IM 系统开发文档

Briquz zgy@bz-inc.com

2017 年 3 月 2 日

version v1

目录

插图

表格

例目录

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）来进行编译安装，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。

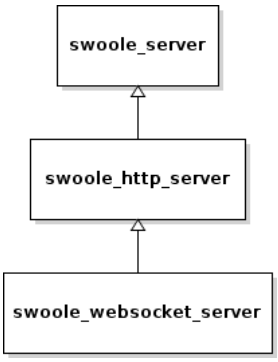

图 4.1: swoole_server 的继承关系

- 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

4.2 swiftmailer

PHP 直接使用 SMTP 协议来同步发送邮件的速度太慢，所以需要基于 Swoole 等来实现邮件的异步发送。

大概来说，Swoole 服务器端程序以 cli 模式并运行在守护进程模式，然后再通过一个客户端去连接服务器端并通知发送邮件，服务器端在收到信息后过回调函数，执行相应的程序来实现邮件的异步发送。

4.2.1 MailServer

使用 composer 安装 SwiftMailer。

```
$ composer require swiftmailer/swiftmailer
```

使用 Swoole 服务器程序来响应客户端的请求并通过 swiftmailer 发送邮件。

```
<?php
/**
 * @author
 * @created 2015/10/18 18:20
 */
require __DIR__ . '/vendor/autoload.php';

class MailServer
{
  const MAIL_USERNAME = 'no-reply@bz-inc.com';
  const MAIL_PASSWORD = 'p@$$w';

  private $logger = null;
  private $server = null;

  public function __construct()
  {
    $this->server = new swoole_server('0.0.0.0',9501);
    $this->server->set(
      [
        'worker_num'=>8,
        'daemonize'=>false,
        'max_request'=>100000,
        'dispatch_mode'=>2,
        'debug_mode'=>1
      ]
    );

    $this->server->on('Start',[$this,'onStart']);
    $this->server->on('Connect',[$this,'onConnnect']);
    $this->server->on('Receive',[$this,'onReceive']);
    $this->server->on('Close',[$this,'onClose');
```

```
    $this->server->start();
  }

  public function onStart()
  {
    //
  }

  public function onConnect($server,$descriptors,$fromId)
  {
    //
  }

  public function onReceive(swoole_server $server,$descriptors,$fromId,$data)
  {
    $msg=json_decode($data,true);
    $sent=$this->sendMail($msg['address'],$msg['subject'],$msg['body']);
    printf("%s mail is sent.\n",$sent);
  }

  public function onClose($server, $descriptors,$fromId)
  {
    //
  }

  public function sendMail($address,$subject,$body)
  {
    $body=htmlspecialchars_decode($body);
    $transport=Swift_SmtpTransport::newInstance('smtp.partner.outlook.cn',587,'tls');
    $transport->setUsername(self::MAIL_USERNAME);
    $transport->setPassword(self::MAIL_PASSWORD);
    $mailer=Swift_Mailer::newInstance($transport);

    $message=Swift_Message::newInstance();
    $message->setFrom([self::MAIL_USERNAME=>'班砖网络']);
    $message->setTo($address);
    $message->setSubject($subject);
    $message->addPart($body,'text/html');
    $message->setBody($body);

    return $mailer->send($message);
  }
}
$server=new MailServer();
```

在配置好 PHP 以及 Swoole 后，使用 cli 启动服务端：

```
$ php mail_server.php
```

如果需要让服务端在后台执行，可以修改配置数组`'daemonize' => false`为`'daemonize'`

=> true。

4.2.2 MailClient

使用 client 连接 server 并发送数据。

```
<?php
/**
 * @author
 * @created 2015/10/18 18:40
 */
class MailClient
{
  private $client;
  public function __construct(){
    $this->client=new swoole_client(SWOOLE_SOCK_TCP);
  }

  public function connect(){
    if(!$this->client->connect('0.0.0.0',9501,1)){
      throw new CException(sprintf('Swoole Error: %s',$this->client->errCode));
    }
  }

  public function send($data)
  {
    if($this->client->isConnect()){
      if(!is_string($data)){
        $data=json_encode($data);
      }
      return $this->client->send($data);
    }else{
      throw new CException('Swoole Server does not connected.');
    }
  }

  public function close(){
    $this->client->close();
  }
}
```

4.2.3 MailInstance

```
$data=array(
  'address'=>$mails,
```

```
  'subject'=>$subject,
  'body'=>htmlspecialchars($body)
);

$mailClient=new MailClient();
$mailClient->connect();

if($mailClient->send($data)){
  echo 'success';
}else{
  echo 'fail';
}

$mailClient->close();
```

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

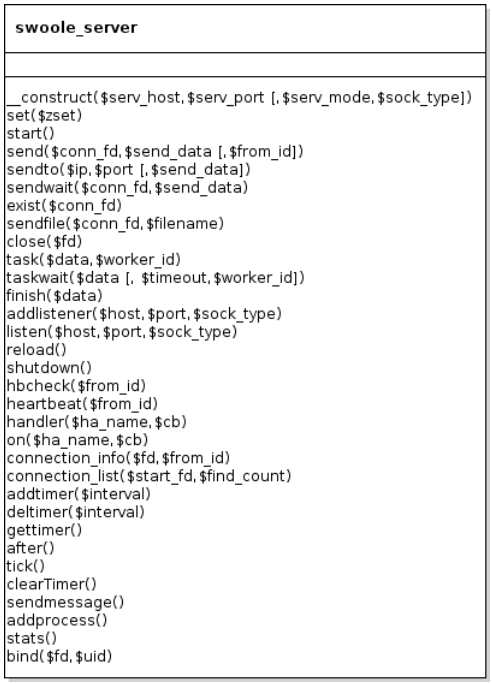

图 11.1: swoole_server 类

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
$echoserver = 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);
  }
}
$echoclient = new Client();
$echoclient->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。

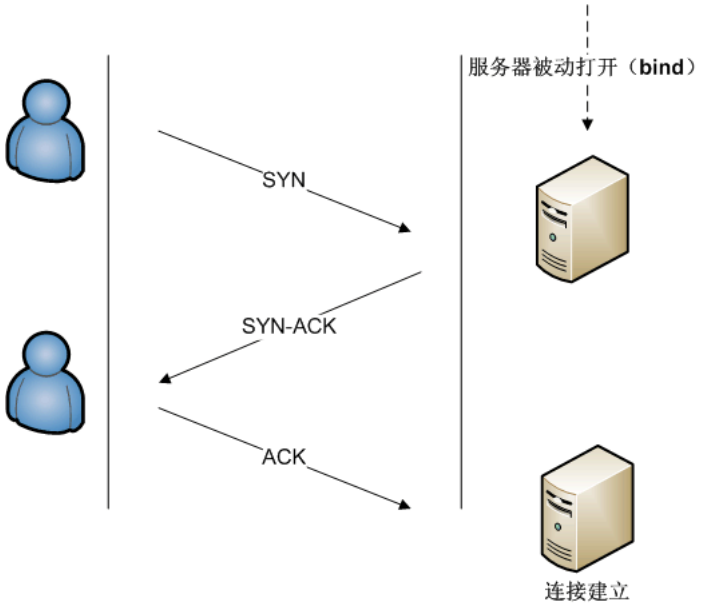

图 12.1: TCP 连接的正常创建

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。

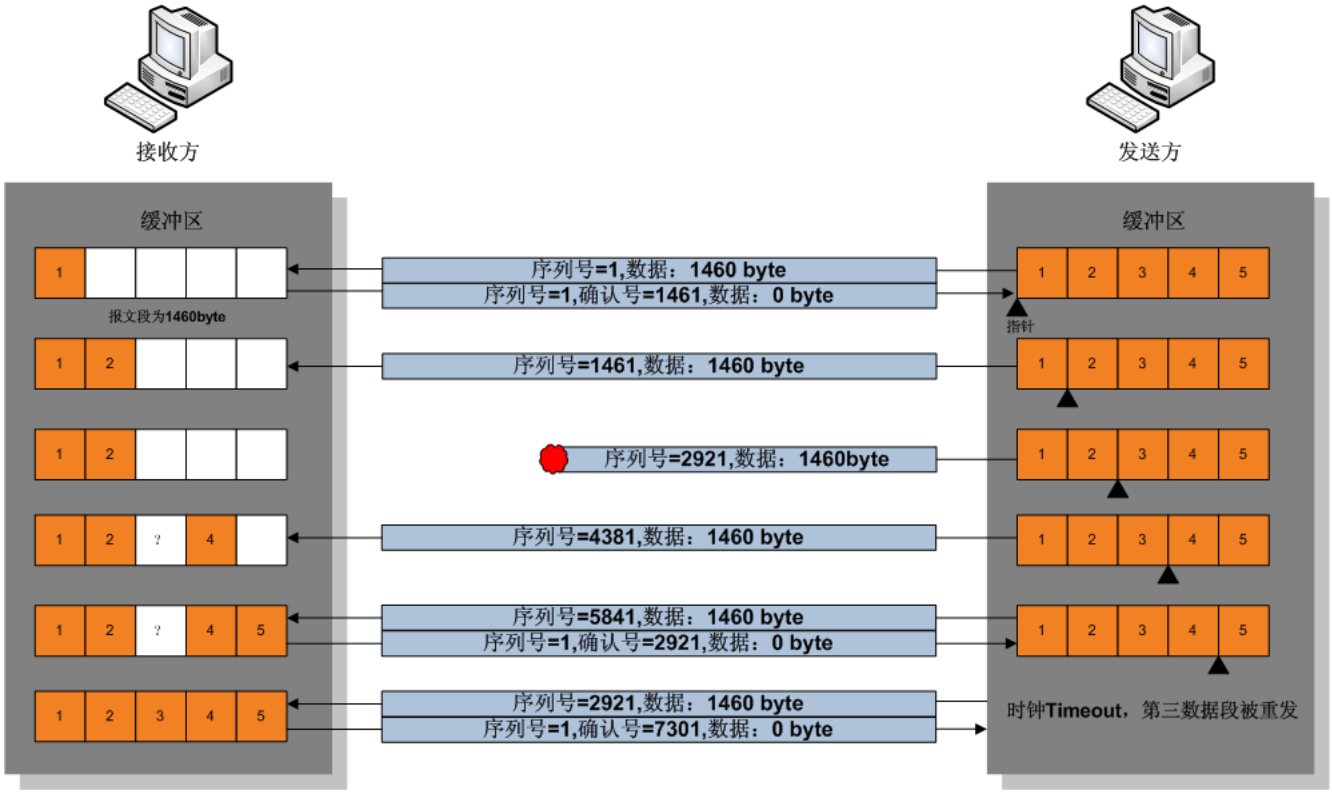

图 12.2: TCP 数据传输

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。

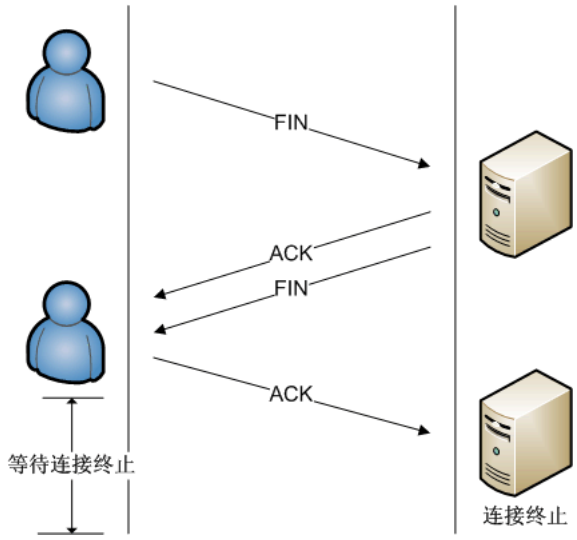

图 12.3: TCP 连接的正常终止

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。

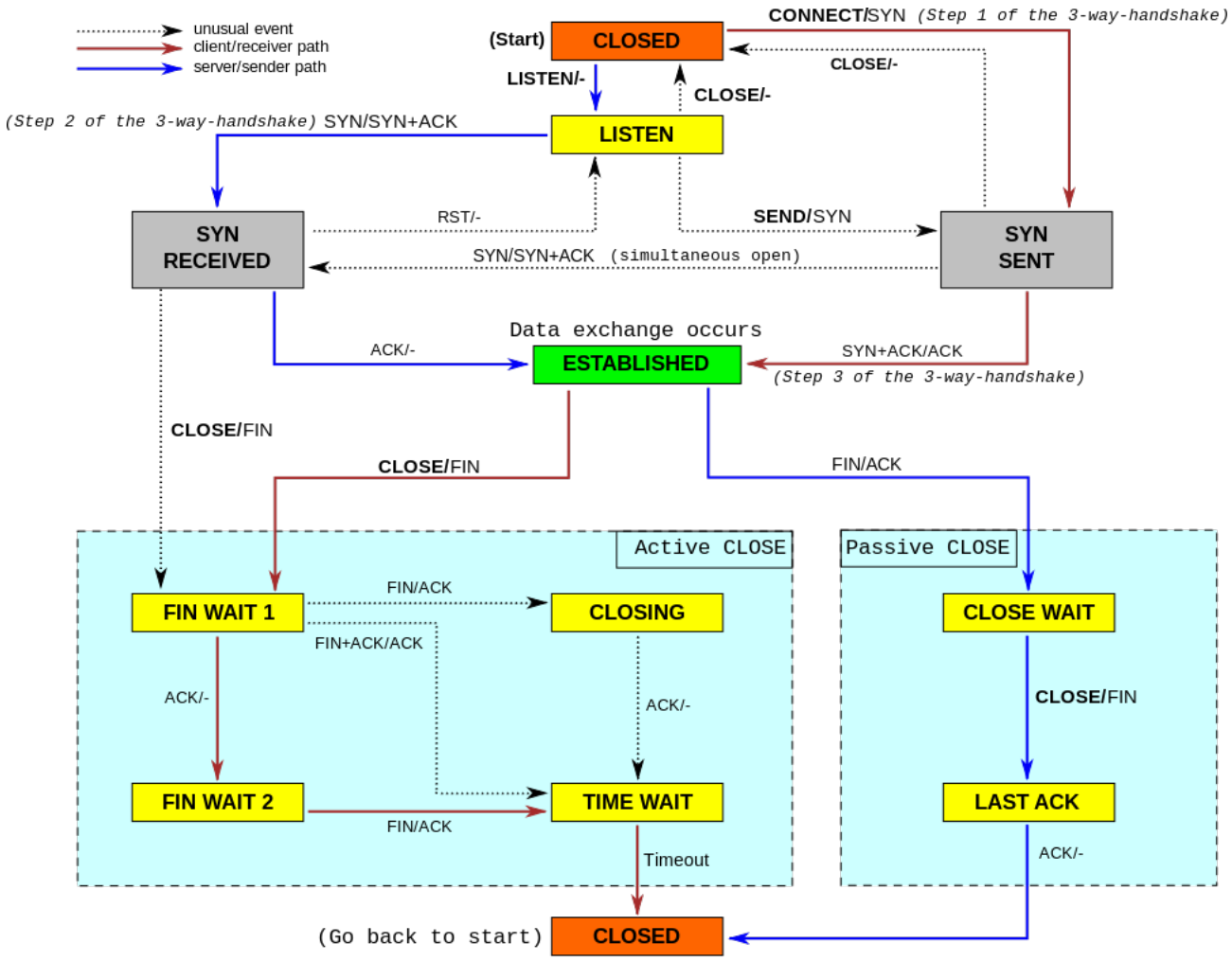

图 12.4: TCP 状态码列表

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
$tcpserver = new swoole_server("127.0.0.1", 9501);

$tcpserver->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
$tcpserver->on('connect', function ($tcpserver, $fd){
    echo "Client:Connect.\n";
});

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

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

//启动服务器
$tcpserver->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。

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

12.4.1 Synchronous TCP Client

下面的示例创建一个 TCP 的同步客户端，该客户端可以用来连接到 TCP 服务器，并且向 TCP 服务器发送一个字符串“hello world”字符串，如果服务器接收到该字符串，则返回一个“Server: hello world”字符串。

```
$sync_tcp_client = new swoole_client(SWOOLE_SOCK_TCP);

// 连接到服务器
if(!$sync_tcp_client->connect('127.0.0.1',9501,0.5)){
  die("connect failed.");
}

// 向服务器发送数据
if(!$sync_tcp_client->send("hello world")){
  die("send failed.");
}

// 从服务器接收数据
$data = $sync_tcp_client->recv();
if(!$data){
  die("recv failed.");
}
// 关闭连接
$sync_tcp_client->close();
```

同步 TCP 客户端是同步阻塞的，connect/send/recv 会等待 IO 完成后再返回。

同步阻塞操作并不消耗 CPU 资源，但是 IO 操作未完成当前进程会自动转入 sleep 模式，当 IO 完成后操作系统才会唤醒当前进程，继续向下执行代码。

- 如果建立连接到服务器需要 100ms，那么 `$client->connect` 就会耗时 100ms，因为 TCP 需要进行 3 次握手，所以 connect 至少需要 3 次网络通信。
- 在发送少量数据时 `$client->send` 都是可以立即返回的。发送大量数据时，socket 缓存区可能会塞满，send 操作会阻塞。
- recv 操作会阻塞等待服务器返回数据，recv 耗时等于服务器处理时间 + 网络传输耗时的总和。

在命令行中执行同步 TCP 客户端程序，结果如下：

```
$ php client.php
Server: hello world
```

12.4.2 Asynchronous TCP Client

异步 TCP 客户端与同步 TCP 客户端的不同在于，异步客户端是非阻塞的，因此可以用于编写高并发的程序。

```
$async_tcp_client = new swoole_client(SWOOLE_SOCK_TCP,SWOOLE_SOCK_ASYNC);

// 注册连接成功时的回调函数
$async_tcp_client->on('connect',function($cli){
  $cli->send("hello world\n");
});

// 注册接收数据时的回调函数
$async_tcp_client->on('receive', function($cli, $data){
  echo "Received: " . $data . "\n";
});

// 注册连接失败时的回调函数
$async_tcp_client->on('error',function($cli){
  echo "Connect failed\n";
});

// 注册连接关闭时的回调函数
$async_tcp_client->on('close',function($cli){
  echo "Connection close\n";
});

// 发起连接
$async_tcp_client->connect('127.0.0.1',9501,0.5);
```

swoole 官方提供的 redis-async、mysql-async 都是基于异步 swoole_client 实现的。

异步客户端需要设置回调函数，有 4 个事件回调必须设置 onConnect、onError、onReceive、onClose，分别在客户端连接成功、连接失败、收到数据、连接关闭时触发。

`$client->connect()` 发起连接的操作会立即返回，不存在任何等待。当对应的 IO 事件完成后，swoole 底层会自动调用设置好的回调函数。

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
$udpserver = new swoole_server("127.0.0.1", 9502, SWOOLE_PROCESS,
    SWOOLE_SOCK_UDP);
$udpserver->set(['worker_num' => 1]);

//监听数据发送事件
$udpserver->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
$udpserver->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
$udpclient = new swoole_client(SWOOLE_SOCK_UDP,SWOOLE_SOCK_SYNC);
$udpclient->connect('127.0.0.1',9502);
$udpclient->send("UDP Connection from bz");
echo $udpclient->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 HTTP Secure

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

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

$httpserver->on('Request', function($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>");
});

$httpserver->start();
```

Http 服务器只需要关注请求响应即可，所以只需要监听一个 onRequest 事件。当有新的 Http 请求进入就会触发此事件。

```
$httpserver = new swoole_http_server("0.0.0.0",9501);

$httpserver->on('request',function(){
  var_dump($request->get,$request->post);
  $response->header("Content-Type","text/html; charset=utf-8");
  $response->header("<h1>Hello Swoole. # " . rand(1000,9999) . "</h1>");
});
$httpserver->start();
```

onRequest 事件回调函数有 2 个参数，分别是 $request 对象和 $response 对象。

- $request 对象，包含了请求的相关信息，如 GET/POST 请求的数据。
- $response 对象，对 request 的响应可以通过操作 response 对象来完成。

`$response->end()` 方法表示输出一段 HTML 内容，并结束此请求。

- 0.0.0.0 表示监听所有 IP 地址，一台服务器可能同时有多个 IP，如 127.0.0.1 本地回环 IP、192.168.1.100 局域网 IP、210.127.20.2 外网 IP，这里也可以单独指定监听一个 IP[2]
- 9501 监听的端口，如果被占用程序会抛出致命错误，中断执行。

下面在命令行中启动 HTTP 服务器，并访问 http://127.0.0.1:9501 来检查返回的 HTTP 响应，而且也可以使用 ab 工具来对服务器进行压力测试。

14.3 HTTPS Server

[2]例如，0.0.0.0 表示监听所有 IP 地址，包括 127.0.0.1、192.168.x.x 以及外网 IP。

Chapter 15

WebSocket Server

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

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

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

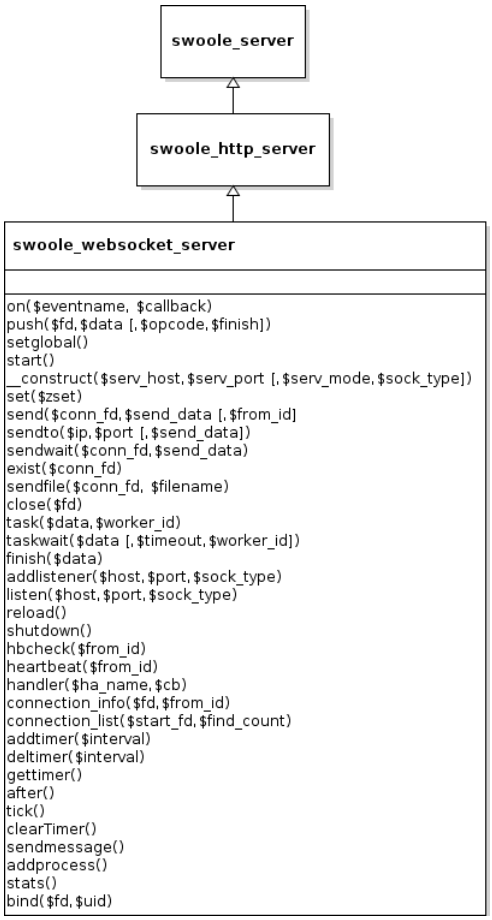

图 15.1: swoole_websocket_server 类

- `$fd`：TCP 连接的文件描述符（file description），在 swoole_server 中是客户端的唯一标识符。
- `$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
/**
 * 创建WebSocket Serverd对象$wsserver，并监听0.0.0.0的9502端口
 */
$wsserver = new swoole_websocket_server("0.0.0.0", 9502);

/**
 * 注册Server的事件回调函数open，并监听WebSocket连接打开事件
 */
$wsserver->on('open',function($ws,$request){
  var_dump($request->fd,$request->get,$request->server);
  $ws->push($request->fd, "hello welcome\n");
});

/**
 * 注册Server的事件回调函数message，并监听WebSocket消息事件
 */
$wsserver->('message',function($ws, $frame){
  echo "Message: {$frame->data}\n";
  $ws->push($frame->fd,"server: {$frame->data}");
});

/**
 * 注册Server的事件回调函数close，并监听WebSocket连接关闭事件
 */
$wsserver->on('close',function($ws,$fd){
  echo "client->{$fd} is closed\n";
});

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

WebSocket 服务器是建立在 Http 服务器之上的长连接服务器，客户端首先会发送一个 Http 的请求与服务器进行握手（handshake）。

如果握手成功，则会触发 onOpen 事件，表示连接已就绪，onOpen 函数中可以得到 $request 对象，其中包含了 Http 握手的相关信息（例如 GET 参数、Cookie、Http 头信息等）。

在 Websocket 客户端和 WebSocket 服务器建立连接后，二者就可以双向通信了，其中：

- 客户端向服务器端发送信息时，服务器端触发 onMessage 事件回调；
- 服务器端可以调用 `$server->push()` 向某个客户端（使用 $fd 标识符）发送消息。

在命令行运行 WebSocket 服务器端程序，并使用下面的 JavaScript 代码在 Chrome 浏览器进行测试。

```
var wsServer = 'ws://0.0.0.0:9501';
var websocket = new WebSocket(wsServer);
websocket.onopen = function(evt){
  console.log("Connected to WebSocket server.");
};

websocket.onclose = function(evt){
  console.log("Disconnected");
};

websocket.onmessage = function(evt){
  console.log('Retrieved data from server: ' + evt.data);
};

websocket.onerror = function(evt,e){
  console.log('Error occured:' + evt.data);
}
```

WebSocket 服务器除了提供 WebSocket 功能之外，实际上也可以处理 Http 长连接，只需要增加 onRequest 事件监听即可实现 Comet 方案 Http 长轮询。

15.2 WSS Server

```
<?php
$wssserver = new
    swoole_websocket_server("0.0.0.0",9527,SWOOLE_PROCESS,SWOOLE_SOCK_TCP|SWOOLE_SSL);
$wssserver->set(
  array(
    "reactor_num" => 4,
    "worker_num" => 4,
    "max_request" => 50,
    "daemonize" => 1,
    "log_file" => "/tmp/swoole.log",
```

```
    "ssl_cert_file" => "/etc/nginx/dev.crt",
    "ssl_key_file" => "/etc/nginx/bz-inc.com.key"
  )
);

$wssserver->on('Start',function(swoole_websocket_server $server){
  swoole_set_process_name('php swoole_websocket_server : master');
});

$wssserver->on('ManagerStart',function(swoole_websocket_server $server){
  swoole_set_process_name('php swoole_websocket_server : manager');
});

$wssserver->on('WorkerStart', function(swoole_websocket_server $server,
    $worker_id){
  if($worker_id >= $server->setting['worker_num']){
    swoole_set_process_name("php swoole_websocket_server : task worker " .
        $worker_id);
  }else{
    swoole_set_process_name("php swoole_websocket_server : event worker " .
        $worker_id);
  }
});

$wssserver->on('Open', function($server, $req){
  echo date('Ymd H:i:s') . " connection open: " . $req->fd;
  echo "\n";
});

$wssserver->('Messager',function($server,$frame){
  echo date('Ymd H:i:s') . " messager: " . $frame->data;
  $server->push($frame->fd,json_encode(["hello","world"]));
  echo "\n";
});

$wssserver->on('Close', function($server,$fd){
  echo date('Ymd H:i:s') . " connection close: " . $fd;
  echo "\n";
});

$wssserver->start();
?>
```

Chapter 16

WebSocket Client

```
class WebSocketClient
{
    const VERSION = '0.1.4';
    const TOKEN_LENGHT = 16;
    const TYPE_ID_WELCOME = 0;
    const TYPE_ID_PREFIX = 1;
    const TYPE_ID_CALL = 2;
    const TYPE_ID_CALLRESULT = 3;
    const TYPE_ID_ERROR = 4;
    const TYPE_ID_SUBSCRIBE = 5;
    const TYPE_ID_UNSUBSCRIBE = 6;
    const TYPE_ID_PUBLISH = 7;
    const TYPE_ID_EVENT = 8;

    private $key;
    private $host;
    private $port;
    private $path;

    /**
     * @var swoole_client
     */
    private $socket;
    private $buffer = '';
    private $origin = null;

    /**
     * @var bool
     */
    private $connected = false;

    /**
```

```
     * @param string $host
     * @param int $port
     * @param string $path
     */
    function __construct($host='127.0.0.1',$port=8080,$path='/',$origin=null)
    {
        $this->host = $host;
        $this->port = $port;
        $this->path = $path;
        $this->origin = $origin;
        $this->key = $this->generateToken(self::TOKEN_LENGHT);
    }

    /**
     * Disconnect on destruct
     */
    function __destruct()
    {
        $this->disconnect();
    }

    /**
     * Connect client to server
     *
     * @return $this
     */
    public function connect()
    {
        $this->socket = new \swoole_client(SWOOLE_SOCK_TCP);
        if(!$this->socket->connect($this->host,$this->port))
        {
            return false;
        }
        $this->socket->send($this->createHeader());
        return $this->recv();
    }

    public function getSocket()
    {
        return $this->socket;
    }

    /**
     * Disconnect from server
     */
```

```
public function disconnect()
{
    $this->connected = false;
    $this->socket->close();
}

public function recv(){
    $data = $this->socket->recv();
    if($data === false)
    {
        echo "Error: {$this->socket->errMsg}";
        return false;
    }
    $this->buffer .= $data;
    $recv_data = $this->parseData($this->buffer);
    if($recv_data)
    {
        $this->buffer = '';
        return $recv_data;
    }
    else
    {
        return false;
    }
}

/**
 * @param        $data
 * @param string $type
 * @param bool|true $masked
 *
 * @return bool
 */
public function send($data,$type = 'text',$masked = true)
{
    return $this->socket->send($this->hybi10Encode($data,$type,$masked));
}

/**
 * Parse received data
 *
 * @param $response
 *
 * @return null|string
 * @throws \Exception
```

```
 */
public function parseData($response)
{
    if(!$this->connected && isset($response['Sec-Websocket-Accept']))
    {
        if(base64_encode(pack('H*',sha1($this->key .
            '258EAFA5-E914-47DA-95CA-C5AB0DC85B11'))) ===
            $response['Sec-Websocket-Accept'])
        {
            $this->connected = true;
        }
        else
        {
            throw new \Exception("error response key.");
        }
    }
    return $this->hybi10Decode($response);
}

/**
 * Create header for websocket client
 *
 * @return string
 */
private function createHeader()
{
    $host = $this->host;
    if($host = '127.0.0.1' || $host = '0.0.0.0')
    {
        $host = 'localhost';
    }
    return "GET {$this->path} HTTP/1.1" . "\r\n" .
    "Origin: {$this->origin}" . "\r\n" .
    "Host: {$host}:{$this->port}" . "\r\n" .
    "Sec-WebSocket-Key: {$this->key}" . "\r\n" .
    "User-Agent: PHPWebSocketClient/" . self::VERSION . "\r\n" .
    "Upgrade: websocket" . "\r\n" .
    "Connection: Upgrade" . "\r\n" .
    "Sec-WebSocket-Protocol: wamp" . "\r\n" .
    "Sec-WebSocket-Version: 13" . "\r\n" . "\r\n";
}

/**
 * Parse raw incoming data
 *
```

```
 * @param $header
 *
 * @return array
 */
private function parseIncomingRaw($header)
{
    $retval = array();
    $content = "";
    $fields = explode("\r\n",preg_replace('/\x0D\x0A[\x09\x20]+/', '
        ',$header));
    foreach ($fields as $field) {
        if(preg_match('/([^:]+): (.+)/m', $field, $match))
        {
            $match[1] = preg_replace_callback('/(?<=^|[\x09\x20\x2D])./',
                function($matches){
                    return strtoupper($matches[0]);
                },
                strtolower(trim($match[1])));
            if(isset($retval[$match[1]]))
            {
                $retval[$match[1]] = array($retval[$match[1]], $match[2]);
            }
            else
            {
                $retval[$match[1]] = trim($match[2]);
            }
        }
        else
        {
            if (preg_match('!HTTP/1\.\d (\d)* .!', $field))
            {
                $retval["status"] = $field;
            }
            else
            {
                $content .= $field . "\r\n";
            }
        }
    }
    $retval['content']=$content;
    return $retval;
}
/**
 * Generate token
 *
```

```
     * @param int $length
     *
     * @return string
     */
    private function generateToken($length)
    {
        $characters =
            'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"§$%&/()=[]{}';
        $useChars = array();
        // select some random chars:
        for ($i = 0; $i < $length; $i++)
        {
            $useChars[] = $characters[mt_rand(0, strlen($characters) - 1)];
        }
        // Add numbers
        array_push($useChars, rand(0, 9), rand(0, 9), rand(0, 9));
        shuffle($useChars);
        $randomString = trim(implode('', $useChars));
        $randomString = substr($randomString, 0, self::TOKEN_LENGHT);
        return base64_encode($randomString);
    }
    /**
     * Generate token
     *
     * @param int $length
     *
     * @return string
     */
    public function generateAlphaNumToken($length)
    {
        $characters =
            str_split('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789');
        srand((float)microtime() * 1000000);
        $token = '';
        do
        {
            shuffle($characters);
            $token .= $characters[mt_rand(0, (count($characters) - 1))];
        } while (strlen($token) < $length);
        return $token;
    }
    /**
     * @param      $payload
     * @param string $type
     * @param bool $masked
```

```
     *
     * @return bool|string
     */
    private function hybi10Encode($payload, $type = 'text', $masked = true)
    {
        $frameHead = array();
        $frame = '';
        $payloadLength = strlen($payload);
        switch ($type)
        {
            case 'text':
                // first byte indicates FIN, Text-Frame (10000001):
                $frameHead[0] = 129;
                break;
            case 'close':
                // first byte indicates FIN, Close Frame(10001000):
                $frameHead[0] = 136;
                break;
            case 'ping':
                // first byte indicates FIN, Ping frame (10001001):
                $frameHead[0] = 137;
                break;
            case 'pong':
                // first byte indicates FIN, Pong frame (10001010):
                $frameHead[0] = 138;
                break;
        }
        // set mask and payload length (using 1, 3 or 9 bytes)
        if ($payloadLength > 65535)
        {
            $payloadLengthBin = str_split(sprintf('%064b', $payloadLength), 8);
            $frameHead[1] = ($masked === true) ? 255 : 127;
            for ($i = 0; $i < 8; $i++)
            {
                $frameHead[$i + 2] = bindec($payloadLengthBin[$i]);
            }
            // most significant bit MUST be 0 (close connection if frame too big)
            if ($frameHead[2] > 127)
            {
                $this->close(1004);
                return false;
            }
        }
        elseif ($payloadLength > 125)
        {
```

```
            $payloadLengthBin = str_split(sprintf('%016b', $payloadLength), 8);
            $frameHead[1] = ($masked === true) ? 254 : 126;
            $frameHead[2] = bindec($payloadLengthBin[0]);
            $frameHead[3] = bindec($payloadLengthBin[1]);
        }
        else
        {
            $frameHead[1] = ($masked === true) ? $payloadLength + 128 :
                $payloadLength;
        }
        // convert frame-head to string:
        foreach (array_keys($frameHead) as $i)
        {
            $frameHead[$i] = chr($frameHead[$i]);
        }
        if ($masked === true)
        {
            // generate a random mask:
            $mask = array();
            for ($i = 0; $i < 4; $i++)
            {
                $mask[$i] = chr(rand(0, 255));
            }
            $frameHead = array_merge($frameHead, $mask);
        }
        $frame = implode('', $frameHead);
        // append payload to frame:
        for ($i = 0; $i < $payloadLength; $i++)
        {
            $frame .= ($masked === true) ? $payload[$i] ^ $mask[$i % 4] :
                $payload[$i];
        }
        return $frame;
    }
    /**
     * @param $data
     *
     * @return null|string
     */
    private function hybi10Decode($data)
    {
        if (empty($data))
        {
            return null;
        }
```

```
$bytes = $data;
$dataLength = '';
$mask = '';
$coded_data = '';
$decodedData = '';
$secondByte = sprintf('%08b', ord($bytes[1]));
$masked = ($secondByte[0] == '1') ? true : false;
$dataLength = ($masked === true) ? ord($bytes[1]) & 127 : ord($bytes[1]);
if ($masked === true)
{
    if ($dataLength === 126)
    {
        $mask = substr($bytes, 4, 4);
        $coded_data = substr($bytes, 8);
    }
    elseif ($dataLength === 127)
    {
        $mask = substr($bytes, 10, 4);
        $coded_data = substr($bytes, 14);
    }
    else
    {
        $mask = substr($bytes, 2, 4);
        $coded_data = substr($bytes, 6);
    }
    for ($i = 0; $i < strlen($coded_data); $i++)
    {
        $decodedData .= $coded_data[$i] ^ $mask[$i % 4];
    }
}
else
{
    if ($dataLength === 126)
    {
        $decodedData = substr($bytes, 4);
    }
    elseif ($dataLength === 127)
    {
        $decodedData = substr($bytes, 10);
    }
    else
    {
        $decodedData = substr($bytes, 2);
    }
}
```

```
        return $decodedData;
    }
}
```

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();
```

Part III

Process

Chapter 19

Overview

19.1 Instruction

19.1.1 Parallel Computing

一般情况下，并行计算（parallel computing）是指许多指令得以同时进行的计算模式，并且在同时进行的前提下，可以将计算的过程分解成小部分，之后以并发方式来加以解决。

计算可以被分成数个运算步骤来运行，其中为了解决某个特定问题的算法可以以一连串指令运行来完成。

传统上，这些指令都被送至单一的中央处理器，以循序方式运行完成，那么在这种处理方式下，单一时间中只有单一指令被运行。并行运算采用了多个运算单元，同时运行来解决问题。

相对于串行计算，并行计算可以划分成时间并行和空间并行。其中，时间并行即流水线技术，空间并行使用多个处理器执行并发计算。

如果以程序和算法设计人员的角度看，并行计算又可分为数据并行和任务并行。其中，数据并行把大的任务化解成若干个相同的子任务，处理起来比任务并行简单。

当前研究的并行计算的主要方面是空间的并行问题，而且空间上的并行导致两类并行机的产生，即按照麦克·弗莱因（Michael Flynn）的说法来划分的单指令流多数据流（SIMD）和多指令流多数据流（MIMD）。

常用的串行机也称为单指令流单数据流（SISD），MIMD 类的机器又可分为常见的五类：并行矢量处理机（PVP）、对称多处理机（SMP）、大规模并行处理机（MPP）、工作站机群（COW）、分布式共享存储处理机（DSM）。

并行计算机有以下五种访存模型：

- 均匀访存模型（UMA）
- 非均匀访存模型（NUMA）
- 全高速缓存访存模型（COMA）
- 一致性高速缓存非均匀存储访问模型（CC-NUMA）
- 非远程存储访问模型（NORMA）

实际上，全世界基本上都在使用冯·诺伊曼的计算模型和 SISD（单指令流单数据流）计算机（即串行计算机），并行计算机并没有一个统一的计算模型，虽然人们已经提出了 PRAM 模型、BSP 模型、LogP 模型、C^3 模型等参考模型。

并行计算机是靠网络将各个处理机或处理器连接起来的，一般来说分为静态连接和动态连接。

1. 静态连接：一维线性连接，网孔连接，超立方体连接，树连接，立方环连接，洗牌交换连接，蝶形连接，金字塔连接等。
2. 动态连接：总线连接（Bus），交叉开关（CS），多级互联网络（MIN）。

和并行计算机一样，并行算法也是一门还没有发展成熟的学科，远远不及串行算法那样丰富，并行算法设计中最常用的的方法是 PCAM 方法，即划分，通信，组合，映射。

首先，划分就是将一个问题平均划分成若干份，并让各个处理器去同时执行；

其次，通信阶段就是要分析执行过程中所要交换的数据和任务的协调情况；

再次，组合则是要求将较小的问题组合到一起以提高性能和减少任务开销；

最后，映射则是要将任务分配到每一个处理器上。

总之，并行算法还需要相当多完善的地方，而且并行算法与串行算法最大的不同之处在于，并行算法不仅要考虑问题本身，而且还要考虑所使用的并行模型和网络连接等。

19.1.2 Concurrent Computing

并发计算（Concurrent computing，或译为并发处理、共时计算）是一种程序计算的形式，在系统中至少有两个以上的计算在同时运行，计算结果可能同时发生。

用来实现并发系统（Concurrent system）的编程语言与各种算法，统称为并发计算。其中，并发程序通常被设计为交互式的运算过程，因为它的运算过程是不确定的，在设计上的难度较高。

设计并发程序最大的挑战，在于确保不同运算运行步骤间的交互或是通讯，能以正确的顺序进行，同时也要确保在不同运行步骤间共享的资源，能够正确被访问。

简单来说，并发计算就是将一个计算任务，分区成几个小的部分，让它们同时被计算，之后再汇整计算结果以完成任务。

- 并发计算可以增加产出，让并发程序以并行方式运行，在某个特定时间内可以让数个进程同时完成计算任务来增加产出。
- 输入/输出的反应时间加快，密集进行输入/输出（I/O）操作的应用程序在多数的时间内都在等待输入或输出操作完成，并发计算编程可以在等待的时间中让另外的进程来运行。
- 更适当的程序架构。某些计算问题或问题的领域，特别适合以并发计算来解决。

并发计算跟并行计算（Parallel computing）与分布式计算的范围有重叠之处，虽然在概念上不同，但是常会让人混淆。

- 并发计算是一种程序运算的特性，可以被视为是并行运算的进一步抽象，它包涵了时间片这种可以被用来实现虚拟并行运算（pseudoparallelism）的技术，因此在实际的物理运作中，计算过程可能是并行，或非并行的。
- 并行计算是指当并发计算的程序，在机器平台上同时被运行的状况，因此并行计算是一种机器运算的形式之一。
- 分布式计算（Distributed computing）是并行计算的一个特例，它采用计算机网络来进行同步。

并发计算中的不同计算单元之间需要进行通讯以保持同步，在某些并发编程语言中，这些通讯方式是被隐藏起来的（例如利用 future 方式），并以外显方式来进行通讯，可分成两种主要方式：

1. 共享存储器通讯

   经由改变共享存储器地址内的数据内容，让不同的并发单元间进行通讯，例如 Java 与 C# 都支持这个方式。

   使用共享存储器通讯类型的并发程序，通常需要应用某种锁定的方式来达成线程间的同步，这些锁定技术包括 mutex、semaphore 或 monitor 等。
2. 消息传递通讯

通过消息的交换来使不同的并发单元间同步，例如 Scala、Erlang 与 occam。

实现并发程序有许多方式，根据编程语言与操作系统的支持方式，可以用进程或线程来实现。

并发程序可以运行在单一处理器上，将不同的运行步骤分散在不同时间片中运行，以非并行方式循序运算，或者也可以用并行计算来实现，将每个进程指定给处理器组中的某个处理器，以单片机多处理器平台，或是通过网络链接的分布式平台来实现。

按照硬件与操作系统的支持方式，并发计算可以在同一个进程中完成，在同一个进程中以多线程来完成，以多个进程来达成，甚至可以分成数个程序来运行。另外，可以用单处理器的循序计算来实现，也可以采并行计算或分布式计算方式实现。

以并行方式来运作并发程序，并不必然会增进运行效率。因为运行结果可能会因为平台特性而改变，也增加它在设计上的难度。

并发性编程语言使用编程语言结构特性进行并发，这些结构涉及到多线程、分布式计算、消息传递、资源共享（包括内存共享），因此也称这些语言是面向并发的编程语言（COPL）。

现在很多常用的语言都拥有并发的特性，例如 Java 和 C# 的语言底层都是通过内存共享以及锁监听机制的并发模型来实现（尽管消息传递模型如今也是基于内存共享模型实现的）。

在消息传递的并发模型世界中，Erlang 最具代表性并被广泛使用至今。

19.1.3 Distributed Computing

在计算机科学中，分布式计算（Distributed computing）主要研究分布式系统（Distributed system）如何进行计算。

分布式系统是一组计算机通过网络相互链接传递消息与通讯后并协调它们的行为而形成的系统，组件之间彼此进行交互以实现一个共同的目标。

分布式系统把需要进行大量计算的工程数据分区成小块，由多台计算机分别计算，再通过计算机网络上传运算结果后，将结果统一合并得出数据结论，因此分布式系统主要应用于面向服务的架构、大型多人在线游戏和对等网络应用等。

19.2 Process

从 swoole-1.7.2 开始增加了一个进程管理模块 Process 来替代 PHP 的 pcntl 扩展。

PHP 的进程控制 pcntl 扩展支持实现了 Unix[1] 方式的进程创建, 程序执行, 信号处理以及进程的中断，但是 pcntl 提供的进程控制不能被应用在 Web 服务器环境，当其被用于 Web 服务环境时可能会带来意外的结果。

pcntl 的缺陷在于其只提供了 fork 这样原始的接口，容易使用错误。

注意，fork 是创建了一个子进程，父进程和子进程都从 fork 的位置开始向下继续执行，不同的是父进程执行过程中得到的 fork 返回值为子进程号，而子进程得到的是 0。

在 PHP 中进程控制支持默认是关闭的，需要使用`--enable-pcntl` 配置选项重新编译 PHP 的 CGI 或 CLI 版本来打开进程控制支持。

PCNTL 现在使用了 ticks 作为信号处理的回调机制，ticks 在速度上远远超过了之前的处理机制。这个变化与“用户 ticks”遵循了相同的语义。您可以使用 declare() 语句在程序中指定允许发生回调的位置。这使得我们对异步事件处理的开销最小化。在编译 PHP 时启用 pcntl 将始终承担这种开销，不论您的脚本中是否真正使用了 pcntl。

[1] pctl 扩展模块没有非 Unix 平台可用的函数（即非 Unix 类系统不支持此模块），因此在 Windows 平台上不可用。

有一个调整是 PHP 4.3.0 之前的所有 pcntl 脚本要使其工作，要么在期望允许回调的（代码）部分使用 declare()，要么使用 declare() 新的全局语法使其在整个脚本范围有效。

- pcntl_alarm —为进程设置一个 alarm 闹钟信号
- pcntl_errno —别名 pcntl_strerror
- pcntl_exec —在当前进程空间执行指定程序
- pcntl_fork —在当前进程当前位置产生分支（子进程）。
- pcntl_get_last_error —Retrieve the error number set by the last pcntl function which failed
- pcntl_getpriority —获取任意进程的优先级
- pcntl_setpriority —修改任意进程的优先级
- pcntl_signal_dispatch —调用等待信号的处理器
- pcntl_signal —安装一个信号处理器
- pcntl_sigprocmask —设置或检索阻塞信号
- pcntl_sigtimedwait —带超时机制的信号等待
- pcntl_sigwaitinfo —等待信号
- pcntl_strerror —Retrieve the system error message associated with the given errno
- pcntl_wait —等待或返回 fork 的子进程状态
- pcntl_waitpid —等待或返回 fork 的子进程状态
- pcntl_wexitstatus —返回一个中断的子进程的返回代码
- pcntl_wifexited —检查状态代码是否代表一个正常的退出。
- pcntl_wifsignaled —检查子进程状态码是否代表由于某个信号而中断
- pcntl_wifstopped —检查子进程当前是否已经停止
- pcntl_wstopsig —返回导致子进程停止的信号
- pcntl_wtermsig —返回导致子进程中断的信号

这个 pcntl 示例用于产生一个守护进程并可以通过信号处理进行关闭。

另外，System_Daemon（http://pear.php.net/package/System_Daemon）也可以用来使用 PHP 创建后台进程。

```
<?php
// Include PEAR's Daemon Class
require_once "System/Daemon.php";

// Bare minimum setup
System_Daemon::setOption("appName", "mydaemonname");

// Spawn Deamon!
System_Daemon::start();

// Your PHP Here!
while (true) {
    doTask();
}

// Stop daemon!
System_Daemon::stop();
```

进程控制示例

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

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

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

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

// 执行无限循环任务
while (1) {
    // do something interesting here
}

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

```
?>
```

19.3 Signal

19.3.1 Mac OS X

```
SIGHUP = 1
SIGINT = 2
SIGQUIT = 3
SIGILL = 4
SIGTRAP = 5
SIGABRT = 6
SIGIOT = 6
SIGBUS = 10
SIGFPE = 8
SIGUSR1 = 30
SIGSEGV = 11
SIGUSR2 = 31
SIGPIPE = 13
SIGALRM = 14
SIGTERM = 15
SIGSTKFLT not defined
SIGCLD not defined
SIGCHLD = 20
SIGCONT = 19
SIGTSTP = 18
SIGTTIN = 21
SIGTTOU = 22
SIGURG = 16
SIGXCPU = 24
SIGXFSZ = 25
SIGVTALRM = 26
SIGPROF = 27
SIGWINCH = 28
SIGPOLL not defined
SIGIO = 23
SIGPWR not defined
SIGSYS = 12
SIGBABY = 12
SIG_BLOCK = 1
SIG_UNBLOCK = 2
SIG_SETMASK = 3
```

19.3.2 Linux

```
$ kill -l
1) SIGHUP     2) SIGINT     3) SIGQUIT    4) SIGILL     5) SIGTRAP
6) SIGABRT    7) SIGBUS     8) SIGFPE     9) SIGKILL   10) SIGUSR1
11) SIGSEGV   12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD   18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU   23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF   28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS    34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

19.3.3 RedHat

```
$ kill -l
Signal SIGHUP = 1
Signal SIGINT = 2
Signal SIGQUIT = 3
Signal SIGILL = 4
Signal SIGTRAP = 5
Signal SIGABRT = 6
Signal SIGIOT = 6
Signal SIGBUS = 7
Signal SIGFPE = 8
Signal SIGUSR1 = 10
Signal SIGSEGV = 11
Signal SIGUSR2 = 12
Signal SIGPIPE = 13
Signal SIGALRM = 14
Signal SIGTERM = 15
Signal SIGSTKFLT = 16
Signal SIGCLD = 17
Signal SIGCHLD = 17
Signal SIGCONT = 18
Signal SIGTSTP = 20
Signal SIGTTIN = 21
Signal SIGTTOU = 22
Signal SIGURG = 23
Signal SIGXCPU = 24
Signal SIGXFSZ = 25
Signal SIGVTALRM = 26
Signal SIGPROF = 27
```

```
Signal SIGWINCH = 28
Signal SIGPOLL = 29
Signal SIGIO = 29
Signal SIGPWR = 30
Signal SIGSYS = 31
Signal SIGBABY = 31
Signal SIG_BLOCK = 0
Signal SIG_UNBLOCK = 1
Signal SIG_SETMASK = 2
```

19.3.4 Ubuntu

```
$ kill -l
1) SIGHUP    2) SIGINT    3) SIGQUIT    4) SIGILL
5) SIGTRAP    6) SIGABRT    7) SIGBUS    8) SIGFPE
9) SIGKILL  10) SIGUSR1  11) SIGSEGV  12) SIGUSR2
13) SIGPIPE  14) SIGALRM  15) SIGTERM  17) SIGCHLD
18) SIGCONT  19) SIGSTOP  20) SIGTSTP  21) SIGTTIN
22) SIGTTOU  23) SIGURG    24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH  29) SIGIO
30) SIGPWR    31) SIGSYS    33) SIGRTMIN  34) SIGRTMIN+1
35) SIGRTMIN+2 36) SIGRTMIN+3 37) SIGRTMIN+4 38) SIGRTMIN+5
39) SIGRTMIN+6 40) SIGRTMIN+7 41) SIGRTMIN+8 42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6
59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX

SIG_IGN, SIG_DFL, SIG_ERR are no real signals
```

19.4 Process

pcntl 本身存在很多不足，例如：

- pcntl 无法用在 fpm/apache 中，只能用于 CLI[2]模式。
- pcntl 没有提供进程间通信的功能
- pcntl 不支持重定向标准输入和输出
- pcntl 只提供了 fork 这样原始的接口，容易使用错误

相比而言，swoole_process 提供了比 pcntl 更强大的功能，更易用的 API，使 PHP 在多进程编程方面更加轻松。更重要的是，swoole_process 可以安全地用于 fpm/apache 环境下。

[2]pcntl only compiles in to the CLI version of PHP, not the Apache module，and `function_exists('pcntl_fork')` returns true just fine from the CLI, and only returns false for HTTP requests. The same is true of ALL of the `pcntl_*()` functions.

- swoole_process 提供了基于 unixsock 的进程间通信，只需调用 write/read 或者 push/pop 即可。
- swoole_process 支持重定向标准输入和输出，在子进程内 echo 不会打印到屏幕，而是写入管道，读键盘输入可以重定向为管道来读取数据。
- swoole_process 允许用于 fpm/apache 的 Web 请求中。
- swoole_process 配合 swoole_event 模块来创建的 PHP 子进程可以支持异步的事件驱动模式。
- swoole_process 提供了 exec 接口，创建的进程可以执行其他程序，而且与原 PHP 父进程之间可以方便的通信

例如，如果是其他类里的其他方法，可以使用匿名函数并在里面在调用对象方法。

```
$process = new \swoole_process(function(swoole_process $worker){
  GateWay::serverFunc($worker);
} ,FALSE );
```

19.4.1 worker process

```
$worker_num = 8;

for($i = 0; $i < $worker_num; $i++)
{
    $process = new swoole_process('callback_function', true);
    $pid = $process->start();
    $workers[$pid] = $process;
}

foreach($workers as $pid => $process)
{
    $process->write("hello worker[$pid]\n");
    echo "From Worker: ".$process->read();
}

for($i = 0; $i < $worker_num; $i++)
{
    $ret = swoole_process::wait();
    $pid = $ret['pid'];
    unset($workers[$pid]);
    echo "Worker Exit, PID=".$pid.PHP_EOL;
}

function callback_function(swoole_process $worker)
{
    //echo "Worker: start. PID=".$worker->pid."\n";
    //recv data from master
    $recv = $worker->read();
    echo "From Master: $recv\n";
```

```
    //send data to master
    $worker->write("hello master\n");

    sleep(2);
    $worker->exit(0);
}
```

19.4.2 child process

子进程事件驱动模式

```
function callback_function_async(swoole_process $worker)
{
    //echo "Worker: start. PID=".$worker->pid."\n";
    //recv data from master
    $GLOBALS['worker'] = $worker;
    swoole_event_add($worker->pipe, function($pipe) {
        $worker = $GLOBALS['worker'];
        $recv = $worker->read();

        echo "From Master: $recv\n";

        //send data to master
        $worker->write("hello master\n");

        sleep(2);

        $worker->exit(0);
    });
}
```

19.4.3 python process

PHP 创建一个 Python 子进程，并与之通信

```
$process = new swoole_process('pyhon_process', true);
$pid = $process->start();

function pyhon_process(swoole_process $worker)
{
    $worker->exec('/usr/bin/python', array("echo.py"));
}

$process->write("hello world\n");
echo $process->read();
```

```
$ret = swoole_process::wait();
var_dump($ret);
```

Python 程序 echo.py

```
import sys

def main():
    s = raw_input()
    print "Python:" + s

main()
```

Part IV

Timer

Chapter 20

Overview

20.1 Countdown

倒计时（或倒数）指用计时方式以时间减少直到计时器为零或停止的形式。

```
<?php
swoole_timer_add(1000, function($interval) {
    echo "timer[$interval] :".microtime(true)." called\n";
});

echo "Added timer1: ".microtime(true)."\n";

swoole_timer_add(3000, function($interval) {
    echo "timer[$interval] :".microtime(true)." called\n";
    static $remove = false;
    if (!$remove) {
    swoole_timer_after(10000, function(){
    echo microtime(true)." Timeout, clear interval\n";
      swoole_timer_del(3000);
    });
    $remove = true;
  }
});

echo "Added timer2: ".microtime(true)."\n";
```

Chapter 21

Timer

swoole 提供了类似 JavaScript 的 setInterval/setTimeout 异步高精度定时器，精度为毫秒级。

```
//每隔2000ms触发一次
swoole_timer_tick(2000,function($timer_id){
  echo "tick-2000ms\n";
});

// 3000ms后执行该函数
swoole_timer_after(3000,function(){
  echo "after 3000ms.\n";
});
```

- swoole_timer_tick 函数就相当于 setInterval，是持续触发的
- swoole_timer_after 函数相当于 setTimeout，仅在约定的时间触发一次

swoole_timer_tick/swoole_timer_after 函数会返回一个整数，表示定时器的 ID。

swoole_timer_clear 可以用来清除定时器，参数为定时器 ID。

Part V

ASYNC

Chapter 22

Overview

计算机的通信方式包括并行（Parallel）通信和串行（Serial）通信。

22.1 Serial Communication

在远程通信和计算机科学中，串行通信（Serial communication）是指在计算机总线或其他数据通道上，每次传输一个位元数据，并连续进行以上单次过程的通信方式。

串行通信被用于长距离通信以及大多数计算机网络，在这些应用场合里，电缆和同步化使并行通信实际应用面临困难。

凭借着串行通信改善的信号完整性和传播速度，串行通信总线正在变得越来越普遍，甚至在短程距离的应用中，其优越性已经开始超越并行总线不需要串行化元件 (serializer)，并解决了时钟偏移（Clock skew）、互联密度（interconnect density）等缺点。例如，PCI 到 PCI Express 的升级就是一个串行通信的例子。

最初，如果集成电路具有更多的引脚的话，那么它的价格通常会更加昂贵。为了减少封装中的引脚数，许多集成电路在速度不是特别重要的情况下，使用串行总线来传输数据。这样的低价串行总线的例子有摩尔斯电码、以太网、MIDI、USB、IEEE 1394、PCI Express、SATA、SPI、I^2C、UNI/O 和 1-Wire 等[1]。

在计算机之间、计算机内部各部分之间，通信可以以串行和并行的方式进行。一个并行连接通过多个通道（例如导线、印制电路布线和光纤）在同一时间内传播多个数据流，而一个串行在同一时间内只连接传输一个数据流。

虽然串行连接单个时钟周期能够传输的数据比并行数据更少，前者传输能力看起来比后者要弱一些，实际的情况却常常是串行通信可以比并行通信更容易提高通信时钟频率，从而提高数据的传输速率。

有以下一些因素允许串行通信具有更高的通信时钟频率：

- 无需考虑不同通道的时钟脉冲相位差（clock skew）；
- 串行连接所需的物理介质，例如电缆和光纤，少于并行通信，从而减少占用空间的体积；

[1]具体来说，串行通信架构的例子包括摩尔斯电码（用于电报）、RS-232（低速，用于串行接口）、RS-422、RS-423、RS-485、I^2C、SPI、ARINC 818Avionics 数字视频总线、通用串行总线（中速，用于连接计算机和多种外部设备）、IEEE 1394、以太网、纤维管路（高速，用于连接计算机和大容量存储器）、InfiniBand（超高速，在规模上类似于 PCI 接口）、MIDI 数字乐器控制、DMX512 舞台灯光控制、SDI-12 工业传感器协议、串行 SCSI、SATA、SpaceWire 航天器通信网络、HyperTransport、PCI Express、同步光网络（光纤高速传输）、T-1 和 E-1 变体（通过铜线对的高速通信）、MIL-STD-1553A/B。

- 串扰的问题可以得到大幅度缓解。

在许多情况里，串行通信都凭借其更低廉的部署成本成为更佳的选择，尤其是在远距离传输中，而且许多集成电路都具有串行通信接口来减少引脚数量来节约成本。

22.2 Parallel Communication

与串行通信对应的是并行通信，它在串行端口上通过一次同时传输若干位元数据的方式进行通信。

具体来说，并行通信是指 8 位数据同时通过并行线进行传送，这样数据传送速度大大提高，但是并行传送的线路长度受到限制，长度增加导致干扰也会增加，数据也就容易出错，因此并行通信通常用于单机。

22.3 Network

计算机科学中的异步通信（Asynchronous conferencing）特指以计算机为媒介，沟通，协作和学习，在互动贡献者中有一定延迟的技术。与之相对的是同步通信，广泛应用于各种“聊天”系统的用户“实时”同步通信。

异步传输模式（Asynchronous Transfer Mode，ATM）又叫信元中继，采用电路交换的方式，并以信元（cell）为单位。每个信元长 53 字节，其中报头占了 5 字节。

ATM 能够比较理想地实现各种 QoS，既能够支持有连接的业务，又能支持无连接的业务，因此成为了宽带 ISDN（B-ISDN）技术的典范。

作为一种交换技术，ATM 在发送数据时，先将数字数据切割成多个固定长度的数据包，之后利用光纤或 DS1/ DS3 发送。到达目的地后，再重新组合。

ATM 网络可同时将声音、视频及数据集成在一起，从而可以针对各种信息型态提供最佳的传输环境。

HTTP/2 是 HTTP 1.1 基于 SPDY 协议的更新[2]，其目标包括异步连接复用，头压缩和请求反馈管线化并保留与 HTTP 1.1 的完全语义兼容。

22.3.1 Protocol

22.4 Process

进程（process）是计算机中已运行程序的实体，也是分时系统的基本运作单位。

- 在面向进程设计的系统（例如早期的 UNIX，Linux 2.4 及更早的版本），进程是程序的基本执行实体。
- 在面向线程设计的系统（例如当代多数操作系统、Linux 2.6 及更新的版本），进程本身不是基本运行单位，而是线程的容器。

程序本身只是指令、数据及其组织形式的描述，如果程序没有被计算机执行，那么程序就只是硬盘上的一个文件而已，操作系统在计算机关闭后也仅仅是硬盘上的一组文件。

[2]httpbis 工作小组最初考虑了 Google 的 SPDY 协议、微软的 SM 协议和 Network-Friendly HTTP 更新，Facebook 对各方案进行了评价并最终推荐了 SPDY 协议，HTTP 2.0 的首个草稿于 2012 年 11 月发布，其内容基本和 SPDY 协议相同。

进程才是程序（那些指令和数据）的真正运行实例，若干进程有可能与同一个程序相关系，而且每个进程都可以以同步（循序）或异步（平行）的方式独立运行。

现代计算机系统可在同一段时间内以进程的形式将多个程序加载到存储器中，并通过时间共享（或称时分复用）来在一个处理器上表现出同时（平行性）运行的感觉。

多线程是每一个线程都代表一个进程内的一个独立执行上下文，因此使用多线程技术的操作系统或计算机架构可以让同样程序的平行线程真正同时运行在在多 CPU 主机或网络上（以及在不同的 CPU 上）。

- 在批处理系统中，进程称为工作（jobs）；
- 在分时系统中，进程称为用户程序（user progams）或任务（tasks）。
- 在多数情况，工作与进程是同义词，但进程（process）已较为人接受。

当用户下达运行程序的命令后，就会产生进程。同一程序可产生多个进程（一对多关系），这样就可以允许同时有多位用户运行同一程序，却不会相冲突。

进程需要一些资源才能完成工作，例如 CPU 使用时间、存储器、文件以及 I/O 设备，并且为依序逐一进行，因此每个 CPU 核心任何时间内仅能运行一个进程。

具体来说，一个计算机系统进程包括（或者说“拥有”）下列数据：

- 那个程序的可运行机器码的一个在存储器的映像。
- 分配到的存储器（通常是虚拟的一个存储器区域）。存储器的内容包括可运行代码、特定于进程的数据（输入、输出）、调用堆栈、堆栈（用于保存运行时运输中途产生的数据）。
- 分配给该进程的资源的操作系统描述符，诸如文件描述符（Unix 术语）或文件句柄（Windows）、数据源和数据终端。
- 安全特性，诸如进程拥有者和进程的权限集（可以容许的操作）。
- 处理器状态（内文），诸如寄存器内容、物理存储器定址等。当进程正在运行时，状态通常存储在寄存器，其他情况在存储器。

进程在运行时，状态（state）会改变，可以将进程状态理解为进程当前的动作：

- 新生（new）：进程新产生中。
- 运行（running）：正在运行。
- 等待（waiting）：等待某事发生，例如等待用户输入完成。亦称“阻塞”（blocked）
- 就绪（ready）：排班中，等待 CPU。
- 结束（terminated）：完成运行。

进程的各个状态名称可能随不同操作系统而不同，例如对于单 CPU 系统，任何时间可能有多个进程为等待、就绪，但是必定仅有一个进程在运行。

22.4.1 IPC

IPC（Inter-Process Communication，进程间通信）指至少两个进程或线程间传送数据或信号的一些技术或方法。

最初，进程是计算机系统分配资源的最小单位（严格说来是线程），每个进程都有自己的一部分独立的系统资源，彼此是隔离的。

后来，为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。

在一个典型的例子中，使用进程间通信的两个应用可以被分类为客户端和服务器，其中客户端进程请求数据，服务端回复客户端的数据请求。

另外，有一些应用（例如分布式计算）本身既是服务器又是客户端，这些进程可以运行在同一台计算机上或网络连接的不同计算机上。

具体来说，进程间通信技术包括消息传递、同步、共享内存和远程过程调用，而且 IPC 已经成为一种标准的 Unix 通信机制，其实现机制如下：

表 22.1: 主要的 IPC 方法

参数名称	提供方（操作系统或其他环境）
文件 (File)	多数操作系统
信号 (Signals)	多数操作系统
套接字 (Sockets)	多数操作系统
消息队列 (Message queue)	多数操作系统
管道 (Pipeline)	所有的 POSIX 系统, Windows
命名管道 ()	所有的 POSIX 系统, Windows
信号量（Semaphore）	所有的 POSIX 系统, Windows
共享内存（Shared Memory）	所有的 POSIX 系统, Windows
消息传递（Message passing，不共享）	用于 MPI 规范，Java RMI, CORBA, MSMQ, MailSlot 以及其他
内存映射文件（Memory-mapped file）	所有的 POSIX 系统, Windows

进程间通信可以实现信息共享，例如 Web 服务器中通过浏览器使用进程间通信来共享 Web 文件（网页等）和多媒体。

进程间通信可以加速数据处理，例如维基百科使用通过进程间通信进行交流的多服务器来满足用户的请求。

进程间通信还可以实现模块化和私有权分离。

不过，与直接共享内存地址空间的多线程技术相比，进程间通信需要采用某种形式的内核开销，导致降低性能。

最后，几乎大部分 IPC 都不是程序设计的自然扩展，往往戏剧性增加了程序复杂度。

22.5 Thread

线程（thread）是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。

一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

在 Unix System V 及 SunOS 中，线程也被称为轻量进程（lightweight processes），但是实际上轻量进程更多指内核线程（kernel thread），而把用户线程（user thread）称为线程。

作为独立调度和分派的基本单位，线程可以操作系统内核调度的内核线程（例如 Win32 线程），由用户进程自行调度的用户线程（例如 Linux 平台的 POSIX Thread），或者由内核与用户进程（例如 Windows 7 的线程）等，进行混合调度。

同一进程中的多条线程将共享该进程中的全部系统资源，例如虚拟地址空间，文件描述符和信号处理等，但是同一进程中的多个线程有各自的调用栈（call stack），自己的寄存器环境（register context），自己的线程本地存储（thread-local storage）。

一个进程可以有很多线程，每条线程并行执行不同的任务。

在多核或多 CPU，或支持 Hyper-threading 的 CPU 上使用多线程程序设计的好处是显而易见，即提高了程序的执行吞吐率。

在单 CPU 单核的计算机上，使用多线程技术，也可以把进程中负责 IO 处理、人机交互而常备阻塞的部分与密集计算的部分分开来执行，编写专门的 workhorse 线程执行密集计算，从而提高了程序的执行效率。

与进程的状态不同，线程有四种基本状态，分别为：

- 产生（spawn）
- 中断（block）
- 非中断（unblock）
- 结束（finish）

22.5.1 UNIX International

SUN Solaris 操作系统使用的线程叫做 UNIX International 线程，支持内核线程、轻权进程和用户线程。

一个进程可有大量用户线程，大量用户线程复用少量的轻权进程，轻权进程与内核线程一一对应。用户级线程在调用核心服务时（如文件读写），需要“捆绑（bound）”在一个 lwp 上。

- 永久捆绑（一个 LWP 固定被一个用户级线程占用，该 LWP 移到 LWP 池之外）；
- 临时捆绑（从 LWP 池中临时分配一个未被占用的 LWP）

在调用系统服务时，如果所有 LWP 已被其他用户级线程所占用（捆绑），则该线程阻塞直到有可用的 LWP。如果 LWP 执行系统线程时阻塞（如 read() 调用），则当前捆绑在 LWP 上的用户级线程也阻塞。

UNIX International 线程的头文件是 <thread.h>。

- 创建用户级线程

```
int thr_create(void * stack_base, size_t stack_size, void
    *(*start_routine, void *), void * arg, long flags, thread_t * new_thr);
```

其中 flags 包括：THR_BOUND（永久捆绑）, THR_NEW_LWP（创建新 LWP 放入 LWP 池），若两者同时指定则创建两个新 LWP，一个永久捆绑而另一个放入 LWP 池。

- 等待用户级线程

```
int thr_join(thread_t wait_for, thread_t *dead, void **status);
```

- 挂起用户级线程

```
int thr_suspend(thread_t thr);
```

- 继续用户级线程

```
int thr_continue(thread_t thr);
```

- 退出用户级线程

```
void thr_exit(void *status);
```

- 返回当前用户级线程的线程标识符

```
thread_t thr_self( void );
```

22.5.2 POSIX Thread

POSIX 线程（POSIX threads），简称 Pthreads，是线程的 POSIX 标准。该标准定义了创建和操纵线程的一整套 API。

在类 Unix 操作系统（Unix、Linux、Mac OS X 等）中，都使用 Pthreads 作为操作系统的线程，Windows 操作系统也有其移植版 pthreads-win32。

Pthreads 定义了一套 C 语言的类型、函数与常量，它以 pthread.h 头文件和一个线程库实现。

Pthreads API 中大致共有 100 个函数调用，全都以"pthread_" 开头，并可以分为四类：

- 线程管理，例如创建线程，等待 (join) 线程，查询线程状态等。
- Mutex：创建、摧毁、锁定、解锁、设置属性等操作
- 条件变量（Condition Variable）：创建、摧毁、等待、通知、设置与查询属性等操作
- 使用了读写锁的线程间的同步管理

POSIX 的 Semaphore API 可以和 Pthreads 协同工作，但这并不是 Pthreads 的标准，因而这部分 API 是以"sem_" 打头，而非"pthread_"。

- 创建用户级线程

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void
    *(*start_routine)(void *), void *arg);
```

- 等待用户级线程

```
int pthread_join(pthread_t thread, void ** retval);
```

- 退出用户级线程

```
void pthread_exit(void *retval);
```

- 返回当前用户级线程的线程标识符

```
pthread_t pthread_self(void);
```

- 用户级线程的取消

```
int pthread_cancel(pthread_t thread);
```

22.5.3 Win32 Thread

Win32 线程是 Windows API 的一部分，其上下文包括寄存器、核心栈、线程环境块和用户栈。

Win32 线程的头文件是 <Windows.h>，仅适用于 Windows 操作系统。

- 创建用户级线程

```
HANDLE WINAPI CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T
    dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter,
    DWORD dwCreationFlags, LPDWORD lpThreadId);
```

- 结束本线程

```
VOID WINAPI ExitThread(DWORD dwExitCode);
```

- 挂起指定的线程

```
DWORD WINAPI SuspendThread( HANDLE hThread );
```

- 恢复指定线程运行

```
DWORD WINAPI ResumeThread(HANDLE hThread);
```

- 等待线程运行完毕

```
DWORD WINAPI WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

- 返回当前线程的线程标识符

```
DWORD WINAPI GetCurrentThreadId(void);
```

- 返回当前线程的线程句柄

```
HANDLE WINAPI GetCurrentThread(void);
```

22.5.4 C++11 Thread

2011 年 8 月 12 日，国际标准化组织（ISO）发布了第三个 C++ 标准，即 ISO/IEC 14882:2011，简称 ISO C++ 11 标准。该标准第一次把线程的概念引入 C++ 标准库。Windows 平台运行的 VS2012 和 Linux 平台运行的 g++4.7，都完美支持 C++11 线程。

C++ 11 线程的头文件是 <thread>

- 创建线程

```
std::thread::thread(Function&& f, Args&&... args);
```

- 等待线程结束

```
std::thread::join();
```

- 脱离线程控制

```
std::thread::detach();
```

- 交换线程

```
std::thread::swap(thread& other);
```

22.5.5 C11 Thread

2011 年 12 月 8 日，国际标准化组织（ISO）发布了第三个 C 语言标准，即 ISO 9899:2011，简称 ISO C 11 标准。该标准第一次把线程的概念引入 C 语言标准库。

C11 线程仅仅是个“建议标准”，也就是说 100% 遵守 C11 标准的 C 编译器是可以不支持 C11 线程的。根据 C11 标准的规定，只要编译器预定义了 __STDC_NO_THREADS__ 宏，就可以没有 <threads.h> 头文件，自然也就也没有下列函数。

C11 线程的头文件是 <threads.h>

- 创建线程

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

- 结束本线程

```
_Noreturn void thrd_exit( int res );
```

- 等待线程运行完毕

```
int thrd_join(thrd_t thr, int *res);
```

- 返回当前线程的线程标识符

```
thrd_t thrd_current();
```

22.5.6 Java Thread

- 最简单的情况是，Thread/Runnable 的 run() 方法运行完毕，自行终止。
- 对于更复杂的情况，比如有循环，则可以增加终止标记变量和任务终止的检查点。
- 最常见的情况，也是为了解决阻塞不能执行检查点的问题，用中断来结束线程，但中断只是请求，并不能完全保证线程被终止，需要执行线程协同处理。
- IO 阻塞和等锁情况下需要通过特殊方式进行处理。
- 使用 Future 类的 cancel() 方法调用。
- 调用线程池执行器的 shutdown() 和 shutdownNow() 方法。
- 守护线程会在非守护线程都结束时自动终止。
- Thread 有 stop() 方法，但已不推荐使用。

22.5.7 Multi Thread

多线程（multithreading）是指从软件或者硬件上实现多个线程并发执行的技术。

具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能，具有多线程能力的系统包括对称多处理机、多核心处理器以及芯片级多处理（Chip-level multithreading）或同时多线程（Simultaneous multithreading）处理器。

程序代码中存在的数据及控制依赖关系使得单线程中所能发掘的指令并行潜力是有限的，因此为了发掘有限的指令级并行潜力而一味强化乱序执行和分支预测，以至于处理器复杂度和功耗急剧上升，有时候是得不偿失的。

现代微处理器多采用硬件多线程技术来发掘线程之间的线程级并行潜力，这样就允许在接口转换的专业领域运算能力大幅提升。

- 既使这样做对于提升单一程序或是线程的性能相当困难，但是目前多数的系统都是使用多任务的方式作业。
- 能够明显的提升整体系统运算能力，总体吞吐量获得提升。

即便处理器只能运行一个线程，操作系统也可以通过快速的在不同线程之间进行切换，由于时间间隔很小，来给用户造成一种多个线程同时运行的假象。这样的程序运行机制被称为软件多线程。例如，微软的 Windows 作业系统和 Linux 就是在各个不同的执行绪间来回切换，被称为单用户多任务作业系统，但是 DOS 这类文字接口作业系统在一个时间只能处理一项工作，因此被视为单用户单任务系统。

除此之外，许多系统及处理器也支持硬件多线程技术。例如，对称多处理机（SMP）系统具有多个处理器，所以具有真正的同时执行多个线程的能力，CMP 技术通过在一块芯片上集成多个核心（Core）也具有真正的多线程能力。

CMT 技术则稍有不同，有的是依靠硬件执行线程切换来获得多线程能力，操作系统不再负责线程切换，因而这部分开销可以减少甚至消除，这方面典型的例子是 Sun 的 UltraSPARC T1，它同时综合了 CMP 和 CMT，微软的 Windows 2000 以后的操作系统皆支持多线程与超线程技术。

对于现在的两种提升运算能力的主要技术多进程和多线程，当共享硬件资源（例如缓存或是 TLB）时多线程会造成干预，而且单线程的运行时间可能不会因为多线程而变短，不过硬件侦测技

术有可能改变这一状况。

多线程的硬件支持会牵涉到软件支持，如此程序与操作系统就需要比多过程化更大幅度的修改。

在粗粒度交替多线程系统中，一个线程持续运行直到该线程被一个事件挡住而制造出长时间的延迟 (可能是内存 load/store 操作，或者程序分支操作)。该延迟通常是因缓存失败而从核心外的内存读写，而这动作会使用到几百个 CPU 周期才能将数据回传。与其要等待延迟的时间，线程化处理器会切换运行到另一个已就绪的线程。只要当之前线程中的数据送达后，上一个线程就会变成已就绪的线程。这种方法来自各个线程的指令交替执行，可以有效的掩盖内存访问时延，填补流水线空洞。

举例来说：

1. 周期 i： 接收线程 A 的指令 j
2. 周期 i+1：接收线程 A 的指令 j+1
3. 周期 i+2：接收线程 A 的指令 j+2，而这指令缓存失败
4. 周期 i+3：线程调度器介入，切换到线程 B
5. 周期 i+4：接收线程 B 的指令 k
6. 周期 i+5：接收线程 B 的指令 k+1

在概念上，粗粒度交替多线程系统与实时操作系统中使用的合作式多任务类似，在该任务需要为一个事件等待一段时间的时候会主动放弃运行时段。

粗粒度交替多线程系统的硬件支持的目标是允许在挡住的线程与已就绪的线程中快速切换。为了要达成这个目标，硬件成本将复制程序看得见的寄存器与一些处理器控制寄存器（像是程序计算器）。从一个线程切换到另一个线程对硬件来讲意谓著从一个寄存器复制到另一个。

- 线程切换能够在一个 CPU 周期内完成 (实际上可以没有开销，上个周期在运行线程 A，下个周期就已在运行线程 B)。
- 每个线程是独自运行的，没有其他线程与目前共享硬件资源。对操作系统来说，通常每个虚拟线程都被视做一个处理器。这样就不需要很大的软件变更（像是特别写支持多线程的操作系统）。

为了要在各个现行中的线程有效率的切换，每个现行中的线程需要有自己的暂存设置（register set）。像是为了能在两个线程中快速切换，硬件的寄存器需要两次例示（instantiated）。

许多微控制器与嵌入式处理器有多重的寄存器列，就能够在中断时快速环境切换，这样架构可以视为程序的线程与中断线程之间的块状多线程处理。

细粒度交替式多线程系统提供了更高性能的多线程做法，所有 CPU 周期轮流切换至不同的线程，来自各线程的指令按顺序交替执行，整个执行过程很像桶形处理器 (Barrel Processor)。

1. 周期 i： 接收线程 A 的一个指令
2. 周期 i+1：接收线程 B 的一个指令
3. 周期 i+2：接收线程 C 的一个指令

细粒度交替式多线程的效果是会将所有从运行管线中的数据从属（data dependency）关系移除掉。因为每个线程是相对独立，管线中的一个指令层次结构需要从已跑完管线中的较旧指令代入输出的机会就相对的变小了。

在概念上，细粒度交替式多线程与操作系统的核心先占多任务（pre-exemptive multitasking）相似。

除了讨论块状多线程的硬件成本，交错式多线程也因每层管线需要追踪运行中指令的线程代码而增加硬件成本。而且，当越来越多的线程同时在管线中运行，像是缓存与 TLB 等共享资源也要加大来避免不同线程之间的冲突。

目前，最先进的多线程技术是应用在超标量处理器上的同步多线程，超标量处理器内在每个 CPU 周期中，单独一个线程会发布众多的指令。应用同步多线程 (SMT) 之后，超标量处理器就可以在每个 CPU 周期中，从多个线程中发布指令。辨识到任何一个单一线程拥有有限数量的指令平行处理，这种类型的多线程是试着利用并行的方式跨越多线程，以减少浪费与闲置的资源。举例来说：

1. 周期 i：线程 A 的 j 指令与 j+1 指令，还有 B 线程的指令 k 同时发布
2. 周期 i+1：线程 A 的 j+2 指令、线程 B 的 k+1 指令，与线程 C 的 m 指令同时发布
3. 周期 i+2：线程 A 的 j+3 指令，与线程 C 的 m+1 与 m+2 指令同时发布

交错式多线程如果不计硬件成本，SMT 在每个管线层次结构的追踪线程指令会有多余的花费，而且缓存与 TLB 等共享的资源可能会因为多出来的线程而变得更大。

在大多数研究领域内是要求线程调度器要能够快速选择其中一个已就绪线程去运行，而不是一个一个运行而降低效率。所以要让调度器去分辨线程的优先级是很重要的。而线程调度器可能是以硬件、软件，或是软硬件并存的形式存在。

另一个研究领域则是要研究何种事件（缓存失败、内部运行续连系、使用 DMA 等）会造成线程切换。

如果多线程的方案会复制所有软件可见的状态，包括特许的控制登录、TLB 等，那就能够让虚拟机去创造各式线程，这样就允许在相同的处理器中每个线程跑各自的操作系统。换句话说，如果只有存储了用户模式的状态，就能够让相同的裸晶大小的芯片在一段时间内处理更多的线程。

22.5.8 Thread Pool

线程池（Thread Pool）是一种成熟的线程使用模式，实现有领导者与跟随者模式和半同步半异步模式。

线程池的伸缩性对性能有较大的影响。

- 创建太多线程，将会浪费一定的资源，有些线程未被充分使用。
- 销毁太多线程，将导致之后浪费时间再次创建它们。
- 创建线程太慢，将会导致长时间的等待，性能变差。
- 销毁线程太慢，导致其它线程资源饥饿。

22.6 Interrupt

在计算机科学中，中断（Interrupt）是指处理器接收到来自硬件或软件的信号，提示发生了某个事件，应该被注意，这种情况就称为中断。

通常，在接收到来自外围硬件（相对于中央处理器和内存）的异步信号，或来自软件的同步信号之后，处理器将会进行相应的硬件/软件处理。发出这样的信号称为进行中断请求（interrupt request，IRQ）。

- 硬件中断导致处理器通过一个运行信息切换（context switch）来保存执行状态（以程序计数器和程序状态字等寄存器信息为主）。
    1. 可屏蔽中断（maskable interrupt）。硬件中断的一类，可通过在中断屏蔽寄存器中设定位掩码来关闭。
    2. 非可屏蔽中断（non-maskable interrupt，NMI）。硬件中断的一类，无法通过在中断屏蔽寄存器中设定位掩码来关闭。典型例子是时钟中断（一个硬件时钟以恒定频率—如 50Hz—发出的中断）。

3. 处理器间中断（interprocessor interrupt）。一种特殊的硬件中断。由处理器发出，被其它处理器接收。仅见于多处理器系统，以便于处理器间通信或同步。
4. 伪中断（spurious interrupt）。一类不希望被产生的硬件中断。发生的原因有很多种，如中断线路上电气信号异常，或是中断请求设备本身有问题。

- 软件中断则通常作为 CPU 指令集中的一个指令，以可编程的方式直接指示这种运行信息切换，并将处理导向一段中断处理代码。
  1. 软件中断。是一条 CPU 指令，用以自陷一个中断。由于软中断指令通常要运行一个切换 CPU 至内核态（Kernel Mode/Ring 0）的子例程，它常被用作实现系统调用（System call）。

处理器通常含有一个内部中断屏蔽位，并允许通过软件来设定。一旦被设定，所有外部中断都将被系统忽略。这个屏蔽位的访问速度显然快于中断控制器上的中断屏蔽寄存器，因此可提供更快速地中断屏蔽控制。

如果一个中断使得机器处于一种确定状态，则称为精确中断（precise interrupt）。精确中断须保证：

- 程序计数器的值被保存在已知位置。
- 程序计数器所指向的指令之前的所有指令已被执行完毕。
- 程序计数器所指向的指令之后的所有指令不可被执行。
  如果中断信号到来后而转入处理前发生了任何针对寄存器/内存的更改，都必须予以还原。
- 程序计数器所指向的指令地执行状态已知。

如果无法满足以上条件，此中断被称作非精确中断（imprecise interrupt）。

中断在计算机多任务处理（尤其是实时系统）中特别有用，这样的系统（包括运行于其上的操作系统）也被称为“中断驱动的”（interrupt-driven）。

中断的典型应用包括系统时钟、磁盘输入输出操作、断电信号以及软件自陷等。‘

- 系统时钟通过一个计数器（多基于某种振动频率）定期向 CPU 发出中断，CPU 通过专门的时钟中断处理程序来保持计时。现代操作系统对系统时钟的另一个主要应用是为进程切换提供时机。一旦时钟中断发生，程序计数器会被自动压栈，而此时操作系统就有机会将程序状态及内存映像转存至别处，并调用进程调度程序来选择下一个进程，并将其进程状态，包括程序计数器，导入寄存器。这样下一个程序就可以运行。应注意进程调度程序的调度时机不止于时钟中断。
- 磁盘中断标识某个磁盘设备完成了数据的发送 / 接收。磁盘中断发生后，等待这个中断的进程可以（但未必，这取决于进程调度程序当时的判断）继续执行。
- 断电中断指示计算机能源即将丧失，计算机可以相应中断程序作有序的关机处理。

INTEL 公司于 20 世纪 70 年代末推出的 16 位处理器 8086A 在中断处理上引入的特色为后续 INTEL 处理器所共有。

- pin（引脚）
  8086A 提供两个中断引脚：第 17 引脚 NMI 和第 18 引脚 INTR，其中的前者用于接收非可屏蔽中断，后者则接收可屏蔽中断。
  通常情况下，INTR 引脚与中断控制器（如 8259A）相连，后者再分别与各设备的中断请求引脚连接。
  除了 INTR 外，8086A 还将自身的 16 位地址总线（配合 M/IO 引脚并通过译码器）及 8 位数据总线与 8259A 连接，并将 INTA 引脚与 8259A 的同名引脚相连。
- interrupt vector table（中断矢量表）
  在存储器地址空间中，规定最低的 1K 空间（即 00000H 到 003FFH）为中断矢量表。

全表共含 256 个中断矢量，每个矢量的长度为 4 字节，包含中断处理程序的起始地址。共有从 0 到 255 共 256 个中断类型码，每个中断类型码对应的中断矢量所在地址为该类型码乘以 4。举例来说，如果中断类型码为 1，则对应中断矢量所在地址为 00004H；如果中断类型码为 33，则对应中断矢量所在地址为 00084H。这样，如果已知一个中断类型码，则需要通过两次地址转换（中断类型码到中断矢量表地址；中断矢量表地址到中断处理程序地址）才能到达中断处理程序。

另外，应该注意每一个中断矢量所包含的地址是以低二字节存储偏移量，高二字节存储段地址的形式存储目标地址值的。

在全部 256 个中断中，前 32 个（0-31）为硬件系统所预留。后 224 个可由用户设定。在初始化 8259A 时，可设定其上各中断引脚（共 8 条）对应的中断类型码。同时，将对应此中断之处理程序的起始地址保存在该中断类型码乘 4 的地址位中，作为中断矢量。

在 INTEL 后续的 32 位 CPU 中，使用中断描述符表来代替中断矢量表。中断描述符表的起始地址由中断描述符表寄存器（IDTR）来定位，因此不再限于底部 1K 位置。另一方面，中断描述符表的每一个项目——称作门描述符——除了含有中断处理程序地址信息外，还包括许多属性/类型位。

门描述符分为三类：任务门、中断门和自陷门，CPU 对不同的门有不同的调用（处理）方式。

在实际运行中，一旦设备通过某引脚 N 向 8259A 发出中断指令，后者便向 8086A 的 INTR 引脚发送中断信号。8086A 通过 INTA 引脚通知 8259A 中断有效（这个过程实际上还包括对此 8259A 的选址），后者即通过地址总线将对应引脚 N 的中断类型码（已预先存好，见上节）发送给 CPU。CPU 得到中断类型码后，先进行现场保护，主要包括：

1. 状态寄存器 FLAGS 压栈（同时堆栈寄存器 SP-2）；
2. 关闭中断（将 FLAGS 寄存器的 IF 位置零）；
3. 将当前代码段寄存器 CS 和程序计数器 IP 压栈（同时堆栈寄存器 SP-4）。

现场保护完成后，CPU 开始按照前述的两步骤翻译中断程序入口地址。在得到中断处理程序地址之后但调用中断处理程序之前，CPU 会再检查一下 NMI 引脚是否有信号，以防在刚才的处理过程中忽略了可能的 NMI 中断。NMI 的优先级始终高于 INTR。

中断处理程序虽然是由程序员编写，但须循一定规范。作为例程，中断处理程序应该先将各寄存器信息（除了 IP 和 CS，此二寄存器现已指向当前中断程序）压入堆栈予以保存，这样才能在中断处理程序内部使用这些寄存器。在程序结束时，应该按与压栈保护时相反的顺序弹出各寄存器的值。中断程序的最后一句始终是 IRET 指令，这条指令将栈顶 6 个字节分别弹出并存入 IP、CS 和 FLAGS 寄存器，完成了现场的还原。

当然，如果是操作系统的中断处理程序，则未必——通常不会——还原中断前的状态。这样的中断处理程序通常会在调用完寄存器保存例程后，调用进程调度程序（多由高级语言编写），并决定下一个运行的进程。随后将此进程的寄存器信息（上次中断时保存下来的）存入寄存器并返回。在中断程序结束之后，主程序也发生了改变。

一些与中断控制相关的指令包括：

1. `CLI` 关闭中断（实为将 FLAGS 的 IF 位置 0）。
2. `STI` 开启中断（实为将 FLAGS 的 IF 位置 1）。
3. `INT n` 调用中断子程序，n 为中断类型码。DOS 系统有一个系统调用 API 即为 `INT 21H`。
4. `INTO` 先判别 FLAGS 的 OF 位是否为 1，如是则直接调用类型为 4 的中断子程序，用以处理溢出中断。
5. `IRET` 将栈顶 6 个字节分别弹出并存入 IP、CS 和 FLAGS 寄存器，用以返回中断现场。

22.6.1 Level-triggered

在依状态触发的中断系统中，一个等待响应的中断会在中断请求线路上以特定的电平标识，如高电平（1）或低电平（0）。当一个设备希望发送中断信号时，它驱动中断请求线路至相应的电平，并在 CPU 发出强制停止命令或处理所请求的中断事件之前始终保持。

一般而言，处理器在总线周期的特定时点响应中断的输出/ 输入。如果在某次采样时刻中断尚未被激活，则在下一次采样前，处理器都不会认为有中断发生。可以应用这个特性，避免响应在噪音较高的线路上出现的伪中断。

中断设备可被设计成与其他设备共享一条状态触发中断线路。中断线路应该包含一个特定的升 / 降压电阻，用于在无中断请求时为线路电平复位。中断设备在请求中断时会保持中断线路为有效电平，而没有请求中断时则令该线路置空。只要有一个或以上的设备发出中断信号，线路都会处于有效的电平。

由于可共享线路的便利，一些应用倾向于使用该类中断。当 CPU 检测到中断线路被断言后，就会逐一检查各共享设备，直至发现请求设备并处理之。当处理完毕后，继续检查中断线路，倘中断线路仍为有效电平则重复之前的步骤。在检查中断设备的顺序上也可做一定规划，比如优先检查那些频繁请求中断的设备，以加快中断处理，改善系统性能。

此类中断模式也有严重问题。只要还有任何设备的中断请求未被处理，线路就会一直保持有效电平状态，而这将导致 CPU 没有机会去探查其他设备所发生的状态变化。推迟服务低优先级设备也不可行，因为这会防止对高优先级设备的探查。倘若在线路上有一个设备持续发送请求而 CPU 不知道怎样对其进行服务，则这个设备就会持久并排他地占有中断线路。

早期的 PCI（外设互连标准）标准出于上述效率层面的理由规定其周边须使用状态触发中断。

22.6.2 Edge-triggered

在依边沿触发的中断系统中，中断设备通过向中断线路发送一个脉冲来表示其中断请求。脉冲可以为上升沿或下降沿。在发送完脉冲后设备立即释放中断线路。如果这个脉冲太短，以至于 I/O 轮询不足以确保知悉其存在，则有必要使用专门的硬件设备来辅助对边沿触发的探查。

中断设备可被设计成与其他设备共享一条边沿触发中断线路。中断线路应该包含一个特定的上拉 / 下拉电阻，用于在无中断请求时为线路电平复位。设备通过发送一个脉冲作为其中断信号。如果多个设备在近乎相同的时间内发送脉冲，则会在线路上合并成一个信号。为防止中断丢失，CPU 必须在一个脉冲之后的下一个边沿（如果脉冲为上升沿则其下一个边沿就是下降沿）立即触发。收到中断请求后 CPU 立即查询各中断设备以定位中断源。

边沿触发中断不会遭受状态触发中断在共享中断引脚时所遇到的问题。低优先级设备的服务可被任意推迟，而高优先级设备的中断请求仍会被 CPU 收到。一个即便是频繁发生的伪中断也不会影响正常设备的中断请求。但是，边沿触发中断容易丢失，特别是当中断被有意屏蔽时。在不引入锁存器的情况下，在屏蔽时段发送的中断信号不可能被恢复。在早期的计算机系统中因为中断丢失而导致处理不能继续的情况时有发生。现代中断硬件多包含有一个或一组中断状态锁存器，用以暂存一逝而过的中断请求。在对边沿触发中断硬件进行编程时，应检查这些中断状态寄存器以确保请求事件不会丢失。

已经过时的 ISA（工业标准架构）标准使用边沿触发中断，但不规定其实现必须能够共享线路。

22.6.3 Hybrid

一些系统使用状态触发与边沿触发兼顾的混合中断模式。其硬件不但探测脉冲，也验证中断信号是否保持一段时间。

非可屏蔽中断多使用混合模式。由于非可屏蔽中断多与重要的系统异常事件相关，十分有必要确保对其中断信号的捕捉快速而正确。这种两步骤探查方式能够有效减轻错误中断或遗失中断给系统带来的影响。

22.6.4 Message-signaled

消息信号式中断并不直接通过对特定物理线路进行断言 / 发送脉冲来通知一个中断。这类中断设备通过在某种通讯媒介（一般是计算机总线）上发送一个有逻辑含义的消息（一串 / 排比特码）来实现中断请求。中断消息可以是通讯总线协议中专门为中断预留的类型，也可以是一个现有的类型，如内存写操作。

消息信号式中断在行为上与边沿触发中断类似，因为它们都是发送一个瞬间的信号。中断处理软件的对此类中断的处理方式也类似于边沿触发中断：如果两个消息相同，则可以合并。消息信号中断矢量（中断处理程序的地址）也可以共享，就如同物理线路可以被共享一般。

由于中断消息的识别基于特定的比特码序列而不是物理线路上的单个信号，可以有效地通过设定不同的中断比特码来划分和处理不同类型的中断。另外，使用串行或并行总线都可以传递中断消息。

由于无论状态触发还是边沿触发都在使用共享线路时存在线路竞争问题，而物理线路数本身也是稀缺资源，不可能被各中断源分别独占，所以消息信号中断是一个解决此问题的较好替代方案。消息信号中断的本质差别在于其中断请求运行在单纯的物理线路之上，具有特定的逻辑含义。这种区别好比计算机网络体系中第一层（物理层）和第二层（链路层）的差别。使用具有逻辑含义的中断请求，可以把诸请求区分开来，形成多条虚通路，而运行于一条物理总线之上。

PCI Express 串行总线标准即使用消息信号模式的中断。

22.6.5 Doorbell

22.7 Polling

尽管中断可以提高计算机处理性能，但是过于密集的中断请求/响应反而会影响系统性能，这类情形被称作中断风暴（interrupt storm），与中断处理相对的是轮询（Polling）。

轮询（Polling）是一种 CPU 决策如何提供周边设备服务的方式，又称“程控输出入”（Programmed I/O），其过程是由 CPU 定时发出询问，依序询问每一个周边设备是否需要其服务，有即给予服务，服务结束后再问下一个周边，接着不断周而复始，问题是虽然轮询法实现容易，但是效率偏低。

中断是用以提高计算机工作效率、增强计算机功能的一项重要技术。最初引入硬件中断，只是出于性能上的考量。如果计算机系统没有中断，则处理器与外部设备通信时，它必须在向该设备发出指令后进行忙等待 (Busy waiting)，反复轮询该设备是否完成了动作并返回结果，从而造成了大量处理器周期被浪费。

引入中断以后，当处理器发出设备请求后就可以立即返回以处理其他任务，而当设备完成动作后，发送中断信号给处理器，后者就可以再回过头获取处理结果。这样，在设备进行处理的周期内，处理器可以执行其他一些有意义的工作，而只付出一些很小的切换所引发的时间代价。后来被用于

CPU 外部与内部紧急事件的处理、机器故障的处理、时间控制等多个方面，并产生通过软件方式进入中断处理（软中断）的概念。

在硬件实现上，中断可以是一个包含控制线路的独立系统，也可以被集成进存储器子系统中。对于前者，在 IBM 个人机上，广泛使用可编程中断控制器（Programmable Interrupt Controller，PIC）来负责中断响应和处理。PIC 被连接在若干中断请求设备和处理器的中断引脚之间，从而实现对处理器中断请求线路（多为一针或两针）的复用。

作为另一种中断实现的形式，即存储器子系统实现方式，可以将中断端口映射到存储器的地址空间，这样对特定存储器地址的访问实际上是中断请求。

22.8 Signal

在计算机科学中，信号（Signals）是 Unix、类 Unix 以及其他 POSIX 兼容的操作系统中进程间通讯的一种有限制的方式。

信号是一种异步的通知机制，用来提醒进程一个事件已经发生。当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断。如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数。

在一个运行的程序的控制终端键入特定的组合键可以向它发送某些信号：

- Ctrl-C 发送 INT 信号（SIGINT）：默认情况下，这会导致进程终止。
- Ctrl-Z 发送 TSTP 信号（SIGTSTP）：默认情况下，这会导致进程挂起。
- Ctrl-\发送 QUIT 信号（SIGQUIT）：默认情况下，这会导致进程终止并且将内存中的信息转储到硬盘（核心转储）。
- 这些组合键可以通过 stty 命令来修改。

kill() 系统调用会在权限允许的情况下向进程发送特定的信号，类似地 kill 命令允许用户向进程发送信号，raise(3) 库函数可以将特定信号发送给当前进程。

另外，除数为零、段错误等异常也会产生信号（这里分别是 SIGFPE 和 SIGSEGV，默认都会导致进程终止和核心转储）。

内核可以向进程发送信号以告知它一个事件发生了。例如，当进程将数据写入一个已经被关闭的渠道是将会收到 SIGPIPE 信号，默认情况下会使进程关闭。

信号处理函数可以通过 signal() 系统调用来设置。如果没有为一个信号设置对应的处理函数，就会使用默认的处理函数，否则信号就被进程截获并调用相应的处理函数。

在没有处理函数的情况下，程序可以指定两种行为：忽略这个信号（SIG_IGN）或者用默认的处理函数（SIG_DFL），但是有两个信号是无法被截获并处理的：SIGKILL 和 SIGSTOP。

竞态条件的存在和信号本身的异步特性使信号的处理有弱点，所以在处理一个信号的过程中，进程可能收到另一个信号（甚至是相同的信号）。

sigprocmask() 系统调用可以用来阻塞和恢复信号的传递，信号可以造成进程中系统调用的中断，并在信号处理完后重新开始未完成的系统调用。

在实践中，信号处理函数应该没有任何不想要的副作用，比如 errno 的改变、信号掩码的改变、信号处理方法的改变，以及其他全局进程性质的改变。在信号处理函数内使用不可重入函数（例如 malloc 和 printf）也是不安全的。

进程的运行也可能导致硬件异常，例如，在类 Unix 系统中将一个数除以零，或者出现 TLB 不命中都会自动运行内核的异常处理程序。

对于某些异常如页缺失，内核有足够的信息来处理完并恢复进程的运行。但是对于另外一些异

常，内核不能处理而只能通过发送信号把异常交给进程自己处理。例如，在 x86 架构的 CPU 上，如果一个进程尝试将一个数除以零，将会产生 divide error 异常，并使内核向出错的进程发送 SIGFPE 信号。相似地，如果一个进程尝试访问虚拟地址空间以外的内存，内核将向进程发送 SIGSEGV 信号。异常与信号的具体对应关系在不同的 CPU 架构上是不同的。

22.9 Queue

22.9.1 Message Passing

在计算机科学中，消息传递（Message passing）是一种通讯的形式，其主要的数学模型为参与者模式、π-calculus，并且应用在并发计算、并行计算、面向对象程序设计与进程间通讯中，进程或对象以发送及接收消息的方式来达成同步。

不同于传统程序设计通过名字直接调用（invoking）一个进程、子例程或者函数，消息传递直接发送消息给一个进程，依赖进程或基础框架来调用实际执行的代码，并且可以分为同步方式与异步方式。

消息传递是一种通讯范型，在这种模型中由一个传信者将消息（messages）送给一个或多个收信者。

根据操作系统与编程语言的支持，消息的形式有所不同，常见的有方法（method）、信号（signals）与数据包（data packets）。

实际应用的消息传递系统有开放网络运算远程过程调用（ONC RPC）、CORBA、Java RMI、Distributed COM、SOAP。

22.9.2 Messager Queue

在计算机科学中，消息队列（Message queue）是一种进程间通信或同一进程的不同线程间的通信方式，软件的队列用来处理一系列的输入，通常是来自使用者。

消息队列提供了异步的通信协议，每一个队列中的记录包含详细说明的信息，包含发生的时间，输入装置的种类，以及特定的输入参数，也就是说消息的发送者和接收者不需要同时与消息队列交互，这样消息就会保存在队列中，直到接收者取回它。

一个 WIMP 环境（例如 Microsoft Windows）可以通过优先的某些形式（通常是事件的时间或是重要性的顺序）来存储使用者产生的事件到一个事件队列中，然后系统把每个事件从事件队列中传递给目标的应用程序。

目前，有很多消息队列有很多开源的实现，包括 JBoss Messaging、JORAM、Apache ActiveMQ、Sun Open Message Queue、Apache Qpid 和 HTTPSQS 等。

实际上，消息队列常常保存在链表结构中，因此消息队列有大小限制，只有拥有权限的进程可以向消息队列中写入或读取消息。

消息队列本身是异步的，它允许接收者在消息发送很长时间后再取回消息，这导致了和大多数通信协议的不同。例如，HTTP 协议就是同步的，客户端在发出请求后必须等待服务器回应。

不过，在很多情况下我们需要异步[3]的通信协议。例如，一个进程通知另一个进程发生了一个事件，但是不需要等待回应。

消息队列的异步特点也造成了一个缺点，就是接收者必须轮询消息队列，才能收到最近的消息。

[3]消息队列非常独特，其本质就是一个消息的链表，因此两个进程不必同时存在，一个进程可以发送一个消息并退出，而该消息可以在数天后才被另一个进程获得。

- 和信号相比，消息队列能够传递更多的信息。
- 与管道相比，消息队列提供了有格式的数据。

22.9.3 Messager Service

Java 消息服务（Java Message Service，JMS）应用程序接口是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

Java 消息服务本身是一个与具体平台无关的 API，绝大多数 MOM 提供商都对 JMS 提供支持。

Java 消息服务的规范包括两种消息模式，点对点和发布者/订阅者，用户可以在他们的分布式软件中实现面向消息的操作，这些操作将具有不同面向消息中间件产品的可移植性。

Java 消息服务支持同步和异步的消息处理，在某些场景下，异步消息是必要的，而且在其他场景下异步消息可能比同步消息操作更加便利。

Java 消息服务支持面向事件的方法接收消息，事件驱动的程序设计现在被广泛认为是一种富有成效的程序设计范例，这样在应用系统开发时，Java 消息服务可以推迟选择面对消息中间件产品，也可以在不同的面对消息中间件切换。

JMS 由以下元素组成：

- JMS 提供者连接面向消息中间件的，JMS 接口的一个实现。提供者可以是 Java 平台的 JMS 实现，也可以是非 Java 平台的面向消息中间件的适配器。
- JMS 客户生产或消费消息的基于 Java 的应用程序或对象。
- JMS 生产者创建并发送消息的 JMS 客户。
- JMS 消费者接收消息的 JMS 客户。
- JMS 消息包括可以在 JMS 客户之间传递的数据的对象
- JMS 队列一个容纳那些被发送的等待阅读的消息的区域。队列暗示，这些消息将按照顺序发送。一旦一个消息被阅读，该消息将被从队列中移走。
- JMS 主题一种支持发送消息给多个订阅者的机制。

Java 消息服务应用程序结构支持两种模型：

- 点对点或队列模型
- 发布/订阅模型

在点对点或队列模型下，一个生产者向一个特定的队列发布消息，一个消费者从该队列中读取消息。这里，生产者知道消费者的队列，并直接将消息发送到消费者的队列，因此这种模式被概括为：

- 只有一个消费者将获得消息
- 生产者不需要在接收者消费该消息期间处于运行状态，接收者也同样不需要在消息发送时处于运行状态。
- 每一个成功处理的消息都由接收者签收

发布者/订阅者模型支持向一个特定的消息主题发布消息，0 或多个订阅者可能对接收来自特定消息主题的消息感兴趣。

在发布者/订阅者模型下，发布者和订阅者彼此不知道对方，因此这种模式好比是匿名公告板，可以被概括为：

- 多个消费者可以获得消息
- 在发布者和订阅者之间存在时间依赖性。发布者需要建立一个订阅（subscription），以便客户能够购订阅。订阅者必须保持持续的活动状态以接收消息，除非订阅者建立了持久的订阅。在那种情况下，在订阅者未连接时发布的消息将在订阅者重新连接时重新发布。

JMS 提供了将应用与提供数据的传输层相分离的方式，同一组 Java 类可以通过 JNDI 中关于提供者的信息，连接不同的 JMS 提供者。这一组类首先使用一个连接工厂以连接到队列或主题，然后发送或发布消息。在接收端，客户接收或订阅这些消息。

Java 消息服务的 API 在 javax.jms 包中提供，具体内容包括：

- ConnectionFactory 接口（连接工厂）
  用户用来创建到 JMS 提供者的连接的被管对象。JMS 客户通过可移植的接口访问连接，这样当下层的实现改变时，代码不需要进行修改。管理员在 JNDI 名字空间中配置连接工厂，这样，JMS 客户才能够查找到它们。根据消息类型的不同，用户将使用队列连接工厂，或者主题连接工厂。
- Connection 接口（连接）
  连接代表了应用程序和消息服务器之间的通信链路。在获得了连接工厂后，就可以创建一个与 JMS 提供者的连接。根据不同的连接类型，连接允许用户创建会话，以发送和接收队列和主题到目标。
- Destination 接口（目标）
  目标是一个包装了消息目标标识符的被管对象，消息目标是指消息发布和接收的地点，或者是队列，或者是主题。JMS 管理员创建这些对象，然后用户通过 JNDI 发现它们。和连接工厂一样，管理员可以创建两种类型的目标，点对点模型的队列，以及发布者 / 订阅者模型的主题。
- MessageConsumer 接口（消息消费者）
  由会话创建的对象，用于接收发送到目标的消息。消费者可以同步地（阻塞模式），或异步（非阻塞）接收队列和主题类型的消息。
- MessageProducer 接口（消息生产者）
  由会话创建的对象，用于发送消息到目标。用户可以创建某个目标的发送者，也可以创建一个通用的发送者，在发送消息时指定目标。
- Message 接口（消息）
  是在消费者和生产者之间传送的对象，也就是说从一个应用程序创送到另一个应用程序。一个消息有三个主要部分：
  1. 消息头（必须）：包含用于识别和为消息寻找路由的操作设置。
  2. 一组消息属性（可选）：包含额外的属性，支持其他提供者和用户的兼容。可以创建定制的字段和过滤器（消息选择器）。
  3. 一个消息体（可选）：允许用户创建五种类型的消息（文本消息，映射消息，字节消息，流消息和对象消息）。

  消息接口非常灵活，并提供了许多方式来定制消息的内容。
- Session 接口（会话）
  表示一个单线程的上下文，用于发送和接收消息。由于会话是单线程的，所以消息是连续的，就是说消息是按照发送的顺序一个一个接收的。会话的好处是它支持事务。如果用户选择了事务支持，会话上下文将保存一组消息，直到事务被提交才发送这些消息。在提交事务之前，用户可以使用回滚操作取消这些消息。一个会话允许用户创建消息生产者来发送消息，创建消息消费者来接收消息。

要使用 Java 消息服务，必须要有一个 JMS 提供者来管理会话和队列。现在既有开源的提供者也有专有的提供者。

如果计划在一个服务器集群中运行程序，需要检查提供者是否实现了对负载均衡和故障恢复的

支持。

22.10 Async

22.10.1 AJAX

AJAX（Asynchronous JavaScript and XML，异步的 JavaScript 与 XML 技术）综合了多项技术来支持浏览器端的网页开发。

与传统的 Web 应用允许用户端填写表单（form）并提交不同，当用户提交表单时就向 Web 服务器发送一个请求，服务器接收并处理传来的表单，然后送回一个新的网页，这个做法浪费了许多带宽。

在前后两个页面中的大部分 HTML 码往往是相同的，每次应用的沟通都需要向服务器发送请求，应用的回应时间依赖于服务器的回应时间，因此这也导致了用户界面的回应比本机应用慢得多。

AJAX 应用可以仅向服务器发送并取回必须的数据，并在客户端采用 JavaScript 处理来自服务器的回应，在服务器和浏览器之间交换的数据大量减少（大约只有原来的 5%），因此服务器回应更快了。同时，很多的处理工作可以在发出请求的客户端机器上完成，因此 Web 服务器的负荷也减少了。

类似于 DHTML 或 LAMP，AJAX 不是指一种单一的技术，而是有机地利用了一系列相关的技术。例如，虽然其名称包含 XML，但实际上数据格式可以由 JSON 代替以进一步减少数据量，形成所谓的 AJAJ。

客户端与服务器也可以不需要异步，例如一些基于 AJAX 的“派生/合成”式（derivative/composite）的技术也正在出现，如 AFLAX。

22.10.2 Method

异步方法调用或异步方法模式是（多线程）面向对象程序设计中用于异步调用对象的潜在的长期运行方法的一种设计模式。

异步方法调用等价于 Allan Vermeulen 提出的 IOU 模式，基于事件的异步模式是异步方法调用的一个变种，开销更大但能更好的表现软件组件对象。

.NET 框架和 Java 中的 java.util.concurrent.FutureTask 类中使用的基于事件的异步模式使用事件来解决同样的问题，而且异步事件处理函数库 libevent 通过一组 API 来让用户可以设定某些事件发生时所执行的函数，也就是说，libevent 可以用来取代网络服务器所使用的事件循环检查框架。

具体来说，libevent 支持 poll、select、/dev/pool、kqueue 和 epoll 等方式来判断 IO 事件，这样 libevent 就省略了对网络的处理，并且拥有不错的性能，因此被 memcached 和 tor 等软件作为网络底层的函数库。

大部分编程语言中对方法的调用是同步执行的。例如，在线程执行体内（即线程的调用函数中），方法的调用就是同步执行的。

如果方法需要很长的时间来完成，比方说从 Internet 加载数据的方法，调用者线程将被阻塞直到方法调用完成。如果不希望调用被阻塞，则可以通过创建新的 worker 线程并在 worker 线程中调用方法，在大多数编程环中上这样做可能需要很长的一段代码，尤其是需要小心处理线程过多的额外开销。

异步方法调用它通过使用一种立即返回的异步的变体并提供额外的方法来支持接受完成通知以及完成等待改进长期运行的（同步）方法。

活动对象（active object）设计模式通常使用异步方法调用，异步方法调用的一个替代方案是同步的方法调用和未来对象（future object）模式。

在 Web 浏览器的实现上可以采用异步方法调用，例如浏览器需要在 Web 页面中的图像加载完成之前将页面显示出来。

22.10.3 Pattern

在计算机科学中，参与者模式（Actor model）是一种并发运算上的模型。

“参与者”是一种程序上的抽象概念，被视为并发运算的基本单元：当一个参与者接收到一则消息，它可以做出一些决策、创建更多的参与者、发送更多的消息、决定要如何回答接下来的消息。

参与者模型推崇的哲学是“一切皆是参与者”，与面向对象编程的“一切皆是对象”类似，但是面向对象编程通常是顺序执行的，而参与者模型是并行执行的。

参与者是一个运算实体，回应接受到的消息，同时并行的：

- 发送有限数量的消息给其他参与者；
- 创建有限数量的新参与者；
- 指定接受到下一个消息时的行为。

以上操作不含有顺序执行的假设，因此可以并行进行。

发送者与已经发送的消息解耦是参与者模型的根本优势，这样就允许进行异步通信，同时满足消息传递的控制结构。

消息接收者是通过地址区分的，有时也被称作“邮件地址”，因此参与者只能和它拥有地址的参与者通信，它可以通过接受到的信息获取地址，或者获取它创建的参与者的地址。

参与者模型的特征是，参与者内部或之间进行并行计算，参与者可以动态创建，参与者地址包含在消息中，交互只有通过直接的异步消息通信，不限制消息到达的顺序。

22.10.4 Task

在 Server 程序中如果需要执行一下很耗时的操作，比如一个聊天服务器发送广播，Web 服务器中发送邮件。如果直接去执行这些函数就会阻塞当前进程，导致服务器响应变慢。

与旧版本（<=2.2）的 Apache 不同，nginx 不采用每客户机一线程的设计模型，而是充分使用异步逻辑来削减上下文调度开销，所以并发服务能力更强，可以支持大量的平行连接。

PHP-FPM 自 PHP-5.3.3 起开始加入到了 PHP 核心，编译时加上`--enable-fpm`就可以提供支持，然后 PHP-FPM 以守护进程在后台运行，在 Nginx 响应请求后，自行处理静态请求，PHP 请求则经过 `fastcgi_pass` 交由 PHP-FPM 处理并在处理完毕后返回，因此 Nginx 和 PHP-FPM 的组合成为了一种稳定、高效的 PHP 运行方式，效率要比传统的 Apache 和 mod_php 高出不少。

Node.js 提供事件驱动和非阻塞 I/O API，其设计目标是任何需要操作 I/O 的函数都使用回调函数，可以优化应用程序的吞吐量和规模，这些技术通常被用于实时应用程序。

libuv 是一个网络和文件系统功能的抽象层，既可以用于 Windows 又可以用于符合 POSIX 标准的系统（例如 Linux、OS X 和 Unix），同时 Node.js 使用 libuv 来处理异步事件，而且 V8 提供了 JavaScript 的实时运行环境。

```
var http = require('http');

http.createServer(function(request, response){
  response.writeHead(200,{'Content-Type': 'text/plain'});
  resposne.end('Hello World\n');
}).listen(8000);
```

```
console.log('Server running at http://127.0.0.1:8000/');
```

基于 Node.js 也可以实现简单的 TCP 服务器，并监听端口 7000 来输出（echo）之前输入的消息：

```
var net = require('net');

net.createServer(function (stream){
  stream.write('hello\n');

  stream.on('end',function(){
    stream.end('goodbye\r\n');
  });
  stream.pipe(stream);
}).listen(7000);
```

Redis 数据库将全部的数据存储在内存中，并且使用快照以半持久耐用模式将数据集以异步方式从内存以 RDB 格式写入硬盘。

Swoole 提供了异步任务处理的功能，可以投递一个异步任务到 TaskWorker 进程池中执行，不影响当前请求的处理速度。

基于第一个 TCP 服务器，只需要增加 onTask 和 onFinish2 个事件回调函数就可以实现异步任务处理。

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

$tcpserver->set(array(
    'worker_num' => 8, //工作进程数量
    'daemonize' => 0, //是否作为守护进程
    'task_worker_num' => 4 // 设置异步任务的工作进程数量
));

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

//监听数据发送事件
$tcpserver->on('receive', function ($tcpserver, $fd, $from_id, $data) {
    $tcpserver->send($fd, 'Swoole TCP Server: '.$data);
    // //投递异步任务
    $task_id = $tcpserver->task($data);
    echo "Dispatch AsyncTask: id = $task_id\n";
    $tcpserver->close($fd);
});

// 处理异步任务
$tcpserver->on('task',function($tcpserver,$task_id,$fram_id,$data){
```

```
    echo "New AsyncTask[id=$task_id]" . PHP_EOL;
    // 返回任务执行的结果
    $tcpserver->finish("$data-> OK");
});

// 处理异步任务的结果
$tcpserver->on('finish',function($tcpserver,$task_id,$data){
    echo "AsyncTask[$task_id] Finish: $data" . PHP_EOL;
});

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

//启动服务器
$tcpserver->start();
```

另外，需要设置 task 进程数量，可以根据任务的耗时和任务量配置适量的 task 进程。

在异步处理任务过程中，调用 `$tcpserver->task()` 后，程序立即返回，继续向下执行代码。onTask 回调函数 Task 在进程池内被异步执行，执行完成后调用 `$tcpserver->finish()` 返回结果。

注意，finish 操作是可选的，也可以不返回任何结果。

Part VI

Memory

Chapter 23

Overview

Chapter 24

Lock

Chapter 25

Buffer

Chapter 26

Table

swoole_table 一个基于共享内存和锁实现的超高性能[1]的并发数据结构，可以用于解决多进程/多线程数据共享和同步加锁问题。

- 性能强悍，单线程每秒可读写 50 万次。8 核机器 8 线程可以支持每秒并发读写 400 万次。
- 无需加锁，swoole_table 内置行锁自旋锁，所有操作均是多线程/多进程安全。用户层完全不需要考虑数据同步问题。
- 支持多进程，swoole_table 可以用于多进程之间共享数据。

swoole_table 使用行锁，而不是全局锁，仅当 2 个进程在同一 CPU 时间，并发读取同一条数据才会进行发生抢锁。

实验性质的 swoole_table 模块不建议在生产环境中使用，应该继续使用 Redis/Apc/Yac 作为数据共享层。

26.1 Countable

swoole_table 类实现了迭代器和 Countable 接口，可以使用 foreach 进行遍历，并使用 count 计算当前行数。

```
foreach($table as $row){
  var_dump($row)
}
echo count($table);
```

后续的 swoole_table 会增加 incr 方法，用于内存级计数器。

[1] swoole_table-0.1 版本，单进程压测每秒可读写 50W 次左右。

Chapter 27

Atomic

Part VII

HttpServer

Chapter 28

Overview

Part VIII

WebSocket

Chapter 29

Overview

在 WebSocket 出现之前，Web 交互一般是基于 HTTP 协议的短连接或者长连接。例如，Comet[1] 曾经作为一个权宜实现来让服务器实时地将更新的信息传送到客户端，而无须客户端发出请求。

虽然 WebSocket 的设计初衷是用于 Web 浏览器和 Web 服务器，但是实际上可以被用于任何类型的客户端/服务器应用中。例如，WebSocket 通信协议实现的是基于浏览器的原生 socket，这样原先只有在 C/S 模式下的开发模式都可以移植到 Web，从而就可以通过浏览器的支持在 Web 上实现与服务器端的 Socket 通信。

29.1 Handshake

在上述的 WebSocket 客户端示例中，当 Web 应用程序调用 `new WebSocket(url)` 接口时，客户端就开始了与地址为 url 的 WebServer 建立握手连接的过程。

1. WebSocket 客户端与 WebSocket 服务器通过 TCP 三次握手建立连接，如果这个建立连接失败，那么后面的过程就不会执行，WebSocket 客户端将收到错误消息通知。
2. 在 TCP 建立连接成功后，WebSocket 客户端通过 http 协议传送 WebSocket 支持的版本号、协议的子版本号、原始地址、主机地址等一些列字段给服务器端。
3. WebSocket 服务器收到 WebSocket 客户端发送来的握手请求后，如果数据包数据和格式正确，客户端和服务器端的协议版本号匹配等，就接受本次握手连接，并给出相应的数据回复，同样回复的数据包也是采用 http 协议传输。
4. WebSocket 客户端收到服务器回复的数据包后，如果数据包内容、格式都没有问题的话，就表示本次连接成功，触发 onopen 消息，此时 Web 开发者就可以在此时通过 send 接口想服务器发送数据。否则，握手连接失败，Web 应用程序会收到 onerror 消息，并且能知道连接失败的原因。

这个握手很像 HTTP，但是实际上却不是，它允许服务器以 HTTP 的方式解释一部分 handshake 的请求，然后切换为 WebSocket。

[1] 具体来说，Comet 基于长轮询或 iframe 流来实现，仍然基于 HTTP 连接，其中：
• 长轮询是在打开一条连接以后保持，等待服务器推送来数据再关闭的方式。
• iframe 流方式是在页面中插入一个隐藏的 iframe，利用其 src 属性在服务器和客户端之间创建一条长链接，服务器向 iframe 传输数据（通常是 HTML，内有负责插入信息的 javascript）来实时更新页面。

29.2 Heartbeat

心跳包就是在客户端（例如即时通信）和服务器之间定时通知对方自己状态的一个自己定义的命令字，按照一定的时间间隔定时发送，类似于心跳，所以叫做心跳包。

通常情况下，心跳包都是由客户端定时向服务器发送简单的固定信息来告诉服务器端自己还在线，并接收服务器响应的固定信息。如果服务端在规定的时间内没有收到客户端信息则将客户端视为断开。

- 发包方：可以是客户端也可以是服务端，看哪边实现方便合理而定（一般是客户端）。
- 收包方：作为收包方的服务器也可以定时轮询发心跳包，不过会消耗服务器资源。

网络中的数据接收和发送都是使用操作系统中的 Socket 进行实现的，但是如果 Socket 已经断开，那么发送数据和接收数据的时候就一定会有问题。

- 如果服务器判定客户端掉线并关闭，那么客户端后续发送的数据包被直接丢弃。
- 如果 Socket 连接断开，那么服务器会继续监听客户端 IP 地址和端口，客户端无法将数据发送出去。

为了判断 Socket 是否还可以继续使用，需要在系统中创建心跳机制，其实质就是为了保持长连接。

其实，TCP 中已经为用户实现了一个叫做心跳[2]的机制，如果用户设置了心跳，那么 TCP 就会在一定的时间（比如设置的是 3 秒钟）内发送当前设置的次数的心跳（比如说 2 次），并且此信息不会影响用户自己定义的协议。

具体来说，心跳包的内容没有什么特别规定，不过一般都是在逻辑层发送很小的包，或者只包含包头的一个空包。例如，所谓 TCP “心跳” 就是定时发送一个自定义的结构体（心跳包或心跳帧），可以让对方知道自己 “在线” 以确保连接的有效性。

如果由服务器端主动发送心跳包来检测客户端在线状态，在一定时间间隔下发送（send）一个空包给客户端后，需要客户端反馈一个同样的空包回来，那么在一定时间内收不到（recv）客户端发送过来的反馈包时，服务器就只有认定客户端掉线。

在长连接下，有可能很长一段时间都没有数据往来，而且在理论上来说，长连接是一直保持连接的，只是实际情况中中间节点出现某些故障是难以知道的。

有些情况下，有的节点（防火墙）会自动把一定时间之内没有数据交互的连接断开，因此特别需要心跳包来维持长连接的保活。

在确定客户端断线之后，服务器的逻辑逻辑层根据需求可能需要做一些事情，比如断线后的数据清理以及重新连接等，因此心跳包主要用于长连接的保活和断线处理。

一般的应用下，判定客户端断线的时间在 30-40 秒左右，如果对于实时性要求高，可以设置 6-9 秒。例如，心跳包在 GRPS 通信和 CDMA 通信的应用方面使用非常广泛，心跳包通常设定在 30-40 秒之间，而且数据网关会定时清理没有数据的路由。

29.3 Protocol

WebSocket 是 html5 提出的一个协议规范，并被 IETF 以 RFC 6455 标准化，后来由 W3C 在 Web IDL 中将 WebSocket API 标准化。

WebSocket 并不基于 HTTP 连接，而是在一个单独的 TCP 连接上提供了全双工的通信信道，而

[2]TCP 本身实现的心跳包机制是 TCP 的选项，默认设置是 2 小时的心跳频率，不过检查不到机器断电、网线拔出、防火墙等断线类型。