

workerman手冊

walkor



目 录

序言

原理

开发必读

入门指引

特性

简单的开发示例

安装

环境要求

下载安装

启动停止

开发流程

开发前必读

目录结构

开发规范

基本流程

通讯协议

通讯协议作用

定制通讯协议

一些例子

Worker类

构造函数

属性

id

count

name

protocol

transport

reusePort

connections

stdoutFile

pidFile

logFile

user

reloadable

daemonize

globalEvent

回调属性

onWorkerStart

onWorkerReload

onConnect

onMessage

onClose

onBufferFull

onBufferDrain

onError

接口

runAll

stopAll

listen

TcpConnection类

属性

id

protocol

worker

maxSendBufferSize

defaultMaxSendBufferSize

maxPackageSize

回调属性

onMessage

onClose

onBufferFull

onBufferDrain

onError

接口

send

getRemoteIp

getRemotePort

close

destroy

pauseRecv

- resumeRecv
 - pipe
- AsyncTcpConnection类
 - 构造函数
 - connect
 - reconnect
 - transport
- Timer定时器类
 - add
 - del
 - 定时器注意事项
- WebServer
- 调试
 - 基本调试
 - status命令查看运行状态
 - 网络抓包
 - 跟踪系统调用
- 常用组件
 - GlobalData数据共享组件
 - GlobalDataServer
 - GlobalDataClient
 - add
 - cas
 - increment
 - Channel分布式通讯组件
 - ChannelServer
 - channelClient
 - connect
 - on
 - publish
 - unsubscribe
 - 例子-集群推送
 - 例子-分组发送
 - FileMonitor文件监控组件
 - MySQL组件
 - workerman/mysql

- react/mysql(异步)
- redis组件
 - react/redis
- 异步http组件
 - react/http-client
- 异步消息队列组件
 - react/zmq
 - react/stomp
- 异步dns组件
 - react/dns
- 进程控制组件
 - react/child-process
- memcache
- 常见问题
 - 心跳
 - 客户端连接失败原因
 - 是否支持多线程
 - 与其它框架整合
 - 运行多个workerman
 - 支持哪些协议
 - 如何设置进程数
 - 查看客户端连接数
 - 对象和资源的持久化
 - 例子无法工作
 - 启动失败
 - 停止失败
 - 支持多少并发
 - 更改代码不生效
 - 向指定客户端发送数据
 - 如何主动推送消息
 - 在其它项目中推送
 - 如何实现异步任务
 - status里send_fail的原因
 - Windows下开发Linux下部署
 - 是否支持socket.io
 - 终端关闭导致workerman关闭

[与nginx apache的关系](#)

[禁用函数检查](#)

[平滑重启原理](#)

[为Flash开843端口](#)

[如何广播数据](#)

[如何建立udp服务](#)

[监听ipv6](#)

[关闭未认证的连接](#)

[传输加密-ssl/tsl](#)

[创建wss服务](#)

[创建https服务](#)

[workerman作为客户端](#)

[作为ws/wss客户端](#)

[PHP的几种回调写法](#)

[附录](#)

[Linux内核调优](#)

[压力测试](#)

[安装扩展](#)

[websocket协议](#)

[ws协议](#)

[text协议](#)

[frame协议](#)

[不支持的函数/特性](#)

[版权信息](#)

序言

序言

Workerman，让你看到PHP不为人知的一面。

Workerman是什么？

Workerman是一款纯PHP开发的开源高性能的PHP socket 服务框架。

Workerman不是重复造轮子，它不是一个MVC框架，而是一个更底层更通用的socket服务框架，你可以用它开发tcp代理、梯子代理、做游戏服务器、邮件服务器、ftp服务器、甚至开发一个php版本的redis、php版本的数据库、php版本的nginx、php版本的php-fpm等等。Workerman可以说是PHP领域的一次创新，让开发者彻底摆脱了PHP只能做WEB的束缚。

实际上Workerman类似一个PHP版本的nginx，核心也是多进程+Epoll+非阻塞IO。

Workerman每个进程能维持上万并发连接。由于本身常驻内存，不依赖Apache、nginx、php-fpm这些容器，拥有超高的性能。同时支持TCP、UDP、UNIXSOCKET，支持长连接，支持Websocket、HTTP、WSS、HTTPS等通讯协议以及各种自定义协议。拥有定时器、异步socket客户端、异步Mysql、异步Redis、异步Http、异步消息队列等众多高性能组件。

Workerman的一些应用方向

Workerman不同于传统MVC框架，Workerman不仅可以用于Web开发，同时还有更广阔的应用领域，例如即时通讯类、物联网、游戏、服务治理、其它服务器或者中间件，这无疑大大提高了PHP开发者的视野。目前这些领域的PHP开发者奇缺，如果想在PHP领域有自己的技术优势，不满足于每天的增删改查工作，或者想向架构师方向或者技术大牛的方向发展，Workerman都是非常值得学习的框架。建议开发者不仅会用，而且能基于Workerman开发出属于自己的开源项目，提升技能增加自己的影响力，比如[Beanbun多进程网络爬虫框架](#)就是一个很好的例子，刚刚上线不久就获得众多好评。

Workerman的一些应用方向如下：

1、即时通讯类

例如网页即时聊天、即时消息推送、微信小程序、手机app消息推送、PC软件消息推送等等
[[示例 workerman-chat聊天室](<http://www.workerman.net/workerman-chat>)、[web消息推送](<http://www.workerman.net/web-sender>)、[小蝌蚪聊天室]

(<http://www.workerman.net/workerman-todpole>)]

2、物联网类

例如Workerman与打印机通讯、与单片机通讯、智能手环、智能家居、共享单车等等。
[客户案例如 易联云、易泊时代等]

3、游戏服务器类

例如棋牌游戏、MMORPG游戏等等。[[示例 browserquest-php]
(<http://www.workerman.net/browserquest>)]

4、SOA服务化

利用Workerman将现有业务不同功能单元封装起来，以服务的形式对外提供统一的接口，达到系统松耦合、易维护、高可用、易伸缩。[[示例 workerman-json-rpc]
(<http://www.workerman.net/workerman-jsonrpc>)、[workerman-thrift]
(<http://www.workerman.net/workerman-thrift>)]

5、其它服务器软件

例如 [GatewayWorker](#) , [PHPSocket.IO](#) , [http代理](#) , [sock5代理](#) , [分布式通讯组件](#) , [分布式变量共享组件](#) , [消息队列](#)、DNS服务器、WebServer、CDN服务器、FTP服务器等等

6、中间件

例如 [异步MySQL组件](#) , [异步redis组件](#) , [异步http组件](#) , [异步消息队列组件](#) , [异步dns组件](#) , [文件监控组件](#) , 还有很多第三方开发的组件框架等等

显然传统的mvc框架很难实现以上的功能，所以也就是workerman诞生的原因。

Workerman理念

极简、稳定、高性能、分布式。

极简

小即是美，Workerman内核极简，仅有几个php文件并且只暴露几个接口，学习起来非常简单。所有其它功能通过组件的方式扩展。

Workerman拥有完善的文档+权威的主页+活跃的社区+数个千人QQ群+众多的高性能组件+N多的例子，这一切都让开发者使用起来更得心应手。

稳定

Workerman已经开源数年，被很多上市公司大规模使用，超级稳定。有些服务2年多没重启过仍然在飞速运行。没有coredump、没有内存泄漏、没有bug。

高性能

Workerman因为常驻内存，本身不依赖apache/nginx/php-fpm，没有容器到PHP的通讯开销，没有每个请求初始化一切又销毁一切的开销，具有超高的性能，比起传统的MVC框架，性能要高数十倍，PHP7下通过ab压力测试QPS甚至高于单独的nginx。

分布式

现在早已经不是单枪匹马的时代了，单台服务器性能再强悍也有极限，分布式多服务器部署才是王道。Workerman直接提供了一套长连接分布式通讯方案[GatewayWorker框架](#)，加服务器只需要简单配置下然后启动即可，业务代码零更改，系统承载能力成倍增加。如果你是开发TCP长连接应用，建议直接用[GatewayWorker](#)，它是对Workerman的一个包装，针对长连接应用提供了更丰富的接口以及强悍的分布式处理能力。

本手册作用范围

WorkerMan有分为Linux版本[WorkerMan](#)和Windows版本[WorkerMan-for-win](#)，windows版本说明参见[这里](#)。Linux版本可用于开发调试及正式环境部署，而由于PHP-CLI在windows系统无法实现多进程以及守护进程，所以windows版本Workerman建议仅作开发调试使用。

注意：Windows版本WorkerMan无法在Linux平台使用，同时Linux版本WorkerMan也无法在Windows平台使用。

windows用户（必读）

windows用户需要使用windows版本的workerman，windows版本workerman本身不依赖任何扩展，只需要配置好PHP环境变量即可，windows版本workerman安装及注意事项参见[windows用户必看](#)。

客户端

WorkerMan的通信协议是开放的，又是可定制的，因此，理论上WorkerMan可以与使用任意协议的任意平台的客户端进行通信。当用户开发客户端时，可以根据相应的通信协议完成与服务端的通信。

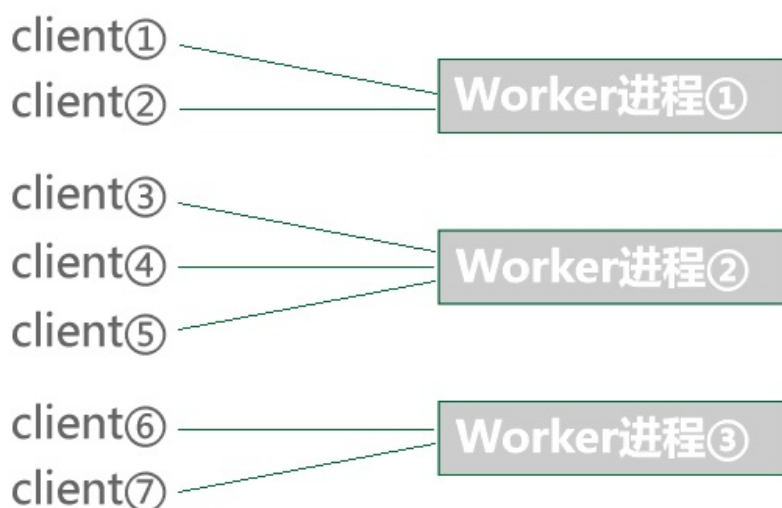
原理

原理

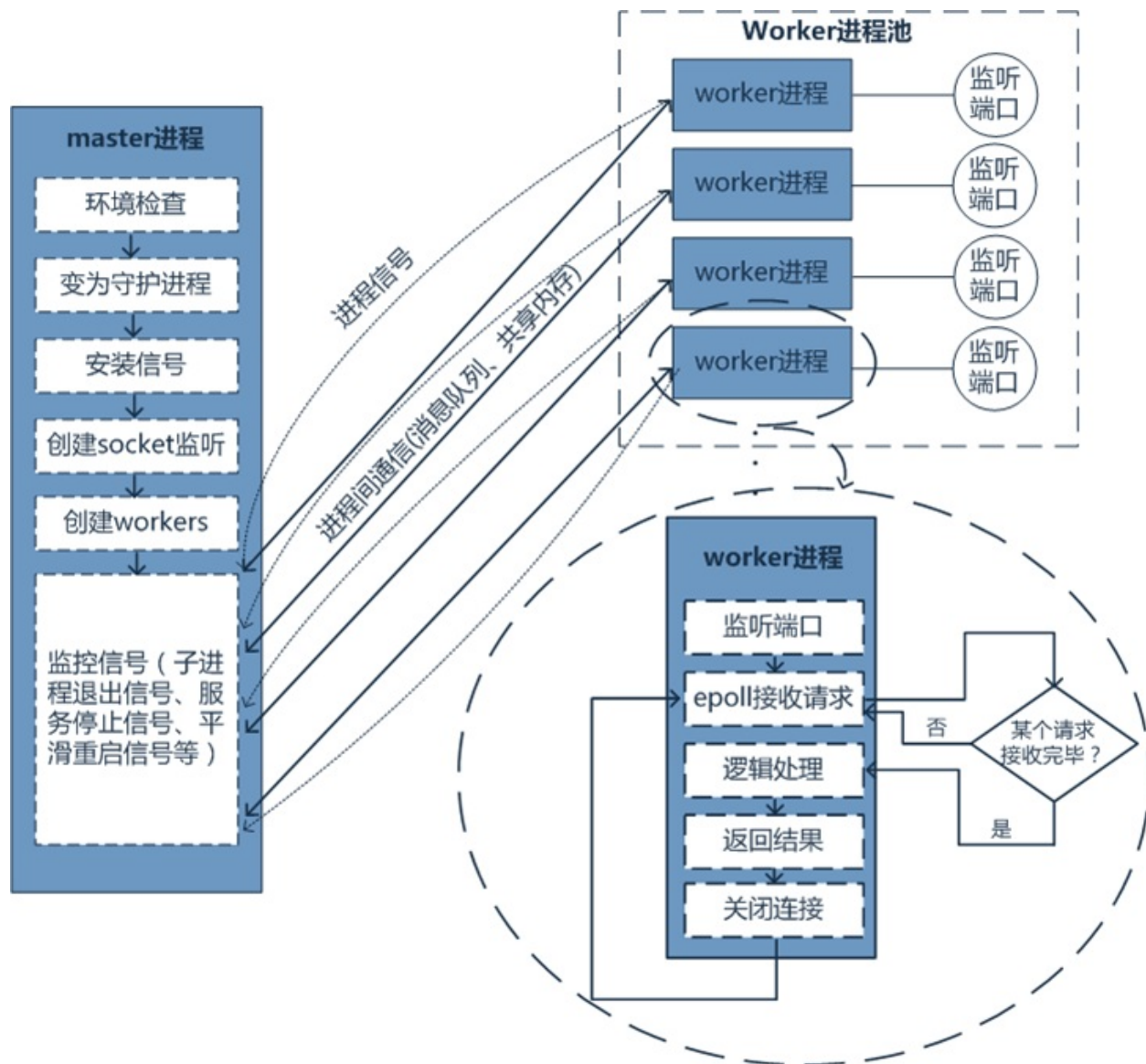
Worker说明

Worker是WorkerMan中最基本容器，Worker可以开启多个进程监听端口并使用特定协议通讯，类似nginx监听某个端口。每个Worker进程独立运作，采用Epoll(需要装event扩展)+非阻塞IO，每个Worker进程都能上万的客户端连接，并处理这些连接上发来的数据。主进程为了保持稳定性，只负责监控子进程，不负责接收数据也不做任何业务逻辑。

客户端与worker进程的关系



主进程与worker子进程关系



特点：

从图上我们可以看出每个Worker维持着各自的客户端连接，能够方便的实现客户端与服务端的实时通讯，基于这种模型我们可以方便实现一些基本的开发需求，例如HTTP服务器、Rpc服务器、一些智能硬件实时上报数据、服务端推送数据、游戏服务器、微信小程序后台等等。

开发必读

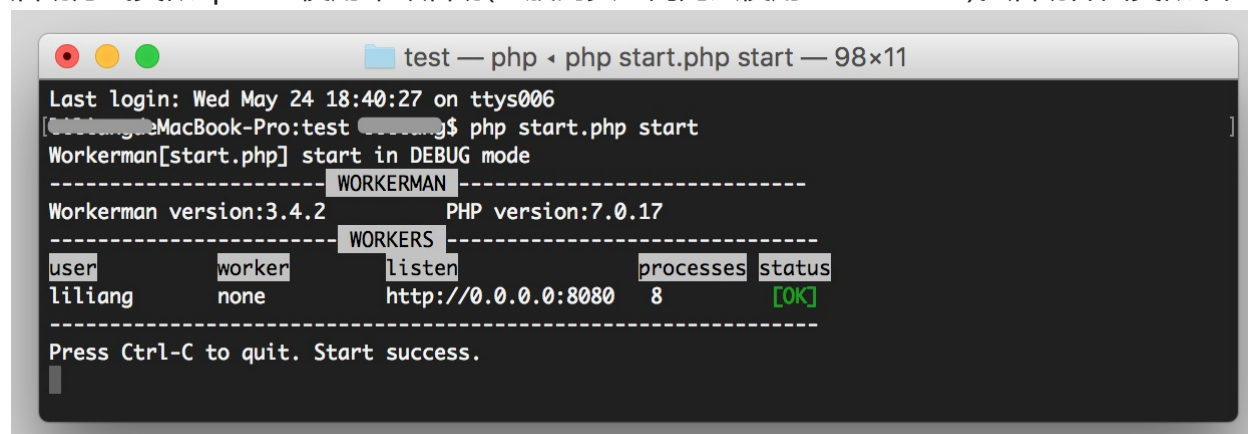
workerman开发者必须知道的几个问题

1、workerman不依赖apache或者nginx

workerman本身已经是一个类似apache/nginx的容器，只要[PHP环境OK](#) workerman就可以运行。

2、workerman是命令行启动的

启动方式类似apache使用命令启动(一般网页空间无法使用workerman)。启动界面类似下面



```
test — php — php start.php start — 98x11
Last login: Wed May 24 18:40:27 on ttys006
MacBook-Pro:test ~$ php start.php start
Workerman[start.php] start in DEBUG mode
----- WORKERMAN -----
Workerman version:3.4.2      PHP version:7.0.17
----- WORKERS -----
user      worker  listen      processes  status
liliang   none     http://0.0.0.0:8080  8          [OK]
-----
Press Ctrl-C to quit. Start success.
```

3、长连接必须加心跳

长连接必须加心跳，长连接必须加心跳，长连接必须加心跳，重要的话说三遍。

长连接长时间不通讯肯定会被防火墙干掉而断开。不加心跳的长连接应用就等着老板KO你吧。

[workerman心跳说明](#)、[gatewayWorker心跳说明](#)

4、客户端和服务端协议一定要对应才能通讯

这个是开发者非常常见的问题。例如客户端是用websocket协议，服务端必须也是websocket协议(服务端 `new Worker('websocket://0.0.0.0...')`)才能连得上，才能通讯。

不要尝试在浏览器地址栏访问websocket协议端口，不要尝试用webscoket协议访问裸tcp协议端口，协议一定要对应。

这里的原理类似如果你要和英国人交流，那么要使用英语。如果要和日本人交流，那么要使

用日语。这里的语言就类似与通许协议，双方(客户端和服务端)必须使用相同的语言才能交流，否则无法通讯。

5、连接失败可能的原因

刚开始使用workerman时很常见的一个问题是客户端连接服务端失败。原因一般如下：

- 1、服务器防火墙(包括云服务器安全组)阻止了连接（50%几率是这个）
- 2、客户端和服务端使用的协议不一致（30%几率）
- 3、ip或者端口写错了(15%的几率)
- 4、服务端没启动

6、不要使用exit die语句

否则进程会退出，并显示WORKER EXIT UNEXPECTED错误。当然，进程退出了会立刻重启一个新的进程继续服务。如果需要返回，可以调用return。

7、改代码要重启

workerman是常驻内存的框架，改代码要重启workerman才能看到新代码的效果。

8、长连接应用建议用GatewayWorker框架

很多发者使用workerman是要开发长连接应用，例如即时通讯、物联网等，长连接应用建议直接使用GatewayWorker框架，它专门在workerman的基础上再次封装，做起长连接应用后台更简单、更易用。

9、支持更高并发

如果业务并发连接数超过1000同时在线，请务必[优化linux内核](#)，并[安装event扩展或者libevent扩展](#)。

入门指引

特性

简单的开发示例

特性

WorkerMan的特性

1、纯PHP开发

使用WorkerMan开发的应用程序不依赖php-fpm、apache、nginx这些容器就可以独立运行。这使得PHP开发者开发、部署、调试应用程序非常方便。

2、支持PHP多进程

为了充分发挥服务器多CPU的性能，WorkerMan默认支持多进程多任务。WorkerMan开启一个主进程和多个子进程对外提供服务，主进程负责监控子进程，子进程独自监听网络连接并接收发送及处理数据，由于进程模型简单，使得WorkerMan更加稳定，更加高效。

3、支持TCP、UDP

WorkerMan支持TCP和UDP两种传输层协议，只需要更改一个属性便可以更换传输层协议，业务代码无需改动。

4、支持长连接

很多时候需要PHP应用程序要与客户端保持长连接，比如聊天室、游戏等，但是传统的PHP容器（apache、nginx、php-fpm）很难做到这一点。使用WorkerMan，只要服务端业务不主动调用关闭连接接口，便可以使用PHP长连接。WorkerMan单个进程可以支持上万的并发连接，多进程则支持数十万的甚至百万并发连接。

5、支持各种应用层协议

WorkerMan接口上支持各种应用层协议，包括自定义协议。在WorkerMan中更换协议同样非常简单，同样只是配置一个字段，协议自动切换，业务代码零改动，甚至可以开启多个不同协议的端口，满足不同的客户端需求。

6、支持高并发

WorkerMan支持Libevent事件轮询库（需要安装Libevent扩展），使用Libevent在高并发时性能非常卓越，如果没有安装Libevent则使用PHP内置的Select相关系统调用，性能也同样非常强悍。

7、支持服务平滑重启

当需要重启服务时（例如发布版本），我们不希望正在处理用户请求的进程被立刻终止，更不希望重启的那一刻导致客户端通讯失败。WorkerMan提供了平滑重启功能，能够保障服务平滑升级，不影响客户端的使用。

8、支持文件更新检测及自动加载

在开发过程中，我们希望在修改代码后能够立刻生效，以便查看结果。WorkerMan提供了[文件检测及自动加载组件](#)，只要文件有更新，WorkerMan会自动运行reload，以便加载新的文件，使之生效。

9、支持以指定用户运行子进程

因为子进程是实际处理用户请求的进程，为了安全考虑，子进程不能有太高的权限，所以WorkerMan支持设置子运行进程运行的用户，使你的服务器更加安全。

10、支持对象或者资源永久保持

WorkerMan在运行过程中只会载入解析一次PHP文件，然后便常驻内存，这使得类及函数声明、PHP执行环境、符号表等不会重复创建销毁，这与Web容器下运行的PHP机制是完全不同的。在WorkerMan中，一个进程生命周期内静态成员或者全局变量在不主动销毁的情况下是永久保持的，也就是将对象或者连接等资源放到全局变量或者类静态成员中则当前进程的整个生命周期内的所有请求都可以复用。例如只要单个进程内初始化一次数据库连接，则以后这个进程的所有请求都可以复用这个数据库连接，避免了频繁连接数据库过程中TCP三次握手、数据库权限验证、断开连接时TCP四次握手的过程，极大的提高了应用程序效率。

11、高性能

由于php文件从磁盘读取解析一次后会常驻内存，下次使用时直接使用内存中的opcode，极大的减少了磁盘IO及PHP中请求初始化、创建执行环境、词法解析、语法解析、编译opcode、请求关闭等诸多耗时过程，并且不依赖nginx、apache等容器，少了nginx等容器与PHP通信的开销，最主要的是资源可以永久保持，不必每次初始化数据库连接等等，所以使用WorkerMan开发应用程序，性能非常高。

12、支持HHVM

支持在HHVM虚拟机上运行，可成倍提升PHP性能。尤其是在cpu密集运算业务中，性能非常优异。通过实际压力测试对比，在没有负载业务的情况下，WorkerMan在HHVM下运行比在Zend PHP5.6运行网络吞吐量提高了30-80%左右

13、支持分布式部署

14、支持守护进程化

15、支持多端口监听

16、支持标准输入输出重定向

特性

简单的开发示例

简单的开发实例

实例一、使用HTTP协议对外提供Web服务

创建http_test.php文件（位置任意，能引用到Workerman/Autoloader.php即可，下同）

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

// 创建一个Worker监听2345端口，使用http协议通讯
$http_worker = new Worker("http://0.0.0.0:2345");

// 启动4个进程对外提供服务
$http_worker->count = 4;

// 接收到浏览器发送的数据时回复hello world给浏览器
$http_worker->onMessage = function($connection, $data)
{
    // 向浏览器发送hello world
    $connection->send('hello world');
};

// 运行worker
Worker::runAll();
```

命令行运行（windows用户用 [cmd命令行](#)，下同）

```
php http_test.php start
```

测试

假设服务端ip为127.0.0.1

在浏览器中访问url <http://127.0.0.1:2345>

注意：

- 1、如果出现无法访问的情况，请参照[手册常见问题-连接失败](#)一节排查。
- 2、服务端是http协议，只能用http协议通讯，用websocket等其它协议无法直接通讯。

实例二、使用WebSocket协议对外提供服务

本文档使用 [看云](#) 构建

创建ws_test.php文件

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

// 注意：这里与上个例子不通，使用的是websocket协议
$ws_worker = new Worker("websocket://0.0.0.0:2000");

// 启动4个进程对外提供服务
$ws_worker->count = 4;

// 当收到客户端发来的数据后返回hello $data给客户端
$ws_worker->onMessage = function($connection, $data)
{
    // 向客户端发送hello $data
    $connection->send('hello ' . $data);
};

// 运行worker
Worker::runAll();
```

命令行运行

```
php ws_test.php start
```

测试

打开chrome浏览器，按F12打开调试控制台，在Console一栏输入(或者把下面代码放入到html页面用js运行)

```
// 假设服务端ip为127.0.0.1
ws = new WebSocket("ws://127.0.0.1:2000");
ws.onopen = function() {
    alert("连接成功");
    ws.send('tom');
    alert("给服务端发送一个字符串：tom");
};
ws.onmessage = function(e) {
    alert("收到服务端的消息：" + e.data);
};
```

注意：

- 1、如果出现无法访问的情况，请参照[手册常见问题-连接失败](#)一节排查。
- 2、服务端是websocket协议，只能用websocket协议通讯，用http等其它协议无法直接通

讯。

实例三、直接使用TCP传输数据

创建tcp_test.php

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

// 创建一个Worker监听2347端口，不使用任何应用层协议
$tcp_worker = new Worker("tcp://0.0.0.0:2347");

// 启动4个进程对外提供服务
$tcp_worker->count = 4;

// 当客户端发来数据时
$tcp_worker->onMessage = function($connection, $data)
{
    // 向客户端发送hello $data
    $connection->send('hello ' . $data);
};

// 运行worker
Worker::runAll();
```

命令行运行

```
php tcp_test.php start
```

测试：命令行运行

(以下是linux命令行效果，与windows下效果有所不同)

```
telnet 127.0.0.1 2347
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
tom
hello tom
```

注意：

- 1、如果出现无法访问的情况，请参照[手册常见问题-连接失败](#)一节排查。
- 2、服务端是裸tcp协议，用websocket、http等其它协议无法直接通讯。

安装

环境要求

下载安装

启动停止

环境要求

环境要求

Windows用户

Windows用户只能使用windows版本的Workerman(`Workerman-for-win`)。

- 1、需要PHP \geq 5.3.3，并配置好PHP的环境变量。
- 2、Windows版本的Workerman不依赖任何扩展。
- 3、安装使用以及注意事项参见[这里](#)。

`=====`本页面以下只适用于Linux用户，Windows用户请忽略。`=====`

Linux用户(含Mac OS)

Linux用户只能使用Linux版本的Workerman。

- 1、安装PHP \geq 5.3.3，并安装了pcntl、posix扩展
- 2、建议安装event或者libevent扩展，但不是必须的（注意event扩展需要PHP \geq 5.4）

Linux环境检查脚本

Linux用户可以运行以下脚本检查本地环境是否满足WorkerMan要求

```
curl -Ss http://www.workerman.net/check.php | php
```

如果脚本中全部提示ok，则代表满足WorkerMan运行环境

（注意：检测脚本中没有检测event扩展或者libevent扩展，如果并发连接数大于1024建议安装event扩展或者libevent扩展，安装方法参见下一节）

详细说明

关于PHP-CLI

WorkerMan是基于[PHP命令行\(PHP-CLI\)](#)模式运行的。PHP-CLI与PHP-FPM或者Apache的MOD-PHP是独立的可执行程序，它们之间并不冲突也不会有相互依赖，完全独立。

关于WorkerMan依赖的扩展

- 1、[pcntl扩展](#)

pcntl扩展是PHP在Linux环境下进程控制的重要扩展，WorkerMan用到了其[进程创建](#)、[信号控制](#)、[定时器](#)、[进程状态监控](#)等特性。此扩展win平台不支持。

2、[posix扩展](#)

posix扩展使得PHP在Linux环境可以调用系统通过[POSIX标准](#)提供的接口。WorkerMan主要使用了其相关的接口实现了守护进程化、用户组控制等功能。此扩展win平台不支持。

3、[libevent扩展](#) 或者 [Event扩展](#)

libevent扩展(或者event扩展)使得PHP可以使用系统[Epoll](#)、Kqueue等高级事件处理机制，能够显著提高WorkerMan在高并发连接时CPU利用率。在高并发长连接相关应用中非常重要。libevent扩展(或者event扩展)不是必须的，如果没安装，则默认使用PHP原生Select事件处理机制。

如何安装扩展

参见 [附录-安装扩展](#) 章节

下载安装

安装说明

WorkerMan实际上就是一个PHP代码包，如果你的PHP环境已经装好，只需要把WorkerMan源代码或者demo下载下来即可运行。

windows用户（必读）

windows用户需要使用windows版本的workerman，windows版本workerman本身不依赖任何扩展，只需要配置好PHP环境变量即可，windows版本workerman安装及注意事项参见[windows用户必看](#)。

===本页面以下仅适用于Linux版本workerman，windows用户请忽略===

Linux系统环境检测

Linux系统可以使用以下脚本测试本机PHP环境是否满足WorkerMan运行要求。

```
` curl -Ss http://www.workerman.net/check.php | php `
```

上面脚本如果全部显示ok，则代表满足WorkerMan要求，直接到[官网](#)下载例子即可运行。

如果不是全部ok，则参考下面文档安装缺失的扩展即可。

（注意：检测脚本中没有检测event扩展或者libevent扩展，如果业务并发连接数大于1024建议安装event扩展或者libevent扩展，安装方法参照下面说明）

已有PHP环境安装缺失扩展

安装pcntl和posix扩展：

centos系统

如果php是通过yum安装的，则命令行运行 `` yum install php-process `` 即可安装pcntl和posix扩展。

如果安装失败或者php本身不是用yum安装的请参考手册[附录-安装扩展](#)一节中方法三源码编译安装。

debian/ubuntu/mac os系统

参考手册[附录-安装扩展](#)一节中方法三源码编译安装。

安装event或者libevent扩展：

为了能支持更大的并发连接数，建议安装event扩展或者libevent扩展(二者作用相同，二选一即可)。以Event为例，安装方法如下：

centos系统

1、安装event扩展依赖的libevent-devel包，命令行运行

```
yum install libevent-devel -y
```

2、安装event扩展，命令行运行

(event扩展要求PHP>=5.4，PHP5.3用户请安装libevent扩展(libevent扩展同时支持php5.4-5.6)，见本页面底部)

```
pecl install event
```

注意提示：`> Include libevent OpenSSL support [yes] :` 时输入 `> no` 回车，其它直接敲回车就行

如果安装失败请跳过以下步骤，尝试安装libevent扩展，见本页面底部。

3、命令行运行（如果ini文件位置不对，可以通过运行 `> php --ini` 找到实际加载的ini文件路径）

```
echo extension=event.so > /etc/php.d/30-event.ini
```

debian/ubuntu系统安装

1、安装event扩展依赖的libevent-dev包，命令行运行

```
apt-get install libevent-dev -y
```

2、安装event扩展，命令行运行

(注意：event扩展要求PHP>=5.4，PHP5.3用户请安装libevent扩展(libevent扩展同时支持php5.4-5.6)，见本页面底部)

```
pecl install event
```

注意提示：` Include libevent OpenSSL support [yes] : ` 时输入 ` no ` 回车，其它直接敲回车就行

如果安装失败请跳过以下步骤，尝试安装libevent扩展，见本页面底部。

3、命令行运行(需要切换到root用户。如果ini文件位置不对，可以通过运行 ` php --ini ` 找到实际加载的ini文件路径)

```
echo extension=event.so > /etc/php5/cli/conf.d/30-event.ini
```

mac os 系统安装教程

mac 系统一般作为开发机，不必安装event扩展。

全新系统安装（全新安装PHP+扩展） centos系统安装教程

1、命令行运行（此步骤包含了安装php-cli主程序以及pcntl、posix、libevent库及git程序）

```
yum install php-cli php-process git gcc php-devel php-pear libevent-devel -y
```

2、安装event扩展，命令行运行

(注意：event扩展要求PHP>=5.4，PHP5.3用户请安装libevent扩展(libevent扩展也支持php5.4-5.6)，见本页面底部)

```
pecl install event
```

注意提示：` Include libevent OpenSSL support [yes] : ` 时输入 ` no ` 回车，其它直接敲回车就行

如果安装失败请跳过以下步骤3，尝试安装libevent扩展，见本页面底部。

3、命令行运行（此步骤是配置event扩展的ini配置，如果ini文件位置不对，可以通过运行 `php --ini` 找到实际加载的ini文件路径）

```
echo extension=event.so > /etc/php.d/30-event.ini
```

4、命令行运行（此步骤是通过github下载WorkerMan主程序）

```
git clone https://github.com/walkor/Workerman
```

5、参考[入门指引--简单开发实例部分](#)写入口文件运行。

或者从[官网](#)下载打包好的demo运行。

debian/ubuntu系统安装教程

1、命令行运行（此步骤包含了安装php-cli主程序、libevent库及git程序）

```
apt-get install php5-cli git gcc php-pear php5-dev libevent-dev -y
```

2、安装event扩展，命令行运行

(注意：event扩展要求PHP>=5.4，PHP5.3用户请安装libevent扩展(libevent扩展也支持php5.4-5.6)，见本页面底部)

```
pecl install event
```

注意提示：`Include libevent OpenSSL support [yes]:` 时输入 `no` 回车，其它直接敲回车就行

如果安装失败请跳过以下步骤3，尝试安装libevent扩展，见本页面底部。

3、命令行运行（需要切换到root用户。此步骤是配置Event扩展的ini配置，如果ini文件位置不对，可以通过运行 `php --ini` 找到实际加载的ini文件路径）

```
echo extension=event.so > /etc/php5/cli/conf.d/30-event.ini
```

4、命令行运行（此步骤是通过github下载WorkerMan主程序）

```
git clone https://github.com/walkor/Workerman
```

5、参考[入门指引--简单开发实例部分](#)写入口文件运行。

或者从[官网](#)下载打包好的demo运行。

mac os 系统安装教程

方法1：mac系统自带PHP Cli，但是可能缺少 ``pcntl`` 扩展。

- 1、参考手册[附录-安装扩展](#)一节中方法三源码编译安装 ``pcntl`` 扩展。
- 2、参考手册[附录-安装扩展](#)一节中方法四利用phpize安装 ``event`` 扩展（作为开发机此可省略）。
- 3、通过<http://www.workerman.net/download/workermanzip> 下载WorkerMan主程序，或者到[官网](#)下载例子运行。

方法2：通过 ``brew`` 命令安装php及对应扩展

- 1、命令行运行以下命令安装 ``brew`` 工具(如果已经安装过 ``brew`` 可以跳过此步骤)

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

- 2、命令行运行以下命令安装 ``php7``

```
brew install php70
```

- 3、命令行运行以下命令安装 ``event`` 扩展

```
brew install php70-event
```

- 4、到[官网](#)下载例子运行

Event扩展说明

[Event扩展](#)不是必须的，当业务需要支撑大于1000的并发连接时，推荐安装Event，能够支持

巨大的并发连接。如果业务并发连接比较低，例如1000以下并发连接，则可以不用安装。

如果无法安装Event扩展，可以用libevent扩展代替，注意目前libevent扩展不支持php7，php7用户只能使用Event扩展。

安装libevnet扩展方法如下：

注意：

- 1、libevnet扩展也同样依赖libevent库，所以首先需要安装libevent-devel包(并非扩展)。
- 2、libevent扩展支持php5.3-5.6，目前还不支持php7。php7用户请使用event扩展，php7用户请不要装libevent扩展，否则会出现coredump错误。

centos系统

```
yum install libevent-devel
pecl install channel://pecl.php.net/libevent-0.1.0 //提示libevent installation [autodetect]: 时按回车
echo extension=libevent.so > /etc/php.d/libevent.ini
```

如果ini文件位置不对，可以通过运行 `php --ini` 找到实际加载的ini文件路径

debian/ubuntu系统

```
apt-get install libevent-dev
pecl install channel://pecl.php.net/libevent-0.1.0 //提示libevent installation [autodetect]: 时按回车
echo extension=libevent.so > /etc/php5/cli/conf.d/libevent.ini
```

如果ini文件位置不对，可以通过运行 `php --ini` 找到实际加载的ini文件路径

启动停止

启动与停止

注意Workerman启动停止等命令都是在命令行中完成的。

要启动Workerman，首先需要有一个启动入口文件，里面定义了服务监听的端口及协议。可以参考[入门指引--简单开发实例部分](#)

这里以[workerman-chat](#)为例，它的启动入口为start.php。

启动

以debug（调试）方式启动

```
` php start.php start `
```

以daemon（守护进程）方式启动

```
` php start.php start -d `
```

停止

```
` php start.php stop `
```

重启

```
` php start.php restart `
```

平滑重启

```
` php start.php reload `
```

查看状态

```
` php start.php status `
```

查看连接状态（需要Workerman版本>=3.5.0）

debug和daemon方式区别

- 1、以debug方式启动，代码中echo、var_dump、print等打印函数会直接输出在终端。
- 2、以daemon方式启动，代码中echo、var_dump、print等打印会默认重定向到/dev/null文件，可以通过设置``Worker::\$stdoutFile = '/your/path/file';``来设置这个文件路径

。

3、以debug方式启动，终端关闭后workerman会随之关闭并退出。

4、以daemon方式启动，终端关闭后workerman继续后台正常运行。

什么是平滑重启？

平滑重启不同于普通的重启，平滑重启可以做到在不影响用户的情况下重启服务，以便重新载入PHP程序，完成业务代码更新。

平滑重启一般应用于业务更新或者版本发布过程中，能够避免因为代码发布重启服务导致的暂时性服务不可用的影响。

****注意：只有在on{...}回调中载入的文件平滑重启后才会自动更新，启动脚本中直接载入的文件或者写死的代码运行reload不会自动更新。****

平滑重启原理

WorkerMan分为主进程和子进程，主进程负责监控子进程，子进程负责接收客户端的连接和连接上发来的请求数据，做相应的处理并返回数据给客户端。当业务代码更新时，其实我们只要更新子进程，便可以达到更新代码的目的。

当WorkerMan主进程收到平滑重启信号时，主进程会向其中一个子进程发送安全退出(让对应进程处理完毕当前请求后才退出)信号，当这个进程退出后，主进程会重新创建一个新的子进程（这个子进程载入了新的PHP代码），然后主进程再次向另外一个旧的进程发送停止命令，这样一个进程一个进程的重启，直到所有旧的进程全部被替换为止。

我们看到平滑重启实际上是让旧的业务进程逐个退出然后并逐个创建新的进程做到的。为了在平滑重启时不影响客用户，这就要求进程中不要保存用户相关的状态信息，即业务进程最好是无状态的，避免因进程退出导致信息丢失。

开发流程

开发前必读

目录结构

开发规范

基本流程

开发前必读

开发前必读

使用WorkerMan开发应用，你需要了解以下内容：

一、WorkerMan开发与普通PHP开发的不同之处

除了与HTTP协议相关的变量函数无法直接使用外，WorkerMan开发与普通PHP开发并没有很大不同。

1、应用层协议不同

- 普通PHP开发一般是基于HTTP应用层协议，WebServer已经帮开发者完成了协议的解析
- WorkerMan支持各种协议，目前内置了HTTP、WebSocket等协议。WorkerMan推荐开发者使用更简单的自定义协议通讯

由于非HTTP协议的应用，所以 `header()` `setcookie()` `session_start` 等函数无法直接使用，需要使用WorkerMan提供的方法，具体参考高级应用-WebServer部分

2、请求周期差异

- PHP在Web应用中一次请求后会释放所有的变量与资源
- WorkerMan开发的应用程序在第一次载入解析后便常驻内存，使得类的定义、全局对象、类的静态成员不会释放，便于后续重复利用

3、注意避免类和常量的重复定义

- 由于WorkerMan会缓存编译后的PHP文件，所以要避免多次require/include相同的类或者常量的定义文件。建议使用require_once/include_once加载文件。

4、注意单例模式的连接资源的释放

- 由于WorkerMan不会在每次请求后释放全局对象及类的静态成员，在数据库等单例模式中，往往会将数据库实例（内部包含了一个数据库socket连接）保存在数据库静态成员中，使得WorkerMan在进程生命周期内都复用这个数据库socket连接。需要注意的是当数据库服务器发现某个连接在一定时间内没有活动后可能会主动关闭socket连接，这时再次使用这个数据库实例时会报错，（错误信息类似mysql gone away）。WorkerMan提供了[数据库类](#)，有断开重连的功能，开发者可以直接使用。

5、注意不要使用exit、die出语句

- WorkerMan运行在PHP命令行模式下，当调用exit、die退出语句时，会导致当前进程退

出。虽然子进程退出后会立刻重新创建一个相同的子进程继续服务，但是还是可能对业务产生影响。

二、需要了解的基本概念

1、TCP传输层协议

TCP是一种面向连接的、可靠的、基于IP的传输层协议。TCP传输层协议一个重要特点是TCP是基于数据流的，客户端的请求会源源不断的发送给服务端，服务端收到的数据可能不是一个完整的请求，也有可能是多个请求连在一起。这就需要我们在这源源不断的数据流中区分每个请求的边界。而应用层协议主要是为请求边界定义一套规则，避免请求数据混乱。

2、应用层协议

应用层协议(application layer protocol)定义了运行在不同端系统上（客户端、服务端）的应用程序进程如何相互传递报文，例如HTTP、WebSocket都属于应用层协议。例如一个简单的应用层协议可以如下 `{"module":"user","action":"getInfo","uid":456}\n`。此协议是以 `"\n"`（注意这里 `"\n"` 代表的是回车）标记请求结束，消息体是字符串。

3、短连接

短连接是指通讯双方有数据交互时，就建立一个连接，数据发送完成后，则断开此连接，即每次连接只完成一项业务的发送。像WEB网站的HTTP服务一般都用短连接。

短连接应用程序开发可以参考基本开发流程一章

4、长连接

长连接，指在一个连接上可以连续发送多个数据包。

注意：长连接应用必须加心跳，否则连接可能由于长时间不活跃而被路由节点防火墙断开。

长连接多用于操作频繁，点对点的通讯的情况。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多。所以长连接在每个操作完后都不断开，下次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

当需要主动向客户端推送数据时，例如聊天类、即时游戏类、手机推送等应用需要长连接。

长连接应用程序开发可以参考Gateway/Worker开发流程

5、平滑重启

一般的重启的过程是把所有进程全部停止后，再开始创建全新的服务进程。在这个过程中会有一个短暂的时间内是没有进程对外提供服务的，这就会导致服务暂时不可用，这在高并发

时势必会导致请求失败。

而平滑重启则不是一次性的停止所有进程，而是一个进程一个进程的停止，每停止一个进程后马上重新创建一个新的进程顶替，直到所有旧的进程都被替换为止。

平滑重启WorkerMan可以使用 `` php your_file.php reload `` 命令，能够做到在不影响服务质量的情况下更新应用程序。

注意：只有在on{...}回调中载入的文件平滑重启后才会自动更新，启动脚本中直接载入的文件或者写死的代码运行reload不会自动更新。

三、区分主进程和子进程

有必要注意下代码是运行在主进程还是子进程，一般来说在 `` Worker::runAll(); `` 调用前运行的代码都是在主进程运行的，onXXX回调运行的代码都属于子进程。注意写在 `` Worker::runAll(); `` 后面的代码永远不会被执行。

例如下面的代码

```
require_once __DIR__ . '/Workerman/Autoloader.php';
use Workerman\Worker;

// 运行在主进程
$tcp_worker = new Worker("tcp://0.0.0.0:2347");
// 赋值过程运行在主进程
$tcp_worker->onMessage = function($connection, $data)
{
    // 这部分运行在子进程
    $connection->send('hello ' . $data);
};

Worker::runAll();
```

注意：不要在主进程中初始化数据库、memcache、redis等连接资源，因为主进程初始化的连接可能会被子进程自动继承（尤其是使用单例的时候），所有进程都持有同一个连接，服务端通过这个连接返回的数据在多个进程上都可读，会导致数据错乱。同样的，如果任何一个进程关闭连接(例如daemon模式运行时主进程会退出导致连接关闭)，都导致所有子进程的连接都被一起关闭，并发生不可预知的错误，例如mysql gone away 错误。

目录结构

目录结构

```
Workerman                                     // workerman内核代码
├── Connection                               // socket连接相关
│   ├── ConnectionInterface.php             // socket连接接口
│   ├── TcpConnection.php                  // Tcp连接类
│   ├── AsyncTcpConnection.php             // 异步Tcp连接类
│   └── UdpConnection.php                  // Udp连接类
├── 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                             // Libev网络事件库
│   └── Select.php                         // Select网络事件库
├── Lib                                      // 常用的类库
│   ├── Constants.php                     // 常量定义
│   └── Timer.php                          // 定时器
├── Protocols                               // 协议相关
│   ├── ProtocolInterface.php              // 协议接口类
│   ├── Http                              // http协议相关
│   │   └── mime.types                     // mime类型
│   ├── Http.php                          // http协议实现
│   ├── Text.php                          // Text协议实现
│   ├── Frame.php                         // Frame协议实现
│   └── WebSocket.php                      // websocket协议的实现
├── Worker.php                             // Worker
├── WebServer.php                           // WebServer
└── Autoloader.php                         // 自动加载类
```

开发规范

开发规范 应用程序目录

应用程序目录可以放到任意位置

入口文件

和nginx+PHP-FPM下的PHP应用程序一样，WorkerMan中的应用程序也需要一个入口文件，入口文件名没有要求，并且这个入口文件是以PHP Cli方式运行的。

入口文件中是创建监听进程相关的代码，例如下面的基于Worker开发的代码片段

test.php

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

// 创建一个Worker监听2345端口，使用http协议通讯
$http_worker = new Worker("http://0.0.0.0:2345");

// 启动4个进程对外提供服务
$http_worker->count = 4;

// 接收到浏览器发送的数据时回复hello world给浏览器
$http_worker->onMessage = function($connection, $data)
{
    // 向浏览器发送hello world
    $connection->send('hello world');
};

Worker::runAll();
```

WorkerMan中的代码规范

1、类采用首字母大写的驼峰式命名，类文件名称必须与文件内部类名相同，以便自动加载。

例如：

```
class UserInfo
{
    ...
}
```

2、使用命名空间，命名空间名字与目录路径对应，并以开发者的项目根目录为基准。

例如项目MyApp/，类文件MyApp/MyClass.php因为在项目根目录，所以命名空间省略。
类文件MyApp/Protocols/MyProtocol.php因为MyProtocol.php在MyApp项目的Protocols目录下，所以要加上命名空间 `namespace Protocols;`，如下：

```
namespace Protocols;  
class MyProtocol  
{  
....
```

3、普通函数及变量名采用小写加下划线方式 例如

```
$connection_list = array();  
function get_connection_list()  
{  
....
```

4、类成员及类的方法采用首字母小写的驼峰形式 例如：

```
public $connectionList;  
public function getConnectionList();
```

5、函数及类的参数采用小写加下划线方式

```
function get_connection_list($one_param, $tow_param)  
{  
....
```


基本流程

基本流程

(以一个简单的Websocket聊天室服务端为例)

1、任意位置建立项目目录

如 SimpleChat/

2、引入Workerman/Autoloader.php

如

```
require_once '/your/path/Workerman/Autoloader.php';
```

3、选定协议

这里我们选定Text文本协议(WorkerMan中自定义的一个协议，格式为文本+换行)

(目前WorkerMan支持HTTP、Websocket、Text文本协议，如果需要使用其它协议，请参照协议一章开发自己的协议)

4、根据需要写入口启动脚本

例如下面这个是一个简单的聊天室的入口文件。

SimpleChat/start.php

```
<?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;
    // 为这个连接分配一个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();
```

5、测试

Text协议可以用telnet命令测试

```
telnet 127.0.0.1 2347
```

通讯协议

[通讯协议作用](#)

[定制通讯协议](#)

[一些例子](#)

通讯协议作用

通讯协议的作用

由于TCP是基于流的，客户端发送的请求数据是像水流一样流入到服务端，服务端探测到有数据到来后应该检查数据是否是完整的，因为可能只是一个请求的部分数据到达服务端，甚至可能是多个请求连在一起到达服务端。如何判断请求是否全部到达或者从多个连在一起的请求中分离请求，就需要规定一套通讯协议。

在WorkerMan中为什么要制定协议？

传统PHP开发都是基于Web的，基本上都是HTTP协议，HTTP协议的解析处理都由WebServer独自承担了，所以开发者不会关心协议方面的事情。然而当我们需要基于非HTTP协议开发时，开发者就需要考虑协议的事情了。

WorkerMan已经支持的协议

WorkerMan目前已经支持HTTP、websocket、text协议(见附录说明)、frame协议(见附录说明)，ws协议(见附录说明)，需要基于这些协议通讯时可以直接使用，使用方法及时在初始化Worker时指定协议，例如

```
use Workerman\Worker;
// websocket://0.0.0.0:2345 表明用websocket协议监听2345端口
$websocket_worker = new Worker('websocket://0.0.0.0:2345');

// text协议
$text_worker = new Worker('text://0.0.0.0:2346');

// frame协议
$frame_worker = new Worker('frame://0.0.0.0:2347');

// tcp Worker, 直接基于socket传输, 不使用任何应用层协议
$tcp_worker = new Worker('tcp://0.0.0.0:2348');

// udp Worker, 不使用任何应用层协议
$udp_worker = new Worker('udp://0.0.0.0:2349');

// unix domain Worker, 不使用任何应用层协议
$unix_worker = new Worker('unix:///tmp/wm.sock');
```

使用自定义的通讯协议

当WorkerMan自带的通讯协议满足不了开发需求时，开发者可以定制自己的通讯协议，定

制方法见下一节内容。

提示：

Workerman内置了一个text协议，协议格式为文本+换行符。text协议开发调试都非常简单，可用于绝大多数自定义协议的场景，并且支持telnet调试。如果开发者要开发自己的应用协议，可以直接使用text协议，不用再单独开发。

text协议说明参考《附录 Text协议部分》

定制通讯协议

如何定制协议

实际上制定自己的协议是比较简单的事情。简单的协议一般包含两部分：

- 区分数据边界的标识
- 数据格式定义

一个例子

协议定义

这里假设区分数据边界的标识为换行符"`\n`"（注意请求数据本身内部不能包含换行符），数据格式为Json，例如下面是一个符合这个规则的请求包。

```
{"type":"message","content":"hello"}
```

注意上面的请求数据末尾有一个换行字符(在PHP中用双引号字符串"`\n`"表示)，代表一个请求的结束。

实现步骤

在WorkerMan中如果要实现上面的协议，假设协议的名字叫JsonNL，所在项目为MyApp，则需要以下步骤

- 1、协议文件放到项目的Protocols文件夹，例如文件MyApp/Protocols/JsonNL.php
- 2、实现JsonNL类，以 `\ namespace Protocols; \` 为命名空间，必须实现三个静态方法分别为 `input`、`encode`、`decode`

注意：workerman会自动调用这三个静态方法，用来实现分包、解包、打包。具体流程参考下面执行流程说明。

workerman与协议类交互流程

- 1、假设客户端发送一个数据包给服务端，服务端收到数据(可能是部分数据)后会立刻调用协议的 `\ input \` 方法，用来检测这包的长度，`\ input \` 方法返回长度值 `\ $length \` 给workerman框架。
- 2、workerman框架得到这个 `\ $length \` 值后判断当前数据缓冲区中是否已经接收到 `\ $length \` 长度的数据，如果没有就会继续等待数据，直到缓冲区中的数据长度不小于 `\`

\$length`。

4、缓冲区的数据长度足够后，workerman就会从缓冲区截取出`\$length`长度的数据(即分包)，并调用协议的`decode`方法解包，解包后的数据为`\$data`。

3、解包后workerman将数据`\$data`以回调`onMessage(\$connection, \$data)`的形式传递给业务，业务在onMessage里就可以使用`\$data`变量得到客户端发来的完整并且已经解包的数据了。

4、当`onMessage`里业务需要通过调用`\$connection->send(\$buffer)`方法给客户端发送数据时，workerman会自动利用协议的`encode`方法将`\$buffer`打包后再发给客户端。

具体实现

MyApp/Protocols/JsonNL.php的实现

```
namespace Protocols;
class JsonNL
{
    /**
     * 检查包的完整性
     * 如果能够得到包长，则返回包的在buffer中的长度，否则返回0继续等待数据
     * 如果协议有问题，则可以返回false，当前客户端连接会因此断开
     * @param string $buffer
     * @return int
     */
    public static function input($buffer)
    {
        // 获得换行字符"\n"位置
        $pos = strpos($buffer, "\n");
        // 没有换行符，无法得知包长，返回0继续等待数据
        if($pos === false)
        {
            return 0;
        }
        // 有换行符，返回当前包长（包含换行符）
        return $pos+1;
    }

    /**
     * 打包，当向客户端发送数据的时候会自动调用
     * @param string $buffer
     * @return string
     */
    public static function encode($buffer)
    {
        // json序列化，并加上换行符作为请求结束的标记
        return json_encode($buffer)."\n";
    }
}
```

```

    * 解包，当接收到的数据字节数等于input返回的值（大于0的值）自动调用
    * 并传递给onMessage回调函数的$data参数
    * @param string $buffer
    * @return string
    */
    public static function decode($buffer)
    {
        // 去掉换行，还原成数组
        return json_decode(trim($buffer), true);
    }
}

```

至此，JsonNL协议实现完毕，可以在MyApp项目中使用，使用方法例如下面

文件：MyApp\start.php

```

use Workerman\Worker;
require_once '/your/path/Workerman/Autoloader.php';
$json_worker = new Worker('JsonNL://0.0.0.0:1234');
$json_worker->onMessage = function($connection, $data) {

    // $data就是客户端传来的数据，数据已经经过JsonNL::decode处理过
    echo $data;

    // $connection->send的数据会自动调用JsonNL::encode方法打包，然后发往客户端
    $connection->send(array('code'=>0, 'msg'=>'ok'));

};
Worker::runAll();
...

```

协议接口说明

在WorkerMan中开发的协议类必须实现三个静态方法，input、encode、decode，协议接口说明见Workerman/Protocols/ProtocolInterface.php，定义如下：

```

namespace Workerman\Protocols;

use \Workerman\Connection\ConnectionInterface;

/**
 * Protocol interface
 * @author walkor <walkor@workerman.net>
 */
interface ProtocolInterface
{
    /**
     * 用于在接收到的recv_buffer中分包
     *
     * 如果可以在$recv_buffer中得到请求包的长度则返回整个包的长度
     */
}

```



```

    * 否则返回0，表示需要更多的数据才能得到当前请求包的长度
    * 如果返回false或者负数，则代表错误的请求，则连接会断开
    *
    * @param ConnectionInterface $connection
    * @param string $recv_buffer
    * @return int|false
    */
    public static function input($recv_buffer, ConnectionInterface $connection);

    /**
     * 用于请求解包
     *
     * input返回值大于0，并且WorkerMan收到了足够的数据，则自动调用decode
     * 然后触发onMessage回调，并将decode解码后的数据传递给onMessage回调的第二个参
数
     * 也就是说当收到完整的客户端请求时，会自动调用decode解码，无需业务代码中手动调用
     * @param ConnectionInterface $connection
     * @param string $recv_buffer
     */
    public static function decode($recv_buffer, ConnectionInterface $connection);

    /**
     * 用于请求打包
     *
     * 当需要向客户端发送数据即调用$connection->send($data);时
     * 会自动把$data用encode打包一次，变成符合协议的数据格式，然后再发送给客户端
     * 也就是说发送给客户端的数据会自动encode打包，无需业务代码中手动调用
     * @param ConnectionInterface $connection
     * @param mixed $data
     */
    public static function encode($data, ConnectionInterface $connection)
    ;
}

```

注意：

Workerman中没有严格要求协议类必须基于ProtocolInterface实现，实际上协议类只要类包含了input、encode、decode三个静态方法即可。

一些例子

一些例子

例子一

协议定义

- 首部固定10个字节长度用来保存整个数据包长度，位数不够补0
- 数据格式为xml

数据包样本

```
00000000121<?xml version="1.0" encoding="ISO-8859-1"?>
<request>
  <module>user</module>
  <action>getInfo</action>
</request>
```

其中00000000121代表整个数据包长度，后面紧跟xml数据格式的包体内容

协议实现

```
namespace Protocols;
class XmlProtocol
{
    public static function input($recv_buffer)
    {
        if(strlen($recv_buffer) < 10)
        {
            // 不够10字节，返回0继续等待数据
            return 0;
        }
        // 返回包长，包长包含 头部数据长度+包体长度
        $total_len = base_convert(substr($recv_buffer, 0, 10), 10, 10);
        return $total_len;
    }

    public static function decode($recv_buffer)
    {
        // 请求包体
        $body = substr($recv_buffer, 10);
        return simplexml_load_string($body);
    }

    public static function encode($xml_string)
    {
        // 包体+包头的长度
        $total_length = strlen($xml_string)+10;
        // 长度部分凑足10字节，位数不够补0
    }
}
```

```
        $total_length_str = str_pad($total_length, 10, '0', STR_PAD_LEFT)
    ;
    // 返回数据
    return $total_length_str . $xml_string;
}
}
```

例子二

协议定义

- 首部4字节网络字节序unsigned int，标记整个包的长度
- 数据部分为Json字符串

数据包样本

```
****{"type":"message","content":"hello all"}
```

其中首部四字节*号代表一个网络字节序的unsigned int数据，为不可见字符，紧接着是Json的数据格式的包体数据

协议实现

```
namespace Protocols;
class JsonInt
{
    public static function input($recv_buffer)
    {
        // 接收到的数据还不够4字节，无法得知包的长度，返回0继续等待数据
        if(strlen($recv_buffer)<4)
        {
            return 0;
        }
        // 利用unpack函数将首部4字节转换成数字，首部4字节即为整个数据包长度
        $unpack_data = unpack('Ntotal_length', $recv_buffer);
        return $unpack_data['total_length'];
    }

    public static function decode($recv_buffer)
    {
        // 去掉首部4字节，得到包体Json数据
        $body_json_str = substr($recv_buffer, 4);
        // json解码
        return json_decode($body_json_str, true);
    }

    public static function encode($data)
    {
        // Json编码得到包体
        $body_json_str = json_encode($data);
        // 计算整个包的长度，首部4字节+包体字节数
        $total_length = 4 + strlen($body_json_str);
    }
}
```

```

        // 返回打包的数据
        return pack('N',$total_length) . $body_json_str;
    }
}

```

例子三（使用二进制协议上传文件）

协议定义

```

struct
{
    unsigned int total_len; // 整个包的长度, 大端网络字节序
    char        name_len;  // 文件名的长度
    char        name[name_len]; // 文件名
    char        file[total_len - BinaryTransfer::PACKAGE_HEAD_LEN - name_len]; // 文件数据
}

```

协议样本

```
*****logo.png*****
```

其中首部四字节*号代表一个网络字节序的unsigned int数据，为不可见字符，第5个*是用一个字节存储文件名长度，紧接着是文件名，接着是原始的二进制文件数据

协议实现

```

namespace Protocols;
class BinaryTransfer
{
    // 协议头长度
    const PACKAGE_HEAD_LEN = 5;

    public static function input($recv_buffer)
    {
        // 如果不够一个协议头的长度, 则继续等待
        if(strlen($recv_buffer) < self::PACKAGE_HEAD_LEN)
        {
            return 0;
        }
        // 解包
        $package_data = unpack('Ntotal_len/Cname_len', $recv_buffer);
        // 返回包长
        return $package_data['total_len'];
    }

    public static function decode($recv_buffer)
    {
        // 解包
        $package_data = unpack('Ntotal_len/Cname_len', $recv_buffer);
    }
}

```

```

        // 文件名长度
        $name_len = $package_data['name_len'];
        // 从数据流中截取文件名
        $file_name = substr($recv_buffer, self::PACKAGE_HEAD_LEN, $name_l
en);
        // 从数据流中截取文件二进制数据
        $file_data = substr($recv_buffer, self::PACKAGE_HEAD_LEN + $name_
len);
        return array(
            'file_name' => $file_name,
            'file_data' => $file_data,
        );
    }

    public static function encode($data)
    {
        // 可以根据自己的需要编码发送给客户端的数据，这里只是当做文本原样返回
        return $data;
    }
}

```

服务端协议使用示例

```

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

$worker = new Worker('BinaryTransfer://0.0.0.0:8333');
// 保存文件到tmp下
$worker->onMessage = function($connection, $data)
{
    $save_path = '/tmp/' . $data['file_name'];
    file_put_contents($save_path, $data['file_data']);
    $connection->send("upload success. save path $save_path");
};

Worker::runAll();

```

客户端文件 client.php（这里用php模拟客户端上传）

```

<?php
/** 上传文件客户端 */
// 上传地址
$address = "127.0.0.1:8333";
// 检查上传文件路径参数
if(!isset($argv[1]))
{
    exit("use php client.php \$file_path\n");
}
// 上传文件路径
$file_to_transfer = trim($argv[1]);
// 上传的文件本地不存在
if(!is_file($file_to_transfer))
{

```

```
        exit("$file_to_transfer not exist\n");
    }
    // 建立socket连接
    $client = stream_socket_client($address, $errno, $errmsg);
    if(!$client)
    {
        exit("$errmsg\n");
    }
    // 设置成阻塞
    stream_set_blocking($client, 1);
    // 文件名
    $file_name = basename($file_to_transfer);
    // 文件名长度
    $name_len = strlen($file_name);
    // 文件二进制数据
    $file_data = file_get_contents($file_to_transfer);
    // 协议头长度 4字节包长+1字节文件名长度
    $PACKAGE_HEAD_LEN = 5;
    // 协议包
    $package = pack('NC', $PACKAGE_HEAD_LEN + strlen($file_name) + strlen($file_data), $name_len) . $file_name . $file_data;
    // 执行上传
    fwrite($client, $package);
    // 打印结果
    echo fread($client, 8192), "\n";
```

客户端使用示例

命令行中运行 ``php client.php <文件路径> ``

例如 ``php client.php abc.jpg ``

例子四（使用文本协议上传文件）

协议定义

json+换行，json中包含了文件名以及base64_encode编码（会增大1/3的体积）的文件数据

协议样本

```
{"file_name":"logo.png","file_data":"PD9waHAKLyo....."}\n
```

注意末尾为一个换行符，在PHP中用双引号字符 ``"\n" `` 标识

协议实现

```
namespace Protocols;
class TextTransfer
{
    public static function input($recv_buffer)
    {
        $recv_len = strlen($recv_buffer);
```

```

        if($recv_buffer[$recv_len-1] !== "\n")
        {
            return 0;
        }
        return strlen($recv_buffer);
    }

    public static function decode($recv_buffer)
    {
        // 解包
        $package_data = json_decode(trim($recv_buffer), true);
        // 取出文件名
        $file_name = $package_data['file_name'];
        // 取出base64_encode后的文件数据
        $file_data = $package_data['file_data'];
        // base64_decode还原回原来的二进制文件数据
        $file_data = base64_decode($file_data);
        // 返回数据
        return array(
            'file_name' => $file_name,
            'file_data' => $file_data,
        );
    }

    public static function encode($data)
    {
        // 可以根据自己的需要编码发送给客户端的数据，这里只是当做文本原样返回
        return $data;
    }
}

```

服务端协议使用示例

说明：写法与二进制上传写法一样，即能做到几乎不用改动任何业务代码便可以切换协议

```

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

$worker = new Worker('TextTransfer://0.0.0.0:8333');
// 保存文件到tmp下
$worker->onMessage = function($connection, $data)
{
    $save_path = './tmp/'.$data['file_name'];
    file_put_contents($save_path, $data['file_data']);
    $connection->send("upload success. save path $save_path");
};

Worker::runAll();

```

客户端文件 textclient.php（这里用php模拟客户端上传）

```
<?php
```

本文档使用 [看云](#) 构建

```
/** 上传文件客户端 */
// 上传地址
$address = "127.0.0.1:8333";
// 检查上传文件路径参数
if(!isset($argv[1]))
{
    exit("use php client.php \"$file_path\"");
}
// 上传文件路径
$file_to_transfer = trim($argv[1]);
// 上传的文件本地不存在
if(!is_file($file_to_transfer))
{
    exit("$file_to_transfer not exist");
}
// 建立socket连接
$client = stream_socket_client($address, $errno, $errmsg);
if(!$client)
{
    exit("$errmsg");
}
stream_set_blocking($client, 1);
// 文件名
$file_name = basename($file_to_transfer);
// 文件二进制数据
$file_data = file_get_contents($file_to_transfer);
// base64编码
$file_data = base64_encode($file_data);
// 数据包
$package_data = array(
    'file_name' => $file_name,
    'file_data' => $file_data,
);
// 协议包 json+回车
$package = json_encode($package_data)."\n";
// 执行上传
fwrite($client, $package);
// 打印结果
echo fread($client, 8192), "\n";
```

客户端使用示例

命令行中运行 `php textclient.php <文件路径>`

例如 `php textclient.php abc.jpg`

Worker类

Worker类

WorkerMan中有两个重要的类Worker与Connection。

Worker类用于实现端口的监听，并可以设置客户端连接事件、连接上消息事件、连接断开事件的回调函数，从而实现业务处理。

可以设置Worker实例的进程数（count属性），Worker主进程会fork出count个子进程同时监听相同的端口，并行的接收客户端连接，处理连接上的事件。

构造函数

构造函数 `__construct`

说明:

```
Worker::__construct([string $listen , array $context])
```

初始化一个Worker容器实例，可以设置容器的一些属性和回调接口，完成特定功能。

参数

\$listen （可选参数，不填写表示不监听任何端口）

如果有设置监听，`$listen` 参数，则会执行socket监听。

`$listen` 的格式为 `<协议>://<监听地址>`

`<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` （text是Workerman内置的文本协议，兼容telnet，详情参见附录Text协议部分）

以及其他自定义协议，参见本手册定制通讯协议部分

`<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

一个数组。用于传递socket的上下文选项，参见[套接字上下文选项](#)

范例

Worker作为http容器监听处理http请求

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('http://0.0.0.0:8686');

$worker->onMessage = function($connection, $data)
{
    var_dump($_GET, $_POST);
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

Worker作为websocket容器监听处理websocket请求

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8686');

$worker->onMessage = function($connection, $data)
{
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

Worker作为tcp容器监听处理tcp请求

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8686');

$worker->onMessage = function($connection, $data)
{
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

Worker作为udp容器监听处理udp请求

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('udp://0.0.0.0:8686');

$worker->onMessage = function($connection, $data)
{
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

Worker监听unix domain套接字（要求Workerman版本>=3.2.7）

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('unix:///tmp/my.sock');

$worker->onMessage = function($connection, $data)
{
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

不执行任何监听的Worker容器，用来处理一些定时任务

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once './Workerman/Autoloader.php';

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 每2.5秒执行一次
    $time_interval = 2.5;
    Timer::add($time_interval, function()
    {
        echo "task run\n";
    });
};

// 运行worker
Worker::runAll();

```

Worker监听自定义协议的端口

最终的目录结构

```

├── Protocols          // 这是要创建的Protocols目录
│   └── MyTextProtocol.php // 这是要创建的自定义协议文件
├── test.php          // 这是要创建的test脚本
└── Workerman         // Workerman源码目录，里面代码不要动

```

1、创建Protocols目录，并创建一个协议文件

Protocols/MyTextProtocol.php（参照上面目录结构）

```

// 用户自定义协议命名空间统一为Protocols
namespace Protocols;
//简单文本协议，协议格式为 文本+换行
class MyTextProtocol
{
    // 分包功能，返回当前包的长度
    public static function input($recv_buffer)
    {
        // 查找换行符
        $pos = strpos($recv_buffer, "\n");
        // 没找到换行符，表示不是一个完整的包，返回0继续等待数据
        if($pos === false)
        {
            return 0;
        }
        // 查找到换行符，返回当前包的长度，包括换行符
        return $pos+1;
    }
}

```

```

    // 收到一个完整的包后通过decode自动解码，这里只是把换行符trim掉
    public static function decode($recv_buffer)
    {
        return trim($recv_buffer);
    }

    // 给客户端send数据前会自动通过encode编码，然后再发送给客户端，这里加了换行
    public static function encode($data)
    {
        return $data."\n";
    }
}

```

2、使用MyTextProtocol协议监听处理请求

参照上面最终目录结构创建test.php文件

```

require_once './Workerman/Autoloader.php';
use Workerman\Worker;

// ##### MyTextProtocol worker #####
$text_worker = new Worker("MyTextProtocol://0.0.0.0:5678");

/*
 * 收到一个完整的数据（结尾是换行）后，自动执行MyTextProtocol::decode('收到的数据')
 * 结果通过$data传递给onMessage回调
 */
$text_worker->onMessage = function($connection, $data)
{
    var_dump($data);
    /*
     * 给客户端发送数据，会自动调用MyTextProtocol::encode('hello world')进行协议编码，
     * 然后再发送到客户端
     */
    $connection->send("hello world");
};

// run all workers
Worker::runAll();

```

3、测试

打开终端，进入到test.php所在目录，执行 `php test.php start`

```

php test.php start
Workerman[test.php] start in DEBUG mode
----- WORKERMAN -----
Workerman version:3.2.7      PHP version:5.4.37

```

```
----- WORKERS -----
user          worker      listen                                processes stat
us
root          none        myTextProtocol://0.0.0.0:5678    1          [OK
]
-----
Press Ctrl-C to quit. Start success.
```

打开终端，利用telnet测试（建议用linux系统的telnet）

假设是本地测试，

终端执行 telnet 127.0.0.1 5678

然后输入 hi回车

会接收到数据hello world\n

```
telnet 127.0.0.1 5678
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hi
hello world
```

属性

属性

id

id

要求 ` (workerman >= 3.2.1) `

说明:

```
int Worker::$id
```

当前worker进程的id编号，范围为 ` 0 ` 到 ` \$worker->count-1 `。

这个属性对于区分worker进程非常有用，例如1个worker实例有多个进程，开发者只想在其中一个进程中设置定时器，则可以通过识别进程编号id来做到这一点，比如只在该worker实例id编号为0的进程设置定时器（见范例）。

注意：

进程重启后id编号值是不变的。

进程编号id的分配是基于每个worker实例的。每个worker实例都从0开始给自己的进程编号，所以worker实例间进程编号会有重复，但是一个worker实例中的进程编号不会重复。例如下面的例子：

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

// worker实例1有4个进程，进程id编号将分别为0、1、2、3
$worker1 = new Worker('tcp://0.0.0.0:8585');
// 设置启动4个进程
$worker1->count = 4;
// 每个进程启动后打印当前进程id编号即 $worker1->id
$worker1->onWorkerStart = function($worker1)
{
    echo "worker1->id={$worker1->id}\n";
};

// worker实例2有两个进程，进程id编号将分别为0、1
$worker2 = new Worker('tcp://0.0.0.0:8686');
// 设置启动2个进程
$worker2->count = 2;
// 每个进程启动后打印当前进程id编号即 $worker2->id
$worker2->onWorkerStart = function($worker2)
{
```

```

        echo "worker2->id={$worker2->id}\n";
    };

    // 运行worker
    Worker::runAll();

```

输出类似

```

worker1->id=0
worker1->id=1
worker1->id=2
worker1->id=3
worker2->id=0
worker2->id=1

```

范例

一个worker实例有4个进程，只在id编号为0的进程上设置定时器。

```

use Workerman\Worker;
use Workerman\Lib\Timer;
require_once './Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8585');
$worker->count = 4;
$worker->onWorkerStart = function($worker)
{
    // 只在id编号为0的进程上设置定时器，其它1、2、3号进程不设置定时器
    if($worker->id === 0)
    {
        Timer::add(1, function(){
            echo "4个worker进程，只在0号进程设置定时器\n";
        });
    }
};
// 运行worker
Worker::runAll();

```

count

count

说明:

```
int Worker::$count
```

设置当前Worker实例启动多少个进程，不设置时默认为1。

如何设置进程数，请参考[这里](#)。

注意：此属性必须在 `Worker::runAll();` 运行前设置才有效。windows系统不支持此特性。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 启动8个进程，同时监听8484端口，以websocket协议提供服务
$worker->count = 8;
$worker->onWorkerStart = function($worker)
{
    echo "Worker starting...\n";
};
// 运行worker
Worker::runAll();
```

name

name

说明:

```
string Worker::$name
```

设置当前Worker实例的名称，方便运行status命令时识别进程。不设置时默认为none。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 设置实例的名称
$worker->name = 'MyWebsocketWorker';
$worker->onWorkerStart = function($worker)
{
    echo "Worker starting...\n";
};
// 运行worker
Worker::runAll();
```

protocol

protocol

要求 ` (workerman >= 3.2.7) `

说明:

```
string Worker::$protocol
```

设置当前Worker实例的协议类。

注：协议处理类可以直接在初始化Worker在监听参数时直接指定。例如

```
$worker = new Worker('http://0.0.0.0:8686');
```

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8686');
$worker->protocol = 'Workerman\Protocols\Http';

$worker->onMessage = function($connection, $data)
{
    var_dump($_GET, $_POST);
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

以上代码等价于下面代码

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

/**
 * 首先会检查用户是否有自定义\Protocols\Http协议类,
 * 如果没有使用workerman内置协议类Workerman\Protocols\Http
 */
$worker = new Worker('http://0.0.0.0:8686');
```

```
$worker->onMessage = function($connection, $data)
{
    var_dump($_GET, $_POST);
    $connection->send("hello");
};

// 运行worker
Worker::runAll();
```

transport

transport

说明:

```
string Worker::$transport
```

设置当前Worker实例所使用的传输层协议，目前只支持3种(tcp、udp、ssl)。不设置默认为tcp。

注意：ssl需要Workerman版本>=3.3.7

范例 1

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('text://0.0.0.0:8484');
// 使用udp协议
$worker->transport = 'udp';
$worker->onMessage = function($connection, $data)
{
    $connection->send('Hello');
};
// 运行worker
Worker::runAll();
```

范例 2

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

// 证书最好是申请的证书
$context = array(
    'ssl' => array(
        'local_cert' => '/etc/nginx/conf.d/ssl/server.pem', // 也可以是crt
        'local_pk' => '/etc/nginx/conf.d/ssl/server.key',
    )
);
// 这里设置的是websocket协议
$worker = new Worker('websocket://0.0.0.0:4431', $context);
// 设置transport开启ssl, websocket+ssl即wss
$worker->transport = 'ssl';
$worker->onMessage = function($con, $msg) {
```

transport

```
        $con->send('ok');  
    };  
    Worker::runAll();
```


reusePort

reusePort

要求 (workerman >= 3.2.1 并且 PHP >= 7.0)

说明:

```
bool Worker::$reusePort
```

设置当前worker是否开启监听端口复用(socket的SO_REUSEPORT选项)，默认为false，不开启。

开启监听端口复用后允许多个无亲缘关系的进程监听相同的端口，并且由系统内核做负载均衡，决定将socket连接交给哪个进程处理，避免了惊群效应，可以提升多进程短连接应用的性能。

注意：此特性需要PHP版本>=7.0

范例 1

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->count = 4;
$worker->reusePort = true;
$worker->onMessage = function($connection, $data)
{
    $connection->send('ok');
};
// 运行worker
Worker::runAll();
```

范例2：workerman多端口(多协议)监听

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('text://0.0.0.0:2015');
$worker->count = 4;
// 每个进程启动后在当前进程新增一个监听
$worker->onWorkerStart = function($worker)
{
    $inner_worker = new Worker('http://0.0.0.0:2016');
```

```
/**
 * 多个进程监听同一个端口（监听套接字不是继承自父进程）
 * 需要开启端口复用，不然会报Address already in use错误
 */
$inner_worker->reusePort = true;
$inner_worker->onMessage = 'on_message';
// 执行监听
$inner_worker->listen();
};

$worker->onMessage = 'on_message';

function on_message($connection, $data)
{
    $connection->send("hello\n");
}

// 运行worker
Worker::runAll();
```

connections

connections

说明:

```
array Worker::$connections
```

格式为

```
array(id=>connection, id=>connection, ...)
```

此属性中存储了当前进程的所有的客户端连接对象，其中id为connection的id编号，详情见手册[TcpConnection的id属性](#)。

、`\$connections` 在广播时或者根据连接id获得连接对象非常有用。

如果得知connection的编号为`\$id`，可以很方便的通过`\$worker->connections[\$id]`获得对应的connection对象，从而操作对应的socket连接，例如通过`\$worker->connections[\$id]->send('...')`发送数据。

范例

```
use Workerman\Worker;
use Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('text://0.0.0.0:2020');
// 进程启动时设置一个定时器，定时向所有客户端连接发送数据
$worker->onWorkerStart = function($worker)
{
    // 定时，每10秒一次
    Timer::add(10, function()use($worker)
    {
        // 遍历当前进程所有的客户端连接，发送当前服务器的时间
        foreach($worker->connections as $connection)
        {
            $connection->send(time());
        }
    });
};
// 运行worker
Worker::runAll();
```


stdoutFile

stdoutFile

说明:

```
static string Worker::$stdoutFile
```

此属性为全局静态属性，如果以守护进程方式(``-d`` 启动)运行，则所有向终端的输出(echo var_dump等)都会被重定向到stdoutFile指定的文件中。

如果不设置，并且是以守护进程方式运行，则所有终端输出全部重定向到 ``/dev/null``。

注意：此属性必须在 ``Worker::runAll();`` 运行前设置才有效。windows系统不支持此特性。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

Worker::$daemonize = true;
// 所有的打印输出全部保存在/tmp/stdout.log文件中
Worker::$stdoutFile = '/tmp/stdout.log';
$worker = new Worker('text://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    echo "Worker start\n";
};
// 运行worker
Worker::runAll();
```

pidFile

pidFile

说明:

```
static string Worker::$pidFile
```

如果无特殊需要，建议不要设置此属性

此属性为全局静态属性，用来设置WorkerMan进程的pid文件路径。

此项设置在监控中比较有用，例如将WorkerMan的pid文件放入固定的目录中，可以方便一些监控软件读取pid文件，从而监控WorkerMan进程状态。

如果不设置，WorkerMan默认会在与Workerman目录平行的位置（注意workerman3.2.3之前版本默认在`sys_get_temp_dir()`下）自动生成一个pid文件，并且为了避免启动多个WorkerMan实例导致pid冲突，WorkerMan生成pid文件包含了当前WorkerMan的路径

注意：此属性必须在`Worker::runAll();`运行前设置才有效。windows系统不支持此特性。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

Worker::$pidFile = '/var/run/workerman.pid';

$worker = new Worker('text://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    echo "Worker start";
};
// 运行worker
Worker::runAll();
```

logFile

logFile

说明:

```
static string Worker::$logFile
```

用来指定workerman日志文件位置。

此文件记录了workerman自身相关的日志，包括启动、停止等。

如果没有设置，文件名默认为workerman.log，文件位置位于Workerman的上一级目录中。

注意：

这个日志文件中仅仅记录workerman自身相关启动停止等日志，不包含任何业务日志。

业务日志类可以利用[file_put_contents](#) 或者 [error_log](#) 等函数自行实现。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

Worker::$logFile = '/tmp/workerman.log';

$worker = new Worker('text://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    echo "Worker start";
};
// 运行worker
Worker::runAll();
```

user

user

说明:

```
string Worker::$user
```

设置当前Worker实例以哪个用户运行。此属性只有当前用户为root时才能生效。不设置时默认以当前用户运行。

建议 ` \$user ` 设置权限较低的用户，例如www-data、apache、nobody等。

注意：此属性必须在 ` Worker::runAll(); ` 运行前设置才有效。windows系统不支持此特性。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 设置实例的运行用户
$worker->user = 'www-data';
$worker->onWorkerStart = function($worker)
{
    echo "Worker starting...\n";
};
// 运行worker
Worker::runAll();
```


reloadable

reloadable

说明:

```
bool Worker::$reloadable
```

设置当前Worker实例是否可以reload，即收到reload信号后是否退出重启。不设置默认为true，收到reload信号后自动重启进程。

有些进程维持着客户端连接，例如Gateway/Worker模型中的gateway进程，当运行reload重新载入业务代码时，却又不想客户端连接断开，则设置gateway进程的reloadable属性为false

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 设置此实例收到reload信号后是否reload重启
$worker->reloadable = false;
$worker->onWorkerStart = function($worker)
{
    echo "Worker starting...\n";
};
// 运行worker
Worker::runAll();
```

daemonize

daemonize

说明:

```
static bool Worker::$daemonize
```

此属性为全局静态属性，表示是否以daemon(守护进程)方式运行。如果启动命令使用了 `-d` 参数，则该属性会自动设置为true。也可以代码中手动设置。

注意：此属性必须在 `Worker::runAll();` 运行前设置才有效。windows系统不支持此特性。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

Worker::$daemonize = true;
$worker = new Worker('text://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    echo "Worker start\n";
};
// 运行worker
Worker::runAll();
```

globalEvent

globalEvent

说明:

```
static Event Worker::$globalEvent
```

此属性为全局静态属性，为全局的eventloop实例，可以向其注册文件描述符的读写事件或者信号事件。

范例

```
use Workerman\Worker;
use Workerman\Events\EventInterface;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker();
$worker->onWorkerStart = function($worker)
{
    echo 'Pid is ' . posix_getpid() . "\n";
    // 当进程收到SIGALRM信号时，打印输出一些信息
    Worker::$globalEvent->add(SIGALRM, EventInterface::EV_SIGNAL, function()
    {
        echo "Get signal SIGALRM\n";
    });
};
// 运行worker
Worker::runAll();
```

测试

Workerman启动后会输出当前进程pid(一个数字)。命令行运行

```
kill -SIGALRM 进程pid
```

服务端会打印出

```
Get signal SIGALRM
```


回调属性

回调属性

onWorkerStart

onWorkerStart

说明:

```
callback Worker::$onWorkerStart
```

设置Worker子进程启动时的回调函数，每个子进程启动时都会执行。

注意：onWorkerStart是在子进程启动时运行的，如果开启了多个子进程(`` $worker->count > 1``)，每个子进程运行一次，则总共会运行 `` $worker->count`` 次。

回调函数的参数

`$worker`

即Worker对象

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    echo "Worker starting...\n";
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onWorkerReload

onWorkerReload

要求 (workerman >= 3.2.5)

说明:

```
callback Worker::$onWorkerReload
```

此特性不常用到。

设置Worker收到reload信号后执行的回调。

可以利用onWorkerReload回调做很多事情，例如在不需要重启进程的情况下重新加载业务配置文件。

注意：

子进程收到reload信号默认的动作是退出重启，以便新进程重新加载业务代码完成代码更新。所以reload后子进程在执行完onWorkerReload回调后便立刻退出是正常现象。

如果在收到reload信号后只想让子进程执行onWorkerReload，不想退出，可以在初始化Worker实例时设置对应的Worker实例的reloadable属性为false。

回调函数的参数

\$worker

即Worker对象

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 设置reloadable为false, 即子进程收到reload信号不执行重启
$worker->reloadable = false;
// 执行reload后告诉所有客户端服务端执行了reload
$worker->onWorkerReload = function($worker)
{
    foreach($worker->connections as $connection)
    {
        $connection->send('worker reloading');
    }
}
```

本文档使用 [看云](#) 构建

```
    }  
};  
// 运行worker  
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onConnect

onConnect

说明:

```
callback Worker::$onConnect
```

当客户端与Workerman建立连接时(TCP三次握手完成后)触发的回调函数。每个连接只会触发一次 `onConnect` 回调。

注意：onConnect事件仅代表客户端与Workerman完成了TCP三次握手，这时客户端还没有发来任何数据，此时除了通过 `$connection->getRemoteIp()` 获得对方ip，没有其他可以鉴别客户端的数据或者信息，所以在onConnect事件里无法确认对方是谁。要想知道对方是谁，需要客户端发送鉴权数据，例如某个token或者用户名密码之类，在[onMessage回调](#)里做鉴权。

回调函数的参数

`$connection`

连接对象，即[TcpConnection实例](#)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    echo "new connection from ip " . $connection->getRemoteIp() . "\n";
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onMessage

onMessage

说明:

```
callback Worker::$onMessage
```

当客户端通过连接发来数据时(Workerman收到数据时)触发的回调函数

回调函数的参数

`$connection`

连接对象，即[TcpConnection实例](#)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

`$data`

客户端连接上发来的数据，如果Worker指定了协议，则\$data是对应协议decode（解码）了的数据

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onMessage = function($connection, $data)
{
    var_dump($data);
    $connection->send('receive success');
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onClose

onClose

说明:

```
callback Worker::$onClose
```

当客户端连接与Workerman断开时触发的回调函数。不管连接是如何断开的，只要断开就会触发 `onClose`。每个连接只会触发一次 `onClose`。

注意：如果对端是由于断网或者断电等极端情况断开的连接，这时由于无法及时发送tcp的fin包给workerman，workerman就无法得知连接已经断开，也就无法及时触发 `onClose`。这种情况需要通过应用层心跳来解决。workerman中连接的心跳实现参见[这里](#)。如果使用的是GatewayWorker框架，则直接使用GatewayWorker框架的心跳机制即可，参见[这里](#)。

回调函数的参数

`$connection`

连接对象，即[TcpConnection实例](#)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onClose = function($connection)
{
    echo "connection closed\n";
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onBufferFull

onBufferFull

说明:

```
callback Worker::$onBufferFull
```

每个连接都有一个单独的应用层发送缓冲区，缓冲区大小由 `TcpConnection::$maxSendBufferSize` 决定，默认值为1MB，可以手动设置更改大小，更改后会对所有连接生效。

该回调可能会在调用`Connection::send`后立刻被触发，比如发送大数据或者连续快速的向对端发送数据，由于网络等原因数据被大量积压在对应连接的发送缓冲区，当超过 `TcpConnection::$maxSendBufferSize` 上限时触发。

当发生onBufferFull事件时，开发者一般需要采取措施，例如停止向对端发送数据，等待发送缓冲区的数据被发送完毕(onBufferDrain事件)等。

当调用`Connection::send($A)`后导致触发onBufferFull时，不管本次send的数据 `$A` 多大，即使大于 `TcpConnection::$maxSendBufferSize`，本次要发送的数据仍然会被放入发送缓冲区。也就是说发送缓冲区实际放入的数据可能远远大于 `TcpConnection::$maxSendBufferSize`，当发送缓冲区的数据已经大于 `TcpConnection::$maxSendBufferSize` 时，仍然继续`Connection::send($B)`数据，则这次send的 `$B` 数据不会放入发送缓冲区，而是被丢弃掉，并触发onError回调。

总结来说，只要发送缓冲区还没满，哪怕只有一个字节的空间，调用`Connection::send($A)`肯定会把 `$A` 放入发送缓冲区，如果放入发送缓冲区后，发送缓冲区大小超过了 `TcpConnection::$maxSendBufferSize` 限制，则会触发onBufferFull回调。

回调函数的参数

`$connection`

连接对象，即[TcpConnection实例]315157)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

范例

本文档使用 [看云](#) 构建

```
use Workerman\Worker;  
require_once __DIR__ . '/Workerman/Autoloader.php';  
  
$worker = new Worker('websocket://0.0.0.0:8484');  
$worker->onBufferFull = function($connection)  
{  
    echo "bufferFull and do not send again\n";  
};  
// 运行worker  
Worker::runAll();
```

参见

onBufferDrain 当连接的应用层发送缓冲区数据全部发送完毕时触发

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

onBufferDrain

onBufferDrain

说明:

```
callback Worker::$onBufferDrain
```

每个连接都有一个单独的应用层发送缓冲区，缓冲区大小由 `TcpConnection::$maxSendBufferSize` 决定，默认值为1MB，可以手动设置更改大小，更改后会对所有连接生效。

该回调在应用层发送缓冲区数据全部发送完毕后触发。一般与onBufferFull配合使用，例如在onBufferFull时停止向对端继续send数据，在onBufferDrain恢复写入数据。

回调函数的参数

`$connection`

连接对象，即[TcpConnection实例](#)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onBufferFull = function($connection)
{
    echo "bufferFull and do not send again\n";
};
$worker->onBufferDrain = function($connection)
{
    echo "buffer drain and continue send\n";
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

参见

onBufferFull 当连接的应用层发送缓冲区满时触发

onError

onError

说明:

```
callback Worker::$onError
```

当客户端的连接上发生错误时触发。

目前错误类型有

1、调用Connection::send由于客户端连接断开导致的失败（紧接着会触发onClose回调）

```
(code:WORKERMAN_SEND_FAIL msg:client closed) `
```

2、在触发onBufferFull后(发送缓冲区已满)，仍然调用Connection::send，并且发送缓冲区仍然是满的状态导致发送失败(不会触发onClose回调)

```
(code:WORKERMAN_SEND_FAIL msg:send buffer full and drop package) `
```

3、使用AsyncTcpConnection异步连接失败时(紧接着会触发onClose回调)

```
(code:WORKERMAN_CONNECT_FAIL msg:stream_socket_client返回的错误消息) `
```

回调函数的参数

```
$connection
```

连接对象，即[TcpConnection实例](#)，用于操作客户端连接，如[发送数据](#)，[关闭连接](#)等

```
$code
```

错误码

```
$msg
```

错误消息

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
```

```
$worker->onError = function($connection, $code, $msg)
{
    echo "error $code $msg\n";
};
// 运行worker
Worker::runAll();
```

提示：除了使用匿名函数作为回调，还可以[参考这里](#)使用其它回调写法。

接口

接口

runAll

runAll

```
void Worker::runAll(void)
```

运行所有Worker实例。

注意：

Worker::runAll()执行后将永久阻塞，也就是说位于Worker::runAll()后面的代码将不会被执行。所有Worker实例化应该都在Worker::runAll()前进行。

参数

无参数

返回值

无返回

范例 运行多个Worker实例

start.php

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

$http_worker = new Worker("http://0.0.0.0:2345");
$http_worker->onMessage = function($connection, $data)
{
    $connection->send('hello http');
};

$ws_worker = new Worker('websocket://0.0.0.0:4567');
$ws_worker->onMessage = function($connection, $data)
{
    $connection->send('hello websocket');
};

// 运行所有Worker实例
Worker::runAll();
```

注意：

windows版本的workerman不支持在同一个文件中实例化多个Worker。
上面的例子无法在windows版本的workerman下运行。

windows版本的workerman需要将多个Worker实例初始化放在不同的文件中，像下面这样

start_http.php

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

$http_worker = new Worker("http://0.0.0.0:2345");
$http_worker->onMessage = function($connection, $data)
{
    $connection->send('hello http');
};

// 运行所有Worker实例(这里只有一个实例)
Worker::runAll();
```

start_websocket.php

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

$ws_worker = new Worker('websocket://0.0.0.0:4567');
$ws_worker->onMessage = function($connection, $data)
{
    $connection->send('hello websocket');
};

// 运行所有Worker实例(这里只有一个实例)
Worker::runAll();
```

stopAll

stopAll

```
void Worker::stopAll(void)
```

停止当前进程（子进程）的所有Worker实例并退出。

此方法用于安全退出当前子进程，作用相当于调用exit/die退出当前子进程。

与直接调用exit/die区别是，直接调用exit或者die无法触发onWorkerStop回调，并且会导致一条WORKER EXIT UNEXPECTED错误日志。

参数

无参数

返回值

无返回

范例 max_request

下面例子子进程每处理完1000个请求后执行stopAll退出，以便重新启动一个全新进程。类似php-fpm的max_request属性，主要用于解决php业务代码bug引起的内存泄露问题。

start.php

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

// 每个进程最多执行1000个请求
define('MAX_REQUEST', 1000);

$http_worker = new Worker("http://0.0.0.0:2345");
$http_worker->onMessage = function($connection, $data)
{
    // 已经处理请求数
    static $request_count = 0;

    $connection->send('hello http');
    // 如果请求数达到1000
    if(++$request_count >= MAX_REQUEST)
    {
        /*
```

```
        * 退出当前进程，主进程会立刻重新启动一个全新进程补充上来
        * 从而完成进程重启
        */
    Worker::stopAll();
};

Worker::runAll();
```


listen

listen

```
void Worker::listen(void)
```

用于实例化Worker后执行监听。

此方法主要用于在Worker进程启动后动态创建新的Worker实例，能够实现同一个进程监听多个端口，支持多种协议。

例如一个http Worker启动后实例化一个websocket Worker，那么这个进程即能通过http协议访问，又能通过websocket协议访问。由于websocket Worker和http Worker在同一个进程中，所以它们可以访问共同的内存变量，共享所有socket连接。可以做到接收http请求，然后操作websocket客户端完成向客户端推送数据类似的效果。

注意：

如果PHP版本 ≤ 7.0 ，则不支持在多个子进程中实例化相同端口的Worker。例如A进程创建了监听2016端口的Worker，那么B进程就不能再创建监听2016端口的Worker，否则会报`Address already in use`错误。例如下面的代码是`无法`运行的。

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker();
// 4个进程
$worker->count = 4;
// 每个进程启动后在当前进程新增一个Worker监听
$worker->onWorkerStart = function($worker)
{
    /**
     * 4个进程启动的时候都创建2016端口的Worker
     * 当执行到worker->listen()时会报Address already in use错误
     * 如果worker->count=1则不会报错
     */
    $inner_worker = new Worker('http://0.0.0.0:2016');
    $inner_worker->onMessage = 'on_message';
    // 执行监听。这里会报Address already in use错误
    $inner_worker->listen();
};

$worker->onMessage = 'on_message';
```

```
function on_message($connection, $data)
{
    $connection->send("hello\n");
}

// 运行worker
Worker::runAll();
```

如果您的PHP版本 ≥ 7.0 ，可以设置`Worker->reusePort=true`，这样可以做到多个子进程创建相同端口的Worker。见下面的例子：

```
use Workerman\Worker;
require_once './Workerman/Autoloader.php';

$worker = new Worker('text://0.0.0.0:2015');
// 4个进程
$worker->count = 4;
// 每个进程启动后在当前进程新增一个Worker监听
$worker->onWorkerStart = function($worker)
{
    $inner_worker = new Worker('http://0.0.0.0:2016');
    // 设置端口复用，可以创建监听相同端口的Worker（需要PHP $\geq 7.0$ ）
    $inner_worker->reusePort = true;
    $inner_worker->onMessage = 'on_message';
    // 执行监听。正常监听不会报错
    $inner_worker->listen();
};

$worker->onMessage = 'on_message';

function on_message($connection, $data)
{
    $connection->send("hello\n");
}

// 运行worker
Worker::runAll();
```

示例 php后端及时推送消息给客户端

原理：

- 1、建立一个websocket Worker，用来维持客户端长连接
- 2、websocket Worker内部建立一个text Worker
- 3、websocket Worker 与 text Worker是同一个进程，可以方便的共享客户端连接
- 4、某个独立的php后台系统通过text协议与text Worker通讯

5、text Worker操作websocket连接完成数据推送

代码及步骤

push.php

```
<?php
use Workerman\Worker;
require_once './Workerman/Autoloader.php';
// 初始化一个worker容器, 监听1234端口
$worker = new Worker('websocket://0.0.0.0:1234');

/*
 * 注意这里进程数必须设置为1, 否则会报端口占用错误
 * (php 7可以设置进程数大于1, 前提是$inner_text_worker->reusePort=true)
 */
$worker->count = 1;
// worker进程启动后创建一个text Worker以便打开一个内部通讯端口
$worker->onWorkerStart = function($worker)
{
    // 开启一个内部端口, 方便内部系统推送数据, Text协议格式 文本+换行符
    $inner_text_worker = new Worker('text://0.0.0.0:5678');
    $inner_text_worker->onMessage = function($connection, $buffer)
    {
        // $data数组格式, 里面有uid, 表示向那个uid的页面推送数据
        $data = json_decode($buffer, true);
        $uid = $data['uid'];
        // 通过workerman, 向uid的页面推送数据
        $ret = sendMessageByUid($uid, $buffer);
        // 返回推送结果
        $connection->send($ret ? 'ok' : 'fail');
    };
    // ## 执行监听 ##
    $inner_text_worker->listen();
};
// 新增加一个属性, 用来保存uid到connection的映射
$worker->uidConnections = array();
// 当有客户端发来消息时执行的回调函数
$worker->onMessage = function($connection, $data)
{
    global $worker;
    // 判断当前客户端是否已经验证, 既是否设置了uid
    if(!isset($connection->uid))
    {
        // 没验证的话把第一个包当做uid (这里为了方便演示, 没做真正的验证)
        $connection->uid = $data;
        /* 保存uid到connection的映射, 这样可以方便的通过uid查找connection,
         * 实现针对特定uid推送数据
         */
        $worker->uidConnections[$connection->uid] = $connection;
        return;
    }
};
```

```

// 当有客户端连接断开时
$worker->onClose = function($connection)
{
    global $worker;
    if(isset($connection->uid))
    {
        // 连接断开时删除映射
        unset($worker->uidConnections[$connection->uid]);
    }
};

// 向所有验证的用户推送数据
function broadcast($message)
{
    global $worker;
    foreach($worker->uidConnections as $connection)
    {
        $connection->send($message);
    }
}

// 针对uid推送数据
function sendMessageByUid($uid, $message)
{
    global $worker;
    if(isset($worker->uidConnections[$uid]))
    {
        $connection = $worker->uidConnections[$uid];
        $connection->send($message);
        return true;
    }
    return false;
}

// 运行所有的worker
Worker::runAll();

```

启动后端服务

```
` php push.php start -d `
```

前端接收推送的js代码

```

var ws = new WebSocket('ws://127.0.0.1:1234');
ws.onopen = function(){
    var uid = 'uid1';
    ws.send(uid);
};
ws.onmessage = function(e){
    alert(e.data);
};

```

后端推送消息的代码

```
// 建立socket连接到内部推送端口
$client = stream_socket_client('tcp://127.0.0.1:5678', $errno, $errmsg, 1
);
// 推送的数据, 包含uid字段, 表示是给这个uid推送
$data = array('uid'=>'uid1', 'percent'=>'88%');
// 发送数据, 注意5678端口是Text协议的端口, Text协议需要在数据末尾加上换行符
fwrite($client, json_encode($data)."\n");
// 读取推送结果
echo fread($client, 8192);
```

TcpConnection类

Connection类提供的接口

WorkerMan中有两个重要的类Worker与Connection。

每个客户端连接对应一个Connection对象，可以设置对象的onMessage、onClose等回调，同时提供了向客户端发送数据send接口与关闭连接close接口，以及其它一些必要的接口。

可以说Worker是一个监听容器，负责接受客户端连接，并把连接包装成connection对象式提供给开发者操作。

属性

id
protocol
worker
maxSendBufferSize
defaultMaxSendBufferSize
maxPackageSize

id

id

说明:

```
int Connection::$id
```

连接的id。这是一个自增的整数。

注意：workerman是多进程的，每个进程内部会维护一个自增的connection id，所以多个进程之间的connection id会有重复。

如果想要不重复的connection id 可以根据需要给connection->id重新赋值，例如加上worker->id前缀。

参见

[Worker的connections属性](#)

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    echo $connection->id;
};
// 运行worker
Worker::runAll();
```


protocol

protocol

说明:

```
string Connection::$protocol
```

设置当前连接的协议类

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    $connection->protocol = 'Workerman\\Protocols\\Http';
};
$worker->onMessage = function($connection, $data)
{
    var_dump($_GET, $_POST);
    // send 时会自动调用$connection->protocol::encode(), 打包数据后再发送
    $connection->send("hello");
};
// 运行worker
Worker::runAll();
```

worker

worker

说明:

```
Worker Connection::$worker
```

此属性为只读属性，即当前connection对象所属的worker实例

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');

// 当一个客户端发来数据时，转发给当前进程所维护的其它所有客户端
$worker->onMessage = function($connection, $data)
{
    foreach($connection->worker->connections as $con)
    {
        $con->send($data);
    }
};
// 运行worker
Worker::runAll();
```

maxSendBufferSize

maxSendBufferSize

说明:

```
int Connection::$maxSendBufferSize
```

此属性用来设置当前连接的应用层发送缓冲区大小。不设置默认为 `Connection::\$defaultMaxSendBufferSize` (1MB)。`Connection::\$maxSendBufferSize` 和 `Connection::\$defaultMaxSendBufferSize` 均可动态设置。

此属性影响onBufferFull回调

范例

```
use Workerman\Worker;
use Workerman\Connection\TcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    // 设置当前连接的应用层发送缓冲区大小为102400字节
    $connection->maxSendBufferSize = 102400;
};
// 运行worker
Worker::runAll();
```

defaultMaxSendBufferSize

defaultMaxSendBufferSize

说明:

```
static int Connection::$defaultMaxSendBufferSize
```

此属性为全局静态属性，用来设置所有连接的默认应用层发送缓冲区大小。不设置默认为 `1MB`。`Connection::\$defaultMaxSendBufferSize` 可以动态设置，设置后只对之后产生的新连接有效

此属性影响onBufferFull回调

范例

```
use Workerman\Worker;
use Workerman\Connection\TcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 设置所有连接的默认应用层发送缓冲区大小
TcpConnection::$defaultMaxSendBufferSize = 2*1024*1024;

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    // 设置当前连接的应用层发送缓冲区大小，会覆盖掉默认值
    $connection->maxSendBufferSize = 4*1024*1024;
};
// 运行worker
Worker::runAll();
```

maxPackageSize

maxPackageSize

说明:

```
static int Connection::$maxPackageSize
```

此属性为全局静态属性，用来设置每个连接能够接收的最大包长。不设置默认为10MB。

如果发来的数据包解析(协议类的input方法返回值)得到包长大于 `Connection::\$maxPackageSize`，则会视为非法数据，连接会断开。

范例

```
use Workerman\Worker;
use Workerman\Connection\TcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 设置每个连接接收的数据包最大为1024000字节
TcpConnection::$maxPackageSize = 1024000;

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onMessage = function($connection, $data)
{
    $connection->send('hello');
};
// 运行worker
Worker::runAll();
```

回调属性

`onMessage`

`onClose`

`onBufferFull`

`onBufferDrain`

`onError`

onMessage

onMessage

说明:

```
callback Connection::$onMessage
```

作用与[Worker::\\$onMessage](#)回调相同，区别是只针对当前连接有效，也就是可以针对某个连接的设置onMessage回调。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 当有客户端连接事件时
$worker->onConnect = function($connection)
{
    // 设置连接的onMessage回调
    $connection->onMessage = function($connection, $data)
    {
        var_dump($data);
        $connection->send('receive success');
    };
};
// 运行worker
Worker::runAll();
```

上面代码与下面的效果是一样的

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 直接设置所有连接的onMessage回调
$worker->onMessage = function($connection, $data)
{
    var_dump($data);
    $connection->send('receive success');
};
// 运行worker
Worker::runAll();
```


onClose

onClose

说明:

```
callback Connection::$onClose
```

此回调与[Worker::\\$onClose](#)回调作用相同，区别是只针对当前连接有效,也就是可以针对某个连接的设置onClose回调。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 当有链接事件时触发
$worker->onConnect = function($connection)
{
    // 设置连接的onClose回调
    $connection->onClose = function($connection)
    {
        echo "connection closed\n";
    };
};
// 运行worker
Worker::runAll();
```

上面代码与下面的效果相同

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
// 设置所有连接的onclose回调
$worker->onClose = function($connection)
{
    echo "connection closed\n";
};
// 运行worker
Worker::runAll();
```


onBufferFull

onBufferFull

说明:

```
callback Connection::$onBufferFull
```

作用与[Worker::\\$onBufferFull](#)回调相同，区别是只针对当前连接起作用，即可以单独设置某个连接的onBufferFull回调

onBufferDrain

onBufferDrain

说明:

```
callback Connection::$onBufferDrain
```

作用与[Worker::\\$onBufferDrain](#)回调相同，区别是只针对当前连接起作用，即可以单独设置某个连接的onBufferDrain回调

onError

onError

说明:

```
callback Connection::$onError
```

作用与[Worker::\\$onError](#)回调相同，区别是只针对当前连接起作用，即可以单独设置某个连接的onError回调

接口

接口

send

send

说明:

```
mixed Connection::send(mixed $data [, $raw = false])
```

向客户端发送数据

参数

`$data`

要发送的数据，如果在初始化Worker类时指定了协议，则会自动调用协议的encode方法,完成协议打包工作后发送给客户端

`$raw`

是否发送原始数据，即不调用协议的encode方法，默认是false，即自动调用协议的encode方法

返回值

true 表示数据已经成功写入到该连接的操作系统层的socket发送缓冲区

null 表示数据已经写入到该连接的应用层发送缓冲区，等待向系统层socket发送缓冲区写入

false 表示发送失败，失败原因可能是客户端连接已经关闭，或者该连接的应用层发送缓冲区已满

注意

send返回 `true`，仅代表数据已经成功写入到该连接的操作系统socket发送缓冲区，并不意味着数据已经成功的发送给对端socket接收缓冲区，更不意味着对端应用程序已经从本地socket接收缓冲区读取了数据。不过即便如此，只要send不返回false并且网络没有断开，而且客户端接收正常，数据基本上可以看做100%能发到对方的。

由于socket发送缓冲区的数据是由操作系统异步发送给对端的，操作系统并没有给应用层提供相应确认机制，所以应用层无法得知socket发送缓冲区的数据何时开始发送，应用层更无法得知socket发送缓冲区的数据是否发送成功。基于以上原因workerman无法直接提消息确认接口。

如果业务需要保证每个消息客户端都收到，可以在业务上增加一种确认机制。确认机制可能根据业务不同而不同，即使同样的业务确认机制也可以有多种方法。

例如聊天系统可以用这样的确认机制。把每条消息都存入数据库，每条消息都有一个是否已读字段。客户端每收到一条消息向服务端发送一个确认包，服务端将对应消息置为已读。当客户端连接到服务端时（一般是用户登录或者断线重连），查询数据库是否有未读的消息，有的话发给客户端，同样客户端收到消息后通知服务端已读。这样可以保证每个消息对方都能收到。当然开发者也可以用自己的确认逻辑。

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onMessage = function($connection, $data)
{
    // 会自动调用\Workerman\Protocols\WebSocket::encode打包成websocket协议数据后发送
    $connection->send("hello\n");
};
// 运行worker
Worker::runAll();
```


getRemoteIp

getRemoteIp

说明:

```
string Connection::getRemoteIp()
```

获得该连接的客户端ip

参数

无参数

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    echo "new connection from ip " . $connection->getRemoteIp() . "\n";
};
// 运行worker
Worker::runAll();
```

getRemotePort

getRemotePort

说明:

```
int Connection::getRemotePort()
```

获得该连接的客户端端口

参数

无参数

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    echo "new connection from address " .
        $connection->getRemoteIp() . ":" . $connection->getRemotePort() . "\n";
};
// 运行worker
Worker::runAll();
```

close

close

说明:

```
void Connection::close(mixed $data = '')
```

安全的关闭连接.

调用close会等待发送缓冲区的数据发送完毕后才关闭连接, 并触发连接的 `onClose` 回调。

参数

`$data`

可选参数, 要发送的数据 (如果有指定协议, 则会自动调用协议的encode方法打包 `$data` 数据), 当数据发送完毕后关闭连接, 随后会触发onClose回调

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onMessage = function($connection, $data)
{
    $connection->close("hello\n");
};
// 运行worker
Worker::runAll();
```

destroy

destroy

说明:

```
void Connection::destroy()
```

立刻关闭连接。

与close不同之处是，调用destroy后即使该连接的发送缓冲区还有数据未发送到对端，连接也会立刻被关闭，并立刻触发该连接的 `onClose` 回调。

参数

无参数

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onMessage = function($connection, $data)
{
    // if something wrong
    $connection->destroy();
};
// 运行worker
Worker::runAll();
```

pauseRecv

pauseRecv

说明:

```
void Connection::pauseRecv(void)
```

使当前连接停止接收数据。该连接的onMessage回调将不会被触发。此方法对于上传流量控制非常有用

参数

无参数

范例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    // 给connection对象动态添加一个属性，用来保存当前连接发来多少个请求
    $connection->messageCount = 0;
};
$worker->onMessage = function($connection, $data)
{
    // 每个连接接收100个请求后就不再接收数据
    $limit = 100;
    if(++$connection->messageCount > $limit)
    {
        $connection->pauseRecv();
    }
};
// 运行worker
Worker::runAll();
```

参见

void Connection::resumeRecv(void) 使得对应连接对象恢复接收数据

resumeRecv

resumeRecv

说明:

```
void Connection::resumeRecv(void)
```

使当前连接继续接收数据。此方法与Connection::pauseRecv配合使用，对于上传流量控制非常有用

参数

无参数

范例

```
use Workerman\Worker;
use Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onConnect = function($connection)
{
    // 给connection对象动态添加一个属性，用来保存当前连接发来多少个请求
    $connection->messageCount = 0;
};
$worker->onMessage = function($connection, $data)
{
    // 每个连接接收100个请求后就不再接收数据
    $limit = 100;
    if(++$connection->messageCount > $limit)
    {
        $connection->pauseRecv();
        // 30秒后恢复接收数据
        Timer::add(30, function($connection){
            $connection->resumeRecv();
        }, array($connection), false);
    }
};
// 运行worker
Worker::runAll();
```

参见

void Connection::pauseRecv(void) 使得对应连接对象停止接收数据

pipe

pipe 说明:

```
void Connection::pipe(TcpConnection $target_connection)
```

参数

将当前连接的数据流导入到目标连接。内置了流量控制。此方法做TCP代理非常有用

范例 TCP代理

```
<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('tcp://0.0.0.0:8483');
$worker->count = 12;

// tcp连接建立后
$worker->onConnect = function($connection)
{
    // 建立本地80端口的异步连接
    $connection_to_80 = new AsyncTcpConnection('tcp://127.0.0.1:80');
    // 设置将当前客户端连接的数据导向80端口的连接
    $connection->pipe($connection_to_80);
    // 设置80端口连接返回的数据导向客户端连接
    $connection_to_80->pipe($connection);
    // 执行异步连接
    $connection_to_80->connect();
};

// 运行worker
Worker::runAll();
```

AsyncTcpConnection类

AsyncTcpConnection类

AsyncTcpConnection是TcpConnection的子类，拥有与TcpConnection一样的属性与接口。AsyncTcpConnection用于异步创建一个TcpConnection连接。

构造函数

__construct 方法

```
void AsyncTcpConnection::__construct(string $remote_address, $context_option = null)
```

创建一个异步连接对象。

AsyncTcpConnection可以让Workerman作为客户端向远程服务端发起异步连接，并通过send接口和onMessage回调异步发送和处理连接上的数据。

参数

参数: remote_address

连接的地址，例如

```
tcp://www.baidu.com:80
```

```
ssl://www.baidu.com:443
```

```
ws://echo.websocket.org:80
```

```
frame://192.168.1.1:8080
```

```
text://192.168.1.1:8080
```

参数: \$context_option

、 此参数要求 (workerman >= 3.3.5) 、

用来设置socket上下文，例如利用 、 bindto 、 设置以哪个(网卡)ip和端口访问外部网络，设置ssl证书等。

参考 [stream_context_create](#)、[套接字上下文选项](#)、[SSL 上下文选项](#)

注意

目前AsyncTcpConnection支持的协议有[tcp](#)、[ssl](#)、[ws](#)、[frame](#)、[text](#)。

同时支持自定义协议，参见[如何自定义协议](#)

其中[ssl](#)要求Workerman>=3.3.4，并安装[openssl](#)扩展。

目前不支持[http](#)协议的AsyncTcpConnection。

可以用 `new AsyncTcpConnection('ws://...')` 像浏览器一样在workerman里发起websocket连接远程websocket服务器，见[示例](#)。但是不能以 `new AsyncTcpConnection('websocket://...')` 的形式在workerman里发起websocket连接。

示例

示例 1、异步访问外部http服务

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$task = new Worker();
// 进程启动时异步建立一个到www.baidu.com连接对象，并发送数据获取数据
$task->onWorkerStart = function($task)
{
    // 不支持直接指定http，但是可以用tcp模拟http协议发送数据
    $connection_to_baidu = new AsyncTcpConnection('tcp://www.baidu.com:80');
    // 当连接建立成功时，发送http请求数据
    $connection_to_baidu->onConnect = function($connection_to_baidu)
    {
        echo "connect success\n";
        $connection_to_baidu->send("GET / HTTP/1.1\r\nHost: www.baidu.com\r\nConnection: keep-alive\r\n\r\n");
    };
    $connection_to_baidu->onMessage = function($connection_to_baidu, $http_buffer)
    {
        echo $http_buffer;
    };
    $connection_to_baidu->onClose = function($connection_to_baidu)
    {
        echo "connection closed\n";
    };
    $connection_to_baidu->onError = function($connection_to_baidu, $code, $msg)
    {
        echo "Error code:$code msg:$msg\n";
    };
    $connection_to_baidu->connect();
};
```

```
// 运行worker
Worker::runAll();
```

示例 2、异步访问外部websocket服务，并设置以哪个本地ip及端口访问

```
<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker){
    // 设置访问对方主机的本地ip及端口(每个socket连接都会占用一个本地端口)
    $context_option = array(
        'socket' => array(
            // ip必须是本机网卡ip, 并且能访问对方主机, 否则无效
            'bindto' => '114.215.84.87:2333',
        ),
    );

    $con = new AsyncTcpConnection('ws://echo.websocket.org:80', $context_option);

    $con->onConnect = function($con) {
        $con->send('hello');
    };

    $con->onMessage = function($con, $data) {
        echo $data;
    };

    $con->connect();
};

Worker::runAll();
```

示例 3、异步访问外部wss端口，并设置本地ssl证书

```
<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/../Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker){
    // 设置访问对方主机的本地ip及端口以及ssl证书
    $context_option = array(
        'socket' => array(
            // ip必须是本机网卡ip, 并且能访问对方主机, 否则无效
            'bindto' => '114.215.84.87:2333',
```

```

    ),
    // ssl选项, 参考http://php.net/manual/zh/context.ssl.php
    'ssl' => array(
        // 本地证书路径。 必须是 PEM 格式, 并且包含本地的证书及私钥。
        'local_cert' => '/your/path/to/pemfile',
        // local_cert 文件的密码。
        'passphrase' => 'your_pem_passphrase',
        // 是否允许自签名证书。
        'allow_self_signed' => true,
        // 是否需要验证 SSL 证书。
        'verify_peer' => false
    )
);

// 发起异步连接
$con = new AsyncTcpConnection('ws://echo.websocket.org:443', $context_option);

// 设置以ssl加密方式访问
$con->transport = 'ssl';

$con->onConnect = function($con) {
    $con->send('hello');
};

$con->onMessage = function($con, $data) {
    echo $data;
};

$con->connect();
};

Worker::runAll();

```

connect

connect 方法

```
void AsyncTcpConnection::connect()
```

执行异步连接操作。此方法会立刻返回。

注意：如果需要设置异步连接的onError回调，则应该在connect执行之前设置，否则onError回调可能无法被触发，例如下面的例子onError回调可能无法触发，无法捕获异步连接失败事件。

```
$connection = new AsyncTcpConnection('tcp://baidu.com:81');
// 执行连接的时候还没设置onError回调
$connection->connect();
$connection->onError = function($connection, $err_code, $err_msg)
{
    echo "$err_code, $err_msg";
};
```

参数

无参数

返回值

无返回值

示例 Mysql代理

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 真实的mysql地址，假设这里是本机3306端口
$REAL_MYSQL_ADDRESS = 'tcp://127.0.0.1:3306';

// 代理监听本地4406端口
$proxy = new Worker('tcp://0.0.0.0:4406');

$proxy->onConnect = function($connection)
{
    global $REAL_MYSQL_ADDRESS;
    // 异步建立一个到实际mysql服务器的连接
    $connection_to_mysql = new AsyncTcpConnection($REAL_MYSQL_ADDRESS);
    // mysql连接发来数据时，转发给对应客户端的连接
```

```

    $connection_to_mysql->onMessage = function($connection_to_mysql, $buffer)use($connection)
    {
        $connection->send($buffer);
    };
    // mysql连接关闭时, 关闭对应的代理到客户端的连接
    $connection_to_mysql->onClose = function($connection_to_mysql)use($connection)
    {
        $connection->close();
    };
    // mysql连接上发生错误时, 关闭对应的代理到客户端的连接
    $connection_to_mysql->onError = function($connection_to_mysql)use($connection)
    {
        $connection->close();
    };
    // 执行异步连接
    $connection_to_mysql->connect();

    // 客户端发来数据时, 转发给对应的mysql连接
    $connection->onMessage = function($connection, $buffer)use($connection_to_mysql)
    {
        $connection_to_mysql->send($buffer);
    };
    // 客户端连接断开时, 断开对应的mysql连接
    $connection->onClose = function($connection)use($connection_to_mysql)
    {
        $connection_to_mysql->close();
    };
    // 客户端连接发生错误时, 断开对应的mysql连接
    $connection->onError = function($connection)use($connection_to_mysql)
    {
        $connection_to_mysql->close();
    };
};
// 运行worker
Worker::runAll();

```

测试

```
mysql -uroot -P4406 -h127.0.0.1 -p
```

```

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25004
Server version: 5.5.31-1~dotdeb.0 (Debian)

```

```
Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its
```


affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

reconnect

reConnect 方法

```
void AsyncTcpConnection::reConnect(float $delay = 0)
```

(要求Workerman版本>=3.3.5)

重连。一般在 ` onClose ` 回调中调用，实现断线重连。

由于网络问题或者对方服务重启等原因导致连接断开，则可以通过调用此方法实现重连。

参数

\$delay

延迟多久后执行重连。单位为秒，支持小数，可精确到毫秒。

如果不传或者值为0则代表立即重连。

最好传递参数让重连延迟执行，避免因为对端服务问题一直不可连导致本机cpu消耗过高。

返回值

无返回值

示例

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker)
{
    $con = new AsyncTcpConnection('ws://echo.websocket.org:80');
    $con->onConnect = function($con) {
        $con->send('hello');
    };
    $con->onMessage = function($con, $msg) {
        echo "recv $msg\n";
    };
    $con->onClose = function($con) {
        // 如果连接断开，则在1秒后重连
        $con->reConnect(1);
    };
    $con->connect();
};
```

reconnect

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

transport

transport属性

要求 (workerman >= 3.3.4)

设置传输属性，可选值为 [tcp](#) 和 [ssl](#)，默认是tcp。

transport为 [ssl](#) 时，要求PHP必须安装[openssl](#)扩展。

当把Workerman作为客户端向服务端发起ssl加密连接(https连接、wss连接等)时请设置此项为 `ssl`，例如下面的例子。

示例 (https连接)

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

$task = new Worker();
// 进程启动时异步建立一个到www.baidu.com连接对象，并发送数据获取数据
$task->onWorkerStart = function($task)
{
    $connection_to_baidu = new AsyncTcpConnection('tcp://www.baidu.com:443');

    // 设置为ssl加密连接
    $connection_to_baidu->transport = 'ssl';

    $connection_to_baidu->onConnect = function($connection_to_baidu)
    {
        echo "connect success\n";
        $connection_to_baidu->send("GET / HTTP/1.1\r\nHost: www.baidu.com\r\nConnection: keep-alive\r\n\r\n");
    };
    $connection_to_baidu->onMessage = function($connection_to_baidu, $http_buffer)
    {
        echo $http_buffer;
    };
    $connection_to_baidu->onClose = function($connection_to_baidu)
    {
        echo "connection closed\n";
    };
    $connection_to_baidu->onError = function($connection_to_baidu, $code, $msg)
    {
        echo "Error code:$code msg:$msg\n";
    };
};
```

```
$connection_to_baidu->connect();  
};  
  
// 运行worker  
Worker::runAll();
```

Timer定时器类

[add](#)

[del](#)

[定时器注意事项](#)

add

add

```
int \Workerman\Lib\Timer::add(float $time_interval, callable $callback [,
    $args = array(), bool $persistent = true])
```

定时执行某个函数或者类方法。

注意：定时器是在当前进程中运行的，workerman中不会创建新的进程或者线程去运行定时器。

参数

`time_interval`

多长时间执行一次，单位秒，支持小数，可以精确到0.001，即精确到毫秒级别。

`callback`

回调函数 `注意：如果回调函数是类的方法，则方法必须是public属性`

`args`

回调函数的参数，必须为数组，数组元素为参数值

`persistent`

是否是持久的，如果只想定时执行一次，则传递false（只执行一次的任务在执行完毕后会自动销毁，不必调用 `Timer::del()` ）。默认是true，即一直定时执行。

返回值

返回一个整数，代表计时器的timerid，可以通过调用 `Timer::del(\$timerid)` 销毁这个计时器。

示例

1、定时函数为匿名函数（闭包）

```
use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';
```

```
$task = new Worker();
// 开启多少个进程运行定时任务，注意业务是否有多进程有并发问题
```

本文档使用 [看云](#) 构建

```

$task->count = 1;
$task->onWorkerStart = function($task)
{
    // 每2.5秒执行一次
    $time_interval = 2.5;
    Timer::add($time_interval, function()
    {
        echo "task run\n";
    });
};

// 运行worker
Worker::runAll();

```

2、只在指定进程中设置定时器

一个worker实例有4个进程，只在id编号为0的进程上设置定时器。

```

use Workerman\Worker;
use Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker();
$worker->count = 4;
$worker->onWorkerStart = function($worker)
{
    // 只在id编号为0的进程上设置定时器，其它1、2、3号进程不设置定时器
    if($worker->id === 0)
    {
        Timer::add(1, function(){
            echo "4个worker进程，只在0号进程设置定时器\n";
        });
    }
};
// 运行worker
Worker::runAll();

```

3、定时函数为匿名函数，利用闭包传递参数

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$ws_worker = new Worker('websocket://0.0.0.0:8080');
$ws_worker->count = 8;
// 连接建立时给对应连接设置定时器
$ws_worker->onConnect = function($connection)
{
    // 每10秒执行一次
    $time_interval = 10;
    $connect_time = time();
    // 给connection对象临时添加一个timer_id属性保存定时器id
    $connection->timer_id = Timer::add($time_interval, function()use($con

```



```

nection, $connect_time)
    {
        $connection->send($connect_time);
    });
};
// 连接关闭时, 删除对应连接的定时器
$ws_worker->onClose = function($connection)
{
    // 删除定时器
    Timer::del($connection->timer_id);
};

// 运行worker
Worker::runAll();

```

4、定时器函数为匿名函数，利用定时器接口传递参数

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$ws_worker = new Worker('websocket://0.0.0.0:8080');
$ws_worker->count = 8;
// 连接建立时给对应连接设置定时器
$ws_worker->onConnect = function($connection)
{
    // 每10秒执行一次
    $time_interval = 10;
    $connect_time = time();
    // 给connection对象临时添加一个timer_id属性保存定时器id
    $connection->timer_id = Timer::add($time_interval, function($connecti
on, $connect_time)
    {
        $connection->send($connect_time);
    }, array($connection, $connect_time));
};
// 连接关闭时, 删除对应连接的定时器
$ws_worker->onClose = function($connection)
{
    // 删除定时器
    Timer::del($connection->timer_id);
};

// 运行worker
Worker::runAll();

```

5、定时函数为普通函数

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 普通的函数

```

```

function send_mail($to, $content)
{
    echo "send mail ...\n";
}

$task = new Worker();
$task->onWorkerStart = function($task)
{
    $to = 'workerman@workerman.net';
    $content = 'hello workerman';
    // 10秒后执行发送邮件任务, 最后一个参数传递false, 表示只运行一次
    Timer::add(10, 'send_mail', array($to, $content), false);
};

// 运行worker
Worker::runAll();

```

6、定时函数为类的方法

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

class Mail
{
    // 注意, 回调函数属性必须是public
    public function send($to, $content)
    {
        echo "send mail ...\n";
    }
}

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 10秒后发送一次邮件
    $mail = new Mail();
    $to = 'workerman@workerman.net';
    $content = 'hello workerman';
    Timer::add(10, array($mail, 'send'), array($to, $content), false);
};

// 运行worker
Worker::runAll();

```

7、定时函数为类方法（类内部使用定时器）

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

class Mail
{

```

```

        // 注意, 回调函数属性必须是public
        public function send($to, $content)
        {
            echo "send mail ...\n";
        }

        public function sendLater($to, $content)
        {
            // 回调的方法属于当前的类, 则回调数组第一个元素为$this
            Timer::add(10, array($this, 'send'), array($to, $content), false)
        }
    }

    $task = new Worker();
    $task->onWorkerStart = function($task)
    {
        // 10秒后发送一次邮件
        $mail = new Mail();
        $to = 'workerman@workerman.net';
        $content = 'hello workerman';
        $mail->sendLater($to, $content);
    };

    // 运行worker
    Worker::runAll();

```

8、定时函数为类的静态方法

```

use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

class Mail
{
    // 注意这个是静态方法, 回调函数属性也必须是public
    public static function send($to, $content)
    {
        echo "send mail ...\n";
    }
}

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 10秒后发送一次邮件
    $to = 'workerman@workerman.net';
    $content = 'hello workerman';
    // 定时调用类的静态方法
    Timer::add(10, array('Mail', 'send'), array($to, $content), false);
};

// 运行worker
Worker::runAll();

```

9、定时函数为类的静态方法(带命名空间)

```
namespace Task;
use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

class Mail
{
    // 注意这个是静态方法, 回调函数属性也必须是public
    public static function send($to, $content)
    {
        echo "send mail ...\n";
    }
}

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 10秒后发送一次邮件
    $to = 'workerman@workerman.net';
    $content = 'hello workerman';
    // 定时调用带命名空间的类的静态方法
    Timer::add(10, array('\Task\Mail', 'send'), array($to, $content), false);
};

// 运行worker
Worker::runAll();
```

10、定时器中销毁当前定时器 (use闭包方式传递\$timer_id)

```
use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 计数
    $count = 1;
    // 要想$timer_id能正确传递到回调函数内部, $timer_id前面必须加地址符 &
    $timer_id = Timer::add(1, function()use(&$timer_id, &$count)
    {
        echo "Timer run $count\n";
        // 运行10次后销毁当前定时器
        if($count++ >= 10)
        {
            echo "Timer::del($timer_id)\n";
            Timer::del($timer_id);
        }
    });
};
```

```
// 运行worker
Worker::runAll();
```

11、定时器中销毁当前定时器（参数方式传递\$timer_id）

```
use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

class Mail
{
    public function send($to, $content, $timer_id)
    {
        // 临时给当前对象添加一个count属性，记录定时器运行次数
        $this->count = empty($this->count) ? 1 : $this->count;
        // 运行10次后销毁当前定时器
        echo "send mail {$this->count}...\n";
        if($this->count++ >= 10)
        {
            echo "Timer::del($timer_id)\n";
            Timer::del($timer_id);
        }
    }
}

$task = new Worker();
$task->onWorkerStart = function($task)
{
    $mail = new Mail();
    // 要想$timer_id能正确传递到回调函数内部，$timer_id前面必须加地址符 &
    $timer_id = Timer::add(1, array($mail, 'send'), array('to', 'content'
, &$timer_id));
};

// 运行worker
Worker::runAll();
```

del

del

```
boolean \Workerman\Lib\Timer::del(int $timer_id)
```

删除某个定时器

参数

`timer_id`

定时器的id，即add接口返回的整型

返回值

boolean

示例

```
use \Workerman\Worker;
use \Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$task = new Worker();
// 开启多少个进程运行定时任务，注意多进程并发问题
$task->count = 1;
$task->onWorkerStart = function($task)
{
    // 每2秒运行一次
    $timer_id = Timer::add(2, function()
    {
        echo "task run\n";
    });
    // 20秒后运行一个一次性任务，删除2秒一次的定时任务
    Timer::add(20, function($timer_id)
    {
        Timer::del($timer_id);
    }, array($timer_id), false);
};

// 运行worker
Worker::runAll();
```

实例(定时器回调中删除当前定时器)

```
use \Workerman\Worker;
use \Workerman\Lib\Timer;
```

del

```
require_once __DIR__ . '/Workerman/Autoloader.php';

$task = new Worker();
$task->onWorkerStart = function($task)
{
    // 注意, 回调里面使用当前定时器id必须使用引用(&)的方式引入
    $timer_id = Timer::add(1, function()use(&$timer_id)
    {
        static $i = 0;
        echo $i++."\n";
        // 运行10次后删除定时器
        if($i === 10)
        {
            Timer::del($timer_id);
        }
    });
};

// 运行worker
Worker::runAll();
```

定时器注意事项

注意事项

定时器使用注意事项

- 1、只能在 `onXXXX` 回调中添加定时器。全局的定时器推荐在 `onWorkerStart` 回调中设置，针对某个连接的定时器推荐在 `onConnect` 中设置。
- 2、添加的定时任务在当前进程执行(不会启动新的进程或者线程)，如果任务很重（特别是涉及到网络IO的任务），可能会导致该进程阻塞，暂时无法处理其它业务。所以最好将耗时的任务放到单独的进程运行，例如建立一个/多个Worker进程运行
- 3、当前进程忙于其它业务时或者当一个任务没有在预期的时间运行完，这时又到了下一个运行周期，则会等待当前任务完成才会运行，这会导致定时器没有按照预期时间间隔运行。也就是说当前进程的业务都是串行执行的，如果是多进程则进程间的任务运行是并行的。
- 4、需要注意多进程设置了定时任务造可能会造成并发问题，例如下面的代码每秒会打印5次。

```
$worker = new Worker();  
// 5个进程  
$worker->count = 5;  
$worker->onWorkerStart = function($worker) {  
    // 5个进程，每个进程都有一个这样的定时器  
    Timer::add(1, function(){  
        echo "hi\r\n";  
    });  
};  
Worker::runAll();
```

如果只想要一个进程运行定时器，参考[Timer::add 示例2](#)

- 5、可能会有1毫秒左右的误差。

WebServer

WebServer

WorkerMan自带了一个简单的Web服务器，同样也是基于Worker实现的。文件位置在Workerman/WebServer.php。这个WebServer开发的目的是为了更方便运行一些简单的Web程序，例如workerman-todpole等web界面程序。

注意：WebServer只能用http协议。

使用方法

在Applications/YourApp/start.php中添加

```
use \Workerman\Worker;
use \Workerman\WebServer;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 这里监听8080端口，如果要监听80端口，需要root权限，并且端口没有被其它程序占用
$webserver = new WebServer('http://0.0.0.0:8080');
// 类似nginx配置中的root选项，添加域名与网站根目录的关联，可设置多个域名多个目录
$webserver->addRoot('www.example.com', '/your/path/of/web/');
$webserver->addRoot('blog.example.com', '/your/path/of/blog/');
// 设置开启多少进程
$webserver->count = 4;

Worker::runAll();
```

WorkerMan的Webserver与普通Web开发异同

1、普通Web程序架构运行机制

一般的Web程序一般都是基于nginx+php-fpm或者apache+php的架构开发的，这些架构的运行机制一般是每个请求都会经过请求初始化、创建执行环境、词法解析、语法解析、编译生成opcode以及请求关闭释放各种资源（如果有opcode缓存会跳过词法解析、语法解析、编译生成opcode步骤）

2、WorkerMan架构Web程序运行机制

WorkerMan是常驻内存的运行机制，只要PHP文件被载入编译过一次，便会常驻内存，不会再去从磁盘读取或者再去编译，并省去了重复的请求初始化、创建执行环境、词法解析、语法解析、编译生成opcode以及请求关闭释放各种资源等诸多耗时的步骤。剩下的只是简单的计算过程，所以性能很高。正因为常驻内存，所以类、函数、常量等定义代码只要运行一次，便可以永久使用，不会被销毁，所以要避免反复加载类、函数、常量等定义文件。比较

本文档使用 [看云](#) 构建

简单的办法是使用require_once加载文件，避免重复加载重复定义。

3、避免使用exit、die语句

同样的，在程序中避免使用exit、die语句，使用exit、die会导致进程退出。可以使用`WorkerMan\Protocols\Http::end(\$msg)`函数替代exit、die函数。

4、HTTP相关函数的使用

WorkerMan运行在PHP CLI模式下，PHP CLI模式下无法使用HTTP相关的函数，例如`header`、`setcookie`、`session_start`等函数，请使用`/WorkerMan/Protocols/Http.php`文件中的`header`、`setcookie`、`sessionStart`等方法替换，调用方式类似`WorkerMan\Protocols\Http::header()`。

5、Web入口文件

WorkerMan的WebServer默认使用index.php作为Web入口文件，例如配置`\$webserver->setRoot('www.example.com', '/home/www/');`，则www.example.com的入口文件为`/home/www/index.php`。当url访问的文件（包括静态文件和PHP文件）不存在时，会自动调用入口文件index.php

6、可用的超全局变量

可用的超全局变量有`\$_SERVER`、`\$_GET`、`\$_POST`、`\$_FILES`、`\$_COOKIE`、`\$_SESSION`、`\$_REQUEST`。

无法使用`php://input`，请用`\$GLOBALS['HTTP_RAW_POST_DATA']`代替。

注意HTTP文件上传中，WorkerMan的`\$_FILES`结构与传统PHP中的`\$_FILES`结构不同，WorkerMan中`\$_FILES`结构类似

```
var_export($_FILES);
array(
    0 => array(
        'file_name' => 'logo.png', // 文件名称
        'file_size' => 23654,      // 文件大小
        'file_data' => '*****',   // 文件的二进制数据
    ),
    1 => array(
        'file_name' => 'file.tar.gz', // 文件名称
        'file_size' => 128966,        // 文件大小
        'file_data' => '*****',     // 文件的二进制数据
    ),
    ...
);
```

保存文件代码类似

```
// 例如保存到/tmp目录下
foreach($_FILES as $file_info)
{
    file_put_contents('/tmp/'.$file_info['file_name'], $file_info['file_data']);
}
```

WorkerMan中无法使用 `move_uploaded_file() is_uploaded_file()` 这些函数。

7、可以设置onWorkerStart、onWorkerStop回调

可以设置onWorkerStart、onWorkerStop回调，做进程启动时全局初始化及进程退出（stop等命令）数据保存清理工作

调试

基本调试

`status`命令查看运行状态

网络抓包

跟踪系统调用

基本调试

基本调试

WorkerMan3.0有两种运行模式，调试模式以及daemon运行模式

运行 ``php start.php start`` 进入调试模式，这时代码中的 ``echo``、``var_dump``、``var_export`` 等函数打印会在终端显示。注意以 ``php start.php start`` 运行的WorkerMan在终端关闭时所有进程会退出。

而运行 ``php start.php start -d`` 则是进入daemon模式，也就是正式上线的运行模式，关闭终端不受影响。

如果想daemon方式运行时也能看到 ``echo``、``var_dump``、``var_export`` 等函数打印，可以设置Worker::\$stdoutFile属性，例如

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 将屏幕打印输出到Worker::$stdoutFile指定的文件中
Worker::$stdoutFile = '/tmp/stdout.log';

$http_worker = new Worker("http://0.0.0.0:2345");
$http_worker->onMessage = function($connection, $data)
{
    $connection->send('hello world');
};

Worker::runAll();
```

这样所有的 ``echo``、``var_dump``、``var_export`` 等函数打印会写入到 ``Worker::$stdoutFile`` 指定的文件中。注意 ``Worker::$stdoutFile`` 指定的路径要有可写权限。

status命令查看运行状态

查看运行状态

运行 `php start.php status`

可以查看到WorkerMan的运行状态，类似如下：

```
-----GLOBAL STATUS-----
Workerman version:3.5.0      PHP version:5.3.29-1~dotdeb.0
start time:2015-02-21 18:05:47  run 86 days 22 hours
load average: 0, 0, 0
3 workers      10 processes
worker_name      exit_status      exit_count
TodpoleGateway      0                0
TodpoleBusinessWorker 0                4
TodpoleBusinessWorker 9                1
WebServer          0                2
-----PROCESS STATUS-----
pid      memory  listening      worker_name      connection
s total_request send_fail timers
936      2.15M  Websocket://0.0.0.0:8585 TodpoleGateway    13
355      0      0
937      2.03M  Websocket://0.0.0.0:8585 TodpoleGateway    5
181      0      0
938      2M     Websocket://0.0.0.0:8585 TodpoleGateway    4
171      0      0
939      2.03M  Websocket://0.0.0.0:8585 TodpoleGateway    5
177      0      0
948      2.15M  none            TodpoleBusinessWorker 4
32      0      0
949      2.16M  none            TodpoleBusinessWorker 4
54      0      0
953      2.16M  none            TodpoleBusinessWorker 4
50      0      0
957      2.15M  none            TodpoleBusinessWorker 4
53      0      0
954      1.84M  http://0.0.0.0:8686 WebServer          0
61      0      0
955      1.84M  http://0.0.0.0:8686 WebServer          1
59      0      0
```

说明

GLOBAL STATUS

从这以栏中我们可以看到

WorkerMan的版本 `version:3.5.0`

启动时间 `2015-02-21 18:05:47`，运行了 `run 86 days 22 hours`

服务器负载 `load average: 0, 0, 0`

`3 workers` (3种进程，包括ChatGateway、ChatBusinessWorker、WebServer进程)

10 processes (共10个进程)

worker_name (worker进程名)

exit_status (worker进程退出状态码)

exit_count (该状态码的退出次数)

一般来说exit_status为0表示为正常退出，如果为其它值，代表进程是异常退出的，并产生一条类似 `WORKER EXIT UNEXPECTED` 错误信息，错误信息会记录到[Worker::logFile](#)指定的文件中。

常见的exit_status及其含义如下：

- 0：表示正常退出，运行reload平滑重启后会出现值为0的退出码，是正常现象。注意在程序中调用exit或die也会导致退出码为0，并产生一条 `WORKER EXIT UNEXPECTED` 错误信息，workerman中不允许业务代码调用exit或者die语句。
- 9：表示进程被SIGKILL信号杀死了。这个退出码主要发生在stop以及reload平滑重启时，导致这个退出码的原因是由于子进程没有在规定时间内响应主进程reload信号(例如mysql、curl等长时间阻塞等待或者业务死循环等)，被主进程强制使用SIGKILL信号杀死。注意，当在linux命令行中使用kill命令发送SIGKILL信号给子进程也会导致这个退出码。
- 65280：导致这个退出码的原因是业务代码有致命错误，例如调用了不存在的函数、语法错误等，具体错误信息会记录到[Worker::logFile](#)指定的文件中，也可以在[php.ini](#)中[error_log](#)指定的文件中(如果有指定的话)找到。
- 64000：导致这个退出码的原因是业务代码抛出了异常，但业务没有捕获这个异常，导致进程退出。如果workerman以debug方式运行时异常调用栈会打印到终端，daemon方式运行时异常调用栈会记录到[Worker::stdoutFile](#)指定的文件中。

PROCESS STATUS

pid：进程pid

memory：该进程当前占用内存（不包括php自身可执行文件的占用的内存）

listening：传输层协议及监听ip端口。如果不监听任何端口则显示none。参见[Worker类构造函数](#)

worker_name：该进程运行的服务服务名，见[Worker类name属性](#)

connections:该进程当前有多少个TCP连接对象实例。注意：每个客户端连接是一个[TcpConnection](#)连接对象实例，同时每个[AsyncTcpConnection](#)连接也是一个连接对象实例，所以connections的计数不一定与客户端连接数相等，例如GatewayWorker中Gateway进程的connections计数包含了客户端连接数和Gateway与Worker内部通讯连接数。

total_request：表示该进程从启动到现在一共接收了多少个请求。这里的请求数不仅包含客户端发来的请求，也包含Workerman内部通讯请求，例如GatewayWorker架构里Gateway与BusinessWorker之间的通讯请求。

send_fail：该进程向客户端发送数据失败次数，失败原因一般为客户端连接断开，此项不为0一般属于正常状态，参见[手册常见问题send_fail原因](#)

timers：该进程活动的定时器数量（不包括被删除的定时器以及已经运行过的一次性定时器）。注意：这个统计需要workerman版本>=3.4.7才支持

status: 进程状态，如果是idle代表空闲，如果是busy代表是繁忙(或者阻塞在网络IO上)。注意：这个统计需要workerman版本>=3.5.0才支持

原理

status脚本运行后，主进程会向所有worker进程发送一个 `\ SIGUSR2 \` 信号，随后status脚本进入100毫秒的睡眠阶段，以便等待各个worker进程状态统计结果。这时空闲的worker进程收到 `\ SIGUSR2 \` 信号后会立刻向特定的磁盘文件写入自己的运行状态（连接数、请求数等等），而正在处理业务逻辑的worker进程，则会等待业务逻辑处理完毕才会去写入自己的状态信息。sleep100毫秒后，status脚本开始读取磁盘中的状态文件，并展示结果到控制台。

注意

status 时可能会发现显示的进程数量少于实际数量，原因是由于进程忙于处理业务（例如业务逻辑长时间阻塞在curl或者数据库请求上），导致worker进程无法及时（100毫秒内）响应status命令，来不及将自己的状态信息写入磁盘，status脚本自然统计不到对应worker的

进程，也就无法及时展示在status结果中。

出现这种问题需要排查业务代码，看哪里导业务致长时间阻塞，并且评估阻塞耗时是否在预期内。

网络抓包

网络抓包

下面的例子中我们通过 `tcpdump` 查看 `workerman-chat` 应用通过 `websocket` 传输的数据。`workerman-chat` 例子中服务端是通过 `7272` 端口对外提供 `websocket` 服务的，所以我们抓取 `7272` 端口上的数据包。

- 1、运行命令 `tcpdump -Ans 4096 -iany port 7272`
- 2、在浏览器地址栏输入 `http://127.0.0.1:55151`
- 3、输入昵称 `mynick`
- 4、发表框输入 `hi, all !`

最终抓取的数据如下：

```
/*
 * TCP第一次握手
 * 浏览器本地端口60653向远程端口7272发送SYN包
 */
17:50:00.523910 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [S], seq 35242
90970, win 32768, options [mss 16396,sackOK,TS val 28679666 ecr 28679554,
nop,wscale 7], length 0
E..<.h@.@.HQ.....h..i.....0....@....
.....

/*
 * TCP第二次握手
 * 远程端口7272向浏览器端口60653回应SYN+ACK包
 */
17:50:00.523935 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [S.], seq 6926
96454, ack 3524290971, win 32768, options [mss 16396,sackOK,TS val 286796
66 ecr 28679666,nop,wscale 7], length 0
E..<..@.@.<.....h..)I....i.....0....@....
.....

/*
 * TCP第三次握手，完成TCP连接
 * 浏览器本地端口60653向远程端口7272发送ACK包
 */
17:50:00.523948 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [.], ack 1, wi
n 256, options [nop,nop,TS val 28679666 ecr 28679666], length 0
E..4.i@.@.HX.....h..i..)I.....(.....
.....
```

```

/*
 * websocket握手
 * 浏览器本地端口60653向远程端口7272发送websocket握手请求数据
 */
17:50:00.524412 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [P.], seq 1:716, ack 1, win 256, options [nop,nop,TS val 28679666 ecr 28679666], length 715
E....j@.E.....h..i.)I.....
.....GET / HTTP/1.1
Host: 127.0.0.1:7272
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:31.0) Gecko/20100101 Firefox/31.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://127.0.0.1:55151
Sec-WebSocket-Key: zPDr6m4czzUd0FnsxIUEAw==
Cookie: Hm_lvt_abcf9330bef79b4aba5b24fa373506d9=1402048017; Hm_lvt_5fedb3bdce89499492c079ab4a8a0323=1403063068,1403141761; Hm_lvt_7b1919221e89d2aa5711e4deb935debd=1407836536; Hm_lvt_7b1919221e89d2aa5711e4deb935debd=1407837000
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

/*
 * websocket握手
 * 远程端口7272向浏览器端口60653发送ACK包, 表明远程7272端口已经收到websocket握手请求数据
 */
17:50:00.524423 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [.], ack 716, win 256, options [nop,nop,TS val 28679666 ecr 28679666], length 0
E..4(u@.M.....h..)I....lf.....
.....

/*
 * websocket握手
 * 远程端口7272向浏览器端口60653发送websocket握手回应, 表明握手成功
 */
17:50:00.535918 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [P.], seq 1:157, ack 716, win 256, options [nop,nop,TS val 28679669 ecr 28679666], length 156
E...(v@.E.....h..)I....lf.....
.....HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Sec-WebSocket-Version: 13
Connection: Upgrade
Sec-WebSocket-Accept: nSsCeIBUsFnDJCRb/BNlFzBUDpM=

/*
 * websocket握手成功
 * 浏览器本地端口60653向远程端口7272发送ACK, 表明接收到websocket握手回应数据
 */
17:50:00.535932 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [.], ack 157,

```

```

win 256, options [nop,nop,TS val 28679669 ecr 28679669], length 0
E..4.k@.H.V.....h..lf)I.#.....(.....
.....

/*
 * 输入昵称请求
 * 浏览器通过websocket协议向7272端口发送 昵称 请求 {"type":"login","name":"myn
  ick"}
 * 由于浏览器向服务端发送的数据为websocket协议掩码处理过的数据，所以无法看到原文 {"t
  ype":"login","name":"mynick"}
 */
17:50:30.652680 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [P.], seq 716:
754, ack 157, win 256, options [nop,nop,TS val 28687198 ecr 28679669], le
ngth 38
E..Z.l@.H/.....h..lf)I.#.....N.....
...^.....&_...+...C}..J0..H}..H>...e..._1..M}.

/*
 * 输入昵称请求
 * 7272端口向浏览器返回ACK，表明昵称请求已经接收，并返回用户列表{"type":"user_list
  " ...
 */
17:50:30.653546 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [P.], seq 157:
267, ack 754, win 256, options [nop,nop,TS val 28687198 ecr 28687198], le
ngth 110
E...(w@.H.....h..)I.#..l.....
...^...^..l{"type":"user_list","user_list":[{"uid":783654164,"name":"\u732
a\u732a"}, {"uid":783700053,"name":"mynick"}]}

/*
 * 输入昵称请求
 * 浏览器返回ACK，表明用户列表数据已经收到
 */
17:50:30.653559 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [.], ack 267,
win 256, options [nop,nop,TS val 28687198 ecr 28687198], length 0
E..4.m@.H.T.....h..l.)I.....(.....
...^...^

/*
 * 输入昵称请求
 * 7272端口向浏览器返回ACK，并返回用登录结果{"type":"login",...
 */
17:50:30.653689 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [P.], seq 267:
346, ack 754, win 256, options [nop,nop,TS val 28687198 ecr 28687198], le
ngth 79
E...(x@.H.....h..)I....l.....w.....
...^...^..M{"type":"login","uid":783700053,"name":"mynick","time":"2014-08
-12 17:50:30"}

/*
 * 输入昵称请求 完毕
 * 浏览器返回ACK，表明登录结果数据包收到
 */
17:50:30.653695 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [.], ack 346,
win 256, options [nop,nop,TS val 28687198 ecr 28687198], length 0
E..4.n@.H.S.....h..l.)I.....(.....

```

```

...^...^

/*
 * 服务端7272端口通知其它浏览器有新用户登录
 */
17:50:30.653749 IP 127.0.0.1.7272 > 127.0.0.1.60584: Flags [P.], seq 436:
515, ack 816, win 256, options [nop,nop,TS val 28687198 ecr 28577913], le
ngth 79
E....@.3.....h..f....G.....w.....
...^...y.M{"type":"login","uid":783700053,"name":"mynick","time":"2014-08
-12 17:50:30"}

/*
 * 其它浏览器返回 ACK, 表明收到新用户登录通知的请求
 */
17:50:30.653755 IP 127.0.0.1.60584 > 127.0.0.1.7272: Flags [.], ack 515,
win 256, options [nop,nop,TS val 28687198 ecr 28687198], length 0
E..4.X@.0.#j.....h.G..f..$......(.....
...^...^

/*
 * mynick用户发言 hi, all !
 * 浏览器向服务端7272端口发送发言数据 {"type":"say","to_uid":"all","content":"
hi, all !"}
 * 由于浏览器向服务端发送的数据为websocket协议掩码处理过的数据, 所以无法看到原文
 */
17:51:02.775205 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [P.], seq 754:
812, ack 346, win 256, options [nop,nop,TS val 28695228 ecr 28687198], le
ngth 58
E..n.o@.0.H.....h..l.)I.....b.....
fTX.d.P[(...9H..C=LT.~.BV=...0SnB-X.

/*
 * mynick用户发言 hi, all !
 * 7272端口向所有浏览器客户端中一个浏览器转发发言数据 {"type":"say","from_uid":..
..
 */
17:51:02.776785 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [P.], seq 346:
448, ack 812, win 256, options [nop,nop,TS val 28695229 ecr 28695228], le
ngth 102
E...(y@.0.....h..)I....l.....
.....d{"type":"say","from_uid":783700053,"to_uid":"all","content":"hi
, all !","time":"2014-08-12 :51:02"}

/*
 * mynick用户发言 hi, all !
 * 浏览器响应ACK, 收到发言数据
 */
17:51:02.776808 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [.], ack 448,
win 256, options [nop,nop,TS val 28695229 ecr 28695229], length 0
E..4.p@.0.HQ.....h..l.)I.F.....(.....
.....

/*
 * mynick用户发言 hi, all !
 * 7272端口向所有浏览器客户端中一个浏览器转发发言数据 {"type":"say","from_uid":..

```

```

..
*/
17:51:02.776827 IP 127.0.0.1.7272 > 127.0.0.1.60584: Flags [P.], seq 515:
617, ack 816, win 256, options [nop,nop,TS val 28695229 ecr 28687198], le
ngth 102
E.....@.@.3g.....h..f..$.G.....
.....^..d{"type":"say","from_uid":783700053,"to_uid":"all","content":"hi
, all !","time":"2014-08-12 :51:02"}

/*
* mynick用户发言 hi, all ! , 所有浏览器都收到转发的发言数据, 发言完毕
* 浏览器响应ACK, 收到发言数据
*/
17:51:02.776842 IP 127.0.0.1.60584 > 127.0.0.1.7272: Flags [.], ack 617,
win 256, options [nop,nop,TS val 28695229 ecr 28695229], length 0
E..4.Y@.@.#i.....h.G..f.....(.....
.....

```

以上是登录+发言的所有所有请求，一共有两个浏览器客户端。

包数据中 `[S]` 代表 `SYN` 请求（发起连接请求）；`[.]` 代表 `ACK` 回应，说明请求对端已经收到；``P``代表发送数据；[P.]代表[P] + [.]

如果端口上传输的数据是二进制数据，则可以以十六进制来查看 `tcpdump -XAns 4096 -iany port 7272``

跟踪系统调用

跟踪系统调用

当想知道一个进程在做什么事情的时候，可以通过 `strace` 命令跟踪一个进程的所有系统调用。

1、运行 `php start.php status` 能看到workerman相关进程的信息 如下：

```
Hello admin
-----GLOBAL STATUS-----
WorkerMan version:3.0.1
start time:2014-08-12 17:42:04    run 0 days 1 hours
load average: 3.34, 3.59, 3.67
1 users          8 workers        14 processes
worker_name      exit_status    exit_count
BusinessWorker   0              0
ChatWeb          0              0
FileMonitor      0              0
Gateway          0              0
Monitor          0              0
StatisticProvider 0              0
StatisticWeb     0              0
StatisticWorker  0              0
-----PROCESS STATUS-----
pid      memory    listening      timestamp      worker_name    total_r
equest packet_err thunder_herd client_close send_fail throw_exception suc
/total
10352    1.5M     tcp://0.0.0.0:55151  1407836524 ChatWeb        12
0        0        2        0        0        0        10
0%
10354    1.25M   tcp://0.0.0.0:7272  1407836524 Gateway        3
0        0        0        0        0        0        10
0%
10355    1.25M   tcp://0.0.0.0:7272  1407836524 Gateway        0
0        1        0        0        0        0        10
0%
10365    1.25M   tcp://0.0.0.0:55757  1407836524 StatisticWeb   0
0        0        0        0        0        0        10
0%
10358    1.25M   tcp://0.0.0.0:7272  1407836524 Gateway        3
0        2        0        0        0        0        10
0%
10364    1.25M   tcp://0.0.0.0:55858  1407836524 StatisticProvider 0
0        0        0        0        0        0        10
0%
10356    1.25M   tcp://0.0.0.0:7272  1407836524 Gateway        3
0        2        0        0        0        0        10
```

```

0%
10366  1.25M  udp://0.0.0.0:55656  1407836524  StatisticWorker  55
      0      0      0      0      0      0      10
0%
10349  1.25M  tcp://127.0.0.1:7373  1407836524  BusinessWorker  5
      0      0      0      0      0      0      10
0%
10350  1.25M  tcp://127.0.0.1:7373  1407836524  BusinessWorker  0
      0      0      0      0      0      0      10
0%
10351  1.5M   tcp://127.0.0.1:7373  1407836524  BusinessWorker  5
      0      0      0      0      0      0      10
0%
10348  1.25M  tcp://127.0.0.1:7373  1407836524  BusinessWorker  2
      0      0      0      0      0      0      10
0%

```

2、例如我们想知道pid为10354的gateway进程在做什么，则可以运行命令 `strace -p 10354` (可能需要root权限) 类似如下：

```

sudo strace -p 10354
Process 10354 attached - interrupt to quit
clock_gettime(CLOCK_MONOTONIC, {118627, 242986712}) = 0
gettimeofday({1407840609, 102439}, NULL) = 0
epoll_wait(3, 985f4f0, 32, -1) = -1 EINTR (Interrupted system call)
--- SIGUSR2 (User defined signal 2) @ 0 (0) ---
send(7, "\f", 1, 0) = 1
sigreturn() = ? (mask now [])
clock_gettime(CLOCK_MONOTONIC, {118627, 699623319}) = 0
gettimeofday({1407840609, 559092}, NULL) = 0
epoll_wait(3, {{EPOLLIN, {u32=9, u64=9}}}, 32, -1) = 1
clock_gettime(CLOCK_MONOTONIC, {118627, 699810499}) = 0
gettimeofday({1407840609, 559277}, NULL) = 0
recv(9, "\f", 1024, 0) = 1
recv(9, 0xb60b4880, 1024, 0) = -1 EAGAIN (Resource temporarily unavailable)
epoll_wait(3, 985f4f0, 32, -1) = -1 EINTR (Interrupted system call)
--- SIGUSR2 (User defined signal 2) @ 0 (0) ---
send(7, "\f", 1, 0) = 1
sigreturn() = ? (mask now [])
clock_gettime(CLOCK_MONOTONIC, {118628, 699497204}) = 0
gettimeofday({1407840610, 558937}, NULL) = 0
epoll_wait(3, {{EPOLLIN, {u32=9, u64=9}}}, 32, -1) = 1
clock_gettime(CLOCK_MONOTONIC, {118628, 699588603}) = 0
gettimeofday({1407840610, 559023}, NULL) = 0
recv(9, "\f", 1024, 0) = 1
recv(9, 0xb60b4880, 1024, 0) = -1 EAGAIN (Resource temporarily unavailable)
epoll_wait(3, 985f4f0, 32, -1) = -1 EINTR (Interrupted system call)
--- SIGUSR2 (User defined signal 2) @ 0 (0) ---

```



```
send(7, "\f", 1, 0)           = 1  
sigreturn()                   = ? (mask now [])
```

3、其中每一行是一个系统调用，从这个信息中我们很容易看到进程在做一些什么事情，可以定位到进程卡在哪里，卡在连接还是读取网络数据等

常用组件

GlobalData数据共享组件

Channel分布式通讯组件

FileMonitor文件监控组件

MySQL组件

redis组件

异步http组件

异步消息队列组件

异步dns组件

进程控制组件

memcache

GlobalData数据共享组件

GlobalData变量共享组件

(要求Workerman版本 $\geq 3.3.0$)

源码地址：<https://github.com/walkor/GlobalData>

注意

GlobalData需要Workerman版本 $\geq 3.3.0$

下载安装

可以使用composer安装，或者直接下载zip

包<https://github.com/walkor/GlobalData/archive/master.zip>。

原理

利用PHP的 `set get isset unset` 魔术方法触发与GlobalData服务端通讯，实际变量存储在GlobalData服务端。例如当给客户端类设置一个不存在的属性时，会触发 `set` 魔术方法，客户端类在 `set` 方法中向GlobalData服务端发送请求，存入一个变量。当访问客户端类一个不存在的变量时，会触发类的 `__get` 方法，客户端会向GlobalData服务端发起请求，读取这个值，从而完成进程间变量共享。

```
require_once __DIR__ . '/../src/Client.php';

// 连接Global Data服务端
$global = new GlobalData\Client('127.0.0.1', 2207);

// 触发$global->__isset('somedata')查询服务端是否存储了key为somedata的值
isset($global->somedata);

// 触发$global->__set('somedata', array(1,2,3)), 通知服务端存储somedata对应的值为array(1,2,3)
$global->somedata = array(1,2,3);

// 触发$global->__get('somedata'), 从服务端查询somedata对应的值
var_export($global->somedata);

// 触发$global->__unset('somedata'), 通知服务端删掉somedata及对应的值
unset($global->somedata);
```


GlobalDataServer

GlobalData 组件服务端

(要求Workerman版本 $\geq 3.3.0$)

__construct

```
void \GlobalData\Server::__construct([string $listen_ip = '0.0.0.0', int $listen_port = 2207])
```

实例化一个\GlobalData\Server服务

参数

`listen_ip`

监听的本机ip地址，不传默认是 `0.0.0.0`

`listen_port`

监听的端口，不传默认是2207

例子

```
use Workerman\Worker;
require_once __DIR__ . '/../../Workerman/Autoloader.php';
require_once __DIR__ . '/../src/Server.php';

// 监听端口
$worker = new GlobalData\Server('127.0.0.1', 2207);

Worker::runAll();
```

GlobalDataClient

GlobalData 组件客户端

(要求Workerman版本 $\geq 3.3.0$)

__construct

```
void \GlobalData\Client::__construct(mixed $server_address)
```

实例化一个\GlobalData\Client客户端对象。通过在客户端对象上赋值属性来进程间共享数据。

参数

GlobalData server 服务端地址，格式 ``<ip地址>:<端口>``，例如 ``127.0.0.1:2207``。

如果是GlobalData server集群，则传入一个地址数组，例如 ``array('10.0.0.10:2207', '10.0.0.11:2207')``。

说明

支持赋值、读取、isset、unset操作。

同时支持cas原子操作。

例子

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/GlobalData/src/Client.php';

// GlobalData Server
$global_worker = new GlobalData\Server('0.0.0.0', 2207);

$worker = new Worker('tcp://0.0.0.0:6636');
// 进程启动时
$worker->onWorkerStart = function()
{
    // 初始化一个全局的global data client
    global $global;
    $global = new \GlobalData\Client('127.0.0.1:2207');
};
// 每次服务端收到消息时
$worker->onMessage = function($connection, $data)
```

```
{
    // 更改$global->somedata的值, 其它进程会共享这个$global->somedata变量
    global $global;
    echo "now global->somedata=".var_export($global->somedata, true)."\n"
;
    echo "set \$global->somedata=$data";
    $global->somedata = $data;
};
Worker::runAll();
```

全部用法 (php-fpm环境也可以使用)

```
require_once __DIR__ . '/GlobalData/src/Client.php';

$global = new Client('127.0.0.1:2207');

var_export(isset($global->abc));

$global->abc = array(1,2,3);

var_export($global->abc);

unset($global->abc);

var_export($global->add('abc', 10));

var_export($global->increment('abc', 2));

var_export($global->cas('abc', 12, 18));
```

注意：

GlobalData组件无法共享资源类型的数据，例如mysql连接、socket连接等无法共享。

如果在Workerman环境中使用GlobalData/Client，请在onXXX回调中实例化GlobalData/Client对象，例如在onWorkerStart中实例化。

不能这样操作共享变量。

```
$global->somekey = array();
$global->somekey[]='xxx';

$global->someObject = new someClass();
$global->someObject->someVar = 'xxx';
```

可以这样

```
$somekey = array();  
$somekey[] = 'xxx';  
$global->somekey = $somekey;  
  
$someObject = new someClass();  
$someObject->someVar = 'xxx';  
$global->someObject = $someObject;
```


add

add

(要求**Workerman**版本**>=3.3.0**)

```
bool \GlobalData\Client::add(string $key, mixed $value)
```

原子添加。如果key已经存在，会返回false。

参数

`$key`

键值。（例如 ``$global->abc``，``abc`` 就是键值）

`$value`

存储的值。

返回值

成功返回true，否则返回false。

范例

```
require_once __DIR__ . '/../src/Client.php';

$global = new GlobalData\Client('127.0.0.1:2207');

if($global->add('some_key', 10))
{
    // $global->some_key赋值成功
    echo "add success " , $global->some_key;
}
else
{
    // $global->some_key已经存在，赋值失败
    echo "add fail " , var_export($global->some_key);
}
```


cas

cas

(要求Workerman版本>=3.3.0)

```
bool \GlobalData\Client::cas(string $key, mixed $old_value, mixed $new_value)
```

原子替换，用 ` \$new_value ` 替换 ` \$old_value `。

仅在当前客户端最后一次取值后，该key对应的值没有被其他客户端修改的情况下，才能够将值写入。

参数

\$key

键值。（例如 ` \$global->abc `，` abc ` 就是键值）

\$old_value

老数据

\$new_value

新数据

返回值

替换成功返回true，否则返回false。

说明：

多进程同时操作同一个共享变量时，有时候要考虑并发问题。

例如A B两个进程同时给用户列表添加一个成员。

A B进程当前用户列表都为 ` \$global->user_list = array(1,2,3) `。

A进程操作 ` \$global->user_list ` 变量，添加一个用户4。

B进程操作 ` \$global->user_list ` 变量，增加一个用户5。

A进程设置变量 ` \$global->user_list = array(1,2,3,4) ` 成功。

B进程设置变量 ` \$global->user_list = array(1,2,3,5) ` 成功。

此时B进程设置的变量将A进程设置的变量覆盖，导致数据丢失。

以上由于读取和设置不是一个原子操作，导致并发问题。

要解决这种并发问题，可以使用cas原子替换接口。

cas接口在改变一个值之前，

会根据 ` \$old_value ` 判断这个值是否被其它进程更改过，

如果有更改，则不替换，返回false。否则替换返回true。

见下面示例。

注意：

有些共享数据被并发覆盖是没问题的，例如竞拍系统某拍卖物当前最大报价，例如某商品当前库存等。

范例

```
require_once __DIR__ . '/../src/Client.php';

$global = new GlobalData\Client('127.0.0.1:2207');

// 初始化列表
$global->user_list = array(1,2,3);

// 向user_list原子添加一个值
do
{
    $old_value = $new_value = $global->user_list;
    $new_value[] = 4;
}
while(!$global->cas('user_list', $old_value, $new_value));

var_export($global->user_list);
```


increment

increment

(要求Workerman版本 $\geq 3.3.0$)

```
bool \GlobalData\Client::increment(string $key[, int $step = 1])
```

原子增加。将一个数值元素增加参数step指定的大小。如果元素的值不是数值类型，将其作为0再做增加处理。如果元素不存在返回false。

参数

\$key

键值。（例如 ``$global->abc`` , ``abc`` 就是键值）

\$value

要将元素的值增加的大小。

返回值

成功返回true，否则返回false。

范例

```
require_once __DIR__ . '/../src/Client.php';

$global = new GlobalData\Client('127.0.0.1:2207');

$global->some_key = 0;

// 非原子增加
$global->some_key++;

echo $global->some_key."\n";

// 原子增加
$global->increment('some_key');

echo $global->some_key."\n";
```


Channel分布式通讯组件

Channel分布式通讯组件

(要求Workerman版本 $\geq 3.3.0$)

源码地址：<https://github.com/walkor/Channel>

Channel是一个分布式通讯组件，用于完成进程间通讯或者服务器间通讯。

特点

- 1、基于订阅发布模型
- 2、非阻塞式IO

原理

Channel包含Channel/Server服务端和Channel/Client客户端

Channel/Client通过connect接口连接Channel/Server并保持长连接

Channel/Client通过调用on接口告诉Channel/Server自己关注哪些事件，并注册事件回调函数（回调发生在Channel/Client所在进程中）

Channel/Client通过publish接口向Channel/Server发布某个事件及事件相关的数据

Channel/Server接收事件及数据后会分发给关注这个事件的Channel/Client

Channel/Client收到事件及数据后触发on接口设置的回调

Channel/Client只会收到自己关注事件并触发回调

下载安装

可以使用composer安装，或者直接下载zip

包<https://github.com/walkor/Channel/archive/master.zip>。

文件目录可以根据需要放在任意位置，使用时能够require到文件src/Server.php 和 src/Client.php 两个文件即可。注意require路径最好使用绝对路径。具体使用方法参考下面章节channelServer和channelClient的说明。

注意

Channel只能用在workerman环境，php-fpm环境下无法使用。

ChannelServer

Channel组件服务端 (要求Workerman版本>=3.3.0)

__construct

```
void \Channel\Server::__construct([string $listen_ip = '0.0.0.0', int $listen_port = 2206])
```

实例化一个\Channel\Server服务端

参数

`listen_ip`

监听的本机ip地址，不传默认是 `0.0.0.0`

`listen_port`

监听的端口，不传默认是2206

例子

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Server.php';

// 不传参数默认是监听0.0.0.0:2206
$channel_server = new Channel\Server();

if(!defined('GLOBAL_START'))
{
    Worker::runAll();
}
```

channelClient

channelClient

(要求Workerman版本 $\geq 3.3.0$)

/Channel/Client 客户端

connect

connect

(要求Workerman版本 $\geq 3.3.0$)

```
void \Channel\Client::connect([string $listen_ip = '127.0.0.1', int $listen_port = 2206])
```

连接Channel/Server

参数

`listen_ip`

Channel/Server 监听的ip地址，不传默认是 `127.0.0.1`

`listen_port`

Channel/Server监听的端口，不传默认是2206

返回值

void

示例

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Client.php';

$http_worker = new Worker('http://0.0.0.0:4237');
$http_worker->onWorkerStart = function()
{
    Channel\Client::connect('127.0.0.1', 2206);
};

Worker::runAll();
```

on

on

(要求Workerman版本 $\geq 3.3.0$)

```
void \Channel\Client::on(string $event_name, callback $callback_function)
```

订阅 ` \$event_name ` 事件并注册事件发生时的回调 ` \$callback_function `

回调函数的参数

`$event_name`

订阅的事件名称，可以是任意的字符串。

`$callback_function`

事件发生时触发的回调函数。函数原型为 ` callback_function(mixed \$event_data) `。

` \$event_data ` 是事件发布(publish)时传递的事件数据。

注意：

如果同一个事件注册了两个回调函数，后一个回调函数将覆盖前一个回调函数。

范例

多进程Worker（多服务器），一个客户端发消息，广播给所有客户端

start.php

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Server.php';
require_once __DIR__ . '/Channel/src/Client.php';

// 初始化一个Channel服务端
$channel_server = new Channel\Server('0.0.0.0', 2206);

// websocket服务端
$worker = new Worker('websocket://0.0.0.0:4236');
$worker->name = 'websocket';
$worker->count = 6;
// 每个worker进程启动时
$worker->onWorkerStart = function($worker)
```

```

{
    // Channel客户端连接到Channel服务端
    Channel\Client::connect('127.0.0.1', 2206);
    // 订阅broadcast事件, 并注册事件回调
    Channel\Client::on('broadcast', function($event_data)use($worker){
        // 向当前worker进程的所有客户端广播消息
        foreach($worker->connections as $connection)
        {
            $connection->send($event_data);
        }
    });
};

$worker->onMessage = function($connection, $data)
{
    // 将客户端发来的数据当做事件数据
    $event_data = $data;
    // 向所有worker进程发布broadcast事件
    \Channel\Client::publish('broadcast', $event_data);
};

Worker::runAll();

```

测试

打开chrome浏览器, 按F12打开调试控制台, 在Console一栏输入(或者把下面代码放入到html页面用js运行)

接收消息的连接

```

// 127.0.0.1换成实际workerman所在ip
ws = new WebSocket("ws://127.0.0.1:4236");
ws.onmessage = function(e) {
    alert("收到服务端的消息：" + e.data);
};

```

广播消息

```
ws.send('hello world');
```

publish

publish

(要求Workerman版本>=3.3.0)

```
void \Channel\Client::publish(string $event_name, mixed $event_data)
```

发布某个事件，所有这个事件的订阅者会收到这个事件并触发 `on($event_name, $callback)` 注册的回调 `$callback`

参数

`$event_name`

发布的事件名称，可以是任意的字符串。如果事件没有任何订阅者，事件将被忽略。

`$event_data`

事件相关的数据，可以是数字、字符串或者数组

返回值

void

示例

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Client.php';

$http_worker = new Worker('http://0.0.0.0:4237');
$http_worker->onWorkerStart = function($http_worker)
{
    Channel\Client::connect('127.0.0.1', 2206);
};
$http_worker->onMessage = function($connection, $data)
{
    $event_name = 'user_login';
    $event_data = array('uid'=>123, 'uname'=>'tom');
    Channel\Client::publish($event_name, $event_data );
};

Worker::runAll();
```


unsubscribe

unsubscribe

(要求Workerman版本>=3.3.0)

```
void \Channel\Client::unsubscribe(string $event_name)
```

取消订阅某个事件，这个事件发生时将不会再触发 `on($event_name, $callback)` 注册的回调 `$callback`

参数

`$event_name`

事件名称

返回值

void

示例

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Client.php';

$http_worker = new Worker('http://0.0.0.0:4237');
$http_worker->onWorkerStart = function()
{
    Channel\Client::connect('127.0.0.1', 2206);
    $event_name = 'user_login';
    Channel\Client::on($event_name, function($event_data){
        var_dump($event_data);
    });
    Channel\Client::unsubscribe($event_name);
};

Worker::runAll();
```

例子-集群推送

例子1

(要求Workerman版本>=3.3.0)

基于Worker的多进程(分布式集群)推送系统，集群群发、集群广播。

start.php

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Server.php';
require_once __DIR__ . '/Channel/src/Client.php';

// 初始化一个Channel服务端
$channel_server = new Channel\Server('0.0.0.0', 2206);

// websocket服务端
$worker = new Worker('websocket://0.0.0.0:4236');
$worker->count=2;
$worker->name = 'pusher';
$worker->onWorkerStart = function($worker)
{
    // Channel客户端连接到Channel服务端
    Channel\Client::connect('127.0.0.1', 2206);
    // 以自己的进程id为事件名称
    $event_name = $worker->id;
    // 订阅worker->id事件并注册事件处理函数
    Channel\Client::on($event_name, function($event_data)use($worker){
        $to_connection_id = $event_data['to_connection_id'];
        $message = $event_data['content'];
        if(!isset($worker->connections[$to_connection_id]))
        {
            echo "connection not exists\n";
            return;
        }
        $to_connection = $worker->connections[$to_connection_id];
        $to_connection->send($message);
    });

    // 订阅广播事件
    $event_name = '广播';
    // 收到广播事件后向当前进程内所有客户端连接发送广播数据
    Channel\Client::on($event_name, function($event_data)use($worker){
        $message = $event_data['content'];
        foreach($worker->connections as $connection)
        {
            $connection->send($message);
        }
    });
}
```

```

    }
  });
};

$worker->onConnect = function($connection)use($worker)
{
    $msg = "workerID:{$worker->id} connectionID:{$connection->id} connected\n";
    echo $msg;
    $connection->send($msg);
};

// 用来处理http请求, 向任意客户端推送数据, 需要传workerID和connectionID
$http_worker = new Worker('http://0.0.0.0:4237');
$http_worker->name = 'publisher';
$http_worker->onWorkerStart = function()
{
    Channel\Client::connect('127.0.0.1', 2206);
};
$http_worker->onMessage = function($connection, $data)
{
    $connection->send('ok');
    if(empty($_GET['content'])) return;
    // 是向某个worker进程中某个连接推送数据
    if(isset($_GET['to_worker_id']) && isset($_GET['to_connection_id']))
    {
        $event_name = $_GET['to_worker_id'];
        $to_connection_id = $_GET['to_connection_id'];
        $content = $_GET['content'];
        Channel\Client::publish($event_name, array(
            'to_connection_id' => $to_connection_id,
            'content'          => $content
        ));
    }
    // 是全局广播数据
    else
    {
        $event_name = '广播';
        $content = $_GET['content'];
        Channel\Client::publish($event_name, array(
            'content' => $content
        ));
    }
};

Worker::runAll();

```

测试（假设都是本机127.0.0.1运行）

1、运行服务端

```

php start.php start
Workerman[start.php] start in DEBUG mode
----- WORKERMAN -----

```

```

Workerman version:3.2.7          PHP version:5.4.37
----- WORKERS -----
user      worker      listen      processes  status
root      ChannelServer  frame://0.0.0.0:2206    1          [OK]
root      pusher         websocket://0.0.0.0:4236 2          [OK]
root      publisher     http://0.0.0.0:4237     1          [OK]
-----
Press Ctrl-C to quit. Start success.

```

2、客户端连接服务端

打开chrome浏览器，按F12打开调试控制台，在Console一栏输入(或者把下面代码放入到html页面用js运行)

```

// 假设服务端ip为127.0.0.1, 测试时请改成实际服务端ip
ws = new WebSocket("ws://127.0.0.1:4236");
ws.onmessage = function(e) {
    alert("收到服务端的消息：" + e.data);
};

```

3、通过调用http接口推送

url访问 `http://127.0.0.1:4237/?content={$content}` 向所有客户端连接推送
`$content` 数据

url访问 `http://127.0.0.1:4237/?to_worker_id={$worker_id}&to_connection_id={$connection_id}&content={$content}` 向某个worker进程中的某个客户端连接推送
```$content``` 数据

注意：测试时把 `127.0.0.1` ```{$worker_id}``` { $connection_id } 和 { $content } 换成实际值`

# 例子-分组发送

## 例子1

(要求Workerman版本>=3.3.0)

基于Worker的多进程(分布式集群)分组推送系统

```
require_once __DIR__ . '/../Workerman/Autoloader.php';
require_once __DIR__ . '/Channel/src/Server.php';
require_once __DIR__ . '/Channel/src/Client.php';
use Workerman\Worker;

$channel_server = new Channel\Server('0.0.0.0', 2206);

$worker = new Worker('websocket://0.0.0.0:1234');
$worker->count = 8;
// 全局群组到连接的映射数组
$group_con_map = array();
$worker->onWorkerStart = function(){
 // Channel客户端连接到Channel服务端
 Channel\Client::connect('127.0.0.1', 2206);

 // 监听全局分组发送消息事件
 Channel\Client::on('send_to_group', function($event_data){
 $group_id = $event_data['group_id'];
 $message = $event_data['message'];
 global $group_con_map;
 var_dump(array_keys($group_con_map));
 if (isset($group_con_map[$group_id])) {
 foreach ($group_con_map[$group_id] as $con) {
 $con->send($message);
 }
 }
 });
};
$worker->onMessage = function($con, $data){
 // 加入群组消息{"cmd":"add_group", "group_id":"123"}
 // 或者 群发消息{"cmd":"send_to_group", "group_id":"123", "message"
 : "这个是消息"}
 $data = json_decode($data, true);
 var_dump($data);
 $cmd = $data['cmd'];
 $group_id = $data['group_id'];
 switch($cmd) {
 // 连接加入群组
 case "add_group":
 global $group_con_map;
 // 将连接加入到对应的群组数组里
 $group_con_map[$group_id][$con->id] = $con;
 // 记录这个连接加入了哪些群组, 方便在onclose的时候清理group_con_
```

```

map对应群组的数据
 $con->group_id = isset($con->group_id) ? $con->group_id :
 array();
 $con->group_id[$group_id] = $group_id;
 break;
 // 群发消息给群组
 case "send_to_group":
 // Channel\Client给所有服务器的所有进程广播分组发送消息事件
 Channel\Client::publish('send_to_group', array(
 'group_id'=>$group_id,
 'message'=>$data['message']
));
 break;
 }
};
// 这里很重要，连接关闭时把连接从全局群组数据中删除，避免内存泄漏
$worker->onClose = function($con){
 global $group_con_map;
 // 遍历连接加入的所有群组，从group_con_map删除对应的数据
 if (isset($con->group_id)) {
 foreach ($con->group_id as $group_id) {
 unset($group_con_map[$group_id][$con->id]);
 }
 if (empty($group_con_map[$group_id])) {
 unset($group_con_map[$group_id]);
 }
 }
};

Worker::runAll();

```

## 测试（假设都是本机127.0.0.1运行）

### 1、运行服务端

```

php start.php start
Workerman[del.php] start in DEBUG mode
----- WORKERMAN -----
Workerman version:3.4.2 PHP version:7.1.3
----- WORKERS -----
user worker listen processes status
liliang ChannelServer frame://0.0.0.0:2206 1 [OK]
liliang none websocket://0.0.0.0:1234 12 [OK]

Press Ctrl-C to quit. Start success.

```

### 2、客户端连接服务端

打开chrome浏览器，按F12打开调试控制台，在Console一栏输入(或者把下面代码放入到html页面用js运行)

```
// 假设服务端ip为127.0.0.1, 测试时请改成实际服务端ip
ws = new WebSocket('ws://127.0.0.1:1234');
ws.onmessage = function(data){console.log(data.data)};
ws.onopen = function() {
 ws.send('{"cmd":"add_group", "group_id":"123"}');
 ws.send('{"cmd":"send_to_group", "group_id":"123", "message":"这个是消息"}');
};
```

# FileMonitor文件监控组件

## 文件监控组件

背景：

Workerman是常驻内存运行的，常驻内存可以避免重复读取磁盘、重复解释编译PHP，以便达到最高性能。所以更改业务代码后需要手动reload或者restart才能生效。

同时workerman提供一个监控文件更新的服务，该服务检测到有文件更新后会自动运行reload，从新载入PHP文件。开发者将其放入到项目中随着项目启动即可。

文件监控服务下载地址：

- 1、无依赖版本：<https://github.com/walkor/workerman-filemonitor>
- 2、依赖inotify版本：<https://github.com/walkor/workerman-filemonitor-inotify> (需要安装inotify扩展)

两个版本区别：

地址1版本使用的是每秒轮询文件更新时间的方法判断文件是否更新，

地址2利用Linux内核inotify机制，文件更新时系统会主动通知workerman。

一般使用第一个无依赖版本即可

使用方法

将Applications/FileMonitor目录拷贝到你项目的Applications目录下即可。

如果你的项目没有Applications目录，可以将Applications/FileMonitor/start.php文件拷贝到你的项目任意位置，在启动脚本中require到启动脚本中即可。

监控组件默认监控的是Applications目录，如果需要更改，可以修改Applications/FileMonitor/start.php中的`\$monitor\_dir`变量，`\$monitor\_dir`的值建议是绝对路径。

注意：

- windows系统不支持reload，无法使用此监控服务。



- 只有在debug模式下才生效，daemon下不会执行文件监控（为何不支持daemon模式见下面说明）。
- 只有在Worker::runAll运行后加载的文件才能热更新，或者说只有在onXXX回调中加载的文件才能热更新。

### 为何不支持daemon模式？

daemon模式一般为线上正式环境运行的模式。正式环境发布版本时，一般一次发布多个文件，文件之间也可能有依赖。由于多个文件同步到磁盘需要一定时间，会存在某个时刻磁盘上的文件不全的情况，如果这时候监控到了文件更新并执行了reload，则会有找不到文件导致致命错误的风险。

另外正式环境中有时候会在线定位bug，如果直接编辑代码保存，就会立刻生效，有可能出现语法错误导致线上服务不可用。正确的方法应该是保存代码后，通过 `php -l yourfile.php` 检查下是否有语法错误，然后再reload热更新代码。

如果开发者确实需要daemon模式开启文件监控及自动更新，可以自行更改Applications/FileMonitor/start.php代码，将Worker::\$daemonize部分的判断去掉即可。

# MySQL组件

---

[workerman/mysql](#)

[react/mysql\(异步\)](#)

# workerman/mysql

## Workerman/MySQL

### 说明

常驻内存的程序在使用mysql时经常会遇到 `mysql gone away` 的错误，这个是由于程序与mysql的连接长时间没有通讯，连接被mysql服务端踢掉导致。本数据库类可以解决这个问题，当发生 `mysql gone away` 错误时，会自动重试一次。

### 依赖的扩展

该mysql类依赖[pdo](#)和[pdo\\_mysql](#)两个扩展，缺少扩展会报 `Undefined class constant 'MYSQL\_ATTR\_INIT\_COMMAND' in ....` 错误。

命令行运行 `php -m` 会列出所有php cli已安装的扩展，如果没有pdo 或者 pdo\_mysql，请自行安装。

### centos系统

#### PHP5.x

```
yum install php-pdo
yum install php-mysql
```

#### PHP7.x

```
yum install php70w-pdo_dblib.x86_64
yum install php70w-mysqlnd.x86_64
```

如果找不到包名，请尝试用 `yum search php mysql` 查找

### ubuntu/debian系统

#### PHP5.x

```
apt-get install php5-mysql
```

#### PHP7.x

```
apt-get install php7.0-mysql
```

如果找不到包名，请尝试用 `apt-cache search php mysql` 查找

以上方法无法安装？

如果以上方法无法安装，请参考[workerman手册-附录-扩展安装-方法三源码编译安装](#)。

## 安装 Workerman/MySQL

方法1：

可以通过composer安装，命令行运行以下命令(composer源在国外，安装过程可能会非常慢)。

```
composer require workerman/mysql
```

上面命令成功后会生成vendor目录，然后在项目中引入vendor下的autoload.php。

```
require_once __DIR__ . '/vendor/autoload.php';
```

方法2：

[下载源码](#)，解压后的目录放到自己项目中(位置任意)，直接require源文件。

```
require_once '/your/path/of/mysql-master/src/Connection.php';
```

## 注意

强烈建议在onWorkerStart回调中初始化数据库连接，避免在 `Worker::runAll();` 运行前就初始化连接，在 `Worker::runAll();` 运行前初始化的连接属于主进程，子进程会继承这个连接，主进程和子进程共用相同的数据库连接会导致错误。

## 示例

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
require_once __DIR__ . '/vendor/autoload.php';

$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
```

```

 // 将db实例存储在全局变量中(也可以存储在某类的静态成员中)
 global $db;
 $db = new \Workerman\MySQL\Connection('host', 'port', 'user', 'password', 'db_name');
 };
 $worker->onMessage = function($connection, $data)
 {
 // 通过全局变量获得db实例
 global $db;
 // 执行SQL
 $all_tables = $db->query('show tables');
 $connection->send(json_encode($all_tables));
 };
 // 运行worker
 Worker::runAll();

```

## 具体MySQL/Connection用法

```

// 初始化db连接
$db = new \Workerman\MySQL\Connection('host', 'port', 'user', 'password', 'db_name');

// 获取所有数据
$db->select('ID, Sex')->from('Persons')->where('sex= :sex AND ID = :id')->bindValues(array('sex'=>'M', 'id' => 1))->query();
//等价于
$db->select('ID, Sex')->from('Persons')->where("sex= 'M' AND ID = 1")->query();
//等价于
$db->query("SELECT ID, Sex FROM `Persons` WHERE sex='M' AND ID = 1");

// 获取一行数据
$db->select('ID, Sex')->from('Persons')->where('sex= :sex')->bindValues(array('sex'=>'M'))->row();
//等价于
$db->select('ID, Sex')->from('Persons')->where("sex= 'M' ")>row();
//等价于
$db->row("SELECT ID, Sex FROM `Persons` WHERE sex='M'");

// 获取一列数据
$db->select('ID')->from('Persons')->where('sex= :sex')->bindValues(array('sex'=>'M'))->column();
//等价于
$db->select('ID')->from('Persons')->where("sex= 'F' ")>column();
//等价于
$db->column("SELECT `ID` FROM `Persons` WHERE sex='M'");

// 获取单个值
$db->select('ID')->from('Persons')->where('sex= :sex')->bindValues(array('sex'=>'M'))->single();
//等价于
$db->select('ID')->from('Persons')->where("sex= 'F' ")>single();

```

```

//等价于
$db->single("SELECT ID FROM `Persons` WHERE sex='M'");

// 复杂查询
$db->select('*')->from('table1')->innerJoin('table2','table1.uid = table2
.uid')->where('age > :age')->groupBy(array('aid'))->having('foo="foo"')->
orderByASC/*orderByDESC*/(array('did'))
->limit(10)->offset(20)->bindValue(array('age' => 13));
// 等价于
$db->query('SELECT * FROM `table1` INNER JOIN `table2` ON `table1`.`uid`
= `table2`.`uid`
WHERE age > 13 GROUP BY aid HAVING foo="foo" ORDER BY did LIMIT 10 OFFSET
20');

// 插入
$insert_id = $db->insert('Persons')->cols(array(
 'Firstname'=>'abc',
 'Lastname'=>'efg',
 'Sex'=>'M',
 'Age'=>13))->query();
等价于
$insert_id = $db->query("INSERT INTO `Persons` (`Firstname`, `Lastname`, `
Sex`, `Age`)
VALUES ('abc', 'efg', 'M', 13)");

// 更新
$row_count = $db->update('Persons')->cols(array('sex'))->where('ID=1')
->bindValue('sex', 'F')->query();
// 等价于
$row_count = $db->update('Persons')->cols(array('sex'=>'F'))->where('ID=1
')->query();
// 等价于
$row_count = $db->query("UPDATE `Persons` SET `sex` = 'F' WHERE ID=1");

// 删除
$row_count = $db->delete('Persons')->where('ID=9')->query();
// 等价于
$row_count = $db->query("DELETE FROM `Persons` WHERE ID=9");

// 事务
$db->beginTrans();
....
$db->commitTrans(); // or $db->rollBackTrans();

```

# react/mysql(异步)

## 异步react/mysql

(要求Workerman版本 $\geq 3.3.6$ )

### 注意:

此组件是第三方组件，可能会有潜在的bug，建议使用Workerman\MySQL组件。

### 安装：

```
composer require react/mysql
```

### 示例：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Workerman\Worker;

$worker = new Worker('text://0.0.0.0:6161');

// 进程启动时
$worker->onWorkerStart = function() {
 global $mysql;
 // 获得workerman的event-loop,
 $loop = Worker::getEventLoop();
 // 连接参数
 $mysql = new React\MySQL\Connection($loop, array(
 'host' => '127.0.0.1', // 不要写localhost
 'dbname' => '数据库名',
 'user' => '用户名',
 'passwd' => '密码',
));
 // 出现错误时
 $mysql->on('error', function($e){
 echo $e;
 });
 // 执行连接
 $mysql->connect(function ($e) {
 if($e) {
 echo $e;
 } else {
 echo "connect success\n";
 }
 });
};
// 收到客户端请求时
$worker->onMessage = function($connection, $data) {
 global $mysql;
```

```

 // 执行异步查询
 $mysql->query('show databases' /*$data*/, function ($command, $mysql)
 use ($connection) {
 if ($command->hasError()) {
 $error = $command->getError();
 } else {
 $results = $command->resultRows;
 $fields = $command->resultFields;
 $connection->send(json_encode($results));
 }
 });
 };

 Worker::runAll();

```

文档：

<https://github.com/bixuehujin/reactphp-mysql>

注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 `\$loop` 变量请使用 `Worker::getEventLoop();` 返回值



# redis组件

---

[react/redis](#)

# react/redis

## redis-react

安装：

```
composer require clue/redis-react
```

注意：此组件为试验性质的第三方组件，可能有隐藏的bug，建议使用redis扩展。

示例：

```
require_once __DIR__ . '/vendor/autoload.php';
use Clue\React\Redis\Factory;
use Clue\React\Redis\Client;
use Workerman\Worker;

$worker = new Worker('text://0.0.0.0:6161');

// 进程启动时
$worker->onWorkerStart = function() {
 global $factory;
 $loop = Worker::getEventLoop();
 $factory = new Factory($loop);
};

$worker->onMessage = function($connection, $data) {
 global $factory;
 $factory->createClient('localhost:6379')->then(function (Client $client) use ($connection) {
 $client->set('greeting', 'Hello world');
 $client->append('greeting', '!');

 $client->get('greeting')->then(function ($greeting) use ($connection){
 // Hello world!
 echo $greeting . PHP_EOL;
 $connection->send($greeting);
 });

 $client->incr('invocation')->then(function ($n) use ($connection)
 {
 echo 'This is invocation #' . $n . PHP_EOL;
 $connection->send($n);
 });
 });
};

Worker::runAll();
```

文档：

<https://github.com/clue/php-redis-react>

注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 `$loop` 变量请使用 `Worker::getEventLoop();` 返回值

# 异步http组件

---

[react/http-client](#)

# react/http-client

## react/http-client

(要求Workerman版本>=3.3.6)

### 安装：

```
composer require react/http-client
```

### 示例：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Workerman\Worker;

$worker = new Worker('text://0.0.0.0:6161');

$worker->onWorkerStart = function() {
 global $client;
 $loop = Worker::getEventLoop();
 $factory = new React\Dns\Resolver\Factory();
 $dns = $factory->createCached('8.8.8.8', $loop);
 $factory = new React\HttpClient\Factory();
 $client = $factory->create($loop, $dns);
};

$worker->onMessage = function($connection, $host) {
 global $client;
 $request = $client->request('GET', $host);
 $request->on('error', function(Exception $e) use ($connection) {
 $connection->send($e);
 });
 $request->on('response', function ($response) use ($connection) {
 $response->on('data', function ($data, $response) use ($connection) {
 $connection->send($data);
 });
 });
 $request->end();
};

Worker::runAll();
```

### 文档：

<https://github.com/reactphp/http-client>

## 注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 `$loop` 变量请使用 `Worker::getEventLoop();` 返回值

## 异步消息队列组件

---

[react/zmq](#)

[react/stomp](#)

# react/zmq

## react/zmq

(要求Workerman版本>=3.3.6)

### 安装：

```
composer require react/zmq
```

### 示例：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Workerman\Worker;

$consumer = new Worker();

$consumer->onWorkerStart = function() {
 global $pull;
 $loop = Worker::getEventLoop();
 $context = new React\ZMQ\Context($loop);
 $pull = $context->getSocket(ZMQ::SOCKET_PULL);
 $pull->bind('tcp://127.0.0.1:5555');

 $pull->on('error', function ($e) {
 var_dump($e->getMessage());
 });

 $pull->on('message', function ($msg) {
 echo "Received: $msg\n";
 });
};

Worker::runAll();
```

### 文档：

<https://github.com/reactphp/zmq>

### 注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 `\$loop` 变量请使用 `Worker::getEventLoop();` 返回值





# react/stomp

## react/stomp

(要求Workerman版本`>=3.3.6`)

STOMP是一个通讯协议。它是支持大多数消息队列如RabbitMQ、Apollo等。

### 安装：

```
composer require react/stomp
```

### 示例：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Workerman\Worker;

$consumer = new Worker();

$consumer->onWorkerStart = function() {
 global $client;
 $loop = Worker::getEventLoop();
 $factory = new React\Stomp\Factory($loop);
 $client = $factory->createClient(array('vhost' => '/', 'login' => 'guest', 'passcode' => 'guest'));

 $client
 ->connect()
 ->then(function ($client) use ($loop) {
 $client->subscribe('/topic/foo', function ($frame) {
 echo "Message received: {$frame->body}\n";
 });
 });
};

Worker::runAll();
```

### 文档：

<https://github.com/reactphp/stomp>

### 注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 `$loop` 变量请使用 `Worker::getEventLoop()` 返回值



# 异步dns组件

---

react/dns

# react/dns

## react/dns

(要求Workerman版本>=3.3.6)

### 安装：

```
composer require react/dns
```

### 示例：

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Workerman\Worker;

$worker = new Worker('text://0.0.0.0:6161');

$worker->onWorkerStart = function() {
 global $dns;
 $loop = Worker::getEventLoop();
 $factory = new React\Dns\Resolver\Factory();
 $dns = $factory->create('8.8.8.8', $loop);
};

$worker->onMessage = function($connection, $host) {
 global $dns;
 $dns->resolve($host)->then(function($ip) use($host, $connection) {
 $connection->send("$host: $ip");
 }, function($e) use($host, $connection){
 $connection->send("$host: {$e->getMessage()}");
 });
};

Worker::runAll();
```

### 文档：

<https://github.com/reactphp/dns>

### 注意：

- 1、所有的异步编码必须在 `onXXX` 回调中编写
- 2、异步客户端需要的 ` \$loop ` 变量请使用 ` Worker::getEventLoop(); ` 返回值



# 进程控制组件

---

[react/child-process](#)

# react/child-process

---



# memcache

---

## memcache

workerman没有提供memcache的类库。  
请使用memcached扩展，参考[php手册](#)

# 常见问题

---

[心跳](#)

[客户端连接失败原因](#)

[是否支持多线程](#)

[与其它框架整合](#)

[运行多个workerman](#)

[支持哪些协议](#)

[如何设置进程数](#)

[查看客户端连接数](#)

[对象和资源的持久化](#)

[例子无法工作](#)

[启动失败](#)

[停止失败](#)

[支持多少并发](#)

[更改代码不生效](#)

[向指定客户端发送数据](#)

[如何主动推送消息](#)

[在其它项目中推送](#)

[如何实现异步任务](#)

[status里send\\_fail的原因](#)

[Windows下开发Linux下部署](#)

[是否支持socket.io](#)

[终端关闭导致workerman关闭](#)

[与nginx apache的关系](#)

[禁用函数检查](#)

[平滑重启原理](#)

[为Flash开843端口](#)

[如何广播数据](#)

[如何建立udp服务](#)

[监听ipv6](#)

[关闭未认证的连接](#)

[传输加密-ssl/tls](#)

[创建wss服务](#)

[创建https服务](#)

[workerman作为客户端](#)

[作为ws/wss客户端](#)

[PHP的几种回调写法](#)

# 心跳

## 心跳

注意：长连接应用必须加心跳，否则连接可能由于长时间未通讯被路由节点强行断开。

心跳作用主要有两个：

- 1、客户端定时给服务端发送点数据，防止连接由于长时间没有通讯而被某些节点的防火墙关闭导致连接断开的情况。
- 2、服务端可以通过心跳来判断客户端是否在线，如果客户端在规定时间内没有发来任何数据，就认为客户端下线。这样可以检测到客户端由于极端情况(断电、断网等)下线的事件。

建议值：

建议心跳间隔小于60秒

## 心跳示例

```
<?php
require_once __DIR__ . '/Workerman/Autoloader.php';
use Workerman\Worker;
use Workerman\Lib\Timer;

// 心跳间隔25秒
define('HEARTBEAT_TIME', 25);

$worker = new Worker('text://0.0.0.0:1234');

$worker->onMessage = function($connection, $msg) {
 // 给connection临时设置一个lastMessageTime属性，用来记录上次收到消息的时间
 $connection->lastMessageTime = time();
 // 其它业务逻辑...
};

// 进程启动后设置一个每秒运行一次的定时器
$worker->onWorkerStart = function($worker) {
 Timer::add(1, function()use($worker){
 $time_now = time();
 foreach($worker->connections as $connection) {
 // 有可能该connection还没收到过消息，则lastMessageTime设置为当前时间
 if (empty($connection->lastMessageTime)) {
 $connection->lastMessageTime = $time_now;
 continue;
 }
 // 上次通讯时间间隔大于心跳间隔，则认为客户端已经下线，关闭连接
 if ($time_now - $connection->lastMessageTime > HEARTBEAT_TIME
```

```
) {
 $connection->close();
}
});
};
Worker::runAll();
```

# 客户端连接失败原因

## 客户端连接失败原因

连接失败客户端一般会有两种报错，`connection refuse` 和 `connection timeout`

### connection refuse ( 连接拒绝 )

一般是以下原因:

- 1、客户端连接的端口错了
- 2、客户端连接的域名或者ip错了
- 3、如果客户端使用了域名连接，域名可能指向了错误的服务器ip
- 4、服务端没有启动或者端口没有被监听
- 5、使用了网络代理软件

### connection timeout ( 连接超时 )

一般是以下原因：

- 1、服务器防火墙阻止了连接，可以临时关闭防火墙试下
- 2、如果是云服务器，安全组也可能会阻止连接建立，需要到管理后台开放对应端口
- 3、服务器不存在或者没有启动
- 4、如果客户端使用了域名连接，域名可能指向了错误的服务器ip
- 5、客户端访问的ip是服务器内网ip，并且客户端和服务端不在一个局域网

### 其它报错

如果发生的报错不是 `connection refuse` 和 `connection timeout` 则一般是以下原因：

- 1、客户端使用的通讯协议与服务端不一致。

例如服务端是http通讯协议，客户端使用websocket通讯协议访问是无法连接的。如果客户端用websocket协议连接，那么服务端必须也是websocket协议。如果服务端是http协议的服务，那么客户端必须用http协议访问。

这里的原理类似如果你要和英国人交流，那么要使用英语。如果要和日本人交流，那么要使用日语。这里的语言就类似通讯协议，双方(客户端和服务端)必须使用相同的语言才能交流，否则无法通讯。

通讯协议不一致导致的常见的报错有：

```
WebSocket connection to 'ws://xxx.com:xx/' failed: Error during WebSocket handshake: Unexpected response code: xxx
```

```
WebSocket connection to 'ws://xxx.com:xx/' failed: Error during WebSocket handshake: net::ERR_INVALID_HTTP_RESPONSE
```

解决办法：

从上面两条报错看出，客户端使用的是ws连接是websocket协议。服务端也需要是websocket协议才行，服务端监听部分代码需要指定websocket协议才能通讯，例如下面这样

如果是gatewayWorker，监听部分代码类似

```
// websocket协议，这样客户端才能用ws://...来连。xxxx为端口不用改动
$gateway = new Gateway('websocket://0.0.0.0:xxxx');
```

如果是Workerman则是

```
// websocket协议，这样客户端才能用ws://...来连。xxxx为端口不用改动
$worker = new Worker('websocket://0.0.0.0:xxxx');
```

## 是否支持多线程

---

### Workerman是否支持多线程？

Workerman有一个依赖[pthreads扩展](#)的[MT多线程版本](#)，但是由于pthreads扩展还不够稳定，所以这个Workerman多线程版本已经不再维护。

目前Workerman及其周边产品都是基于多进程单线程的。



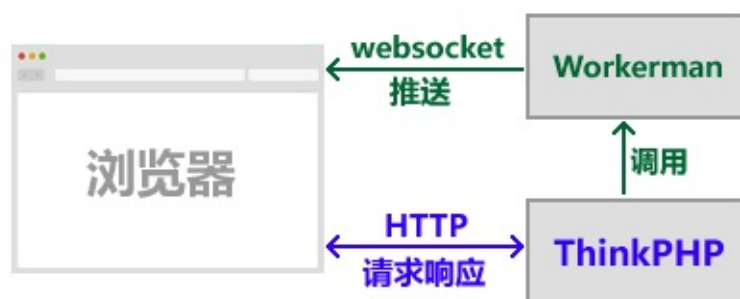
## 与其它框架整合

### 如何与其它框架整合

问：

如何与其它mvc框架（thinkPHP、Yii等）整合？

答：



与其它mvc框架结合建议以上图的方式(ThinkPHP为例)：

- 1、ThinkPHP与Workerman是两个独立的系统，独立部署(可部署在不同服务器)，互不干扰。
- 2、ThinkPHP以HTTP协议提供网页页面在浏览器渲染展示。
- 3、ThinkPHP提供的页面的js发起websocket连接，连接workerman
- 4、连接后给Workerman发送一个数据包(包含用户名密码或者某种token串)用于验证websocket连接属于哪个用户。
- 5、仅在ThinkPHP需要向浏览器推送数据时，才调用workerman的socket接口推送数据。
- 6、其余请求还是按照原本ThinkPHP的HTTP方式调用处理。

总结：

把Workerman作为一个可以向浏览器推送的通道，仅仅在需要向浏览器推送数据时才调用Workerman接口完成推送。业务逻辑全部在ThinkPHP中完成。

ThinkPHP如何调用Workerman socket接口推送数据参考[见常见问题-在其它项目中推送一节](#)

ThinkPHP官方已经支持了workerman，参见[ThinkPHP5手册](#)

## 运行多个workerman

---

### 运行多个WorkerMan实例

可以运行多个WorkerMan实例，一般只要启动文件不同并且监听端口不同即可。

## 支持哪些协议

### WorkerMan支持哪些协议

WorkerMan在接口上支持各种协议，只要符合 `ConnectionInterface` 接口即可（参见定制通讯协议章节）。

为了方便开发者，WorkerMan提供了HTTP协议、WebSocket协议以及非常简单的Text文本协议、可用于二进制传输的frame协议。开发者可以直接使用这些协议，不必再二次开发。如果这些协议都不满足需要，开发者可以参照定制协议章节实现自己的协议。

开发者也可以直接基于tcp或者udp协议。

#### 协议使用示例

```
// http协议
$worker1 = new Worker('http://0.0.0.0:1221');
// websocket协议
$worker2 = new Worker('websocket://0.0.0.0:1222');
// text文本协议（telnet协议）
$worker3 = new Worker('text://0.0.0.0:1223');
// frame文本协议（可用于二进制数传输）
$worker3 = new Worker('frame://0.0.0.0:1223');
// 直接基于tcp传输
$worker4 = new Worker('tcp://0.0.0.0:1224');
// 直接基于udp传输
$worker5 = new Worker('udp://0.0.0.0:1225');
```

# 如何设置进程数

## 应该开启多少进程 如何设置进程数

进程数是由 `count` 属性决定的，例如下面代码

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$http_worker = new Worker("http://0.0.0.0:2345");

// ## 启动4个进程对外提供服务 ##
$http_worker->count = 4;

...
```

## 进程数设置需要考虑以下条件

- 1、cpu核数
- 2、内存大小
- 3、业务偏向IO密集还是CPU密集型

## 进程数设置原则

- 1、每个进程占用内存之和需要小于总内存（一般来说每个业务进程占用内存大概40M左右）
- 2、如果是IO密集型，也就是业务中涉及到一些阻塞式IO，比如一般的访问Mysql、Redis等存储都是阻塞式访问的，进程数可以开大一些，如配置成CPU核数的3倍。如果业务中涉及的阻塞等待很多，可以再适当加大进程数，例如CPU核数的5倍甚至更高。注意非阻塞式IO属于CPU密集型，而不属于IO密集型。
- 3、如果是CPU密集型，也就是业务中没有阻塞式IO开销，例如使用异步IO读取网络资源，进程不会被业务代码阻塞的情况下，可以把进程数设置成和CPU核数一样

进程数设置原理有点像生产线上工人们工作的情景，有4个生产线，就相当于CPU是4核的，生产线上的产品相当于任务，工人相当于进程。

工人并非越多越好，因为只有4条生产线，生产能力有限，并且工人多了大家轮流上下生产线开销也很大，同样的进程多了进程间切换开销也很大。

如果流水线上的商品每一步操作起来比较耗时（IO密集型），就需要多一点的工人去操作。

如果流水线上的商品每一步操作都比较快（CPU密集型），只需要少数工人即可

## 进程数设置参考值

如果业务代码偏向IO密集型，也就是业务代码有IO阻塞的地方，则根据IO密集程度设置进程数，例如CPU核数的3倍。

如果业务代码偏向CPU密集型，也就是业务代码中无IO通讯或者无阻塞式IO通讯，则可以将进程数设置成cpu核数。

## 注意

WorkerMan自身的IO都是非阻塞的，例如 `Connection->send` 等都是非阻塞的，属于CPU密集型操作。如果不清楚自己业务偏向于哪种类型，可设置进程数为CPU核数的2倍左右即可。

## 查看客户端连接数

---

### 查看当前客户端连接数

运行 `php start.php status` 能看到当前服务器的WorkerMan运行的状态，`connections` 字段标记了每个进程当前TCP连接数。需要注意的是这个字段不仅包括客户端的TCP连接数，也包括WorkerMan内部通讯的TCP连接数。例如WorkerMan中的Gateway/Worker模型中，每个Gateway进程当前的客户端连接数为 `connections` 字段的值减去Worker进程数。

# 对象和资源的持久化

## 对象和资源的持久化

在传统的Web开发中，PHP创建的对象、数据、资源等会在请求完毕后全部释放，导致很难做到持久化。而在WorkerMan中可以轻松做到这些。

在WorkerMan中如果想在内存中永久保存某些数据资源，可以将资源放到全局变量中或者类的静态成员中。

例如下面的代码：

用一个全局变量 ` \$connection\_count ` 保存一个当前进程的客户端连接数。

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

// 全局变量，保存当前进程的客户端连接数
$connection_count = 0;

$worker = new Worker('tcp://0.0.0.0:1236');

$worker->onConnect = function($connection)
{
 // 有新的客户端连接时，连接数+1
 global $connection_count;
 ++$connection_count;
 echo "now connection_count=$connection_count\n";
};

$worker->onClose = function($connection)
{
 // 客户端关闭时，连接数-1
 global $connection_count;
 $connection_count--;
 echo "now connection_count=$connection_count\n";
};
```

PHP变量作用域参见：

<http://php.net/manual/zh/language.variables.scope.php>





## 例子无法工作

---

### workerman例子无法工作现象

workerman已经正常启动，但是按照官网写的例子或者下载的demo无法工作，例如页面打不开，socket连接失败等

### 解决方法

一般这种workerman启动没报错，但是无法打开页面或者无法连接的问题都是由于服务器防火墙导致的。请先关闭服务器防火墙再进行测试，如果经确认是防火墙问题，请重新设置防火墙规则。

# 启动失败

## workerman启动失败

### 现象1

启动后报错类似如下：

```
php start.php start
PHP Warning: stream_socket_server(): unable to connect to tcp://xx.xx.xx
.xx:xxxx (Address already in use) in /home/workerman-chat/Workerman/Worke
r.php on line xxxx
```

关键字：`Address already in use`

失败原因：

端口被占用，无法启动。

可以通过命令`netstat -anp | grep 端口号`来找出哪个程序占用了端口。  
然后停止对应的程序释放端口解决。

如果不能停止对应端口的程序，可以通过更换workerman的端口解决。

如果是Workerman占用的端口，又无法通过stop命令停止(一般是丢失pid文件或者主进程被开发者kill了导致)，可以通过运行以下两个命令杀死Workerman进程。

```
killall php
ps aux|grep WorkerMan|awk '{print $2}'|xargs kill -9
```

如果确实没有程序监听这个端口，那么可能是开发者在workerman里设置了两个或两个以上的监听，并且监听的端口相同导致，请开发者自行检查启动脚本是否监听了相同的端口。

### 现象2

启动后报错类似如下：

```
PHP Warning: stream_socket_server(): unable to connect to tcp://xx.xx.xx
.xx:xxx (Cannot assign requested address) in /home/GatewayWorker/Workerma
n/Worker.php on line xxxx
```

关键字： `` Cannot assign requested address ``

失败原因：

启动脚本监听ip参数写错，不是本机ip，请填写本机ip机或者填写 `` 0.0.0.0 ``（表示监听本机所有ip）即可解决。

提示：Linux系统可以通过命令 `` ifconfig `` 查看本机所有网卡ip。

如果您是云服务器(阿里云/腾讯云等)用户，注意您的公网ip实际可能是个代理ip(例如阿里云的专有网络)，公网ip并不属于当前的服务器，所以无法通过公网ip监听。虽然不能用公网ip监听，但是仍然可以通过0.0.0.0来绑定。

### 现象3

```
Warning stream_socket_server has been disabled for security reasons in ...
```

失败原因：

stream\_socket\_server 函数被php.ini禁用

解决方法

- 1、运行 `` php --ini `` 找到php.ini文件
- 2、打开php.ini找到disable\_functions一项，将stream\_socket\_server禁用项删掉

### 现象4

```
PHP Warning: stream_socket_server(): unable to connect to tcp://0.0.0.0:xxx (Permission denied)
```

失败原因

linux下监听端口如果小于1024，需要root权限。

解决办法

使用大于1024的端口或者使用root用户启动服务。



# 停止失败

## 停止失败

现象：

运行 ``php start.php stop`` 提示 `stop fail``

原因：几种可能性

第一种可能性：

前提是以debug方式启动的workerman，开发者在终端按了 ``ctrl z`` 给workerman发送了 ``SIGSTOP`` 信号，导致workerman进入后台并挂起(暂停)，所以无法响应stop命令(``SIGINT`` 信号)。

解决：

在启动workerman的终端输入 ``fg`` (发送 ``SIGCONT`` 信号)然后回车，将workerman切回前台运行，按 ``ctrl c`` (发送 ``SIGINT`` 信号)停止workerman。

如果无法停止，尝试运行以下两条命令

```
killall -9 php
```

```
ps aux|grep WorkerMan|awk '{print $2}'|xargs kill -9
```

第二种可能性：

运行stop的用户和workerman启动用户不一致，即stop用户没有权限停止workerman。

解决：

切换到启动workerman的用户，或者用权限更高的用户停止workerman。

第三种可能性：

保存workerman主进程pid文件被删除，导致脚本找不到pid进程，导致停止失败。

解决：

将pid文件保存到安全的位置，参见手册[Worker::\\$pidFile](#)。

第四种可能性：

workerman主进程pid文件对应的进程不是workerman进程。

解决：

打开workerman的主进程的pid文件查看主进程pid，pid文件默认在Workerman平行的目录里。运行命令 `ps aux | grep 主进程pid` 查看对应的进程是否是Workerman进程，如果不是，可能是服务器重启过，导致workerman保存的pid是过期的pid，而这个pid刚好被其它进程使用，导致停止失败。如果是这种情况，将pid文件删除即可。

# 支持多少并发

## WorkerMan支持多少并发

并发概念太模糊，这里以两种可以量化的指标并发连接数和并发请求数来说明。

并发连接数是指服务器当前时刻一共维持了多少TCP连接，而这些连接上是否有数据通讯并不关注，例如一台消息推送服务器上可能维持了百万的设备连接，由于连接上很少有数据通讯，所以这台服务器上负载可能几乎为0，只要内存足够，还可以继续接受连接。

并发请求数一般用QPS（服务器每秒处理多少请求）来衡量，而当前时刻服务器上有多少个tcp连接并不十分关注。例如一台服务器只有10个客户端连接，每个客户端连接上每秒有1W个请求，那么要求服务端需要至少能支撑 $10 \times 1W = 10W$ 每秒的吞吐量（QPS）。假设10W吞吐量每秒是这台服务器的极限，如果每个客户端每秒发送1个请求给服务端，那么这台服务器能够支撑10W个客户端。

并发连接数受限于服务器内存，一般24G内存workerman服务器可以支持大概120W并发连接。

并发请求数受限于服务器cpu处理能力，一台24核workerman服务器可以达到45W每秒的吞吐量(QPS)，实际值根据业务复杂度以及代码质量有所变化。

### 注意

高并发场景必须安装libevent扩展，参考安装配置章节。另外需要优化linux内核，尤其是进程打开文件数限制，请参考附录内核调优章节。

### 压测数据

这里仅提供workerman压测的QPS数据参考。

#### 测试环境：

- 系统：debian 6.0 64位
- 内存：64G
- cpu：Intel(R) Xeon(R) CPU E5-2420 0 @ 1.90GHz（2颗物理cpu，6核心，2线程）
- Workerman：开启200个Benchmark进程
- 压测脚本：benchmark
- 业务：发送并返回hello字符串



## 普通PHP（版本5.3.10）压测

短连接（每次请求完成后关闭连接，下次请求建立新的连接）：

条件：压测脚本开500个并发线程模拟500个并发用户，每个线程连接Workerman 10W次，每次连接发送1个请求

结果：吞吐量：2.3W/S，cpu利用率：36%

长连接（每次请求后不关闭连接，下次请求继续复用这个连接）：

条件：压测脚本开2000个并发线程模拟2000个并发用户，每个线程连接Workerman 1次，每个连接发送10W请求

结果：吞吐量：36.7W/S，cpu利用率：69%

内存：每个进程内存稳定在6444K，无内存泄漏

以上是php5.3版本压测数据，如果用php7，性能会再次提升40%左右。

## HHVM环境压测

短连接（每次请求完成后关闭连接，下次请求建立新的连接）：

条件：压测脚本开1000个并发线程模拟1000个并发用户，每个线程连接Workerman 10W次，每次连接发送1个请求

结果：吞吐量：3.5W/S，cpu利用率：35%

长连接（每次请求后不关闭连接，下次请求继续复用这个连接）：

条件：压测脚本开6000个并发线程模拟6000个并发用户，每个线程连接Workerman 1次，每个连接发送10W请求

结果：吞吐量：45W/S，cpu利用率：67%

内存：HHVM环境每个进程内存稳定在46M，无内存泄漏

以上压测脚本与WorkerMan运行在同一台机器上，并且使用的是较低的php版本

## 更改代码不生效

---

### 更改代码后不生效

原因：

Workerman是常驻内存运行的，常驻内存可以避免重复读取磁盘、重复解释编译PHP，以便达到最高性能。所以更改业务代码后需要手动reload或者restart才能生效。

同时workerman提供一个监控文件更新的服务，该服务检测到有文件更新后会自动运行reload，从新载入PHP文件。开发者将其放入到项目中随着项目启动即可。

注意：windows系统不支持reload，无法使用监控服务

文件监控服务下载地址：

- 1、无依赖版本：<https://github.com/walkor/workerman-filemonitor>
- 2、依赖inotify版本：<https://github.com/walkor/workerman-filemonitor-inotify>

两个版本区别：

地址1版本使用的是每秒轮询文件更新时间的方法判断文件是否更新，

地址2利用Linux内核inotify机制，文件更新时系统会主动通知workerman。

一般使用地址1无依赖版本即可

## 向指定客户端发送数据

### WorkerMan中如何向某个特定客户端发送数据

使用worker来做服务器，没有用GatewayWorker，如何实现向指定用户推送消息？

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
// 初始化一个worker容器，监听1234端口
$worker = new Worker('websocket://workerman.net:1234');
// ===这里进程数必须必须必须设置为1===
$worker->count = 1;
// 新增加一个属性，用来保存uid到connection的映射(uid是用户id或者客户端唯一标识)
$worker->uidConnections = array();
// 当有客户端发来消息时执行的回调函数
$worker->onMessage = function($connection, $data)
{
 global $worker;
 // 判断当前客户端是否已经验证,即是否设置了uid
 if(!isset($connection->uid))
 {
 // 没验证的话把第一个包当做uid（这里为了方便演示，没做真正的验证）
 $connection->uid = $data;
 /* 保存uid到connection的映射，这样可以方便的通过uid查找connection,
 * 实现针对特定uid推送数据
 */
 $worker->uidConnections[$connection->uid] = $connection;
 return $connection->send('login success, your uid is ' . $connection->uid);
 }
 // 其它逻辑，针对某个uid发送 或者 全局广播
 // 假设消息格式为 uid:message 时是对 uid 发送 message
 // uid 为 all 时是全局广播
 list($recv_uid, $message) = explode(':', $data);
 // 全局广播
 if($recv_uid == 'all')
 {
 broadcast($message);
 }
 // 给特定uid发送
 else
 {
 sendMessageByUid($recv_uid, $message);
 }
};

// 当有客户端连接断开时
$worker->onClose = function($connection)
{
 global $worker;
```

```

 if(isset($connection->uid))
 {
 // 连接断开时删除映射
 unset($worker->uidConnections[$connection->uid]);
 }
 };

 // 向所有验证的用户推送数据
 function broadcast($message)
 {
 global $worker;
 foreach($worker->uidConnections as $connection)
 {
 $connection->send($message);
 }
 }

 // 针对uid推送数据
 function sendMessageByUid($uid, $message)
 {
 global $worker;
 if(isset($worker->uidConnections[$uid]))
 {
 $connection = $worker->uidConnections[$uid];
 $connection->send($message);
 }
 }

 // 运行所有的worker（其实当前只定义了一个）
 Worker::runAll();

```

说明：

以上例子可以针对uid推送，虽然是单进程，但是支持个10W在线是没问题的。

注意这个例子只能单进程，也就是\$worker->count 必须是1。要支持多进程或者服务器集群的话需要Channel组件完成进程间通讯，开发也非常简单，可以参考[Channel组件集群推送例子](#)一节。

如果希望在其它系统中推送消息给客户端，可以参考[在其它项目中推送](#)一节

## 如何主动推送消息

## 如何主动推送消息

### 1、可以用定时器定时推送数据

```
require_once __DIR__ . '/Workerman/Autoloader.php';
use Workerman\Worker;
use Workerman\Lib\Timer;

$worker = new Worker('websocket://0.0.0.0:1234');
// 进程启动后定时推送数据给客户端
$worker->onWorkerStart = function($worker){
 Timer::add(1, function()use($worker){
 foreach($worker->connections as $connection) {
 $connection->send('hello');
 }
 });
};
Worker::runAll();
```

### 2、其它项目中发生某个事件时通知workerman推送数据

参见 [常见问题-在其它项目中推送](#)

## 在其它项目中推送

---

### 在其它项目中利用WorkerMan给客户端推送数据 问：

我有一个普通的web项目，想在这个项目中调用WorkerMan的接口，给客户端推送数据。

答：

基于Workerman可以参考以下连接

1、[Channel组件推送例子](#)（支持多进程/服务器集群，需要下载Channel组件）

2、[基于Worker推送](#)(单进程，最简单)

基于GatewayWorker参考下面连接

[在其它项目中通过GatewayWorker推送](#)(支持多进程/服务器集群，支持分组、组播、单个发送)

基于PHPSocket.IO参考下面连接

[web消息推送](#)(默认单进程，基于socket.io，浏览器兼容性最好)

# 如何实现异步任务

## 如何实现异步任务

问：

如何异步处理繁重的业务，避免主业务被长时间阻塞。例如我要给1000用户发送邮件，这个过程很慢，可能要阻塞数秒，这个过程中因为主流程被阻塞，会影响后续的请求，如何将这样的繁重任务交给其它进程异步处理。

答：

可以在本机或者其它服务器甚至服务器集群预先建立一些任务进程处理繁重的业务，任务进程数可以开多一些，例如cpu的10倍，然后调用方利用AsyncTcpConnection将数据异步发送给这些任务进程异步处理，异步得到处理结果。

任务进程服务端

```
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
// task worker, 使用Text协议
$task_worker = new Worker('Text://0.0.0.0:12345');
// task进程数可以根据需要多开一些
$task_worker->count = 100;
$task_worker->name = 'TaskWorker';
$task_worker->onMessage = function($connection, $task_data)
{
 // 假设发来的是json数据
 $task_data = json_decode($task_data, true);
 // 根据task_data处理相应的任务逻辑.... 得到结果, 这里省略....
 $task_result =
 // 发送结果
 $connection->send(json_encode($task_result));
};
Worker::runAll();
```

在workerman中调用

```
use Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';

// websocket服务
$worker = new Worker('websocket://0.0.0.0:8080');
```

```

$worker->onMessage = function($ws_connection, $message)
{
 // 与远程task服务建立异步连接, ip为远程task服务的ip, 如果是本机就是127.0.0.1,
 如果是集群就是lvs的ip
 $task_connection = new AsyncTcpConnection('Text://127.0.0.1:12345');
 // 任务及参数数据
 $task_data = array(
 'function' => 'send_mail',
 'args' => array('from'=>'xxx', 'to'=>'xxx', 'contents'=>'xx
x'),
);
 // 发送数据
 $task_connection->send(json_encode($task_data));
 // 异步获得结果
 $task_connection->onMessage = function($task_connection, $task_result
)use($ws_connection)
 {
 // 结果
 var_dump($task_result);
 // 获得结果后记得关闭异步连接
 $task_connection->close();
 // 通知对应的websocket客户端任务完成
 $ws_connection->send('task complete');
 };
 // 执行异步连接
 $task_connection->connect();
}

Worker::runAll();

```

这样，繁重的任务交给本机或者其它服务器的进程去做，任务完成后会异步收到结果，业务进程就不会阻塞了。



## status里send\_fail的原因

---

### status里send\_fail原因

现象：

运行status命令，看到有send\_fail的情况，是什么原因？

答：

有send\_fail通常来说不是什么大问题，一般是由于客户端主动关闭连接或者客户端停止接收数据导致的数据发送失败。

send\_fail有两种原因

1、调用send接口向客户端发送数据时发现客户端已经断开，则send\_fail计数加1。由于是客户端主动断开的，属于正常现象，一般可以忽略。

2、客户端接收缓冲区满（一般是由于客户端停止读取socket数据导致），但是服务端仍然调用send接口向客户端发送数据，导致数据积压在服务端数据发送缓冲区中（workerman为每个客户端建立了一个发送缓冲区），如果缓冲区大小超过限值（TcpConnection::\$maxSendBufferSize默认1M）则会被丢弃，触发onError事件（如果有），并导致send\_fail计数加1。

例如浏览器最小化后js一般会停止运行，浏览器不再读取服务端给它发送的websocket数据，如果服务端不断的向这个浏览器发送数据，这将导致数据积压在workerman的发送缓冲区，当缓冲数据达到上限值后，再调用send接口向客户端发送数据，则会触发onError，并且send\_fail计数加1。

总结：

send\_fail一般都是由客户端引起，一般来说由于浏览器断开导致的send\_fail一般不用担心。如果是由于客户端停止接收数据导致的send\_fail需要检查下客户端是否正常。

# Windows下开发Linux下部署

---

## Win下开发Linux部署

问：

我在用workerman-chat 开发了一个简单的即时通讯工具，不过是用的是windows版本 现在要把它放到linux环境下面该怎么操作呢？

答：

两种方法：

- 1、下载linux版本的workerman-chat，然后把你的Applications下的文件替换上去即可。
- 2、把Workerman目录换成Linux版本的Workerman即可。

第2种方法适用于所有windows版本移植到Linux系统的情况。

## 是否支持socket.io

---

### 是否支持socket.IO

这里是基于workerman实现的PHP版本的socket.IO。

项目：<https://github.com/walkor/phpsocket.io>

手册：<https://github.com/walkor/phpsocket.io/tree/master/docs/zh>

## 终端关闭导致workerman关闭

---

### 终端关闭导致服务关闭

问：

为什么我关闭了终端，Workerman就自己关闭了？

答：

Workerman有两种启动模式，debug调试模式和daemon守护进程模式。

运行 ``php xxx.php start `` 是进入debug调试模式，用于开发调试问题，当终端关闭后Workerman会随之关闭。

运行 ``php xxx.php start -d `` 进入的是daemon守护进程模式，终端关闭不会影响Workerman。

如果想Workerman不受终端影响，可以使用daemon模式启动。

## 与nginx apache的关系

---

### 与Apache和Nginx的关系

问：

Workerman和Apache/nginx/php-fpm是什么关系？Workerman和Apache/nginx/php-fpm

冲突么？

答：

Workerman和Apache/nginx/php-fpm没有任何关系，并且Workerman的运行不依赖于Apache/nginx/php-fpm。他们都是独立的容器，互不干扰，也不会冲突（在不监听同一个端口的情况下）。

Workerman是一个通用的socket服务器框架，支持长连接，支持各种协议如HTTP、WebSocket以及自定义协议。而Apache/nginx/php-fpm一般来说只用于开发HTTP协议的Web项目。

如果服务器已经部署了Apache/nginx/php-fpm，部署Workerman不会影响到它们的运行。

## 禁用函数检查

---

### 禁用函数检查

使用这个脚本检查是否有禁用函数。命令行运行 `curl -Ss http://www.workerman.net/check.php | php`

如果有提示 `Function stream_socket_server may be disabled. Please check disable_functions in php.ini` 说明workerman依赖的函数被禁用，需要在php.ini中解除禁用才能正常使用workerman。

步骤如下：

- 1、运行 `php --ini` 找到php cli所使用的php.ini文件位置
- 2、打开php.ini，找到 `disable_functions` 一项解除 `stream_socket_server` 的禁用

# 平滑重启原理

---

## 平滑重启原理

### 什么是平滑重启？

平滑重启不同于普通的重启，平滑重启可以做到在不影响用户的情况下重启服务，以便重新载入PHP程序，完成业务代码更新。

平滑重启一般应用于业务更新或者版本发布过程中，能够避免因为代码发布重启服务导致的暂时性服务不可用的影响。

注意：只有在on{...}回调中载入的文件平滑重启后才会自动更新，启动脚本中直接载入的文件或者写死的代码运行reload不会自动更新。

## 平滑重启原理

WorkerMan分为主进程和子进程，主进程负责监控子进程，子进程负责接收客户端的连接和连接上发来的请求数据，做相应的处理并返回数据给客户端。当业务代码更新时，其实我们只要更新子进程，便可以达到更新代码的目的。

当WorkerMan主进程收到平滑重启信号时，主进程会向其中一个子进程发送安全退出(让对应进程处理完毕当前请求后才退出)信号，当这个进程退出后，主进程会重新创建一个新的子进程（这个子进程载入了新的PHP代码），然后主进程再次向另外一个旧的进程发送停止命令，这样一个进程一个进程的重启，直到所有旧的进程全部被置换为止。

我们看到平滑重启实际上是让旧的业务进程逐个退出然后并逐个创建新的进程做到的。为了在平滑重启时不影响客用户，这就要求进程中不要保存用户相关的状态信息，即业务进程最好是无状态的，避免由于进程退出导致信息丢失。

## 为Flash开843端口

### 为Flash开启843端口

Flash发起socket连接远程服务端时，首先会到对应服务端的843端口请求一个安全策略文件。否则Flash无法建立与服务端的连接。在Workerman中可以用如下方法开启一个843端口，返回安全策略文件。

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

$flash_policy = new Worker('tcp://0.0.0.0:843');
$flash_policy->onMessage = function($connection, $message)
{
 $connection->send('<?xml version="1.0"?><cross-domain-policy><site-control permitted-cross-domain-policies="all"/><allow-access-from domain="*" to-ports="*" /></cross-domain-policy>'. "\0");
};

if(!defined('GLOBAL_START'))
{
 Worker::runAll();
}
```

其中xml的安全策略内容可以根据你的需要进行自定义设置。



## 如何广播数据

### 如何广播(群发)数据 范例 ( 定时广播 )

```
use Workerman\Worker;
use Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:2020');
// 这个例子中进程数必须为1
$worker->count = 1;
// 进程启动时设置一个定时器，定时向所有客户端连接发送数据
$worker->onWorkerStart = function($worker)
{
 // 定时，每10秒一次
 Timer::add(10, function()use($worker)
 {
 // 遍历当前进程所有的客户端连接，发送当前服务器的时间
 foreach($worker->connections as $connection)
 {
 $connection->send(time());
 }
 });
};
// 运行worker
Worker::runAll();
```

### 范例 ( 群聊 )

```
use Workerman\Worker;
use Workerman\Lib\Timer;
require_once './Workerman/Autoloader.php';

$worker = new Worker('websocket://0.0.0.0:2020');
// 这个例子中进程数必须为1
$worker->count = 1;
// 客户端发来消息时，广播给其它用户
$worker->onMessage = function($connection, $message)use($worker)
{
 foreach($worker->connections as $connection)
 {
 $connection->send($message);
 }
};
// 运行worker
Worker::runAll();
```

## 说明：

单进程：

以上例子只能单进程（``$worker->count=1``），因为多进程时多个客户端可能连接到不同的进程中，进程间的客户端是隔离的，无法直接通讯，也就是A进程中无法直接操作B进程的客户端connection对象发送数据。（要做到这点，需要进程间通讯，比如可以使用Channel组件，例如[例子-集群发送](#)、[例子-分组发送](#)）。

建议用GatewayWorker

在workerman基础上开发的GatewayWorker框架提供了更方便推送机制，包括组播、广播等，可以设置多进程甚至可以多服务器部署，如果需要给客户端推送数据，建议使用GatewayWorker框架。

GatewayWorker手册地址 <http://www.workerman.net/gatewaydoc/>

GatewayWorker下载地址(linux版本)

<http://www.workerman.net/download/GatewayWorker.zip>

GatewayWorker下载地址(windows版本)

<http://www.workerman.net/download/GatewayWorker-for-win.zip>

## 如何建立udp服务

---

### 如何建立udp服务

在workerman中建立udp服务很简单，类似如下代码

```
$udp_worker = new Worker('udp://127.0.0.1:9090');
$udp_worker->onMessage = function($connection, $data){
 var_dump($data);
 $connection->send('get');
};
Worker::runAll();
```

注意：因为udp是无连接的，所以udp服务没有onConnect和onClose事件。

# 监听ipv6

---

## ipv6

问：如何让客户端即能通过ipv4地址访问，也能通过ipv6地址访问？

答：在初始化容器的时候监听地址写 `:::` 即可。

例如

```
$worker = new Worker('http://:::8080');
$gateway = new Gateway('websocket://:::8081');
```

# 关闭未认证的连接

## 关闭未认证连接

问题：

如何关闭规定时间内未发送过数据的客户端，

比如30秒内没收到一条数据就自动关闭这个客户端连接，

目的是为了让未认证的连接必须在规定时间内认证

答案：

```
use Workerman\Lib\Timer;
require_once __DIR__ . '/Workerman/Autoloader.php';
$worker = new Worker('xxx://x.x.x.x:x');
$worker->onConnect = function($connection)
{
 // 临时给$connection对象添加一个auth_timer_id属性存储定时器id
 // 定时30秒关闭连接，需要客户端30秒内发送验证删除定时器
 $connection->auth_timer_id = Timer::add(30, function()use($connection)
){
 $connection->close();
 }, null, false);
};
$worker->onMessage = function($connection, $msg)
{
 $msg = json_decode($msg, true);
 switch($msg['type'])
 {
 case 'login':
 ...略
 // 验证成功，删除定时器，防止连接被关闭
 Timer::del($connection->auth_timer_id);
 break;
 ... 略
 }
 }
}
```



# 传输加密-ssl/tls

## 传输加密-ssl/tls

问：

如何保证和Workerman之间通讯安全？

答：

比较方便的做法是在通讯协议之上增加一层SSL加密层，比如wss、[https](#)协议都是基于SSL加密传输的，非常安全。Workerman自身支持SSL( ` 需要Workerman>=3.3.7 ` )，只需要设置下属性即可开启SSL。

当然开发者也可以基于某些加解密算法实现一套自己的加解密机制。

## Workerman开启ssl方法如下：

准备工作：

- 1、Workerman版本不小于3.3.7
- 2、PHP安装了openssl扩展
- 3、已经申请了证书（pem/crt文件及key文件）放在了/etc/nginx/conf.d/ssl下

代码：

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

// 证书最好是申请的证书
$context = array(
 'ssl' => array(
 'local_cert' => '/etc/nginx/conf.d/ssl/server.pem', // 也可以是cr
t文件
 'local_pk' => '/etc/nginx/conf.d/ssl/server.key',
 'verify_peer' => false,
)
);
// 这里设置的是websocket协议，也可以http协议或者其它协议
$worker = new Worker('http://0.0.0.0:443', $context);
// 设置transport开启ssl
$worker->transport = 'ssl';
$worker->onMessage = function($con, $msg) {
```

```
 $con->send('ok');
 };
 Worker::runAll();
```



# 创建wss服务

## 创建wss服务

问：

Workerman如何创建一个wss服务，使得客户端可以用wss来连接通讯，比如在微信小程序中连接服务端。

答：

wss协议实际是websocket+SSL，就是在websocket协议上加入SSL层，类似https(http+SSL)。Workerman支持websocket+SSL协议，同时也支持SSL(需要Workerman版本>=3.3.7)，所以只需要在websocket协议的基础上开启SSL即可支持wss协议。

### 方法一，直接用Workerman开启SSL

准备工作：

- 1、Workerman版本不小于3.3.7
- 2、PHP安装了openssl扩展
- 3、已经申请了证书（pem/crt文件及key文件）放在了/etc/ssl下（路径随意）

代码：

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

// 证书最好是申请的证书
$context = array(
 'ssl' => array(
 // 请使用绝对路径
 'local_cert' => '磁盘路径/server.pem', // 也可以是crt文件
 'local_pk' => '磁盘路径1/server.key',
 'verify_peer' => false,
)
);
// 这里设置的是websocket协议（端口任意，但是需要保证没被其它程序占用）
$worker = new Worker('websocket://0.0.0.0:4431', $context);
// 设置transport开启ssl, websocket+ssl即wss
$worker->transport = 'ssl';
```

```
$worker->onMessage = function($con, $msg) {
 $con->send('ok');
};

Worker::runAll();
```

通过以上的代码，Workerman就监听了wss协议，客户端就可以通过wss协议来连接workerman实现安全即时通讯了。

## 测试

打开chrome浏览器，按F12打开调试控制台，在Console一栏输入(或者把下面代码放入到html页面用js运行)

```
// 证书是会检查域名的，请使用域名连接
ws = new WebSocket("wss://域名:4431");
ws.onopen = function() {
 alert("连接成功");
 ws.send('tom');
 alert("给服务端发送一个字符串：tom");
};
ws.onmessage = function(e) {
 alert("收到服务端的消息：" + e.data);
};
```

注意：

- 1、wss端口只能通过wss协议访问，ws无法访问wss端口。
- 2、证书一般是与域名绑定的，所以测试的时候客户端请使用域名连接，不要使用ip去连。
- 3、如果出现无法访问的情况，请检查服务器防火墙。
- 4、微信小程序要求PHP版本>=5.6，因为PHP5.6以下版本不支持tls1.2。

## 方法二、利用nginx作为SSL的代理

除了用Workerman自身的SSL，也可以利用nginx作为SSL代理实现wss（注意如使用nginx代理SSL，则workerman部分不要设置ssl，避免会冲突）。

通讯原理及流程是：

- 1、客户端发起wss连接连到Nginx
- 2、nginx将wss协议的数据转换成ws协议并转发到Workerman的websocket协议端口

3、Workerman收到数据后做业务逻辑处理

4、Workerman给客户端发送消息时，则是相反的过程，数据经过nginx转换成wss协议然后发给客户端

## nginx配置参考

前提条件及准备工作：

- 1、假设Workerman监听的是8282端口(websocket协议)
- 2、已经申请了证书（pem/crt文件及key文件）放在了/etc/nginx/conf.d/ssl下
- 3、打算利用nginx开启4431端口对外提供wss代理服务（端口可以根据需要修改）

nginx配置类似如下：

```
server {
 listen 4431;

 ssl on;
 ssl_certificate /etc/ssl/server.pem;
 ssl_certificate_key /etc/ssl/server.key;
 ssl_session_timeout 5m;
 ssl_session_cache shared:SSL:50m;
 ssl_protocols SSLv3 SSLv2 TLSv1 TLSv1.1 TLSv1.2;
 ssl_ciphers ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;

 location /
 {
 proxy_pass http://127.0.0.1:8282;
 proxy_http_version 1.1;
 proxy_set_header Upgrade $http_upgrade;
 proxy_set_header Connection "Upgrade";
 proxy_set_header X-Real-IP $remote_addr;
 }
}
```

## 透过nginx wss代理如何获取客户端真实ip？

使用nginx作为wss代理，nginx实际上充当了workerman的客户端，所以在workerman上获取的客户端ip为nginx服务器的ip，并非实际的客户端ip。如何获取客户端真实ip可以参考下面的方法。

原理：

nginx将客户端真实ip通过http header传递进来，即上面nginx配置中location里的`

proxy\_set\_header X-Real-IP \$remote\_addr; 设置。workerman通过读取这个header值，将此值保存到`\$connection`对象里，(GatewayWorker可以保存到`\$\_SESSION`变量里)，使用的时候直接读取变量即可。

workerman从nginx设置的header里读取客户端ip

```
<?php
require_once __DIR__ . '/../Workerman/Autoloader.php';
use Workerman\Worker;
$worker = new Worker('websocket://0.0.0.0:7272');

// 客户端连上来时，即完成TCP三次握手后的回调
$worker->onConnect = function($connection) {
 /**
 * 客户端websocket握手时的回调onWebSocketConnect
 * 在onWebSocketConnect回调中获得nginx通过http头中的X_REAL_IP值
 */
 $connection->onWebSocketConnect = function($connection){
 /**
 * connection对象本没有realIP属性，这里给connection对象动态添加个realIP
 属性
 * 记住php对象是可以动态添加属性的，你也可以用自己喜欢的属性名
 */
 $connection->realIP = $_SERVER['HTTP_X_REAL_IP'];
 };
};
$worker->onMessage = function($connection, $data)
{
 // 当使用客户端真实ip时，直接使用$connection->realIP即可
 $connection->send($connection->realIP);
};
Worker::runAll();
```

GatewayWorker从nginx设置的header里获取客户端ip

在start\_gateway.php加上下面的代码

```
$gateway->onConnect = function($connection)
{
 $connection->onWebSocketConnect = function($connection , $http_header
)
 {
 $_SESSION['realIP'] = $_SERVER['HTTP_X_REAL_IP'];
 };
};
```

代码加完后需要重启GatewayWorker。

这样就可以在Events.php中通过 `$_SESSION['realIP']` 得到客户端的真实ip了

# 创建https服务

## 创建https服务

问：

Workerman如何创建一个https服务，使得客户端可以用过https协来连接通讯。

答：

https协议实际是http+SSL，就是在http协议上加入SSL层。Workerman支持http协议，同时也支持SSL（需要Workerman版本 $\geq 3.3.7$ ），所以只需要在http协议的基础上开启SSL即可支持https协议。

## Workerman开启SSL

准备工作：

- 1、Workerman版本不小于3.3.7
- 2、PHP安装了openssl扩展
- 3、已经申请了证书（pem/crt文件及key文件）放在了/etc/nginx/conf.d/ssl下

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

// 证书最好是申请的证书
$context = array(
 'ssl' => array(
 'local_cert' => '/etc/nginx/conf.d/ssl/server.pem', // 也可以是crt文件
 'local_pk' => '/etc/nginx/conf.d/ssl/server.key',
 'verify_peer' => false,
)
);
// 这里设置的是http协议
$worker = new Worker('http://0.0.0.0:443', $context);
// 设置transport开启ssl, 变成http+SSL即https
$worker->transport = 'ssl';
$worker->onMessage = function($con, $msg) {
 $con->send('ok');
};

Worker::runAll();
```

通过Workerman以上的代码就创建了https服务，客户端就可以通过https协议来连接workerman实现安全加密通讯了。

测试：

浏览器地址栏输入 `https://域名:4431` 访问。

注意：

- 1、https端口必须用https协议访问，http协议无法访问。
- 2、证书一般是与域名绑定的，所以测试的时候请使用域名，不要使用ip。
- 3、如果使用https无法访问请检查服务器防火墙。

## 利用nginx作为ssl的代理

除了用Workerman自身的SSL，也可以利用nginx作为SSL代理实现https。

通讯原理及流程是：

- 1、客户端发起https连接连到nginx
- 2、nginx将https协议的数据转换成http协议并转发到Workerman的http端口
- 3、Workerman收到数据后做业务逻辑处理，返回http协议的数据给nginx
- 4、nginx再将http协议的数据转换成https，转发给客户端

## nginx配置参考

前提条件及准备工作：

- 1、假设Workerman监听的是80端口(http协议)
- 2、已经申请了证书（pem/crt文件及key文件）放在了/etc/nginx/conf.d/ssl下
- 3、打算利用nginx开启4431端口对外提供wss代理服务（端口可以根据需要修改）

nginx配置类似如下：

```
server {
 listen 4431;

 ssl on;
 ssl_certificate /etc/nginx/conf.d/ssl/laychat/laychat.pem;
```

```
ssl_certificate_key /etc/nginx/conf.d/ssl/laychat/laychat.key;
ssl_session_timeout 5m;
ssl_session_cache shared:SSL:50m;
ssl_protocols SSLv3 SSLv2 TLSv1 TLSv1.1 TLSv1.2;
ssl_ciphers ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;

location /
{
 proxy_pass http://127.0.0.1:80;
 proxy_http_version 1.1;
 proxy_set_header X-Real-IP $remote_addr;
}
}
```



## workerman作为客户端

---

### workerman可以作为客户端接收处理来自远程服务端的数据么？

可以利用AsyncTcpConnection发起异步连接，让workerman作为客户端与服务端交互。

例如下面的例子

- 1、[workerman作为websocket客户端](#)
- 2、[workerman作为mysql代理](#)
- 3、[workerman作为http客户端](#)
- 4、[workerman作为http代理](#)
- 5、[workerman作为socks5代理](#)

# 作为ws/wss客户端

## 作为ws/wss客户端

有时候需要让workerman作为客户端以ws/wss协议去连接某个服务端，并与之交互。  
以下是示例。

### 1、workerman作为ws客户端

```
<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/../Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker){

 $con = new AsyncTcpConnection('ws://echo.websocket.org:80');

 $con->onConnect = function($con) {
 $con->send('hello');
 };

 $con->onMessage = function($con, $data) {
 echo $data;
 };

 $con->connect();
};

Worker::runAll();
```

### 2、workerman作为wss(ws+ssl)客户端

```
<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/../Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker){
 // ssl需要访问443端口
 $con = new AsyncTcpConnection('ws://echo.websocket.org:443');

 // 设置以ssl加密方式访问，使之成为wss
```

```

$con->transport = 'ssl';

$con->onConnect = function($con) {
 $con->send('hello');
};

$con->onMessage = function($con, $data) {
 echo $data;
};

$con->connect();
};

Worker::runAll();

```

### 3、workerman作为wss(ws+ssl)客户端 (需要本地ssl证书)

```

<?php
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/../Workerman/Autoloader.php';

$worker = new Worker();

$worker->onWorkerStart = function($worker){
 // 设置访问对方主机的本地ip及端口以及ssl证书
 $context_option = array(
 // ssl选项, 参考http://php.net/manual/zh/context.ssl.php
 'ssl' => array(
 // 本地证书路径。 必须是 PEM 格式, 并且包含本地的证书及私钥。
 'local_cert' => '/your/path/to/pemfile',
 // local_cert 文件的密码。
 'passphrase' => 'your_pem_passphrase',
 // 是否允许自签名证书。
 'allow_self_signed' => true,
 // 是否需要验证 SSL 证书。
 'verify_peer' => false
)
);

 // ssl需要访问443端口
 $con = new AsyncTcpConnection('ws://echo.websocket.org:443', $context_option);

 // 设置以ssl加密方式访问, 使之成为wss
 $con->transport = 'ssl';

 $con->onConnect = function($con) {
 $con->send('hello');
 };

 $con->onMessage = function($con, $data) {
 echo $data;
 };
};

```

```
};
$con->connect();
};
Worker::runAll();
```

# PHP的几种回调写法

## PHP几种回调写法

PHP里通过匿名函数写回调是最方便的，但是除了匿名函数方式的回调，PHP还有其它的回调写法。以下是PHP几种回调写法的示例。

### 1、匿名函数回调

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
$http_worker = new Worker("http://0.0.0.0:2345");

// 匿名函数回调
$http_worker->onMessage = function($connection, $data)
{
 // 向浏览器发送hello world
 $connection->send('hello world');
};

Worker::runAll();
```

### 2、普通函数回调

```
<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';
$http_worker = new Worker("http://0.0.0.0:2345");

// 匿名函数回调
$http_worker->onMessage = 'on_message';

// 普通函数
function on_mesage($connection, $data)
{
 // 向浏览器发送hello world
 $connection->send('hello world');
}

Worker::runAll();
```

### 3、类方法作为回调

MyClass.php

```
class MyClass{
```

```

 public function __construct(){}
 public function onWorkerStart($worker){}
 public function onConnect($connection){}
 public function onMessage($connection, $message) {}
 public function onClose($connection){}
 public function onWorkerStop($connection){}
}

```

## 启动脚本 start.php

```

<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 载入MyClass
require_once __DIR__ . '/MyClass.php';

$worker = new Worker("websocket://0.0.0.0:2346");

// 创建一个对象
$my_object = new MyClass();

// 调用类的方法
$worker->onWorkerStart = array($my_object, 'onWorkerStart');
$worker->onConnect = array($my_object, 'onConnect');
$worker->onMessage = array($my_object, 'onMessage');
$worker->onClose = array($my_object, 'onClose');
$worker->onWorkerStop = array($my_object, 'onWorkerStop');

Worker::runAll();

```

### 注意：

以上的代码结构不允许在构造函数里初始化资源(MySQL连接、Redis连接、Memcache连接等)，因为 ` \$my\_object = new MyClass(); ` 运行在主进程。以MySQL为例，在主进程初始化的MySQL连接等资源会被子进程继承，每个子进程都可以操作这个数据库连接，但是这些连接在MySQL服务端对应的是同一个连接，会发生不可预期的错误，例如 ` mysql gone away ` 错误。

以上代码结构如果需要在类的构造函数里初始化资源，可以采用以下写法。

### MyClass.php

```

class MyClass{
 protected $db = null;
 public function __construct(){
 // 假设数据库连接类是MyDbClass
 $db = new MyDbClass();
 }
}

```

```

 public function onWorkerStart($worker){}
 public function onConnect($connection){}
 public function onMessage($connection, $message) {}
 public function onClose($connection){}
 public function onWorkerStop($connection){}
}

```

## 启动脚本 start.php

```

<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

$worker = new Worker("websocket://0.0.0.0:2346");

// 在onWorkerStart里初始化类
$worker->onWorkerStart = function($worker) {
 // 载入MyClass
 require_once __DIR__.'/MyClass.php';

 // 创建一个对象
 $my_object = new MyClass();

 // 调用类的方法
 $worker->onConnect = array($my_object, 'onConnect');
 $worker->onMessage = array($my_object, 'onMessage');
 $worker->onClose = array($my_object, 'onClose');
 $worker->onWorkerStop = array($my_object, 'onWorkerStop');
};

Worker::runAll();

```

上面代码结构中onWorkerStart运行时已经是属于子进程，等于每个子进程各自建立自己的MySQL连接，所以不会有共享连接的问题。

这样还有一个好处就是支持业务代码reload。由于MyClass.php是在子进程载入的，根据reload规则业务更改MyClass.php后直接reload即可生效。

## 4、类的静态方法作为回调

### 静态类MyClass.php

```

class MyClass{
 public static function onWorkerStart($worker){}
 public static function onConnect($connection){}
 public static function onMessage($connection, $message) {}
 public static function onClose($connection){}
 public static function onWorkerStop($connection){}
}

```

## 启动脚本 start.php

```

<?php
use Workerman\Worker;
require_once __DIR__ . '/Workerman/Autoloader.php';

// 载入MyClass
require_once __DIR__ . '/MyClass.php';

$worker = new Worker("websocket://0.0.0.0:2346");

// 调用类的静态方法。
$worker->onWorkerStart = array('MyClass', 'onWorkerStart');
$worker->onConnect = array('MyClass', 'onConnect');
$worker->onMessage = array('MyClass', 'onMessage');
$worker->onClose = array('MyClass', 'onClose');
$worker->onWorkerStop = array('MyClass', 'onWorkerStop');

// 如果类带命名空间，则是类似这样的写法
// $worker->onWorkerStart = array('your\namespace\MyClass', 'onWorkerStart');
// $worker->onConnect = array('your\namespace\MyClass', 'onConnect');
// $worker->onMessage = array('your\namespace\MyClass', 'onMessage');
// $worker->onClose = array('your\namespace\MyClass', 'onClose');
// $worker->onWorkerStop = array('your\namespace\MyClass', 'onWorkerStop');

Worker::runAll();

```

注意：根据PHP的运行机制，如果没用调用new 则不会调用构造函数，另外静态类的方法里面不允许使用 ` \$this `。



## 附录

---

[Linux内核调优](#)

[压力测试](#)

[安装扩展](#)

[websocket协议](#)

[ws协议](#)

[text协议](#)

[frame协议](#)

[不支持的函数/特性](#)

# Linux内核调优

## Linux内核调优

打开文件 /etc/sysctl.conf，增加以下设置

```
#该参数设置系统的TIME_WAIT的数量，如果超过默认值则会被立即清除
net.ipv4.tcp_max_tw_buckets = 20000
#定义了系统中每一个端口最大的监听队列的长度，这是个全局的参数
net.core.somaxconn = 65535
#对于还未获得对方确认的连接请求，可保存在队列中的最大数目
net.ipv4.tcp_max_syn_backlog = 262144
#在每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许送到队列的数据包的最大数目
net.core.netdev_max_backlog = 30000
#能够更快地回收TIME-WAIT套接字。此选项会导致处于NAT网络的客户端超时，建议为0
net.ipv4.tcp_tw_recycle = 0
#系统所有进程一共可以打开的文件数量
fs.file-max = 6815744
#防火墙跟踪表的大小。注意：如果防火墙没开则会提示error: "net.netfilter.nf_contrack_max" is an unknown key, 忽略即可
net.netfilter.nf_conntrack_max = 2621440
```

运行 `sysctl -p` 即可生效。

说明：

/etc/sysctl.conf 可设置的选项很多，其它选项可以根据自己的环境需要进行设置

## 打开文件数

设置系统打开文件数设置，解决高并发下 `too many open files` 问题。此选项直接影响单个进程容纳的客户端连接数。

Soft open files 是Linux系统参数，影响系统单个进程能够打开最大的文件句柄数量，这个值会影响到长连接应用如聊天中单个进程能够维持的用户连接数，运行 `ulimit -n` 能看到这个参数值，如果是1024，就是代表单个进程只能同时最多只能维持1024甚至更少（因为有其它文件的句柄被打开）。如果开启4个进程维持用户连接，那么整个应用能够同时维持的连接数不会超过4\*1024个，也就是说最多只能支持4x1024个用户在线可以增大这个设置以便服务能够维持更多的TCP连接。

Soft open files 修改方法：

( 1 ) ulimit -HSn 102400

这只是在当前终端有效，退出之后，open files 又变为默认值。

( 2 ) 将ulimit -HSn 102400写到/etc/profile中，这样每次登录终端时，都会自动执行/etc/profile。

( 3 ) 令修改open files的数值永久生效，则必须修改配置文件：/etc/security/limits.conf. 在这个文件后加上：

```
* soft nfile 1024000
* hard nfile 1024000
root soft nfile 1024000
root hard nfile 1024000
```

这种方法需要重启机器才能生效。

# 压力测试

## 压力测试

### 测试环境：

- 系统：debian 6.0 64位
- 内存：64G
- cpu：Intel(R) Xeon(R) CPU E5-2420 0 @ 1.90GHz（2颗物理cpu，6核心，2线程）
- Workerman：开启200个Benchmark进程
- 压测脚本：benchmark
- 业务：发送并返回hello字符串

### 普通PHP（版本5.3.10）压测

短连接（每次请求完成后关闭连接，下次请求建立新的连接）：

条件：压测脚本开500个并发线程模拟500个并发用户，每个线程连接Workerman 10W次，每次连接发送1个请求

结果：吞吐量：2.3W/S，cpu利用率：36%

长连接（每次请求后不关闭连接，下次请求继续复用这个连接）：

条件：压测脚本开2000个并发线程模拟2000个并发用户，每个线程连接Workerman 1次，每个连接发送10W请求

结果：吞吐量：36.7W/S，cpu利用率：69%

内存：每个进程内存稳定在6444K，无内存泄漏

### HHVM环境压测

短连接（每次请求完成后关闭连接，下次请求建立新的连接）：

条件：压测脚本开1000个并发线程模拟1000个并发用户，每个线程连接Workerman 10W次，每次连接发送1个请求

结果：吞吐量：3.5W/S，cpu利用率：35%

长连接（每次请求后不关闭连接，下次请求继续复用这个连接）：

条件：压测脚本开6000个并发线程模拟6000个并发用户，每个线程连接Workerman 1次，每个连接发送10W请求

结果：吞吐量：45W/S，cpu利用率：67%

内存：HHVM环境每个进程内存稳定在46M，无内存泄漏

以上压测脚本与WorkerMan运行在同一台机器上，使用的是较低版本的PHP5.3，如果使用PHP7，则性能会再次提高40%左右。

## 压力测试需要内核调优

参见 [附录-内核调优](#) 章节



## 安装扩展

### 安装扩展

#### 注意

与Apache+PHP或者Nginx+PHP的运行模式不同，WorkerMan是基于PHP命令行 [PHP CLI](#) 运行的，使用的是不同的PHP可执行程序，使用的php.ini文件也可能不同。所以在网页中打印 `phpinfo()` 看到安装了某个扩展，不代表命令行的PHP CLI也安装了对应的扩展。

#### 如何确定PHP CLI安装了哪些扩展

运行 `php -m` 会列出命令行 PHP CLI 已经安装的扩展，结果类似如下：

```
~# php -m
[PHP Modules]
libevent
posix
pcntl
...
```

#### 如何确定PHP CLI 的php.ini文件的位置

当我们安装扩展时，可能需要手动配置php.ini文件，把扩展加进去，所以要确认PHP CLI的php.ini文件的位置。可以运行 `php --ini` 查找PHP CLI的ini文件位置，结果类似如下(各个系统显示结果会有差异)：

```
~# php --ini
Configuration File (php.ini) Path: /etc/php5/cli
Loaded Configuration File: /etc/php5/cli/php.ini
Scan for additional .ini files in: /etc/php5/cli/conf.d
Additional .ini files parsed: /etc/php5/cli/conf.d/apc.ini,
 /etc/php5/cli/conf.d/libevent.ini,
 /etc/php5/cli/conf.d/memcached.ini,
 /etc/php5/cli/conf.d/mysql.ini,
 /etc/php5/cli/conf.d/pdo.ini,
 /etc/php5/cli/conf.d/pdo_mysql.ini
...
```

### 给PHP CLI安装扩展（安装memcached扩展为例）

#### 方法一、使用apt或者yum命令安装

如果PHP是通过 apt 或者 yum 命令安装的，则扩展也可以通过 apt 或者 yum 安装

debian/ubuntu等系统apt安装PHP扩展方法（非root用户需要加sudo命令）

### 1、利用 ` apt-cache search ` 查找扩展包

```
~# apt-cache search memcached php
php-apc - APC (Alternative PHP Cache) module for PHP 5
php5-memcached - memcached module for php5
```

### 2、使用 ` apt-get install ` 安装扩展包

```
~# apt-get install -y php5-memcached
Reading package lists... Done
Reading state information... Done
...
```

centos等系统yum安装PHP扩展方法

### 1、利用 ` yum search ` 查找扩展包

```
~# yum search memcached php
php-pecl-memcached - memcached module for php5
```

### 2、使用 ` yum install ` 安装扩展包

```
~# yum install -y php-pecl-memcached
Reading package lists... Done
Reading state information... Done
...
```

说明：

使用apt或者yum安装PHP扩展会自动配置php.ini文件，安装完直接可用，十分方便。缺点是有些扩展在apt或者yum中没有对应的扩展安装包。

## 方法二、使用pecl安装

使用 ` pecl install ` 命令安装扩展

### 1、 ` pecl install ` 安装

```
~# pecl install memcached
downloading memcached-2.2.0.tgz ...
```

```
Starting to download memcached-2.2.0.tgz (70,449 bytes)
.....
```

## 2、配置php.ini

通过运行 `php --ini` 查找php.ini文件位置，然后在文件中添加 `extension=memcached.so`

## 方法三、源码编译安装（一般是安装PHP自带的扩展，以安装pcntl扩展为例）

### 1、利用 `php -v` 命令查看当前的PHP CLI的版本

```
~# php -v
PHP 5.3.29-1~dotdeb.0 with Suhosin-Patch (cli) (built: Aug 14 2014 19:55:20)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2014 Zend Technologies
```

### 2、根据版本下载PHP源代码

PHP历史版本下载页面：<http://php.net/releases/>

### 3、解压源码压缩包

例如下载的压缩包名称是 `php-5.3.29.tar.gz`

```
~# tar -zxvf php-5.3.29.tar.gz
php-5.3.29/
php-5.3.29/README.WIN32-BUILD-SYSTEM
php-5.3.29/netware/
...
```

### 4、进入源码中的ext/pcntl目录

```
~# cd php-5.3.29/ext/pcntl/
```

### 5、运行 `phpize` 命令

```
~# phpize
Configuring for:
PHP Api Version: 20090626
Zend Module Api No: 20090626
```



```
Zend Extension Api No: 220090626
```

## 6、运行 `configure` 命令

```
~# ./configure
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
...
```

## 7、运行 `make` 命令

```
~# make
/bin/bash /tmp/php-5.3.29/ext/pcntl/libtool --mode=compile cc ...
-I/usr/include/php5 -I/usr/include/php5/main -I/usr/include/php5/TSRM -I/
usr/include/php5/Zend...
...
```

## 8、运行 `make install` 命令

```
~# make install
Installing shared extensions: /usr/lib/php5/20090626/
```

## 9、配置ini文件

通过运行 `php --ini` 查找php.ini文件位置，然后在文件中添加 `extension=pcntl.so`

说明：

此方法一般用来安装PHP自带的扩展，例如posix扩展和pcntl扩展。除了用phpize编译某个扩展，也可以重新编译整个PHP，在编译时用参数添加扩展，例如在源码根目录运行

```
~# ./configure --enable-pcntl --enable-posix ...
~# make && make install
```

## 方法四、phpize安装

如果要安装的扩展在php源码ext目录中没有，那么这个扩展需要到<http://pecl.php.net> 搜索下载

以安装libevent扩展为例（假设系统安装了libevent-dev库）

## 1、下载libevent扩展文件压缩包（在当前系统哪个目录下下载随意）

```
~# wget http://pecl.php.net/get/libevent-0.1.0.tgz
--2015-05-26 21:43:40-- http://pecl.php.net/get/libevent-0.1.0.tgz
Resolving pecl.php.net... 104.236.228.160
Connecting to pecl.php.net|104.236.228.160|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9806 (9.6K) [application/octet-stream]
Saving to: "libevent-0.1.0.tgz"

100%[=====>] 9,806
 41.4K/s in 0.2s
```

## 2、解压扩展文件压缩包

```
~# tar -zxvf libevent-0.1.0.tgz
package.xml
libevent-0.1.0/config.m4
libevent-0.1.0/CREDITS
libevent-0.1.0/libevent.c
....
```

## 3、进入到源码目录

```
~# cd libevent-0.1.0/
```

## 4、运行 phpize 命令

```
~# phpize
Configuring for:
PHP Api Version: 20090626
Zend Module Api No: 20090626
Zend Extension Api No: 220090626
```

## 5、运行 configure 命令

```
~# ./configure
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for a sed that does not truncate output... /bin/sed
checking for cc... cc
checking whether the C compiler works... yes
...
```

## 6、运行 make 命令

```
~# /bin/bash /data/test/libevent-0.1.0/libtool --mode=compile cc -I. -I/
data/test/libevent-0.1.0 -DPHP_ATOM_INC -I/data/test/libevent-0.1.0/inclu
de
...
```

## 7、运行 make install 命令

```
~# make install
Installing shared extensions: /usr/lib/php5/20090626/
```

## 8、配置ini文件

通过运行 `php --ini` 查找php.ini文件位置，然后在文件中添加 `extension=libevent.so`

# websocket协议

## WebSocket协议

目前Workerman的WebSocke协议版本为13

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信

## WebSocket与TCP关系

WebSocket和HTTP一样是一种应用层协议，都是基于TCP传输的，WebSocket本身和Socket并没有多大关系，更不能等同。

## WebSocket协议握手

WebSocket协议有一个握手的过程，握手时浏览器和服务端是以HTTP协议通信的，在Workerman中可以这样介入到握手过程。

```
...
$ws = new Worker('websocket://0.0.0.0:8181');
$ws->onConnect = function($connection)
{
 $connection->onWebSocketConnect = function($connection , $http_header
)
 {
 // 可以在这里判断连接来源是否合法，不合法就关掉连接
 // $_SERVER['HTTP_ORIGIN']标识来自哪个站点的页面发起的websocket连接
 if($_SERVER['HTTP_ORIGIN'] != 'http://chat.workerman.net')
 {
 $connection->close();
 }
 // onWebSocketConnect 里面$_GET $_SERVER是可用的
 // var_dump($_GET, $_SERVER);
 };
};
```

## WebSocket协议传输二进制数据

websocket协议默认只能传输utf8文本，如果要传输二进制数据，请阅读以下部分。

websocket协议中在协议头中使用一个标记位来标记传输的是二进制数据还是utf8文本数据，浏览器会验证标记和传输的内容类型是否符合，如果不符合则会报错断开连接。

所以服务端发送数据的时候需要根据传输的数据类型设置这个标记位，在Workerman中如果是普通utf8文本，则需要设置（默认就是此值，一般不用再手动设置）

```
use Workerman\Protocols\WebSocket;
$connection->websocketType = WebSocket::BINARY_TYPE_BLOB;
```

如果是二进制数据，则需要设置

```
use Workerman\Protocols\WebSocket;
$connection->websocketType = WebSocket::BINARY_TYPE_ARRAYBUFFER;
```

注意：如果没设置\$connection->websocketType，则\$connection->websocketType默认为BINARY\_TYPE\_BLOB（也就是utf8文本类型）。一般应用传输的都是utf8文本，例如传输的是json数据，所以不用手动设置\$connection->websocketType。只有在传输二进制数据时（例如图片数据、protobuf数据等）才要设置此属性为BINARY\_TYPE\_ARRAYBUFFER。

## 把workerman作为WebSocket客户端

可以利用[AsyncTcpConnection](#)配合[ws协议](#)让workerman作为websocket客户端连接远程websocket服务端，完成双向实时通讯。

# ws协议

## ws协议

目前Workerman的ws协议版本为13

workerman可以作为客户端通过ws协议发起websocket连接，连到远程websocket服务器，实现双向通讯。

注意：ws协议只能通过AsyncTcpConnection作为客户端使用，不能作为websocket服务端监听协议。也就是说以下写法是错误的。

```
$worker = new Worker('ws://0.0.0.0:8080');
```

如果想workerman作为websocket服务端，请使用[websocket协议](#)。

ws作为websocket客户端协议示例：

```
use Workerman\Worker;
use Workerman\Connection\AsyncTcpConnection;
require_once __DIR__ . '/Workerman/Autoloader.php';
$worker = new Worker();
// 进程启动时
$worker->onWorkerStart = function()
{
 // 以websocket协议连接远程websocket服务器
 $ws_connection = new AsyncTcpConnection("ws://echo.websocket.org:80")
;
 // 连上后发送hello字符串
 $ws_connection->onConnect = function($connection){
 $connection->send('hello');
 };
 // 远程websocket服务器发来消息时
 $ws_connection->onMessage = function($connection, $data){
 echo "recv: $data\n";
 };
 // 连接上发生错误时，一般是连接远程websocket服务器失败错误
 $ws_connection->onError = function($connection, $code, $msg){
 echo "error: $msg\n";
 };
 // 当连接远程websocket服务器的连接断开时
 $ws_connection->onClose = function($connection){
 echo "connection closed\n";
 };
 // 设置好以上各种回调后，执行连接操作
 $ws_connection->connect();
};
```

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

# text协议

## text协议

Workerman定义了一种叫做text的文本协议，协议格式为 ``数据包+换行符``，即在每个数据包末尾加上一个换行符表示包的结束。

例如下面的buffer1和buffer2字符串符合text协议

```
// 文本加一个回车
$buffer1 = 'abcdefghijklmn'
';
// 在php中双引号中的\n代表一个换行符，例如"\n"
$buffer2 = '{"type":"say", "content":"hello"}'."\n";

// 与服务端建立socket连接
$client = new stream_socket_client('tcp://127.0.0.1:5678');
// 以text协议发送buffer1数据
fwrite($client, $buffer1);
// 以text协议发送buffer2数据
fwrite($client, $buffer2);
```

text协议非常简单易用，如果开发者需要一个属于自己的协议，例如与手机App传输数据或者与硬件通讯等等，可以考虑使用text协议，开发调试都非常方便。

### text协议调试

text协议可以使用telnet客户端调试，例如下面的例子。

### test.php

```
require_once __DIR__ . '/Workerman/Autoloader.php';
use Workerman\Worker;

$text_worker = new Worker("text://0.0.0.0:5678");

$text_worker->onMessage = function($connection, $data)
{
 var_dump($data);
 $connection->send("hello world");
};

Worker::runAll();
```



执行 `php test.php start` 显示如下

```
php test.php start
Workerman[test.php] start in DEBUG mode
----- WORKERMAN -----
Workerman version:3.2.7 PHP version:5.4.37
----- WORKERS -----
user worker listen processes stat
us
root none myTextProtocol://0.0.0.0:5678 1 [OK
]

Press Ctrl-C to quit. Start success.
```

重新打开一个终端，利用telnet测试（建议用linux系统的telnet）

假设是本地测试，

终端执行 `telnet 127.0.0.1 5678`

然后输入 `hi`回车

会接收到数据`hello world\n`

```
telnet 127.0.0.1 5678
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hi
hello world
```

# frame协议

## frame协议

workerman定义了一种叫做frame的协议，协议格式为 ``总包长+包体``，其中包长为4字节网络字节序的整数，包体可以是普通文本或者二进制数据。

以下是frame协议实现。

```
```php
class Frame
{
    public static function input($buffer ,TcpConnection $connection)
    {
        if(strlen($buffer)<4)
        {
            return 0;
        }
        $unpack_data = unpack('Ntotal_length', $buffer);
        return $unpack_data['total_length'];
    }

    public static function decode($buffer)
    {
        return substr($buffer, 4);
    }

    public static function encode($buffer)
    {
        $total_length = 4 + strlen($buffer);
        return pack('N',$total_length) . $buffer;
    }
}
```

不支持的函数/特性

不支持的函数

不支持的函数/语句	替代方案	说明
php://input	\$GLOBALS['HTTP_RAW_POST_DATA']	用于HTTP协议下的应用获取POST的原始数据
exit	return	使用exit会导致进程退出，如果要返回请直接用return语句
die	return	使用die会导致进程退出，如果要返回请直接用return语句
header	Workerman\Protocols\Http::header	header函数只能用于http协议，其他协议不支持header

版权信息

版权信息

Copyright © 2013 - 2015 , workerman.net 所有。workerman开发者和使用者需要服从[workerman许可协议](#)。