fatcache代码学习

fatcache是ssd版本的memcache，使用ssd存储item，内存中一部分空间维护metadata作为索引，一部分存储item，作为ssd的buffer，也有助于提高性能。当内存中存储满后，便将其中的slab写入disk；当二者都满后，将disk中最老的slab删除，为新的item腾出位置。

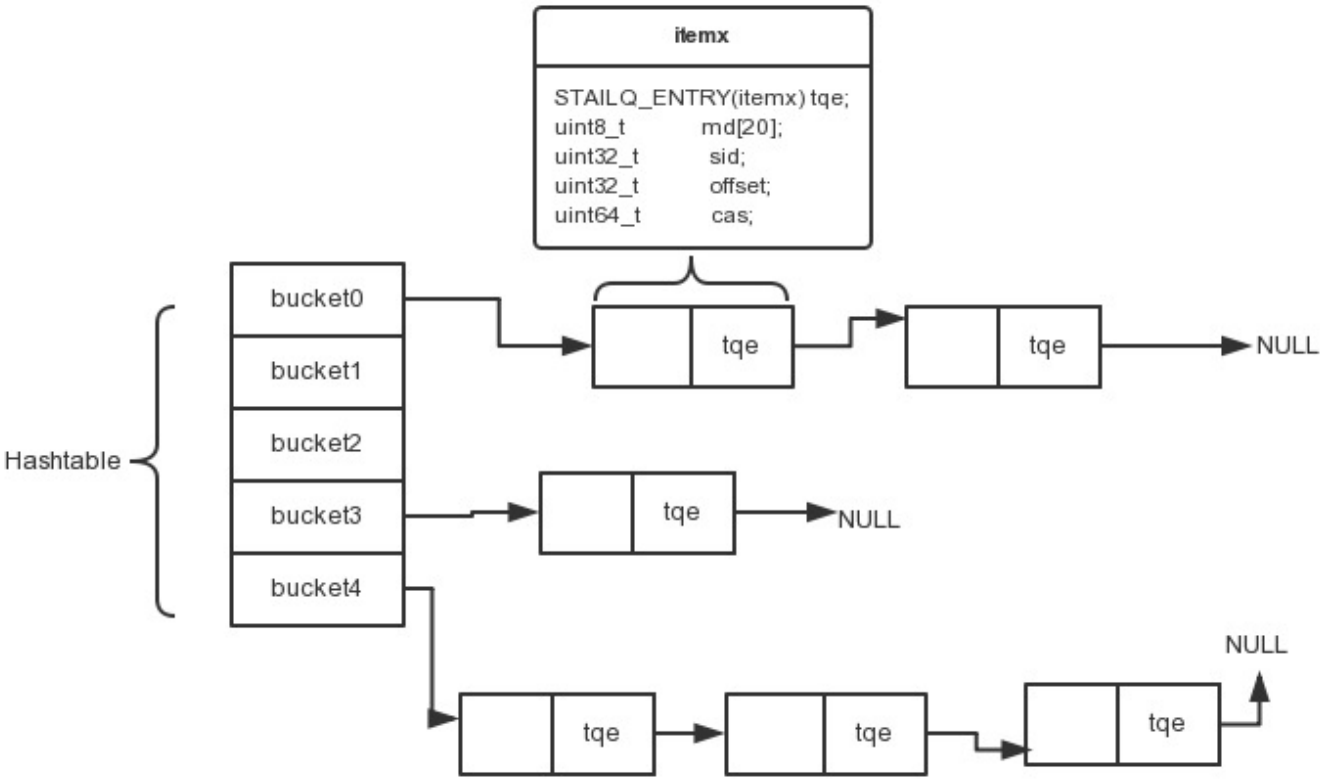
overview

init

为了减少经常性的内存分配和释放，在启动时就设定好相关的参数，比如hash-power(索引表的大小)、factor（slabclass的增长因子）、max-slab-memory（内存中最多放多少item）等等，在初始化时就根据设定分配固定大小的内存，并使用mmap直接读写disk，减少disk带来的io性能损耗。

itemx

在内存中维护index，可以判断key对应的item是否存在、知道item的具体位置等，而在系统中删除item时只是将其对应的itemx删除、并在其slab中标记，并不进行真正的删除，这样能够大大减少开销。



索引表的数据结构如图所示，采用线性探查法；为了压缩不同长度的key，使用md作为key的替代值，经过hash算法后，每个key的md唯一。索引表的大小也是根据参数分配的，大小固定，所以如果索引表满，也会剔除最老的索引。

tail queue

fatcache中的许多数据使用tail queue组织，而没有直接使用链表。

其本质是一个双向链表，头节点定义如下：

```
#define TAILQ_HEAD(name, type)              \
struct name {                           \
    struct type *tqh_first; /* first element */ \
    struct type **tqh_last; /* addr of last next element */\
}
```

一个指针指向第一个节点，而另一个二级指针，指向**最后一个节点的next指针**。

而节点中用于连接的指针定义如下：

```
#define TAILQ_ENTRY(type)                   \
struct {                                \
    struct type *tqe_next;  /* next element */      \
    struct type **tqe_prev;/* addr of previous next element*/ \
}
```

其中使用一个二级指针，指向前一个节点的next指针的地址，与头节点对应，这样的设计能够减少插入时的指针操作，同时，有`tqe_prev==&(previous_elm->field.tqe_next)`,`*tqe_prev==&elm,**tqe_prev==elm`

获取最后一个节点：

```
#define TAILQ_LAST(head, headname)
    (*(((struct headname *)((head)->tqh_last))->tqh_last))
```

如果能够得到最后一个节点的**tqe_prev**，即可获取最后一个节点。

- (head)->tqh_last)==&(elm->field.tqe_next)，同时也可以看作是field的地址
- 接下来要获得tqe_prev的地址，可以直接加4（在field中跨过当前指针），但是为了**增强可移植性**，发现field内存结构与headname相同，所以将其强制转换为headname
- 通过headname来获取tqe_prev->tqh_last

获取前一个节点：

```
#define TAILQ_PREV(elm, headname, field)
        (*(((struct headname *)((elm)->field.tqe_prev))->tqh_last))
```

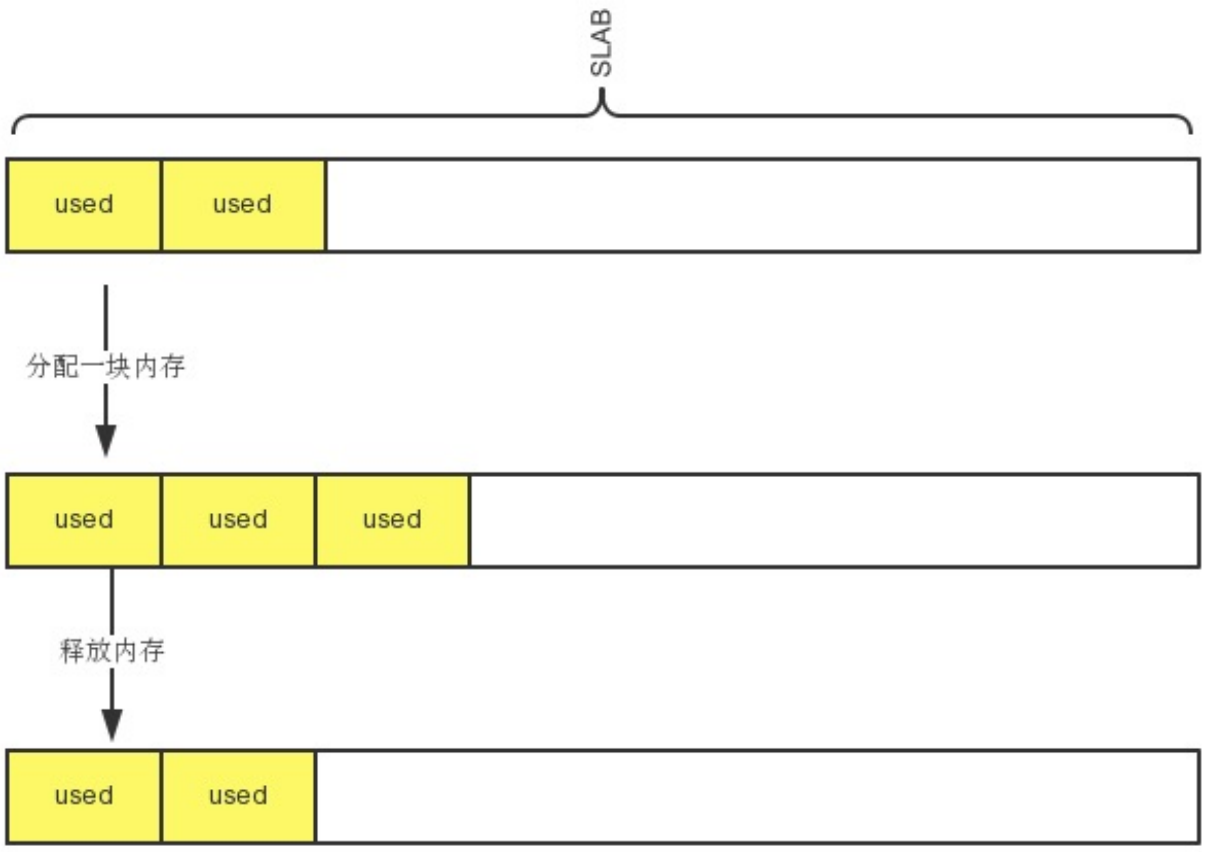
- (elm)->field.tqe_prev)==&(previous_elm->field.tqe_next)，可以看作是前一个节点的field地址
- 通过headname获得tqe_prev，*tqe_prev==pprevious_elm->field.tqe_next==&previous_elm
- 找一个节点，必须先找到其前一个节点

插入：

```
#define TAILQ_INSERT_TAIL(head, elm, field) do {        \
    (elm)->field.tqe_next = NULL;
    //连接二级指针
    (elm)->field.tqe_prev = (head)->tqh_last;     \
      //使用当前尾节点的next将elm连接到队尾
    *(head)->tqh_last = (elm);                \
    (head)->tqh_last = &(elm)->field.tqe_next;        \
} while (0)
```

slab

为了较少内存的分配和回收开销、提高ssd的io性能，在初始化时就分配好空间，即固定数目的slab。每个slab的大小固定，可以存放多个item；为了适应不同大小的item，引入slabclass的概念，按照item大小的不同将slab进行分类，每一级的item大小由初始的参数确定，当写入item时，按照其大小确定slabclass



每次需要分配内存时，就从对应的slab中获取；用完后，在逻辑上释放，放回slab。slab分为三个状态，full、partial、free，引入slabclass之后，full、和free是所有class公用，而每一个class都有一个自己的partial slab队列，写入时优先写入对应class的partial class

epoll

在网络io时，网卡可以将数据传送到内存之中，之后向cpu发出中断信号，os可以通过网卡中断程序处理数据。

最基础的网络io代码：

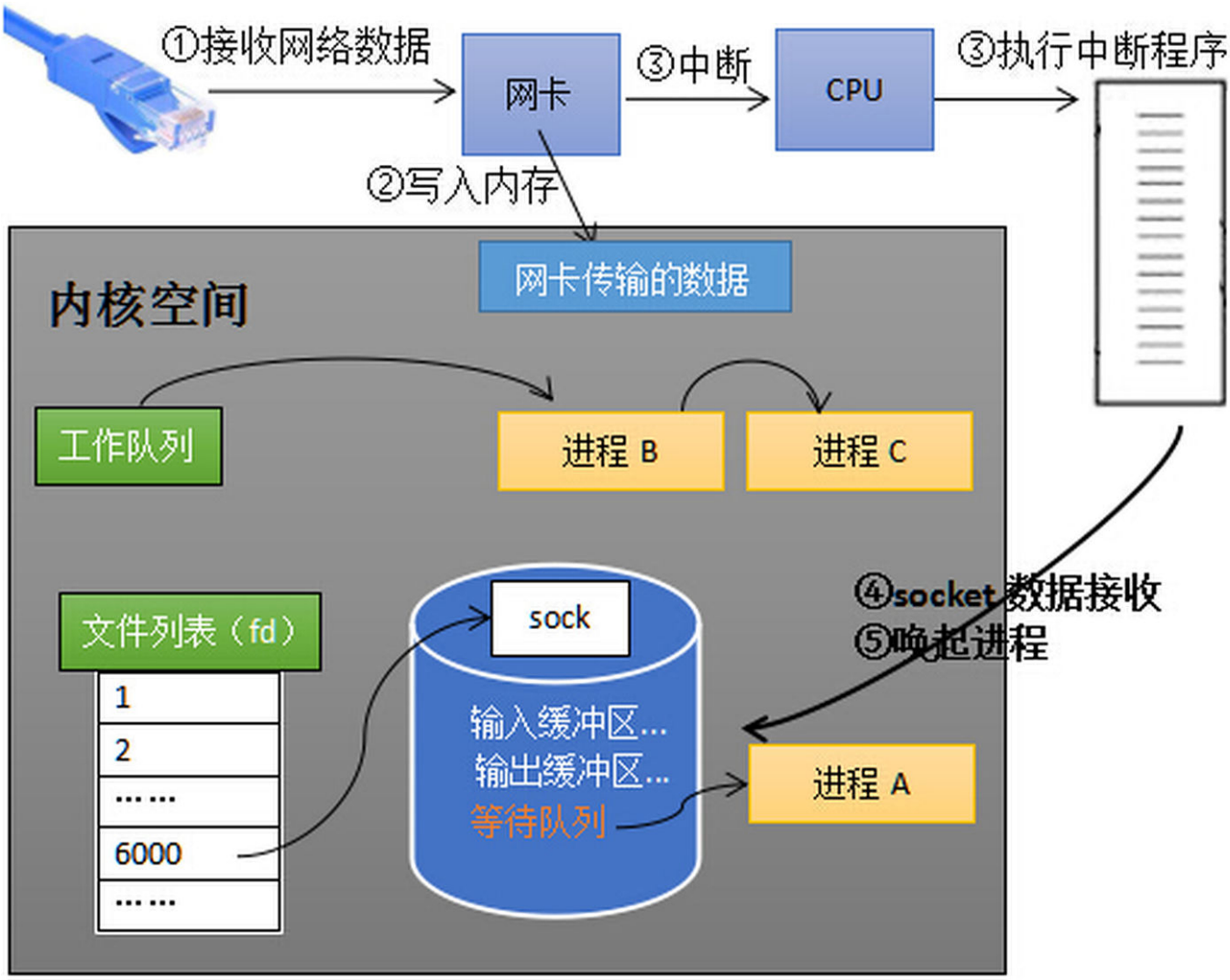
```
//创建socket
int s = socket(AF_INET, SOCK_STREAM, 0);
//绑定
bind(s, ...)
//监听
listen(s, ...)
//接受客户端连接
int c = accept(s, ...)
//接收客户端数据
recv(c, ...);
//将数据打印出来
printf(...)
```

当运行到recv时，进程会被阻塞，一直等待直到接收到数据；而在创建socket时，os会创建一个由filesystem管理的socket对象，其中包含等待队列，指向所有等待该socket的进程。

socket接受到数据后，os将其等待队列上的所有进程放回工作队列，变成运行状态，recv接受到的数据就是socket中接收缓冲区的状态。

os接收数据过程：

- 网卡->内存
- 网卡->中断信号->cpu->中断程序（写数据，唤醒进程）



多路复用

epoll是主要特点是能够**高效监视多个socket**，改进了select和poll

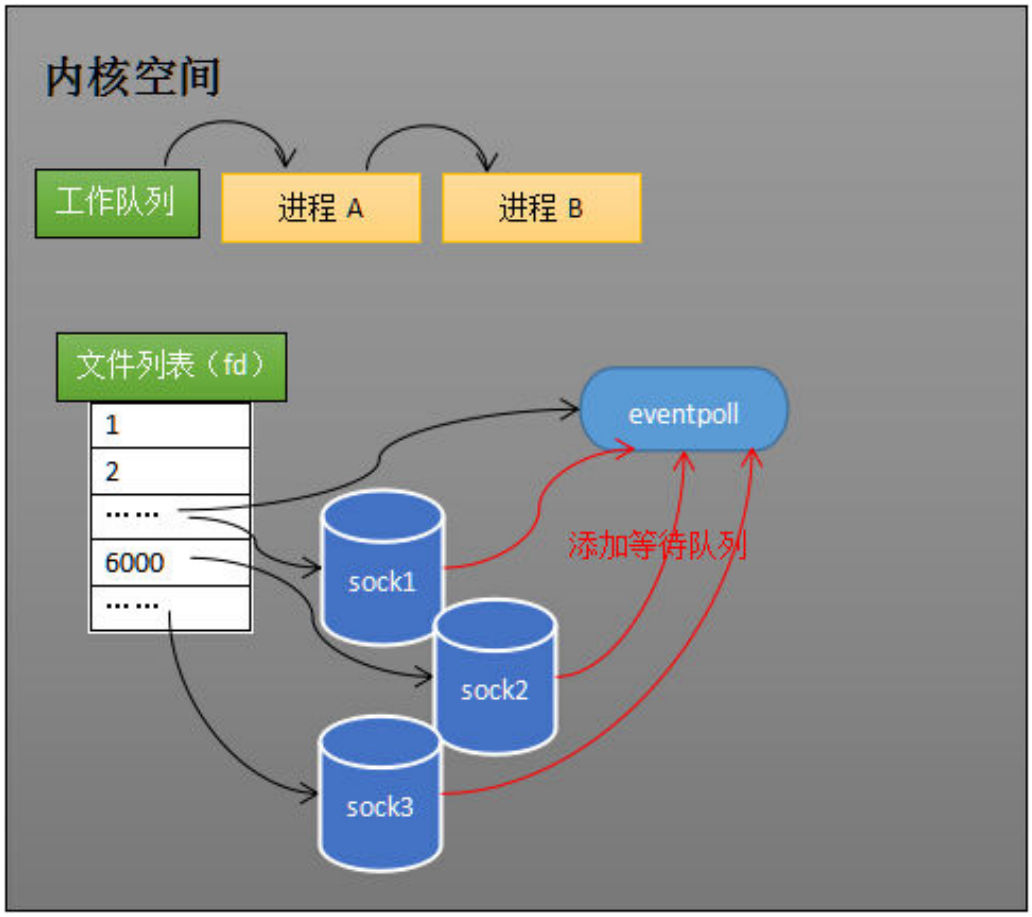
用法：

```
int s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, ...)
listen(s, ...)

int epfd = epoll_create(...);
epoll_ctl(epfd, ...); //将所有需要监听的socket添加到epfd中

while(1){
    int n = epoll_wait(...)
    for(接收到数据的socket){
        //处理
    }
}
```

- 只添加一次需要监听的socket
- 维护一个就绪列表（eventpoll），指向所有收到数据的socket
- 需要被监听的socket将eventpoll加入各自的等待队列，中断程序操作eventpoll

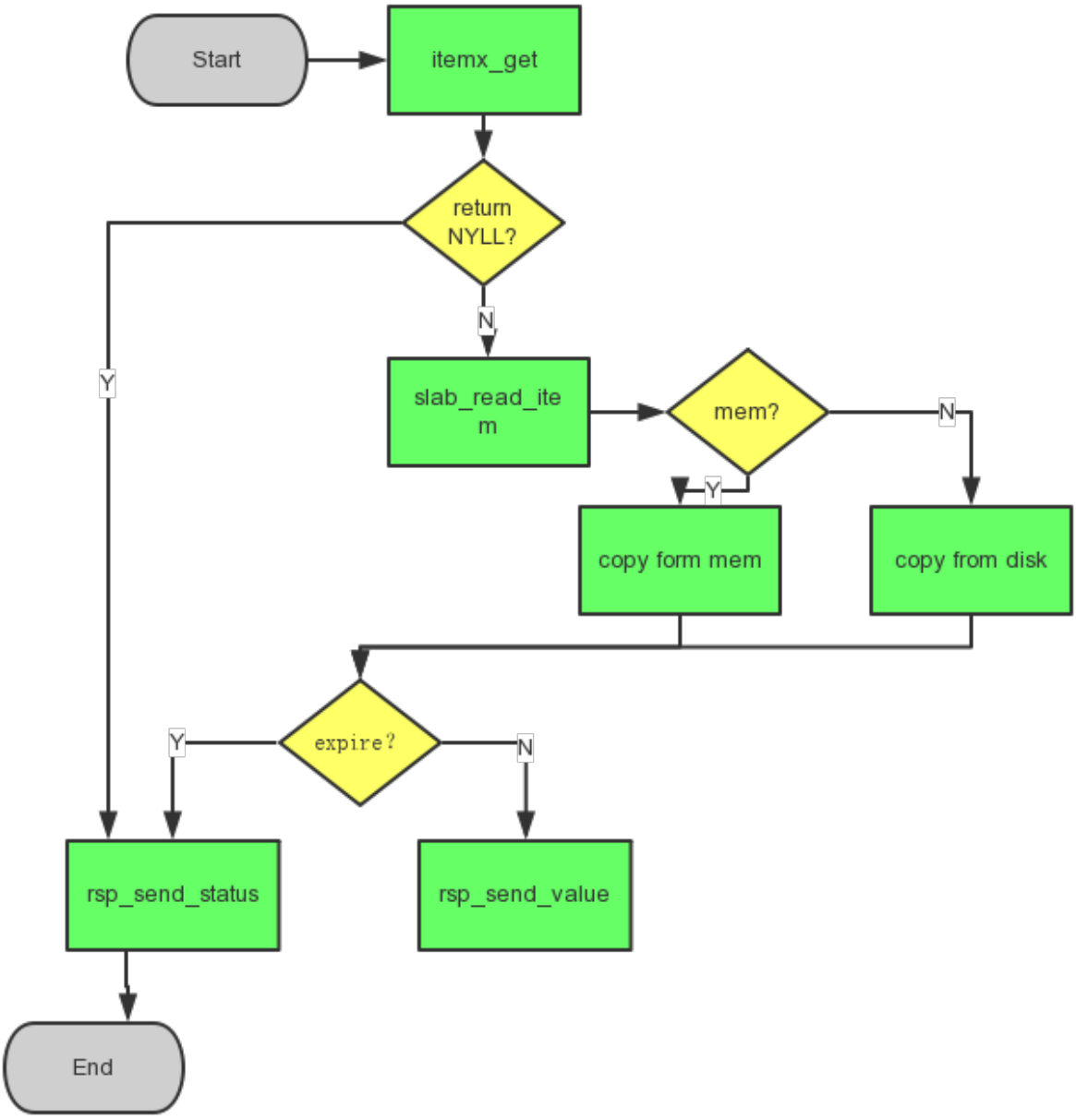


- 进程阻塞后，进入eventpoll的等待队列，rdlist有socket后，恢复进程

epollevent作为socket和进程之间的对象，维护rdlist使得进程不需要便利所有要监听的socket，用的数据结构为双向链表，便于加入和删除socket；同时维护等待队列与所有要监听的socket相连接，要便于查询哪个socket接收到数据，使用红黑树。

operation

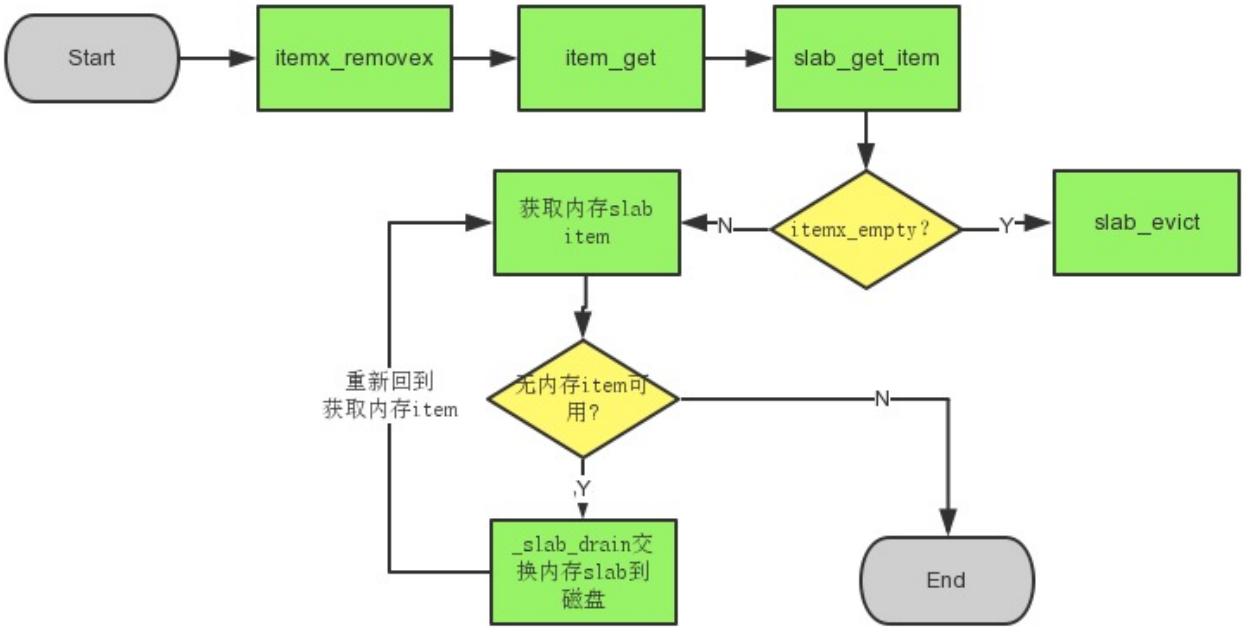
get



- 用hash作为key得到bucket的序号，遍历其中的itemx，寻找md（认为不同的key，md不同）匹配的itemx
- 如果找到了索引且没有过期，则开始查找具体的item
    - 通过slabid确定是否memory slab，如果是的话，直接从内存中根据offset计算其实际地址，根据slabclass大小读取item

- 如果属于disk slab，则首先进行对齐，接着使用pread直接读取
- 每个item中存有一定size的信息和动态大小的data，从其结尾获取data即可

set



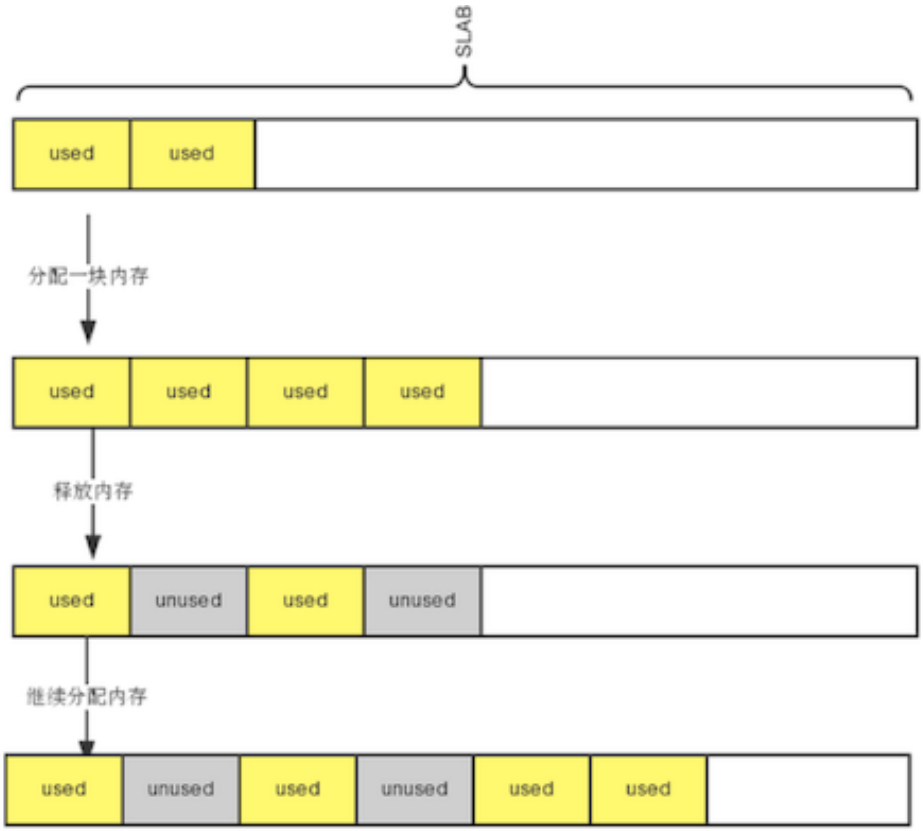
- 检查itemx中是否有相同的key，如果有，删除原有的itemx，并在逻辑上删除slab中的item（后续直接覆盖这块空间）
- 接下来寻找放入item的位置：
  - 根据item大小寻找符合其大小的slabclass，确定cid
  - 如果itemx已经满了，说明整个fatcache都满了，则将ssd中最老的slab删除
    - 读取该slab，将其中的item对应的itemx全部删除，并将其看作free sla
  - 如果slabclass中有partial slab，则从中获取item位置
  - 如果还有free slab，则从中取一个slab变为partial，用与上一步相同的函数即可获得item位置
  - 如果memory slab都满了，则根据disk slab的情况，可以通过清除最老的disk slab，将一个memory slab写入到disk slab
- 最后进行递归调用，多次尝试写入

delete

只从itemx和对应的slab中逻辑上删除

slab hole

在item删除时，直接在metadata中标记item被删除；但是在新item加入slab时，会直接将item放在最新的位置上，被删除的item并没有被利用。只有当其从memory到了disk，最终被删除后，恢复成free slab，才能重新利用这些item。若delete较多，则会造成较大的空间浪费。



reference

[fatcachenote]

[epoll]

[mmap]