

# 目录

前言	1.1
1. 数据库篇	1.2
1.1 memcached	1.2.1
1.2 redis	1.2.2
1.3 mongodb	1.2.3
1.4 sqlite3	1.2.4
1.5 mysql	1.2.5
2. PHP篇	1.3
1.1 基础语法	1.3.1
1.2 扩展库	1.3.2
1.3 实战案例	1.3.3
1.4 算法案例	1.3.4
1.4.1 PHP的LRU实现	1.3.4.1
1.4.2 排序算法	1.3.4.2
1.4.3 链表算法	1.3.4.3
1.4.4 算法案例	1.3.4.4
1.5 设计模式	1.3.5
1.5.1 工厂模式	1.3.5.1
1.5.2 单例模式	1.3.5.2
1.5.3 注册树模式	1.3.5.3
1.5.4 适配器模式	1.3.5.4
1.5.5 策略模式	1.3.5.5
1.5.6 数据对象映射模式	1.3.5.6
1.5.7 观察者模式	1.3.5.7
1.5.8 原型模式	1.3.5.8
1.5.9 装饰器模式	1.3.5.9
1.5.10 迭代器模式	1.3.5.10
1.5.11 代理模式	1.3.5.11

---

3. Python篇	1.4
1.1 基础语法	1.4.1
1.2 扩展库	1.4.2
1.3 实战案例	1.4.3
4. 网络篇	1.5
1.1 socket	1.5.1
尾声	1.6

---



空着，没想好写什么.....

[!NOTE] 这是一个简单的Note类型的使用，所有的属性都是默认值。



emoji

[!TIP] 这是一个简单的Note类型的使用，所有的属性都是默认值。

emoji

[!WARNING] An alert of type 'comment' using style 'callout' with default settings.

emoji

[!DANGER] An alert of type 'comment' using style 'callout' with default settings.

知识点

Any content here

点击显示

要被隐藏的内容

[success] INFO

Use this for success messages.

This text is highlighted in red!

## 1. memcached

1. 安装
2. 配置说明
3. 命令

## 2. redis

1. 安装
2. 配置说明
3. 命令

## 3. mongodb

1. 安装
2. 配置说明
3. 命令

## 4. sqlite3

1. 安装
2. 配置说明
3. 命令

## 5. mysql

1. 安装
2. 配置说明
3. 命令



## 1. 安装

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

## 2. 配置说明

启动方式: /usr/local/bin/memcached -d -m 1024 -u root -l 127.0.0.1 -p 11211 -c 1024 -P /tmp/memcached.pid

- -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。

访问方式: telnet IP PORT

## 3. 命令

### 1. 设置命令

1、set: 如果set的key已经存在，该命令可以更新该key所对应的原来的数据，也就是实现更新的作用。

```
set key flags exptime bytes [noreply]  
value
```

2、add: 如果 add 的 key 已经存在，则不会更新数据(过期的 key 会更新)，之前的值将仍然保持相同，并且您将获得响应 NOT\_STORED。

```
add key flags exptime bytes [noreply]  
value
```

## 1.1 memcached

---

3、replace：如果 key 不存在，则替换失败，并且您将获得响应 NOT\_STORED。

```
replace key flags exptime bytes [noreply]  
value
```

4、append：Memcached append 命令用于向已存在 key(键) 的 value(数据值) 后面追加数据。

```
append key flags exptime bytes [noreply]  
value
```

5、prepend：Memcached prepend 命令用于向已存在 key(键) 的 value(数据值) 前面追加数据。

```
prepend key flags exptime bytes [noreply]  
value
```

6、cas：Memcached CAS (Check-And-Set 或 Compare-And-Swap) 命令用于执行一个"检查并设置"的操作。它仅在当前客户端最后一次取值后，该key 对应的值没有被其他客户端修改的情况下，才能够将值写入。检查是通过cas\_token参数进行的，这个参数是Memcach指定给已经存在的元素的一个唯一的64位值。

## 1.1 memcached

```
cas key flags exptime bytes unique_cas_token [noreply]
value

cas tp 0 900 9
ERROR          #缺少 token

cas tp 0 900 9 2
memcached
NOT_FOUND      #键 tp 不存在

set tp 0 900 9
memcached
STORED

gets tp
VALUE tp 0 9 1
memcached
END

cas tp 0 900 5 1
redis
STORED

get tp
VALUE tp 0 5
redis
END
```

参数说明如下：

- key：键值 key-value 结构中的 key，用于查找缓存值。
- flags：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- exptime：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- bytes：在缓存中存储的字节数
- unique\_cas\_token 通过 gets 命令获取的一个唯一的64位值。
- noreply（可选）：该参数告知服务器不需要返回数据
- value：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

## 2.查找命令

1、get：Memcached get 命令获取存储在 key(键) 中的 value(数据值)，如果 key 不存在，则返回空。

```
get key
get key1 key2 key3
```

## 1.1 memcached

---

2、gets：Memcached gets 命令获取带有 CAS 令牌存的 value(数据值)，如果 key 不存在，则返回空。

```
gets key  
gets key1 key2 key3
```

3、delete：Memcached delete 命令用于删除已存在的 key(键)。

```
delete key [noreply]
```

4、incr/decr：Memcached incr 与 decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。incr 与 decr 命令操作的数据必须是十进制的32位无符号整数。如果 key 不存在返回 NOT\_FOUND，如果键的值不为数字，则返回 CLIENT\_ERROR，其他错误返回 ERROR。

```
incr key increment_value  
value
```

## 3.统计命令

1、stats：Memcached stats 命令用于返回统计信息例如 PID(进程号)、版本号、连接数等。

```
stats
```

2、stats items：Memcached stats items 命令用于显示各个 slab 中 item 的数目和存储时长(最后一次访问距离现在的秒数)。

```
stats items
```

3、stats slabs：Memcached stats slabs 命令用于显示各个slab的信息，包括chunk的大小、数目、使用情况等。

```
stats slabs
```

4、stats sizes：Memcached stats sizes 命令用于显示所有item的大小和个数。该信息返回两列，第一列是 item 的大小，第二列是 item 的个数。

```
stats sizes
```

## 1.1 memcached

5、flush\_all: Memcached flush\_all 命令用于清理缓存中的所有 key=>value(键=>值) 对。该命令提供了一个可选参数 time, 用于在制定的时间后执行清理缓存操作。

```
flush_all [time] [noreply]
```

6、stats detail [on|off|dump]: 设置或者显示详细操作记录

```
stats detail [on|off|dump]
```

7、stats cachedump slab\_id limit\_num: 显示某个slab中的前limit\_num个key列表.

```
stats cachedump 7 2
```

8、stats reset: 清空统计数据

```
stats reset
```

## 4.其他信息

1、统计信息 (stats)

参数	值	描述
pid	1700	memcache服务器进程ID
uptime	5335604	服务器已运行秒数
time	1557036895	服务器当前Unix时间戳
version	1.5.10	memcache版本
libevent	2.0.21-stable	libevent版本
pointer_size	64	操作系统指针大小
rusage_user	805.137191	进程累计用户时间
rusage_system	460.201782	进程累计系统时间
max_connections	256	
curr_connections	3	当前连接数量
total_connections	7742	Memcached运行以来连接总数

rejected_connections	0	
connection_structures	5	Memcached分配的连接结构数量
reserved_fds	20	内部使用的FD数
cmd_get	6977	get命令请求次数
cmd_set	193	set命令请求次数
cmd_flush	0	flush命令请求次数
cmd_touch	0	touch命令请求次数
get_hits	6767	get命令命中次数
get_misses	210	get命令未命中次数
get_expired	1	
get_flushed	0	
delete_misses	4	delete命令未命中次数
delete_hits	4	delete命令命中次数
incr_misses	0	incr命令未命中次数
incr_hits	0	incr命令命中次数
decr_misses	0	decr命令未命中次数
decr_hits	0	decr命令命中次数
cas_misses	0	cas命令未命中次数
cas_hits	0	cas命令命中次数
cas_badval	0	使用擦拭次数
touch_hits	0	touch命令命中次数
touch_misses	0	touch命令未命中次数
auth_cmds	0	认证命令处理的次数
auth_errors	0	认证失败数目
bytes_read	9227571	读取总字节数
bytes_written	63626954	发送总字节数

limit_maxbytes	67108864	分配的内存总大小 (字节)
accepting_conns	1	接受新的连接
listen_disabled_num	0	失效的监听数
time_in_listen_disabled_us	0	
threads	4	当前线程数
conn_yields	0	连接操作主动放弃数目
hash_power_level	16	hash表等级
hash_bytes	524288	当前hash表大小
hash_is_expanding	0	hash表正在扩展
slab_reassign_rescues	0	
slab_reassign_chunk_rescues	0	
slab_reassign_evictions_nomem	0	
slab_reassign_inline_reclaim	0	
slab_reassign_busy_items	0	
slab_reassign_busy_deletes	0	
slab_reassign_running	0	
slabs_moved	0	
lru_crawler_running	0	
lru_crawler_starts	384795	
lru_maintainer_juggles	32123450	
malloc_fails	0	
log_worker_dropped	0	
log_worker_written	0	
log_watcher_skipped	0	
log_watcher_sent	0	
bytes	1714945	当前存储占用的字节数
curr_items	6	当前存储的数据总数

## 1.1 memcached

total_items	193	启动以来存储的数据总数
slab_global_page_pool	0	
expired_unfetched	9	已过期但未获取的对象数目
evicted_unfetched	0	已驱逐但未获取的对象数目
evicted_active	0	
evictions	0	LRU释放的对象数目
reclaimed	171	已过期的数据条目来存储新数据的数目
crawler_reclaimed	1	
crawler_items_checked	6406	
lru tail_reflocked	3584	
moves_to_cold	6295	
moves_to_warm	6104	
moves_within_lru	258	
direct_reclaims	0	
lru_bumps_dropped	0	

## 2、区块信息 (stats slabs)

参数	值	描述
active_slabs	10	slab数量
total_malloced	11534336	总内存数量

选择内存区块：

SLAB : 1

参数	值	描述
chunk_size	96	chunk大小 (byte)
chunks_per_page	10922	每个page的chunk数量

## 1.1 memcached

total_pages	1	page数量
total_chunks	10922	chunk总数量 (chunks_per_page*total_pages)
used_chunks	0	已被分配的chunk数量
free_chunks	10922	过期数据空出的chunk数
free_chunks_end	0	从未被使用过的chunk数
mem_requested	0	请求存储的字节数
get_hits	2	get命令命中数
cmd_set	2	set命令请求数
delete_hits	0	delete命令命中数
incr_hits	0	incr命令命中数
decr_hits	0	decr命令命中数
cas_hits	0	cas命令命中数
cas_badval	0	cas数据类型错误数
touch_hits	0	touch命令命中数

### 3、ITEMS信息 (stats items)

参数	值	描述
number	1	该slab中对象数 (不包含过期对象)
number_hot	0	
number_warm	0	
number_cold	1	
age_hot	0	
age_warm	0	
age	284	LRU队列中最老对象的过期时间
evicted	0	LRU释放对象数
evicted_nonzero	0	设置了非0时间的LRU释放对象数
evicted_time	0	最后一次LRU释放的对象存在时间

## 1.1 memcached

outofmemory	0	不能存储对象次数
tailrepairs	0	修复slabs次数
reclaimed	35	使用过期对象空间存储对象次数
expired_unfetched	0	已过期但未获取的对象数目
evicted_unfetched	0	已驱逐但未获取的对象数目
evicted_active	0	
crawler_reclaimed	0	
crawler_items_checked	850	
lru tail_reflocked	484	
moves_to_cold	1004	
moves_to_warm	968	
moves_within_lru	15	
direct_reclaims	0	
hits_to_hot	1	
hits_to_warm	19	
hits_to_cold	1012	
hits_to_temp	0	

## 1. 安裝

```
wget http://download.redis.io/releases/redis-5.0.4.tar.gz 下载最新版本
tar -zxvf redis-5.0.4.tar.gz 解压源码
cd redis-5.0.4 进入目录
make 配置
```

## 2. 配置说明

启动方式： ./redis-server ../redis.conf

- `redis.conf` 配置文件
- `daemonize no` 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
- `pidfile /var/run/redis.pid` 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
- `port 6379` 指定Redis监听端口，默认端口为6379
- `bind 127.0.0.1` 绑定的主机地址
- `timeout 300` 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
- `loglevel verbose` 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`
- `logfile stdout` 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`
- `databases 16` 设置数据库的数量，默认数据库为0，可以使用`SELECT`命令在连接上指定数据库id
- `save <seconds> <changes>` 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 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。
- `rdbcompression yes` 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
- `dbfilename dump.rdb` 指定本地数据库文件名，默认值为`dump.rdb`
- `dir ./` 指定本地数据库存放目录
- `slaveof <masterip> <masterport>` 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
- `masterauth <master-password>` 当master服务设置了密码保护时，slav服务连接master的密码
- `requirepass foobared` 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`命令提供密码，默认关闭

- `maxclients 128` 设置同一时间最大客户端连接数， 默认无限制
- `maxmemory <bytes>` 指定Redis最大内存限制
- `appendonly no` 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
- `appendfilename appendonly.aof` 指定更新日志文件名， 默认为appendonly.aof
- `appendfsync everysec` 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（折中， 默认值）
- `vm-enabled no` 指定是否启用虚拟内存机制， 默认值为no，
- `vm-swap-file /tmp/redis.swap` 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-max-memory 0` 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-page-size 32` 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；如果存储很大对象，则可以使用更大的page，如果不确定，就使用默认值
- `vm-pages 134217728` 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-max-threads 4` 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
- `glueoutputbuf yes` 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， 默认认为开启
- `hash-max-zipmap-entries 64` 、 `hash-max-zipmap-value 512`
- `activerehashing yes` 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
- `include /path/to/local.conf` 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

访问方式：redis-cli -h 192.168.1.27 -p 6379

关闭服务：redis-cli shutdown

### 3. 数据类型介绍

基本类型：字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）

范围查询类型：bitmaps, hyperloglogs 和 地理空间（geospatial）索引半径查询

- 二进制安全的字符串
- Lists: 按插入顺序排序的字符串元素的集合。他们基本上就是链表（linked lists）。
- Sets: 不重复且无序的字符串元素的集合。
- Sorted sets,类似Sets,但是每个字符串元素都关联到一个叫score浮动数值（floating number value）。里面的元素总是通过score进行着排序，所以不同的是，它是可以检索的一系列元素。（例如你可能会问：给我前面10个或者后面10个元素）。
- Hashes,由field和关联的value组成的map。field和value都是字符串的。这和Ruby、Python的hashes很像。
- Bit arrays (或者说 simply bitmaps): 通过特殊的命令，你可以将 String 值当作一系列 bits 处理：可以设置和清除单独的 bits，数出所有设为 1 的 bits 的数量，找到最前的被设为 1 或 0 的 bit，等等。
- HyperLogLogs: 这是被用于估计一个 set 中元素数量的概率性的数据结构。

### 3. 命令

#### 1. 键（key）

1、del：如果set的key已经存在，该命令可以更新该key所对应的原来的数据，也就是实现更新的作用。

```
DEL key [key ...]
```

2、dump：序列化给定 key，并返回被序列化的值，使用 RESTORE 命令可以将这个值反序列化为 Redis 键。

```
DUMP key
```

3、restore：反序列化给定的序列化值，并将它和给定的 key 关联。

```
RESTORE key ttl "\x00\x04liao\t\x00\xb4\x a8\xbb\xf1\x01C\x1b\xf6"
```

4、exists：返回key是否存在。1 如果key存在,0 如果key不存在

```
EXISTS key [key ...]
```

5、keys：查找所有符合给定模式pattern（正则表达式）的key。

```
KEYS pattern
```

6、expire (pexpire 毫秒)：设置key的过期时间，超过时间后，将会自动删除该key。在Redis的术语中一个key的相关超时是不确定的。1 如果成功设置过期时间。0 如果key不存在或者不能设置过期时间。

```
EXPIRE key 10
```

7、expireat (pexpireat 毫秒)：的作用和 EXPIRE类似，都用于为 key 设置生存时间。不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳 Unix timestamp。

```
EXPIREAT key 1293840000
```

8、migrate: 将 key 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功，key 保证会出现在目标实例上，而当前实例上的 key 会被删除。

```
migrate ip port key | destination-db timeout [copy] [replace] [keys key]
```

9、move: 将当前数据库的 key 移动到给定的数据库 db 当中。如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 key，或者 key 不存在于当前数据库，那么MOVE 没有任何效果。移动成功返回 1，失败则返回 0

```
move key db、move name 2
```

10、object: OBJECT 命令可以在内部调试(debugging)给出keys的内部对象，它用于检查或者了解你的keys是否用到了特殊编码的数据类型来存储空间z。当redis作为缓存使用的时候，你的应用也可能用到这些由OBJECT命令提供的信息来决定应用层的key的驱逐策略(eviction policies)

```
OBJECT subcommand [arguments [arguments ...]]  
lpush mylist "Hello World"  
object refcount mylist  
object encoding mylist  
object idletime mylist
```

- OBJECT 命令可以在内部调试(debugging)给出keys的内部对象，它用于检查或者了解你的keys是否用到了特殊编码的数据类型来存储空间z。当redis作为缓存使用的

时候，你的应用也可能用到这些由OBJECT命令提供的信息来决定应用层的key的驱逐策略(eviction policies)

- OBJECT ENCODING 该命令返回指定key对应value所使用的内部表示(representation)(译者注：也可以理解为数据的压缩方式).
- OBJECT IDLETIME 该命令返回指定key对应的value自被存储之后空闲的时间，以秒为单位(没有读写操作的请求)，这个值返回以10秒为单位的秒级别时间，这一点可能在以后的实现中改善

11、persist: 移除给定key的生存时间，将这个 key 从『易失的』(带生存时间 key )转换成『持久的』(一个不带生存时间、永不过期的 key )。

```
SET mykey "Hello"
EXPIRE mykey 10
TTL mykey
PERSIST mykey
TTL mykey
```

12、ttl (pttl 毫秒) : 返回key剩余的过期时间

```
TTL mykey
```

13、randomkey: 从当前数据库返回一个随机的key。

```
randomkey
```

14、rename: 将key重命名为newkey，如果key与newkey相同，将返回一个错误。如果newkey已经存在，则值将被覆盖。

```
RENAME mykey myotherkey
```

15、renamenx: 当且仅当 newkey 不存在时，将 key 改名为 newkey 。当 key 不存在时，返回一个错误。

```
RENAMENX mykey myotherkey
```

16、scan: 它们每次执行都只会返回少量元素，所以这些命令可以用于生产环境，而不会出现像 KEYS 或者 SMEMBERS 命令带来的可能会阻塞服务器的问题。

```
SCAN cursor [MATCH pattern] [COUNT count]
```

17、touch: 修改指定key(s) 最后访问时间 若key不存在，不做操作

```
TOUCH key [key ...]
```

18、unlink: 该命令和DEL十分相似：删除指定的key(s),若key不存在则该key被跳过。但是，相比DEL会产生阻塞，该命令会在另一个线程中回收内存，因此它是非阻塞的。这也是该命令名字的由来：仅将keys从keyspace元数据中删除，真正的删除会在后续异步操作。

```
UNLINK key [key ...]
```

## 2.字符串 (string)

1、append: 如果 key 已经存在，并且值为字符串，那么这个命令会把 value 追加到原来值 (value) 的结尾。如果 key 不存在，那么它将首先创建一个空字符串的key，再执行追加操作，这种情况 APPEND 将类似于 SET 操作。返回append后字符串值 (value) 的长度。

```
APPEND key value
```

2、bitcount: 统计字符串被设置为1的bit数.

```
BITCOUNT key [start end]
```

3、decr/incr: 对key对应的数字做减1操作。如果key不存在，那么在操作之前，这个key对应的值会被置为0。如果key有一个错误类型的value或者是一个不能表示成数字的字符串，就返回错误。这个操作最大支持在64位有符号的整型数字。

```
DECR key
```

4、decrby/incrby: 将key对应的数字减decrement。如果key不存在，操作之前，key就会被置为0。如果key的value类型错误或者是个不能表示成数字的字符串，就返回错误。这个操作最多支持64位有符号的正型数字。

```
DECRBY mykey 5/DECRBY mykey 5
```

5、get: 返回key的value。如果key不存在，返回特殊值nil。如果key的value不是string，就返回错误，因为GET只处理string类型的values。

```
GET mykey
```

6、getbit：返回key对应的string在offset处的bit值 当offset超出了字符串长度的时候，这个字符串就被假定为由0比特填充的连续空间。

```
GETBIT mykey 100
```

7、getrange：这个命令是被改成GETRANGE的，在小于2.0的Redis版本中叫SUBSTR。返回key对应的字符串value的子串，这个子串是由start和end位移决定的（两者都在string内）。可以用负的位移来表示从string尾部开始数的下标。所以-1就是最后一个字符，-2就是倒数第二个，以此类推。

```
GETRANGE key start end
```

8、getset：自动将key对应到value并且返回原来key对应的value。如果key存在但是对应的value不是字符串，就返回错误。返回之前的旧值，如果之前Key不存在将返回nil。

```
INCR mycounter  
GETSET mycounter "0"  
GET mycounter
```

9、mget：返回所有指定的key的value。对于每个不对应string或者不存在的key，都返回特殊值nil。正因为此，这个操作从来不会失败。

```
MGET key1 key2 nonexisting
```

10、mset：对应给定的keys到他们相应的values上。MSET会用新的value替换已经存在的value，就像普通的SET命令一样。如果你不想覆盖已经存在的values，请参看命令MSETNX。

```
MSET key1 "Hello" key2 "World"
```

11、msetnx：对应给定的keys到他们相应的values上。只要有一个key已经存在，MSETNX一个操作都不会执行。由于这种特性，MSETNX可以实现要么所有的操作都成功，要么一个都不执行，这样可以用来设置不同的key，来表示一个唯一的对象的不同字段。

```
MSETNX key1 "Hello" key2 "there"
```

12、set: 将键key设定为指定的“字符串”值。如果key已经保存了一个值，那么这个操作会直接覆盖原来的值，并且忽略原始类型。当set命令执行成功之后，之前设置的过期时间都将失效

```
SET key value [EX seconds] [PX milliseconds] [NX|XX]
set phone xiaomi ex 50
```

13、setbit: 设置或者清空key的value(字符串)在offset处的bit值。

```
SETBITS mykey 7 1
```

14、setex: 设置key对应字符串value，并且设置key在给定的seconds时间之后超时过期。这个命令等效于执行下面的命令：

```
SET mykey value
EXPIRE mykey seconds
SETEX mykey 10 "Hello"
```

### 3.列表 (list)

1、blpop/brpop：当没有元素的时候会弹出一个 nil 的多批量值，并且 timeout 过期。当有元素弹出时会返回一个双元素的多批量值，其中第一个元素是弹出元素的 key，第二个元素是 value。

```
BLPOP key [key ...] timeout
```

2、brpoplpush/rpoplpush：Redis Brpoplpush 命令从列表中取出最后一个元素，并插入到另外一个列表的头部；如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

```
BRPOPLPUSH msg receiver 500/BRPOPLPUSH LIST1 ANOTHER_LIST TIMEOUT
```

3、lindex：返回列表里的元素的索引 index 存储在 key 里面。下标是从0开始索引的，所以 0 是表示第一个元素，1 表示第二个元素，并以此类推。负数索引用于指定从列表尾部开始索引的元素。在这种方法下，-1 表示最后一个元素，-2 表示倒数第二个元素，并以此往前推。

```
LINDEX mylist 0
```

4、linsert: 把 value 插入存于 key 的列表中在基准值 pivot 的前面或后面。当 key 不存在时，这个list会被看作是空list，任何操作都不会发生。当 key 存在，但保存的不是一个list的时候，会返回error。

```
RPUSH mylist "Hello"
(integer) 1
RPUSH mylist "World"
(integer) 2
LINSERT mylist BEFORE "World" "There"
(integer) 3
LRANGE mylist 0 -1
1) "Hello"
2) "There"
3) "World"
```

5、llen: 返回存储在 key 里的list的长度。如果 key 不存在，那么就被看作是空list，并且返回长度为 0。当存储在 key 里的值不是一个list的话，会返回error。

```
LLEN mylist
```

其他命令：

- 1、LPOP key 移除并且返回 key 对应的 list 的第一个元素。
- 2、LPUSH key value [value ...] 将所有指定的值插入到存于 key 的列表的头部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。如果 key 对应的值不是一个 list 的话，那么会返回一个错误。
- 3、LPUSHX key value 只有当 key 已经存在并且存着一个 list 的时候，在这个 key 下面的 list 的头部插入 value。与 LPUSH 相反，当 key 不存在的时候不会进行任何操作。
- 4、LRANGE key start stop 返回存储在 key 的列表里指定范围内的元素。start 和 end 偏移量都是基于0的下标，即list的第一个元素下标是0 (list的表头)，第二个元素下标是1，以此类推。
- 5、LREM key count value 从存于 key 的列表里移除前 count 次出现的值为 value 的元素。这个 count 参数通过下面几种方式影响这个操作：
  - count > 0: 从头往尾移除值为 value 的元素。
  - count < 0: 从尾往头移除值为 value 的元素。
  - count = 0: 移除所有值为 value 的元素
- 6、LSET key index value 设置 index 位置的list元素的值为 value。
- 7、LTRIM key start stop 修剪(trim)一个已存在的 list，这样 list 就会只包含指定范围的指定元素。start 和 stop 都是由0开始计数的，这里的 0 是列表里的第一个元素 (表头)，1 是第二个元素，以此类推。

- 8、RPOP key 移除并返回存于 key 的 list 的最后一个元素。
- 9、RPUSH key value [value ...] 向存于 key 的列表的尾部插入所有指定的值。如果 key 不存在，那么会创建一个空的列表然后再进行 push 操作。当 key 保存的不是一个列表，那么会返回一个错误。
- 10、RPUSHX key value 将值 value 插入到列表 key 的表尾, 当且仅当 key 存在并且是一个列表。和 RPUSH 命令相反, 当 key 不存在时, RPUSHX 命令什么也不做。

### 4.哈希 (hash)

1、hdel：从 key 指定的哈希集中移除指定的域。在哈希集中不存在的域将被忽略。如果 key 指定的哈希集不存在，它将被认为是一个空的哈希集，该命令将返回0。

```
HDEL key field [field ...]
```

2、hexists：返回hash里面field是否存在

```
HEXISTS myhash field2
```

3、hget：返回 key 指定的哈希集中该字段所关联的值

```
HGET key field
```

4、hgetall：返回 key 指定的哈希集中所有的字段和值。

```
HGETALL myhash
```

5、hincrby/hincrbyfloat：增加 key 指定的哈希集中指定字段的数值。如果 key 不存在，会创建一个新的哈希集并与 key 关联。如果字段不存在，则字段的值在该操作执行前被设置为 0

HINCRBY 支持的值的范围限定在 64位 有符号整数

```
HINCRBY key field increment  
hincrby people age 1
```

6、hkeys：返回 key 指定的哈希集中所有字段的名字。

```
HKEYS key
```

7、hlen：返回 key 指定的哈希集包含的字段的数量。

```
HLEN key
```

8、hmget：返回 key 指定的哈希集中指定字段的值。

```
HMGET key field [field ...]
```

9、hmset：设置 key 指定的哈希集中指定字段的值。该命令将重写所有在哈希集中存在的字段。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联

```
HMSET key field value [field value ...]
```

10、hset：设置 key 指定的哈希集中指定字段的值。

```
HSET key field value
```

11、hsetnx：只在 key 指定的哈希集中不存在指定的字段时，设置字段的值。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。如果字段已存在，该操作无效果。

```
HSETEX key field value
```

12、hstrlen：返回hash指定field的value的字符串长度，如果hash或者field不存在，返回0.

```
HSTRLEN key field
```

13、hvals：返回 key 指定的哈希集中所有字段的值。

```
HVALS key
```

## 5.集合（set）

1、sadd：添加一个或多个指定的member元素到集合的 key 中。指定的一个或者多个元素 member 如果已经在集合key中存在则忽略。如果集合key不存在，则新建集合key，并添加 member元素到集合key中。

```
SADD key member [member ...]
```

2、scard：返回集合存储的key的基数 (集合元素的数量).

```
SCARD key
```

3、sdiff：返回一个集合与给定集合的差集的元素.

```
SDIFF key [key ...]
```

4、sdiffstore：该命令类似于 SDIFF, 不同之处在于该命令不返回结果集，而是将结果存放在destination集合中.

如果destination已经存在, 则将其覆盖重写.

```
SDIFFSTORE destination key [key ...]
```

5、sinter：返回指定所有的集合的成员的交集.

```
SINTER key [key ...]
```

6、sinterstore：这个命令与SINTER命令类似, 但是它并不是直接返回结果集,而是将结果保存在 destination集合中.

```
SINTERSTORE destination key [key ...]
```

7、sismember：返回成员 member 是否是存储的集合 key的成员.

```
SISMEMBER key member
```

8、smembers：返回key集合所有的元素.该命令的作用与使用一个参数的SINTER 命令作用相同.

```
SMEMBERS key
```

9、smove：将member从source集合移动到destination集合中. 对于其他的客户端,在特定的时间元素将会作为source或者destination集合的成员出现.

```
SMOVE source destination member  
SMOVE myset myotherset "two"
```

10、**spop**: 从存储在key的集合中移除并返回一个或多个随机元素。此操作与SRANDMEMBER类似，它从一个集合中返回一个或多个随机元素，但不删除元素。

```
SPOP key [count]
```

11、**srandmember**: 那么随机返回key集合中的一个元素.

```
SRANDMEMBER key [count]
```

12、**srem**: 在key集合中移除指定的元素. 如果指定的元素不是key集合中的元素则忽略如果key集合不存在则被视为一个空的集合，该命令返回0.

```
SREM key member [member ...]
```

13、**sunion**: 返回给定的多个集合的并集中的所有成员.

```
SUNION key [key ...]
```

14、**sunionstore**: 该命令作用类似于SUNION命令,不同的是它并不返回结果集,而是将结果存储在destination集合中.如果destination 已经存在,则将其覆盖.

```
SUNIONSTORE destination key [key ...]
```

### 6.有序集合 (sorted set)

1、**zadd**: 将所有指定成员添加到键为key有序集合 (sorted set) 里面。添加时可以指定多个分数/成员 (score/member) 对。如果指定添加的成员已经是有序集合里面的成员，则会更新改成员的分数 (scrore) 并更新到正确的排序位置。

```
ZADD key [NX|XX] [CH] [INCR] score member [score member ...]
```

2、**zcard**: 返回key的有序集元素个数。

```
ZCARD key
```

3、**zcount**: 指定分数范围的元素个数。

```
ZCOUNT key min max
```

4、zincrby：为有序集key的成员member的score值加上增量increment。如果key中不存在member，就在key中添加一个member，score是increment（就好像它之前的score是0.0）。如果key不存在，就创建一个只含有指定member成员的有序集合

```
ZINCRBY key increment member
```

5、zinterstore：计算给定的numkeys个有序集合的交集，并且把结果放到destination中。在给定要计算的key和其它参数之前，必须先给定key个数(numberkeys)。

```
ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight] [SUM|MIN|MAX]
redis> ZADD zset1 1 "one"
(integer) 1
redis> ZADD zset1 2 "two"
(integer) 1
redis> ZADD zset2 1 "one"
(integer) 1
redis> ZADD zset2 2 "two"
(integer) 1
redis> ZADD zset2 3 "three"
(integer) 1
redis> ZINTERSTORE out 2 zset1 zset2 WEIGHTS 2 3
(integer) 2
redis>ZRANGE out 0 -1 WITHSCORES
1) "one"
2) "5"
3) "two"
4) "10"
```

6、zpopmax：删除并返回有序集合key中的最多count个具有最高得分的成员。

```
ZPOPMAX key [count]
```

7、zpopmin：删除并返回有序集合key中的最多count个具有最低得分的成员。

```
ZPOPMIN key [count]
```

8、zrange：返回存储在有序集合key中的指定范围的元素。

```
ZRANGE key start stop [WITHSCORES]
```

8、zrevrange：返回存储在有序集合key中的指定范围的元素。其中成员的位置按score值递减(从大到小)来排列。

```
ZREVRANGE key start stop [WITHSCORES]
```

9、zrank：返回有序集key中成员member的排名。

```
ZRANK key member
```

10、zrevrank：返回有序集key中成员member的排名，其中有序集成员按score值从大到小排列。排名以0为底，也就是说，score值最大的成员排名为0。

```
ZREVRANK key member
```

11、zrem：返回的是从有序集合中删除的成员个数，不包括不存在的成员。

```
ZREM key member [member ...]
```

## 7.地图（Geo）

1、geoadd：将指定的地理空间位置（纬度、经度、名称）添加到指定的key中。这些数据将会存储到sorted set这样的目的是为了方便使用GEORADIUS或者GEORADIUSBYMEMBER命令对数据进行半径查询等操作。

```
GEOADD key longitude latitude member [longitude latitude member ...]
```

2、geodist：返回两个给定位置之间的距离。如果两个位置之间的其中一个不存在，那么命令返回空值。

```
GEODIST key member1 member2 [unit]
```

3、geohash：返回一个或多个位置元素的 Geohash 表示。

```
GEOHASH key member [member ...]
```

4、geopos：从key里返回所有给定位置元素的位置（经度和纬度）。

```
GEOPOS key member [member ...]
```

5、georadius：以给定的经纬度为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离的所有位置元素

```
GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]
```

6、georadiusbymember：这个命令和 GEORADIUS 命令一样，都可以找出位于指定范围内的元素，但是 GEORADIUSBYMEMBER 的中心点是由给定的位置元素决定的，而不是像 GEORADIUS 那样，使用输入的经度和纬度来决定中心点指定成员的位置被用作查询的中心。

```
GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]
```

## 8.HyperLogLog

1、pfadd：将除了第一个参数以外的参数存储到以第一个参数为变量名的HyperLogLog 结构中.

```
PFADD key element [element ...]  
PFADD h11 a b c d e f g
```

2、pfcount：当参数为一个key时,返回存储在HyperLogLog结构体的该变量的近似基数,如果该变量不存在,则返回0.

```
PFCOUNT key [key ...]  
PFCOUNT h11
```

3、pfmerge：将多个 HyperLogLog 合并 (merge) 为一个 HyperLogLog , 合并后的 HyperLogLog 的基数接近于所有输入 HyperLogLog 的可见集合 (observed set) 的并集.

```
PFMERGE destkey sourcekey [sourcekey ...]  
PFMERGE h113 h111 h112
```

## 4、大量数据插入

data.txt文件

```
SET Key0 Value0  
SET Key1 Value1  
...  
SET KeyN ValueN
```

```
cat data.txt | redis-cli -h localhost -p 6380 -a liao --pipe
```

```
unix2dos d1.txt 转码 cat d1.txt | redis-cli -h localhost -p 6380 -a liao [ --pipe ]
```

## 5、过期策略

Keys的过期时间 通常Redis keys创建时没有设置相关过期时间。他们会一直存在，除非使用显示的命令移除，例如，使用DEL命令。

EXPIRE一类命令能关联到一个有额外内存开销的key。当key执行过期操作时，Redis会确保按照规定时间删除他们。

key的过期时间和永久有效性可以通过EXPIRE和PERSIST命令（或者其他相关命令）来进行更新或者删除过期时间。

过期精度 在 Redis 2.4 及以前版本，过期时间可能不是十分准确，有0-1秒的误差。

从 Redis 2.6 起，过期时间误差缩小到0-1毫秒。

过期和持久 Keys的过期时间使用Unix时间戳存储(从Redis 2.6开始以毫秒为单位)。这意味着即使Redis实例不可用，时间也是一直在流逝的。

要想过期的工作处理好，计算机必须采用稳定的时间。如果你将RDB文件在两台时钟不同步的电脑间同步，有趣的事会发生（所有的keys装载时就会过期）。

即使正在运行的实例也会检查计算机的时钟，例如如果你设置了一个key的有效期是1000秒，然后设置你的计算机时间为未来2000秒，这时key会立即失效，而不是等1000秒之后。

Redis如何淘汰过期的keys Redis keys过期有两种方式：被动和主动方式。

当一些客户端尝试访问它时，key会被发现并主动的过期。

当然，这样是不够的，因为有些过期的keys，永远不会访问他们。无论如何，这些keys应该过期，所以定时随机测试设置keys的过期时间。所有这些过期的keys将会从密钥空间删除。

具体就是Redis每秒10次做的事情：

测试随机的20个keys进行相关过期检测。删除所有已经过期的keys。如果有大于25%的keys过期，重复步奏1. 这是一个平凡的概率算法，基本上的假设是，我们的样本是这个密钥控件，并且我们不断重复过期检测，直到过期的keys的百分比低于25%，这意味着，在任何给定的时刻，最多会清除1/4的过期keys。

在复制AOF文件时如何处理过期 为了获得正确的行为而不牺牲一致性，当一个key过期，DEL将会随着AOF文字一起合成到所有附加的slaves。在master实例中，这种方法是

集中的，并且不存在一致性错误的机会。

然而，当slaves连接到master时，不会独立过期keys（会等到master执行DEL命令），他们任然会在数据集里面存在，所以当slave当选为master时淘汰keys会独立执行，然后成为master。

## 5、事务

MULTI、EXEC、DISCARD 和 WATCH 是 Redis 事务相关的命令。事务可以一次执行多个命令，并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

EXEC 命令负责触发并执行事务中的所有命令：

如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC，那么事务中的所有命令都不会被执行。另一方面，如果客户端成功在开启事务之后执行 EXEC，那么事务中的所有命令都会被执行。

放弃事务 当执行 DISCARD 命令时，事务会被放弃，事务队列会被清空，并且客户端会从事务状态中退出：

WATCH 命令可以为 Redis 事务提供 check-and-set (CAS) 行为。

被 WATCH 的键会被监视，并会发觉这些键是否被改动过了。如果有至少一个被监视的键在 EXEC 执行之前被修改了，那么整个事务都会被取消，EXEC 返回nil-reply来表示事务已经失败。

## redis 主从配置操作步骤详解

主服务器：127.0.0.1 6379 从服务器：127.0.0.1 6380

1、首先，修改 Master上的如下配置（在 redis.conf 修改配置）：

a. 禁用 主服务器 snapshot

- save 900 1 #禁用Snapshot
- save 300 10
- save 60 10000

b. 禁用 AOF

- appendonly no #禁用AOF （该操作默认就是禁用的） c. 设置 master 密码 （可选）

- requirepass liao

2、修改 Slave (redis6666) 上的如下配置：

a. 启动 从服务器的 Snapshot

- save 900 1 #启用Snapshot (默认开启)
- save 300 10
- save 60 10000

b. 启动 从服务器 AOF

- appendonly yes #启用AOF (默认关闭)
- appendfilename appendonly.aof #AOF文件的名称

c. 设置 slaveof

- slaveof 127.0.0.1 6379
- masterauth liao

查看主服务器连接状态： info replication

## 四、备份数据测试

1、Master 插入一条数据：

```
127.0.0.1:6666> set name helloworld
OK
```

2、Slave 获取数据：

```
127.0.0.1:7777> get name
"helloworld"
```

操作到这里： redis 主从配置成功了。

## 五、容灾机制测试

1、Master 挂掉了：

```
127.0.0.1:6666> SHUTDOWN # 或者 直接 kill 掉 Master 进程
not connected>
```

2、Slave 再次获取数据：

```
127.0.0.1:7777> get name  
"helloworld"
```

### 3、数据恢复

先在从服务器执行： save 将Slave上数据文件 dump.rdb 和 appendonly.aof 复制到 Master 目录上。

### 4、启动 Master， 分别在 Master 和 Slave 获取数据。

注意： 操作步骤3、4 不可互换， 否则会造成数据丢失。 Master 启动后， Slave 会自动去同步数据。

#### Redis的过期策略

我们都知道， Redis是key-value数据库， 我们可以设置Redis中缓存的key的过期时间。 Redis的过期策略就是指当Redis中缓存的key过期了， Redis如何处理。 过期策略通常有以下三种：

- 定时过期： 每个设置过期时间的key都需要创建一个定时器， 到过期时间就会立即清除。 该策略可以立即清除过期的数据， 对内存很友好； 但是会占用大量的CPU资源去处理过期的数据， 从而影响缓存的响应时间和吞吐量。
- 惰性过期： 只有当访问一个key时， 才会判断该key是否已过期， 过期则清除。 该策略可以最大化地节省CPU资源， 却对内存非常不友好。 极端情况可能出现大量的过期key没有再次被访问， 从而不会被清除， 占用大量内存。
- 定期过期： 每隔一定的时间， 会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。 该策略是前两者的一个折中方案。 通过调整定时扫描的时间间隔和每次扫描的限定耗时， 可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

(expires字典会保存所有设置了过期时间的key的过期时间数据， 其中， key是指向键空间中的某个键的指针， value是该键的毫秒精度的UNIX时间戳表示的过期时间。 键空间是指该Redis集群中保存的所有键。 )

Redis中同时使用了惰性过期和定期过期两种过期策略。

#### Redis的内存淘汰策略

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时， 怎么处理需要新写入且需要申请额外空间的数据。

- noeviction： 当内存不足以容纳新写入数据时， 新写入操作会报错。
- allkeys-lru： 当内存不足以容纳新写入数据时，在键空间中， 移除最近最少使用的key。
- allkeys-random： 当内存不足以容纳新写入数据时，在键空间中， 随机移除某个

key。

- volatile-lru: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。
- volatile-random: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。
- volatile-ttl: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

## 1. 安装

```
 wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-4.0.9.tgz 下载最新版本  
 tar -zxvf mongodb-linux-x86_64-4.0.9.tgz 解压源码  
 mv mongodb-linux-x86_64-4.0.9 .. /mongodb 进入目录  
 vim /etc/profile source /etc/profile 配置
```

## 2. 配置说明

启动方式：

```
 mongod --dbpath=/data/db --port=27017 --fork --logpath=/home/logs/mongodb/mongod.log  
 dbpath = /data/db  
 logpath = /data/db/mongod.log  
 logappend = true  
 bind_ip = 0.0.0.0  
 port = 27017  
 fork = true
```

开启服务： mongod -f mongodb.conf 关闭服务： mongod -f mongodb.conf --shutdown

## 3. 存贮关系

数据库、集合、文档相当于mysql中数据库、表、记录的关系

## 4. 命令

### 1. 数据库

#### 1、创建数据库

```
 use DATABASE_NAME  
 use runoob  
 db  
 db.runoob.insert({"name": "菜鸟教程"})
```

#### 2、删除数据库

```
 db.dropDatabase()
```

### 2. 集合

## 1、创建集合

```
db.createCollection(name, options)
db.createCollection("mycol", { capped : true, autoIndexId : true, size : 6142800, max : 10000 } )
db.mycol2.insert({ "name" : "菜鸟教程" }) #插入记录自动创建mycol2集合
```

字段	类型	描述
capped	布尔	(可选) 如果为 true, 则创建固定集合。固定集合是指有着固定大小的集合, 当达到最大值时, 它会自动覆盖最早的文档。当该值为 true 时, 必须指定 size 参数。
autoIndexId	布尔	(可选) 如为 true, 自动在 _id 字段创建索引。默认为 false。
size	数值	(可选) 为固定集合指定一个最大值(以字节计)。如果 capped 为 true, 也需要指定该字段。
max	数值	(可选) 指定固定集合中包含文档的最大数量。

在插入文档时, MongoDB 首先检查固定集合的 size 字段, 然后检查 max 字段。 2、删除集合

```
db.collection.drop()
db.mycol2.drop()
```

## 2. 文档

### 1、插入文档

```
db.COLLECTION_NAME.insert(document)
db.col.insert([{title: 'MongoDB 教程', description: 'MongoDB 是一个 Nosql 数据库', by: '菜鸟教程', url: 'http://www.rungoob.com', tags: ['mongodb', 'database', 'NoSQL'], likes: 100}, {title: 'MongoDB 教程', description: 'MongoDB 是一个 Nosql 数据库', by: '菜鸟教程', url: 'http://www.rungoob.com', tags: ['mongodb', 'database', 'NoSQL'], likes: 100}])
db.col.insert([{}, {}]) #插入多个
db.collection.insertOne({"a": 3})
db.collection.insertMany([{"b": 3}, {"c": 4}])
```

### 2、更新文档

## 1.3 mongodb

```
db.collection.update(<query>, <update>, {upsert: <boolean>, multi: <boolean>, writeConcern: <document>})
db.collection.save(<document>, {writeConcern: <document>}) //以下实例中我们替换了 _id 为
56064f89ade2f21f36b03136 的文档数据:
```

### 3、删除文档

```
db.collection.remove(<query>, {justOne: <boolean>, writeConcern: <document>})
db.col.remove({ "title": "mongodb"}, {"justOne": true}) //db.col.deleteMany({ status : "A" })
| db.col.deleteOne( { status: "D" } )
db.col.remove({}) //删除所有数据 |db.col.deleteMany({})
db.repairDatabase() //回收磁盘空间
db.runCommand({ repairDatabase: 1 }) //回收磁盘空间
```

### 4、查询文档

```
db.collection.find(query, projection)
db.col.find().pretty()
db.col.find({"likes":{$lte:50}}).pretty()
db.col.find({key1:value1, key2:value2}).pretty() //and
db.col.find({$or: [{key1: value1}, {key2:value2}]}).pretty() //or
db.col.find({"likes": {$gt:50}, $or: [{"by": "菜鸟教程"}, {"title": "MongoDB 教程"}]}).pretty()
db.col.find({title:/教/})
db.col.find({title:/^教/})
db.col.find({title:/教$/})
db.col.find({"title" : {$type : 'string'}})
db.COLLECTION_NAME.find().limit(NUMBER)
db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

在 MongoDB 中使用 sort() 方法对数据进行排序，sort() 方法可以通过参数指定排序的字段，并使用 1 和 -1 来指定排序的方式，其中 1 为升序排列，而 -1 是用于降序排列。

```
db.COLLECTION_NAME.find().sort({KEY:1})
```

操作	格式	范例	RDBMS中的类似语句
等于	{<key>:<value>}	db.col.find({"by": "菜鸟教程"}).pretty()	where by = '菜鸟教程'
小于	{<key>:{\$lt: <value>}}	db.col.find({"likes": {\$lt:50}}).pretty()	where likes < 50
小于或	{<key>:{\$lte: <value>}}	db.col.find({"likes": {\$lte:50}}).pretty()	where likes <= 50

等于	<value>}}	{\$lte:50}}).pretty()	50
大于	{<key>:{\$gt:<value>}}	db.col.find({"likes": {\$gt:50}}).pretty()	where likes > 50
大于或等于	{<key>:{\$gte:<value>}}	db.col.find({"likes": {\$gte:50}}).pretty()	where likes >= 50
不等于	{<key>:{\$ne:<value>}}	db.col.find({"likes": {\$ne:50}}).pretty()	where likes != 50

## 5、索引

```
db.collection.createIndex(keys, options)
db.col.createIndex({"title":1,"description":-1})
```

字段	类型	描述
background	Boolean	建索引过程会阻塞其它数据库操作，background可指定以后台方式创建索引，即增加 "background" 可选参数。 "background" 默认值为false。
unique	Boolean	建立的索引是否唯一。指定为true创建唯一索引。默认值为false.
name	string	索引的名称。如果未指定，MongoDB的通过连接索引的字段名和排序顺序生成一个索引名称。
sparse	Boolean	对文档中不存在的字段数据不启用索引；这个参数需要特别注意，如果设置为true的话，在索引字段中不会查询出不包含对应字段的文档。默认值为 false.
expireAfterSeconds	integer	指定一个以秒为单位的数值，完成 TTL 设定，设定集合的生存时间。
v	index version	索引的版本号。默认的索引版本取决于 mongod 创建索引时运行的版本。
weights	document	索引权重值，数值在 1 到 99,999 之间，表示该索引相对于其他索引字段的得分权重。
default_language	string	对于文本索引，该参数决定了停用词及词干和词器的规则的列表。默认为英语
language_override	string	对于文本索引，该参数指定了包含在文档中的字段名，语言覆盖默认的language，默认值为 language.

## 6、聚合

```
db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}] ) //等同于
下
select by_user, count(*) from mycol group by by_user
```

表达式	描述	实例
\$sum	计算总和。	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}])
\$avg	计算平均值	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}])
\$min	获取集合中所有文档对应值得最小值。	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}])
\$max	获取集合中所有文档对应值得最大值。	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}])
\$push	在结果文档中插入值到一个数组中。	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$push: "\$url"}}}])
\$addToSet	在结果文档中插入值到一个数组中，但不创建副本。	db.mycol.aggregate([{\$group : {_id : "\$by_user", url : {\$addToSet : "\$url"}}}])
\$first	根据资源文档的排序获取第一个文档数据。	db.mycol.aggregate([{\$group : {_id : "\$by_user", first_url : {\$first : "\$url"}}}])
\$last	根据资源文档的排序获取最后一个文档数据	db.mycol.aggregate([{\$group : {_id : "\$by_user", last_url : {\$last : "\$url"}}}])

## 7、备份与恢复

mongodb操作命令： mongodump -h 192.168.0.171:5000 -d game -o ./back 导出

mongodump -h 192.168.0.171:5000 -o

/data/wwwroot/jpoa.ifanspoker.com/tmp/backmongo 导出

mongorestore -h localhost:27017 -d game ./game 导入 mongorestore -h

localhost:27017 /备份位置

## 8、ObjectId

ObjectId 是一个12字节 BSON 类型数据，有以下格式：

前4个字节表示时间戳

接下来的3个字节是机器标识码

紧接着的两个字节由进程id组成 (PID)

最后三个字节是随机数。

MongoDB中存储的文档必须有一个"\_id"键。这个键的值可以是任何类型的，默认是个 ObjectId对象。

在一个集合里面，每个文档都有唯一的"\_id"值，来确保集合里面每个文档都能被唯一标识。

MongoDB采用ObjectId，而不是其他比较常规的做法（比如自动增加的主键）的主要原因，因为在多个服务器上同步自动增加主键值既费力还费时。

```
newObjectId = ObjectId()  
ObjectId("5349b4ddd2781d08c09890f4").getTimestamp()  
new ObjectId().str
```

## 0. 介绍

SQLite 是一个软件库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。SQLite 是在世界上最广泛部署的 SQL 数据库引擎。

## 1. 安装

```
tar xvzf sqlite-autoconf-3071502.tar.gz    解压源码
cd sqlite-autoconf-3071502    进入目录
./configure --prefix=/usr/local
make && make install    编译安装
```

## 2. 配置说明

进入服务:sqlite3

## 3. 命令

### 1.数据库

```
sqlite3 DatabaseName.db          //创建数据库
sqlite>.databases                //检查数据库
sqlite>.quit                      //退出
sqlite3 rule.db .dump > aa.sql    //导出数据库成sql语句
sqlite3 tools.db < aa.sql        //导入sql语句到库中
```

### 2.附加数据库

假设这样一种情况，当在同一时间有多个数据库可用，您想使用其中的任何一个。SQLite 的 ATTACH DATABASE 语句是用来选择一个特定的数据库，使用该命令后，所有的 SQLite 语句将在附加的数据库下执行。

```
sqlite>ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';
```

注：数据库名称 main 和 temp 被保留用于主数据库和存储临时表及其他临时数据对象的数据库。

### 3.分离数据库

SQLite的 DETACH DTABASE 语句是用来把命名数据库从一个数据库连接分离和游离出来，连接是之前使用 ATTACH 语句附加的。如果同一个数据库文件已经被附加上多个别

名，DETACH 命令将只断开给定名称的连接，而其余的仍然有效。您无法分离 main 或 temp 数据库。

```
sqlite>DETACH DATABASE 'Alias-Name';
sqlite>DETACH DATABASE 'TEST';
```

#### 4.创建表/删除表

```
CREATE TABLE database_name.table_name(
    column1 datatype PRIMARY KEY(one or more columns),
    column2 datatype,
    column3 datatype,
    ....
    columnN datatype,
);
```

```
create table yh_user( id INTEGER primary key autoincrement, username varchar(255) default "", password char(32) default "", real_name varchar(20) default "", sex tinyint default 0 );
```

```
DROP TABLE database_name.table_name;
```

查看表信息

```
sqlite> SELECT sql FROM sqlite_master WHERE type = 'table' AND tbl_name = 'COMPANY';
```

#### 5.数据类型

SQLite 数据类型是一个用来指定任何对象的数据类型的属性。SQLite 中的每一列，每个变量和表达式都有相关的数据类型。

您可以在创建表的同时使用这些数据类型。SQLite 使用一个更普遍的动态类型系统。在 SQLite 中，值的数据类型与值本身是相关的，而不是与它的容器相关。

SQLite 存储类 每个存储在 SQLite 数据库中的值都具有以下存储类之一：

存储类	描述
NULL	值是一个 NULL 值。
INTEGER	值是一个带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。
REAL	值是一个浮点值，存储为 8 字节的 IEEE 浮点数字。
TEXT	值是一个文本字符串，使用数据库编码（UTF-8、UTF-16BE 或

TEXT	UTF-16LE) 存储。
BLOB	值是一个 blob 数据，完全根据它的输入存储。

### SQLite 亲和(Affinity)类型

SQLite支持列的亲和类型概念。任何列仍然可以存储任何类型的数据，当数据插入时，该字段的数据将会优先采用亲缘类型作为该值的存储方式。SQLite目前的版本支持以下五种亲缘类型：

亲和类型	描述
TEXT	数值型数据在被插入之前，需要先被转换为文本格式，之后再插入到目标字段中。
NUMERIC	当文本数据被插入到亲缘性为NUMERIC的字段中时，如果转换操作不会导致数据信息丢失以及完全可逆，那么SQLite就会将该文本数据转换为INTEGER或REAL类型的数据，如果转换失败，SQLite仍会以TEXT方式存储该数据。对于NULL或BLOB类型的新数据，SQLite将不做任何转换，直接以NULL或BLOB的方式存储该数据。需要额外说明的是，对于浮点格式的常量文本，如"30000.0"，如果该值可以转换为INTEGER同时又不会丢失数值信息，那么SQLite就会将其转换为INTEGER的存储方式。
INTEGER	对于亲缘类型为INTEGER的字段，其规则等同于NUMERIC，唯一差别是在执行CAST表达式时。
REAL	其规则基本等同于NUMERIC，唯一的差别是不会将"30000.0"这样的文本数据转换为INTEGER存储方式。
NONE	不做任何的转换，直接以该数据所属的数据类型进行存储。

### SQLite 亲和类型(Affinity)及类型名称

下表列出了当创建 SQLite3 表时可使用的各种数据类型名称，同时也显示了相应的亲和类型：

数据类型	亲和类型
INT、INTEGER、TINYINT、SMALLINT、MEDIUMINT、BIGINT、UNSIGNED、BIG、INT、INT2、INT8	INTEGER
CHARACTER(20)、VARCHAR(255)、VARYING、CHARACTER(255)、NCHAR(55)、NATIVE、CHARACTER(70)、NVARCHAR(100)、TEXT、CLOB	TEXT
BLOB、no datatype specified	NONE
REAL、DOUBLE、DOUBLE、PRECISION、FLOAT	REAL
NUMERIC、DECIMAL(10,5)、BOOLEAN、DATE、DATETIME	NUMERIC

NUMERIC、DECIMAL(10,5)、BOOLEAN、DATE、DATETIME

## 6.命令

Insert、Select、Update、Delete、Like、Glob、Distinct、Limit、Having

交叉连接：CROSS JOIN

交叉连接（CROSS JOIN）把第一个表的每一行与第二个表的每一行进行匹配。如果两个输入表分别有  $x$  和  $y$  行，则结果表有  $x*y$  行。由于交叉连接(CROSS JOIN)有可能产生非常大的表，使用时必须谨慎，只在适当的时候使用它们。

```
select emp_id, name, dept from company cross join department;
```

内连接：INNER JOIN 内连接（INNER JOIN）根据连接谓词结合两个表（table1 和 table2）的列值来创建一个新的结果表。查询会把 table1 中的每一行与 table2 中的每一行进行比较，找到所有满足连接谓词的行的匹配对。当满足连接谓词时，A 和 B 行的每个匹配对的列值会合并成一个结果行。

```
select emp_id, name, dept from company inner join department on company.id = department.emp_id;
```

外连接：OUTER JOIN

外连接（OUTER JOIN）是内连接（INNER JOIN）的扩展。虽然 SQL 标准定义了三种类型的外连接：LEFT、RIGHT、FULL，但 SQLite 只支持左外连接（LEFT OUTER JOIN）。

```
select emp_id, name, dept from company left outer join department on company.id = department.emp_id;
```

Unions 子句 SQLite 的 UNION 子句/运算符用于合并两个或多个 SELECT 语句的结果，不返回任何重复的行。

为了使用 UNION，每个 SELECT 被选择的列数必须是相同的，相同数目的列表达式，相同的数据类型，并确保它们有相同的顺序，但它们不必具有相同的长度。

UNION ALL 运算符用于结合两个 SELECT 语句的结果，包括重复行。

适用于 UNION 的规则同样适用于 UNION ALL 运算符。

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT  
      ON COMPANY.ID = DEPARTMENT.EMP_ID  
UNION  
      SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT  
      ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

TRUNCATE TABLE//等同于下面数据 DELETE FROM table\_name; DELETE FROM sqlite\_sequence WHERE name = 'table\_name'; UPDATE sqlite\_sequence SET seq = 0 WHERE name = 'table\_name';

## 7.SQLite 约束

- NOT NULL 约束：确保某列不能有 NULL 值。
- DEFAULT 约束：当某列没有指定值时，为该列提供默认值。
- UNIQUE 约束：确保某列中的所有值是不同的。
- PRIMARY Key 约束：唯一标识数据库表中的各行/记录。
- CHECK 约束：CHECK 约束确保某列中的所有值满足一定条件。

```
CREATE TABLE COMPANY3(  
    ID INT PRIMARY KEY      NOT NULL DEFAULT 0,  
    NAME          TEXT      NOT NULL UNIQUE,  
    AGE           INT       NOT NULL DEFAULT 0,  
    ADDRESS        CHAR(50),  
    SALARY         REAL      CHECK(SALARY > 0)  
);  
  
CREATE TABLE table_name(  
    column1 INTEGER AUTOINCREMENT,  
    column2 datatype,  
    column3 datatype,  
    ....  
    columnN datatype,  
);
```

## 8.索引 (Index)

索引 (Index) 是一种特殊的查找表，数据库搜索引擎用来加快数据检索。简单地说，索引是一个指向表中数据的指针。一个数据库中的索引与一本书后边的索引是非常相似的。

```
CREATE INDEX index_name ON table_name;
```

//单列索引

```
CREATE INDEX index_name ON table_name (column_name);
```

//唯一索引

```
CREATE UNIQUE INDEX index_name on table_name (column_name);
```

//联合索引

```
CREATE INDEX index_name on table_name (column1, column2);
```

//查看表索引

```
.indices COMPANY
```

//删除索引

```
DROP INDEX index_name;
```

什么情况下要避免使用索引?

- 虽然索引的目的在于提高数据库的性能，但这里有几个情况需要避免使用索引。使用索引时，应重新考虑下列准则：
- 索引不应该使用在较小的表上。
- 索引不应该使用在有频繁的大批量的更新或插入操作的表上。
- 索引不应该使用在含有大量的 NULL 值的列上。
- 索引不应该使用在频繁操作的列上。

# 1. 安装mysql基础操作

## 1. 数据库基础操作

```
mysql -u用户名 -p密码 [--tee=d:\mysql.log]      登录mysql客户端
show databases;                                查看所有的数据库
use database;                                 切换到某数据库
show tables;                                  查看某数据库下面的所有表
\s 或 status                                查看数据库的状态
```

## 2. 数据库操作

```
show databases;                      查看数据库
create database 数据库名;          创建数据库
drop database 数据库名;           删除数据库
show create database 数据库名     查看数据库创建情况
show create table 表名             查看表的创建情况
例: create database test default character set utf8;
create database jdbc default charset=utf8;
create database if not exists test default charset utf8 collate utf8_general_ci;
```

## 3. 数据表操作

```
show tables;                      查看数据表
create table 表名(
    int 属性,
    name 属性,
    age 属性
)                                创建数据表
drop table 表名                  删除数据表
rename table old表名 to new表名  修改数据表表名
desc 表名                         查看表结构
forexample:
CREATE TABLE IF NOT EXISTS `member` (
    `id` int(10) unsigned NOT NULL DEFAULT '0' COMMENT '编号',
    `user` varchar(20) NOT NULL,
    `passwd` char(40) NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## 4. 数据库设计

### a) 字段类型

数值: int, float  
字符: varchar, char, text, longtext  
日期: date, datetime, time

### b) 字段属性

unsigned  
zerofill  
auto\_increment  
null  
not null  
default

```
alter table 表名 default character set 字符集          //修改表的默认字符集
alter table 表名 CONVERT TO CHARACTER SET charset_name; //修改表的字符集包括列
```

## 5. 字符集设置

```
my.ini
[mysqld]服务器
character-set-server = utf8                         //服务器、数据库和表字符集
collation-server = utf8_general_ci                  //服务器、数据库和表校验字符集
[mysql]客户端
default-character-set=utf8                          //客户端和连接字符集
```

## 6. 表字段索引

```
desc select * from 表名 where field = ''\G;
创建带索引的表:
create table t1(
    id int unsigned not null auto_increment,
    name varchar(20),
    primary key (id),
    index in_name(name) || key(name)
)engine=InnoDB default character set utf8;

show index from table_name;      //查看索引情况
? [参数]          //查看某帮助信息

普通索引: 1.添加
            alter table 表名 add index in_name(field);
            alter table 表名 add key(field);
        2.删除
            alter table 表名 drop index in_name;
            alter table 表名 drop key(field)
```

## 7. 表字段维护

### 1.添加字段

```
alter table 表名 add field 属性 after field;
```

### 2.修改字段

```
alter table 表名 modify field 属性
```

### 3.删除字段

```
alter table 表名 drop field
```

### 4.修改字段名

```
alter table 表名 change oldfield newfield 属性
```

## 8.mysql操作语句

### 1.concat()

```
for: select id,name,concat(id,'--',name) as newfield from table_name;
```

### 2.rand()

```
for: select id,name,rand()*2 as newfield from table_name;
```

### 3.sum()

### 4.avg()

### 5.max()

### 6.min()

### 7.distinct 去重复

```
select distinct * from table_name;
```

### 7.分组聚合

```
for:
```

```
    a)select name,count(id) as newfield from table_name group by name order by newfield asc;
```

```
    b)select name,count(id) as newfield from table_name group by name having newfield>5 order by newfield asc;
```

### 8.多表查询

#### a)普通多表查询

```
    for: select name,count(name) as num from user as u,post as p where u.id=p.u_id group by u.name;
```

#### b)嵌套查询-多表

```
    for:select name from user where id in (select distinct u_id from post);
```

#### c)左连接查询-多表

```
    for:select a.name,b.title from user as a left join post as b on a.id=b.u_id;
```

```
drop table if exists p_user;
```

```
create table p_user(
```

## 1.5 mysql

```
    id bigint(20) unsigned not null auto_increment primary key comment '编号',
    u_name varchar(20) null comment '姓名',
    u_spell text comment '姓名拼音',
    key u_name(u_name)
)engine=InnoDB default charset=utf8 comment='人员表';
desc p_user;
show create table p_user;

DROP TABLE IF EXISTS `tdl_workshop_daily_two_config`;
CREATE TABLE IF NOT EXISTS `tdl_workshop_daily_two_config`(
    `twt_id` BIGINT(20) UNSIGNED NOT NULL DEFAULT '0' PRIMARY KEY COMMENT '主键',
    `twd_id` BIGINT(20) UNSIGNED NOT NULL DEFAULT '0' COMMENT '主表主键',
    `did` BIGINT(20) UNSIGNED NULL COMMENT '值班地点'
)ENGINE=MYISAM ROW_FORMAT=DYNAMIC DEFAULT CHARSET=utf8 COMMENT='车间日报配置主表-值班地点'
```

mysql更改密码:

1. update mysql.user set password = password('newpasswd') where user = '用户名'; //修改密码
2. flush privileges; //刷新数据库  
或
3. grant select,insert,update,delete,show\_db on 数据库名.\* to 用户名@"主机名" Identified by "密码";

mysqlcheck使用:

1. mysqlcheck -r -u root -p -A 修复所有的表
2. mysqlcheck -r -u root -p db\_name table\_name 修复某数据库的某表

```
CREATE DATABASE db_name
DEFAULT CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

数据库事物:

- 1.原子性: 不可分割。
- 2.一致性: 事物必须使数据库从一个一致性状态变换到另一个一致性状态。
- 3.隔离性: 不被其他事物干扰。
- 4.持久性: 一个事物一旦被提交, 它对数据库数据的改变就是永久性的。

问题:

- 1.脏读:
- 2.不可重复读
- 3.幻读

空着，待写.....

空着，待写.....

空着，待写.....

## 1. PHP实现从1累加到100(1+2+....+100=)的几种思路？

```
function sum1($num)
{
    if ($num > 0) {
        return $num + sum1($num - 1);
    }
}

$a = sum1(100);
echo $a;

function sum2($num)
{
    $sum = 0;
    for ($i = 1; $i <= $num; $i++) {
        $sum += $i;
    }
    return $sum;
}

$a = sum2(100);
echo $a;

function sum3($num)
{
    $sum = 0;
    while ($num > 0) {
        $sum += $num;
        $num--;
    }
    return $sum;
}

$a = sum3(100);
echo $a;

echo array_sum(range(1, 100));
```

## 2. PHP 链式操作

```
class DB
{
    public function __construct()
    {

    }

    public function select()
    {
        return $this;
    }

    public function where()
    {
        return $this;
    }

    public function limit()
    {
        return $this;
    }

    public function orderBy()
    {
        return $this;
    }
}

$db = new DB();

$db->select()->where()->limit()->orderBy();
```

空着，待写.....

#### 1.4.1 PHP的LRU实现

LRU (Least recently used, 最近最少使用) 算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

```
class Node
{
    private $key;
    private $val;
    private $previous;
    private $next;

    public function __construct($key, $val)
    {
        $this->key = $key;
        $this->val = $val;
    }

    public function setVal($val)
    {
        $this->val = $val;
    }

    public function getKey()
    {
        return $this->key;
    }

    public function getVal()
    {
        return $this->val;
    }

    public function setPrevious($previous)
    {
        $this->previous = $previous;
    }

    public function getPrevious()
    {
        return $this->previous;
    }

    public function setNext($next)
    {
        $this->next = $next;
    }

    public function getNext()
    {
```

#### 1.4.1 PHP的LRU实现

```
        return $this->next;
    }

}

class LRU
{
    private $hashmap = array();

    private $head;

    private $tail;

    private $cacheNum = 0;

    public function __construct($cacheNum)
    {
        $this->cacheNum = $cacheNum;

        $this->hashmap = array();

        $this->head = new Node(null, null);

        $this->tail = new Node(null, null);

        $this->head->setNext($this->tail);

        $this->tail->setPrevious($this->head);
    }

    public function get($key)
    {
        if (!isset($this->hashmap[$key])) {
            return null;
        }

        $node = $this->hashmap[$key];

        if (count($this->hashmap) == 1) {
            return $node->getVal();
        }

        //处理链表

        return $node->getVal();
    }

    public function put($key, $val)
    {
        if ($this->cacheNum <= 0) {

```

#### 1.4.1 PHP的LRU实现

```
        return false;
    }

    if (isset($this->hashmap[$key]) && !empty($this->hashmap[$key])) {
        //更新数据
        $node = $this->hashmap[$key];

        $this->detach($node);

        $this->attach($this->head, $node);

        $node->setVal($val);
    } else {
        $node = new Node($key, $val);
        $this->hashmap[$key] = $node;

        //处理数据
        $this->attach($this->head, $node);

        if (count($this->hashmap) > $this->cacheNum) {
            $nodeRemove = $this->tail->getPrevious();
            $this->detach($nodeRemove);

            unset($this->hashmap[$nodeRemove->getKey()]);
        }
    }
    return true;
}

private function attach($head, $node)
{
    $node->setPrevious($head);
    $node->setNext($head->getNext());
    $node->getNext()->setPrevious($node);
    $node->getPrevious()->setNext($node);
}

private function detach($node)
{
    $node->getPrevious()->setNext($node->getNext());
    $node->getNext()->setPrevious($node->getPrevious());
}
```

#### 1.4.1 PHP的LRU实现

```
error_reporting(0);
echo '开始内存: '.round(memory_get_usage()/1024/1024, 2)."\n";

$numEntries = 100000;
$lru        = new LRU($numEntries);

while ($numEntries > 0) {
    $lru->put($numEntries - 99999, 'some value...' . $numEntries);
    $numEntries--;
}

echo '运行后内存: '.round(memory_get_usage()/1024/1024, 2)."\n";
```

## 1. 冒泡排序

冒泡排序算法的原理如下：

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
<?php
function bubbling($arr)
{
    $num = count($arr);
    for ($i=0; $i < $num-1; $i++) {
        for ($j=0; $j < $num-$i-1; $j++) {
            if($arr[$j] > $arr[$j+1])
            {
                $temp = $arr[$j+1];
                $arr[$j+1] = $arr[$j];
                $arr[$j] = $temp;
            }
        }
    }
    return $arr;
}

$arr = [18,29,37,13,9,19];

$res = bubbling($arr);
print_r($res);
?>
```

## 2. 选择排序

选择排序（Selection sort）是一种简单直观的排序算法。

它的工作原理是每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。

以此类推，直到全部待排序的数据元素排完。选择排序是不稳定的排序方法。

```

function select($arr)
{
    $num = count($arr);

    for ($i = 0; $i < $num - 1; $i++) {
        $min = $i;

        for ($j = $i + 1; $j < $num; $j++) {
            if ($arr[$min] > $arr[$j]) {
                $min = $j;
            }
        }

        if ($min != $i) {
            $temp      = $arr[$min];
            $arr[$min] = $arr[$i];
            $arr[$i]   = $temp;
        }
    }

    return $arr;
}

$arr = [18, 29, 37, 13, 9, 19];

$res = select($arr);
print_r($res);

```

### 3. 插入排序

插入排序 (Insertion sort) 是一种简单直观且稳定的排序算法。

如果有一个已经有序的数据序列，要求在这个已经排好的数据序列中插入一个数，但要求插入后此数据序列仍然有序，这个时候就要用到一种新的排序方法——插入排序法。插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为 $O(n^2)$ 。是稳定的排序方法。

## 1.4.2 排序算法

```
function insert($arr)
{
    $num = count($arr);

    for ($i=1; $i < $num; $i++) {
        $temp = $arr[$i];

        $j = $i;

        while ($j > 0 && $arr[$j-1] > $temp) {
            $arr[$j] = $arr[$j-1];
            $j--;
        }

        // for ($j; $j > 0; $j--) {
        //     if ($arr[$j - 1] > $temp) {
        //         $arr[$j] = $arr[$j - 1];
        //     }else{
        //         break;
        //     }
        // }

        $arr[$j] = $temp;
    }
    return $arr;
}

$arr = [18, 29, 37, 13, 9, 19];

$res = insert($arr);
print_r($res);
```

数据结构与算法之PHP实现链表类（单链表/双链表/循环链表）

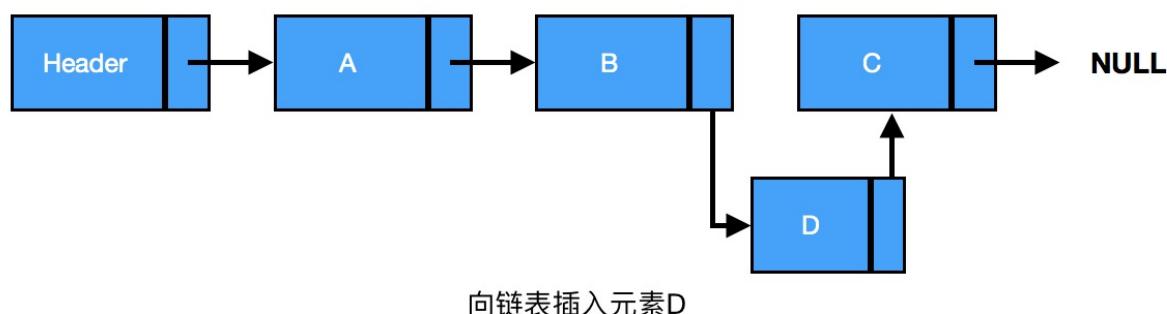
链表是由一组节点组成的集合。每个节点都使用一个对象的引用指向它的后继。指向另一个节点的引用叫做链。

链表分为单链表、双链表、循环链表。

## 1. 单向链表



插入：链表中插入一个节点的效率很高。向链表中插入一个节点，需要修改它前面的节点（前驱），使其指向新加入的节点，而新加入的节点则指向原来前驱指向的节点（见下图）。

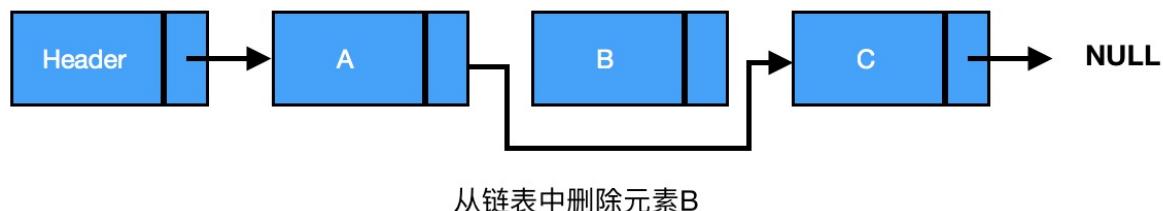


由上图可知，B、C之间插入D，三者之间的关系为

current为插入节点的前驱节点 `current->next = new` // B节点指向新节点D

`new->next = current->next` // 新节点D指向B的后继节点C

删除：从链表中删除一个元素，将待删除元素的前驱节点指向待删除元素的后继节点，同时将待删除元素指向 null，元素就删除成功了（见下图）。



由上图可知，A、C之间删除B，三者之间的关系为

current为要删除节点的前驱节点

`current->next = current->next->next` // A节点指向C节点

### 1.4.3 链表算法

```
<?php

class Node
{
    public $data;

    private $next;

    public function __construct($data)
    {
        $this->data = $data;
    }

    public function setNext($next)
    {
        $this->next = $next;
    }

    public function getNext()
    {
        return $this->next;
    }
}

class Links
{
    private $header;
    public function __construct($data)
    {
        $this->header = new Node($data);
        // $this->header->setNext($this->header); 加上这句话为循环列表
    }

    public function find($data)
    {
        $current = $this->header;
        while ($current->data != $data) {
            $current = $current->getNext();
        }
        return $current;
    }

    public function put($header, $data)
    {
        $node = new Node($data);

        $current = $this->find($header);
        if ($current->data && $current->getNext() != $this->header) {
            $node->setNext($current->getNext());
            $current->setNext($node);
        } else {
            $current->getNext() = $node;
        }
    }
}
```

### 1.4.3 链表算法

```
    $current->setNext($node);
    $node->setNext($this->header);
}

public function del($data)
{
    $current = $this->header;
    while ($current->getNext()->data != $data) {
        $current = $current->getNext();
    }

    if ($current->getNext() != $this->header) {
        // 链表中间
        $current->setNext($current->getNext()->getNext());
    } else {
        // 链表尾端
        $current->setNext($this->header);
    }
}

$ln = new Links('');
// print_r($ln);

$ln->put('', 'two');
// print_r($ln);

$ln->put('two', 'three');
// print_r($ln);

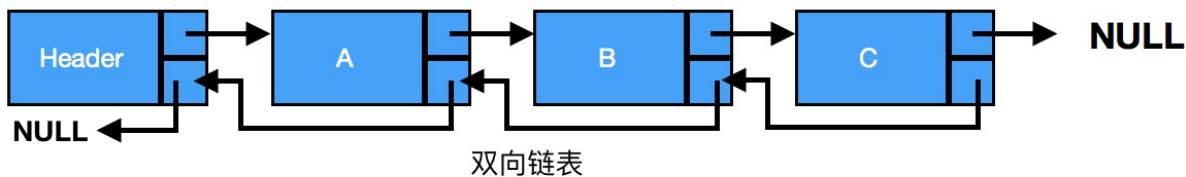
$ln->put('three', 'four');
// print_r($ln);
// exit;

$ln->del('three');
print_r($ln);
```

## 2. 双向链表

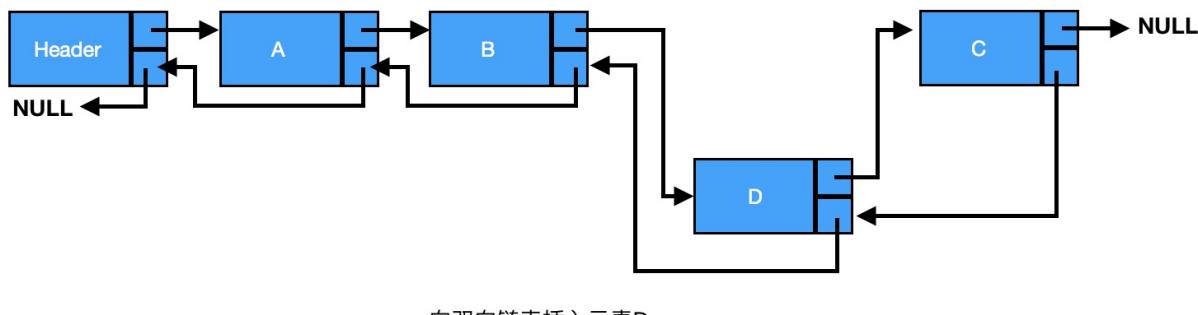
单链表从链表的头节点遍历到尾节点很简单，但从后向前遍历就没那么简单了。它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。

所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。



插入：插入一个节点时，需要指出该节点正确的前驱和后继。

修改待插入节点的前驱节点的next属性，使其指向新加入的节点，而新插入的节点的next属性则指向原来前驱指向的节点，同时将原来前驱指向的节点的previous属性指向新节点，而新加入节点的previous属性指向它前驱节点（见下图）。



由上图可知，B、C之间插入D，三者之间的关系为

current为插入节点的前驱节点

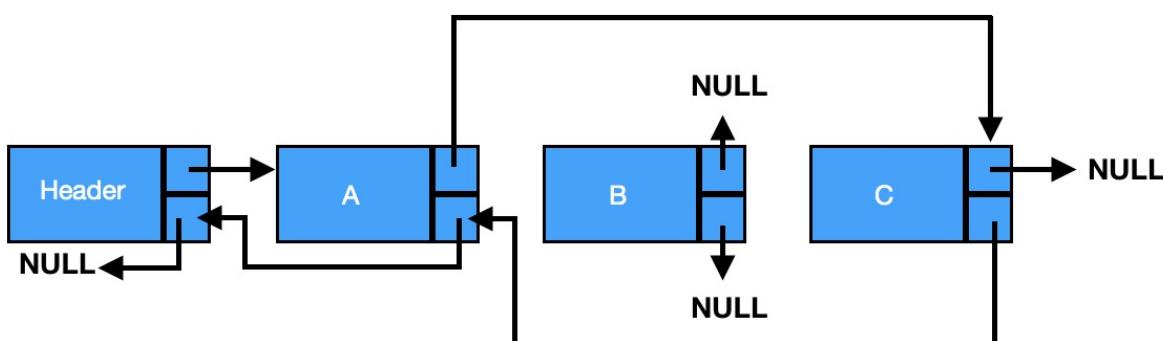
`current->next = new` // B的next属性指向新节点D

`new->next = current->next` // 新节点D的next属性指向B的后继节点C

`current->next->previous = new` // B的后继节点C的previous属性指向新节点D（原先是C的previous属性指向B）

删除：双向链表的删除操作比单向链表的效率更高，因为不需要再查找前驱节点了。

首先需要在链表中找出存储待删除数据的节点，然后设置该节点前驱的 next 属性，使其指向待删除节点的后继;设置该节点后继的 previous 属性，使其指向待删除节点的前驱。



从双向链表中删除元素B

由上图可知，B、C之间删除D，三者之间的关系为

current为要删除的节点

### 1.4.3 链表算法

current->previous->next = current->next // B的前驱节点A的next属性指向B的后继节点C  
current->next->previous = current->previous // B的后继节点C的previous属性指向B的前驱节点A

current->previous = null // B的previous属性指向null  
current->next = null // B的next属性指向null

```
<?php

class Node
{
    public $data;

    private $next;

    public function __construct($data)
    {
        $this->data = $data;
    }

    public function setNext($next)
    {
        $this->next = $next;
    }

    public function getNext()
    {
        return $this->next;
    }
}

class Links
{
    private $header;
    public function __construct($data)
    {
        $this->header = new Node($data);
        $this->header->setNext($this->header);
    }

    public function find($data)
    {
        $current = $this->header;
        while ($current->data != $data) {
            $current = $current->getNext();
        }
        return $current;
    }

    public function put($header, $data)
    {
```

### 1.4.3 链表算法

```
$node = new Node($data);

$current = $this->find($header);
if ($current->data && $current->getNext() != $this->header) {
    $node->setNext($current->getNext());
    $current->setNext($node);
} else {
    $current->setNext($node);
    $node->setNext($this->header);
}

}

public function del($data)
{
    $current = $this->header;
    while ($current->getNext()->data != $data) {
        $current = $current->getNext();
    }

    if ($current->getNext() != $this->header) {
        // 链表中间
        $current->setNext($current->getNext()->getNext());
    } else {
        // 链表尾端
        $current->setNext($this->header);
    }
}

}

$ln = new Links('');
// print_r($ln);

$ln->put('', 'two');
// print_r($ln);

$ln->put('two', 'three');
// print_r($ln);

$ln->put('three', 'four');
// print_r($ln);
// exit;

$ln->del('three');
print_r($ln);
```

#### 1.4.4 算法案例

---

空着，待写.....

### 1.5.1 工厂模式

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

```
<?php
namespace Lib;

class Factory
{

    public static function getDb()
    {
        $db = Register::get('db');
        if(!$db)
        {
            $db = new \Lib\Database\Mysqli();
            Register::set('db',$db);
        }
        return $db;
    }

    public static function getUser($id)
    {
        $key = 'user' . $id;
        $user = Register::get($key);
        if(!$user)
        {
            $user = new \Lib\User($id);
            Register::set($key,$user);
        }
        return $user;
    }
}
```

## 1.5.2 单例模式

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其他对象提供这一实例。

```
<?php

namespace Lib;

class Db
{
    private static $instance;

    private function __construct(){}

    public static function getInstance()
    {
        if(!self::$instance instanceof self)
        {
            self::$instance = new self();
        }

        return self::$instance;
    }
}
```

### 1.5.3 注册树模式

将对象注册到一个类中  
通过该类实现全局访问操作对象

```
<?php

namespace Lib;

class Register
{
    public static $objects;
    public static function set($alias , $obj)
    {
        self::$objects[$alias] = $obj;
    }

    public static function get($alias)
    {
        return isset(self::$objects[$alias] ) ? self::$objects[$alias] : null;
    }

    public static function _unset($alias)
    {
        unset(self::$objects[$alias]);
    }
}
```

```
$user = new \Lib\User($id);
Register::set($key,$user);
```

## 1.5.4 适配器模式

适配器模式:即将截然不同的函数接口封装成统一的接口API

例如 MYSQL的数据库扩展操作 mysql,mysqli,pdo三种,可以用适配器模式统一成一致.

类似的场景还有cache操作,例如 redis,memcached,mongodb,apc等不同的缓存函数,统一成一致

```
<?php

namespace Lib;

interface Database
{
    function connect($host,$name,$pass,$db);
    function query($sql);
    function close();
}
```

```
<?php

namespace Lib\Database;

class Mysql implements \Lib\Database{
    public function connect($host,$name,$pass,$db)
    {

    }
    public function query($sql)
    {

    }

    public function close()
    {

    }
}
```

## 1.5.4 适配器模式

```
<?php

namespace Lib\Database;

class Mysqli implements \Lib\Database{
    private $conn;
    public function connect($host,$user,$pass,$db)
    {
        $this->conn = mysqli_connect($host,$user,$pass,$db);
    }
    public function query($sql)
    {
        return mysqli_query($this->conn ,$sql);
    }

    public function close()
    {
        mysqli_close($this->conn);
    }
}
```

```
$db = new \Lib\Database\Mysqli();
print_r($db);
```

## 1.5.5 策略模式

将一组特定的行为和算法封装成类,以适应某些特定的上下文环境.这种模式就是策略模式.

```
<?php
namespace Lib;

interface UserStrategy{
    function showAd();
    function showCategory();
}
```

```
<?php
//男性策略类
namespace Lib;

class ManStrategy implements \Lib\UserStrategy
{
    public function showAd()
    {
        echo "游戏";
    }

    public function showCategory()
    {
        echo "理财";
    }
}
```

```
<?php
//女性策略类
namespace Lib;

class WomanStrategy implements \Lib\UserStrategy
{
    public function showAd()
    {
        echo "动漫";
    }

    public function showCategory()
    {
        echo "购物";
    }
}
```

## 1.5.5 策略模式

```
<?php
//页面展示类
namespace Lib;

class Page
{
    private $strategy;

    public function setContext(\Lib\UserStrategy $userStrategy)
    {
        $this->strategy = $userStrategy;
    }

    public function action()
    {
        $this->strategy->showCategory();
    }
}
```

```
$page = new \Lib\Page();
//根据条件执行
$page->setContext(new \Lib\WomanStrategy());

$page->action();
```

## 1.5.6 数据对象映射模式

是将对象和数据存储映射起来，对一个对象的操作会映射为对数据存储的操作。例如在代码中 new 一个对象，使用数据对象映射模式就可以将对象的一些操作比如设置一些属性，就会自动保存到数据库，跟数据库中表的一条记录对应起来。

```
<?php

namespace Lib;

class User
{
    public $id;
    public $name;
    public $email;
    public $addtime;

    private $db;

    public function __construct($id)
    {
        $this->db = Factory::getDb();
        $this->db->connect('127.0.0.1','root',123456,'demo');
        $result = $this->db->query("select * from user where id = $id");
        $row = $result->fetch_assoc();
        $this->id = $row['id'];
        $this->name = $row['name'];
        $this->email = $row['email'];
        $this->addtime = $row['addtime'];
    }

    public function __destruct()
    {
        $this->db->query("update user set name = '{$this->name}',email = '{$this->email}' ,addtime = '{$this->addtime}' where id = '{$this->id}'");
    }
}
```

```
public static function getUser($id)
{
    $key = 'user' . $id;
    $user = Register::get($key);
    if (!$user)
    {
        $user = new \Lib\User($id);
        Register::set($key, $user);
    }
    return $user;
}
```

## 1.5.6 数据对象映射模式

---

```
$user = \Lib\Factory::getUser(1);
print_r($user);
```

## 1.5.7 观察者模式

观察者模式(Observer),当一个对象的状态发生改变时,依赖他的对象会全部收到通知,并自动更新。

场景:一个事件发生后,要执行一连串更新操作.传统的编程方式,就是在事件的代码之后直接加入处理逻辑,当更新得逻辑增多之后,代码会变得难以维护.这种方式是耦合的,侵入式的,增加新的逻辑需要改变事件主题的代码

观察者模式实现了低耦合,非侵入式的通知与更新机制

```
<?php
//观察者接口类
namespace Lib;

interface Observer{
    function update();
}
```

```
<?php
//事件产生类
namespace Lib;

class Subject
{
    private $observers;

    public function attach(\Lib\Observer $observer)
    {
        $this->observers[] = $observer;
    }

    public function notify()
    {
        if($this->observers)
        {
            foreach ($this->observers as $k=>$v)
            {
                $v->update();
            }
        }
    }
}
```

## 1.5.7 观察者模式

```
<?php
//事件
namespace Test;

class Event extends \Lib\Subject
{
    public function trigger()
    {
        echo "做了一堆动作，引来某些人";
        $this->notify();
    }
}
```

```
<?php
//观察者1
namespace Test;

class Observer1 implements \Lib\Observer
{
    public function update()
    {
        echo '观察者1';
    }
}
```

```
<?php
//观察者2
namespace Test;

class Observer2 implements \Lib\Observer
{
    public function update()
    {
        echo '观察者2';
    }
}
```

```
$event = new \Test\Event();

=event->attach(new \Test\observer1());
=event->attach(new \Test\observer2());

=event->trigger();
```

## 1.5.8 原型模式

原型模式就是clone就是内存拷贝，比new的好处是创建对象快速，适合大对象创建

1.原型模式与工厂模式作用类似,都是用来创建对象

2.与工厂模式的实现不同,原型模式是先创建好一个原型对象,然后通过clone原型对象来创建新的对象,这

3.原型模式适用于大对象的创建,创建一个大对象需要很大的开销,如果每次new就会消耗很大,原型模式仅需内存拷贝即可

```
<?php

namespace Lib;

interface Prototype
{
    public function shallowCopy();

    public function deepCopy();
}
```

```
<?php

namespace Lib;

class ConcretePrototype implements \Lib\Prototype
{
    public function shallowCopy()
    {
        return clone $this;
    }

    public function deepCopy()
    {
        $serialize_obj = serialize($this);
        return unserialize($serialize_obj);
    }
}
```

## 1.5.8 原型模式

```
$pro = new \Lib\ConcretePrototype();

$pro->name = 'liaozongchao';
$pro->age = 25;

print_r($pro);

$proCopy = $pro->shallowCopy();
$proCopy->age = 26;

print_r($proCopy);
print_r($pro);
```

## 1.5.9 装饰器模式

- 1.装饰器模式（Decorator），可以动态地添加修改类的功能
- 2.一个类提供了一项功能，如果要在修改并添加额外的功能，传统的编程模式，需要写一个子类继承它，并重新实现类的方法
- 3.使用装饰器模式，仅需在运行时添加一个装饰器对象即可实现，可以实现最大的灵活性

```
<?php

namespace Lib;

interface Decorator
{
    public function before();

    public function after();
}
```

## 1.5.9 装饰器模式

```
<?php

namespace Lib;

class Draw
{
    protected $decorator;

    public function pencil()
    {
        $this->beforePencil();

        echo "我是装饰器";

        $this->afterPencil();
    }

    public function addDecorator($decorator)
    {
        $this->decorator[] = $decorator;
    }

    public function beforePencil()
    {
        if($this->decorator)
        {
            foreach ($this->decorator as $decorator)
            {
                $decorator->before();
            }
        }
    }

    public function afterPencil()
    {
        if($this->decorator)
        {
            $tmp = array_reverse($this->decorator);
            foreach ($tmp as $decorator) {
                $decorator->after();
            }
        }
    }
}
```

## 1.5.9 装饰器模式

```
<?php

namespace Lib;

class ColorDecorator implements \Lib\Decorator
{

    public function before()
    {
        // TODO: Implement before() method.

        echo "画上颜色";
    }

    public function after()
    {
        // TODO: Implement after() method.
    }

}
```

```
$draw = new \Lib\Draw();

$colorDecorator = new \Lib\ColorDecorator();

$draw->addDecorator($colorDecorator);
$draw->pencil();
```

## 1.5.10 迭代器模式

迭代器：类继承PHP的Iterator接口，批量操作。

1. 迭代器模式，在不需要了解内部实现的前提下，遍历一个聚合对象的内部元素。
2. 相比传统的编程模式，迭代器模式可以隐藏遍历元素的所需操作。

```
<?php

namespace Lib;

class AllUser implements \Iterator
{
    protected $index = 0;

    private $data;

    public function __construct()
    {
        $db = Factory::getDb();
        $db->connect('127.0.0.1', 'root', 123456, 'demo');
        $result = $db->query("select id from user");
        $this->data = $result->fetch_all(MYSQLI_ASSOC);

        var_dump($this->data);
    }

    public function current(){
        $id = $this->data[$this->index]['id'];
        return Factory::getUser($id);
    }

    public function next(){
        return $this->index++;
    }

    public function key(){
        return $this->index;
    }

    public function valid(){
        return $this->index < count($this->data);
    }

    public function rewind(){
        $this->index = 0;
    }
}
```

## 1.5.10 迭代器模式

---

```
$users = new \Lib\AllUser();

foreach ($users as $user){
    print_r($user);
}
```



### 3. Python篇

---

空着，待写.....

空着，待写.....

空着，待写.....

### 1.3 实战案例

---

空着，待写.....



## 什么是 socket?

socket 的原意是“插座”，在计算机通信领域，socket 被翻译为“套接字”，它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定，一台计算机可以接收其他计算机的数据，也可以向其他计算机发送数据。

socket 的典型应用就是 Web 服务器和浏览器：浏览器获取用户输入的 URL，向服务器发起请求，服务器分析接收到的 URL，将对应的网页内容返回给浏览器，浏览器再经过解析和渲染，就将文字、图片、视频等元素呈现给用户。

### UNIX/Linux 中的 socket 是什么？

在 UNIX/Linux 系统中，为了统一对各种硬件的操作，简化接口，不同的硬件设备也都被看成一个文件。对这些文件的操作，等同于对磁盘上普通文件的操作。

你也许听很多高手说过，UNIX/Linux 中的一切都是文件！那个家伙说的没错。

为了表示和区分已经打开的文件，UNIX/Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。例如：

通常用 0 来表示标准输入文件（stdin），它对应的硬件设备就是键盘；

通常用 1 来表示标准输出文件（stdout），它对应的硬件设备就是显示器。

UNIX/Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。

请注意，网络连接也是一个文件，它也有文件描述符！你必须理解这句话。

我们可以通过 socket() 函数来创建一个网络连接，或者说打开一个网络文件，socket() 的返回值就是文件描述符。有了文件描述符，我们就可以使用普通的文件操作函数来传输数据了，例如：

用 read() 读取从远程计算机传来的数据；

用 write() 向远程计算机写入数据。

你看，只要用 socket() 创建了连接，剩下的就是文件操作了，网络编程原来就是如此简单！

### Window 系统中的 socket 是什么？

Windows 也有类似“文件描述符”的概念，但通常被称为“文件句柄”。因此，本教程如果涉及 Windows 平台将使用“句柄”，如果涉及 Linux 平台则使用“描述符”。

与 UNIX/Linux 不同的是，Windows 会区分 socket 和文件，Windows 就把 socket 当做一个网络连接来对待，因此需要调用专门针对 socket 而设计的数据传输函数，针对普通文

件的输入输出函数就无效了。

这个世界上有很多种套接字（socket），比如 DARPA Internet 地址（Internet 套接字）、本地节点的路径名（Unix 套接字）、CCITT X.25 地址（X.25 套接字）等。但本教程只讲第一种套接字——Internet 套接字，它是最具代表性的，也是最经典最常用的。以后我们提及套接字，指的都是 Internet 套接字。

### 流格式套接字（SOCK\_STREAM）

- SOCK\_STREAM 有以下几个特征：
- 数据在传输过程中不会消失；
- 数据是按照顺序传输的；
- 数据的发送和接收不是同步的（有的教程也称“不存在数据边界”）。

### 数据报格式套接字（SOCK\_DGRAM）

- 可以将 SOCK\_DGRAM 比喻成高速移动的摩托车快递，它有以下特征：
- 强调快速传输而非传输顺序；
- 传输的数据可能丢失也可能损毁；
- 限制每次传输的数据大小；
- 数据的发送和接收是同步的（有的教程也称“存在数据边界”）。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(){
    //创建套接字
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    //将套接字和IP、端口绑定
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr)); //每个字节都用0填充
    serv_addr.sin_family = AF_INET; //使用IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); //具体的IP地址
    serv_addr.sin_port = htons(1234); //端口
    bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    //进入监听状态，等待用户发起请求
    listen(serv_sock, 20);
    //接收客户端请求
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_size = sizeof(clnt_addr);
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
    //向客户端发送数据
    char str[] = "http://c.biancheng.net/socket/";
    write(clnt_sock, str, sizeof(str));

    //关闭套接字
    close(clnt_sock);
    close(serv_sock);
    return 0;
}
```

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
int main(){
    //创建套接字
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    //向服务器（特定的IP和端口）发起请求
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr)); //每个字节都用0填充
    serv_addr.sin_family = AF_INET; //使用IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); //具体的IP地址
    serv_addr.sin_port = htons(1234); //端口
    connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    //读取服务器传回的数据
    char buffer[40];
    read(sock, buffer, sizeof(buffer)-1);

    printf("Message from server: %s\n", buffer);

    //关闭套接字
    close(sock);
    return 0;
}

```

socket缓冲区 每个 socket 被创建后，都会分配两个缓冲区，输入缓冲区和输出缓冲区。

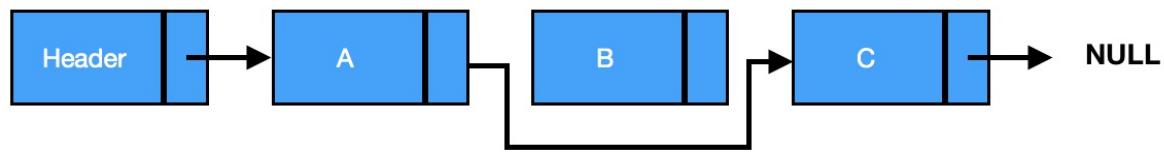
write()/send() 并不立即向网络中传输数据，而是先将数据写入缓冲区中，再由TCP协议将数据从缓冲区发送到目标机器。一旦将数据写入到缓冲区，函数就可以成功返回，不管它们有没有到达目标机器，也不管它们何时被发送到网络，这些都是TCP协议负责的事情。

TCP协议独立于 write()/send() 函数，数据有可能刚被写入缓冲区就发送到网络，也可能在缓冲区中不断积压，多次写入的数据被一次性发送到网络，这取决于当时的网络情况、当前线程是否空闲等诸多因素，不由程序员控制。

read()/recv() 函数也是如此，也从输入缓冲区中读取数据，而不是直接从网络中读取。

TCP套接字的I/O缓冲区示意图 图：TCP套接字的I/O缓冲区示意图

这些I/O缓冲区特性可整理如下： I/O缓冲区在每个TCP套接字中单独存在； I/O缓冲区在创建套接字时自动生成； 即使关闭套接字也会继续传送输出缓冲区中遗留的数据； 关闭套接字将丢失输入缓冲区中的数据。



从链表中删除元素B

# 有些情，只能止于唇齿，掩于岁月

卷首引：从此天涯陌路，天各一方。从此互不来往，斩断情愫。删除了记忆，清空了珍藏。不再徘徊在彷徨的路上。不再泪水央央度过夜的漫长。崭新的一天湛蓝，轻装上阵的舒畅。



风送来你的剪影，搅乱了愁怀萦肠，望着你离开时走的那条路，蹲下身子，久久的看着，看着，直到细雨飘摇，直到天幕落下，直到无声的哽咽。

不知道这世上，还有多少人，能与你和我一般，初遇时便禁不住心动，仿佛认识了很久，我们是那么的相似，连说话的语气，处事的方式也是这么的相像，好像是遇见了另一个自己，明明还是那么陌生，却熟悉的像是老朋友。

越相似，越了解，也越明白对方，似乎能将彼此看透，知道你的心中带着的伤，想要走近，想要安抚，想要带着清风明月，为你黯然的生活，带去新的面貌。

你也知道我揣着的远方和过往，和未来的路将会走向何方，却只愿面对现在，绝口不提以后，也似乎还沉浸在，那些本该尘封的旧年月里。

明晰了彼此内心深处的所想，懂得了各自的固执和坚持，即便有些话不说出口，也知道分开的时候到了。我们之间是这样的了解，连告别也不再需要，只需要默契的不再联系，不再约定晨昏之时相伴，不再述说这一天，这许多天发生过什么事，看过什么值得一笑的话。

如果还有什么迷惑的话，那便是我不知道，我们之间算是什么，是爱，是吸引，或者只是有缘相伴一程，想了很久，也想不出这究竟是怎样的一种情感，姑且称它为情。

一日又一日过去，拉开了当初的距离，从熟悉到更熟悉，从更熟悉再到陌生，原来忘却只需要不再提起，人海之中的似曾相识，是一切开始的源头。似曾相识的源头，是有着相同的过去，有着同样不能忘怀的人，执拗的不肯放下，我们之间一步之遥的距离，便成了天涯海角的隔阂。

我们这样的一段情，只能止于唇齿，你这样的一个人，只能掩于岁月。不知道在失去的时候，那是一种什么样的感觉，有些酸涩，有些不甘，有些遗憾，却又深知，这一切是必然。遇见后相互牵引，是必然，相互深深的懂得，是必然，懂得之后放手，也是必然。

曾为这样的人，写过这样的一首《尘心》：情人滴泪夜娑婆，万花盛放饰人间，一树一叶尽尘缘，满目空待山河远。从相遇到相离，万花盛放到秋叶凋零，所有的期待成空，所有的向往变成了尘缘里的妄想。

于是慢慢的忘却，没有错，是忘却，将他丢在风里，斩断了情丝，重新过回一个人的日子。因为找到一个彼此懂得的人，是曾经最为的期待的爱情，遇到了这样的人，却只能分离，便需要忘记，让爱消失在记忆消失前。

止于唇齿，掩于岁月。

