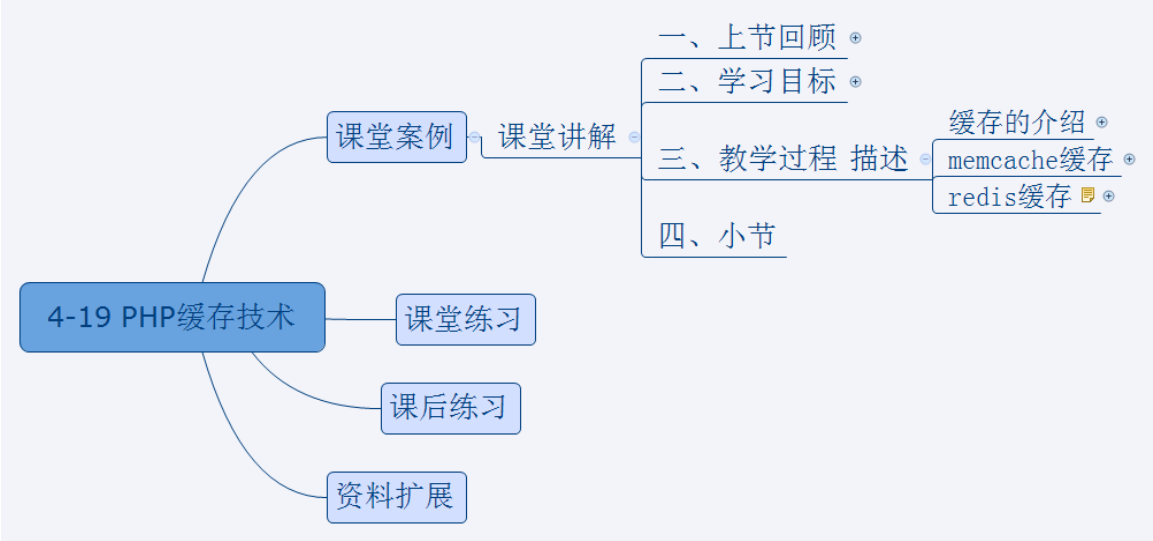
# 4-19 PHP缓存技术




# 1. 课堂案例

## 课堂讲解

### 一、上节回顾

**linux 下LAMP/LNMP环境的配置**

### 二、学习目标

了解缓存的原理

学会缓存的选择

了解memcache与redis区别

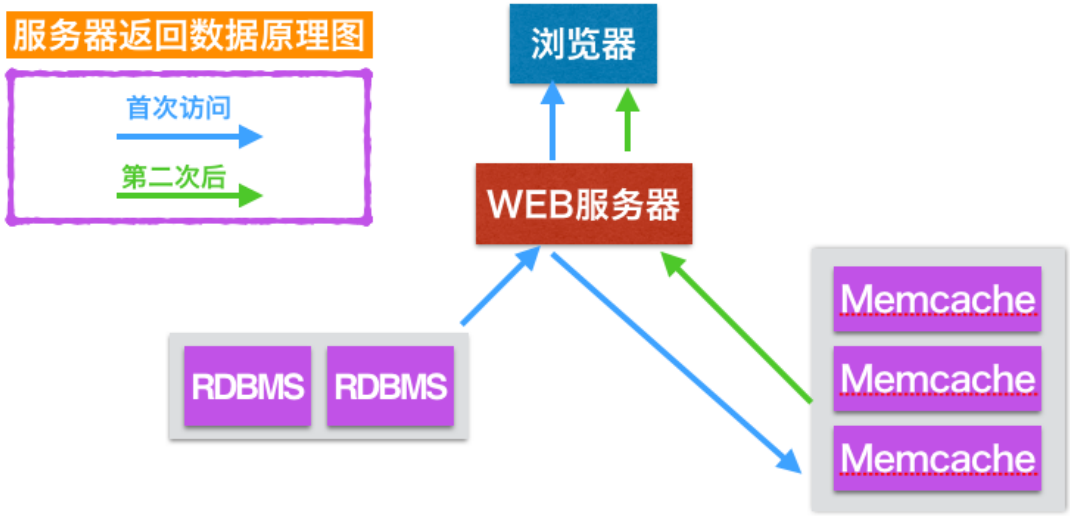
### 三、教学过程 描述

缓存的介绍

#### 为什么要使用缓存

缓存的使用在大访问量的情况下能够极大的减少对数据库操作的次数，明显降低系统负荷提高系统性能

## 缓存的原理




## 选择合适的缓存系统

常见缓存分类

一、PHP编译缓存：代表（xcache、eaccelerator、apc）

二、PHP内存缓存：代表(Memcache、Redis)

三、文件缓存：普通磁盘缓存

# 缓存的选择

使用频率　　数据块大小　　作用域

**通常各种类型缓存是交叉使用的，一个系统里即有内存缓存，又有文件缓存，甚至还有编译缓存**

## 内存缓存对比

redis和memecache的不同在于

1、存储方式：

memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小

redis有部份存在硬盘上，这样能保证数据的持久性，支持数据的持久化（笔者注：有快照和AOF日志两种持久化方式，在实际应用的时候，要特别注意配置文件快照参数，要不就很有可能服务器频繁满载做dump）。

2、数据支持类型：

redis在数据支持上要比memecache多的多。

3、使用底层模型不同：

新版本的redis直接自己构建了VM　　机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

4、运行环境不同：

redis目前官方只支持Linux上去行，从而省去了对于其它系统的支持，这样的话可以更好的把精力用于本系统环境上的优化，虽然后来微软有一个小组为其写了补丁。但是没有放到主干上

## memcache缓存

# Memcache解析

## 基础介绍

Memcache是一个高性能的分布式的内存对象缓存系统，通过在内存里维护一个统一的巨大的hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。简单的说就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。

## 理解Memcache

在 Memcached中可以保存的item数据量是没有限制的，只要内存足够 。
Memcached单进程在32位系统中最大使用内存为2G，若在64位系统则没有限制,这是由于32位系统限制单进程最多可使用2G内存,要使用更多内存，可以分多个端口开启多个Memcached进程 ,

最大30天的数据过期时间,设置为永久的也会在这个时间过期，常量REALTIME_MAXDELTA 60*60*24*30控制

最大键长为250字节，大于该长度无法存储，常量KEY_MAX_LENGTH 250控制

单个item最大数据是1MB，超过1MB数据不予存储，常量POWER_BLOCK 1048576进行控制，
□□它是默认的slab大小

最大同时连接数是200，通过 conn_init()中的freetotal进行控制，

最大软连接数是1024，通过 settings.maxconns=1024 进行控制

跟空间占用相关的参数：
settings.factor=1.25,
settings.chunk_size=48,
影响slab的数据占用和步进方式
memcached是一种无阻塞的socket通信方式服务，基于libevent库，由于无阻塞通信，对内存读写速度非常之快。

memcached分服务器端和客户端，可以配置多个服务器端和客户端，应用于分布式的服务非常广泛。

memcached作为小规模的数据分布式平台是十分有效果的。

memcached是键值一一对应，key默认最大不能超过128个字
节，value默认大小是1M，也就是一个slabs，如果要存2M的值（连续的），不能用两个slabs，因为两个slabs不是连续的，无法在内存中
存储，故需要修改slabs的大小，多个key和value进行存储时，即使这个slabs没有利用完，那么也不会存放别的数据。
memcached已经可以支持C/C++、Perl、PHP、Python、Ruby、Java、C#、Postgres、Chicken Scheme、Lua、MySQL和Protocol等语言客户端。

## 使用场景

应用场景一：缓解数据库压力，提高交互速度。

应用场景二：秒杀功能。

应用场景三：中继 MySQL 主从延迟数据

不适用memcached的业务场景

缓存对象的大小大于1MB
Memcached本身就不是为了处理庞大的多媒体（large media）和巨大的二进制块（streaming huge blobs）而设计的。

key的长度大于250字符（所以我们把一些key先md5再存储）。

应用运行在不安全的环境中Memcached为提供任何安全策略，仅仅通过telnet就可以访问到memcached。如果应用运行在共享的系统上，需要着重考虑安全问题。
业务本身需要的是持久化数据。

Memcache的安全
只说一下思路:把memcached的端口给禁止掉(这时只能本ip访问),让其他ip的使用者只能通过对外开放的80端口访问PHP脚本文件,再通过PHP的脚本文件去访问memcache;

iptables -a input -p 协议 -s 可以访问ip -dport 端口 -j ACCEPT

## Memcahce安装

### 安装服务端

**window 版本安装：**

**32位系统 1.4.4版本：**

**http://static.runoob.com/download/memcached-win32-1.4.4-14.zip**

**64位系统 1.4.4版本：**

**http://static.runoob.com/download/memcached-win64-1.4.4-14.zip**

**解压后，命令行进行目执行如下命令**

**c:\memcached\memcached.exe -d install**

**c:\memcached\memcached.exe -d start**

**c:\memcached\memcached.exe -d stop**

**-p 监听的端口**

**-l 连接的IP地址, 默认是本机**

**-d start 启动memcached服务**

**-d restart 重起memcached服务**

**-d stop|shutdown 关闭正在运行的memcached服务**

**-d install 安装memcached服务**

**-d uninstall 卸载memcached服务**

**-u 以的身份运行 (仅在以root运行的时候有效)**

**-m 最大内存使用，单位MB。默认64MB**

**-M 内存耗尽时返回错误，而不是删除项**

**-c 最大同时连接数，默认是1024**

**-f 块大小增长因子，默认是1.25**

**-n 最小分配空间，key+value+flags默认是48**

**-h 显示帮助**

**linux 版本安装**

**yum install memcached**

源代码安装

从其官方网站（http://memcached.org）下载memcached最新版本

wget http://memcached.org/latest 下载最新版

tar -zxvf memcached-1.x.x.tar.gz 解压源码

cd memcached-1.x.x 进入目录

./configure --prefix=/usr/local/memcached 配置

make && make test 编译

sudo make install 安装

Memcached 运行

Memcached命令的运行：

$ /usr/local/memcached/bin/memcached -h 命令帮助

注意：如果使用自动安装 memcached 命令位于 /usr/local/bin/memcached。

启动选项：

-d是启动一个守护进程；

-m是分配给Memcache使用的内存数量，单位是MB；

-u是运行Memcache的用户；

-l是监听的服务器IP地址，可以有多个地址；

-p是设置Memcache监听的端口，，最好是1024以上的端口；

-c是最大运行的并发连接数，默认是1024；

-P是设置保存Memcache的pid文件。

/usr/local/memcached/bin/memcached -p 11211 -m 64m -vv

守护进程

/usr/local/memcached/bin/memcached -p 11211 -m 64m -d

或者

/usr/local/memcached/bin/memcached -d -m 64M -u root -l 192.168.0.200 -p 11211 -c 256 -P /tmp/memcached.pid

## 安装PHP扩展

**window 下php扩展安装**

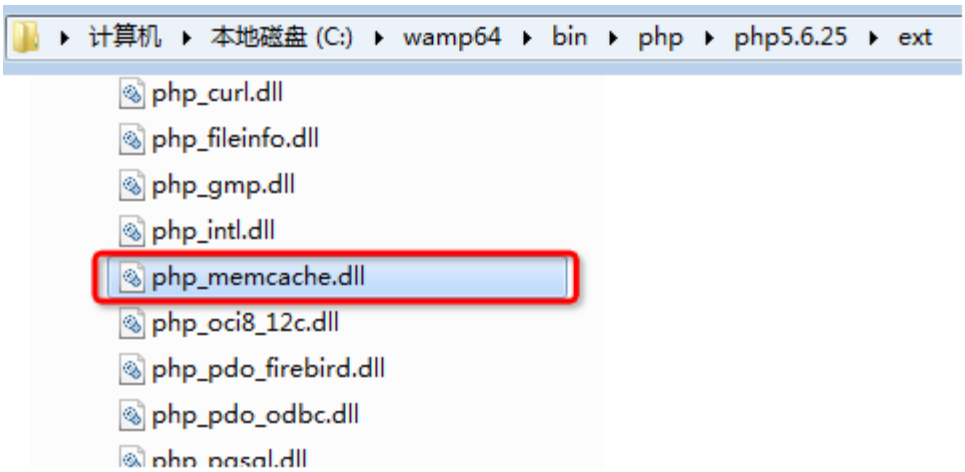
**下载:http://windows.php.net/downloads/pecl/releases/memcache/3.0.8/**

| | | | |
|---|---|---|---|
| 12/3/2016 | 2:00 AM | <dir> | logs |
| 10/22/2013 | 2:22 AM | 177451 | php_memcache-3.0.8-5.3-nts-vc9-x86.zip |
| 10/22/2013 | 2:26 AM | 179294 | php_memcache-3.0.8-5.3-ts-vc9-x86.zip |
| 10/22/2013 | 2:15 AM | 180586 | php_memcache-3.0.8-5.4-nts-vc9-x86.zip |
| 10/22/2013 | 2:19 AM | 182281 | php_memcache-3.0.8-5.4-ts-vc9-x86.zip |
| 10/22/2013 | 2:01 AM | 186792 | php_memcache-3.0.8-5.5-nts-vc11-x64.zip |
| 10/22/2013 | 2:08 AM | 181116 | php_memcache-3.0.8-5.5-nts-vc11-x86.zip |
| 10/22/2013 | 2:04 AM | 188269 | php_memcache-3.0.8-5.5-ts-vc11-x64.zip |
| 10/22/2013 | 2:12 AM | 183246 | php_memcache-3.0.8-5.5-ts-vc11-x86.zip |
| 4/10/2014 | 9:05 PM | 185627 | php_memcache-3.0.8-5.6-nts-vc11-x64.zip |
| 4/10/2014 | 8:57 PM | 181438 | php_memcache-3.0.8-5.6-nts-vc11-x86.zip |
| 4/10/2014 | 9:10 PM | 187946 | php_memcache-3.0.8-5.6-ts-vc11-x64.zip |
| 4/10/2014 | 9:01 PM | 183053 | php_memcache-3.0.8-5.6-ts-vc11-x86.zip |

**1. 解压下载的扩展文件**

| | | | |
|---|---|---|---|
| .. | | | 文件夹 |
| CREDITS | 32 | 32 | 文件 |
| example.php | 509 | 262 | PHP 文件 |
| LICENSE | 3,208 | 1,441 | 文件 |
| memcache.php | 29,110 | 8,105 | PHP 文件 |
| php_memcache.dll | 90,112 | 42,310 | 应用程序扩展 |
| php_memcache.pdb | 494,592 | 132,009 | PDB 文件 |
| README | 5,381 | 2,103 | 文件 |

**2. 将php_memcache.dll 文件拷贝到 php安装目录下面的 ext/ 文件夹下如:c:\wamp64\bin\php\php5.6.25\ext**




**3. 打开php.ini 添加如下代码**

**extension=php_memcache.dll**

**4. 重启apaceh服务 ,phpinfo 查看效果**

**如下表示已安装成功**

**memcache**

| memcache support | enabled |
|---|---|
| Version | 3.0.8 |
| Revision | $Revision: 329835 $ |

**linux 下安装**

**让php能使用memcached服务的扩展有两种：memcache 和 memcached**

**我们以 memcached为例**

**1. 先安装libmemcached扩展**

**yum -y install libmemcached**

**2. 安装php-pecl-memcache扩展**

**yum -y install php-pecl-memcache**

**3. 重启php**

**4.可通过phpinfo()查看是否安装了memcache扩展**

**5.安装成功后有可能在服务器能够通过telnet连接使用memcached服务，但是在php中通过new Memcache，加connect后返回的错误是连接被拒绝，这个原因是因为selinux安全机制的不允许memcached访问11211端口，所以必须对sulinux进行设置**

**临时生效的方法：setenforce Permissive**
**永久生效的方法：修改/etc/selinux/config文件，SELINUX=enforcing 改为SELINUX=disabled，从而关闭selinux**

## Memcahce类的操作和封装

### add

Memcache::add — 增加一个条目到缓存服务器
Memcache::add ( string $key , mixed $var [, int $flag [, int $expire ]] )

参数
key
将要分配给变量的key。

var
将要被存储的变量。字符串和整型被以原文存储，其他类型序列化后存储。

flag
使用MEMCACHE_COMPRESSED标记对数据进行压缩(使用zlib)。

expire
当前写入缓存的数据的失效时间。如果此值设置为0表明此数据永不过期。你可以设置一个UNIX时间戳或
以秒为单位的整数（从当前算起的时间差）来说明此数据的过期时间，但是在后一种设置方式中，不能超过 2592000秒（30天）。

```
<?php

$memcache_obj = memcache_connect("localhost", 11211);

/* 面向过程编程 API */
memcache_add($memcache_obj, 'var_key', 'test variable', false, 30);

/* 面向对象编程 API */
$memcache_obj->add('var_key', 'test variable', false, 30);

?>
```

## addServer

向连接池中添加一个memcache服务器

语法

bool Memcache::addServer ( string $host [, int $port = 11211 [, bool $persistent [, int $weight [, int $timeout [, int $retry_interval [, bool $status [, callback $failure_callback [, int $timeoutms ]]]]]]] )

参数

host

要连接的memcached服务端监听的主机位置。这个参数通常指定其他类型的传输比如Unix域套接字使用 unix:///path/to/memcached.sock，这种情况下参数port 必须设置为0。

port

要连接的memcached服务端监听的端口。当使用UNIX域套接字连接时设置为0。

persistent

控制是否使用持久化连接。默认TRUE。

weight

为此服务器创建的桶的数量，用来控制此服务器被选中的权重，单个服务器被选中的概率是相对于所有服务器weight总和而言的。

timeout

连接持续(超时)时间(单位秒)，默认值1秒，修改此值之前请三思，过长的连接持续时间可能会导致失去所有的缓存优势。

retry_interval
服务器连接失败时重试的间隔时间，默认值15秒。如果此参数设置为-1表示不重试。此参数和persistent参数在扩展以 dl()函数动态加载的时候无效。

每个失败的连接结构有自己的超时时间，并且在它失效之前选择后端服务请求时该结构会被跳过。一旦一个连接失效， 它将会被成功重新连接或被标记为失败连接以在下一个retry_interval秒重连。典型的影响是每个web服务子进程在服务于一个页面时将会每retry_interval秒 尝试重新连接一次。

status
控制此服务器是否可以被标记为在线状态。设置此参数值为FALSE并且retry_interval参数 设置为-1时允许将失败的服务器保留在一个池中以免影响key的分配算法。对于这个服务器的请求会进行故障转移或者立即失败，
这受限于memcache.allow_failover参数的设置。该参数默认TRUE，表明允许进行故障转移。

failure_callback
允许用户指定一个运行时发生错误后的回调函数。回调函数会在故障转移之前运行。回调函数会接受到两个参数，分别是失败主机的 主机名和端口号。

timeoutms

```
<?php

/* OO API */

$memcache = new Memcache;
$memcache->addServer('memcache_host', 11211);
$memcache->addServer('memcache_host2', 11211);

/* procedural API */

$memcache_obj = memcache_connect('memcache_host', 11211);
memcache_add_server($memcache_obj, 'memcache_host2', 11211);
```

```
?>
```

## close

关闭memcache连接

语法:

```
bool Memcache::close ( void )
```

```
<?php

/* 面向过程 API */
$memcache_obj = memcache_connect('memcache_host', 11211);
/*
do something here ..
*/
memcache_close($memcache_obj);

/* 面向对象 API */
$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);
/*
do something here ..
*/
$memcache_obj->close();

?>
```

## connect

打开一个memcached服务端连接

语法:

```
bool Memcache::connect ( string $host [, int $port [, int $timeout ]] )
```

参数

host

memcached服务端监听主机地址。这个参数也可以指定为其他传输方式比如unix:///path/to/memcached.sock 来使用Unix域socket，在这种方式下，port参数必须设置为0。

port
memcached服务端监听端口。当使用Unix域socket的时候要设置此参数为0。

timeout
连接持续(超时)时间，单位秒。默认值1秒，修改此值之前请三思，过长的连接持续时间可能会导致失去所有的缓存优势。

```
<?php

/* procedural API */

$memcache_obj = memcache_connect('memcache_host', 11211);

/* OO API */

$memcache = new Memcache;
$memcache->connect('memcache_host', 11211);

?>
```

## decrement

减小元素的值

语法
int Memcache::decrement ( string $key [, int $value = 1 ] )

参数

key
要减小值的元素的key。

value
value参数指要将指定元素的值减小多少。

```
<?php

/* procedural API */
$memcache_obj = memcache_connect('memcache_host', 11211);
/* 将test_item对应的值减小2 */
$new_value = memcache_decrement($memcache_obj, 'test_item', 2);

/* OO API */
$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);
/* decrement item by 3 */
$new_value = $memcache_obj->decrement('test_item', 3);
?>
```

## delete

从服务端删除一个元素

语法

```
bool Memcache::delete ( string $key [, int $timeout = 0 ] )
```

参数

key

要删除的元素的key。

timeout

删除该元素的执行时间。如果值为0,则该元素立即删除，如果值为30,元素会在30秒内被删除

```
<?php

/* procedural API */
$memcache_obj = memcache_connect('memcache_host', 11211);

/* 10秒后key_to_delete对应的值会被从服务端删除 */
```

```
memcache_delete($memcache_obj, 'key_to_delete', 10);

/* OO API */
$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);

$memcache_obj->delete('key_to_delete', 10);

?>
```

## flush

清洗（删除）已经存储的所有的元素

语法：

```
bool Memcache::flush ( void )
```

```
<?php

/* procedural API */
$memcache_obj = memcache_connect('memcache_host', 11211);

memcache_flush($memcache_obj);

/* OO API */

$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);

$memcache_obj->flush();

?>
```

## get

从服务端检回一个元素

语法

string Memcache::get ( string $key [, int &$flags ] )
array Memcache::get ( array $keys [, array &$flags ] )

参数

key
要获取值的key或key数组。

flags
如果给定这个参数（以引用方式传递），该参数会被写入一些key对应的信息。这些标记和Memcache::set()方法中的同名参数
意义相同。用int值的低位保留了pecl/memcache的内部用法（比如：用来说明压缩和序列化状态）。（译注：最后一位表明是否序列化，倒数第二位表明是否经过压缩，
比如：如果此值为1表示经过序列化，但未经过压缩，2表明压缩而未序列化，3表明压缩并且序列化，0表明未经过压缩和序列化，具体算法可查找linux文件权限算法相关资料）

```
<?php

/* procedural API */
$memcache_obj = memcache_connect('memcache_host', 11211);
$var = memcache_get($memcache_obj, 'some_key');

/* OO API */
$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);
$var = $memcache_obj->get('some_key');

/*
你同样可以使用数组key作为参数，如果某个元素没有在服务端发现，结果数组中将不会包含这些key。
*/

/* procedural API */
$memcache_obj = memcache_connect('memcache_host', 11211);
$var = memcache_get($memcache_obj, Array('some_key', 'another_key'));

/* OO API */
```

```
$memcache_obj = new Memcache;
$memcache_obj->connect('memcache_host', 11211);
$var = $memcache_obj->get(Array('some_key', 'second_key'));

?>
```

## set

设置数据，从服务器上

语法

```
bool Memcache::set ( string $key , mixed $var [, int $flag [, int $expire ]] )
```

参数

key

要设置值的key。

var

要存储的值，字符串和数值直接存储，其他类型序列化后存储。

flag

使用MEMCACHE_COMPRESSED指定对值进行压缩(使用zlib)。

expire

当前写入缓存的数据的失效时间。如果此值设置为0表明此数据永不过期。你可以设置一个UNIX时间戳或
以秒为单位的整数（从当前算起的时间差）来说明此数据的过期时间，但是在后一种设置方式中，不能超过 2592000秒（30天）。

```
<?php
/* procedural API */

/* connect to memcached server */
$memcache_obj = memcache_connect('memcache_host', 11211);
```

```
/*
设置'var_key'对应存储的值
flag参数使用0,值没有经过压缩
失效时间为30秒
*/
memcache_set($memcache_obj, 'var_key', 'some variable', 0, 30);

echo memcache_get($memcache_obj, 'var_key');

?>
```

```
<?php
/* OO API */

$memcache_obj = new Memcache;

/* connect to memcached server */
$memcache_obj->connect('memcache_host', 11211);

/*
设置'var_key'对应值，使用即时压缩
失效时间为50秒
*/
$memcache_obj->set('var_key', 'some really big variable', MEMCACHE_COMPRESSED, 50);

echo $memcache_obj->get('var_key');

?>
```

## getStats

获取服务器统计信息

语法

array Memcache::getStats ([ string $type [, int $slabid [, int $limit = 100 ]]] )

参数

type

期望抓取的统计信息类型，可以使用的值有{reset，malloc，maps，cachedump，slabs，items，sizes}。通过memcached协议指定这些附加参数是为了方便memcache开发者(检查其中的变动)。

slabid

用于与参数type联合从指定slab分块拷贝数据，cachedump命令会完全占用服务器通常用于比较严格的调试。

limit

用于和参数type联合来设置cachedump时从服务端获取的实体条数。

## getVersion

**返回服务器版本信息**

**string Memcache::getVersion ( void )**

```
<?php

/* OO API */
$memcache = new Memcache;
$memcache->connect('memcache_host', 11211);
echo $memcache->getVersion();

/* procedural API */
$memcache = memcache_connect('memcache_host', 11211);
echo memcache_get_version($memcache);

?>
```

### 封闭自己的memcache类

写一个 memcached 常见操作的class 类

## Memcache的整合

### 项目中使用Memcahce

TP中使用memecahce 配置

```
'DATA_CACHE_TYPE' => 'Memcache',
'MEMCACHE_HOST' => '127.0.0.1',
'MEMCACHE_PORT'   =>    '11211',
'DATA_CACHE_TIME' => '3600',
```

接着使用

S() 方法

### 使用Memcache

demo

## Memcahce安全

**内网访问**

**两台服务器之间的访问设为内网访问，一般是Web服务器跟Memcached服务器之间。普遍的服务器都是有两块网卡，一块指向互联网，一块指向内网，那么就让Web服务器通过内网的网卡来访问Memcached服务器，Memcached的服务器启动的时候就监听内网的IP地址和端口，内网间的访问能够有效阻止其他非法的访问。**

**# memcached -d -m 1024 -u root -l 192.168.0.200 -p 11211 -c 1024 -P /tmp/memcached.pid**

**Memcached服务器端设置监听通过内网的192.168.0.200的ip的11211端口，占用1024MB内存，并且允许最大1024个并发连接**

**设置防火墙**

**防火墙是简单有效的方式，如果两台服务器都是挂在网的并且需要通过外网IP来访问Memcached的话，那么可以考虑使用防火墙或者代理程序来过滤非法访问。**

**一般我们在Linux下可以使用iptables或者FreeBSD下的ipfw来指定一些规则防止一些非法的访问，比如我们可以设置只允许我们的Web服务器来访问我们Memcached服务器，同时阻止其他的访问。**

**# iptables -F**

**# iptables -P INPUT DROP**

**# iptables -A INPUT -p tcp -s 192.168.0.2 –dport 11211 -j ACCEPT**

**# iptables -A INPUT -p udp -s 192.168.0.2 –dport 11211 -j ACCEPT**

**上面的iptables规则就是只允许192.168.0.2这台Web服务器对Memcached服务器的访问，能够有效的阻止一些非法访问，相应的也可以增加一些其他的规则来加强安全性，这个可以根据自己的需要来做**

## redis缓存

http://www.redis.cn/documentation.html

### 1.课程简介

#### 1-1 什么是redis

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）

#### 1-2 redis的应用场景

1. 需要缓存的同时，需要持久化

2. 存储的数据类型多样化，那么请选择redis

### 2. redis的安装

#### 2-1 redis服务器端安装

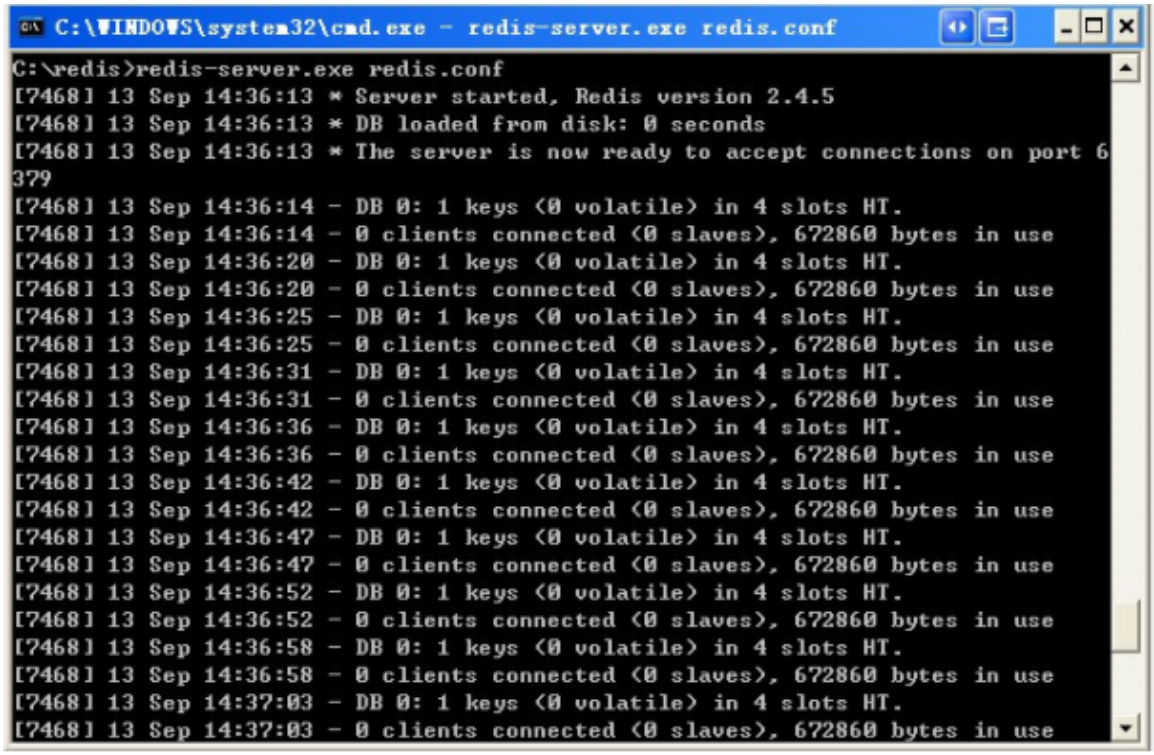
**Window 下安装**

**下载地址：https://github.com/dmajkic/redis/downloads**

**下载到的Redis支持32bit和64bit。根据自己实际情况选择，将64bit的内容cp到自定义盘符安装目录取名redis。如 C:\reids**

**打开一个cmd窗口 使用cd命令切换目录到 C:\redis 运行 redis-server.exe redis.conf 。**

**如果想方便的话，可以把redis的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个redis.conf可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：**

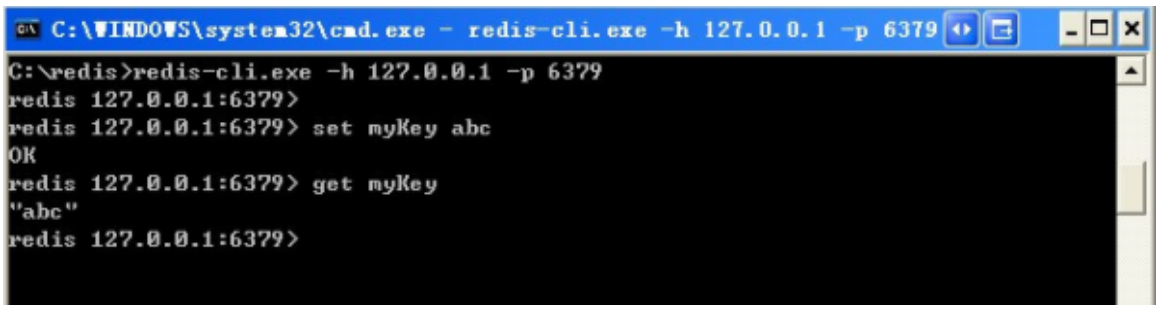



**这时候另启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。**

**切换到redis目录下运行 redis-cli.exe -h 127.0.0.1 -p 6379 。**

**设置键值对 set myKey abc**

**取出键值对 get myKey**




**Linux 下安装**

**下载地址：http://redis.io/download，下载最新文档版本**

**本教程使用的最新文档版本为 2.8.17，下载并安装：**

**$ wget http://download.redis.io/releases/redis-2.8.17.tar.gz**

**$ tar xzf redis-2.8.17.tar.gz**

**$ cd redis-2.8.17**

$ make

make完后 redis-2.8.17目录下会出现编译后的redis服务程序redis-server,还有用于测试的客户端程序redis-cli

下面启动redis服务.

$ ./redis-server

注意这种方式启动redis

使用的是默认配置。也可以通过启动参数告诉redis使用指定配置文件使用下面命令启动。

$ ./redis-server redis.conf

redis.conf是一个默认的配置文件。我们可以根据需要使用自己的配置文件。

启动redis服务进程后，就可以使用测试客户端程序redis-cli和redis服务交互了。比如：

$ ./redis-cli

redis> set foo bar

OK

redis> get foo

"bar"

启动 Redis

$redis-server

查看 redis 是否启动？

$redis-cli

编辑配置

你可以通过修改 redis.conf 文件或使用 CONFIG set 命令来修改配置。

参数说明

redis.conf 配置项说明如下：

1. Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程

   daemonize no

2.

当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定

pidfile /var/run/redis.pid

3.

指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

port 6379

4. 绑定的主机地址

bind 127.0.0.1

5.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能

timeout 300

6. 指定日志记录级别，Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose

loglevel verbose

7.

日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

logfile stdout

8. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

databases 16

9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

save <seconds> <changes>

Redis默认配置文件中提供了三个条件：

save 900 1

save 300 10

save 60 10000

分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10000个更改。

10.

指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

rdbcompression yes

11. 指定本地数据库文件名，默认值为dump.rdb

dbfilename dump.rdb

12. 指定本地数据库存放目录

dir ./

13.

设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

slaveof <masterip> <masterport>

14. 当master服务设置了密码保护时，slav服务连接master的密码

masterauth <master-password>

15. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

requirepass foobared

16. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

maxclients 128

17. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区

maxmemory <bytes>

18. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

appendonly no

19. 指定更新日志文件名，默认为appendonly.aof

appendfilename appendonly.aof

20. 指定更新日志条件，共有3个可选值：

no：表示等操作系统进行数据缓存同步到磁盘（快）

always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）

everysec：表示每秒同步一次（折衷，默认值）

appendfsync everysec

21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）

vm-enabled no

22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

vm-swap-file /tmp/redis.swap

23. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

vm-max-memory 0

24. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的
数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值

vm-page-size 32

25.
设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。

vm-pages 134217728

26.
设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

vm-max-threads 4

27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

glueoutputbuf yes

28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

hash-max-zipmap-entries 64

hash-max-zipmap-value 512

29. 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）

activerehashing yes

30.
指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

include /path/to/local.conf

# 3. redis数据类型介绍

## 3-1 redis的五种数据类型

Redis支持五种数据类型

string（字符串），

hash（哈希），

list（列表），

**set（集合）**

**zset(sorted set：有序集合)。**

## 3-2 string类型操作

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

实例

```
redis 127.0.0.1:6379> SET name "w3cschool.cn"
OK
redis 127.0.0.1:6379> GET name
"w3cschool.cn"
```

在以上实例中我们使用了 Redis 的 SET 和 GET 命令。键为 name，对应的值为w3cschool.cn。

注意：一个键最大能存储512MB。

## 3-3 list类型操作

List（列表）

Redis

列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。

实例

```
redis 127.0.0.1:6379> lpush w3cschool.cn redis
(integer) 1
redis 127.0.0.1:6379> lpush w3cschool.cn mongodb
(integer) 2
redis 127.0.0.1:6379> lpush w3cschool.cn rabitmq
(integer) 3
redis 127.0.0.1:6379> lrange w3cschool.cn 0 10
1) "rabitmq"
2) "mongodb"
3) "redis"
redis 127.0.0.1:6379>
```

列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储40多亿)。

## 3-4 set类型操作

Set（集合）

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

sadd 命令

添加一个string元素到,key对应的set集合中，成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误。

sadd key member

实例

```
redis 127.0.0.1:6379> sadd w3cschool.cn redis
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cn mongodb
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cn rabitmq
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cn rabitmq
(integer) 0
redis 127.0.0.1:6379> smembers w3cschool.cn

1) "rabitmq"
2) "mongodb"
3) "redis"
```

注意:以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

## 3-5 hash类型操作

Hash（哈希）

Redis hash 是一个键值对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

实例

```
redis 127.0.0.1:6379> HMSET user:1 username w3cschool.cn password w3cschool.cn points 200
OK
redis 127.0.0.1:6379> HGETALL user:1
```

1) "username"
2) "w3cschool.cn"
3) "password"
4) "w3cschool.cn"
5) "points"
6) "200"
redis 127.0.0.1:6379>

以上实例中 hash 数据类型存储了包含用户脚本信息的用户对象。 实例中我们使用了 Redis HMSET, HEGTALL 命令，user:1 为键值。

每个 hash 可以存储 232 - 1 键值对（40多亿）。

## 3-6 sort set类型操作

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令

添加元素到集合，元素在集合中存在则更新对应score

zadd key score member

实例

```
redis 127.0.0.1:6379> zadd w3cschool.cn 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cn 0 mongodb
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cn 0 rabitmq
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cn 0 rabitmq
(integer) 0
redis 127.0.0.1:6379> ZRANGEBYSCORE w3cschool.cn 0 1000

1) "redis"
2) "mongodb"
3) "rabitmq"
```

## 4. php的redis扩展安装

window下安装扩展

php_redis.dll下载地址：http://windows.php.net/downloads/pecl/snaps/redis/2.2.5/

将下载解压后的php_redis.dll放入php的ext目录下

然后修改php.ini，加入

```
; php-redis
extension=php_igbinary.dll
extension=php_redis.dll
```

linux 下安装 redis扩展

PHP安装redis扩展

以下操作需要在下载的 phpredis 目录中完成：

```
$ wget https://github.com/phpredis/phpredis/archive/2.2.4.tar.gz
$ cd phpredis-2.2.7                    # 进入 phpredis 目录
$ /usr/local/php/bin/phpize            # php安装后的路径
$ ./configure --with-php-config=/usr/local/php/bin/php-config
$ make && make install
```

## 5. php操作redis

### 5-1 redis 的链接操作

1，connect

描述：实例连接到一个Redis.

参数：host: string，port: int

返回值：BOOL 成功返回：TRUE;失败返回：FALSE

示例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$result = $redis->connect('127.0.0.1', 6379);
var_dump($result); //结果：bool(true)
?>
```

### 5-2 string 类型操作

**set**

描述：设置key和value的值

参数：Key Value

返回值：BOOL 成功返回：TRUE;失败返回：FALSE

示例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$result = $redis->set('test',"11111111111");
var_dump($result);   //结果：bool(true)
?>
```

**get**

描述：获取有关指定键的值

参数：key

返回值：string或BOOL 如果键不存在，则返回 FALSE。否则，返回指定键对应的value值。

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$result = $redis->get('test');
var_dump($result);   //结果：string(11) "11111111111"
?>
```

**delete**

描述：删除指定的键

参数：一个键，或不确定数目的参数，每一个关键的数组：key1 key2 key3 … keyN

返回值：删除的项数

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test',"1111111111111");
echo $redis->get('test');   //结果：1111111111111
```

```
$redis->delete('test');
var_dump($redis->get('test'));  //结果:bool(false)
?>
```

**setnx**

描述:如果在数据库中不存在该键，设置关键值参数

参数:key value

返回值:BOOL 成功返回:TRUE;失败返回:FALSE

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test',"1111111111111");
$redis->setnx('test',"22222222");
echo $redis->get('test');  //结果:1111111111111
$redis->delete('test');
$redis->setnx('test',"22222222");
echo $redis->get('test');  //结果:22222222
?>
```

**exists**

描述:验证指定的键是否存在

参数key

返回值:Bool 成功返回:TRUE;失败返回:FALSE

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test',"1111111111111");
var_dump($redis->exists('test'));  //结果:bool(true)
?>
```

**incr**

描述：数字递增存储键值键.

参数：key value：将被添加到键的值

返回值：INT the new value

实例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test',"123");
var_dump($redis->incr("test"));  //结果：int(124)
var_dump($redis->incr("test"));  //结果：int(125)
?>
```

**decr**

描述：数字递减存储键值。

参数：key value：将被添加到键的值

返回值：INT the new value

实例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test',"123");
var_dump($redis->decr("test"));  //结果：int(122)
var_dump($redis->decr("test"));  //结果：int(121)
?>
```

**getMultiple**

描述：取得所有指定键的值。如果一个或多个键不存在，该数组中该键的值为假

参数：其中包含键值的列表数组

返回值：返回包含所有键的值的数组

实例：

复制代码 代码如下：

```
<?php
```

```
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->set('test1',"1");
$redis->set('test2',"2");
$result = $redis->getMultiple(array('test1','test2'));
print_r($result);   //结果:Array ( [0] => 1 [1] => 2 )
?>
```

## 5-3 list类型操作

**lpush**

描述:由列表头部添加字符串值。如果不存在该键则创建该列表。如果该键存在，而且不是一个列表，返回FALSE。

参数:key,value

返回值:成功返回数组长度，失败false

实例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
var_dump($redis->lpush("test","111"));   //结果:int(1)
var_dump($redis->lpush("test","222"));   //结果:int(2)
?>
```

**rpush**

描述:由列表尾部添加字符串值。如果不存在该键则创建该列表。如果该键存在，而且不是一个列表，返回FALSE。

参数:key,value

返回值:成功返回数组长度，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
var_dump($redis->lpush("test","111"));   //结果:int(1)
var_dump($redis->lpush("test","222"));   //结果:int(2)
```

```
var_dump($redis->rpush("test","333"));  //结果:int(3)
var_dump($redis->rpush("test","444"));  //结果:int(4)
?>
```

**lpop**

描述:返回和移除列表的第一个元素

参数:key

返回值:成功返回第一个元素的值，失败返回false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush("test","111");
$redis->lpush("test","222");
$redis->rpush("test","333");
$redis->rpush("test","444");
var_dump($redis->lpop("test"));  //结果:string(3) "222"
?>
```

**lsize,llen**

描述:返回的列表的长度。如果列表不存在或为空，该命令返回0。如果该键不是列表，该命令返回FALSE。

参数:Key

返回值:成功返回数组长度，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush("test","111");
$redis->lpush("test","222");
$redis->rpush("test","333");
$redis->rpush("test","444");
var_dump($redis->lsize("test"));  //结果:int(4)
?>
```

**lget**

描述：返回指定键存储在列表中指定的元素。 0第一个元素，1第二个… -1最后一个元素，-2的倒数第二…错误的索引或键不指向列表则返回FALSE。

参数：key index

返回值：成功返回指定元素的值，失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush("test","111");
$redis->lpush("test","222");
$redis->rpush("test","333");
$redis->rpush("test","444");
var_dump($redis->lget("test",3));  //结果：string(3) "444"
?>
```

**lset**

描述：为列表指定的索引赋新的值,若不存在该索引返回false.

参数：key index value

返回值：成功返回true,失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush("test","111");
$redis->lpush("test","222");
var_dump($redis->lget("test",1));  //结果：string(3) "111"
var_dump($redis->lset("test",1,"333"));  //结果：bool(true)
var_dump($redis->lget("test",1));  //结果：string(3) "333"
?>
```

**lgetrange**

描述：

返回在该区域中的指定键列表中开始到结束存储的指定元素，lGetRange(key, start, end)。0第一个元素，1第二个元素… -1最后一个元素，-2的倒数第二…

参数：key start end

返回值：成功返回查找的值，失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush("test","111");
$redis->lpush("test","222");
print_r($redis->lgetrange("test",0,-1));  //结果：Array ( [0] => 222 [1] => 111 )
?>
```

**lremove**

描述：从列表中从头部开始移除count个匹配的值。如果count为零，所有匹配的元素都被删除。如果count是负数，内容从尾部开始删除。

参数：key count value

返回值：成功返回删除的个数，失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->lpush('test','a');
$redis->lpush('test','b');
$redis->lpush('test','c');
$redis->rpush('test','a');
print_r($redis->lgetrange('test', 0, -1)); //结果：Array ( [0] => c [1] => b [2] => a [3] => a )
var_dump($redis->lremove('test','a',2));   //结果：int(2)
print_r($redis->lgetrange('test', 0, -1)); //结果：Array ( [0] => c [1] => b )
?>
```

## 5-4 set类型操作

**sadd**

描述：为一个Key添加一个值。如果这个值已经在这个Key中，则返回FALSE。

参数:key value

返回值:成功返回true,失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
var_dump($redis->sadd('test','111'));   //结果:bool(true)
var_dump($redis->sadd('test','333'));   //结果:bool(true)
print_r($redis->sort('test')); //结果:Array ( [0] => 111 [1] => 333 )
?>
```

**sremove**

描述:删除Key中指定的value值

参数:key member

返回值:true or false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd('test','111');
$redis->sadd('test','333');
$redis->sremove('test','111');
print_r($redis->sort('test'));    //结果:Array ( [0] => 333 )
?>
```

**smove**

描述:将Key1中的value移动到Key2中

参数:srcKey dstKey member

返回值:true or false

范例

复制代码 代码如下:

```
<?php
```

```
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->delete('test1');
$redis->sadd('test','111');
$redis->sadd('test','333');
$redis->sadd('test1','222');
$redis->sadd('test1','444');
$redis->smove('test',"test1",'111');
print_r($redis->sort('test1'));    //结果:Array ( [0] => 111 [1] => 222 [2] => 444 )
?>
```

**scontains**

描述:检查集合中是否存在指定的值。

参数:key value

返回值:true or false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd('test','111');
$redis->sadd('test','112');
$redis->sadd('test','113');
var_dump($redis->scontains('test', '111')); //结果:bool(true)
?>
```

**ssize**

描述:返回集合中存储值的数量

参数:key

返回值:成功返回数组个数，失败0

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd('test','111');
```

```
$redis->sadd('test','112');
echo $redis->ssize('test');   //结果:2
?>
```

**spop**

描述:随机移除并返回key中的一个值

参数:key

返回值:成功返回删除的值，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
var_dump($redis->spop("test"));  //结果:string(3) "333"
?>
```

**sinter**

描述:返回一个所有指定键的交集。如果只指定一个键，那么这个命令生成这个集合的成员。如果不存在某个键，则返回FALSE。

参数:key1, key2, keyN

返回值:成功返回数组交集，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
$redis->sadd("test1","444");
var_dump($redis->sinter("test","test1"));  //结果:array(1) { [0]=> string(3) "111" }
?>
```

**sinterstore**

描述：执行sInter命令并把结果储存到新建的变量中。

参数：

Key: dstkey, the key to store the diff into.

Keys: key1, key2… keyN. key1..keyN are intersected as in sInter.

返回值：成功返回，交集的个数，失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
$redis->sadd("test1","444");
var_dump($redis->sinterstore('new',"test","test1"));  //结果：int(1)
var_dump($redis->smembers('new'));  //结果:array(1) { [0]=> string(3) "111" }
?>
```

**sunion**

描述：

返回一个所有指定键的并集

参数：

Keys: key1, key2, … , keyN

返回值：成功返回合并后的集，失败false

范例：

复制代码 代码如下：

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
$redis->sadd("test1","444");
```

print_r($redis->sunion("test","test1")); //结果:Array ( [0] => 111 [1] => 222 [2] => 333 [3] => 444 )
?>

**sunionstore**

描述:执行sunion命令并把结果储存到新建的变量中。

参数:

Key: dstkey, the key to store the diff into.

Keys: key1, key2… keyN. key1..keyN are intersected as in sInter.

返回值:成功返回，交集的个数，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
$redis->sadd("test1","444");
var_dump($redis->sinterstore('new',"test","test1")); //结果:int(4)
print_r($redis->smembers('new')); //结果:Array ( [0] => 111 [1] => 222 [2] => 333 [3] => 444 )
?>
```

**sdiff**

描述:返回第一个集合中存在并在其他所有集合中不存在的结果

参数:Keys: key1, key2, … , keyN: Any number of keys corresponding to sets in redis.

返回值:成功返回数组，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
```

```
$redis->sadd("test1","444");
print_r($redis->sdiff("test","test1"));  //结果:Array ( [0] => 222 [1] => 333 )
?>
```

**sdiffstore**

描述:执行sdiff命令并把结果储存到新建的变量中。

参数:

Key: dstkey, the key to store the diff into.

Keys: key1, key2, … , keyN: Any number of keys corresponding to sets in redis

返回值:成功返回数字，失败false

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
$redis->sadd("test","333");
$redis->sadd("test1","111");
$redis->sadd("test1","444");
var_dump($redis->sdiffstore('new',"test","test1"));  //结果:int(2)
print_r($redis->smembers('new'));  //结果:Array ( [0] => 222 [1] => 333 )
?>
```

**smembers, sgetmembers**

描述:

返回集合的内容

参数:Key: key

返回值:An array of elements, the contents of the set.

范例:

复制代码 代码如下:

```
<?php
$redis = new redis();
$redis->connect('127.0.0.1', 6379);
$redis->delete('test');
$redis->sadd("test","111");
$redis->sadd("test","222");
```

```
print_r($redis->smembers('test'));  //结果:Array ( [0] => 111 [1] => 222 )
?>
```

## 5-5 hash类型操作

redis

hash是一个string类型的field和value的映射表.它的添加，删除操作都是.hash特别适合用于存储对象。

```
$redis->hSet('h', 'name', 'TK'); // 在h表中 添加name字段 value为TK
$redis->hSetNx('h', 'name', 'TK'); // 在h表中 添加name字段 value为TK 如果字段name的value存在返回false 否则返回 true
$redis->hGet('h', 'name'); // 获取h表中name字段value
$redis->hLen('h'); // 获取h表长度即字段的个数
$redis->hDel('h','email'); // 删除h表中email 字段
$redis->hKeys('h'); // 获取h表中所有字段
$redis->hVals('h'); // 获取h表中所有字段value
$redis->hGetAll('h'); // 获取h表中所有字段和value 返回一个关联数组(字段为键值)
$redis->hExists('h', 'email'); //判断email 字段是否存在与表h 不存在返回false
$redis->hSet('h', 'age', 28);
$redis->hIncrBy('h', 'age', -2); // 设置h表中age字段value加(-2) 如果value是个非数值 则返回false 否则，返回操作后的value
$redis->hIncrByFloat('h', 'age', -0.33); // 设置h表中age字段value加(-2.6) 如果value是个非数值 则返回false 否则返回操作后的value(小数点保留15位)
$redis->hMset('h', array('score' => '80', 'salary' => 2000)); // 表h 批量设置字段和value
$redis->hMGet('h', array('score','salary')); // 表h 批量获取字段的value
```

## 5-6 sort set类型操作

和 set 一样是字符串的集合，不同的是每个元素都会关联一个 double 类型的 score

redis的list类型其实就是一个每个子元素都是string类型的双向链表。**

```
$redis->zAdd('tkey', 1, 'A'); // 插入集合tkey中，A元素关联一个分数，插入成功返回1 同时集合元素不可以重复，如果元素已经存在返回 0
$redis->zRange('tkey',0,-1); // 获取集合元素，从0位置 到 -1 位置
$redis->zRange('tkey',0,-1, true); // 获取集合元素，从0位置 到 -1 位置，返回一个关联数组 带分数 array([A] => 0.01,[B] => 0.02,[D] => 0.03) 其中小数来自zAdd方法第二个参数
$redis->zDelete('tkey', 'B'); // 移除集合tkey中元素B 成功返回1 失败返回 0
$redis->zRevRange('tkey', 0, -1); // 获取集合元素，从0位置 到 -1 位置，数组按照score降序处理
```

$redis->zRevRange('tkey', 0, -1,true); // 获取集合元素，从0位置 到 -1 位置，数组按照score降序处理返回score关联数组

$redis->zRangeByScore('tkey', 0, 0.2,array('withscores' => true)); //获取几个tkey中score在区间[0,0.2]元素

,score由低到高排序,元素具有相同的score，那么会按照字典顺序排列 , withscores 控制返回关联数组

$redis->zRangeByScore('tkey', 0.1, 0.36, array('withscores' => TRUE, 'limit' => array(0, 1))); //其中limit中 0和1 表示取符合条件集合中 从0位置开始，向后扫描1个 返回关联数组

$redis->zCount('tkey', 2, 10); // 获取tkey中score在区间[2, 10]元素的个数

$redis->zRemRangeByScore('tkey', 1, 3); // 移除tkey中score在区间[1, 3](含边界)的元素

$redis->zRemRangeByRank('tkey', 0, 1); //默认元素score是递增的，移除tkey中元素 从0开始到-1位置结束

$redis->zSize('tkey'); //返回存储在key对应的有序集合中的元素的个数

$redis->zScore('tkey', 'A'); // 返回集合tkey中元素A的score值

$redis->zRank('tkey', 'A'); // 返回集合tkey中元素A的索引值 z集合中元素按照score从低到高进行排列，即最低的score index索引为0

$redis->zIncrBy('tkey', 2.5, 'A'); // 将集合tkey中元素A的score值 加 2.5

$redis->zUnion('union', array('tkey', 'tkey1')); // 将集合tkey和集合tkey1元素合并于集合union ，并且新集合中元素不能重复返回新集合的元素个数，

如果元素A在tkey和tkey1都存在，则合并后的元素A的score相加

$redis->zUnion('ko2', array('k1', 'k2'), array(5, 2)); // 集合k1和集合k2并集于k02 ，array(5,1)中元素的个数与子集合对应，然后 5 对应k1 k1每个元素score都要乘以5 ，同理1对应k2，k2每个元素score乘以1 然后元素按照递增排序，默认相同的元素score(SUM)相加

$redis->zUnion('ko2', array('k1', 'k2'), array(10, 2),'MAX'); // 各个子集乘以因子之后，元素按照递增排序，相同的元素的score取最大值(MAX) 也可以设置MIN 取最小值

$redis->zInter('ko1', array('k1', 'k2')); // 集合k1和集合k2取交集于k01 ，且按照score值递增排序 如果集合元素相同，则新集合中的元素的score值相加

$redis->zInter('ko1', array('k1', 'k2'), array(5, 1)); //集合k1和集合k2取交集于k01 ，array(5,1)中元素的个数与子集合对应，然后 5 对应k1 k1每个元素score都要乘以5 ，同理1对应k2，k2每个元素score乘以1

，然后元素score按照递增排序，默认相同的元素score(SUM)相加

$redis->zInter('ko1', array('k1', 'k2'), array(5, 1),'MAX'); // 各个子集乘以因子之后，元素score按照递增排序，相同的元素score取最大值(MAX)也可以设置MIN 取最小值

## 6. redis配置

## 6-1 持久化

对于Redis来说是存储在缓存之中的，因此缓存数据丢失问题一直是程序员们相当关注的话题，因此对缓存中的数据定时进行持久化的必要性就相当突出了，以下是redis持久化的相关配置：

1 第一种: RDB持久化方式

1.1概述

默认redis是会以快照的形式将数据持久化到磁盘的（一个二进制文件，dump.rdb，这个文件名字可以指定），在配置文件中的格式是：save N M表示在N秒之内，redis至少发生M次修改则redis抓快照到磁盘。当然我们也可以手动执行save或者bgsave（异步）做快照。

1.2实现机制

当redis需要做持久化时，redis会fork一个子进程；子进程将数据写到磁盘上一个临时RDB文件中；当子进程完成写临时文件后，将原来的RDB替换掉，这样的好处就是可以copy-on-write

1.3 相关配置

redis.conf配置文件：

1)

```
#  save ""
save 900 1
save 300 10
save 60 10000
```

2)

```
# The filename where to dump the DB
dbfilename dump.rdb
```

3)

```
# Note that you must specify a directoryhere, not a file name.
dir ./
```

2 第二种:AOF持久化方式

2.1概述

还有一种持久化方法是Append-

only:filesnapshotting方法在redis异常死掉时，最近的数据会丢失（丢失数据的多少视你save策略的配置），所以这是它最大的缺点，当业务量很大时，丢失的数据是很多的。Append-

only方法可以做到全部数据不丢失，但redis的性能就要差些。AOF就可以做到全程持久化，只需要在配置文件中开启（默认是no），appendonly

yes开启AOF之后，redis每执行一个修改数据的命令，都会把它添加到aof文件中，当redis重启时，将会读取AOF文件进行“重放”以恢复到redis关闭前的最后时刻。

LOG

Rewriting随着修改数据的执行AOF文件会越来越大，其中很多内容记录某一个key的变化情况。因此redis有了一种比较有意思的特性:在后台重建AOF文件，而不会影响client端操作。在任何时候执行BGREWRITEAOF命令，都会把当前内存中最短序列的命令写到磁盘，这些命令可以完全构建当前的数据情况，而不会存在多余的变化情况（比如状态变化，计数器变化等），缩小的AOF文件的大小。所以当使用AOF时，redis推荐同时使用BGREWRITEAOF。

AOF文件刷新的方式，有三种，参考配置参数appendfsync :appendfsync always每提交一个修改命令都调用fsync刷新到AOF文件，非常非常慢，但也非常安全;appendfsynceverysec每秒钟都调用fsync刷新到AOF文件，很快，但可能会丢失一秒以内的数据;appendfsync no依靠OS进行刷新，redis不主动刷新AOF，这样最快，但安全性就差。默认并推荐每秒刷新，这样在速度和安全上都做到了兼顾。

可能由于系统原因导致了AOF损坏，redis无法再加载这个AOF，可以按照下面步骤来修复:首先做一个AOF文件的备份，复制到其他地方;修复原始AOF文件，执行:$ redis-check-aof –fix ;可以通过diff –u命令来查看修复前后文件不一致的地方;重启redis服务。

2.2实现机制

1)Redis将数据库做个快照，遍历所有数据库，将数据库中的数据还原为跟客户端发送来的指令的协议格式的字符串，

2)然后Redis新建一个临时文件将这些快照数据保存，待快照程序结束后将临时文件名修改为正常的aof文件名，原有的文件则自动丢弃。

由于在快照进行的过程中可能存在新增的命令修改了数据库中的数据，则在快照程序结束后需要将新修改的数据追加到aof文件中，后续的从客户端过来的命令都会不断根据不同的安全级别写到磁盘里面去。这样就支持了实时的持久化，只是可能会有短时间内的数据丢失，对一般系统还是可以容忍的。

2.3 相关配置

```
redis 127.0.0.1:6380> config get*append*
1) "appendonly"
2) "yes"
3) "no-appendfsync-on-rewrite"
4) "no"
5) "appendfsync"
6) "everysec"
redis 127.0.0.1:6380> config get*aof*
1) "auto-aof-rewrite-percentage"
2) "100"
3) "auto-aof-rewrite-min-size"
4) "67108864"
```

## 6-2 事务

1 DISCARD

取消事务，放弃执行事务块内的所有命令。

2 EXEC

执行所有事务块内的命令。

3 MULTI

标记一个事务块的开始。

4 UNWATCH

取消 WATCH 命令对所有 key 的监视。

5 WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

案例：

```
<?php
$redis = new redis();
$redis -> connect('127.0.0.1',6379);
$redis -> flushAll();

$redis -> watch('number');
$redis -> unwatch();
$redis -> multi();
$redis -> set('favorite_fruit','cherry');
$redis -> incrBy('number',3);
$redis -> get('favorite_fruit');
```

```
$redis -> ping();
$redis -> discard();          // 取消事务
var_dump($redis -> exec());    // null
```

### 7. redis安全

我们可以通过 redis 的配置文件设置密码参数，这样客户端连接到 redis 服务就需要密码验证，这样可以让你的 redis 服务更安全

配置：

1、打开redis.conf，找到requirepass去掉前面的#后面改成你想要设置的密码

2、重启redis服务

3. 使用：

```
$redis = new Redis();
$redis->connect($host, $port);
$redis->auth('设置的密码');
$redis->set('needpassword', '123456');
echo $redis->get('needpassword');
```

## 四、小节

## 2. 课堂练习

## 3. 课后练习

## 4. 资料扩展