Memcached技术

**[1](#_Toc18885)** [概述 3](#_Toc18885)

**[2](#_Toc17756)** [安装 4](#_Toc17756)

**[2.1](#_Toc11765)** [windows 4](#_Toc11765)

[2.1.1 安装 4](#_Toc12351)

[2.1.2 可以安装成windows的服务 4](#_Toc19085)

[2.1.3 将memcached目录配置到环境变量PATH 5](#_Toc22479)

**[2.2](#_Toc7286)** [linux 6](#_Toc7286)

[2.2.1 基础依赖 7](#_Toc18025)

[2.2.2 libevent, memcache所基于的事件驱动程序 7](#_Toc26954)

[2.2.3 安装memcached 7](#_Toc17707)

[2.2.4 配置环境变量 8](#_Toc12273)

[2.2.5 开机自动启动 8](#_Toc20627)

**[3](#_Toc30529)** [C/S架构, 客户端/服务器端 9](#_Toc30529)

**[3.1](#_Toc10027)** [memcached服务器端 10](#_Toc10027)

**[3.2](#_Toc12925)** [客户端 10](#_Toc12925)

**[3.3](#_Toc27270)** [PHP安装memcache扩展 13](#_Toc27270)

[3.3.1 windows 13](#_Toc29412)

[3.3.2 linux 16](#_Toc19725)

**[4](#_Toc5814)** [操作 - PHP 19](#_Toc5814)

**[4.1](#_Toc23421)** [Key-Value型存储 20](#_Toc23421)

**[4.2](#_Toc26964)** [memcache提供的OOP语法 20](#_Toc26964)

**[4.3](#_Toc9982)** [实例化Memcache对象, 连接memcached服务器 20](#_Toc9982)

**[4.4](#_Toc1379)** [操作memcached服务器 20](#_Toc1379)

[4.4.1 set(key, value), 设置 21](#_Toc32055)

[4.4.2 get(key), 获取 21](#_Toc4388)

[4.4.3 delete(key), 删除 21](#_Toc1491)

**[4.5](#_Toc28246)** [设置 21](#_Toc28246)

[4.5.1 key 21](#_Toc6475)

[4.5.2 value 22](#_Toc608)

[4.5.3 flag标志 24](#_Toc761)

[4.5.4 有效期 25](#_Toc8340)

[4.5.5 添加 - add() 26](#_Toc12476)

[4.5.6 替换 - replace() 26](#_Toc12237)

[4.5.7 递增 - increment() 27](#_Toc9391)

[4.5.8 递减 - decrement() 27](#_Toc11702)

**[4.6](#_Toc25037)** [获取 – get 27](#_Toc25037)

**[4.7](#_Toc1395)** [删除 – delete 27](#_Toc1395)

**[4.8](#_Toc4722)** [清空(刷新) – flush 28](#_Toc4722)

**[4.9](#_Toc7311)** [关闭连接 – close() 28](#_Toc7311)

**[5](#_Toc24598)** [telnet操作 28](#_Toc24598)

**[5.1](#_Toc20569)** [支持以上全部PHP的命令 29](#_Toc20569)

**[5.2](#_Toc28089)** [stats – 获取当前的服务器状态 29](#_Toc28089)

**[6](#_Toc12047)** [并发处理 - 乐观锁 – CAS(Check && Set) 30](#_Toc12047)

**[7](#_Toc13016)** [分布式部署 32](#_Toc13016)

**[7.1](#_Toc21392)** [实现 33](#_Toc21392)

**[7.2](#_Toc8931)** [分布式算法 34](#_Toc8931)

[7.2.1 求余 35](#_Toc16231)

[7.2.2 一致性哈希算法 36](#_Toc11662)

**[8](#_Toc11890)** [概念 39](#_Toc11890)

**[8.1](#_Toc18030)** [懒惰模式 40](#_Toc18030)

**[8.2](#_Toc30830)** [LRU, Least Recently Use 最近最少使用 40](#_Toc30830)

**[9](#_Toc22212)** [应用 41](#_Toc22212)

**[9.1](#_Toc30764)** [分类表 42](#_Toc30764)

**[9.2](#_Toc21097)** [后台的分类列表 42](#_Toc21097)

[9.2.1 控制器-动作, back/Category/list 42](#_Toc27972)

[9.2.2 模型 43](#_Toc12588)

[9.2.3 视图-模板 43](#_Toc13434)

**[9.3](#_Toc22572)** [缓存系统 44](#_Toc22572)

[9.3.1 查询 44](#_Toc21557)

[9.3.2 更新(删除)缓存 45](#_Toc6895)

# 概述

内存缓存服务器.

memory cache

最常规的一款内存缓存产品. 类似的产品, Xcache, APC, …

当应用程序, 需要将数据缓存在一个快速存储区时, 选择使用内存.

对于PHP这类的语言, 不能直接操作内存. 需要借助与第三方的内存管理器来实现.



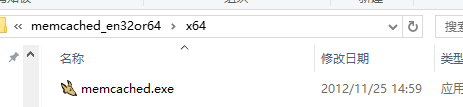
充当缓存的容器, 不仅仅是内存, 所有的存储器, 都可以作为缓存容器. 内存最有速度优势!

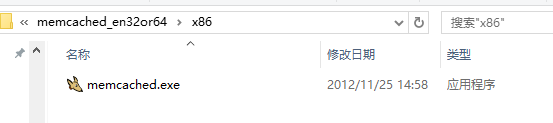
# 安装

## windows

### 安装

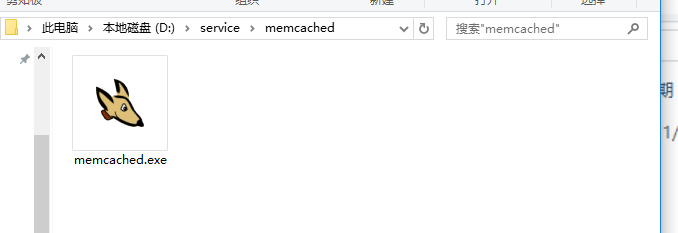
提供了可以直接运行的执行程序:





将其拷贝的固定的工作目录:

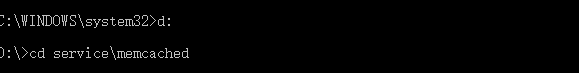
我的演示: d:/service/memcached



### 可以安装成windows的服务

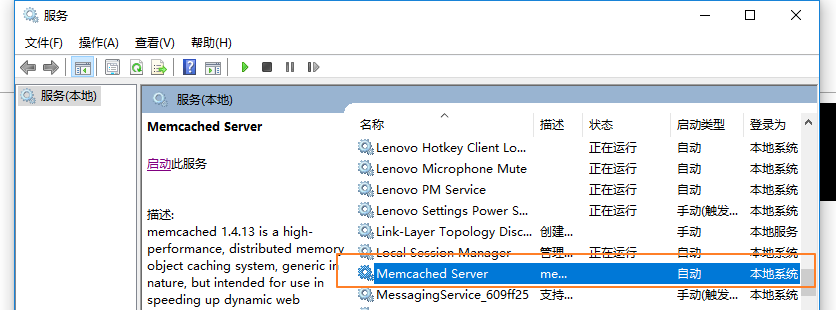
使用管理员身份运行CMD

进入memcached目录



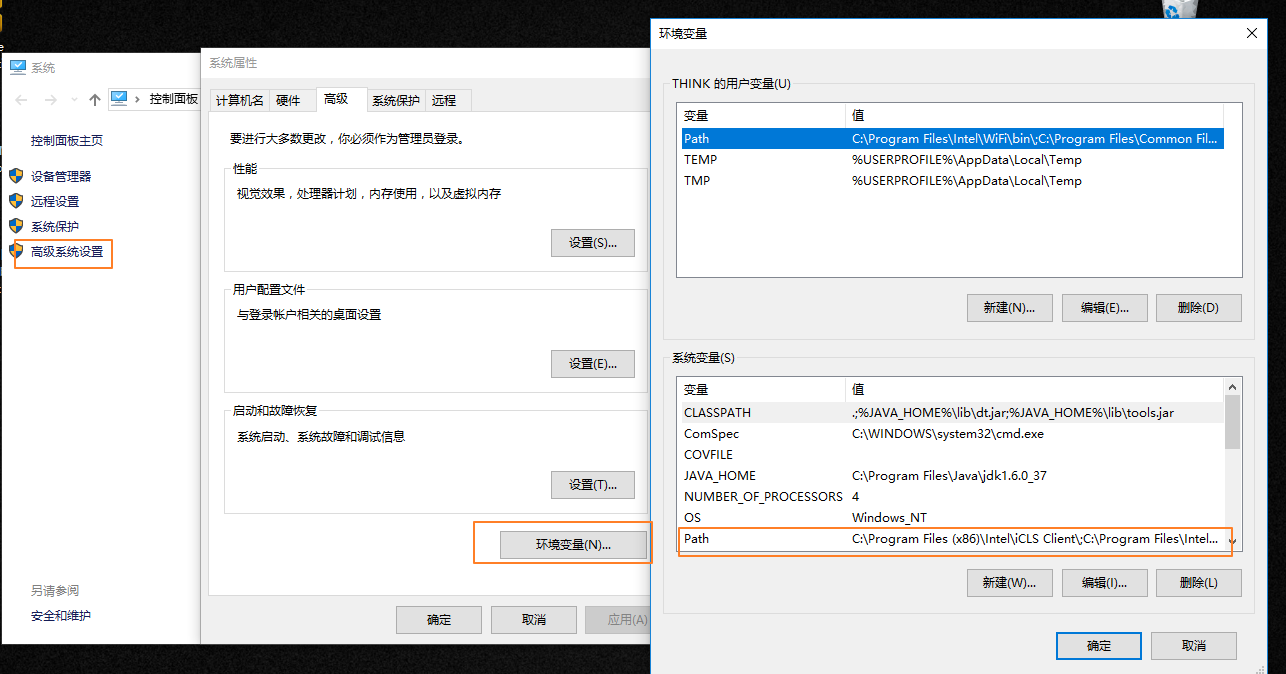
-d install 选项

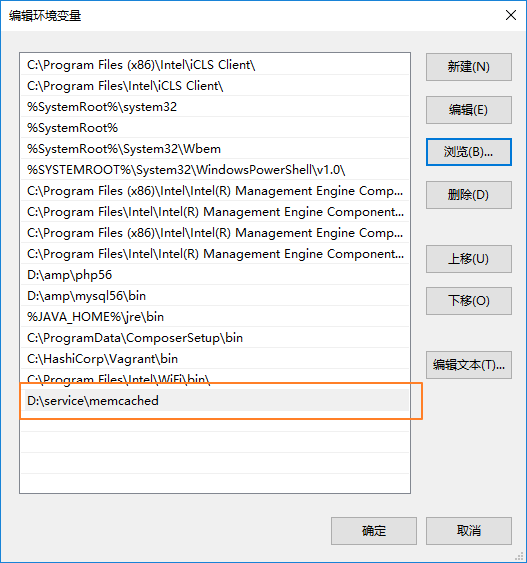


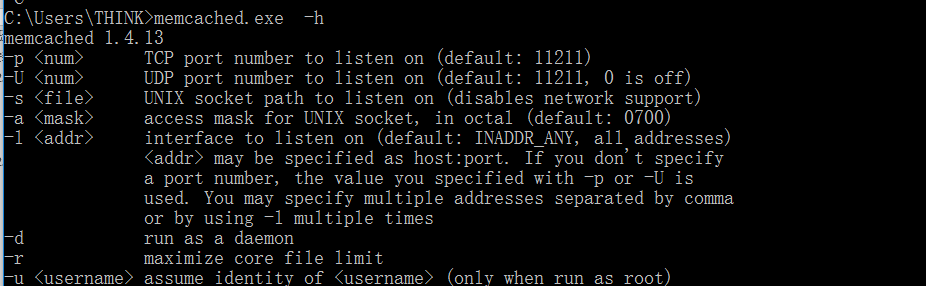


### 将memcached目录配置到环境变量PATH

便于, 在CMD运行管理







## linux

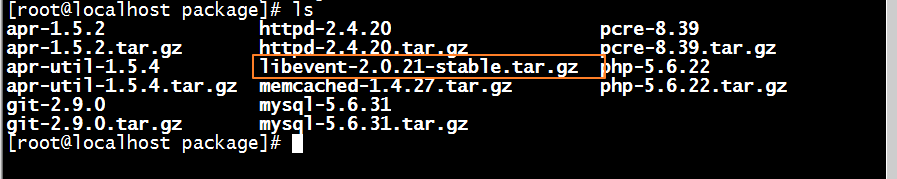
### 基础依赖

mount /dev/cdrom /media/CentOS/

yum --disablerepo=\\* --enablerepo=c6-media install gcc gcc-c++

### libevent, memcache所基于的事件驱动程序

上传到linux



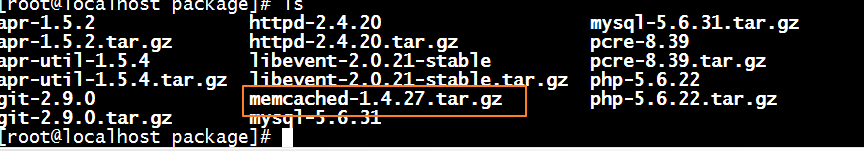
解压 进入 配置 编译 安装





### 安装memcached

上传源码到linux



解压, 进入

配置, 编译, 安装

配置时, 需要指明libevent的所在的目录, 使用选项—with-libevent做到

./configure --prefix=/usr/local/memcached --with-libevent=/usr/local/libevent && make && make install



### 配置环境变量

vi /etc/profile



source



### 开机自动启动

将启动的命令加入到 /etc/rc.local

没有提供支持守护进程的脚本!

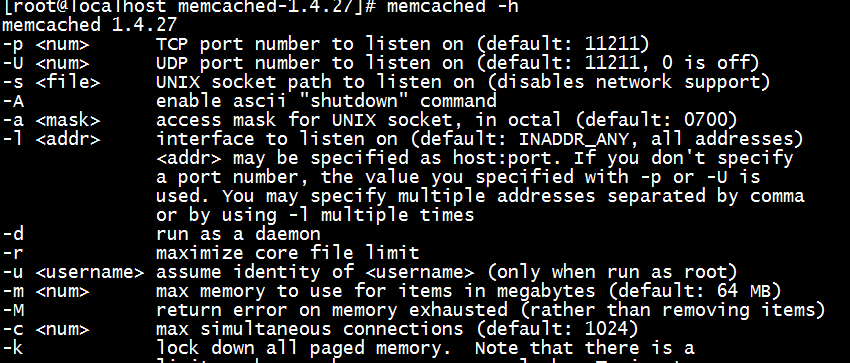
vi /etc/rc.local



# C/S架构, 客户端/服务器端

## memcached服务器端

一直守护运行, 保证客户端可以随时访问到!



运行时, 通常需要设置一些参数:

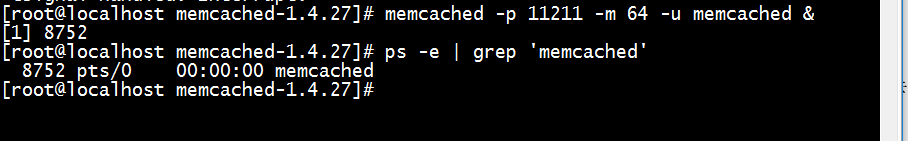
-m, memcached所管理的内存最大值

-p, 所监听的端口port

-l, 所监听的IP地址, 在一台电脑存在多块网卡的情况, 仅仅监听那块网卡进入的请求.

-u, 在linux下运行时, 不准以root运行, 使用-u切换到其他用户.

运行memcached服务器(win&linux通用)



## 客户端

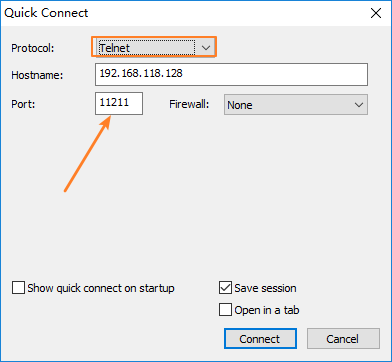
memcached软件, 没有自带客户端!

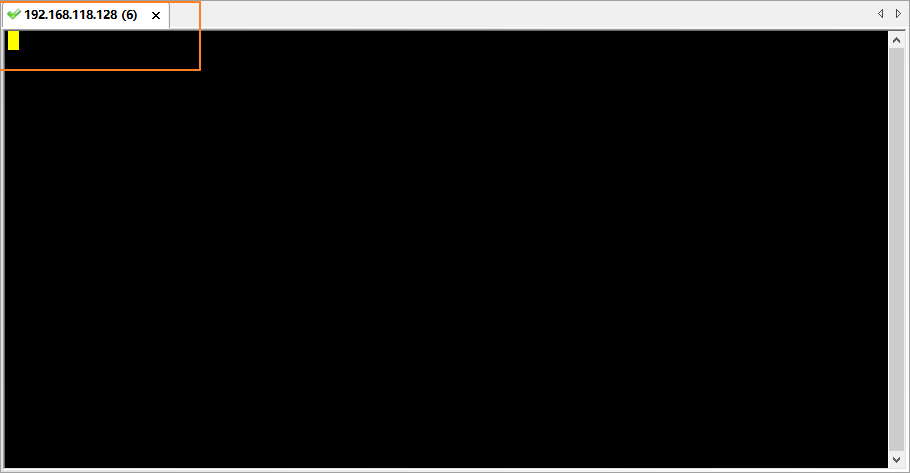
需要使用第三方的客户端.

最常见的客户端, 就是PHP程序

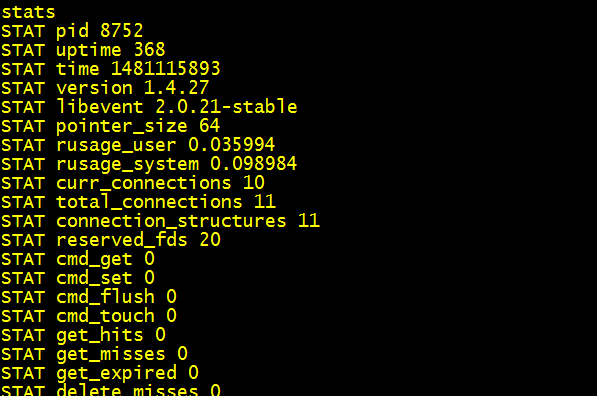


除了PHP程序外, 凡是支持 telnet远程登录协议的, 客户端, 都可以作为memcached的客户端使用. 例如 secureCRT软件, 就支持telnet





一般在调试, 测试时, 会使用到这种telnet客户端!



## PHP安装memcache扩展

PHP要操作memcached, 就必须要安装支持memcached的扩展才可以

使用PHP的memcache的扩展, 完成对memcached服务器的操作.

### windows

windows下, 是没自带memcache扩展

在php.ini中, 没有提供好的载入扩展的配置

在php/ext扩展文件目录中, 没有php\_memcache.dll这个扩展文件

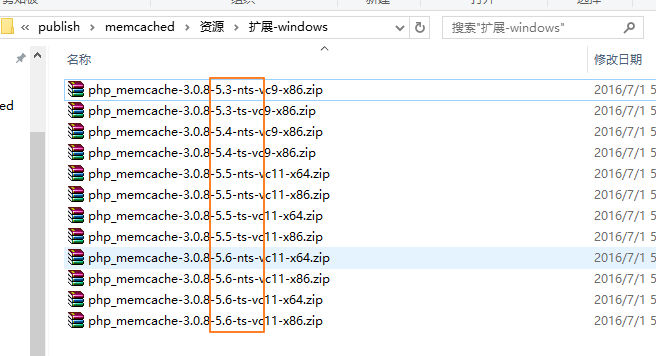
#### 找到合适的Php\_memcache.dll, 放入到php/ext扩展目录中:

我们需要根据:

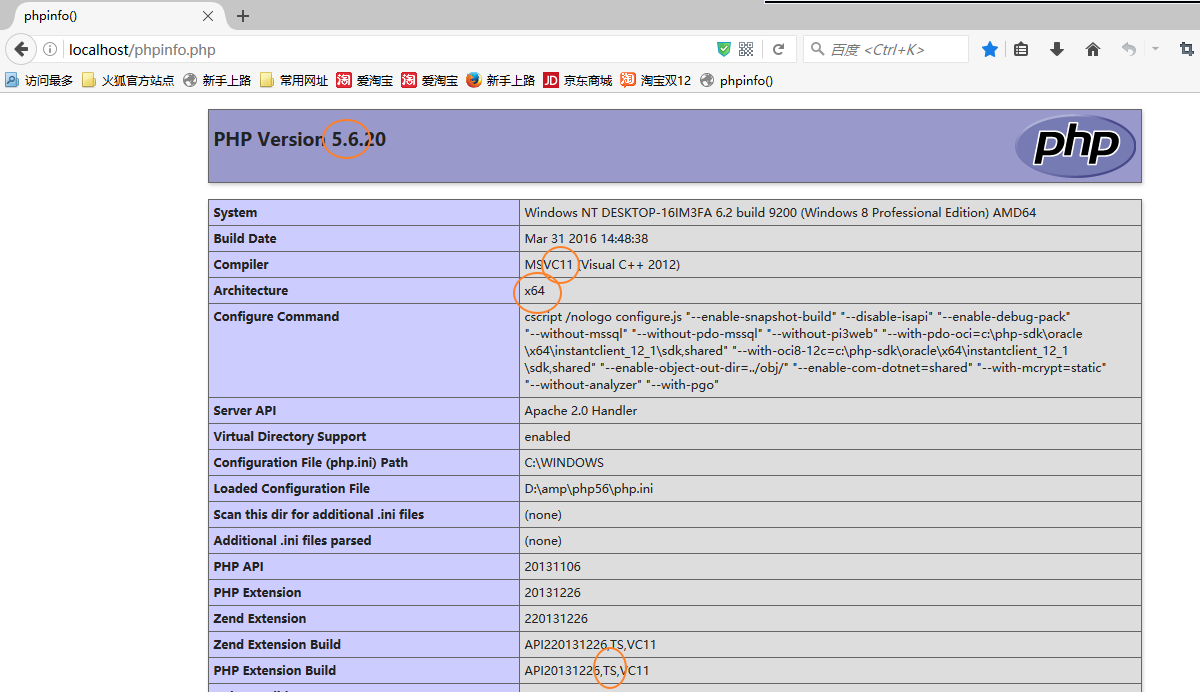
PHP版本, TS|NTS, 32or64位 进行选择.

ts, 线程安全

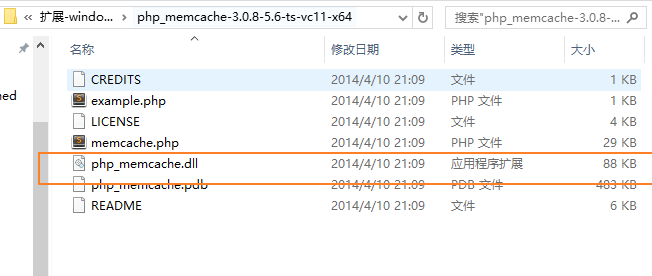
nts, 非线程安装



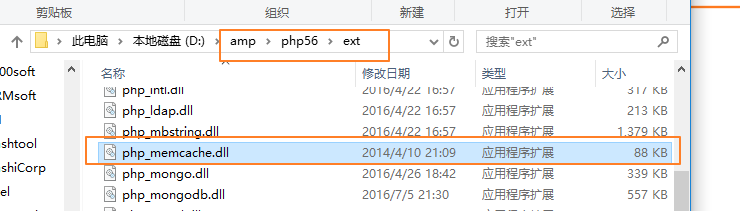
根据PHPinfo进行选项:



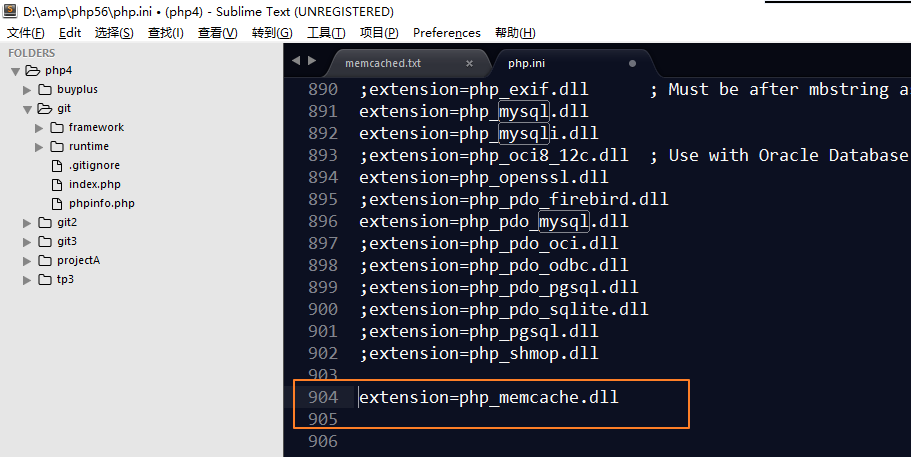
解压后获取dll文件



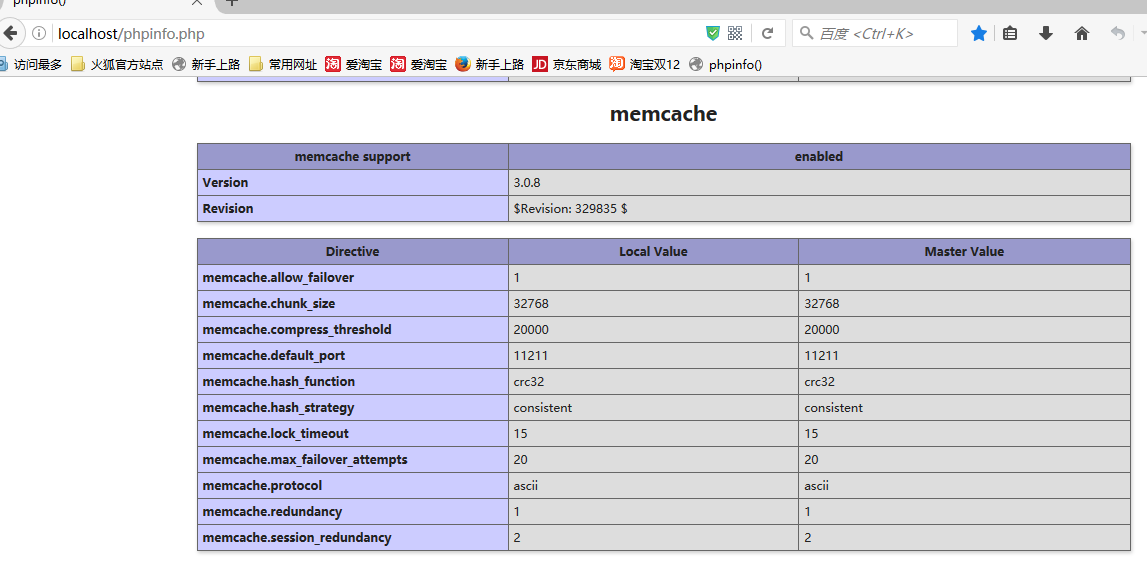
放置在 php/ext目录



#### 修改 php.ini



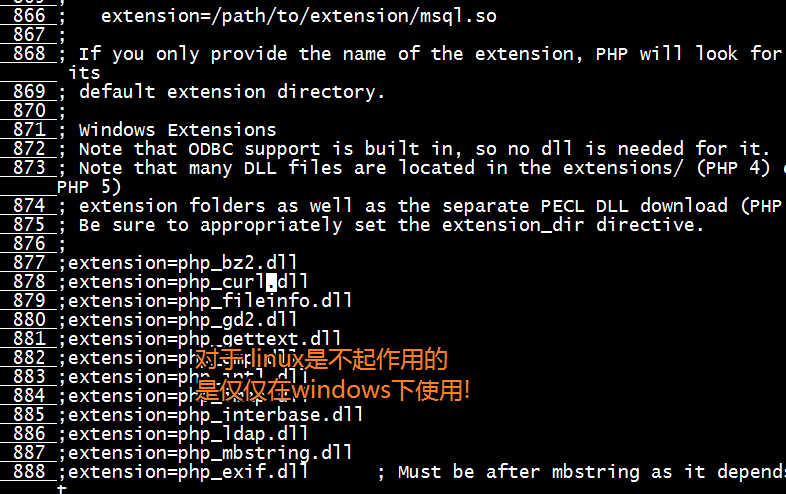
重启, 看phpinfo界面, 出现memcache才成功!



### linux

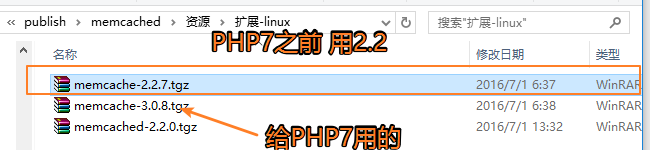
linux下的php.ini

vi /usr/local/php/lib/php.ini

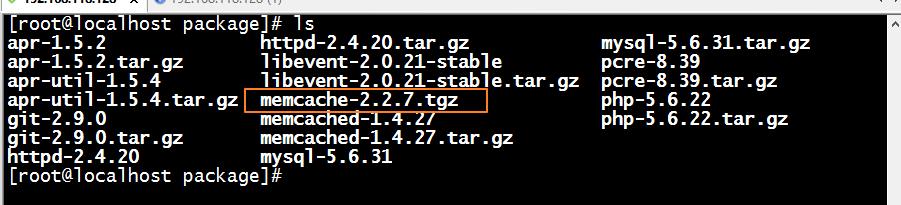


在linux下, 扩展的后缀不是.dll, 而是.so

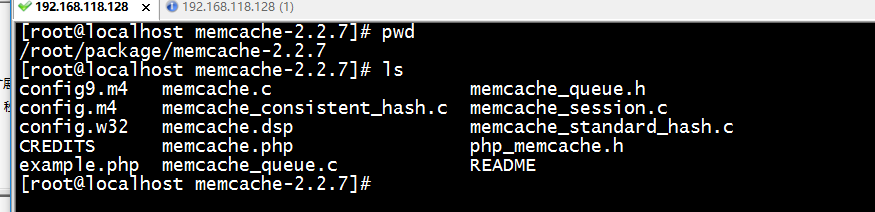
#### 提供了扩展的源码, 源码编译安装即可



#### 源码上传

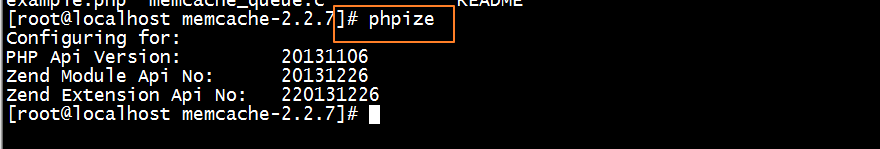


#### 解压, 进入

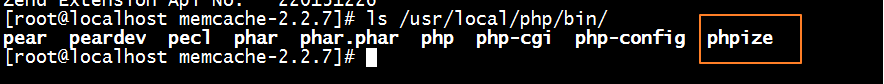


#### phpize配置成PHP的扩展

phpize, 是PHP的工具, 用来将PHP的扩展与PHP程序建立关联.



phpize在php/bin下可以找到

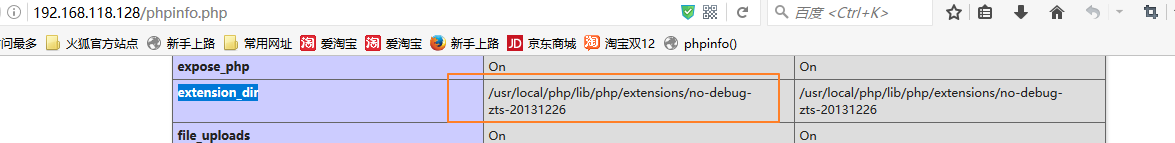


#### 配置编译安装

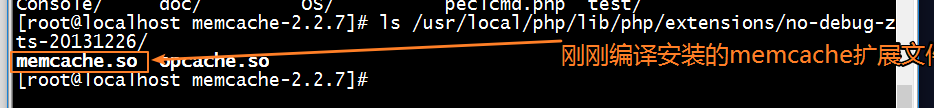
./configure && make && make install



自动生成 扩展文件, 并放置在PHP的扩展目录:

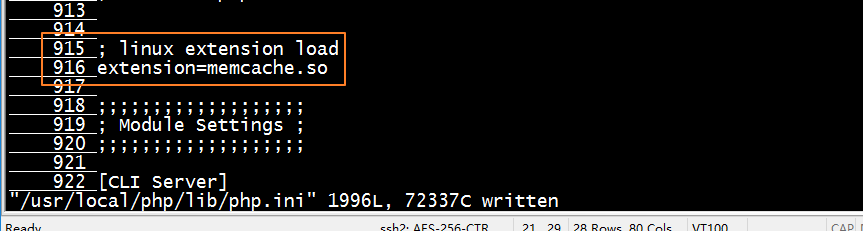


查看该目录:

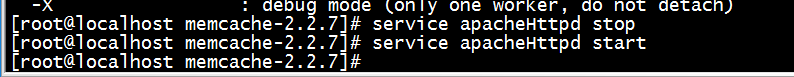


#### 修改php.ini

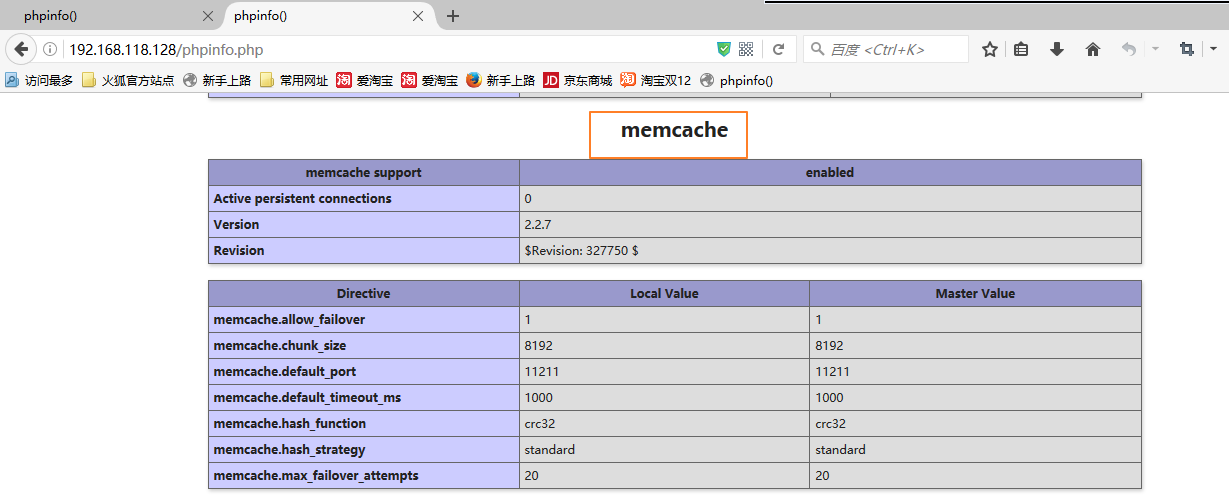
vi /usr/local/php/lib/php.ini



重启, 查看结果



安装成功!~



# 操作 - PHP

## Key-Value型存储

memcached 数据模式, 是键值对的.

一个key对应一个值

(PHP的关联数组类似)

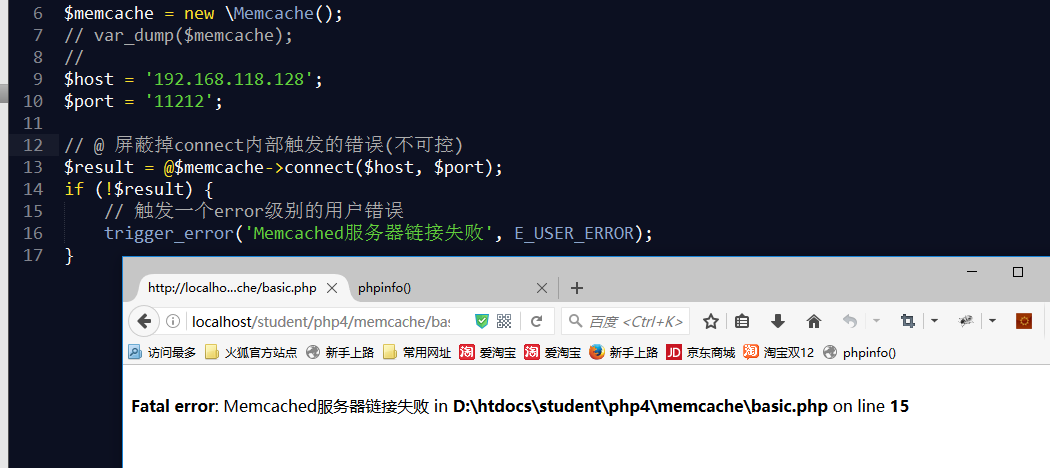
## memcache提供的OOP语法

语法上, 实例化对象, 操作属性和方法, 完成工作!

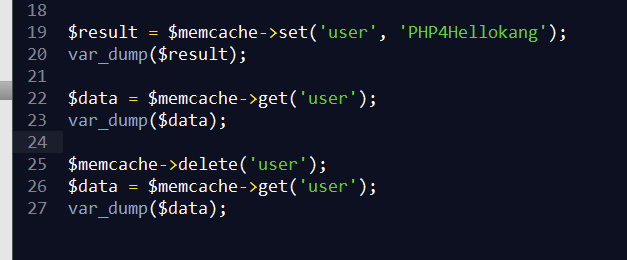
## 实例化Memcache对象, 连接memcached服务器

实操代码, 连接失败, 报告错误:

使用trigger\_error()触发用户级别的错误



## 操作memcached服务器



### set(key, value), 设置

### get(key), 获取

### delete(key), 删除

## 设置

需要指定缓存项的key和value.

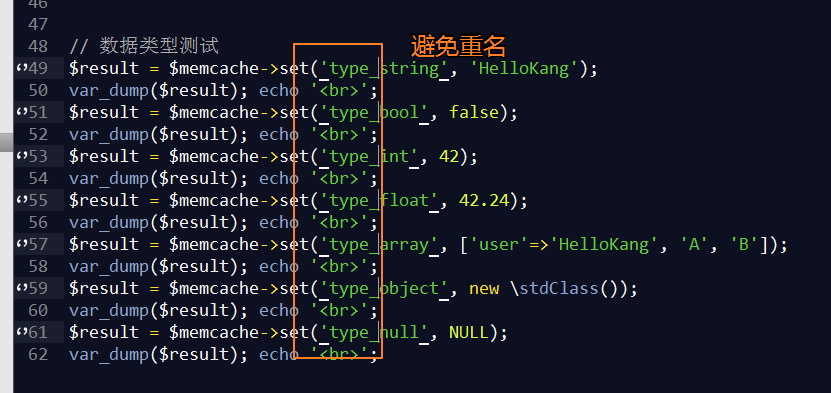
除此之外, 还需要指定, 2个属性, 标志和有效期

key, value, 标志, 有效期

bool **Memcache::set** ( string $key , [mixed](mk:@MSITStore:D:\\document\\php_enhanced_zh.chm::/res/language.pseudo-types.html" \l "language.types.mixed) $var [, int $flag [, int $expire ]] )

### key

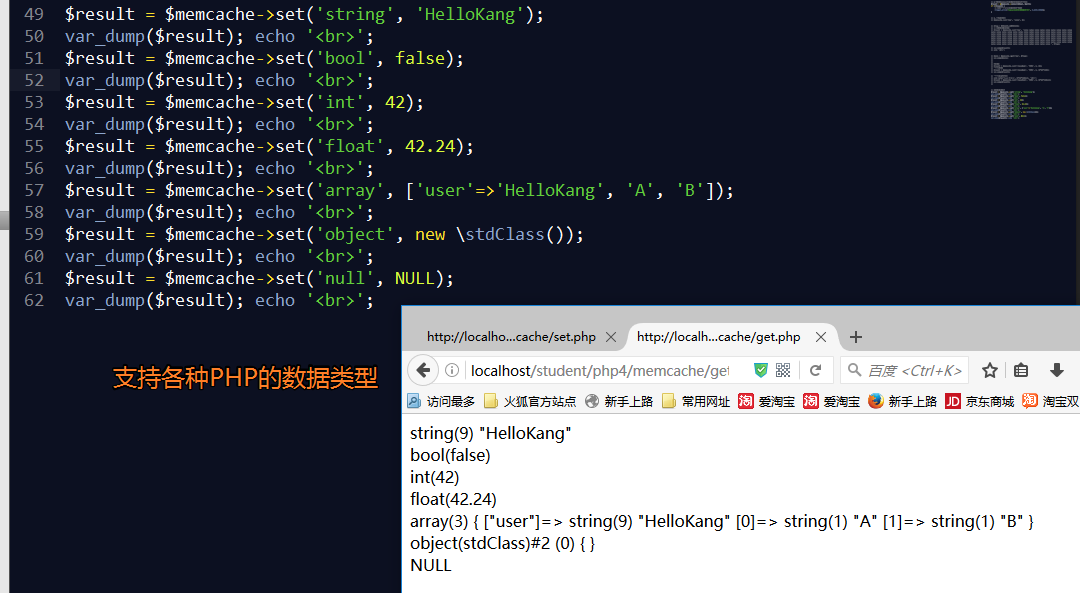
实操中, 通常会使用key前缀



### value

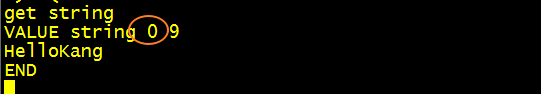
最大长度为1M, 如果非常大, 选择. 分段或者压缩试试.

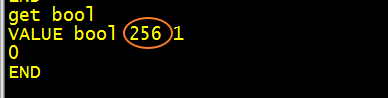
支持多种PHP的数据类型

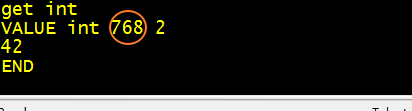


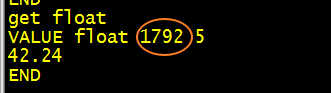
也是使用不同的flag标志做到的:

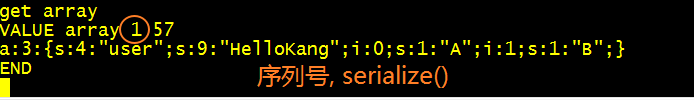
通过telnet测试:















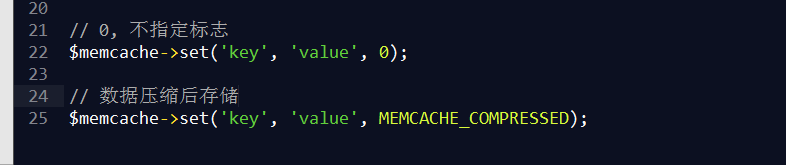
存储时, 对于不同的类型做不同的处理, 设置对应的标志.

获取时, 通过标志判断, 存储是的操作, 做逆向操作即可!

### flag标志

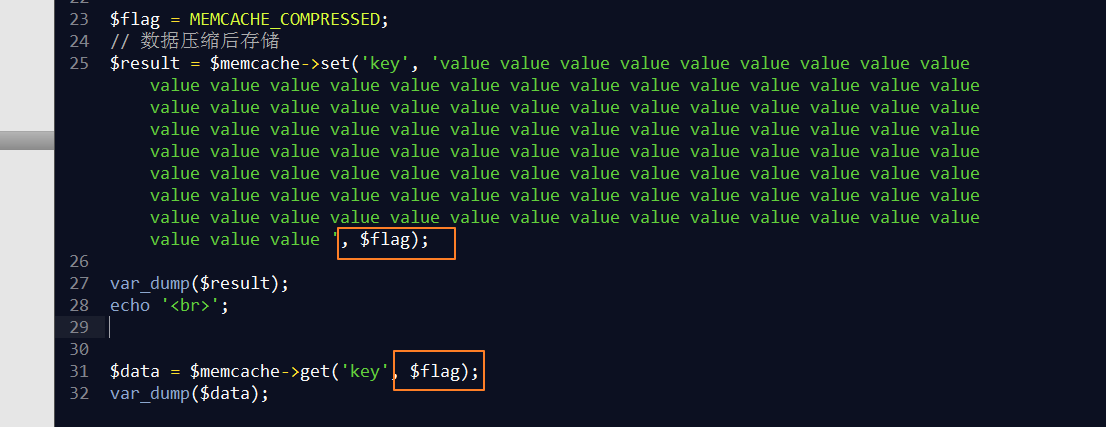
标志, memcache扩展内部使用的. 我们通常不需要处理. 基本上都是0.

可能会用到一个值: **MEMCACHE\_COMPRESSED(2),** 表示数据需要压缩处理.将数据压缩后, 再存储到memcached服务器中:



一旦设置了标志, 需要获取时, 做相应的处理:

演示 压缩存储数据:



通过 telnet测试查看:



标志的执行流程时:



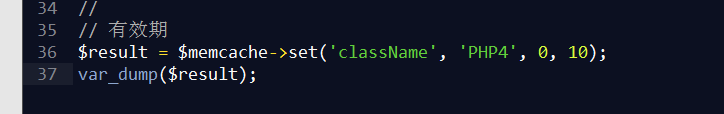
### 有效期

表示缓存项的有效期.

默认0, 不写, 表示永久有效. 除非主动删除, 否则一直都在!

写, 整数表示时间间隔和时间戳, 表表示 在什么时候过期:

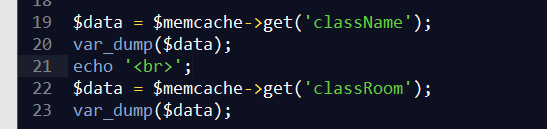
10秒后失效

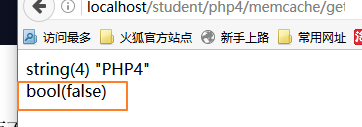


当整数值大于: 24\*30\*3600=2592000. 整数就表示的是时间戳, 而不是时间间隔:



获取不到, 已经过期:





实操中:

固定长度的有效期,就是时间间隔

在某个时间点失效, 使用时间戳.

### 添加 - add()

与set一致, 仅仅可以执行添加操作, 不能执行修改操作.

当key已经存在时, 则add失败



if (!$mem->get(key)) {

return $mem->set(key, value)

} else {

return false;

}

### 替换 - replace()

与set一致, 仅仅可以执行替换操作. 仅仅在key存在时, 才可以执行.

当可以, 不存在时, 替换是失败的.

逻辑:

if ($mem->get(key)) {

return $mem->set(key, value);

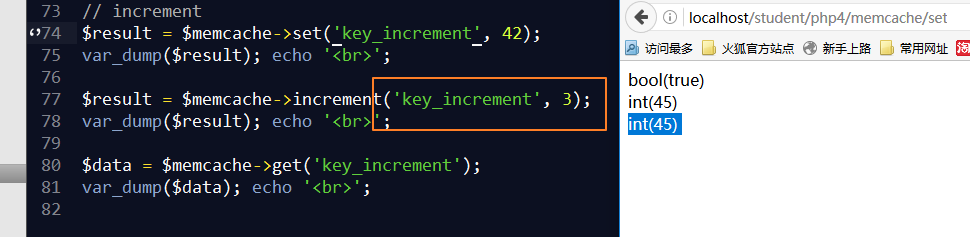
} else {

return false;

}

### 递增 - increment()

在原有值的基础上, 增加



第二个参数默认是1.

### 递减 - decrement()

在原有值的基础上, 减少.

## 获取 – get

获取时, 有时需要设置第二个参数, flag标志!

## 删除 – delete

指定key删除缓存项

delete(key)

## 清空(刷新) – flush

删除全部缓存项

$memcache->flush();

## 关闭连接 – close()

使用完毕应该尽快断开连接. 一台memcached服务器的支持的链接数量是有限的, 默认是: 1024个.



# telnet操作

## 支持以上全部PHP的命令

有些命令是语法不一致

不一致的有:

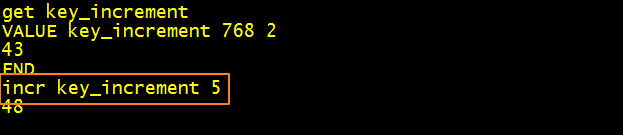
set:



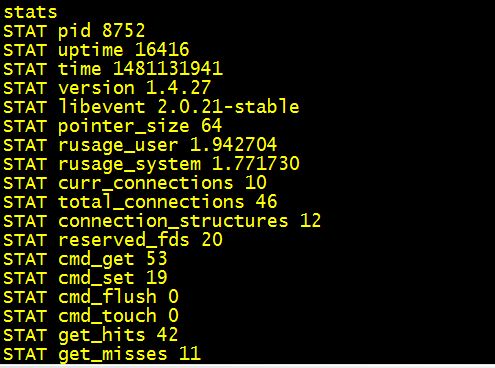
需要指定长度. 保证数据长度与给定的长度一致!

incr

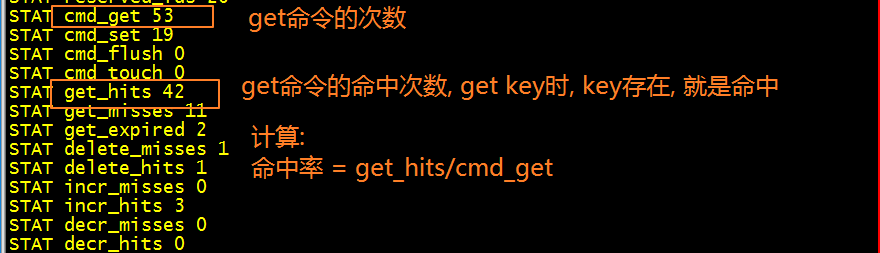
decr



## stats – 获取当前的服务器状态



其中, 使用的角度, 最关心的状态指标: 命中率



# 并发处理 - 乐观锁 – CAS(Check && Set)

在新版本的memcached中, 增加对并发的控制. 处理方案是:乐观锁.

并发: 多个进程(连接), 同时操作一个key. 就是并发操作.

乐观锁:

进程A, 先操作了缓存项

在进程A第二次操作缓存项前, 进程B操作了缓存项.

之后, 进程A第二次操作缓存项. 检查, 在进程A第一次操作后, 是否有其他进程操作过需要的缓存项. 如果有, 则放弃第二次操作. 采取乐观的处理态度.(乐观锁定)

支持乐观锁的操作:

gets(获取)

cas(设置, check and set)

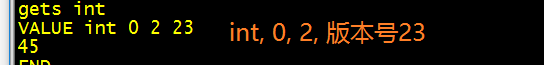
(PS: memcache还不支持这两个操作, 在telnet上演示)

进程A, 设置一个缓存项

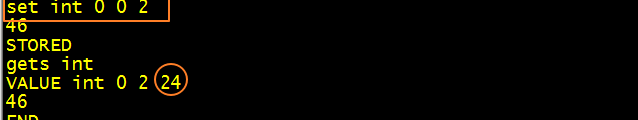
检查缓存的版本, 使用gets 进行获取

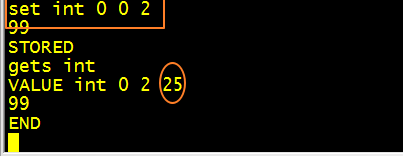
gets与get一致, 多返回一个数据, 就是缓存项的版本号:

版本号, 是memcached内部记录的, 通过gets可以获取到:



每当设置一次, 版本号, 就是累加:





基于版本号, check and set机制就成为了可能.

进程A, 操作某个缓存项



在进程A, 第二次操作(设置)缓存项int之前, 进程B(另外的链接), 操作了 int这个缓存项



当进程B, 操作完毕后,进程A, 又想继续操作缓存项int:

准备将值设置为 56. 要去判断, 该key, 是否在获取后, 这段时期被别人操作过:

命令 cas 即可完成: 与set一致, 需要传递, 一个版本号, 用于检测check.

如果发现, 更新的版本, 则放弃本次设置工作.



# 分布式部署

## 实现

分布式: 将数据分散到不同的存储中进行存储.



分布式部署的实现, 由分布式算法, 和 分布式缓存服务器集群构成.

分布式算法:

key与某台memcached服务器映射的算法.

分布式缓存服务器集群, 多台memcached而已.

其中, 分布式算法, 由 客户端实现的. 而不是由memcached服务器实现. 称之为(服务器间)互不通讯的分布式架构.

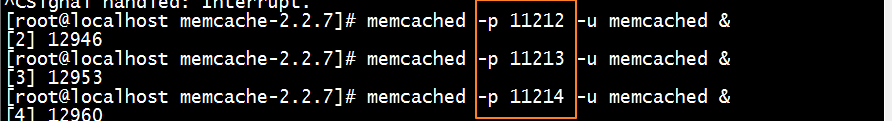
(上图中, memcached1-4, 之间毫无联系, 不知道对方的存在)

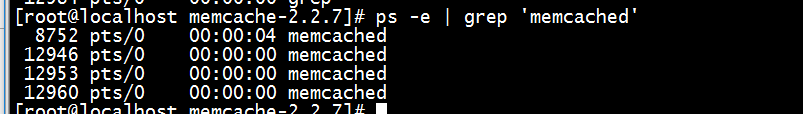
PHP的memcache扩展, 实现了分布式算法. 典型的: 一致性哈希算法. (同时也支持求余算法)

语法实现

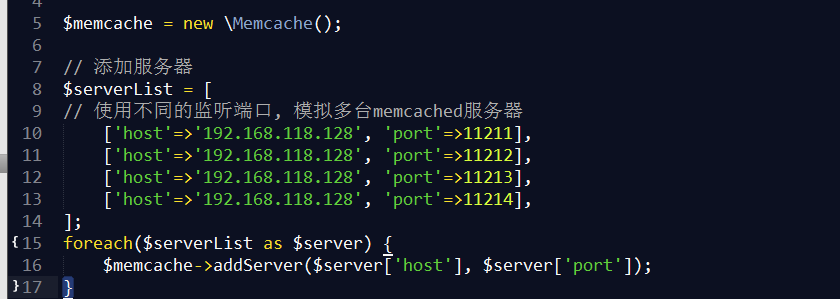
需要设置, 一共有哪些memcached服务器即可. 后续的操作不受任何影响.

准备好多台memcached服务器

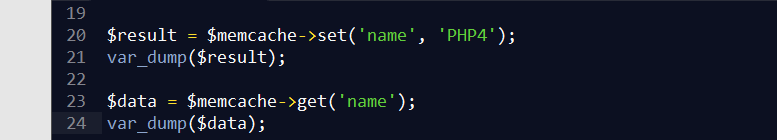




使用 addServer()即可添加:



后续的操作, 设置, 获取, 删除, 没有任何改变(分布式算法被封装到了memcache扩展内部)



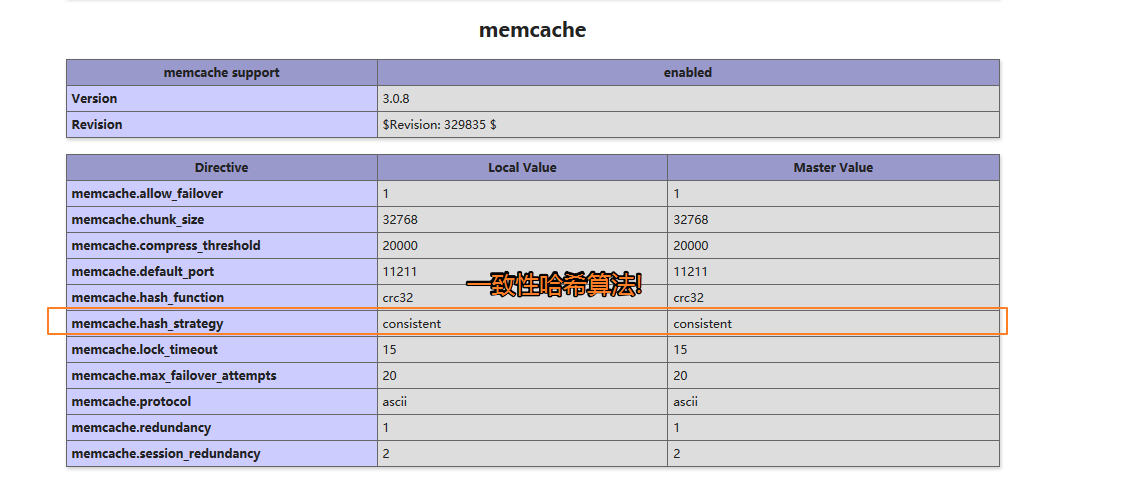
## 分布式算法

分布式, 将数据分散到不同的存储介质上, 而不是在不同的介质上, 存储多分.

能够决定, key对应的服务器节点.

目前的最典型的分布式算法(不仅仅是memcached)算法就是: 一致性哈希算法.

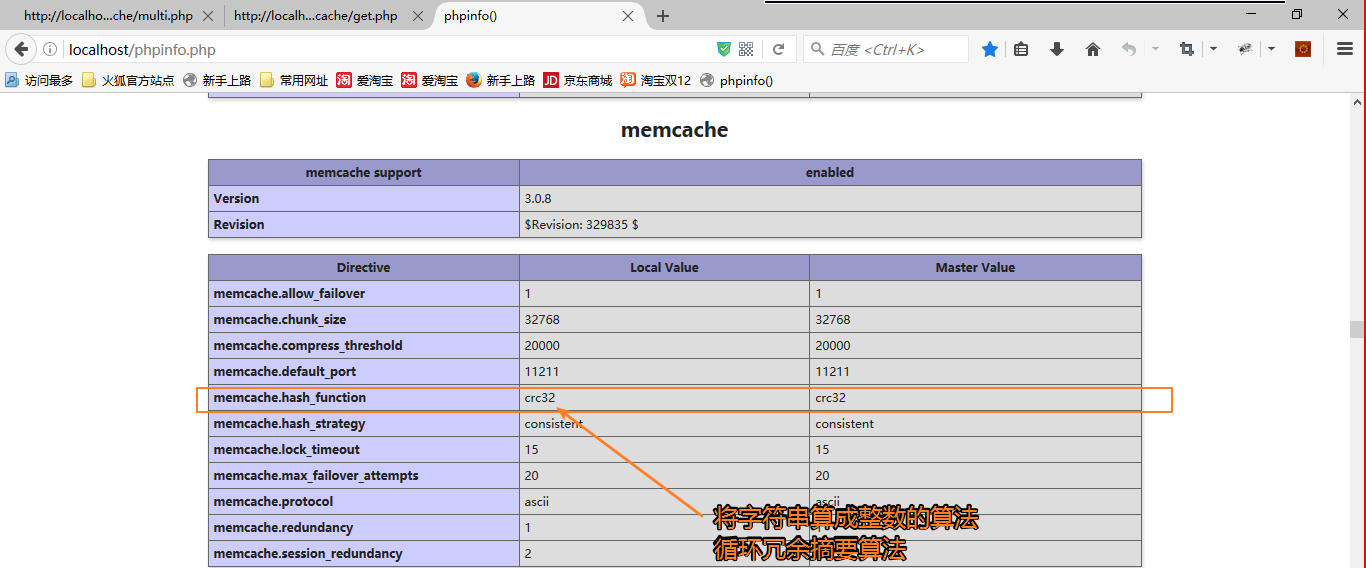
通过memcached的配置:



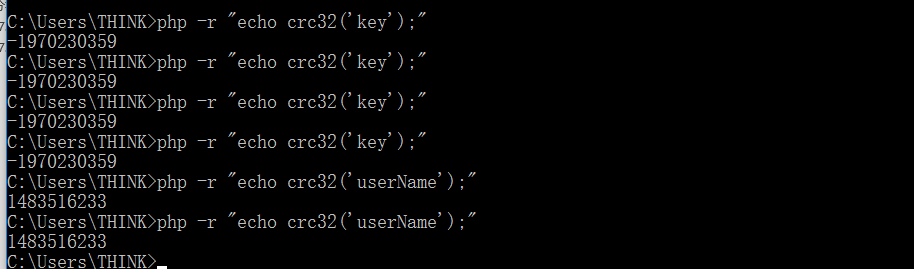
除了典型的一致性hash算法外, 还有一种更简单(不稳定, 容错性差)的算法, 就是求余算法.

### 求余

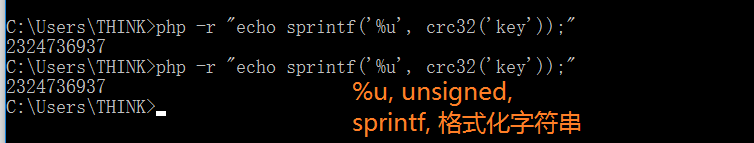
利用key, 算个整数, 利用整数对服务器数量求余, 使用余数对应哪台服务器.



该算法crc32, 给一个字符串, 计算出来一个32bit的整数.



负数可以转成无符号的正数:



相同的字符串, 计算的整数是一致的.

设置时, 利用key计算的整数, 与 获取时, 利用key计算的整数, 是一样的.

再利用整数对服务器的数量求余即可:

整数%4 = [0123]

每个余数对应一台服务器, 即可!

优势: 简单, 快速.

劣势: 服务器不能增减. 一旦增减, 导致绝大多数的缓存项失效. 没有办法再次找到.

### 一致性哈希算法

(额外的: 哈希算法, 利用一个key, 计算出来一个对应的值, 都称之为哈希算法)

目的: 在服务器增减时, 尽可能减少缓存项的失效.

实现逻辑如下:

1, 先利用32bit的整数, 计算出来全部的节点, 形成一个圈

由于crc32算法, 只能返回32bit的整数.

2, 计算每台服务器主机和端口的crc32摘要值, 对应到相应的节点上

crc32(‘host’.’port’)

server I = 4

server II= 1024

serverIII= 2048

serverIV = 65536

3, 计算缓存项key对应的服务器时

计算出来key的crc32

crc32(key) = 1000

从1000开始, 顺时针找,最近的服务器节点, 存储在该服务器上.



4, 增减服务器

当某台服务器增加或减少时, 分析带来的影响

仅仅会影响, 从增减的服务器节点开始逆时针到上一个最近的服务器节点间的数据.



# 概念

## 懒惰模式

当缓存项的有效时间到期时(缓存过期), 缓存项不会被立即删除.而是等到下次获取该缓存项时, 再删除.

过期的缓存项, 在get才会被删除.

这种设计, 就是懒惰模式.

目的: 节省了监视所有的缓存项是否过期的开销. 提缓存的操作速度!

可以想到, 会出现大量的过期的缓存项在内存中!

## LRU, Least Recently Use 最近最少使用

该算法的目的, 是在缓存服务器空间不足时, 增加新的缓存项, memcached会删除已经存在的某些缓存项.

删除缓存项的筛选算法, 就是 LRU算法.

逻辑是:

维护一个LRU队列, set和get的缓存项都被重新加入到队列:



当缓存项空间满时, 从最下面的开始删除, 直到空间够用

该机制可以保证, 最近使用的缓存项, 被保留, 最近好久没有使用的缓存项,优先被删除.

# 应用

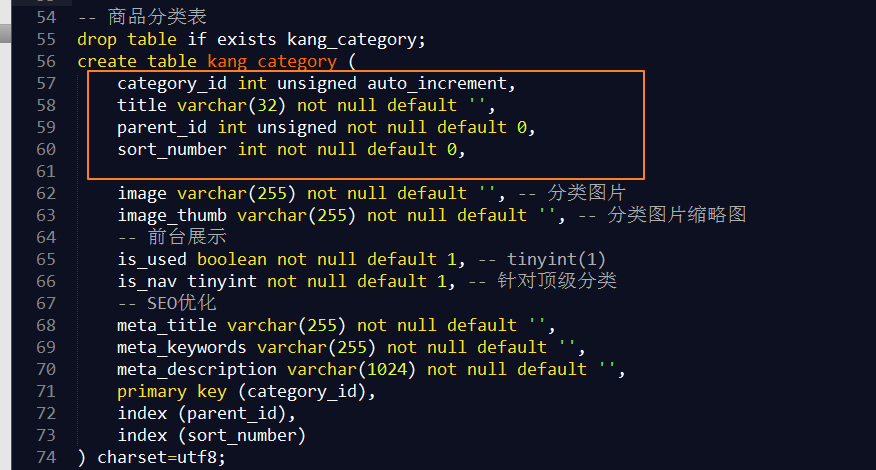
无限树状分类的缓存演示

数据特征: 1, 更新频率低. 2, 获取复杂.

以上两个特征, 满足一项 都应该放在缓存中. 同时满足, 肯定放!

项目中, 增加分类数据, 演示. 缓存的处理:

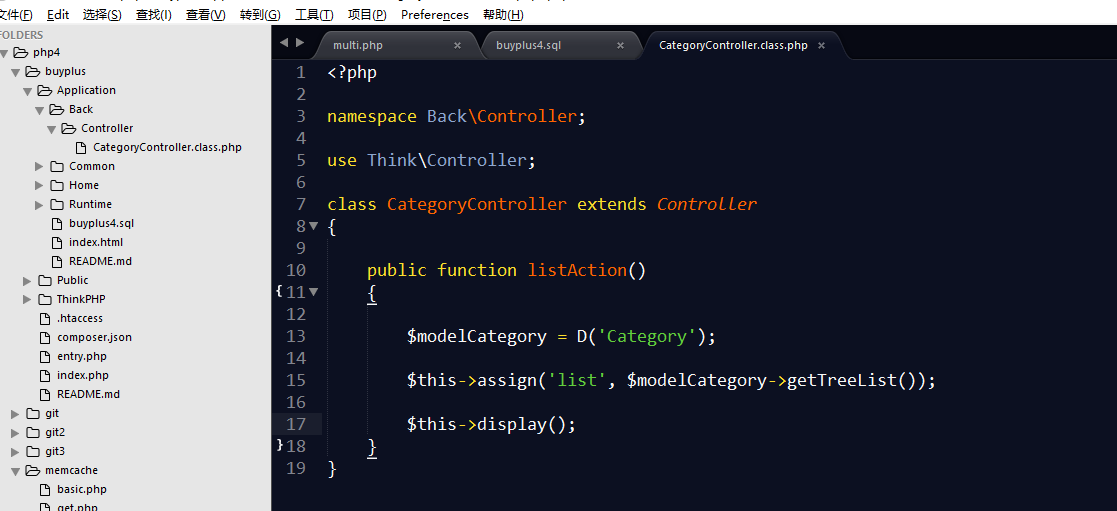
## 分类表



## 后台的分类列表

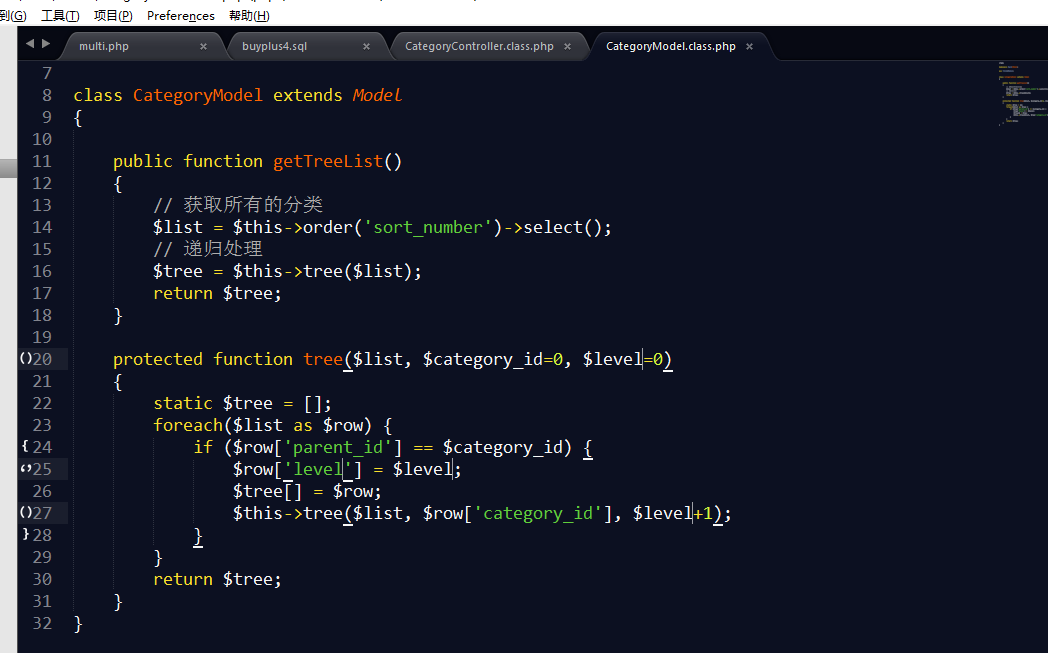
### 控制器-动作, back/Category/list

application/back/controller/CategoryController.class.php



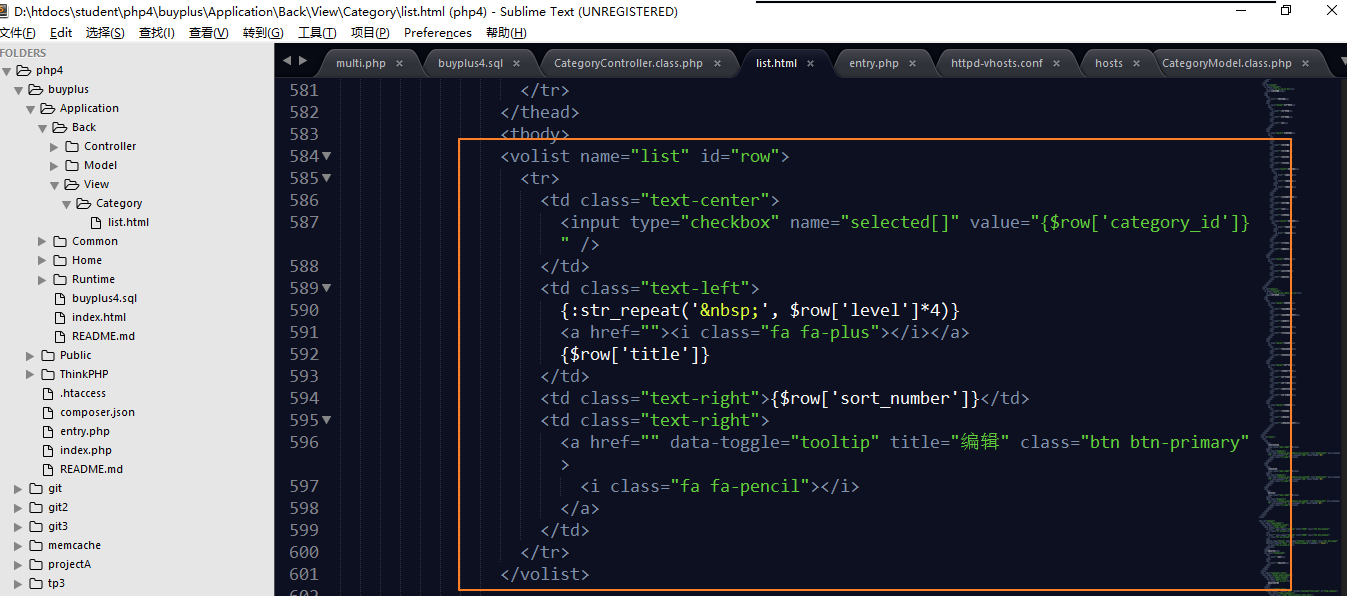
### 模型

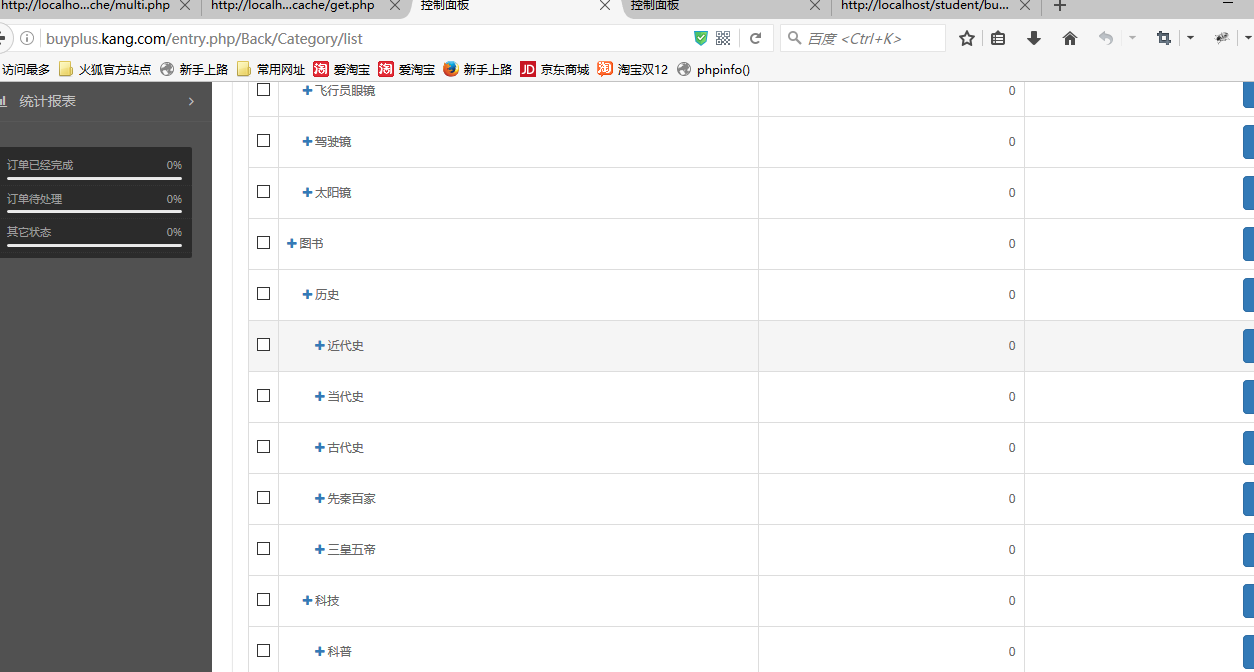
application/back/Model/CategoryModel.class.php



### 视图-模板

view/Cateogry/list.html





## 缓存系统

逻辑

查询时: 先去判断缓存是否存在, 如果有直接使用, 没有查询, 生成缓存.

编辑时: 缓存要被删除.

### 查询

定义在模型中

ThinkPHP中所有的缓存操作都由S()完成

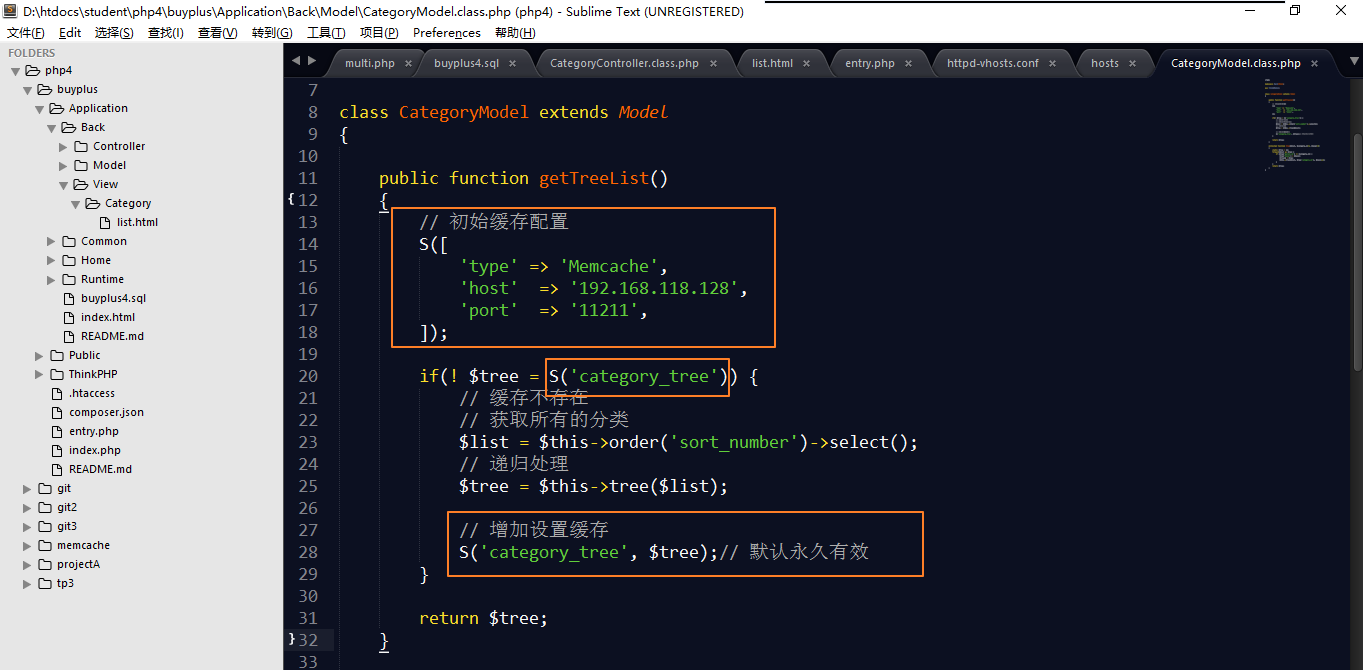
S(array), 配置

S(key), 获取

S(key, value), 设置

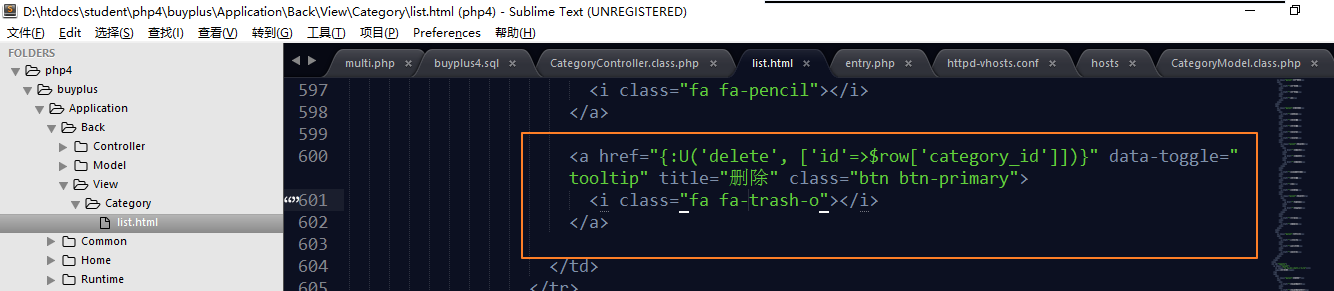
S(key, null), 删除

示例代码如下:



### 更新(删除)缓存

列表删除链接



删除动作, deleteAciton

