Table of Contents

Concise Guide to Swoole

Author

Lancelot（李丹阳 ID: 会敲代码的喵） from **LinkedDestiny Studio**（牵机工作室）
Company：Bilibili

Description

A concise guide to php swoole extension.

Contact

simonarthur2012@gmail.com (Lancelot)

Notice

I'm sorry that I can't translate my book to English. If someone wants to help me, Please send an email to me, thanks

写在前面的话

本书默认读者已具备如下能力：

- 熟练使用PHP语言
- 熟练使用MySQL、Redis数据库
- 熟练使用Linux操作系统
- 基本了解Unix网络编程相关知识（参阅《Unix网络编程（卷1）》）
- 基本的gdb使用

第一章将讲解如何配置PHP&Swoole的开发环境，会一步步列出安装所需的依赖和命令。

第二章将讲解Swoole的基本功能和配置选项，包括Worker进程、Task Worker进程、Timer计时器、Process进程、swoole_table内存表等，也会讲解这些功能的基本使用方法。

第三章将讲解Swoole的内置协议部分，讲解如何自定义TCP的应用层通信协议。同时也会介绍Swoole内置的多种协议解析方式，比如Http服务器、WebSocket服务器等等。

第四章将讲解Swoole Client的相关内容，讲解如何创建和使用Swoole提供的多种Client，如TCP Client、异步Http Client、异步MySQL Client等。

第五章将讲解Swoole的异步IO部分，包括异步文件读写和异步EventLoop事件循环。

第六章将讲解Swoole的一些实战用法，比如使用Task进程进行异步任务处理、使用Process执行监控命令等

第七章将讲解Swoole的一些相关框架，比如ZPHP，Hprose，Dora-rpc等等

第八章将讲解Swoole与一些现有框架的结合，比如Swoole-Yaf，Swoole-Phalcon等

第九章开始将讲解Swoole实战，通过一些实际项目来深入了解Swoole的应用。（构思中）

环境搭建

本章将详细介绍如何搭建Swoole的开发环境，包括PHP的安装、Swoole的安装、相关扩展的安装等等。 通过学习本章，你可以学到如何快速搭建一套完整的Swoole开发环境。

相关软件

类别	名称	版本
操作系统（Linux）	Ubuntu	16.04
操作系统（Mac）	OSX	10.11
语言	PHP	5.6.22
扩展	Swoole	1.8.5-stable
扩展	Redis	3.0.0
数据库	MySQL	5.7.12
数据库	Redis	3.0.7

相关框架

类别	名称	版本
MVC	ZPHP	Master分支
MVC	Yaf	Master分支
RPC	Hprose	Master分支
PRC	Dora-RPC	Master分支
DB Model	ThinkPHP	3.2.2 (仅使用Model模块)

环境搭建

[TOC]

Linux环境下安装

Linux操作系统通常都有自己的包管理软件（Ubuntu的apt-get，CentOS的yum，Mac OSX的HomeBrew等），因此一般情况下可以通过这些包管理软件直接安装PHP。但是这样安装的PHP不太适用于运行Swoole，因此本章将介绍如何通过源码编译安装。

编译环境

想要编译安装PHP首先需要安装对应的编译工具。 Ubuntu上使用如下命令安装编译工具和依赖包：

```
sudo apt-get install \
build-essential \
gcc \
g++ \
autoconf \
libiconv-hook-dev \
libmcrypt-dev \
libxml2-dev \
libmysqlclient-dev \
libcurl4-openssl-dev \
libjpeg8-dev \
libpng12-dev \
libfreetype6-dev \
```

PHP安装

PHP下载地址 在这里挑选你想用的版本即可。下载源码包后，解压至本地任意目录（保证读写权限）。

使用如下命令编译安装PHP：

```
cd php-5.6.22/
./configure --prefix=/usr/local/php \
--with-config-file-path=/etc/php \
--enable-fpm \
--enable-pcntl \
--enable-mysqlnd \
--enable-opcache \
--enable-sockets \
--enable-sysvmsg \
--enable-sysvsem \
--enable-sysvshm \
--enable-shmop \
--enable-zip \
--enable-soap \
--enable-xml \
--enable-mbstring \
--disable-rpath \
--disable-debug \
--disable-fileinfo \
--with-mysql=mysqlnd \
--with-mysqli=mysqlnd \
--with-pdo-mysql=mysqlnd \
--with-pcre-regex \
--with-iconv \
--with-zlib \
--with-mcrypt \
--with-gd \
--with-openssl \
--with-mhash \
--with-xmlrpc \
--with-curl \
--with-imap-ssl

sudo make
sudo make install
sudo mkdir /etc/php
sudo cp php.ini-development /etc/php/php.ini
```

注意，以上PHP编译选项根据实际情况可调整。

另外，还需要将PHP的可执行目录添加到环境变量中。 使用Vim/Sublime打开~/.bashrc，在末尾添加如下内容：

```
export PATH=/usr/local/php/bin:$PATH
export PATH=/usr/local/php/sbin:$PATH
```

保存后，终端输入命令：

```
source ~/.bashrc
```

此时即可通过 `php --version` 查看php版本。

Mac环境下安装

Mac系统自带PHP，但是Mac上对于OpenSSL的相关功能做了一些限制，使用了一个 `Secure Transport` 来取代OpenSSL。因此仍然建议重新编译安装PHP环境。

安装OpenSSL

Mac原装的0.9.8版本的OpenSSL使用的时候会有些Warning，反正我看不惯……

安装命令：

```
brew install openssl
```

安装之后，还需要链接新的openssl到环境变量中。

```
brew link --force openssl
```

安装Curl

Mac系统原装的Curl默认使用了Secure Transport，导致通过option函数设置的证书全部无效。果断重新安装之。

```
brew install curl --with-openssl && brew link curl --force
```

安装PHP

PHP官网上下载某个版本的PHP（我选择的是5.6.22），使用如下命令编译安装。

```
cd /path/to/php/
./configure
--prefix=/usr/local/php
--with-config-file-path=/etc/php
--with-openssl=/usr/local/Cellar/openssl/1.0.2g/
--with-curl=/usr/local/Cellar/curl/7.48.0/

make && make install
```

这里我仅列出两个需要特殊设置的选项 `with-openssl` 和 `with-curl` 。 安装完成后，执行如下命令：

```
sudo cp /usr/local/php/bin/php /usr/bin/
sudo cp /usr/local/php/bin/phar* /usr/bin/
sudo cp /usr/local/php/bin/php-config /usr/bin/
sudo cp /usr/local/php/bin/phpize /usr/bin/
```

随后，设置php.ini

```
sudo mkdir /etc/php
sudo cp php.ini.development /etc/php/php.ini
```

Swoole扩展安装

Swoole扩展下载地址 解压源码至任意目录，执行如下命令：

```
cd swoole-src-swoole-1.7.6-stable/
phpize
./configure
sudo make
sudo make install
```

> swoole的./configure有很多额外参数，可以通过./configure --help命令查看,这里均选择默认项)

安装完成后，进入/etc/php目录下，打开php.ini文件，在其中加上如下一句：

```
extension=swoole.so
```

随后在终端中输入命令 `php -m` 查看扩展安装情况。如果在列出的扩展中看到了swoole，则说明安装成功。

搭建Echo服务器

[TOC]

所有讲解网络通信编程的书籍都会最先讲解如何编写一个Echo服务器，本书也不例外。本章将讲解如何快速编写一个基于Swoole扩展的Echo服务器。

服务端

创建一个 `Server.php` 文件并输入如下内容：

```
// Server
class Server
{
    private $serv;

    public function __construct() {
        $this->serv = new swoole_server("0.0.0.0", 9501);
        $this->serv->set(array(
            'worker_num' => 8,
            'daemonize' => false,
        ));

        $this->serv->on('Start', array($this, 'onStart'));
        $this->serv->on('Connect', array($this, 'onConnect'));
        $this->serv->on('Receive', array($this, 'onReceive'));
        $this->serv->on('Close', array($this, 'onClose'));

        $this->serv->start();
    }

    public function onStart( $serv ) {
        echo "Start\n";
    }

    public function onConnect( $serv, $fd, $from_id ) {
        $serv->send( $fd, "Hello {$fd}!" );
    }

    public function onReceive( swoole_server $serv, $fd, $from_id, $data ) {
        echo "Get Message From Client {$fd}:{$data}\n";
        $serv->send($fd, $data);
    }

    public function onClose( $serv, $fd, $from_id ) {
        echo "Client {$fd} close connection\n";
    }
}
// 启动服务器
$server = new Server();
```

客户端

创建一个`Client.php`文件并输入如下内容：

```php
<?php
class Client
{
    private $client;

    public function __construct() {
        $this->client = new swoole_client(SWOOLE_SOCK_TCP);
    }

    public function connect() {
        if( !$this->client->connect("127.0.0.1", 9501 , 1) ) {
            echo "Error: {$this->client->errMsg}[{$this->client->errCode}]\n";
        }

        fwrite(STDOUT, "请输入消息：");
        $msg = trim(fgets(STDIN));
        $this->client->send( $msg );

        $message = $this->client->recv();
        echo "Get Message From Server:{$message}\n";
    }
}

$client = new Client();
$client->connect();
```

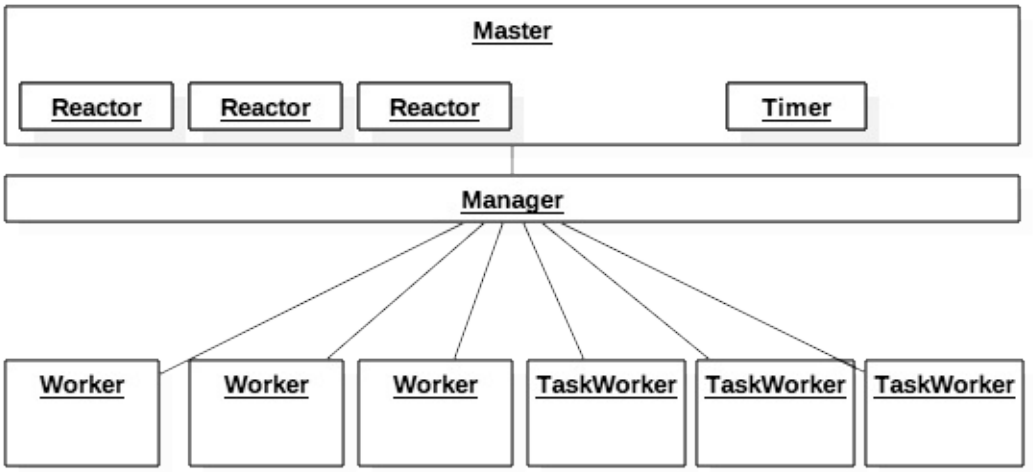
运行

在Terminal下执行命令 `php Server.php` 即可启动服务器，在另一个Terminal下执行 `php Client.php` ，输入要发送的内容，即可发送消息到服务器，并收到来自服务器的消息。

Worker进程

Swoole进程模型

首先，我们需要了解一下Swoole的进程模型。Swoole是一个多进程模式的框架（可以类比Nginx的进程模型），当启动一个Swoole应用时，一共会创建2 + n + m个进程，其中n为Worker进程数，m为TaskWorker进程数，2为一个Master进程和一个Manager进程，它们之间的关系如下图所示。



其中，Master进程为主进程，该进程会创建Manager进程、Reactor线程等工作进/线程。

- Reactor线程实际运行epoll实例，用于accept客户端连接以及接收客户端数据；
- Manager进程为管理进程，该进程的作用是创建、管理所有的Worker进程和TaskWorker进程。

Worker进程简介

Worker进程作为Swoole的工作进程，所有的业务逻辑代码均在此进程上运行。当Reactor线程接收到来自客户端的数据后，会将数据打包通过管道发送给某个Worker进程（数据分配方法见dispatch_mode）。

Worker进程生命周期

一个Worker进程的生命周期如图所示：

```
st=>start: Create
start=>operation: onWorkerStart
recv=>operation: onReceive/onConnect/onClose
op=>operation: Receive and Handle Data
cond=>condition: Max Request or Error
stop=>operation: onWorkerStop
e=>end: Stop

st->start->recv->op->cond
cond(yes)->stop
cond(no)->recv
stop->e
```

当一个Worker进程被成功创建后，会调用 `onWorkerStart` 回调，随后进入事件循环等待数据。当通过回调函数接收到数据后，开始处理数据。如果处理数据过程中出现严重错误导致进程退出，或者Worker进程处理的总请求数达到指定上限，则Worker进程调用 `onWorkerStop` 回调并结束进程。

[1]:

Task Worker

简介

Task Worker是Swoole中一种特殊的工作进程，该进程的作用是处理一些耗时较长的任务，以达到释放Worker进程的目的。Worker进程可以通过 `swoole_server` 对象的task方法投递一个任务到Task Worker进程，其流程如下所示：

```
Worker->Task Worker: task()
Note right of Task Worker: onTask()
Task Worker-->Worker: finish()
Note left of Worker: onFinish()
```

"'sequence Title: Here is a title A->B: Normal line B-->C: Dashed line C->>D: Open arrow D-->>A: Dashed open arrow "'

Worker进程通过Unix Sock管道将数据发送给Task Worker，这样Worker进程就可以继续处理新的逻辑，无需等待耗时任务的执行。需要注意的是，由于Task Worker是独立进程，因此无法直接在两个进程之间共享全局变量，需要使用Redis、MySQL或者swoole_table来实现进程间共享数据。

实例

要使用Task Worker，需要进行一些必要的操作。

首先，需要设置swoole_server的配置参数：

```
$serv->set(array(
    'task_worker_num' => 2, // 设置启动2个task进程
));
```

接着，绑定必要的回调函数：

```
$serv->on('Task', 'onTask');
$serv->on('Finish','onFinish');
```

其中两个回调函数的原型如下所示：

```
/**
 * @param $serv swoole_server swoole_server对象
 * @param $task_id int 任务id
 * @param $from_id int 投递任务的worker_id
 * @param $data string 投递的数据
 */
function onTask(swoole_server $serv, $task_id, $from_id, $data);

/**
 * @param $serv swoole_server swoole_server对象
 * @param $task_id int 任务id
 * @param $data string 任务返回的数据
 */
function onFinish(swoole_server $serv, $task_id, $data);
```

在实际逻辑中，当需要发起一个任务请求时，可以使用如下方法调用：

```
$data = "task data";
$serv->task($data , -1 ); // -1代表不指定task进程

// 在1.8.6+的版本中，可以动态指定onFinish函数
$serv->task($data, -1, function (swoole_server $serv, $task_id, $data) {
    echo "Task Finish Callback\n";
});
```

Timer定时器

定时器原理

Timer定时器是Swoole扩展提供的一个毫秒级定时器，其作用是每隔指定的时间间隔之后执行一次指定的回调函数，以实现定时执行任务的功能。 新版本的Swoole中，定时器是基于epoll方法的timeout机制实现的，不再依赖于单独的定时线程，准确度更高。同时，Swoole扩展使用最小堆存储定时器，减少定时器的检索次数，提高了运行效率。

定时器使用

在Swoole中，定时器的函数原型如下：

```
// function onTimer(int $timer_id, mixed $params = null); // 回调函数的原型
int swoole_timer_tick(int $ms, mixed $callback, mixed $param = null);
int swoole_server::(int $ms, mixed $callback, mixed $param = null);

// function onTimer(); // 回调函数的原型（不接受任何参数）
void swoole_timer_after(int $after_time_ms, mixed $callback_function);
void swoole_server::after(int $after_time_ms, mixed $callback_function);
```

tick定时器是一个永久定时器，使用tick方法创建的定时器会一直运行，每隔指定的毫秒数之后执行一次callback函数。在创建定时器的时候，可以通过tick函数的第三个参数，传递一些自定义参数到callback回调函数中。另外，也可以使用PHP的闭包（use关键字）实现传参。具体实例如下：

```
$str = "Say ";
$timer_id = swoole_timer_tick( 1000 , function($timer_id , $params) use ($str) {
    echo $str . $params;  // 输出“Say Hello”

} , "Hello" );
```

tick函数会返回定时器的id。当我们不再需要某个定时器的时候，可以根据这个id，调用 `swoole_timer_clear` 函数删除定时器。需要注意的是，创建的定时器是不能跨进程的，因此，在一个Worker进程中创建的定时器，也只能在这个Worker进程中删除，这一点一定要注意（使用 `$worker_id` 变量来区分Worker进程）；

after定时器是一个临时定时器。使用after方法创建的定时器仅在指定毫秒数之后执行一次callback函数，执行完成后定时器就会删除。after定时器的回调函数不接受任何参数，可以通过闭包方式传递参数，也可以通过类成员变量的方式传递。具体实例如下：

```
class Test
{
    private $str = "Say Hello";
    public function onAfter()
    {
        echo $this->str; // 输出”Say Hello“
    }
}

$test = new Test();
swoole_timer_after(1000, array($test, "onAfter"); // 成员变量

swoole_timer_after(2000, function() use($test){ // 闭包
    $test->onAfter(); // 输出”Say Hello“
});
```

多端口监听

多端口监听

在实际运用场景中，服务器可能需要监听不同host下的不同端口。比如，一个应用服务器，可能需要监听外网的服务端口，同时也需要监听内网的管理端口。在Swoole中，可以轻松的实现这样的功能。 Swoole提供了addlistener函数用于给服务器添加一个需要监听的host及port，并指定对应的Socket类型（TCP，UDP，Unix Socket以及对应的IPv4和IPv6版本）。代码如下：

```
$serv = new swoole_server("192.168.1.1", 9501); // 监听外网的9501端口
$serv->addlistener("127.0.0.1", 9502 , SWOOLE_TCP); // 监听本地的9502端口
$serv->start(); // addlistener必须在start前调用
```

此时，swoole_server就会同时监听两个host下的两个端口。这里要注意的是，来自两个端口的数据会在同一个 `onReceive` 回调函数中获取到，这时就要用到swoole的另一个成员函数connection_info，通过这个函数获取到fd的from_port，就可以判定消息的类型。

```
$info = $serv->connection_info($fd, $from_id);
//来自9502的内网管理端口
if($info['from_port'] == 9502) {
    $serv->send($fd, "welcome admin\n");
}
//来自外网
else {
    $serv->send($fd, 'Swoole: '.$data);
}
```

多端口混合协议接听

通过上面的实例可以看到，使用上面的方法进行多端口监听有诸多的局限性：协议单一，回调函数无法区分等。在实际应用中，我们往往希望服务能够同时监听两个端口，并且两个端口分别采用不同的协议，比如一个端口采用RPC协议提供服务，另一个端口提供Http协议用于Web管理页面。 因此，Swoole从1.8.0版本开始提供了一套全新的多端口监听方式。在1.8.0以后的版本，Server可以监听多个端口，每个端口都可以设置不同的协议处理方式(set)和回调函数(on) 开始监听新端口的代码如下：

```
$port1 = $server->listen("127.0.0.1", 9501, SWOOLE_SOCK_TCP);
$port2 = $server->listen("127.0.0.1", 9502, SWOOLE_SOCK_UDP);
$port3 = $server->listen("127.0.0.1", 9503, SWOOLE_SOCK_TCP | SWOOLE_SSL);
```

可以看到，新添加的监听端口可以设置多种属性，监听的IP，端口号，TCP或者UDP，是否需要SSL加密。 除此之外，每个新建立的Port对象还可以分别设置配置选项，如下所示：

```
$port1->set(    // 开启固定包头协议
    'open_length_check' => true,
    'package_length_type' => 'N',
    'package_length_offset' => 0,
    'package_max_length' => 800000,
);

$port3->set( // 开启EOF协议并设置SSL文件
    'open_eof_split' => true,
    'package_eof' => "\r\n",
    'ssl_cert_file' => 'ssl.cert',
    'ssl_key_file' => 'ssl.key',
);
```

除了协议不同，每个Port还可以设置自己独有的回调函数，由此避免了在同一个回调函数里针对数据来源进行判定的问题。

```
$port1->on('receive', function ($serv, $fd, $from_id, $data) {
    $serv->send($fd, 'Swoole: '.$data);
    $serv->close($fd);
});
$port3->on('receive', function ($serv, $fd, $from_id, $data) {
    echo "Hello {$fd} : {$data}\n";
});
```

注意事项

- 未设置协议处理选项的监听端口，默认使用无协议模式
- 未设置回调函数的监听端口，使用$server对象的回调函数
- 监听端口返回的对象类型为swoole_server_port
- 不同监听端口的回调函数，仍然是相同的Worker进程空间内执行
- 主服务器是WebSocket或Http协议，新监听的TCP端口默认会继承主Server的协议设置。必须单独调用 `set` 方法设置新的协议才会启用新协议

SSL支持

本章将详细讲解如何制作证书以及如何开启Swoole的SSL的单向、双向认证。

准备工作

选择任意路径，执行如下命令创建文件夹结构

```
mkdir ca
cd ca
mkdir private
mkdir server
mkdir newcerts
```

在ca目录下创建 `openssl.conf` 文件，文件内容如下

```
[ ca ]
default_ca      = foo                   # The default ca section

[ foo ]
dir             = /path/to/ca         # top dir
database        = /path/to/ca/index.txt          # index file.
new_certs_dir   = /path/to/ca/newcerts           # new certs dir

certificate     = /path/to/ca/private/ca.crt         # The CA cert
serial          = /path/to/ca/serial             # serial no file
private_key     = /path/to/ca/private/ca.key  # CA private key
RANDFILE        = /path/to/ca/private/.rand      # random number file

default_days    = 365                     # how long to certify for
default_crl_days= 30                      # how long before next CRL
default_md      = md5                     # message digest method to use
unique_subject = no                       # Set to 'no' to allow creation of
                                          # several ctificates with same subject.
policy          = policy_any              # default policy

[ policy_any ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = match
localityName            = optional
commonName              = optional
emailAddress            = optional
```

其中，`/path/to/ca/` 是ca目录的绝对路径。

创建ca证书

在ca目录下创建一个shell脚本，命名为 `new_ca.sh` 。文件内容如下：

```
#!/bin/sh
openssl genrsa -out private/ca.key
openssl req -new -key private/ca.key -out private/ca.csr
openssl x509 -req -days 365 -in private/ca.csr -signkey private/ca.key -out private/ca
.crt
echo FACE > serial
touch index.txt
openssl ca -gencrl -out private/ca.crl -crldays 7 -config "./openssl.conf"
```

执行 `sh new_ca.sh` 命令，创建ca证书。生成的证书存放于private目录中。

> 注意 在创建ca证书的过程中，需要输入一些信息。其中，countryName、stateOrProvinceName、organizationName、organizationalUnitName这四个选项的内容必须要填写，并且需要记住。在生成后续的证书过程中，要保证这四个选项的内容一致。

创建服务端证书

在ca目录下创建一个shell脚本，命名为 `new_server.sh` 。文件内容如下：

```
#!/bin/sh
openssl genrsa -out server/server.key
openssl req -new -key server/server.key -out server/server.csr
openssl ca -in server/server.csr -cert private/ca.crt -keyfile private/ca.key -out ser
ver/server.crt -config "./openssl.conf"
```

执行 `sh new_ca.sh` 命令，创建ca证书。生成的证书存放于server目录中。

创建客户端证书

在ca目录下创建一个shell脚本，命名为 `new_client.sh` 。文件内容如下：

```
#!/bin/sh

base="./"
mkdir -p $base/users/
openssl genrsa -des3 -out $base/users/client.key 1024
openssl req -new -key $base/users/client.key -out $base/users/client.csr
openssl ca -in $base/users/client.csr -cert $base/private/ca.crt -keyfile $base/privat
e/ca.key -out $base/users/client.crt -config "./openssl.conf"
openssl pkcs12 -export -clcerts -in $base/users/client.crt -inkey $base/users/client.k
ey -out $base/users/client.p12
```

执行 `sh new_ca.sh` 命令，创建ca证书。生成的证书存放于users目录中。 进入users目录，可以看到有一个 `client.p12` 文件，这个就是客户端可用的证书了，但是这个证书是不能在php中使用的，因此需要做一次转换。命令如下：

```
openssl pkcs12 -clcerts -nokeys -out cer.pem -in client.p12
openssl pkcs12 -nocerts -out key.pem -in client.p12
```

以上两个命令会生成cer.pem和key.pem两个文件。其中，生成key.pem时会要求设置密码，这里记为 `client_pwd`

> 注意 如果在创建客户端证书时，就已经给client.p12设置了密码，那么在转换格式的时候，需要输入密码进行转换

最终结果

以上步骤执行结束后，会得到不少文件，其中需要用的文件如下表所示：| 文件名 | 路径 | 说明 | | --- | --- | --- | | ca.crt | ca/private/ | ca证书 | | server.crt | ca/server/ | 服务器端证书 | | server.key | ca/server/ | 服务器端秘钥 | | cer.pem | ca/client/ | 客户端证书 | | key.pem | ca/client/ | 客户端秘钥 |

SSL单向认证

Swoole开启SSL

Swoole开启SSL功能需要如下参数：

```
$server = new swoole_server("127.0.0.1", "9501" , SWOOLE_PROCESS, SWOOLE_SOCK_TCP | SW
OOLE_SSL );
$server = new swoole_http_server("127.0.0.1", "9501" , SWOOLE_PROCESS, SWOOLE_SOCK_TCP
 | SWOOLE_SSL );
```

并在swoole的配置选项中增加如下两个选项：

```
$server->set(array(
    'ssl_cert_file' => '/path/to/server.crt',
    'ssl_key_file' =>  '/path/to/server.key',
));
```

这时，swoole服务器就已经开启了单向SSL认证，可以通过 `https://127.0.0.1:9501/` 进行访问。

SSL双向认证

服务器端设置

双向认证指服务器也要对发起请求的客户端进行认证，只有通过认证的客户端才能进行访问。 为了开启SSL双向认证，swoole需要额外的配置参数如下：

```
$server->set(array(
    'ssl_cert_file' => '/path/to/server.crt',
    'ssl_key_file' =>  '/path/to/server.key',
    'ssl_client_cert_file' => '/path/to/ca.crt',
    'ssl_verify_depth' => 10,
));
```

客户端设置

这里我们使用CURL进行https请求的发起。 首先，需要配置php.ini,增加如下配置：

```
curl.cainfo=/path/to/ca.crt
```

发起curl请求时，增加如下配置项：

```
$ch = curl_init();

curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, '2');
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true);   // 只信任CA颁布的证书
curl_setopt($ch, CURLOPT_SSLCERT, "/path/to/cer.pem");
curl_setopt($ch, CURLOPT_SSLKEY,  "/path/to/key.pem");
curl_setopt($ch, CURLOPT_SSLCERTTYPE, 'PEM');
curl_setopt($ch, CURLOPT_SSLCERTPASSWD, '******'); // 创建客户端证书时标记的client_pwd密码
```

这时，就可以发起一次https请求，并且被swoole服务器验证通过了。

配置选项

[TOC]

在swoole中，一个swoole_server的相关属性可以通过

```
$serv->set( array $configs );
```

函数来配置，这些配置选项使得swoole更加灵活。 示例：

```
$serv = new swoole_server("0.0.0.0", 9501);
$serv->set(array(
    'worker_num' => 8,
    'max_request' => 10000,
    'max_conn' => 100000,
    'dispatch_mode' => 2,
    'debug_mode'=> 1，
    'daemonize' => false,
));
```

配置选项以及相关介绍如下：

reactor_num

描述：指定Reactor线程数
说明：设置主进程内事件处理线程的数量，默认会启用CPU核数相同的数量， 一般设置为CPU核数的**1-4**倍，最大不得超过CPU核数*4。
示例：

```
'reactor_num' => 8
```

worker_num

描述：指定启动的worker进程数 说明：开启的worker进程数越多，server负载能力越大，但是相应的server占有的内存也会更多。同时，当worker进程数过多时，进程间切换带来的系统开销也会更大。因此建议开启的worker进程数为cpu核数的**1-4**倍。 示例：

```
'worker_num' => 8
```

max_request

描述：每个worker进程允许处理的最大任务数。
说明：设置该值后，每个worker进程在处理完max_request个请求后就会自动重启。设置该值的主要目的是为了防止worker进程处理大量请求后可能引起的内存溢出。
示例：

```
'max_request' => 10000
```

max_conn

描述：服务器允许维持的最大TCP连接数
说明：设置此参数后，当服务器已有的连接数达到该值时，新的连接会被拒绝。另外，该参数的值不能超过操作系统**ulimit -n**的值，同时此值也不宜设置过大，因为swoole_server会一次性申请一大块内存用于存放每一个connection的信息。
示例：

```
'max_conn' => 10000
```

ipc_mode

描述：设置进程间的通信方式。
说明：共有三种通信方式，参数如下：

> - 1 => 使用unix socket通信
> - 2 => 使用消息队列通信
> - 3 => 使用消息队列通信，并设置为争抢模式

示例：

```
'ipc_mode' => 1
```

dispatch_mode

描述：指定数据包分发策略。
说明：共有三种模式，参数如下：

> - 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

示例：

```
'dispatch_mode' => 2
```

task_worker_num

描述：服务器开启的task进程数。
说明：设置此参数后，服务器会开启异步task功能。此时可以使用**task**方法投递异步任务。

> 设置此参数后，必须要给swoole_server设置onTask/onFinish两个回调函数，否则启动服务器会报错。

示例：

```
'task_worker_num' => 8
```

task_max_request

描述：每个task进程允许处理的最大任务数。
说明：参考max_request task_worker_num
示例：

```
'task_max_request' => 10000
```

task_ipc_mode

描述：设置task进程与worker进程之间通信的方式。
说明：参考ipc_mode
示例：

```
'task_ipc_mode' => 2
```

task_tmpdir

描述：设置task的数据临时目录
说明：在swoole_server中，如果投递的数据超过8192字节，将启用临时文件来保存数据。这里的task_tmpdir就是用来设置临时文件保存的位置。

> 需要swoole-1.7.7+

示例：

```
'task_tmpdir' => '/tmp/task/'
```

daemonize

描述：设置程序进入后台作为守护进程运行。
说明：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。启用守护进程后，标准输入和输出会被重定向到 log_file，如果 log_file未设置，则所有输出会被丢弃。
示例：

```
'daemonize' => 0
```

log_file

描述：指定日志文件路径
说明：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。
示例：

```
'log_file' => '/data/log/swoole.log'
```

log_level

描述：设置 `swoole_server` 错误日志打印的等级
说明：等级范围是0-5，低于log_level设置的日志信息不会抛出。

示例：

```
'log_level' => '1
```

heartbeat_check_interval

描述：设置心跳检测间隔
说明：此选项表示每隔多久轮循一次，单位为秒。每次检测时遍历所有连接，如果某个连接在间隔时间内没有数据发送，则强制关闭连接（会有onClose回调）。
示例：

```
'heartbeat_check_interval' => 60
```

heartbeat_idle_time

描述：设置某个连接允许的最大闲置时间。
说明：该参数配合heartbeat_check_interval使用。每次遍历所有连接时，如果某个连接在heartbeat_idle_time时间内没有数据发送，则强制关闭连接。默认设置为heartbeat_check_interval * 2。
示例：

```
'heartbeat_idle_time' => 600
```

open_eof_split

描述：打开eof检测功能
说明：与package_eof 配合使用。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package_eof 字符串时才会将数据包投递至Worker进程，否则会一直拼接数据包直到缓存溢出或超时才会终止。一旦出错，该连接会被判定为恶意连接，数据包会被丢弃并强制关闭连接。

> 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

示例：

```
'open_eof_split' => true
```

package_eof

描述：设置EOF字符串
说明：package_eof最大只允许传入**8**个字节的字符串
示例：

```
'package_eof ' => '/r/n'
```

open_length_check

描述：打开包长检测
说明：包长检测提供了固定包头**+**包体这种格式协议的解析，。启用后，可以保证Worker进程onReceive每次都会收到一个完整的数据包。
示例：

```
'open_length_check' => true
```

package_length_offset

描述：包头中第几个字节开始存放了长度字段
说明：配合open_length_check使用，用于指明长度字段的位置。
示例：

```
'package_length_offset' => 5
```

package_body_offset

描述：从第几个字节开始计算长度。
说明：配合open_length_check使用，用于指明包头的长度。
示例：

```
'package_body_offset' => 10
```

package_length_type

描述：指定包长字段的类型
说明：配合open_length_check使用，指定长度字段的类型，参数如下：

- 's' => int16_t 机器字节序
- 'S' => uint16_t 机器字节序
- 'n' => uint16_t 大端字节序
- 'N' => uint32_t 大端字节序
- 'L' => uint32_t 机器字节序
- 'l' => int 机器字节序

示例：

```
'package_length_type' => 'N'
```

package_max_length

描述：设置最大数据包尺寸
说明：该值决定了数据包缓存区的大小。如果缓存的数据超过了该值，则会引发错误。具体错误处理由开启的协议解析的类型决定。
示例：

```
'package_max_length' => 8192
```

open_cpu_affinity

描述：启用CPU亲和性设置
说明：在多核的硬件平台中，启用此特性会将swoole的reactor线程/worker进程绑定到固定的一个核上。可以避免进程/线程的运行时在多个核之间互相切换，提高CPU Cache的命中率。
示例：

```
'open_cpu_affinity' => true
```

cpu_affinity_ignore

描述：设置不使用指定的CPU核
说明：IO密集型程序中，所有网络中断都是用CPU0来处理，如果网络IO很重，CPU0负载过高会导致网络中断无法及时处理，那网络收发包的能力就会下降。此参数用于设置将特定的CPU内核空出专门用于处理网络中断。
接受一个数组作为参数，array(0, 1) 表示不使用CPU0,CPU1

> 此选项必须与open_cpu_affinity同时设置才会生效

示例：

```
'cpu_affinity_ignore' => array(0, 1)
```

open_tcp_nodelay

描述：启用open_tcp_nodelay
说明：开启后TCP连接发送数据时会无关闭Nagle合并算法，立即发往客户端连接。在某些场景下，如http服务器，可以提升响应速度。
示例：

```
'open_tcp_nodelay' => true
```

tcp_defer_accept

描述：启用tcp_defer_accept特性
说明：启动后，只有一个TCP连接有数据发送时才会触发accept。
示例：

```
'tcp_defer_accept' => true
```

open_tcp_keepalive

描述：打开TCP的KEEP_ALIVE选项
说明：使用TCP内置的keep_alive属性，用于保证连接不会因为长时闲置而被关闭。
示例：

```
'open_tcp_keepalive' => true
```

tcp_keepidle

描述：指定探测间隔。
说明：配合open_tcp_keepalive使用，如果某个连接在tcp_keepidle内没有任何数据来往，则进行探测。
示例：

```
'tcp_keepidle' => 600
```

tcp_keepinterval

描述：指定探测时的发包间隔
说明：配合open_tcp_keepalive使用
示例：

```
'tcp_keepinterval' => 60
```

tcp_keepcount

描述：指定探测的尝试次数
说明：配合open_tcp_keepalive使用，若tcp_keepcount次尝试后仍无响应，则判定连接已关闭。
示例：

```
'tcp_keepcount' => 5
```

backlog

描述：指定Listen队列长度
说明：此参数将决定最多同时有多少个等待accept的连接。
示例：

```
'backlog' => 128
```

chroot

描述：重定向Worker进程的文件系统根目录
说明：此设置可以使进程对文件系统的读写与实际的操作系统文件系统隔离。提升安全性。

> 需要swoole-1.7.9+

示例：

```
'chroot' => '/data/server/'
```

user/group

描述：设置worker/task子进程的所属用户/用户组
说明：此设置可以使进程对文件系统的读写与实际的操作系统文件系统隔离。提升安全性。

> 需要swoole-1.7.9+ 仅在使用root用户启动时有效

示例：

```
'user' => 'www'
'group' => 'www'
```

discard_timeout_request

描述：设置是否丢弃已经超时的请求
说明：swoole在配置dispatch_mode=1或3后，系统无法保证onConnect/onReceive/onClose的顺序，因此可能会有一些请求数据在连接关闭后，才能到达Worker进程。

> discard_timeout_request配置默认为true，表示如果worker进程收到了已关闭连接的数据请求，将自动丢弃。 discard_timeout_request如果设置为false，表示无论连接是否关闭Worker进程都会处理数据请求。 需要swoole-1.7.16+

示例：

```
'discard_timeout_request' => true
```

enable_reuse_port

描述：设置端口重用
说明：此参数用于优化TCP连接的Accept性能，启用端口重用后多个进程可以同时进行Accept操作。

> 仅在Linux-3.9.0以上版本的内核可用

示例：

```
'enable_reuse_port' => true
```

ssl_cert_file和ssl_key_file

描述：设置SSL隧道加密
说明：设置值为一个文件名字符串，指定**cert**证书和**key**的路径。
示例：

```
'ssl_cert_file' => '/config/ssl.crt',
'ssl_key_file' => '/config//ssl.key',
```

ssl_client_cert_file

描述：指定CA证书路径
说明：开启双向SSL认证时，需要指定客户端所使用的CA证书

示例：

```
'ssl_client_cert_file' => '../ca.crt'
```

ssl_verify_depth

描述：设置客户证书认证链的长度
说明：开启双向SSL认证时使用

示例：

```
'ssl_verify_depth' => 1
```

ssl_method

描述：设置OpenSSL隧道加密的算法
说明：Server与Client使用的算法必须一致，否则SSL/TLS握手会失败，连接会被切断。 默认算法为 SWOOLE_SSLv23_METHOD

> SWOOLE_SSLv3_METHOD SWOOLE_SSLv3_SERVER_METHOD
> SWOOLE_SSLv3_CLIENT_METHOD SWOOLE_SSLv23_METHOD（默认加密方法）
> SWOOLE_SSLv23_SERVER_METHOD SWOOLE_SSLv23_CLIENT_METHOD
> SWOOLE_TLSv1_METHOD SWOOLE_TLSv1_SERVER_METHOD
> SWOOLE_TLSv1_CLIENT_METHOD SWOOLE_TLSv1_1_METHOD
> SWOOLE_TLSv1_1_SERVER_METHOD SWOOLE_TLSv1_1_CLIENT_METHOD
> SWOOLE_TLSv1_2_METHOD SWOOLE_TLSv1_2_SERVER_METHOD
> SWOOLE_TLSv1_2_CLIENT_METHOD SWOOLE_DTLSv1_METHOD
> SWOOLE_DTLSv1_SERVER_METHOD SWOOLE_DTLSv1_CLIENT_METHOD

示例：

```
'ssl_method' => SWOOLE_SSLv23_METHOD
```

ssl_ciphers

描述：配置SSL加密套件
说明：启用http2协议后，Chrome/Firefox等浏览器要求必须使用高强度加密套件。某些低版本浏览器可能不支持默认的加密套件，可设置为空字符串兼容低版本浏览器
∂

示例：

```
'ssl_ciphers' => 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH'
```

03.swoole_server函数列表

[TOC]

swoole_server::__construct

功能描述：创建一个swoole_server资源对象
函数原型：

```
// 类成员函数
public function swoole_server::__construct(string $host, int $port, int $mode = SWOOLE
_PROCESS,
    int $sock_type = SWOOLE_SOCK_TCP);
// 公共函数
function swoole_server_create(string $host, int $port, int $mode = SWOOLE_PROCESS,
    int $sock_type = SWOOLE_SOCK_TCP);
```

返回：一个swoole_server对象
参数说明：

参数	说明
string host	监听的IP地址
int port	监听的端口号
int mode	运行模式
int sock_type	指定的socket类型

说明： host、port、socket_type的详细说明见swoole_server::addlistener。
mode指定了swoole_server的运行模式，共有如下三种：

mode	类型	说明
SWOOLE_BASE	Base模式	传统的异步非阻塞Server。在Reactor内直接回调PHP的函数。如果回调函数中有阻塞操作会导致Server退化为同步模式。worker_num参数对与BASE模式仍然有效，swoole会启动多个Reactor进程
SWOOLE_THREAD	线程模式（已废弃）	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。多线程模式比进程模式轻量一些，而且线程之间可以共享堆栈和资源。 访问共享内存时会有同步问题，需要使用Swoole提供的锁机制来保护数据。
SWOOLE_PROCESS	进程模式（默认）	Swoole提供了完善的进程管理、内存保护机制。 在业务逻辑非常复杂的情况下，也可以长期稳定运行，适合业务逻辑非常复杂的场景。

样例:

```
$serv = new swoole_server("127.0.0.1" , 8888 , SWOOLE_PROCESS , SWOOLE_SOCK_TCP);
```

swoole_server::set

功能描述：设置swoole_server运行时的各项参数
函数原型：

```
// 类成员函数
public function swoole_server::set(array $setting);
// 公共函数
function swoole_server_set(swoole_server $server, array $setting);
```

返回：无
参数说明：

参数	说明
array setting	配置选项数组，采用key-value形式

说明：
该函数必须在swoole_server::start函数调用前调用。
全部swoole_server的配置参数点此查看
样例:

```
$serv->set(
    array(
        'worker_num' => 8,
        'max_request' => 10000,
        'max_conn' => 100000,
        'dispatch_mode' => 2,
        'debug_mode'=> 1,
        'daemonize' => false,
    )
);
```

swoole_server::on

功能描述：绑定swoole_server的相关回调函数
函数原型：

```
// 类成员函数
public function bool swoole_server->on(string $event, mixed $callback);
```

返回：设置成功返回true，否则返回false
参数说明：

参数	说明
string event	回调的名称（大小写不敏感）
mixed callback	回调的PHP函数，可以是函数名的字符串，类静态方法，对象方法数组，匿名函数

说明：
该函数必须在swoole_server::start函数调用前调用。
此方法与swoole_server::handler功能相同，作用是与swoole_client风格保持一致。
swoole_server::on中事件名称字符串不要加on。
全部的回调函数列表点此查看
样例:

```
$serv->on('connect', function ($serv, $fd){
    echo "Client:Connect.\n";
});
$serv->on('receive', array( $myclass, 'onReceive' ) ); // onReceive是myclass的成员函数
```

swoole_server::addlistener

功能描述：给swoole_server增加一个监听的地址和端口
函数原型：

```
// 类成员函数
public function swoole_server::addlistener(string $host, int $port, $type = SWOOLE_SOC
K_TCP);
// 公共函数
function swoole_server_addlisten(swoole_server $serv, string $host, int $port, $type =
 SWOOLE_SOCK_TCP);
```

返回：无
参数说明：

参数	说明
string host	监听的IP地址
int port	监听的端口号
int sock_type	指定的socket类型

说明： swoole支持如下socket类型:

sock_type	说明
SWOOLE_TCP/SWOOLE_SOCK_TCP	TCP IPv4 Socket
SWOOLE_TCP6/SWOOLE_SOCK_TCP6	TCP IPv6 Socket
SWOOLE_UDP/SWOOLE_SOCK_UDP	UDP IPv4 Socket
SWOOLE_UDP6/SWOOLE_SOCK_UDP6	UDP IPv4 Socket
SWOOLE_UNIX_DGRAM	Unix UDP Socket
SWOOLE_UNIX_STREAM	Unix TCP Socket

> Unix Socket仅在1.7.1+后可用，此模式下**host**参数必须填写可访问的文件路径，**port**参数忽略
> Unix Socket模式下，客户端**fd**将不再是数字，而是一个文件路径的字符串
> SWOOLE_TCP等是1.7.0+后提供的简写方式，与1.7.0前的SWOOLE_SOCK_TCP是等同的

样例:

```
$serv->addlistener("127.0.0.1", 9502, SWOOLE_SOCK_TCP);
$serv->addlistener("192.168.1.100", 9503, SWOOLE_SOCK_TCP);
$serv->addlistener("0.0.0.0", 9504, SWOOLE_SOCK_UDP);
$serv->addlistener("/var/run/myserv.sock", 0, SWOOLE_UNIX_STREAM);

swoole_server_addlisten($serv, "127.0.0.1", 9502, SWOOLE_SOCK_TCP);
```

swoole_server::handler

功能描述：设置Server的事件回调函数
函数原型：

```
// 类成员函数
public function swoole_server::handler(string $event_name, mixed $event_callback_funct
ion);
// 公共函数
function swoole_server_handler(swoole_server $serv, string $event_name, mixed $event_c
allback_function);
```

返回：设置成功返回true，否则返回false
参数说明：

参数	说明
string event_name	回调的名称（大小写不敏感）
mixed event_callback_function	回调的PHP函数，可以是函数名的字符串，类静态方法，对象方法数组，匿名函数

说明： 该函数必须在swoole_server::start函数调用前调用。
事件名称字符串要加on。
全部的回调函数列表点此查看

> onConnect/onClose/onReceive这3个回调函数必须设置。其他事件回调函数可选
> 如果设定了timer定时器，onTimer事件回调函数也必须设置
> 如果启用了task_worker，onTask/onFinish事件回调函数必须设置

样例:

```
$serv->handler('onStart', 'my_onStart');
$serv->handler('onStart', array($this, 'my_onStart'));
$serv->handler('onStart', 'myClass::onStart');
```

swoole_server::start

功能描述：启动server，开始监听所有TCP/UDP端口
函数原型：

```
// 类成员函数
public function swoole_server::start()
```

返回：启动成功返回true，否则返回false
参数说明：无
说明：
启动成功后会创建worker_num+2个进程：Master进程+Manager进程+worker_num 个Worker进程。
另外。启用task_worker会增加task_worker_num个Worker进程
三种进程的说明如下：

进程类型	说明
Master进程	主进程内有多个Reactor线程，基于epoll/kqueue进行网络事件轮询。收到数据后转发到Worker进程去处理
Manager进程	对所有Worker进程进行管理，Worker进程生命周期结束或者发生异常时自动回收，并创建新的Worker进程
Worker进程	对收到的数据进行处理，包括协议解析和响应请求

样例:

```
$serv->start();
```

swoole_server::reload

功能描述：重启所有worker进程。
函数原型：

```
// 类成员函数
public function swoole_server::reload()
```

返回：调用成功返回true，否则返回false
参数说明：无
说明：

调用后会向Manager发送一个SIGUSR1信号，平滑重启全部的Worker进程（所谓平滑重启，是指重启动作会在Worker处理完正在执行的任务后发生，并不会中断正在运行的任务。）

小技巧：在onWorkerStart回调中require相应的php文件，当这些文件被修改后，只需要通过SIGUSR1信号即可实现服务器热更新。

1.7.7版本增加了仅重启task_worker的功能。只需向服务器发送SIGUSR2即可
样例:

```
$serv->reload();
```

swoole_server::shutdown

功能描述：关闭服务器。
函数原型：

```
// 类成员函数
public function swoole_server::shutdown()
```

返回：调用成功返回true，否则返回false
参数说明：无
说明：
此函数可以用在worker进程内，平滑关闭全部的Worker进程。
也可向Master进程发送SIGTERM信号关闭服务器。
样例:

```
$serv->shutdown();
```

swoole_server::addtimer

功能描述：设置一个固定间隔的定时器
函数原型：

```
// 类成员函数
public function swoole_server::addtimer(int $interval);
// 公共函数
function swoole_server_addtimer(swoole_server $serv, int $interval);
```

返回：设置成功返回true，否则返回false
参数说明：

参数	说明
int interval	定时器的时间间隔，单位为毫秒ms

说明：
swoole定时器的最小颗粒是1毫秒，支持多个不同间隔的定时器。
注意不能存在2个相同间隔时间的定时器。
使用多个定时器时，其他定时器必须为最小定时器时间间隔的整数倍。

该函数只能在onWorkerStart/onConnect/onReceive/onClose回调函数中调用。
增加定时器后需要为Server设置onTimer回调函数，否则Server将无法启动。

样例:

```
$serv->addtimer(1000);              //1s
swoole_server_addtimer($serv,20);   //20ms
```

swoole_server::deltimer

功能描述：删除指定的定时器。
函数原型：

```
// 类成员函数
public function swoole_server::deltimer(int $interval);
```

返回：无
参数说明：

参数	说明
int interval	定时器的时间间隔，单位为毫秒ms

说明：
删除间隔为interval的定时器

样例:

```
$serv->deltimer(1000);
```

swoole_server::after

功能描述：在指定的时间后执行函数
函数原型：

```
// 类成员函数
public function swoole_server::after(int $after_time_ms, mixed $callback_function, mix
ed params);
// 公共函数
function swoole_timer_after(swoole_server $serv, int $after_time_ms， mixed $callback_f
unction, mixed params);
```

返回：无
参数说明：

参数	说明
int after_time_ms	指定时间，单位为毫秒ms
mixed callback_function	指定的回调函数
mixed params	指定的回调函数的参数

说明：
创建一个一次性定时器，在指定的after_time_ms时间后调用callback_funciton回调函数，执行完成后就会销毁。
callback_function函数的参数为指定的params
需要swoole-1.7.7以上版本。

样例:

```
$serv->after(1000, function( $params ){
    echo "Do something\n";
} , "data" );
```

swoole_server::close

功能描述：关闭客户端连接
函数原型：

```
// 类成员函数
public function swoole_server::close(int $fd, int $from_id = 0);
```

返回：关闭成功返回true，失败返回false
参数说明：

参数	说明
int fd	指定关闭的fd
int from_id	fd来自于哪个Reactor（swoole-1.6以后已废弃该参数）

说明：
调用close关闭连接后，连接并不会马上关闭，因此不要在close之后立即写清理逻辑，而是应该在onClose回调中处理

样例:

```
$serv->close( $fd );
```

swoole_server::send

功能描述：向客户端发送数据
函数原型：

```
// 类成员函数
public function swoole_server::send(int $fd, string $data, int $from_id = 0);
```

返回：发送成功返回true，失败返回false
参数说明：

参数	说明
int fd	指定发送的fd
string data	发送的数据
int from_id	fd来自于哪个Reactor

说明：

- data，发送的数据，最大不得超过2M。send操作具有原子性，多个进程同时调用send向同一个连接发送数据，不会发生数据混杂。
- 如果要发送超过2M的数据，建议将数据写入临时文件，然后通过sendfile接口直接发送文件
- UDP协议，send会直接在worker进程内发送数据包
- 向UDP客户端发送数据，必须要传入from_id
- 发送成功会返回true，如果连接已被关闭或发送失败会返回false
- swoole-1.6以上版本不需要from_id
- UDP协议使用fd保存客户端IP，from_id保存from_fd和port
- UDP协议如果是在onReceive后立即向客户端发送数据，可以不传from_id

样例:

```
$serv->send( $fd , "Hello World");
```

swoole_server::sendfile

功能描述：发送文件到TCP客户端连接
函数原型：

```
// 类成员函数
public function swoole_server::sendfile(int $fd, string $filename);
```

返回：发送成功返回true，失败返回false
参数说明：

参数	说明
int fd	指定发送的fd
string filename	发送的文件名

说明：
sendfile函数调用OS提供的sendfile系统调用，由操作系统直接读取文件并写入socket。
sendfile只有2次内存拷贝，使用此函数可以降低发送大量文件时操作系统的CPU和内存占用。

样例:

```
$serv->sendfile($fd, __DIR__.'/test.jpg');
```

swoole_server::connection_info

功能描述：获取连接的信息
函数原型：

```
// 类成员函数
public function swoole_server::connection_info(int $fd, int $from_id = 0);
```

返回：如果fd存在，返回一个数组，连接不存在或已关闭返回false
参数说明：

参数	说明
int fd	指定发送的fd
int from_id	来自于哪个Reactor

说明：
UDP socket调用该参数时必须传入from_id.
返回的结果如下：

名称	说明
int from_id	来自于哪个Reactor
int from_fd	来自哪个Server Socket
int from_port	来自哪个Server端口
int remote_port	客户端连接的端口
string remote_ip	客户端连接的ip
int connect_time	连接到Server的时间，单位秒
int last_time	最后一次发送数据的时间，单位秒

sendfile函数调用OS提供的sendfile系统调用，由操作系统直接读取文件并写入socket。sendfile只有2次内存拷贝，使用此函数可以降低发送大量文件时操作系统的CPU和内存占用。

样例:

```
$fdinfo = $serv->connection_info($fd);
$udp_client = $serv->connection_info($fd, $from_id);
```

swoole_server::connection_list

功能描述：遍历当前Server的全部客户端连接
函数原型：

```
// 类成员函数
public function swoole_server::connection_list(int $start_fd = 0, int $pagesize = 10);
```

返回：调用成功将返回一个数字索引数组，元素是取到的fd。数组会按从小到大排序。最后一个fd作为新的start_fd再次尝试获取。
调用失败返回false
参数说明：

参数	说明
int start_fd	起始fd
int pagesize	每页取多少条，最大不得超过100.

说明：
connection_list仅可用于TCP，UDP服务器需要自行保存客户端信息

样例:

```
$start_fd = 0;
while(true)
{
    $conn_list = $serv->connection_list($start_fd, 10);
    if($conn_list===false)
    {
        echo "finish\n";
        break;
    }
    $start_fd = end($conn_list);
    var_dump($conn_list);
    foreach($conn_list as $fd)
    {
        $serv->send($fd, "broadcast");
    }
}
```

swoole_server::stats

功能描述：获取当前Server的活动TCP连接数，启动时间，accpet/close的总次数等信息。
函数原型：

```
// 类成员函数
public function swoole_server->stats();
```

返回：结果数组
参数说明：无
说明：
stats方法在1.7.5+后可用

名称	说明
int start_time	启动时间
int connection_num	当前的连接数
int accept_count	accept总次数
int close_count	close连接的总数

样例:

```
$status = $serv->stats();
```

swoole_server::task

功能描述：投递一个异步任务到task_worker池中
函数原型：

```
// 类成员函数
public function swoole_server::task(string $data, int $dst_worker_id = -1);
```

返回：调用成功返回task_worker_id，失败返回false
参数说明：

参数	说明
string data	task数据
int dst_worker_id	指定投递给哪个task进程，默认随机投递

说明：
此功能用于将慢速的任务异步地去执行，调用task函数会立即返回，Worker进程可以继续处理新的请求。
函数会返回一个 `$task_id` 数字，表示此任务的ID
任务完成后，可以通过return(低于swoole-1.7.2版本使用finish函数)将结果通过onFinish回调返回给Worker进程。
发送的data必须为字符串，如果是数组或对象，请在业务代码中进行serialize处理，并在onTask/onFinish中进行unserialize。
data可以为二进制数据，最大长度为**8K**(超过8K可以使用临时文件共享)。字符串可以使用gzip进行压缩。
使用task必须为Server设置onTask和onFinish回调,并指定task_worker_num配置参数。

样例:

```
$task_id = $serv->task("some data");
```

swoole_server::taskwait

功能描述：投递一个同步任务到task_worker池中
函数原型：

```
// 类成员函数
public function swoole_server::taskwait(string $task_data, float $timeout = 0.5, int $
dst_worker_id = -1);
```

返回：task执行的结果
参数说明：

参数	说明
string data	task数据
float timeout	超时时间
int dst_worker_id	指定投递给哪个task进程，默认随机投递

说明：
taskwait与task方法作用相同，用于投递一个异步的任务到task进程池去执行。与task不同的是taskwait是阻塞等待的，直到任务完成或者超时返回。
任务完成后，可以通过return直接返回结果

样例:

```
$task_id = $serv->taskwait("some data", 30);
```

swoole_server::finish

功能描述：传递Task结果数据给worker进程
函数原型：

```
// 类成员函数
public function swoole_server::finish(string $task_data);
```

返回：无
参数说明：

参数	说明
string data	结果数据

说明：
使用swoole_server::finish函数必须为Server设置onFinish回调函数。此函数只可用于Task Worker进程的onTask回调中
swoole_server::finish是可选的。如果Worker进程不关心任务执行的结果，可以不调用此函数
此函数在swoole-1.7.2以上版本已废弃，使用return代替。
样例:

```
$serv->finish("result data");
```

swoole_server::heartbeat

功能描述：进行心跳检测
函数原型：

```
// 类成员函数
public function swoole_server::heartbeat(bool $if_close_connection = true);
```

返回：无
参数说明：

参数	说明
bool if_close_connection	是否关闭超时的连接，默认为true

说明：
该函数会主动发起一次检测，遍历全部连接，根据设置的heartbeat_check_interval和heartbeat_idle_time参数，找到那些处于idle闲置状态的连接
swoole默认会直接关闭这些连接，heartbeat会返回这些连接的fd
如果if_close_connection为false，则heartbeat会返回这些idle连接的fd，但不会关闭这些连接
if_close_connection参数 在swoole-1.7.4以上版本可用
样例:

```
$serv->heartbeat();
```

swoole_get_mysqli_sock

功能描述：获取mysqli的socket文件描述符
函数原型：

```
// 公共函数
int swoole_get_mysqli_sock(mysqli $db)
```

返回：mysqli的fd
参数说明：

参数	说明
mysqli db	mysqli的连接

说明：
可将mysql的socket增加到swoole中，执行异步MySQL查询。如果想要使用异步MySQL，需要在编译swoole时制定--enable-async-mysql
swoole_get_mysqli_sock仅支持mysqlnd驱动
即使是异步MySQL也需要一个连接池，并发SQL必须有多个连接。
样例:

```
$db = new mysqli;
$db->connect('127.0.0.1', 'root', 'root', 'test');
$db->query("show tables", MYSQLI_ASYNC);
swoole_event_add(swoole_get_mysqli_sock($db), function($db_sock) {
    global $db;
    $res = $db->reap_async_query();
    var_dump($res->fetch_all(MYSQLI_ASSOC));
    swoole_event_exit();
});
```

swoole_set_process_name

功能描述：设置进程的名称
函数原型：

```
// 公共函数
void swoole_set_process_name(string $name);
```

返回：无
参数说明：

参数	说明
string name	进程名称

说明：
修改进程名称后，通过ps命令看到的将不再是php your_file.php，而是设定的字符串

> 在swoole_server_create之前修改为manager进程名称 onStart调用时修改为主进程名称 onWorkerStart修改为worker进程名称

swoole_set_process_name存在兼容性问题，优先使用PHP内置的cli_set_process_title函数
样例:

```
woole_set_process_name("swoole server");
```

swoole_version

功能描述：获取swoole扩展的版本号
函数原型：

```
// 公共函数
string swoole_version();
```

返回：swoole扩展的版本号
参数说明：无
说明：
样例:

```
echo swoole_version();
```

swoole_strerror

功能描述：将标准的Unix Errno错误码转换成错误信息
函数原型：

```
// 公共函数
string swoole_strerror(int $errno);
```

返回：转化后的错误信息
参数说明：

参数	说明
int errno	errno错误码

说明：
样例:

```
echo swoole_strerror( $errno );
```

swoole_errno

功能描述：获取最近一次系统调用的错误码
函数原型：

```
// 公共函数
int swoole_errno();
```

返回：最近一次系统调用的错误码
参数说明：无
说明：
错误码的值与操作系统有关。可是使用swoole_strerror将错误转换为错误信息。
样例:

```
echo swoole_strerror(swoole_errno());
```

swoole_get_local_ip

功能描述：此函数用于获取本机所有网络接口的IP地址
函数原型：

```
// 公共函数
array swoole_get_local_ip();
```

返回：以interface名称为key的关联数组
参数说明：无
说明：
目前只返回IPv4地址，返回结果会过滤掉本地loop地址127.0.0.1
返回结果样例array("eth0" => "192.168.1.100")； 样例:

```
var_dump(swoole_get_local_ip());
```