

MFS1.6.11源码剖析

唯品会

财务信息研发部/FCS研发组

高级开发经理/王晗

目录

- 系统体系结构的介绍
- 系统模块的主要数据结构和算法介绍
- 系统主要的调用介绍(Read/Write)
- 设计技巧的介绍
- 基于我们的应用，可以做的一些改进
- QA环节

系统体系结构的介绍

- MFS基于FUSE（File system for user space）实现。（一些其他的文件系统也是基于这套机制来实现的，例如gluster/ZFS等）
- 一般操作系统的文件系统，都是在内核态实现的：
 - 优点：运行效率高，内核态task之间是共享物理内存的。
 - 缺点：开发难度大，代码和版本难以维护，影响范围甚广，甚至导致kernel panic等严重问题。
- FUSE包括内核模块和用户态的动态链接库两部分。
 - 优点：开发难度小，用户态开发，代码和版本容易维护，影响范围小，用户态进程地址空间相互独立（虚拟内存管理）。
 - 执行效率较内核态程序低，内核态和用户态需要数据交互。（中断，系统调用，内存映射等手段，需要耗费额外的时钟周期）

系统体系结构介绍



VFS, virtual file system, 是内核态向用户态提供统一的interface。重要的数据结构包括:

1. Super block, 存储文件系统信息
2. inode, 文件控制块的信息
3. file, 用户态进程打开的文件信息, 同inode是多对一。
4. dentry, 目录缓存。

系统体系结构介绍

- FUSE应用的开发很简单，整个FUSE的开发框架类似于Linux Driver的框架。

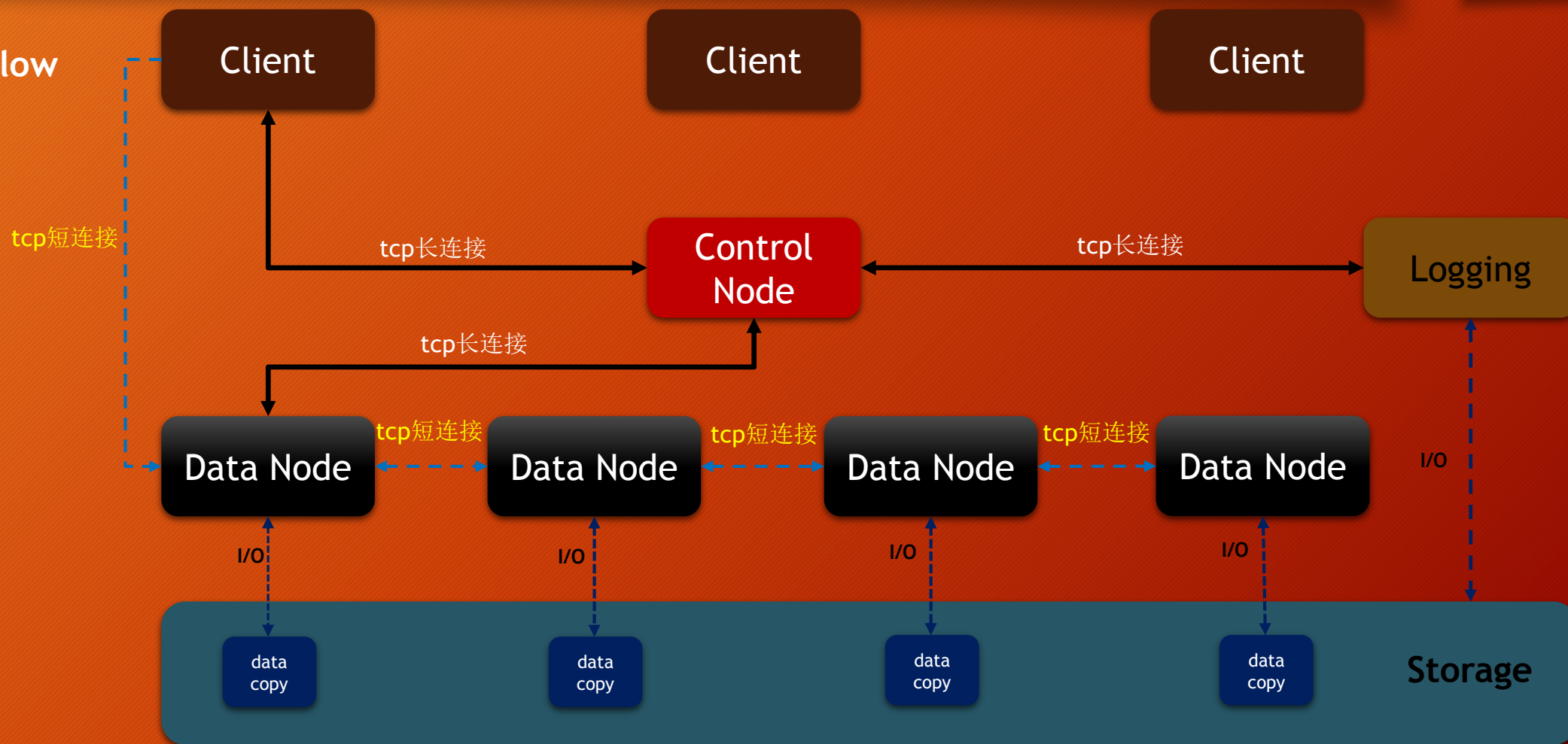
```
00080: static struct fuse_lowlevel_ops mfs_oper = {
00081:     .init          = mfs_fsinit,
00082:     .statfs        = mfs_statfs,
00083:     .lookup        = mfs_lookup,
00084:     .getattr       = mfs_getattr,
00085:     .setattr       = mfs_setattr,
00086:     .mknod         = mfs_mknod,
00087:     .unlink        = mfs_unlink,
00088:     .mkdir         = mfs_mkdir,
00089:     .rmdir         = mfs_rmdir,
00090:     .symlink       = mfs_symlink,
00091:     .readlink      = mfs_readlink,
00092:     .rename        = mfs_rename,
00093:     .link          = mfs_link,
00094:     .opendir       = mfs_opendir,
00095:     .readdir       = mfs_readdir,
00096:     .releasedir    = mfs_releasedir,
00097:     .create        = mfs_create,
00098:     .open          = mfs_open,
00099:     .release       = mfs_release,
00100:     .flush         = mfs_flush,
00101:     .fsync         = mfs_fsync,
00102:     .read          = mfs_read,
00103:     .write         = mfs_write,
00104:     .access        = mfs_access,
00105:     #if FUSE_VERSION >= 26
00106:     /* locks are still in development
00107:     .getlk         = mfs_getlk,
00108:     .setlk         = mfs_setlk,
00109:     */
00110:     #endif
00111: };
```

系统体系结构的介绍

- MFS包括四个主要的模块：Client，Data Node，Control Node和Logging。
 - Client（mfsmount）：FUSE应用，负责接受用户文件系统请求以及转发。
 - Data Node（mfschunkserver）：数据节点的代理进程，执行具体的数据I/O操作。
 - Control Node（mfsmaster）：文件系统资源分配，I/O操作的控制以及负责集群进程的管理。
 - Logging（mfsmetallogger）：负责接受Control Node发送过来的change log的记录。

系统体系结构的介绍

主要的data flow



系统模块的主要数据结构和算法介绍

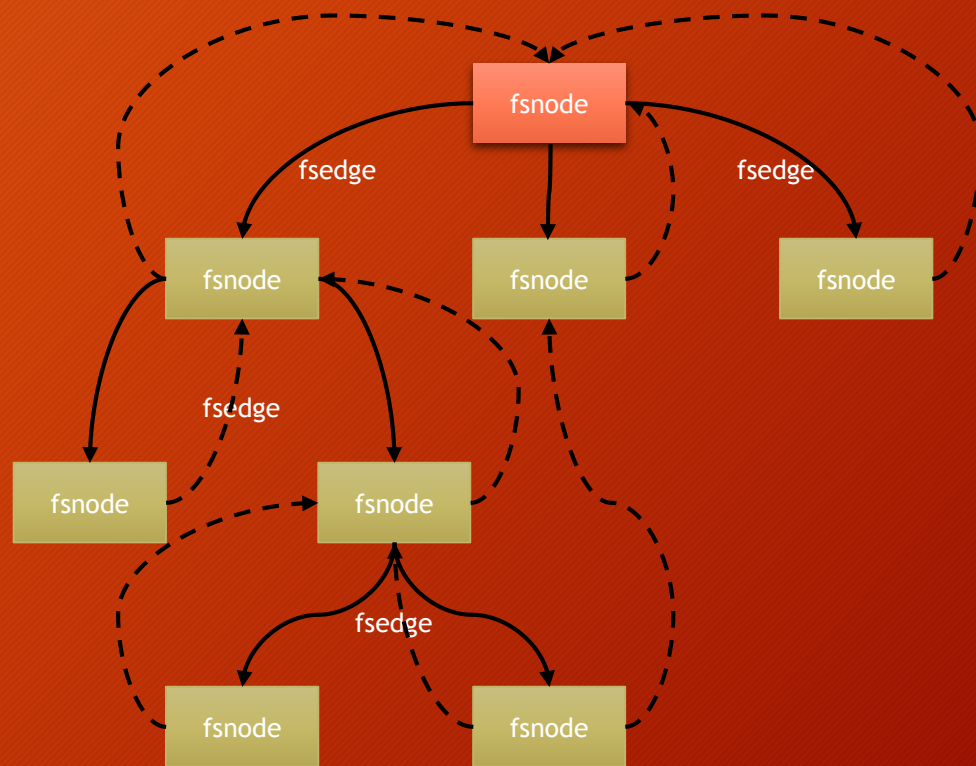
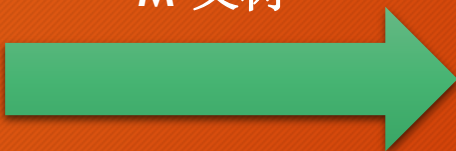
- **mfsmaster**是文件系统在分布式环境下的资源分配和管理者，也是文件I/O操作的中央调度者。
- 每一个挂载目录，I/O操作接收者，**mfsmount**进程都将同**mfsmaster**建立TCP长连接通信信道。
- 每一个I/O操作的执行者和路由者，**mfschunkserver**，都将同**mfsmaster**建立TCP长连接通信信道。
- 每一个**metalogger**进程实例，都将同**mfsmaster**建立TCP长连接通信信道，**mfsmaster**会将change log实时的发送到**metalogger**进程，进行异步change log存储。

系统模块的主要数据结构和算法介绍

- Mfsmaster文件系统主要数据结构，fsnode, fsedge, freenode, freebitmask, chunk和statistic data, 也组成了MFS的Meta data。

逻辑结构

M-叉树



fsnode: 文件的控制信息

fsedge: 文件控制节点的关联关系

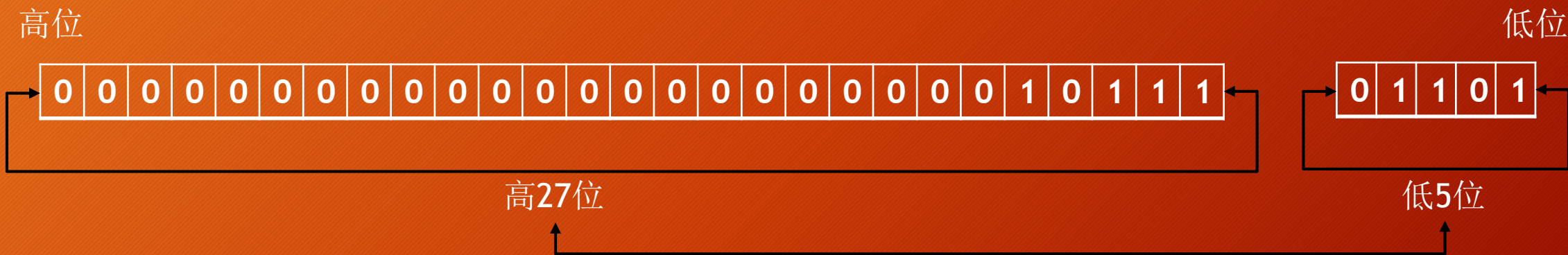
chunk: 文件的数据信息

freenode: 文件控制节点缓存

freebitmask: 文件系统的空闲资源

系统模块的主要数据结构和算法介绍

- 空闲资源位图，freebitmask
 - 管理整个文件系统控制节点fsnode资源状态。
 - 为控制节点fsnode分配node id



```
00455: void fsnodes_used_inode (uint32_t id) {
00456:     uint32_t pos,mask;
00457:     pos = id>>5;
00458:     mask = 1<<(id&0x1F);
00459:     freebitmask[pos]|=mask;
00460: }
```

32位 nodeid

Node Id: 0x02DD
Hash position: 0x1D
Mask position: 0x0B

系统模块的主要数据结构和算法介绍

- 空闲资源位图，freebitmask（续）
 - 空闲fsnode的初始数量为：256到 $2^{20}+256$
 - 空闲fsnode的动态增长

```
00447: void fsnodes_init_freebitmask (void) {
00448:     bitmasksize = 0x100+(((maxnodeid)>>5)&0xFFFFF80);
00449:     freebitmask = (uint32_t*)malloc(bitmasksize*sizeof(uint32_t));
00450:     memset(freebitmask,0,bitmasksize*sizeof(uint32_t));
00451:     freebitmask[0]=1;    // reserve inode 0
00452:     searchpos = 0;
00453: }
```

```
00369: uint32_t fsnodes_get_next_id() {
00370:     uint32_t i,mask;
00371:     while (searchpos<bitmasksize && freebitmask[searchpos]==0xFFFFFFFF) {
00372:         searchpos++;
00373:     }
00374:     if (searchpos==bitmasksize) {    // no more freeinodes
00375:         bitmasksize+=0x80;
00376:         freebitmask = (uint32_t*)realloc(freebitmask,bitmasksize*sizeof(uint32_t));
00377:         memset(freebitmask+searchpos,0,0x80*sizeof(uint32_t));
00378:     }
00379:     mask = freebitmask[searchpos];
00380:     i=0;
00381:     while (mask&1) {
00382:         i++;
00383:         mask>>=1;
00384:     }
00385:     mask = 1<<i;
00386:     freebitmask[searchpos] |= mask;
00387:     i+=(searchpos<<5);
00388:     if (i>maxnodeid) {
00389:         maxnodeid=i;
00390:     }
00391:     return i;
00392: } ? end fsnodes_get_next_id ?
```


系统模块的主要数据结构和算法介绍

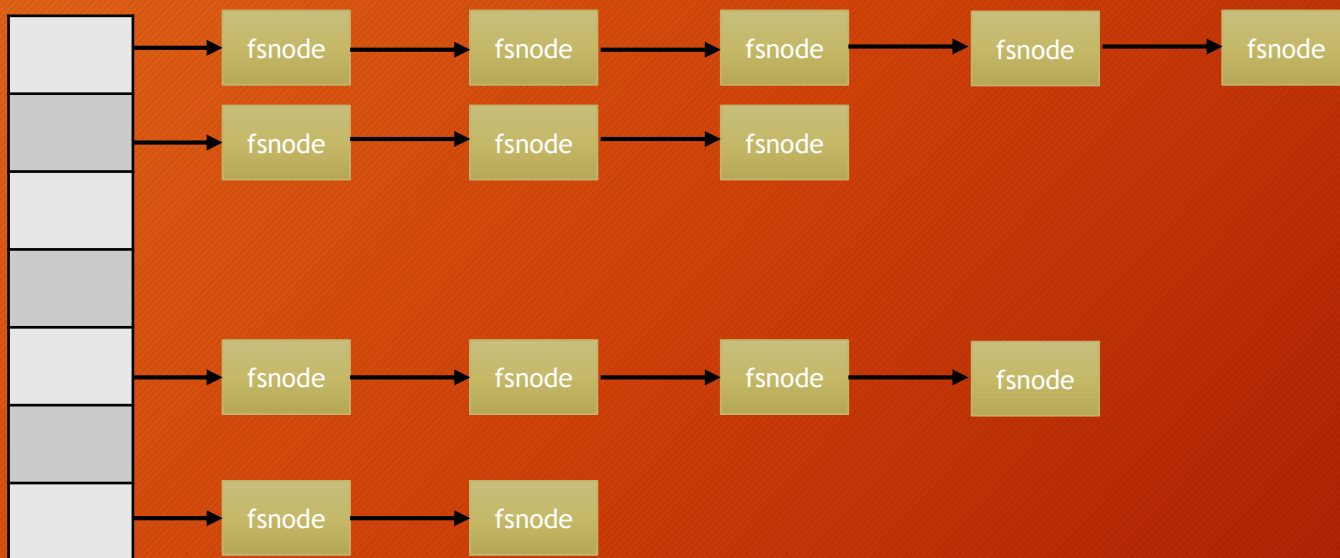
- Mfsmaster核心数据结构，fsnode（续）

```
00137: typedef struct _fsnode {
00138:     uint32_t id;
00139:     uint32_t ctime, mtime, atime;
00140:     uint8_t type;
00141:     uint8_t goal;
00142:     uint16_t mode; // only 12 lowest bits are used for mode, in unix standard upper 4 are used for object type, but since there is field "type" this bits can be used as extra flags
00143:     uint32_t uid;
00144:     uint32_t gid;
00145:     uint32_t trashtime;
00146:     union _data {
00147:         struct _ddata { // type==TYPE_DIRECTORY
00148:             fsedge *children;
00149:             uint32_t nlink;
00150:             uint32_t elements;
00151:             // uint8_t quotaexceeded:1; // quota exceeded
00152: #ifndef METARESTORE
00153:             statsrecord *stats;
00154:             quotanode *quota;
00155: #endif
00156:         } ddata;
00157:         struct _sdata { // type==TYPE_SYMLINK
00158:             uint32_t pleng;
00159:             uint8_t *path;
00160:         } sdata;
00161:         uint32_t rdev; // type==TYPE_BLOCKDEV ; type==TYPE_CHARDEV
00162:         struct _fdata { // type==TYPE_FILE
00163:             uint64_t length;
00164:             uint64_t *chunktab;
00165:             uint32_t chunks;
00166:             sessionidrec *sessionids;
00167:         } fdata;
00168:     } ? end_data ? data;
00169:     fsedge *parents;
00170:     struct _fsnode *next;
00171: } ? end_fsnode ? fsnode;
```

系统模块的主要数据结构和算法介绍

- Mfsmaster核心数据结构，fsnode（续）

Node hash



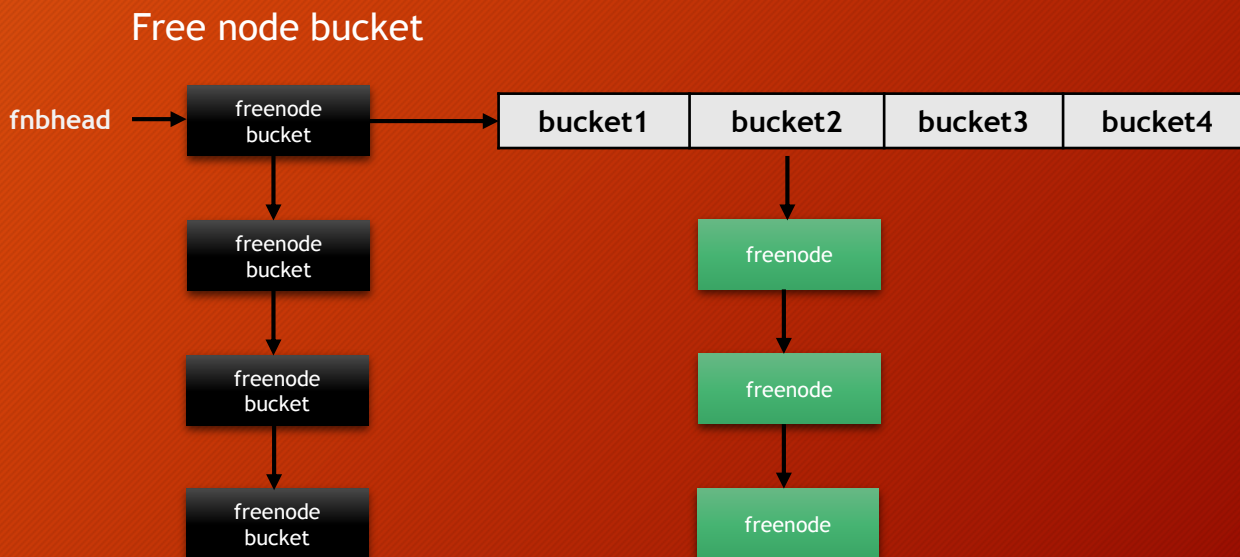
```
00057: #define NODEHASHBITS (22)
00058: #define NODEHASHSIZE (1<<NODEHASHBITS)
00059: #define NODEHASHPOS(nodeid) ((nodeid)&(NODEHASHSIZE-1))
```

系统模块的主要数据结构和算法介绍

- Mfsmaster核心数据结构，freenode（续）
 - 当remove/unlink/purge掉fsnode时，除了会释放fsnode的内存空间之外，还会将已释放的node id和free time保存在free node cache中。（并没有真正释放资源）
 - 有一个定时器线程，每一秒执行一次，将超过一天的freenode进行释放，并且重置freebitmask。

```
00173: typedef struct _freenode {  
00174:     uint32_t id;  
00175:     uint32_t ftime;  
00176:     struct _freenode *next;  
00177: } freenode;
```

```
00276: #define FREENODE_BUCKET_SIZE 5000  
00277:  
00278: typedef struct _freenode_bucket {  
00279:     freenode bucket[FREENODE_BUCKET_SIZE];  
00280:     uint32_t firstfree;  
00281:     struct _freenode_bucket *next;  
00282: } freenode_bucket;
```



系统模块的主要数据结构和算法介绍

- Mfsmaster核心数据结构，freenode（续）

```
00404: #ifndef METARESTORE
00405: void fsnodes_freeinodes(void) {
00406: #else
00407: uint8_t fs_freeinodes(uint32_t ts,uint32_t freeinodes) {
00408: #endif
00409:     uint32_t fi,now,pos,mask;
00410:     freenode *n,*an;
00411: #ifndef METARESTORE
00412:     now = main_time();
00413: #else
00414:     now = ts;
00415: #endif
00416:     fi = 0;
00417:     n = freelist;
00418:     while (n && n->ftime+86400<now) {
00419:         fi++;
00420:         pos = (n->id >> 5);
00421:         mask = 1<<(n->id&0x1F);
00422:         freebitmask[pos] &= ~mask;
00423:         if (pos<searchpos) {
00424:             searchpos = pos;
00425:         }
00426:         an = n->next;
00427:         freenode_free(n);
00428:         n = an;
00429:     }
00430:     if (n) {
00431:         freelist = n;
00432:     } else {
00433:         freelist = NULL;
00434:         freetail = &(freelist);
00435:     }
00436: #ifndef METARESTORE
00437:     changelog(version+,"%PRIu32"|FREEINODES():%"PRIu32,(uint32_t)main_time(),fi);
00438: #else
00439:     version++;
00440:     if (freeinodes!=fi) {
00441:         return 1;
00442:     }
00443:     return 0;
00444: #endif
00445: } ? end fsnodes_freeinodes ?
```

真正释放文件系统fsnode节点资源
重置位图掩码

系统模块的主要数据结构和算法介绍

- Mfsmaster核心数据结构，fsedge（续）
 - 建立fsnode的关联关系

```
00098: typedef struct _fsedge {
00099:     struct _fsnode *child, *parent;
00100:     struct _fsedge *nextchild, *nextparent;
00101:     struct _fsedge **prevchild, **prevparent;
00102: #ifdef EDGEHASH
00103:     struct _fsedge *next, **prev;
00104: #endif
00105:     uint16_t nleng;
00106:     uint8_t *name;
00107: } fsedge;
```

Child指针，指向该边关联的子节点

Parent指针，指向该边关联的父节点

Nextchild/Prevchild指针，将该边链入parent节点的和子节点关联的fsedge双向链表中

Nextparent/Prevparent指针，将该边链入child节点的和父节点关联的fsedge双向链表中

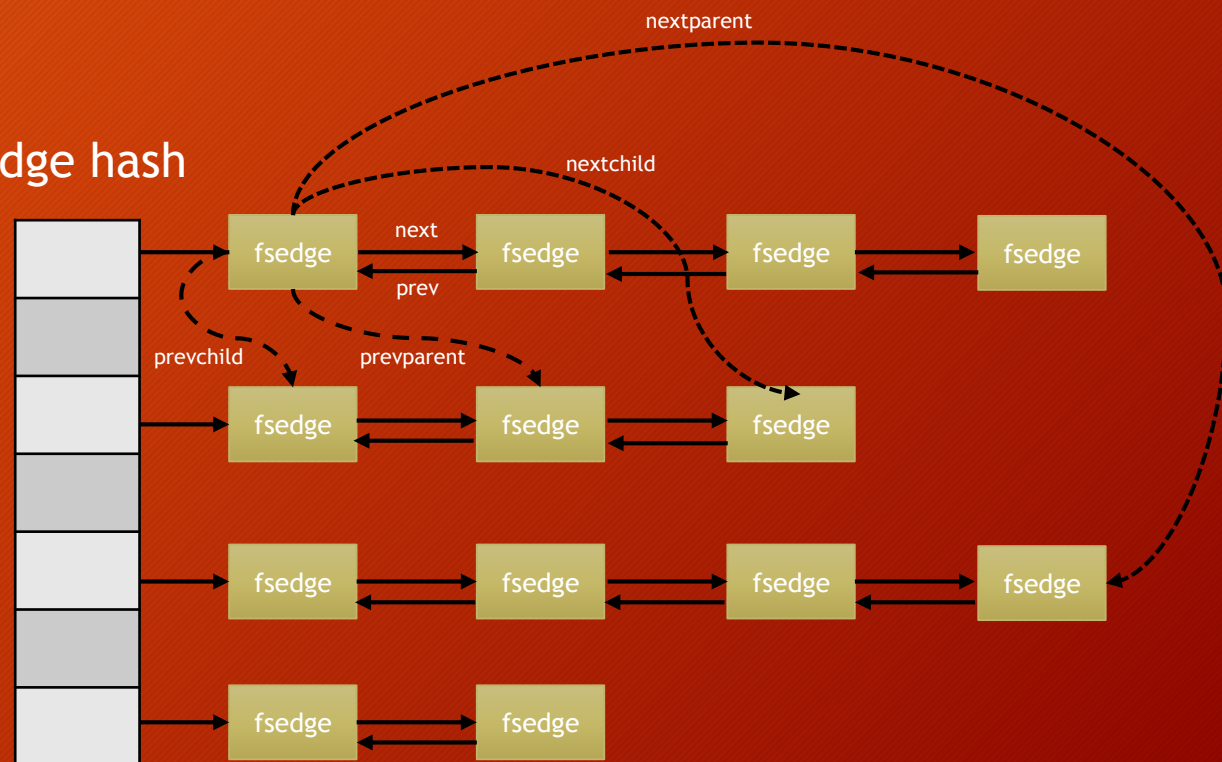
系统模块的主要数据结构和算法介绍

- Mfsmaster核心数据结构，fsedge（续）
 - 建立fsnode的关联关系

```
00062: #define EDGEBHASHBITS (22)
00063: #define EDGEBHASHSIZE (1<<EDGEBHASHBITS)
00064: #define EDGEBHASHPOS(hash) ((hash)&(EDGEBHASHSIZE-1))
00065: #define LOOKUPNOHASHLIMIT 10
```

```
00521: static inline uint32_t fsnodes_hash(uint32_t parentid, uint16_t nleng, const uint8_t *name) {
00522:     uint32_t hash, i;
00523:     hash = ((parentid * 0x5F2318BD) + nleng);
00524:     for (i=0 ; i<nleng ; i++) {
00525:         hash = hash*33+name[i];
00526:     }
00527:     return hash;
00528: }
```

Edge hash



系统模块的主要数据结构和算法介绍

• Mfsmaster创建fsnode的过程

```
static inline fsnode* fsnodes_create_node(uint32_t ts, fsnode* node, uint16_t nlen, const uint8_t *name, uint8_t type, uint16_t mode, uint32_t uid, uint32_t gid) {
    fsnode *p;
#ifdef METARESTORE
    statsrecord *sr;
#endif
    uint32_t nodespos;
    p = malloc(sizeof(fsnode));
    nodes++;
    if (type==TYPE_DIRECTORY) {
        dirnodes++;
    }
    if (type==TYPE_FILE) {
        filenodes++;
    }
    /* create node */
    p->id = fsnodes_get_next_id();
    p->type = type;
    p->xtime = p->mtime = p->atime = ts;
    if (type==TYPE_DIRECTORY || type==TYPE_FILE) {
        p->goal = node->goal;
        p->trashtime = node->trashtime;
    } else {
        p->goal = DEFAULT_GOAL;
        p->trashtime = DEFAULT_TRASHTIME;
    }
    if (type==TYPE_DIRECTORY) {
        p->mode = (mode&0777) | (node->mode&0xF000);
    } else {
        p->mode = (mode&0777) | (node->mode&(0xF000&~(EATTR_NOECACHE<<12))));
    }
    p->uid = uid;
    p->gid = gid;
    switch (type) {
        case TYPE_DIRECTORY:
#ifdef METARESTORE
            sr = malloc(sizeof(statsrecord));
            memset(sr, 0, sizeof(statsrecord));
            p->data.ddata.stats = sr;
            p->data.ddata.quota = NULL;
#endif
            p->data.ddata.children = NULL;
            p->data.ddata.nlink = 2;
            p->data.ddata.elements = 0;
            break;
        case TYPE_FILE:
            p->data.fdata.length = 0;
            p->data.fdata.chunks = 0;
            p->data.fdata.chunktab = NULL;
            p->data.fdata.sessionids = NULL;
            break;
        case TYPE_SYMLINK:
            p->data.sdata.pleng = 0;
            p->data.sdata.path = NULL;
            break;
        case TYPE_BLOCKDEV:
        case TYPE_CHARDEV:
            p->data.rdev = 0;
    } ? end switch type ?
    p->parents = NULL;

    nodespos = NODEHASHPOS(p->id);
    p->next = nodehash[nodespos];
    nodehash[nodespos] = p;
    fsnodes_link(ts, node, p, nlen, name);
    return p;
} ? end fsnodes_create_node ?
```

系统模块的主要数据结构和算法介绍

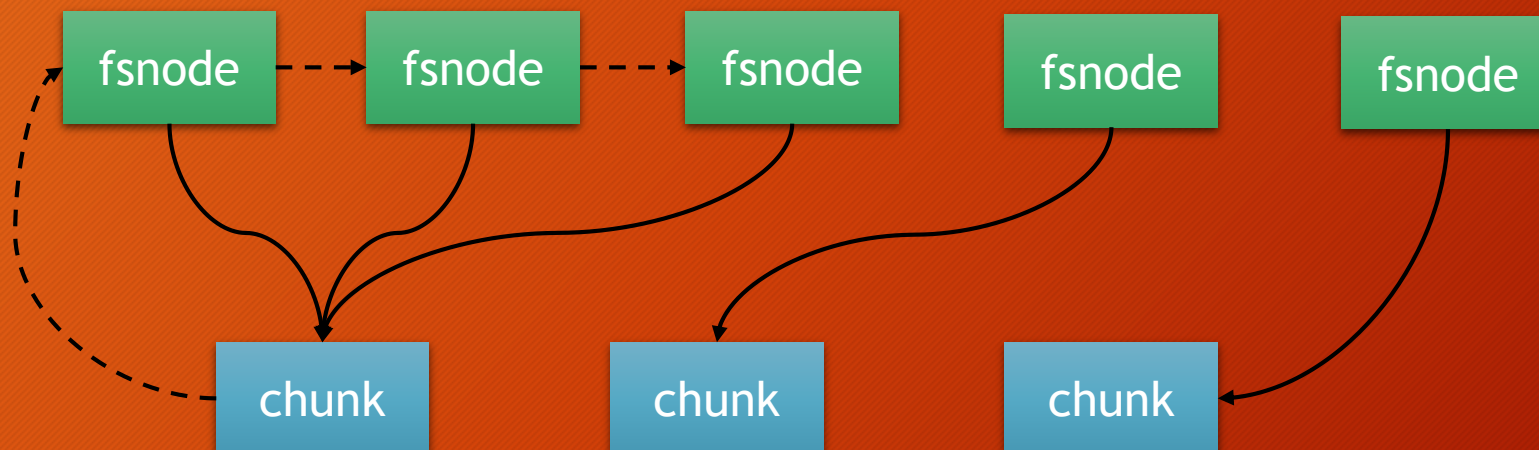
• Mfsmaster创建fsnode的过程（续）

```
00921: static inline void fsnodes_link(uint32_t ts, fsnode *parent, fsnode *child, uint16_t nleng, const uint8_t *name) {
00922:     fsedge *e;
00923:     #ifndef METARESTORE
00924:         statsrecord sr;
00925:     #endif
00926:     #ifdef EDGEHASH
00927:         uint32_t hpos;
00928:     #endif
00929:
00930:     e = malloc(sizeof(fsedge));
00931:     e->nleng = nleng;
00932:     e->name = malloc(nleng);
00933:     memcpy(e->name, name, nleng);
00934:     e->child = child;
00935:     e->parent = parent;
00936:     e->nextchild = parent->data.ddata.children;
00937:     if (e->nextchild) {
00938:         e->nextchild->prevchild = &(e->nextchild);
00939:     }
00940:     parent->data.ddata.children = e;
00941:     e->prevchild = &(parent->data.ddata.children);
00942:     e->nextparent = child->parents;
00943:     if (e->nextparent) {
00944:         e->nextparent->prevparent = &(e->nextparent);
00945:     }
00946:     child->parents = e;
00947:     e->prevparent = &(child->parents);
00948:     #ifdef EDGEHASH
00949:         hpos = EDGEHASHPOS(fsnodes_hash(parent->id, nleng, name));
00950:         e->next = edgelist[hpos];
00951:         if (e->next) {
00952:             e->next->prev = &(e->next);
00953:         }
00954:         edgelist[hpos] = e;
00955:         e->prev = &(edgelist[hpos]);
00956:     #endif
00957:
00958:     parent->data.ddata.elements++;
00959:     if (child->type == TYPE_DIRECTORY) {
00960:         parent->data.ddata.nlink++;
00961:     }
00962:     #ifndef METARESTORE
00963:         fsnodes_get_stats(child, &sr);
00964:         fsnodes_add_stats(parent, &sr);
00965:     #endif
00966:     if (ts > 0) {
00967:         parent->mtime = parent->ctime = ts;
00968:     }
00969: } ? end fsnodes_link ?
```

建立fsnode和fsedge之间指针关系

系统模块的主要数据结构和算法介绍

- Mfsmaster的chunk结构



系统模块的主要数据结构和算法介绍

- Mfsmaster的chunk结构（续）

```
00117: typedef struct chunk {
00118:     uint64_t chunkid;
00119:     uint32_t version;
00120:     uint8_t goal;
00121: #ifndef METARESTORE
00122:     uint8_t allvalidcopies;
00123:     uint8_t regularvalidcopies;
00124:     uint8_t needverincrease:1;
00125:     uint8_t interrupted:1;
00126:     uint8_t operation:4;
00127: #endif
00128:     uint32_t lockedto;
00129: #ifndef METARESTORE
00130: // uint32_t lockedby;
00131:     slist *slisthead;
00132: // bcddata *bestchunk;
00133: #endif
00134:     flist *flisthead;
00135:     struct chunk *next;
00136: } ? end chunk ? chunk;
```

Chunk server list which is aligned with goal!

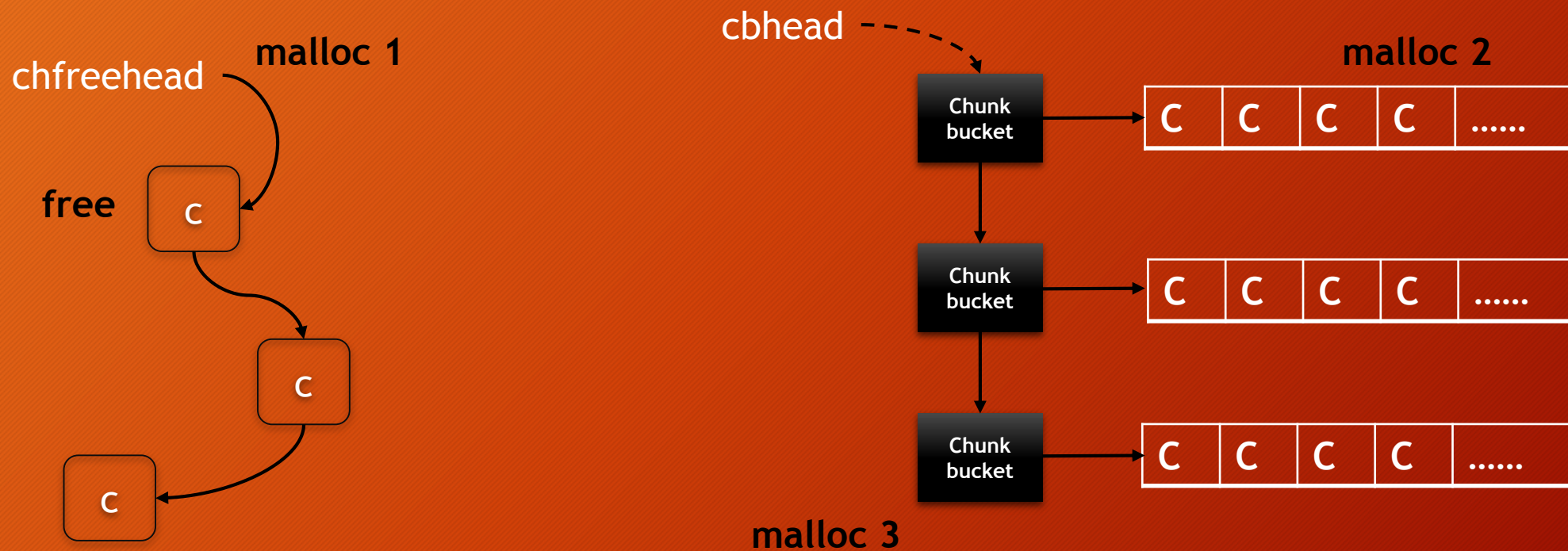
```
01009:     if (servcount < goal) {
01010:         c->allvalidcopies = servcount;
01011:         c->regularvalidcopies = servcount;
01012:     } else {
01013:         c->allvalidcopies = goal;
01014:         c->regularvalidcopies = goal;
01015:     }
01016:     for (i=0 ; i<c->allvalidcopies ; i++) {
01017:         s = slist_malloc();
01018:         s->ptr = ptrs[i];
01019:         s->valid = BUSY;
01020:         s->version = c->version;
01021:         s->next = c->slisthead;
01022:         c->slisthead = s;
01023:         matocsserv_send_createchunk(s->ptr, c->chunkid, c->version);
01024:     }
```

Create chunk

node list

系统模块的主要数据结构和算法介绍

- Mfsmaster的chunk结构（续）
 - 内存管理-预分配技术



系统模块的主要数据结构和算法介绍

- 内存管理-预分配技术

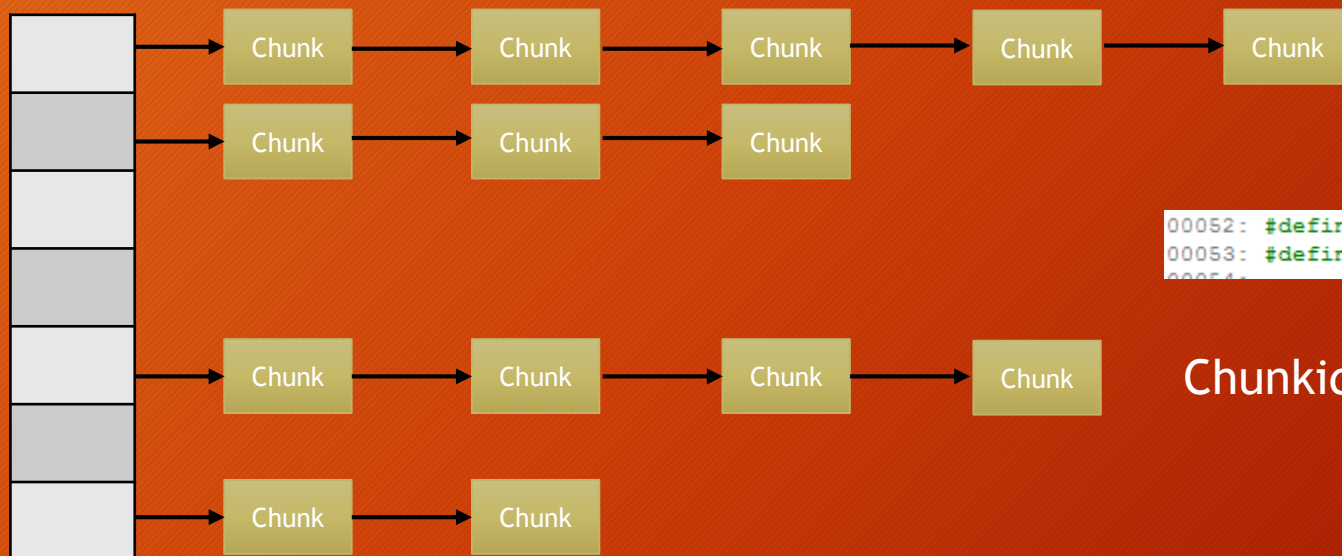
```
00138: #ifdef USE_CHUNK_BUCKETS
00139: #define CHUNK_BUCKET_SIZE 20000
00140: typedef struct _chunk_bucket {
00141:     chunk bucket[CHUNK_BUCKET_SIZE];
00142:     uint32_t firstfree;
00143:     struct _chunk_bucket *next;
00144: } chunk_bucket;
00145:
00146: static chunk_bucket *cbhead = NULL;
00147: static chunk *chfreehead = NULL;
00148: #endif /* USE_CHUNK_BUCKETS */
00149:
00150: static chunk *chunkhash[HASHSIZE];
```

```
00308: static inline chunk* chunk_malloc() {
00309:     chunk_bucket *cb;
00310:     chunk *ret;
00311:     if (chfreehead) {
00312:         ret = chfreehead;
00313:         chfreehead = ret->next;
00314:         return ret;
00315:     }
00316:     if (cbhead==NULL || cbhead->firstfree==CHUNK_BUCKET_SIZE) {
00317:         cb = (chunk_bucket*)malloc(sizeof(chunk_bucket));
00318:         cb->next = cbhead;
00319:         cb->firstfree = 0;
00320:         cbhead = cb;
00321:     }
00322:     ret = (cbhead->bucket)+(cbhead->firstfree);
00323:     cbhead->firstfree++;
00324:     return ret;
00325: }
00326:
00327: static inline void chunk_free(chunk *p) {
00328:     p->next = chfreehead;
00329:     chfreehead = p;
00330: }
```


系统模块的主要数据结构和算法介绍

- Mfsmaster的chunk结构（续）
 - Chunk管理（快速定位）

Chunk hash



```
00052: #define HASHSIZE 65536
00053: #define HASHPOS(chunkid) (((uint32_t) chunkid) & 0xFFFF)
00054:
```

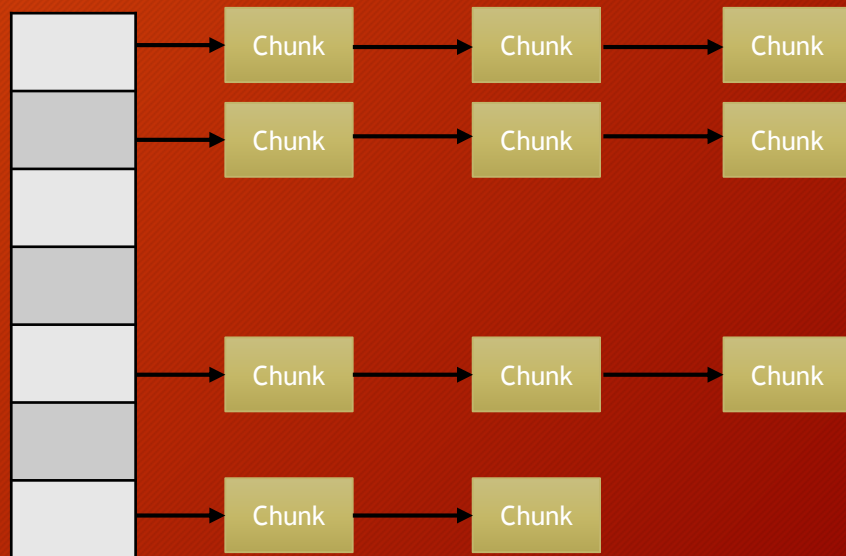
Chunkid (uint64_t) 是一个从1开始的自增id

系统模块的主要数据结构和算法介绍

- Mfschunkserver的chunk结构

```
00121: typedef struct chunk {
00122:     char *filename;
00123:     uint64_t chunkid;
00124:     struct folder *owner;
00125:     uint32_t version;
00126:     uint16_t blocks;
00127:     uint16_t crcrefcount;
00128:     uint8_t opensteps;
00129:     uint8_t crcsteps;
00130:     uint8_t crcchanged;
00131: #ifdef _THREAD_SAFE
00132: #define CH_AVAIL 0
00133: #define CH_LOCKED 1
00134: #define CH_DELETED 2
00135: #define CH_TOBEDELETED 3
00136:     uint8_t state; // CH_AVAIL, CH_LOCKED, CH_DELETED
00137:     cntcond *ccond;
00138: #endif
00139:     uint8_t *crc;
00140:     int fd;
00141:
00142: #ifdef PRESERVE_BLOCK
00143:     uint8_t *block;
00144:     uint16_t blockno; // 0xFFFF == invalid
00145:     uint8_t blocksteps;
00146: #endif
00147: #ifdef _THREAD_SAFE
00148:     uint32_t testtime; // at start use max(ctime, mtime) then every operation set it to current time
00149:     struct chunk *testnext, **testprev;
00150: #endif
00151:     struct chunk *next;
00152: } ? end chunk ? chunk;
```

Hash table



系统模块的主要数据结构和算法介绍

- Mfschunkserver的chunk结构（续）

```
[root@gd2cwkfvm118 data]# ls
00 07 0E 15 1C 23 2A 31 38 3F 46 4D 54 5B 62 69 70 77 7E 85 8C 93 9A A1 A8 AF B6 BD C4 CB D2 D9 E0 E7 EE F5 FC
01 08 0F 16 1D 24 2B 32 39 40 47 4E 55 5C 63 6A 71 78 7F 86 8D 94 9B A2 A9 B0 B7 BE C5 CC D3 DA E1 E8 EF F6 FE
02 09 10 17 1E 25 2C 33 3A 41 48 4F 56 5D 64 6B 72 79 80 87 8E 95 9C A3 AA B1 B8 BF C6 CD D4 DB E2 E9 F0 F7 FD
03 0A 11 18 1F 26 2D 34 3B 42 49 50 57 5E 65 6C 73 7A 81 88 8F 96 9D A4 AB B2 B9 C0 C7 CE D5 DC E3 EA F1 F8 FF
04 0B 12 19 20 27 2E 35 3C 43 4A 51 58 5F 66 6D 74 7B 82 89 90 97 9E A5 AC B3 BA C1 C8 CF D6 DD E4 EB F2 F9
05 0C 13 1A 21 28 2F 36 3D 44 4B 52 59 60 67 6E 75 7C 83 8A 91 98 9F A6 AD B4 BB C2 C9 D0 D7 DE E5 EC F3 FA
06 0D 14 1B 22 29 30 37 3E 45 4C 53 5A 61 68 6F 76 7D 84 8B 92 99 A0 A7 AE B5 BC C3 CA D1 D8 DF E6 ED F4 FB
```

```
chunk_0000000000000009_00000001.mfs
```

Chunk folder
 $16 \times 16 = 256$

Chunk data

生成规则

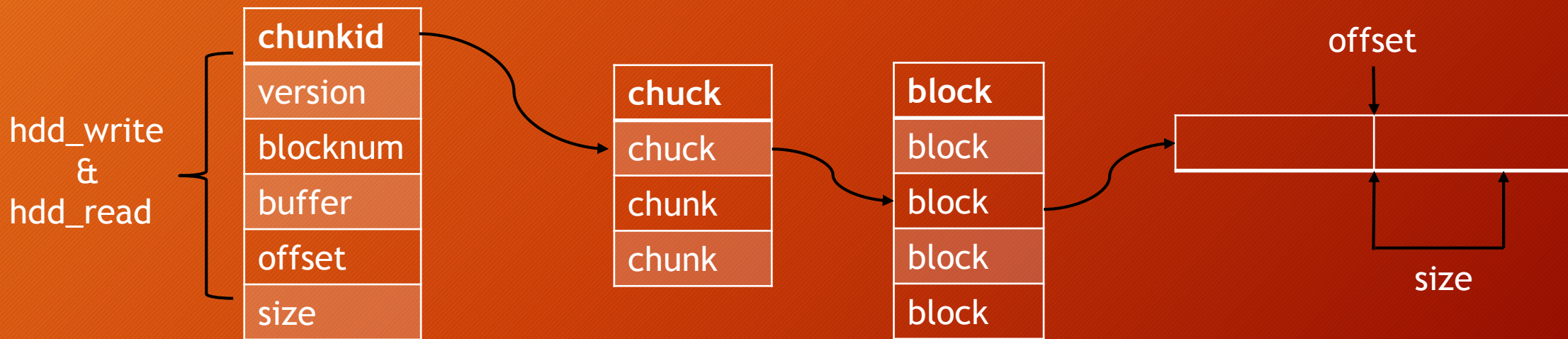
```
sprintf(c->filename+leng, "%02X/chunk_%016"PRIx64"_%08"PRIx32".mfs", (unsigned int)(chunkid&255), chunkid, version);
```


系统模块的主要数据结构和算法介绍

- Mfschunkserver的block

- 一个block的大小为0x10000，Blocknum为block的编号。

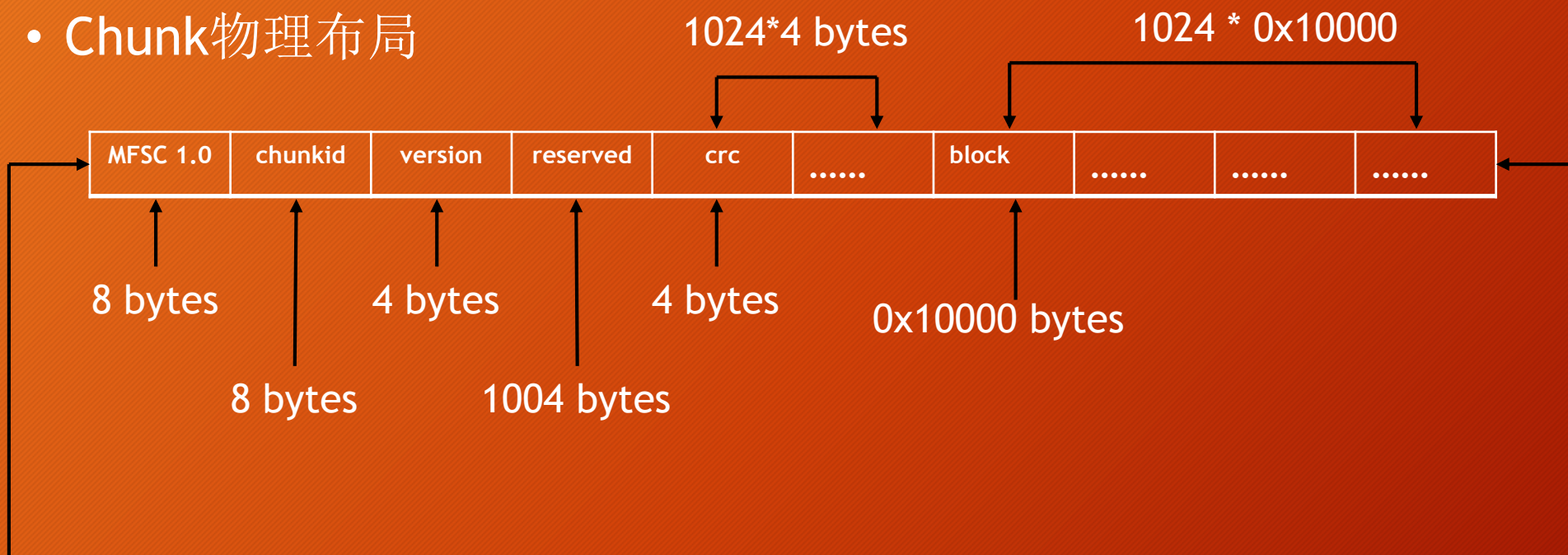
```
lseek(c->fd, CHUNKHDRSIZE + (((uint32_t)blocknum) << 16), SEEK_SET);  
ret = write(c->fd, buffer, 0x10000);
```



Max size: 0x400=1024个block

系统模块的主要数据结构和算法介绍

- Chunk物理布局



$$(1024 + 1024 * 4) + 1024 * 0x10000$$

Header + Data

系统模块的主要数据结构和算法介绍

- 读和写都是以block为单位进行
 - 举例

```
blocknum = (eptr->offset)>>16;  
blockoffset = (eptr->offset)&0xFFFF;  
if (((eptr->offset+eptr->size-1)>>16) == blocknum) {    // last block  
    size = eptr->size;  
} else {  
    size = 0x10000-blockoffset;  
}
```

read_block函数对读取size进行调整

系统模块的主要数据结构和算法介绍

- Mfs确保data copy的数据一致性，进程后台timer，做data copy的一致性校验，定时向chunk server发送删除无效chunk的消息，确保data copy是一致的。
 - calculate number of valid and invalid copies & verify
 - check number of copies
 - delete invalid copies
 - return if chunk is during some operation
 - check busy count
 - delete unused chunk
 - if chunk has too many copies then delete some of them
 - if chunk has one copy on each server and some of them have status TODEL then delete one of it
 - if chunk has number of copies less than goal then make another copy of this chunk
 - if there is too big difference between chunk servers then make copy of chunk from server with biggest disk usage on server with lowest disk usage

```
void chunk_do_jobs(chunk *_c, uint16_t scount, double minusage, double maxusage) {
```

系统模块的主要数据结构和算法介绍

- Mfs确保data copy的数据一致性（续）

```
if (c->flisthead==NULL) {
// syslog(LOG_WARNING,"unused - delete");
if (delcount<TmpMaxDel) {
for (s=c->slisthead ; s ; s=s->next) {
if (s->valid==VALID || s->valid==TDVALID) {
if (s->valid==TDVALID) {
chunk_state_change(c->goal,c->goal,c->allvalidcopies,c->allvalidcopies-1,c->regularvalidcopies,c->regularvalidcopies);
c->allvalidcopies--;
} else {
chunk_state_change(c->goal,c->goal,c->allvalidcopies,c->allvalidcopies-1,c->regularvalidcopies,c->regularvalidcopies-1);
c->allvalidcopies--;
c->regularvalidcopies--;
}
c->needverincrease=1;
s->valid = DEL;
stats.deletions++;
matocsserv_send_deletechunk(s->ptr,c->chunkid,c->version);
delcount++;
inforec.done.del_unused++;
}
}
} ? end if delcount<TmpMaxDel ? else {
for (s=c->slisthead ; s ; s=s->next) {
if (s->valid==VALID || s->valid==TDVALID) {
inforec.notdone.del_unused++;
}
}
}
return ;
} ? end if c->flisthead==NULL ?
```

Master

```
static void hdd_chunk_delete(chunk *c) {
folder *f;
pthread_mutex_lock(&hashlock);
f = c->owner;
if (c->ccond) {
c->state = CH_DELETED;
// printf("wake up one thread waiting for DELETED chunk: %"PR
// printbacktrace();
pthread_cond_signal(&(c->ccond->cond));
} else {
hdd_chunk_remove(c);
}
pthread_mutex_unlock(&hashlock);
pthread_mutex_lock(&folderlock);
f->chunkcount--;
f->needrefresh = 1;
pthread_mutex_unlock(&folderlock);
}
#else
static void hdd_chunk_delete(chunk *c) {
c->owner->chunkcount--;
c->owner->needrefresh = 1;
hdd_chunk_remove(c);
}
#endif
```

Chunk server

系统模块的主要数据结构和算法介绍

- Mfschunkserver进程后台起了一个timer，定时对chunk文件进行有效性验证，步骤如下：
 - 根据chunk id查找内存中是否有分配相应的chunk数据结构。
 - Chunk的版本
 - 是否能够成功的打开chunk文件
 - 对每个block的数据进行CRC（32位）校验。
 - 是否能够关闭chunk文件
 - 将damaged的chunk缓存到全局链表damagedchunks，上报给master。

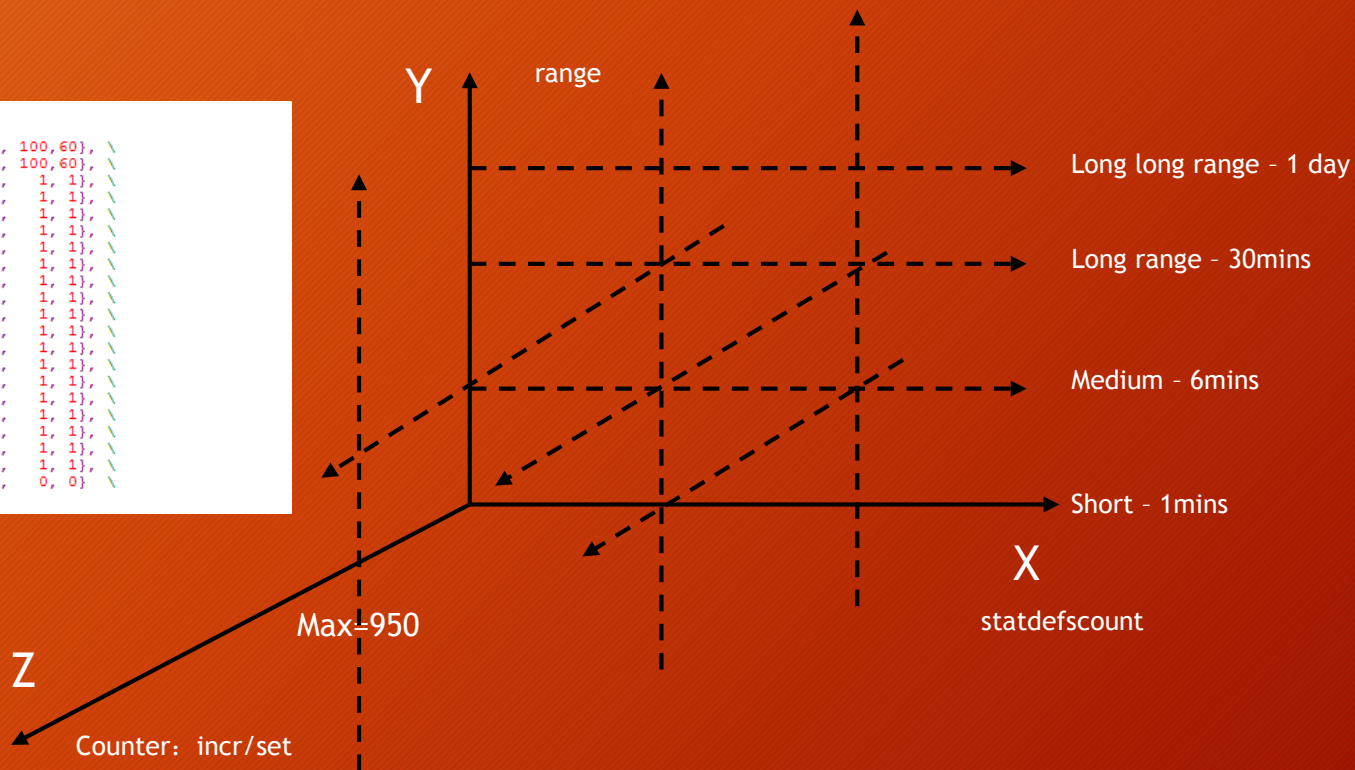
```
00168: typedef struct folder {
00169:     char *path;
00170:     unsigned int needrefresh:1;
00171:     unsigned int todel:1;
00172:     unsigned int damaged:1;
00173:     uint64_t leavefree;
00174:     uint64_t avail;
00175:     uint64_t total;
00176:     hddstats cstat;
00177:     hddstats stats[STATSHISTORY];
00178:     uint32_t statspos;
00179:     ioerror lasterrtab[LASTERRSIZE];
00180:     uint32_t chunkcount;
00181:     uint32_t lasterrindx;
00182:     uint32_t lastrefresh;
00183:     dev_t devid;
00184:     ino_t lockinode;
00185:     double carry;
00186: #ifdef _THREAD_SAFE
00187:     pthread_mutex_t searchmutex;
00188:     struct chunk *testhead,**testtail;
00189: #endif
00190:     struct folder *next;
00191: } ? end folder ? folder;
```


系统模块的主要数据结构和算法介绍

• Mfsmaster的Chart data

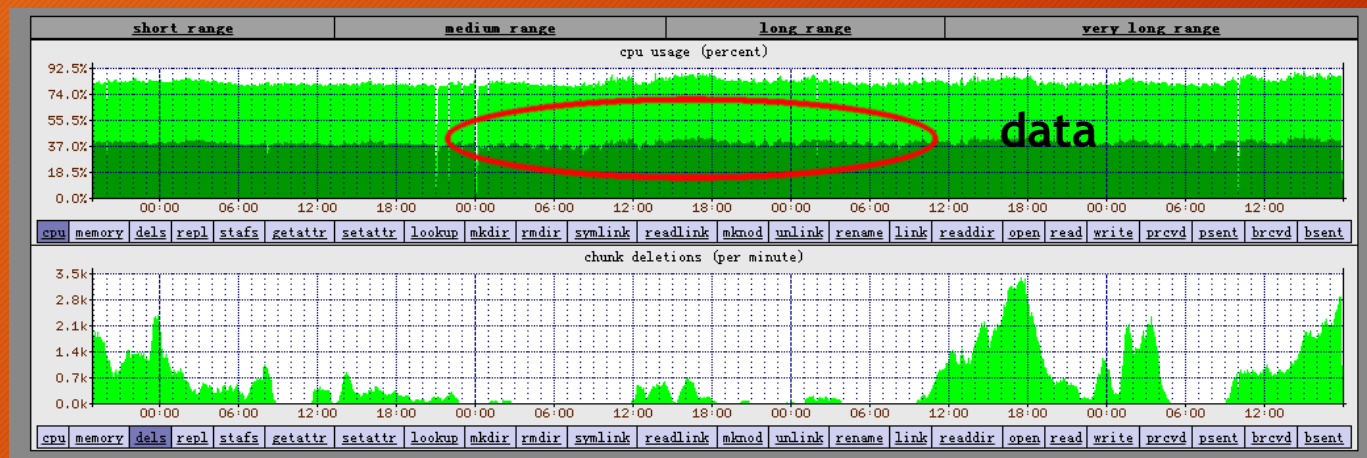
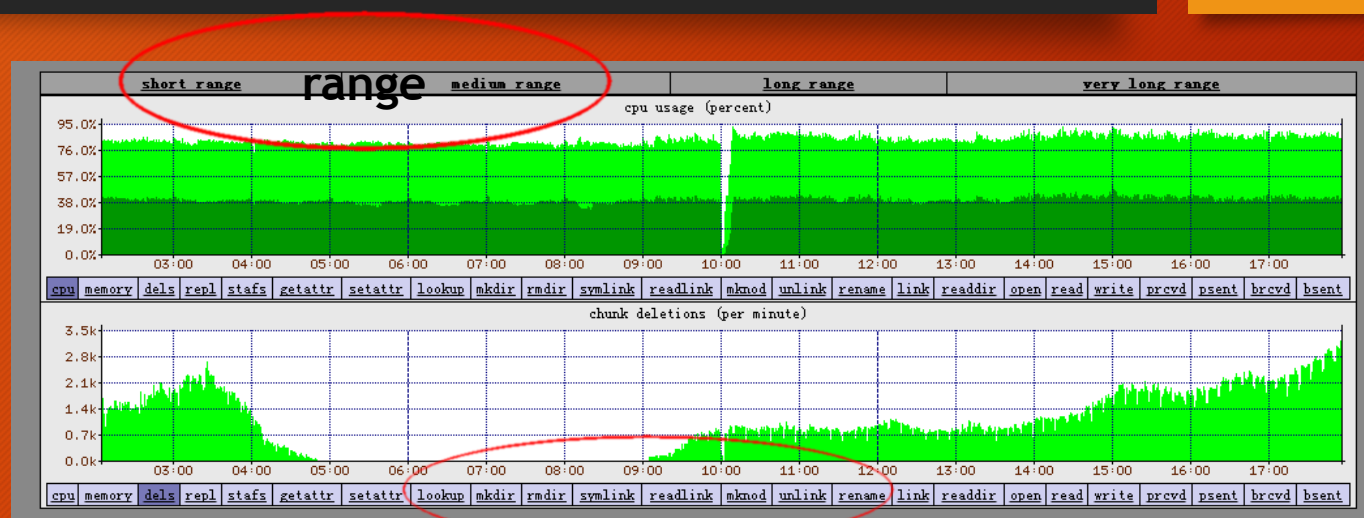
```
00062: /* name, join mode, percent, scale, multiplier, divisor */
00063: #define STATDEFS { \
00064:     {"ucpu", CHARTS_MODE_ADD, 1, CHARTS_SCALE_MICRO, 100, 60}, \
00065:     {"scpu", CHARTS_MODE_ADD, 1, CHARTS_SCALE_MICRO, 100, 60}, \
00066:     {"delete", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00067:     {"replicate", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00068:     {"statfs", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00069:     {"getattr", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00070:     {"setattr", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00071:     {"lookup", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00072:     {"mkdir", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00073:     {"rmdir", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00074:     {"symlink", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00075:     {"readlink", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00076:     {"mknod", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00077:     {"unlink", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00078:     {"rename", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00079:     {"link", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00080:     {"readdir", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00081:     {"open", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00082:     {"read", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00083:     {"write", CHARTS_MODE_ADD, 0, CHARTS_SCALE_NONE, 1, 1}, \
00084:     {NULL, 0, 0, 0, 0, 0} \
00085: };
```

```
00081:
00082: typedef uint64_t stat_record[RANGES][LENG];
00083:
00084: static stat_record *series;
```



每分钟flush一次-> stats.mfs、csstats.mfs

系统模块的主要数据结构和算法介绍



operation

系统模块的主要数据结构和算法介绍

- Mfschunkserver的Chart data

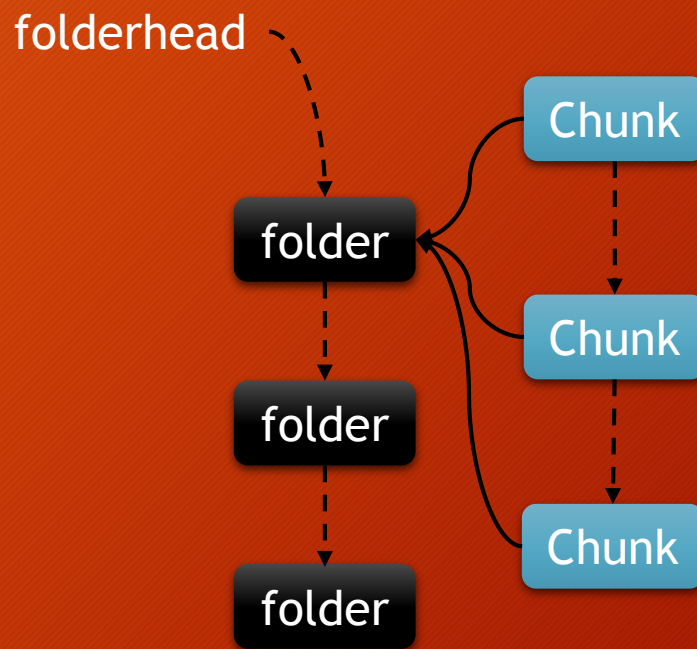
```
00075: /* name, join mode, percent, scale, multiplier, divisor */
00076: #define STATDEFS { \
00077:     {"ucpu"          ,CHARTS_MODE_ADD,1,CHARTS_SCALE_MICRO, 100,60}, \
00078:     {"scpu"          ,CHARTS_MODE_ADD,1,CHARTS_SCALE_MICRO, 100,60}, \
00079:     {"masterin"       ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00080:     {"masterout"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00081:     {"csconnin"       ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00082:     {"csconnout"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00083:     {"csservin"       ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00084:     {"csservout"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,8000,60}, \
00085:     {"bytesr"         ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,1000,60}, \
00086:     {"bytesw"         ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,1000,60}, \
00087:     {"llopr"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00088:     {"llopw"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00089:     {"databytesr"     ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,1000,60}, \
00090:     {"databytesw"     ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MILI ,1000,60}, \
00091:     {"datallopr"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00092:     {"datallopw"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00093:     {"hlopr"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00094:     {"hlopw"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00095:     {"rtime"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MICRO , 1,60}, \
00096:     {"wtime"          ,CHARTS_MODE_ADD,0,CHARTS_SCALE_MICRO , 1,60}, \
00097:     {"repl"           ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00098:     {"create"         ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00099:     {"delete"         ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00100:     {"version"        ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00101:     {"duplicate"      ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00102:     {"truncate"       ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00103:     {"duptrunc"       ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00104:     {"test"           ,CHARTS_MODE_ADD,0,CHARTS_SCALE_NONE  , 1, 1}, \
00105:     {"chunkiojobs"    ,CHARTS_MODE_MAX,0,CHARTS_SCALE_NONE  , 1, 1}, \
00106:     {"chunkopjobs"    ,CHARTS_MODE_MAX,0,CHARTS_SCALE_NONE  , 1, 1}, \
00107:     {NULL             , 0, 0, 0, 0}, \
00108: };
```

数据存储结构，同mfsmaster一样

系统模块的主要数据结构和算法介绍

- Mfschunkserver的folder数据结构

```
00168: typedef struct folder {
00169:     char *path;
00170:     unsigned int needrefresh:1;
00171:     unsigned int todel:1;
00172:     unsigned int damaged:1;
00173:     uint64_t leavefree;
00174:     uint64_t avail;
00175:     uint64_t total;
00176:     hddstats cstat;
00177:     hddstats stats[STATSHISTORY];
00178:     uint32_t statspos;
00179:     ioerror lasterrtab[LASTERRSIZE];
00180:     uint32_t chunkcount;
00181:     uint32_t lasterrindx;
00182:     uint32_t lastrefresh;
00183:     dev_t devid;
00184:     ino_t lockinode;
00185:     double carry;
00186: #ifdef _THREAD_SAFE
00187:     pthread_t scanthread;
00188:     struct chunk *testhead,**testtail;
00189: #endif
00190:     struct folder *next;
00191: } ? end folder ? folder;
```



系统模块的主要数据结构和算法介绍

- Mfschunkserver在create_chunk的时候，如何选择合适的分区？

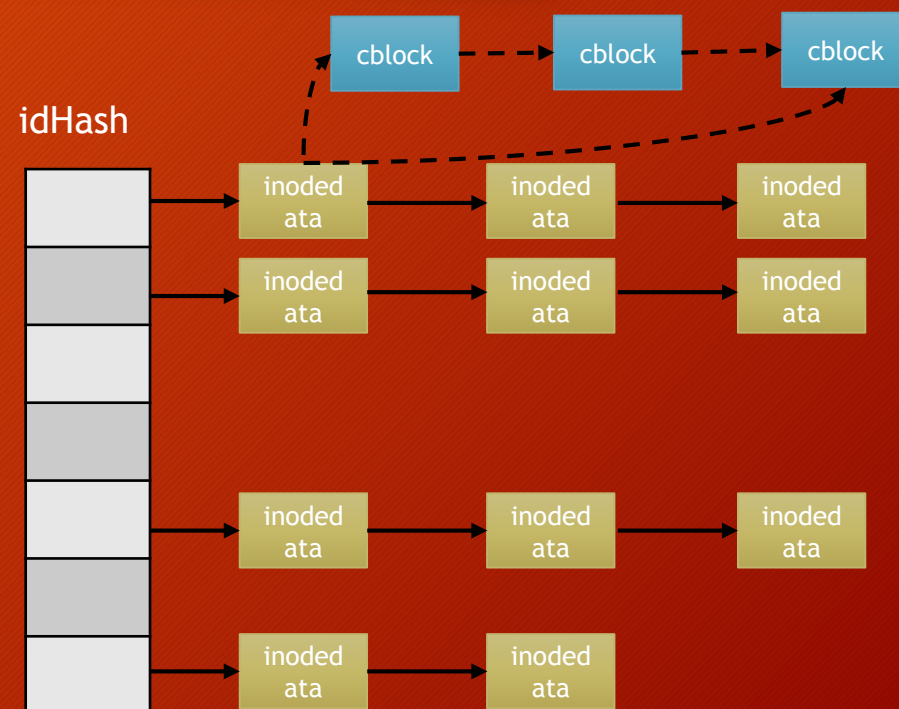
```
01045: static inline folder* hdd_getfolder() {
01046:     folder *f,*bf;
01047:     double maxcarry;
01048:     double minavail,maxavail;
01049:     double s,d;
01050:     double pavail;
01051:     int ok;
01052:     // uint64_t minavail;
01053:
01054:     minavail=0.0;
01055:     maxavail=0.0;
01056:     maxcarry=1.0;
01057:     bf=NULL;
01058:     ok=0;
01059:     for (f=folderhead ; f ; f=f->next) {
01060:         if (f->damaged || f->todel || f->total==0 || f->avail==0) {
01061:             continue;
01062:         }
01063:         if (f->carry >= maxcarry) {
01064:             maxcarry = f->carry;
01065:             bf=f;
01066:         }
01067:         pavail = (double) (f->avail)/(double) (f->total);
01068:         if (ok==0 || minavail>pavail) {
01069:             minavail=pavail;
01070:             ok=1;
01071:         }
01072:         if (pavail>maxavail) {
01073:             maxavail=pavail;
01074:         }
01075:     }
01076:     if (bf) {
01077:         bf->carry-=1.0;
01078:         return bf;
01079:     }
01080:     if (maxavail==0.0) { // no space
01081:         return NULL;
01082:     }
01083: }
```

```
01083:     if (maxavail<0.01) {
01084:         s=0.0;
01085:     } else {
01086:         s=minavail*0.8;
01087:         if (s<0.01) {
01088:             s=0.01;
01089:         }
01090:     }
01091:     d = maxavail-s;
01092:     maxcarry=1.0;
01093:     for (f=folderhead ; f ; f=f->next) {
01094:         if (f->damaged || f->todel || f->total==0 || f->avail==0) {
01095:             continue;
01096:         }
01097:         pavail = (double) (f->avail)/(double) (f->total);
01098:         if (pavail>s) {
01099:             f->carry += ((pavail-s)/d);
01100:         }
01101:         if (f->carry >= maxcarry) {
01102:             maxcarry = f->carry;
01103:             bf=f;
01104:         }
01105:     }
01106:     if (bf) { // should be always true
01107:         bf->carry-=1.0;
01108:     }
01109:     return bf;
01110: } // end hdd_getfolder ?
```

系统模块的主要数据结构和算法介绍

- Mfsmount的inode

```
00043: typedef struct cblock_s {
00044:     uint8_t data[65536];    // modified only when writeid==0
00045:     uint16_t chindx;        // chunk number
00046:     uint16_t pos;           // block in chunk (0...1023) - never modified
00047:     uint32_t writeid;        // 0 = not sent, >0 = block was sent (modified and accessed only when wchunk is locked)
00048:     uint32_t from;          // first filled byte in data (modified only when writeid==0)
00049:     uint32_t to;            // first not used byte in data (modified only when writeid==0)
00050:     struct cblock_s *next, *prev;
00051: } cblock;
00052:
00053: typedef struct inodedata_s {
00054:     uint32_t inode;
00055:     uint64_t maxfleng;
00056:     uint32_t cacheblocks;
00057:     int status;
00058:     uint16_t flushwaiting;
00059:     uint16_t writewaiting;
00060:     uint16_t cachewaiting;
00061:     uint16_t lcmt;
00062:     uint32_t trycnt;
00063:     uint8_t waitingworker;
00064:     uint8_t inqueue;
00065:     int pipe[2];
00066:     cblock *datachainhead, *datachaintail;
00067:     pthread_cond_t flushcond;    // wait for inqueue==0 (flush)
00068:     pthread_cond_t writecond;    // wait for flushwaiting==0 (write)
00069:     pthread_cond_t cachecond;    // wait for cache blocks
00070:     struct inodedata_s *next;
00071: } inodedata;
```



1. 1024 BLOCKS IN CHUNK
2. THE SIZE OF BLOCK IS 65532
3. IN EACH HDD_WRITE, THE SIZE IS 16 BYTES

系统模块的主要数据结构和算法介绍

- 写文件系统，mfsmount进程将从mfsmaster进程获取mfschunkserver的最佳路由表。将要写入文件系统的数据根据路由表进行数据路由，mfschunkserver执行写操作。

```
01521:     for (s=c->slisthead ; s ; s=s->next) {
01522:         if (s->valid!=INVALID && s->valid!=DEL) {
01523:             if (cnt<100 && matocsserv_getlocation(s->ptr,&(lstab[cnt].ip),&(lstab[cnt].port))==0) {
01524:                 lstab[cnt].dist = (lstab[cnt].ip==cuip)?0:1;    // in the future prepare more sophisticated distance function
01525:                 lstab[cnt].rnd = rndu32();
01526:                 cnt++;
01527:             }
01528:             //         sptr[cnt++]=s->ptr;
01529:         }
01530:     }
01531:     qsort(lstab,cnt,sizeof(locsort),chunk_locsort_cmp);
01532:     wptr = loc;
```

系统模块的主要数据结构和算法介绍

- Mfsmaster访问控制数据结构

```
00019: typedef struct _acl {
00020:     uint32_t pleng;
00021:     const uint8_t *path;    // without '/' at the begin and at the end
00022:     uint32_t fromip,toip;
00023:     uint32_t minversion;
00024:     uint8_t passworddigest[16];
00025:     unsigned alldirs:1;
00026:     unsigned needpassword:1;
00027:     unsigned meta:1;
00028:     unsigned rootredefined:1;
00029:     // unsigned old:1;
00030:     uint8_t sesflags;
00031:     uint32_t rootuid;
00032:     uint32_t rootgid;
00033:     uint32_t mapalluid;
00034:     uint32_t mapallgid;
00035:     struct _acl *next;
00036: } acl;
```

- 进程刚启动的时候，会初始化访问控制链表

```
00789: int acl_init(FILE *msgfd) {
00790:     ExportsFileName = cfg_getstr("EXPORTS_FILENAME",ETC_PATH "/mfsexports.cfg");
00791:     acl_records = NULL;
00792:     acl_loadexports(msgfd);
00793:     main_reloadregister(acl_reloadexports);
00794:     return 0;
00795: }
```

- Mfsmount向mfsmaster发送注册消息，会进行访问权限的校验
 - IP地址范围校验
 - 目录校验
 - 密码校验
 - 访问权限校验read/write/gid/uid

系统模块的主要数据结构和算法介绍

- Mfsmaster会话数据结构

- 当有新的mfsmount向master注册的时候，会创建一个新的会话
- 将会话信息存储在本地，用于master进程恢复

```
00079: typedef struct session {
00080:     uint32_t sessionid;
00081:     char *info;
00082:     uint32_t peerip;
00083:     uint8_t newsession;
00084:     uint8_t sesflags;
00085:     uint32_t rootuid;
00086:     uint32_t rootgid;
00087:     uint32_t mapalluid;
00088:     uint32_t mapallgid;
00089:     uint32_t rootinode;
00090:     uint32_t disconnected; // 0 = connected ; other = disconnection timestamp
00091:     uint32_t nsocks; // >0 - connected (number of active connections) ; 0 - not connected
00092:     uint32_t currentopstats[16];
00093:     uint32_t lasthouropstats[16];
00094:     filelist *openedfiles;
00095:     struct session *next;
00096: } session;
```


系统模块的主要数据结构和算法介绍

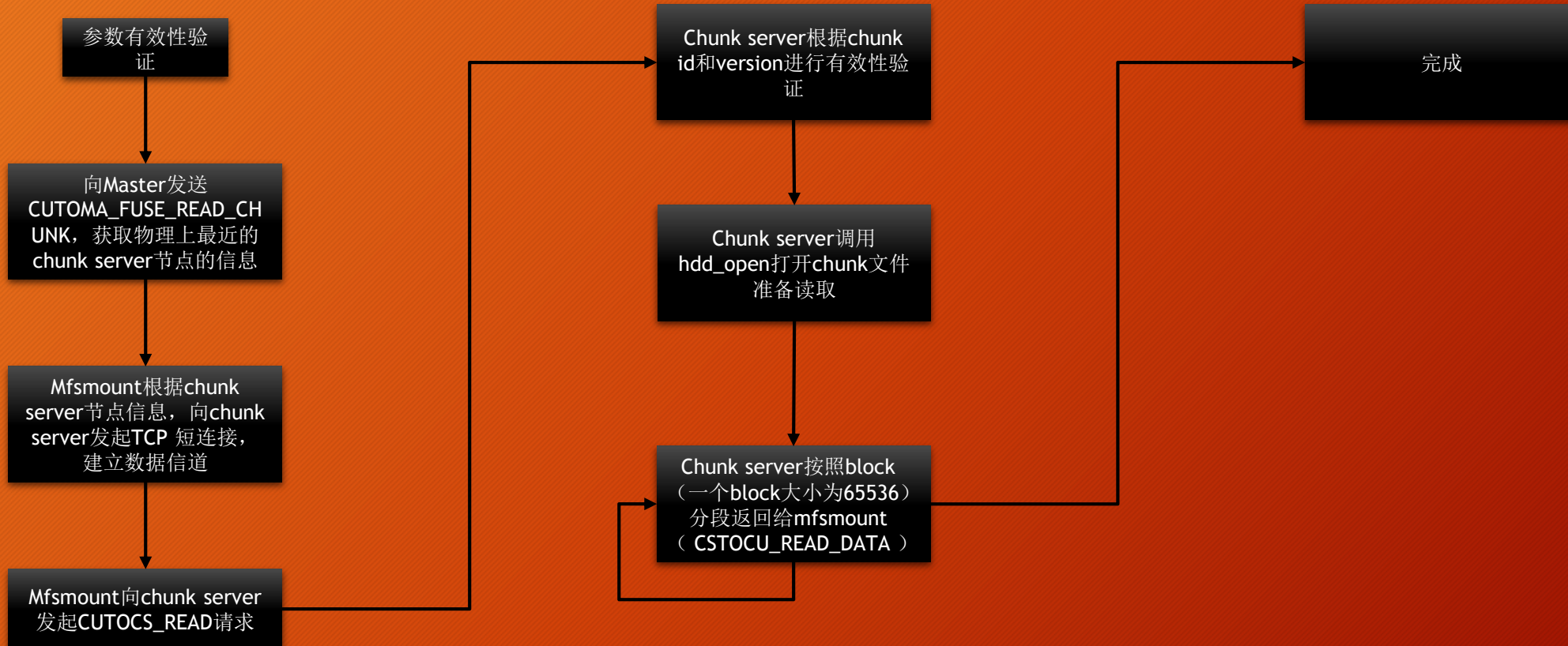
- Mfsmetalogger

- Mfsmetalogger进程和mfsmaster建立TCP长连接
- 从mfsmaster下载metadata数据，存于metadata_ml.mfs
- 从mfsmaster下载session数据，存于session_ml.mfs

```
00729:  main_timeregister(TIMEMODE_RUNONCE, MetaDLFreq*3600, 630, masterconn_metadownloadinit);  
00730:  main_timeregister(TIMEMODE_RUNONCE, 60, 0, masterconn_sessionsdownloadinit);
```

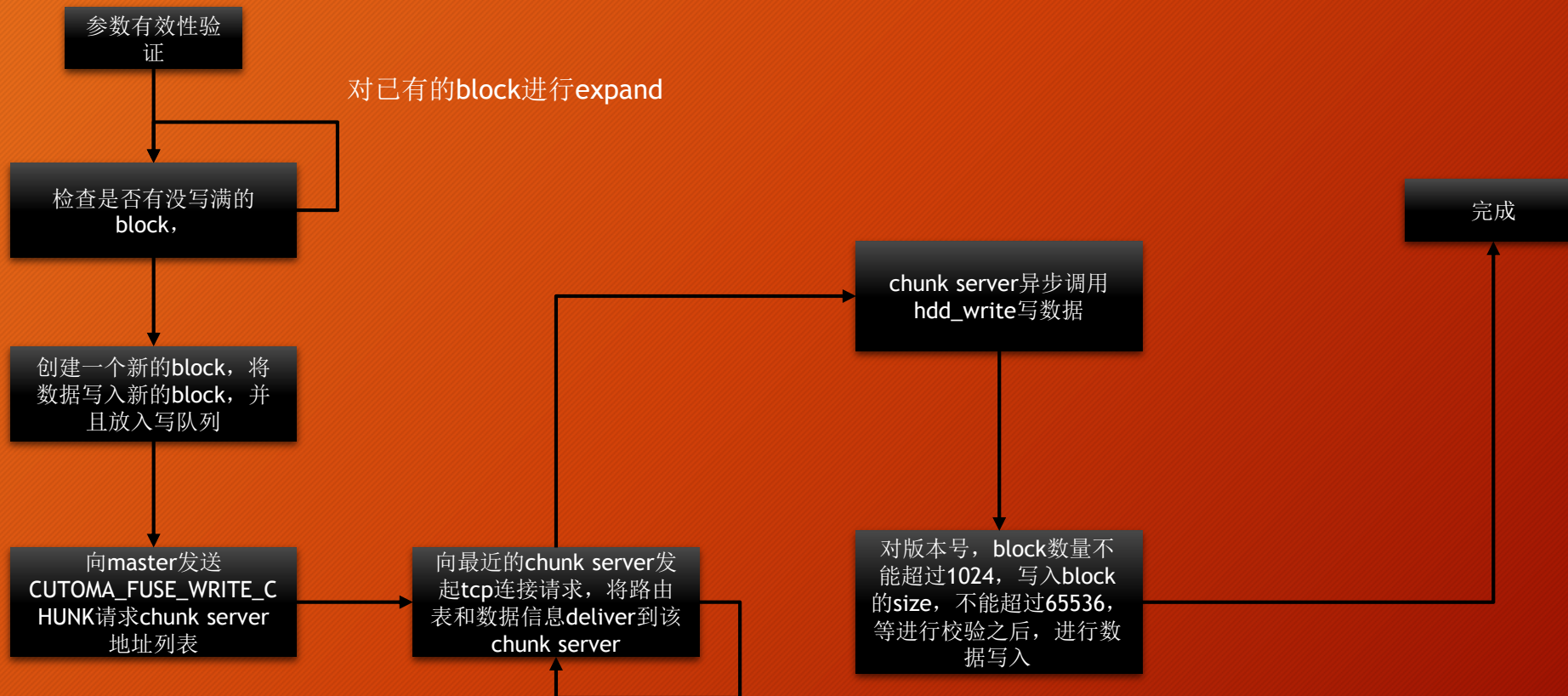
系统主要的调用流程

```
void mfs_read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off, struct fuse_file_info *fi)
```



系统主要的调用流程

```
void mfs_write(fuse_req_t req, fuse_ino_t ino, const char *buf, size_t size, off_t off, struct fuse_file_info *fi)
```

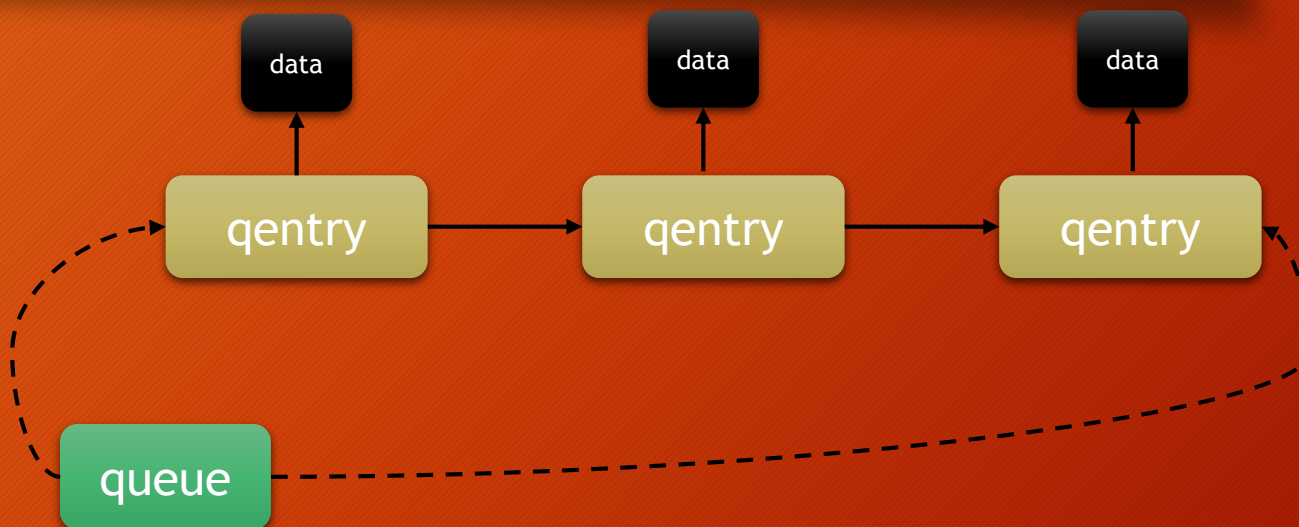


设计技巧的介绍

- 进程内队列通信

```
00027: typedef struct _qentry {
00028:     uint32_t id;
00029:     uint32_t op;
00030:     uint8_t *data;
00031:     uint32_t leng;
00032:     struct _qentry *next;
00033: } qentry;
00034:
00035: typedef struct _queue {
00036:     qentry *head, **tail;
00037:     void *semfree, *semfull;
00038:     pthread_mutex_t lock;
00039: } queue;
```

```
00025: typedef struct _semaphore {
00026:     uint32_t count;
00027:     pthread_mutex_t lock;
00028:     pthread_cond_t cond;
00029: } semaphore;
00030:
```



线程安全，互斥锁
线程同步，信号量

信号量，条件变量，线程同步

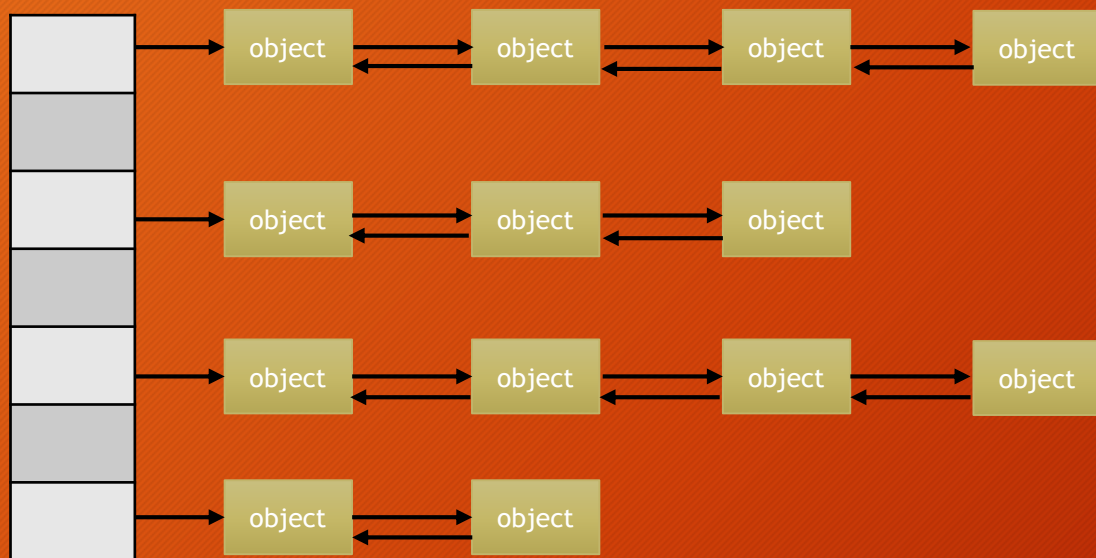
应用：

1. Mfsmount进程写block。
2. Mfschunkserver进程后台模式

设计技巧的介绍

- 内存预分配技术

Object Pool



freelist



Delay free

Advanced

Global MM by Buddy system

基于我们的应用，可以做的一些改进

- 问题

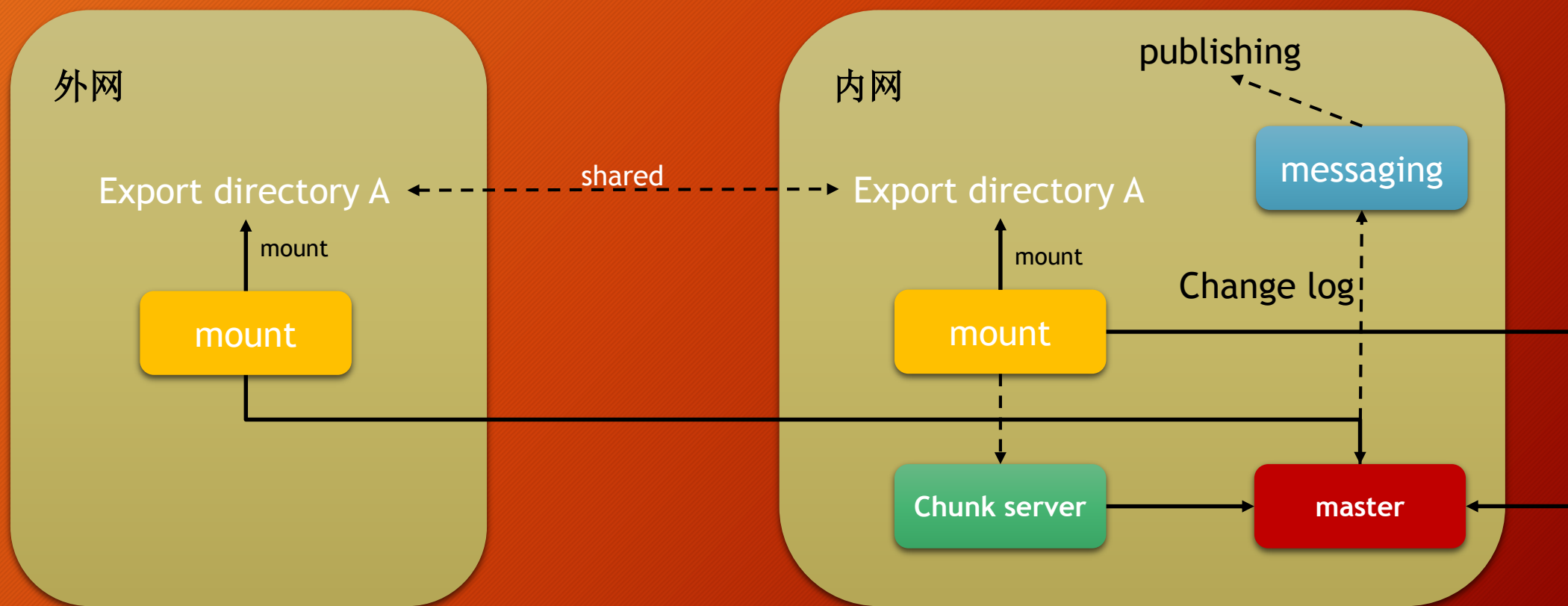
- 财务系统有大量的账单文件，每个月增长量为3万个，保留18个月，每个文件大小平均为8M左右（账单明细），几十K（账单详情）。
- 需要内网和外网共享，提供给供应商在线下载账单文件。
- 考虑到信息安全，外网的账单只能部分和零时共享到外网，提供供应商下载。

- 目前的做法

- 内外网共享文件系统，通过MFS。
- 供应商需要下载的时候，才将内网的账单copy到外网的共享目录，待供应商下载完毕在删除。

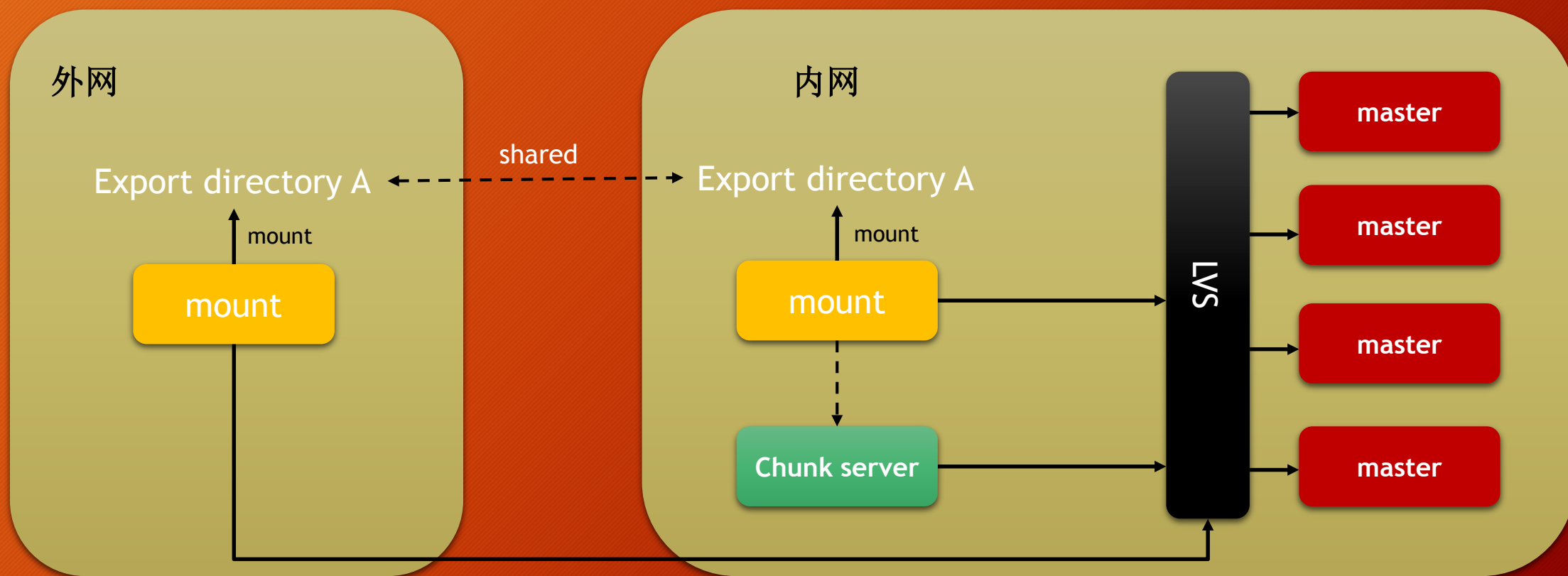
基于我们的应用， 可以做的一些改进

- 监控文件系统的变化



基于我们的应用，可以做的一些改进

- 多master实例，负载均衡



QA环节

Thanks