一、配置参数

合并文件存储相关的配置都在tracker.conf中。配置完成后，重启tracker和storage server。

1、use\_trunk\_file ：是否使用合并存储，缺省关闭。

2、slot\_min\_size：为一个小文件分配的最小空间，缺省256字节。即使你上传的文件只有0字节，也会分配这么多空间。

3、slot\_max\_size：使用合并存储的最大文件大小，缺省为16MB，超过这个size的文件，不会存储到trunk file中，而是作为一个单独的文件直接存储到文件系统中。

4、 trunk\_file\_size：trunk file大小，缺省为64MB。不要配置得过大或者过小，最好不要超过256MB。

5、trunk\_create\_file\_advance：是否提前创建trunk file，缺省关闭。

6、trunk\_create\_file\_time\_base：提前创建trunk file的时间点，比如可以指定半夜02:00创建

7、trunk\_create\_file\_interval：提前创建trunk file的时间间隔，单位为秒。如果一天创建一次，可以设置为86400。

8、trunk\_create\_file\_space\_threshold：创建trunk file需要达到的可用空间大小。如果空闲的trunk file空间大于本参数，则不会提前创建。

9、trunk\_init\_check\_occupying：trunk server初始化加载空闲blocks时，是否检查对应trunk file，看这个block是否已被占用。缺省为false。启用了本参数，trunk server初始化时间将大大延长。除非必要，请不要打开本选项。

10、trunk\_init\_reload\_from\_binlog：忽略快照文件，只从trunk binlog中加载空闲block。缺省为false。只要当从v3.10以下版本升级到v3.10以上版本时，可能才需要打开本选项。

二、原理

1、storage server内置trunk manager的功能，在storage server上对trunk file可用区块进行管理。在一个时间点，一个group只有一台storage server提供管理和查询服务，简称trunk manager。该group的其余storage server作为备机，只接收binlog。  
如何做到一个group只有一台storage server提供trunk管理服务，这个由tracker server统一协调完成。如果承担trunk manager那台storage server挂了，本组其余的一台storage server会自动升级为trunk manager，接替其工作。

2、trunk manager将trunk相关数据，全部存放到内存中管理。当storage server要存储一个小于阀值的文件（也就是小文件）时，先询问trunk manager，trunk manager返回存储到的trunk file文件名，以及存储起始的偏移量。当storage server成功完成文件存储后，向trunk manager报告。如果报告失败，则文件上传当失败处理。

3、trunk manager管理方案说明：  
trunk manager将trunk相关数据，全部存放到内存中管理。对于trunk更新操作（包括增加和删除两种），会记录到单独的binlog文件中，有专门的线程（trunk\_sync\_thread\_start创建该线程）将binlog文件同步给本组的其他storage server。为了节约内存空间，trunk file文件名，会单独存放。trunk file可用空间链表中，trunk filename采用指针方式指向。

当storage server向trunk manager请求分配文件空间时，trunk manager会先在内存中扫描有没有满足条件的可用trunk，如果有，那么直接返回。否则，一个创建trunk file，然后将新的trunk file记录到binlog和内存中，并完成分配。

为了提高分配效率，trunk manager将采取slot的方式对可用空间进行组织，比如初始的字节数为256，最大字节数为32MB，每次以2倍的速度递增，形如：  
256，512， 1K， 2K， 4K，。。。，1M，2M，4M，。。。，16M，32M  
在slot 256中的可用空间，是 >= 256，< 512的  
在slot 512中的可用空间，是 >= 512，< 1K的  
在slot 1K中的可用空间，是 >= 1K，< 2K 的  
以此类推。  
初次分配时，新创建的trunk file是在slot 32M中。随着trunk file被逐渐使用，可能会从slot 32M移动到slot 16M中，后面又可能被移动到slot 1MB中，如此等等  
每个slot下可用的空间信息，按可使用空间大小升序排列。这么做的好处是分配的效率会比较高，直接取第一个结点（链表头）即可。

为了简洁起见，不采用相邻空闲空间合并机制。

4、为了简化，trunk server的确存在单点问题。为了减少单点风险，trunk server会把更新binlog同步到同组的其他storage server上。  
万一trunk server挂了，由tracker server来协调，选举出新的trunk server。新选举出的trunk server，从binlog中加载已有数据，然后承担trunk server的功能。  
判断trunk server挂掉，有一定的超时机制。比如2分钟内，trunk server都处于离线状态，则认为trunk server挂掉。  
storage server升级为trunk server，由storage server主动申请的方式。  
多台tracker server，按ip地址升序排列。storage server向第一台tracker server申请成为trunk server，申请成功后，通知其他tracker server。

三、trunk manager的选出

relationship\_thread\_entrance->relationship\_select\_leader->被选为leader的tracker server将调用tracker\_mem\_find\_trunk\_servers->对每个组，如果pTrunkServer == NUL，则进一步调用tracker\_mem\_find\_trunk\_server(\*ppGroup, true)。

1. static int tracker\_mem\_find\_trunk\_server(FDFSGroupInfo \*pGroup,
2. const bool save)
3. {
4. FDFSStorageDetail \*pStoreServer;
5. FDFSStorageDetail \*\*ppServer;
6. FDFSStorageDetail \*\*ppServerEnd;
7. int result;
8. int64\_t file\_size;
9. int64\_t max\_file\_size;
11. pStoreServer = pGroup->pStoreServer;
12. if (pStoreServer == NULL)
13. {
14. return ENOENT;
15. }
17. result = tracker\_mem\_get\_trunk\_binlog\_size(pStoreServer->ip\_addr,
18. pGroup->storage\_port, &max\_file\_size);
19. if (result != 0)
20. {
21. return result;
22. }
24. ppServerEnd = pGroup->active\_servers + pGroup->active\_count;
25. for (ppServer=pGroup->active\_servers; ppServer<ppServerEnd; ppServer++)
26. {
27. if (\*ppServer == pStoreServer)
28. {
29. continue;
30. }
32. result = tracker\_mem\_get\_trunk\_binlog\_size((\*ppServer)->ip\_addr,
33. pGroup->storage\_port, &file\_size);
34. if (result != 0)
35. {
36. continue;
37. }
39. if (file\_size > max\_file\_size)
40. {
41. pStoreServer = \*ppServer; //找到trunk binlog长度最大的那个storage server。
42. }
43. }
45. return tracker\_mem\_do\_set\_trunk\_server(pGroup, \
46. pStoreServer, save); //设置trunk server并log。
47. }

四、从文件上传看trunk

1、storage\_deal\_task->storage\_upload\_file。storage\_upload\_file部分代码如下：

1. if (pFileContext->extra\_info.upload.file\_type & \_FILE\_TYPE\_TRUNK)
2. {
3. FDFSTrunkFullInfo \*pTrunkInfo;
5. pFileContext->extra\_info.upload.if\_sub\_path\_alloced = true;
6. pTrunkInfo = &(pFileContext->extra\_info.upload.trunk\_info); //此时pTrunkInfo只是做了清零初始化，还没有数据。
7. if ((result=trunk\_client\_trunk\_alloc\_space( \
8. TRUNK\_CALC\_SIZE(file\_bytes), pTrunkInfo)) != 0) //首先分配trunk的空间。从此跟踪可以查看trunk的内存管理。
9. {
10. pClientInfo->total\_length = sizeof(TrackerHeader);
11. return result;
12. }
14. clean\_func = dio\_trunk\_write\_finish\_clean\_up;
15. file\_offset = TRUNK\_FILE\_START\_OFFSET((\*pTrunkInfo));
16. pFileContext->extra\_info.upload.if\_gen\_filename = true;
17. trunk\_get\_full\_filename(pTrunkInfo, pFileContext->filename, \
18. sizeof(pFileContext->filename));
19. pFileContext->extra\_info.upload.before\_open\_callback = \
20. dio\_check\_trunk\_file\_when\_upload;
21. pFileContext->extra\_info.upload.before\_close\_callback = \
22. dio\_write\_chunk\_header;
23. pFileContext->open\_flags = O\_RDWR | g\_extra\_open\_file\_flags;
24. }
25. else
26. {
27. ......
28. }
30. return storage\_write\_to\_file(pTask, file\_offset, file\_bytes, \
31. p - pTask->data, dio\_write\_file, \
32. storage\_upload\_file\_done\_callback, \
33. clean\_func, store\_path\_index);

2、trunk\_client\_trunk\_alloc\_space

1. int trunk\_client\_trunk\_alloc\_space(const int file\_size, \
2. FDFSTrunkFullInfo \*pTrunkInfo)
3. {
4. int result;
5. ConnectionInfo trunk\_server;
6. ConnectionInfo \*pTrunkServer;
8. if (g\_if\_trunker\_self) //如果自己就是trunk server。
9. {
10. return trunk\_alloc\_space(file\_size, pTrunkInfo); //本例的trunk server就是自身，执行此处。新建trunk文件，调用trunk\_split(),这应该就是讲真个trunk文件划分成多个slot管理，便于分配。
11. }
13. if (\*(g\_trunk\_server.ip\_addr) == '\0')
14. {
15. logError("file: "\_\_FILE\_\_", line: %d, " \
16. "no trunk server", \_\_LINE\_\_);
17. return EAGAIN;
18. }
20. memcpy(&trunk\_server, &g\_trunk\_server, sizeof(ConnectionInfo));
21. if ((pTrunkServer=tracker\_connect\_server(&trunk\_server, &result)) == NULL)
22. {
23. return result;
24. }
26. result = trunk\_client\_trunk\_do\_alloc\_space(pTrunkServer, \
27. file\_size, pTrunkInfo);//发送STORAGE\_PROTO\_CMD\_TRUNK\_ALLOC\_SPACE命令给trunk server。
29. tracker\_disconnect\_server\_ex(pTrunkServer, result != 0);
30. return result;
31. }

3、trunk内存管理的数据结构：

（1）static struct fast\_mblock\_man free\_blocks\_man; //管理free节点。  
static struct fast\_mblock\_man tree\_nodes\_man;

1. struct fast\_mblock\_man
2. {
3. struct fast\_mblock\_node \*free\_chain\_head; //free node chain
4. struct fast\_mblock\_malloc \*malloc\_chain\_head; //malloc chain to be freed（fast\_mblock\_malloc管理一段连续的fast\_mblock\_node，每个32字节，本例一次性分配了32768个节点，也就是我们可以创建32768个trunk文件，用完了这么多节点后，才需要重新分配。）
5. int element\_size; //element size
6. int alloc\_elements\_once; //alloc elements once
7. pthread\_mutex\_t lock; //the lock for read / write free node chain
8. };

（2）FDFSTrunkNode \*pTrunkNode;  //其实一个FDFSTrunkNode就指代一个trunk文件

struct fast\_mblock\_node \*pMblockNode;（从上述free\_blocks\_man管理的free节点中取出一个节点）

pTrunkNode = (FDFSTrunkNode \*)pMblockNode->data;

1. typedef struct tagFDFSTrunkNode {
2. FDFSTrunkFullInfo trunk; //trunk文件 info
3. struct fast\_mblock\_node \*pMblockNode; //for free
4. struct tagFDFSTrunkNode \*next;
5. } FDFSTrunkNode;

（3）FDFSTrunkFullInfo trunk;

1. typedef struct tagFDFSTrunkFullInfo {
2. char status; //normal or hold
3. FDFSTrunkPathInfo path;
4. FDFSTrunkFileInfo file;
5. } FDFSTrunkFullInfo

4、trunk\_alloc\_space

1. int trunk\_alloc\_space(const int size, FDFSTrunkFullInfo \*pResult)
2. {
3. FDFSTrunkSlot target\_slot;
4. FDFSTrunkSlot \*pSlot;
5. FDFSTrunkNode \*pPreviousNode;
6. FDFSTrunkNode \*pTrunkNode;
7. int result;
9. STORAGE\_TRUNK\_CHECK\_STATUS();
11. target\_slot.size = (size > g\_slot\_min\_size) ? size : g\_slot\_min\_size;
12. target\_slot.head = NULL;
14. pPreviousNode = NULL;
15. pTrunkNode = NULL;
16. pthread\_mutex\_lock(&trunk\_mem\_lock);
17. while (1)
18. {
19. pSlot = (FDFSTrunkSlot \*)avl\_tree\_find\_ge(tree\_info\_by\_sizes \
20. + pResult->path.store\_path\_index, &target\_slot);
21. if (pSlot == NULL)
22. {
23. break;
24. }
26. pPreviousNode = NULL;
27. pTrunkNode = pSlot->head;
28. while (pTrunkNode != NULL && \
29. pTrunkNode->trunk.status == FDFS\_TRUNK\_STATUS\_HOLD)
30. {
31. pPreviousNode = pTrunkNode;
32. pTrunkNode = pTrunkNode->next;
33. }
35. if (pTrunkNode != NULL)
36. {
37. break;
38. }
40. target\_slot.size = pSlot->size + 1;
41. }
43. if (pTrunkNode != NULL)
44. {
45. if (pPreviousNode == NULL)
46. {
47. pSlot->head = pTrunkNode->next;
48. if (pSlot->head == NULL)
49. {
50. trunk\_delete\_size\_tree\_entry(pResult->path. \
51. store\_path\_index, pSlot);
52. }
53. }
54. else
55. {
56. pPreviousNode->next = pTrunkNode->next;
57. }
59. trunk\_free\_block\_delete(&(pTrunkNode->trunk));
60. }
61. else
62. {
63. pTrunkNode = trunk\_create\_trunk\_file(pResult->path. \
64. store\_path\_index, &result); //第一次执行时。最终是要获得一个pTrunkNode（相当于一个文件）。
65. if (pTrunkNode == NULL)
66. {
67. pthread\_mutex\_unlock(&trunk\_mem\_lock);
68. return result;
69. }
70. }
71. pthread\_mutex\_unlock(&trunk\_mem\_lock);
73. result = trunk\_split(pTrunkNode, size); //未跟踪，猜测：由于这是一个trunk file（64M），我们要写的文件远小于这个大小，当然应该将要写入的和剩余的部分分开。剩余的部分重新放入到free block中去（trunk\_add\_free\_block）。
74. if (result != 0)
75. {
76. return result;
77. }
79. pTrunkNode->trunk.status = FDFS\_TRUNK\_STATUS\_HOLD;//将状态从free改为hold。
80. result = trunk\_add\_free\_block(pTrunkNode, true); //将这个要写的文件块也加入free block中（同时写trunk binlog），用于之后分配。
81. if (result == 0)
82. {
83. memcpy(pResult, &(pTrunkNode->trunk), \
84. sizeof(FDFSTrunkFullInfo)); //其中设置好了即将写的trunk文件的id，offset，要写的大小。注意这里的pResult即&(pFileContext->extra\_info.upload.trunk\_info)，也就是说这个地址已经记录在pFileContext中了。之后写文件就是用这个trunk node（状态为hold），写完后会trunk\_delete\_space删掉
85. }
87. return result;
88. }

最后，有了文件trunk文件的id、offset、要写的大小，之后的写文件就很简单了（与非trunk文件的写类似）。开启了truck线程后，同步线程也变成了两个，一个是同步文件（与之前相同），一个是同步trunk binlog（trunk file在split时，及块分配时，都会产生binlog；有了binlog同步，才能防止trunk server单点故障。）