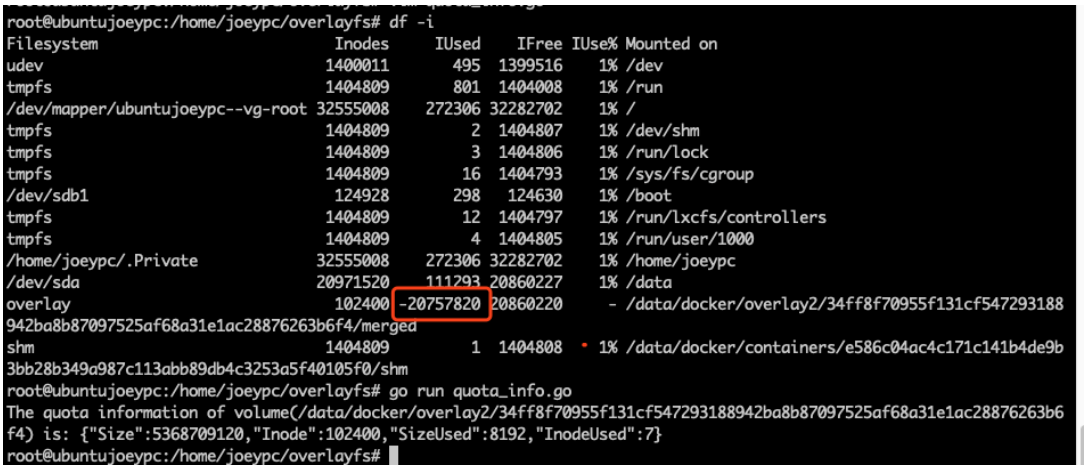
overlay2 driver支持xfs inode隔离

docker inode隔离

问题描述

在给 docker overlay2 driver 加xfs inode quota 限制时，遇到一个bug：df –i看到 容器目录是被限制了 inodes上限已经是设置的 102400，但是 IUsed 却为负数。

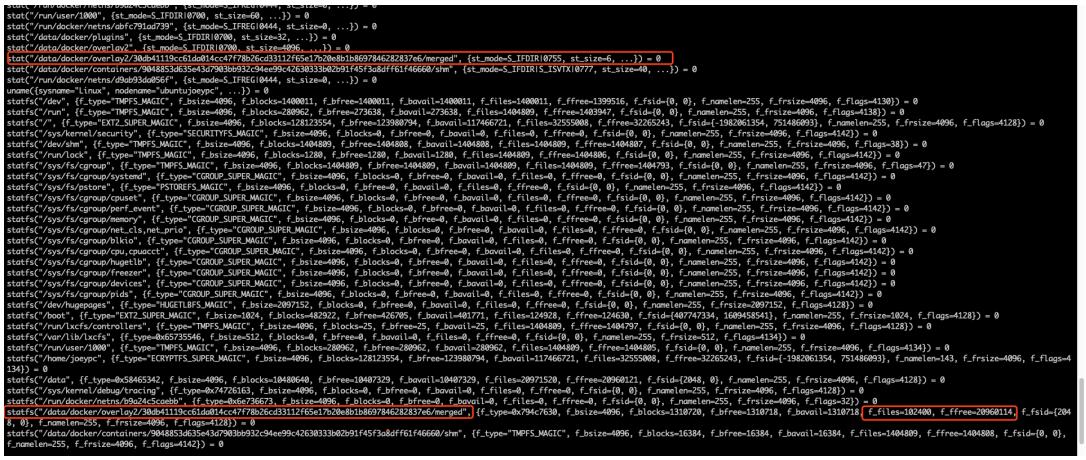


经过验证，在容器rootfs中实际能够使用的文件inode上限是已经被限制，只是df显示的问题。

先做了一些简单的测试：

比如对 fs_disk_quota_t 结构体的其他成员变量进行检查，排除了可以设置IFree or IUsed值溢出的问题。
https://github.com/torvalds/linux/blob/master/include/uapi/linux/dqblk_xfs.h
并且也尝试去 修改SetInodeQuota的路径，为 /data/docker/overlay2/<container-uuid>/merged ,或 /data/docker/overlay2/<container-uuid>/diff 仍然无效。

跟踪df –i的系统调用



发现df命令 其实是通过 stat 和 statfs 这两个系统调用，去拿/proc/self/mountinfo 里挂载点相应的文件系统统计信息（即struct statfs *buf）。

```
statfs（）系统调用返回有关已装入文件系统的信息。路径是安装的文件系统中任何文件的路径名。 buf是一个指向statfs结构的指针，大致定义如下：

struct statfs {
        __fsword_t f_type;    /* 文件系统的类型 (见下文) */
        __fsword_t f_bsize;   /* 最佳传输块大小 */
        fsblkcnt_t f_blocks;  /* 文件系统中的总数据块*/
        fsblkcnt_t f_bfree;   /* 文件系统中的空闲块*/
        fsblkcnt_t f_bavail;  /* 空闲块可用于非特权用户 */
        fsfilcnt_t f_files;   /* 文件系统中的文件总数 */
        fsfilcnt_t f_ffree;   /* 文件系统中的空闲文件节点 */
        fsid_t     f_fsid;    /*文件系统ID */
        __fsword_t f_namelen; /* 文件名的最大长度 */
        __fsword_t f_frsize;  /* 片段大小（自Linux 2.6以来） */
        __fsword_t f_flags;   /* 挂载文件系统的标志（从Linux 2.6.36开始） */
        __fsword_t f_spare[xxx]; /* 填充字节保留供将来使用 */
};

```

Not using native diff for overlay2: opaque flag erroneously copied up, consider update to kernel 4.8 or later to fi

通过ftrace 跟踪 statfs系统调用：

```
 6)               |      vfs_statfs() {
 6)               |        statfs_by_dentry() {
 6)   0.088 us    |          security_sb_statfs();
 6)               |          xfs_fs_statfs [xfs]() {
 6)   0.084 us    |            _raw_spin_lock_irqsave();
 6)               |            _raw_spin_unlock_irqrestore() {
 6)   0.029 us    |              __pv_queued_spin_unlock();
 6)   0.254 us    |            }
 6)   0.029 us    |            _raw_spin_lock_irqsave();
 6)               |            _raw_spin_unlock_irqrestore() {
 6)   0.028 us    |              __pv_queued_spin_unlock();
 6)   0.235 us    |            }
```

```
 6)   0.029 us    |            _raw_spin_lock_irqsave();
 6)               |            _raw_spin_unlock_irqrestore() {
 6)   0.029 us    |              __pv_queued_spin_unlock();
 6)   0.242 us    |            }
 6)   0.027 us    |            _raw_spin_lock();
 6)   0.030 us    |            __pv_queued_spin_unlock();
 6)               |            xfs_qm_statvfs [xfs]() {
 6)               |              xfs_qm_dqget [xfs]() {
 6)               |                mutex_lock() {
 6)   0.026 us    |                  _cond_resched();
 6)   0.391 us    |                }
 6)               |                mutex_lock() {
 6)   0.028 us    |                  _cond_resched();
 6)   0.345 us    |                }
 6)   0.028 us    |                mutex_unlock();
 6)   1.940 us    |              }
 6)   0.056 us    |              xfs_fill_statvfs_from_dquot [xfs]();
 6)               |              xfs_qm_dqput [xfs]() {
 6)   0.027 us    |                mutex_unlock();
 6)   0.222 us    |              }
 6)   2.864 us    |            }
 6)   6.781 us    |          }
 6)   7.666 us    |        }
 6)   7.944 us    |      }
 6)               |      path_put() {
 6)   0.040 us    |        dput();
 6)               |        mntput() {
 6)   0.040 us    |          mntput_no_expire();
 6)   0.299 us    |        }
 6)   0.710 us    |      }
 6) + 30.327 us   |    }
 6)   0.158 us    |    do_statfs_native();
 6) + 30.998 us   |  }

```

跟踪xfs源码

```
/*
 * Dquots are structures that hold quota information about a user or a group,
 * much like inodes are for files. In fact, dquots share many characteristics
 * with inodes. However, dquots can also be a centralized resource, relative
 * to a collection of inodes. In this respect, dquots share some characteristics
 * of the superblock.
 * XFS dquots exploit both those in its algorithms. They make every attempt
 * to not be a bottleneck when quotas are on and have minimal impact, if any,
 * when quotas are off.
 */
```

```
/*
 * The incore dquot structure
```

```
 */
typedef struct xfs_dquot {
    uint         dq_flags;  /* various flags (XFS_DQ_*) */
    struct list_head q_lru;     /* global free list of dquots */
    struct xfs_mount*q_mount;   /* filesystem this relates to */
    struct xfs_trans*q_transp;  /* trans this belongs to currently */
    uint         q_nrefs;   /* # active refs from inodes */
    xfs_daddr_t  q_blkno;   /* blkno of dquot buffer */
    int      q_bufoffset;   /* off of dq in buffer (# dquots) */
    xfs_fileoff_t    q_fileoffset;  /* offset in quotas file */

    xfs_disk_dquot_t q_core;    /* actual usage & quotas */
    xfs_dq_logitem_t q_logitem; /* dquot log item */
    xfs_qcnt_t   q_res_bcount;  /* total regular nblks used+reserved */
    xfs_qcnt_t   q_res_icount;  /* total inos allocd+reserved */
    xfs_qcnt_t   q_res_rtbcount;/* total realtime blks used+reserved */
    xfs_qcnt_t   q_prealloc_lo_wmark;/* prealloc throttle wmark */
    xfs_qcnt_t   q_prealloc_hi_wmark;/* prealloc disabled wmark */
    int64_t      q_low_space[XFS_QLOWSP_MAX];
    struct mutex     q_qlock;   /* quota lock */
    struct completion q_flush;  /* flush completion queue */
    atomic_t          q_pincount;   /* dquot pin count */
    wait_queue_head_t q_pinwait;    /* dquot pinning wait queue */
} xfs_dquot_t;
```

```
/*
 * Directory tree accounting is implemented using project quotas, where
 * the project identifier is inherited from parent directories.
 * A statvfs (df, etc.) of a directory that is using project quota should
 * return a statvfs of the project, not the entire filesystem.
 * This makes such trees appear as if they are filesystems in themselves.
 */
void
xfs_qm_statvfs(
    xfs_inode_t     *ip,
    struct kstatfs      *statp)
{
    xfs_mount_t     *mp = ip->i_mount;
    xfs_dquot_t     *dqp;

    if (!xfs_qm_dqget(mp, NULL, xfs_get_projid(ip), XFS_DQ_PROJ, 0, &dqp))
{
        xfs_fill_statvfs_from_dquot(statp, dqp);
        xfs_qm_dqput(dqp);
    }
}
```

```
/*
 * Given the file system, inode OR id, and type (UDQUOT/GDQUOT), return a
 * a locked dquot, doing an allocation (if requested) as needed.
```

```
* When both an inode and an id are given, the inode's id takes precedence.
* That is, if the id changes while we don't hold the ilock inside this
* function, the new dquot is returned, not necessarily the one requested
* in the id argument.
*/
int
xfs_qm_dqget(
    xfs_mount_t *mp,
    xfs_inode_t *ip,      /* locked inode (optional) */
    xfs_dqid_t  id,   /* uid/projid/gid depending on type */
    uint        type,     /* XFS_DQ_USER/XFS_DQ_PROJ/XFS_DQ_GROUP */
    uint        flags,    /* DQALLOC, DQSUSER, DQREPAIR, DOWARN */
    xfs_dquot_t **O_dqpp) /* OUT : locked incore dquot */
{

}
```

```
STATIC void
xfs_fill_statvfs_from_dquot(
    struct kstatfs      *statp,
    struct xfs_dquot    *dqp)
{
    __uint64_t      limit;

    limit = dqp->q_core.d_blk_softlimit ?
        be64_to_cpu(dqp->q_core.d_blk_softlimit) :
        be64_to_cpu(dqp->q_core.d_blk_hardlimit);
    if (limit && statp->f_blocks > limit) {
        statp->f_blocks = limit;
        statp->f_bfree = statp->f_bavail =
            (statp->f_blocks > dqp->q_res_bcount) ?
             (statp->f_blocks - dqp->q_res_bcount) : 0;
    }

    limit = dqp->q_core.d_ino_softlimit ?
        be64_to_cpu(dqp->q_core.d_ino_softlimit) :
        be64_to_cpu(dqp->q_core.d_ino_hardlimit);
    if (limit && statp->f_files > limit) {
        statp->f_files = limit;
        statp->f_ffree =
            (statp->f_files > dqp->q_res_icount) ?
             (statp->f_ffree - dqp->q_res_icount) : 0;
    }
}
```

查看parent statfs

```
#include <sys/vfs.h>
#include <stdlib.h>
```

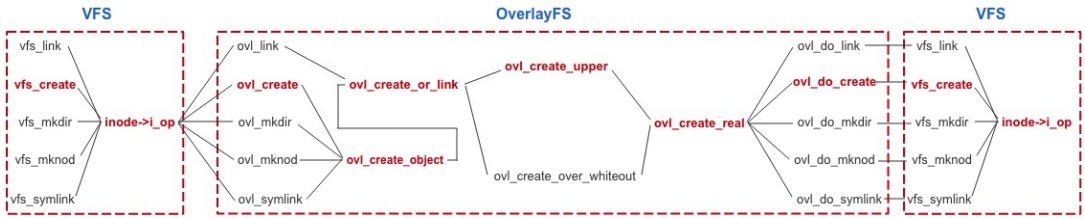
```c
#include <stdio.h>

int main(int argc,char **argv)
{
        struct statfs buf;

//if(statfs("/data/docker/overlay2/eff38954f57aa0007a0d4613136f0dcfad55842758dd2f54cb4b16833a296e43",&buf)==-1)
    if(statfs("/data/docker/overlay2",&buf)==-1)
        {
                printf("statfs bad\n");
                exit(1);
        }

        printf("type = %ld \n",buf.f_type);
        printf("bsize = %ld \n",buf.f_bsize);
        printf("blocks = %ld\n",buf.f_blocks);
        printf("bfree = %ld\n",buf.f_bfree);
        printf("bavail = %ld\n",buf.f_bavail);
        printf("files = %ld\n",buf.f_files);
        printf("ffree = %ld\n",buf.f_ffree);
        printf("fsid = %d %d\n",buf.f_fsid.__val[0],buf.f_fsid.__val[1]);
        printf("namelen = %ld\n",buf.f_namelen);
        printf("frsize = %ld\n",buf.f_frsize);
        printf("flags = %ld\n",buf.f_flags);

        return 0;

}
```

```
root@ubuntujoeypc:/home/joeypc/overlayfs# gcc statfs.c -o test
拿/data/docker/overlay2 目录的 statfs
root@ubuntujoeypc:/home/joeypc/overlayfs# ./test
type = 1481003842
bsize = 4096
blocks = 10480640
bfree = 10406809
bavail = 10406809
files = 20971520
ffree = 20960069
fsid = 2048 0
namelen = 255
frsize = 4096
拿/data/docker/overlay2/13d59bffdfc0a2ab62e4a8393b67d7b1becc245bd381f610e2ee06b6b35f9edc 目录的 statfs
type = 1481003842
bsize = 4096
blocks = 1310720
bfree = 1310718
bavail = 1310718
```

```
files = 102400
ffree = 20960062
fsid = 2048 0
namelen = 255
frsize = 4096
flags = 4128

使用quota_info.go 查看/data/docker/overlay2/13d59bffdfc0a2ab62e4a8393b67d7b1becc245bd381f610e2ee06b6b35f9edc 目录的 fs_disk_quota_t.（d_ino_hardlimit/d_icount）
The quota information of volume(/data/docker/overlay2/13d59bffdfc0a2ab62e4a8393b67d7b1becc245bd381f610e2ee06b6b35f9edc) is:
{"Size":5368709120,"Inode":102400,"SizeUsed":8192,"InodeUsed":7}
```

结论：

整个链路大致可以理解为下图：



来自：https://arkingc.github.io/2017/12/22/2017-12-22-linux-code-overlayfs-create_delete/

内核OverlayFS 的数据结构

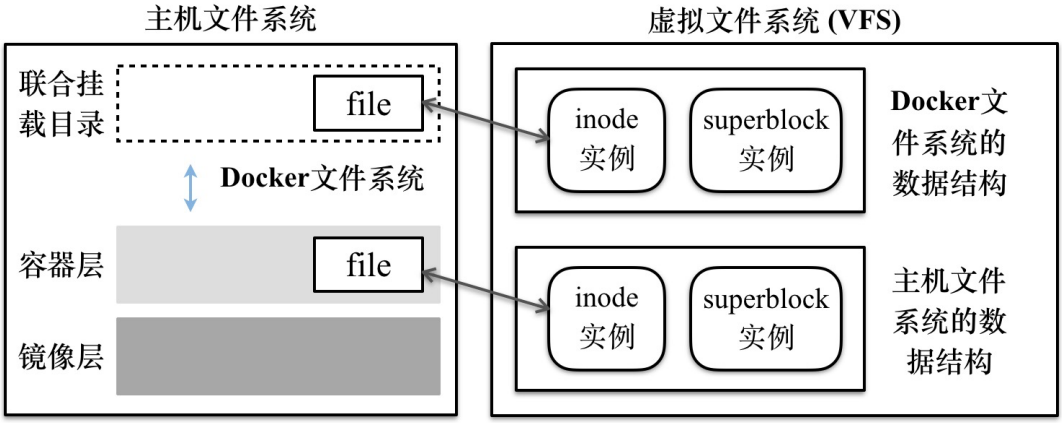
```
struct ovl_entry {
    struct dentry *__upperdentry;//记录upper层dentry
    struct ovl_dir_cache *cache;
    union {
        struct {
            u64 version;
            bool opaque;
        };
        struct rcu_head rcu;
    };
    unsigned numlower;//lower层数
    struct path lowerstack[];//记录lower层路径
};

struct ovl_entry *oe = dentry->d_fsdata;

```

结构体ovl_entry记录了OverlayFS中文件的层次信息，通过这个结构体，内核可以根据一个OverlayFS文件的dentry来实现对相应upper层和lower层的文件访问。由于OverlayFS的本质是将对文件的操作转化为对底层文件系统upper层或lower层文件的操

作。因此在OverlayFS中，会大量涉及到对文件层次信息的访问。理解这个结构有助于理解OverlayFS如何实现操作转化。因此这个结构很重要。

在docker 容器中创建文件的底层原理

我们知道，对于overlayfs来说, 创建文件时，同名文件上层会覆盖下层，而同名目录则是会进行合并 。但是实际在docker中创建文件，在主机backing文件系统 和 docker rootfs 这两层中，同时创建文件消耗inode。但是，在overlayfs中直接创建文件，对底层的backingfs是不会有inode消耗。



root@ubuntujoeypc:/home/joeypc/overlayfs# docker exec –ti 9048853d635e touch test.txt

```
root@ubuntujoeypc:/home/joeypc/overlayfs# tree -L 2 /data/docker/overlay2/30db41119cc61da014cc47f78b26cd33112f65e17b20e8b1b8697846282837e6
/data/docker/overlay2/30db41119cc61da014cc47f78b26cd33112f65e17b20e8b1b8697846282837e6
|-- diff
|   `-- test.txt
|-- link
|-- lower
|-- merged
|   |-- bin
|   |-- boot
|   |-- dev
|   |-- etc
|   |-- home
|   |-- lib
|   |-- lib64
|   |-- media
|   |-- mnt
|   |-- opt
|   |-- proc
|   |-- root
|   |-- run
|   |-- sbin
|   |-- srv
|   |-- sys
|   |-- test.txt
|   |-- tmp
|   |-- usr
|   `-- var
`-- work
    `-- work

23 directories, 4 files
```

分析：如下例子演示了在容器中创建文件，实际消耗的inode情况：

/data/docker/overlay2/ea81d7d05803fe2e44e79e821e32f32f4efaf26362b103651191f42532c03829/

|

|----------------upper

|----------------merged

限制
了/data/docker/overlay2/ea81d7d05803fe2e44e79e821e32f32f4efaf26362b103651191f42532c03829/的话，按理来说只有容器内创建文件时，会消耗inode，也就是会在upper这个目录下创建文件。merged里面是overlayfs，它只是提供多层联合挂载得到的一个文件视图，这些文件在内存中有都有inode，但是不会消耗磁盘的inode，也就是不会消耗XFS的inode，所以按理是不会有影响的。

```
/data/docker/overlay2/ea81d7d05803fe2e44e79e821e32f32f4efaf26362b103651191f42532c03829/
    |
    |-----------------upper
                        |----------file
    |-----------------merged
                        |------------file
```

也就是upper里边有个file的话，mount之后merged里面也有个file。这两个file在VFS中有不同的inode对应。但是merged里的inode属于Overlayfs，并不会消耗主机文件系统XFS的inode。overlayfs的inode只在内存中，容器停掉后就都释放掉了。

讨论

1. overlay xfs inode quota到底应该在哪一层目录去设置？是merged层，upper(diff)层，还是上一层 /data/docker/overlay2/<container-uuid>/ ？
2. 为何出现回滚docker二进制，创建容器之后，inode仍被限制为inode 100K的情况？