

# 【原创】PHP 超时处理全面总结

作者: heiyeluren (黑夜路人)

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/](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  $\geq$  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/heyeshuwu/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)

libmemcached 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 alarm 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.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。