Home (/ucsd-udisk/udisk/wikis/home) Pages (/ucsd-udisk/udisk/wikis/pages)

Git Access (/ucsd-udisk/udisk/wikis/git_access)

✚ New Page

Udisk v4 设计文档 · last edited by bruce.li@ucloud.cn about a year ago

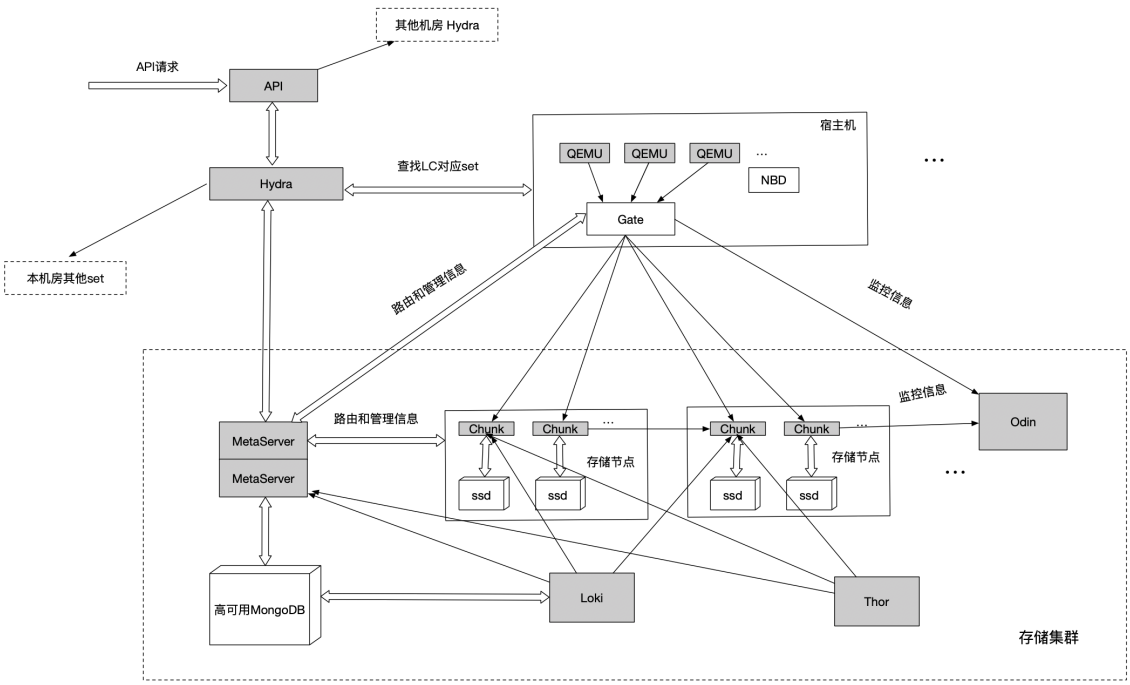
Page History (/ucsd-udisk/udisk/wikis/udisk-v4-%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3/history)

Edit (/ucsd-udisk/udisk/wikis/udisk-v4-%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3/edit)

Delete (/ucsd-udisk/udisk/wikis/udisk-v4-%E8%AE%BE%E8%AE%A1%E6%96%87%E6%A1%A3)

udisk新架构V4

架构图:



主要的变化

- 取消接入层Proxy，取消逻辑盘索引，改用一致性哈希算法，物理分片从1G减小到16M，实际的物理空间由chunk服务自己管理。
- 下层SSD能提供20W~25W IOPS，使用单个chunk服务(采用多线程)负责单块磁盘的读写
- 底层从直接读写物理磁盘，换成文件系统，便于对磁盘空间的管理。
- 取消对于Iscsi的支持，在需要在宿主机上挂载块设备的场景下通过NBD来实现，实际还是走gate。
- 使用QoS算法，按逻辑盘容量提供IOPS和带宽的限制

- 中心元数据存储采用3.4 MongoDB 提供高可用，强一致元数据存储。

性能指标

SSD性能指标

- IOPS = min{1200 + 30 * 容量, 24000}
- 吞吐量 = min{80 + 0.5 * 容量, 260} MBps
- 延迟小于2ms
- QoS

SATA机械盘性能指标

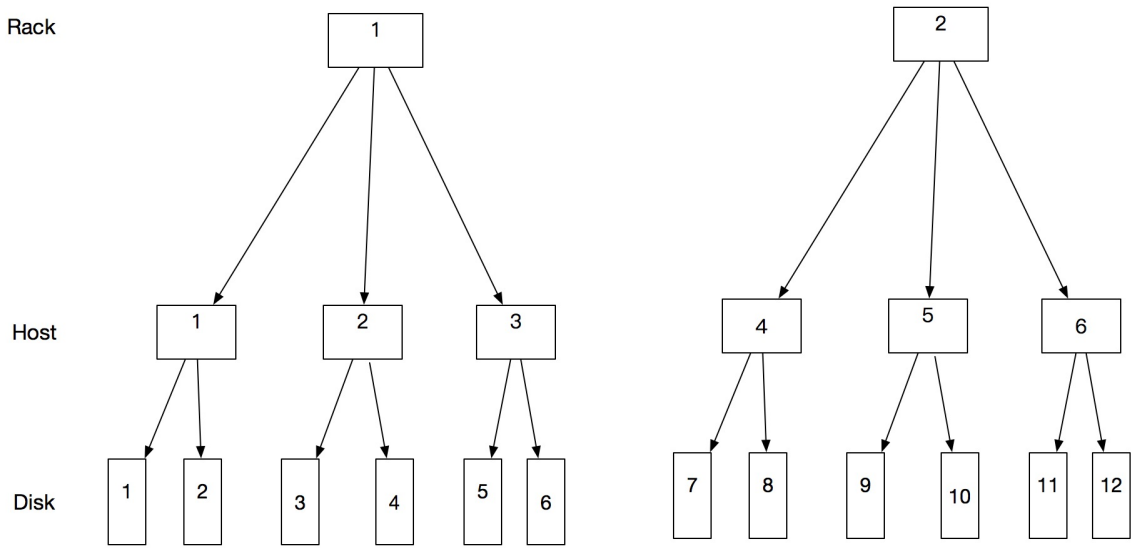
模块及功能

- MetaServer: 元数据管理模块，提供路由信息的管理和更新，实现创建，列表，回收，并监控整个集群的健康状况，控制chunk服务的上下线。
- gate: IO接入模块，处理从QEMU传出的IO，提供逻辑盘路由计算，转发IO至后端Chunk，方舟IO转发，旧版本功能兼容等功能。
- Chunk:管理底层文件系统空间，提供IO多副本读写，写时分配分片文件，读写错误上报，错误修复时数据同步。
- doctor: 修复模块，负责控制数据的同步。
- monitor：监控模块，接收集群的统计信息以及异常告警信息，

一致性hash算法

副本集（3份冗余）的划分

集群拓扑结构图：



将集群中的所有disk按Host和Rack交叉排列，按上图则排列成 Disk {1 7 3 9 5 11 2 8 4 10 6 12}，按连续的三个disk组成pgt(partion group tuple)，这三个磁盘分布在三个不同的Host，至少两个不同的Rack，按上图可以划分为如下pgt:

```
pgt_id0: disk {1  7  3}
pgt_id1: disk {7  3  9}
pgt_id2: disk {3  9  5}
pgt_id3: disk {9  5 11}
pgt_id4: disk {5  11  2}
pgt_id5: disk {11  2  8}
pgt_id6: disk {2   8  4}
pgt_id7: disk {8   4 10}
pgt_id8: disk {4  10  6}
pgt_id9: disk {10   6  12}
pgt_id10: disk {6   12  1}
pgt_id11: disk {12   1  7}
```

为了使主副本均匀分布在三个chunk server上，将pgt分成3个pg(partion group), 这三个pg有相同的路由以及不同的主副本，pg作为物理节点分布在哈希环上。

```
pg_id0: {pgt_id0, primary_chunk_id0}
pg_id1: {pgt_id0, primary_chunk_id1}
pg_id2: {pgt_id0, primary_chunk_id2}

pg_id3: {pgt_id1, primary_chunk_id3}
pg_id4: {pgt_id1, primary_chunk_id4}
pg_id5: {pgt_id1, primary_chunk_id5}

pg_id6: {pgt_id2, primary_chunk_id6}
pg_id7: {pgt_id2, primary_chunk_id7}
pg_id8: {pgt_id2, primary_chunk_id8}
...
```

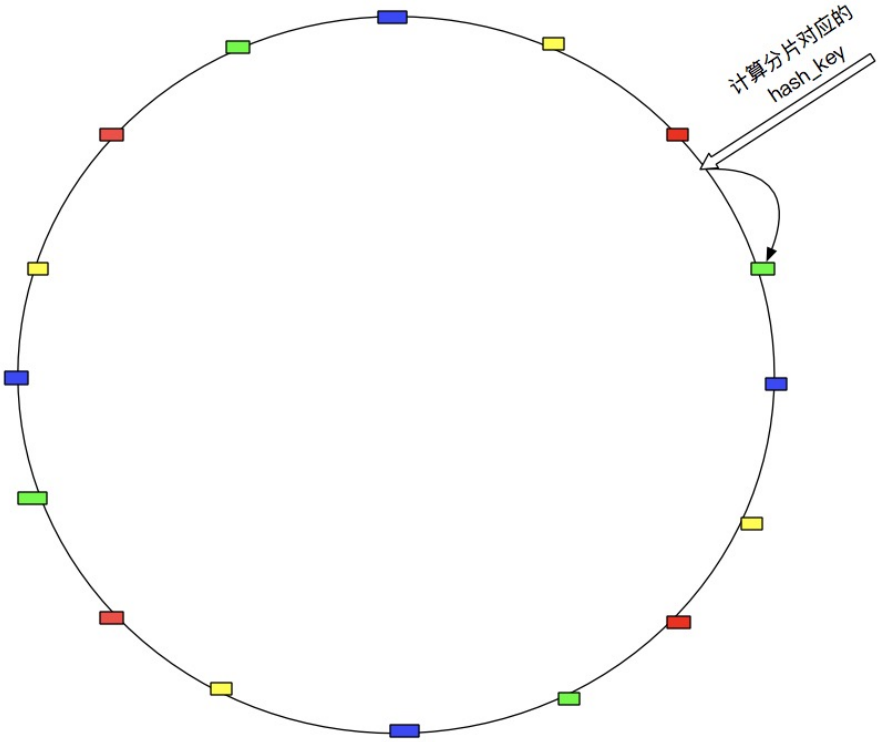
部署时为了能够将主打散到多台机器上至少需要5台机器，分别和前两台和后两台组成pg, 机器交叉然后，在每台机器逐次选一块盘，这样盘必然满足于前两个盘和后两个盘在不同的机器上，这样修复是占用的网卡流量分摊在最多4台机器上。
这种部署方式复ubs2的分配磁盘算法类似，只要容量最大的两个机架下的机器数相等，机器下的磁盘容量也相等，就能符合容灾要求(三个备份至少在两个机架)使用完所有的容量。

hash环生成

hash环相关的三类参数

- pg_num: pg的个数，即物理节点的个数
- scale_num: 物理节点放大倍数，即为每个物理节点生成 scale_num个虚节点
- conflict_a conflict_b：冲突解决参数，当放置虚节点的时候有冲突时通过 key=a*hash(pg_id)+b , key=a*key+b ..., 直到冲突解决。

假设有4个pg_num（蓝黄红绿,正常是三的整数倍），为每个pg生成4个虚拟节点，按如图方式放置在环上，当有来时，用lc_id+pcg_no计算hash，顺时针查找第一个虚节点，找到对应的路由。



索引结构

数据库中的collection结构

- t_cluster_info

```
{
    pg_num : 集群中的chunk个数, 整数
    scale_num : 每个chunk的个数的放大倍数, 整数
    conflict_a : chunk hash冲突解决规则, 整数
    conflict_b : chunk hash冲突解决规则, 整数
}
```

- t_chunk_info

```
{
    id : chunk的ID,整数
    ip : chunk的ip,字符串
    port : chunk的port,字符串
    host : chunk所在的host ID,字符串
    rack : chunk所在的host所在的rack ID,字符串
    state : chunk的状态
}
```

- t_pg_tuple_info

```
{
    pg_tuple_id : 整数而且必须按序增加
    pg_0_id : 整数
    pg_0_primary_chunk : pg_0的primary chunk的ID
    pg_1_id : 整数
    pg_1_primary_chunk : pg_1的primary chunk的ID
    pg_2_id : 整数
    pg_2_primary_chunk : pg_2的primary chunk的ID
    chunk_0 : 第一个 chunk的ID
    chunk_1 : 第二个 chunk的ID
    chunk_2 : 第三个 chunk的ID
}
```

- t_lc_info

```
{
    id : lc的id，字符串
    name : lc的名字，字符串
    size : lc的大小，整数
    oid : lc所属的用户的OrganizationId，字符串
    top_oid : lc所属的用户的TopOrganizationId，字符串
    create_time : lc的创建时间，整数
    throw_time : lc进回收站的时间
    delete_time : lc删除时间
    recycle_time : lc被回收时间
    last_attach_time : lc的最近的一次挂载时间，整数
    last_detach_time : lc的最近的一次卸载时间，整数
    last_resize_time : lc的最近的一次的扩容时间，整数
}
```

- t_gray_chunk_info

```
{
    lc_id : lc的ID，字符串
    mother_chunk_id_0 : 灰度的母Chunk 0 ID，字符串
    mother_chunk_id_1 : 灰度的母Chunk 1 ID，字符串
    mother_chunk_id_2 : 灰度的母Chunk 2 ID，字符串
    mother_chunk_id_0_gray_chunk_id : 灰度的Chunk 0
    mother_chunk_id_1_gray_chunk_id : 灰度的Chunk 1
    mother_chunk_id_2_gray_chunk_id : 灰度的Chunk 2
}
```

IO流程

io协议及相关数据结构

```
#define  MAX_PC_NAME_LEN 32// uuid(32)_pcg_no(10)
struct udisk_gate_io_hdr
{
  uint32_t size;  //io size 包含头的大小
  int8_t proto_version;  // gate 协议版本号
  uint64_t pgp_version; //pgp 路由版本号
  uint32_t cmd;   // io type : read write
  uint64_t flowno; // io 编号
  uint32_t fragno;  // io 分片号
  uint32_t offset;  // io 在分片文件内的 offset
  uint32_t length;  // 每次读写的文件长度, 读文件需要考虑
  int8_t retcode;
  char pc_name[MAX_PC_NAME_LEN]; //分片文件名 lc_id+pcg_no
  uint32_t magic_num;  //
} __attribute__((packed));

struct vnode {
  const uint32_t  pg_id;
}

struct pg_info {
  uint32_t pg_id;
  uint32_t chunk0_id;
  uint32_t chunk1_id;
  uint32_t chunk2_id
  uint32_t primary_chunk_id;
}

struct chunk_info {
  string chunk_id;
  string gray_chunk_id; //对应的灰度chunk id
  string ip;
  uint32_t port;
  uint32_t state;
  uint64_t conn_key;  // 由ip port 生成连接的key，不用每次计算
}
```

- gate 通过对pc_name进行hash得到vnode 结构，通过vnode中的pgp_id 找到路由版本号，通过pg_id从pg_map中找到Primary_chunk_id，通过 chunk_map查找到对应的连接将IO发送出去
- chunk 收到IO后通过对pc_name进行hash查找到pg_id的Primary_chunk_id, 如果是自身，则写IO并进行转发到正常的从，否则只写本地后返回。

IO失败处理

- 1） gate IO超时，超时后重发。
- 2） gate IO失败，失败原因
  - 路由版本不相等1s后重试, 因为心跳会在1s内更新路由
  - EIO和磁盘超时错误3s后重试，这种错误需要上报MetaServer处理时间稍长。
  - event error 1s后重试该链接上的所有等待的IO

集群路由更新方式

集群路由在以下情况下会主动更新，需要依赖心跳下发

- MetaServer与某个chunk间的心跳丢失，并且互相结对的chunk也上报了该chunk心跳丢失
- chunk主动上报磁盘读写错误 (改为心跳上报)
- chunk上报读写磁盘超时，超过一定的次数 (改为心跳上报， 次数由metaserver确定，chunk磁盘超时就报故障)
- 由于运维的需要，人工触发Chunk服务下线
- 当新加磁盘或者机器时，新加Chunk服务上线。

gate和chunk会上报心跳到MetaServer，上报时会带上各自的集群路由版本号，MetaServer根据版本号是否陈旧判端是否下发pgp路由，保证集群路由变化后所有chunk和gate能最终感知到。

集群路由下发到Chunk

Chunk通过心跳向MetaServer上报的ChunkInfo协议：

```
message ChunkHeartbeatRequest {
  required PgpVersionInfo pgp_version_info;
  required ChunkServerInfo chunk_info;
}

message ChunkServerInfo {
    required uint32 chunk_id;
    required uint32 chunk_state;
    required uint32 logical_used_space; // chunk管理的空间的逻辑使用量
    required uint32 total_space; // 总空间大小
    required uint32 availabel_space; //  实际可用的空间大小
}
```

Chunk心跳中会带上pgp_version，MetaServer 收到心跳后会比较每个chunk服务的pgp_version，如果已经陈旧,将最新的pgp路由返回给该chunk，并且带上最新的pgp_version。

```
message ChunkHeartbeatResponse {
    optional PgpInfo pgp_info;
    repeated ChunkRouteInfo chunk_route_info;
}

message PgpInfo {
    required uint32 pgp_id;
    required uint64 pgp_version;
    required string chunk_id0;
    required string chunk_id1;
    required string chunk_id2;
    required string primary_chunk_id0;
    required string primary_chunk_id1;
    required string primary_chunk_id2;
}

message ChunkRouteInfo {
  required string chunk_id;
  required string ip;
  required uint32 port;
  required uint32 state;
}
```

集群路由下发到Gate

Gate到MetaServer的心跳会带上所有的pgp_version，MetaServer会将陈旧的pgp_verision对应的最新路由返回，并带上最新的版本号。

```
message GateHeartbeatRequest {
    repeated PgpVersionInfo pgp_version_info;
}

message GateHeartbeatResponse {
  repeated PgpInfo pgp_info;
  repeated ChunkRouteInfo chunk_route_info;
}
```

故障上报

1、Primary EIO或则副本返回EIO

EIO是磁盘严重故障，出现则说明磁盘损坏，需要将对应磁盘从集群中剔除，EIO的具体处理流程如下：

- 1）Primary Chunk发生EIO或则副本返回EIO后，返回Gate EIO
- 2）Primary Chunk记录下EIO故障，然后由Chunk与MetaServer的heartbeat外带故障信息，告知MetaServer那个Chunk Server发生了EIO
- 3）MetaServer收到Primary Chunk heartbeat外带的故障消息后，向后端数据库请求修改对应Chunk Server的状态，并增加对应的pg_pair_version

2、Primary IOTimeout，副本Timeout或则副本返回IOTimeout

- 1）Primary Chunk发生IOTimeout或则复制IO超时，则返回GATE IOTimeout
- 2）Primary Chunk记录下IOTimeout的信息，然后有Chunk与MetaServer的heartbeat外带故障信息，告知MetaServer哪个Chunk Server发生了IOTimeout
- 3）MetaServer在收到Primary Chunk heartbeat外带的IOTimeout后，如果MetaServer第一次收到Timeout对应的Chunk的上报，则仅仅记录下来上报的时间，如果不是第一次上报的话，则比较当前时间和对应的Chunk第一次上报的时间，如果相差超过一定时间，则设置对应的超时计数为1，并将第一次超时时间设置为当前时间，如果未超过一定时间，判断超时计数是否超过一定次数，如果超过的话，则认为这个Chunk故障，需要将其移除集群，具体步骤是更新数据库中对应的chunk的状态，增加相关的pg_pair_version,并清理内存中相关的统计信息，如果未超过一定次数，则仅仅增加超时计数

故障修复

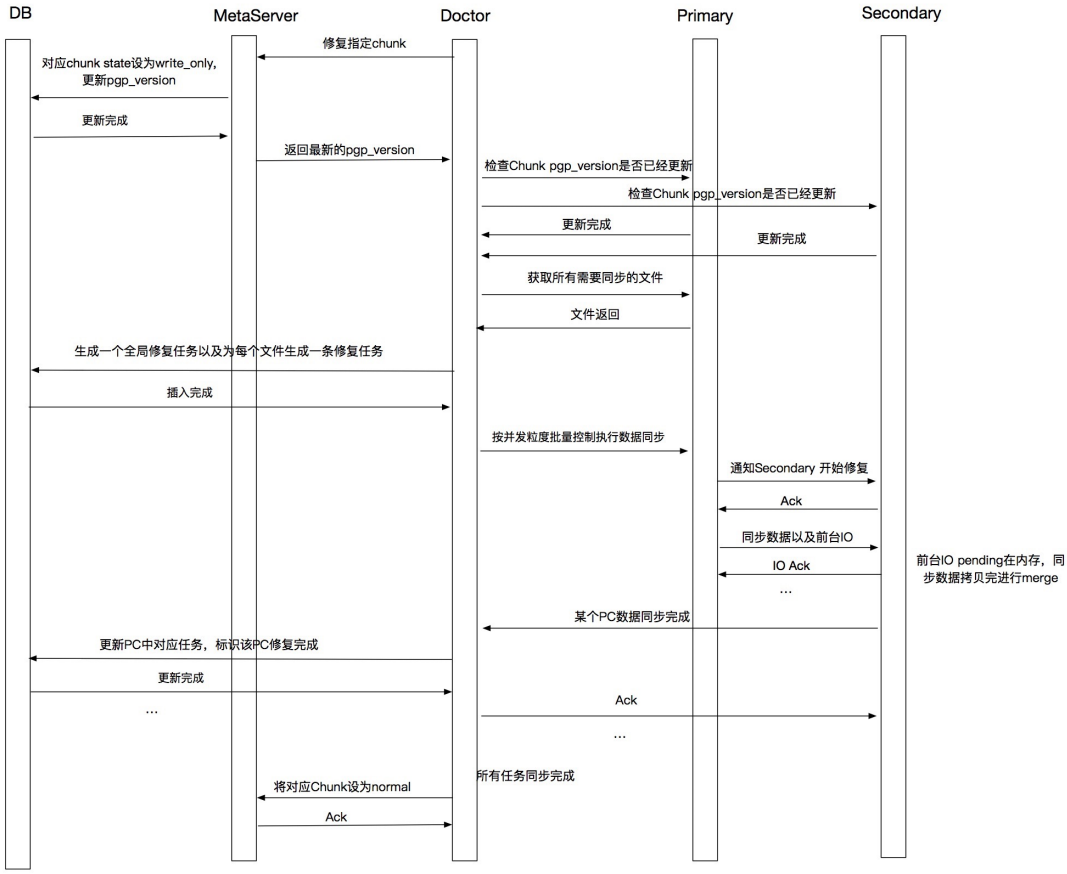
故障修复流程为:

- 1）loki 发送新chunk信息给metaserver，metaserver把新chunk加入到cluster_map, 并标记为ERROR状态.
- 2）loki服务发送修复chunk请求给metaserver，metaserver检查task表中是否有该chunk未执行完毕的修复任务，若有则直接返回异常，没有则标记该chunk为WriteOnly，并更新cluster_version.
- 3）loki服务拉取metaserver中的最新版本号，并记录.
- 4）获取新chunk所在pg的primary_chunk的cluster_version和新chunk的cluster_version, 与从metaserver处获取的最新cluster_version比较，直到与meteserver中cluster_version一致.
- 5）从新chunk所在pg的primary_chunk中拉取属于新chunk的所有文件。相同pg的文件都记录在pg_id目录下，直接遍历该目录下的文件即可。 loki服务对该chunk所在的每个pg、每个文件分别向

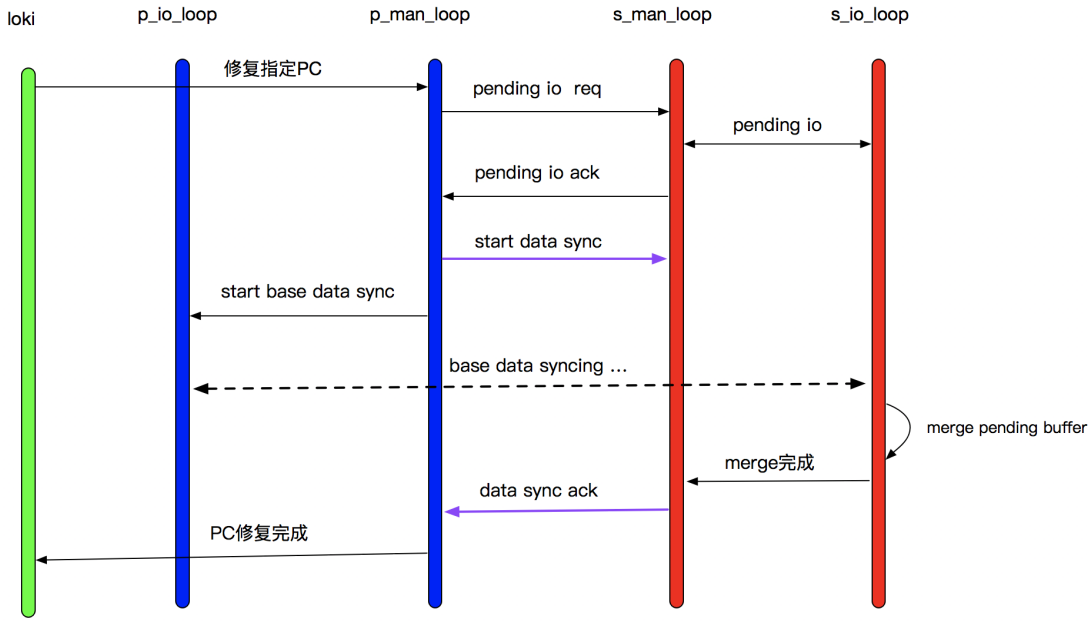
task表、job表插入修复记录。需要实现在primary chunk或者loki服务重启情况下，修复任务可以继续执行。

- 6）存量文件和新增文件处理
  在5中拉取所有chunk所属文件后，可能有新的写请求产生新增文件.
  新增文件: 新chunk是WriteOnly状态，主chunk的写请求可以同步过来，可以保证新增写请求在新chunk上和primary_chunk一致.
  存量文件: 执行下图中的数据同步流程
- 7）新chunk在修复过程中被标记下线，task表中任务无法继续执行，需要特殊处理.
- pending问题：
  a、文件被删除时，需要及时发现
  b、任意一个IO是否都可以创建文件?

PC修复控制流程：

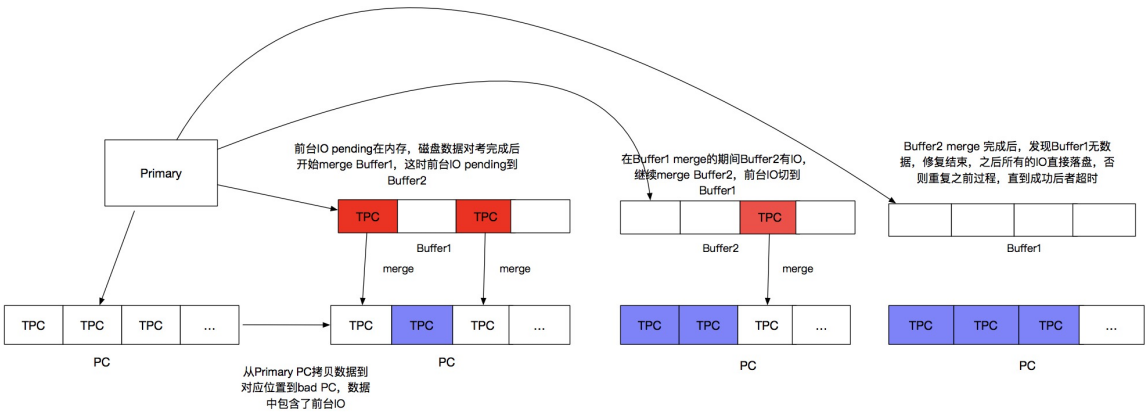


高性能架构下数据同步流程：



- 1）loki想p_man_loop通知修复指定的PC
- 2）p_man_loop 通知 s_man_loop pending前台IO，s_man_loop会同步的pending所有s_io_loop的IO
- 3）收到s_man_loop的pending io的应答后，p_man_loop随机的选择一个p_io_loop开始修复
- 4）p_man_loop通知p_io_loop 开始同步base数据
- 5）p_io_loop和s_io_loop之间进行数据同步
- 6）指定PC的base数据同步完成后，s_io_lop通知其他的s_io_loop开始merge pending buffer
- 7）s_io_loop merge完成后通知s_man_loop， s_man_loop收到所有的通知后，应答3中的请求，表示修复完成。 *注意*
- 高性能gate 每个LC的IO通过多个连接发送，会分配到多个线程，在多个线程merge时，已经应答的IO会乱序（不同于并发IO的乱序）所以在gate中不能按每个io来轮询选择线程发送。在gate中将IO按TPC切割，通过为TPC编号做hash,分配到固定的gate线程。
- 由于修复的过程中可能会发生线程切换，导致同一个TPC可能在两个线程中发生merge, 如果发生这种情况，改PC修复失败。

pending buffer合并流程：



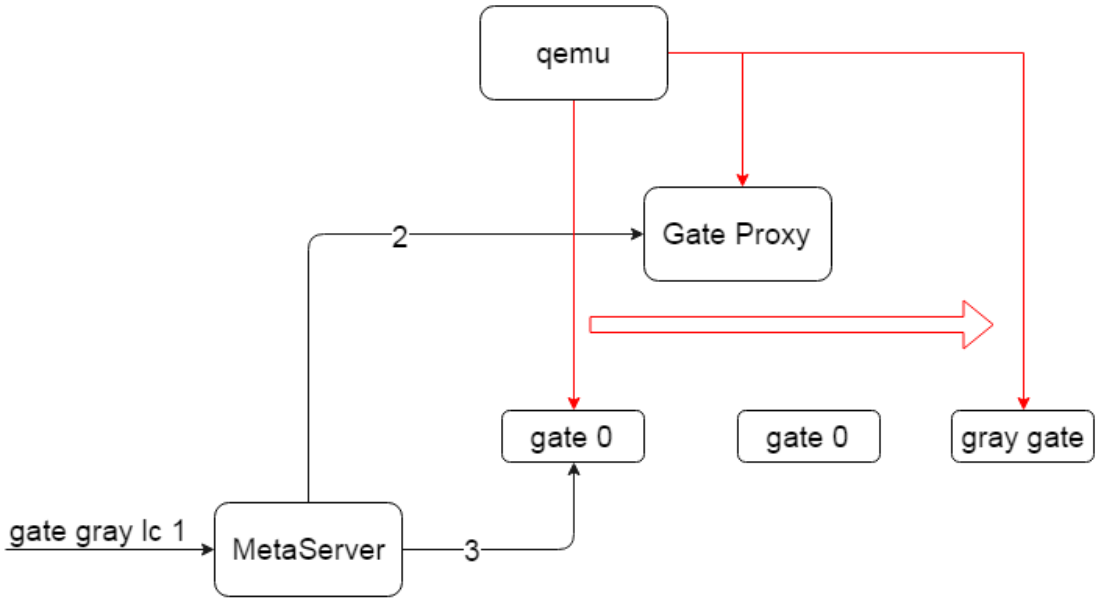
- 数据同步时，Primary的前台IO直接落磁盘，并且拷贝分片的全量数据到Secondary 分片，这种拷贝IO直接落 Secondary磁盘，但是对于由Primary转发过来的前台写IO，Secondary需要将其pending在内存的Buffer1里。

- 当拷贝结束之后，检查Buffer1有哪些TPC（Tiny PC，数据修复时更小的粒度，暂定1M）写过，没有写过的TPC上的前台IO之后可以直接落磁盘，这个TPC的同步过程结束（蓝色块标识）。
- 对于写过的TPC（红色块标识），将前台IO pending到Buffer2后，将内存Buffer1中对应的数据与磁盘数据进行merge。
- merge完所有的红色TPC后，检查Buffer2 是否有红色TPC，如果有将前台IO切回Buffer1，重复以上过程直至某次merge完成后没有红色TPC。
- 某个TPC有大量长时间的IO写时，TPC会持续红色，本次数据同步会超时失败。

灰度

GATE服务灰度

灰度目标：可以指定对应的lc到灰度的GATE服务上
灰度控制流程如下：

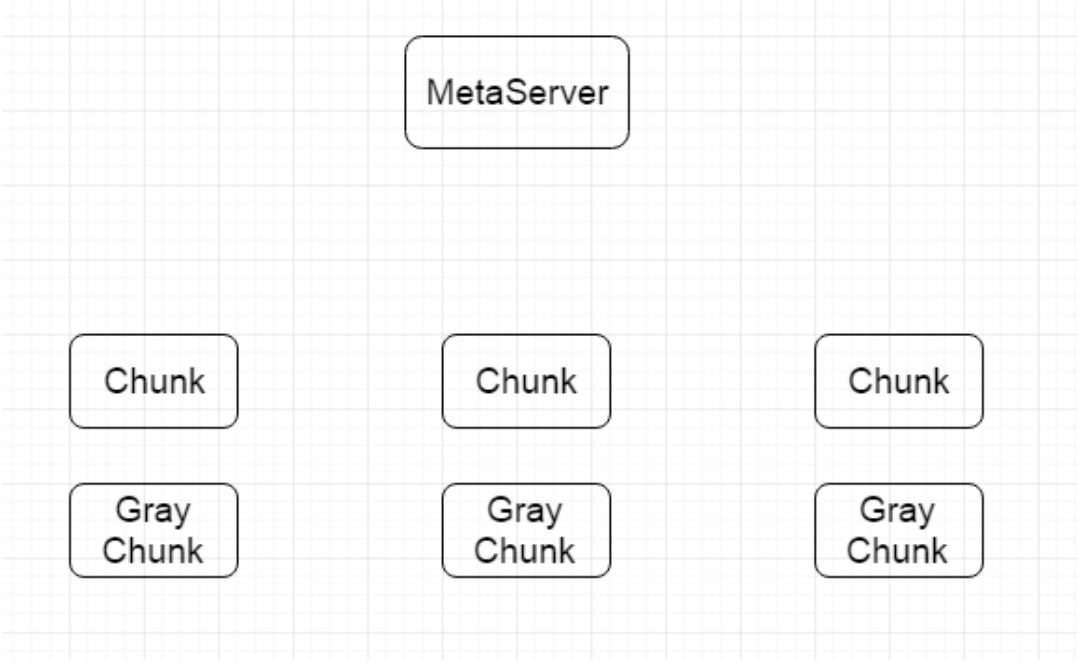


- 1） 向MetaServer请求灰度指定id的lc到灰度的Gate服务上，MetaServer收到Gate灰度请求后，通过Gate服务与MetaServer之间的heartbeat找到指定id的lc Gate服务所在的机器，从而找到对应的Gate Proxy服务（端口固定）
- 2） MetaServer向Gate Proxy服务请求灰度指定id的lc到灰度Gate服务上，Gate Proxy服务收到请求后，Gate Proxy就设置lc的连接会被转发到灰度Gate服务上,并将灰度信息持久化（当重新连接的时候也可以应用灰度规则）
- 3） MetaServer向原有的Gate服务请求其断开与Qemu的连接，使Qemu重新连接，这时如果lc连接到达Gate Proxy的时，Gate Proxy就会应用灰度规则将其转发到灰度Gate服务上

协议灰度

代码内灰度

Chunk服务灰度



Chunk服务灰度与IO

- 1）向MetaServer请求灰度一个逻辑盘,MetaServer将灰度信息持久化到数据库,并增加集群路由版本号
- 2）MetaServer与Gate服务和Chunk服务的heartbeat会将这个灰度的信息传播到Gate服务和Chunk服务上
- 3）如果Gate服务感知灰度信息后，灰度的lc的读写操作会应用灰度hash-ring,这样的话这个lc的io请求被发送到灰度的Chunk服务上

Chunk服务灰度与故障上报

Chunk服务上报的处理是相互独立的话，即gray Chunk服务故障后仅仅标记gray Chunk为故障，例如Chunk服务 A, B, C,以及灰度Chunk A', B', C'，当前A'故障，B故障已经被标记，当灰度结束切换的时候，会将B故障信息覆盖,因此故障上报的时候，需要将母Chunk服务和灰度Chunk服务都标记故障，这样的话，问题是灰度Chunk的故障会影响到正常的Chunk服务

Chunk服务灰度与物理资源管理

如果底层存储采用预分配（dd），然后由Chunk服务为LC分配物理分片。这样的话，灰度Chunk与母Chunk服务共享底层磁盘的物理分片，需要互斥的分配。初步讨论使用文件锁

Chunk服务灰度与修复逻辑

Loki修复模块会将灰度的lc的修复任务发送到灰度的Chunk服务上，由灰度的Chunk进行数据同步以及前端io的merge

性能监控模块Monitor灰度

修复模块doctor灰度

MetaServer灰度

坏盘修复对应关系

坏盘修复对应关系的目的是，便于前端展示集群的坏盘修复情况，涉及到的修改包括：

mongodb
坏盘修复对应关系复用数据库t_chunk_repair_task表，并在原表基础上扩充字段

- t_chunk_repair_task

```
原表结构：
{
    id :            修复任务ID，字符串
    bad_chunk :     新盘chunkid，  整数
    create_time :   修复任务创建时间，  整数
}


扩充后表结构：
{
    id :              修复任务ID，  字符串
    bad_chunkid :     坏盘chunkid，  整数
    bad_chunkip :     坏盘ip，  字符串
    bad_uuid :        坏盘uuid（创建文件系统后 通过blkid获取），  字符串
    new_chunkid :     新盘chunkid，   //对应原表中bad_chunk字段，  整数
    new_chunkip :     新盘ip，  字符串
    new_uuid :        新盘uuid（创建文件系统后 通过blkid获取），  字符串
    status :          修复状态：2 修复中，1 修复完成，  整数
    create_time :     修复任务创建时间，  整数
}
```

message协议修改

- LokiRepairChunkPrepareRequest

```
原协议：
message LokiRepairChunkPrepareRequest {
    required uint32 bad_chunk_id = 10;
    optional uint32 new_chunk_id = 20;
};

新协议：
message LokiRepairChunkPrepareRequest {
    required uint32 bad_chunk_id = 10;
    required uint32 new_chunk_id = 20;
    optional string bad_chunkip  = 30;
    optional string bad_uuid     = 40;
    optional string new_chunkip  = 50;
    optional string new_uuid     = 60;
    optional uint32 status       = 70;
};
```

- ChunkRepairTaskInfoPb

原协议：

```
message ChunkRepairTaskInfoPb {
  required string id = 10;
  required uint32 bad_chunk = 20;
  required uint64 create_time = 30;
};
```

新协议：

```
message ChunkRepairTaskInfoPb {
  required string id = 10;
  required uint32 bad_chunkid = 20;
  optional string bad_chunkip = 30;
  optional string bad_uuid = 40;
  required uint32 new_chunkid = 50;
  optional string new_chunkip = 60;
  optional string new_uuid = 70;
  required uint64 create_time = 80;
  optional uint32 status = 90;
};
```

limax修复脚本修改*

prepare_repair.py增加可选参数

loki修复过程中，PC修复前后集群版本号校验问题

原因:

为了处理修复过程中primary chunk变动带来的问题，所以PC开始修复、修复结束时都会向metaserver请求集群version，对比version是否相同，不相同的话终止修复。

修复过程中由主chunk向从chunk主动推送数据，推送过程中如果发生primary chunk切换，原主chunk切换为从，另一从切为主，可能造成PC数据不一致。

Loki修复Chunk过程及问题整理 (https://gitlab.ucloudadmin.com/ucsd-udisk/udisk/issues/3)
修复中发生主chunk切换 (https://gitlab.ucloudadmin.com/ucsd-udisk/udisk/issues/10)
修复验证关键指标 (https://gitlab.ucloudadmin.com/ucsd-udisk/udisk/issues/30)