Memcached分布式缓存系统

==========

Memcached介绍

什么是Memcached缓存数据库

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

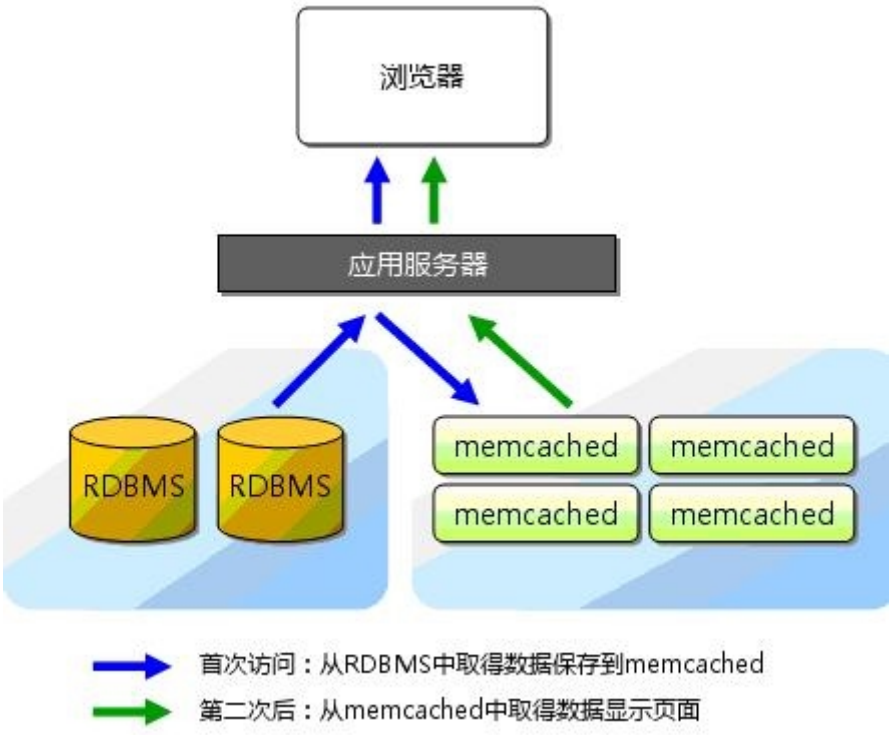
Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于**内存的key-value存储**，用来存储**小**块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。



Memcached 官网：https://memcached.org/。

Memcached和Redis之间的区别

我们都知道，把一些热数据存到缓存中可以极大的提高速度，那么问题来了，是用Redis好还是Memcached好呢，以下是它们两者之间一些简单的区别与比较：

1. Redis不仅支持简单的k/v类型的数据，同时还支持list、set、zset(sorted set)、hash等**丰富数据结构**的存储，使得它拥有更广阔的应用场景。
2. Redis最大的亮点是支持**数据持久化**，它在运行的时候可以将数据备份在磁盘中，断电或重启后，缓存数据可以再次加载到内存中，只要Redis配置的合理，基本上不会丢失数据。
3. Redis支持**主从模式**的应用。
4. Redis单个value的最大限制是1GB，而Memcached则只能保存**1MB**内的数据。
5. Memcache在并发场景下，能用cas保证一致性，而Redis事务支持比较弱，只能保证事务中的每个操作连续执行。
6. 性能方面，根据网友提供的测试，**Redis**在读操作和写操作上是略**领先**Memcached的。
7. Memcached的内存管理不像Redis那么复杂，元数据metadata更小，相对来说额外**开销就很少**。Memcached唯一支持的数据类型是字符串string，非常适合缓存只读数据，因为字符串不需要额外的处理。

快速开始

- 安装软件

```
# 下载相关依赖软件包
[root@server1 ~]# yum install libevent libevent-devel -y
# 下载memcached
[root@server1 ~]# yum install memcached -y
```

- 使用memcached命令管理服务，相关参数介绍
  - -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。
- 启动memcached服务

```
[root@server1 ~]# memcached  -d -m 1024 -u memcached -l 127.0.0.1 -p 11211 -c 1024 -P /tmp/memcached.pid
```

- 查找memcached的进程

```
[root@server1 ~]# ps -ef|grep memcached | grep -v grep
memcach+   1494      1  0 14:50 ?        00:00:00 memcached -d -m 1024 -u memcached -l 127.0.0.1 -p 11211 -c 1024 -P /tmp/memcached.pid
```

- 查看端口号

```
[root@server1 ~]# ss -tuan | grep 11211
udp    UNCONN     0      0      127.0.0.1:11211                 *:*

tcp    LISTEN     0      128    127.0.0.1:11211                 *:*
```

- memcached 连接

```
[root@server1 ~]# yum install telnet -y
```

```
[root@server1 ~]# telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
set name 0 0 8                          保存命令
zhangsan                                数据
STORED
get name                                查询数据
VALUE foo 0 8
zhangsan
END
quit                                    退出
Connection closed by foreign host.

```

- 关闭服务

```
[root@server1 ~]# pkill memcached
[root@server1 ~]# ss -tuan | grep 11211
[root@server1 ~]#
```

slab存储机制

参考博客:

https://www.jianshu.com/p/2ec61d727c4d

相关概念

item 数据存储节点

item数据存储节点主要用于存储数据

- 源码

```
typedef struct _stritem {
    /* Protected by LRU locks */
    //一个item的地址, 主要用于LRU链和freelist链
    struct _stritem *next;
    //下一个item的地址,主要用于LRU链和freelist链
    struct _stritem *prev;

    /* Rest are protected by an item lock */
    //用于记录哈希表槽中下一个item节点的地址
    struct _stritem *h_next;    /* hash chain next */
    //最近访问时间
    rel_time_t      time;       /* least recent access */
    //缓存过期时间
    rel_time_t      exptime;    /* expire time */
    int             nbytes;     /* size of data */
    //当前item被引用的次数, 用于判断item是否被其它的线程在操作中
    //refcount == 1的情况下该节点才可以被删除
    unsigned short  refcount;
    uint8_t         nsuffix;    /* length of flags-and-length string */
    uint8_t         it_flags;   /* ITEM_* above */
    //记录该item节点位于哪个slabclass_t中
```

```
    uint8_t         slabs_clsid;/* which slab class we're in */
    uint8_t         nkey;       /* key length, w/terminating null and
padding */
    /* this odd type prevents type-punning issues when we do
     * the little shuffle to save space when not using CAS. */
    union {
        uint64_t cas;
        char end;
    } data[];
    /* if it_flags & ITEM_CAS we have 8 bytes CAS */
    /* then null-terminated key */
    /* then " flags length\r\n" (no terminating null) */
    /* then data with terminating \r\n (no terminating null; it's binary!)
*/
} item;
```

slab与trunk

slab是一块内存空间，默认大小为1M，memcached会把一个slab分割成一个个chunk, 这些被切割的小的内存块，主要用来存储item

slabclass

每个item的大小都可能不一样，item存储于chunk,如果chunk大小不够，则不足以分配给item使用，如果chunk过大，则太过于浪费内存空间。因此memcached采取的做法是，将slab切割成不同大小的chunk，这样就满足了不同大小item的存储。被划分不同大小chunk的slab的内存在memcached就是用slabclass这个结构体来表现的

- slabclass结构体源码

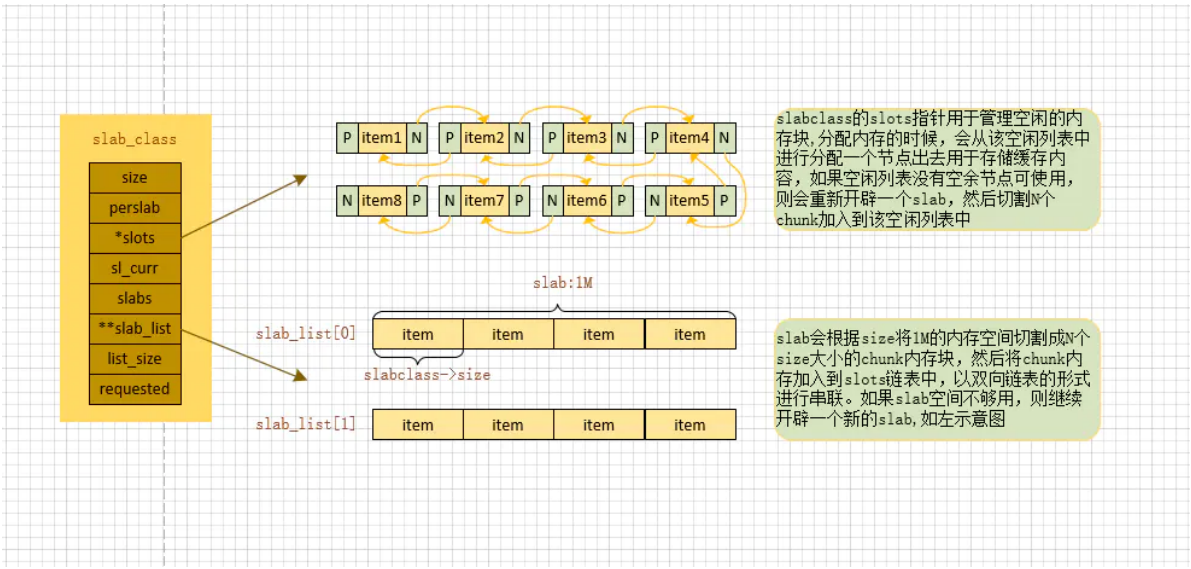
```
typedef struct {
    //chunk大小
    unsigned int size;      /* sizes of items */
    //1M内存大小被分割为多少个chunck
    unsigned int perslab;   /* how many items per slab */

    //空闲chunk链表
    void *slots;           /* list of item ptrs */
    //空闲chunk的个数
    unsigned int sl_curr;   /* total free items in list */

    //当前slabclass已经分配了所少个1M空间的slab
    unsigned int slabs;     /* how many slabs were allocated for this class
*/

    //slab指针数组
    void **slab_list;       /* array of slab pointers */
    //slab指针数组的大小
    unsigned int list_size; /* size of prev array */

    size_t requested; /* The number of requested bytes */
} slabclass_t;
```



1.slabclass数组初始化的时候，每个slabclass_t都会分配一个1M大小的slab，slab会被切分为N个小的内存块，这个小的内存块的大小取决于slabclass_t结构上的size的大小
2.每个slabclass_t都只存储一定大小范围的数据，并且下一个slabclass切割的chunk块大于前一个slabclass切割的chunk块大小
3.memcached中slabclass数组默认大小为64，slabclass切割块大小的增长因子默认是1.25
例如：slabclass[1]切割的chunk块大小为100字节，slabclass[2]为125，如果需要存储一个110字节的缓存，那么就需要到slabclass[2] 的空闲链表中获取一个空闲节点进行存储

item节点分配流程

1、 根据大小，找到合适的slabclass
2、 slabclass空闲列表中是否有空闲的item节点，如果有直接分配item，用于存储内容
3、 空闲列表没有空闲的item可以分配，会重新开辟一个slab(默认大小为1M)的内存块，然后切割slab并放入到空闲列表中，然后从空闲列表中获取节点

item节点释放

释放一个item节点，并不会free内存空间，而是将item节点归还到slabclass的空闲列表中

memcached存储命令

set命令

- 语法

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

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（可以理解为序号）。
- **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
- **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）

示例

```
set name 0 900 8
zhangsan
STORED
get name
VALUE name 0 8
zhangsan
END
```

add命令

Memcached add 命令用于将 **value(数据值)** 存储在不存在的 **key(键)** 中。

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
```

replace命令

Memcached replace 命令用于替换已存在的 **key(键)** 的 **value(数据值)**。

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
```

append命令

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
```

示例

```
set key1 0 900 9
memcached
STORED
get key1
VALUE key1 0 9
memcached
END
append key1 0 900 5
redis
STORED
get key1
VALUE key1 0 14
memcachedredis
END
```

prepend命令

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
```

示例

```
set key1 0 900 9
memcached
STORED
get key1
VALUE key1 0 9
memcached
END
prepend key1 0 900 5
redis
STORED
get key1
VALUE key1 0 14
redismemcached
END
```

cas命令

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。

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

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）

示例

```
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
```

输出信息说明：

- **STORED**：保存成功后输出。
- **ERROR**：保存出错或语法错误。
- **EXISTS**：在最后一次取值后另外一个用户也在更新该数据。
- **NOT_FOUND**：Memcached 服务上不存在该键值。

memcached查找命令

get命令

get 命令的基本语法格式如下：

```
get key
```

多个 key 使用空格隔开，如下：

```
get key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

gets命令

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。

语法

gets 命令的基本语法格式如下：

```
gets key
```

多个 key 使用空格隔开，如下：

```
gets key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

delete命令

Memcached delete 命令用于删除已存在的 key(键)。

语法

delete 命令的基本语法格式如下：

```
delete key [noreply]
```

参数说明如下：

- **key：**键值 key-value 结构中的 key，用于查找缓存值。
- **noreply（可选）**： 该参数告知服务器不需要返回数据

输出信息说明：

- **DELETED**：删除成功。
- **ERROR**：语法错误或删除失败。
- **NOT_FOUND**：key 不存在。

incr与decr命令

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 命令的基本语法格式如下：

```
incr key increment_value
```

参数说明如下：

- **key：**键值 key-value 结构中的 key，用于查找缓存值。
- **increment_value**： 增加的数值。

decr 命令的基本语法格式如下：

```
decr key decrement_value
```

参数说明如下：

- **key：**键值 key-value 结构中的 key，用于查找缓存值。
- **decrement_value**： 减少的数值。

memcached统计命令

stat命令

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
```

```
stats
STAT pid 1162
STAT uptime 5022
STAT time 1415208270
STAT version 1.4.14
STAT libevent 2.0.19-stable
STAT pointer_size 64
STAT rusage_user 0.096006
STAT rusage_system 0.152009
STAT curr_connections 5
STAT total_connections 6
STAT connection_structures 6
STAT reserved_fds 20
STAT cmd_get 6
STAT cmd_set 4
STAT cmd_flush 0
STAT cmd_touch 0
STAT get_hits 4
STAT get_misses 2
STAT delete_misses 1
STAT delete_hits 1
STAT incr_misses 2
STAT incr_hits 1
STAT decr_misses 0
STAT decr_hits 1
STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT touch_hits 0
STAT touch_misses 0
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 262
STAT bytes_written 313
STAT limit_maxbytes 67108864
STAT accepting_conns 1
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT hash_power_level 16
STAT hash_bytes 524288
STAT hash_is_expanding 0
STAT expired_unfetched 1
STAT evicted_unfetched 0
STAT bytes 142
STAT curr_items 2
STAT total_items 6
STAT evictions 0
STAT reclaimed 1
END
```

这里显示了很多状态信息，下边详细解释每个状态项：

- **pid**： memcache服务器进程ID
- **uptime**：服务器已运行秒数
- **time**：服务器当前Unix时间戳
- **version**：memcache版本

- **pointer_size**：操作系统指针大小
- **rusage_user**：进程累计用户时间
- **rusage_system**：进程累计系统时间
- **curr_connections**：当前连接数量
- **total_connections**：Memcached运行以来连接总数
- **connection_structures**：Memcached分配的连接结构数量
- **cmd_get**：get命令请求次数
- **cmd_set**：set命令请求次数
- **cmd_flush**：flush命令请求次数
- **get_hits**：get命令命中次数
- **get_misses**：get命令未命中次数
- **delete_misses**：delete命令未命中次数
- **delete_hits**：delete命令命中次数
- **incr_misses**：incr命令未命中次数
- **incr_hits**：incr命令命中次数
- **decr_misses**：decr命令未命中次数
- **decr_hits**：decr命令命中次数
- **cas_misses**：cas命令未命中次数
- **cas_hits**：cas命令命中次数
- **cas_badval**：使用擦拭次数
- **auth_cmds**：认证命令处理的次数
- **auth_errors**：认证失败数目
- **bytes_read**：读取总字节数
- **bytes_written**：发送总字节数
- **limit_maxbytes**：分配的内存总大小（字节）
- **accepting_conns**：服务器是否达到过最大连接（0/1）
- **listen_disabled_num**：失效的监听数
- **threads**：当前线程数
- **conn_yields**：连接操作主动放弃数目
- **bytes**：当前存储占用的字节数
- **curr_items**：当前存储的数据总数
- **total_items**：启动以来存储的数据总数
- **evictions**：LRU释放的对象数目
- **reclaimed**：已过期的数据条目来存储新数据的数目

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)。

语法

```
stats items
```

示例

```
stats items
STAT items:1:number 1
STAT items:1:age 7
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 0
STAT items:1:expired_unfetched 0
STAT items:1:evicted_unfetched 0
END
```

stats slab

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
```

示例

```
stats slabs
STAT 1:chunk_size 96
STAT 1:chunks_per_page 10922
STAT 1:total_pages 1
STAT 1:total_chunks 10922
STAT 1:used_chunks 1
STAT 1:free_chunks 10921
STAT 1:free_chunks_end 0
STAT 1:mem_requested 71
STAT 1:get_hits 0
STAT 1:cmd_set 1
STAT 1:delete_hits 0
STAT 1:incr_hits 0
STAT 1:decr_hits 0
STAT 1:cas_hits 0
STAT 1:cas_badval 0
STAT 1:touch_hits 0
STAT active_slabs 1
STAT total_malloced 1048512
END
```

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。

语法：stats sizes 命令的基本语法格式如下：

```
stats sizes
```

实例

```
stats sizes
STAT 96 1
END
```

flush_all命令

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。

语法：

flush_all 命令的基本语法格式如下：

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

实例

清理缓存:

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
flush_all
OK
get runoob
END
```