workerman-manual

walkor

Published
with GitBook

Table of Contents

WorkerMan 3.x 手册

本手册只适用于WorkerMan3.x版本

Home Page: www.workerman.net

版权信息

序言

PHP是一种被广泛应用的开源脚本语言，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laravel、Yii、thinkphp等。

传统的PHP应用程序基本上是在Apache等Web容器中运行的，浏览器与Web容器采用HTTP协议通信，然而在很多实际项目中HTTP协议无法满足我们的需求，尤其是在服务端和客户端要保持长连接，做实时双向通讯时，HTTP协议显得力不从心。例如即时IM通讯，游戏服务器通讯，与硬件传感器通讯等等，开发这些应用程序我们无法直接使用nginx/apache + PHP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。

WorkerMan是一款纯PHP开发的开源的高性能的PHP socket服务器框架，基于WorkerMan开发者可以开发出各种网络服务器，例如基于websocket的服务器、游戏服务器、移动通讯服务器、智能家居服务端、物联网服务、web服务器、RPC服务器等等。几乎任何基于TCP/UDP通讯的服务端都可以用WorkerMan来开发。WorkerMan使得开发者摆脱PHP只能用于Web开发的束缚，向更广阔的前景发展。

本手册作用范围

WorkerMan有分为多进程版本WorkerMan和多线程版本WorkerMan-MT两个版本，多进程版本是稳定版本并运行在Linux系统下，多线程版本处于研发阶段，可以运行在windows系统下用作开发

本手册主要是针对Linux多进程3.x版本的说明

客户端

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

入门指引

WorkerMan的特性

1、纯PHP开发

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

2、支持PHP多进程

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

3、支持TCP、UDP

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

4、支持长连接

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

5、支持各种应用层协议

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

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-Flapyy-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，性能也同样非常强悍。

7、支持服务平滑重启

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

使用。

8、支持文件更新检测及自动加载

在开发过程中，我们希望在我们改动代码后能够立刻生效，以便查看结果。WorkerMan提供了文件检测及自动加载功能，只要文件有更新，WorkerMan会自动运行reload，以便加载新的文件，使之生效。

9、支持以指定用户运行子进程

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

10、支持对象或者资源永久保持

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

11、高性能

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

14、支持HHVM

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

15、支持分布式部署

16、支持守护进程化

17、支持多端口监听

18、支持标准输入输出重定向

简单的开发实例

实例一、使用HTTP协议对外提供Web服务

创建**http_test.php**文件

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

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

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

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

// 运行worker
Worker::runAll();
```

运行

```
php http_test.php start
```

测试

假设服务端ip为127.0.0.1

在浏览器中访问url http://127.0.0.1:2345

实例二、使用WebSocket协议对外提供服务

创建**ws_test.php**文件

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

// 创建一个Worker监听2346端口，使用websocket协议通讯
$ws_worker = new Worker("websocket://0.0.0.0:2346");

// 启动4个进程对外提供服务
```

```
$ws_worker->count = 4;

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

// 运行worker
Worker::runAll();
```

运行

```
php ws_test.php start
```

测试

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

```
// 假设服务端ip为127.0.0.1
ws = new WebSocket("ws://127.0.0.1:2346");
ws.onopen = function() {
    alert("连接成功");
    ws.send('tom');
    alert("给服务端发送一个字符串：tom");
};
ws.onmessage = function(e) {
    alert("收到服务端的消息：" + e.data);
};
```

实例三、直接使用TCP传输数据

创建tcp_test.php

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

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

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

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

```
// 运行worker
Worker::runAll();
```

运行

```
php tcp_test.php start
```

测试

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

安装配置

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(版本高于5.3.3),并安装了pcnlt、posix扩展

3、建议安装libevent扩展，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。

关于**WorkerMan**依赖的扩展

1、pcntl扩展

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

2、posix扩展

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

3、libevent扩展

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。

如何安装扩展

参见 附录-安装扩展 章节

安装

WorkerMan实际上没有安装脚本，如果你的php环境已经装好，只需要把WorkerMan源代码下载下来即可运行。 下载方法建议用Github clone，你也可以通过下载zip文件的方式下载WorkerMan代码程序。

通过Github安装

centos系统安装教程

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

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

2、命令行运行（此步骤是通过pecl安装libevent扩展，如果失败请尝试按照 4.1 环境要求 一节中使用源码phpize的方式安装）

```
pecl install channel://pecl.php.net/libevent-0.1.0
```

3、命令行运行（此步骤是配置libevent的ini配置）

```
echo extension=libevent.so > /etc/php.d/libevent.ini
```

注意在提示libevent installation [autodetect]: 时按回车即可

4、命令行运行（此步骤是通过github下载WorkerMan主程序）

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

5、进入到WorkerMan主程序根目录，命令行运行以下命令启动WorkerMan

```
php start.php 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（此步骤包含了安装php-cli主程序、libevent扩展及github程序）

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

2、命令行运行（此步骤是通过pecl安装libevent扩展，如果失败请尝试按照 4.1 环境要求 一节中使用源码phpize的方式安装）

```
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（此步骤是配置libevent的ini配置）

```
echo extension=libevent.so > /etc/php5/cli/conf.d/libevent.ini
```

4、命令行运行（此步骤是通过github下载WorkerMan主程序）

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

5、进入到WorkerMan主程序根目录，命令行运行以下命令启动WorkerMan

```
php start.php 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、进入到WorkerMan主程序根目录，命令行运行以下命令启动WorkerMan

```
php start.php start
```

启动与停止

启动

以debug方式启动

```
php start.php start
```

以daemon方式启动

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

停止

```
php start.php stop
```

重启

```
php start.php restart
```

平滑重启

```
php start.php reload
```

查看状态

```
php start.php status
```

启动文件说明

WorkerMan自带一个启动文件start.php，用于启动applications下的所有应用。它的原理是扫描查找applications/应用/下的start.php文件，并载入。applications/应用/start.php里面是服务启动的具体脚本，包含端口、进程数等设置。

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，在平滑重启时会导致客户端连接断开。为了避免这种情况，WorkerMan将即时通讯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，进而再转发给用户。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进程的中配置noReload=true来避免平滑重启Gateway进程导致客户端连接断开。

开发流程

开发前必读

使用WorkerMan开发应用，你需要了解以下内容：

一、WorkerMan开发与普通PHP开发的不同之处

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

1、应用层协议不同

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

由于非*HTTP*协议的应用，所以 `header()` `setcookie()` `session_start` 等函数无法直接使用，需要使用*WorkerMan*提供的方法

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载入解析后便常驻内存，使得类的定义、全局对象、类的静态成员不会释放，便于后续重复利用

3、注意避免类和常量的重复定义

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

4、注意单例模式的链接资源的释放

- 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。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）。WorkerMan提供了数据库类，有断开重连的功能，开发者可以直接使用。

5、注意不要使用exit、die出语句

- WorkerMan运行在PHP命令行模式下，当调用exit、die退出语句时，会导致当前进程退出。虽然子进程退出后会立刻重新创建一个的相同的子进程继续服务，但是还是可能对业务产生影响。

二、需要了解的基本概念

1、TCP传输层协议

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

2、应用层协议

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

3、短连接

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

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。所以长连接在每个操作完后都不断开，下次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

当需要主动向客户端推送数据时，例如聊天类、即时游戏类、手机推送等应用需要长连接。 长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`php your_file.php reload`命令，能够做到在不影响服务质量的情况下更新应用程序

三、两种开发模式

在WorkerMan中有两种开发模式

1、基于**Worker**开发

即直接使用Worker类来开发，例如下面的代码

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

// 创建一个Worker监听2347端口，不使用任何应用层协议
$tcp_worker = new Worker("tcp://0.0.0.0:2347");
// 当客户端发来数据时
$tcp_worker->onMessage = function($connection, $data)
{
    // 向客户端发送hello $data
    $connection->send('hello ' . $data);
};
```

Worker是WorkerMan中最基本的功能单元，提供了接收并维护大量客户端连接的能力，同时也可以做相应的业务逻辑。基于Worker能开发出各种复杂的进程模型，以满足各种应用需求。例如下面讲到的Gateway/Worker进程模型也是基于Worker开发的。

2、基于Gateway/Worker开发

Gateway/Worker也是基于Worker开发的，这种进程模型分为两组进程，Gateway进程和Worker进程，其中Gateway进程只负责网络IO，Worker进程只负责处理业务逻辑。所有客户端与Gateway进程开放的端口建立连接发送及接收数据，Gateway进程将接收到的数据交给Worker进程处理，Worker处理过程中如果需要给其它客户端发送数据，则将数据交给Gateway转发。

Gateway/Worker模型非常适合游戏类、即时IM、聊天室等客户端与客户端需要即时通讯的业务。

下面是基于Gateway/Worker开发小蝌蚪服务端代码片段

```
use \Workerman\WebServer;
use \GatewayWorker\Gateway;
use \GatewayWorker\BusinessWorker;

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

// gateway
$gateway = new Gateway("websocket://0.0.0.0:8585");

$gateway->name = 'TodpoleGateway';

$gateway->count = 4;

$gateway->reloadable = false;

$gateway->lanIp = '127.0.0.1';

$gateway->startPort = 4000;



// bussinessWorker
$worker = new BusinessWorker();
```

```
$worker->name = 'TodpoleBusinessWorker';

$worker->count = 4;
```

四、区分主进程和子进程

有必要注意下代码是运行在主进程还是子进程，一般来说，直接在启动脚本中运行的代码属于主进程的，而 `onXXXX` 回调都是运行在子进程的。

例如下面的代码

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

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

注意：不要在主进程中初始化数据库、memcache、redis等连接资源，因为主进程初始化的连接可能会被子进程自动继承（尤其是使用单例的时候），但是这个链接资源在子进程是无法使用的，因为服务端通过这个连接返回的数据在多个进程上都可读，会导致数据错乱。

目录结构

```
.
├── start.php             // workerman启动脚本，用于启动applications下的所有应用
├── Applications          // 基于workerman开发的所有应用程序
│   └── Todpole           // 基于workerman开发的小蝌蚪应用程序
│       ├── Config        // 配置相关
│       │   ├── Db.php    // 数据库配置
│       │   └── Store.php// memcache存储配置
│       ├── Event.php     // 业务实现【开发主要关注这个文件】
│       ├── start.php     // 启动脚本，定义监听端口、进程数量等等
│       └── Web           // 小蝌蚪应用自身的Web界面文件
│
├── GatewayWorker             // Gateway/Worker模型公共类库
│   ├── BusinessWorker.php    // Worker业务进程
│   ├── Gateway.php           // Gateway进程
│   └── Lib                   // 类库
│       ├── Context.php       // Gateway与Worker通讯的上下文
│       ├── DbConnection.php // 数据库连接类
│       ├── Db.php            // 数据库连接管理类，对应配置在Applications/YourApp/Config/Db.php
│       ├── Gateway.php       // 与客户端通讯的基础类
│       ├── Lock.php          // 锁相关
│       ├── StoreDriver       // 存储引擎相关
│       │   └── File.php      // 文件存储引擎
│       └── Store.php         // 存储管理类，对应配置在Applications/YourApp/Config/Store.php
└── Workerman                       // workerman内核代码
    ├── Autoloader.php              // 自动加载类
    ├── Connection                  // socket连接相关
    │   ├── ConnectionInterface.php// socket连接接口
    │   ├── TcpConnection.php       // Tcp连接类
    │   ├── AsyncTcpConnection.php // 异步Tcp连接类
    │   └── UdpConnection.php       // Udp连接类
    ├── Events                      // 网络事件库
    │   ├── EventInterface.php      // 网络事件库接口
    │   ├── Libevent.php            // Libevent网络事件库
    │   └── Select.php              // Select网络事件库
    ├── Lib                         // 常用的类库
    │   ├── Constants.php           // 常量定义
    │   └── Timer.php               // 定时器
    ├── Protocols                   // 协议相关
    │   ├── ProtocolInterface.php  // 协议接口类
    │   ├── Http                    // http协议相关
    │   │   └── mime.types          // mime类型
    │   ├── Http.php                // http协议实现
    │   ├── Websocket.php           // websocket协议的实现
    │   └── GatewayProtocol.php     // Gateway/Worker模型中的通讯协议
    ├── WebServer.php               // WebServer
    └── Worker.php                  // Worker
```

开发规范

应用程序目录

应用程序目录一般放在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，WorkerMan的入口文件为start.php，放在applications/YourApp/下（YourApp为你应用的名称）。

applications/YourApp/start.php 中是创建监听进程相关的代码，例如下面的基于Worker开发的代码片段

```
<?php
use Workerman\Worker;

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

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

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

注意

applications下的启动文件start.php中不要运行`Worker::runAll();`，`Worker::runAll();`统一由WorkerMan根目录中的start.php运行

WorkerMan中的代码规范

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

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

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

例如项目Applications/MyApp/，类文件Applications/MyApp/MyClass.php因为在项目根目录，所以命名空

间省略。类文件Applications/MyApp/Protocols/MyProtocol.php因为MyProtocol.php在MyApp项目的Protocols目录下，所以要加上命名空间 `namespace Protocols;`，如下：

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

3、普通函数及变量名采用小写加下划线方式 例如

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

4、类成员及类的方法采用首字母小写的驼峰形式 例如：

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

5、函数及类的参数采用小写加下划线方式

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

基本流程

（以一个简单的Websocket聊天室服务端为例）

1、在applications下建立项目目录，例如建立目录

Applications/SimpleChat/

2、选定开发模型，基于Worker开发还是基于Worker/Gateway开发

假设用户量不大，这里基于Worker开发

3、选定协议

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

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

4、写启动脚本

Applications/SimpleChat/start.php

```
<?php
use Workerman\Worker;

$global_uid = 0;
$connections_array = array();

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

// 当客户端发送消息过来时，转发给所有人
function handle_message($connection, $data)
{
    global $connections_array;
    foreach($connections_array as $conn)
    {
        $conn->send("user[{$connection->uid}] said: $data");
    }
}

// 当客户端断开时，从连接数组中删除
function handle_close($connection)
{
    global $connections_array;
```

```
    unset($connections_array[$connection->uid]);
}


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

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

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

5、测试

Text协议可以用telnet命令测试

```
telnet 127.0.0.1 2347
```

定制通讯协议

通讯协议的作用

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

在WorkerMan中为什么要制定协议？

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

WorkerMan已经支持的协议

WorkerMan目前已经支持HTTP、Websocket、Telnet协议，需要基于这些协议通讯时可以直接使用，使用方法及时在初始化Worker或者Gateway类时指定协议，例如

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

使用自定义的通讯协议

当WorkerMan自带的通讯协议满足不了开发需求时，开发者可以定制自己的通讯协议，定制方法见下一节内容

如何定制协议

实际上制定自己的协议是比较简单的事情。简单的协议一般包含两部分:

- 区分数据边界的标识
- 数据格式定义

一个例子

协议定义

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

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

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

实现步骤

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

1、协议文件放到Applications下项目的Protocols文件夹，例如文件Applications/MyApp/Protocols/JsonNL.php

2、实现JsonNL类，以`namespace Protocols;`为命名空间，必须实现三个静态方法分别为 input、encode、decode

具体实现

Applications/MyApp/Protocols/JsonNL.php的实现

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

```
        // 没有换行符，无法得知包长，返回0继续等待数据
        if($pos === false)
        {
            return 0;
        }
        // 有换行符，返回当前包长（包含换行符）
        return $pos+1;
    }

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

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

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

文件：Applications\MyApp\start.php

```
use Workerman\Worker;
$json_worker = new Worker('JsonNL://0.0.0.0:1234');
$json_worker->onMessage = ...
...
```

协议接口说明

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

```
namespace Workerman\Protocols;

use \Workerman\Connection\ConnectionInterface;

/**
 * Protocol interface
* @author walkor <walkor@workerman.net>
```

```
 */
interface ProtocolInterface
{
    /**
     * 用于在接收到的recv_buffer中分包
     *
     * 如果可以在$recv_buffer中得到请求包的长度则返回整个包的长度
     * 否则返回0，表示需要更多的数据才能得到当前请求包的长度
     * 如果返回false或者负数，则代表错误的请求，则连接会断开
     *
     * @param ConnectionInterface $connection
     * @param string $recv_buffer
     * @return int|false
     */
    public static function input($recv_buffer, ConnectionInterface $connection);

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

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

一些例子

例子一

协议定义

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

数据包样本

```
0000000121<?xml version="1.0" encoding="ISO-8859-1"?>
<request>
    <module>user</module>
    <action>getInfo</action>
</request>
```

其中0000000121代表整个数据包长度，后面紧跟xml数据格式的包体内容

协议实现

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

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

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

```
            // 返回数据
            return $total_length_str . $xml_string;
        }
    }
```

例子二

协议定义

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

数据包样本

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

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

协议实现

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

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

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

```
        return pack('N',$total_length) . $body_json_str;
    }
}
```

例子三（使用二进制协议上传文件）

协议定义

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

协议样本

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

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

协议实现

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

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


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

```
        // 文件名长度
        $name_len = $package_data['name_len'];
        // 从数据流中截取出文件名
        $file_name = substr($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);
         return array(
             'file_name' => $file_name,
             'file_data' => $file_data,
         );
    }

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

服务端协议使用示例

```
use Workerman\Worker;

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

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), $name
// 执行上传
fwrite($client, $package);
// 打印结果
echo fread($client, 8192),"\n";
```

例子四（使用文本协议上传文件）

协议定义

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

协议样本

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

注意末尾为一个换行符，在PHP中用双引号字符`"\n"`标识

协议实现

```
namespace Protocols;
class TextTransfer
{
    public static function input($recv_buffer)
    {
        $recv_len = strlen($recv_buffer);
        if($recv_buffer[$recv_len-1] !== "\n")
        {
            return 0;
        }
        return strlen($recv_buffer);
    }

    public static function decode($recv_buffer)
    {
        // 解包
        $package_data = json_decode(trim($recv_buffer), true);
        // 取出文件名
        $file_name = $package_data['file_name'];
```

```
        // 取出base64_encode后的文件数据
        $file_data = $package_data['file_data'];
        // base64_decode还原回原来的二进制文件数据
        $file_data = base64_decode($file_data);
        // 返回数据
        return array(
             'file_name' => $file_name,
             'file_data' => $file_data,
         );
    }

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

服务端协议使用示例

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

```
use Workerman\Worker;

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

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
$address = "127.0.0.1:8333";
// 检查上传文件路径参数
if(!isset($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("$errmsg\n");
}
stream_set_blocking($client, 1);
// 文件名
$file_name = basename($file_to_transfer);
// 文件二进制数据
$file_data = file_get_contents($file_to_transfer);
// base64编码
$file_data = base64_encode($file_data);
// 数据包
$package_data = array(
    'file_name' => $file_name,
    'file_data' => $file_data,
);
// 协议包 json+回车
$package = json_encode($package_data)."\n";
// 执行上传
fwrite($client, $package);
// 打印结果
echo fread($client, 8192),"\n";
```

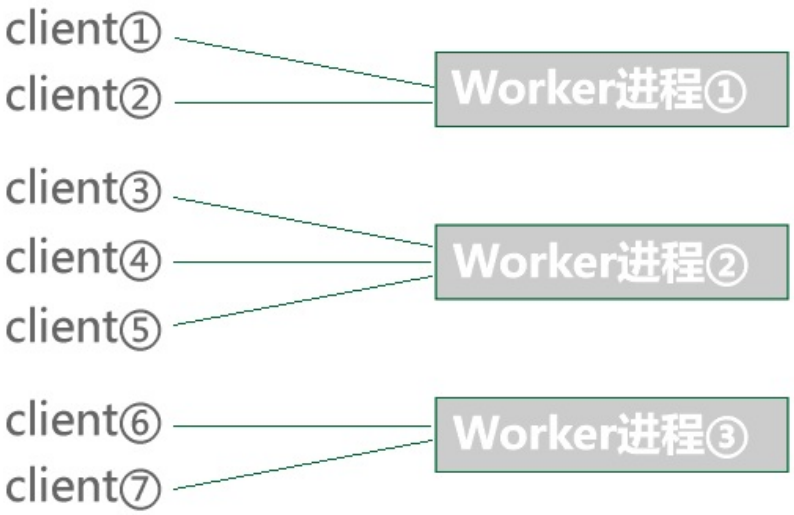
基于Worker开发流程

Worker开发适用范围

Worker说明

Worker是WorkerMan中最基本的功能单元，Worker可以开启多个进程监听端口并使用特定协议通讯。每个Worker进程独立运作，每个Worker进程都能接收无数的客户端连接，并处理这些连接上发来的数据。

Worker 的进程模型



特点：

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

缺点：

基于Worker无法方便的实现客户端与可客户端的通讯，例如上图中client①要给client⑤发送数据，由于两个客户端在不同的Worker进程，Worker进程①无法直接给Worker进程②的client⑤发送数据，所以需要我们做一些开发工作，才能实现客户端之间的通讯。

如果需要客户端之间通讯，可以基于WorkerMan的Gateway/Worker开发，见下一章节。

适用范围：

其实基于Worker能够开发出几乎所有的网络应用服务端程序，但是如果你需要客户端之间的实时通讯，例如游戏服务器、聊天服务器等，可以直接使用WorkerMan提供的Gateway/Worker开发模式。除此之外，所有的网络服务程序都可以基于Worker开发。

Worker类

WorkerMan中有两个重要的类Worker与Connection。

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

可以设置Worker实例的进程数（count属性），则会创建count个Worker进程同时监听相同的端口，并行的接收客户端连接，处理连接上的事件。

属性

count

说明:

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

设置当前Worker实例启动多少个进程，不设置时默认为1。

范例

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

name

说明:

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

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

范例

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

user

说明:

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

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

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

范例

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

reloadable

说明:

```
string Worker::$reloadable
```

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

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

范例

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

transport

说明:

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

设置当前Worker实例所使用的传输层协议，目前只支持两种，tcp和udp。不设置默认为tcp。

范例

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

connections

说明:

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

此属性中存储了当前进程的所有的客户端连接，这在广播时非常有用。

范例

```
use Workerman\Worker;
use Workerman\Lib\Timer;
$worker = new Worker('Text://0.0.0.0:8484');
// 进程启动时设置一个定时器，定时向所有客户端连接发送数据
$worker->onWorkerStart = function($worker)
{
    // 定时，每10秒一次
    Timer::add(10, function()use($worker)
    {
        // 遍历当前进程所有的客户端连接，发送当前服务器的时间
        foreach($worker->connections as $connection)
        {
            $connection->send(time());
        }
    });
};
```

daemonize

说明:

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

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

范例

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

stdoutFile

说明:

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

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

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

范例

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

pidFile

说明:

```
static Event Worker::$pidFile
```

如果无特殊需要，建议不要设置此属性

此属性为全局静态属性，用来设置WorkerMan进程的pid文件路径。

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

如果不设置，WorkerMan默认会在`sys_get_temp_dir()`下自动生成一个pid文件，并且为了避免启动多个WorkerMan实例导致pid冲突，WorkerMan生成pid文件包含了当前WorkerMan的路径

范例

```
use Workerman\Worker;
Worker::$pidFile = '/var/run/workerman.pid';

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

globalEvent

说明:

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

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

范例

```
use Workerman\Worker;
use Workerman\Events\EventInterface;

$worker = new Worker('Text://0.0.0.0:8484');
$worker->onWorkerStart = function($worker)
{
    // 当进程收到SIGALRM信号时，打印输出一些信息
    Worker::$globalEvent->add(SIGALRM, EventInterface::EV_SIGNAL, function()
    {
        echo "Get signal SIGALRM\n";
    });
};
```

回调接口

onWorkerStart

说明:

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

设置Worker启动时的回调函数，即当Worker启动后立即执行Worker::onWorkerStart成员指定的回调函数

回调函数的参数

`$worker`

即Worker对象

范例

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

onWorkerStop

说明:

```
callback Worker::$onWorkerStop
```

设置Workert停止时的回调函数，即当Worker收到stop信号后执行Worker::onWorkerStop指定的回调函数

回调函数的参数

`$worker`

即Worker对象

范例

```
use Workerman\Worker;
$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onWorkerStop = function($worker)
{
    echo "Worker stopping...\n";
};
```

onConnect

说明:

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

当有客户端连接时触发的回调函数

回调函数的参数

`$connection`

连接对象，连接对象的说明见下一节

范例

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

onMessage

说明:

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

当有客户端的连接上有数据发来时触发

回调函数的参数

`$connection`

连接对象，连接对象的说明见下一节

`$data`

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

范例

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

onClose

说明:

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

当客户端的连接断开时触发，不管连接是如何断开的，只要断开就会触发

回调函数的参数

`$connection`

连接对象，连接对象的说明见下一节

范例

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

onBufferFull

说明:

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

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

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

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

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

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

回调函数的参数

`$connection`

连接对象

范例

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

参见

onBufferDrain 当连接的应用层发送缓冲区数据全部发送完毕时触发

onBufferDrain

说明:

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

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

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

回调函数的参数

`$connection`

连接对象

范例

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

参见

onBufferFull 当连接的应用层发送缓冲区满时触发

onError

说明:

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

当客户端的连接上发生错误时触发。

目前错误类型有

1、调用Connection::send由于客户端连接断开导致的失败 `(code:WORKERMAN_SEND_FAIL msg:client closed)`

2、在触发onBufferFull后，仍然调用Connection::send，并且发送缓冲区仍然是满的状态，导致发送失败 `(code:WORKERMAN_SEND_FAIL msg:send buffer full and drop package)`

3、使用AsyncTcpConnection异步连接失败时 `(code:WORKERMAN_CONNECT_FAIL msg:stream_socket_client返回的错误消息)`

回调函数的参数

`$connection`

连接对象，连接对象的说明见下一节

`$code`

错误码

`$msg`

错误消息

范例

```
use Workerman\Worker;
$worker = new Worker('websocket://0.0.0.0:8484');
$worker->onError = function($connection, $code, $msg)
{
    echo "error $code $msg\n";
};
```

Connection类提供的接口

WorkerMan中有两个重要的类Worker与Connection。

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

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

属性

id

说明:

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

连接的id。这是一个自增的整数

范例

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

protocol

说明:

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

设置此链的应用层协议打包解包所使用的类

范例

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

worker

说明:

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

此属性为只读属性，即当前connection对象所属的worker实例

范例

```
use Workerman\Worker;
$worker = new Worker('Websocket://0.0.0.0:8484');

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

maxSendBufferSize

说明:

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

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

此属性影响onBufferFull回调

范例

```
use Workerman\Worker;
use Workerman\Protocols\TcpConnection;

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

defaultMaxSendBufferSize

说明:

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

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

此属性影响onBufferFull回调

范例

```
use Workerman\Worker;
use Workerman\Protocols\TcpConnection::;

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

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

maxPackageSize

说明:

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

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

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

范例

```
use Workerman\Worker;
use Workerman\Protocols\TcpConnection;

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

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

回调接口

onMessage

说明:

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

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

范例

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

上面代码与下面的效果是一样的

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

onClose

说明:

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

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

范例

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

上面代码与下面的效果相同

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

onBufferFull

说明:

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

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

onBufferDrain

说明:

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

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

onError

说明:

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

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

接口

send

说明:

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

向客户端发送数据

参数

`$data`

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

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

返回值

true 表示发送成功

null 表示放入待发送队列，等待异步发送

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

范例

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

getRemoteIp

说明:

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

获得该连接的客户端ip

参数

无参数

范例

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

getRemotePort

说明:

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

获得该连接的客户端端口

参数

无参数

范例

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

close

说明:

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

安全的关闭连接.

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

参数

`$data`

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

范例

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

destroy

说明:

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

立刻关闭连接。

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

参数

无参数

范例

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

pauseRecv

说明:

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

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

参数

无参数

范例

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

参见

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

resumeRecv

说明:

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

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

参数

无参数

范例

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

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

参见

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

AsyncTcpConnection类

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

__construct

```
void \Workerman\Connection\AsyncTcpConnection::__construct(string $remote_address)
```

创建一个异步连接对象

参数

`remote_address`

连接的地址，例如`tcp://www.baidu.com:80`

返回值

无返回值

示例

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;

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

connect

```
void \Workerman\Connection\AsyncTcpConnection::connect()
```

执行异步连接操作。此方法会立刻返回

参数

无参数

返回值

无返回值

示例 **Mysql**代理

```
use \Workerman\Worker;
use \Workerman\Connection\AsyncTcpConnection;

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

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

$proxy->onConnect = function($connection)
{
    global $REAL_MYSQL_ADDRESS;
    // 异步建立一个到实际mysql服务器的连接
    $connection_to_mysql = new AsyncTcpConnection($REAL_MYSQL_ADDRESS);
    // mysql连接发来数据时，转发给对应客户端的连接
    $connection_to_mysql->onMessage = function($connection_to_mysql, $buffer)use($connect
    {
        $connection->send($buffer);
    };
    // mysql连接关闭时，关闭对应的代理到客户端的连接
    $connection_to_mysql->onClose = function($connection_to_mysql)use($connection)
    {
        $connection->close();
    };
    // mysql连接上发生错误时，关闭对应的代理到客户端的连接
    $connection_to_mysql->onError = function($connection_to_mysql)use($connection)
    {
        $connection->close();
    };
    // 执行异步连接
    $connection_to_mysql->connect();

    // 客户端发来数据时，转发给对应的mysql连接
    $connection->onMessage = function($connection, $buffer)use($connection_to_mysql)
    {
```

```
        $connection_to_mysql->send($buffer);
    };
    // 客户端连接断开时，断开对应的mysql连接
    $connection->onClose = function($connection)use($connection_to_mysql)
    {
        $connection_to_mysql->close();
    };
    // 客户端连接发生错误时，断开对应的mysql连接
    $connection->onError = function($connection)use($connection_to_mysql)
    {
        $connection_to_mysql->close();
    };

};
```

测试

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

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

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

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql>
```

定时器Timer类

add

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

定时执行某个函数或者类方法

参数

`time_interval`

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

`callback`

回调函数

`args`

回调函数的参数，必须为数组

`persistent`

是否是持久的，如果只想定时执行一次，则传递false。默认是true，即一直定时执行。

返回值

返回一个整数，代表计时器的timerid，可以通过这个timerid删除对应的计时器

示例

```php
use \Workerman\Worker;

$task = new Worker();
// 开启多少个进程运行定时任务，注意多进程并发问题
$task->count = 1;
$task->onWorkerStart = function($task)
{
    $time_interval = 2.5;
    \Workerman\Lib\Timer::add($time_interval, function()
    {
        echo "task run\n";
    });
};
```

del

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

删除某个定时器

参数

`timer_id`

定时器的id，即add接口返回的整型

返回值

boolean

示例

```
use \Workerman\Worker;

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

注意事项

定时器使用注意事项

1、只能在`onXXXX`回调中添加定时器。全局的定时器推荐在`onWorkerStart`回调中设置，针对某个连接的定时器推荐在`onConnect`中设置。除非业务需要，`onMessage`中一般不适合设置定时器。

2、添加的定时任务在当前进程执行，如果任务很重（特别是涉及到网络IO的任务），可能会导致该进程阻塞，暂时无法处理其它业务。所以最好将耗时的任务放到单独的进程运行，例如建立一个/多个Worker进程运行

3、当一个任务没有在预期的时间运行完，这时又到了下一个运行周期，则会等待当前任务完成才会运行。也就是说当前进程的任务都是串行执行的，如果是多进程则进程间的任务运行是并行的。

4、要考虑到多进程设置了定时任务造成并发问题

5、可能会有1毫秒左右的误差

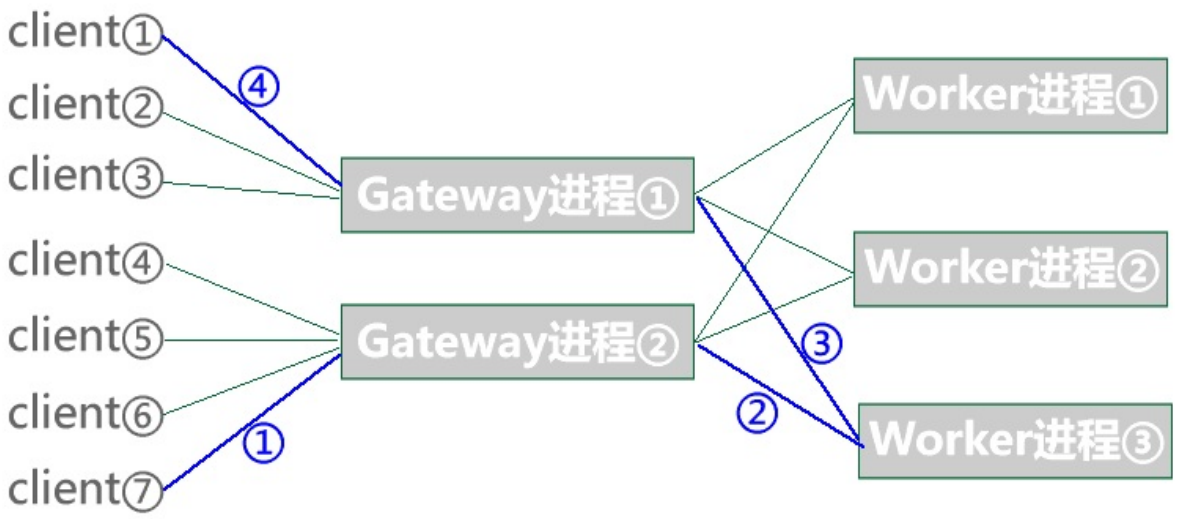
基于Gateway/BusinessWorker开发

适用范围

Gateway/Worker说明

Gateway/Worker模型是基于基础的Worker模型开发的一套可以实现客户端与客户端实时通讯的架构，主要用于游戏服务器、聊天服务器等需要客户端之间通讯的项目

Gateway/Worker 的进程模型



特点：

从图上我们可以看出Gateway负责接收客户端的连接以及连接上的数据，然后Worker接收Gateway发来的数据做处理，然后再经由Gateway把结果转发给其它客户端。每个客户端都有很多的路由到达另外一个客户端，例如client⑦与client①可以经由蓝色路径完成数据通讯

优点：

1、可以方便的实现客户端之间的通讯

2、Gateway与Worker之间是基于socket长连接通讯，也就是说Gateway、Worker可以部署在不同的服务器上，非常容易实现分布式部署，扩容服务器

3、Gateway进程只负责网络IO，业务实现都在Worker进程上，可以reload Worker进程，实现不影用户的情况下实现代码热更新

适用范围：

适用于客户端与客户端需要实时通讯的项目。如果只需要客户端与服务端之间的通讯，可以考虑基于Worker模型开发

Gateway类的使用

文件位置：GatewayWorker/Gateway.php

Gateway类是基于基础的Worker开发的。可以给Gateway对象的onWorkerStart、onWorkerStop、onConnect、onClose设置回调函数，但是无法给设置onMessage回调。Gateway的onMessage行为固定为将客户端数据转发给Worker。

Gateway类可以定制的内容

1、 协议

和Worker一样，在初始化Gateway对象时设置Gateway的协议，例如下面设置Gateway的通讯协议为websocket

```
use \GatewayWorker\Gateway;

// 指定websocket协议
$gateway = new Gateway("websocket://0.0.0.0:8585");
```

2、name

和Worker一样，可以设置Gateway进程的名称，方便status命令中查看统计

3、count

和Worker一样，可以设置Gateway进程的数量，以便充分利用多cpu资源

4、lanIp

lanIp是Gateway所在服务器的内网IP，只有在分布式部署时才需要设置

5、startPort

Gateway进程启动后会监听一个本机端口，用来给BusinessWorker提供链接服务，然后Gateway与BusinessWorker之间就通过这个连接通讯。这里设置的是Gateway监听本机端口的起始端口。比如启动了4个Gateway进程，startPort为2000，则每个Gateway进程分别启动的本地端口一般为2001、2003、2003、2004。

当本机有多个Gateway/BusinessWorker项目时，需要把每个项目的startPort设置成不同的段

6、心跳设置，具体说明见心跳一节

7、onWorkerStart

和Worker一样，可以设置Gateway进程启动后的回调函数，一般在这个回调里面初始化一些全局数据

8、onWorkerStop

和Worker一样，可以设置Gateway进程关闭的回调函数，一般在这个回调里面做数据清理或者保存数据工作

9、onConnect（比较少用到，开发者一般不用关注）

和Worker一样，可以设置onConnect回调，当有客户端连接上来时触发。与Event::onConnect的区别是Event::onConnect运行在BusinessWorker进程上。Gateway::onConnect是运行在Gateway进程上，无法使用\GatewayWorker\Lib\Gateway类提供的接口

10、onClose（比较少用到，开发者一般不用关注）

和Worker一样，可以设置onClose回调，当有客户端连接关闭时触发。同样与Event::onClose的区别是Gateway::onClose是运行在Gateway进程上，无法使用\GatewayWorker\Lib\Gateway类提供的接口

BusinessWorker类的使用

BusinessWorker类其实也是基于基础的Worker开发的。由于BusinessWorker进程中无法直接操作Gateway进程的连接，也就无法获得连接对象，所以无法使用BusinessWorker的onConnect、onMessage、onClose属性，请开发者不要设置以上回调属性。开发者仍然可以设置onWorkerStart、onWorkerStop属性。

BusinessWorker的相关回调函数在项目中的Event.php中定义，具体内容参见下一节

BusinessWorker类可以定制的内容

1、name

和Worker一样，可以设置BusinessWorker进程的名称，方便status命令中查看统计

2、count

和Worker一样，可以设置BusinessWorker进程的数量，以便充分利用多cpu资源

3、onWorkerStart

和Worker一样，可以设置BusinessWorker启动后的回调函数，一般在这个回调里面初始化一些全局数据

4、onWorkerStop

和Worker一样，可以设置BusinessWorker关闭的回调函数，一般在这个回调里面做数据清理或者保存数据工作

Event类的回调接口

Event::onConnect

说明:

```
void Event::onConnect(int $client_id);
```

当客户端连接上gateway进程时触发。

绝大多数应用不用实现这个函数。

参数

`$client_id` 全局唯一的客户端socket_id

返回值

无返回值，任何返回值都会被视为无效的

范例

```
use \GatewayWorker\Lib\Gateway;

public onConnect($client)
{
   Gateway::sendToCurrentClient('hello');
}
```

Event::onMessage

说明:

```
void Event::onMessage(int $client_id, mixed $recv_data);
```

当收到一个客户端请求后触发

参数

`$client_id`

全局唯一的客户端

`$recv_buffer`

完整的客户端请求数据，数据类型取决于Gateway所使用协议的decode方法返的回值类型

返回值

无返回值，任何返回值都会被视为无效的

范例

```
use \GatewayWorker\Lib\Gateway;

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

Event::onClose

说明:

```
void Event::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

返回值

无返回值，任何返回值都会被视为无效的

范例

```
use \GatewayWorker\Lib\Gateway;

/**
 * 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"));
}
```

Lib\Gateway类提供的接口

文件位置：GatewayWorker/Lib/Gateway.php

Lib\Gateway类是Gateway/BusinessWorker模型中给客户端发送数据的类。

提供了单发、群发以及关闭客户端连接的接口。

\GatewayWorker\Lib\Gateway::sendToAll

说明:

```
void Gateway::sendToAll(mixed $send_data [, array $client_id_array=array()]);
```

向所有客户端或者client_id_array指定的客户端发送`$send_data`数据。如果指定的$client_id_array中的client_id不存在则自动丢弃

参数

- `$send_data`

要发送的数据，此数据会被Gateway所使用协议的encode方法打包后发送给客户端

- `$client_id_array`

指定向哪些client_id发送，如果不传递该参数，则是向所有在线客户端发送`$send_data`数据

范例

```
use \GatewayWorker\Lib\Gateway;

class Event
{
...

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

\GatewayWorker\Lib\Gateway::sendToClient

说明:

```
void Gateway::sendToClient(int $client_id, mixed $send_data);
```

向客户端client_id发送`$send_data`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_data`

要发送的数据，此数据会被Gateway所使用协议的encode方法打包后再发送给客户端

范例

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

    public static function onMessage($client_id, $message)
    {
        // $message = '{"type":"say_to_one","to_client_id":100,"content":"hello"}'
        $req_data = json_decode($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']);
        }
    }

...
}
```

\GatewayWorker\Lib\Gateway::sendToCurrentClient

说明:

```
void Gateway::sendToCurrentClient(mixed $send_data);
```

作用与Gateway::sendToClient相同，只不过是只能给当前用户发送

\GatewayWorker\Lib\Gateway::closeClient

说明:

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

断开与client_id对应的客户端的连接

参数

- `$client_id`

全局唯一标识客户端连接的id

范例

```
use \GatewayWorker\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(!$is_ok)
        {
            Gateway::closeClient($client_id);
        }
    }

...
}
```

\GatewayWorker\Lib\Gateway::closeCurrentClient

说明:

```
void Gateway::closeCurrentClient();
```

作用与Gateway::closeClient相同，只不过是断开当前客户端的连接

\GatewayWorker\Lib\Gateway::isOnline

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 \GatewayWorker\Lib\Gateway;
class Event
{
...

    public static function onMessage($client_id, $message)
    {
        // $message = '{"type":"say_to_one","to_client_id":100,"content":"hello"}'
        $req_data = json_decode($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']);
            }
        }
    }

...
}
```

\GatewayWorker\Lib\Gateway::getOnlineStatus

说明:

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

获取当前所有在线client_id列表

范例

```
use \GatewayWorker\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,
);
```

router

说明:

```
callback Gateway::$router
```

设置Gateway到BusinessWorker路由规则。不设置默认是Gateway随机选择一个BusinessWorker进程把数据转发给它处理。

期待该回调函数从所有到BusinessWorker进程的连接对象中选择一个并返回。

回调函数的参数

`$worker_connnections`

是一个数组，里面包含了所有到BusinessWorker进程的连接对象。数组的key为BusinessWorker进程的通讯地址，格式为ip:port。回调函数最终将返回该数组中一个连接对象。

`$client_connection`

客户端连接对象,可以通过此对象获得客户端ip端口等信息

`$cmd`

当前什么类型的消息，是个数字，分别可能为

1: CMD_ON_CONNECTION，即连接事件

2: CMD_ON_MESSAGE，即消息事件

3: CMD_ON_CLOSE，即客户端关闭事件

`$buffer`

客户端发来的数据。注意只有当 `$cmd` 为 `2` 时 `$buffer` 才有值

返回值

返回 `$worker_connnections` 中的一个连接对象

范例 1 随机路由

```
use \GatewayWorker\Gateway;
$gateway = new Gateway("Websocket://0.0.0.0:8585");
// 随机路由
$gateway->router = function($worker_connections, $client_connection, $cmd, $buffer)
```

```
{
    return $worker_connections[array_rand($worker_connections)];
};
```

范例 2 随机绑定

```
use \GatewayWorker\Gateway;
$gateway = new Gateway("Websocket://0.0.0.0:8585");
$gateway->router = function($worker_connections, $client_connection, $cmd, $buffer)
{
    if(!isset($client_connection->businessworker))
    {
        $client_connection->businessworker = $worker_connections[array_rand($worker_conne
    }
    return $client_connection->businessworker;
};
```

超全局数组 $_SESSION

$_SESSION 是什么

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组中的数据，而不用去反复读取存储。

$_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。

$_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 中无法保存资源类型的数据
- 当客户端连接断开后，对应的客户端 $_SESSION 将会清除

$_SESSION 实现原理

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

$_SERVER是什么

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 // geteway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

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

$_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。

$_SERVER原理

在WorkerMan的Gateway/BusinessWorker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`数组便可以使用了。

服务端到客户端的心跳检测

WorkerMan支持服务端与客户端定时发送心跳检测

为什么需要心跳检测？

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。

什么情况需要定时心跳检测？

一般的应用其实不需要心跳检测，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，则需要服务端与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？

目前只有Gateway/BusinessWorker模型实现了应用层心跳检测，Worker默认没有应用层的心跳检测，如有需要可以自行开发。

例子：

心跳检测在初始化Gateway时设置，例如

```
$gateway = new Gateway("Websocket://0.0.0.0:8585");

$gateway->pingInterval = 10;

$gateway->pingNotResponseLimit = 2;

$gateway->pingData = '{"type":"ping"}';
```

说明：

`Gateway::$pingInterval`

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

`Gateway::$pingNotResponseLimit`

客户端连续$pingNotResponseLimit次$pingInterval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）

`Gateway::$pingData`

要发送的心跳请求数据，心跳数据是任意的，只要客户端能识别即可。

技巧1

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`pingData=''`,例如如下配置

```
$gateway = new Gateway("Websocket://0.0.0.0:8585");

$gateway->pingInterval = 10;

$gateway->pingNotResponseLimit = 2;

$gateway->pingData = '';
```

代表服务端不发送任何心跳数据，但是客户端如果 `pingInterval*pingNotResponseLimit=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。

```
$gateway = new Gateway("Websocket://0.0.0.0:8585");

$gateway->pingInterval = 10;

$gateway->pingNotResponseLimit = 0;

$gateway->pingData = '{"type":"ping"}';
```

其中`pingNotResponseLimit = 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。

分布式部署

WorkerMan分布式的好处

（只针对Gateway/Worker模型）

1、成倍提高系统承载能力并降低成本

单机遇到资源瓶颈时，要想单机支持更大的用户量，一般是优化业务和增加服务器配置。然而这么做只能是杯水车薪，成本巨大并且效果非常有限。

WorkerMan支持分布式部署，你可以利用多台价格低廉的普通服务器，组成一个庞大的服务器集群，成倍的增加系统承载能力，这不管在资金成本上还是人力成本上都是最划算的方案。

2、提高系统稳定性

单机对外提供服务，则风险很大，服务器任何故障都可能引起整个服务的不可用。

WorkerMan分布式可以有效的降低这个风险，如果一台服务器故障宕机，还有其它服务器可以继续工作，可以做到对服务无影响或者影响最小化。例如WorkerMan中一台Gateway服务器宕机，可以利用LVS健康探测等技术立刻踢掉故障ip，集群立刻恢复服务。如果WorkerMan中任意一台Worker机器宕机，则GateWay会立刻踢掉故障Worker机器，做到对外网服务几乎无影响。

3、平滑过渡

请求量突然增大，系统已经无法支撑，而你却束手无策。

WorkerMan分布式可以让你从容应对，只需要再启动几台WorkerMan服务器，便可以让你的系统增加几倍的承载能力，轻松应对突发流量。等请求量降下去时，你可以为降低成本将服务器回收，而不影响任何用户。

如何分布式WorkerMan

关键点

1、设置Gateway实例的 `lanIp` 与当前服务器内网ip一致

2、 `Applications/XXX/Config/Store.php` 中 redis 相关配置

```
// self::DRIVER_REDIS代表使用redis存储
public static $driver = self::DRIVER_REDIS;
// 改成redis服务器内网ip端口
public static $gateway = array(
     'redis内网ip:redis端口',
);
```

部署示例

以Applications/Demo为例，假如需要部署三台服务器(192.168.1.1-3)提供高可用服务。。另外有一台redis服务器（ip 192.168.1.4，端口6379）做全局数据共享。

1、给三台服务器的PHP添加redis扩展。ubuntu/debian可使用 `sudo apt-get install php5-redis` 安装。centos系统使用 `yum install php-pecl-redis` 。

2、配置三台服务器 `Applications/Demo/Config/Store.php` 如下

```
// 存储驱动改为redis
public static $driver = self::DRIVER_REDIS
// 更改redis ip和端口
public static $gateway = array(
     '192.168.1.4:6379',
);
// 如果有其它的配置如workerman-chat中的$room配置，也需要将其改成redis的ip和端口
...
```

3、分别配置三台服务器Gateway对象的 `lanIp` 为当前服务器的内网ip。例如配置192.168.1.1服务器Gateway实例

Applications/Demo/start.php中设置

```
$gateway = new Gateway("yourProtocol://0.0.0.0:your_port");
....
$gateway->lanIp = 192.168.1.1;
....
```

4、逐台启动WorkerMan，至此WorkerMan分布式部署完毕。

说明：

1、三台WorkerMan机器都运行了Gateway进程和Worker进程，客户端连接上任意一台WorkerMan的Gateway端口即可。

2、为了方便前端接入和扩容，可以在Gateway前加一层DNS、LVS等负载均衡策略

3、如果服务器不够用可以使用同样的方法增加服务器

4、如果需要下线服务器，可以停止WorkerMan，然后执行后续停机等下线操作(由于Gateway进程维护着客户端连接，当对应服务器下线时，对应服务器的客户端会掉线一次。如何做到下线机器不影响用户参考下一节)。

gateway worker 分离部署

什么是Gateway Worker分离部署

Gateway/Worker模式有两组进程，Gateway进程负责网络IO，Worker进程负责业务处理，Gateway与Worker之间使用TCP长连接通讯。当系统出现负载时，一般都是业务进程Worker出现瓶颈。我们可以把Gateway Worker分开部署在不同的服务器上，单独增加Worker服务器提升系统负载能力。同理，如果Gateway进程出现瓶颈，则增加Gateway服务器。

部署示例

以Applications/Todpole为例，假如需要部署三台服务器提供高可用服务。瓶颈在worker进程，则可使用1台作为gateway服务器，另外两台做worker服务器。（如果瓶颈在gateway进程（一般是带宽瓶颈），则可以2台gateway机器，1台worker机器，部署方法类似）。

gateway worker 分离部署扩容步骤

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上的初始化BusinessWorker示例的地方注释或者删掉，避免运行BusinessWorker进程，例如

打开文件Applications/Todpole/start.php，注释掉bussinessWorker初始化

```
...
// bussinessWorker
//$worker = new BusinessWorker();
//$worker->name = 'TodpoleBusinessWorker';
//$worker->count = 4;
...
```

3、配置Gateway服务器(192.168.0.1)上的Gateway实例的 `lanIp=192.168.0.1` 与本机ip一致，Gateway服务器的初始化文件最终类似下面配置(如果有单独的Web服务器运行蝌蚪界面，可以把WebServer初始化部分也去掉)

文件Applications/Todpole/start.php

```
<?php
use \Workerman\WebServer;
use \GatewayWorker\Gateway;
use \GatewayWorker\BusinessWorker;

// gateway
$gateway = new Gateway("Websocket://0.0.0.0:8282");
$gateway->name = 'TodpoleGateway';
```

```
$gateway->count = 4;
$gateway->lanIp = '192.168.0.1';
$gateway->startPort = 2000;
$gateway->pingInterval = 10;
$gateway->pingData = '{"type":"ping"}';

// bussinessWorker
//$worker = new BusinessWorker();
//$worker->name = 'TodpoleBusinessWorker';
//$worker->count = 4;

// WebServer
$web = new WebServer("http://0.0.0.0:8383");
$web->count = 12;
$web->addRoot('kedou.workerman.net', __DIR__.'/Web');
```

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Gateway初始化注释掉或者删掉，避免运行Gateway进程，BusinessWorker服务器初始化文件类似下面(如果有单独的Web服务器运行蝌蚪界面，可以把WebServer初始化部分也去掉)

```
<?php
use \Workerman\WebServer;
use \GatewayWorker\Gateway;
use \GatewayWorker\BusinessWorker;

// gateway
//$gateway = new Gateway("Websocket://0.0.0.0:8282");
//$gateway->name = 'TodpoleGateway';
//$gateway->count = 4;
//$gateway->lanIp = '192.168.0.1';
//$gateway->startPort = 2000;
//$gateway->pingInterval = 10;
//$gateway->pingData = '{"type":"ping"}';

// bussinessWorker
$worker = new BusinessWorker();
$worker->name = 'TodpoleBusinessWorker';
$worker->count = 4;

// WebServer
$web = new WebServer("http://0.0.0.0:8383");
$web->count = 12;
$web->addRoot('kedou.workerman.net', __DIR__.'/Web');
```

4、由于物理机之间需要共享一些数据，需要部署一台redis服务器，假设部署在Gateway（192.168.0.1）这台机器上，redis服务端口为6379

5、给三台服务器的PHP添加redisd或者redis扩展。推荐用redisd扩展，ubuntu/debian可使用 `sudo apt-get install php5-redis` 安装；centos系统使用 `yum install php-pecl-redis` 安装。

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

```
// 存储驱动改为redis
public static $driver = self::DRIVER_REDIS
// 更改redis ip和端口
public static $gateway = array(
     '192.168.0.1:6379',
);
```

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

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，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服务器则一般不需要变动，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。

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`php start.php 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`php start.php 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。

Config/Store 配置

配置作用

配置Gateway/Worker模型中存储驱动类型及存储位置，以便Store类存储Gateway与BusinessWorker之间的通讯ip与端口，以及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;
    // memcache存储
    const DRIVER_MC = 2;
    // redis存储
    const DRIVER_REDIS = 3;

    /* 使用哪种存储驱动 文件存储DRIVER_FILE 或者 memcache存储DRIVER_MC或者redis存储DRIVER_REDIS,
     */
    public static $driver = self::DRIVER_FILE;

    // 如果是memcache/redis存储，则在这里设置memcache/redis的ip端口，注意确保你安装了memcache/red
    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（$driver = self::DRIVER_FILE），则在这里设置数据存储的目录，默认/tmp/下
     */
    public static $storePath = '/tmp/workerman-chat/';
}
```

1、Store类有三种存储驱动，文件存储（DRIVER_FILE）及memcache存储(DRIVER_MC)以及redis存储(DRIVER_REDIS)。正式环境建议使用redis存储。

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。

3、如果是memcache/redis存储，请安装memcached/redis服务端及php的memcached/redis扩展。并设定`$gateway` `$room`（如果有的话）中的ip和端口为memcache/redis服务端的ip端口。

4、如果是memcache/redis存储，并且业务想添加自己的存储实例，可以开启新的memcache/redis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/redis 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

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

请开发者注意

1、线上环境请使用redis存储，即设置

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

2、开发者不要使用`Store::instance('gateway');`、`Store::instance('room');`这两个实例，也要避免业务新增配置memcache/redis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。

3、redis扩展安装方法

centos系统：yum install php-pecl-redis

ubuntu/debian系统：apt-get install php5-redis

pecl 命令安装：pecl install redis

其他方法参见手册11.2安装扩展章节

调试

基本调试

WorkerMan3.0有两种运行模式，调试模式以及daemon运行模式

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

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

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

```
use Workerman\Worker;

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

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

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

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 35242
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 [
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 25
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: zPDr6m4czzUdOFnsxIUEAw==
```

```
Cookie: Hm_lvt_abcf9330bef79b4aba5b24fa373506d9=1402048017; Hm_lvt_5fedb3bdce89499492c079
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
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
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
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","na
 */
17:50:30.652680 IP 127.0.0.1.60653 > 127.0.0.1.7272: Flags [P.], seq 716:754, ack 157, wi
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, wi
E...(w@.@............h..)I.#..l...........
...^...^.l{"type":"user_list","user_list":[{"uid":783654164,"name":"\u732a\u732a"},{"uid"

/*
 * 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
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, wi
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
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, wi
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
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, wi
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, wi
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
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, wi
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
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 php start.php status 能看到workerman相关进程的信息 如下：

```
Hello admin
------------------------------------GLOBAL STATUS------------------------------------
WorkerMan version:3.0.1
start time:2014-08-12 17:42:04   run 0 days 1 hours
load average: 3.34, 3.59, 3.67
1 users          8 workers       14 processes
worker_name       exit_status     exit_count
BusinessWorker    0                 0
ChatWeb           0                 0
FileMonitor       0                 0
Gateway           0                 0
Monitor           0                 0
StatisticProvider 0                 0
StatisticWeb      0                 0
StatisticWorker   0                 0
------------------------------------PROCESS STATUS-----------------------------------
pid    memory      listening         timestamp  worker_name       total_request packet_err
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

高级应用

WebServer

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

使用方法

在Applications/YourApp/start.php中添加

```
use \Workerman\WebServer;

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

WorkerMan的Webserver与普通Web开发异同

1、普通Web程序架构运行机制

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

2、WorkerMan架构Web程序运行机制

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

3、避免使用exit、die语句

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

4、HTTP相关函数的使用

WorkerMan运行在PHP CLI模式下，PHP CLI模式下无法使用HTTP相关的函数，例如 `header、setcookie、session_start` 等函数，请使用 `/Workerman/Protocols/Http.php` 文件中的 `header、`

`setcookie、session_start`等方法替换。

5、Web入口文件

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

6、可用的超全局变量

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

其中$_FILES结构类似

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

保存文件代码类似

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

7、可以设置onWorkerStart、onWorkerStop回调

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

查看运行状态

运行 `php start.php status` 可以查看到WorkerMan的运行状态，类似如下:

```
---------------------------------------GLOBAL STATUS--------------------------------------
Workerman version:3.0.3          PHP version:5.3.29-1~dotdeb.0
start time:2014-07-21 18:05:47   run 86 days 22 hours
load average: 0, 0, 0
3 workers       10 processes
worker_name           exit_status     exit_count
TodpoleGateway        0                0
TodpoleBusinessWorker 0                4
TodpoleBusinessWorker 9                1
WebServer             0                2
---------------------------------------PROCESS STATUS-------------------------------------
pid     memory  listening                worker_name           connections total_request
936     2.15M   Websocket://0.0.0.0:8585 TodpoleGateway        13          355
937     2.03M   Websocket://0.0.0.0:8585 TodpoleGateway        5           181
938     2M      Websocket://0.0.0.0:8585 TodpoleGateway        4           171
939     2.03M   Websocket://0.0.0.0:8585 TodpoleGateway        5           177
948     2.15M   none                     TodpoleBusinessWorker 4           32
949     2.16M   none                     TodpoleBusinessWorker 4           54
953     2.16M   none                     TodpoleBusinessWorker 4           50
957     2.15M   none                     TodpoleBusinessWorker 4           53
954     1.84M   http://0.0.0.0:8686      WebServer             0           61
955     1.84M   http://0.0.0.0:8686      WebServer             1           59
```

说明

GLOBAL STATUS

从这以栏中我们可以看到

WorkerMan的版本 `version:3.0.3`

启动时间 `2015-02-21 18:05:47`，运行了 `run 6 days 22 hours`

服务器负载 `load average: 0, 0, 0`

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

`14 processes` (共10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）

其中exit_status为0为正常退出，如果为其它值，代表进程是异常退出的，比如exit_status为65280，exit_count为60，代表业务代码有FatalError，导致进程退出60次，需要根据php的错误日志查找FatalError原因

PROCESS STATUS

pid：进程pid

memory：该进程占用内存（不包括php自身可执行文件的占用的内存）

listening：传输层协议及监听ip端口

timestamp：该进程启动时间戳

worker_name：该进程运行的服务服务名

connections:该进程当前有多少个TCP连接，包括客户端连接和WorkerMan内部通讯的连接

total_request：该进程接收多少请求，注意每个连接上可能有多个请求

send_fail：该进程向客户端发送数据失败次数，失败原因一般为客户端连接断开

throw_exception：该进程内业务未捕获的异常数量

在非WorkerMan项目中推送消息

需求

有时候需要在非WorkerMan环境中向客户端推送数据。例如在一个普通的Web项目中通过WorkerMan推送数据（前提是已经部署了WorkerMan，客户端已经连接WorkerMan）。目前有三种比较方便方法推送数据。（本节内容主要是针对Gateway/Worker模型的推送方法的讲解）

方法一、使用GatewayClient客户端推送

客户端地址：

https://github.com/walkor/GatewayClient

使用方法：

*注意:如果项目与WorkerMan不在同一台服务器，需要安装redis，参考8.10 Config/Store配置章节。

1、拷贝WorkerMan项目中的Applications/YourApp/Config目录到GatewayClient下。

拷贝后的GatewayClient目录结构如下

```
GatewayClient/
├── Config
│   └── Store.php
└── Gateway.php
```

2、引入GatewayClient/Gateway.php文件开始使用。接口使用方法与\GatewayWorker\Lib\Gateway接口相同，接口说明参见8.5章节。

客户端使用示例

```
require_once '/your/path/GatewayClient/Gateway.php';

Gateway::sendToAll('{"type":"broadcast","content":"hello all"}');

Gateway::sendToClient($client_id,'{"type":"say","content":"hello"}');

Gateway::isOnline($client_id);

...
```

方法二、用一个特殊的账号当做管理客户端，通过这个账号推送数据

此方法通俗易懂，可以通过现有客户端直接操作，具体代码根据自己的业务实现

方法三、开启一个内部Gateway端口，用于推送数据

方法二虽然简单，但是局限于只能通过客户端界面操作，定时推送等需求不好直接操作客户端，而通过PHP模拟客户端可能会受到复杂协议的限制不好操作，这时我们可以开启一个内部文本协议的Gateway端口，通过PHP代码使用文本协议连接WorkerMan作为客户端向其它客户端推送数据。

示例（**workerman-chat**为例）

服务端：Applications/Chat/start.php中新增一个文本协议Gateway端口

```
// gateway
$gateway = new Gateway("Websocket://0.0.0.0:7272");
$gateway->name = 'ChatGateway';
$gateway->count = 4;
$gateway->lanIp = '127.0.0.1';
$gateway->startPort = 2500;
$gateway->pingInterval = 10;
$gateway->pingData = '{"type":"ping"}';

// #### 内部推送端口(假设内网ip为192.168.100.100) ####
$internal_gateway = new Gateway("Text://192.168.100.100:7273");
$internal_gateway->name='internalGateway';
$internal_gateway->startPort = 2800;
// #### 内部推送端口设置完毕 ####

// bussinessWorker
$worker = new BusinessWorker();
$worker->name = 'ChatBusinessWorker';
$worker->count = 4;

// WebServer
$web = new WebServer("http://0.0.0.0:55151");
$web->count = 2;
$web->addRoot('www.workerman.net', __DIR__.'/Web');
```

客户端：在其它项目中就可以直接用PHP socket 使用文本协议调用，代码类似如下:

```
// 建立连接，@see http://php.net/manual/zh/function.stream-socket-client.php
$client = stream_socket_client('tcp://192.168.100.100:7273');
if(!$client)exit("can not connect");
// 模拟超级用户，以文本协议发送数据，注意文本协议末尾有换行符（发送的数据中最好有能识别超级用户的字段），这
fwrite($client, '{"type":"send","content":"hello all", "user":"admin", "pass":"******"}'.
```

多协议支持

需求

有时我们需要一套应用程序支持多个客户端，进而需要应用支持多个协议。例如一个IM即时通讯应用，可能需要同时支持浏览器使用，又要支持移动App客户端。而二者所使用的协议可能完全不同。

如何支持多协议

在WorkerMan中最简单的实现方法是开启多个端口，每个端口使用一种协议。不同客户端使用各自的协议去连特定的端口。

示例（小蝌蚪）

小蝌蚪应用程序是运行在PC浏览器里面的，使用Websocket协议与WorkerMan通讯，当我们需要把它移植到手机App上却有没有合适的客户端Websocket库时，我们可以使用更简单的协议来实现App与WorkerMan通讯，例如Text文本协议(以换行符为结尾的文本数据包)。

下面是开启多端口支持多协议示例

```
use \Workerman\WebServer;
use \GatewayWorker\Gateway;
use \GatewayWorker\BusinessWorker;

// gateway 原有代码
$gateway = new Gateway("Websocket://0.0.0.0:8282");
$gateway->name = 'TodpoleGateway';
$gateway->count = 4;
$gateway->lanIp = '127.0.0.1';
$gateway->startPort = 2000;
$gateway->pingInterval = 10;
$gateway->pingData = '{"type":"ping"}';

// ##########新增端口支持Text协议 开始##########
// 新增8283端口，开启Text文本协议
$gateway_text = new Gateway("Text://0.0.0.0:8283");
// 进程名称，主要是status时方便识别
$gateway_text->name = 'TodpoleGatewayText';
// 开启多少text协议的gateway进程
$gateway_text->count = 4;
// 本机ip（分布式部署时需要设置成内网ip）
$gateway_text->lanIp = '127.0.0.1';
// gateway内部通讯起始端口，起始端口不要重复
$gateway_text->startPort = 2500;
// 也可以设置心跳，这里省略
// ##########新增端口支持Text协议 结束##########

// bussinessWorker 原有代码
$worker = new BusinessWorker();
$worker->name = 'TodpoleBusinessWorker';
```

```
$worker->count = 4;

// WebServer 原有代码
$web = new WebServer("http://0.0.0.0:8383");
$web->count = 2;
$web->addRoot('www.workerman.net', __DIR__.'/Web');
```

测试效果

由于是文本协议，我们可以通过telnet命令方便的模拟文本协议客户端。以下运行telnet命令的结果

```
telnet 127.0.0.1 8283
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
{"type":"update","id":156,"angle":3.636,"momentum":0,"x":-64.8,"y":147.1,"life":1,"name":
{"type":"update","id":156,"angle":4.27,"momentum":0,"x":-64.8,"y":147.1,"life":1,"name":"
{"type":"update","id":156,"angle":5.766,"momentum":0,"x":-64.8,"y":147.1,"life":1,"name":
{"type":"update","id":156,"angle":6.284,"momentum":3,"x":-58.8,"y":146.7,"life":1,"name":
```

我们能看到其它PC客户端通过WorkerMan转发来的蝌蚪的实时坐标数据，我们也可以输入自己的坐标数据，然后按回车键，我们就能在PC客户端上看到自己了。这样通过使用telnet客户端+文本协议，我们可以方便的调试数据，开发新的客户端了。

说明：

以上是WorkerMan多协议支持示例，我们看到只需要简单的初始化端口及协议即可，服务端的业务代码不用任何更改。开发者也可以使用其它协议初始化端口，也可以参考《定制通讯协议章节》定义自己的协议

以上是Gateway/Worker模型的多协议支持示例，基于Worker的多协议也是同样的道理

支持多协议还有其他的方法，比如通过协议自身的特点区分当前是哪种协议，然后分别调用相应协议的解码方法，这样可以做到只开一个端口就可以支持多种协议的效果

附录

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Benchark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：2.3W/S ， cpu利用率：36%

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
    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

压力测试需要内核调优

参见 附录-内核调优 章节

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

安装扩展

注意

与Apache+PHP或者Nginx+PHP的运行模式不同，WorkerMan是基于命令行 PHP Cli 运行的，使用的是不同的PHP可执行程序，使用的php.ini文件也可能不同。所以在网页中打印 `phpinfo()` 看到安装了某个扩展，不代表命令行的PHP Cli也安装了对应的扩展。

如何确定PHP Cli安装了哪些扩展

运行 `php -m` 会列出命令行 PHP Cli 已经安装的扩展，结果类似如下：

```
~# php -m
[PHP Modules]
libevent
posix
pcntl
...
```

如何确定PHP Cli 的php.ini文件的位置

当我们安装扩展时，可能需要手动配置php.ini文件，把扩展加进去，所以要确认PHP Cli的php.ini文件的位置。可以运行 `php --ini` 查找PHP Cli的ini文件位置，结果类似如下：

```
~# php --ini
Configuration File (php.ini) Path: /etc/php5/cli
Loaded Configuration File:         /etc/php5/cli/php.ini
Scan for additional .ini files in: /etc/php5/cli/conf.d
Additional .ini files parsed:      /etc/php5/cli/conf.d/apc.ini,
/etc/php5/cli/conf.d/libevent.ini,
/etc/php5/cli/conf.d/memcached.ini,
/etc/php5/cli/conf.d/mysql.ini,
/etc/php5/cli/conf.d/pdo.ini,
/etc/php5/cli/conf.d/pdo_mysql.ini
...
```

给PHP Cli安装扩展（安装memcached扩展为例）

方法一、使用apt或者yum命令安装

如果PHP是通过 apt 或者 yum 命令安装的，则扩展也可以通过 apt 或者 yum 安装

debian/ubuntu等系统apt安装PHP扩展方法（非root用户需要加sudo命令）

1、利用`apt-cache search`查找扩展包

```
~# apt-cache search memcached php
php-apc - APC (Alternative PHP Cache) module for PHP 5
php5-memcached - memcached module for php5
```

2、使用`apt-get install`安装扩展包

```
~# apt-get install -y php5-memcached
Reading package lists... Done
Reading state information... Done
...
```

centos等系统yum安装PHP扩展方法

1、利用`yum search`查找扩展包

```
~# yum search memcached php
php-pecl-memcached - memcached module for php5
```

2、使用`yum install`安装扩展包

```
~# yum install -y php-pecl-memcached
Reading package lists... Done
Reading state information... Done
...
```

说明：

使用apt或者yum安装PHP扩展会自动配置php.ini文件，安装完直接可用，十分方便。缺点是有些扩展在apt或者yum中没有对应的扩展安装包。

方法二、使用**pecl**安装

使用`` `pecl install`` 命令安装扩展

1、 `pecl install`安装

```
~# pecl install memcached
downloading memcached-2.2.0.tgz ...
Starting to download memcached-2.2.0.tgz (70,449 bytes)
....
```

2、配置php.ini

通过运行 `php --ini` 查找php.ini文件位置，然后在文件中添加 `extension=memcached.so`

方法三、源码编译安装（一般是安装PHP自带的扩展，以安装posix扩展为例）

1、利用 `php -v` 命令查看当前的PHP Cli的版本

```
~# php -v
PHP 5.3.29-1~dotdeb.0 with Suhosin-Patch (cli) (built: Aug 14 2014 19:55:20)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2014 Zend Technologies
```

2、根据版本下载PHP源代码

PHP历史版本下载页面：http://php.net/releases/

3、解压源码压缩包

例如下载的压缩包名称是 `php-5.3.29.tar.gz`

```
~# tar -zxvf php-5.3.29.tar.gz
php-5.3.29/
php-5.3.29/README.WIN32-BUILD-SYSTEM
php-5.3.29/netware/
...
```

4、进入源码中的ext/posix目录

```
~# cd php-5.3.29/ext/posix/
```

5、运行 `phpize` 命令

```
~# phpize
Configuring for:
PHP Api Version:         20090626
Zend Module Api No:      20090626
Zend Extension Api No:   220090626
```

6、运行 `configure` 命令

```
~# ./configure
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
...
```

7、运行 `make` 命令

```
~# make
/bin/bash /tmp/php-5.3.29/ext/posix/libtool --mode=compile cc ...
-I/usr/include/php5 -I/usr/include/php5/main -I/usr/include/php5/TSRM -I/usr/include/php5
...
```

8、运行`make install`命令

```
~# make install
Installing shared extensions:     /usr/lib/php5/20090626/
```

9、配置ini文件

通过运行`php --ini`查找php.ini文件位置，然后在文件中添加`extension=posix.so`

说明：此方法一般用来安装PHP自带的扩展，例如posix扩展和pcntl扩展。除了用phpize编译某个扩展，也可以重新编译整个PHP，在编译时用参数添加扩展，例如在源码根目录运行

```
~# ./configure --enable-pcntl --enable-posix ...
~# make && make install
```

Linux内核调优

网络

调整linux内核参数以便满足高并发访问，解决大量time_wait占用太多本地端口导致的 `Cannot assign requested address` 问题。

客户端与服务端每建立一个连接，客户端一侧都会占用一个本地端口（假设没有启用SO_REUSEADDR选项），本地端口数量是有限制的（默认是 `net.ipv4.ip_local_port_range=32768 61000` ），也就是说在没设置socket的SO_REUSEADDR选项时，一台Linux服务器作为客户端（ `注意是作为客户端` ）默认只能建立大概3万个TCP连接（服务端没有这个限制），可以更改 `net.ipv4.ip_local_port_range` 增大作为客户端可发起的并发连接数，但最多不会超过65535个（服务端没有这个限制）。

当Linux服务器作为客户端频繁建立TCP短连接时，本地会可能会产生很多TIME_WAIT状态的连接，客户端侧的TIME_WAIT状态的连接会占用一个本地端口直到达到2MSL（最长分解生命期）的时间，这样会导致本地端口被暂时占用，当短连接建立速度过快时（例如做压测时），会导致 `Cannot assign requested address` 错误，解决办法有几种，比如像下面这样设置端口复用（复用TIME_WAIT状态的连接）。

打开文件 /etc/sysctl.conf，增加以下设置

```
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
```

运行 `sysctl -p` 即可生效

说明：

/etc/sysctl.conf 可设置的选项很多，其它选项可以根据自己的环境需要进行设置

打开文件数

设置系统打开文件数设置，解决高并发下 `too many open files` 问题。此选项直接影响单个进程容纳的客户端连接数。

Soft open files 是Linux系统参数，影响系统单个进程能够打开最大的文件句柄数量，这个值会影响到长链接应用如聊天中单个进程能够维持的用户连接数， 运行 `ulimit -n` 能看到这个参数值，如果是1024，就是代表单个进程只能同时最多只能维持1024甚至更少（因为有其它文件的句柄被打开）。如果开启4个进程维持用户链接（例如gateway进程），那么整个应用能够同时维持的连接数不会超过4*1024个，也就是说最多只能支持4x1024个用户在线可以增大这个设置以便服务能够维持更多的TCP连接。

Soft open files 修改方法：

（1）ulimit -HSn 102400

这只是在当前终端有效，退出之后，open files 又变为默认值。

（2）将ulimit -HSn 102400写到/etc/profile中，这样每次登录终端时，都会自动执行/etc/profile。

（3）令修改open files的数值永久生效，则必须修改配置文件：/etc/security/limits.conf. 在这个文件后加上： soft nofile 102400 hard nofile 102400

这种方法需要重启机器才能生效。

Gateway/Worker模型 数据库使用示例

说明

常驻内存的程序在使用mysql时经常会遇到 `mysql gone away` 的错误，这个是由于程序与mysql的连接长时间没有通讯，连接被mysql服务端踢掉导致。workerman提供了一个mysql类，可以解决这个问题，当发生 `mysql gone away` 错误时，会自动重试一次。

注意

不要直接在 `sart.php` 直接使用这个mysql类，会导致错误。请在 `onXXXX` 回调中使用这个数据库类。

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>
     * 等价于
     * $user_array = Db::instance('user')->query('SELECT `name`,`age` FROM `users` WHERE
     * @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 \GatewayWorker\Lib\Gateway;
use \GatewayWorker\Lib\Db;

/**
 * 数据库示例，假设有个user库，里面有个user表
 */
class Event
```

```
{

    /**
     * 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
        $commend = trim($message);
        if($commend !== 'get_user_list')
        {
            Gateway::sendToClient($client_id, "unknown commend\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));
    }

}
```

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(
        'host'    => '127.0.0.1',
        'port'    => 3306,
        'user'    => 'mysql_user',
        'password' => 'mysql_password',
        'dbname'  => 'db2',
```

```
        'charset'    => 'utf8',
    );
}
```

使用方法

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

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

// 获取单个值
$db1->select('ID,Sex')->from('Persons')->where('sex= :sex')->bindValues(array('sex'=>'M')
//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('
y('age' => 13));
// 等价于
$db1->query(SELECT * FROM `table1` INNER JOIN `table2` ON `table1`.`uid` = `table2`.`uid`

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

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

```php
$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");
```

常见问题

运行多个WorkerMan实例

可以运行多个WorkerMan实例，一般只要端口不同即可（注意使用Gateway/Worker时，需要注意Config/Store.php配置不能相同）。

虽然可以运行多个WorkerMan实例，但是在一个实例上运行多个应用更方便，即将所有应用放置于Applications下运行。

WorkerMan支持哪些协议

WorkerMan在接口上支持各种协议，只要符合 `ConnectionInterface` 接口即可（参见定制通讯协议章节）。

为了方便开发者，WorkerMan实现了HTTP协议、WebSocket协议以及非常简单的Text文本协议。开发者可以直接使用这些协议，不必再二次开发。如果这些协议都不满足需要，开发者可以参照定制协议章节实现自己的协议。

应该开启多少进程

如何设置进程数

进程数是由`count`属性决定的，例如下面代码

```
use Workerman\Worker;
$http_worker = new Worker("http://0.0.0.0:2345");

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

进程数设置需要考虑以下条件

1、cpu核数

2、内存大小

3、业务属于IO密集型还是CPU密集型

进程数设置原则

1、每个进程占用内存之和需要小于总内存（一般来说每个业务进程占用内存大概40M左右）

2、如果是IO密集型，也就是业务中涉及到一些阻塞式网络通讯开销，比如一般的访问Mysql、Redis等存储都是阻塞式访问的，进程数可以开大一些，如配置成CPU核数的3倍。如果业务中涉及的阻塞网络开销很多，可以再适当加大进程数，例如CPU核数的5倍甚至更高。

3、如果是CPU密集型，也就是没有外部网络IO开销，或者没有阻塞的网络IO开销，例如使用异步IO读取网络资源，进程不会被业务代码阻塞的情况下，可以把进程数设置成和CPU核数一样

进程数设置原理有点像生产线上工人们工作的情景，有4个生产线，就相当于CPU是4核的，生产线上的产品相当于任务，工人相当于进程。

工人并非越多越好，因为只有4条生产线，生产能力有限，并且工人多了大家轮流上下生产线开销也很大，同样的进程多了进程间切换开销也很大。

如果流水线上的商品每一步操作起来比较耗时（IO密集型），就需要多一点的工人去操作。

如果流水线上的商品每一步操作都比较快（CPU密集型），只需要少数工人即可

如果是基于Worker开发

如果是基于Worker开发，业务代码是IO密集型的（业务代码有IO阻塞的地方），则根据IO密集程度设置进

程数，例如CPU核数的3倍。

如果业务代码中无阻塞式IO通讯，则可以将进程数设置成cpu核数

如果是基于Gateway/Worker开发

Gateway/Worker模型与Nginx+PHP-FPM模型非常相像，Gateway相当于Nginx，Worker相当于PHP-FPM。进程数设置原则也类似。

Gateway进程虽然处理大量IO操作，但是由于Gateway进程的IO都是非阻塞的，所以它不是IO密集型进程，而是CPU密集型，将进程数设置成CPU核数即可

Worker进程是处理业务的，根据业务是否有IO阻塞操作以及阻塞程度设置进程数

注意

WorkerMan自身的IO都是非阻塞的，例如 `Connection->send`、`Gateway::sendToClient`、`Gateway::sendToAll` 等都是非阻塞的，属于CPU密集型操作。如果不清楚自己业务偏向于哪种类型，可设置进程数为CPU核数的2倍左右即可（注意Gateway为CPU密集型进程，设置与CPU核数相同）。

查看当前客户端连接数

运行 `php start.php status` 能看到当前服务器的WorkerMan运行的状态， `connections` 字段标记了每个进程当前TCP连接数。需要注意的是这个字段不仅包括客户端的TCP连接数，也包括WorkerMan内部通讯的TCP连接数。例如WorkerMan中的Gateway/Worker模型中，每个Gateway进程当前的客户端连接数为 `connections` 字段的值减去Worker进程数。

另外在Gateway/Worker模型中，可以调用 `Gateway::getOnlineStatus` 以数组的形式返回WorkerMan集群当前所有在线客户端的client_id，数组元素个数即为当前WorkerMan集群的所有客户端连接数。

对象和资源的持久化

在传统的Web开发中，PHP创建的对象、数据、资源等会在请求完毕后全部释放，导致很难做到持久化。而在WorkerMan中可以轻松做到这些。

在WorkerMan中如果想在内存中永久保存某些数据资源，可以将资源放到全局变量中或者类的静态成员中。

例如下面的代码：

用一个全局变量`$connection_count`保存一个当前进程的客户端连接数。

```
<?php
use \Workerman\Worker;

// 全局变量，保存当前进程的客户端连接数
$connection_count = 0;

$worker = new Worker('tcp://0.0.0.0:1236');

$worker->onConnect = function($connection)
{
    // 有新的客户端连接时，连接数+1
    global $connection_count;
    ++$connection_count;
    echo "now connection_count=$connection_count\n";
};

$worker->onClose = function($connection)
{
    // 客户端关闭时，连接数-1
    global $connection_count;
    $connection_count--;
    echo "now connection_count=$connection_count\n";
};
```

PHP变量作用域参见：

http://php.net/manual/zh/language.variables.scope.php