**【原创】PHP超时处理全面总结**

*作者：heiyeluren*

*博客：<http://blog.csdn.net/heiyeshuwu>*

*时间：2012/8/8*

**【 概述 】**

 在PHP开发中工作里非常多使用到超时处理到超时的场合，我说几个场景：

1. 异步获取数据如果某个后端数据源获取不成功则跳过，不影响整个页面展现

2. 为了保证Web服务器不会因为当个页面处理性能差而导致无法访问其他页面，则会对某些页面操作设置

3. 对于某些上传或者不确定处理时间的场合，则需要对整个流程中所有超时设置为无限，否则任何一个环节设置不当，都会导致莫名执行中断

4. 多个后端模块（MySQL、Memcached、HTTP接口），为了防止单个接口性能太差，导致整个前面获取数据太缓慢，影响页面打开速度，引起雪崩

5. 。。。很多需要超时的场合

这些地方都需要考虑超时的设定，但是PHP中的超时都是分门别类，各个处理方式和策略都不同，为了系统的描述，我总结了PHP中常用的超时处理的总结。

**【Web服务器超时处理】**

**[ Apache ]**

一般在性能很高的情况下，缺省所有超时配置都是30秒，但是在上传文件，或者网络速度很慢的情况下，那么可能触发超时操作。

目前 apache fastcgi php-fpm 模式 下有三个超时设置：

fastcgi 超时设置：

修改 httpd.conf 的fastcgi连接配置，类似如下：

|  |
| --- |
| <IfModule mod\_fastcgi.c>      FastCgiExternalServer /home/forum/apache/apache\_php/cgi-bin/php-cgi -socket /home/forum/php5/etc/php-fpm.sock      ScriptAlias /fcgi-bin/ "/home/forum/apache/apache\_php/cgi-bin/"      AddHandler php-fastcgi .php      Action php-fastcgi /fcgi-bin/php-cgi      AddType application/x-httpd-php .php  </IfModule> |

缺省配置是 30s，如果需要定制自己的配置，需要修改配置，比如修改为100秒：(修改后重启 apache)：

|  |
| --- |
| <IfModule mod\_fastcgi.c>      FastCgiExternalServer /home/forum/apache/apache\_php/cgi-bin/php-cgi -socket /home/forum/php5/etc/php-fpm.sock  -idle-timeout **100**      ScriptAlias /fcgi-bin/ "/home/forum/apache/apache\_php/cgi-bin/"      AddHandler php-fastcgi .php      Action php-fastcgi /fcgi-bin/php-cgi      AddType application/x-httpd-php .php  </IfModule> |

如果超时会返回500错误，断开跟后端php服务的连接，同时记录一条apache错误日志：

[Thu Jan 27 18:30:15 2011] [error] [client 10.81.41.110] FastCGI: comm with server "/home/forum/apache/apache\_php/cgi-bin/php-cgi" aborted: idle timeout (30 sec)

[Thu Jan 27 18:30:15 2011] [error] [client 10.81.41.110] FastCGI: incomplete headers (0 bytes) received from server "/home/forum/apache/apache\_php/cgi-bin/php-cgi"

其他 fastcgi 配置参数说明：

|  |
| --- |
| IdleTimeout 发呆时限  ProcessLifeTime 一个进程的最长生命周期，过期之后无条件kill  MaxProcessCount 最大进程个数  DefaultMinClassProcessCount 每个程序启动的最小进程个数  DefaultMaxClassProcessCount 每个程序启动的最大进程个数  IPCConnectTimeout 程序响应超时时间  IPCCommTimeout 与程序通讯的最长时间，上面的错误有可能就是这个值设置过小造成的  MaxRequestsPerProcess 每个进程最多完成处理个数，达成后自杀 |

**[ Lighttpd ]**

配置：lighttpd.conf

Lighttpd配置中，关于超时的参数有如下几个（篇幅考虑，只写读超时，写超时参数同理）：

主要涉及选项：

server.max-keep-alive-idle = 5

server.max-read-idle = 60

server.read-timeout = 0

server.max-connection-idle = 360

|  |
| --- |
| --------------------------------------------------  # 每次keep-alive 的最大请求数, 默认值是16  server.max-keep-alive-requests = 100    # keep-alive的最长等待时间, 单位是秒，默认值是5  server.max-keep-alive-idle = 1200    # lighttpd的work子进程数，默认值是0，单进程运行  server.max-worker = 2    # 限制用户在发送请求的过程中，最大的中间停顿时间(单位是秒)，  # 如果用户在发送请求的过程中(没发完请求)，中间停顿的时间太长，lighttpd会主动断开连接  # 默认值是60(秒)  server.max-read-idle = 1200    # 限制用户在接收应答的过程中，最大的中间停顿时间(单位是秒)，  # 如果用户在接收应答的过程中(没接完)，中间停顿的时间太长，lighttpd会主动断开连接  # 默认值是360(秒)  server.max-write-idle = 12000    # 读客户端请求的超时限制，单位是秒, 配为0表示不作限制  # 设置小于max-read-idle时，read-timeout生效  server.read-timeout = 0    # 写应答页面给客户端的超时限制，单位是秒，配为0表示不作限制  # 设置小于max-write-idle时，write-timeout生效  server.write-timeout = 0    # 请求的处理时间上限，如果用了mod\_proxy\_core，那就是和后端的交互时间限制, 单位是秒  server.max-connection-idle = 1200  -------------------------------------------------- |

**说明：**

对于一个keep-alive连接上的连续请求，发送第一个请求内容的最大间隔由参数max-read-idle决定，从第二个请求起，发送请求内容的最大间隔由参数max-keep-alive-idle决定。请求间的间隔超时也由max-keep-alive-idle决定。发送请求内容的总时间超时由参数read-timeout决定。Lighttpd与后端交互数据的超时由max-connection-idle决定。

延伸阅读：

http://www.snooda.com/read/244

**[ Nginx ]**

配置：nginx.conf

|  |
| --- |
| http {      #Fastcgi: (针对后端的fastcgi 生效, fastcgi 不属于proxy模式)      fastcgi\_connect\_timeout 5;    #连接超时      fastcgi\_send\_timeout 10;       #写超时      fastcgi\_read\_timeout 10;        #读取超时        #Proxy: (针对proxy/upstreams的生效)      proxy\_connect\_timeout 15s;    #连接超时      proxy\_read\_timeout 24s;          #读超时      proxy\_send\_timeout 10s;         #写超时  } |

**说明：**

Nginx 的超时设置倒是非常清晰容易理解，上面超时针对不同工作模式，但是因为超时带来的问题是非常多的。

延伸阅读：

http://hi.baidu.com/pibuchou/blog/item/a1e330dd71fb8a5995ee3753.html

http://hi.baidu.com/pibuchou/blog/item/7cbccff0a3b77dc60b46e024.html

http://hi.baidu.com/pibuchou/blog/item/10a549818f7e4c9df703a626.html

http://www.apoyl.com/?p=466

**【PHP本身超时处理】**

**[ PHP-fpm ]**

配置：php-fpm.conf

|  |
| --- |
| <?xml version="1.0" ?>  <configuration>  //...    Sets the limit on the number of simultaneous requests that will be served.    Equivalent to Apache MaxClients directive.    Equivalent to PHP\_FCGI\_CHILDREN environment in original php.fcgi    Used with any pm\_style.    #php-cgi的进程数量    <value name="max\_children">128</value>        The timeout (in seconds) for serving a single request after which the worker process will be terminated    Should be used when 'max\_execution\_time' ini option does not stop script execution for some reason    '0s' means 'off'   #php-fpm 请求执行超时时间，0s为永不超时，否则设置一个 Ns 为超时的秒数    <value name="request\_terminate\_timeout">0s</value>      The timeout (in seconds) for serving of single request after which a php backtrace will be dumped to slow.log file    '0s' means 'off'    <value name="request\_slowlog\_timeout">0s</value>    </configuration> |

**说明：**

在 php.ini 中，有一个参数 max\_execution\_time 可以设置 PHP 脚本的最大执行时间，但是，在 php-cgi(php-fpm) 中，该参数不会起效。真正能够控制 PHP 脚本最大执行时：

<value name="request\_terminate\_timeout">0s</value>

就是说如果是使用 mod\_php5.so 的模式运行 max\_execution\_time 是会生效的，但是如果是php-fpm模式中运行时不生效的。

延伸阅读：

http://blog.s135.com/file\_get\_contents/

**[ PHP ]**

配置：php.ini

选项：

max\_execution\_time = 30

或者在代码里设置：

ini\_set("max\_execution\_time", 30);

set\_time\_limit(30);

**说明：**

对当前会话生效，比如设置0一直不超时，但是如果php的 safe\_mode 打开了，这些设置都会不生效。

效果一样，但是具体内容需要参考php-fpm部分内容，如果php-fpm中设置了 request\_terminate\_timeout 的话，那么 max\_execution\_time 就不生效。

**【后端&接口访问超时】**

**【HTTP访问】**

一般我们访问HTTP方式很多，主要是：curl, socket, file\_get\_contents() 等方法。

如果碰到对方服务器一直没有响应的时候，我们就悲剧了，很容易把整个服务器搞死，所以在访问http的时候也需要考虑超时的问题。

[ CURL 访问HTTP]

CURL 是我们常用的一种比较靠谱的访问HTTP协议接口的lib库，性能高，还有一些并发支持的功能等。

CURL:

curl\_setopt($ch, opt) 可以设置一些超时的设置，主要包括：

\*(重要) CURLOPT\_TIMEOUT 设置cURL允许执行的最长秒数。

\*(重要) CURLOPT\_TIMEOUT\_MS 设置cURL允许执行的最长毫秒数。 (在cURL 7.16.2中被加入。从PHP 5.2.3起可使用。 )

CURLOPT\_CONNECTTIMEOUT 在发起连接前等待的时间，如果设置为0，则无限等待。

CURLOPT\_CONNECTTIMEOUT\_MS 尝试连接等待的时间，以毫秒为单位。如果设置为0，则无限等待。  在cURL 7.16.2中被加入。从PHP 5.2.3开始可用。

CURLOPT\_DNS\_CACHE\_TIMEOUT 设置在内存中保存DNS信息的时间，默认为120秒。

curl普通秒级超时：

        $ch = curl\_init();

        curl\_setopt($ch, CURLOPT\_URL,$url);

        curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, 1);

        curl\_setopt($ch, CURLOPT\_TIMEOUT, 60);   //只需要设置一个秒的数量就可以

        curl\_setopt($ch, CURLOPT\_HTTPHEADER, $headers);

        curl\_setopt($ch, CURLOPT\_USERAGENT, $defined\_vars['HTTP\_USER\_AGENT']);

curl普通秒级超时使用：

        curl\_setopt($ch, CURLOPT\_TIMEOUT, 60);

curl如果需要进行毫秒超时，需要增加：

        curl\_easy\_setopt(curl, CURLOPT\_NOSIGNAL, 1L);

        或者是：

        curl\_setopt ( $ch,  CURLOPT\_NOSIGNAL, true); 是可以支持毫秒级别超时设置的

curl一个毫秒级超时的例子：

|  |
| --- |
| <?php  if (!isset($\_GET['foo'])) {          // Client          $ch = curl\_init('http://example.com/');          curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, true);          curl\_setopt($ch, CURLOPT\_NOSIGNAL, 1);    //注意，毫秒超时一定要设置这个          curl\_setopt($ch, CURLOPT\_TIMEOUT\_MS, 200);  //超时毫秒，cURL 7.16.2中被加入。从PHP 5.2.3起可使用          $data = curl\_exec($ch);          $curl\_errno = curl\_errno($ch);          $curl\_error = curl\_error($ch);          curl\_close($ch);            if ($curl\_errno > 0) {                  echo "cURL Error ($curl\_errno): $curl\_error\n";          } else {                  echo "Data received: $data\n";          }  } else {          // Server          sleep(10);          echo "Done.";  }  ?> |

**其他一些技巧：**

1. 按照经验总结是：cURL 版本 >= libcurl/7.21.0 版本，毫秒级超时是一定生效的，切记。

2. curl\_multi的毫秒级超时也有问题。。单次访问是支持ms级超时的，curl\_multi并行调多个会不准

[流处理方式访问HTTP]

除了curl，我们还经常自己使用fsockopen、或者是file操作函数来进行HTTP协议的处理，所以，我们对这块的超时处理也是必须的。

一般连接超时可以直接设置，但是流读取超时需要单独处理。

自己写代码处理:

$tmCurrent = gettimeofday();

            $intUSGone = ($tmCurrent['sec'] - $tmStart['sec']) \* 1000000

                    + ($tmCurrent['usec'] - $tmStart['usec']);

            if ($intUSGone > $this->\_intReadTimeoutUS) {

                return false;

            }

或者使用内置流处理函数 stream\_set\_timeout() 和 stream\_get\_meta\_data() 处理：

|  |
| --- |
| <?php  // Timeout in seconds  $timeout = 5;  $fp = fsockopen("example.com", 80, $errno, $errstr, $timeout);  if ($fp) {          fwrite($fp, "GET / HTTP/1.0\r\n");          fwrite($fp, "Host: example.com\r\n");          fwrite($fp, "Connection: Close\r\n\r\n");          stream\_set\_blocking($fp, true);   //重要，设置为非阻塞模式          stream\_set\_timeout($fp,$timeout);   //设置超时          $info = stream\_get\_meta\_data($fp);          while ((!feof($fp)) && (!$info['timed\_out'])) {                  $data .= fgets($fp, 4096);                  $info = stream\_get\_meta\_data($fp);                  ob\_flush;                  flush();          }          if ($info['timed\_out']) {                  echo "Connection Timed Out!";          } else {                  echo $data;          }  } |

**file\_get\_contents** 超时：

|  |
| --- |
| <?php  $timeout = array(      'http' => array(          'timeout' => 5 //设置一个超时时间，单位为秒      )  );  $ctx = stream\_context\_create($timeout);  $text = file\_get\_contents("http://example.com/", 0, $ctx);  ?> |

**fopen**超时：

|  |
| --- |
| <?php  $timeout = array(      'http' => array(          'timeout' => 5 //设置一个超时时间，单位为秒      )  );  $ctx = stream\_context\_create($timeout);  if ($fp = fopen("http://example.com/", "r", false, $ctx)) {    while( $c = fread($fp, 8192)) {      echo $c;    }    fclose($fp);  }  ?> |

**【MySQL】**

php中的mysql客户端都没有设置超时的选项，mysqli和mysql都没有，但是libmysql是提供超时选项的，只是我们在php中隐藏了而已。

那么如何在PHP中使用这个操作捏，就需要我们自己定义一些MySQL操作常量，主要涉及的常量有：

MYSQL\_OPT\_READ\_TIMEOUT=11;

MYSQL\_OPT\_WRITE\_TIMEOUT=12;

这两个，定义以后，可以使用 options 设置相应的值。

不过有个注意点，mysql内部实现：

1. 超时设置单位为秒，最少配置1秒

2. 但mysql底层的read会重试两次，所以实际会是 3 秒

重试两次 +　自身一次 = 3倍超时时间，那么就是说最少超时时间是3秒，不会低于这个值，对于大部分应用来说可以接受，但是对于小部分应用需要优化。

查看一个设置访问mysql超时的php实例：

|  |
| --- |
| <?php  //自己定义读写超时常量  if (!defined('MYSQL\_OPT\_READ\_TIMEOUT')) {          define('MYSQL\_OPT\_READ\_TIMEOUT',  11);  }  if (!defined('MYSQL\_OPT\_WRITE\_TIMEOUT')) {          define('MYSQL\_OPT\_WRITE\_TIMEOUT', 12);  }  //设置超时  $mysqli = mysqli\_init();  $mysqli->options(MYSQL\_OPT\_READ\_TIMEOUT, 3);  $mysqli->options(MYSQL\_OPT\_WRITE\_TIMEOUT, 1);    //连接数据库  $mysqli->real\_connect("localhost", "root", "root", "test");  if (mysqli\_connect\_errno()) {     printf("Connect failed: %s/n", mysqli\_connect\_error());     exit();  }  //执行查询 sleep 1秒不超时  printf("Host information: %s/n", $mysqli->host\_info);  if (!($res=$mysqli->query('select sleep(1)'))) {      echo "query1 error: ". $mysqli->error ."/n";  } else {      echo "Query1: query success/n";  }  //执行查询 sleep 9秒会超时  if (!($res=$mysqli->query('select sleep(9)'))) {      echo "query2 error: ". $mysqli->error ."/n";  } else {      echo "Query2: query success/n";  }  $mysqli->close();  echo "close mysql connection/n";  ?> |

延伸阅读：

http://blog.csdn.net/heiyeshuwu/article/details/5869813

**【Memcached】**

**[PHP扩展]**

php\_memcache 客户端：

连接超时：bool Memcache::connect ( string $host [, int $port [, int $timeout ]] )

在get和set的时候，都没有明确的超时设置参数。

libmemcached 客户端：在php接口没有明显的超时参数。

说明：所以说，在PHP中访问Memcached是存在很多问题的，需要自己hack部分操作，或者是参考网上补丁。

**[C&C++访问Memcached]**

客户端：libmemcached 客户端

说明：memcache超时配置可以配置小点，比如5，10个毫秒已经够用了，超过这个时间还不如从数据库查询。

下面是一个连接和读取set数据的超时的C++示例：

|  |
| --- |
| //创建连接超时（连接到Memcached）  memcached\_st\* MemCacheProxy::\_create\_handle()  {          memcached\_st \* mmc = NULL;          memcached\_return\_t prc;          if (\_mpool != NULL) {  // get from pool            mmc = memcached\_pool\_pop(\_mpool, false, &prc);            if (mmc == NULL) {              \_\_LOG\_WARNING\_\_("MemCacheProxy", "get handle from pool error [%d]", (int)prc);            }            return mmc;          }            memcached\_st\* handle = memcached\_create(NULL);          if (handle == NULL){            \_\_LOG\_WARNING\_\_("MemCacheProxy", "create\_handle error");            return NULL;          }            // 设置连接/读取超时          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_HASH, MEMCACHED\_HASH\_DEFAULT);          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_NO\_BLOCK, \_noblock);  //参数MEMCACHED\_BEHAVIOR\_NO\_BLOCK为1使超时配置生效，不设置超时会不生效，关键时候会悲剧的，容易引起雪崩          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_CONNECT\_TIMEOUT, \_connect\_timeout);  //连接超时          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_RCV\_TIMEOUT, \_read\_timeout);    //读超时          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_SND\_TIMEOUT, \_send\_timeout);    //写超时          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_POLL\_TIMEOUT, \_poll\_timeout);            // 设置一致hash          //      memcached\_behavior\_set\_distribution(handle, MEMCACHED\_DISTRIBUTION\_CONSISTENT);          memcached\_behavior\_set(handle, MEMCACHED\_BEHAVIOR\_DISTRIBUTION, MEMCACHED\_DISTRIBUTION\_CONSISTENT);            memcached\_return rc;          for (uint i = 0; i < \_server\_count; i++){            rc = memcached\_server\_add(handle, \_ips[i], \_ports[i]);            if (MEMCACHED\_SUCCESS != rc) {              \_\_LOG\_WARNING\_\_("MemCacheProxy", "add server [%s:%d] failed.", \_ips[i], \_ports[i]);            }          }            \_mpool = memcached\_pool\_create(handle, \_min\_connect, \_max\_connect);          if (\_mpool == NULL){            \_\_LOG\_WARNING\_\_("MemCacheProxy", "create\_pool error");            return NULL;          }            mmc = memcached\_pool\_pop(\_mpool, false, &prc);          if (mmc == NULL) {            \_\_LOG\_WARNING\_\_("MyMemCacheProxy", "get handle from pool error [%d]", (int)prc);          }          //\_\_LOG\_DEBUG\_\_("MemCacheProxy", "get handle [%p]", handle);          return mmc;  }    //设置一个key超时（set一个数据到memcached）  bool MemCacheProxy::\_add(memcached\_st\* handle, unsigned int\* key, const char\* value, int len, unsigned int timeout)  {          memcached\_return rc;            char tmp[1024];          snprintf(tmp, sizeof (tmp), "%u#%u", key[0], key[1]);          //有个timeout值          rc = memcached\_set(handle, tmp, strlen(tmp), (char\*)value, len, **timeout**, 0);          if (MEMCACHED\_SUCCESS != rc){            return false;          }          return true;  } |

//Memcache读取数据超时 (没有设置)

libmemcahed 源码中接口定义：

LIBMEMCACHED\_API char \*memcached\_get(memcached\_st \*ptr,const char \*key, size\_t key\_length,size\_t \*value\_length,uint32\_t \*flags,memcached\_return\_t \*error);

LIBMEMCACHED\_API memcached\_return\_t memcached\_mget(memcached\_st \*ptr,const char \* const \*keys,const size\_t \*key\_length,size\_t number\_of\_keys);

从接口中可以看出在读取数据的时候，是没有超时设置的。

延伸阅读：

http://hi.baidu.com/chinauser/item/b30af90b23335dde73e67608

http://libmemcached.org/libMemcached.html

**【如何实现超时】**

程序中需要有超时这种功能，比如你单独访问一个后端Socket模块，Socket模块不属于我们上面描述的任何一种的时候，它的协议也是私有的，那么这个时候可能需要自己去实现一些超时处理策略，这个时候就需要一些处理代码了。

**[PHP中超时实现]**

**一、初级：最简单的超时实现 （秒级超时）**

思路很简单：链接一个后端，然后设置为非阻塞模式，如果没有连接上就一直循环，判断当前时间和超时时间之间的差异。

php socket 中实现原始的超时：(每次循环都当前时间去减，性能会很差，cpu占用会较高)

|  |
| --- |
| <?      $host = "127.0.0.1";      $port = "80";      $timeout = 15;  //timeout in seconds        $socket = socket\_create(AF\_INET, SOCK\_STREAM, SOL\_TCP)        or die("Unable to create socket\n");        socket\_set\_nonblock($socket)     //务必设置为阻塞模式        or die("Unable to set nonblock on socket\n");        $time = time();      //循环的时候每次都减去相应值      while (!@socket\_connect($socket, $host, $port))    //如果没有连接上就一直死循环      {        $err = socket\_last\_error($socket);        if ($err == 115 || $err == 114)        {          if ((time() - $time) >= $timeout)    //每次都需要去判断一下是否超时了          {            socket\_close($socket);            die("Connection timed out.\n");          }          sleep(1);          continue;        }        die(socket\_strerror($err) . "\n");      }      socket\_set\_block($this->socket)    //还原阻塞模式        or die("Unable to set block on socket\n");  ?> |

**二、升级：使用PHP自带异步IO去实现（毫秒级超时）**

**说明：**异步IO：异步IO的概念和同步IO相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。异步IO将比特分成小组进行传送，小组可以是8位的1个字符或更长。发送方可以在任何时刻发送这些比特组，而接收方从不知道它们会在什么时候到达。  
  
多路复用：复用模型是对多个IO操作进行检测，返回可操作集合，这样就可以对其进行操作了。这样就避免了阻塞IO不能随时处理各个IO和非阻塞占用系统资源的确定。

​

使用 socket\_select() 实现超时

socket\_select(..., floor($timeout), ceil($timeout\*1000000));

select的特点：能够设置到微秒级别的超时！

使用socket\_select() 的超时代码（需要了解一些异步IO编程的知识去理解）

|  |
| --- |
| ### 调用类 ####  <?php  $server = new Server;  $client = new Client;    for (;;) {    foreach ($select->can\_read(0) as $socket) {        if ($socket == $client->socket) {        // New Client Socket        $select->add(socket\_accept($client->socket));      }      else {        //there's something to read on $socket      }    }  }  ?>    ### 异步多路复用IO & 超时连接处理类 ###  <?php  class select {    var $sockets;      function select($sockets) {        $this->sockets = array();        foreach ($sockets as $socket) {        $this->add($socket);      }    }      function add($add\_socket) {      array\_push($this->sockets,$add\_socket);    }      function remove($remove\_socket) {      $sockets = array();        foreach ($this->sockets as $socket) {        if($remove\_socket != $socket)          $sockets[] = $socket;      }        $this->sockets = $sockets;    }      function can\_read($timeout) {      $read = $this->sockets;      socket\_select($read,$write = NULL,$except = NULL,$timeout);      return $read;    }      function can\_write($timeout) {      $write = $this->sockets;      socket\_select($read = NULL,$write,$except = NULL,$timeout);      return $write;    }  }  ?> |

**[C&C++中超时实现]**

一般在Linux C/C++中，可以使用：alarm() 设置定时器的方式实现秒级超时，或者：select()、poll()、epoll() 之类的异步复用IO实现毫秒级超时。也可以使用二次封装的异步io库（libevent, libev）也能实现。

**一、使用alarm中用信号实现超时 （秒级超时）**

说明：Linux内核connect超时通常为75秒，我们可以设置更小的时间如10秒来提前从connect中返回。这里用使用信号处理机制，调用alarm，超时后产生SIGALRM信号 （也可使用select实现）

用 alarym 秒级实现　connect 设置超时代码示例：

|  |
| --- |
| //信号处理函数  static void connect\_alarm(int signo)  {       debug\_printf("SignalHandler");       return;  }    //alarm超时连接实现  static void conn\_alarm()  {  　　Sigfunc \* sigfunc ; //现有信号处理函数  　　sigfunc=signal(SIGALRM, connect\_alarm); //建立信号处理函数connect\_alarm,(如果有)保存现有的信号处理函数      int timeout = 5;        //设置闹钟  　　if( alarm(timeout)!=0 ){  　　  //... 闹钟已经设置处理  　　}        //进行连接操作      if (connect(m\_Socket, (struct sockaddr \*)&addr, sizeof(addr)) < 0 ) {          if ( errno == EINTR ) { //如果错误号设置为EINTR，说明超时中断了              debug\_printf("Timeout");              m\_connectionStatus = STATUS\_CLOSED;              errno = ETIMEDOUT; //防止三次握手继续进行              return ERR\_TIMEOUT;          }          else {              debug\_printf("Other Err");              m\_connectionStatus = STATUS\_CLOSED;              return ERR\_NET\_SOCKET;          }      }  　　alarm(0);//关闭时钟  　　signal(SIGALRM, sigfunc); //(如果有)恢复原来的信号处理函数  　　return;  } |

//读取数据的超时设置

同样可以为 recv 设置超时，5秒内收不到任何应答就中断

signal( ... );

alarm(5);

recv( ... );

alarm(0);

static void sig\_alarm(int signo){return;}

当客户端阻塞于读(readline,...)时，如果此时服务器崩了，客户TCP试图从服务器接收一个ACK，持续重传 数据分节，大约要等9分钟才放弃重传，并返回一个错误。因此，在客户读阻塞时，调用超时。

​

二、使用异步复用IO使用 （毫秒级超时）

异步IO执行流程：

1.首先将标志位设为Non-blocking模式，准备在非阻塞模式下调用connect函数

2.调用connect，正常情况下，因为TCP三次握手需要一些时间；而非阻塞调用只要不能立即完成就会返回错误，所以这里会返回EINPROGRESS，表示在建立连接但还没有完成。

3.在读套接口描述符集(fd\_set rset)和写套接口描述符集(fd\_set wset)中将当前套接口置位（用FD\_ZERO()、FD\_SET()宏），并设置好超时时间(struct timeval \*timeout)

4.调用select( socket, &rset, &wset, NULL, timeout )

返回0表示connect超时，如果你设置的超时时间大于75秒就没有必要这样做了，因为内核中对connect有超时限制就是75秒。

//select 实现毫秒级超时示例：

|  |
| --- |
| static void conn\_select() {      // Open TCP Socket      m\_Socket = socket(PF\_INET,SOCK\_STREAM,0);      if( m\_Socket < 0 )      {          m\_connectionStatus = STATUS\_CLOSED;          return ERR\_NET\_SOCKET;      }        struct sockaddr\_in addr;      inet\_aton(m\_Host.c\_str(), &addr.sin\_addr);      addr.sin\_port = htons(m\_Port);      addr.sin\_family = PF\_INET;        // Set timeout values for socket      struct timeval timeouts;      timeouts.tv\_sec = SOCKET\_TIMEOUT\_SEC ;   // const -> 5      timeouts.tv\_usec = SOCKET\_TIMEOUT\_USEC ; // const -> 0      uint8\_t optlen = sizeof(timeouts);        if( setsockopt( m\_Socket, SOL\_SOCKET, SO\_RCVTIMEO,&timeouts,(socklen\_t)optlen) < 0 )      {          m\_connectionStatus = STATUS\_CLOSED;          return ERR\_NET\_SOCKET;      }        // Set the Socket to TCP Nodelay ( Send immediatly after a send / write command )      int flag\_TCP\_nodelay = 1;      if ( (setsockopt( m\_Socket, IPPROTO\_TCP, TCP\_NODELAY,              (char \*)&flag\_TCP\_nodelay, sizeof(flag\_TCP\_nodelay))) < 0)      {          m\_connectionStatus = STATUS\_CLOSED;          return ERR\_NET\_SOCKET;      }      // Save Socket Flags      int opts\_blocking = fcntl(m\_Socket, F\_GETFL);      if ( opts\_blocking < 0 )      {          return ERR\_NET\_SOCKET;      }      //设置为非阻塞模式      int opts\_noblocking = (opts\_blocking | O\_NONBLOCK);      // Set Socket to Non-Blocking      if (fcntl(m\_Socket, F\_SETFL, opts\_noblocking)<0)      {          return ERR\_NET\_SOCKET;      }      // Connect      if ( connect(m\_Socket, (struct sockaddr \*)&addr, sizeof(addr)) < 0)      {          // EINPROGRESS always appears on Non Blocking connect          if ( errno != EINPROGRESS )          {              m\_connectionStatus = STATUS\_CLOSED;              return ERR\_NET\_SOCKET;          }          // Create a set of sockets for select          fd\_set socks;          FD\_ZERO(&socks);          FD\_SET(m\_Socket,&socks);          // Wait for connection or timeout          int fdcnt = select(m\_Socket+1,NULL,&socks,NULL,&timeouts);          if ( fdcnt < 0 )          {              return ERR\_NET\_SOCKET;          }          else if ( fdcnt == 0 )          {              return ERR\_TIMEOUT;          }      }      //Set Socket to Blocking again      if(fcntl(m\_Socket,F\_SETFL,opts\_blocking)<0)      {          return ERR\_NET\_SOCKET;      }        m\_connectionStatus = STATUS\_OPEN;      return 0;  } |

说明：在超时实现方面，不论是什么脚本语言：PHP、Python、Perl 基本底层都是C&C++的这些实现方式，需要理解这些超时处理，需要一些Linux 编程和网络编程的知识。

延伸阅读：

http://blog.sina.com.cn/s/blog\_4462f8560100tvgo.html

http://blog.csdn.net/thimin/article/details/1530839

http://hi.baidu.com/xjtdy888/item/93d9daefcc1d31d1ea34c992

http://blog.csdn.net/byxdaz/article/details/5461142

http://blog.163.com/xychenbaihu@yeah/blog/static/13222965520112163171778/

http://hi.baidu.com/suyupin/item/df10004decb620e91f19bcf5

http://stackoverflow.com/questions/7092633/connect-timeout-with-alarm

http://stackoverflow.com/questions/7089128/linux-tcp-connect-with-select-fails-at-testserver?lq=1

http://cppentry.com/bencandy.php?fid=54&id=1129

**【 总结 】**

**1. PHP应用层如何设置超时?**

PHP在处理超时层次有很多，不同层次，需要前端包容后端超时：

浏览器（客户端） -> 接入层 -> Web服务器  -> PHP  -> 后端 (MySQL、Memcached）

就是说，接入层（Web服务器层）的超时时间必须大于PHP（PHP-FPM）中设置的超时时间，不然后面没处理完，你前面就超时关闭了，这个会很杯具。还有就是PHP的超时时间要大于PHP本身访问后端（MySQL、HTTP、Memcached）的超时时间，不然结局同前面。

**2. 超时设置原则是什么？**

如果是希望永久不超时的代码（比如上传，或者定期跑的程序），我仍然建议设置一个超时时间，比如12个小时这样的，主要是为了保证不会永久夯住一个php进程或者后端，导致无法给其他页面提供服务，最终引起所有机器雪崩。

如果是要要求快速响应的程序，建议后端超时设置短一些，比如连接500ms，读1s，写1s，这样的速度，这样能够大幅度减少应用雪崩的问题，不会让服务器负载太高。

**3. 自己开发超时访问合适吗？**

一般如果不是万不得已，建议用现有很多网络编程框架也好、基础库也好，里面一般都带有超时的实现，比如一些网络IO的lib库，尽量使用它们内置的，自己重复造轮子容易有bug，也不方便维护（不过如是是基于学习的目的就当别论了）。

**4. 其他建议**

超时在所有应用里都是大问题，在开发应用的时候都要考虑到。我见过一些应用超时设置上百秒的，这种性能就委实差了，我举个例子：

比如你php-fpm开了128个php-cgi进程，然后你的超时设置的是32s，那么我们如果后端服务比较差，极端情况下，那么最多每秒能响应的请求是：

128 / 32 = 4个

你没看错，1秒只能处理4个请求，那服务也太差了！虽然我们可以把php-cgi进程开大，但是内存占用，还有进程之间切换成本也会增加，cpu呀，内存呀都会增加，服务也会不稳定。所以，尽量设置一个合理的超时值，或者督促后端提高性能。