

Daos

1. Daos 存储模型

1.1 存储架构

1.1.1 Daos Target

Daos Pool

Daos Container

Daos Object

1.2 事务模型

1.2.1 Epoch

1.2.2 Container 快照

1.2.3 分布式事务

1.3 容错模型

1.3.1 分层故障域

1.3.2 错误检测

1.3.3 错误隔离

1.3.4 错误恢复

1.4 安全模型

2. Daos 内部构件

2.1 Daos 组件

2.1.1 Daos 系统

2.1.2 客户端API、Tools和I/O中间件

2.1.3 Agent

2.2 网络交互模型

2.2.1 gRPC

2.2.2 dRPC

2.2.3 CART

2.3 Daos 层级和服务

2.3.1 架构

2.3.2 源码框架

2.3.3 Daos基础库

2.3.4 Daos服务

2.4.1 Protocol Compatibility(协议兼容性)

2.4.2 PM模式的兼容性和升级

3. Daos Control Plane

3.1 Doc

3.2 Configuration

3.3 Subcommands

3.4 Shell Usage

3.5 NVMe 管理

3.6 SCM 管理

3.7 架构

4. Daos Data Plane

模块接口

线程模型与 Argobot 集成

Thread-local Storage (TLS)

Incast Variable 集成

dRPC 服务器

dRPC 进程

dRPC 处理程序注册

5. Daos 数据备份

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

7. Daos Pool

Pool Service

Pool Operations

Pool Connect

Pool 相关函数

Storage Target

Pool Target

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

8. Daos Container

container 概念

Metadata Layout

Target Service

Object ID Allocator

Container Operations

Epoch Protocol

container的相关Task 结构体

Go端container

Container cmd命令

containerBaseCmd

containerCreateCmd

C端container

[Daos Object](#)

[Daos 并发模型 \(VOS\)](#)

[Blob I/O](#)

[Daos 数据结构与算法](#)

[18. Daos 公共库](#)

[19. 客户端库和I/O中间件](#)

[4. Mga Tool](#)

[5. Daos Agent](#)

[6. Daos 验证管理](#)

[7. Daos 安全](#)

[DAOS Client Library](#)

[Client API:](#)

[Management API:](#)

[Pool Client API](#)

[Container API](#)

[Object, Array and KV APIs](#)

[Event, Event Queue, and Task API](#)

[Addons API](#)

[DFS API](#)

[DAOS Common Libraries](#)

[Task Scheduler Engine\(TSE\)](#)

[Scheduler API](#)

[任务接口](#)

[Event & Event Queue](#)

[Task Engine Integration](#)

[DPRC](#)

[枚举类型](#)

[drpc_module](#)

[Server](#)

[server struct](#)

[server](#)

[ControlServer](#)

```
mgmSvc  
srvModule  
securityModule  
server func  
入口函数 Start
```

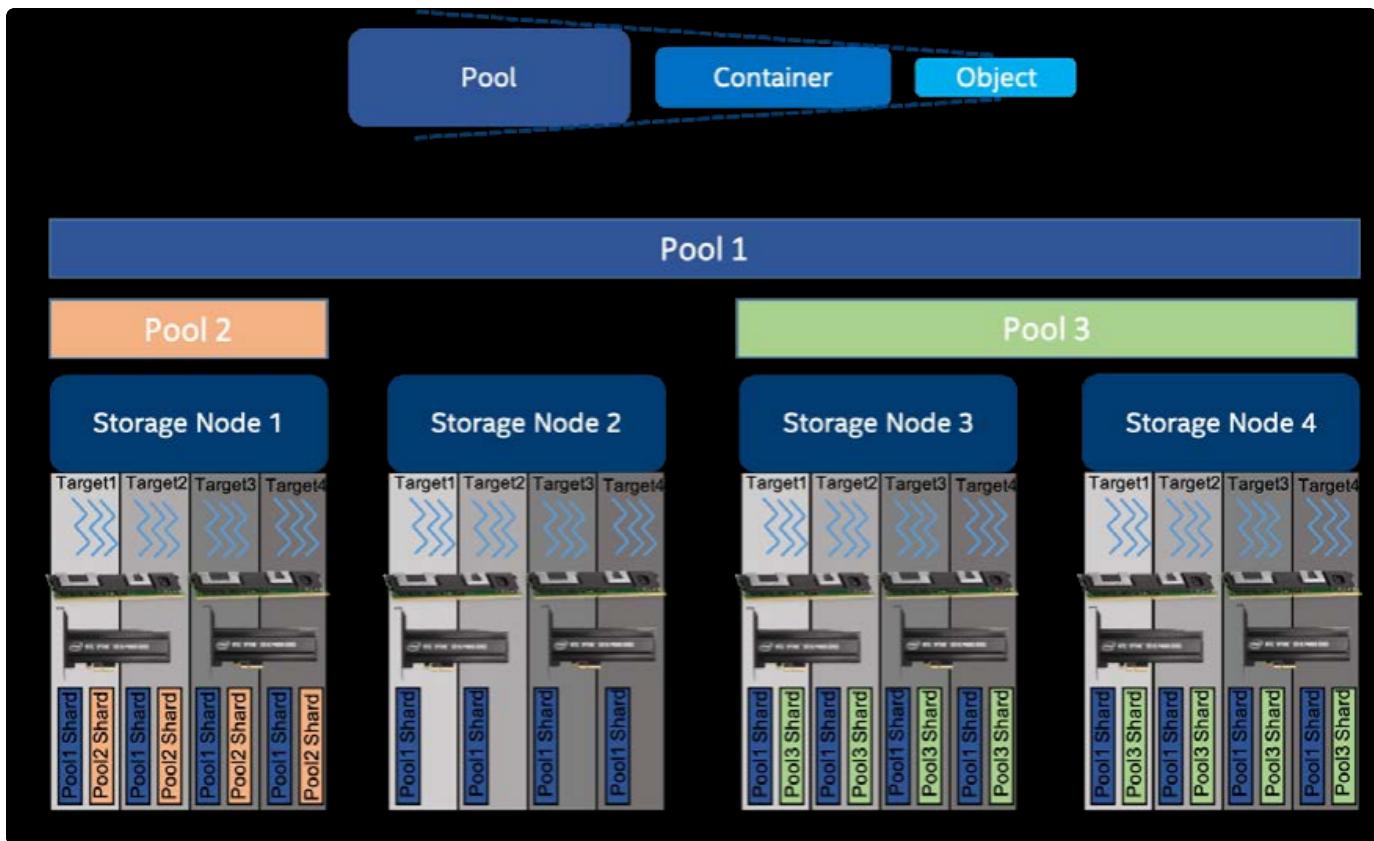
1. Daos 存储模型

1.1 存储架构

我们考虑一个数据中心，其中数十万个计算节点通过可扩展的高性能结构(即以太网、RoCE或InfinBand)相互连接，所有节点的子集，称为存储节点，可以直接访问字节寻址的存储类存储(SCM)，也可以访问基于块的NVMe存储。DAOS服务器是一个多租户守护进程，运行在每个存储节点的Linux实例上（即，在物理节点上或虚拟机或容器中）上，并通过网络导出本地连接的存储。在DAOS服务器中，存储将跨多个目标进行静态分区，以优化并发性。为了避免争用，每个目标都有其私有存储、自己的服务线程池和专用的网络上下文，它们可以在结构上独立于托管在同一存储节点上的其他目标进行直接处理。由DAOS服务器实例导出的目标的数量是可配置的，并取决于底层硬件(例如，SCM模块、cpu、NVMessd等的数量)。目标是故障的单位。连接到同一结构的所有DAOS服务器都被分组组成一个DAOS系统，由一个系

统名称标识。DAOS服务器的成员资格被记录到系统映射中，该映射为每个服务器分配一个唯一的标识。两个不同的系统由互不相关的服务器组成。

下图表示了DAOS存储模型的基本抽象部分：



1.1.1 Daos Target

Daos Pool

Daos Container

Daos Object

1.2 事务模型

1.2.1 Epoch

1.2.2 Container 快照

1.2.3 分布式事务

1.3 容错模型

1.3.1 分层故障域

1.3.2 错误检测

1.3.3 错误隔离

1.3.4 错误恢复

1.4 安全模型

2. Daos 内部构件

2.1 Daos 组件

2.1.1 Daos 系统

2.1.2 客户端API、Tools和I/O中间件

2.1.3 Agent

DAOS Agent是驻留在客户端节点上的守护进程，并通过dRPC与DAOS客户端库进行交互，以对应用程序进程进行身份验证。它是一个受信任的实体，可以使用本地证书签名DAOS客户端凭据。该代理可以支持不同的身份验证框架，并使用Unix域套接字与客户端库进行通信。DAOS代理用Go编写，并通过gRPC与每个DAOS服务器的控制平面组件进行通信，以向客户端库提供DAOS系统成员资格信息，并支持Pool列表。

2.2 网络交互模型

DAOS使用了三种不同的通信方式：gRPC、dRPC、CART

2.2.1 gRPC

gRPC为DAOS管理提供了一个双向的安全通道。它依赖于TLS/SSL来对管理员角色和服务器进行身份验证。协议缓冲区用于RPC序列化，并且所有的原型文件都位于 src\proto 目录中。

2.2.2 dRPC

dRPC是在Unix域套接字上构建的通信信道，用于进程间的通信。它提供了一个C接口和Go接口来支持以下两者之间的交互：

- 用于agent 和 libdaos 中应用程序的身份验证
- daos_server（控制平面）和daos_io_server（数据平面）守护进程，如gRPC, RPC, 通过协议缓冲区序列化

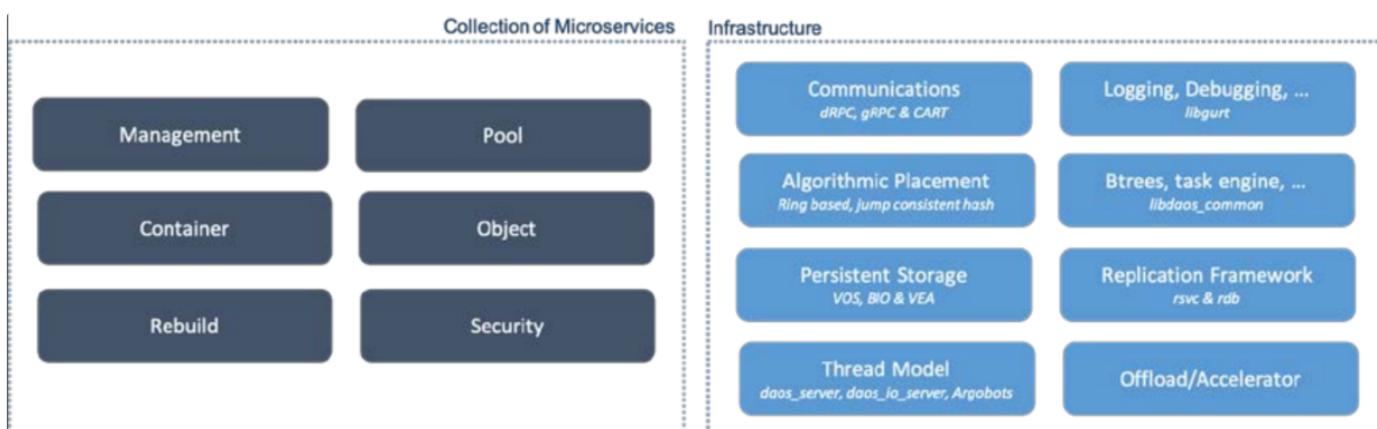
2.2.3 CART

CART是一个用户空间函数交付库，它为DAOS数据平面提供低延迟的高带宽通信。它支持RDMA功能和可伸缩的集体操作。CART是在Mercury和libfabric建造的。CART库用于libdaos和daos_io_server实例之间的所有通信。

2.3 Daos 层级和服务

2.3.1 架构

如下图所示，DAOS堆栈被结构化为客户机/服务器架构上的存储服务的集合。DAOS服务的示例是池、容器、对象和重建服务



DAOS服务可以分布在控制平面和数据平面上，并通过dRPC进行内部通信。大多数服务都有可以通过gRPC或CART进行同步的客户端和服务器组件。跨服务通信总是通过直接的API调用来完成的。这些函数调用可以通过服务的客户端或服务器组件进行调用。虽然每个DAOS服务都被设计为是相当自治和孤立的，但有些服务比其他服务耦合得更紧密。这通常是重建服务的情况，它需要与池、容器和对象服务紧密交互，以在DAOS服务器发生故障后恢复数据冗余。

基于服务的体系结构提供了灵活性和可扩展性，它还可以与一组基础库相结合使用，这些库提供了丰富的软件生态系统（例如，通信、持久存储访问、具有依赖图的异步任务执行、加速器支持等）。

2.3.2 源码框架

每个基础库和服务都在src/下分配了一个专用目录。服务的客户机组件和服务器组件分别存储在单独的文件中。

部分客户端组件带有前缀为dc_(代表DAOS客户端)，而服务器端功能使用ds_前缀(代表DAOS Server)。客户机和服务器组件之间使用的协议和RPC格式通常在一个名为rpc.h的头文件中定义。

在控制平面的上下文中执行的所有Go代码都位于src/控制下。管理和安全是分布在控制(Go语言)和数据(C语言)平面上的服务，并通过dRPC进行内部通信。为向最终用户(即I/O中间件或应用程序开发人员)公开的官方DAOS API的头在src/包含下，并使用daos_前缀。每个基础架构库导出一个在src/cnue/daos下可用的API，可以被任何服务使用。由给定服务导出的客户端API(带dc_前缀)也存储在src/包含/daos下，而服务器端接口(带ds_前缀)存储在src/包含/daos_srv下。

2.3.3 Daos基础库

GURT和 libdaos_common 库为DAOS服务提供了日志记录、调试和公共数据结构(例如，哈希表、btree、……)。本地NVM存储由版本控制对象存储(VOS)和blob I/O(BIO)库进行管理。VOS在SCM中实现了持久性索引，而BIO则负责在NVMe SSD或SCM中存储应用程序数据。VEA层被集成到VOS中，并管理NVMe SSD上的块分配。为了实现性能（即分片）和弹性（即复制或擦除代码），DAOS对象分布在多个目标上。放置库实现了不同的算法（例如，ring-based placement, jump consistent hash和其他算法），以从Target List和Object identifier中生成对象的布局。复制服务(RSVC)库最终提供了一些通用代码来支持容错。这个库由Pool、Container和Mge Svc与RDB库一起使用，该库实现了在Raft上复制的键值存储。

以便进一步阅读这些基础设施库

[Versioning Object Store \(VOS\)](#)

[Blob I/O \(BIO\)](#)

Algorithmic object placement

Replicated database (RDB)

Replicated service framework (RSVC)

2.3.4 Daos服务

垂直框表示DAOS服务，而水平框代表基础设施库。为了进一步了解每个服务的内部结构：

- Pool service
- Container service
- Key–array object service
- Self–healing (aka rebuild)
- Security

2.4.1 Protocol Compatibility(协议兼容性)

DAOS存储堆栈将提供执行版本**协议兼容性检查**，以验证：

- 同一 Pool 中的所有Target都运行相同的协议版本。
- 与应用程序链接的客户端库最多可能比目标版本早一个协议版本

如果在同一Pool中的存储Target之间检测到协议版本不匹配，则整个DAOS系统将无法启动，并将向控制API报告失败。类似地，来自运行与Target不兼容的协议版本的客户端的连接将返回一个错误

2.4.2 PM模式的兼容性和升级

持久性数据结构的模式可能会不时地发展，以修复错误、添加新的优化或支持新的特性。为此目的，持久性数据结构支持模式版本控制。

升级模式版本不会自动完成，必须由管理员启动。将提供一个专用的升级工具，将模式版本升级到最新版本。同一个池中的所有目标都必须具有相同的架构版本。在系统初始化时执行版本检查，以执行此约束。

为了限制验证矩阵，每个新的DAOS版本都将与受支持的模式版本列表一起发布。要使用新的DAOS版本运行，管理员将需要将DAOS系统升级到支持的模式版本之一。新目标将始终使用最新版本重新格式化。此版本控制模式仅适用于存储在持久性内存中的数据结构，而不适用于存储没有元数据的块存储。

3. Daos Control Plane

DAOS包括控制和数据。数据平面处理重起重运输操作，而控制平面协调过程和存储管理，便于数据平面的操作。

DAOS服务器实现了DAOS控制平面，并且是用Golang编写的。除了在同一台主机上运行的DAOS IO服务器(用C格式编写的数据平面)的实例化和管理外，它的任务还包括网络和存储硬件的配置和分配。DAOS的用户只能以DAOS服务器和相关工具的形式直接与控制平面进行交互。

DAOS Server实现了gRPC协议，以与客户端gRPC应用程序进行通信，并通过Unix域套接字与DAOS IO服务器进行交互。控制服务器加载了多个gRPC服务器模块。目前包含的模块是安全和管理。控制平面作为DAOS服务器的一部分实现了一个复制的管理服务，负责处理整个DAOS系统的分布式操作。shell是一个示例客户端应用程序，它既可以连接到代理以执行安全功能（如提供凭据和检索安全上下文），也可以连接到本地管理服务器以执行管理功能（如存储设备发现）

3.1 Doc

- [Management API](#)
- [Management Internals](#)
- [Agent API](#)
- [Agent internals](#)
- [dRPC](#)

3.2 Configuration

当启动daos_server进程时，daos_server配置文件会被解析；它的位置可以在命令行(-o选项)或默认位置(<daos安装dir>/install/etc/daos_server.yml)上指定。

示例配置文件可以在示例文件夹中找到。

如果在配置文件中不存在，则将使用默认的daos服务器配置中记录的默认值来解析和填充一些参数。

在命令行上作为应用程序选项（不包括环境变量）传递给daos_server的参数优先于配置文件中指定的值。

为了方便起见，活动解析的配置值被写入服务器配置文件被读取的目录，或/tmp/如果失败。

如果执行daos_server的用户shell设置了环境变量CRT_PHY_ADDR_STR，则在生成daos_io_server实例时将使用用户os环境。在这种情况下，将打印一个以“使用os env vars...”开头的错误消息，并且不会添

加在服务器的每个服务器部分的env_vars列表中指定的环境变量。此行为与通过环境变量指定所有参数的历史机制提供了向后兼容性。

强烈建议在服务器配置文件中指定运行DAOS服务器的所有参数和环境。

澄清有关影响 `daos_io_server` 实例行为的环境变量：

- 如果触发器环境变量是在用户的shell中设置的，则控制平面将使用在shell中设置的环境变量。该配置文件将被忽略。
- 如果没有在用户的shell中设置了触发器环境变量，则shell环境变量将被配置文件中设置的参数所覆盖

3.3 Subcommands

3.4 Shell Usage

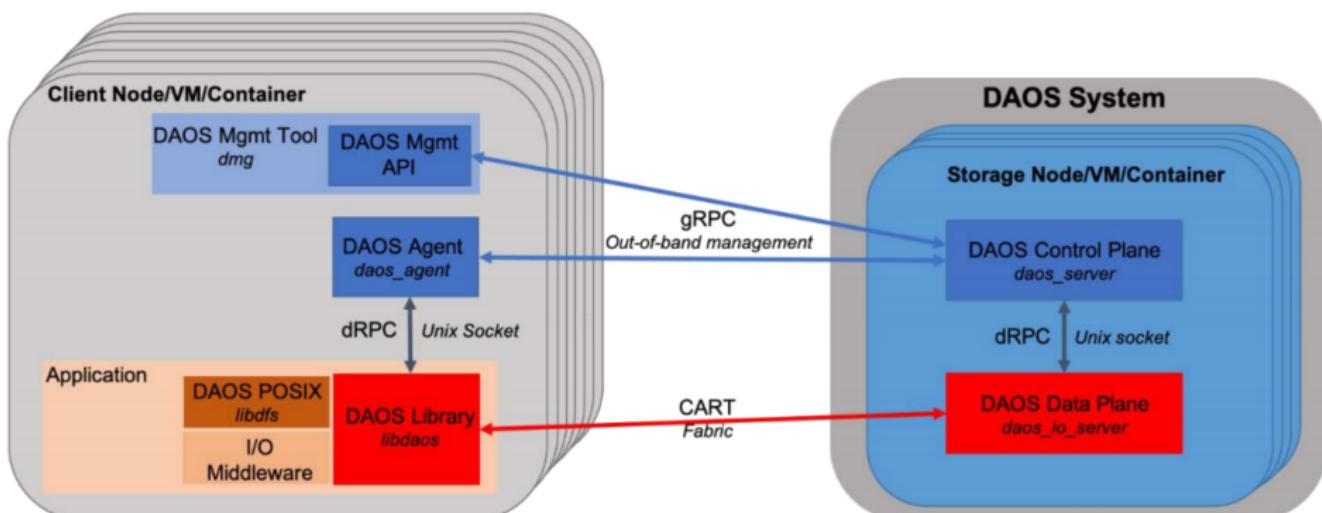
3.5 NVMe 管理

在NVMe SSD设备上的操作使用go-spdk绑定执行，通过SPDK框架发出命令。

3.6 SCM 管理

3.7 架构

软件组件的体系结构：



4. Daos Data Plane

DAOS 通过两个平面（DataPlane & ControlPlane）进行交互工作。DataPlane 处理大量的数据传输任务，而 ControlPlane 负责进程编排和存储管理，简化 DataPlane 的操作。

模块接口

I/O 引擎支持一个模块接口，该接口允许按需加载服务器端代码。每个模块实际上都是一个库，由 I/O 引擎通过 `dlopen` 动态加载。模块和 I/O 引擎之间的接口在 `dss_module` 数据结构中定义。

```
1 struct dss_module vos_srv_module = {
2     .sm_name      = "vos_srv",           // 模块名
3     .sm_mod_id   = DAOS_VOS_MODULE,    // 模块标识符
4     .sm_ver       = 1,                 // 特征位掩码
5     .sm_init      = vos_mod_init,      // 模块初始化函数,
6     .sm_fini      = vos_mod_fini,      // 销毁函数
7     .sm_key       = &vos_module_key,
8 };
```

每个模块应指定：

- 模块名
- `daos_module_id` 中的模块标识符
- 特征位掩码
- 一个模块初始化和销毁函数

此外，模块还可以选择配置：

- 在整个堆栈启动并运行后调用的配置和清理函数
- CART RPC 处理程序
- dRPC 处理程序

线程模型与 Argobot 集成

I/O 引擎是一个多线程进程，使用 [Argobot](#) 进行非阻塞处理。

默认情况下，每个 Target 都会创建一个 main xstream 和 no offload xstreams。offload xstream 的实际数量可以通过 `daos_engine` 命令行参数进行配置。此外，还创建了一个额外的 xstream 来处理传入的元数据请求。每个 xstream 都绑定到一个特定的 CPU 核心。main xstream 接收来自客户端和其他服务

器的 Target 传入请求。一个特定的 **ULT** (User Level Thread) 会在网络和 NVMe I/O 操作方面提供帮助。

Thread-local Storage (TLS)

每个 xstream 分配的私有存储可以通过 `dss_tls_get()` 函数进行访问。

```
1 static inline struct dss_thread_local_storage * dss_tls_get()
2 {
3     return (struct dss_thread_local_storage *)
4         pthread_getspecific(dss_tls_key);
5 }
```

注册时，每个模块可以指定一个模块密钥，该密钥的数据结构大小将由 TLS 中的每个 xstream 进行分配。

`dss_module_key_get()` 函数的作用是：返回特定注册模块密钥的数据结构。

```
1 /**
2  * Get value from context by the key
3  *
4  * Get value inside dtls by key. So each module will use this API to
5  * retrieve their own value in the thread context.
6  *
7  * \param[in] dtls  the thread context.
8  * \param[in] key    key used to retrieve the dtls_value.
9  *
10 * \retval      the dtls_value retrieved by key.
11 */
12 static inline void * dss_module_key_get(struct dss_thread_local_storage *d
13                                         tls,
14                                         struct dss_module_key *key)
15 {
16     D_ASSERT(key->dmk_index >= 0);
17     D_ASSERT(key->dmk_index < DAOS_MODULE_KEYS_NR);
18     D_ASSERT(dss_module_keys[key->dmk_index] == key);
19     D_ASSERT(dtls != NULL);
20
21     return dtls->dtls_values[key->dmk_index];
22 }
```

Incast Variable 集成

DAOS 使用 IV (incast variable) 在单个 IV 命名空间（组织结构为树）下的服务器之间共享值和状态。树的根节点称为 IV leader，服务器可以是叶子节点也可以是非叶子节点。

每个服务器都维护自己的 IV 缓存。在获取过程中，如果本地缓存不能完成请求，它会将请求转发给其父缓存，直到到达根缓存 (IV leader)。对于更新操作，服务器首先更新它的本地缓存，然后转发到它的父缓存，直到到达根缓存，然后将更改传播到其他的服务器。

IV 命名空间是属于每个 Pool 的，在 Pool 连接期间创建，在 Pool 断开连接期间销毁。

要使用 IV，每个用户需要在 IV 命名空间下注册自己以获得标识符，然后用户将使用这个 ID 来获取或更新自己在 IV 命名空间下的 IV 值。

dRPC 服务器

I/O 引擎包括一个 dRPC 服务器，它监听给定 Unix Domain Socket 上的活动。

有关 dRPC 的基础知识以及 Go 和 C 中的底层 API 的更多详细信息，请参阅 [dRPC Documentation](#)。

dRPC 服务器定期轮询传入的客户端连接和请求。它可以通过 `struct drpc_progress_context` 对象同时处理多个客户端连接，该对象管理监听 Socket 的 `struct drpc` 对象以及任何活动的客户端连接。

```

1  struct drpc_progress_context {
2      struct drpc *listener_ctx; /* Just a pointer, not a copy */
3      d_list_t    session_ctx_list; /* Head of the session list */
4  };
5
6  /*
7   * Stands up a drpc listener socket and creates a corresponding progress
8   * context.
9   */
10 static int setup_listener_ctx(struct drpc_progress_context **new_ctx)
11 {
12     struct drpc *listener;
13     char        *sockpath = drpc_listener_socket_path;
14
15     /* If there's something already in the socket path, it will fail */
16     unlink(sockpath);
17     listener = drpc_listen(sockpath, drpc_hdlr_process_msg);
18     if (listener == NULL) {
19         D_ERROR("Failed to create listener socket at '%s'\n",
20                sockpath);
21         return -DER_MISC;
22     }
23
24     *new_ctx = drpc_progress_context_create(listener);
25     if (*new_ctx == NULL) {
26         D_ERROR("Failed to create drpc_progress_context\n");
27         drpc_close(listener);
28         return -DER_NOMEM;
29     }
30
31     return 0;
32 }
33
34 /*
35  * Sets up the listener socket and kicks off a ULT to listen on it.
36  */
37 static int drpc_listener_start_ult(ABT_thread *thread)
38 {
39     int             rc;
40     struct drpc_progress_context *ctx = NULL;
41
42     rc = setup_listener_ctx(&ctx);
43     if (rc != 0) {
44         D_ERROR("Listener setup failed, aborting ULT creation\n");
45         return rc;

```

```

46     }
47
48     /* Create a ULT to start the drpc listener */
49     rc = dss_ult_create_with_name(drpc_listener_run, (void *)ctx, DSS_XS_D
50     RPC,
51     0, 0, thread, "drpc_listener_run");
52     if (rc != 0) {
53         D_ERROR("Failed to create drpc listener ULT: "DF_RC"\n",
54             DP_RC(rc));
55         drpc_progress_context_close(ctx);
56         return rc;
57     }
58
59     return 0;
60 }
61 int drpc_listener_init(void)
62 {
63     int rc;
64
65     rc = generate_socket_path();
66     if (rc != 0)
67         return rc;
68
69     memset(&status, 0, sizeof(status));
70     rc = ABT_mutex_create(&status.running_mutex);
71     if (rc != ABT_SUCCESS) {
72         D_ERROR("Failed to create mutex\n");
73         return dss_abterr2der(rc);
74     }
75
76     return drpc_listener_start_ult(&status.thread);
77 }

```

服务器在 xstream 0 自己的 **ULT** (User Level Thread) 中循环运行。dRPC Socket 已设置为非阻塞的，并且使用无超时轮询。这允许服务器在 ULT 中运行，而不是在自己的 xstream 中运行，预计该通道的流量相对较低。

dRPC 进程

`drpc_progress` 表示 dRPC 服务器循环的一次迭代。其工作流程如下：

1. 在 `listening socket` 和任何打开的 `client connection` 上同时进行超时轮询。

▼ The dRPC listener thread

C | 复制代码

```
1 static void drpc_listener_run(void *arg)
2 {
3     struct drpc_progress_context *ctx;
4
5     D_ASSERT(arg != NULL);
6     ctx = (struct drpc_progress_context *)arg;
7
8     D_INFO("Starting dRPC listener\n");
9     set_listener_running(true);
10    while (is_listener_running()) {
11        int rc;
12
13        /* wait a second */
14        rc = drpc_progress(ctx, 1000);
15        if (rc != DER_SUCCESS && rc != -DER_TIMEDOUT) {
16            D_ERROR("dRPC listener progress error: %s\n",
17                   DP_RC(rc));
18        }
19
20        ABT_thread_yield();
21    }
22
23    D_INFO("Closing down dRPC listener\n");
24    drpc_progress_context_close(ctx);
25 }
```

2. 如果在 `client connection` 上发现任何活动:

a. 如果数据已输入: 调用 `drpc_recv_call` 处理输入的数据。

C | 复制代码

```
1 int drpc_progress(struct drpc_progress_context *ctx, int timeout_ms)
2 {
3     size_t num_comms;
4     struct unixcomm_poll *comms;
5     int rc;
6     ...
7
8     rc = unixcomm_poll(comms, num_comms, timeout_ms);
9     if (rc > 0) {
10         //活动发现后的处理函数入口 `process_activity`
11         rc = process_activity(ctx, comms, num_comms);
12     }
13     return rc;
14 }
15
16 static int handle_incoming_call(struct drpc *session_ctx)
17 {
18     int rc;
19     Drpc__Call *call = NULL;
20     Drpc__Response *resp = NULL;
21     struct drpc_call_ctx *call_ctx;
22
23     //调用 drpc_recv_call 处理输入的数据
24     rc = drpc_recv_call(session_ctx, &call);
25
26     /* Need to respond even if it was a bad call */
27     if (rc != 0 && rc != -DER_PROTO)
28         return rc;
29     ...
30
31     return 0;
32 }
```

- b. 如果客户端已断开连接或连接被破坏：释放 `struct drpc` 对象并将其从 `drpc_progress_context` 中删除。

C | 复制代码

```
1 static int process_session_activity(struct drpc_list *session_node,
2                                     struct unixcomm_poll *session_comm)
3 {
4     int rc = 0;
5
6     D_ASSERT(session_comm->comm->fd == session_node->ctx->comm->fd);
7
8     switch (session_comm->activity) {
9         case UNIXCOMM_ACTIVITY_DATA_IN:
10            rc = handle_incoming_call(session_node->ctx);
11            if (rc != 0 && rc != -DER_AGAIN) {
12                D_ERROR("Error processing incoming session %u data\n",
13                       session_comm->comm->fd);
14                destroy_session_node(session_node);
15
16                /* No further action needed */
17                rc = 0;
18            }
19            break;
20
21         case UNIXCOMM_ACTIVITY_ERROR:
22         case UNIXCOMM_ACTIVITY_PEER_DISCONNECTED:
23             D_INFO("Session %u connection has been terminated\n",
24                   session_comm->comm->fd);
25
26             //释放 struct drpc 对象并将其从 drpc_progress_context 中删除。
27             destroy_session_node(session_node);
28             break;
29
30         default:
31             break;
32     }
33
34     return rc;
35 }
36
37 static void destroy_session_node(struct drpc_list *session_node)
38 {
39     drpc_close(session_node->ctx);
40     d_list_del(&session_node->link);
41     D_FREE(session_node);
42 }
```

3. 如果在 listening socket 上发现任何活动：

- a. 如果有新的连接进入：调用 `drpc_accept` 并将新的 `struct drpc` 对象添加到 `drpc_progress_context` 中的客户端连接列表中。

```
1  /**
2   * Wait for a client to connect to a listening drpc context, and return the
3   * context for the client's session.
4   *
5   * \param    ctx drpc context created by drpc_listen()
6   *
7   * \return   new drpc context for the accepted client session, or
8   *           NULL if failed to get one
9   */
10  struct drpc * drpc_accept(struct drpc *listener_ctx)
11  {
12      struct drpc *session_ctx;
13      struct unixcomm *comm;
14
15      if (!drpc_is_valid_listener(listener_ctx)) {
16          D_ERROR("dRPC context is not a listener\n");
17          return NULL;
18      }
19
20      D_ALLOC_PTR(session_ctx);
21      if (session_ctx == NULL)
22          return NULL;
23
24      comm = unixcomm_accept(listener_ctx->comm);
25      if (comm == NULL) {
26          D_FREE(session_ctx);
27          return NULL;
28      }
29
30      init_drpc_ctx(session_ctx, comm, listener_ctx->handler);
31      return session_ctx;
32  }
33
34  static int drpc_progress_context_accept(struct drpc_progress_context *ctx)
35  {
36      struct drpc     *session;
37      struct drpc_list *session_node;
38
39      session = drpc_accept(ctx->listener_ctx);
40      if (session == NULL) {
41          /*
42             * Any failure to accept is weird and surprising
43             */
44          D_ERROR("Failed to accept new drpc connection\n");
45      }
46  }
```

```
45         return -DER_MISC;
46     }
47
48     D_ALLOC_PTR(session_node);
49     if (session_node == NULL) {
50         D_FREE(session);
51         return -DER_NOMEM;
52     }
53
54     session_node->ctx = session;
55     d_list_add(&session_node->link, &ctx->session_ctx_list);
56
57     return DER_SUCCESS;
58 }
```

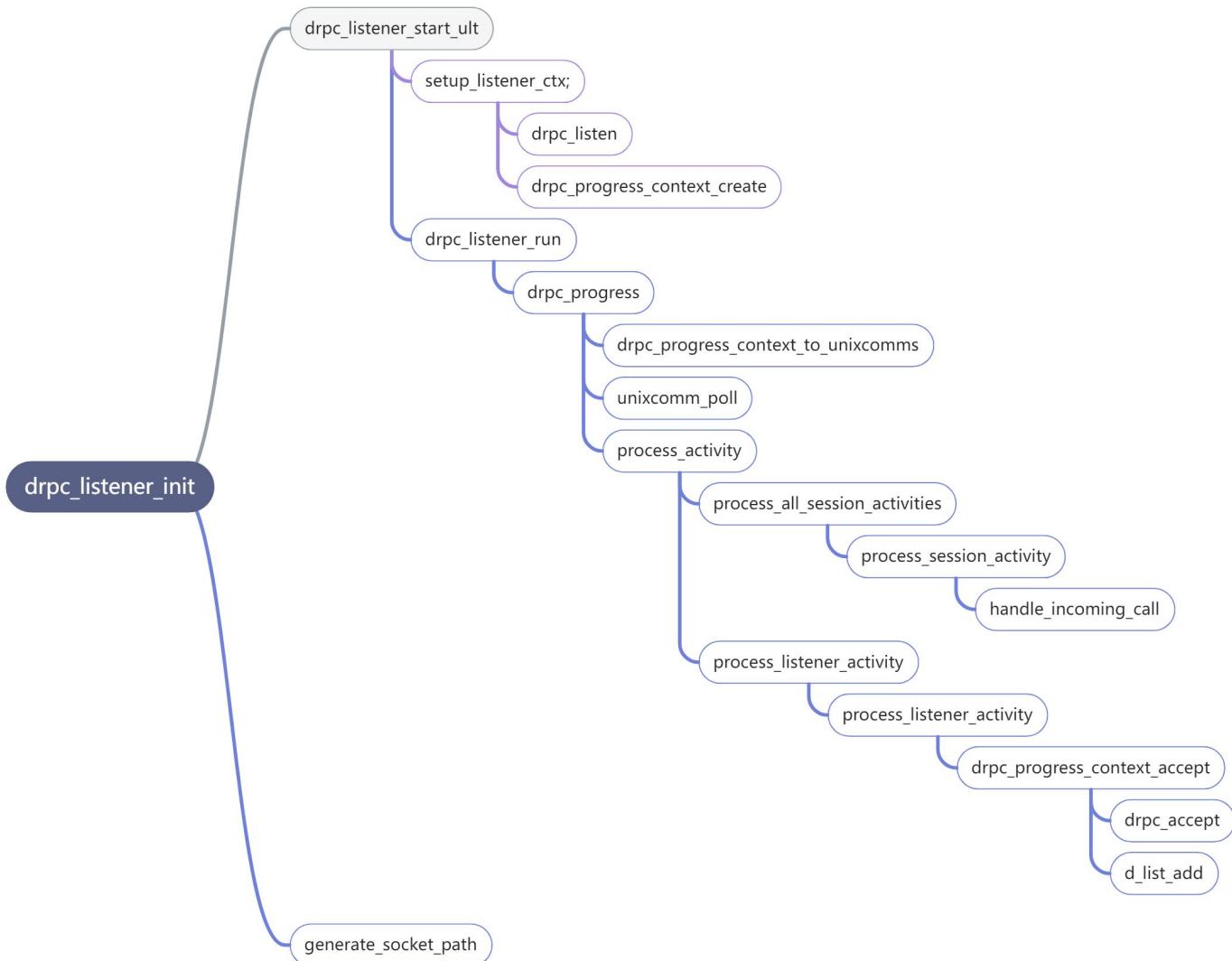
- b. 如果有连接错误：将 `-DER_MISC` 返回给调用者。I/O 引擎中会记录该错误，但不会中断 dRPC 服务器循环。在listening socket上获取到错误是意外情况。

C | 复制代码

```
1 static int
2 process_listener_activity(struct drpc_progress_context *ctx,
3     struct unixcomm_poll *comms, size_t num_comms)
4 {
5     int rc = 0;
6     size_t last_idx = num_comms - 1;
7     struct unixcomm_poll *listener_comm = &(comms[last_idx]);
8     /* Last comm is the listener */
9
10    D_ASSERT(listener_comm->comm->fd == ctx->listener_ctx->comm->fd);
11
12    switch (listener_comm->activity) {
13        case UNIXCOMM_ACTIVITY_DATA_IN:
14            rc = drpc_progress_context_accept(ctx);
15            break;
16
17        case UNIXCOMM_ACTIVITY_ERROR:
18        case UNIXCOMM_ACTIVITY_PEER_DISCONNECTED:
19            /* Unexpected – don't do anything */
20            D_INFO("Ignoring surprising listener activity: %u\n",
21                  listener_comm->activity);
22            rc = -DER_MISC;
23            break;
24
25        default:
26            break;
27    }
28
29    return rc;
30 }
```

4. 如果没有检测到任何活动，则将 `-DER_TIMEDOUT` 返回给调用者。这仅仅为了调试，实际上，I/O 引擎会忽略此错误代码，因为缺少活动实际上并不是一种错误。

dRPC 进程函数调用关系如下：



dRPC 处理程序注册

单个 DAOS 模块可以通过注册一个或多个 dRPC 模块 ID 的处理函数来实现对 dRPC 消息的处理。

注册处理程序很简单。在 `dss_server_module` 的字段 `sm_drpc_handlers` 中，静态分配一个 `struct dss_drpc_handler` 数组，该数组的最后一项为零，以指示列表的结尾。将字段设置为 NULL 表示没有要注册的处理程序。当 I/O 引擎加载 DAOS 模块时，它将自动注册所有 dRPC 处理程序。

C | 复制代码

```
1  /**任何通过Unix域套接字接受dRPC通信的dss_module必须提供一个或多个dRPC处理程序函数。  
I/O引擎使用该处理程序多路复用传入的dRPC消息以进行处理。  
2  dRPC消息传递模块ID与dss_module的ID不同。dss_ module可以处理多个dRPC模块I  
3  */  
4  struct dss_drpc_handler {  
5      int      module_id;    /* dRPC messaging module ID */  
6      drpc_handler_t  handler;    /* dRPC handler for the module */  
7  };  
8  
9  struct dss_module {  
10     /* Name of the module */  
11     const char          *sm_name;  
12     ...  
13     /* dRPC handlers, for unix socket comm, last entry must be empty */  
14     struct dss_drpc_handler *sm_drpc_handlers;  
15     ...  
16 };  
17  
18  
19 //drpc注册处理程序  
20 int drpc_hdlr_register_all(struct dss_drpc_handler *handlers)  
21 {  
22     int      rc = DER_SUCCESS;  
23     struct dss_drpc_handler *current;  
24  
25     if (registry_table == NULL) {  
26         D_ERROR("Table not initialized\n");  
27         return -DER_UNINIT;  
28     }  
29  
30     if (handlers == NULL) {  
31         /* Nothing to do */  
32         return DER_SUCCESS;  
33     }  
34  
35     /* register as many as we can */  
36     current = handlers;  
37     while (current->handler != NULL) {  
38         int handler_rc;  
39  
40         handler_rc = drpc_hdlr_register(current->module_id,  
41                                         current->handler);  
42         if (handler_rc != DER_SUCCESS) {  
43             rc = handler_rc;  
44         }  
45     }  
46 }
```

```

45         current++;
46     }
47
48     return rc;
49 }
50
51
52 int drpc_hdlr_register(int module_id, drpc_handler_t handler)
53 {
54     if (registry_table == NULL) {
55         D_ERROR("Table not initialized\n");
56         return -DER_UNINIT;
57     }
58
59     if (!module_id_is_valid(module_id)) {
60         D_ERROR("Module ID %d out of range\n", module_id);
61         return -DER_INVAL;
62     }
63
64     if (handler == NULL) {
65         D_ERROR("Tried to register a null handler\n");
66         return -DER_INVAL;
67     }
68
69     if (registry_table[module_id] != NULL) {
70         D_ERROR("Tried to register module ID %d more than once\n",
71                 module_id);
72         return -DER_EXIST;
73     }
74
75     registry_table[module_id] = handler;
76
77     return DER_SUCCESS;
78 }

```

注意：

- dRPC 模块 ID 与 DAOS 模块 ID 不同。
- 这是因为给定的 DAOS 模块可能需要注册多个 dRPC 模块 ID，具体数量取决于 DAOS 模块所涵盖的功能。
- dRPC 模块 ID 必须是系统范围内唯一的，并且列在一个中心头文件 `src/include/daos/drpc_modules.h` 中。

dRPC 服务器使用函数 `drpc_hdlr_process_msg` 来处理传入的消息。此函数检查传入消息的模块 ID，搜索处理程序。

C | 复制代码

```
1  /*
2   * Top-level handler for incoming dRPC messages. Looks up the appropriate
3   * registered dRPC handler and runs it on the message.
4   */
5  void drpc_hdlr_process_msg(Drpc__Call *request, Drpc__Response *resp)
6  {
7      drpc_handler_t handler;
8
9      D_ASSERT(request != NULL);
10     D_ASSERT(resp != NULL);
11
12     handler = drpc_hdlr_get_handler(request->module);
13     if (handler == NULL) {
14         D_ERROR("Message for unregistered dRPC module: %d\n",
15                request->module);
16         resp->status = DRPC__STATUS__UNKNOWN_MODULE;
17         return;
18     }
19
20     handler(request, resp);
21 }
```

- 如果找到处理程序，则执行该处理程序，并返回 `Drpc_Response`。
- 如果找不到，它将生成自己的 `Drpc_Response`，指示模块 ID 未注册。

5. Daos 数据备份

Pool、Container和管理服务可以通过复制它们的内部元数据来获得高度的可用性。在这种通用方法中复制的服务可以容忍其任何少数副本的失败。因此，通过在故障域中扩展每个服务的复制副本，池和容器服务可以容忍合理数量的目标故障。

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。

```

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 nt);

```

```

1  /* rsvc_client在处理leadership hint 未命中情况下。*/
2  static void
3  rsvc_client_process_error(struct rsvc_client *client, int rc,
4                  const crt_endpoint_t *ep)
5
6  // rsvc_client在处理leadership hint 命中情况下。
7  static void
8  rsvc_client_process_hint(struct rsvc_client *client,
9                  const struct rsvc_hint *hint, bool from_leader,
10                 const crt_endpoint_t *ep)

```

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。

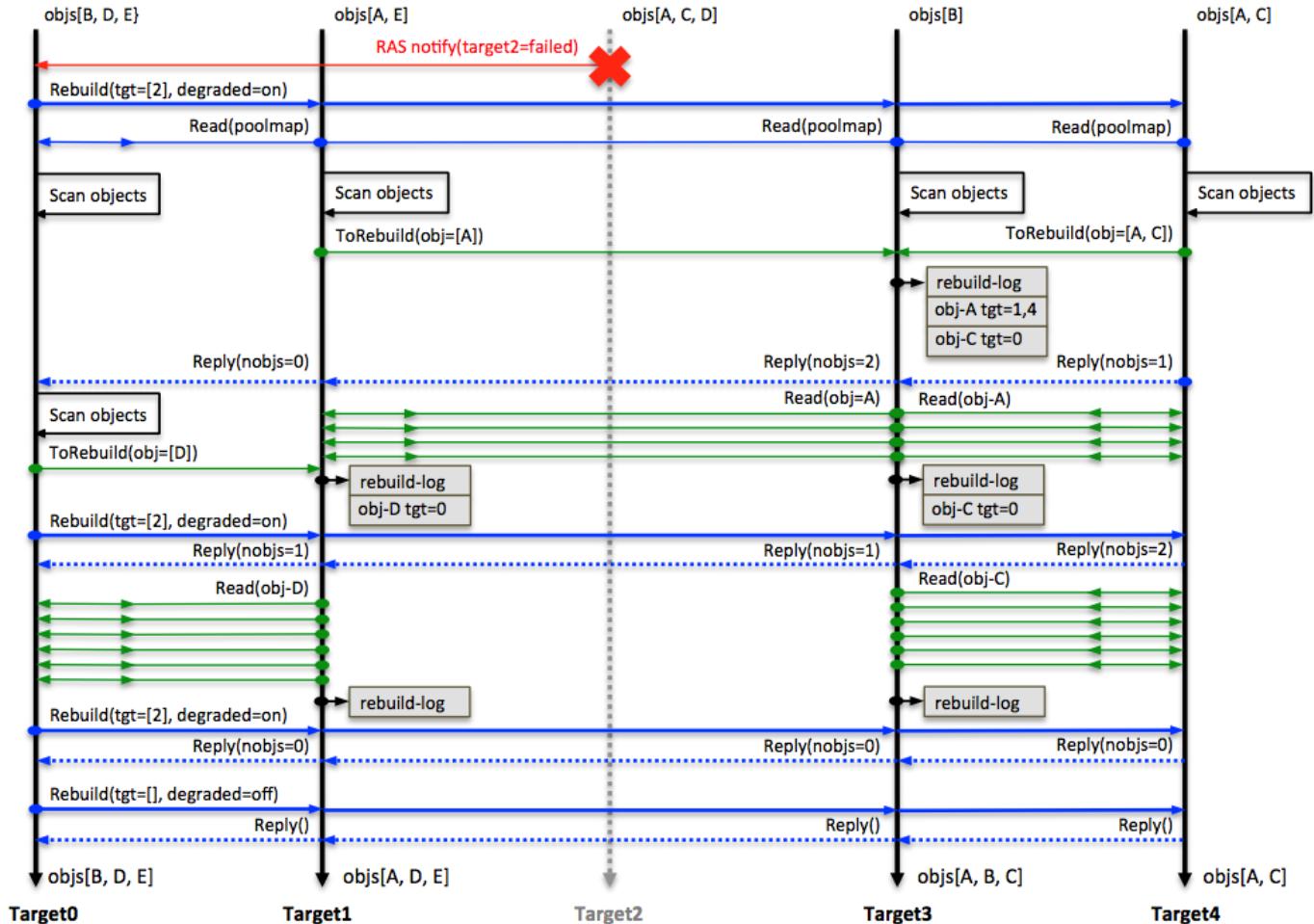
Rebuild Detection

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。



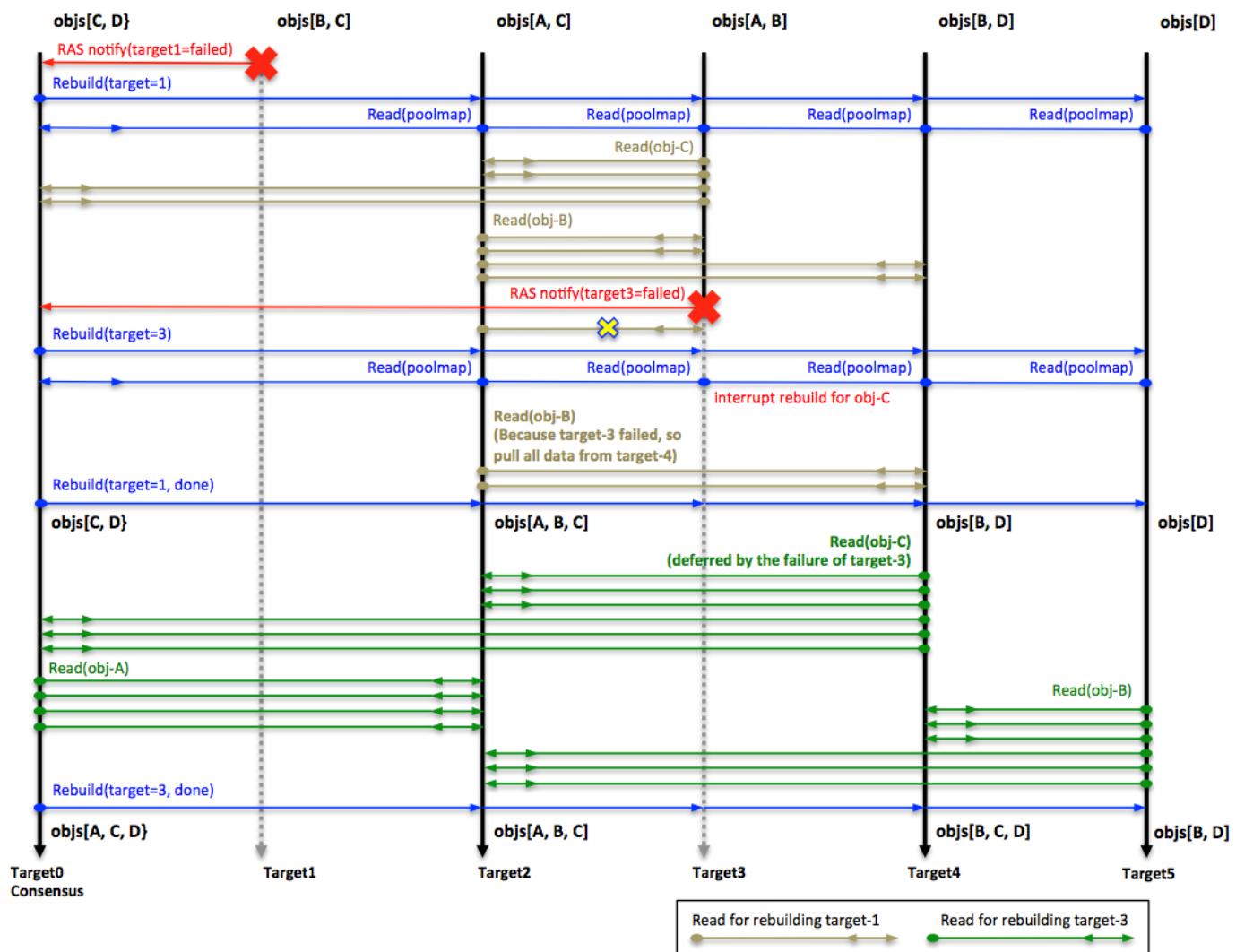
上图是此过程的一个示例：群集中有五个对象：对象 A 是 3 向复制的，对象 B、C、D 和 E 是双向复制的。当目标 2 失败时，目标 0（即 Raft 领导者）向所有幸存的目标广播失败，以通知它们进入降级模式并进行扫描：

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

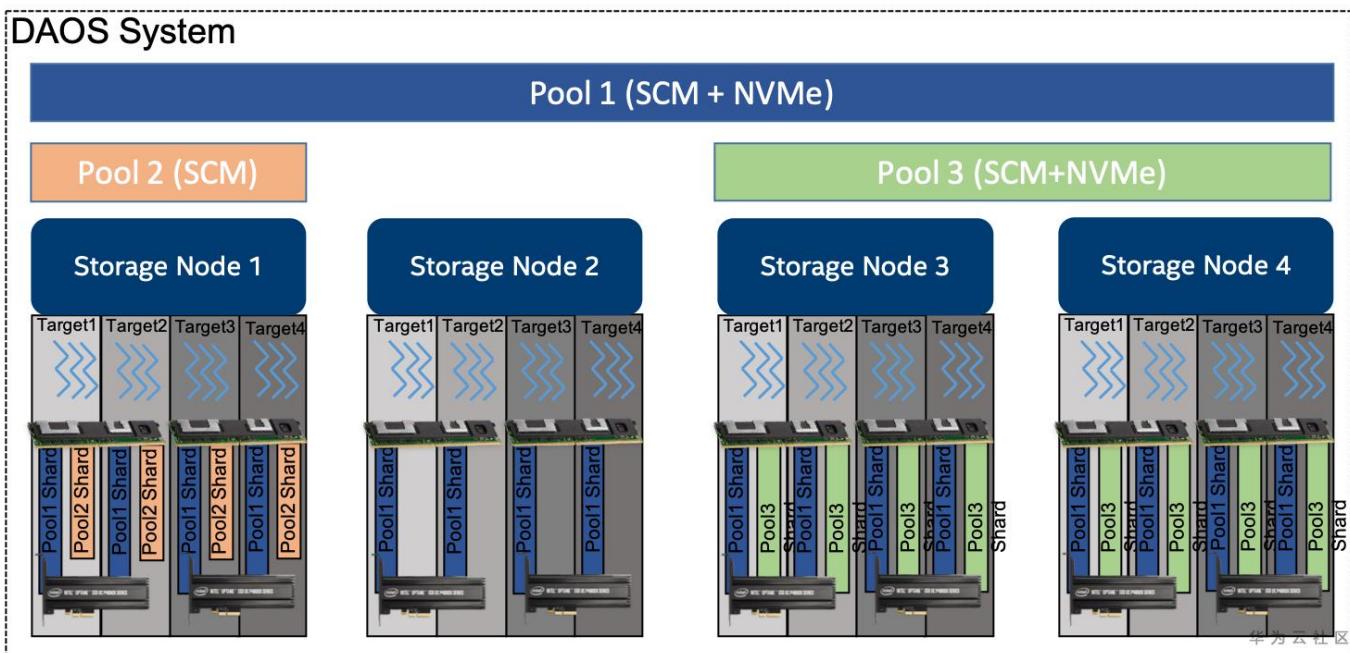
- `dc_obj_fetch_task_create`：将 `csum iov` 设置为 `daos_obj_fetch_t args`。这些参数设置为 `rw_cb_args.shard_args.api_args`，并通过 `cli_shard.c` 中的访问器函数 (`rw_args2csum iov`) 进行访问，以便 `rw_args_store_csum` 可以轻松访问它。从 `dc_rw_cb_csum_verify` 调用的此函数将从服务器接收的数据校验和打包到 `iov` 中。
- `dc_obj_list_obj_task_create`：将 `csum iov` 设置为 `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。

7. Daos Pool



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。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。

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。

C | 复制代码

```
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。

C | 复制代码

```
1 // Storage pool
2
3 | typedef struct { ... } daos_pool_info_t;
29
30 | struct ds_pool { ... };
```

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。

Pool Service

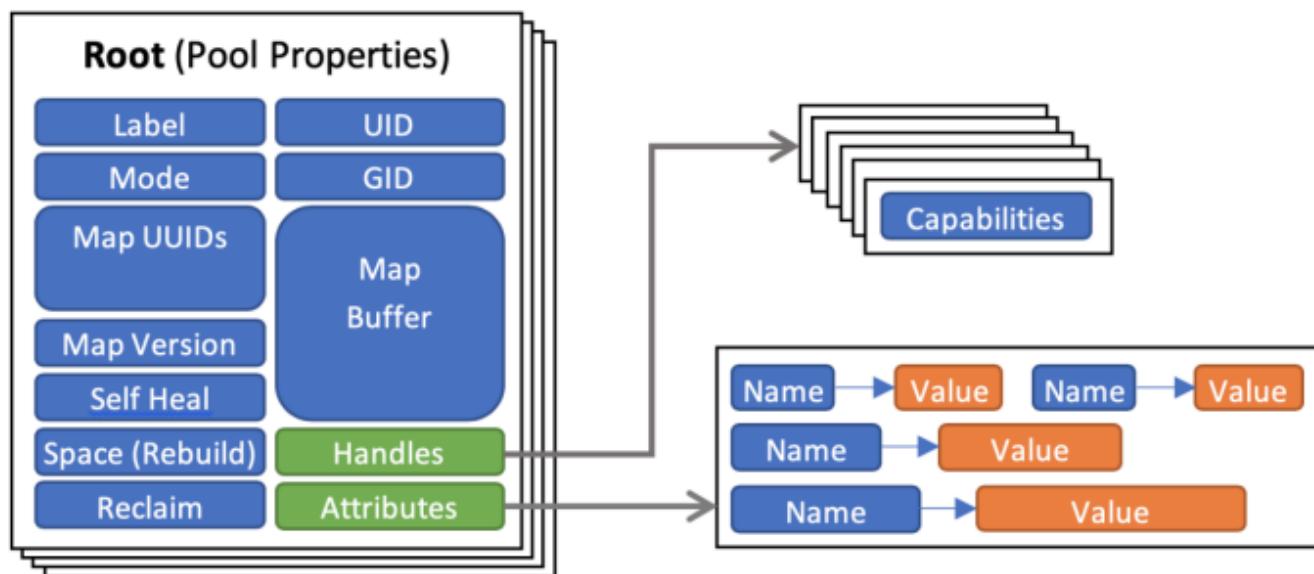
pool 服务 (Pool_svc) 存储pool的元数据，并提供用于查询和更新pool 配置的API。

```

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 }
```

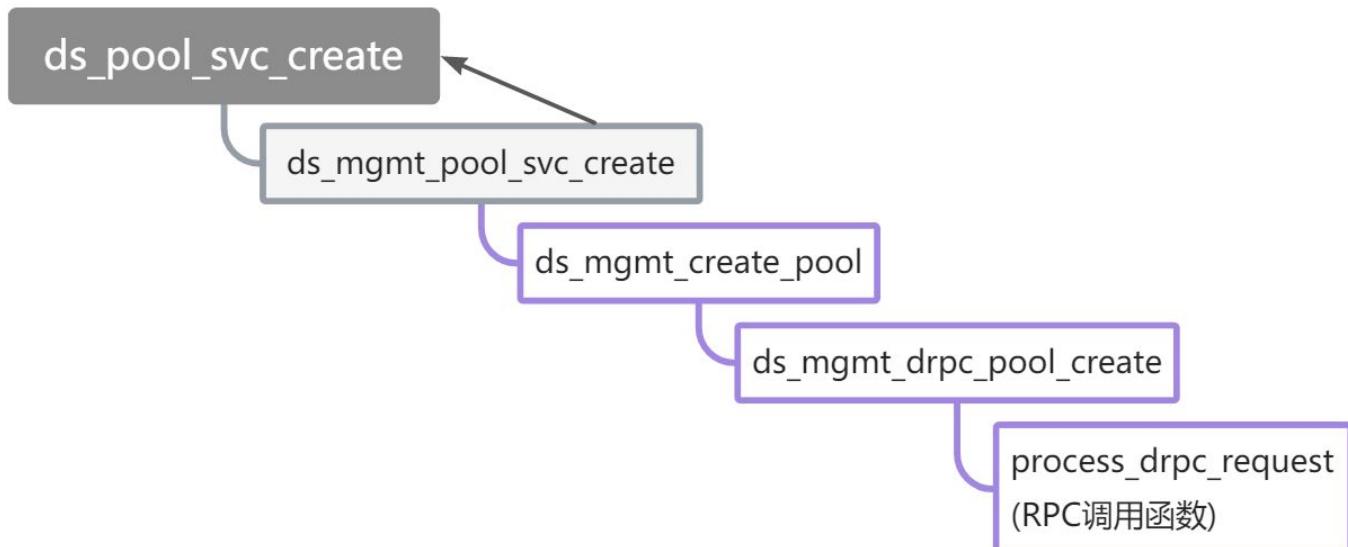
此时，`management module` 通过调用 `DS_pool_svc_create` 将控制权传递给pool模块，这将在combined Pool 和 Container Service 的选定节点子集上初始化服务复制。pool 模块现在向创建服务数据库的服务负责人发送pool 创建请求；然后，目标及其容错域的列表将转换为pool 映射的初始版本，并与其他初始pool 元数据一起存储在pool 服务中。

```

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
19
20
21 };

```

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。
- 如果存在另一个Pool handle，且当前请求的或现有的 `pool handle` 是需要独占访问权限，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 以便当前请求的接入。

注：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`。共享此连接的其他客户端进程应该在本地销毁其池句柄的副本，最好是在代表所有人调用断开连接方法之前。如果一组客户端进程在有机会调用pool disconnect方法之前过早终止，那么一旦pool svc从运行时环境了解到该事件，它们的池连接最终将被驱逐。

Pool 相关函数

Storage Target

Target 的类型有 4 种：

```
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;
```

C | 复制代码

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 };
```

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 };
```

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;
```

Pool Target

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

C | 复制代码

```
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。

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，要求的缓冲区大可能会被写入 `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    DAOS_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:

```
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
 $\text{[ ]},$ 
2           void const *const values[], size_t const sizes[],
3           daos_event_t *ev)
4 | { ... }
```

`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

```

1 int daos_pool_del_attr(daos_handle_t poh, int n, char const *const names
2 [],  

3     daos_event_t *ev)
4 {
5     daos_pool_del_attr_t *args;
6     tse_task_t *task;
7     int rc;
8
9     // 判断 *args 大小是否与 daos_pool_del_attr_t 的预期大小相等
10    DAOS_API_ARG_ASSERT(*args, POOL_DEL_ATTR);
11
12    // 创建新任务 dc_pool_del_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_del_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
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    // 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

```

1 typedef struct {
2     /** Pool open handle. */
3     daos_handle_t      poh;
4     /** [in] length of \a cont_buf. [out] num of containers in the pool. */
5     daos_size_t         *ncont;
6     /** Array of container structures. */
7     struct daos_pool_cont_info  *cont_buf;
8 } daos_pool_list_cont_t;
```

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：

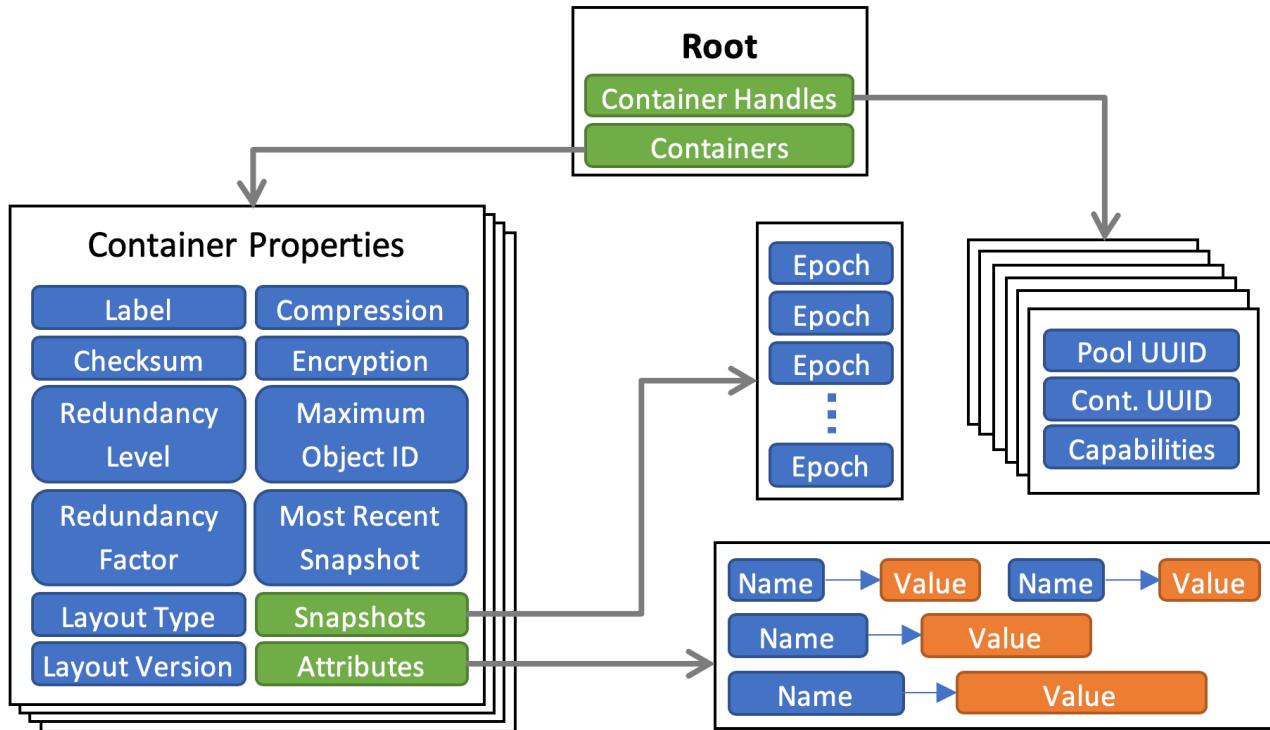
- 如果成功，返回 0。
- 如果失败，返回
 - -DER_INVAL: 无效的参数。
 - -DER_TRUNC: cbuf 没有足够的空间存储 ncont 个元素。

8. Daos Container

container 概念

Metadata Layout

Container 代表 Pool 中的对象地址空间，由 Container UUID 标识。Container Service (cont_svc) 存储 container 的元数据，并提供用于查询和更新状态以及管理 container 生命周期的 API。container 元数据被组织为键值存储 (KVS) 的层次结构，这些键值存储在多个服务器上进行复制，这些服务器由 Raft 共识协议支持；client 请求只能由 service leader 提供服务，而 non-leader 副本仅会以指向当前负责人的提示进行响应，以便 client 重试。cont_svc 源自通用复制服务模块 rsrv (see: [Replicated Services: Architecture](#))



KVS 根目录有两个子节点：

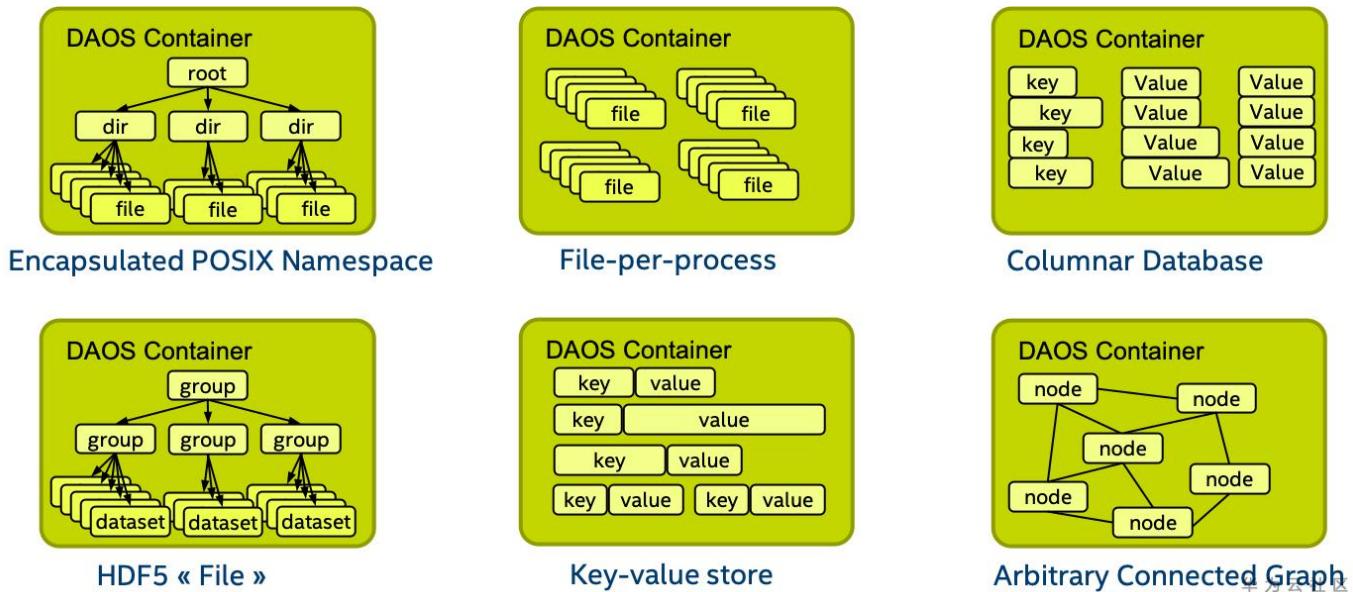
- **Container KVS**：//保存由用户在创建新container时提供的container UUID索引的container 属性列表。
- **Container Handles KVS**：用于存储有关由各种应用程序打开的container handle的数据，并由client 在打开container时生成的handle UUID索引。与container handle关联的元数据包括其功能（例如，只读或读写）及其每个handle的状态。当container关闭时，相应的条目将从此存储中删除。

container属性KVS用于存储每个container的元数据，这些元数据由许多可变和不可变的标量值属性以及上图所示的其他KVS组成。

用户可以创建、删除和检索持久快照列表，这些快照本质上是不会被聚合掉的时代。快照在显式销毁之前保持可读性。还可以将container回滚到特定快照 (see: [Storage Model: DAOS Container](#) and [Transaction Model: Container Snapshot](#))。

用户还可以为本质上是名称–值对的container定义自定义属性；名称为以null结尾的字符串，而值为任意字节序列。container服务允许client 一次检索和更新多个属性，以及列出存储属性的名称。

下图显示了用户 (I/O 中间件、特定领域的数据格式、大数据或 AI 框架等) 如何使用 Container 来存储相关数据集：

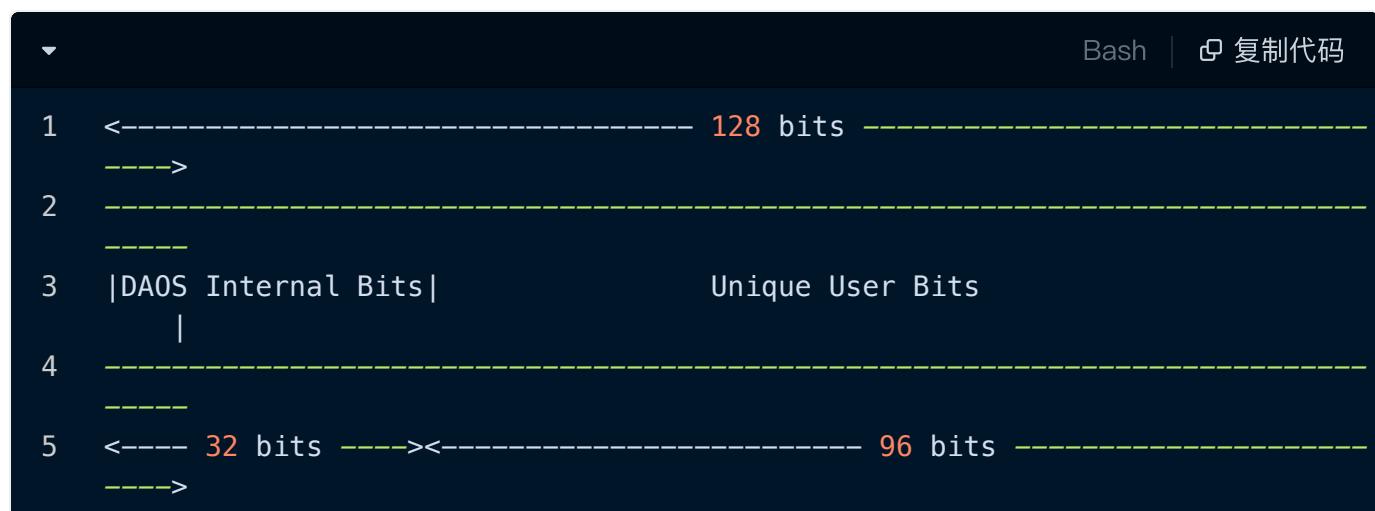


与 Pool一样，Container 可以存储用户属性。Container 在创建时必须传递一组属性，以配置不同的功能，例如校验和。

要访问 Container，应用程序必须首先连接到 Pool，然后打开 Container。如果应用程序被授权访问 Container，则返回 Container 句柄，它的功能包括授权应用程序中的任何进程访问 Container 及其内容。打开进程可以与所有对等进程共享此句柄。它们的功能在 Container 关闭时被撤销。

Container 中的对象可能具有不同的模式，用于处理数据分布和 Target 上的冗余，定义对象模式所需的一些参数包括动态或静态条带化、复制或纠删码。

如下所示，Container 中的每个对象都由一个唯一的 128 位对象地址标识。对象地址的高 32 位保留给 DAOS 来编码内部元数据，比如 Object 类。剩下的 96 位由用户管理，在 Container 中应该是唯一的。只要保证唯一性，栈的上层就可以使用这些位来编码它们的元数据。DAOS API 为每个 Container 提供了 64 位可伸缩对象 ID 分配器。应用程序要存储的对象 ID 是完整的 128 位地址，该地址仅供一次性使用，并且只能与单个对象模式相关联。



Container 是事务和版本控制的基本单元。所有的对象操作都被 DAOS 库隐式地标记为一个称为 epoch 的时间戳。DAOS 事务 API 允许组合多个对象更新到单个原子事务中，并基于 epoch 顺序进行多版本并发控制。所有版本更新都可以定期聚合，以回收重叠写入所占用的空间，并降低元数据复杂性。快照是一个永久引用，可以放置在特定的 epoch 上以防止聚合。

Container 元数据（快照列表、打开的句柄、对象类、用户属性、属性和其他）存储在持久性内存中，并由专用 Container 元数据服务维护，该服务使用与父元数据 Pool 服务相同的复制引擎或自己的引擎，这在创建 Container 时是可配置的。

与 Pool 一样，对 Container 的访问由 Container 句柄控制。要获取有效的句柄，应用程序进程必须打开 Container 并通过安全检查。然后，可以通过 Container 的 `local2global()` 和 `global2local()` 操作与其他对等应用程序进程共享此句柄。

Target Service

Target Service 将 DAOS container 的全局对象地址空间映射到 target VOS pool (vpool) 中 VOS container 的本地对象地址空间，并通过 Container Service 调用 VOS 方法 (see: [VOS Concepts](#))。它在 container 对象上缓存每个线程的信息，并在易失性内存中打开 handle 以便随时访问。

Object ID Allocator

OID 分配器是一个辅助例程服务，它允许用户在容器中分配一组唯一的 64 位无符号整数。这对于那些无法以可伸缩的方式轻松分配唯一的 DAOS 对象 ID 的应用程序或中间件很有帮助。在容器属性 KVS 中跟踪分配的最大 ID，以便将来访问该容器。此服务不能保证分配的 ID 是顺序的，并且可以在容器关闭时丢弃多个 ID 范围。

分配器使用服务器端的 `Incast` 变量实现，该变量跟踪 IV 树根目录上的容器上使用的最高对象 ID。客户端可以从运行容器目标服务的任何服务器（即 IV 中的任何节点）请求分配 ID。当新请求到达时，服务器首先检查本地是否有可用的 ID。如果不是，则将请求转发给父级（在这种情况下，请求更大范围的 OID）。父级执行相同的检查并继续转发到其父级，直到满足请求或我们到达 IV 根，这将更新最大值的 incast 变量在容器元数据中分配的 OID。在每个树级别，请求的 OID 数量将会增加，以便更快地满足未来 OID 分配请求。

代码 `src\container\srv_container.c`

Container Operations

client 客户端通过向带有Pool handle 和 UUID 的容器服务发送 `CONT_CREATE` 请求来创建一个新的容器

```
1 int dc_cont_create(tse_task_t *task)
2 {
3     ...
4     rc = cont_req_create(daos_task2ctx(task), &ep, CONT_CREATE, &rpc);
5     if (rc != 0) {
6         D_ERROR("failed to create rpc: %s\n", DP_RC(rc));
7         D_GOTO(err_prop, rc);
8     }
9     ...
10 }
11
```

client 必须首先建立pool 连接才能获得pool handle。或者，请求还可以包含要在新创建的 container上设置的属性列表。作为响应，container Service 使用UUID作为密钥创建相应的 Container Properties KVS。创建container不需要 Target Service的参与。

client 现在可以通过提供open pool handle和container UUID以及参数（例如，只读或读写）来打开container。client 库向Container Service发送带有本地生成UUID的 `CONT_OPEN` 请求，然后使用IV（Incast变量）将handle异步广播到pool 中所有启用的target。

```
1 static int
2 dc_cont_open_internal(tse_task_t *task, const char *label, struct dc_pool
3 *pool)
4 {
5     enum cont_operation cont_op;
6     cont_op = label ? CONT_OPEN_BYLABEL : CONT_OPEN;
7
8     rc = cont_req_create(daos_task2ctx(task), &ep, cont_op, &rpc);
9     if (rc != 0) {
10         D_ERROR("failed to create rpc: %s\n", DP_RC(rc));
11         goto err;
12     }
13     ...
14     /* send the request */
15     return daos_rpc_send(rpc, task);
16 }
17
```

成功完成后，它会在container handle KVS中创建一个新条目。

客户端可以通过向容器服务发送 `CONT_CLOSE` 请求来广播关闭作为闭容器句柄。然后，它从容器句柄 KVS 中删除相应的条目，并丢弃对未提交的句柄执行的更新。

当客户端向容器服务发送一个 `CONT_DESTROY` 请求其清除所有元数据时，容器将被销毁。类似地，Target 对象集体接收来自容器服务的 `CONT_TGT_DESTROY` 请求，并删除与该容器相关联的所有数据，包括该容器内的所有对象。当容器处理正在当前已打开的 Handles 时，客户端可以选择性地强制销毁它。

Epoch Protocol

epoch protocol 实现事务模型中描述的 epoch 模型。Container service 管理 container 的 epoch；它将确定的 epoch 状态作为 container 元数据的一部分进行维护，而 target service 对全局 epoch 状态知之甚少。因此，Epoch 提交、丢弃和聚合过程都是由 container service 驱动的。

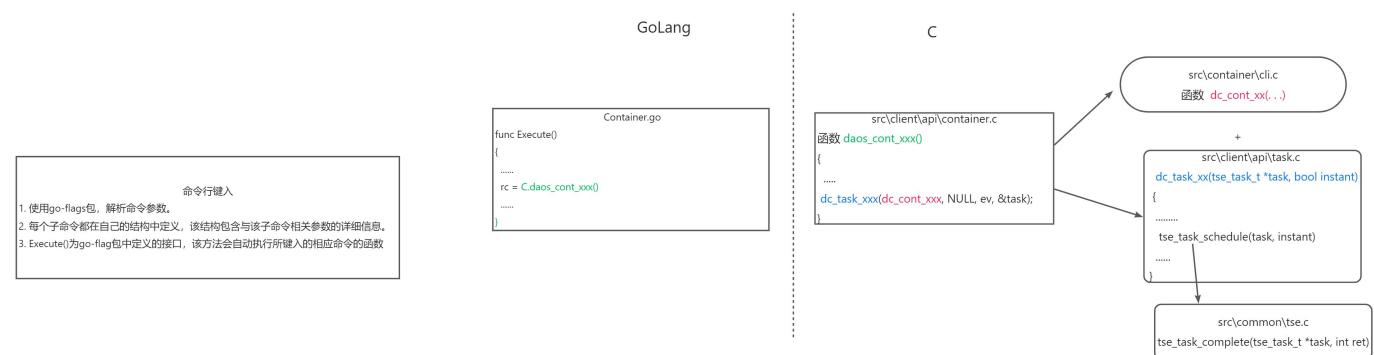
在每个 target 上，target service 急切地将传入的写操作存储到匹配的 VOS container 中。如果 container handle 丢弃一个 epoch，VOS 将帮助丢弃与该 container handle 关联的所有写入操作。当写入操作成功时，在相同或更高的时间段中，冲突操作会立即看到该操作。VOS 将拒绝具有相同 epoch 的冲突写入操作，除非该操作与相同的 container handle 关联，并且具有与已执行的操作相同的内容。

在提交一个 epoch 之前，应用程序必须确保 target services 已经为这个 epoch 保留了足够的一组写操作。应用程序可能决定丢失一些写操作是可以接受的，这取决于每个操作所采用的冗余方案。提交 container handle 的 epoch 将导致对相应 container 服务的 `CONT_EPOCH_COMMIT` 请求，该请求只更新元数据。当更新持久化时，container 服务将以新的状态回复客户机。

container 的相关 Task 结构体

src\container\cli.c

src\client\api\container.c



- 命令行键入
1. 使用 go-flag 包，解析命令参数。
2. 每个子命令都在自己的结构中定义，该结构包含与该子命令相关参数的详细信息。
3. Execute() 为 go-flag 包中定义的接口，该方法会自动执行所键入的相应命令的函数

Go端container

Container cmd命令

```

1  type containerCmd struct {
2      Create      containerCreateCmd      `command:"create" description:"cre
3      List       containerListCmd        `command:"list" alias:"ls" descrip
4      Destroy     containerDestroyCmd    `command:"destroy" description:"de
5      ListObjects containerListObjectsCmd `command:"list-objects" alias:"lis
t-obj" description:"list all objects in container"`
6      Query      containerQueryCmd      `command:"query" description:"quer
y a container"`
7      Stat       containerStatCmd       `command:"stat" description:"get c
ontainer statistics"`
8      Clone      containerCloneCmd     `command:"clone" description:"clon
e a container"`
9      Check      containerCheckCmd     `command:"check" description:"che
ck objects' consistency in a container"`
10
11      ListAttributes containerListAttrsCmd `command:"list-attr" alias:"list
-atr" alias:"lsattr" description:"list container user-defined attribute
s"`
12      DeleteAttribute containerDelAttrCmd `command:"del-attr" alias:"delat
tr" description:"delete container user-defined attribute"`
13      GetAttribute   containerGetAttrCmd `command:"get-attr" alias:"getat
tr" description:"get container user-defined attribute"`
14      SetAttribute   containerSetAttrCmd `command:"set-attr" alias:"setat
tr" description:"set container user-defined attribute"`
15
16      GetProperty   containerGetPropCmd `command:"get-prop" alias:"getprop" de
scription:"get container user-defined attribute"`
17      SetProperty   containerSetPropCmd `command:"set-prop" alias:"setprop" de
scription:"set container user-defined attribute"`
18
19      GetACL       containerGetACLCmd   `command:"get-acl" descriptio
n:"get a container's ACL"`
20      OverwriteACL containerOverwriteACLCmd `command:"overwrite-acl" alia
s:"replace" description:"replace a container's ACL"`
21      UpdateACL    containerUpdateACLCmd `command:"update-acl" descriptio
n:"update a container's ACL"`
22      DeleteACL    containerDeleteACLCmd `command:"delete-acl" descriptio
n:"delete a container's ACL"`
23      SetOwner     containerSetOwnerCmd  `command:"set-owner" alias:"chow
n" description:"change ownership for a container"`
24
25

```

```

26     CreateSnapshot      containerSnapCreateCmd      `command:"create-snap" alias:"snap" description:"create container snapshot"`
27     DestroySnapshot    containerSnapDestroyCmd    `command:"destroy-snap" description:"destroy container snapshot"`
28     ListSnapshots      containerSnapListCmd      `command:"list-snaps" alias:"list-snaps" description:"list container snapshots"`
29     RollbackSnapshot   containerSnapshotRollbackCmd `command:"rollback-snap" description:"rollback container snapshot"`
30     QuerySnapshotRollback containerSnapshotRBQueryCmd `command:"query-snap-rollback" description:"query container snapshot rollback"`
31     QuerySnapshotDestory  containerSnapshotDESQueryCmd `command:"query-snap-destory" description:"query container snapshot destory"`
}

```

containerBaseCmd

```

1 ▾ type containerBaseCmd struct {
2     poolBaseCmd
3     contUUID  uuid.UUID
4     contLabel string
5
6     cContHandle C.daos_handle_t
7 }
8
9 //containerBaseCmd包含了五种方法
10 func (cmd *containerBaseCmd) contUUIDPtr()
11 func (cmd *containerBaseCmd) openContainer(openFlags C.uint)
12 func (cmd *containerBaseCmd) closeContainer()
13 func (cmd *containerBaseCmd) queryContainer()
14 func (cmd *containerBaseCmd) connectPool(flags C.uint, ap *C.struct_cmd_args_s)

```

containerCreateCmd

用于container的创建

```
1 type containerCreateCmd struct {
2     containerBaseCmd
3
4     Type      ContTypeFlag      `long:"type" short:"t" description:"c
5         ontainer type"`
6     Path      string           `long:"path" short:"d" description:"c
7         ontainer namespace path"`
8     ChunkSize ChunkSizeFlag    `long:"chunk-size" short:"z" descript
9         ion:"container chunk size"`
10    ObjectClass ObjClassFlag   `long:"oclass" short:"o" descriptio
11        n:"default object class"`
12    Properties CreatePropertiesFlag `long:"properties" description:"conta
13        iner properties"`
14    Label      string           `long:"label" short:"l" descriptio
15        n:"container label"`
16    Mode      ConsModeFlag    `long:"mode" short:"M" description:"D
17        FS consistency mode"`
18    ACLFile   string           `long:"acl-file" short:"A" descriptio
19        n:"input file containing ACL"`
20    User      string           `long:"user" short:"u" description:"u
21        ser who will own the container (username@[domain])"`
22    Group     string           `long:"group" short:"g" descriptio
23        n:"group who will own the container (group@[domain])"`
24    ContFlag   ContainerID     `long:"cont" short:"c" description:"c
25        ontainer UUID (optional)"`
26 }
```

Go | 复制代码

```
1 func (cmd *containerCreateCmd) Execute(_ []string) (err error){
2     //解析参数
3     ap, deallocCmdArgs, err := allocCmdArgs(cmd.log)
4     if err != nil {
5         return err
6     }
7     defer deallocateCmdArgs()
8
9     ...
10    //连接到Pool
11    disconnectPool, err := cmd.connectPool(C.DAO5_PC_RW, ap)
12
13    //ap是C端代码的参数结构体
14    ap.c_op = C.CONT_CREATE
15
16    var rc C.int
17    rc = C.cont_create_hdldr(ap) //调用C端代码创建Container
18    ...
19 }
```

C端container

C | 复制代码

```
1 /* cont_create_hdldr() - create container by UUID */
2 int cont_create_hdldr(struct cmd_args_s *ap)
3 {
4     int rc;
5     ...
6
7     //Daos conatiner接口
8     rc = daos_cont_create(ap->pool, ap->c_uuid, ap->props, NULL);
9
10    if (rc != 0) {
11        DH_PERROR_DER(ap, rc, "failed to create container");
12        return rc;
13    }
14
15    return rc;
16 }
```

C | 复制代码

```
1 daos_cont_create(daos_handle_t poh, uuid_t *cuuid, daos_prop_t *cont_prop
, daos_event_t *ev)
2 {
3     tse_task_t      *task;
4     ...
5
6     rc = dc_task_create(dc_cont_create, NULL, ev, &task);
7     ...
8
9     return dc_task_schedule(task, true);
10 }
```

其中 `daos_cont_create` 中上面代码中出现了三个重要的函数 `dc_task_create` , `dc_cont_create` , `dc_task_schedule` 和一个用于管理task的调度结构体 `tse_task_t` 。下面将展开介绍

- `dc_task_create`
- `dc_cont_create`
- `dc_task_schedule`

C | 复制代码

```
1 /* 调度由dc_task_create_ev () 创建的任务, 如果任务的关联事件是私有事件,
2  * 则此函数将等待任务完成, 否则它将立即返回, 并通过测试事件或轮询EQ找到其完成
3  * 如果instant 真, 任务将立即执行
4  */
5 int dc_task_schedule(tse_task_t *task, bool instant)
6 {
7     daos_event_t *ev;
8     int          rc;
9
10    D_ASSERT(task_is_valid(task));
11
12    ev = task_ptr2args(task)->ta_ev;
13    if (ev) {
14        rc = daos_event_launch(ev);
15        ...
16    }
17
18    rc = tse_task_schedule(task, instant);
19    ...
20 }
```

```
1 int tse_task_schedule(tse_task_t *task, bool instant)
2 {
3     return tse_task_schedule_with_delay(task, instant, 0 /* delay */);
4 }
5
6
7 int tse_task_schedule_with_delay(tse_task_t *task, bool instant, uint64_t
8 delay)
9 {
10     struct tse_task_private *dtp = tse_task2priv(task);
11     struct tse_sched_private *dsp = dtp->dtp_sched;
12     int rc = 0;
13
14     /* Add task to scheduler */
15     D_MUTEX_LOCK(&dsp-> dsp_lock);
16     if (dtp->dtp_func == NULL || instant) {
17         /** If task has no body function, mark it as running */
18         dsp-> dsp_inflight++;
19         dtp-> dtp_running = 1;
20         dtp-> dtp_wakeup_time = 0;
21         d_list_add_tail(&dtp-> dtp_list, & dsp-> dsp_running_list);
22
23         /** +1 in case task is completed in body function */
24         if (instant)
25             tse_task_addrf_locked(dtp);
26     } else if (delay == 0) {
27         /** Otherwise, scheduler will process it from init list */
28         dtp-> dtp_wakeup_time = 0;
29         d_list_add_tail(&dtp-> dtp_list, & dsp-> dsp_init_list);
30     } else { ... }
31     ...
32
33     // 调度器立即执行task任务
34     if (instant) { ... }
35 }
36
37
38
39
40
41
42
43
44
45
46
47
48 }
49
```

C | 复制代码

```
1 daos_cont_create_t cont_create;
2 daos_cont_open_t cont_open;
3 daos_cont_close_t cont_close;
4 daos_cont_destroy_t cont_destroy;
5 daos_cont_query_t cont_query;
6 daos_cont_aggregate_t cont_aggregate;
7 daos_cont_rollback_t cont_rollback;
8 daos_cont_subscribe_t cont_subscribe;
9 daos_cont_list_attr_t cont_list_attr;
10 daos_cont_get_attr_t cont_get_attr;
11 daos_cont_set_attr_t cont_set_attr;
12 daos_cont_alloc_oids_t cont_alloc_oids;
13 daos_cont_list_snap_t cont_list_snap;
14 daos_cont_create_snap_t cont_create_snap;
15 daos_cont_destroy_snap_t cont_destroy_snap;
```

1. 各种DAO组件（如容器）的通用handle

C | 复制代码

```
1 /** Generic handle for various DAOS components like container, object, et
c. */
2 typedef struct {
3     uint64_t cookie;
4 } daos_handle_t;
```

2. 创建container

```
1  /** Container create args */
2  typedef struct {
3      /** Pool open handle. */
4      daos_handle_t          poh;
5      /**
6       * Deprecated, the container UUID is generated by the DAOS system.
7       * and returned in the cuuid field.
8       */
9      uuid_t                 uuid;
10     /** Optional container properties. */
11     daos_prop_t            *prop;
12     /** Optional returned the allocated container UUID */
13     uuid_t                 *cuuid;
14 } daos_cont_create_t;
15
16 //历史版本
17 int daos_cont_create(daos_handle_t poh, uuid_t *cuuid,
18                      daos_prop_t *cont_prop, daos_event_t *ev)
19
20 //Create version that requires uuid to be passed in
21 int daos_cont_create1(daos_handle_t poh, const uuid_t cuuid,
22                      daos_prop_t *cont_prop, daos_event_t *ev)
23
24 //z最新版本使用
25 int daos_cont_create2(daos_handle_t poh, uuid_t *cuuid, daos_prop_t *cont_
26 prop,
27                         daos_event_t *ev)
28 //
29 int daos_cont_create_with_label(daos_handle_t poh, const char *label,
30                                 daos_prop_t *cont_prop, uuid_t *uuid,
31                                 daos_event_t *ev)
```

3. 打开container

C | 复制代码

```
1  /** Container create args */
2  typedef struct {
3      /** Pool open handle. */
4      daos_handle_t      poh;
5      /**
6       * Deprecated, the container UUID is generated by the DAOS system.
7       * and returned in the cuuid field.
8       */
9      uuid_t            uuid;
10     /** Optional container properties. */
11     daos_prop_t       *prop;
12     /** Optional returned the allocated container UUID */
13     uuid_t            *cuuid;
14 } daos_cont_create_t;
15
16 int daos_cont_open(daos_handle_t poh, const char *cont, unsigned int flags
17 ,
18                 daos_handle_t *coh, daos_cont_info_t *info, daos_event_t *ev)
19 //新版本
20 int daos_cont_open2(daos_handle_t poh, const char *cont, unsigned int flags,
21                      daos_handle_t *coh, daos_cont_info_t *info, daos_event_t *
22                      ev)
```

4. 关闭container

C | 复制代码

```
1  /** Container close args */
2  typedef struct {
3      /** Container open handle. */
4      daos_handle_t      coh;
5 } daos_cont_close_t;
6
7 int daos_cont_close(daos_handle_t coh, daos_event_t *ev)
```

5. 销毁container

C | 复制代码

```
1 typedef struct {
2     /** Pool open handle. */
3     daos_handle_t      poh;
4     /**
5     * Deprecated, container UUID replaced by UUID string or label via the
6     * cont arg at the end of this structure.
7     */
8     uuid_t            uuid;
9     /** Force destroy even if there is outstanding open handles. */
10    int               force;
11    /** Container (UUID string or label) to destroy, API v1.3.0 */
12    const char         *cont;
13 } daos_cont_destroy_t;
14
15 int daos_cont_destroy(daos_handle_t poh, const char *cont, int force,
16             daos_event_t *ev)
17
18 //当前使用
19 int daos_cont_destroy2(daos_handle_t poh, const char *cont, int force,
20             daos_event_t *ev)
```

6. 查询container

C | 复制代码

```
1 typedef struct {
2     /** Container open handle. */
3     daos_handle_t      coh;
4     /** Returned container information. */
5     daos_cont_info_t   *info;
6     /** Optional, returned container properties. */
7     daos_prop_t        *prop;
8 } daos_cont_query_t;
9
10 int daos_cont_query(daos_handle_t coh, daos_cont_info_t *info,
11             daos_prop_t *cont_prop, daos_event_t *ev)
```

7. 要回滚到的持久快照的epoch

```
1 typedef struct {
2     /** Container open handle. */
3     daos_handle_t coh;
4     /** Epoch of a persistent snapshot to rollback to. */
5     daos_epoch_t epoch;
6 } daos_cont_rollback_t;
7
8 int daos_cont_rollback(daos_handle_t coh, daos_epoch_t epoch, daos_event_t
*ev)
```

8. Container subscribe

```
1 typedef struct {
2     /** Container open handle. */
3     daos_handle_t coh;
4     /*
5      * [in]: epoch of snapshot to wait for.
6      * [out]: epoch of persistent snapshot taken.
7      */
8     daos_epoch_t *epoch;
9 } daos_cont_subscribe_t;
10
11 int daos_cont_subscribe(daos_handle_t coh, daos_epoch_t *epoch,
daos_event_t *ev)
```

10. OID分配

```
1 /** Container Object ID allocation args */
2 typedef struct {
3     /** Container open handle. */
4     daos_handle_t coh;
5     /** Number of unique IDs requested. */
6     daos_size_t num_oids;
7     /** starting oid that was allocated up to oid + num_oids. */
8     uint64_t *oid;
9 } daos_cont_alloc_oids_t;
10
11 int daos_cont_alloc_oids(daos_handle_t coh, daos_size_t num_oids,
uint64_t *oid, daos_event_t *ev)
```

11. 创建container快照

```
1  /** Container snapshot creation args */
2  typedef struct {
3      /** Container open handle. */
4      daos_handle_t coh;
5      /** bit flags, see enum daos_snapshot_opts */
6      unsigned int opts;
7      /** Returned epoch of persistent snapshot taken. */
8      daos_epoch_t *epoch;
9      /** Optional null terminated name for snapshot. */
10     char *name;
11 } daos_cont_create_snap_t;
12
13 int daos_cont_create_snap(daos_handle_t coh, daos_epoch_t *epoch, char *na
14 me,
                           daos_event_t *ev)
```

12. 销毁container快照

```
1  /** Container snapshot destroy args */
2  typedef struct {
3      /** Container open handle. */
4      daos_handle_t coh;
5      /** Epoch range of snapshots to destroy. */
6      daos_epoch_range_t epr;
7 } daos_cont_destroy_snap_t;
8
9 int daos_cont_destroy_snap(daos_handle_t coh, daos_epoch_range_t epr,
10                           daos_event_t *ev)
```

13. container快照列表

C | 复制代码

```
1  /** Container snapshot listing args */
2  typedef struct {
3      /** Container open handle. */
4      daos_handle_t coh;
5      /*
6       * [in]: Number of snapshots in epochs and names.
7       * [out]: Actual number of snapshots returned
8       */
9      int *nr;
10     /** preallocated array of epochs to store snapshots. */
11     daos_epoch_t *epochs;
12     /** preallocated array of names of the snapshots. */
13     char **names;
14     /** Hash anchor for the next call. */
15     daos_anchor_t *anchor;
16 } daos_cont_list_snap_t;
17
18 int daos_cont_list_snap(daos_handle_t coh, int *nr, daos_epoch_t *epochs,
19                         char **names, daos_anchor_t *anchor, daos_event_t
*ev)
```

Daos Object

Daos 并发模型 (VOS)

Blob I/O

Daos 数据结构与算法

18. Daos 公共库

19. 客户端库和I/O中间件

4. Mga Tool

5. Daos Agent

6. Daos 验证管理

7. Daos 安全

DAOS Client Library

DAOS API分为以下几个功能：

- Management API: pool and target management
- Pool Client API: pool access
- Container API: container management and access, container 快照
- Transaction API: transaction model and concurrency control
- Object, Array and KV APIs: object and data management and access
- Event, Event Queue, and Task API: non-blocking operations
- Addons API: array and KV operations built over the DAOS object API
- DFS API: DAOS file system API to emulate a POSIX namespace over DAOS
- DUNS API: DAOS unified namespace API for integration with an existing system namespace.

Client API:

Management API:

Pool Client API

文件路径 `src\client\api\mgmt.c`

Container API

Object, Array and KV APIs

Event, Event Queue, and Task API

Addons API

DFS API

DAOS Common Libraries

所有 DAOS 组件之间共享的通用功能和基础结构在外部共享库中提供。这包括以下功能：

- Hash 和 checksum
- 支持对非阻塞操作的事件和事件队列
- 日志记录和调试基础结构
- 锁定primitives
- 网络传输

Task Scheduler Engine(TSE)

TSE 是一个通用库，用于创建具有函数回调的通用task，可以选择在这些task之间添加依赖关系，并将它们安排在引擎中，该引擎将按照插入这些task的依赖关系图确定的顺序执行这些task。task相关性图是计划程序不可或缺的一部分，允许用户创建多个task并以非阻塞方式推进它们

TSE不是特定于DAOS的，但曾经是DAOS核心的一部分，后来作为独立的API被提取到common src中。该API是通用的，允许在引擎中创建task。DAOS库提供了一个基于TSE构建的task API。此外，DAOS在内部使用TSE来跟踪和推进与API事件关联的所有API task，并且在某些情况下，安排与单个API task相对应的几个正在进行的“子”任务，并添加对该task的依赖关系以跟踪所有这些正在进行的“子”任务。

Scheduler API

调度程序 API 允许用户创建通用调度程序并向其添加 task。在创建调度程序时，用户可以注册一个完成回调函数，以便在调度程序完成时调用。

添加到计划程序中的 task不会自行进行。必须显式调用调度程序上的进度函数 (daos_sched_progress) 才能在引擎中的 task上取得进展。用户可以偶尔在其程序中调用此进度函数，也可以分叉重复调用进度函数的单个线程。

任务接口

Task API 允许使用通用函数创建任务，并将其添加到计划程序中。在计划程序中创建任务后，如果没有用户对任务计划函数的显式调用，则不会实际计划该任务运行，除非它是任务相关性图的一部分，在这种情况下，仅对关系图中的第一个任务需要显式计划调用。创建任务后，用户可以为任务注册任意数量的依赖项，这些依赖项需要先完成，然后才能计划运行该任务。此外，用户将能够注册任务的准备和完成回调：

- 准备回调在 task准备好运行但尚未执行时执行，这意味着创建 task时使用的依赖项已完成，并且计划程序已准备好计划 task。当要计划的 task需要的信息在 task创建时不可用，但在 task的依赖关系完成后可用时，这很有用;例如，为 task正文函数设置一些输入参数。
- 完成回调在 task完成执行时执行，并且用户需要执行更多工作或处理。一个有用的示例是设置在 TSE 之上构建的更高级别事件或请求的完成，或者跟踪依赖项列表中多个 task的错误状态。

task API 上还存在其他几个功能来支持：

- 在 task本身上设置一些可以查询的私有数据。
- 在没有数据拷贝的情况下在 task堆栈空间上/从 task堆栈空间推送和弹出数据
- 通用 task列表

有关该功能的更多详细信息，请参阅[此处](#) DAOS 代码中的 TSE 头文件。

Event & Event Queue

DAOS API 函数可以在阻塞或非阻塞模式下使用。这是通过指向传递给每个 API 调用的 DAOS 事件的指针确定的

- 如果 NULL 指示要阻止该操作。完成操作后，操作将返回。所有故障情况的错误码将通过 API 函数本身的返回码返回。
- 如果使用了valid event，则操作将在非阻塞模式下运行，并在内部调度程序调度操作和将 RPC 提交到底层堆栈后 立即返回。如果调度成功，则操作的返回值为成功，但不表示实际操作成功。返回时可以捕获的错误要么是无效的参数，要么是调度问题。当事件完成时，操作的实际返回代码将在事件错误代码 (event.ev_error) 中可用。

必须首先使用单独的 API 调用创建要使用的有效事件。为了允许用户一次跟踪多个事件，可以将事件创建为event queue的一部分，该队列基本上是可以一起进行和轮询的事件的集合。事件队列还在内部为所有 DAOS 任务以及新的网络上下文创建单独的任务计划程序。在某些网络提供商上，网络上下文创建是一项代价高昂的操作，因此用户应尝试限制在其应用程序或 IO 中间件库（位于 DAOS 之上）中创建的事件队列的数量。或者，可以在没有事件队列的情况下创建事件，并单独跟踪。在这种情况下，对于阻塞操作，内部全局任务计划程序和网络上下文将用于将为事件队列创建的独立任务计划程序和网络上下文。一旦事件完成，它可以重新用于另一个DAOS API调用，以最大限度地减少在DAOS库中创建和分配事件的需求。

Task Engine Integration

DAOS TASK API提供了一种以非阻塞方式使用DAOS API的替代方法，同时在DAOS API操作之间构建 task 依赖关系树。这对于使用DAOS的应用程序和中间件库非常有用，因为它们需要构建 DAOS 操作的时间表，并且 DAOS 操作之间存在依赖关系 (N-1、1-N、N-N) 。

要利用TASK API，用户需要创建一个调度器 `tse_sched_t`，在该调度器中可以创建DAOS TASK作为TASK API的一部分。TASK API足够通用，允许用户混合DAOS特定的 task（通过 DAOS task API）和其他用户定义的 task，并在这些 task 之间添加依赖项。

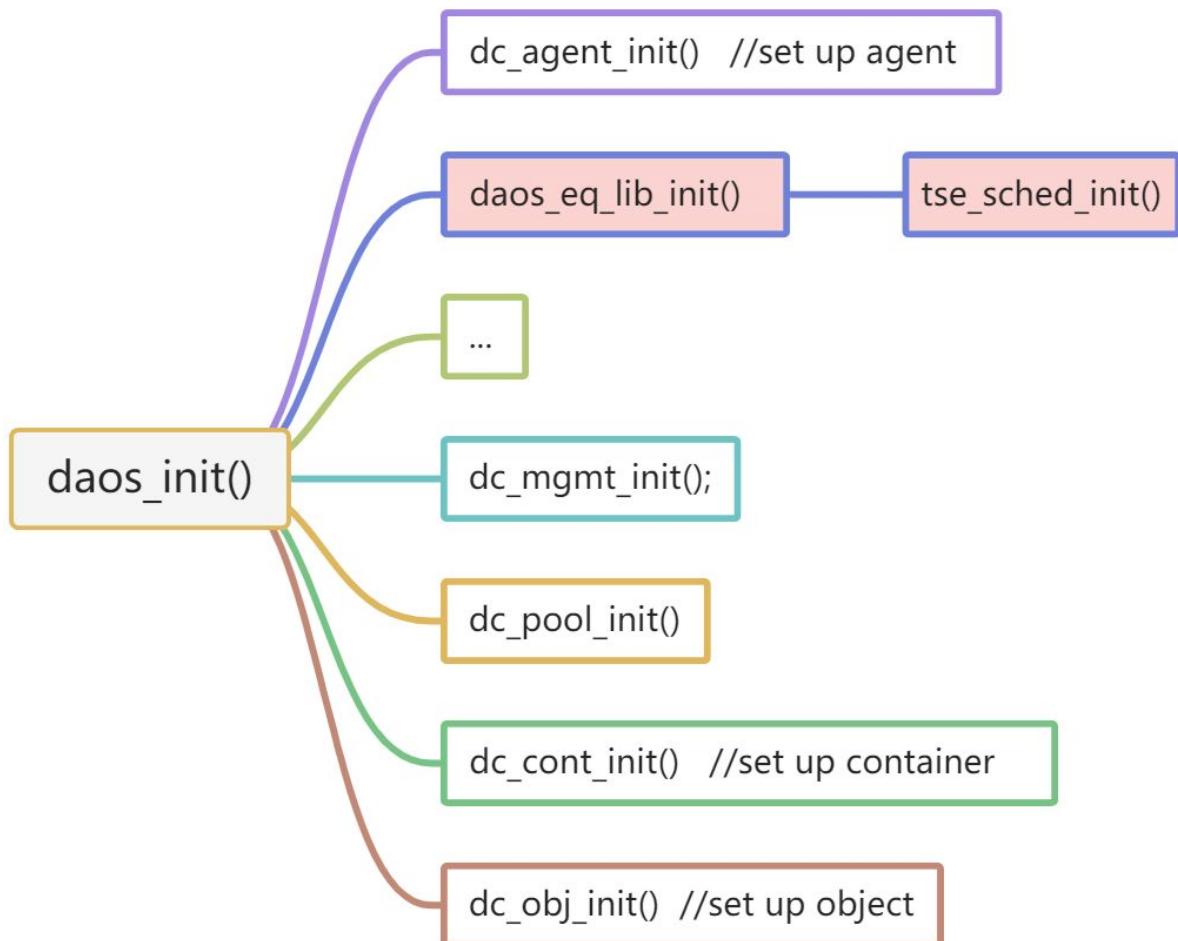
tse_sched_t

C | 复制代码

```

1 // 使用schedule 追踪所有 task
2 typedef struct {
3     int      ds_result;
4
5     /* user data associated with the scheduler (completion cb data, etc.)
6 */
7
8     /* daos schedule internal */
9     struct {
10        uint64_t    ds_space[48];
11        ds_private;
12 } tse_sched_t;

```

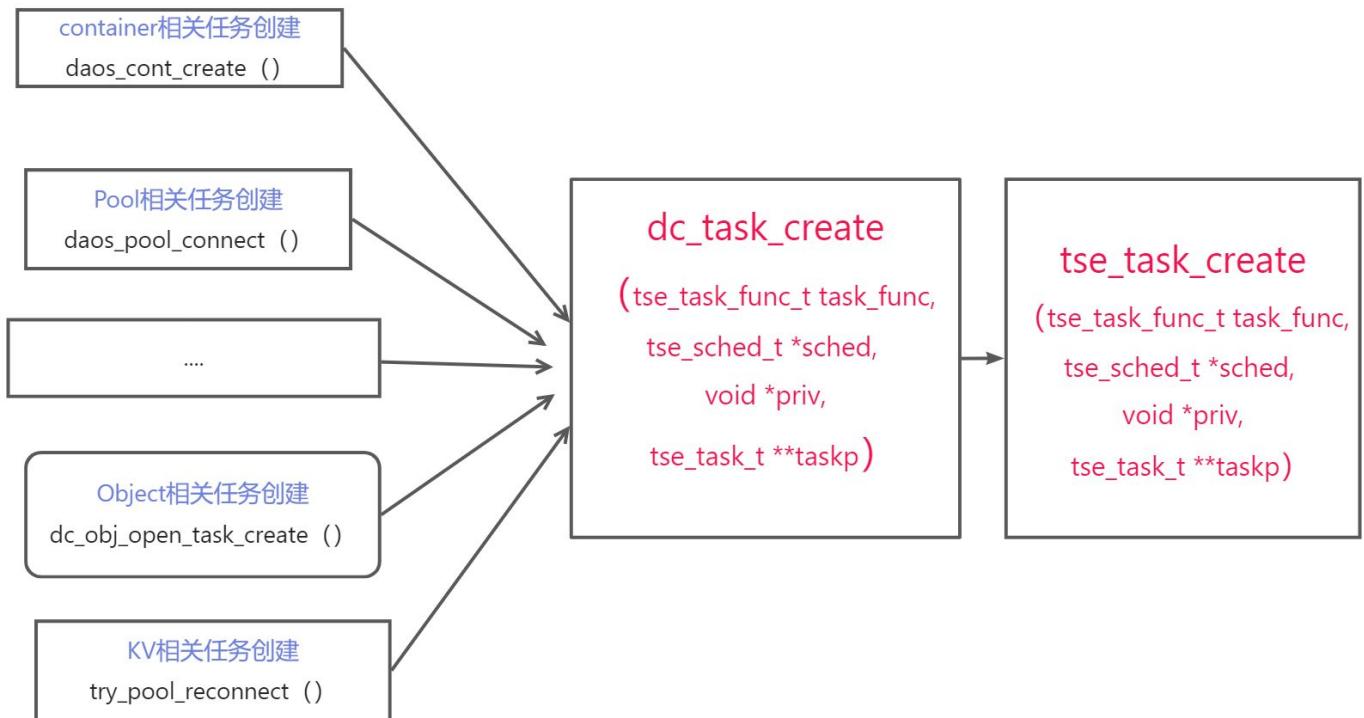


`tse_sched_init` 使用可选的完成回调和指向用户数据的指针初始化调度程序。调用者负责完成或取消计划

```

1  /* sched [input]          scheduler to be initialized.
2  * comp_cb [input]        Optional callback to be called when scheduler is d
3  one.
4  *
5  *
6  *                      Optional pointer to user data to associate with
7  the scheduler. This is stored in ds_udata in the
8  scheduler struct and passed in to comp_cb as the
9  argument when the callback is invoked.
10 *
11 int tse_sched_init(tse_sched_t *sched, tse_sched_comp_cb_t comp_cb,
12                     void *udata)
13 {
14     struct tse_sched_private    *dsp = tse_sched2priv(sched);
15     pthread_mutexattr_t         attr;
16     int rc;
17
18     ...
19
20     if (comp_cb != NULL) {
21         // 在调度器上 注册回调函数
22         rc = tse_sched_register_comp_cb(sched, comp_cb, udata);
23         if (rc != 0)
24             return rc;
25     }
26
27     sched->ds_udata = udata;
28     sched->ds_result = 0;
29
30     return 0;
31 }
```

- **dc_task_create ()** : 负责一系列的event的自动调度 (Pool、Container、Object、KV等) , 创建新task并将其与输入event关联。如果事件为空, 则将获取私有事件。此函数创建的任务只能通过调用dc_task_sched_ev () 来调度, 否则事件将永远不会完成。
- **tse_task_create ()** : 初始化tse_task。该task将添加到计划程序任务列表中, 并在稍后进行计划。如果提供了相关任务, 则该任务将被添加到相关任务的dep列表中, 一旦相关任务完成, 则将该任务添加到计划计划程序列表中。



Array的事件创立

C array.c src\client\api 2

```

rc = dc_task_create(dc_array_create, NULL, ev, &task);
rc = dc_task_create(dc_array_open, NULL, ev, &task);
rc = dc_task_create(dc_array_open, NULL, ev, &task);
rc = dc_task_create(dc_array_open, NULL, ev, &task);
rc = dc_task_create(dc_array_close, NULL, ev, &task);
rc = dc_task_create(dc_array_destroy, NULL, ev, &task);
rc = dc_task_create(dc_array_read, NULL, ev, &task);
rc = dc_task_create(dc_array_write, NULL, ev, &task);
rc = dc_task_create(dc_array_punch, NULL, ev, &task);
rc = dc_task_create(dc_array_get_size, NULL, ev, &task);
rc = dc_task_create(dc_array_set_size, NULL, ev, &task);

```

Container的事件创立

✓ C container.c src\client\api 2, M 23 X

```
rc = dc_task_create(dc_cont_create, NULL, ev, &task);
rc = dc_task_create(dc_cont_create, NULL, ev, &task);
rc = dc_task_create(dc_cont_open, NULL, ev, &task);
rc = dc_task_create(dc_cont_open, NULL, ev, &task);
rc = dc_task_create(dc_cont_close, NULL, ev, &task);
rc = dc_task_create(dc_cont_destroy, NULL, ev, &task);
rc = dc_task_create(dc_cont_destroy, NULL, ev, &task);
rc = dc_task_create(dc_cont_query, NULL, ev, &task);
rc = dc_task_create(dc_cont_set_prop, NULL, ev, &task);
rc = dc_task_create(dc_cont_set_prop, NULL, ev, &task);
rc = dc_task_create(dc_cont_update_acl, NULL, ev, &task);
rc = dc_task_create(dc_cont_delete_acl, NULL, ev, &task);
rc = dc_task_create(dc_cont_aggregate, NULL, ev, &task);
rc = dc_task_create(dc_cont_rollback, NULL, ev, &task);
rc = dc_task_create(dc_cont_subscribe, NULL, ev, &task);
rc = dc_task_create(dc_cont_alloc_oids, NULL, ev, &task);
rc = dc_task_create(dc_cont_list_attr, NULL, ev, &task);
rc = dc_task_create(dc_cont_get_attr, NULL, ev, &task);
rc = dc_task_create(dc_cont_set_attr, NULL, ev, &task);
```

Pool 的事件创立

C pool.c src\client\api

```
rc = dc_task_create(dc_pool_connect, NULL, ev, &task);
rc = dc_task_create(dc_pool_connect, NULL, ev, &task);
rc = dc_task_create(dc_pool_disconnect, NULL, ev, &task);
rc = dc_task_create(dc_pool_query, NULL, ev, &task);
rc = dc_task_create(dc_pool_query_target, NULL, ev, &task);
rc = dc_task_create(dc_pool_list_cont, NULL, ev, &task);
rc = dc_task_create(dc_pool_list_attr, NULL, ev, &task);
rc = dc_task_create(dc_pool_get_attr, NULL, ev, &task);
rc = dc_task_create(dc_pool_set_attr, NULL, ev, &task);
rc = dc_task_create(dc_pool_del_attr, NULL, ev, &task);
rc = dc_task_create(dc_pool_stop_svc, NULL, ev, &task);
```

Object 的事件创立

✓ C obj_task.c src\object

```

rc = dc_task_create(dc_obj_open, tse, ev, task);
rc = dc_task_create(dc_obj_open, tse, ev, task);
rc = dc_task_create(dc_obj_close, tse, ev, task);
rc = dc_task_create(dc_obj_punch_task, tse, ev, task);
rc = dc_task_create(dc_obj_punch_dkeys_task, tse, ev, task);
rc = dc_task_create(dc_obj_punch_akeys_task, tse, ev, task);
rc = dc_task_create(dc_obj_query_key, tse, ev, task);
rc = dc_task_create(dc_obj_sync, tse, ev, task);
rc = dc_task_create(dc_obj_fetch_task, tse, ev, task);
rc = dc_task_create(dc_hyper_obj_fetch_task, tse, ev, task);
rc = dc_task_create(dc_obj_update_task, tse, ev, task);
rc = dc_task_create(dc_obj_list_dkey, tse, ev, task);
rc = dc_task_create(dc_obj_list_akey, tse, ev, task);
rc = dc_task_create(dc_obj_list_rec, tse, ev, task);
rc = dc_task_create(dc_obj_list_obj, tse, ev, task);
rc = dc_task_create(dc_obj_update_task, tse, ev, task);
rc = dc_task_create(dc_obj_fetch_task, tse, ev, task);
rc = dc_task_create(dc_notify_hang_task, tse, ev, task);
rc = dc_task_create(dc_notify_update_task, tse, ev, task);
rc = dc_task_create(dc_obj_update_task, tse, ev, task);
rc = dc_task_create(dc_keepalive_task, tse, ev, task);
rc = dc_task_create(dc_hyper_obj_list_obj, tse, ev, task);
rc = dc_task_create(dc_list_online_watchers_task, tse, ev, task);
rc = dc_task_create(dc_cont_check_exist_task, tse, ev, task);

```

Go代码

```
src\control\cmd\daos\pool.go
func (cmd *poolQueryCmd) Execute(_ []string) error {
    C.daos_pool_query()
```

C.daos_pool_query()
GoLang 调用 C

C代码

```
src\client\api\pool.c
int daos_pool_query{
    rc = dc_task_create(dc_pool_query, NULL, ev,
    &task);
}
```

src\client\api\task.c
创建task并绑定event

Path: src\pool\cli.c
daos client 函数

值得注意的是，在同一文件下 `src\client\api\task.c`，还有一个类似的函数 `daos_task_create()`，以下是两个函数的对比，为何这样做待研究？？？：

```
1 int daos_task_create(daos_opc_t opc, tse_sched_t *sched, unsigned int num_d  
2 eps,  
3 tse_task_t *dep_tasks[], tse_task_t **taskp)  
4 int dc_task_create(tse_task_func_t func, tse_sched_t *sched, daos_event_t *  
5 ev,  
6 tse_task_t **taskp)
```

DPRC

枚举类型

drpc_module

dRPC模块用于通过Unix域套接字将通信多路传输到适当的处理程序。在Drpc_Call结构中，dRPC模块ID必须是唯一的。以下是DAOS DRPC模块的列表，分为五个部分，每个部分保留ID号100。

```
1 //drpc模块名称ID  
2 enum drpc_module {  
3     DRPC_MODULE_TEST      = 0,    /* Reserved for testing */  
4     DRPC_MODULE_SEC_AGENT = 1,    /* daos_agent security */  
5     DRPC_MODULE_MGMT      = 2,    /* daos_server mgmt */  
6     DRPC_MODULE_SRV       = 3,    /* daos_server */  
7     DRPC_MODULE_SEC       = 4,    /* daos_server security */  
8  
9     NUM_DRPC_MODULES      /* Must be last */  
10};
```

C | ⚒ 复制代码

```
1 enum drpc_sec_agent_method {
2     DRPC_METHOD_SEC_AGENT_REQUEST_CREDS = 101,
3
4     NUM_DRPC_SEC_AGENT_METHODS          /* Must be last */
5 };
```

```
1 //管理模块的函数ID
2 enum drpc_mgmt_method {
3     DRPC_METHOD_MGMT_KILL_RANK      = 201,
4     DRPC_METHOD_MGMT_SET_RANK       = 202,
5     DRPC_METHOD_MGMT_CREATE_MS      = 203,
6     DRPC_METHOD_MGMT_START_MS       = 204,
7     DRPC_METHOD_MGMT_JOIN           = 205,
8     DRPC_METHOD_MGMT_GET_ATTACH_INFO = 206,
9     DRPC_METHOD_MGMT_POOL_CREATE    = 207,
10    DRPC_METHOD_MGMT_POOL_DESTROY   = 208,
11    DRPC_METHOD_MGMT_SET_UP         = 209,
12    DRPC_METHOD_MGMT_BIO_HEALTH_QUERY = 210,
13    DRPC_METHOD_MGMT_SMD_LIST_DEVS  = 211,
14    DRPC_METHOD_MGMT_SMD_LIST_POOLS = 212,
15    DRPC_METHOD_MGMT_POOL_GET_ACL   = 213,
16    DRPC_METHOD_MGMT_LIST_POOLS     = 214,
17    DRPC_METHOD_MGMT_POOL_OVERWRITE_ACL = 215,
18    DRPC_METHOD_MGMT_POOL_UPDATE_ACL = 216,
19    DRPC_METHOD_MGMT_POOL_DELETE_ACL = 217,
20    DRPC_METHOD_MGMT_PREP_SHUTDOWN   = 218,
21    DRPC_METHOD_MGMT_DEV_STATE_QUERY = 219,
22    DRPC_METHOD_MGMT_DEV_SET_FAULTY = 220,
23    DRPC_METHOD_MGMT_DEV_REPLACE    = 221,
24    DRPC_METHOD_MGMT_LIST_CONTAINERS = 222,
25    DRPC_METHOD_MGMT_POOL_QUERY     = 223,
26    DRPC_METHOD_MGMT_POOL_SET_PROP   = 224,
27    DRPC_METHOD_MGMT_PING_RANK      = 225,
28    DRPC_METHOD_MGMT_REINTEGRATE    = 226,
29    DRPC_METHOD_MGMT_CONT_SET_OWNER = 227,
30    DRPC_METHOD_MGMT_EXCLUDE        = 228,
31    DRPC_METHOD_MGMT_EXTEND         = 229,
32    DRPC_METHOD_MGMT_POOL_EVICT     = 230,
33    DRPC_METHOD_MGMT_DRAIN          = 231,
34    DRPC_METHOD_MGMT_GROUP_UPDATE   = 232,
35    DRPC_METHOD_MGMT_NOTIFY_EXIT    = 233,
36    DRPC_METHOD_MGMT_DEV_IDENTIFY   = 234,
37    DRPC_METHOD_MGMT_NOTIFY_POOL_CONNECT = 235,
38    DRPC_METHOD_MGMT_NOTIFY_POOL_DISCONNECT = 236,
39    DRPC_METHOD_MGMT_POOL_GET_PROP   = 237,
40    DRPC_METHOD_MGMT_SET_LOG_MASKS  = 238,
41    DRPC_METHOD_MGMT_POOL_HEALTH     = 239,
42    DRPC_METHOD_MGMT_POOL_DUMP       = 240,
43    DRPC_METHOD_MGT_TGT_OBJNUM_GET   = 241,
44    DRPC_METHOD_MGMT_ENGINE_CMD      = 242,
45    DRPC_METHOD_MGMT_DYCONFIG_SET    = 243,
```

```
46     DRPC_METHOD_MGMT_DYCONFIG_GET           = 244,
47     DRPC_METHOD_MGMTFAULTMAP_DEL = 245,
48     DRPC_METHOD_MGMT_SET_MAINTENANCE_FLAG = 246,
49     DRPC_METHOD_MGMT_PROFILE               = 247,
50     DRPC_METHOD_MGMT_PROFILE_START        = 248,
51     DRPC_METHOD_MGMT_PROFILE_STOP         = 249,
52     DRPC_METHOD_MGMT_PROFILE_CLEAR        = 250,
53     DRPC_METHOD_MGMT_PROFILE_DUMP         = 251,
54     DRPC_METHOD_MGMT_POOL_CEA_VER_SET    = 252,
55     DRPC_METHOD_MGMT_POOL_QUERY_IO_STATS = 253,
56     DRPC_METHOD_MGMT_STORAGE_QUERY_IO_STATS = 254,
57     NUM_DRPC_MGMT_METHODS                /* Must be last */
58 };
```

C | 复制代码

```
1 //service 模块函数ID
2 enum drpc_srv_method {
3     DRPC_METHOD_SRV_NOTIFY_READY      = 301,
4     DRPC_METHOD_SRV_BIO_ERR          = 302,
5     DRPC_METHOD_SRV_GET_POOL_SVC    = 303,
6     DRPC_METHOD_SRV_CLUSTER_EVENT   = 304,
7     DRPC_METHOD_SRV_POOL_FIND_BYLABEL = 305,
8     DRPC_METHOD_SRV_GET_DEV_SN       = 306,
9
10    NUM_DRPC_SRV_METHODS           /* Must be last */
11};
```

C | 复制代码

```
1 // 安全验证模块函数 ID
2 enum drpc_sec_method {
3     DRPC_METHOD_SEC_VALIDATE_CREDS   = 401,
4
5     NUM_DRPC_SEC_METHODS           /* Must be last */
6};
```

统一 Execute 入口

```
func (cmd *PoolCreateCmd) Execute(args []string) error  
(src\control\cmd\dmg\pool.go)
```

DPRC调用

```
func (cmd *PoolCreateCmd) CreateInnerPool(ctx context.Context, oreq  
    *control.PoolCreateReq) error  
{  
    ...  
    err = control.PoolCreate(ctx, cmd.ctlInvoker, req)  
}  
(src\control\cmd\dmg\pool.go)
```

Server

server的入口地址 `src\control\server\server.go` 文件中 `Start` 函数。

server struct

server

server的结构体包含容器状态和Daos server的各个组件

C | 复制代码

```
1 type server struct {
2     log          *logging.LeveledLogger
3     cfg          *config.Server
4     hostname    string
5     runningUser string
6     faultDomain *system.FaultDomain
7     ctlAddr     *net.TCPAddr
8     netDevClass uint32
9     listener    net.Listener
10
11    harness      *EngineHarness
12    membership   *system.Membership
13    sysdb        *system.Database
14    pubSub       *events.PubSub
15    evtForwarder *control.EventForwarder
16    evtLogger    *control.EventLogger
17    ctlSvc       *ControlService           //ControlService实现控制平面控制服务
18    mgmtSvc     *mgmtSvc                  //mgmtSvc 实现Go部分管理服务
19    grpcServer   *grpc.Server
20
21    cbLock       sync.Mutex
22    onEnginesStarted []func(context.Context) error
23    onShutdown    []func()
24 }
```

ControlServer

ControlService 实现控制平面控制服务

C | 复制代码

```
1 type ControlService struct {
2     ctlpb.UnimplementedCtlSvcServer
3     StorageControlService
4     membership *system.Membership
5     rpcClient  control.UnaryInvoker
6     harness    *EngineHarness
7     srvCfg     *config.Server
8     events     *events.PubSub
9     sysdb      *system.Database
10    ctx         *context.Context
11    grpcServer *grpc.Server
12 }
```

mgmSvc

mgmSvc实施 (Go部分) 管理服务模块, `src\control\cmd\daos_agent\mgmt_rpc.go`

```
1 - type mgmSvc struct {
2     cfg *config.Server
3     mgmtpb.UnimplementedMgmtSvcServer
4     log           logging.Logger
5     harness       *EngineHarness
6     membership    *system.Membership // if MS leader, system membershi
7     p list
8     sysdb         *system.Database
9     rpcClient     control.UnaryInvoker
10    events        *events.PubSub
11    clientNetworkHint *mgmtpb.ClientNetHint
12    joinReqs      joinReqChan
13    groupUpdateReqs chan bool
14    lastMapVer    uint32
15    ctx           *context.Context
16    grpcServer    *grpc.Server
17    srvCfg        *config.Server
18 }
```

srvModule

srvModule代表daos_server dRPC服务发送模块。它处理daos_engine (src/engine) 发送的DRPC。

`src\control\server\mgmt_drpc.go`

```
1 - type srvModule struct {
2     log      logging.Logger
3     sysdb   poolResolver
4     engines []Engine
5     events  *events.PubSub
6 }
```

securityModule

SecurityModule是安全模块, `src\control\server\security_rpc.go`

Go | 复制代码

```
1 // SecurityModule is the security drpc module struct
2 type SecurityModule struct {
3     log    logging.Logger
4     config *security.TransportConfig
5 }
```

server func

入口函数 Start

Go | 复制代码

```
1 func Start(log *logging.LeveledLogger, cfg *config.Server) error {
2     faultDomain, err := processConfig(log, cfg)
3     if err != nil {
4         return err
5     }
6
7     ctx, shutdown := context.WithCancel(context.Background())
8     defer shutdown()
9
10    srv, err := newServer(ctx, log, cfg, faultDomain)
11
12
13    if err := srv.createServices(ctx); err != nil {
14        return err
15    }
16
17    if err := srv.initNetwork(ctx); err != nil {
18        return err
19    }
20
21    if err := srv.initStorage(); err != nil {
22        return err
23    }
24
25    if err := srv.addEngines(ctx); err != nil {
26        return err
27    }
28
29    if err := srv.setupGrpc(); err != nil {
30        return err
31    }
32
33    srv.registerEvents()
34
35    return srv.start(ctx, shutdown)
36
37 }
```

Go | 复制代码

```
1 func (srv *server) start(ctx context.Context, shutdown context.CancelFunc) error {
2
3     go func() {
4         _ = srv.grpcServer.Serve(srv.listener) //开启监听
5     }()
6
7     ...
8     // 单个daos_server dRPC服务器，用于处理所有引擎请求
9     if err := drpcServerSetup(ctx, drpcSetupReq); err != nil {
10         return errors.WithMessage(err, "dRPC server setup")
11     }
12
13 }
```

```
1 // drpcServerSetup specifies socket path and starts drpc server.
2 func drpcServerSetup(ctx context.Context, req *drpcServerSetupReq) error {
3     // Clean up any previous execution's sockets before we create any new
4     // sockets
5     if err := drpcCleanup(req.sockDir); err != nil {
6         return err
7     }
8
9     sockPath := getDrpcServerSocketPath(req.sockDir)
10    drpcServer, err := drpc.NewDomainSocketServer(ctx, req.log, sockPath)
11    if err != nil {
12        return errors.Wrap(err, "unable to create socket server")
13    }
14
15    // Create and add our modules
16    drpcServer.RegisterRPCModule(NewSecurityModule(req.log, req.tc))
17    drpcServer.RegisterRPCModule(newMgmtModule())
18    drpcServer.RegisterRPCModule(newSrvModule(req.log, req.sysdb, req.engines,
19                                         req.events))
20    if err := drpcServer.Start(); err != nil {
21        return errors.Wrapf(err, "unable to start socket server on %s", so-
22                           ckPath)
23    }
24    return nil
25 }
```

