# IM 系统开发文档

Briquz zgy@bz-inc.com

2015 年 9 月 26 日

version v1

# 目录

# 插图

# 表格

# Part I

# Swoole

# Chapter 1

# Introduction

Swoole 是一种基于 PHP 核心开发的高性能网络通信框架[1]，提供了 PHP 语言的异步多线程服务器，异步 TCP/UDP 网络客户端，异步 MySQL，数据库连接池，AsyncTask，消息队列，毫秒定时器，异步文件读写，异步 DNS 查询。

- swoole_server，高并发高性能功能强大的异步并行 TCP/UDP Server。
- swoole_client，支持同步/异步/并发的 socket 客户端实现[2]。
- swoole_event，基于 epoll/kqueue 的全自动 IO 事件发生器。
- swoole_task，基于进程池实现的异步任务处理器。

其中，swoole_event 比 libevent 更简单，仅需 add/set/del 几个操作即可，这样使用者就可以将原有 PHP 代码中的 streams/fsockopen/sockets 代码加入到 swoole 实现异步化，而且利用 swoole_event 还可以实现真正的 PHP 异步 MySQL。

用户可以使用 swoole_task 实现 PHP 的数据库连接池，慢操作异步化[3]，可以说 Swoole 开始将多线程、异步、阻塞引入 PHP 应用开发。

实际上，Swoole 底层内置了异步非阻塞、多线程的网络 IO 服务器，这样仅需处理事件回调[4]即可，无需关心底层。

与 Nginx/Tornado/Node.js 等全异步的框架不同，Swoole 既支持全异步[5]，也支持同步（或者半异步半同步）。

## 1.1 Overview

最初，PHP 在每个请求进来后都需要重新初始化资源，并且在请求执行完毕后全部丢弃，不过在 CLI 模式下可以不释放资源。

[1] Swoole 本质是 PHP 的一种异步并行扩展，因此要应用 Swoole，首先需要有 PHP 环境。

[2] Swoole 内置了 Socket 客户端的实现，但是采用的是同步 + 并行方式来执行。PHP 本身虽然也提供了 socket 的功能，但是某些函数存在 bug，而且比较复杂，因此使用 Swoole 内置的客户端类更加安全和简化。

[3] Swoole 使用了传统 Linux 下半同步半异步多 Worker 的实现方式，业务代码可以按照更简单的同步方式编写，只有慢操作才考虑使用异步。

[4] 如果在使用 Node.js 等进行开发时的代码太复杂，就会产生嵌套多层回调，使代码丧失可读性，程序流程变得很乱。

[5] Node.js 支持全异步回调，而且内置了异步高性能的 Socket Server/Client 实现，在此基础上提供了内置的 Web 服务器。

## 1.2 FastCGI

## 1.3 Multithread

同步和多线程的关系如下：

1. 没有多线程环境就不需要同步。
2. 即使有多线程环境也不一定需要同步。

### 1.3.1 Block

一旦一个线程处于一个标记为 synchronized 的方法中，那么在这个线程从该方法中返回之前，其他所有要调用类中任何标记为 synchronized 方法的线程都会被阻塞。

每个对象都有一个单一的锁，这个锁本身就是对象的一部分。

当在对象上调用其任意 synchronized 方法的时候，此对象都被加锁，这时该对象上的其他 synchronized 方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。

原子操作不需要进行同步（对除 double 和 long 以外的其他基本类型的进行读取或赋值的操作也是原子操作），然而只要给 long 或 double 加上 volatile，操作就是原子操作了。

只能在同步控制方法或同步块中调用 wait()、notify() 和 notifyAll()。如果在非同步的方法里调用这些方法，在运行时会抛出 java.lang.IllegalMonitorStateException 异常。

- 在调用 wait 的时候，线程自动释放其占有的对象锁，同时不会去申请对象锁。
- 当线程被唤醒的时候，它才再次获得了去获得对象锁的权利。其中：
    1. notify 仅唤醒一个线程并允许它去获得锁；
    2. notifyAll 是唤醒所有等待这个对象的线程并允许它们去获得对象锁。

### 1.3.2 Dispatch

OOP 的实现相对来说比较复杂，如果低层实现不好，用户的感受只能是难以使用，因此为了保证效率同时避免不必要的运行判定，有以下两个问题一定解决：

- 调度（Dispatch）：相对于静态判定，当调度和重新调度时，在程序执行时动态决定所要调用的子程序。
- 多继承（Multiple Inheritance）：从两个或更多的父类型中继承成员及操作。

## 1.4 MultiProcess

## 1.5 Synchronization

## 1.6 Asynchronization

# Chapter 2

# Installation

swoole 是标准的 PHP 扩展，而且不依赖 PHP 的 stream、sockets、pcntl、posix、sysvmsg 等扩展，因此只需下载 Swoole 源码包并解压至本地任意目录（保证读写权限），PHP 则只需安装最基本的扩展即可。

按照 PHP 标准扩展构建的构建流程，首先使用 `phpize`[1]来生成 PHP 编译配置，然后执行`./configure` 来做编译配置检测，`make` 进行编译，`make install` 进行安装。

除了手工下载编译外，还可以通过 PHP 官方提供的 pecl 命令来在线安装 swoole：

```
# pecl install swoole
```

## 2.1 Linux

在安装 PHP 前，需要安装编译环境和 PHP 的相关依赖[2]。

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

如果在 CentOS/RHEL 环境下编译安装 PHP，则需要预先执行：

1. 编译环境

[1]phpize 命令需要 autoconf 工具，需要预先安装。

[2]swoole 编译为 libswoole.so 作为 C/C++ 库时需要使用 cmake。

```
$ sudo yum -y install \
  gcc \
  gcc-c++ \
  autoconf \
  libjpeg \
  libjpeg-devel \
  libpng \
  libpng-devel \
  freetype \
  freetype-devel \
  libxml2 \
  libxml2-devel \
  zlib \
  zlib-devel \
  glibc \
  glibc-devel \
  glib2 \
  glib2-devel \
  bzip2 \
  bzip2-devel \
  ncurses \
  ncurses-devel \
  curl \
  curl-devel \
  e2fsprogs \
  e2fsprogs-devel \
  krb5 \
  krb5-devel \
  libidn \
  libidn-devel \
  openssl \
  openssl-devel \
  openldap \
  openldap-devel \
  nss_ldap \
  openldap-clients \
  openldap-servers \
  gd \
  gd2 \
  gd-devel \
  gd2-devel \
  perl-CPAN \
  pcre-devel
```

2. 编译 PHP

```
$ cd /usr/local/php-src
$ ./configure \
  --prefix=/usr/local/php \
  --with-config-file-path=/etc/php \
  --enable-fpm \
  --enable-pcntl \
  --enable-mysqlnd \
  --enable-opcache \
  --enable-sockets \
  --enable-sysvmsg \
  --enable-sysvsem \
  --enable-sysvshm \
  --enable-shmop \
  --enable-zip \
  --enable-ftp \
  --enable-soap \
  --enable-xml \
  --enable-mbstring \
  --disable-rpath \
  --disable-debug \
  --disable-fileinfo \
  --with-mysql=mysqlnd \
  --with-mysqli=mysqlnd \
  --with-pdo-mysql=mysqlnd \
  --with-pcre-regex \
  --with-iconv \
  --with-zlib \
  --with-mcrypt \
  --with-gd \
  --with-openssl \
  --with-mhash \
  --with-xmlrpc \
  --with-curl \
  --with-imap-ssl
$ sudo make
$ sudo make install
$ sudo cp php.ini-development /etc/php/
$ cat >> ~/.bashrc
export PATH=/usr/local/php/bin:$PATH
export PATH=/usr/local/php/sbin:$PATH

$ source ~/.bashrc
$ php --version
PHP 5.6.12 (cli) (built: Aug 31 2015 11:09:49)
Copyright (c) 1997-2015 The PHP Group
```

```
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
```

3. 编译 swoole

```
$ cd /usr/local/src/
$ sudo tar zxvf swoole-1.7.19-stable.tar.gz
$ cd swoole-src-swoole-1.7.19-stable
$ sudo phpize
$ sudo ./configure \
  --enable-openssl \
  --enable-swoole-debug \
  --enable-ringbuffer \
  --with-php-config=/usr/local/bin/php-config \
  --enable-async-mysql=/usr/local/mysql \
  --with-swoole \
  --enable-swoole \
  --with-gnu-ld \
  --with-pic
$ sudo make
$ sudo make install
$ sudo echo "extension=swoole.so" >> /usr/local/lib/php.ini
$ php -m
[PHP Modules]
Core
ctype
date
dom
ereg
filter
hash
iconv
json
libxml
mysql
mysqlnd
pcre
PDO
pdo_sqlite
Phar
posix
propro
raphf
Reflection
session
SimpleXML
SPL
```

```
sqlite3
standard
swoole
tokenizer
xml
xmlreader
xmlwriter

[Zend Modules]
```

4. 查看 swoole 信息

```
# php --ri swoole
swoole
swoole support => enabled
Version => 1.7.19
Author => tianfeng.han[email: mikan.tenny@gmail.com]
epoll => enabled
eventfd => enabled
timerfd => enabled
signalfd => enabled
cpu affinity => enabled
spinlock => enabled
rwlock => enabled
sockets => enabled
openssl => enabled
ringbuffer => enabled
Linux Native AIO => enabled
Gcc AIO => enabled
pcre => enabled
zlib => enabled
mutex_timedlock => enabled
pthread_barrier => enabled

Directive => Local Value => Master Value
swoole.aio_thread_num => 2 => 2
swoole.display_errors => On => On
swoole.message_queue_key => 0 => 0
swoole.unixsock_buffer_size => 8388608 => 8388608
```

通过 `php -m`[3]或 `phpinfo()` 来查看是否成功加载了 swoole，如果没有可能是 php.ini 的路径不对，可以使用 `php -i |grep php.ini` 来定位到 php.ini 的绝对路径。

[3]首先查看加载的 php.ini 路径并确认加载了正确的 php.ini，其次可以使用绝对路径指定 swoole 的位置，最后在 php.ini 中打开错误显示来检查是否存在启动时错误。

### 2.1.1 Error

修改 php.ini，打开错误显示，查看是否存在启动时错误。

```
display_errors => Off => Off
display_startup_errors => Off => Off
html_errors => Off => Off
ignore_repeated_errors => Off => Off
log_errors => On => On
log_errors_max_len => 1024 => 1024
track_errors => Off => Off
xmlrpc_errors => Off => Off
swoole.display_errors => On => On
```

默认情况下，可以在 php.ini 修改错误报告的设置，例如：

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Error handling and logging ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; This directive informs PHP of which errors, warnings and notices you would like
; it to take action for. The recommended way of setting values for this
; directive is through the use of the error level constants and bitwise
; operators. The error level constants are below here for convenience as well as
; some common settings and their meanings.
; By default, PHP is set to take action on all errors, notices and warnings EXCEPT
; those related to E_NOTICE and E_STRICT, which together cover best practices and
; recommended coding standards in PHP. For performance reasons, this is the
; recommend error reporting setting. Your production server shouldn't be wasting
; resources complaining about best practices and coding standards. That's what
; development servers and development settings are for.
; Note: The php.ini-development file has this setting as E_ALL. This
; means it pretty much reports everything which is exactly what you want during
; development and early testing.
;
; Error Level Constants:
; E_ALL           - All errors and warnings (includes E_STRICT as of PHP 5.4.0)
; E_ERROR         - fatal run-time errors
; E_RECOVERABLE_ERROR - almost fatal run-time errors
; E_WARNING       - run-time warnings (non-fatal errors)
; E_PARSE         - compile-time parse errors
; E_NOTICE        - run-time notices (these are warnings which often result
;                   from a bug in your code, but it's possible that it was
;                   intentional (e.g., using an uninitialized variable and
;                   relying on the fact it is automatically initialized to an
;                   empty string)
; E_STRICT        - run-time notices, enable to have PHP suggest changes
```

```
;                  to your code which will ensure the best interoperability
;                  and forward compatibility of your code
; E_CORE_ERROR    - fatal errors that occur during PHP's initial startup
; E_CORE_WARNING  - warnings (non-fatal errors) that occur during PHP's
;                  initial startup
; E_COMPILE_ERROR - fatal compile-time errors
; E_COMPILE_WARNING - compile-time warnings (non-fatal errors)
; E_USER_ERROR    - user-generated error message
; E_USER_WARNING  - user-generated warning message
; E_USER_NOTICE   - user-generated notice message
; E_DEPRECATED    - warn about code that will not work in future versions
;                  of PHP
; E_USER_DEPRECATED - user-generated deprecation warnings
;
; Common Values:
;   E_ALL (Show all errors, warnings and notices including coding standards.)
;   E_ALL & ~E_NOTICE (Show all errors, except for notices)
;   E_ALL & ~E_NOTICE & ~E_STRICT (Show all errors, except for notices and coding
    standards warnings.)
;   E_COMPILE_ERROR|E_RECOVERABLE_ERROR|E_ERROR|E_CORE_ERROR (Show only errors)
; Default Value: E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED
; Development Value: E_ALL
; Production Value: E_ALL & ~E_DEPRECATED & ~E_STRICT
; http://php.net/error-reporting
error_reporting = E_ALL & ~E_DEPRECATED & ~E_STRICT

; This directive controls whether or not and where PHP will output errors,
; notices and warnings too. Error output is very useful during development, but
; it could be very dangerous in production environments. Depending on the code
; which is triggering the error, sensitive information could potentially leak
; out of your application such as database usernames and passwords or worse.
; For production environments, we recommend logging errors rather than
; sending them to STDOUT.
; Possible Values:
;   Off = Do not display any errors
;   stderr = Display errors to STDERR (affects only CGI/CLI binaries!)
;   On or stdout = Display errors to STDOUT
; Default Value: On
; Development Value: On
; Production Value: Off
; http://php.net/display-errors
display_errors = Off

; The display of errors which occur during PHP's startup sequence are handled
; separately from display_errors. PHP's default behavior is to suppress those
```

```
; errors from clients. Turning the display of startup errors on can be useful in
; debugging configuration problems. We strongly recommend you
; set this to 'off' for production servers.
; Default Value: Off
; Development Value: On
; Production Value: Off
; http://php.net/display-startup-errors
display_startup_errors = Off

; Besides displaying errors, PHP can also log errors to locations such as a
; server-specific log, STDERR, or a location specified by the error_log
; directive found below. While errors should not be displayed on productions
; servers they should still be monitored and logging is a great way to do that.
; Default Value: Off
; Development Value: On
; Production Value: On
; http://php.net/log-errors
log_errors = On

; Set maximum length of log_errors. In error_log information about the source is
; added. The default is 1024 and 0 allows to not apply any maximum length at all.
; http://php.net/log-errors-max-len
log_errors_max_len = 1024
; Do not log repeated messages. Repeated errors must occur in same file on same
; line unless ignore_repeated_source is set true.
; http://php.net/ignore-repeated-errors
ignore_repeated_errors = Off

; Ignore source of message when ignoring repeated messages. When this setting
; is On you will not log errors with repeated messages from different files or
; source lines.
; http://php.net/ignore-repeated-source
ignore_repeated_source = Off

; If this parameter is set to Off, then memory leaks will not be shown (on
; stdout or in the log). This has only effect in a debug compile, and if
; error reporting includes E_WARNING in the allowed list
; http://php.net/report-memleaks
report_memleaks = On

; This setting is on by default.
;report_zend_debug = 0

; Store the last error/warning message in $php_errormsg (boolean). Setting this
    value
```

```
; to On can assist in debugging and is appropriate for development servers. It
    should
; however be disabled on production servers.
; Default Value: Off
; Development Value: On
; Production Value: Off
; http://php.net/track-errors
track_errors = Off

; Turn off normal error reporting and emit XML-RPC error XML
; http://php.net/xmlrpc-errors
;xmlrpc_errors = 0

; An XML-RPC faultCode
;xmlrpc_error_number = 0

; When PHP displays or logs an error, it has the capability of formatting the
; error message as HTML for easier reading. This directive controls whether
; the error message is formatted as HTML or not.
; Note: This directive is hardcoded to Off for the CLI SAPI
; Default Value: On
; Development Value: On
; Production value: On
; http://php.net/html-errors
html_errors = On
; If html_errors is set to On *and* docref_root is not empty, then PHP
; produces clickable error messages that direct to a page describing the error
; or function causing the error in detail.
; You can download a copy of the PHP manual from http://php.net/docs
; and change docref_root to the base URL of your local copy including the
; leading '/'. You must also specify the file extension being used including
; the dot. PHP's default behavior is to leave these settings empty, in which
; case no links to documentation are generated.
; Note: Never use this feature for production boxes.
; http://php.net/docref-root
; Examples
;docref_root = "/phpmanual/"

; http://php.net/docref-ext
;docref_ext = .html

; String to output before an error message. PHP's default behavior is to leave
; this setting blank.
; http://php.net/error-prepend-string
; Example:
```

```
;error_prepend_string = "<span style='color: #ff0000'>"

; String to output after an error message. PHP's default behavior is to leave
; this setting blank.
; http://php.net/error-append-string
; Example:
;error_append_string = "</span>"

; Log errors to specified file. PHP's default behavior is to leave this value
; empty.
; http://php.net/error-log
; Example:
;error_log = php_errors.log
; Log errors to syslog (Event Log on Windows).
;error_log = syslog

;windows.show_crt_warning
; Default value: 0
; Development value: 0
; Production value: 0
```

### 2.1.2 Debug

如果编译时使用了`--enable-swoole-debug`，在初始化服务的时候可以 set 是否 debug，这样就可以在开发阶段输出调试数据。

### 2.1.3 Xdebug

PHP 的 xdebug（或 Zend 的 zend_xdebug[4]）扩展都可能会 swoole 崩溃，运行 swoole 程序时务必去掉 xdebug 扩展。

另外，如果 make 或 make install 无法执行或编译错误，原因可能是 php 版本和编译时使用的 phpize 和 php-config 不对应，需要使用绝对路径来进行编译。使用绝对路径执行 PHP。

```
NOTICE: PHP message: PHP Warning: PHP Startup: swoole: Unable to initialize module
Module compiled with module API=20090626
PHP compiled with module API=20121212
These options need to match
in Unknown on line 0
$ sudo /usr/local/php-5.4.17/bin/phpize \
  ./configure \
  --with-php-config=/usr/local/php-5.4.17/bin/php-config \
  /usr/local/php-5.4.17/bin/php server.php
```

[4]如果出现 `error: too many arguments to function 'zend_exception_error'` 错误，则说明 PHP 版本号低于 5.3.10。

### 2.1.4 MySQL

如果出现“缺少 mysql 头文件”错误，说明--enable-async-mysql 需要指定路径，需要在编译前执行：

```
# ./configure
  --with-php-config=/usr/local/bin/php-config \
  --enable-sockets \
  --enable-async-mysql=/usr/local/mysql
```

如果出现“缺少 mysqli 头文件”错误，说明没有找到 mysqlclient 的头文件，需要安装 mysqlclient-dev。

```
php_mysqli_structs.h:64:23: fatal error: my_global.h: No such file or directory
```

建议自行编译 PHP，不要使用 Linux 包管理系统自带的 php 版本，或者可以更全面的配置 MySQL 相关信息，如下面示例中所示：

```
# LDFLAGS="-L/usr/local/mysql/lib" \
  CPPFLAGS="-I/usr/local/mysql/include" \
  ./configure \
  --with-php-config=/usr/local/php/bin/php-config \
  --enable-async-mysql=/usr/local/mysql/include \
  --enable-swoole \
  --with-swoole \
  --enable-openssl \
  --enable-ringbuffer \
  --enable-sockets=/usr/local/php/ext/sockets \
  --enable-swoole-debug
```

### 2.1.5 PCRE

如果出现“缺少 pcre.h 头文件”错误，原因是缺少 pcre，需要安装 libpcre。

```
fatal error: pcre.h: No such file or directory
```

## 2.2 ARM

swoole 也可以运行在 ARM 平台，在编译 Swoole 时手工修改 Makefile 去掉-O2 编译参数。

## 2.3 Windows

从 swoole-1.7.7 开始，swoole 增加了对 cygwin 环境的支持，在 Windows 环境下可以直接使用 cygwin + php 来运行 swoole 程序。

- 安装 cygwin，并安装 gcc、make、autoconf、php[5]。

[5]cygwin 模式下需要对 PHP 进行简化，去掉不使用的扩展，避免进程占用内存过大，导致 Fork 操作失败。

- 下载 swoole 源码，在 cygwin-shell 中进行 `phpize/configure/make/make install`。
- 修改 php.ini，加入 swoole.so。

# Chapter 3

# Configuration

Swoole 和 Yaf 等虽然是标准的基于 Zend API 开发的 PHP C 扩展，实际上与普通的扩展不同。

- 普通的扩展只是提供一个库函数。
- swoole 扩展在运行后会接管 PHP 的控制权，进入事件循环（event-loop）。

当 IO 事件发生后，swoole 会自动回调指定的 PHP 函数。

## 3.1 Configure options

一般来说，./configure 编译配置的额外参数用于开启某些特性。

### 3.1.1 --enable-msgqueue

使用消息队列作为 IPC 通信方式，消息队列的好处是 buffer 区域可以很大。

- 1.7.5+ 已经移除了此编译选项，改为由 swoole_server->set 动态设置开启；
- 1.7.9 版本已移除消息队列模式

另外，dispatch_mode=3 时，消息队列天然支持争抢。

使用消息队列作为 IPC 时，worker 进程内将无法使用异步，包括异步 swoole_client，task/finish，swoole_event_add，swoole_timer_add。

### 3.1.2 --enable-swoole-debug

打开调试日志来允许 swoole 打印各类细节的调试信息，生产环境不要启用。

### 3.1.3 --enable-sockets

增加对 sockets 资源的支持，依赖 PHP 的 sockets 扩展[1]。

在开启`--enable-sockets`参数后，swoole_event_add 就可以添加 sockets 扩展创建的连接到 swoole 的事件循环中。

[1]PHP 的 sockets 扩展基于流行的 BSD sockets，实现了和 socket 进行通信的底层接口，它可以和客户端一样当做一个 socket 服务器。

### 3.1.4 --enable-async-mysql

增加异步 MySQL 支持，依赖 mysqli 和 mysqlnd。

- mysqli（MySQL Improved Extension）扩展允许用户使用 MySQL4.1.3 或更新版本中新的高级特性。
- mysqlnd（MySQL Native Driver）扩展是 MySQL Client Library（libmysqlclient）的替代，并且从 PHP5.3 开始进入官方 PHP 源码库。

其中，PHP 的 MySQL 数据库扩展 mysql、mysqli 和 pdo_mysql 以前都需要通过 MySQL Client Library 提供的服务来与 MySQL 服务器通信。

为了兼容 MySQL 提供的客户端-服务器协议（MySQL Client/Server Protocol），MySQL Client Library 被废弃，后续的 PHP 数据库扩展都被编译为使用 MySQL Native Driver 来代替 MySQL Client Library。

### 3.1.5 --enable-ringbuffer

开启 RingBuffer 内存池（试验性质），主要用于提升性能。

### 3.1.6 --enable-openssl

启用 SSL 支持。

# Chapter 4

# Server

## 4.1 swoole_server

swoole_server 是强大的 TCP/UDP Server 框架，多线程，EventLoop，事件驱动，异步，Worker 进程组，Task 异步任务，毫秒定时器，SSL/TLS 隧道加密。

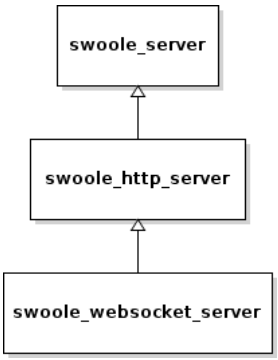


图 4.1: swoole_server 的继承关系

- swoole_http_server 是 swoole_server 的子类，内置了 Http 的支持；
- swoole_websocket_server 是 swoole_http_server 的子类，在 Http 基础上内置了 WebSocket 的支持。

Swoole 的网络 IO 部分基于 epoll/kqueue 事件循环，因此全异步非阻塞执行，而且业务逻辑部分使用多进程同步阻塞方式来运行，从而既保证了 Server 能够应对高并发和大量 TCP 连接，又能够保证业务代码仍然可以简单地编写。

### 4.1.1 swoole_http_server

### 4.1.2 swoole_websocket_server

# Chapter 5

# Client

swoole_client 是 TCP/UDP 客户端，支持同步并发调用，也支持异步事件驱动。

# Chapter 6

# Event

EventLoop API[1]，让用户可以直接操作底层的事件循环，将 socket，stream，管道等 Linux 文件加入到事件循环中。

## 6.1 swoole_event

[1]eventloop 接口仅可用于 socket 类型的文件描述符，不能用于磁盘文件读写。

# Chapter 7

# Async

## 7.1 swoole_async

# Chapter 8

# Process

共享内存的性能虽然很好，但是存在安全问题，需要读写时加锁。

- 锁的粒度过大会导致只有一个线程在运行。
- 锁太复杂又会有死锁问题。

## 8.1 swoole_process

进程管理模块，可以方便的创建子进程，进程间通信，进程管理。

与 Node.js 的网络库本身没有提供多进程/多线程的实现的情况不同，swoole 用户不需要自己手动管理进程的创建与回收，swoole 内核根据配置文件自动完成，Node.j 开发者需要自行创建进程，或者通过 cluster 来利用多核，否则只能使用单线程。

# Chapter 9

# Buffer

## 9.1 swoole_buffer

强大的内存区管理工具，像 C 一样进行指针计算，又无需关心内存的申请和释放，而且不用担心内存越界，底层全部做好了。

# Chapter 10

# Table

## 10.1 swoole_table

swoole_table 是基于共享内存和自旋锁实现的超高性能内存表[1]，可以彻底解决线程，进程间数据共享，加锁同步等问题。

[1]swoole_table 的性能可以达到单线程每秒读写 50W 次。

# Part II

# Server

# Chapter 11

# Overview

首先，软件意义上的服务器就是一个管理资源并为用户提供服务的计算机软件，通常可以分为文件服务器（能使用户在其它计算机访问文件），数据库服务器和应用程序服务器（例如 TCP 服务器、UDP 服务器、HTTP 服务器、WebSocket 服务器以及异步 IO 和 Task 服务器等）。

其次，运行服务器软件的计算机一般称为网络主机（host）[1]，可以通过网络对外提供服务。例如，可以通过 Intranet 对内网提供服务，也可以通过 Internet 对外提供服务。

Web 服务器的定义有时会引起混淆，例如可以指用于网站的计算机，也可能是指 Apache 或 Nginx 等软件，而且运行 Web 服务器软件的计算机可以管理网页组件和回应网页浏览器的请求。

按照服务器软件工作在客户端-服务器还是浏览器-服务器模式，可以有很多形式的服务器，例如：

- 文件服务器（File Server）或网络存储设备（Network Attached Storage），例如 NetWare
- 数据库服务器（Database Server），例如 Oracle，MySQL，PostgreSQL，SQL Server 等
- 邮件服务器（Mail Server），例如 Sendmail、Postfix、Qmail、Microsoft Exchange、Lotus Domino 等
- 网页服务器（Web Server），例如 Apache、httpd 和 IIS 等
- FTP 服务器（FTP Server），例如 Pureftpd、Proftpd、WU-ftpd、Serv-U 等
- 域名服务器（DNS Server），例如 Bind9 等
- 应用程序服务器（Application Server/AP Server），例如 WebLogic、JBoss 和 GlassFish
- 代理服务器（Proxy Server），例如 Squid cache
- NAT 服务器，例如 WINS（Windows Internet Name Service）服务器

[1]服务器与主机的意义可能不同，其中主机是通过终端给用户使用的，服务器是通过网络给客户端用户使用的。

## 11.1 Echo Server

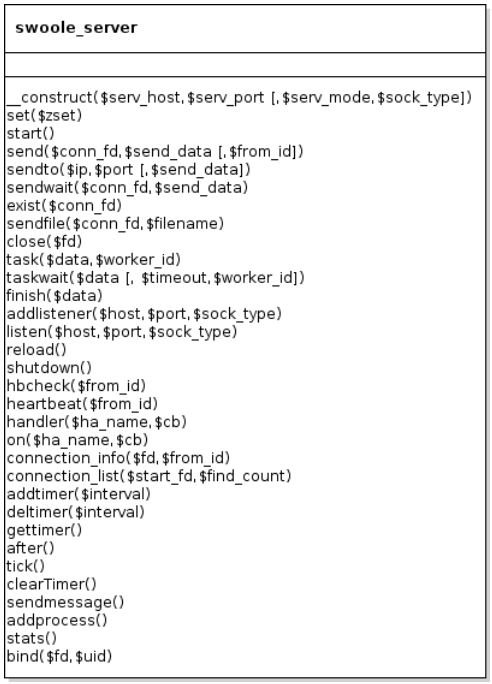


图 11.1: swoole_server 类

下面是一个基本的基于 swoole 的 echo 服务器实现，其中表示监听所有 IP 地址（包括 127.0.0.1、192.168.x.x 以及外网 IP）。

```
<?php
// Server
class Server{
  private $serv;

  public function __construct(){
    $this->serv = new swoole_server("0.0.0.0",9501);
    $this->serv->set(array(
      'worker_num' => 8, // worker进程数
      'daemonize' => 0, // uint32_t, 是否守护进程化
      'max_request' => 10000, // worker进程的最大任务数
      'dispatch_mode' => 2, // 数据包分发策略, 默认为2 (固定模式)
      'debug_mode' => 1 // 无效参数, 可以传入, 不会执行
    ));

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

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

```
    }
    // onStart回调在server运行前被调用
    public function onStart($serv){
      echo "Start\n";
    }
    // onConnect在有新客户端连接过来时被调用
    public function onConnect($serv,$fd,$from_id){
      $serv->send( $fd,"Hello {$fd}!" );
    }
    // onReceive函数在有数据发送到server时被调用
    public function onReceive(swoole_server $serv,$fd,$from_id,$data){
      echo "Get Message From Client {$fd}:{$data}\n";
    }
    // onClose在有客户端断开连接时被调用
    public function onClose($serv, $fd, $from_id){
      echo "Client {$fd} close connection\n";
    }
}

// 启动服务器
$server = new Server();
?>
```

创建一个 swoole_server 基本分为如下三步：

- 通过构造函数创建 swoole_server 实例 server；
- 调用 set 函数设置 swoole_server 实例的相关配置选项；
- 调用 on 函数设置相关回调函数。

这里，on 方法的作用是注册 swoole_server 的事件回调函数。

- 在 onStart 处注册 Start 回调函数来启动 swoole_server 实例 server。
- 在 onConnect 处注册 Connect 回调函数来让 server 监听新的连接。

  如果有数据传入，则 server 可以调用 send 函数将处理结果发送出去。
- 在 onReceive 处注册 Receive 回调函数来接收数据并处理。
- 在 onClose 处注册 Close 回调函数处理客户端下线的事件。

这里，需要注意的是启动 swoole_server 实例之前，swoole_server 必须预先完成下面的操作：

- 已创建了 manager 进程
- 已创建了 worker 子进程
- 已监听所有 TCP/UDP 端口
- 已监听了定时器

在完成上述操作后，swoole_server 的主 Reactor 开始接收事件，客户端可以 connect 到 server。

onStart 回调中仅允许 echo、打印 Log、修改进程名称，不得执行其他操作，而且 onWorkerStart 和 onStart 回调是在不同进程中并行执行的，不存在先后顺序。

在命令行中启动 Echo Server 的示例如下：

```
$ php server.php
```

```
Start
```

## 11.2 Echo Client

下面使用 swoole_client 创建一个基于 TCP 的客户端实例，然后调用 connect 方法并指定同步还是模式来向指定的 IP 和端口发起连接请求。

- 如果设置为 1，则客户端使用同步阻塞模式（默认），recv 和 send 操作都会产生网络阻塞。
- 如果设置为 0，则客户端使用异步传输模式，connect 会立即返回 true，但是实际上连接并未建立。

在使用异步传输（即非阻塞 socket）时，send/recv 执行前必须使用 swoole_client_select 来检测是否完成了连接。

无论使用同步还是异步模式来进行数据传输，在连接成功后才可以通过 recv() 和 send() 方法来接收和发送请求。

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

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

  public function connect(){
    if(!$this->client->connect("127.0.0.1",9501,1)){
      echo "Error: {$fp->errMsg}[{$fp->errCode}]\n";
    }
    $message = $this->client->recv();
    echo "Get Message From Server: {$message}\n";

    fwrite(STDOUT,"请输入消息：  ");
    $msg = trim(fgets(STDIN));
    $this->client->send($msg);
  }
}
$client = new Client();
$client->connect();
?>
```

在使用非阻塞 socket 时，不能在 connect 后使用 send/recv，通过 isConnected() 判断也是 false。只有当连接成功后，系统会自动回调 onConnect，这时才可以使用 send/recv。

在命令行中运行 echo client 的代码如下：

```
$ php client.php
```

```
Get Message From Server: Hello 1!
```
请输入消息:

# Chapter 12

# TCP

## 12.1 Overview

TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议，在简化的计算机网络 OSI 模型中完成第四层传输层所指定的功能，用户数据报协议（UDP）则是同一层内另一个重要的传输协议。

在因特网协议族（Internet protocol suite）中，TCP 层是位于 IP 层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像渠道一样的连接，但是 IP 层不提供这样的流机制，而是提供不可靠的包交换。

应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，然后 TCP 把数据流分区成适当长度的报文段（通常受该计算机连接的网络的数据链路层的最大传输单元（MTU）的限制）。之后 TCP 把结果包传给 IP 层，由它来通过网络将包传送给接收端实体的 TCP 层。

TCP 为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。

接收端实体对已成功收到的包发回一个相应的确认（ACK），如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失将会被进行重传。

TCP 用一个校验和函数来检验数据是否有错误，而且在发送和接收时都要计算校验和。

TCP 连接包括三个状态：连接创建、数据传送和连接终止，操作系统将 TCP 连接抽象为套接字的编程接口给程序使用，因此要经历一系列的状态改变。

TCP 使用了端口号（Port number）的概念来标识发送方和接收方的应用层，可能的和被正式承认的端口号有 65535（$2^{16}-1$）个。

对每个 TCP 连接的一端都有一个相关的 16 位的无符号端口号分配给它们，端口可以被分为三类：众所周知的、注册的和动态/私有的。

- 众所周知的端口号是由因特网赋号管理局（IANA）来分配的，并且通常被用于系统一级或根进程。

  众所周知的应用程序作为服务器程序来运行，并被动地侦听经常使用这些端口的连接。例如 FTP、TELNET、SMTP、HTTP 等。
- 注册的端口号通常被用来作为终端用户连接服务器时短暂地使用的源端口号，但它们也可以用来标识已被第三方注册了的、被命名的服务。

- 动态/私有的端口号在任何特定的 TCP 连接外不具有任何意义。

注意，TCP 并不是对所有的应用都适合，一些新的带有一些内在的脆弱性的运输层协议也被设计出来。比如，实时应用并不需要甚至无法忍受 TCP 的可靠传输机制。

在实时类型的应用（例如视频通话等）中，通常允许一些丢包、出错或拥塞，而不是去校正它们，因此在实时流多媒体（如因特网广播）、实时多媒体播放器和游戏、IP 电话（VoIP）中可以不使用 TCP。

任何不是很需要可靠性或者是想将功能减到最少的应用可以避免使用 TCP，因此在很多情况下，当只需要多路复用应用服务时，用户数据报协议（UDP）可以代替 TCP 为应用提供服务。

### 12.1.1 Establishment

TCP 用三路握手（three-way handshake）过程创建一个连接。在连接创建过程中，很多参数要被初始化，例如序号被初始化以保证按序传输和连接的强壮性。

一对终端同时初始化一个它们之间的连接是可能的。但通常是由一端打开一个套接字（socket）然后监听来自另一方的连接，这就是通常所指的被动打开（passive open）。服务器端被被动打开以后，用户端就能开始创建主动打开（active open）。

1. 客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三路握手的一部分。客户端把这段连接的序号设定为随机数 A。
2. 服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机序号 B。
3. 最后，客户端再发送一个 ACK。当服务端受到这个 ACK 的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号 A+1，而响应则为 B+1。

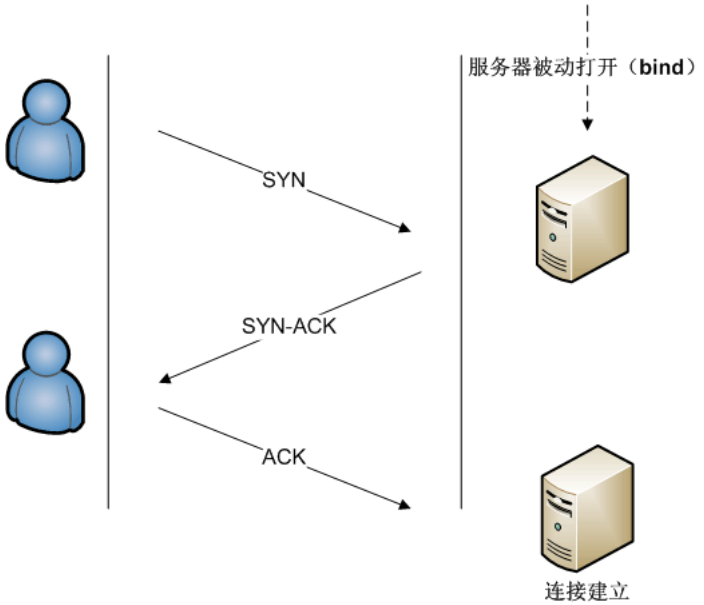


图 12.1: TCP 连接的正常创建

### 12.1.2 Transmission

在 TCP 的数据传送状态，很多重要的机制保证了 TCP 的可靠性和强壮性，其中包括：

- 使用序号，对收到的 TCP 报文段进行排序以及检测重复的数据；
- 使用校验和来检测报文段的错误；
- 使用确认和计时器来检测和纠正丢包或延时。

在 TCP 的连接创建状态，两个主机的 TCP 层间要交换初始序号（ISN，Initial Sequence Number），而且这些序号用于标识字节流中的数据，并且还是对应用层的数据字节进行记数的整数。

通常情况下，在每个 TCP 报文段中都有一对序号和确认号，TCP 报文发送者认为自己的字节编号为序号，而认为接收者的字节编号为确认号。

TCP 报文的接收者为了确保可靠性，在接收到一定数量的连续字节流后才发送确认，其实这是对 TCP 的一种扩展，通常称为选择确认（Selective Acknowledgement），通过选择确认使得 TCP 接收者可以对乱序到达的数据块进行确认，而且每一个字节传输过后，ISN 号都会递增 1。

通过使用序号和确认号，TCP 层可以把收到的报文段中的字节按正确的顺序交付给应用层。

序号是 32 位的无符号数，在它增大到 $2^{32}-1$ 时，便会回绕到 0，因此可以确保 TCP 中关键的一个操作——ISN 的选择的强壮性和安全性。

1. 发送方首先发送第一个包含序列号为 1（可变化）和 1460 字节数据的 TCP 报文段给接收方。接收方以一个没有数据的 TCP 报文段来回复（只含报头），用确认号 1461 来表示已完全收到并请求下一个报文段。
2. 发送方然后发送第二个包含序列号为 1461 和 1460 字节数据的 TCP 报文段给接收方。正常情况下，接收方以一个没有数据的 TCP 报文段来回复，用确认号 2921（1461+1460）来表示已完全收到并请求下一个报文段。发送接收这样继续下去。
3. 然而当这些数据包都是相连的情况下，接收方没有必要每一次都回应。比如，他收到第 1 到 5 条 TCP 报文段，只需回应第五条就行了。在例子中第 3 条 TCP 报文段被丢失了，所以尽管他收到了第 4 和 5 条，然而他只能回应第 2 条。
4. 发送方在发送了第三条以后，没能收到回应，因此当定时器（timer）到期（expire）时，他重发第三条。（每次发送者发送一条 TCP 报文段后，都会再次启动一次时钟：RTT）。
5. 这次第三条被成功接收，接收方可以直接确认第 5 条，因为 4，5 两条已收到。

TCP 的 16 位的校验和（checksum）的计算和检验过程如下：

发送者将 TCP 报文段的头部和数据部分的和计算出来，再对其求反码（一的补数），就得到了校验和，然后将结果装入报文中传输。

这里，用反码和的原因是这种方法的循环进位使校验和可以在 16 位、32 位、64 位等情况下的计算结果再叠加后相同。

接收者在收到报文后再按相同的算法计算一次校验和。这里使用的反码使得接收者不用再将校验和字段保存起来后清零，而可以直接将报文段连同校验加总。如果计算结果是全部为一，那么就表示了报文的完整性和正确性。

TCP 校验和也包括了 96 位的伪头部，其中有源地址、目的地址、协议以及 TCP 的长度，从而可以避免报文被错误地路由。

按照现在的标准，TCP 的校验和是一个比较脆弱的校验。出错概率高的数据链路层需要更高的能力来探测和纠正连接错误。

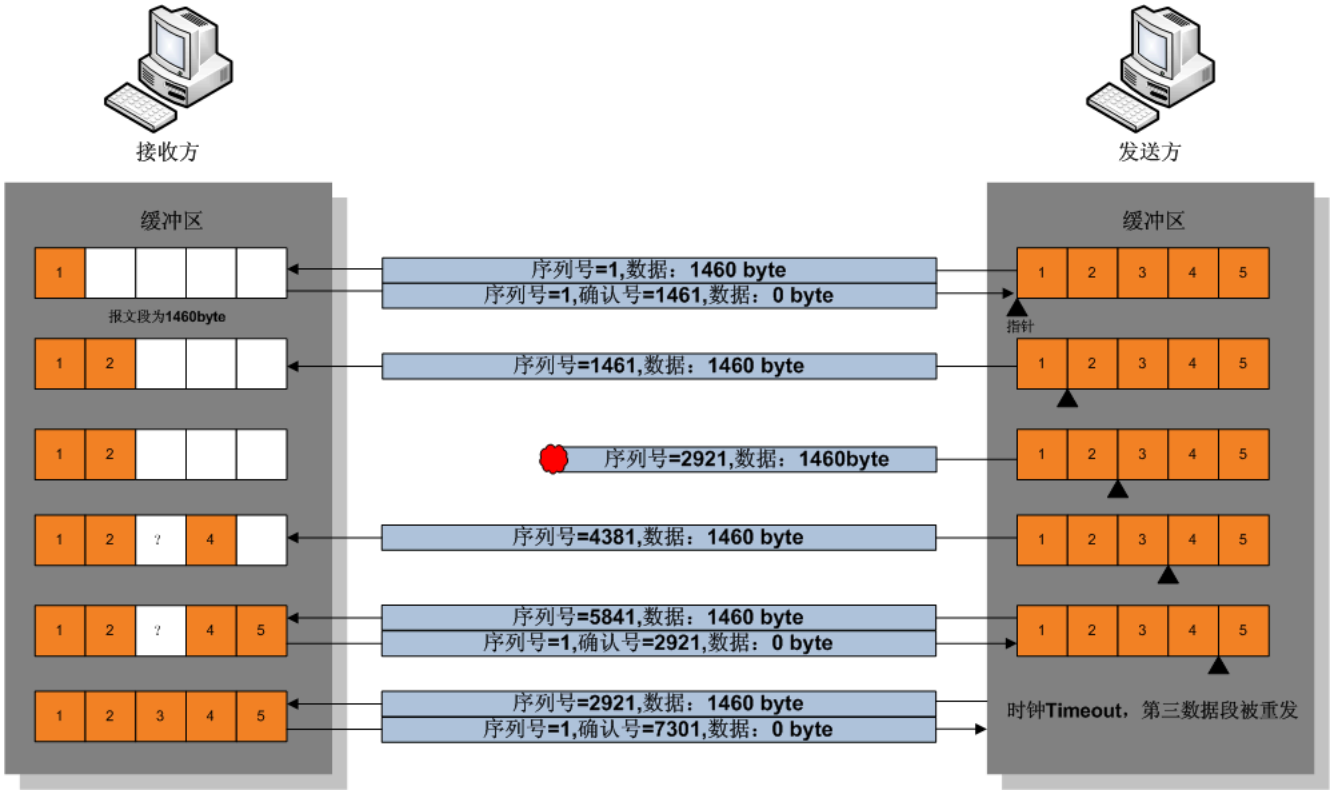


图 12.2: TCP 数据传输

如果在今天重新设计 TCP，很可能使用一个 32 位的 CRC 校验来纠错，而不是使用校验和。但是通过在第二层使用通常的 CRC 校验或更完全一点的校验可以部分地弥补这种脆弱的校验。

第二层是在 TCP 层和 IP 层之下的（比如 PPP 或以太网）使用了这些校验，但是这也并不意味着 TCP 的 16 位校验和是冗余的。

根据对因特网传输的观察，表明在受 CRC 校验保护的各跳之间，软件和硬件的错误通常也会在报文中引入错误，而端到端的 TCP 校验能够捕捉到很多的这种错误，这就是应用中的端到端原则。

流量控制可以避免主机分组发送得过快而使接收方来不及完全收下，因此 TCP 和 UDP 的主要不同在于：

- 有序数据传输
- 重发丢失的数据包
- 舍弃重复的数据包
- 无错误数据传输
- 阻塞/流量控制
- 面向连接（确认有创建三方交握，连接已创建才作传输）

### 12.1.3 Termination

连接终止使用了四路握手过程（four-way handshake），在这个过程中每个终端的连接都能独立地被终止，因此一个典型的拆接过程需要每个终端都提供一对 FIN 和 ACK。

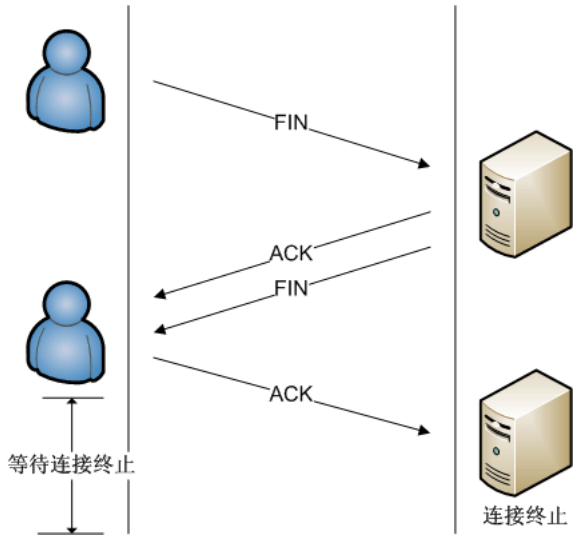


图 12.3: TCP 连接的正常终止

## 12.2 State Diagram

下表为 TCP 状态码列表，以 S 指代服务器，C 指代客户端，S&C 表示两者，S/C 表示两者之一：

表 12.1: TCP 状态码列表

| 状态码 | 状态 | 描述 |
|---|---|---|
| LISTEN S | 侦听状态 | 等待从任意远程 TCP 端口的连接请求。 |
| SYN-SENT C | 在发送连接请求后等待匹配的连接请求。 | 通过 connect() 函数向服务器发出一个同步（SYNC）信号后进入此状态。 |
| SYN-RECEIVED S | | 已经收到并发送同步（SYNC）信号之后等待确认（ACK）请求。 |
| ESTABLISHED S&C | 连接已经打开，收到的数据可以发送给用户。 | 数据传输步骤的正常情况。此时连接两端是平等的。 |
| FIN-WAIT-1 S&C | | 主动关闭端调用 close() 函数发出 FIN 请求包，表示本方的数据发送全部结束，等待 TCP 连接另一端的确认包或 FIN 请求包。 |
| FIN-WAIT-2 S&C | | 主动关闭端在 FIN-WAIT-1 状态下收到确认包，进入等待远程 TCP 的连接终止请求的半关闭状态。这时可以接收数据，但不再发送数据。 |
| CLOSE-WAIT S&C | | 被动关闭端接到 FIN 后，就发出 ACK 以回应 FIN 请求，并进入等待本地用户的连接终止请求的半关闭状态。这时可以发送数据，但不再接收数据。 |
| CLOSING S&C | | 在发出 FIN 后，又收到对方发来的 FIN 后，进入等待对方对连接终止（FIN）的确认（ACK）的状态。 |
| LAST-ACK S&C | | 被动关闭端全部数据发送完成之后，向主动关闭端发送 FIN，进入等待确认包的状态。 |
| TIME-WAIT S/C | | 主动关闭端接收到 FIN 后，就发送 ACK 包，等待足够时间[1]以确保被动关闭端收到了终止请求的确认包。 |
| CLOSED S&C | | 完全没有连接。 |

[1]按照 RFC 793，一个连接可以在 TIME-WAIT 保证最大四分钟，即最大分段寿命（maximum segment lifetime）的 2 倍。

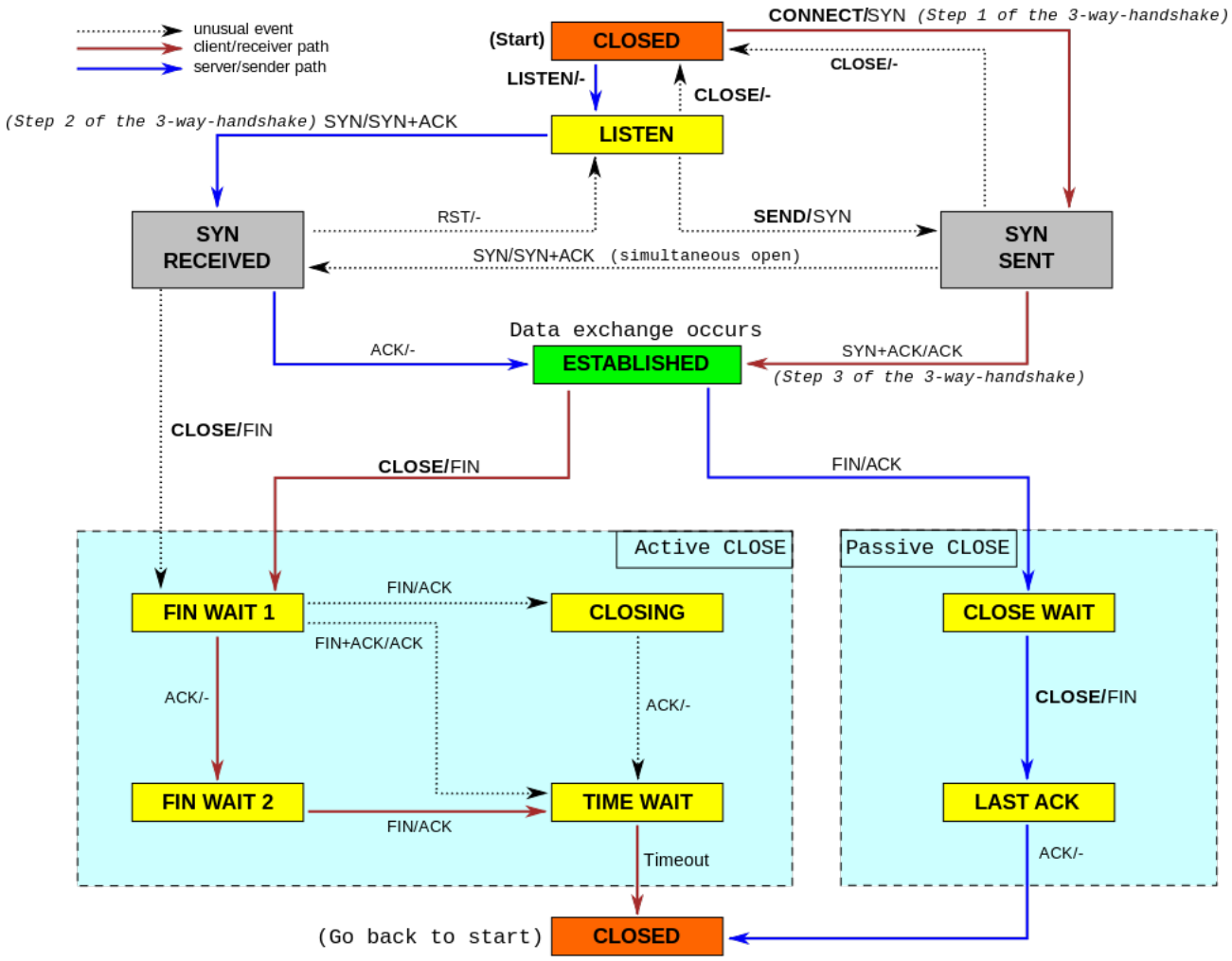


图 12.4: TCP 状态码列表

## 12.3 TCP Server

下面的示例创建一个 TCP 服务器，监听本机 9501 端口，当客户端 Socket 通过网络发送一个 hello 字符串时，服务器会回复一个 Swoole TCP Server: hello 字符串。

```
//创建swoole_server对象，在127.0.0.1监听9501端口
$serv = new swoole_server("127.0.0.1", 9501);

$serv->set(array(
    'worker_num' => 8, //工作进程数量
    'daemonize' => 0 //是否作为守护进程
));

// 监听连接进入事件
$serv->on('connect', function ($serv, $fd){
    echo "Client:Connect.\n";
});

//监听数据发送事件
$serv->on('receive', function ($serv, $fd, $from_id, $data) {
    $serv->send($fd, 'Swoole TCP Server: '.$data);
    $serv->close($fd);
});

//监听连接关闭事件
$serv->on('close', function ($serv, $fd) {
    echo "Client: Close.\n";
});

//启动服务器
$serv->start();
```

swoole_server 是异步服务器，所以是通过监听事件的方式来编写程序的。

当对应的事件发生时底层会主动回调指定的 PHP 函数，例如当有新的 TCP 连接进入时会执行 onConnect 事件回调，当某个连接向服务器发送数据时会回调 onReceive 函数。

- 服务器可以同时被成千上万个客户端连接，`$fd` 就是客户端连接的唯一标识符
- 调用 `$server->send()` 方法向客户端连接发送数据，参数就是 `$fd` 客户端标识符
- 调用 `$server->close()` 方法可以强制关闭某个客户端连接
- 客户端可能会主动断开连接，此时会触发 onClose 事件回调

在命令行下运行 server.php 程序来启动 Swoole TCP Server，如果启动成功后可以使用 netstat 工具看到，已经在监听 9501 端口，然后就可以使用 telnet/netcat 工具连接服务器。

```
$ php tcp_server.php
```

- 在本机使用 telnet/netcat 连接 Swoole TCP Server：

```
$ telnet 127.0.0.1 9501
Trying 127.0.0.1...
```

```
Connected to 127.0.0.1.
Escape character is '^]'.
hello
Swoole TCP Server: hello
```

- 在远程主机使用 telnet/netcat 连接 Swoole TCP Server：

```
$ telnet 121.40.126.146 9501
Trying 121.40.126.146...
telnet: connect to address 121.40.126.146: Connection refused
```

## 12.4 TCP Client

除了使用 telnet/netcat 来连接 Swoole TCP Server 之外，也可以自己实现 TCP Client 来进行连接和通信。

```
$client = new swoole_client(SWOOLE_SOCK_TCP, SWOOLE_SOCK_ASYNC);
//设置事件回调函数
$client->on("connect", function($cli) {
    $cli->send("hello world\n");
});
$client->on("receive", function($cli, $data){
    echo "Received: ".$data."\n";
});
$client->on("error", function($cli){
    echo "Connect failed\n";
});
$client->on("close", function($cli){
    echo "Connection close\n";
});
//发起网络连接
//$client->connect('127.0.0.1', 9501, 0.5);
$client->connect('121.40.126.146', 9501, 0.5);
```

- SWOOLE_SOCK_TCP 创建 tcp socket
- SWOOLE_SOCK_TCP6 创建 tcp ipv6 socket
- SWOOLE_SOCK_UDP 创建 udp socket
- SWOOLE_SOCK_UDP6 创建 udp ipv6 socket
- SWOOLE_SOCK_SYNC 同步客户端
- SWOOLE_SOCK_ASYNC 异步客户端

在远程服务器上开启 Server TCP Server：

```
$ php tcp_server.php
```

在本地计算机上执行 Swoole TCP Client：

```
$ php tcp_client.php
Received: Swoole TCP Server: hello world

Connection close
```

实际上，`hello world`从远程服务器上发回，因此如果远程服务器关闭，那么就可能返回如下的错误：

```
$ php tcp_client.php
Unknown: connect to server [121.40.126.146:9501] failed.
Error: Connection refused [111]. in Unknown on line 0

Warning: Unknown: connect to server [121.40.126.146:9501] failed.
Error: Connection refused [111]. in Unknown on line 0

Connect failed
```

正常情况下，远程服务器上的 Swoole TCP Server 会产生如下的输出：

```
$ php tcp_server.php
Client:Connect.
Client: Close.
```

# Chapter 13

# UDP

## 13.1 Overview

UDP（User Datagram Protocol，用户数据报协议）是一个简单的面向数据报的传输层协议，其正式规范为 RFC 768。

在 TCP/IP 模型中，UDP 为网络层以上和应用层以下提供了一个简单的接口。

UDP 只提供数据的不可靠传递，它一旦把应用程序发给网络层的数据发送出去，就不保留数据备份，所以 UDP 有时候也被认为是不可靠的数据报协议，而且 UDP 在 IP 数据报的头部仅仅加入了复用和数据校验（字段）。

UDP 首部字段由 4 个部分组成，其中两个是可选的，其中分别都是 16bit 的来源端口和目的端口用来标记发送和接受的应用进程。

UDP 不需要应答，所以来源端口是可选的，如果来源端口不用，那么置为零。在目的端口后面是长度固定的以字节为单位的长度域，用来指定 UDP 数据报包括数据部分的长度，长度最小值为 8byte。

首部剩下的 16bit 是用来对首部和数据部分一起做校验和（Checksum）的，这部分是可选的，但是在实际应用中一般都使用这一功能。

由于缺乏可靠性且属于非连接导向协定，UDP 应用一般必须允许一定量的丢包、出错和复制贴上，但是有些应用（比如 TFTP），如果需要则必须在应用层增加根本的可靠机制。

绝大多数 UDP 应用都不需要可靠机制，甚至可能因为引入可靠机制而降低性能，因此流媒体（串流技术）、即时多媒体游戏和 IP 电话（VoIP）一定是典型的 UDP 应用。如果某个应用需要很高的可靠性，那么可以用 TCP 来代替 UDP。

由于缺乏拥塞控制（congestion control），需要基于网络的机制来减少因失控和高速 UDP 流量负荷而导致的拥塞崩溃效应。换句话说，因为 UDP 发送者不能够检测拥塞，所以像使用包队列和丢弃技术的路由器这样的网络基本设备往往就成为降低 UDP 过大通信量的有效工具，后来的 DCCP（数据报拥塞控制协议）被设计成通过在诸如流媒体类型的高速率 UDP 流中，增加主机拥塞控制来减小这个潜在的问题。

基于 UDP 协议的关键应用在一定程度上是相似的，这些应用包括域名系统（DNS）、简单网络管理协议（SNMP）、动态主机配置协议（DHCP）、路由信息协议（RIP）和某些影音串流服务等等。

## 13.2 UDP Server

Swoole 支持 CPU Affinity 设置，守护进程化，并且混合 UDP/TCP 多端口监听，多定时器等。

```
//创建Server对象，监听 127.0.0.1:9502端口，类型为SWOOLE_SOCK_UDP
$serv = new swoole_server("127.0.0.1", 9502, SWOOLE_PROCESS, SWOOLE_SOCK_UDP);
$server->set(['worker_num' => 1]);

//监听数据发送事件
$server->on('Packet', function (swoole_server $serv, $data, $addr)
{
    $serv->sendto($addr['address'], $addr['port'], "Swoole UDP Server: $data" );
    var_dump( $addr, strlen($data));
});

//启动服务器
$serv->start();
```

UDP 服务器与 TCP 服务器不同，UDP 没有连接的概念，因此启动 Server 后，客户端无需 Connect，直接可以向 Server 监听的 9502 端口发送数据包，服务器端对应的事件为 onPacket。

- `$clientInfo` 是客户端的相关信息，是一个数组，有客户端的 IP 和端口等内容
- 调用 `$server->send` 方法向客户端发送数据

在启动 Swoole UDP Server 后就可以使用 netcat 来尝试连接，或者自己实现 UDP Client 来连接。

```
$ php udp_server.php
$ netcat -u 127.0.0.1 9502
```

## 13.3 UDP Client

```
 <?php
$client = new swoole_client(SWOOLE_SOCK_UDP,SWOOLE_SOCK_SYNC);
$client->connect('127.0.0.1',9502);
$client->send("UDP Connection from bz");
echo $client->recv();
?>
```

- 如果从本地执行 udp_server.php，那么首先会看到 UDP 服务器在等待接收客户端数据传入。
- 如果从本地执行 udp_client.php，那么 UDP 客户端首先向 UDP 服务器发送数据，然后接收 UDP 服务器响应。
- 在接收到 UDP 客户端发送的数据并回送响应后，UDP 服务器会输出关于 UDP 客户端连接的数据。

```
$ php udp_server.php

$ php udp_client.php
```

```
Swoole UDP Server: UDP Connection from bz
$ php udp_server.php
array(3) {
  ["server_socket"]=>
  int(3)
  ["address"]=>
  string(9) "127.0.0.1"
  ["port"]=>
  int(42617)
}
int(22)
```

# Chapter 14

# HTTP

## 14.1 Overview

最初，设计 HTTP（HyperText Transfer Protocol，超文本转移协议）的目的是为了提供一种发布和接收 HTML 页面的方法，并且统一使用 URI（Uniform Resource Identifiers）来标识 HTTP 或者 HTTPS 协议请求的资源。

现在，HTTP 已经演化为一个客户端终端（用户）和服务器端（网站）请求和应答的标准（TCP）。

- 通过使用 Web 浏览器、网络爬虫或者其它的工具，客户端发起一个 HTTP 请求到服务器上指定端口（默认端口为 80），我们称这个客户端为用户代理程序（user agent）。
- 应答的 HTTP 服务器上存储着一些资源，比如 HTML 文件和图像，我们称这个应答服务器为源服务器（origin server）。
- 在用户代理和源服务器中间可能存在多个“中间层”，比如代理服务器、网关或者隧道（tunnel）。

尽管 TCP/IP 协议是互联网上最流行的应用，但是 HTTP 协议中并没有规定必须使用它或它支持的层，事实上 HTTP 可以在任何互联网协议上，或其他网络上实现。

HTTP 假定其下层协议提供可靠的传输，因此任何能够提供这种保证的协议都可以被其使用，因此构建在 TCP/IP 协议族之上的 HTTP 协议使用 TCP 作为其传输层。

通常情况下，由 HTTP 客户端发起一个请求，并创建一个到服务器指定端口（默认是 80 端口）的 TCP 连接。

HTTP 服务器会以守护进程形式运行，并且在对应的端口监听客户端的请求，这样一旦收到请求，服务器就会向客户端返回一个状态（比如`"HTTP/1.1 200 OK"`）以及返回的内容，例如请求的文件、错误消息或者其它信息。

超文本传输协议已经演化出了很多版本，它们中的大部分都是向下兼容的。

- 客户端在请求的开始告诉服务器它采用的协议版本号。
- 服务器在响应中采用相同或者更早的协议版本。

在 HTTP 0.9 和 1.0[1]使用非持续连接，非持续连接下的每个 tcp 只连接一个 Web 对象，连接在每个请求-回应对后都会关闭，一个连接可被多个请求重复利用的保持连接机制被引入。

这种连接持续化显著地减少了请求延迟，因为客户不用在首次请求后再次进行 TCP 交互确认创建连接，现在在 HTTP 1.1 已经默认使用持续连接，不必为每个 web 对象创建一个新的连接，一个连接可以传送多个对象。

HTTP/1.1 相较于 HTTP/1.0 协议的区别主要体现在：

[1]HTTP 1.0 是第一个在通讯中指定版本号的 HTTP 协议版本，至今仍被广泛采用（特别是在代理服务器中）。

- 缓存处理
- 带宽优化及网络连接的使用
- 错误通知的管理
- 消息在网络中的发送
- 互联网地址的维护
- 安全性及完整性

HTTP1.1 还进行了带宽优化，例如 1.1 引入了分块传输编码来允许流化传输持续连接上发送的内容，取代原先的 buffer 式传输。

HTTP 1.1 能很好地配合代理服务器工作，而且还支持以渠道方式在同时发送多个请求，以便降低线路负载，提高传输速度，HTTP 渠道允许客户在上一个回应被收到前发送多重请求从而进一步减少了延迟时间。

另一项协议的改进是 byte serving（字节服务），允许服务器根据客户的请求仅仅传输资源的一部分。

- 在 HTTP1.0，单一 TCP 连接内仅执行一个“客户端发送请求—服务器发送应答”周期，之后释放 TCP 连接。
- 在 HTTP1.1 优化支持持续活跃连接：客户端连续多次发送请求、接收应答；批量多请求时，同一 TCP 连接在活跃（Keep-Live）间期内复用，避免重复 TCP 初始握手活动，减少网络负荷和响应周期。

此外，支持应答到达前继续发送请求（通常是两个），称为“流线化”（stream）。

### 14.1.1 Request Message

HTTP 客户端发出的请求信息包括请求行、（请求）头、空行和其他消息体。

- 请求行，例如 `GET /images/logo.gif HTTP/1.1` 表示从/images 目录下请求 logo.gif 这个文件。
- （请求）头，例如 `Accept-Language: en`
- 空行
- 其他消息体

请求行和标题必须以 <CR><LF> 作为结尾，而且空行内必须只有 <CR><LF> 而无其他空格。

在 HTTP/1.1 协议中，所有的请求头，除 Host 外都是可选的。

### 14.1.2 Request Method

HTTP/1.1 协议中共定义了 8 种方法（也叫“动作”）来以不同方式操作指定的资源。

表 14.1: HTTP 1.1 请求方法

| 方法 | 说明 | 应用 |
| --- | --- | --- |
| OPTIONS | OPTION 方法可使服务器传回该资源所支持的所有 HTTP 请求方法。 | 用'*' 来代替资源名称向 Web 服务器发送 OPTIONS 请求，可以测试服务器功能是否正常运作。 |
| HEAD | 与 GET 方法一样，都是向服务器发出指定资源的请求。只不过服务器将不传回资源的本文部分。 | HEAD 方法的好处在于，使用这个方法可以在不必传输全部内容的情况下，就可以获取其中“关于该资源的信息”（元信息或称元数据）。 |
| GET | 向指定的资源发出“显示”请求。 | 使用 GET 方法应该只用在读取数据，而不应当被用于产生“副作用”的操作中（例如 Web Application），还有一个原因是 GET 可能会被网络蜘蛛等随意访问。 |

| 方法 | 说明 | 应用 |
| --- | --- | --- |
| POST | 向指定资源提交数据，请求服务器进行处理（例如提交表单或者上传文件）。 | 数据被包含在请求本文中。这个请求可能会创建新的资源或修改现有资源，或二者皆有。 |
| PUT | 向指定资源位置上传其最新内容。 | |
| DELETE | 请求服务器删除 Request-URI 所标识的资源。 | |
| TRACE | 回显服务器收到的请求，主要用于测试或诊断。 | |
| CONNECT | HTTP/1.1 协议中预留给能够将连接改为渠道方式的代理服务器。 | 通常用于 SSL 加密服务器的链接（经由非加密的 HTTP 代理服务器）。 |
| PATCH（可选） | 用于将局部修改应用到资源。 | 由 RFC 5789 指定的方法 |

方法名称是区分大小写的，而且 HTTP 服务器至少应该实现 GET 和 HEAD 方法，其他方法都是可选的。

下面是一个 HTTP 客户端与服务器之间会话的例子，运行于 www.google.com，端口 80。

- 客户端请求

```
GET / HTTP/1.1
Host: www.google.com
```

客户端请求的末尾有一个空行。第一行指定方法、资源路径、协议版本；第二行是在 1.1 版里必带的一个 header 作用指定主机

- 服务器应答

```
HTTP/1.1 200 OK
Content-Length: 3059
Server: GWS/2.0
Date: Sat, 11 Jan 2003 02:44:04 GMT
Content-Type: text/html
Cache-control: private
Set-Cookie:
    PREF=ID=73d4aef52e57bae9:TM=1042253044:LM=1042253044:S=SMCc_HRPCQiqy
X9j; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com
Connection: keep-alive
```

服务器应答头的末尾紧跟着一个空行，并且由 HTML 格式的文本组成了 Google 的主页。

当某个请求所针对的资源不支持对应的请求方法的时候，服务器应当返回状态码 405（Method Not Allowed），当服务器不认识或者不支持对应的请求方法的时候，应当返回状态码 501（Not Implemented）。

当然，所有的方法支持的实现都应当符合各自的语义定义。此外，除了上述方法，特定的 HTTP 服务器还能够扩展自定义的方法（例如 PATCH）。

### 14.1.3 Safe Method

对于 GET 和 HEAD 方法而言，除了进行获取资源信息外，这些请求不应当再有其他意义。也就是说，这些方法应当被认为是“安全的”。

客户端可能会使用其他“非安全”方法（例如 POST，PUT 及 DELETE），应该以特殊的方式（通常是按钮而不是超链接）告知客户可能的后果（例如一个按钮控制的资金交易），或请求的操作可能是不安全的（例如某个文件将被上传或删除）。

不过，不能想当然地认为服务器在处理某个 GET 请求时不会产生任何副作用。

事实上，很多动态资源会把这作为其特性，这里重要的区别在于用户并没有请求这一副作用，因此不应由用户为这些副作用承担责任。

### 14.1.4 Side Reaction

假如在不考虑诸如错误或者过期等问题的情况下，若干次请求的副作用与单次请求相同或者根本没有副作用，那么这些请求方法就能够被视作“幂等”的。GET，HEAD，PUT 和 DELETE 方法都有这样的幂等属性，同样由于根据协议，OPTIONS，TRACE 都不应有副作用，因此也理所当然也是幂等的。

假如某个由若干个请求做成的请求序列产生的结果在重复执行这个请求序列或者其中任何一个或多个请求后仍没有发生变化，则这个请求序列便是“幂等”的。但是，可能出现若干个请求做成的请求序列是“非幂等”的，即使这个请求序列中所有执行的请求方法都是幂等的。例如，这个请求序列的结果依赖于某个会在下次执行这个序列的过程中被修改的变量。

### 14.1.5 Status Code

所有 HTTP 响应的第一行都是状态行，依次是当前 HTTP 版本号，3 位数字组成的状态代码，以及描述状态的短语，彼此由空格分隔。

状态代码的第一个数字代表当前响应的类型：

- 1xx 消息——请求已被服务器接收，继续处理
- 2xx 成功——请求已成功被服务器接收、理解、并接受
- 3xx 重定向——需要后续操作才能完成这一请求
- 4xx 请求错误——请求含有词法错误或者无法被执行
- 5xx 服务器错误——服务器在处理某个正确请求时发生错误

虽然 RFC 2616 中已经推荐了描述状态的短语，例如"200 OK" 和"404 Not Found" 等，但是 Web 开发者仍然能够自行决定采用何种短语，用以显示本地化的状态描述或者自定义信息。

### 14.1.6 HTTPS

目前有两种方法来创建安全超文本协议连接，分别是 HTTPS URI 方案和 HTTP 1.1 请求头（由 RFC 2817 引入）。

由于浏览器对后者的几乎没有任何支持，因此 HTTPS URI 方案仍是创建安全超文本协议连接的主要手段，安全超文本连接协议使用 https://代替 http://。

## 14.2 HTTP Server

```
$serv = new swoole_http_server("127.0.0.1", 9502);

$serv->on('Request', function($request, $response) {
    var_dump($request->get);
    var_dump($request->post);
    var_dump($request->cookie);
    var_dump($request->files);
```

```
    var_dump($request->header);
    var_dump($request->server);

    $response->cookie("User", "Swoole");
    $response->header("X-Server", "Swoole");
    $response->end("<h1>Hello Swoole!</h1>");
});

$serv->start();
```

## 14.3 HTTPS Server

# Chapter 15

# WebSocket Server

```php
<?php
/**
 * 创建WebSocket Server
 */
$serv = new swoole_websocket_server("127.0.0.1", 9502);

/**
 * 注册Server的事件回调函数open
 */
$serv->on('Open', function($server, $req) {
    echo "connection open: ".$req->fd;
});
/**
 * 注册Server的事件回调函数message
 */
$serv->on('Message', function($server, $frame) {
    echo "message: ".$frame->data;
    $server->push($frame->fd, json_encode(["hello", "world"]));
});
/**
 * 注册Server的事件回调函数close
 */
$serv->on('Close', function($server, $fd) {
    echo "connection close: ".$fd;
});

/**
 * 启动WebSocket Server
 */
$serv->start();
?>
```

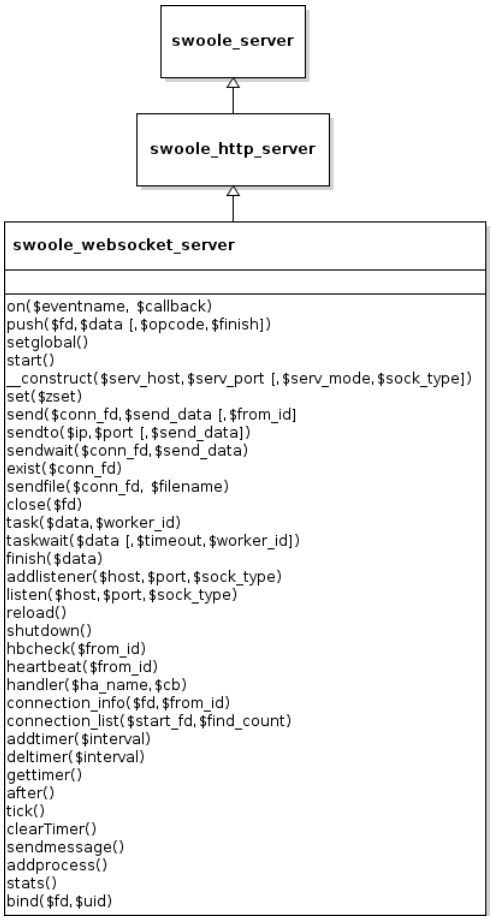


图 15.1: swoole_websocket_server 类

- `$fd`：TCP 连接的文件描述符（file description），在 swoole_server 中是客户端的唯一标识符。
- `$from_id`：来自于哪个 reactor 线程。
- `$conn_fd`：网络字节序（long 类型字段，IPv4 的第 4 字节最小为 1）。

## 15.1 WS Server

```php
<?php
/**
 * 创建WebSocket Server
 */
$wsserver = new swoole_websocket_server("127.0.0.1", 9502);

/**
 * 注册Server的事件回调函数open
 */
$wsserver->on('Open', function($server, $req) {
    echo "connection open: ".$req->fd;
});

/**
 * 注册Server的事件回调函数message
 */
$wsserver->on('Message', function($server, $frame) {
    echo "message: ".$frame->data;
    $server->push($frame->fd, json_encode(["hello", "world"]));
});

/**
 * 注册Server的事件回调函数close
 */
$wsserver->on('Close', function($server, $fd) {
    echo "connection close: ".$fd;
});

/**
 * 启动WebSocket Server
 */
$wsserver->start();
?>
```

## 15.2 WSS Server

```
<?php
$wssserver = new swoole_websocket_server("0.0.0.0",9527,SWOOLE_PROCESS,);
?>
```

Chapter 16

# WebSocket Client

# Chapter 17

# Async-IO

```
$fp = stream_socket_client("tcp://127.0.0.1:80", $code, $msg, 3);
$http_request = "GET /index.html HTTP/1.1\r\n\r\n";
fwrite($fp, $http_request);
swoole_event_add($fp, function($fp){
    echo fread($fp, 8192);
    swoole_event_del($fp);
    fclose($fp);
});
swoole_timer_after(2000, function() {
    echo "2000ms timeout\n";
});
swoole_timer_tick(1000, function() {
    echo "1000ms interval\n";
});
```

# Chapter 18

# Task

```
$serv = new swoole_server("127.0.0.1", 9502);
$serv->set(array('task_worker_num' => 4));
$serv->on('Receive', function($serv, $fd, $from_id, $data) {
    $task_id = $serv->task("Async");
    echo "Dispath AsyncTask: id=$task_id\n";
});
$serv->on('Task', function ($serv, $task_id, $from_id, $data) {
    echo "New AsyncTask[id=$task_id]".PHP_EOL;
    $serv->finish("$data -> OK");
});
$serv->on('Finish', function ($serv, $task_id, $data) {
    echo "AsyncTask[$task_id] Finish: $data".PHP_EOL;
});
$serv->start();
```