

Table of Contents

1. [Introduction](#)
2. [版权信息](#)
3. [WorkerMan手册](#)
 - i. [序言](#)
4. [入门指引](#)
 - i. [简介](#)
 - ii. [WorkerMan的特性](#)
5. [开发前必读](#)
6. [流程与规范](#)
 - i. [目录结构](#)
 - ii. [系统流程](#)
 - iii. [开发流程](#)
 - iv. [相关规范](#)
7. [安装与配置](#)
 - i. [环境要求](#)
 - ii. [安装](#)
 - iii. [启动与停止](#)
 - iv. [配置规范](#)
 - v. [主配置说明](#)
 - vi. [应用配置说明](#)
 - vii. [FileMonitor.conf](#)
 - viii. [Monitor.conf](#)
 - ix. [Statistics统计模块](#)
8. [基础模型开发流程](#)
 - i. [制定协议](#)
 - ii. [实现dealInput/dealProcess](#)
 - iii. [配置与启动](#)
 - iv. [基本流程开发示例](#)
9. [Gateway/Worker开发流程](#)
 - i. [基本流程](#)
 - ii. [快速入门](#)
 - iii. [配置与启动](#)
 - iv. [Gateway/Worker开发示例](#)
 - v. [Config/Store 配置](#)
10. [类函数等参考](#)
 - i. [SocketWorker类](#)
 - i. [onStart](#)
 - ii. [dealInput](#)
 - iii. [dealProcess](#)
 - iv. [onStop](#)
 - v. [onReload](#)
 - vi. [sendToClient](#)
 - vii. [getRemoteAddress](#)
 - viii. [closeClient](#)
 - ii. [Gateway类](#)

- i. [sendToAll](#)
 - ii. [sendToClient](#)
 - iii. [sendToCurrentClient](#)
 - iv. [kickClient](#)
 - v. [kickCurrentClient](#)
 - vi. [isOnline](#)
 - vii. [getOnlineStatus](#)
- iii. [Event类](#)
 - i. [onGatewayConnect](#)
 - ii. [onGatewayMessage](#)
 - iii. [onMessage](#)
 - iv. [onClose](#)
- iv. [超全局数组\\$_SESSION](#)
 - i. [什么是超全局数组\\$_SESSION](#)
 - ii. [使用场景及注意事项](#)
 - iii. [超全局数组\\$_SESSION原理](#)
- v. [超全局数组\\$_SERVER](#)
 - i. [什么是超全局数组\\$_SERVER](#)
 - ii. [使用场景及注意事项](#)
 - iii. [超全局数组\\$_SERVER原理](#)
- 11. [调试](#)
 - i. [基本调试](#)
 - ii. [网络抓包](#)
 - iii. [跟踪系统调用](#)
- 12. [安全](#)
- 13. [高级应用](#)
 - i. [查看运行状态](#)
 - ii. [telnet远程登录控制](#)
 - iii. [分布式部署](#)
 - iv. [心跳检测](#)
- 14. [统计监控系统](#)
 - i. [关于统计监控系统](#)
 - ii. [统计监控系统特点](#)
 - iii. [原理](#)
 - iv. [如何使用](#)
 - v. [权限验证](#)
- 15. [压力测试](#)
- 16. [常见问题](#)
 - i. [WorkerMan下扩展的安装](#)
 - ii. [WorkerMan日志](#)
- 17. [WorkerMan进程模型](#)
 - i. [master-worker模型](#)
 - ii. [master-gateway-worker模型](#)
- 18. [一些开发示例](#)
 - i. [文件传输\(二进制协议\)](#)
 - ii. [文件传输\(文本协议\)](#)
 - iii. [Mysql数据库的使用](#)
 - iv. [p2p之UDP打洞](#)

19. 待续...

WorkerMan手册

- [版权信息](#)
- [WorkerMan手册](#)
 - [序言](#)
- [入门指引](#)
 - [简介](#)
 - [WorkerMan的特性](#)
- [开发前必读](#)
- [流程与规范](#)
 - [目录结构](#)
 - [系统流程](#)
 - [开发流程](#)
 - [相关规范](#)
- [安装与配置](#)
 - [环境要求](#)
 - [安装](#)
 - [启动与停止](#)
 - [配置规范](#)
 - [主配置说明](#)
 - [应用配置说明](#)
 - [FileMonitor.conf](#)
 - [Monitor.conf](#)
 - [Statistics统计模块](#)
- [基础模型开发流程](#)
 - [制定协议](#)
 - [实现dealInput/dealProcess](#)
 - [配置与启动](#)
 - [基本流程开发示例](#)
- [Gateway/Worker开发流程](#)
 - [基本流程](#)
 - [快速入门](#)
 - [配置与启动](#)
 - [Gateway/Worker开发示例](#)
 - [Config/Store 配置](#)
- [类函数等参考](#)
 - [SocketWorker类](#)
 - [onStart](#)
 - [dealInput](#)
 - [dealProcess](#)
 - [onStop](#)
 - [onReload](#)
 - [sendToClient](#)
 - [getRemoteAddress](#)
 - [closeClient](#)
 - [Gateway类](#)
 - [sendToAll](#)

- [sendToClient](#)
 - [sendToCurrentClient](#)
 - [kickClient](#)
 - [kickCurrentClient](#)
 - [isOnline](#)
 - [getOnlineStatus](#)
- [Event类](#)
 - [onGatewayConnect](#)
 - [onGatewayMessage](#)
 - [onMessage](#)
 - [onClose](#)
- [超全局数组\\$_SESSION](#)
 - [什么是超全局数组\\$_SESSION](#)
 - [使用场景及注意事项](#)
 - [超全局数组\\$_SESSION原理](#)
- [超全局数组\\$_SERVER](#)
 - [什么是超全局数组\\$_SERVER](#)
 - [使用场景及注意事项](#)
 - [超全局数组\\$_SERVER原理](#)
- [调试](#)
 - [基本调试](#)
 - [网络抓包](#)
 - [跟踪系统调用](#)
- [安全](#)
- [高级应用](#)
 - [查看运行状态](#)
 - [telnet远程登录控制](#)
 - [分布式部署](#)
 - [心跳检测](#)
- [统计监控系统](#)
 - [关于统计监控系统](#)
 - [统计监控系统特点](#)
 - [原理](#)
 - [如何使用](#)
 - [权限验证](#)
- [压力测试](#)
- [常见问题](#)
 - [WorkerMan下扩展的安装](#)
 - [WorkerMan日志](#)
- [WorkerMan进程模型](#)
 - [master-worker模型](#)
 - [master-gateway-worker模型](#)
- [一些开发示例](#)
 - [文件传输\(二进制协议\)](#)
 - [文件传输\(文本协议\)](#)
 - [Mysql数据库的使用](#)
- [待续....](#)

版权信息

Copyright © 2013 - 2014 , workerman.net 所有。workerman开发者和使用者需要服从 [workerman许可协议](#)。

WorkerMan手册

- [序言](#)

序言

PHP是一种被广泛应用的开源通用脚本语言，绝大多数开发者使用PHP做基于Web的应用程序，并且有了很多非常知名的Web框架，如Yii、thinkphp等。

传统的PHP应用程序基本上是基于HTTP协议开发的，但是在实际项目中通信协议并不一直都是HTTP的，例如基于websocket的聊天室、即时通讯的移动应用、需要长链接的游戏服务器开发、和硬件打印机传感器等的通信等等，开发这些应用程序我们无法直接使用nginx、apache、php-fpm来实现，也更无法使用传统的PHP框架来做。这就迫使我们寻找一种新的解决方案，这时候WorkerMan就是你的最佳选择。

1、入门指引

- [简介](#)
- [WorkerMan的特性](#)

简介

WorkerMan是什么

workerman是一个高性能的PHP Socket服务器框架，它类似[PHP-FPM](#)，提供进程控制及socket通讯功能，区别是[PHP-FPM](#)是以[FAST-CGI](#)的协议对外提供服务的，而workerman却可以支持各种协议（包括自定义协议），并且支持长链接，支持进程内全局对象资源等永久保持等特性。

WorkerMan能做什么

虽然workerman可以作为Webserver的替代Nginx PHP-FPM等架构，并且性能也比Nginx PHP-FPM高，但是我们不推荐这样做，因为PHP的WebServer市场上已经很成熟了，workerman不会再去做重复的事情。反而workerman把精力花在传统WebServer无法胜任的角色上，例如[非HTTP](#)协议的应用、TCP长链接应用、UDP应用、IM、游戏服务器、物联网等。

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退出信号，并负责生成新的子进程去处理服务，这样做不仅提高了应用程序的性能，而且使得WorkerMan更加稳定。

3、支持TCP、UDP

WorkerMan支持TCP和UDP两种传输层协议，只需要更改配置的一个字段，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，你只需要配置一个字段，便可以使用PHP长连接。PHP单个进程可以支持几千的并发连接，多进程支持几万的并发连接没有任何问题。

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。WorkerMan相关应用中已经用到的协议有统计监控系统（HTTP协议、自定义Statistic二进制协议）、WorkerMan-chat\WorkerMan-Todpole\WorkerMan-Flappy-Bird（Websocket协议）、WorkerMan-thrift-rcp（Thrift协议）、WorkerMan-json-rpc（自定义json协议）。以上应用中的协议可以拿来直接用，或者开发者选择使用自己的协议。

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。

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没有足够的进程对外提供服务，为了保证任意时刻都有足够的进程对外提供服务，则可以使用平滑重启命令，平滑重启过程中WorkerMan会让子进程处理完请求后才退出，并且能够保证在任意时刻都有足够的进程对外服务。

8、支持文件更新检测及自动加载

基于WorkerMan开发应用程序过程中，我们希望在我们改动代码后能够立刻生效，则只要你开启配置中

debug功能即可。开启后WorkerMan会有一个单独的进程轮询（因为PHP在Mac系统无法使用linux内核提供的inotify机制，所以只好轮询）应用程序真实使用的所有PHP文件，当文件有更新时，启动平滑重启，自动载入新的PHP文件。

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。

10、支持telnet远程控制及监控

WorkerMan内部带有监控统计模块，能够统计WorkerMan自身的一些数据，如进程退出次数及退出状态，每个进程占用内存大小及监听的ip端口、每个进程启动时间、进程运行的服务名、每个进程处理请求数、数据包错误量、数据包发送失败量等等。这些信息可以本地运行./bin/WorkerMan status本地查看或者通过telnet ip port(默认2000端口)远程获得。通过telnet，你还可以远程重启某个进程（kill命令）、平滑重启服务（reload命令）等。

11、支持子服务次数配置

可以设置进程服务次数，类似php-fpm配置中的max_requests配置，例如设置max_requests为1000，则每个子进程对外服务1000次后会安全退出。开启这个配置的目的是为了业务代码不规范导致的内存泄露给系统带来的内存占用过高的影响。

12、支持子进程监控

子进程监控包括内存占用监控（子进程内存超过配置Monitor.max_mem_limit时安全重启对应进程）、网络收发成功率监控、子进程FatalErr监控、子进程频繁退出监控。以上监控阈值均在conf/conf.d/Monitor.conf中配置，达到阈值时，便会触发告警（告警方法在workers/Monitor:sendSms()中实现）。

13、支持对象或者资源永久保持

在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，也就是只要初始化一次静态成员或者全局变量则整个进程生命周期内的所有请求都可以复用这个静态成员或者全局变量。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，不用每个用户请求都去重连数据库，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。memcache、redis等初始化也是同样的道理。

14、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通信的网络开销，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。

15、支持HHVM

支持在HHVM虚拟机上运行，可成倍提升PHP性能。

16、支持标准输入输出重定向

17、支持守护进程化

18、支持多端口监听

19、支持分布式部署

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
- 用WorkerMan开发非HTTP协议应用程序需要开发者自行解析应用层协议

由于非HTTP协议的应用，所以 `header()` `setcookie()` `session_start` 等函数无法直接使用

2、请求周期差异

- PHP在Web应用中一次请求过后会释放所有的变量与资源
- WorkerMan开发的应用程序在第一次请求过后会缓存编译后的文件，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文件在进程生命周期内都不会释放，所以要避免在每次请求中使用require/include重复载入类或者常量的定义。建议使用require_once/include_once载入类或者函数定义的文件

4、注意单例模式的链接资源的释放

- 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链接）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链接（避免重复链接非常高效），需要注意的是当数据库服务器发现某个链接在一定时间内（一般很长）没有活动后可能会主动关闭socket链接，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），并且有固定的错误码，这时开发者只需判断错误码并重连数据库即可（也可以在每次请求后自行删除实例）。

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

是一种面向连接（连接导向）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，

也有可能服务端收到的数据是多个请求连在一起。这就需要我们在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。例如一个简单的应用层次协议如下 `{"module": "user", "action": "getInfo", "uid": 456}\n`。此协议是以 `\n` 标记请求结束。

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。

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服务一般都用短链接。

短链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

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。长链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 `./workerman/bin/workermmand 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

三、两种开发模型

在WorkerMan中有两种开发模型，即：

1、基础开发模型

基础开发模型分为主进程（负责监控其它进程）、子进程（即负责与客户端通讯，也处理客户端的请求）。基础开发模型非常适合短连接应用，或者逻辑简单的(客户端之间不需要即时通讯)长连接应用。如果开发此类应用可参考基础模型开发流程开发。

2、Gateway/Worker开发模型

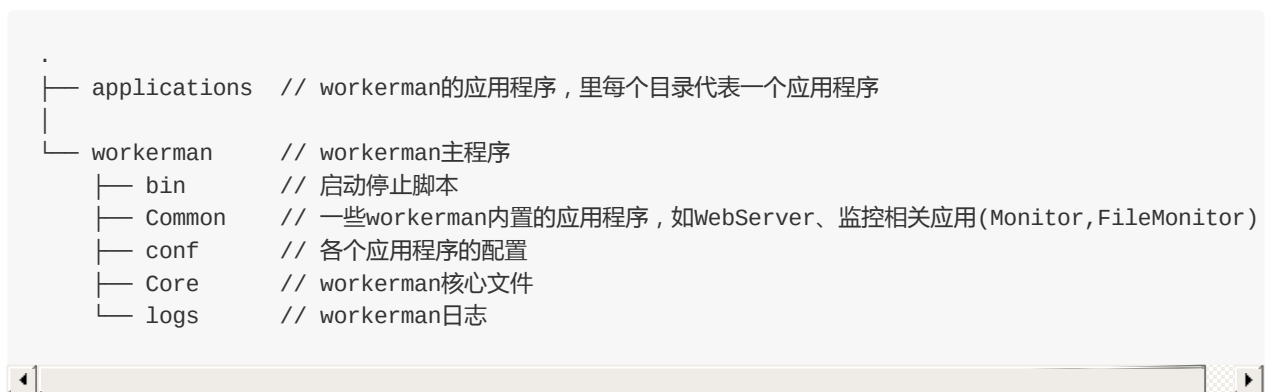
Gateway/Worker开发模型分为主进程（负责监控其它进程）、Gateway进程（负责与客户端通讯）、BusinessWorker进程（负责处理业务逻辑）。例如聊天室、网络游戏、移动通讯、物联网等需要客户端之间即时通讯类的应用。如果开发此类应用可以参考Gateway/Worker开发流程。

流程与规范

- [目录结构](#)
- [系统流程](#)
- [开发流程](#)
- [相关规范](#)

目录结构

```
.
├── applications  // workerman的应用程序，里每个目录代表一个应用程序
└── workerman    // workerman主程序
    ├── bin      // 启动停止脚本
    ├── Common   // 一些workerman内置的应用程序，如WebServer、监控相关应用(Monitor,FileMonitor)
    ├── conf     // 各个应用程序的配置
    ├── Core     // workerman核心文件
    └── logs     // workerman日志
```



系统流程

请求处理流程

- 1、客户端发来请求
- 2、WorkerMan读取请求数据交给dealInput(\$recv_buffer)函数
- 3、dealInput(\$recv_buffer)函数判断接收的请求数据\$recv_buffer是否是一个完整的请求
- 4、不完整则返回一个大于零的数字N，则WorkerMan会继等待N字节的请求数据
- 5、数据完整会自动调用dealProcess(\$recv_buffer)处理请求
- 6、请求处理完毕根据需要向客户端发送(sendToClient(\$response_buffer))数据

流程说明

- 1、为什么有个dealInput(\$recv_buffer)函数？作用是什么？

由于TCP是基于流的，也就是连接建立后可以一直不停的发送请求数据，发送的请求之间并没有明确的边界。在请求数据传输过程中由于很多原因（TCP分片、Socket缓冲区满等），请求的数据可能不会一次全部传输到WorkerMan，也有可能多个请求连在一起（粘包）同时到达WorkerMan。

dealInput就是WorkerMan留给开发者的一个钩子函数，开发者根据自己的协议实现dealInput,用来区分TCP流中每个请求边界，避免由于请求数据接收不全或者粘包导致请求处理异常。

WorkerMan正是通过这个钩子函数实现了请求边界的划分，并且通过这个函数支持各种应用层协议。

- 2、dealProcess(\$recv_buffer)的作用

当dealInput(\$recv_buffer)返回0代表请求接收完整,便进入请求处理流程

dealProcess(\$recv_buffer),dealProcess根据\$recv_buffer中的数据按照与客户端约定的协议规则解析出请求的具体内容进行相应的处理。

开发流程

基本开发流程

- 1、和客户端协定请求协议
- 2、实现dealInput，使得dealInput能够识别你的协议，并且能够分辨出请求边界
- 3、实现dealProcess，使得请求完整到达时，能够进行相应的处理。

详细流程参考后面基本开发流程一章

基于Gateway/Worker模型开发

WorkerMan中有一个例子Demo，这个例子虽然是个聊天的例子（通过 `telnet ip 8480` 测试），但是适合很多长连接相关的应用，例如移动APP通讯、游戏服务器、与硬件通讯等等。

如果你的应用是需要长连接，并且需要向客户端推送数据，那么可以直接使用Demo。

Gateway/Worker开发流程

- 1、和客户端协定请求协议
- 2、根据协议实现Demo/Event.php 中onGatewayMessage(\$recv_buffer)方法，用来判断请求边界，作用同dealInput(\$recv_buffer)。
- 4、实现Demo/Event.php 中的onMessage(\$client_id, \$recv_buffer)方法。当已经验证的客户端发来数据请求时触发，可以在这里做业务逻辑，比如向其他客户端发送消息(Gateway::sendToClient/sendToAll)
- 4、实现Demo/Event.php 中的 onClose方法。当客户端主动断开连接时触发此方法。一般在这里做一些状态记录如下线和数据清理工作。

详细流程参考后面Gateway/Worker开发流程一章

相关规范

应用程序目录

应用程序目录没有规定固定的位置，一般放在applications目录下，如applications/ChatApp

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，但是入口文件一般不叫index.php，而是和项目相关的有意义的名字，如ChatApp.php。入口文件需要满足以下约定：

- 1、应用入口文件必须继承 `Man\Core\SocketWorker` 类
- 2、入口文件名规定和文件里面类名一样，如 `class ChatApp extends Man\Core\SocketWorker` ,则入口文件的名字为ChatApp.php
- 3、入口文件必须实现 `Man\Core\SocketWorker` 中的 `dealInput` 、 `dealProcess` 方法

在Gateway/Worker模型中入口文件已经写好，开发者可以直接使用

配置文件

- 1、配置文件在 `workerman/conf/conf.d/` 下
- 2、配置文件名称一般与入口文件名相同并以conf为后缀，如 `workerman/conf/conf.d/ChatApp.conf`

GateWay/Worker模型中配置文件已经配置好，开发者可以直接使用，必要时可以做相应修改

安装与配置

- [环境要求](#)
- [安装](#)
- [启动与停止](#)
- [配置规范](#)
- [主配置说明](#)
- [应用配置说明](#)
- [FileMonitor.conf](#)
- [Monitor.conf](#)
- [Statistics统计模块](#)

环境要求

运行所需环境

- 1、WorkerMan 要求运行在Linux环境下 (centos、RedHat、Ubuntu、debian、mac os等)
- 2、安装有PHP-CLI(版本不低于5.3),并安装了pcntl、posix扩展
- 3、sysvshm、sysvmsg、libevent、proctitle扩展建议安装，但不是必须的

详细说明

关于PHP-CLI

WorkerMan是以PHP[命令行](#)的模式运行的，所以需要安装PHP-CLI,并且WorkerMan核心使用了命名空间等特性，所以需要PHP版本不小于5.3。

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、[sysvshm](#)、[sysvmsg](#)扩展

sysvshm([共享内存](#))、sysvmsg ([消息队列](#)) 扩展使得PHP在Linux环境下实现高效的进程间通信 ([IPC](#))，WorkerMan主要通过sysvshm实现进程间信息的共享（主要是主进程将全局信息如所有子进程进程pid共享给其它进程），使用sysvmsg实现进程间消息的传递(如子进程将自己使用的PHP文件通过消息队列传递给监控进程，一边监控进程监控文件更新自动加载)。

这些扩展不是必须安装，如果未安装会影响以下功能：文件监控及自动更新、WorkerMan状态查看(即workmand satus 命令不可用)。

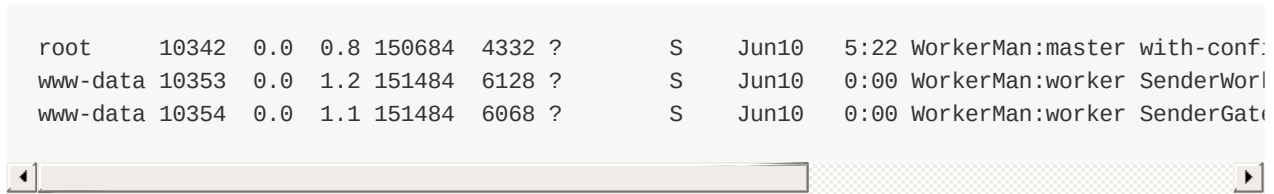
win平台不支持这些扩展。

4、[libevent](#)扩展

libevent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。

5、[proctitle](#)扩展

此扩展不是必须的。proctitle扩展使得WorkerMan可以更改进程的名称，在运行 ps aux 命令时比较有用，可以通过进程名称看到每个进程的比较详细的信息 类似下面



root	10342	0.0	0.8	150684	4332	?	S	Jun10	5:22	WorkerMan:master	with-conf:
www-data	10353	0.0	1.2	151484	6128	?	S	Jun10	0:00	WorkerMan:worker	SenderWorl
www-data	10354	0.0	1.1	151484	6068	?	S	Jun10	0:00	WorkerMan:worker	SenderGat

如何安装扩展

正常情况按照1.3安装方法即可满足WorkerMan所需环境。如果发现所需扩展没有安装可尝试以下方法：

如果您的php是源码编译

那么请进到php的源码目录，再进入ext目录下，分别找到相应的php模块目录，进行编译

- 1、假设php目录为/usr/local/php, 进到相应的php模块目录，执行 `/usr/local/php/bin/phpize`
- 2、接着执行 `./configure --with-php-config=/usr/local/php/bin/php-config`
- 3、接着执行以下命令 (没有权限则在命令前加sudo) `make && make install`
- 4、编译完成后，会显示so在哪个目录下，然后打开php.ini之后，在相应地方加入 `extension=xx.so`

centos系统并且PHP是yum安装

- 1、命令行运行 `yum install php-cli php-process git php-devel php-pear libevent-devel`
- 2、命令行运行 `pecl install channel://pecl.php.net/libevent-0.1.0`
- 3、命令行运行 `echo extension=libevent.so > /etc/php.d/libevent.ini`

debian/ubuntu并且PHP是apt-install安装

- 1、命令行运行 `apt-get update && apt-get install php5-cli git php-pear php5-dev libevent-dev`
- 2、命令行运行 `pecl install channel://pecl.php.net/libevent-0.1.0`
- 3、命令行运行 `echo extension=libevent.so > /etc/php5/cli/conf.d/libevent.ini`

版本说明

WorkerMan目前有两个版本，linux平台下的多进程版本和win平台下的[多线程版本](#)。Linux多进程版本是最稳定，功能最完善的版本，完全可用于生产环境。win平台多线程版本目前是测试版本，不建议用在生产环境。本手册适用于Linux多进程版本，在win多线程版本可能有所不同。

通过Github安装

centos系统安装教程

- 1、命令行运行 `yum install php-cli php-process git gcc php-devel php-pear libevent-devel`
- 2、命令行运行 `pecl install channel://pecl.php.net/libevent-0.1.0`
- 3、命令行运行 `echo extension=libevent.so > /etc/php.d/libevent.ini`
提示libevent installation [autodetect]: 时按回车
- 4、命令行运行 `git clone https://github.com/walkor/workerman`
- 5、命令行运行 `./workerman/workerman/bin/workerman start`

debian/ubuntu系统安装教程(如果不是root用户请用sudo 后面加命令)

- 1、命令行运行 `apt-get install php5-cli git gcc php-pear php5-dev libevent-dev`
- 2、命令行运行 `pecl install channel://pecl.php.net/libevent-0.1.0`
提示libevent installation [autodetect]: 时按回车
- 3、命令行运行 `echo extension=libevent.so > /etc/php5/cli/conf.d/libevent.ini`
- 4、命令行运行 `git clone https://github.com/walkor/workerman`
- 5、命令行运行 `./workerman/workerman/bin/workerman start`

下载ZIP文件安装

- 1、前提条件你本地安装了必要的运行环境，安装方法根据你的系统参考上面1-3步骤
- 2、通过<http://www.workerman.net/download/workermanzip> 连接下载WorkerMan
- 3、解压后命令行运行 `./workerman/workerman/bin/workerman start`

启动与停止

启动

```
workerman/bin/workermmand start
```

停止

```
workerman/bin/workermmand stop
```

重启

```
workerman/bin/workermmand restart
```

平滑重启

```
workerman/bin/workermmand reload
```

查看状态

```
workerman/bin/workermmand status
```

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。

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。

然而像聊天类的长连接应用中，势必要进程保存客户端的socket连接，那么在平滑重启时由于进程退出将导致客户端连接断开。为了避免这种情况，WorkerMan将像聊天类的长连接应用分为Gateway进程和BusinessWorker进程。Gateway进程负责接收客户端连接和请求数据，并将请求数据交给BusinessWorker

处理，BusinessWorker进程负责业务处理，并把处理结果转发给Gateway进程，由Gateway进程再转发给用户。这种Gateway BusinessWorker进程模型在业务代码更新时，其实只要平滑重启BusinessWorker进程即可，Gateway进程其实不用重启，所以一般在Gateway进程的配置文件中这是no_reload=1来避免平滑重启Gateway进程导致客户端连接断开。

配置规范

- 1、WorkerMan配置统一放在 `workerman/conf/` 下
- 2、其中 `workerman/conf/workerman.conf`是WorkerMan 主配置
- 3、应用程序配置放置于 `workerman/conf/conf.d/` 下,如 `workerman/conf/conf.d/ChatApp.conf`
- 4、建议配置的名称与入口文件的名称相同，后缀统一使用.conf
- 5、配置中涉及路径的可以用相对路径，也可以用绝对路径。其中相对路径是以workerman主程序目录为基准

主配置说明

主配置conf/workerman.conf

```
;debug=1则var_dump、echo、php notcie等会在终端上打印出来
debug=1
;保存主进程pid的文件
pid_file=/var/run/workerman.pid
;日志文件目录
log_dir=./logs/
;backlog
backlog=1024
;应用配置
include=/conf.d/*.conf
```

主配置说明

1、debug选项

debug=1，则在应用程序中 `echo var_dump` 等函数打印的数据都会在终端展示出来，方便调试。并且当有文件更新时，会自动加载更新的文件

debug=0，则应用程序中的 `echo var_dump` 等函数打印的数据不会在终端展示，并且当有文件更新时，也不会自动加载文件，要想加载文件需要手动运行 `workerman reload` 或者重启WorkerMan

建议开发环境debug=1,生产环境debug=0

2、pid_file选项

pid_file用于设置WorkerMan进程pid存放位置。WorkerMan启动后，会将主进程的pid保存到一个文件，以便workerman脚本运行停止、重启、平滑重启时能够得到主进程pid，然后向主进程发送相应的信号

如果你的服务器运行了多个WorkerMan，可以将多个WorkerMan的pid_file设置成不同的文件，避免workerman脚本操作了错误的WorkerMan主进程

3、log_dir选项

用于设置WorkerMan日志文件目录。WorkerMan日志会按照日期分割，每天——一个日志文件。日志文件记录的日志包括WorkerMan启动、停止、重启、平滑重启、以及一些异常日志如子进程异常退出。

WorkerMan日志会被WorkerMan定时清理

4、backlog 设置完成TCP三次握手等待被处理的客户端连接队列长度。此选项的值是全局默认的值，如果conf.d/下的配置中的有设置backlog，则对应的backlog将被替换。

5、include选项(预留选项，目前无作用)

用于设置应用程序配置目录。目前固定为/conf.d/*.conf

应用配置说明

一个典型的应用配置 conf/conf.d/Gateway.conf如下

```
;进程入口文件
worker_file = ../applications/Demo/Bootstrap/Gateway.php

;传输层协议及监听的ip端口
listen = tcp://0.0.0.0:8480

;backlog
backlog=1024

;是否是长连接
persistent_connection = 1

;开多少服务进程
start_workers = 5

;以哪个用户运行，为了安全，应该使用权限较低的用户，例如www-data nobody
user = www-data

;每个请求预读长度，避免读取数据超过一个协议包，
;一般设置为协议头的长度，当请求到来时在dealInput中根据头部标识的数据包长度计算还有多少数据没接收完毕，并
preread_length = 4

;接收缓冲区大小设置
max_recv_buffer_size = 10485760

;发送缓冲区大小设置
max_send_buffer_size = 20971520

;不reload，当有reload命令时是否安全重启这个进程
no_reload = 1

;workerman.conf.debug=1 时有效。echo var_dump 等输出是否打印到终端
no_debug = 1

;接收多少请求后重启进程，这个是gateway进程，不需要设置
;max_requests = 1000
```

1、worker_file选项

worker_file 用来设置应用程序入口文件。这个入口文件必须满足以下要求：

- 必须继承 `Man\Core\SocketWorker` 类并实现 `dealInput`、`dealProcess` 方法。
- 入口文件名与内部类名一样。

2、listen选项

listen用来设置应用程序传输层协议(tcp/udp)以及监听的ip与端口。

- TCP:如果应用程序使用tcp传输协议则设置类似listen=tcp://....

- UDP:如果应用程序使用udp传输协议则设置类似listen=udp://....
- ip:如果是设置为0.0.0.0,则是监听了所有网卡的ip,既可以可以通过该主机的内网ip也可以通过该主机的外网ip访问服务
- ip:如果设置为 xxx.xxx.xxx.xxx,则只能通过xxx.xxx.xxx.xxx访问服务
- 端口:端口范围一般为1 - 65535,其中1-1024一般是系统预留的常用端口,常用于系统服务等,例如http服务的端口号是80。所以端口范围建议一般为1025-65535。当然如果你清楚1-1024端口的意义是可以使用这些端口的,例如你确认服务器上没有apache、nginx之类的http服务完全可以使用80端口用WorkerMan对外提供http服务。

3、 backlog

设置完成TCP三次握手等待被处理的客户端连接队列长度。如果没有设置,则使用conf/workerman.conf.backlog

4、 persistent_connection选项

persistent_connection用来设置该应用是否是长连接应用。1为长连接,0为短连接。不设置默认为0

- 长连接:是指建立TCP连接后,会通过这个连接接收和发送多个请求。典型的长连接应用如聊天类应用、游戏应用、物联网等。
- 短连接:是指建立TCP连接后,客户端发送一个请求并得到服务器的回应后便断开了此链接,下次如果需要再次请求时则会再次建立TCP链接。典型的短连接应用如http服务等。

5、 start_workers选项

start_workers用来设置使用多少个进程来提供此服务。

start_workers一般设置规则需要参考cpu核数、应用程序类型、内存限制:。下面提供一个规则可供参考:

- 每个进程会占用大概40M内存,所以进程数*进程占用内存不能超过系统内存。
- 如果是CPU密集型应用,则建议设置为cpu核数的1-5倍
- 如果是IO密集型应用,则建议设置为cpu核数的5-10倍

6、 user选项

user选项用来设置应用进程运行用户。为了系统安全,建议使用权限较低的用户(如www-data nobody等)而不是root用户运行服务

7、 preread_length

preread_length用来设置一个新的请求到来时从socket缓冲区读取的数据长度。这个选项用来防止应用程序读取了大于一个请求长度的数据,导致处理这个请求异常。

由于TCP是基于流的,服务端收到的数据可能不是完整请求的也可能是多个请求连在一起。所以需要配置中的preread_length和应用程序的dealInput区分TCP流中的请求边界。

preread_length设置规则参考如下：

- 如果是短连接应用，因为一个连接只发一次请求，所以不会有多个请求连在一起到达服务端的情况，所以这个值可以设置一个较大值，例如65535
- 如果是长连接，则会有多个请求连在一起到达服务端的情况，这时需要设置为小于一个请求长度的值，这个值一般为你的应用层协议头的长度，通过读取协议头，我们能够得到当前请求的数据长度，就能区分出请求边界。

8、max_recv_buffer_size选项

max_recv_buffer_size用来设置应用单个用户的请求接收缓冲区大小(单位：字节)。这个选项主要是防止客户端发送了超大的请求数据包，导致服务进程内存耗尽风险。这个值默认是10M，也就是默认一次只能接受最大10M的大小的请求数据。假如你的服务是上传文件类的服务，文件大小可能大于10M，则可以更改此设置。

当客户端发来一个超大请求并且超过max_recv_buffer_size设置时，WorkerMan会断开此客户端的连接，并且记录日志

9、max_send_buffer_size选项

max_send_buffer_size用来设置应用单个用户发送缓冲区大小（单位：字节）。这个选项用来防止客户端网络阻塞等原因导致该用户的发送缓冲区越来越大造成服务端内存耗尽风险。这个值默认是20M，你可以更改成你期望的值。

当某个客户端的发送缓冲区大于max_send_buffer_size的大小时，WorkerMan会断开此客户端的连接，并记录日志

10、no_reload选项

no_reload选项影响该服务进程的文件更新检测以及服务平滑重启行为。

no_reload=1：当文件更新时，该进程不会重新载入文件，即该进程不会重启（WorkerMan是通过重启进程完成文件重新载入的）。

no_reload=1：当运行workerman reload时，该进程不会重启

在聊天类的应用中，进程一般分为gateway进程和businessWorker进程，gateway进程用来保持与客户端的连接，businessWorker进程用来处理业务。当业务更新时其实只要重启businessWorker进程便可以重新载入磁盘文件按完成业务更新，gateway进程由于维持着客户端连接，是不能重启的，否则会导致客户端断开。所以在gateway服务中经常会设置no_reload=1

11、no_debug选项

当workerman开启debug=1时，开发者不希望某个服务进程打印数据到终端时，可设置对应服务的no_debug=1

12、max_requests选项

max_requests用来设置对应进程接受多少用户请求后安全重启。此选项主要是为了避免业务代码有bug导致内存泄露而导致的内存耗尽，如果业务代码没有内存泄漏可以不设置此选项。

FileMonitor.conf

(WorkkerMan内置应用模块)

文件位置：workerman/conf/conf.d/FileMonitor.conf

作用：FileMonitor是Workerman内置的应用模块，用于监控workerman所使用文件的更新，如果文件有更新，并且conf/Workerman.conf中debug=1则自动运行reload，即平滑重启所有进程以便重新加载修改的php文件到内存

默认配置:

```
;进程入口文件
worker_file = Common/FileMonitor.php
;此worker进程不监听端口，主要用来监控文件更新，需要root权限
;listen=
;启动多少worker进程
start_workers=1
;以哪个用户运行该worker进程，需要root权限
user=root
;排除文件或者目录,这些文件或者目录下的文件将不会被监控，可以是相对路径或者绝对路径
exclude_path[]=./logs/
exclude_path[]=/tmp/
exclude_path[]=/path/example.php
```

注意：

有时文件有更新时我们并不希望自动运行reload命令重启子进程去更新文件，例如Gateway/Worker模型中当Store类的数据文件 `/tmp/xxx/gateway.store.cache.php`，这是我们可以把这个文件或者文件的目录加到FileMonitor的监控白名单中，例如FileMonitor.conf加上`exclude_path[]=/tmp/xxx/`代表将/tmp/xxx/下的所有文件加入到监控白名单中，也就是不监控这些文件的更新。

FileMonitor默认是开启的，如果不需要此模块，可以将此配置文件按删除或者移出

Monitor.conf

(WorkkerMan内置应用模块)

位置：workerman/conf/conf.d/Monitor.conf

作用：

- 1、监听2009端口（每个项目可能有所不同），并提供telnet远程控制功能
- 2、监控worker进程退出次数及状态，有异常时告警（告警发送邮件或者短信需要自己实现）
- 3、监控master进程是否异常退出
- 4、监控每个worker进程内存是否大于设定值，大于设定值则安全重启对应进程
- 5、提供workerman status 命令的统计数据

配置示例：

```
;入口文件
worker_file = Common/Monitor.php
;监听ip及端口，不使用的情况为了安全请绑定到127.0.0.1只限本机访问，如果绑定0.0.0.0，则记得更改下面telnet
listen = tcp://127.0.0.1:2009
;telnet需要长连接
persistent_connection = 1
;启动多少进程，1个就够
start_workers=1
;以哪个用户运行这个worker进程，需要root权限
user=root
;预读长度
preread_length=64

;==以下是自定义的配置==
;如果worker进程1分钟内退出max_worker_exit_count次则触发告警
max_worker_exit_count=2000
;worker进程最大内存阈值(单位KByte)，超过这个值安全重启(reload)这个进程
max_mem_limit=124000
;telnet密码
password=yourpassword
```

说明：

max_worker_exit_count：如果worker进程1分钟内退出max_worker_exit_count次则触发告警

max_mem_limit:子进程最大内存阈值(单位KByte)，如果某个子进程超过这个值则安全重启(reload)这个进程，以避免内存耗尽

password:远程登录时需要提供的密码

Statistics统计模块

WorkerMan的Gateway/Worker模型集成了统计模块，用来统计系统中各个模块接口的调用量、成功率、耗时、错误日志等，并以曲线表格的方式表现出来，方便监控系统运行情况及排查系统故障。

注意

Statistics统计模块是支持分布式的，比如在三台服务器上运行了WorkerMan，而这些服务器上也都运行了Statistics统计模块，则只是要通过浏览器登录任意一台服务器的查看统计数据的页面，就能查看到整个WorkerMan集群的运行状况（各个模块接口调用量、成功率、延迟等图表曲线），Statistics统计模块展示时会把你设置的数据源的服务器上的统计数据做汇总统计展示。

配置及作用

1、StatisticsWorker.conf StatisticsWorker进程负责将上报的数据整理并存储成本机数据文件，以便StatisticsProvider进程根据这些文件按对外提供统计数据查询服务。

```
进程入口文件
worker_file = ../applications/Statistics/Bootstrap/StatisticWorker.php
;监听ip及端口
listen = udp://0.0.0.0:55656
;启动多少进程，这里可以只启动一个，如果统计量很大（3000+/S）可以多开一些进程
start_workers=1
;以哪个用户运行这个worker进程，要设置成权限较低的用户，如www-data
user=root
;预读长度
preread_length=65507
```

2、StatisticsProvider.conf StatisticsProvider进程对外提供真正的数据查询服务

```
;提供查询统计信息数据接口
;进程入口文件
worker_file = ../applications/Statistics/Bootstrap/StatisticProvider.php
;监听ip及端口
listen = tcp://0.0.0.0:55858
;启动多少进程，只开1个
start_workers=1
;以哪个用户运行这个worker进，要设置成权限较低的用户
user=root
;预读长度
preread_length=65507
```

3、StatisticsWeb.conf

StatisticsWeb进程以HTTP协议对外提供查询服务，而数据的来源就是每台服务器上的StatisticsProvider进程提供的基础统计数据。也就是说StatisticsWeb进程可以将其它服务器上的统计数据通过运行在其它服务器上的StatisticsProvider进程汇总展示，从而实现分布式监控。

```
;进程入口文件
worker_file = ../Common/WebServer.php
```

```
;监听的端口
listen = tcp://0.0.0.0:55757
;http 协议 这里设置成短连接
persistent_connection = 0
;启动多少worker进程,只开一个
start_workers=1
;接收多少请求后退出
max_requests=1000
;以哪个用户运行该worker进程,要设置成权限较低的用户
user=root
;socket有数据可读的时候预读长度,一般设置为应用层协议包头的长度
preread_length=84000

;mime
include = ./Common/Protocols/Http/mime.types
;域名 统计服务
server_name = workerman.net
;统计服务根目录
root[workerman.net] = ../applications/Statistics/Web
```

基本模型开发流程

注意：如果你开发的应用是**短链接**应用，或者逻辑简单的长连接应用（客户端之间不需要通讯），可以参考此流程。

注意：如果是**长连接**应用（客户端之间需要通讯，例如IM聊天室、物联网、移动通讯等应用），建议直接使用Gateway/Worker模型开发（在Demo的基础上修改开发修改），官网的workerman-chat、workerman-todpole、workerman-flappy-bird等都是基于此模型开发的

基本开发流程

- [制定协议](#)
- [实现dealInput/dealProcess](#)
- [配置与启动](#)
- [基本流程开发示例](#)

制定协议

(WorkerMan>=2.0, 基础开发模型)

使用WorkerMan的第一步就是要和客户端商定应用层通信协议。

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，传统WebServer无法满足我们的需求，这时需要基于WorkerMan开发，WorkerMan无法预先知道开发者所用的协议，所以需要开发者实现协议解析部分。

如何制定协议？

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

1、区分数据边界的标识

作用

数据大小标识主要是为了区分出TCP流中的每个请求边界。由于TCP是基于流的，服务端收到客户端的请求可能是不完整的，也可能是多个请求连在一起的。

一个示例：

假如使用"\n"作为区分数据边界的标识，有两个请求"ABC\n" 和 "EFG\n"先后发送给服务端，由于TCP分片等机制，服务端获取的数据可能的情况有以下两种：

收到的数据是"AB"即不完整的

对于收到"AB"这种不完整的情况，应该继续等待剩余请求数据到来再处理这个请求（在dealInput中返回1，说明这个请求还要等待一个字节）

收到的数据是"ABC\nEFG\n"，多个请求连在一起的。

对于"ABC\nEFG\n"这样的请求，我们需要逐个字节读取（在dealInput中返回1，说明这个请求还要读取一个字节），直到读取到"\n"字符,然后处理这个请求。处理完毕后WorkerMan发现Socket缓冲区还有请求数据没读取，同样逐个字节读取，直到读取到"\n"后处理对应请求。

区分数据边界的标识没有固定的规定，例如可以用单个特殊字符放在请求数据结尾来标记，也可以在请求数据头部放置固定4个字节的pack过的int值来标记数据包长度，还可以用其它的方法。

2、数据格式定义

传输的内容是按照商定的能识别格式传递的。例如定义一个通信协议的数据格式是json，区分边界的标识

是"\n"，则一个请求的数据包可能是这样的 `{"module":"user","action":"get"}\n`。当服务端收到这个请求时，去掉请末尾"\n"，用`json_decode`便可以得到客户端传递实际请求。

几个示例协议

1、"\n"作为数据边界标识，数据格式为json

```
{"module":"user","action":"get"}\n
```

2、首部四个字节是以网络字节序pack的int值，标识请求数据包长度，数据格式为普通文本

```
$pack_len = pack('N', strlen('我是要请求的普通文本数据'));
```

```
$pack_len.'我是要请求的普通文本数据'
```

3、首部10个字节明文数字表示数据包长度，空白位以0代替，数据格式为xml

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

其中00000000121代表请求数据长度为121个字节，也就是整个请求的数据长度

实现dealInput/dealProcess接口

(WorkerMan>=2.0，基础开发模型)

dealInput

dealInput(\$recv_buffer)接口是WorkerMan给开发者的一个钩子函数，是专门用来处理请求边界问题的。

当Workeran收到请求后，会将之前收到的未处理的buffer加上这次收到的buffer拼接并通过\$recv_buffer传递给dealInput函数，让dealInput函数来判断\$recv_buffer是否是一个完整的请求数据，如果是完整的则返回0，如果不是则返回N，返回N代表还差N个字节请求数据才接收完整，则WorkerMan会继续等待/读取其余数据。如果 `N<0` 或者 `N===false` 则代表数据包出错，则当前链接会断开，这时WorkerMan状态统计中对应进程的packet_err会+1，WorkerMan状态可以通过 `./workerman/bin/workermmand status` 命令查看。

dealProcess

dealProcess(\$recv_buffer)接口是WorkerMan给开发者的一个钩子函数，当dealInput(\$recv_buffer)返回0即请求完整时被调用。一般在这里根据\$recv_buffer获得请求的具体数据，然后做相应的处理

几个示例，假如应用叫做MyApp

示例一、以 `\n` 为请求结束标记，以json为数据格式

如 `{"module":"user","action":"getInfo"}\n` 是一个请求，我们要能正确读取它，需要以下步骤：

1、设置配置文件preread_length=1，代表一个请求开始时，预先读取的数据1字节长度。为什么是1字节，因为协议规定以 `\n` 标记一个请求结束，有可能客户端就只发来一个 `\n` 字符,我们需要处理这种情况，固设置请求到来时先预读一个字符。

2、创建applications/MyApp/MyApp.php并实现dealInput/dealProcess如下

```
class MyApp extends Man\Core\SocketWorker
{
    public function dealInput($recv_buffer)
    {
        // 如果最后一个字符是\n代表数据读取完整，返回0
        if($recv_buffer[strlen($recv_buffer)-1] === "\n")
        {
            return 0;
        }

        // 说明还有请求数据没收到，但是由于不知道还有多少数据没收到，所以只能返回1，因为有可能下一个字符
        return 1;
    }

    public function dealProcess($recv_buffer)
    {
        // 去除末尾\n，得到完整json字符串
        $json_str = trim($recv_buffer);
        // 根据json字符串长解析出$req_data
        $req_data = json_decode(, true);
    }
}
```



```

        // 根据$req_data的值进入不同的处理逻辑
        // .....
    }
}

```

实例二、首部四个字节网络字节序pack的int+json数据格式

如客户端这样打包

```

// 请求的包体
$req_data = '{"module":"user","action":"getInfo"}';
// 整个请求数据长度，首部4字节+包体
$total_len = pack('N', strlen($req_data)+4)
$req_package = $total_len . $req_data;

```

得到类似这样的请求数据 *****{"module":"user","action":"getInfo"}，由于首部四个字节是pack的二进制数据，所以首部看起来是乱码。要想正确处理这样的请求，需要如下步骤：

- 1、设置配置文件中preread_length=4，因为通过请求头部的四个字节，我们能获取到整个数据包的长度，进而能够区分出请求的边界。
- 2、创建applications/MyApp/MyApp.php并实现dealInput/dealProcess如下

```

class MyApp extends Man\Core\SocketWorker
{
    public function dealInput($recv_buffer)
    {
        // 接收到的数据长度
        $recv_len = strlen($recv_buffer);
        // 如果接收的长度还不够四字节，那么要等够四字节才能解包到请求长度
        if($recv_len < 4)
        {
            // 不够四字节，等够四字节
            return 4 - $recv_len;
        }
        // 从请求数据头部解包出整体数据长度
        $unpack_data = unpack('Ntotal_len', $recv_buffer);
        $total_len = $unpack_data['total_len'];

        // 返回还差多少字节没有收到，这里可能返回0，代表请求接收完整
        return $total_len - $recv_len;
    }

    public function dealProcess($recv_buffer)
    {
        // 去掉首部四个字节，得到完整json字符串
        $json_str = substr($recv_buffer, 4);
        // 根据json字符串长解析出$req_data
        $req_data = json_decode($json_str, true);
        // 根据$req_data的值进入不同的处理逻辑
        // .....
    }
}

```

实例三、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

json

请求数据类似 0000000046{"module":"user","action":"getInfo"} ,其中 0000000046 代表整个请求 0000000046{"module":"user","action":"getInfo"} 的长度。要想正确处理这样的请求需要如下步骤：

1、设置preread_length=10，因为要完整读取前10个字节才能获取整个数据包的长度。

2、创建applications/MyApp/MyApp.php并实现dealInput/dealProcess如下

```
class MyApp extends Man\Core\SocketWorker
{
    public function dealInput($recv_buffer)
    {
        // 接收到的数据长度
        $recv_len = strlen($recv_buffer);
        // 如果接收的长度还不够十字节，那么要等够十字节才能解包到请求长度
        if($recv_len < 10)
        {
            // 不够十字节，等够十字节
            return 10 - $recv_len;
        }
        // 从请求数据头部解包出整体数据长度
        $json_str = substr($recv_buffer, 0, 10);
        $total_len = base_convert($json_str, 10, 10);

        // 返回还差多少字节没有收到，这里可能返回0，代表请求接收完整
        return $total_len - $recv_len;
    }

    public function dealProcess($recv_buffer)
    {
        // 去掉首部十个字节，得到完整json字符串
        $json_str = substr($recv_buffer, 10);
        // 根据json字符串长解析出$req_data
        $req_data = json_decode($json_str, true);
        // 根据$req_data的值进入不同的处理逻辑
        // .....
    }
}
```

当然还有其它非常多种类的协议....

配置

(WorkerMan>=2.0 , 基础开发模型)

workerman/conf/conf.d/MyApp.conf如下

```
;进程入口文件
worker_file = ../applications/MyApp/MyApp.php

;传输层协议及监听的ip端口
listen = tcp://0.0.0.0:8480

;是否是长连接, 根据自己的需要设置
persistent_connection = 1

;开多少服务进程, 根据自己的需要设置
start_workers = 5

;以哪个用户运行, 为了安全, 应该使用权限较低的用户, 例如www-data nobody
user = www-data

;每个请求预读长度, 假如使用\n作为请求结束边界
preread_length = 1

;接收缓冲区大小设置
max_recv_buffer_size = 10485760

;发送缓冲区大小设置
max_send_buffer_size = 20971520

;不reload, 当有reload命令时是否安全重启这个进程, 根据需要设置
no_reload = 1

;workerman.conf.debug=1 时有效。echo var_dump 等输出是否打印到终端, 根据需要设置
no_debug = 1

;接收多少请求后重启进程, 根据需要设置
;max_requests = 1000
```

详细配置说明参考配置章节

启动

运行命令: `workerman/bin/workermmand start`

应用开发实例（提供服务器信息）

（WorkerMan>=2.0，基础开发模型）

本示例功能

- 开放端口8181给外部
- 通过这个端口能够返回当前服务器日期
- 通过这个端口能够返回当前服务器负载
- 你可以返回其它数据...

本示例工作流程

- 1、telnet 到服务端8181端口
- 2、输入date回车，回显服务器日期
- 3、输入load回车，回显服务器负载
- 5、输入quit回车，断开链接

开发流程流程

- 1、订制协议 由于需要使用telnet客户端，则协议应该是 文本+回车
- 2、建立一个文件 applications/EchoService/EchoService.php 并实现dealInput、dealProcess如下

```
<?php
class EchoService extends \Man\Core\SocketWorker
{
    /**
     * 判断telnet客户端发来的数据是否接收完整
     */
    public function dealInput($recv_buffer)
    {
        // 根据协议,判断最后一个字符是不是回车 \n
        if($recv_buffer[strlen($recv_buffer)-1] != "\n")
        {
            // 不是回车返回1告诉workerman我还需要再读一个字符
            return 1;
        }
        // 告诉workerman数据完整了
        return 0;
    }

    /**
     * 处理业务逻辑,这里只是按照telnet客户端发来的命令返回对应的数据
     */
    public function dealProcess($recv_buffer)
    {
        // 判断telnet客户端发来的是什么
        $cmd = trim($recv_buffer);
        switch($cmd)
        {
```

```

        // 获得服务器的日期
        case 'date':
            return $this->sendToClient(date('Y-m-d H:i:s')."\n");
        // 获得服务器的负载
        case 'load':
            return $this->sendToClient(var_export(sys_getloadavg(), true)."\n");
        case 'quit':
            return $this->closeClient($this->currentDealFd);
        default:
            return $this->sendToClient("unknown cmd\n");
    }
}
}

```

3、配置与启动 建立配置文件 `workerman/conf/conf.d/EchoService.conf` ,内容如下

```

;监听ip及端口
;进程入口文件
worker_file = ../applications/EchoService/EchoService.php
;监听的端口
listen = tcp://0.0.0.0:8181
;是否是长连接
persistent_connection = 1
;启动多少进程
start_workers=4
;以哪个用户运行这个worker进程，应该用权限较低的用户如www-data nobody等
user=root
;预读长度，由于协议是 文本+回车 ，最小请求为 回车1个字符，这里设置为1
preread_length=1

```

启动workerman 运行 `./workerman/bin/workerman start`

4、测试（以telnet作为客户端）运行 telnet

```

telnet 127.0.0.1 8181
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
date
2014-08-22 15:50:17
load
array (
  0 => 1.83,
  1 => 1.69,
  2 => 1.65,
)
quit
Connection closed by foreign host.

```

php客户端

一般我们都是通过php去获取数据，就像在php中读取mysql数据一样。在php调用EchoService服务如下：

```

<?php
// 建立与服务端的链接

```

```
$socket = stream_socket_client("tcp://127.0.0.1:8181", $err_no, $err_msg);
if(!$socket)
{
    exit($err_msg);
}

// 设置为阻塞模式
stream_set_blocking($socket, true);

// 发送date命令
stream_socket_sendto($socket, "date\n");
// 读取服务端返回的数据
echo stream_socket_recvfrom($socket, 65535);

// 发送load命令
stream_socket_sendto($socket, "load\n");
// 读取服务端返回的数据
echo stream_socket_recvfrom($socket, 65535);

// 关闭链接
fclose($socket);
```

说明

本示例是一个简单的测试示例，是一个长链接应用（即telnet链接后通过这个链接发送多个请求），由于不需要telnet客户端之间传递数据，所有使用基本开发流程开发即可

如果是复杂的长链接应用，例如需要客户端之间通讯，建议使用Gateway/Worker开发流程，能够极大的减少开发工作量。例如 `applications/Demo` 是一个基于Gateway/Worker模型开发的聊天示例，支持群聊私聊，客户端同样是telnet

基于Gateway/Worker模型开发

Gateway/Worker模型一般用来开发长连接应用，例如聊天类、实时推送、即时游戏、与硬件实时通讯等。如果开发者需要开发这类应用，可以直接基于applications/Demo这个应用程序开发，applications/Demo是基于Gateway/Worker模型的一个基本应用框架。

什么是Gateway/Worker模型？

Gateway/Worker模型的应用程序由两组进程Gateway进程和Worker进程组成，两组进程互相配合，形成一个高性能长连接通讯框架。

Gateway进程

Gateway进程负责接受客户端的连接和连接上发来的请求数据，并转发Worker进程给客户端发送的数据。即Gateway进程无任何业务相关的逻辑，它只负责网络IO。

Worker进程（ BusinessWorker ）

负责接收Gateway进程转发来的请求数据，进行相应的处理，如果有需要，将结果通过Gateway进程转发给其它客户端。

为什么用Gateway/Worker进程模型？

- Gateway进程只负责网络IO，Worker进程只负责业务逻辑，各司其职，简单高效
- Gateway进程保存着客户端的连接，业务逻辑简单不易出错
- Worker进程是无状态的，即使单个请求因为业务出现致命错误也不会影响其他请求，更不会影响客户端连接
- 由于Worker进程无状态，则可以平滑重启业务进程，做到不影响用户的情况下完成代码热更新
- 由于Gateway进程和Worker进程是分开的，我们可以针对两种进程的负载情况动态扩容，甚至将Gateway Worker服务部署在多台服务器上，做到水平扩展

Gateway/Worker开发流程

- [基本流程](#)
- [快速入门](#)
- [配置与启动](#)
- [Gateway/Worker开发示例](#)
- [Config/Store 配置](#)

基本流程

(WorkerMan>=2.0 , Gateway/Worker模型)

基于Gateway/Worker模型的长链接应用开发者可以直接使用applications/Demo这个应用框架来做。

applications/Demo这个框架提供了以下基本功能。

- 1、维持客户端长链接
- 2、为每个客户端连接分配client_id标识
- 3、在服务端可以调用相关接口给其它客户端发送消息数据
- 4、提供了定时向客户端发送心跳的功能，用来检测与客户端的联通性
- 5、客户端可以是任意的，Demo中的客户端是telnet应用程序。即在终端运行 `telnet ip 8480` ,开启多个telnet窗口，每个窗口打字可以实时聊天

Demo的测试方法

- 终端运行 `telnet ip 8480` (ip为WorkerMan运行的服务器ip，本机的话可以使用127.0.0.1)
- 直接打字回车是向所有人发消息
- `$uid:xxx` 是向\$uid用户发送消息xxx，类似私聊
- 开启多个telnet窗口，窗口间可以实时聊天

基于applications/Demo开发流程

1、建立一个新的项目

1.1、选定项目名，例如项目名叫ChatRoom，更改 `applications/Demo` 为 `applications/ChatRoom`

1.2、将配置 `workerman/conf/conf.d/Gateway.php` 中的 `worker_file` 设置为新的路径 `worker_file=../applications/ChatRoom/Bootstrap/Gateway.php`

1.3、将配置 `workerman/conf/conf.d/BusinessWorker.conf` 中的 `worker_file` 设置为新的路径 `worker_file = ../applications/Demo/Bootstrap/BusinessWorker.php`

1.4、重新启动WorkerMan `workerman/bin/workerman restart`

1.5 至此，一个新的基于Gateway/Worker模型的项目建立好了

2、选定协议

applications/Demo中使用的默认应用层协议是文本请求末尾加回车，即 `Text+"\n"`,协议解析的文件是 `Procotols\TextProtocol.php`

下面是一个请求数据的样本：`Hi,大家好，我是小明\n`（注意末尾的\n是回车字符，代表一个请求的结束）

TextProtocol协议只是一个例子，开发者可以订制自己的协议，具体可参考上一章WorkerMan基本开发流程中的订制协议部分

3、通过接口实现相关业务逻辑

Gateway/Worker模型的业务逻辑入口全部在 `applications/ChatRoom/Event.php` 中。接口如下：

3.1、Event::onGatewayConnect()

当Gateway进程每收一个客户端链接时触发，如果你的应用需要在此时需要做些操作的话可以在这里实现。如果没有需要可以不实现这个方法。

3.2、Event::onGatewayMessage(\$recv_buffer)

此接口就是 Gateway 进程的 `dealInput` 函数，Gateway 用这个函数来区分TCP流中的请求边界。根据协议判断请求是否完整，`onGatewayMessage` 返回数字N，如果 `N=0`，代表 Gateway 进程当前的请求接收完整，紧接着 Gateway 进程会将客户端这个请求转发到 BusinessWorker 处理(`onConnect` 或者 `onMessage`)。 `onGatewayMessage` 的实现可以参考基本开发流程章节的实现 `dealInput/dealProcess` 小节中的 `dealInput` 部分

3.4、Event::onMessage(\$client_id, \$recv_message)

当WorkerMan接收到客户端发来的一个完整的请求时触发，`$client_id`是系统自动生成的（大于0的int整型），用来唯一标识客户端。在 `onMessage` 里面一般是根据协议解析请求并做处理，如果有需要通过 `Gateway::sendToAll/sendToClient` 向其它用户发送消息。

3.5、Event::onClose(\$client_id)

当客户端断开时触发，不管是客户端主动断开还是服务端主动断开都会触发。一般在这里清理用户的数据，例如更新数据库中的在线状态为下线

4、与客户端调试

调试除了在程序中打断点，还可以通过 `tcpdump` 等命令抓取网络的请求来判断网络请求是否整正确。见调试章节

5、发布

Gateway/Worker模型开发快速入门

(WorkerMan>=2.0 , Gateway/Worker模型)

Gateway/Worker模型开发者只需要关注一个文件 `Event.php` 即可

applications/XXX/Event.php

Event.php中几乎包含了所有的你需要关注的内容。你需要关注：

1、当监测到客户端有请求数据到来时我们要做什么

当客户端发来一个请求消息时，Gateway进程最先探测到有数据流入，这时会调用 `Event::onGatewayMessage($recv_buffer)` 方法区分数据流中的请求边界，所以开发者最先要实现 `Event::onGatewayMessage($recv_buffer)` 方法。其实 `Event::onGatewayMessage($recv_buffer)` 方法就是Gateway进程的dealInput函数，请参考[基本开发流程](#)中[制定协议](#)部分及[dealInput实现](#)部分实现 `Event::onGatewayMessage($recv_buffer)`

2、当接收到一个完整的请求时我们要做什么

当接收一个完整的请求时，`Event::onMessage($client_id, $recv_buffer)` 方法会被自动触发，其中 `$recv_buffer` 就是客户端发送的消息，如果这个消息是json字符串，则就可以用 `json_decode` 解码出请求内容，然后做相应的处理。而 `$client_id` 是全局唯一的，用来标识当前客户端，每个客户端在连接那一刻框架自动为其分配了一个全局唯一的 `$client_id`，可以说这个 `client_id` 就是客户端的身份证（直到这个客户端断开后才失效），给客户端发送消息时就需要这个身份证，例如调用向某一个客户端发送消息就可以用 `Gateway::sendToClient($client_id, $buffer)`，向某些客户端发送消息就用 `Gateway::sendToAll($buffer, $client_id_array)`，如果 `$client_id_array` 为空则是向所有客户端发送消息。

3、如果客户端断开是我们要做什么

如果客户端断开，不管是客户端主动断开还是服务端主动断开（踢出用户），都会触发 `Event::onClose($client_id)`，如果有需要可以在这里做一些数据清理工作，或者这里什么也不做。

以上便是Event.php的全部内容，而开发者实现了上面3个方法后便开发出了自己所要的网络服务

一个Event.php实现实例

```
// 协议为 文本+回车
class Event
{
    /**
     * 网关有消息时，区分请求边界，判断消息是否完整
     */
    public static function onGatewayMessage($buffer)
    {
        // 判断最后一个字符是否是回车("\n")
        if($buffer[strlen($buffer)-1] === "\n")
```

```

    {
        return 0;
    }

    // 说明还有请求数据没收到，但是由于不知道还有多少数据没收到，所以只能返回1，因为有可能下一个字符
    return 1;
}

/**
 * 有消息时触发该方法
 * @param int $client_id 发消息的client_id
 * @param string $message 消息
 * @return void
 */
public static function onMessage($client_id, $message)
{
    // 获得客户端发来的消息具体内容，trim去掉了请求末尾的回车
    $message_data = trim($message);

    // ****如果没有$_SESSION['not_first_time']说明是第一次发消息****
    if(empty($_SESSION['not_first_time']))
    {
        $_SESSION['not_first_time'] = true;

        // 广播所有用户，xxx come
        Gateway::sendToAll("client_id:$client_id come\n");
    }

    // 向所有人转发消息
    return Gateway::sendToAll("client[$client_id] said :". $message));
}

/**
 * 当用户断开连接时触发的方法
 * @param integer $client_id 断开连接的用户id
 * @return void
 */
public static function onClose($client_id)
{
    // 广播 xxx logout
    Gateway::sendToAll("client[$client_id] logout\n");
}
}

```

以上便完成了一个简单的聊天室，我们可以通过telnet命令使用它，运行 telnet ip port ,运行多个telnet窗口，则窗口之间可以互相聊天了

配置

(WorkerMan \geq 2.0 , Gateway/Worker模型)

Gateway/Worker模型的配置是现成的，在workerman/conf.conf.d/下的Gateway.conf和BusinessWorker.conf，可以直接复用。

启动

```
workerman/bin/workermmand restart
```

Gateway/Worker开发示例

(WorkerMan>=2.0 , Gateway/Worker模型)

待续...

Config/Store 配置

(WorkerMan>=2.0 , Gateway/Worker模型)

配置作用

配置Gateway/Worker模型中存储驱动类型及存储位置（文件位置或者memcache通讯ip端口）。以便Store类存储Gateway与BusinessWorker之间的通讯ip与端口，存储各个客户端的通讯ip与端口。

配置示例及说明

```
class Store
{
    // 使用文件存储，注意使用文件存储无法支持workerman分布式部署
    const DRIVER_FILE = 1;
    // 使用memcache存储，支持workerman分布式部署
    const DRIVER_MC = 2;

    /* 使用哪种存储驱动 文件存储DRIVER_FILE 或者 memcache存储DRIVER_MC，为了更好的性能请使用DRIVER_
    * 注意： DRIVER_FILE只适合开发环境，生产环境或者压测请使用DRIVER_MC，需要php cli 安装memcache扩展
    */
    public static $driver = self::DRIVER_FILE;

    // 如果是memcache存储，则在这里设置memcache的ip端口，注意确保你安装了memcache扩展
    public static $gateway = array(
        '127.0.0.1:22301',
    );

    public static $room = array(
        '127.0.0.1:22301',
    );

    /* 如果使用文件存储，则在这里设置数据存储的目录，默认/tmp/下
    * 注意：如果修改了storePath，要将storePath加入到conf/conf.d/FileMonitor.conf的忽略目录中
    * 例如 $storePath = '/home/data/'，则需要在conf/conf.d/FileMonitor.conf加一行 exclude_path
    */
    public static $storePath = '/tmp/workerman-chat/';
}
```

1、Store类有两种存储驱动，文件存储（DRIVER_FILE）及memcache存储(DRIVER_MC)。文件存储适合开发调试使用，memcache存储适合生产环境使用。

2、如果使用文件存储，存储文件会放置于 \$storePath 指定的路径下。如果运行多个Gateway/Worker模型的项目，请确保多个项目之间的 \$storePath 路径不要冲突。如果 \$storePath 有修改，请确保将改路径加到conf/conf.d/FileMonitor.conf中的exclude_path[]中。

3、如果是memcache存储，请安装memcached服务端及php的memcached扩展。并设定 \$gateway \$room 中的ip和端口为memcache服务端的ip端口。

4、如果是memcache存储，并且业务想添加自己的存储实例，可以开启新的memcache服务端ip端口，然后添加一项配置。例如新增加一个user存储实例，memcache ip端口为192.168.1.2:11211，则只需要增加如下一项到\Config\Store类中

```
public static $user= array(  
    '127.0.0.1:11211',  
);
```

使用的时候可以这样使用

```
\Lib\Store::instance('user')->get/set..
```

请开发者注意

1、线上环境请使用memcache存储，即设置

```
public static $driver = self::DRIVER_MC
```

2、如果更改了 `public static $storePath` 配置，请将配置的路径加到 `conf/conf.d/FileMonitor.conf` 的监视排除列表中，避免该路径下的文件更新导致WorkerMan的自动reload。例如 `$storePath = '/home/data/';`，则需要`conf/conf.d/FileMonitor.conf`中需要有一项 `exclude_path[]=/home/data/`

3、开发者不要使用 `Store::instance('gateway');`、`Store::instance('room');`，也要避免业务新增配置memcache的ip端口与gateway或者room的配置相同，以免造成业务数据与框架数据冲突。

函数参考

- [SocketWorker类](#)
 - [SocketWorker::onStart](#)
 - [SocketWorker::dealInput](#)
 - [SocketWorker::dealProcess](#)
 - [SocketWorker::onStop](#)
 - [SocketWorker::onReload](#)
 - [SocketWorker::sendToClient](#)
 - [SocketWorker::getRemoteAddress](#)
 - [SocketWorker::closeClient](#)
- [Gateway类](#)
 - [Gateway::sendToAll](#)
 - [Gateway::sendToClient](#)
 - [Gateway::sendToCurrentClient](#)
 - [Gateway::kickClient](#)
 - [Gateway::kickCurrentClient](#)
 - [Gateway::isOnline](#)
 - [Gateway::getOnlineStatus](#)
- [Event类](#)
 - [Event::onGatewayConnect](#)
 - [Event::onGatewayMessage](#)
 - [Event::onMessage](#)
 - [Event::onClose](#)
- [超全局数组\\$_SESSION](#)
 - [什么是超全局数组\\$_SESSION](#)
 - [使用场景及注意事项](#)
 - [超全局数组\\$_SESSION原理](#)
- [超全局数组\\$_SERVER](#)
 - [什么是超全局数组\\$_SERVER](#)
 - [使用场景及注意事项](#)
 - [超全局数组\\$_SERVER原理](#)

SocketWorker类(基础开发模型)

(WorkerMan>=2.0 , 基础开发模型)

文件位置:

workerman/Core/SocketWorker.php

基于WorkerMan基础接口开发：

SocketWorker类提供了WorkerMan的基本接口，如果开发者想基于WorkerMan基础接口进行开发，入口文件需要继承SocketWorker类，并且必须实现 `dealInput`（请求接收）和 `dealProcess`（请求处理）方法，才能完成服务的开发。

基于Gateway/Worker模型开发：

在Gateway/Worker模型开发中（TCP长链接应用），开发者可以直接使用 `applications/Demo` 例子开发，`dealInput`和`dealProcess`等都应该实现，开发者只需要关注 `applications/Demo/Event.php` 即可

SocketWorker::onStart

(WorkerMan>=2.0 , 基础开发模型) 此方法不是必须实现的

说明:

```
void SocketWorker::onStart(void)
```

当子进程启动时触发，可以用来设置每个子进程启动后做的工作，例如初始化一个定时任务等。

注意：

- 整个进程生命周期只触发一次onStart，只对子进程有效
- 多进程时注意并发问题，例如多个进程同时操作一个文件

范例

```
/**
 * 进程启动时初始化一个定时任务
 * @see Man\Core.SocketWorker::onStart()
 */
protected function onStart()
{
    // 初始化任务
    \Man\Core\Lib\Task::init($this->event);

    // 每隔10秒清理一次磁盘上的日志文件 ( clearDiskLog函数的实现这里省略 )
    $time_interval = 10;
    \Man\Core\Lib\Task::add($time_interval, array($this, 'clearDiskLog'));
}
```

SocketWorker::dealInput

(WorkerMan>=2.0 , 基础开发模型)

说明:

```
mixed SocketWorker::dealInput(string $recv_buffer)
```

当客户端的socket链接上有数据可读时触发。该函数作用是从TCP请求流中判断请求的边界和支持各种协议。

由于TCP是基于流的，服务端收到的数据可能是不完整的，也可能是多个请求连在一起（粘包），所以要通过这个函数区分请求边界。

区分方法为根据协议约定去解析 `$recv_buffer`，如果 `$recv_buffer` 是一个完整的请求，则返回。如果不是一个完整的请求则返回 `N`，即再接收 `N` 字节的数据。如果 `$recv_buffer` 不符合协议格式可以返回 `false`，此时会主动断开与客户端的链接。

参数

`$recv_buffer`

客户端请求的数据，包含了之前接收到的不完整的请求数据。

返回值

返回值N

- `N>0`代表当前请求还有N字节数据未接收(N不可以无限大，如果大于conf中配置的 `max_recv_buffer_size`时会记录日志并断开链接)
- `N==0`代表请求接收完整，进入SocketWorker::dealProcess(\$recv_buffer)请求处理流程
- `N<0` 或者 `N===false` 代表协议出错，会断开与客户端的链接，并记录日志

注意：

`SocketWorker::dealInput` 函数的实现完全依赖与开发者所使用的应用层通信协

议。`SocketWorker::dealInput` 通常与conf配置中 `preread_length`配合使用。conf配置中

的 `preread_length` 配置是为了避免从客户端socket缓冲区内读取了大于一个请求的数据，也就是避免读取越界（在长链接应用中尤其重要）。一个**新的**请求数据到来时WorkerMan预先从客户端的socket缓冲区读取的数据长度，这个长度一定是小于等于**最小合法请求**的数据长度，一般设置为请求协议头的长度，然后通过请求协议头获得本次请求实际数据长度，然后 `SocketWorker::dealInput` 再返回 `N` 告诉WorkerMan 本次请求我还需要 `N` 字节的数据才是一个完整请求。

范例

1、自定义协议: json+回车

conf配置: preread_length=1, 最小合法请求为 \n, 即一个回车字符, 所以为1

协议样例: {"module":"user","action":"getInfo"}\n, 其中 \n 代表回车字符

```
public function dealInput($recv_buffer)
{
    // 如果最后一个字符是\n代表数据读取完整, 返回0
    if($recv_buffer[strlen($recv_buffer)-1] === "\n")
    {
        return 0;
    }

    // 说明还有请求数据没收到, 但是由于不知道还有多少数据没收到, 所以只能返回1, 因为有可能下一个字符就是\
    return 1;
}
```

2、自定义协议: 首部四字节int+json

其中首部四字节是一个pack的int, 代表请求的数据长度

conf配置: preread_length=4, 最小合法请求可能为值为0的四字节int (一个空请求), 所以设置为4

协议样例: ****{"module":"user","action":"getInfo"}, 其中 **** 首部四字节int, 由于是二进制, 所以首部四字节看起来是乱码

```
public function dealInput($recv_buffer)
{
    // 接收到的数据长度
    $recv_len = strlen($recv_buffer);
    // 如果接收的长度还不够四字节, 那么要等够四字节才能解包到请求长度
    if($recv_len < 4)
    {
        // 不够四字节, 等够四字节
        return 4 - $recv_len;
    }
    // 从请求数据头部解包出整体数据长度
    $unpack_data = unpack('Ntotal_len', $recv_buffer);
    $total_len = $unpack_data['total_len'];

    // 返回还差多少字节没有收到, 这里可能返回0, 代表请求接收完整
    return $total_len - $recv_len;
}
```

3、HTTP协议:

HTTP协议包含协议头与包体, 二者直接使用\r\n\r\n分割, POST请求协议头中包含 Content-Length 代表包体的长度。

请求样例:

```
GET / HTTP/1.1\r\n
Host: www.baidu.com\r\n
Connection: keep-alive\r\n
Cache-Control: max-age=0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0
Accept-Encoding: gzip,deflate,sdch\r\n
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6\r\n
Cookie: uc_login_unique=1fe1b19668b1419d8c45a6ac738fed76;\r\n
\r\n
```

配置: `preread_length=65535`，由于HTTP协议是短链接的，即一个链接上只发一个请求，所以不会有多个请求连在一起的粘包现象，也就是说不会有读取越界的情况，所以预读长度可以设置为很大的值。短链接请求中主要任务就是判断读取的数据是否是一个完整的请求

```
public function dealInput($recv_buffer)
{
    // 协议头不完整，再读一些数据
    if(!strpos($recv_buffer, "\r\n\r\n"))
    {
        return 65535;
    }

    // GET请求只有协议头，POST请求还要读包体，包体长度在协议头中的Content-Length中存储
    if(strpos($recv_buffer, "POST"))
    {
        // 找Content-Length
        $match = array();
        if(preg_match("/\r\nContent-Length: ?(\d*)\r\n/", $recv_buffer, $match))
        {
            $content_lenght = $match[1];
        }
        else
        {
            return 0;
        }
        // 看包体长度是否符合
        $tmp = explode("\r\n\r\n", $recv_buffer, 2);
        $remain_length = $content_lenght - strlen($tmp[1]);
        return $remain_length >= 0 ? $remain_length : 0;
    }

    return 0;
}
```

SocketWorker::dealProcess

(WorkerMan>=2.0 , 基础开发模型)

说明:

```
void SocketWorker::dealProcess(string $recv_buffer)
```

当 `SocketWorker::dealInput($recv_buffer)` 返回0，即收到一个完整请求时触发。此函数是业务处理函数，即根据请求内容做相应的处理。

参数

`$recv_buffer`

一个完整的客户端请求

范例

例如json+回车协议，一个请求 `{"type":"say_to_all", "content":"Hello everybody!"}\n`

```
public function dealProcess($recv_str)
{
    $req_data = json_decode(trim($recv_str));
    switch($req_data['type'])
    {
        case 'say_to_all':
            // say_to_all函数实现省略
            say_to_all($req_data['content']);
        case .....
    }
}
```

SocketWorker::onStop

(WorkerMan>=2.0 , 基础开发模型) 此方法不是必须实现的

说明:

```
void SocketWorker::onStop(void)
```

当子进程停止时触发, 可以用来设置每个子进程停止即将退出时的工作, 例如将内存中的数据保存到磁盘。

注意

- 整个进程生命周期只触发一次onStop, 只对子进程有效

以下情况将导致子进程停止并触发onStop函数

- 1、运行 workermmand reload , 全部进程都会收到reload信号, 如果对应进程conf没有配置no_reload , 会重启对应进程
- 2、开启workerman.conf.debug=1 , 并且子进程使用的文件在磁盘里有更新, 全部进程会收到reload信号, 如果对应进程conf没有配置no_reload , 会重启对应进程
- 3、telnet远程控制workerman , 运行 reload 命令, 全部进程会收到reload信号, , 如果对应进程conf没有配置no_reload , 会重启对应进程
- 4、telnet远程控制workerman , 运行 kill pid 命令, pid对应进程会收到reload信号, , 如果对应进程conf没有配置no_reload , 会重启对应进程
- 5、当前进程内存占用大于Monitor.conf.max_mem_limit 时当前进程会被重启
- 6、当运行workermmand stop 或者 workermmand restart时, 所有进程会被重启

范例

```
/**
 * 进程停止时需要将数据写入磁盘
 * @see Man\Core.SocketWorker::onStop()
 */
protected function onStop()
{
    // 将内存中的日志保存到磁盘(writeLogToDisk函数实现忽略)
    $this->writeLogToDisk();
}
```

SocketWorker::onReload

(WorkerMan>=2.0 , 基础开发模型) 此方法不是必须实现的

说明:

```
void SocketWorker::onReload(void)
```

当子进程收到reload信号时触发，reload信号主要是为了更新进程或者更新配置。

当对应子进程conf中没有配置no_realod时，收到reload信号默认操作是子进程自动退出。

当对应子进程conf中配置no_reload=1时，收到reload信号所做的操作在onReload实现

范例

conf配置中no_reload=1

```
/**
 * 收到reload信号后重新从磁盘加载配置文件
 * @see Man\Core.SocketWorker::onReload()
 */
protected function onReload()
{
    // 重新从磁盘加载配置文件(updateConfigFromDisk函数实现忽略)
    $this->updateConfigFromDisk();
}
```


SocketWorker::sendToClient

(WorkerMan>=2.0 , 基础开发模型) 此方法直接可以调用

说明:

```
mixed SocketWorker::sendToClient($buffer_so_send)
```

关闭客户端的连接

参数

`$buffer_so_send`

要发送的数据 (可以是二进制数据)

返回值

如果返回true , 说明发送成功 (成功写入到本地系统socket发送缓冲区) 。

如果返回null , 则说明没有发送成功 (可能是本地系统socket发送缓冲区已满) , 数据会放到WorkerMan自身的发送缓冲区中 , WorkerMan会异步等待数据可发送时(本地系统socket发送缓冲区可写)再发送。如果发送失败 , 则在WorkerMan统计 `send_fail` 会加一。使用 `workermmand status` 可以看到计数

如果返回false , 则代表发送失败。失败原因可能是超过WorkerMan发送缓冲区限制

(在 `max_send_buffer_size` 中配置 , 这种情况会有日志产生) ;可能是客户端已经关闭;发送的数超过了udp包的大小限制 (当配置listen指定的是udp传输层协议时)

范例

```
public function dealProcess($recv_buf)
{
    // send what client send
    $this->sendToClient($recv_buf);
}
```

SocketWorker::getRemoteAddress

(WorkerMan>=2.0 , 基础开发模型) 此方法直接可以调用

说明:

```
string SocketWorker::getRemoteAddress(void)
```

获取客户端ip及端口 (注意 : 当客户端处于一个封闭内网环境时 , 获取到的是客户端内网的出口ip及端口)

返回值

返回客户端ip及端口 , 例如返回 111.222.333.444:36540

范例

```
public function dealProcess($recv_buf)
{
    $recv_buf = trim($recv_buf);
    switch($recv_buf)
    {
        case 'tell_me_address':
            // $address will be xxx.xxx.xxx.xxx:xxx
            $address = $this->getRemoteAddress();
            $this->sendToClient($address);
            break;
        case 'tell_me_ip':
            // $ip will be xxx.xxx.xxx.xxx
            $ip = $this->getRemoteIp();
            $this->sendToClient($ip);
            break;
        case 'tel_me_port':
            // $port will be xxx
            $port = $this->getRemotePort();
            $this->sendToClient($port);
            break;
        default:
            $this->sendToClient('unknow commend');
    }
}
```

SocketWorker::closeClient

(WorkerMan>=2.0 , 基础开发模型) 此方法直接可以调用

说明:

```
void SocketWorker::closeClient([int $fd])
```

关闭客户端的连接

参数

`$fd`

如果传递了 `$fd` , 则将关闭 `$fd` 对应的客户端连接。否则将关闭当前处理的客户端连接

范例

```
public function dealProcess($recv_buf)
{
    $recv_buf = trim($recv_buf);
    switch($recv_buf)
    {
        case 'say_hello':
            $this->sendToClient('hello');
            break;
        case 'quit':
            $this->closeClient();
            break;
        default:
            $this->sendToClient('unknow commend');
    }
}
```

Gateway类 (基于Gateway/Worker模型开发)

(WorkerMan>=2.0 , Gateway/Worker模型)

文件位置:

applications/Demo/Lib/Gateway.php

作用 :

提供了与客户端通讯的基本接口

Gateway::sendToAll

(WorkerMan>=2.0 , Gateway/Worker模型)

说明:

```
void Gateway::sendToAll(string $send_buffer [, array $client_id_array=array()]);
```

向所有客户端或者指定的client_id_array客户端发送 \$send_buffer 数据。如果指定的\$client_id_array中的client_id不存在则自动丢弃对应的client的发送数据

参数

- \$send_buffer

要发送的数据（可能是二进制数据）

- \$client_id_array

(WorkerMan>=2.1.2)

指定向哪些client_id发送，如果不传递该参数或者为空，则是向所有在线客户端发送 \$send_buffer 数据

范例

```
use \Lib\Gateway;

class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        // $message = '{"type":"say_to_all","content":"hello"}'
        $req_data = json_decode(trim($message), true);
        // 如果是向所有客户端发送消息
        if($req_data['type'] == 'say_to_all')
        {
            // 向所有客户端发送数据
            Gateway::sendToAll($req_data['content']);
        }
    }

    ...
}
```

Gateway::sendToClient

(WorkerMan>=2.1.3 , Gateway/Worker模型)

说明:

```
void Gateway::sendToClient(int $client_id, string $send_buffer);
```

向客户端client_id发送 \$send_buffer 数据。如果client_id对应的客户端不存在或者不在线则自动丢弃发送数据

参数

- \$client_id

客户端的client_id，当客户端连接Gateway的那一刻框架便为其分配了一个全局唯一的client_id用来全局标识一个客户端连接。对某个客户端的操作都需要知道客户端的client_id

- \$send_buffer

要发送的数据（可能是二进制数据）

范例

```
use \Lib\Gateway;
class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        // $message = '{"type":"say_to_one","to_client_id":100,"content":"hello"}'
        $req_data = json_decode(trim($message), true);
        // 如果是向某个客户端发送消息
        if($req_data['type'] == 'say_to_one')
        {
            // 转发消息给对应的客户端
            Gateway::sendToClient($req_data['to_client_id'], $req_data['content']);
        }
    }

    ...
}
```

Gateway::sendToCurrentClient

(WorkerMan>=2.1.3 , Gateway/Worker模型)

说明:

```
void Gateway::sendToCurrentClient(string $send_buffer);
```

向当前客户端发送 \$send_buffer 数据。

参数

- \$send_buffer

要发送的数据（可能是二进制数据）

范例

```
use \Lib\Gateway;
class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        if($message == 'tell me the time')
        {
            // 向当前客户端发送数据
            Gateway::sendToCurrentClient(date('Y-m-d H:i:s'));
        }
    }

    ...
}
```

Gateway::kickClient

(WorkerMan>=2.1.3 , Gateway/Worker模型)

说明:

```
void Gateway::kickClient(int $client_id);
```

断开与client_id对应的客户端的连接，使对应的客户端下线

参数

- \$client_id

全局唯一标识客户端连接的id

范例

```
use \Lib\Gateway;

class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        // 如果传递的消息不ok就踢掉对应客户端
        $is_ok = your_check_fun($message);
        if(!$is_ok)
        {
            Gateway::kickClient($client_id);
        }
    }

    ...
}
```


Gateway::kickCurrentClient

(WorkerMan>=2.1.3 , Gateway/Worker模型)

说明:

```
void Gateway::kickCurrentClient();
```

踢掉当前socket连接，使当前的客户端下线

范例

```
use \Lib\Gateway;

class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        // 如果传递的消息不ok就踢掉对应客户端
        $is_ok = your_check_fun($message);
        if(!$is_ok)
        {
            Gateway::kickCurrentClient();
        }
    }

    ...
}
```

Gateway::isOnline

(WorkerMan>=2.1.2 , Gateway/Worker模型)

说明:

```
int Gateway::isOnline(int $client_id);
```

判断\$client_id是否还在线

参数

- \$client_id

全局唯一的客户端client_id

返回值

在线返回1，不在线返回0

范例

```
use \Lib\Gateway;
class Event
{
    ...

    public static function onMessage($client_id, $message)
    {
        // $message = '{"type":"say_to_one","to_client_id":100,"content":"hello"}'
        $req_data = json_decode(trim($message), true);
        // 如果是向某个客户端发送消息
        if($req_data['type'] == 'say_to_one'))
        {
            // 如果不在线就先存起来
            if(!Gateway::isOnline($req_data['to_client_id']))
            {
                your_store_fun($message);
            }
            else
            {
                // 在线就转发消息给对应的客户端
                Gateway::sendToClient($req_data['to_client_id'], $req_data['content']);
            }
        }
    }

    ...
}
```

Gateway::getOnlineStatus

(WorkerMan>=2.1.2 , Gateway/Worker模型)

说明:

```
array Gateway::getOnlineStatus(void);
```

获取当前所有在线client_id列表

范例

```
use \Lib\Gateway;  
  
// 打印在线client_id列表  
var_export(Gateway::getOnlineStatus());
```

打印出的数据类似如下：

```
array(  
    0=>1001,  
    1=>1009,  
    2=>99,  
);
```

Event类 (基于Gateway/Worker模型开发)

文件位置:

applications/Demo/Event.php

作用:

开发者通过实现这个类的方法完成对业务逻辑的实现，是开发者最需要关注的类

Event::onGatewayConnect

(WorkerMan>=2.0 , Gateway/Worker模型)

说明:

```
void Gateway::onGatewayConnect(int $client_id);
```

当客户端连接上gateway进程时触发。例如客户端连接到gateway进程后，服务端要向新连接的客户端发送一个token，客户端需要根据token计算登录的凭证，然后通过这个凭证向服务端发起登录请求。

绝大多数应用在不用实现这个函数。

参数

(WorkerMan>=2.1.3)

`$client_id` 全局唯一的客户端socket_id

范例

```
use \Lib\Gateway;

public onGatewayConnect($client)
{
    // 客户端一旦连接到服务端，服务端便向客户端发送一个token
    // 客户端根据这个token数据生成登录请求凭证
    $salt = '!@#$$%^';
    Gateway::sendToCurrentClient('token='.md5(time()).$salt));
}
```

Event::onGatewayMessage

(WorkerMan>=2.0 , Gateway/Worker模型)

说明:

```
mixed Gateway::onGatewayMessage(string $recv_buffer);
```

当Gateway进程收到数据的时候触发，其实就是Gateway进程的dealInput函数，用来判断TCP请求数据流中的请求边界。

由于TCP是基于流的，服务端收到的数据可能是不完整的，也可能是多个请求连在一起（粘包），所以要通过这个函数区分请求边界。

区分方法为根据协议约定去解析 \$recv_buffer，如果 \$recv_buffer 是一个完整的请求，则返回。如果不是一个完整的请求则返回 N，即再接收 N 字节的数据。如果 \$recv_buffer 不符合协议格式可以返回 false，此时会主动断开与客户端的连接。

参数

\$recv_buffer

客户端请求的数据，包含了之前接收到的不完整的请求数据。

返回值

返回值N

- N>0代表当前请求还有N字节数据未接收(N不可以无限大，如果大于Gateway.conf中配置的 max_recv_buffer_size时会记录日志并断开链接)
- N==0代表请求接收完整，进入SocketWorker::dealProcess(\$recv_buffer)请求处理流程
- N<0 或者 N===false 代表协议出错，会断开与客户端的连接，并记录日志

注意：

Gateway::onGatewayMessage 函数的实现完全依赖与开发者所使用的应用层通信协议。Gateway::onGatewayMessage 通常与Gateway.conf配置中 preread_length配合使用。Gateway.conf配置中的 preread_length 配置是为了避免从客户端socket缓冲区内读取了大于一个请求的数据，也就是避免读取越界（在长链接应用中尤其重要）。一个**新的**请求数据到来时WorkerMan预先从客户端的socket缓冲区读取的数据长度，这个长度一定是小于等于**最小合法请求**的数据长度，一般设置为请求协议头的长度，然后通过请求协议头获得本次请求实际数据长度，然后 Gateway::onGatewayMessage 再返回 N 告诉WorkerMan本次请求我还需要 N 字节的数据才是一个完整请求。

范例

1、自定义协议: json+回车

Gateway.conf配置: preread_length=1, 最小合法请求为 \n , 即一个回车字符, 所以为1

协议样例: {"module":"user","action":"getInfo"}\n ,其中 \n 代表回车字符

```
public function onGatewayMessage($recv_buffer)
{
    // 如果最后一个字符是\n代表数据读取完整, 返回0
    if($recv_buffer[strlen($recv_buffer)-1] === "\n")
    {
        return 0;
    }

    // 说明还有请求数据没收到, 但是由于不知道还有多少数据没收到, 所以只能返回1, 因为有可能下一个字符就是\
    return 1;
}
```

2、自定义协议: 首部四字节int+json

其中首部四字节是一个pack的int, 代表请求的数据长度

Gateway.conf配置: preread_length=4, 最小合法请求可能为值为0的四字节int (一个空请求), 所以设置为4

协议样例: ****{"module":"user","action":"getInfo"} ,其中 **** 首部四字节int, 由于是二进制, 所以首部四字节看起来是乱码

```
public function onGatewayMessage($recv_buffer)
{
    // 接收到的数据长度
    $recv_len = strlen($recv_buffer);
    // 如果接收的长度还不够四字节, 那么要等够四字节才能解包到请求长度
    if($recv_len < 4)
    {
        // 不够四字节, 等够四字节
        return 4 - $recv_len;
    }
    // 从请求数据头部解包出整体数据长度
    $unpack_data = unpack('Ntotal_len', $recv_buffer);
    $total_len = $unpack_data['total_len'];

    // 返回还差多少字节没有收到, 这里可能返回0, 代表请求接收完整
    return $total_len - $recv_len;
}
```

3、HTTP协议:

HTTP协议包含协议头与包体, 二者直接使用\r\n\r\n分割, POST请求协议头中包含 Content-Length 代表包体的长度。

请求样例:

```
GET / HTTP/1.1\r\n
Host: www.baidu.com\r\n
Connection: keep-alive\r\n
Cache-Control: max-age=0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0
Accept-Encoding: gzip,deflate,sdch\r\n
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6\r\n
Cookie: uc_login_unique=1fe1b19668b1419d8c45a6ac738fed76;\r\n
\r\n
```

Gateway.conf配置: preread_length=65535，由于HTTP协议是短链接的，即一个链接上只发一个请求，所以不会有多个请求连在一起的粘包现象，也就是说不会有读取越界的情况，所以预读长度可以设置为很大的值。短链接请求中主要任务就是判断读取的数据是否是一个完整的请求

```
public function onGatewayMessage($recv_buffer)
{
    // 协议头不完整，再读一些数据
    if(!strpos($recv_buffer, "\r\n\r\n"))
    {
        return 65535;
    }

    // GET请求只有协议头，POST请求还要读包体，包体长度在协议头中的Content-Length中存储
    if(strpos($recv_buffer, "POST"))
    {
        // 找Content-Length
        $match = array();
        if(preg_match("/\r\nContent-Length: ?(\d*)\r\n/", $recv_buffer, $match))
        {
            $content_lenght = $match[1];
        }
        else
        {
            return 0;
        }
        // 看包体长度是否符合
        $tmp = explode("\r\n\r\n", $recv_buffer, 2);
        $remain_length = $content_lenght - strlen($tmp[1]);
        return $remain_length >= 0 ? $remain_length : 0;
    }

    return 0;
}
```


Event::onMessage

(WorkerMan>=2.0 , Gateway/Worker模型)

说明:

```
void Gateway::onMessage(int $client_id, string $recv_buffer);
```

当收到一个完整的客户端请求时触发

参数

`$client_id`

全局唯一的客户端

`$recv_buffer`

完整的客户端请求数据。

范例

```
use \Lib\Gateway;

class Event
{
    ...
    /**
     * 有消息时触发该方法
     * @param int $client_id 发消息的client_id
     * @param string $message 消息
     * @return void
     */
    public static function onMessage($client_id, $message)
    {
        // 群聊，转发请求给其它所有的客户端
        return Gateway::sendToAll($message);
    }
    ...
}
```

Event::onClose

(WorkerMan>=2.0 , Gateway/Worker模型)

说明:

```
void Gateway::onClose(int $client_id);
```

客户端主动断开时触发。一般在这里做一些数据清理工作

参数

`$client_id`

全局唯一的client_id

范例

```
/**
 * 当用户断开连接时触发的方法
 * @param integer $client_id 断开连接的客户端client_id
 * @return void
 */
public static function onClose($client_id)
{
    // 广播 xxx 退出了
    GateWay::sendToAll("client[$client_id] logout\n");
}
```

超全局数组 `$_SESSION`

(WorkerMan>=2.1.2 , Gateway/Worker模型)

注意：

超全局数组 `$_SESSION` 只在Gateway /Worker模型中支持

什么是超全局数组 `$_SESSION`

(WorkerMan>=2.1.2 , Gateway/Worker模型)

什么是超全局数组 `$_SESSION`

WorkerMan中的超全局数组 `$_SESSION` 和PHP自身的 `$_SESSION` 功能基本相同。每个client_id对应一个 `$_SESSION` 数组，`$_SESSION` 数组中可以保存对应客户端的会话数据，对应的client_id的后续请求可以直接使用这个数据。

`$_SESSION` 使用场景

(WorkerMan>=2.1.2 , Gateway/Worker模型)

例如客户端链接WorkerMan后，需要发送验证数据让服务端验证是否合法，一般要传递一次用户名和密码数据，然后在 `Gateway::onMessage($client_id, $message)` 中通过查询数据库验证 `$message` 中的用户名密码是否正确，如果正确就可以将用户的uid写入到 `$_SESSION` 中如 `$_SESSION['uid']=$uid;`，那么当这个client_id再次发来数据时，要判断这个客户端是否是被验证过的，就可以用 `$_SESSION['uid']` 是否被设置来判断。

WorkerMan中`$_SESSION`使用注意事项

- 使用 `$_SESSION` 时无需调用`session_start`等函数，可直接使用
- `$_SESSION` 中无法保存对象以及资源

WorkerMan中 \$_SESSION 原理

(WorkerMan>=2.1.2 , Gateway/Worker模型)

在WorkerMan的Gateway/Worker模型中，每个客户端的 \$_SESSION 数据是存储在Gateway进程内存中的，每次Gateway进程转发消息给BusibuessWorker进程时，都会顺便携带上对应客户端的 \$_SESSION 数据给BusibuessWorker进程，这时BusibuessWorker进程就能使用 \$_SESSION 了。而当 \$_SESSION 数据有更改时，BusibuessWorker会将新的 \$_SESSION 数据传递给Gateway进程进行保存。

超全局数组 `$_SERVER`

(WorkerMan>=2.1.2 , Gateway/Worker模型)

注意：

超全局数组 `$_SERVER` 只在Gateway /Worker模型中支持

什么是超全局数组 `$_SERVER`

(WorkerMan>=2.1.2 , Gateway/Worker模型)

什么是超全局数组 `$_SERVER`

WorkerMan中的超全局数组 `$_SERVER` 包含了5个元素，分别是：

- `REMOTE_ADDR` // 客户端ip (如果客户端处于局域网，则是客户端所在局域网的出口ip)
- `REMOTE_PORT` // 客户端端口 (如果客户端处于局域网，则是客户端所在局域网的出口端口)
- `GATEWAY_ADDR` // gateway所在服务器的ip
- `GATEWAY_PORT` // gateway所在服务器的端口
- `GATEWAY_CLIENT_ID` // 全局唯一的客户端id

\$_SERVER 使用场景

(WorkerMan>=2.1.2 , Gateway/Worker模型)

当需要在Event.php中获得客户端的ip及端口信息时，可以使用 `$_SERVER['REMOTE_ADDR']` 和 `$_SERVER['REMOTE_PORT']` 获得。当想在某个函数逻辑处理时获得当前客户端的client_id时，可以使用 `$_SERVER['GATEWAY_CLIENT_ID']` 方便的获得

WorkerMan中\$_SERVER使用注意事项

- `$_SERVER['GATEWAY_ADDR']` 和 `$_SERVER['GATEWAY_PORT']` 开发者一般用不到，可以忽略。

WorkerMan中 \$_SERVER 原理

(WorkerMan>=2.1.2 , Gateway/Worker模型)

在WorkerMan的Gateway/Worker模型中，每个客户端都会连接在gateway进程上，当gateway进程收到客户端的数据时，会将客户端的ip端口及client_id连通消息传递给worker进程，worker进程初始化\$_SERVER数组（包含GATEWAY_ADDR/GATEWAY_PORT）。

调试

- [基本调试](#)
- [网络抓包](#)
- [跟踪系统调用](#)

基本调试

在WorkerMan中调试需开启 `workerman.conf.debug=1`，然后重启WorkerMan，然后可以在想要debug的地方使用 `echo`、`var_dump`、`var_export` 等函数打印变量，等程序运行到此处后便可以将变量打印在终端。

开启 `workerman.conf.debug=1` 后，载入到业务进程的任何PHP文件有修改都会被自动检测到，并自动运行 `reload` 命令平滑重启业务进程，以便新的业务代码载入到内存。如果不想某组进程被平滑重启，可以在对应进程的配置中加入 `no_reload=1`

如果不想某组进程打印数据到终端，可以单独设置对应进程的配置 `no_debug=1`，则即使 `workerman.conf.debug=1` 并且用了 `echo`、`var_demp`、`var_export` 等函数打印了数据，对应进程也不会打印数据到终端

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 3524290970, win 32768,
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 692696454, ack 35242909
E..<..@.@.<.....h..)I....i.....0....@....
.....

/*
 * TCP第三次握手，完成TCP链接
 * 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, win 256, options [no
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,
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_abc9330bef79b4aba5b24fa373506d9=1402048017; Hm_lvt_5fedb3bdce89499492c079ab
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 [I
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
E...(v@.@.....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 [I
E..4.k@.@.HV.....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":"mynick"}

* 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 {"type":"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 :
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 :
E...(w@.@.....h..)I.#..l.....
...^...^..l{"type":"user_list","user_list":[{"uid":"783654164","name":"\u732a\u732a"}, {"uid":"783654165","name":"\u732a\u732a"}]}

/*

* 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 [I
E..4.m@.@.HT.....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 :
E...(x@.@.....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 [I
E..4.n@.@.HS.....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 :
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 [I
E..4.X@.@.#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 :
E..n.o@.@.H.....h..l.)I.....b.....
fTX.d.P[(...9H..C=LT.~.BV=...0SnB-X.

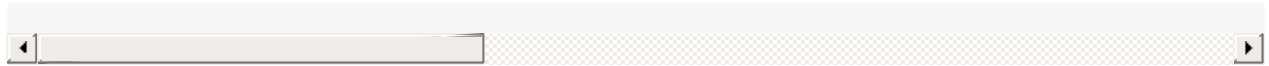
/*
 * mynick用户发言 hi, all !
 * 7272端口向所有浏览器客户端中一个浏览器转发发言数据 {"type":"say","from_uid":....
 */
17:51:02.776785 IP 127.0.0.1.7272 > 127.0.0.1.60653: Flags [P.], seq 346:448, ack 812, win :
E...(y@.@.....h..)I...l.....
.....d{"type":"say","from_uid":783700053,"to_uid":"all","content":"hi, all !","time":"

/*
 * 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 [I
E..4.p@.@.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 :
E.....@.@.3g.....h..f..$.G.....
.....^d{"type":"say","from_uid":783700053,"to_uid":"all","content":"hi, all !","time":"

/*
 * 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 [I
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（发起链接请求）；`[.]` 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

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 `workerman status` 能看到workerman相关进程的信息 如下：

```
Hello admin
-----GLOBAL STATUS-----
WorkerMan version:2.1.2
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_request packet_err t
10352   1.5M      tcp://0.0.0.0:55151 1407836524 ChatWeb        12            0
10354   1.25M     tcp://0.0.0.0:7272  1407836524 Gateway        3             0
10355   1.25M     tcp://0.0.0.0:7272  1407836524 Gateway        0             0
10365   1.25M     tcp://0.0.0.0:55757 1407836524 StatisticWeb   0             0
10358   1.25M     tcp://0.0.0.0:7272  1407836524 Gateway        3             0
10364   1.25M     tcp://0.0.0.0:55858 1407836524 StatisticProvider 0             0
10356   1.25M     tcp://0.0.0.0:7272  1407836524 Gateway        3             0
10366   1.25M     udp://0.0.0.0:55656 1407836524 StatisticWorker 55            0
10349   1.25M     tcp://127.0.0.1:7373 1407836524 BusinessWorker 5             0
10350   1.25M     tcp://127.0.0.1:7373 1407836524 BusinessWorker 0             0
10351   1.5M      tcp://127.0.0.1:7373 1407836524 BusinessWorker 5             0
10348   1.25M     tcp://127.0.0.1:7373 1407836524 BusinessWorker 2             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，卡在链接还是读取网络数据等

安全

- 1、配置中除了 `Monitor.conf` `FileMonitor.conf` 中 `user=root` , 其它进程的配置文件中的user应该设置为权限较低的用户, 例如 `www-data` `nobody` 等
- 2、`Monitor.conf` 中的 `listen` 设置的端口是远程登录的端口, 如果不想开启远程登录, 请将ip设置为 `127.0.0.1` (默认都是配置的 `127.0.0.1` ,只允许本机登录), 如 `listen=tcp://127.0.0.1:2009` (端口号可以设置成其它值)。
- 3、如果要开启远程控制, 可将 `Monitor.conf` 中的 `listen` 中的ip设置为内网ip或者外网ip或者 `0.0.0.0` , 例如 `listen=tcp://127.0.0.1:2009` (端口号可以设置成其它值)
- 4、如果开启远程控制, 记得将 `Monitor.conf` 中的 `password` 设置成你自己的密码

高级应用

- [查看运行状态](#)
- [telnet远程登录控制](#)
- [分布式部署](#)
- [心跳检测](#)

查看运行状态

运行 `./workerman/bin/workermmand status` 可以查看到WorkerMan的运行状态，类似如下：

```
Hello admin
-----GLOBAL STATUS-----
WorkerMan version:2.1.1
start time:2014-07-14 23:37:00    run 6 days 22 hours
load average: 0, 0, 0
1 users      8 workers      14 processes
worker_name  exit_status  exit_count
BusinessWorker  0          0
ChatWeb        0          2
FileMonitor    0          0
Gateway        0          0
Monitor        0          0
StatisticProvider 0          0
StatisticWeb    0          24
StatisticWorker 0          0
-----PROCESS STATUS-----
pid    memory    listening    timestamp    worker_name    total_request    packet_err
17508   2.25M    tcp://127.0.0.1:7373  1405352220  BusinessWorker  0                0
17509   2.25M    tcp://127.0.0.1:7373  1405352220  BusinessWorker  0                0
17510   2.25M    tcp://127.0.0.1:7373  1405352220  BusinessWorker  0                0
17511   2.5M     tcp://127.0.0.1:7373  1405352220  BusinessWorker  0                0
24345   2.25M    tcp://0.0.0.0:55151  1405950078  ChatWeb        14               0
17514   2.25M    tcp://0.0.0.0:7272   1405352220  Gateway        5                0
17515   2.25M    tcp://0.0.0.0:7272   1405352220  Gateway        17               0
17516   2.25M    tcp://0.0.0.0:7272   1405352220  Gateway        12               0
17517   2.25M    tcp://0.0.0.0:7272   1405352220  Gateway        199              0
17519   4.5M     tcp://0.0.0.0:55858  1405352220  StatisticProvider 32055            0
14256   5M       tcp://0.0.0.0:55757  1405929526  StatisticWeb    676              0
17527   2M       udp://0.0.0.0:55656  1405352220  StatisticWorker 3929424          0
```

说明

GLOBAL STATUS

从这以栏中我们可以看到

WorkerMan的版本 `version:2.1.1`

启动时间 `2014-07-14 23:37:00 run 6 days 22 hours`

服务器负载 `load average: 0, 0, 0`

`1 users`（一个WorkerMan远程登录管理员） `8workers`（8种进程，包括Monitor、FileMonitor进程）
`14 processes`（共14个进程）

`worker_name`（服务名） `exit_status`（退出状态值） `exit_count`（该状态的退出次数）

可以看到其中 `ChatWeb` 退出2次，退出状态码为0（为0一般为正常退出，例如配置设置了`max_requests`）

PROCESS STATUS

pid：进程pid

memory：该进程占用内存

listening：传输层协议及监听ip端口

timestamp：该进程启动时间戳

worker_name：该进程运行的服务服务名，对应于conf/conf.d下的配置名

total_request：该进程接收多少连接

packet_err：该进程接收的错误包数量

thunder_herd：该进程惊群数，注意多核处理器该值大于1是正常的

client_close：该进程对应的客户端主动关闭连接次数

send_fail：该进程向客户端发送数据失败次数

throw_exception：该进程内业务未捕获的异常数量

suc/total：该进程处理请求成功率

telnet远程登录控制

WorkerMan允许管理员通过Monitor.conf开放的端口远程telnet远程登录WorkerMan（相关的安全信息参见安全章节）。

```
输入
telnet xxx.xxx.xxx.xxx 2009
远程登录需要输入密码
输入命令
status
展示workerman状态
status
Hello admin

-----GLOBAL STATUS-----
WorkerMan version:2.1.1
start time:2014-07-14 23:37:00    run 6 days 22 hours
load average: 0, 0, 0
1 users      8 workers      14 processes
worker_name  exit_status  exit_count
BusinessWorker  0           0
ChatWeb        0           2
FileMonitor    0           0
Gateway        0           0
Monitor        0           0
StatisticProvider 0           0
StatisticWeb    0           24
StatisticWorker 0           0

-----PROCESS STATUS-----
pid    memory    listening    timestamp worker_name    total_request packet_err
17508  2.25M    tcp://127.0.0.1:7373 1405352220 BusinessWorker  0              0
17509  2.25M    tcp://127.0.0.1:7373 1405352220 BusinessWorker  0              0
17510  2.25M    tcp://127.0.0.1:7373 1405352220 BusinessWorker  0              0
17511  2.5M     tcp://127.0.0.1:7373 1405352220 BusinessWorker  0              0
24345  2.25M    tcp://0.0.0.0:55151 1405950078 ChatWeb        14             0
17514  2.25M    tcp://0.0.0.0:7272 1405352220 Gateway        5              0
17515  2.25M    tcp://0.0.0.0:7272 1405352220 Gateway        17             0
17516  2.25M    tcp://0.0.0.0:7272 1405352220 Gateway        12             0
17517  2.25M    tcp://0.0.0.0:7272 1405352220 Gateway        199            0
17519  4.5M     tcp://0.0.0.0:55858 1405352220 StatisticProvider 32055          0
14256  5M       tcp://0.0.0.0:55757 1405929526 StatisticWeb    676            0
17527  2M       udp://0.0.0.0:55656 1405352220 StatisticWorker 3929424        0
```

status 展示的信息与 workerman/bin/workermmand status 展示的信息相同，status详细信息参见第一章中查看运行状态章节

远程控制支持的命令

1、 status

展示WorkerMan运行状态

2、 stop

停止WorkerMan（注意运行stop后telnet远程控制也将断开）

3、 reload

平滑重启WorkerMan

4、 kill \$pid

安全重启\$pid对应进程。 \$pid为 status 中看到的进程pid , 为数字

5、 quit

断开telnet远程控制

分布式部署

首先确保你的WorkerMan版本大于2.1.1 (运行 `workerma/bin/workermmand status` 查看)

以applications/Demo为例，假如需要部署三台服务器提供高可用服务。

1、首先将进程切分，将Gateway进程部署在一台机器上(假设内网ip为192.168.0.1)，BusinessWorker部署在另外两台机器上 (内网ip为192.168.0.2/3)

2、由于192.168.0.1这台机器只部署Gateway进程，所以将该ip上的 `workerman/conf/conf.d/BusinessWorker.conf` 删掉，避免运行BusinessWorker进程

3、配置Gateway服务器(192.168.0.1)上的 `workerman/conf/conf.d/Gateway.conf` 中的 `lan_ip=192.168.0.1` 与本机ip一致

3、由于192.168.0.2/3 两台服务器只部署BusinessWorker进程，所以将这两台ip上的 `workerman/conf/conf.d/Gateway.conf` 删掉，避免运行Gateway进程

4、由于物理机之间需要共享一些数据，需要部署一台memcache服务器，假设部署在Gateway (192.168.0.1) 这台机器上，memcache服务端口为22322

5、由于要使用memcache，所以要给三台服务器的PHP添加memcache扩展。ubuntu/debian可使用 `sudo apt-get install php5-memcache` 安装

6、配置memcache，更改三台服务器上 `applications/Demo/Config/Store.php` 中的 `driver`、`gateway` 两项配置如下，

```
// 存储驱动改为memcache
public static $driver = self::DRIVER_MC
// 更改memcache ip和端口
public static $gateway = array(
    '192.168.0.1:22322',
);
```

7、首先启动Gateway服务器192.168.0.1，然后启动BusinessWorker的服务器192.168.0.2/3

至此，WorkerMan分布式部署完毕。

一些问题及解答

一、为什么将Gateway与BusinesWorker分别部署在不同的服务器上？

首先说明的是不一定非要将Gateway BusinessWorker分开部署，但是推荐分开部署，原因如下：

1、由于Gateway只负责网络IO，只要服务器带宽够用，绝大多数情况下Gateway服务器不会成为瓶颈，所以在很长时间我们只需要一台或者少数几台Gateway服务器即可。由于我们不想BusinessWorker影响到Gateway，所以将Gateway和BusinessWorker分开部署

2、BusinessWorker主要负责业务逻辑。当请求量增大时，由于可能BusinessWorker业务比较复杂，当请

求量增大时，负载可能会明显升高，这时我们只要单纯增加BusinessWorker服务器即可，Gateway服务器则一般不许要变动，也就是不用通知客户端Gateway的ip有所变动

3、当系统BusinessWorker负载较低，需要下线服务器时，我们只需要下线BusinessWorker服务器即可，无需变动Gateway服务器，也就不会导致客户端链接因为服务器下线而断开。

二、当Gateway服务器集群负载较高时，我们怎么扩容？

假如memcache扩展已经安装

扩容Gateway（假设ip为192.168.0.100）

- 1、删除 `workerman/conf/conf.d/BusinessWorker.conf`，确保BusinessWorker在这台服务器不会运行
- 2、配置 `workerman/conf/conf.d/Gateway.conf` 中的 `lan_ip=192.168.0.100` 与本机内网ip一致
- 3、配置 `applications/Demo/Config/Store.php` 中的 `driver`、`gateway` 两项配置如下，

```
// 存储驱动改为memcache
public static $driver = self::DRIVER_MC
// 设置gateway存储ip端口为memcache服务的ip和端口，可以设置多个
public static $gateway = array(
    '192.168.0.1:22322',
);
```

- 4、启动workerman `./workerman/bin/workerman start`

至此Gateway服务器扩容完成。BusinessWorker会自动感知到有Gateway服务器有扩容，并且与该Gateway服务器建立联系，如果Gateway服务器接入LVS则整个扩容过程客户端无感知

三、当BusinessWorker服务器集群负载较高时，我们怎么扩容？

假如memcache扩展已经安装

扩容BusinessWorker：

- 1、删除 `workerman/conf/conf.d/Gateway.conf`，确保Gateway进程不会在该服务器上运行
- 2、配置 `applications/Demo/Config/Store.php` 中的 `driver`、`gateway` 两项配置如下

```
// 存储驱动改为memcache
public static $driver = self::DRIVER_MC
// 设置gateway存储ip端口为memcache服务的ip和端口，可以设置多个
public static $gateway = array(
    '192.168.0.1:22322',
);
```

- 3、启动workerman `./workerman/bin/workerman start`

至此BusinessWorker服务器扩容完成。新扩容的BusinessWorker会自动与线上Gateway进程建立联系，整个扩容过程客户端无感知

当BusinessWorker服务器集群负载较低时，需要下线一些机器怎么实施？

只需要停止BusinessWorker的服务，运行 `./workerman/bin/workermmand stop`，然后下线即可。Gateway服务器会自动感知有BusinessWorker服务器下线，不会再将请求转发给下线的机器，整个下线过程中不影响服务质量。

当Gateway服务器集群负载较低时，需要下线一些机器怎么实施？

首先还是要说明下Gateway服务器一般情况下不会成为系统瓶颈，所以一般你很长时间内Gateway服务器数量是一个稳定的值，一般一台即可

下线Gateway服务器，首先停止服务，运行 `./workerman/bin/workermmand stop`，此时会导致该服务器上已有的客户端链接断开，然后下线服务器即可。此时BusinessWorker会感知到有Gateway服务器下线，会自动断开与Gateway进程的联系。

服务端到客户端的心跳检测

WorkerMan支持服务端向客户端定时发送心跳检测数据

为什么服务端要向客户端发心跳检测？

有些极端情况如客户端掉电、网络关闭、拔网线、路由故障等，这些极端情况都属于连接断开的情况，然而这些情况如果没有应用层的心跳检测，服务端是无法快速感知的。而服务端定时向客户端发送心跳数据可以解决这个问题。

什么情况需要服务定时向客户端发送心跳检测数据？

一般的应用其实不需要向客户端发送心跳数据，因为正常的情况客户端断开服务端是能立刻感知到的。如果应用要求对于极端连接断开的情况也要及时检测到，则可以使用服务端到客户端的定时心跳检测。

心跳检测的原理是什么？

服务端向客户端发送心跳检测，客户端接收到心跳数据后，可以忽略不做任何处理，也可以回应心跳检测（向服务端发送一段任意数据）。这就分为两种情况，

- 1、当服务端不要求客户端必须回应心跳检测时，假如客户端遇到掉电等极端情况，这时服务端向客户端发送的心跳数据在TCP层面就会发送超时，遇到这种超时情况TCP会重试多次（次数及间隔依赖操作系统的配置），多次无果后会断开连接。这种极端情况从连接断开到服务端检测到可能要持续至少10分钟。
- 2、当服务端要求必须回应检测时，如果服务端在规定的时间（Gateway.conf配置）内没有收到客户端的任何数据，则立刻判定客户端已经断开，服务端就立即断开连接。

WorkerMan中如何配置心跳检测？

在workerman/conf/conf.d/Gateway.php中设置心跳检测，主要是如下三个选项

```
;服务端向客户端发送心跳数据的时间间隔 单位：秒。如果设置为0代表不发送心跳检测
ping_interval = 10

;客户端连续ping_not_response_limit次ping_interval时间内不回应心跳则断开链接。
;如果设置为0代表客户端不用发送回应数据，即通过TCP层面检测连接的连通性（极端情况至少10分钟才能检测到）
ping_not_response_limit = 2

;要发送的心跳请求数据，将心跳请求保存成文件，然后配置文件路径 如ping_data=/yourpath/ping.data，
;workerman会将此文件中的内容当作心跳请求发送给客户端
;注意 心跳请求数据一定要符合你的通讯协议
ping_data = ../applications/Chat/ping.data
```

ping.data 文件如何生成

workerman会将 ping_data 文件中的内容当作心跳请求发送给客户端。首先一定要注意 ping.data 要符合你的通讯协议，例如是websocket协议，则ping.data中保存的是websocket协议的数据。

假如要生成一个内容为 `{"type":"ping"}` 的websocket 协议的心跳检测数据，可以这样生成：

```
php -a
include './applications/Chat/Protocols/WebSocket.php';
file_put_contents('./applications/Chat/ping.data', \Protocols\WebSocket::encode('{"type":"p
```

然后配置 `ping_data = ./applications/Chat/ping.data` 即可。`ping_data` 文件按的位置可以随意。注意相对路径是以workerman路径为参考的，例如workerman-chat/workerman。

假如生成一个同样内容为 `{"type":"ping"}` 的 jsonProtocol 协议的心跳数据,可以这样生成

```
php -a
include './applications/Chat/Protocols/JsonProtocol.php';
file_put_contents('./applications/Chat/ping.data', \Protocols\JsonProtocol::encode(array('t
```

假如生成一个内容为 `{"type":"ping"}` 的 json+回车 为协议的心跳检测数据，则可以直接新建一个文件，输入`{"type":"ping"}`然后后面打一个回车即可。

技巧1

WorkerMan>=2.1.4

如果客户端有定时向服务端发送心跳检测，则服务端可以不必向客户端发送心跳检测，即利用客户端主动发送的数据判断客户端是否存活。这时我们需要设置 `ping_data=` ,例如如下配置

```
ping_interval = 10
ping_not_response_limit = 2
ping_data =
```

代表服务端不发送任何心跳数据，但是客户端如果 `ping_interval*ping_not_response_limit=20` 秒内连接上没有任何请求则断开连接

技巧2

服务端可以只发送心跳检测，而不要求客户端必须回应，则可以像下面这样设置（假设ping.data文件已经准备好）。

```
ping_interval = 10
ping_not_response_limit = 0
ping_data = ../applications/Chat/ping.data
```

其中 `ping_not_response_limit = 0` 代表服务端允许客户端不响应心跳，也就是通过TCP层面检测连接的状态，这样如果客户端因为断电等极端情况断开连接，可能需要等待TCP超时重传多次才能感应到连接断开，耗时较长。

统计监控模块

- [关于统计监控系统](#)
- [统计监控系统特点](#)
- [原理](#)
- [如何使用](#)
- [权限验证](#)

关于统计监控模块

当我们的程序部署在正式环境后，我们需要监控我的程序是否运行正常。统计监控模块就是用来监控系统运转情况的一个工具，它提供了一些重要接口的调用情况，如各个时段的调用量、调用耗时、成功率、失败日志等等，并以曲线图和表格的形式通过网页展示出来。

例如：<http://www.workerman.net:55757>

代码及配置

- 统计监控系统代码位置在 applications/Statistics 下
- 对应的配置为 workerman/conf/conf.d/下的 StatisticWeb.conf StatisticPrivoder.conf StatisitcWorker.conf。配置说明见[Statistics统计模块配置](#)

统计监控系统特点

- 1、业务使用统计监控模块提供的api上报需要监控的数据
- 2、统计监控数据以分时曲线图及表格的形式通过网页展现，例如 <http://www.workerman.net:55757>
- 3、数据以udp传输层协议上报，不影响正常业务逻辑
- 4、支持分布式部署
- 5、分布式部署可以通过探测自动感知带运行统计模块的服务器
- 6、分部署部署时只需要用浏览器打开任意一台服务器的监控页面，便可以掌控整个集群的运行情况，统计结果会分布式查询并自动汇总，类似MapReduce。
- 7、统计监控模块可以直接部署在应用服务器上，统计过程中不占用网络带宽。也可以独立出来部署成统计监控系统集群
- 8、由于监控是通过udp接口方式上报的，所以统计监控系统也可以用在其它项目中

原理

- 1、业务通过api的形式上报数据，上报数据内容包括模块名、接口名、是否成功、错误码、错误日志。然后统计结果以udp的形式上报给统计系统的接收进程StatisticWorker（对应配置为StatisticWorker.conf）
- 2、统计系统接收进程StatisticWorker收到udp统计数据后，会把统计数据在内存中进行分类汇总，当汇总数据量达到一定值（默认1M）或者到达该存盘的时间间隔（默认1分钟）或者WorkerMan收到stop命令，就会将汇总数据以文件的形式存储在磁盘。
- 3、通过浏览器查看统计数据时，浏览器发出的HTTP请求会被WorkerMan的WebServer进程接收（统计系统的WebServer配置在StatisticWeb.conf）。
- 4、WebServer接收HTTP请求后根据StatisticWeb.conf配置会移交applications/Statistics/Web/index.php处理，index.php会将请求路由到指定函数。对应函数会并发异步去各个服务器（如果是分布式部署）查询各自服务器上StatisticWorker进程的统计结果。
- 5、各个服务器上使用StatisticProvider进程提供本机的查询服务(配置为StatisticProvider.conf)，StatisticProvider进程收到index.php路由到的函数上的请求后，在本地磁盘查找数据，并将数据返回。
- 6、index.php路由到的函数会异步收到各个服务器上StatisticProvider进程返回的查询结果，然后做全局的数据汇总，再通过WebServer以HTTP协议返回全局的汇总结果
- 7、浏览器上便展示了整个WorkerMan集群的整体的运行情况

如何使用

WorkerMan的一些demo或者例子已经集成了统计监控系统，会自动统计一些重要的监控数据。

如果业务想要监控自己感兴趣的接口或者想在其它项目中使用，可以参考本节

客户端的使用

- 1、客户端位置在applications/Statistics/Clients/StatisticClient.php，客户端只有一个文件，将该文件拷贝到需要监控的项目
- 2、引入客户端文件
- 3、上报示例

```
<?php
require 'yourpath/StatisticClient.php';
// 统计开始
StatisticClient::tick("User", 'getInfo');
// 统计的产生，接口调用是否成功、错误码、错误日志
$success = true; $code = 0; $msg = '';
// 假如有个User::getInfo方法要监控
$user_info = User::getInfo();
if(!$user_info){
    // 标记失败
    $success = false;
    // 获取错误码，假如getErrCode()获得
    $code = User::getErrCode();
    // 获取错误日志，假如getErrMsg()获得
    $msg = User::getErrMsg();
}
// 上报结果
StatisticClient::report('User', 'getInfo', $success, $code, $msg);
```

服务端的使用

- 1、如果要单独部署服务端，可以从这个连接下载服务端 <http://www.workerman.net/workerman-statistics>
- 2、如果应用上已经运行了统计系统，例如Gateway/Worker模型的applications/Demo、workerman-chat等,则可以不用再部署服务端
- 3、服务端的配置可以用默认值
- 4、通过网页浏览器访问 ip:55757查询统计数据，55757是默认端口，可以在workerman\conf\conf.d\StatisticsWeb.conf里更改
- 5、55757端口可以改成80端口，但是要确认80端口没有被其它WebServer占用，例如apache、nginx等
- 6、可以把ip:55757这个服务部署在apache或者nginx下，只要将根目录指向applications\Statistics\Web即可
- 7、首次使用服务端需要配置数据源的ip（运行了统计监控系统的服务器ip），可以自动探测或者手动添

加。

权限验证

如果统计数据需要设置访问用户名密码，可以在applications\Statistics\Config\Config.php中设置

例如设置登录用户名为 admin，密码为123456，则配置如下：

```
namespace Statistics\Config;
class Config
{
    // 数据源端口，会向这个端口发送udp广播获取ip，然后从这个端口以tcp协议获取统计信息
    public static $ProviderPort = 55858;

    // 管理员用户名，用户名密码都为空字符串时说明不用验证
    public static $adminName = 'admin';

    // 管理员密码，用户名密码都为空字符串时说明不用验证
    public static $adminPassword = '123456';
}
```

如果不加登录验证，设置

```
public static $adminName = '';
public static $adminPassword = '';
```

即可

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

短链接（每次请求完成后关闭链接，下次请求建立新的链接）：

条件：压测脚本开500个并发线程模拟500个并发用户，每个线程链接Workerman 10W次，每次链接发送1个请求

结果：吞吐量：1.9W/S，cpu利用率：32%

长链接（每次请求后不关闭链接，下次请求继续复用这个链接）：

条件：压测脚本开2000个并发线程模拟2000个并发用户，每个线程链接Workerman 1次，每个链接发送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

短链接（每次请求完成后关闭链接，下次请求建立新的链接）：

条件：压测脚本开1000个并发线程模拟1000个并发用户，每个线程链接Workerman 10W次，每次链接发送1个请求

结果：吞吐量：3.5W/S，cpu利用率：35%

长链接（每次请求后不关闭链接，下次请求继续复用这个链接）：

条件：压测脚本开6000个并发线程模拟6000个并发用户，每个线程链接Workerman 1次，每个链接发送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

压力测试代码及脚本连接: <https://github.com/walkor/workerman-bench>

常见问题

- [WorkerMan下扩展的安装](#)
- [WorkerMan日志](#)

WorkerMan下扩展的安装

首先

虽然你的环境Apache或者php-fpm支持某个扩展，但是不代表着WorkerMan能直接使用这个扩展，原因如下：

- php是有很多运行模式的，或者说很多sapi。比如 apache 的mod_php、apache2handler php-fpm的fpm 还有isapi、phttpd、cli等有20多个sapi
- 每种运行模式的php可执行文件一般都不是相同的,所以可能内置的扩展有所差异
- 每种运行模式的 php.ini 文件可能都不是相同的。比如apache可能用的是 `/etc/php5/apache2/php.ini` php-fpm可能用的是 `/etc/php5/fpm/php.ini` 而 workerman是以cli模式运行的，用的可能是 `/etc/php5/cli/php.ini`
- 而有时安装扩展的时候是需要php.ini中配置的，所以即使apache配置了php.ini，不代表cli模式下的workerman就支持了那个扩展，还是需要在cli的php.ini配置一下的

现在的问题

的问题就是找到php cli模式使用的php.ini。运行如下命令，可以列出所有php cli所使用的扩展的ini文件包括php.ini

```
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/curl.ini,
/etc/php5/cli/conf.d/gd.ini,
/etc/php5/cli/conf.d/libevent.ini,
/etc/php5/cli/conf.d/memcache.ini,
/etc/php5/cli/conf.d/mysql.ini,
/etc/php5/cli/conf.d/mysqli.ini,
/etc/php5/cli/conf.d/pdo.ini,
/etc/php5/cli/conf.d/pdo_mysql.ini,
/etc/php5/cli/conf.d/proctitle.ini
```

上面的路径可能因为系统不同而不一样，如果你有扩展的.so文件，只是缺少一个php.ini配置，可以参考上面的规则配置下

最后

如果你的cli是apt-get install 或者 yum install 安装的，那么扩展也可以通过apt-get 或者 yum安装，php cli的php.ini 会自动配置，非常方便。

apt/yum可以通过以下命令查找扩展，例如memcache 扩展

apt安装运行: `apt-cache search memcache | grep php`

yum安装运行: `yum search memcache | grep php`

然后 apt-get install 你搜到的要装的扩展名
yum install 你搜到的要装的扩展名

WorkerMan日志

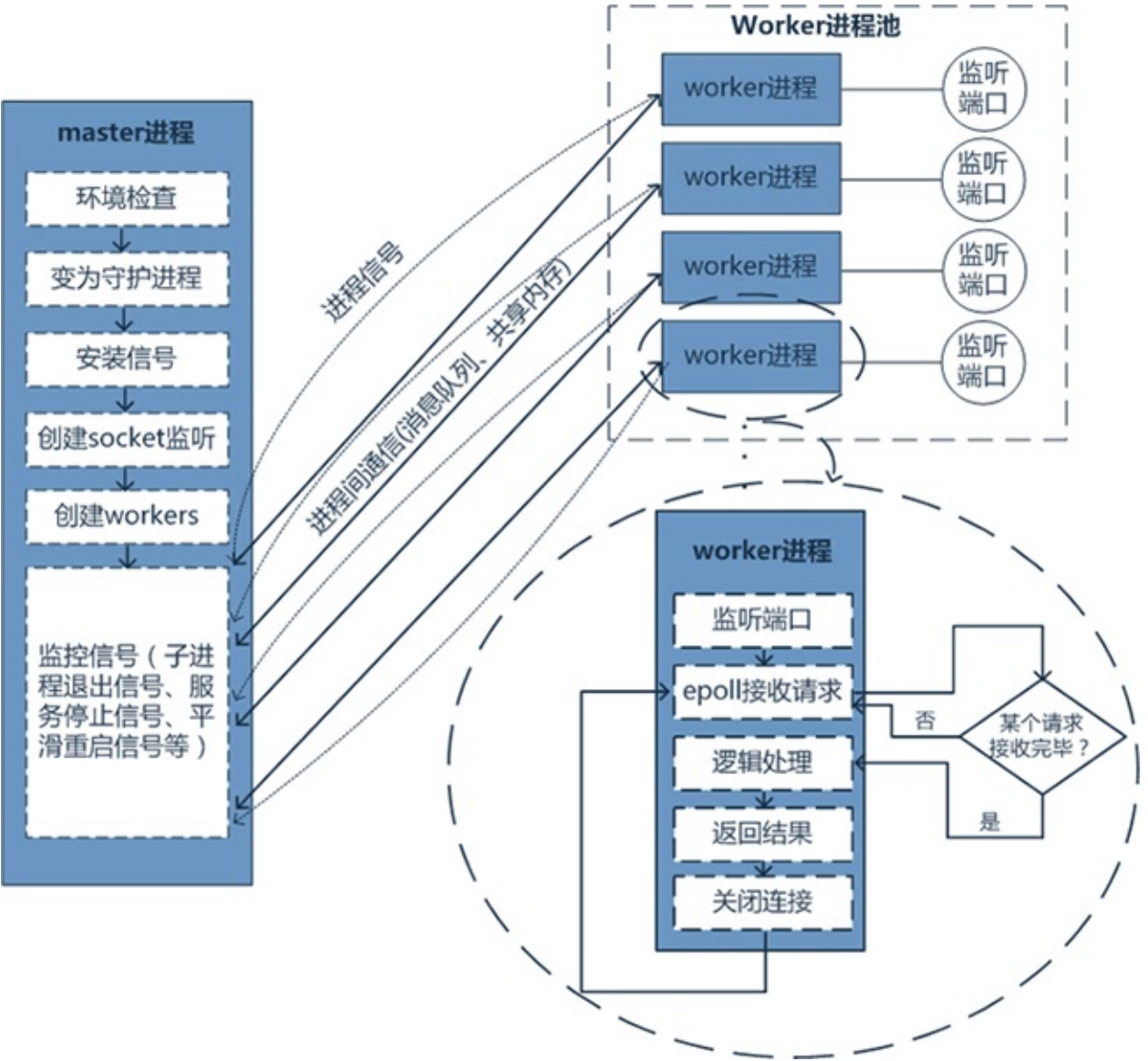
位置

WorkerMan运行过程中产生的日志默认放在 `workerman/logs/{date}/server.log` 下。可以在 `workerman/conf/workerman.conf` 中的 `log_dir` 选项配置。

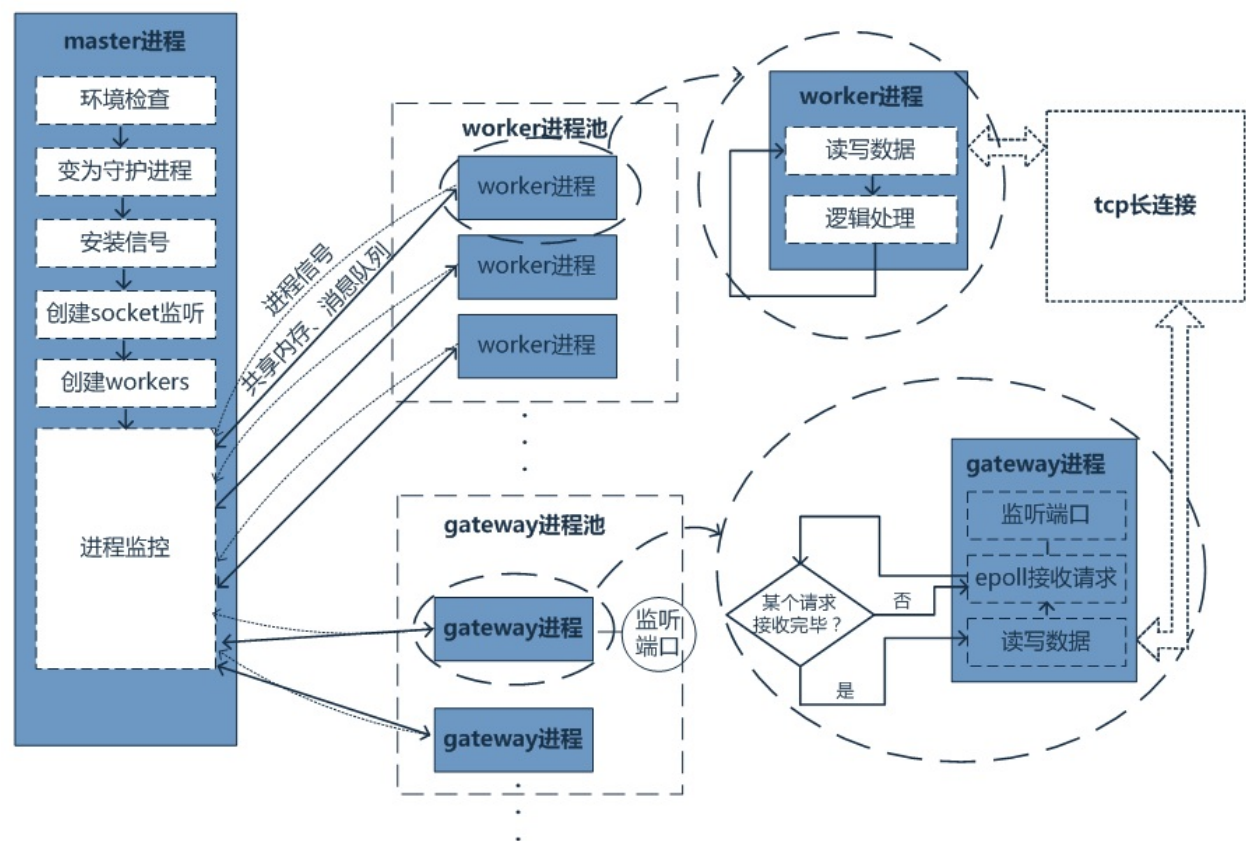
记录的内容

WorkerMan日志记录的是WorkerMan内核运行过程中需要开发者注意的日志，包括WorkerMan运行、停止、重启、平滑重启等正常操作的日志，也包含一些要特别注意的异常日志，例如进程异常（一般是因为业务代码有FatalError）退出的日志、客户端异常关闭连接的日志等等

master-worker模型



master-gateway-worker模型



一些开发示例

文件传输(二进制协议)

二进制传输协议说明

四字节网络字节序unsigned int标识整个包的长度 + 一字节char标识文件名长度 + 文件名 + 文件二进制数据

说明

下面以基本开发模型为例，Gateway/Worker模型开发与其类似（onGatewayMessage相当于dealInput，onMessage相当于dealProcess）。

创建文件 applications/FileTransferDemo/BinaryTransfer.php

```
<?php

/*
 * 文件传输协议
 * 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
 * }
 */

class BinaryTransfer extends Man\Core\SocketWorker
{
    // 协议头长度
    const PACKAGE_HEAD_LEN = 5;

    /**
     * 分包，判断数据是否完整
     * 不完整返回还需要多少字节数据要接收
     * 完整的话返回0
     */
    public function dealInput($recv_buffer)
    {
        $recv_len = strlen($recv_buffer);
        // 如果不够一个协议头的长度，则继续等待
        if($recv_len < self::PACKAGE_HEAD_LEN)
        {
            return self::PACKAGE_HEAD_LEN - $recv_len;
        }
        // 解包
        $package_data = unpack('Ntotal_len/Cname_len', $recv_buffer);
        // 整个包的长度
        $total_len = $package_data['total_len'];
        // 返回还有多少字节没有收完
        return $total_len - $recv_len;
    }

    /**
```

```

* 收到完整上传数据后如何处理
* 这里将文件保存到了tmp目录
*/
public function dealProcess($recv_buffer)
{
    // 解包
    $package_data = unpack('Ntotal_len/Cname_len', $recv_buffer);
    // 文件名长度
    $name_len = $package_data['name_len'];
    // 从数据流中截取出文件名
    $file_name = substr($recv_buffer, self::PACKAGE_HEAD_LEN, $name_len);
    // 从数据流中截取出文件二进制数据
    $file_data = substr($recv_buffer, self::PACKAGE_HEAD_LEN + $name_len);
    // 文件保存路径
    $save_path = "/tmp/$file_name";
    // 保存文件
    file_put_contents($save_path, $file_data);
    // 发送给客户端结果
    $this->sendToClient("upload $file_name to $save_path success.\n");
}
}

```

创建配置文件

workerman/conf/conf.d/BinaryProtocolTransfer.conf

```

;业务进程入口文件
worker_file = ../applications/FileTransferDemo/BinaryTransfer.php
;传输层协议 ip 及端口
listen = tcp://0.0.0.0:8333
;启动多少服务进程
start_workers = 5
;以哪个用户运行该进程，为了安全请使用权限较低的用户，例如www-data nobody
user = root
;请求到来时预读长度，这里固定5
preread_length = 5
;长链接
persistent_connection = 1

```

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);

// 执行上传
fwrite($client, $package);

// 打印结果
echo fread($client, 8192);
```

测试

```
// 命令行运行，其中$filepath是本地文件的路径
php client.php $filepath
```


文件传输(文本协议)

文本传输协议说明

客户端将二进制文件用base64_encode编码（会增大1/3的体积），转变成明文方便传输。。整体协议采用json+回车 格式传输，json包含了文件名、文件base64_encode后的数据。协议格式类似如下：

```
{"file_name":"xxx.jpg","file_data":"PD9waHAKLyO....."}\n
```

说明

下面以基本开发模型为例，Gateway/Worker模型开发与其类似（onGatewayMessage相当于dealInput，onMessage相当于dealProcess）。

创建文件 applications/FileTransferDemo/TextTransfer.php

```
<?php
/*
 * 文件传输协议
 * 客户端将文件base64_encode 然后通过 json+回车 协议发送过来
 * 协议样例：
 * {"file_name":"xxx","file_data":"base64_encode(file)"}\n
 *
 */
class TextTransfer extends Man\Core\SocketWorker
{
    /**
     * 分包，判断数据是否完整
     * 不完整返回还需要多少字节数据要接收
     * 完整的话返回0
     *
     */
    public function dealInput($recv_buffer)
    {
        if($recv_buffer[strlen($recv_buffer)-1] !== "\n")
        {
            return 1;
        }
        return 0;
    }

    /**
     * 收到完整上传数据后如何处理
     *
     */
    public function dealProcess($recv_buffer)
    {
        // 解包
        $package_data = json_decode(trim($recv_buffer), true);
        if(!isset($package_data['file_name']) || !isset($package_data['file_data']))
        {
            $this->sendToClient("package error\n");
            return;
        }
    }
}
```

```

        // 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);
        // 文件保存路径
        $save_path = "/tmp/$file_name";
        // 保存文件
        file_put_contents($save_path, $file_data);
        // 发送给客户端结果
        $this->sendToClient("upload $file_name to $save_path success.\n");
    }
}

```

创建配置文件 workerman/conf/conf.d/TextProtocolTransfer.conf

```

;业务进程入口文件
worker_file = ../applications/FileTransferDemo/TextTransfer.php
;传输层协议 ip 及端口
listen = tcp://0.0.0.0:8444
;启动多少服务进程
start_workers = 5
;以哪个用户运行该进程，为了安全请使用权限较低的用户，例如www-data nobody
user = root
;请求到来时预读长度，这里固定1
preread_length = 1
;长链接
persistent_connection = 1

```

客户端文件 textclient.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 = "127.0.0.1:8444";

// 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");
}
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);
~
```

测试

```
// 命令行运行，其中$filepath是本地文件的路径
php textclient.php $filepath
```

Mysql数据库的使用

Workerman中封装了数据库类，在Gateway/Worker模型开发时可以直接使用。基本开发模型中需要拷贝Gateway/Worker模型中相关目录文件才能使用数据库类。

Gateway/Worker模型 数据库使用示例

1、数据库配置applications/XXX/Config/Db.php

```
<?php
namespace Config;
/**
 * mysql配置
 * @author walkor
 */
class Db
{
    /**
     * 数据库的一个实例配置，则使用时像下面这样使用
     * $user_array = Db::instance('user')->select('name,age')->from('users')->where('age>12');
     * 等价于
     * $user_array = Db::instance('user')->query('SELECT `name`,`age` FROM `users` WHERE `age`>12');
     * @var array
     */
    public static $user = array(
        'host'    => '127.0.0.1',
        'port'    => 3306,
        'user'     => 'your_user_name',
        'password' => 'your_password',
        'dbname'   => 'user',
        'charset'  => 'utf8',
    );
}
```

2、applications/XXX/Event.php

```
<?php
use \Lib\Gateway;
use \Protocols\TextProtocol;
use \Lib\Db;

/**
 * 数据库示例，假设有个user库，里面有个user表
 */
class Event
{
    /**
     * 网关有消息时，判断消息是否完整
     */
    public static function onGatewayMessage($buffer)
    {
        return TextProtocol::check($buffer);
    }
}
```

```

/**
 * 有消息时触发该方法，根据发来的命令打印2个用户信息
 * @param int $client_id 发消息的client_id
 * @param string $message 消息
 * @return void
 */
public static function onMessage($client_id, $message)
{
    // 发来的消息
    $command = trim($message);
    if($command !== 'get_user_list')
    {
        Gateway::sendToClient($client_id, "unknown command\n");
        return;
    }
    // 获取用户列表（这里是临时的一个测试数据库）
    $ret = Db::instance('user')->select('*')->from('users')->where('uid>3')->offset(5)-:
    // 打印结果
    return Gateway::sendToClient($client_id, var_export($ret, true));
}
}

```

基本开发模型数据库使用示例

1、目录结构如下（目录文件参考Gateway/Worker模型）

```

applications/MysqlDemo
├── Config
│   └── Db.php
├── Lib
│   ├── Autoloader.php
│   ├── DbConnection.php
│   └── Db.php
└── MysqlDemo.php

```

2、applications/MysqlDemo/MysqlDemo.php

```

<?php
use \Lib\Db;
require_once __DIR__ . '/Lib/Autoloader.php';
/**
 *
 */
class MysqlDemo extends Man\Core\SocketWorker
{
    /**
     * 分包，判断数据是否完整
     * 不完整返回还需要多少字节数据要接收
     * 完整的话返回0
     */
    /**
     *
     */
    public function dealInput($recv_buffer)
    {
        if($recv_buffer[strlen($recv_buffer)-1] !== "\n")

```

```

        {
            return 1;
        }
        return 0;
    }

    /**
     * 收到完整上传数据后如何处理
     */
    /**/
    public function dealProcess($recv_buffer)
    {
        $commend = trim($recv_buffer);
        if($commend !== 'get_user_list')
        {
            return $this->sendToClient("unknown commend\n");
        }
        $ret = Db::instance('one_demo')->select('*')->from('wenda.wenda_users')->where('uid:');
        return $this->sendToClient(var_export($ret, true));
    }
}

```

测试

终端输入telnet ip port

输入 get_user_list

服务器返回 数据

数据库类使用的一些示例

配置

在Config/Db.php中配置数据库信息，如果有多个数据库，可以按照one_demo的配置在Db.php中配置多个实例 例如下面配置了两个数据库实例

```

<?php
namespace Config;
class Db
{
    // 数据库实例1
    public static $db1 = array(
        'host'    => '127.0.0.1',
        'port'    => 3306,
        'user'     => 'mysql_user',
        'password' => 'mysql_password',
        'dbname'  => 'db1',
        'charset'  => 'utf8',
    );

    // 数据库实例2
    public static $db2 = array array(

```

```

        'host'      => '127.0.0.1',
        'port'      => 3306,
        'user'       => 'mysql_user',
        'password'   => 'mysql_password',
        'dbname'     => 'db2',
        'charset'    => 'utf8',
    );
}

```

使用方法

```

$db1 = \Lib\Db::instance('db1');
$db2 = \Lib\Db::instance('db2');

// 获取所有数据
$db1->select('ID,Sex')->from('Persons')->where('sex= :sex')->bindValue(array('sex'=>'M'))->query();
// 等价于
$db1->select('ID,Sex')->from('Persons')->where("sex= 'F' ")>query();
// 等价于
$db1->query("SELECT ID,Sex FROM `Persons` WHERE sex='M'");

// 获取一行数据
$db1->select('ID,Sex')->from('Persons')->where('sex= :sex')->bindValue(array('sex'=>'M'))->query();
// 等价于
$db1->select('ID,Sex')->from('Persons')->where("sex= 'F' ")>row();
// 等价于
$db1->row("SELECT ID,Sex FROM `Persons` WHERE sex='M'");

// 获取一列数据
$db1->select('ID,Sex')->from('Persons')->where('sex= :sex')->bindValue(array('sex'=>'M'))->query();
// 等价于
$db1->select('ID,Sex')->from('Persons')->where("sex= 'F' ")>column();
// 等价于
$db1->column("SELECT ID,Sex FROM `Persons` WHERE sex='M'");

// 获取单个值
$db1->select('ID,Sex')->from('Persons')->where('sex= :sex')->bindValue(array('sex'=>'M'))->query();
// 等价于
$db1->select('ID,Sex')->from('Persons')->where("sex= 'F' ")>single();
// 等价于
$db1->single("SELECT ID,Sex FROM `Persons` WHERE sex='M'");

// 复杂查询
$db1->select('*')->from('table1')->innerJoin('table2','table1.uid = table2.uid')->where('age > 13')->query();
// 等价于
$db1->query("SELECT * FROM `table1` INNER JOIN `table2` ON `table1`.`uid` = `table2`.`uid` WHERE `age` > 13");

// 插入
$insert_id = $db1->insert('Persons')->cols(array('Firstname'=>'abc', 'Lastname'=>'efg', 'Sex'=>'M', 'Age'=>13))->query();
// 等价于
$insert_id = $db1->query("INSERT INTO `Persons` ( `Firstname`,`Lastname`,`Sex`,`Age`) VALUES ('abc','efg','M',13)");

// 更新
$row_count = $db1->update('Persons')->cols(array('sex'))->where('ID=1')->bindValue('sex', 'F')->query();
// 等价于
$row_count = $db1->update('Persons')->cols(array('sex'=>'F'))->where('ID=1')->query();

```

```
// 等价于
$row_count = $db1->query("UPDATE `Persons` SET `sex` = 'F' WHERE ID=1");

// 删除
$row_count = $db1->delete('Persons')->where('ID=9')->query();
// 等价于
$row_count = $db1->query("DELETE FROM `Persons` WHERE ID=9");
```


p2p之UDP打洞

创建applications/p2p/p2p.php

```
<?php
class p2p extends Man\Core\SocketWorker
{
    /**
     * 分包，判断数据是否完整
     * 不完整返回还需要多少字节数据要接收
     * 完整的话返回0
     *
     */
    /**
     * public function dealInput($recv_buffer)
     * {
     *     if($recv_buffer[strlen($recv_buffer)-1] !== "\n")
     *     {
     *         return 1;
     *     }
     *     return 0;
     * }
     *
     *
     *
     * 收到完整上传数据后如何处理
     *
     */
    /**
     * public function dealProcess($recv_buffer)
     * {
     *     $commend = trim($recv_buffer);
     *     if($commend !== 'get_address')
     *     {
     *         return $this->sendToClient("unknown commend\n");
     *     }
     *     // 返回远程客户端的ip及端口
     *     $address = $this->getRemoteAddress();
     *     return $this->sendToClient($address);
     * }
     *
     *
     *
     * }
}
```

配置 workerman/conf/conf.d/p2p.conf

```
;业务进程入口文件
worker_file = ../applications/p2p/p2p.php
;传输层协议 ip 及端口
listen = udp://0.0.0.0:8666
;启动多少服务进程
start_workers = 5
;以哪个用户运行该进程，为了安全请使用权限较低的用户，例如www-data nobody
user = root
;请求到来时预读长度，这里固定1
preread_length = 1
;长链接
persistent_connection = 1
```

获得客户端出口ip及端口 (php模拟)

```
<?php
// xxx.xxx.xxx.xxx 为远程workerman服务器ip
$client = stream_socket_client("udp://xxx.xxx.xxx.xxx:8666", $errno, $errmsg);
if(!$client)
{
    exit($errmsg);
}
fwrite($client, "get_address\n");

echo "your address is ".fread($client, 8192)."\n";
```

待续....
