www.phparch.com

October 2020
Volume 19 - Issue 10

php[architect]

Parallel Running

Build An All-In-One Application Server Using Swoole

More Than Asynchronous I/O, Introduction To Swoole PHP

Serverless File Uploading

Free Sample Article

ALSO INSIDE

Education Station: Race Conditions and Dead Locks

Community Corner: Larabelles

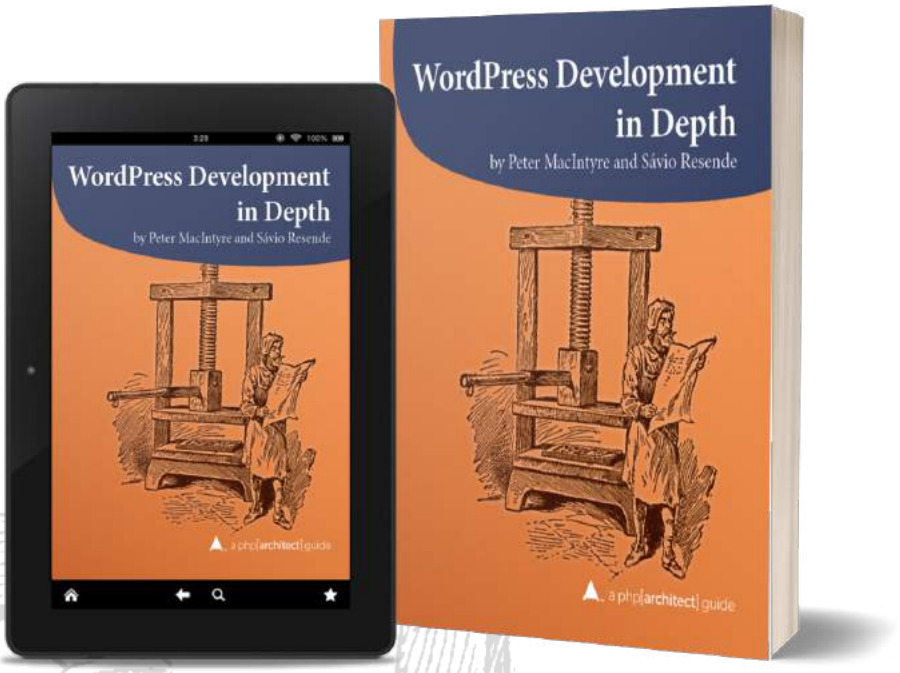
The Workshop: PHP Development with Homestead in WSL

Sustainable PHP: Refactor to Competitive Advantage

Security Corner: Observable Security

PHP Puzzles: Improved Directions

finally{}: Async Life



Build a custom, secure, multilingual website with WordPress

WordPress is more than a blogging platform—it powers one-fifth of all sites. Since its release, enthusiastic contributors have pushed the envelope to use it as a platform for building specialized social networks, e-commerce storefronts, business sites, and more. This guide helps you sort through the vast number of plugins available to find the ones that work for you. You'll also learn how to use CSS and PHP code to tailor WordPress to your needs further.

Written by PHP professionals Peter MacIntyre and Sávio Resende, this book distills their experience building online solutions for other WordPress site builders, developers, and designers to leverage and get up to speed quickly.

Read a sample and purchase your copy at the link below.

Order Your Copy
http://phpa.me/wpdevdepth-book

More Than Asynchronous I/O, Introduction To Swoole PHP

Bruce Dou

Swoole PHP is a coroutine based asynchronous network application framework. It is a PHP extension that extends PHP core and utilizes more power provided by Linux OS. Unlike the callback style single thread asynchronous I/O provided by the other networking libraries like Node.js, Swoole PHP has multiple asynchronous I/O threads and native coroutines to manage the execution of concurrent tasks. Other than asynchronous I/O and coroutine, we can also use the Linux system API and interface exposed by Swoole PHP, such as process management, system signals handling, timer and scheduler, poll and epoll I/O, memory management, etc.

What Problem Does Swoole PHP Solve?

Swoole PHP[1] is designed for building large scale concurrent systems. The system needs to manage more than 10K concurrent connections. It is a solution to the famous C10K problem with PHP and a stateful architecture.

We cannot use PHP-FPM's stateless nature to build a system with modern web protocols like WebSockets, which require establishing persistent connections with every user and sending, receiving data in real-time.

With Swoole PHP, building a real-time service is as simple as building PHP-FPM applications we are familiar with. You can define how to respond to requests in the predefined callback functions to build a WebSocket server or TCP/UDP server with only several lines of code.

Compared with the other single thread callback style asynchronous I/O frameworks built with PHP or JavaScript, Swoole is released as a PHP extension, including high-performance protocol parsers written with C/C++, multiple asynchronous I/O handling threads, native lightweight thread coroutine, and multiple processes architecture.

Swoole PHP also exposes the under layer Linux API to the PHP layer. With it, we can use PHP to build system-level software.

It is not designed to replace the stateless PHP-FPM but as a different and additional stateful programming model for PHP developers and system architects. Unlike PHP-FPM, a Swoole application is started as a Linux process you can define with PHP code. If you are not deploying it with Docker, you can use a Linux process manager such as `supervisord` or `systemd` to manage the Swoole application.

Swoole PHP provides coroutine and asynchronous I/O for concurrent programming, which we can use to manage thousands of persistent concurrent connections either as a server or as a client with only one process.

The First Glance: HTTP Echo Server

Let's start with an echo server with HTTP protocol shown in Listing 1, which processes an HTTP request and sends back a `hello world` HTTP response to a browser or HTTP client. It is executed as a PHP CLI application. Use a port number more than 1024 if you like to run the application as a non-root user.

Compared with PHP-FPM applications we are familiar with, there is a bit more boilerplate code. We have to define a server with an IP address, port to bind on first, and then define how we process the HTTP request. It gives us more flexibility since we can choose the IP address, port, protocol at the PHP application layer. Traditional PHP applications that sit behind a web server couldn't change those values.

When the server starts, it initializes several reactor threads implemented with `epoll` in Linux or `kqueue` in OSX to perform non-blocking I/O operations. Event loops manage the status of file descriptors in these reactor threads. Swoole

Listing 1.

```
1. <?php
2. declare(strict_types=1);
3.
4. use Swoole\HTTP\Server;
5. use Swoole\Http\Request;
6. use Swoole\Http\Response;
7.
8. $server = new Server("0.0.0.0", 9900);
9.
10. $server->on("request",
11.    function (Request $request, Response $response) {
12.       $response->header("Content-Type", "text/plain");
13.       $response->end("Hello World\n");
14.    }
15. );
16.
17. $server->start();
```

1 *Swoole PHP:* https://www.swoole.co.uk

doesn't support Windows OS unless you run it inside a Linux environment with Docker.

To better utilize the power of the multiple CPU cores, we create numerous `worker processes` to serve the requests sent to the server. The data received by reactor threads are dispatched to the callback functions in the worker processes registered on the server.

We can also write similar code to create a Server speaking with other protocols such as TCP, UDP, WebSocket, MQTT, etc. The code in Listing 2 is a similar server with TCP protocol.

If we build a similar TCP server with socket API provided by PHP, there will be much more boilerplate code, and you have to understand how Linux sockets work. You can compare the above code with the PHP sockets example[2].

You can send a TCP package `hey` to the server with Netcat `nc` and receive the response generated by the server: `Hello: hey`.

```
(echo 'hey'; sleep 1) | nc 127.0.0.1 9900
```

You can define a custom protocol using the TCP protocol specifically fitting for your application to reduce HTTP overhead.

Event Emitter And Timer

When thinking about web applications, the execution can be triggered by an HTTP request or triggered by some predefined events or rules.

For example, many PHP applications perform housekeeping jobs with fixed intervals like sending the daily email, sending a queued email, refreshing caches, recalculating a billboard, or pushing fresh data to the user's browser.

We can do this with Linux CRON jobs to simulate user requests by sending an HTTP request to the HTTP application, or run a PHP CLI script to avoid the HTTP server-side timeout limitation. But this approach relies on an external system `CRON` with a minimum minute interval scheduler. To schedule second interval tasks with `CRON`, `sleep` is required, which is more complex and inconvenient.

In Swoole PHP, we can define a ticker executed every second or even every millisecond within `Server` and start the ticker with the `Server`, see Listing 3.

Notice the ticker only executes in one `worker`, which is the one with `worker_id` of `0` (zero). As mentioned before, the server creates multiple worker processes, and we only want the events to be triggered once per second. Otherwise, all the worker threads would fire every second.

With the ability to trigger an event every second or every millisecond, we can build near-real-time applications.

Compare With PHP-FPM

Swoole PHP runs as a PHP CLI application. It doesn't rely on external process manager PHP-FPM or Apache server.

2 *PHP sockets example: https://php.net/sockets.examples*

Listing 2.

```
<?php
declare(strict_types=1);

use Swoole\Server;

$server = new Server("0.0.0.0", 9900);

$server->on("receive",
    function (Server $server, int $fd,
              int $reactor_id, string $data) {
        $server->send($fd, "Hello: {$data}");
        $server->close($fd);
    }
);

$server->start();
```

Listing 3.

```
<?php
declare(strict_types=1);

use Swoole\HTTP\Server;
use Swoole\Http\Request;
use Swoole\Http\Response;

$server = new Server("0.0.0.0", 9900);

$server->on("workerStart",
    function (Server $server, int $worker_id) {
        if ($worker_id === 0) {
            $server->tick(1000, function () {
                echo time() . "\n";
            });
        }
    }
);

$server->on("request",
    function (Request $request, Response $response) {
        $response->header("Content-Type", "text/plain");
        $response->end("Hello World\n");
    }
);

$server->start();
```

PHP-FPM helps us to manage multiple PHP processes serving the HTTP requests. So, we don't have to think about Process Management when writing PHP applications. The downside is that we cannot define or customize how PHP processes are launched or managed with PHP.

PHP-FPM is a stateless design; the whole context is created when a new request is received and destroyed when the request finishes. The stateless and share-nothing design means we can't save the global state within the PHP process for different requests. We have to re-create them every time or save that

state somewhere. Although most global state resources do not change from request to request, they are repeatedly created and destroyed. In Swoole PHP, resources are reused by multiple requests, thus has a much better performance.

The I/O in PHP-FPM is blocking by default. Everything is sequential, even for the independent tasks within a request, they have to be executed one by one. In Swoole PHP, separate logics within a request can be performed concurrently with multiple coroutines to reduce the request latency.

There is only one protocol supported by PHP-FPM, which is FastCGI. We need a proxy like NGINX to convert the HTTP protocol to FastCGI and back. Swoole PHP provides the most commonly used protocols such as TCP, UDP, HTTP, WebSocket, etc. You can find more about the protocols supported by Swoole[3].

Process Management

We always like to finish a task faster. But you might see it is difficult to utilize all the CPU cores on your machine when running PHP CLI scripts to process a large amount of data.

We can use the power of all CPU cores with a message queue and manage multiple identical consumer processes with `Swoole\Process\Pool` (Listing 4).

You can send tasks or events into the process pool as JSON strings with TCP protocol, or pull the data from an external message queue like Redis or RabbitMQ and process the messages with these processes.

When a process terminates unexpectedly, the pool will launch another process to maintain the defined number of processes.

Integrate With Linux Processes

Swoole PHP provides a set of complete and straightforward APIs similar to `pcntl_fork`. We can define multiple processes, wrap the logic into the processes, access Linux native processes, and easily communicate with them.

Let's see an example about how to expose a Linux process into your PHP application. Refer to Listing 5.

We have created a `cat` process reading the Linux `loadavg` status and send the data to the main process in this example. Swoole PHP has given PHP developers the power to access any Linux processes. It is convenient for system integration or exposes any Linux command-line application as an HTTP service. `Swoole\Process::exec()` is an API provided by Swoole to execute external commands and communicate with the command. You can find more about `Swoole\Process::exec()`[4]

3 *the protocols supported by Swoole:* https://www.swoole.co.uk/protocols

4 `Swoole\Process::exec()`: http://phpa.me/swoole-process-exec

Listing 4.

```
<?php

use Swoole\Process\Pool;

const N = 10;

$pool = new Pool(N, SWOOLE_IPC_SOCKET);

$pool->on("message", function ($pool, $message) {
    echo "Message: {$message}\n";
});

$pool->listen("127.0.0.1", 8089);

$pool->start();
```

Listing 5.

```
<?php
declare(strict_types = 1);

use Swoole\Process;

$loadavg = new Process(function($process){
    $process->exec("/bin/cat", ["/proc/loadavg"]);
}, TRUE);

$loadavg->start();
$result = $loadavg->read();
echo $result;
```

Coroutine

Execution Containers: Process, Thread, And Coroutines

Processes are fundamental top-level execution containers in an operating system. They are separate tasks. Each has its dedicated memory system, and their heaps and stacks run concurrently within the OS. The Linux kernel manages the scheduling of different processes running on the same CPU cores and hardware.

Threads in Linux, on the other hand, run within one process and share some resources between them. They can share memory. If we want to communicate between two threads, we can define a `global` variable in one thread and access it directly from another one.

A coroutine in Swoole PHP is the lightweight thread created within one Linux process. Coroutines have their stack and share the global status of a PHP process like heap and other resources.

Compared with a Linux process that may reserve 8MB memory for the stack, coroutine in Swoole PHP only uses 8KB for each coroutine stack by default. By minimizing memory allocation, a coroutine is more scalable for handling concurrent I/O.

Instead of being run on kernel threads and scheduled by the operating system, coroutines are scheduled by the Swoole PHP scheduler[5]. It is not necessary to jump into the kernel space, so there is less overhead for doing the context switch between different execution containers. Only the pointers to local coroutine stacks are changed. The global space stays the same, which is much more efficient.

Coroutine Concurrent Execution

A coroutine is a closure function initialized with `coroutine::create()` or the short name form `go()`, we can pass contextual variables into the closure with `use` the same as PHP-FPM.

Multiple coroutines within the same process have the minimum isolated execution context, use the minimum dedicated resources on your machine and schedule based on I/O waiting and ready status.

Instead of binding concurrent execution with processes like PHP-FPM, you can bind concurrent execution with the lightweight coroutines using much less memory within the Swoole PHP process. Instead of handling several hundred connections with hundreds of processes, we can handle at least 1K connections with one process.

Compared with the nested callback programming style dealing with concurrent programming, it is more convenient to write the codes with sequential coroutine blocks. We can think of each concurrent block as a lightweight thread coroutine to avoid the callback hell.

The synchronization and communication between different coroutines can be done through a `Channel` object similar to the `channel` in Golang, as in Listing 6.

The example code sums the random numbers generated in two coroutines. Both of the coroutines execute concurrently. Once both coroutines have completed their computation, it calculates the final result.

The `co::sleep` is used to simulate I/O latency within the coroutine. PHP's functions using `syscall`, such as `sleep()`, should not be used within the coroutine context because it is scheduled by the Linux kernel and blocks the whole process. Instead, use the coroutine version function `co::sleep()`, scheduled by Swoole PHP scheduler with the event loop.

The total time used by an application should be the maximum time of each coroutine in the execution flow, but not the sum time of each coroutine because they should run in parallel. In this way, we can massively reduce the overall latency of a user request.

The above code shows how multiple coroutines execute concurrently and communicate with each other through a channel.

We can think of `channel` as a message queue with fixed sizes, pushes or pops block until the other side is ready. It can be used by coroutines to synchronize without explicit locks.

5 *Swoole PHP scheduler: http://phpa.me/swoole-coroutine*

Listing 6.

```
<?php

use Swoole\Coroutine\Channel;

Co\run(function() {
    $chan = new Channel(1);

    go(function() use ($chan) {
        Co::sleep(0.1);
        $n = rand(1000, 9999);
        echo $n. "\n";
        $chan->push($n);
    });

    go(function() use ($chan) {
        Co::sleep(0.5);
        $n = rand(1000, 9999);
        echo $n. "\n";
        $chan->push($n);
    });

    $sum = $chan->pop() + $chan->pop();
    echo $sum. "\n";
});
```

You can write a similar piece of code within the HTTP callback block and execute multiple blocks with the database I/O or cache I/O concurrently to achieve low latency.

Co And Coroutine Context

As seen in the code from the previous section, a coroutine context is created with function `co\run()`. Then multiple concurrent coroutines can be created with the function `go()` within the coroutine context.

A coroutine context is created in a server automatically for each `request` or `receive` callback function. Multiple independent coroutines can be created and executed concurrently within the `request` or `receive` callback functions.

Variable Scope And Life Cycle

The scope of a variable is the context within which it is defined and can be used safely. The variable scope in Swoole PHP is different with PHP-FPM. Understanding the variable scope can help you avoid memory leaks and unexpected effects.

We can use several types of variables in Swoole PHP, similar to PHP-FPM: local variables, global variables, static variables.

Local variables within a coroutine are created and destroyed with the life cycle of the coroutine. They are not visible to other coroutines.

As mentioned before, coroutines are lightweight threads, `multiple coroutines share global` variables, and `static` variables within the same process. But you have to be very careful when using these global variables and take care of the life cycle. The reason is Swoole PHP may launch multiple

processes, one worker process may be terminated, and a new worker process may be launched at any time.

On the other hand, it is worth mentioning `global` and `static` variables can not be shared across multiple processes. So it is not recommended to use them in your Swoole PHP application. The superglobal variables provided by PHP-FPM like `$GLOBALS`, `$_GET` also should not be used within a Swoole PHP server. Instead, `Swoole\HTTP\Request`[6] should be used to access the variables of an HTTP request, and `Swoole\HTTP\Response` should be used to write the response data.

In general, it is not recommended to use global variables in Swoole PHP. Instead, use external storage or the built-in memory storage provided by Swoole PHP when required.

Built-in Memory Storage

Where to store the application states is always a tradeoff about the scalability and performance when designing a system because of the latency and overhead. Correctly using and managing the local status is the key to building a high-performance system.

In a typical PHP-FPM system, your application would either need to make a network call to a remote database or connect to the database process on the same machine. Either way, there is an amount of latency overhead. With Swoole PHP, you can store states like variables that can be reused across different requests or database connections locally within the same memory space as the application.

Internally, Swoole PHP may launch multiple processes depending on the server configuration. Global variables should not be used across multiple processes. So, in Swoole PHP, there is an atomic counter `Atomic` that can be safely accessed and updated concurrently by numerous processes. You can also use in-memory ephemeral key-value storage Table[7] to store more complex data structure.

The typical use case of `Table` is storing the per-connection user status of a WebSocket Server.

CPU Intensive Logics And Preemptive Scheduling

A good consumer-facing system has to ensure fairness to serve different users, different requests, and avoid the long-tail latency. You can use Swoole PHP where you need soft real-time latency guarantees, such as in a real-time bidding or trading system.

Other than I/O based scheduling, Swoole PHP has the preemptive scheduling mechanism for soft real-time latency guarantees.

Like the scheduler of Linux OS, each coroutine is allowed to run for a small amount of time, 5ms. When this time has expired, another coroutine is selected to run. The original

6 `Swoole\HTTP\Request`: http://phpa.me/swoole-http-request

7 *Table*: http://phpa.me/swoole-table

coroutine is made to wait for a little while until it can run again.

This is essential when there is CPU intensive logic within your application. Because multiple requests are processing concurrently within one process, we don't want one request to block all the other requests just because the CPU calculation is not finished. Instead, the scheduler should pause the current coroutine and switch to the other coroutines for better utilizing the resources.

Use Cases And Patterns

Compared with the stateless PHP-FPM model, Swoole provides a stateful server model for PHP developers. Much more can be accomplished with PHP syntax.

HTTP Services

As Swoole HTTP Server provides the request and response mechanism, it can be used as an HTTP server running your PHP application. But you might have to make changes to the `global` variables and `static` variables, binding them with an HTTP request.

Besides running as an HTTP Server similar to PHP-FPM, there are several other everyday use cases.

Client-side Connection Pooling

Swoole can be used as a connection pool sidecar of PHP-FPM and establish persistent connections to a remote database or services. In this way, the latency with the remote database is mostly reduced.

HTTP Proxy And API Gateway

Swoole PHP can be used as an HTTP proxy in front of your application to inspect, modify the HTTP request or response.

You can build a throttling and rate-limiting authentication proxy layer with PHP logics.

Message Queue Consumer Workers

Lots of people use it to manage the message queue consumer processes to do large scale data processing. It is a perfect fit for event-driven architecture.

The events can be pulled with multiple processes within one Swoole Process Pool and pushed back to your internal PHP-FPM HTTP services or serverless APIs.

Access And Integrate With Linux Processes

For instance, you can also build a simple Linux monitoring system by scraping `/proc` stats every second and sending a notification to a Slack channel or your email when the status reaches the threshold you have defined. Or the system status can be reported to a central server and provide you with an aggregated status.

As An Aggregate And Routing Layer For Microservices

The routing logic can be implemented with PHP and dynamically created and updated by the PHP application. As a front door aggregate layer of your microservices, multiple calls to different services can happen concurrently.

Integrate With Your ServerLess Stack

Serverless architecture computing is gaining popularity. Swoole PHP can be used to glue Serverless services.

You can use Swoole PHP as the API gateway or aggregate layer for your ServerLess stack. Or generate the ticker to trigger the execution flow of the ServerLess stack. Deliver the events in a message queue to your serverless APIs.

Conclusion

The typical use case of Swoole PHP is the same as PHP-FPM, building a high-performance HTTP service. We can also use it to make TCP services with a custom protocol or stateful applications. Or run it as a sidecar with PHP-FPM application for background data processing or scheduler. You can also use Swoole PHP as an integration layer to access and integrate with native Linux applications and processes.

After reading this article, I hope you can see how Swoole can be used as a new component within your existing system and architecture. Web framework authors are encouraged to integrate the framework with Swoole PHP to gain a much better performance. For a large scale web system, by migrating into Swoole PHP, you usually can save 80% of the server resources.

Swoole PHP opened the door of system-level programming for PHP developers, providing the lightweight thread coroutine and asynchronous I/O API to the PHP userland. It is a new PHP programming model compared with the PHP-FPM application we are familiar with. A little piece of PHP code wrapped with `Co\run` can do much more than we expected.

Bruce Dou is the Director at Transfon, a UK based company building modern infrastructure for publishers and marketers, providing support, consultancy and managed services of PHP Swoole systems. He is one of the maintainers of open source PHP Swoole, specialized in application performance optimization, system cost reduction, cloud migration, web infrastructure and large-scale system architecture.

Related Reading

- *Asynchronous Programming in PHP* by Lochemem Bruno Michael, June 2020. https://phpa.me/async-php-june-20
- *Evolving PHP* by Chris Pitt, March 2018. https://phparch.com/article/evolving-php/
- *Distributed Workers and Events* by Christopher Pitt, June 2016. https://www.phparch.com/magazine/2016-2/august/

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]