命令手册

目录
概述
命令类型
命令结构剖析
必需的注册参数
快速简易执行
可选注册参数
Annotating with PHPDoc
短描述
长描述
参数别名
YAML 参数选项
文档块标签
WP_CLI::add_command() 的第三个 $args 参数 Command internals
Accepting arguments
Effectively reusing WP-CLI internal APIs
帮助信息渲染
编写测试
Adding commands globally
分发
包含在插件或主题中
Distribute as a stand-alone command Git 仓库
ZIP 文件

创建自定义 WP-CLI 命令比看起来更简单 ——您可以使用 wp scaffold package（仓库）动态生成除命令本身外的所有内容。

概述
WP-CLI 的目标是提供 WordPress 管理后台的完整替代方案；对于您可能在 WordPress
管理后台执行的任何操作，都应有一个等效的 WP-CLI 命令。 命令 是 WP-CLI 功能的原子单元。 wp plugin install (文档) 就是这样一个命令， wp plugin activate (文档) 也是。命令对 WordPress
用户很有用，因为它们可以为执行复杂任务提供简单、精确的接口。

但是 ，WordPress 管理后台是一把功能无限复杂的瑞士军刀。仅靠本项目不可能处理所有用例。这就是为什么
WP-CLI 包含一套 常见内置命令 ，同时也提供 丰富的内部 API供第三方编写和注册自己的命令。

WP-CLI 命令可以 作为独立包分发 ，也可以与 WordPress 插件或主题捆绑。对于前者，您可以使用 wp
scaffold package (仓库) 动态生成除命令本身之外的所有内容。

包之于 WP-CLI，犹如插件之于 WordPress。在创建 WP-CLI 包时，您应采取的方法存在明显差异。虽然 WP-CLI 是 /wp-admin 不断增长的替代方案，但重要的是要注意，在考虑如何使用 WordPress API
之前，您必须首先编写您的包以与 WP-CLI 内部 API 协同工作。

命令类型

捆绑命令：

通常涵盖标准安装 WordPress 提供的功能。但也有例外，特别是 wp search-replace（文档 ）。
不依赖插件、主题等其他组件。
由 WP-CLI 团队维护。

第三方命令：

可以在插件或主题中定义。
可以使用 wp scaffold package（仓库 ）轻松搭建为独立项目。
可以在 包索引 中独立于插件或主题分发。

所有命令：

遵循 文档标准 。

命令结构剖析

WP-CLI 支持将任何可调用的类、函数或闭包注册为命令。 WP_CLI::add_command() (文档) 用于内部和第三方命令注册。

命令的 概要 定义了命令接受哪些 位置 参数和 关联 参数。让我们看看 `wp plugin install` 的概要:

```
$ wp plugin install
usage: wp plugin install <plugin|zip|url>... [--version=<version>] [--force] [--activate] [--
activate-network]
```

在此示例中, `<plugin|zip|url>...` 是接受的 位置 参数。实际上, `wp plugin install` 可以多次接受相同的位置参数(要安装的插件别名、 ZIP 文件或 URL)。 `[--version=<version>]` 是接受的 关联 参数之一,用于指定要安装的插件版本。同时注意参数定义周围的方括号;方括号表示该参数是可选的。

WP-CLI 还有一系列适用于所有命令的 全局参数。例如,包含 `--debug` 意味着命令执行将显示所有 PHP 错误,并为 WP-CLI 引导过程增加额外的详细信息。

必需的注册参数

注册命令时, `WP_CLI::add_command()` 需要两个参数:

1. `$name` 是命令在 WP-CLI 命名空间内的名称(例如 `plugin install` 或 `post list`)。
2. `$callable` 是命令的实现,可以是可调用的类、函数或闭包。

在以下示例中,每个 `wp foo` 实例在功能上都是等效的:

```
// 1. 命令是一个函数
function foo_command( $args ) {
    WP_CLI::success( $args[0] );
}
WP_CLI::add_command( 'foo', 'foo_command' );

// 2. 命令是一个闭包
$foo_command = function( $args ) {
    WP_CLI::success( $args[0] );
}
WP_CLI::add_command( 'foo', $foo_command );

// 3. 命令是类上的一个方法
class Foo_Command {
    public function __invoke( $args ) {
        WP_CLI::success( $args[0] );
    }
}
WP_CLI::add_command( 'foo', 'Foo_Command' );
```

```
// 4. 命令是带有构造函数参数的类上的一个方法
class Foo_Command {
    protected $bar;
    public function __construct( $bar ) {
        $this->bar = $bar;
    }
    public function __invoke( $args ) {
        WP_CLI::success( $this->bar . ':' . $args[0] );
    }
}
$instance = new Foo_Command( 'Some text' );
WP_CLI::add_command( 'foo', $instance );
```

重要的是，类的行为与函数和闭包略有不同：

- 类上的任何公共方法都会注册为命令的子命令。例如，根据上面的示例，类 `Foo` 上的方法 `bar()` 将被注册为 `wp foo bar`。但是 ...
- `__invoke()` 被视为一个魔术方法。如果一个类实现了 `__invoke()`，命令名称将注册到该方法，并且该类的其他方法都不会被注册为命令。

注意： 历史上， WP-CLI 提供了一个基础的 `WP_CLI_Command` 类供扩展，但扩展此类并非必需，也不会改变命令的行为方式。

所有命令都可以注册到它们自己的顶级命名空间（例如 `wp foo`），或者作为现有命名空间的子命令（例如 `wp core foo`）。对于后者，只需将现有命名空间作为命令定义的一部分包含在内。

```
class Foo_Command {
    public function __invoke( $args ) {
        WP_CLI::success( $args[0] );
    }
}
WP_CLI::add_command( 'core foo', 'Foo_Command' );
```

快速简易执行

需要为一次性任务编写简短脚本，且无需通过 `WP_CLI::add_command()` 正式注册？ `wp eval-file` 正是您需要的工具（ 文档）。

假设有一个 `simple-command.php` 文件：

```
<?php
WP_CLI::success( "□□□□□□□" );
```

□□ “□□ ”□□□ `wp eval-file simple-command.php` □□□□□□□□□ WordPress □ WordPress □□□□□□□ `--skip-wordpress` □□□□□□ WordPress□

□□□□□

WP-CLI □□□□□□□□□□□□□□□□□□□□□□□□□□□□ PHPDoc □□□□□□□□□ `$args` □□□□□ `WP_CLI::add_command()` □

Annotating with PHPDoc

A typical WP-CLI class looks like this:

```
<?php
/**
 * Implements example command.
 */
class Example_Command {

□/**
□ * Prints a greeting.
□ *
□ * ## OPTIONS
□ *
□ * <name>
□ * : The name of the person to greet.
□ *
□ * [--type=<type>]
□ * : Whether or not to greet the person with success or error.
□ * ---
□ * default: success
□ * options:
□ *   - success
□ *   - error
□ * ---
□ *
□ * ## EXAMPLES
□ *
```

```
 *     wp example hello Newman
 *
 * @when after_wp_load
 */
function hello( $args, $assoc_args ) {
  list( $name ) = $args;

  // Print the message with type
  $type = $assoc_args['type'];
  WP_CLI::$type( "Hello, $name!" );
}
}

WP_CLI::add_command( 'example', 'Example_Command' );
```

This command's PHPDoc is interpreted in three ways:

□□□

□□□□　　PHPDoc □□□□□□

```
/**
 *□□□□□□
```

□□□

□□□□　　PHPDoc □□□□□□

```
 * ##□□
*
 * <name>
 * :□□□□□□□□□□
*
 * [--type=<type>]
 * :□□□□□□□□□□□□□□□□□□□□
* ---
 *□□□: success
 *□□:
 *   - success
 *   - error
 * ---
```

```
 *
 * ##
*
 *     wp example hello Newman
```

- `<name>` `<name>...`
  `[<name>]`
  `[<name>...]`
- `[--type=<type>]` 'success' 'success' 'error'
  `[--error]`
- `[--field[=<value>]]` `--skip-plugins[=<plugins>]`
- `[--status=<status>...]` `...`
  - `--status=active --status=inactive` WP-CLI `$assoc_args['status']` `[ 'active', 'inactive' ]`
  - `...` ——
  - `[--verbose]`

/ `[--<field>=<value>]`

```
 * [--<field>=<value>]
 * :
```


| `$assoc_args`

```
 * [--with-dependencies|w]
 * :
```

`--with-dependencies` `-w` `$assoc_args['with-dependencies']`

|

```
* [--verbose|v|wordy]
* : 启用详细输出。
```

别名也适用于携带值的关联参数：

```
* [--format=<format>|f]
* : 输出格式。
```

运行 `wp example hello -f=json` 等同于 `wp example hello --format=json`。请注意，对于短格式别名，WP-CLI 同时接受 `-<alias>=value`（单破折号）和 `--<name>=value`（双破折号）形式。

YAML 参数选项

参数描述后的 `---` 块允许你使用 YAML 为该参数设置额外的元数据。

`default` 设置参数省略时使用的值：

```
* [--type=<type>]
* : 是否以成功或错误问候此人。
* ---
* default: success
* options:
*   - success
*   - error
* ---
```

`options` 限制可接受的值。WP-CLI 会根据此列表验证用户输入，对于不在列表中的任何值返回错误。

`sensitive` 将参数标记为包含敏感数据（例如密码、API 密钥）：

- 当使用 `--prompt` 全局标志时，WP-CLI 会将其即将运行的完整命令记录到 STDOUT。
- 任何标记为 `sensitive: true` 的参数，其值在该日志行中将被替换为 `[REDACTED]`。
- 这可以防止机密信息泄露到日志或终端历史记录中。

```
* [--password=<password>]
* : 数据库密码。
* ---
* sensitive: true
* ---
```

长描述在调用 `help` 命令时也会显示，例如 `wp help example hello`。其语法是 Markdown Extra，以下是 WP-CLI 如何处理它的一些补充说明：

- 长描述通常被视为自由格式文本。`OPTIONS` 和 `EXAMPLES` 部分名称并非强制要求，只是常见且推荐的做法。
- 部分名称（`## NAME`）会着色并以零缩进打印。
- 其他所有内容缩进 2 个字符，选项描述会再额外缩进 2 个字符。
- 自动换行有点棘手。如果你想充分利用每行的空间，并且避免出现像下一行只有一两个词这样的换行瑕疵，请遵循以下规则：
- 在冒号和空格后，将选项描述硬换行在 **75** 个字符处。
- 其他所有内容硬换行在 **90** 个字符处。

有关如何格式化命令文档的更多详细信息，请参阅 WP-CLI 的文档标准。

文档块标签

这是最后一个部分，紧接在长描述之后开始：

```
 * @when after_wp_load
 */
```

以下是已定义标签列表：

@subcommand

在某些情况下，您无法让方法名与子命令名称保持一致。例如，您不能拥有名为 `list` 的方法，因为 `list` 是 PHP 中的保留关键字。

这时 `@subcommand` 标签就能发挥作用：

```
/**
 * @subcommand list
 */
function _list( $args, $assoc_args ) {
  ...
}

/**
 * @subcommand do-chores
 */
function do_chores( $args, $assoc_args ) {
  ...
}
```

@alias

通过 `@alias` 标签，您可以为子命令添加另一种调用方式。示例：

```
/**
 * @alias hi
 */
function hello( $args, $assoc_args ) {
  ...
}
```

```
$ wp example hi Joe
Success: Hello, Joe!
```

@when

WP-CLI WP-CLI

WP-CLI WordPress

```
@when after_wp_load
```

WP-CLI WordPress

```
@when before_wp_load
```

WP-CLI WordPress `@when`

WordPress

@skipglobalargcheck

`--debug` `--user` `--quiet` WP-CLI

`@skipglobalargcheck`

```
/**
 * @skipglobalargcheck
 * @when before_wp_load
 */
function my_command( $args, $assoc_args ) {
  ...
```

```
□}
```

WP_CLI::add_command() □□□□　　$args □□

PHPDoc □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
$hello_command = function( $args, $assoc_args ) {
□list( $name ) = $args;
□$type = $assoc_args['type'];
□WP_CLI::$type( "Hello, $name!" );
□if ( isset( $assoc_args['honk'] ) ) {
□□WP_CLI::log( 'Honk!' );
□}
};
WP_CLI::add_command( 'example hello', $hello_command, array(
□'shortdesc' => '□□□□□□□',
□'synopsis' => array(
□□array(
□□□'type'        => 'positional',
□□□'name'        => 'name',
□□□'description' => '□□□□□□□□□□',
□□□'optional'    => false,
□□□'repeating'   => false,
□□),
□□array(
□□□'type'        => 'assoc',
□□□'name'        => 'type',
□□□'description' => '□□□□□□□□□□□□□□□□□□□',
□□□'optional'    => true,
□□□'default'     => 'success',
□□□'options'     => array( 'success', 'error' ),
□□),
□□array(
□□□'type'     => 'flag',
□□□'name'     => 'honk',
□□□'optional' => true,
□□),
□),
□'when' => 'after_wp_load',
□'longdesc' =>   '##□□' . "\n\n" . 'wp example hello Newman',
```

```
) );
```

□□□□ `longdesc` □□□□□□□ synopsis
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ synopsis□ `longdesc`
□□□□□□□□□

Command internals

Now that you know how to register a command, the world is your oyster. Inside your callback, your command can do whatever it wants.

Accepting arguments

In order to handle runtime arguments, you have to add two parameters to your callable: `$args` and `$assoc_args`.

```
function hello( $args, $assoc_args ) {
□/* Code goes here*/
}
```

`$args` variable will store all the positional arguments:

```
$ wp example hello Joe Doe
```

```
WP_CLI::line( $args[0] ); // Joe
WP_CLI::line( $args[1] ); // Doe
```

`$assoc_args` variable will store all the arguments defined like `--key=value` or `--flag` or `--no-flag`

```
$ wp example hello --name='Joe Doe' --verbose --no-option
```

```
WP_CLI::line( $assoc_args['name'] ); // Joe Doe
WP_CLI::line( $assoc_args['verbose'] ); // true
WP_CLI::line( $assoc_args['option'] ); // false
```

Also, you can combine argument types:

```
$ wp example hello --name=Joe foo --verbose bar
```

```
WP_CLI::line( $assoc_args['name'] ); // Joe
WP_CLI::line( $assoc_args['verbose'] ); // true
```

```
WP_CLI::line( $args[0] ); // foo
WP_CLI::line( $args[1] ); // bar
```

Effectively reusing WP-CLI internal APIs

As an example, say you were tasked with finding all unused themes on a multisite network (#2523). If you had to perform this task manually through the WordPress admin, it would probably take hours, if not days, of effort. However, if you're familiar with writing WP-CLI commands, you could complete the task in 15 minutes or less.

Here's what such a command looks like:

```
/**
 * Find unused themes on a multisite network.
 *
 * Iterates through all sites on a network to find themes which aren't enabled
 * on any site.
 */
$find_unused_themes_command = function() {
	$response = WP_CLI::launch_self( 'site list', array(), array( 'format' => 'json' ), false, true );
	$sites = json_decode( $response->stdout );
	$unused = array();
	$used = array();
	foreach( $sites as $site ) {
		WP_CLI::log( "Checking {$site->url} for unused themes..." );
		$response = WP_CLI::launch_self( 'theme list', array(), array( 'url' => $site->url, 'format' => 'json' ), false, true );
		$themes = json_decode( $response->stdout );
		foreach( $themes as $theme ) {
			if ( 'no' == $theme->enabled && 'inactive' == $theme->status && ! in_array( $theme->name, $used ) ) {
				$unused[ $theme->name ] = $theme;
			} else {
				if ( isset( $unused[ $theme->name ] ) ) {
					unset( $unused[ $theme->name ] );
				}
				$used[] = $theme->name;
			}
		}
	}
```

```
□WP_CLI\Utils\format_items( 'table', $unused, array( 'name', 'version' ) );
};
WP_CLI::add_command( 'find-unused-themes', $find_unused_themes_command, array(
□'before_invoke' => function(){
□□if ( ! is_multisite() ) {
□□□WP_CLI::error( 'This is not a multisite installation.' );
□□}
□},
) );
```

Let's run through the internal APIs this command uses to achieve its goal:

- `WP_CLI::add_command()` (doc) is used to register a `find-unused-themes` command to the `$find_unused_themes_command` closure. The `before_invoke` argument makes it possible to verify the command is running on a multisite install, and error if not.
- `WP_CLI::error()` (doc) renders a nicely formatted error message and exits.
- `WP_CLI::launch_self()` (doc) initially spawns a process to get a list of all sites, then is later used to get the list of themes for a given site.
- `WP_CLI::log()` (doc) renders informational output to the end user.
- `WP_CLI\Utils\format_items()` (doc) renders the list of unused themes after the command has completed its discovery.

□□□□□

□□□□ PHPDoc□□□□□□□□□□□□ `help` □□□□□□□□□□□□□□□

1. □□□□
2. □□□□
3. □□□□□□□□□□□□
4. □□□□

□□□□

WP-CLI □□□□□ Behat □□□□□□□□□□□□□□□ Behat □□□ WP-CLI □□□□□□□□□□□□

- □□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Behat □□□□□□□ `features/` □□□□□□□□□ `features/cli-info.feature` □□□□□□

```
Feature: □□ CLI □□

  Scenario: □□□□□□□□□
    Given □□□□□

    When □□□ `wp cli info --format=json`
    Then STDOUT □□□□□□□□□□ JSON□
      """
      {"wp_cli_packages_dir_path":"/tmp/wp-cli-home/.wp-cli/packages/"}
      """

    When □□□ `WP_CLI_PACKAGES_DIR=/tmp/packages wp cli info --format=json`
    Then STDOUT □□□□□□□□□□ JSON□
      """
      {"wp_cli_packages_dir_path":"/tmp/packages/"}
      """
```

□□□□□□□□□□□□

- **Given** □□□□□□□
- **When** □□□□□□□□□
- **Then** □□□□□□□ X□□□ Y □ Z□□

□□□□□□□ wp-cli/scaffold-package-command □□□□

Adding commands globally

If you want to have custom commands available globally, without needing to create a full plugin or package, you can use WP-CLI's `require` configuration option.

Edit your global config file (usually found at `~/.wp-cli/config.yml`) to require a PHP file:

```
require:
  - ~/.wp-cli/commands.php
```

Then create the file and add your custom commands:

```
<?php

WP_CLI::add_command( 'hello-world', function () {
```

```
    WP_CLI::success( "Hello World!" );
} );
```

Any commands registered in that file will be globally available whenever you run WP-CLI. This is great for personal helper commands that you want available across all your WordPress projects.

You can also pass the `--require=<path>` flag on the command line, or set the `WP_CLI_REQUIRE` environment variable, to load a custom PHP file for a single invocation.

分发

现在你已经创建了一个引以为傲的命令，是时候与全世界分享了。通常有两种方式可以实现。

包含在插件或主题中

分享 WP-CLI 命令的一种方式是将它们打包到你的插件或主题中。许多人通过根据 `WP_CLI` 常量是否存在来有条件地加载（并注册）命令来实现这一点。

```
if ( defined( 'WP_CLI' ) && WP_CLI ) {
	require_once dirname( __FILE__ ) . '/inc/class-plugin-cli-command.php';
}
```

或者，你可以使用 `cli_init` 钩子来注册你的命令。这个钩子在 WP-CLI Runner 启动过程中触发，为命令注册提供了一个专用的事件，无需检查 `WP_CLI` 常量。

```
/**
 * 使用 cli_init 钩子注册自定义 WP-CLI 命令。
 */
function myplugin_register_cli_commands() {
	require_once dirname( __FILE__ ) . '/inc/class-plugin-cli-command.php';
	WP_CLI::add_command( 'myplugin', 'MyPlugin_CLI_Command' );
}
add_action( 'cli_init', 'myplugin_register_cli_commands' );
```

两种方法都有效。当你需要根据 WP-CLI 是否存在来有条件地加载代码时，使用 `WP_CLI` 常量检查。当你想要挂钩到 WP-CLI 初始化过程中的特定点时，使用 `cli_init` 钩子。

Distribute as a stand-alone command

Standalone WP-CLI commands can be installed from any git repository, ZIP file or folder. The only technical requirement is to include a valid composer.json file with an autoload declaration. We recommended including `"type": "wp-cli-package"` to distinguish your project explicitly as a WP-CLI package.

Here's a full composer.json example from the server command:

```
{
"name": "wp-cli/server-command",
"description": "Start a development server for WordPress",
"type": "wp-cli-package",
"homepage": "https://github.com/wp-cli/server-command",
"license": "MIT",
"authors": [
    {
        "name": "Package Maintainer",
         "email": "packagemaintainer@homepage.com",
         "homepage": "https://www.homepage.com"
    }
  ],
"require": {
"php": ">=5.3.29"
},
"autoload": {
"files": [ "command.php" ]
}
}
```

Note the `autoload` declaration, which loads `command.php`.

Once you've added a valid composer.json file to your project repository, WP-CLI users can pull it in via the package manager from the location you opted to store it in. Here's a few examples of storage locations and the corresponding syntax of installing it via the package manager:

Git □□

□□□□□ Git □□□□□□□□□□ `package install` □□□□□ Git □□□ HTTPS □ SSH □□□

```
# □□ HTTPS □□□□□
$ wp package install https://github.com/wp-cli/server-command.git
```

```
# 使用 SSH 链接安装包
$ wp package install git@github.com:wp-cli/server-command.git
```

ZIP 文件

您可以通过将 ZIP 文件的路径提供给 `wp package install` 命令，从 ZIP 文件安装包。

```
# 使用 ZIP 文件安装包
$ wp package install ~/Downloads/server-command-main.zip
```

版本号 #2
由 壳壳虫 创建于 2026-03-26 17:08:46 UTC
由 壳壳虫 更新于 2026-03-26 17:08:46 UTC