编程语言

算法

- 查找算法： 顺序、二分
- 算法时间复杂度计算方法
- 链表的php实现
- 排序算法：冒泡、选择、快速

开发工具

- 微信消息推送服务：Server酱

网络

- 浏览器从输入URL到页面展示经历的流程
- HTTP 提交和接收数据的常用方式
- HTTP和HTTPS详解
- TCP是如何保证数据可靠传输?
- TCP握手介绍
- 计算机网络模型 七层OSI & 四层TCP/IP
- http状态码
- 一台电脑配置无限好，可以同时打开多少个网页?
- session 与 cookie
- 禁用Cookie时，PHP共享Session文件解决方案

php

- php 遍历目录文件方法
- php 检测mysql表是否存在
- php curl 请求302跳转页面
- php pdo异常处理
- php curl post请求超过1024字节解决方法
- 一定范围内取几个不重复的随机数PHP方法
- php文件以绝对路径引入
- php array_map与array_walk使用对比
- php解析url方法

设计模式

查找算法： 顺序、二分

1、查找的方法：

```
1   顺序查找 、  二分法
```

2、顺序查找：

对某个数组，按照顺序，一个一个比较，找到你要的数据。

3、顺序查找实例：

```php
<?php
//顺序查找数组中某个数
//如从一个数组中找到一个数：34
//$arr = array(23,45,67,34,9,34,6)如果查到则输出下标，否则输出查无此数

$arr = array(23,45,67,34,9,34,6);
//设一个标志位
$flag = false;
foreach($arr as $x => $x_val)
{

    if ($x_val == 34)
    {
        echo 'arr['.$x.']=34'."<br>";
        $flag = true;
    }
}
if ($flag==false)
{
    echo "查无此数！";
}
```

```
?>
```

4、二分查找：

首先找到数组中间这个数，然后与要查找的数比较，如果要查找的数大于中间这个数，则说明应该向后找，否则向前找，如果想等，则说明找到。
前提：该数组必须是有序数列，如果该数组无序，必须先排序后查找

递归形式：

```php
<?php
//二分查找数组中某个数
//如从一个数组中找到一个数：134
//$arr = array(23,45,67,89,90,134,236)如果查到则输出下标，否则输出查无此数

function binarySearch(&$arr,$val,$leftindex,$rightindex)
{
    if($rightindex < $leftindex)
    {
        echo "查无此数！";
        return 0;
    }
    //四舍五入取整数值
    $middleindex = round(($leftindex + $rightindex)/2);
    if($val > $arr[$middleindex])
    {
        binarySearch($arr,$val,$middleindex + 1,$rightindex);
    }
    elseif($val < $arr[$middleindex])
    {
        binarySearch($arr,$val,$leftindex,$middleindex - 1);
    }
    else
    {
        echo 'arr['."$middleindex".']=134'."<br>";
    }
}
```

```
    $arr = array(23,45,67,89,90,134,236);
//  $leftindex = 0;左下标
//  $rightindex = count($arr)-1;右下标
//      $val = 134;要找的值
    binarySearch($arr,134,0,count($arr) - 1)
?>
```

非递归形式：

```
$nums = [-1, 0, 3, 5, 9, 12, 13];
$target = 9;
function search($nums, $target)
{
    if (empty($nums)) {
        return -1;
    }
    $l = 0;
    $r = count($nums) - 1;
    while ($l <= $r) {
        $mid = ceil(($l + $r) / 2);
        if ($nums[$mid] == $target) {
            return $mid;
        } elseif ($nums[$mid] > $target) {
            $r = $mid - 1;
        } else {
            $l = $mid + 1;
        }
    }
    return -1;
}

var_dump(search($nums, $target));
```

算法时间复杂度计算方法

概念:

时间复杂度是总运算次数表达式中受n的变化影响最大的那一项(不含系数)

比如：一般总运算次数表达式类似于这样：

`a*2^n+b*n^3+c*n^2+d*n*lg(n)+e*n+f`

a ！ =0时，时间复杂度就是O(2^n);

a=0,b<>0 =>O(n^3);

a,b=0,c<>0 =>O(n^2)依此类推

实例:

```
for(i=1;i<=n;i++) {//循环了n*n次，当然是O(n^2)
    for(j=1;j<=n;j++) {
        s++;
    }
}
for(i=1;i<=n;i++) {//循环了(n+n-1+n-2+...+1)≈(n^2)/2，因为时间复杂度是不考虑系数的，所以也是O(n^2)
    for(j=i;j<=n;j++) {
        s++;
    }
}
for(i=1;i<=n;i++) {//循环了(1+2+3+...+n)≈(n^2)/2,当然也是O(n^2)
    for(j=1;j<=i;j++) {
        s++;
    }
}
i=1;k=0;
while(i<=n-1){ //循环了n-1≈n次，所以是O(n)
    k+=10*i;
    i++;
}
for(i=1;i<=n;i++) { //循环了(1^2+2^2+3^2+...+n^2)=n(n+1)(2n+1)/6(这个公式要记住哦)≈(n^3)/3，不考虑系数，自然是O(n^3)
    for(j=1;j<=i;j++) {
        for(k=1;k<=j;k++) {
```

```
            x=x+1;
        }
    }
}

// 由于每次i乘以2之后，就距离n更近了一分。 也就是说，有多少个2相乘后大于n，则会退出循环。
// 由2^x=n 得到x=logn。 所以这个循环的时间复杂度为O(logn)。
i=1;
while (i<=n) {
    i=i*2;
}

/*解：语句1的频度是1,
    设语句2的频度是t,  则：nt<=n;  t<=log2n
    考虑最坏情况，取最大值t=log2n,
    T(n) = 1 + log2n
    f(n) = log2n
    lim(T(n)/f(n)) = 1/log2n + 1 = 1
    T(n) = O(log2n)*/
```

另外，在时间复杂度中，log(2,n)(以2为底)与lg(n)(以10为底)是等价的，因为对数换底公式：
log(a,b)=log(c,b)/log(c,a)
所以，log(2,n)=log(2,10)*lg(n),忽略掉系数，二者当然是等价的

计算方法:

求解算法的时间复杂度的具体步骤是：

⑴ 找出算法中的基本语句；

算法中执行次数最多的那条语句就是基本语句，通常是最内层循环的循环体。

⑵ 计算基本语句的执行次数的数量级；

只需计算基本语句执行次数的数量级，这就意味着只要保证基本语句执行次数的函数中的最高次幂正确即可，可以忽略所有低次幂和最高次幂的系数。这样能够简化算法分析，并且使注意力集中在最重要的一点上：增长率。

⑶ 用大O记号表示算法的时间性能。

将基本语句执行次数的数量级放入大O记号中。

如果算法中包含嵌套的循环，则基本语句通常是最内层的循环体，如果算法中包含并列的循环，则将并列循环的时间复杂度相加。例如：

for (i=1; i<=n; i++)

x++;

for (i=1; i<=n; i++)

for (j=1; j<=n; j++)

x++;

第一个for循环的时间复杂度为O(n)，第二个for循环的时间复杂度为O($n^{2)}$，则整个算法的时间复杂度为O($^{(n+n}$2)=O(n^2)。

常见的时间复杂度:

常见的算法时间复杂度由小到大依次为：

O(1)＜O(log2n)＜O(n)＜O(nlog2n)＜O($n^{2)<O(n}$3)＜...＜O(2^n)＜O(n!)

其中，

1.O(1)表示基本语句的执行次数是一个常数，一般来说，只要算法中不存在循环语句，其时间复杂度就是O(1)。

2.O(n)，O(n^2)， 立方阶O(n^3),...， k次方阶O(n^k) 为多项式阶时间复杂度，分别称为一阶时间复杂度，二阶时间复杂度。。。。

3.O(2^n)，指数阶时间复杂度，该种不实用

4.对数阶O(log2n)，线性对数阶O(nlog2n)，除了常数阶以外，该种效率最高

例：算法：

```
for (i=1;i<=n;++i)
  {
     for(j=1;j<=n;++j)
     {
          c[ i ][ j ]=0; //该步骤属于基本操作 执行次数：n^2
           for(k=1;k<=n;++k)
                c[ i ][ j ]+=a[ i ][ k ]*b[ k ][ j ]; //该步骤属于基本操作 执行次数：n^3
     }
  }
```

则有 T（n）= n^{2+n}3，根据上面括号里的同数量级，我们可以确定 n^3为T（n）的同数量级

则有f（n）= n^3，然后根据T（n）/f（n）求极限可得到常数c

则该算法的 时间复杂度：T（n）=O（n^3）

链表的php实现

什么是链表

链表(Linked List)是一种常见的线性结构。它不需要一块连续的内存空间，通过指针即可将一组零散的内存块串联起来。将那种碎片内存进行合理的利用，解决空间的问题。

我们把内存块存为链表的节点，为了将所有的节点串起来，每个链表的节点除了存储数据之外，还需要记录链表的下一个节点的地址，这个记录下个节点地址的指针我们叫做后驱指针。

搜索链表需要O(N)的时间复杂度，这点和数组类似，但是链表不能像数组一样，通过索引的方式以O(1)的时间读取第n个数。

链表的优势在于能够以较高的效率在任意位置插入或者删除一个节点。

链表是线性表的一种，所谓的线性表包含顺序线性表和链表，**顺序线性表是用数组实现的，在内存中有顺序排列**，通过改变数组大小实现。而链表不是用顺序实现的，用指针实现，在**内存中不连续**。链表有很多种不同的类型：单向链表、双向链表及循环链表。

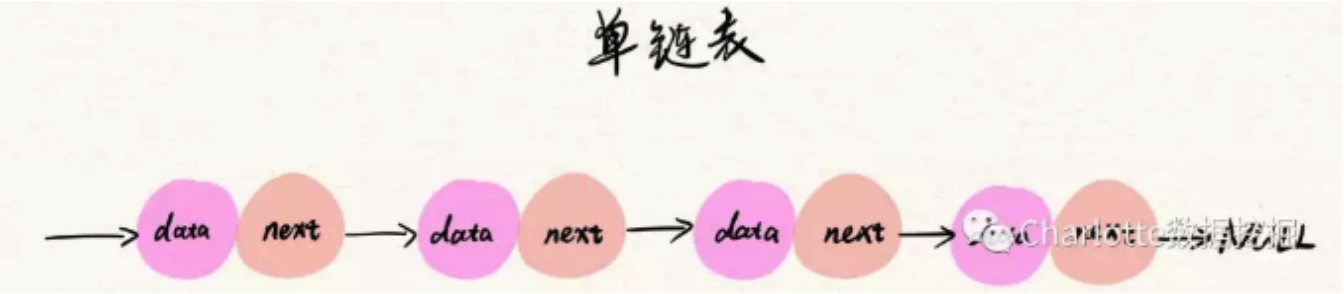
链表类别

单向链表

每个节点有一个next指针指向后一个节点，还有一个成员变量用于存储数值。

单向链表中有两个节点比较特殊，分别是头结点和尾节点。

头结点用来记录链表的基地址，知道头结点我们就可以遍历得到整条链表。尾结点的特殊在于指针指向的是一个空指针NULL。



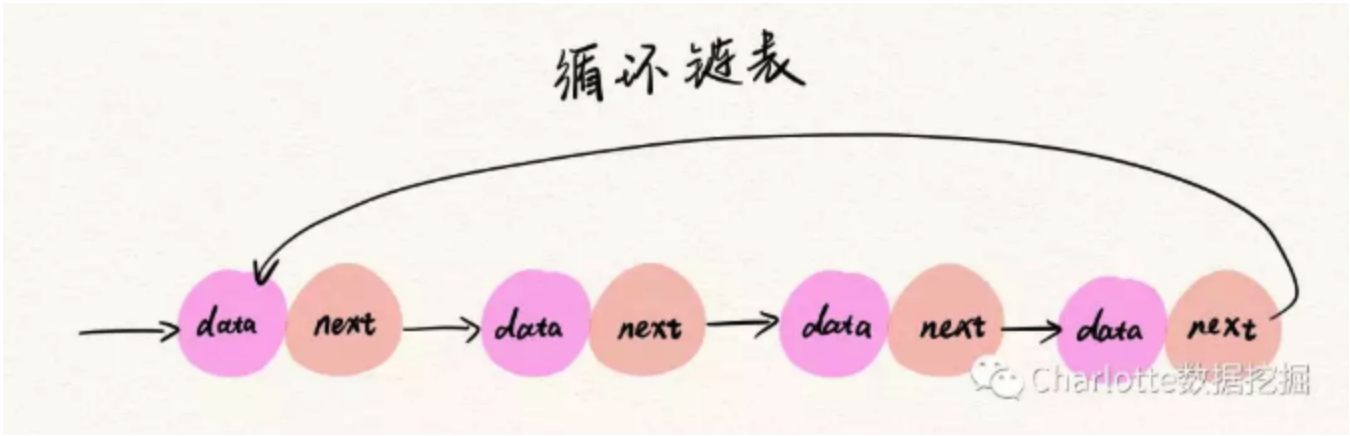
双向链表

双向链表支持两个方向，每个节点不只有一个后驱指针next指向后面的节点，还有一个前驱指针prev指向前面的节点。

循环链表

循环链表是一种特殊的单链表，与单链表不同的是尾节点不指向空地址，指向链表的头结点。

优点是从链尾到链头比较方便，当要处理的数据具有环形结构特点是，非常适合用循环链表来处理。



优缺点

优点：动态扩容，不需要占用过多的内存

缺点：不能随机访问，如果要访问一个元素的话，不能根据索引访问，只能从头开始遍历，找到对应的元素O(n)

代码实现

单向链表

```php
<?php
/**
 * 定义节点类，创建节点
 * Class Node
 */
class ListNode
{
    public $data;   // 节点数据
    public $next;   // 下一节点

    public function __construct($data)
    {
        $this->data = $data;
    }
}

/**
 * 单链表类
 * Class SingleLinkedList
 */
class SingleLinkedList
{
    private $head = null;
    private $length = 0;
    private $currentNode;
    private $currentPosition;

    /**
     * 插入一个节点
     * @param string|null $data
     * @return bool
     * complexity O(n)
     */
    public function insert(string $data = null)
    {
        $newNode = new ListNode($data);
```

```
37
38          if ($this->head === null) {
39              $this->head = &$newNode;
40          } else {
41              $currentNode = $this->head;
42              while ($currentNode->next !== null) {
43                  $currentNode = $currentNode->next;
44              }
45
46              $currentNode->next = $newNode;
47          }
48
49          $this->length++;
50          return true;
51      }
52
53      /**
54       * 在特定节点前插入
55       * @param string $data
56       * @param string $query
57       * complexity O(n)
58       */
59      public function insertBefore(string $data, string $query)
60      {
61          $newNode = new ListNode($data);
62
63          if ($this->head) {
64              $previous = null;
65              $currentNode = $this->head;
66              while ($currentNode !== null) {
67                  if ($currentNode->data === $query) {
68                      $newNode->next = $currentNode;
69                      if (empty($previous)) {
70                          $newNode->next = $currentNode;
71                          $this->head = $newNode;
72                      } else {
73                          $previous->next = $newNode;
74                      }
75                      $this->length++;
76                      break;
```

```
                }

                $previous = $currentNode;
                $currentNode = $currentNode->next;
            }
        }
    }

    /**
     * 在特定节点后插入
     * @param string|null $data
     * @param string|null $query
     * complexity O(n)
     */
    public function insertAfter(string $data = null, string $query = null)
    {
        $newNode = new ListNode($data);

        if ($this->head) {
            $nextNode = null;
            $currentNode = $this->head;

            while ($currentNode !== null) {
                if ($currentNode->data === $query) {

                    if ($nextNode !== null) {
                        $newNode->next = $nextNode;
                        $currentNode->next = $newNode;
                        $this->length++;
                        break;
                    } else {
                        $currentNode->next = $newNode;
                        $currentNode->next = $newNode;
                        $this->length++;
                        break;
                    }

                }

```

```
                $currentNode = $currentNode->next;
                $nextNode = $currentNode->next;
            }
        }
    }

    /**
     * 在最前方插入节点
     * @param string $data
     * complexity O(1)
     */
    public function insertAtFirst(string $data)
    {
        $newNode = new ListNode($data);

        if ($this->head === null) {
            $this->head = &$newNode;
        } else {
            $currentFirstNode = $this->head;
            $newNode->next = $currentFirstNode;
            $this->head = &$newNode;
        }

        $this->length++;
    }

    /**
     * 搜索一个节点
     * @param string $data
     * @return bool|ListNode
     * complexity O(n)
     */
    public function search(string $data)
    {
        if ($this->length > 0) {
            $currentNode = $this->head;
            while ($currentNode !== null) {
                if ($currentNode->data === $data) {
                    return $currentNode;
                }
```

```
156
157                 $currentNode = $currentNode->next;
158             }
159         }
160
161         return false;
162     }
163
164     /**
165      * 删除最前面的节点
166      * @return bool
167      * complexity O(1)
168      */
169     public function deleteFirst()
170     {
171         if ($this->head !== null) {
172             if ($this->head->next !== null) {
173                 $this->head = $this->head->next;
174             } else {
175                 $this->head = null;
176             }
177
178             $this->length--;
179
180             return true;
181         }
182
183         return false;
184     }
185
186     /**
187      * 删除最后面的节点
188      * @return bool
189      * complexity O(1)
190      */
191     public function deleteLast()
192     {
193         if ($this->head !== null) {
194             $currentNode = $this->head;
195
```

```
196             if ($currentNode->next !== null) {
197                 $previousNode = null;
198                 while ($currentNode->next !== null) {
199                     $previousNode = $currentNode;
200                     $currentNode = $currentNode->next;
201                 }
202 
203                 $previousNode->next = null;
204             } else {
205                 $this->head = null;
206             }
207 
208             $this->length--;
209             return true;
210         }
211 
212         return false;
213     }
214 
215     /**
216      * 删除特定节点
217      * @param string $query
218      * @return bool
219      * complexity O(n)
220      */
221     public function delete(string $query)
222     {
223         if ($this->head !== null) {
224             $currentNode = $this->head;
225             $previous = null;
226             while ($currentNode !== null) {
227 
228                 if ($currentNode->data === $query) {
229                     if ($currentNode->next === null) {
230                         $previous->next = null;
231                     } else {
232                         $previous->next = $currentNode->next;
233                     }
234 
235                     $this->length--;
```

```
236                     return true;
237                 }
238 
239                 $previous = $currentNode;
240                 $currentNode = $currentNode->next;
241             }
242 
243         }
244 
245         return false;
246     }
247 
248     /**
249      *反转链表
250      * complexity O(n)
251      */
252     public function reverse()
253     {
254         if ($this->head !== null) {
255             if ($this->head->next !== null) {
256                 $reveredList = null;
257                 $next = null;
258                 $currentNode = $this->head;
259 
260                 while ($currentNode !== null) {
261                     $next = $currentNode->next;
262                     $currentNode->next = $reveredList;
263                     $reveredList = $currentNode;
264                     $currentNode = $next;
265                 }
266 
267                 $this->head = $reveredList;
268             }
269         }
270 
271     }
272 
273     /**
274      * 返回特定位置的节点
275      * @param int $n
```

```
276      * @return null
277      * complexity O(n)
278      */
279     public function getNthNode(int $n = 0)
280     {
281         $count = 0;
282         if ($this->head !== null && $n <= $this->length) {
283             $currentNode = $this->head;
284 
285             while ($currentNode !== null) {
286                 if ($count === $n) {
287                     return $currentNode;
288                 }
289 
290                 $count++;
291                 $currentNode = $currentNode->next;
292             }
293 
294         }
295         return false;
296     }
297 
298     public function current()
299     {
300         return $this->currentNode->data;
301     }
302 
303     public function next()
304     {
305         $this->currentPosition++;
306         $this->currentNode = $this->currentNode->next;
307     }
308 
309     public function rewind()
310     {
311         $this->currentPosition = 0;
312         $this->currentNode = $this->head;
313     }
314 
315     public function key()
```

```
316     {
317         return $this->currentPosition;
318     }
319 
320     public function valid()
321     {
322         return $this->currentNode !== NULL;
323     }
324 
325     public function getSize()
326     {
327         return $this->length;
328     }
329 
330     public function display()
331     {
332         echo 'LinkList length: ' . $this->length . PHP_EOL;
333         $currentNode = $this->head;
334 
335         while ($currentNode !== null) {
336             echo $currentNode->data . PHP_EOL;
337             $currentNode = $currentNode->next;
338         }
339     }
340 
341 }
342 
343 $linkedList = new SingleLinkedList();
344 $linkedList->insert('China');
345 $linkedList->insert('USA');
346 $linkedList->insert('England');
347 $linkedList->insert('Australia');
348 echo '链表为：';
349 $linkedList->display();
350 $linkedList->reverse();
351 echo PHP_EOL;
352 echo '链表为：';
353 $linkedList->display();
```

排序算法：冒泡、选择、快速

1、数组的排序常用方法：

```
冒泡法、选择排序、插入排序、快速排序、
```

2、排序分类：

```
内部排序：指将需要处理的所有数据都加载到内部存储器中进行排序。

    内部排序又分为交换式（冒泡法、快速排序）、选择式、插入式排序法

    冒泡法、选择排序、插入排序、快速排序都属于内部排序


外部排序：数据量过大，无法全部加载到内存中，需要借助外部存储进行排序。
```

3、排序方法效率：

```
冒泡发<选择排序<插入排序

快速排序法速度很快，但效率并不高，占用空间太多
```

4、冒泡排序法实例：

冒泡排序的基本概念是：依次比较相邻的两个数，将小数放在前面，大数放在后面。即首先比较第1 个和第2个数，将小数放前，大数放后。然后比较第2个数和第3个数，将小数放前，大数放后，如此继续，直至比较最后两个数，将小数放前，大数放后。重复以上过程，仍从第一对数开始比较（因为可能由于第2个数和第3个数的交换，使得第1个数不再大于第2个数），将小数放前，大数放后，一直比较到最小数前的一对

相邻数，将小数放前，大数放后，第二趟结束，在倒数第二个数中得到一个新的最小数。如此下去，直至最终完成排序。由于在排序过程中总是小数往前放，大数往后放，相当于气泡往上升，所以称作冒泡排序。

```php
<?php
//echo "<br>";echo"<br>";echo "<br>";echo "<br>";

$arr =array(1,5,3,9,4,8,12,3,0,7,7.6,3.4,6);
//定义一个中间变量
$temp = 0;
//升序排序
for($i=0;$i<count($arr)-1;$i++)
{
        for($j=0;$j<count($arr)-1-$i;$j++)
        {       //说明前面的数比后面的大就要交换
                if($arr[$j]>$arr[$j+1])
                {
                        $temp= $arr[$j];
                        $arr[$j]= $arr[$j+1];
                        $arr[$j+1]= $temp;
                }
        }
}
//输出排序后数组
foreach($arr as $key=>$key_value)
{
        echo'$arr['.$key."]=".$key_value."<br>";
}
?>
```

冒泡排序时间复杂度为O(n^2);当数组顺序本来就是顺序时，此时是最好复杂度，可以认为是O(n);当数组顺序是逆序时，此时是最坏复杂度O(n^2);

5、选择排序法实例详解：

选择排序法的思路：就是再第一次循环中，假设第一个数是最小的；然后跟第二个数比较，一直比到最后，找出最小值，然后把最小值跟第一个数的位置互换；再进行下一次循环，找出最小值跟第二个位置的

数互换；一直循环数组的个数减去1次；数组就成了有序的了

```php
<?php
function selectSort(&$arr)
//注意此处要加地址传递符号；因为数组默认传递的是值，不是地址；若不地址传递，排序的是$arr而不是$asd。
{
    //$arr = array(1,5,3,9,4,8,12,3,0,7,7.6,3.4,6);
    //定义一个中间变量
    $temp = 0;
    //升序排序
    for($i=0;$i<count($arr)-1;$i++)
    {
        //假设$i就是最小的数
        $minVal=$arr[$i];
        //记录我认为最小数的下标
        $minIndex=$i;
        for($j=$i+1;$j<count($arr);$j++)
        {   //如果我认为的最小值不是最小
            if($minVal>$arr[$j])
            {
                $minVal = $arr[$j];
                $minIndex = $j;
            }
        }
        //最后交换
        $temp = $arr[$i];
        $arr[$i] = $arr[$minIndex];
        $arr[$minIndex] = $temp;
    }
}
$asd = array(1,0,2,9,3,8,4,7,5,6);
//调用选择排序法的函数
selectSort($asd);
//输出排序后数组  升序显示
foreach($asd as $key=>$key_value)
{
    echo '$asd['.$key."]=".$key_value."<br>";
}
```

```
?>
```

6、PHP快速排序法：

```
<?php
//快速排序法封装函数
function quick_Sort($array){
    //先判断是否需要继续进行，若所要排序数组只有一个元素或没有元素则不需要排序
    $len = count($array);
    if($len <= 1)
    {
        return $array;
    }
    //如果所给数组元素大于1个，需要排序
    //选择数组第一个元素作为标尺
    $key = $array[0];
    //初始化两个数组
    $left_array = array();//小于标尺的
    $right_array = array();//大于标尺的

    //遍历所给数组除了标尺外的所有元素，按照大小关系放入两个数组内
    for($i=1;$i<$len;$i++){
        if($array[$i]<$key){
    //如果数组元素小于标尺则将该元素放入左数组
            $left_array[] = $array[$i];
        }else{
            //如果数组元素大于标尺则将该元素放入右数组
            $right_array[] = $array[$i];
        }
    }
    //再分别对 左数组 和 右数组进行相同的排序处理方式
    //递归调用这个函数,并记录结果
    $left_array = quick_Sort($left_array);
    $right_array = quick_Sort($right_array);
    //合并左数组 标尺 右数组
    //array_merge() 函数把两个或多个数组合并为一个数组。
    //如果键名有重复，后面的键名的值覆盖前面的键名的值。如果数组是数字索引的，则键
```

```
    名会以连续方式重新索引。
34      //语法   array_merge(array1,array2,array3...)
35      return array_merge($left_array,array($key),$right_array);
36  }
37
38  $sortarray = array(13,89,23,9,19,88,56,78,34,69,10,14);
39  print_r(quick_Sort($sortarray));
40  ?>
```

微信消息推送服务：Server酱

Server酱

「Server酱」，英文名「ServerChan」，是一款「程序员」和「服务器」之间的通信软件。

说人话？就是从服务器推报警和日志到手机的工具。

程序员可以用它做程序监控报警服务。

官网地址： http://sc.ftqq.com

开通并使用上它，只需要一分钟：

```
1 登入：用GitHub账号登入网站，就能获得一个SCKEY（在「发送消息」页面）
2 绑定：点击「微信推送」，扫码关注同时即可完成绑定
3 发消息：往 http://sc.ftqq.com/SCKEY.send 发GET请求，就可以在微信里收到消息啦
```

类似的应用还有钉钉机器人 https://open-doc.dingtalk.com/docs/doc.htm?treeId=257&articleId=105733&docType=1， 也支持接口消息推送服务

浏览器从输入URL到页面展示经历的流程

1、输入地址

当我们开始在浏览器中输入网址的时候，浏览器其实就已经在智能的匹配可能得 url 了，他会从历史记录，书签等地方，找到已经输入的字符串可能对应的 url，然后给出智能提示，让你可以补全url地址。对于 google的chrome 的浏览器，他甚至会直接从缓存中把网页展示出来，就是说，你还没有按下 enter，页面就出来了。

2、浏览器查找域名的 IP 地址

2.1、请求一旦发起，浏览器首先要做的事情就是解析这个域名，一般来说，浏览器会首先查看本地硬盘的 hosts 文件，看看其中有没有和这个域名对应的规则，如果有的话就直接使用 hosts 文件里面的 ip 地址。

2.2、如果在本地的 hosts 文件没有能够找到对应的 ip 地址，浏览器会发出一个 DNS请求到本地DNS服务器 。本地DNS服务器一般都是你的网络接入服务器商提供，比如中国电信，中国移动。

2.3、查询你输入的网址的DNS请求到达本地DNS服务器之后，本地DNS服务器会首先查询它的缓存记录，如果缓存中有此条记录，就可以直接返回结果，此过程是递归的方式进行查询。如果没有，本地DNS服务器还要向DNS根服务器进行查询。

2.4、根DNS服务器没有记录具体的域名和IP地址的对应关系，而是告诉本地DNS服务器，你可以到域服务器上去继续查询，并给出域服务器的地址。这种过程是迭代的过程。

2.5、本地DNS服务器继续向域服务器发出请求，在这个例子中，请求的对象是.com域服务器。.com域服务器收到请求之后，也不会直接返回域名和IP地址的对应关系，而是告诉本地DNS服务器，你的域名的解析服务器的地址。

2.6、最后，本地DNS服务器向域名的解析服务器发出请求，这时就能收到一个域名和IP地址对应关系，本地DNS服务器不仅要把IP地址返回给用户电脑，还要把这个对应关系保存在缓存中，以备下次别的用户查询时，可以直接返回结果，加快网络访问。

3、浏览器向 web 服务器发送一个 HTTP 请求

拿到域名对应的IP地址之后，浏览器会以一个随机端口（1024<端口<65535）向服务器的WEB程序（常用的有httpd,nginx等）
80端口发起TCP的连接请求。这个连接请求到达服务器端后（这中间通过各种路由设备，局域网内除外），进入到网卡，
然后是进入到内核的TCP/IP协议栈（用于识别该连接请求，解封包，一层一层的剥开），还有可能要经过Netfilter防火墙
（属于内核的模块）的过滤，最终到达WEB程序，最终建立了TCP/IP的连接。

建立了TCP连接之后，发起一个http请求。一个典型的 http request header 一般需要包括请求的方法，例如 GET 或者 POST 等，不常用的还有 PUT 和 DELETE 、HEAD、OPTION以及 TRACE 方法，一般的浏览器只能发起 GET 或者 POST 请求。

客户端向服务器发起http请求的时候，会有一些请求信息，请求信息包含三个部分：

```
1 | 请求方法URI协议/版本
2
3 | 请求头(Request Header)
4
5 | 请求正文：
```

下面是一个完整的HTTP请求例子：

```
GET/sample.jsp
HTTP/1.1   Accept:image/gif.image/jpeg,*/*
Accept-Language:zh-cn
Connection:Keep-Alive
Host:localhost
User-Agent:Mozila/4.0(compatible;MSIE5.01;Window NT5.0)
Accept-Encoding:gzip,deflate

username=jinqiao&password=1234
```

注意：最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

（1）请求的第一行是“方法URL议/版本”：GET/sample.jsp HTTP/1.1
（2）请求头(Request Header)

请求头包含许多有关的客户端环境和请求正文的有用信息。例如，请求头可以声明浏览器所用的语言，请求正文的长度等。

```
Accept:image/gif.image/jpeg.*/*
Accept-Language:zh-cn
Connection:Keep-Alive
Host:localhost
User-Agent:Mozila/4.0(compatible:MSIE5.01:Windows NT5.0)
Accept-Encoding:gzip,deflate.
```

（3）请求正文

请求头和请求正文之间是一个空行，这个行非常重要，它表示请求头已经结束，接下来的是请求正文。请求正文中可以包含客户提交的查询字符串信息：

```
username=jinqiao&password=1234
```

4、服务器的永久重定向响应

服务器给浏览器响应一个301永久重定向响应，这样浏览器就会访问“http://www.google.com/”而非“http://google.com/”。

为什么服务器一定要重定向而不是直接发送用户想看的网页内容呢？其中一个原因跟搜索引擎排名有关。如果一个页面有两个地址，就像http://www.yy.com/和http://yy.com/，搜索引擎会认为它们是两个网站，结果造成每个搜索链接都减少从而降低排名。而搜索引擎知道301永久重定向是什么意思，这样就会把访问带www的和不带www的地址归到同一个网站排名下。还有就是用不同的地址会造成缓存友好性变差，当一个页面有好几个名字时，它可能会在缓存里出现好几次。

扩展知识

1）301和302的区别。

301和302状态码都表示重定向，就是说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的URL地址，这个地址可以从响应的Location首部中获取（用户看到的效果就是他输入的地址A瞬间变成了另一个地址B）——这是它们的共同点。

他们的不同在于。301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址；

302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，搜索引擎会抓取新的内容而保存旧的网址。 SEO 302好于301

2）重定向原因：

（1）网站调整（如改变网页目录结构）；

（2）网页被移到一个新地址；

（3）网页扩展名改变(如应用需要把.php改成.Html或.shtml)。

这种情况下，如果不做重定向，则用户收藏夹或搜索引擎数据库中旧地址只能让访问客户得到一个404页面错误信息，访问流量白白丧失；再者某些注册了多个域名的网站，也需要通过重定向让访问这些域名的用户自动跳转到主站点等。

3）什么时候进行301或者302跳转呢?

当一个网站或者网页24—48小时内临时移动到一个新的位置，这时候就要进行302跳转，而使用301跳转的场景就是之前的网站因为某种原因需要移除掉，然后要到新的地址访问，是永久性的。

清晰明确而言：使用301跳转的大概场景如下：

1、域名到期不想续费（或者发现了更适合网站的域名），想换个域名。

2、在搜索引擎的搜索结果中出现了不带www的域名，而带www的域名却没有收录，这个时候可以用301重定向来告诉搜索引擎我们目标的域名是哪一个。

3、空间服务器不稳定，换空间的时候。

5、浏览器跟踪重定向地址

现在浏览器知道了 "http://www.google.com/"才是要访问的正确地址，所以它会发送另一个http请求。这里没有啥好说的

6、服务器处理请求

经过前面的重重步骤，我们终于将我们的http请求发送到了服务器这里，其实前面的重定向已经是到达服务器了，那么，服务器是如何处理我们的请求的呢?

后端从在固定的端口接收到TCP报文开始，它会对TCP连接进行处理，对HTTP协议进行解析，并按照报文格式进一步封装成HTTP Request对象，供上层使用。

一些大一点的网站会将你的请求到反向代理服务器中，因为当网站访问量非常大，网站越来越慢，一台服务器已经不够用了。于是将同一个应用部署在多台服务器上，将大量用户的请求分配给多台机器处理。此时，客户端不是直接通过HTTP协议访问某网站应用服务器，而是先请求到Nginx，Nginx再请求应用服务器，然后将结果返回给客户端，这里Nginx的作用是反向代理服务器。同时也带来了一个好处，其中一台服务器万一挂了，只要还有其他服务器正常运行，就不会影响用户使用。

通过Nginx的反向代理，我们到达了web服务器，服务端脚本处理我们的请求，访问我们的数据库，获取需要获取的内容等等，当然，这个过程涉及很多后端脚本的复杂操作。

扩展阅读：
1）什么是反向代理?
客户端本来可以直接通过HTTP协议访问某网站应用服务器，网站管理员可以在中间加上一个Nginx，客户端请求Nginx，Nginx请求应用服务器，然后将结果返回给客户端，此时Nginx就是反向代理服务器。

7、服务器返回一个 HTTP 响应

经过前面的6个步骤，服务器收到了我们的请求，也处理我们的请求，到这一步，它会把它的处理结果返回，也就是返回一个HTPP响应。

HTTP响应与HTTP请求相似，HTTP响应也由3个部分构成，分别是：

```
l 状态行

l 响应头(Response Header)

l 响应正文
```

```
HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html;charset=ISO-8859-1   Content-Length: 122
<html>
<head>
<title>http</title>
</head>
<body>
<!-- body goes here -->
```

```
</body>
</html>
```

状态行：

状态行由协议版本、数字形式的状态代码、及相应的状态描述，各元素之间以空格分隔。

```
格式:    HTTP-Version Status-Code Reason-Phrase CRLF

例如:    HTTP/1.1 200 OK \r\n

-- 协议版本：是用http1.0还是其他版本

-- 状态描述：状态描述给出了关于状态代码的简短的文字描述。比如状态代码为200时的描述为ok

-- 状态代码：状态代码由三位数字组成，第一个数字定义了响应的类别，且有五种可能取值。
```

状态码表地址

响应头：

响应头部：由关键字/值对组成，每行一对，关键字和值用英文冒号":"分隔，典型的响应头有：

响应正文

包含着我们需要的一些具体信息，比如cookie，html,image，后端返回的请求数据等等。这里需要注意，响应正文和响应头之间有一行空格，表示响应头的信息到空格为止.

8、浏览器显示 HTML

在浏览器没有完整接受全部HTML文档时，它就已经开始显示这个页面了，浏览器是如何把页面呈现在屏幕上的呢？不同浏览器可能解析的过程不太一样，这里我们只介绍webkit的渲染过程，下图对应的就是WebKit渲染的过程，这个过程包括：

解析html以构建dom树 -> 构建render树 -> 布局render树 -> 绘制render树

浏览器在解析html文件时，会”自上而下“加载，并在加载过程中进行解析渲染。在解析过程中，如果遇到请求外部资源时，如图片、外链的CSS、iconfont等，请求过程是异步的，并不会影响html文档进行加载。

解析过程中，浏览器首先会解析HTML文件构建DOM树，然后解析CSS文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上。这个过程比较复杂，涉及到两个概念: reflow(回流)和repain(重绘)。

DOM节点中的各个元素都是以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为relow;当盒模型的位置,大小以及其他属性，如颜色,字体,等确定下来之后，浏览器便开始绘制内容，这个过程称为repain。

页面在首次加载时必然会经历reflow和repain。reflow和repain过程是非常消耗性能的，尤其是在移动设备上，它会破坏用户体验，有时会造成页面卡顿。所以我们应该尽可能少的减少reflow和repain。

当文档加载过程中遇到js文件，html文档会挂起渲染（加载解析渲染同步）的线程，不仅要等待文档中js文件加载完毕，还要等待解析执行完毕，才可以恢复html文档的渲染线程。因为JS有可能会修改DOM，最为经典的document.write，这意味着，在JS执行完成前，后续所有资源的下载可能是没有必要的，这是js阻塞后续资源下载的根本原因。所以我明平时的代码中，js是放在html文档末尾的。

JS的解析是由浏览器中的JS解析引擎完成的，比如谷歌的是V8。JS是单线程运行，也就是说，在同一个时间内只能做一件事，所有的任务都需要排队，前一个任务结束，后一个任务才能开始。但是又存在某些任务比较耗时，如IO读写等，所以需要一种机制可以先执行排在后面的任务，这就是：同步任务(synchronous)和异步任务(asynchronous)。

JS的执行机制就可以看做是一个主线程加上一个任务队列(task queue)。同步任务就是放在主线程上执行的任务，异步任务是放在任务队列中的任务。所有的同步任务在主线程上执行，形成一个执行栈;异步任务有了运行结果就会在任务队列中放置一个事件；脚本运行时先依次运行执行栈，然后会从任务队列里提取事件，运行任务队列中的任务，这个过程是不断重复的，所以又叫做事件循环(Event loop)。

###9、浏览器发送请求获取嵌入在 HTML 中的资源（如图片、音频、视频、CSS、JS等等）
其实这个步骤可以并列在步骤8中，在浏览器显示HTML时，它会注意到需要获取其他地址内容的标签。这时，浏览器会发送一个获取请求来重新获得这些文件。比如我要获取外图片，CSS，JS文件等，类似于下面的链接：

图片：http://static.ak.fbcdn.net/rsrc.php/z12E0/hash/8q2anwu7.gif

CSS式样表：http://static.ak.fbcdn.net/rsrc.php/z448Z/hash/2plh8s4n.css

JavaScript 文件：http://static.ak.fbcdn.net/rsrc.php/zEMOA/hash/c8yzb6ub.js

这些地址都要经历一个和HTML读取类似的过程。所以浏览器会在DNS中查找这些域名，发送请求，重定向等等...

不像动态页面，静态文件会允许浏览器对其进行缓存。有的文件可能会不需要与服务器通讯，而从缓存中直接读取，或者可以放到CDN中

文章转自： https://mp.weixin.qq.com/s/TN2LuOwN-XiRjThp0CzABw

HTTP 提交和接收数据的常用方式

我们知道，HTTP 协议是以 ASCII 码传输，建立在 TCP/IP 协议之上的应用层规范。HTTP 协议规定的 HTTP 请求方法有 OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE、CONNECT 这几种。其中 POST 一般用来向服务端提交数据，本文主要讨论 POST 提交数据的几种方式。

协议规定 POST 提交的数据必须放在消息主体（entity-body）中，但协议并没有规定数据必须使用什么编码方式。但是，数据发送出去，还要服务端解析成功才有意义。

服务端通常是根据请求头（headers）中的 Content-Type 字段来获知请求中的消息主体是用何种方式编码，再对主体进行解析。

POST 提交数据方案，包含了 Content-Type 和消息主体编码方式两部分。下面就正式开始介绍它们。

application/x-www-form-urlencoded （默认常用的）

这应该是最常见的 POST 提交数据的方式了。浏览器的原生 表单，如果不设置 enctype 属性，那么最终就会以 `application/x-www-form-urlencoded` 方式提交数据。

Content-Type 被指定为 `application/x-www-form-urlencoded`；其次，提交的数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL 转码。大部分服务端语言都对这种方式有很好的支持。例如 PHP 中，$*POST['title'] 可以获取到 title 的值，$*POST['sub'] 可以得到 sub 数组。

get方式的数据提交方式（编码方式）只有一种，就是`application/x-www-form-urlencoding`

multipart/form-data

这又是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，必须让 表单的 enctyped 等于 `multipart/form-data`。
它会将表单的数据处理为一条消息，以标签为单元，用分隔符分开。既可以上传键值对，也可以上传文件。当上传的字段是文件时，会有Content-Type来表名文件类型；content-disposition，用来说明字段的一些信息；
由于有boundary隔离，所以multipart/form-data既可以上传文件，也可以上传键值对，它采用了键值对的方式，所以可以上传多个文件。

application/json

application/json 这种方案，可以方便的提交复杂的结构化数据， 这个 Content-Type 作为响应头大家肯定不陌生。现在越来越多的人把它作为请求头，用来告诉服务端消息主体是序列化后的 JSON 字符串。

由于 JSON 规范的流行，除了低版本 IE 之外的各大浏览器都原生支持 JSON.stringify，服务端语言也都有处理 JSON 的函数，使用 JSON 不会遇上什么麻烦。

JSON 格式支持比键值对复杂得多的结构化数据，这一点也很有用 AngularJS 中的 Ajax 功能，默认就是提交 JSON 字符串。例如下面这段代码：

```
var data = {'title':'test', 'sub' : [1,2,3]};
$http.post(url, data).success(function(result) {
    ...
});
```

最终发送的请求是：

```
Content-Type: application/json;charset=utf-8
 {"title":"test","sub":[1,2,3]}
```

php需要通过`file_get_contents("php://input")`接受数据

text/xml

现在几乎不用

对于php服务端数据的接收：

1、如果是 `application/x-www-form-urlencoded` 和 `multipart/form-data` 格式 用 `$_POST`；
2、如果不能获取的时候比如 `text/xml`、`application/json`、`soap`，使用 `file_get_contents('php://input')`；

HTTP和HTTPS详解

加密类型

先科普一下，加密算法的类型基本上分为了两种：

> 对称加密，比较有代表性的就是 AES 加密算法；

> 非对称加密，经常使用到的 RSA 加密算法就是非对称加密的；

对称加密的意思就是说双方都有一个共同的密钥，然后通过这个密钥完成加密和解密，这种加密方式速度快，但是安全性不如非对称加密好。

非对称加密就是有两把密钥，公钥和私钥。私钥自己藏着，不告诉任何人；而公钥可以公开给别人。相比较对称加密而言，非对称加密安全性更高，但是加解密耗费的时间更长，速度慢。

HTTPS加密原理

HTTPS = HTTP + SSL

从这个公式中可以看出，HTTPS 和 HTTP 就差在了 SSL 上。所以我们可以猜到，HTTPS 的加密就是在 SSL 中完成的。

所以我们的目的就是要搞懂在 SSL 中究竟干了什么？

这就要从 CA 证书讲起了。CA 证书其实就是数字证书，是由 CA 机构颁发的。至于 CA 机构的权威性，那么是毋庸置疑的，所有人都是信任它的。CA 证书内一般会包含以下内容：

```
证书的颁发机构、版本

证书的使用者

证书的公钥

```

```
证书的有效时间

证书的数字签名 Hash 值和签名 Hash 算法

…
```

客户端如何校验 CA 证书:

CA 证书中的 Hash 值，其实是用证书的私钥进行加密后的值（证书的私钥不在 CA 证书中）。然后客户端得到证书后，利用证书中的公钥去解密该 Hash 值，得到 Hash-a ；然后再利用证书内的签名 Hash 算法去生成一个 Hash-b 。最后比较 Hash-a 和 Hash-b 这两个的值。如果相等，那么证明了该证书是对的，服务端是可以被信任的；如果不相等，那么就说明该证书是错误的，可能被篡改了，浏览器会给出相关提示，无法建立起 HTTPS 连接。除此之外，还会校验 CA 证书的有效时间和域名匹配等。

HTTPS 中的 SSL 握手建立过程

假设现在有客户端 A 和服务器 B ：

> 1.首先，客户端 A 访问服务器 B ，比如我们用浏览器打开一个网页 https://www.baidu.com ，这时，浏览器就是客户端 A ，百度的服务器就是服务器 B 了。这时候客户端 A 会生成一个随机数1，把随机数1 、自己支持的 SSL 版本号以及加密算法等这些信息告诉服务器 B 。

> 2.服务器 B 知道这些信息后，然后确认一下双方的加密算法，然后服务端也生成一个随机数 B ，并将随机数 B 和 CA 颁发给自己的证书一同返回给客户端 A 。

> 3.客户端 A 得到 CA 证书后，会去校验该 CA 证书的有效性，校验方法在上面已经说过了。校验通过后，客户端生成一个随机数3 ，然后用证书中的公钥加密随机数3 并传输给服务端 B 。
> 4.服务端 B 得到加密后的随机数3，然后利用私钥进行解密，得到真正的随机数3。

> 5.最后，客户端 A 和服务端 B 都有随机数1、随机数2、随机数3，然后双方利用这三个随机数生成一个对话密钥。之后传输内容就是利用对话密钥来进行加解密了。这时就是利用了对称加密，一般用的都是 AES 算法。

> 6.客户端 A 通知服务端 B ，指明后面的通讯用对话密钥来完成，同时通知服务器 B 客户端 A 的握手过程结束。

> 7.服务端 B 通知客户端 A，指明后面的通讯用对话密钥来完成，同时通知客户端 A 服务器 B 的握手过程结束。

> 8.SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户端 A 和服务器 B 开始使用相同的对话密钥进行数据通讯。

到此，SSL 握手过程就讲完了。可能上面的流程太过于复杂，我们简单地来讲：

```
1.客户端和服务端建立 SSL 握手，客户端通过 CA 证书来确认服务端的身份；
2.互相传递三个随机数，之后通过这随机数来生成一个密钥；
3.互相确认密钥，然后握手结束；
4.数据通讯开始，都使用同一个对话密钥来加解密；
```

我们可以发现，在 HTTPS 加密原理的过程中把对称加密和非对称加密都利用了起来。即利用了非对称加密安全性高的特点，又利用了对称加密速度快，效率高的好处。

http 和 https 的区别总结：

http: 超文本传输协议，是互联网上应用最为广泛的一种网络协议。

https: 是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。

Https 协议需要 CA 证书，费用较高。

http 信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

端口不同，一般而言，http 协议的端口为 80，https 的端口为 443。

http 的连接很简单，是无状态的。

https 协议是由 ssl+http 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

TCP是如何保证数据可靠传输?

运输层分为两个协议UDP和TCP，UDP是一个不可靠的协议，也就是说他仅仅提供复用和分用的功能但是对于比特差错或者丢弃不做任何处理。

TCP提供一种面向连接的、可靠的字节流服务，发送方发送的数据到达接收方的时候不会发生错误，不会丢失，不会乱序。

面向连接：意味着两个使用TCP的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个TCP连接。在一个TCP连接中，仅有两方进行彼此通信。广播和多播不能用于TCP。

TCP通过下列方式来提供可靠性：

三次握手：

通过三次握手建立可靠地通信连接

合理截断数据包：

应用数据被分割成TCP认为最适合发送的数据块,即将数据截断为合理的长度。这和UDP完全不同，应用程序产生的数据报长度将保持不变。

超时重发：

当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

包校验确认：

当TCP收到发自TCP连接另一端的数据，它将发送一个确认响应。这个确认不是立即发送，通常将推迟几分之一秒，对包做完整校验。

差错检测：

也就是引入校验和。在TCP的首部中有一个占据16为的空间用来放置校验和的结果。在源主机的运输层开始接受到一个从应用进程传下来的数据的时候，会将他封装成一个报文段，加上至少20字节的首部。同时会将这个报文段首部和数据还有伪首部部分一起根据取反码和的形式计算出校验和添加到首部中。传输到目的主机的运输层之后，会计算这个通过这个校验和检查是否存在比特差错。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。校验出包有错，丢弃报文段，不给出响应，TCP发送数据端，超时时会重发数据。

失序数据重新排序：

既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文段的到达也可能会失序。如果必要，TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。

能够丢弃重复数据：

既然IP数据报会发生重复，TCP的接收端必须丢弃重复的数据。

####可以进行流量控制：
TCP还能提供流量控制。TCP连接的每一方都有固定大小的缓冲空间。TCP的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。TCP使用的流量控制协议是可变大小的滑动窗口协议。

字节流服务：两个应用程序通过TCP连接交换8bit字节构成的字节流。TCP不在字节流中插入记录标识符。我们将这称为字节流服务（bytestreamservice）。

TCP对字节流的内容不作任何解释:TCP对字节流的内容不作任何解释。TCP不知道传输的数据字节流是二进制数据，还是ASCII字符、EBCDIC字符或者其他类型数据。对字节流的解释由TCP连接双方的应用层解释。

TCP握手介绍

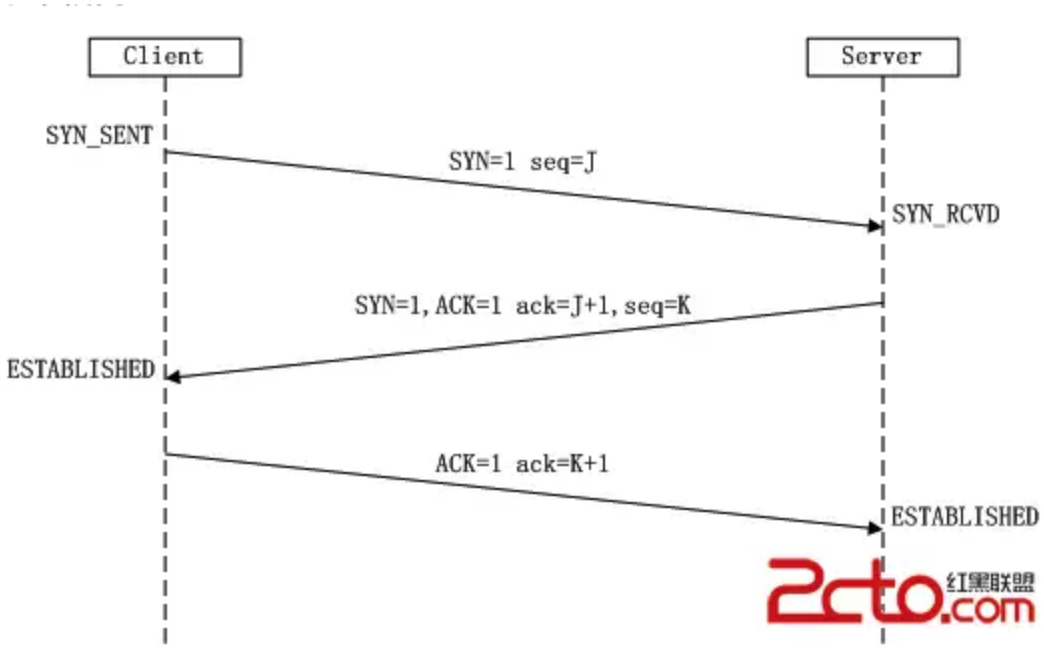
TCP三次握手

第一次握手：客户端A将标志位SYN置为1,随机产生一个值为seq=J（J的取值范围为=1234567）的数据包到服务器，客户端A进入SYN_SENT状态，等待服务端B确认；（a等待b确认）

第二次握手：服务端B收到数据包后由标志位SYN=1知道客户端A请求建立连接，服务端B将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给客户端A以确认连接请求，服务端B进入SYN_RCVD状态。（b告诉a已确认）

第三次握手：客户端A收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给服务端B，服务端B检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，客户端A和服务端B进入ESTABLISHED状态，完成三次握手，随后客户端A与服务端B之间可以开始传输数据了。（建立连接）

如图所示：



为什需要三次握手?

《计算机网络》第四版中讲“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”
书中的例子是这样的，“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接

假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”。主要目的防止server端一直等待，浪费资源。

TCP四次挥手

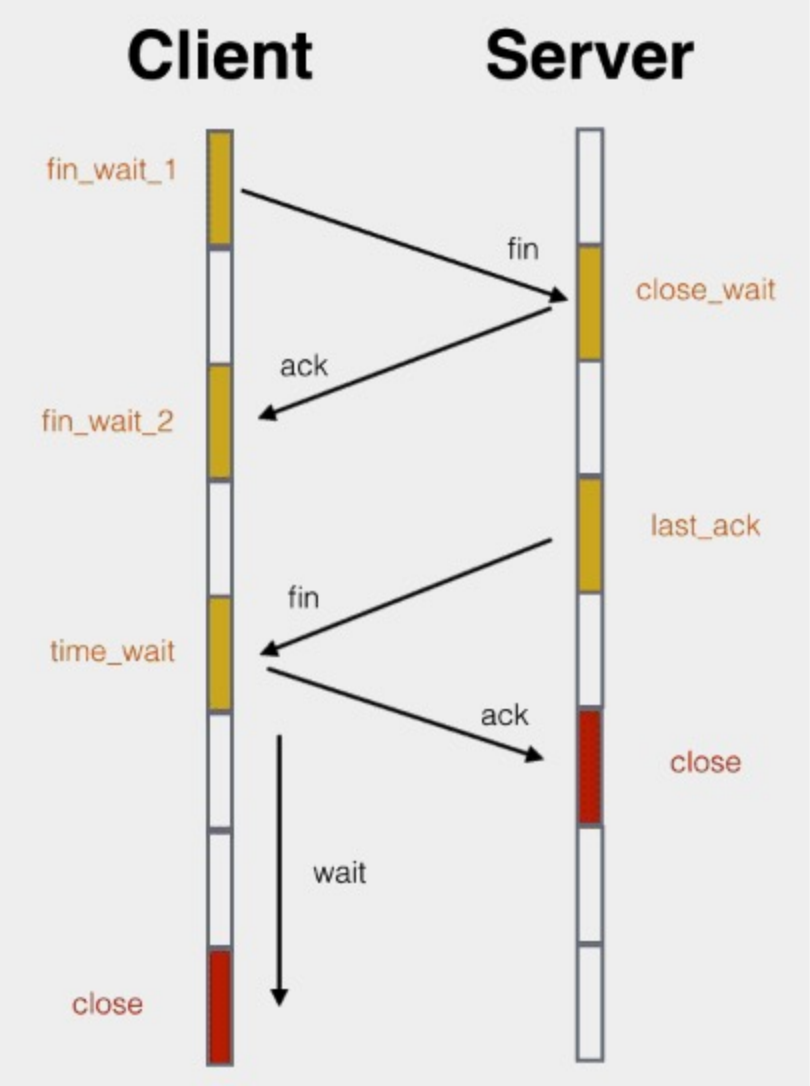
第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN_WAIT_1状态。

第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE_WAIT状态。

第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST_ACK状态。

第四次挥手：Client收到FIN后，Client进入TIME_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

如图所示：



为什么建立连接是三次握手，而关闭连接却是四次挥手呢?

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

计算机网络模型 七层OSI & 四层TCP/IP

OSI七层和TCP/IP四层的关系

1、 OSI引入了服务、接口、协议、分层的概念，TCP/IP借鉴了OSI的这些概念建立TCP/IP模型。

2、 OSI先有模型，后有协议，先有标准，后进行实践；而TCP/IP则相反，先有协议和应用再提出了模型，且是参照的OSI模型。

3、 OSI是一种理论下的模型，而TCP/IP已被广泛使用，成为网络互联事实上的标准。

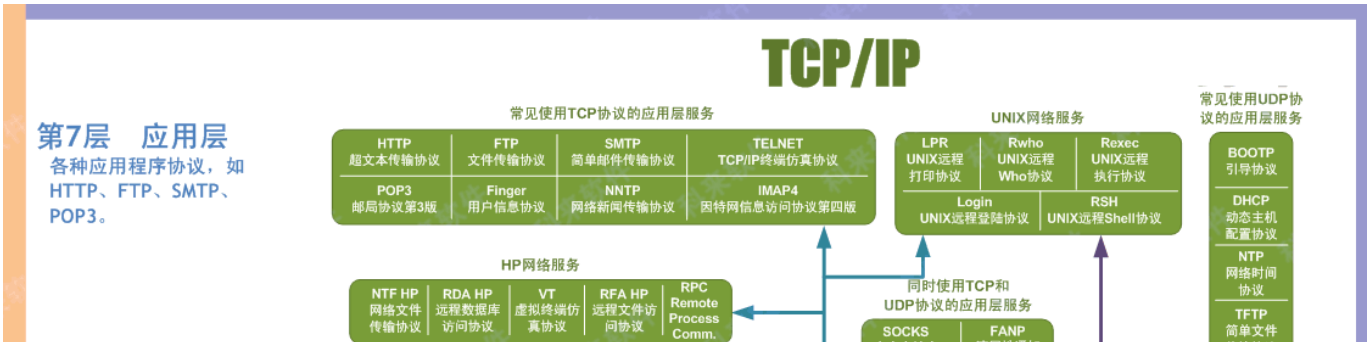
TCP：transmission control protocol 传输控制协议

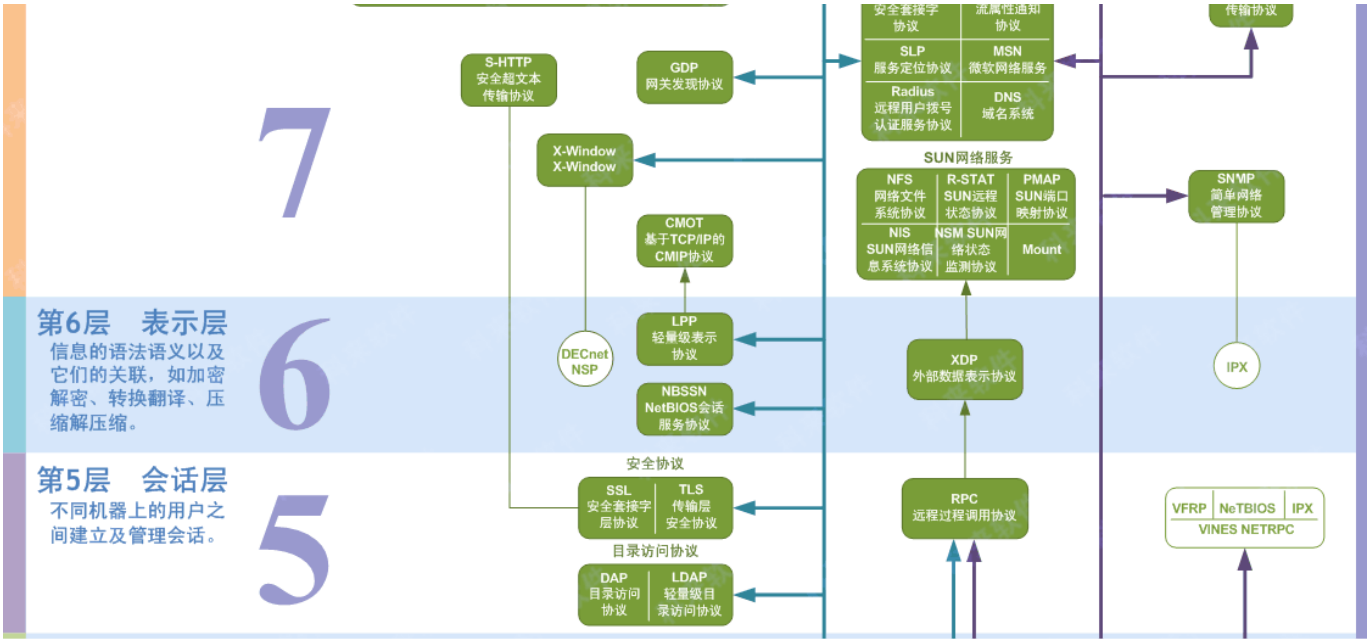
UDP：user data protocol 用户数据报协议

OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	HTTP、TFTP, FTP, NFS, WAIS、SMTP
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

OSI七层协议模型

osi: 开放式系统互联




7
S-HTTP
安全超文本传输协议
GDP
网关发现协议
安全套接字协议
流属性通知协议
SLP
服务定位协议
MSN
微软网络服务
Radius
远程用户拨号认证服务协议
DNS
域名系统
传输协议
X-Window
X-Window
SUN网络服务
NFS
网络文件系统协议
R-STAT
SUN远程状态协议
PMAP
SUN端口映射协议
NIS
SUN网络信息系统协议
NSM SUN网络状态监测协议
Mount
SNMP
简单网络管理协议
CMOT
基于TCP/IP的CMIP协议
第6层 表示层
信息的语法语义以及它们的关联，如加密解密、转换翻译、压缩解压缩。
6
DECnet NSP
LPP
轻量级表示协议
NBSSN
NetBIOS会话服务协议
XDP
外部数据表示协议
IPX
安全协议
第5层 会话层
不同机器上的用户之间建立及管理会话。
5
SSL
安全套接字层协议
TLS
传输层安全协议
RPC
远程过程调用协议
VFRP
NeTBIOS
IPX
VINES NETRPC
目录访问协议
DAP
目录访问协议
LDAP
轻量级目录访问协议

1、应用层（Application Layer）

术语“应用层”并不是指运行在网络上的某个特别应用程序 ，应用层提供的服务包括文件传输、文件管理以及电子邮件的信息处理。

2、表示层（Presentation Layer）

数据的表示、安全、压缩、编码、加解密。可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。

格式有：JPEG、ASCII、DECOIC、加密格式等。

应用程序和网络之间的翻译官，在表示层，数据将按照网络能理解的方案进行格式化；这种格式化也因所使用网络的类型不同而不同。

表示层管理数据的解密与加密，如系统口令的处理。例如：在 Internet上查询你银行账户，使用的即是一种安全连接。你的账户数据在发送前被加密，在网络的另一端，表示层将对接收到的数据解密。除此之外，表示层协议还对图片和文件格式信息进行解码和编码。

3、会话层（Session Layer）

建立、管理、终止会话，对应主机进程，指本地主机与远程主机正在进行的会话。

通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。

负责在网络中的两节点之间建立、维持和终止通信。 会话层的功能包括：建立通信链接，保持会话过程通信链接的畅通，同步两个节点之间的对话，决定通信是否被中断以及通信中断时决定从何处重新发送。

你可能常常听到有人把会话层称作网络通信的“交通警察”。当通过拨号向你的 ISP （因特网服务提供商）请求连接到因特网时，ISP 服务器上的会话层向你与你的 PC 客户机上的会话层进行协商连接。若你的电话线偶然从墙上插孔脱落时，你终端机上的会话层将检测到连接中断并重新发起连接。会话层通过决定节点通信的优先级和通信时间的长短来设置通信期限。

4、传输层（Transport Layer）
定义传输数据的协议端口号，以及流控和差错校验。
协议有：TCP UDP等，数据包一旦离开网卡即进入网络传输层。

定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做`段`。

ＯＳＩ模型中最重要的一层。传输协议同时进行流量控制或是基于接收方可接收数据的快慢程度规定适当的发送速率。除此之外，传输层按照网络能处理的最大尺寸将较长的数据包进行强制分割。例如，以太网无法接收大于１５００字节的数据包。发送方节点的传输层将数据分割成较小的数据片，同时对每一数据片安排一序列号，以便数据到达接收方节点的传输层时，能以正确的顺序重组。该过程即被称为排序。工作在传输层的一种服务是ＴＣＰ／ＩＰ协议套中的ＴＣＰ（传输控制协议），另一项传输层服务是ＩＰＸ／ＳＰＸ协议集的ＳＰＸ（序列包交换）。

5、网络层（Network Layer）
进行逻辑地址寻址，实现不同网络之间的路径选择。
协议有：ICMP IGMP IP（IPV4 IPV6） ARP RARP等。

在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。

ＯＳＩ模型的第三层，其主要功能是将网络地址翻译成对应的物理地址，并决定如何将数据从发送方路由到接收方。

网络层通过综合考虑发送优先权、网络拥塞程度、服务质量以及可选路由的花费来决定从一个网络中节点A到另一个网络中节点B 的最佳路径。由于网络层处理，并智能指导数据传送，路由器连接网络各段，所以`路由器属于网络层`。在网络中，“路由”是基于编址方案、使用模式以及可达性来指引数据的发送。

网络层负责在源机器和目标机器之间建立它们所使用的路由。这一层本身没有任何错误检测和修正机制，因此，网络层必须依赖于端端之间的由D L L提供的可靠传输服务。

网络层用于本地L A N网段之上的计算机系统建立通信，它之所以可以这样做，是因为它有自己的路由地址结构，这种结构与第二层机器地址是分开的、独立的。这种协议称为路由或可路由协议。路由协议包括I P、N o v e l l公司的I P X以及A p p l e Ta l k协议。

网络层是可选的，它只用于当两个计算机系统处于不同的由路由器分割开的网段这种情况，或者当通信应用要求某种网络层或传输层提供的服务、特性或者能力时。例如，当两台主机处于同一个L A N网段的直接相连这种情况，它们之间的通信只使用L A N的通信机制就可以了(即OSI 参考模型的一二层)。

6、数据链路层（Datalink Layer）
建立逻辑连接、进行硬件地址寻址、差错校验等功能。（由底层网络定义协议）
将比特组合成字节进而组合成帧，用MAC地址访问介质，错误发现但不能纠正。

数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。

定义了如何让格式化数据以进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。

OSI模型的第二层，它控制网络层与物理层之间的通信。它的主要功能是如何在不可靠的物理线路上进行数据的可靠传递。为了保证传输，从网络层接收到的数据被分割成特定的可被物理层传输的帧。帧是用来移动数据的结构包，它不仅包括原始数据，还包括发送方和接收方的物理地址以及检错和控制信息。其中的地址确定了帧将发送到何处，而纠错和控制信息则确保帧无差错到达。 如果在传送数据时，接收点检测到所传数据中有差错，就要通知发送方重发这一帧。

数据链路层的功能独立于网络和它的节点和所采用的物理层类型，它也不关心是否正在运行 W o r d 、E x c e l 或使用I n t e r n e t 。有一些连接设备，如交换机，由于它们要对帧解码并使用帧信息将数据发送到正确的接收方，所以它们是工作在数据链路层的。

数据链路层（DataLinkLayer):在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧（Frame）在信道上无差错的传输，并进行各电路上的动作系列。

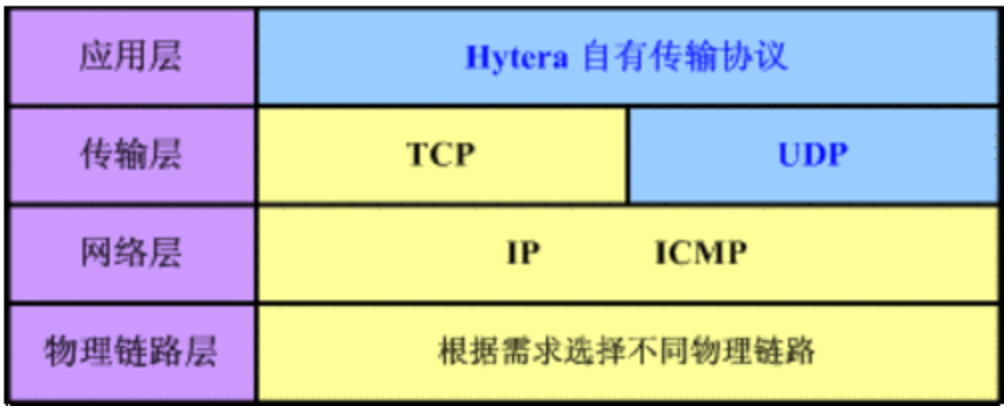
数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。

7、物理层（Physical Layer）
建立、维护、断开物理连接。（由底层网络定义协议）

主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输,到达目的地后在转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

ＯＳＩ模型的最低层或第一层，该层包括物理连网媒介，如电缆连线连接器。物理层的协议产生并检测电压以便发送和接收携带数据的信号。在你的桌面Ｐ Ｃ 上插入网络接口卡，你就建立了计算机连网的基础。换言之，你提供了一个物理层。尽管物理层不提供纠错服务，但它能够设定数据传输速率并监测数据出错率。网络物理问题，如电线断开，将影响物理层。 用户要传递信息就要利用一些物理媒体，如双绞线、同轴电缆等，但具体的物理媒体并不在OSI的7层之内，有人把物理媒体当做第0层，物理层的任务就是为它的上一层提供一个物理连接，以及它们的机械、电气、功能和过程特性。如规定使用电缆和接头的类型、传送信号的电压等。在这一层，数据还没有被组织，仅作为原始的位流或电气电压处理，单位是bit比特。

TCP/IP四层模型



1、 链路层（数据链路层/网络接口层）：包括操作系统中的设备驱动程序、计算机中对应的网络接口卡

2、 网络层（互联网层）：处理分组在网络中的活动，比如分组的选路。

3、 传输层：主要为两台主机上的应用提供端到端的通信。网络层IP提供的是一种不可靠的服务。它只是尽可能快地把分组从源节点送到目的节点，但不提供任何可靠性的保证。Tcp在不可靠的ip层上，提供了一个可靠的运输层。

4、 应用层：负责处理特定的应用程序细节。

几个协议

IP协议（Internet protocol）

这里的IP不是指的我们通常所说的192.168.1.1.这个IP指的是一种协议。192.168.1.1.指的是IP地址。IP协议的作用在于把各种数据包准确无误的传递给对方，其中两个重要的条件是IP地址和MAC地址（Media Access Control Address）。由于IP地址是稀有资源，不可能每个人都拥有一个IP地址，所以我们通常的IP地址是路由器给我们生成的IP地址，路由器里面会记录我们的MAC地址。而MAC地址是全球唯一的，除去人为因素外不可能重复。举一个现实生活中的例子，IP地址就如同是我们居住小区的地址，而MAC地址就是我们住的那栋楼那个房间那个人。

TCP协议

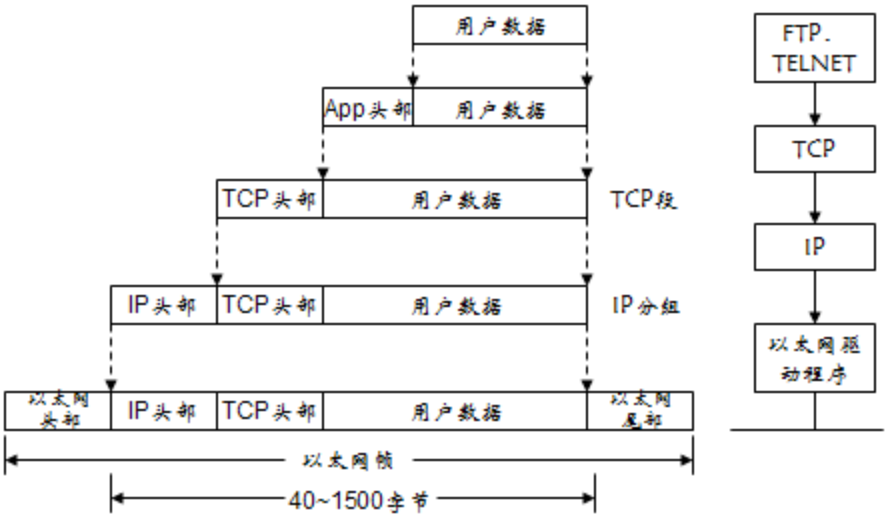
按层次分，TCP属于传输层，提供可靠的字节流服务。所谓的字节流，其实就类似于信息切割。比如你是一个卖自行车的，你要去送货。安装好的自行车，太过庞大，又不稳定，容易损伤。不如直接把自行车拆开来，每个零件上都贴上收货人的姓名。最后送到后按照把属于同一个人的自行车再组装起来，这个拆解、运输、拼装的过程其实就是TCP字节流的过程。

数据包说明：

IP层传输单位是IP分组，属于点到点的传输；TCP层传输单位是TCP段，属于端到端的传输

数据封装

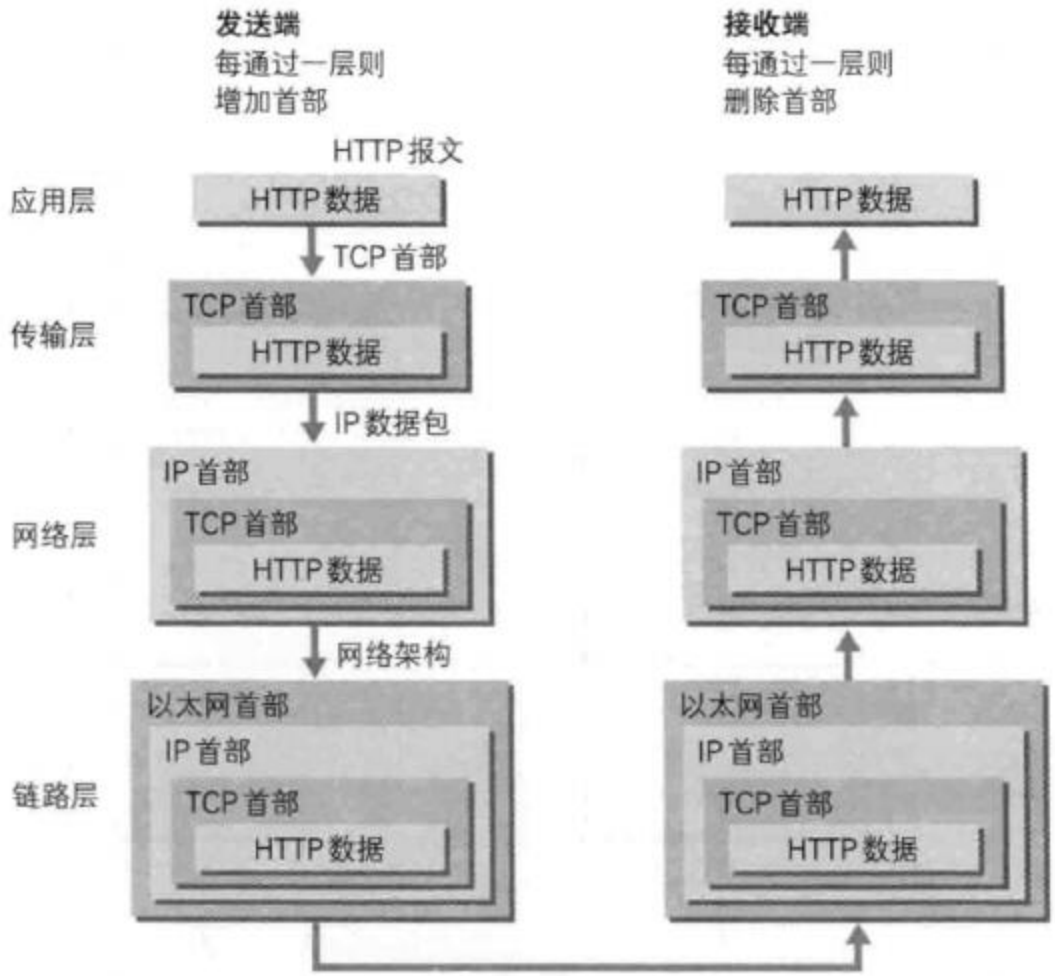
就是从应用程序给出用户的数据,然后一层一层加上对应的头部,以便于对方的每一层进行识别转交.



对等通信

对于每一个层来说,都仿佛好像和对面的这个层进行通信,实际上的信息传输是自上而下（逐层数据封装）,然后通过以太网或者IEEE进行BIT流传输到对方的链路层,然后自下而上到达对应的层逐层数据解析）.

在以太网传输过来数据的时候,每层进行解析,看是什么协议交由什么层,做出相应的应答,最后如果是TCP或者UDP,则根据端口号提交给相应的应用程序,传输数据。



端口

1、众所周知的端口

0到1023端口,由IANA分配,紧密绑定服务.通常这些端口明确表明了某种服务协议.例如21端口为FTP服务端口

2、注册端口

1024到49151.不受IANA控制,但是已经被IANA登记了,松散绑定一些服务,可以用于其他目的

3、动态或者私有端口

49152到65535,IANA不管,所以一般机器从1024开始分配动态端口

http状态码

状态代码由三位数字组成，第一个数字定义了响应的类别，且有五种可能取值。

1xx：信息性状态码，表示服务器已接收了客户端请求，客户端可继续发送请求。

100 Continue 继续。客户端应继续其请求

101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。
只能切换到更高级的协议，例如，切换到HTTP的新版本协议

2xx：成功状态码，表示服务器已成功接收到请求并进行处理。

200 OK 请求成功。一般用于GET与POST请求

202 Accepted 已接受。已经接受请求，但未处理完成

203 Non-Authoritative Information 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本

204 No Content 成功，但不返回任何实体的主体部分

205 Reset Content 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。
可通过此返回码清除浏览器的表单域

206 Partial Content 部分内容。服务器成功处理了部分GET请求

3xx：重定向状态码，表示服务器要求客户端重定向。

300 Multiple Choices 多种选择。请求的资源可包括多个位置，
相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择

301 Moved Permanently 永久性重定向，响应报文的Location首部应该有该资源的新URL

302 Found 临时性重定向，响应报文的Location首部给出的URL用来临时定位资源

303 See Other 请求的资源存在着另一个URI，客户端应使用GET方法定向获取请求的资源

304 Not Modified 服务器内容没有更新，可以直接读取浏览器缓存

305 Use Proxy 使用代理。所请求的资源必须通过代理访问

307 Temporary Redirect 临时重定向。与302 Found含义一样。302禁止POST变换为GET，
但实际使用时并不一定，307则更多浏览器可能会遵循这一标准，但也依赖于浏览器具体实现

4xx：客户端错误状态码，表示客户端的请求有非法内容。

400 Bad Request 表示客户端请求有语法错误，不能被服务器所理解

401 Unauthonzed 表示请求未经授权，请求要求用户的身份认证。
该状态代码必须与 WWW-Authenticate 报头域一起使用

403 Forbidden 表示服务器收到请求，但是拒绝提供服务，通常会在响应正文中给出不提供服务的原因

404 Not Found 请求的资源不存在，例如，输入了错误的URL

405 Method not allowed 方法不被允许,如某请求只接受get方法，不允许post方法

408 Request Time-out 服务器等待客户端发送的请求时间过长，超时

414 Request-URI Too Large 请求的URI过长（URI通常为网址），服务器无法处理

415 Unsupported Media Type 服务器无法处理请求附带的媒体格式

499 Client Closed Request， 客户端主动断开连接。是指一次http请求在客户端指定的时间内没有返回响应，
此时，客户端会主动断开连接，此时表象为客户端无响应返回，而nginx的日志中会status code 为499。
此状态码在浏览器请求时几乎不可见，因为浏览器默认的超时时间会很长。多见于服务之间的调用，
在业务架构中常常会分层设计，拆分为不同的子系统或者微服务，这样系统之间就会常常通过http方式来请求，
并且会设置每次请求的超时时间，当请求在请求时间内所调用的上游服务无返回，
则会主动关闭连接，上游服务日志中会记录一条499。

5xx：服务器错误状态码，表示服务器未能正常处理客户端的请求而出现意外错误。

500 Internel Server Error 服务器内部错误。表示服务器发生不可预期的错误，导致无法完成客户端的请求。
日常开发中500错误几乎都是由于php脚本语法出现错误导致php-fpm无法正常执行。

502 Bad Gateway，网关错误，它往往表示网关(nginx)从上游服务器(php-fpm)中接收到的响应是无效的。
如php-fpm进程被关闭、nginx无法与php-fpm通信、
php-fpm运行时因为超过自身允许的执行时间而不能正常生成响应数据等。

503 Service Unavailable 表示服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常。
如由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中

504 Gateway Timeout，网关超时。它表示网关没有从上游及时获取响应数据。
注意它和502在超时场景下的区别，502是指上游php-fpm因为超过自身允许的执行时间而不能正常生成响应数据，
而504是指在php-fpm还未执行完成的某一时刻，由于超过了nginx自身的超时时间，
nginx则以为上游php-fpm没有按照设置时间返回响应数据就会返回504，

此时对于php-fpm而言还会继续执行下去，直到执行完成。

505 HTTP Version not supported 服务器不支持请求的HTTP协议的版本，无法完成处理

一台电脑配置无限好，可以同时打开多少个网页?

65535 – 1024 = 64511 个

拿到域名对应的IP地址之后，浏览器会以一个随机端口（1024<端口<65535）向服务器的WEB程序（常用的有httpd,nginx等）80端口发起TCP的连接请求。

1、众所周知的端口

0到1023端口,由IANA分配,紧密绑定服务.通常这些端口明确表明了某种服务协议.例如21端口为FTP服务端口

2、注册端口

1024到49151.不受IANA控制,但是已经被IANA登记了,松散绑定一些服务,可以用于其他目的

3、动态或者私有端口

49152到65535,IANA不管,所以一般机器从1024开始分配动态端口

session 与 cookie

session 与 cookie 的区别

1、存放位置：Session 保存在服务器，Cookie 保存在客户端。

2、存放的形式：Session 是以对象的形式保存在服务器，Cookie 以字符串的形式保存在客户端。

3、用途：Cookies 适合做保存用户的个人设置，爱好等，Session 适合做客户的身份验证

4、路径：Session 不能区分路径，同一个用户在访问一个网站期间，所有的 Session 在任何一个地方都可以访问到。而 Cookie 中如果设置了路径参数，那么同一个网站中不同路径下的 Cookie 互相是访问不到的。

5、安全性：Cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗，考虑到安全应当使用 session

6、大小以及数量限制：每个域名所包含的 cookie 数：IE7/8,FireFox:50 个，Opera30 个； Cookie 总大小：Firefox 和 Safari 允许 cookie 多达 4097 个字节，Opera 允许 cookie 多达 4096 个字 节，InternetExplorer 允许 cookie 多达 4095 个字节；一般认为 Session 没有大小和数量限制。

session 与 cookie 的联系

Session 需要借助 Cookie 才能正常工作。如果客户端完全禁止 Cookie，Session 将失效！因为 Session 是由应用服务器维持的一个 服务器端的存储空间，用户在连接服务器时，会由服务器生成一个唯一的 SessionID, 用该 SessionID 为标识符来存取服务器端的 Session 存储空间。而 SessionID 这一数据则是保存到客户端，用 Cookie 保存的，用户提交页面时，会将这一 SessionID 提交到服务器端，来存取 Session 数据。这一过程，是不用开发人员干预的。所以一旦客户端禁用 Cookie，那么 Session 也会失效。

当然上面失效可以通过其他方式解决，详见：禁用Cookie时，PHP共享Session文件解决方案

禁用Cookie时，PHP共享Session文件解决方案

默认SESSION配置
在默认的PHP配置中，SessionID是需要存储在Cookie中的

以PHP为例：

你第一次访问网站时，

服务端脚本中开启了Sessionsession_start();，

服务器会生成一个不重复的 SESSIONID 的文件session_id();，比如在/var/lib/php/session目录

并将返回(Response)如下的HTTP头 Set-Cookie:PHPSESSIONID=xxxxxxx

客户端接收到Set-Cookie的头，将PHPSESSIONID写入cookie

当你第二次访问页面时，所有Cookie会附带的请求头(Request)发送给服务器端

服务器识别PHPSESSIONID这个cookie，然后去session目录查找对应session文件，

找到这个session文件后，检查是否过期，如果没有过期，去读取Session文件中的配置；如果已经过期，清空其中的配置

如果客户端禁用了Cookie，那PHPSESSIONID都无法写入客户端，Session就不能用了。

并且服务端因为没有得到PHPSESSIONID的cookie，会不停的生成session_id文件

解决该问题要从cookie sessoin 和http协议入手解决解决方案如下：

1、取巧传递session_id
但是这难不倒服务端程序，聪明的程序员想到，如果一个Cookie都没接收到，基本上可以预判客户端禁用了Cookie，那将session_id附带在每个网址后面(包括POST)，
比如：

GET http://www.xx.com/index.php?session_id=xxxxx
POST http://www.xx.com/post.php?session_id=xxxxx

然后在每个页面的开头使用session_id($_GET['session_id'])，来强制指定当前session_id

这样就可以了。

聪明的你肯定想到，那将这个网站发送给别人，那么他将会以你的身份登录并做所有的事情
（目前很多订阅公众号就将openid附带在网址后面，这是同样的漏洞）。

其实不仅仅如此，cookie也可以被盗用，比如XSS注入，通过XSS漏洞获取大量的Cookie，也就是控制了大量的用户，腾讯有专门的XSS漏洞扫描机制，因为大量的QQ盗用，发广告就是因为XSS漏洞

所以Laravel等框架中，内部实现了Session的所有逻辑，并将PHPSESSIONID设置为httponly并加密，这样，前端JS就无法读取和修改这些敏感信息，降低了被盗用的风险。

2、使用常量SID
常量SID相当于“PHPSESSID=xxxxxxxxxxxxxxxxxxxxxxxxxxxx”
使用方法如下:
在超链接 action header(“Location: xx”) 可以直接拼接 SID常量即可
echo "西游记
";

3、可以配置 php.ini 文件,启用session.use_trans_sid 指定是否启用透明 SID 支持
即可以这样设置
ssssion. use_trans_sid = 1 ，这样重启apache即可生效.
设为1后，在href、action、header会自动加 SID ，但是js的跳转不会加

php 遍历目录文件方法

```php
<?php
/**********************
一个简单的目录递归函数
第一种实现办法：用dir返回对象
**********************/
function tree($directory) {
    if(is_dir($directory)) {
        //返回一个 Directory 类实例
        $mydir = dir($directory);
        echo "<ul>\n";
        //从目录句柄中读取条目
        while($file = $mydir->read()) {
            if(is_dir("$directory/$file") && $file != "." && $file != "..") {
                echo "<li><font color=\"#ff00cc\"><b>$file</b></font></li>\n";
                //递归读取目录
                tree("$directory/$file");
            } elseif ($file != "." && $file != "..") {
                echo "<li>$file</li>\n";
            }

        }
        echo "</ul>\n";
        // 释放目录句柄
        $mydir->close();
    } else {
        echo $directory . '<br>';
    }

}
//开始运行

echo "<h2>目录为粉红色</h2><br>\n";
tree("../www.test.com");
```

```php
<?php
/************************
第二种实现办法：用readdir()函数
************************/
function listDir($dir) {
    if(is_dir($dir)) {
        //打开目录句柄
        if ($dh = opendir($dir)) {
            //从目录句柄中读取条目
            while (($file = readdir($dh)) !== false) {
                if(is_dir($dir."/".$file) && $file!="." && $file!
="..") {
                    echo "<b><font color='red'>文件名：</font></b>",$f
ile,"<br><hr>";
                    listDir($dir."/".$file."/");
                }
                else {
                    if($file != "." && $file != "..") {
                        echo $file."<br>";
                    }
                }
            }
            closedir($dh);
        }
    } else {
        echo $dir . '<br>';
    }
}
//开始运行
listDir("../www.test.com");
?>
```

以数组形式返回整个文件夹中多重文件

```php
<?php
/**
```

```
     * Create a Directory Map
     *
     * Reads the specified directory and builds an array
     * representation of it. Sub-folders contained with the
     * directory will be mapped as well.
     *
     * @param   string  $source_dir     Path to source
     * @param   int $directory_depth    Depth of directories to trav
erse
     *                      (0 = fully recursive, 1 = current dir, e
tc)
     * @param   bool    $hidden         Whether to show hidden files
     * @return  array
     */
function directory_map($source_dir, $directory_depth = 0, $hidden =
 FALSE)
    {
        if ($fp = @opendir($source_dir))
        {
            $filedata   = array();
            $new_depth  = $directory_depth - 1;
            $source_dir = rtrim($source_dir, DIRECTORY_SEPARATOR).DI
RECTORY_SEPARATOR;

            while (FALSE !== ($file = readdir($fp)))
            {
                // Remove '.', '..', and hidden files [optional]
                if ($file === '.' OR $file === '..' OR ($hidden ===
 FALSE && $file[0] === '.'))
                {
                    continue;
                }

                is_dir($source_dir.$file) && $file .= DIRECTORY_SEPA
RATOR;

                if (($directory_depth < 1 OR $new_depth > 0) && is_d
ir($source_dir.$file))
                {
                    $filedata[$file] = directory_map($source_dir.$fi
```

```
le, $new_depth, $hidden);
                }
                else
                {
                    $filedata[] = $file;
                }
            }

            closedir($fp);
            return $filedata;
        }

        return FALSE;
    }

    $aa = directory_map('../www.test.com');
    var_dump($aa);
```

php 检测mysql表是否存在

pdo:

```php
<?php
$dsn = 'mysql:dbname=test;host=127.0.0.1';
$user = 'root';
$password = '';
try {
    $pdo = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    die("数据库连接失败".$e->getMessage());
}

$table = 'cy_news';
//判断表是否存在
$result = $pdo->query("SHOW TABLES LIKE '". $table."'");
$row = $result->fetchAll();

if('1' == count($row)){
    echo "Table exists";
} else {
    echo "Table does not exist";
}
?>
```

mysql:

```php
<?php
$con = mysql_connect("localhost","root","");
mysql_select_db("php_cms", $con);
$table = 'cy_news';
if(mysql_num_rows(mysql_query("SHOW TABLES LIKE '". $table."'"))==1)
{
    echo "Table exists";
```

```
} else {
    echo "Table does not exist";
}
?>
```

php curl 请求302跳转页面

若请求url已改变且设置了自动跳转，可通过设置 CURLOPT_FOLLOWLOCATION 参数实现自动请求跳转链接。若要手动判断是否跳转，可通过 curl_getinfo($ch) 方法获取具体请求信息，其中包括要跳转路径。

实例：

```
<?php
$url = 'http://auto.jrj.com.cn/';
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
//若给定url自动跳转到新的url,有了下面参数可自动获取新url内容：302跳转
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
//设置cURL允许执行的最长秒数。
curl_setopt($ch, CURLOPT_TIMEOUT, 10);
curl_setopt($ch, CURLOPT_USERAGENT, 'Mozilla/5.0 (Windows NT 6.1; WO
W64; rv:47.0) Gecko/20100101 Firefox/47.0');
curl_setopt($ch, CURLOPT_REFERER, $url);
curl_setopt($ch, CURLOPT_ENCODING, 'gzip, deflate');
$content = curl_exec($ch);
//获取请求返回码，请求成功返回200
$code = curl_getinfo($ch,CURLINFO_HTTP_CODE);
echo $code . "\n\n";

//获取一个cURL连接资源句柄的信息。
//$headers 中包含跳转的url路径
$headers = curl_getinfo($ch);
var_dump($headers);

// $content 为url请求内容
//echo "\n\n" . $content . "\n";
```

curl_getinfo($ch) 返回信息如下：

```
array (
  'url' => 'http://auto.jrj.com.cn/',
  'content_type' => 'text/html',
  'http_code' => 301,
  'header_size' => 352,
  'request_size' => 206,
  'filetime' => -1,
  'ssl_verify_result' => 0,
  'redirect_count' => 0,
  'total_time' => 0.13020600000000004,
  'namelookup_time' => 0.021582,
  'connect_time' => 0.053791999999999951,
  'pretransfer_time' => 0.05381700000000004,
  'size_upload' => 0.0,
  'size_download' => 182.0,
  'speed_download' => 1397.0,
  'speed_upload' => 0.0,
  'download_content_length' => 182.0,
  'upload_content_length' => 0.0,
  'starttransfer_time' => 0.13016799999999995,
  'redirect_time' => 0.0,
  'redirect_url' => 'http://ucheke.jrj.com.cn/',
  'primary_ip' => '211.135.187.48',
  'certinfo' => array ( ),
)
```

php pdo异常处理

pdo异常处理设置：

设置PDO::ATTR_ERRMODE ，有以下三个值：

```
PDO::ERRMODE_SILENT：  默认模式，不主动报错，需要主动以 $pdo->errorInfo()的形式获取错误信息。

PDO::ERRMODE_WARNING: 引发 E_WARNING 错误，主动报错

PDO::ERRMODE_EXCEPTION: 主动抛出 exceptions 异常，需要以try{}cath(){}输出错误信息。
```

实例：

方式一：

```php
<?php
//默认是不提示的 需要用 errorCode() errorInfo();
try{
    $pdo = new PDO("mysql:host=localhost;dbname=jikexueyuan","root","");
    //下面这句是默认设置，有无均可
    $pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_SILENT);
}catch(PDOException $e){
    die("数据库连接失败".$e->getMessage());
}
$sql = "insert into stuu values(null,'jike','w',55)";
$res = $pdo->exec($sql);
if($res){
    echo "OK";
}else{
    echo $pdo->errorCode();
```

```
    print_r($pdo->errorInfo());
}
```

方式二：

```
<?php
try{
    $pdo = new PDO("mysql:host=localhost;dbname=jikexueyuan","root","");
    //主动以警告的形式报错
    $pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_WARNING);
}catch(PDOException $e){
    die("数据库连接失败".$e->getMessage());
}
$sql = "insert into stuu values(null,'jike','w',55)";
//若有错误直接在浏览器页面中显示错误信息
$res = $pdo->exec($sql);
```

方式三：

```
<?php
try{
    $pdo = new PDO("mysql:host=localhost;dbname=jikexueyuan","root","");
    //主动抛出异常
    $pdo->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION );
}catch(PDOException $e){
    die("数据库连接失败".$e->getMessage());
}
$sql = "insert into stuu values(null,'jike','w',55)";
//用try{}catch(){}抓取异常
try{
    $res = $pdo->exec($sql);
}catch(PDOException $e){
    echo $e->getMessage();
```

```
15 }
```

php curl post请求超过1024字节解决方法

基础知识背景：

> “Expect: 100-continue”的来龙去脉：
> HTTP/1.1 协议里设计 100 (Continue) HTTP 状态码的的目的是，在客户端发送 Request Message 之前，HTTP/1.1 协议允许客户端先判定服务器是否愿意接受客户端发来的消息主体（基于 Request Headers）。即， Client 和 Server 在 Post （较大）数据之前，允许双方“握手”，如果匹配上了，Client 才开始发送（较大）数据。这么做的原因是，如果客户端直接发送请求数据，但是服务器又将该请求拒绝的话，这种行为将带来很大的资源开销。

libcurl 发送大于1024字节数据时启用“Expect:100-continue‘特性：

```
在使用 curl 做 POST 的时候，当要 POST 的数据大于 1024 字节的时候，curl 并不会直接就发起 POST 请求，而是会分为两步：
1. 发送一个请求，包含一个 "Expect: 100-continue" 头域，询问 Server 是否愿意接收数据；
2. 接收到 Server 返回的 100-continue 应答以后，才把数据 POST 给 Server；
```

PHP Curl-library 可以主动封禁此特性：

PHP curl 遵从 libcurl 的特性。由于不是所有 web servers 都支持这个特性，所以会产生各种各样的错误。如果你遇到了，可以用下面的命令封禁"Expect"头域：

```
<?php
    //添加如下head头就可传输大于1024字节请求
    curl_setopt($ch, CURLOPT_HTTPHEADER, array('Expect:'));
?>
```

一定范围内取几个不重复的随机数PHP方法

方法一：

```php
<?php
//range 是将1到42 列成一个数组

$numbers = range (1,42);

//shuffle 将数组顺序随即打乱

shuffle ($numbers);

//array_slice 取该数组中的某一段

$result = array_slice($numbers,0,3);

 print_r($result);
 ?>
```

方法二：

```php
<?php

$numbers = range (1,20);

shuffle ($numbers);

//list() 函数用于在一次操作中给一组变量赋值。
//each() 函数返回当前元素的键名和键值，并将内部指针向前移动。

while (list (, $number) = each ($numbers)) {

```

```
12      echo "$number";
13
14  }
15
16  ?>
```

方法三：

```
1   <?php
2
3   $tmp=array();
4
5   while(count($tmp)<5){
6
7       //mt_rand() 函数生成随机整数。
8
9       $tmp[]=mt_rand(1,20);
10
11      //array_unique() 函数用于移除数组中重复的值。如果两个或更多个数组值相同，只保留第一个值，其他的值被移除
12
13      $tmp=array_unique($tmp);
14
15  }
16
17  //join() 函数返回一个由数组元素组合成的字符串。
18  //join() 函数是 implode() 函数的别名。
19
20  print join(',',$tmp);
21
22  ?>
```

php文件以绝对路径引入

实例：

```
<?php
require dirname(__FILE__) . '\test.php';
$sitemapPath = dirname(dirname(dirname(dirname(__FILE__)))) . '/www.cheyun.com/sitemap';


echo __FILE__;    // 取得当前文件的绝对地址
echo dirname(__FILE__);   // 取得当前文件所在的绝对目录
echo dirname(dirname(__FILE__));  //取得当前文件的上一层目录名
echo dirname(dirname(dirname(__FILE__)));  //取得当前文件的上一层目录的上层目录名

以上代码输出：
E:\webserver\test\index.php
E:\webserver\test
E:\webserver
E:\

?>
```

建议：

```
//直接这样定义，不要中间常量，这样可以通过点击文件路径直接跳转到目的文件
require_once dirname(__FILE__) . '/config.php';
require_once dirname(__FILE__) . '/function.php';
require_once dirname(__FILE__) . '/db.class.php';
```

php array_map与array_walk使用对比

array_map():

1、array_map() 函数将用户自定义函数作用到数组中的每个值上，并返回用户自定义函数作用后的带有新值的数组,若函数作用后无返回值，则对应的新值数组中为空。

2、回调函数接受的参数数目应该和传递给 array_map() 函数的数组数目一致。

3、**提示：**您可以向函数输入一个或者多个数组。

```
若相函数输入2个数组，其函数接受参数也应该有两个,map给函数传值时，是每次从两个数组中分别取一个传给函
数。即多个数组之间是同步提交的，不是提交完一个数组再提交下一个
提交几个数组，则函数也需要有几个参数
```

4、语法： array array_map (callable $callback , array $arr1 [, array $...])

实例：

```php
<?php
//单数组形式
function myfunction($v)
{
    if ($v==="Dog")
    {
        return "Fido";
    }
    return $v;
}

$a=array("Horse","Dog","Cat");
print_r(array_map("myfunction",$a));

//多数组形式
function myfunction1($v1,$v2)
{
    if ($v1===$v2)
```

```
    {
        return "same";
    }
    return "different";
}

$a1=array("Horse","Dog","Cat");
$a2=array("Cow","Dog","Rat");
print_r(array_map("myfunction1",$a1,$a2));
?>
```

array_walk():

1、array_walk — 使用用户自定义函数对数组中的每个元素做回调处理

2、语法： bool array_walk (array &$array , callable $funcname [, mixed $userdata = NULL])

参数：

```
$array      输入的数组。
$funcname   回调函数，典型情况下 $funcname 接受两个参数。$array 参数的值作为第一个， 键名作为第二个。
$userdata   如果提供了可选参数 $userdata ，将被作为第三个参数传递给 $funcname。
```

注意:

```
如果 $funcname 需要直接作用于数组中的值，则给 funcname 的第一个参数指定为引用（添加&符号）。这样
任何对这些单元的改变也将会改变原始数组本身。
```

返回值：

```
成功时返回 TRUE， 或者在失败时返回 FALSE。
```

实例：

```php
<?php
$fruits = array("d" => "lemon", "a" => "orange", "b" => "banana", "c" => "apple");
//传引用，改变了所传参数组
function test_alter(&$item1, $key, $prefix)
{
    $item1 = "$prefix: $item1";
}

function test_print($item2, $key)
{
    echo "$key. $item2<br />\n";
}

echo "Before ...:\n";
//单数组
array_walk($fruits, 'test_print');

//带额外参数
array_walk($fruits, 'test_alter', 'fruit');
echo "... and after:\n";

array_walk($fruits, 'test_print');
```

以上例程会输出：

```
Before ...:
d. lemon
a. orange
b. banana
c. apple
 ... and after:
d. fruit: lemon
a. fruit: orange
b. fruit: banana
c. fruit: apple
```

关键点：

```
map     主要是为了得到你的回调函数处理后的新数组，要的是结果。
walk    主要是对每个参数都使用一次你的回调函数，要的是处理的过程。
walk    可以认为提供额外参数给回调函数，map不可以
walk    主要是要对数组内的每个值进行操作，操作结果影响原来的数组
map     主要是对数组中的值进行操作后返回数组，以得到一个新数组
walk    可以没有返回值 map要有，因为要填充数组
```

php解析url方法

1、利用pathinfo：

pathinfo() 函数以数组的形式返回文件路径的信息。

```
<?php
$test = pathinfo("http://localhost/index.php");
print_r($test);
?>
```

结果如下

```
Array
(
    [dirname] => http://localhost //url的路径
    [basename] => index.php  //完整文件名
    [extension] => php  //文件名后缀
    [filename] => index //文件名
)
```

2、利用parse_url：

parse_url –解析URL并返回其组成部分

```
<?php
$test = parse_url("http://localhost/index.php?name=tank&sex=1#top");
print_r($test);
?>
```

结果如下

```
Array
(
    [scheme] => http //使用什么协议
    [host] => localhost //主机名
    [path] => /index.php //路径
    [query] => name=tank&sex=1 // 所传的参数
    [fragment] => top //后面根的锚点
)
```

```
<?php
$url1 = 'http://username:password@hostname/path/fsd?arg=value#anchor';
print_r(parse_url($url1));
?>
```

输出结果为：

```
Array
(
    [scheme] => http
    [host] => hostname
    [user] => username
    [pass] => password
    [path] => /path/fsd
    [query] => arg=value
    [fragment] => anchor
)
```

3、利用basename：

basename() 函数返回路径中的文件名部分

```
<?php
$test = basename("http://localhost/index.php?name=tank&sex=1#top");
```

```
echo $test;
?>
```

结果如下

```
index.php?name=tank&sex=1#top
```

4、parse_str() 把查询字符串解析到变量中

```
parse_str(string,array)
```

参数 描述
string 必需。规定要解析的字符串。
array 可选。规定存储变量的数组的名称。该参数指示变量将被存储到数组中。
注释：如果未设置 array 参数，则由该函数设置的变量将覆盖已存在的同名变量。
注释：php.ini 文件中的 magic_quotes_gpc 设置影响该函数的输出。如果已启用，那么在 parse_str() 解析之前，变量会被 addslashes() 转换。
语法

parse_str()方法可用于解析parse_url函数生成的 query部分，两个方法相互配合使用！

```
<?php
parse_str("name=Bill&age=60");
echo $name."<br>";
echo $age;
?>
```

输出：

```
Bill

60
```

```php
<?php
parse_str("name=Bill&age=60",$myArray);
print_r($myArray);
?>
```

输出：

```
Array ( [name] => Bill [age] => 60 )
```

phpexcel读写excel表格详解

phpexcel库代码地址：https://github.com/PHPOffice/PHPExcel

代码实例：
数据导入excel表格：

可以**通过命令直接导出：**

```php
<?php
/**
 * php 输出excel文件到固定文件夹中
 * 注意输出的是 xxx.xlsx文件，
 * 若想输出 .xls文件，将下面语句替换为
 * include './PHPExcel/Writer/Excel5.php';
 * $objWriter = new PHPExcel_Writer_Excel5($objPHPExcel);
 */
include './PHPExcel.php';
//用于输出.xlsx文件
include './PHPExcel/Writer/Excel2007.php';
//创建一个excel
$objPHPExcel = new PHPExcel();


//设置excel的属性：
//创建人
$objPHPExcel->getProperties()->setCreator("wml");
//最后修改人
$objPHPExcel->getProperties()->setLastModifiedBy("wml");
//标题
$objPHPExcel->getProperties()->setTitle("wml_excel");
//题目
$objPHPExcel->getProperties()->setSubject("wml_subject");
//备注
$objPHPExcel->getProperties()->setDescription("wml_des");
//标记
$objPHPExcel->getProperties()->setKeywords("wml_key");
```

```php
//类别
$objPHPExcel->getProperties()->setCategory("wml_kind");


//设置当前的sheet
$objPHPExcel->setActiveSheetIndex(0);
//设置sheet的name
$objPHPExcel->getActiveSheet()->setTitle('test1');
//设置单元格的值
$objPHPExcel->getActiveSheet()->setCellValue('A1', 'numben1');
$objPHPExcel->getActiveSheet()->setCellValue('B1', 'numben2');
$objPHPExcel->getActiveSheet()->setCellValue('C1', 'numben3');
$objPHPExcel->getActiveSheet()->setCellValue('D1', 'numben4');
$objPHPExcel->getActiveSheet()->setCellValue('E1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('F1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('G1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('H1', 'numben5');
//从第二行开始设置记录值
for ($i = 2; $i <= 4; $i++) {
    $objPHPExcel->getActiveSheet()->setCellValue('A' . $i, convertGBK('地方'));
    $objPHPExcel->getActiveSheet()->setCellValue('B' . $i, convertGBK('002'));
    $objPHPExcel->getActiveSheet()->setCellValue('C' . $i, convertGBK('003'));
    $objPHPExcel->getActiveSheet()->setCellValue('D' . $i, convertGBK('004'));
    $objPHPExcel->getActiveSheet()->setCellValue('E' . $i, convertGBK('005'));
    $objPHPExcel->getActiveSheet()->setCellValue('F' . $i, convertGBK('006'));
    $objPHPExcel->getActiveSheet()->setCellValue('G' . $i, convertGBK('007'));
    $objPHPExcel->getActiveSheet()->setCellValue('H' . $i, convertGBK('008'));
}

//合并单元格,两个值分别是要合并范围的对角线值
$objPHPExcel->getActiveSheet()->mergeCells('A18:E22');
//分离单元格
```

```php
$objPHPExcel->getActiveSheet()->unmergeCells('A18:E22');

//设置列宽width
//自适应宽度
$objPHPExcel->getActiveSheet()->getColumnDimension('B')->setAutoSize(true);
//固定宽度
$objPHPExcel->getActiveSheet()->getColumnDimension('D')->setWidth(12);

//设置文字大小
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFont()->setSize(20);
$objPHPExcel->getActiveSheet()->getStyle('B3')->getFont()->setSize(20);

//设置align,水平垂直对齐设置
//水平对齐
$objPHPExcel->getActiveSheet()->getStyle('D1')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_RIGHT);
$objPHPExcel->getActiveSheet()->getStyle('D2')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_LEFT);
$objPHPExcel->getActiveSheet()->getStyle('D3')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_CENTER);
//垂直对齐
$objPHPExcel->getActiveSheet()->getStyle('D1')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_BOTTOM);
$objPHPExcel->getActiveSheet()->getStyle('D2')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_TOP);
$objPHPExcel->getActiveSheet()->getStyle('A2')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_CENTER);


//设置填充颜色（背景色）
$objPHPExcel->getActiveSheet()->getStyle('A1')->getFill()->setFillType(PHPExcel_Style_Fill::FILL_SOLID);
$objPHPExcel->getActiveSheet()->getStyle('A1')->getFill()->getStartColor()->setARGB('FF808080');
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFill()->setFillType(PHPExcel_Style_Fill::FILL_SOLID);
```

```php
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFill()->getStartColor()->setARGB('FF808080');


//添加图片
$objDrawing = new PHPExcel_Worksheet_Drawing();
$objDrawing->setName('Logo');
$objDrawing->setDescription('Logo');
$objDrawing->setPath('./image/sample.png');
$objDrawing->setHeight(36);
$objDrawing->setWorksheet($objPHPExcel->getActiveSheet());
$objDrawing = new PHPExcel_Worksheet_Drawing();
$objDrawing->setName('Paid');
$objDrawing->setDescription('Paid');
$objDrawing->setPath('./image/sample1.png');
$objDrawing->setCoordinates('B15');
$objDrawing->setOffsetX(110);
//旋转
$objDrawing->setRotation(25);
$objDrawing->getShadow()->setVisible(true);
$objDrawing->getShadow()->setDirection(45);
$objDrawing->setWorksheet($objPHPExcel->getActiveSheet());



//多sheet表格设置
//在默认sheet后，创建一个worksheet
$objPHPExcel->createSheet();

//设置新的的sheet
$objPHPExcel->setActiveSheetIndex(1);
//设置sheet的name
$objPHPExcel->getActiveSheet()->setTitle('test12');
//设置单元格的值（字段值）
$objPHPExcel->getActiveSheet()->setCellValue('A1', 'numben12');
$objPHPExcel->getActiveSheet()->setCellValue('B1', 'numben22');
$objPHPExcel->getActiveSheet()->setCellValue('C1', 'numben32');
$objPHPExcel->getActiveSheet()->setCellValue('D1', 'numben42');
$objPHPExcel->getActiveSheet()->setCellValue('E1', 'numben52');
$objPHPExcel->getActiveSheet()->setCellValue('F1', 'numben52');
```

```php
$objPHPExcel->getActiveSheet()->setCellValue('G1', 'numben52');
$objPHPExcel->getActiveSheet()->setCellValue('H1', 'numben52');


for ($i = 2; $i <= 14; $i++) {
    $objPHPExcel->getActiveSheet()->setCellValue('A' . $i, convertGBK('0012'));
    $objPHPExcel->getActiveSheet()->setCellValue('B' . $i, convertGBK('0022'));
    $objPHPExcel->getActiveSheet()->setCellValue('C' . $i, '32');
    $objPHPExcel->getActiveSheet()->setCellValue('D' . $i, convertGBK('0042'));
    $objPHPExcel->getActiveSheet()->setCellValue('E' . $i, convertGBK('0052'));
    $objPHPExcel->getActiveSheet()->setCellValue('F' . $i, convertGBK('0062'));
    $objPHPExcel->getActiveSheet()->setCellValue('G' . $i, convertGBK('0072'));
    $objPHPExcel->getActiveSheet()->setCellValue('H' . $i, convertGBK('0082'));
    //注意 表格中也可以插入公式，会自动计算
    $objPHPExcel->getActiveSheet()->setCellValue('I' . $i, '=SUM(C2:C4)');
}


//保存excel—2007格式
$objWriter = new PHPExcel_Writer_Excel2007($objPHPExcel);
//保存文件到具体路径
$objWriter->save("./excel/2016test.xlsx");



//可选择性对导出数据进行转码
function convertGBK($str)
{
    if(empty($str)) return '';
    //return  iconv('utf-8', 'GBK', $str);
    //return  iconv('GBK', 'utf-8', $str);
```

```
    return $str;
}
```

通过浏览器导出：

```
<?php
/**
 * php 输出excel文件到浏览器中
 * 注意输出的是 xxx.xlsx文件，
 * 若想输出 .xls文件，将下面语句替换为
 * include './PHPExcel/Writer/Excel5.php';
 * $objWriter = new PHPExcel_Writer_Excel5($objPHPExcel);
 */
include 'PHPExcel.php';
include 'PHPExcel/Writer/Excel2007.php';
//创建一个excel
$objPHPExcel = new PHPExcel();


//设置excel的属性：
//创建人
$objPHPExcel->getProperties()->setCreator("wml");
//最后修改人
$objPHPExcel->getProperties()->setLastModifiedBy("wml");
//标题
$objPHPExcel->getProperties()->setTitle("wml_excel");
//题目
$objPHPExcel->getProperties()->setSubject("wml_subject");
//备注
$objPHPExcel->getProperties()->setDescription("wml_des");
//标记
$objPHPExcel->getProperties()->setKeywords("wml_key");
//类别
$objPHPExcel->getProperties()->setCategory("wml_kind");


//设置当前的sheet
$objPHPExcel->setActiveSheetIndex(0);
```

```php
//设置sheet的name
$objPHPExcel->getActiveSheet()->setTitle('test1');
//设置单元格的值
$objPHPExcel->getActiveSheet()->setCellValue('A1', 'numben1');
$objPHPExcel->getActiveSheet()->setCellValue('B1', 'numben2');
$objPHPExcel->getActiveSheet()->setCellValue('C1', 'numben3');
$objPHPExcel->getActiveSheet()->setCellValue('D1', 'numben4');
$objPHPExcel->getActiveSheet()->setCellValue('E1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('F1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('G1', 'numben5');
$objPHPExcel->getActiveSheet()->setCellValue('H1', 'numben5');
//从第二行开始设置记录值
for ($i = 2; $i <= 4; $i++) {
    $objPHPExcel->getActiveSheet()->setCellValue('A' . $i, convertGBK('地方'));
    $objPHPExcel->getActiveSheet()->setCellValue('B' . $i, convertGBK('002'));
    $objPHPExcel->getActiveSheet()->setCellValue('C' . $i, convertGBK('003'));
    $objPHPExcel->getActiveSheet()->setCellValue('D' . $i, convertGBK('004'));
    $objPHPExcel->getActiveSheet()->setCellValue('E' . $i, convertGBK('005'));
    $objPHPExcel->getActiveSheet()->setCellValue('F' . $i, convertGBK('006'));
    $objPHPExcel->getActiveSheet()->setCellValue('G' . $i, convertGBK('007'));
    $objPHPExcel->getActiveSheet()->setCellValue('H' . $i, convertGBK('008'));
}

//合并单元格,两个值分别是要合并范围的对角线值
$objPHPExcel->getActiveSheet()->mergeCells('A18:E22');
//分离单元格
$objPHPExcel->getActiveSheet()->unmergeCells('A18:E22');

//设置列宽width
//自适应宽度
$objPHPExcel->getActiveSheet()->getColumnDimension('B')->setAutoSize(true);
```

```php
//固定宽度
$objPHPExcel->getActiveSheet()->getColumnDimension('D')->setWidth(12);

//设置文字大小
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFont()->setSize(20);
$objPHPExcel->getActiveSheet()->getStyle('B3')->getFont()->setSize(20);

//设置align,水平垂直对齐设置
//水平对齐
$objPHPExcel->getActiveSheet()->getStyle('D1')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_RIGHT);
$objPHPExcel->getActiveSheet()->getStyle('D2')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_LEFT);
$objPHPExcel->getActiveSheet()->getStyle('D3')->getAlignment()->setHorizontal(PHPExcel_Style_Alignment::HORIZONTAL_CENTER);
//垂直对齐
$objPHPExcel->getActiveSheet()->getStyle('D1')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_BOTTOM);
$objPHPExcel->getActiveSheet()->getStyle('D2')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_TOP);
$objPHPExcel->getActiveSheet()->getStyle('A2')->getAlignment()->setVertical(PHPExcel_Style_Alignment::VERTICAL_CENTER);


//设置填充颜色（背景色）
$objPHPExcel->getActiveSheet()->getStyle('A1')->getFill()->setFillType(PHPExcel_Style_Fill::FILL_SOLID);
$objPHPExcel->getActiveSheet()->getStyle('A1')->getFill()->getStartColor()->setARGB('FF808080');
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFill()->setFillType(PHPExcel_Style_Fill::FILL_SOLID);
$objPHPExcel->getActiveSheet()->getStyle('B1')->getFill()->getStartColor()->setARGB('FF808080');


//添加图片
$objDrawing = new PHPExcel_Worksheet_Drawing();
```

```php
$objDrawing->setName('Logo');
$objDrawing->setDescription('Logo');
$objDrawing->setPath('./image/sample.png');
$objDrawing->setHeight(36);
$objDrawing->setWorksheet($objPHPExcel->getActiveSheet());
$objDrawing = new PHPExcel_Worksheet_Drawing();
$objDrawing->setName('Paid');
$objDrawing->setDescription('Paid');
$objDrawing->setPath('./image/sample1.png');
$objDrawing->setCoordinates('B15');
$objDrawing->setOffsetX(110);
//旋转
$objDrawing->setRotation(25);
$objDrawing->getShadow()->setVisible(true);
$objDrawing->getShadow()->setDirection(45);
$objDrawing->setWorksheet($objPHPExcel->getActiveSheet());



//多sheet表格设置
//在默认sheet后，创建一个worksheet
$objPHPExcel->createSheet();

//设置新的的sheet
$objPHPExcel->setActiveSheetIndex(1);
//设置sheet的name
$objPHPExcel->getActiveSheet()->setTitle('test12');
//设置单元格的值（字段值）
$objPHPExcel->getActiveSheet()->setCellValue('A1', 'numben12');
$objPHPExcel->getActiveSheet()->setCellValue('B1', 'numben22');
$objPHPExcel->getActiveSheet()->setCellValue('C1', 'numben32');
$objPHPExcel->getActiveSheet()->setCellValue('D1', 'numben42');
$objPHPExcel->getActiveSheet()->setCellValue('E1', 'numben52');
$objPHPExcel->getActiveSheet()->setCellValue('F1', 'numben52');
$objPHPExcel->getActiveSheet()->setCellValue('G1', 'numben52');
$objPHPExcel->getActiveSheet()->setCellValue('H1', 'numben52');


for ($i = 2; $i <= 14; $i++) {
    $objPHPExcel->getActiveSheet()->setCellValue('A' . $i, convertG
```

```
BK('0012'));
132     $objPHPExcel->getActiveSheet()->setCellValue('B' . $i, convertG
BK('0022'));
133     $objPHPExcel->getActiveSheet()->setCellValue('C' . $i, convertG
BK('0032'));
134     $objPHPExcel->getActiveSheet()->setCellValue('D' . $i, convertG
BK('0042'));
135     $objPHPExcel->getActiveSheet()->setCellValue('E' . $i, convertG
BK('0052'));
136     $objPHPExcel->getActiveSheet()->setCellValue('F' . $i, convertG
BK('0062'));
137     $objPHPExcel->getActiveSheet()->setCellValue('G' . $i, convertG
BK('0072'));
138     $objPHPExcel->getActiveSheet()->setCellValue('H' . $i, convertG
BK('0082'));
139 }
140
141
142 //直接输出到浏览器
143 //保存excel-2007格式
144 $objWriter = new PHPExcel_Writer_Excel2007($objPHPExcel);
145
146 header("Pragma: public");
147 header("Expires: 0");
148 header("Cache-Control:must-revalidate, post-check=0, pre-check=0");
149 header("Content-Type:application/force-download");
150 header("Content-Type:application/vnd.ms-execl");
151 header("Content-Type:application/octet-stream");
152 header("Content-Type:application/download");
153 //文件名称
154 header('Content-Disposition:attachment;filename="resume.xlsx"');
155 header("Content-Transfer-Encoding:binary");
156 $objWriter->save('php://output');
157
158
159
160 //可选择性对导出数据进行转码
161 function convertGBK($str)
162 {
163     if(empty($str)) return '';
```

```
    //return  iconv('utf-8', 'GBK', $str);
    //return  iconv('GBK', 'utf-8', $str);
    return $str;
}
```

excel中数据读取：

```php
<?php
/**
 * 读取excel文件，并进行相应处理
 */

$fileName = "./excel/2016test.xlsx";

if (!file_exists($fileName)) {
    exit("文件".$fileName."不存在");
}

$startTime = time();
//方法一：
/* include './PHPExcel/Writer/Excel2007.php';
$objReader = new PHPExcel_Reader_Excel2007;
$objPHPExcel = $objReader->load($fileName); */
//方法二：
require_once './PHPExcel/IOFactory.php';
$objPHPExcel = PHPExcel_IOFactory::load($fileName);

//获取sheet表数目
$sheetCount = $objPHPExcel->getSheetCount();

//默认选中sheet0表
$sheetSelected = 0;
$objPHPExcel->setActiveSheetIndex($sheetSelected);

//获取表格行数
$rowCount = $objPHPExcel->getActiveSheet()->getHighestRow();

//获取表格列数
```

```php
$columnCount = $objPHPExcel->getActiveSheet()->getHighestColumn();

echo "<div>Sheet Count : ".$sheetCount."    行数： ".$rowCount."    列数: ".$columnCount."</div>";


$dataArr = array();


/** 循环读取每个单元格的数据 */
//行数循环
for ($row = 1; $row <= $rowCount; $row++){
    //列数循环 , 列数是以A列开始
    for ($column = 'A'; $column <= $columnCount; $column++) {
        $dataArr[] = $objPHPExcel->getActiveSheet()->getCell($column.$row)->getValue();
        echo $column.$row.":".$objPHPExcel->getActiveSheet()->getCell($column.$row)->getValue()."<br />";
    }
}

echo "<br/>消耗的内存为: ".(memory_get_peak_usage(true) / 1024 / 1024)."M";

$endTime = time();
echo "<div>解析完后，当前的时间为: ".date("Y-m-d H:i:s")."      总共消耗的时间为: ".(($endTime - $startTime))."秒</div>";
var_dump($dataArr);
$dataArr = NULL;
```

注意：

PHPExcel不仅可以读取excel；

还可以读写以下格式文件：

```
Reading

BIFF 5-8 (.xls) Excel 95 and above
Office Open XML (.xlsx) Excel 2007 and above
```

```
SpreadsheetML (.xml) Excel 2003
Open Document Format/OASIS (.ods)
Gnumeric
HTML
SYLK
CSV

Writing

BIFF 8 (.xls) Excel 95 and above
Office Open XML (.xlsx) Excel 2007 and above
HTML
CSV
PDF (using either the tcPDF, DomPDF or mPDF libraries, which need to be installed separately)
```

若仅仅是通过浏览器将数据导入excel表格中，没有复杂操作，可采用以下方法：
以下方法比较简单，不需要用其他类库：

```php
<?php
$host="127.0.0.1";
$db_user="";
$db_pass="";
$db_name="";
$timezone="Asia/Shanghai";

$link=mysql_connect($host,$db_user,$db_pass);
mysql_select_db($db_name,$link);
mysql_query("SET names UTF8");

header("Content-Type: text/html; charset=utf-8");

//需要统计的天数
$day_number = 4;

$date_now = date("Ymd",time());

// 多天汇总表格
```

```php
$table_all = 'keywords_alltoone_' . $date_now;

//导出表格
$result = mysql_query("select * from $table_all");
//第一行数据，这个要根据查询天数改变
$str = "关键词\t昨天\t前天\t大前天\t\n";
$str = iconv('utf-8','GBK',$str);
while($row=mysql_fetch_array($result)){
    $name = iconv('utf-8','GBK',$row['name']);
    $before1 = iconv('utf-8','GBK',$row['before1']);
    $before2 = iconv('utf-8','GBK',$row['before2']);
    $before3 = iconv('utf-8','GBK',$row['before3']);
    $str .= $name."\t".$before1."\t".$before2."\t".$before3."\t\n";
}
$filename = date('Ymd').'.xls';
exportExcel($filename,$str);

function exportExcel($filename,$content){
    header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
    header("Content-Type: application/vnd.ms-execl");
    header("Content-Type: application/force-download");
    header("Content-Type: application/download");
    header("Content-Disposition: attachment; filename=".$filename);
    header("Content-Transfer-Encoding: binary");
    header("Pragma: no-cache");
    header("Expires: 0");
    echo $content;
}
```

php时间戳与js时间戳的比较

单位问题：

php中取时间戳时，大多通过time()方法来获得，它获取到数值是以秒作为单位的，而javascript中从

Date对象的getTime()方法中获得的数值是以毫秒为单位 ，所以，要比较它们获得的时间是否是同一天，必须要注意

把它们的单位转换成一样，1秒=1000毫秒

时区问题：

第一点中说过，php中用time()方法来获得时间戳，通过为了显示的方便，我们在php代码中会设置好

当前服务器所在的时区，如中国大陆的服务器通常会设置成东八区，这样一样，time()方法获得的方法就不再是从

1970年1月1日0时0分0秒起，而是从1970年1月1日8时0分0秒起的了，而js中通常没有作时区相关的设置，所以是

以1970年1月1日0时0分0秒为计算的起点的，所以容易在这个地方造成不一致。

time() = Math.round(new Date().getTime()/1000-28800)

php 语言中需要注意的易错点

isset 和 empty 的注意事项

isset — 检测变量是否已设置并且非 NULL

要注意当变量被定义但赋值为null时， isset返回false

这样在判断时会出现不严谨的地方

我们来看一个例子：

```
if ($_POST['active']) {
    $postData = extractSomething($_POST);
}

// ...

if (!isset($postData)) {
    echo 'post not active';
}
```

上述代码，通常认为，假如 $*POST['active'] 返回 true，那么 postData 必将存在，因此 isset($postData) 也将返回 true。反之， isset($postData) 返回 false 的唯一可能是 $*POST['active'] 也返回 false。

然而事实并非如此！

如我所言，如果$postData 存在且被设置为 null， isset($postData) 也会返回 false 。 也就是说，即使 $_POST['active'] 返回 true， isset($postData) 也可能会返回 false 。 再一次说明上面的逻辑不严谨。

对于这种情况，虽然检查一个变量是否真的存在很重要（即：区分一个变量是未被设置还是被设置为 null）；但是使用 array_key_exists() 这个函数却是个更健壮的解决途径。

array_key_exists 比 isset更加严谨

比如，我们可以像下面这样重写上面第一个例子：

```
$data = fetchRecordFromStorage($storage, $identifier);
if (! array_key_exists('keyShouldBeSet', $data)) {
    // do this if 'keyShouldBeSet' isn't set
}
```

另外，通过结合 array_key_exists() 和 get_defined_vars()， 我们能更加可靠的判断一个变量在当前作用域中是否存在：

get_defined_vars — 返回由所有已定义变量所组成的数组

```
if (array_key_exists('varShouldBeSet', get_defined_vars())) {
    // variable $varShouldBeSet exists in current scope
}
```

empty — 检查一个变量是否为空,即变量不存在或它的值等同于FALSE

在 foreach循环后不要留下数组的引用

如果你在想遍历数组时操作数组中每个元素，在 foreach 循环中使用引用会十分方便，例如

```
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
        $value = $value * 2;
}
// $arr 现在是 array(2, 4, 6, 8)
```

问题是，如果你不注意的话这会导致一些意想不到的负面作用。在上述例子，在代码执行完以后，$value 仍保留在作用域内，并保留着对数组最后一个元素的引用。之后与 $value 相关的操作会无意中修改数组中最后一个元素的值。

你要记住 foreach 并不会产生一个块级作用域。因此，在上面例子中 $value 是一个全局引用变量。在 foreach 遍历中，每一次迭代都会形成一个对 $arr 下一个元素的引用。当遍历结束后， $value 会引用 $arr 的最后一个元素，并保留在作用域中

这种行为会导致一些不易发现的，令人困惑的bug，以下是一个例子

```
<?php
$array = [1, 2, 3];
echo implode(',', $array), "\n";

foreach ($array as &$value) {}    // 通过引用遍历
echo implode(',', $array), "\n";

foreach ($array as $value) {}     // 通过赋值遍历
echo implode(',', $array), "\n";
```

以上代码会输出

1,2,3
1,2,3
1,2,2
你没有看错，最后一行的最后一个值是 2 ，而不是 3 ，为什么?

在完成第一个 foreach 遍历后， $array 并没有改变，但是像上述解释的那样， $value 留下了一个对 $array 最后一个元素的危险的引用（因为 foreach 通过引用获得 $value ）

这导致当运行到第二个 foreach ，这个"奇怪的东西"发生了。当 $value 通过赋值获得， foreach 按顺序复制每个 $array 的元素到 $value 时，第二个 foreach 里面的细节是这样的

第一步：复制 $array[0] （也就是 1 ）到 $value （$value 其实是 $array最后一个元素的引用，即 $array[2]），所以 $array[2] 现在等于 1。所以 $array 现在包含 [1, 2, 1]
第二步：复制 $array[1]（也就是 2 ）到 $value （ $array[2] 的引用），所以 $array[2] 现在等于 2。所以 $array 现在包含 [1, 2, 2]
第三步：复制 $array[2]（现在等于 2 ） 到 $value （ $array[2] 的引用），所以 $array[2] 现在等于 2 。所以 $array 现在包含 [1, 2, 2]
为了在 foreach 中方便的使用引用而免遭这种麻烦，请在 foreach 执行完毕后 unset() 掉这个保留着引用的变量。例如

```php
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
unset($value);   // $value 不再引用 $arr[3]
```

参考文章：https://juejin.im/entry/5ac202605188255cb32e4daf?utm_medium=be&utm_source=weixinqun

PHP垃圾回收机制

引用计数

每个php变量存在一个叫"zval"的变量容器中。一个zval变量容器，除了包含变量的类型和值，还包括两个字节的额外信息。第一个是"is_ref"，是个bool值，用来标识这个变量是否是属于引用集合(reference set)。通过这个字节，php引擎才能把普通变量和引用变量区分开来，由于php允许用户通过使用&来使用自定义引用，zval变量容器中还有一个内部引用计数机制，来优化内存使用。第二个额外字节是"refcount"，用以表示指向这个zval变量容器的变量(也称符号即symbol)个数。所有的符号存在一个符号表中，其中每个符号都有作用域(scope)，那些主脚本(比如：通过浏览器请求的的脚本)和每个函数或者方法也都有作用域。

当一个变量被赋常量值时，就会生成一个zval变量容器，如下例这样：

1 生成一个新的zval容器

```
<?php
$a = "new string";
?>
```

在上例中，新的变量a，是在当前作用域中生成的。并且生成了类型为 string 和值为new string的变量容器。在额外的两个字节信息中，"is_ref"被默认设置为 FALSE，因为没有任何自定义的引用生成。"refcount" 被设定为 1，因为这里只有一个变量使用这个变量容器. 注意到当"refcount"的值是1时，"is_ref"的值总是FALSE. 如果你已经安装了» Xdebug，你能通过调用函数 xdebug_debug_zval()显示"refcount"和"is_ref"的值。

2 显示zval信息

```
<?php
xdebug_debug_zval('a');
?>
```

以上例程会输出：

```
a: (refcount=1, is_ref=0)='new string'
```

把一个变量赋值给另一变量将增加引用次数(refcount).

3 增加一个zval的引用计数

```
<?php
$a = "new string";
$b = $a;
xdebug_debug_zval( 'a' );
?>
```

以上例程会输出：

```
a: (refcount=2, is_ref=0)='new string'
```

这时，引用次数是2，因为同一个变量容器被变量 a 和变量 b关联.当没必要时，php不会去复制已生成的变量容器。变量容器在”refcount“变成0时就被销毁. 当任何关联到某个变量容器的变量离开它的作用域(比如：函数执行结束)，或者对变量调用了函数 unset()时，”refcount“就会减1，下面的例子就能说明:

4 减少引用计数

```
<?php
$a = "new string";
$c = $b = $a;
xdebug_debug_zval( 'a' );
unset( $b, $c );
xdebug_debug_zval( 'a' );
?>
```

以上例程会输出：

```
a: (refcount=3, is_ref=0)='new string'
a: (refcount=1, is_ref=0)='new string'
```

如果我们现在执行 unset($a);，包含类型和值的这个变量容器就会从内存中删除。

复合类型

当考虑像 array和object这样的复合类型时，事情就稍微有点复杂. 与 标量(scalar)类型的值不同，array 和 object类型的变量把它们的成员或属性存在自己的符号表中。这意味着下面的例子将生成三个zval变量容器。

5 Creating a array zval

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
xdebug_debug_zval( 'a' );
?>
```

以上例程的输出类似于：

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=1, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42
)
```

这三个zval变量容器是: a，meaning和 number。增加和减少”refcount”的规则和上面提到的一样. 下面，我们在数组中再添加一个元素,并且把它的值设为数组中已存在元素的值:

6 添加一个已经存在的元素到数组中

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
```

```
xdebug_debug_zval( 'a' );
?>
```

以上例程的输出类似于：

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=2, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42,
   'life' => (refcount=2, is_ref=0)='life'
)
```

带有引用的简单数组的zval
从以上的xdebug输出信息，我们看到原有的数组元素和新添加的数组元素关联到同一个"refcount"2的zval变量容器. 尽管 Xdebug的输出显示两个值为'life'的 zval 变量容器，其实是同一个。 函数xdebug_debug_zval()不显示这个信息，但是你能通过显示内存指针信息来看到。

删除数组中的一个元素，就是类似于从作用域中删除一个变量. 删除后,数组中的这个元素所在的容器的“refcount”值减少，同样，当“refcount”为0时，这个变量容器就从内存中被删除，下面又一个例子可以说明：

7 从数组中删除一个元素

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
unset( $a['meaning'], $a['number'] );
xdebug_debug_zval( 'a' );
?>
```

以上例程的输出类似于：

```
a: (refcount=1, is_ref=0)=array (
   'life' => (refcount=1, is_ref=0)='life'
)
```

现在，当我们添加一个数组本身作为这个数组的元素时，事情就变得有趣，下个例子将说明这个。例中我们加入了引用操作符，否则php将生成一个复制。

8 把数组作为一个元素添加到自己

```
<?php
$a = array( 'one' );
$a[] =& $a;
xdebug_debug_zval( 'a' );
?>
```

以上例程的输出类似于：

```
a: (refcount=2, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=2, is_ref=1)=...
)
```

自引用(curcular reference,自己是自己的一个元素)的数组的zval
能看到数组变量 (a) 同时也是这个数组的第二个元素(1) 指向的变量容器中“refcount”为 2。上面的输出结果中的"..."说明发生了递归操作, 显然在这种情况下意味着"..."指向原始数组。

跟刚刚一样，对一个变量调用unset，将删除这个符号，且它指向的变量容器中的引用次数也减1。所以，如果我们在执行完上面的代码后，对变量$a调用unset, 那么变量 $a 和数组元素 "1" 所指向的变量容器的引用次数减1, 从"2"变成"1". 下例可以说明:

9 Unsetting $a

```
(refcount=1, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=1, is_ref=1)=...
)
```

清理变量容器的问题

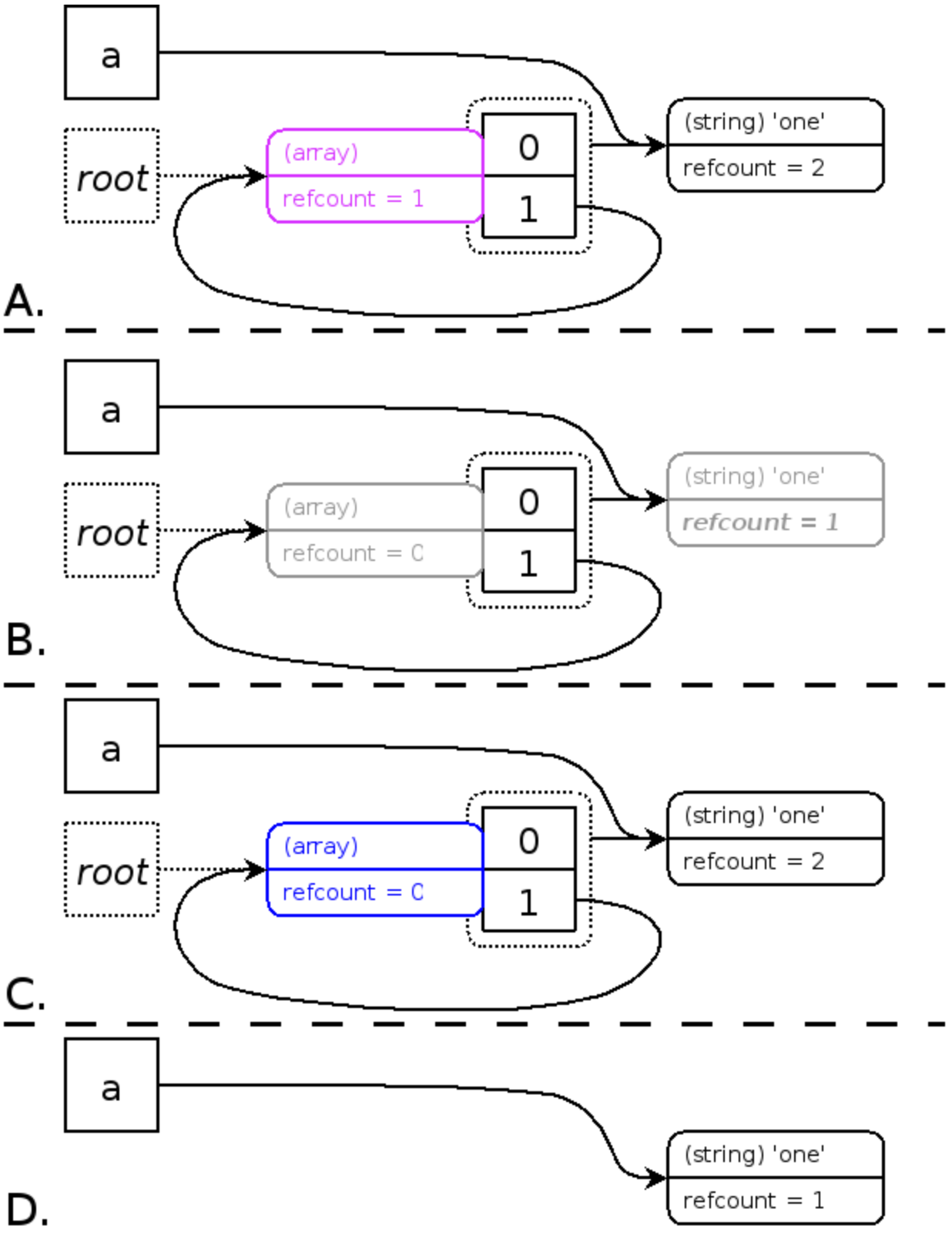
尽管不再有某个作用域中的任何符号指向这个结构(就是变量容器)，由于数组元素“1”仍然指向数组本身，所以这个容器不能被清除 。因为没有另外的符号指向它，用户没有办法清除这个结构，结果就会导致内存泄漏。庆幸的是，php将在脚本执行结束时清除这个数据结构，但是在php清除之前，将耗费不少内存。如果你要实现分析算法，或者要做其他像一个子元素指向它的父元素这样的事情，这种情况就会经常发生。当然，同样的情况也会发生在对象上，实际上对象更有可能出现这种情况，因为对象总是隐式的被引用。

如果上面的情况发生仅仅一两次倒没什么，但是如果出现几千次，甚至几十万次的内存泄漏，这显然是个大问题。这样的问题往往发生在长时间运行的脚本中，比如请求基本上不会结束的守护进程(deamons)或者单元测试中的大的套件(sets)中。后者的例子：在给巨大的eZ(一个知名的PHP Library) 组件库的模板组件做单元测试时，就可能会出现问题。有时测试可能需要耗用2GB的内存，而测试服务器很可能没有这么大的内存。

回收周期

传统上，像以前的 php 用到的引用计数内存机制，无法处理循环的引用内存泄漏。然而 5.3.0 PHP 使用引用计数系统中的同步周期回收中的同步算法，来处理这个内存泄漏问题。

首先，我们先要建立一些基本规则，如果一个引用计数增加，它将继续被使用，当然就不再在垃圾中。如果引用计数减少到零，所在变量容器将被清除(free)。就是说，仅仅在引用计数减少到非零值时，才会产生垃圾周期(garbage cycle)。其次，在一个垃圾周期中，通过检查引用计数是否减1，并且检查哪些变量容器的引用次数是零，来发现哪部分是垃圾。



为避免不得不检查所有引用计数可能减少的垃圾周期，这个算法把所有可能根(possible roots 都是zval变量容器),放在根缓冲区(root buffer)中(用紫色来标记，称为疑似垃圾)，这样可以同时确保每个可能的垃圾根(possible garbage root)在缓冲区中只出现一次。仅仅在根缓冲区满了时，才对缓冲区内部所有不同的变量容器执行垃圾回收操作。看上图的步骤 A。

在步骤 B 中，模拟删除每个紫色变量。模拟删除时可能将不是紫色的普通变量引用数减"1"，如果某个普通变量引用计数变成0了，就对这个普通变量再做一次模拟删除。每个变量只能被模拟删除一次，模拟删除后标记为灰（原文说确保不会对同一个变量容器减两次"1",不对的吧）。

在步骤 C 中，模拟恢复每个紫色变量。恢复是有条件的，当变量的引用计数大于0时才对其做模拟恢复。同样每个变量只能恢复一次，恢复后标记为黑，基本就是步骤 B 的逆运算。这样剩下的一堆没能恢复的就是该删除的蓝色节点了，在步骤 D 中遍历出来真的删除掉。

算法中都是模拟删除、模拟恢复、真的删除，都使用简单的遍历即可（最典型的深搜遍历）。复杂度为执行模拟操作的节点数正相关，不只是紫色的那些疑似垃圾变量。

现在，你已经对这个算法有了基本了解，我们回头来看这个如何与PHP集成。默认的，PHP的垃圾回收机制是打开的，然后有个 php.ini 设置允许你修改它：zend.enable_gc 。

当垃圾回收机制打开时，每当根缓存区存满时，就会执行上面描述的循环查找算法。根缓存区有固定的大小，可存10,000个可能根,当然你可以通过修改PHP源码文件Zend/zend_gc.c中的常量GC_ROOT_BUFFER_MAX_ENTRIES，然后重新编译PHP，来修改这个10,000值。当垃圾回收机制关闭时，循环查找算法永不执行，然而，可能根将一直存在根缓冲区中，不管在配置中垃圾回收机制是否激活。

当垃圾回收机制关闭时，如果根缓冲区存满了可能根，更多的可能根显然不会被记录。那些没被记录的可能根，将不会被这个算法来分析处理。如果他们是循环引用周期的一部分，将永不能被清除进而导致内存泄漏。

即使在垃圾回收机制不可用时，可能根也被记录的原因是，相对于每次找到可能根后检查垃圾回收机制是否打开而言，记录可能根的操作更快。不过垃圾回收和分析机制本身要耗不少时间。

除了修改配置zend.enable_gc ，也能通过分别调用gc_enable() 和 gc_disable()函数来打开和关闭垃圾回收机制。调用这些函数，与修改配置项来打开或关闭垃圾回收机制的效果是一样的。即使在可能根缓冲区还没满时，也能强制执行周期回收。你能调用gc_collect_cycles()函数达到这个目的。这个函数将返回使用这个算法回收的周期数。

允许打开和关闭垃圾回收机制并且允许自主的初始化的原因，是由于你的应用程序的某部分可能是高时效性的。在这种情况下，你可能不想使用垃圾回收机制。当然，对你的应用程序的某部分关闭垃圾回收机制，是在冒着可能内存泄漏的风险，因为一些可能根也许存不进有限的根缓冲区。因此，就在你调用gc_disable()函数释放内存之前，先调用gc_collect_cycles()函数可能比较明智。因为这将清除已存放在根缓冲区中的所有可能根，然后在垃圾回收机制被关闭时，可留下空缓冲区以有更多空间存储可能根。

性能方面考虑的因素

在上面我们已经简单的提到：回收可能根有细微的性能上影响，但这是把PHP 5.2与PHP 5.3比较时才有的。尽管在PHP 5.2中，记录可能根相对于完全不记录可能根要慢些，而PHP 5.3中对 PHP run-time 的其他修改减少了这个性能损失。

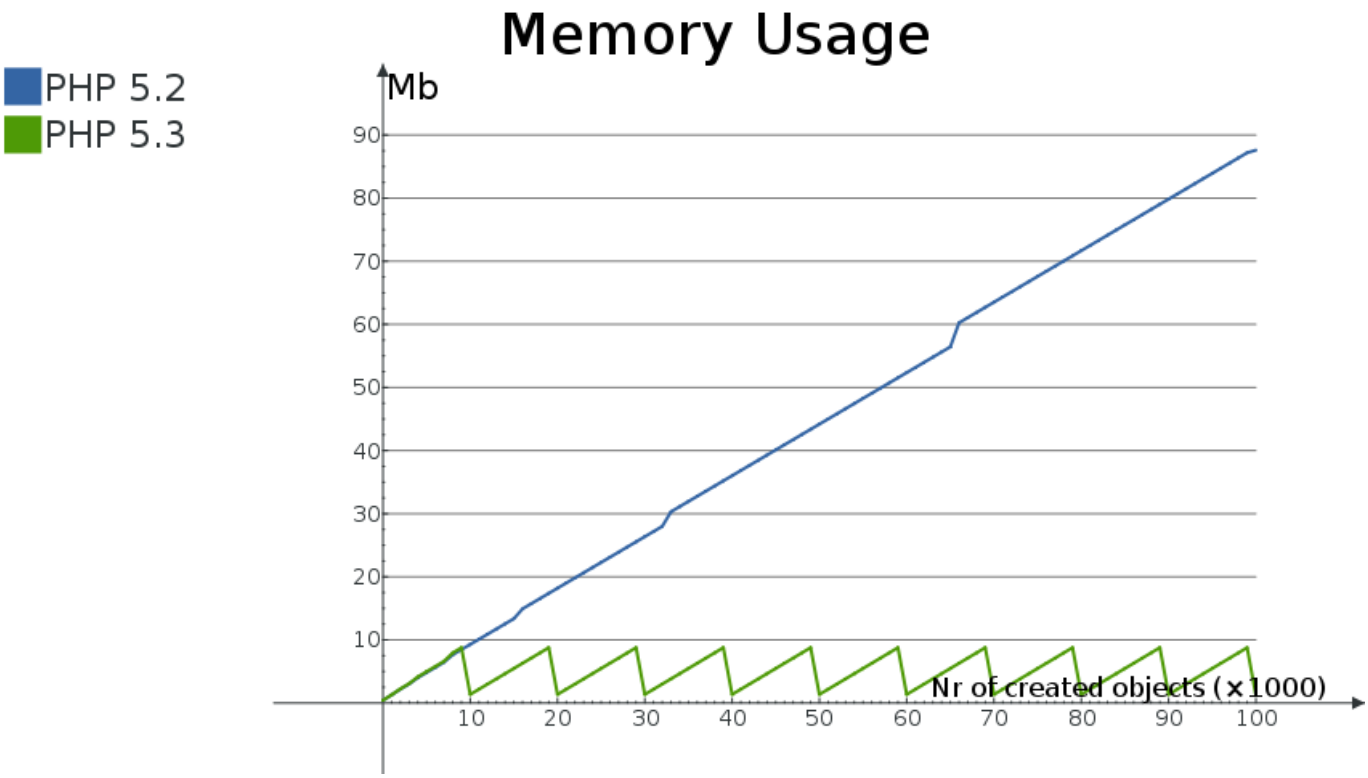
这里主要有两个领域对性能有影响。第一个是内存占用空间的节省，另一个是垃圾回收机制执行内存清理时的执行时间增加(run-time delay)。我们将研究这两个领域。

内存占用空间的节省

首先，实现垃圾回收机制的整个原因是为了，一旦先决条件满足，通过清理循环引用的变量来节省内存占用。在PHP执行中，一旦根缓冲区满了或者调用gc_collect_cycles() 函数时，就会执行垃圾回收。在下图中，显示了下面脚本分别在PHP 5.2 和 PHP 5.3环境下的内存占用情况，其中排除了脚本启动时PHP本身占用的基本内存。

1 Memory usage example

```
<?php
class Foo
{
    public $var = '3.1415962654';
}

$baseMemory = memory_get_usage();

for ( $i = 0; $i <= 100000; $i++ )
{
    $a = new Foo;
    $a->self = $a;
    if ( $i % 500 === 0 )
    {
        echo sprintf( '%8d: ', $i ), memory_get_usage() - $baseMemor
y, "\n";
    }
}
?>
```



在这个很理论性的例子中，我们创建了一个对象，这个对象中的一个属性被设置为指回对象本身。在循环的下一个重复(iteration)中，当脚本中的变量被重新复制时，就会发生典型性的内存泄漏。在这个例子中，两个变量容器是泄漏的(对象容器和属性容器)，但是仅仅能找到一个可能根：就是被unset的那个变量。在10,000次重复后(也就产生总共10,000个可能根)，当根缓冲区满时，就执行垃圾回收机制，并且释放那些关联的可能根的内存。这从PHP 5.3的锯齿型内存占用图中很容易就能看到。每次执行完10,000次重复后，执行垃圾回收，并释放相关的重复使用的引用变量。在这个例子中由于泄漏的数据结构非常简单，所以垃圾回收机制本身不必做太多工作。从这个图表中，你能看到 PHP 5.3的最大内存占用大概是9 Mb，而PHP 5.2的内存占用一直增加。

执行时间增加

垃圾回收影响性能的第二个领域是它释放已泄漏的内存耗费的时间。为了看到这个耗时时多少，我们稍微改变了上面的脚本，有更多次数的重复并且删除了循环中的内存占用计算，第二个脚本代码如下：

2 GC性能影响

```
<?php
class Foo
{
    public $var = '3.1415962654';
}
```

```
6
7 for ( $i = 0; $i <= 1000000; $i++ )
8 {
9     $a = new Foo;
10    $a->self = $a;
11 }
12
13 echo memory_get_peak_usage(), "\n";
14 ?>
```

我们将运行这个脚本两次，一次通过配置zend.enable_gc 打开垃圾回收机制时，另一次是它关闭时。

3 执行以上脚本

```
1 time php -dzend.enable_gc=0 -dmemory_limit=-1 -n example2.php
2 # and
3 time php -dzend.enable_gc=1 -dmemory_limit=-1 -n example2.php
```

在我的机器上，第一个命令持续执行时间大概为10.7秒，而第二个命令耗费11.4秒。时间上增加了7%。然而，执行这个脚本时内存占用的峰值降低了98%，从931Mb 降到 10Mb。这个基准不是很科学，或者并不能代表真实应用程序的数据，但是它的确显示了垃圾回收机制在内存占用方面的好处。好消息就是，对这个脚本而言，在执行中出现更多的循环引用变量时，内存节省的更多的情况下，每次时间增加的百分比都是7%。

PHP内部 GC 统计信息

在PHP内部，可以显示更多的关于垃圾回收机制如何运行的信息。但是要显示这些信息，你需要先重新编译PHP使benchmark和data-collecting code可用。你需要在按照你的意愿运行./configure前，把环境变量CFLAGS设置成-DGC_BENCH=1。下面的命令串就是做这个事：

4 重新编译PHP以启用GC benchmarking

```
1 export CFLAGS=-DGC_BENCH=1
2 ./config.nice
3 make clean
```

```
4 make
```

当你用新编译的PHP二进制文件来重新执行上面的例子代码，在PHP执行结束后，你将看到下面的信息：

5 GC 统计数据

```
 1 GC Statistics
 2 ----------
 3 Runs:               110
 4 Collected:          2072204
 5 Root buffer length: 0
 6 Root buffer peak:   10000
 7
 8       Possible            Remove from  Marked
 9         Root    Buffered     buffer     grey
10       --------  --------  -----------  ------
11 ZVAL   7175487   1491291    1241690   3611871
12 ZOBJ  28506264   1527980     677581   1025731
```

主要的信息统计在第一个块。你能看到垃圾回收机制运行了110次，而且在这110次运行中，总共有超过两百万的内存分配被释放。只要垃圾回收机制运行了至少一次，根缓冲区峰值(Root buffer peak)总是10000.

结论

通常，PHP中的垃圾回收机制，仅仅在循环回收算法确实运行时会有时间消耗上的增加。但是在平常的(更小的)脚本中应根本就没有性能影响。

然而，在平常脚本中有循环回收机制运行的情况下，内存的节省将允许更多这种脚本同时运行在你的服务器上。因为总共使用的内存没达到上限。

这种好处在长时间运行脚本中尤其明显，诸如长时间的测试套件或者daemon脚本此类。同时，对通常比Web脚本运行时间长的» PHP-GTK应用程序，新的垃圾回收机制，应该会大大改变一直以来认为内存泄漏问题难以解决的看法。

日志实时监控php脚本

若有进程a实时写入文件rateReport.log，想用进程b去实时监控rateReport.log并分析，可用如下php代码实现：

```php
$shell = 'tail -f /alidata/log/nginx/access/rateReport.log 2>&1';

$handle = popen($shell, 'r');
$i = 0;
while(!feof($handle)) {
    $i++;
    $buffer = fgets($handle);
    var_dump($buffer);
    flush();
    if($i == 1000){
        echo date('Y-m-d H:i:s') . "\t" . $i . PHP_EOL;
        $i = 0;
    }
}
pclose($handle);
```

注意：

1、若rateReport.log文件需要每日切割，若用logrotate进行切割，要用copytruncate方式，因为日志切割时tail -f 一直监控老文件

2、守护进程用supervisor进行管理

php保留n位小数方法

round 对浮点数进行四舍五入

```
$num = 10.4567;
echo round($num,2); //10.46
```

sprintf格式化字符串,四舍五入

```
$num = 10.4567;
$format_num = sprintf("%.2f",$num);
echo $format_num; //10.46
```

number_format 以千位分隔符方式格式化一个数字,四舍五入

```
$num = 10.4567;
echo number_format($num, 2); //10.46
//或者如下
echo number_format($num, 2, '.', ''); //10.46

使用说明:
string number_format ( float $number [, int $decimals = 0 ] )
string number_format ( float $number , int $decimals = 0 , string $dec_point = "." , string $thousands_sep = "," )
本函数可以接受1个、2个或者4个参数（注意：不能是3个）:

如果只提供第一个参数，number的小数部分会被去掉 并且每个千位分隔符都是英文小写逗号","

如果提供两个参数，number将保留小数点后的位数到你设定的值，其余同楼上

如果提供了四个参数，number 将保留decimals个长度的小数部分，小数点被替换为dec_po
```

```
    int，千位分隔符替换为thousands_sep
```

舍去法保留n位小数，可使用精度计算方法

```
1 $num = 10.4567;
2 echo bcadd ( $num, 0, 2);   //10.45
```

floor 舍去法取整

```
1 echo floor(4.3);   // 4
2 echo floor(9.999); // 9
3 echo floor(-3.14); // -4
```

ceil 进一法取整

```
1 <?php
2 echo ceil(4.3);    // 5
3 echo ceil(9.999);  // 10
4 echo ceil(-3.14);  // -3
```

php精度计算

如果用php的+-*/计算浮点数的时候，可能会遇到一些计算结果错误的问题，比如echo intval(0.58*100);会打印57，而不是58，这个其实是计算机底层二进制无法精确表示浮点数的一个bug。为解决这个问题基本上大部分语言都提供了精准计算的类库或函数库，比如php有BC高精确度函数库。

例子

```
<?php
$f = 0.58;
var_dump(intval($f * 100)); //为啥输出57?>
```

原因说明

```
浮点数, 以64位的长度(双精度)为例, 会采用1位符号位(E), 11指数位(Q), 52位尾数(M)表示(一共64位).

符号位: 最高位表示数据的正负, 0表示正数, 1表示负数。

指数位: 表示数据以2为底的幂, 指数采用偏移码表示

尾数: 表示数据小数点后的有效数字.

这里的关键点就在于, 小数在二进制的表示, 关于小数如何用二进制表示, 大家可以百度一下, 我这里就不再赘述, 我们关键的要了解, 0.58 对于二进制表示来说, 是无限长的值(下面的数字省掉了隐含的1)..

   0.58的二进制表示基本上(52位)是: 0010100011110101110000101000111101011000010100011110.57的二进制表示基本上(52位)是: 0010000111010111000010100011110101110000101000111110而两者的二进制, 如果只是通过这52位计算的话,分别是:

   0.58 -> 0.579999999999999960.57 -> 0.5699999999999999至于0.58 * 100的具体浮点数乘法, 我们不考虑那么细, 有兴趣的可以看(Floating point), 我们就模糊
```

```
的以心算来看… 0.58 * 100 = 57.999999999

    那你intval一下, 自然就是57了….

    可见, 这个问题的关键点就是: “你看似有穷的小数, 在计算机的二进制表示里却是无穷的”

    显然简单的十进制分数如同 0.1 或 0.7 不能在不丢失一点点精度的情况下转换为内部二进制的格式。这就会造成混乱的结果：例如，floor((0.1+0.7)*10) 通常会返回 7 而不是预期中的 8，因为该结果内部的表示其实是类似 7.9999999999…。

    这和一个事实有关，那就是不可能精确的用有限位数表达某些十进制分数。例如，十进制的 1/3 变成了 0.3333333. . .。

    所以永远不要相信浮点数结果精确到了最后一位，也永远不要比较两个浮点数是否相等。如果确实需要更高的精度，应该使用任意精度数学函数或者 gmp 函数
```

精度计算方法

为解决上面问题，可以用精度计算方法

```
bcadd – 将两个高精度数字相加

bccomp – 比较两个高精度数字，返回-1, 0, 1

bcdiv – 将两个高精度数字相除

bcmod – 求高精度数字余数

bcmul – 将两个高精度数字相乘

bcpow – 求高精度数字乘方

bcpowmod – 求高精度数字乘方求模，数论里非常常用

bcscale – 配置默认小数点位数，相当于就是Linux bc中的”scale=”

```

```
bcsqrt – 求高精度数字平方根

bcsub – 将两个高精度数字相减
```

php BC高精确度函数库包含了：相加，比较，相除，相减，求余，相乘，n次方，配置默认小数点数目，求平方。这些函数在涉及到有关金钱计算时比较有用，比如电商的价格计算。

注意 ： 精度计算是舍去法保留n位小数

base64 & urlbase64 介绍

Base64是一种用64个字符来表示任意二进制数据的方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit。然后将24bit重新划为4组，每组正好6个bit。
这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用\x00字节在末尾补足后，再在编码的末尾加上1个或2个=号，表示补了多少字节，解码的时候，会自动去掉。

注意：

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种”url safe”的base64编码，其实就是把字符+和/分别变成-和_：

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

实例：

```
<?php
# urlbase64_encode
function base64url_encode($input) {
    return str_replace('=', '', strtr(base64_encode($input), '+/',
```

```
    '-_'));
}

# urlbase64_decode
function base64url_decode($input) {
    $remainder = strlen($input) % 4;
    if ($remainder) {
        $padlen = 4 - $remainder;
        $input .= str_repeat('=', $padlen);
    }
    return base64_decode(strtr($input, '-_', '+/'));
}
```

QueryList 爬虫乱码处理方法

使用QueryList内置的乱码解决方案

使用编码转换插件，设置输入输出编码

```
$html =<<<STR
<div>
    <p>这是内容</p>
</div>
STR;
$rule = [
    'content' => ['div>p:last','text']
];
$data = QueryList::html($html)->rules($rule)
                ->encoding('UTF-8','GB2312')->query()->getData();
```

设置输入输出编码,并移除html头部

如果设置输入输出参数仍然无法解决乱码，那就使用 removeHead()方法移除html头部

```
$html =<<<STR
<div>
    <p>这是内容</p>
</div>
STR;
$rule = [
    'content' => ['div>p:last','text']
];
$data = QueryList::html($html)->rules($rule)
                ->removeHead()->query()->getData();
// 或者
$data = QueryList::html($html)->rules($rule)
```

```
13                 ->encoding('UTF-8','GB2312')->removeHead()->query()->getData();
```

自己手动转码页面，然后再把页面传给QueryList

```
1 $url = 'http://top.etao.com/level3.php?spm=0.0.0.0.Ql86zl&cat=16&show=focus&up=true&ad_id=&am_id=&cm_id=&pm_id=';
2 //手动转码
3 $html = iconv('GBK','UTF-8',file_get_contents($url));
4 $data = QueryList::html($html)->rules([
5     "text" => [".title a","text"]
6     ])->query()->getData();
7 print_r($data);
```

无法获取meta编码，手动转换

在抓取某些网页时，本地是uff-8的，目标页面是utf-8编码，抓过来后以utf-8的编码输出会是乱码。这种情况原因之一就是在抓取目标页面代码时，未能获取到meta中的编码信息时一律转换为ISO-8859-1编码。所以需要手动转一下编码。

```
1 $data = $ql::html($html)->rules([
2     'title' => ['div[class="novelslistss"]>ul>li>span[class="s2"]>a', 'text', '', function ($content) {
3         # 将utf-8 转为 ISO-8859-1
4         $content = mb_convert_encoding($content, 'ISO-8859-1', 'utf-8');
5         return $content;
6     }],
7 ])->query()->getData();
```

php7 升级点

1、性能与底层

PHP7 速度是 PHP5.6 的两倍以上。

新版 Zend 引擎，Zend 引擎的优化

2、指定函数的传参类型

PHP开始以可选的方式支持类型定义。除此之外，还引入了一个开关指令`declare(strict_type=1)`;，当这个指令一旦开启，将会强制当前文件下的程序遵循严格的函数传参类型和返回类型。
如果不开启strict_type，即`declare(strict_type=0)`,PHP将会尝试帮你转换成要求的类型，而开启之后，会改变PHP就不再做类型转换，类型不匹配就会抛出错误。

```php
<?php

//declare(strict_types=1);

function test(string $a,int $b){
   echo $a, $b;
}
test('asd', 1);
```

3、返回值类型声明

指定返回值是整型

```php
<?php

function f($a):int{
    return '123';
```

```
}
var_dump(f(1)); # 输出  int(123)
```

4、移除 mysql_* 类函数

7.0开始

5、更多的Error变为可捕获的Exception

PHP7实现了一个全局的throwable接口，原来的Exception和部分Error都实现了这个接口（interface）， 以接口的方式定义了异常的继承结构。于是，PHP7中更多的Error变为可捕获的Exception返回给开发者，如果不进行捕获则为Error，如果捕获就变为一个可在程序内处理的Exception。这些可被捕获的Error通常都是不会对程序造成致命伤害的Error，例如函数不存。PHP7进一步方便开发者处理，让开发者对程序的掌控能力更强。因为在默认情况下，Error会直接导致程序中断，而PHP7则提供捕获并且处理的能力，让程序继续执行下去，为程序员提供更灵活的选择。

6、JSON_THROW_ON_ERROR

解析 JSON 响应数据，有 json_encode() 以及 json_decode() 两个函数可供使用。不幸的是，它们都没有恰当的错误抛出表现。json_encode 失败时仅会返回 false；json_decode 失败时则会返回 null，而 null 可作为合法的 JSON 数值。唯一获取错误的方法是，调用 json_last_error() 或 json_last_error_msg()，它们将分别返回机器可读和人类可读的全局错误状态。

从PHP 7.3 开始 为 JSON 函数新增 JSON_THROW_ON_ERROR 常量用于忽略全局错误状态。当错误发生时，JSON 函数将会抛出 JsonException 异常，异常消息（message）为 json_last_error() 的返回值，异常代码（code）为 json_last_error_msg() 的返回值。如下是调用例子：

```
json_encode($data, JSON_THROW_ON_ERROR);

json_decode("invalid json", null, 512, JSON_THROW_ON_ERROR);

// 抛出 JsonException 异常
```

7、非常常用且开销不大的的函数直接变成了引擎支持的opcode

```
call_user_function(_array) => ZEND_INIT_USER_CALL
is_int、is_string、is_array、... => ZEND_TYPE_CHECK
strlen => ZEND_STRLEN
defined => ZEND+DEFINED
```

8、核心排序的优化

PHP5（zend_qsort）
快速排序（非稳定排序）

PHP7（zend_sort）
快速排序+选择排序（稳定排序）

小于16个元素的使用选择排序，大于16个按照16个为单位去分割，分别使用选择排序，然后再全部合起来使用快速排序。排序较之前相比，内部元素由非稳定排序变成稳定排序，减少元素的交换次数，减少对内存的操作次数，性能提升40%

9、太空船操作符（组合比较符）

太空船操作符用于比较两个表达式。 当$a小于、 等于或大于$b时它分别返回-1、 0或1。

```
// Integers
echo 1 <=> 1; // 0
echo 1 <=> 2; // -1
echo 2 <=> 1; // 1
// Floatsecho 1.5 <=> 1.5; // 0
echo 1.5 <=> 2.5; // -1
echo 2.5 <=> 1.5; // 1
// Strings
echo "a" <=> "a"; // 0
echo "a" <=> "b"; // -1
echo "b" <=> "a"; // 1
```

10、null合并运算符

```
$username = isset($_GET['user']) ? $_GET['user'] : 'nobody';    //php5
$username = $_GET['user'] ?? 'nobody';    //php7
```

11、匿名类，参照匿名函数理解

```
(new class {
    public function a(){
        echo 123;
    }
 })->a();
```

12、统一的语法变量

```
$$foo['bar']['baz']

PHP5: ${$foo[‘bar’]['baz']}
PHP7: ($$foo)[‘bar’][‘baz']【从左至右法则】
```

开启php-fpm状态页详解

php-fpm内建了个状态页，可以通过该状态页了解监控php-fpm的状态

配置方法：

php-fpm.conf开用php-fpm状态功能

```
# grep pm.status_path /usr/local/php/etc/php-fpm.conf
pm.status_path = /phpfpm_status
```

默认情况下为/status，当然也可以改成/phpfpm_status等，我这里是改成/phpfpm_status啦

nginx配置

在`默认主机`里面加上location或者你希望能访问到的主机里面。

```
server {
        listen 80;
        server_name 127.0.0.1;

        location /phpfpm_status {
                fastcgi_pass  unix:/tmp/php-cgi.sock;
                include fastcgi_params;
                fastcgi_param SCRIPT_FILENAME $fastcgi_script_name;
        }
}
```

重启nginx、php-fpm使配置生效

```
# /etc/init.d/nginx restart
```

```
# /etc/init.d/php-fpm restart
```

打开status页面

在浏览器打开 http://127.0.0.1/phpfpm_status

```
pool:                 www
process manager:      dynamic
start time:           24/Nov/2019:14:16:26 +0800
start since:          913
accepted conn:        44
listen queue:         0
max listen queue:     0
listen queue len:     0
idle processes:       9
active processes:     1
total processes:      10
max active processes: 1
max children reached: 0
slow requests:        0
```

php-fpm status详解

```
pool   – fpm池子名称，大多数为www
process manager   – 进程管理方式,值：static, dynamic or ondemand. dynamic
start time   – 启动日期,如果reload了php-fpm，时间会更新
start since   – 运行时长
accepted conn   – 当前池子接受的请求数
listen queue   – 请求等待队列，如果这个值不为0，那么要增加FPM的进程数量
max listen queue   – 请求等待队列最高的数量
listen queue len   – socket等待队列长度
idle processes   – 空闲进程数量
active processes   – 活跃进程数量
total processes   – 总进程数量
max active processes   – 最大的活跃进程数量（FPM启动开始算）
```

```
13 max children reached   - 大道进程最大数量限制的次数，如果这个数量不为0，那说明
   你的最大进程数量太小了，请改大一点。
14 slow requests   - 启用了php-fpm slow-log，缓慢请求的数量
```

注意

在nginx中可以开启Basic Auth登录认证，经常更换密码，这样就不会对外暴露php-fpm 信息；

若nginx配置加到其他域名server中访问不成功，就单独配置出来了。

附录：

php-fpm状态页可以通过带参数实现个性化，可以带参数json、xml、html并且前面三个参数可以分别和full做一个组合。

json格式:

http://127.0.0.1/phpfpm_status?json

xml格式:

http://127.0.0.1/phpfpm_status?xml

html格式:

http://127.0.0.1/phpfpm_status?html

full格式

http://127.0.0.1/phpfpm_status?full

组合形式： http://127.0.0.1/phpfpm_status?json&full

full格式详解

```
1 pid - 进程PID，可以单独kill这个进程。You can use this PID to kill a long
```

```
   running process.
state – 当前进程的状态 (Idle, Running, …)
start time – 进程启动的日期
start since – 当前进程运行时长
requests – 当前进程处理了多少个请求
request duration – 请求时长 (微妙)
request method – 请求方法 (GET, POST, …)
request URI – 请求URI
content length – 请求内容长度 (仅用于 POST)
user – 用户 (PHP_AUTH_USER) (or '-' 如果没设置)
script – PHP脚本 (or '-' if not set)
last request cpu – 最后一个请求CPU使用率。
last request memorythe – 上一个请求使用的内存
```

php 扩展安装方法

以fileinfo 扩展安装为例：

1、下载当前php版本源码

可通过 php.net/download 官网下载

2、解压下载的源码，并进入扩展目录。

```
cd ${php 源码的位置}/ext/fileinfo
```

3、执行 phpize

```
${php 安装目录}/bin/phpize
```

phpize 是什么?
php官方的说明：http://php.net/manual/en/install.pecl.phpize.php
phpize 是用来扩展 php 扩展模块的工具，通过 phpize 可以建立 php 的外挂模块，比如你想在原来编译好的 php 中加入 memcached 或者 ImageMagick 等扩展模块，均可以使用 phpize。

返回信息如下：

```
Configuring for:
PHP Api Version:         20151012
Zend Module Api No:      20151012
Zend Extension Api No:   320151012
```

4、编译 fileinfo

```
./configure --with-php-config=${php 安装目录}/bin/php-config

```

```
make && make install
```

编译成功大概输出如下：

```
Build complete.
Don't forget to run 'make test'.

Installing shared extensions:     /usr/local/php/lib/php/extensions/no-debug-non-zts-20151012/
```

其中，Installing shared extensions 的内容就是您的扩展存放位置。

也可通过如下命令得到：

```
${php 的安装位置}/bin/php-config --extension-dir
```

5、填写扩展配置：

复制粘贴，修改 extension 目录为您自己的 php 扩展存放目录即可。

修改 ${php 的安装位置}/etc/php.ini文件

查找：extension = 再最后一个extension= 后面添加上extension = "fileinfo.so"

但要注意extension-dir的目录配置，若与上面不一致可以用绝对路径

```
extension = {php 扩展存放目录}/fileinfo.so
```

6、 重启 php-fpm 进程

service php-fpm restart

phpinfo 中将会显示安装的 fileinfo 信息：

fileinfo

filcinfo support	enabled
version	1.0.5
libmagic	522

也可用 php –m 命令查看已安装扩展

注意：

如果要安装的扩展在php源码ext目录中没有，那么这个扩展需要到 http://pecl.php.net 搜索下载

下载扩展包，解压，进入扩展包目录，再从上述phpize那一步开始执行。在执行 ./configure 时不同的扩展包可能有不同的配置，要注意，查一下

Composer 遇见问题解决思路

Composer命令执行遇见问题时从以下几个方面查看、解决

1、使用调试模式执行

composer 命令增加 -vvv 参数可输出命令执行详细的信息，可通过查看详细信息查找问题所在

```
composer -vvv require laravel/framework
```

如果不能看出问题所在继续下面操作

2、建议先将Composer版本升级到最新

```
composer self-update
```

3、执行诊断命令

```
composer diagnose
```

诊断命令会给出修改建议，可以根据建议操作。

4、清除缓存

```
composer clear
```

5、若项目之前已通过其他源安装，则需要更新 composer.lock 文件，执行命令：

```
composer update --lock
```

之后重新执行出问题的命令，若还有问题，找度娘！

> 注意上面步骤不是要全部执行，是上面步骤无法解决才考虑下面步骤。

Composer 加速，镜像源修改

默认情况下执行 composer 各种命令是去国外的 composer 官方镜像源获取需要安装的具体软件信息，在正常情况下国内访问国外服务器的速度相对比较慢

修改镜像源

以阿里巴巴提供的 Composer 全量镜像为例：

镜像地址： https://mirrors.aliyun.com/composer/

配置只在当前项目生效

```
composer config repo.packagist composer https://mirrors.aliyun.com/composer/
```

取消当前项目配置

```
composer config --unset repos.packagist
```

配置全局生效

```
composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/
```

取消全局配置

执行之后，composer 会利用默认值（也就是官方源）重置源地址。

```
composer config -g --unset repos.packagist
```

第三方工具快速切换源CRM

CRM – Composer源管理工具

Composer Registry Manager 默认带了一个镜像列表,可以帮助你轻松、快速地切换到另外一个镜像。

GitHub地址: https://github.com/slince/composer-registry-manager

安装 crm

```
omposer global require slince/composer-registry-manager
```

使用方法

列出所有可使用的镜像

```
$ composer repo:ls

--- ------------- ------------------------------------------------ -
------------------------------
      composer      https://packagist.org
Europe, Canada and Singapore
      aliyun        https://mirrors.aliyun.com/composer
China
      tencent       https://mirrors.cloud.tencent.com/composer
China
  *   huawei        https://mirrors.huaweicloud.com/repository/php
China
      cnpkg         https://php.cnpkg.org
China
      sjtug         https://packagist.mirrors.sjtug.sjtu.edu.cn
China
      phpcomposer   https://packagist.phpcomposer.com
China
```

```
      kkame         https://packagist.kr
South Korea
      hiraku        https://packagist.jp
Japan
      webysther     https://packagist.com.br
Brazil
      solidworx     https://packagist.co.za
South Africa
      indra         https://packagist.phpindonesia.id
Indonesia
      varun         https://packagist.in
India
 --- ------------- ------------------------------------------------
  ------------------------------
```

标“*”表示当前正在使用的源;

你可以使用 --location xx 按地区过滤

```
$ composer repo:ls --location China
```

切换镜像

```
$ composer repo:use

Please select your favorite repository (defaults to composer) [compo
ser]:
  [0 ] composer
  [1 ] aliyun
  [2 ] tencent
  [3 ] huawei
  [4 ] cnpkg
  [5 ] sjtug
  [6 ] phpcomposer
  [7 ] kkame
```

```
  [8 ] hiraku
  [9 ] webysther
  [10] solidworx
  [11] indra
  [12] varun
>
```

你也可以直接追加镜像名称来跳过选择

```
$ composer repo:use aliyun
```

添加选项 `--current/-c` 为当前项目切换源，默认是修改全局的源。

重置命令

如果你想丢弃所有自定义的镜像源，你可以使用下面命令：

```
$ composer repo:reset
```

所有命令

执行下面命令查看

```
$ composer repo

  _____   _____         ___  ___
 /  ___| |  _  \       /   |/   |
 | |     | |_| |      / /|   /| |
 | |     |  _  /     / / |__/ | |
 | |___  | | \ \    / /        | |
 \_____| |_|  \_\ /_/         |_|

 Composer Repository Manager version 2.0.0

 Usage:
   command [options] [arguments]
```

```
14
15 Options:
16   -h, --help            Display this help message
17   -q, --quiet           Do not output any message
18   -V, --version         Display this application version
19       --ansi            Force ANSI output
20       --no-ansi         Disable ANSI output
21   -n, --no-interaction  Do not ask any interactive question
22   -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for n
   ormal output, 2 for more verbose output and 3 for debug
23
24 Available commands for the "repo" namespace:
25   repo:add     Creates a repository
26   repo:ls      List all available repositories
27   repo:remove  Remove a repository
28   repo:use     Change current repository
```

依赖管理Composer

人生苦短，我用 Composer

在 PHP 生态中，Composer 是最先进的依赖管理方案。我们推荐 PHP: The Right Way 中关于 依赖管理的完整章节。
如果你没有使用 Composer 来管理应用的依赖，最终（hopefully later but most likely sooner）会导致应用里某个依赖会严重过时，然后老旧版本中的漏洞会被利用于计算机犯罪。
重要： 开发软件时，时常记得保持依赖的更新。幸运地，这只需一行命令：

```
composer update
```

如果你正在使用某些专业的，需要使用 PHP 扩展（C 语言编写），那你不能使用 Composer 管理，而需要 PECL 。

lumen 多redis连接方法

在没有使用服务器集群的情况下，连接多个redis服务，可以传递服务器名到connection方法来获取指定Redis配置中定义的指定服务器。

```
$redis = Redis::connection('other');
```

配置方法

1、复制 vendor/laravel/lumen-framework/config/database.php 文件到 config/ 下，修改config/database.php文件redis配置，避免直接修改vendor下文件

```
'redis' => [

        'cluster' => env('REDIS_CLUSTER', false),

        'default' => [
            'host'     => env('REDIS_HOST', '127.0.0.1'),
            'port'     => env('REDIS_PORT', 6379),
            'database' => env('REDIS_DATABASE', 0),
            'password' => env('REDIS_PASSWORD', null),
        ],

        # 新增测试redis配置
        'test_redis' => [
            'host'     => env('TEST_REDIS_HOST', '127.0.0.1'),
            'port'     => env('TEST_REDIS_PORT', 6379),
            'database' => env('TEST_REDIS_DATABASE', 0),
            'password' => env('TEST_REDIS_PASSWORD', null),
        ],

    ],
```

2、在.env文件中添加test_redis配置

```
TEST_REDIS_HOST=127.0.0.1
TEST_REDIS_PORT=6379
TEST_REDIS_DATABASE=0
TEST_REDIS_PASSWORD=null
```

3、在 bootstrap/app.php 中引入 config/database.php配置文件

```
$app->configure('database');
```

4、使用实例

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redis;

class RedisController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }


    public function test(){
        $key = 'key';
        # 默认redis实例
        Redis::hgetall($key);
```

```
25
26          # 获取指定配置的Redis服务器实例
27          $redis = Redis::connection('test_redis');
28          $redis->hMset($key, $data);
29      }
30
31      //
32  }
```

lumen 多语言支持

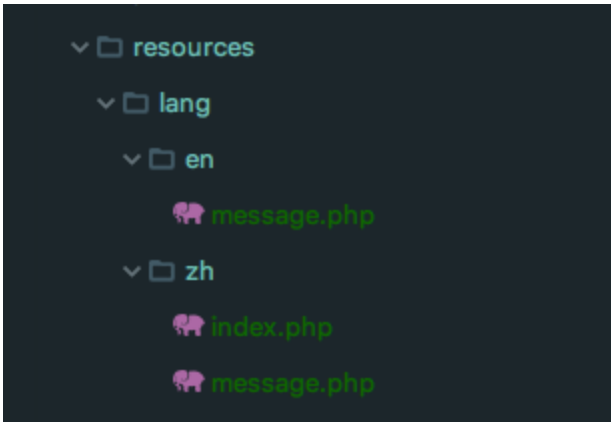
先增加一个中间件app/Http/Middleware/LangMiddleware.php

```php
<?php

namespace App\Http\Middleware;

use Closure;

class LangMiddleware
{
    /**
     * 多语言处理中间件
     * @param         $request
     * @param Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $host = $request->getHost();
        switch ($host){
            case 'zh.baidu.io':
                $lang = 'zh';
                break;
            case 'en.baidu.io':
                $lang = 'en';
                break;
            default:
                $lang = 'zh';
        }
        app('translator')->setLocale($lang);
        return $next($request);
    }
}
```

然后在bootstrap/app.php里注册为全局中间件

```php
<?php

$app->middleware([
    # App\Http\Middleware\CrossHttp::class,
    App\Http\Middleware\LangMiddleware::class,
    # App\Http\Middleware\ExampleMiddleware::class
]);
```

语言定义文件在 resource/lang目录下：



resources/lang/zh/message.php内容实例：

```php
<?php

return [
    'validation_failed' => '验证失败。',
];
```

代码里调用翻译：

```php
use Illuminate\Support\Facades\Lang;

$output = Lang::get('message.validation_failed');
```

laravel权限管理Entrust扩展包

项目地址

https://github.com/Zizaco/entrust

使用参考文章

https://laravelacademy.org/post/3755.html

https://laravelacademy.org/post/3761.html

使用时可以以路由名称作为操作标识，判断当前路由是否在权限表中，存在有权限，否则无权限

若不想创建操作和路由标记表可直接遍历项目路由，以路由名称作为操作说明

```
Route::get('/test', 'TestController@index')->name('操作描述');
```

遍历所有路由

```
use Illuminate\Routing\Router;

$routes = $router->getRoutes();
```

Laravel 最佳实践

这并非laravel官方强制要求的规范，而是我们在日常开发过程中遇到的一些容易忽视的优秀实现方式。

内容

避免直接从 `.env` 文件里获取数据

使用标准格式来存储日期，用访问器和修改器来修改日期格式

其他的好建议

单一职责原则

一个类和一个方法应该只有一个责任。

例如:

```
public function getFullNameAttribute()
{
    if (auth()->user() && auth()->user()->hasRole('client') && auth()->user()->isVerified()) {
        return 'Mr. ' . $this->first_name . ' ' . $this->middle_name . ' ' . $this->last_name;
    } else {
        return $this->first_name[0] . '. ' . $this->last_name;
    }
}
```

更优的写法:

```
public function getFullNameAttribute()
{
    return $this->isVerifiedClient() ? $this->getFullNameLong() : $this->getFullNameShort();
}

public function isVerifiedClient()
{
    return auth()->user() && auth()->user()->hasRole('client') && auth()->user()->isVerified();
}

```

```
public function getFullNameLong()
{
    return 'Mr. ' . $this->first_name . ' ' . $this->middle_name . ' ' . $this->last_name;
}

public function getFullNameShort()
{
    return $this->first_name[0] . '. ' . $this->last_name;
}
```

保持控制器的简洁

如果您使用的是查询生成器或原始SQL查询，请将所有与数据库相关的逻辑放入Eloquent模型或Repository类中。

例如:

```
public function index()
{
    $clients = Client::verified()
        ->with(['orders' => function ($q) {
            $q->where('created_at', '>', Carbon::today()->subWeek());
        }])
        ->get();

    return view('index', ['clients' => $clients]);
}
```

更优的写法:

```
public function index()
{
    return view('index', ['clients' => $this->client->getWithNewOrders()]);
```

```
}

class Client extends Model
{
    public function getWithNewOrders()
    {
        return $this->verified()
            ->with(['orders' => function ($q) {
                $q->where('created_at', '>', Carbon::today()->subWeek());
            }])
            ->get();
    }
}
```

使用自定义Request类来进行验证

把验证规则放到 Request 类中.

例子:

```
public function store(Request $request)
{
    $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
        'publish_at' => 'nullable|date',
    ]);

    ....
}
```

更优的写法:

```
public function store(PostRequest $request)
{
```

```
    ....
}

class PostRequest extends Request
{
    public function rules()
    {
        return [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
            'publish_at' => 'nullable|date',
        ];
    }
}
```

业务代码要放到服务层中

控制器必须遵循单一职责原则，因此最好将业务代码从控制器移动到服务层中。

例子:

```
public function store(Request $request)
{
    if ($request->hasFile('image')) {
        $request->file('image')->move(public_path('images') . 'tem
p');
    }

    ....
}
```

更优的写法:

```
public function store(Request $request)
{
    $this->articleService->handleUploadedImage($request->file('imag
```

```
e'));

    ....
}

class ArticleService
{
    public function handleUploadedImage($image)
    {
        if (!is_null($image)) {
            $image->move(public_path('images') . 'temp');
        }
    }
}
```

DRY原则 不要重复自己

尽可能重用代码，SRP可以帮助您避免重复造轮子。 此外尽量重复使用Blade模板，使用Eloquent的scopes 方法来实现代码。

例子:

```
public function getActive()
{
    return $this->where('verified', 1)->whereNotNull('deleted_at')->get();
}

public function getArticles()
{
    return $this->whereHas('user', function ($q) {
            $q->where('verified', 1)->whereNotNull('deleted_at');
        })->get();
}
```

更优的写法:

```php
public function scopeActive($q)
{
    return $q->where('verified', 1)->whereNotNull('deleted_at');
}

public function getActive()
{
    return $this->active()->get();
}

public function getArticles()
{
    return $this->whereHas('user', function ($q) {
            $q->active();
        })->get();
}
```

使用ORM而不是纯sql语句，使用集合而不是数组

使用Eloquent可以帮您编写可读和可维护的代码。 此外Eloquent还有非常优雅的内置工具，如软删除，事件，范围等。

例子:

```sql
SELECT *
FROM `articles`
WHERE EXISTS (SELECT *
              FROM `users`
              WHERE `articles`.`user_id` = `users`.`id`
              AND EXISTS (SELECT *
                          FROM `profiles`
                          WHERE `profiles`.`user_id` = `users`.`id`)
              AND `users`.`deleted_at` IS NULL)
AND `verified` = '1'
AND `active` = '1'
ORDER BY `created_at` DESC
```

更优的写法:

```
Article::has('user.profile')->verified()->latest()->get();
```

集中处理数据

例子:

```
$article = new Article;
$article->title = $request->title;
$article->content = $request->content;
$article->verified = $request->verified;
// Add category to article
$article->category_id = $category->id;
$article->save();
```

更优的写法:

```
$category->article()->create($request->validated());
```

不要在模板中查询，尽量使用惰性加载

例子 (对于100个用户，将执行101次DB查询):

```
@foreach (User::all() as $user)
    {{ $user->profile->name }}
@endforeach
```

更优的写法 (对于100个用户，使用以下写法只需执行2次DB查询):

```
$users = User::with('profile')->get();
```

```
2
3 ...
4
5 @foreach ($users as $user)
6     {{ $user->profile->name }}
7 @endforeach
```

注释你的代码，但是更优雅的做法是使用描述性的语言来编写你的代码

例子:

```
1 if (count((array) $builder->getQuery()->joins) > 0)
```

加上注释:

```
1 // 确定是否有任何连接
2 if (count((array) $builder->getQuery()->joins) > 0)
```

更优的写法:

```
1 if ($this->hasJoins())
```

不要把 JS 和 CSS 放到 Blade 模板中，也不要把任何 HTML 代码放到 PHP 代码里

例子:

```
1 let article = `{{ json_encode($article) }}`;
```

更好的写法:

```
<input id="article" type="hidden" value='@json($article)'>

Or

<button class="js-fav-article" data-article='@json($article)'>{{ $art
icle->name }}<button>
```

在Javascript文件中加上:

```
let article = $('#article').val();
```

当然最好的办法还是使用专业的PHP的JS包传输数据。

在代码中使用配置、语言包和常量，而不是使用硬编码

例子:

```
public function isNormal()
{
    return $article->type === 'normal';
}

return back()->with('message', 'Your article has been added!');
```

更优的写法:

```
public function isNormal()
{
    return $article->type === Article::TYPE_NORMAL;
}

return back()->with('message', __('app.article_added'));
```

使用社区认可的标准Laravel工具

强力推荐使用内置的Laravel功能和扩展包，而不是使用第三方的扩展包和工具。
如果你的项目被其他开发人员接手了，他们将不得不重新学习这些第三方工具的使用教程。
此外，当您使用第三方扩展包或工具时，你很难从Laravel社区获得什么帮助。 不要让你的客户为额外的问题付钱。

想要实现的功能	标准工具	第三方工具
权限	Policies	Entrust, Sentinel 或者其他扩展包
资源编译工具	Laravel Mix	Grunt, Gulp, 或者其他第三方包
开发环境	Homestead	Docker
部署	Laravel Forge	Deployer 或者其他解决方案
自动化测试	PHPUnit, Mockery	Phpspec
页面预览测试	Laravel Dusk	Codeception
DB操纵	Eloquent	SQL, Doctrine
模板	Blade	Twig
数据操纵	Laravel集合	数组
表单验证	Request classes	他第三方包,甚至在控制器中做验证
权限	Built-in	他第三方包或者你自己解决
API身份验证	Laravel Passport	第三方的JWT或者 OAuth 扩展包
创建 API	Built-in	Dingo API 或者类似的扩展包
创建数据库结构	Migrations	直接用 DB 语句创建
本土化	Built-in	第三方包
实时消息队列	Laravel Echo, Pusher	使用第三方包或者直接使用 WebSockets
创建测试数据	Seeder classes, Model Factories, Faker	手动创建测试数据
任务调度	Laravel Task Scheduler	脚本和第三方包
数据库	MySQL, PostgreSQL, SQLite,	MongoDB

	SQL Server	

遵循laravel命名约定

来源 PSR standards.

另外，遵循Laravel社区认可的命名约定：

对象	规则	更优的写法	应避免的写法
控制器	单数	ArticleController	~~ArticlesController~~
路由	复数	articles/1	~~article/1~~
路由命名	带点符号的蛇形命名	users.show_active	~~users.show-active~~, ~~show-active-users~~
模型	单数	User	~~Users~~
hasOne或belongsTo关系	单数	articleComment	~~articleComments~~, ~~article_comment~~
所有其他关系	复数	articleComments	~~articleComment~~, ~~article_comments~~
表单	复数	article_comments	~~article_comment~~, ~~articleComments~~
透视表	按字母顺序排列模型	article_user	~~user_article~~, ~~articles_users~~
数据表字段	使用蛇形并且不要带表名	meta_title	~~MetaTitle~~; ~~article_meta_title~~
模型参数	蛇形命名	$model->created_at	~~$model->createdAt~~
外键	带有_id后缀的单数模型名称	article_id	~~ArticleId~~, ~~id_article~~, ~~articles_id~~
主键	–	id	~~custom_id~~
迁移	–	2017_01_01_000000_create_articles_table	~~2017_01_01_000000_articles~~
方法	驼峰命名	getAll	~~get_all~~
资源控制器	table	store	~~saveArticle~~
测试类	驼峰命名	testGuestCannotSee	~~test_guest_cannot_s~~

		Article	~~ee_article~~
变量	驼峰命名	$articlesWithAuthor	~~$articles_with_author~~
集合	描述性的, 复数的	$activeUsers = User::active()->get()	~~$active, $data~~
对象	描述性的, 单数的	$activeUser = User::active()->first()	~~$users, $obj~~
配置和语言文件索引	蛇形命名	articles_enabled	~~ArticlesEnabled; articles-enabled~~
视图	短横线命名	show-filtered.blade.php	~~showFiltered.blade.php, show_filtered.blade.php~~
配置	蛇形命名	google_calendar.php	~~googleCalendar.php, google-calendar.php~~
内容 (interface)	形容词或名词	Authenticatable	~~AuthenticationInterface, IAuthentication~~
Trait	使用形容词	Notifiable	~~NotificationTrait~~

尽可能使用简短且可读性更好的语法

例子:

```
1 $request->session()->get('cart');
2 $request->input('name');
```

更优的写法:

```
1 session('cart');
2 $request->name;
```

更多示例:

常规写法	更优雅的写法

`Session::get('cart')`	`session('cart')`
`$request->session()->get('cart')`	`session('cart')`
`Session::put('cart', $data)`	`session(['cart' => $data])`
`$request->input('name'), Request::get('name')`	`$request->name, request('name')`
`return Redirect::back()`	`return back()`
`is_null($object->relation) ? null : $object->relation->id`	`optional($object->relation)->id`
`return view('index')->with('title', $title)->with('client', $client)`	`return view('index', compact('title', 'client'))`
`$request->has('value') ? $request->value : 'default';`	`$request->get('value', 'default')`
`Carbon::now(), Carbon::today()`	`now(), today()`
`App::make('Class')`	`app('Class')`
`->where('column', '=', 1)`	`->where('column', 1)`
`->orderBy('created_at', 'desc')`	`->latest()`
`->orderBy('age', 'desc')`	`->latest('age')`
`->orderBy('created_at', 'asc')`	`->oldest()`
`->select('id', 'name')->get()`	`->get(['id', 'name'])`
`->first()->name`	`->value('name')`

使用IOC容器来创建实例 而不是直接new一个实例

创建新的类会让类之间的更加耦合，使得测试越发复杂。请改用IoC容器或注入来实现。

例子:

```
1 $user = new User;
2 $user->create($request->validated());
```

更优的写法:

```
public function __construct(User $user)
{
    $this->user = $user;
}

....

$this->user->create($request->validated());
```

避免直接从 `.env` 文件里获取数据

将数据传递给配置文件，然后使用`config()`帮助函数来调用数据

例子:

```
$apiKey = env('API_KEY');
```

更优的写法:

```
// config/api.php
'key' => env('API_KEY'),

// Use the data
$apiKey = config('api.key');
```

使用标准格式来存储日期，用访问器和修改器来修改日期格式

例子:

```
{{ Carbon::createFromFormat('Y-d-m H-i', $object->ordered_at)->toDate
String() }}
{{ Carbon::createFromFormat('Y-d-m H-i', $object->ordered_at)->format
```

```
('m-d') }}
```

更优的写法:

```
// Model
protected $dates = ['ordered_at', 'created_at', 'updated_at'];
public function getSomeDateAttribute($date)
{
    return $date->format('m-d');
}

// View
{{ $object->ordered_at->toDateString() }}
{{ $object->ordered_at->some_date }}
```

其他的一些好建议

永远不要在路由文件中放任何的逻辑代码。

尽量不要在Blade模板中写原始 PHP 代码。

转自： https://github.com/alexeymezenin/laravel-best-practices/blob/master/chinese.md

Laravel 请求到达控制器的过程

入口

Laravel5.8 入口文件为 public/index.php

```
    $kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

    $response = $kernel->handle(
        $request = Illuminate\Http\Request::capture()
    );

    $response->send();

    $kernel->terminate($request, $response);
```

创建了一个 Kernel 对象，调用 handler 处理请求，获取返回结果。将返回结果输出到客户端，处理 terminate 操作。

Kernel 中如何处理请求

容器里绑定的是 App\Http\Kernel, 继承于 Illuminate\Foundation\Http\Kernel。

```
    /**
     * Handle an incoming HTTP request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function handle($request)
    {
        try {
            $request->enableHttpMethodParameterOverride();

```

```
            $response = $this->sendRequestThroughRouter($request);
        } catch (Exception $e) {
            $this->reportException($e);

            $response = $this->renderException($request, $e);
        } catch (Throwable $e) {
            $this->reportException($e = new FatalThrowableError
($e));

            $response = $this->renderException($request, $e);
        }

        $this->app['events']->dispatch(
            new Events\RequestHandled($request, $response)
        );

        return $response;
    }
```

Kernel 中调用 sendRequestThroughRouter 方法，将请求传递到路由处理当中。

```
    /**
     * Send the given request through the middleware / router.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    protected function sendRequestThroughRouter($request)
    {
        $this->app->instance('request', $request);

        Facade::clearResolvedInstance('request');

        $this->bootstrap();

        return (new Pipeline($this->app))
                    ->send($request)
                    ->through($this->app->shouldSkipMiddleware() ?
```

```
    [] : $this->middleware)
                    ->then($this->dispatchToRouter());
    }
```

在 sendRequestThroughRouter 当中，在 app 中绑定了 request 实例，并解绑掉其他 request 实例对象。这样在程序其他地方都能通过 app()->make('request') 获取到 request 实例对象。

调用 bootstrap 方法，加载引导类。

创建一个 Pipeline 对象，将路由调度与中间件放入调用链当中。所有 request 先经过全局的中间件，然后在通过路由分发。

```
    /**
     * Get the route dispatcher callback.
     *
     * @return \Closure
     */
    protected function dispatchToRouter()
    {
        return function ($request) {
            $this->app->instance('request', $request);

            return $this->router->dispatch($request);
        };
    }
```

因为 Piepline 调用链都是一个个的回调方法，所以在 dispatchToRouter 返回了一个匿名回调函数。使用 Kernel 的 route 属性进行调度。

Kernel 的 route 是一个 Illuminate\Routing\Router 对象。

路由调度

```
    //Illuminate\Routing\Router

    /**
```

```
     * Dispatch the request to the application.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response|\Illuminate\Http\JsonRespon
se
     */
    public function dispatch(Request $request)
    {
        $this->currentRequest = $request;

        return $this->dispatchToRoute($request);
    }

    /**
     * Dispatch the request to a route and return the response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response|\Illuminate\Http\JsonRespon
se
     */
    public function dispatchToRoute(Request $request)
    {
        return $this->runRoute($request, $this->findRoute($reques
t));
    }

    /**
     * Return the response for the given route.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Illuminate\Routing\Route  $route
     * @return \Illuminate\Http\Response|\Illuminate\Http\JsonRespon
se
     */
    protected function runRoute(Request $request, Route $route)
    {
        $request->setRouteResolver(function () use ($route) {
            return $route;
        });

```

```
40          $this->events->dispatch(new Events\RouteMatched($route, $req
    uest));
41
42          return $this->prepareResponse($request,
43              $this->runRouteWithinStack($route, $request)
44          );
45      }
```

从上面的方法可以看出，最终通过 findRoute 查找当前匹配的路由对象，并调用 runRoute 处理请求返回结果。

怎么找到路由的

```
1       //Illuminate\Routing\Router
2
3       /**
4        * Find the route matching a given request.
5        *
6        * @param  \Illuminate\Http\Request  $request
7        * @return \Illuminate\Routing\Route
8        */
9       protected function findRoute($request)
10      {
11          $this->current = $route = $this->routes->match($request);
12
13          $this->container->instance(Route::class, $route);
14
15          return $route;
16      }
```

对路由的匹配，是通过 routes 这个路由 Collections 去匹配的。

```
1       //Illuminate\Routing\RouteCollection
2
3       /**
4        * Find the first route matching a given request.
```

```
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Routing\Route
     *
     * @throws \Symfony\Component\HttpKernel\Exception\NotFoundHttpE
xception
     */
    public function match(Request $request)
    {
        $routes = $this->get($request->getMethod());

        // First, we will see if we can find a matching route for th
is current request
        // method. If we can, great, we can just return it so that i
t can be called
        // by the consumer. Otherwise we will check for routes with
 another verb.
        $route = $this->matchAgainstRoutes($routes, $request);

        if (! is_null($route)) {
            return $route->bind($request);
        }

        // If no route was found we will now check if a matching rou
te is specified by
        // another HTTP verb. If it is we will need to throw a Metho
dNotAllowed and
        // inform the user agent of which HTTP verb it should use fo
r this route.
        $others = $this->checkForAlternateVerbs($request);

        if (count($others) > 0) {
            return $this->getRouteForMethods($request, $others);
        }

        throw new NotFoundHttpException;
    }

    /**
     * Determine if a route in the array matches the request.
```

```
38      *
39      * @param  array  $routes
40      * @param  \Illuminate\Http\Request  $request
41      * @param  bool  $includingMethod
42      * @return \Illuminate\Routing\Route|null
43      */
44     protected function matchAgainstRoutes(array $routes, $request,
    $includingMethod = true)
45     {
46         [$fallbacks, $routes] = collect($routes)->partition(function
   ($route) {
47             return $route->isFallback;
48         });
49
50         return $routes->merge($fallbacks)->first(function ($value) u
   se ($request, $includingMethod) {
51             return $value->matches($request, $includingMethod);
52         });
53     }
```

先通过请求的方法获取当前方法下可用的路由集合，在从这些集合中去遍历获取第一个匹配的路由。集合中每个 item 是一个 Illuminate\Routing\Router 对象。因此最终判断路由与请求是否匹配调用的是 Illuminate\Routing\Router 中的 matches 方法。

```
1
2     //Illuminate\Routing\Router
3
4     /**
5      * Determine if the route matches given request.
6      *
7      * @param  \Illuminate\Http\Request  $request
8      * @param  bool  $includingMethod
9      * @return bool
10     */
11    public function matches(Request $request, $includingMethod = tru
   e)
12    {
13        $this->compileRoute();
```

```
14
15          foreach ($this->getValidators() as $validator) {
16              if (! $includingMethod && $validator instanceof MethodVa
    lidator) {
17                  continue;
18              }
19
20              if (! $validator->matches($this, $request)) {
21                  return false;
22              }
23          }
24
25          return true;
26      }
27
28      /**
29           * Get the route validators for the instance.
30           *
31           * @return array
32           */
33          public static function getValidators()
34          {
35              if (isset(static::$validators)) {
36                  return static::$validators;
37              }
38
39              // To match the route, we will use a chain of responsibi
    lity pattern with the
40              // validator implementations. We will spin through each
     one making sure it
41              // passes and then we will know if the route as a whole
     matches request.
42              return static::$validators = [
43                  new UriValidator, new MethodValidator,
44                  new SchemeValidator, new HostValidator,
45              ];
46          }
```

在 Illuminate\Routing\Router 提供了四个默认的验证器，当四个验证器通过的时候才会匹配成功。四个验证器分别是 UriValidator 验证访问路径，MethodValidator 验证请求方法，SchemeValidator 验证访问协议，HostValidator 验证域名。其中对 uri 的验证内部是使用正则表达式验证。

路由调度怎么处理请求

```
    //Illuminate\Routing\Router

    /**
     * Run the given route within a Stack "onion" instance.
     *
     * @param  \Illuminate\Routing\Route  $route
     * @param  \Illuminate\Http\Request  $request
     * @return mixed
     */
    protected function runRouteWithinStack(Route $route, Request $request)
    {
        $shouldSkipMiddleware = $this->container->bound('middleware.disable') &&
                                $this->container->make('middleware.disable') === true;

        $middleware = $shouldSkipMiddleware ? [] : $this->gatherRouteMiddleware($route);

        return (new Pipeline($this->container))
                        ->send($request)
                        ->through($middleware)
                        ->then(function ($request) use ($route) {
                            return $this->prepareResponse(
                                $request, $route->run()
                            );
                        });
    }

    /**
     * Run the route action and return the response.
```

```
     *
     * @return mixed
     */
    public function run()
    {
        $this->container = $this->container ?: new Container;

        try {
            if ($this->isControllerAction()) {
                return $this->runController();
            }

            return $this->runCallable();
        } catch (HttpResponseException $e) {
            return $e->getResponse();
        }
    }
```

路由对请求的处理也是返回一个 Pipeline, 先将请求通过中间件，然后在执行路由的 run 方法。在 run 方法里面判断当前是执行控制器方法还是回调方法，根据不同类型分开执行。

怎么执行

```
    /**
     * Checks whether the route's action is a controller.
     *
     * @return bool
     */
    protected function isControllerAction()
    {
        return is_string($this->action['uses']);
    }

    /**
     * Run the route action and return the response.
     *
     * @return mixed
```

```
15      */
16     protected function runCallable()
17     {
18         $callable = $this->action['uses'];
19 
20         return $callable(...array_values($this->resolveMethodDependencies(
21             $this->parametersWithoutNulls(), new ReflectionFunction($this->action['uses'])
22         )));
23     }
24 
25     /**
26      * Run the route action and return the response.
27      *
28      * @return mixed
29      *
30      * @throws \Symfony\Component\HttpKernel\Exception\NotFoundHttpException
31      */
32     protected function runController()
33     {
34         return $this->controllerDispatcher()->dispatch(
35             $this, $this->getController(), $this->getControllerMethod()
36         );
37     }
```

通过当前路由的 action 配置判断是否是控制器或者回调方法。从代码中可以看到，其实就是我们路由配置中的第二个参数对应到 action['user']。当我们第二参数是一个字符串的时候则认为是控制器方法，将请求转发到控制器里去处理。否则执行回调函数处理。

到这里，我们的请求就真的到达了我们的控制器的方法中，开始执行我们写的代码了。

lumen框架解决非简单请求 cors 跨域问题

Lumen在做前后端分离项目时，在浏览器中访问，若前后端域名不一致会导致跨域问题，简单跨域好解决。若发送的是非简单跨域，此时浏览器会先发送option请求进行预检，预检通过才发送真正的请求。此时服务端要实现option请求的接收。服务端代码实现如下：

添加如下中间件：

```php
<?php

namespace App\Http\Middleware;

use Closure;

class CrossHttp
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if($request->getMethod() == "OPTIONS") {
            $allowOrigin = [
                'http://192.168.1.47',
                'http://localhost',
            ];
            $Origin = $request->header("Origin");
            if(in_array($Origin, $allowOrigin)){
                return response()->json('ok', 200, [
                    # 下面参数视request中header而定
                    'Access-Control-Allow-Origin' => $Origin,
                    'Access-Control-Allow-Headers' => 'x-token',
                    'Access-Control-Allow-Methods' => 'GET,POST,OPTIONS']);
```

```
            } else {
                return response()->json('fail', 405);
            }
        }

        $response = $next($request);
        $response->header('Access-Control-Allow-Origin', '*');
        return $response;
    }
}
```

在 boostrap/app.php 里注册一下全局中间件即可完成

```
$app->middleware([
    \App\Http\Middleware\CrossHttp::class,
]);
```

php定位和分析执行效率方法

```
1. 代码脚本里打点计时；

2. 查看 php-fpm.conf 中配置的慢日志；

3. xdebug统计函数执行次数和具体时间进行分析。最好使用工具winCacheGrind分析；

4. 网页上用strace跟踪相关进程的具体系统调用。
```

array_search 和 in_array 函数效率问题

问题

array_search 查找数组中的元素的 key 时，效率随着数组变大，耗时增加。特别是多次查找大数组时，非常耗时。在函数 in_array 也有这个问题。

解决办法

采用 array_flip (array_flip — 交换数组中的键和值) 翻转后，用 isset 代替 in_array 函数，用 $array[key] 替代 array_search, 这样能解决大数组超时耗时问题.

注意：

1、这种优化只适用于无重复数据的大数组且多次查找时有效，原因见下面。
2、array_search、in_array、array_flip 原理都是数组遍历。array_search、in_array是发现便终止，array_flip遍历整个数组
3、因array_flip遍历整个数组，相比array_search和in_array，array_flip更耗时，只有多次使用array_search和in_array时才能显示出array_flip效果
4、isset 走的 hash 表,查找时间复杂度为 O(1),更高效。
5、使用场景不多，基本用处不大 只是提供一种思路

实例：

```
<?php
$array = array();

for ($i=0; $i<200000; $i++){
    ##随机字符串
    $array[$i] = get_rand().$i;
}

$str = $array[150000];
$time1 = microtime(true);
```

```
array_search($str, $array);
$time2 = microtime(true);
echo '原始方法:'.($time2-$time1)."\n";

$time3 =  microtime(true);
$new_array = array_flip($array);
isset($new_array[$str]);
$time4 = microtime(true);
echo '新方法:'.($time4-$time3);


$array = array();

for ($i=0; $i<200000; $i++){
    ##随机字符串
    $array[$i] = get_rand().$i;
}

$str = $array[199999];
$time1 = microtime(true);
for ($i=0; $i<5000; $i++){
    array_search($str, $array);
}
$time2 = microtime(true);

echo '原始方法:'.($time2-$time1)."\n";

$time3 =  microtime(true);
$new_array = array_flip($array);
for ($i=0; $i<5000; $i++){
    isset($new_array[$str]);
}
$time4 = microtime(true);

echo '新方法:'.($time4-$time3);
```

转自： https://learnku.com/laravel/t/34826

phpredis 手册

目录（点击下面链接查看详情）

转自GitHub，地址： https://github.com/phpredis/phpredis

PHPMailer 在本地发送成功 阿里云上发送失败原因

阿里云默认不允许访问SMTP 25端口，可申请解封25端口或换ssl加密方式连接和端口

已腾讯企业邮箱为例：

```
    $mail = new PHPMailer();
$mail->IsSMTP();
# $mail->Host = 'smtp.exmail.qq.com';  # 本地可直接使用
$mail->Host = 'ssl://smtp.exmail.qq.com'; # 阿里云线上使用ssl加密方式
$mail->Port = 465; # ssl方式 用465端口
$mail->SMTPAuth = true;            // 打开SMTP认证
$mail->Username = 'xxx@xxx.com';
$mail->Password = 'yyyyyyy';
$mail->From = 'xxx@xxx.com';
$mail->FromName = iconv('utf-8', 'GBK', "xxx");  // 发件人

$mail->CharSet = 'GB2312';
$mail->Encoding = "base64";
$mail->IsHTML(true);
// 邮件主题
$mail->Subject = iconv('utf-8', 'GBK', $subject);
// 邮件内容
$mail->Body = iconv('utf-8', 'GBK', $message);
$mail->AltBody = "text/html";
if ($mail->Send()) {
    exit('success');
} else {
    exit($mail->ErrorInfo);
}
```

php获取用户和服务器ip及其地理位置详解

浏览器访问获取用户ip：

```php
/**
 * php获取用户真实 IP
 * 注意这种方式只适用于浏览器访问时
 */
function getIP()
{
    if (isset($_SERVER)){
        if (isset($_SERVER["HTTP_X_FORWARDED_FOR"])){
            $realip = $_SERVER["HTTP_X_FORWARDED_FOR"];
        } else if (isset($_SERVER["HTTP_CLIENT_IP"])) {
            $realip = $_SERVER["HTTP_CLIENT_IP"];
        } else {
            $realip = $_SERVER["REMOTE_ADDR"];
        }
    } else {
        if (getenv("HTTP_X_FORWARDED_FOR")){
            $realip = getenv("HTTP_X_FORWARDED_FOR");
        } else if (getenv("HTTP_CLIENT_IP")) {
            $realip = getenv("HTTP_CLIENT_IP");
        } else {
            $realip = getenv("REMOTE_ADDR");
        }
    }
    return $realip;
}
```

注意：

1、以上方式只适用于用浏览器访问后台服务时可用

2、以浏览器访问和在后台直接执行php脚本所生成的 $_SERVER 变量是不同的

后台脚本执行获取服务器ip：

```php
<?php
/**
*方法一：使用 gethostbyname() 方法,此方法获取的是内网ip
*/
$realip =  gethostbyname(gethostname());
print_r($realip);

echo "\n";

/**
*方法二：php执行linux系统命令 ifconfig
* 利用正则表达式获取 ip
*/

$output = shell_exec('ifconfig');
preg_match("/\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}/",$output,$realip );
print_r($realip [0]);

/**
*方法三：php执行linux系统命令 ifconfig ，此方法获取外网ip
* 利用grep获取 ip
*/
$shell = "/sbin/ifconfig | grep -oP '(?<=addr:).*(?=\s+B)' | sed '1
d'";
$output = shell_exec($shell);
$shell = "/sbin/ifconfig | grep -oP '(?<=addr:).*(?=\s+B)'";
$output = exec($shell);
return $output;

?>
```

注意：

1、gethostname() 获取的是 eth0 的ip，虚拟机下linux没有eth0项，所以当在虚拟机下执行该方法时返回 127.0.0.1

其他方法解释：

```
gethostbyname(string $hostname) : 获取对应主机名的一个ipv4地址
```

```
2
3 gethostbyaddr ( string $ip_address ) : 获取指定的IP地址对应的主机名
4
5 gethostbynamel ( string $hostname ) : 获取对应主机名的一系列所以ipv4地址
```

php获取ip所属城市：

```
1 /**
2  * php获取 IP  地理位置
3  * 淘宝IP接口
4  * @Return: array
5  */
6 function getCity($ip = '')
7 {
8     if($ip == ''){
9         $url = "http://int.dpool.sina.com.cn/iplookup/iplookup.php?f
  ormat=json";
10         $ip=json_decode(file_get_contents($url),true);
11         $data = $ip;
12     }else{
13         $url="http://ip.taobao.com/service/getIpInfo.php?ip=".$ip;
14         $ip=json_decode(file_get_contents($url));
15         if((string)$ip->code=='1'){
16             return false;
17         }
18         $data = (array)$ip->data;
19     }
20
21     return $data['city'];
22 }
```

urldecode & rawurldecode 说明

区别：

urlencode把空格编码为 '+'， rawurlencode()把空格编码为 '%20'

urldecode() 会把 '+' 破解为空格，rawurldecode() 不会

注意

因为'+' 号是 base64的编码字符，当urldecode与base64_decode配合使用时，要用rawurldecode()

推荐在PHP中使用用rawurlencode。弃用urlencode;
大部分使用场景下都适合使用rawurlencode()

url中base64之后的参数 一定要rawurlencode, 因为base64之后的字符串会包含"+" "/" 等特殊字符(此时base64可以替换为urlbase64，也可以解决这个问题)

超全局变量 $*GET 和 $*REQUEST 已经被解码了。对 $*GET 或 $*REQUEST 里的元素使用 urldecode() 将会导致不可预计和危险的结果，即接受urlencode转义的请求时不需要urldecode转义。其他的如$_POST等请求则需要自己转义

PHP在linux上执行系统命令

方法一：用PHP提供的专门函数（四个）：
1）exec():

```
string exec ( string $command [, array &$output [, int &$return_var ]
)
```

说明: exec执行系统外部命令时不会输出结果，而是返回结果的最后一行。如果想得到结果，可以使用第二个参数，让其输出到指定的数组。此数组一个记录代表输出的一行。即如果输出结果有20行，则这个数组就有20条记录，所以如果需要反复输出调用不同系统外部命令的结果，最好在输出每一条系统外部命令结果时清空这个数组unset($output)，以防混乱。第三个参数用来取得命令执行的状态码，通常**执行成功都是返回0**。

```
<?php
    // 输出运行中的 php/httpd 进程的创建者用户名
    //  （在可以执行 "whoami" 命令的系统上）
    echo exec('whoami');
?>
```

2）system():

```
string system ( string $command [, int &$return_var ] )
```

说明: system和exec的区别在于，system在执行系统外部命令时，它执行给定的命令，输出和返回结果。
成功
则返回命令输出的最后一行， 失败则返回 FALSE。第二个参数是可选的，用来得到命令执行后的状态码。

```
<?php
    $res = system("pwd",$result);
    print $result;//输出命令的结果状态码
```

```
    print $res;//输出命令输出的最后一行

?>
```

关于第二个参数结果状态码的简单介绍：
如果**返回0是运行成功**，
在Bash中，当错误发生在致命信号时，bash会返回128+signal number做为返回值。
如果找不到命令，将会返回127。
如果命令找到了，但该命令是不可执行的，将返回126。
除此以外，Bash本身会返回最後一个指令的返回值。
若是执行中发生错误，将会返回一个非零的值。
Fatal Signal : 128 + signo
Can't not find command : 127
Can't not execute : 126
Shell script successfully executed : return the last command exit status
Fatal during execution : return non-zero

3）passthru():

```
void passthru ( string $command [, int &$return_var ] )
```

说明: passthru与system的区别，passthru直接将结果输出到游览器，不返回任何值，且其可以输出二进制，比如图像数据。第二个参数可选，是状态码。

```
同 exec() 函数类似， passthru() 函数 也是用来执行外部命令（command）的。 当所执行的 Unix 命令输出二进制数据， 并且需要直接传送到浏览器的时候， 需要用此函数来替代 exec() 或 system() 函数。 常用来执行诸如 pbmplus 之类的可以直接输出图像流的命令。 通过设置 Content-type 为 image/gif， 然后调用 pbmplus 程序输出 gif 文件， 就可以从 PHP 脚本中直接输出图像到浏览器。
```

```
<?php
    header("Content-type:image/gif");
    passthru("/usr/bin/ppm2tiff  /usr/share/tk8.4/demos/images/teapot.ppm");
?>
```

4）shell_exec():

```
string shell_exec ( string $cmd )
```

说明: 直接执行命令$cmd，将完整的命令输出以字符串的方式返回。如果执行过程中发生错误或者进程不产生
输出，则返回 NULL。所以，使用本函数无法通过返回值检测进程是否成功执行。 如果需要检查进程执行的退
出码，请使用 exec() 函数。

```
<?php
    $output = shell_exec('ls -lart');
    echo "
<pre>$output</pre>";
?>
```

方法二：反撇号：
原型: 反撇号`（和~在同一个键）执行系统外部命令，相当于 shell_exec
说明: 在使用这种方法执行系统外部命令时，要确保shell_exec函数可用，否则是无法使用这种反撇号执行系统外部命令的。

```
<?php
    echo `dir`;
?>
```

用户自定义输入命令转义：
若命令需要用户输入，此时为了安全应该使用以下方法对用户输入命令或参数进行转义

1、shell 元字符转义

```
string escapeshellcmd ( string $command )

参数说明:
command    要转义的命令。
```

```
返回          转义后的字符串。
```

除去了字符串中的特殊符号，可以防止使用者耍花招来破解该服务器系统。
escapeshellcmd() 对字符串中可能会欺骗 shell 命令执行任意命令的字符进行转义。 此函数保证用户输入的数据在传送到 exec() 或 system() 函数，或者 执行操作符 之前进行转义。

反斜线（\）会在以下字符之前插入： #&;`|*?~<>^()[]{}$, \x0A 和 \xFF。 ' 和 " 仅在不配对儿的时候被转义。 在 Windows 平台上，所有这些字符以及 % 都会被空格代替。

实例：

```php
<?php
    // 我们故意允许任意数量的参数
    $command = './configure '.$_POST['configure_options'];

    $escaped_command = escapeshellcmd($command);

    system($escaped_command);
?>
```

escapeshellcmd() 应被用在完整的命令字符串上。 即使如此，攻击者还是可以传入任意数量的参数。 请使用 escapeshellarg() 函数 对单个参数进行转义。

2、shell 参数转义

```
string escapeshellarg ( string $arg )

arg ：需要被转码的参数。

返回值：转换之后字符串。
```

escapeshellarg() 将给字符串增加一个单引号并且能引用或者转码任何已经存在的单引号，这样以确保能够直接将一个字符串传入 shell 函数，并且还是确保安全的。对于用户输入的部分参数就应该使用这个函数。shell 函数包含 exec(), system() 执行运算符 。
实例：

```php
<?php
    system('ls '.escapeshellarg($dir));
?>
```

面向对象特性与设计原则

三大特性是：封装、继承、多态

所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

封装是面向对象的特征之一，是对象和类概念的主要特性。 简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

所谓继承是指可以让某个类型的对象获得另一个类型的对象的属性的方法,它支持按级分类的概念。

继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。

多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

五大设计原则

单一职责原则 (The Single Responsibility Principle)

一个类的功能要单一，不能包罗万象；
修改某个类的理由应该只有一个，如果超过一个，说明类承担不止一个职责，要视情况拆分。

开放封闭原则 (The Open Closed Principle)

软件实体应该对扩展开放，对修改封闭。一般不要直接修改类库源码（即使你有源代码），通过继承等方式扩展。

里氏替代原则（The Liskov Substitution Principle）

当一个子类的实例能够被替换成任何超类的实例时，它们之间才是真正的 is-a 关系。
即子类应当可以替换父类并出现在父类能够出现的任何地方。

依赖倒置原则（The Dependency Inversion Principle）

高层模块不应该依赖于底层模块，二者都应该依赖于抽象。换句话说，依赖于抽象，不要依赖于具体实现。比方说，你不会把电器电源线焊死在室内电源接口处，而是用标准的插头插在标准的插座（抽象）上。

接口分离原则（The Interface Segregation Principle）

模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来

不要强迫用户去依赖它们不使用的接口。换句话说，使用多个专门的接口比使用单一的大而全接口要好。

php运算符优先级

https://www.php.net/manual/zh/language.operators.precedence.php

生成二维码

qr-code项目

github 地址为：https://github.com/endroid/qr-code

可设置参数：
`setSize` – 二维码大小 px
`setWriterByName` – 写入文件的后缀名
`setMargin` – 二维码内容相对于整张图片的外边距
`setEncoding` – 编码类型
`setErrorCorrectionLevel` – 容错等级，分为L、M、Q、H四级
`setForegroundColor` – 前景色
`setBackgroundColor` – 背景色
`setLabel` – 二维码标签
`setLogoPath` – 二维码logo路径
`setLogoWidth` – 二维码logo大小 px

使用说明： https://juejin.im/entry/5a7bc1976fb9a0634a390122?utm_medium=be&utm_source=weixinqun

时间格式判断

```php
# 判断所给时间数据格式是否为`m-d`的形式
# 注意这种方式只能判断当年的，若判断往年的还得考虑平、闰年的情况
$date = '02-03';
if (date('m-d', strtotime(date('Y') . '-' . $date)) === $date) {
    echo '是';
} else {
    echo '否';
}
```

反斜杠过滤

```
# 过滤小于等于两个反斜杠
$str = stripslashes($str);

#过滤不限数量反斜杠
$str = stripslashes(trim(implode("", explode("\\", $str))));
```

语雀知识库备份

使用到的项目yuque2book： https://github.com/yuque-helper/yuque2book/wiki/%E8%AF%AD%E9%9B%80%E6%96%87%E6%A1%A3%E5%B7%A5%E5%85%B7

参考文档：https://www.yuque.com/yuque/developer/api

代码：

```php
<?php

namespace App\Console\Commands\Shell;

use Illuminate\Console\Command;

/**
 * 语雀知识库导出脚本
 * 使用说明：
 * 改脚本依赖于语雀文档工具 yuque2book
 * 使用说明见 https://github.com/yuque-helper/yuque2book/wiki/%E8%AF%AD%E9%9B%80%E6%96%87%E6%A1%A3%E5%B7%A5%E5%85%B7
 * Class YuqueBackUp
 * @package App\Console\Commands\Shell
 */
class YuqueBackUp extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'Shell:YuqueBackUp';

    /**
     * The console command description.
     *
```

```
27      * @var string
28      */
29     protected $description = '语雀备份';
30 
31     private $token = '{}';
32 
33     private $savePath = '{}';
34 
35     private $userId = '{}';
36 
37     /**
38      * Create a new command instance.
39      *
40      * @return void
41      */
42     public function __construct()
43     {
44         parent::__construct();
45     }
46 
47     /**
48      * Execute the console command.
49      *
50      * @return mixed
51      */
52     public function handle()
53     {
54         # 判断备份目录是否存在
55         if (!is_dir($this->savePath)) {
56             mkdir($this->savePath, 0777, true);
57         }
58         $repos = $this->getRepoList();
59         $repos = json_decode($repos, true);
60         $slugList = array_map(function ($v) {
61             return $v['slug'];
62         }, $repos['data']);
63         foreach ($slugList as $k => $slug) {
64             if($slug == 'pw0zfo'){
65                 continue;
66             }
```

```
67              $shell = "cd {$this->savePath} && rm -rf $this->userId" . '_' . "$slug && yuque2book -t {$this->token} https://www.yuque.com/{$this->userId}/{$slug} -l";
68              echo $k + 1 . "\t" . $shell . PHP_EOL;
69              var_dump(shell_exec($shell));
70          }
71          # 备份到git
72          $shell = "cd {$this->savePath} && /usr/bin/git add . && /usr/bin/git commit -m 'yuque bak at'$(date +\"\%Y\%m\%d\") &&  /usr/bin/git push";
73          echo $shell . PHP_EOL;
74          var_dump(shell_exec($shell));
75      }
76
77      /**
78       * 获取知识库列表
79       * @date 2020-04-20 19:27
80       * @return bool|string
81       */
82      private function getRepoList()
83      {
84          $curl = curl_init();
85
86          curl_setopt_array($curl, array(
87              CURLOPT_URL => "https://www.yuque.com/api/v2/users/{$this->userId}/repos",
88              CURLOPT_RETURNTRANSFER => true,
89              CURLOPT_ENCODING => "",
90              CURLOPT_MAXREDIRS => 10,
91              CURLOPT_TIMEOUT => 0,
92              CURLOPT_FOLLOWLOCATION => true,
93              CURLOPT_HTTP_VERSION => CURL_HTTP_VERSION_1_1,
94              CURLOPT_CUSTOMREQUEST => "GET",
95              CURLOPT_HTTPHEADER => array(
96                  "Content-Type: application/json",
97                  "X-Auth-Token: " . $this->token,
98                  "User-Agent: {}"
99              ),
100         ));
101
```

```
        $response = curl_exec($curl);

        curl_close($curl);
        return $response;
    }
}
```

Go 通道（channel）

通道（channel）

是用来传递数据的一个数据结构。

通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。

```
ch <- v    // 把 v 发送到通道 ch
v := <-ch  // 从 ch 接收数据
           // 并把值赋给 v
```

声明一个通道很简单，我们使用chan关键字即可，通道在使用前必须先创建：

```
ch := make(chan int)
```

注意：默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须又接收端相应的接收数据。

以下实例通过两个 goroutine 来计算数字之和，在 goroutine 完成计算后，它会计算两个结果的和：

```
package main

import "fmt"

func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // 把 sum 发送到通道 c
}

```

```
func main() {
        s := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // 从通道 c 中接收

        fmt.Println(x, y, x+y)
}
```

通道缓冲区

通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

```
ch := make(chan int, 100)
```

带缓冲区的通道允许发送端的数据发送和接收端的数据获取处于异步状态，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。

不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

注意：如果通道不带缓冲，发送方会阻塞直到接收方从通道中接收了值。如果通道带缓冲，发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞。

实例

```
package main

import "fmt"

func main() {
    // 这里我们定义了一个可以存储整数类型的带缓冲通道
        // 缓冲区大小为2
```

```
        ch := make(chan int, 2)

        // 因为 ch 是带缓冲的通道，我们可以同时发送两个数据
        // 而不用立刻需要去同步读取数据
        ch <- 1
        ch <- 2

        // 获取这两个数据
        fmt.Println(<-ch)
        fmt.Println(<-ch)
}
```

遍历通道与关闭通道

Go 通过 range 关键字来实现遍历读取到的数据，类似于与数组或切片。格式如下：

```
v, ok := <-ch
```

如果通道接收不到数据后 ok 就为 false，这时通道就可以使用 close() 函数来关闭。

实例

```
package main

import (
        "fmt"
)

func fibonacci(n int, c chan int) {
        x, y := 0, 1
        for i := 0; i < n; i++ {
                c <- x
                x, y = y, x+y
        }
        close(c)
}
```

```
15
16 func main() {
17         c := make(chan int, 10)
18         go fibonacci(cap(c), c)
19         // range 函数遍历每个从通道接收到的数据，因为 c 在发送完 10 个
20         // 数据之后就关闭了通道，所以这里我们 range 函数在接收到 10 个数据
21         // 之后就结束了。如果上面的 c 通道不关闭，那么 range 函数就不
22         // 会结束，从而在接收第 11 个数据的时候就阻塞了。
23         for i := range c {
24                 fmt.Println(i)
25         }
26 }
```

Go 并发（goroutine）

Go 语言支持并发，我们只需要通过 go 关键字来开启 goroutine 即可。

goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。

goroutine 语法格式：

```
go 函数名( 参数列表 )
```

例如：

```
go f(x, y, z)
```

Go 允许使用 go 语句开启一个新的运行期线程， 即 goroutine，以一个不同的、新创建的 goroutine 来执行一个函数。 同一个程序中的所有 goroutine 共享同一个地址空间。

实例

```
package main

import (
        "fmt"
        "time"
)

func say(s string) {
        for i := 0; i < 5; i++ {
                time.Sleep(100 * time.Millisecond)
                fmt.Println(s)
        }
}

func main() {
```

```
16          go say("world")
17          say("hello")
18  }
```

执行以上代码，你会看到输出的 hello 和 world 是没有固定先后顺序。因为它们是两个 goroutine 在执行：

```
1  world
2  hello
3  hello
4  world
5  world
6  hello
7  hello
8  world
9  world
10 hello
```

Go 错误处理

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // 实现
}
```

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

实例

```go
package main

import (
    "fmt"
)

// 定义一个 DivideError 结构
type DivideError struct {
    dividee int
    divider int
}

// 实现 `error` 接口
func (de *DivideError) Error() string {
    strFormat := `
    Cannot proceed, the divider is zero.
    dividee: %d
    divider: 0
`
    return fmt.Sprintf(strFormat, de.dividee)
}

// 定义 `int` 类型除法运算的函数
func Divide(varDividee int, varDivider int) (result int, errorMsg st
ring) {
    if varDivider == 0 {
            dData := DivideError{
                    dividee: varDividee,
                    divider: varDivider,
            }
            errorMsg = dData.Error()
            return
    } else {
            return varDividee / varDivider, ""
    }

}

func main() {
```

```
39
40      // 正常情况
41      if result, errorMsg := Divide(100, 10); errorMsg == "" {
42              fmt.Println("100/10 = ", result)
43      }
44      // 当被除数为零的时候会返回错误信息
45      if _, errorMsg := Divide(100, 0); errorMsg != "" {
46              fmt.Println("errorMsg is: ", errorMsg)
47      }
48
49  }
```

执行以上程序，输出结果为：

```
1  100/10 =  10
2  errorMsg is:
3      Cannot proceed, the divider is zero.
4      dividee: 100
5      divider: 0
```

Go 语言接口（interface）

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

```
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 */
func (struct_name_variable struct_name) method_name1() [return_type]
{
   /* 方法实现 */
}
...
func (struct_name_variable struct_name) method_namen() [return_type]
{
   /* 方法实现*/
}
```

实例

```
package main

import (
```

```
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()。然后我们在main函数里面定义了一个Phone类型变量，并分别为之赋值为NokiaPhone和IPhone。然后调用call()方法，输出结果如下：

I am Nokia, I can call you!
I am iPhone, I can call you!

Go 语言Map(集合)

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，**Map 是无序的，我们无法决定它的返回顺序**，这是因为 Map 是使用 hash 表来实现的。

定义 Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type

/* 使用 make 函数 */
map_variable := make(map[key_data_type]value_data_type)
```

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

实例

```
package main

import "fmt"

func main() {
    var countryCapitalMap map[string]string /*创建集合 */
    countryCapitalMap = make(map[string]string)

    /* map插入key - value对,各个国家对应的首都 */
    countryCapitalMap [ "France" ] = "巴黎"
    countryCapitalMap [ "Italy" ] = "罗马"
    countryCapitalMap [ "Japan" ] = "东京"
```

```
    countryCapitalMap [ "India " ] = "新德里"

    /*使用键输出地图值 */
    for country := range countryCapitalMap {
        fmt.Println(country, "首都是", countryCapitalMap [country])
    }

    /*查看元素在集合中是否存在 */
    capital, ok := countryCapitalMap [ "American" ] /*如果确定是真实的,则存在,否则不存在 */
    /*fmt.Println(capital) */
    /*fmt.Println(ok) */
    if (ok) {
        fmt.Println("American 的首都是", capital)
    } else {
        fmt.Println("American 的首都不存在")
    }

    /*查看元素在集合中是否存在 */
    capital, ok := countryCapitalMap [ "India" ] /*如果确定是真实的,则存在,否则不存在 */
    fmt.Println(capital) // 新德里
    fmt.Println(ok) // true
}
```

delete() 函数

delete() 函数用于删除集合的元素, 参数为 map 和其对应的 key。实例如下：

```
package main

import "fmt"

func main() {
        /* 创建map */
        countryCapitalMap := map[string]string{"France": "Paris", "Italy": "Rome", "Japan": "Tokyo", "India": "New delhi"}
```

```
 8
 9          fmt.Println("原始地图")
10
11          /* 打印地图 */
12          for country := range countryCapitalMap {
13                  fmt.Println(country, "首都是", countryCapitalMap [ co
    untry ])
14          }
15
16          /*删除元素*/
17          delete(countryCapitalMap, "France")
18
19          fmt.Println("法国条目被删除")
20
21          fmt.Println("删除元素后地图")
22
23          /*打印地图*/
24          for country := range countryCapitalMap {
25                  fmt.Println(country, "首都是", countryCapitalMap [ co
    untry ])
26          }
27  }
```

Go 语言范围(Range)

Go 语言中 range 关键字用于 for 循环中迭代数组(array)、切片(slice)、通道(channel)或集合(map)的元素。在数组和切片中它返回元素的索引和索引对应的值，在集合中返回 key-value 对的 key 值。

```
package main
import "fmt"
func main() {
    //这是我们使用range去求一个slice的和。使用数组跟这个很类似
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum) // 9
    //在数组上使用range将传入index和值两个变量。上面那个例子我们不需要使用该元素的序号，所以我们使用空白符"_"省略了。有时侯我们确实需要知道它的索引。
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i) // 1
        }
    }
    //range也可以用在map的键值对上。
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v) // a -> apple
    }
    //range也可以用来枚举Unicode字符串。第一个参数是字符的索引，第二个是字符（Unicode的值）本身。
    for i, c := range "go" {
        fmt.Println(i, c) // 0 103   /n   1 111
    }
}
```

Go 语言切片(Slice)

Go 语言切片是对数组的抽象。

Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

定义切片

你可以声明一个未指定大小的数组来定义切片：

```
1 var identifier []type
```

切片不需要说明长度。

或使用make()函数来创建切片:

```
1 var slice1 []type = make([]type, len)
```

也可以简写为

```
1 slice1 := make([]type, len)
```

也可以指定容量，其中capacity为可选参数。

```
1 make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

切片初始化

```
1 s :=[] int {1,2,3 }
```

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是1,2,3.其cap=len=3

```
1 s := arr[:]
```

初始化切片s,是数组arr的引用

```
1 s := arr[startIndex:endIndex]
```

将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片

```
1 s := arr[startIndex:]
```

默认 endIndex 时将表示一直到arr的最后一个元素

```
1 s := arr[:endIndex]
```

默认 startIndex 时将表示从arr的第一个元素开始

```
1 s1 := s[startIndex:endIndex]
```

通过切片s初始化切片s1

```
1 s :=make([]int,len,cap)
```

通过内置函数make()初始化切片s,[]int 标识为其元素类型为int的切片

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

实例

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)

   printSlice(numbers)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x) // len=3 cap=5 slice=[0 0 0]
}
```

空(nil)切片

一个切片在未初始化之前默认为 nil，长度为 0，实例如下：

```
package main

import "fmt"

func main() {
   var numbers []int

   printSlice(numbers)

```

```
    if(numbers == nil){
       fmt.Printf("切片是空的")
    }
}

func printSlice(x []int){
    fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

切片截取

可以通过设置下限及上限来设置截取切片 [lower-bound:upper-bound]，实例如下：

```
package main

import "fmt"

func main() {
    /* 创建切片 */
    numbers := []int{0,1,2,3,4,5,6,7,8}
    printSlice(numbers)

    /* 打印原始切片 */
    fmt.Println("numbers ==", numbers)

    /* 打印子切片从索引1(包含) 到索引4(不包含)*/
    fmt.Println("numbers[1:4] ==", numbers[1:4]) // [1 2 3]

    /* 默认下限为 0*/
    fmt.Println("numbers[:3] ==", numbers[:3]) // [0 1 2]

    /* 默认上限为 len(s)*/
    fmt.Println("numbers[4:] ==", numbers[4:]) //[4 5 6 7 8]

    numbers1 := make([]int,0,5)
    printSlice(numbers1) // len=0 cap=5 slice=[]

```

```
   /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
   number2 := numbers[:2]
   printSlice(number2) // len=2 cap=9 slice=[0 1]

   /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
   number3 := numbers[2:5]
   printSlice(number3) // len=3 cap=7 slice=[2 3 4]

}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

append() 和 copy() 函数

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
```

```
   numbers = append(numbers, 2,3,4)
   printSlice(numbers) // len=5 cap=6 slice=[0 1 2 3 4]

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)  // len=5 cap=12 slice=[0 1 2 3 4]
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

Go 语言结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

结构体表示一项记录，比如保存图书馆的书籍记录，每本书有以下属性：

```
Title ：标题
Author ： 作者
Subject：学科
ID：书籍ID
```

定义结构体

结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体有中有一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
或
variable_name := structure_variable_type { key1: value1, key2: value
2..., keyn: valuen}
```

实例如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}


func main() {

    // 创建一个新的结构体
    fmt.Println(Books{"Go 语言", "www.runoob.com", "Go 语言教程", 6495407})

    // 也可以使用 key => value 格式
    fmt.Println(Books{title: "Go 语言", author: "www.runoob.com", subject: "Go 语言教程", book_id: 6495407})

    // 忽略的字段为 0 或 空
   fmt.Println(Books{title: "Go 语言", author: "www.runoob.com"})
}
```

访问结构体成员

如果要访问结构体成员，需要使用点号 . 操作符，格式为：

```
结构体.成员名"
```

结构体类型变量使用 struct 关键字定义，实例如下：

```go
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* 打印 Book2 信息 */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

结构体作为函数参数

你可以像其他数据类型一样将结构体类型作为参数传递给函数。并以以上实例的方式访问结构体变量：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(Book1)

   /* 打印 Book2 信息 */
   printBook(Book2)
}
```

```
34
35 func printBook( book Books ) {
36    fmt.Printf( "Book title : %s\n", book.title);
37    fmt.Printf( "Book author : %s\n", book.author);
38    fmt.Printf( "Book subject : %s\n", book.subject);
39    fmt.Printf( "Book book_id : %d\n", book.book_id);
40 }
```

结构体指针

你可以定义指向结构体的指针类似于其他指针变量，格式如下：

```
1 var struct_pointer *Books
```

以上定义的指针变量可以存储结构体变量的地址。查看结构体变量地址，可以将 & 符号放置于结构体变量前：

```
1 struct_pointer = &Book1;
```

使用结构体指针访问结构体成员，使用 "." 操作符：

```
1 struct_pointer.title;
```

接下来让我们使用结构体指针重写以上实例，代码如下：

```
1 package main
2
3 import "fmt"
4
5 type Books struct {
6    title string
7    author string
8    subject string
```

```
   book_id int
}

func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(&Book1)

   /* 打印 Book2 信息 */
   printBook(&Book2)
}
func printBook( book *Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

Go 语言指针

什么是指针

一个指针变量指向了一个值的内存地址。

类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

```
var var_name *var-type
```

var-type 为指针类型，var_name 为指针变量名，* 号用于指定变量是作为一个指针。以下是有效的指针声明：

```
var ip *int        /* 指向整型*/
var fp *float32    /* 指向浮点型 */
```

指针使用流程：

定义指针变量。

为指针变量赋值。

访问指针变量中指向地址的值。

在指针类型前面加上 * 号（前缀）来获取指针所指向的内容。

实例

```
package main

import "fmt"

```

```
func main() {
   var a int= 20   /* 声明实际变量 */
   var ip *int         /* 声明指针变量 */

   ip = &a  /* 指针变量的存储地址 */

   fmt.Printf("a 变量的地址是: %x\n", &a  ) //20818a220

   /* 指针变量的存储地址 */
   fmt.Printf("ip 变量储存的指针地址: %x\n", ip ) //20818a220

   /* 使用指针访问值 */
   fmt.Printf("*ip 变量的值: %d\n", *ip ) //20
}
```

Go 空指针

当一个指针被定义后没有分配到任何变量时，它的值为 nil。

nil 指针也称为空指针。

nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。

一个指针变量通常缩写为 ptr。

实例

```
package main

import "fmt"

func main() {
   var  ptr *int

   fmt.Printf("ptr 的值为 : %x\n", ptr  ) // 0
}
```

空指针判断：

```
if(ptr != nil)     /* ptr 不是空指针 */
if(ptr == nil)    /* ptr 是空指针 */
```

指针数组

有时我们可能需要保存数组，这样我们就需要使用到指针。

以下声明了整型指针数组：

```
var ptr [MAX]*int;
```

ptr 为整型指针数组。因此每个元素都指向了一个值。以下实例的三个整数将存储在指针数组中：

```
package main

import "fmt"

const MAX int = 3

func main() {
   a := []int{10,100,200}
   var i int
   var ptr [MAX]*int;

   for  i = 0; i < MAX; i++ {
      ptr[i] = &a[i] /* 整数地址赋值给指针数组 */
   }

   for  i = 0; i < MAX; i++ {
      fmt.Printf("a[%d] = %d\n", i,*ptr[i] )
   }
```

```
}
```

指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：

指向指针的指针变量声明格式如下：

```
var ptr **int;
```

以上指向指针的指针变量为整型。

访问指向指针的指针变量值需要使用两个 * 号，如下所示：

```
package main

import "fmt"

func main() {

   var a int
   var ptr *int
   var pptr **int

   a = 3000

   /* 指针 ptr 地址 */
   ptr = &a

   /* 指向指针 ptr 地址 */
   pptr = &ptr

   /* 获取 pptr 的值 */
   fmt.Printf("变量 a = %d\n", a ) //3000
```

```
    fmt.Printf("指针变量 *ptr = %d\n", *ptr ) //3000
    fmt.Printf("指向指针的指针变量 **pptr = %d\n", **pptr) //3000
}
```

指针作为函数参数

Go 语言允许向函数传递指针，只需要在函数定义的参数上设置为指针类型即可。

以下实例演示了如何向函数传递指针，并在函数调用后修改函数内的值，：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前 a 的值 : %d\n", a )
   fmt.Printf("交换前 b 的值 : %d\n", b )

   /* 调用函数用于交换值
   * &a 指向 a 变量的地址
   * &b 指向 b 变量的地址
   */
   swap(&a, &b);

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址的值 */
   *x = *y      /* 将 y 赋值给 x */
   *y = temp    /* 将 temp 赋值给 y */
```

```
28 }
```

Go 语言数组

数组

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

数组元素可以通过索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。

声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

```
var variable_name [SIZE] variable_type

例：

var balance [10] float32
```

初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
var balance = [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

该实例与上面的实例是一样的，虽然没有设置数组的大小。

balance[4] = 50.0
以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。

访问数组元素

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

```
var salary float32 = balance[9]
```

以上实例读取了数组balance第10个元素的值。

实例

```
package main

import "fmt"

func main() {
   var n [10]int /* n 是一个长度为 10 的数组 */
   var i,j int

   /* 为数组 n 初始化元素 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* 设置元素为 i + 100 */
   }

   /* 输出每个数组元素的值 */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type

var threedim [5][10][4]int
```

二维数组

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

```
var arrayName [ x ][ y ] variable_type
```

variable_type 为 Go 语言的数据类型，arrayName 为数组名，二维数组可认为是一个表格，x 为行，y 为列.

二维数组中的元素可通过 a[i][j] 来访问。

初始化二维数组

多维数组可通过大括号来初始值。以下实例为一个 3 行 4 列的二维数组：

```
a = [3][4]int{
 {0, 1, 2, 3} ,   /*  第一行索引为 0 */
 {4, 5, 6, 7} ,   /*  第二行索引为 1 */
 {8, 9, 10, 11},   /* 第三行索引为 2 */
}
```

注意：以上代码中倒数第二行的 } 必须要有逗号，因为最后一行的 } 不能单独一行，也可以写成这样：

```
a = [3][4]int{
```

```
 {0, 1, 2, 3} ,   /*  第一行索引为 0 */
 {4, 5, 6, 7} ,   /*  第二行索引为 1 */
 {8, 9, 10, 11}}   /* 第三行索引为 2 */
```

访问二维数组

二维数组通过指定坐标来访问。如数组中的行索引与列索引，例如：

```
val := a[2][3]
或
var value int = a[2][3]
```

实例

```
package main

import "fmt"

func main() {
   /* 数组 - 5 行 2 列*/
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int

   /* 输出数组元素 */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
```

向函数传递数组

如果你想向函数传递数组参数，你需要在函数定义时，声明形参为数组，我们可以通过以下两种方式来声明：

方式一
形参设定数组大小：

```
void myFunction(param [10]int)
{
.
.
.
}
```

方式二
形参未设定数组大小：

```
void myFunction(param []int)
{
.
.
.
}
```

实例

```
func getAverage(arr []int, size int) float32
{
   var i int
   var avg, sum float32

   for i = 0; i < size; ++i {
      sum += arr[i]
   }

   avg = sum / size

```

```
    return avg;
}
```

接下来我们来调用这个函数：

实例

```
package main

import "fmt"

func main() {
   /* 数组长度为 5 */
   var  balance = [5]int {1000, 2, 3, 17, 50}
   var avg float32

   /* 数组作为参数传递给函数 */
   avg = getAverage( balance, 5 ) ;

   /* 输出返回的平均值 */
   fmt.Printf( "平均值为: %f ", avg );
}

func getAverage(arr [5]int, size int) float32 {
   var i,sum int
   var avg float32

   for i = 0; i < size;i++ {
      sum += arr[i]
   }

   avg = float32(sum) / float32(size)

   return avg;
}
```

以上实例执行输出结果为：

平均值为: 214.399994

以上实例中我们使用的形参并未设定数组大小。

浮点数计算输出有一定的偏差，你也可以转整型来设置精度。

实例

```
package main
import (
    "fmt"
)
func main() {
    a := 1.69
    b := 1.7
    c := a * b      // 结果应该是2.873
    fmt.Println(c)  // 输出的是2.8729999999999998
}
```

设置固定精度：

```
package main
import (
    "fmt"
)
func main() {
    a := 1690           // 表示1.69
    b := 1700           // 表示1.70
    c := a * b          // 结果应该是2873000表示 2.873
    fmt.Println(c)      // 内部编码
    fmt.Println(float64(c) / 1000000) // 显示
}
```

Go 语言函数

函数声明告诉了编译器函数的名称，返回类型，和参数。

Go 语言函数定义格式如下：

```
func function_name( [parameter list] ) [return_types] {
   函数体
}

函数定义解析：

func：函数由 func 开始声明
function_name：函数名称，函数名和参数列表一起构成了函数签名。
parameter list：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。
    参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
return_types：返回类型，函数返回一列值。return_types 是该列值的数据类型。
    有些功能不需要返回值，这种情况下 return_types 不是必须的。
函数体：函数定义的代码集合。
```

Go 语言最少有个 main() 函数。

实例

以下实例为 max() 函数的代码，该函数传入两个整型参数 num1 和 num2，并返回这两个参数的最大值：

```
/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 声明局部变量 */
   var result int

```

```
    if (num1 > num2) {
       result = num1
    } else {
       result = num2
    }
    return result
}
```

函数调用

当创建函数时，你定义了函数需要做什么，通过调用该函数来执行指定任务。

调用函数，向函数传递参数，并返回值，例如：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200
   var ret int

   /* 调用函数并返回最大值 */
   ret = max(a, b)

   fmt.Printf( "最大值是 : %d\n", ret )
}

/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 定义局部变量 */
   var result int

   if (num1 > num2) {
      result = num1
```

```
    } else {
        result = num2
    }
    return result
}
```

Go 函数可以返回多个值，例如：

```
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("Google", "Runoob")
    fmt.Println(a, b)
}
```

函数参数

函数如果使用参数，该变量可称为函数的形参。

形参就像定义在函数体内的局部变量。

调用函数，可以通过两种方式来传递参数：

值传递

值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

引用传递

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

引用传递实例：

```
package main

import "fmt"

func main() {
    /* 定义局部变量 */
    var a int = 100
    var b int= 200

    fmt.Printf("交换前，a 的值 : %d\n", a )
    fmt.Printf("交换前，b 的值 : %d\n", b )

    /* 调用 swap() 函数
     * &a 指向 a 指针，a 变量的地址
     * &b 指向 b 指针，b 变量的地址
     */
    swap(&a, &b)

    fmt.Printf("交换后，a 的值 : %d\n", a )
    fmt.Printf("交换后，b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
    var temp int
    temp = *x    /* 保存 x 地址上的值 */
    *x = *y      /* 将 y 值赋给 x */
    *y = temp    /* 将 temp 值赋给 y */
}
```

函数用法

函数作为另外一个函数的实参

函数定义后可作为另外一个函数的实参数传入

实例

```
package main

import (
   "fmt"
   "math"
)

func main(){
   /* 声明函数变量 */
   getSquareRoot := func(x float64) float64 {
      return math.Sqrt(x)
   }

   /* 使用函数 */
   fmt.Println(getSquareRoot(9))

}
```

闭包

闭包是匿名函数，可在动态编程中使用

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence()，返回另外一个函数。该函数的目的是在闭包中递增 i 变量，代码如下：

实例

```go
package main

import "fmt"

func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
     return i
   }
}

func main(){
   /* nextNumber 为一个函数，函数 i 为 0 */
   nextNumber := getSequence()

   /* 调用 nextNumber 函数，i 变量自增 1 并返回 */
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())

   /* 创建新的函数 nextNumber1，并查看结果 */
   nextNumber1 := getSequence()
   fmt.Println(nextNumber1())
   fmt.Println(nextNumber1())
}
```

以上代码执行结果为：

```
1
2
3
1
2
```

方法

方法就是一个包含了接受者的函数

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。
所有给定类型的方法属于该类型的方法集。语法格式如下：

```
func (variable_name variable_data_type) function_name() [return_type]{
   /* 函数体*/
}
```

下面定义一个结构体类型和该类型的一个方法：

```
package main

import (
   "fmt"
)

/* 定义结构体 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  fmt.Println("圆的面积 = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

Go 语言循环语句

for循环

语法
Go语言的For循环有3中形式，只有其中的一种使用分号。

```
for init; condition; post { }


init:  一般为赋值表达式，给控制变量赋初值；
condition:  关系表达式或逻辑表达式，循环控制条件；
post:  一般为赋值表达式，给控制变量增量或减量。
```

```
for condition { }
```

```
for { }
```

for 循环的 range 格式可以对 **slice**、**map**、**数组**、**字符串**等进行迭代循环。格式如下：

```
for key, value := range oldMap {
    newMap[key] = value
}
```

实例

```
package main

import "fmt"

```

```
func main() {

   var b int = 15
   var a int

   numbers := [6]int{1, 2, 3, 5}

   /* for 循环 */
   for a := 0; a < 10; a++ {
      fmt.Printf("a 的值为: %d\n", a)
   }

   for a < b {
      a++
      fmt.Printf("a 的值为: %d\n", a)
   }

   for i,x:= range numbers {
      fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)
   }

   var i, j int

   // 嵌套循环
   for i=2; i < 100; i++ {
      for j=2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // 如果发现因子，则不是素数
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d  是素数\n", i);
      }
   }

   // 无限循环
   for true  {
        fmt.Printf("这是无限循环。\n");
    }
}
```

break & continue, 用法同php

break 用于循环语句中跳出循环，并开始执行循环之后的语句。

continue 用于跳过当前循环执行下一次循环语句。

for 循环中，执行 continue 语句会触发for增量语句的执行。

goto 语句(一般不用)

Go 语言的 goto 语句可以无条件地转移到过程中指定的行。

goto 语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。

但是，在结构化程序设计中一般不主张使用 goto 语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

实例

在变量 a 等于 15 的时候跳过本次循环并回到循环的开始语句 LOOP 处：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 循环 */
   LOOP: for a < 20 {
      if a == 15 {
         /* 跳过迭代 */
         a = a + 1
         goto LOOP
      }
```

```
        fmt.Printf("a的值为 : %d\n", a)
        a++
    }
}
```

Go 语言条件语句

if 语句

```
package main

import "fmt"

func main() {
  /* 局部变量定义 */
   var a int = 100;

   /* 判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" );
   } else {
       /* 如果条件为 false 则执行以下语句 */
       fmt.Printf("a 不小于 20\n" );
   }
   fmt.Printf("a 的值为 : %d\n", a);
}
```

switch语句

switch 语句用于基于不同条件执行不同动作，**每一个 case 分支都是唯一的，从上至下逐一测试，直到匹配为止**。

switch 语句执行的过程从上至下，直到找到匹配项，**匹配项后面也不需要再加 break**。

switch 默认情况下 case 最后自带 break 语句，匹配成功后就不会执行其他 case，如果我们需要执行后面的 case，可以使用 fallthrough 。

```
package main
```

```
2
3 import "fmt"
4
5 func main() {
6    /* 定义局部变量 */
7    var grade string = "B"
8    var marks int = 90
9
10   switch marks {
11      // case种值必须是相同的类型；或者最终结果为相同类型的表达式。
12      case 90: grade = "A"
13      case 80: grade = "B"
14      case 50,60,70 : grade = "C"
15      default: grade = "D"
16   }
17
18   switch {
19      case grade == "A" :
20         fmt.Printf("优秀!\n" )
21      case grade == "B", grade == "C" :
22         fmt.Printf("良好\n" )
23      case grade == "D" :
24         fmt.Printf("及格\n" )
25      case grade == "F":
26         fmt.Printf("不及格\n" )
27      default:
28         fmt.Printf("差\n" );
29   }
30   fmt.Printf("你的等级是 %s\n", grade );
31 }
```

Type Switch

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

```
1 package main
2
```

```
import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
      case nil:
         fmt.Printf(" x 的类型 :%T",i)
      case int:
         fmt.Printf("x 是 int 型")
      case float64:
         fmt.Printf("x 是 float64 型")
      case func(int) float64:
         fmt.Printf("x 是 func(int) 型")
      case bool, string:
         fmt.Printf("x 是 bool 或 string 型" )
      default:
         fmt.Printf("未知型")
   }
}
```

fallthrough

使用 fallthrough 会强制执行后面的 case 语句，fallthrough 不会判断下一条 case 的表达式结果是否为 true。(相当于php没有break的情况)

实例

```
package main

import "fmt"

func main() {

    switch {
    case false:
            fmt.Println("1、case 条件语句为 false")
```

```
            fallthrough
    case true:
            fmt.Println("2、case 条件语句为 true")
            fallthrough
    case false:
            fmt.Println("3、case 条件语句为 false")
            fallthrough
    case true:
            fmt.Println("4、case 条件语句为 true")
    case false:
            fmt.Println("5、case 条件语句为 false")
            fallthrough
    default:
            fmt.Println("6、默认 case")
    }
}
```

以上代码执行结果为：

```
2、case 条件语句为 true
3、case 条件语句为 false
4、case 条件语句为 true
```

从以上代码输出的结果可以看出：switch 从第一个判断表达式为 true 的 case 开始执行，如果 case 带有 fallthrough，程序会继续执行下一条 case，且它不会去判断下一个 case 的表达式是否为 true。

select 语句

select 是 Go 中的一个控制结构，类似于用于通信的 switch 语句。每个 case 必须是一个通信操作，要么是发送要么是接收。

select 随机执行一个可运行的 case。如果没有 case 可运行，它将阻塞，直到有 case 可运行。一个默认的子句应该总是可运行的。

语法:

Go 编程语言中 select 语句的语法如下：

```
select {
    case communication clause  :
       statement(s);
    case communication clause  :
       statement(s);
    /* 你可以定义任意数量的 case */
    default : /* 可选 */
       statement(s);
}
```

每个 case 都必须是一个通信
所有 channel 表达式都会被求值
所有被发送的表达式都会被求值
如果任意某个通信可以进行，它就执行，其他被忽略。
如果有多个 case 都可以运行，Select 会随机公平地选出一个执行。其他不会执行。

否则：
如果有 default 子句，则执行该语句。
如果没有 default 子句，select 将阻塞，直到某个通信可以运行；Go 不会重新对 channel 或值进行求值。

实例

```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
      case c2 <- i2:
         fmt.Printf("sent ", i2, " to c2\n")
      case i3, ok := (<-c3):  // same as: i3, ok := <-c3
```

```
14          if ok {
15             fmt.Printf("received ", i3, " from c3\n")
16          } else {
17             fmt.Printf("c3 is closed\n")
18          }
19       default:
20          fmt.Printf("no communication\n")
21    }
22 }
```

以上代码执行结果为：

no communication

Go 语言运算符

算术运算符

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

```
+    相加   A + B 输出结果 30
-    相减   A - B 输出结果 -10
*    相乘   A * B 输出结果 200
/    相除   B / A 输出结果 2
%    求余   B % A 输出结果 0
++   自增   A++ 输出结果 11
--   自减   A-- 输出结果 9
```

关系运算符

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

```
==   检查两个值是否相等，如果相等返回 True 否则返回 False。    (A == B) 为 False
!=   检查两个值是否不相等，如果不相等返回 True 否则返回 False。 (A != B) 为 True
>    检查左边值是否大于右边值，如果是返回 True 否则返回 False。 (A > B) 为 False
<    检查左边值是否小于右边值，如果是返回 True 否则返回 False。 (A < B) 为 True
>=   检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。    (A >= B)
   为 False
<=   检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。    (A <= B)
   为 True
```

逻辑运算符

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

```
&&   逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。 (A
```

```
  && B) 为 False
2 ||  逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。
   (A || B) 为 True
3 !   逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。
   !(A && B) 为 True
```

实例：

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main() {
 6
 7    var a int = 21
 8    var b int = 10
 9    var c int
10
11    c = a + b
12    fmt.Printf("第一行 - c 的值为 %d\n", c )
13
14
15    if( a == b ) {
16       fmt.Printf("第一行 - a 等于 b\n" )
17    } else {
18       fmt.Printf("第一行 - a 不等于 b\n" )
19    }
20
21    var d bool = true
22    var e bool = false
23    if ( d && e ) {
24       fmt.Printf("第一行 - 条件为 true\n" )
25    }
26
27 }
```

位运算符

位运算符对整数在内存中的二进制位进行操作。

下表列出了位运算符 &, |, 和 ^ 的计算：

```
p   q   p & q     p | q  p ^ q
0   0     0       0      0
0   1     0       1      1
1   1     1       1      0
1   0     0       1      1
```

假定 A = 60; B = 13; 其二进制数转换为：

```
A = 0011 1100

B = 0000 1101
```

```
A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001
```

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

```
&    按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。  (A & B) 结果为 12, 二进制为 0000 1100
|    按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或   (A | B) 结果为 61, 二进制为 0011 1101
^    按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。   (A ^ B) 结果为 49, 二进制为 0011 0001
<<   左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。    A << 2 结果为 240 ，二进制为 1111 0000
```

```
5 >>  右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。 A >> 2 结果为 15 ，二进制为 0000 1111
```

以下实例演示了位运算符的用法：

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main() {
 6
 7    var a uint = 60      /* 60 = 0011 1100 */
 8    var b uint = 13      /* 13 = 0000 1101 */
 9    var c uint = 0
10
11    c = a & b       /* 12 = 0000 1100 */
12    fmt.Printf("第一行 - c 的值为 %d\n", c )
13
14    c = a | b       /* 61 = 0011 1101 */
15    fmt.Printf("第二行 - c 的值为 %d\n", c )
16
17    c = a ^ b       /* 49 = 0011 0001 */
18    fmt.Printf("第三行 - c 的值为 %d\n", c )
19
20    c = a << 2     /* 240 = 1111 0000 */
21    fmt.Printf("第四行 - c 的值为 %d\n", c )
22
23    c = a >> 2     /* 15 = 0000 1111 */
24    fmt.Printf("第五行 - c 的值为 %d\n", c )
25 }
```

赋值运算符

下表列出了所有Go语言的赋值运算符。

```
=    简单的赋值运算符，将一个表达式的值赋给一个左值 C = A + B 将 A + B 表达式结果赋值给 C
+=   相加后再赋值   C += A 等于 C = C + A
-=   相减后再赋值   C -= A 等于 C = C - A
*=   相乘后再赋值   C *= A 等于 C = C * A
/=   相除后再赋值   C /= A 等于 C = C / A
%=   求余后再赋值   C %= A 等于 C = C % A
<<= 左移后赋值    C <<= 2 等于 C = C << 2
>>= 右移后赋值    C >>= 2 等于 C = C >> 2
&=   按位与后赋值   C &= 2 等于 C = C & 2
^=   按位异或后赋值 C ^= 2 等于 C = C ^ 2
|=   按位或后赋值   C |= 2 等于 C = C | 2
```

以下实例演示了赋值运算符的用法：

实例

```
package main

import "fmt"

func main() {
   var a int = 21
   var c int

   c =  a
   fmt.Printf("第 1 行 - =  运算符实例，c 值为 = %d\n", c )

   c +=  a
   fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

   c  = 200;

   c <<=  2
   fmt.Printf("第 6行  - <<= 运算符实例，c 值为 = %d\n", c )

   c >>=  2
   fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )
```

```
22
23    c &=  2
24    fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )
25
26    c ^=  2
27    fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )
28
29    c |=  2
30    fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )
31
32 }
```

其他运算符

下表列出了Go语言的其他运算符。

```
1 &    返回变量存储地址      &a; 将给出变量的实际地址。
2 *    指针变量。     *a; 是一个指针变量
```

以下实例演示了其他运算符的用法：

实例

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main() {
 6    var a int = 4
 7    var b int32
 8    var c float32
 9    var ptr *int
10
11    /* 运算符实例 */
12    fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );
13    fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );
```

```
    fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

    /*  & 和 * 运算符实例 */
    ptr = &a     /* 'ptr' 包含了 'a' 变量的地址 */
    fmt.Printf("a 的值为  %d\n", a);
    fmt.Printf("*ptr 为 %d\n", *ptr);
}
```

以上实例运行结果：

```
第 1 行 - a 变量类型为 = int
第 2 行 - b 变量类型为 = int32
第 3 行 - c 变量类型为 = float32
a 的值为  4
*ptr 为 4
```

运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

```
优先级 运算符
  7    ^ !
  6    * / % << >> & &^
  5    + - | ^
  4    == != < <= >= >
  3    <-
  2    &&
  1    ||
```

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

Go 语言常量

常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

```
const identifier [type] = value
```

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

```
显式类型定义：  const b string = "abc"
隐式类型定义：  const b = "abc"
```

多个相同类型的声明可以简写为：

```
const c_name1, c_name2 = value1, value2
```

常量的应用：

实例

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int
   const a, b, c = 1, false, "str" //多重赋值

```

```
    area = LENGTH * WIDTH
    fmt.Printf("面积为 : %d", area)
    println()
    println(a, b, c)
}
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量可以用len(), cap(), unsafe.Sizeof()函数计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

实例

```
package main

import "unsafe"
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)

func main(){
    println(a, b, c)
}
```

iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

iota 在 const关键字出现时将被重置为 0(const 内部的第一行之前)，const 中每新增一行常量声明将使 iota 计数一次(iota 可理解为 const 语句块中的行索引)。

iota 可以被用作枚举值：

```
const (
    a = iota
    b = iota
    c = iota
)
```

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

```
const (
    a = iota
    b
    c
)
```

```
package main

import "fmt"

func main() {
    const (
            a = iota   //0
            b          //1
            c          //2
            d = "ha"   //独立值，iota += 1
            e          //"ha"   iota += 1
            f = 100    //iota +=1
            g          //100  iota +=1
```

```
            h = iota    //7,恢复计数
            i           //8
    )
    fmt.Println(a,b,c,d,e,f,g,h,i)
}
```

以上实例运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

再看个有趣的的 iota 实例：

```
package main

import "fmt"
const (
    i=1<<iota
    j=3<<iota
    k
    l
)

func main() {
    fmt.Println("i=",i)
    fmt.Println("j=",j)
    fmt.Println("k=",k)
    fmt.Println("l=",l)
}
```

以上实例运行结果为：

```
i= 1
j= 6
k= 12
l= 24
```

iota 表示从 0 开始自动加 1，所以 i=1<<0, j=3<<1（<< 表示左移的意思），即：i=1, j=6，这没问题，关键在 k 和 l，从输出结果看 k=3<<2，l=3<<3。

简单表述:

```
i=1：左移 0 位,不变仍为 1;
j=3：左移 1 位,变为二进制 110, 即 6;
k=3：左移 2 位,变为二进制 1100, 即 12;
l=3：左移 3 位,变为二进制 11000,即 24。
```

Go 语言变量

变量声明

第一种，指定变量类型，声明后若不赋值，使用默认值。

```
var v_name v_type
v_name = value

或
var v_name v_type = value
```

第二种，根据值自行判定变量类型。

```
var v_name = value
```

第三种，省略var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。

> 这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

```
v_name := value

// 例如
var a int = 10
var b = 10
c := 10
```

实例：

```
package main
```

```
2
3 import "fmt"
4
5 var a = "haha"
6 var b string = "oumu"
7 var c bool
8
9 func main(){
10     fmt.Println(a, b, c)
11 }
```

以上实例执行结果为：

haha oumu false

多变量声明

```
1 //类型相同多个变量, 非全局变量
2 var vname1, vname2, vname3 type
3 vname1, vname2, vname3 = v1, v2, v3
4
5 var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类
  型，自动推断
6
7 vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明
  过的，否则会导致编译错误
8
9
10 // 这种因式分解关键字的写法一般用于声明全局变量
11 var (
12     vname1 v_type1
13     vname2 v_type2
14 )
```

实例如下：

```go
package main

var x, y int
var (  // 这种因式分解关键字的写法一般用于声明全局变量
    a int
    b bool
)

var c, d int = 1, 2
var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现
//g, h := 123, "hello"

func main(){
    g, h := 123, "hello"
    println(x, y, a, b, c, d, e, f, g, h)
}
```

以上实例执行结果为：

0 0 0 false 1 2 123 hello 123 hello

值类型和引用类型

> 值类型：所有像int、float、bool和string这些类型都属于值类型，使用这些类型的变量直接指向存在内存中的值，值类型的变量的值存储在栈中。当使用等号=将一个变量的值赋给另一个变量时，如 j = i ,实际上是在内存中将 i 的值进行了拷贝。可以***通过 &i 获取变量 i 的内存地址***

> 引用类型：复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。一个引用类型的变量r1存储的是r1的值所在的内存地址（数字），或内存地址中第一个字所在的位置，这个内存地址被称之为指针，这个指针实际上也被存在另外的某一个字中。当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，r2 也会受到影响。

变量作用域

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

局部变量

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

实例

```
package main

import "fmt"

func main() {
   /* 声明局部变量 */
   var a, b, c int

   /* 初始化参数 */
   a = 10
   b = 20
   c = a + b

   fmt.Printf ("结果:  a = %d, b = %d and c = %d\n", a, b, c)
}
```

全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

实例

```
package main

import "fmt"

/* 声明全局变量 */
var g int
```

```
7
8  func main() {
9
10    /* 声明局部变量 */
11    var a, b int
12
13    /* 初始化参数 */
14    a = 10
15    b = 20
16    g = a + b
17
18    fmt.Printf("结果:  a = %d, b = %d and g = %d\n", a, b, g)
19 }
```

Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。

形式参数

函数定义中的变量称为形式参数, 形式参数会作为函数的局部变量来使用。

实例

```
1  package main
2
3  import "fmt"
4
5  /* 声明全局变量 */
6  var a int = 20;
7
8  func main() {
9     /* main 函数中声明局部变量 */
10    var a int = 10
11    var b int = 20
12    var c int = 0
13
14    fmt.Printf("main()函数中 a = %d\n",  a);
15    c = sum( a, b);
```

```
   fmt.Printf("main()函数中 c = %d\n",  c);
}

/* 函数定义-两数相加 */
func sum(a, b int) int {
   fmt.Printf("sum() 函数中 a = %d\n",  a);
   fmt.Printf("sum() 函数中 b = %d\n",  b);

   return a + b;
}
```

初始化局部和全局变量

不同类型的局部和全局变量默认值为：

```
int 0
float32 0
pointer nil
```

注意事项

> 如果你声明了一个局部变量却没有在相同的代码块中使用它，会得到编译错误，go语言中要求创建的变量必须被使用。

> 全局变量是允许声明但不使用。 同一类型的多个变量可以声明在同一行,例 a, b, c = 5, 7, "abc"

> 如果你想要交换两个变量的值，则可以简单地使用 a, b = b, a。

> 空白标识符 *也被用于抛弃值，如值 5 在：* , b = 5, 7 中被抛弃。
> _ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

> 并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

Go 语言数据类型

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

数据类型：

```
布尔型
    布尔型的值只可以是常量 true 或者 false，默认为 false。
    一个简单的例子：var b bool = true。
数字类型
    整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，
    并且原生支持复数，其中位的运算采用补码。
    包括：
    uint8    无符号 8 位整型 (0 到 255)
    uint16    无符号 16 位整型 (0 到 65535)
    uint32    无符号 32 位整型 (0 到 4294967295)
    uint64    无符号 64 位整型 (0 到 18446744073709551615)
    int8    有符号 8 位整型 (-128 到 127)
    int16    有符号 16 位整型 (-32768 到 32767)
    int32    有符号 32 位整型 (-2147483648 到 2147483647)
    int64    有符号 64 位整型 (-9223372036854775808 到 9223372036854775
807)
    float32    IEEE-754 32位浮点型数
    float64    IEEE-754 64位浮点型数
    complex64    32 位实数和虚数
    complex128    64 位实数和虚数
    byte    类似 uint8
    rune    类似 int32
    uint    32 或 64 位
    int    与 uint 一样大小
    uintptr    无符号整型，用于存放一个指针
字符串类型:
    字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来
```

```
   的。
27     Go语言的字符串的字节使用UTF-8编码标识Unicode文本。
28 派生类型:
29     包括:
30     (a) 指针类型 (Pointer)
31     (b) 数组类型
32     (c) 结构化类型(struct)
33     (d) Channel 类型
34     (e) 函数类型
35     (f) 切片类型
36     (g) 接口类型 (interface)
37     (h) Map 类型
```

类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
1 type_name(expression)
```

type_name 为类型，expression 为表达式。

以下实例中将整型转化为浮点型，并计算结果，将结果赋值给浮点型变量：

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main() {
 6    var sum int = 17
 7    var count int = 5
 8    var mean float32
 9
10    mean = float32(sum)/float32(count) // 整型转化为浮点型
11    fmt.Printf("mean 的值为: %f\n",mean) // 3.400000
12 }
```

Go 语言结构、特性说明

Go 语言组成部分

当前包声明
引入其他包
函数
变量
语句 & 表达式
注释

go语言代码实例

```
1 package main
2
3 import "fmt"
4
5 func main() {
6    /* 这是我的第一个简单的程序 */
7    fmt.Println("Hello, World!")
8 }
```

以上程序的各个部分说明：

第一行代码 package main 定义了包名。你***必须在源文件中非注释的第一行指明这个文件属于哪个包***，如：package main。package main表示一个可独立执行的程序，***每个 Go 应用程序都包含一个名为 main 的包***。

下一行 import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。

下一行 func main() 是程序开始执行的函数。***main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）***。

下一行 `/*...*/`是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 `//` 开头的单行注释。多行注释也叫块注释，均已以 `/*` 开头，并以 `*/` 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。

下一行 fmt.Println(...) 可以将字符串输出到控制台，并在最后自动增加换行字符 n。 使用 fmt.Print("hello, worldn") 可以得到相同的结果。 Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。

结构详细说明

```
// 当前程序的包名
package main

// 导入其他包
import "fmt"

// 常量定义
const PI = 3.14

// 全局变量的声明和赋值
var name = "gopher"

// 一般类型声明
type newType int

// 结构的声明
type gopher struct{}

// 接口的声明
type golang interface{}

// 由main函数作为程序入口点启动
func main() {
    fmt.Println("Hello World!")
}
```

包导入方式：

通过 import 关键字单个导入：

```
import "fmt"
import "io"
```

同时导入多个：

```
import {
    "fmt",
    "io"
}
```

包调用方式

使用 `<PackageName>.<FunctionName>` 调用：

```
package 别名：
// 为fmt起别名为fmt2
import fmt2 "fmt"
```

省略调用(不建议使用)：

```
// 调用的时候只需要Println()，而不需要fmt.Println()
import . "fmt"
```

前面加个点表示省略调用，那么调用该模块里面的函数，可以不用写模块名称了：

```
import . "fmt"
func main (){
```

```
    Println("hello,world")
}
```

可见性规则

Go语言中，使用大小写来决定该常量、变量、类型、接口、结构或函数是否可以被外部包所调用。

函数名首字母小写即为 private：

```
func getId() {}
```

函数名首字母大写即为 public：

```
func Printf() {}
```

执行 Go 程序

```
$ go run xxx.go

或

$ go build
```

语言特性说明

[scode type="yellow"]
1、Go 程序是通过 package 来组织的。

2、只有 package 名称为 main 的包可以包含 main 函数。

3、一个可执行程序必须有且仅有一个 main 包，内部包含main函数，作为项目入口文件。

4、通过 import 关键字来导入其他非 main 包。

5、通过 const 关键字来进行常量的定义。

6、通过在函数体外部使用 var 关键字来进行全局变量的声明和赋值。

7、通过 type 关键字来进行结构(struct)和接口(interface)的声明。

8、通过 func 关键字来进行函数的声明。

9、一行代表一个语句结束。每个语句不需要以分号；结尾，同python

10、Go 语言的字符串拼接通过 + 实现：

11、go字符串必须用双引号包裹

[/scode]

Go 代码中会使用到的 25 个关键字或保留字：

```
1 break    default      func   interface    select
2 case    defer    go  map     struct
3 chan    else     goto   package  switch
4 const   fallthrough      if   range  type
5 continue     for     import  return     var
```

Go 语言还有 36 个预定义标识符：

```
1 append  bool    byte    cap  close   complex    complex64    complex1
  28  uint16
2 copy     false  float32  float64    imag    int     int8     int16  u
  int32
3 int32    int64   iota    len    make     new    nil     panic    uint
  64
4 print    println    real    recover     string   true    uint    uint
  8   uintptr
```

Go 语言环境安装

UNIX/Linux/Mac OS X, 和 FreeBSD 下安装

> 以下介绍了在UNIX/Linux/Mac OS X, 和 FreeBSD系统下使用源码安装方法：

```
1、下载源码包：go1.4.linux-amd64.tar.gz。

安装包下载地址：  https://golang.org/dl/

2、将下载的源码包解压至 /usr/local目录。

tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz

3、设置 $GOPATH & $GOROOT
vim ~/.bash_profile
在文件最后添加
export GOPATH="$HOME/alidata/www/go" # 全局工作目录
export GOROOT="/usr/local/go" # go语言工作链目录
保存退出

4、将 $GOPATH & $GOROOT 目录添加至PATH环境变量,方便之后调用生成的GO程序

export PATH=$PATH:$GOROOT/bin:$GOPATH/bin

5、使生效：
source ~/.bash_profile

6、在终端执行 go version 查看go语言版本
  执行 go env 查看与go语言相关的环境变量

7、工作目录下要求创建三个目录，src、pkg、bin，其中所有源码都必须放在src目录下
  mkdir src pkg bin
```

MAC 系统下你可以使用 .pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

MAC 下可通过 brew 安装

```
brew install go

或者

brew install golang
```

beego 控制器使用

```
package controllers

import (
        "github.com/astaxie/beego"
)

type MainController struct {
        beego.Controller
}

func (this *MainController) Get() {
        this.Data["Website"] = "beego.me"
        this.Data["Email"] = "astaxie@gmail.com"
        this.TplName = "index.tpl"
}
```

上面的代码显示首先我们声明了一个控制器 MainController，这个控制器里面内嵌了 beego.Controller，这就是 Go 的嵌入方式，也就是 MainController 自动拥有了所有 beego.Controller 的方法。

而 beego.Controller 拥有很多方法，其中包括 Init、Prepare、Post、Get、Delete、Head 等方法。我们可以通过重写的方式来实现这些方法，而我们上面的代码就是重写了 Get 方法。

我们先前介绍过 beego 是一个 RESTful 的框架，所以我们的请求默认是执行对应 req.Method 的方法。例如浏览器的是 GET 请求，那么默认就会执行 MainController 下的 Get 方法。这样我们上面的 Get 方法就会被执行到，这样就进入了我们的逻辑处理。

里面的代码是需要执行的逻辑，这里只是简单的输出数据，我们可以通过各种方式获取数据，然后赋值到 this.Data 中，这是一个用来存储输出数据的 map，可以赋值任意类型的值，这里我们只是简单举例输出两个字符串。

最后一个就是需要去渲染的模板，this.TplName 就是需要渲染的模板，这里指定了 index.tpl，如果用户不设置该参数，那么默认会去到模板目录的 Controller/<方法名>.tpl 查找，例如上面的方法会去 maincontroller/get.tpl (文件、文件夹必须小写)。

用户设置了模板之后系统会自动的调用 Render 函数（这个函数是在 beego.Controller 中实现的），所以无需用户自己来调用渲染。

当然也可以不使用模版，直接用 this.Ctx.WriteString 输出字符串，如：

```
func (this *MainController) Get() {
        this.Ctx.WriteString("hello")
}
```

beego 路由设置

beego项目中 main 函数是入口函数，在这里我们看到引入了一个包 _ "quickstart/routers",这个包只引入执行了里面的 init 函数，那么让我们看看这个里面做了什么事情：

```
1 package routers
2
3 import (
4     "quickstart/controllers"
5     "github.com/astaxie/beego"
6 )
7
8 func init() {
9     beego.Router("/", &controllers.MainController{})
10 }
```

路由包里面我们看到执行了路由注册 beego.Router, 这个函数的功能是映射 URL 到 controller，第一个参数是 URL（用户请求的地址），这里我们注册的是 /，也就是我们访问的不带任何参数的 URL，第二个参数是对应的 Controller，也就是我们即将把请求分发到那个控制器来执行相应的逻辑，我们可以执行类似的方式注册如下路由：

```
1 beego.Router("/user", &controllers.UserController{})
```

这样用户就可以通过访问 /user 去执行 UserController 的逻辑。这就是我们所谓的路由，

beego 安装使用

beego 的安装

```
go get github.com/astaxie/beego
```

bee 工具的安装

```
go get github.com/beego/bee
```

注意： 安装完之后，bee 可执行文件默认存放在 $GOPATH/bin 里面，所以您需要把 $GOPATH/bin 添加到您的环境变量中，才可以进行下一步。

bee 工具命令详解

我们在命令行输入 bee，可以看到如下的信息：

```
Bee is a Fast and Flexible tool for managing your Beego Web Application.

Usage:

    bee command [arguments]

The commands are:

    version     show the bee & beego version
    migrate     run database migrations
    api         create an api application base on beego framework
    bale        packs non-Go files to Go source files
    new         create an application base on beego framework
```

```
    run         run the app which can hot compile
    pack        compress an beego project
    fix         Fixes your application by making it compatible with
 newer versions of Beego
    dlv         Start a debugging session using Delve
    dockerize   Generates a Dockerfile for your Beego application
    generate    Source code generator
    hprose      Creates an RPC application based on Hprose and Beego
frameworks
    new         Creates a Beego application
    pack        Compresses a Beego application into a single file
    rs          Run customized scripts
    run         Run the application by starting a local development
 server
    server      serving static content over HTTP on port

Use bee help [command] for more information about a command.
```

new 命令

new 命令是新建一个 Web 项目，我们在命令行下执行 bee new <项目名> 就可以创建一个新的项目。但是注意该命令必须在 $GOPATH/src 下执行。最后会在 $GOPATH/src 相应目录下生成如下目录结构的项目：

api 命令

上面的 new 命令是用来新建 Web 项目，不过很多用户使用 beego 来开发 API 应用。所以这个 api 命令就是用来创建 API 应用的.

同时，该命令还支持一些自定义参数自动连接数据库创建相关 model 和 controller:
bee api [appname] [-tables=""] [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
如果 conn 参数为空则创建一个示例项目，否则将基于链接信息链接数据库创建项目。

run 命令

我们在开发 Go 项目的时候最大的问题是经常需要自己手动去编译再运行，bee run 命令是监控 beego 的项目，通过 fsnotify监控文件系统。但是注意该命令必须在 $GOPATH/src/appname 下执行。

这样我们在开发过程中就可以实时的看到项目修改之后的效果.

pack 命令

pack 目录用来发布应用的时候打包，会把项目打包成 zip 包，这样我们部署的时候直接把打包之后的项目上传，解压就可以部署了.

bale 命令

这个命令目前仅限内部使用，具体实现方案未完善，主要用来压缩所有的静态文件变成一个变量申明文件，全部编译到二进制文件里面，用户发布的时候携带静态文件，包括 js、css、img 和 views。最后在启动运行时进行非覆盖式的自解压。

version 命令

这个命令是动态获取 bee、beego 和 Go 的版本，这样一旦用户出现错误，可以通过该命令来查看当前的版本

generate 命令

这个命令是用来自动化的生成代码的，包含了从数据库一键生成 model，还包含了 scaffold 的，通过这个命令，让大家开发代码不再慢

```
bee generate scaffold [scaffoldname] [-fields=""] [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    The generate scaffold command will do a number of things for you.
    -fields: a list of table fields. Format: field:type, ...
    -driver: [mysql | postgres | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test
    example: bee generate scaffold post -fields="title:string,body:text"

bee generate model [modelname] [-fields=""]
    generate RESTful model based on fields
```

```
    -fields: a list of table fields. Format: field:type, ...

bee generate controller [controllerfile]
    generate RESTful controllers

bee generate view [viewpath]
    generate CRUD view in viewpath

bee generate migration [migrationfile] [-fields=""]
    generate migration file for making database schema update
    -fields: a list of table fields. Format: field:type, ...

bee generate docs
    generate swagger doc file

bee generate test [routerfile]
    generate testcase

bee generate appcode [-tables=""] [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"] [-level=3]
    generate appcode based on an existing database
    -tables: a list of table names separated by ',', default is empty, indicating all tables
    -driver: [mysql | postgres | sqlite], the default is mysql
    -conn:   the connection string used by the driver.
             default for mysql:    root:@tcp(127.0.0.1:3306)/test
             default for postgres: postgres://postgres:postgres@127.0.0.1:5432/postgres
    -level:  [1 | 2 | 3], 1 = models; 2 = models,controllers; 3 = models,controllers,router
```

migrate 命令

这个命令是应用的数据库迁移命令，主要是用来每次应用升级，降级的SQL管理。

```
bee migrate [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    run all outstanding migrations
```

```
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test

bee migrate rollback [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback the last migration operation
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test

bee migrate reset [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback all migrations
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test

bee migrate refresh [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback all migrations and run them all again
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test
```

dockerize 命令

这个命令可以通过生成Dockerfile文件来实现docker化你的应用。

例子:
生成一个以1.6.4版本Go环境为基础镜像的Dockerfile,并暴露9000端口:

```
$ bee dockerize -image="library/golang:1.6.4" -expose=9000
```

js判断移动端换将并切换域名代码

```
<script type="text/javascript">
var mobileAgent = new Array("iphone", "ipod", "ipad", "android", "mobile", "blackberry", "webos", "incognito", "webmate", "bada", "nokia", "lg", "ucweb", "skyfire");
var browser = navigator.userAgent.toLowerCase();
for (var i=0; i<mobileAgent.length; i++){
    if (browser.indexOf(mobileAgent[i])!=-1){
                // 替换域名，  也可以做其他操作
        window.location.href = window.location.href.replace('www', 'm');
        break;
    }
}
</script>
```

在html页面中引入css的几种方式

css主要有三种引用方法：

引用外部CSS文件

用标签引用外部的css文件，将样式引用到html文档来。

```
<link rel="stylesheet" type="text/css" href="style.css">
```

内部定义Style样式

在html文档中在<style>标签里面写的css样式

```
<style>
 bdoy{font-size:14px;}
</style>
```

内联样式

在标签中使用style属性将当前的标签样式改变。

```
<div style="color:red"></div>
```

优先级

内联样式 > 内部定义Style样式 > 引用外部CSS文件

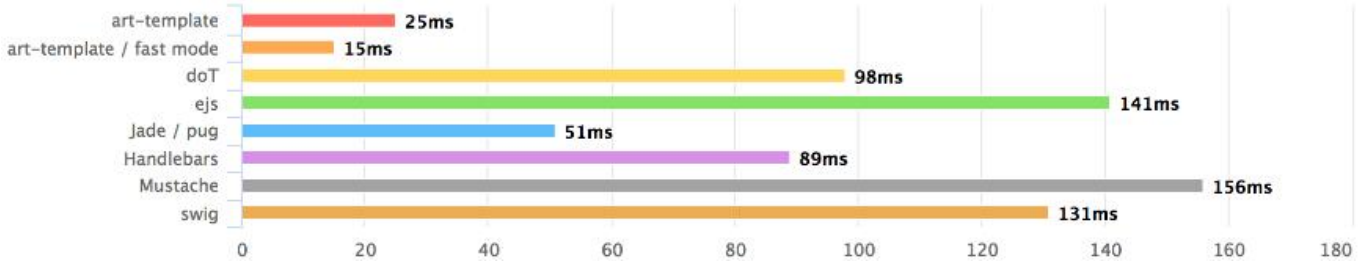
前端模板引擎 artTemplate介绍

对于服务端开发人员来说在做前后端分离页面时往往会用js拼接html,但这样可读性不大好。在服务端渲染页面时我们会使用模板引擎像smarty等。最近查了一下前端也是有很多模板引擎的，可以像smarty一样渲染页面，下面介绍前端art-template模板引擎的使用：

GitHub地址: https://github.com/aui/art-template

项目文档地址：https://aui.github.io/art-template/zh-cn/docs/

特性



性能卓越，执行速度通常是 Mustache 与 tmpl 的 20 多倍

支持运行时调试，可精确定位异常模板所在语句

对 NodeJS Express 友好支持

安全，默认对输出进行转义、在沙箱中运行编译后的代码（Node版本可以安全执行用户上传的模板）

支持include语句，可在浏览器端实现按路径加载模板

支持预编译，可将模板转换成为非常精简的 js 文件

模板语句简洁，无需前缀引用数据

支持所有流行的浏览器

arttemplate模板引擎具体的使用。

1、引入template.js

2、编写前端的模版

```
// 数据写入的位置
<div id="content">
    // js模板样式
    <script id="test" type="text/html">
        <h1>{{title}}</h1>
        <ul>
            {{each list as value i}}
                <li>索引 {{i + 1}} ：{{value}}</li>
            {{/each}}
        </ul>
    </script>
</div>
```

3、渲染模板

```
var data = {
    title: '标签',
    list: ['文艺', '博客', '摄影', '电影', '民谣', '旅行', '吉他']
};
var html = template('test', data);
document.getElementById('content').innerHTML = html;
```

4、不编码输出：
{{#content}}
编码可以防止数据中含有 HTML 字符串，避免引起 XSS 攻击。

5、条件表达式

```
{{if admin}}
    <p>admin</p>
{{else if code > 0}}
```

```
    <p>master</p>
{{else}}
    <p>error!</p>
{{/if}}
```

6、遍历表达式

无论数组或者对象都可以用 each 进行遍历。

```
{{each list as value index}}
    <li>{{index}} - {{value.user}}</li>
{{/each}}
```

亦可以被简写：

```
{{each list}}
    <li>{{$index}} - {{$value.user}}</li>
{{/each}}
```

7、模板包含表达式

用于嵌入子模板。

```
{{include 'template_name'}}
```

子模板默认共享当前数据，亦可以指定数据：

```
{{include 'template_name' news_list}}
```

8、辅助方法

使用template.helper(name, callback)注册公用辅助方法：

```
template.helper('dateFormat', function (date, format) {
    // ..
    return value;
});
```

模板中使用的方式：

```
{{time | dateFormat:'yyyy-MM-dd hh:mm:ss'}}
```

支持传入参数与嵌套使用：

```
{{time | say:'cd' | ubb | link}}
```

到这里arttemplatejs模板引擎最基本的使用方法就介绍完毕了。要是想更多的了解arttemplatejs模板引擎的使用方法，请看官网的介绍。

如何正确的使用arttemplatejs模板引擎。

1、前后端分离页面，ajax请求数据赋值

2、后端渲染时，替换后端模板引擎，直接将数据返回到页面js中，到浏览器时再渲染。减少了后端渲染的性能消耗。

使用实例：

```

<!DOCTYPE html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,user-scalable=0">
    <title>使用artTemplate</title>
    <link rel="stylesheet" href="http://me.52fhy.com/demo/jstemp/static/weui.css"/>
```

```
    <script src="http://me.52fhy.com/demo/jstemp/static/zepto.min.js"></script>

    <!-- 首先引入template-web.js文件 artTemplate.js-->
    <script src="https://unpkg.com/art-template@4.13.2/lib/template-web.js"></script>
</head>

<body>
<div class="weui_panel weui_panel_access" id="wenzhang_list">
    <!--模板: 注意这里模板与前面的示例不一样了，直接使用一个type="text/html"的script标签存放模板-->
    <script id="js-tmp" type="text/html">
        <div class="weui_panel_hd">文章列表</div>
        <div class="weui_panel_bd js-blog-list">
            {{each list}}
            <div class="weui_media_box weui_media_text">
                <a href="{{ $value.url }}" class="" target="_blank">
                    <h4 class="weui_media_title">{{ $value.title }}</h4>
                </a>
                <p class="weui_media_desc">{{ $value.desc }}</p>
            </div>
            {{/each}}
        </div>
    </script>
    <!--/模板-->
</div>

<script type="text/javascript">

    $(function () {
        //根据id编译模板在进行渲染
        var data = {
            "list": [
                {
                    "title": "[置顶]Laravel5.0学习--01 入门",
                    "url": "http://www.cnblogs.com/52fhy/p/5271447.html",
                    "desc": "摘要: 本文以laravel5.0.22为例。"
```

```
                },
                {
                    "title": "[置顶]Laravel5.0学习--01 入门",
                    "url": "http://www.cnblogs.com/52fhy/p/5271447.html",
                    "desc": "摘要: 本文以laravel5.0.22为例。 "
                }]
        }
        var html = template('js-tmp', data);
        //填充渲染结果到content
        document.getElementById('wenzhang_list').innerHTML = html;
    });
</script>
</body>
```

支持条件和循环语法

设计模式之： 观察者模式

概念

一个对象通过添加一个方法（该方法允许另一个对象，即观察者 注册自己）使本身变得可观察。当可观察的对象更改时，它会将消息发送到已注册的观察者。这些观察者使用该信息执行的操作与可观察的对象无关。结果是对象可以相互对话，而不必了解原因。观察者模式是一种事件系统，意味着这一模式允许某个类观察另一个类的状态，当被观察的类状态发生改变的时候，观察类可以收到通知并且做出相应的动作;观察者模式为您提供了避免组件之间紧密耦。

解决的问题

在我们的开发过程中，应该都或多或少的碰到过改动其中一部分代码会引起其他一连串改变的问题，显然想要完全避免这种情况不太可能，但我们也应答尽量减少对其他组件的依赖，而观察者模式就是为了解决这个问题。

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新【GOF95】 又称为发布-订阅（Publish-Subscribe）模式、模型-视图（Model-View）模式、源-监听（Source-Listener）模式、或从属者(Dependents)模式

主要角色

1.抽象被观察者角色：主题角色将所有对观察者对象的引用保存在一个集合中，每个主题可以有任意多个观察者。 抽象主题提供了增加和删除观察者对象的接口。
2.抽象观察者角色：为所有的具体观察者定义一个接口，在观察的主题发生改变时更新自己。
3.具体被观察者角色：存储相关状态到具体观察者对象，当具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个具体子类实现。
4.具体观察者角色（多个）：存储一个具体主题对象，存储相关状态，实现抽象观察者角色所要求的更新接口，以使得其自身状态和主题的状态保持一致。

优点：

1.观察者和主题之间的耦合度较小；
2.支持广播通信；

缺点：

由于观察者并不知道其它观察者的存在，它可能对改变目标的最终代价一无所知。这可能会引起意外的更新。

适用场景

当一个抽象模型有两个方面，其中一个方面依赖于另一个方面。
当对一个对象的改变需要同时改变其它对象，而不知道具体有多少个对象待改变。
当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换句话说，你不希望这些对象是紧密耦合的。

理解

从面向过程的角度来看，首先是观察者向被观察者注册，注册完之后，被观察者再通知观察者做出相应的操作，整个事情就完了。

被观察者提供注册和通知的接口，观察者提供自身操作的接口。（这些观察者拥有一个同一个接口。）观察者利用被观察者的接口向被观察者注册，而被观察者利用观察者的接口通知观察者。耦合度相当之低。

如何实现观察者注册？通过前面的注册者模式很容易给我们提供思路，把这些对象加到一棵注册树上就好了嘛。如何通知？这就更简单了，对注册树进行遍历，让每个对象实现其接口提供的操作。

实例：

假如一个小贩， 他把产品的价格提升了， 不同的消费者会对此产生不同的反应。一般的编程模式无非是获取提升的价格，然后获取所有的消费者，再循环每个消费者， 不同的消费者根据价格涨幅做出决定，如果消费者的类型有限，因而进行的判断也不多，这种无可厚非，但如果有更多的类型的消费者加入进来， 那这个代码就变得臃肿且难以维护， 因为要不停的往里面加入判断代码，这个时候其实就适用观察者模式了

思路：

观察者模式分为两个角色， 观察者(observer)和被观察者(observables), 先在被观察者注册一系列的被观察者， 在被观察者发生变化的时候，通知观察者，进而观察者自动进行更新，这种一对多的关系就像你是一个小贩(被观察者)，卖东西，有很多人(观察者)在买你的东西，假如你要升价， 这个时候所有的消费

者(观察者)可以决定继续买，还是不买，还是其他动作，作为小贩(被观察者)的你只需要把价格增加，继而通知一下，而不用去管其他人(观察者)的动作。

```php
<?php

//先定义一个被观察者的接口，这个接口要实现注册观察者，删除观察者和通知的功能。
interface Observables
{
    public function attach(observer $ob);

    public function detach(observer $ob);

    public function notify();
}

class Saler implements Observables
{
    protected $obs = [];       //把观察者保存在这里
    protected $range = 0;

    public function attach(Observer $ob)
    {
        $this->obs[] = $ob;
    }

    public function detach(Observer $ob)
    {
        foreach ($this->obs as $o) {
            if ($o != $ob)
                $this->obs[] = $o;
        }
    }

    public function notify()
    {
        foreach ($this->obs as $o) {
            $o->doActor($this);
        }
    }
```

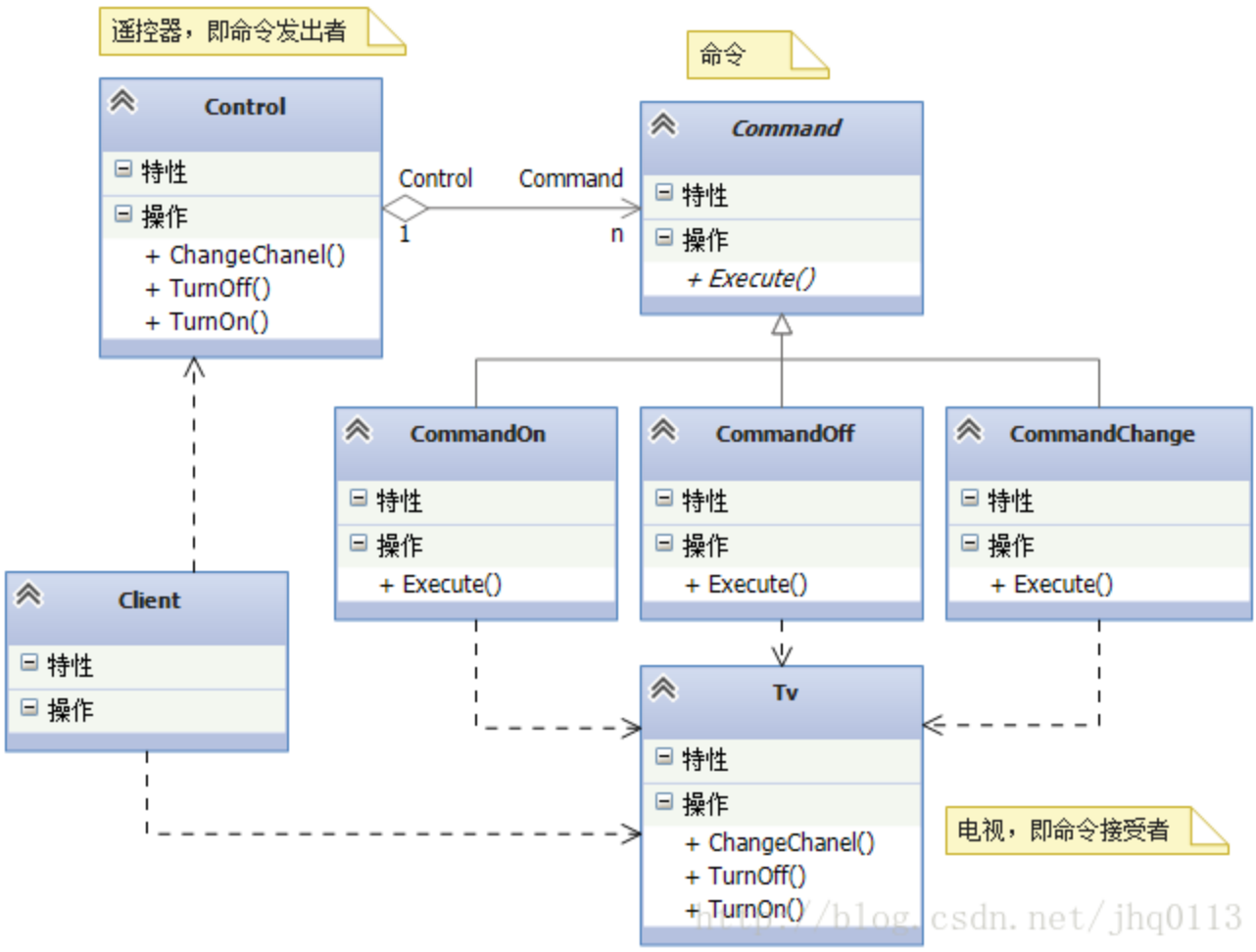
```
37
38      public function increPrice($range)
39      {
40          $this->range = $range;
41      }
42
43      public function getAddRange()
44      {
45          return $this->range;
46      }
47  }
48
49
50  //定义一个观察者的接口，这个接口要有一个在被通知的时候都要实现的方法
51  interface Observer
52  {
53      public function doActor(Observables $obv);
54  }
55
56
57  class PoorBuyer implements Observer
58  {
59      //PoorBurer的做法
60      public function doActor(observables $obv)
61      {
62          if ($obv->getAddRange() > 10)
63              echo '不买了.<br />' . PHP_EOL;
64          else
65              echo '还行，买一点吧.<br />' . PHP_EOL;
66      }
67  }
68
69  class RichBuyer implements Observer
70  {
71      //RichBuyer的做法
72      public function doActor(observables $obv)
73      {
74          echo '你再涨我也不怕，咱不差钱.<br />' . PHP_EOL;
75      }
76  }
```

```php

$saler = new Saler();  //小贩(被观察者)
$saler->attach(new PoorBuyer()); //注册一个低收入的消费者(观察者)
$saler->attach(new RichBuyer()); //注册一个高收入的消费者(观察者)
$saler->notify(); //通知

$saler->increPrice(11);  //涨价
$saler->notify();  //通知
```

设计模式之：命令模式

命令模式：

在软件系统中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合，比如要对行为进行“记录、撤销/重做、事务”等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，将“行为请求者”与“行为实现者”解耦，实现二者之间的松耦合。这就是命令模式。
命令模式的本质是对命令进行封装，将发出命令的责任和执行命令的责任分割开，实现二者之间的松耦合。



角色分析：

```
1 抽象命令：定义命令的接口，声明执行的方法。
2
```

具体命令：命令接口实现对象，是“虚”的实现；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。

命令接收者：接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

控制者：要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。
　　这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。

实例：

```php
<?php

/**
 * 电视机是请求的接收者，
 *遥控器上有一些按钮，不同的按钮对应电视机的不同操作。
 *抽象命令角色由一个命令接口来扮演，有三个具体的命令类实现了抽象命令接口，
 *这三个具体命令类分别代表三种操作：打开电视机、关闭电视机和切换频道。
 *显然，电视机遥控器就是一个典型的命令模式应用实例。
 */

/**
 * 命令接收者
 * Class Tv
 */
class Tv
{
    public $curr_channel = 0;

    /**
     * 打开电视机
     */
    public function turnOn()
    {
        echo "The television is on." . PHP_EOL;
    }
```

```
26
27      /**
28       * 关闭电视机
29       */
30      public function turnOff()
31      {
32          echo "The television is off." . PHP_EOL;
33      }
34
35      /**
36       * 切换频道
37       * @param $channel    频道
38       */
39      public function turnChannel($channel)
40      {
41          $this->curr_channel = $channel;
42          echo "This TV Channel is " . $this->curr_channel . PHP_EOL;
43      }
44  }
45
46  /**
47   * 执行命令接口
48   * Interface ICommand
49   */
50  interface ICommand
51  {
52      public function execute();
53
54      /**
55       * 撤销接口
56       * @return mixed
57       */
58      public function undo();
59  }
60
61  /**
62   * 开机命令
63   * Class CommandOn
64   */
65  class CommandOn implements ICommand
```

```
66 {
67     private $tv;
68 
69     public function __construct($tv)
70     {
71         $this->tv = $tv;
72     }
73 
74     public function execute()
75     {
76         $this->tv->turnOn();
77     }
78 
79     public function undo()
80     {
81         $this->tv->turnOff();
82     }
83 }
84 
85 /**
86  * 关机命令
87  * Class CommandOn
88  */
89 class CommandOff implements ICommand
90 {
91     private $tv;
92 
93     public function __construct($tv)
94     {
95         $this->tv = $tv;
96     }
97 
98     public function execute()
99     {
100         $this->tv->turnOff();
101     }
102 
103     public function undo()
104     {
105         $this->tv->turnOn();
```

```
106     }
107 }
108 
109 /**
110  * 切换频道命令
111  * Class CommandOn
112  */
113 class CommandChannel implements ICommand
114 {
115     private $tv;
116     private $channel;
117 
118     public function __construct($tv, $channel)
119     {
120         $this->tv = $tv;
121         $this->channel = $channel;
122     }
123 
124     public function execute()
125     {
126         $this->tv->turnChannel($this->channel);
127     }
128 
129     public function undo()
130     {
131 
132     }
133 }
134 
135 /**
136  * 遥控器
137  * Class Control
138  */
139 class Control
140 {
141     private $_onCommand;
142     private $_offCommand;
143     private $_changeChannel;
144 
145     public function __construct($on, $off, $channel)
```

```
146     {
147         $this->_onCommand = $on;
148         $this->_offCommand = $off;
149         $this->_changeChannel = $channel;
150     }
151 
152     public function turnOn()
153     {
154         $this->_onCommand->execute();
155     }
156 
157     public function turnOff()
158     {
159         $this->_offCommand->execute();
160     }
161 
162     public function changeChannel()
163     {
164         $this->_changeChannel->execute();
165     }
166 }
167 
168 
169 //测试代码
170 // 命令接收者
171 $myTv = new Tv();
172 // 开机命令
173 $on = new CommandOn($myTv);
174 // 关机命令
175 $off = new CommandOff($myTv);
176 
177 // 频道切换命令
178 $channel = new CommandChannel($myTv, 2);
179 // 命令控制对象
180 $control = new Control($on, $off, $channel);
181 // 开机
182 $control->turnOn();
183 // 切换频道
184 $control->changeChannel();
185 // 关机
```

```
186 $control->turnOff();
```

适用场景：

1.系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。

2.系统需要在不同的时间指定请求、将请求排队和执行请求。

3.系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作(定义一个相反的命令即可)。

4.系统需要将一组操作组合在一起，即支持宏命令。

优点

1.降低对象之间的耦合度。

2.新的命令可以很容易地加入到系统中。

3.可以比较容易地设计一个组合命令。

4.调用同一方法实现不同的功能

缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。