语言

2017年8月21日 9:54

1. 指针和引用区别

指针和引用的定义和性质区别：

(1)指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。如：

int a=1;int *p=&a;

int a=1;int &b=a;

上面定义了一个整形变量和一个指针变量p，该指针变量指向a的存储单元，即p的值是a存储单元的地址。

而下面2句定义了一个整形变量a和这个整形a的引用b，事实上a和b是同一个东西，在内存占有同一个存储单元。

(2)可以有const指针，但是没有const引用；

(3)指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a是不合法的）

(4)指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；

(5)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。

(6)"sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；

(7)指针和引用的自增(++)运算意义不一样；

来自 <http://www.cnblogs.com/dolphin0520/archive/2011/04/03/2004869.html>

2. 堆和栈的区别

堆栈空间分配

栈（操作系统）：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表

堆栈缓存方式

栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放。堆则是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

堆栈数据结构区别

堆（数据结构）：堆可以被看成是一棵树，如：堆排序。
栈（数据结构）：一种先进后出的数据结构。

来自 <http://www.cnblogs.com/mysticCoder/p/4921724.html>

一个由C/C++编译的程序占用的内存分为以下几个部分
1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其
操作方式类似于**数据结构**中的栈。
2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回
收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的
全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另
一块区域。 - 程序结束后由系统释放。
4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放
5、程序代码区—存放函数体的二进制代码。

二、例子程序
这是一个前辈写的，非常详细

```
//main.cpp
int a = 0; 全局初始化区
char *p1; 全局未初始化区
main()
{
int b; 栈
char s[] = "abc"; 栈
char *p2; 栈
char *p3 = "123456"; 123456/0在常量区，p3在栈上。
static int c =0； 全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
```

分配得来得10和20字节的区域就在堆区。
strcpy(p1, "123456"); 123456/0放在常量区，编译器可能会将它与p3所指向的"123456"
优化成一个地方。

}

二、堆和栈的理论知识

2.1申请方式

stack:

由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空

间

heap:

需要程序员自己申请，并指明大小，在c中malloc函数

如p1 = (char *)malloc(10);

在C++中用new运算符

如p2 = new char[10];

但是注意p1、p2本身是在栈中的。

2.2

申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢

出。

堆：首先应该知道**操作系统**有一个记录空闲内存地址的链表，当系统收到程序的申请时，

会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表

中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的

首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。

另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部

分重新放入空闲链表中。

2.3申请大小的限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意

思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有

的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间

时，将
提示overflow。因此，能从栈获得的空间较小。
堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储
的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小
受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

2.4申请效率的比较：
栈由系统自动分配，速度较快。但程序员是无法控制的。
堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.
另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是
直接在进程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。

2.5堆和栈中的存储内容
栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可
执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈
的，然后是函数中的局部变量。注意静态变量是不入栈的。
当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地
址，也就是主函数中的下一条指令，程序由该点继续运行。
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

2.6存取效率的比较

```
char  s1[]  =  "aaaaaaaaaaaaaa";
char  *s2  =  "bbbbbbbbbbbbbbbbb";
```

aaaaaaaaaa是在运行时刻赋值的；
而bbbbbbbbbb是在编译时就确定的；
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。
比如：

```
#include
void  main()
{
char  a  =  1;
char  c[]  =  "1234567890";
char  *p  ="1234567890";
a  =  c[1];
a  =  p[1];
return;
}
```

对应的汇编代码

```
10:  a  =  c[1];
00401067  8A  4D  F1  mov  cl,byte  ptr  [ebp-0Fh]
0040106A  88  4D  FC  mov  byte  ptr  [ebp-4],cl
11:  a  =  p[1];
0040106D  8B  55  EC  mov  edx,dword  ptr  [ebp-14h]
00401070  8A  42  01  mov  al,byte  ptr  [edx+1]
00401073  88  45  FC  mov  byte  ptr  [ebp-4],al
```

第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到

edx中，再根据edx读取字符，显然慢了。

2.7小结：

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就

走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自

由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由

度大。

来自 <http://blog.csdn.net/hairetz/article/details/4141043/>

? **3. sort的实现机制**

http://blog.csdn.net/ohmygirl/article/details/42145895

? **4. 八大排序算法**

5. \0 的含义

'\0'的含义

6. **get和post都可以取：@_REQUEST**
7. **== 和=== 区别**

比如你一个函数会返回这几种情况：

1、大于0的数

2、小于0的数

3、等于0的数（也就是0啦）

4、False（失败时）

这时候如果你想捕获失败的情况，你就必须用===，而不能用==

因为==除了会匹配第4种情况外，还会匹配第3种情况，因为0也是假！

三个等号代表比较对象的类型也要一致。两个等号表示只要值相等就满足条件。

再来补充一些：

$a='2';//字符型2

$b=2;//数值型2

$a==$b,是对的，都是2

$a===$b，是不对的，因为$a是字符型$b是数值型，值虽一样，但类型不一样。

还有就是“linvo1986 - 六级”说的那种“0”了。

来自 <http://www.jb51.net/article/40338.htm>

8. **字符集**
9. **PHP的垃圾回收机制**

PHP是一门托管型语言，在PHP编程中程序员不需要手工处理内存资源的分配与释放(使用C编写PHP或Zend扩展除外)，这就意味着PHP本身实现了垃圾回收机制(Garbage Collection)。现在如果去PHP官方网站(php.net)可以看到，目前PHP5的两个分支版本PHP5.2和PHP5.3是分别更新的，这是因为许多项目仍然使用5.2版本的PHP，而5.3版本对5.2并不是完全兼容。PHP5.3在PHP5.2的基础上做了诸多改进，其中垃圾回收算法就属于一个比较大的改变。本文将分别讨论PHP5.2和PHP5.3的垃圾回收机制，并讨论这种演化和改进对于程序员编写PHP的影响以及要注意的问题。

PHP变量及关联内存对象的内部表示

垃圾回收说到底是对变量及其所关联内存对象的操作，所以在讨论PHP的垃圾回收机制之前，先简要介绍PHP中变量及其内存对象的内部表示(其C源代码中的表示)。

PHP官方文档中将PHP中的变量划分为两类：标量类型和复杂类型。标量类型包括布尔型、整型、浮点型和字符串;复杂类型包括数组、对象和资源;还有一个NULL比较特殊，它不划分为任何类型，而是单独成为一类。

所有这些类型，在PHP内部统一用一个叫做zval的结构表示，在PHP源代码中这个结构名称为“_zval_struct”。zval的具体定义在PHP源代码的“Zend/zend.h”文件中，下面是相关代码的摘录。

```
1. typedef union _zvalue_value {
```

```
    long lval;                /* long value */
    double dval;              /* double value */
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;            /* hash table value */
    zend_object_value obj;
} zvalue_value;

struct _zval_struct {
    /* Variable information */
    zvalue_value value;
/* value */
    zend_uint refcount__gc;
    zend_uchar type;    /* active type */
    zend_uchar is_ref__gc;
};
```

其中联合体“_zvalue_value”用于表示PHP中所有变量的值，这里之所以使用union，是因为一个zval在一个时刻只能表示一种类型的变量。可以看到_zvalue_value中只有5个字段，但是PHP中算上NULL有8种数据类型，那么PHP内部是如何用5个字段表示8种类型呢?这算是PHP设计比较巧妙的一个地方，它通过复用字段达到了减少字段的目的。例如，在PHP内部布尔型、整型及资源(只要存储资源的标识符即可)都是通过lval字段存储的;dval用于存储浮点型;str存储字符串;ht存储数组(注意PHP中的数组其实是哈希表);而obj存储对象类型;如果所有字段全部置为0或NULL则表示PHP中的NULL，这样就达到了用5个字段存储8种类型的值。

而当前zval中的value(value的类型即是_zvalue_value)到底表示那种类型，则由“_zval_struct”中的type确定。_zval_struct即是zval在C语言中的具体实现，每个zval表示一个变量的内存对象。除了value和type，可以看到_zval_struct中还有两个字段refcount__gc和is_ref__gc，从其后缀就可以断定这两个家伙与垃圾回收有关。没错，PHP的垃圾回收全靠这俩字段了。其中refcount__gc表示当前有几个变量引用此zval，而is_ref__gc表示当前zval是否被按引用引用，这话听起来很拗口，这和PHP中zval的“Write-On-Copy”机制有关，由于这个话题不是本文重点，因此这里不再详述，读者只需记住refcount__gc这个字段的作用即可。

PHP5.2中的垃圾回收算法——Reference Counting

PHP5.2中使用的内存回收算法是大名鼎鼎的Reference Counting，这个算法中文翻译叫做“引用计数”，其思想非常直观和简洁：为每个内存对象分配一个计数器，当一个内存对象建立时计数器初始化为1(因此此时总是有一个变量引用此对象)，以后每有一个新变量引用此内存对象，则计数器加1，而每当减少一个引用此内存对象的变量则计数器减1，当垃圾回收机制运作的时候，将所有计数器为0的内存对象销毁并回收其占用的内存。而PHP中内存对象就是zval，而计数器就是refcount__gc。

例如下面一段PHP代码演示了PHP5.2计数器的工作原理(计数器值通过xdebug得到)：

```
<?php

$val1 = 100; //zval(val1).refcount_gc = 1;
$val2 = $val1; //zval(val1).refcount_gc = 2,zval(val2).refcount_gc = 2(因为是Write on copy，当前val2与val1共同引用一个zval)
$val2 = 200; //zval(val1).refcount_gc = 1,zval(val2).refcount_gc = 1(此处val2新建了一个zval)
unset($val1); //zval(val1).refcount_gc = 0($val1引用的zval再也不可用，会被GC回收)

?>
```

Reference Counting简单直观，实现方便，但却存在一个致命的缺陷，就是容易造成内存泄露。很多朋友可能已经意识到了，如果存在循环引用，那么Reference Counting就可能导致内存泄露。例如下面的代码：

```
<?php

$a = array();
$a[] = & $a;
unset($a);

?>
```

这段代码首先建立了数组a，然后让a的第一个元素按引用指向a，这时a的zval的refcount就变为2，然后我们销毁变量a，此时a最初指向的zval的refcount为1，但是我们再也没有办法对其进行操作，因为其形成了一个循环自引用，如下图所示：

其中灰色部分表示已经不复存在。由于a之前指向的zval的refcount为1(被其HashTable的第一个元素引用)，这个zval就不会被GC销毁，这部分内存就泄露了。

这里特别要指出的是，PHP是通过符号表(Symbol Table)存储变量符号的，全局有一个符号表，而每个复杂类型如数组或对象有自己的符号表，因此上面代码中，a和a[0]是两个符号，但是a储存在全局符号表中，而a[0]储存在数组本身的符号表中，且这里a和a[0]引用同一个zval(当然符号a后来被销毁了)。希望读者朋友注意分清符号(Symbol)的zval的关系。

在PHP只用于做动态页面脚本时，这种泄露也许不是很要紧，因为动态页面脚本的生命周期很短，PHP会保证当脚本执行完毕后，释放其所有资源。但是PHP发展到目前已经不仅仅用作动态页面脚本这么简单，如果将PHP用在生命周期较长的场景中，例如自动化测试脚本或deamon进程，那么经过多次循环后积累下来的内存泄露可能就会很严重。这并不是我在耸人听闻，我曾经实习过的一个公司就通过PHP写的deamon进程来与数据存储服务器交互。

由于Reference Counting的这个缺陷，PHP5.3改进了垃圾回收算法。

PHP5.3中的垃圾回收算法——Concurrent Cycle Collection in Reference Counted Systems

PHP5.3的垃圾回收算法仍然以引用计数为基础，但是不再是使用简单计数作为回收准则，而是使用了一种同步回收算法，这个算法由IBM的工程师在论文Concurrent Cycle Collection in Reference Counted Systems中提出。

这个算法可谓相当复杂，从论文29页的数量我想大家也能看出来，所以我不打算(也没有能力)完整论述此算法，有兴趣的朋友可以阅读上面的提到的论文(强烈推荐，这篇论文非常精彩)。

我在这里，只能大体描述一下此算法的基本思想。

首先PHP会分配一个固定大小的“根缓冲区”，这个缓冲区用于存放固定数量的zval，这个数量默认是10,000，如果需要修改则需要修改源代码Zend/zend_gc.c中的常量GC_ROOT_BUFFER_MAX_ENTRIES然后重新编译。

由上文我们可以知道，一个zval如果有引用，要么被全局符号表中的符号引用，要么被其它表示复杂类型的zval中的符号引用。因此在zval中存在一些可能根(root)。这里我们暂且不讨论PHP是如何发现这些可能根的，这是个很复杂的问题，总之PHP有办法发现这些可能根zval并将它们投入根缓冲区。

当根缓冲区满额时，PHP就会执行垃圾回收，此回收算法如下：

1、对每个根缓冲区中的根zval按照深度优先遍历算法遍历所有能遍历到的zval，并将每个zval的refcount减1，同时为了避免对同一zval多次减1(因为可能不同的根能遍历到同一个zval)，每次对某个zval减1后就对其标记为“已减”。

2、再次对每个缓冲区中的根zval深度优先遍历，如果某个zval的refcount不为0，则对其加1，否则保持其为0。

3、清空根缓冲区中的所有根(注意是把这些zval从缓冲区中清除而不是销毁它们)，然后销毁所有refcount为0的zval，并收回其内存。

如果不能完全理解也没有关系，只需记住PHP5.3的垃圾回收算法有以下几点特性：

1、并不是每次refcount减少时都进入回收周期，只有根缓冲区满额后在开始垃圾回收。

2、可以解决循环引用问题。

3、可以总将内存泄露保持在一个阈值以下。

PHP5.2与PHP5.3垃圾回收算法的性能比较

由于我目前条件所限，我就不重新设计试验了，而是直接引用PHP Manual中的实验，关于两者的性能比较请参考PHP Manual中的相关章节：http://www.php.net/manual/en/features.gc.performance-considerations.php。

首先是内存泄露试验，下面直接引用PHP Manual中的实验代码和试验结果图：

```
1. <?php
2. class Foo
3. {
4.     public $var = '3.1415962654';
5. }
```

```
6.
7.  $baseMemory = memory_get_usage();
8.
9.  for ( $i = 0; $i <= 100000; $i++ )
10. {
11.     $a = new Foo;
12.     $a->self = $a;
13.     if ( $i % 500 === 0 )
14.     {
15.         echo sprintf( '%
    8d: ', $i ), memory_get_usage() - $baseMemory, "\n";
16.     }
17. }
18. ?>
```

可以看到在可能引发累积性内存泄露的场景下，PHP5.2发生持续累积性内存泄露，而PHP5.3则总能将内存泄露控制在一个阈值以下(与根缓冲区大小有关)。

另外是关于性能方面的对比：

```
1.  <?php
2.  class Foo
3.  {
4.      public $var = '3.1415962654';
5.  }
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. ?>
```

这个脚本执行1000000次循环，使得延迟时间足够进行对比。

然后使用CLI方式分别在打开内存回收和关闭内存回收的的情况下运行此脚本：

```
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
```

在我的机器环境下，运行时间分别为6.4s和7.2s，可以看到PHP5.3的垃圾回收机制会慢一些，但是影响并不大。

与垃圾回收算法相关的PHP配置

可以通过修改php.ini中的zend.enable_gc来打开或关闭PHP的垃圾回收机制，也可以通过

调用gc_enable()或gc_disable()打开或关闭PHP的垃圾回收机制。在PHP5.3中即使关闭了垃圾回收机制，PHP仍然会记录可能根到根缓冲区，只是当根缓冲区满额时，PHP不会自动运行垃圾回收，当然，任何时候您都可以通过手工调用gc_collect_cycles()函数强制执行内存回收。

来自 <http://www.cnblogs.com/lovehappying/p/3679356.html>

参考：https://secure.php.net/manual/zh/features.gc.php

10. **消息队列的应用场景**

1异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种1.串行的方式；2.并行方式。

（1）串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。（架构KKQ：466097527，欢迎加入）

（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。

假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次（1000/150）。并行方式处理的请求量是10次（1000/100）。

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：

按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍。

2应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图：

传统模式的缺点：

1） 假如库存系统无法访问，则订单减库存将失败，从而导致订单失败；

2） 订单系统与库存系统耦合；

如何解决以上问题呢？引入应用消息队列后的方案，如下图：

- 订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功。
- 库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。
- 假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

3流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

1. 可以控制活动的人数；
2. 可以缓解短时间内高流量压垮应用；
3. 用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面；
4. 秒杀业务根据消息队列中的请求信息，再做后续处理。

4日志处理

日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。架构简化如下：

- 日志采集客户端，负责日志数据采集，定时写受写入Kafka队列；
- Kafka消息队列，负责日志数据的接收，存储和转发；
- 日志处理应用：订阅并消费kafka队列中的日志数据；

以下是新浪kafka日志处理应用案例：

(1)Kafka：接收用户日志的消息队列。

(2)Logstash：做日志解析，统一成JSON输出给Elasticsearch。

(3)Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能。

(4)Kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ELK stack的重要原因。

5消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

点对点通讯：

客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。

来自 <http://www.cnblogs.com/stopfalling/p/5375492.html>

11. 有哪些线程安全的函数

什么是线程安全？

一个函数被多个并发线程**反复调用时，它会一直产生正确的结果**，则该函数是线程安全函数。

那么什么又是可重入函数？

当一个函数在被一个线程调用时，可以**允许被其他线程再调用**。即两个函数“**同时**”发生。则该函数是可重入函数。

所以，显而易见，如果一个函数是可重入的，那么它肯定是线程安全的。但反之未然，一个函数是线程安全的，却未必是可重入的。比如我们在一个函数中调用到了一个全局变量NUM用来标记某一东西的数量。学个操作系统的同学都知道，如果我们在修改它的值的时候发生了中断，两一个函数又对他进行了修改，此时该变量的值会出错。这种函数就是线程不安全函数。他是属于没有保护共享变量的线程不安全函数。在单线程时运行毫无问题，但一旦放到多线程中就容易出bug。但如果我们在修改这个全局变量NUM前对他进行加锁，再操作完后再进行解锁。这样即使有两个线程在调用这个函数，其结果也不会出问题。此时，这个函数就是线程安全函数。但他依旧不是可重入函数。因为他不能保证两个函数“同时”运行，必须等待解锁后才能运行。而我们在平时开发中应该尽量编写可重入的函数。 如下图：

线程不安全函数主要分为以下四大类：

第一类：不保护共享变量的函数，

a, 函数中访问全局变量和堆。

共享变量在多线程中是共享数据比如全局变量和堆，如果不保护共享变量，多线程时会出bug。

可以通过同步机制来保护共享数据，比如加锁。

第二类：函数中分配，重新分配释放全局资源。

与上面第一点基本相同，通过加锁可解决

第三类：返回指向静态变量的指针的函数，函数中通过句柄和指针的不直接访问。

比如，我们要计算a,b两个变量的和，于是将a，b的指针传入某一个函数，然而此时可能有另一个线程改变了a,b的值，此时在函数中我们通过地址取到的两个数的值已经改变了，所以计算出的结果也就是错的了。

又比如某些函数（如gethostbyname）将计算结果放在静态结构中，并返回一个指向这个结构的指针。在多线程中一个线程调用的结构可能被另一个线程覆盖。可以通过重写函数和加锁拷贝技术来消除。加锁拷贝技术指在每个位置对互斥锁加锁，调用线程不安全函数，动态的为结果分配存储器，拷贝函数返回的结构，然后解锁。

第四类：调用线程不安全函数

常见的系统线程不安全函数：

线程不安全函数	线程不安全 类	unix线程安全版本
rand	2	rand_r
strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
geyhostbyname	3	gethostbyname_r
inet_ntoa	3	
localtime	3	localtime_r

*UNIX*环境高级编程列出 *POSIX.1*规范中的非线程安全的函数：

asctime	ecvt	gethostent	getutxline	putc_unlocked
basename	encrypt	getlogin	gmtime	putchar_unlock
catgets	endgrent	getnetbyaddr	hcreate	putenv
crypt	endpwent	getnetbyname	hdestroy	pututxline
ctime	endutxent	getopt	hsearch	rand
dbm_clearer	fcvt	getprotobyname	inet_ntoa	readdir
dbm_close	ftw	getprotobynumb	L64a	setenv
dbm_delete	getcvt	getprotobynumb	lgamma	setgrent

dbm_error	getc_unlocked	getprotoent	lgammaf	setkey
dbm_fetch	getchar_unlock	getpwent	lgammal	setpwent
dbm_firstke	getdate	getpwnam	localecon	setutxent
dbm_nextke	getenv	getpwuid	lrand48	strerror
dbm_open	getgrent	getservbyname	mrand48	strtok
dbm_store	getgrgid	getservbyport	nftw	ttyname
dirname	getgrnam	getservent	nl_langinf	unsetenv
dlerror	gethostbyaddr	getutxent	ptsname	wcstombs
drand48	gethostbyname	getutxid	ptsname	ectomb

目前大部分上述函数目前已经有了对应的线程安全版本的实现，例如：针对 *getpwnam*的 *getpwnam_r()*，（这里的 *_r*表示可重入 (*reentrant*)，如前所述，可重入的函数都是线程安全的）。在多线程软件开发中，如果需要使用到上所述函数，应优先使用它们对应的线程安全版本。

因此在编写线程安全函数时，要注意两点：

- 1, 减少对临界资源的依赖，尽量避免访问全局变量，静态变量或其它共享资源，如果必须要使用共享资源，所有使用到的地方必须要进行互斥锁 (*Mutex*) 保护；
- 2, 线程安全的函数所调用到的函数也应该是线程安全的，如果所调用的函数不是线程安全的，那么这些函数也必须被互斥锁 (Mutex) 保护；

来自 <http://www.cnblogs.com/xyxxs/p/4655692.html>

12. **redis的数据结构**

String——字符串
Hash——字典
List——列表
Set——集合
Sorted Set——有序集合
下面我们就来简单说明一下它们各自的使用场景：

1. String——字符串

String 数据结构是简单的 key-value 类型，value 不仅可以是 String，也可以是数字（当数字类型用 Long 可以表示的时候encoding 就是整型，其他都存储在 sdshdr 当做字符串）。使用 Strings 类型，可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受 Redis 的定时持久化（可以选择 RDB 模式或者 AOF 模式），操作日志及 Replication 等功能。除了提供与 Memcached 一样的 get、set、incr、decr 等操作外，Redis 还提供了下面一些操作：

复制代码代码如下:

1.LEN niushuai：O(1)获取字符串长度

2.APPEND niushuai redis：往字符串 append 内容，而且采用智能分配内存（每次2倍）

3.设置和获取字符串的某一段内容

4.设置及获取字符串的某一位（bit）

5.批量设置一系列字符串的内容

6.原子计数器

7.GETSET 命令的妙用，请于清空旧值的同时设置一个新值，配合原子计数器使用

2. Hash——字典

在 Memcached 中，我们经常将一些结构化的信息打包成 hashmap，在客户端序列化后存储为一个字符串的值（一般是 JSON 格式），比如用户的昵称、年龄、性别、积分等。这时候在需要修改其中某一项时，通常需要将字符串（JSON）取出来，然后进行反序列化，修改某一项的值，再序列化成字符串（JSON）存储回去。简单修改一个属性就干这么多事情，消耗必定是很大的，也不适用于一些可能并发操作的场合（比如两个并发的操作都需要修改积分）。而 Redis 的 Hash 结构可以使你像在数据库中 Update 一个属性一样只修改某一项属性值。

复制代码代码如下:

存储、读取、修改用户属性

3. List——列表

List 说白了就是链表（redis 使用双端链表实现的 List），相信学过数据结构知识的人都应该能理解其结构。使用 List 结构，我们可以轻松地实现最新消息排行等功能（比如新浪微博的 TimeLine ）。List 的另一个应用就是消息队列，可以利用 List 的 *PUSH 操作，将任务存在 List 中，然后工作线程再用 POP 操作将任务取出进行执行。Redis 还提供了操作 List 中某一段元素的 API，你可以直接查询，删除 List 中某一段的元素。

复制代码代码如下:

1.微博 TimeLine

2.消息队列

4. Set——集合

Set 就是一个集合，集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Set 数据结构，可以存储一些集合性的数据。比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。因为 Redis 非常人性化的为集合提供了求交集、并集、差集等操作，那么就可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

1.共同好友、二度好友

2.利用唯一性，可以统计访问网站的所有独立 IP

3.好友推荐的时候，根据 tag 求交集，大于某个 threshold 就可以推荐

5. Sorted Set——有序集合

和Sets相比，Sorted Sets是将 Set 中的元素增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，比如一个存储全班同学成绩的 Sorted Sets，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用 Sorted Sets 来做带权重的队列，比如普通消息的 score 为1，重要消息的 score 为2，然后工作线程可以选择按 score 的倒序来获取工作任务。让重要的任

务优先执行。

1.带有权重的元素，比如一个游戏的用户得分排行榜

2.比较复杂的数据结构，一般用到的场景不算太多

二、redis 其他功能使用场景

1. 订阅-发布系统

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

2. 事务——Transactions

谁说 NoSQL 都不支持事务，虽然 Redis 的 Transactions 提供的并不是严格的 ACID 的事务（比如一串用 EXEC 提交执行的命令，在执行中服务器宕机，那么会有一部分命令执行了，剩下的没执行），但是这个 Transactions 还是提供了基本的命令打包执行的功能（在服务器不出问题的情况下，可以保证一连串的命令是顺序在一起执行的，中间有会有其它客户端命令插进来执行）。Redis 还提供了一个 Watch 功能，你可以对一个 key 进行 Watch，然后再执行 Transactions，在这过程中，如果这个 Watched 的值进行了修改，那么这个 Transactions 会发现并拒绝执行。

来自 <http://www.jb51.net/article/54774.htm>

13. 接口和抽象类的区别

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。

6、抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果

7、抽象类里可以没有抽象方法

8、如果一个类里有抽象方法，那么这个类只能是抽象类

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承。

1.抽象类 和 接口 都是用来抽象具体对象的. 但是接口的抽象级别最高

2.抽象类可以有具体的方法 和属性，接口只能有抽象方法和不可变常量
3.抽象类主要用来抽象类别,接口主要用来抽象功能.
4、抽象类中，且不包含任何实现，派生类必须覆盖它们。接口中所有方法都必须是未实现的。

当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

抽象类的功能要远超过接口，但是，定义抽象类的代价高。因为高级语言来说（从实际设计上来说也是）每个类只能继承一个类。在这个类中，你必须继承或编写出其所有子类的

所有共性。虽然接口在功能上会弱化许多，但是它只是针对一个动作的描述。而且你可以在一个类中同时实现多个接口。在设计阶段会降低难度的。

接口的使用

接口：interface

在PHP中，我们可以规定，一个对象应该具有哪些公共的外部操作，即可使用interface来规定。

公共的方法就是接口。用于规定一个对象应该用于哪些公共的操作方法（接口），这个也叫接口（公共操作方法的集合）

即：接口（interface结构，公共方法集合）

公共方法（接口方法）

定义：用于限定某个对象所必须拥有的公共操作方法的一种结构，称之为接口（interface）

语法：定义接口结构，使用interface关键字。接口内定义的都是一些公共方法。

注意：

1.接口方法，访问权限必须是公共的 public

2.接口内只能有公共方法，不能存在成员变量

3.接口内只能含有未被实现的方法，也叫抽象方法，但是不用abstract关键字。

类实现接口，利用关键字implements完成。

这样，实现该接口的类，必须实现接口内所有的抽象方法。而且可以肯定，该方法一定是公

共的外部操作方法。

多实现：该功能，在理论上可以通过抽象类来实现，但是抽象类，不专业。

使用接口则专业些，实现上，因为php支持多实现，而仅支持单继承。

php对象接口的支持，可以定义类常量，接口之间也可以继承

抽象方法和抽象类

在OOP 语言中，一个类可以有一个或多个子类，而每个类都有至少一个公有方法做为外部代码访问其的接口。而抽象方法就是为了方便继承而引入的，我们先来看一下抽象类和抽象方法的定义再说明它的用途。

什么是抽象方法？我们在类里面定义的没有方法体的方法就是抽象方法，所谓的没有方法体指的是，在方法声明的时候没有大括号以及其中的内容，而是直接在声明时在方法名后加上分号结束，另外在声明抽象方法时还要加一个关键字“abstract”来修饰；

例如：

abstract function fun1();

abstract function fun2();

上例是就是“abstract”修饰的没有方法体的抽象方法“fun1()”和“fun2()”，不要忘记

抽象方法后面还要有一个分号；那么什么是抽象类呢？只要一个类里面有一个方法是抽象方法，那么这个类就要定义为抽象类，抽象类也要使用“abstract”关键字来修饰；在抽象类里

面可以有不是抽象的方法和成员属性，但只要有一个方法是抽象的方法，这个类就必须声明为抽象类，使用“abstract”来修饰。

http://hovertree.com/menu/php/

上例中定义了一个抽象类“Demo”使用了“abstract”来修饰，在这个类里面定义了一个成员属性“$test”，和两个抽象方法“fun1”和“fun2”还有一个非抽象的方法 fun3()；那

么抽象类我们怎么使用呢？最重要的一点就是抽象类不能产生实例对象，所以也不能直接使用，前面我们多次提到过类不能直接使用，我们使用的是通过类实例化出来的对象，那么抽象类不能产生实例对象我们声明抽象类有什么用呢？我们是将抽象方法是做为子类重载的模板使用的，定义抽象类就相当于定义了一种规范，这种规范要求子类去遵守，子类继函抽象

类之后，把抽象类里面的抽象方法按照子类的需要实现。子类必须把父类中的抽象方法全部都实现，否则子类中还存在抽象方法，那么子类还是抽象类，还是不能实例化对；为什么我们非要从抽象类中继承呢？因为有的时候我们要实现一些功能就必须从抽象类中继承，否则这些功能你就实现不了，如果继承了抽象类，就要实现类其中的抽象方法；

单例模式

1. 单例模式（职责模式）：
2. 简单的说，一个对象（在学习设计模式之前，需要比较了解面向对象思想）只负责一个特定的任务；
3. 单例类：
4. 1、构造函数需要标记为private（访问控制：防止外部代码使用new操作符创建对象），单例类不能在其他类中实例化，只能被其自身实例化；
5. 2、拥有一个保存类的实例的静态成员变量
6. 3、拥有一个访问这个实例的公共的静态方法（常用getInstance()方法进行实例化单例类，通过instanceof操作符可以检测到类是否已经被实例化）
7. 另外，需要创建__clone()方法防止对象被复制（克隆）
8. 为什么要使用PHP单例模式？
9. 1、php的应用主要在于数据库应用, 所以一个应用中会存在大量的数据库操作, 使用单例模式, 则可以避免大量的new 操作消耗的资源。
10. 2、如果系统中需要有一个类来全局控制某些配置信息, 那么使用单例模式可以很方便的实现. 这个可以参看ZF的FrontController部分。
11. 3、在一次页面请求中, 便于进行调试, 因为所有的代码(例如数据库操作类db)都集中在一个类中, 我们可以在类中设置钩子, 输出日志，从而避免到处var_dump, echo。
12. 代码实现：
13. /1**
14. * 设计模式之单例模式
15. * $_instance必须声明为静态的私有变量
16. * 构造函数和析构函数必须声明为私有,防止外部程序new
17. * 类从而失去单例模式的意义
18. * getInstance()方法必须设置为公有的,必须调用此方法
19. * 以返回实例的一个引用
20. * ::操作符只能访问静态变量和静态函数
21. * new对象都会消耗内存
22. * 使用场景:最常用的地方是数据库连接。
23. * 使用单例模式生成一个对象后，
24. * 该对象可以被其它众多对象所使用。
25. */

```
class Danli {

//保存类实例的静态成员变量
private static $_instance;

//private标记的构造方法
private function __construct(){
echo 'This is a Constructed method;';
}

//创建__clone方法防止对象被复制克隆
public function __clone(){
trigger_error('Clone is not allow!',E_USER_ERROR);
}

//单例方法,用于访问实例的公共的静态方法
public static function getInstance(){
if(!(self::$_instance instanceof self)){
self::$_instance = new self;
}
return self::$_instance;
}

public function test(){
echo '调用方法成功';
}

}
   // 何问起 hovertree.com
//用new实例化private标记构造函数的类会报错
//$danli = new Danli();

//正确方法,用双冒号::操作符访问静态方法获取实例
$danli = Danli::getInstance();
$danli->test();

//复制(克隆)对象将导致一个E_USER_ERROR
$danli_clone = clone $danli;
```

来自 <http://www.cnblogs.com/yongjiapei/p/5494894.html>

参考：http://blog.csdn.net/wenwen091100304/article/details/48381023

14. **对称和非对称算法的区别**
15. **new和mallloc的区别：**
    http://blog.jobbole.com/102002/

16. **如何防止被sql注入**
17. **表单提交可以跨域吗**

造成跨域的两种策略

浏览器的同源策略会导致跨域，这里同源策略又分为以下两种

- DOM同源策略：禁止对不同源页面DOM进行操作。这里主要场景是iframe跨域

的情况，不同域名的iframe是限制互相访问的。

- XmlHttpRequest同源策略：禁止使用XHR对象向不同源的服务器地址发起HTTP请求。

只要协议、域名、端口有任何一个不同，都被当作是不同的域，之间的请求就是跨域操作。

为什么要有跨域限制

了解完跨域之后，想必大家都会有这么一个思考，为什么要有跨域的限制，浏览器这么做是出于何种原因呢。其实仔细想一想就会明白，跨域限制主要是为了安全考虑。

AJAX同源策略主要用来防止CSRF攻击。如果没有AJAX同源策略，相当危险，我们发起的每一次HTTP请求都会带上请求地址对应的cookie，那么可以做如下攻击：
用户登录了自己的银行页面 http://mybank.com，http://mybank.com向用户的cookie中添加用户标识。
用户浏览了恶意页面 http://evil.com。执行了页面中的恶意AJAX请求代码。
http://evil.com向http://mybank.com发起AJAX HTTP请求，请求会默认把http://mybank.com对应cookie也同时发送过去。
银行页面从发送的cookie中提取用户标识，验证用户无误，response中返回请求数据。此时数据就泄露了。
而且由于Ajax在后台执行，用户无法感知这一过程。

DOM同源策略也一样，如果iframe之间可以跨域访问，可以这样攻击：
做一个假网站，里面用iframe嵌套一个银行网站 http://mybank.com。
把iframe宽高啥的调整到页面全部，这样用户进来除了域名，别的部分和银行的网站没有任何差别。
这时如果用户输入账号密码，我们的主网站可以跨域访问到http://mybank.com的dom节点，就可以拿到用户的输入了，那么就完成了一次攻击。
所以说有了跨域跨域限制之后，我们才能更安全的上网了。

跨域的解决方式

跨域资源共享

CORS是一个W3C标准，全称是”跨域资源共享”（Cross-origin resource sharing）。

对于这个方式，阮一峰老师总结的文章特别好，希望深入了解的可以看一下http:<//www.ruanyifeng.com/blog/2016/04/cors.html>。
这里我就简单的说一说大体流程。
对于客户端，我们还是正常使用xhr对象发送ajax请求。
唯一需要注意的是，我们需要设置我们的xhr属性withCredentials为true，不然的话，cookie是带不过去的哦，设置： xhr.withCredentials = true;
对于服务器端，需要在 response header中设置如下两个字段:
Access-Control-Allow-Origin: http://www.yourhost.com
Access-Control-Allow-Credentials:true
这样，我们就可以跨域请求接口了。

jsonp实现跨域

基本原理就是通过动态创建script标签,然后利用src属性进行跨域。
这么说比较模糊，我们来看个例子:
返回的js脚本，直接会执行。所以就执行了事先定义好的fun函数了，并且把数据传入了进来。
当然，这个只是一个原理演示，实际情况下，我们需要动态创建这个fun函数，并且在数据返回的时候销毁它。
因为在实际使用的时候，我们用的各种ajax库，基本都包含了jsonp的封装，不过我们还是要知道一下原理，不然就不知道为什么jsonp不能发post请求了~

服务器代理

浏览器有跨域限制，但是服务器不存在跨域问题，所以可以由服务器请求所要域的资源再返回给客户端。

服务器代理是万能的。

document.domain来跨子域

对于主域名相同，而子域名不同的情况，可以使用document.domain来跨域
这种方式非常适用于iframe跨域的情况，直接看例子吧
比如a页面地址为 a.yourhost.com b页面为 b.yourhost.com。
这样就可以通过分别给两个页面设置 document.domain = yourhost.com 来实现跨域。
之后，就可以通过 parent 或者 window[‘iframename’]等方式去拿到iframe的window对象了。

使用window.name进行跨域

window.name跨域同样是受到同源策略限制，父框架和子框架的src必须指向统一域名。window.name的优势在于，name的值在不同的页面(或者不同的域名)，加载后仍然存在，除非你显示的更改。并且支持的长度达到2M.

来自 <https://mp.weixin.qq.com/s?__biz=MzU0OTExNzYwNg==&mid=2247483685&idx=1&sn=543e9736146405e9e5b37ec5a1c4b448&chksm=fbb58aecccc203faa4517d77e3f0f723896d2cd5c3ddba2c360b0f0a03f0dc873a1df552d563&scene=0&key=72da87db120ba9cf61099ab1aa1018a958a343b288aa45db018dfe9121cff1879fabc215189a94b80df3231def1863854c558191cee62b44b7723f16099866bb8c3922d77f3430599807dfa31f570e9f&ascene=0&uin=MjIxODIxNjA0MA%3D%3D&devicetype=iMac+MacBookPro12%2C1+OSX+OSX+10.11.6+build(15G1004)&version=12010110&nettype=WIFI&fontScale=100&pass_ticket=yNbtn1PSEE2%2B4fxr6HsvYzix%2F29VqEk%2FSw%2BUz7LY0%2FJn75cfBnENdWcok3S%2FcXRr>

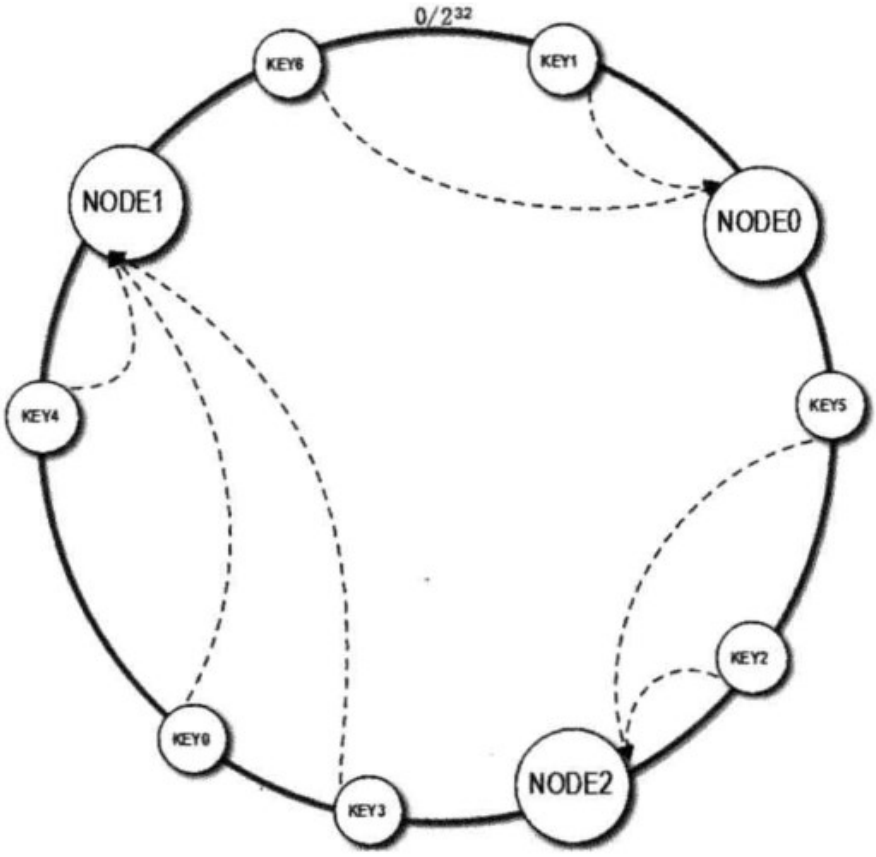
18. 如何保证分布式缓存的一致性

当服务器不多，并且不考虑扩容的时候，可直接使用简单的路由算法，用服务器数除缓存数据KEY的hash值，余数作为服务器下标即可。

但是当业务发展，网站缓存服务需要扩容时就会出现问题，比如3台缓存服务器要扩容到4台，就会导致75%的数据无法命中，当100台服务器中增加一台，不命中率会到达99%（n/（n+1））,这显然是不能接受的。

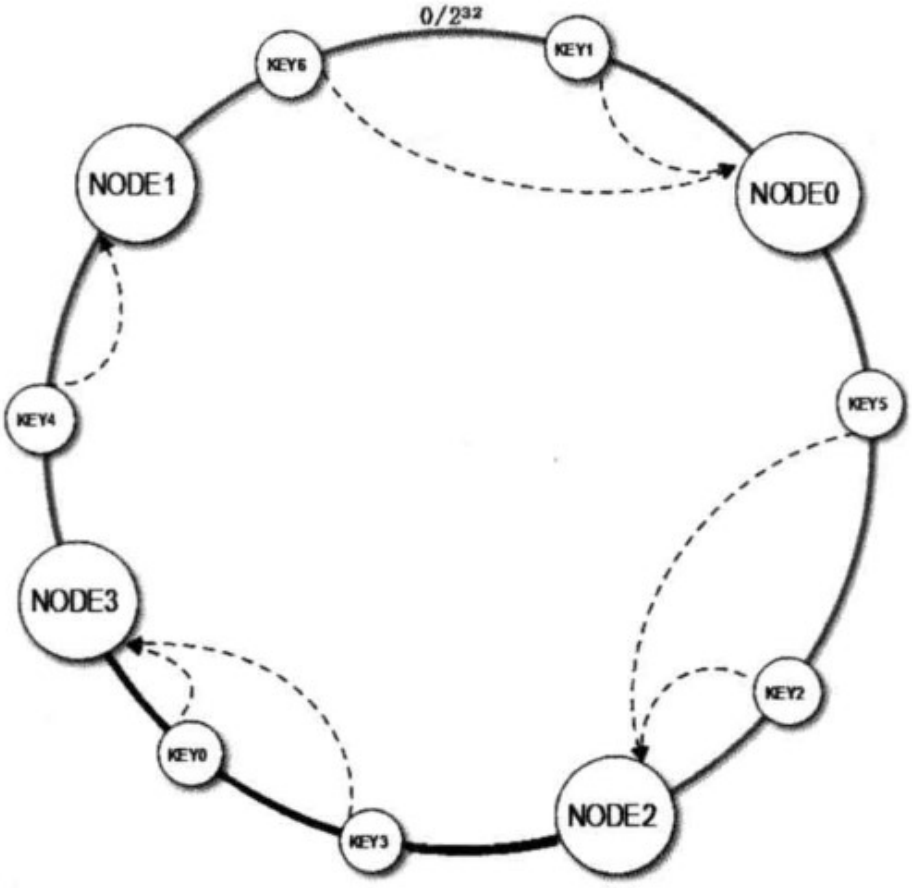
在设计分布式缓存集群的时候，需要考虑集群的伸缩性，也就是当向集群中增加服务器的时候，要尽量减小对集群的影响，而一致性hash算法就是用来解决集群伸缩性。

一致性hash算法通过构造一个长度为2^32的整数环，根据节点名的hash值将缓存服务器节点放置在这个环上，然后计算要缓存的数据的key的hash值，顺时针找到最近的服务器节点，将数据放到该服务器上。



有Node0,Node1,Node2三个节点，假设Node0的hash值是1024,key1的hash值是500，key1在环上顺时针查找，最近的节点就是Node0。

当服务器集群又开始扩容，新增了Node3节点，从三个节点扩容到了四个节点。



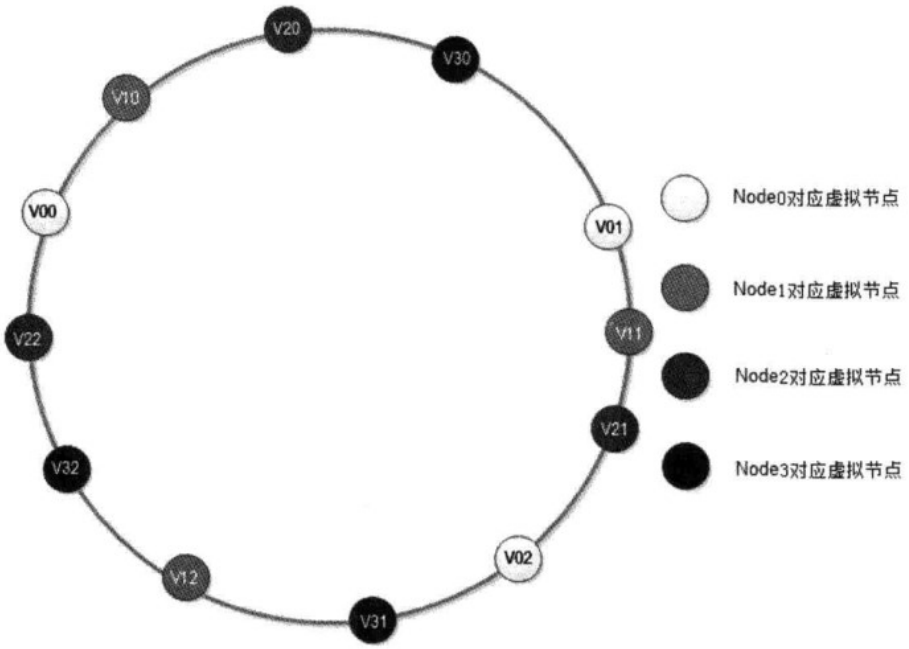
Node3加到了Node2和Node1之间，除了Node2到Node3之间原本是Node1的数据无法再命中，其它的数据不受影响，3台扩容到4台可命中率高达75%，

而且集群越大，影响越小，100台服务器增加一台，命中率可达到99%。

查找不小于查找树的最小值是用的二叉查找树实现的。

但是这样子还是会存在一个问题，就是负载不均衡的问题，当Node3加到Node2和Node1之间时，原本会访问Node1的缓存数据有50%的概率会缓存到Node3上了，这样Node0和Node2的负载会是Node1和Node3的两倍。

要解决一致性hash算法带来的负载不均衡问题，可通过将每台物理服务器虚拟成一组虚拟缓存服务器，将虚拟服务器的hash值放置在hash环上，KEY在环上先找到虚拟服务器节点，然后再映射到实际的服务器上。



这样在Node0,1,2虚拟节点都已存在的情况下，将Node3的多个虚拟节点分散到它们中间，多个虚拟的Node3节点会影响到其它的多个虚拟节点，而不是只影响其中一个，这样将命中率不会有变化，但是负载却更加均衡了而且虚拟节点越多越均衡。

来自 <http://www.cnblogs.com/akaneblog/p/6736386.html>

19. MD5加密原理

MD5消息摘要算法，属Hash算法一类。MD5算法对输入任意长度的**消息**进行运行，产生一个128位的消息摘要。

以下所描述的消息长度、填充数据都以位(Bit)为单位，字节序为小端字节。

算法原理

1、数据填充

对消息进行数据填充，使消息的长度对512取模得448，设消息长度为X，即满足X mod 512 =448。根据此公式得出需要填充的数据长度。

填充方法：在消息后面进行填充，填充第一位为1，其余为0。

2、添加消息长度

在第一步结果之后再填充上原消息的长度，可用来进行的存储长度为64位。如果消息长度大于2^{64}，则只使用其低64位的值，即（消息长度 对 2^{64}取模）。

在此步骤进行完毕后，最终消息长度就是512的整数倍。

3、数据处理

准备需要用到的数据：

- 4个常数： A = 0x67452301, B = 0x0EFCDAB89, C = 0x98BADCFE, D = 0x10325476;
- 4个函数：F(X,Y,Z)=(X & Y) | ((~X) & Z); G(X,Y,Z)=(X & Z) | (Y & (~Z)); H(X,Y,Z)=X ^ Y ^ Z; I(X,Y,Z)=Y ^ (X | (~Z));

把消息分以512位为一分组进行处理，每一个分组进行4轮变换，以上面所说4个常数为起始变量进行计算，重新输出4个变量，以这4个变量再进行下一分组的运算，如果已经是最后一个分组，则这4个变量为最后的结果，即MD5值。

具体计算的实现较为复杂，建议查阅相关书籍，下面给出在C++上的实现代码。

来自 <http://www.cnblogs.com/hjgods/p/3998570.html>

20. **为什么需要长连接**

可以实现连接复用，只是需要心跳包来keep live

原因：

如果是正常终止客户端进程，系统会对进程正在占用的资源进行回收。此时客户端所占用的socket端口会被释放，服务器端也会被告知对方断开socket连接了，因此终止与该客户端的连接。而如果客户端非正常断开（比如客户端主机突然断电），则客户端的**操作系统**会在第一时间产生中断，保护操作系统。哪个操作系统还会特意耗费时间去回收socket资源←_←。所以即便客户端明明已经断开了连接，但服务器却迟迟没办法知道客户端断开的消息。因此会在较长一段时间内不会告诉编程者“某客户端已断开”，即便你写了在客户端断开后立刻回收资源的代码。

解决：

相信不少人都听说过“心跳包”吧。保持客户端与服务器长连接的话，心跳包是必不可少的。设定阈值n，心跳间隔时间T，当服务器在n*T的时间内没有听到客户端的心跳，那么就可以判定客户端“死亡”了，主动与它断开连接，回收资源。至于n和T怎么设置，纯看项目的需求和编程者的心情了。当然，心跳包的写法很多种，可以是服务器向客户端发心跳要

求回应，也可以是客户端直接向服务器发心跳。不同的应用场景可能都不同。

来自 <http://blog.csdn.net/qq_21882325/article/details/66479400>

21. 长连接怎么实现的

基于HTTP的长连接,是一种通过长轮询方式实现"服务器推"的技术,它弥补了HTTP简单的请求应答模式的不足,极大地增强了程序的实时性和交互性。

一、什么是长连接、长轮询？

用通俗易懂的话来说，就是客户端不停的向服务器发送请求以获取最新的数据信息。这里的“不停”其实是有停止的，只是我们人眼无法分辨是否停止，它只是一种快速的停下然后又立即开始连接而已。

二、长连接、长轮询的应用场景

长连接、长轮询一般应用与WebIM、ChatRoom和一些需要及时交互的网站应用中。其真实案例有：WebQQ、Hi网页版、Facebook IM等。

如果你对服务器端的反向Ajax感兴趣，可以参考这篇文章 DWR 反向Ajax 服务器端推的方式：http://www.cnblogs.com/hoojo/category/276235.html

欢迎大家继续支持和关注我的博客：

http://hoojo.cnblogs.com

http://blog.csdn.net/IBM_hoojo

也欢迎大家和我交流、探讨IT方面的知识。

email：hoojo_@126.com

三、优缺点

轮询：客户端定时向服务器发送Ajax请求，服务器接到请求后马上返回响应信息并关闭连接。

优点：后端程序编写比较容易。

缺点：请求中有大半是无用，浪费带宽和服务器资源。

实例：适于小型应用。

长轮询：客户端向服务器发送Ajax请求，服务器接到请求后hold住连接，直到有新消息才返回响应信息并关闭连接，客户端处理完响应信息后再向服务器发送新的请求。

优点：在无消息的情况下不会频繁的请求，耗费资源小。

缺点：服务器hold连接会消耗资源，返回数据顺序无保证，难于管理维护。

实例：WebQQ、Hi网页版、Facebook IM。

长连接：在页面里嵌入一个隐藏iframe，将这个隐藏iframe的src属性设为对一个长连接的请求或是采用xhr请求，服务器端就能源源不断地往客户端输入数据。

优点：消息即时到达，不发无用请求；管理起来也相对方便。

缺点：服务器维护一个长连接会增加开销。

实例：Gmail聊天

Flash Socket：在页面中内嵌入一个使用了Socket类的 Flash 程序JavaScript通过调用此Flash程序提供的Socket接口与服务器端的Socket接口进行通信，JavaScript在收到服务器端传送的信息后控制页面的显示。

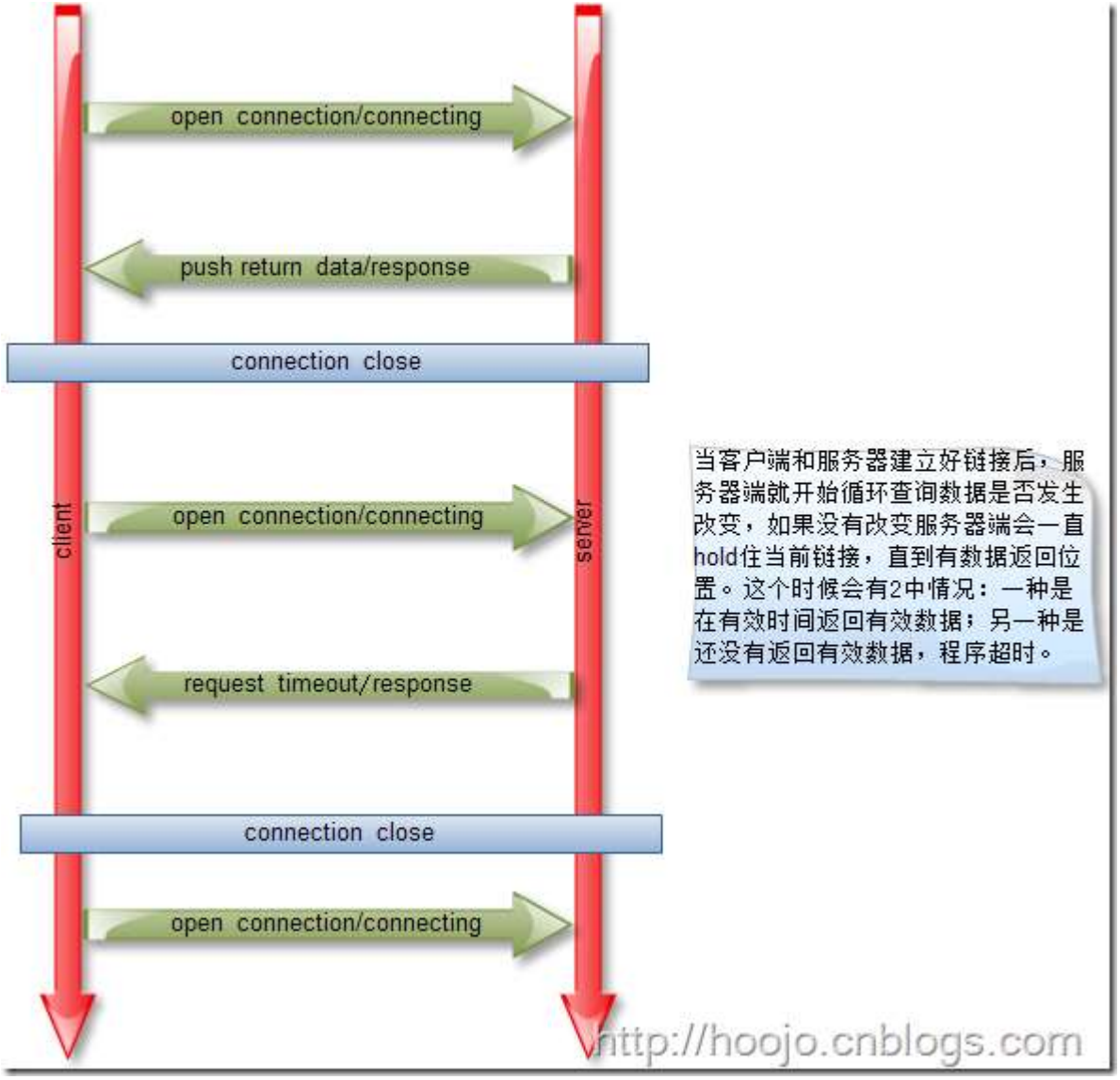
优点：实现真正的即时通信，而不是伪即时。

缺点：客户端必须安装Flash插件；非HTTP协议，无法自动穿越防火墙。

实例：网络互动游戏。

四、实现原理

所谓长连接，就是要在客户端与服务器之间创建和保持稳定可靠的连接。其实它是一种很早就存在的技术，但是由于浏览器技术的发展比较缓慢，没有为这种机制的实现提供很好的支持。所以要达到这种效果，需要客户端和服务器的程序共同配合来完成。通常的做法是，在服务器的程序中加入一个死循环，在循环中监测数据的变动。当发现新数据时，立即将其输出给浏览器并断开连接，浏览器在收到数据后，再次发起请求以进入下一个周期，这就是常说的长轮询（long-polling）方式。如下图所示，它通常包含以下几个关键过程：



1. 轮询的建立

建立轮询的过程很简单，浏览器发起请求后进入循环等待状态，此时由于服务器还未做出应答，所以HTTP也一直处于连接状态中。

2. 数据的推送

在循环过程中，服务器程序对数据变动进行监控，如发现更新，将该信息输出给浏览器，随即断开连接，完成应答过程，实现“服务器推”。

3. 轮询的终止

轮询可能在以下3种情况时终止：

3.1. 有新数据推送

当循环过程中服务器向浏览器推送信息后，应该主动结束程序运行从而让连接断开，这样浏览器才能及时收到数据。

3.2. 没有新数据推送

循环不能一直持续下去，应该设定一个最长时限，避免WEB服务器超时（Timeout），若一直没有新信息，服务器应主动向浏览器发送本次轮询无新信息的正常响应，并断开连接，这也被称为“心跳”信息。

3.3. 网络故障或异常

由于网络故障等因素造成的请求超时或出错也可能导致轮询的意外中断，此时浏览器将收到错误信息。

4. 轮询的重建

浏览器收到回复并进行相应处理后，应马上重新发起请求，开始一个新的轮询周期。

来自 <http://www.cnblogs.com/hoojo/p/longPolling_comet_jquery_iframe_ajax.html>

22.