## Architecture Overview

#### Terminology

|  |  |
| --- | --- |
| Acronym | Expansion |
| ABT | Argobots |
| ACL | Access Control List |
| ACE | Access Control Entry |
| ACID | Atomicity, consistency, isolation, durability |
| BIO | Blob I/O |
| CART | Collective and RPC Transport |
| CGO | Go tools that enable creation of Go packages that call C code |
| CN | Compute Node |
| COTS | Commercial off-the-shelf |
| CPU | Central Processing Unit |
| Daemon | A process offering system-level resources. |
| DAOS | Distributed Asynchronous Object Storage |
| PMEM | Intel Optane Persistent Memory |
| DPDK | Data Plane Development Kit |
| dRPC | DAOS Remote Procedure Call |
| gRPC | gRPC Remote Procedure Calls |
| GURT | A common library of Gurt Useful Routines and Types |
| HLC | Hybrid Logical Clock |
| HLD | High-level Design |
| ISA-L | Intel Storage Acceleration Library |
| I/O | Input/Output |
| KV store | Key-Value store |
| libfabric | Open Fabrics Interface |
| Mercury | A user-space RPC library that can use libfabrics as a transport |
| MTBF | Mean Time Between Failures |
| NVM | Non-Volatile Memory |
| NVMe | Non-Volatile Memory express |
| OFI | Open Fabrics Interface |
| OS | Operating System |
| PM | Persistent Memory |
| PMDK | Persistent Memory Devevelopment Kit |
| RAFT | Raft is a consensus algorithm used to distribute state transitions among DAOS server nodes. |
| RAS | Reliability, Availability & Serviceability |
| RDB | Replicated Database, containing pool metadata and maintained across DAOS servers using the Raft algorithm. |
| RDMA/RMA | Remote (Direct) Memory Access |
| RDG | Redundancy Group |
| RPC | Remote Procedure Call |
| SCM | Storage-Class Memory |
| SWIM | Scalable Weakly-consistent Infection-style process group Membership |
| SPDK | Storage Performance Development Kit |
| SSD | Solid State Drive |
| SWIM | Scalable Weakly-consistent Infection-style process group Membership protocol |
| ULT | User Level Thread |
| UPI | Intel Ultra Path Interconnect |
| UUID | Universal Unique Identifier |
| VOS | Versioning Object Store |

#### Architecture

DAOS is an open-source software-defined scale-out object store that provides high bandwidth and high IOPS storage containers to applications and enables next-generation data-centric workflows combining simulation, data analytics, and machine learning.

DAOS 是一种开源软件定义横向扩展对象存储，可为应用程序提供高带宽和高 IOPS 存储容器，并支持结合模拟、数据分析和机器学习的下一代以数据为中心的工作流。

Unlike the traditional storage stacks that were primarily designed for rotating media, DAOS is architected from the ground up to exploit new NVM technologies and is extremely lightweight since it operates End-to-End (E2E) in user space with full OS bypass. DAOS offers a shift away from an I/O model designed for block-based and high-latency storage to one that inherently supports fine-grained data access and unlocks the performance of the next-generation storage technologies.

与主要为旋转媒体设计的传统存储堆栈不同，DAOS 从头开始构建以利用新的 NVM 技术，并且非常轻量级，因为它在用户空间中运行端到端 (E2E)，完全绕过操作系统。 DAOS 提供了一种从专为基于块和高延迟存储设计的 I/O 模型转变为一种本质上支持细粒度数据访问并释放下一代存储技术性能的模型。

Unlike traditional Burst Buffers, DAOS is a high-performant independent and fault-tolerant storage tier that does not rely on a third-party tier to manage metadata and data resilience.

与传统的 Burst Buffers 不同，DAOS 是一个高性能的独立容错存储层，不依赖第三方层来管理元数据和数据弹性。

##### DAOS Features

DAOS relies on OFI for low-latency communications and stores data on both storage-class memory (SCM) and NVMe storage. DAOS presents a native key-array-value storage interface that offers a unified storage model over which domain-specific data models are ported, such as HDF5, MPI-IO, and Apache Arrow. A POSIX I/O emulation layer implementing files and directories over the native DAOS API is also available.

DAOS 依靠 OFI 进行低延迟通信，并将数据存储在存储级内存 (SCM) 和 NVMe 存储上。 DAOS 提供了一个原生的键-数组-值存储接口，它提供了一个统一的存储模型，通过该模型可以移植特定领域的数据模型，例如 HDF5、MPI-IO 和 Apache Arrow。 还可以使用 POSIX I/O 仿真层，通过本机 DAOS API 实现文件和目录。

DAOS I/O operations are logged and then inserted into a persistent index maintained in SCM. Each I/O is tagged with a particular timestamp called epoch and is associated with a particular version of the dataset. No read-modify-write operations are performed internally. Write operations are non-destructive and not sensitive to alignment. Upon read request, the DAOS service walks through the persistent index and creates a complex scatter-gather Remote Direct Memory Access (RDMA) descriptor to reconstruct the data at the requested version directly in the buffer provided by the application.

DAOS I/O 操作被记录下来，然后插入到 SCM 中维护的持久索引中。 每个 I/O 都标有称为 epoch 的特定时间戳，并与数据集的特定版本相关联。 内部不执行读-修改-写操作。 写操作是非破坏性的并且对对齐不敏感。 根据读取请求，DAOS 服务遍历持久索引并创建一个复杂的分散-聚集远程直接内存访问 (RDMA) 描述符，以直接在应用程序提供的缓冲区中重建请求版本的数据。

The SCM storage is memory-mapped directly into the address space of the DAOS service that manages the persistent index via direct load/store. Depending on the I/O characteristics, the DAOS service can decide to store the I/O in either SCM or NVMe storage. As represented in Figure 2-1, latency-sensitive I/Os, like application metadata and byte-granular data, will typically be stored in the former, whereas checkpoints and bulk data will be stored in the latter. This approach allows DAOS to deliver the raw NVMe bandwidth for bulk data by streaming the data to NVMe storage and maintaining internal metadata index in SCM. The Persistent Memory Development Kit (PMDK) allows managing transactional access to SCM and the Storage Performance Development Kit (SPDK) enables user-space I/O to NVMe devices.

SCM 存储通过内存直接映射到 DAOS 服务的地址空间，该服务通过直接加载/存储来管理持久索引。 根据 I/O 特性，DAOS 服务可以决定将 I/O 存储在 SCM 或 NVMe 存储中。 如图 2-1 所示，对延迟敏感的 I/O，如应用程序元数据和字节粒度数据，通常存储在前者中，而检查点和批量数据将存储在后者中。 这种方法允许 DAOS 通过将数据流式传输到 NVMe 存储并在 SCM 中维护内部元数据索引来为批量数据提供原始 NVMe 带宽。 持久内存开发套件 (PMDK) 允许管理对 SCM 的事务访问，存储性能开发套件 (SPDK) 允许用户空间 I/O 到 NVMe 设备。

图示

描述已自动生成

DAOS aims at delivering:

* High throughput and IOPS at arbitrary alignment and size
* Fine-grained I/O operations with true zero-copy I/O to SCM
* Support for massively distributed NVM storage via scalable collective communications across the storage servers
* Non-blocking data and metadata operations to allow I/O and computation to overlap
* Advanced data placement taking into account fault domains
* Software-managed redundancy supporting both replication and erasure code with an online rebuild
* End-to-end data integrity
* Scalable distributed transactions with guaranteed data consistency and automated recovery
* Dataset snapshot
* Security framework to manage access control to storage pools
* Software-defined storage management to provision, configure, modify and monitor storage pools over COTS hardware
* Native support for Hierarchical Data Format (HDF)5, MPI-IO and POSIX namespace over the DAOS data model
* Tools for disaster recovery
* Seamless integration with the Lustre parallel filesystem
* Mover agent to migrate datasets among DAOS pools and from parallel filesystems to DAOS and vice versa

DAOS旨在提供：

任意对齐和大小的高吞吐量和IOPS

具有真正零拷贝I/O到SCM的细粒度I/O操作

通过跨存储服务器的可扩展集体通信支持大规模分布式NVM 存储

允许I/O和计算重叠的非阻塞数据和元数据操作

考虑到故障域的高级数据放置 软件管理的冗余支持通过在线重建复制和纠删码

端到端数据完整性 具有保证数据一致性和自动恢复的可扩展分布式事务

数据集快照 用于管理对存储池的访问控制的安全框架

软件定义的存储管理，用于通过COTS硬件供应、配置、修改和监控存储池

对DAOS数据模型上的分层数据格式 (HDF)5、MPI-IO 和 POSIX 命名空间的本机支持

灾难恢复工具

与Lustre 并行文件系统无缝集成

Mover代理用于在 DAOS 池之间以及从并行文件系统到 DAOS 之间迁移数据集，反之亦然

##### DAOS System

A data center may have hundreds of thousands of compute instances interconnected via a scalable high-performance network, where all, or a subset of the instances called storage nodes, have direct access to NVM storage. A DAOS installation involves several components that can be either collocated or distributed.

一个数据中心可能有数十万个计算实例通过一个可扩展的高性能网络互连，其中所有实例或称为存储节点的实例子集都可以直接访问NVM存储。DAOS安装涉及多个组件，这些组件既可以并置，也可以分布.

A DAOS system is identified by a system name, and consists of a set of DAOS storage nodes connected to the same network. The DAOS storage nodes run one DAOS server instance per node, which in turn starts one DAOS Engine process per physical socket. Membership of the DAOS servers is recorded into the system map, that assigns a unique integer rank to each Engine process. Two different DAOS systems comprise two disjoint sets of DAOS servers, and do not coordinate with each other.

DAOS 系统由系统名称标识，由一组连接到同一网络的 DAOS 存储节点组成。 DAOS 存储节点为每个节点运行一个 DAOS server实例，然后每个物理套接字启动一个 DAOS 引擎进程。 DAOS server的成员资格被记录到系统映射中，它为每个引擎进程分配一个唯一的整数等级。 两个不同的 DAOS 系统包含两组不相交的 DAOS server，并且彼此不协调。

The DAOS server is a multi-tenant daemon running on a Linux instance (either natively on the physical node or in a VM or container) of each storage node. Its Engine sub-processes export the locally-attached SCM and NVM storage through the network. It listens to a management port (addressed by an IP address and a TCP port number), plus one or more fabric endpoints (addressed by network URIs). The DAOS server is configured through a YAML file in /etc/daos, including the configuration of its Engine sub-processes. The DAOS server startup can be integrated with different daemon management or orchestration frameworks (for example a systemd script, a Kubernetes service, or even via a parallel launcher like pdsh or srun).

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（本地在物理节点上或在 VM 或容器中）上。 其引擎子进程通过网络导出本地连接的 SCM 和 NVM 存储。 它侦听管理端口（由 IP 地址和 TCP 端口号寻址）以及一个或多个结构端点（由网络 URI 寻址）。 DAOS 服务器通过 /etc/daos 中的 YAML 文件进行配置，包括其 Engine 子进程的配置。 DAOS 服务器启动可以与不同的守护进程管理或编排框架（例如 systemd 脚本、Kubernetes 服务，甚至通过 pdsh 或 srun 等并行启动器）集成。

Inside a DAOS Engine, the storage is statically partitioned across multiple targets to optimize concurrency. To avoid contention, each target has its private storage, its own pool of service threads, and its dedicated network context that can be directly addressed over the fabric independently of the other targets hosted on the same storage node. The SCM modules are typically configured in AppDirect interleaved mode. They are thus presented to the operating system as a single PMEM namespace per socket (in fsdax mode). When N targets per engine are configured, each target is using 1/N of the capacity of the fsdax SCM capacity of that socket, independently of the other targets. Each target is also using a fraction of the NVMe capacity of the NVMe drives that are attached to this socket.

在 DAOS 引擎中，存储在多个target之间静态分区以优化并发性。 为避免争用，每个target都有自己的私有存储、自己的服务线程池和专用网络上下文，可以独立于同一存储节点上托管的其他target，通过结构直接寻址。 SCM 模块通常配置为 AppDirect 交错模式。 因此，它们作为每个套接字的单个 PMEM 命名空间（在 fsdax 模式下）呈现给操作系统。 当每个引擎配置 N 个target时，每个target使用该套接字 fsdax SCM 容量的 1/N，独立于其他target。 每个target还使用连接到该插槽的 NVMe 驱动器的 NVMe 容量的一小部分。

A target does not implement any internal data protection mechanism against storage media failure. As a result, a target is a single point of failure and the unit of fault. A dynamic state is associated with each target: Its state can be either "up and running", or "down and not available".

target没有实现任何针对存储介质故障的内部数据保护机制。 因此，target是单点故障和故障单元。 动态状态与每个target相关联：其状态可以是“启动并运行”或“关闭且不可用”。

A target is the unit of performance. Hardware components associated with the target, such as the backend storage medium, the CPU core(s), and the network, have limited capability and capacity.

target是绩效的单位。 与target相关联的硬件组件，例如后端存储介质、CPU 内核和网络，具有有限的能力和容量。

The number of targets exported by a DAOS Engine instance is configurable, and depends on the underlying hardware (in particular, the number of SCM modules and the number of NVMe SSDs that are served by this engine instance). As a best practice, the number of targets of an engine should be an integer multiple of the number of NVMe drives that are served by this engine.

DAOS 引擎实例导出的target数量是可配置的，并且取决于底层硬件（特别是该引擎实例所服务的 SCM 模块数量和 NVMe SSD 数量）。 作为最佳实践，引擎的target数量应该是该引擎所服务的 NVMe 驱动器数量的整数倍。

##### SDK and Tools

Applications, users, and administrators can interact with a DAOS system through two different client APIs. The management API offers the ability to administrate a DAOS system. It is intended to be integrated with different vendor-specific storage management or open-source orchestration frameworks. The dmg CLI tool is built over the DAOS management API. On the other hand, the DAOS library (libdaos) implements the DAOS storage model. It is primarily targeted at application and I/O middleware developers who want to store datasets in a DAOS system. User utilities like the daos command are also built over the API to allow users to manage datasets from a CLI.

Applications can access datasets stored in DAOS either directly through the native DAOS API, through an I/O middleware library (e.g. POSIX emulation, MPI-IO, HDF5) or through frameworks like Spark or TensorFlow that have already been integrated with the native DAOS storage model.

应用程序、用户和管理员可以通过两个不同的客户端 API 与 DAOS 系统交互。 管理 API 提供了管理 DAOS 系统的能力。 它旨在与不同供应商特定的存储管理或开源编排框架集成。 dmg CLI 工具是基于 DAOS 管理 API 构建的。 另一方面，DAOS 库（libdaos）实现了 DAOS 存储模型。 它主要面向希望在 DAOS 系统中存储数据集的应用程序和 I/O 中间件开发人员。 daos 命令等用户实用程序也是基于 API 构建的，以允许用户从 CLI 管理数据集。 应用程序可以直接通过原生 DAOS API、I/O 中间件库（例如 POSIX 仿真、MPI-IO、HDF5）或通过 Spark 或 TensorFlow 等已经与原生 DAOS 存储集成的框架访问存储在 DAOS 中的数据集 模型。

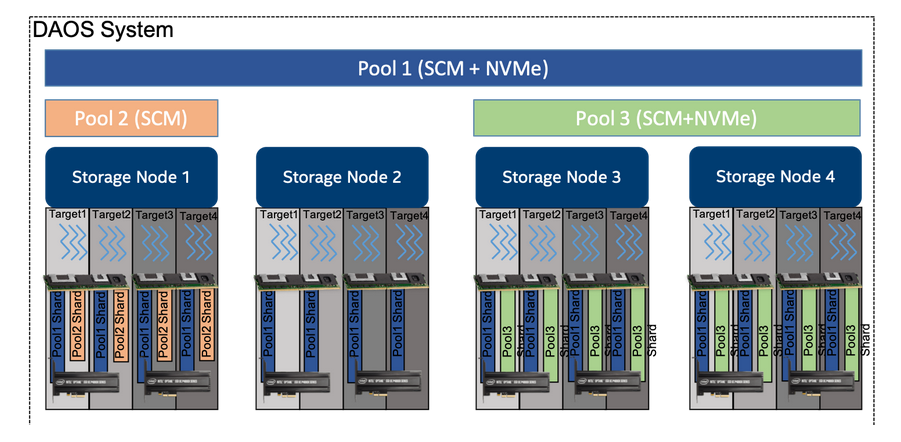
##### Agent

The DAOS agent is a daemon residing on the client nodes that interacts with the DAOS library to authenticate the application processes. It is a trusted entity that can sign the DAOS library credentials using certificates. The agent can support different authentication frameworks, and uses a Unix Domain Socket to communicate with the DAOS library.

DAOS 代理是驻留在客户端节点上的守护进程，它与 DAOS 库交互以验证应用程序进程。 它是一个可信实体，可以使用证书签署 DAOS 库凭据。 代理可以支持不同的身份验证框架，并使用 Unix Domain Socket 与 DAOS 库进行通信。

#### Storage Model

The [figure](https://docs.daos.io/overview/storage/#f4.1) below represents the fundamental abstractions of the DAOS storage model.



A DAOS pool is a storage reservation distributed across a collection of targets. The actual space allocated to the pool on each target is called a pool shard. The total space allocated to a pool is decided at creation time. It can be expanded over time by resizing all the pool shards (within the limit of the storage capacity dedicated to each target), or by spanning more targets (adding more pool shards). A pool offers storage virtualization and is the unit of provisioning and isolation. DAOS pools cannot span across multiple systems.

DAOS 池是分布在target集合中的存储预留。 每个target上分配给池的实际空间称为池分片。 分配给池的总空间在创建时决定。 它可以通过调整所有池分片的大小（在专用于每个target的存储容量的限制内）或跨越更多target（添加更多池分片）来随着时间的推移进行扩展。 池提供存储虚拟化，是配置和隔离的单元。 DAOS 池不能跨越多个系统

A pool can host multiple transactional object stores called DAOS containers. Each container is a private object address space, which can be modified transactionally and independently of the other containers stored in the same pool. A container is the unit of snapshot and data management. DAOS objects belonging to a container can be distributed across any target of the pool for both performance and resilience and can be accessed through different APIs to represent structured, semi-structured and unstructured data efficiently

一个池可以托管多个称为 DAOS 容器的事务对象存储。 每个容器都是一个私有对象地址空间，可以事务性地修改，并且独立于存储在同一池中的其他容器。 容器是快照和数据管理的单元。 属于容器的 DAOS 对象可以分布在池的任何target上以获得性能和弹性，并且可以通过不同的 API 访问以有效地表示结构化、半结构化和非结构化数据

The table below shows the targeted level of scalability for each DAOS concept.

下表显示了每个 DAOS 概念的target可扩展性级别

|  |  |
| --- | --- |
| **DAOS Concept** | **Scalability (Order of Magnitude)** |
| System | 105 Servers (hundreds of thousands) and 102 Pools (hundreds) |
| Server | 101 Targets (tens) |
| Pool | 102 Containers (hundreds) |
| Container | 109 Objects (billions) |

##### DAOS Pool

A pool is identified by a unique pool UUID and maintains target memberships in a persistent versioned list called the pool map. The membership is definitive and consistent, and membership changes are sequentially numbered. The pool map not only records the list of active targets, it also contains the storage topology in the form of a tree that is used to identify targets sharing common hardware components. For instance, the first level of the tree can represent targets sharing the same motherboard, and then the second level can represent all motherboards sharing the same rack and finally the third level can represent all racks in the same cage. This framework effectively represents hierarchical fault domains, which are then used to avoid placing redundant data on targets subject to correlated failures. At any point in time, new targets can be added to the pool map, and failed targets can be excluded. Moreover, the pool map is fully versioned, which effectively assigns a unique sequence to each modification of the map, particularly for failed node removal.

池由唯一的池 UUID 标识，并在称为池映射的持久版本化列表中维护target成员资格。成员资格是确定的和一致的，成员资格变化按顺序编号。池映射不仅记录活动target的列表，还包含树形式的存储拓扑，用于识别共享公共硬件组件的target。例如，树的第一层可以表示共享同一主板的target，然后第二层可以表示共享同一机架的所有主板，最后第三层可以表示同一机箱中的所有机架。该框架有效地表示了分层故障域，然后用于避免在受相关故障影响的target上放置冗余数据。在任何时间点，都可以将新target添加到池映射中，并且可以排除失败的target。此外，池映射是完全版本化的，这有效地为映射的每次修改分配了一个唯一的序列，特别是对于失败的节点删除。

A pool shard is a reservation of persistent memory optionally combined with a pre-allocated space on NVMe storage on a specific target. It has a fixed capacity and fails operations when full. Current space usage can be queried at any time and reports the total amount of bytes used by any data type stored in the pool shard.

池分片是持久内存的保留，可选择与特定target上 NVMe 存储上的预分配空间相结合。它具有固定的容量，并且在满时失败。可以随时查询当前空间使用情况，并报告存储在池分片中的任何数据类型使用的总字节数。

Upon target failure and exclusion from the pool map, data redundancy inside the pool is automatically restored online. This self-healing process is known as rebuild. Rebuild progress is recorded regularly in special logs in the pool stored in persistent memory to address cascading failures. When new targets are added, data is automatically migrated to the newly added targets to redistribute space usage equally among all the members. This process is known as space rebalancing and uses dedicated persistent logs as well to support interruption and restart. A pool is a set of targets spread across different storage nodes over which data and metadata are distributed to achieve horizontal scalability, and replicated or erasure-coded to ensure durability and availability.

在target失败并从池映射中排除时，池内的数据冗余会自动在线恢复。这种自我修复过程称为重建。重建进度定期记录在存储在持久内存中的池中的特殊日志中，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，以在所有成员之间平均重新分配空间使用量。此过程称为空间重新平衡，并使用专用的持久日志来支持中断和重新启动。池是分布在不同存储节点上的一组target，数据和元数据分布在这些节点上以实现水平可扩展性，并进行复制或EC编码以确保持久性和可用性。

When creating a pool, a set of system properties must be defined to configure the different features supported by the pool. Also, users can define their attributes that will be stored persistently.

创建池时，必须定义一组系统属性来配置池支持的不同功能。 此外，用户可以定义将永久存储的属性。

A pool is only accessible to authenticated and authorized applications. Multiple security frameworks could be supported, from NFSv4 access control lists to third party-based authentication (such as Kerberos). Security is enforced when connecting to the pool. Upon successful connection to the pool, a connection context is returned to the application process.

只有经过身份验证和授权的应用程序才能访问池。 可以支持多种安全框架，从 NFSv4 访问控制列表到基于第三方的身份验证（例如 Kerberos）。 连接到池时会强制执行安全性。 成功连接到池后，连接上下文将返回给应用程序进程。

As detailed previously, a pool stores many different sorts of persistent metadata, such as the pool map, authentication and authorization information, user attributes, properties, and rebuild logs. Such metadata is critical and requires the highest level of resiliency. Therefore, the pool metadata is replicated on a few nodes from distinct high-level fault domains. For very large configurations with hundreds of thousands of storage nodes, only a very small fraction of those nodes (in the order of tens) run the pool metadata service. With a limited number of storage nodes, DAOS can afford to rely on a consensus algorithm to reach agreement, to guarantee consistency in the presence of faults, and to avoid the split-brain syndrome.

如前所述，池存储多种不同类型的持久元数据，例如池映射、身份验证和授权信息、用户属性、属性和重建日志。 此类元数据至关重要，需要最高级别的弹性。 因此，池元数据被复制到来自不同高级故障域的几个节点上。 对于具有数十万个存储节点的超大型配置，这些节点中只有很小一部分（大约数十个）运行池元数据服务。 由于存储节点数量有限，DAOS 可以依靠共识算法达成一致，在出现故障时保证一致性，并避免裂脑综合症。

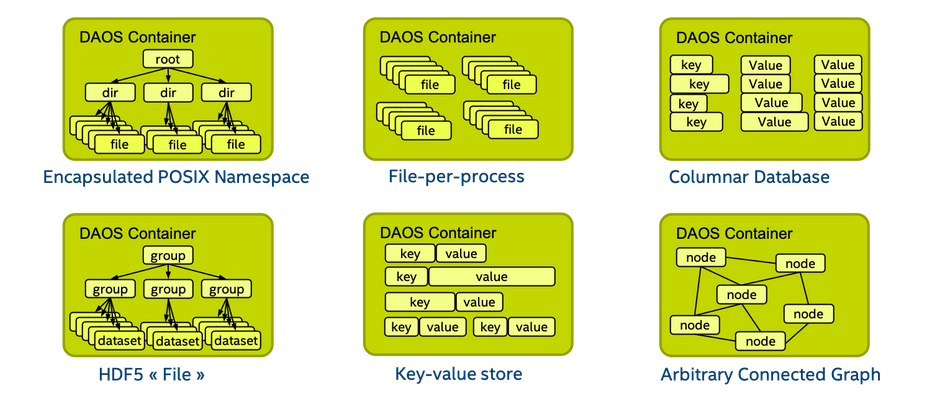
To access a pool, a user process should connect to this pool and pass the security checks. Once granted, a pool connection can be shared (via local2global() and global2local() operations) with any or all of its peer application processes (similar to the openg() POSIX extension). This collective connect mechanism helps to avoid metadata request storm when a massively distributed job is run on the datacenter. A pool connection is revoked when the original process that issued the connection request disconnects from the pool.

要访问一个池，用户进程应该连接到这个池并通过安全检查。 一旦被授予，池连接就可以与其任何或所有对等应用程序进程（类似于 openg() POSIX 扩展）共享（通过 local2global() 和 global2local() 操作）。 当在数据中心上运行大规模分布式作业时，这种集体连接机制有助于避免元数据请求风暴。 当发出连接请求的原始进程与池断开连接时，池连接将被撤销。

##### DAOS Container

A container represents an object address space inside a pool and is identified by a container UUID. The diagram below represents how the user (I/O middleware, domain-specific data format, big data or AI framework, ...) could use the container concept to store related datasets.

容器表示池内的对象地址空间，由容器 UUID 标识。 下图表示用户（I/O 中间件、特定领域数据格式、大数据或 AI 框架等）如何使用容器概念来存储相关数据集。



Like pools, containers can store user attributes. A set of properties must be passed at container creation time to configure different features like checksums.

与池一样，容器可以存储用户属性。 必须在容器创建时传递一组属性以配置不同的功能，例如校验和。

To access a container, an application must first connect to the pool and then open the container. If the application is authorized to access the container, a container handle is returned. This includes capabilities that authorize any process in the application to access the container and its contents. The opening process may share this handle with any or all of its peers. Their capabilities are revoked on container close.

要访问容器，应用程序必须首先连接到池，然后打开容器。 如果应用程序被授权访问容器，则返回容器句柄。 这包括授权应用程序中的任何进程访问容器及其内容的功能。 打开过程可以与其任何或所有对等体共享该句柄。 他们的能力在容器关闭时被撤销。

Objects in a container may have different schemas for data distribution and redundancy over targets. Dynamic or static striping, replication, or erasure code are some parameters required to define the object schema. The object class defines common schema attributes for a set of objects. Each object class is assigned a unique identifier and is associated with a given schema at the pool level. A new object class can be defined at any time with a configurable schema, which is then immutable after creation (or at least until all objects belonging to the class have been destroyed). For convenience, several object classes that are expected to be the most commonly used will be predefined by default when the pool is created, as shown in the [table](https://docs.daos.io/overview/storage/#t4.2) below.

容器中的对象可能具有不同的模式用于数据分布和target上的冗余。 动态或静态条带化、复制或EC是定义对象模式所需的一些参数。 对象类定义一组对象的通用模式属性。 每个对象类都被分配了一个唯一标识符，并与池级别的给定模式相关联。 可以随时使用可配置模式定义新的对象类，该模式在创建后是不可变的（或至少在属于该类的所有对象都被销毁之前）。 为方便起见，在创建池时将默认预定义几个最常用的对象类，如下表所示。

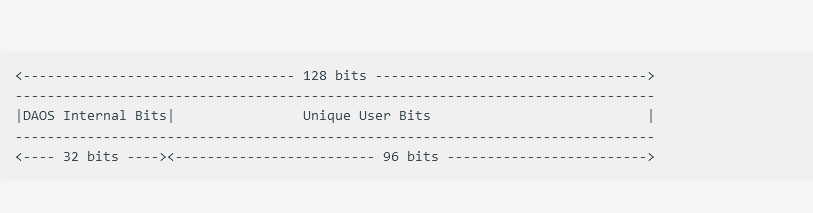
**Sample of Pre-defined Object Classes**

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

As shown below, each object is identified in the container by a unique 128-bit object address. The high 32 bits of the object address is reserved for DAOS to encode internal metadata such as the object class. The remaining 96 bits are managed by the user and should be unique inside the container. Those bits can be used by upper layers of the stack to encode their metadata, as long as unicity is guaranteed. A per-container 64-bit scalable object ID allocator is provided in the DAOS API. The object ID to be stored by the application is the full 128-bit address, which is for single use only and can be associated with only a single object schema.

如下所示，容器中的每个对象都由唯一的 128 位对象地址标识。 对象地址的高 32 位保留给 DAOS 用于编码内部元数据，例如对象类。 剩余的 96 位由用户管理，并且在容器内应该是唯一的。 只要保证唯一性，堆栈的上层就可以使用这些位来对其元数据进行编码。 DAOS API 中提供了每个容器的 64 位可扩展对象 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。

**DAOS Object ID Structure**



A container is the basic unit of transaction and versioning. All object operations are implicitly tagged by the DAOS library with a timestamp called an epoch. The DAOS transaction API allows to combine multiple object updates into a single atomic transaction, with multi-version concurrency control based on epoch ordering. All the versioned updates may be periodically aggregated, to reclaim space utilized by overlapping writes and to reduce metadata complexity. A snapshot is a permanent reference that can be placed on a specific epoch to prevent aggregation.

容器是事务和版本控制的基本单元。 所有对象操作都由 DAOS 库使用称为epoch的时间戳隐式标记。 DAOS 事务 API 允许将多个对象更新组合到一个原子事务中，并具有基于epoch排序的多版本并发控制。 所有版本更新都可以定期聚合，以回收重叠写入所使用的空间并降低元数据的复杂性。 快照是一个永久引用，可以放置在特定时期以防止聚合。

Container metadata (list of snapshots, container open handles, object class, user attributes, properties, and others) are stored in persistent memory and maintained by a dedicated container metadata service that either uses the same replicated engine as the parent metadata pool service, or has its own engine. This is configurable when creating a container.

容器元数据（快照列表、容器打开句柄、对象类、用户属性、属性等）存储在持久内存中，并由专用容器元数据服务维护，该服务使用与父元数据池服务相同的复制引擎，或者 有自己的引擎。 这在创建容器时是可配置的。

Like a pool, access to a container is controlled by the container handle. To acquire a valid handle, an application process must open the container and pass the security checks. This container handle may then be shared with other peer application processes via the container local2global() and global2local() operations.

与池一样，对容器的访问由容器句柄控制。 要获得有效句柄，应用程序进程必须打开容器并通过安全检查。 然后可以通过容器 local2global() 和 global2local() 操作与其他对等应用程序进程共享此容器句柄。

##### DAOS Object

To avoid scaling problems and overhead common to a traditional storage system, DAOS objects are intentionally simple. No default object metadata beyond the type and schema is provided. This means that the system does not maintain time, size, owner, permissions or even track openers. To achieve high availability and horizontal scalability, many object schemas (replication/erasure code, static/dynamic striping, and others) are provided. The schema framework is flexible and easily expandable to allow for new custom schema types in the future. The layout is generated algorithmically on object open from the object identifier and the pool map. End-to-end integrity is assured by protecting object data with checksums during network transfer and storage.

为了避免传统存储系统常见的扩展问题和开销，DAOS 对象有意简化。 没有提供超出类型和架构的默认对象元数据。 这意味着系统不维护时间、大小、所有者、权限甚至跟踪开启者。 为了实现高可用性和水平可伸缩性，提供了许多对象模式（复制/EC、静态/动态条带化等）。 架构框架灵活且易于扩展，以允许将来使用新的自定义架构类型。 分布是根据对象标识符和池映射在对象打开时通过算法生成的。 通过在网络传输和存储期间使用校验和保护对象数据来确保端到端完整性。

A DAOS object can be accessed through different APIs:

* **Multi-level key-array** API is the native object interface with locality feature. The key is split into a distribution (dkey) and an attribute (akey) key. Both the dkey and akey can be of variable length and type (a string, an integer or even a complex data structure). All entries under the same dkey are guaranteed to be collocated on the same target. The value associated with akey can be either a single variable-length value that cannot be partially overwritten, or an array of fixed-length values. Both the akeys and dkeys support enumeration.
* **Key-value** API provides a simple key and variable-length value interface. It supports the traditional put, get, remove and list operations.
* **Array API** implements a one-dimensional array of fixed-size elements addressed by a 64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.

DAOS 对象可以通过不同的 API 访问：

• 多级键数组API 是具有局部性特征的本机对象接口。 密钥分为分布（dkey）和属性（akey）密钥。 dkey 和 akey 都可以是可变长度和类型（字符串、整数甚至复杂的数据结构）。 同一个 dkey 下的所有条目都保证被配置在同一个target上。 与 akey 关联的值可以是无法部分覆盖的单个可变长度值，也可以是固定长度值的数组。 akeys 和 dkeys 都支持枚举。

• Key-value API 提供了一个简单的键和可变长度的值接口。 它支持传统的 put、get、remove 和 list 操作。

• Array API 实现了一个由64 位偏移量寻址的固定大小元素的一维数组。 DAOS 数组支持任意范围读取、写入和打孔操作。

#### Transaction Model

The DAOS API supports distributed transactions that allow any update operations against objects belonging to the same container to be combined into a single ACID transaction. Distributed consistency is provided via a lockless optimistic concurrency control mechanism based on multi-version timestamp ordering. DAOS transactions are serializable and can be used on an ad-hoc basis for parts of the datasets that need it.

DAOS API 支持分布式事务，允许对属于同一容器的对象的任何更新操作合并到单个 ACID 事务中。 通过基于多版本时间戳排序的无锁乐观并发控制机制提供分布式一致性。 DAOS 事务是可序列化的，可以临时用于需要它的部分数据集。

The DAOS versioning mechanism allows creating persistent container snapshots which provide point-in-time distributed consistent views of a container which can be used to build producer-consumer pipeline.

DAOS 版本控制机制允许创建持久的容器快照，这些快照提供容器的时间点分布式一致视图，可用于构建生产者-消费者管道。

##### Epoch and Timestamp Ordering

Each DAOS I/O operation is tagged with a timestamp called epoch. An epoch is a 64-bit integer that integrates both logical and physical clocks (see [HLC paper](https://cse.buffalo.edu/tech-reports/2014-04.pdf)). The DAOS API provides helper functions to convert an epoch to traditional POSIX time (i.e., struct timespec, see clock\_gettime(3)).

每个 DAOS I/O 操作都标有一个称为 epoch 的时间戳。 epoch是一个 64 位整数，它集成了逻辑时钟和物理时钟（参见 HLC 论文）。 DAOS API 提供了帮助函数来将epoch转换为传统的 POSIX 时间（即 struct timespec，请参阅 clock\_gettime(3)）。

##### Container Snapshot

As shown in the [figure](https://docs.daos.io/overview/transaction/#f4.4) below, the content of a container can be snapshot at any time.

如下图所示，容器的内容可以随时进行快照。

日程表

描述已自动生成

DAOS snapshots are very lightweight and are tagged with the epoch associated with the time when the snapshot was created. Once successfully created, a snapshot remains readable until it is explicitly destroyed. The content of a container can be rolled back to a particular snapshot.

DAOS 快照非常轻量级，并标有与创建快照的时间相关联的epoch。 一旦成功创建，快照将保持可读，直到它被显式销毁。 容器的内容可以回滚到特定的快照。

The container snapshot feature allows supporting native producer/consumer pipelines as represented in the diagram below.

容器快照功能允许支持本机生产者/消费者管道，如下图所示。

图示

描述已自动生成

The producer will generate a snapshot once a consistent version of the dataset has been successfully written. The consumer applications may subscribe to container snapshot events, so that new updates can be processed as the producer commits them. The immutability of the snapshots guarantees that the consumer sees consistent data, even while the producer continues with new updates. Both the producer and consumer indeed operate on different versions of the container and do not need any serialization. Once the producer generates a new version of the dataset, the consumer may query the differences between the two snapshots and process only the incremental changes.

一旦成功写入数据集的一致版本，生产者将生成快照。 消费者应用程序可以订阅容器快照事件，以便在生产者提交它们时可以处理新的更新。 快照的不变性保证消费者看到一致的数据，即使生产者继续进行新的更新。 生产者和消费者确实在不同版本的容器上运行，不需要任何序列化。 一旦生产者生成了新版本的数据集，消费者就可以查询两个快照之间的差异并只处理增量变化。

##### Distributed Transactions

Unlike POSIX, the DAOS API does not impose any worst-case concurrency control mechanism to address conflicting I/O operations. Instead, individual I/O operations are tagged with a different epoch and applied in epoch order, regardless of execution order. This baseline model delivers the maximum scalability and performance to data models and applications that do not generate conflicting I/O workload. Typical examples are collective MPI-IO operations, POSIX file read/write or HDF5 dataset read/write.

与 POSIX 不同，DAOS API 没有强加任何最坏情况并发控制机制来解决冲突的 I/O 操作。 取而代之的是，无论执行顺序如何，单个 I/O 操作都被标记为不同的epoch并按epoch顺序应用。 此基线模型为不会产生冲突 I/O 工作负载的数据模型和应用程序提供最大的可扩展性和性能。 典型示例是集体 MPI-IO 操作、POSIX 文件读/写或 HDF5 数据集读/写。

For parts of the data model that require conflict serialization, DAOS provides distributed serializable transaction based on multi-version concurrency control. Transactions are typically needed when different user processes can overwrite the value associated with a dkey/akey pair. Examples are a SQL database over DAOS or a consistent POSIX namespace accessed concurrently by uncoordinated clients. All I/O operations (including reads) submitted in the context of the same operation will use the same epoch. The DAOS transaction mechanism automatically detects the traditional read/write, write/read and write/write conflicts and aborts one of the conflicting transactions (the transaction fails to commit with -DER\_RESTART). The failed transaction then has to be restarted by the user/application.

对于需要冲突序列化的数据模型部分，DAOS 提供了基于多版本并发控制的分布式可序列化事务。 当不同的用户进程可以覆盖与 dkey/akey 对关联的值时，通常需要事务。 示例是基于 DAOS 的 SQL 数据库或由非协调客户端并发访问的一致 POSIX 命名空间。 在同一操作的上下文中提交的所有 I/O 操作（包括读取）将使用相同的 epoch。 DAOS 事务机制会自动检测传统的读/写、写/读和写/写冲突，并中止冲突的事务之一（事务提交失败，使用 -DER\_RESTART）。 失败的事务然后必须由用户/应用程序重新启动。

In the initial implementation, the transaction API has the following limitations that will be addressed in future DAOS versions:

* no support for the array API
* transactional object update and key-value put operations are not visible via object fetch/list and key-value get/list operations executed in the context of the same transaction.

在初始实现中，事务 API 具有以下限制，将在未来的 DAOS 版本中解决：

• 不支持数组 API

• 事务对象更新和键值放置操作通过在同一事务上下文中执行的对象获取/列表和键值获取/列表操作是不可见的。

#### Fault Model

DAOS relies on massively distributed single-ported storage. Each target is thus effectively a single point of failure. DAOS achieves availability and durability of both data and metadata by providing redundancy across targets in different fault domains. DAOS internal pool and container metadata are replicated via a robust consensus algorithm. DAOS objects are then safely replicated or erasure-coded by transparently leveraging the DAOS distributed transaction mechanisms internally. The purpose of this section is to provide details on how DAOS achieves fault tolerance and guarantees object resilience.

DAOS 依赖于大规模分布式单端口存储。 因此，每个target实际上都是单点故障。 DAOS 通过在不同故障域中的targe之间提供冗余来实现数据和元数据的可用性和持久性。 DAOS 内部池和容器元数据通过强大的共识算法进行复制。 然后通过在内部透明地利用 DAOS 分布式事务机制，安全地复制或EC DAOS 对象。 本节的目的是提供有关 DAOS 如何实现容错和保证对象弹性的详细信息。

##### Hierarchical Fault Domains

A fault domain is a set of servers sharing the same point of failure and which are thus likely to fail altogether. DAOS assumes that fault domains are hierarchical and do not overlap. The actual hierarchy and fault domain membership must be supplied by an external database used by DAOS to generate the pool map.

容错域是一组共享相同故障点的服务器，因此它们很可能完全失败。 DAOS 假定故障域是分层的并且不重叠。 实际的层次结构和容错域成员资格必须由 DAOS 用来生成池映射的外部数据库提供。

Pool metadata are replicated on several nodes from different high-level fault domains for high availability, whereas object data is replicated or erasure-coded over a variable number of fault domains depending on the selected object class.

池元数据在来自不同高级故障域的多个节点上复制以实现高可用性，而对象数据根据所选对象类别在可变数量的故障域上复制或EC。

##### Fault Detection

DAOS servers are monitored within a DAOS system through a gossip-based protocol called [SWIM](http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1028914) that provides accurate, efficient, and scalable server fault detection. Storage attached to each DAOS target is monitored through periodic local health assessment. Whenever a local storage I/O error is returned to the DAOS server, an internal health check procedure will be called automatically. This procedure will make an overall health assessment by analyzing the IO error code and device SMART/Health data. If the result is negative, the target will be marked as faulty, and further I/Os to this target will be rejected and re-routed.

DAOS 服务器在 DAOS 系统内通过称为 SWIM的协议进行监控，该协议提供准确、高效和可扩展的服务器故障检测。 附加到每个 DAOS target的存储通过定期的本地健康评估进行监控。 每当本地存储 I/O 错误返回到 DAOS 服务器时，都会自动调用内部健康检查程序。 此过程将通过分析 IO 错误代码和设备 SMART/健康数据来进行整体健康评估。 如果结果是否定的，target将被标记为有故障，并且对该target的进一步 I/O 将被拒绝并重新路由。

##### Fault Isolation

Once detected, the faulty target or servers (effectivelly a set of targets) must be excluded from the pool map. This process is triggered either manually by the administrator or automatically. Upon exclusion, the new version of the pool map is eagerly pushed to all storage targets. At this point, the pool enters a degraded mode that might require extra processing on access (e.g. reconstructing data out of erasure code). Consequently, DAOS client and storage nodes retry RPC indefinitely until they find an alternative replacement target from the new pool map. At this point, all outstanding communications with the evicted target are aborted, and no further messages should be sent to the target until it is explicitly reintegrated (possibly only after maintenance action).

一旦检测到，故障target或服务器（实际上是一组target）必须从池映射中排除。 此过程由管理员手动或自动触发。 排除后，新版本的池映射会急切地推送到所有存储target。 此时，池进入降级模式，可能需要额外的访问处理（例如，从纠删码中重建数据）。 因此，DAOS 客户端和存储节点会无限期地重试 RPC，直到它们从新的池映射中找到替代的target。 此时，与被驱逐target的所有未完成的通信都将中止，并且在明确重新集成之前不应向target发送进一步的消息（可能仅在维护操作之后）。

All storage targets are promptly notified of pool map changes by the pool service. This is not the case for client nodes, which are lazily informed of pool map invalidation each time they communicate with servers. To do so, clients pack in every RPC their current pool map version. Servers reply not only with the current pool map version. Consequently, when a DAOS client experiences RPC timeout, it regularly communicates with the other DAOS target to guarantee that its pool map is always current. Clients will then eventually be informed of the target exclusion and enter into degraded mode.

池服务会及时通知所有存储target池映射更改。 客户端节点的情况并非如此，它们每次与服务器通信时都会延迟通知池映射失效。 为此，客户端在每个 RPC 中打包其当前池映射版本。 服务器不仅回复当前池映射版本。 因此，当 DAOS 客户端遇到 RPC 超时时，它会定期与其他 DAOS target通信以确保其池映射始终是最新的。 然后客户端将最终被告知target排除并进入降级模式。

This mechanism guarantees global node eviction and that all nodes eventually share the same view of target aliveness.

这种机制保证了全局节点驱逐，并且所有节点最终共享相同的target活跃度视图。

##### Fault Recovery

Upon exclusion from the pool map, each target starts the rebuild process automatically to restore data redundancy. First, each target creates a list of local objects impacted by the target exclusion. This is done by scanning a local object table maintained by the underlying storage layer. Then for each impacted object, the location of the new object shard is determined and redundancy of the object restored for the whole history (i.e., snapshots). Once all impacted objects have been rebuilt, the pool map is updated a second time to report the target as failed out. This marks the end of collective rebuild process and the exit from degraded mode for this particular fault. At this point, the pool has fully recovered from the fault and client nodes can now read from the rebuilt object shards.

从池映射中排除后，每个target都会自动启动重建过程以恢复数据冗余。 首先，每个target创建一个受target排除影响的本地对象列表。 这是通过扫描底层存储层维护的本地对象表来完成的。 然后对于每个受影响的对象，确定新对象分片的位置，并为整个历史（即快照）恢复对象的冗余。 重建所有受影响的对象后，将再次更新池映射以将target报告为失败。 这标志着集体重建过程的结束以及此特定故障的降级模式退出。 此时，池已完全从故障中恢复，客户端节点现在可以从重建的对象分片中读取。

This rebuild process is executed online while applications continue accessing and updating objects.

当应用程序继续访问和更新对象时，此重建过程在线执行。

#### Security Model

DAOS uses a flexible security model that separates authentication from authorization. It is designed to have a minimal impact on the I/O path.

DAOS 使用灵活的安全模型，将身份验证与授权分开。 它旨在对 I/O 路径的影响最小。

DAOS does not provide any transport security for the fabric network used for I/O transfers. When deploying DAOS, the administrator is responsible for secure configuration of their specific fabric network. For RDMA over Ethernet, enabling IPsec is recommended. See the [RDMA protocol spec (RFC 5040)](https://tools.ietf.org/html/rfc5040#section-8.2) for more information.

DAOS 不为用于 I/O 传输的结构网络提供任何传输安全性。 部署 DAOS 时，管理员负责其特定结构网络的安全配置。 对于以太网上的 RDMA，建议启用 IPsec。 有关详细信息，请参阅 RDMA 协议规范 (RFC 5040)。

There are two areas where DAOS implements its own layer of security. At the user level, clients must be able to read and modify only pools and containers to which they have been granted access. At the system and administrative levels, only authorized components must be able to access the DAOS management network.

DAOS 在两个方面实现了自己的安全层。 在用户级别，客户端必须只能读取和修改他们已被授予访问权限的池和容器。 在系统和管理级别，只有经过授权的组件才能访问 DAOS 管理网络。

##### Authentication

There are different means of authentication, depending on whether the caller is accessing client resources or the DAOS management network.

根据调用者是访问客户端资源还是访问 DAOS 管理网络，有不同的身份验证方法。

##### Client Library

The client library libdaos is an untrusted component. The daos user-level command that uses the client library is also an untrusted component. A trusted process, the DAOS agent (daos\_agent), runs on each client node and authenticates the user processes.

客户端库 libdaos 是一个不受信任的组件。 使用客户端库的 daos 用户级命令也是一个不受信任的组件。 受信任的进程，即 DAOS 代理 (daos\_agent)，在每个客户端节点上运行并对用户进程进行身份验证。

The DAOS security model is designed to support different authentication methods for client processes. Currently, we support AUTH\_SYS authentication only.

DAOS 安全模型旨在支持客户端进程的不同身份验证方法。 目前，我们仅支持 AUTH\_SYS 身份验证。

##### DAOS Management Network

Each trusted DAOS component (daos\_server, daos\_agent, and the dmg administrative tool) is authenticated by means of a certificate generated for that component. These components identify one another over the DAOS management network via mutually-authenticated TLS.

每个受信任的 DAOS 组件（daos\_server、daos\_agent 和 dmg 管理工具）都通过为该组件生成的证书进行身份验证。 这些组件通过相互认证的 TLS 在 DAOS 管理网络上相互识别。

##### Authorization

Client authorization for resources is controlled by the Access Control List (ACL) on the resource. Authorization on the management network is achieved by settings on the [certificates](https://daos-stack.github.io/admin/deployment/#certificate-configuration) that are generated while setting up the DAOS system.

资源的客户端授权由资源上的访问控制列表 (ACL) 控制。 管理网络的授权是通过设置DAOS系统时生成的证书来实现的。

##### Component Certificates

Access to DAOS management RPCs is controlled via the CommonName (CN) set in each management component certificate. A given management RPC may only be invoked by a component which connects with the correct certificate.

通过在每个管理组件证书中设置的 CommonName (CN) 来控制对 DAOS 管理 RPC 的访问。 给定的管理 RPC 只能由与正确证书连接的组件调用。

##### Access Control Lists

Client access to resources like pools and containers is controlled by DAOS Access Control Lists (ACLs). These ACLs are derived in part from NFSv4 ACLs, and adapted for the unique needs of a distributed system.

客户端对池和容器等资源的访问由 DAOS 访问控制列表 (ACL) 控制。 这些 ACL 部分源自 NFSv4 ACL，并针对分布式系统的独特需求进行了调整。

The client may request read-only or read-write access to the resource. If the resource ACL doesn't grant them the requested access level, they won't be able to connect. While connected, their handle to that resource grants permissions for specific actions.

客户端可以请求对资源的只读或读写访问。 如果资源 ACL 没有授予他们请求的访问级别，他们将无法连接。 连接后，他们对该资源的处理授予特定操作的权限。

The permissions of a handle last for the duration of its existence, similar to an open file descriptor in a POSIX system. A handle cannot currently be revoked.

句柄的权限在其存在期间持续，类似于 POSIX 系统中的打开文件描述符。 当前无法撤销句柄。

#### Access Control Entries

In the input and output of DAOS tools, an Access Control Entry (ACE) is defined using a colon-separated string with the following format: TYPE:FLAGS:PRINCIPAL:PERMISSIONS

The contents of all the fields are case-sensitive.

在 DAOS 工具的输入和输出中，访问控制条目 (ACE) 使用以下格式的冒号分隔字符串定义：TYPE:FLAGS:PRINCIPAL:PERMISSIONS 所有字段的内容都区分大小写。

##### Type

The type of ACE entry (mandatory). Only one type of ACE is supported at this time.

* A (Allow): Allow access to the specified principal for the given permissions.

ACE 条目的类型（必填）。 目前仅支持一种类型的 ACE。

• A（允许）：允许对给定权限的指定主体进行访问。

##### Flags

The (optional) flags provide additional information about how the ACE should be interpreted.

* G (Group): The principal should be interpreted as a group.

（可选）标志提供有关如何解释 ACE 的附加信息。

• G（组）：应将主体解释为一个组。

##### Principal

The principal (also called the identity) is specified in the name@domain format. The domain should be left off if the name is a UNIX user/group on the local domain. Currently, this is the only case supported by DAOS.

主体（也称为身份）以 name@domain 格式指定。 如果名称是本地域上的 UNIX 用户/组，则应保留域。 目前，这是 DAOS 支持的唯一案例。

There are three special principals, OWNER@, GROUP@, and EVERYONE@, which align with User, Group, and Other from traditional POSIX permission bits. When providing them in the ACE string format, they must be spelled exactly as written here, in uppercase with no domain appended. The GROUP@ entry must also have the G (group) flag.

共有三个特殊主体，OWNER@、GROUP@ 和 EVERYONE@，它们与传统 POSIX 权限位中的 User、Group 和 Other 保持一致。 当以 ACE 字符串格式提供它们时，它们必须完全按照此处所写的拼写，大写且不附加域。 GROUP@ 条目还必须具有 G（组）标志。

##### Permissions

The permissions in a resource's ACE permit a certain type of user access to the resource. The order of the permission "bits" (characters) within the PERMISSIONS field of the ACE is not significant.

资源 ACE 中的权限允许特定类型的用户访问该资源。 ACE 的 PERMISSIONS 字段中的权限“位”（字符）的顺序并不重要。

|  |  |  |
| --- | --- | --- |
| **Permission** | **Pool Meaning** | **Container Meaning** |
| r (Read) | Alias for 't' | Read container data and attributes |
| w (Write) | Alias for 'c' + 'd' | Write container data and attributes |
| c (Create) | Create containers | N/A |
| d (Delete) | Delete any container | Delete this container |
| t (Get-Prop) | Connect/query | Get container properties |
| T (Set-Prop) | N/A | Set/Change container properties |
| a (Get-ACL) | N/A | Get container ACL |
| A (Set-ACL) | N/A | Set/Change container ACL |
| o (Set-Owner) | N/A | Set/Change container's owner user and group |

#### [Data Integrity](https://docs.daos.io/overview/data_integrity/)

##### Introduction

Arguably, one of the worst things a data storage system can do is to return incorrect data without the requester knowing. While each component in the system (network layer, storage devices) may offer protection against silent data corruption, DAOS provides end-to-end data integrity using checksums to better ensure that user data is not corrupted silently.

可以说，数据存储系统可以做的最糟糕的事情之一就是在请求者不知情的情况下返回不正确的数据。 虽然系统中的每个组件（网络层、存储设备）都可以提供防止静默数据损坏的保护，但 DAOS 使用校验和提供端到端数据完整性，以更好地确保用户数据不会静默损坏。

For DAOS, end-to-end means that the client will calculate and verify checksums, providing protection for data through the entire I/O stack. During a write or update, the DAOS Client library (libdaos.so) calculates a checksum and appends it to the RPC message before transferred over the network. Depending on the configuration, the DAOS Server may or may not calculate checksums to verify the data on receipt. On a fetch, the DAOS Server will send a known good checksum with the requested data to the DAOS Client, which will calculate checksums on the data received and verify.

对于 DAOS，端到端意味着客户端将计算和验证校验和，通过整个 I/O 堆栈为数据提供保护。 在写入或更新期间，DAOS 客户端库 (libdaos.so) 在通过网络传输之前计算校验和并将其附加到 RPC 消息。 根据配置，DAOS 服务器可能会或可能不会计算校验和来验证接收到的数据。 在获取时，DAOS 服务器将向 DAOS 客户端发送带有请求数据的已知良好校验和，DAOS 客户端将计算接收到的数据的校验和并进行验证。

##### Requirements

###### Key Requirements

There are two key requirements that DAOS will support. 1. Detect silent data corruption - Corruption will be detected on the distribution and attribute keys and records within a DAOS object. At a minimum, when corruption is detected, an error will be reported. 1. Correct data corruption - When data corruption is detected, an attempt will be made to recover the data using data redundancy mechanisms.

DAOS 将支持两个关键要求。 1. 检测静默数据损坏 - 将在 DAOS 对象内的分布和属性键和记录上检测损坏。 至少，当检测到损坏时，将报告错误。 1. 纠正数据损坏 - 当检测到数据损坏时，将尝试使用数据冗余机制恢复数据。

###### Supportive/Additional Requirements

Additionally, DAOS will support ... 1. End to End Data Integrity as a Quality of Service Attribute - Container properties are used to enable/disable the use of checksums for data integrity as well as define specific attributes of data integrity feature. See https://daos-stack.github.io/user/container/#data-integrity for details on configuring a container with checksums enabled. 1. Minimize Performance Impact - When there is no data corruption, the End to End Data Integrity feature should have minimal performance impacted. If data corruption is detected, performance can be impacted to correct the data. Work is ongoing to minimize performance impact. 1. Inject Errors - The ability to corrupt data within a specific record, key, or checksum will be necessary for testing purposes. Fault injection is used to simulate corruption over the network and on disk. The DAOS\_CSUM\_CORRUPT\_\* flags used for data corruption are defined in src/include/daos/common.h. 1. Logging - When data corruption is detected, error logs are captured in the client and server logs.

此外，DAOS 将支持... 1. 端到端数据完整性作为服务质量属性 - 容器属性用于启用/禁用数据完整性校验和的使用以及定义数据完整性功能的特定属性。有关配置启用校验和的容器的详细信息，请参阅 https://daos-stack.github.io/user/container/#data-integrity。 1. 最小化性能影响 - 当没有数据损坏时，端到端数据完整性功能应该对性能影响最小。如果检测到数据损坏，则可能会影响性能以更正数据。正在努力将性能影响降至最低。 1. 注入错误 - 破坏特定记录、密钥或校验和中数据的能力对于测试来说是必要的。故障注入用于模拟网络和磁盘上的损坏。用于数据损坏的 DAOS\_CSUM\_CORRUPT\_\* 标志在 src/include/daos/common.h 中定义。 1. 日志记录 - 当检测到数据损坏时，会在客户端和服务器日志中捕获错误日志。

Up coming features not supported yet 1. Event Logging - When silent data corruption is discovered, an event should be logged in such a way that it can be retrieved with other system health and diagnostic information. 1. Proactive background service task - A background task on the server which scans for and detects (audits checksums) silent data corruption and corrects.

尚不支持即将推出的功能 1. 事件记录 - 当发现静默数据损坏时，应以一种可以与其他系统运行状况和诊断信息一起检索的方式记录事件。 1. 主动后台服务任务 - 服务器上的后台任务，用于扫描和检测（审核校验和）静默数据损坏并进行更正。

##### Keys and Value Objects

Because DAOS is a key/value store, the data for both keys and values is protected, however, the approach for both is slightly different. For the two different value types, single and array, the approach is also slightly different.

由于 DAOS 是键/值存储，因此键和值的数据都受到保护，但是，两者的方法略有不同。 对于两种不同的值类型，single 和 array，方法也略有不同。

###### Keys

On an update and fetch, the client calculates a checksum for the data used as the distribution and attribute keys and will send it to the server within the RPC. The server verifies the keys with the checksum. While enumerating keys, the server will calculate checksums for the keys and pack within the RPC message to the client. The client will verify the keys received.

在更新和获取时，客户端计算用作分布和属性键的数据的校验和，并将其发送到 RPC 中的服务器。 服务器使用校验和验证密钥。 在枚举密钥时，服务器将计算密钥的校验和并将其打包到客户端的 RPC 消息中。 客户端将验证收到的密钥。

Note：

Checksums for keys are not stored on the server. A hash of the key is calculated and used to index the key in the server tree of the keys (see [VOS Key Array Stores](https://docs.daos.io/src/vos/README.md#key-array-stores)). It is also expected that keys are stored only in Storage Class Memory which has reliable data integrity protection.

密钥的校验和不存储在服务器上。 密钥的散列被计算并用于在密钥的服务器树中索引密钥（请参阅 VOS 密钥阵列存储）。 还期望密钥仅存储在具有可靠数据完整性保护的存储类内存中。

###### Values

On an update, the client will calculate a checksum for the data of the value and will send it to the server within the RPC. If "server verify" is enabled, the server will calculate a new checksum for the value and compare with the checksum received from the client to verify the integrity of the value. If the checksums don't match, then data corruption has occurred and an error is returned to the client indicating that the client should try the update again. Whether "server verify" is enabled or not, the server will store the checksum. See [VOS](https://docs.daos.io/src/vos/README.md) for more info about checksum management and storage in VOS.

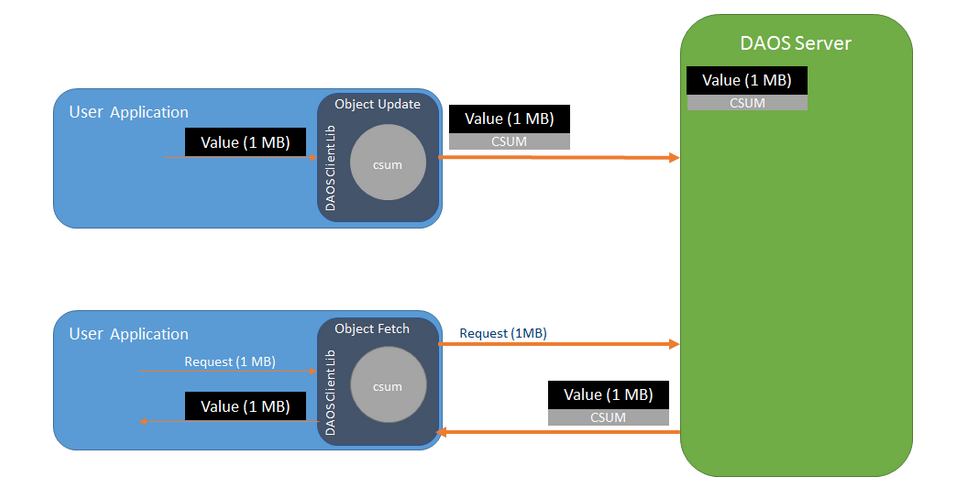
在更新时，客户端将为该值的数据计算校验和，并将其发送到 RPC 中的服务器。 如果启用了“服务器验证”，服务器将为该值计算一个新的校验和，并与从客户端收到的校验和进行比较以验证该值的完整性。 如果校验和不匹配，则发生数据损坏，并向客户端返回错误，指示客户端应再次尝试更新。 无论是否启用“服务器验证”，服务器都会存储校验和。 有关 VOS 中校验和管理和存储的更多信息，请参阅 VOS。

On a fetch, the server will return the stored checksum to the client with the values fetched so the client can verify the values received. If the checksums don't match, then the client will fetch from another replica if available in an attempt to get uncorrupted data.

在获取时，服务器会将存储的校验和连同获取的值返回给客户端，以便客户端可以验证接收到的值。 如果校验和不匹配，则客户端将从另一个副本（如果可用）获取以尝试获取未损坏的数据。

There are some slight variations to this approach for the two different types of values. The following diagram illustrates a basic example. (See [Storage Model](https://docs.daos.io/overview/storage/) for more details about the single value and array value types)

对于两种不同类型的值，这种方法有一些细微的变化。 下图说明了一个基本示例。 （有关单值和数组值类型的更多详细信息，请参阅存储模型）



###### Single Value

A Single Value is an atomic value, meaning that writes to a single value will update the entire value and reads retrieve the entire value. Other DAOS features such as Erasure Codes might split a Single Value into multiple shards to be distributed among multiple storage nodes. Either the whole Single Value (if going to a single node) or each shard (if distributed) will have a checksum calculated, sent to the server, and stored on the server.

单个值是一个原子值，这意味着写入单个值将更新整个值并读取检索整个值。 其他 DAOS 功能（例如纠删码）可能会将单个值拆分为多个分片，以分布在多个存储节点之间。 整个 Single Value（如果转到单个节点）或每个分片（如果分布式）都将计算校验和，发送到服务器并存储在服务器上。

Note that it is possible for a single value, or shard of a single value, to be smaller than the checksum derived from it. It is advised that if an application needs many small single values to use an Array Type instead.

请注意，单个值或单个值的分片可能小于从中得出的校验和。 建议如果应用程序需要许多小的单个值来使用数组类型。

###### Array Values

Unlike Single Values, Array Values can be updated and fetched at any part of an array. In addition, updates to an array are versioned, so a fetch can include parts from multiple versions of the array. Each of these versioned parts of an array are called extents. The following diagrams illustrate a couple examples (also see [VOS Key Array Stores](https://docs.daos.io/src/vos/README.md#key-array-stores) for more information):

A single extent update (blue line) from index 2-13. A fetched extent (orange line) from index 2-6. The fetch is only part of the original extent written. ![](../graph/data\_integrity/array\_example\_1.png)

Many extent updates and different epochs. A fetch from index 2-13 requires parts from each extent. ![Array Example 2](../graph/data\_integrity/array\_example\_2.png)

与单值不同，数组值可以在数组的任何部分更新和获取。 此外，对数组的更新是受版本控制的，因此一次提取可以包含来自数组的多个版本的部分。 数组的这些版本化部分中的每一个都称为范围。 下图说明了几个示例（有关更多信息，另请参阅 VOS 密钥阵列存储）：

The nature of the array type requires that a more sophisticated approach to creating checksums is used. DAOS uses a "chunking" approach where each extent will be broken up into "chunks" with a predetermined "chunk size." Checksums will be derived from these chunks. Chunks are aligned with an absolute offset (starting at 0), not an I/O offset. The following diagram illustrates a chunk size configured to be 4 (units is arbitrary in this example). Though not all chunks have a full size of 4, an absolute offset alignment is maintained. The gray boxes around the extents represent the chunks.

数组类型的性质要求使用更复杂的方法来创建校验和。 DAOS 使用“分块”方法，其中每个范围将被分成具有预定“块大小”的“块”。 校验和将从这些块中派生。 块与绝对偏移量（从 0 开始）对齐，而不是 I/O 偏移量。 下图说明了配置为 4 的块大小（在此示例中单位是任意的）。 尽管并非所有块的完整大小都是 4，但仍保持绝对偏移对齐。 范围周围的灰色框代表块。

图片包含 图形用户界面

描述已自动生成

##### Checksum calculations

The actual checksum calculations are done by the [isa-l](https://github.com/intel/isa-l) and [isa-l\_crypto](https://github.com/intel/isa-l_crypto) libraries. However, these libraries are abstracted away from much of DAOS and a common checksum library is used with appropriate adapters to the actual isa-l implementations. [common checksum library](https://docs.daos.io/src/common/README.md#checksum)

实际的校验和计算由 isa-l 和 isa-l\_crypto 库完成。 然而，这些库是从大部分 DAOS 中抽象出来的，一个通用的校验和库与适当的适配器一起用于实际的 isa-l 实现。 通用校验和库

##### Performance Impact

Calculating checksums can be CPU intensive and will impact performance. To mitigate performance impact, checksum types with hardware acceleration should be chosen. For example, CRC32C is supported by recent Intel CPUs, and many are accelerated via SIMD.

计算校验和可能是 CPU 密集型的，并且会影响性能。 为了减轻性能影响，应选择具有硬件加速的校验和类型。 例如，最近的 Intel CPU 支持 CRC32C，并且许多 CPU 都通过 SIMD 加速。

##### Quality

Unit and functional testing is performed at many layers.

|  |  |  |
| --- | --- | --- |
| **Test executable** | **What's tested** | **Key test files** |
| common\_test | daos\_csummer, utility functions to help with chunk alignment | src/common/tests/checksum\_tests.c |
| vos\_test | vos\_obj\_update/fetch apis with checksum params to ensure updating and fetching checksums | src/vos/tests/vts\_checksum.c |
| srv\_checksum\_tests | Server side logic for adding fetched checksums to an array request. Checksums are appropriately copied or created depending on extent layout. | src/object/tests/srv\_checksum\_tests.c |
| daos\_test | daos\_obj\_update/fetch with checksums enabled. The -z flag can be used for specific checksum tests. Also --csum\_type flag can be used to enable checksums with any of the other daos\_tests | src/tests/suite/daos\_checksum.c |

##### Running Tests

**With daos\_server not running**

./commont\_test

./vos\_test -z

./srv\_checksum\_tests

**With daos\_server running**

export DAOS\_CSUM\_TEST\_ALL\_TYPE=1

./daos\_server -z

./daos\_server -i --csum\_type crc64

##### Life of a checksum (WIP)

###### Rebuild

In order for rebuild/migrate process to get checksums so it doesn't have to recalculate them, the object list and object fetch task api's provide a checksum iov parameter. If memory is allocated for the iov, then the daos client will pack the checksums into the it. If insufficient memory is allocated in the buffer, the iov\_len will be set to the required capacity and the checksums packed into the buffer is truncated.

为了让重建/迁移过程获得校验和，以便不必重新计算它们，对象列表和对象获取任务 api 提供了一个校验和 iov 参数。 如果为 iov 分配了内存，那么 daos 客户端会将校验和打包到其中。 如果缓冲区中分配的内存不足，则 iov\_len 将设置为所需的容量，并截断打包到缓冲区中的校验和。

###### Client Task API Touch Points

* **dc\_obj\_fetch\_task\_create**: sets csum iov to daos\_obj\_fetch\_t args. These args are set to the rw\_cb\_args.shard\_args.api\_args and accessed through an accessor function (rw\_args2csum\_iov) in cli\_shard.c so that rw\_args\_store\_csum can easily access it. This function, called from dc\_rw\_cb\_csum\_verify, will pack the data checksums received from the server into the iov.
* **dc\_obj\_list\_obj\_task\_create**: sets csum iov to daos\_obj\_list\_obj\_t args. args.csum is then copied to obj\_enum\_args.csum in dc\_obj\_shard\_list(). On enum callback (dc\_enumerate\_cb()) the packed csum buffer is copied from the rpc args to obj\_enum\_args.csum (which points to the same buffer as the caller's)

###### Rebuild Touch Points

* migrate\_fetch\_update\_(inline|single|bulk) - the rebuild/migrate functions that write to vos locally must ensure that the checksum is also written. These must use the csum iov param for fetch to get the checksum, then unpack the csums into iod\_csum.
* obj\_enum.c is relied on for enumerating the objects to be rebuilt. Because the fetch\_update functions will unpack the csums from fetch, it will also unpack the csums for enum, so the unpacking process in obj\_enum.c will simply copy the csum\_iov to the io (dss\_enum\_unpack\_io) structure in **enum\_unpack\_recxs()** and then deep copy to the mrone (migrate\_one) structure in **migrate\_one\_insert()**.

###### Packing/unpacking checksums

When checksums are packed (either for fetch or object list) only the data checksums are included. For object list, only checksums for data that is inlined is included. During a rebuild, if the data is not inlined, then the rebuild process will fetch the rest of the data and also get the checksums.

当校验和被打包（用于获取或对象列表）时，只包含数据校验和。 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() - "packs" checksums by appending the struct to an iov and then appending the checksum info buffer to the iov. This puts the actual checksum just after the checksum structure that describes the checksum.
* ci\_cast() - "unpacks" the checksum and describing structure. It does this by casting an iov's buffer to a dcs\_csum\_info struct and setting the csum\_info's checksum pointer to point to the memory just after the structure. It does not copy anything, but really just "casts". To get all dcs\_csum\_infos, a caller would cast the iov, copy the csum\_info to a destination, then move to the next csum\_info(ci\_move\_next\_iov) in the iov. Because this process modifies the iov structure it is best to use a copy of the iov as a temp structure.

###### VOS

* akey\_update\_begin - determines how much extra space needs to be allocated in SCM to account for the checksum

###### Arrays

* evt\_root\_activate - evtree root is activated. If has a csum them the root csum properties are set (csum\_len, csum\_type, csum\_chunk\_size)
* evt\_desc\_csum\_fill - if root was activated with a punched record then it won't have had the csum fields set correctly so set them here. Main purpose is to copy the csum to the end of persistent evt record (evt\_desc). Enough SCM should have been reserved in akey\_update\_begin.
* evt\_entry\_csum\_fill - Copy the csum from the persistent memory to the evt\_entry returned. Also copy the csum fields from the evtree root to complete the csum\_info structure in the evt\_entry.
* akey\_fetch\_recx - checksums are saved to the ioc for each found extent. Will be used to be added to to the result later.

###### Update/Fetch (copied from vos/README.md)

* SV Update: vos\_update\_end -> akey\_update\_single -> svt\_rec\_store
* Sv Fetch: vos\_fetch\_begin -> akey\_fetch\_single -> svt\_rec\_load
* EV Update: vos\_update\_end -> akey\_update\_recx -> evt\_insert
* EV Fetch: vos\_fetch\_begin -> akey\_fetch\_recx -> evt\_fill\_entry

###### Enumeration

For enumeration the csums for the keys and values are packed into an iov dedicated to csums. - fill\_key\_csum - Checksum is calcuated for the key and packed into the iov - fill\_data\_csum - pack/serialize the csum\_info structure into the iov.

对于枚举，键和值的 csum 被打包到一个专用于 csum 的 iov 中。 - fill\_key\_csum - 计算密钥的校验和并将其打包到 iov - fill\_data\_csum - 将 csum\_info 结构打包/序列化到 iov 中。

###### Aggregation

* srv\_csum\_recalc.c - the checksum verification and calculations occur here

##### Checksum Scrubbing (In Development)

A background task will scan (when the storage server is idle to limit performance impact) the Version Object Store (VOS) trees to verify the data integrity with the checksums. Corrective actions can be taken when corruption is detected. See [Corrective Actions](https://docs.daos.io/overview/data_integrity/#corrective-actions)

后台任务将扫描（当存储服务器空闲以限制性能影响时）版本对象存储 (VOS) 树以使用校验和验证数据完整性。 当检测到损坏时，可以采取纠正措施。 查看纠正措施

##### Scanner

###### Goals/Requirements

* **Detect Silent Data Corruption Proactively** - The whole point of the scrubber is to detect silent data corruption before it is fetched.
* **Minimize CPU and I/O Bandwidth** - Checksum scrubbing scanner will impact CPU and the I/O bandwidth because it must iterate the VOS tree (I/O to SCM) fetch data (I/O to SSD) and calculate checksums (CPU intensive). To minimize both of these impacts, the server scheduler must be able to throttled the scrubber's I/O and CPU usage.
* **Minimize Media Wear** - The background task will minimize media wear by preventing objects from being scrubbed too frequently. A container config/tunable will be used by an operator to define the minimum number of days that should pass before an object is scanned again.
* **Continuous** - The background task will be a continuous processes instead of running on a schedule. Once complete immediately start over. Throttling approaches should prevent from scrubbing same objects too frequently.

###### High Level Design

* Per Pool ULT (I/O xstream) that will iterate containers. If checksums and scrubber is enabled then iterate the object tree. If a record value (SV or array) is not marked corrupted then scan.
  + Fetch the data.
  + Create new ULTs (helper xstream) to calculate checksum for data
  + Compare calculated checksum with stored checksum.
  + After every checksum is calculated, determine if need to [sleep or yield](https://docs.daos.io/overview/data_integrity/#sleep-or-yield).
  + If checksums don't match confirm record is still there (not deleted by aggregation) then update record as corrupted
* After each object scanned yield to allow the server scheduler to reschedule the next appropriate I/O.

###### Sleep or Yield

Sleep for sufficient amount of time to ensure that scanning completes no sooner than configured interval (i.e. once a week or month). For example, if the interval is 1 week and there are 70 checksums that need to be calculated, then at a maximum 10 checksums are calculated a day, spaced roughly every 2.4 hours. If it doesn't need to sleep, then it will yield to allow the server scheduler to prioritize other jobs.

睡眠充足的时间以确保扫描不早于配置的时间间隔（即每周或每月一次）完成。 例如，如果间隔为 1 周并且需要计算 70 个校验和，那么每天最多计算 10 个校验和，间隔大约为 2.4 小时。 如果它不需要休眠，那么它会让服务器调度程序优先处理其他作业。

##### Corrective Actions

There are two main options for corrective actions when data corruption is discovered, in place data repair and SSD eviction.

当发现数据损坏时，有两种主要的纠正措施选项，数据修复和 SSD 驱逐。

###### In Place Data Repair

If enabled, when corruption is detected, the value identifier (dkey, akey, recx) will be placed in a queue. When there are available cycles, the value identifier will be used to request the data from a replica if exists and rewrite the data locally. This will continue until the SSD Eviction threshold is reached, in which case, the SSD is assumed to be bad enough that it isn't worth fixing locally and it will be requested to be evicted.

如果启用，当检测到损坏时，值标识符（dkey、akey、recx）将被放入队列中。 当有可用周期时，值标识符将用于从副本请求数据（如果存在）并在本地重写数据。 这将一直持续到达到 SSD 驱逐阈值，在这种情况下，假定 SSD 已经足够糟糕，不值得在本地修复，并将被要求驱逐。

###### SSD Eviction

If enabled, when the SSD Eviction Threshold is reached the SSD will be evicted. Current eviction methods are pool and target based so there will need to be a mapping and mechanism in place to evict an SSD. When an SSD is evicted, the rebuild protocol will be invoked.

Also, once the SSD Eviction Threshold is reached, the scanner should quit scanning anything on that SSD.

如果启用，当达到 SSD 驱逐阈值时，SSD 将被驱逐。 当前的驱逐方法是基于池和target的，因此需要有适当的映射和机制来驱逐 SSD。 当 SSD 被驱逐时，将调用重建协议。 此外，一旦达到 SSD 驱逐阈值，扫描仪应停止扫描该 SSD 上的任何内容

###### Additional Checksum Properties > doc/user/container.md / doc/user/pool.md?

These properties are provided when a container or pool is created, but should also be able to update them. When updated, they should be active right away. - Scanner Interval - Minimum number of days scanning will take. Could take longer, but if only a few records will pad so takes longer. (Pool property) - Disable scrubbing - at container level & pool level - Threshold for when to evict SSD (number of corruption events) - In Place Correction - If the number checksum errors is below the Eviction Threshold, DAOS will attempt to repair the corrupted data using replicas if they exist.

这些属性在创建容器或池时提供，但也应该能够更新它们。 更新后，它们应该立即处于活动状态。 - 扫描间隔 - 扫描所需的最少天数。 可能需要更长的时间，但如果只有几条记录会填充，那么需要更长的时间。 （池属性） - 禁用清理 - 在容器级别和池级别 - 何时驱逐 SSD 的阈值（损坏事件的数量） - 就地更正 - 如果校验和错误数低于驱逐阈值，DAOS 将尝试修复损坏的 数据使用副本（如果存在）。

##### Design Details & Implementation

###### Pool ULT

The code for the pool ULT is found in srv\_pool\_scrub.c. It can be a bit difficult to follow because there are several layers of callback functions due to the nature of how ULTs and the vos\_iterator work, but the file is organized such that functions typically call the function above it (either directly or indirectly as a callback). For example (~> is an indirect call, -> is a direct call):

ds\_start\_scrubbing\_ult ~> scrubbing\_ult -> scrub\_pool ~> cont\_iter\_scrub\_cb ->

scrub\_cont ~> obj\_iter\_scrub\_cb ...

###### Silent Data Corruption Detection (TODO)

::Still todo::

obj\_iter\_scrub(coh, epr, csummer, pool\_uuid, event\_handlers, entry, type)

{

build\_iod

vos\_obj\_fetch(coh, oid, epoch, dkey, iod, sgl);

// for single value

csum = calc\_checksum(type, csummer, iod, sgl)

compare(csum, entry.csum)

// for recx

for each chunk calc csum and compare

}

###### VOS Layer

* In order to mark data as corrupted a flag field is added to bio\_addr\_t which includes a CORRUPTED bit.
* The vos update api already accepts a flag, so a CORRUPTED flag is added and handled during an update so that, if set, the bio address will be updated to be corrupted.
* On fetch, if a value is already marked corrupted, return -DER\_CSUM

###### Object Layer

* When corruption is detected on the server during a fetch, aggregation, or rebuild the server calls VOS to update value as corrupted.
* (TBD) Add Server Side Verifying on fetch so can know if media or network corruption (note: need something so extents aren't double verified?)

###### Debugging

* In the server.yml configuration file set the following env\_vars

- D\_LOG\_MASK=DEBUG

- DD\_SUBSYS=pool

- DD\_MASK=csum

## Administration Guide

System administration topics are covered in the Administration Guide.

系统管理主题包含在管理指南中

#### Hardware Requirements

The purpose of this section is to describe processor, storage, and network requirements to deploy a DAOS system.

本节的目的是描述部署 DAOS 系统的处理器、存储和网络要求。

##### Processor Requirements

DAOS requires a 64-bit processor architecture and is primarily developed on Intel x86\_64 architecture. The DAOS software and the libraries it depends on (e.g., ISA-L, SPDK, PMDK, and DPDK) can take advantage of Intel SSE and AVX extensions.

DAOS 需要 64 位处理器架构，主要在 Intel x86\_64 架构上开发。 DAOS 软件及其依赖的库（例如 ISA-L、SPDK、PMDK 和 DPDK）可以利用英特尔 SSE 和 AVX 扩展。

DAOS is also regularly tested on 64-bit ARM processors configured in Little Endian mode. The same build instructions that are used for x86\_64 are applicable for ARM builds as well. DAOS and its dependencies will make the necessary adjustments automatically in their respective build systems for ARM platforms.

DAOS 还在以 Little Endian 模式配置的 64 位 ARM 处理器上定期测试。 用于 x86\_64 的相同构建指令也适用于 ARM 构建。 DAOS 及其依赖项将在其各自的 ARM 平台构建系统中自动进行必要的调整。

##### Network Requirements

The DAOS network layer relies on libfabrics and supports OFI providers for Ethernet/sockets, InfiniBand/verbs, RoCE, and Intel Omni-Path Architecture (OPA). An RDMA-capable fabric is preferred for better performance. DAOS can support multiple rails by binding different instances of the DAOS server to individual network cards.

DAOS 网络层依赖于 libfabrics 并支持以太网/套接字、InfiniBand/verbs、RoCE 和英特尔 Omni-Path 架构 (OPA) 的 OFI 提供程序。 首选支持 RDMA 的结构以获得更好的性能。 DAOS 可以通过将 DAOS 服务器的不同实例绑定到单个网卡来支持多个导轨。

The DAOS control plane provides methods for administering and managing the DAOS servers using a secure socket layer interface. An additional out-of-band network connecting the nodes in the DAOS service cluster is required for DAOS administration. Management traffic between clients and servers uses IP over Fabric.

DAOS 控制平面提供了使用安全套接字层接口管理和管理 DAOS 服务器的方法。 DAOS 管理需要一个额外的带外网络来连接 DAOS 服务集群中的节点。 客户端和服务器之间的管理流量使用 IP over Fabric。

##### Storage Requirements

DAOS requires each storage node to have direct access to storage-class memory (SCM). While DAOS is primarily tested and tuned for Optane DC Persistent Memory, the DAOS software stack is built over the Persistent Memory Development Kit (PMDK) and the DAX feature of the Linux and Windows operating systems as described in the SNIA NVM Programming Model[1](https://docs.daos.io/admin/hardware/#fn:1). As a result, the open-source DAOS software stack should be able to run transparently over any storage-class memory supported by the PMDK.

DAOS 要求每个存储节点都可以直接访问存储级内存（SCM）。 虽然 DAOS 主要针对 Optane DC 持久内存进行测试和调整，但 DAOS 软件堆栈是基于持久内存开发套件 (PMDK) 以及 Linux 和 Windows 操作系统的 DAX 功能构建的，如 SNIA NVM 编程模型 1 中所述。 因此，开源 DAOS 软件堆栈应该能够在 PMDK 支持的任何存储级内存上透明地运行。

The storage node can optionally be equipped with NVMe (non-volatile memory express) SSDs to provide capacity. HDDs, as well as SATA and SAS SSDs, are not supported by DAOS. Both NVMe 3D-NAND and Optane SSDs are supported. Optane SSDs are preferred for DAOS installation that targets a very high IOPS rate. NVMe-oF devices are also supported by the userspace storage stack but have never been tested.

存储节点可以选择配备 NVMe（非易失性内存快速）SSD 以提供容量。 DAOS 不支持 HDD 以及 SATA 和 SAS SSD。 支持 NVMe 3D-NAND 和 Optane SSD。 Optane SSD 是 DAOS 安装的首选，其目标是非常高的 IOPS 速率。 用户空间存储堆栈也支持 NVMe-oF 设备，但从未经过测试。

A minimum 6% ratio of SCM to SSD capacity will guarantee that DAOS has enough space in SCM to store its internal metadata (e.g., pool metadata, SSD block allocation tracking).

SCM 与 SSD 容量的最低 6% 比率将保证 DAOS 在 SCM 中有足够的空间来存储其内部元数据（例如，池元数据、SSD 块分配跟踪）。

For testing purposes, SCM can be emulated with DRAM by mounting a tmpfs filesystem, and NVMe SSDs can be also emulated with DRAM or a loopback file.

出于测试目的，可以通过挂载 tmpfs 文件系统使用 DRAM 模拟 SCM，也可以使用 DRAM 或环回文件模拟 NVMe SSD。

##### Storage Server Design

The hardware design of a DAOS storage server balances the network bandwidth of the fabric with the aggregate storage bandwidth of the NVMe storage devices. This relationship sets the number of NVMe drives. For example, 8 PCIe gen4 x4 NVMe SSDs balance two 200Gbps PCIe gen4 x16 network adapters. The capacity of the SSDs will determine the minimum capacity of the Optane PMem DIMMs needed to provide the 6% ratio for DAOS metadata.

DAOS 存储服务器的硬件设计平衡了结构的网络带宽与 NVMe 存储设备的聚合存储带宽。 此关系设置 NVMe 驱动器的数量。 例如，8 个 PCIe gen4 x4 NVMe SSD 平衡了两个 200Gbps PCIe gen4 x16 网络适配器。 SSD 的容量将决定为 DAOS 元数据提供 6% 比率所需的 Optane PMem DIMM 的最小容量。

日程表

低可信度描述已自动生成

##### CPU Affinity

Recent Intel Xeon data center platforms use two processor CPUs connected together with the Ultra Path Interconnect (UPI). PCIe lanes in these servers have a natural affinity to one CPU. Although globally accessible from any of the system cores, NVMe SSDs and network interface cards connected through the PCIe bus may provide different performance characteristics (e.g., higher latency, lower bandwidth) to each CPU. Accessing non-local PCIe devices may involve traffic over the UPI link that might become a point of congestion. Similarly, persistent memory is non-uniformly accessible (NUMA), and CPU affinity must be respected for maximal performance.

最近的英特尔至强数据中心平台使用两个通过 Ultra Path Interconnect (UPI) 连接在一起的处理器 CPU。 这些服务器中的 PCIe 通道与一个 CPU 具有天然的亲和力。 尽管可从任何系统内核全局访问，但通过 PCIe 总线连接的 NVMe SSD 和网络接口卡可为每个 CPU 提供不同的性能特征（例如，更高的延迟、更低的带宽）。 访问非本地 PCIe 设备可能涉及 UPI 链路上的流量，这可能会成为拥塞点。 同样，持久内存是非统一可访问的 (NUMA)，必须考虑 CPU 亲和性才能获得最大性能。

Therefore, when running in a multi-socket and multi-rail environment, the DAOS service must be able to detect the CPU to PCIe device and persistent memory affinity and minimize as much as possible non-local access. This can be achieved by spawning one instance of the I/O Engine per CPU, then accessing only the persistent memory and PCI devices local to that CPU from that server instance. The DAOS control plane is responsible for detecting the storage and network affinity and starting the I/O Engines accordingly.

因此，在多路多轨环境中运行时，DAOS 服务必须能够检测 CPU 到 PCIe 设备和持久内存的亲和性，并尽可能减少非本地访问。 这可以通过为每个 CPU 生成一个 I/O 引擎实例来实现，然后从该服务器实例仅访问该 CPU 本地的持久内存和 PCI 设备。 DAOS 控制平面负责检测存储和网络关联并相应地启动 I/O 引擎。

图形用户界面, 图示

描述已自动生成

##### Fault Domains

DAOS relies on single-ported storage massively distributed across different storage nodes. Each storage node is thus a single point of failure. DAOS achieves fault tolerance by providing data redundancy across storage nodes in different fault domains.

DAOS 依赖于大规模分布在不同存储节点上的单端口存储。 因此，每个存储节点都是单点故障。 DAOS 通过跨不同容错域中的存储节点提供数据冗余来实现容错。

DAOS assumes that fault domains are hierarchical and do not overlap. For instance, the first level of a fault domain could be the racks and the second one, the storage nodes.

DAOS 假定故障域是分层的并且不重叠。 例如，故障域的第一级可能是机架，第二级可能是存储节点。

For efficient placement and optimal data resilience, more fault domains are better. As a result, it is preferable to distribute storage nodes across as many racks as possible.

为了实现高效放置和最佳数据弹性，故障域越多越好。 因此，最好将存储节点分布在尽可能多的机架上。

#### Software Installation

Please check the [support matrix](https://daos-stack.github.io/release/support_matrix) to select the appropriate software combination.

请检查支持矩阵以选择合适的软件组合。

##### Distribution Packages

DAOS packages are available for CentOS7 and openSUSE Leap 15. For other distribution, manual building is required (see next section).

DAOS 包适用于 CentOS7 和 openSUSE Leap 15。对于其他发行版，需要手动构建（见下一节）。

##### CentOS 7.9

Configure the DAOS package repository as follows:

$ sudo wget -O /etc/yum.repos.d/daos-packages.repo http://packages.daos.io/v1.2/CentOS7/packages/x86\_64/daos\_packages.repo

epel-release is required for both client and server:

$ sudo yum install -y epel-release

To install the DAOS client packages:

$ sudo yum install -y daos-client

To install the DAOS server packages:

$ sudo yum install -y daos-server

Debug and source RPMs available in the respective [debug](http://packages.daos.io/v1.2/CentOS7/debug/x86_64/daos_debug.repo) and [source](http://packages.daos.io/v1.2/CentOS7/source/daos_source.repo) repositories.

##### openSUSE Leap 15

Add the DAOS package repository via zypper:

$ sudo zypper ar http://packages.daos.io/v1.2/Leap15.2/packages x86\_64/daos\_packages

$ sudo rpm --import http://packages.daos.io/RPM-GPG-KEY

$ sudo zypper --non-interactive ref

To install the DAOS client packages:

$ sudo zypper install -y daos-client

To install the DAOS server packages:

$ sudo zypper install -y daos-server

Debug and source RPMs available in the respective [debug](http://packages.daos.io/v1.2/Leap15.2/debug/x86_64/daos_debug.repo) and [source](http://packages.daos.io/v1.2/Leaps15.2/source/daos_source.repo) repositories.

##### DAOS from Scratch

The following instructions have been verified with CentOS. Installations on other Linux distributions might be similar with some variations. Developers of DAOS may want to review the additional sections below before beginning, for suggestions related specifically to development. Contact us in our [forum](https://daos.groups.io/g/daos) for further help with any issues.

##### Build Prerequisites

To build DAOS and its dependencies, several software packages must be installed on the system. This includes scons, libuuid, cmocka, ipmctl, and several other packages usually available on all the Linux distributions. Moreover, a Go version of at least 1.10 is required.

An exhaustive list of packages for each supported Linux distribution is maintained in the Docker files (please click on the link):

* [CentOS 7](https://github.com/daos-stack/daos/tree/release/1.2/utils/docker/Dockerfile.centos.7#L19-L79)
* [openSUSE Leap 15](https://github.com/daos-stack/daos/tree/release/1.2/utils/docker/Dockerfile.leap.15#L36-L85)
* [Ubuntu 20.04](https://github.com/daos-stack/daos/tree/release/1.2/utils/docker/Dockerfile.ubuntu.20.04#L14-L22)

The command lines to install the required packages can be extracted from the Docker files by removing the "RUN" command, which is specific to Docker. Check the [utils/docker](https://github.com/daos-stack/daos/tree/release/1.2/utils/docker) directory for different Linux distribution versions.

Some DAOS tests use MPI. The DAOS build process uses the environment modules package to detect the presence of MPI. If none is found, the build will skip building those tests.

##### DAOS Source Code

The DAOS repository is hosted on [GitHub](https://github.com/daos-stack/daos).

To checkout the latest 1.2 version, simply run:

$ git clone --recurse-submodules -b release/1.2 https://github.com/daos-stack/daos.git

This command clones the DAOS git repository (path referred as ${daospath} below) and initializes all the submodules automatically.

##### Building DAOS & Dependencies

If all the software dependencies listed previously are already satisfied, then type the following command in the top source directory to build the DAOS stack:

$ scons --config=force install

If you are a developer of DAOS, we recommend following the instructions in the [DAOS for Development](https://daos-stack.github.io/dev/development/#building-daos-for-development) section.

Otherwise, the missing dependencies can be built automatically by invoking scons with the following parameters:

$ scons --config=force --build-deps=yes install

By default, DAOS and its dependencies are installed under ${daospath}/install. The installation path can be modified by adding the PREFIX= option to the above command line (e.g., PREFIX=/usr/local).

Note

Several parameters can be set (e.g., COMPILER=clang or COMPILER=icc) on the scons command line. Please see scons --help for all the possible options. Those options are also saved for future compilations.

##### Environment setup

Once built, the environment must be modified to search for binaries and header files in the installation path. This step is not required if standard locations (e.g. /bin, /sbin, /usr/lib, ...) are used.

CPATH=${daospath}/install/include/:$CPATH

PATH=${daospath}/install/bin/:${daospath}/install/sbin:$PATH

export CPATH PATH

If using bash, PATH can be set up for you after a build by sourcing the script utils/sl/utils/setup\_local.sh from the daos root. This script utilizes a file generated by the build to determine the location of daos and its dependencies.

If required, ${daospath}/install must be replaced with the alternative path specified through PREFIX.

##### DAOS in Docker

This section describes how to build and run the DAOS service in a Docker container. A minimum of 5GB of DRAM and 16GB of disk space will be required. On Mac, please make sure that the Docker settings under "Preferences/{Disk, Memory}" are configured accordingly.

##### Building a Docker Image

To build the Docker image directly from GitHub, run the following command:

$ docker build https://github.com/daos-stack/daos.git#release/1.2 \

-f utils/docker/Dockerfile.centos.7 -t daos

or from a local tree:

$ docker build . -f utils/docker/Dockerfile.centos.7 -t daos

This creates a CentOS 7 image, fetches the latest DAOS version from GitHub, builds it, and installs it in the image. For Ubuntu and other Linux distributions, replace Dockerfile.centos.7 with Dockerfile.ubuntu.20.04 and the appropriate version of interest.

##### Simple Docker Setup

Once the image created, one can start a container that will eventually run the DAOS service:

$ docker run -it -d --privileged --cap-add=ALL --name server -v /dev:/dev daos

Note

If you want to be more selective with the devices that are exported to the container, individual devices should be listed and exported as volume via the -v option. In this case, the hugepages devices should also be added to the command line via -v /dev/hugepages:/dev/hugepages and -v /dev/hugepages-1G:/dev/hugepages-1G

Warning

If Docker is being run on a non-Linux system (e.g., OSX), -v /dev:/dev should be removed from the command line.

The daos\_server\_local.yml configuration file sets up a simple local DAOS system with a single server instance running in the container. By default, it uses 4GB of DRAM to emulate persistent memory and 16GB of bulk storage under /tmp. The storage size can be changed in the yaml file if necessary.

The DAOS service can be started in the docker container as follows:

$ docker exec server daos\_server start \

-o /home/daos/daos/utils/config/examples/daos\_server\_local.yml

Note

Please make sure that the uio\_pci\_generic module is loaded on the host.

Once started, the DAOS server waits for the administrator to format the system. This can be triggered in a different shell, using the following command:

$ docker exec server dmg -i storage format

Upon successful completion of the format, the storage engine is started, and pools can be created using the daos admin tool (see next section).

For more advanced configurations involving SCM, SSD or a real fabric, please refer to the next section.

#### Pre-deployment Checklist

This section covers the preliminary setup required on the compute and storage nodes before deploying DAOS.

本节介绍部署 DAOS 之前计算和存储节点上所需的初步设置。

##### Enable IOMMU (Optional)

In order to run the DAOS server as a non-root user with NVMe devices, the hardware must support virtualized device access, and it must be enabled in the system BIOS. On Intel® systems, this capability is named Intel® Virtualization Technology for Directed I/O (VT-d). Once enabled in BIOS, IOMMU support must also be enabled in the Linux kernel. Exact details depend on the distribution, but the following example should be illustrative:

为了使用 NVMe 设备以非 root 用户身份运行 DAOS 服务器，硬件必须支持虚拟化设备访问，并且必须在系统 BIOS 中启用。 在英特尔® 系统上，此功能称为面向定向 I/O 的英特尔® 虚拟化技术 (VT-d)。 在 BIOS 中启用后，还必须在 Linux 内核中启用 IOMMU 支持。 确切的细节取决于分布，但下面的例子应该是说明性的：（在计算机领域，IOMMU（Input/Output Memory Management Unit）是一个内存管理单元（Memory Management Unit），它的作用是连接DMA-capable I/O总线（Direct Memory Access-capable I/O Bus）和主存（main memory）。传统的内存管理单元会把CPU访问的虚拟地址转化成实际的物理地址。而IOMMU则是把设备（device）访问的虚拟地址转化成物理地址。为了防止设备错误地访问内存，有些IOMMU还提供了访问内存保护机制。）

# Enable IOMMU on CentOS 7

# All commands must be run as root/sudo!

$ sudo vi /etc/default/grub # add the following line:

GRUB\_CMDLINE\_LINUX\_DEFAULT="intel\_iommu=on"

# after saving the file, run the following to reconfigure

# the bootloader:

$ sudo grub2-mkconfig --output=/boot/grub2/grub.cfg

# if the command completed with no errors, reboot the system

# in order to make the changes take effect

$ sudo reboot

Warning

VFIO support is a new feature for DAOS 1.2 and has been tested on CentOS 7.7

To force SPDK to use UIO rather than VFIO at daos\_server runtime, set 'disable\_vfio' in the [server config file](https://github.com/daos-stack/daos/blob/master/utils/config/daos_server.yml#L109), but note that this will require running daos\_server as root.

警告： VFIO 支持是 DAOS 1.2 的新功能，已在 CentOS 7.7 上测试 要强制 SPDK 在 daos\_server 运行时使用 UIO 而不是 VFIO，请在服务器配置文件中设置“disable\_vfio”，但请注意，这需要以 root 身份运行 daos\_server

##### Time Synchronization

The DAOS transaction model relies on timestamps and requires time to be synchronized across all the storage and client nodes. This can be done using NTP or any other equivalent protocol.

DAOS 事务模型依赖于时间戳，并且需要在所有存储和客户端节点之间同步时间。 这可以使用 NTP 或任何其他等效协议来完成。

##### User/Group Synchronization

DAOS ACLs store the actual user and group names (instead of numeric IDs), and therefore the servers do not need access to a synchronized user/group database. The DAOS Agent (running on the client nodes) is responsible for resolving UID/GID to user/group names which are added to a signed credential and sent to the DAOS storage nodes.

DAOS ACL 存储实际的用户名和组名（而不是数字 ID），因此服务器不需要访问同步的用户/组数据库。 DAOS 代理（在客户端节点上运行）负责将 UID/GID 解析为用户/组名称，这些名称被添加到签名凭证并发送到 DAOS 存储节点。

##### Multi-rail/NIC Setup

Storage nodes can be configured with multiple network interfaces to run multiple engine instances.

存储节点可以配置多个网络接口来运行多个引擎实例。

##### Subnet

Since all DAOS engines need to be able to communicate, the different network interfaces need to be on the same subnet or routing capabilities across the different subnet might be configured.

由于所有 DAOS 引擎都需要能够进行通信，因此不同的网络接口需要位于同一子网上，或者可能会配置跨不同子网的路由功能。

##### Infiniband/RoCE

Some special configuration is required to use librdmacm with multiple interfaces.

需要一些特殊配置才能使用具有多个接口的 librdmacm。

Firstly, the accept\_local feature must be enabled on the network interfaces to be used by DAOS. This can be done using the following command ( must be replaced with the interface names):

首先，必须在 DAOS 使用的网络接口上启用 accept\_local 功能。 这可以使用以下命令完成（必须替换为接口名称）：

$ sudo sysctl -w net.ipv4.conf.all.accept\_local=1

Secondly, Linux must be configured to only send ARP replies on the interface targeted in the ARP request. This is configured via the arp\_ignore parameter. This should be set to 2 if all the IPoIB interfaces on the client and storage nodes are in the same logical subnet (e.g. ib0 == 10.0.0.27, ib1 == 10.0.1.27, prefix=16).

其次，Linux 必须配置为仅在 ARP 请求中的目标接口上发送 ARP 回复。 这是通过 arp\_ignore 参数配置的。 如果客户端和存储节点上的所有 IPoIB 接口都在同一逻辑子网中（例如 ib0 == 10.0.0.27、ib1 == 10.0.1.27、prefix=16），则应将其设置为 2。

$ sysctl -w net.ipv4.conf.all.arp\_ignore=2

If separate logical subnets are used (e.g. prefix = 24), then the value must be set to 1.

如果使用单独的逻辑子网（例如前缀 = 24），则该值必须设置为 1。

$ sysctl -w net.ipv4.conf.all.arp\_ignore=1

Finally, the rp\_filter is set to 1 by default on several distributions (e.g. on CentOS 7) and should be set to either 0 or 2, with 2 being more secure. This is true even if the configuration uses a single logical subnet.

最后，rp\_filter 在几个发行版上默认设置为 1（例如在 CentOS 7 上），并且应该设置为 0 或 2，2 更安全。 即使配置使用单个逻辑子网也是如此。

$ sysctl -w net.ipv4.conf.<ifaces>.rp\_filter=2

All those parameters can be made persistent in /etc/sysctl.conf by adding a new sysctl file under /etc/sysctl.d (e.g. /etc/sysctl.d/95-daos-net.conf) with all the relevant settings.

For more information, please refer to the [librdmacm documentation](https://github.com/linux-rdma/rdma-core/blob/master/Documentation/librdmacm.md)

通过在 /etc/sysctl.d 下添加一个新的 sysctl 文件（例如 /etc/sysctl.d/95-daos-net.conf）和所有相关设置，所有这些参数都可以在 /etc/sysctl.conf 中持久化。 更多信息请参考 librdmacm 文档

##### Runtime Directory Setup

DAOS uses a series of Unix Domain Sockets to communicate between its various components. On modern Linux systems, Unix Domain Sockets are typically stored under /run or /var/run (usually a symlink to /run) and are a mounted tmpfs file system. There are several methods for ensuring the necessary directories are setup.

DAOS 使用一系列 Unix 域套接字在其各个组件之间进行通信。 在现代 Linux 系统上，Unix 域套接字通常存储在 /run 或 /var/run（通常是 /run 的符号链接）下，并且是挂载的 tmpfs 文件系统。 有几种方法可以确保设置必要的目录。

A sign that this step may have been missed is when starting daos\_server or daos\_agent, you may see the message:

这一步可能被遗漏的迹象是在启动 daos\_server 或 daos\_agent 时，您可能会看到以下消息：

$ mkdir /var/run/daos\_server: permission denied

Unable to create socket directory: /var/run/daos\_server

无法创建套接字目录：/var/run/daos\_server

##### Non-default Directory

By default, daos\_server and daos\_agent will use the directories /var/run/daos\_server and /var/run/daos\_agent respectively. To change the default location that daos\_server uses for its runtime directory, either uncomment and set the socket\_dir configuration value in install/etc/daos\_server.yml, or pass the location to daos\_server on the command line using the -d flag. For the daos\_agent, an alternate location can be passed on the command line using the --runtime\_dir flag.

默认情况下，daos\_server 和 daos\_agent 将分别使用目录 /var/run/daos\_server 和 /var/run/daos\_agent。 要更改 daos\_server 用于其运行时目录的默认位置，请在 install/etc/daos\_server.yml 中取消注释并设置 socket\_dir 配置值，或者在命令行上使用 -d 标志将该位置传递给 daos\_server。 对于 daos\_agent，可以使用 --runtime\_dir 标志在命令行上传递备用位置。

##### Default Directory (non-persistent)

Files and directories created in /run and /var/run only survive until the next reboot. However, if reboots are infrequent, an easy solution while still utilizing the default locations is to create the required directories manually. To do this execute the following commands.

在 /run 和 /var/run 中创建的文件和目录只能在下次重新启动之前保留。 但是，如果不经常重新启动，在仍然使用默认位置的情况下一个简单的解决方案是手动创建所需的目录。 为此，请执行以下命令。

daos\_server:

$ mkdir /var/run/daos\_server

$ chmod 0755 /var/run/daos\_server

$ chown user:user /var/run/daos\_server (where user is the user you

will run daos\_server as)

daos\_agent:

$ mkdir /var/run/daos\_agent

$ chmod 0755 /var/run/daos\_agent

$ chown user:user /var/run/daos\_agent (where user is the user you

will run daos\_agent as)

##### Default Directory (persistent)

If the server hosting daos\_server or daos\_agent will be rebooted often, systemd provides a persistent mechanism for creating the required directories called tmpfiles.d. This mechanism will be required every time the system is provisioned and requires a reboot to take effect.

如果托管 daos\_server 或 daos\_agent 的服务器经常重启，systemd 提供了一种持久机制来创建名为 tmpfiles.d 的所需目录。 每次配置系统时都需要此机制，并且需要重新启动才能生效。

To tell systemd to create the necessary directories for DAOS:

* Copy the file utils/systemd/daosfiles.conf to /etc/tmpfiles.d\ cp utils/systemd/daosfiles.conf /etc/tmpfiles.d
* Modify the copied file to change the user and group fields (currently daos) to the user daos will be run as
* Reboot the system, and the directories will be created automatically on all subsequent reboots.

告诉 systemd 为 DAOS 创建必要的目录：

• 将文件 utils/systemd/daosfiles.conf 复制到 /etc/tmpfiles.d\ cp utils/systemd/daosfiles.conf /etc/tmpfiles.d

• 修改复制的文件以将用户和组字段（当前为 daos）更改为用户 daos 将作为

• 重新启动系统，随后的所有重新启动都会自动创建目录

##### Elevated Privileges

DAOS employs a privileged helper binary (daos\_admin) to perform tasks that require elevated privileges on behalf of daos\_server.

DAOS 使用特权帮助程序二进制文件 (daos\_admin) 来代表 daos\_server 执行需要提升权限的任务。

##### Privileged Helper Configuration

When DAOS is installed from RPM, the daos\_admin helper is automatically installed to the correct location with the correct permissions. The RPM creates a "daos\_server" system group and configures permissions such that daos\_admin may only be invoked from daos\_server.

当从 RPM 安装 DAOS 时，daos\_admin 助手会自动安装到正确的位置并具有正确的权限。 RPM 创建一个“daos\_server”系统组并配置权限，使得 daos\_admin 只能从 daos\_server 调用。

For non-RPM installations, there are two supported scenarios:

1. daos\_server is run as root, which means that daos\_admin is also invoked as root, and therefore no additional setup is necessary.
2. daos\_server is run as a non-root user, which means that daos\_admin must be manually installed and configured.

对于非 RPM 安装，有两种支持的方案：

1. daos\_server 以 root 身份运行，这意味着 daos\_admin 也以 root 身份调用，因此无需额外设置。

2. daos\_server 以非root用户运行，这意味着必须手动安装和配置daos\_admin。

The steps to enable the second scenario are as follows (steps are assumed to be running out of a DAOS source tree which may be on a NFS share):

启用第二个场景的步骤如下（假设步骤用完可能位于 NFS 共享上的 DAOS 源树）：

$ chmod -x $daospath/bin/daos\_admin # prevent this copy from being executed

$ sudo cp $daospath/bin/daos\_admin /usr/bin/daos\_admin

$ sudo chmod 4755 /usr/bin/daos\_admin # make this copy setuid root

$ sudo mkdir -p /usr/share/daos/control # create symlinks to SPDK scripts

$ sudo ln -sf $daospath/share/daos/control/setup\_spdk.sh \

/usr/share/daos/control

$ sudo mkdir -p /usr/share/spdk/scripts

$ sudo ln -sf $daospath/share/spdk/scripts/setup.sh \

/usr/share/spdk/scripts

$ sudo ln -sf $daospath/share/spdk/scripts/common.sh \

/usr/share/spdk/scripts

$ sudo ln -s $daospath/include \

/usr/share/spdk/include

Note

The RPM installation is preferred for production scenarios. Manual installation is most appropriate for development and predeployment proof-of-concept scenarios.

注意：RPM 安装是生产场景的首选。 手动安装最适合开发和部署前的概念验证场景。

##### Memory Lock Limits/内存锁定限制

Low ulimit for memlock can cause SPDK to fail and emit the following error:

memlock 的低 ulimit 会导致 SPDK 失败并发出以下错误：

daos\_engine:1 EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)

daos\_engine:1 EAL：无法设置 DMA 重映射，错误 12（无法分配内存）

The memlock limit only needs to be manually adjusted when daos\_server is not running as a systemd service. Default ulimit settings vary between OSes.

仅当 daos\_server 未作为 systemd 服务运行时，才需要手动调整 memlock 限制。 默认 ulimit 设置因操作系统而异。

For RPM installations, the service will typically be launched by systemd and the limit is pre-set to unlimited in the daos\_server.service unit file: https://github.com/daos-stack/daos/blob/master/utils/systemd/daos\_server.service#L16. Note that values set in /etc/security/limits.conf are ignored by services launched by systemd.

对于 RPM 安装，该服务通常由 systemd 启动，并且在 daos\_server.service 单元文件中将限制预设为无限制：https://github.com/daos-stack/daos/blob/master/utils/systemd /daos\_server.service#L16。 请注意，systemd 启动的服务会忽略 /etc/security/limits.conf 中设置的值。

For non-RPM installations where daos\_server is launched directly from the commandline, limits should be adjusted in /etc/security/limits.conf as per <https://software.intel.com/content/www/us/en/develop/blogs/best-known-methods-for-setting-locked-memory-size.html>.

对于直接从命令行启动 daos\_server 的非 RPM 安装，应根据 https://software.intel.com/content/www/us/en/develop/blogs 在 /etc/security/limits.conf 中调整限制 /best-known-methods-for-setting-locked-memory-size.html。

#### [System Deployment](https://docs.daos.io/admin/deployment/)

#### [System Administration](https://docs.daos.io/admin/administration/)

#### [Pool Operations](https://docs.daos.io/admin/pool_operations/)

#### [Tiering and Unified Namespace](https://docs.daos.io/admin/tiering_uns/)

#### [Performance Tuning](https://docs.daos.io/admin/performance_tuning/)

#### [Troubleshooting](https://docs.daos.io/admin/troubleshooting/)

#### [Utilities and Usage Examples](https://docs.daos.io/admin/utilities_examples/)

#### [Environment Variables](https://docs.daos.io/admin/env_variables/)

## User Guide

Documentation for users including the different interfaces that are supported.

用户文档，包括支持的不同接口

#### Container Management

DAOS containers are the unit of data management for users.

DAOS 容器是用户数据管理的单位。

##### Container Creation/Destroy

Containers can be created and destroyed through the daos\_cont\_create/destroy() functions exported by the DAOS API. A user tool called daos is also provided to manage containers.

可以通过 DAOS API 导出的 daos\_cont\_create/destroy() 函数创建和销毁容器。 还提供了一个名为 daos 的用户工具来管理容器。

To create a container:

$ daos cont create --pool=a171434a-05a5-4671-8fe2-615aa0d05094

Successfully created container 008123fc-6b6c-4768-a88a-a2a5ef34a1a2

The container type (i.e., POSIX or HDF5) can be passed via the --type option. As shown below, the pool UUID, container UUID, and container attributes can be stored in the extended attributes of a POSIX file or directory for convenience. Then subsequent invocations of the daos tools need to reference the path to the POSIX file or directory.

容器类型（即 POSIX 或 HDF5）可以通过 --type 选项传递。 如下图，为了方便，可以将池UUID、容器UUID、容器属性存放在POSIX文件或目录的扩展属性中。 然后daos工具的后续调用需要引用POSIX文件或目录的路径

$ daos cont create --pool=a171434a-05a5-4671-8fe2-615aa0d05094 \

--path=/tmp/mycontainer --type=POSIX --oclass=large \

--chunk\_size=4K

Successfully created container 419b7562-5bb8-453f-bd52-917c8f5d80d1 type POSIX

$ daos container query --path=/tmp/mycontainer

Pool UUID: a171434a-05a5-4671-8fe2-615aa0d05094

Container UUID: 419b7562-5bb8-453f-bd52-917c8f5d80d1

Number of snapshots: 0

Latest Persistent Snapshot: 0

DAOS Unified Namespace Attributes on path /tmp/mycontainer:

Container Type: POSIX

Object Class: large

Chunk Size: 4096

##### Container Properties

At creation time, a list of container properties can be specified:

在创建时，可以指定容器属性列表

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

While those properties are currently stored persistently with container metadata, many of them are still under development. The ability to modify some of these properties on an existing container will also be provided in a future release.

虽然这些属性目前与容器元数据一起永久存储，但其中许多仍在开发中。 将来的版本中还将提供修改现有容器上的某些属性的功能。

##### Data Integrity

DAOS allows to detect and fix (when data protection is enabled) silent data corruptions. This is done by calculating checksums for both data and metadata in the DAOS library on the client side and storing those checksums persistently in SCM. The checksums will then be validated on access and on update/write as well on the server side if server verify option is enabled.

DAOS 允许检测和修复（启用数据保护时）静默数据损坏。 这是通过在客户端的 DAOS 库中计算数据和元数据的校验和并将这些校验和持久存储在 SCM 中来完成的。 如果启用了服务器验证选项，则校验和将在访问和更新/写入以及服务器端验证。

Corrupted data will never be returned to the application. When a corruption is detected, DAOS will try to read from a different replica, if any. If the original data cannot be recovered, then an error will be reported to the application.

损坏的数据永远不会返回给应用程序。 当检测到损坏时，DAOS 将尝试从不同的副本（如果有）读取。 如果无法恢复原始数据，则会向应用程序报告错误。

To enable and configure checksums, the following container properties are used during container create.

要启用和配置校验和，在容器创建期间使用以下容器属性。

* DAOS\_PROP\_CO\_CSUM: Type of checksum algorithm to use. Supported values are

DAOS\_PROP\_CO\_CSUM\_OFF, // default

DAOS\_PROP\_CO\_CSUM\_CRC16,

DAOS\_PROP\_CO\_CSUM\_CRC32,

DAOS\_PROP\_CO\_CSUM\_CRC64,

DAOS\_PROP\_CO\_CSUM\_SHA1,

DAOS\_PROP\_CO\_CSUM\_SHA256,

DAOS\_PROP\_CO\_CSUM\_SHA512,

DAOS\_PROP\_CO\_CSUM\_ADLER32

* DAOS\_PROP\_CO\_CSUM\_CHUNK\_SIZE: defines the chunk size used for creating checksums of array types. (default is 32K).
* DAOS\_PROP\_CO\_CSUM\_SERVER\_VERIFY: Because of the probable decrease to IOPS, in most cases, it is not desired to verify checksums on an object update on the server side. It is sufficient for the client to verify on a fetch because any data corruption, whether on the object update, storage, or fetch, will be caught. However, there is an advantage to knowing if corruption happens on an update. The update would fail right away, indicating to the client to retry the RPC or report an error to upper levels.

由于可能会降低 IOPS，因此在大多数情况下，不需要在服务器端验证对象更新的校验和。 客户端验证获取就足够了，因为任何数据损坏，无论是对象更新、存储还是获取，都将被捕获。 但是，了解更新是否发生损坏是有好处的。 更新将立即失败，指示客户端重试 RPC 或向上级报告错误

Note

Note that currently, once a container is created, its checksum configuration cannot be changed.

请注意，目前，一旦创建了容器，就无法更改其校验和配置

##### Inline Deduplication (Preview)

Data deduplication (dedup) is a process that allows to eliminate duplicated data copies in order to decrease capacity requirements. DAOS has some initial support of inline dedup.

重复数据删除 (dedup) 是一个允许消除重复数据副本以降低容量需求的过程。 DAOS 初步支持内联重复数据删除。

When dedup is enabled, each DAOS server maintains a per-pool table indexing extents by their hash (i.e. checksum). Any new I/Os bigger than the deduplication threshold will thus be looked up in this table to find out whether an existing extent with the same signature has already been stored. If an extent is found, then two options are provided:

启用重复数据删除后，每个 DAOS 服务器通过其hash（即校验和）维护每个池表的索引范围。 任何大于重删阈值的io都会再表中查找是否存在相同的指纹。 如果找到，则提供两个选项：

* Transferring the data from the client to the server and doing a memory compare (i.e. memcmp) of the two extents to verify that they are indeed identical.
* Trusting the hash function and skipping the data transfer. To minimize issue with hash collision, a cryptographic hash function (i.e. SHA256) is used in this case. The benefit of this approarch is that the data to be written does not need to be transferred to the server. Data processing is thus greatly accelerated.

• 将数据从客户端传输到服务器，并对两个区进行内存比较（即 memcmp）以验证它们确实相同。

• 信任散列函数并跳过数据传输。 为了尽量减少散列冲突问题，在这种情况下使用加密散列函数（即 SHA256）。 这种方法的好处是要写入的数据不需要传输到服务器。 数据处理因此大大加速。

The inline dedup feature can be enabled on a per-container basis. To enable and configure dedup, the following container properties are used:

可以在每个容器的基础上启用内联重复数据删除功能。 要启用和配置重复数据删除，使用以下容器属性

* DAOS\_PROP\_CO\_DEDUP: Type of dedup mechanism to use. Supported values are

要使用的重复数据删除机制的类型。 支持的值为

DAOS\_PROP\_CO\_DEDUP\_OFF, // default

DAOS\_PROP\_CO\_DEDUP\_MEMCMP, // memory compare

DAOS\_PROP\_CO\_CSUM\_HASH // hash-based using SHA256

* DAOS\_PROP\_CO\_DEDUP\_THRESHOLD: defines the minimal I/O size to consider the I/O for dedup (default is 4K).

Warning

Dedup is a feature preview in 1.2 (i.e. master) and has some known limitations. Aggregation of deduplicated extents isn't supported and the checksum tree isn't persistent yet. This means that aggregation is disabled for a container with dedplication enabled and duplicated extents won't be matched after a server restart. NVMe isn't supported for dedup enabled container, so please make sure not using dedup on the pool with NVMe enabled.

Dedup 是 1.2（即 master）中的功能预览，并且有一些已知的限制。 不支持去重范围的聚合，校验和树尚未持久化。 这意味着对于启用了重复数据删除的容器禁用聚合，并且在服务器重新启动后不会匹配重复的范围。 启用了重复数据删除的容器不支持 NVMe，因此请确保不要在启用了 NVMe 的池上使用重复数据删除

##### Compression & Encryption

The DAOS\_PROP\_CO\_COMPRESS and DAOS\_PROP\_CO\_ENCRYPT properties are reserved for configuring respectively online compression and encryption. These features are currently not on the roadmap.

DAOS\_PROP\_CO\_COMPRESS 和 DAOS\_PROP\_CO\_ENCRYPT 属性分别用于配置在线压缩和加密。 这些功能目前不在路线图上。

##### Snapshot & Rollback

Similar to container create/destroy, a container can be snapshotted through the DAOS API by calling daos\_cont\_create\_snap(). Additional functions are provided to destroy and list container snapshots.

与容器创建/销毁类似，可以通过调用 daos\_cont\_create\_snap() 通过 DAOS API 对容器进行快照。 提供了附加功能来销毁和列出容器快照。

The API also provides the ability to subscribe to container snapshot events and to rollback the content of a container to a previous snapshot, but those operations are not yet fully implemented.

该 API 还提供订阅容器快照事件并将容器内容回滚到先前快照的功能，但这些操作尚未完全实现。

This section will be updated once support for container snapshot is supported by the daos tool.

一旦 daos 工具支持容器快照，本节将更新

The DAOS\_PROP\_CO\_SNAPSHOT\_MAX property is used to limit the maximum number of snapshots to retain. When a new snapshot is taken, and the threshold is reached, the oldest snapshot will be automatically deleted.

DAOS\_PROP\_CO\_SNAPSHOT\_MAX 属性用于限制要保留的最大快照数。 当拍摄新快照并达到阈值时，将自动删除最旧的快照。

Rolling back the content of a container to a snapshot is planned for future DAOS versions.

未来的 DAOS 版本计划将容器的内容回滚到快照。

##### User Attributes[¶](https://docs.daos.io/user/container/#user-attributes)

Similar to POSIX extended attributes, users can attach some metadata to each container through the daos\_cont\_{list/get/set}\_attr() API.

类似于 POSIX 扩展属性，用户可以通过 daos\_cont\_{list/get/set}\_attr() API 向每个容器附加一些元数据

##### Access Control Lists[¶](https://docs.daos.io/user/container/#access-control-lists)

Client user and group access for containers is controlled by [Access Control Lists (ACLs)](https://daos-stack.github.io/overview/security/#access-control-lists).

Access-controlled container accesses include:

* Opening the container for access.
* Reading and writing data in the container.
* Reading and writing objects.
* Getting, setting, and listing user attributes.
* Getting, setting, and listing snapshots.
* Deleting the container (if the pool does not grant the user permission).
* Getting and setting container properties.
* Getting and modifying the container ACL.
* Modifying the container's owner.

This is reflected in the set of supported [container permissions](https://daos-stack.github.io/overview/security/#permissions).

##### Pool vs. Container Permissions[¶](https://docs.daos.io/user/container/#pool-vs-container-permissions)

In general, pool permissions are separate from container permissions, and access to one does not guarantee access to the other. However, a user must have permission to connect to a container's pool before they can access the container in any way, regardless of their permissions on that container. Once the user has connected to a pool, container access decisions are based on the individual container ACL. A user need not have read/write access to a pool in order to open a container with read/write access, for example.

一般而言，池权限与容器权限是分开的，访问一个权限并不能保证访问另一个权限。 但是，用户必须具有连接到容器池的权限，然后才能以任何方式访问容器，而不管他们对该容器的权限如何。 一旦用户连接到池，容器访问决策将基于单个容器 ACL。 例如，用户无需对池具有读/写访问权限即可打开具有读/写访问权限的容器。

There is one situation in which the pool can grant a container-level permission: Container deletion. If a user has Delete permission on a pool, this grants them the ability to delete any container in the pool, regardless of their permissions on that container.

在一种情况下，池可以授予容器级权限：容器删除。 如果用户对池具有删除权限，这将授予他们删除池中任何容器的能力，而不管他们对该容器的权限如何。

If the user does not have Delete permission on the pool, they will only be able to delete containers for which they have been explicitly granted Delete permission in the container's ACL.

如果用户没有池的删除权限，他们将只能删除在容器的 ACL 中明确授予他们删除权限的容器。

##### Creating Containers with Custom ACL[¶](https://docs.daos.io/user/container/#creating-containers-with-custom-acl)

To create a container with a custom ACL:

要使用自定义 ACL 创建容器

$ daos cont create --pool=<UUID> --acl-file=<path>

The ACL file format is detailed in the [ACL section](https://daos-stack.github.io/overview/security/#acl-file).

##### Displaying a Container's ACL[¶](https://docs.daos.io/user/container/#displaying-a-containers-acl)

To view a container's ACL:

$ daos cont get-acl --pool=<UUID> --cont=<UUID>

The output is in the same string format used in the ACL file during creation, with one ACE per line.

输出与创建期间 ACL 文件中使用的字符串格式相同，每行一个 ACE

##### Modifying a Container's ACL[¶](https://docs.daos.io/user/container/#modifying-a-containers-acl)

For all of these commands using an ACL file, the ACL file must be in the format noted above for container creation.

对于使用 ACL 文件的所有这些命令，ACL 文件必须采用上述用于创建容器的格式

##### Overwriting the ACL[¶](https://docs.daos.io/user/container/#overwriting-the-acl)

To replace a container's ACL with a new ACL:

$ daos cont overwrite-acl --pool=<UUID> --cont=<UUID> \

--acl-file=<path>

##### Adding and Updating ACEs[¶](https://docs.daos.io/user/container/#adding-and-updating-aces)

To add or update multiple entries in an existing container ACL:

$ daos cont update-acl --pool=<UUID> --cont=<UUID> \

--acl-file=<path>

To add or update a single entry in an existing container ACL:

$ daos cont update-acl --pool=<UUID> --cont=<UUID> --entry <ACE>

If there is no existing entry for the principal in the ACL, the new entry is added to the ACL. If there is already an entry for the principal, that entry is replaced with the new one.

如果 ACL 中没有主体的现有条目，则将新条目添加到 ACL。 如果主体已有条目，则该条目将替换为新条目

##### Removing an ACE[¶](https://docs.daos.io/user/container/#removing-an-ace)

To delete an entry for a given principal in an existing container ACL:

要删除现有容器 ACL 中给定主体的条目

$ daos cont delete-acl --pool=<UUID> --cont=<UUID> \

--principal=<principal>

The principal corresponds to the principal portion of an ACE that was set during container creation or a previous container ACL operation. For the delete operation, the principal argument must be formatted as follows:

主体对应于在容器创建或先前容器 ACL 操作期间设置的 ACE 的主体部分。 对于删除操作，主体参数的格式必须如下：

* Named user: u:username@
* Named group: g:groupname@
* Special principals:
* OWNER@
* GROUP@
* EVERYONE@

The entry for that principal will be completely removed. This does not always mean that the principal will have no access. Rather, their access to the container will be decided based on the remaining ACL rules.

该主体的条目将被完全删除。 这并不总是意味着主体将无权访问。 相反，他们对容器的访问将根据剩余的 ACL 规则来决定。

##### Ownership[¶](https://docs.daos.io/user/container/#ownership)

The ownership of the container corresponds to the special principals OWNER@ and GROUP@ in the ACL. These values are a part of the container properties. They may be set on container creation and changed later.

容器的所有权对应于 ACL 中的特殊主体 OWNER@ 和 GROUP@。 这些值是容器属性的一部分。 它们可以在容器创建时设置并稍后更改。

##### Privileges[¶](https://docs.daos.io/user/container/#privileges)

The owner-user (OWNER@) has implicit privileges on their container. The owner-user can always open the container, and has set-ACL (A) and get-ACL (a) permissions. These permissions are included alongside any permissions that the user was explicitly granted by entries in the ACL.

所有者用户 (OWNER@) 对其容器具有隐式权限。 owner-user 始终可以打开容器，并且具有 set-ACL (A) 和 get-ACL (a) 权限。 这些权限与 ACL 中的条目明确授予用户的任何权限一起包含在内。

Because the owner's special permissions are implicit, they apply to access control decisions even if they do not appear in the OWNER@ entry, and even if the OWNER@ entry is deleted.

The owner-group (GROUP@) has no special permissions outside what they are granted by the ACL.

由于所有者的特殊权限是隐式的，因此即使它们没有出现在 OWNER@ 条目中，并且即使 OWNER@ 条目被删除，它们也适用于访问控制决策。 除了 ACL 授予的权限之外，所有者组 (GROUP@) 没有任何特殊权限。

##### Creating Containers with Specific Ownership[¶](https://docs.daos.io/user/container/#creating-containers-with-specific-ownership)

The default owner user and group are the effective user and group of the user creating the container. However, a specific user and/or group may be specified at container creation time.

默认所有者用户和组是创建容器的用户的有效用户和组。 但是，可以在容器创建时指定特定用户和/或组。

$ daos cont create --pool=<UUID> --user=<owner-user> \

--group=<owner-group>

The user and group names are case sensitive and must be formatted as [DAOS ACL user/group principals](https://daos-stack.github.io/overview/security/#principal).

用户名和组名区分大小写，并且必须格式化为 DAOS ACL 用户/组主体。

##### Changing Ownership[¶](https://docs.daos.io/user/container/#changing-ownership)

To change the owner user:

$ daos cont set-owner --pool=<UUID> --cont=<UUID> \

--user=<owner-user>

To change the owner group:

$ daos cont set-owner --pool=<UUID> --cont=<UUID> \

--group=<owner-group>

The user and group names are case sensitive and must be formatted as [DAOS ACL user/group principals](https://daos-stack.github.io/overview/security/#principal).

用户名和组名区分大小写，并且必须格式化为 DAOS ACL 用户/组主体。

#### Native Programming Interface

##### Building against the DAOS library[¶](https://docs.daos.io/user/interface/#building-against-the-daos-library)

To build application or I/O middleware against the native DAOS API, include the daos.h header file in your program and link with -Ldaos. Examples are available under src/tests.

要针对本机 DAOS API 构建应用程序或 I/O 中间件，请在程序中包含 daos.h 头文件并使用 -Ldaos 链接。 示例可在 src/tests 下找到。

##### DAOS API Reference[¶](https://docs.daos.io/user/interface/#daos-api-reference)

libdaos is written in C and uses Doxygen comments that are added to C header files. The Doxygen documentation is available [here](https://daos-stack.github.io/html/).

libdaos 是用 C 编写的，并使用添加到 C 头文件中的 Doxygen 注释。 Doxygen 文档可在此处获得

##### Python Bindings[¶](https://docs.daos.io/user/interface/#python-bindings)

A python module called [PyDAOS](https://github.com/daos-stack/daos/blob/master/src/client/pydaos) provides the DAOS API to python users.

一个名为 PyDAOS 的 Python 模块为 Python 用户提供了 DAOS API

##### pydaos[¶](https://docs.daos.io/user/interface/" \l "pydaos" \o "Permanent link)

[pydaos](https://github.com/daos-stack/daos/blob/master/src/client/pydaos/pydaos_core.py) provides a native DAOS python interface exported by a C module. It integrates the DAOS key-value store API with python dictionaries. Only strings are supported for both the key and value for now.

pydaos 提供了一个由 C 模块导出的原生 DAOS python 接口。 它将 DAOS 键值存储 API 与 Python 字典集成在一起。 目前只支持键和值的字符串。

Key-value pairs can be inserted/looked up one at a time (see put/get) or in bulk (see bput/bget) taking a python dict as an input. The bulk operations are issued in parallel (up to 16 operations in flight) to maximize the operation rate.

键值对可以一次插入/查找（参见 put/get）或批量插入/查找（参见 bput/bget），以 python dict 作为输入。 批量操作并行发出（最多 16 个飞行操作），以最大限度地提高操作率。

Key-value pairs are deleted via the put/bput operations by setting the value to either None or the empty string. Once deleted, the key won't be reported during iteration. It also supports the del operation via 'del dkv.key'. The DAOS KV objects behave like a python dictionary and supports:

通过将值设置为 None 或空字符串，通过 put/bput 操作删除键值对。 一旦删除，在迭代过程中将不会报告该密钥。 它还支持通过“del dkv.key”进行的del操作。 DAOS KV 对象的行为类似于 Python 字典并支持：

* 'dkv[key]' which invokes 'dkv.get(key)'
* 'dkv[key] = val' which invokes 'dkv.put(key, val)'
* 'for key in dkv:' allows to walk through the key space via the support of python iterators
* 'if key is in dkv:' allows to test whether a give key is present in the DAOS KV store.
* 'len(dkv)' returns the number of key-value pairs.
* 'bool(dkv)' reports 'False' if there is no key-value pairs in the DAOS KV and 'True' otherwise.

Python iterators are supported, which means that "for key in kvobj:" will allow you to walk through the key space. For each method, a PyDError exception is raised with a proper DAOS error code (in string format) if the operation cannot be completed.

支持 Python 迭代器，这意味着“for key in kvobj:”将允许您遍历键空间。 对于每个方法，如果操作无法完成，则会引发 PyDError 异常并带有正确的 DAOS 错误代码（以字符串格式）。

Both python 2.7 and 3.x are supported.

##### pydaos.raw[¶](https://docs.daos.io/user/interface/" \l "pydaosraw" \o "Permanent link)

The pydaos.raw submodule provides access to DAOS API functionality via Ctypes and was developed with an emphasis on test use cases. While the majority of unit tests are written in C, higher-level tests are written primarily using the Python API. Interfaces are provided for accessing DAOS management and DAOS API functionality from Python. This higher level interface allows a faster turnaround time on implementing test cases for DAOS.

pydaos.raw 子模块通过 Ctypes 提供对 DAOS API 功能的访问，其开发重点是测试用例。 虽然大多数单元测试是用 C 编写的，但更高级别的测试主要是使用 Python API 编写的。 提供了用于从 Python 访问 DAOS 管理和 DAOS API 功能的接口。 这个更高级别的接口允许更快地为 DAOS 实现测试用例

##### Layout[¶](https://docs.daos.io/user/interface/#layout)

The Python API is split into several files based on functionality:

* The Python object API: [daos\_api.py](https://github.com/daos-stack/daos/tree/master/src/client/pydaos/raw/daos_api.py).
* The mapping of C structures to Python classes [daos\_cref.py](https://github.com/daos-stack/daos/tree/master/src/client/pydaos/raw/daos_cref.py)

Python API 根据功能分为几个文件：

* • Python 对象API：daos\_api.py。
* • C 结构到 Python 类 daos\_cref.py 的映射

High-level abstraction classes exist to manipulate DAOS storage:

存在用于操作 DAOS 存储的高级抽象类：

class DaosPool(object)

class DaosContainer(object)

class DaosObj(object)

class IORequest(object)

DaosPool is a Python class representing a DAOS pool. All pool-related functionality is exposed from this class. Operations such as creating/destroying a pool, connecting to a pool, and adding a target to a storage pool are supported.

DaosPool 是一个表示 DAOS 池的 Python 类。 所有与池相关的功能都从这个类公开。 支持创建/销毁池、连接池和向存储池添加目标等操作。

DaosContainer is a Python class representing a DAOS container. As with the DaosPool class, all container-related functionality is exposed here. This class also exposes abstracted wrapper functions for the flow of creating and committing an object to a DAOS container.

DaosContainer 是一个表示 DAOS 容器的 Python 类。 与 DaosPool 类一样，所有与容器相关的功能都在这里公开。 此类还公开了用于创建对象并将对象提交到 DAOS 容器的流程的抽象包装函数。

DaosObj is a Python class representing a DAOS object. Functionality such as creating/deleting objects in a container, 'punching' objects (delete an object from the specified transaction only), and object query.

DaosObj 是一个表示 DAOS 对象的 Python 类。 功能，例如在容器中创建/删除对象、“打孔”对象（仅从指定事务中删除对象）和对象查询。

IORequest is a Python class representing a read or write request against a DAOS object.

IORequest 是一个 Python 类，表示对 DAOS 对象的读或写请求。

Several classes exist for management purposes as well:

也存在几个用于管理目的的类：

class DaosContext(object)

class DaosLog

class DaosApiError(Exception)

DaosContext is a wrapper around the DAOS libraries. It is initialized with the path where DAOS libraries can be found.

DaosContext 是 DAOS 库的包装器。 它使用可以找到 DAOS 库的路径进行初始化。

DaosLog exposes functionality to write messages to the DAOS client log.

DaosLog 公开了将消息写入 DAOS 客户端日志的功能。

DaosApiError is a custom exception class raised by the API internally in the event of a failed DAOS action.

DaosApiError 是 API 在 DAOS 操作失败时在内部引发的自定义异常类。

Most functions exposed in the DAOS C API support both synchronous and asynchronous execution, and the Python API exposes this same functionality. Each API takes an input event. DaosEvent is the Python representation of this event. If the input event is NULL, the call is synchronous. If an event is supplied, the function will return immediately after submitting API requests to the underlying stack and the user can poll and query the event for completion.

DAOS C API 中公开的大多数函数都支持同步和异步执行，Python API 公开了相同的功能。 每个 API 接受一个输入事件。 DaosEvent 是此事件的 Python 表示。 如果输入事件为 NULL，则调用是同步的。 如果提供了一个事件，该函数将在向底层堆栈提交 API 请求后立即返回，用户可以轮询和查询该事件是否完成。

##### Ctypes[¶](https://docs.daos.io/user/interface/" \l "ctypes" \o "Permanent link)

Ctypes is a built-in Python module for interfacing Python with existing libraries written in C/C++. The Python API is built as an object-oriented wrapper around the DAOS libraries utilizing ctypes.

Ctypes 是一个内置的 Python 模块，用于将 Python 与用 C/C++ 编写的现有库连接起来。 Python API 构建为使用 ctypes 围绕 DAOS 库的面向对象的包装器。

Ctypes documentation can be found here <https://docs.python.org/3/library/ctypes.html>

The following demonstrates a simplified example of creating a Python wrapper for the C function daos\_pool\_tgt\_exclude\_out, with each input parameter to the C function being cast via ctypes. This also demonstrates struct representation via ctypes:

下面演示了一个为 C 函数 daos\_pool\_tgt\_exclude\_out 创建 Python 包装器的简化示例，C 函数的每个输入参数都通过 ctypes 进行转换。 这也演示了通过 ctypes 的结构表示：

// daos\_exclude.c

#include <stdio.h>

int

daos\_pool\_tgt\_exclude\_out(const uuid\_t uuid, const char \*grp,

struct d\_tgt\_list \*tgts, daos\_event\_t \*ev);

All input parameters must be represented via ctypes. If a struct is required as an input parameter, a corresponding Python class can be created. For struct d\_tgt\_list:

所有输入参数必须通过 ctypes 表示。 如果需要结构体作为输入参数，则可以创建相应的 Python 类。 对于结构 d\_tgt\_list：

struct d\_tgt\_list {

d\_rank\_t \*tl\_ranks;

int32\_t \*tl\_tgts;

uint32\_t tl\_nr;

};

class DTgtList(ctypes.Structure):

\_fields\_ = [("tl\_ranks", ctypes.POINTER(ctypes.c\_uint32)),

("tl\_tgts", ctypes.POINTER(ctypes.c\_int32)),

("tl\_nr", ctypes.c\_uint32)]

The shared object containing daos\_pool\_tgt\_exclude\_out can then be imported and the function called directly:

所有输入参数必须通过 ctypes 表示。如果需要结构体作为输入参数，则可以创建相应的 Python 类。

# api.py

import ctypes

import uuid

import conversion # utility library to convert C <---> Python UUIDs

# init python variables

p\_uuid = str(uuid.uuid4())

p\_tgts = 2

p\_ranks = DaosPool.\_\_pylist\_to\_array([2])

# cast python variables via ctypes as necessary

c\_uuid = str\_to\_c\_uuid(p\_uuid)

c\_grp = ctypes.create\_string\_buffer(b"daos\_group\_name")

c\_tgt\_list = ctypes.POINTER(DTgtList(p\_ranks, p\_tgts, 2))) # again, DTgtList must be passed as pointer

# load the shared object

my\_lib = ctypes.CDLL('/full/path/to/daos\_exclude.so')

# now call it

my\_lib.daos\_pool\_tgt\_exclude\_out(c\_uuid, c\_grp, c\_tgt\_list, None)

##### rror Handling[¶](https://docs.daos.io/user/interface/#error-handling)

The API was designed using the EAFP (**E**asier to **A**sk **F**orgiveness than get **P**ermission) idiom. A given function will raise a custom exception on error state, DaosApiError. A user of the API is expected to catch and handle this exception as needed:

API 是使用 EAFP（比获得许可更容易请求宽恕）习惯用法设计的。 给定的函数将在错误状态下引发自定义异常 DaosApiError。 API 的用户应根据需要捕获并处理此异常：

# catch and log

try:

daos\_some\_action()

except DaosApiError as e:

self.d\_log.ERROR("My DAOS action encountered an error!")

##### Logging[¶](https://docs.daos.io/user/interface/#logging)

The Python DAOS API exposes functionality to log messages to the DAOS client log. Messages can be logged as INFO, DEBUG, WARN, or ERR log levels. The DAOS log object must be initialized with the environmental context in which to run:

Python DAOS API 公开了将消息记录到 DAOS 客户端日志的功能。 消息可以记录为 INFO、DEBUG、WARN 或 ERR 日志级别。 DAOS 日志对象必须使用运行的环境上下文进行初始化

from pydaos.raw import DaosLog

self.d\_log = DaosLog(self.context)

self.d\_log.INFO("FYI")

self.d\_log.DEBUG("Debugging code")

self.d\_log.WARNING("Be aware, may be issues")

self.d\_log.ERROR("Something went very wrong")

##### Go Bindings[¶](https://docs.daos.io/user/interface/#go-bindings)

API bindings for Go[2](https://docs.daos.io/user/interface/#fn:2) are also available.

#### POSIX Namespace

A regular POSIX namespace can be encapsulated into a DAOS container. This capability is provided by the libdfs library that implements the file and directory abstractions over the native libdaos library. The POSIX emulation can be exposed directly to applications or I/O frameworks (e.g., for frameworks like Spark or TensorFlow, or benchmarks like IOR or mdtest that support different storage backend plugins). It can also be exposed transparently via a FUSE daemon, combined optionally with an interception library to address some of the FUSE performance bottlenecks by delivering full OS bypass for POSIX read/write operations.

可以将常规 POSIX 命名空间封装到 DAOS 容器中。 此功能由 libdfs 库提供，该库通过本机 libdaos 库实现文件和目录抽象。 POSIX 仿真可以直接暴露给应用程序或 I/O 框架（例如，对于 Spark 或 TensorFlow 等框架，或支持不同存储后端插件的 IOR 或 mdtest 等基准测试）。 它还可以通过 FUSE 守护程序透明地公开，并可选择与拦截库结合，通过为 POSIX 读/写操作提供完整的操作系统绕过来解决一些 FUSE 性能瓶颈。

日程表

描述已自动生成

##### libdfs[¶](https://docs.daos.io/user/posix/" \l "libdfs" \o "Permanent link)

The DAOS File System (DFS) is implemented in the libdfs library, and allows a DAOS container to be accessed as a hierarchical POSIX namespace. libdfs supports files, directories, and symbolic links, but not hard links. Access permissions are inherited from the parent pool and are not implemented on a per-file or per-directory basis. setuid() and setgid() programs, as well as supplementary groups, are currently not supported.

DAOS 文件系统 (DFS) 在 libdfs 库中实现，并允许将 DAOS 容器作为分层 POSIX 命名空间进行访问。 libdfs 支持文件、目录和符号链接，但不支持硬链接。 访问权限是从父池继承的，不是在每个文件或每个目录的基础上实现的。 目前不支持 setuid() 和 setgid() 程序以及补充组。

It is possible to use libdfs in a parallel application from multiple nodes. When the same POSIX container is mounted concurrently by multiple processes, a few limitations exist in DAOS v1.0. In particular:

* Unlinking a file in one process while another process has the same file open: This may or may not cause an I/O error on the open file.
* The atomicity of rename operations is not guaranteed.

可以在来自多个节点的并行应用程序中使用 libdfs。 当多个进程同时挂载同一个 POSIX 容器时，DAOS v1.0 存在一些限制。 特别是：

• 在另一个进程打开同一文件时取消一个进程中的文件的链接：这可能会也可能不会导致打开文件的 I/O 错误。

• 不能保证重命名操作的原子性。

These corner cases will be addressed in a future DAOS release.

##### DFuse[¶](https://docs.daos.io/user/posix/" \l "dfuse" \o "Permanent link)

DFuse provides DAOS File System access through the standard libc/kernel/VFS POSIX infrastructure. This allows existing applications to use DAOS without modification, and provides a path to upgrade those applications to native DAOS support. Additionally, DFuse provides an Interception Library libioil to transparently allow POSIX clients to talk directly to DAOS servers, providing OS-Bypass for I/O without modifying or recompiling of the application.

DFuse builds heavily on DFS. Data written via DFuse can be accessed by DFS and vice versa.

DFuse 通过标准的 libc/kernel/VFS POSIX 基础架构提供 DAOS 文件系统访问。 这允许现有应用程序无需修改即可使用 DAOS，并提供将这些应用程序升级到本机 DAOS 支持的途径。 此外，DFuse 提供了一个拦截库 libioil，以透明地允许 POSIX 客户端直接与 DAOS 服务器对话，为 I/O 提供 OS-Bypass，而无需修改或重新编译应用程序。 DFuse 大量构建在 DFS 上。 通过 DFuse 写入的数据可以被 DFS 访问，反之亦然。

##### DFuse Daemon[¶](https://docs.daos.io/user/posix/#dfuse-daemon)

The dfuse daemon runs a single instance per node to provide a user POSIX access to DAOS. It should be run with the credentials of the user, and typically will be started and stopped on each compute node as part of the prolog and epilog scripts of any resource manager or scheduler in use. One DFuse daemon per node can process requests for multiple clients.

A single DFuse instance can provide access to multiple pools and containers concurrently, or can be limited to a single pool, or a single container.

dfuse 守护进程为每个节点运行一个实例，为用户提供对 DAOS 的 POSIX 访问。 它应该使用用户的凭据运行，并且通常会在每个计算节点上启动和停止，作为正在使用的任何资源管理器或调度程序的序言和结尾脚本的一部分。 每个节点一个 DFuse 守护进程可以处理多个客户端的请求。 单个 DFuse 实例可以同时提供对多个池和容器的访问，也可以限制为单个池或单个容器。

##### Restrictions[¶](https://docs.daos.io/user/posix/#restrictions)/限制

DFuse is limited to a single user. Access to the filesystem from other users, including root, will not be honored. As a consequence of this, the chown and chgrp calls are not supported. Hard links and special device files, except symbolic links, are not supported, nor are any ACLs.

DFuse 仅限于单个用户。 其他用户（包括 root）对文件系统的访问将不被接受。 因此，不支持 chown 和 chgrp 调用。 不支持硬链接和特殊设备文件（符号链接除外），也不支持任何 ACL。

DFuse can run in the foreground, keeping the terminal window open, or it can daemonize to run like a system daemon. However, to do this and still be able to access DAOS it needs to daemonize before calling daos\_init(). This in turns means it cannot report some kinds of startup errors either on stdout/stderr or via its return code.  
When initially starting with DFuse it is recommended to run in foreground mode (--foreground) to better observe any failures.

DFuse 可以在前台运行，保持终端窗口打开，或者它可以像系统守护进程一样运行。 然而，要做到这一点并且仍然能够访问 DAOS，它需要在调用 daos\_init() 之前进行守护进程。 这反过来意味着它无法在 stdout/stderr 或通过其返回码报告某些类型的启动错误。 最初使用 DFuse 时，建议在前台模式 (--foreground) 下运行以更好地观察任何故障。

Inodes are managed on the local node by DFuse. So while inode numbers will be consistent on a node for the duration of the session, they are not guaranteed to be consistent across restarts of DFuse or across nodes.

inode 由 DFuse 在本地节点上管理。 因此，虽然 inode 编号在会话期间在节点上保持一致，但不能保证它们在 DFuse 重新启动或跨节点时保持一致。

It is not possible to see pool/container listings through DFuse. So if readdir, ls or others are used, DFuse will return ENOTSUP.

无法通过 DFuse 查看池/容器列表。 因此，如果使用了 readdir、ls 或其他，DFuse 将返回 ENOTSUP。

##### Launching[¶](https://docs.daos.io/user/posix/#launching)

DFuse should be run with the credentials (user/group) of the user who will be accessing it, and who owns any pools that will be used.

DFuse 应该使用将访问它的用户的凭据（用户/组）运行，并且拥有将使用的任何池。

There are two mandatory command-line options, these are:

有两个强制性命令行选项，它们是

|  |  |
| --- | --- |
| **Command-line Option** | **Description** |
| --mountpoint=<path> | path to mount dfuse |

The mount point specified should be an empty directory on the local node that is owned by the user.

指定的挂载点应该是用户拥有的本地节点上的空目录。

Additionally, there are several optional command-line options:

此外，还有几个可选的命令行选项：

|  |  |
| --- | --- |
| **Command-line Option** | **Description** |
| --pool=<uuid> | pool uuid to connect to |
| --container=<uuid> | container uuid to open |
| --sys-name=<name> | DAOS system name |
| --foreground | run in foreground |
| --singlethreaded | run single threaded |

When DFuse starts, it will register a single mount with the kernel, at the location specified by the --mountpoint option. This mount will be visible in /proc/mounts, and possibly in the output of df.  
The contents of multiple pools/containers will be accessible via this single kernel mountpoint.

当 DFuse 启动时，它将在内核中注册一个挂载，在 --mountpoint 选项指定的位置。 该挂载将在 /proc/mounts 中可见，并且可能在 df 的输出中可见。 多个池/容器的内容可以通过这个单一的内核挂载点访问。

##### Pool/Container Paths[¶](https://docs.daos.io/user/posix/#poolcontainer-paths)

DFuse will only create one kernel level mount point regardless of how it is launched. How POSIX containers are represented within that mount point varies depending on the DFuse command-line options:

DFuse 将只创建一个内核级挂载点，而不管它是如何启动的。 POSIX 容器在该挂载点内的表示方式因 DFuse 命令行选项而异：

If both a pool uuid and a container uuid are specified on the command line, then the mount point will map to the root of the container itself. Files can be accessed by simply concatenating the mount point and the name of the file, relative to the root of the container.

如果在命令行中同时指定了池 uuid 和容器 uuid，则挂载点将映射到容器本身的根。 文件可以通过简单地连接挂载点和文件名来访问，相对于容器的根。

If neither a pool or container is specified, then pools and container can be accessed by the path <mount point>/<pool uuid>/<container uuid>. However it should be noted that readdir() and therefore ls do not work on either mount points or directories representing pools here. So the pool and container uuids will have to be provided from an external source.

如果未指定池或容器，则可以通过路径 <mount point>/<pool uuid>/<container uuid> 访问池和容器。 但是应该注意的是， readdir() 和 ls 不适用于此处表示池的挂载点或目录。 因此，池和容器 uuid 必须从外部来源提供。

If a pool uuid is specified but not a container uuid, then the containers can be accessed by the path <mount point>/<container uuid>. The container uuid will have to be provided from an external source.

如果指定了池 uuid 但未指定容器 uuid，则可以通过路径 <mount point>/<container uuid> 访问容器。 容器 uuid 必须从外部来源提供。

It is anticipated that in most cases, both pool uuid and container uuid will be used, so the mount point itself will map directly onto a POSIX container.

预计在大多数情况下，池 uuid 和容器 uuid 都会被使用，因此挂载点本身将直接映射到 POSIX 容器上。

##### Links into other Containers[¶](https://docs.daos.io/user/posix/#links-into-other-containers)

It is possible to link to other containers in DFuse, where subdirectories within a container resolve not to regular directories, but rather to the root of entirely different POSIX containers.

可以链接到 DFuse 中的其他容器，其中容器内的子目录不解析为常规目录，而是解析为完全不同的 POSIX 容器的根目录。

To create a new container and link it into the namespace of an existing one, use the following command.

要创建一个新容器并将其链接到现有容器的命名空间，请使用以下命令。

$ daos container create --type POSIX --pool <pool uuid> --path <path to entry point>

The pool uuid should already exist, and the path should specify a location somewhere within a DFuse mount point that resolves to a POSIX container. Once a link is created, it can be accessed through the new path. Following the link is virtually transparent. No container uuid is required. If one is not supplied, it will be created.

To destroy a container again, the following command should be used.

$ daos container destroy --path <path to entry point>

This will both remove the link between the containers and remove the container that was linked to.

There is no support for adding links to already existing containers or removing links to containers without also removing the container itself.

Information about a container, for example, the presence of an entry point between containers, or the pool and container uuids of the container linked to can be read with the following command.

$ daos container info --path <path to entry point>

##### Enabling Caching[¶](https://docs.daos.io/user/posix/#enabling-caching)

DFuse in normal mode simply provides a communication path between the kernel and DAOS. However, this can come with a performance impact. To help alleviate this it is possible to turn on caching, both within dfuse itself and by allowing the kernel to cache certain data. Where and when data is cached, there is no attempt made to invalidate the caches based on changes to DAOS, other than simple timeouts.

正常模式下的 DFuse 只是提供了内核和 DAOS 之间的通信路径。 但是，这可能会影响性能。 为了帮助缓解这种情况，可以在 dfuse 本身内以及通过允许内核缓存某些数据来打开缓存。 无论何时何地缓存数据，除了简单的超时之外，不会根据对 DAOS 的更改尝试使缓存无效。

Enabling this option will turn on the following features:

* Kernel caching of dentries
* Kernel caching of negative dentries
* Kernel caching of inodes (file sizes, permissions etc)
* Kernel caching of file contents
* Readahead in dfuse and inserting data into kernel cache
* MMAP write optimization

启用此选项将打开以下功能：

• dentry 的内核缓存

• 负 dentry 的内核缓存

• inode 的内核缓存（文件大小、权限等）

• 文件内容的内核缓存

• 在 dfuse 中预读并将数据插入内核缓存

• MMAP 写入优化

To turn on caching use the --enable-caching command-line option for dfuse. This will enable the feature for all accessed containers. When this option is used, the containers accessed should only be accessed from one node, so it may be necessary to create a container per node in this model.

要打开缓存，请使用 dfuse 的 --enable-caching 命令行选项。 这将为所有访问的容器启用该功能。 使用此选项时，访问的容器只能从一个节点访问，因此可能需要在此模型中为每个节点创建一个容器。

##### Stopping DFuse[¶](https://docs.daos.io/user/posix/" \l "stopping-dfuse" \o "Permanent link)

When done, the file system can be unmounted via fusermount:

$ fusermount3 -u /tmp/daos

When this is done, the local DFuse daemon should shut down the mount point, disconnect from the DAOS servers, and exit. You can also verify that the mount point is no longer listed in /proc/mounts.

完成后，本地 DFuse 守护程序应关闭挂载点，断开与 DAOS 服务器的连接，然后退出。 您还可以验证挂载点是否不再列在 /proc/mounts 中。

##### Interception Library[¶](https://docs.daos.io/user/posix/#interception-library)

An interception library called libioil is available to work with DFuse. This library works in conjunction with DFuse and allows the interception of POSIX I/O calls and issue the I/O operations directly from the application context through libdaos without any application changes. This provides kernel-bypass for I/O data, leading to improved performance. To use this, set LD\_PRELOAD to point to the shared library in the DAOS install directory:

一个名为 libioil 的拦截库可用于 DFuse。 该库与 DFuse 结合使用，允许拦截 POSIX I/O 调用，并通过 libdaos 直接从应用程序上下文发出 I/O 操作，无需任何应用程序更改。 这为 I/O 数据提供了内核旁路，从而提高了性能。 要使用它，请将 LD\_PRELOAD 设置为指向 DAOS 安装目录中的共享库：

LD\_PRELOAD=/path/to/daos/install/lib/libioil.so

LD\_PRELOAD=/usr/lib64/libioil.so # when installed from RPMs

#### MPI-IO Support[¶](https://docs.daos.io/user/mpi-io/#mpi-io-support)

The Message Passing Interface (MPI) Standard, maintained by the [MPI Forum](https://www.mpi-forum.org/docs/), includes a chapter on MPI-IO.

[ROMIO](https://www.mcs.anl.gov/projects/romio/) is a well-known implementation of MPI-IO and is included in many MPI implementations. DAOS provides its own MPI-IO ROMIO ADIO driver. This driver has been merged in the upstream MPICH repository, see https://github.com/pmodels/mpich/tree/main/src/mpi/romio/adio/ad\_daos for details.

由 MPI 论坛维护的消息传递接口 (MPI) 标准包括有关 MPI-IO 的一章。 ROMIO 是众所周知的 MPI-IO 实现，并包含在许多 MPI 实现中。 DAOS 提供了自己的 MPI-IO ROMIO ADIO 驱动程序。 此驱动已合并到上游 MPICH 存储库中，详情请参见 https://github.com/pmodels/mpich/tree/main/src/mpi/romio/adio/ad\_daos。

Note

Starting with DAOS 1.2, the --svc parameter (number of service replicas) is no longer needed, and the DAOS API has been changed accordingly Patches have been contributed to MPICH that detect the DAOS API version to gracefully handle this change, but those patches have not yet been picked up in the MPI releases below. For details check the latest commits [here](https://github.com/pmodels/mpich/commits/main?author=mchaarawi).

##### MPI Implementations that support DAOS

###### MPICH[¶](https://docs.daos.io/user/mpi-io/#mpich)

The DAOS ROMIO ADIO driver has been accepted into [MPICH](https://www.mpich.org/). It is included in [mpich-3.4.1 (released Jan 2021)](https://www.mpich.org/downloads/).

###### Building MPICH with DAOS Support[¶](https://docs.daos.io/user/mpi-io/#building-mpich-with-daos-support)

To build MPICH, including ROMIO with the DAOS ADIO driver:

export MPI\_LIB=""

git clone https://github.com/daos-stack/mpich

cd mpich

./autogen.sh

mkdir build; cd build

../configure --prefix=dir --enable-fortran=all --enable-romio \

--enable-cxx --enable-g=all --enable-debuginfo --with-device=ch3:sock \

--with-file-system=ufs+daos --with-daos=/usr --with-cart=/usr

make -j8; make install

This assumes that DAOS is installed into the /usr tree, which is the case for the DAOS RPM installation.

Note

In DAOS 1.0, CART was packaged separately from DAOS, and the --with-cart option was needed to allow separate installation locations. With DAOS 1.1 or higher, CART is included in the main DAOS packages and the --with-cart option is no longer needed. An MPICH pull request is in process to remove the option from the MPICH configure step, but currently it is still required to build MPICH.

Set the PATH and LD\_LIBRARY\_PATH to where you want to build your client apps or libs that use MPI to the path of the installed MPICH.

###### Intel MPI[¶](https://docs.daos.io/user/mpi-io/#intel-mpi)

The [Intel MPI Library](https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html) includes DAOS support since the [2019.8 release](https://software.intel.com/content/www/us/en/develop/articles/intel-mpi-library-release-notes-linux.html).

Note that Intel MPI uses libfabric (both 2019.8 and 2019.9 use libfabric-1.10.1-impi). Care must be taken to ensure that the version of libfabric that is used is at a level that includes the patches that are critical for DAOS. DAOS 1.0.1 includes libfabric-1.9.0, and the DAOS 1.2 releases includes libfabric-1.12.

To use DAOS 1.1 with Intel MPI 2019.8 or 2019.9, the libfabric that is supplied by DAOS (and that is installed into /usr/lib64 by default) needs to be used by listing it first in the library search path:

export LD\_LIBRARY\_PATH="/usr/lib64/:$LD\_LIBRARY\_PATH"

The next Intel MPI release is expected to contain a version of libfabric that includes all patches to work with DAOS. It will then no longer be necessary to override the libfabric version that is shipped with Intel MPI by the version provided by DAOS.

###### Open MPI[¶](https://docs.daos.io/user/mpi-io/#open-mpi)

[Open MPI](https://www.open-mpi.org/) 4.0.5 does not yet provide DAOS support. Since one of its MPI-IO implementations is based on ROMIO, it will likely pick up DAOS support in an upcoming release.

###### MVAPICH2[¶](https://docs.daos.io/user/mpi-io/#mvapich2)

[MVAPICH2](https://mvapich.cse.ohio-state.edu/) 2.3.4 does not yet provide DAOS support. Since its MPI-IO implementation is based on ROMIO, it will likely pick up DAOS support in an upcoming release.

###### Testing MPI-IO with DAOS Support[¶](https://docs.daos.io/user/mpi-io/#testing-mpi-io-with-daos-support)

Build any client (HDF5, ior, mpi test suites) normally with the mpicc command and mpich library installed above (see child pages).

To run an example with MPI-IO:

1. Create a DAOS pool on the DAOS server(s). This will return a pool uuid "puuid".
2. Create a POSIX type container: daos cont create --pool=puuid --type=POSIX This will return a container uuid "cuuid".
3. At the client side, the following environment variables need to be set: export DAOS\_POOL=puuid; export DAOS\_CONT=cuuid. Alternatively, the unified namespace mode can be used instead.
4. Run the client application or test. MPI-IO applications should work seamlessly by just prepending daos: to the filename/path to use the DAOS ADIO driver.

##### Known limitations[¶](https://docs.daos.io/user/mpi-io/#known-limitations)

Limitations of the current implementation include:

* No support for MPI file atomicity, preallocate, or shared file pointers.

#### HDF5 Support

#### [Spark and Hadoop](https://docs.daos.io/user/spark/)

## [Developer Guide](https://github.com/daos-stack/daos/blob/master/src/README.md)

Overview of the DAOS internal code structure and major algorithms for DAOS developers.

DAOS开发人员的DAOS内部代码结构和主要算法概述。

#### DAOS Internals

The purpose of this document is to describe the internal code structure and major algorithms used by DAOS. It assumes prior knowledge of the [DAOS storage model](https://github.com/daos-stack/daos/blob/master/doc/overview/storage.md) and [acronyms](https://github.com/daos-stack/daos/blob/master/doc/overview/terminology.md). This document contains the following sections:

本文档的目的是描述 DAOS 使用的内部代码结构和主要算法。 它假定 DAOS 存储模型和首字母缩略词的先验知识。 本文档包含以下部分：

* [DAOS Components](https://github.com/daos-stack/daos/blob/master/src/README.md#1)
  + [DAOS System](https://github.com/daos-stack/daos/blob/master/src/README.md#11)
  + [Client APIs, Tools and I/O Middleware](https://github.com/daos-stack/daos/blob/master/src/README.md#12)
  + [Agent](https://github.com/daos-stack/daos/blob/master/src/README.md#13)
* [Network Transport and Communications](https://github.com/daos-stack/daos/blob/master/src/README.md#2)
  + [gRPC and Protocol Buffers](https://github.com/daos-stack/daos/blob/master/src/README.md#21)
  + [dRPC](https://github.com/daos-stack/daos/blob/master/src/README.md#22)
  + [CART](https://github.com/daos-stack/daos/blob/master/src/README.md#23)
* [DAOS Layering and Services](https://github.com/daos-stack/daos/blob/master/src/README.md#3)
  + [Architecture](https://github.com/daos-stack/daos/blob/master/src/README.md#31)
  + [Code Structure](https://github.com/daos-stack/daos/blob/master/src/README.md#32)
  + [Infrastructure Libraries](https://github.com/daos-stack/daos/blob/master/src/README.md#33)
  + [DAOS Services](https://github.com/daos-stack/daos/blob/master/src/README.md#34)
* [Software compatibility](https://github.com/daos-stack/daos/blob/master/src/README.md#4)
  + [Protocol Compatibility](https://github.com/daos-stack/daos/blob/master/src/README.md#41)
  + [PM Schema Compatibility and Upgrade](https://github.com/daos-stack/daos/blob/master/src/README.md#42)

• DAOS 组件

o DAOS系统

o 客户端 API、工具和 I/O 中间件

o 代理

• 网络传输和通信

o gRPC 和协议缓冲区 dRPC

o 购物车

• DAOS 分层和服务

o 架构

o 代码结构

o 基础设施库

o DAOS 服务

• 软件兼容性

o 协议兼容性

o PM 架构兼容性和升级

##### DAOS Components

As illustrated in the diagram below, a DAOS installation involves several components that can be either colocated or distributed. The DAOS software-defined storage (SDS) framework relies on two different communication channels: an out-of-band TCP/IP network for management and a high-performant fabric for data access. In practice, the same network can be used for both management and data access. IP over fabric can also be used as the management network.

如下图所示，DAOS 安装涉及多个组件，这些组件可以位于同一位置，也可以分布式。 DAOS 软件定义存储 (SDS) 框架依赖于两个不同的通信通道：用于管理的带外 TCP/IP 网络和用于数据访问的高性能结构。 实际上，同一网络可用于管理和数据访问。 IP over Fabric 也可以用作管理网络。

图示

描述已自动生成

##### DAOS System

A DAOS server is a multi-tenant daemon running on a Linux instance (i.e. physical node, VM or container) and managing the locally-attached SCM and NVM storage allocated to DAOS. It listens to a management port, addressed by an IP address and a TCP port number, plus one or more fabric endpoints, addressed by network URIs. The DAOS server is configured through a YAML file (/etc/daos/daos\_server.yml, or a different path provided on the command line). Starting and stopping the DAOS server can be integrated with different daemon management or orchestration frameworks (e.g. a systemd script, a Kubernetes service or even via a parallel launcher like pdsh or srun).

DAOS 服务器是在 Linux 实例（即物理节点、VM 或容器）上运行的多租户守护进程，并管理分配给 DAOS 的本地附加 SCM 和 NVM 存储。 它侦听由 IP 地址和 TCP 端口号寻址的管理端口，以及由网络 URI 寻址的一个或多个结构端点。 DAOS 服务器通过 YAML 文件（/etc/daos/daos\_server.yml，或命令行上提供的不同路径）进行配置。 启动和停止 DAOS 服务器可以与不同的守护进程管理或编排框架（例如 systemd 脚本、Kubernetes 服务，甚至通过 pdsh 或 srun 等并行启动器）集成。

A DAOS system is identified by a system name and consists of a set of DAOS servers connected to the same fabric. Two different systems comprise two disjoint sets of servers and do not coordinate with each other. DAOS pools cannot span across multiple systems.

DAOS 系统由系统名称标识，由一组连接到同一结构的 DAOS 服务器组成。 两个不同的系统包含两组不相交的服务器并且彼此不协调。 DAOS 池不能跨越多个系统。

Internally, a DAOS server is composed of multiple daemon processes. The first one to be started is the [control plane](https://github.com/daos-stack/daos/blob/master/src/control/README.md) (binary named daos\_server) which is responsible for parsing the configuration file, provisionning storage and eventually starting and monitoring one or multiple instances of the [data plane](https://github.com/daos-stack/daos/blob/master/src/engine/README.md) (binary named daos\_engine). The control plane is written in Go and implements the DAOS management API over the gRPC framework that provides a secured out-of-band channel to administrate a DAOS system. The number of data plane instances to be started by each server as well as the storage, CPU and fabric interface affinity can be configured through the daos\_server.yml YAML configuration file.

在内部，DAOS 服务器由多个守护进程组成。 第一个要启动的是控制平面（名为 daos\_server 的二进制文件），它负责解析配置文件、提供存储并最终启动和监控数据平面（名为 daos\_engine 的二进制文件）的一个或多个实例。 控制平面是用 Go 编写的，并在 gRPC 框架上实现了 DAOS 管理 API，该框架提供了一个安全的带外通道来管理 DAOS 系统。 每个服务器要启动的数据平面实例的数量以及存储、CPU 和结构接口亲和性可以通过 daos\_server.yml YAML 配置文件进行配置。

The data plane is a multi-threaded process written in C that runs the DAOS storage engine. It processes incoming metadata and I/O requests though the CART communication middleware and accesses local NVM storage via the PMDK (for storage-class memory, aka SCM) and SPDK (for NVMe SSDs) libraries. The data plane relies on Argobots for event-based parallel processing and exports multiple targets that can be independently addressed via the fabric. Each data plane instance is assigned a unique rank inside a DAOS system.

数据平面是一个用 C 语言编写的多线程进程，它运行着 DAOS 存储引擎。 它通过 CART 通信中间件处理传入的元数据和 I/O 请求，并通过 PMDK（用于存储级内存，又名 SCM）和 SPDK（用于 NVMe SSD）库访问本地 NVM 存储。 数据平面依赖 Argobots 进行基于事件的并行处理，并导出可通过结构独立寻址的多个targets。 每个数据平面实例在 DAOS 系统内都被分配了一个唯一的rank。

The control plane and data plane processes communicate locally through Unix Domain Sockets and a custom lightweight protocol called dRPC.

控制平面和数据平面进程通过 Unix 域套接字和称为 dRPC 的自定义轻量级协议在本地进行通信。

For further reading:/ 进一步阅读：

###### [DAOS control plane (daos\_server)](https://github.com/daos-stack/daos/blob/master/src/control/README.md)/ DAOS 控制平面

DAOS Control Plane/ DAOS 控制平面

DAOS operates over two, closely integrated planes, Control and Data. The Data plane handles the heavy lifting transport operations while the Control plane orchestrates process and storage management, facilitating the operation of the Data plane.

DAOS 在两个紧密集成的平面上运行，控制和数据。 数据平面处理繁重的传输操作，而控制平面协调流程和存储管理，促进数据平面的操作。

The DAOS Control Plane is written in Go and runs as the DAOS Server (daos\_server) process. It is tasked with network and storage hardware provisioning and allocation in addition to instantiation and management of the DAOS Data Plane (Engine) processes that run on the same host.

DAOS 控制平面是用 Go 编写的，并作为 DAOS 服务器（daos\_server）进程运行。 除了在同一主机上运行的 DAOS 数据平面（引擎）进程的实例化和管理之外，它还负责网络和存储硬件的配置和分配。

Code Organization/代码组织

The control directory contains a "cmd" subdirectory for server, agent, and dmg applications. These applications import the control API (src/control/lib/control) or server packages along with peripheral shared packages common, drpc, fault, logging, and security where necessary to provide the given features.

控制目录包含用于服务器、代理和 dmg 应用程序的“cmd”子目录。 这些应用程序导入控制 API (src/control/lib/control) 或服务器包以及外围共享包 common、drpc、fault、logging 和 security，以提供给定的功能。

Specific library packages can be found in lib/ which provide access to native storage libraries through language bindings, e.g. lib/spdk or specific formatting capabilities e.g., lib/hostlist or lib/txtfmt.

可以在 lib/ 中找到特定的库包，它们通过语言绑定提供对本机存储库的访问，例如 lib/spdk 或特定的格式化功能，例如 lib/hostlist 或 lib/txtfmt。

The events package provides the golang component of the RAS framework for receipt of events over dRPC from the DAOS Engine and forwarding of management service actionable events to the MS leader.

事件包提供了 RAS 框架的 golang 组件，用于从 DAOS 引擎通过 dRPC 接收事件并将管理服务可操作事件转发给 MS 领导者。

The pbin package provides a framework for forwarding of requests to be executed by the privileged binary daos\_admin on behalf of daos\_server.

pbin 包提供了一个框架，用于转发由特权二进制 daos\_admin 代表 daos\_server 执行的请求。

The provider package contains interface shims to the external environment, initially just to the Linux operating system.

The system package encapsulates the concept of the DAOS system, and its associated membership.

提供程序包包含外部环境的接口垫片，最初仅用于 Linux 操作系统。

系统包封装了 DAOS 系统的概念及其关联的成员资格。

Developer Documentation

Please refer to package-specific README's.

DAOS Server

The server package implements the internals of the DAOS Server, and the daos\_server user-facing application is implemented by the [daos\_server](https://github.com/daos-stack/daos/blob/master/src/control/cmd/daos_server/README.md) package.

server 包实现了 DAOS Server 的内部结构，daos\_server 面向用户的应用程序由 daos\_server 包实现。

I/O Engine Instances

DAOS I/O Engine processes (daos\_engine binary) are forked by the DAOS Control Server (daos\_server binary) and perform the main userspace I/O operations of DAOS. [instance.go](https://github.com/daos-stack/daos/blob/master/src/control/server/instance.go) provides the EngineInstance abstraction and relevant methods.

DAOS I/O 引擎进程（daos\_engine 二进制）由 DAOS 控制服务器（daos\_server 二进制）派生，并执行 DAOS 的主要用户空间 I/O 操作。 instance.go 提供了 EngineInstance 抽象和相关方法。

Underlying abstractions for the control and configuration of the I/O Engine processes are encapsulated in the [engine](https://github.com/daos-stack/daos/blob/master/src/control/server/engine) package.

I/O 引擎进程的控制和配置的底层抽象封装在引擎包中。

I/O Engine Harness

DAOS I/O Engine processes are managed and monitored by the DAOS Server and logically reside as members of the I/O Engine harness. [harness.go](https://github.com/daos-stack/daos/blob/master/src/control/server/harness.go) provides the EngineHarness abstraction and relevant methods.

DAOS I/O 引擎进程由 DAOS 服务器管理和监视，并在逻辑上作为 I/O 引擎线束的成员驻留。 Harness.go 提供了 EngineHarness 抽象和相关方法。

Communications/通讯

The DAOS Server implements the [gRPC protocol](https://grpc.io/) to communicate with client gRPC applications and interacts with DAOS I/O Engines through Unix domain sockets.

Multiple gRPC server modules are loaded by DAOS Server, currently included modules are security and management.

DAOS Server 实现 gRPC 协议以与客户端 gRPC 应用程序通信，并通过 Unix 域套接字与 DAOS I/O 引擎交互。

DAOS Server 加载了多个 gRPC 服务器模块，目前包含的模块是安全和管理。

DAOS Server (daos\_server) instances will open a gRPC channel to listen for requests from control-plane client applications and other DAOS Server instances.

[server.go](https://github.com/daos-stack/daos/blob/master/src/control/server/server.go) contains main setup routines, including the establishment of the gRPC server and registering of RPCs.

DAOS Server (daos\_server) 实例将打开一个 gRPC 通道来侦听来自控制平面客户端应用程序和其他 DAOS Server 实例的请求。

server.go 包含主要的设置例程，包括 gRPC 服务器的建立和 RPC 的注册。

Control API

The [control](https://github.com/daos-stack/daos/blob/master/src/control/lib/control/README.md) package exposes an RPC-based API for control-plane client applications to communicate with DAOS Server processes.

[control](https://github.com/daos-stack/daos/blob/master/src/control/lib/control/README.md)包公开了一个基于 RPC 的 API，用于控制平面客户端应用程序与 DAOS 服务器进程进行通信。

Protobuf Definitions

Protobuf definitions are described in the [proto](https://github.com/daos-stack/daos/blob/master/src/proto/README.md) directory.

protobuf 定义在 proto 目录中描述。

Control Service

The gRPC server registers the [control service](https://github.com/daos-stack/daos/blob/master/src/proto/ctl/control.proto) to handle requests from the management tool.

gRPC 服务器注册控制服务来处理来自管理工具的请求。

Control service requests are operations that will be performed on one or more daos\_server processes in parallel, such as hardware provisioning. The handlers triggered on receipt of control service RPCs will typically end-up calling into native-C storage or network libraries through the relevant go bindings e.g. [ipmctl](https://github.com/daos-stack/daos/blob/master/src/control/lib/ipmctl/README.md), [spdk](https://github.com/daos-stack/daos/blob/master/src/control/lib/ipmctl/README.md) or [netdetect](https://github.com/daos-stack/daos/blob/master/src/control/lib/netdetect/README.md).

控制服务请求是将在一个或多个 daos\_server 进程上并行执行的操作，例如硬件供应。 在接收到控制服务 RPC 时触发的处理程序通常最终会通过相关的 go 绑定调用原生 C 存储或网络库，例如 ipmctl、spdk 或 netdetect。

Such broadcast commands (which will be issued after connecting to a list of hosts) will usually be issued by the [management tool](https://github.com/daos-stack/daos/blob/master/src/control/cmd/dmg/README.md), a gRPC client that communicates with daos\_server processes through the control API.

此类广播命令（将在连接到主机列表后发出）通常由管理工具发出，即通过控制 API 与 daos\_server 进程通信的 gRPC 客户端。

These commands will not usually trigger dRPCs and will mostly perform functions such as hardware (network and storage) provisioning.

这些命令通常不会触发 dRPC，主要执行硬件（网络和存储）配置等功能。

The control service RPC handler code is contained in /src/control/server/ctl\_\*.go files and protobuf specific un/wrapping code in /src/control/server/ctl\_\*\_rpc.go files.

控制服务 RPC 处理程序代码包含在 /src/control/server/ctl\_\*.go 文件中，protobuf 特定的解包代码包含在 /src/control/server/ctl\_\*\_rpc.go 文件中。

Management Service/管理服务

The Control Plane implements a management service as part of the DAOS Server, responsible for handling distributed operations across the DAOS System.

控制平面作为 DAOS 服务器的一部分实施管理服务，负责处理跨 DAOS 系统的分布式操作。

Some dmg commands will trigger MS requests to be issued to a daos\_server process on a storage node running as the MS leader, this happens under the hood and the logic for the request steering is handled in the control API which is utilized by the [dmg](https://github.com/daos-stack/daos/blob/master/src/control/cmd/dmg/README.md) tool.

一些 dmg 命令将触发 MS 请求发送到作为 MS 领导者运行的存储节点上的 daos\_server 进程，这发生在幕后，请求转向的逻辑在 dmg 工具使用的控制 API 中处理。

When necessary, requests will be forwarded to the data plane [engine](https://github.com/daos-stack/daos/blob/master/src/engine/README.md) over dRPC channel and handled by the [mgmt](https://github.com/daos-stack/daos/blob/master/src/mgmt/srv.c) module.

MS RPC related code is contained in /src/control/server/mgmt\_\*.go files.

必要时，请求将通过 dRPC 通道转发到数据平面引擎并由 mgmt 模块处理。 MS RPC 相关代码包含在 /src/control/server/mgmt\_\*.go 文件中。

Server-to-server Fan-out

Some control service RPC handlers will trigger fan-out to multiple remote harnesses over gRPC, in order to send these fan-out requests the client in the [control API](https://github.com/daos-stack/daos/blob/master/src/control/lib/control/README.md) is used.

一些控制服务 RPC 处理程序将通过 gRPC 触发对多个远程线束的扇出，以便使用控制 API 中的客户端发送这些扇出请求。

An example of a management tool command that executes gRPC fan-out over multiple remote harnesses is dmg system stop, the server side handler for which is SystemStop in [ctl\_system.go](https://github.com/daos-stack/daos/blob/master/src/control/server/ctl_system.go) which issues requests to remote harnesses. The use of the control API client (which implements the UnaryInvoker interface) to issue a fan-out request is demonstrated in [system.go](https://github.com/daos-stack/daos/blob/master/src/control/lib/control/system.go) SystemStop client call.

在多个远程线束上执行 gRPC 扇出的管理工具命令的一个示例是 dmg system stop，它的服务器端处理程序是 ctl\_system.go 中的 SystemStop，它向远程线束发出请求。 在 system.go SystemStop 客户端调用中演示了使用控制 API 客户端（实现 UnaryInvoker 接口）发出扇出请求。

System Command Handling/系统命令处理

System commands use fan-out and send unary RPCs to selected ranks across the system for actions stop, start and reformat.

系统命令使用扇出并将一元 RPC 发送到整个系统中的选定列，以执行停止、启动和重新格式化的操作。

Storage Command Handling/存储命令处理

Storage related RPCs, whose handlers are defined in [ctl\_storage\*.go](https://github.com/daos-stack/daos/blob/master/src/control/server/ctl_storage.go) delegate operations to backend providers encapsulated in the bdev and scm [storage subsystem packages](https://github.com/daos-stack/daos/blob/master/src/control/server/storage).

与存储相关的 RPC，其处理程序在 ctl\_storage\*.go 中定义，将操作委托给封装在 bdev 和 scm 存储子系统包中的后端提供程序。

Bootstrapping and DAOS System Membership/引导和 DAOS 系统成员资格

When starting a data-plane instance, we look at the superblock to determine whether the instance should be started as a MS (management service) replica. The daos\_server.yml's access\_points parameter is used (only during format) to determine whether an instance is to be a MS replica or not.

启动数据平面实例时，我们会查看超级块以确定该实例是否应作为 MS（管理服务）副本启动。 daos\_server.yml 的 access\_points 参数用于（仅在格式化期间）确定实例是否为 MS 副本。

When the starting instance is identified as an MS replica, it performs bootstrap and starts. If the DAOS system has only one replica (as specified by access\_points parameter), the host of the bootstrapped instance is now the MS leader. Whereas if there are multiple replicas, elections will happen in the background and eventually a leader will be elected.

当起始实例被识别为 MS 副本时，它会执行引导程序并启动。 如果 DAOS 系统只有一个副本（由 access\_points 参数指定），则引导实例的主机现在是 MS 领导者。 而如果有多个副本，选举将在后台进行，最终将选出一个领导者。

When the starting instance is not identified as an MS replica, the instance's host calls Join on the control API client which triggers a gRPC request to the MS leader. The joining instance's control address is populated in the request.

当起始实例未被识别为 MS 副本时，该实例的主机调用控制 API 客户端上的 Join，从而触发对 MS 领导者的 gRPC 请求。 加入实例的控制地址填充在请求中。

The gRPC server running on the MS leader handles the Join request and allocates a DAOS system rank which is recorded in the MS membership (which is backed by the distributed system database). The rank is returned in the Join response and communicated to the data-plane (engine) over dRPC.

在 MS 领导者上运行的 gRPC 服务器处理加入请求并分配一个 DAOS 系统等级，该等级记录在 MS 成员资格中（由分布式系统数据库支持）。 rank在加入响应中返回并通过 dRPC 传送到数据平面（引擎）。

Storage Management/存储管理

Operations on NVMe SSD devices are performed using [go-spdk bindings](https://github.com/daos-stack/daos/blob/master/src/control/lib/spdk) to issue commands through the SPDK framework native C libraries.

Operations on SCM persistent memory modules are performed using [go-ipmctl bindings](https://github.com/daos-stack/daos/blob/master/src/control/lib/ipmctl) to issue commands through the ipmctl native C libraries.

Storage RPC related code which concerns the server-side handling of requests is contained within /src/control/server/ctl\_storage\*.go files.

NVMe SSD 设备上的操作是使用 go-spdk 绑定执行的，通过 SPDK 框架原生 C 库发出命令。

SCM 持久内存模块上的操作是使用 go-ipmctl 绑定执行的，以通过 ipmctl 本机 C 库发出命令。

与服务器端处理请求有关的存储 RPC 相关代码包含在 /src/control/server/ctl\_storage\*.go 文件中。

Storage Format/存储格式

Storage is required to be formatted before the DAOS data plane can be started.

DAOS数据平面启动前，需要对存储进行格式化。

图示

描述已自动生成

If storage has not been previously formatted, daos\_server will halt on start-up waiting for storage format to be triggered by issuing the dmg storage format command.

Storage format is expected only to be performed when setting up the DAOS system for the first time.

如果之前没有格式化存储，daos\_server 将在启动时暂停，等待通过发出 dmg storage format 命令触发存储格式。

预计仅在首次设置 DAOS 系统时执行存储格式。

SCM Format

Formatting SCM involves creating an ext4 filesystem on the nvdimm device. Mounting SCM results in an active mount using the DAX extension enabling direct access without restrictions imposed by traditional block storage.

格式化 SCM 涉及在 nvdimm 设备上创建 ext4 文件系统。 挂载 SCM 会导致使用 DAX 扩展进行主动挂载，从而实现直接访问，而不受传统块存储的限制。

Formatting and mounting of SCM device namespace is performed as specified in config file parameters prefixed with scm\_.

SCM 设备命名空间的格式化和挂载按照以 scm\_ 为前缀的配置文件参数中的指定执行。

NVMe Format

In the context of what is required from the control plane to prepare NVMe devices for operation with DAOS data plane, "formatting" refers to the reset of storage media which will remove blobstores and remove any filesystem signatures from the SSD controller namespaces.

在控制平面需要什么来准备 NVMe 设备以与 DAOS 数据平面一起运行的上下文中，“格式化”是指存储介质的重置，这将删除 blobstore 并从 SSD 控制器命名空间中删除任何文件系统签名。

Formatting will be performed on devices identified by PCI addresses specified in config file parameter bdev\_list when bdev\_class is equal to nvme.

当 bdev\_class 等于 nvme 时，将在配置文件参数 bdev\_list 中指定的 PCI 地址标识的设备上执行格式化。

In order to designate NVMe devices to be used by DAOS data plane instances, the control plane will generate a daos\_nvme.conf file to be consumed by SPDK which will be written to the scm\_mount (persistent) mounted location as a final stage of formatting before the superblock is written, signifying the server has been formatted.

为了指定 DAOS 数据平面实例使用的 NVMe 设备，控制平面将生成一个供 SPDK 使用的 daos\_nvme.conf 文件，该文件将写入 scm\_mount（持久性）挂载位置作为格式化之前的最后阶段 写入超级块，表示服务器已被格式化。

Architecture

A view of DAOS' software component architecture:

DAOS 的软件组件架构图：

图示

描述已自动生成

###### [DAOS data plane (daos\_engine)](https://github.com/daos-stack/daos/blob/master/src/engine/README.md)/ DAOS 数据平面

Module Interface/模块接口

The I/O Engine supports a module interface that allows to load server-side code on demand. Each module is effectively a library dynamically loaded by the I/O Engine via dlopen. The interface between the module and the I/O Engine is defined in the dss\_module data structure.

I/O 引擎支持允许按需加载服务器端代码的模块接口。 每个模块实际上是一个由 I/O 引擎通过 dlopen 动态加载的库。 模块和 I/O 引擎之间的接口在 dss\_module 数据结构中定义。

Each module should specify:

* a module name
* a module identifier from daos\_module\_id
* a feature bitmask
* a module initialization and finalize function

每个模块应指定：

• 模块名称

• 来自 daos\_module\_id 的模块标识符

• 功能位掩码

• 模块初始化和完成功能

In addition, a module can optionally configure:

* a setup and cleanup function invoked once the overall stack is up and running
* CART RPC handlers
* dRPC handlers

此外，模块可以选择配置：

• 整个堆栈启动并运行后调用的设置和清理函数

• CART RPC 处理程序

• dRPC 处理程序

Thread Model & Argobot Integration/线程模型和 Argobot 集成

The I/O Engine is a multi-threaded process using Argobots for non-blocking processing.

I/O 引擎是一个多线程进程，使用 Argobots 进行非阻塞处理。

By default, one main xstream and no offload xstreams are created per target. The actual number of offload xstream can be configured through daos\_engine command line parameters. Moreover, an extra xstream is created to handle incoming metadata requests. Each xstream is bound to a specific CPU core. The main xstream is the one receiving incoming target requests from both client and the other servers. A specific ULT (User-Level Thread) is started to make progress on network and NVMe I/O operations.

默认情况下，每个target创建一个主 xstream 和无卸载 xstream。 offload xstream的实际数量可以通过daos\_engine命令行参数配置。 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 内核。 主 xstream 是从客户端和其他服务器接收传入目标请求的一个。 启动特定的 ULT 以在网络和 NVMe I/O 操作方面取得进展。

Thread-local Storage (TLS)/ 线程本地存储

Each xstream allocates private storage that can be accessed via the dss\_tls\_get() function. When registering, each module can specify a module key with a size of data structure that will be allocated by each xstream in the TLS. The dss\_module\_key\_get() function will return this data structure for a specific registered module key.

每个xstream分配可以通过dss\_tls\_get（）函数访问的私有存储。注册时，每个模块都可以指定一个具有数据结构大小的模块key，该数据结构将由TLS中的每个xstream分配。dss\_module\_key\_get（）函数将返回特定注册模块key的此数据结构。

Incast Variable Integration/ Incast 变量集成

DAOS uses IV (incast variable) to share values and statuses among servers under a single IV namespace, which is organized as a tree. The tree root is called IV leader, and servers can either be leaves or non-leaves. Each server maintains its own IV cache. During fetch, if the local cache can not fulfill the request, it forwards the request to its parents, until reaching the root (IV leader). As for update, it updates its local cache first, then forwards to its parents until it reaches the root, which then propagate the changes to all the other servers. The IV namespace is per pool, which is created during pool connection, and destroyed during pool disconnection. To use IV, each user needs to register itself under the IV namespace to get an identification, then it will use this ID to fetch or update its own IV value under the IV namespace.

DAOS 使用 IV（incast 变量）在单个 IV 命名空间下的服务器之间共享值和状态，该命名空间被组织为树。 树根称为IV领导者，服务器可以是叶子或非叶子。 每个服务器都维护自己的 IV 缓存。 在 fetch 期间，如果本地缓存不能满足请求，它会将请求转发给它的父级，直到到达根（IV 领导者）。 至于更新，它首先更新它的本地缓存，然后转发到它的父级直到它到达根，然后将更改传播到所有其他服务器。 IV 命名空间是每个池的，它在池连接期间创建，并在池断开连接期间销毁。 要使用IV，每个用户需要在IV命名空间下注册自己以获得一个标识，然后它会使用这个ID在IV命名空间下获取或更新自己的IV值。

dRPC Server

The I/O Engine includes a dRPC server that listens for activity on a given Unix Domain Socket. See the [dRPC documentation](https://github.com/daos-stack/daos/blob/master/src/control/drpc/README.md) for more details on the basics of dRPC, and the low-level APIs in Go and C.

I/O 引擎包括一个 dRPC 服务器，它侦听给定 Unix 域套接字上的活动。 有关 dRPC 基础知识以及 Go 和 C 中的低级 API 的更多详细信息，请参阅 dRPC 文档。

The dRPC server polls periodically for incoming client connections and requests. It can handle multiple simultaneous client connections via the struct drpc\_progress\_context object, which manages the struct drpc objects for the listening socket as well as any active client connections.

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 对象处理多个并发客户端连接，该对象管理用于侦听套接字的 struct drpc 对象以及任何活动的客户端连接。

The server loop runs in its own User-Level Thread (ULT) in xstream 0. The dRPC socket has been set up as non-blocking and polling uses timeouts of 0, which allows the server to run in a ULT rather than its own xstream. This channel is expected to be relatively low-traffic.

服务器循环在 xstream 0 中自己的用户级线程 (ULT) 中运行。 dRPC 套接字已设置为非阻塞，轮询使用超时 0，这允许服务器在 ULT 中运行，而不是在其自己的 xstream 中运行 . 预计该渠道的流量相对较低。

dRPC Progress

drpc\_progress represents one iteration of the dRPC server loop. The workflow is as follows:

drpc\_progress 代表 dRPC 服务器循环的一次迭代。 工作流程如下：

1. Poll with a timeout on the listening socket and any open client connections simultaneously. 同时轮询侦听套接字和任何打开的客户端连接超时。
2. If any activity is seen on a client connection:
   1. If data has come in: Call drpc\_recv to process the incoming data.
   2. If the client has disconnected or the connection has been broken: Free the struct drpc object and remove it from the drpc\_progress\_context.

如果在客户端连接上看到任何活动：

1、如果有数据进来：调用drpc\_recv处理进来的数据。 2.如果客户端断开连接或连接已断开：释放struct drpc对象并将其从drpc\_progress\_context中删除。

1. If any activity is seen on the listener:
   1. If a new connection has come in: Call drpc\_accept and add the new struct drpc object to the client connection list in the drpc\_progress\_context.
   2. If there was an error: Return -DER\_MISC to the caller. This causes an error to be logged in the I/O Engine, but does not interrupt the dRPC server loop. Getting an error on the listener is unexpected.

如果在监听器上看到任何活动：

1. 如果有新连接进来：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。

2. 如果出现错误：将 -DER\_MISC 返回给调用者。 这会导致在 I/O 引擎中记录错误，但不会中断 dRPC 服务器循环。 在侦听器上出现错误是意料之外的。

1. If no activity was seen, return -DER\_TIMEDOUT to the caller. This is purely for debugging purposes. In practice the I/O Engine ignores this error code, since lack of activity is not actually an error case. 如果没有看到任何活动，将 -DER\_TIMEDOUT 返回给调用者。 这纯粹是为了调试目的。 实际上，I/O 引擎会忽略此错误代码，因为缺乏活动实际上并不是错误情况。

dRPC Handler Registration

Individual DAOS modules may implement handling for dRPC messages by registering a handler function for one or more dRPC module IDs.

单个 DAOS 模块可以通过为一个或多个 dRPC 模块 ID 注册处理程序函数来实现对 dRPC 消息的处理。

Registering handlers is simple. In the dss\_server\_module field sm\_drpc\_handlers, statically allocate an array of struct dss\_drpc\_handler with the last item in the array zeroed out to indicate the end of the list. Setting the field to NULL indicates nothing to register. When the I/O Engine loads the DAOS module, it will register all of the dRPC handlers automatically.

注册处理程序很简单。 在 dss\_server\_module 字段 sm\_drpc\_handlers 中，静态分配一个 struct dss\_drpc\_handler 数组，数组中的最后一项被清零以指示列表的结尾。 将该字段设置为 NULL 表示没有要注册的内容。 当 I/O 引擎加载 DAOS 模块时，它会自动注册所有 dRPC 处理程序。

**Note:** The dRPC module ID is **not** the same as the DAOS module ID. This is because a given DAOS module may need to register more than one dRPC module ID, depending on the functionality it covers. The dRPC module IDs must be unique system-wide and are listed in a central header file: src/include/daos/drpc\_modules.h

注意：dRPC 模块 ID 与 DAOS 模块 ID 不同。 这是因为一个给定的 DAOS 模块可能需要注册多个 dRPC 模块 ID，具体取决于它所涵盖的功能。 dRPC 模块 ID 必须在系统范围内唯一，并列在中央头文件中：src/include/daos/drpc\_modules.h

The dRPC server uses the function drpc\_hdlr\_process\_msg to handle incoming messages. This function checks the incoming message's module ID, searches for a handler, executes the handler if one is found, and returns the Drpc\_\_Response. If none is found, it generates its own Drpc\_\_Response indicating the module ID was not registered.

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，如果找到则执行处理程序，并返回 Drpc\_\_Response。 如果没有找到，它会生成自己的 Drpc\_\_Response 指示模块 ID 未注册。

##### Client APIs, Tools and I/O Middleware/客户端 API、工具和 I/O 中间件

Applications, users and administrators can interact with a DAOS system through two different client APIs. 应用程序、用户和管理员可以通过两个不同的客户端 API 与 DAOS 系统交互。

The DAOS management Go package allows to administrate a DAOS system from any nodes that can communicate with the DAOS servers through the out-of-band management channel. This API is reserved for the DAOS system administrators who are authenticated through a specific certificate. The DAOS management API is intended to be integrated with different vendor-specific storage management or open-source orchestration frameworks. A CLI tool called dmg is built over the DAOS management API. For further reading on the management API and the dmg tool:

DAOS 管理 Go 包允许从可以通过带外管理通道与 DAOS 服务器通信的任何节点管理 DAOS 系统。 此 API 是为通过特定证书进行身份验证的 DAOS 系统管理员保留的。 DAOS 管理 API 旨在与不同供应商特定的存储管理或开源编排框架集成。 名为 dmg 的 CLI 工具是基于 DAOS 管理 API 构建的。 如需进一步阅读管理 API 和 dmg 工具：

###### [DAOS\_management\_Go\_package](https://godoc.org/github.com/daos-stack/daos/src/control/client).doc

###### [DAOS\_Management\_tool(aka\_dmg)](https://github.com/daos-stack/daos/blob/master/src/control/cmd/dmg/README.md).doc

The DAOS library (libdaos) implements the DAOS storage model and is primarily targeted at application and I/O middleware developers who want to store datasets into DAOS containers. It can be used from any nodes connected to the fabric used by the targeted DAOS system. The application process is authenticated via the DAOS agent (see next section). The API exported by libdaos is commonly called the DAOS API (in contrast to the DAOS management API) and allows to manage containers and access DAOS objects through different interfaces (e.g. key-value store or array API). The libdfs library emulates POSIX file and directory abstractions over libdaos and provides a smooth migration path for applications that require a POSIX namespace. For further reading on libdaos, bindings for different programming languages and libdfs:

DAOS 库 (libdaos) 实现了 DAOS 存储模型，主要面向希望将数据集存储到 DAOS 容器中的应用程序和 I/O 中间件开发人员。 它可以从连接到目标 DAOS 系统使用的结构的任何节点使用。 应用程序进程通过 DAOS 代理进行身份验证（请参阅下一节）。 libdaos 导出的 API 通常称为 DAOS API（与 DAOS 管理 API 相对），它允许通过不同的接口（例如键值存储或数组 API）管理容器和访问 DAOS 对象。 libdfs 库通过 libdaos 模拟 POSIX 文件和目录抽象，并为需要 POSIX 命名空间的应用程序提供平滑的迁移路径。 要进一步阅读 libdaos、不同编程语言和 libdfs 的绑定：

* [DAOS Library (libdaos)](https://github.com/daos-stack/daos/blob/master/src/client/api/README.md) and [array interface](https://github.com/daos-stack/daos/blob/master/src/client/array/README.md) and [KV interface](https://github.com/daos-stack/daos/blob/master/src/client/kv/README.md) built on top of the native DAOS API
* [Python API bindings](https://github.com/daos-stack/daos/blob/master/src/src/client/pydaos/raw/README.md)
* [Go bindings](https://github.com/daos-stack/go-daos) and [API documentation](https://godoc.org/github.com/daos-stack/go-daos/pkg/daos)
* [POSIX File & Directory Emulation (libdfs)](https://github.com/daos-stack/daos/blob/master/src/client/dfs/README.md)

• DAOS 库 (libdaos) 以及建立在原生 DAOS API 之上的数组接口和 KV 接口

• Python API 绑定

• Go 绑定和 API 文档

• POSIX 文件和目录仿真 (libdfs)

The libdaos and libdfs libraries provide the foundation to support domain-specific data formats like HDF5 and Apache Arrow. For further reading on I/O middleware integration, please check the following external references:

* [DAOS VOL connector for HDF5](https://bitbucket.hdfgroup.org/projects/HDFFV/repos/hdf5/browse?at=refs%2Fheads%2Fhdf5_daosm)
* [ROMIO DAOS ADIO driver for MPI-IO](https://github.com/daos-stack/mpich/tree/daos_adio)

libdaos 和 libdfs 库为支持特定领域的数据格式（如 HDF5 和 Apache Arrow）提供了基础。 如需进一步阅读 I/O 中间件集成，请查看以下外部参考资料：

• 用于 HDF5 的 DAOS VOL 连接器

• 用于 MPI-IO 的 ROMIO DAOS ADIO 驱动程序

###### [DAOS Library (libdaos)](https://github.com/daos-stack/daos/blob/master/src/client/api/README.md)/ DAOS 库

DAOS Client Library/客户端库

The DAOS API is divided along several functionalities to address the different features that DAOS exposes:

DAOS API 分为几个功能，以解决 DAOS 公开的不同功能：

* Management API: pool and target management
* Pool Client API: pool access
* Container API: container management and access, container snapshots
* 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.

• 管理 API：池和target管理 • 池客户端 API：池访问 • Container API：容器管理和访问、容器快照 • 事务API：事务模型和并发控制 • 对象、数组和 KV API：对象和数据管理和访问 • 事件、事件队列和任务 API：非阻塞操作 • 插件 API：基于 DAOS 对象 API 构建的数组和 KV 操作 • DFS API：DAOS 文件系统 API，用于在 DAOS 上模拟 POSIX 命名空间 • DUNS API：DAOS 统一命名空间 API，用于与现有系统命名空间集成。

Each of those components have associated README.md files that provide more details about the functionality they support except for APIs to support non-blocking operations which is discussed here.

这些组件中的每一个都有关联的 README.md 文件，这些文件提供了有关它们支持的功能的更多详细信息，但支持非阻塞操作的 API 除外，这在此处进行了讨论。

The libdaos API is available under [/src/include/daos\_\*](https://github.com/daos-stack/daos/blob/master/src/include) and associated man pages under [/doc/man/man3/](https://github.com/daos-stack/daos/blob/master/doc/man/man3).

Event & Event Queue/事件和事件队列

DAOS API functions can be used in either blocking or non-blocking mode. This is determined through a pointer to a DAOS event passed to each API call that:

DAOS API 函数可以在阻塞或非阻塞模式下使用。 这是通过一个指向传递给每个 API 调用的 DAOS 事件的指针确定的：

* if NULL indicates that the operation is to be blocking. The operation will return after completing the operation. The error codes for all failure cases will be returned through the return code of the API function itself.
* if a valid event is used, the operation will run in non-blocking mode and return immediately after scheduling the operation in the internal scheduler and after RPCs are submitted to the underlying stack. The return value of the operation is success if the scheduling succeeds, but does not indicate that the actual operation succeeds. The errors that can be caught on return are either invalid parameters or scheduling problems. The actual return code of the operation will be available in the event error code (event.ev\_error) when the event completes.

• 如果NULL 表示操作是阻塞的。 操作完成后将返回操作。 所有失败案例的错误代码将通过 API 函数本身的返回代码返回。

• 如果使用了有效事件，则该操作将在非阻塞模式下运行，并在内部调度程序中调度该操作后以及 RPC 提交到底层堆栈后立即返回。 调度成功则返回操作成功，但不代表实际操作成功。 返回时可以捕获的错误是无效参数或调度问题。 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。

A valid event to be used must be created first with a separate API call. To allow users to track multiple events at a time, an event can be created as part of an event queue, which is basically a collection of events that can be progressed and polled together. Alternatively, an event can be created without an event queue, and be tracked individually. Once an event is completed, it can re-used for another DAOS API call to minimize the need for event creation and allocations inside the DAOS library.

要使用的有效事件必须首先通过单独的 API 调用创建。 为了允许用户一次跟踪多个事件，可以将事件创建为事件队列的一部分，该队列基本上是可以一起进行和轮询的事件的集合。 或者，可以在没有事件队列的情况下创建事件，并单独跟踪。 事件完成后，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要。

Task Engine Integration

任务引擎集成

The DAOS Task API provides an alternative way to use the DAOS API in a non-blocking manner and at the same time build a task dependency tree between DAOS API operation. This is useful for applications and middleware libraries using DAOS and needing to build a schedule of DAOS operations with dependencies between each other (N-1, 1-N, N-N).

DAOS Task API 提供了一种以非阻塞方式使用 DAOS API 的替代方式，同时在 DAOS API 操作之间构建任务依赖树。 这对于使用 DAOS 的应用程序和中间件库非常有用，并且需要构建彼此之间具有依赖性（N-1、1-N、N-N）的 DAOS 操作计划。

To leverage the task API, the user would need to create a scheduler where DAOS tasks can be created as a part of. The task API is generic enough to allow the user to mix DAOS specific tasks (through the DAOS task API) and other user defined tasks and add dependencies between those.

为了利用任务 API，用户需要创建一个调度程序，在其中可以创建 DAOS 任务作为其中的一部分。 任务 API 足够通用，允许用户混合 DAOS 特定任务（通过 DAOS 任务 API）和其他用户定义的任务，并在它们之间添加依赖项。

For more details on how TSE is used in client library, see [TSE internals documentation](https://github.com/daos-stack/daos/blob/master/src/common/README.md) for more details.

有关如何在客户端库中使用 TSE 的更多详细信息，请参阅 TSE 内部文档以获取更多详细信息。

###### [array interface](https://github.com/daos-stack/daos/blob/master/src/client/array/README.md)/数组接口

DAOS Arrays

A DAOS Array is a special DAOS object to expose a logical 1-dimentional array to the user. The array is created by the user with an immutable record size and chunk size. Additional APIs are provided to access the array (read, write, punch).

DAOS 数组是一种特殊的 DAOS 对象，用于向用户公开逻辑一维数组。 该数组由用户创建，具有不可变的记录大小和块大小。 提供了额外的 API 来访问数组（读、写、打孔）。

Array Representation/数组表示

The Array representation over the DAOS KV API is done with integer typed DKeys, where each DKey holds chunk\_size records. Each DKey has 1 AKey with a NULL value that holds the user array data in an array type extent. The first DKey (which is 0) does not hold any user data, but only the array metadata:

DAOS KV API 上的数组表示是使用整数类型的 DKey 完成的，其中每个 DKey 保存 chunk\_size 记录。 每个 DKey 有 1 个带有 NULL 值的 AKey，用于在数组类型范围内保存用户数组数据。 第一个 DKey（为 0）不保存任何用户数据，而只保存数组元数据：

图片包含 背景图案

描述已自动生成

To illustrate the array mapping, suppose we have a logical array of 10 elements and chunk size being 3. The DAOS KV representation would be:

为了说明数组映射，假设我们有一个包含 10 个元素且块大小为 3 的逻辑数组。 DAOS KV 表示将是：

图片包含 背景图案

描述已自动生成

API and Implementation/ API 和实现

The API (include/daos\_array.h) includes operations to:

* create an array with the required, immutable metadata of the array.
* open an existing array which returns the metadata associated with the array.
* read from an array object.
* write to an array object.
* set size (truncate) of an array. Note this is not equivaluent to preallocation.
* get size of an array.
* punch a range of records from the array.
* destroy/remove an array.

API (include/daos\_array.h) 包括以下操作：

• 使用数组所需的不可变元数据创建一个数组。 • 打开一个现有数组，该数组返回与该数组关联的元数据。 • 从数组对象中读取。 • 写入数组对象。 • 设置数组的大小（截断）。 请注意，这不等同于预分配。 • 获取数组的大小。 • 从数组中打出一系列记录。 • 销毁/删除数组。

The Array API is implemented using the DAOS Task API. For example, the read and write operations create an I/O operation for each DKey and inserts them into the task engine with a parent task that depends on all the child tasks that do the I/O.

Array API 是使用 DAOS Task API 实现的。 例如，读写操作为每个 DKey 创建一个 I/O 操作，并将它们插入到任务引擎中，父任务依赖于执行 I/O 的所有子任务。

The API is currently tested with daos\_test.

###### [KV interface](https://github.com/daos-stack/daos/blob/master/src/client/kv/README.md)

DAOS KV API

The KV API simplifies the DAOS Object API and exposes a simple API to manipulate a Key-Value object with simple put/get/remove/list operations. The API exposes only a single Key (no multi-level keys) and a value associated with that key which is overwritten entirely anytime the key is updated. So internally the mapping of a the HL KV object looks like:

KV API 简化了 DAOS 对象 API，并公开了一个简单的 API，可以通过简单的 put/get/remove/list 操作来操作 Key-Value 对象。 该 API 仅公开一个密钥（没有多级密钥）和与该密钥关联的值，该值在密钥更新时被完全覆盖。 所以内部 HL KV 对象的映射如下所示：

图片包含 背景图案

描述已自动生成

The API is currently tested with daos\_test.

###### [Python API bindings](https://github.com/daos-stack/daos/blob/master/src/src/client/pydaos/raw/README.md)

##### Agent

The [DAOS agent](https://github.com/daos-stack/daos/blob/master/src/control/cmd/daos_agent/README.md) is a daemon residing on the client nodes. It interacts with the DAOS client library through dRPC to authenticate the application process. It is a trusted entity that can sign the DAOS Client credentials using local certificates. The DAOS agent can support different authentication frameworks and uses a Unix Domain Socket to communicate with the client library. The DAOS agent is written in Go and communicates through gRPC with the control plane component of each DAOS server to provide DAOS system membership information to the client library and to support pool listing.

DAOS 代理是驻留在客户端节点上的守护进程。 它通过 dRPC 与 DAOS 客户端库交互以对应用程序进程进行身份验证。 它是一个可信实体，可以使用本地证书签署 DAOS 客户端凭据。 DAOS 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 代理是用 Go 编写的，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 系统成员信息并支持池列表。

##### DAOS Layering and Services/分层和服务

###### Architecture

As shown in the diagram below, the DAOS stack is structured as a collection of storage services over a client/server architecture. Examples of DAOS services are the pool, container, object and rebuild services.

如下图所示，DAOS 堆栈被构造为客户端/服务器架构上的存储服务集合。 DAOS 服务的示例是池、容器、对象和重建服务。

形状

描述已自动生成

A DAOS service can be spread across the control and data planes and communicate internally through dRPC. Most services have client and server components that can synchronize through gRPC or CART. Cross-service communications are always done through direct API calls. Those function calls can be invoked across either the client or server component of the services. While each DAOS service is designed to be fairly autonomous and isolated, some are more tightly coupled than others. That is typically the case of the rebuild service that needs to interact closely with the pool, container and object services to restore data redundancy after a DAOS server failure.

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。 大多数服务都有客户端和服务器组件，可以通过 gRPC 或 CART 进行同步。 跨服务通信始终通过直接 API 调用完成。 可以跨服务的客户端或服务器组件调用这些函数调用。 虽然每个 DAOS 服务都被设计为相当自治和隔离，但有些服务比其他服务更紧密地耦合。 这就是重建服务的典型情况，它需要与池、容器和对象服务密切交互，以在 DAOS 服务器发生故障后恢复数据冗余。

While the service-based architecture offers flexibility and extensibility, it is combined with a set of infrastucture libraries that provide a rich software ecosystem (e.g. communications, persistent storage access, asynchronous task execution with dependency graph, accelerator support, ...) accessible to all the DAOS services.

虽然基于服务的架构提供了灵活性和可扩展性，但它与一组基础设施库相结合，提供了丰富的软件生态系统（例如通信、持久存储访问、具有依赖关系图的异步任务执行、加速器支持等）， 所有 DAOS 服务。

###### Source Code Structure/源代码结构

Each infrastructure library and service is allocated a dedicated directory under src/. The client and server components of a service are stored in separate files. Functions that are part of the client component are prefixed with dc\\_ (stands for DAOS Client) whereas server-side functions use the ds\\_ prefix (stands for DAOS Server). The protocol and RPC format used between the client and server components is usually defined in a header file named rpc.h.

每个基础架构库和服务都在 src/ 下分配了一个专用目录。 服务的客户端和服务器组件存储在单独的文件中。 作为客户端组件一部分的函数以 dc\\_（代表 DAOS 客户端）为前缀，而服务器端函数使用 ds\\_ 前缀（代表 DAOS 服务器）。 客户端和服务器组件之间使用的协议和 RPC 格式通常在名为 rpc.h 的头文件中定义。

All the Go code executed in context of the control plane is located under src/control. Management and security are the services spread across the control (Go language) and data (C language) planes and communicating internally through dRPC.

在控制平面上下文中执行的所有 Go 代码都位于 src/control 下。 管理和安全是跨越控制（Go 语言）和数据（C 语言）平面并通过 dRPC 进行内部通信的服务。

Headers for the official DAOS API exposed to the end user (i.e. I/O middleware or application developers) are under src/include and use the daos\\_ prefix. Each infrastructure library exports an API that is available under src/include/daos and can be used by any services. The client-side API (with dc\\_ prefix) exported by a given service is also stored under src/include/daos whereas the server-side interfaces (with ds\\_ prefix) are under src/include/daos\_srv.

公开给最终用户（即 I/O 中间件或应用程序开发人员）的官方 DAOS API 的标头位于 src/include 下并使用 daos\\_ 前缀。 每个基础架构库都导出一个 API，该 API 在 src/include/daos 下可用，并且可以被任何服务使用。 由给定服务导出的客户端 API（带有 dc\\_ 前缀）也存储在 src/include/daos 下，而服务器端接口（带有 ds\\_ 前缀）在 src/include/daos\_srv 下。

###### Infrastructure Libraries/基础设施库

The GURT and common DAOS (i.e. libdaos\\_common) libraries provide logging, debugging and common data structures (e.g. hash table, btree, ...) to the DAOS services.

GURT 和通用 DAOS（即 libdaos\\_common）库为 DAOS 服务提供日志记录、调试和通用数据结构（例如哈希表、btree 等）。

Local NVM storage is managed by the Versioning Object Store (VOS) and blob I/O (BIO) libraries. VOS implements the persistent index in SCM whereas BIO is responsible for storing application data in either NVMe SSD or SCM depending on the allocation strategy. The VEA layer is integrated into VOS and manages block allocation on NVMe SSDs.

本地 NVM 存储由版本对象存储 (VOS) 和 blob I/O (BIO) 库管理。 VOS 在 SCM 中实现持久索引，而 BIO 负责根据分配策略将应用程序数据存储在 NVMe SSD 或 SCM 中。 VEA 层集成到 VOS 中并管理 NVMe SSD 上的块分配。

DAOS objects are distributed across multiple targets for both performance (i.e. sharding) and resilience (i.e. replication or erasure code). The placement library implements different algorithms (e.g. ring-based placement, jump consistent hash, ...) to generate the layout of an object from the list of targets and the object identifier.

DAOS 对象分布在多个targets上，以实现性能（即分片）和弹性（即复制或EC）。 placement库实现了不同的算法（例如，基于ring-based的放置、跳转一致哈希等）以根据targets列表和对象标识符生成对象的布局。

The replicated service (RSVC) library finally provides some common code to support fault tolerance. This is used by the pool, container & management services in conjunction with the RDB library that implements a replicated key-value store over Raft.

复制服务（RSVC）库最终提供了一些通用代码来支持容错。 它由池、容器和管理服务与 RDB 库结合使用，该库在 Raft 上实现了复制的键值存储。

For further reading on those infrastructure libraries, please see:

有关这些基础设施库的进一步阅读，请参阅

[Common Library](https://github.com/daos-stack/daos/blob/master/src/common/README.md)/公共图书馆

Common functionality and infrastructure shared across all DAOS components are provided in external shared libraries. This includes the following features:

* Hash and checksum routines
* Event and event queue support for non-blocking operations
* Logging and debugging infrastructure
* Locking primitives
* Network transport

在所有 DAOS 组件之间共享的通用功能和基础架构在外部共享库中提供。 这包括以下功能： • 散列和校验和例程

• 事件和事件队列支持非阻塞操作

• 日志和调试基础设施 • 锁定原语 • 网络传输

Task Scheduler Engine (TSE)/ 任务调度引擎

The TSE is a generic library to create generic tasks with function callbacks, optionally add dependencies between those tasks, and schedule them in an engine that is progressed to execute those tasks in an order determined by a dependency graph in which they were inserted in. The task dependency graph is the integral part of the scheduler to allow users to create several tasks and progress them in a non-blocking manner.

TSE 是一个通用库，用于创建具有函数回调的通用任务，可选地在这些任务之间添加依赖关系，并在引擎中安排它们，该引擎会按照插入它们的依赖关系图确定的顺序执行这些任务。 任务依赖图是调度程序的组成部分，允许用户创建多个任务并以非阻塞方式推进它们。

The TSE is not DAOS specific, but used to be part of the DAOS core and was later extracted into the common src as a standalone API. The API is generic and allows creation of tasks in an engine without any DAOS specific functionality. The DAOS library does provide a task API that is built on top of the TSE. For more information on that see [here](https://github.com/daos-stack/daos/blob/master/src/client/api/README.md). Furthermore, DAOS uses the TSE internally to track and progress all API tasks that are associated with the API event and, in some cases, to schedule several inflight "child" tasks corresponding to a single API task and add a dependency on that task to track all those inflight "child" tasks. An example of that would be the Array API in the DAOS library and the object update with multiple replicas.

TSE 不是 DAOS 特定的，但曾经是 DAOS 核心的一部分，后来作为独立 API 提取到公共 src 中。 API 是通用的，允许在没有任何 DAOS 特定功能的引擎中创建任务。 DAOS 库确实提供了一个构建在 TSE 之上的任务 API。 有关更多信息，请参见此处。 此外，DAOS 在内部使用 TSE 来跟踪和处理与 API 事件关联的所有 API 任务，并且在某些情况下，调度与单个 API 任务相对应的多个飞行中的“子”任务，并添加对该任务的依赖以进行跟踪 所有这些飞行中的“子”任务。 一个例子是 DAOS 库中的 Array API 和具有多个副本的对象更新。

Scheduler API

The scheduler API allows a user to create a generic scheduler and add tasks to it. At the time of scheduler creation, the user can register a completion callback to be called when the scheduler is finalized.

调度程序 API 允许用户创建通用调度程序并向其添加任务。 在调度器创建时，用户可以注册一个完成回调以在调度器完成时调用。

The tasks that are added to the scheduler do not progress on their own. There has to be explicit calls to a progress function (daos\_sched\_progress) on the scheduler to make progress on the tasks in the engine. This progress function can be called by the user occasionally in their program, or a single thread can be forked that calls the progress function repeatedly.

添加到调度程序的任务不会自行进行。 必须显式调用调度程序上的进度函数 (daos\_sched\_progress) 才能在引擎中的任务上取得进展。 用户可以在他们的程序中偶尔调用这个进度函数，或者可以分叉出一个重复调用进度函数的线程。

Task API

The task API allows the creation of tasks with generic body functions and adding them to a scheduler. Once a task is created within a scheduler, it will not be actually scheduled to run without an explicit call from the user to the task schedule function, unless it's part of a task dependency graph where in this case the explicit schedule call is required only to the first task in the graph. After creating the task, the user can register any number of dependencies for the task that would be required to complete before the task can be scheduled to run. In addition, the user will be able to register preparation and completion callback on the task:

任务 API 允许创建具有通用主体功能的任务并将它们添加到调度程序中。 一旦在调度程序中创建了一个任务，如果没有用户对任务调度函数的显式调用，它就不会被实际调度运行，除非它是任务依赖图的一部分，在这种情况下，显式调度调用只需要 图中的第一个任务。 创建任务后，用户可以为任务注册任意数量的依赖项，这些依赖项在可以安排任务运行之前需要完成。 此外，用户将能够在任务上注册准备和完成回调：

* Preparation Callbacks are executed when the task is ready to run but has not been executed yet, meaning the dependencies that the task was created with are done and the scheduler is ready to schedule the task. This is useful when the task to be scheduled needs information that is not available at the time of task creation but will be available after the dependencies of the task complete; for example setting some input parameters for the task body function.
* 当任务准备好运行但尚未执行时执行准备回调，这意味着创建任务的依赖关系已完成并且调度程序已准备好调度任务。 当要调度的任务需要在创建任务时不可用但在任务的依赖关系完成后可用的信息时，这很有用； 例如为任务主体功能设置一些输入参数。
* Completion Callbacks are executed when the task is finished executing and the user needs to do more work or handling when that happens. An example where this would be useful is setting the completion of a higher level event or request that is built on top of the TSE, or to track error status of multiple tasks in a dependency list.
* 完成回调在任务完成执行时执行，并且在发生这种情况时用户需要做更多的工作或处理。 这将很有用的一个示例是设置构建在 TSE 之上的更高级别的事件或请求的完成，或者跟踪依赖项列表中多个任务的错误状态。

Several other functionality on the task API exists to support:

* setting some private data on the task itself that can be queried.
* pushing and popping data on/from task stack space without data copy
* generic task lists

存在任务 API 上的其他几个功能以支持：

• 在任务本身上设置一些可以查询的私有数据。

• 在没有数据复制的情况下在任务堆栈空间上推送和弹出数据

• 通用任务列表

More detail about that functionality can be found in the TSE header in the DAOS code [here](https://github.com/daos-stack/daos/blob/master/src/include/daos/tse.h).

可以在此处 DAOS 代码的 TSE 标头中找到有关该功能的更多详细信息。

dRPC C API

For a general overview of dRPC concepts and the corresponding Go API, see [here](https://github.com/daos-stack/daos/blob/master/src/control/drpc/README.md).

有关 dRPC 概念和相应 Go API 的一般概述，请参见此处。

In the C API, an active dRPC connection is represented by a pointer to a context object (struct drpc). The context supplies all the state information required to communicate over the Unix Domain Socket.

在 C API 中，活动的 dRPC 连接由指向上下文对象 (struct drpc) 的指针表示。 上下文提供通过 Unix 域套接字进行通信所需的所有状态信息。

By default a context starts with one reference to it. You can add references to a given context with drpc\_add\_ref(). When finished with a context, the object should be released by using drpc\_close(). When the last reference is released, the object is freed.

默认情况下，上下文以对它的一个引用开始。 您可以使用 drpc\_add\_ref() 添加对给定上下文的引用。 完成上下文后，应使用 drpc\_close() 释放对象。 当最后一个引用被释放时，对象被释放。

dRPC calls and responses are represented by the Protobuf-generated structures Drpc\_\_Call and Drpc\_\_Response.

dRPC 调用和响应由 Protobuf 生成的结构 Drpc\_\_Call 和 Drpc\_\_Response 表示。

C Client

Connecting to a valid Unix Domain Socket returns a dRPC context, which can be used to execute any number of dRPC calls to the server that set up the socket.

连接到一个有效的 Unix 域套接字会返回一个 dRPC 上下文，它可用于对设置套接字的服务器执行任意数量的 dRPC 调用。

**Note:** Currently synchronous calls (using flag R\_SYNC) are the only type supported. Asynchronous calls receive an instantaneous response but are never truly processed.

注意：目前同步调用（使用标志 R\_SYNC）是唯一支持的类型。 异步调用会收到即时响应，但从未真正处理过。

Basic Client Workflow

1. Open a connection to the server's Unix Domain Socket:
2. struct drpc \*ctx;
3. rc = drpc\_connect("/var/run/my\_socket.sock", &ctx);
4. Send a dRPC call:
5. Drpc\_\_Call \*call;
6. /\* Alloc and set up your Drpc\_\_Call \*/
7. Drpc\_\_Response \*resp = NULL; /\* Response will be allocated by drpc\_call \*/
8. int result = drpc\_call(ctx, R\_SYNC, call, &resp);

1. 打开到服务器的 Unix Domain Socket 的连接： 2. struct drpc \*ctx; 3.rc = drpc\_connect("/var/run/my\_socket.sock", &ctx); 4. 发送 dRPC 调用： 5. Drpc\_\_Call \*call; 6. /\* 分配和设置你的 Drpc\_\_Call \*/ 7. Drpc\_\_Response \*resp = NULL; /\* 响应将由 drpc\_call 分配 \*/ 8. int result = drpc\_call(ctx, R\_SYNC, call, &resp);

An error code returned from drpc\_call() indicates that the message could not be sent, or there was no response. If drpc\_call() returned success, the content of the response still needs to be checked for errors returned from the server.

drpc\_call() 返回的错误代码表示无法发送消息，或者没有响应。 如果 drpc\_call() 返回成功，则仍然需要检查响应的内容是否有从服务器返回的错误。

1. Send as many dRPC calls as desired./ 根据需要发送尽可能多的 dRPC 调用。
2. When finished with the connection, close it: drpc\_close(ctx); **Note**: After drpc\_close() is called, the dRPC context pointer has been freed and is no longer valid./ 连接完成后，关闭它：drpc\_close(ctx); 注意：调用drpc\_close()后，dRPC上下文指针已被释放，不再有效。

C Server

The dRPC server sets up a Unix Domain Socket and begins listening on it for client connections. In general, this means creating a listening dRPC context to detect any incoming connections. Then, when a client connects, a new dRPC context is generated for that specific session. The session context is the one that actually sends and receives data. It is possible to have multiple session contexts open at the same time.

dRPC 服务器建立一个 Unix 域套接字并开始侦听客户端连接。 一般来说，这意味着创建一个监听 dRPC 上下文来检测任何传入的连接。 然后，当客户端连接时，会为该特定会话生成一个新的 dRPC 上下文。 会话上下文是实际发送和接收数据的上下文。 可以同时打开多个会话上下文。

The socket is always set up as non-blocking, so it is necessary to poll for activity on the context's file descriptor (ctx->comm->fd) using a system call like poll() or select() on POSIX-compliant systems. This applies not only to the listening dRPC context, but also to any dRPC context generated for a specific client session.

套接字始终设置为非阻塞，因此有必要在符合 POSIX 的系统上使用 poll() 或 select() 等系统调用轮询上下文文件描述符 (ctx->comm->fd) 上的活动 . 这不仅适用于侦听 dRPC 上下文，还适用于为特定客户端会话生成的任何 dRPC 上下文。

The server flow is dependent on a custom handler function, whose job is to dispatch incoming Drpc\_\_Call messages appropriately. The handler function should inspect the module and method IDs, ensure that the desired method is executed, and populate a Drpc\_\_Response based on the results.

服务器流依赖于自定义处理程序函数，其工作是适当地分派传入的 Drpc\_\_Call 消息。 处理程序函数应检查模块和方法 ID，确保执行所需的方法，并根据结果填充 Drpc\_\_Response。

Basic Server Workflow

1. Set up the Unix Domain Socket at a given path and create a listening context using a custom handler function:
2. void my\_handler(Drpc\_\_Call \*call, Drpc\_\_Response \*resp) {
3. /\* Handle the message based on module/method IDs \*/
4. }
5. ...
6. struct drpc \*listener\_ctx = drpc\_listen("/var/run/drpc\_socket.sock",
7. my\_handler);
8. Poll on the listener context's file descriptor (listener\_ctx->comm->fd).
9. On incoming activity, accept the connection:
10. struct drpc \*session\_ctx = drpc\_accept(listener\_ctx);

This creates a session context for the specific client. All of that client's communications will come over the session context.

1. Poll on the session context's file descriptor (session\_ctx->comm->fd) for incoming data.
2. On incoming data, receive the message:
3. Drpc\_\_Call \*incoming;
4. int result = drpc\_recv\_call(session\_ctx, &incoming);
5. if (result != 0) {
6. /\* process errors \*/
7. }

This unmarshals the incoming data into a Drpc\_\_Call. If the data isn't a Drpc\_\_Call, it returns an error.

1. Allocate a Drpc\_\_Response and pass it into the session handler.
2. Drpc\_\_Response \*resp = drpc\_response\_create(call);
3. session\_ctx->handler(call, resp);

Your session handler should fill out the response with any errors or payloads.

1. Send the response to the caller and clean up:
2. int result = drpc\_send\_response(session\_ctx, resp);
3. if (result != 0) {
4. /\* process errors \*/
5. }
6. drpc\_response\_free(resp);
7. drpc\_call\_free(call);
8. If the client has closed the connection, close the session context to free the pointer:
9. drpc\_close(session\_ctx);
10. When it's time to shut down the server, close any open session contexts, as noted above. Then drpc\_close() the listener context.

1. 在给定路径上设置 Unix Domain Socket 并使用自定义处理程序函数创建侦听上下文： 2. void my\_handler(Drpc\_\_Call \*call, Drpc\_\_Response \*resp) { 3. /\* 根据模块/方法 ID 处理消息 \*/ 4. } 5. ... 6. struct drpc \*listener\_ctx = drpc\_listen("/var/run/drpc\_socket.sock", 7. my\_handler); 8. 轮询侦听器上下文的文件描述符 (listener\_ctx->comm->fd)。 9. 在传入活动中，接受连接： 10. struct drpc \*session\_ctx = drpc\_accept(listener\_ctx); 这会为特定客户端创建会话上下文。该客户端的所有通信都将通过会话上下文。 11. 轮询会话上下文的文件描述符 (session\_ctx->comm->fd) 以获取传入数据。 12. 在传入数据上，接收消息： 13. Drpc\_\_Call \*incoming; 14. int 结果 = drpc\_recv\_call(session\_ctx, &incoming); 15.如果（结果！= 0）{ 16. /\* 处理错误 \*/ 17. } 这会将传入数据解组为 Drpc\_\_Call。如果数据不是 Drpc\_\_Call，则返回错误。 18. 分配一个 Drpc\_\_Response 并将其传递给会话处理程序。 19. Drpc\_\_Response \*resp = drpc\_response\_create(call); 20. session\_ctx->handler(call, resp); 您的会话处理程序应填写带有任何错误或有效负载的响应。 21.将响应发送给调用者并清理： 22. int 结果 = drpc\_send\_response(session\_ctx, resp); 23.如果（结果！= 0）{ 24. /\* 处理错误 \*/ 25. } 26. drpc\_response\_free(resp); 27. drpc\_call\_free(call); 28、如果客户端已经关闭了连接，则关闭会话上下文以释放指针： 29. drpc\_close(session\_ctx); 30. 当需要关闭服务器时，关闭所有打开的会话上下文，如上所述。然后 drpc\_close() 监听器上下文。

Checksummer

A "Checksummer" is used to create checksums from a scatter gather list. The checksummer uses a function table (specified when initialized) to adapt common checksum calls to the underlying library that implements the checksum algorithm. Currently the isa-l and isa-l\_crypto libraries are used to support adler32, crc16, crc32, crc64, sha1, sha256, and sha512. All of the function tables to support these algorithms are in [src/common/multihash\_isal.c](https://github.com/daos-stack/daos/blob/master/src/common/multihash_isal.c). These function tables are not made public, but there is a helper function (daos\_mhash\_type2algo) that will return the appropriate function table given a DAOS\_CSUM\_TYPE. There is another helper function (daos\_contprop2csumtype) that will convert a container property value to the appropriate DAOS\_CSUM\_TYPE. The double "lookups" from container property checksum value to function table was done to remove the coupling from the checksummer and container info.

“校验和”用于从分散收集列表创建校验和。 校验和使用一个函数表（在初始化时指定）来适应对实现校验和算法的底层库的常见校验和调用。 目前使用isa-l和isa-l\_crypto库支持adler32、crc16、crc32、crc64、sha1、sha256和sha512。 所有支持这些算法的函数表都在 src/common/multihash\_isal.c 中。 这些函数表没有公开，但有一个辅助函数 (daos\_mhash\_type2algo) 将返回给定 DAOS\_CSUM\_TYPE 的适当函数表。 还有另一个辅助函数 (daos\_contprop2csumtype) 可以将容器属性值转换为适当的 DAOS\_CSUM\_TYPE。 从容器属性校验和值到函数表的双重“查找”是为了消除校验和和容器信息的耦合。

All checksummer functions should start with daos\_csummer\_\* and take a struct daos\_csummer as the first argument. To initialize a new daos\_csummer, daos\_csummer\_init takes the address to a pointer (so memory can be allocated), the address to a function table implementing the desired checksum algorithm, and because this is a DAOS checksummer, the size to be used for "chunks" (See [VOS](https://github.com/daos-stack/daos/blob/master/src/vos/README.md) for details on chunks and chunk size). If it wasn't for the need to break the incoming data into chunks, the checksummer would not need the chunk size. When done with a checksummer, daos\_csummer\_destroy should be called to free allocated resources. Most checksummer functions are simple passthroughs to the function table if implemented. The main exception is daos\_csummer\_calc which, using the other checksummer functions, creates a checksum from the appropriate memory represented by the scatter gather list (d\_sg\_list\_t) and the extents (daos\_recx\_t) of an I/O descriptor (daos\_iod\_t). The checksums are put into a collection of checksum buffers (daos\_csum\_buf\_t), each containing multiple checksums. The memory for the daos\_csum\_buf\_t's and the checksums will be allocated. Therefore, when done with the checksums, daos\_csummer\_destroy\_csum\_buf should be called to free this memory.

所有校验和函数都应以 daos\_csummer\_\* 开头，并采用 struct daos\_csummer 作为第一个参数。为了初始化一个新的 daos\_csummer，daos\_csummer\_init 将地址指向一个指针（因此可以分配内存），实现所需校验和算法的函数表的地址，并且因为这是一个 DAOS 校验和，用于“块”的大小（有关块和块大小的详细信息，请参阅 VOS）。如果不是需要将传入的数据分成块，校验和将不需要块大小。完成校验和后，应调用 daos\_csummer\_destroy 以释放分配的资源。如果实现，大多数校验和函数是对函数表的简单传递。主要的例外是 daos\_csummer\_calc，它使用其他校验和函数，从由分散收集列表 (d\_sg\_list\_t) 和 I/O 描述符 (daos\_iod\_t) 的范围 (daos\_recx\_t) 表示的适当内存中创建校验和。校验和被放入一组校验和缓冲区 (daos\_csum\_buf\_t)，每个缓冲区包含多个校验和。 daos\_csum\_buf\_t 和校验和的内存将被分配。因此，完成校验和后，应调用 daos\_csummer\_destroy\_csum\_buf 以释放此内存。

There are a set of helper functions (prefixed with dcb) to act on a daos\_csum\_buf\_t. These functions should be straight forward. The daos\_csum\_buf\_t contains a pointer to the first byte of the checksums and information about the checksums, including count, size of checksum, etc.

有一组辅助函数（以 dcb 为前缀）作用于 daos\_csum\_buf\_t。 这些功能应该是直截了当的。 daos\_csum\_buf\_t 包含一个指向校验和的第一个字节的指针和有关校验和的信息，包括计数、校验和的大小等。

[Versioning Object Store (VOS)](https://github.com/daos-stack/daos/blob/master/src/vos/README.md)

The Versioning Object Store (VOS) is responsible for providing and maintaining a persistent object store that supports byte-granular access and versioning for a single shard in a [DAOS pool](https://github.com/daos-stack/daos/blob/master/doc/storage_model.md#DAOS_Pool). It maintains its metadata in persistent memory and may store data either in persistent memory or on block storage, depending on available storage and performance requirements. It must provide this functionality with minimum overhead so that performance can approach the theoretical performance of the underlying hardware as closely as possible, both with respect to latency and bandwidth. Its internal data structures, in both persistent and non-persistent memory, must also support the highest levels of concurrency so that throughput scales over the cores of modern processor architectures. Finally, and critically, it must validate the integrity of all persisted object data to eliminate the possibility of silent data corruption, both in normal operation and under all possible recoverable failures.

版本对象存储 (VOS) 负责提供和维护持久对象存储，该对象存储支持 DAOS 池中单个分片的字节粒度访问和版本控制。 它在持久内存中维护其元数据，并且可以将数据存储在持久内存或块存储中，具体取决于可用存储和性能要求。 它必须以最小的开销提供此功能，以便性能可以尽可能接近底层硬件的理论性能，无论是在延迟还是带宽方面。 它的内部数据结构，无论是在持久性还是非持久性内存中，还必须支持最高级别的并发性，以便吞吐量在现代处理器架构的内核上扩展。 最后，至关重要的是，它必须验证所有持久对象数据的完整性，以消除在正常操作和所有可能的可恢复故障下无提示数据损坏的可能性。

This section provides the details for achieving the design goals discussed above in building a versioning object store for DAOS.

本节提供了在为 DAOS 构建版本控制对象存储时实现上述设计目标的详细信息。

This document contains the following sections:

本文档包含以下部分：

* [Persistent Memory based Storage](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#62)
  + [In-Memory Storage](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#63)
  + [Lightweight I/O Stack: PMDK Libraries](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#64)
* [VOS Concepts](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#71)
  + [VOS Indexes](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#711)
  + [Object Listing](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#712)
* [Key Value Stores](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#72)
  + [Operations Supported with Key Value Store](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#721)
  + [Key in VOS KV Stores](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#723)
  + [Internal Data Structures](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#724)
* [Key Array Stores](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#73)
* [Conditional Update and MVCC](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#82)
  + [VOS Timestamp Cache](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#821)
  + [Read Timestamps](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#822)
  + [Write Timestamps](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#823)
  + [MVCC Rules](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#824)
  + [Punch Propagation](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#825)
* [Epoch Based Operations](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#74)
  + [VOS Discard](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#741)
  + [VOS Aggregate](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#742)
* [VOS Checksum Management](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#79)
* [Metadata Overhead](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#80)
* [Replica Consistency](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#81)
  + [DAOS Two-phase Commit](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#811)
  + [DTX Leader Election](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#812)

• 基于持久内存的存储 o 内存存储 o 轻量级 I/O 堆栈：PMDK 库

• VOS 概念 o VOS 索引 o 对象列表

• 键值存储 o 键值存储支持的操作 o KEY IN VOS KV stores o 内部数据结构

• 键数组存储

• 条件更新和 MVCC o VOS 时间戳缓存 o 读取时间戳 o 写时间戳 o MVCC 规则 o 打孔传播 • 基于epoch的操作 o VOS 丢弃 o VOS 聚合

• VOS 校验和管理

• 元数据开销

• 副本一致性 o DAOS 两阶段提交 o DTX 领导人选举

Persistent Memory based Storage/基于持久内存的存储

In-Memory Storage/内存存储

The VOS is designed to use a persistent-memory storage model that takes advantage of byte-granular, sub-microsecond storage access possible with new NVRAM technology. This enables a disruptive change in performance compared to conventional storage systems for application and system metadata and small, fragmented, and misaligned I/O. Direct access to byte-addressable low-latency storage opens up new horizons where metadata can be scanned in less than a second without bothering with seek time and alignment.

VOS 旨在使用持久内存存储模型，该模型利用新 NVRAM 技术可能实现的字节粒度、亚微秒存储访问。 与传统存储系统相比，对于应用程序和系统元数据以及小型、碎片化和未对齐的 I/O，这可以实现性能的破坏性变化。 直接访问字节可寻址的低延迟存储开辟了新的视野，可以在不到一秒的时间内扫描元数据，而无需费心寻找时间和对齐。

The VOS relies on a log-based architecture using persistent memory primarily to maintain internal persistent metadata indexes. The actual data can be stored either in persistent memory directly or in block-based NVMe storage. The DAOS service has two tiers of storage: Storage Class Memory (SCM) for byte-granular application data and metadata, and NVMe for bulk application data. Similar to how PMDK is currently used to facilitate access to SCM, the Storage Performance Development Kit ([SPDK](https://spdk.io/)) is used to provide seamless and efficient access to NVMe SSDs. The current DAOS storage model involves three DAOS server xstreams per core, along with one main DAOS server xstream per core mapped to an NVMe SSD device. DAOS storage allocations can occur on either SCM by using a PMDK pmemobj pool, or on NVMe, using an SPDK blob. All local server metadata will be stored in a per-server pmemobj pool on SCM and will include all current and relevant NVMe devices, pool, and xstream mapping information. Please refer to the [Blob I/O](https://github.com/daos-stack/daos/blob/master/src/bio/README.md) (BIO) module for more information regarding NVMe, SPDK, and per-server metadata. Special care is taken when developing and modifying the VOS layer because any software bug could corrupt data structures in persistent memory. The VOS, therefore, checksums its persistent data structures despite the presence of hardware ECC.

VOS 依赖于使用持久内存的基于日志的架构，主要用于维护内部持久元数据索引。实际数据可以直接存储在持久内存中，也可以存储在基于块的 NVMe 存储中。 DAOS 服务有两层存储：用于字节粒度应用程序数据和元数据的存储类内存 (SCM)，以及用于批量应用程序数据的 NVMe。使用PMDK对 SCM 的访问，存储性能开发套件 (SPDK) 用于提供对 NVMe SSD 的无缝和高效访问。当前的 DAOS 存储模型每个核三个 DAOS server xstream，以及映射到 NVMe SSD 设备的每个核一个主 DAOS server xstream。 DAOS 存储分配可以在 SCM 上使用 PMDK pmemobj 池进行，也可以在 NVMe 上使用 SPDK blob 进行。所有本地服务器元数据都将存储在 SCM 上的每服务器 pmemobj 池中，并将包括所有当前和相关的 NVMe 设备、池和 xstream 映射信息。有关 NVMe、SPDK 和每服务器元数据的更多信息，请参阅 Blob I/O (BIO) 模块。在开发和修改 VOS 层时要特别小心，因为任何软件错误都可能破坏持久内存中的数据结构。因此，尽管存在硬件 ECC，VOS 仍会对其持久数据结构进行校验和。

The VOS provides a lightweight I/O stack fully in user space, leveraging the [PMDK](https://github.com/daos-stack/daos/blob/master/src/vos/pmem.io) open-source libraries developed to support this programming model.

VOS 完全在用户空间中提供轻量级 I/O 堆栈，利用为支持此编程模型而开发的 PMDK 开源库。

Lightweight I/O Stack: PMDK Libraries/轻量级 I/O 堆栈：PMDK 库

Although persistent memory is accessible via direct load/store, updates go through multiple levels of caches, including the processor L1/2/3 caches and the NVRAM controller. Durability is guaranteed only after all those caches have been explicitly flushed. The VOS maintains internal data structures in persistent memory that must retain some level of consistency so that operation may be resumed without loss of durable data after an unexpected crash or power outage. The processing of a request will typically result in several memory allocations and updates that must be applied atomically.

尽管可以通过直接加载/存储访问持久内存，但更新会通过多个级别的缓存，包括处理器 L1/2/3 缓存和 NVRAM 控制器。 只有在所有这些缓存都被明确刷新后才能保证持久性。 VOS 在持久内存中维护内部数据结构，这些结构必须保持一定程度的一致性，以便在意外崩溃或断电后可以恢复操作而不会丢失持久数据。 请求的处理通常会导致必须以原子方式应用的多个内存分配和更新。

Consequently, a transactional interface must be implemented on top of persistent memory to guarantee internal VOS consistency. It is worth noting that such transactions are different from the DAOS transaction mechanism. Persistent memory transactions must guarantee consistency of VOS internal data structures when processing incoming requests, regardless of their epoch number. Transactions over persistent memory can be implemented in many different ways, e.g., undo logs, redo logs, a combination of both, or copy-on-write.

因此，必须在持久内存之上实现事务接口以保证内部 VOS 一致性。 值得注意的是，此类交易不同于DAOS交易机制。 持久内存事务在处理传入请求时必须保证 VOS 内部数据结构的一致性，无论它们的epoch数如何。 持久内存上的事务可以通过许多不同的方式实现，例如撤消日志、重做日志、两者的组合或写时复制。

[PMDK](https://pmem.io) is an open source collection of libraries for using persistent memory, optimized specifically for NVRAM. Among these is the libpmemobj library, which implements relocatable persistent heaps called persistent memory pools. This includes memory allocation, transactions, and general facilities for persistent memory programming. The transactions are local to one thread (not multi-threaded) and rely on undo logs. Correct use of the API ensures that all memory operations are rolled back to the last committed state upon opening a pool after a server failure. VOS utilizes this API to ensure consistency of VOS internal data structures, even in the event of server failures.

PMDK 是用于使用持久内存的开源库集合，专门针对 NVRAM 进行了优化。 其中包括 libpmemobj 库，它实现了称为持久内存池的可重定位持久堆。 这包括内存分配、事务和用于持久内存编程的一般设施。 事务对于一个线程（不是多线程）是本地的，并且依赖于撤消日志。 正确使用 API 可确保在服务器故障后打开池时所有内存操作都回滚到上次提交的状态。 VOS 利用此 API 来确保 VOS 内部数据结构的一致性，即使在服务器出现故障的情况下也是如此。

VOS Concepts/ vos 概念

The versioning object store provides object storage local to a storage target by initializing a VOS pool (vpool) as one shard of a DAOS pool. A vpool can hold objects for multiple object address spaces called containers. Each vpool is given a unique UID on creation, which is different from the UID of the DAOS pool. The VOS also maintains and provides a way to extract statistics like total space, available space, and number of objects present in a vpool.

版本对象存储通过将 VOS 池 (vpool) 初始化为 DAOS 池的一个分片来提供存储target本地的对象存储。 一个 vpool 可以为称为容器的多个对象地址空间保存对象。 每个 vpool 在创建时都会被赋予一个唯一的 UID，它与 DAOS 池的 UID 不同。 VOS 还维护并提供了一种提取统计数据的方法，例如总空间、可用空间和 vpool 中存在的对象数量。

The primary purpose of the VOS is to capture and log object updates in arbitrary time order and integrate these into an ordered epoch history that can be traversed efficiently on demand. This provides a major scalability improvement for parallel I/O by correctly ordering conflicting updates without requiring them to be serialized in time. For example, if two application processes agree on how to resolve a conflict on a given update, they may write their updates independently with the assurance that they will be resolved in the correct order at the VOS.

VOS 的主要目的是以任意时间顺序捕获和记录对象更新，并将它们集成到可以按需高效遍历的有序epoch历史记录中。 这通过正确排序冲突更新而无需及时序列化它们，为并行 I/O 提供了重大的可扩展性改进。 例如，如果两个应用进程就如何解决给定更新的冲突达成一致，则它们可以独立编写它们的更新，并保证它们将在 VOS 以正确的顺序解决

The VOS also allows all object updates associated with a given epoch and process group to be discarded. This functionality ensures that when a DAOS transaction must be aborted, all associated updates are invisible before the epoch is committed for that process group and becomes immutable. This ensures that distributed updates are atomic - i.e. when a commit completes, either all updates have been applied or been discarded.

VOS 还允许丢弃与给定时期和进程组关联的所有对象更新。 此功能可确保当必须中止 DAOS 事务时，所有关联的更新在为该进程组提交 epoch 之前都是不可见的，并且变得不可变。 这确保分布式更新是原子的 - 即当提交完成时，所有更新都已应用或被丢弃。

Finally, the VOS may aggregate the epoch history of objects in order to reclaim space used by inaccessible data and to speed access by simplifying indices. For example, when an array object is "punched" from 0 to infinity in a given epoch, all data updated after the latest snapshot before this epoch becomes inaccessible once the container is closed.

最后，VOS 可以聚合对象的epoch历史，以回收不可访问数据使用的空间并通过简化索引来加快访问速度。 例如，当数组对象在给定的 epoch 中从 0 到无穷大“打孔”时，一旦容器关闭，在此 epoch 之前的最新快照之后更新的所有数据都将无法访问。

Internally, the VOS maintains an index of container UUIDs that references each container stored in a particular pool. The container itself contains three indices. The first is an object index used to map an object ID and epoch to object metadata efficiently when servicing I/O requests. The other two indices are for maintining active and committed [DTX](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#811) records for ensuring efficient updates across multiple replicas.

在内部，VOS 维护一个容器 UUID 索引，该索引引用存储在特定池中的每个容器。 容器本身包含三个索引。 第一个是对象索引，用于在为 I/O 请求提供服务时有效地将对象 ID 和epoch映射到对象元数据。 另外两个索引用于维护活动和提交的 DTX 记录，以确保跨多个副本的有效更新。

DAOS supports two types of values, each associated with a Distribution Key (DKEY) and an Attribute Key (AKEY): Single value and Array value. The DKEY is used for placement, determining which VOS pool is used to store the data. The AKEY identifies the data to be stored. The ability to specify both a DKEY and an AKEY provides applications with the flexibility to either distribute or co-locate different values in DAOS. A single value is an atomic value meaning that writes to an AKEY update the entire value and reads retrieve the latest value in its entirety. An array value is an index of equally sized records. Each update to an array value only affects the specified records and reads read the latest updates to each record index requested. Each VOS pool maintains the VOS provides a per container hierarchy of containers, objects, DKEYs, AKEYs, and values as shown [below](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7a). The DAOS API provides generic Key-Value and Array abstractions built on this underlying interface. The former sets the DKEY to the user specified key and uses a fixed AKEY. The latter uses the upper bits of the array index to create a DKEY and uses a fixed AKEY thus evenly distributing array indices over all VOS pools in the object layout. For the remainder of the VOS description, Key-Value and Key-Array shall be used to describe the VOS layout rather than these simplifying abstractions. In other words, they shall describe the DKEY-AKEY-Value in a single VOS pool.

DAOS 支持两种类型的值，每种类型都与分发密钥 (DKEY) 和属性密钥 (AKEY) 相关联：单个值和数组值。 DKEY 用于放置，确定使用哪个 VOS 池来存储数据。 AKEY 标识要存储的数据。指定 DKEY 和 AKEY 的能力为应用程序提供了在 DAOS 中分发或共同定位不同值的灵活性。单个值是原子值，这意味着写入 AKEY 会更新整个值并读取完整的最新值。数组值是相同大小记录的索引。对数组值的每次更新仅影响指定的记录，并读取请求的每个记录索引的最新更新。每个 VOS 池维护 VOS 提供容器、对象、DKEY、AKEY 和值的每个容器层次结构，如下所示。 DAOS API 提供了基于此底层接口构建的通用键值和数组抽象。前者将 DKEY 设置为用户指定的密钥并使用固定的 AKEY。后者使用数组索引的高位来创建 DKEY 并使用固定的 AKEY 从而在对象布局中的所有 VOS 池上均匀分布数组索引。对于 VOS 描述的其余部分，应使用 Key-Value 和 Key-Array 来描述 VOS 布局而不是这些简化的抽象。换句话说，它们将在单个 VOS 池中描述 DKEY-AKEY-Value

VOS objects are not created explicitly but are created on the first write by creating the object metadata and inserting a reference to it in the owning container's object index. All object updates log the data for each update, which may be an object, DKEY, AKEY, a single value, or array value punch or an update to a single value or array value. Note that "punch" of an extent of an array object is logged as zeroed extents, rather than causing relevant array extents or key values to be discarded. A punch of an object, DKEY, AKEY, or single value is logged, so that reads at a later timestamp see no data. This ensures that the full version history of objects remain accessible. The DAOS api, however, only allows accessing data at snapshots so VOS aggregation can aggressively remove objects, keys, and values that are no longer accessible at a known snapshot.

VOS 对象不是显式创建的，而是通过创建对象元数据并在拥有容器的对象索引中插入对它的引用在第一次写入时创建的。 所有对象更新记录每次更新的数据，可能是对象、DKEY、AKEY、单个值或数组值打孔或对单个值或数组值的更新。 请注意，数组对象范围的“打孔”记录为归零范围，而不是导致相关数组范围或键值被丢弃。 记录对象、DKEY、AKEY 或单个值的冲击，以便在稍后的时间戳读取时看不到数据。 这确保了对象的完整版本历史记录仍然可以访问。 然而，DAOS api 只允许在快照上访问数据，因此 VOS 聚合可以积极地删除在已知快照上不再可访问的对象、键和值。

图示

描述已自动生成

When performing lookup on a single value in an object, the object index is traversed to find the index node with the highest epoch number less than or equal to the requested epoch (near-epoch) that matches the key. If a value or negative entry is found, it is returned. Otherwise, a "miss" is returned, meaning that this key has never been updated in this VOS. This ensures that the most recent value in the epoch history of is returned irrespective of the time-order in which they were integrated and that all updates after the requested epoch are ignored.

当对对象中的单个值执行查找时，遍历对象索引以查找与键匹配的具有小于或等于请求的epoch（近epoch）的最高epoch数的索引节点。 如果找到值或负条目，则将其返回。 否则，返回一个“miss”，这意味着这个key在这个 VOS 中从未被更新过。 这确保了 epoch 历史中的最新值被返回，而不管它们被集成的时间顺序，并且在请求的 epoch 之后的所有更新都被忽略。

Similarly, when reading an array object, its index is traversed to create a gather descriptor that collects all object extent fragments in the requested extent with the highest epoch number less than or equal to the requested epoch. Entries in the gather descriptor either reference an extent containing data, a punched extent that the requestor can interpret as all zeroes, or a "miss", meaning that this VOS has received no updates in this extent. Again, this ensures that the most recent data in the epoch history of the array is returned for all offsets in the requested extent, irrespective of the time-order in which they were written, and that all updates after the requested epoch are ignored.

类似地，在读取数组对象时，遍历其索引以创建收集描述符，该描述符收集请求的范围内具有小于或等于请求的epoch的最高epoch号的所有对象区片段。 收集描述符中的条目要么引用包含数据的范围，请求者可以解释为全零的打孔范围，要么是“未命中”，这意味着此 VOS 在此范围内未收到任何更新。 同样，这可确保为请求范围内的所有偏移量返回数组的历元历史中的最新数据，而不管写入它们的时间顺序，并且忽略请求的历元之后的所有更新。

VOS Indexes

The value of the object index table, indexed by OID, points to a DKEY index. The values in the DKEY index, indexed by DKEY, point to an AKEY index. The values in the AKEY index, indexed by AKEY, point to either a Single Value index or an Array index. A single value index is referenced by epoch and will return the latest value inserted at or prior to the epoch. An array value is indexed by the extent and the epoch and will return portions of extents visible at the epoch.

对象索引表的值，由 OID 索引，指向一个 DKEY 索引。 DKEY 索引中的值由 DKEY 索引，指向 AKEY 索引。 AKEY 索引中的值由 AKEY 索引，指向单值索引或数组索引。 单个值索引由纪元引用，并将返回在纪元或之前插入的最新值。 数组值由范围和纪元索引，并将返回在纪元可见的范围部分。

Hints about the expectations of the object can be encoded in the object ID. For example, an object can be replicated, erasure coded, use checksums, or have integer or lexical DKEYs and/or AKEYs. If integer or lexical keys are used, the object index is ordered by keys, making queries, such as array size, more efficient. Otherwise, keys are ordered by the hashed value in the index. The object ID is 128 bits. The upper 32 bits are used to encodes the object type, and key types while the lower 96 bits are a user defined identifier that must be unique to the container.

可以在对象 ID 中编码有关对象期望的提示。 例如，一个对象可以被复制、纠删码、使用校验和，或者具有整数或keys DKEY 和/或 AKEY。 如果使用整数或词法keys，对象索引按键排序，从而使查询（例如数组大小）更有效。 否则，keys按索引中的散列值排序。 对象 ID 为 128 位。 高 32 位用于对对象类型和key类型进行编码，而低 96 位是用户定义的标识符，必须是容器唯一的标识符。

Each object, dkey, and akey has an associated incarnation log. The incarnation log can be described as an in-order log of creation and punch events for the associated entity. The log is checked for each entity in the path to the value to ensure the entity, and therefore the value, is visible at the requested time.

每个对象、dkey 和 akey 都有一个关联的化身日志。 化身日志可以描述为关联实体的创建和打孔事件的有序日志。 检查值路径中每个实体的日志，以确保该实体以及该值在请求的时间可见。

Object Listing

VOS provides a generic iterator that can be used to iterate through containers, objects, DKEYs, AKEYs, single values, and array extents in a VOS pool. The iteration API is shown in the [figure](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7b) below.

VOS 提供了一个通用迭代器，可用于在 VOS 池中迭代容器、对象、DKEY、AKEY、单个值和数组范围。 迭代API如下图所示。

图形用户界面, 文本

描述已自动生成

The generic VOS iterator API enables both the DAOS enumeration API as well as DAOS internal features supporting rebuild, aggregation, and discard. It is flexible enough to iterate through all keys, single values, and extents for a specified epoch range. Additionally, it supports iteration through visible extents.

通用 VOS 迭代器 API 支持 DAOS 枚举 API 以及支持重建、聚合和丢弃的 DAOS 内部功能。 它足够灵活，可以迭代指定时期范围内的所有键、单个值和范围。 此外，它还支持通过可见范围进行迭代。

Key Value Stores (Single Value)

High-performance simulations generating large quantities of data require indexing and analysis of data, to achieve good insight. Key Value (KV) stores can play a vital role in simplifying the storage of such complex data and allowing efficient processing.

生成大量数据的高性能模拟需要对数据进行索引和分析，以获得良好的洞察力。 键值 (KV) 存储可以在简化此类复杂数据的存储和实现高效处理方面发挥重要作用。

VOS provides a multi-version, concurrent KV store on persistent memory that can grow dynamically and provide quick near-epoch retrieval and enumeration of key values.

VOS 在持久内存上提供多版本、并发 KV 存储，该存储可以动态增长并提供快速的近epoch检索和键值枚举。

Although there is an array of previous work on KV stores, most of them focus on cloud environments and do not provide effective versioning support. Some KV stores provide versioning support but expect monotonically increasing ordering of versions and further, do not have the concept of near-epoch retrieval.

虽然之前有一系列关于 KV 存储的工作，但大多数都专注于云环境，并没有提供有效的版本支持。 一些 KV 存储提供版本控制支持，但期望版本的单调递增排序，并且没有近epoch检索的概念。

VOS must be able to accept insertion of KV pairs at any epoch and must be able to provide good scalability for concurrent updates and lookups on any key-value object. KV objects must also be able to support any type and size of keys and values.

VOS 必须能够在任何时期接受 KV 对的插入，并且必须能够为任何键值对象的并发更新和查找提供良好的可扩展性。 KV 对象还必须能够支持任何类型和大小的键和值。

Operations Supported with Key Value Store/键值存储支持的操作

VOS supports large keys and values with four types of operations; update, lookup, punch, and key enumeration.

VOS 支持大键值和四种操作类型； 更新、查找、打孔和键枚举。

The update and punch operations add a new key to a KV store or log a new value of an existing key. Punch logs the special value "punched", effectively a negative entry, to record the epoch when the key was deleted. Sharing the same epoch for both an update and a punch of the same object, key, value, or extent is disallowed, and VOS will return an error when such is attempted.

更新和打孔操作将新key添加到 KV 存储或记录现有key的新值。 Punch 记录特殊值“punched”，实际上是一个否定条目，以记录删除key时的epoch。 不允许为同一对象、键、值或范围的更新和打孔共享同一epoch，并且当尝试这样做时，VOS 将返回错误。

Lookup traverses the KV metadata to determine the state of the given key at the given epoch. If the key is not found at all, a "miss" is returned to indicate that the key is absent from this VOS. Otherwise, the value at the near-epoch or greatest epoch less than or equal to the requested epoch is returned. If this is the special "punched" value, it means the key was deleted in the requested epoch. The value here refers to the value in the internal tree-data structure. The key-value record of the KV-object is stored in the tree as the value of its node. So in case of punch this value contains a "special" return code/flag to identify the punch operation.

查找遍历 KV 元数据以确定给定key在给定时期的状态。 如果根本没有找到该key，则返回“未命中”以指示该 VOS 中不存在该key。 否则，返回小于或等于请求的epoch的近epoch或最大epoch的值。 如果这是特殊的“打孔”值，则表示该key已在请求的epoch中被删除。 这里的值指的是内部树数据结构中的值。 KV 对象的键值记录作为其节点的值存储在树中。 因此，在打孔的情况下，此值包含一个“特殊”返回代码/标志来标识打孔操作。

VOS also supports the enumeration of keys belonging to a particular epoch.

VOS 还支持枚举属于特定时期的key。

Key in VOS KV Stores/输入 VOS KV 存储

VOS KV supports key sizes from small keys to extremely large keys. For AKEYs and DKEYs, VOS supports either hashed keys or one of two types of "direct" keys: lexical or integer.

VOS KV 支持从小key到极大key的key大小。 对于 AKEY 和 DKEY，VOS 支持散列键或两种“直接”键之一：词法或整数。

Hashed Keys

The most flexible key type is the hashed key. VOS runs two fast hash algorithms on the user supplied key and uses the combined hashed key values for the index. The intention of the combined hash is to avoid collisions between keys. The actual key still must be compared for correctness.

最灵活的key类型是散列key。 VOS 在用户提供的key上运行两种快速散列算法，并使用组合的散列key值作为索引。 组合散列的目的是避免key之间的冲突。 仍然必须比较实际key的正确性。

Direct Keys/直接键

The use of hashed keys results in unordered keys. This is problematic in cases where the user's algorithms may benefit from ordering. Therefore, VOS supports two types of keys that are not hashed but rather interpreted directly.

使用散列key会导致key无序。 在用户的算法可能从排序中受益的情况下，这是有问题的。 因此，VOS 支持两种类型的key，它们不是散列而是直接解释的。

Lexical Keys/ 词法键

Lexical keys are compared using a lexical ordering. This enables usage such as sorted strings. Presently, lexical keys are limited in length, however to 80 characters.

词法键使用词法排序进行比较。 这允许使用例如排序的字符串。 目前，词法键的长度受到限制，但最多为 80 个字符。

Integer Keys/整数键

Integer keys are unsigned 64-bit integers and are compared as such. This enables use cases such as DAOS array API using the upper bits of the index as a dkey and the lower bits as an offset. This enables such objects to use the the DAOS key query API to calculate the size more efficiently.

整数键是无符号的 64 位整数，并按原样进行比较。 这启用了诸如使用索引的高位作为 dkey 和低位作为偏移量的 DAOS 数组 API 之类的用例。 这使此类对象能够使用 DAOS key查询 API 更有效地计算大小。

KV stores in VOS allow the user to maintain versions of the different KV pairs in random order. For example, an update can happen in epoch 10, and followed by another update in epoch 5, where HCE is less than 5. To provide this level of flexibility, each key in the KV store must maintain the epoch of update/punch along with the key. The ordering of entries in index trees first happens based on the key, and then based on the epochs. This kind of ordering allows epochs of the same key to land in the same subtree, thereby minimizing search costs. Conflict resolution and tracking is performed using [DTX](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#81) described later. DTX ensures that replicas are consistent, and failed or uncommitted updates are not visible externally.

VOS 中的 KV 存储允许用户以随机顺序维护不同 KV 对的版本。 例如，更新可能发生在第 10 轮，然后是在第 5 轮的另一个更新，其中 HCE 小于 5。为了提供这种级别的灵活性，KV 存储中的每个键必须保持更新/打孔的 epoch 关键。 索引树中条目的排序首先基于key，然后基于epoch。 这种排序允许相同键的 epoch 出现在相同的子树中，从而最小化搜索成本。 使用稍后描述的 DTX 执行冲突解决和跟踪。 DTX 确保副本一致，并且失败或未提交的更新在外部不可见。

Internal Data Structures/内部数据结构

Designing a VOS KV store requires a tree data structure that can grow dynamically and remain self-balanced. The tree needs to be balanced to ensure that time complexity does not increase with an increase in tree size. Tree data structures considered are red-black trees and B+ Trees, the former is a binary search tree, and the latter an n-ary search tree.

设计一个 VOS KV 存储需要一个可以动态增长并保持自我平衡的树数据结构。 需要平衡树以确保时间复杂度不会随着树大小的增加而增加。 考虑的树数据结构有红黑树和B+树，前者是二叉搜索树，后者是n元搜索树。

Although red-black trees provide less rigid balancing compared to AVL trees, they compensate by having cheaper rebalancing cost. Red-black trees are more widely used in examples such as the Linux kernel, the java-util library, and the C++ standard template library. B+ trees differ from B trees in the fact they do not have data associated with their internal nodes. This can facilitate fitting more keys on a page of memory. In addition, leaf-nodes of B+ trees are linked; this means doing a full scan would require just one linear pass through all the leaf nodes, which can potentially minimize cache misses to access data in comparison to a B Tree.

尽管与 AVL 树相比，红黑树提供的刚性平衡较少，但它们通过更便宜的重新平衡成本进行补偿。 红黑树在Linux内核、java-util库、C++标准模板库等示例中应用更为广泛。 B+ 树与 B 树的不同之处在于它们没有与其内部节点相关联的数据。 这可以有助于在内存页面上安装更多键。 另外，B+树的叶子节点是相互连接的； 这意味着进行一次完整扫描只需要一次线性遍历所有叶节点，与 B 树相比，这可以最大限度地减少缓存未命中以访问数据。

To support update and punch as mentioned in the previous section ([Operations Supported with Key Value Stores](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#721)), an epoch-validity range is set along with the associated key for every update or punch request, which marks the key to be valid from the current epoch until the highest possible epoch. Updates to the same key on a future epoch or past epoch modify the end epoch validity of the previous update or punch accordingly. This way only one key has a validity range for any given key-epoch pair lookup while the entire history of updates to the key is recorded. This facilitates nearest-epoch search. Both punch and update have similar keys, except for a simple flag identifying the operation on the queried epoch. Lookups must be able to search a given key in a given epoch and return the associated value. In addition to the epoch-validity range, the container handle cookie generated by DAOS is also stored along with the key of the tree. This cookie is required to identify behavior in case of overwrites on the same epoch.

为了支持上一节（键值存储支持的操作）中提到的更新和打孔，为每个更新或打孔请求设置了一个epoch有效性范围以及关联的键，这将key标记为从当前epoch开始有效直到可能的最高epoch。在未来的epoch或过去的epoch上更新相同的key会相应地修改先前更新或打孔的结束epoch有效性。这样，对于任何给定的key-epoch对查找，只有一个key具有有效范围，同时记录了key更新的整个历史记录。这有利于最近时期搜索。打孔和更新都有相似的键，除了一个简单的标志来标识查询时代的操作。查找必须能够在给定的 epoch 中搜索给定的键并返回关联的值。除了 epoch-validity 范围外，DAOS 生成的容器句柄 cookie 也与树的 key 一起存储。如果在同一时期发生覆盖，则需要此 cookie 来识别行为。

A simple example input for crearting a KV store is listed in the [Table](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7c) below. Both a B+ Tree based index and a red-black tree based index are shown in the [Table](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7c) and [figure](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7d) below, respectively. For explanation purposes, representative keys and values are used in the example.

下表列出了用于创建 KV 存储的简单示例输入。 基于B+树的索引和基于红黑树的索引分别如下表和图所示。 出于说明目的，示例中使用了代表性的键和值。

**Example VOS KV Store input for Update/Punch**

更新/打孔的 VOS KV 存储输入示例

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **Value** | **Epoch** | **Update (U/P)** |
| Key 1 | Value 1 | 1 | U |
| Key 2 | Value 2 | 2 | U |
| Key 3 | Value 3 | 4 | U |
| Key 4 | Value 4 | 1 | U |
| Key 1 | NIL | 2 | P |
| Key 2 | Value 5 | 4 | U |
| Key 3 | Value 6 | 1 | U |

手机屏幕截图

低可信度描述已自动生成

The red-black tree, like any traditional binary tree, organizes the keys lesser than the root to the left subtree and keys greater than the root to the right subtree. Value pointers are stored along with the keys in each node. On the other hand, a B+ Tree-based index stores keys in ascending order at the leaves, which is where the value is stored. The root nodes and internal nodes (color-coded in blue and maroon accordingly) facilitate locating the appropriate leaf node. Each B+ Tree node has multiple slots, where the number of slots is determined from the order. The nodes can have a maximum of order-1 slots. The container handle cookie must be stored with every key in case of red-black trees, but in case of B+ Trees having cookies only in leaf nodes would suffice, since cookies are not used in traversing.

红黑树与任何传统的二叉树一样，将小于根的键组织到左子树，将大于根的键组织到右子树。 值指针与每个节点中的键一起存储。 另一方面，基于 B+ 树的索引将键以升序存储在叶子上，这是存储值的地方。 根节点和内部节点（相应地用蓝色和栗色进行颜色编码）有助于定位适当的叶节点。 每个 B+ Tree 节点有多个槽，槽的数量由顺序决定。 节点最多可以有 1 阶时隙。 在红黑树的情况下，容器句柄 cookie 必须与每个键一起存储，但在 B+ 树的情况下，仅在叶节点中具有 cookie 就足够了，因为在遍历中不使用 cookie。

In the [table](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7e) below, n is the number of entries in the tree, m is the number of keys, k is the number of the key, epoch entries between two unique keys.

在下表中，n 是树中的条目数，m 是键的数量，k 是键的数量，两个唯一键之间的纪元条目。

**Comparison of average case computational complexity for index**

索引的平均情况计算复杂度比较

|  |  |  |
| --- | --- | --- |
| **Operation** | **Red-black tree** | **B+Tree** |
| Update | O(log2n) | O(logbn) |
| Lookup | O(log2n) | O(logbn) |
| Delete | O(log2n) | O(logbn) |
| Enumeration | O(m\* log2(n) + log2(n)) | O(m \* k + logb (n)) |

Although both these solutions are viable implementations, determining the ideal data structure would depend on the performance of these data structures on persistent memory hardware.

尽管这两种解决方案都是可行的实现方式，但确定理想的数据结构将取决于这些数据结构在持久内存硬件上的性能

VOS also supports concurrent access to these structures, which mandates that the data structure of choice provides good scalability while there are concurrent updates. Compared to B+ Tree, rebalancing in red-black trees causes more intrusive tree structure change; accordingly, B+ Trees may provide better performance with concurrent accesses. Furthermore, because B+ Tree nodes contain many slots depending on the size of each node, prefetching in cache can potentially be easier. In addition, the sequential computational complexities in the [Table](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7e) above show that a B+ Tree-based KV store with a reasonable order, can perform better in comparison to a Red-black tree.

VOS 还支持对这些结构的并发访问，这要求选择的数据结构在并发更新时提供良好的可扩展性。 与 B+ 树相比，红黑树中的重新平衡会导致更多侵入性的树结构变化； 因此，B+ 树可以提供更好的并发访问性能。 此外，由于 B+ 树节点根据每个节点的大小包含许多槽，因此在缓存中预取可能更容易。 此外，上表中的顺序计算复杂度表明，与红黑树相比，具有合理顺序的基于 B+ 树的 KV 存储可以表现得更好。

VOS supports enumerating keys valid in a given epoch. VOS provides an iterator-based approach to extract all the keys and values from a KV object. Primarily, KV indexes are ordered by keys and then by epochs. With each key holding a long history of updates, the size of a tree can be huge. Enumeration with a tree-successors approach can result in an asymptotic complexity of O(m\* log (n) + log (n)) with red-black trees, where m is the number of keys valid in the requested epoch. It takes O(log2 (n)) to locate the first element in the tree and O(log2 (n)) to locate a successor. Because "m" keys need to be retrieved, O( m \* log2 (n)) would be the complexity of this enumeration.

VOS 支持枚举在给定epoch中有效的key。 VOS 提供了一种基于迭代器的方法来从 KV 对象中提取所有键和值。 首先，KV 索引按键排序，然后按纪元排序。 由于每个密钥都有很长的更新历史，因此树的大小可能很大。 使用树后继方法进行枚举可以导致 O(m\* log (n) + log (n)) 的渐近复杂度，其中 m 是在请求的 epoch 中有效的键数。 定位树中的第一个元素需要 O(log2 (n)) 和 O(log2 (n)) 来定位后继元素。 由于需要检索“m”个键，因此该枚举的复杂度为 O( m \* log2 (n)) 。

In the case of B+-trees, leaf nodes are in ascending order, and enumeration would be to parse the leaf nodes directly. The complexity would be O (m \* k + logbn), where m is the number of keys valid in an epoch, k is the number of entries between two different keys in B+ tree leaf nodes, and b is the order for the B+tree. Having "k" epoch entries between two distinct keys incurs in a complexity of O(m \* k). The additional O(logbn) is required to locate the first leftmost key in the tree. The generic iterator interfaces as shown in [Figure](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7d) above would be used for KV enumeration also.

B+树的情况下，叶子节点是升序排列的，枚举就是直接解析叶子节点。 复杂度为 O (m \* k + logbn)，其中 m 是一个时期内有效的键数，k 是 B+ 树叶节点中两个不同键之间的条目数，b 是 B+ 的顺序 树。 在两个不同的键之间具有“k”个纪元条目会导致 O(m \* k) 的复杂性。 需要额外的 O(logbn) 来定位树中第一个最左边的键。 如上图所示的通用迭代器接口也将用于 KV 枚举。

In addition to the enumeration of keys for an object valid in an epoch, VOS also supports enumerating keys of an object modified between two epochs. The epoch index table provides keys updated in each epoch. On aggregating the list of keys associated with each epoch, (by keeping the latest update of the key and discarding the older versions) VOS can generate a list of keys with their latest epoch. By looking up each key from the list in its associated index data structure, VOS can extract values with an iterator-based approach.

除了枚举一个 epoch 中有效的对象的键，VOS 还支持枚举在两个 epoch 之间修改的对象的键。 纪元索引表提供在每个纪元更新的键。 在聚合与每个时期关联的密钥列表时，（通过保持密钥的最新更新并丢弃旧版本）VOS 可以生成具有最新时期的密钥列表。 通过在其关联的索引数据结构中查找列表中的每个键，VOS 可以使用基于迭代器的方法提取值。

Key Array Stores/键数组存储

The second type of object supported by VOS is a Key-Array object. Array objects, similar to KV stores, allow multiple versions and must be able to write, read, and punch any part of the byte extent range concurrently. The [figure](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7f) below shows a simple example of the extents and epoch arrangement within a Key-Array object. In this example, the different lines represent the actual data stored in the respective extents and the color-coding points to different threads writing that extent range.

VOS 支持的第二种对象是 Key-Array 对象。 数组对象，类似于 KV 存储，允许多个版本，并且必须能够同时写入、读取和打孔字节范围范围的任何部分。 下图显示了 Key-Array 对象内的范围和epoch排列的简单示例。 在此示例中，不同的行表示存储在相应范围中的实际数据，以及指向写入该范围范围的不同线程的颜色编码点。

**Example of extents and epochs in a Key Array object**

Key Array 对象中的范围和epoch示例

图形用户界面

中度可信度描述已自动生成

In the [above](https://github.com/daos-stack/daos/blob/master/src/vos/7f) example, there is significant overlap between different extent ranges. VOS supports nearest-epoch access, which necessitates reading the latest value for any given extent range. For example, in the [figure](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7f) above, if there is a read request for extent range 4 - 10 at epoch 10, the resulting read buffer should contain extent 7-10 from epoch 9, extent 5-7 from epoch 8, and extent 4-5 from epoch 1. VOS array objects also support punch over both partial and complete extent ranges.

在上面的示例中，不同范围范围之间存在显着重叠。 VOS 支持最近时期访问，这需要读取任何给定范围范围的最新值。 例如，在上图中，如果在 epoch 10 有对范围范围 4 - 10 的读取请求，则生成的读取缓冲区应包含来自 epoch 9 的范围 7-10、来自 epoch 8 的范围 5-7 和范围 4- 5 从 epoch 1. VOS 数组对象还支持对部分和完整范围范围的打孔。

**Example Input for Extent Epoch Table**

范围epoch表的示例输入

|  |  |  |
| --- | --- | --- |
| **Extent Range** | **Epoch** | **Write (or) Punch** |
| 0 - 100 | 1 | Write |
| 300 - 400 | 2 | Write |
| 400 - 500 | 3 | Write |
| 30 - 60 | 10 | Punch |
| 500 - 600 | 8 | Write |
| 600 - 700 | 9 | Write |

R-Trees provide a reasonable way to represent both extent and epoch validity ranges in such a way as to limit the search space required to handle a read request. VOS provides a specialized R-Tree, called an Extent-Validity tree (EV-Tree) to store and query versioned array indices. In a traditional R-Tree implementation, rectangles are bounded and immutable. In VOS, the "rectangle" consists of the extent range on one axis and the epoch validity range on the other. However, the epoch validity range is unknown at the time of insert so all rectangles are inserted assuming an upper bound of infinity. Originally, the DAOS design called for splitting such in-tree rectangles on insert to bound the validity range but a few factors resulted in the decision to keep the original validity range. First, updates to persistent memory are an order of magnitude more expensive than lookups. Second, overwrites between snapshots can be deleted by aggregation, thus maintaining a reasonably small history of overlapping writes. As such, the EV-Tree implements a two part algorithm on fetch.

R-Trees 提供了一种合理的方式来表示范围和epoch有效性范围，以限制处理读取请求所需的搜索空间。 VOS 提供了一个专门的 R-Tree，称为 Extent-Validity 树 (EV-Tree) 来存储和查询版本化数组索引。在传统的 R-Tree 实现中，矩形是有界且不可变的。在 VOS 中，“矩形”由一个轴上的范围范围和另一个轴上的epoch有效性范围组成。然而，在插入时，epoch有效性范围是未知的，因此所有矩形都假定为无穷大的上限而被插入。最初，DAOS 设计要求在插入时拆分此类树内矩形以限制有效范围，但一些因素导致决定保留原始有效范围。首先，对持久内存的更新比查找要昂贵一个数量级。其次，快照之间的覆盖可以通过聚合删除，从而保持相当小的重叠写入历史。因此，EV-Tree 在获取时实现了两部分算法。

1. Find all overlapping extents. This will include all writes that happened before the requested epoch, even if they are covered by a subsequent write.
2. Sort this by extent start and then by epoch
3. Walk through the sorted array, splitting extents if necessary and marking them as visible as applicable
4. Re-sort the array. This final sort can optionally keep or discard holes and covered extents, depending on the use case.

1. 找到所有重叠的范围。 这将包括在请求的纪元之前发生的所有写入，即使它们被后续写入覆盖。 2.按范围开始排序，然后按纪元排序 3.遍历已排序的数组，必要时拆分范围并将它们标记为适用 4. 重新排序数组。 根据用例，最后的排序可以选择保留或丢弃漏洞和覆盖范围。

TODO: Create a new figure **Rectangles representing extent\_range.epoch\_validity arranged in 2-D space for an order-4 EV-Tree using input in the table** [**above**](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7g)

todo：使用上表中的输入，为 4 阶 EV 树创建一个新图形矩形，表示在二维空间中排列的 extent\_range.epoch\_validity

图表, 箱线图

描述已自动生成

The figure [below](https://github.com/daos-stack/daos/blob/master/src/vos/7l) shows the rectangles constructed with splitting and trimming operations of EV-Tree for the example in the previous [table](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7g) with an additional write at offset {0 - 100} introduced to consider the case for extensive splitting. The figure [above](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7k) shows the EV-Tree construction for the same example.

下图显示了使用 EV-Tree 的拆分和修剪操作构建的矩形，用于上表中的示例，并在偏移量 {0 - 100} 处引入了额外的写入，以考虑广泛拆分的情况。 上图显示了同一示例的 EV-Tree 构造。

**Tree (order - 4) for the example in Table 6 3 (pictorial representation shown in the figure** [**above**](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#7g)

表 6 中示例的树（顺序 - 4） 3（上图所示的图形表示）

图示

描述已自动生成

Inserts in an EV-Tree locate the appropriate leaf-node to insert, by checking for overlap. If multiple bounding boxes overlap, the bounding box with the least enlargement is chosen. Further ties are resolved by choosing the bounding box with the least area. The maximum cost of each insert can be O (logbn).

EV 树中的插入通过检查重叠来定位要插入的适当叶节点。 如果多个边界框重叠，则选择放大最少的边界框。 通过选择面积最小的边界框来解决进一步的关系。 每个插入的最大成本可以是 O(logbn)。

Searching an EV-Tree would work similar to R-Tree, aside from the false overlap issue described above. All overlapping internal nodes must be pursued, till there are matching internal nodes and leaves. Since extent ranges can span across multiple rectangles, a single search can hit multiple rectangles. In an ideal case (where the entire extent range falls on one rectangle), the read cost is O(logbn) where b is the order of the tree. The sorting and splitting phase adds the additional overhead of O(n log n) where n is the number of matching extents. In the worst case, this is equivalent to all extents in the tree, but this is mitigated by aggregation and the expectation that the tree associated with a single shard of a single key will be relatively small.

除了上面描述的错误重叠问题之外，搜索 EV-Tree 的工作方式类似于 R-Tree。 必须追求所有重叠的内部节点，直到有匹配的内部节点和叶子。 由于范围范围可以跨越多个矩形，因此一次搜索可以命中多个矩形。 在理想情况下（整个范围范围落在一个矩形上），读取成本为 O(logbn)，其中 b 是树的顺序。 排序和拆分阶段增加了 O(n log n) 的额外开销，其中 n 是匹配范围的数量。 在最坏的情况下，这相当于树中的所有范围，但是这可以通过聚合和与单个键的单个分片相关联的树相对较小的预期来缓解。

For deleting nodes from an EV-Tree, the same approach as search can be used to locate nodes, and nodes/slots can be deleted. Once deleted, to coalesce multiple leaf-nodes that have less than order/2 entries, reinsertion is done. EV-tree reinserts are done (instead of merging leaf-nodes as in B+ trees) because on deletion of leaf node/slots, the size of bounding boxes changes, and it is important to make sure the rectangles are organized into minimum bounding boxes without unnecessary overlaps. In VOS, delete is required only during aggregation and discard operations. These operations are discussed in a following section ([Epoch Based Operations](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#74)).

对于从 EV-Tree 中删除节点，可以使用与搜索相同的方法来定位节点，并且可以删除节点/槽。 一旦删除，要合并具有少于 order/2 个条目的多个叶节点，重新插入就完成了。 重新插入 EV 树（而不是像 B+ 树中那样合并叶节点），因为在删除叶节点/槽时，边界框的大小会发生变化，并且确保将矩形组织成最小边界框而没有 不必要的重叠。 在 VOS 中，只有在聚合和丢弃操作期间才需要删除。 这些操作将在下一节（基于纪元的操作）中讨论。

Conditional Update and MVCC/条件更新和 MVCC

VOS supports conditional operations on individual dkeys and akeys. The following operations are supported:

* Conditional fetch: Fetch if the key exists, fail with -DER\_NONEXIST otherwise
* Conditional update: Update if the key exists, fail with -DER\_NONEXIST otherwise
* Conditional insert: Update if the key doesn't exist, fail with -DER\_EXIST otherwise
* Conditional punch: Punch if the key exists, fail with -DER\_NONEXIST otherwise

VOS 支持对单个 dkey 和 akey 进行条件操作。 支持以下操作：

• 条件获取：如果key存在则获取，否则失败并返回 -DER\_NONEXIST

• 条件更新：如果key存在则更新，否则失败并返回 -DER\_NOEXIST

• 条件插入：如果key不存在则更新，否则以 -DER\_EXIST 失败

• 条件打孔：如果key存在则打孔，否则以 -DER\_NONEXIST 失败

These operations provide atomic operations enabling certain use cases that require such. Conditional operations are implemented using a combination of existence checks and read timestamps. The read timestamps enable limited MVCC to prevent read/write races and provide serializability guarantees.

这些操作提供原子操作，支持某些需要这样的用例。 使用存在检查和读取时间戳的组合来实现条件操作。 读取时间戳使有限的 MVCC 能够防止读/写竞争并提供可序列化保证。

VOS Timestamp Cache/ VOS 时间戳缓存

VOS maintains an in-memory cache of read and write timestamps in order to enforce MVCC semantics. The timestamp cache itself consists of two parts:

VOS 维护读取和写入时间戳的内存缓存，以强制执行 MVCC 语义。 时间戳缓存本身由两部分组成：

1. Negative entry cache. A global array per target for each type of entity including objects, dkeys, and akeys. The index at each level is determined by the combination of the index of the parent entity, or 0 in the case of containers, and the hash of the entity in question. If two different keys map to the same index, they share timestamp entries. This will result in some false conflicts but does not affect correctness so long as progress can be made. The purpose of this array is to store timestamps for entries that do not exist in the VOS tree. Once an entry is created, it will use the mechanism described in #2 below. Note that multiple pools in the same target use this shared cache so it is also possible for false conflicts across pools before an entity exists. These entries are initialized at startup using the global time of the starting server. This ensures that any updates at an earlier time are forced to restart to ensure we maintain automicity since timestamp data is lost when a server goes down. 否定条目缓存。每种类型实体的每个target的全局数组，包括对象、dkeys 和 akeys。每个级别的索引由父实体的索引（或在容器的情况下为 0）和相关实体的哈希值的组合确定。如果两个不同的键映射到同一个索引，则它们共享时间戳条目。这会导致一些错误的冲突，但只要能取得进展，不影响正确性。此数组的目的是存储 VOS 树中不存在的条目的时间戳。创建条目后，它将使用下面#2 中描述的机制。请注意，同一target中的多个池使用此共享缓存，因此也可能在实体存在之前跨池发生虚假冲突。这些条目在启动时使用启动服务器的全局时间进行初始化。这确保了较早时间的任何更新都被强制重新启动，以确保我们保持自动化，因为当服务器出现故障时时间戳数据会丢失。
2. Positive entry cache. An LRU cache per target for existing containers, objects, dkeys, and akeys. One LRU array is used for each level such that containers, objects, dkeys, and akeys only conflict with cache entries of the same type. Some accuracy is lost when existing items are evicted from the cache as the values will be merged with the corresponding negative entry described in #1 above until such time as the entry is brought back into cache. The index of the cached entry is stored in the VOS tree though it is only valid at runtime. On server restarts, the LRU cache is initialized from the global time when the restart occurs and all entries are automatically invalidated. When a new entry is brought into the LRU, it is initialized using the corresponding negative entry. The index of the LRU entry is stored in the VOS tree providing O(1) lookup on subsequent accesses. 正项缓存。 现有容器、对象、dkey 和 akey 的每个target的 LRU 缓存。 每个级别使用一个 LRU 数组，以便容器、对象、dkey 和 akey 仅与相同类型的缓存条目冲突。 当现有项目从缓存中被逐出时，一些准确性会丢失，因为这些值将与上面 #1 中描述的相应否定条目合并，直到条目被带回缓存为止。 缓存条目的索引存储在 VOS 树中，尽管它仅在运行时有效。 在服务器重新启动时，LRU 缓存从重新启动时的全局时间开始初始化，并且所有条目都会自动失效。 当一个新条目被带入 LRU 时，它使用相应的否定条目进行初始化。 LRU 条目的索引存储在 VOS 树中，为后续访问提供 O(1) 查找。

Read Timestamps

Each entry in the timestamp cache contains two read timestamps in order to provide serializability guarantees for DAOS operations. These timestamps are

时间戳缓存中的每个条目都包含两个读取时间戳，以便为 DAOS 操作提供可序列化保证。 这些时间戳是

1. A low timestamp (entity.low) indicating that all nodes in the subtree rooted at the entity have been read at entity.low
2. A high timestamp (entity.high) indicating that at least one node in the subtree rooted at the entity has been read at entity.high.

1. 一个低时间戳（entity.low），表明以实体为根的子树中的所有节点都在 entity.low 处被读取

2. 一个高时间戳（entity.high），表明以实体为根的子树中的至少一个节点已在 entity.high 处被读取。

For any leaf node (i.e., akey), low == high; for any non-leaf node, low <= high.

The usage of these timestamps is described [below](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#824)

下面描述了这些时间戳的用法

Write Timestamps

In order to detect epoch uncertainty violations, VOS also maintains a pair of write timestamps for each container, object, dkey, and akey. Logically, the timestamps represent the latest two updates to either the entity itself or to an entity in a subtree. At least two timestamps are required to avoid assuming uncertainty if there are any later updates. The figure [below](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#8a) shows the need for at least two timestamps. With a single timestamp only, the first, second, and third cases would be indistinguishable and would be rejected as uncertain. The most accurate write timestamp is used in all cases. For instance, if the access is an array fetch, we will check for conflicting extents in the absence of an uncertain punch of the corresponding key or object.

为了检测epoch不确定性违规，VOS 还为每个容器、对象、dkey 和 akey 维护了一对写入时间戳。 从逻辑上讲，时间戳表示实体本身或子树中实体的最新两次更新。 如果有任何后续更新，至少需要两个时间戳以避免假设不确定性。 下图显示了至少需要两个时间戳。 仅使用单个时间戳，第一种、第二种和第三种情况将无法区分，并且会因为不确定而被拒绝。 在所有情况下都使用最准确的写入时间戳。 例如，如果访问是数组获取，我们将在没有相应键或对象的不确定冲击的情况下检查是否存在冲突的范围。

**Scenarios illustrating utility of write timestamp cache**

说明写入时间戳缓存的效用的场景

日程表

描述已自动生成

MVCC Rules

Every DAOS I/O operation belongs to a transaction. If a user does not associate an operation with a transaction, DAOS regards this operation as a single-operation transaction. A conditional update, as defined above, is therefore regarded as a transaction comprising a conditional check, and if the check passes, an update, or punch operation.

每个 DAOS I/O 操作都属于一个事务。 如果用户没有将操作与事务相关联，DAOS 会将此操作视为单操作事务。 因此，如上定义的条件更新被视为包括条件检查的事务，并且如果检查通过，则更新或打孔操作。

Every transaction gets an epoch. Single-operation transactions and conditional updates get their epochs from the redundancy group servers they access, snapshot read transactions get their epoch from the snapshot records and every other transaction gets its epoch from the HLC of the first server it accesses. (Earlier implementations use client HLCs to choose epochs in the last case. To relax the clock synchronization requirement for clients, later implementations have moved to use server HLCs to choose epochs, while introducing client HLC Trackers that track the highest server HLC timestamps clients have heard of.) A transaction performs all operations using its epoch.

每笔交易都有一个epoch。 单操作事务和条件更新从它们访问的冗余组服务器获取它们的epoch，快照读取事务从快照记录中获取它们的epoch，每个其他事务从它访问的第一个服务器的 HLC 获取其epoch。 （早期的实现在最后一种情况下使用客户端 HLC 来选择epoch。为了放宽客户端的时钟同步要求，后来的实现已转移到使用服务器 HLC 来选择epoch，同时引入了客户端 HLC 跟踪器来跟踪客户端听到的最高服务器 HLC 时间戳 of.) 事务使用其epoch执行所有操作。

The MVCC rules ensure that transactions execute as if they are serialized in their epoch order while ensuring that every transaction observes all conflicting transactions commit before it opens, as long as the system clock offsets are always within the expected maximum system clock offset (epsilon). For convenience, the rules classify the I/O operations into reads and writes:

MVCC 规则确保事务按照它们的epoch顺序执行，同时确保每个事务在打开之前观察到所有冲突的事务提交，只要系统时钟偏移始终在预期的最大系统时钟偏移 (epsilon) 内。 为方便起见，规则将 I/O 操作分为读和写：

* Reads
  + Fetch akeys [akey level]
  + Check object emptiness [object level]
  + Check dkey emptiness [dkey level]
  + Check akey emptiness [akey level]
  + List objects under container [container level]
  + List dkeys under object [object level]
  + List akeys under dkey [dkey level]
  + List recx under akey [akey level]
  + Query min/max dkeys under object [object level]
  + Query min/max akeys under dkey [dkey level]
  + Query min/max recx under akey [akey level]
* Writes
  + Update akeys [akey level]
  + Punch akeys [akey level]
  + Punch dkey [dkey level]
  + Punch object [object level]

• 读取 o 获取key [key级别] o 检查对象空性 [对象级别] o 检查 dkey 是否为空 [dkey 级别] o 检查密钥空性 [akey level] o 列出容器下的对象[容器级别] o 列出对象 [对象级别] 下的 dkeys o 列出 dkey [dkey level] 下的 akeys o 列出 akey [akey level] 下的 recx o 查询对象[对象级别]下的最小/最大dkeys o 查询 dkey [dkey level] 下的 min/max akeys o 查询 akey 下的 min/max recx [akey level]

• 写入 o 更新密钥 [密钥级别] o Punch akeys [akey level] o 打孔 dkey [dkey 级别] o 打孔对象 [对象级别]

And each read or write is at one of the four levels: container, object, dkey, and akey. An operation is regarded as an access to the whole subtree rooted at its level. Although this introduces a few false conflicts (e.g., a list operation versus a lower level update that does not change the list result), the assumption simplifies the rules.

并且每次读取或写入都在四个级别之一：容器、对象、dkey 和 akey。 一个操作被认为是对以其级别为根的整个子树的访问。 尽管这会引入一些错误的冲突（例如，列表操作与不更改列表结果的较低级别更新），但该假设简化了规则。

A read at epoch e follows these rules:

epoch e 的读取遵循以下规则：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

A write at epoch e follows these rules:

图形用户界面, 文本, 应用程序

描述已自动生成

A transaction involving both reads and writes must follow both sets of rules. As optimizations, single-read transactions and snapshot (read) transactions do not need to update read timestamps. Snapshot creations, however, must update the read timestamps as if it is a transaction reading the whole container.

涉及读取和写入的事务必须遵循两组规则。 作为优化，单读事务和快照（读）事务不需要更新读时间戳。 但是，快照创建必须更新读取时间戳，就好像它是读取整个容器的事务一样。

When a transaction is rejected, it restarts with the same transaction ID but a higher epoch. If the epoch becomes higher than the original epoch plus epsilon, the epoch becomes certain, guaranteeing the restarts due to the epoch uncertainty checks are bounded.

当一个事务被拒绝时，它会以相同的事务 ID 重新启动，但具有更高的epoch。 如果epoch变得高于原始epoch加 epsilon，则epoch变得确定，保证由于epoch不确定性检查而重新启动是有界的。

Deadlocks among transactions are impossible. A transaction t\_1 with epoch e\_1 may block a transaction t\_2 with epoch e\_2 only when t\_2 needs to wait for t\_1's writes to commit. Since the client caching is used, t\_1 must be committing, whereas t\_2 may be reading or committing. If t\_2 is reading, then e\_1 <= e\_2. If t\_2 is committing, then e\_1 < e\_2. Suppose there is a cycle of transactions reaching a deadlock. If the cycle includes a committing-committing edge, then the epochs along the cycle must increase and then decrease, causing a contradiction. If all edges are committing-reading, then there must be two such edges together, causing a contradiction that a reading transaction cannot block other transactions. Deadlocks are, therefore, not a concern.

事务之间的死锁是不可能的。 仅当 t\_2 需要等待 t\_1 的写入提交时，具有 epoch e\_1 的事务 t\_1 可能会阻塞具有 epoch e\_2 的事务 t\_2。 由于使用了客户端缓存，t\_1 必须正在提交，而 t\_2 可能正在读取或提交。 如果 t\_2 正在读取，则 e\_1 <= e\_2。 如果 t\_2 正在提交，则 e\_1 < e\_2。 假设有一个事务循环达到死锁。 如果循环包含提交-提交边缘，那么循环中的epoch必须先增后减，从而导致矛盾。 如果所有的边都是提交-读，那么肯定有两个这样的边在一起，这就造成了一个读事务不能阻塞其他事务的矛盾。 因此，死锁不是问题。

If an entity keeps getting reads with increasing epochs, writes to this entity may keep being rejected due to the entity's ever-increasing read timestamps. Exponential backoffs with randomizations (see d\_backoff\_seq) have been introduced during daos\_tx\_restart calls. These are effective for dfs\_move workloads, where readers also write.

如果实体随着时间的增加不断获得读取，由于实体不断增加的读取时间戳，对该实体的写入可能会不断被拒绝。???? 在 daos\_tx\_restart 调用期间引入了具有随机化的指数退避（参见 d\_backoff\_seq）。 这些对于 dfs\_move 工作负载很有效，读者也在其中写入。

Punch propagation

Since conditional operations rely on an emptiness semantic, VOS read operations, particularly listing can be very expensive because they would require potentially reading the subtree to see if the entity is empty or not. In order to alieviate this problem, VOS instead does punch propagation. On a punch operation, the parent tree is read to see if the punch causes it to be empty. If it does, the parent tree is punched as well. Propagation presently stops at the dkey level, meaning the object will not be punched. Punch propagation only applies when punching keys, not values.

由于条件操作依赖于空语义，因此 VOS 读取操作，尤其是列表可能非常昂贵，因为它们可能需要读取子树以查看实体是否为空。 为了缓解这个问题，VOS 转而进行打孔传播。 在打孔操作中，读取父树以查看打孔是否导致其为空。 如果是这样，父树也会被打孔。 传播目前在 dkey 级别停止，这意味着对象不会被打孔。 冲压传播仅适用于冲压键，而不适用于值。

Epoch Based Operations/基于epoch的操作

Epochs provide a way for modifying VOS objects without destroying the history of updates/writes. Each update consumes memory and discarding unused history can help reclaim unused space. VOS provides methods to compact the history of writes/updates and reclaim space in every storage node. VOS also supports rollback of history in case transactions are aborted. The DAOS API timestamp corresponds to a VOS epoch. The API only allows reading either the latest state or from a persistent snapshot, which is simply a reference on a given epoch.

Epochs 提供了一种在不破坏更新/写入历史的情况下修改 VOS 对象的方法。 每次更新都会消耗内存，丢弃未使用的历史记录有助于回收未使用的空间。 VOS 提供了压缩写入/更新历史记录并回收每个存储节点中的空间的方法。 VOS 还支持在事务中止时回滚历史记录。 DAOS API 时间戳对应一个 VOS epoch。 API 只允许读取最新状态或从持久快照中读取，这只是给定epoch的引用。

To compact epochs, VOS allows all epochs between snapshots to be aggregated, i.e., the value/extent-data of the latest epoch of any key is always kept over older epochs. This also ensures that merging history does not cause loss of exclusive updates/writes made to an epoch. To rollback history, VOS provides the discard operation.

为了压缩epoch，VOS 允许聚合快照之间的所有epoch，即，任何键的最新epoch的值/范围数据始终保留在较旧的epoch上。 这也确保了合并历史不会导致对 epoch 进行的独占更新/写入丢失。 为了回滚历史，VOS 提供了丢弃操作。

int vos\_aggregate(daos\_handle\_t coh, daos\_epoch\_range\_t \*epr);

int vos\_discard(daos\_handle\_t coh, daos\_epoch\_range\_t \*epr);

int vos\_epoch\_flush(daos\_handle\_t coh, daos\_epoch\_t epoch);

Aggregate and discard operations in VOS accept a range of epochs to be aggregated normally corresponding to ranges between persistent snapshots.

VOS 中的聚合和丢弃操作接受一系列要聚合的epoch，通常对应于持久快照之间的范围。

VOS Discard/丢弃

Discard forcefully removes epochs without aggregation. This operation is necessary only when the value/extent-data associated with a pair needs to be discarded. During this operation, VOS looks up all objects associated with each cookie in the requested epoch range from the cookie index table and removes the records directly from the respective object trees by looking at their respective epoch validity. DAOS requires a discard to service abort requests. Abort operations require a discard to be synchronous.

Discard 强行删除epoch而不进行聚合。 仅当需要丢弃与一对关联的值/范围数据时，才需要此操作。 在此操作期间，VOS 从 cookie 索引表中查找与请求的 epoch 范围内的每个 cookie 关联的所有对象，并通过查看它们各自的 epoch 有效性直接从相应的对象树中删除记录。 DAOS 需要丢弃来服务中止请求。 中止操作需要丢弃同步。

During discard, keys and byte-array rectangles need to be searched for nodes/slots whose end-epoch is (discard\_epoch - 1). This means that there was an update before the now discarded epoch, and its validity got modified to support near-epoch lookup. This epoch validity of the previous update has to be extended to infinity to ensure future lookups at near-epoch would fetch the last known updated value for the key/extent range.

在丢弃期间，需要在键和字节数组矩形中搜索结束时期为 (discard\_epoch - 1) 的节点/槽。 这意味着在现在被丢弃的 epoch 之前有一个更新，它的有效性被修改以支持近 epoch 查找。 必须将先前更新的这个epoch有效性扩展到无穷大，以确保未来在近epoch的查找将获取键/范围范围的最后一个已知更新值。

VOS Aggregate/聚合

During aggregation, VOS must retain the latest update to a key/extent-range discarding the others and any updates visible at a persistent snapshot. VOS can freely remove or consolidate keys or extents so long as it doesn't alter the view visible at the latest timestamp or any persistent snapshot epoch. Aggregation makes use of the vos\_iterate API to find both visible and hidden entries between persistent snapshots and removes hidden keys and extents and merges contiguous partial extents to reduce metadata overhead. Aggregation can be an expensive operation but doesn't need to consume cycles on the critical path. A special aggregation ULT processes aggregation, frequently yielding to avoid blocking the continuing I/O.

在聚合期间，VOS 必须保留对键/范围范围的最新更新，丢弃其他内容以及在持久快照中可见的任何更新。 VOS 可以自由删除或合并键或范围，只要它不改变在最新时间戳或任何持久快照时期可见的视图。 聚合使用 vos\_iterate API 来查找持久快照之间的可见和隐藏条目，并删除隐藏的键和范围并合并连续的部分范围以减少元数据开销。 聚合可能是一项昂贵的操作，但不需要在关键路径上消耗周期。 一个特殊的聚合 ULT 处理聚合，经常让步以避免阻塞持续的 I/O。

VOS Checksum Management/校验和管理

VOS is responsible for storing checksums during an object update and retrieve checksums on an object fetch. Checksums will be stored with other VOS metadata in storage class memory. For Single Value types, a single checksum is stored. For Array Value types, multiple checksums can be stored based on the chunk size.

VOS 负责在对象更新期间存储校验和，并在对象获取时检索校验和。 校验和将与其他 VOS 元数据一起存储在存储类内存中。 对于单值类型，存储单个校验和。 对于数组值类型，可以根据块大小存储多个校验和。

The **Chunk Size** is defined as the maximum number of bytes of data that a checksum is derived from. While extents are defined in terms of records, the chunk size is defined in terms of bytes. When calculating the number of checksums needed for an extent, the number of records and the record size is needed. Checksums should typically be derived from Chunk Size bytes, however, if the extent is smaller than Chunk Size or an extent is not "Chunk Aligned," then a checksum might be derived from bytes smaller than Chunk Size.

块大小定义为校验和所源自的最大数据字节数。 虽然范围是根据记录定义的，但块大小是根据字节定义的。 在计算一个范围所需的校验和数时，需要记录数和记录大小。 校验和通常应从块大小字节派生，但是，如果范围小于块大小或范围不是“块对齐”，则校验和可能从小于块大小的字节派生。

The **Chunk Alignment** will have an absolute offset, not an I/O offset. So even if an extent is exactly, or less than, Chunk Size bytes long, it may have more than one Chunk if it crosses the alignment barrier.

块对齐将具有绝对偏移量，而不是 I/O 偏移量。 因此，即使范围正好或小于 Chunk Size 字节长，如果它跨越对齐障碍，它也可能有多个 Chunk。

Configuration

Checksums will be configured for a container when a container is created. Checksum specific properties can be included in the daos\_cont\_create API. This configuration has not been fully implemented yet, but properties might include checksum type, chunk size, and server side verification.

创建容器时，将为容器配置校验和。 校验和特定属性可以包含在 daos\_cont\_create API 中。 此配置尚未完全实现，但属性可能包括校验和类型、块大小和服务器端验证。

Storage

Checksums will be stored in a record(vos\_irec\_df) or extent(evt\_desc) structure for Single Value types and Array Value types respectfully. Because the checksum can be of variable size, depending on the type of checksum configured, the checksum itself will be appended to the end of the structure. The size needed for checksums is included while allocating memory for the persistent structures on SCM (vos\_reserve\_single/vos\_reserve\_recx).

校验和将分别存储在单值类型和数组值类型的记录（vos\_irec\_df）或范围（evt\_desc）结构中。 由于校验和的大小可变，取决于配置的校验和的类型，校验和本身将附加到结构的末尾。 在为 SCM (vos\_reserve\_single/vos\_reserve\_recx) 上的持久结构分配内存时，包括校验和所需的大小。

The following diagram illustrates the overall VOS layout and where checksums will be stored. Note that the checksum type isn't actually stored in vos\_cont\_df yet.

下图说明了整个 VOS 布局以及校验和的存储位置。 请注意，校验和类型实际上尚未存储在 vos\_cont\_df 中。

图片包含 文本

描述已自动生成

Checksum VOS Flow (vos\_obj\_update/vos\_obj\_fetch)校验和 VOS 流程 (vos\_obj\_update/vos\_obj\_fetch)

On update, the checksum(s) are part of the I/O Descriptor. Then, in akey\_update\_single/akey\_update\_recx, the checksum buffer pointer is included in the internal structures used for tree updates (vos\_rec\_bundle for SV and evt\_entry\_in for EV). As already mentioned, the size of the persistent structure allocated includes the size of the checksum(s). Finally, while storing the record (svt\_rec\_store) or extent (evt\_insert), the checksum(s) are copied to the end of the persistent structure.

更新时，校验和是 I/O 描述符的一部分。 然后，在 akey\_update\_single/akey\_update\_recx 中，校验和缓冲区指针包含在用于树更新的内部结构中（SV 的 vos\_rec\_bundle 和 EV 的 evt\_entry\_in）。 如前所述，分配的持久结构的大小包括校验和的大小。 最后，在存储记录 (svt\_rec\_store) 或范围 (evt\_insert) 时，校验和被复制到持久结构的末尾。

On a fetch, the update flow is essentially reversed.

在获取时，更新流程本质上是相反的。

For reference, key junction points in the flows are:

作为参考，流程中的关键连接点是：

* SV Update: vos\_update\_end -> akey\_update\_single -> svt\_rec\_store
* Sv Fetch: vos\_fetch\_begin -> akey\_fetch\_single -> svt\_rec\_load
* EV Update: vos\_update\_end -> akey\_update\_recx -> evt\_insert
* EV Fetch: vos\_fetch\_begin -> akey\_fetch\_recx -> evt\_fill\_entry

Metadata Overhead/元数据开销

There is a tool available to estimate the metadata overhead. It is described on the [storage estimator](https://github.com/daos-stack/daos/blob/master/src/client/storage_estimator/README.md) section.

有一个工具可用于估计元数据开销。 它在存储估计器部分进行了描述。

Replica Consistency

DAOS supports multiple replicas for data high availability. Inconsistency between replicas is possible when a target fails during an update to a replicated object and when concurrent updates are applied on replicated targets in an inconsistent order.

DAOS 支持多个副本以实现数据高可用性。 当目标在更新复制对象期间失败以及以不一致的顺序在复制目标上应用并发更新时，副本之间可能存在不一致。

The most intuitive solution to the inconsistency problem is distributed lock (DLM), used by some distributed systems, such as Lustre. For DAOS, a user-space system with powerful, next generation hardware, maintaining distributed locks among multiple, independent application spaces will introduce unacceptable overhead and complexity. DAOS instead uses an optimized two-phase commit transaction to guarantee consistency among replicas.

不一致问题最直观的解决方案是分布式锁（DLM），一些分布式系统使用，例如Lustre。 对于 DAOS，具有强大的下一代硬件的用户空间系统，在多个独立的应用程序空间之间维护分布式锁将引入不可接受的开销和复杂性。 DAOS 使用优化的两阶段提交事务来保证副本之间的一致性。

Single redundancy group based DAOS Two-Phase Commit (DTX) 基于单冗余组的 DAOS 两阶段提交 (DTX)

When an application wants to modify (update or punch) a multiple replicated object or EC object, the client sends the modification RPC to the leader shard (via [DTX Leader Election](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#812) algorithm discussed

below). The leader dispatches the RPC to the other related shards, and each shard makes its modification in parallel. Bulk transfers are not forwarded by the leader but rather transferred directly from the client, improving load balance and decreasing latency by utilizing the full client-server bandwidth.

当应用程序想要修改（更新或打孔）多个复制对象或 EC 对象时，客户端将修改 RPC 发送到领导者分片（通过下面讨论的 DTX 领导者选举算法）。 领导者将 RPC 调度到其他相关分片，每个分片并行进行修改。 批量传输不是由领导者转发，而是直接从客户端传输，通过利用完整的客户端 - 服务器带宽来改善负载平衡并减少延迟。

Before modifications are made, a local transaction, called 'DTX', is started on each related shard (both leader and non-leaders) with a client generated DTX identifier that is unique for the modification within the container. All the modifications in a DTX are logged in the DTX transaction table and back references to the table are kept in related modified record. After local modifications are done, each non-leader marks the DTX state as 'prepared' and replies to the leader. The leader sets the DTX state to 'committable' as soon as it has completed its modifications and has received successful replies from all non-leaders. If any shard(s) fail to execute the modification, it will reply to the leader with failure, and the leader will globally abort the DTX. Once the DTX is set by the leader to 'committable' or 'aborted', it replies to the client with the appropriate status.

在进行修改之前，一个名为“DTX”的本地事务在每个相关分片（领导者和非领导者）上启动，客户端生成的 DTX 标识符对于容器内的修改是唯一的。 DTX 中的所有修改都记录在 DTX 事务表中，对该表的反向引用保存在相关的修改记录中。 本地修改完成后，每个非领导者将 DTX 状态标记为“准备好”并回复领导者。 领导者完成修改并收到所有非领导者的成功回复后，立即将 DTX 状态设置为“可提交”。 如果任何一个shard（s）执行修改失败，它会回复leader失败，leader将全局中止DTX。 一旦领导者将 DTX 设置为“可提交”或“中止”，它就会以适当的状态回复客户端。

The client may consider a modification complete as soon as it receives a successful reply from the leader, regardless of whether the DTX is actually 'committed' or not. It is the responsibility of the leader to commit the 'committable' DTX asynchronously. This can happen if the 'committable' count or DTX age exceed some thresholds or the DTX is piggybacked via other dispatched RPCs due to potential conflict with subsequent modifications.

客户端可以在收到领导者的成功回复后立即认为修改完成，无论 DTX 是否实际“提交”。 领导者有责任异步提交“可提交的”DTX。 如果“可提交”计数或 DTX 年龄超过某些阈值，或者由于与后续修改的潜在冲突，DTX 被其他分派的 RPC 捎带，则可能发生这种情况。

When an application wants to read something from an object with multiple replicas, the client can send the RPC to any replica. On the server side, if the related DTX has been committed or is committable, the record can be returned to. If the DTX state is prepared, and the replica is not the leader, it will reply to the client telling it to send the RPC to the leader instead. If it is the leader and is in the state 'committed' or 'committable', then such entry is visible to the application. Otherwise, if the DTX on the leader is also 'prepared', then for transactional read, ask the client to wait and retry via returning -DER\_INPROGRESS; for non-transactional read, related entry is ignored and the latest committed modification is returned to the client.

当应用程序想要从具有多个副本的对象中读取某些内容时，客户端可以将 RPC 发送到任何副本。 在服务器端，如果相关的 DTX 已经提交或可提交，则可以返回记录。 如果 DTX 状态准备好，并且副本不是领导者，它将回复客户端，告诉它改为将 RPC 发送给领导者。 如果它是领导者并且处于“已提交”或“可提交”状态，则该条目对应用程序可见。 否则，如果领导者上的 DTX 也“准备好了”，那么对于事务性读取，通过返回 -DER\_INPROGRESS 要求客户端等待并重试； 对于非事务性读取，相关条目将被忽略，并将最新提交的修改返回给客户端。

If the read operation refers to an EC object and the data read from a data shard (non-leader) has a 'prepared' DTX, the data may be 'committable' on the leader due to the aforementioned asynchronous batched commit mechanism. In such case, the non-leader will refresh related DTX status with the leader. If the DTX status after refresh is 'committed', then related data can be returned to the client; otherwise, if the DTX state is still 'prepared', then for transactional read, ask the client to wait and retry via returning -DER\_INPROGRESS; for non-transactional read, related entry is ignored and the latest committed modification is returned to the client.

如果读取操作引用一个 EC 对象并且从数据分片（非领导者）读取的数据具有“准备好的”DTX，则由于上述异步批量提交机制，数据可能在领导者上是“可提交的”。 在这种情况下，非领导者将与领导者刷新相关的 DTX 状态。 如果刷新后的DTX状态为'committed'，则可以将相关数据返回给客户端； 否则，如果 DTX 状态仍然是“准备好的”，那么对于事务性读取，通过返回 -DER\_INPROGRESS 要求客户端等待并重试； 对于非事务性读取，相关条目将被忽略，并将最新提交的修改返回给客户端。

The DTX model is built inside a DAOS container. Each container maintains its own DTX table that is organized as two B+trees in SCM: one for active DTXs and the other for committed DTXs. The following diagram represents the modification of a replicated object under the DTX model.

DTX 模型构建在 DAOS 容器内。 每个容器维护自己的 DTX 表，该表在 SCM 中组织为两个 B+树：一个用于活动 DTX，另一个用于提交的 DTX。 下图表示在 DTX 模型下对复制对象的修改。

**Modify multiple replicated object under DTX model**

修改DTX模型下的多个复制对象

图示

描述已自动生成

Single redundancy group based DTX Leader Election/基于单冗余组的DTX Leader选举

In single redundancy group based DTX model, the leader selection is done for each object or dkey following these general guidelines:

R1: When different replicated objects share the same redundancy group, the same leader should not be used for each object.

R2: When a replicated object with multiple DKEYs span multiple redundancy groups, the leaders in different redundancy groups should be on different servers.

R3: Servers that fail frequently should be avoided in leader selection to avoid frequent leader migration.

R4: For EC object, the leader will be one of the parity nodes within current redundancy group.

在基于单冗余组的 DTX 模型中，按照以下一般准则为每个对象或 dkey 完成领导选择： R1：当不同的复制对象共享同一个冗余组时，不应为每个对象使用相同的领导者。 R2：当具有多个DKEY的复制对象跨越多个冗余组时，不同冗余组中的leader应该在不同的服务器上。 R3：在leader选择中应该避免频繁出现故障的服务器，以避免频繁的leader迁移。 R4：对于 EC 对象，leader 将是当前冗余组中的奇偶校验节点之一。

[Blob I/O (BIO)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md)

The Blob I/O (BIO) module was implemented for issuing I/O over NVMe SSDs. The BIO module covers NVMe SSD support, faulty device detection, device health monitoring, NVMe SSD hot plug functionality, and also SSD identification with the use of Intel VMD devices.

Blob I/O (BIO) 模块用于通过 NVMe SSD 发出 I/O。 BIO 模块涵盖 NVMe SSD 支持、故障设备检测、设备运行状况监控、NVMe SSD 热插拔功能以及使用 Intel VMD 设备进行 SSD 识别。

This document contains the following sections:

本文档包含以下部分

* [NVMe SSD Support](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#1)
  + [Storage Performance Development Kit (SPDK)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#2)
  + [Per-Server Metadata Management (SMD)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#3)
  + [DMA Buffer Management](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#4)
* [NVMe Threading Model](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#5)
* [Device Health Monitoring](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#6)
* [Faulty Device Detection (SSD Eviction)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#7)
* [NVMe SSD Hot Plug](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#8)
* [SSD Identification](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#9)
  + [Intel Volume Management Device (VMD)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#10)
* [Device States](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#11)
* [User Interfaces](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#12)

• NVMe SSD 支持 o 存储性能开发套件 (SPDK) o 每服务器元数据管理 (SMD) o DMA 缓冲区管理

• NVMe 线程模型

• 设备健康监控

• 故障设备检测（SSD 驱逐）

• NVMe SSD 热插拔

• SSD 识别 o 英特尔卷管理设备 (VMD)

• 设备状态

• 用户界面

NVMe SSD Support

The DAOS service has two tiers of storage: Storage Class Memory (SCM) for byte-granular application data and metadata, and NVMe for bulk application data. Similar to how PMDK is currently used to facilitate access to SCM, the Storage Performance Development Kit (SPDK) is used to provide seamless and efficient access to NVMe SSDs. DAOS storage allocations can occur on either SCM by using a PMDK pmemobj pool, or on NVMe, using an SPDK blob. All local server metadata will be stored in a per-server pmemobj pool on SCM and will include all current and relevant NVMe device, pool, and xstream mapping information. Background aggregation allows for data migration from SCM to an NVMe SSD by coalescing smaller data records into a larger one. The DAOS control plane handles all SSD configuration, and the DAOS data plane handles all allocations through SPDK, with finer block allocations using the in-house Versioned Extent Allocator (VEA).

DAOS 服务有两层存储：用于字节粒度应用程序数据和元数据的存储类内存 (SCM)，以及用于批量应用程序数据的 NVMe。 类似于 PMDK 目前用于促进对 SCM 的访问，存储性能开发套件 (SPDK) 用于提供对 NVMe SSD 的无缝和高效访问。 DAOS 存储分配可以在 SCM 上使用 PMDK pmemobj 池进行，也可以在 NVMe 上使用 SPDK blob 进行。 所有本地服务器元数据都将存储在 SCM 上的每服务器 pmemobj 池中，并将包括所有当前和相关的 NVMe 设备、池和 xstream 映射信息。 后台聚合允许将较小的数据记录合并为较大的数据记录，从而将数据从 SCM 迁移到 NVMe SSD。 DAOS 控制平面处理所有 SSD 配置，DAOS 数据平面通过 SPDK 处理所有分配，使用内部版本化范围分配器 (VEA) 进行更精细的块分配。

Storage Performance Development Kit (SPDK)

SPDK is an open source C library that when used in a storage application, can provide a significant performance increase of more than 7X over the standard NVMe kernel driver. SPDK's high performance can mainly be attributed to the user space NVMe driver, eliminating all syscalls and enabling zero-copy access from the application. In SPDK, the hardware is polled for completions as opposed to relying on interrupts, lowering both total latency and latency variance. SPDK also offers a block device layer called bdev which sits immediately above the device drivers like in a traditional kernel storage stack. This module offers pluggable module APIs for implementing block devices that interface with different types of block storage devices. This includes driver modules for NVMe, Malloc (ramdisk), Linux AIO, Ceph RBD, and others.

SPDK 是一个开源 C 库，在存储应用程序中使用时，可以提供比标准 NVMe 内核驱动程序高 7 倍以上的显着性能提升。 SPDK 的高性能主要归功于用户空间 NVMe 驱动程序，消除了所有系统调用并实现了应用程序的零拷贝访问。 在 SPDK 中，硬件轮询完成而不是依赖中断，从而降低了总延迟和延迟差异。 SPDK 还提供了一个名为 bdev 的块设备层，它位于设备驱动程序的正上方，就像在传统的内核存储堆栈中一样。 该模块提供可插拔模块 API，用于实现与不同类型块存储设备接口的块设备。 这包括用于 NVMe、Malloc（ramdisk）、Linux AIO、Ceph RBD 等的驱动程序模块。

图形用户界面

低可信度描述已自动生成

SPDK NVMe Driver

The NVMe driver is a C library linked to a storage application providing direct, zero-copy data transfer to and from NVMe SSDs. Other benefits of the SPDK NVMe driver are that it is entirely in user space, operates in polled-mode vs. interrupt-dependent, is asynchronous and lock-less.

NVMe 驱动程序是一个链接到存储应用程序的 C 库，提供与 NVMe SSD 之间的直接、零拷贝数据传输。 SPDK NVMe 驱动程序的其他优点是它完全在用户空间中，以轮询模式与中断相关模式运行，是异步和无锁的。

SPDK Block Device Layer (bdev)

The bdev directory contains a block device abstraction layer used to translate from a common block protocol to specific protocols of backend devices, such as NVMe. Additionally, this layer provides automatic queuing of I/O requests in response to certain conditions, lock-less sending of queues, device configuration and reset support, and I/O timeout trafficking.

bdev 目录包含一个块设备抽象层，用于将通用块协议转换为后端设备的特定协议，例如 NVMe。 此外，该层提供 I/O 请求自动排队以响应某些条件、队列的无锁发送、设备配置和重置支持以及 I/O 超时交易。

SPDK Blobstore

The blobstore is a block allocator for a higher-level storage service. The allocated blocks are termed 'blobs' within SPDK. Blobs are designed to be large (at least hundreds of KB), and therefore another allocator is needed in addition to the blobstore to provide efficient small block allocation for the DAOS service. The blobstore provides asynchronous, un-cached, and parallel blob read and write interfaces

Blobstore 是用于更高级别存储服务的块分配器。 分配的块在 SPDK 中称为“blob”。 Blob 被设计为很大（至少数百 KB），因此除了 blobstore 之外还需要另一个分配器来为 DAOS 服务提供高效的小块分配。 blobstore 提供异步、非缓存和并行 blob 读写接口

SPDK Integration

The BIO module relies on the SPDK API to initialize/finalize the SPDK environment on the DAOS server start/shutdown. The DAOS storage model is integrated with SPDK by the following:

BIO 模块依赖 SPDK API 在 DAOS 服务器启动/关闭时初始化/完成 SPDK 环境。 DAOS 存储模型通过以下方式与 SPDK 集成：

* Management of SPDK blobstores and blobs: NVMe SSDs are assigned to each DAOS server xstream. SPDK blobstores are created on each NVMe SSD. SPDK blobs are created and attached to each per-xstream VOS pool.
* Association of SPDK I/O channels with DAOS server xstreams: Once SPDK I/O channels are properly associated to the corresponding device, NVMe hardware completion pollers are integrated into server polling ULTs.

• SPDK blobstore 和 blob 的管理：NVMe SSD 分配给每个 DAOS 服务 xstream。 SPDK blobstore 是在每个 NVMe SSD 上创建的。 SPDK blob 被创建并附加到每个 per-xstream VOS 池。

• SPDK I/O 通道与DAOS 服务器xstreams 的关联：一旦SPDK I/O 通道正确关联到相应的设备，NVMe 硬件完成轮询器就会集成到服务器轮询ULT 中。

Per-Server Metadata Management (SMD)

One of the major subcomponenets of the BIO module is per-server metadata management. The SMD submodule consists of a PMDK pmemobj pool stored on SCM used to track each DAOS server's local metadata.

BIO 模块的主要子组件之一是每服务器元数据管理。 SMD 子模块由存储在 SCM 上的 PMDK pmemobj 池组成，用于跟踪每个 DAOS 服务器的本地元数据。

Currently, the persistent metadata tables tracked are :

* **NVMe Device Table**: NVMe SSD to DAOS server xstream mapping (local PCIe attached NVMe SSDs are assigned to different server xstreams to avoid hardware contention). A persistent device state is also stored (supported device states are: NORMAL and FAULTY).
* **NVMe Pool Table**: NVMe SSD, DAOS server xstream, and SPDK blob ID mapping (SPDK blob to VOS pool:xstream mapping). Blob size is also stored along with the SPDK blob ID in order to support creating new blobs on a new device in the case of NVMe device hotplug.

目前，跟踪的持久元数据表是：

• NVMe 设备表：NVMe SSD 到 DAOS 服务xstream 映射（本地 PCIe 连接的 NVMe SSD 分配给不同的服务 xstream 以避免硬件争用）。 还存储持久设备状态（支持的设备状态是：正常和故障）。

• NVMe 池表：NVMe SSD、DAOS 服务 xstream 和 SPDK blob ID 映射（SPDK blob 到 VOS 池：xstream 映射）。 Blob 大小也与 SPDK Blob ID 一起存储，以便在 NVMe 设备热插拔的情况下支持在新设备上创建新 Blob。

On DAOS server start, these tables are loaded from persistent memory and used to initialize new, and load any previous blobstores and blobs. Also, there is potential to expand this module to support other non-NVMe related metadata in the future.

Useful admin commands to query per-server metadata: [dmg storage query (list-devices | list-pools)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#80) [used to query both SMD device table and pool table]

在 DAOS 服务器启动时，这些表从持久内存加载并用于初始化新的，并加载任何以前的 blobstores 和 blob。 此外，未来还有可能扩展此模块以支持其他非 NVMe 相关的元数据。 查询每个服务器元数据的有用管理命令：dmg storage query (list-devices | list-pools) [用于查询 SMD 设备表和池表]

DMA Buffer Management

BIO internally manages a per-xstream DMA safe buffer for SPDK DMA transfer over NVMe SSDs. The buffer is allocated using the SPDK memory allocation API and can dynamically grow on demand. This buffer also acts as an intermediate buffer for RDMA over NVMe SSDs, meaning on DAOS bulk update, client data will be RDMA transferred to this buffer first, then the SPDK blob I/O interface will be called to start local DMA transfer from the buffer directly to NVMe SSD. On DAOS bulk fetch, data present on the NVMe SSD will be DMA transferred to this buffer first, and then RDMA transferred to the client.

BIO 在内部管理每个 xstream DMA 安全缓冲区，用于通过 NVMe SSD 进行 SPDK DMA 传输。 缓冲区是使用 SPDK 内存分配 API 分配的，可以按需动态增长。 此缓冲区还充当 RDMA over NVMe SSD 的中间缓冲区，这意味着在 DAOS 批量更新时，客户端数据将首先通过 RDMA 传输到此缓冲区，然后将调用 SPDK blob I/O 接口以从缓冲区开始本地 DMA 传输 直接连接到 NVMe SSD。 在 DAOS 批量提取中，NVMe SSD 上的数据将首先通过 DMA 传输到此缓冲区，然后通过 RDMA 传输到客户端。

NVMe Threading Model

* Device Owner Xstream: In the case there is no direct 1:1 mapping of VOS XStream to NVMe SSD, the VOS xstream that first opens the SPDK blobstore will be named the 'Device Owner'. The Device Owner Xstream is responsible for maintaining and updating the blobstore health data, handling device state transitions, and also media error events. All non-owner xstreams will forward events to the device owner.
* Init Xstream: The first started VOS xstream is termed the 'Init Xstream'. The init xstream is responsible for initializing and finalizing the SPDK bdev, registering the SPDK hotplug poller, handling and periodically checking for new NVMe SSD hot remove and hotplug events, and handling all VMD LED device events.

• 设备所有者 Xstream：如果没有 VOS XStream 到 NVMe SSD 的直接 1:1 映射，首先打开 SPDK blobstore 的 VOS xstream 将被命名为“设备所有者”。 设备所有者 Xstream 负责维护和更新 blobstore 健康数据、处理设备状态转换以及媒体错误事件。 所有非所有者 xstreams 都会将事件转发给设备所有者。

• Init Xstream：第一个启动的VOS xstream 称为“Init Xstream”。 init xstream 负责初始化和最终确定 SPDK bdev、注册 SPDK 热插拔轮询器、处理和定期检查新的 NVMe SSD 热拔出和热插拔事件，以及处理所有 VMD LED 设备事件。

图示

描述已自动生成

Above is a diagram of the current NVMe threading model. The 'Device Owner' xstream is responsible for all faulty device and device reintegration callbacks, as well as updating device health data. The 'Init' xstream is responsible for registering the SPDK hotplug poller and maintaining the current device list of SPDK bdevs as well as evicted and unplugged devices. Any device metadata operations or media error events that do not occur on either of these two xstreams will be forwarded to the appropriate xstream using the SPDK event framework for lockless inter-thread communication. All xstreams will periodically poll for I/O statistics (if enabled in server config), but only the device owner xstream will poll for device events, making necessary state transitions, and update device health stats, and the init xstream will poll for any device removal or device hot plug events.

上图是当前 NVMe 线程模型的示意图。 “设备所有者”xstream 负责所有故障设备和设备重新集成回调，以及更新设备运行状况数据。 'Init' xstream 负责注册 SPDK 热插拔轮询器并维护 SPDK bdev 的当前设备列表以及被驱逐和拔出的设备。 任何未发生在这两个 xstream 上的设备元数据操作或媒体错误事件都将使用 SPDK 事件框架转发到适当的 xstream，以进行无锁线程间通信。 所有 xstreams 将定期轮询 I/O 统计信息（如果在服务器配置中启用），但只有设备所有者 xstream 将轮询设备事件，进行必要的状态转换，并更新设备健康状态，而 init xstream 将轮询任何设备 移除或设备热插拔事件。

Device Health Monitoring

The device owner xstream is responsible for maintaining anf updating all device health data and all media error events as apart of the device health monitoring feature. Device health data consists of raw SSD health stats queried via SPDK admin APIs and in-memory health data. The raw SSD health stats returned include useful and critical data to determine the current health of the device, such as temperature, power on duration, unsafe shutdowns, critical warnings, etc. The in-memory health data contains a subset of the raw SSD health stats, in addition to I/O error (read/write/unmap) and checksum error counters that are updated when a media error event occurs on a device and stored in-memory.

设备所有者 xstream 负责维护和更新所有设备健康数据和所有媒体错误事件，作为设备健康监控功能的一部分。 设备运行状况数据包括通过 SPDK 管理 API 查询的原始 SSD 运行状况统计数据和内存中运行状况数据。 返回的原始 SSD 运行状况统计信息包括用于确定设备当前运行状况的有用和关键数据，例如温度、开机持续时间、不安全关机、严重警告等。内存中运行状况数据包含原始 SSD 运行状况的子集 stats，以及 I/O 错误（读/写/取消映射）和校验和错误计数器，这些计数器在设备上发生媒体错误事件并存储在内存中时更新。

The DAOS data plane will monitor NVMe SSDs every 60 seconds, including updating the health stats with current values, checking current device states, and making any necessary blobstore/device state transitions. Once a FAULTY state transition has occurred, the monitoring period will be reduced to 10 seconds to allow for quicker transitions and finer-grained monitoring until the device is fully evicted.

DAOS 数据平面将每 60 秒监控一次 NVMe SSD，包括使用当前值更新运行状况统计信息、检查当前设备状态以及进行任何必要的 blobstore/设备状态转换。 一旦发生 FAULTY 状态转换，监视周期将减少到 10 秒，以实现更快的转换和更细粒度的监视，直到设备完全被逐出。

Useful admin commands to query device health: 查询设备运行状况的有用管理命令

* [dmg storage query (device-health | target-health)](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#81) [used to query SSD health stats]

While monitoring this health data, an admin can now make the determination to manually evict a faulty device. This data will also be used to set the faulty device criteria for automatic SSD eviction (available in a future release). 在监控此运行状况数据的同时，管理员现在可以决定手动驱逐故障设备。 此数据还将用于设置自动 SSD 驱逐的故障设备标准（在未来版本中可用）。

Faulty Device Detection (SSD Eviction) 故障设备检测（SSD 驱逐）

Faulty device detection and reaction can be referred to as NVMe SSD eviction. This involves all affected pool targets being marked as down and the rebuild of all affected pool targets being automatically triggered. A persistent device state is maintained in SMD and the device state is updated from NORMAL to FAULTY upon SSD eviction. The faulty device reaction will involve various SPDK cleanup, including all I/O channels released, SPDK allocations (termed 'blobs') closed, and the SPDK blobstore created on the NVMe SSD unloaded. Currently only manual SSD eviction is supported, and a future release will support automatic SSD eviction.

故障设备检测和反应可称为 NVMe SSD 驱逐。 这涉及将所有受影响的池targets标记为停机，并自动触发所有受影响的池targets的重建。 在 SMD 中维护持久的设备状态，并且在 SSD 驱逐时设备状态从 NORMAL 更新为 FAULTY。 故障设备反应将涉及各种 SPDK 清理，包括释放所有 I/O 通道、关闭 SPDK 分配（称为“blob”）以及卸载在 NVMe SSD 上创建的 SPDK blobstore。 目前仅支持手动 SSD 驱逐，未来版本将支持自动 SSD 驱逐。

Useful admin commands to manually evict an NVMe SSD:

* [dmg storage set nvme-faulty](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#82) [used to manually set an NVMe SSD to FAULTY (ie evict the device)]

手动驱逐 NVMe SSD 的有用管理命令：

• dmg storage set nvme-faulty [用于手动将 NVMe SSD 设置为 FAULTY（即驱逐设备）]

NVMe SSD Hot Plug

**Full NVMe hot plug capability will be available and supported in DAOS 2.0 release. Use is currently intended for testing only and is not supported for production.**

DAOS 2.0 版本将提供并支持完整的 NVMe 热插拔功能。 目前仅用于测试，不支持生产。

The NVMe hot plug feature includes device removal (an NVMe hot remove event) and device reintegration (an NVMe hotplug event) when a faulty device is replaced with a new device.

NVMe 热插拔功能包括设备移除（NVMe 热移除事件）和设备重新集成（NVMe 热插拔事件），当故障设备更换为新设备时。

For device removal, if the device is a faulty or previously evicted device, then nothing further would be done when the device is removed. The device state would be displayed as UNPLUGGED. If a healthy device that is currently in use by DAOS is removed, then all SPDK memory stubs would be deconstructed, and the device state would also display as UNPLUGGED.

对于设备移除，如果该设备是有故障的或先前被驱逐的设备，则移除该设备后将不再进行任何操作。 设备状态将显示为 UNPLUGGED。 如果 DAOS 当前正在使用的健康设备被移除，那么所有 SPDK 内存存根都将被解构，并且设备状态也将显示为 UNPLUGGED。

For device reintegration, if a new device is plugged to replace a faulty device, the admin would need to issue a device replacement command. All SPDK in-memory stubs would be created and all affected pool targets automatically reintegrated on the new device. The device state would be displayed as NEW initially and NORMAL after the replacement event occurred. If a faulty device or previously evicted device is re-plugged, the device will remain evicted, and the device state would display EVICTED. If a faulty device is desired to be reused (NOTE: this is not advised, mainly used for testing purposes), the admin can run the same device replacement command setting the new and old device IDs to be the same device ID. Reintegration will not occur on the device, as DAOS does not currently support incremental reintegration.

对于设备重新集成，如果插入新设备来更换故障设备，管理员将需要发出设备更换命令。 将创建所有 SPDK 内存存根，并将所有受影响的池targets自动重新集成到新设备上。 设备状态最初将显示为 NEW，在更换事件发生后将显示为 NORMAL。 如果重新插入有故障的设备或先前被驱逐的设备，该设备将保持被驱逐状态，并且设备状态将显示为 EVICTED。 如果需要重新使用故障设备（注意：不建议这样做，主要用于测试目的），管理员可以运行相同的设备更换命令，将新旧设备 ID 设置为相同的设备 ID。 设备上不会发生重新集成，因为 DAOS 目前不支持增量重新集成。

NVMe hot plug with Intel VMD devices is currently not supported in this release, but will be supported in a future release.

Useful admin commands to replace an evicted device:

* [dmg storage replace nvme](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#83) [used to replace an evicted device with a new device]
* [dmg storage replace nvme](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#84) [used to bring an evicted device back online (without reintegration)]

此版本当前不支持使用 Intel VMD 设备的 NVMe 热插拔，但将在未来版本中支持。 替换被驱逐设备的有用管理命令：

• dmg storage replace nvme [用于用新设备替换被驱逐的设备]

• dmg 存储替换 nvme [用于使被驱逐的设备重新联机（无需重新集成）]

SSD Identification/鉴定

The SSD identification feature is a way to quickly and visually locate a device. It requires the use of Intel VMD, which needs to be physically available on the hardware as well as enabled in the system BIOS. The feature supports two LED events: locating a healthy device and locating an evicted device.

SSD 识别功能是一种快速、直观地定位设备的方法。 它需要使用英特尔VMD，它需要在硬件上物理可用并在系统 BIOS 中启用。 该功能支持两个 LED 事件：定位健康设备和定位被驱逐的设备。

Intel Volume Management Device (VMD)

Intel VMD is a technology embedded in the processor silicon that aggregates the NVMe PCIe SSDs attached to its root port, acting as an HBA does for SATA and SAS. Currently, PCIe storage lacks a standardized method to blink LEDs and indicated the status of a device. Intel VMD, along with NVMe, provides this support for LED management.

英特尔 VMD 是一种嵌入在处理器芯片中的技术，可聚合连接到其根端口的 NVMe PCIe SSD，充当 HBA 对 SATA 和 SAS 的作用。 目前，PCIe 存储缺乏使 LED 闪烁和指示设备状态的标准化方法。 英特尔 VMD 与 NVMe 一起为 LED 管理提供这种支持。

图形用户界面, 图示

描述已自动生成

Intel VMD places a control point in the PCIe root complex of the servers, meaning that NVMe drives can be hot-swapped, and the status LED is always reliable.

英特尔 VMD 在服务器的 PCIe 根复合体中放置了一个控制点，这意味着 NVMe 驱动器可以热插拔，并且状态 LED 始终可靠。

图形用户界面

描述已自动生成

The Amber LED (status LED) is what VMD provides. It represents the LED coming from the slot on the backplane. The Green LED is the activity LED.

琥珀色 LED（状态 LED）是 VMD 提供的。 它代表来自背板上插槽的 LED。 绿色 LED 是活动 LED。

The status LED on the VMD device has four states: OFF, FAULT, REBUILD, and IDENTIFY. These are communicated by blinking patterns specified

VMD 设备上的状态 LED 有四种状态：OFF、FAULT、REBUILD 和 IDENTIFY。 这些是通过指定的闪烁模式传达的

表格

描述已自动生成

Locate a Health Device/ Locate a Health Device

Upon issuing a device identify command with a specified device ID, an admin now can quickly identify a device in question. The status LED on the VMD device would be set to an IDENTIFY state, represented by a quick, 4Hz blinking amber light. The device would quickly blink by default for 60 seconds and then return to the default OFF state. The LED event duration can be customized by setting the VMD\_LED\_PERIOD environment variable if a duration other than the default value is desired.

在发出具有指定设备 ID 的设备识别命令后，管理员现在可以快速识别有问题的设备。 VMD 设备上的状态 LED 将设置为 IDENTIFY 状态，由快速、 4Hz 闪烁的琥珀色灯表示。 默认情况下，设备会快速闪烁 60 秒，然后返回默认关闭状态。 如果需要非默认值的持续时间，可以通过设置 VMD\_LED\_PERIOD 环境变量来自定义 LED 事件持续时间。

Locate an Evicted Device/定位被驱逐的设备

If an NVMe SSD is evicted, the status LED on the VMD device will be set to a FAULT state, represented by a solidly ON amber light. No additional command apart from the SSD eviction would be needed, and this would visually indicate that the device needs to be replaced and is no longer in use by DAOS. The LED of the VMD device will remain in this state until replaced by a new device.

Useful admin command to locate a VMD-enabled NVMe SSD:

* [dmg storage identify vmd](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#85) [used to change the status LED state on the VMD device to quickly blink for 60 seconds]

如果 NVMe SSD 被逐出，VMD 设备上的状态 LED 将设置为 FAULT 状态，由常亮的琥珀色灯表示。 除了 SSD 驱逐之外，不需要任何其他命令，这将在视觉上表明该设备需要更换并且不再被 DAOS 使用。 VMD 设备的 LED 将保持此状态，直到被新设备更换。 用于定位启用 VMD 的 NVMe SSD 的有用管理命令：

• dmg storage identify vmd [用于将 VMD 设备上的 LED 状态更改为快速闪烁 60 秒]

Device States

The device states that are returned from a device query by the admin are dependent on both the persistently stored device state in SMD, and the in-memory BIO device list.

* NORMAL: A fully functional device in use by DAOS (or in setup).
* EVICTED: A device has been manually evicted and is no longer in use by DAOS.
* UNPLUGGED: A device previously used by DAOS is unplugged.
* NEW: A new device is available for use by DAOS.

管理员从设备查询返回的设备状态取决于 SMD 中永久存储的设备状态和内存中的 BIO 设备列表。

• NORMAL：DAOS 使用的全功能设备（或设置中）。

• EVICTED：设备已被手动驱逐并且不再被DAOS 使用。

• UNPLUGGED：DAOS 先前使用的设备已拔掉。

• 新：DAOS 可以使用新设备。

图示

描述已自动生成

Useful admin command to query device states:

* [dmg storage query list-devices](https://github.com/daos-stack/daos/blob/master/src/bio/README.md#31) [used to query NVMe SSD device states]

查询设备状态的有用管理命令： • dmg storage query list-devices [用于查询 NVMe SSD 设备状态]

User Interfaces:

* Query Per-Server Metadata (SMD): **$dmg storage query (list-devices | list-pools)**

To list all devices:

文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

[Algorithmic object placement](https://github.com/daos-stack/daos/blob/master/src/placement/README.md)/算法对象放置

DAOS uses the pool map to create a set of placement maps that are used to compute algorithmic object layouts and to drive consensus data distribution. This approach uses consistent hash based algorithms to generate object layout based on object ID, object schema, and one of the placement maps. DAOS uses a modular approach that allows different placement maps to be used by different objects to obtain the performance characteristics required by the application.

DAOS 使用池映射创建一组放置映射，用于计算算法对象布局并驱动共识数据分布。 这种方法使用一致的基于散列的算法来根据对象 ID、对象模式和放置图之一生成对象布局。 DAOS 使用模块化方法，允许不同的对象使用不同的布局图来获得应用程序所需的性能特征。

Placement Map/布局图

A placement map is essentially an abstracted and permuted pool map; it does not necessarily include all details of the pool map. Instead it only retains component relationships that can be used to distribute object shards for the resilience and performance requirements of the application.

放置图本质上是一个抽象和置换的池图； 它不一定包括游泳池地图的所有细节。 相反，它仅保留可用于分发对象分片以满足应用程序的弹性和性能要求的组件关系。

**Pool-map and placement maps**

图示

描述已自动生成

A placement map does not maintain a copy of status or any characteristics of the corresponding pool map components, but only references pool map components. Each time DAOS computes an object distribution based on a placement map, it also needs to check the corresponding component status and attributes from the pool map. This adds an extra step for indirect memory access, but can significantly reduce cache pollution and memory consumption when there are many placement maps but only one pool map in a DAOS pool.

放置映射不维护相应池映射组件的状态或任何特征的副本，而仅引用池映射组件。 DAOS 每次根据放置图计算对象分布时，还需要从池图中检查相应的组件状态和属性。 这为间接内存访问增加了一个额外的步骤，但是当 DAOS 池中有许多放置映射但池映射只有一个时，可以显着减少缓存污染和内存消耗。

As shown in the [figure](https://github.com/daos-stack/daos/blob/master/src/placement/README.md#f10.2), a storage pool may have multiple types of placement maps because different applications can have various fault tolerance and performance requirements. In addition, there can be many instances of the same placement map in order to accelerate rebuild and rebalance by workload declustering.

如图所示，一个存储池可能有多种类型的放置图，因为不同的应用程序可能有不同的容错和性能要求。 此外，为了通过工作负载分簇加速重建和重新平衡，可以有许多相同放置图的实例。

DAOS today includes two placement map algorithms:

今天的 DAOS 包括两种布局图算法：

[**Jump\_Placement\_Map**](https://github.com/daos-stack/daos/blob/master/src/placement/JUMP_MAP.md)**.doc**

The Jump Placement Map is the default placement map in DAOS. It utilizes the Jump Consistent Hashing algorithm in order to pseudorandomly distribute objects amongst different fault domains. This distributes them across fault domains as far apart from one another as possible in order to avoid data loss in the event of a failure affecting an entire fault domain. It was designed to efficiently move data between systems when the physical configuration of the system changes (i.e. more capacity is added).

Jump Placement Map 是 DAOS 中的默认布局图。 它利用 Jump Consistent Hashing 算法在不同的故障域之间伪随机地分布对象。 这将它们分布在彼此尽可能远离的容错域中，以避免在影响整个容错域的故障发生时丢失数据。 它旨在在系统的物理配置发生变化（即添加更多容量）时有效地在系统之间移动数据。

[**Ring\_Placement\_Map**](https://github.com/daos-stack/daos/blob/master/src/placement/RING_MAP.md)**.doc**

The Ring Placement Map was the original placement map developed for DAOS. It utilizes a ring memory structure that puts targets on the ring in a pattern such that given any random location on the ring, that location and its neighbors will be physically in separate fault domains. This makes it extremely fast to compute placement locations, but also makes it difficult to modify dynamically. It can not currently be used as it does not support several of the newer API methods required by DAOS - specifically those for server reintegration, drain, and addition.

Ring Placement Map 是为 DAOS 开发的原始布局图。 它利用一种环形存储器结构，将目标以某种模式放置在环上，以便给定环上的任何随机位置，该位置及其相邻位置将在物理上位于不同的故障域中。 这使得计算放置位置的速度非常快，但也很难动态修改。 它目前无法使用，因为它不支持 DAOS 所需的几种较新的 API 方法——特别是那些用于服务器重新集成、排空和添加的方法。

[Replicated database (RDB)](https://github.com/daos-stack/daos/blob/master/src/rdb/README.md)

Pool and container services are made highly available by replicating their internal metadata using Raft-based consensus and strong leadership. A service replicated in this generic approach tolerates the failure of any minority of its replicas. By spreading replicas of each service across the fault domains, pool and container services can therefore tolerate a reasonable number of target failures.

通过使用基于 Raft 的共识和强大的领导力复制其内部元数据，池和容器服务变得高度可用。 以这种通用方法复制的服务可以容忍其任何少数副本的故障。 通过跨故障域传播每个服务的副本，池和容器服务因此可以容忍合理数量的目标故障。

Architecture

A replicated service is built around a Raft replicated log. The service transforms RPCs into state queries and deterministic state updates. All state updates are committed to the replicated log first, before being applied by any of the service replicas. Since Raft guarantees consistency among log replicas, the service replicas end up applying the same set of state updates in the same order and go through identical state histories.

复制服务是围绕 Raft 复制日志构建的。 该服务将 RPC 转换为状态查询和确定性状态更新。 在被任何服务副本应用之前，所有状态更新都首先提交到复制日志。 由于 Raft 保证日志副本之间的一致性，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。

Among all replicas of a replicated service, only the current leader can handle service RPCs. The leader of a service is the current Raft leader (i.e., a Raft leader with the highest term number at the moment). Non-leader replicas reject all service RPCs and try to redirect the clients to the current leader to the best of their knowledge. Clients cache the addresses of the replicas as well as who current leader is. Occasionally, a client may not get any meaningful redirection hints and can find current leader by communicating to a random replicas.

在复制服务的所有副本中，只有当前领导者可以处理服务 RPC。 服务的领导者是当前的 Raft 领导者（即当前任期号最高的 Raft 领导者）。 非领导者副本拒绝所有服务 RPC，并尽其所知尝试将客户端重定向到当前领导者。 客户端缓存副本的地址以及当前的领导者是谁。 有时，客户端可能无法获得任何有意义的重定向提示，并且可以通过与随机副本通信来找到当前的领导者。

The [figure](https://github.com/daos-stack/daos/blob/master/src/rdb/README.md#f8.1) below shows the modules constituting a service replica. The service module handles RPCs by transforming them into state queries and deterministic state updates. The Raft module implements the replicated log following the Raft protocol, by communicating with Raft modules on other replicas. It provides methods for the service module to perform the queries and the updates. The storage module, which in this case is the persistent memory and the file system, stores the service and Raft state. It uses VOS to update the state stored in persistent memory atomically.

下图显示了构成服务副本的模块。 服务模块通过将 RPC 转换为状态查询和确定性状态更新来处理 RPC。 Raft 模块通过与其他副本上的 Raft 模块通信来实现遵循 Raft 协议的复制日志。 它为服务模块提供了执行查询和更新的方法。 存储模块，在这种情况下是持久内存和文件系统，存储服务和 Raft 状态。 它使用 VOS 以原子方式更新存储在持久内存中的状态。

**Service replication modules**服务复制模块

图示

描述已自动生成

RPC Handling

When an RPC request arrives at the leader, a service thread of the service module picks up the request and handles it by executing a handler function designed for this type of request. As far as service replication is concerned, a handler comprises state queries (e.g., reading the pool properties), state updates (e.g., writing a new version of the pool map), and RPCs to other services (e.g., CONT\_TGT\_DESTROY RPCs sent to target services). Some handlers involve only queries, some involve updates as well as queries, and others involve all three kinds of actions; rarely, if ever, do handlers involve only updates but no queries.

当 RPC 请求到达领导者时，服务模块的服务线程接收请求并通过执行为此类请求设计的处理程序函数来处理它。 就服务复制而言，处理程序包括状态查询（例如，读取池属性）、状态更新（例如，写入池映射的新版本）以及到其他服务的 RPC（例如，发送到目标的 CONT\_TGT\_DESTROY RPC） 服务）。 一些处理程序只涉及查询，一些涉及更新和查询，还有一些涉及所有三种操作； 很少，如果有的话，处理程序只涉及更新而不涉及查询。

A handler must assemble all its updates into a single log entry, commit the log entry, and wait for the log entry to become applicable before applying the updates to the service state. Using a single log entry per update RPC easily makes each update RPC atomic with regard to leader crashes and leadership changes. If RPCs that cannot satisfy this requirement are introduced in the future, additional transaction recovery mechanisms will be required. A leader's service state therefore always represents the effects of all completed update RPCs this leader has handled so far.

处理程序必须将其所有更新组合到一个日志条目中，提交该日志条目，并在将更新应用于服务状态之前等待该日志条目变得适用。 每次更新 RPC 使用单个日志条目可以轻松地使每个更新 RPC 在领导崩溃和领导更改方面具有原子性。 如果将来引入不能满足此要求的 RPC，将需要额外的事务恢复机制。 因此，领导者的服务状态始终代表该领导者迄今为止处理的所有已完成更新 RPC 的影响

Queries, on the other hand, can read directly from the service state, without going through the replicated log. To make sure a request sees the effects of all completed update RPCs handled by all leaders ever elected, however, the handler must ask the Raft module whether there has been any leadership changes. If there has been none, all queries made for this request so far are not stale. If the leader has lost its leadership, the handler aborts the request with an error redirecting the client to the new leader.

另一方面，查询可以直接从服务状态中读取，而无需通过复制的日志。 为确保请求看到所有完整的更新的效果由所有领导者所处理的所有领导者所处理的RPC，但是处理程序必须要求RAFT模块是否有任何领导变化。 如果没有，则到目前为止为此请求所做的所有查询都不是过时的。 如果领导者失去了领导权，处理程序会中止请求，并将客户端重定向到新领导者的错误。

RPCs to other services, if they update state of destination services, must be idempotent. In case of a leadership change, the new leader may send them again, if the client resent the service request in question.

到其他服务的 RPC，如果它们更新目标服务的状态，则必须是幂等的。 在领导发生变化的情况下，如果客户端重新发送有问题的服务请求，新的领导可能会再次发送它们。

Handlers need to cope with reasonable concurrent executions. Conventional local locking on the leader is sufficient to make RPC executions linearizable. Once a leadership change happens, the old leader can no longer perform any updates or leadership verifications with-out noticing the leadership change, which causes all RPCs in execution to abort. The RPCs on the new leader are thus not in conflict with those still left on the old leader. The locks therefore do not need to be replicated as part of the service state.

处理程序需要处理合理的并发执行。 对领导者的传统本地锁定足以使 RPC 执行线性化。 一旦领导层发生变化，旧的领导者就不能再在没有注意到领导层变化的情况下执行任何更新或领导力验证，这会导致所有正在执行的 RPC 中止。 因此，新领导者上的 RPC 与旧领导者上仍然存在的 RPC 不会发生冲突。 因此，不需要将锁作为服务状态的一部分进行复制。

[Replicated service framework (RSVC)](https://github.com/daos-stack/daos/blob/master/src/rsvc/README.md)

Module rsvc implements a generic replicated service framework. This file covers service replication in general, before focusing specifically on module rsvc.

模块 rsvc 实现了一个通用的复制服务框架。 此文件涵盖了一般的服务复制，然后专门关注模块 rsvc。

Introduction/介绍

Certain DAOS RPC services, such as Pool Service (pool\_svc), and Container Service (cont\_svc), are replicated using the state machine approach with Raft. Each of these services tolerates the failure of any minority of its replicas. By spreading its replicas across different fault domains, the service can be highly available. Since this replication approach is self-contained in the sense that it requires only local persistent storage and point to point unreliable messaging, but not any external configuration management service, these services are mandatory for bootstrapping DAOS systems as well as managing the configuration of the lighter-weight I/O replication protocol.

某些 DAOS RPC 服务，例如池服务 (pool\_svc) 和容器服务 (cont\_svc)，使用 Raft 的状态机方法进行复制。 这些服务中的每一个都可以容忍其任何少数副本的故障。 通过将其副本分布在不同的容错域中，该服务可以具有高可用性。 由于这种复制方法是自包含的，因为它只需要本地持久存储和点对点不可靠的消息传递，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lighter-weight I/O 复制协议。

Architecture

An RPC service handles incoming service requests (or just requests) based on its current service state (or just state). To replicate a service is, therefore, to replicate its state so that each request is handled based on the state reached through all prior requests.

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（或仅请求）。 因此，复制服务就是复制其状态，以便根据所有先前请求所达到的状态来处理每个请求。

The state of a service is replicated with a Raft log. The service transforms requests into state queries and deterministic state updates. All state updates are committed to the Raft log first, before being applied to the state. Since Raft guarantees the consistency among the log replicas, the service replicas end up applying the same set of state updates in the same order and go through identical state histories.

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，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。

Raft adopts a strong leadership design, so does each replicated service. A service leader of a term is the Raft leader of the same Raft term. Among the replicas of a service, only the leader of the highest term can handle requests. For the server side, the service code is similar to that of a non-replicated RPC service, except for the handling of leadership change events. For the client side, the service requests must be sent to the current leader, which must be searched for if not known already.

Raft 采用强领导设计，每个复制服务也是如此。 一个term的service leader是同一个Raft term的Raft leader。 在一个服务的副本中，只有最高任期的领导者才能处理请求。 对于服务器端，服务代码类似于非复制 RPC 服务的代码，除了处理领导层更改事件。 对于客户端，必须将服务请求发送给当前的领导者，如果不知道，则必须进行搜索。

A replicated service is implemented using a stack of modules:

复制服务是使用模块堆栈实现的：

背景图案

低可信度描述已自动生成

pool\_svc, and cont\_svc implement the request handlers and the leadership change event handlers of the respective services. They define their respective service state in terms of the RDB data model provided by rdb, implement state queries and updates using RDB transactions, and register their leadership change event handlers into the framework rsvc offers.

pool\_svc 和 cont\_svc 实现了各自服务的请求处理程序和领导层变更事件处理程序。 他们根据 rdb 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，并将他们的领导变化事件处理程序注册到 rsvc 提供的框架中。

rdb (daos\_srv/rdb) implements a hierarchical key-value store data model with transactions, replicated using Raft. It delivers Raft leadership change events to ds\_rsvc, implements transactions using the Raft log, and stores a service's data model and its own internal metadata using the VOS data model. rdb on the leader replica, interfaces with VOS to monitor available persistent storage and, when free space drops below a threshold, rejects new transactions before appending entries to the Raft log that could otherwise result in the service becoming unavailable (due to a mix of successful and "out of space" failures in applying the entries). It also interfaces with VOS to periodically compact storage by triggering aggregation of older versions (epochs) of the log.

rdb (daos\_srv/rdb) 实现了一个带有事务的分层键值存储数据模型，使用 Raft 进行复制。 它向 ds\_rsvc 传递 Raft 领导变化事件，使用 Raft 日志实现事务，并使用 VOS 数据模型存储服务的数据模型和它自己的内部元数据。 领导副本上的 rdb，与 VOS 接口以监控可用的持久存储，并且当可用空间低于阈值时，在将条目附加到 Raft 日志之前拒绝新事务，否则可能导致服务变得不可用（由于成功的混合 和应用条目时“空间不足”失败）。 它还与 VOS 交互，通过触发日志旧版本（epochs）的聚合来定期压缩存储。

raft (rdb/raft/include/raft.h) implements the Raft core protocol in a library. Its integration with VOS and CaRT is done inside rdb via callback functions.

raft (rdb/raft/include/raft.h) 在一个库中实现了 Raft 核心协议。 它与 VOS 和 CarRT 的集成是通过回调函数在 rdb 内部完成的。

A replicated service client (e.g., dc\_pool) uses dc\_rsvc to search for the current service leader:

复制服务客户端（例如 dc\_pool）使用 dc\_rsvc 搜索当前服务领导者：

[ dc\_pool ]

[ dc\_rsvc ]

The search is accomplished with a combination of a client maintained list of candidate service replicas and server RPC error responses in some cases containing a hint where the current leader may be found. A server not running the service responds with an error a client uses to remove that server from its list. A server acting as a non-leader replica responds with a different error, including a hint a client uses to (if necessary) add to its list and alter its search for the leader. And, dc\_rsvc both at client startup and later when the client's list of candidate service replicas may become empty (e.g., due to membership changes in the service), contacts one of the DAOS servers running on a management service node to get an up-to-date list of service replicas for the pool.

在某些情况下，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，其中包含可以找到当前领导者的提示。 未运行该服务的服务器以客户端用于从其列表中删除该服务器的错误进行响应。 充当非领导者副本的服务器以不同的错误响应，包括客户端用来（如有必要）添加到其列表并更改其对领导者的搜索的提示。 并且，dc\_rsvc 在客户端启动时和稍后当客户端的候选服务副本列表可能变空时（例如，由于服务中的成员资格更改），联系在管理服务节点上运行的其中一个 DAOS 服务器以获取最新信息 - 池的服务副本的日期列表。

Module rsvc

The main purpose of rsvc is to avoid code duplication among different replicated service implementations. The callback-intensive API follows from the attempt to extract as much common code as possible, even at the expense of API simplicity. This is a key difference from how other module APIs are designed.

rsvc 的主要目的是避免不同复制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公共代码，即使以 API 简单性为代价。 这是与其他模块 API 的设计方式的关键区别。

rsvc has two parts:

* ds\_rsvc (daos\_srv/rsvc.h): server-side framework.
* dc\_rsvc (daos/rsvc.h): client-side library.

dc\_rsvc is currently still called rsvc. A rename will be done in a future patch.

dc\_rsvc 目前仍称为 rsvc。 重命名将在未来的补丁中完成。

###### DAOS Services

The diagram below shows the internal layering of the DAOS services and interactions with the different libraries mentioned above.

下图显示了 DAOS 服务的内部分层以及与上述不同库的交互。

图片包含 图形用户界面

描述已自动生成

Vertical boxes represent DAOS services whereas horizontal ones are for infrastructure libraries.

垂直框代表 DAOS 服务，而水平框代表基础设施库。

For further reading on the internals of each service:

要进一步阅读每个服务的内部结构：

[Pool service](https://github.com/daos-stack/daos/blob/master/src/pool/README.md)

A pool is a set of targets spread across different storage nodes over which data and metadata are distributed to achieve horizontal scalability, and replicated or erasure-coded to ensure durability and availability (see: [Storage Model: DAOS Pool](https://github.com/daos-stack/daos/blob/master/doc/overview/storage.md#daos-pool)).

池是分布在不同存储节点上的一组targets，数据和元数据分布在这些节点上以实现水平可扩展性，并进行复制或擦除编码以确保持久性和可用性（请参阅：存储模型：DAOS 池）。

Pool Service

The Pool Service (pool\_svc) stores the metadata for pools, and provides an API to query and update the pool configuration. Pool metadata are organized as a hierarchy of key-value stores (KVS) that is replicated over a number of servers backed by Raft consensus protocol which uses strong leadership; client requests can only be serviced by the service leader while non-leader replicas merely respond with a hint pointing to the current leader for the client to retry. pool\_svc derives from a generic replicated service module rsvc (see: [Replicated Services: Architecture](https://github.com/daos-stack/daos/blob/master/src/rsvc/README.md#architecture)) whose implementation facilitates the client search for the current leader.

池服务 (pool\_svc) 存储池的元数据，并提供 API 来查询和更新池配置。 池元数据被组织为键值存储 (KVS) 的层次结构，在多个服务器上复制，由使用强大领导力的 Raft 共识协议支持； 客户端请求只能由服务领导者提供服务，而非领导者副本仅响应指向当前领导者的提示，以便客户端重试。 pool\_svc 派生自通用复制服务模块 rsvc（请参阅：复制服务：体系结构），其实现有助于客户端搜索当前领导者。

**Metadata Layout**

图示

描述已自动生成

The top-level KVS stores the pool map, security attributes such as the UID, GID and mode, information related to space management and self-healing (see: [Rebuild](https://github.com/daos-stack/daos/blob/master/src/rebuild/README.md)) as well as a second-level KVS containing user-defined attributes (see: [Container Service: Metadata Layout](https://github.com/daos-stack/daos/blob/master/src/container/README.md#metadata-layout)). In addition, it also stores information on pool connections, represented by a pool handle and identified by a client-generated handle UUID. The terms "pool connection" and "pool handle" may be used interchangeably.

顶层 KVS 存储池映射、UID、GID 和模式等安全属性、与空间管理和自愈相关的信息（参见：重建）以及包含用户定义属性的二级 KVS（参见 ：容器服务：元数据布局）。 此外，它还存储有关池连接的信息，由池句柄表示并由客户端生成的句柄 UUID 标识。 术语“池连接”和“池句柄”可以互换使用。

Pool Operations

Pool / Pool Service Creation

Pool creation is driven entirely by the Management Service since it requires special privileges for steps related to allocation of storage and querying of fault domains. After formatting all the targets, the Target component calls ds\_pool\_create of the Pool Module on each target, which simply generates a new UUID for the current target and stores it in the DSM\_META\_FILE. At this point the management module passes the control to the pool module by calling theds\_pool\_svc\_create, which initializes service replication on the selected subset of nodes for the combined Pool and Container Service. The Pool module now sends a POOL\_CREATE request to the service leader which creates the service database; the list of targets and their fault domains are then converted into the initial version of the pool map and stored in the pool service, along with other initial pool metadata.

池创建完全由管理服务驱动，因为它需要与存储分配和故障域查询相关的步骤的特殊权限。 格式化所有targets后，Target 组件调用每个targets上的 Pool Module 的 ds\_pool\_create，它只是为当前targets生成一个新的 UUID 并将其存储在 DSM\_META\_FILE 中。 此时，管理模块通过调用 ds\_pool\_svc\_create 将控制权传递给池模块，该模块会在选定的节点子集上为池和容器服务组合初始化服务复制。 Pool 模块现在向创建服务数据库的服务领导发送 POOL\_CREATE 请求； 然后将目标列表及其故障域转换为池映射的初始版本，并与其他初始池元数据一起存储在池服务中。

Pool Connection

To establish a pool connection, a client process calls the daos\_pool\_connect method in the client library with the pool UUID, connection information (such as group name and list of service ranks) and connection flags; this initiates a POOL\_CONNECT request to the Pool Service. The Pool Service tries to authenticate the request according to the security model in use (e.g., UID/GID in a POSIX-like model), and to authorize the requested capabilities to the client-generated pool handle UUID. Before proceeding, the pool map is transferred to the client; if there are errors from this point onwards, the server can simply ask the client to discard the pool map.

为了建立池连接，客户端进程调用客户端库中的 daos\_pool\_connect 方法，使用池 UUID、连接信息（如组名和服务等级列表）和连接标志； 这会向池服务发起 POOL\_CONNECT 请求。 池服务尝试根据使用的安全模型（例如，类似 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池句柄 UUID。 在继续之前，池映射被传输到客户端； 如果从这点开始出现错误，服务器可以简单地要求客户端丢弃池映射。

At this point, the Pool Service checks for existing pool handles:

* If a pool handle with the same UUID already exists, a pool connection has already been established and nothing else needs to be done.
* If another pool handle exists such that either the currently requested or the existing one has exclusive access, the connection request is rejected with a busy status code.

此时，池服务会检查现有池句柄：

• 如果已存在具有相同 UUID 的池句柄，则池连接已建立，无需执行任何其他操作。

• 如果存在另一个池句柄，使得当前请求的或现有的池句柄具有独占访问权限，则连接请求将被拒绝并显示忙状态代码。

If everything goes well, the pool service sends a collective POOL\_TGT\_CONNECT request to all targets in the pool with the pool handle UUID. The Target Service creates and caches the local pool objects and opens the local VOS pool for access.

如果一切顺利，池服务使用池句柄 UUID 向池中的所有targets发送集体 POOL\_TGT\_CONNECT 请求。 Target Service 创建并缓存本地池对象，并打开本地 VOS 池进行访问。

A group of peer application processes may share a single pool connection handle (see: [Storage Model: DAOS Pool](https://github.com/daos-stack/daos/blob/master/doc/overview/storage.md#daos-pool) and [Use Cases: Storage Management and Workflow Integration](https://github.com/daos-stack/daos/blob/master/doc/overview/use_cases.md#storage-management--workflow-integration)).

一组对等应用程序进程可能共享一个池连接句柄（请参阅：存储模型：DAOS 池和用例：存储管理和工作流集成）。

To close a pool connection, a client process calls the daos\_pool\_disconnect method in the client library with the pool handle, triggering a POOL\_DISCONNECT request to the Pool Service, which sends a collective POOL\_TGT\_DISCONNECT request to all targets in the pool. These steps destroy all state associated with the connection, including all container handles. Other client processes sharing this connection should destroy their copies of the pool handle locally, preferably before the disconnect method is called on behalf of everyone. If a group of client processes terminate prematurely, before having a chance to call the pool disconnect method, their pool connection will eventually be evicted once the pool service learns about the event from the run-time environment.

要关闭池连接，客户端进程使用池句柄调用客户端库中的 daos\_pool\_disconnect 方法，触发对池服务的 POOL\_DISCONNECT 请求，池服务向池中的所有目标发送集体 POOL\_TGT\_DISCONNECT 请求。 这些步骤会破坏与连接关联的所有状态，包括所有容器句柄。 共享此连接的其他客户端进程应该在本地销毁它们的池句柄副本，最好在代表每个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池断开连接方法之前提前终止，一旦池服务从运行时环境中获悉该事件，它们的池连接最终将被逐出。

[Container service](https://github.com/daos-stack/daos/blob/master/src/container/README.md)

A container represents an object address space inside a pool and is identified by a UUID. To access a container, an application must first connect to the pool and then create or open the container. If the application is authorized to access the container, it obtains a container handle. This includes capabilities that authorize any process in the application to access the container and its contents. The opening process may share this handle with any or all of its peers. Their capabilities are revoked on closing the container.

容器表示池内的对象地址空间，并由 UUID 标识。 要访问容器，应用程序必须首先连接到池，然后创建或打开容器。 如果应用程序被授权访问容器，它会获得一个容器句柄。 这包括授权应用程序中的任何进程访问容器及其内容的功能。 打开过程可以与其任何或所有对等体共享该句柄。 他们的能力在关闭容器时被撤销。

Metadata Layout

The Container Service (cont\_svc) stores the metadata for containers and provides an API to query and update the state as well as for managing the life-cycle of a container. Container metadata are organized as a hierarchy of key-value stores (KVS) that is replicated over a number of servers backed by Raft consensus protocol which uses strong leadership; client requests can only be serviced by the service leader while non-leader replicas merely respond with a hint pointing to the current leader for the client to retry. cont\_svc derives from a generic replicated service module rsvc (see: [Replicated Services: Architecture](https://github.com/daos-stack/daos/blob/master/src/rsvc/README.md#architecture)) whose implementation facilitates the client search for the current leader.

容器服务 (cont\_svc) 存储容器的元数据，并提供 API 来查询和更新状态以及管理容器的生命周期。 容器元数据被组织为键值存储 (KVS) 的层次结构，在多个服务器上复制，由使用强大领导力的 Raft 共识协议支持； 客户端请求只能由服务领导者提供服务，而非领导者副本仅响应指向当前领导者的提示，以便客户端重试。 cont\_svc 派生自通用复制服务模块 rsvc（参见：复制服务：架构），其实现有助于客户端搜 索当前领导者。

图示

描述已自动生成

The top-level KVS root has two children:

1. **Containers KVS:** Holds a list of Container Properties KVSs indexed by UUID of the Container which is supplied by the user at the time of creating a new container.
2. **Container Handles KVS:** Used for storing data about container handles opened by various applications and indexed by a handle UUID which is generated by the client at the time of opening a container. The metadata associated with a container handle include its capabilities (e.g., read-only or read-write) and its per-handle epoch state. When a container is closed, the corresponding entry is removed from this store.

顶级 KVS 根有两个孩子：

1. Containers KVS：保存由用户在创建新容器时提供的由容器的 UUID 索引的容器属性 KVS 列表。

2. Container Handles KVS：用于存储各种应用程序打开的容器句柄的数据，并由客户端在打开容器时生成的句柄UUID索引。 与容器句柄关联的元数据包括其功能（例如，只读或读写）及其每个句柄的纪元状态。 当一个容器关闭时，相应的条目就会从这个存储中删除。

The container properties KVS is used to store per-container metadata that consists of many mutable and immutable scalar valued properties as well as other KVSs as shown in the figure above.

容器属性 KVS 用于存储每个容器的元数据，该元数据由许多可变和不可变标量值属性以及其他 KVS 组成，如上图所示。

Users can create, delete and retrieve a list of persistent snapshots, which are essentially epochs that will not be aggregated away. A snapshot remains readable until it is explicitly destroyed. A container can also be rolled back to a particular snapshot. (see: [Storage Model: DAOS Container](https://github.com/daos-stack/daos/blob/master/doc/overview/storage.md#daos-container) and [Transaction Model: Container Snapshot](https://github.com/daos-stack/daos/blob/master/doc/overview/transaction.md#container-snapshot)).

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聚合的epochs。 快照在被显式销毁之前保持可读。 容器也可以回滚到特定的快照。 （请参阅：存储模型：DAOS 容器和事务模型：容器快照）。

Users can also define custom attributes for containers which are essentially name-value pairs; with the name being a null-terminated string while the value is an arbitrary sequence of bytes. The Container Service allows clients to retrieve and update multiple attributes at a time as well as to list names of stored attributes.

用户还可以为本质上是名称-值对的容器定义自定义属性； 名称是一个以空字符结尾的字符串，而值是一个任意的字节序列。 容器服务允许客户端一次检索和更新多个属性以及列出存储属性的名称。

Target Service (TO BE UPDATED)

The Target Service maps the global object address space of a DAOS container onto the local object address space of a VOS container within the target's VOS pool (vpool), and and calls VOS methods on behalf of the Container Service (see: [VOS Concepts](https://github.com/daos-stack/daos/blob/master/src/vos/README.md#71)). It caches per-thread information on container objects and open handles in volatile memory for ready access.

Target Service 将 DAOS 容器的全局对象地址空间映射到target VOS 池（vpool）内的 VOS 容器的本地对象地址空间，并代表容器服务调用 VOS 方法（参见：VOS 概念）。 它缓存容器对象上的每线程信息，并在易失性内存中打开句柄以供随时访问。

Target Faults

Given hundreds of thousands of targets, the epoch protocol must allow progress in the presence of target faults. Since pool and container services are highly available, the problem is mainly concerned with target services. The solution is based on the assumption that losing some targets may not necessarily cause any application data loss, as there may be enough redundancy created by the DAOS-SR layer to hide the faults from applications. Moreover, an application might even want to ignore a particular data loss (which the DAOS-SR layer is unable to hide), for it has enough application-level redundancy to cope or it simply does not care.

给定数十万个目标，epoch协议必须允许在存在targets故障的情况下取得进展。 由于池和容器服务是高可用的，所以问题主要与targets服务有关。 该解决方案基于这样一个假设，即丢失某些targets不一定会导致任何应用程序数据丢失，因为 DAOS-SR 层可能会创建足够的冗余来隐藏应用程序的故障。 此外，应用程序甚至可能想要忽略特定的数据丢失（DAOS-SR 层无法隐藏），因为它有足够的应用程序级冗余来应对或者根本不在乎。

When a write, flush, or discard operation fails, the DAOS-SR layer calculates if there is sufficient redundancy left to continue with the epoch. If the failure can be hidden, and assuming that the target in question has not already been disabled in the pool map (e.g., as a result of a RAS notification), the DAOS-SR layer must disable the target before committing the epoch. For the epoch protocol, the resulting pool map update effectively records the fact that the target may store an undefined set of write operations in the epoch, and should be avoided. This also applies to applications that would like to ignore similar failures which the DAOS-SR layer cannot hide.

当写入、刷新或丢弃操作失败时，DAOS-SR 层会计算是否有足够的冗余剩余来继续 epoch。 如果故障可以隐藏，并假设有问题的target尚未在池映射中禁用（例如，作为 RAS 通知的结果），DAOS-SR 层必须在提交 epoch 之前禁用target。 对于epoch协议，由此产生的池映射更新有效地记录了这样一个事实，即target可能在epoch中存储一组未定义的写操作，应该避免。 这也适用于希望忽略 DAOS-SR 层无法隐藏的类似故障的应用程序。

Object ID Allocator

The OID allocator is a helper routine service that allows users to allocate a unique set of 64 bit unsigned integers within a container. This is helpful for applications or middleware that do not have a way to easily allocate a unique DAOS object ID in a scalable manner. The largest allocated ID is tracked in the Container Properties KVS for future access to that container. This service does not guarantee that the IDs allocated are sequential and several ID ranges may be discarded at container close.

OID 分配器是一个辅助例程服务，它允许用户在容器内分配一组唯一的 64 位无符号整数。 这对于无法以可扩展方式轻松分配唯一 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。 此服务不保证分配的 ID 是连续的，并且在容器关闭时可能会丢弃多个 ID 范围

The allocator is implemented using an Incast Variable on the server side that tracks the highest used object ID on a container on the root of the IV tree. A client may request a new allocation from any server running the Container Target Service, i.e. any node in the IV tree. When a new request arrives, the server first checks whether there are any allocated IDs available locally. If not, it forwards a request to the parent (asking for a bigger range of OIDs in that case). The parent does the same check and keeps forwarding to its parent until a request is satisfied or we reach the IV root, which updates the incast variable for the max OID allocated in the container metadata. At each tree level, the number of OIDs asked for is increased to be able to satisfy future OID allocation requests faster.

分配器是使用服务器端的 Incast 变量实现的，该变量跟踪 IV 树根上容器上使用的最高对象 ID。 客户端可以向任何运行容器target服务的服务器请求新的分配，即 IV 树中的任何节点。 当新请求到达时，服务器首先检查本地是否有任何可用的分配 ID。 如果没有，它将请求转发给父级（在这种情况下要求更大范围的 OID）。 父节点执行相同的检查并继续转发给它的父节点，直到满足请求或我们到达 IV 根，这将更新容器元数据中分配的最大 OID 的 incast 变量。 在每个树级别，请求的 OID 数量都会增加，以便能够更快地满足未来的 OID 分配请求。

Container Operations

A client creates a new container by sending a CONT\_CREATE request to the Container Service with the pool handle and a UUID. The client must first establish a pool connection to obtain a pool handle. Optionally, the request can also contain a list of properties to be set on the newly created container. In response, the Container Service creates the corresponding Container Properties KVS with the UUID as the key. Creating a container does not require involvement of the Target Service.

客户端通过向容器服务发送一个带有池句柄和 UUID 的 CONT\_CREATE 请求来创建一个新容器。 客户端必须首先建立池连接以获取池句柄。 可选地，请求还可以包含要在新创建的容器上设置的属性列表。 作为响应，容器服务以 UUID 为键创建对应的容器属性 KVS。 创建容器不需要target服务的参与。

Clients may now open a container by supplying the open pool handle and the container UUID along with flags (e.g., read-only or read-write). The client library sends a CONT\_OPEN request with a locally generated UUID to the Container Service, then use IV(Incast Variable) to broadcast handle asynchronously to all enabled targets in the pool. On successful completion it creates a new entry in the Container Handles KVS.

客户端现在可以通过提供开放池句柄和容器 UUID 以及标志（例如，只读或读写）来打开容器。 客户端库向容器服务发送一个带有本地生成的 UUID 的 CONT\_OPEN 请求，然后使用 IV(Incast Variable) 向池中所有启用的目标异步广播句柄。 成功完成后，它会在 Container Handles KVS 中创建一个新条目。

A client can close a container handle that is no longer needed by sending a CONT\_CLOSE request to the Container Service which it broadcasts to all enabled targets as a collective CONT\_TGT\_CLOSE in order to close the container handle. It then deletes the corresponding entry from the Container Handles KVS and discards updates performed on the handle that were not committed.

客户端可以通过向容器服务发送 CONT\_CLOSE 请求来关闭不再需要的容器句柄，容器服务将其作为集合 CONT\_TGT\_CLOSE 广播到所有启用的目标，以关闭容器句柄。 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。

A container is destroyed when the client sends a CONT\_DESTROY request to the Container Service causing it to purge all metadata. Similarly, the targets collectively receive a CONT\_TGT\_DESTROY request from the Container Service and drop all data associated with that container including all the objects within that container. The client can optionally destroy a container forcibly in case it has handles that are currently open.

当客户端向容器服务发送 CONT\_DESTROY 请求导致其清除所有元数据时，容器将被销毁。 类似地，目标共同接收来自容器服务的 CONT\_TGT\_DESTROY 请求，并删除与该容器关联的所有数据，包括该容器内的所有对象。 如果容器具有当前打开的句柄，则客户端可以选择强行销毁容器。

Epoch Protocol

The epoch protocol implements the epoch model described in the [Transaction Model](https://github.com/daos-stack/daos/blob/master/doc/overview/transaction.md). The Container service manages the epochs of a container; it maintains the definitive epoch state as part of the container metadata, whereas the target services have little knowledge of the global epoch state. Epoch commit, discard, and aggregate procedures are therefore all driven by the container service.

epoch协议实现了事务模型中描述的epoch模型。 容器服务管理容器的epoch； 它将确定的epoch状态作为容器元数据的一部分来维护，而target服务对全局epoch状态知之甚少。 因此，Epoch 提交、丢弃和聚合过程都是由容器服务驱动的。

On each target, the target service eagerly stores incoming write operations into the matching VOS container. If a container handle discards an epoch, VOS helps discard all write operations associated with that container handle. When a write operation succeeds, it is immediately visible to conflicting operations in equal or higher epochs. A conflicting write operation with the same epoch will be rejected by VOS unless it is associated with the same container handle and has the same content as the one that is already executed.

在每个target上，target服务急切地将传入的写入操作存储到匹配的 VOS 容器中。 如果容器句柄丢弃了一个epoch，VOS 会帮助丢弃与该容器句柄关联的所有写操作。 当写操作成功时，它对相等或更高时期内的冲突操作立即可见。 具有相同epoch的冲突写操作将被 VOS 拒绝，除非它与相同的容器句柄相关联并且与已执行的内容具有相同的内容。

Before committing an epoch, an application must ensure that a sufficient set of write operations for this epoch have been persisted by the target services. The application may decide that losing some write operations is acceptable, depending on the redundancy scheme each of them employs. Committing an epoch of a container handle results in a CONT\_EPOCH\_COMMIT request to the corresponding container service, which simply updates the metadata. When the update becomes persistent, the container service replies to the client with the new epoch state.

在提交一个epoch之前，应用程序必须确保target服务已经为这个epoch保留了一组足够的写操作。 应用程序可能会决定丢失一些写操作是可以接受的，这取决于它们每个人采用的冗余方案。 提交容器句柄的 epoch 会导致对相应容器服务的 CONT\_EPOCH\_COMMIT 请求，该请求只是更新元数据。 当更新变得持久时，容器服务用新的epoch状态回复客户端

[Key-array object service](https://github.com/daos-stack/daos/blob/master/src/object/README.md)

Object

DAOS object stores user's data, it is identified by object ID which is unique within the DAOS container it belongs to. Objects can be distributed across any target of the pool for both performance and resilience. DAOS object in DAOS storage model is shown in the diagram -The object module implements the object I/O stack.

DAOS 对象存储用户的数据，它由对象 ID 标识，对象 ID 在其所属的 DAOS 容器中是唯一的。 为了性能和弹性，对象可以分布在池的任何target上。 DAOS 存储模型中的 DAOS 对象如图所示 - 对象模块实现了对象 I/O 堆栈。

手机屏幕的截图

描述已自动生成

KV store, dkey and akey

Each DAOS object is a Key-Value store with locality feature. The key is split into a **dkey** (distribution key) and an **akey** (attribute key). All entries with the same dkey are guaranteed to be collocated on the same targets. Enumeration of the akeys of a dkey is provided.

The value can be either atomic **single value** (i.e. value replaced on update) or a **byte array** (i.e. arbitrary extent fetch/update).

每个 DAOS 对象都是一个具有局部性特征的键值存储。 key分为dkey（分发密钥）和akey（属性密钥）。 保证所有具有相同 dkey 的条目都配置在相同的target上。 提供了 dkey 的 akeys 的枚举。 该值可以是原子单个值（即更新时替换的值）或字节数组（即任意范围获取/更新）。

Object Class

To avoid scaling problems and overhead common to traditional storage stack, DAOS objects are intentionally very simple. No default object metadata beyond the object class is provided. This means that the system does not maintain time, size, owner, permissions and opener tracking attributes.

为了避免传统存储堆栈常见的扩展问题和开销，DAOS 对象有意地非常简单。 不提供超出对象类的默认对象元数据。 这意味着系统不维护时间、大小、所有者、权限和开启者跟踪属性。

The DAOS **object class** describes the definitions for object types, data protection methods, and data distribution strategies. An **object class** has a unique class ID, which is a 16-bit value, and can represent a category of objects that use the same schema(data protection, distribution). DAOS provides some pre-defined object class for the most common use (see daos\_obj\_classes). In addition user can register customized object class by daos\_obj\_register\_class() (not implemented yet). A successfully registered object class is stored as container metadata; it is valid in the lifetime of the container.

DAOS 对象类描述了对象类型、数据保护方法和数据分发策略的定义。 一个对象类有一个唯一的类 ID，它是一个 16 位的值，可以代表使用相同模式（数据保护、分发）的一类对象。 DAOS 为最常见的用途提供了一些预定义的对象类（参见 daos\_obj\_classes）。 此外，用户可以通过 daos\_obj\_register\_class()（尚未实现）注册自定义对象类。 成功注册的对象类存储为容器元数据； 它在容器的生命周期内有效。

The object class ID is embedded in object ID. By daos\_obj\_generate\_id() user can generate an object ID for the specific object class ID. DAOS uses this class ID to find the corresponding object class, and then distribute and protect object data based on algorithm descriptions of this class.

对象类 ID 嵌入在对象 ID 中。 通过 daos\_obj\_generate\_id() 用户可以为特定的对象类 ID 生成对象 ID。 DAOS 使用这个类 ID 找到对应的对象类，然后根据这个类的算法描述来分发和保护对象数据。

Data Protection Method

Two types data protection methods supported by DAOS - replication and erasure coding. In addition, checksums can be used with both methods to ensure end-to-end data integrity. If checksums discovers silent data corruption, the data protection method (replication or erasure codes) might be able to recover the data.

DAOS 支持两种类型的数据保护方法——复制和纠删码。 此外，两种方法都可以使用校验和来确保端到端数据的完整性。 如果校验和发现静默数据损坏，数据保护方法（复制或擦除代码）可能能够恢复数据

Replication

Replication ensures high availability of object data because objects are accessible while any replica exists. Replication can also increase read bandwidth by allowing concurrent reads from different replicas.

复制确保对象数据的高可用性，因为对象在任何副本存在时都可以访问。 复制还可以通过允许来自不同副本的并发读取来增加读取带宽。

**Server-side Replication**

DAOS supports server replication, which has stronger consistency of replicas with a trade-off in performance and latency. In server replication mode DAOS client selects a leader shard to send the IO request with the need-to- forward shards embedded in the RPC request, when the leader shard gets that IO request it handles it as below steps:

* firstly forwards the IO request to others shards For the request forwarding, it is offload to the vos target's offload xstream to release the main IO service xstream from IO request sending and reply receiving (see shard\_req\_forward).
* then serves the IO request locally
* waits the forwarded IO's completion and reply client IO request.

DAOS 支持服务器复制，具有更强的副本一致性，并在性能和延迟上进行权衡。 在服务器复制模式下，DAOS 客户端选择一个leader分片发送 IO 请求，并在 RPC 请求中嵌入需要转发的分片，当领导分片收到该 IO 请求时，它按以下步骤处理：

• 首先将IO 请求转发到其他shards 请求转发是卸载到vos target的offload xstream 以释放IO 请求发送和回复接收的主IO 服务xstream（参见shard\_req\_forward）。

• 然后在本地处理 IO 请求 • 等待转发的 IO 完成并回复客户端 IO 请求。

In server replication mode, the DAOS client-side IO error handling is relative simpler because all operations only sent to only one server shard target, so need not to compare replied pool map version from multiple shard targets, other error handing is same as client replication mode described above.

在服务器复制模式下，DAOS客户端IO错误处理相对简单，因为所有操作只发送到一个服务器分片目标，所以不需要比较来自多个分片目标的回复池映射版本，其他错误处理与客户端复制相同 模式如上所述。

In this mode the conflict writes can be detected and serialized by the leader shard server. Now both modes are supported by DAOS, it can be dynamically configured by environment variable DAOS\_IO\_SRV\_DISPATCH before loading DAOS server. By default DAOS works in server replication mode, and if the ENV set as zero then will work in client replication mode.

在这种模式下，Leader 分片服务器可以检测并序列化冲突写入。 现在DAOS都支持这两种模式，可以在加载DAOS服务器之前通过环境变量DAOS\_IO\_SRV\_DISPATCH动态配置。 默认情况下，DAOS 在服务器复制模式下工作，如果 ENV 设置为零，则将在客户端复制模式下工作。

**Client-side Replication**

Client replication is the mode that it is synchronous and fully in the client stack, to provide high concurrency and low latency I/O for the upper layer. **This mode is not default and is only provided for testing purposes, consistency between replicas of this mode is not guaranteed when failure occurs**.

* I/O requests against replicas are directly issued via DAOS client; there is no sequential guarantee on writes in the same epoch, and concurrent writes for a same object can arrive at different replicas in an arbitrary order.
* Because there is no communication between servers in this way, there is no consistent guarantee if there are overlapped writes or KV updates in the same epoch. The DAOS server should detect overlapped updates in the same epoch, and return errors or warnings for the updates to the client. The only exception is multiple updates to the same extent or KV having the exactly same data. In this case, it is allowed because these updates could potentially be the resending requests.

客户端复制是同步且完全在客户端堆栈中的模式，为上层提供高并发和低延迟的I/O。此模式不是默认的，仅用于测试目的，当发生故障时，无法保证此模式的副本之间的一致性。

• 针对副本的 I/O 请求通过 DAOS 客户端直接发出；同一 epoch 中的写入没有顺序保证，同一对象的并发写入可以以任意顺序到达不同的副本。

• 由于这种方式服务器之间没有通信，因此在同一epoch 中是否有重叠写入或KV 更新无法保证一致。 DAOS 服务器应检测同一时期内的重叠更新，并将更新的错误或警告返回给客户端。唯一的例外是对相同范围的多次更新或具有完全相同数据的 KV。在这种情况下，这是允许的，因为这些更新可能是重新发送的请求。

Erasure Code

In the case of replicating a whole object, the storage overhead would be 100% for each replica. This is unaffordable in some cases, so DAOS also provides erasure code as another option of data protection, with better storage efficiency.

Erasure codes may be used to improve resilience, with lower space overhead. This feature is still working in progress.

在复制整个对象的情况下，每个副本的存储开销将为 100%。 这在某些情况下是无法承受的，因此 DAOS 还提供纠删码作为另一种数据保护选项，具有更好的存储效率。 纠删码可用于提高弹性，同时具有较低的空间开销。 此功能仍在进行中。

Checksum

The checksum feature attempts to provide end-to-end data integrity. On an update, the DAOS client calculates checksums for user data and sends with the RPC to the DAOS server. The DAOS server returns the checksum with the data on a fetch so the DAOS client can verify the integrity of the data. See [End-to-end Data Integrity Overiew](https://github.com/daos-stack/daos/blob/master/doc/overview/data_integrity.md) for more information.

校验和功能试图提供端到端的数据完整性。 更新时，DAOS 客户端计算用户数据的校验和，并通过 RPC 发送到 DAOS 服务器。 DAOS 服务器返回校验和以及获取的数据，以便 DAOS 客户端可以验证数据的完整性。 有关详细信息，请参阅端到端数据完整性概述。

Checksums are configured at the container level and when a client opens a container, the checksum properties will be queried automatically, and, if enabled, both the server and client will init and hold a reference to a [daos\_csummer](https://github.com/daos-stack/daos/blob/master/src/object/src/common/README.md) in ds\_cont\_hdl and dc\_cont respectively.

校验和在容器级别配置，当客户端打开容器时，将自动查询校验和属性，如果启用，服务器和客户端将分别在 ds\_cont\_hdl 和 dc\_cont 中初始化并保存对 daos\_csummer 的引用。

For Array Value Types, the DAOS server might need to calculate new checksums for requested extents. After extents are fetched by the server object layer, the checksums srv\_csum

对于数组值类型，DAOS 服务器可能需要为请求的范围计算新的校验和。 在服务器对象层获取范围后，校验和 srv\_csum

**Object Update**

On an object update (dc\_obj\_update) the client will calculate checksums using the data in the sgl as described by an iod (daos\_csummer\_calc\_iod). Memory will be allocated for the checksums and the iod checksum structures that represent the checksums (dcs\_iod\_csums). The checksums will be sent to the server as part of the IOD and the server will store in [VOS] (src/vos/README.md).

在对象更新 (dc\_obj\_update) 时，客户端将使用 sql 中的数据计算校验和，如 iod (daos\_csummer\_calc\_iod) 所述。 将为校验和和表示校验和的 iod 校验和结构 (dcs\_iod\_csums) 分配内存。 校验和将作为 IOD 的一部分发送到服务器，服务器将存储在 [VOS] (src/vos/README.md) 中。

**Object Fetch - Server**

On handling an object fetch (ds\_obj\_rw\_handler), the server will allocate memory for the checksums and iod checksum structures. Then during the vos\_fetch\_begin stage, the checksums will be fetched from [VOS](https://github.com/daos-stack/daos/blob/master/src/object/src/vos/README.md). For Array Value Types, the extents fetched will need to be compared to the requested extent and new checksums might need to be calculated. ds\_csum\_add2iod will look at the fetched bio\_sglist and the iod request to determine if the stored checksums for the request are sufficient or if new ones need to be calculated.

在处理对象获取 (ds\_obj\_rw\_handler) 时，服务器将为校验和和 iod 校验和结构分配内存。 然后在 vos\_fetch\_begin 阶段，将从 VOS 获取校验和。 对于数组值类型，需要将获取的范围与请求的范围进行比较，并且可能需要计算新的校验和。 ds\_csum\_add2iod 将查看获取的 bio\_sglist 和 iod 请求，以确定为请求存储的校验和是否足够或者是否需要计算新的校验和。

**cc\_need\_new\_csum Logic**

The following are some examples of when checksums are copied and when new checksums are needed. There are more examples in the unit tests for this logic( ./src/object/tests/srv\_checksum\_tests.c)

以下是一些何时复制校验和以及何时需要新校验和的示例。 此逻辑的单元测试中有更多示例（./src/object/tests/srv\_checksum\_tests.c）

Request |----|----|----|----|

Extent 2 |----|----|

Extent 1 |----|----|

Chunk length is 4. Extent 1 is bytes 0-7, extent 2 is bytes 8-15. Request is bytes 0-15. There is no overlap of extents and each extent is completely requested. Therefore, the checksum for each chunk of each extent is copied.

块长度为 4。区 1 是字节 0-7，区 2 是字节 8-15。 请求是字节 0-15。 范围没有重叠，并且每个范围都是完全请求的。 因此，复制每个范围的每个块的校验和。

Request |----|----|----

Extent 2 | |----|----

Extent 1 |----|----|

Chunk length is 4. Extent 1 is bytes 0-7. Extent 2 is bytes 8-11. Request is bytes 0-1. Even though there is overlap, the extents are aligned to chunks, therefore each chunk's checksum is copied. The checksum for the first chunk will come from extent 1, the second and third checksums come from extent 2, just like the data does.

块长度为 4。区 1 为字节 0-7。 数据区 2 是字节 8-11。 请求是字节 0-1。 即使存在重叠，范围也与块对齐，因此每个块的校验和都会被复制。 第一个块的校验和将来自范围 1，第二个和第三个校验和来自范围 2，就像数据一样。

Request | ---- |

Extent 1 |--------|

Chunk length is 8. Extent 1 is bytes 0-7. Request is bytes 2-5. Because the request is only part of the stored extent, a new checksum will need to be created

块长度为 8。区 1 为字节 0-7。 请求是字节 2-5。 由于请求只是存储范围的一部分，因此需要创建新的校验和

Request |--------|--------|

Extent 2 | -----|--------|

Extent 1 |------ | |

Chunk length is 8. Extent 1 is bytes 0-5. Extent 2 is bytes 3-15. Request is bytes 0-15. The first chunk needs a new checksum because it will be made up of data from extent 1 and extent 2. The checksum for the second chunk is copied.

块长度为 8。区 1 为字节 0-5。 数据区 2 是字节 3-15。 请求是字节 0-15。 第一个块需要一个新的校验和，因为它将由来自范围 1 和范围 2 的数据组成。第二个块的校验和被复制

Note: Anytime the server calculates a new checksum; it will use the stored checksum to verify the original chunks.

注意：任何时候服务器计算新的校验和； 它将使用存储的校验和来验证原始块。

**Object Fetch - Client**

In the client RPC callback, the client will calculate checksums for the data fetched and compare to the checksums fetched (daos\_csummer\_verify).

在客户端 RPC 回调中，客户端将为获取的数据计算校验和，并与获取的校验和 (daos\_csummer\_verify) 进行比较。

Object Sharding

DAOS supports different data distribution strategies.

DAOS 支持不同的数据分布策略。

Single (unstriped) Object

For replication, single (unstriped) object always has one stripe and each shard of it is a full replica, for Erasure code, single object only has one parity group and a shard of it can either be a data chunk or parity chunk of the parity group. A single (unstriped) object can be either a byte-array or a KV.

对于复制，单个（非条带化）对象总是有一个条带，它的每个分片都是一个完整的副本，对于纠删码，单个对象只有一个奇偶校验组，它的一个分片可以是奇偶校验的数据块或奇偶校验块 团体。 单个（非条带化）对象可以是字节数组或 KV。

Fixed Stripe Object

A fixed stripe object has a constant number of stripes and each stripe has a fixed stripe size. These stripe attributes are predefined by object class, DAOS uses these attributes to compute object layout.

固定条带对象具有恒定数量的条带，并且每个条带具有固定的条带大小。 这些条带属性由对象类预定义，DAOS 使用这些属性来计算对象布局。

Dynamically Striped Object (Not implemented)

A fixed stripe object always has the same number of stripes since it was created. In contrast, a dynamically stripped object could be created with a single stripe. It will increase its stripe count as its size grows to some boundary, to achieve more storage space and better concurrent I/O performance.

Now the dynamically Striped Object is not implemented yet.

固定条带对象自创建以来始终具有相同数量的条带。 相比之下，可以使用单个条带创建动态剥离的对象。 当它的大小增长到某个边界时，它会增加它的条带数量，以获得更多的存储空间和更好的并发 I/O 性能。 现在还没有实现动态条纹对象。

[Self-healing (aka rebuild)](https://github.com/daos-stack/daos/blob/master/src/rebuild/README.md)

In DAOS, if the data is replicated with multiple copies on different targets, once one of the targets fails, it's data will be rebuilt on the other targets automatically, so the data redundancy will not be impacted due to the target failure. In future versions, DAOS will also support Erasure Coding to protect the data; then the rebuild process might be updated accordingly.

在 DAOS 中，如果数据在不同的target上进行多份复制，一旦其中一个target发生故障，它的数据将自动重建到其他target上，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 Erasure Coding 来保护数据； 那么重建过程可能会相应地更新。

Rebuild Detection

When a target failed, it should be detected promptly and notify the pool (Raft) leader, and then the leader will exclude the target from the pool and trigger the rebuild process immediately.

当target失败时，应及时检测并通知池（Raft）领导者，然后领导者将target从池中排除并立即触发重建过程。

Current status and long-term goal

Currently, since the raft leader can not exclude the target automatically, the sysadmin has to manually exclude the target from the pool, which then triggers the rebuild.

In the future, the leader should be able to detect the target failure promptly and then trigger the rebuild automatically by itself, without the help of the sysadmin.

目前，由于 raft leader 不能自动排除target，系统管理员必须手动从池中排除target，然后触发重建。 将来，leader应该能够及时发现target故障，然后自己自动触发rebuild，不需要sysadmin的帮助

Rebuild process

The rebuild is divided into 2 phases, scan and pull.

重建分为 2 个阶段，扫描和拉取。

Scan

Initially, the leader will propagate the failure notification to all other surviving targets by a collective RPC. Any target that receives this RPC will start to scan its object table to determine the objects lost data redundancy on the faulty target. If it does, send their IDs and related metadata to the rebuild targets(rebuild initiator). As for how to choose the rebuild target for faulty target, it will be described in placement/README.md

最初，领导者将通过集体 RPC 将失败通知传播到所有其他幸存的target。 任何收到此 RPC 的target都将开始扫描其对象表，以确定在故障target上丢失数据冗余的对象。 如果是，将它们的 ID 和相关元数据发送到重建target（重建发起者）。 关于faulty target如何选择rebuild target，会在placement/README.md中说明

Pull

Once the rebuild initiators get the object list from the scanning target, it will pull the data of these objects from other replicas and then write data locally. Each target will report its rebuild status, rebuilding objects, records, is\_finished? etc, to the pool leader. Once the leader learned all of targets finished its scanning and rebuilding phase, it will notify all targets the rebuild has finished, and they can release all of the resources held during the rebuild process.

重建发起者一旦从扫描target那里得到对象列表，就会从其他副本拉取这些对象的数据，然后在本地写入数据。 每个target都会报告它的重建状态，重建对象，记录，is\_finished？ 等等，给泳池领导。 当Leader得知所有target都完成了它的扫描和重建阶段后，它会通知所有target重建已经完成，他们可以释放重建过程中持有的所有资源。

**Rebuild Protocol**

图片包含 日历

描述已自动生成

The [figure](https://github.com/daos-stack/daos/blob/master/src/rebuild/README.md#f10.18) above is an example of this process: There are five objects in the cluster: object A is 3-way replicated, object B, C, D and E are 2-way replicated. When target-2 failed, target-0, which is the Raft leader, broadcasted the failure to all surviving targets to notify them to enter the degraded mode and scan:

* Target-0 found that object D lost a replica and calculated out target-1 is the rebuild target for D, so it sent object D's ID and its metadata to target-1.
* Target-1 found that object A lost a replica and calculated out target-3 is the rebuild target for A, so it sent object A's ID and its metadata to target-3.
* Target-4 found objects A and C lost replicas and it calculated out target-3 is the rebuild target for both objects A and C, so it sent IDs for objects A and C and their metadata to target-3.
* After receiving these object IDs and their metadata, target-1 and target-3 can compute out surviving replicas of these objects, and rebuild these objects by pulling data from these replicas.

上图是这个过程的一个例子：集群中有五个对象：对象A是3路复制，对象B、C、D和E是2路复制。当target-2失败时，作为Raft leader的target-0将失败广播给所有幸存的target，通知他们进入降级模式并进行扫描：

• 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 及其元数据发送给target-1。

• Target-1 发现对象A 丢失了一个副本并计算出target-3 是A 的重建目标，因此它将对象A 的ID 及其元数据发送给target-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target-3。

• 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。

Multiple pool and targets rebuild

In a large-scale storage cluster, multiple failures might occur when a rebuild from a previous failure is still in progress. In this case, DAOS should neither simultaneously handle these failures, nor interrupt and reset the earlier rebuilding progress for later failures. Otherwise, the time consumed for rebuilds for each failure might grow significantly and rebuilds may never end if new failures overlap with ongoing rebuilds. So for multiple failures, these rules are applied

在大型存储集群中，当从先前的故障重建仍在进行中时，可能会发生多次故障。 在这种情况下，DAOS 既不应该同时处理这些故障，也不应该为以后的故障中断和重置较早的重建进度。 否则，每次故障重建所消耗的时间可能会显着增加，如果新故障与正在进行的重建重叠，重建可能永远不会结束。 所以对于多次失败，应用这些规则

* If the rebuild initiator fails during rebuild, then the object shards being rebuilt on the initiator should be ignored, which will be handled by next rebuild.
* If rebuild initiator can not fetch the data from other replicas due to the failure, it will switch to other replicas if available.
* A target in rebuild does not need to re-scan its objects or reset rebuild progress for the current failure if another failure has occurred.
* When there are multiple failures, if the number of failed targets from different domains exceeds the fault tolerance level, there could be unrecoverable errors and applications could suffer from data loss. In this case, upper layer stack software could see errors while sending I/O to the object that could have missing data.

• 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，这将由下一次重建处理。

• 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）。 • 如果发生另一个故障，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。 • 当有多个故障时，如果来自不同域的故障目标数量超过容错级别，可能会出现不可恢复的错误，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丢失数据的对象发送 I/O 时可能会发现错误。

The following [figure](https://github.com/daos-stack/daos/blob/master/src/rebuild/README.md#f10.20) is an example of this protocol.

**Multi-failure protocol**

日历

描述已自动生成

* In this example, object A is 2-way replicated, object B, C and D are 3-way replicated.
* After failure of target-1, target-2 is the initiator of rebuilding object B, it is pulling data from target-3 and target-4; target-3 is the initiator of rebuilding object C, it is pulling data from target-0 and target-2.
* Target-3 failed before completing rebuild for target-1, so rebuild of object C should be abandoned at this point, because target-3 is the initiator of it. The missing data redundancy of object C will be reconstructed while rebuilding target-3.
* Because target-3 is also contributor of rebuilding object B, based on the protocol, the initiator of object B, which is target-2, should switch to target-4 and continue rebuild of object B.
* Rebuild process of target-1 can complete after finishing rebuild of object B. By this time, object C still lost a replica. This is quite understandable, because even if two failures have no overlap, object C will still lose the replica on target-3.
* In the process of rebuilding target-3, target-4 is the new initiator of rebuilding object C.

If there are multiple pools being impacted by the failing target, these pools can be rebuilt concurrently.

• 在本例中，对象A 为2 路复制，对象B、C 和D 为3 路复制。 • target-1 失败后，target-2 是重建对象B 的发起者，它正在从target-3 和target-4 拉取数据； target-3 是重建对象 C 的发起者，它正在从 target-0 和 target-2 拉取数据。 • 目标3 在完成目标1 的重建之前失败，因此此时应放弃对象C 的重建，因为目标3 是它的发起者。对象 C 的缺失数据冗余将在重建目标 3 时重建。 • 由于target-3也是重建对象B的贡献者，根据协议，对象B的发起者，即target-2，应该切换到target-4，继续重建对象B。 • 对象B 的重建完成后，目标1 的重建过程可以完成。此时，对象C 仍然丢失了一个副本。这很好理解，因为即使两次失败没有重叠，对象 C 仍然会丢失目标 3 上的副本。 • 在重建target-3的过程中，target-4是重建对象C的新发起者。

如果有多个池受到失败目标的影响，则可以同时重建这些池。

I/O during rebuild

If there are concurrent writes during rebuild, the rebuild protocol should guarantee that new writes will never be lost. Those writes should be either directly stored in the new object shard or pulled to the new object shard by the rebuild initiator. And also fetch should be guarantee to get the correct data. To achieve these, these protocols are applied

如果在重建期间有并发写入，重建协议应该保证新的写入永远不会丢失。 这些写入要么直接存储在新对象分片中，要么由重建发起者拉到新对象分片中。 并且还应该保证获取正确的数据。 为了实现这些，这些协议被应用

1. Fetch will always skip the rebuilding target.
2. Update can complete only if updates of all the object shards have successfully completed.

* If any of these updates failed, the client will infinitely retry until it succeeds, or there is a pool map change which shows the target failed. In the second case, the client will switch to the new pool map, and send the update to the new destination, which is the rebuild target of this object.

1. There are no synchronization between normal I/O and rebuild process, so during rebuild process, the data might be written duplicately by rebuild initiator and normal I/O.

1. Fetch 将始终跳过重建目标。

2. 只有所有对象分片的更新都成功完成后才能完成更新。 • 如果这些更新中的任何一个失败，客户端将无限重试直到成功，或者池映射更改显示目标失败。 在第二种情况下，客户端将切换到新的池映射，并将更新发送到新目的地，即此对象的重建目标。

3. 正常I/O和rebuild过程没有同步，所以在rebuild过程中，rebuild发起者和正常I/O可能会重复写入数据

Rebuild resource throttle

During rebuild process, the user can set the throttle to guarantee the rebuild will not use more resource than the user setting. The user can only set the CPU cycle for now. For example, if the user set the throttle to 50, then the rebuild will at most use 50% of CPU cycle to do rebuild job. The default rebuild throttle for CPU cycle is 30.

在重建过程中，用户可以设置节流阀以保证重建不会使用比用户设置更多的资源。 用户目前只能设置 CPU 周期。 例如，如果用户将油门设置为 50，那么重建将最多使用 50% 的 CPU 周期来完成重建工作。 CPU 周期的默认重建限制为 30。

Rebuild status

As described earlier, each target will report its rebuild status to the pool leader by IV, then the leader will summurize the status of all targets, and print out the whole rebuild status by every 2 seconds, for example these messages.

如前所述，每个目标都会通过IV向池领导报告其重建状态，然后领导者将汇总所有目标的状态，并每2秒打印一次整个重建状态，例如这些消息。

图片包含 文本

描述已自动生成

There are 2 pools being rebuilt (pool 8799e471 and pool 419d9c11, note: only first 8 letters of the pool uuid are shown here).

有 2 个池正在重建（池 8799e471 和池 419d9c11，注意：此处仅显示池 uuid 的前 8 个字母）。

-------------------------------------------------------------------------

The 1st line means the rebuild for pool 8799e471 is started, whose pool map version is 41.

The 2nd line means the rebuild for pool 8799e471 is in scanning phase, and no objects & records are being rebuilt yet.

The 3rd line means a rebuild job for pool 419d9c11 is being queued.

The 4th line means the rebuild for pool 419d9c11 is started, whose pool map version is 2.

The 5th line means the rebuild for pool 419d9c11 is in scanning phase, and no objects & records are being rebuilt yet.

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.

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.

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.

---------------------------------------------------------------------------

第 1 行表示开始重建池 8799e471，池映射版本为 41。 第 2 行表示池 8799e471 的重建处于扫描阶段，尚未重建任何对象和记录。 第 3 行表示池 419d9c11 的重建作业正在排队。 第 4 行表示已启动池 419d9c11 的重建，其池映射版本为 2。 第 5 行表示池 419d9c11 的重建处于扫描阶段，尚未重建任何对象和记录。 第6行表示pool 8799e471的rebuild在pull阶段，有75个对象需要rebuild(toberb\_obj=75)，全部rebuild(rb\_obj=75)，但是这些对象rebuild的记录还没有完成(done 0) 并且只重建了 11937 条记录 (rec = 11937)。 第7行表示池419d9c11的rebuild完成（done 1），总共有10个对象和1026条记录被重建，大约需要8秒。 第8行表示池8799e471的rebuild完成（done 1），总共有75个对象和13184条记录被重建，大约需要14秒。

During the rebuild, if the client query the pool status to the pool leader, which will return its rebuild status to client as well.

在重建过程中，如果客户端向池领导查询池状态，池领导也会将其重建状态返回给客户端。

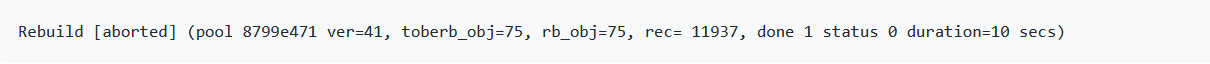
文本

描述已自动生成

Rebuild failure

If the rebuild is failed due to some failures, it will be aborted, and the related message will be shown on the leader console. For example:

如果由于某些故障而导致重建失败，则会中止，并在领导者控制台上显示相关消息。 例如



Rebuilding with Checksums

During a rebuild, the server being rebuilt will act as a DAOS Client in the sense that it will read the data and checksum from a replica server and verify the integrity of the data before it uses it for the rebuild. If corrupted data is detected, then the read will fail, and the replica server will be notified of the corruption. The rebuild will then attempt to use a different replica.

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，并在将数据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不同的副本。

A checksum iov parameter is available for the object list and object fetch task API's. This is for rebuild to provide memory that the checksums can be packed into. Otherwise, rebuild would have to recalculate the checksums while writing to the local VOS instance. If insufficient memory is allocated in the buffer, the iov\_len will be set to the required capacity and the checksums packed into the buffer is truncated.

校验和 iov 参数可用于对象列表和对象获取任务 API。 这是用于重建以提供校验和可以打包到的内存。 否则，在写入本地 VOS 实例时，重建将不得不重新计算校验和。 如果缓冲区中分配的内存不足，则 iov\_len 将设置为所需的容量，并截断打包到缓冲区中的校验和。

The following describes "touch points" of the life of a checksum for rebuild. The client task APIs and packing/unpacking info is included here because rebuild is the primary user of these APIs with checksums.

下面描述了用于重建的校验和生命周期的“接触点”。 客户端任务 API 和打包/解包信息包含在此处，因为重建是这些带有校验和的 API 的主要用户。

Rebuild Touch Points

* migrate\_fetch\_update\_(inline|single|bulk) - the rebuild/migrate functions that write to vos locally must ensure that the checksum is also written. These must use the csum iov param for fetch to get the checksum, then unpack the csums into iod\_csum.
* obj\_enum.c is relied on for enumerating the objects to be rebuilt. Because the fetch\_update functions will unpack the csums from fetch, it will also unpack the csums for enum, so the unpacking process in obj\_enum.c will simply copy the csum\_iov to the io (dss\_enum\_unpack\_io) structure in **enum\_unpack\_recxs()** and then deep copy to the mrone (migrate\_one) structure in **migrate\_one\_insert()**.

migrate\_fetch\_update\_(inline|single|bulk) - 写入本地vos 的重建/迁移函数必须确保校验和也被写入。 这些必须使用 csum iov 参数来获取校验和，然后将 csum 解压到 iod\_csum 中。 • obj\_enum.c 用于枚举要重建的对象。 因为 fetch\_update 函数会从 fetch 中解压 csum，也会为 enum 解包 csum，所以 obj\_enum.c 中的解包过程只是简单的将 csum\_iov 复制到 enum\_unpack\_recxs() 中的 io(dss\_enum\_unpack\_io) 结构体，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**: sets csum iov to daos\_obj\_fetch\_t args. These args are set to the rw\_cb\_args.shard\_args.api\_args and accessed through an accessor function (rw\_args2csum\_iov) in cli\_shard.c so that rw\_args\_store\_csum can easily access it. This function, called from dc\_rw\_cb\_csum\_verify, will pack the data checksums received from the server into the iov.
* **dc\_obj\_list\_obj\_task\_create**: sets csum iov to daos\_obj\_list\_obj\_t args. args.csum is then copied to obj\_enum\_args.csum in dc\_obj\_shard\_list(). On enum callback (dc\_enumerate\_cb()) the packed csum buffer is copied from the rpc args to obj\_enum\_args.csum (which points to the same buffer as the caller's)

dc\_obj\_fetch\_task\_create：将csum iov 设置为daos\_obj\_fetch\_t 参数。 这些 args 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) 访问，以便 rw\_args\_store\_csum 可以轻松访问它。 该函数从 dc\_rw\_cb\_csum\_verify 调用，将从服务器接收到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 参数。 然后将 args.csum 复制到 dc\_obj\_shard\_list() 中的 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（指向与调用者相同的缓冲区）

Packing/unpacking checksums

When checksums are packed (either for fetch or object list) only the data checksums are included. For object list, only checksums for data that is inlined is included. During a rebuild, if the data is not inlined, then the rebuild process will fetch the rest of the data and also get the checksums.

当校验和被打包（用于获取或对象列表）时，只包含数据校验和。 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() - "packs" checksums by appending the struct to an iov and then appending the checksum info buffer to the iov. This puts the actual checksum just after the checksum structure that describes the checksum.
* ci\_cast() - "unpacks" the checksum and describing structure. It does this by casting an iov's buffer to a dcs\_csum\_info struct and setting the csum\_info's checksum pointer to point to the memory just after the structure. It does not copy anything, but really just "casts". To get all dcs\_csum\_infos, a caller would cast the iov, copy the csum\_info to a destination, then move to the next csum\_info(ci\_move\_next\_iov) in the iov. Because this process modifies the iov structure it is best to use a copy of the iov as a temp structure.

• 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 的缓冲区转换为 dcs\_csum\_info 结构并将 csum\_info 的校验和指针设置为指向该结构之后的内存来实现此目的。 它不复制任何内容，而实际上只是“强制转换”。 要获取所有 dcs\_csum\_infos，调用者将转换 iov，将 csum\_info 复制到目的地，然后移动到 iov 中的下一个 csum\_info(ci\_move\_next\_iov)。 由于此过程会修改 iov 结构，因此最好使用 iov 的副本作为临时结构。

[Security](https://github.com/daos-stack/daos/blob/master/src/security/README.md)

Data Plane Security Module

The DAOS security module centralizes all access and security-related functionality in the DAOS data plane in a single module.

The functionality in this module includes:

* Client library requests to the DAOS Agent for a credential for authentication.
* Server requests to the Control Plane to validate a signed client credential.
* Generating default Access Control Lists for pools and containers.
* Deriving pool and container capabilities from the combination of the Access Control List and the client credential.
* Access control checks for pool and container operations.

Credential Generation and Validation

Details on the credential generation and validation process are outlined [in the Control Plane documentation](https://github.com/daos-stack/daos/blob/master/src/control/security/README.md).

Access Control Internals

This section covers the implementation details of pool and container access control.

See the [security overview](https://github.com/daos-stack/daos/blob/master/doc/overview/security.md#access-control-lists) for background on DAOS Access Control Lists.

See the [Admin Guide](https://github.com/daos-stack/daos/blob/master/doc/admin/pool_operations.md#access-control-lists) for a higher-level view of pool access control.

See the [User Guide](https://github.com/daos-stack/daos/blob/master/doc/user/container.md#access-control-lists) for a higher-level view of container access control.

Pool Access

When a client connects to a pool, the user credential that was used to make the connection is used, in combination with the pool ACL and the requested access type (RO or RW), to generate an internal set of security capabilities for the pool handle. These capabilities map to the valid permissions set for pools and are used for future access decisions for that pool handle, for pool-level operations such as container creation.

These capabilities persist for the life of the handle, even if someone modifies the pool ACL or ownership. The modifications won't be used for access decisions until the user attempts to get a new handle for the pool. The pool handle cannot be revoked.

In addition, the validated credential is saved in the handle data in the DAOS data plane during pool connect, and remains associated with the pool handle for its lifetime. This original credential is used for access decisions when the user interacts with existing containers. It is used both when the user attempts to open a container (acquire a container handle) and when the user attempts to delete a container, for which they may not be holding a handle.

Container Access

The client process to open a container is similar to that for pool connection. A set of security capabilities is determined for the container handle based on the combination of the container ACL and the credential associated with the pool handle, along with the type of access requested (RO or RW). The security capabilities calculated at the time of the container open are used for container access decisions throughout the lifetime of the container handle. The container handle cannot be revoked.

Container Destroy

Container deletion is a special operation in the context of access control. There are two levels of permission that may have been granted.

At the pool level, a user may have been granted the administrator-like privilege of deleting any container, even those that they have no access to. This is the first and fastest check: this permission is included in the pool handle's security capabilities.

If the user does not have the pool level delete privilege, we must determine the security capabilities of the pool handle's credential on the container before we know if the holder of the handle is permitted to delete it. This is the same process used during container open, wherein the credential and container ACL are used together. If the user has permission via the container ACL to delete that specific container, the operation is allowed to proceed.

###### Software Compatibility

Interoperability in DAOS is handled via protocol and schema versioning for persistent data structures.

Protocol Compatibility

Limited protocol interoperability is to be provided by the DAOS storage stack. Version compatibility checks will be performed to verify that:

* All targets in the same pool run the same protocol version.
* Client libraries linked with the application may be up to one protocol version older than the targets.

If a protocol version mismatch is detected among storage targets in the same pool, the entire DAOS system will fail to start up and will report failure to the control API. Similarly, connection from clients running a protocol version incompatible with the targets will return an error.

PM Schema Compatibility and Upgrade

The schema of persistent data structures may evolve from time to time to fix bugs, add new optimizations or support new features. To that end, the persistent data structures support schema versioning.

Upgrading the schema version is not done automatically and must be initiated by the administrator. A dedicated upgrade tool will be provided to upgrade the schema version to the latest one. All targets in the same pool must have the same schema version. Version checks are performed at system initialization time to enforce this constraint.

To limit the validation matrix, each new DAOS release will be published with a list of supported schema versions. To run with the new DAOS release, administrators will then need to upgrade the DAOS system to one of the supported schema version. New target will always be reformatted with the latest version. This versioning schema only applies to data structure stored in persistent memory and not to block storage that only stores user data with no metadata.

###### Network Transport and Communications

As introduced in the previous section, DAOS uses three different communication channels.

gRPC and Protocol Buffers

gRPC provides a bi-directional secured channel for DAOS management. It relies on TLS/SSL to authenticate the administrator role and the servers. Protocol buffers are used for RPC serialization and all proto files are located in the [proto](https://github.com/daos-stack/daos/blob/master/src/proto) directory.

dRPC

dRPC is communication channel built over Unix Domain Socket that is used for inter-process communications. It provides both a [C](https://github.com/daos-stack/daos/blob/master/src/common/README.md#dRPC-C-API) and [Go](https://github.com/daos-stack/daos/blob/master/src/control/drpc/README.md) interface to support interactions between:

* the daos\_agent and libdaos for application process authentication
* the daos\_server (control plane) and the daos\_engine (data plane) daemons Like gRPC, RPC are serialized via protocol buffers.

CART

[CART](https://github.com/daos-stack/cart) is a userspace function shipping library that provides low-latency high-bandwidth communications for the DAOS data plane. It supports RDMA capabilities and scalable collective operations. CART is built over [Mercury](https://github.com/mercury-hpc/mercury) and [libfabric](https://ofiwg.github.io/libfabric/). The CART library is used for all communications between libdaos and daos\_engine instances.

## Developer Zone

#### Development Environment

This section covers specific instructions to create a developer-friendly environment to contribute to the DAOS development. This includes how to regenerate the protobuf files or add new Go package dependencies, which is only required for development purposes.

##### Building DAOS for Development

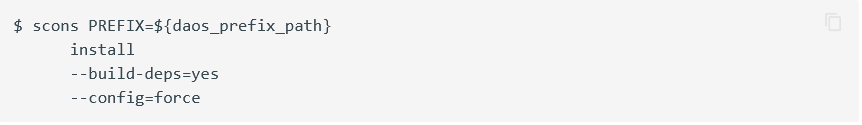
The DAOS repository is hosted on [GitHub](https://github.com/daos-stack/daos). To checkout the current development version, simply run:

$ git clone --recurse-submodules https://github.com/daos-stack/daos.git

For a specific branch or tag (e.g. v1.2), add -b v1.2 to the command line above.

Prerequisite when built using --build-deps are installed in component specific directories under PREFIX/prereq/$TARGET\_TYPE.

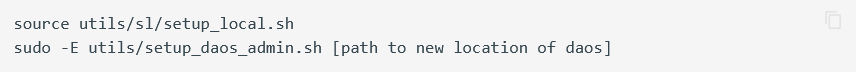
Run the following scons command:



Installing the components into separate directories allow upgrading the components individually by replacing --build-deps=yes with --update-prereq={component\\_name}. This requires a change to the environment configuration from before. For automated environment setup, source utils/sl/utils/setup\_local.sh.

The install path should be relocatable with the exception that daos\_admin will not be able to find the new location of daos and dependencies. All other libraries and binaries should work without any change due to relative paths. Editing the .build-vars.sh file to replace the old with the new can restore the capability of setup\_local.sh to automate path setup.

To run daos\_server, either the rpath in daos\_admin needs to be patched to the new installation location of spdk and isal or LD\_LIBRARY\_PATH needs to be set. This can be done using SL\_SPDK\_PREFIX and SL\_ISAL\_PREFIX set when sourcing setup\_local.sh. This can also be done with the following commands:



This script is intended only for developer setup of daos\_admin.

With this approach, DAOS gets built using the prebuilt dependencies in ${daos\_prefix\_path}/prereq, and required options are saved for future compilations. So, after the first time, during development, only "scons --config=force" and "scons --config=force install" would suffice for compiling changes to DAOS source code.

If you wish to compile DAOS with clang rather than gcc, set COMPILER=clang on the scons command line. This option is also saved for future compilations.

Additionally, users can specify BUILD\_TYPE=[dev|release|debug] and scons will save the intermediate build for the various BUILD\_TYPE, COMPILER, and TARGET\_TYPE options so a user can switch between options without a full rebuild and thus with minimal cost. By default, TARGET\_TYPE is set to 'default' which means it uses the BUILD\_TYPE setting. To avoid rebuilding prerequisites for every BUILD\_TYPE setting, TARGET\_TYPE can be explicitly set to a BUILD\_TYPE setting to always use that set of prerequisites. These settings are stored in daos.conf so setting the values on subsequent builds is not necessary.

If needed, ALT\_PREFIX can be set to a colon separated prefix path where to look for already built components. If set, the build will check these paths for components before proceeding to build.

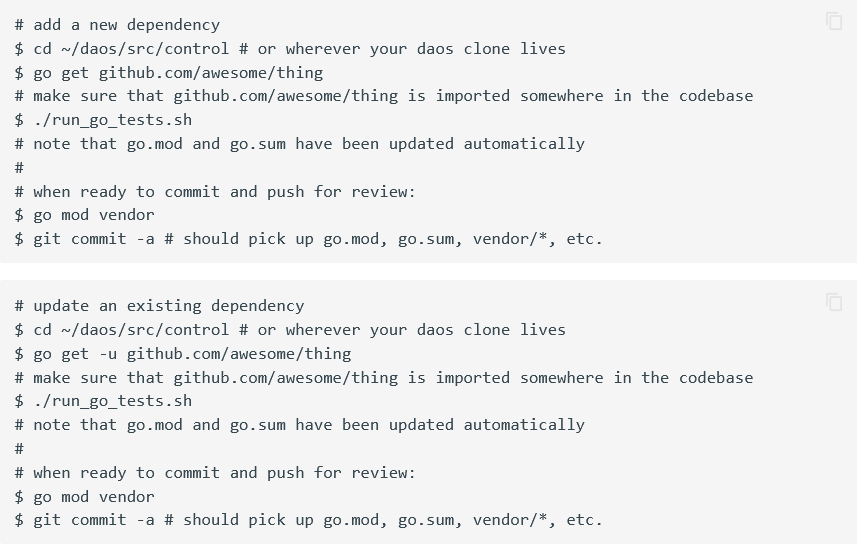
##### Go dependencies

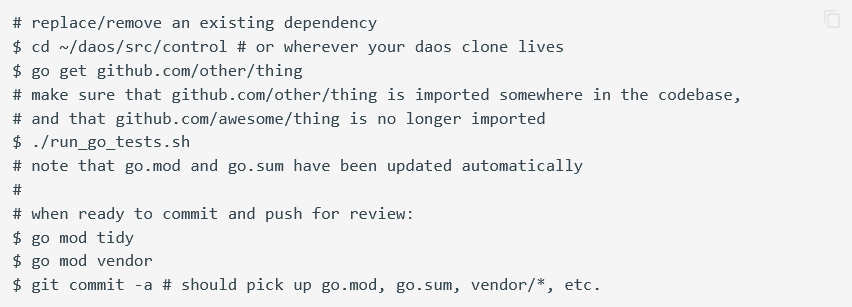
Developers contributing Go code may need to change the external dependencies located in the src/control/vendor directory. The DAOS codebase uses [Go Modules](https://github.com/golang/go/wiki/Modules) to manage these dependencies. As this feature is built in to Go distributions starting with version 1.11, no additional tools are needed to manage dependencies.

Among other benefits, one of the major advantages of using Go Modules is that it removes the requirement for builds to be done within the $GOPATH, which simplifies our build system and other internal tooling.

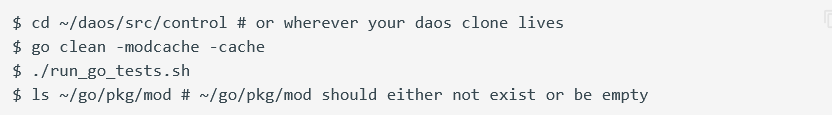
While it is possible to use Go Modules without checking a vendor directory into SCM, the DAOS project continues to use vendored dependencies in order to insulate our build system from transient network issues and other problems associated with nonvendored builds.

The following is a short list of example workflows. For more details, please refer to [one](https://github.com/golang/go/wiki/Modules#quick-start) of [the](https://engineering.kablamo.com.au/posts/2018/just-tell-me-how-to-use-go-modules/) [many](https://blog.golang.org/migrating-to-go-modules) resources available online.





In all cases, after updating the vendor directory, it is a good idea to verify that your changes were applied as expected. In order to do this, a simple workflow is to clear the caches to force a clean build and then run the test script, which is vendor-aware and will not try to download missing modules:



##### Protobuf Compiler

The DAOS control plane infrastructure uses [Protocol Buffers](https://github.com/protocolbuffers/protobuf) as the data serialization format for its RPC requests. Not all developers will need to compile the \\*.proto files, but if Protobuf changes are needed, the developer must regenerate the corresponding C and Go source files using a Protobuf compiler compatible with proto3 syntax.

##### Recommended Versions

The recommended installation method is to clone the git repositories, check out the tagged releases noted below, and install from source. Later versions may work, but are not guaranteed. You may encounter installation errors when building from source relating to insufficient permissions. If that occurs, you may try relocating the repo to /var/tmp/ in order to build and install from there.

* [Protocol Buffers](https://github.com/protocolbuffers/protobuf) v3.11.4. [Installation instructions](https://github.com/protocolbuffers/protobuf/blob/master/src/README.md).
* [Protobuf-C](https://github.com/protobuf-c/protobuf-c) v1.3.3. [Installation instructions](https://github.com/protobuf-c/protobuf-c/blob/master/README.md).
* gRPC plugin: [protoc-gen-go](https://github.com/golang/protobuf) is the version specified in [go.mod](https://github.com/daos-stack/daos/blob/master/src/control/go.mod). This plugin is automatically installed by the Makefile in $DAOSREPO/src/proto.

##### Compiling Protobuf Files

The source (.proto) files live under $DAOSREPO/src/proto. The preferred mechanism for generating compiled C/Go protobuf definitions is to use the Makefile in this directory. Care should be taken to keep the Makefile updated when source files are added or removed, or generated file destinations are updated.

Note that the generated files are checked into SCM and are not generated as part of the normal DAOS build process. This allows developers to ensure that the generated files are correct after any changes to the source files are made.

$ cd ~/daos/src/proto # or wherever your daos clone lives

$ make

protoc -I /home/foo/daos/src/proto/mgmt/ --go\_out=plugins=grpc:/home/foo/daos/src/control/common/proto/mgmt/ acl.proto

protoc -I /home/foo/daos/src/proto/mgmt/ --go\_out=plugins=grpc:/home/foo/daos/src/control/common/proto/mgmt/ mgmt.proto

...

$ git status

...

# modified: ../control/common/proto/mgmt/acl.pb.go

# modified: ../control/common/proto/mgmt/mgmt.pb.go

...

After verifying that the generated C/Go files are correct, add and commit them as you would any other file.

####  [Contributing](https://docs.daos.io/dev/contributing/)

####  [DAOS Internals](https://github.com/daos-stack/daos/blob/master/src/README.md)

####  [DAOS API Documentation](https://docs.daos.io/html/)