Memcached
源码剖析笔记

Xguru

Memcached 是一个自由、源码开放、高性能、分布式内存对象缓存系统，目的在于通过减轻数据库负载来使动态 Web 应用程序提速。

目录

1.背景

Memcached 是一个高性能的分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提供动态、数据库驱动网站的速度。Memcached 基于一个存储键/值对的 hashmap。

Memcached 是一个自由、源码开放、高性能、分布式内存对象缓存系统，目的在于通过减轻数据库负载来使动态 Web 应用程序提速。

Memcached 是一个在内存中对任意的数据（比如字符串，对象等）所使用的 key-value 存储。数据可以来自数据库调用，API 调用，或者页面渲染的结果。

Memcached 设计理念就是小而强大，它简单的设计促进了快速部署、易于开发，并解决面对大规模的数据缓存的许多难题。所开放的 API 能用于大部分流行的程序语言

2. memcached 的安装

由于 memcached 采用 libevent 的事件处理机制，因此安装 memcached 之前需要先安装 libevent。

Memcached: http://memcached.org/

Libevent : http://www.monkey.org/~provos/libevent/

在 Ubuntu 下可以使用 sudo apt-get install libevent 和 sudo apt-get install memcached 来安装

或者使用传统 wget 的方式

~$ wget http://memcached.googlecode.com/files/memcached-1.2.8.tar.gz.

tar.gz

~$ tar zxf memcached1.2.8.

tar.gz

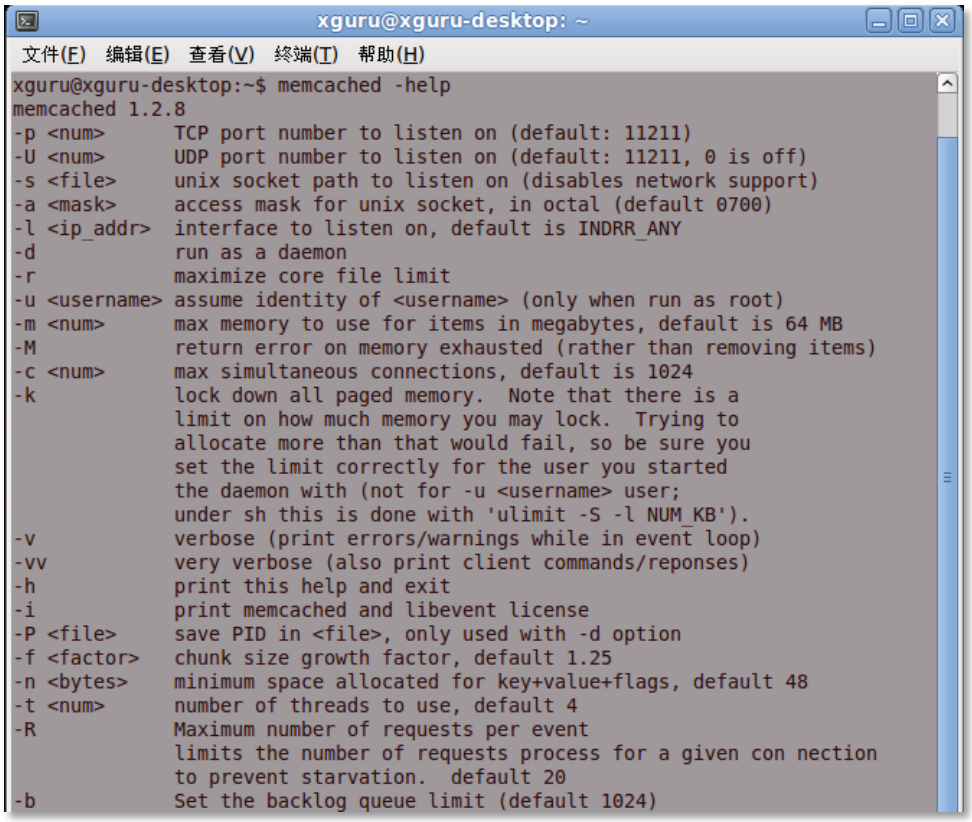
~$ cd memcached1.2.8

~$ ./configure

~$ make

目前最新的版本为 1.4.4

3. memcached 的配置



主要使用的命令

-d 以守护程序（daemon）方式运行 memcached；

-m 设置 memcached 可以使用的内存大小，单位为 M；

-l 设置监听的 IP 地址，如果是本机的话，通常可以不设置此参数；

-p 设置监听的端口，默认为 11211，所以也可以不设置此参数；

-u 指定用户，如果当前为 root 的话，需要使用此参数指定用户。

-f 设置增长因子（调优时使用）

-v/-vv 详细显示工作时各种参数

Memcached 采用典型的 getopt()函数获取各种配置

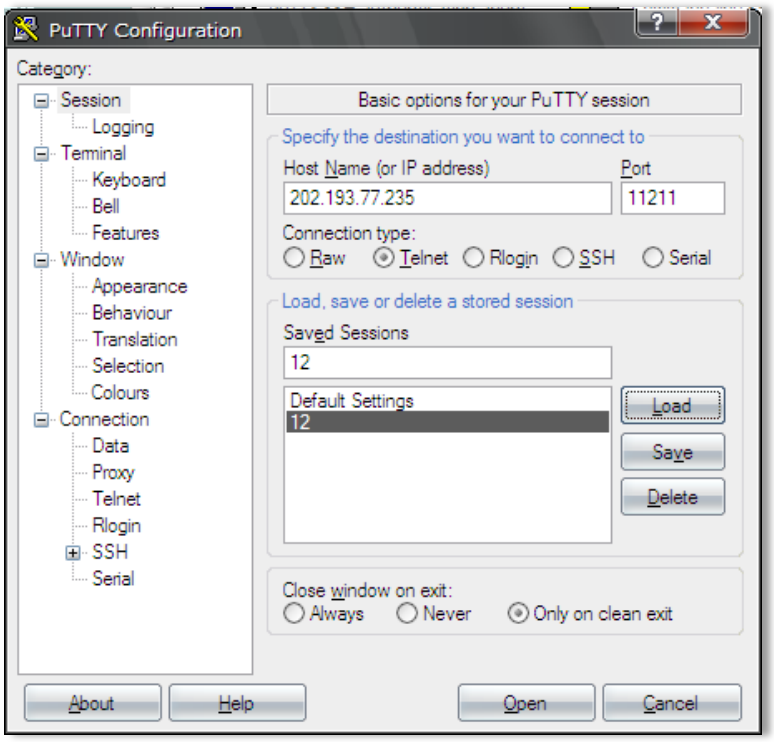
比如

./memcached -m 512 -p 11211 -vv

该例分配给 memcached 的可用内存 512M，监听 11211 端口，显示详细的运行信息。

4. memcached 的使用

memcached 提供的 API 可以在大多数的编程语言使用，在这里测试使用的是 PuTTy 的 telnet 方式，使用 telnet 连接其 11211 端口。



Memcached 有 4 种类型的命令：

- **存储命令**（set/add /replace/append/prepend）指示服务器储存一些由键值标识的数据。客户端发送一行命令，后面跟着数据区块；然后，客户端等待接收服务器回传的命令行，指示成功与否。
- **读取命令**（get/bget/gets）指示服务器返回与所给键值相符合的数据（一个请求中右一个或多个键值）。客户端发送一行命令，包括所有请求的键值；服务器每找到一项内

容，都会发送回客户端一行关于这项内容的信息，紧跟着是对应的数据区块；直到服务器以一行“END”回应命令结束。

- **状态命令**（stat）被用于查询服务器的运行状态和其他内部数据。
- **其他命令**，如 flush_all,version,quit 等。

4.1. 存储命令

命令格式：

<command name> <key> <flags> <exptime> <bytes>

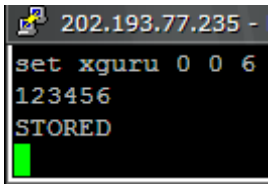
<data block>

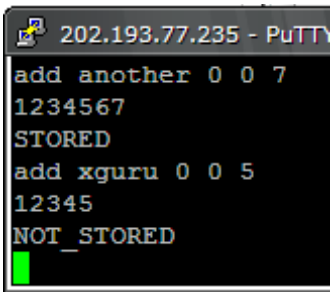
命令解释：

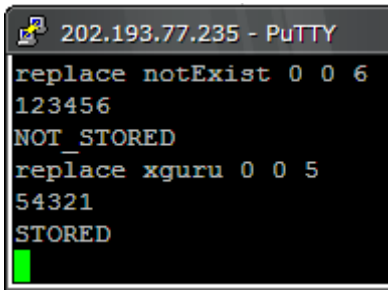
<command name>	set/add/replace
<key>	查找关键字
<flags>	客户机使用它存储关于键值对的额外信息
<exptime>	该数据的存活时间，0 为永远
<bytes>	存储字节数
<data block>	存储的数据块

存储命令区别

set	add	repalce
无论如何都进行存储	只有数据**不存在**时进行添加	只有数据**存在**时进行替换



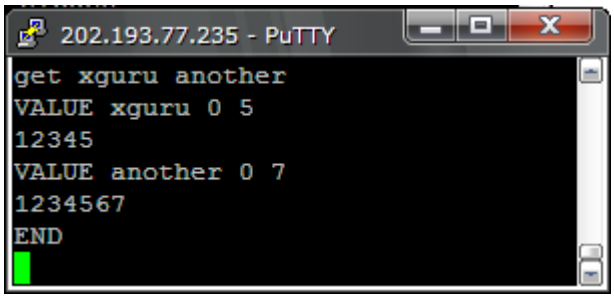




4.2. 读取命令

get <key>

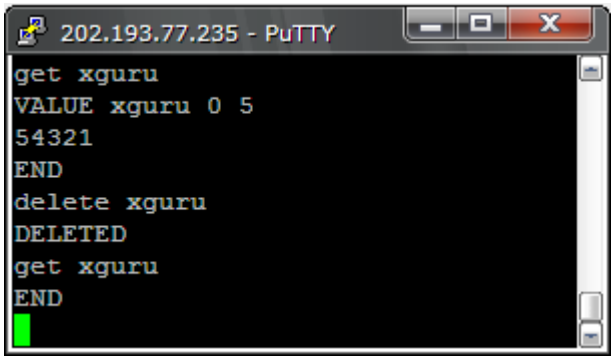
<key>可以表示一个或多个键值，由空格隔开的字串



4.3. 删除命令

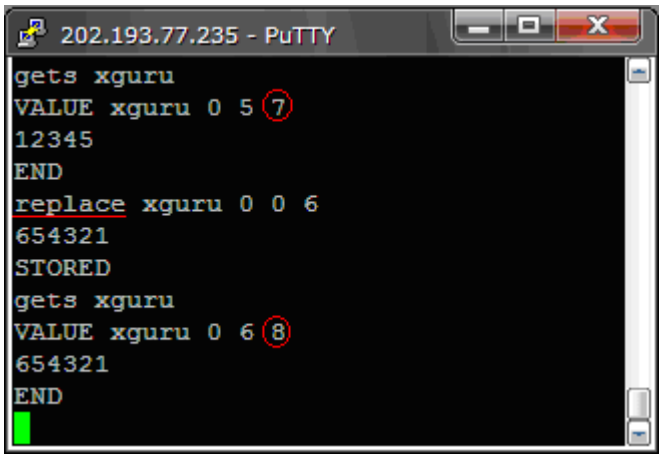
delete <key>

删除键值为 key 的数据。

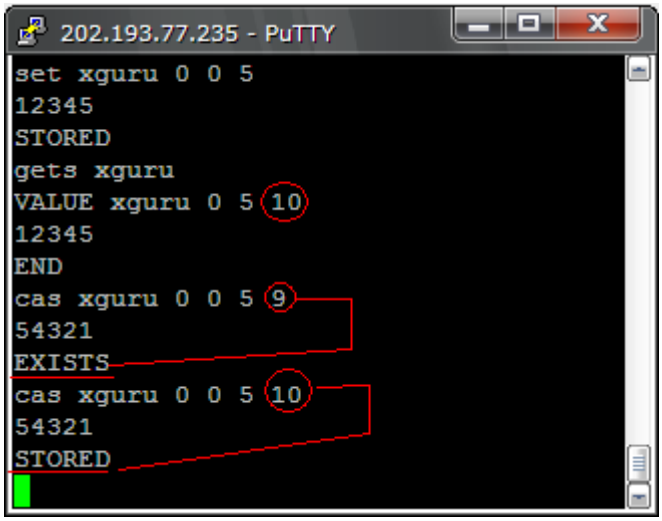


4.4. 高级命令

值得一提的是，新版本的 memcached 加入了 gets 和 cas 命令



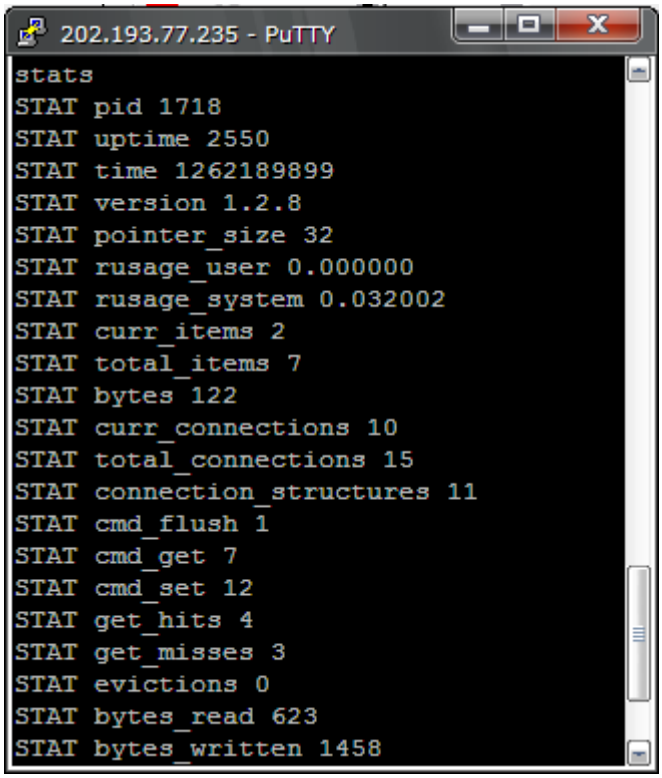
可以看到此处 gets 比普通的 get 多返回一个数字，这个数字可以用作检查数据是否发生改变。当 key 对应的数据改变的时候，该数会发生改变,如图中红圈所示。



cas 就是 check and set 之意，只有当最后一个参数与 gets 所获取的参数匹配时才能存储，否则返回“EXISTS”。这种设计的意图是防止使用经过改变了的 value/key 对。

4.5. 其他命令

查看状态



其他更多的命令可以参看 memcached 的协议：http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt。

了解了其基本工作方式以后再来看源代码发现清晰很多。

5. Memcached 内部工作机制

5.1. Memcached 基本的数据结构

Item(_stritem)结构示意图
_stritem *next; 指向链表下一个item的指针
_stritem *prev 指向链表上一个item的指针
*h_nexthash 指向hash表该桶(Bucket)的下一项
rel_time_t time 最近访问时间
rel_time_t exptime 消亡时间
int nbytes 数据大小
unsigned short refcount 引用计数
void * end[] 存放的数据
uint8_t nsuffix,it_flags,slabs_clsid,nkey 杂项

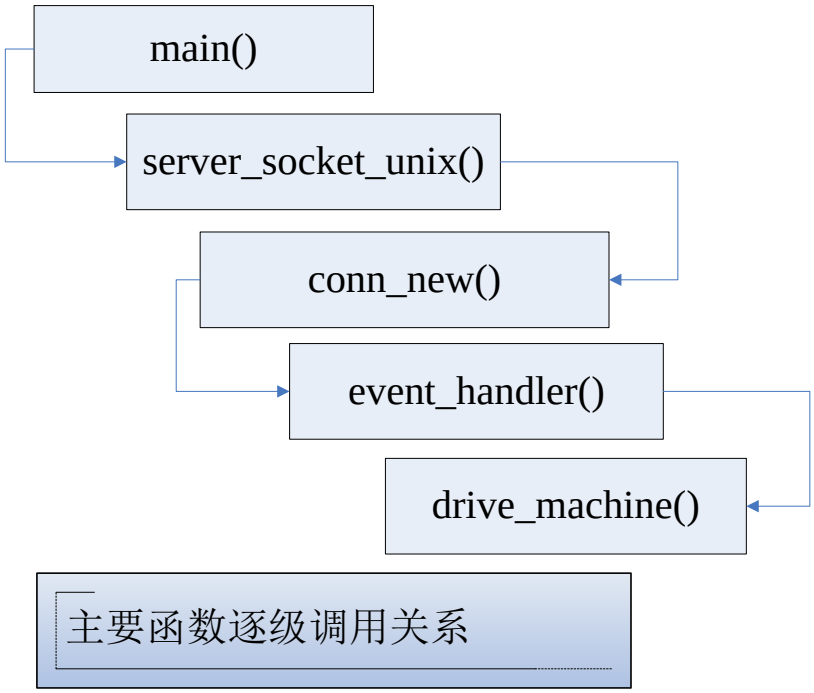
Slab(slabclass_t)结构示意图
unsigned int size 每个chunk的大小
unsigned int perslab 能存放size大小chunk的个数
void **slots 目前空闲可插入item的插糟
unsigned int sl_totalslots 插糟的总容量
unsigned int sl_curr 当前可用的插糟
void *end_page_ptr 最后的slab空闲内存起始地址
unsigned int slabs Slab数
void **slab_list; 存储slab指针的数组

item 为 memcached 中的存储数据最小单位，其中还记录有数据和最近访问时间数据大小，桶的下一项等数据，每个不同 slab 的元素内含有具有统一分配的尺寸的各个 item。可以这样理解，每个 item 是存储在其对应大小的 slabclass_t 里的,同时又在 hash 表中有记录。既可以使用自己的内存分配机制来减少操作系统在处理内存碎片，添加释放等多余的操作，又可以使用 hash 表的性质对其进行快速的定位。

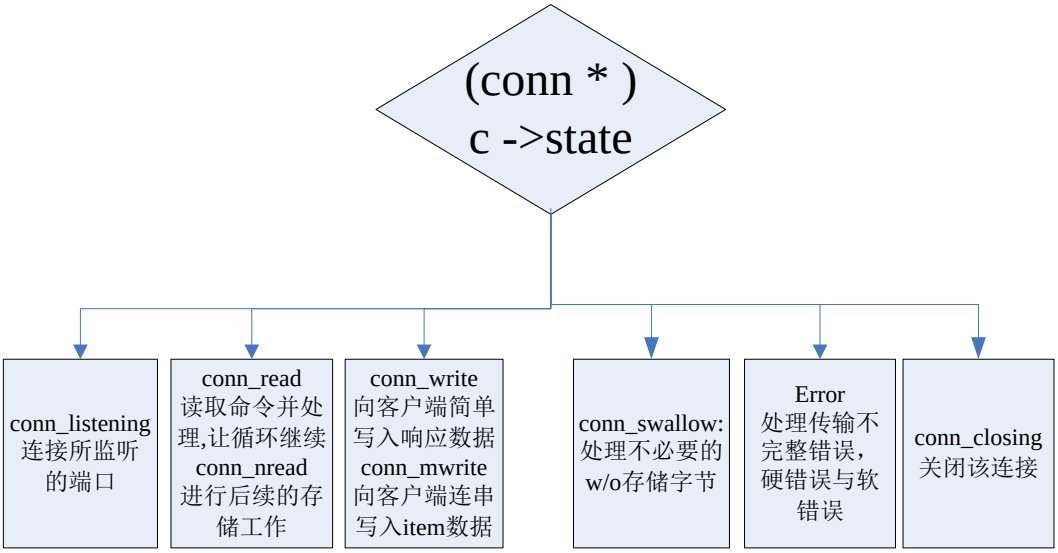
slabclass 是由(POWER_LARGEST + 1)个 slabclass_t 结构体构成的数组，每个 slabclass_t 结构体的大小是根据增长因子递增的，增长因子可以由客户端来设定，1.28 版本的默认值为 2，合理的调优增长因子可以避免空间的浪费。

5.2. 基本设计概念和处理流程

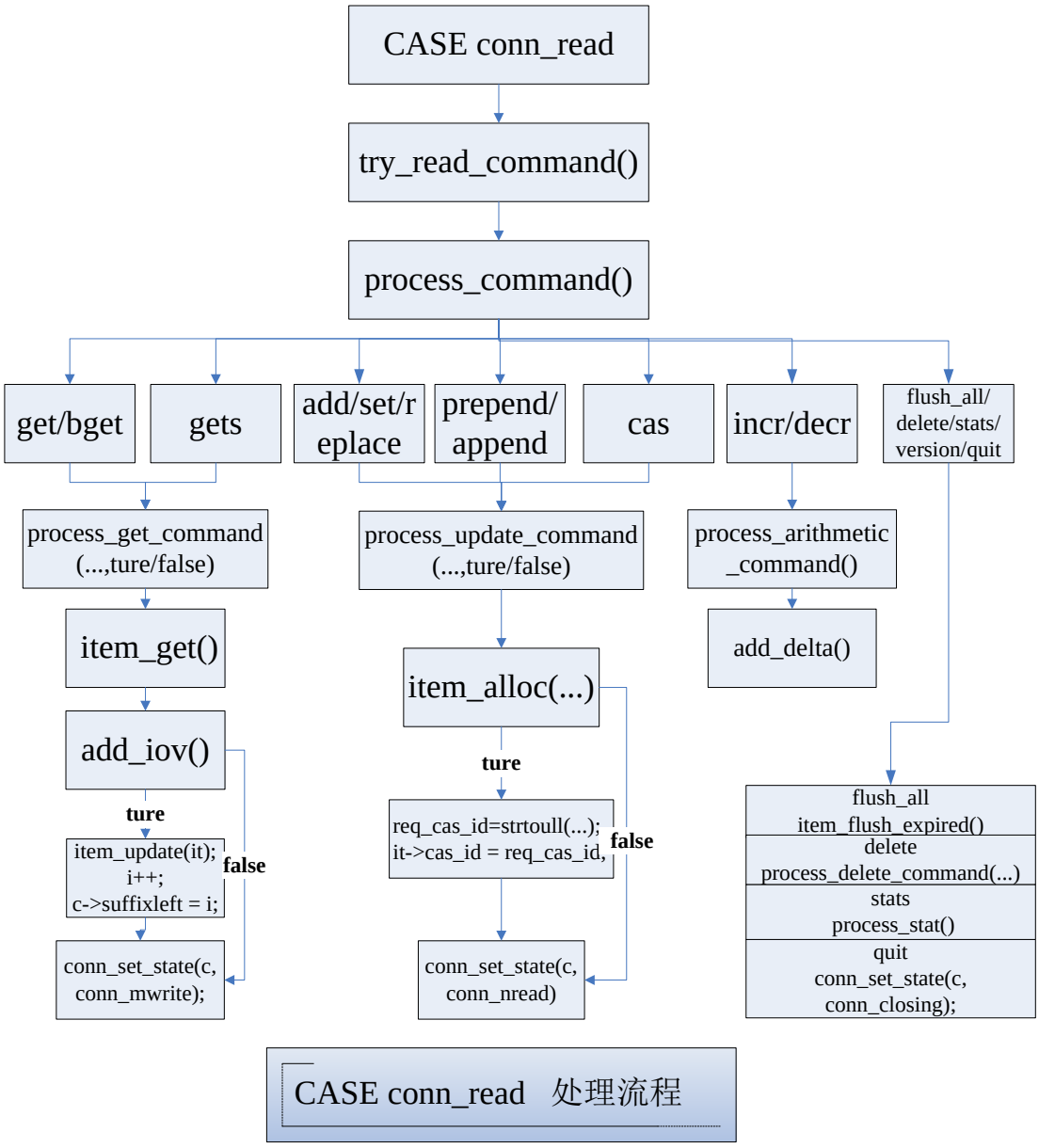
Memcached主函数处理流程图



主要函数逐级调用关系

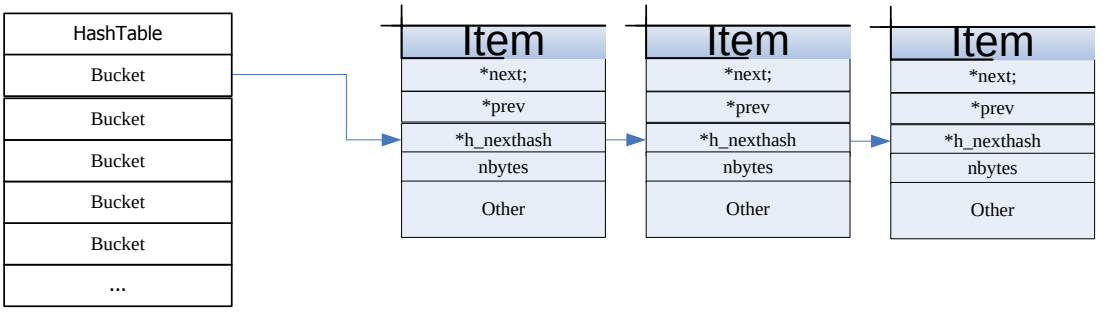


Memcached核心函数drive_machine()



在这里值得注意的是，add/set/replace/cas 这类存储命令在 conn_read 里的操作只是分配了 item 的空间，并没有对其进行存储工作，当 conn_read 结束后设置状态为 conn_nread 再做进一步的处理。

5.3. 内部 Hash 机制



Memcached采用开链法（separate chaining）解决冲突

5.3.1. Hash 函数及冲突解决

memcached 采用的 hash 函数是 Bob Jenkins 先生在 1996 创立的一个算法，复杂度为 O（6n+35），而且冲突率极低，该算法具体过程可以参阅这里。冲突处理的方法为开链法。

memcached 中实际有两个 hash 表，一个是“主 hash 表”（primary_hashtable），另外一个是“原有 hash 表”（old_hashtable）。每次操作的时候，先会检测表是否正处于扩展(expanding)状态，如果扩展还没完成时，先在原有 hash 表中操作数据。

5.3.2. HashTable 主要函数

assoc_init()

初始化 hash 表，为主 hash 表分配空间。

assoc_find()

根据 key 值来查找 item，如果有冲突存在，则不停往桶的下一个元素（h_next）里查找，直至找到为止，并返回其指针。

assoc_insert()

在 hash 表中插入 item，如果装载因子大于 1.5,则扩展哈希表

assoc_delete()

用_hashitem_befor()找键值为 key 的 item 之前的指针，改变其指向，数据实际上是没有被释放的，在这里只是从 hash 表中移除。

assoc_expend()

扩展 hash 表到 2 的下一次方，比如现在是 hash 表的大小 2^16，扩展后大小则为 2^17。再进行数据迁移，扩展时候不能再分配内存时，就保持原有的表不变。

do_assoc_move_next_bucket()

将下一个桶迁移到先前的我们扩充的哈希表中

_hashitem_before ()

（内部使用）返回该键值所对应的 item 的之前的指针。

5.4. slab 内存处理机制

slab 源于 Jeff Bonwick 为 SunOS 操作系统首次引入的一种内存处理机制，SLAB 的设计理念是基于对象缓冲的，基本想法是避免重复大量的初始化和清理操作。SLAB 主要可以用于频繁分配释放的内存对象。如果是采用系统自带的 malloc/free 话，反复地操作会造成大量内存碎片，操作系统将会花费大量的时间去查找连续的内存块来满足 malloc 的请求。

memcached 中内存分配机制主要理念

1. 先为分配相应的大块内存，再在上面进行无缝小对象填充
2. 懒惰检测机制，Memcached 不花过多的时间在检测各个 item 对象是否超时，当 get 获取数据时，才检查 item 对象是否应该删除，你不访问，我就不处理。
3. 懒惰删除机制，在 memecached 中删除一个 item 对象的时候，并不是从内存中释放，而是单单的进行标记处理，再将其指针放入 slot 回收插糟，下次分配的时候直接使用。

5.4.1. slab 主要函数

slabs_init()

slab 初始化，如果配置时采用预分配机制(prealloc)则在先在这使用 malloc 分配所有内存。

再根据增长因子 factor 给每个 slabclass 分配容量。

slabs_clsid()

计算出哪个 slabclass 适合用来储存大小给定为 size 的 item，如果返回值为 0 则存储的物件过大，无法进行存储。

do_slabs_alloc()

在这个函数里面，由宏定义来决定采用系统自带的 malloc 机制还是 memcached 的 slab 机制对内存进行分配，理所当然，在大多数情况下，系统的 malloc 会比 slab 慢上一个数量级。

分配时首先考虑 slot 内的空间(被回收的空间),再检查 end_page_ptr 指针指向的的空闲空间，还是没有的空间的话，再试试分配新的内存。如果所有空间都用尽的时候，则返回 NULL 表示目前资源已经枯竭了。

do_slabs_free()

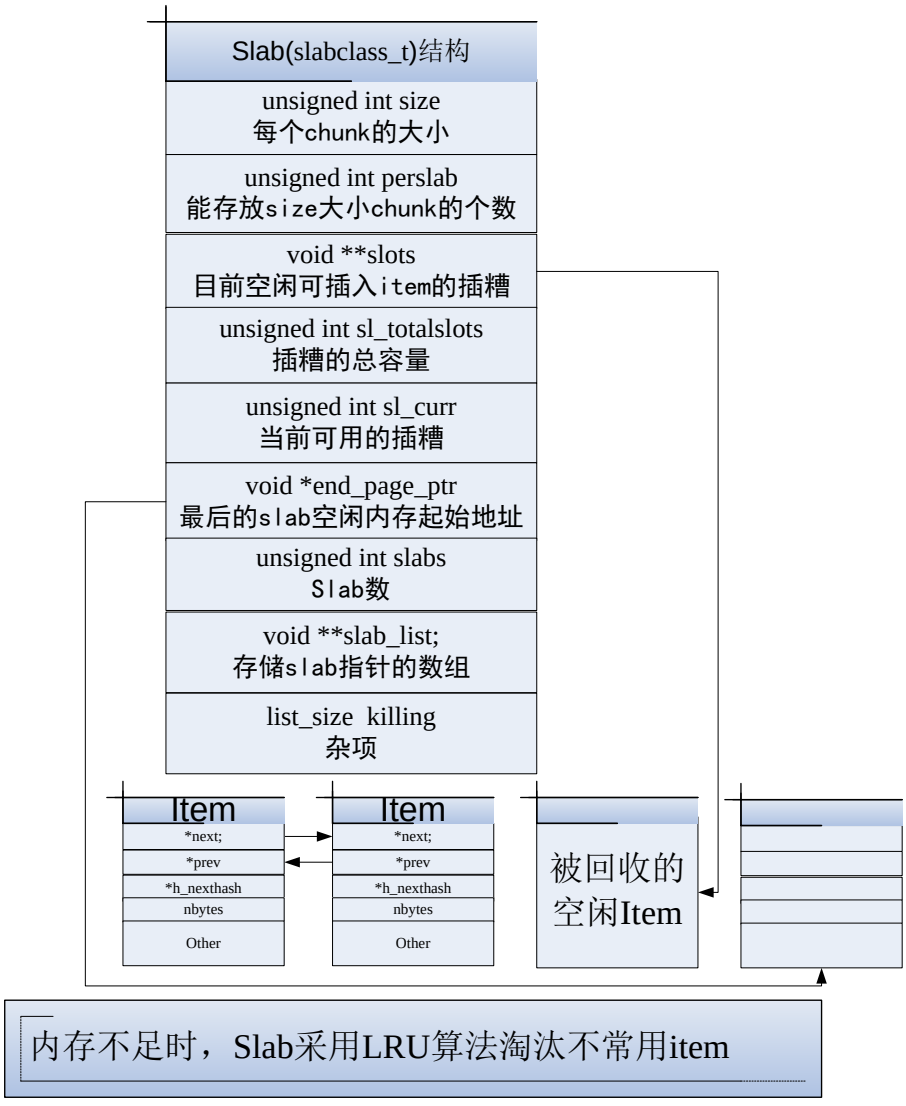
首先检查当目前的插糟是否已经达到可用总插糟的总容量，如果达到就为其重新分配空间，再将该回收的 item 的指针插入对应当前 id 的 slabclass 的插糟（slots）之中。

do_slabs_stats()

将目前 slab 的状态填充至 buf 缓存中并将其返回。

do_slab_reassign()

清除在一个 slab class 里的所有的 item，将其移动到另外一个 class 。只有在处理”slab reassign”命令选择手动调整内存分配的时候才会使用，默认是禁止的。



5.4.2. slab 机制中所采用的 LRU 算法

在 memcached 运行过程中，要把一个 item 调入内存，但内存已无空闲空间时，为了保证程序能正常运行，系统必须从内存中调出一部分数据，送磁盘的对换区中。但应将哪些数据调出，须根据一定的算法来确定。通常，把选择换出数据(页面)的算法称为页面置换算法(Page Replacement Algorithms)。 Memcached 采用最近最久未使用（LRU）置换算法，是根据数据（页面）调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU 置换算法是选择最近最久未使用的页面予以淘汰。当内存不足时，memcached 会从 slab 各个 class 中的双向链表的尾部开始检测，即最近最久未使用的页面，往前一直寻找合适的 item 予以淘汰。所以该 LRU 算法为 slab 局部 class 淘汰的机制。但是在一些特定情形也会可能引起一些不必要的麻烦，可以在运行时加入”-M”参数禁止该算法。

5.5. 控制 item 各种函数

do_item_init()

初始化 item：将 itemstats 结构体数组全部初始化为 0，将各个头，尾指针初始化。

do_item_alloc()

首先调用 slabs_clsid 给 item 归类，然后调用 slabs_alloc()函数分配内存，当可使用空间枯竭了的时候，就开始使用 LRU 算法了，从一个对应大小的尾部 tails[id]开始，向前不断尝试能否释放，当发现一个 item 当前没有被使用(引用计数 refcount 为 0)，且其生存时间已经为 0 或生存时间大于现在时间，就果断的把它释放掉。并再次调用 slabs_alloc(),作者在此提到有一种非常罕见的 bug 能够使引用计数(refcount)发生泄漏，处理方法是，当 item 连续保持锁定状态 3 小时以上，说明它已经足够陈旧了，应该果断将其释放。最后再将数据复制到分配的空间内。

item_free()

调用 slabs_free 将其释放。

do_item_link()

1.调用 assoc_insert()将 item 指针插入 hash 表中

2.调用 get_cas_id()给 it 的 cas_id 赋值。

3.调用 item_link_q(),把该 item 插入 LRU 队列的最前面

do_item_unlink ()

1.调用 assoc_delete()在 hash 表中删除此 item

2.调用 item_unlink_q()从该 slab class 去除此 item 的连接，此处考虑了 item 在链表头部，尾部等各种情况。

3.最后当引用计数为 0 的时候，调用 item_free()将其释放。

do_item_remove ()

减少引用计数 refcount，当发现引用计数为 0 的时候，就将其释放。

do_item_update ()

先调用 item_unlink_q(),更新了时间以后，再调用 item_link_q()。将其重新连接到 LRU 队列之中，即让该 item 移到 LRU 队列的最前。

do_item_replace ()

调用 do_item_unlink()解除原有 item 的连接，再调用 do_item_link()连接到新的 item。

item_get ()

值得说明的是，memcached 的懒惰删除逻辑在这里有体现。就是当你需要 get 一个 item 的时候才考虑该 item 是否应该删除。

首先调用 do_item_get_notedeleted()函数，根据关键字使用 assoc_find()查找 item,如果没找到，返回 NULL，再判断是否达到最大的生存时间，如果是的话，就 do_item_unlink 该 item，返回 NULL。

如果该 item 没有满足删除条件，将其引用计数加 1，并返回该 item 的指针。

5.6. 守护进程机制

Memcached 使用经典的 UNIX daemon 模式(daemon.c)。具体工作流程处理如下

Memcached守护进程处理流程图

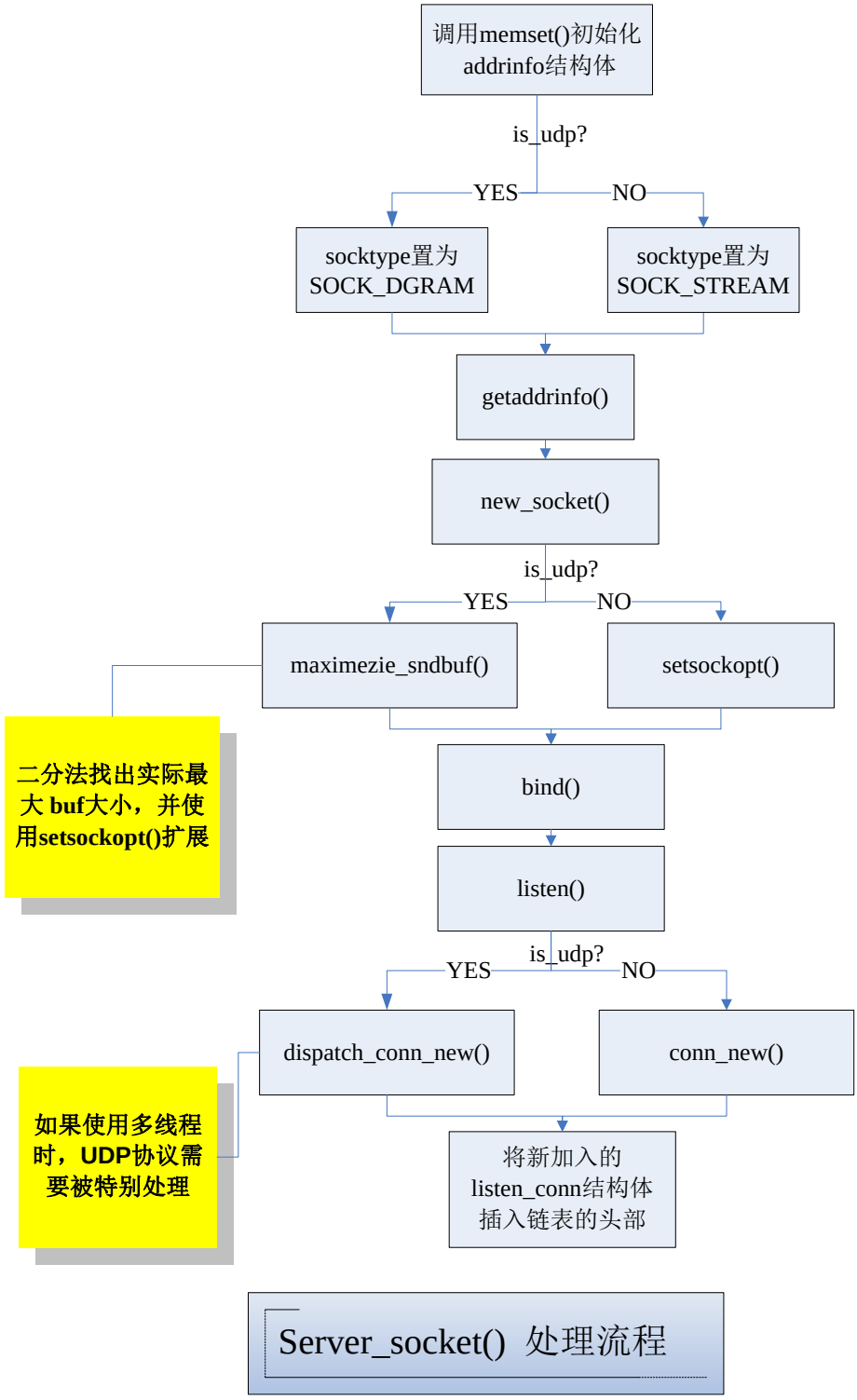
5.7. Socket 处理机制

5.7.1. Unix 域协议

Unix 域协议并不是一个实际的协议族，它只是在同一主机上进行客户--服务器通信时，使用与在不同主机上进行客户--服务器通信时使用的相同的 API 的一种方法。memcached 默认是不使用该机制，可以使用-s 参数设置 sockpath 开启。

server_socket_unix()处理流程图

5.7.2. TCP/UDP 协议



通过设置 TCP 或者 UDP 端口为 0 可以选择不打开其中的协议。

5.8. 多线程处理机制

memcached 1.2 之后开始加入多线程机制，在头文件 memcached.h 中使用宏来决定是否使用线程，一般以“do_”开头的就为单线程函数，以“mt_”开头则为多线程的版本。

memcached 使用的是 pthread（POSIX）线程模式，默认并没有开启，在编译前使用“./configure --enable-threads”可以开启多线程模式。

目前来说，memcached 的多线程设计还是比较粗糙，特别是其中的锁(locking)不够完善，毕竟该项目一开始并没有考虑加入多线程，现在加入的多线程机制也只是通过一系列的包裹(wrapper)函数来实现。当你的网站负载过重时，可以考虑开启多线程模式，官方的文档建议“one thread per processor core”(线程数对应处理器核心数)，开启过多的线程没有实际的优点。可以通过”-t”参数设置线程数。

线程模型可以参考这篇文章。

5.9. 事件处理机制

memcached 采用的是 libevent 框架。libevent 是一种处理异步事件程序库。该库的性能非常优秀，提供的 API 简单易用。这里提一下 memcached 主要用到的几个 API：

event_init() （事件初始化）

表示初始化 libevent 所使用到的变量，并返回这个新建立的 even_base 结构体。

event_set(struct event *ev, int fd, short events,

void (*callback)(int, short, void *), void *arg) (事件设置)

参数：

ev：要设置的 event 对象。

fd：该 event 绑定的“句柄”，对于信号事件，它就是关注的信号。

event：在该 fd 上关注的事件类型，它可以是 EV_READ, EV_WRITE, EV_SIGNAL

callback：这是一个函数指针，当 fd 上的事件 event 发生时，调用该函数执行处理

它有三个参数，调用时由 event_base 负责传入，按顺序，

实际上就是 event_set 时的 fd, event 和 arg。

arg：传递给 cb 函数指针的参数。

event_add(struct event *ev, const struct timeval *tv) (事件添加)

参数：

ev：指向要注册的事件。

tv：超时时间。

```
struct timeval {
    time_t tv_sec;
    useconds_t tv_usec;
    };
```

把 ev 注册到 事件队列之中，第二个参数指定的是 Timeout 时间，设定成 NULL 表示忽略这项设定。

event_base_loop(struct event_base *base, int flag)（进入事件循环）

当事件队列里面的任何一个文件描述符发生事件的时候就会进入回调函数执行。

值得一提的是，libevent 还有较为完善的缓存管理模块和信号处理模块，这里不展开讨论。有兴趣的朋友可以自己读读 libevent 代码。

6. 未完善之处

由于 memcached 的设计目的为一个单一的轻量级的缓存系统，为了速度方面的考虑，忽略了以下的方面

1.安全问题，任何机器都可以通过 telnet 等方式连接到缓存服务器而无须身份验证，这为安全留下了隐患，在设置服务器时需考虑周全。

2.没有合理的日志功能，一旦服务器遇到错误而崩溃，将难以找到错误原因。

3.内存中的数据不够安全，服务器停电或者其他因素将会造成数据的丢失。

4.slab 的 LRU 算法只顾及局部，没有对全局进行操作。

5.slab 处理对象时，会先对其归类，比如 100KB 的对象会放到 120KB 的空间内，会浪费较多的内存空间。

采用 memcached 之前，应该在速度上和安全稳定性考虑折中的方案。如使用和数据库联系的 memcacheDB 或者专门的内存式数据库。

7.参考文献

[1].Masahiro Nagano[JP] & charlee(译).memcached 全面剖析.2008-7-2

[2].W.Richard Stevens & 杨继张(译).UNIX 网络编程(第三版).2004

[3]. W.Richard Stevens.UNIX 环境高级编程(第二版).2005

[4]. dsallings.Memcached FAQ.2009-9

[5]. bachmozart .Memcached 源码分析(线程模型).

[6]. 爱写字开发博客.Linux 下启用 Wordpress 的 memcached 支持.