

# DAOS Pool

---

## 概述

[DAOS Object](#)

[Pool Service](#)

[Pool Operations](#)

[Pool Connect](#)

[Replicated Services](#)

[Pool Self-Healing](#)

[Rebuild Detection](#)

[Rebuild 过程](#)

[rebuild 多个Pool和Target](#)

[rebuild 期间的I/O](#)

[rebuild资源限制](#)

[rebuild状态](#)

[rebuild失败](#)

[使用校验和rebuild](#)

[Rebuild Touch Points](#)

[Client Task API Touch Points](#)

[Packing/unpacking checksums](#)

## Daos pool 相关函数

[Storage Target](#)

[Storage Pool](#)

[daos\\_pool\\_connect](#)

[daos\\_pool\\_disconnect](#)

[daos\\_pool\\_local2global](#)

[daos\\_pool\\_global2local](#)

[daos\\_pool\\_query](#)

[daos\\_pool\\_query\\_target](#)

[daos\\_pool\\_list\\_attr](#)

[daos\\_pool\\_get\\_attr](#)

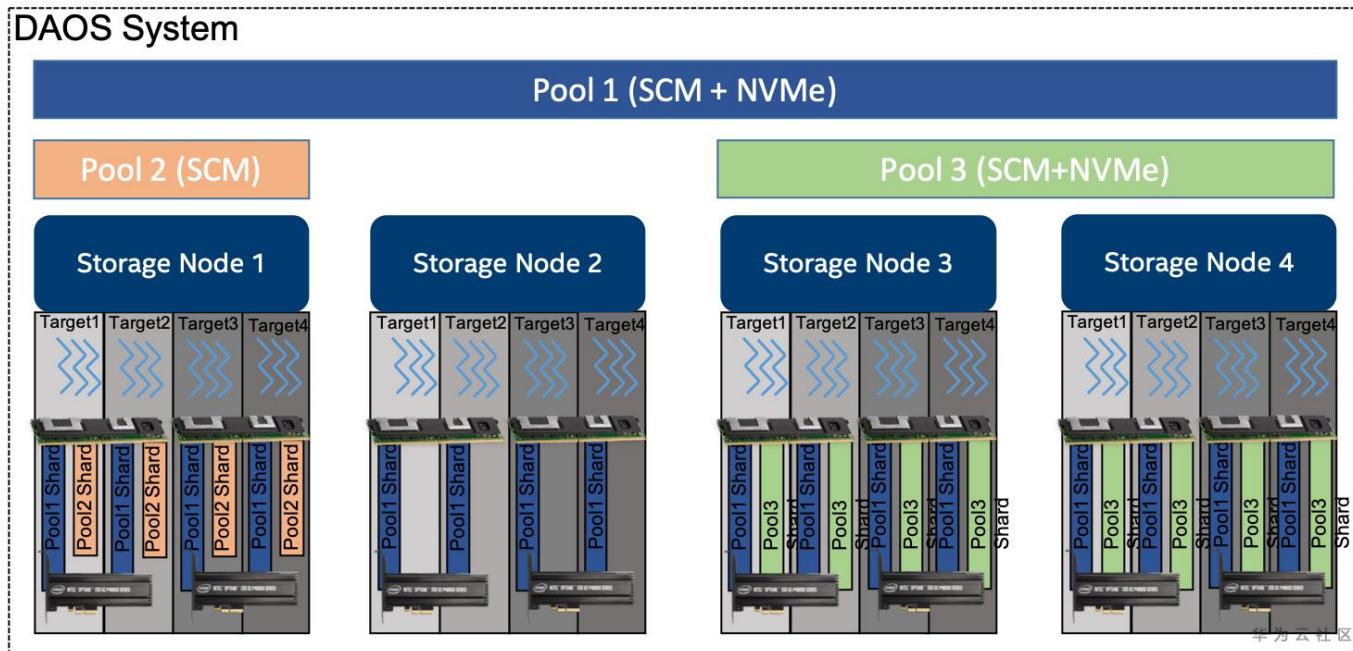
[daos\\_pool\\_set\\_attr](#)

[daos\\_pool\\_del\\_attr](#)

[daos\\_pool\\_list\\_cont](#)

DAOS (Distributed Asynchronous Object Storage) 是一个开源的对象存储系统，专为大规模分布式非易失性内存设计，利用了 SCM 和 NVMe 等的下一代 NVM 技术。DAOS 同时在硬件之上提供了键值存储接口，提供了诸如事务性非阻塞 I/O、具有自我修复的高级数据保护、端到端数据完整性、细粒度数据控制和弹性存储的高级数据保护，从而优化性能并降低成本。

## 概述



DAOS Pool 是分布在 Target 集合上的存储资源预留。分配给每个 Target 上的 Pool 的实际空间称为 Pool Shard。

分配给 Pool 的总空间在创建时确定，后期可以通过调整所有 Pool Shard 的大小（在每个 Target 专用的存储容量限制内）或跨越更多 Target（添加更多 Pool Shard）来随时间扩展。

Pool 提供了存储虚拟化，是资源调配和隔离的单元。DAOS Pool 不能跨多个系统。

一个 Pool 可以承载多个称为 DAOS Container 的事务对象存储。每个 Container 都是一个私有的对象地址空间，可以对其进行事务性修改，并且独立于存储在同一 Pool 中的其他 Container。Container 是快照和数据管理的单元。属于 Container 的 DAOS 对象可以分布在当前 Pool 的任何一个 Target 上以提高性能和恢复能力，并且可以通过不同的 API 访问，从而高效地表示结构化、半结构化和非结构化数据。

下表显示了每个 DAOS 概念的目标可伸缩性级别：

DAOS 概念	可伸缩性（数量级）
System	$10^{5105}$ Servers and $10^{2102}$ Pools
Server	$10^{1101}$ Targets
Pool	$10^{2102}$ Containers
Container	$10^{9109}$ Objects

Pool 由唯一的 Pool UUID 标识，并在称为 Pool map 的持久版本控制列表中维护 Target 成员身份。成员资格是确定且一致的，成员资格的变更按顺序编号。Pool map 不仅记录活跃 Target 的列表，还以树的形式包含存储拓扑，用于标识共享公共硬件组件的 Target。例如，树的第一级可以表示共享同一主板的 Target，第二级可以表示共享同一机架的所有主板，最后第三级可以表示同一机房中的所有机架。

## ▼ struct pool\_map

C | 复制代码

```
1  struct pool_map {
2      /* protect the refcount */
3      pthread_mutex_t po_lock;
4      /* Current version of pool map */
5      uint32_t po_version;
6      /* refcount on the pool map */
7      int po_ref;
8      /* # domain layers */
9      unsigned int po_domain_layers;
10     /*fault map version*/
11     uint32_t po_fault_map_version;
12     uint64_t po_fault_map_len;
13
14     /* fault domain type */
15     pool_comp_type_t po_fault_domain_type;
16
17     //用于不同域类型的二进制搜索的分类器。按升序进行的二进制搜索。.
18     struct pool_comp_sorter *po_domain_sorters;
19     /* sorter for binary search of target */
20     struct pool_comp_sorter po_target_sorter;
21     /*
22     * Tree root of all components.
23     * NB: All components must be stored in contiguous buffer.
24     */
25     struct pool_domain *po_tree;
26     /*
27     * number of currently failed pool components of each type
28     * of component found in the pool
29     */
30     struct pool_fail_comp *po_comp_fail_cnts;
31
32     struct pool_pds *po_pds;
33     uint64_t po_reclaim_need:1;
34 }
```

该框架有效地表示了层次化的容错域，然后使用这些容错域来避免将冗余数据放置在发生相关故障的 Target 上。在任何时候，都可以将新 Target 添加到 Pool map 中，并且可以排除失败的 Target。此外，Pool map 版本化，这有效地为 map 的每次修改分配了唯一的序列，特别是对于失效节点的删除。

C | 复制代码

```
1 //添加新 Target
2 int pool_target_id_list_append(struct pool_target_id_list *id_list,
3                                 struct pool_target_id *id)
4
5 //排除失效的Target
6 void pool_target_id_list_remove(struct pool_target_id_list *id_list,
7                                  int index, bool rebuild_start)
8
9 //根据ID查找 Target
10 bool pool_target_id_found(struct pool_target_id_list *id_list,
11                            struct pool_target_id *tgt)
12
13 //根据状态查找 Target, 包括UP、down、downout、down_up、failed、upin_tgts
14 int pool_map_find_tgts_by_state(struct pool_map *map,
15                                   pool_comp_state_t match_states,
16                                   struct pool_target **tgt_pp, unsigned int *tgt_cnt)
```

Pool Shard 是永久内存的预留，可以选择与特定 Target 上 NVMe 预先分配的空间相结合。它有一个固定的容量，满了就不能运行。可以随时查询当前空间使用情况，并报告 Pool Shard 中存储的任何数据类型所使用的总字节数。

一旦 Target 失败并从 Pool map 中排除，Pool 中的数据冗余将自动在线恢复。这种自愈过程称为重建。重建进度定期记录在永久内存中存储的 Pool 中的特殊日志中，以解决级联故障。添加新 Target 时，数据会自动迁移到新添加的 Target，以便在所有成员之间平均分配占用的空间。这个过程称为空间再平衡，使用专用的持久性日志来支持中断和重启。

Pool 是分布在不同存储节点上的一组 Target，在这些节点上分布数据和元数据以实现水平可伸缩性，并使用复制或纠删码 (erasure code) 确保持久性和可用性。

创建 Pool 时，必须定义一组系统属性以配置 Pool 支持的不同功能。此外，用户还可以定义将持久存储的属性。

C | 复制代码

```
1 /* Create a pool map from components stored in \a buf.
2
3 int pool_map_create(struct pool_buf *buf,      //The buffer to input pool comp
4                      uint32_t version,        //Version for the new created p
5                      pool_map **mapp)
```

Pool 只能由经过身份验证和授权的应用程序访问。DAOS 支持多种安全框架，例如 NFSv4 访问控制列表或基于第三方的身份验证 (Kerberos)。连接到 Pool 时强制执行安全性检查。成功连接到 Pool 后，将向应用程序进程返回连接上下文。

```
1 int daos_pool_connect(const uuid_t uuid, const char *grp,
2                         unsigned int flags,
3                         daos_handle_t *poh, daos_pool_info_t *info, daos_event_t *ev)
```

如前文所述，Pool 存储许多不同种类的持久性元数据，如 `Pool map`、身份验证和授权信息、用户属性、特性和重建日志。这些元数据非常关键，需要最高级别的恢复能力。因此，Pool 的元数据被复制到几个来自不同高级容错域的节点上。对于具有数十万个存储节点的非常大的配置来说，这些节点中只有很小的一部分（大约几十个）运行 Pool 元数据服务。在存储节点数量有限的情况下，DAOS 可以依赖一致性算法来达成一致，在出现故障时保证一致性，避免脑裂。

```
1 // Storage pool
2
3 typedef struct { ... } daos_pool_info_t;
29
30 - struct ds_pool {
31     struct daos_llink    sp_entry;
32     uuid_t                sp_uuid;      /* pool UUID */
33     ABT_rwlock            sp_lock;
34     struct pool_map        *sp_map;
35     d_list_t               agg_tt_list;
36     /**fault map buffer */
37     struct fault_map_buf   *sp_fault_map;
38     /**存放从fault map中解析出的所有pool buffer */
39     struct pool_buf_list    sp_pool_buf_list;
40     /**从pool_buf_list中解析出的pool map */
41     struct pool_map_list    sp_pool_map_list;
42     /**从pool_map_list中解析出的pl map */
43     struct pl_map_list      sp_pl_map_list;
44     /**从pool_buf_list中解析出的fault targets */
45     struct fault_target_list sp_fault_target_list;
46     uint32_t                sp_map_version; /* temporary */
47     uint32_t                sp_ec_cell_sz;
48     uint64_t                sp_reclaim;
49     crt_group_t             *sp_group;
50     ABT_mutex                sp_mutex;
51     ABT_cond                 sp_fetch_hdls_cond;
52     ABT_cond                 sp_fetch_hdls_done_cond;
53     struct ds_iv_ns          *sp_iv_ns;
54
55     /* structure related to EC aggregate epoch query */
56     d_list_t               sp_ec_ephs_list;
57     struct sched_request     *sp_ec_ephs_req;
58
59     uint32_t                sp_dtx_resync_version;
60     /* Special pool/container handle uuid, which are
61      * created on the pool leader step up, and propagated
62      * to all servers by IV. Then they will be used by server
63      * to access the data on other servers.
64      */
65     uuid_t                  sp_srv_cont_hdl;
66     uuid_t                  sp_srv_pool_hdl;
67     uint32_t                sp_stopping:1,
68                           sp_fetch_hdls:1;
69
70     int                     sp_reintegrating;
```

```

71     uint64_t      sp_stable_epoch;
72     /** path to ephemeral metrics */
73     char          sp_path[D_TM_MAX_NAME_LEN];
74     bool         timer_initied;
75     int          flow_flag;
76     timer_t       agg_token_timer;
77     ABT_mutex    agg_mutex;
78     /**
79      * Per-pool per-module metrics, see ${modname}_pool_metrics for the
80      * actual structure. Initialized only for modules that specified a
81      * set of handlers via dss_module::sm_metrics handlers and reported
82      * DAOS_SYS_TAG.
83      */
84     void          *sp_metrics[DAOS_NR_MODULE];
85 };

```

要访问 Pool，用户进程应该连接到 Pool 并通过安全检查。一旦授权，Pool 就可以与任何或所有对等应用程序进程（类似 `open()` POSIX 扩展）共享（通过 `local2global()` 和 `global2local()` 操作）连接。这种集体连接机制有助于在数据中心上运行大规模分布式作业时避免元数据请求风暴。当发出连接请求的原始进程与 Pool 断开连接时，Pool 连接将被注销。

## DAOS Object

Object 类定义了一组对象的公共模式属性，每个 Object 类都被分配一个唯一的标识符，并在 Pool 级别与给定的模式相关联。一个新的 Object 类可以在任何时候用一个可配置的模式来定义，这个模式在创建之后是不可变的（或者至少在属于这个类的所有对象都被销毁之前）。

为了方便起见，在创建 Pool 时，默认情况下会预定义几个最常用的 Object 类：

Object Class (RW = read/write, RM = read-mostly)	Redundancy	Layout (SC = stripe count, RC = replica count, PC = parity count, TGT = target)
Small size & RW	Replication	static SCxRC, e.g. 1x4
Small size & RM	Erasure code	static SC+PC, e.g. 4+2
Large size & RW	Replication	static SCxRC over max #targets)
Large size & RM	Erasure code	static SCx(SC+PC) w/ max #TGT)
Unknown size & RW	Replication	SCxRC, e.g. 1x4 initially and grows
Unknown size & RM	Erasure code	SC+PC, e.g. 4+2 initially and grows

为了避免传统存储系统常见的扩展问题和开销，DAOS 有意将对象简化，不提供类型和架构之外的默认对象元数据。这意味着系统不维护时间、大小、所有者、权限，甚至不跟踪开启者。

为了实现高可用性和水平伸缩性，DAOS 提供了许多对象模式（复制/纠删码、静态/动态条带化等）。模式框架是灵活的，并且易于扩展，以允许将来使用新的自定义模式类型。模式布局是在对象标识符和 Pool 映射打开的对象上通过算法生成的。通过在网络传输和存储期间使用校验和保护对象数据，确保了端到端的完整性。

可以通过不同的 API 访问 DAOS 对象：

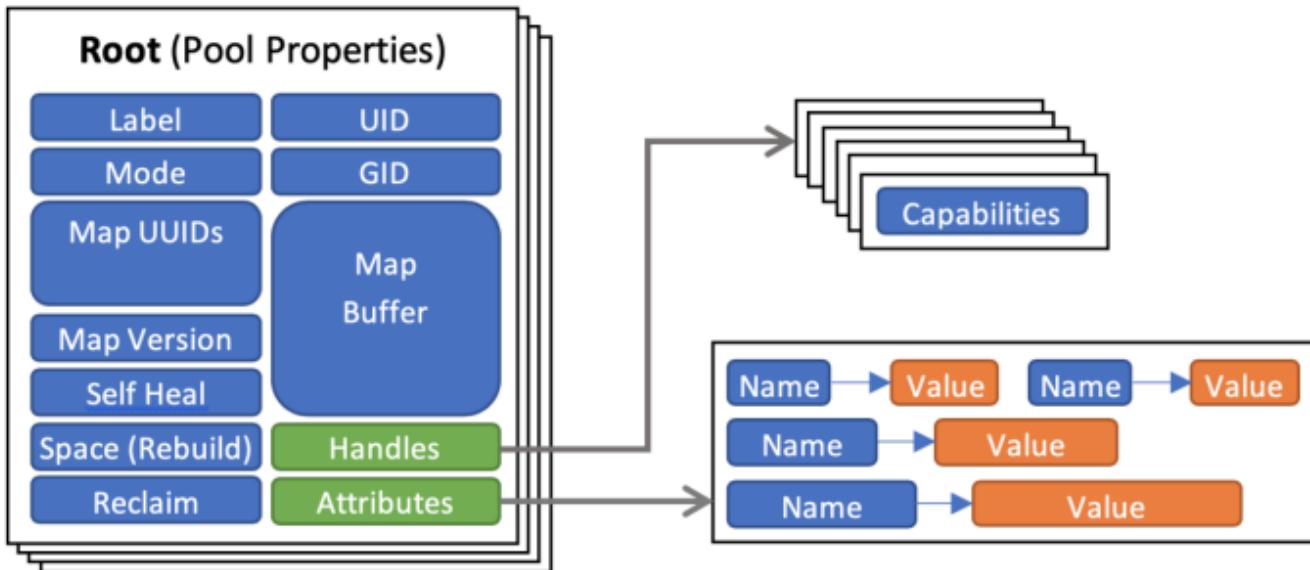
- **Multi-level key-array API** 是具有局部性特征的本机对象接口。key 分为 distribution key (dkey) 和 attribute key (akey)。dkey 和 akey 都可以是可变长度的类型（字符串、整数或其它复杂的数据结构）。同一 dkey 下的所有条目都保证在同一 Target 上并置。与 akey 关联的值可以是不能部分修改的单个可变长度值，也可以是固定长度值的数组。akeys 和 dkey 都支持枚举。
- **Key-value API** 提供了一个简单的键和可变长度值接口。它支持传统的 put、get、remove 和 list 操作。
- **Array API** 实现了一个由固定大小的元素组成的一维数组，该数组的寻址方式是 64 位偏移寻址。DAOS 数组支持任意范围的 read、write 和 punch 操作。

# Pool Service

pool 服务 (Pool\_svc) 存储pool的元数据，并提供用于查询和更新pool 配置的API。

```
1  struct pool_svc {
2      struct ds_rsvc      ps_rsvc;
3      uuid_t             ps_uuid;    /* pool UUID */
4      struct cont_svc    *ps_cont_svc; /* one combined svc for now */
5      ABT_rwlock         ps_lock;    /* for DB data */
6      rdb_path_t         ps_root;    /* root KVS */
7      rdb_path_t         ps_handles; /* pool handle KVS */
8      rdb_path_t         ps_user;    /* pool user attributes KVS */
9      struct ds_pool     *ps_pool;
10     struct pool_svc_events  ps_events;
11     struct pool_svc_events  ps_events_sub;
12  };
```

pool元数据被组织为键值存储 (KVS) 的层次结构，这些键值存储 (KVS) 通过Raft consensus protocol在多个服务器上复制；client 请求只能由svc Leader提供服务，而非负责人副本仅会以指向当前Leader的提示进行响应，以便client 重试。`pool_svc` 源自通用复制服务模块rsvc（请参阅：[Replicated Services: Architecture](#)），其实现有助于 client搜索当前的服务端 leader。



顶层KVS模块存储pool的映射、安全属性（如UID、GID和模式）、与空间管理和自愈相关的信息，以及包含用户定义属性的第二级KVS。此外，它还存储有关pool连接的信息，这些信息由handle表示，并由client生成的handle UUID标识。术语“pool connection”和“pool handle”可以互换使用。

## Pool Operations

pool的创建完全由 Management Service 驱动，因为它需要与存储分配和容错域查询相关的步骤的特权。格式化所有目标后，目标组件在每个目标上调用pool模块的 `ds_pool_create`，它只为当前目标生成一个新的UUID，并将其存储在DSM\_META\_文件中。

```
1 //this RPC to not only create the pool metadata but also initialize the po
2 ol/container service DB.
3
4 void ds_pool_create_handler(crt_rpc_t *rpc)
5 {
6     struct pool_create_in *in = crt_req_get(rpc);
7     struct pool_create_out *out = crt_reply_get(rpc);
8     struct pool_svc       *svc;
9     struct rdb_kvs_attr attr;
10
11     ...
12
13     rc = init_pool_metadata(&tx, &svc->ps_root, in->pri_ntgts, NULL /* gro
14     up */,
15                 in->pri_tgt_ranks, prop_dup, in->pri_ndomains,
16                 in->pri_domains.ca_arrays, in->pri_fault_domain_type,
17                 in->pri_topo);
18
19     if (rc != 0)
20         D_GOTO(out_tx, rc);
21     rc = ds_cont_init_metadata(&tx, &svc->ps_root, in->pri_op.pi_uuid);
22     if (rc != 0)
23         D_GOTO(out_tx, rc);
24 }
```

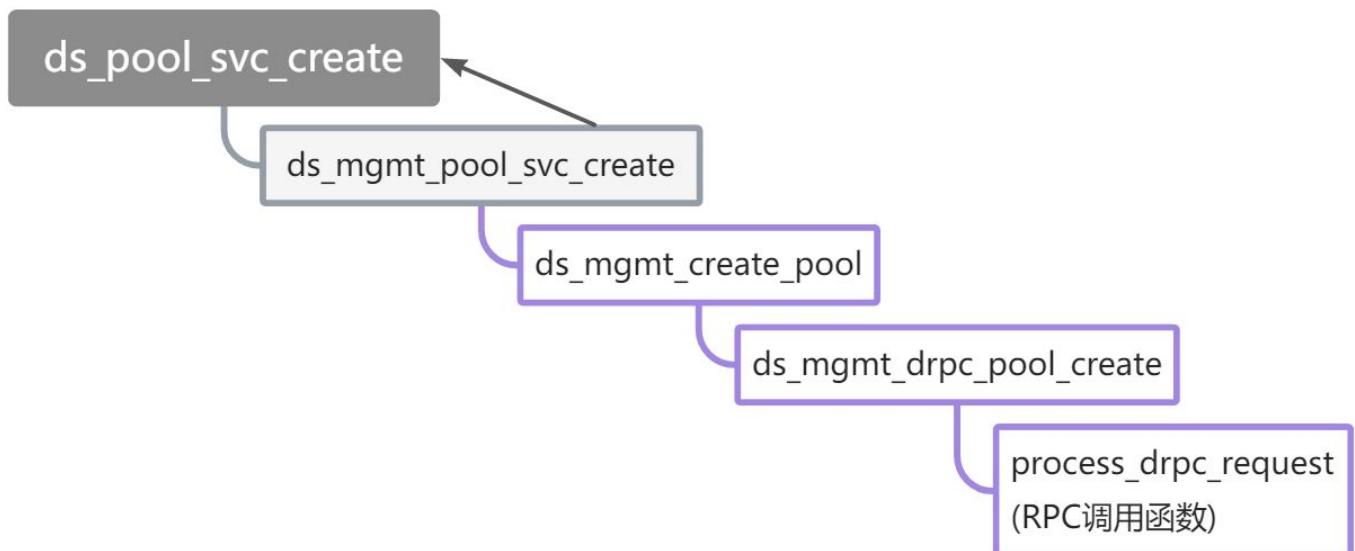
此时，management module 通过调用 `DS_pool_svc_create` 将控制权传递给pool模块，这将在combined Pool 和 Container Service 的选定节点子集上初始化服务复制。pool 模块现在向创建服务数据库的服务负责人发送pool 创建请求；然后，目标及其容错域的列表将转换为pool 映射的初始版本，并与其他初始pool 元数据一起存储在pool 服务中。

```

1  /**
2   * \param[in]          pool_uuid    pool UUID
3   * \param[in]          ntargets     number of targets in the pool
4
5   * \param[in,out]      svc_addrs   \a svc_addrs.rl_nr inputs how many
6   *                           replicas shall be created; returns the list of pool se
7   *                           rvice replica ranks
8   */
9  int ds_pool_svc_create(const uuid_t pool_uuid, int ntargets, const char *g
10    group,
11            const d_rank_list_t *target_addrs, int ndomains, const uint32_t
12    *domains,
13            daos_prop_t *prop, d_rank_list_t *svc_addrs, const char* fault
14    domaintype,
15            const ds_pool_topo *pool_topo)
16 {
17 ...
18 // 选择副本数，输出实际选择的副本数，可能小于要求的
19 rc = select_svc_ranks(svc_addrs->rl_nr, target_addrs, ndomains,
20                      domains, &ranks, faultdomaintype);
21 ...
22 };

```

pool的创建函数调用过程如下图：



## Pool Connect

为了建立pool连接，客户端进程使用pool UUID、连接信息（如组名和服务等级列表）和连接标志，并调用客户端库中的daos\_pool\_connect方法；这会向Pool Service服务发起 daos\_pool\_connect 请求。

```
1 int daos_pool_connect(const uuid_t uuid, const char *grp,
2                         unsigned int flags,
3                         daos_handle_t *poh, daos_pool_info_t *info, daos_event_t *ev)
4 {
5     daos_pool_connect_t *args;
6     tse_task_t *task;
7     int rc;
8     ...
9
10    // 创建新任务 dc_pool_connect，并将其与输入事件 ev 关联
11    // 如果事件 ev 为 NULL，则将获取私有事件
12    rc = dc_task_create(dc_pool_connect, NULL, ev, &task);
13    if (rc)
14        // dc_task_create 成功返回 0，失败返回负数
15        return rc;
16
17    // 调度创建的任务 task
18    // 如果该任务的关联事件是私有事件，则此函数将等待任务完成
19    // 否则它将立即返回，并通过测试事件或在 EQ 上轮询找到其完成情况
20    // 第二个参数 instant 为 true，表示任务将立即执行
21    return dc_task_schedule(task, true);
22 }
```

Pool Service尝试根据使用的安全模型（例如，类POSIX模型中的UID/GID）对请求进行身份验证，并将请求的功能授权给客户端生成的 pool 句柄 UUID。在继续之前，pool map 被传递到客户端；如果此时出现错误，服务器可以要求客户机放弃 pool map。此时，Pool Service 将检查现有的 pool handles。如果已经存在具有相同UUID的 pool handle，则表示已经建立了 pool 连接，无需执行其他操作。使得当前请求的或现有的 pool handle 具有独占访问权限，则连接请求将被拒绝，并显示忙碌状态代码。如果一切顺利，pool 服务将使用 pool handle UUID 向pool 中的所有 targets 发送一个集合POOL\_TGT\_CONNECT 连接请求。Target Service 创建并缓存本地pool 对象，并打开本地 VOS pool。

(see: [Storage Model: DAOS Pool](#) and [Use Cases: Storage Management and Workflow Integration](#)).

一组对等应用程序进程可以共享单个 pool connection handle。

要关闭pool 连接，connection handle 进程使用 pool handle 调用client 库中的 daos\_pool\_disconnect 方法，触发 Pool Service 的 pool\_disconnect\_hdls 断开请求，

pool Service向pool 中的所有目标发送一组 `POOL_TGT_DISCONNECT` 断开请求。

```
1 //client 库中的daos_pool_disconnect方法
2 int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev)
3 {
4     daos_pool_disconnect_t *args;
5     tse_task_t *task;
6     int rc;
7
8     DAOS_API_ARG_ASSERT(*args, POOL_DISCONNECT);
9     rc = dc_task_create(dc_pool_disconnect, NULL, ev, &task);
10    if (rc)
11        return rc;
12
13    args = dc_task_get_args(task);
14    args->poh = poh;
15
16    return dc_task_schedule(task, true);
17 }
18 }
```

C | 复制代码

```

1 // Pool Service的 POOL_DISCONNECT 断开请求
2 static int pool_disconnect_hdls(struct rdb_tx *tx, struct pool_svc *svc,
3                                     uuid_t *hdl_uuids, int n_hdl_uuids, crt_co
4                                     ntext_t ctx)
5 {
6     ...
7     rc = pool_disconnect_bcast(ctx, svc, hdl_uuids, n_hdl_uuids);
8     ...
9     return rc;
10 }
11
12 static int pool_disconnect_bcast(crt_context_t ctx, struct pool_svc *svc,
13                                     uuid_t *pool_hdls, int n_pool_hdls)
14 {
15     ...
16
17     //向pool 中的所有目标发送一组断开请求
18     rc = bcast_create(ctx, svc, POOL_TGT_DISCONNECT, NULL, &rpc);
19     ...
20     return rc;
21 }

```

这些步骤将销毁与连接关联的所有状态，包括所有 `container handle`。共享此连接的其他 client 进程应该在本地销毁它们的 `pool handle` 副本，最好是在代表所有人调用 `disconnect` 方法之前。如果一组 client 进程在有机会调用 `pool` 断开连接方法之前提前终止，那么一旦 `pool service` 从运行时环境了解到事件，它们的 `pool` 连接最终将被逐出。

## Replicated Services

代码实现在 `src\include\daos_srv\rsvc.h`

```
▼ /** Replicated service */
```

C | 复制代码

```
1 struct ds_rsvc {
2     d_list_t          s_entry;      /* in rsvc_hash */
3     enum ds_rsvc_class_id  s_class;
4     d_iov_t           s_id;        /*< for lookups */
5     char              *s_name;      /*< for printing */
6     struct rdb        *s_db;        /*< DB handle */
7     char              *s_db_path;
8     uuid_t            s_db_uuid;
9     int                s_ref;
10    ABT_mutex         s_mutex;      /* for the following members */
11    bool               s_stop;
12    uint64_t          s_term;      /*< leader term */
13    enum ds_rsvc_state s_state;
14    ABT_cond          s_state_cv;
15    int                s_leader_ref; /* on leader state */
16    ABT_cond          s_leader_ref_cv;
17    bool               s_map_dist; /* has a map dist request? */
18    ABT_cond          s_map_dist_cv;
19    ABT_thread         s_map_disted;
20    bool               s_map_disted_stop;
21    bool               is_in_up_progress; //swim检测到主挂后，判断是否应该触发选举
22};
```

```
▼
```

C | 复制代码

```
1 /** Replicated service state in ds_rsvc.s_term */
2 enum ds_rsvc_state {
3     DS_RSVC_UP_EMPTY,    /*< up but DB newly-created and empty */
4     DS_RSVC_UP,          /*< up and ready to serve */
5     DS_RSVC_DRAINING,   /*< stepping down */
6     DS_RSVC_DOWN         /*< down */
7};
```

DAOS 中的 RPC 服务（如 Pool\_svc 和 cont\_svc）使用 Raft 进行复制。这些服务中的每一个都可以容忍其少数副本的故障。通过将其副本分布在不同的容错域中，该服务可以高度可用。由于这种复制方法是自包含的，因为它只需要本地持久性存储和点对点不可靠消息传递，而不需要任何外部配置管理服务，因此这些服务对于引导 DAOS 系统以及管理轻量级 I/O 复制协议的配置是必需的。

RPC 服务根据其当前服务状态（或仅状态）处理传入的服务请求。因此，复制服务就是复制其状态，以便根据通过所有先前请求达到的状态处理每个请求。

使用 Raft 日志复制服务的状态。该服务将请求转换为状态查询和确定性状态更新。所有状态更新都首先提交到 Raft 日志，然后再应用于状态。由于 Raft 保证了日志副本之间的一致性，因此服务副本最终会

以相同的顺序应用相同的状态更新集，并经历相同的状态历史记录。

Raft采用 leadership 设计，每个复制的服务也是如此。一个 service 的 Leader 者是同一个 Raft 的 Leader。在服务的各个副本中，只有最高权限 leader 才能处理请求。对于服务器端，其代码实现类似于non-replicated RPC service的代码，除了处理 leadership 变更事件。对于客户端，必须将服务请求发送给当前Leader，如果 Leader 未知，则必须先搜索该 Leader。

replicated service 是使用模块堆栈实现的：

```
Bash | 复制代码
```

```
1 [ pool_svc, cont_svc, ... ]
2 [ ds_rsrc ]
3 [           rdb           ]
4 [ raft      ]
5 [           vos           ]
```

pool\_svc 实现相应服务的请求处理程序和领导更改事件处理程序。它们根据提供的 RDB 数据模型定义各自的服务状态，使用 RDB 事务实现状态查询和更新，并将其领导更改事件处理程序注册到框架产品/服务中。

```

1  struct rdb {
2      /* General fields */
3      d_list_t          d_entry;      /* in rdb_hash */
4      uuid_t            d_uuid;       /* of database */
5      ABT_mutex         d_mutex;      /* d_replies, d_replies_cv */
6      int               d_ref;        /* of callers and RPCs */
7      ABT_cond          d_ref_cv;     /* for d_ref decrements */
8      struct rdb_cbs   *d_cbs;       /* callers' callbacks */
9      void              *d_arg;       /* for d_cbs callbacks */
10     struct daos_lru_cache *d_kvss;    /* rdb_kvs cache */
11     daos_handle_t     d_pool;       /* VOS pool */
12     daos_handle_t     d_mc;        /* metadata container */
13
14     /* rdb_raft fields */
15     raft_server_t     *d_raft;
16     bool              d_raft_loaded; /* from storage (see rdb_raft_load) */
17     ABT_mutex         d_raft_mutex; /* for raft state machine */
18     daos_handle_t     d_lc;        /* log container */
19     struct rdb_lc_record d_lc_record; /* of d_lc */
20     daos_handle_t     d_slc;       /* staging log container */
21     struct rdb_lc_record d_slc_record; /* of d_slc */
22     uint64_t          d_applied;    /* last applied index */
23     uint64_t          d_debut;      /* first entry in a term */
24     ABT_cond          d_applied_cv; /* for d_applied updates */
25     struct d_hash_table d_results;  /* rdb_raft_result hash */
26     d_list_t          d_requests;   /* RPCs waiting for replies */
27     d_list_t          d_replies;    /* RPCs received replies */
28     ABT_cond          d_replies_cv; /* for d_replies enqueues */
29     struct rdb_raft_event d_events[2]; /* rdb_raft_events queue */
30     int               d_nevents;    /* d_events queue len from 0 */
31     ABT_cond          d_events_cv;   /* for d_events enqueues */
32     uint64_t          d_compact_thres; /* of compactable entries */
33     ABT_cond          d_compact_cv;   /* for base updates */
34     bool              d_stop;        /* for rdb_stop() */
35     ABT_thread        d_timerd;
36     ABT_thread        d_callbackd;
37     ABT_thread        d_recvfd;
38     ABT_thread        d_compactd;
39     size_t            d_ae_max_size;
40     unsigned int      d_ae_max_entries;
41 };

```

rdb (daos\_srv/rdb) 实现了具有事务的分层键值存储数据模型，使用 Raft 进行复制。它将 Raft Leader 变更事件传递，使用 Raft log 实现事务，并使用 VOS 数据模型存储服务的数据模型及其自己的

内部元数据。在Leader 副本上，与 VOS 接口以监视可用的持久性存储，并且当可用空间降至阈值以下时，在将entries 追加到 Raft 日志之前拒绝新事务，否则可能导致服务变得不可用（由于应用entries 时混合了成功和“空间不足”故障）。它还与 VOS 接口，通过触发日志的旧版本（epochs）的聚合来定期压缩存储。

`raft` (`rdb/raft/include/raft.h`) : 实现 Raft 核心协议。它与VOS和CaRT的集成是通过回调函数在内部完成的。

搜索是通过客户端维护的候选服务副本列表和服务器RPC错误响应的组合来完成的，在某些情况下，这些响应包含可以找到当前领导者的提示。未运行该服务的服务器将响应一个错误，客户端使用该错误从其列表中删除该服务器。充当非领导者副本的服务器会以不同的错误进行响应，包括客户端用于添加到其列表并更改其对领导者的搜索的提示。并且，无论是在客户端启动时，还是在客户端的候选服务副本列表可能变为空时（例如，由于服务中的成员身份更改），都会联系在管理服务节点上运行的DAOS服务器之一，以获取pool的最新服务副本列表。

模块化 `rsvc` 主要目的是避免不同复制的服务实现之间的代码重复。回调密集型 API 源于尝试提取尽可能多的通用代码，即使以牺牲 API 的简单性为代价。这是与其他模块 API 设计方式的关键区别。

`rsvc`有两个部分：

- `ds_rsvc` (`daos_srv/rsvc.h`) : 服务器端框架。
- `dc_rsvc` (`daos/rsvc.h`) : 客户端库。

C | 复制代码

```
1  /** Replicated service client (opaque) */
2  struct rsvc_client {
3      d_rank_list_t *sc_ranks;          /* of rsvc replicas */
4      bool         sc_leader_known;    /* cache nonempty */
5      unsigned int sc_leader_aliveness; /* 0 means dead */
6      uint64_t     sc_leader_term;
7      int          sc_leader_index;   /* in sc_ranks */
8      int          sc_next;           /* in sc_ranks */
9      uint32_t     *sc_choosed;        /* choose record */
10 };
11
12 int rsvc_client_init(struct rsvc_client *client, const d_rank_list_t *rank
13 s);
14 void rsvc_client_fini(struct rsvc_client *client);
15 //为RPC调用选择一个endpoint
16 int rsvc_client_choose(struct rsvc_client *client, crt_endpoint_t *ep);
17
18 int rsvc_client_choose_one_round(struct rsvc_client *client, crt_endpoint_
19 t *ep);
20 int rsvc_client_complete_rpc(struct rsvc_client *client,
21                         const crt_endpoint_t *ep, int rc_crt, int rc_svc,
22                         const struct rsvc_hint *hint);
23 size_t rsvc_client_encode(const struct rsvc_client *client, void *buf);
24 ssize_t rsvc_client_decode(void *buf, size_t len, struct rsvc_client *cli
25 ent);
```

C | 复制代码

```
1  /* rsvc_client在没有leadership hint的情况下处理错误。 */
2  static void
3  rsvc_client_process_error(struct rsvc_client *client, int rc,
4                           const crt_endpoint_t *ep)
```

## Pool Self-Healing

Rebuild

在DAOS中，如果数据在不同的 Target 上复制多个副本，一旦其中一个 Target 发生故障，它的数据将自动在其他 Target 上重建，因此数据冗余不会因 Target 故障而受到影响。在未来的版本中，DAOS还将支持纠删码来保护数据；然后重建过程可能会相应地更新。

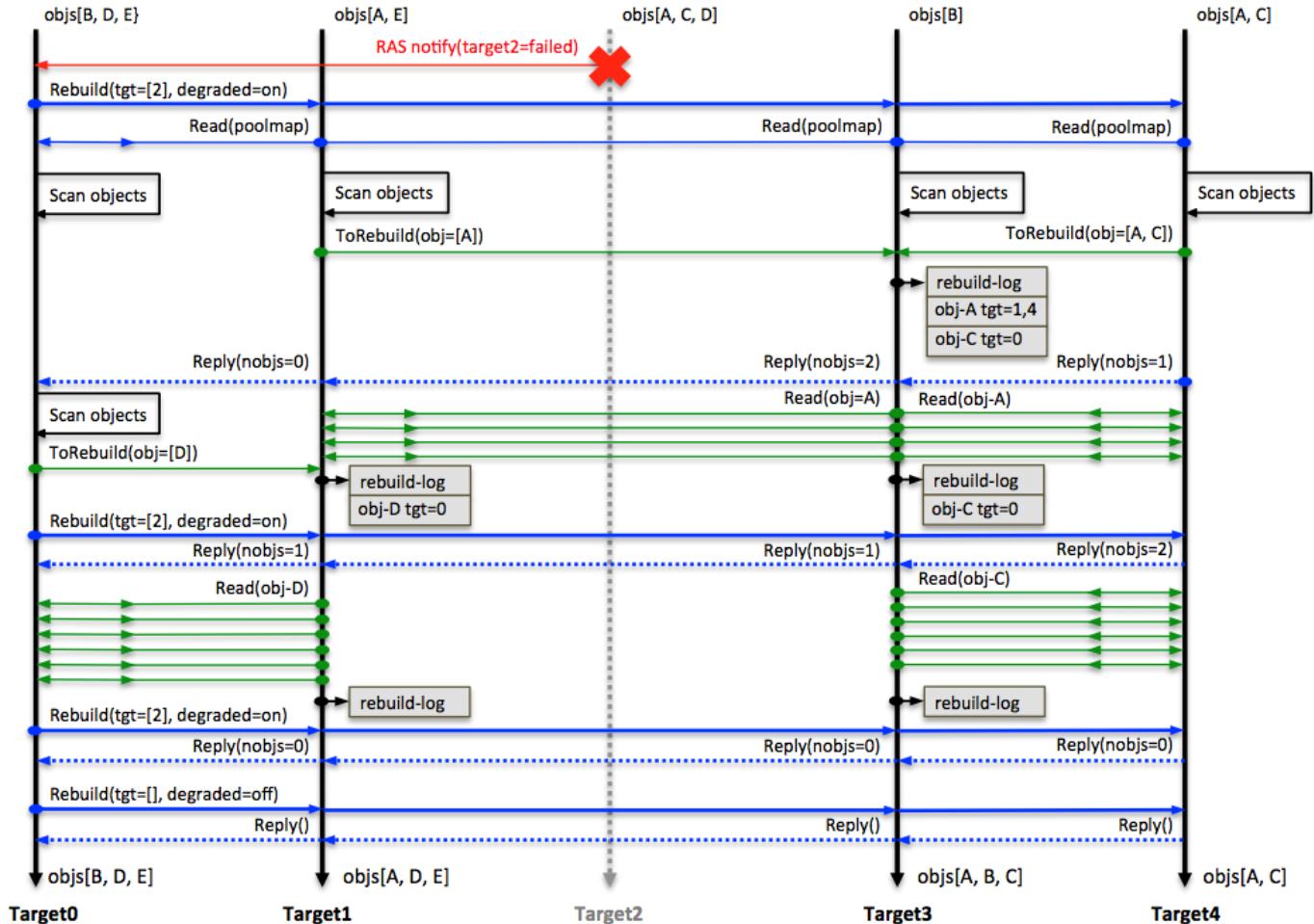
## Rebuild Detection

当目标失败时，应立即检测到该 Target 并通知 Pool (Raft) Leader，然后 Leader 将从 Pool 中排除 Target 并立即触发重建过程。目前Daos无法自动排除目标，因此系统管理员必须手动从Pool中排除故障Target，然后触发重建。将来，Leader 应该能够及时检测到Target故障，然后自行触发重建，而无需系统管理员的帮助。

## Rebuild 过程

重建分为两个阶段，扫描 (scan) 和拉动 (Pull)

- 扫描 (scan)：最初 Leader 会通过集合 RPC 将失败通知传播到所有其他幸存的Target。接收此 RPC 的任何Target将开始扫描其对象表，以确定对象丢失了故障Target上的数据冗余。如果是这样，请将其 ID 和相关元数据发送到重建目标（重建启动器）。至于如何为故障目标选择重建目标
- 拉动 (Pull)：重建启动器从扫描Target获取对象列表后，将从其他副本中提取这些对象的数据，然后在本地写入数据。每个Target都将向集群Leader报告其重建状态、重建对象、记录是否完成等。一旦Leader了解到所有Target都已完成扫描和重建阶段，它将通知所有Target重建已完成，随即便可以释放重建过程中持有的所有资源。



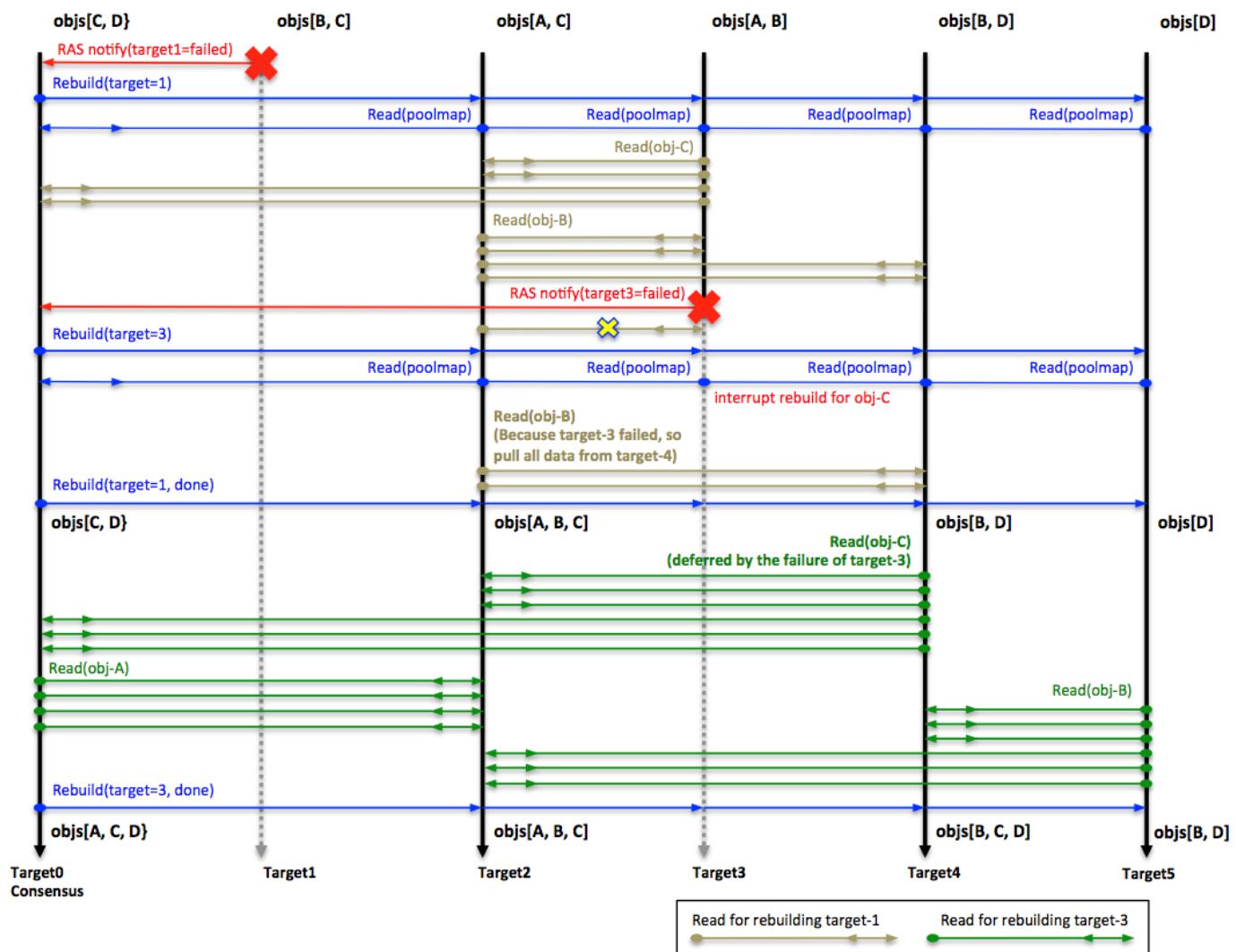
上图是此过程的一个示例：群集中有五个对象：对象 A 是 3 向复制的，对象 B、C、D 和 E 是双向复制的。当目标 2 失败时，目标 0（即 Raft 领导者）向所有幸存的目标广播失败，以通知它们进入降级模式并进行扫描：

1. Target-0 发现对象 D 丢失了一个副本，并计算出 target-1 是 D 的重建目标，因此它将对象 D 的 ID 及其元数据发送到目标 1。
2. Target-1 发现对象 A 丢失了一个副本，并计算出目标 3 是 A 的重建目标，因此它将对象 A 的 ID 及其元数据发送到目标 3。
3. Target-4 发现对象 A 和 C 丢失了副本，并且计算出 target-3 是对象 A 和 C 的重建目标，因此它将对象 A 和 C 的 ID 及其元数据发送到目标 3。
4. 在收到这些对象 ID 及其元数据后，target-1 和 target-3 可以计算出这些对象的幸存副本，并通过从这些副本中提取数据来重建这些对象。

## rebuild 多个Pool和Target

在大规模存储群集中，当从以前的故障重建仍在进行时，可能会发生多次故障。在这种情况下，DAOS既不应同时处理这些故障，也不应中断和重置较早的重建进度以应对以后的故障。否则，每次故障的重建所花费的时间可能会显著增加，并且如果新故障与正在进行的重建重叠，则重建可能永远不会结束。因此，对于多个故障，将应用这些规则

1. 如果重建启动器在重建期间失败，则应忽略在启动器上重建的对象分片，这将由下次重建处理。
2. 如果重建启动器由于故障而无法从其他副本获取数据，它将切换到其他副本（如果可用）。
3. 如果发生另一个故障，则重建中的目标不需要重新扫描其对象或重置当前故障的重建进度。
4. 当存在多个故障时，如果来自不同域的失败目标数超过容错级别，则可能存在不可恢复的错误，并且应用程序可能会遭受数据丢失。在这种情况下，上层堆栈软件在向可能缺少数据的对象发送 I/O 时可能会看到错误



上图示例中，对象 A 是双向复制的，对象 B、C 和 D 是 3 向复制的。

1. 在Target 1失败后，目标2是重建对象B的发起者，它从Target 3和Target 4中提取数据；Target 3是重建对象 C 的发起方，它从Target 0 和Target 2 中提取数据。
2. Target-3 在完成Target-1 的重建之前失败了，因此此时应该放弃对象 C 的重建，因为Target-3 是

它的发起者。对象 C 缺少的数据冗余将在重建Target-3 时重建。

3. 由于Target-3 也是重建对象 B 的贡献者，因此基于协议，对象 B 的发起方（即Target-2）应切换到Target-4 并继续重建对象 B。
4. Target-1的重建过程可以在完成对象B的重建后完成。此时，对象 C 在Target-3 重建时失败，因此仍然丢失了一个副本。
5. 在重建Target-3 的过程中，Target-4 是重建对象 C 的新发起者。

## rebuild 期间的I/O

如果在重建期间存在并发写入，则rebuild协议应保证新写入不会存在丢失。这些写入操作应直接存储在新对象分片中，或由重建启动器拉到新对象分片。并且还应该保证获取正确的数据。为了实现这些目标，应用了以下协议：

1. Fetch 将始终跳过重建目标。
2. 仅当所有对象分片的更新都已成功完成时，更新才能完成。
3. 如果这些更新中的任何一个失败，客户端将无限期重试，直到成功或者存在Pool map更改。在第二种情况下，客户端将切换到新的Pool map，并根据新的Pool map的来重建Target。
4. 正常 I/O 和重建过程之间没有同步，因此在重建过程中，重建启动器和正常 I/O 可能会重复写入数据。

## rebuild资源限制

在重建过程中，用户可以设置限制，以确保重建不会使用比用户设置更多的资源。用户现在只能设置CPU 周期。例如，如果用户将限制设置为 50，则重建最多将使用 50% 的 CPU 周期来执行重建作业。CPU 周期的默认重建限制为 30。

## rebuild状态

如前所述，每个目标将按 IV 向Pool leader报告其重建状态，然后leader将汇总所有Target的状态，并每隔 2 秒打印出整个重建状态，例如这些消息。

Bash | 复制代码

```
1 ▾ Rebuild [started] (pool 8799e471 ver=41)
2 ▾ Rebuild [scanning] (pool 8799e471 ver=41, toberb_obj=0, rb_obj=0, rec= 0, done 0 status 0 duration=0 secs)
3 ▾ Rebuild [queued] (419d9c11 ver=2)
4 ▾ Rebuild [started] (pool 419d9c11 ver=2)
5 ▾ Rebuild [scanning] (pool 419d9c11 ver=2, toberb_obj=0, rb_obj=0, rec= 0, done 0 status 0 duration=0 secs)
6 ▾ Rebuild [pulling] (pool 8799e471 ver=41, toberb_obj=75, rb_obj=75, rec= 11937, done 0 status 0 duration=10 secs)
7 ▾ Rebuild [completed] (pool 419d9c11 ver=2, toberb_obj=10, rb_obj=10, rec= 1026, done 1 status 0 duration=8 secs)
8 ▾ Rebuild [completed] (pool 8799e471 ver=41, toberb_obj=75, rb_obj=75, rec= 13184, done 1 status 0 duration=14 secs)
```

有 2 个Pool 正在重建 (Pool 8799e471 和 Pool 419d9c11, 注意: 此处仅显示池 uuid 的前 8 个字母)

Bash | 复制代码

- 1 The 1st line means the rebuild *for* pool 8799e471 is started, whose pool map version is **41**.
- 2 The 2nd line means the rebuild *for* pool 8799e471 is *in* scanning phase, and no objects & records are being rebuilt yet.
- 3 The 3rd line means a rebuild job *for* pool 419d9c11 is being queued.
- 4 The 4th line means the rebuild *for* pool 419d9c11 is started, whose pool map version is **2**.
- 5 The 5th line means the rebuild *for* pool 419d9c11 is *in* scanning phase, and no objects & records are being rebuilt yet.
- 6 The 6th line means the rebuild *for* pool 8799e471 is *in* pulling phase, and there are **75** objects to be rebuilt(*toberb\_obj=75*), and all of them are rebuilt(*rb\_obj=75*), but records rebuilt *for* these objects are not finished yet(*done 0*) and only **11937** records (*rec = 11937*) are rebuilt.
- 7 The 7th line means the rebuild *for* pool 419d9c11 is *done* (*done 1*), and there are totally **10** objects and **1026** records are rebuilt, which costs about **8** seconds.
- 8 The 8th line means the rebuild *for* pool 8799e471 is *done* (*done 1*), and there are totally **75** objects and **13184** records are rebuilt, which costs about **14** seconds.

在重建期间，如果客户端向Pool Leader查询Pool 状态，则Pool Leader也会将其重建状态返回到客户端。

```

1  struct daos_rebuild_status {
2      /** pool map version in rebuilding or last completed rebuild */
3      uint32_t          rs_version;
4      /** Time (Seconds) for the rebuild */
5      uint32_t          rs_seconds;
6      /** errno for rebuild failure */
7      int32_t           rs_errno;
8      /**
9       * rebuild state, DRS_COMPLETED is valid only if @rs_version is non-ze
10      ro
11      */
12      union {
13          int32_t          rs_state;
14          int32_t          rs_done;
15      };
16      /* padding of rebuild status */
17      int32_t           rs_padding32;
18
19      /* Failure on which rank */
20      int32_t           rs_fail_rank;
21      /** # total to-be-rebuilt objects, it's non-zero and increase when
22       * rebuilding in progress, when rs_state is DRS_COMPLETED it will
23       * not change anymore and should equal to rs_obj_nr. With both
24       * rs_toberb_obj_nr and rs_obj_nr the user can know the progress
25       * of the rebuilding.
26      */
27      uint64_t          rs_toberb_obj_nr;
28      /** # rebuilt objects, it's non-zero only if rs_state is completed */
29      uint64_t          rs_obj_nr;
30      /** # rebuilt records, it's non-zero only if rs_state is completed */
31      uint64_t          rs_rec_nr;
32
33      /** rebuild space cost */
34      uint64_t          rs_size;
35  };

```

## rebuild失败

如果重建由于某些故障而失败，它将被中止，并且相关消息将显示在领导者控制台上。例如：

```
1 ▾ Rebuild [aborted] (pool 8799e471 ver=41, toerb_obj=75, rb_obj=75, rec= 119
 37, done 1 status 0 duration=10 secs)
```

## 使用校验和rebuild

在重建期间，正在重建的服务器将充当 DAOS 客户端，因为它将从副本服务器读取数据和校验和，并在其用于重建之前验证数据的完整性。如果检测到损坏的数据，则读取将失败，并且副本服务器将收到损坏的通知。然后，重建将尝试使用其他复制副本。

校验和 iov 参数可用于对象列表和对象提取任务 API。这是为了重建，以提供校验和可以打包到的内存。否则，重新生成必须在写入本地 VOS 实例时重新计算校验和。如果在缓冲区中分配的内存不足，则 iov\_len 将设置为所需的容量，并且打包到缓冲区中的校验和将被截断。

下面描述了用于重建的校验和生命周期的“Touch Points”。此处包含客户端任务 API 和打包/解压缩信息，因为 rebuild 是校验和 API 的主要调用者。

### Rebuild Touch Points

- `migrate_fetch_update_ (inline|single|bulk)` – 本地写入 vos 的重建/迁移函数必须确保也写入校验和。这些必须使用 `csum iov` 参数进行获取以获取校验和，然后将 csums 解压缩到 `iod_csum`。
- `obj_enum.c` 用于枚举要重建的对象。由于 `fetch_update` 函数将从 `fetch` 中解压缩 `csum`，因此它还会解压缩 `csum` 以进行枚举，因此 `obj_enum.c` 中的解压缩过程只需将 `csum iov` 复制到 `enum_unpack_recxs ()` 中的 `io (dc_obj_enum_unpack_io)` 结构，然后深度复制到 `migrate_one_insert()` 中的 `mrone (migrate_one)` 结构。

### Client Task API Touch Points

- `dc_obj_fetch_task_create`：将 `csum iov` 设置为 `daos_obj_fetch_t args`。这些参数设置为 `rw_cb_args.shard_args.api_args`，并通过 `cli_shard.c` 中的访问器函数 (`rw_args2csum iov`) 进行访问，以便 `rw_args_store_csum` 可以轻松访问它。从 `dc_rw_cb_csum_verify` 调用的此函数将从服务器接收的数据校验和打包到 `iov` 中。
- `dc_obj_list_obj_task_create`：将 `csum iov` 设置为 `daos_obj_list_obj_t args.dc_obj_shard_list ()` 中的 `args.csum` 复制到 `obj_enum_args.csum`。在枚举回调 (`dc_enumerate_cb ()`) 上，打包的 `csum` 缓冲区从 `rpc args` 复制到 `obj_enum_args.csum`（它指向与调用方相同的缓冲区）

### Packing/unpacking checksums

打包校验和（用于读取或对象列表）时，仅包括数据校验和。对于对象列表，仅包括内联数据的校验和。在重建期间，如果数据未内联，则重建过程将获取其余数据并获取校验和。

- `ci_serialize ()` – 通过将结构附加到 iov，然后将校验和信息缓冲区附加到 iov 来“打包”校验和。这会将实际校验和放在描述校验和的校验和结构之后。
- `ci_cast ()` – “解包”校验和和描述结构。它通过将 iov 的缓冲区强制转换为 `dcs_csum_info` 结构，并将 `csum_info` 的校验和指针设置为指向紧靠结构之后的内存来实现此目的。它没有复制任何东西，但实际上只是“投射”。要获取所有 `dcs_csum_infos`，调用方将转换 iov，将 `csum_info` 复制到目标，然后移动到 iov 中的下一个 `csum_info` (`ci_move_next iov`)。由于此过程修改了 iov 结构，因此最好使用 iov 的副本作为临时结构。

## Daos pool 相关函数

### Storage Target

Target 的类型有 4 种：

```
▼ C 复制代码
1 typedef enum {
2     DAOS_TP_UNKNOWN,      // 未知
3     DAOS_TP_HDD,          // 机械硬盘
4     DAOS_TP_SSD,          // 闪存
5     DAOS_TP_PM,           // 持久内存
6     DAOS_TP_VM,           // 易失性内存
7 } daos_target_type_t;
```

Target 当前的状态有 6 种：

```
1 ▾ typedef enum {
2     // 未知
3     DAOS_TS_UNKNOWN,
4     // 不可用
5     DAOS_TS_DOWN_OUT,
6     // 不可用, 可能需要重建
7     DAOS_TS_DOWN,
8     // 启动
9     DAOS_TS_UP,
10    // 启动并运行
11    DAOS_TS_UP_IN,
12    // Pool 映射改变导致的中间状态
13    DAOS_TS_NEW,
14    // 正在被清空
15    DAOS_TS_DRAIN,
16 } daos_target_state_t;
```

C | 复制代码

结构体 **daos\_target\_perf\_t** 用于描述 Target 的性能:

```
1 ▾ typedef struct {
2     // TODO: 存储/网络带宽、延迟等
3     int foo;
4 } daos_target_perf_t;
```

C | 复制代码

结构体 **daos\_space** 表示 Pool Target 的空间使用情况:

```
1 ▾ struct daos_space {
2     uint64_t s_total[DAOS_MEDIA_MAX];    // 全部空间 (字节)
3     uint64_t s_free[DAOS_MEDIA_MAX];     // 空闲空间 (字节)
4 };
```

C | 复制代码

其中, **DAOS\_MEDIA\_MAX** 表示存储空间介质的数量, 一共有两种:

```
1 enum {
2     DAOS_MEDIA_SCM = 0,
3     DAOS_MEDIA_NVME,
4     DAOS_MEDIA_MAX
5 };
```

C | 复制代码

即 `s_total[DAOS_MEDIA_SCM]` 和 `s_free[DAOS_MEDIA_SCM]` 表示 SCM (Storage-Class Memory) 的使用信息, `s_total[DAOS_MEDIA_NVME]` 和 `s_free[DAOS_MEDIA_NVME]` 表示 NVMe (Non-Volatile Memory express) 的使用信息。

结构体 `daos_target_info_t` 表示 Target 的信息:

```
1 typedef struct {
2     daos_target_type_t ta_type;      // 类型
3     daos_target_state_t ta_state;    // 状态
4     daos_target_perf_t ta_perf;     // 性能
5     struct daos_space ta_space;    // 空间使用情况
6 } daos_target_info_t;
```

C | 复制代码

## Storage Pool

结构体 `daos_pool_space` 表示 Pool 的空间使用情况:

```
1 struct daos_pool_space {
2     // 所有活动的 Target 的聚合空间
3     struct daos_space ps_space;
4     // 所有 Target 中的最大可用空间 (字节)
5     uint64_t          ps_free_min[DAOS_MEDIA_MAX];
6     // 所有 Target 中的最小可用空间 (字节)
7     uint64_t          ps_free_max[DAOS_MEDIA_MAX];
8     // Target 平均可用空间 (字节)
9     uint64_t          ps_free_mean[DAOS_MEDIA_MAX];
10    // Target(VOS, Versioning Object Store) 数量
11    uint32_t          ps_ntargets;
12    uint32_t          ps_padding;
13};
```

结构体 `daos_rebuild_status` 表示重建状态：

C | 复制代码

```
1 struct daos_rebuild_status {
2     // Pool 映射在重建过程中的版本或上一个完成重建的版本
3     uint32_t rs_version;
4     // 重建的时间 (秒)
5     uint32_t rs_seconds;
6     // 重建错误的错误码
7     int32_t rs_errno;
8     // 重建是否完成 该字段只有在 rs_version 非 0 时有效
9     int32_t rs_done;
10    // 重建状态的填充
11    int32_t rs_padding32;
12    // 失败的 rank
13    int32_t rs_fail_rank;
14
15    // 要重建的对象总数, 它不为 0 并且在重建过程中增加
16    // 当 rs_done = 1 时, 它将不再更改, 并且应等于 rs_obj_nr
17    // 使用 rs_toberb_obj_nr 和 rs_obj_nr, 用户可以知道重建的进度
18    uint64_t rs_toberb_obj_nr;
19    // 重建的对象数量, 该字段非 0 当且仅当 rs_done = 1
20    uint64_t rs_obj_nr;
21    // 重建的记录数量, 该字段非 0 当且仅当 rs_done = 1
22    uint64_t rs_rec_nr;
23    // 重建的空间开销
24    uint64_t rs_size;
25 };
```

C | 复制代码

```
1 enum daos_pool_info_bit {
2     // 如果为真, 查询 Pool 的空间使用情况
3     DPI_SPACE = 1ULL << 0,
4     // 如果为真, 查询重建状态
5     DPI_REBUILD_STATUS = 1ULL << 1,
6     // 查询所有的可选信息
7     DPI_ALL = -1,
8 };
```

结构体 **daos\_pool\_info\_t** 表示 Pool 的信息:

C | 复制代码

```
1 ▾ typedef struct {
2     // UUID
3     uuid_t      pi_uuid;
4     // Target 数量
5     uint32_t    pi_ntargets;
6     // Node 数量
7     uint32_t    pi_nnodes;
8     // 不活跃的 Target 数量
9     uint32_t    pi_ndisabled;
10    // 最新的 Pool 映射版本
11    uint32_t    pi_map_ver;
12    // 当前的 Raft Leader
13    uint32_t    pi_leader;
14    // Pool 信息查询位, 其值为枚举类型 daos_pool_info_bit
15    uint64_t    pi_bits;
16    // 空间使用情况
17    struct daos_pool_space pi_space;
18    // 重建状态
19    struct daos_rebuild_status pi_rebuild_st;
20 } daos_pool_info_t;
```

对于每个 `daos_pool_query()` 调用, 将始终查询基本 Pool 信息, 如从 `pi_uuid` 到 `pi_leader` 的字段。但是 `pi_space` 和 `pi_rebuild_st` 是基于 `pi_bits` 的可选查询字段。

结构体 `daos_pool_cont_info` 表示 Pool Container 的信息:

C | 复制代码

```
1 ▾ struct daos_pool_cont_info {
2     // UUID
3     uuid_t  pci_uuid;
4 };
```

## daos\_pool\_connect

`daos_pool_connect` 函数连接到由 UUID `uuid` 标识的 DAOS Pool。

成功执行后, `poh` 返回 Pool 句柄, `info` 返回最新的 Pool 信息。

参数:

- `uuid [in]`: 标识 Pool 的 UUID。
- `grp [in]`: 管理 Pool 的 DAOS 服务器的进程集合名称。
- `flags [in]`: 由 `DAOS_PC_` 位表示的连接模式。
- `poh [out]`: 返回的打开句柄。
- `info [in, out]`: 可选参数, 返回的 Pool 信息, 参考枚举类型 `daos_pool_info_bit`。
- `ev [in]`: 结束事件, 该参数是可选的, 可以为 `NULL`。当该参数为 `NULL` 时, 该函数在阻塞模式下运行。

返回值, 在非阻塞模式下会被写入 `ev::ev_error`:

- 如果成功, 返回 0。
- 如果失败, 返回
  - `-DER_INVAL`: 无效的参数。
  - `-DER_UNREACH`: 无法访问网络。
  - `-DER_NO_PERM`: 没有访问权限。
  - `-DER_NONEXIST`: Pool 不存在。

```

1 int
2 daos_pool_connect(const uuid_t uuid, const char *grp,
3                     unsigned int flags,
4                     daos_handle_t *poh, daos_pool_info_t *info, daos_event_t *ev)
5 {
6     daos_pool_connect_t *args;
7     tse_task_t *task;
8     int rc;
9
10    // 判断 *args 大小是否与 daos_pool_connect_t 的预期大小相等
11    DAOS_API_ARG_ASSERT(*args, POOL_CONNECT);
12    if (!daos_uuid_valid(uuid))
13        // UUID 无效
14        return -DER_INVAL;
15
16    // 创建新任务 dc_pool_connect, 并将其与输入事件 ev 关联
17    // 如果事件 ev 为 NULL, 则将获取私有事件
18    rc = dc_task_create(dc_pool_connect, NULL, ev, &task);
19    if (rc)
20        // dc_task_create 成功返回 0, 失败返回负数
21        return rc;
22
23    // 从 task 中获取参数
24    args = dc_task_get_args(task);
25    args->grp = grp;
26    args->flags = flags;
27    args->poh = poh;
28    args->info = info;
29    uuid_copy((unsigned char *)args->uuid, uuid);
30
31    // 调度创建的任务 task
32    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
33    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
34    //
35    // 第二个参数 instant 为 true, 表示任务将立即执行
36    return dc_task_schedule(task, true);
37 }

```

Pool 的连接模式有三种, 由 `DAOS_PC_` 位表示:

C | 复制代码

```
1 // 以只读模式连接 Pool
2 #define DAOS_PC_R0      (1U << 0)
3 // 以读写模式连接 Pool
4 #define DAOS_PC_RW       (1U << 1)
5 // 以独占读写模式连接到 Pool
6 // 如果当前存在独占 Pool 句柄，则不允许与 DSM_PC_RW 模式的连接。
7 #define DAOS_PC_EX       (1U << 2)
8
9 // 表示连接模式的位个数
10 #define DAOS_PC_NBITS    3
11 // 连接模式位掩码
12 #define DAOS_PC_MASK     ((1U << DAOS_PC_NBITS) - 1)
```

结构体 `daos_pool_connect_t` 表示 Pool 连接参数：

C | 复制代码

```
1 typedef struct {
2     // Pool 的 UUID
3     uuid_t          uuid;
4     // 管理 Pool 的 DAOS 服务器的进程集合名称。
5     const char      *grp;
6     // 由 DAOS_PC_ 位表示的连接模式
7     unsigned int    flags;
8     // 返回的打开句柄
9     daos_handle_t  *poh;
10    // 可选，返回的 Pool 信息
11    daos_pool_info_t *info;
12 } daos_pool_connect_t;
```

## daos\_pool\_disconnect

`daos_pool_disconnect` 函数断开 DAOS Pool 的连接。它应该撤销该 Pool 的所有打开的 Container 句柄。

参数：

- `poh [in]`: 连接到 Pool 的句柄。
- `ev [in]`: 结束事件，该参数是可选的，可以为 `NULL`。当该参数为 `NULL` 时，该函数在阻塞模式下运行。

返回值，在非阻塞模式下会被写入 `ev::ev_error`：

- 如果成功，返回 0。
- 如果失败，返回
  - `-DER_UNREACH`: 无法访问网络。
  - `-DER_NO_HDL`: Pool 句柄无效。

```

1 int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev)
2 {
3     daos_pool_disconnect_t *args;
4     tse_task_t *task;
5     int rc;
6
7     // 判断 *args 大小是否与 daos_pool_disconnect_t 的预期大小相等
8     DAOS_API_ARG_ASSERT(*args, POOL_DISCONNECT);
9     // 创建新任务 dc_pool_disconnect，并将其与输入事件 ev 关联
10    // 如果事件 ev 为 NULL，则将获取私有事件
11    rc = dc_task_create(dc_pool_disconnect, NULL, ev, &task);
12    if (rc)
13        // dc_task_create 成功返回 0，失败返回负数
14        return rc;
15
16    // 从 task 中获取参数
17    args = dc_task_get_args(task);
18    args->poh = poh;
19
20    // 调度创建的任务 task
21    // 如果该任务的关联事件是私有事件，则此函数将等待任务完成
22    // 否则它将立即返回，并通过测试事件或在 EQ 上轮询找到其完成情况
23    //
24    // 第二个参数 instant 为 true，表示任务将立即执行
25    return dc_task_schedule(task, true);
26 }

```

结构体 `daos_pool_disconnect_t` 表示断开 Pool 连接到参数：

```

1 typedef struct {
2     // 打开的 Pool 句柄
3     daos_handle_t poh;
4 } daos_pool_disconnect_t;

```

## daos\_pool\_local2global

```
1 int daos_pool_local2global(daos_handle_t poh, d iov_t *glob)
2 {
3     return dc_pool_local2global(poh, glob);
4 }
```

`daos_pool_local2global` 函数将本地 Pool 连接转换为可与对等进程共享的全局表示数据。

如果 `glob->iov_buf` 设置为 NULL，则通过 `glob->iov_buf_len` 返回全局句柄的实际大小。

此功能不涉及任何通信，也不阻塞。

参数：

- `poh [in]`: 要共享的打开的 Pool 连接句柄。
- `glob [out]`: 指向 IO vector 缓冲区的指针，用于存储句柄信息。

返回值，在非阻塞模式下：

- 如果成功，返回 0。
- 如果失败：
  - `-DER_INVAL`: 无效的参数。
  - `-DER_NO_HDL`: Pool 句柄无效。
  - `-DER_TRUNC`: `glob` 中的缓冲区过小，要求更大的缓冲区。在这种情况下，要求的缓冲区大学会被写入 `glob->iov_buf_len`。

## daos\_pool\_global2local

```
1 int daos_pool_global2local(d iov_t glob, daos_handle_t *poh)
2 {
3     return dc_pool_global2local(glob, poh);
4 }
```

`daos_pool_global2local` 函数为全局表示数据创建本地 Pool 连接。

参数：

- `glob [in]`: 要提取的集合句柄的全局（共享）表示。

- **poh [out]**: 返回的本地 Pool 连接句柄。

返回值，在非阻塞模式下：

- 如果成功，返回 0。
- 如果失败，返回
  - **-DER\_INVAL**: 无效的参数。

```
int daos_pool_global2local(d_iov_t glob, daos_handle_t *poh) {    return  
dc_pool_global2local(glob, poh); }
```

## daos\_pool\_query

C | 复制代码

```
1 int daos_pool_query(daos_handle_t poh, d_rank_list_t *tgts, daos_pool_info
2 _t *info,
3         daos_prop_t *pool_prop, daos_event_t *ev)
4 {
5     daos_pool_query_t *args;
6     tse_task_t *task;
7     int rc;
8
9     // 判断 *args 大小是否与 daos_pool_query_t 的预期大小相等
10    DAOSS_API_ARG_ASSERT(*args, POOL_QUERY);
11
12    if (pool_prop != NULL && !daos_prop_valid(pool_prop, true, false)) {
13        // 无效输入
14        D_ERROR("invalid pool_prop parameter.\n");
15        return -DER_INVAL;
16    }
17
18    // 创建新任务 dc_pool_query, 并将其与输入事件 ev 关联
19    // 如果事件 ev 为 NULL, 则将获取私有事件
20    rc = dc_task_create(dc_pool_query, NULL, ev, &task);
21    if (rc)
22        // dc_task_create 成功返回 0, 失败返回负数
23        return rc;
24
25    // 从 task 中获取参数
26    args = dc_task_get_args(task);
27    args->poh = poh;
28    args->tgts = tgts;
29    args->info = info;
30    args->prop = pool_prop;
31
32    // 调度创建的任务 task
33    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
34    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
35    // 第二个参数 instant 为 true, 表示任务将立即执行
36    return dc_task_schedule(task, true);
37 }
```

`daos_pool_query` 函数查询 Pool 信息。用户应至少提供 `info` 和 `tgts` 中的一个作为输出缓冲区。

参数:

- `poh [in]`: Pool 连接句柄。
- `tgts [out]`: 可选, 返回的 Pool 中的 Target。

- `info [in, out]`: 可选, 返回的 Pool 信息, 参考枚举类型 `daos_pool_info_bit`。
- `pool_prop [out]`: 可选, 返回的 Pool 属性。
  - 如果为空, 则不需要查询属性。
  - 如果 `pool_prop` 非空, 但其 `dpp_entries` 为空, 则将查询所有 Pool 属性, DAOS 在内部分配所需的缓冲区, 并将指针分配给 `dpp_entries`。
  - 如果 `pool_prop` 的 `dpp_nr > 0` 且 `dpp_entries` 非空, 则会查询特定的 `dpe_type` 属性, DAOS 会在内部为 `dpe_str` 或 `dpe_val_ptr` 分配所需的缓冲区, 如果具有立即数的 `dpe_type` 则会直接将其分配给 `dpe_val`。
  - 用户可以通过调用 `daos_prop_free()` 释放关联的缓冲区。
- `ev [in]`: 结束事件, 该参数是可选的, 可以为 `NULL`。当该参数为 `NULL` 时, 该函数在阻塞模式下运行。

返回值, 在非阻塞模式下:

- 如果成功, 返回 0。
- 如果失败:
  - `-DER_INVAL`: 无效的参数。
  - `-DER_UNREACH`: 无法访问网络。
  - `-DER_NO_HDL`: Pool 句柄无效。

结构体 `daos_prop_t` 表示 DAOS

Pool 或 Container 的属性:

```

1 ▾ typedef struct {
2     // 项的数量
3     uint32_t             dpp_nr;
4     // 保留供将来使用, 现在用于 64 位对齐
5     uint32_t             dpp_reserv;
6     // 属性项数组
7     struct daos_prop_entry *dpp_entries;
8 } daos_prop_t;

```

DAOS Pool 的属性类型包括:

C | 复制代码

```
1 enum daos_pool_props {
2     // 在 (DAOS_PROP_PO_MIN, DAOS_PROP_PO_MAX) 范围内有效
3     DAOS_PROP_PO_MIN = 0,
4
5     // 标签: 用户与 Pool 关联的字符串
6     // default = ""
7     DAOS_PROP_PO_LABEL,
8
9     // ACL: Pool 的访问控制列表
10    // 详细说明用户和组访问权限的访问控制项的有序列表。
11    // 期望的主体类型: Owner, User(s), Group(s), Everyone
12    DAOS_PROP_PO_ACL,
13
14    // 保留空间比例: 每个 Target 上为重建目的保留的空间量。
15    // default = 0%.
16    DAOS_PROP_PO_SPACE_RB,
17
18    // 自动/手动 自我修复
19    // default = auto
20    // 自动/手动 排除
21    // 自动/手动 重建
22    DAOS_PROP_PO_SELF_HEAL,
23
24    // 空间回收策略 = time|batched|snapshot
25    // default = snapshot
26    // time: 时间间隔
27    // batched: commits
28    // snapshot: 快照创建
29    DAOS_PROP_PO_RECLAIM,
30
31    // 充当 Pool 所有者的用户
32    // 格式: user@[domain]
33    DAOS_PROP_PO_OWNER,
34
35    // 充当 Pool 所有者的组
36    // 格式: group@[domain]
37    DAOS_PROP_PO_OWNER_GROUP,
38
39    // Pool 的 svc rank list
40    DAOS_PROP_PO_SVC_LIST,
41
42    DAOS_PROP_PO_MAX,
43 };
44
45 // Pool 属性类型数量
```

```
46 #define DAOS_PROP_PO_NUM      (DAOS_PROP_PO_MAX - DAOS_PROP_PO_MIN - 1)
```

结构体 `daos_pool_query_t` 表示 Pool 查询的参数：

```
1 typedef struct {
2     // 打开的 Pool 句柄
3     daos_handle_t      poh;
4     // 可选, 返回的 Pool 中的 Target
5     d_rank_list_t      *tgts;
6     // 可选, 返回的 Pool 信息
7     daos_pool_info_t   *info;
8     // 可选, 返回的 Pool 属性
9     daos_prop_t        *prop;
10 } daos_pool_query_t;
```

C | 复制代码

## daos\_pool\_query\_target

C | 复制代码

```
1 int daos_pool_query_target(daos_handle_t poh, uint32_t tgt, d_rank_t rank,
2                               daos_target_info_t *info, daos_event_t *ev)
3 {
4     daos_pool_query_target_t *args;
5     tse_task_t *task;
6     int rc;
7
8     // 判断 *args 大小是否与 daos_pool_query_target_t 的预期大小相等
9     DAOS_API_ARG_ASSERT(*args, POOL_QUERY_INFO);
10
11    // 创建新任务 dc_pool_query_target, 并将其与输入事件 ev 关联
12    // 如果事件 ev 为 NULL, 则将获取私有事件
13    rc = dc_task_create(dc_pool_query_target, NULL, ev, &task);
14    if (rc)
15        // dc_task_create 成功返回 0, 失败返回负数
16        return rc;
17
18    // 从 task 中获取参数
19    args = dc_task_get_args(task);
20    args->poh = poh;
21    args->tgt_idx = tgt_idx;
22    args->rank = rank;
23    args->info = info;
24
25    // 调度创建的任务 task
26    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
27    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
28    //
29    // 第二个参数 instant 为 true, 表示任务将立即执行
30    return dc_task_schedule(task, true);
31 }
```

`daos_pool_query_target` 函数在 DAOS Pool 中查询 Target 信息。

参数:

- `poh [in]`: Pool 连接句柄。
- `tgt [in]`: 要查询的单个 Target 的索引。
- `rank [in]`: 要查询的 Target 索引的排名。
- `info [out]`: 返回的有关 `tgt` 的信息。
- `ev [in]`: 结束事件, 该参数是可选的, 可以为 `NULL`。当该参数为 `NULL` 时, 该函数在阻塞模式下运行。

返回值，在非阻塞模式下：

- 如果成功，返回 0。
- 如果失败：
  - `-DER_INVAL`: 无效的参数。
  - `-DER_UNREACH`: 无法访问网络。
  - `-DER_NO_HDL`: Pool 句柄无效。
  - `-DER_NONEXIST`: 指定 Target 上没有 Pool。

结构体 `daos_pool_query_target_t` 表示 Pool 的 Target 查询参数：

```
1 ▾ typedef struct {  
2     // 打开的 Pool 句柄  
3     daos_handle_t      poh;  
4     // 要查询的单个 Target  
5     uint32_t           tgt_idx;  
6     // 要查询的 Target 的等级  
7     d_rank_t          rank;  
8     // 返回的 Target 信息  
9     daos_target_info_t *info;  
10 } daos_pool_query_target_t;
```

## daos\_pool\_list\_attr

C | 复制代码

```
1 int daos_pool_list_attr(daos_handle_t poh, char *buffer, size_t *size,
2                           daos_event_t *ev)
3 {
4     daos_pool_list_attr_t *args;
5     tse_task_t *task;
6     int rc;
7
8     // 判断 *args 大小是否与 daos_pool_list_attr_t 的预期大小相等
9     DAOS_API_ARG_ASSERT(*args, POOL_LIST_ATTR);
10
11    // 创建新任务 dc_pool_list_attr, 并将其与输入事件 ev 关联
12    // 如果事件 ev 为 NULL, 则将获取私有事件
13    rc = dc_task_create(dc_pool_list_attr, NULL, ev, &task);
14    if (rc)
15        // dc_task_create 成功返回 0, 失败返回负数
16        return rc;
17
18    // 从 task 中获取参数
19    args = dc_task_get_args(task);
20    args->poh = poh;
21    args->buf = buf;
22    args->size = size;
23
24    // 调度创建的任务 task
25    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
26    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
27    //
28    // 第二个参数 instant 为 true, 表示任务将立即执行
29    return dc_task_schedule(task, true);
30 }
```

`daos_pool_list_attr` 函数列出所有用户定义的 Pool 属性的名称。

参数:

- `poh [in]`: Pool 句柄。
- `buffer [out]`: 包含所有属性名的串联的缓冲区, 每个属性名以空字符结尾。不执行截断, 只返回全名。允许为 `NULL`, 在这种情况下, 只检索聚合大小。
- `size [in, out]`:
  - `[in]`: 缓冲区大小。
  - `[out]`: 所有属性名 (不包括终止的空字符) 的聚合大小, 忽略实际缓冲区大小。
- `ev [in]`: 结束事件, 该参数是可选的, 可以为 `NULL`。当该参数为 `NULL` 时, 该函数在阻塞模

式下运行。

## daos\_pool\_get\_attr

```
1 int daos_pool_get_attr(daos_handle_t poh, int n, char const *const names
2 [ ],
3             void *const buffers[], size_t sizes[], daos_event_t *ev)
4 {
5     daos_pool_get_attr_t *args;
6     tse_task_t *           task;
7     int                   rc;
8
9     // 判断 *args 大小是否与 daos_pool_get_attr_t 的预期大小相等
10    DAOS_API_ARG_ASSERT(*args, POOL_GET_ATTR);
11
12    // 创建新任务 dc_pool_get_attr, 并将其与输入事件 ev 关联
13    // 如果事件 ev 为 NULL, 则将获取私有事件
14    rc = dc_task_create(dc_pool_get_attr, NULL, ev, &task);
15    if (rc)
16        // dc_task_create 成功返回 0, 失败返回负数
17        return rc;
18
19    // 从 task 中获取参数
20    args      = dc_task_get_args(task);
21    args->poh   = poh;
22    args->n     = n;
23    args->names = names;
24    args->values = values;
25    args->sizes = sizes;
26
27    // 调度创建的任务 task
28    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
29    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
30    //
31    // 第二个参数 instant 为 true, 表示任务将立即执行
32    return dc_task_schedule(task, true);
33 }
```

daos\_pool\_get\_attr 函数获取用户定义的 Pool 属性值列表。

参数:

- **poh [in]**: Pool 句柄。
- **n [in]**: 属性的数量。
- **names [in]**: 存储以空字符结尾的属性名的 **n** 个数组。
- **buffer [out]**: 存储属性值的 **n** 个缓冲区的数组。大于相应缓冲区大小的属性值将被截断。允许为 **NULL**, 将被视为与零长度缓冲区相同, 在这种情况下, 只检索属性值的大小。
- **sizes [in, out]**:
  - **[in]**: 存储缓冲区大小的 **n** 个数组。
  - **[out]**: 存储属性值实际大小的 **n** 个数组, 忽略实际缓冲区大小。
- **ev [in]**: 结束事件, 该参数是可选的, 可以为 **NULL**。当该参数为 **NULL** 时, 该函数在阻塞模式下运行。

结构体 `daos_pool_get_attr_t` 表示 Pool 获取属性的参数:

```

1  typedef struct {
2      // 打开的 Pool 句柄
3      daos_handle_t          poh;
4      // 属性数量
5      int                  n;
6      // 存储 n 个以空字符结尾的属性名的
7      char const *const    *names;
8      // 存储 n 个属性值的缓冲区
9      void *const          *values;
10     // [in]: 存储 n 个缓冲区大小
11     // [out]: 存储 n 个属性值实际大小
12     size_t              *sizes;
13 } daos_pool_get_attr_t;

```

## daos\_pool\_set\_attr

```

1  int daos_pool_set_attr(daos_handle_t poh, int n, char const *const names
2  [],
3  void const *const values[], size_t const sizes[],
4  daos_event_t *ev)
5  { ... }

```

`daos_pool_set_attr` 函数创建或更新用户定义的 Pool 属性值列表。

参数：

- **poh [in]**: Pool 句柄。
- **n [in]**: 属性的数量。
- **names [in]**: 存储以空字符结尾的属性名的 **n** 个数组。
- **values [in]**: 存储属性值的 **n** 个数组。
- **sizes [in]**: 存储相应属性值大小的 **n** 个元素的数组。
- **ev [in]**: 结束事件，该参数是可选的，可以为 **NULL**。当该参数为 **NULL** 时，该函数在阻塞模式下运行。

结构体 **daos\_pool\_set\_attr\_t** 表示 Pool 设置属性的参数：

```
1 typedef struct {  
2     // 打开的 Pool 句柄  
3     daos_handle_t          poh;  
4     // 属性的数量  
5     int                  n;  
6     // 存储 n 个以空字符结尾的属性名  
7     char const *const    *names;  
8     // 存储 n 个属性值  
9     void const *const    *values;  
10    // 存储 n 个相应属性值的大小。  
11    size_t const        *sizes;  
12 } daos_pool_set_attr_t;
```

## daos\_pool\_del\_attr

C | 复制代码

```
1 int daos_pool_del_attr(daos_handle_t poh, int n, char const *const names
2 [],  
3 {  
4     daos_pool_del_attr_t *args;  
5     tse_task_t *          task;  
6     int                  rc;  
7  
8     // 判断 *args 大小是否与 daos_pool_del_attr_t 的预期大小相等  
9     DAOS_API_ARG_ASSERT(*args, POOL_DEL_ATTR);  
10  
11    // 创建新任务 dc_pool_del_attr, 并将其与输入事件 ev 关联  
12    // 如果事件 ev 为 NULL, 则将获取私有事件  
13    rc = dc_task_create(dc_pool_del_attr, NULL, ev, &task);  
14    if (rc)  
15        // dc_task_create 成功返回 0, 失败返回负数  
16        return rc;  
17  
18    // 从 task 中获取参数  
19    args      = dc_task_get_args(task);  
20    args->poh  = poh;  
21    args->n   = n;  
22    args->names = names;  
23  
24    // 调度创建的任务 task  
25    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成  
26    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况  
27    //  
28    // 第二个参数 instant 为 true, 表示任务将立即执行  
29    return dc_task_schedule(task, true);  
30 }
```

`daos_pool_del_attr` 函数删除用户定义的 Pool 属性值列表。

参数:

- `poh [in]`: Pool 句柄。
- `n [in]`: 属性的数量。
- `names [in]`: 存储以空字符结尾的属性名的 `n` 个数组。
- `ev [in]`: 结束事件, 该参数是可选的, 可以为 `NULL`。当该参数为 `NULL` 时, 该函数在阻塞模式下运行。

返回值, 在非阻塞模式下会被写入 `ev::ev_error`:

- 如果成功，返回 0。
- 如果失败，返回
  - `-DER_INVAL`: 无效的参数。
  - `-DER_UNREACH`: 无法访问网络。
  - `-DER_NO_PERM`: 没有访问权限。
  - `-DER_NO_HDL`: 无效的 Container 句柄。
  - `-DER_NOMEM`: 内存不足。

结构体 `daos_pool_del_attr_t` 表示 Pool 删除属性的参数：

```
1 typedef struct {  
2     // 打开的 Pool 句柄  
3     daos_handle_t      poh;  
4     // 属性的数量  
5     int                 n;  
6     // 存储 n 个以空字符结尾的属性名  
7     char const *const   *names;  
8 } daos_pool_del_attr_t;
```

## daos\_pool\_list\_cont

C | 复制代码

```
1 int daos_pool_list_cont(daos_handle_t poh, daos_size_t *ncont,
2                         struct daos_pool_cont_info *cbuf, daos_event_t *ev)
3 {
4     daos_pool_list_cont_t *args;
5     tse_task_t *task;
6     int rc;
7
8     // 判断 *args 大小是否与 daos_pool_list_cont_t 的预期大小相等
9     DAOS_API_ARG_ASSERT(*args, POOL_LIST_CONT);
10
11    if (ncont == NULL) {
12        // 无效输入
13        D_ERROR("ncont must be non-NULL\n");
14        return -DER_INVAL;
15    }
16
17    // 创建新任务 dc_pool_list_cont, 并将其与输入事件 ev 关联
18    // 如果事件 ev 为 NULL, 则将获取私有事件
19    rc = dc_task_create(dc_pool_list_cont, NULL, ev, &task);
20    if (rc)
21        // dc_task_create 成功返回 0, 失败返回负数
22        return rc;
23
24    // 从 task 中获取参数
25    args = dc_task_get_args(task);
26    args->poh = poh;
27    args->ncont = ncont;
28    args->cont_buf = cbuf;
29
30    // 调度创建的任务 task
31    // 如果该任务的关联事件是私有事件, 则此函数将等待任务完成
32    // 否则它将立即返回, 并通过测试事件或在 EQ 上轮询找到其完成情况
33    //
34    // 第二个参数 instant 为 true, 表示任务将立即执行
35    return dc_task_schedule(task, true);
36 }
```

`daos_pool_list_cont` 函数列出 Pool 的 Container。

参数:

- `poh [in]`: Pool 连接句柄。
- `ncont [in, out]`:
  - `[in]`: 以元素为单位的 `cbuf` 长度。

- [out]: Pool 中的 Container 数量。
- cbuf [out]: 存储 Container 结构的数组。允许为 NULL，在这种情况下只会讲 Container 的数量写入 ncont 返回。
- ev [in]: 结束事件，该参数是可选的，可以为 NULL。当该参数为 NULL 时，该函数在阻塞模式下运行。

返回值：

- 如果成功，返回 0。
- 如果失败，返回
  - -DER\_INVAL: 无效的参数。
  - -DER\_TRUNC: cbuf 没有足够的空间存储 ncont 个元素。