Hooks

- Hooks
- Actions
- Advanced Topics
- Custom Hooks
- Filters

Hooks

Hooks are a way for one piece of code to interact/modify another piece of code at specific, pre-defined spots. They make up the foundation for how plugins and themes interact with WordPress Core, but they're also used extensively by Core itself.

There are two types of hooks: Actions and Filters. To use either, you need to write a custom function known as a `Callback`, and then register it with a WordPress hook for a specific action or filter.

Actions allow you to add data or change how WordPress operates. Actions will run at a specific point in the execution of WordPress Core, plugins, and themes. Callback functions for Actions can perform some kind of a task, like echoing output to the user or inserting something into the database. Callback functions for an Action do not return anything back to the calling Action hook.

Filters give you the ability to change data during the execution of WordPress Core, plugins, and themes. Callback functions for Filters will accept a variable, modify it, and return it. They are meant to work in an isolated manner, and should never have side effects such as affecting global variables and output. Filters expect to have something returned back to them.

WordPress provides many hooks that you can use, but you can also create your own so that other developers can extend and modify your plugin or theme.

Actions vs. Filters

The main difference between an action and a filter can be summed up like this:

- An action takes the info it receives, does something with it, and returns nothing. In other words: it *acts* on something and then exits, returning nothing back to the calling hook.
- A filter takes the info it receives, modifies it somehow, and returns it. In other words: it *filters* something and passes it back to the hook for further use.

Said another way:

- An action interrupts the code flow to do something, and then returns back to the normal flow without modifying anything;
- A filter is used to modify something in a specific way so that the modification is then used by code later on.

The *something* referred to is the parameter list sent via the hook definition. More on this in later sections.

More Resources

- Filter Reference
- Action Reference

Actions

Actions are one of the two types of Hooks. They provide a way for running a function at a specific point in the execution of WordPress Core, plugins, and themes. Callback functions for an Action do not return anything back to the calling Action hook. They are the counterpart to Filters. Here is a refresher of the difference between actions and filters.

Adding an Action

The process of adding an action includes two steps:

Create a *callback function*

First, create a *callback function*. This function will be run when the action it is hooked to is run.

The callback function is just like a normal function: it should be prefixed, and it should be in `functions.php` or somewhere callable. The parameters it should accept will be defined by the action you are hooking to; most hooks are well defined, so review the hooks docs to see what parameters the action you have selected will pass to your function.

Assign (*hook*) your callback function

Second, add your callback function to the action. This is called *hooking* and tells the action to run your callback function when the action is run.

When your callback function is ready, use add_action() to hook it to the action you have selected. At a minimum, `add_action()` requires two parameters:

1. `string $hook_name` which is the name of the action you're hooking to, and
2. `callable $callback` the name of your callback function.

The example below will run `wporg_callback()` when the `init` hook is executed:

```
function wporg_callback() {
  // do something
}
```

```
add_action( 'init', 'wporg_callback' );
```

You can refer to the Hooks chapter for a list of available hooks.

As you gain more experience, looking through WordPress Core source code will allow you to find the most appropriate hook.

Additional Parameters

`add_action()` can accept two additional parameters, `int $priority` for the priority given to the callback function, and `int $accepted_args` for the number of arguments that will be passed to the callback function.

Priority

Many callback functions can be hooked to a single action. The `init` hook for example gets a lot of use. There may be cases where you need to ensure that your callback function runs before or after other callback functions, even when those other functions may not yet have been hooked.

WordPress determines the order that callback functions are run based on two things: The first way is by manually setting the *priority*. This is done using the third argument to `add_action()`.

Here are some important facts about priorities:

- priorities are positive integers, typically between 1 and 20
- the default priority (meaning, the priority assigned when no `priority` value is manually supplied) is 10
- there is no theoretical upper limit on the priority value, but the realistic upper limit is 100

A function with a priority of 11 will run *after* a function with a priority of 10; and a function with a priority of 9 will run *before* a function with a priority of 10.

The second way that callback function order is determined is simply by the order in which it was registered *within the same priority value*. So if two callback functions are registered for the same hook with the same priority, they will be run in the order that they were registered to the hook.

For example, the following callback functions are all registered to the `init` hook, but with different priorities:

```
add_action( 'init', 'wporg_callback_run_me_late', 11 );
add_action( 'init', 'wporg_callback_run_me_normal' );
add_action( 'init', 'wporg_callback_run_me_early', 9 );
add_action( 'init', 'wporg_callback_run_me_later', 11 );
```

In the example above:

- The first function run will be `wporg_callback_run_me_early()`, because it has a manual priority of 9
- Next, `wporg_callback_run_me_normal(),` because it has no priority set and so its priority is 10
- Next, `wporg_callback_run_me_late()` is run because it has a manual priority of 11
- Finally, `wporg_callback_run_me_later()` is run: it also has a priority of 11, but it was hooked after `wporg_callback_run_me_late()`.

Number of Arguments

Sometimes it's desirable for a callback function to receive some extra data related to the action being hooked to.

For example, when WordPress saves a post and runs the `save_post` hook, it passes two parameters to the callback function: the ID of the post being saved, and the post object itself:

```
do_action( 'save_post', $post->ID, $post );
```

When a callback function is registered for the `save_post` hook, it can specify that it wants to receive those two parameters. It does so by telling `add_action` to expect them by (in this case) putting `2` as the fourth argument:

```
add_action( 'save_post', 'wporg_custom', 10, 2 );
```

In order to actually receive those parameters in your callback function, modify the parameters your callback function will accept, like this:

```
function wporg_custom( $post_id, $post ) {
  // do something
}
```

[tip]]It's good practice to give your callback function parameters the same name as the passed parameters, or as close as you can.[/tip]

Advanced Topics

Removing Actions and Filters

Sometimes you want to remove a callback function from a hook that another plugin, theme or even WordPress Core has registered.

To remove a callback function from a hook, you need to call `remove_action()` or `remove_filter()`, depending whether the callback function was added as an Action or a Filter.

The parameters passed to `remove_action()` / `remove_filter()` must be identical to the parameters passed to `add_action()` / `add_filter()` that registered it, or the removal won't work.

[alert]To successfully remove a callback function you must perform the removal after the callback function was registered. The order of execution is important.[/alert]

Example

Let's say we want to improve the performance of a large theme by removing unnecessary functionality.

Let's analyze the theme's code by looking into `functions.php`.

```
function wporg_setup_slider() {
  // ...
}
add_action( 'template_redirect', 'wporg_setup_slider', 9 );
```

The `wporg_setup_slider` function is adding a slider that we don't need, which probably loads a huge CSS file followed by a JavaScript initialization file which uses a custom written library the size of 1MB. We can can get rid of that.

Since we want to hook into WordPress after the `wporg_setup_slider` callback function was registered (`functions.php` executed) our best chance would be the `after_setup_theme` hook.

```
function wporg_disable_slider() {
  // Make sure all parameters match the add_action() call exactly.
  remove_action( 'template_redirect', 'wporg_setup_slider', 9 );
}
// Make sure we call remove_action() after add_action() has been called.
add_action( 'after_setup_theme', 'wporg_disable_slider' );
```

Removing All Callbacks

You can also remove all of the callback functions associated with a hook by using `remove_all_actions()` / `remove_all_filters()`.

Determining the Current Hook

Sometimes you want to run an Action or a Filter on multiple hooks, but behave differently based on which one is currently calling it.

You can use the `current_action()` / `current_filter()` to determine the current Action / Filter.

```
function wporg_modify_content( $content ) {
  switch ( current_filter() ) {
    case 'the_content':
      // Do something.
      break;
    case 'the_excerpt':
      // Do something.
      break;
  }
  return $content;
}

add_filter( 'the_content', 'wporg_modify_content' );
add_filter( 'the_excerpt', 'wporg_modify_content' );
```

Checking How Many Times a Hook Has Run

Some hooks are called multiple times in the course of execution, but you may only want your callback function to run once.

In this situation, you can check how many times the hook has run with the `did_action()`.

```
function wporg_custom() {
  // If save_post has been run more than once, skip the rest of the code.
  if ( did_action( 'save_post' ) !== 1 ) {
    return;
  }
  // ...
}
add_action( 'save_post', 'wporg_custom' );
```

Debugging with the "all" Hook

If you want a callback function to fire on every single hook, you can register it to the `all` hook. Sometimes this is useful in debugging situations to help determine when a particular event is happening or when a page is crashing.

```
function wporg_debug() {
  echo '<p>' . current_action() . '</p>';
}
add_action( 'all', 'wporg_debug' );
```

Custom Hooks

An important, but often overlooked practice is using custom hooks in your plugin so that other developers can extend and modify it.

Custom hooks are created and called in the same way that WordPress Core hooks are.

Create a Hook

To create a custom hook, use `do_action()` for Actions and `apply_filters()` for Filters.

[info]We recommend using `apply_filters()` on any text that is output to the browser. Particularly on the frontend.

This makes it easier for plugins to be modified according to the user's needs.[/info]

Add a Callback to the Hook

To add a callback function to a custom hook, use `add_action()` for Actions and `add_filter()` for Filters.

Naming Conflicts

Naming conflicts ("collisions") occur when two developers use the same hook name for completely different purposes. This leads to difficult to find bugs. So it's important to prefix your hook names with a unique string to avoid hook name collisions.collisions with other plugins.

For example, a filter named `email_body` is generic enough that two or more developers could use this hook in different plugins for different purposes. So to avoid this, a prefix is added. For example, functions used as examples in this handbook use `wporg_` as the prefix.

When you choose your prefix, you can use your company name, your wp handle, the plugin name, anything you like really. The goal is to make it unique so choose wisely.

Examples

Extensible Action: Settings Form

If your plugin adds a settings form to the Administrative Panels, you can use Actions to allow other plugins to add their own settings to it.

```
do_action( 'wporg_after_settings_page_html' );
```

Now another plugin can register a callback function for the `wporg_after_settings_page_html` hook and inject new settings:

```
add_action( 'wporg_after_settings_page_html', 'myprefix_add_settings' );
```

Note that because this is an action, no value is returned. Also note that since no priority is given, it will run at default priority 10.

Extensible Filter: Custom Post Type

In this example, when the new post type is registered, the parameters that define it are passed through a filter, so another plugin can change them before the post type is created.

```
function wporg_create_post_type() {
  $post_type_params = [/* ... */];

  register_post_type(
    'post_type_slug',
    apply_filters( 'wporg_post_type_params', $post_type_params )
  );
}
```

Now another plugin can register a callback function for the `wporg_post_type_params` hook and change post type parameters:

```
function myprefix_change_post_type_params( $post_type_params ) {
  $post_type_params['hierarchical'] = true;
  return $post_type_params;
}
add_filter( 'wporg_post_type_params', 'myprefix_change_post_type_params' );
```

Note that filters take data, modify it, and return it. So the code called (`myprefix_change_post_type_params`) doesn't output anything using echo or html, or anything else directly to the screen. Also note that the returned value is used directly by `register_post_type` without being assigned to a variable first. This is simple to skip that extra (an unnecessary) step.

Also note that since no priority is given, it will run at default priority 10. And since there is no value for the number of parameters expected, the default of one is assumed.

Filters

Filters are one of the two types of Hooks.

They provide a way for functions to modify data during the execution of WordPress Core, plugins, and themes. They are the counterpart to Actions.

Unlike Actions, filters are meant to work in an isolated manner, and should never have side effects such as affecting global variables and output. Filters expect to have something returned back to them.

Add Filter

The process of adding a filter includes two steps.

First, you need to create a Callback function which will be called when the filter is run. Second, you need to add your Callback function to a hook which will perform the calling of the function.

You will use the `add_filter()` function, passing at least two parameters:

1. `string $hook_name` which is the name of the filter you're hooking to, and
2. `callable $callback` the name of your callback function.

The example below will run when the `the_title` filter is executed.

```
function wporg_filter_title( $title ) {
  return 'The ' . $title . ' was filtered';
}
add_filter( 'the_title', 'wporg_filter_title' );
```

Lets say we have a post title, "Learning WordPress", the above example will modify it to be "The Learning WordPress was filtered".

You can refer to the Hooks chapter for a list of available hooks.

As you gain more experience, looking through WordPress Core source code will allow you to find the most appropriate hook.

Additional Parameters

`add_filter()` can accept two additional parameters, `int $priority` for the priority given to the callback function, and `int $accepted_args` for the number of arguments that will be passed to the callback function.

For detailed explanation of these parameters please read the article on Actions.

Example

To add a CSS class to the `<body>` tag when a certain condition is met:

```
function wporg_css_body_class( $classes ) {
  if ( ! is_admin() ) {
    $classes[] = 'wporg-is-awesome';
  }
  return $classes;
}
add_filter( 'body_class', 'wporg_css_body_class' );
```