---

layout: post

title: "PHP编程相关问题总结"

date: 2017-04-05 00:05:00

categories: 开发总结

tags: PHP

excerpt: ""

---

\* content

{:toc}

# PHP

---

---

## PHP语言与扩展

---

### foreach循环中使用引用有什么潜在问题？

```php

<?php

$array = [1, 2, 3];

echo implode(',', $array), "<br/>"; // 1,2,3

foreach ($array as &$value) {}

echo implode(',', $array), "<br/>"; // 1,2,3

foreach ($array as $value) {}

echo implode(',', $array), "<br/>"; // 1,2,2

```

\*\*第一个循环过后，$value是数组中最后一个元素的引用。\*\*

第二个循环开始：

\* 第1步：复制$arr[0]到$value（注意此时$value是$arr[2]的引用），这时数组变成[1,2,1]

\* 第2步：复制$arr[1]到$value，这时数组变成[1,2,2]

\* 第3步：复制$arr[2]到$value，这时数组变成[1,2,2]

综上，最终结果就是1,2,2

避免这种错误最好的办法就是\*\*在循环后立即用unset函数销毁变量\*\*：

```php

<?php

$arr = array(1, 2, 3, 4);

foreach ($arr as &$value) {

$value = $value \* 2;

}

unset($value);

```

### \_\_autoload和spl\_autoload\_register()的主要区别？

函数原型：

```php

<?php

void \_\_autoload(string $class)

bool spl\_autoload\_register ([ callable $autoload\_function [, bool $throw = true [, bool $prepend = false ]]] )

```

区别：

\* spl\_autoload\_register()函数会将Zend Engine中的\_\_autoload()函数取代为spl\_autoload()或spl\_autoload\_call()。\_\_

\* \_\_spl\_autoload\_register 可以很好地处理需要`多个加载器`的情况，这种情况下spl\_autoload\_register会`按顺序依次调用`之前注册过的加载器\*\*。作为对比， \_\_autoload 因为是一个函数，所以只能被定义一次。

### 使用spl\_autoload\_register()调用静态方法

```php

<?php

class test {

public static function loadprint( $class ) {

$file = $class . '.class.php';

if (is\_file($file)) {

require\_once($file);

}

}

}

spl\_autoload\_register(['test','loadprint']);

// 另一种写法：spl\_autoload\_register("test::loadprint");

$obj = new PRINTIT();

$obj->doPrint();

```

### CGI、FastCGI的理解和区别

\*\*CGI\*\*

CGI即\*\*公共网关接口\*\*(Common Gateway Interface)，描述了客户端和服务器程序之间传输数据的一种协议，就是规定要传哪些数据、以什么样的格式传递给后方处理这个请求的协议。总之，\*\*CGI是一种标准，一种协议\*\*。

\*\*CGI程序\*\*

CGI程序确保CGI协议的顺利执行，并且返回结果，用来\*\*沟通程序（如PHP、Python、 Java）和Web服务器（Apache2、Nginx）\*\*，充当桥梁的作用。即CGI程序是介于Web服务器与Web程序之间的，用来保证CGI协议的程序。如php-fpm。

Web Server（比如说nginx）只是内容的分发者。比如，如果客户端请求/index.html，那么web server会去文件系统中找到这个文件，发送给浏览器，这里分发的是静态数据。如果现在请求的是/index.php，根据配置文件，nginx知道这个不是静态文件，那它就要找到相应的CGI程序来处理了，根据配置，找到了php-cgi这个程序（也就是PHP解析器），那么他会把这个请求交个php-cgi（安装PHP时会附带安装的一个程序）。nginx会启动CGI程序，然后会根据CGI协议，把需要的数据传给全部丢给php-cgi，比如请求的url，查询字符串，POST数据，HTTP header等等。php-cgi会解析php.ini文件，初始化执行环境，然后找到index.php，编译，执行。程序的输出会由Web服务器收集，并加上合适的HTTP头，再发送回客户端。

\*\*CGI执行的特点\*\*：每次请求过来，再启动CGI程序，去处理请求。所以就会造成，如果n个请求过来，就得启动n个CGI程序的进程来处理，处理完之后，就销毁相应的线程。但是启动一个进程，肯定是需要时间的，这样大量的启动、销毁，造成了大量的浪费。

一些更有效的技术可以让脚本解释器直接作为模块集成在 Web 服务器（例如：Apache，mod\_php）中，这样就能避免重复载入和初始化解释器。不过这只是就那些需要解释器的高级语言（即解释型语言）而言的，使用诸如C一类的编译语言则可以避免这种额外负荷。

\*\*FastCGI\*\*

与为每个请求创建一个新的进程不同，FastCGI使用持续的进程来处理一连串的请求。FastCGI会启动FastCGI进程管理器（简称`master`），解析配置文件，初始化执行环境，master再启动多个CGI程序（简称`worker`）在那里等候。

当进来一个请求时，web服务器把环境变量和这个页面请求通过一个socket比如FastCGI进程与web服务器(都位于本地）或者一个TCP connection（FastCGI进程在远端的server farm）传递给FastCGI进程。Apache通过mod\_fcgid以及较早的第三方mod\_fastcgi模块来实现。

\*\*master的执行特点\*\*：请求过来，把请求传递给到空闲的worker，然后立即可以接受下一个请求，再传递。

\*\*worker的职责\*\*：每个worker都一直在等候，接到从master传递过来的请求之后，立即执行并返回，但是\*\*执行完毕后，不销毁，而且继续等待下个请求\*\*。当然，master还时时刻刻监控着worker的情况，如果worker不够了，master会启动多几个，如果空闲的太多了，停掉一些。这样一种方法，\*\*比起CGI执行，省去了大量的启动、销毁线程的时间和节约了大量的资源\*\*。

修改php.ini后，FastCGI可以平滑的重启（执行`php-fpm reload`）。php-cgi进程是没办法平滑重启的，需要restart。

### Cookie与Session

Cookie与Session都属于\*\*会话跟踪技术\*\*。理论上，一个用户的所有请求操作都应该属于同一个会话。HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭（指非长连接情况下），再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话。在Session出现之前，基本上所有的网站都采用Cookie来跟踪会话。目前Cookie已经成为标准，所有的主流浏览器都支持Cookie（需要浏览器的支持：保存、更新、发送、删除。不同的浏览器保存Cookie的方式不同）。

\*\*查看网站Cookie的简单方式\*\*

直接在浏览器地址栏中输入

```

javascript:alert(document.cookie)

```

\*\*Cookie\*\*

\*\*修改Cookie\*\*

\*\*要想修改Cookie只能使用一个同名的Cookie来覆盖原来的Cookie\*\*。\*\*删除时只需要把maxAge修改为0即可\*\*（Cookie并不提供直接的修改、删除操作）。修改、删除Cookie时，新建的Cookie除value、maxAge之外的所有属性，例如name、path、domain等，都要与原Cookie完全一样。否则，浏览器将视为两个不同的Cookie不予覆盖，导致修改、删除失败。

从客户端读取Cookie时，包括maxAge在内的其他属性都是不可读的，也不会被提交。\*\*浏览器提交Cookie时只会提交name与value属性\*\*。maxAge属性只被浏览器用来判断Cookie是否过期。

\*\*Expires\*\*

Cookie的Expires属性标识了Cookie的有效时间，当Cookie的有效时间过了之后，这些数据就被自动删除了。默认情况下Cookie是暂时存在的，他们存储的值只在浏览器会话期间存在，当用户退出浏览器后这些值也会丢失，如果想让Cookie存在一段时间，就要为expires属性设置为未来的一个过期日期。

\*\*max-age\*\*

expires属性现在已经被max-age属性所取代，max-age用秒来设置cookie的生存期。

\*\*path\*\*

`path属性`决定允许访问Cookie的路径。页面只能获取它属于的Path的Cookie。例如/session/test/a.php不能获取到路径为/session/abc/的Cookie。

\*\*secure\*\*

如果不希望Cookie在HTTP等非安全协议中传输，可以设置Cookie的`secure属性`为true。浏览器只会在HTTPS和SSL等安全协议中传输此类Cookie。secure属性并不能对Cookie内容加密，因而不能保证绝对的安全性。如果需要高安全性，需要在程序中对Cookie内容加密、解密，以防泄密。

W3C标准的浏览器会阻止JavaScript读写任何不属于自己网站的Cookie。

\*\*临时性Cookie\*\*

`maxAge`为负数的Cookie，为\*\*临时性Cookie，不会被持久化，不会被写到Cookie文件中\*\*。Cookie信息\*\*保存在浏览器内存中\*\*，因此关闭浏览器该Cookie就消失了。Cookie默认的maxAge值为-1。

\*\*Session\*\*

Session在用户第一次访问服务器的时候`自动创建`。Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。为防止内存溢出，服务器会把长时间内没有活跃的Session从内存删除。这个时间就是\*\*Session的超时时间\*\*。如果超过了超时时间没访问过服务器，Session就自动失效了。

虽然Session保存在服务器，对客户端是透明的，\*\*它的正常运行仍然需要客户端浏览器的支持\*\*（如果使用Cookie来发送SessionID的话）。这是因为Session需要使用Cookie作为识别标志。\*\*如果浏览器不支持Cookie，则需要依赖URL重写。\*\*

URL地址重写是对客户端不支持Cookie的解决方案。URL地址重写的原理是将该用户Session的id信息重写到URL地址中。服务器能够解析重写后的URL获取Session的id。

\*\*Cookie与Session的比较\*\*

\* Session的使用比Cookie方便，但是过多的Session存储在服务器内存中，会对服务器造成压力。

\* Cookie具有`不可跨域名性`（这里指全域名，而不是仅仅指根域名。同一个一级域名下的两个二级域名如www.demo.com和images.demo.com也不能交互使用Cookie，因为二者的域名并不严格相同。如果想所有demo.com名下的二级域名都可以使用该Cookie，需要设置Cookie的domain参数为`.demo.com`（以.开头），这样所有以`demo.com`结尾的域名都可以访问该Cookie。

\* Cookie中使用Unicode字符时需要对Unicode字符进行编码（\*\*Cookie中保存中文只能编码\*\*，推荐使用UTF-8，因为JavaScript不支持GBK编码）。

\* Cookie的内容应该少而精（由于浏览器每次请求服务器都会携带Cookie，因此Cookie内容不宜过多，否则影响速度）。

### cgi.fix\_pathinfo配置项的作用是什么？

如果WebServer为Nginx，则须在PHP的配置文件php.ini中配置cgi.fix\_pathinfo = 0，防止nginx文件解析漏洞。

在cgi.fix\_pathinfo = 1的情况下，假设有如下的 URL：http://xxx.net/foo.jpg，当访问 http://xxx.net/foo.jpg/a.php 时，foo.jpg 将会被执行，如果 foo.jpg 是一个普通文件，那么 foo.jpg 的内容会被直接显示出来，但是如果把一段php代码保存为foo.jpg，那么问题就来了，这段代码就会被直接执行。

### mysqli的Prepared语句有什么好处？

mysqli支持prepared语句，好处有2：

\* 对于在执行大量具有不同数据的相同查询时，可以提高执行速度（MySQL省去了部分解析工作）；

\* 可以\*\*避免SQL注入攻击\*\*；

通常的数据库插入操作如下：

```php

<?php

$query = "insert into books values('".$isbn."','".$author."','".$title."','".$price."')";

$result = $db->query($query);

if($result){

echo $db->affected\_rows;

...

```

Prepared语句的基本思想是\*\*向MySQL发送一个需要执行的查询模板，然后再单独发送数据\*\*。因此可以向相同的Prepared语句发送大量相同的数据，对于批处理的插入操作来说是非常有用的。

```php

<?php

$query = “insert into books values(?,?,?,?)”;

$stmt = $db->prepare($query);

$stmt->bind\_param("sssd",$isbn,$author,$title,$price);

$stmt->execute();

echo $smtt->affected\_rows;

$stmt->close();

```

对于查询操作，也可以绑定查询结果至变量：

```php

<?php

$stmt->bind\_result($isbn,$author,$title,$price);

```

### PHP中有哪几种表达字符串的方式？区别是什么？

一个字符串可以用4种方式表达（\*茴香豆的四种写法\*）：

\* 单引号

\* 双引号

\* heredoc语法结构

\* nowdoc语法结构（自 PHP 5.3.0 起）

不像双引号和heredoc语法结构，\*\*在单引号字符串中的变量和特殊字符的转义序列将不会被替换\*\*。

要注意的是heredoc结束标识符这行除了可能有一个分号（;）外，绝对不能包含其它字符。这意味着标识符不能缩进，分号的前后也不能有任何空白或制表符。

就像heredoc结构类似于双引号字符串，\*\*nowdoc结构是类似于单引号字符串的\*\*（不进行解析操作、转义）。

字符串会被按照该脚本文件相同的编码方式来编码。因此如果一个脚本的编码是 ISO-8859-1，则其中的字符串也会被编码为 ISO-8859-1。

### PHP中的闭包无须像js（非ES6）一样通过self=this的形式来扩展作用域

```php

<?php

class Demo{

private $list = [1,2,3,4,5,6,7,8,9,10];

public $delList = [];

public function test(){

$arr = array\_filter($this->list,function($n){

if($n % 2 == 0){

return true;

}

else{

$this->delList[] = $n; // 此$this指向Demo的对象，而非闭包函数

return false;

}

});

return $arr;

}

}

$demo = new Demo();

$arr = $demo->test();

var\_dump($arr); // 1 3 5 7 9

var\_dump($demo->delList); // 2 4 6 8 10

```

### PHP中那些函数是同步阻塞的？

\*\*同步阻塞函数\*\*

\* mysql、mysqli、pdo以及其他DB操作函数

\* sleep、usleep

\* curl

\* stream、socket扩展的函数

\* swoole\_client同步模式

\* memcache、redis扩展函数

\* file\_get\_contents/fread等文件读取函数

\* swoole\_server->taskwait

\* swoole\_server->sendwait

swoole\_server的PHP代码中有上述函数，Server就是`同步服务器`。代码中没有上述函数就是`异步服务器`。

### 异步非阻塞函数

\* swoole\_client异步模式

\* mysql-async库

\* redis-async库

\* swoole\_timer\_tick/swoole\_timer\_after

\* swoole\_event系列函数

\* swoole\_table/swoole\_atomic/swoole\_buffer

\* swoole\_server->task/finish函数

### PHP对象注入漏洞的原因是什么？

\*\*序列化与反序列化\*\*

PHP支持\*\*对象的序列化和反序列化\*\*操作（serialize、unserialize）。

如：

```php

<?php

class User{

public $age = 0;

public $name = '';

public function PrintData(){

echo 'User ' . $this->name . ' is ' . $this->age . ' years old. <br />';

}

}

$usr = unserialize('O:4:"User":2:{s:3:"age";i:20;s:4:"name";s:4:"John";}');

$usr->PrintData();

```

输出：

```

User John is 20 years old.

```

\*\*当一个对象进行序列化和反序列化操作时也会自动调用其他相应的魔术方法\*\*：

\* 当对象进行序列化操作时魔术方法`\_\_sleep()`会被自动调用（必须返回一个包含序列化的类变量名的数组）。

\* 当对象进行反序列化操作时魔术方法`\_\_wakeup()`会被自动调用。

### 反序列化存在的安全问题

攻击者可以\*\*操作类变量\*\*来攻击web应用，比如：

```php

<?php

$usr = unserialize('O:7:"LogFile":1:{s:8:"filename";s:9:".htaccess";}');

$usr->PrintData();

```

从而\*\*意外地执行了LogFile的`\_\_construct()`和`\_destruct()`\*\*。

在处理由用户提供数据的地方不要使用`unserialize()`，可以使用`json\_decode(...)`。

### PHP支持哪些数据类型？

PHP共支持\*\*8种原始数据类型\*\*，其中包括：

\* 4种标量类型：`boolean`、`integer`、`float（double）`、`string`

\* 2种复合类型：`array`、`object`

\* 2种特殊类型：`resource`、`NULL`

\*\*double和float是相同的\*\*，由于一些历史的原因，这两个名称同时存在。

变量的类型通常不是由程序员设定的，确切地说，是\*\*由PHP根据该变量使用的上下文在运行时决定的\*\*。

### PHP的变量作用域规则是什么样的？

\*\*6项基本的作用域规则\*\*：

\* `超级全局变量`（内置）可以在脚本的任何地方使用和可见。

\* \*\*`常量`一旦被声明，将全局可见\*\*，即可以在函数内外使用。

\* 在一个`脚本中声明的全局变量`在整个脚本中是可见的。

\* 函数内部使用的变量声明为全局变量时，其名称要与全局变量一致。

\* 在函数内部创建的`静态变量`，在函数外部不可见，但是可以在函数的多次执行中保持值。

\* 在函数内部创建的非静态变量，当函数终止时就不存在了。

### SAPI

无论是Web模式还是Cli模式运行，PHP的工作原理都是一样的， 都是\*\*作为一种SAPI在运行\*\*（Server Application Programming Interface： \*the API used by PHP to interface with Web Servers\*）。\*\*SAPI就是PHP和外部环境的代理器\*\*。它把外部环境抽象后, \*\*为内部的PHP提供一套固定的，统一的接口，使得PHP自身实现能够不受错综复杂的外部环境影响\*\*，保持一定的独立性。

### PHP扩展程序的基本执行方式

\*\*MINIT\*\*

\*\*在PHP随着Apache的启动而常驻在内存里时\*\*， 会把自己所有已加载扩展的`MINIT方法`(全称Module Initialization，是由每个模块自己定义的函数)都执行一遍。 在这个时间里，扩展可以定义一些自己的常量、类、资源等所有会被用户端的PHP脚本用到的东西。

\*\*RINIT\*\*

\*\*当一个页面请求到来时候\*\*，PHP会迅速的开辟一个新的环境，并重新扫描自己的各个扩展， 遍历执行它们各自的`RINIT方法`(俗称Request Initialization)， 这时候一个扩展可能会初始化在本次请求中会使用到的变量等， 还会初始化稍后用户端（即PHP脚本）中的变量之类的。

\*\*RSHUTDOWN\*\*

\*\*当请求完成\*\*（或者别die等终结），PHP便会启动回收程序，执行所有已加载扩展的`RSHUTDOWN`（Request Shutdown）方法， 这时候扩展可以抓紧利用内核中的变量表之类的做一些事情， 因为一旦PHP把所有扩展的RSHUTDOWN方法执行完， 便会释放掉这次请求使用过的所有东西， 包括变量表的所有变量、所有在这次请求中申请的内存等等。

\*\*MSHUTDOWN\*\*

\*\*当Apache通知PHP自己要Stop的时候\*\*，PHP便进入`MSHUTDOWN`（Module Shutdown）阶段。一旦PHP把扩展的MSHUTDOWN执行完，便会进入自毁程序，所以这里一定要把自己擅自申请的内存给释放掉。

\*\*一个最简单的例子\*\*

walu.c

```c

int time\_of\_minit; // 每次请求都不变

PHP\_MINIT\_FUNCTION(walu)

{

time\_of\_minit=time(NULL);

return SUCCESS;

}

int time\_of\_rinit; // 每次请求都改变

PHP\_RINIT\_FUNCTION(walu)

{

time\_of\_rinit=time(NULL);

return SUCCESS;

}

// 每次页面请求都会往time\_rshutdown.txt中写入数据

PHP\_RSHUTDOWN\_FUNCTION(walu)

{

FILE \*fp=fopen("/cnan/www/erzha/time\_rshutdown.txt","a+");

fprintf(fp,"%d\n",time(NULL));

fclose(fp);

return SUCCESS;

}

// 只有在apache结束后time\_mshutdown.txt才写入有数据

PHP\_MSHUTDOWN\_FUNCTION(walu)

{

FILE \*fp=fopen("/cnan/www/erzha/time\_mshutdown.txt","a+");

fprintf(fp,"%d\n",time(NULL));

return SUCCESS;

}

PHP\_FUNCTION(walu\_test)

{

php\_printf("%d&lt;br /&gt;",time\_of\_minit);

php\_printf("%d&lt;br /&gt;",time\_of\_rinit);

return;

}

```

### PHP的生命周期

PHP扩展程序的两种init（MINIT、RINIT）和两种shutdown（RSHUTDOWN、MSHUTDOWN）各会执行多少次、各自的执行频率有多少\*\*取决于PHP是用什么SAPI与宿主通信的\*\*。最常见的四种SAPI通信方式如下：

\*\*1.直接以CLI/CGI模式调用\*\*

PHP的生命周期完全在一个单独的请求中完成，两种init和两种shutdown仍然都会被执行。

以`php -f test.php`为例，执行过程如下：

\* （1）调用每个扩展的MINIT；

\* （2）请求test.php文件；

\* （3）调用每个扩展的RINIT；

\* （4）执行test.php；

\* （5）调用每个扩展的RSHUTDOWN；

\* （6）执行清理操作；

\* （7）调用每个扩展的MSHUTDOWN；

\* （8）终止php；

\*\*2.多进程模块\*\*

如编译成Apache2的Pre-fork MPM，当Apache启动的时候，会立即把自己fork出好几个子进程，每一个进程都有自己独立的内存空间，在每个进程里的PHP的工作方式如下：

\* （1）调用每个扩展的MINIT；

\* （2）\*\*循环\*\*：{ a.调用每个扩展的RINIT; b.执行脚本； c.调用每个扩展的RSHUTDOWN；}

\* （3）调用每个扩展的MSHUTDOWN；

\*\*3.多线程模块\*\*

如IIS的isapi和Apache MPM worker，\*\*只有一个服务器进程在运行着，但会同时运行很多线程\*\*，这样可以减少一些资源开销，像Module init和Module shutdown就只需要运行一次就行了，一些全局变量也只需要初始化一次， 因为线程独具的特质，使得各个请求之间方便的共享一些数据成为可能。

\*\*4.Embedded(嵌入式，在自己的C程序中调用Zend Engine)\*\*

Embed SAPI是一种比较特殊但不常用的SAPI，允许在C/C++语言中调用PHP/ZE提供的函数。 这种SAPI和上面的三种一样，按Module Init、Request Init、Rshutdown、mshutdown的流程执行着。

### PHP配置中的open\_basedir配置项的作用是什么？

open\_basedir是PHP配置中为了\*\*防御跨目录进行文件（目录）读写\*\*的配置，所有PHP中有关文件读、写的函数都会经过open\_basedir的检查。实际上是一些目录的集合，在定义了open\_basedir以后，\*\*PHP可以读写的文件、目录都将被限制在这些目录中\*\*。在Linux下，不同的目录由`:`分割，如`/var/www/:/tmp/`。

注意用open\_basedir指定的限制实际上是前缀，而不是目录名。

\*\*Apache运行PHP有三种配置open\_basedir的方法\*\*

\* （1）在php.ini里配置

`open\_basedir = .:/tmp/`

\* （2）在Apache配置的VirtualHost里设置(httpd-vhosts.conf)

`php\_admin\_value open\_basedir .:/tmp/`

\* （3）在Apache配置的Direcotry里设置

`php\_admin\_value open\_basedir .:/tmp/`

\*\*关于三个配置方法的解释\*\*

（1）方法（2）的优先级高于方法（1），也就是说方法（2）会覆盖方法（1）；方法（3）的优先级高于方法（2），也就是说方法（3）会覆盖方法（2）；

（2）配置目录里加了`/tmp/`是因为PHP默认的临时文件（如上传的文件、session等）会放在该目录，所以一般需要添加该目录，否则部分功能将无法使用；

（3）配置目录里加了`.`是指运行PHP文件的当前目录，这样做可以避免每个站点一个一个设置；

（4）如果站点还使用了站点目录外的文件，需要单独在对应VirtualHost设置该目录；

### 使用数组、SplQueue实现队列的性能比较

PHP的SPL标准库中提供了内置的队列数据结构SplQueue。另外PHP的数组也提供了array\_pop和array\_shift可以使用数组模拟队列数据结构。

\*\*虽然使用Array可以实现队列，但实际上性能会非常差\*\*。在一个大并发的服务器程序上，建议使用SplQueue作为队列数据结构。

100万条数据随机入队、出队，使用SplQueue仅用2312.345ms即可完成，而使用Array模拟的队列的程序根本无法完成测试，CPU一直持续在100%，降低到1万条后，也需要260ms才能完成测试。

SplQueue

```php

<?php

$splq = new SplQueue;

for($i = 0; $i < 1000000; $i++){

$data = "hello $i\n";

$splq->push($data);

if ($i % 100 == 99 and count($splq) > 100){

$popN = rand(10, 99);

for ($j = 0; $j < $popN; $j++){

$splq->shift();

}

}

}

$popN = count($splq);

for ($j = 0; $j < $popN; $j++){

$splq->pop();

}

```

Array队列

```php

<?php

$arrq = array();

for($i = 0; $i <1000000; $i++){

$data = "hello $i\n";

$arrq[] = $data;

if ($i % 100 == 99 and count($arrq) > 100){

$popN = rand(10, 99);

for ($j = 0; $j < $popN; $j++){

array\_shift($arrq);

}

}

}

$popN = count($arrq);

for ($j = 0; $j < $popN; $j++){

array\_shift($arrq);

}

```

### 为什么PHP不做编译优化？

C程序通常一次编译，多次运行或长时间运行，因此在编译上多耗些时间、多做些优化被认为是值得的。而解释型语言往往作为胶水语言，也就是\*\*完成一项用后即弃的特定任务\*\*。官方的答复是，PHP程序运行时间往往很短暂，比如10ms；如果花100ms做编译优化，把运行时间压缩到1ms，总的时间消耗是101ms，反而更慢了（不考虑中间代码缓存）。

### 为什么在被包含文件中仍然要使用PHP起始标志？

当一个文件被包含时，\*\*语法解析器在目标文件的开头脱离PHP模式并进入HTML模式，到文件结尾处恢复\*\*。由于此原因，目标文件中需要作为PHP代码执行的任何代码都必须被包括在有效的PHP起始和结束标记之中。

### 为什么要关闭register\_globals配置？

`register\_globals`是php.ini里的一个配置，这个配置影响到PHP如何接收传递过来的参数。

register\_globals的意思就是注册为全局变量，所以\*\*当On的时候，传递过来的值会被直接的注册为（与HTML控件的name属性同名的）全局变量直接使用\*\*，而Off的时候，需要到特定的全局变量数组里去得到它。

### 什么样的变量会被认为是NULL？如何判断？

特殊的NULL值表示一个变量没有值。\*\*NULL类型唯一可能的值就是NULL\*\*（不区分大小写）。在下列情况下一个变量被认为是 NULL：

\* 被赋值为 NULL

\* 尚未被赋值

\* 被unset()

判断是否为NULL：`is\_null()`

\*\*递减NULL值也没有效果，但是递增NULL的结果是 1\*\*。

### 哪些变量不能使用可变变量行为？

超全局变量不能用作可变变量。$this变量也是一个特殊变量，不能被动态引用。

### 在PHP中基于yield实现简单的协程通信（双向异步信息传递）

包含yield关键字的函数比较特殊，返回值是一个`Generator对象`，\*\*此时函数内语句尚未真正执行\*\*。Generator对象是Iterator接口实例，可以通过rewind()、current()、next()、valid()系列接口进行操纵。Generator可以视为一种\*\*“可中断”的函数\*\*，而yield构成了一系列的“中断点”。

```php

<?php

function gen() {

for($i=1;$i<=100;$i++) {

// yield既是语句，又是表达式，既具备类似return语句的功能，同时也有类似表达式的返回值（通过send得到的值）

$cmd = (yield $i);

if($cmd=='stop') {

return;

}

}

}

$gen = gen();

$i=0;

foreach($gen as $item) {

echo $item."\n";

if($i>=10) {

$gen->send('stop'); // 向Generator发送值

}

$i++;

}

```

### 如何控制数组json\_encode后为json对象或者json数组？

```php

<?php

$foo = array(

"item1" => (Object)[], // 关键点：将Array转换成Object类型

"item2" => []

);

echo json\_encode($foo);

```

输出：

```php

{"item1":{},"item2":[]}

```

### 在方法中使用的$this一定指向该方法所从属的对象吗？

`$this`是一个到主叫对象的引用，通常是该方法所从属的对象，但如果是从第二个对象静态调用时也可能是另一个对象。

```php

<?php

class A{

function foo(){

echo get\_class($this);

}

}

class B{

function bar(){

// Note: the next line will issue a warning if E\_STRICT is enabled.

A::foo();

}

}

$b = new B();

$b->bar();

```

输出：

```

B

```

### 在脚本文件中使用return，有哪些不同的场景和行为？

在脚本文件中使用return语言结构：

\* 如果在全局范围中调用，则当前脚本文件中止运行；

\* 如果当前脚本文件是被include的或者require的，则控制交回调用文件；且如果当前脚本是被include的，则return的值会被当作include调用的返回值。

\* 如果当前脚本文件是在php.ini中的配置选项auto\_prepend\_file或者auto\_append\_file所指定的，则此脚本文件中止运行；

注意既然\*\*return是语言结构而不是函数\*\*，因此其参数没有必要用括号将其括起来。通常都不用括号，实际上也应该不用，这样可以降低 PHP 的负担。 如果没有提供参数，则一定不能用括号，此时返回 NULL。如果调用 return 时加上了括号却又没有参数会导致解析错误。

### 如何使用OpCache提高PHP应用的性能？

php.ini中加入：

```ini

; 开关打开

opcache.enable=1

; 可用内存, 酌情而定, 单位megabytes(M)

opcache.memory\_consumption=256

; 最大缓存的文件数目, 命中率不到100%的话,可以试着提高这个值

opcache.max\_accelerated\_files=5000

; Opcache会在一定时间内去检查文件的修改时间, 这里设置检查的时间周期, 默认为2,单位为秒

opcache.revalidate\_freq=240

; interned string的内存大小,也可调

opcache.interned\_strings\_buffer=8

; 是否快速关闭, 打开后在PHP Request Shutdown的时候回收内存的速度会提高

opcache.fast\_shutdown=1

; 不保存文件/函数的注释

opcache.save\_comments=0

```

检查：

php -v

```

PHP 5.5.3-1ubuntu2.2 (cli) (built: Feb 28 2014 20:06:05)

Copyright (c) 1997-2013 The PHP Group

Zend Engine v2.5.0, Copyright (c) 1998-2013 Zend Technologies

with Zend OPcache v7.0.3-dev, Copyright (c) 1999-2013, by Zend Technologies

```

注意其中多出了

```

with Zend OPcache v7.0.3-dev

```

需要提醒的是，\*\*在生产环境中使用上述配置之前，必须经过严格测试\*\*。 因为上述配置存在一个已知问题，它会引发一些框架和应用的异常， 尤其是在存在文档使用了备注注解的时候。

如果在更新代码之后，发现没有执行的还是旧代码，可使用函数opcache\_reset()来清除缓存。

### 如何收集PHP的错误日志？

php在没有连接到数据库或者其他情况下会有提示错误，一般错误信息中会包含php脚本当前的路径信息或者查询的SQL语句等信息，这类信息暴露后，是不安全的，所以服务器建议禁止错误提示：

```

display\_errors = Off

```

在关闭display\_errors后为了能够把错误信息记录下来，便于查找服务器运行的原因,同时也要设置错误日志存放的目录，建议跟webserver的日志放在一起。

打开php.ini，安全加固配置方式如下，打开错误日志记录并设置错误日志存放路径：

```

log\_errors = On

# 该文件必须允许webserver的用户和组具有写的权限

error\_log = /usr/local/apache2/logs/php\_error.log

```

### 如何禁止在PHP中使用危险函数？

Web木马程序通常利用PHP的特殊函数执行系统命令，查询任意目录文件，修改文件等。

比如：

```php

<?php

eval($\_POST[cmd]);

?>

```

其实就是使用了一些危险函数使得应用存在漏洞，最好的防范方式就是不允许使用这些函数。

打开php.ini，安全加固配置方式如下，禁止使用这些危险函数：

```ini

disable\_functions = dl,assert,exec,popen,system,passthru,shell\_exec,proc\_close,proc\_open,pcntl\_exec

```

### 常量和变量有哪些不同？作用域是什么样的？

常量只能包含标量数据（boolean、integer、float和string）。可以定义resource常量，但应尽量避免，因为会造成不可预料的结果。

\*\*常量和变量有如下不同\*\*：

\* 常量前面没有美元符号（$）；

\* 常量只能用 `define()函数`或`const关键字`定义；

\* 用define定义的常量可以不用理会变量的作用域而在任何地方定义和访问；

\* 常量一旦定义就不能被重新定义或者取消定义；

\* 常量的值只能是标量；

如果使用了一个未定义的常量，PHP假定想要的是该常量本身的名字，如同用字符串调用它一样（CONSTANT 对应 "CONSTANT"）。\*\*常量名区分大小写\*\*。

```php

define("CONSTANT", "Hello world.");

echo CONSTANT; // Hello world.

echo Constant; // Use of undefined constant Constant - assumed 'Constant' in

```

如果只想检查是否定义了某常量，用 `defined()` 函数。

\*\*使用关键字const定义常量\*\*：

```php

// 以下代码在 PHP 5.3.0 后可以正常工作

const CONSTANT = 'Hello World';

echo CONSTANT;

```

和使用define()来定义常量相反的是，\*\*使用const关键字定义常量必须处于最顶端的作用区域\*\*，\*\*因为用此方法是在编译时定义的\*\*。这就意味着\*\*不能在函数内、循环内以及if语句之内用const来定义常量\*\*:

```php

<?php

function fun(){

const TEST = 1;

}

```

输出：

```

PHP Parse error: syntax error, unexpected 'const' (T\_CONST) in /usercode/file.php on line 4

```

### 打开allow\_url\_fopen、allow\_url\_include配置会有什么问题？

允许访问URL远程资源（就是允许fopen这样的函数打开url）使得PHP应用程序的漏洞变得更加容易被利用，php脚本若存在远程文件包含漏洞会使得攻击者直接获取网站权限及上传web木马。一般会在php配置文件中关闭该功能，若需要访问远程服务器建议采用其他方式如libcurl库。

```ini

allow\_url\_fopen = Off

allow\_url\_include = Off

```

比如有这样的代码：

```php

<?php

if(isset($HTTP\_GET\_VARS)){

reset($HTTP\_GET\_VARS);

while ( list($var, $val) = each($HTTP\_GET\_VARS) ) {

$$var=$val;

}

}

```

一些较偶然的场景会导致将\*\*以http://开头的get参数所表示的远程文件\*\*直接包含进来，然后执行。

### 有哪些通过PHP在Web服务器上执行命令的方式？

有4种主要的在Web服务器上执行命令的方法：

\* `exec()`函数：没有直接的输出，将返回命令执行结果的最后一行

\* `passthru()`函数：直接将输出显示到浏览器，如果输出是二进制，比如图片，这将是非常有用的

\* `system()`函数：将命令的输出回显到浏览器，它将每一行的输出向后对齐

\* 反引号``

更复杂的需求可以使用popen()、proc\_open()、proc\_close()，这些函数可以启动外部进程，并且在这些进程之间传递数据。

如果要把用户提交的数据包含在命令中，考虑到安全性问题，可以使用`escapeshellcmd`来阻止用户在系统上执行恶意的命令：

```php

system(escapeshellcmd($command));

```

### 有哪些魔术常量？

\* `\_\_LINE\_\_` 文件中的当前行号

\* `\_\_FILE\_\_` 文件的完整路径和文件名

\* `\_\_DIR\_\_` 文件所在的目录

\* `\_\_FUNCTION\_\_` 函数名称（PHP 4.3.0 新加）

\* `\_\_CLASS\_\_` 类的名称（PHP 4.3.0 新加）

\* `\_\_TRAIT\_\_`Trait 的名字（PHP 5.4.0 新加）

\* `\_\_METHOD\_\_` 类的方法名（PHP 5.0.0 新加）

\* `\_\_NAMESPACE\_\_` 当前命名空间的名称（区分大小写）

### 用于属性重载的魔术方法有哪些？

在给不可访问属性赋值时，`\_\_set()` 会被调用。

读取不可访问属性的值时，`\_\_get()` 会被调用。

当对不可访问属性调用 isset() 或 empty() 时，`\_\_isset()` 会被调用。

当对不可访问属性调用 unset() 时，`\_\_unset()` 会被调用。

属性重载只能在对象中进行。\*\*在静态方法中，这些魔术方法将不会被调用\*\*。

### 用于方法重载的魔术方法有哪些？

在对象中调用一个不可访问方法时，`\_\_call()` 会被调用。

用静态方式中调用一个不可访问方法时，`\_\_callStatic()` 会被调用。

### 为什么在声明静态变量时不能使用表达式进行赋值？

\*\*静态变量的声明是在编译时解析的\*\*，因此在声明静态变量时不能用表达式进行赋值：

```php

<?php

function foo(){

static $int = 0; // ok

static $int = 1+2; // error (as it is an expression)

static $int = sqrt(121); // error (as it is an expression too)

$int++;

echo $int;

}

```

### 能否将非对象类型转换为对象？如果可以，转换得到的对象有哪些属性？

\* 如果将一个对象转换成对象，它将不会有任何变化。

\* 如果其它任何类型的值被转换成对象，将会创建一个内置类`stdClass`的实例。

\* 如果该值为NULL，则新的实例为空。

\* 数组转换成对象将使键名成为属性名并具有相对应的值。

\* 对于任何其它的值，名为`scalar`的成员变量将包含该值。

```php

$a = (Object)'abc';

echo $a->scalar; // abc

echo gettype($a); // object

```

### 被包含文件的搜索顺序是什么样的？

\* （1）先按参数给出的路径寻找；

\* （2）如果没有给出目录（只有文件名）时则按照`include\_path`指定的目录寻找；

\* （3）如果在include\_path下没找到该文件则include最后才在\*\*调用脚本文件所在的目录和当前工作目录下寻找\*\*；

\* （4）如果最后仍未找到文件则include结构会发出一条警告；这一点和require不同，后者会发出一个致命错误；

\* （5）如果定义了路径，\*\*不管是绝对路径还是当前目录的相对路径\*\*，include\_path都会被完全忽略；

### 包含文件的作用域关系

\* （1）当一个文件被包含时，\*\*其中所包含的代码继承了include所在行的变量范围\*\*。从该处开始，\*\*调用文件在该行处可用的任何变量在被调用的文件中也都可用\*\*。不过\*\*所有在包含文件中定义的函数和类都具有全局作用域\*\*；

\* （2）如果include出现于调用文件中的一个函数里，则\*\*被调用的文件中所包含的所有代码将表现得如同它们是在该函数内部定义的一样\*\*。所以它将遵循该函数的变量范围。此规则的一个例外是魔术常量，它们是在发生包含之前就已被解析器处理的；

### Swoole的基本IO模型

\*\*Swoole采用多线程Reactor+多进程Worker\*\*

\* （1）Master进程启动一个Main Reactor线程和多个普通Reactor线程；

\* （2）请求到达Main Reactor；

\* （3）Main Reactor根据Reactor的情况，将请求注册给对应的Reactor(每个Reactor都有epoll，用来监听客户端的变化)

\* （4）客户端有变化时，交给worker来处理；

\* （5）worker处理完毕，通过进程间通信(比如管道、共享内存、消息队列)发给对应的reactor。

\* （6）reactor将响应结果发给相应的连接。

\* （7）请求处理完成；

因为reactor基于epoll，所以每个reactor可以处理无数个连接请求。如此，swoole就轻松的处理了高并发。

\*\*Swoole的worker进程有2种类型\*\*

\*\*一种是普通的worker进程，一种是task worker进程。\*\*worker进程是用来处理普通的耗时不是太长的请求；task worker进程用来处理耗时较长的请求，比如数据库的I/O操作。以异步MySQL举例：

\* （1）耗时较长的MySQL查询进入worker；

\* （2）worker通过管道将这个请求交给task worker来处理；

\* （3）worker再去处理其他请求；

\* （4）task worker处理完毕后，处理结果通过管道返回给worker；

\* （5）worker将结果返回给reactor；

\* （6）reactor将结果返回给请求方；

如此，通过worker、task worker结合的方式，就实现了异步I/O。

### Swoole的生命周期

\*\*Cli模式下PHP的基本工作方式\*\*

Swoole运行有个前提条件：\*\*必需在Cli模式下执行\*\*。

Cli下执行一个php文件时的关键步骤：

\* （1）调用每个扩展的MINIT；

\* （2）调用每个扩展的RINIT；

\* （3）执行test.php；

\* （4）调用每个扩展的RSHUTDOWN；

\* （5）调用每个扩展的MSHUTDOWN；

\*\*FPM每个请求都是在执行（2）~（4）步\*\*。

\*\*opcode cache是把第（3）步的词法分析、语法分析、生成opcode代码这几个操作给缓存起来了，从而达到加速的作用\*\*。

### Swoole的生命周期

\*\*Swoole在第（3）步接管了PHP\*\*，进入swoole的生命周期，以多进程模式为例：

\*\*（1）onStart()\*\*

在回调此函数之前Swoole Server已进行了如下操作：

- 创建了manager进程

- 创建了worker子进程

- 监听所有TCP/UDP端口

- 监听了定时器

\*\*此函数是在主进程回调的\*\*，和worker进程的onWorkStart是并行的没有先后之分，在此回调里强烈要求只做log记录，设置进程名操作，不做业务逻辑，否则业务逻辑代码的错误导致master进程crash，让整个swoole server不对对外提供服务了。

\*\*（2）onWorkStart()\*\*

\*\*每个worker或task进程在启动之后，会回调此函数\*\*，由于此回调类似于fpm里的MINIT，所以可以在这里做一个全局的资源加载，框架初始化之类的操作，这样可以对每个请求做全局共享，而达到提升性能的目的。

\*\*（3）onReceive()\*\*

\*\*每个请求到达\*\*（也称数据到达），会回调此函数，然后进行业务逻辑处理，输出结果

\*\*（4）onWorkerStop()\*\*

\*\*worker退出时\*\*，会回调此函数。

\*\*（5）onShutDown()\*\*

\*\*Swoole服务停止\*\*回调此函数，然后继续PHP的Cli工作步骤，进而退出PHP生命周期。

### 使用Swoole时出现mysql server gone away的原因

\*\*MySQL本身是一个多线程的程序\*\*，每个连接过来，会开一个线程去处理相关的query，MySQL会定期回收长时间没有任何query的连接（时间周期受wait\_timeout配置影响），由于\*\*Swoole是一个长驻内存的服务\*\*，建立了一个MySQL的连接，不主动关闭或者是用pconnect（持久连接）的方式，那么这个MySQL连接会一直保存着，然后长时间没有和数据库有交互，就主动被mysql server关闭了，之后继续用这个连接，就报mysql server gone away了。

\*\*解决方法\*\*

\* 修改MySQL的`wait\_timeout`值为一个非常大的值，此方法不太可取，可能会产生大量的sleep连接，导致MySQL连接上限了， 建议不使用。

\* 每次query之前主动进行连接检测

如果是用mysqli，可用内置的mysqli\_ping

```php

<?php

if (!$mysqli->ping()) {

mysqli->connect(); //重连

}

```

如果是pdo，可以检测mysql server的服务器信息来判断

```php

<?php

try {

$pdo->getAttribute(\PDO::ATTR\_SERVER\_INFO);

}catch (\Exception $e) {

if ($e->getCode() == 'HY000') {

$pdo = new PDO(xxx); //重连

} else {

throw $e;

}

}

```

这个方案有个\*\*缺点：额外多一次请求\*\*，所以改进方法: 用一个全局变量存放最后一次query的时间，下一次query的时候先和现在时间对比一下，超过waite\_timeout再重连，或者也可以用swoole\_tick定时检测。

\* 被动检测，每次query用try catch包起来，如有mysql gone away异常，则重新连接，再执行一次当前sql.

```php

<?php

try {

query($sql);

} catch (\Exception $e) {

if ($e->getCode() == 'HY000') {

reconnect(); // 重连

query($sql)

} else {

throw $e;

}

}

```

\* 用`短连接`，务必每次操作完之后，手动close；

## PHP框架与包

---

# Web Server

---

---

## Nginx

---

### Nginx有哪些内置的全局变量？

\* $args 请求中的参数;

\* $content\_length HTTP请求信息里的"Content-Length";

\* $content\_type 请求信息里的"Content-Type";

\* $document\_root 针对当前请求的根路径设置值;

\* $document\_uri 与$uri相同;

\* $host http请求的域名

\* $http\_user\_agent 客户端agent信息;

\* $http\_cookie 客户端cookie信息;

\* $limit\_rate 对连接速率的限制;

\* $request\_body\_file 客户端请求主体信息的临时文件名;

\* $request\_method 请求的方法，比如"GET"、"POST"等;

\* $remote\_addr 客户端地址;

\* $remote\_port 客户端端口号;

\* $remote\_user 客户端用户名，认证用;

\* $request\_filename 当前请求的文件路径名;

\* $request\_body\_file 客户端请求主体的临时文件名;

\* $request\_uri 包含请求参数的原始URI，不包含主机名，如："/foo/bar.php?arg=baz";

\* $query\_string 与$args相同;

\* $scheme 所用的协议，比如http或者是https;

\* $server\_addr 服务器地址，如果没有用listen指明服务器地址，使用这个变量将发起一次系统调用以取得地址(造成资源浪费);

\* $server\_name 请求到达的服务器名;

\* $server\_port 请求到达的服务器端口号;

\* $uri 不带请求参数的当前URI，$uri不包含主机名，如"/foo/bar.html";

\* $fastcgi\_script\_name 这个变量等于一个以斜线结尾的请求URI加上fastcgi\_index给定的参数。可以用这个变量代替SCRIPT\_FILENAME 和PATH\_TRANSLATED，以确定php脚本的名称。

如请求"/info/":

```

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME /home/www/scripts/php$fastcgi\_script\_name;

```

SCRIPT\_FILENAME等于`/home/www/scripts/php/info/index.php`

### Nginx fastcgi\_index配置的作用

语法：

```

fastcgi\_index file

```

如果URI以斜线结尾，文件名将追加到URI后面，这个值将存储在变量`$fastcgi\_script\_name`中。

\*\*例如：\*\*

```

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME /home/www/scripts/php$fastcgi\_script\_name;

```

请求`/page.php`时，SCRIPT\_FILENAME将被设置为`/home/www/scripts/php/page.php`，但是请求`/`则为`/home/www/scripts/php/index.php`。

### Nginx fastcgi\_param配置理解

fastcgi\_param用于定义一些fastcgi模块的环境变量：

```

# 脚本文件请求的路径

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

# 请求的参数，如?app=123

fastcgi\_param QUERY\_STRING $query\_string;

# 请求的动作(GET,POST)

fastcgi\_param REQUEST\_METHOD $request\_method;

# 请求头中的Content-Type字段

fastcgi\_param CONTENT\_TYPE $content\_type;

# 请求头中的Content-length字段

fastcgi\_param CONTENT\_LENGTH $content\_length;

# 脚本名称

fastcgi\_param SCRIPT\_NAME $fastcgi\_script\_name;

# 请求的地址（不带参数）

fastcgi\_param REQUEST\_URI $request\_uri;

# 与$uri相同

fastcgi\_param DOCUMENT\_URI $document\_uri;

# 网站的根目录，在server配置中root指令中指定的值

fastcgi\_param DOCUMENT\_ROOT $document\_root;

# 请求使用的协议，通常是HTTP/1.0或HTTP/1.1

fastcgi\_param SERVER\_PROTOCOL $server\_protocol;

# cgi 版本

fastcgi\_param GATEWAY\_INTERFACE CGI/1.1;

# nginx 版本号，可修改、隐藏

fastcgi\_param SERVER\_SOFTWARE nginx/$nginx\_version;

# 客户端IP

fastcgi\_param REMOTE\_ADDR $remote\_addr;

# 客户端端口

fastcgi\_param REMOTE\_PORT $remote\_port;

# 服务器IP地址

fastcgi\_param SERVER\_ADDR $server\_addr;

# 服务器端口

fastcgi\_param SERVER\_PORT $server\_port;

# 服务器名，域名在server配置中指定的server\_name

fastcgi\_param SERVER\_NAME $server\_name;

# 可自定义变量

# fastcgi\_param PATH\_INFO $path\_info;

# PHP only, required if PHP was built with --enable-force-cgi-redirect

# fastcgi\_param REDIRECT\_STATUS 200;

```

\*\*在php可打印出上面的服务环境变量\*\*

```php

echo $\_SERVER['REMOTE\_ADDR']

```

### Nginx fastcgi\_pass配置理解

指定FastCGI服务器监听端口与地址。

\* 直接使用IP地址和端口号指定

```

fastcgi\_pass localhost:9000;

```

\* 使用Unix Socket指定

```

fastcgi\_pass unix:/tmp/fastcgi.socket;

```

\* 使用upstream指定

```

upstream backend {

server localhost:1234;

}

fastcgi\_pass backend;

```

### Nginx fastcgi\_read\_timeout配置理解

前端FastCGI服务器的响应超时时间，如果有一些直到它们运行完才有输出的长时间运行的FastCGI进程，或者在错误日志中出现前端服务器响应超时错误，可能需要调整这个值。

### Nginx支持的IO模型有哪些？

Nginx支持如下处理连接的方法（I/O复用方法），这些方法可以通过use指令指定：

\* select：如果当前平台没有更有效的方法，它是编译时默认的方法。可以使用配置参数–with-select\_module和–without-select\_module来启用或禁用这个模块。

\* poll：如果当前平台没有更有效的方法，它是编译时默认的方法。可以使用配置参数–with-poll\_module和–without-poll\_module来启用或禁用这个模块。

\* kqueue：高效的方法，使用于FreeBSD 4.1+、 OpenBSD 2.9+、NetBSD 2.0和MacOS X。使用双处理器的MacOS X系统使用kqueue可能会造成内核崩溃。

\* epoll：高效的方法，使用于Linux内核2.6版本及以后的系统。在某些发行版本中，如SuSE 8.2, 有让2.4版本的内核支持epoll的补丁。

\* rtsig：可执行的实时信号，使用于Linux内核版本2.2.19以后的系统。默认情况下整个系统中不能出现大于1024个POSIX实时(排队)信号。这种情况对于高负载的服务器来说是低效的；所以有必要通过调节内核参数 /proc/sys/kernel/rtsig-max来增加队列的大小。可是从Linux内核版本2.6.6-mm2开始， 这个参数就不再使用了，并且对于每个进程有一个独立的信号队列，这个队列的大小可以用RLIMIT\_SIGPENDING 参数调节。当这个队列过于拥塞，nginx就放弃它并且开始使用poll方法来处理连接直到恢复正常。

\* /dev/poll：高效的方法，使用于Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+.

\* eventport：高效的方法，使用于Solaris 10。

在linux下面，只有epoll是高效的方法。

### Nginx与php-fpm配合工作的流程

\* （1）FastCGI进程管理器php-fpm自身初始化，启动主进程php-fpm和启动start\_servers个CGI 子进程。

主进程php-fpm主要是管理fastcgi子进程，监听9000端口。

fastcgi子进程等待来自Web Server的连接。

\*（2）当客户端请求到达Nginx时，Nginx通过location指令，将所有以php为后缀的文件都交给127.0.0.1:9000来处理。

\* （3）FastCGI进程管理器PHP-FPM选择并连接到一个子进程CGI解释器。Web server将CGI环境变量和标准输入发送到FastCGI子进程。

\* （4）FastCGI子进程完成处理后将标准输出和错误信息从同一连接返回Web Server。当FastCGI子进程关闭连接时，请求便告处理完成。

\* （5）FastCGI子进程接着等待并处理来自FastCGI进程管理器（运行在 WebServer中）的下一个连接。

### Nginx的配置理解

nginx配置文件主要分为六个区域：

\* main

控制子进程的所属用户/用户组、派生子进程数、错误日志位置/级别、pid位置、子进程优先级、进程对应cpu、进程能够打开的文件描述符数目等

\* events

控制nginx处理连接的方式

\* http

\* sever

\* location

\* upstream

\*\*实例：\*\*

```

# 运行用户

user www-data;

# 启动进程数,通常设置成和cpu的数量相等

worker\_processes 1;

# 全局错误日志

error\_log /var/log/nginx/error.log;

# PID文件

pid /var/run/nginx.pid;

events {

# 使用epoll多路复用模式

use epoll;

# 单个后台worker process进程的最大并发链接数

worker\_connections 1024;

# multi\_accept on;

}

http {

# 设定mime类型,类型由mime.type文件定义

include /etc/nginx/mime.types;

# 1 octet = 8 bit

default\_type application/octet-stream;

# 设定访问日志

access\_log /var/log/nginx/access.log;

# sendfile指令指定nginx是否调用sendfile函数（zero copy方式）来输出文件，对于普通应用，必须设为on,如果用来进行下载等应用磁盘IO重负载应用，可设置为off，以平衡磁盘与网络I/O处理速度，降低系统的uptime.

sendfile on;

# 在一个数据包里发送所有头文件，而不一个接一个的发送

#tcp\_nopush on;

# 连接超时时间

keepalive\_timeout 65;

# 作用于socket参数TCP\_NODELAY，禁用nagle算法，也即不缓存数据

tcp\_nodelay on;

# 开启gzip压缩

gzip on;

gzip\_disable "MSIE [1-6]\.(?!.\*SV1)";

# 设定请求缓冲

client\_header\_buffer\_size 1k;

large\_client\_header\_buffers 44k;

include /etc/nginx/conf.d/\*.conf;

include /etc/nginx/sites-enabled/\*;

# 设定负载均衡的服务器列表

upstream mysvr {

# weigth参数表示权值，权值越高被分配到的几率越大

# 本机上的Squid开启3128端口

server 192.168.8.1:3128 weight=5;

server 192.168.8.2:80 weight=1;

server 192.168.8.3:80 weight=6;

}

server {

# 侦听80端口

listen 80;

# 定义使用www.xx.com访问

server\_name www.xx.com;

# 设定本虚拟主机的访问日志

access\_log logs/www.xx.com.access.log main;

# 默认请求

location / {

# 定义服务器的默认网站根目录位置

root /root;

# 定义首页索引文件的名称

index index.php index.html index.htm;

fastcgi\_pass localhost:9000;

fastcgi\_param SCRIPT\_FILENAME $document\_root/$fastcgi\_script\_name;

include /etc/nginx/fastcgi\_params;

}

# 定义错误提示页面

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root /root;

}

# 静态文件，nginx自己处理

location ~ ^/(images|javascript|js|css|flash|media|static)/ {

root /var/www/virtual/htdocs;

# 过期时间30天

expires 30d;

}

# PHP脚本请求全部转发到FastCGI处理，使用FastCGI默认配置

location ~ \.php$ {

root /root;

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME /home/www/www$fastcgi\_script\_name;

include fastcgi\_params;

}

# 设定查看Nginx状态的地址

location /NginxStatus {

stub\_status on;

access\_log on;

auth\_basic "NginxStatus";

auth\_basic\_user\_file conf/htpasswd;

}

# 禁止访问 .htxxx 文件

location ~ /\.ht {

deny all;

}

}

}

```

### Nginx反向代理提高性能的理解

对于后端是动态服务来说，比如Java和PHP。这类服务器（如JBoss和PHP-FPM）的IO处理能力往往不高。\*\*Nginx有个好处是它会把Request在读取完整之前buffer住，这样交给后端的就是一个完整的HTTP请求\*\*，从而提高后端的效率，而不是断断续续的传递（互联网上连接速度一般比较慢）。同样，Nginx\*\*也可以把response给buffer住，同样也是减轻后端的压力\*\*。

### Nginx rewrite与Location的比较

\*\*rewrite\*\*

使用Nginx提供的全局变量或自己设置的变量，结合正则表达式和标志位\*\*实现url重写以及重定向\*\*。rewrite只能放在server{},location{},if{}中，并且\*\*只能对域名后边的除去传递的参数外的字符串起作用\*\*。如，`http://demo.com/a/we/index.php?id=1&u=str` 只对`/a/we/index.php`重写。

如果想对域名或参数字符串起作用，可以使用全局变量匹配，也可以使用proxy\_pass反向代理。

\*\*Rewrite标志位\*\*

\* last 相当于Apache的[L]标记，表示完成rewrite

\* break 停止执行当前虚拟主机的后续rewrite指令集

\* redirect 返回302临时重定向，地址栏会显示跳转后的地址

\* permanent 返回301永久重定向，地址栏会显示跳转后的地址

\*\*Rewrite实例\*\*

```

// 应用于Server

server {

listen 80;

server\_name demo.com;

index index.html index.php;

root html;

if ($http\_host !~ “^star\.igrow\.cn$&quot {

rewrite ^(.\*) http://star.igrow.cn$1 redirect;

}

}

// 防盗链

location ~\* \.(gif|jpg|swf)$ {

valid\_referers none blocked start.igrow.cn sta.igrow.cn;

if ($invalid\_referer) {

rewrite ^/ http://$host/logo.png;

}

}

// 根据文件类型设置过期时间

location ~\* \.(js|css|jpg|jpeg|gif|png|swf)$ {

if (-f $request\_filename) {

expires 1h;

break;

}

}

// 禁止访问某个目录

location ~\* \.(txt|doc)${

root /data/www/wwwroot/linuxtone/test;

deny all;

}

```

\*\*rewrite和location的比较\*\*

rewrite和location都能实现跳转，主要区别在于\*\*rewrite是在同一域名内更改获取资源的路径，而location是对一类路径做控制访问或反向代理，可以proxy\_pass到其他机器\*\*。很多情况下rewrite也会写在location里，它们的执行顺序是：

\* （1）执行server块的rewrite指令

\* （2）执行location匹配

\* （3）执行选定的location中的rewrite指令

如果其中某步URI被重写，则重新循环执行（1）~（3），直到找到真实存在的文件；循环超过10次，则返回500 Internal Server Error错误。

正则匹配会覆盖普通匹配，\*\*location的执行逻辑跟location的编辑顺序无关\*\*。

\*\*location语法格式\*\*：

```

location [=|~|~\*|^~|@] /uri/ { … }

```

\* = 表示精确匹配

\* ~ 区分大小写匹配

\* ~\* 不区分大小写匹配

\* !~ 区分大小写不匹配

\* !~\* 不区分大小写不匹配

\* ^ 以什么开头的匹配

\* $ 以什么结尾的匹配

\* ^~ 表示uri以某个常规字符串开头，不是正则匹配，优先级高于正则

\* / 通用匹配,如果没有其它匹配,任何请求都会匹配到

\* 代表任意字符

\* . 匹配除换行符以外的任意字符

\* ? 重复0次或1次

\* \+ 重复1次或更多次

\* \\* 重复0次或更多次

\* \d 匹配数字

\* {n}重复n次

\* {n,}重复n次或更多次

\* [c]匹配单个字符c

\* [a-z]匹配a-z小写字母的任意一个

\* \转义字符

\* -f和!-f判断是否存在文件

\* -d和!-d判断是否存在目录

\* -e和!-e判断是否存在文件或目录

\* -x和!-x判断文件是否可执行

\*\*实际使用中一般至少有三个匹配规则定义：\*\*

\* 直接匹配网站根，通过域名访问网站首页比较频繁，使用这个会加速处理。这里是直接转发给后端应用服务器了，也可以是一个静态首页。

```

location = / {

proxy\_pass http://tomcat:8080/index

}

```

\* 处理静态文件的请求，这是nginx作为http服务器的强项。有两种配置模式，目录匹配或后缀匹配,任选其一或搭配使用。

```

location ^~ /static/ {

root /webroot/static/;

}

location ~\* \.(gif|jpg|jpeg|png|css|js|ico)$ {

root /webroot/res/;

}

```

\* 通用规则，用来转发动态请求到后端应用服务器非静态文件请求就默认是动态请求

```

location / {

proxy\_pass http://tomcat:8080/

}

```

### 如何使用Nginx实现负载均衡和反向代理？

设定http服务器，利用它的反向代理功能提供负载均衡支持

```

http {

# 设定mime类型,类型由mime.type文件定义

include /etc/nginx/mime.types;

default\_type application/octet-stream;

# 设定日志格式

access\_log /var/log/nginx/access.log;

# 其他配置，略

# 设定负载均衡的服务器列表

upstream mysvr {

# weigth参数表示权值，权值越高被分配到的几率越大

# 本机上的Squid开启3128端口

server 192.168.8.1x:3128 weight=5;

server 192.168.8.2x:80 weight=1;

server 192.168.8.3x:80 weight=6;

}

upstream mysvr2 {

# weigth参数表示权值，权值越高被分配到的几率越大

server 192.168.8.x:80 weight=1;

server 192.168.8.x:80 weight=6;

}

# 第一个虚拟服务器

server {

# 侦听192.168.8.x的80端口

listen 80;

server\_name 192.168.8.x;

# 对aspx后缀的进行负载均衡请求

# 定义服务器的默认网站根目录位置

location ~ .\*\.aspx$ {

# 定义服务器的默认网站根目录位置

root /root;

# 定义首页索引文件的名称

index index.php index.html index.htm;

# 请求转向mysvr定义的服务器列表

proxy\_pass http://mysvr ;

# 反向代理的配置

proxy\_redirect off;

# 后端的Web服务器可以通过X-Forwarded-For获取用户真实IP

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

# 允许客户端请求的最大单文件字节数

client\_max\_body\_size 10m;

# 缓冲区代理缓冲用户端请求的最大字节数

client\_body\_buffer\_size 128k;

# nginx跟后端服务器连接超时时间(代理连接超时)

proxy\_connect\_timeout 90;

# 后端服务器数据回传时间(代理发送超时)

proxy\_send\_timeout 90;

# 连接成功后，后端服务器响应时间(代理接收超时)

proxy\_read\_timeout 90;

# 设置代理服务器（nginx）保存用户头信息的缓冲区大小

proxy\_buffer\_size 4k;

# proxy\_buffers缓冲区，网页平均在32k以下的话，这样设置

proxy\_buffers 4 32k;

# 高负荷下缓冲大小（proxy\_buffers\*2）

proxy\_busy\_buffers\_size 64k;

# 设定缓存文件夹大小，大于这个值，将从upstream服务器传

proxy\_temp\_file\_write\_size 64k;

}

}

}

```

### 服务器出现大量TIME\_WAIT和CLOSE\_WAIT的可能原因是什么？

\*\*查看当前服务器网络连接状态\*\*

```

netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

```

输出：

```

TIME\_WAIT 814

CLOSE\_WAIT 1

FIN\_WAIT1 1

ESTABLISHED 634

SYN\_RECV 2

LAST\_ACK 1

```

常用的三个状态是：

\* ESTABLISHED 表示正在通信

\* TIME\_WAIT 表示主动关闭

\* CLOSE\_WAIT 表示被动关闭

![image](/images/tech/net\_12.png)

如果服务器出了异常，百分之八九十都是下面两种情况：

\* 服务器保持了大量TIME\_WAIT状态

\* 服务器保持了大量CLOSE\_WAIT状态

因为Linux分配给一个用户的文件句柄是有限的，而TIME\_WAIT和CLOSE\_WAIT两种状态如果一直被保持，那么意味着对应数目的通道就一直被占着，一旦达到句柄数上限，新的请求就无法被处理了，接着就是大量`Too Many Open Files`异常。

\*\*服务器保持了大量TIME\_WAIT状态的原因\*\*

TIME\_WAIT是主动关闭连接的一方保持的状态，对于爬虫服务器来说他本身就是客户端，在完成一个爬取任务之后，就会发起主动关闭连接，从而进入TIME\_WAIT的状态，然后在保持这个状态2MSL（max segment lifetime）时间之后，彻底关闭回收资源。

而对于HTTP的交互跟上面画的那个图是不一样的，\*\*关闭连接的不是客户端，而是服务器\*\*，所以web服务器也是会出现大量的TIME\_WAIT的情况的。解决思路很简单，就是让服务器能够快速回收和重用那些TIME\_WAIT的资源，通过修改`/etc/sysctl.conf`文件实现：

```

net.ipv4.tcp\_tw\_reuse = 1 # 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭

net.ipv4.tcp\_tw\_recycle = 1 # 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭

```

\*\*服务器保持了大量CLOSE\_WAIT状态的原因\*\*

TIME\_WAIT状态可以通过优化服务器参数得到解决，因为发生TIME\_WAIT的情况是服务器自己可控的，要么就是对方连接的异常，要么就是自己没有迅速回收资源，总之不是由于自己程序错误导致的。

如果一直保持在CLOSE\_WAIT状态，那么只有一种情况，就是在\*\*对方关闭连接之后服务器程序自己没有进一步发出ack信号\*\*。换句话说，就是在对方连接关闭之后，程序里没有检测到，或者程序压根就忘记了这个时候需要关闭连接，于是这个资源就一直被程序占着。

如：服务器A是一台爬虫服务器，它使用简单的HttpClient去请求资源服务器B上面的apache获取文件资源，正常情况下，如果请求成功，那么在抓取完资源后，服务器A会主动发出关闭连接的请求，这个时候就是主动关闭连接，服务器A的连接状态我们可以看到是TIME\_WAIT。如果一旦发生异常呢？假设 请求的资源服务器B上并不存在，那么这个时候就会由服务器B发出关闭连接的请求，服务器A就是被动的关闭了连接，如果服务器A被动关闭连接之后程序员忘了 让HttpClient释放连接，那就会造成CLOSE\_WAIT的状态了。

### Apache与Nginx的I/O模型差异

\*\*Apache处理一个请求是同步阻塞的模式\*\*：每到达一个请求，Apache都会去fork一个子进程去处理这个请求，直到这个请求处理完毕。

\*\*Nginx是基于epoll的异步非阻塞的服务器程序\*\*。

## Apache

---

# 数据存储

---

---

## MySQL

---

### MySQL语句的执行顺序

MySQL的语句执行一共分为11步，最先执行的总是FROM操作，最后执行的是LIMIT操作。其中\*\*每一个操作都会产生一张虚拟的表\*\*，这个虚拟的表作为一个处理的输入，只是这些虚拟的表对用户来说是透明的，但是\*\*只有最后一个虚拟的表才会被作为结果返回\*\*。如果没有在语句中指定某一个子句，那么将会跳过相应的步骤。

\* `FORM`：对FROM的左边的表和右边的表计算笛卡尔积，产生虚表VT1。

\* `ON`：对虚表VT1进行ON筛选，只有那些符合<join-condition>的行才会被记录在虚表VT2中。

\* `JOIN`：如果指定了OUTER JOIN（比如left join、 right join），那么保留表中未匹配的行就会作为外部行添加到虚拟表VT2中，产生虚拟表VT3, 如果from子句中包含两个以上的表的话，那么就会对上一个join连接产生的结果VT3和下一个表重复执行步骤1~3这三个步骤，一直到处理完所有的表为止。

\* `WHERE`：对虚拟表VT3进行WHERE条件过滤。只有符合<where-condition>的记录才会被插入到虚拟表VT4中。

\* `GROUP BY`：根据group by子句中的列，对VT4中的记录进行分组操作，产生VT5。

\* `CUBE|ROLLUP`：对表VT5进行cube或者rollup操作，产生表VT6。

\* `HAVING`：对虚拟表VT6应用having过滤，只有符合<having-condition>的记录才会被 插入到虚拟表VT7中。

\* `SELECT`：执行select操作，选择指定的列，插入到虚拟表VT8中。

\* `DISTINCT`：对VT8中的记录进行去重。产生虚拟表VT9。

\* `ORDER BY`：将虚拟表VT9中的记录按照<order\_by\_list>进行排序操作，产生虚拟表VT10。

\* `LIMIT`：取出指定行的记录，产生虚拟表VT11, 并将结果返回。

### SQL\_CALC\_FOUND\_ROWS

在很多分页的程序中都这样写:

```

SELECT COUNT(\*) from 'table' WHERE ......; # 查出符合条件的记录总数

SELECT \* FROM 'table' WHERE ...... LIMIT M,N; # 查询当页要显示的数据

```

这样的语句可以改成:

```

SELECT SQL\_CALC\_FOUND\_ROWS \* FROM 'table' WHERE ...... LIMIT M, N;

SELECT FOUND\_ROWS();

```

这样\*\*只要执行一次较耗时的复杂查询\*\*可以同时得到\*\*与不带LIMIT同样的记录条数\*\*：第二个SELECT返回一个数字，指示了在没有LIMIT子句的情况下，第一个SELECT返回了多少行。

### 两大类触发器

\*\*DML触发器\*\*

是\*\*基于表而创建的\*\*，可以在一张表创建多个DML触发器。其特点是定义在表或者视图上、\*\*自动触发、不能被直接调用\*\*。用户可以针对INSERT、UPDATE、DELETE语句分别设置触发器，也可以针对一张表上的特定操作设置。触发器可以容纳非常复杂的SQL语句，但不管操作多么复杂，也只能作为一个独立的单元被执行、看作一个事务。如果在执行触发器的过程中发生了错误，则整个事务都会回滚。

\*\*DDL触发器\*\*

是一种特殊的触发器，它在响应数据定义语言(DDL)语句时触发。可以用于在数据库中执行管理任务，例如审核以及规范数据库操作。

### 主键与唯一索引的区别

\* UNIQUE KEY可空，PRIMARY KEY 不可空不可重复；

\* UNIQUE KEY可以在一个表里的一个或多个字段定义（多个UNIQUE KEY可以同时存在），在一个表中只能有一个PRIMARY KEY；

\* PRIMARY KEY一般在逻辑设计中\*\*用作记录标识\*\*，这也是设置PRIMARY KEY的本来用意，而UNIQUE KEY只是为了保证域/域组的唯一性。

​

### InnDB行级锁理解

如果是InnoDB引擎，就可以在事务里使用行锁，比如使用FOR UPDATE：

```

SELECT xx FROM xx [FORCE INDEX(PRIMARY)] WHERE xx FOR UPDATE

```

被加锁的行，其他事务也能读取但如果想写的话就必须等待锁的释放（\*\*乐观锁\*\*）。

InnoDB的行锁是针对索引加的锁，不是针对记录加的锁，\*\*只有查询能够使用索引时才可以使用行级锁\*\*。不论是使用主键索引、唯一索引或普通索引，InnoDB都会使用行锁来对数据加锁。

### 冷备份和热备份比较

\*\*冷备份\*\*

发生在\*\*数据库已经正常关闭\*\*的情况下，当正常关闭时会提供给我们一个完整的数据库。\*\*冷备份是将关键性文件拷贝到另外位置的一种说法\*\*。对于备份数据库信息而言，冷备份是最快和最安全的方法。

\*\*优点：\*\*

\* 是非常快速的备份方法（只需拷贝文件）

\* 容易归档（简单拷贝即可）

\* 容易恢复到某个时间点上（只需将文件再拷贝回去）

\* 能与归档方法相结合，做数据库最新状态的恢复。

\* 低度维护，高度安全。

\*\*缺点： \*\*

\* 单独使用时，只能提供到某一时间点上的恢复

\* 在实施备份的全过程中，数据库必须要作备份而不能作其它工作。也就是说，在冷备份过程中，数据库必须是关闭状态

\* 若磁盘空间有限，只能拷贝到磁带等其它外部存储设备上，速度会很慢

\* 不能按表或按用户恢复

值得注意的是冷备份必须在数据库关闭的情况下进行，当数据库处于打开状态时，执行数据库文件系统备份是无效的 。而且在恢复后一定要把数据库文件的属组和属主改为mysql。

\*\*热备份\*\*

\*\*在数据库运行的情况下\*\*，备份数据库操作的SQL语句，当数据库发生问题时，可以重新执行一遍备份的SQL语句。

\*\*优点：\*\*

\* 可在表空间或数据文件级备份，备份时间短。

\* 备份时数据库仍可使用。

\* 可达到秒级恢复（恢复到某一时间点上）。

\* 可对几乎所有数据库实体作恢复。

\* 恢复是快速的，在大多数情况下在数据库仍工作时恢复。

\*\*缺点：\*\*

\* 不能出错，否则后果严重。

\* 若热备份不成功，所得结果不可用于时间点的恢复。

\* 因难于维护，所以要特别仔细小心，不允许以失败而告终。

MySQL原生支持多机热备。

### 数据库连接池理解

由于\*\*创建连接和释放连接都有很大的开销\*\*（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，再加上网络延迟，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以\*\*事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销\*\*，这是典型的用\*\*空间换取时间\*\*的策略。

\*\*连接池仅在超大型应用中才有价值\*\*。普通的应用采用MySQL长连接方案，每个php-fpm创建一个MySQL连接，每台机器开启100个php-fpm进程。如果有10台机器，每台机器并发的请求为100。实际上只需要创建1000个MySQL连接就能满足需求，数据库的压力并不大。即使有100台机器，硬件配置好的存储服务器依然可以承受。

达到数百或者数千台应用服务器时，MySQL服务器就需要维持十万级的连接。这时数据库的压力就会非常大了。连接池技术就可以派上用场了，可以大大降低数据库连接数。

基于swoole的AsyncTask模块实现的连接池是完美方案，编程简单，没有数据同步和锁的问题。甚至可以多个服务共享连接池。缺点是灵活性不如多线程连接池，无法动态增减连接，且有一次进程间通信的开销。

node.js/ngx\_lua等在多进程的模式下，无法开发出真正的连接池，除非也像swoole\_task这样来实现。

### 外联接

LEFT OUTER JOIN或LEFT JOIN \ RIGHT OUTER JOIN或RIGHT JOIN \ FULL OUTER JOIN或FULL JOIN。

左外联接的结果集中除了包括满足条件的行外，还包括\*\*左表所有的行\*\*，左表中没有满足条件的以空值的形式出现。

### 为SELECT语句添加一个自动增加的列

```

set @N = 0;

SELECT @N := @N +1 AS number, name, surname FROM gbtags\_users;

```

### 分析MySQL语句执行时间和消耗资源

```

SET profiling=1;  # 启动profiles，默认是没开启的

SELECT \* FROM customers; # 执行要分析的SQL语句

SHOW profiles; # 查看SQL语句具体执行步骤及耗时

SHOW profile cpu,block io FOR QUERY 41; # 查看ID为41的查询在各个环节的耗时和资源消耗

```

### 使用EXPLAIN分析MySQL语句的执行情况

```

mysql> explain select \* from t\_online\_group\_records where UNIX\_TIMESTAMP(gre\_updatetime) > 123456789;

+----+-------------+------------------------+------+---------------+------+---------+------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+------------------------+------+---------------+------+---------+------+------+-------------+

| 1 | SIMPLE | t\_online\_group\_records | ALL | NULL | NULL | NULL | NULL | 47 | Using where |

+----+-------------+------------------------+------+---------------+------+---------+------+------+-------------+

1 row in set (0.00 sec)

```

重点关注type，rows和Extra。

\*\*type\*\*

操作的类型，可以用来\*\*判断有无使用到索引\*\*。结果值从好到坏：

```

... > RANGE(使用到索引) > INDEX > ALL(全表扫描)

```

\*\*一般查询应达到range级别\*\*，具体可能值如下：

\* SYSTEM ：CONST的特例，当表上只有一条记录时

\* CONST ：WHERE条件筛选后表上至多有一条记录匹配时，比如`WHERE ID = 2`

\* EQ\_REF ：参与连接运算的表是内表（两表连接时作为循环中的内循环遍历的对象，这样的表称为内表）。基于索引（连接字段上存在唯一索引或者主键索引，且操作符必须是“=”谓词，索引值不能为NULL）做扫描，使得\*\*对外表的一条元组，内表只有唯一一条元组与之对应\*\*。

\* REF ：可以用于单表扫描或者连接。参与连接运算的表，是内表。

基于索引（连接字段上的索引是非唯一索引，操作符必须是“=”谓词，连接字段值不可为NULL）做扫描，使得对外表的一条元组，内表可有若干条元组与之对应。

\* REF\_OR\_NULL ：类似REF，只是搜索条件包括：连接字段的值可以为NULL的情况，比如 where col = 2 or col is null

\* RANGE ：\*\*范围扫描\*\*，基于索引做范围扫描，为诸如BETWEEN、IN、>=、LIKE类操作提供支持

\* INDEX\_SCAN ：索引做扫描，是基于索引在索引的叶子节点上找满足条件的数据（不需要访问数据文件）

\* ALL ：`全表扫描`或者范围扫描：不使用索引，顺序扫描，直接读取表上的数据（访问数据文件）

\* UNIQUE\_SUBQUERY ：在子查询中，基于唯一索引进行扫描，类似于EQ\_REF

\* INDEX\_SUBQUERY ：在子查询中，基于除唯一索引之外的索引进行扫描

\* INDEX\_MERGE ：多重范围扫描。两表连接的每个表的连接字段上均有索引存在且索引有序，结果合并在一起。适用于作集合的并、交操作。

\* FT ：FULL TEXT，`全文检索`

\*\*rows\*\*

SQL执行检查的记录数

\*\*Extra\*\*

SQL执行的附加信息，如\*\*Using index表示查询只用到索引列\*\*，不需要去读表等。

### CASE…WHEN…THEN

使用CASE来重新定义数值类型

```

SELECT id,title,(CASE date WHEN '0000-00-00' THEN '' ELSE date END) AS date

FROM your\_table

SELECT id,title,

(CASE status WHEN 0 THEN 'open' WHEN 1 THEN 'close' ELSE 'standby' END) AS status

FROM your\_table

```

### 通俗地理解三个范式

\* 第一范式：是\*\*对属性的原子性约束\*\*，要求属性具有原子性，不可再分解；

\* 第二范式：是\*\*对记录的惟一性约束\*\*，要求记录有惟一标识，即实体的惟一性；

\* 第三范式：是\*\*对字段冗余性的约束\*\*，即任何字段不能由其他字段派生出来，它要求字段没有冗余（没有冗余的数据库未必是最好的数据库）。

### MySQL基本备份与恢复操作

\*\*导出数据\*\*

```

# mysqldump -u 用户名 -p 数据库名 [表名] > 导出的文件名

mysqldump -uroot -p test mytable > mytable.20140921.bak.sql

```

### 导出备份数据之后发送的写操作

\*\*先使用mysqlbinlog导出这部分写操作SQL\*\*(基于时间点或位置)

```

# 导出2014-09-21 09:59:59之后的binlog

mysqlbinlog --database="test" --start-date="2014-09-21 09:59:59" /var/lib/mysql/mybinlog.000001 > binlog.data.sql

# 导出起始id为123456之后的binlog：

mysqlbinlog --database="test" --start-position="123456" /var/lib/mysql/mybinlog.000001 > binlog.data.sql

```

\*\*导入备份数据\*\*

```

mysql -uroot -p test < mytable.20140921.bak.sql

```

\*\*导入binlog\*\*

```

mysql -uroot -p test < binlog.data.sql

```

### 存储过程的概念以及优缺点理解

存储过程是一套已经预先编译好的SQL代码，是SQL语句和可选控制语句的集合及一个独立的数据库对象。存储过程在数据库内可以由应用程序调用执行，而且允许用户声明变量、有条件执行以及其他强大的编程功能。由于存储过程是已经编译好的代码，所以执行的时候不需要分析也不需要再次编译，能够提高程序的运行效率。

存储过程可以包含程序流、逻辑以及对数据库的查询。可以接受参数、输出参数、返回单个或者多个结果集以及返回值。

\*\*带简单参数的存储过程\*\*

```sql

/\*带学号参数的存储过程\*/

CREATE PROCEDURE s

@id int/\*参数\*/

AS

SELECT \* FROM student

WHERE id=@id

GO

```

```

/\*输入参数2001002的学生号，查询学号2001002学生的信息\*/

s 2001002

GO

```

\*\*优点\*\*

\* 存储过程可以用于\*\*降低网络流量\*\*，存储过程代码直接存储于数据库中，所以不会产生大量T-sql语句的代码流量。

\* 通过向用户授予对存储过程（而不是基于表）的访问权限，它们可以提供对特定数据的访问

\*\*缺点\*\*

\* 如果更改范围大到需要对输入存储过程的参数进行更改，或者要更改由其返回的数据，则仍需要更新程序集中的代码以添加参数、更新 GetValue() 调用，等等，比较繁琐。

\* 可移植性差：由于存储过程将应用程序绑定到 SQL Server，因此使用存储过程封装业务逻辑将限制应用程序的可移植性。

\* 很多存储过程不支持面向对象的设计，无法采用面向对象的方式将业务逻辑进行封装，从而无法形成通用的可支持复用的业务逻辑框架。

​

### ON DUPLICATE KEY UPDATE

```

# VALUES用来取插入的值，存在主键冲突时就更新，没有删除操作

INSERT INTO ... ON DUPLICATE KEY UPDATE col=VALUES(col)

```

例：更新统计表

```

select \* from player\_count where player\_id = 1; # 查询统计表中是否有记录

insert into player\_count(player\_id,count) value(1,1); # 没有记录就执行insert 操作

update player\_count set count = count+1 where player\_id = 1; # 有记录就执行update操作

```

用ON DUPLICATE KEY UPDATE的做法如下：

```

insert into player\_count(player\_id,count) value(1,1) on duplicate key update count=count+1;

```

### 慢查询日志分析

\*\*慢查询日志格式\*\*

```

User@Host: edu\_online[edu\_online] @ [10.139.10.167]

Query\_time: 1.958000 Lock\_time: 0.000021 Rows\_sent: 254786 Rows\_examined: 254786

SET timestamp=1410883292;

select \* from t\_online\_group\_records;

```

日志显示该查询用了1.958秒，返回254786行记录，一共遍历了254786行记录。及具体的时间戳和SQL语句。

### 使用mysqldumpslow进行慢查询日志分析

输入：

```

mysqldumpslow -s t -t 5 slow\_log\_20140819.txt

```

-s：排序方法，t表示按时间（此外，c为按次数，r为按返回记录数等）

-t：取Top多少条，-t 5表示取前5条

输出：

```

Count: 1076100 Time=0.09s (99065s) Lock=0.00s (76s) Rows=408.9 (440058825), edu\_online[edu\_online]@28hosts

select \* from t\_online\_group\_records where UNIX\_TIMESTAMP(gre\_updatetime) > N

Count: 1076099 Time=0.05s (52340s) Lock=0.00s (91s) Rows=62.6 (67324907), edu\_online[edu\_online]@28hosts

select \* from t\_online\_course where UNIX\_TIMESTAMP(c\_updatetime) > N

Count: 63889 Time=0.78s (49607s) Lock=0.00s (3s) Rows=0.0 (18), edu\_online[edu\_online]@[10.213.170.137]

select f\_uin from t\_online\_student\_contact where f\_modify\_time > N

...

```

以第1条为例，表示这类SQL（N可以取很多值，这里mysqldumpslow会归并起来）在8月19号的慢查询日志内出现了1076100次，总耗时99065秒，总返回440058825行记录，有28个客户端IP用到。

通过慢查询日志分析，就可以找到最耗时的SQL，然后进行具体的SQL分析了

\*\*慢查询相关的配置参数\*\*

```

log\_slow\_queries # 是否打开慢查询日志，得先确保=ON后面才有得分析

long\_query\_time # 查询时间大于多少秒的SQL被当做是慢查询，一般设为1S

log\_queries\_not\_using\_indexes # 是否将没有使用索引的记录写入慢查询日志

slow\_query\_log\_file # 慢查询日志存放路径

```

### MySQL性能优化技巧小结

\* 优化MySQL查询语句，使其使用查询缓存

对于相同的查询MySQL引擎会使用缓存，但是如果在SQL语句中使用函数，如NOW()、RAND()、 CURDATE()等等，则拼凑出的查询不会被认为是相同的查询。

```php

// 查询缓存不开启

$r = mysql\_query("SELECT username FROM user WHERE signup\_date >= CURDATE()");

// 开启查询缓存

$today = date("Y-m-d");

$r = mysql\_query("SELECT username FROM user WHERE signup\_date >= $today");

```

\* 当只要一行数据时使用LIMIT 1

这样MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

\* 为搜索字段建索引

索引并不一定就是给主键或是唯一的字段。如果在表中有某个字段总要会经常用来做搜索，那么就为其建立索引。

\* \*\*在JOIN表的时候使用相同类型的列，并将其索引\*\*

对于那些STRING类型，还需要有相同的字符集才行（两个表的字符集有可能不一样）

\* 千万不要ORDER BY RAND()

You cannot use a column with RAND() values in an ORDER BY clause, because ORDER BY

would evaluate the column multiple times.

当记录数据过多时，会非常慢。

\* 避免SELECT \*

\* 使用ENUM而不是VARCHAR

ENUM实际保存的是TINYINT，但其外表上显示为字符串。如果有一个字段的取值是有限而且固定的，那么，应该使用ENUM而不是VARCHAR。

\* \*\*尽可能的使用NOT NULL\*\*

“NULL columns require additional space in the row to record whether their values are NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte.”

NULL值需要额外的存储空间，而且在比较时也需要额外的逻辑。

\* 把IP地址存成UNSIGNED INT，而不是VARCHAR(15)

\* \*\*固定长度的表会更快\*\*

如果表中的所有字段都是固定长度的，整个表会被认为是`static`或`fixed-length`。

\* 垂直分割

垂直分割是一种把数据库中的大表按列变成几张小表的方法，这样可以降低表的规模、方便使用缓存。

需要注意的是，这些被分出去的字段所形成的表，不应该会被经常性地去JOIN，否则性能会比不分割时还要差很多。

\* 拆分大的DELETE或INSERT语句

因为这两个操作是会锁表的。如果有一个大的处理，一定把其拆分，使用LIMIT条件是一个好的方法。下面是一个示例：

```php

while (1) {

// 每次只做1000条

mysql\_query("DELETE FROM logs WHERE log\_date <= '2009-11-01' LIMIT 1000");

if (mysql\_affected\_rows() == 0) {

// 没得可删了，退出！

break;

}

// 每次都要休息一会儿

usleep(50000);

}

```

\* 越小的列会越快

如使用TINYINT而不是INT，使用DATE而不是DATETIME。

\* 选择一个正确的存储引擎

MyISAM对于SELECT COUNT(\*)这类的计算非常快，但是不支持行锁（写操作会锁表），也不支持事务。

InnoDB的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比MyISAM还慢。

\* 持久链接

持久链接的目的是用来减少重新创建MySQL链接的次数。当一个链接被创建了，它会永远处在连接的状态，就算是数据库操作已经结束了。自从Apache开始重用它的子进程后下一次的HTTP请求会重用Apache的子进程，并重用相同的MySQL链接。

\* 尽量早做过滤，使JOIN或者UNION等后续操作的数据量尽量小。

\* 把能在逻辑层算的提到逻辑层来处理，如一些数据排序、时间函数计算等。

### 查看MySQL编码设置

```

SHOW VARIABLES LIKE 'character\_set\_%';

Variable\_name Value

------------------------ ----------------------------------

character\_set\_client utf8mb4

character\_set\_connection utf8mb4

character\_set\_database utf8mb4

character\_set\_filesystem binary

character\_set\_results utf8mb4

character\_set\_server utf8mb4

character\_set\_system utf8

character\_sets\_dir /usr/local/mysql/share/charsets/

```

### MySQL覆盖索引理解

对于

```

SELECT a FROM … WHERE b = …

```

这种查询，通常的做法是在b字段上建立索引，执行查询时系统会查询b索引进行定位，然后再利用此定位去表里查询需要的数据a。即该过程存在两次查询，一次是查询索引，一次是查询表。

使用Covering Index可以只查询一次索引就完成。建立一个组合索引`b,a`，当查询时，通过组合索引的b部分去定位，至于需要的数据a，立刻就可以在索引里得到，从而\*\*省略了表查询的过程\*\*。

如果使用Covering Index，要\*\*注意SELECT的方式，只SELECT必要的字段\*\*，而不能SELECT \*，因为不太可能把所有的字段一起做索引。

可以使用EXPLAIN命令来确认是否使用了组合索引：\*\*如果在Extra里出现`Using Index`，就说明使用的是Covering Index\*\*。

\*\*实例1：\*\*

```

SELECT COUNT(\*) FROM articles WHERE category\_id = …

```

当在category\_id建立索引后，这个查询使用的就是Covering Index（即，只查索引，而没有查表）。

\*\*实例2：\*\*

比如说在文章系统里分页显示的时候，一般的查询是这样的：

```

SELECT id, title, content FROM article ORDER BY created DESC LIMIT 10000, 10;

```

通常这样的查询会把索引建在created字段（其中id是主键），不过当LIMIT偏移很大时，查询效率仍然很低，改变一下查询：

```

SELECT id, title, content FROM article

INNER JOIN (

SELECT id FROM article ORDER BY created DESC LIMIT 10000, 10

) AS page USING(id)

```

此时，建立复合索引`created, id`就可以在子查询里利用上Covering Index，快速定位id。

### 视图理解

也被称为\*\*虚拟的表\*\*，其内容由SELECT查询语句定义。同真实的表一样，视图包含了一系列带有名称的列和行的数据。但是，\*\*视图并不在数据库中以存储的数据集合形式存在\*\*。用行和列的数据，来自由定义视图的查询所引用的表，并且在引用视图时动态生成。

视图一经定义，便存储在数据库中，\*\*与其相对应的数据并没有像表那样又在数据库中再存储一份\*\*。通过视图看到的数据只是存放在基表中的数据。对视图的操作与对表的操作一样，可以查询、修改、删除。\*\*通过对视图看到的数据进行修改时，相应的基表的数据也要发生变化，同时，若基表的数据发生变化，这种变化也可以自动地反映到视图中\*\*。

视图和查询最主要的差别是：视图的存储是作为数据库开发者设计数据库的一部分；而查询仅仅是对表的查询并非数据库设计的一部分。

### B树理解

B树是对二叉查找树的改进。它的设计思想是，将相关数据尽量集中在一起，以便一次读取多个数据，减少硬盘操作次数。

\*\*特点如下：\*\*

\* 一个节点可以容纳多个值

\* 除非数据已经填满，否则不会增加新的层。也就是说，\*\*B树追求"层"越少越好\*\*。

\* 子节点中的值，与父节点中的值，有严格的大小对应关系。一般来说，如果父节点有a个值，那么就有a+1个子节点。

这种数据结构，非常有利于减少读取硬盘的次数。\*\*假定一个节点可以容纳100个值，那么3层的B树可以容纳100万个数据，如果换成二叉查找树，则需要20层\*\*。假定操作系统一次读取一个节点，并且根节点保留在内存中，那么B树在100万个数据中查找目标值，只需要读取两次硬盘。

数据库以B树格式储存，只解决了按照"主键"查找数据的问题。如果想查找其他字段，就需要建立索引（index）。所谓索引，就是以某个字段为关键字的B树文件（这里仅指基于B树的索引）。

### MySQL GROUP BY注意点

\* 在SELECT指定的字段要么就要包含在GROUP BY语句的后面，作为分组的依据；要么就要被包含在聚合函数中

\* HAVING子句的作用是筛选满足条件的组，即\*\*在分组之后过滤数据\*\*

​

### innodb\_buffer\_pool\_size

innodb\_buffer\_pool\_size这个参数主要作用是设置缓存innodb表的索引、数据、插入数据时的缓冲的缓存区大小。

默认值：128M，操作系统内存的70%-80%最佳。

此外，这个参数是非动态的，要修改这个值，\*\*需要重启mysqld服务\*\*。

如果因为内存不够，MySQL无法启动，就会在错误日志中出现如下报错：

```

InnoDB: mmap(137363456 bytes) failed; errno 12

```

### LIMIT语句理解

返回不多于5行（小于等于）

```

SELECT prod\_name

FROM products

LIMIT 5;

```

返回从第6行开始的5行（行号从0开始）

```

SELECT prod\_name

FROM products

LIMIT 5,5;

```

返回从第6行开始的5行（LIMIT的一种替代语法）

```

SELECT prod\_name

FROM products

LIMIT 5 OFFSET 5;

```

### MyISAM和InnoDB的比较

\* MySQL默认采用的是MyISAM。

\* \*\*MyISAM不支持事务\*\*，而InnoDB支持。\*\*InnoDB的AUTOCOMMIT默认是打开的\*\*，即每条SQL语句会默认被封装成一个事务，自动提交，这样会影响速度，所以最好是把多条SQL语句显示放在begin和commit之间，组成一个事务去提交。

\* InnoDB支持数据行级锁，\*\*MyISAM不支持行锁定\*\*，只支持锁定整个表。即MyISAM同一个表上的读锁和写锁是互斥的，MyISAM并发读写时如果等待队列中既有读请求又有写请求，默认写请求的优先级高，即使读请求先到，所以MyISAM不适合于有大量查询和修改并存的情况，那样查询进程会长时间阻塞。

\* InnoDB支持外键，\*\*MyISAM不支持外键\*\*。

\* \*\*InnoDB的主键范围更大\*\*，最大是MyISAM的2倍。

\* \*\*InnoDB不支持全文索引\*\*，而MyISAM支持。全文索引是指对char、varchar和text中的每个词（停用词除外）建立倒排序索引。MyISAM的全文索引其实作用不大，因为它不支持中文分词，必须由使用者分词后加入空格再写到数据表里，而且少于4个汉字的词会和停用词一样被忽略掉。

\* \*\*MyISAM支持GIS数据\*\*，InnoDB不支持。即MyISAM支持以下空间数据对象：Point、Line、Polygon、Surface等。

\* \*\*没有where的count(\*)使用MyISAM要比InnoDB快得多\*\*。因为MyISAM内置了一个计数器，`count(\*)`时它直接从计数器中读，而InnoDB必须扫描全表。所以在InnoDB上执行`count(\*)`时一般要伴随where，且where中要包含\*\*主键以外的\*\*索引列（因为InnoDB中PRIMARY KEY是和raw data存放在一起的，而其他index则是单独存放，然后有个指针指向PRIMARY KEY。所以只是`count(\*)`的话使用其他index扫描更快，而PRIMARY KEY则主要在扫描索引同时要返回raw data时的作用较大）。

​

### MySQL一行记录最多能有多少个VARCHAR(255)类型的列？

\*\*MySQL表中一行的长度不能超过65535字节\*\*，VARCHAR(N)使用额外的1到2字节来存储值的长度，如果N<=255，则使用一个字节，否则使用两个字节；如果表格的编码为UTF8（一个字符占3个字节），那么VARCHAR(255)占用的字节数为255 \* 3 + 2 = 767，这样，一行就最多只能有65535 / 765 = 85个VARCHAR(255)类型的列。

### MySQL事务隔离级别（ISOLATION LEVEL）理解

\*\*READ UNCOMMITTED\*\*

最低的隔离级别，可读取其他事务未提交的数据（事务可以看到其他事务尚未提交的修改），可能造成脏读。

\*\*READ COMMITTED\*\*

只能读取已提交的数据，但是不可重复读（避免脏读）

\*\*REPEATABLE READ\*\*

可重复读。

用户A查询完之后，用户B将无法更新用户A所查询到的数据集中的任何数据（但是可以更新、插入和删除用户A查询到的数据集之外的数据），直到用户A事务结束才可以进行更新，这样就有效的\*\*防止用户在同一个事务中读取到不一致的数据\*\*。

\*\*SERIALIZABLE\*\*

事务串行化，必须等待当前事务执行完，其他事务才可以执行写操作，有多个事务同时设置SERIALIZABLE时会产生死锁：

```

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

```

这是四个隔离级别中限制最大的级别。因为并发级别较低，所以应只在必要时才使用该选项。

\*\*使用事务时设置隔离级别\*\*

```

START TRANSACTION

SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}

COMMIT

ROLLBACK

```

### MySQL分区表理解

分区是一种粗粒度，简易的索引策略，适用于大数据的过滤场景。\*\*对于大数据（如10TB）而言，索引起到的作用相对小\*\*，因为索引的空间与维护成本很高，另外如果不是索引覆盖查询，将导致回表，造成大量磁盘IO。

分区表分为RANGE、LIST、HASH、KEY四种类型，并且\*\*分区表的索引是可以局部针对分区表建立的\*\*。

用RANGE创建分区表：

```

CREATE TABLE sales (

id INT AUTO\_INCREMENT,

amount DOUBLE NOT NULL,

order\_day DATETIME NOT NULL,

PRIMARY KEY(id, order\_day)

) ENGINE=Innodb PARTITION BY RANGE(YEAR(order\_day)) (

PARTITION p\_2010 VALUES LESS THAN (2010),

PARTITION p\_2011 VALUES LESS THAN (2011),

PARTITION p\_2012 VALUES LESS THAN (2012),

PARTITION p\_catchall VALUES LESS THAN MAXVALUE);

```

如果这么做，则order\_day必须包含在主键中，且会产生一个问题：当年份超过阈值，到了2013，2014时需要手动创建这些分区，更好的方法是使用HASH：

```

CREATE TABLE sales (

id INT PRIMARY KEY AUTO\_INCREMENT,

amount DOUBLE NOT NULL,

order\_day DATETIME NOT NULL

) ENGINE=Innodb PARTITION BY HASH(id DIV 1000000);

```

这种分区表示每100W条数据建立一个分区，且没有阈值范围的影响。

如果想为一个表创建分区，这个表最多只能有一个唯一索引（主键也是唯一索引）。如果没有唯一索引，可指定任何一列为分区列；否则就只能指定唯一索引中的任何一列为分区列。查询时需用到分区的列，不然会遍历所有的分区，比不分区的查询效率还低，MySQL支持子分区。

在表建立后也可以新增、删除、合并分区。

### MySQL主从同步理解

\*\*复制机制（Replication）\*\*

master通过复制机制，将master的写操作通过`binlog`传到slave生成中继日志(`relaylog`)，slave再将中继日志redo，使得主库和从库的数据保持同步。

\*\*slave主动拉取模式下复制相关的3个MySQL线程\*\*

\* \*\*slave上的I/O线程\*\*：向master请求数据

\* \*\*master上的Binlog Dump线程\*\*：读取binlog事件并把数据发送给slave的I/O线程

\* \*\*slave上的SQL线程\*\*：读取中继日志并执行，更新数据库

\*\*相关监控命令\*\*

```

show processlist # 查看MySQL进程信息，包括3个同步线程的当前状态

show master status # 查看master配置及当前复制信息

show slave status # 查看slave配置及当前复制信息

```

### MySQL异步

MySQL异步是指\*\*将MySQL连接事件驱动化\*\*，这样就变成了非阻塞IO。数据库操作并不会阻塞进程，\*\*在MySQL-Server返回结果时再执行对应的逻辑\*\*。

\*\*注意点\*\*：

\* 异步MySQL并没有节省SQL执行的时间

\* 一个MySQL连接同时只能执行1个SQL，如果异步MySQL存在并发那么必须创建多个MySQL连接

\* 异步回调程序中，异步MySQL并没有提升性能。\*\*异步最大的好处是可以高并发\*\*，如果并发1万个请求，那么就需要建立1万个MySQL连接，这会给MySQL-Server带来巨大的压力。

虽然应用层代码使用异步回调避免了自身的阻塞，实际上真正的瓶颈是数据库服务器。异步MySQL还带来了额外的编程复杂度，所以除非是特殊场景的需求，否则不建议使用异步MySQL。如果程序中坚持要使用异步，那么必须是异步MySQL+连接池的形式。超过规定的MySQL最大连接后，应当对SQL请求进行排队，而不是创建新连接，避免大量并发请求导致MySQL服务器崩溃。

### MySQL性能相关的配置参数

\* \*\*max\_connecttions\*\* ：最大连接数

\* \*\*table\_cache\*\* ：缓存打开表的数量

\* \*\*key\_buffer\_size\*\* ：索引缓存大小

\* \*\*query\_cache\_size\*\* ：查询缓存大小

\* \*\*sort\_buffer\_size\*\* ：排序缓存大小(会将排序完的数据缓存起来)

\* \*\*read\_buffer\_size\*\* ：顺序读缓存大小

\* \*\*read\_rnd\_buffer\_size\*\* ：某种特定顺序读缓存大小(如order by子句的查询)

查看配置方法：

```

show variables like '%max\_connecttions%';

```

### MySQL的索引类型

\* 普通索引：最基本的索引，没有任何限制，MyISAM中默认的BTREE类型的索引，也是大多数情况下用到的索引。

```

// 直接创建索引

CREATE INDEX index\_name ON table(column(length))

// 修改表结构的方式添加索引

ALTER TABLE table\_name ADD INDEX index\_name ON (column(length))

// 创建表的时候同时创建索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL ,

`content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

INDEX index\_name (title(length))

)

// 删除索引

DROP INDEX index\_name ON table

```

\* 唯一索引

索引列的值必须唯一，但\*\*允许有空值\*\*（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。

```

// 创建唯一索引

CREATE UNIQUE INDEX indexName ON table(column(length))

// 修改表结构

ALTER TABLE table\_name ADD UNIQUE indexName ON (column(length))

// 创建表的时候直接指定

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL ,

`content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

UNIQUE indexName (title(length))

);

```

\* 全文索引

\*\*仅可用于MyISAM表\*\*。可以从CHAR、VARCHAR或TEXT列中作为CREATE TABLE语句的一部分被创建，或是随后使用ALTER TABLE或CREATE INDEX被添加。对于较大的数据集，将资料输入一个没有FULLTEXT索引的表中，然后创建索引，其速度比把资料输入现有FULLTEXT索引的速度更为快。不过对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法。

```

// 创建表的适合添加全文索引

CREATE TABLE `table` (

`id` int(11) NOT NULL AUTO\_INCREMENT ,

`title` char(255) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL ,

`content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL ,

`time` int(10) NULL DEFAULT NULL ,

PRIMARY KEY (`id`),

FULLTEXT (content)

);

// 修改表结构添加全文索引

ALTER TABLE article ADD FULLTEXT index\_content(content)

// 直接创建索引

CREATE FULLTEXT INDEX index\_content ON article(content)

```

\* 单列索引、多列索引

多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

\* 组合索引

```

ALTER TABLE article ADD INDEX index\_titme\_time (title(50),time(10));

```

建立这样的组合索引，\*\*相当于分别建立了下面两组组合索引\*\*：

```

title,time

title

```

没有time这样的组合索引，因为MySQL组合索引执行\*\*最左前缀匹配\*\*，简单的理解就是只从最左面的开始组合。如：

```

SELECT \* FROM article WHREE title='测试' AND time=1234567890; # 使用title,time索引

SELECT \* FROM article WHREE utitle='测试'; # 使用title索引

SELECT \* FROM article WHREE time=1234567890; # 未使用索引

```

\*\*MySQL只对以下操作符才使用索引：\*\*

```

<、<=、=、>、>=、between、in、以及某些时候的like(不以通配符%或\_开头的情形)。

```

\*\*理论上每张表里面最多可创建16个索引\*\*。

### MySQL自增主键出现不连续情况的原因？如何修复？

\*\*造成自增主键不连续的原因\*\*

\* INSERT语句不管是否成功，都会增加AUTO\_INCREMENT值。

\* 进行了DELETE相关操作。

\* ROLLBACK相关。

\*\*修复自增主键，使其连续\*\*

```

INSERT INTO th\_page2(site,url,title,title\_index,content,tag,created\_at,updated\_at,deleted\_at)

SELECT site,url,title,title\_index,content,tag,created\_at,updated\_at,deleted\_at FROM th\_page ORDER BY tag;

DROP TABLE th\_page;

ALTER TABLE th\_page2 RENAME th\_page;

```

### ON与WHERE有什么区别？

执行连接操作时，可先用ON先进行过滤，减少连接操作的中间结果，然后用WHERE对连接产生的结果再一次过滤。但是，如果是左/右连接，在ON条件里对主表的过滤是无效的，仍然会用到主表的所有记录，连接产生的记录如果不满足主表的过滤条件那么从表部分的数据会置为NULL。

## Redis

---

### Redis中几种实现锁的方式的比较

\*\*使用INCRE\*\*

```c

$value = $redis->get($lock);

if($value < 1 ){

$redis->incr($lock,1);

// ...

$redis->decr($lock,1);

}

```

\*\*使用WATCH\*\*

```c

// 被WATCH的键会被监视，并会发觉这些键是否被改动过了。 如果有至少一个被监视的键在EXEC执行之前被修改了，那么整个事务都会被取消

WATCH mykey

$val = GET mykey // 乐观锁

$val = $val + 1

MULTI

SET mykey $val

EXEC

```

\*\*使用SETNX\*\*

是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置。

```c

// 缓存过期时通过SetNX获取锁，如果成功了就更新缓存，然后删除锁

$ok = $redis->setNX($key, $value);

if ($ok) {

$cache->update();

$redis->del($key);

}

```

存在问题：如果请求执行因为某些原因意外退出了，导致创建了锁但是没有删除锁，那么这个锁将一直存在，以至于以后缓存再也得不到更新。

因此需要给锁加一个过期时间以防不测。

```php

// 加锁

$redis->multi();

$redis->setNX($key, $value);

$redis->expire($key, $ttl);

$redis->exec();

```

存在问题：当多个请求到达时，虽然只有一个请求的SetNX可以成功，但是任何一个请求的Expire却都可以成功，如此就意味着即便获取不到锁，也可以刷新过期时间，如果请求比较密集的话，那么过期时间会一直被刷新，导致锁一直有效。

从 2.6.12 起，SET涵盖了SETEX的功能，并且SET本身已经包含了设置过期时间的功能：

```php

$ok = $redis->set($key, $value, array('nx', 'ex' => $ttl));

if ($ok) {

$cache->update();

$redis->del($key);

}

```

### Redis HyperLogLog理解

\*\*基数\*\*

如数据集 {1, 3, 5, 7, 5, 7, 8}，那么这个数据集的基数集为 {1, 3, 5 ,7, 8}，基数（不重复元素个数）为5。

基数估计：在误差可接受的范围内，快速计算基数。

Redis HyperLogLog是\*\*用来做基数统计的算法\*\*，HyperLogLog的优点是，\*\*在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的\*\*。

因为HyperLogLog只会根据输入元素来计算基数，而不会储存输入元素本身，所以HyperLogLog不能像集合那样，返回输入的各个元素。

\*\*例：\*\*

```

redis 127.0.0.1:6379> PFADD w3ckey "redis"

1) (integer) 1

redis 127.0.0.1:6379> PFADD w3ckey "mongodb"

1) (integer) 1

redis 127.0.0.1:6379> PFADD w3ckey "mysql"

1) (integer) 1

redis 127.0.0.1:6379> PFCOUNT w3ckey

(integer) 3

```

\*\*基本命令\*\*

\* PFADD key element [element ...] # 添加指定元素到HyperLogLog中

\* PFCOUNT key [key ...] # 返回给定HyperLogLog的基数估算值

\* PFMERGE destkey sourcekey [sourcekey ...] # 将多个HyperLogLog合并为一个HyperLogLog

### Redis常见配置选项

\*\*使用指定的配置文件启动Redis\*\*

```

./redis-server redis.conf

```

\*\*通过Redis命令查看、设置配置项\*\*

```

# 查看loglevel配置项

redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"

2) "notice"

# 查看所有配置项

redis 127.0.0.1:6379> CONFIG GET \*

1) "dbfilename"

2) "dump.rdb"

3) "requirepass"

4) ""

5) "masterauth"

6) ""

7) "unixsocket"

8) ""

# 设置loglevel项

redis 127.0.0.1:6379> CONFIG SET loglevel "notice"

OK

```

### Redis常见配置参数

\* daemonize no # 启用守护进程，默认不启用

\* pidfile /var/run/redis.pid # 指定pid文件，当以守护进程方式运行时用到

\* port 6379 # 指定Redis监听端口

\* bind 127.0.0.1 # 绑定的主机地址

\* timeout 300 # 当客户端闲置指定时间后关闭连接，如果指定为0，表示关闭该功能

\* loglevel verbose # 指定日志记录级别：debug、verbose、notice、warning

\* logfile stdout # 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

\* databases 16 # 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

\* 持久化阈值设置

指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

```

save <seconds> <changes>

```

Redis默认配置文件中提供了三个条件：

```

save 900 1 # 900秒（15分钟）内有1个更改

save 300 10 # 300秒（5分钟）内有10个更改

save 60 10000 # 60秒内有10000个更改

```

\* rdbcompression yes # 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

\* dbfilename dump.rdb # 指定本地数据库文件名，默认值为dump.rdb

\* dir ./ # 指定本地数据库存放目录

\* slaveof <masterip> <masterport> # 当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

\* masterauth <master-password> # 当master服务设置了密码保护时，slav服务连接master的密码

\* requirepass xxxxx # 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

\* maxclients 128 # 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

\* maxmemory <bytes> # 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区。

\* appendonly no # 指定是否在每次更新操作后进行日志记录，Redis在\*\*默认情况下是异步的把数据写入磁盘\*\*，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

\* appendfilename appendonly.aof # 指定更新日志文件名，默认为appendonly.aof

\* appendfsync everysec #指定更新日志条件，共有3个可选值：

```

no # 等操作系统进行数据缓存同步到磁盘（快）

always # 每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）

everysec # 每秒同步一次（折衷，默认值）

```

\* vm-enabled no # 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中

\* vm-swap-file /tmp/redis.swap # 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

\* vm-max-memory 0 # 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

\* vm-page-size 32 # Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不确定，就使用默认值

\* vm-pages 134217728 # 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，在磁盘上每8个pages将消耗1byte的内存。

\* vm-max-threads 4 # 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

\* glueoutputbuf yes # 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

\* 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

```

hash-max-zipmap-entries 64

hash-max-zipmap-value 512

```

\* activerehashing yes # 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）

\* include /path/to/local.conf # 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

### Redis通信协议理解

\*\*发送格式\*\*

```

\*<参数的个数>CR LF

$<参数1字节数>CR LF

<参数1>CR LF

...

$<参数n字节数>CR LF

<参数n>CR LF

```

例如`set mykey myvalue`命令，相应的字符串为：

```

\*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n

```

\*\*响应格式\*\*

响应的类型都是由返回数据的第一个字节决定的，有如下几种类型：

\* "+" 代表一个状态信息，如 +ok

\* "-" 代表发生了错误，如操作运算操作了错误的类型

\* ":" 返回的是一个整数，如：":11\r\n。

一些命令返回一些没有任何意义的整数，如LastSave返回一个时间戳的整数值， INCR返回一个加1后的数值；一些命令如exists将返回0或者1代表是否true or false；其他一些命令如SADD, SREM 在确实执行了操作时返回1 ，否则返回0

\* "$" 返回一个块数据，被用来返回一个二进制安全的字符串

\* "\\*" 返回多个块数据（用来返回多个值， 总是第一个字节为"\*"， 后面写着包含多少个相应值，如：

```

C:LRANGE mylist 0 3

S:\*4

S:$3

S:foo

S:$3

S:bar

S:$5

$:world

```

如果指定的值不存在，那么返回\*0

### Redis的管道技术理解

Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。这意味着通常情况下一个请求会遵循以下步骤：

\* （1）客户端向服务端发送一个查询请求，并监听Socket返回，\*\*通常是以阻塞模式\*\*，等待服务端响应。

\* （2）服务端处理命令，并将结果返回给客户端。

\*\*Redis管道技术可以在服务端未响应时，客户端能够继续向服务端发送请求，并最终一次性读取所有服务端的响应\*\*。

如：

```

$(echo -en "PING\r\n SET w3ckey redis\r\nGET w3ckey\r\nINCR visitor\r\nINCR visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379

+PONG

+OK

redis

:1

:2

:3

```

以上实例中通过使用PING命令查看redis服务是否可用，之后设置了w3ckey的值为redis，然后获取w3ckey的值并使得visitor自增3次。在返回的结果中可以看到\*\*这些命令一次性向redis服务提交，并最终一次性读取所有服务端的响应\*\*。管道技术\*\*最显著的优势是提高了redis服务的性能\*\*（批量提交命令）。

# Web前端

---

---

## HTML

---

### DOCTYPE

<!DOCTYPE> 声明`不是HTML标签`；它是指示web浏览器关于页面使用哪个HTML版本进行编写的指令。必须是HTML文档的第一行，位于<html>标签之前。没有结束标签。对大小写不敏感。

在HTML 4.01中有三种<!DOCTYPE>声明。在HTML5中只有一种：<!DOCTYPE html>

应该始终向HTML文档添加<!DOCTYPE>声明，这样浏览器才能获知文档类型。

### meat标签的http-equiv属性

http-equiv属性可用于模拟一个HTTP响应头。

// 设定网页的到期时间（一旦网页过期，必须到服务器上重新传输）

＜meta http-equiv="expires" content="Wed, 20 Jun 2007 22:33:00 GMT"＞

// 禁止浏览器从本地机的缓存中调阅页面内容（这样设定，访问者将无法脱机浏览）

＜meta http-equiv="Pragma" content="no-cache"＞

// 自动刷新并指向新页面（停留2秒钟后自动刷新到URL网址）

＜meta http-equiv="Refresh" content="2; URL=http://www.net.cn/"＞

// 设置Cookie（如果网页过期，那么存盘的cookie将被删除）

＜meta http-equiv="Set-Cookie" content="cookievalue=xxx;expires=Wednesday, 20-Jun-2007 22:33:00 GMT;path=/"＞

// 显示窗口的设定（强制页面在当前窗口以独立页面显示，防止别人在框架里调用自己的页面）

＜meta http-equiv="Window-target" content="\_top"＞

// 设定页面使用的字符集

＜meta http-equiv="Content-Type" content="text/html; charset=gb2312"＞

// 网页等级评定

<meta http-equiv="Pics-label" contect="">

在IE的internet选项中有一项内容设置，可以防止浏览一些受限制的网站，而网站的限制级别就是通过meta属性来设置的。

// 设定进入页面时的特殊效果

<meta http-equiv="Page-Enter" contect="revealTrans(duration=1.0,transtion=12)">

// 设定离开页面时的特殊效果

<meta http-equiv="Page-Exit" contect="revealTrans(duration=1.0,transtion=12)">

// 清除缓存

<meta http-equiv="cache-control" content="no-cache">

// 关键字，给搜索引擎用的

<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">

// 页面描述，给搜索引擎用的

<meta http-equiv="description" content="This is my page">

## CSS

---

### CSS选择器总结

`类型选择器`、`元素选择器`、`简单选择器`：

```css

p {color:black;}

```

`后代选择器`：

```css

blockquote p{color:black;}

```

`ID选择器`：

```css

#intro{color:black;}

```

`类选择器`：

```css

.intro{color:black;}

```

`伪类`：根据文档结构之外的其他条件对元素应用样式，例如表单元素或链接的状态

```css

tr:hover{background-color:red;}

input:focus{background-color:red;}

a:hover,a:focus,a:active{color:red;}

```

:link和:visited称为链接伪类，只能应用于锚元素。:hover、:active和:focus称为动态伪类，理论上可以应用于任何元素。

可以把伪类连接在一起，创建更复杂的行为：

```css

a:visited:hover{color:red;}

```

`通用选择器`：匹配所有可用元素

```css

\*{

padding:0;

margin:0;

}

```

通用选择器与其他选择器结合使用时，可以用来对某个元素的所有后代应用样式。

`子选择器`：只选择元素的直接后代，而不是像后代选择器一样选择元素的所有后代。

```css

#nav>li{

padding-left:20px;

color:red;

}

```

`相邻同胞选择器`：用于定位同一个父元素下与某个元素相邻的下一个元素。

```css

h2 + p{

font-size:1.4em;

}

```

`属性选择器`：根据某个属性是否存在或者属性的值来寻找元素。

```css

abbr[title]{

border-bottom:1px dotted #999;

}

abbr[title]:hover{

cursor:help;

}

a[rel=’nofollow’]{

color:red;

}

```

注意：属性名无引号，属性值有引号。

对于属性可以有多个值的情况（空格分割），属性选择器允许根据属性值之一来寻找元素：

```css

.blogroll a[rel~=’co-worker’]{...}

```

## JavaScript

---

### self=this

This question is not specific to jQuery, but specific to JavaScript in general. The core problem is how to "channel" a variable in embedded functions. This is the example:

```javascript

var abc = 1; // we want to use this variable in embedded functions

function xyz(){

console.log(abc); // it is available here!

function qwe(){

console.log(abc); // it is available here too!

}

...

};

```

This technique relies on using a closure. But it doesn't work with this because this is a pseudo variable that may change from scope to scope dynamically:

```javascript

// we want to use "this" variable in embedded functions

function xyz(){

// "this" is different here!

console.log(this); // not what we wanted!

function qwe(){

// "this" is different here too!

console.log(this); // not what we wanted!

}

...

};

```

What can we do? Assign it to some variable and use it through the alias:

```javascript

var self = this; // we want to use this variable in embedded functions

function xyz(){

// "this" is different here! --- but we don't care!

console.log(self); // now it is the right object!

function qwe(){

// "this" is different here too! --- but we don't care!

console.log(self); // it is the right object here too!

}

...

};

```

this is not unique in this respect: arguments is the other pseudo variable that should be treated the same way — by aliasing.

### 使用 [].slice.call将对象转换为数组的局限性

```javascript

var arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c',

length: 3

};

var arr = [].slice.call(arrayLike);

console.log(arr); // ["a", "b", "c"]

```

被转换为数组的对象必须有length属性，所谓类似数组的对象，本质特征只有一点，即必须有length属性。

```javascript

var arrayLike = {

'0': 'a',

'1': 'b',

'2': 'c'

};

var arr = [].slice.call(arrayLike);

console.log(arr); // []

```

### ~运算符在逻辑判断中的作用

```javascript

console.log(!0); // true

console.log(!1); // false

console.log(!2); // false

console.log(!-1); // false

console.log(~0); // -1

console.log(~1); // -2

console.log(~2); // -3

console.log(~-1); // 0

console.log(!~0); // false

console.log(!~1); // false

console.log(!~2); // false

console.log(!~-1); // true

```

### 如何解决jQuery不同版本之间、与其他js库之间的冲突？

（1）同一页面jQuery多个版本或冲突解决方法

```javascript

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

</head>

<body>

<!-- 引入 jquery 1.8.0 -->

<script type="text/javascript" src="http://code.jquery.com/jquery-1.8.0.min.js"></script>

<script type="text/javascript">

var $180 = $;

</script>

<!-- 引入 jquery 1.9.0 -->

<script type="text/javascript" src="http://code.jquery.com/jquery-1.9.0.min.js"></script>

<script type="text/javascript">

var $190 = $;

</script>

<!-- 引入 jquery 2.0.0 -->

<script type="text/javascript" src="http://code.jquery.com/jquery-2.0.0.min.js"></script>

<script type="text/javascript">

var $200 = $;

</script>

<script type="text/javascript">

console.log($180.fn.jquery);

console.log($190.fn.jquery);

console.log($200.fn.jquery);

</script>

</body>

</html>

```

（2）同一页面jQuery和其他js库冲突解决方法

1）jQuery在其他js库之前

```javascript

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

</head>

<body>

<!-- 引入 jquery 1.8.0 -->

<script type="text/javascript" src="http://code.jquery.com/jquery-1.8.0.min.js"></script>

<script type="text/javascript">

var $180 = $;

console.log($.fn.jquery); # 1.8.0

</script>

<!-- 引入 其他库-->

<script type="text/javascript">

$ = {

fn:{

jquery:"111cn.net"

}

};

</script>

<script type="text/javascript">

console.log($.fn.jquery); # 111cn.net

console.log($180.fn.jquery); # 1.8.0

</script>

</body>

</html>

```

2）jQuery在其他js库后

```javascript

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

</head>

<body>

<!-- 引入 其他库-->

<script type="text/javascript">

$ = {

fn:{

jquery:"111cn.net"

}

};

</script>

<!-- 引入 jquery 1.8.0 -->

<script type="text/javascript" src="http://code.jquery.com/jquery-1.8.0.min.js"></script>

<script type="text/javascript">

console.log($.fn.jquery); # 1.8.0

var $180 = $.noConflict();

console.log($.fn.jquery); # 111cn.net

console.log($180.fn.jquery); # 1.8.0

</script>

</body>

</html>

```

### JavaScript中\_\_proto\_\_与prototype的关系

\_\_proto\_\_ 对象的内部原型

prototype 构造函数的原型

1.所有函数/构造函数（包括内置的、自定义的）的\_\_proto\_\_都指向Function.prototype，它是一个空函数（Empty function）

Number.\_\_proto\_\_ === Function.prototype // true

Global对象的\_\_proto\_\_不能直接访问；

Arguments对象仅在函数调用时由JS引擎创建；

Math，JSON是以对象形式存在的，无需new，它们的\_\_proto\_\_是Object.prototype：

JSON.\_\_proto\_\_ === Object.prototype // true

2.构造函数都来自于Function.prototype，包括Object及Function自身，因此都继承了Function.prototype的属性及方法。如length、call、apply、bind等

3.Function.prototype也是唯一一个typeof XXX.prototype为 “function”的prototype。其它的构造器的prototype都是一个对象：

console.log(typeof Function.prototype) // function 一个空函数

console.log(typeof Object.prototype) // object

console.log(typeof Number.prototype) // object

4.Function.prototype的\_\_proto\_\_等于Object的prototype：

console.log(Function.prototype.\_\_proto\_\_ === Object.prototype) // true 体现了在Javascript中`函数也是一等公民`

5.Object.prototype的\_\_proto\_\_为null

Object.prototype.\_\_proto\_\_ === null // true 到顶了

6.所有对象的\_\_proto\_\_都指向其构造器的prototype

var obj = {name: 'jack'}

var arr = [1,2,3]

var reg = /hello/g

console.log(obj.\_\_proto\_\_ === Object.prototype) // true

console.log(arr.\_\_proto\_\_ === Array.prototype) // true

console.log(reg.\_\_proto\_\_ === RegExp.prototype) // true

function Person(name) {

this.name = name

}

var p = new Person('jack')

console.log(p.\_\_proto\_\_ === Person.prototype) // true

7.每个对象都有一个`constructor属性`，可以获取它的构造器

function Person(name) {

this.name = name

}

Person.prototype.getName = function() {} // 修改原型

var p = new Person('jack')

console.log(p.\_\_proto\_\_ === Person.prototype) // true

console.log(p.\_\_proto\_\_ === p.constructor.prototype) // true

8.使用对象字面量方式定义对象的构造函数，则对象的constructor的prototype可能不等于对象的\_\_proto\_\_

function Person(name) {

this.name = name

}

// 使用对象字面量方式定义的对象其constructor指向Object，Object.prototype是一个空对象{}

Person.prototype = {

getName: function() {}

}

var p = new Person('jack')

console.log(p.\_\_proto\_\_ === Person.prototype) // true

console.log(p.\_\_proto\_\_ === p.constructor.prototype) // false

### JS中{}+[]和[]+{}的返回值情况是怎样的？

先说 [] + {} 。一个数组加一个对象。

加法会进行隐式类型转换，规则是调用其 valueOf() 或 toString() 以取得一个非对象的值（primitive value）。如果两个值中的任何一个是字符串，则进行字符串串接，否则进行数字加法。

[] 和 {} 的 valueOf() 都返回对象自身，所以都会调用 toString()，最后的结果是字符串串接。[].toString() 返回空字符串，({}).toString() 返回“[object Object]”。最后的结果就是“[object Object]”。

然后说 {} + [] 。看上去应该和上面一样。但是 {} 除了表示一个对象之外，也可以表示一个空的 block。在 [] + {} 中，[] 被解析为数组，因此后续的 + 被解析为加法运算符，而 {} 就解析为对象。但在 {} + [] 中，{} 被解析为空的 block，随后的 + 被解析为正号运算符。即实际上成了：

{ // empty block }

+[]

即对一个空数组执行正号运算，实际上就是把数组转型为数字。首先调用 [].valueOf() 。返回数组自身，不是primitive value，因此继续调用 [].toString() ，返回空字符串。空字符串转型为数字，返回0，即最后的结果。

### js对象遍历顺序问题

```javascript

var a = {

b:'a',

10: "vv",

1:"a",

a:''

}

console.log(a);

```

Object:

1:"a"

10:"vv"

a:""

b:"a"

### JSONP的原理

JSONP是一种解决跨域传输JSON数据的问题的解决方案，是一种非官方跨域数据交互协议。

Ajax（或者说js）直接请求普通文件存在跨域无权限访问的问题，但是`Web页面上凡是拥有"src"属性的标签引用文件时则不受是否跨域的影响`。如果想通过纯web端（ActiveX控件、服务端代理、Websocket等方式不算）跨域访问数据就只有一种可能：在远程服务器上设法把数据装进js格式的文件里，供客户端调用和进一步处理。

为了便于客户端使用数据，逐渐形成了一种非正式传输协议:JSONP，该协议的一个要点就是`允许用户传递一个callback参数给服务端，然后服务端返回数据时会将这个callback参数作为函数名来包裹住JSON数据`，这样客户端就可以随意定制自己的函数来自动处理返回数据了。

例：使用Javascript实现JSONP

```html

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title></title>

<script type="text/javascript">

// 回调函数

var flightHandler = function(data){

alert('你查询的航班结果是：票价 ' + data.price + ' 元，' + '余票 ' + data.tickets + ' 张。');

};

// 拼凑url

var url = "http://flightQuery.com/jsonp/flightResult.aspx?code=CA1998&callback=flightHandler";

// 拼凑<script>标签，用于发出JSONP请求

var script = document.createElement('script');

script.setAttribute('src', url);

document.getElementsByTagName('head')[0].appendChild(script);

</script>

</head>

<body>

</body>

</html>

```

服务器端返回格式：

```json

flightHandler({

"code": "CA1998",

"price": 1780,

"tickets": 5

});

```

例：使用jQuery实现JSONP

```html

<script type="text/javascript" src=jquery.min.js"></script>

<script type="text/javascript">

jQuery(document).ready(function(){

$.ajax({

type: "get",

async: false,

url: "http://flightQuery.com/jsonp/flightResult.aspx?code=CA1998",

dataType: "jsonp",

jsonp: "callback", // 传递给请求处理程序或页面的，用以获得jsonp回调函数名的参数名(一般默认为:callback)

jsonpCallback:"flightHandler", // 自定义的jsonp回调函数名称，没有定义的话会jQuery会自动生成以jQuery开头的函数

success: function(json){

alert('您查询到航班信息：票价： ' + json.price + ' 元，余票： ' + json.tickets + ' 张。');

},

error: function(){

alert('fail');

}

});

});

</script>

```

jquery在处理jsonp类型的ajax时自动生成回调函数并把数据（即不含函数名的纯json格式的数据）取出来供success属性方法来调用。

### AMD规范、requireJS

因为JavaScript本身的灵活性：框架没办法绝对的约束你的行为，一件事情总可以用多种途径去实现，所以我们只能在方法学上去引导正确的实施方法。

AMD规范：Asynchronous Module Definition，即异步模块加载机制。AMD规范简单到只有一个API，即define函数：

　　define([module-name?], [array-of-dependencies?], [module-factory-or-object]);

module-name: 模块标识，可以省略。

array-of-dependencies: 所依赖的模块，可以省略。

module-factory-or-object: 模块的实现，或者一个JavaScript对象。

当define函数执行时，它首先会异步地去调用第二个参数中列出的依赖模块，当所有的模块被载入完成之后，如果第三个参数是一个回调函数则执行，然后告诉系统模块可用，也就通知了依赖于自己的模块自己已经可用。

实例：

```javascript

define("alpha", ["require", "exports", "beta"], function (require, exports, beta) { // 依赖的模块做参数传入

　　 exports.verb = function() {

　　 return beta.verb();

　　 }

　　});

```

requireJS例一：使用requirejs动态加载jquery

目录结构：

/web

/index.html # 页面文件

/jquery-1.7.2.js # jquery模块

/main.js # js加载主入口，在引用require.js文件时通过data-main属性指定

/require.js # requireJS文件

index.html文件内容：

```html

<!doctype html>

<html>

<head>

<meta charset="utf-8">

<script data-main="main" src="require.js"></script> # “main”指向模块加载入口文件main.js

</head>

<body>

...

</body>

</html>

```

main.js文件内容：

```javascript

require.config({

paths: {

jquery: 'jquery-1.7.2'

}

});

require(['jquery'], function($) {

alert($().jquery);

});

```

引用模块jquery，因为这里配置了jquery的paths参数，所以将使用参数所对应的值'jquery-1.7.2'（js后缀名省略）。

jQuery从1.7以后支持AMD规范，所以当jQuery作为一个AMD模块运行时，它的模块名是jquery（区分大小写）。

如果文件名'jquery-1.7.2'改为jquery，则无需配置path参数。

requireJS例二：使用自定义模块

目录结构：

/web

/js

/cache.js # 自定义模块

/event.js # 自定义模块

/main.js

/selector.js # 自定义模块

/index.html

/require.js

index.html文件内容：

```html

<html>

<head>

<meta charset="utf-8">

<style type="text/css">

p {

width: 200px;

background: gray;

}

</style>

</head>

<body>

<p>p1</p><p>p2</p><p>p3</p><p>p4</p><p>p5</p>

<script data-main="js/main" src="require.js"></script>

</body>

</html>

```

cache模块内容：返回一个js对象

```javascript

define(function() {

...

return {

set: function(el, key, val) {

var c = ...;

...

return c;

},

...

};

});

```

event模块内容：依赖于cache模块（define第一个参数为依赖的模块列表，第二个参数为一个函数，函数形参直接使用模块）

```javascript

define(['cache'], function(cache) {

...

return { # 返回一个js对象

bind : bind,

unbind : unbind,

trigger : trigger

};

});

```

selector模块内容：

```javascript

define(function() {

function query(selector,context) {

...

}

return query; # 返回一个js函数

});

```

main.js内容：

```javascript

require.config({

baseUrl: 'js'

});

require(['selector', 'event'], function($, E) { # 函数两个形参一一对应所依赖的两个模块

var els = $('p');

for (var i=0; i < els.length; i++) {

E.bind(els[i], 'click', function() {

alert(this.innerHTML);

});

}

});

```

### call()、apply()

call和apply都是为了改变某个函数运行时的context即上下文而存在的，换句话说，就是为了改变函数体内部this的指向。因为JavaScript 的函数存在`定义时上下文`和`运行时上下文`以及`上下文是可以改变的`这样的概念。

二者的作用完全一样，只是接受参数的方式不太一样。例如，有一个函数 func1 定义如下：

var func1 = function(arg1, arg2) {};

就可以通过 func1.call(this, arg1, arg2); 或者 func1.apply(this, [arg1, arg2]); 来调用。其中 this 是你想指定的上下文，他可以任何一个 JavaScript 对象(JavaScript 中一切皆对象)，call 需要把参数按顺序传递进去，而 apply 则是把参数放在数组里。

JavaScript 中，某个函数的参数数量是不固定的，因此要说适用条件的话，当你的参数是明确知道数量时，用 call，而不确定的时候，用 apply，然后把参数 push 进数组传递进去。当参数数量不确定时，函数内部也可以通过 arguments 这个数组来便利所有的参数。

call和apply是为了动态改变this而出现的，当一个object没有某个方法，但是其他的有，我们可以借助call或apply用其它对象的方法来操作。

# 计算机系统

---

---

## C语言

---

### new、delete与malloc、free之间的联系和区别

\*\*malloc/free和new/delete的联系\*\*

1. 存储方式相同。malloc和new动态申请的内存\*\*都位于堆中\*\*。申请的内存\*\*都不能自动被操作系统收回\*\*，都需要配套的free和delete来释放。

2. 除了带有构造函数和析构函数的类等数据类型以外，\*\*对于一般数据类型，如int、char等等，两组动态申请的方式可以通用\*\*，作用效果一样，只是形式不一样。

3. 内存泄漏对于malloc或者new都是可以检查出来的，区别在于new可以指明是哪个文件的哪一行，而malloc没有这些信息。

4. 两组都需要配对使用，malloc配free，new配delete。在C++中，两组之间不能混着用（虽说有时能编译过，但容易存在较大的隐患）。

\*\*malloc/free和new/delete的区别\*\*

1. malloc返回void类型指针，free的形参为void指针，new和delete直接带具体类型的指针。

2. malloc和free属于C语言中的\*\*函数\*\*，需要库的支持，而new/delete是C++中的\*\*运算符\*\*，况且可以重载，所以\*\*new/delete的执行效率高些\*\*。

3. 在C++中，\*\*new是类型安全的\*\*，而malloc不是。例如：

```c

// 编译时指出错误

int\* p = new char[10];

//对数组需要加中括号“[]”

delete [] p;

// 编译时无法指出错误

int\* p = malloc(sizeof(char)\*10);

//只需要所释放内存的头指针（free释放malloc分配的数组）。在malloc和free的面前没有对象没有数组，只有“内存块”。一次malloc分配的东西，一次free一定能回收。至于内存块的大小内存管理会进行记录，这应该是库函数的事。free的真正弊端在于它不会调用析构函数。

free (p);

```

4. 使用new动态申请类对象的内存空间时，类对象的构建要调用构造函数，相当于对内存空间进行了初始化。而\*\*malloc动态申请的类对象的内存空间时，不会初始化\*\*，也就是说申请的内存空间无法使用，因为类的初始化是由构造函数完成的。

5. 不能用malloc和free来完成类对象的动态创建和删除。

\*\*calloc、realloc：\*\*

```c

void \*calloc(int n,int size);

```

函数返回值为void型指针。如果执行成功，函数从堆上获得size \* n的字节空间，并返回该空间的首地址。如果执行失败，函数返回NULL。\*\*该函数与malloc函数的一个显著不同时是，calloc函数得到的内存空间是经过初始化的，其内容全为0\*\*。calloc函数适合为数组申请空间，可以将size设置为数组元素的空间长度，将n设置为数组的容量。

realloc函数的功能比malloc函数和calloc函数的功能更为丰富，可以实现内存分配和内存释放的功能，其函数声明如下：

```c

void \* realloc(void \* p,int n);

```

其中，指针p必须为指向堆内存空间的指针，即由malloc函数、calloc函数或realloc函数分配空间的指针。realloc函数将指针p指向的内存块的大小改变为n字节。如果n小于或等于p之前指向的空间大小，那么。保持原有状态不变。如果n大于原来p之前指向的空间大小，那么，系统将重新为p从堆上分配一块大小为n的内存空间，同时，将原来指向空间的内容依次复制到新的内存空间上，p之前指向的空间被释放。\*\*relloc函数分配的空间也是未初始化的\*\*。

注意：使用malloc函数，calloc函数和realloc函数分配的内存空间都要\*\*使用free函数或指针参数为NULL的realloc函数来释放\*\*。

### 为什么不建议让异常离开析构函数？

程序抛出异常时候，会导致栈展开，局部对象依次析构。\*\*如果析构过程中再次抛出异常，程序将会立即中止\*\*。

### 使用define定义一年有多少毫秒

```

#define MS\_OF\_YEAR (365\*24\*60\*60\*1000)UL

```

（对于整数溢出的考虑）

### 使用define定义max函数

```c++

#define MAX(a,b) (a)>(b)?(a): (b)

```

（对于define中()使用的把握）

例：

```

#define TEST1 a+b

#define TEST2 (a+b)

void main(void)

{

int a, b, c, d;

c = TEST1; //相当于 c = a+b;

d = TEST2; //相当于 d = (a+b);

}

```

这样写是防止忽略运算符优先级而导致的错误。

### 关键字volatile的意义及使用场景

意思是volatile告诉编译器\*\*不要持有变量的临时拷贝\*\*;

场景可抽象为两个线程，线程1和线程2通过某种方式共享一个变量，线程1根据变量状态进行某种操作，但线程2有可能改变变量的值，由于线程1中对变量的值一直保存到寄存器中，就不会发现变量的改变，此时线程1会做出错误的行为，当然，这个问题也可以由锁进行同步，但会有较大的时间消耗，volatile可以较好的解决此问题。

### 写一个病毒

```c++

while(1){

int \*p = new int[ 10000000 ]；

}

```

### 如何避免同一头文件的多次include

`#ifndef … `或者 `#pragma once`

`#pragma once`是编译器相关的，就是说即使这个编译系统上有效，但在其他编译系统也不一定可以，不过现在基本上已经是每个编译器都有这个杂注了。

`#ifndef，#define，#endif`是C/C++语言中的宏定义，通过宏定义避免文件多次编译。所以在所有支持C++语言的编译器上都是有效的，如果写的程序要跨平台，最好使用这种方式。

### 字节对齐理解

在结构中，编译器为结构的每个成员按其\*\*自然边界\*\*（alignment）分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

\*\*为了使CPU能够对变量进行快速的访问，变量的起始地址应该具有某些特性，即所谓的对齐\*\*。比如4字节的int型，其起始地址应该位于4字节的边界上，即起始地址能够被4整除。

对于标准数据类型，它的地址只要是它的长度的整数倍就行了，而\*\*非标准数据类型按下面的原则对齐\*\*：

\* 数组：按照基本数据类型对齐，第一个对齐了后面的自然也就对齐了。

\* 联合：按其包含的长度最大的数据类型对齐。

\* 结构体：结构体中每个数据类型都要对齐。

比如有如下一个结构体：

```c

struct stu{

char sex;

int length;

char name[10];

};

struct stu my\_stu;

```

由于\*\*在x86下，GCC默认按4字节对齐\*\*，它会在sex后面跟name后面分别填充三个和两个字节使length和整个结构体对齐。于是我们sizeof(my\_stu)会得到长度为20，而不是15。

![image](/images/tech/img\_4.png)

\*\*需要字节对齐的根本原因在于CPU访问数据的效率问题\*\*。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。而如果变量在自然对齐位置上，则只要一次就可以取出数据。

### 对一个包含虚函数成员的对象bzero()会有什么问题？

\*\*对包含虚函数成员的对象bzero会破坏该对象的虚函数表（VTABLE）\*\*，调用该虚函数时将core。

原型：

```c

extern void bzero（void \*s, int n）;

```

用法：

```c

#include <string.h>

```

功能：置字节字符串s的前n个字节为零且包括`\0`。

说明：bzero无返回值，并且使用strings.h头文件，strings.h曾经是posix标准的一部分，但是在POSIX.1-2001标准里面，这些函数被标记为了遗留函数而不推荐使用。在POSIX.1-2008标准里已经没有这些函数了。推荐使用memset替代bzero。

### 常量存储器与栈空间

在函数体内声明

```

[1] char \*str="abc";

[2] char str[]={'a','b','c'};

```

有什么区别？

\* \[1\]\[2\] 中str变量都分配在栈上

\* [1] 中str指向常量存储区的字符串”abc”，其中字符串末尾会补0

\* [2] 中str数组的内容存储于栈空间，数组大小为3，字符串不会补0

### 异或操作总结

```

1 ^ 0 = 1

0 ^ 1 = 1

0 ^ 0 = 0

1 ^ 1 = 0

0 ^ a = a

```

代码验证：

```

int a = 123, b = 456;

a ^ b = 435;

a ^ b ^ a = 456 (b)

a ^ b ^ b = 123 (a)

```

应用：不用临时变量交换两值：

```

a = a ^ b

b = b ^ a

a = a ^ b

```

### `int \*p[5]`和`int (\*p)[5]`的区别

前者定义了指针的数组，后者定义了指向数组的指针

### 用C++实现一个不能被继承的类

\*\*将构造函数和析构函数声明为私有函数，该函数就不可被继承\*\*。同时为了该类可以被实例化，在类中定义一个静态函数，返回初始化的一个类对象。

### 结构体大小如何判断？

\*\*偏移量\*\*

偏移量指的是结构体变量中成员的地址和结构体变量地址的差。结构体大小等于最后一个成员的偏移量加上最后一个成员的大小。

由于存储变量时地址对齐的要求，编译器在编译程序时会\*\*遵循两条原则\*\*：

\* 1.结构体变量中成员的偏移量必须是成员大小的整数倍（0被认为是任何数的整数倍）

\* 2.结构体大小必须是所有成员大小的整数倍。

此外：\*\*结构体变量的首地址能够被其最宽基本类型成员的大小所整除\*\*；

因此\*\*不同的定义顺序会影响到结构体的大小\*\*：

```c

struct s{

char c;

int i;

char cc;

}; // 大小为12

struct s{

char c;

char cc;

int i;

}; // 大小为8

```

当结构体中的成员又是另外一种结构体类型时，只需要把其展开，展开后的结构体的第一个成员的偏移量应当是被展开的结构体中最大的成员的整数倍。

\*\*基本数据类型所占字节数\*\*

```

类型 字节

char 1

short int 2

int 2(16bit)/4(32bit)/4(64bit)

long 4(16bit)/4(32bit)/8(64bit)

指针变量 4

float 4

double 8

long long 8

long double 10

```

各种数据类型所占字节长度，主要是int型,long型和指针数据类型的差异。

\* int型数据，如果是16bit平台,则是2个字节，如果是32bit的，则占4个字节,64bit仍然是4字节。

\* long型数据，如果是16bit平台,则是4个字节，如果是32bit的，则占4个字节,64bit仍然是8字节。

\* 指针型数据，比较特殊，大多是4个字节，只有在16bit平台，并且指针式段内寻址时才是2个字节。

另外注意：sizeof(表达式)这样的使用，sizeof是给出其操作数所需要占用的内存大小，在编译时就可以确定。因此不需要去计算表达式的值；

因此有：

```c

int i = 3;

cout << sizeof(i++) << endl;

cout << i << endl;

```

输出4,3。i++根本没有执行。

![image](/images/tech/img\_3.png)

### 虚函数和纯虚函数的区别

定义一个函数为虚函数，不代表函数为不被实现的函数，定义它为虚函数是为了允许用基类的指针来调用子类的这个函数。定义一个函数为纯虚函数，才代表函数没有被实现，定义他是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

\*\*虚函数有实现，纯虚函数没有方法的实现\*\*。包含纯虚函数的类将成为抽象类，不可实例化对象。纯虚函数必循在其子类中进行重写，不然其子类也成为抽象类。不能实例化对象。

### 虚基类成员的可见性

假定通过多个派生路径继承名为X的成员，有下面三种可能性：

\* 如果在每个路径中 X 表示同一虚基类成员，则没有二义性，因为共享该成员的单个实例。

\* 如果在某个路径中 X 是虚基类的成员，而在另一路径中 X 是后代派生类的成员，也没有二义性——特定派生类实例的优先级高于共享虚基类实例。

\* 如果沿每个继承路径 X 表示后代派生类的不同成员，则该成员的直接访问是二义性的。

像非虚多重继承层次一样，这种二义性最好用在派生类中提供覆盖实例的类来解决。

\*\*特殊的初始化语义\*\*：通常，每个类只初始化自己的直接基类。如果使用常规规则，就可能会多次初始化虚基类。类将沿着包含该虚基类的每个继承路径初始化。为了解决这个重复初始化问题，从具有虚基类的类继承的类对初始化进行特殊处理。在虚派生中，由最低层派生类的构造函数初始化虚基类。虽然由最低层派生类初始化虚基类，但是任何直接或间接继承虚基类的类一般也必须为该基类提供自己的初始化式。只要可以创建虚基类派生类类型的独立对象，该类就必须初始化自己的虚基类，这些初始化式只有创建中间类型的对象时使用。

### 试图用宏开始或结束一段注释是不行的

```c

#define BSC //

#define BMC /\*

#define EMC \*/

```

\* （1）BSC my single-linecomment

\* （2）BMC my multi-linecomment EMC

（1）和（2）都错误，因为\*\*注释先于预处理指令被处理\*\*，当这两行被展开成`//…`或`/\*…\*/`时,注释已处理完毕此时再出现`//…`或`/\*…\*/`自然错误.

### C++中模板的编译过程

\* 第一遍扫描到模板定义时将token流存入语法树中，不做其它操作

\* 第二遍当模板被实例化时用模板实参代入进行运算，将所有的模板参数换为实参进行语法和语义分析

特别需要注意的是\*\*类模板的成员函数只有在调用的时候才会被实例化\*\*。

### C++中虚函数的实现机制？

表面现象：虚函数是在类中被声明为virtual的成员函数，\*\*当编译器看到通过指针或引用调用此类函数时，对其执行晚绑定\*\*，即通过指针（或引用）指向的类的类型信息来决定该函数是哪个类的。

实现机制：\*\*编译器对每个包含虚函数的类创建一个表（称为VTABLE）\*\*。在VTABLE中，编译器放置特定类的虚函数地址。在每个带有虚函数的类中，编译器置一指针，称为vpointer（缩写为VPTR），指向这个对象的VTABLE。通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入取得这个VPTR，并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生。

### C++局部变量，全局变量，静态变量的作用域，生命期？

C++变量根据定义位置的不同，具有不同的作用域，作用域可分为6种：全局作用域，局部作用域，语句作用域，类作用域，命名作用域和文件作用域。

\*\*从作用域看\*\*

\* 全局变量：具有全局作用域。全局变量只需在一个源文件中定义，就可以作用于所有的源文件。当然，其他不包括全局变量定义的源文件需要用extern关键字再次声明这个全局变量。

\* 静态局部变量：具有局部作用域。它只被初始化一次，自从第一次初始化直到程序结束都一直存在，他和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。

\* 局部变量：也只有局部作用域，他是自动对象，他在程序运行期间不是一直存在，而是只在函数执行期间存在，函数的一次调用结束后，变量就被撤销，其所占用的内存也被收回。

\* 静态全局变量：也具有全局作用域，他与全局变量的区别在于如果程序包含多个文件的话，他\*\*作用于定义它的文件里，不能作用到其他文件里\*\*，即被static关键字修饰过的变量具有\*\*文件作用域\*\*。这样即使两个不同的源文件都定义了相同的静态全局变量，他们也是不同的变量。

\*\*从分配内存空间看\*\*

全局变量、静态局部变量、静态全局变量都在\*\*静态存储区\*\*分配空间，而局部变量在\*\*栈\*\*分配空间。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上没有什么不同。区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

\* 静态变量会被放在程序的静态数据存储区里，这样可以在下一次调用的时候还可以保持原来的赋值。这一点是他与堆栈变量和堆变量的区别

\* 变量用static告知编译器，自己仅仅在变量的作用域范围内可见。这一点是他与全局变量的区别。

从以上分析可以看出，\*\*把局部变量改变为静态变量后是改变了他的存储方式，即改变了他的生存期。把全局变量改变为静态变量后是改变了他的作用域，限制了他的使用范围，因此static这个说明符在不同的地方起的作用是不同的\*\*。

\*\*TIPS\*\*

\* 若全局变量仅在单个文件中访问，则可以讲这个变量修改为静态全局变量。

\* 若全局变量仅在单个函数中使用，则可以将这个变量修改为该函数的静态局部变量。

\* 全局变量、静态局部变量、静态全局变量都存放在静态数据存储区。

\* 函数中必须要使用static变量的情况：当某函数的返回值为指针类型时，则必须是static的局部变量的地址作为返回值，若为auto类型，则返回为错指针。

### char的整型运算

```c

char a = '5';

char b = '6';

cout << a+b << endl; // 输出107

a = a - '0';

cout << a << endl; // 输出 ♣

```

\*\*附：ASCII码\*\*

```

0:48、A:65、a:97

'A' == 'a'-32

```

### define中为何经常会使用 do{...}while(0);来包装多条语句代码

`do{...}while(0)`的目的是为了在for循环和if语句时，避免出现下面的情况：

```c++

#define xxx i++; i--;

for (I = 0 ; I < 10; I ++) xxx;

```

展开后变为

```c++

for ( I = 0 ; I < 10; I ++ ) I ++; I --;

```

(对define中do{}while(0)的理解)

### define的一些注意点

\* `#define SQR(x) printf("Thesquareof x is%d.\n",((x)\*(x)))`;

如果这样使用宏：

```

SQR(8);

```

则输出为：

```

The squareof x is 64.

```

注意，\*\*引号中的字符x被当作普通文本来处理，而不被当作一个可以被替换的语言符号\*\*。假如确实希望在字符串中包含宏参数，那就可以使用`#`，它可以把语言符号转化为字符串。上面的例子改一改：

```

#define SQR(x) printf("The squareof "#x" is%d.\n",((x)\*(x)));

```

再使用：

```

SQR(8);

```

则输出的是：

```

The squareof 8 is 64.

```

\* 求两个数的平方

```

#define SQR(x) x \* x

```

假设x的值是个表达式`10+1，SQR(x)`被替换后变成`10+1\*10+1`这并不是想要得到的。括起来就好了：

```

#define SQR(x) （（x）\*（x））

```

求两个数的和：

```

#define SUM (x)（x）+（x）

```

而代码又写成这样：

```

SUM (x)\* SUM (x)

```

替换后变成：

```

（5\*3）+（5\*3）\*（5\*3）+（5\*3）

```

所以又错了！所以最外层的括号最好也别省了。

要搞定宏定义表达式其实很简单，别吝啬括号就行了。

注意这一点：\*\*宏函数被调用时是以实参代换形参。而不是值传送\*\*。

\* 和#运算符一样，\*\*##运算符可以用于宏函数的替换部分\*\*。这个运算符把两个语言符号组合成单个语言符号。看例子：

```

#define XNAME(n) x##n

```

如果这样使用宏：`XNAME(8)`

则会被展开成这样：`x8`

`##`就是个粘合剂，将前后两部分粘合起来。

### static、const、volatile、typeof关键字的用途描述

\* static：静态函数、静态变量、静态类成员

\* const：const变量，const指针、const函数

\* volatile：多线程共享变量

\* typeof：获取类型值

### std::vector实现原理及特定场景下的改进

\* 说一下std::vector的实现原理，主要讲一下和内存管理相关的内容

\* 常驻内存程序，一个std::vector的生命周期和程序生命周期相同，且会频繁的调用std::vector的push\_back()和clear()方法，调用clear()方法时，vector.size()小于1万的概率为0.95,vector.size()可能出现的最大值为100万。如果程序中有多个这样的std::vector实例，程序长期运行后，会导致内存持续增长，一定时间后，可能将内存耗尽。请问，如何用较小的代价修改vector的设计，来避免内存持续增长问题。

答：

\* 内存不够用时，双倍扩容。clear()时，不释放内存，以减少内存分配次数。

\* 修改clear()方法，当vector.size() 大于1万时，释放内存。

### STL中的stable\_sort()的时间复杂度是多少

`stable\_sort()`使用的算法类似于自适应算法，当空间足够的时，时间复杂度是`O(nlogn)`；当空间比较紧张时，时间复杂度是`O(nlogn\*logn)`。

## Linux

---

### IO阻塞、非阻塞、同步、异步

\*\*同步和异步\*\*

同步和异步是\*\*针对应用程序和内核的交互而言\*\*的，同步指的是用户进程触发I/O操作并等待或者轮询的去查看I/O操作是否就绪，而异步是指用户进程触发I/O操作以后便开始做自己的事情，而当I/O操作已经完成的时候会得到I/O完成的通知。

\*\*阻塞和非阻塞\*\*

阻塞和非阻塞是\*\*针对于进程在访问数据的时候\*\*，根据I/O操作的就绪状态来采取的不同方式，是一种读取或者写入函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

\*\*服务器IO模型\*\*

\* 阻塞式模型（blocking IO）

大部分的socket接口都是阻塞型的（ listen()、accpet()、send()、recv() 等）。阻塞型接口是指系统调用（一般是 IO接口）不返回调用结果并让当前线程一直阻塞，只有当该系统调用获得结果或者超时出错时才返回。在线程被阻塞期间，线程将无法执行任何运算或响应任何的网络请求，这给多客户机、多业务逻辑的网络编程带来了挑战。

![image](/images/tech/basic\_9.png)

\* 多线程的服务器模型（Multi-Thread）

应对多客户机的网络应用，最简单的解决方式是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是\*\*让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接\*\*。但是如果要同时响应成千上万路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应效率。

在多线程的基础上，可以考虑使用线程池或连接池，线程池旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。连接池维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统。

\* 非阻塞式模型（Non-blocking IO）

相比于阻塞型接口的显著差异在于，在被调用之后立即返回。

![image](/images/tech/basic\_10.png)

需要应用程序调用许多次来等待操作完成。这可能效率不高，因为在很多情况下，当内核执行这个命令时，应用程序必须要进行\*\*忙碌等待\*\*，直到数据可用为止。

另一个问题，在循环调用非阻塞IO的时候，将大幅度占用CPU，所以一般使用select等来检测是否可以操作。

\* 多路复用IO（IO multiplexing）

支持I/O复用的系统调用有select、poll、epoll、kqueue等。使用select返回后，仍然需要轮询再检测每个socket的状态（读、写），这样的轮询检测在大量连接下也是效率不高的。因为\*\*当需要探测的句柄值较大时，select () 接口本身需要消耗大量时间去轮询各个句柄\*\*。

很多操作系统提供了更为高效的接口，如Linux 提供了`epoll`，BSD提供了`kqueue`，Solaris提供了`/dev/poll `…。如果需要实现更高效的服务器程序，类似epoll这样的接口更被推荐，能显著\*\*提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率\*\*。

![image](/images/tech/basic\_11.png)

\* 使用事件驱动库libevent的服务器模型

libevent是一个\*\*事件触发的\*\*网络库，适用于Windows、Linux、BSD等多种平台，\*\*内部使用select、epoll、kqueue、IOCP等系统调用管理事件机制\*\*。

libevent库提供一种事件机制，它作为底层网络后端的包装器。事件系统让为连接添加处理函数变得非常简便，同时降低了底层IO复杂性。这是libevent系统的核心。

创建libevent服务器的基本方法是，注册当发生某一操作（比如接受来自客户端的连接）时应该执行的函数，然后调用主事件循环event\_dispatch()。执行过程的控制现在由libevent系统处理。注册事件和将调用的函数之后，事件系统开始自治；在应用程序运行时，可以在事件队列中添加（注册）或 删除（取消注册）事件。事件注册非常方便，可以通过它添加新事件以处理新打开的连接，从而构建灵活的网络处理系统。

\* 信号驱动IO模型（Signal-driven IO）

让内核在描述符就绪时发送SIGIO信号通知应用程序。

![image](/images/tech/basic\_12.png)

\* 异步IO模型（asynchronous IO）

告知内核启动某个操作，并让内核在整个操作（包括将数据从内核复制到用户的缓冲区）完成后通知用户。这种模型与信号驱动模型的主要区别在于：信号驱动式I/O是由内核通知何时可以启动一个I/O操作，而异步I/O模型是由内核通知I/O操作何时完成。

![image](/images/tech/basic\_13.png)

\*\*异步IO与同步IO的区别\*\*

A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;

An asynchronous I/O operation does not cause the requesting process to be blocked;

两者的区别就在于synchronous IO做IO operation的时候会将process阻塞。按照这个定义阻塞、非阻塞、IO多路复用其实都属于同步IO。

\*\*异步IO与非阻塞IO的区别\*\*

在non-blocking IO中，虽然进程大部分时间都不会被block，但是它仍然要求进程去主动的check，并且当数据准备完成以后，也需要进程主动的再次调用`recvfrom`来将数据拷贝到用户内存。而asynchronous IO则完全不同。它就像是用户进程\*\*将整个IO操作（分为两步：准备数据、将数据从内核复制到用户空间）交给了他人（kernel）完成，然后他人做完后发信号通知\*\*。在此期间，用户进程\*\*不需要去检查IO操作的状态，也不需要主动的去拷贝数据\*\*。

### select、poll、epoll

\*\*文件描述符（fd）\*\*

\*\*文件描述符是一个简单的整数\*\*，用以标明每一个\*\*被进程所打开的文件和socket的索引\*\*。第一个打开的文件是0，第二个是1，依此类推。最前面的三个文件描述符（0、1、2）分别与标准输入（stdin），标准输出（stdout）和标准错误（stderr）对应。\*\*Unix操作系统通常给每个进程能打开的文件数量强加一个限制\*\*。当用完所有的文件描述符后，将不能接收用户新的连接，直到一部分当前请求完成，相应的文件和socket被关闭。

\*\*IO多路复用技术\*\*

select，poll，epoll都是IO多路复用的机制。I/O多路复用通过一种机制，可以\*\*监视多个文件描述符\*\*，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。\*\*select、poll、epoll本质上都是同步I/O，因为它们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间\*\*。

\*\*epoll的改进\*\*

\* \*\*select、poll需要自己不断轮询所有fd集合\*\*，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait中进入睡眠的进程。虽然都要睡眠和交替，但是\*\*select和poll在醒着的时候要遍历整个fd集合，而epoll在醒着的时候只要判断一下就绪链表是否为空就行了\*\*，这节省了大量的CPU时间。这就是\*\*回调机制带来的性能提升\*\*（本质的改进在于epoll采用基于事件的就绪通知方式）。

\* select、poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。（本质的改进就是使用了\*\*内存映射\*\*（mmap）技术）

epoll被公认为Linux2.6下性能最好的多路I/O就绪通知方法，\*\*实现高效处理百万句柄\*\*。

### nohup与&的区别

最直观的区别：

使用&执行的命令，要是关闭终端（将发出SIGHUP信号），命令会停止。而使用nohup执行的命令，既使把终端关了，命令仍然会继续运行。

nohup执行的命令会\*\*忽略所有挂断（SIGHUP）信号\*\*。

一般结合使用：nohup command &

### 启动Linux守护进程的方法

\*\*Linux进程类型\*\*

Linux操作系统包括如下3种不同类型的进程，每种进程都有其自己的特点和属性。

\* \*\*交互进程\*\*：由shell启动的进程。可在前台运行，也可在后台运行；

\* \*\*批处理进程\*\*：一个进程序列；

\* \*\*守护进程\*\*：守护进程是指在后台运行而又没有启动终端或登录shell。守护进程一般由系统开机时通过脚本自动激活启动或者由root用户通过shell启动。守护进程总是活跃的，一般在后台运行，所以它所处的状态是等待处理任务的请求。

\*\*启动守护进程有如下几种方法\*\*

\* 在引导系统时启动：通过脚本启动，这些脚本一般位于`/etc/rc.d`中。在/etc目录下的很多rc文件都是启动脚本 。rc0.d、rc1.d、rc2.d、rc3.d、rc4.d、rc5.d、rc6.d，其中的数字代表在指定的runlevel下运行相应的描述，0代表关机，6代表重启。其中，以k开头的文件表示关闭，以s开头的文件表示重启。可查看相应文件夹中的readme文件。rc0.d、rc1.d、rc2.d、rc3.d、rc4.d、rc5.d、rc6.d、rcS.d都连接到`/etc/init.d`文件夹，\*\*该目录中存放着守护进程的运行文件\*\*。

\* 人工手动从shell提示符启动：\*\*任何具有权限的用户都可以启动相应的守护进程\*\*。

```

# 启动FTP服务器，ubuntu下默认已经安装了vsfptd服务器

root@Ubuntu:~# /etc/init.d/vsftpd start

```

\* 使用crond守护进程启动

\* 执行at命令启动

### 静态链接与动态链接比较

静态链接就是把外部函数库，拷贝到可执行文件中。这样做的好处是，兼容性好，不用担心用户机器缺少某个库文件；缺点是安装包会比较大，而且多个应用程序之间，无法共享库文件。

动态连接的做法正好相反，外部函数库不进入安装包，只在运行时动态引用。好处是安装包会比较小，多个应用程序可以共享库文件；缺点是用户必须事先安装好库文件，而且版本和安装位置都必须符合要求，否则就不能正常运行。

现实中，大部分软件采用动态连接，共享库文件。这种动态共享的库文件，Linux平台是后缀名为.so的文件，Windows平台是.dll文件，Mac平台是.dylib文件。

### Linux中通过编译安装的方式安装程序，各步骤操作分别做什么工作？

源码要运行，必须先转成二进制的机器码。这是编译器的任务。

对于简单的代码，可以直接调用编译器生成二进制文件后运行，如：

```

$ gcc test.c

$ ./a.out

```

对于复杂的项目，编译过程通常分成3个部分：

```

$ ./configure

$ make

$ make install

```

整个编译安装过程分为以下步骤：

\* 配置

配置信息保存在一个配置文件之中，约定俗成是一个叫做`configure的`脚本文件。通常它是由\*\*autoconf工具\*\*生成的。\*\*编译器通过运行这个脚本，获知编译参数\*\*。如果用户的系统环境比较特别，或者有一些特定的需求，就需要手动向configure脚本提供编译参数，如：

```

# 指定安装后的文件保存在www目录，并且编译时加入mysql模块的支持

$ ./configure --prefix=/www --with-mysql

```

\* 确定标准库和头文件的位置

从配置文件中知道标准库和头文件的位置。

\* 确定依赖关系

源码文件之间往往存在依赖关系，编译器需要确定编译的先后顺序。假定A文件依赖于B文件，编译器应该保证：只有在B文件编译完成后，才开始编译A文件。且当B文件发生变化时，A文件会被重新编译。

\*\*编译顺序保存在一个叫做`makefile`的文件中\*\*，里面列出哪个文件先编译，哪个文件后编译。而\*\*makefile文件由configure脚本运行生成\*\*，这就是为什么编译时configure必须首先运行的原因。

\* 预编译头文件

不同的源码文件，可能引用同一个头文件（比如stdio.h）。编译的时候，头文件也必须一起编译。为了节省时间，\*\*编译器会在编译源码之前，先编译头文件\*\*。这保证了头文件只需编译一次，不必每次用到的时候，都重新编译了。不过，并不是头文件的所有内容都会被预编译。用来声明宏的#define命令，就不会被预编译。

\* 预处理

编译器就开始替换掉源码中的头文件和宏以及移除注释。

\* 编译

编译器就\*\*开始生成机器码\*\*。对于某些编译器来说，还存在一个中间步骤，会先把源码转为汇编码（assembly），然后再把汇编码转为机器码。这种转码后的文件称为对象文件（object file）。

\* 链接

\*\*把外部函数的代码（通常是后缀名为.lib和.a的文件）添加到可执行文件中\*\*。这就叫做链接（linking）。这种通过拷贝，将外部函数库添加到可执行文件的方式，叫做`静态连接`（static linking）

make命令的作用，就是从第（4）步头文件预编译开始，一直到做完这一步。

\* 安装

\*\*将可执行文件保存到用户事先指定的安装目录\*\*。这一步还必须完成创建目录、保存文件、设置权限等步骤。这整个的保存过程就称为安装（Installation）。

\* 操作系统链接

以某种方式\*\*通知操作系统，让其知道可以使用这个程序了\*\*。这就要求在操作系统中，登记这个程序的元数据：文件名、文件描述、关联后缀名等等。Linux系统中，这些信息通常保存在`/usr/share/applications`目录下的`.desktop`文件中。

make install命令，就用来完成安装和操作系统连接这两步。

\* 生成安装包

将上一步生成的可执行文件，做成可以分发的安装包。通常是将可执行文件（连带相关的数据文件），以某种目录结构，保存成压缩文件包，交给用户。

\* 动态链接

开发者可以在编译阶段选择可执行文件连接外部函数库的方式，到底是静态连接（编译时连接），还是动态连接（运行时连接）。

### 进程通信的类型

\* 共享存储器系统(Shared-Memory System)

全局变量、共享数据结构、共享存储区

\* 消息传递系统(Message passing system)

进程间的数据交换，是以格式化的消息(message)为单位的；在计算机网络中，又把message称为报文。

\* 管道通信(Pipe)

管道是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)， 以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。

### 进程为什么要挂起？

在多进程程序系统中，进程在处理器上交替运行，在运行、就绪和阻塞3种基本状态之间不断地发生变化。由于进程的不断创建，系统资源（特别是主存资源）已不能满足进程运行的要求。此时就必须将某些进程挂起，对换到磁盘镜像区，暂时不参与进程调度，以平衡系统负载的目的。如果系统出现故障，或者是用户调试程序，也可能需要将进程挂起检查问题。

所谓挂起状态，\*\*实际上就是一种静止的状态\*\*。一个进程被挂起后，不管它是否在就绪状态，系统都不分配给它处理机（区别于阻塞状态）。这样\*\*进程的三态模型\*\*（执行、就绪、阻塞）就变为\*\*五态模型\*\*：执行状态、活动就绪状态、静止就绪状态、活动阻塞状态和静止阻塞状态

\* 活动就绪：指进程\*\*在主存并且可被调度\*\*的状态 （对应于三态的就绪状态）

\* 静止就绪：指进程\*\*被对换到辅存\*\*时的就绪状态，是不能被直接调度的状态，只有当主存中没有活动就绪态进程，或者是挂起态进程具有更高的优先级，系统将把挂起就绪态进程调回主存并转换为活动就绪。

\* 活动阻塞：指进程在主存中。\*\*一旦等待的事件产生，便进入活动就绪状态\*\*（对应于三态的阻塞状态）

\* 静止阻塞：指进程对换到辅存时的阻塞状态。\*\*一旦等待的事件产生，便进入静止就绪状态\*\*。

### Linux ext2和ext3文件系统的区别？

\*\*Linux文件系统\*\*

Linux ext2/ext3文件系统\*\*使用索引节点来记录文件信息\*\*。索引节点是一个结构，它包含了一个文件的长度、创建及修改时间、权限、所属关系、磁盘中的位置等信息。\*\*一个文件系统维护了一个索引节点的数组，每个文件或目录都与索引节点数组中的唯一一个元素对应\*\*。系统给每个索引节点分配了一个号码，也就是该节点在数组中的索引号，称为\*\*索引节点号\*\*。

Linux文件系统将文件索引节点号和文件名同时保存在目录中。所以，\*\*目录只是将文件的名称和它的索引节点号结合在一起的一张表\*\*，目录中每一对文件名称和索引节点号称为一个`连接`。 \*\*对于一个文件来说有唯一的索引节点号与之对应，对于一个索引节点号，却可以有多个文件名与之对应\*\*。因此，在磁盘上的同一个文件可以通过不同的路径去访问它。

Linux缺省情况下使用的文件系统为ext2，ext2文件系统的确高效稳定。但是，随着Linux系统在关键业务中的应用，Linux文件系统的弱点也渐渐显露出来：其中系统缺省使用的\*\*ext2文件系统是非日志文件系统\*\*。这在关键行业的应用是一个致命的弱点。

ext3文件系统是直接从Ext2文件系统发展而来，目前ext3文件系统已经非常稳定可靠。它完全兼容ext2文件系统。用户可以平滑地过渡到一个日志功能健全的文件系统中来。这实际上了也是ext3日志文件系统初始设计的初衷。

\*\*ext3日志文件系统的特点\*\*

\* 高可用性

系统使用了ext3文件系统后，即使在非正常关机后，系统也不需要检查文件系统。宕机发生后，恢复ext3文件系统的时间只要数十秒钟。

\* 数据的完整性:

ext3文件系统能够极大地提高文件系统的完整性，避免了意外宕机对文件系统的破坏。在保证数据完整性方面，ext3文件系统有2种模式可供选择。其中之一就是\*\*同时保持文件系统及数据的一致性\*\*模式。采用这种方式，你永远不再会看到由于非正常关机而存储在磁盘上的垃圾文件。

\* 文件系统的速度

尽管使用ext3文件系统时，有时在存储数据时可能要多次写数据，但是，\*\*从总体上看来，ext3比ext2的性能还要好一些\*\*。这是因为ext3的日志功能对磁盘的驱动器读写头进行了优化。所以，文件系统的读写性能较之ext2文件系统并来说，性能并没有降低。

\* 数据转换

由ext2文件系统转换成ext3文件系统非常容易，只要简单地键入两条命令即可完成整个转换过程，用户不用花时间备份、恢复、格式化分区等。用一个ext3文件系统提供的小工具tune2fs，它可以将ext2文件系统轻松转换为ext3日志文件系统。另外，ext3文件系统可以不经任何更改，而直接加载成为ext2文件系统。

\* 多种日志模式

ext3有多种日志模式，一种工作模式是对所有的文件数据及metadata（定义文件系统中数据的数据,即数据的数据）进行日志记录（data=journal模式）；另一种工作模式则是只对metadata记录日志，而不对数据进行日志记录，也即所谓data=ordered或者data=writeback模式。系统管理人员可以根据系统的实际工作要求，在系统的工作速度与文件数据的一致性之间作出选择。

### 乐观锁与悲观锁的区别

\*\*悲观锁\*\*

`悲观锁`（Pessimistic Lock）每次去读数据的时候都认为数据会被其他任务修改，所以会上锁，这样其他任务想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在做操作之前先上锁。

\*\*乐观锁\*\*

`乐观锁`（Optimistic Lock）每次去读数据的时候都认为别的任务不会修改，所以不会上锁，但是\*\*在更新（写）的时候\*\*会判断一下在此期间其他任务有没有去更新这个数据(可以使用版本号等机制）。\*\*乐观锁适用于多读的应用类型\*\*，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

### 僵尸状态是每个子进程必经的状态吗？

是的。 任何一个子进程（init除外）在exit()之后，并非马上就消失掉，而是留下一个称为`僵尸进程`（Zombie）的数据结构，等待父进程处理。\*\*这是每个子进程在结束时都要经过的阶段\*\*。

如果子进程在exit()之后，父进程没有来得及处理，这时用`ps -el`命令就能看到子进程的状态是`Z`。如果父进程能及时处理，可能用ps命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵尸状态。如果父进程在子进程结束之前退出，则子进程将由init接管。init将会以父进程的身份对僵尸状态的子进程进行处理。

\*\*僵尸进程的危害\*\*

由于子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。UNIX提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息，就可以得到。这种机制就是： 在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。 但是仍然为其保留一定的信息（包括进程号、退出状态、运行时间等）。直到父进程通过wait/waitpid来取时才释放。但这样就导致了问题，如果进程不调用wait/waitpid的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。此即为僵尸进程的危害，应当避免。

### 分页和分段的主要区别

\* \*\*页是信息的物理单位\*\*，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要而不是用户的需要。\*\*段则是信息的逻辑单位\*\*，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。

\* \*\*页的大小固定且由系统决定\*\*，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而\*\*段的长度却不固定\*\*，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

\* \*\*分页的作业地址空间是一维的\*\*，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；而\*\*分段的作业地址空间则是二维的\*\*，程序员在标识一个地址时，既需给出段名， 又需给出段内地址。

### 后台进程与守护进程有什么区别？

\* 最直观的区别：\*\*守护进程没有控制终端，而后台进程还有\*\*。如通过命令`firefox &`在后台运行firefox，此时firefox虽然在后台运行，但是并没有脱离终端的控制，如果把终端关掉则firefox也会一起关闭。

\* 后台进程的文件描述符继承自父进程，例如shell，所以它也可以在当前终端下显示输出数据。但是\*\*守护进程自己变成进程组长\*\*，其文件描述符号和控制终端没有关联，是控制台无关的。

\* \*\*守护进程肯定是后台进程，但后台进程不一定是守护进程\*\*。基本上任何一个程序都可以后台运行，但守护进程是具有特殊要求的程序，比如它能够脱离自己的父进程，成为自己的会话组长等（这些需要在程序代码中显式地写出来）。

### 如何查看僵尸进程？

`ps -el`，查看`S`状态：

\* `Z`：僵尸进程

\* `S`：休眠状态

\* `D`：不可中断的休眠状态

\* `R`：运行状态

\* `T`：停止或跟踪状态

### 僵尸进程变为孤儿进程

父进程死后，僵尸进程成为"孤儿进程"，过继给1号进程init，init会负责清理僵尸进程。

### 如何查看Linux进程之间的关系？

```

ps -o pid,pgid,ppid,comm | cat

```

输出：

```

PID PGID PPID COMMAND

3003 3003 2986 su

3004 3004 3003 bash

3423 3423 3004 ps

3424 3423 3004 cat

```

每个进程都会属于一个进程组(process group)，每个进程组中可以包含多个进程。进程组会有一个组长进程 (process group leader)，\*\*组长进程的PID成为进程组的ID\*\* (process group ID, PGID)，以识别进程组。`PID`为进程自身的ID，`PGID`为进程所在的进程组的ID， `PPID`为进程的父进程ID。

### 子进程结束后为什么要进入僵尸状态?

因为父进程可能要取得子进程的退出状态等信息。

### 协程理解

\*\*普通程序调用的执行方式\*\*

子程序，或者称为函数，在所有语言中都是\*\*层级调用\*\*，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。

所以\*\*子程序调用是通过栈实现的\*\*，一个线程就是执行一个子程序。子程序调用总是一个入口，一次返回，调用顺序是明确的。

\*\*协程的执行方式\*\*

\*\*协程\*\*

又称微线程，纤程。英文名Coroutine。

协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，\*\*在子程序内部可中断\*\*，然后转而执行别的子程序（是中断后执行，而不是函数调用其他的子程序），在适当的时候再返回来接着执行。

\*\*优点\*\*

\*\*协程的特点在于是一个线程执行（所以不是多线程）\*\*。优势就是极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，\*\*没有线程切换的开销\*\*，和多线程比，线程数量越多，协程的性能优势就越明显。另一个优势就是\*\*不需要多线程的锁机制\*\*，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

\*\*缺点\*\*

\*\*无法利用多核资源\*\*：协程的本质是个单线程，它不能同时将单个CPU的多个核用上，协程需要和进程配合才能运行在多CPU上。当然我们日常所编写的绝大部分应用都没有这个必要，除非是CPU密集型应用。

\*\*进行阻塞操作会阻塞掉整个程序\*\*：这一点和事件驱动一样，可以使用异步IO操作来解决。

### 死锁产生的四个必要条件

\* \*\*互斥条件\*\*：一个资源每次只能被一个进程使用。

\* \*\*请求与保持条件\*\*：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

\* \*\*不剥夺条件\*\*：进程已获得的资源，在未使用完之前，不能强行剥夺。

\* \*\*循环等待条件\*\*：若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件。只要系统发生死锁，这些条件必然成立，而只要上述条件之

一不满足，就不会发生死锁。

### Linux用户身份切换

将一般用户变成root：

```

[test@test test]$ su

```

将身份变成username的身份:

```

[root @test /root ]# sudo [-u username] [command]

```

root可以执行test用户的指令，建立test的文件不需要root的密码仍可以执行root的工具，这时就可以使用sudo。由于执行root身份的工作时，输入的密码是用户的密码，而不是root的密码，所以可以减少root密码外泄的可能性。

# 网络协议

---

---

## HTTP

---

### GET和POST在TCP层的区别

\*\*GET产生一个TCP数据包，POST产生两个TCP数据包\*\*：

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

### HTTP协议缓存协商机制相关的6个HTTP头

HTTP缓存协商机制基于6个HTTP头信息进行，动态内容本身并不受浏览器缓存机制的排斥，\*\*只要HTTP头信息中包含相应的缓存协商信息，动态内容一样可以被浏览器缓存\*\*。不过对于POST类型的请求，浏览器一般不启用本地缓存。除了浏览器缓存，HTTP缓存协商机制同样适用于HTTP缓存代理服务器。

主要涉及以下6个HTTP Header：

`Expires`

`Cache-Control`

`Last-Modified`、`If-Modified-Since`

`ETag`、`If-None-Match`。

\*\*Expires/Cache-Control\*\*是控制浏览器\*\*是否直接从浏览器缓存取数据还是重新发请求到服务器取数据\*\*。只是Cache-Control比Expires可以控制的多一些，而且\*\*Cache-Control会重写Expires的规则\*\*。Cache-Control常见的取值有private、no-cache、max-age、must-revalidate等。如果指定Cache-Control的值为private、no-cache、must-revalidate，那么打开新窗口访问时都会重新访问服务器。而如果指定了max-age值，那么\*\*在此值内的时间里就不会重新访问服务器\*\*，例如：`Cache-control: max-age=5`表示当访问此网页后的5秒内再次访问不会去服务器。

\*\*Last-Modified/If-Modified-Since\*\*和\*\*ETag/If-None-Match\*\*是\*\*浏览器发送请求到服务器后判断文件是否已经修改过\*\*，如果没有修改过就只发送一个304回给浏览器，告诉浏览器直接从自己本地的缓存取数据；如果修改过那就整个数据重新发给浏览器。

\*\*Expires和Cache-Control max-age的区别与联系\*\*

1. Expires在HTTP/1.0中已经定义，Cache-Control:max-age在HTTP/1.1中才有定义。

2. Expires指定一个\*\*绝对的过期时间\*\*(GMT格式)，这么做会导致至少2个问题：

\* 客户端和服务器时间不同步导致Expires的配置出现问题。

\* 很容易在配置后忘记具体的过期时间，导致过期来临出现浪涌现象；（而Cache-Control:max-age指定的是从文档被访问后的存活时间，这个时间是个相对值，相对的是文档第一次被请求时服务器记录的请求时间。

3. Expires指定的时间可以是相对文件的最后访问时间或者修改时间，而max-age相对对的是文档的请求时间。

4. 在Apache中，max-age是根据Expires的时间来计算出来的max-age = expires- request\_time:(mod\_expires.c)

目前主流的浏览器都将HTTP/1.1作为首选，所以当HTTP响应头中同时含有Expires和Cache-Control时，浏览器会优先考虑Cache-Control。

\*\*Last-Modified/If-Modified-Since和ETag/If-None-Match工作方式\*\*

1. 浏览器把缓存文件的最后修改时间通过If-Modified-Since来告诉Web服务器（浏览器缓存里存储的不只是网页文件，还有服务器发过来的该文件的最后服务器修改时间）。服务器会把这个时间与服务器上实际文件的最后修改时间进行比较。如果时间一致，那么返回HTTP状态码304（但不返回文件内容），客户端接到之后，就直接把本地缓存文件显示到浏览器中。如果时间不一致，就返回HTTP状态码200和新的文件内容，客户端接到之后，会丢弃旧文件，把新文件缓存起来，并显示到浏览器中（当文件发生改变，或者第一次访问时，服务器返回的HTTP头标签中有Last-Modified，告诉客户端页面的最后修改时间）。

2. 浏览器把缓存文件的ETag，通过If-None-Match，来告诉Web服务器。思路与第一种类似。

\*\*一个例子\*\*

Request Headers

```

Host localhost

User-Agent Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.8.1.16) Gecko/20080702 Firefox/2.0.0.16

Accept text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,\*/\*;q=0.5

...

If-Modified-Since Tue, 19 Aug 2008 06:49:35GMT

If-None-Match 7936caeeaf6aee6ff8834b381618b513

Cache-Control max-age=0

```

Response Headers

```

Date Tue, 19 Aug 2008 06:50:19 GMT

...

Expires Tue, 19 Aug 2008 07:00:19 GMT

Last-Modified Tue, 19 Aug 2008 06:49:35GMT

Etag 7936caeeaf6aee6ff8834b381618b513

```

对应以上两组缓存控制Header，按F5刷新浏览器和在地址栏里输入网址然后回车。这两个行为是不一样的。\*\*按F5刷新浏览器，浏览器会去Web服务器验证缓存。如果是在地址栏输入网址然后回车，浏览器会直接使用有效的缓存，而不会发http request去服务器验证缓存，这种情况叫做`缓存命中`\*\*。

Cache-Control: public 指可以`公有缓存`，可以是数千名用户共享的。

Cache-Control: private 指只支持`私有缓存`，私有缓存是单个用户专用的。

此外，针对不同的Cache-Control值，对浏览器执行不同的操作，其缓存访问行为也不一样，这些操作包括：打开新窗口、在地址栏回车、按后退按钮、按刷新按钮。

\*\*Last-Modified/If-Modified-Since和ETag/If-None-Match工作方式的区别\*\*

```

<?php

header('Last-Modified:' . gmdate('D, d M Y H:i:s') . ' GMT');

echo time();

?>

```

此时再通过浏览器请求该动态文件，HTTP响应中将会添加一个头信息：

```

Last-Modified:Fri, 20 Mar 2009 07:53:02 GMT

```

对于带有`Last-Modified`的响应，浏览器会对文件进行缓存，并打上一些标记，下次再发出请求时会带上如下的HTTP头信息：

```

If-Modified-Since:Fri, 20 Mar 2009 07:53:02 GMT

```

如果没有修改，服务器会返回304信息：

```

HTTP/1.1 304 Not Modified

...

```

意味着浏览器可以直接使用本地缓存的内容。

\*\*使用基于最后修改时间的缓存协商存在一些缺点\*\*：

1. 很可能文件内容没有变化，而只是时间被更新，此时浏览器仍然会获取全部内容。

2. 当使用多台机器实现负载均衡时，用户请求会在多台机器之间轮询，而不同机器上的相同文件最后修改时间很难保持一致，可能导致用户的请求每次切换到新的服务器时就需要重新获取所有内容。

比如服务器返回如下带ETag的响应：

```

ETag:"74123-b-938fny4nfi8"

```

浏览器在下次请求该内容时会在HTTP头中添加如下信息：

```

If-None-Match:"74123-b-938fny4nfi8"

```

如果相同的话，服务器返回304。

Web服务器可以自由定义ETag的格式和计算方法。

### Referer头的安全问题

\*\*Referer的作用\*\*

Referer是HTTP协议中的一个请求报头，用于\*\*告知服务器用户的来源页面\*\*。比如说从Google搜索结果中点击进入了某个页面，那么该次HTTP请求中的Referer就是Google搜索结果页面的地址。如果某篇博客中引用了其他地方的一张图片，那么对该图片的HTTP请求中的Referer就是那篇博客的地址。

一般\*\*Referer主要用于统计\*\*，像CNZZ、百度统计等可以通过Referer统计访问流量的来源和搜索的关键词（包含在URL中）等等，方便站长们有针性对的进行推广和SEO。

Referer另一个用处就是\*\*防盗链\*\*。可以用referrer-killer（一个js库）来实现反反盗链。

Referer是由浏览器自动加上的，\*\*以下情况是不带Referer的\*\*：

\* （1）直接输入网址或通过浏览器书签访问

\* （2）使用JavaScript的Location.href或者是Location.replace()

\* （3）HTTPS等加密协议

\*\*Referer的安全问题\*\*

以新浪微博曾经的一个漏洞（新浪微博gsid劫持）为例。

gsid是一些网站移动版的认证方式，移动互联网之前较老的手机浏览器不支持cookie，为了能够识别用户身份（实现类似cookie的作用），就在用户的请求中加入了一个类似sessionid的字符串，通过GET方式传递，带有这个id的请求，就代表用户的帐号发起的操作。后来又因用户多次认证体验不好，gsid的失效期是很长甚至永久有效的（即使改了密码也无用，这个问题在很多成熟的web产品上仍在发生）。也就是说，一旦攻击者获取到了这个gsid，就等同于长期拥有了用户的身份权限。

只要攻击者在微博上给用户发一个链接（指向攻击者的服务器），用户通过手机点击进入之后，手机当前页面的URL就通过Referer主动送到了攻击者的服务器上，攻击者自然就可以轻松拿到用户的gsid进而控制账号。

## TCP

---

### TCP粘包理解

TCP是\*\*面向字节\*\*的，即以流式传送，也就是连接建立后可以一直不停的发送，并\*\*没有明确的边界定义\*\*。而UDP是\*\*面向报文\*\*的，发送的时候是可以按照一个一个数据包去发送的，一个数据包就是一个明确的边界。因为TCP是流式传送，所以会开辟一个缓冲区，发送端往其中写入数据，每过一段时间就发送出去，因此有可能后续发送的数据（属于另一个包）和之前发送的数据同时存在缓冲区中并一起发送，造成粘包。接收端也有缓存，因此也存在粘包。

处理粘包的唯一方法就是\*\*制定应用层的数据通讯协议\*\*，通过协议来规范现有接收的数据是否满足消息数据的需要。在应用中处理粘包的基础方法主要有两种分别是\*\*以4节字描述消息大小\*\*或\*\*用结束符\*\*，实际上也有两者相结合的如HTTP，redis的通讯协议等。

### Nagle算法理解

Nagle算法为福特航空和通信公司1984年定义的\*\*TCP拥塞控制方法\*\*。

从键盘输入的一个字符，占用一个字节，可能在传输上造成41字节的包，其中包括1字节的有用信息和40字节的首部数据。这种情况转变成了4000%的消耗，且这些小包同样都需要经过ACK等。这样的情况对于轻负载的网络来说还是可以接受的，但是重负载的网络就受不了了，\*\*会导致网络由于太多的包而过载\*\*。

事实上，Nagle算法所谓的提高网络利用率只是它的一个副作用，\*\*Nagle算法的主旨在于避免发送大量的小包\*\*。Nagle算法并没有阻止发送小包，它只是阻止了发送大量的小包！

\*\*Nagle算法的基本定义是任意时刻，最多只能有一个未被确认的小段\*\*。 所谓`小段`，指的是小于MSS尺寸的数据块，所谓`未被确认`，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。Nagle算法会在TCP程序里添加两行代码，在未确认数据发送的时候让发送器把数据送到缓存里。任何数据随后继续\*\*直到得到明确的数据确认或者直到攒到了一定数量的数据了再发包\*\*。

默认情况下，发送数据采用Nagle算法。这样\*\*虽然提高了网络吞吐量，但是实时性却降低了\*\*，在一些交互性很强的应用程序来说是不允许的，使用`TCP\_NODELAY`选项可以禁止Nagle 算法。

### TCP同时打开，同时关闭

\*\*同时打开\*\*

两个应用程序同时执行主动打开。每一端都发送一个SYN，并传递给对方，且每一端都使用对端所知的端口作为本地端口。例如：

主机a中一应用程序使用7777作为本地端口，并连接到主机b 8888端口做主动打开。

主机b中一应用程序使用8888作为本地端口，并连接到主机a 7777端口做主动打开。

\*\*tcp协议在遇到这种情况时，只会打开一条连接\*\*。

这个连接的建立过程需要4次数据交换，而一个典型的连接建立只需要3次交换（即3次握手）

但多数伯克利版的tcp/ip实现并不支持同时打开。

![image](/images/tech/net\_10.png)

\*\*同时关闭\*\*

如果应用程序同时发送FIN，则在发送后会首先进入FIN\_WAIT\_1状态。在收到对端的FIN后，回复一个ACK，会进入CLOSING状态。在收到对端的ACK后，进入TIME\_WAIT状态。这种情况称为同时关闭。

同时关闭也需要有4次报文交换，与典型的关闭相同。

![image](/images/tech/net\_11.png)

## 其他网络协议

---

### OAuth2.0工作过程理解

OAuth是一个关于授权（authorization）的开放网络标准，目前的版本是2.0版。其作用就是让"客户端"（第三方应用）安全可控地获取"用户"的授权，与"服务提供商"（平台，比如微信）进行互动。

OAuth\*\*在"客户端"与"服务提供商"之间，设置了一个授权层\*\*（authorization layer）。"客户端"不能直接登录"服务提供商"，只能登录授权层，以此将用户与客户端区分开来。"客户端"登录授权层所用的令牌（token），与用户的密码不同。\*\*用户可以在登录的时候，指定授权层令牌的权限范围和有效期\*\*。"客户端"登录授权层以后，"服务提供商"根据令牌的权限范围和有效期，向"客户端"开放用户储存的资料。

\*\*OAuth 2.0的运行流程\*\*

![image](/images/tech/net\_7.png)

客户端的授权模式（步骤B）

\*\*OAuth 2.0的四种授权方式\*\*

\*\*1.授权码模式（authorization code） \*\*

适用于有server端的应用授权，是功能最完整、流程最严密的授权模式。它的特点就是\*\*通过客户端的后台服务器，与"服务提供商"的认证服务器进行互动\*\*。

\* （A）用户访问客户端，后者将前者导向认证服务器。

\* （B）用户选择是否给予客户端授权。

\* （C）假设用户给予授权，认证服务器将用户导向客户端事先指定的"重定向URI"（redirection URI），同时附上一个授权码。

\* （D）客户端收到授权码，附上早先的"重定向URI"，向认证服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。

\* （E）认证服务器核对了授权码和重定向URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

即：用一个URI去申请，获得用户授权后得到一个对应该URI的授权码。之后就可以用该URI+对应的授权码来获取一个令牌，之后就可以使用该令牌来通过授权层。

\*\*2.隐式授权（implicit）\*\*

适用于通过客户端访问的应用授权，不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了"授权码"这个步骤，因此得名。\*\*所有步骤在浏览器中完成\*\*，令牌对访问者是可见的，且客户端不需要认证。

\* （A）客户端将用户导向认证服务器。

\* （B）用户决定是否给于客户端授权。

\* （C）假设用户给予授权，认证服务器将用户导向客户端指定的"重定向URI"，并在URI的Hash部分包含了访问令牌。

\* （D）浏览器向资源服务器发出请求，其中不包括上一步收到的Hash值。

\* （E）资源服务器返回一个网页(typically an HTML document with an embedded script)，其中包含的代码可以获取Hash值中的令牌。

\* （F）浏览器执行上一步获得的脚本，提取出令牌。

\* （G）浏览器将令牌发给客户端（客户端就可以凭借此令牌来获取数据）。

\*\*实例：\*\*

其中短暂停留的那个页面的url为：

https://www.zhihu.com/oauth/callback/login/qqconn?code=680726D150FF0B9DF2EBBE2EFEEEC0D4&state=7f13b99dc94e506e69ecb9ec83296eec

页面效果：

![image](/images/tech/net\_8.png)

页面代码：

![image](/images/tech/net\_9.png)

\*\*3.密码模式（resource owner password credentials）\*\*

用户向客户端提供自己的用户名和密码。客户端使用这些信息，向"服务商提供商"索要授权。这通常用在用户对客户端高度信任的情况下。

\*\*4.客户端模式（client credentials）\*\*

指客户端以自己的名义，而不是以用户的名义，向"服务提供商"进行认证。\*\*严格地说，客户端模式并不属于OAuth框架所要解决的问题\*\*。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求"服务提供商"提供服务，其实不存在授权问题。`

# 后端架构

---

---

## 性能

---

### 秒杀系统性能优化思路

将请求尽量拦截在系统上游，并且充分利用缓存。由上游至低层优化如下：

\* 1.前端（浏览器、APP）

控制实际往后端发送请求的数量，如用户点击“查询”后，将按钮置灰，禁止用户在短时间内重复提交。

\* 2.站点层（访问后端数据，拼写html返回）

\*\*对uid进行请求计数和去重，比如5秒内只准透过一个请求\*\*（可以使用redis设置过期时间实现）。缺点是当有多台机器时（此时相当于5s内限制n个访问），数据可能不准（脏读,但数据库层面真实数据是没问题的）。

假设有海量真实的对站点层的请求，可以通过增加机器来扩容，实在不行只能抛弃部分请求（返回稍后再试），原则是要保护系统，不能让所有用户都失败；

\* 3.服务层（提供数据访问）

对于读请求，使用缓存。

对于写请求，使用请求队列（队列成本很低），每次只透有限的写请求（如总票数）去数据层，如果均成功，再放下一批。可以不用统一一个队列，这样的话每个服务透过更少量的请求（总票数/服务个数），这样简单。统一一个队列又复杂了。对于失败的处理无需重放，返回用户查询失败或者下单失败，架构设计原则之一是fail fast。

\* 4.数据层（数据库、缓存）

经过以上步骤，到数据库层的请求已经有限。

此外还可以做一些业务规则上的优化，如：12306分时分段售票、数据粒度优化（如只展示有、无，而不是具体的数量）、业务逻辑异步（先创建订单，但是状态为未支付，如果超时仍未支付，则恢复库存）。

## 安全

---

# 算法与设计模式

---

---

## 基础算法

---

### 选择排序（C）

首先找到最小元素置于起始位置，再从剩余元素中继续寻找最小者放到已排序序列末尾，依次类推（不稳定排序）

```c++

void SelectionSort(int arr[],int len){

int i,j,min,tmp;

for(i = 0; i < len; i++){

min = i;

for(j = i; j < len; j++){

if(arr[j] < arr[min]){

min = j;

}

}

tmp = arr[i];

arr[i] = arr[min];

arr[min] = tmp;

}

}

```

### 插入排序（C）

对于未排序数据在已排序序列中从后向前扫描，找到相应位置并插入（稳定排序）

```c++

void InsertionSort(int arr[],int len){

int i,j,tmp;

for(i = 1; i < len; i++){

tmp = arr[i];

for(j = i; j>0 && arr[j-1]>tmp; j--){

arr[j] = arr[j-1];

arr[j-1] = tmp;

}

}

}

```

### 冒泡排序（C）

（稳定排序）

```c

void BubbleSort(int arr[],int len){

int i,j,tmp;

for(i = 0; i < len; i++){

for(j = i+1; j < len; j++){

if(arr[j] < arr[i]){

tmp = arr[j];

arr[j] = arr[i];

arr[i] =tmp;

}

}

}

}

```

### 希尔排序（C）

先取一个正整数d1 < N，把所有相隔d1的元素放一组，共d1组，组内进行直接插入排序，再取d2 < d1，重复上述步骤，直至d=1.`只要最终步长为1，任何步长序列都可以`，当步长为1时，算法即为插入排序。（不稳定排序）

```c++

void ShellSort(int arr[],int len){

int i,j,incr,tmp;

// 14,7,3,1

for(incr = len/2; incr > 0; incr /= 2){

for(i = incr; i < len; i++){

tmp = arr[i];

for(j = i; j >= incr; j -= incr){

if(tmp < arr[j-incr]){

arr[j] = arr[j-incr];

}

else{

break;

}

}

arr[j] = tmp;

}

}

}

```

### 快速排序（C）

（不稳定排序）

```c++

void QuickSort(int a[],int low,int high){

int i = low;

int j = high;

int temp = a[i];

if( low < high){

while(i < j) // 若条件为i<=j，则将所有判定都加等号则会发生死循环

{

while((a[j] >= temp) && (i < j)){

j--;

}

a[i] = a[j];

while((a[i] <= temp) && (i < j)){

i++;

}

a[j]= a[i];

}

a[i] = temp;

QuickSort(a,low,i-1);

QuickSort(a,j+1,high);

}

}

```

### 归并排序（C）

（稳定排序）

```c++

void Merge( int arr[], int tmpArray[], int lBegin, int rBegin, int rEnd )

{

int i, lEnd, len, tmpPos;

lEnd = rBegin - 1;

tmpPos = lBegin;

len = rEnd - lBegin + 1;

/\* main loop \*/

while( lBegin <= lEnd && rBegin <= rEnd ){

if( arr[ lBegin ] <= arr[ rBegin ] ){

tmpArray[ tmpPos++ ] = arr[ lBegin++ ];

}

else{

tmpArray[ tmpPos++ ] = arr[ rBegin++ ];

}

}

while( lBegin <= lEnd ){

tmpArray[ tmpPos++ ] = arr[ lBegin++ ];

}

while( rBegin <= rEnd ){

tmpArray[ tmpPos++ ] = arr[ rBegin++ ];

}

for( i = 0; i < len; i++, rEnd-- ){

arr[ rEnd ] = tmpArray[ rEnd ];

}

}

void MSort( int arr[ ], int tmpArray[ ], int left, int right )

{

int mid;

if( left < right )

{

mid = ( left + right ) / 2;

MSort( arr, tmpArray, left, mid );

MSort( arr, tmpArray, mid + 1, right );

Merge( arr, tmpArray, left, mid + 1, right );

}

}

void MergeSort( int arr[ ], int len )

{

int \*tmpArray;

tmpArray = malloc( len \* sizeof( int ) );

if( tmpArray != NULL )

{

MSort( arr, tmpArray, 0, len - 1 );

free( tmpArray );

}

else

printf( "No space for tmp array!!!" );

}

```

### 堆排序（C）

（不稳定排序）

```c++

#define LeftChild(i) (2\*(i) + 1)

//对数组A中以下标为i的元素作为根，大小为len的元素序列构成的堆进行堆调整，使该根节点放到合适的位置

void Sink(int arr[],int i,int len){

int child,tmp;

for(tmp = arr[i]; LeftChild(i)<len; i=child){

child = LeftChild(i);

if( child != len-1 && arr[child+1] > arr[child]){

child++;

}

if( tmp < arr[child] ){

arr[i] = arr[child];

}

else{

break;

}

}

arr[i] = tmp;

}

void HeapSort(int arr[],int len){

int i,tmp;

for( i = len/2; i >= 0; i-- ){ // BuildHeap从下往上建堆

Sink( arr, i, len );

}

for( i = len - 1; i > 0; i-- ){

/\* DeleteMax \*/

tmp = arr[0];

arr[0] = arr[i];

arr[i] = tmp;

Sink( arr, 0, i );

}

}

```

### 计数排序（C）

（稳定排序）

```c++

void CountingSort(int arr[],int len){

int i, min, max;

min = max = arr[0];

// 找出范围

for(i = 1; i < len; i++) {

if (arr[i] < min)

min = arr[i];

else if (arr[i] > max)

max = arr[i];

}

int range = max - min + 1;

int \*count = (int\*)malloc(range \* sizeof(int));

for(i = 0; i < range; i++){

count[i] = 0;

}

for(i = 0; i < len; i++){

count[ arr[i] - min ]++;

}

int j, z = 0;

for(i = min; i <= max; i++){

for(j = 0; j < count[ i - min ]; j++){

arr[z++] = i;

}

}

free(count);

}

```

### 二分查找法（C）

```c

int HalfSearch(int arr[], int low, int high, int num){

int mid;

mid = (low+high) / 2;

if( (low>=high) && (arr[mid]!=num) ){

return -1;

}

else{

if( arr[mid]==num ){

return mid;

}

else if( arr[mid]>num ){

high = mid-1;

}

else{

low = mid+1;

}

return HalfSearch(arr,low,high,num);

}

}

```

### 前序遍历二叉树

遍历二叉树的算法中基本操作是访问结点，因此，`无论是哪种次序的遍历，对有n个结点的二叉树，其时间复杂度均为O(n)` 。

递归算法:

```c

void PreorderTraverse(BTNode \*T){

if( T!=NULL ){

visit(T->data) ; // 访问根结点

PreorderTraverse(T->Lchild) ;

PreorderTraverse(T->Rchild) ;

}

}

```

非递归算法:

设T是指向二叉树根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T；

⑴ 访问p所指向的结点；

⑵ q=p->Rchild ，若q不为空，则q进栈；

⑶ p=p->Lchild ，若p不为空，转(1)，否则转(4)；

⑷ 退栈到p ，转(1)，直到栈空为止。

```c++

#define MAX\_NODE 50

void PreorderTraverse( BTNode \*T){

BTNode \*Stack[MAX\_NODE] , \*p=T, \*q ;

int top=0 ;

if (T==NULL){

printf(“ Binary Tree is Empty!\n”) ;

}

else {

do{

visit( p-> data ) ;

q=p->Rchild ;

if ( q!=NULL ){

Stack[++top]=q ;

}

p=p->Lchild ;

if (p==NULL){

p=Stack[top--] ;

}

}

while (p!=NULL) ;

}

}

```

### 中序遍历二叉树

递归算法

```c

void InorderTraverse(BTNode \*T){

if (T!=NULL){

InorderTraverse(T->Lchild) ;

visit(T->data) ; // 访问根结点

InorderTraverse(T->Rchild) ;

}

}

```

中序遍历二叉树（非递归算法）

设T是指向二叉树根结点的指针变量，非递归算法是：

若二叉树为空，则返回；否则，令p=T

⑴ 若p不为空，p进栈， p=p->Lchild ；否则(即p为空)，退栈到p，访问p所指向的结点；

⑵ p=p->Rchild ，转(1)；

直到栈空为止。

```c

#define MAX\_NODE 50

void InorderTraverse( BTNode \*T){

BTNode \*Stack[MAX\_NODE] ,\*p=T ;

int top=0 , bool=1 ;

if (T==NULL){

printf(“ Binary Tree is Empty!\n”) ;

}

else{

do{

while (p!=NULL){

stack[++top]=p ;

p=p->Lchild ;

}

if (top==0){

bool=0 ;

}

else{

p=stack[top--] ;

visit( p->data ) ;

p=p->Rchild ;

}

} while (bool!=0) ;

}

}

```

### 后序遍历二叉树

递归算法

```c++

void PostorderTraverse(BTNode \*T){

if (T!=NULL) {

PostorderTraverse(T->Lchild) ;

PostorderTraverse(T->Rchild) ;

visit(T->data) ; // 访问根结点

}

}

```

设T是指向根结点的指针变量，后序遍历二叉树的非递归算法是：

若二叉树为空，则返回；否则，令p=T；

⑴ 第一次经过根结点p，不访问： p进栈S1 ， tag 赋值0，进栈S2，p=p->Lchild 。

⑵ 若p不为空，转(1)，否则，取状态标志值tag；

⑶ 若tag=0：对栈S1，不访问，不出栈；修改S2栈顶元素值(tag赋值1) ，取S1栈顶元素的右子树，即p=S1[top]->Rchild ，转(1)；

⑷ 若tag=1：S1退栈，访问该结点；

直到栈空为止。

### 层序遍历二叉树（C）

```c++

#define MAX\_NODE 50

void LevelorderTraverse( BTNode \*T){

BTNode \*Queue[MAX\_NODE] ,\*p=T ;

int front=0 , rear=0 ;

if (p!=NULL){

Queue[++rear]=p; // 根结点入队

// 当队列不为空时

while (front < rear){

p = Queue[++front];

visit( p->data );

if (p->Lchild!=NULL){

Queue[++rear]=p; // 左结点入队

}

if (p->Rchild!=NULL){

Queue[++rear]=p; // 右结点入队

}

}

}

}

```

### 判断二叉树是否相等（C）

```c++

typedef struct \_TreeNode{

char c;

TreeNode \*leftchild;

TreeNode \*rightchild;

}TreeNode;

// A、B两棵树相等当且仅当RootA->c==RootB-->c,而且A和B的左右子树相等或者左右互换相等。

int CompTree(TreeNode\* tree1,TreeNode\* tree2){

if( tree1==NULL && tree2 == NULL ){

Return 0;

}

if( tree1 == NULL || tree2 == NULL ){

return 1;

}

if( tree1->c != tree2->c){

return 1;

}

if( CompTree(tree1->leftchild, tree2->leftchild) == 0 && CompTree(tree1->rightchild, tree2->rightchild) == 0 ){

return 0;

}

if( CompTree(tree1->leftchild, tree2->rightchild) == 0 && CompTree(tree1->rightchild, tree2->leftchild) == 0 ){

return 0;

}

}

```

由于需要比较的状态是两棵树的任意状态，而二叉树上的每一个节点的左右子节点都可以交换，因此一共需要对比2^n种状态。算法复杂度是O（2^n）

### 求二叉树的叶子节点数（C）

```c++

#define MAX\_NODE 50

int search\_leaves( BTNode \*T){

BTNode \*Stack[MAX\_NODE] ,\*p=T;

int top=0, num=0;

if(T!=NULL){

stack[++top]=p ;

while( top > 0 ){

p = stack[top--] ;

if( p->Lchild==NULL && p->Rchild==NULL ){

num++ ;

}

if( p->Rchild != NULL ){

stack[++top]=p->Rchild;

}

if(p->Lchild != NULL ){

stack[++top]=p->Lchild;

}

}

}

return(num) ;

}

```

### 求二叉树的深度（C）

```c++

#define MAX\_NODE 50

int search\_depth( BTNode \*T){

BTNode \*Stack[MAX\_NODE] ,\*p=T;

int front=0 , rear=0, depth=0, level ;

// level总是指向访问层的最后一个结点在队列的位置

if (T!=NULL){

Queue[++rear]=p; // 根结点入队

level=rear ; // 根是第1层的最后一个节点

while (front < rear){

p=Queue[++front];

if (p->Lchild != NULL){

Queue[++rear]=p; // 左结点入队

}

if (p->Rchild!=NULL){

Queue[++rear]=p; // 右结点入队

}

if (front == level){

// 正访问的是当前层的最后一个结点

depth++ ;

level=rear ;

}

}

}

}

```

### 求最大的子序 列和的联机算法（C）

```c++

int MaxSubSequenceSum(const int arr[],int len){

int tmpSum, maxSum, j;

tmpSum = maxSum = 0;

for( j=0; j<len; j++ ){

tmpSum += arr[j];

if(tmpSum > maxSum){

maxSum = tmpSum;

}

else if(tmpSum < 0){

tmpSum = 0;

}

}

return maxSum;

}

```

### 分析排序算法时间空间复杂度和各自的稳定性

![image](/images/tech/algo.png)

### 写内存拷贝

```c++

void \* memcpy (void \* dst, const void \* src, size\_t count){

void \* ret = dst;

while (count--) {

\*(char \*)dst = \*(char \*)src;

dst = (char \*)dst + 1;

src = (char \*)src + 1;

}

return(ret);

}

```

### 图的广度优先遍历（C）

(1)邻接表表示图的广度优先搜索算法

```c++

// 以vk为源点对用邻接表表示的图G进行广度优先搜索

void BFS(ALGraph\*G，int k){

int i;

CirQueue Q; //须将队列定义中DataType改为int

EdgeNode \*p;

InitQueue(&Q); //队列初始化

printf("visit vertex：％e",G->adjlist[k].vertex); //访问源点vk

visited[k]=TRUE;

EnQueue(&Q，k); //vk已访问，将其人队。（实际上是将其序号人队）

while(!QueueEmpty(&Q)){ //队非空则执行

i=DeQueue(&Q); //相当于vi出队

p=G->adjlist[i].firstedge; //取vi的边表头指针

while(p){ //依次搜索vi的邻接点vj(令p->adjvex=j)

if(!visited[p->adivex]){ //若vj未访问过

printf("visitvertex：％c",C->adjlistlp->adjvex].vertex); //访问vj

visited[p->adjvex]=TRUE;

EnQueue(&Q，p->adjvex);//访问过的vj人队

}//endif

p=p->next; //找vi的下一邻接点

}//endwhile

}//endwhile

}//end of BFS

```

（2）邻接矩阵表示的图的广度优先搜索算法

```c++

// 以vk为源点对用邻接矩阵表示的图G进行广度优先搜索

void BFSM(MGraph \*G，int k){

int i,j;

CirQueue Q;

InitQueue(&Q);

printf("visit vertex:％c",G->vexs[k]); //访问源点vk

visited[k]=TRUE;

EnQueue(&Q,k);

while(!QueueEmpty(&Q)){

i=DeQueue(&Q); //vi出队

//依次搜索vi的邻接点vj

for(j=0;j<G->n;j++){

if(G->edges[i][j]==1&&!visited[j]){ //vi未访问

printf("visit vertex:％c"，G->vexs[j]);//访问vi

visited[j]=TRUE;

EnQueue(&Q,j);//访问过的vi人队

}

}

}//endwhile

}//BFSM

```

对于具有n个顶点和e条边的无向图或有向图，每个顶点均入队一次。广度优先遍历(BFSTraverse)图的时间复杂度和DFSTraverse算法相同。

当图是连通图时，BFSTraverse算法只需调用一次BFS或BFSM即可完成遍历操作，此时BFS和BFSM的时间复杂度分别为O(n+e)和0(n2)。

### 图的深度优先遍历（C）

（1）深度优先遍历算法

```c++

typedef enum{FALSE，TRUE} Boolean; // FALSE为0，TRUE为1

Boolean visited[MaxVertexNum]; // 访问标志向量是全局量

// 深度优先遍历以邻接表表示的图G，而以邻接矩阵表示G时，算法完全与此相同

void DFSTraverse(ALGraph \*G){

int i;

for(i=0;i<G->n;i++){

visited[i]=FALSE; //标志向量初始化

}

for(i=0; i<G->n; i++){

if(!visited[i]){ //vi未访问过

DFS(G，i)； //以vi为源点开始DFS搜索

}

}

}

```

（2）邻接表表示的深度优先搜索算法

```c++

void DFS(ALGraph \*G，int i){

// 以vi为出发点对邻接表表示的图G进行深度优先搜索

EdgeNode \*p;

printf("visit vertex：％c"，G->adjlist[i].vertex); //访问顶点vi

visited[i]=TRUE; //标记vi已访问

p=G->adjlist[i].firstedge; // 取vi边表的头指针

while(p){ //依次搜索vi的邻接点vj，这里j=p->adjvex

if (!visited[p->adjvex]){// 若vi尚未被访问

DFS(G，p->adjvex);//则以Vj为出发点向纵深搜索

}

p=p->next; // 找vi的下一邻接点

}

}

```

（3）邻接矩阵表示的深度优先搜索算法

```c++

void DFSM(MGraph \*G，int i){

// 以vi为出发点对邻接矩阵表示的图G进行DFS搜索，设邻接矩阵是0,l矩阵

int j;

printf("visit vertex：％c"，G->vexs[i]); //访问顶点vi

visited[i]=TRUE;

for(j=0;j<G->n;j++){ //依次搜索vi的邻接点

if(G->edges[i][j]==1&&!visited[j]){

DFSM(G，j)//(vi，vj)∈E，且vj未访问过，故vj为新出发点

}

}

}

```

对于具有n个顶点和e条边的无向图或有向图，遍历算法DFSTraverse对图中每顶点至多调用一次DFS或DFSM。从DFSTraverse中调用DFS(或DFSM)及DFS(或DFSM)内部递归调用自己的总次数为n。

当访问某顶点vi时，DFS(或DFSM)的时间主要耗费在从该顶点出发搜索它的所有邻接点上。用邻接矩阵表示图时，其搜索时间为O(n)；用邻接表表示图时，需搜索第i个边表上的所有结点。因此，对所有n个顶点访问，在邻接矩阵上共需检查n2个矩阵元素，在邻接表上需将边表中所有O(e)个结点检查一遍。

所以，DFSTraverse的时间复杂度为O(n2) （调用DFSM）或0(n+e)（调用DFS）。

### 图的邻接矩阵存储结构形式说明（C）

```c++

#define MaxVertexNum l00

typedef struct{

char vexs[MaxVertexNum]; // 顶点表

int edges[MaxVertexNum][MaxVertexNum]; // 邻接矩阵，可看作边表

int n,e; // 图中当前的顶点数和边数

}MGragh;

```

### 图的邻接表的形式说明及其建表算法（C）

对图的每个顶点建立一个单链表（n个顶点建立n个单链表），第i个单链表中的结点包含顶点Vi的所有邻接顶点。又称链接表。

（1）邻接表的形式说明

```c++

// 边表结点

typedef struct node{

int adjvex; // 邻接点域

struct node \*next; // 链域

// 若要表示边上的权，则应增加一个数据域

}EdgeNode;

// 顶点表结点

typedef struct vnode{

int vertex; // 顶点域

EdgeNode \*firstedge; // 边表头指针

}VertexNode;

typedef VertexNode AdjList[MaxVertexNum]; //AdjList是邻接表类型

```

（2）建立无向图的邻接表算法

```c++

// 建立无向图的邻接表表示

void CreateALGraPh(ALGrahp \*G){

int i,j,k;

EdgeNode \*s；

scanf("％d％d"，&G->n，&G->e); // 读入顶点数和边数

for(i=0;i<G->n;i++){

//建立顶点表

G->adjlist[i].vertex=getchar(); //读入顶点信息

G->adjlist[i].firstedge=NULL;//边表置为空表

}

for(k=0; k<G->e; k++){

//建立边表

scanf("％d％d",&i,&j);否则读入边(vi，vj)的顶点对序号

s=(EdgeNode \*)malloc(sizeof(EdgeNode)); //生成边表结点

s->adjvex=j; //邻接点序号为j

s->next=G->adjlist[i].firstedge;

G->adjlist[i].firstedge=s; //将新结点\*s插入顶点vi的边表头部

s=(EdgeNode \*)malloc(sizeof(EdgeNode));

s->adjvex=i; //邻接点序号为i

s->next=G->adjlist[j].firstedge;

G->adjlistk[j].firstedge=s; //将新结点\*s插入顶点vj的边表头部

}//end for

}CreateALGraph

```

### 排序相关问题

1.基于比较的排序，其复杂度最佳是多少？

2.快排排序、归并排序，其复杂度是多少？

3.既然快排O(n^2) > O(nlogn), 为什么实际应用中，快排的表现经常优于归并排序？

4.存在O(n)级别的排序算法么？

答案：

1.O(nlogn)

2.快排O(n^2)，归并排序O(nlogn)

3.inplace、cache性能好

4.存在，非比较排序。如counting sort, radix sort。

（最基础的问题，必须正确。）

### 说明链表和数组作为数据的不同组织形式，各自的优缺点

数组，在内存上给出了连续的空间。链表，内存地址上可以是不连续的，每个链表的节点包括原来的内存和下一个节点的信息(单向的一个，双向链表的话，会有两个)。

\*\*数组优于链表的\*\*

1.内存空间占用的少，因为链表节点会附加上一块或两块下一个节点的信息。但是数组在建立时就固定了。所以也有可能会因为建立的数组过大或不足引起内存上的问题。

2.数组内的数据可随机访问，但链表不具备随机访问性。这个很容易理解，数组在内存里是连续的空间，比如如果一个数组地址从100到200，且每个元素占用两个字节，那么100-200之间的任何一个偶数都是数组元素的地址，可以直接访问。链表在内存地址可能是分散的。所以必须通过上一节点中的信息找能找到下一个节点。

3.查找速度。

\*\*链表优于数组的\*\*

1.插入与删除的操作。如果数组的中间插入一个元素，那么这个元素后的所有元素的内存地址都要往后移动。删除的话同理。只有对数据的最后一个元素进行插入删除操作时，才比较快。链表只需要更改有必要更改的节点内的节点信息就够了。并不需要更改节点的内存地址。

2.内存地址的利用率方面。不管你内存里还有多少空间，如果没办法一次性给出数组所需的要空间，那就会提示内存不足，磁盘空间整理的原因之一在这里。而链表可以是分散的空间地址。

3.链表的扩展性比数组好。因为一个数组建立后所占用的空间大小就是固定的，如果满了就没法扩展，只能新建一个更大空间的数组;而链表不是固定的，可以很方便的扩展。

### 链表反转（C）

```c++

struct Item{

char c;

Item \*next;

};

Item \*Reverse( Item \*x ){

Item \*prev = NULL,\*curr = x;

while ( curr ){

Item \*next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

return prev;

}

int main(){

Item \*x,

d = {"d", Null},

c = {"c", &d},

b = {"b", &c},

a = {"a", &b};

x = Reverse( &a );

}

```

## 设计模式与重构

---

### 面向对象编程应该遵守的几个原则

\* 依赖倒换原则

要针对接口编程，不要对实现编程：

\* 1.高层模块不应该依赖于低层模块。

\* 2.抽象不应该依赖细节，细节应该依赖抽象。

![image](https://github.com/woojean/woojean.github.io/blob/master/assets/images/dm\_1.png)

\* 单一职责原则

就一个类而言，应该仅有一个引起它变化的原因。如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。比如设计游戏显示区域，将绝对坐标改成相对坐标，实现程序逻辑和界面的分离。

\* 开放-封闭原则

软件实体（类、模块、函数等等）应该可以扩展，但是不可修改。

面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。

最初编写代码时，假设变化不会发生，当变化发生时，就创建抽象来隔离以后发生的同类变化。

开发人员应该仅对程序中呈现出频繁变化的那部分作出抽象，然而对于程序中的每个部分都刻意地进行抽象同样不是一个好主意。

\* 迪米特法则

如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。在类的结构设计上，每一个类都应当尽量降低成员的访问权限。其根本思想是强调了类之间的松耦合，类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。

\* 里氏代换原则

一个软件实体，如果使用的是一个父类的话，那么一定适用于其子类，而且它觉察不出父类对象和子类对象的区别。正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展。

### 依赖注入、控制反转

IoC（Inversion of Control）控制反转

DI（Dependency Injection）依赖注入

\*\*DI是IoC的一种具体实现\*\*，另一种主要的实现方式是服务定位器（Service Locator）。

没有IoC的时候，常规的A类使用C类的示意图：

![image](/images/tech/img\_1.png)

有IoC的时候，A类不再主动去创建C，而是被动等待，等待IoC的容器获取一个C的实例，然后反向地注入到A类中。

![image](/images/tech/img\_2.png)

## 开放思路

---

### 一个数组A[N]，包含取值为[1,N]的元素，请判断是否有重复元素

解法：

1、Sum(1…N)!=sum(A[0],A[N-1])则重复

2、hash记数法

3、排序后再判重

### 为手机的T9输入法设计一个索引

提供一个包含所有英文单词的字典，为手机的T9输入法设计一个索引，例如输入4能够提示出g、h、i开头的英文单词（greate、hello、...），输入43能够提示出ge、he、id、if (hello...) 等词开通的英文单词。

思路：

使用trie树存储索引，但是如果使用普通的以字母形式组织的trie树检索非常麻烦。因此需要先将所有的英文单词改写成数字，例如：

Greate:473283

Hello:43556

然后组织成数字的trie树，所有的单词都挂在这棵trie树上。

### 内存受限情况下的海量字符串查找问题

方案1：可以估计每个文件的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件（记为a0,a1,...,a999）中。这样每个小文件的大约为300M。

遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为b0,b1,...,b999）。`这样处理后，所有可能相同的url都在对应的小文件（a0vsb0,a1vsb1,...,a999vsb999）中，不对应的小文件不可能有相同的url`。然后我们只要求出1000对小文件中相同的url即可。

求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set（STL）中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

方案2：如果允许有一定的错误率，可以使用`Bloom filter`，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。

Bloom filter：Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。`Bloom filter 采用的是哈希函数的方法，将一个元素映射到一个 m 长度的阵列上的一个点，当这个点是 1 时，那么这个元素在集合内，反之则不在集合内`。这个方法的`缺点就是当检测的元素很多的时候可能有冲突，解决方法就是使用 k 个哈希 函数对应 k 个点，如果所有点都是 1 的话，那么元素在集合内，如果有 0 的话，元素则不再集合内。随着元素的插入，Bloom filter 中修改的值变多，出现误判的几率也随之变大`，当新来一个元素时，满足其在集合内的条件，即所有对应位都是 1 ，这样就可能有两种情况，一是这个元素就在集合内，没有发生误判；还有一种情况就是发生误判，出现了哈希碰撞，这个元素本不在集合内。

### 内存有限情况下进行排序

有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

方案：顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。

如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。

对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用trie树/hash\_map等），并取出出现频率最大的100个词（可以用含100个结点的最小堆），并把100个词及相应的频率存入文件，这样又得到了5000个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

### 写出斐波那契数列的递归与迭代代码，并分析时间和空间复杂度

斐波那契数列指的是这样一个数列：1、1、2、3、5、8、13、21、……

用数学公式表示出来就是：

F（1）= 1，F（2）=1 (n=1,2)

F(n)=F(n-1)+ F(n-2) (n>2)

递归法：

Fib(1) = 1 [基本情况]

Fib(2) = 1 [基本情况]

对所有n > 1的整数：Fib(n) = (Fib(n-1) + Fib(n-2)) [递归定义]

关键代码：

```c++

if(n == 1|| n== 2){

return 1;

}

else{

return fib(n - 1) + fib(n - 2);

}

```

迭代法:

```c++

int f(int n){

int i, f1 = 1, f2 = 1, f3;

if(n<=0){

printf("输入错误.\n");

}

else if( n==1 || n==2 ){

printf("1");

}

else{

for( i=0; i < n-2; i++ ){

f3 = f1+f2; // f1 表示当前的值

f2=f1;

f1=f3;

}

printf("%d\n",f1);

}

}

```

### 分药问题

有5瓶药，药片形状大小一样，每瓶数量一样。其中一瓶是坏的，每片9mg，好的药片每片10mg。需要一次区分哪瓶是坏药，怎么做？

1.你有一个高灵敏度的天平，有任意规格砝码

给5个瓶依次编号，分别取出1、2、3、4、5片药，根据砝码不同的质量区分坏药瓶的序号

2.你有一个高灵敏度的天平，但没有砝码

给5个瓶依次编号，第5瓶放开不取，从1~4瓶中分别取出100、1、100、1片药，放置1、2和3、4分别于天平左右侧:

（1）看是否平衡，若平衡，5是坏的；如不平衡；

（2）左侧低右侧高，看灵敏天平的偏转角度，大3，小4；

（3）左侧高右侧低，看灵敏天平的偏转角度，大1，小2。

### 合并有序链表

递归算法所体现的“重复”一般有三个要求：

一是每次调用在规模上都有所缩小(通常是减半)；

二是相邻两次重复之间有紧密的联系，前一次要为后一次做准备(通常前一次的输出就作为后一次的输入)；

三是在问题的规模极小时必须用直接给出解答而不再进行递归调用，因而每次递归调用都是有条件的(以规模未达到直接解答的大小为条件)，无条件递归调用将会成为死循环而不能正常结束。

如何设计递归算法

1.确定递归公式

2.确定边界(终了)条件递归实现：

算法思想：

递归终止条件：若head1为空，返回head2指针（head）；若head2为空，返回head1指针（head）

递归过程：

1.若head1->data > head2->data; head 指针应该指向head2所指向的节点，而且head->next应该指向head1和head2->next两个链表的合成序列的头指针；

2.否则head 指针应该指向head1所指向的节点，而且head->next应该指向head->next和head2两个链表的合成序列的头指针；

实现代码：

```c++

#include <iostream>

using namespace std;

// 节点的类定义

class Node{

public:

int data;

Node \* next;

Node(int data){

this->data=data;

}

};

// 链表的类定义

class LinkedList{

public:

Node \* head;

// 用一个整形数组作为参数的构造函数

LinkedList(int array[]){

head=new Node(array[0]);

Node \* temp = head;

int i;

for(i=1;i<3;i++){

temp->next=new Node(array[i]);

temp=temp->next;

}

temp->next=NULL;

}

};

// 递归的合并两个有序链表

Node \* mergeLinkedList(Node \* head1, Node \* head2){

Node \*p=NULL;

if(head1==NULL && head2==NULL){

return p;

}

else if( head1==NULL ){

return head2;

}

else if( head2==NULL ){

return head1;

}

else{

if( head1->data < head2->data ){

p = head1;

p->next = mergeLinkedList( head1->next,head2 );

}

else{

p = head2;

p->next = mergeLinkedList( head1,head2->next );

}

return p;

}

}

// 打印链表的所有元素

void printList(Node \* head){

Node \* temp=head;

while(temp!=NULL){

cout<<temp->data<<" ";

temp=temp->next;

}

}

int main(){

int array1[3]={2,5,8};

int array2[3]={1,6,7};

// 构造两个有序链表--list1和list2

LinkedList list1(array1);

LinkedList list2(array2);

// 递归的将这两个有序链表合并成一个有序链表

Node \* new\_head = mergeLinkedList(list1.head, list2.head);

// 打印有序链表

printList(new\_head);

return 0;

}

```

### 向单链表中满足条件的位置插入一个元素

通常是向有序单链表中插入一个新元素结点，使得插入后仍然有序。其插入过程为：

(1)为新元素动态分配结点并赋值；

(2)若单链表为空或者新元素小于表头结点的值，则应把新元素结点插入到表头并返

(3)从表头开始顺序查找新元素的插入位置，在查找过程中必须保留当前结点的前驱g指针，以便插入新结点时使用；

(4)在插入位置上完成插入操作，即把新结点插入到当前结点和其前驱结点之间。

### 在一个有序数组中，有些元素重复出现。输入一个数值，求此值在数组中重复的次数

思路有两种:

1.upperbound() – lowerbound()

2.使用类似线段树的思想直接统计

iterator lower\_bound( const key\_type &key ): 返回一个迭代器，`指向键值>= key的第一个元素`。

iterator upper\_bound( const key\_type &key ):返回一个迭代器，`指向键值> key的第一个元素`。

例如：map中已经插入了1，2，3，4的话，如果lower\_bound(2)的话，返回的2，而upper\_bound（2）的话，返回的就是3

### 在2.5亿个整数中找出不重复的整数

在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。

方案1：采用`2-Bitmap`（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 \* 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

方案2：也可采用划分小文件的方法，然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

`BitMap算法`：来自于《编程珠玑》。所谓的Bit-map就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0 然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0×01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位置为1。

然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1。

然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。

### 多信号量生产者-消费者问题

桌上有一空盘，允许存放一只水果。爸爸可向盘中放苹果，也可向盘中放桔子，儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一只水果供吃者取用，请用P、V原语实现爸爸、儿子、女儿三个并发进程的同步。

分析 在本题中，爸爸、儿子、女儿共用一个盘子，盘中一次只能放一个水果。当盘子为空时，爸爸可将一个水果放入果盘中。若放入果盘中的是桔子，则允许儿子吃，女儿必须等待；若放入果盘中的是苹果，则允许女儿吃，儿子必须等待。本题实际上是生产者-消费者问题的一种变形。这里，生产者放入缓冲区的产品有两类，消费者也有两类，每类消费者只消费其中固定的一类产品。

解：在本题中，应设置三个信号量S、So、Sa，信号量S表示盘子是否为空，其初值为l；信号量So表示盘中是否有桔子，其初值为0；信号量Sa表示盘中是否有苹果，其初值为0。同步描述如下：

```

int S＝1;

int Sa＝0;

int So＝0;

main()

{

cobegin

father(); /\*父亲进程\*/

son(); /\*儿子进程\*/

daughter(); /\*女儿进程\*/

coend

｝

father()

{

while(1)

{

P(S);

将水果放入盘中;

if（放入的是桔子）V(So);

else V(Sa);

}

}

son()

{

while(1)

{

P(So);

从盘中取出桔子;

V(S);

吃桔子;

｝

}

daughter()

{

while(1)

{

P(Sa);

从盘中取出苹果;

V(S);

吃苹果;

｝

｝

```

### 大数相加问题

实现A+B=C,其中A、B位数超过100位

算法思想：大数使用字符串存储，每一个单元存储操作数的每一位，之后执行位相加。

基本思路：字符串反转、字符变数字、位运算、反序输出

C语言代码：

```c++

#include <stdio.h>

#include<string.h>

#define Max 101

void print(char sum[], int result\_len);

int bigNumAdd(char a[],char b[],char sum[]);

int main() {

char a[Max]={0};

char b[Max]={0};

char sum[Max]={0};

puts("input a:");

gets(a); /\* char\* gets(char\*buffer); 头文件stdio.h .gets(s)函数与scanf("%s",s)相似，但不完全相同，使用scanf("%s",s) 函数输入字符串时存在一个问题，就是如果输入了空格会认为字符串结束，空格后的字符将作为下一个输入项处理，但gets()函数将接收输入的整个字符串直到遇到换行为止 \*/

puts("input b:");

gets(b);

print(sum, bigNumAdd(a,b,sum));

return 0;

}

int bigNumAdd(char a[], char b[], char sum[]){

int i=0;

int c=0; // 表示进位

char m[Max]={0};

char n[Max]={0};

memset(sum,0,Max\*sizeof(char)); // 重要

// 字符串反转且字符串变数字

int lenA=strlen(a);

int lenB=strlen(b);

int result\_len = (lenA > lenB)?lenA:lenB;

for (i=0;i<lenA;i++){

m[i]=a[lenA-i-1]-'0';

}

for (i=0;i<lenB;i++){

n[i]=b[lenB-i-1]-'0';

}

// 按位运算

for (i=0;i<lenA||i<lenB;i++){

sum[i]=(m[i]+n[i]+c)%10+'0'; // 得到末位

c=(m[i]+n[i]+c)/10; // 得到进位

}

if (c){

result\_len++;// 最后一次有进位，长度+1

}

return result\_len;

}

void print(char sum[], int result\_len){

int j=0;

int i=0;

for(j=result\_len-1; j>=0; j--){

i++;

printf("%c",sum[j]);

}

puts("\n");

}

```

### 如何使用P、V操作来结局各种生产者-消费者问题？

PV操作由P操作原语和V操作原语组成（原语是不可中断的过程），对信号量进行操作，具体定义如下：

```

P（S）：①将信号量S的值减1，即S=S-1；

②如果S0，则该进程继续执行；否则该进程置为等待状态，排入等待队列。

V（S）：①将信号量S的值加1，即S=S+1；

②如果S>0，则该进程继续执行；否则释放队列中第一个等待信号量的进程。

```

PV操作的意义：我们用信号量及PV操作来实现进程的同步和互斥。PV操作属于进程的低级通信。

信号量（semaphore）的数据结构为一个值和一个指针，指针指向等待该信号量的下一个进程。信号量的值与相应资源的使用情况有关。当它的值大于0时，表示当前可用资源的数量；当它的值小于0时，其绝对值表示等待使用该资源的进程个数。注意，信号量的值仅能由PV操作来改变。

一般来说，信号量S>0时，S表示可用资源的数量。执行一次P操作意味着请求分配一个单位资源，因此S的值减1；当S<0时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去。而执行一个V操作意味着释放一个单位资源，因此S的值加1；若S0，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。

利用信号量和PV操作实现进程互斥：

```

进程P1 进程P2 …… 进程Pn

…… …… ……

P（S）； P（S）； P（S）；

临界区； 临界区； 临界区；

V（S）； V（S）； V（S）；

…… …… …… ……

```

其中信号量S用于互斥，初值为1。

使用PV操作实现进程互斥时应该注意的是：

（1）每个程序中用户实现互斥的P、V操作必须成对出现，先做P操作，进临界区，后做V操作，出临界区。若有多个分支，要认真检查其成对性。

（2）P、V操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。

（3）互斥信号量的初值一般为1。

利用信号量和PV操作实现进程同步

PV操作是典型的同步机制之一。用一个信号量与一个消息联系起来，当信号量的值为0时，表示期望的消息尚未产生；当信号量的值非0时，表示期望的消息已经存在。用PV操作实现进程同步时，调用P操作测试消息是否到达，调用V操作发送消息。

使用PV操作实现进程同步时应该注意的是：

（1）分析进程间的制约关系，确定信号量种类。在保持进程间有正确的同步关系情况下，哪个进程先执行，哪些进程后执行，彼此间通过什么资源（信号量）进行协调，从而明确要设置哪些信号量。

（2）信号量的初值与相应资源的数量有关，也与P、V操作在程序代码中出现的位置有关。

（3）同一信号量的P、V操作要成对出现，但它们分别在不同的进程代码中。

### 如何判断一个链表是否存在回路?

给指针加一个标志域，如访问过则置1.当遍历到标志为1的项说明有了回路。

定义2个指针，一快(fast)一慢(slow)，即：从头向后遍历过程中，每循环一次，快指针一次向后移动2个元素，慢指针移动一个元素，每次判断( fast==slow || slow==fast->nest ),如果成立，说明慢指针赶上了快指针，则为循环链表，否则，如果有一个指针到达NULL，则为单链表。

### 如何对递归程序进行时间复杂度分析？

例子：求N!。 这是一个简单的"累乘"问题，用递归算法也能解决。

```

n! = n \* (n - 1)! n > 1

0! = 1, 1! = 1 n = 0,1

```

因此，递归算法如下：

```c++

fact(int n) {

if(n == 0 || n == 1)

return 1;

else

return n \* fact(n - 1);

}

```

以n=3为例，看运行过程如下：

```

fact(3) ----- fact(2) ----- fact(1) ------ fact(2) -----fact(3)

------------------------------> ------------------------------>

递归 回溯

```

递归算法在运行中不断调用自身降低规模的过程，当规模降为1，即递归到fact(1)时，满足停止条件停止递归，开始回溯(返回调用算法)并计算，从fact(1)=1计算返回到fact(2);计算2\*fact(1)=2返回到fact(3)；计算3\*fact(2)=6，结束递归。

递归算法的分析方法比较多，最常用的便是迭代法。

迭代法的基本步骤是先将递归算法简化为对应的递归方程，然后通过反复迭代，将递归方程的右端变换成一个级数，最后求级数的和，再估计和的渐进阶。

```

<1> 例：n!

算法的递归方程为： T(n) = T(n - 1) + O(1);

迭代展开： T(n) = T(n - 1) + O(1)

= T(n - 2) + O(1) + O(1)

= T(n - 3) + O(1) + O(1) + O(1)

= ......

= O(1) + ... + O(1) + O(1) + O(1)

= n \* O(1)

= O(n)

这个例子的时间复杂性是线性的。

```

```

<2> 例：如下递归方程：

T(n) = 2T(n/2) + 2, 且假设n=2的k次方。

T(n) = 2T(n/2) + 2

= 2(2T(n/2\*2) + 2) + 2

= 4T(n/2\*2) + 4 + 2

= 4(2T(n/2\*2\*2) + 2) + 4 + 2

= 2\*2\*2T(n/2\*2\*2) + 8 + 4 + 2

= ...

= 2的(k-1)次方 \* T(n/2的(i-1)次方) + $(i:1~(k-1))2的i次方

= 2的(k-1)次方 + (2的k次方) - 2

= (3/2) \* (2的k次方) - 2

= (3/2) \* n - 2

= O(n)

这个例子的时间复杂性也是线性的。

```

```

<3> 例：如下递归方程：

T(n) = 2T(n/2) + O(n), 且假设n=2的k次方。

T(n) = 2T(n/2) + O(n)

= 2T(n/4) + 2O(n/2) + O(n)

= ...

= O(n) + O(n) + ... + O(n) + O(n) + O(n)

= k \* O(n)

= O(k\*n)

= O(nlog2n) //以2为底

一般地，当递归方程为`T(n) = aT(n/c) + O(n)`, T(n)的解为：

O(n) (a<c && c>1)

O(nlog2n) (a=c && c>1) //以2为底

O(nlogca) (a>c && c>1) //n的(logca)次方，以c为底

上面介绍的3种递归调用形式，比较常用的是第一种情况，第二种形式也有时出现，而第三种形式(间接递归调用)使用的较少，且算法分析比较复杂。下面举个第二种形式的递归调用例子。

```

```

<4> 递归方程为：T(n) = T(n/3) + T(2n/3) + n

为了更好的理解，先画出递归过程相应的递归树：

n --------> n

n/3 2n/3 --------> n

n/9 2n/9 2n/9 4n/9 --------> n

...... ...... ...... ....... ......

--------

总共O(nlogn)

累计递归树各层的非递归项的值，每一层和都等于n，从根到叶的最长路径是：

n --> (2/3)n --> (4/9)n --> (12/27)n --> ... --> 1

设最长路径为k，则应该有：(2/3)的k次方 \* n = 1

得到 k = log(2/3)n // 以(2/3)为底

于是 T(n) <= (K + 1) \* n = n (log(2/3)n + 1)

即 T(n) = O(nlogn)

由此例子表明，对于第二种递归形式调用，借助于递归树，用迭代法进行算法分析是简单易行的。

```

### 如果收到一个字符串型的浮点数，比如“1234.56”，如何变成浮点数

```c++

double atof(char s[]){

double val,power;

int i,sign;

for( i=0; isspace(s[i]); i++) //跳过空白

;

sign=(s[i]=='-') ? -1 : 1; //判断符号

if( s[i]=='+' || s[i]=='-' ){

i++;

}

for( val=0.0; isdigit(s[i]); i++){

val = 10.0\*val + (s[i]-'0'); // 此步骤也可用于求解“将一个字符串的整数变成整数”

}

if(s[i]=="."){

i++;

}

for(power=1.0;isdigit(s[i]);i++){

val=10.0\*val+(s[i]-'0');

power\*=10.0;

}

return sign\*val/power;

}

```

### 完成一个trim函数，将一个字符串两端的空格、回车、tab符号去掉

```c++

void trim( char \*str){

int i, j;

assert( str != NULL); // <assert.h>

/\*find the first non-space char's position \*/

for (i = 0; (str[i] == ' ' || str[i] == '\t') && str[i] != '\0'; i++)

;

/\*find the last non-space char's position \*/

for (j = strlen(str) - 1; (str[j] == ' ' || str[j] == '\t') && j; j--)

;

memmove(str, str + i, j - i); // < String.h >

str[j + 1] = '\0';

}

```

memmove用于从src拷贝count个字节到dest，如果目标区域和源区域有重叠的话，memmove能够保证源串在被覆盖之前将重叠区域的字节拷贝到目标区域中。但复制后src内容会被更改。当目标区域与源区域没有重叠则和memcpy函数功能相同。

### 定一棵二叉树的前序和中序遍历序列，重建这个二叉树

如先序为 124356， 中序为 421536

1．首先要面试者正确写出一个二叉树的先序和中序序列

2．然后询问根据先序和中序结果，是否可以重建出这棵二叉树？看面试者是否能够意识到二叉树节点内容相同的情况。

3．简化问题，假设二叉树节点只使用标号表示，没有重复，写程序根据先序和中序序列重构出二叉树。

4．分析：

先序中的第一个节点1是二叉树的根，找到该节点在中序中的位置，则1之前的为二叉树的左子树（42），1之后的为二叉树的右子树（536），然后根据先序中的对应子树24和356，递归重建即可。

重建二叉树：先序+中序（可以），后序+中序（可以），`先序+后序（不可以）`

### 尽可能的优化一段代码的性能

如下代码成为了系统的瓶颈，请尽可能的优化；要求找到优化的点和优化方案，分析说明原因。

代码如下：

```

#define M 10000

#define N 10000

#define L 3

int arr[L][M][N];

int xxx[M \* N];

int main()

{

// init arr and xxx first, omit

// start

for (int r=0; r<10000; ++r){

f(arr);

}

return 0;

}

void f( int arr[L][M][N] )

{

int k = 0;

for ( int m=0; m<M; m++ ){

for( int n=0; n<N; n++ ){

for ( int l=0; l<L; l++){

int ss = arr[l][m][0] + 1111;

int tmp = sss + power( log( arr[l][m][n] ), 3 );

arr[l][m][n] = a[l][m][n] + tmp;

xxx[k] += arr[l][m][n];

}

k = k + 1;

}

}

}

```

修改点：

1.power函数，可以直接写成a\*a\*a，这个优化效果与机器型号有关

2.改变数据结构，arr[L][M][N]修改成arr[M][N][L]，修改后的好处：

a) 内层循环的一些共有计算可以提前

b) 内层数据访问被连续存储，cache命中率极度提升（最重要也是最根本的优化点）

3.改变循环方式，for l, for m, for n；配合2，修改循环方式后，可以对l循环进行循环展开，减少分支预测失败

4.….

### 快排

这是一套题目的3个子问题，难度依次递进：

问题1:描述快排的基本思想并进行编码，以及一个典型的应用。

问题2:快排可以应用在链表上么？

问题3:除了快排，还有其他的排序可以在链表上达到O(lnN)的复杂度么？

解答1:典型应用：求第N大或者第N小的数。

解答2:`快排可以应用在链表上`。每次迭代需要两个指针，一个指向比pivot大的结点的链表，一个指向比pivot小的结点的链表，然后合并。

解答3:

还可以使用归并排序，逻辑比较复杂，需要考虑临时指针的使用，每个临时指针在过程中多次复用，否则会消耗大量的空间。具体参考stl中list的排序算法。

### 把两个链表合并成一个新的有序链表，返回头指针

递归终止条件：若head1为空，返回head2指针（head）；若head2为空，返回head1指针（head）

递归过程：

（1）若head1->data > head2->data; head 指针应该指向head2所指向的节点，而且head->next应该指向head1和head2->next两个链表的合成序列的头指针；

（2）否则head 指针应该指向head1所指向的节点，而且head->next应该指向head->next和head2两个链表的合成序列的头指针；

实现代码（C++）：

```c++

#include <iostream>

using namespace std;

/\*节点的类定义\*/

class Node{

public:

int data;

Node \* next;

Node(int data){

this->data=data;

}

};

/\*链表的类定义\*/

class LinkedList{

public:

Node \* head;

/\*用一个整形数组作为参数的构造函数\*/

LinkedList(int array[]){

head=new Node(array[0]);

Node \* temp=head;

int i;

for(i=1;i<3;i++){

temp->next=new Node(array[i]);

temp=temp->next;

}

temp->next=NULL;

}

};

/\*递归的合并两个有序链表\*/

Node \* mergeLinkedList(Node \* head1,Node \* head2){

Node \*p=NULL;

if(head1==NULL && head2==NULL){

return p;

}

else if(head1==NULL){

return head2;

}

else if(head2==NULL){

return head1;

}

else{

if(head1->data < head2->data){

p = head1;

p->next = mergeLinkedList(head1->next,head2);

}

else{

p = head2;

p->next = mergeLinkedList(head1,head2->next);

}

return p;

}

}

/\*打印链表的所有元素\*/

void printList(Node \* head){

Node \* temp=head;

while(temp!=NULL){

cout<<temp->data<<" ";

temp=temp->next;

}

}

int main(){

int array1[3]={2,5,8};

int array2[3]={1,6,7};

/\*构造两个有序链表--list1和list2\*/

LinkedList list1(array1);

LinkedList list2(array2);

/\*递归的将这两个有序链表合并成一个有序链表\*/

Node \* new\_head=mergeLinkedList(list1.head,list2.head);

/\*打印有序链表\*/

printList(new\_head);

return 0;

}

```

### 撞球问题

在水平光滑无限长的管道两端，有若干相向而行的小球，球的直径和管的直径相同，所有球的质量和速率均相同，球与球之间有一定间距。当两个相向而行的小球相遇时，发生完全弹性碰撞，即为一次碰撞。求下列情况下的碰撞总次数

（1）两端各3个球；

（2）一端6个球，一端3个球；

（3）一端M个球，一端N个球

答案：

（1）9；

（2）18；

（3）M\*N

思路:

对于（1）或（2），可以用穷举法计算，目的是引导推导出M\*N的规律

（2）可以从（1）中推导，即将（2）分成两组（1）来考虑

（3）有两种思路，简单的，可以考虑将（2）的分组考虑成18个1\*1组合，类推出M\*N；另一种思路，当发生一次碰撞时，可以看成两个小球碰撞后交换位置。

### 染色问题

现在平面上n条直线，n条直线将平面划分为若干个区域，现在用若干种颜色对这些区域染色，那么至少需要多少种颜色，才能使相邻区域（两个区域的公共部分只有一个点不能视为相邻）的颜色不同？

答案是两种（无论n条直线将平面划分后是怎样的）。首先是考虑一条直线的情况，显然可以用黑白这两种颜色就能满足要求。当已经将n-1条直线划分的区域染成黑白两种颜色时，那么增加一条新的直线时候，将直线左边的区域的颜色全部从黑变白，白变黑。而右边的区域的颜色保持不变，那么最终相邻区域的颜色还是保持不同。

### 海量字符串查找问题

有一个庞大的字符串数组，然后给你一个单独的字符串，让你从这个数组中查找是否有这个字符串并找到它，你会怎么做？

有一个方法最简单，老老实实从头查到尾，一个一个比较，直到找到为止...

`所谓Hash，一般是一个整数，通过某种算法，可以把一个字符串"压缩" 成一个整数`。当然，无论如何，一个32位整数是无法对应回一个字符串的，但在程序中，两个字符串计算出的Hash值相等的可能非常小。

是不是把第一个算法改进一下，改成逐个比较字符串的Hash值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个哈希表(Hash Table)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，例如1024，每一个Hash值通过取模运算 (mod) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置有没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的O(1)。

冲突解决：分离链接法，用链表解决冲突。

`一个好的hash函数`：

```c++

/\*key为一个字符串，nTableLength为哈希表的长度

\*该函数得到的hash值分布比较均匀\*/

unsigned long getHashIndex( const char \*key, int nTableLength )

{

unsigned long nHash = 0;

while (\*key)

{

nHash = (nHash<<5) + \*key++; // nHash = nHash\*32 + \*key; key++

}

return ( nHash % nTableLength );

}

```

### 海量文件统计问题

有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序。

还是典型的TOP K算法，解决方案如下：

方案1：

顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。

找一台内存在2G左右的机器，依次对用hash\_map(query, query\_count)来统计每个query出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的query\_cout输出到文件中。这样得到了10个排好序的文件。

对这10个文件进行归并排序（内排序与外排序相结合）。

方案2：

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash\_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案3：

与方案1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

### 海量日志数据，提取出某日访问百度次数最多的那个IP

首先是这一天，并且是访问百度的日志中的IP取出来，逐个写入到一个大文件中。注意到IP是32位的，最多有个2^32个IP。同样可以采用映射的方法，比如模1000，把整个大文件映射为1000个小文件，再找出每个小文中出现频率最大的IP（可以采用hash\_map进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这1000个最大的IP中，找出那个频率最大的IP，即为所求。

算法思想：`分而治之+Hash`

1.IP地址最多有2^32=4G种取值情况，所以不能完全加载到内存中处理；

2.可以考虑采用“分而治之”的思想，按照IP地址的Hash(IP)%1024值，把海量IP日志分别存储到1024个小文件中。这样，每个小文件最多包含4MB个IP地址；

3.对于每一个小文件，可以构建一个IP为key，出现次数为value的Hash map，同时记录当前出现次数最多的那个IP地址；

4.可以得到1024个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP；

### 生产者-消费者问题

在多道程序环境下，进程同步是一个十分重要又令人感兴趣的问题，而生产者-消费者问题是其中一个有代表性的进程同步问题。下面我们给出了各种情况下的生产者-消费者问题，深入地分析和透彻地理解这个例子，对于全面解决操作系统内的同步、互斥问题将有很大帮助。

（1）一个生产者，一个消费者，公用一个缓冲区。

定义两个同步信号量：

empty——表示缓冲区是否为空，初值为1。

full——表示缓冲区中是否为满，初值为0。

生产者进程

```

while(TRUE){

生产一个产品;

P(empty);

产品送往Buffer;

V(full);

}

```

消费者进程

```

while(True){

P(full);

从Buffer取出一个产品;

V(empty);

消费该产品;

}

```

（2）一个生产者，一个消费者，公用n个环形缓冲区。

定义两个同步信号量：

empty——表示缓冲区是否为空，初值为n。

full——表示缓冲区中是否为满，初值为0。

设缓冲区的编号为1～n-1，定义两个指针in和out，分别是生产者进程和消费者进程使用的指针，指向下一个可用的缓冲区。

生产者进程

```

while(TRUE){

生产一个产品;

P(empty);

产品送往buffer（in）；

in=(in+1)mod n；

V(full);

}

```

消费者进程

```

while(TRUE){

P(full);

从buffer（out）中取出产品；

out=(out+1)mod n；

V(empty);

消费该产品;

}

```

（3）一组生产者，一组消费者，公用n个环形缓冲区

在这个问题中，不仅生产者与消费者之间要同步，而且各个生产者之间、各个消费者之间还必须互斥地访问缓冲区。

定义四个信号量：

empty——表示缓冲区是否为空，初值为n。

full——表示缓冲区中是否为满，初值为0。

mutex1——生产者之间的互斥信号量，初值为1。

mutex2——消费者之间的互斥信号量，初值为1。

设缓冲区的编号为1～n-1，定义两个指针in和out，分别是生产者进程和消费者进程使用的指针，指向下一个可用的缓冲区。

生产者进程

```

while(TRUE){

生产一个产品;

P(empty);

P(mutex1)；

产品送往buffer（in）；

in=(in+1)mod n；

V(mutex1);

V(full);

}

```

消费者进程

```

while(TRUE){

P(full)

P(mutex2)；

从buffer（out）中取出产品；

out=(out+1)mod n；

V（mutex2）；

V(empty);

消费该产品;

}

```

需要注意的是无论在生产者进程中还是在消费者进程中，两个P操作的次序不能颠倒。应先执行同步信号量的P操作，然后再执行互斥信号量的P操作，否则可能造成进程死锁。

### 直线分割平面问题

一条线把平面分成两块，两条线把平面分成四块，如果任意两条线不平行，且没有3条线交在同一点，问100条线将平面分成多少块。

答案：5051

1条直线最多将平面分成2个部分；2条直线最多将平面分成4个部分；3条直线最多将平面分成7个部分；现在添上第4条直线．它与前面的3条直线最多有3个交点，这3个交点将第4条直线分成4段，其中每一段将原来所在平面部分一分为二，所以4条直线最多将平面分成7+4=11个部分．

完全类似地，5条直线最多将平面分成11+5=16个部分；6条直线最多将平面分成16+6=22个部分；7条直线最多将平面分成22+7=29个部分；8条直线最多将平面分成29+8=37个部分．

一般地，n条直线最多将平面分成2+2+3....+N=（N\*N+N+2）/2

### 称重问题

有N个球，其中只有一个是重量较轻的，用天平只称三次就能找到较轻的球，以下的N值哪个是可能的？

A 12

B 16

C 20

D 24

E 28

3 个一次可以测出来，3\*3 = 9 个以内 2 次，3\*3\*3 = 27 个以内，3次！

### 称重问题2

有一架天平，左右两侧均可以放砝码。现在希望能用它称量出从1克开始，尽可能多的连续的整数质量。如果现在允许你任意选择砝码组合，那么对于N个砝码来说最多可以称量出多少个从1克开始的连续质量？

(3N-1)/2，即1+3+32+…+3N-1。砝码的组合方式是使用1克，3克，9克，…，3N-1克的砝码各一个。例如N=2时，称量出1~4克的方法是：

1 = 1;

2 = 3-1;

3 = 3;

4 = 3+1;

而N=3时，依此不难得到5~13克的构造方法。形式化的证明可以使用数学归纳法。

最可能的得到上述答案的思路是从N较小的情况入手，使用递推方法，发现每增加一个砝码时应选择3N-1克的质量。随后使用数学归纳法证明。

### 称重问题3

现在共有13个球，这批球重有一个球的质量和其它球的质量不同（轻重未知）。给你一个天平，至多只有三次的称量机会，怎样将那个质量不一样的球找出来？

将13个球分为4球，4球，5球三组。

(1) 第一次称两个4球组，若不想等，则5球组全是标准球。然后就可以用12球类似的方法解决。

1.1 abcd轻。在efgh中取出fgh，替换掉abcd中的bcd。在ijkl中取出jkl，补充到原来fgh的位置。

如果afgh轻，说明答案为a或e。称量ab，如果相等，答案为e；如果不等，答案为a。

如果afgh重，说明答案在fgh中。称量fg，如果相等，答案为h；如果不等，重者为答案。如果一样重，答案在bcd中。称量bc，如果相等，答案为d；如果不等，轻者为答案。

1.2 abcd重。在efgh中取出fgh，替换掉abcd中的bcd。在ijkl中取出jkl，补充到原来fgh的位置。

如果afgh重，答案为a或e。称量ab，如果相等，答案为e；如果不等，答案为a。

如果afgh轻，答案在fgh中。称量fg，如果相等，答案为h；如果不等，轻者为所求。如果一样重，答案在bcd中。称量bc，如果相等，答案为d；如果不等，重者为答案。

(2) 若两个4球组相等，则异常球存在于5球组中。则从两个4球组中任取一个作为标准球。

### 统计海量数据中的前10个热门数据

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录（这些查询串的重复度比较高，`虽然总数是1千万，但如果除去重复后，不超过3百万个`。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

`外排序`（External sorting）是指能够处理极大量数据的排序算法。通常来说，`外排序处理的数据不能一次装入内存`，只能放在读写较慢的外存储器（通常是硬盘）上。外排序通常采用的是一种`“排序-归并”`的策略。在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，`将待排序数据组织为多个有序的临时文件`。尔后在归并段阶将这些临时文件组合为一个大的有序文件，也即排序结果。

`外归并排序`：外排序的一个例子是外归并排序（External merge sort），它读入一些能放在内存内的数据量，在内存中排序后输出为一个顺串（即是内部数据有序的临时文件），处理完所有的数据后再进行归并。比如，要对 900 MB 的数据进行排序，但机器上只有 100 MB 的可用内存时，外归并排序按如下方法操作：

1.读入 100 MB 的数据至内存中，用某种常规方式（如快速排序、堆排序、归并排序等方法）在内存中完成排序。

2.将排序完成的数据写入磁盘。

3.重复步骤 1 和 2 直到所有的数据都存入了不同的 100 MB 的块（临时文件）中。在这个例子中，有 900 MB 数据，单个临时文件大小为 100 MB，所以会产生 9 个临时文件。

4.读入每个临时文件（顺串）的前 10 MB （ = 100 MB / (9 块 + 1)）的数据放入内存中的输入缓冲区，最后的 10 MB 作为输出缓冲区。（实践中，将输入缓冲适当调小，而适当增大输出缓冲区能获得更好的效果。）

5.执行`九路归并算法`，将结果输出到输出缓冲区。一旦输出缓冲区满，将缓冲区中的数据写出至目标文件，清空缓冲区。直至所有数据归并完成。

### 统计热门查询

要统计最热门查询，首先就是要统计每个Query出现的次数，然后根据统计结果，找出Top 10。所以我们可以基于这个思路分两步来设计该算法。

即，此问题的解决分为以下俩个步骤：

\*\*第一步：Query统计\*\*

Query统计有以下俩个方法，可供选择：

1.直接排序法

首先我们最先想到的的算法就是排序了，首先对这个日志里面的所有Query都进行排序，然后再遍历排好序的Query，统计每个Query出现的次数了。但是题目中有明确要求，那就是内存不能超过1G，一千万条记录，每条记录是255Byte，很显然要占据2.375G内存，这个条件就不满足要求了。

让我们回忆一下数据结构课程上的内容，当数据量比较大而且内存无法装下的时候，我们可以采用外排序的方法来进行排序，这里我们可以采用归并排序，因为归并排序有一个比较好的时间复杂度O(NlgN)。排完序之后我们再对已经有序的Query文件进行遍历，统计每个Query出现的次数，再次写入文件中。综合分析一下，排序的时间复杂度是O(NlgN)，而遍历的时间复杂度是O(N)，因此该算法的总体时间复杂度就是O(N+NlgN)=O（NlgN）。

2.`Hash Table法`

在第1个方法中，我们采用了排序的办法来统计每个Query出现的次数，时间复杂度是NlgN，那么能不能有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，`虽然有一千万个Query，但是由于重复度比较高，因此事实上只有300万的Query`，每个Query255Byte，因此我们可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，Hash Table绝对是我们优先的选择，因为`Hash Table的查询速度非常的快，几乎是O(1)的时间复杂度`。

那么，我们的算法就有了：`维护一个Key为Query字串，Value为该Query出现次数的HashTable，每次读取一个Query，如果该字串不在Table中，那么加入该字串，并且将Value值设为1；如果该字串在Table中，那么将该字串的计数加一即可`。最终我们在O(N)的时间复杂度内完成了对该海量数据的处理。

本方法相比算法1：在时间复杂度上提高了一个数量级，为O（N），但不仅仅是时间复杂度上的优化，该方法只需要IO数据文件一次，而算法1的IO次数较多的，因此该算法2比算法1在工程上有更好的可操作性。

\*\*第二步：找出Top 10\*\*

算法一：普通排序

我想对于排序算法大家都已经不陌生了，这里不在赘述，我们要注意的是排序算法的时间复杂度是NlgN，在本题目中，三百万条记录，用1G内存是可以存下的。

算法二：部分排序

题目要求是求出Top 10，因此我们没有必要对所有的Query都进行排序，我们只需要维护一个10个大小的数组，初始化放入10个Query，按照每个Query的统计次数由大到小排序，然后遍历这300万条记录，每读一条记录就和数组最后一个Query对比，如果小于这个Query，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的Query。最后当所有的数据都遍历完毕之后，那么这个数组中的10个Query便是我们要找的Top10了。

不难分析出，这样，算法的最坏时间复杂度是N\*K， 其中K是指top多少。

算法三：堆

在算法二中，我们已经将时间复杂度由NlogN优化到NK，不得不说这是一个比较大的改进了，可是有没有更好的办法呢？

分析一下，在算法二中，每次比较完成之后，需要的操作复杂度都是K，因为要把元素插入到一个线性表之中，而且采用的是顺序比较。这里我们注意一下，该数组是有序的，一次我们每次查找的时候可以采用二分的方法查找，这样操作的复杂度就降到了logK，可是，随之而来的问题就是数据移动，因为移动数据次数增多了。不过，这个算法还是比算法二有了改进。

基于以上的分析，我们想想，有没有一种既能快速查找，又能快速移动元素的数据结构呢？回答是肯定的，那就是堆。

`借助堆结构，我们可以在log量级的时间内查找和调整/移动`。因此到这里，我们的算法可以改进为这样，维护一个K(该题目中是10)大小的小根堆，然后遍历300万的Query，分别和根元素进行对比。

思想与上述算法二一致，只是算法在算法三，我们采用了最小堆这种数据结构代替数组，把查找目标元素的时间复杂度有O（K）降到了O（logK）。

那么这样，采用堆数据结构，算法三，最终的时间复杂度就降到了N\*logK，和算法二相比，又有了比较大的改进。

\*\*总结：\*\*

至此，算法就完全结束了，经过上述第一步、先用Hash表统计每个Query出现的次数，O（N）；然后第二步、采用堆数据结构找出Top 10，N\*O（logK）。所以，我们最终的时间复杂度是：O（N） + N'\*O（logK）。（N为1000万，N’为300万）。

### 编写函数strcpy

```c++

//若参数没有const属性，则需要考虑重叠的情况

char \*strcpy(char \*strDest, const char \*strSrc) {

if ( strDest == NULL || strSrc == NULL)

return NULL ;

if ( strDest == strSrc)

return strDest ;

char \*tempptr = strDest ;

while( (\*strDest++ = \*strSrc++) != '/0')

;

return tempptr ;

}

```

### 试毒问题

有1000瓶水，其中有一瓶有毒，小白鼠只要尝一点带毒的水24小时后就会死亡至少要多少只小白鼠才能在24小时鉴别出哪瓶水有毒。

给1000个瓶分别标上如下标签（10位长度）：

0000000001 （第1瓶）

0000000010 （第2瓶）

0000000011 （第3瓶）

......

1111101000 （第1000瓶）

从编号最后1位是1的所有的瓶子里面取出1滴混在一起（比如从第一瓶，第三瓶，…里分别取出一滴混在一起）并标上记号为1。以此类推，从编号第一位是1的所有的瓶子里面取出1滴混在一起并标上记号为10。现在得到有10个编号的混合液，小白鼠排排站，分别标上10，9，…1号，并分别给它们灌上对应号码的混合液。24小时过去了，过来验尸：

从左到右，死了的小白鼠贴上标签1，没死的贴上0，最后得到一个序号，把这个序号换成10进制的数字，就是有毒的那瓶水的编号。

### 调整数组中数字的顺序，使得所有奇数位于数组的前半部分

输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分，要求时间复杂度为O(n)

维护两个指针，第一个指针指向数组的第一个数字，只向后移动，第二个指针指向最后一个数字，只向前移动。当第一个指针指向数字为偶数，且第二个指针指向数字为奇数时，交换数字，并移动两个指针。

### 输入一个单向链表，输出该链表中倒数第k个结点

输入一个单向链表，输出该链表中倒数第k个结点。链表的倒数第0个结点为链表的尾指针。

1.从头节点开始遍历链表，直到链表为节点，统计链表节点个数n。那么倒数第k个节点就是从头结点开始的第n-k-1个节点，则再从头结点开始遍历链表，直到第n-k-1个节点为止。这种思路需要遍历链表两次。

2.在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第k-1步之前，第二个指针保持不动；在第k-1步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第k个结点。这种思路只需要遍历链表一次。

### 输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变

句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。

例如输入"I am a student."，则输出"student. a am I"。

思路：

先颠倒句子中的所有字符，再颠倒每个单词内的字符。

如例句中，"I am a student."第一次整体翻转后得到".tneduts a ma I"；

第二次在每个单词中内部翻转得到"students. a am I"，即为题解。

### 颜色变换问题

有一个8\*8的矩形，分成64个1\*1的小方块。每个方块要么染成黑色或者白色。现在存在两种颜色互换操作：

第一种是将一个任意3\*3的矩形里面的所有小方块的颜色互换（即黑变白，白变黑）；

第二种是将一个任意4\*4的矩形里面的所有小方块的颜色互换；

那么对于任意一种染色方案，是否都可以通过这两种颜色互换操作（可以多次操作）将所有的64个小方块的颜色都变成白色？

思路：

显然，对于初始的矩形而言，一共有2^64种染色方案。第一种颜色互换操作一共有（8-3+1）\*(8-3+1)共36种小类型，第二种颜色互换操作一共有5\*5=25种类型，一共有61种小类型。由于在颜色互换的过程时，每种小类型最多出现一次（两次相同小类型的操作相当于没有操作，颜色没有变化），且最终结果与操作的顺序无关。所以颜色互换操作的状态最多只有2^61种，是小于2^64种染色方案的。因此，最终的答案是不可能的。

### 100w个数中找出最大的100个数

方案1：用一个含100个元素的最小堆完成。复杂度为O(100w\*lg100)。

方案2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，直到比轴大的一部分在比100多的时候，采用传统排序算法排序，取前100个。复杂度为O(100w\*100)。

方案3：采用局部淘汰法。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把x利用插入排序的思想，插入到序列L中。依次循环，知道扫描了所有的元素。复杂度为O(100w\*100)。

### 3\*4的格子有几个矩形

M\*N网格中有横竖各M+1、N+1条直线，其中，任意各取两条都可以组成一个长方形。

C(4,2)\*C(5,2)=6\*10=60;

A(N,N)=N!

A(N,M)=N\*(N-1)\*…\*(N-M+1)

C(N,M)=A(N,M)/A(M,M)

### 50个红球，50个篮球，2个袋子，一个袋子能装任意个球(0~100)

现由你将这100个球，以一定方法装入这两个袋子。另找一个不明真相的路人，闭眼，随机从两个袋子中任意摸一个球。

要使得他摸出红球的概率最高，你应该如何分配这100个球。

答案：一个袋子一个红球，另一个袋子49个红球+50个蓝球

首先可能会列方程解，说明思路比较清晰。但方程比较难解，如果可以解出来就加分。如果解不出来，建议他通过思考解答

能首先将问题分解为两个袋子红球概率和最大，加分

首先优化一个袋子红球的概率(50个红球全在袋子中)，其次不损失多余的红球，即可得出答案

还需要通过迭代法验证答案的正确性

### 54张扑克牌，一半红色一半黑色，随机取两张，一红一黑的概率

27/53