HSM 与 Lemur

目录

1 HSM 引言

Lustre HSM（Hierarchical Storage Management）是 lustre 文件系统提供的一种多层级存储管理功能。该功能使得可以将 lustre 绑定到一个或者多个下级存储系统（以下简称 HSM 存储），从而使得可以将 lustre 作为一个上层的高速缓存进行使用。

lustre 所提供的 HSM 主要架构如下：

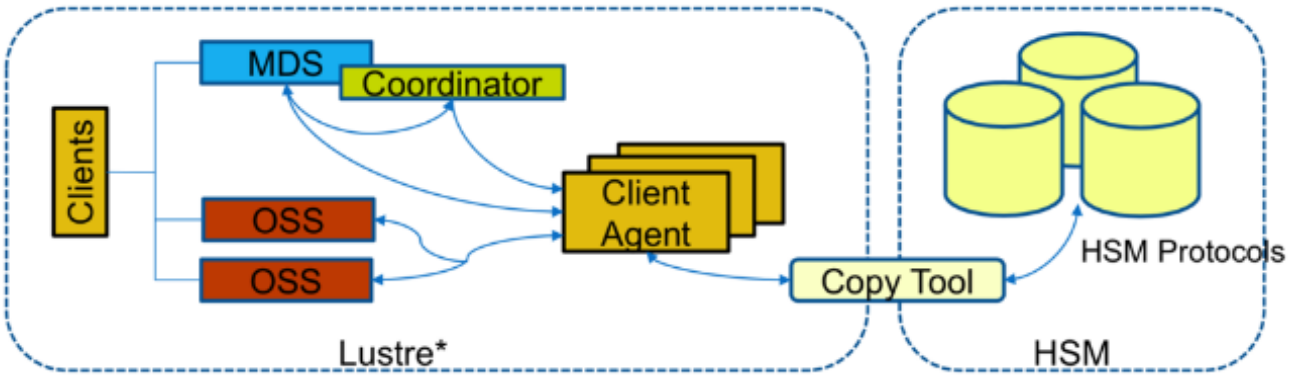

图 1: HSM 架构

lustre 的每个 mdt 都可以启动一个 hsm 服务，该服务又可称为 Coordinator，负责收集客户端发起的 hsm 请求，并按照一定的策略将该请求转发到对应的 Agent。

Agent 是运行在 lustre 客户端上的一个或者多我之前个用户进程。Agent 负责接收来自 Coordinator 的 hsm 请求，并将请求解析请求，向具体的 Copy tool 发送 HSM 命令。

Copytool 则负责执行具体的 hsm 操作。Copytool 仅响应来自于 Agent 的命令，其本质上是一个用户态进程。Copytool 是一个针对 lustre 和 hsm 存储专用设计的数据迁移工具，例如，如果 hsm 后端是对象存储，提供 S3 接口，则 Copytool 需要设计成 lustre 的 posix-to-s3 的数据迁移工具。Copytool 在具体执行时可以做一些其他操作，比如压缩、去重以及加密等。

HSM 架构可以设置一个集中式的策略引擎 PolicyEngine。PolicyEngine 是一个运行在 lustre 客户端的后台服务，负责实时监控 lustre mdt 的 changelog 等其他信息，之后根据用户指定的具体策略，向 lustre 发送 HSM 命令。

当前针对 lustre 且工业上较为成熟的 PolicyEngine 是 Robinhood

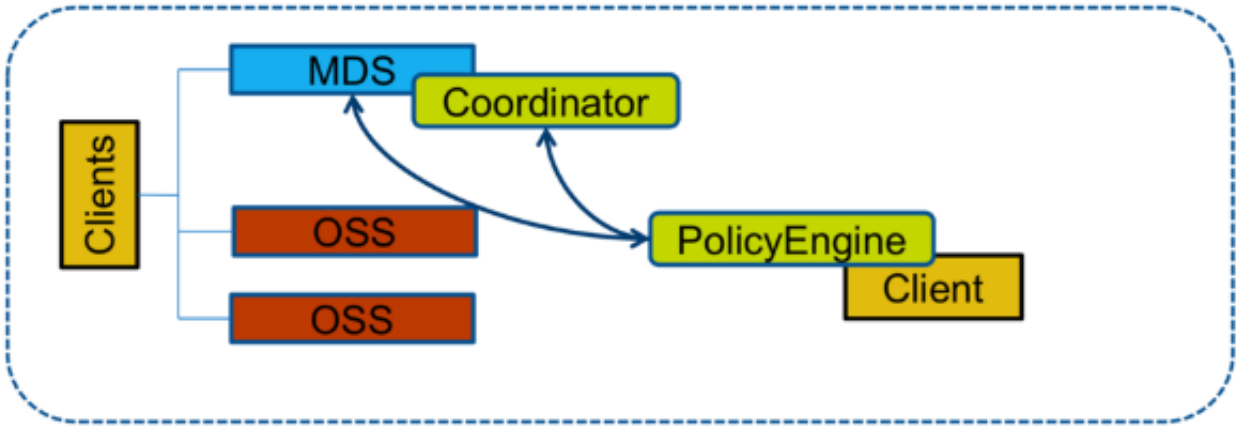

图 2: HSM PolicyEngine

2 HSM 支持的操作

每个 HSM Agent 均需要向 cdt 注册。注册 agent 时需要指定–archieve 参数。Lustre HSM 功能支持最多 32 个带有编号的 agent ID，编号从 1 开始。另外支持一个特殊的 agent，该 agent 的编号为 0，表示该 agent 可以用于服务所有的 hsm 后端。每个 ID 对应的 agent 可以注册无穷多个，lustre coordinator 会根据一定的策略从所注册的 agent 中选出一个最适合的，之后向其发送 hsm request。

HSM 的命令由客户端发起，具体的命令有以下几种：

```
1   hsm_archive [−−archieve=ID] FILE1 [FILE2···]
2   // 复制filelist中的文件下沉到HSM存储中，如果不指定−−archieve，则默认使用ID=0
3
4   hsm_release FILE1 [FILE2···]
5   // 删除lustre中对应该文件的数据部分。执行之前确保数据已被archive到HSM中
6
7   hsm_restore FILE1 [FILE2···]
8   // 从HSM中恢复文件列表中的数据到lustre中
9
10  hsm_cancel FILE1 [FILE2···]
11  // 取消针对文件列表正在执行的HSM请求
12
13  hsm_action FILE [···]
14  // 显示针对文件列表中的文件正在执行的hsm请求
15
```

```
16    hsm_state FILE [···]
17    // 显示文件列表中的HSM状态
```

当前支持的状态有以下几种:

state	explanation	annotation
Non-archive	标识为该状态的文件将不会被 archive	
Non-release	标识为该状态的文件将不会被 release	
Dirty	存储在 lustre 中的文件以被更改，需要执行 archive 操作以便更新 HSM 中的副本	
Lost	该文件已被 archive，但是存储的 HSM 中的副本已丢失	打该标签时需要以 root 权限
Exist	文件的数据拷贝在 HSM 中已经存在	
Rleased	文件的 LOV 信息以及 LOV 对象已从 lustre 中移除	
Archived	文件数据已整个从 lustre 中拷贝至 HSM 中	

表 1: HSM 支持的状态

```
1     hsm_set [FLAGS] FILE [···]
2     // 将文件列表中的文件打上上述标签
3
4     hsm_clear [FLAGS] FILE [···]
5     // 移除文件列表中相应的标签
6
7     hsm_import path [flags···]
8     // 在lustre中创建文件存根，路径为path，
9     // 其他信息由flags指定。文件的hsm状态为released，
10    // 即文件的内容存储在后端存储，
11    // 仅当文件被打开或者用户主动调用restore时文件的内容才被加载
```

3 HSM 操作流程

3.1 文件的状态的转换

一个 HSM 请求总是由 lustre 客户端发起。当客户端创建一个新的文件时，可以执行 archive 操作，此时该文件将被拷贝到 HSM 后端存储，并被标记为 archived 状态。当客户端针对该文件执行 release 操作时，文件的数据将从 lustre 中删除，此时 lustr 仅存储该文件的元数据信息且文件将被标记为 released 状态。当客户端针对该文件执行 restore 操作时，该文件的数据将从 hsm 中拷贝到 lustre 中，且文件的状态被标记为 archived 状态。当 lustre 中的文件被修改时，文件将被标记为 dirty 状态，此时客户端需要再次执行 archive 操作，以保证 HSM 中的数据与 lustre 中的数据同步。

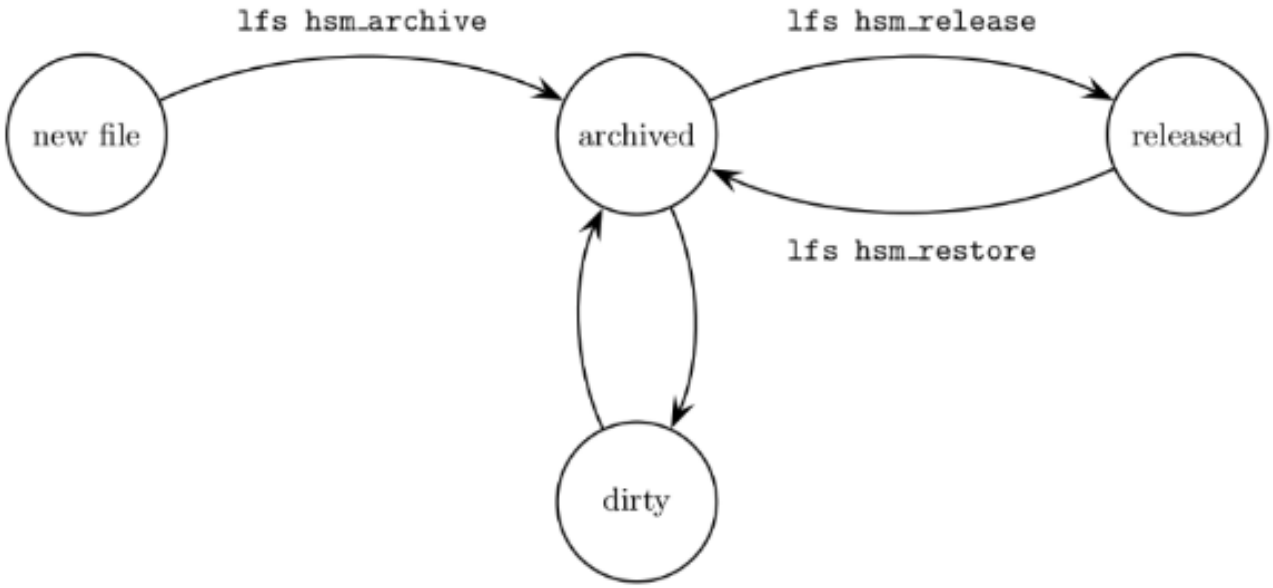

图 3: 文件的状态的转换

3.2 lustre 的 HSM 架构设计

lustre 的 HSM 旨在提供一套多层存储的操作接口，并不提供接口的具体实现。在 lustre 的 HSM 中，文件的数据可以从 lustre 的 OST 中迁移到 HSM 存储中，但是文件的元数据信息，如文件名、用户 ID、权限信息等将会一直保存在 lustre 的 MDT 中。当客户端打开一个已被 release 的文件或者主动执行 restore 操作时，lustre 将自动从 HSM 中重新加载文件对应的数据。restore 的整个过程如如下：

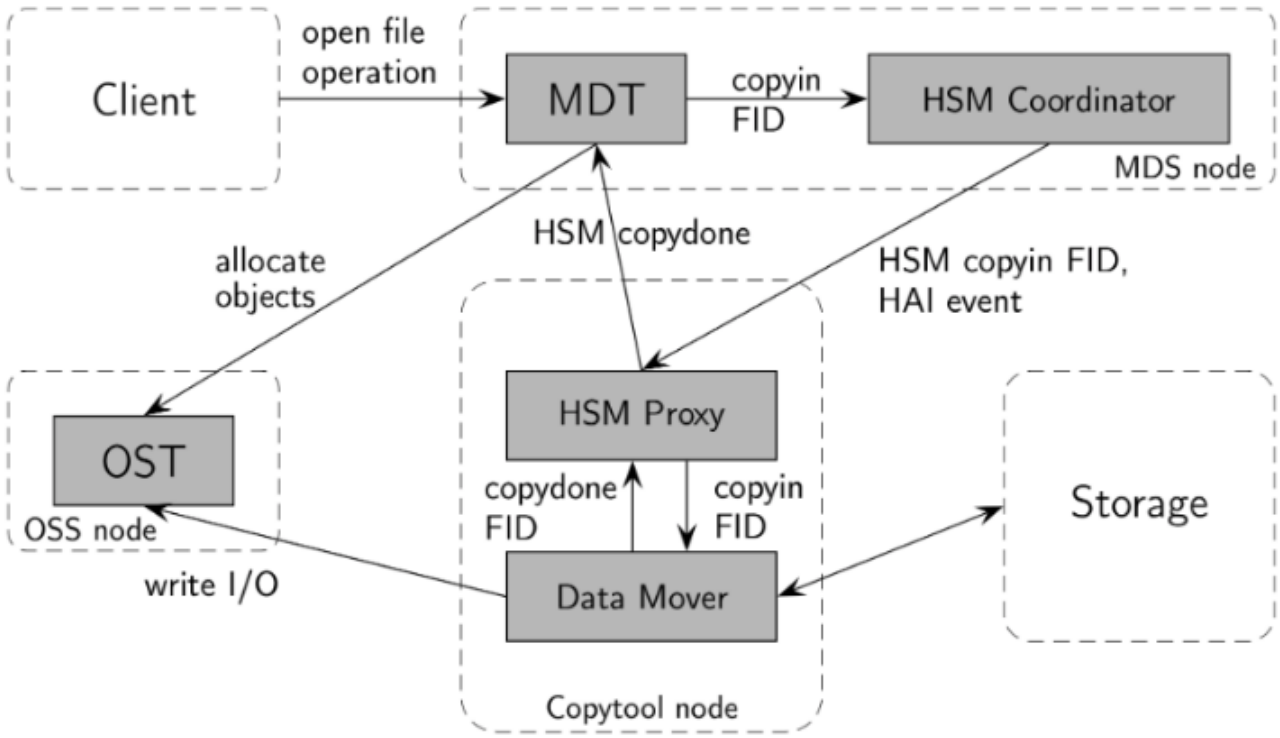

图 4: HSM 架构设计

客户端打开或者主动执行 restore 操作，该请求将首先被发送到相关的 MDT 服务中。MDT 检测到文件的相应数据不在 lustre 中，将首先在 OST 中为该文件分配对应的存储空间，之后将文件的 FID 发送到 HSM 的 CDT 服务中。CDT 服务解析该请求，并将该请求进一步封装发送到注册到该 MDT 中的合适的 agent 中。agent 服务中产生对应的 HSM 事件，并从事件中解析到请求类型以及请求关联的 FID。agent 服务之后将请求进一步封装并发送到对应的 Data Mover 进程中。Data Mover 服务具体的数据搬运工作，同时还可以进一步做一些数据的压缩、去重、加密等操作。在数据的搬运过程中，Data Mover 可以向 agent 汇报数据搬运的进度，此时 agent 亦可将该进度汇报给 CDT 服务。数据搬运完成后 Data Mover 将向 agent 汇报数据搬运完成事件，之后 agent 将该事件进一步封装并汇报给 MDT 服务。MDT 收到数据搬运完成的事件后将通知 client 文件打开成功。至此整个文件的 restore 完成。在此期间，客户端将一直处于阻塞状态。

更一般的 hsm 请求控制流如下:

1. 客户端调用llapi_hsm_request()创建hsm请求。
2. lustre client通过mdc_ioc_hsm_request()将该请求发送到mdt。
3. mdt创建相应的hsm请求日志llog，并唤醒cdt服务。
4. cdt服务被唤醒，从llog读取对应的日志内容，创建具体的HSM事件，并将事件发送到agent所在的客户端。
5. agent接收到hsm事件，并处理事件中封装的具体hsm请求
6. agent可以随时调用llapi_hsm_action_progress()向CDT汇报数据搬运的进度。

7	7. 数据搬运完成后agent调用llapi_hsm_action_end()用于汇报mdt数据已搬运完成。

3.3 lustre MDT 提供的 HSM 调节参数

lustre 的 mdt 参数项中包含一些关于 hsm 的参数。参数路径为/proc/fs/lustre/mdt/<fsname>/

参数项	属性	释义	备注
hsm_control	rw	控制 cdt 服务的启停，可选配置如下: enabled: 开启 cdt 服务线程 disabled: 暂停 cdt 一切服务 shutdown: 关闭 cdt 服务，释放所有 cdt 资源 purge: 清除所有的 hsm 请求	
hsm/actions	r		
hsm/active_request_timeout	rw	一个 active 请求执行的最大超时时间	
hsm/active_requests	r		
hsm/agents	r	查看当前已经注册的所有 agent	
hsm/archive_count	r	当前已经执行 archive 的统计计数	
hsm/default_archive_id	rw	cdt 默认使用的 archive ID	
hsm/grace_delay	rw	一个成功的或者失败的 hsm 请求被从请求列表中移除的最长时间	
hsm/group_request_mask	rw	组访问权限相关	
hsm/user_request_mask	rw	用户访问权限相关	
hsm/loop_period	rw	hsm CDT 扫描 llog 的周期	
hsm/max_requests	rw	每个 mdt 最大的 active 请求数	
hsm/policy	rw	hsm 处理策略， 有以下两种：NRA：No Retry Action, 如果 restore 失败，则不会再次执行 NBR：Non Blocking Restore，如果访问一个已经被 released 的文件，将不会出发自动 restore 操作。	
hsm/remove_archive_on_last_unlink	rw	lustre 文件中的数据被 rm 后是否发出 hsm 请求删除 hsm 存储中对应的数据。	
hsm/remove_count	r		
hsm/restore_count	r		

表 2

3.4 lustre 的 HSM 接口

lustre 向外部导出 C 语言的 API 接口，主要的接口定义如下:

4 获取 HSM 状态

4.1 HSM 状态

这个 struct 是为了显示当前 Lustre files 状态，描述了当前的 HSM 状态和进程中的行为。
其结构与成员描述如下：

```
1  struct hsm_user_state {
2      /* hus_states和hus_archive_id表示当前HSM状态 */
3      // hus状态
4      __u32   hus_states;
5      // hus归档ID
6      __u32   hus_archive_id;
7      /* 若当前存在 HSM 行为
8          则用如下的四个元素：
9          hus_in_progress_state, hus_in_progress_action,
10         hus_in_progress_location和hus_extended_info数组
11         描述该行为
12     */
13     // hus进程中状态
14     __u32   hus_in_progress_state;
15     // hus进程中行为
16     __u32   hus_in_progress_action;
17     // hus进程中位置
18     struct hsm_extent hus_in_progress_location;
19     // hus扩展信息
20     char    hus_extended_info[];
21 };
```

4.2 通过文件标识符获取 hsm 状态

```
1  /* 此API的目的是返回HSM状态和HSM请求
2  API调用方给出一个文件标识符fd
3  和一个hsm_user_state结构的指针hus
4  然后通过hus指针返回该HSM状态。*/
```

```
5    int llapi_hsm_state_get_fd(int fd, struct hsm_user_state *hus);
```

若该 API 返回 int 值为 0 则表示操作成功，为负则报错，负值为出错码。

```
1    /* 此API中，主要是调用对I/O通道进行管理的ioctl函数 */
2    rc = ioctl(fd, LL_IOC_HSM_STATE_GET, hus);
```

而传入该 ioctl 函数中的为文件标识符 fd
和 CMD 命令 LL_IOC_HSM_STATE_GET，此 CMD 命令参数为 hus

```
1    #define LL_IOC_HSM_STATE_GET  _IOR('f', 211, struct hsm_user_state)
2    /* 如上的define语句可看出，
3    LL_IOC_HSM_STATE_GET命令实际为IO read命令
4    传入ioctl的参数hus的作用为传回一个HSM状态 */
```

4.3 通过文件路径获取 HSM 状态

```
1    /* 此API的目的是返回HSM状态和HSM请求
2    API调用方给出一个文件路径path
3    和一个hsm_user_state结构的指针hus
4    然后通过hus指针返回该HSM状态。 */
5    int llapi_hsm_state_get(const char *path, struct hsm_user_state *hus);
```

而此 API 的具体实现是：

```
1    // 先通过路径名获取fd文件标识符
2    fd = open(path, O_RDONLY | O_NONBLOCK);
3    // 再调用上述的get_fd函数获取hsm状态
4    rc = llapi_hsm_state_get_fd(fd, hus);
```

5 设置 HSM 状态

5.1 通过文件标识符设置 hsm 状态

```
/* 此API的目的是通过文件标识符fd设置HSM状态*/
int llapi_hsm_state_set_fd(int fd, ___u64 setmask, ___u64 clearmask, ___u32 archive_id);
```

该函数会新建一个如下的 hsm state set 结构的指针 hss：

```
struct hsm_state_set {
    ___u32 hss_valid;
    // hss_valid初始设置为0x03
    ___u32 hss_archive_id;
    /* 如果传入的archive_id大于0,则hss_valid设置为0x07,
    并且hss_archive_id设置为传入archive_idhss_valid */
    ___u64 hss_setmask;
    // hss_setmark设置得与传入参数一致
    ___u64 hss_clearmask;
    // has_clearmask设置得与传入参数一致
};
```

之后调用 ioctl 函数：

```
    rc = ioctl(fd, LL_IOC_HSM_STATE_SET, &hss);
    // 其中CMD命令为：
    LL_IOC_HSM_STATE_SET
    // 此命令实际为：
    #define LL_IOC_HSM_STATE_SET   _IOW('f', 212, struct hsm_state_set)
    // 它实际上使用IO write方法
    // 将hss指向的的hsm state set写入文件标识符fd所指的文件中
```

5.2 通过文件路径设置 hsm 状态

```
    /* 此API的目的是通过文件路径path设置HSM状态*/
    int llapi_hsm_state_set(const char *path, ___u64 setmask, ___u64 clearmask, ___u32
        archive_id);
```

此 API 中重要步骤为:

```
1    /* 此函数通过open函数打开文件, 找到文件标识符fd */
2    fd = open(path, O_WRONLY | O_LOV_DELAY_CREATE | O_NONBLOCK);
3    /* 然后调用3.1中的函数修改hsm状态 */
4    rc = llapi_hsm_state_set_fd(fd, setmask, clearmask, archive_id);
```

5.3 HSM 日志报错

```
1    /* 此API的目的是通过文件标识符fd设置HSM状态
2    当一段监控FIFO队列被注册后，自定义的日志回调被调用。
3    将日志条目格式化成Json事件格式，这样可以被copytool监控进程所消费。*/
4    void llapi_hsm_log_error(enum llapi_message_level level, int _rc, const char *fmt, va_list
         args);
```

此 API 的输入为:

```
1    // enum的llapi_message_level有几种值
2    enum llapi_message_level {
3        LLAPI_MSG_OFF = 0, // OFF状态位为0
4        LLAPI_MSG_FATAL = 1, // FATAL状态位为1
5        LLAPI_MSG_ERROR = 2, // ERROR状态位为2
6        LLAPI_MSG_WARN = 3, // WSRN状态位为3
7        LLAPI_MSG_NORMAL = 4, // NORMAL状态位为4
8        LLAPI_MSG_INFO = 5, // INFO状态位为5
9        LLAPI_MSG_DEBUG = 6, // DEBUG状态位为6
10       LLAPI_MSG_MAX
11   };
12
13   int _rc // rc设置返回消息的代码
14   const char *fmt // fmt格式化字符串
15   va_list args // args待格式化的字符串
```

6 HSM 事件队列

6.1 注册 HSM 事件队列

```
1  /* 此API的目的是通过文件路径path注册HSM事件队列
2  该事件队列是为了监控进程可以从中读取
3  而在没有读取器读取时，写入事件将丢失
4  */
5  int llapi_hsm_register_event_fifo(const char *path);
```

此 API 的重要实现为：

```
1  /* 从路径中读取(若读取不存在则新建)一个FIFO */
2  mkfifo(path, 0644)
3  /* 先以读取打开文件, 为了防止写指令不会马上失效 */
4  read_fd = open(path, O_RDONLY | O_NONBLOCK);
5  /* 再将HSM事件文件标识符指向该文件
6       以非阻塞方式写入该队列 */
7  llapi_hsm_event_fd = open(path, O_WRONLY | O_NONBLOCK);
```

6.2 注销 HSM 事件队列

```
1  /* 此API的目的是通过文件路径path注销HSM事件队列*/
2  int llapi_hsm_unregister_event_fifo(const char *path);
```

此 API 的重要实现为：

```
1  /* 关闭队列的读取与写入 */
2  close(llapi_hsm_event_fd)
3  // 取消与path的连接
4  unlink(path);
5  // 设置HSM事件队列的存在标志位为否
6  created_hsm_event_fifo = false;
7  // 设置HSM事件文件夹标识符为−1
8  llapi_hsm_event_fd = −1;
```

7 HSM copytool

7.1 copytool 结构

该结构体如下：

```
1  struct hsm_copytool_private {
2      int     magic;
3      char    *mnt; // 代指Lustre的mount point
4      struct kuc_hdr   *kuch; // 指向的是-> KUC信息头
5      // 所以当前和未来的KUC信息必须包含在此头内
6      // 目的是为了避免不得不去包含Lustre的libcfs头文件
7      int     mnt_fd; // mount所对接的文件
8      int     open_by_fid_fd;
9      struct lustre_kernelcomm *kuc; // 指向的是-> kernelcomm控制结构
10     // 从用户态到内核态
11     // 内核以存档的位图方式读取lk_data_count
12 };
```

7.2 copyaction 结构

该结构体如下：

```
1  struct hsm_copyaction_private {
2      __u32      magic;
3      __u32      source_fd; // 源文件标识符
4      __s32      data_fd; // 数据标识符
5      const struct hsm_copytool_private *ct_priv; // 指向copytool的指针
6      struct hsm_copy     copy;
7      lstat_t      stat;
8  };
```

7.3 注册 copytool

具体 API 如下:

```
/* 此API的目的是注册一个copytool
若返回值为0说明成功注册，返回值为负则出错 */
int llapi_hsm_copytool_register(struct hsm_copytool_private **priv,
const char *mnt, int archive_count, int *archives, int rfd_flags);
/*API调用方给出一个hsm_copytool_private结构的二重指针priv,
一个Lustre的mount point的路径mnt,
一个有效归档总数archive_count,
一个copytool负责的归档集号码archives,
和适用于通道读取文件的标志rfd_flags */
```

7.4 注销 copytool

具体 API 如下:

```
/* 此API的目的是注册一个copytool
若返回值为0说明成功注册，返回值为负则出错 */
int llapi_hsm_copytool_unregister(struct hsm_copytool_private **priv);
// 注销hsm_copytool_private结构的二重指针priv所指向的copytool
```

7.5 获取 copytool 的文件标识符

具体 API 如下:

```
/* 此API的目的是获取copytool的文件标识符*/
int llapi_hsm_copytool_get_fd(struct hsm_copytool_private *ct);
/* 输入一个不透明的 hsm_copytool_private结构的指针ct
    返回ct->kuc->rfd, 即返回kuc的读取文件标识符 */
```

8 HSM 活动列表

8.1 接收活动列表

```
1  /*
2  此API的目的是让copytool接收活动列表
3  若返回值为0证明执行成功，返回负数则说明接收数据失败
4  */
5  int llapi_hsm_copytool_recv(struct hsm_copytool_private *priv,
6      struct hsm_action_list **hal, int *msgsize);
7  /*
8  输入当前的copytool−−−>ct， 和一个活动表的指针−−−>hal，
9  以及一个信息的bytes数量−−−> msgsize.
10 按照bytes数量msgsize读取hal指向的活动表
11 并将其传给priv指向的copytool
12 */
```

8.2 开始执行 hsm 活动表

```
1  /*
2  此API的目的是让copytool开始执行活动列表
3  此API必须在开始执行请求前，被copytool所调用
4  若copytool仅仅想报一个错，则跳过此程序，跳转到llapi_hsm_action_end()
5  若返回值为0证明执行成功，返回负数则说明执行失败
6  */
7  int llapi_hsm_action_begin(struct hsm_copyaction_private **phcp,
8      const struct hsm_copytool_private *ct,const struct hsm_action_item *hai,
9      int restore_mdt_index, int restore_open_flags,bool is_error);
10 /*
11 通过hcp的选择，该操作跳到llapi_hsm_action_progress() 和 llapi_hsm_action_end()
12 ct指向copytool注册所需的hsm私有结构
13 hai描述当前请求
14 restore_mdt_index: 用于创建易变文件的MDT index，使用−1为默认值
15 restore_open_flags: 易变文件创建模式
16 is_error: 此次调用是否是仅仅为了返回一个错误
```

```
17      */
```

8.3 终止 HSM 活动表

具体 API 如下：

```
1       /*
2       此API的目的是终止一个HSM活动进程
3       该API只有在copytool终止处理所有请求后，才可被调用
4       若返回值为0证明成功终止HSM活动进程，返回负数则说明终止进程失败
5       */
6       int llapi_hsm_action_end(struct hsm_copyaction_private **phcp,
7               const struct hsm_extent *he, int hp_flags, int errval);
8       /*
9       he:  被复制数据最终的范围
10      errval: 操作的状态码
11      hp_flags: 终止状态的标志位
12      */
```

8.4 通知 HSM 进程

具体 API 如下：

```
1       /*
2       此API的目的是在处理一个HSM行为时，通知一个进程
3       若返回值为0证明成功，返回负数则说明失败
4       */
5       int llapi_hsm_action_progress(struct hsm_copyaction_private *hcp,
6               const struct hsm_extent *he, __u64 total, int hp_flags);
7       /*
8       hcp 指向的是当前的复制行为
9       he:  被复制数据的范围
10      total: 所需全部复制数据
11      hp_flags: HSM进程标志位
12      */
```

8.5 获取被复制对象的文件标识符

具体 API 如下：

```
/*
此API的目的是获取被复制对象的文件标识符
该文件标识符是通过传入的指针fid返回
返回0如果操作成功，返回错误码如果此行为不是一个复制行为
*/
int llapi_hsm_action_get_dfid(const struct hsm_copyaction_private *hcp, struct lu_fid *
    fid);
/*
hcp 为复制行为的指针
fid为传入的指针，用于返回被复制对象的文件标识符
*/
```

该 API 会创建一个 HSM 活动项目：

```
/* 该结构是为了描述copytool项目行为的 */
struct hsm_action_item {
    __u32 hai_len; /* 结构的合理大小 */
    __u32 hai_action; /* 用已知的大小构建的copytool行为 */
    struct lu_fid hai_fid; /* 需要去操控的Lustre文件标识符 */
    struct lu_fid hai_dfid; /* 用于获取数据的文件标识符*/
    struct hsm_extent hai_extent; /* 需要操控的字节范围 */
    __u64 hai_cookie; /* 协调器的行为cookie */
    __u64 hai_gid; /* 组锁ID */
    char hai_data[0]; /* 可变长度 */
} __attribute__((packed));
```

```
// 若是以下两个条件不满足，则报错
if (hcp->magic != CP_PRIV_MAGIC)
    return -EINVAL;
if (hai->hai_action != HSMA_RESTORE && hai->hai_action != HSMA_ARCHIVE)
    return -EINVAL;
// 上面创建的copytool项目行为的dfid, 就是待返回的dfid
*fid = hai->hai_dfid;
```

8.6 获取文件描述符

具体 API 如下：

```
/*
此API的目的是获取文件描述符用于复制数据
该文件描述符是通过传入的copyaction指针返回
返回0如果操作成功，返回负的错误码如果操作失败
*/
int llapi_hsm_action_get_fd(const struct hsm_copyaction_private *hcp);
```

该 API 会创建一个 HSM 活动项目：

```
/* 该结构是为了描述copytool项目行为的 */
struct hsm_action_item {
    __u32 hai_len; /* 结构的合理大小 */
    __u32 hai_action; /* 用已知的大小构建的copytool行为 */
    struct lu_fid hai_fid; /* 需要去操控的Lustre文件标识符 */
    struct lu_fid hai_dfid; /* 用于获取数据的文件标识符*/
    struct hsm_extent hai_extent; /* 需要操控的字节范围 */
    __u64 hai_cookie; /* 协调器的行为cookie */
    __u64 hai_gid; /* 组锁ID */
    char hai_data[0]; /* 可变长度 */
} __attribute__((packed));
```

```
// 如果magic不对，报错
if (hcp->magic != CP_PRIV_MAGIC)
    return -EINVAL;
// 如果活动为Archive或Restore并且fd<0，则报错
if (hai->hai_action == HSMA_ARCHIVE) {
    fd = dup(hcp->source_fd);
    return fd < 0 ? -errno : fd;
} else if (hai->hai_action == HSMA_RESTORE) {
    fd = dup(hcp->data_fd);
    return fd < 0 ? -errno : fd;
} else {
    return -EINVAL;
}
```

8.7 引入已存在的 HSM 归档文件

具体 API 如下：

```
int llapi_hsm_import(
    const char *dst, // dst指向Lustre目的地路径
    int archive, // archive指向档案数
    const struct stat *st, // st包含文件所属关系，置换等
    // stripe:条目选项，当前默认忽视。
    unsigned long long stripe_size,
    int stripe_offset,
    int stripe_count,
    int stripe_pattern,
    char *pool_name,
    struct lu_fid *newfid // newfid: 输入新的Lustre文件的fid，作为输出
);
```

调用完此 API 后，调用者必须用获得的新文件标识符去获取文件

9 HSM 用户界面

9.1 分配 HSM 用户请求

具体 API 如下:

```
1  /*
2  此API的目的是用特定的特征去分配HSM用户请求
3  如果返回一个被分配的结构则成功，否则失败。
4  */
5  struct hsm_user_request *llapi_hsm_user_request_alloc(int itemcount, int data_len);
```

此结构应该能使用 free() 函数进行释放。

9.2 发送 HSM 请求

具体 API 如下:

```
1  /*
2  此API的目的是用特定的特征去分配HSM用户请求
3  返回0表示发送成功，负数表示发送失败
4  */
5  int llapi_hsm_request(
6      const char *path, // 待操作文件的全路径
7      const struct hsm_user_request *request // 需要向Lustre发送的用户请求
8  );
```

9.3 返回当前 HSM 请求

```
1  /*
2  此API的目的是通过被path所指的文件，返回其当前的HSM请求
3  返回0表示发送成功，负数表示发送失败
4  */
5  int llapi_hsm_current_action(
6      const char *path, // 通过path获取所指的文件
```

```
7           struct hsm_current_action *hca // 被调用者所分配，指向当前文件行为
8       );
```

10 Lemur 总体架构分析

Lemur 是一种遵循 lustre HSM 架构的跨层冷热数据实现方案。lustre 提供一套针对 HSM 相关的用户调用接口，由用户自定义实现满足业务需求的冷热数据迁移方案。Lemur 与 Lustre 的关系图如下：

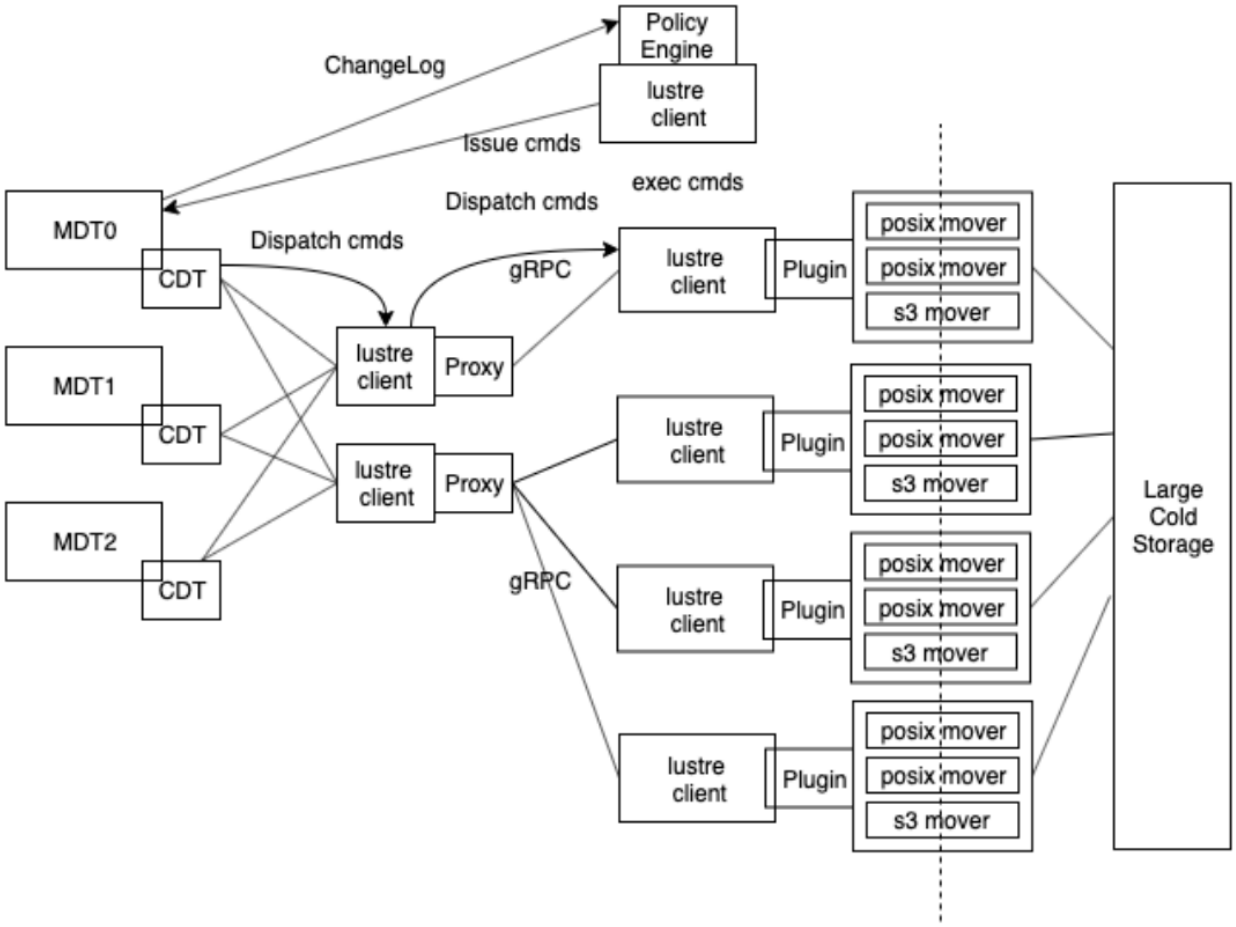

图 5: Lemur 总体架构

上图中，对于每个 MDT 可以开启相应的 HSM 功能。开启 lustre 的 HSM 功能时，在会相应的 MDT 上启动一个 Coordinator 的守护进程，用于处理和派发 HSM 命令。Lemur 在选定的 lustre 客户端上通过调用 lustre 的开发接口向所有开启 HSM 功能的 MDT 上注册一个或者多个 Proxy。Proxy 用于轮训来自 MDT 的 HSM 事件并接收其命令，之后将该命令按照一定的策略派发到相应的 Copytool 服务中。Copytool 与 Proxy 通过 gRPC 连接，其所为 gRPC 的客户端存在，负责执行相应的 HSM 命令。

每个 MDT 可以注册多个 Proxy，MDT 会按照一定的最佳选择策略将 HSM 命令派发到某个 Proxy。每个 Proxy 可以连接多个 Copytool，Proxy 将 HSM 命令按照 Archive ID 与 Copytool

映射表，将 HSM 命令再次派发到对应的 Copytool 进行执行。每个 Proxy 最多可注册 32 个不同 Archive ID 的 Copytool。Copytool 执行具体的 HSM 命令时可以向 Proxy 返回其执行进度，以方便 Proxy 确认命令是否按时按需执行。

每个 Copytool 同样运行在一个 lustre 客户端上，其包括 Plugin 和 DataMover 两个主要模块。Plugin 是对 DataMover 集合的一种抽象的表示，每个 DataMover 作为 PluginClient 由 Plugin 模块进行管理，且 DataMover 可以按照 pluginClient 的接口由用户进行自定义。当前 Lemur 支持 posix-to-posix 的 DataMover 以及 posix-to-s3 的 DataMover。顾名思义，不同的 DataMover 应对不同的 HSM 后端。posix DataMover 可以将冷数据存储到支持 posix 接口的冷数据存储中，而 s3 DataMover 可以将冷数据存储到支持 s3 接口的冷数据存储中。

除此之外，lustre 的 HSM 架构中，还有一个 PolicyEngine 模块是必不可少的。PolicyEngine 是运行在一个或者多个 lustre 客户端上的 HSM 策略服务，其通过 lustre MDT 导出的 ChangeLog，收集存储在 lustre 中的元数据以及用户操作日志，并将这些日志存储到后端大数据分析平台中。这些数据会被用于分析当前 lustre 存储系统的状态和数据的分布及活动状况，通过用户定义 HSM 策略，执行相应的 HSM 命令。比如，可以将超过 10 天未被访问的某些类型的文件下沉到冷数据存储中；可以根据当前业务的需求按照某种数据预期策略提前将需要的数据预取到 lustre 中等操作。

11 Proxy 模块分析

Proxy 充当 HSM 命令派发与 Copytool 管理的功能，其主要架构如下:

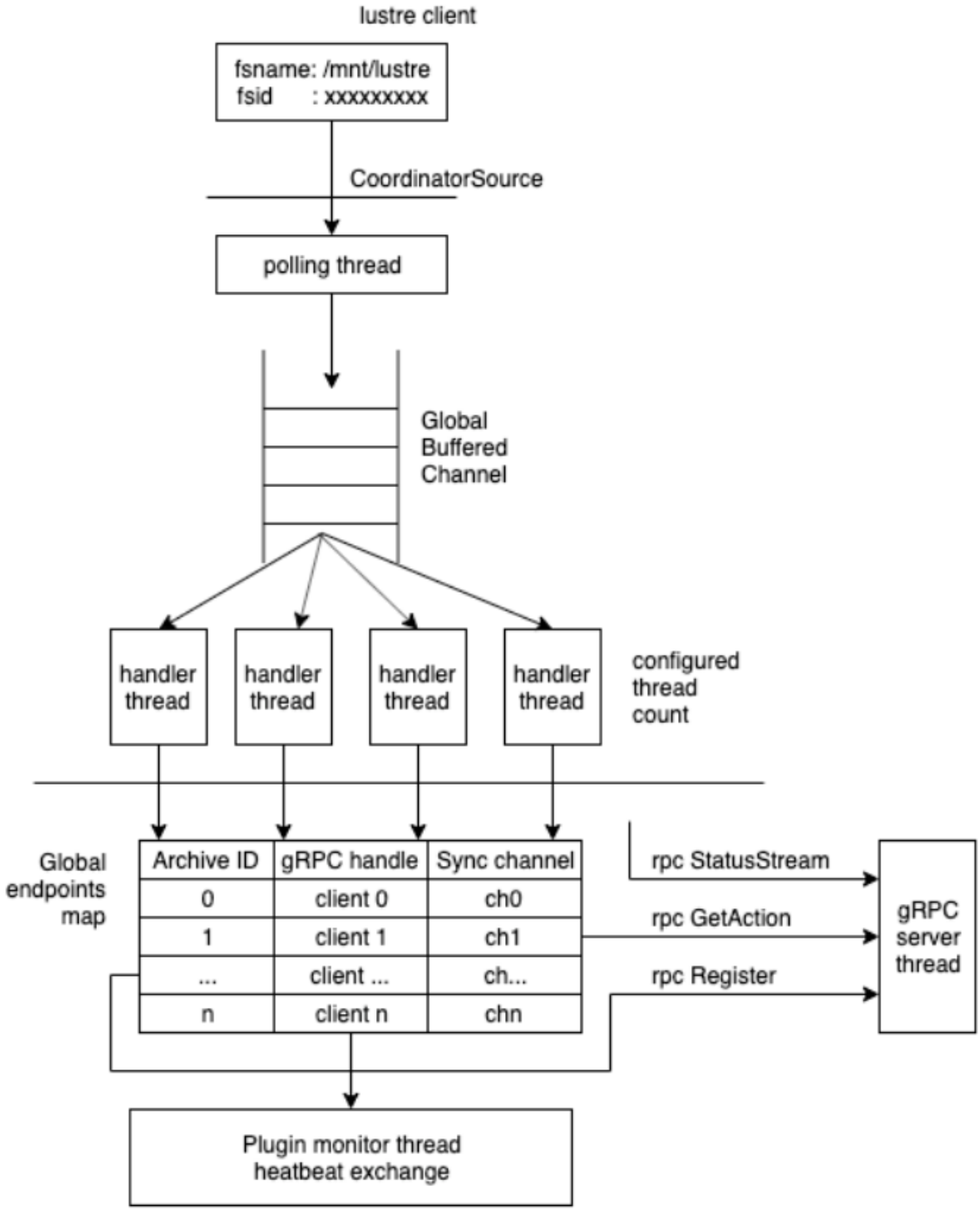

图 6: Proxy 模块

Proxy 启动时首先加载配置文件，之后根据配置文件挂载 lustre 客户端，并创建以及注册 lustre

HSM 的客户端。该客户端同时又作为 gRPC 的服务端，管理 DataMover 客户端以及 HSM 命令的派发。创建 HSM 客户端之后，proxy 会启动 HSM poll 线程以同步的方式接收来自 lustre 的 HSM 命令，并将这些命令放入一个全局队列中。之后启动一个或多个 handler 线程用于从全部队列中取出 HSM 命令，并放入对应的客户端的 sync channel 中。HSM 同时会启动一个 Plugin 监控线程以及一个 gRPC 服务线程。Plugin 监控线程用于周期性的确认每个 DataMover 实例是否工作正常，以及服务不可用时的处理办法。gRPC 服务线程用于为 gRPC 客户端提供服务。gRPC 服务启动时，会向外部导出三个服务接口：

11.1 rpc Register

该 rpc 接口由 DataMover 调用，用于向 Proxy 注册一个 DataMover 实例。每个注册的 DataMover 实例都会按照 Archive ID 为索引的全部 map 中。每个 DataMover 实例在一个 Proxy 中必须唯一。通过 Archive ID 可以索引到对应的 gRPC 客户端 handle，服务端可以通过该 handle 与 gRPC 客户端进行通信。另外，通过 Archive ID 也可以获取对应到该客户端的 sync channel，每个派发到该客户端的 HSM 命令都会放到该 channel 中。channel 中的 HSM 命令并不会由 Proxy 主动推送到客户端中，必须由客户端主动调用 rpc GetAction 获取对应的 HSM 命令。

11.2 rpc GetAction

客户端主动调用该 rpc，用户同步的获取属于该客户端的 HSM 命令。如果不存在发往该客户端的命令，则该客户端会被阻塞。如果 Proxy 端存在发往该客户端的 HSM 命令，则 Proxy 会调用 rpc Send 将 sync channel 的 HSM 命令取出并发往对应的客户端。

11.3 rpc StatusStream

该 rpc 调用用于 DataMover 实例向 Proxy 汇报数据搬运的进度，Proxy 接收到进度信息时会进度进一步封装并发回到 CDT 服务中，CDT 服务根据进度信息可以间接确认 Proxy 的活动状态。

Global endpoints map 是 Proxy 中重要的数据结构。客户端主动调用 Register 时 Proxy 会在该 map 中登记客户端对应的信息。Plugin monitor 监测到客户端 DataMover 出现不可恢复异常时，会将对应的客户端项从该表中删除。

12 Plugin 模块分析

Plugin 用于管理一个或者多个 DataMover，其提供一种通用的 DataMover 操作接口，具体的接口的实现则由用户根据不同的存储后端实现不同的 DataMover。Plugin 模块的主要架构如下：

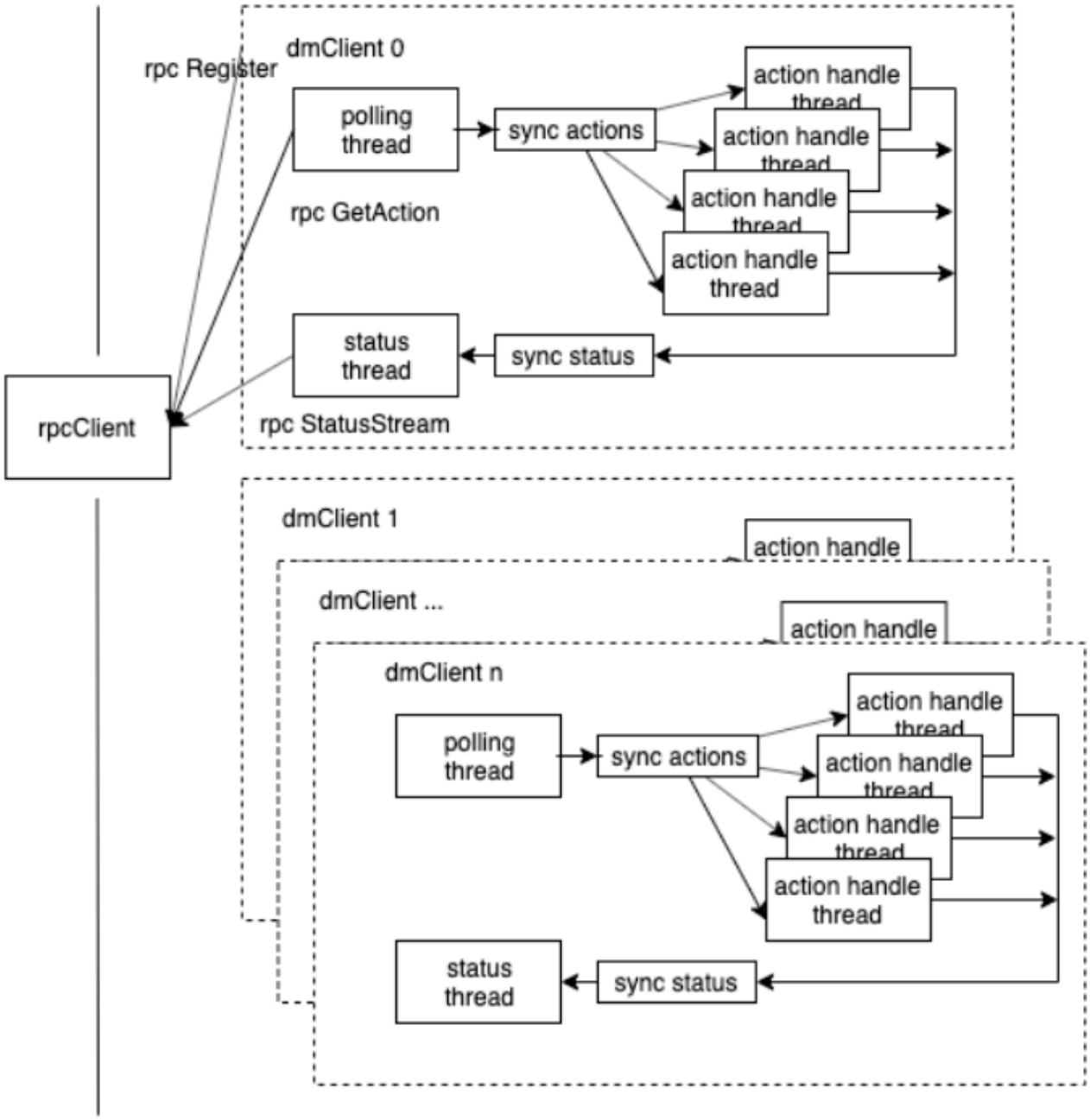

图 7: Plugin 模块

Plugin 模块定义可一种通用的 DataMover 操作框架，用户仅需要实现 Plugin 中定义的 archive，restore 以及 remove 操作。在 Plugin 中，每一个 DataMover 对应一个 dmClient。每个 dmClient

对应一个 archive ID，负责从 Proxy 中获取属于自己的 HSM 命令。dmClient 在启动时会创建一个 polling 线程，该线程不断的调用 GetAction 从 Proxy 获取一个 HSM 命令，之后将该命令放到一个同步的全局 actions channel 中。dmClient 启动一个或者多个 action handle 线程，每个线程从全局的同步 channel 中不断的读取 HSM 命令。读取到 HSM 命令时将调用 dmio 定义的接口进行数据拷贝，同时该线程不断的将数据拷贝的进度发送到一个同步的全局 status channel 中。status 线程不断的从 status channel 获取进度数据，并通过调用 StatusStream 将进度数据不断的发送到 Proxy 中。

13 DataMover 模块

DataMover 当前需要实现下属三种接口:

```
1    // 将数据从lustre中下沉到冷存储中
2    func (m *Mover) Archive(action dmplugin.Action) error
3    // 将数据从冷存储中拷贝到lustre中
4    func (m *Mover) Restore(action dmplugin.Action) error
5    // 将数据中冷数据中删除
6    func (m *Mover) Remove(action dmplugin.Action) error
```

上述函数的参数为 Action 接口。在 Lemur 中，Action 的实现者是 dmAction，其定义如下:

```
1    dmAction struct {
2        // 操作执行进度信息的读写channel
3        status chan *pb.ActionStatus
4        // 由protobuf定义的ActionItem的指针
5        item *pb.ActionItem
6        // Action实际需要拷贝数据的长度
7        actualLength *int64
8        // lustre xattr属性中uuid属性
9        uuid string
10       // lustre xattr属性中hash属性
11       hash []byte
12       // lustre xattr属性中url属性
13       url string
14   }
```

其中 uuid，url 以及 hash 可以用于唯一标识一个文件的 id，或者存储一些用户希望的数据。比如 hash 值可以用于确认文件内容是否相同，在数据下沉时可以做数据去重。

Lemur 当前实现了 posix DataMover 以及 s3 DataMover。