs-dir:** 默认为 `$cache-dir/vcs`。此目录用于存储 VCS 克隆的 `git` / `hg` 类型的元数据，并加快安装速度。
- **cache-files-ttl:** 默认为 `15552000`（6个月）。默认情况下 Composer 缓存的所有数据都将在闲置6个月后被删除，这个选项允许你来调整这个时间，你可以将其设置为0以禁用缓存。
- **cache-files-maxsize:** 默认为 `300MiB`。Composer 缓存的最大容量，超出后将优先清除旧的缓存数据，直到缓存量低于这个数值。
- **prepend-autoloader:** 默认为 `true`。如果设置为 false，composer autoloader 将不会附加到现有的自动加载机制中。这有时候用来解决与其它自动加载机制产生的冲突。
- **autoloader-suffix:** 默认为 `null`。Composer autoloader 的后缀，当设置为空时将会产生一个随机的字符串。
- **optimize-autoloader** Defaults to `false`. Always optimize when dumping the autoloader.
- **github-domains:** 默认为 `["github.com"]`。一个 github mode 下的域名列表。这是用于GitHub的企业设置。
- **notify-on-install:** 默认为 `true`。Composer 允许资源仓库定义一个用于通知的 URL，以便有人从其上安装资源包时能够得到一个反馈通知。此选项允许你禁用该行为。

- **discard-changes:** 默认为 `false`，它的值可以是 `true`、`false` 或 `stash`。这个选项允许你设置在非交互模式下，当处理失败的更新时采用的处理方式。`true` 表示永远放弃更改。`"stash"` 表示继续尝试。Use this for CI servers or deploy scripts if you tend to have modified vendors.

实例：

```
{
    "config": {
        "bin-dir": "bin"
    }
}
```

scripts (root-only)

Composer 允许你在安装过程中的各个阶段挂接脚本。

更多细节和案例请查看 脚本。

extra

任意的，供 `scripts` 使用的额外数据。.

这可以是几乎任何东西。若要从脚本事件访问处理程序，你可以这样做：

```
$extra = $event->getComposer()->getPackage()->getExtra();
```

可选。

bin

该属性用于标注一组应被视为二进制脚本的文件，他们会被软链接到（config 对象中的）`bin-dir` 属性所标注的目录，以供其他依赖包调用。

详细请查看 Vendor Binaries。

可选。

archive

这些选项在创建包存档时使用。

支持以下选项：

- **exclude:** 允许设置一个需要被排除的路径的列表。使用与 .gitignore 文件相同的语法。一个前导的（！）将会使其变成白名单而无视之前相同目录的排除设定。前导斜杠只会在项目的相对路径的开头匹配。星号为通配符。

实例：

```
{
    "archive": {
        "exclude": ["/foo/bar", "baz", "/*.test", "!/foo/bar/baz"]
    }
}
```

在这个例子中我们 include `/dir/foo/bar/file`、`/foo/bar/baz`、`/file.php`、`/foo/my.test` 但排除了 `/foo/bar/any`、`/foo/baz`、`/my.test`。

可选。

资源库

本章将解释包和库的概念，什么样的存储库是可用的，以及它们如何工作。

概述

在此之前，我们看到存在不同类型的资源库，我们需要了解一些基本概念，以理解 Composer 是如何构建于其上的。

包

Composer 是一个依赖管理工具。它在本地安装一些资源包。一个包本质上就是一个包含东西的目录。通常情况下它存储 PHP 代码，但在理论上它可以是任何东西。并且它包含一个描述，其中有一个名称和一个版本号，这个名称和版本号用于识别该包。

事实上，在 composer 内部将每一个版本都视为一个单独的包。尽管在你使用 composer 时这种区别无关紧要，但当你想改变它时，这就显得至关重要。

除了名称和版本号，还存放了有用的元数据。与安装关系最密切的是 source 信息，它申明了在哪里可以获得资源包的内容。包数据指向包内容，并有两种指向方式：dist 和 source。

Dist: dist 指向一个存档，该存档是对一个资源包的某个版本的数据进行的打包。通常是已经发行的稳定版本。

Source: source 指向一个开发中的源。这通常是一个源代码仓库，例如 git。当你想要对下载下来的资源包进行修改时，可以这样获取。

你可以使用其中任意一个，或者同时使用。这取决于其它的一些因素，比如“user-supplied 选项”和“包的稳定性”，前者将会被优先考虑。

资源库

一个资源库是一个包的来源。它是一个 packages/versions 的列表。Composer 将查看所有你定义的 repositories 以找到你项目需要的资源包。

默认情况下已经将 Packagist.org 注册到 Composer。你可以在 `composer.json` 中申明更多的资源库，把它们加入你的项目中。

资源库的定义仅可用于“root 包”，而在你依赖的包中定义的资源库将不会被加载。如果你想了解其中的原因，请阅读 FAQ entry。

Types

Composer

主资源库的类型为 `composer`。它使用一个单一的 `packages.json` 文件，包含了所有的资源包元数据。

这也是 packagist.org 所使用的资源类型。要引用一个 `composer` 资源库，只需要提供一个存放 `packages.json` 文件的 **目录路径**。比如要引用 `packagist.org` 下的 `/packages.json`，它的 URL 就应该是 `packagist.org`。而 `example.org/packages.json` 的 URL 应该是 `example.org`。

packages

唯一必须的字段是 `packages`。它的 JSON 结构如下：

```
{
    "packages": {
        "vendor/package-name": {
            "dev-master": { @composer.json },
            "1.0.x-dev": { @composer.json },
            "0.0.1": { @composer.json },
            "1.0.0": { @composer.json }
        }
    }
}
```

`@composer.json` 标记将会从此包的指定版本中读取 `composer.json` 的内容，其内至少应包含以下信息：

- name
- version
- dist or source

这是一个最简单的包定义：

```
{
    "name": "smarty/smarty",
    "version": "3.1.7",
    "dist": {
        "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
        "type": "zip"
    }
}
```

它还可以包含任何在 composer.json 架构 中介绍的字段。

notify-batch

notify-batch 字段允许你指定一个 URL，它将会在用户安装每一个包时被调用。该 URL 可以是（与其资源库相同域名的）绝对路径或者一个完整的 URL 地址。

例如使用下面的值：

```
{
    "notify-batch": "/downloads/"
}
```

对于 example.org/packages.json 包含的 monolog/monolog 包，它将会发送一个 POST 请求到 example.org/downloads/ ，使用下面的 JSON request body：

```
{
    "downloads": [
        {"name": "monolog/monolog", "version": "1.2.1.0"},
    ]
}
```

version 字段将包含标准化的版本号。

notify-batch 字段是可选的。

includes

对于较大的资源库，可以拆分 packages.json 为多个文件。 includes 字段允许你引用这些额外的文件。

实例：

```
{
    "includes": {
        "packages-2011.json": {
            "sha1": "525a85fb37edd1ad71040d429928c2c0edec9d17"
        },
        "packages-2012-01.json": {
            "sha1": "897cde726f8a3918faf27c803b336da223d400dd"
        },
        "packages-2012-02.json": {
            "sha1": "26f911ad717da26bbcac3f8f435280d13917efa5"
        }
    }
}
```

文件的 SHA-1 码允许它被缓存，仅在 hash 值改变时重新请求。

此字段是可选的。你也许并不需要它来自定义存储库。

provider-includes and providers-url

的对于非常大的资源库，像 packagist.org 使用 so-called provider 文件是首选方法。`provider-includes` 字段允许你设置一个列表，来申明这个资源库提供的包名称。在这种情况下文件的哈希算法必须使用 sha256。

`providers-url` 描述了如何在服务器上找到这些 provider 文件。它是以资源库的根目录为起点的绝对路径。

实例：

```
{
    "provider-includes": {
        "providers-a.json": {
            "sha256": "f5b4bc0b354108ef08614e569c1ed01a2782e67641744864a74e788982886f4
c"
        },
        "providers-b.json": {
            "sha256": "b38372163fac0573053536f5b8ef11b86f804ea8b016d239e706191203f6efac
"
        }
    },
    "providers-url": "/p/%package%$%hash%.json"
}
```

这些文件包含资源包的名称以及哈希值，以验证文件的完整性，例如：

```
{
    "providers": {
        "acme/foo": {
            "sha256": "38968de1305c2e17f4de33aea164515bc787c42c7e2d6e25948539a14268bb
82"
        },
        "acme/bar": {
            "sha256": "4dd24c930bd6e1103251306d6336ac813b563a220d9ca14f4743c032fb0472
33"
        }
    }
}
```

上述文件申明了 `acme/foo` 和 `acme/bar` 可以在这个资源库找到，通过加载由 `providers-url` 引用的文件，替换 `%package%` 为包名并且替换 `%hash%` 为 sha256 的值。这些文件本身只包含上文提到的 packages的定义。

这些字段是可选的。你也许并不需要它们来自定义存储库。

stream options

`packages.json` 文件是用一个 PHP 流加载的。你可以使用 `options` 参数来设定额外的流信息。你可以设置任何有效的PHP 流上下文选项。更多相关信息请查看 Context options and parameters 。

VCS

VCS 表示版本控制系统。这包括像 git、svn 或 hg 这样的版本管理系统。Composer 有一个资源类型可以从这些系统安装软件包。

从 VCS 资源库加载一个包

这里有几个用例。最常见的是维护自己 fork 的第三方库。如果你在项目中使用某些库，并且你决定改变这些库内的某些东西，你会希望你项目中使用的是你自己的修正版本。如果这个库是在 GitHub 上（这种情况经常出现），你可以简单的 fork 它并 push 你的变更到这个 fork 里。在这之后你更新项目的 `composer.json` 文件，添加你的 fork 作为一个资源库，变更版本约束来指向你的自定义分支。关于版本约束的命名约定请查看 库（资源包）。

例如，假设你 fork 了 monolog，在 `bugfix` 分支修复了一个 bug：

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "https://github.com/igorw/monolog"
        }
    ],
    "require": {
        "monolog/monolog": "dev-bugfix"
    }
}
```

当你运行 `php composer.phar update` 时，你应该得到你修改的版本，而不是 packagist.org 上的 `monolog/monolog`。

注意，你不应该对包进行重命名，除非你真的打算摆脱原来的包，并长期的使用你自己的 fork。这样 Composer 就会正确获取你的包了。如果你确定要重命名这个包，你应该在默认分支（通常是 master 分支）上操作，而不是特性分支，因为包的名字取自默认分支。

如果其它包依赖你 fork 的这个分支，可能要对它做版本号的行内别名设置，才能够准确的识别版本约束。更多相关信息请查看 别名。

使用私有资源库

完全相同的解决方案，也可以让你使用你 GitHub 和 BitBucket 上的私人代码库进行工作：

```
{
    "require": {
        "vendor/my-private-repo": "dev-master"
    },
    "repositories": [
        {
            "type": "vcs",
            "url":  "git@bitbucket.org:vendor/my-private-repo.git"
        }
    ]
}
```

唯一的要求是为一个 git 客户端安装 SSH 秘钥。

Git 的备选方案

Git 并不是 VCS 资源库唯一支持的版本管理系统。

以下几种都是被支持的：

- **Git:** git-scm.com
- **Subversion:** subversion.apache.org
- **Mercurial:** mercurial.selenic.com

为了从这些系统获取资源包，你必须安装对应的客户端，这可能是不方便的。基于这个原因，这里提供了 GitHub 和 BitBucket 的 API 的特殊支持，以便在无需安装版本控制系统的情况下获取资源包。在 VCS 资源库提供的 `dist` 中获取 zip 存档。

- **GitHub:** github.com (Git)
- **BitBucket:** bitbucket.org (Git and Mercurial)

VCS 驱动将基于 URL 自动检测版本库类型。但如果可能，你需要明确的指定一个 `git`、`svn` 或 `hg` 作为资源库类型，而不是 `vcs`。

If you set the `no-api` key to `true` on a github repository it will clone the repository as it would with any other git repository instead of using the GitHub API. But unlike using the `git` driver directly, composer will still attempt to use github's zip files.

Subversion 选项

由于 Subversion 没有原生的分支和标签的概念，Composer 假设在默认情况下该代码位于 `$url/trunk`、`$url/branches` 和 `$url/tags` 内。如果你的存储库使用了不同的布局，你可以更改这些值。例如，如果你使用大写的名称，你可以像这样配置资源库：

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "http://svn.example.org/projectA/",
            "trunk-path": "Trunk",
            "branches-path": "Branches",
            "tags-path": "Tags"
        }
    ]
}
```

如果你的存储库目录中没有任何分支或标签文件夹，你可以将 `branches-path` 或 `tags-path` 设置为 `false`。

如果是一个位于子目录的包，例如， `/trunk/foo/bar/composer.json` 和 `/tags/1.0/foo/bar/composer.json`，那么你可以让 composer 通过 `"package-path"` 选项设置的子目录进行访问，在这个例子中可以将其设置为 `"package-path": "foo/bar/"`。

PEAR

`pear` 类型资源库，使得从任何 PEAR 渠道安装资源包成为可能。Composer 将为所有此类型的包增加前缀（类似于 `pear-{渠道名称}/`）以避免冲突。而在之后使用别名时也增加前缀（如 `pear-{渠道别名}/`）。

例如使用 `pear2.php.net`：

```
{
    "repositories": [
        {
            "type": "pear",
            "url": "http://pear2.php.net"
        }
    ],
    "require": {
        "pear-pear2.php.net/PEAR2_Text_Markdown": "*",
        "pear-pear2/PEAR2_HTTP_Request": "*"
    }
}
```

在这种情况下渠道的简称（别名）是 `pear2`，因此 `PEAR2_HTTP_Request` 包的名称应该写作 `pear-pear2/PEAR2_HTTP_Request`。

> **注意：** `pear` 类型的资源库对每个 requires 都要做完整的请求，因此可能大大降低安装速度。

自定义供应商别名

通过自定义供应商名称，对 PEAR 渠道包进行别名是允许的。

例：

假设你有一个私人 PEAR 库，并希望使用 Composer 从 VCS 集成依赖。你的 PEAR 库包含以下资源包：

- `BasePackage`。
- `IntermediatePackage` 依赖于 `BasePackage`。
- `TopLevelPackage1` 和 `TopLevelPackage2` 都依赖于 `IntermediatePackage`。

如果没有一个供应商别名，Composer 将使用 PEAR 渠道名称作为包名的一部分：

- `pear-pear.foobar.repo/BasePackage`
- `pear-pear.foobar.repo/IntermediatePackage`
- `pear-pear.foobar.repo/TopLevelPackage1`
- `pear-pear.foobar.repo/TopLevelPackage2`

假设之后的某个时间，你希望将你的 PEAR 包迁移，使用 Composer 资源库和命名方案，并且采用 `foobar` 作为供应商名称。这样之前使用 PEAR 包的项目将不会看到更新的资源包，因为它们有不同的供应商名称（`foobar/IntermediatePackage` 与 `pear-pear.foobar.repo/IntermediatePackage`）。

你可以通过从一开始就为 PEAR 资源库指定 `vendor-alias` 来避免这种情况的发生，以得到一个不会过时的包名。

为了说明这一点，下面的例子会从你的 PEAR 资源库中得到 `BasePackage`、`TopLevelPackage1` 和 `TopLevelPackage2` 资源包，并从 Github 资源库中获取 `IntermediatePackage` 资源包：

```
{
    "repositories": [
        {
            "type": "git",
            "url": "https://github.com/foobar/intermediate.git"
        },
        {
            "type": "pear",
            "url": "http://pear.foobar.repo",
            "vendor-alias": "foobar"
        }
    ],
    "require": {
        "foobar/TopLevelPackage1": "*",
        "foobar/TopLevelPackage2": "*"
    }
}
```

Package

如果你想使用一个项目，它无法通过上述任何一种方式支持 composer，你仍然可以使用 `package` 类型定义资源库。

基本上，你可以定义与 `packages.json` 中 `composer` 类型资源库相同的信息，但需要为每个这样的资源包分别定义。同样，至少应该包含以下信息：`name`、`version`、（`dist` 或 `source`）。

这是一个 smarty 模板引擎的例子：

```
{
    "repositories": [
        {
            "type": "package",
            "package": {
                "name": "smarty/smarty",
                "version": "3.1.7",
                "dist": {
                    "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
                    "type": "zip"
                },
                "source": {
                    "url": "http://smarty-php.googlecode.com/svn/",
                    "type": "svn",
                    "reference": "tags/Smarty_3_1_7/distribution/"
                },
                "autoload": {
                    "classmap": ["libs/"]
                }
            }
        }
    ],
    "require": {
        "smarty/smarty": "3.1.*"
    }
}
```

通常你不需要去定义 `source`，因为你并不是真的需要它。

> **注意：** 该资源库类型存在以下限制，因此应尽可能避免使用：
>
> - Composer 将不会更新资源包，除非你修改了 `version` 字段。
> - Composer 将不会更新 commit references，因此如果你使用 `master` reference，将不得不删除该程序包以强制更新，并且将不得不面对一个不稳定的 lock 文件。

Hosting your own

尽管大部分的时间，你大概都会把资源包放在 packagist.org 上，但这里还将告诉你一些用例，以便你可以自行托管资源库。

- **Private company packages:** 如果你是一个公司的职员，对公司内部的资源包使用 composer，你可能会想让这些包保持私有的状态。
- **Separate ecosystem:** 如果你的项目有自己的生态系统，并且自己的资源包不需要被其它项目所复用，你可能会想将它们从 packagist.org 上分离出来。其中一个例子就是 wordpress 的插件。

对于自行托管的软件包，建议使用 `composer` 类型资源库设置，它将提供最佳的性能。

这里有一些工具，可以帮助你创建 `composer` 类型的资源库。

Packagist

packagist 的底层是开源的。这意味着你可以只安装你自己拷贝的 packagist，改造并使用它。这真的是很直接简单的事情。然而，由于其规模和复杂性，对于大多数中小型企业还是建议使用 Satis。

Packagist 是一个 Symfony2 应用程序，并且托管在 GitHub 上 github.com/composer/packagist。它内部使用了 composer 并作为 VCS 资源库与 composer 用户之间的代理。它拥有所有 VCS 资源包的列表，定期重新抓取它们，并将其作为一个 composer 资源库。

要设置你的副本，只需要按照 github.com/composer/packagist 的说明进行操作。

Satis

Satis 是一个静态的 `composer` 资源库生成器。它像是一个超轻量级的、基于静态文件的 packagist 版本。

你给它一个包含 `composer.json` 的存储库，定义好 VCS 和 资源库。它会获取所有你列出的包，并打印 `packages.json` 文件，作为 `composer` 类型的资源库。

更多详细信息请查看 github.com/composer/satis 和 Satis article。

Artifact

在某些情况下，或许没有能力拥有之前提到的任何一种线上资源库。Typical example could be cross-organisation library exchange through built artifacts。当然大部分的时间他们都是私有的。为了简化维护，可以简单的使用 `artifact` 资源库类型，来引用一个包含那些私有包的 ZIP 存档的文件夹：

```
{
    "repositories": [
        {
            "type": "artifact",
            "url": "path/to/directory/with/zips/"
        }
    ],
    "require": {
        "private-vendor-one/core": "15.6.2",
        "private-vendor-two/connectivity": "*",
        "acme-corp/parser": "10.3.5"
    }
}
```

每个 zip artifact 都只是一个 ZIP 存档，放置在 `composer.json` 所在的根目录：

```
unzip -l acme-corp-parser-10.3.5.zip

composer.json
...
```

如果有两个不同版本的资源包，它们都会被导入。当有一个新版本的存档被添加到 artifact 文件夹，并且你运行了 `update` 命令，该版本就会被导入，并且 Composer 将更新到最新版本。

禁用 Packagist

你可以在 `composer.json` 中禁用默认的 Packagist 资源库。

```
{
    "repositories": [
        {
            "packagist": false
        }
    ]
}
```

社区

已经有很多人在使用 composer，也有很多人为它做出了贡献。

贡献

如果你想为 composer 做出自己的贡献，请阅读 README。

最重要的原则介绍如下：

> 所有贡献的代码 - 包括那些具有提交权限的人 - 必须通过一个 pull request 提交，并在合并前由核心开发人员的核准。
>
> Fork 这个项目，创建一个特性分支，并给我们发送 pull request。
>
> 为了与基础代码保持一致，你应该确保代码遵循 编码规范。

IRC频道 / 邮件列表

邮件列表：用户支持 / 开发者。

irc.freenode.org 上的 IRC 频道：#composer （用户）/ #composer-dev（开发者）。

Stack Overflow 上有越来越多 Composer 相关问题的收藏。