workerman.php源代码阅读

火魂

目　录

概述

1 目录结构

```
Workerman/                          ;Workerman内核代码
    Connection/                     ;socket连接
        ConnectionInterface.php     ;socket连接接口
        TcpConnecttion.php          ;Tcp连接
        AsyncTcpConnecttion.php     ;异步Tcp连接
        UdpConnection.php           ;Udp连接

    Events/                         ;网络事件库
        EventInterface.php          ;网络事件库接口
        Libevent.php                ;Libevent网络事件库
        Ev.php                      ;LivEv网络事件库
        Select.php                  ;Select网络事件库

    Lib/                            ;类库
        Constants.php               ;常量定义
        Timer.php                   ;定时器

    Protocols/
        ProtocolInterface.php       ;协议接口
        Http
            mime.types              ;mime类型
        Http.php                    ;http协议
        Text.php                    ;Text协议
        Frame.php                   ;Frame协议
        Websocket.php               ;websocket协议
    Worker.php                      ;Worker
    WebServer.php                   ;WebServer
    Autoloader.php                  ;自动加载器
```

2 主体流程

```
<?php
use Workerman\Worker;
require_once '/your/path/Workerman/Autoloader.php';

$global_uid = 0;

// 当客户端连上来时分配uid，并保存连接，并通知所有客户端
function handle_connection($connection)
{
    global $text_worker, $global_uid;
    // 为这个链接分配一个uid
    $connection->uid = ++$global_uid;
}

// 当客户端发送消息过来时，转发给所有人
```

```
function handle_message($connection, $data)
{
    global $text_worker;
    foreach($text_worker->connections as $conn)
    {
        $conn->send("user[{$connection->uid}] said: $data");
    }
}

// 当客户端断开时，广播给所有客户端
function handle_close($connection)
{
    global $text_worker;
    foreach($text_worker->connections as $conn)
    {
        $conn->send("user[{$connection->uid}] logout");
    }
}

// 创建一个文本协议的Worker监听2347接口
$text_worker = new Worker("text://0.0.0.0:2347");

// 只启动1个进程，这样方便客户端之间传输数据
$text_worker->count = 1;

$text_worker->onConnect = 'handle_connection';
$text_worker->onMessage = 'handle_message';
$text_worker->onClose = 'handle_close';

Worker::runAll();
```

1 创建worker处理对象

`$text_worker = new Worker("text://0.0.0.0:2347");`

2 设置启动进程数目

`$text_worker->count = 1;`

3 注册事件接口

```
$text_worker->onConnect = 'handle_connection';
$text_worker->onMessage = 'handle_message';
$text_worker->onClose = 'handle_close';
```

4 启动worker

```
Worker::runAll();
```

框架流程

主：worker初始化

1 源代码

```
Worker.php

public function __construct($socket_name = '', $context_option = array())
    {
        // Save all worker instances.
        $this->workerId                    = spl_object_hash($this);
        self::$_workers[$this->workerId] = $this;
        self::$_pidMap[$this->workerId]  = array();

        // Get autoload root path.
        $backrace                = debug_backtrace();
        $this->_autoloadRootPath = dirname($backrace[0]['file']);

        // Context for socket.
        if ($socket_name) {
            $this->_socketName = $socket_name;
            if (!isset($context_option['socket']['backlog'])) {
                $context_option['socket']['backlog'] = self::DEFAUL_BACKL
OG;
            }
            $this->_context = stream_context_create($context_option);
        }

        // Set an empty onMessage callback.
        $this->onMessage = function () {
        };
    }
```

2 分析

1 函数接口

`__construct($socket_name = '', $context_option = array())`

> $socket_name: <协议>://<监听地址>类参数

> $context_option: socket的上下文选项

>> <协议> 可以为以下格式：

> tcp: 例如 tcp://0.0.0.0:8686

> udp: 例如 udp://0.0.0.0:8686

> unix: 例如 unix:///tmp/my_file (需要Workerman>=3.2.7)

> http: 例如 http://0.0.0.0:80

> websocket: 例如 websocket://0.0.0.0:8686

> text: 例如 text://0.0.0.0:8686

>> <监听地址> 可以为以下格式：

> 如果是unix套接字，地址为本地一个磁盘路径

> 非unix套接字，地址格式为 <本机ip>:<端口号>

> <本机ip>可以为0.0.0.0表示监听本机所有网卡，包括内网ip和外网ip及本地回环127.0.0.1

> <本机ip>如果以为127.0.0.1表示监听本地回环，只能本机访问，外部无法访问

> <本机ip>如果为内网ip，类似192.168.xx.xx，表示只监听内网ip，则外网用户无法访问

> <本机ip>设置的值不属于本机ip则无法执行监听，并且提示Cannot assign requested address错误

> 注意：<端口号>不能大于65535。<端口号>如果小于1024则需要root权限才能监听。监听的端口必须是本机未被占用的端口，否则无法监听，并且提示Address already in use错误

> $context_option: 套接字选项见 基础原理 socket上下文选项呢

2 函数分析

```
$this->workerId                  = spl_object_hash($this);
self::$_workers[$this->workerId] = $this;
self::$_pidMap[$this->workerId]  = array();
```

1 分配workeId
设置workerId。spl_object_hash()函数获取对象的hash值。
spl_object_hash()见 基础原理 基础函数
注册workerId到$_workers和$_pidMap

```
$backrace                = debug_backtrace();
$this->_autoloadRootPath = dirname($backrace[0]['file']);
```

2 设置自动加载根目录
debug_backtrace() 见 基础函数
dirname()返回文件目录信息 见 基础函数

```
if ($socket_name) {
    $this->_socketName = $socket_name;
    if (!isset($context_option['socket']['backlog'])) {
        $context_option['socket']['backlog'] = self::DEFAUL_BACKLOG;
    }
    $this->_context = stream_context_create($context_option);
}
```

3 设置socket上下文。
设置_socketName

检查socket的backlog选项，见 socket上下文选项
最后调用straem_context_create() 创建资源流上下文
stream_context_create() 见 socket上下文选项

```
$this->onMessage = function () {
        };
```

4 注册默认的onMessage接口函数

```
$worker->count = 8;
```

5 设置启动进程数

3 总结

构造函数完成worker的基础属性设置

> 1 workerId分配
> 2 加载根目录
> 3 socket上下文选项
> 4 注册OnMessage回调接口
> 5 设置启动进程数

主：worker事件接口

1 源代码

(onWorkerStart)worker启动回调

```
public $onWorkerStart = null;

worker::run()

if ($this->onWorkerStart) {
    try {
        call_user_func($this->onWorkerStart, $this);
    } catch (\Exception $e) {
        echo $e;
        exit(250);
    }
}
```

(onWorkerReload)worker重载回调

```
public $onWorkerReload = null;
```

```
worker::reload()

if ($worker->onWorkerReload) {
    try {
        call_user_func($worker->onWorkerReload, $worker);
    } catch (\Exception $e) {
        echo $e;
        exit(250);
    }
}

if ($worker->reloadable) {
    self::stopAll();
}
```

(onWorkerStop)worker停止回调函数

```
public $onWorkerStop = null;

worker::stop()

if ($this->onWorkerStop) {
    try {
        call_user_func($this->onWorkerStop, $this);
    } catch (\Exception $e) {
        echo $e;
        exit(250);
    }
}
```

(onConnect)客户端连接时回调函数

```
public $onConnect = null;

worker::acceptConnection()

if ($this->onConnect) {
    try {
        call_user_func($this->onConnect, $connection);
    } catch (\Exception $e) {
        echo $e;
        exit(250);
    }
}
```

(onMessage)接受到客户端数据时回调函数

```
public $onMessage = null;
```

```
worker::__construct()
{
......
    $this->onMessage = function () {
        };
}
```

```
worker::acceptConnection()
{
 ......
 $connection->onMessage              = $this->onMessage;
 ......
}
```

```
worker::acceptUdpConnection()
{
......
    if ($this->onMessage) {
        if ($this->protocol) {
            $parser      = $this->protocol;
            $recv_buffer = $parser::decode($recv_buffer, $connection);
        }
        ConnectionInterface::$statistics['total_request']++;
        try {
            call_user_func($this->onMessage, $connection, $recv_buffer);
        } catch (\Exception $e) {
            echo $e;
            exit(250);
        }
    }
.....
}
```

(onClose)客户端连接断开时回调

```
public $onClose = null;

worker::acceptConnection()

$connection->onClose                = $this->onClose;
```

(onBufferFull)发送缓冲区数据达到上限回调函数

```
public $onBufferFull = null;

worker::acceptConnection()
```

```
$connection->onBufferFull           = $this->onBufferFull;
```

(onBufferDrain)发送缓冲区数据发送完毕回调函数

```
public $onBufferDrain = null;
worker::acceptConnection()
$connection->onBufferDrain          = $this->onBufferDrain;
```

(onError) 客户端连接发送错误时回调

```
public $onError = null;
worker::acceptConnection()
$connection->onError                = $this->onError;
```

2 文件分析

1 worker相关

onWokerStart: worker启动回调
onWorkerReload: worker重载回调
onWokrerStop: worker停止回调

2 connect相关

onConnet: 客户端连接建立时回调
onMessage: 服务器接受到数据时回调
onClose: 客户端连接关闭时回调
onBufferFull: 数据缓冲区达到上限时回调
onBufferDrain: 数据缓冲区发送完毕时回调
onError: 客户端连接错误时回调

3 总结

事件接口注册相关函数。

Worker接口 3个
Connect接口 4个

主：worker启动过程

1 源代码

```
public static function runAll()
{
    self::checkSapiEnv();
    self::init();
    self::parseCommand();
    self::daemonize();
    self::initWorkers();
    self::installSignal();
    self::saveMasterPid();
    self::forkWorkers();
    self::displayUI();
    self::resetStd();
    self::monitorWorkers();
}
```

2 文件分析

0 总体流程

```
checkSapiEnv()   ;检查sapi环境是否为cli
```

```
init()            ;worker环境初始化

parseComand()     ;命令行参数解析
daemonize()       ;主进程守护模式启动
initWorkers()     ;初始化worker实例
installSignal()   ;安装信号处理句柄
saveMasterPid()   ;保存主pid
forkWorkers()     ;创建worker子进程
displayUI()       ;启动信息显示
resetStd()        ;重定向输入输出
monitorWorkers() ;监控所有worker子进程
```

1 checkSapiEnv() 检查sapi环境是否为wcli

```
if (php_sapi_name() != "cli") {
        exit("only run in command line mode \n");
    }
```

调用php_sapi_name()检查sapi运行环境是否为cli

2 init() 环境初始化

```
$backtrace        = debug_backtrace();
self::$_startFile = $backtrace[count($backtrace) - 1]['file'];
```

启动文件

```
if (empty(self::$pidFile)) {
            self::$pidFile = __DIR__ . "/../" . str_replace('/', '_', sel
f::$_startFile) . ".pid";
        }
```

进程文件

```
if (empty(self::$logFile)) {
    self::$logFile = __DIR__ . '/../workerman.log';
}
touch(self::$logFile);
chmod(self::$logFile, 0622);
```

日志文件

`self::$_status = self::STATUS_STARTING;`

worker状态

```
self::$_globalStatistics['start_timestamp'] = time();
self::$_statisticsFile                      = sys_get_temp_dir() . '/work
erman.status';
```

启动状态记录

`self::setProcessTitle('WorkerMan: master process  start_file=' . se`

设置进程名称

worker::setProcessTitle() 见框架工具 worker文件

`self::initId();`

初始化workerid数据

worker::setProcessTitle()

`Timer::init();`

计时器初始化

3 parseComand() 命令行解析

`global $argv;`

命令行参数

`$start_file = $argv[0];`

启动文件

```
if (!isset($argv[1])) {
    exit("Usage: php yourfile.php {start|stop|restart|reload|status|kill}
\n");
}
```

启动文件后的信号参数检查

```
$command  = trim($argv[1]);
$command2 = isset($argv[2]) ? $argv[2] : '';
```

命令参数获取

```
$mode = '';
if ($command === 'start') {
    if ($command2 === '-d') {
        $mode = 'in DAEMON mode';
    } else {
```

```
            $mode = 'in DEBUG mode';
        }
    }
```

php yourfile.php start debug调试模式启动
php yourfile.php start -d 守护进程模式启动

`self::log("Workerman[$start_file] $command $mode");`
日志记录启动信息
worker::log()见 框架工具 worker

```
$master_pid      = @file_get_contents(self::$pidFile);
$master_is_alive = $master_pid && @posix_kill($master_pid, 0);
```

主进程id获取与状态检测
posix_kill 见 基础原理 基础函数

```
if ($master_is_alive) {
    if ($command === 'start') {
        self::log("Workerman[$start_file] already running");
        exit;
    }
} elseif ($command !== 'start' && $command !== 'restart') {
    self::log("Workerman[$start_file] not run");
}
```

主进程启动检测

其他命令执行
`switch ($command) {}`

```
case 'kill':
    exec("ps aux | grep $start_file | grep -v grep | awk '{print $2}' |xa
rgs kill -SIGINT");
    exec("ps aux | grep $start_file | grep -v grep | awk '{print $2}' |xa
rgs kill -SIGKILL");
    break;
```

> kill命令的执行

exec() 见 基础原理 基础函数

```
case 'start':
    if ($command2 === '-d') {
        Worker::$daemonize = true;
    }
    break;
```

> start 命令的执行

设置守护进程模式

```
case 'status':
    if (is_file(self::$_statisticsFile)) {
        @unlink(self::$_statisticsFile);
    }

    posix_kill($master_pid, SIGUSR2);

    usleep(100000);

    @readfile(self::$_statisticsFile);
    exit(0);
```

> status命令的执行

删除状态文件

发送信号

等待一会

读取状态文件

```
case 'restart':
case 'stop':
    self::log("Workerman[$start_file] is stoping ...");
    // Send stop signal to master process.
    $master_pid && posix_kill($master_pid, SIGINT);
    // Timeout.
    $timeout    = 5;
    $start_time = time();
    // Check master process is still alive?
    while (1) {
        $master_is_alive = $master_pid && posix_kill($master_pid, 0);
        if ($master_is_alive) {
            // Timeout?
            if (time() - $start_time >= $timeout) {
                self::log("Workerman[$start_file] stop fail");
                exit;
```

```
            }
            // Waiting amoment.
            usleep(10000);
            continue;
        }
        // Stop success.
        self::log("Workerman[$start_file] stop success");
        if ($command === 'stop') {
            exit(0);
        }
        if ($command2 === '-d') {
            Worker::$daemonize = true;
        }
        break;
    }
    break;
```

> restart,stop命令的执行

```
self::log("Workerman[$start_file] is stoping ...");
```

日志记录 正在停止信息

```
$master_pid && posix_kill($master_pid, SIGINT);
```

主进程停止信号

```
$timeout    = 5;
$start_time = time();
```

延迟计时

```
$master_is_alive = $master_pid && posix_kill($master_pid, 0);
```

```
if ($master_is_alive) {
    // Timeout?
    if (time() - $start_time >= $timeout) {
        self::log("Workerman[$start_file] stop fail");
        exit;
    }
    // Waiting amoment.
    usleep(10000);
    continue;
```

```
}
```

检测是否停止成功

```
self::log("Workerman[$start_file] stop success");
```

日志记录停止成功

```
if ($command === 'stop') {
    exit(0);
}
```

stop命令 直接退出

```
if ($command2 === '-d') {
    Worker::$daemonize = true;
}
break;
```

restart -d 命令 设置守护模式

```
case 'reload':
    posix_kill($master_pid, SIGUSR1);
    self::log("Workerman[$start_file] reload");
    exit;
```

> reload 命令

```
default :
    exit("Usage: php yourfile.php {start|stop|restart|reload|status|kill}
\n");
```

提示 命令行格式信息

4 daemonize() 以守护模式启动

```
if (!self::$daemonize) {
    return;
}
```

检测是否需要守护模式启动

`umask(0);`
修改mask

```
$pid = pcntl_fork();
if (-1 === $pid) {
    throw new Exception('fork fail');
} elseif ($pid > 0) {
    exit(0);
}
```

创建进程

pcntl_fork() 见 基础原理 基础函数

```
if (-1 === posix_setsid()) {
    throw new Exception("setsid fail");
}
```

session初始化
posix_setsid() 见 基础原理 基础函数

```
$pid = pcntl_fork();
if (-1 === $pid) {
    throw new Exception("fork fail");
} elseif (0 !== $pid) {
    exit(0);
}
```

??

5 initWorkers() 初始化所有worker实例

`foreach (self::$_workers as $worker) {}`
遍历worker实例数组

```
if (empty($worker->name)) {
    $worker->name = 'none';
}
```

获取worker名称

```
$worker_name_length = strlen($worker->name);
if (self::$_maxWorkerNameLength < $worker_name_length) {
    self::$_maxWorkerNameLength = $worker_name_length;
}

$socket_name_length = strlen($worker->getSocketName());
if (self::$_maxSocketNameLength < $socket_name_length) {
    self::$_maxSocketNameLength = $socket_name_length;
}
```

名称长度检测

```
if (empty($worker->user)) {
    $worker->user = self::getCurrentUser();
} else {
    if (posix_getuid() !== 0 && $worker->user != self::getCurrentUser())
{
        self::log('Warning: You must have the root privileges to change u
id and gid.');
    }
}
```

获取用户信息

```
$user_name_length = strlen($worker->user);
if (self::$_maxUserNameLength < $user_name_length) {
    self::$_maxUserNameLength = $user_name_length;
}
```

用户名称长度检测

```
 if (!$worker->reusePort) {
    $worker->listen();
}
```

开启监听端口

worker::listen() 见 框架工具的 worker类

6 installSignal() 安装主进程信号处理函数

```
pcntl_signal(SIGINT, array('\Workerman\Worker', 'signalHandler'), false);

pcntl_signal(SIGUSR1, array('\Workerman\Worker', 'signalHandler'), false)
;
```

```
pcntl_signal(SIGUSR2, array('\Workerman\Worker', 'signalHandler'), false)
;

pcntl_signal(SIGPIPE, SIG_IGN, false);
```

stop,reload,status,ignore信号接口函数注册为worker::signalHanlder()

signalHanlder() 见 框架工具 worker类

7 saveMasterPid() 保存主进程id到pid文件

```
self::$_masterPid = posix_getpid();
```

获取pid信息

```
if (false === @file_put_contents(self::$pidFile, self::$_masterPid)) {
            throw new Exception('can not save pid to ' . self::$pidFile);
        }
```

写入pid信息

8 forkWorkers() 创建workers子进程

```
foreach (self::$_workers as $worker) {}
```

遍历workers数组

```
if (self::$_status === self::STATUS_STARTING) {
    if (empty($worker->name)) {
        $worker->name = $worker->getSocketName();
    }
    $worker_name_length = strlen($worker->name);
    if (self::$_maxWorkerNameLength < $worker_name_length) {
        self::$_maxWorkerNameLength = $worker_name_length;
    }
}
```

worker名称获取

```
while (count(self::$_pidMap[$worker->workerId]) < $worker->count) {
    static::forkOneWorker($worker);
}
```

依次创建worker进程

forkOneWorker() 见 Worker功能函数

9 displayUI() 输出服务器启动信息

10 resetStd() 重定向标准输出

设置标准输出到指定输出文件，

11 monitorWorkers() 监控worker进程

`self::$_status = self::STATUS_RUNNING;`

设置启动状态

`pcntl_signal_dispatch();`

根据信号调用相关

```
$status = 0;
$pid    = pcntl_wait($status, WUNTRACED);
```

等待子进程退出信号

`pcntl_signal_dispatch();`

根据信号调用相关

`if ($pid > 0) {}`

子进程退出处理

```
else {
    // If shutdown state and all child processes exited then master proce
ss exit.
    if (self::$_status === self::STATUS_SHUTDOWN && !self::getAllWorkerPi
ds()) {
        self::exitAndClearAll();
    }
}
```

所有子进程退出后清理

3 总结

worker启动过程，完成启动主流程。

> 1 环境检测与初始化

```
    self::checkSapiEnv();
    self::init();
```

2 命令行解析

`self::parseCommand();`

3 主进程创建与运行

```
self::daemonize();
self::initWorkers();
self::installSignal();
self::saveMasterPid();
```

4 子进程创建与启动

`self::forkWorkers();`

5 监控子进程信号

```
self::displayUI();
self::resetStd();
self::monitorWorkers();
```

框架工具

- 另：Worker功能函数
- 另：Connection网络连接
- 另：Event事件监听
- 另：Protocols网络协议

另：Worker功能函数

1 成员列表

1 成员属性

```
;版本号
const VERSION = '3.3.1';

;启动状态，运行状态,停止状态，重载状态
const STATUS_STARTING = 1;
const STATUS_RUNNING = 2;
const STATUS_SHUTDOWN = 4;
const STATUS_RELOADING = 8;

;子进程强制关闭时间,默认baclog长度,udp数据包最大
const KILL_WORKER_TIMER_TIME = 2;
const DEFAUL_BACKLOG = 1024;
const MAX_UDP_PACKAGE_SIZE = 65535;

;进程id编号,进程名称,worker进程数量
public $id = 0;
public $name = 'none';
```

```
public $count = 1;

;进程用户，进程组
public $user = '';
public $group = '';

;是否可以重载，是否复用端口
public $reloadable = true;
public $reusePort = false;

;worker 3个回调接口
public $onWorkerStart = null;
public $onWorkerStop = null;
public $onWorkerReload = null;

;connect 6个回调接口
public $onConnect = null;
public $onMessage = null;
public $onClose = null;
public $onError = null;
public $onBufferFull = null;
public $onBufferDrain = null;

;传输层协议,应用层协议
public $transport = 'tcp';
public $protocol = '';

;所有连接
public $connections = array();

;自动加载根目录
protected $_autoloadRootPath = '';

;是否守护进程模式
public static $daemonize = false;

;输出文件,pid文件，日志文件
public static $stdoutFile = '/dev/null';
public static $pidFile = '';
public static $logFile = '';

; 全局事件循环
public static $globalEvent = null;

; 主进程id
protected static $_masterPid = 0;

; 监听socket
protected $_mainSocket = null;

; socket名称,socket上下文选项
protected $_socketName = '';
protected $_context = null;

;workers实例数组，worker进程id数组
protected static $_workers = array();
```

```
protected static $_pidMap = array();

;等待重启worker进程数组, worker的pid与进程编号映射
protected static $_pidsToRestart = array();
protected static $_idMap = array();

;当前状态
protected static $_status = self::STATUS_STARTING;

;workername最大长度,socketname最大长度,username最大长度
protected static $_maxWorkerNameLength = 12;
protected static $_maxSocketNameLength = 12;
protected static $_maxUserNameLength = 12;

;状态文件,启动文件
protected static $_statisticsFile = '';
protected static $_startFile = '';

;worker进程状态信息格式,
protected static $_globalStatistics

;可选事件循环,当前事件循环名称
protected static $_availableEventLoops
protected static $_eventLoopName

;内置协议
protected static $_builtinTransports
```

2 成员函数

```
1 setProcessTitle() 设置进程名称

2 initId()          初始化$_idMap

3 log()             日志记录

4 getSocketName()   获取socket名称

5 getCurrentUser()  获取当前用户

6 listen()          启动监听端口

7 signalHandler()   信号处理函数

8 stopAll()         停止所有

9 reload()          重载

10 writeStatisticsToStatusFile() 状态信息

11 getAllWorkerPids()  获取worker的pid

12 forkOneWorker()     创建一个worker进程
```

```
13 getId()              获取workerid
14 setUserAndGroup()    设置用户信息
15 run()             worker启动
16 getEventLoopName()  获取事件循环名称
17 reinstallSignal() worker信号处理注册
18 acceptConnection()  创建一个连接
19 acceptUdpConnection() upd数据包
20 stop()                worker子进程关闭
```

2 函数分析

1 setProcessTitle()

`protected static function setProcessTitle($title) 设置进程名称`

> $title:进程名称

调用cli_set_process_title()或者setproctitle()

2 initId()

`protected static function initId() 初始化$_idMap`

将workers的pid与workers的进程编号关联到$_idMap

3 log()

`protected static function log($msg) 日志信息`

> $msg:待记录信息

debug模式 直接输出
daemoniz模式 输出到日志文件

4 getSocketName()

`public function getSocketName() 获取socketname`

将第一个字母小写返回，或者返回none

5 getCurrentUser()

```
protected static function getCurrentUser() 当前进程的用户
```

调用posix_getpwuid() 获取当前进程的用户信息

6 listen()[重点]

`public function listen()` 启动端口监听

```
if (!$this->_socketName || $this->_mainSocket) {
            return;
        }
```

检查_socketName与_mainSocket参数。

```
Autoloader::setRootPath($this->_autoloadRootPath);
```

注册自动加载根目录

```
$local_socket = $this->_socketName;
list($scheme, $address) = explode(':', $this->_socketName, 2);
```

解析$_socketName为协议$scheme,和监听地址$address

```
if (!isset(self::$_builtinTransports[$scheme])) {
    $scheme         = ucfirst($scheme);
    $this->protocol = '\\Protocols\\' . $scheme;
    if (!class_exists($this->protocol)) {
        $this->protocol = "\\Workerman\\Protocols\\$scheme";
        if (!class_exists($this->protocol)) {
            throw new Exception("class \\Protocols\\$scheme not exist");
        }
    }
    $local_socket = $this->transport . ":" . $address;
} else {
    $this->transport = self::$_builtinTransports[$scheme];
}
```

应用层协议检测与初始化

协议有关 见 另 ：Protocols协议

`$flags  = $this->transport === 'udp' ? STREAM_SERVER_BIND : STREAM_`

协议标识字段获取

```
if ($this->reusePort) {
```

```
    stream_context_set_option($this->_context, 'socket', 'so_reuseport',
1);
}
```

端口复用选项设置

```
if ($this->transport === 'unix') {
    umask(0);
    list(, $address) = explode(':', $this->_socketName, 2);
    if (!is_file($address)) {
        register_shutdown_function(function () use ($address) {
            @unlink($address);
        });
    }
}


$this->_mainSocket = stream_socket_server($local_socket, $errno, $errmsg,
 $flags, $this->_context);
        if (!$this->_mainSocket) {
            throw new Exception($errmsg);
        }
```

Uinx套接字协议

首先检查$address是否是文件
然后创建unix套接字服务socket

```
if (function_exists('socket_import_stream') && $this->transport === 'tcp'
) {
    $socket = socket_import_stream($this->_mainSocket);
    @socket_set_option($socket, SOL_SOCKET, SO_KEEPALIVE, 1);
    @socket_set_option($socket, SOL_TCP, TCP_NODELAY, 1);
}

stream_set_blocking($this->_mainSocket, 0);
```

tcp协议处理

```
if (self::$globalEvent) {
    if ($this->transport !== 'udp') {
        self::$globalEvent->add($this->_mainSocket, EventInterface::EV_RE
AD, array($this, 'acceptConnection'));
    } else {
        self::$globalEvent->add($this->_mainSocket, EventInterface::EV_RE
```

```
AD,
            array($this, 'acceptUdpConnection'));
    }
}
```

注册事件监听器
事件相关见 另：Event事件

7 signalHandler()[重点]

`public static function signalHandler($signal) 主进程信号处理转发`

> $signal:待处理信号

根据$signal，分别调用stopAll(),reload()或者
writeStatisticsToStatusFile()

8 stopAll()[重点]

`public static function stopAll() 主进程stop信号处理`

```
self::$_status = self::STATUS_SHUTDOWN;
```

设置当前状态为关闭状态

```
if (self::$_masterPid === posix_getpid()) {
    self::log("Workerman[" . basename(self::$_startFile) . "] Stopping ..
.");
    $worker_pid_array = self::getAllWorkerPids();
    // Send stop signal to all child processes.
    foreach ($worker_pid_array as $worker_pid) {
        posix_kill($worker_pid, SIGINT);
        Timer::add(self::KILL_WORKER_TIMER_TIME, 'posix_kill', array($wor
ker_pid, SIGKILL), false);
    }
}
```

主进程关闭处理
记录关闭信息
获取所有worker子进程id
发送关闭信号到所有worker子进程

```
else {
```

```
    foreach (self::$_workers as $worker) {
        $worker->stop();
    }
    exit(0);
}
```

子进程关闭处理

直接关闭

9 reload()

`protected static function reload() 进程重载`

`if (self::$_masterPid === posix_getpid()) {}`

> 主进程重载

```
if (self::$_status !== self::STATUS_RELOADING && self::$_status !== self:
:STATUS_SHUTDOWN) {
                self::log("Workerman[" . basename(self::$_startFile) . "]
 reloading");
                self::$_status = self::STATUS_RELOADING;
            }
```

设置当前状态为重载状态

```
$reloadable_pid_array = array();

foreach (self::$_pidMap as $worker_id => $worker_pid_array) {
    $worker = self::$_workers[$worker_id];
    if ($worker->reloadable) {
        foreach ($worker_pid_array as $pid) {
            $reloadable_pid_array[$pid] = $pid;
        }
    } else {
        foreach ($worker_pid_array as $pid) {
            // Send reload signal to a worker process which reloadable is
 false.
            posix_kill($pid, SIGUSR1);
        }
    }
}
```

发送reload信号到子进程

`self::$_pidsToRestart = array_intersect(self::$_pidsToRestart, $rel`

获取所有等待重载进程id

```
if (empty(self::$_pidsToRestart)) {
        if (self::$_status !== self::STATUS_SHUTDOWN) {
            self::$_status = self::STATUS_RUNNING;
        }
        return;
    }
```

检测是否完全重载

```
$one_worker_pid = current(self::$_pidsToRestart);
posix_kill($one_worker_pid, SIGUSR1);
```

发送reload信号到没有完成重载的子进程

```
$worker = current(self::$_workers);

if ($worker->onWorkerReload) {
    try {
        call_user_func($worker->onWorkerReload, $worker);
    } catch (\Exception $e) {
        echo $e;
        exit(250);
    }
}

if ($worker->reloadable) {
    self::stopAll();
}
```

> 子进程重载

回调onWorkerReload函数
不可重载则关闭所有

10 protected static function writeStatisticsToStatusFile()

`if (self::$_masterPid === posix_getpid()) {}`
主进程状态信息保存

子进程状态新保存

20 stop()

`public function stop()` 关闭worker子进程

回调onWorkerStop
移除监听器
关闭socket描述符

3 函数关系

另：Connection网络连接

1 网络连接文件

```
Workerman/Connection/
    ConnectionInterface.php  ;网络连接接口
    TcpConnection.php        ;Tcp网络连接
    UpdConnection.php        ;Udp网络连接
    AsyncTcpConnection.php   ;异步Tcp网络连接
```

2 网络连接接口(ConnectionInterface.php)

```
  ;连接统计:连接总数,请求总数,异常总数,发送失败总数
  public static $statistics = array(
        'connection_count' => 0,
        'total_request'    => 0,
        'throw_exception'  => 0,
        'send_fail'        => 0,
    );

  ;接受数据回调,连接关闭回调,发生错误回调
  public $onMessage = null;
  public $onClose = null;
  public $onError = null;
```

```
;连接发送数据接口
abstract public function send($send_buffer);
;获取客户端Ip接口
abstract public function getRemoteIp();
;获取客户端端口接口
abstract public function getRemotePort();
;关闭连接接口
abstract public function close($data = null);
```

3 Tcp网络连接(TcpConnection.php)

> 成员变量
>
> ```
>   ;数据读取缓存区上限
> ```

```
    const READ_BUFFER_SIZE = 65535;
    ;连接状态:建立连接,完成建立,正在关闭,关闭完成
    const STATUS_CONNECTING = 1;
    const STATUS_ESTABLISH = 2;
    const STATUS_CLOSING = 4;
    const STATUS_CLOSED = 8;
```

;连接回调接口:接受数据,连接关闭,发生错误,数据满,数据发送
public $onMessage = null;
public $onClose = null;
public $onError = null;
public $onBufferFull = null;
public $onBufferDrain = null;

;应用层协议,所属worker进程,连接id,所属worker的id
public $protocol = null;
public $worker = null;
public $id = 0;
protected $_id = 0;

;数据包发送缓存最大上限，数据包发送缓存默认上限，
public $maxSendBufferSize = 1048576;
public static $defaultMaxSendBufferSize = 1048576;

;最大可发送数据包
public static $maxPackageSize = 10485760;

protected static $_idRecorder = 1;

;socket套接符,发送缓存，接受缓存,当前数据包大小
protected $_socket = null;
protected $_sendBuffer = '';
protected $_recvBuffer = '';
protected $_currentPackageLength = 0;

;当前链接状态，客户端地址，是否阻塞
protected $_status = self::STATUS_ESTABLISH;
protected $_remoteAddress = '';
protected $_isPaused = false;

```
>[info] 成员方法

### __construct($socket, $remote_address = ''):Tcp连接构造函数
```

public function __construct($socket, $remote_address = '')

```
> $socket: 监听套接字
> $remote_address:客户端地址

* * * * *

### send($send_buffer, $raw = false):tcp数据发送接口
```

public function send($send_buffer, $raw = false)

```
> $send_buffer:发送内容
> $raw:??

* * * * *

### getRemoteIp():获取客户端ip
```

public function getRemoteIp()

```
### getRemotePort():获取客户端端口
```

public function getRemotePort()

```
### pauseRecv():阻塞读数据
```

public function pauseRecv()

```
### resumeRecv():恢复读数据
```

public function resumeRecv()

```
### baseRead($socket, $check_eof = true):tcp读数据
```

public function baseRead($socket, $check_eof = true)

```
> $socket:socket描述符
> $check_eof:??

* * * * *

### baseWrite():tcp写数据
```

public function baseWrite()

```
### pipe($dest):数据重定向
```

public function pipe($dest)

```
> $dest:重定向目的

* * * * *

### consumeRecvBuffer($length):截取数据接收区
```

public function consumeRecvBuffer($length)

```
> $length:截取长度

* * * * *

### close($data = null):tcp连接关闭接口
```

public function close($data = null)

```
> $data:关闭时发送数据

* * * * *

### destroy():tcp链接注销接口
```

public function destroy()

```
### getSocket():获取连接socket描述符
```

public function getSocket()

```
### checkBufferIsFull():检测发送缓冲区是否达到上限
```

protected function checkBufferIsFull()

```
## 4 Udp网络连接(UpdConnection.php)
>[info] 成员变量
```

;udp应用层协议
public $protocol = null;
;udp连接socket描述
protected $_socket = null;
;客户端ip
protected $_remoteIp = '';
;客户端端口
protected $_remotePort = 0;
;客户端地址
protected $_remoteAddress = '';

```
>[info] 成员方法

### __construct($socket, $remote_address):Udp构造函数
`public function __construct($socket, $remote_address)`

### send($send_buffer, $raw = false):Udp连接数据发送接口
`public function send($send_buffer, $raw = false)`

### getRemoteIp():获取客户端ip
`public function getRemoteIp()`

### getRemotePort():获取客户端端口
`public function getRemotePort()`
```

```
### close($data = null):连接关闭
`public function close($data = null)`

## 5 异步Tcp网络连接(AsyncTcpConnection.php)

>[info] 成员变量
```

;连接回调
public $onConnect = null;
;连接默认状态
protected $_status = self::STATUS_CONNECTING;
;客户端主机
protected $_remoteHost = '';

```
>[info] 成员方法

###  __construct($remote_address):异步Tcp构造函数
public function __construct($remote_address)

### connect():建立 异步Tcp连接
public function connect()

### getRemoteHost():获取客户端主机
public function getRemoteHost()

### emitError($code, $msg):手动触发错误回调
protected function emitError($code, $msg)

### checkConnection($socket):检查连接是否建立
public function checkConnection($socket)

## 4 网络连接总结
网络连接实现服务器与客户端的对应关系。
```

ConnectionInterface.php 网络连接抽象
Tcp 连接网络连接
Udp 数据包网络连接
异步Tcp 异步连接

另：Event事件监听

1 事件文件

```
Workerman\Events\
    EventInterface.php    ;事件接口
    Ev.php                ;EV事件机制
    Event.php             ;Event事件机制
    Libvent.php           ;Libevent事件机制
    Select.php            ;Select事件机制
```

2 事件接口(EventInterface.php)

> 成员变量

```
;读事件,写事件,信号事件,周期事件,定时事件
const EV_READ = 1;
const EV_WRITE = 2;
const EV_SIGNAL = 4;
const EV_TIMER = 8;
const EV_TIMER_ONCE = 16;
```

> 成员方法

add()注册事件

public function add($fd, $flag, $func, $args = null);

> $fd:文件描述符
> $flag:事件类型
> $func:事件处理接口
> $args:参数

del():注销事件

public function del($fd, $flag);

> $fd:文件描述符
> $flag:事件类型

clearAllTimer():移除所有定时器事件
public function clearAllTimer();

loop():开启事件循环
public function loop();

3 Ev事件机制(Ev.php)

> 成员变量

```
;所有读写事件,信号事件, 定时器事件
protected $_allEvents = array();
protected $_eventSignal = array();
protected $_eventTimer = array();
;定时器id
protected static $_timerId = 1;
```

> 成员方法

add($fd, $flag, $func, $args = null)注册事件
```
public function add($fd, $flag, $func, $args = null)
```

> $fd:文件描述符
> $flag:事件类型
> $func:事件回调函数
> $args:参数

del($fd, $flag) 注销事件
```
public function del($fd, $flag)
```

> $fd:文件描述符
> $flag:事件类型

timerCallback($event) 定时器回调接口

```
public function timerCallback($event)
```

> $event:事件参数

clearAllTimer() 移除所有定时器
```
public function clearAllTimer()
```

loop() 开启事件循环
```
public function loop()
```

4 Event事件机制(Event.php)

> 成员变量

```
;事件对象
protected $_eventBase = null;
;读写事件,信号事件,定时器事件
protected $_allEvents = array();
protected $_eventSignal = array();
protected $_eventTimer = array();
;定时器id
protected static $_timerId = 1;
```

> 成员方法

__construct()：构造函数
```
public function __construct()
```

add($fd, $flag, $func, $args=array())：注册事件
```
public function add($fd, $flag, $func, $args=array())
```

del($fd, $flag)：注销事件
```
public function del($fd, $flag)
```

timerCallback($fd, $what, $param)：定时器回调
```
public function timerCallback($fd, $what, $param)
```

clearAllTimer()：清除所有定时
public function clearAllTimer()

loop()：开启事件循环
```
public function loop()
```

5 Livevent事件机制

> 成员变量

```
;事件对象
protected $_eventBase = null;
;读写事件,信号事件,定时器事件
protected $_allEvents = array();
protected $_eventSignal = array();
protected $_eventTimer = array();
```

> 成员方法

__construct()：构造函数

```
public function __construct()
```

add($fd, $flag, $func, $args = array())：注册事件

```
public function add($fd, $flag, $func, $args = array())
```

del($fd, $flag：注销事件

```
public function del($fd, $flag)
```

timerCallback($_null1, $_null2, $timer_id)：定时器回调

```
protected function timerCallback($_null1, $_null2, $timer_id)
```

clearAllTimer()：清除定时器事件

```
public function clearAllTimer()
```

loop()：开启事件循环

```
public function loop()
```

6 Select事件机制

> 成员变量

```
;读写事件,信号事件
public $_allEvents = array();
public $_signalEvents = array();
;读描述符,写描述符
protected $_readFds = array();
protected $_writeFds = array();
;定时器调度栈,定时器监听,
protected $_scheduler = null;
protected $_task = array();
;定时器id
protected $_timerId = 1;
```

```
;延时
protected $_selectTimeout = 100000000;
;socket管道
protected $channel = array();
```

> 成员方法

__construct()：构造函数
public function __construct()

add($fd, $flag, $func, $args = array())：注册事件
public function add($fd, $flag, $func, $args = array())

signalHandler($signal)：信号处理接口
public function signalHandler($signal)

del($fd, $flag)：注销事件
public function del($fd, $flag)

tick()：定时器调度
protected function tick()

clearAllTimer()：清除定时器
public function clearAllTimer()

loop()：开启事件循环
public function loop()

7 事件总结
事件机制实现对socket描述的事件回调处理

EventInterface.php 事件接口

Ev.php libev事件接口

Event.php Event扩展事件接口

Libevent.php Livevent扩展事件接口

Select.php Select事件接口

有关事件扩展机制 见基础原理 事件循环

另：Protocols网络协议

- 8 应用协议总结

1 网络协议文件

```
Workerman\Protocols\
    ProtocolInterface.php   ;协议接口
    Frame.php               ;frame应用层协议
    Http.php                ;http应用层协议
    Text.php                ;text应用层协议
    Websocket.php           ;websocket应用层协议
    Ws.php                  ;websocket客户端协议
```

2 网络协议接口(ProtocolInterface.php)

> 成员方法

input():检查数据包完整性

```
public static function input($recv_buffer, ConnectionInterface $connectio
n);
```

decode():解码数据包数据回调onMessage

```
public static function decode($recv_buffer, ConnectionInterface $connecti
on);
```

encode():编码数据包协议等待发送

```
public static function encode($data, ConnectionInterface $connection);
```

3 frame应用层协议(Frame.php)

> 成员方法

input():Frame数据格式检查完整性

```
public static function input($buffer, TcpConnection $connection)
```

decode($buffer):Frame数据包解码

```
public static function decode($buffer)
```

encode($buffer):Frame数据包编码

```
public static function encode($buffer)
```

4 http应用层协议(Http.php)

> 成员方法

input():Http数据包格式检查完整性

```
public static function input($recv_buffer, TcpConnection $connection)
```

decode():Http数据包解码

```
public static function decode($recv_buffer, TcpConnection $connection)
```

encode():Http数据包编码

```
public static function encode($content, TcpConnection $connection)
```

5 text应用层协议(Text.php)

> 成员方法

input():Text数据包格式检查完整性

public static function input($buffer, TcpConnection $connection)

encode():encode数据包编码

public static function encode($buffer)

decode():decode数据包解码

public static function decode($buffer)

6 Websocket应用层协议(Websocket.php)

> 成员方法

input():Text数据包格式检查完整性

```
public static function input($buffer, ConnectionInterface $connection)
```

encode():encode数据包编码

```
public static function encode($buffer, ConnectionInterface $connection)
```

decode():decode数据包解码

```
public static function decode($buffer, ConnectionInterface $connection)
```

dealHandshake()握手处理

```
protected static function dealHandshake($buffer, $connection)
```

parseHttpHeader()解析http头

```
protected static function parseHttpHeader($buffer)
```

7 Ws客户端协议(Ws.php)

> 成员变量

```
;协议最小头部
const MIN_HEAD_LEN = 2;

;blob格式
const BINARY_TYPE_BLOB = "\x81";

;array格式
const BINARY_TYPE_ARRAYBUFFER = "\x82";
```

> 成员方法

input() 数据包完整性检查

```
public static function input($buffer, $connection)
```

encode() 数据包编码

```
public static function encode($payload, $connection)
```

decode() 数据包解码

```
public static function decode($bytes, $connection)
```

sendHandshake()发送握手协议

```
public static function sendHandshake($connection)
```

dealHandshake()处理握手协议

```
public static function dealHandshake($buffer, $connection)
```

8 应用协议总结

应用协议负责数据包数据的格式解析。

框架驱动

网络连接

事件机制

1 Ev事件处理器

1 扩展简介

```
内置事件处理器，无需安装
```

2 事件接口

final Ev{} 事件处理器核心

> Ev::run()
>
> ```
> final public static void  Ev::run  ([ int $flags  ] )
> ```
>
> 启动事件循环，等待注册的事件监视器相应事件发生
> 事件发生后，自动调用事件监视器的回调函数

class EvSignal{} 信号事件监视器

```
public EvSignal::__construct  ( int $signum  , callable  $callback  [, mi
xed  $data  = NULL    [, int $priority  = 0  ]] )
```

创建信号事件监视器并自动启动

class EvTimer{} 定时器事件监视器

```
public EvTimer::__construct  ( double $after  , double $repeat  , callabl
e  $callback  [, mixed  $data  = NULL    [, int $priority  = 0  ]] )
```

创建定时器事件监视器并自动启动

class EvIo{} IO事件监视器

```
public EvIo::__construct  ( mixed  $fd  , int $events  , callable  $callb
ack  [, mixed  $data  [, int $priority  ]] )
```

创建IO事件监视器并自动启动

2 Event事件处理器

1 扩展简介

安装libevent扩展

2 事件接口

final EventBase {} 事件循环体

```
public bool EventBase::loop  ([ int $flags  ] )
```

启动事件循环，等待事件发生

final Event {} 事件监视器

```
public Event::__construct  ( EventBase  $base  , mixed  $fd  , int $what
 , callable  $cb  [, mixed  $arg  = NULL  ] )
```

创建各类事件监视器

```
public static Event  Event::signal  ( EventBase  $base  , int $signum  ,
callable  $cb  [, mixed  $arg  ] )
```

创建信号事件监视器

3 Livevent事件处理器

1 扩展简介

```
需要安装libevent扩展
```

2 事件接口

event_base_new() 创建事件循环

```
resource event_base_new  ( void )
```

event_new() 创建事件监视器

```
resource event_new  ( void )
```

event_set() 设置事件监视器

```
bool event_set  ( resource $event  , mixed  $fd  , int $events  , mixed
$callback  [, mixed  $arg  ] )
```

event_base_set() 注册事件监视器到事件循环

```
bool event_base_set  ( resource $event  , resource $event_base  )
```

event_add() 添加事件监视器

```
bool event_add  ( resource $event  [, int $timeout  = -1  ] )
```

event_del() 删除事件监视器

```
bool event_del  ( resource $event  )
```

event_base_loop() 启动事件循环

```
int event_base_loop  ( resource $event_base  [, int $flags  = 0  ] )
```

4 Select事件处理器

数据协议

使用范例

基础原理

基础函数

1 spl_object_hash()
2 debug_backtrace()
返回以下信息

名称	类型	描述
function	字符	当前的函数名
line	整数	当前的行号
file	字符串	当前的文件名
class	字符串	当前的类名
object	对象	当前对象
type	字符串	当前的调用类型，可能的调用:"->" - 方法调用;"::" - 静态方法调用; nothing - 函数调用
args	数组	如果在函数中，列出函数参数。如果在被引用的文件中，列出被引用的文件名

进程操作

1 PCNTL进程控制

1 简介

进程控制PCNTL实现了unix方式的进程操作

不能用于web服务器环境模式

只使用于unix平台

2 使用

```
<?php
//定义ticks
declare( ticks = 1 );

 //产生子进程分支
$pid  =  pcntl_fork ();
if ( $pid  == - 1 ) {
     //pcntl_fork返回-1标明创建子进程失败
     die( "could not fork" );
 } else if ( $pid ) {
     //父进程中pcntl_fork返回创建的子进程进程号
     exit();
 } else {
      // 子进程pcntl_fork返回的时0
 }

 // 从当前终端分离
if ( posix_setsid () == - 1 ) {
    die( "could not detach from terminal" );
```

```
}

 // 安装信号处理器
 pcntl_signal ( SIGTERM ,  "sig_handler" );
 pcntl_signal ( SIGHUP ,  "sig_handler" );

 // 执行无限循环任务
while ( 1 ) {

 // 等待用户操作请求

}

function  sig_handler ( $signo )
{

     switch ( $signo ) {
         case  SIGTERM :
              // 处理中断信号
              exit;
             break;
         case  SIGHUP :
              // 处理重启信号
              break;
         default:
              // 处理所有其他信号
      }

}

 ?>
```

3 函数接口

pcntl_fork() 创建子进程

```
int pcntl_fork  ( void )
```

父进程返回子进程pid>0

子进程返回0

创建失败父进程上下文返回-1

pcntl_singal() 安装信号处理器

```
bool pcntl_signal  ( int $signo  , callback  $handler  [, bool $restart_s
yscalls  = true  ] )
```

> $signo:信号
> $handler:回调处理

为指定的信号创建回调处理函数

pcntl_signal_dispatch() 分发信号
将接受到的信号进行广播分发

pcntl_wait() pcntl_waitpid()等待进程退出

```
int pcntl_wait  ( int &$status  [, int $options  = 0  ] )
```

> $status: 获取子进程退出的状态信息
> $options:
> >> WNOHANG 如果没有子进程退出立刻返回。
> >> WUNTRACED 子进程已经退出并且其状态未报告时返回。

挂起当前进程直到一个子进程退出
或接受到一个信号要求中断当前进程
或调用一个信号处理函数

```
int pcntl_waitpid  ( int $pid  , int &$status  [, int $options  = 0
```

> $pid: 等待进程id

>> [

> [-1] 等待任意子进程;与pcntl_wait函数行为一致。
> [0] 等待任意与调用进程组ID相同的子进程。
> [> 0] 等待进程号等于参数pid值的子进程。

挂起当前进程直到pid指定的进程号的进程退出
或接受到一个信号要求中断当前进程
或调用一个信号处理函数

pcntl_exex() 当前进程执行指定的程序

```
void  pcntl_exec  ( string $path  [, array $args  [, array $envs  ]
```

> $path: 可执行程序路径
> $args: 参数数组
> $envs: 环境变量数组

当前进程运行指定可执行文件

pcntl_alarm() 设置一个alarm闹钟信号
`int pcntl_alarm  ( int $seconds  )`

> $seconds:定时时间秒数

在指定的秒数后向进程发送一个 SIGALRM 信号
每次对 pcntl_alarm() 的调用都会取消之前设置的alarm信号
可以用来实现计时器

返回alarm调度剩余的描述，没有alarm调度返回0

2 POSIX标准接口

> 当前进程操作

```
    posix_getpid()
          返回当前进程的进程id
        posix_getppid()
          返回当前进程的父进程id
        posix_getpgrp()
          返回当前进程的进程组id

        posix_getgid()
          返回当前进程的gid
        posix_setgid()
          设置当前进程的gid
        posix_getgroups()
          返回当前进程的进程组集合
```

> 当前进程其他操作

```
    posix_times()
          返回当前进程cpu耗时
    posix_getuid()
      返回当前进程的用户id

        posix_getegid()
          返回当前进程的有效gid
    posix_setegid()
      设置当前进程的有效gid
    posix_geteuid()
```

```
      返回当前进程的有效uid
    posix_seteuid()
      设置当前进程的有效uid

    posix_getpgid()
      获取进程的pgid
    posix_setpgid()
      设置进程的pgid

    posix_getsid()
      获取进程的当前会话id
    posix_setsid()
      设置进程的当前会话id

        posix_getgrgid()
          获取进程组的信息
        posix_getgrnam()
          获取进程组的信息
        posix_getpgid()
          返回进程的进程组
        posix_getsid()
          返回进程的会话id
```

用户信息操作

```
posix_getlogin()
      获取登录用户名
posix_getpwnam()
      获取指定用户信息
posix_getwuid()
      获取指定用户信息
```

文件操作

```
posix_access()
      设置文件权限模式
posix_ctermid()
      获取控制器终端当前路径

posix_getcwd()
      获取当前目录信息

posix_mkfifo()
posix_mknod()
```

其他操作

posix_kill() 发送sig信号到pid进程
posix_uname() 获取系统名称
posix_ttyname() 设置终端设备名称
posix_strerror() 输出错误代号
posix_getrlimit() 获取系统资源限制

```
## 3 执行外部程序
```

exec()
passthru()
shell_exec()
system()

proc_open()
proc_terminate()
proc_nice()
proc_get_status()
proc_close()

数据流操作

操作接口

```
stream_context_create()
        创建返回资源流的上下文
stream_context_set_option()
        资源流，数据包或者上下文设置选项参数
stream_socket_server()
        创建服务器的socket接口
stream_set_blocking()
        开启与关闭资源流阻塞模式
stream_socket_accept()
        等待服务器socket接口的连接
stream_socket_recvfrom()
        获取socket的数据信息
```

框架心得

框架结构

服务器结构