
swob

目　录

开始

本文示例代码详见：https://github.com/52fhy/swoole_demo。

简介

Swoole是一个PHP扩展，提供了PHP语言的异步多线程服务器，异步TCP/UDP网络客户端，异步MySQL，异步Redis，数据库连接池，AsyncTask，消息队列，毫秒定时器，异步文件读写，异步DNS查询。 Swoole内置了Http/WebSocket服务器端/客户端、Http2.0服务器端。

Swoole: PHP的异步、并行、高性能网络通信引擎
http://www.swoole.com/

Github:
https://github.com/swoole
https://github.com/matyhtf

安装

Swoole需要使用源码安装。暂无Windows版扩展。

源码安装

```
wget -O swoole.zip https://github.com/swoole/swoole-src/archive/v1.9.11.zip
unzip swoole.zip
cd swoole
phpize
./configure
make && make install
```

pecl安装

由于pecl是需要编译的，所以需要先安装编译器（已安装编译器可以忽略）：

```
yum install -y gcc gcc-c++ make cmake bison autoconf
```

然后：

```
pecl install swoole
pecl install redis
```

pecl安装扩展完成后会提示添加so文件到php.ini。示例：

```
Build process completed successfully
Installing '/usr/lib64/php/modules/swoole.so'
install ok: channel://pecl.php.net/swoole-1.9.11
configuration option "php_ini" is not set to php.ini location
You should add "extension=swoole.so" to php.ini
```

添加示例：

```
[swoole]
extension = /usr/lib64/php/modules/swoole.so
```

版本选择

建议使用的版本（截止时间2017-6-3）

```
稳定版：v1.9.9
预览版：v2.0.7
```

> `1.9.x` 分支已进入特性锁定期，不再开发新功能，仅修复BUG。

最低版本：
建议 **`1.8.6+`** 。PHP7建议使用 **`1.9.2+`** 。

建议使用的PHP版本

```
PHP5.5或更高版本
PHP7.0.13或更高版本
```

> 使用 `TP3.1+` 框架的朋友升级到 `PHP7.1.0` 可能会出现rewrite失效问题。建议 `PHP7.0.x` 系列。

快速查看当前swoole的版本：

```
php --ri swoole
```

`1.8.6~1.8.13` 都是小范围BUG修复及功能新增。其中 `1.8.11` 增加SIGRTMIN信号处理函数，用于重新打开日志文件。

`1.8.6` 版本是一个重要的BUG修复版本，主要修复了PHP7环境下HttpServer、TCPClient、HttpClient、Redis等客户端存在的内存泄漏、崩溃问题。

`1.9.0` 版本增加了多项新特性，修复了多个已知问题。1.9版本是100%向下兼容1.8的，用户可无缝升级。

`1.9.1` 修复PHP7下启用opcache导致崩溃的问题；重构 `reopen log file` 特性，收到 `SIGRTMIN` 信号后重新打开日志文件并重定向标准输出 等。

`1.9.2` 修复PHP7下发生 `zend_mm_heap corrupted` 的问题 等。

`1.9.4` 修复WebSocket服务器默认onRequest方法内存泄漏问题 等。

`1.9.5` 增加pid_file选项，在Server启动时将主进程ID写入指定的文件 等。

`1.9.6` 修复添加超过1万个以上定时器时发生崩溃的问题；增加swoole_serialize模块，PHP7下高性能序列化库；修复监听UDP端口设置onPacket无效的问题 等。

`1.9.9` 修复Http2客户端POST数据时协议错误问题 等。

`1.9.11` 修复WebSocket服务器onOpen回调函数存在内存泄漏的问题；修复Http服务器文件上传在5.6版本发生崩溃的问题；优化添加Task和Timer的定时器性能，提升分支预测成功率 等。

Swoole的进程模型

运行模式

Swoole目前总共有三种运行模式，默认为多进程模式（`SWOOLE_PROCESS`）。

```
# Base模式（SWOOLE_BASE）
传统的异步非阻塞Server，reactor和worker是同一个角色。TCP连接是在worker进程中维持的。
如果客户端连接之间不需要交互，可以使用BASE模式。如Memcache、Http服务器等。

# 线程模式
多线程Worker模式，Reactor线程来处理网络事件轮询，读取数据。得到的请求交给Worker线程去处理。

缺点：一个线程发生内存错误，整个进程会全部结束。
由于PHP的ZendVM在多线程模式存在内存错误，多线程模式在v1.6.0版本后已关闭。

# 进程模式
与多线程Worker模式不同的是，线程换成了进程。Reactor线程来处理网络事件轮询，读取数据。得到的请求交给Worker进程去处理。适合业务逻辑非常复杂的场景。如WebSocket服务器等。
```

```
$serv = new swoole_server(string $host, int $port, int $mode = SWOOLE_PROCESS, int $sock_type = SWOOLE_SOCK_TCP);
```

实例分析

我们来使用实例进行分析：

```
<?php
$server = new \swoole_server("127.0.0.1",8088);//默认是多进程模式、TCP类型

$server->on('connect', function ($serv, $fd){ });
$server->on('receive', function ($serv, $fd, $from_id, $data){ });
$server->on('close', function ($serv, $fd){ });

$server -> start();
```

继续在Shell中输入以下命令：

```
php swoole_tcp_server.php
pstree -ap|grep swoole_tcp_server
  |   |       `-php,2454 swoole_tcp_server.php
  |   |           |-php,2456 swoole_tcp_server.php
  |   |           |   `-php,2458 swoole_tcp_server.php
```

从系统的输出中，我们可以很容看出server其实有3个进程，进程的pid分别是2454、2456、2458，其中2454是2456的父进程，而2456又是2458的父进程。

> 所以，其实我们虽然看起来只是启动了一个Server，其实最后产生的是三个进程。

这三个进程中，所有进程的根进程(2454)，就是所谓的 `Master` 进程；而2456进程，则是 `Manager` 进程；最后的2458进程，是 `Worker` 进程。

基于此，我们简单梳理一下，当执行的start方法之后，发生了什么：

- 守护进程模式下，当前进程fork出Master进程，然后退出，Master进程触发OnMasterStart事件。
- Master进程启动成功之后，fork出Manager进程，并触发OnManagerStart事件。
- Manager进程启动成功时候，fork出Worker进程，并触发OnWorkerStart事件。

> 非守护进程模式下，则当前进程直接作为Master进程工作。

所以，一个最基础的Swoole Server，至少需要有3个进程，分别是Master进程、Manager进程和Worker进程。

事实上，一个多进程模式下的Swoole Server中，有且只有一个Master进程；有且只有一个Manager进程；却可以有n个Worker进程。

进程模型

`Master` 进程是一个多线程进程，其中有一组非常重要的线程，叫做 `Reactor` 线程（组），每当一个客户端连接上服务器的时候，都会由Master进程从已有的Reactor线程中，根据一定规则挑选一个，专门负责向这个客户端提供维持链接、处理网络IO与收发数据等服务。分包拆包等功能也是在这里完成。

`Manager` 进程，某种意义上可以看做一个代理层，它本身并不直接处理业务，其主要工作是将Master进程中收到的数据转交给Worker进程，或者将Worker进程中希望发给客户端的数据转交给Master进程进行发送。

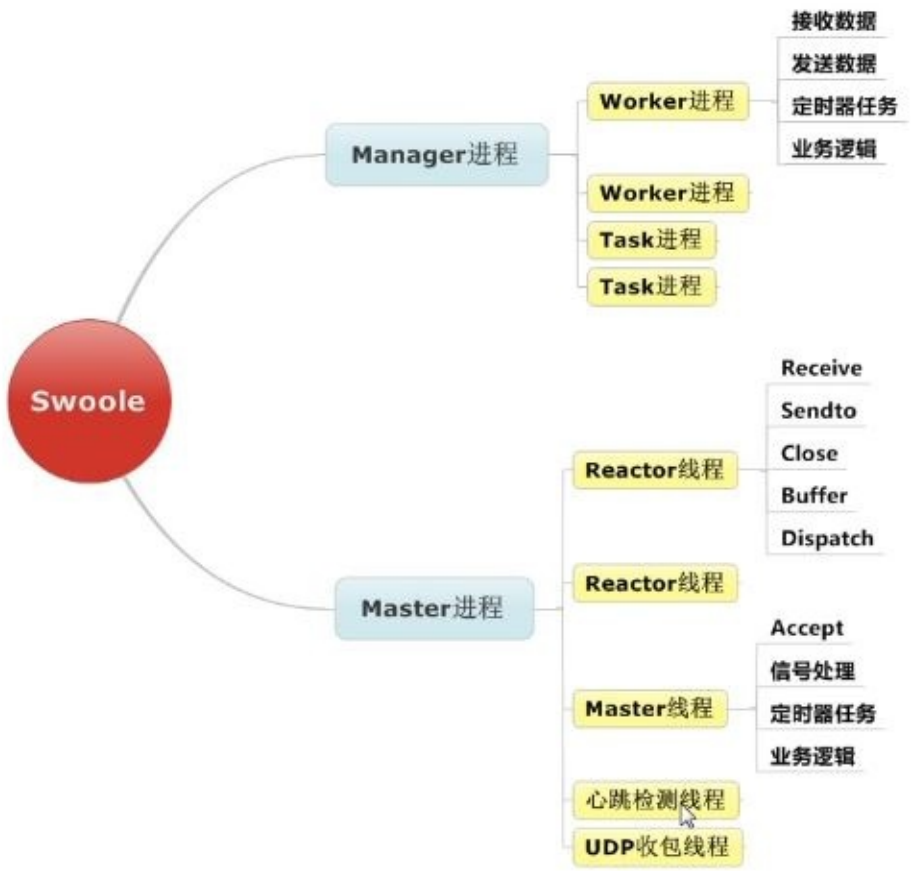
`Manager` 进程还负责监控Worker进程，如果Worker进程因为某些意外挂了，Manager进程会重新拉起新的Worker进程，有点像Supervisor的工作。而这个特性，也是最终实现热重载的核心机制。

`Worker` 进程其实就是处理各种业务工作的进程，Manager将数据包转交给Worker进程，然后Worker进程进行具体的处理，并根据实际情况将结果反馈给客户端。

我们可以总结出来上面简单的Server，当客户端连接的时候这个过程中，三种进程之间是怎么协作的：

1. Client主动Connect的时候，Client实际上是与Master进程中的某个Reactor线程发生了连接。
2. 当TCP的三次握手成功了以后，由这个Reactor线程将连接成功的消息告诉Manager进程，再由Manager进程转交给Worker进程。
3. 在这个Worker进程中触发了OnConnect的方法。
4. 当Client向Server发送了一个数据包的时候，首先收到数据包的是Reactor线程，同时Reactor线程会完成组包，再将组好的包交给Manager进程，由Manager进程转交给Worker。
5. 此时Worker进程触发OnReceive事件。
6. 如果在Worker进程中做了什么处理，然后再用Send方法将数据发回给客户端时，数据则会沿着这个路径逆流而上。

Swoole进程/线程结构图：



现在，我们基于上面的例子修改代码，来看看一个简单的多进程Swoole Server的几个基本配置：

```
<?php
$server->set(array(
        'demonize' => false,//是否后台运行
        'reactor_num' => 2,
        'worker_num' => 4
));

$server -> start();
```

`reactor_num`：表示Master进程中，Reactor线程总共开多少个，注意，这个可不是越多越好，因为计算机的CPU是有限的，所以一般设置为与CPU核心数量相同，或者两倍即可。

`worker_num`：表示启动多少个Worker进程，同样，Worker进程数量不是越多越好，仍然设置为与CPU核心数量相同，或者两倍即可。

我们可以在Shell里运行，使用pstree查看进程模型结构：

```
  php swoole_tcp_server.php
```

```
pstree -ap|grep swoole_tcp
  |   |       `-php,2505 swoole_tcp_server.php
  |   |           |-php,2507 swoole_tcp_server.php
  |   |           |   |-php,2510 swoole_tcp_server.php
  |   |           |   |-php,2511 swoole_tcp_server.php
  |   |           |   |-php,2512 swoole_tcp_server.php
  |   |           |   `-php,2513 swoole_tcp_server.php
```

回调函数

Swoole作为Server时，回调函数有很多。但可以简单分个类：
1) 进程启动时执行的：onStart、onManagerStart、onWorkerStart；onWorkerStop、onManagerStop、onShutdown；onWorkerError
2) 客户端交互时触发的：onReceive/onRequest/onPacket/onMessage、onOpen/onConnect、onClose
3) Task：onTask、onFinish
4) Timer：onTimer

事件执行顺序：

- 所有事件回调均在 `$server->start` 后发生
- 服务器关闭程序终止时最后一次事件是 `onShutdown`
- 服务器启动成功后， `onStart/onManagerStart/onWorkerStart` 会在不同的进程内并发执行。
- `onReceive/onConnect/onClose/onTimer` 在worker进程(包括task进程)中各自触发
- worker/task进程启动/结束时会分别调 用`onWorkerStart/onWorkerStop`
- `onTask` 事件仅在task进程中发生
- onFinish事件仅在worker进程中发生
- `onStart/onManagerStart/onWorkerStart` 3个事件的执行顺序是不确定的
- UDP协议下只有 `onReceive` 事件，没有 `onConnect/onClose` 事件
- 如果未设置 `onPacket` 回调函数，收到UDP数据包默认会回调 `onReceive` 函数
- `onOpen` 事件回调是可选的：当WebSocket客户端与服务器建立连接并完成握手后会回调此函数

实际使用的时候不是所有回调都可以使用的，例如UDP服务器没有 `onConnect/onClose` ；例如接收数据，在WebSocket里使用onReceive，在HttpServer使用onRequest，在UDPServer使用onPacket。

示例：

```
<?php
$server = new \swoole_server("127.0.0.1",8088);

$server->set(array(
        'daemonize' => false,
        'reactor_num' => 2,
        'worker_num' => 4
));

$server->on('connect', function ($serv, $fd){
        echo "client connect. fd is {$fd}\n";
});

$server->on('receive', function ($serv, $fd, $from_id, $data){
        echo "client connect. fd is {$fd}\n";
});
```

```
$server->on('close', function ($serv, $fd){
        echo "client close. fd is {$fd}\n";
});

// 以下回调发生在Master进程
$server->on("start", function (\swoole_server $server){
        echo "On master start.\n";
});
$server->on('shutdown', function (\swoole_server $server){
        echo "On master shutdown.\n";
});

// 以下回调发生在Manager进程
$server->on('ManagerStart', function (\swoole_server $server){
        echo "On manager start.\n";
});
$server->on('ManagerStop', function (\swoole_server $server){
        echo "On manager stop.\n";
});

// 以下回调也发生在Worker进程
$server->on('WorkerStart', function (\swoole_server $server, $worker_id){
        echo "Worker start\n";
});
$server->on('WorkerStop', function(\swoole_server $server, $worker_id){
        echo "Worker stop\n";
});
$server->on('WorkerError', function(\swoole_server $server, $worker_id, $worker_pid, $exit_code){
        echo "Worker error\n";
});

$server -> start();
```

编程须知

- 不要在代码中执行 `sleep` 以及其他睡眠函数，这样会导致整个进程阻塞
- `exit/die` 是危险的，会导致worker进程退出
- 可通过 `register_shutdown_function` 来捕获致命错误，在进程异常退出时做一些请求工作，具体参看/wiki/page/305.html
- PHP代码中如果有异常抛出，必须在回调函数中进行 `try/catch` 捕获异常，否则会导致工作进程退出
- swoole不支持 `set_exception_handler` ，必须使用 `try/catch` 方式处理异常
- Worker进程不得共用同一个 `Redis` 或 `MySQL` 等网络服务客户端，Redis/MySQL创建连接的相关代码可以放到 `onWorkerStart` 回调函数中。原因是如果共用1个连接，那么返回的结果无法保证被哪个进程处理。持有连接的进程理论上都可以对这个连接进行读写，这样数据就发生错乱了。具体参考/wiki/page/325.html
- 不能使用类的属性保存客户端连接信息，因为一个worker进程可以处理多个客户端连接，导致类属性数据错乱。常量则是可以的。

重新打开日志

在1.8.11及之后版本支持重新打开日志：向Server主进程发送SIGRTMIN信号。假设主进程id是3427，那么我们可以：

```
kill -34 3427
```

> 注：SIGRTMIN信号的id是`34`。通过`kill -l`查看。

那么如何利用这个特征实现每天自动写入新的日志文件里面呢？

假设日志文件是`/log/swoole.log`，我们可以在每天0点运行shell命令：

```
mv /log/swoole.log /log/$(date -d '-1 day' +%y-%m-%d).log
kill -34 $(ps aux|grep swoole_task|grep swoole_task_matser|grep -v grep|awk '{print $2}') # 找到主进程，需要提前命名
```

我们也可以把master进程的PID写入到文件：

```
$server->set(array(
    'pid_file' => __DIR__.'/server.pid',
));
```

在Server关闭时自动删除PID文件。此选项在1.9.5或更高版本可用。

信号管理

Swoole支持的信号：

```
SIGKILL -9 pid 强制杀掉进程
SIGUSR1 -10 master_pid 重启所有worker进程
SIGUSR2 -12 master_pid 重启所有task_worker进程
SIGRTMIN -34 master_pid 重新打开日志(版本1.8.11+)
```

master_pid代表主进程pid。示例（假设主进程名称是swoole_server，pid是3427）：

```
# 杀掉进程swoole_server
kill -9 $(ps aux|grep swoole_server|grep -v grep|awk '{print $2}')

# 重启swoole_server的worker进程
kill -10 $(ps aux|grep swoole_server|grep -v grep|awk '{print $2}')

# 重新打开日志
kill -34 3427
```

Task

我们可以在worker进程中投递一个异步任务到task_worker池中。此函数是非阻塞的，执行完毕会立即返回。worker进程可以继续处理新的请求。

通常会把耗时的任务交给task_worker来处理。

> 我们可以通过如下代码判断是Worker进程还是TaskWorker进程：
>
> ```
> function onWorkerStart($serv, $worker_id) {
>     if ($worker_id >= $serv->setting['worker_num']) {  //超过worker_num，表示这是一个task进程
> ```

```
}
```

}

```
看一个示例：
``` php
<?php
$server = new \swoole_server("127.0.0.1",8088);

$server->set(array(
        'daemonize' => false,
        'reactor_num' => 2,
        'worker_num' => 1,
        'task_worker_num' => 1,
));

$server->on('start', function ($serv){
        swoole_set_process_name("swoole_task_matser"); //主进程命名
});

$server->on('connect', function ($serv, $fd){
        echo "client connect. fd is {$fd}\n";
});

$server->on('receive', function ($serv, $fd, $from_id, $data){

        echo sprintf("onReceive. fd: %d , data: %s\n", $fd, json_encode($data) );

        $serv->task(json_encode([
                'fd' => $fd,
                'task_name' => 'send_email',
                'email_content' => $data,
                'email' => 'admin@qq.com'
        ]));
});

$server->on('close', function ($serv, $fd){
        echo "client close. fd is {$fd}\n";
```
```

```
});

$server->on('task', function (swoole_server $serv, $task_id, $from_id,  $data){
        echo $data;

        $data = json_decode($data, true);
        $serv->send($data['fd'], "send eamil to {$data['email']}, content is : {$data['email_content']}
\n");

        //echo 'task finished';
        //return 'task finished';
        $serv->finish('task finished');
});

$server->on('finish', function (swoole_server $serv, $task_id, $data){
        echo 'onFinish:' .$data;
});

$server -> start();
```

这里新建了一个tcp服务器，参数里设置`worker_num`进程为1，`task_worker_num`为1。

配置了`task_worker_num`参数后将会启用task功能，`swoole_server`务必要注册`onTask/onFinish`2个事件回调函数。如果没有注册，服务器程序将无法启动。

onTask回调接收4个参数，分别是serv对象、任务ID、自于哪个worker进程、任务的内容。注意的是，`$data`必须是字符串。我们可以在worker进程里使用`swoole_server->task($data)`进行任务投递。

onFinish回调用于将处理结果告知worker进程，此回调必须有，但是否被调用由OnTask决定。在OnTask里使用`return`或者`finish()`可以将处理结果发生到onFinish回调，否则onFinish回调是不会被调用的。也就是说：`finish()`是可选的。如果worker进程不关心任务执行的结果，不需要调用此函数。onFinish回调里的`$data`同样必须是字符串。

我们新起一个窗口，使用`telnet`发送消息到服务端进行测试：
client端：

```
telnet 127.0.0.1 8088
Trying 127.0.0.1...
Connected to 127.0.0.1.

hhh
send eamil to admin@qq.com, content is : hhh
```

server端：

```
client connect. fd is 1
onReceive. fd: 1 , data: "hhh\r\n"
{"fd":1,"task_name":"send_email","email_content":"hhh\r\n","email":"admin@qq.com"}
onFinish:task finished
```

onFinish回调里不使用`return`或者`finish()`，我们将看不到server端最后一行输出。

此时服务器进程模型：

```
pstree -ap | grep swoole
  |   |       `-php,3190 swoole_task.php
  |   |           |-php,3192 swoole_task.php
  |   |           |   |-php,3194 swoole_task.php
  |   |           |   `-php,3195 swoole_task.php
```

看到两个worker进程，其中一个是worker进程，另外一个是task_worker进程。

定时器

Swoole提供强大的异步毫秒定时器，基于timerfd+epoll实现。主要方法：

1、swoole_timer_tick：周期性定时器，类似于JavaScript里的`setInterval()`。

2、swoole_timer_after：一次性定时器。

3、swoole_timer_clear：清除定时器。

```
# 周期性定时器
int swoole_timer_tick(int $ms, callable $callback, mixed $user_param);

# 一次性定时器
swoole_timer_after(int $after_time_ms, mixed $callback_function, mixed $user_param);

# 清除定时器
bool swoole_timer_clear(int $timer_id)

# 定时器回调函数
function callbackFunction(int $timer_id, mixed $params = null);
```

注意：

- `$ms` 最大不得超过 86400000。
- manager进程中不能添加定时器。
- 建议在 `WorkerStart` 回调里写定时器。

定时器示例：

```
$server->on('WorkerStart', function (\swoole_server $server, $worker_id){
    if ($server->worker_id == 0){//防止重复
        //每隔2000ms触发一次
        swoole_timer_tick(2000, function ($timer_id) {
            echo "tick-2000ms\n";
        });

        //3000ms后执行此函数
        swoole_timer_after(3000, function () {
            echo "after 3000ms.\n";
        });
    }
});
```

WebSocket

使用Swoole可以很简单的搭建异步非阻塞多进程的WebSocket服务器。

WebSocket服务器

```
<?php
$server = new swoole_websocket_server("0.0.0.0", 9501);

$server->set(array(
        'daemonize' => false,
        'worker_num' => 2,
));

$server->on('Start', function (swoole_websocket_server $server) {
    echo "Server Start... \n";
    swoole_set_process_name("swoole_websocket_server");
});

$server->on('ManagerStart', function (swoole_websocket_server $server) {
    echo "ManagerStart\n";
});

$server->on('WorkerStart', function (swoole_websocket_server $server, $worker_id) {
    echo "WorkerStart \n";
    if ($server->worker_id == 0){
        swoole_timer_tick(10000,function($id) use ($server) {
            echo "test timer\n";
        });
    }
});

$server->on('Open', function (swoole_websocket_server $server, $request) {
    echo "server: handshake success with fd{$request->fd}\n";
});

$server->on('Message', function (swoole_websocket_server $server, $frame) {
    echo "receive from {$frame->fd}:{$frame->data},opcode:{$frame->opcode},fin:{$frame->finish}\n";
    $server->push($frame->fd, "this is server");
});

$server->on('Close', function ($ser, $fd) {
    echo "client {$fd} closed\n";
});

$server->start();
```

shell里直接运行 `php swoole_ws_server.php` 启动即可。如果设置了后台运行，可以使用下列命令强杀进程：

```
kill -9 $(ps aux|grep swoole|grep -v grep|awk '{print $2}')
```

或者重新启动worker进程：

```
kill -10 $(ps aux|grep swoole_websocket_server|grep -v grep|awk '{print $2}')
```

输出：

```
[2017-06-01 22:06:21 $2479.0]   NOTICE  Server is reloading now.
WorkerStart
WorkerStart
```

注意：

- onMessage回调函数为必选，当服务器收到来自客户端的数据帧时会回调此函数。

```
/**
* @param $server
* @param $frame 包含了客户端发来的数据帧信息；使用$frame->fd获取fd；$frame->data获取数据内容
*/
function onMessage(swoole_server $server, swoole_websocket_frame $frame)
```

- 使用 `$server->push()` 向客户端发送消息。长度最大不得超过2M。发送成功返回true，发送失败返回false。

```
function swoole_websocket_server->push(int $fd, string $data, int $opcode = 1, bool $finish = true);
```

WebSocket客户端

最简单的是使用JS编写：

```
socket = new WebSocket('ws://192.168.1.107:9501/');
socket.onopen = function(evt) {
    // 发送一个初始化消息
    socket.send('I am the client and I\'m listening!');
};

// 监听消息
socket.onmessage = function(event) {
    console.log('Client received a message', event);
};

// 监听Socket的关闭
socket.onclose = function(event) {
    console.log('Client notified socket has closed',event);
};

socket.onerror = function(evt) {
    console.log('Client onerror',event);
};
```

Swoole里没有直接提供swoole_websocket客户端，不过通过引入WebSocketClient.php文件可以实现：

```
<?php

require_once __DIR__ . '/WebSocketClient.php';

$client = new WebSocketClient('192.168.1.107', 9501);

if (!$client->connect())
{
        echo "connect failed \n";
        return false;
}

$send_data = "I am client.\n";
if (!$client->send($send_data))
{
        echo $send_data. " send failed \n";
        return false;
}

echo "send succ \n";
return true;
```

上面代码实现的是一个同步的swoole_websocket客户端。发送完消息会自动关闭，可以用来与php-fpm应用协作：将耗时任务使用客户端发送到swoole_websocket_server。

如何创建一个聊天室

实际项目里，我们可以将用户uid和fd进行双向绑定（暂不考虑多台服务器分布式部署情况），例如使用Redis保存：在onMessage进行用户信息验证后:

```
$this->redis->set($fd, $uid);
$this->redis->set($uid, $fd);
```

后续需要指定给某人发消息，只需要根据uid/fd发送即可。在onClose事件里进行解绑操作。群发的话只需要遍历一遍 `$server->connections` 即可。

示例（该项目只实现群发）：
moell-peng/webim: PHP + Swoole 实现的简单聊天室
https://github.com/moell-peng/webim

HttpServer

swoole内置Http服务器的支持。swoole版的http server相对于php-fpm，最大优势在于高性能：代码一次载入内存，后续无需再解释执行。缺点是调试没有nginx+php-fpm方便。

使用swoole，通过几行代码即可写出一个异步非阻塞多进程的Http服务器：

```
<?php
$serv = new swoole_http_server("0.0.0.0", 9502);

$serv->on('Start', function() {
        echo 'Start';
});

$serv->on('Request', function($request, $response) {
    var_dump($request->get);
    var_dump($request->post);
    var_dump($request->cookie);
    var_dump($request->files);
    var_dump($request->header);
    var_dump($request->server);

    $response->cookie("User", "Swoole");
    $response->header("X-Server", "Swoole");
    $response->end("<h1>Hello Swoole!</h1>");
});

$serv->start();
```

shell里使用 `php swoole_http_server.php` 运行server。浏览器打开http://192.168.1.107:9502/即可看到输出。

Process

Process是swoole内置的进程管理模块，用来替代PHP的pcntl扩展。

swoole_process支持重定向标准输入和输出，在子进程内echo不会打印屏幕，而是写入管道，读键盘输入可以重定向为管道读取数据。

配合swoole_event模块，创建的PHP子进程可以异步的事件驱动模式。

创建子进程

函数原型：

```
int swoole_process::__construct(mixed $function, $redirect_stdin_stdout = false, $create_pipe = true);
```

- `$function`，子进程创建成功后要执行的函数，底层会自动将函数保存到对象的callback属性上。
- 如果希望更改执行的函数，可赋值新的函数到对象的callback属性。
- `$redirect_stdin_stdout`，重定向子进程的标准输入和输出。启用此选项后，在进程内echo将不是打印屏幕，而是写入到管道。读取键盘输入将变为从管道中读取数据。默认为阻塞读取。
- `$create_pipe`，是否创建管道，启用`$redirect_stdin_stdout`后，此选项将忽略用户参数，强制为true 如果子进程内没有进程间通信，可以设置为false。
- 1.7.22或更高版本允许设置管道的类型，默认为`SOCK_STREAM`流式参数`$create_pipe`为2时，管道类型将设置为`SOCK_DGRAM`。

```
<?php
$process = new swoole_process(function(swoole_process $worker){
        while (true){
        $cmd = $worker->read();
        passthru($cmd);
    }
}, true, 2);
$process->start();

$process->write('ls -l');//将参数传入子进程内
echo $data = $process->read();//获取执行结果
```

实例：web版本的shell
https://github.com/52fhy/swoole_demo/blob/master/swoole_shell_server.php

EventLoop

swoole还提供了直接操作底层epoll/kqueue事件循环的接口。可将其他扩展创建的socket，PHP代码中stream/socket扩展创建的socket等加入到Swoole的EventLoop中。

swoole_event_add函数用于将一个socket加入到swoole的reactor事件监听中。 函数原型：

```
bool swoole_event_add(int $sock, mixed $read_callback, mixed $write_callback = null, int `$flags` = nul
l);
```

- `$sock` 支持文件描述符、stream资源、sockets资源。
- `$read_callback` 为可读回调函数。
- `$write_callback` 为可写事件回调。
- `$flags` 为事件类型的掩码，可选择关闭/开启可读可写事件，如 `SWOOLE_EVENT_READ` ，`SWOOLE_EVENT_WRITE` ，或者 `SWOOLE_EVENT_READ | SWOOLE_EVENT_WRITE` 。

在Server程序中使用，可以理解为在worker/taskworker进程中将此socket注册到epoll事件中。

在Client程序中使用，可以理解为在客户端进程中将此socket注册到epoll事件中。

示例：

```
<?php
$fp = stream_socket_client("tcp://www.52fhy.com:80", $errno, $errstr, 30);
fwrite($fp, "GET / HTTP/1.1\r\nHost: www.52fhy.com\r\n\r\n");

swoole_event_add($fp, function($fp) {

    echo $resp = fread($fp, 1024);
    //socket处理完成后，从epoll事件中移除socket
    swoole_event_del($fp);
    fclose($fp);
});
echo "Finish\n";  //swoole_event_add不会阻塞进程，这行代码会顺序执行
```

对比一下，下面这个会阻塞进程：

```
<?php
$fp  =  stream_socket_client ( "tcp://www.52fhy.com:80" ,  $errno ,  $errstr ,  30 );
if (! $fp ) {
    echo  " $errstr  ( $errno )<br />\n" ;
} else {
     fwrite ( $fp ,  "GET / HTTP/1.1\r\nHost: www.52fhy.com\r\n\r\n" );
    while (! feof ( $fp )) {
        echo  fgets ( $fp ,  1024 );
    }
     fclose ( $fp );
```

```
}

echo "Finish\n";
```

配置说明


```
$server->set(array(
        'daemonize' => true,
        'log_file' => '/www/log/swoole.log',
        'reactor_num' => 2,
        'worker_num' => 2,
        'task_worker_num' => 4,
        'max_request' => 100,

));
```

daemonize

设置是否后台运行。默认是false。设置 `daemonize => 1` 时，程序将转入后台作为守护进程运行。长时间运行的服务器端程序必须启用此项。

如果不启用守护进程，当ssh终端退出后，程序将被终止运行。

注意：

- 启用守护进程后，标准输入和输出会被重定向到 log_file。
- 如果未设置log_file，将重定向到 `/dev/null`，所有打印屏幕的信息都会被丢弃。

log_file

指定swoole错误日志文件。在swoole运行期发生的异常信息会记录到这个文件中。默认会打印到屏幕。

注意log_file不会自动切分文件，所以需要定期清理此文件。通过重新打开日志，可以实现按天记录日

志。

log_level

设置swoole_server错误日志打印的等级，范围是0-5。低于log_level设置的日志信息不会抛出。默认是0也就是所有级别都打印。

```
0 =>DEBUG
1 =>TRACE
2 =>INFO
3 =>NOTICE
4 =>WARNING
5 =>ERROR
```

reactor_num

reactor线程数，通过此参数来调节主进程内事件处理线程的数量，以充分利用多核。默认会启用CPU核数相同的数量。`reactor_num`一般设置为CPU核数的1-4倍，在swoole中`reactor_num`最大不得超过`CPU核数*4`。

Swoole的主进程是一个多线程的程序。其中有一组很重要的线程，称之为Reactor线程。它就是真正处理TCP连接，收发数据的线程。Reactor线程进行协议解析，将请求投递到Worker进程。

推荐配置：CPU核数。

注意：`reactor_num`必须小于或等于`worker_num`。如果设置的`reactor_num`大于`worker_num`，那么swoole会自动调整使`reactor_num`等于`worker_num`。

worker_num

设置启动的worker进程数。`worker_num >= 1`，至少会有一个，默认是 1。设置方法：

- 业务代码是全异步非阻塞的，这里设置为CPU的1-4倍最合理。
- 业务代码为同步阻塞，需要根据请求响应时间和系统负载来调整。

比如1个请求耗时100ms，要提供1000QPS的处理能力，那必须配置100个进程或更多。但是需要考虑内存占用。假设每个进程占用40M内存，那100个进程就需要占用4G内存。

推荐配置：CPU核数*2。

task_worker_num

配置task进程的数量，配置此参数后将会启用task功能。如果业务用不到异步任务，可以不设置。一旦设置次参数，必须设置`onTask/onFinish`2个事件回调。

注意：

- task进程内不能使用`swoole_server->task`方法
- task进程内不能使用`mysql-async/redis-async/swoole_event`等异步IO函数

推荐配置：根据实际异步任务比重设置。

dispatch_mode

数据包分发策略。可以选择3种类型，默认为2。一般情况下，HttpServer可以使用1、3；WebSocketServer可以使用默认的2。

- 1，轮循模式，收到会轮循分配给每一个worker进程。
- 2，固定模式，根据连接的文件描述符分配worker。这样可以保证同一个连接发来的数据只会被同一个worker处理。
- 3，抢占模式，主进程会根据Worker的忙闲状态选择投递，只会投递给处于闲置状态的Worker。
- 4，IP分配，根据客户端IP进行取模hash，分配给一个固定的worker进程。可以保证同一个来源IP的连接数据总会被分配到同一个worker进程。算法为 `ip2long(ClientIP) % worker_num`。
- 5，UID分配，需要用户代码中调用 `$serv->bind()` 将一个连接绑定1个uid。然后swoole根据UID的值分配到不同的worker进程。算法为 `UID % worker_num`，如果需要使用字符串作为UID，可以使用 `crc32(UID_STRING)`。

`dispatch_mode` 配置在BASE模式是无效的，因为BASE不存在投递任务。

max_request

设置worker进程的最大任务数，默认为0。

这个参数的主要作用是解决PHP进程内存溢出问题。一个worker进程在处理完超过max_request数值的任务后将自动退出，进程退出后会释放所有内存和资源。

例如设置为3，假设有2个worker进程，执行5次请求，必然会有一个worker进程会退出并被重新拉起一个新的。如果不设置，就不会执行这个操作。

PHP应用程序有缓慢的内存泄漏，但无法定位到具体原因、无法解决，可以通过设置max_request解决。

如果代码没有内存泄露的问题，没有每访问一次，内存就增加一点，那不设置(默认为0)也不会有内存泄露。反之，max_request，就可以限制内存无限制增长,从而防止内存泄露。(参考http://group.swoole.com/question/106049)

task_max_request

类似于max_request，默认为5000。用于设置task进程的最大任务数。一个task进程在处理完超过此数值的任务后将自动退出。这个参数是为了防止PHP进程内存溢出。如果不希望进程自动退出可以设置为0。

max_conn (max_connection)

服务器程序，最大允许的连接数，如max_conn => 10000, 此参数用来设置Server最大允许维持多少个tcp连接。超过此数量后，新进入的连接将被拒绝。

- 默认值为 `ulimit -n` 的值。系统的 `ulimit -n` 可能太小，需要手动设置。例如 `ulimit -n 65535`。

- 最大不得超过操作系统 `ulimit -n` 的值，否则会报一条警告信息，并重置为 `ulimit -n` 的值。
- 此参数不要调整的过大，根据机器内存的实际情况来设置。

heartbeat_idle_time、heartbeat_check_interval

heartbeat_idle_time与heartbeat_check_interval配合使用。表示连接最大允许空闲的时间。如：

```
array(
    'heartbeat_idle_time' => 600, //表示连接最大允许空闲的时间
    'heartbeat_check_interval' => 60, //表示每隔少秒轮循一次
);
```

表示每60秒遍历一次，一个连接如果600秒内未向服务器发送任何数据，此连接将被强制关闭。

- 启用heartbeat_idle_time后，服务器并不会主动向客户端发送数据包，而是被动等待客户端发送心跳。
- 如果只设置了heartbeat_idle_time未设置heartbeat_check_interval底层将不会创建心跳检测线程，PHP代码中可以调用heartbeat方法手工处理超时的连接。

open_eof_check、package_eof

设置EOF字符串。package_eof最大只允许传入8个字节的字符串。

```
array(
    'open_eof_check' => true, //打开EOF检测
    'package_eof' => "\r\n", //设置EOF
)
```

打开EOF检测，此选项将检测客户端连接发来的数据，当数据包结尾是指定的字符串时才会投递给Worker进程。否则会一直拼接数据包，直到超过缓存区或者超时才会中止。当出错时swoole底层会认为是恶意连接，丢弃数据并强制关闭连接。

常见的Memcache/SMTP/POP等协议都是以\r\n结束的，就可以使用此配置。开启后可以保证Worker进程一次性总是收到一个或者多个完整的数据包。

open_eof_split

EOF检测不会从数据中间查找eof字符串，所以Worker进程可能会同时收到多个数据包，需要在应用层代码中自行 `explode("\r\n", $data)` 来拆分数据包。

1.7.15版本增加了open_eof_split，支持从数据中查找EOF，并切分数据。

pid_file

在Server启动时自动将master进程的PID写入到文件，在Server关闭时自动删除PID文件。（1.9.5+支持）

```
$server->set(array(
    'pid_file' => __DIR__.'/server.pid',
));
```

其它参数详见：配置选项-Swoole-Swoole文档中心
https://wiki.swoole.com/wiki/page/274.html

服务器调优

常见优化参数：

```
# 内核优化
ulimit -n 65535
net.core.somaxconn = 262144
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_syncookies = 1

# nginx优化
worker_rlimit_nofile 65535;
use epoll;
worker_connections 65535;
keepalive_timeout 60;
```

内核优化如何修改：

1、ulimit参数直接运行`ulimit -SHn 65535`即可，可以使用`ulimit -n`查看；

2、其它参数优化示例：

`net.core.somaxconn`对应文件`/proc/sys/net/core/somaxconn`：

```
# 查看
cat /proc/sys/net/core/somaxconn

# 修改
echo 65535 > /proc/sys/net/core/somaxconn
```

`net.ipv4.tcp_tw_recycle`对应文件`/proc/sys/net/ipv4/tcp_tw_recycle`：

```
# 查看
cat /proc/sys/net/ipv4/tcp_tw_recycle

# 修改
echo 65535 > /proc/sys/net/ipv4/tcp_tw_recycle
```

3、也可以修改`/etc/sysctl.conf`文件：

```
# 调高系统的 IP 以及端口数据限制，从可以接受更多的连接
net.ipv4.ip_local_port_range = 2000 65000

net.ipv4.tcp_window_scaling = 1

# 设置协议栈可以缓存的报文数阀值，超过阀值的报文将被内核丢弃
net.ipv4.tcp_max_syn_backlog = 3240000

# 调高 socket 侦听数阀值
net.core.somaxconn = 3240000
```

```
net.ipv4.tcp_max_tw_buckets = 1440000

# 调大 TCP 存储大小
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
net.ipv4.tcp_congestion_control = cubic

# tcp重新回收
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_tw_reuse = 1
```

修改配置之后需要执行以下命令使之生效：

```
sysctl -p /etc/sysctl.conf
```

相关应用

1、https://github.com/matyhtf/framework
2、衍生开源项目-Swoole-Swoole文档中心
https://wiki.swoole.com/wiki/page/p-open_source.html
3、moell-peng/webim: PHP + Swoole 实现的简单聊天室
https://github.com/moell-peng/webim

参考

1、Server-Swoole-Swoole文档中心
https://wiki.swoole.com/wiki/page/p-server.html
2、swoole_study/Swoole的进程模型.md at master · szyhf/swoole_study
https://github.com/szyhf/swoole_study/blob/master/Swoole的进程模型.md

关于作者

- GitHub: https://github.com/52fhy/
- Website：http://52fhy.com/
- Email：yjc@52fhy.com

更新于：2017-8-22 21:44:41