**SPDK（存储性能开发套件）官方文档中文版**

（2019年8月版，译：王海亮）

目录

[第一章 简介 1](#_Toc17204871)

[1.1.什么是SPDK？ 1](#_Toc17204872)

[1.2.入门 1](#_Toc17204873)

[1.3. Vagrant开发环境 3](#_Toc17204874)

[1.4.更新日志（略） 6](#_Toc17204875)

[第二章 概念 6](#_Toc17204876)

[2.1. 用户空间驱动程序\*\* 6](#_Toc17204877)

[2.2. 来自用户空间的DMA\*\* 7](#_Toc17204878)

[2.3. 消息传递和并发\*\* 9](#_Toc17204879)

[2.4. NAND Flash SSD内部 13](#_Toc17204880)

[2.5. 将I / O提交到NVMe设备\*\* 15](#_Toc17204881)

[2.5.1 NVMe规范 15](#_Toc17204882)

[2.5.2 SPDK NVMe驱动程序I / O路径 15](#_Toc17204883)

[2.6. 使用Vhost-user进行虚拟化I / O. 16](#_Toc17204884)

[2.6.1 介绍 16](#_Toc17204885)

[2.6.2 QEMU 17](#_Toc17204886)

[2.6.3 设备初始化 18](#_Toc17204887)

[2.6.4 I / O路径 19](#_Toc17204888)

[2.6.5 SPDK优化 20](#_Toc17204889)

[2.7. SPDK目录结构概述 20](#_Toc17204890)

[2.8. SPDK移植指南 22](#_Toc17204891)

[第三章 用户指南 22](#_Toc17204892)

[3.1. 系统配置用户指南 22](#_Toc17204893)

[3.1.1 IOMMU配置 22](#_Toc17204894)

[3.2. SPDK应用程序概述 23](#_Toc17204895)

[3.2.1 配置SPDK应用程序 23](#_Toc17204896)

[3.3. iSCSI Target 26](#_Toc17204897)

[3.3.1. iSCSI Target入门指南 26](#_Toc17204898)

[3.3.2. 通过配置文件配置iSCSI Target 27](#_Toc17204899)

[3.3.3. 通过RPC方法配置iSCSI Target 28](#_Toc17204900)

[3.3.4. 配置iSCSI启动器 29](#_Toc17204901)

[3.3.5. rpc配置示例\*\*\* 30](#_Toc17204902)

[3.3.6. iSCSI 热插拔 32](#_Toc17204903)

[3.4. NVMe over Fabrics Target 32](#_Toc17204904)

[3.5. Vhost Target（略） 37](#_Toc17204905)

[3.6 块设备用户指南 38](#_Toc17204906)

[3.6.1 bdev介绍 38](#_Toc17204907)

[3.6.2 通用RPC命令 38](#_Toc17204908)

[3.6.3 Ceph RBD 39](#_Toc17204909)

[3.6.4 压缩虚拟Bdev模块 40](#_Toc17204910)

[3.6.5 加密虚拟Bdev模块 41](#_Toc17204911)

[3.6.6 延迟vbdev模块 41](#_Toc17204912)

[3.6.7 GPT（GUID分区表） 42](#_Toc17204913)

[3.6.8 iSCSI bdev 43](#_Toc17204914)

[3.6.9 Linux AIO bdev 43](#_Toc17204915)

[3.6.10 OCF虚拟bdev 43](#_Toc17204916)

[3.6.11 Malloc bdev 44](#_Toc17204917)

[3.6.12 NULL bdev 44](#_Toc17204918)

[3.6.13 NVMe bdev 44](#_Toc17204919)

[3.6.14 逻辑卷Lvol 45](#_Toc17204920)

[3.6.15 RAID 46](#_Toc17204921)

[3.6.16 Passthru 46](#_Toc17204922)

[3.6.17 Pmem 46](#_Toc17204923)

[3.6.18 Virtio Block 47](#_Toc17204924)

[3.6.19 Virtio SCSI 47](#_Toc17204925)

[3.7 BlobFS（Blobstore文件系统） 48](#_Toc17204926)

[3.7.1 RocksDB集成 48](#_Toc17204927)

[3.7.2 FUSE插件 49](#_Toc17204928)

[3.8 JSON-RPC方法（略） 49](#_Toc17204929)

[第四章 程序员指南 49](#_Toc17204930)

[4.1. Blobstore程序员指南 49](#_Toc17204931)

[4.1.1 介绍 50](#_Toc17204932)

[4.1.2 运作理论 50](#_Toc17204933)

[4.1.3 设计注意事项 52](#_Toc17204934)

[4.1.4 例子 54](#_Toc17204935)

[4.1.5配置 54](#_Toc17204936)

[4.1.6 组件细节 54](#_Toc17204937)

[4.2. 块设备层编程指南 56](#_Toc17204938)

[4.3 编写自定义块设备模块 58](#_Toc17204939)

[4.3.1 介绍 58](#_Toc17204940)

[4.3.2 创建一个新模块 59](#_Toc17204941)

[4.3.3创建虚拟Bdev 60](#_Toc17204942)

[4.4 NVMe over Fabrics目标编程指南 61](#_Toc17204943)

[4.4.1 介绍 61](#_Toc17204944)

[4.4.2 原语结构体 61](#_Toc17204945)

[4.4.3 基础函数 62](#_Toc17204946)

[4.4.4访问控制 62](#_Toc17204947)

[4.4.5发现子系统 62](#_Toc17204948)

[4.4.6 传输 63](#_Toc17204949)

[4.4.7选择线程模型 63](#_Toc17204950)

[4.4.8 跨CPU核心扩展 63](#_Toc17204951)

[4.4.9 零拷贝支持 63](#_Toc17204952)

[4.4.10 RDMA 63](#_Toc17204953)

[4.5 Flash传输层 64](#_Toc17204954)

[4.5.1 术语 64](#_Toc17204955)

[4.5.2 使用方法 67](#_Toc17204956)

[4.6 GDB宏用户指南 69](#_Toc17204957)

[4.6.1 介绍 69](#_Toc17204958)

[4.6.2 加载gdb宏 71](#_Toc17204959)

[4.6.3 使用gdb数据目录 72](#_Toc17204960)

[4.6.4 使用.gdbinit加载宏 72](#_Toc17204961)

[4.6.5 为什么我们需要显式调用spdk\_load\_macros 72](#_Toc17204962)

[4.6.6 以上可用的宏总结 73](#_Toc17204963)

[4.6.7 添加新宏 73](#_Toc17204964)

[4.7 SPDK “Reduce”块压缩算法 73](#_Toc17204965)

[4.7.1 介绍 73](#_Toc17204966)

[4.7.2 例子 74](#_Toc17204967)

[4.8 通知库 78](#_Toc17204968)

[第五章 基本信息 79](#_Toc17204969)

[5.1 事件框架 79](#_Toc17204970)

[5.1.1 事件框架设计注意事项 80](#_Toc17204971)

[5.1.2 SPDK事件框架组件 80](#_Toc17204972)

[5.1.3 应用框架 80](#_Toc17204973)

[5.2 逻辑卷 81](#_Toc17204974)

[5.2.1 术语 81](#_Toc17204975)

[5.2.2 配置逻辑卷 84](#_Toc17204976)

[5.3 矢量数据包处理（略） 86](#_Toc17204977)

[第六章 杂项 86](#_Toc17204978)

[6.1 介绍 86](#_Toc17204979)

[6.2 NVMe的P2P API 86](#_Toc17204980)

[6.3 确定设备支持 87](#_Toc17204981)

[6.4 P2P问题 87](#_Toc17204982)

[第七章 驱动程序 88](#_Toc17204983)

[7.1 NVMe驱动程序\*\*\* 88](#_Toc17204984)

[7.1.1 介绍 88](#_Toc17204985)

[7.1.2 例子 88](#_Toc17204986)

[7.1.3 公共接口 89](#_Toc17204987)

[7.1.4 NVMe驱动程序设计 89](#_Toc17204988)

[7.1.5 NVMe over Fabrics主机支持 91](#_Toc17204989)

[7.1.6 NVMe多进程 91](#_Toc17204990)

[7.1.7 NVMe Hotplug 92](#_Toc17204991)

[7.2 I/OAT驱动程序 93](#_Toc17204992)

[7.2.1 公共接口 93](#_Toc17204993)

[7.2.2 关键功能 93](#_Toc17204994)

[7.3 Virtio驱动程序 93](#_Toc17204995)

[7.3.1 介绍 93](#_Toc17204996)

[7.3.2 2MB大页面 93](#_Toc17204997)

[第八章 工具 94](#_Toc17204998)

[8.1 SPDK CLI 94](#_Toc17204999)

[8.1.1 安装所需的依赖项 94](#_Toc17205000)

[8.1.2 运行SPDK应用程序实例 94](#_Toc17205001)

[8.1.3 运行SPDK CLI 94](#_Toc17205002)

[8.1.4 可选 - 创建Python虚拟环境 94](#_Toc17205003)

[8.2 nvme-CLI 95](#_Toc17205004)

[8.2.1 nvme-cli with SPDK入门指南 95](#_Toc17205005)

[8.2.2 使用场景 95](#_Toc17205006)

[第九章 性能测试报告（略） 96](#_Toc17205007)

[第十章NVMe-oF Target跟踪点\*\*\* 96](#_Toc17205008)

[10.1 介绍 96](#_Toc17205009)

[10.2 启用跟踪点 97](#_Toc17205010)

[10.3 捕获事件的快照 97](#_Toc17205011)

[10.4 捕获足够的跟踪事件 98](#_Toc17205012)

[10.5 添加新的跟踪点 99](#_Toc17205013)

# 第一章 简介

## 1.1.什么是SPDK？

存储性能开发工具包（SPDK）提供了一组工具和库，用于编写高性能，可伸缩的用户模式存储应用程序。它通过使用一些关键技术实现了高性能：

将所有必需的驱动程序移动到用户空间，这样可以避免系统调用并启用应用程序的零拷贝访问。

轮询硬件用于完成而不是依赖中断，这降低了总延迟和延迟差异。

避免I / O路径中的所有锁定，而是依赖于消息传递。

SPDK的基石是用户空间，轮询模式，异步，无锁[NVMe](http://www.nvmexpress.org/)驱动程序。这提供了从用户空间应用程序直接到SSD的零拷贝，高度并行访问。驱动程序被编写为带有单个公共头的C库。有关详细信息，请参阅7.1 NVMe驱动程序。

SPDK还提供了一个完整的块堆栈作为用户空间库，它执行许多与操作系统中的块堆栈相同的操作。这包括统一不同存储设备之间的接口，排队以处理诸如内存不足或I / O挂起以及逻辑卷管理等情况。有关详细信息，请参阅3.6块设备用户指南。

最后，SPDK提供基于这些组件构建的[NVMe-oF](http://www.nvmexpress.org/nvm-express-over-fabrics-specification-released)，[iSCSI](https://en.wikipedia.org/wiki/ISCSI)和[vhost](http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html)服务器，这些服务器能够通过网络或其他进程提供磁盘。NVMe-oF和iSCSI的标准Linux内核启动器与这些目标以及带有vhost的QEMU互操作。与其他实现相比，这些服务器的CPU效率可高达一个数量级。这些目标可用作如何实现高性能存储目标的示例，或用作生产部署的基础。

## 1.2.入门

获取源代码

git clone https://github.com/spdk/spdk

cd spdk

git submodule update --init

安装先决条件

该scripts/pkgdep.sh脚本将自动安装构建和开发SPDK所需的全部依赖项。

sudo scripts/pkgdep.sh

建造

Linux的：

./configure

make

FreeBSD：注意：确保在/ usr / src /中有匹配的内核源代码

./configure

gmake

配置脚本有许多选项，可以通过运行查看

./configure --help

请注意，默认情况下并非所有功能都已启用。例如，默认情况下不启用RDMA支持（因此NVMe over Fabrics）。您可以通过执行以下操作来启用它：

./configure --with-rdma

make

运行单元测试

通过运行单元测试确认您的构建工作总是一个好主意。

./test/unit/unittest.sh

运行单元测试时，您将看到几条错误消息，但它们是测试套件的一部分。脚本末尾的最后一条消息表示成功或失败。

运行示例应用程序

在运行SPDK应用程序之前，必须分配一些大页面，并且必须从本机内核驱动程序中取消绑定任何NVMe和I / OAT设备。SPDK包含一个脚本，可以在Linux和FreeBSD上自动执行此过程。该脚本应该以root身份运行。它只需要在系统上运行一次。

sudo scripts/setup.sh

要将设备重新绑定回内核，您可以运行

sudo scripts/setup.sh reset

默认情况下，脚本分配2048MB的大页面。要更改此数字，请指定HUGEMEM（以MB为单位），如下所示：

sudo HUGEMEM = 4096 scripts/setup.sh

在Linux机器上，HUGEMEM将向上舍入到系统默认的巨大页面大小边界。

可以通过运行查看所有可用的参数

scripts/setup.sh help

示例代码位于examples目录中。这些示例是作为构建过程的一部分自动编译的。只需调用任何没有参数的示例即可查看帮助输出。如果您的系统启用了IOMMU，则可以以普通用户身份运行示例。如果没有，则需要以特权用户（root）身份运行。

一个很好的例子是examples/nvme/identify/identify，打印出有关系统上所有NVMe设备的信息。

app目录中提供了更大，功能更全面的应用程序。这包括iSCSI和NVMe-oF target。

## 1.3. Vagrant开发环境

**简介**

[Vagrant](https://www.vagrantup.com/)提供了一种快速方法，可以在不需要任何特殊硬件的情况下运行基于NVMe的基本虚拟机沙盒。SPDK的Vagrant环境支持各种Linux发行版以及FreeBSD。运行scripts / vagrant / create\_vbox.sh -h以查看完整列表。此环境需要Vagrant 1.9.4或更高版本以及具有匹配VirtualBox扩展包的VirtualBox 5.1或更高版本。

注意：如果您在企业防火墙后面，请在尝试启动VM之前设置http\_proxy并https\_proxy在您的环境中。还要确保已安装可选的vagrant模块 vagrant-proxyconf:

export http\_proxy=...

export https\_proxy=...

vagrant plugin install vagrant-proxyconf

如果你想使用kvm / libvirt，你也应该安装 vagrant-libvirt

VM 配置

要创建具有vagrant的已配置VM，您需要运行create\_vbox.sh脚本。

基本上，脚本将根据您选择的分发创建一个新的子目录，将vagrant配置文件（aka Vagrantfile）复制到它，并运行vagrant up脚本参数定义的一些设置。

默认情况下，创建的VM配置为：

* 2个vCPU
* 4G的RAM
* 2个NIC（1个NAT - 主机访问，1个专用网络）

要修改某些高级设置（如配置和rsyncing），您可能需要更改Vagrantfile源。

有关其他支持，请使用Vagrant帮助功能来了解如何销毁，重新启动等。下面是成功VM启动和执行NVMe hello world示例应用程序的示例输出。

vagrant --help

**运行一个例子**

下面显示了启动Ubuntu18 VM的示例输出，在其上编译SPDK并运行NVMe示例应用程序hello\_world。如果您在lspci输出和应用程序输出中都没有看到如下所示的NVMe设备，则可能存在VirtualBox和/或Vagrant版本控制问题。

user@dev-system:~$ cd spdk/scripts/vagrant

user@dev-system:~/spdk/scripts/vagrant$ ./create\_vbox.sh ubuntu18

mkdir: created directory '/home/user/spdk/scripts/vagrant/ubuntu18'

~/spdk/scripts/vagrant/ubuntu18 ~/spdk/scripts/vagrant

vagrant-proxyconf already installed... skipping

Bringing machine 'default' up with 'virtualbox' provider...

==> default: Box 'bento/ubuntu-18.04' could not be found. Attempting to find and install...

default: Box Provider: virtualbox

default: Box Version: 201803.24.0

==> default: Loading metadata for box 'bento/ubuntu-18.04'

default: URL: https://vagrantcloud.com/bento/ubuntu-18.04

==> default: Adding box 'bento/ubuntu-18.04' (v201803.24.0) for provider: virtualbox

default: Downloading: https://vagrantcloud.com/bento/boxes/ubuntu-18.04/versions/201803.24.0/providers/virtualbox.box

==> default: Box download is resuming from prior download progress

==> default: Successfully added box 'bento/ubuntu-18.04' (v201803.24.0) for 'virtualbox'!

==> default: Importing base box 'bento/ubuntu-18.04'...

==> default: Matching MAC address for NAT networking...

==> default: Setting the name of the VM: ubuntu18\_default\_1237088131451\_82174

==> default: Fixed port collision for 22 => 2222. Now on port 2202.

==> default: Clearing any previously set network interfaces...

==> default: Preparing network interfaces based on configuration...

default: Adapter 1: nat

default: Adapter 2: hostonly

==> default: Forwarding ports...

default: 22 (guest) => 2202 (host) (adapter 1)

==> default: Running 'pre-boot' VM customizations...

==> default: Booting VM...

==> default: Waiting for machine to boot. This may take a few minutes...

default: SSH address: 127.0.0.1:2202

default: SSH username: vagrant

default: SSH auth method: private key

default: Warning: Remote connection disconnect. Retrying...

default: Warning: Connection reset. Retrying...

<<some output trimmed>>

default: Warning: Connection reset. Retrying...

default: Warning: Remote connection disconnect. Retrying...

default:

default: Vagrant insecure key detected. Vagrant will automatically replace

default: this with a newly generated keypair for better security.

default:

default: Inserting generated public key within guest...

default: Removing insecure key from the guest if it's present...

default: Key inserted! Disconnecting and reconnecting using new SSH key...

==> default: Machine booted and ready!

==> default: Checking for guest additions in VM...

==> default: Configuring and enabling network interfaces...

==> default: Configuring proxy for Apt...

==> default: Configuring proxy environment variables...

==> default: Rsyncing folder: /home/user/spdk/ => /home/vagrant/spdk\_repo/spdk

==> default: Mounting shared folders...

default: /vagrant => /home/user/spdk/scripts/vagrant/ubuntu18

==> default: Running provisioner: file...

SUCCESS!

cd to ubuntu18 and type "vagrant ssh" to use.

Use vagrant "suspend" and vagrant "resume" to stop and start.

Use vagrant "destroy" followed by "rm -rf ubuntu18" to destroy all trace of vm.

检查环境

user@dev-system:~/spdk/scripts/vagrant$ cd ubuntu18

user@dev-system:~/spdk/scripts/vagrant/ubuntu18$ vagrant ssh

Welcome to Ubuntu Bionic Beaver (development branch) (GNU/Linux 4.15.0-12-generic x86\_64)

<<some output trimmed>>

vagrant@vagrant:~$ lspci | grep "Non-Volatile"

00:0e.0 Non-Volatile memory controller: InnoTek Systemberatung GmbH Device 4e56

vagrant@vagrant:~$ ls

spdk\_repo

编译SPDK并运行一个示例

vagrant@vagrant:~/spdk\_repo/spdk$ sudo apt update

<<output trimmed>>

vagrant@vagrant:~/spdk\_repo/spdk$ sudo scripts/pkgdep.sh

<<output trimmed>>

vagrant@vagrant:~/spdk\_repo/spdk$ ./configure

Creating mk/config.mk...done.

Type 'make' to build.

vagrant@vagrant:~/spdk\_repo/spdk$ make

<<output trimmed>>

vagrant@vagrant:~/spdk\_repo/spdk$ sudo ./scripts/setup.sh

0000:00:0e.0 (80ee 4e56): nvme -> uio\_pci\_generic

vagrant@vagrant:~/spdk\_repo/spdk$ sudo examples/nvme/hello\_world/hello\_world

Starting SPDK v18.10-pre / DPDK 18.05.0 initialization...

[ DPDK EAL parameters: hello\_world -c 0x1 --legacy-mem --file-prefix=spdk0 --base-virtaddr=0x200000000000 --proc-type=auto ]

EAL: Detected 4 lcore(s)

EAL: Detected 1 NUMA nodes

EAL: Auto-detected process type: PRIMARY

EAL: Multi-process socket /var/run/dpdk/spdk0/mp\_socket

EAL: Probing VFIO support...

Initializing NVMe Controllers

EAL: PCI device 0000:00:0e.0 on NUMA socket 0

EAL: probe driver: 80ee:4e56 spdk\_nvme

Attaching to 0000:00:0e.0

Attached to 0000:00:0e.0

Using controller ORCL-VBOX-NVME-VER12 (VB1234-56789 ) with 1 namespaces.

Namespace ID: 1 size: 1GB

Initialization complete.

INFO: using host memory buffer for IO

Hello world!

## 1.4.更新日志（略）

# 第二章 概念

## 2.1. 用户空间驱动程序\*\*

**从用户空间控制硬件**

SPDK的大部分文档都讨论了用户空间驱动程序，因此了解技术层面的含义非常重要。首先，驱动程序是直接控制连接到计算机的特定设备的软件。其次，操作系统根据权限级别（[内核空间和用户空间）](https://en.wikipedia.org/wiki/User_space)将系统的虚拟内存分为两类地址。这种分离由CPU本身的功能辅助，这些功能强制执行称为[保护环的](https://en.wikipedia.org/wiki/Protection_ring)内存分离。通常，驱动程序在内核空间中运行（即x86上的ring 0）。SPDK包含的驱动程序设计为在用户空间中运行，但它们仍然直接与它们控制的硬件设备连接。

为了让SPDK控制设备，它必须首先指示操作系统放弃控制。这通常被称为解除设备内核驱动程序的绑定，而Linux则通过[写入sysfs中的文件](https://lwn.net/Articles/143397/)来完成。然后，SPDK将驱动程序重新绑定到与Linux捆绑的两个特殊设备驱动程序之一 - [uio](https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html)或[vfio](https://www.kernel.org/doc/Documentation/vfio.txt)。这两个驱动程序是“虚拟”驱动程序，因为它们主要向操作系统指示设备有绑定的驱动程序，因此它不会自动尝试重新绑定默认驱动程序。它们实际上并没有以任何方式初始化硬件，也不了解它是什么类型的设备。

uio和vfio的主要区别在于vfio能够对平台进行编程[IOMMU](https://en.wikipedia.org/wiki/Input%E2%80%93output_memory_management_unit)，是确保用户空间驱动程序内存安全的关键硬件。有关完整详细信息，请参阅[从用户空间直接内存访问（DMA）](https://spdk.io/doc/memory.html)。

一旦设备从操作系统内核解除绑定，操作系统就不能再使用它了。例如，如果在Linux上取消绑定NVMe设备，则与其对应的设备（例如/ dev / nvme0n1）将消失。这进一步意味着安装在设备上的文件系统也将被删除，内核文件系统也无法再与设备交互。实际上，不再涉及整个内核块存储堆栈。相反，SPDK在典型的操作系统存储堆栈中提供了大多数层的重新构想的实现，所有这些都是可以直接嵌入到应用程序中的C库。这主要包括[块设备抽象层](https://spdk.io/doc/bdev.html)，但也包括[块分配器](https://spdk.io/doc/blob.html)和类[文件系统组件](https://spdk.io/doc/blobfs.html)。

用户空间驱动程序利用uio或vfio中的功能将设备的[PCI BAR](https://en.wikipedia.org/wiki/PCI_configuration_space)映射到当前进程，从而允许驱动程序直接执行[MMIO](https://en.wikipedia.org/wiki/Memory-mapped_I/O)。例如，SPDK [NVMe驱动程序](https://spdk.io/doc/nvme.html)映射NVMe设备的BAR，然后跟随[NVMe规范](http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf)来初始化设备，创建队列对，并最终发送I / O.

**中断**

SPDK轮询设备以进行完成，而不是等待中断。这样做有很多原因：

1）实际上，在用户空间进程中将中断路由到处理程序对于大多数硬件设计来说是不可行的;

2）中断引入软件抖动并且由于强制上下文而具有显着的开销开关。SPDK中的操作几乎是普遍异步的，允许用户在完成时提供回调。

调用回调以响应用户调用函数以轮询完成。轮询NVMe设备很快，因为只需要读取主机内存（无MMIO）来检查队列对的位翻转，而英特尔的[DDIO](https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html)等技术将确保更新后检查的主机内存存在于CPU缓存中通过设备。

**线程**

NVMe设备公开了多个队列，用于向硬件提交请求。可以在没有协调的情况下访问单独的队列，因此软件可以从多个并行执行线程向设备发送请求而无需锁定。遗憾的是，内核驱动程序必须设计为处理来自操作系统或系统上各种进程中许多不同位置的I / O，并且这些进程的线程拓扑会随时间而变化。大多数内核驱动程序选择将硬件队列映射到CPU的cores（尽可能接近1：1），然后在提交请求时，它们会查找当前线程正在运行的任何核心的正确硬件队列。通常，他们会需要获取队列周围的锁或临时禁用中断以防止在同一核上运行的线程抢占，这个开销可能很大。这是对旧硬件接口的一个很大的改进，它只有一个队列或根本没有队列，但仍然不是最佳的。

另一方面，用户空间驱动程序嵌入到单个应用程序中。此应用程序确切知道存在多少个线程（或进程），因为应用程序创建了它们。因此，SPDK驱动程序选择将硬件队列直接暴露给应用程序，并要求一次只能从一个线程访问硬件队列。实际上，应用程序为每个线程分配一个硬件队列（而不是内核驱动程序中每个核心一个硬件队列）。这保证了线程可以提交请求，而不必与系统中的其他线程执行任何类型的协调（即锁定）。

## 2.2. 来自用户空间的DMA\*\*

以下是尝试解释为什么必须使用[spdk\_dma\_malloc()](https://spdk.io/doc/env_8h.html#a0874731c44ac31e4b14d91c6844a87d1)或其兄弟[节点](https://spdk.io/doc/env_8h.html#a0874731c44ac31e4b14d91c6844a87d1)分配传递给SPDK的所有数据缓冲区，以及为什么SPDK依赖DPDK经过验证的基本功能来实现内存管理。

计算平台通常将物理内存划分为称为页面的4KiB段。它们从可寻址存储器的开头开始对从0到N的页面进行编号。然后，操作系统使用任意复杂的映射在这些物理页面上覆盖4KiB虚拟内存页面。有关概述，请参阅[虚拟内存](https://en.wikipedia.org/wiki/Virtual_memory)。

物理内存连接在通道上，每个内存通道提供一定量的带宽。为了优化总存储器带宽，通常将物理寻址设置为在信道之间自动交错。例如，页面0可以位于通道0，通道1上的页面1，通道2上的页面2，等等。这样，顺序地自动写入存储器可以自动利用所有可用的通道。实际上，交错是在比整页更细粒度的级别上完成的。

现代计算平台支持其内存管理单元（MMU）内的虚拟到物理转换的硬件加速。MMU通常支持多种不同的页面大小。在最近的x86\_64系统上，支持4KiB，2MiB和1GiB页面。通常，操作系统默认使用4KiB页面。

NVMe设备使用直接内存访问（DMA），将数据传输到系统内存，或从系统内存传输数据。具体而言，它们通过PCI总线发送请求数据传输的消息。在没有IOMMU的情况下，这些消息包含物理内存地址。这些数据传输在不涉及CPU的情况下发生，并且MMU负责访问存储器的连贯性。

NVMe设备还可能对这些传输的内存物理布局提出额外要求。NVMe 1.0规范要求所有物理内存都可以通过所谓的PRP列表进行描述。要由PRP列表描述，内存必须具有以下属性：

1.内存分为物理4KiB页面，我们称之为设备页面。  
2.第一个设备页面可以是从任何4字节对齐地址开始的部分页面。它可能会延伸到当前物理页面的末尾，但不会超出。  
3.如果有多个设备页面，则第一个设备页面必须以物理4KiB页面边界结束。  
4.最后一个设备页面从物理4KiB页面边界开始，但不需要在物理4KiB页面边界上结束。

该规范允许设备页面的大小不是4KiB，但是本文中所有已知设备都使用4KiB。

NVMe 1.1规范增加了对完全灵活的分散收集列表的支持，但该功能是可选的，现在大多数设备都不支持它。

用户空间驱动程序在常规进程的上下文中运行，因此可以访问虚拟内存。为了用物理地址正确编程设备，必须实现一些地址转换方法。

在Linux上执行此操作的最简单方法是/proc/self/pagemap从进程内进行检查。此文件包含物理地址映射的虚拟地址。从Linux 4.0开始，访问这些映射需要root权限。但是，操作系统绝对不能保证虚拟页面到物理页面的映射是静态的。操作系统无法看到PCI设备是否直接将数据传输到一组物理地址，因此必须非常小心地协调DMA请求和页面移动。当操作系统标记页面使得无法修改虚拟到物理地址映射时，这称为固定页面。

虚拟到物理映射也可能发生变化的原因有多种。到目前为止，最常见的原因是页面交换到磁盘。但是，操作系统还会在压缩的过程中移动页面，这会将相同的虚拟页面折叠到同一物理页面上以节省内存。某些操作系统也能够进行透明的内存压缩。也可以越来越多地热添加额外的内存，这可能会触发物理地址重新平衡以优化交错。

POSIX提供mlock强制虚拟内存页面始终由物理页面支持的调用。实际上，这是禁用交换。但是，这并不能保证虚拟到物理地址映射是静态的。该mlock调用不应与pin调用混淆，事实证明POSIX没有定义用于固定内存的API。因此，分配固定内存的机制是特定于操作系统的。

SPDK依靠DPDK来分配固定内存。在Linux上，DPDK通过分配hugepages（默认情况下为2MiB）来完成此操作。Linux内核处理大页面的方式与常规4KiB页面不同。具体来说，操作系统永远不会改变其物理位置。这不是故意的，因此在未来的版本中可能会发生变化，但今天确实如此并且已经存在多年（请参阅IOMMU后面的部分以获得面向未来的解决方案）。

有了这个解释，希望现在很清楚为什么必须使用[spdk\_dma\_malloc()](https://spdk.io/doc/env_8h.html#a0874731c44ac31e4b14d91c6844a87d1)或它的兄弟分配所有传递给SPDK的数据缓冲区。必须专门分配缓冲区，以便固定它们，以便知道物理地址。

**IOMMU支持**

许多平台包含一个称为I / O内存管理单元（IOMMU）的额外硬件。IOMMU很像常规MMU，除了它为外围设备（即PCI总线）提供虚拟化地址空间。MMU了解系统上每个进程的虚拟到物理映射，因此IOMMU将特定设备与这些映射之一相关联，然后允许用户分配任意总线地址到他们的过程中的虚拟地址。然后，通过将总线地址转换为虚拟地址，然后将虚拟地址转换为物理地址，PCI设备和系统存储器之间的所有DMA操作都通过IOMMU进行转换。这允许操作系统自由地修改虚拟到物理地址映射，而不会破坏正在进行的DMA操作。Linux提供了一个设备驱动程序，vfio-pci允许用户使用当前进程配置IOMMU。

这是一种面向未来的硬件加速解决方案，用于执行进出用户空间进程的DMA操作，并构成SPDK和DPDK内存管理策略的长期基础。我们强烈建议使用vfio和启用的IOMMU部署应用程序，这一点现在已得到全面支持。

## 2.3. 消息传递和并发\*\*

**原理**

SPDK的主要目标之一是通过添加硬件进行线性扩展。这可能意味着实践中的许多事情。例如，从一个SSD移动到两个SSD应该是每秒I / O数量的两倍。或者将CPU核心数量增加一倍应该可以使计算量增加一倍。或者甚至将NIC数量加倍也应该使网络吞吐量翻倍。为实现此目的，必须设计软件使得执行线程尽可能彼此独立。在实践中，这意味着避免软件锁甚至原子指令。

传统上，软件通过将一些共享数据放到堆上，用锁保护它，然后让所有执行线程仅在需要访问共享数据时获得锁来实现并发。这个模型有很多很棒的属性：

* 将单线程程序转换为多线程程序相对容易，因为您不必从单线程版本更改数据模型。您只需在数据周围添加锁定即可。
* 您可以将程序编写为从上到下阅读的同步，命令性语句列表。
* 您的线程可以在后台由操作系统调度程序中断并进入休眠状态，从而实现CPU资源的高效时间共享。

不幸的是，随着线程数量的增加，对共享数据周围锁定的争用也会发生。更细粒度的锁定有所帮助，但也大大增加了程序的复杂性。即使这样，超过一定数量的高度争用的锁，线程将花费大部分时间来尝试获取锁，并且程序将不会受益于任何额外的CPU核。

SPDK完全采用不同的方法。SPDK通常会将该数据分配给单个线程，而不是将共享数据放在所有线程获取锁定后访问的全局位置。当其他线程想要访问数据时，它们会向拥有的线程传递一条消息，以代表它们执行操作。当然，这种策略并不是什么新鲜事。例如，它是[Erlang](http://erlang.org/download/armstrong_thesis_2003.pdf)的核心设计原则之一，也是[Go中](https://tour.golang.org/concurrency/2)的主要并发机制。SPDK中的消息通常由函数指针和指向某个上下文的指针组成，并使用[无锁环](http://dpdk.org/doc/guides/prog_guide/ring_lib.html)在线程之间传递。消息传递通常比大多数软件开发人员的直觉导致他们相信的速度快得多，这主要是由于缓存效应。如果单个核心始终访问相同的数据（代表所有其他核心），则该数据更可能位于更靠近该核心的缓存中。通常最有效的方法是让每个核心工作在一个相对较小的数据集中，然后将其放在本地缓存中，然后在完成后将一条小消息传递给下一个核心。

在更极端的情况下，即使是消息传递也可能成本过高，将为每个线程创建一个数据副本。然后该线程将只引用其本地副本。为了改变数据，线程将向每个其他线程发送一条消息，告诉它们在本地副本上执行更新。当数据不经常变化，但可能经常被读取，并且经常被用于I/O路径时，这是非常理想的。这当然是为了计算效率而交换内存大小，因此它的使用仅限于最关键的代码路径。

**消息传递基础结构**

SPDK提供了多层消息传递基础结构。例如，SPDK中最基本的库不会自行传递任何消息，而是枚举关于何时可以在其文档中调用函数的规则（例如[NVMe驱动程序](https://spdk.io/doc/nvme.html)）。但是，大多数库依赖于SPDK的[线程](http://www.spdk.io/doc/thread_8h.html)抽象，位于libspdk\_thread.a。线程抽象提供了一个基本的消息传递框架，并定义了一些关键元素。

首先，spdk\_thread是执行线程的抽象，spdk\_poller是一个应该在给定线程上定期调用的函数的抽象。

在用户希望与SPDK一起使用的每个系统线程上，它们必须首先调用spdk\_thread\_create（）。

该库还定义了另外两个抽象：spdk\_io\_device和[spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html)。在实施SPDK的过程中，我们注意到在许多不同的库中出现了相同的模式。为了实现消息传递策略，代码将描述具有全局状态的一些对象以及与在I / O路径中访问的该对象相关联的一些每线程上下文，以避免锁定全局状态。在I / O被提交给块设备的最低层中，这种模式最清晰。这些设备通常会公开多个可以分配给线程的队列，然后在没有锁的情况下进行访问以提交I / O。

抽象地说，我们将设备概括为spdk\_io\_device，将特定于线程的队列[概括](https://spdk.io/doc/structspdk__io__channel.html)为[spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html)。然而，随着时间的推移，这种模式出现在很多地方，与我们最初选择的名字不太匹配。在今天的代码中，spdk\_io\_device是任何指针，其唯一性仅取决于其内存地址，[spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html)是与特定spdk\_io\_device相关联的每线程上下文。

线程抽象提供了将消息发送到任何其他线程、逐个向所有线程发送消息、以及向那些拥有io\_channel的给定io\_device的所有线程发送消息的功能。

**事件框架**

随着SPDK中示例应用程序数量的增长，很明显每个代码中的大部分代码都实现了调用spdk\_thread\_create（）所需的基本消息传递基础结构。这包括为每个核心生成一个线程，将每个线程固定到一个唯一的核心，并在线程之间分配无锁环以传递消息。

SPDK不是为每个示例应用程序重新实现该基础结构，而是提供SPDK [事件框架](https://spdk.io/doc/event.html)。

该库处理：设置所有消息传递基础结构、安装信号处理程序来干净地关闭、实现定期轮询，以及执行基本命令行解析。通过[spdk\_app\_start（）](https://spdk.io/doc/event_8h.html#ab4c22e2920f70becd9b2ef752efd7975)启动时，库会自动生成所有请求的线程，将它们固定，然后使用适当的函数指针调用spdk\_thread\_create spdk\_allocate\_thread（）。这使得实现全新的SPDK应用程序变得更加容易，并且是那些刚开始使用的推荐方法。只有具有足够消息传递基础结构的已建立应用程序才应考虑直接集成低级库。

**C语言的局限性**

消息传递是有效的，但它会导致异步代码。不幸的是，异步代码在C中是一个挑战。它通常通过传递操作完成时调用的函数指针来实现。这会削减代码，因此不容易遵循，特别是通过逻辑分支。最好的解决方案是使用支持futures和promises的语言（注：在[计算机科学中](https://en.wikipedia.org/wiki/Computer_science)，**future**, **promise**, **delay**和**deferred**是指用于在一些[并发编程语言中](https://en.wikipedia.org/wiki/Concurrent_programming_language)[同步](https://en.wikipedia.org/wiki/Synchronization_(computer_science))程序[执行的](https://en.wikipedia.org/wiki/Execution_(computing))构造），例如C ++，Rust，Go或几乎任何其他更高级别的语言。但是，SPDK是一个低级库，需要非常广泛的兼容性和可移植性，所以我们选择留在普通的旧C中。

不过，我们确实有一些建议要分享。对于*简单的*回调链，如果从下到上编写函数，这是最简单的。我们的意思是，如果函数foo执行一些异步操作，并且当bar调用完成函数时，函数bar执行一些baz在完成时调用函数的操作，写它的好方法是这样的：

void baz(void \*ctx) {

...

}

void bar(void \*ctx) {

async\_op(baz, ctx);

}

void foo(void \*ctx) {

async\_op(bar, ctx);

}

不要拆分这些功能 - 将它们保存为一个可以从下到上阅读的好单元。

对于更复杂的回调链，尤其是具有逻辑分支或循环的回调链，最好写出状态机。事实证明，支持未来和承诺的更高级语言只是在编译时生成状态机，所以即使我们没有能力在C中生成它们，我们仍然可以手动编写它们。举个例子，这里有一个回调链，它执行5次foo，然后调用bar——实际上是异步for循环。

enum states {

FOO\_START = 0,

FOO\_END,

BAR\_START,

BAR\_END

};

struct state\_machine {

enum states state;

int count;

};

static void

foo\_complete(void \*ctx)

{

struct state\_machine \*sm = ctx;

sm->state = FOO\_END;

run\_state\_machine(sm);

}

static void

foo(struct state\_machine \*sm)

{

do\_async\_op(foo\_complete, sm);

}

static void

bar\_complete(void \*ctx)

{

struct state\_machine \*sm = ctx;

sm->state = BAR\_END;

run\_state\_machine(sm);

}

static void

bar(struct state\_machine \*sm)

{

do\_async\_op(bar\_complete, sm);

}

static void

run\_state\_machine(struct state\_machine \*sm)

{

enum states prev\_state;

do {

prev\_state = sm->state;

switch (sm->state) {

case FOO\_START:

foo(sm);

break;

case FOO\_END:

/\* This is the loop condition \*/

if (sm->count++ < 5) {

sm->state = FOO\_START;

} else {

sm->state = BAR\_START;

}

break;

case BAR\_START:

bar(sm);

break;

case BAR\_END:

break;

}

} while (prev\_state != sm->state);

}

void do\_async\_for(void)

{

struct state\_machine \*sm;

sm = malloc(sizeof(\*sm));

sm->state = FOO\_START;

sm->count = 0;

run\_state\_machine(sm);

}

当然，这很复杂，但是run\_state\_machine可以从上到下阅读该函数，以便清楚地了解代码中发生的情况，而无需追逐每个回调。

## 2.4. NAND Flash SSD内部

固态设备（SSD）是复杂的设备，其性能取决于它们的使用方式。以下描述旨在帮助软件开发人员了解SSD内部发生的情况，以便他们能够提供更好的软件设计。它不应被视为SSD硬件真正工作的严格准确指南。

在撰写本文时，SSD通常在[NAND闪存](https://en.wikipedia.org/wiki/Flash_memory)之上实现。在很高的层面上，这种媒体有一些重要的属性：

将介质分组到称为NAND管芯的芯片上，并且每个管芯可以并行操作。

翻转是一个高度不对称的过程。以一种方式翻转它很容易，但是将其翻转是非常困难的。

NAND闪存介质被分组为通常称为擦除块的大单元。擦除块的大小是高度特定于实现的，但可以认为介于1MiB和8MiB之间。对于每个擦除块，可以用位粒度一次写入每个位（即，其位从0翻转到1）。为了第二次写入擦除块，必须擦除整个块（即块中的所有位都被翻转回0）。这是上面的不对称部分。擦除块会导致可测量的磨损量，并且每个块可能仅被擦除有限次数。

SSD向主机系统公开接口，使其看起来好像驱动器由一组固定大小的逻辑块组成，这些逻辑块的大小通常为512B或4KiB。这些块完全是设备固件的逻辑结构，并且它们不会静态映射到后备介质上的位置。相反，在每次写入逻辑块时，选择并写入NAND闪存上的新位置，并更新逻辑块到其物理位置的映射。选择此位置的算法是整体SSD性能的关键部分，通常称为闪存转换层（FTL）。该算法必须正确分配块以考虑磨损（称为磨损均衡）并将它们分布在NAND裸片上以提高总体可用性能。最简单的模型是使用类似于RAID的算法将每个芯片上的所有物理介质分组在一起，然后按顺序写入该集合。真正的SSD要复杂得多，但对于软件开发人员来说这是一个非常好的简单模型 - 想象他们只是记录到RAID卷并更新内存中的散列表。

闪存转换层的一个结果是逻辑块不一定总是对应于NAND上的物理位置。实际上，有一个命令可以清除块的转换。在NVMe中，此命令称为deallocate，在SCSI中称为unmap，在SATA中称为trim。当用户尝试读取没有映射到物理位置的块时，驱动器将执行以下两项操作之一：

1. 立即成功完成读取请求，而不执行任何数据传输。这是可以接受的，因为驱动器将返回的数据不再比用户数据缓冲区中已有的数据有效。
2. 返回全0作为数据。

选择＃1更为常见，对完全解除分配的设备执行读取操作通常会显示远远超出驱动器声称的能力，因为它实际上并没有传输任何数据。在进行基准测试时，在读取之前写入所有块！

在写入SSD时，内部日志最终将消耗所有可用的擦除块。为了继续写入，SSD必须释放其中一些。此过程通常称为垃圾回收。所有SSD都保留了一些擦除块，以便它们可以保证有可用于垃圾收集的空闲擦除块。垃圾收集通常由以下方式进行：

1. 选择一个目标擦除块（一个好的心理模型是它选择最近最少使用的擦除块）
2. 遍历擦除块中的每个条目并确定它是否仍然是有效的逻辑块。
3. 通过读取它们并将它们写入不同的擦除块（即日志的当前头部）来移动有效的逻辑块
4. 擦除整个擦除块并标记它可供使用。

由于擦除块已经为空，因此当跳过步骤＃3时，垃圾收集显然更有效。有两种方法可以使步骤＃3更有可能被跳过。首先，SSD会保留超出其报告容量的额外擦除块（称为过度供应），因此从统计上来说，擦除块不太可能包含有效数据。第二种是软件可以按顺序以圆形模式写入设备上的块，不再需要时丢弃旧数据。在这种情况下，软件保证最近最少使用的擦除块不包含必须移动的任何有效数据。

如果工作负载填满整个设备，则设备过度配置的数量会显着影响随机读写工作负载的性能。然而，通常可以通过简单地在软件中保留设备上的给定量的空间来获得相同的效果。这种理解对于产生一致的基准是至关重要的。特别是，如果后台垃圾收集无法跟上，并且驱动器必须切换到按需垃圾收集，则写入的延迟将显着增加。因此，在运行基准以保持一致性之前，必须强制设备的内部状态进入某种已知状态。这通常是通过从开始到结束顺序写入设备两次来完成的。有关如何强制SSD进入已知状态进行基准测试的详细说明，请参阅[SNIA文章](http://www.snia.org/sites/default/files/SSS_PTS_Enterprise_v1.1.pdf)。

## 2.5. 将I / O提交到NVMe设备\*\*

### 2.5.1 NVMe规范

NVMe规范描述了用于与存储设备交互的硬件接口。该规范包括远程存储的网络传输定义以及本地PCIe设备的硬件寄存器布局。以下是对如何通过SPDK将I / O提交到本地PCIe设备的概述。

NVMe设备允许主机软件（在我们的例子中，SPDK NVMe驱动程序）在主机内存中分配队列对。术语“主机”经常使用，因此要明确这是NVMe SSD插入的系统。队列对由两个队列组成 - 提交队列和完成队列。这些队列更准确地描述为固定大小条目的圆环。提交队列是一个64字节命令结构的数组，加上2个整数（头部和尾部索引）。完成队列类似于16字节完成结构的数组，加上2个整数（头部和尾部索引）。还涉及两个32位寄存器，称为门铃。

通过构造64字节命令将I / O提交到NVMe设备，将其放入提交队列头索引的当前位置的提交队列中，然后将提交队列头的新索引写入提交队列头门铃登记。将一整套命令复制到环中的开放槽中，然后只需一次写门铃就可以提交整个批次，这实际上是有效的。

NVMe规范中的命令提交和完成过程有一个非常详细的描述，可以从[NVM Express的](https://nvmexpress.org/)主页面上方便地获得。

最重要的是，命令本身描述了操作，并且如果需要，还描述了主机存储器中的位置，该位置包含与命令相关联的主机存储器的描述符。该主机存储器是要写入写入命令的数据，或者是将数据置于读取命令上的位置。使用NVMe设备上的DMA引擎将数据传输到此位置或从此位置传输数据。

完成队列的工作方式类似，但设备是将条目写入环中的设备。每个条目都包含一个“相位”位，通过整个环在每个循环上切换0到1。当队列对设置为生成中断时，中断包含完成队列头的索引。但是，SPDK不会启用中断，而是轮询相位以检测完成。中断是非常繁重的操作，因此轮询此相位通常要高效得多。

### 2.5.2 SPDK NVMe驱动程序I / O路径

现在我们知道环结构的工作原理，让我们来介绍SPDK NVMe驱动程序如何使用它们。用户将在程序的生命周期中的某个早期时间构建队列对，因此这不是“热”路径的一部分。然后，他们将调用[spdk\_nvme\_ns\_cmd\_read（）之](https://spdk.io/doc/nvme_8h.html#a084c6ecb53bd810fbb5051100b79bec5)类的函数来执行I / O操作。用户提供数据缓冲区，目标LBA和长度，以及命令所针对的NVMe命名空间和要使用的NVMe队列对等其他信息。最后，用户提供了一个回调函数和上下文指针，在稍后调用[spdk\_nvme\_qpair\_process\_completions（）](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782)期间发现结果命令的完成时将调用该函数和上下文指针。

驱动程序的第一个阶段是分配一个请求对象来跟踪操作。这些操作是异步的，因此它不能简单地跟踪调用堆栈上的请求状态。在堆上分配新的请求对象会太慢，因此SPDK会在NVMe队列对对象内部保留一组预先分配的请求对象——struct spdk\_nvme\_qpair。分配给队列对的请求数大于NVMe提交队列的实际队列深度，因为SPDK支持几个关键的便利功能。第一个是软件排队 - SPDK将允许用户提交比硬件队列实际可以容纳的更多请求，SPDK将自动在软件中排队。第二是拆分。SPDK将出于多种原因拆分请求，其中一些原因将在下面列出。请求对象的数量可在队列对创建时配置，如果未指定，SPDK将根据硬件队列深度选择合理的数字。

第二阶段是构建64字节NVMe命令本身。该命令内置于嵌入请求对象的内存中——而不是直接嵌入到NVMe提交队列槽中。构建命令后，SPDK会尝试在NVMe提交队列中获取一个打开的插槽。对于提交队列中的每个元素，将分配一个称为跟踪器的对象。跟踪器以数组形式分配，因此可以通过索引快速查找它们。跟踪器本身包含指向当前占用该槽的请求的指针。获取特定跟踪器时，将使用跟踪器的索引更新命令的CID值。NVMe规范在完成时提供CID值，因此可以通过CID值查找跟踪器然后跟随指针来恢复请求。

一旦获得跟踪器（槽），就处理与其关联的数据缓冲区以构建PRP列表。这本质上是一个NVMe分散收集列表（SGL），虽然它有点受限制。用户向SPDK提供缓冲区的虚拟地址，因此SPDK必须查找页表以查找支持该虚拟内存的物理地址（pa）或I / O虚拟地址（iova）。虚拟连续的存储器区域可能不是物理上连续的，因此这可能导致具有多个元件的PRP列表。有时，这可能会导致一组实际上无法表示为单个PRP列表的物理地址，因此SPDK会自动将用户操作分成两个单独的请求。有关如何管理内存的更多信息，请参阅[从用户空间直接内存访问（DMA）](https://spdk.io/doc/memory.html)。

之所以在获得跟踪器之前不构建PRP列表，是因为PRP列表的描述必须在DMA-able内存中分配，并且可能非常大。由于SPDK通常会分配大量请求，因此我们不希望分配足够的空间来预构建最坏情况的场景PRP列表，特别是在一般情况下根本不需要单独的PRP列表的情况下。

每个NVMe命令都嵌入了两个PRP列表元素，因此如果请求是4KiB（或者如果它是8KiB且完全对齐），则不需要单独的PRP列表。分析显示，代码的这一部分并不是CPU总体使用的主要贡献者。

在跟踪器填写完毕后，SPDK将64字节命令复制到实际的NVMe提交队列插槽中，然后按响提交队列尾门铃，告诉设备进行处理。然后SPDK返回给用户，而不等待完成。

用户可以定期调用[spdk\_nvme\_qpair\_process\_completions（）](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782)告诉SPDK检查完成队列。具体来说，它读取下一个预期完成时隙的相位位，当它翻转时，查看CID值以找到指向请求对象的跟踪器。请求对象包含用户最初提供的函数指针，然后调用该函数指针以完成命令。

该[spdk\_nvme\_qpair\_process\_completions()](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782)函数将继续前进到下一个完成时段，直到完成时段结束，此时它将写入完成队列头门铃，让设备知道它可以使用完成队列时段进行新的完成并返回。

## 2.6. 使用Vhost-user进行虚拟化I / O.

### 2.6.1 介绍

本文档旨在概述Vhost如何在幕后工作。为了便于阅读，本文档中使用的代码片段可能已经过简化，不应将其用作API或实现参考。

阅读[Virtio规范](http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html)：

virtio和[virtio]规范的目的，是虚拟环境和guest虚拟机应该为虚拟设备提供简单、高效、标准和可扩展的机制，而不是精确的每个环境或每个操作系统机制。

Virtio设备使用virtqueues有效地传输数据。Virtqueue是一组三种不同的单生产者，单消费者环形结构，用于存储通用的分散-集合I / O。 Virtio最常用于QEMU VMs，其中QEMU本身公开虚拟PCI设备，客户操作系统使用特定的Virtio PCI驱动程序与其通信。由于只涉及Virtio，QEMU进程始终处理所有I / O进程。

Vhost是可通过进程间通信访问的设备的协议。它使用与Virtio相同的virtqueue布局，以允许Vhost设备直接映射到Virtio设备。这允许由SPDK应用程序公开的Vhost设备由QEMU进程内的客户操作系统直接访问，并具有现有的Virtio（PCI）驱动程序。只有配置，I / O提交通知和I / O完成中断通过QEMU进行管道传输。另请参见[SPDK优化](https://spdk.io/doc/vhost_processing.html#vhost_spdk_optimizations)。

最初的vhost实现是Linux内核的一部分，使用ioctl接口与用户空间应用程序进行通信。使SPDK能够公开vhost设备的，是vhost用户协议。

所述[虚拟主机用户规范](https://git.qemu.org/?p=qemu.git;a=blob_plain;f=docs/interop/vhost-user.txt;hb=HEAD)描述了协议如下：

[Vhost-user协议]旨在补充用于控制Linux内核中的vhost实现的ioctl接口。 它实现了在同一主机上与用户空间进程建立virtqueue共享所需的控制平面。 它使用Unix域套接字上的通信来共享消息的辅助数据中的文件描述符。

该协议定义了通信的两个方面，主从和从属。 在我们的案例QEMU中，Master是分享其优点的应用程序。 奴隶是美德的消费者。

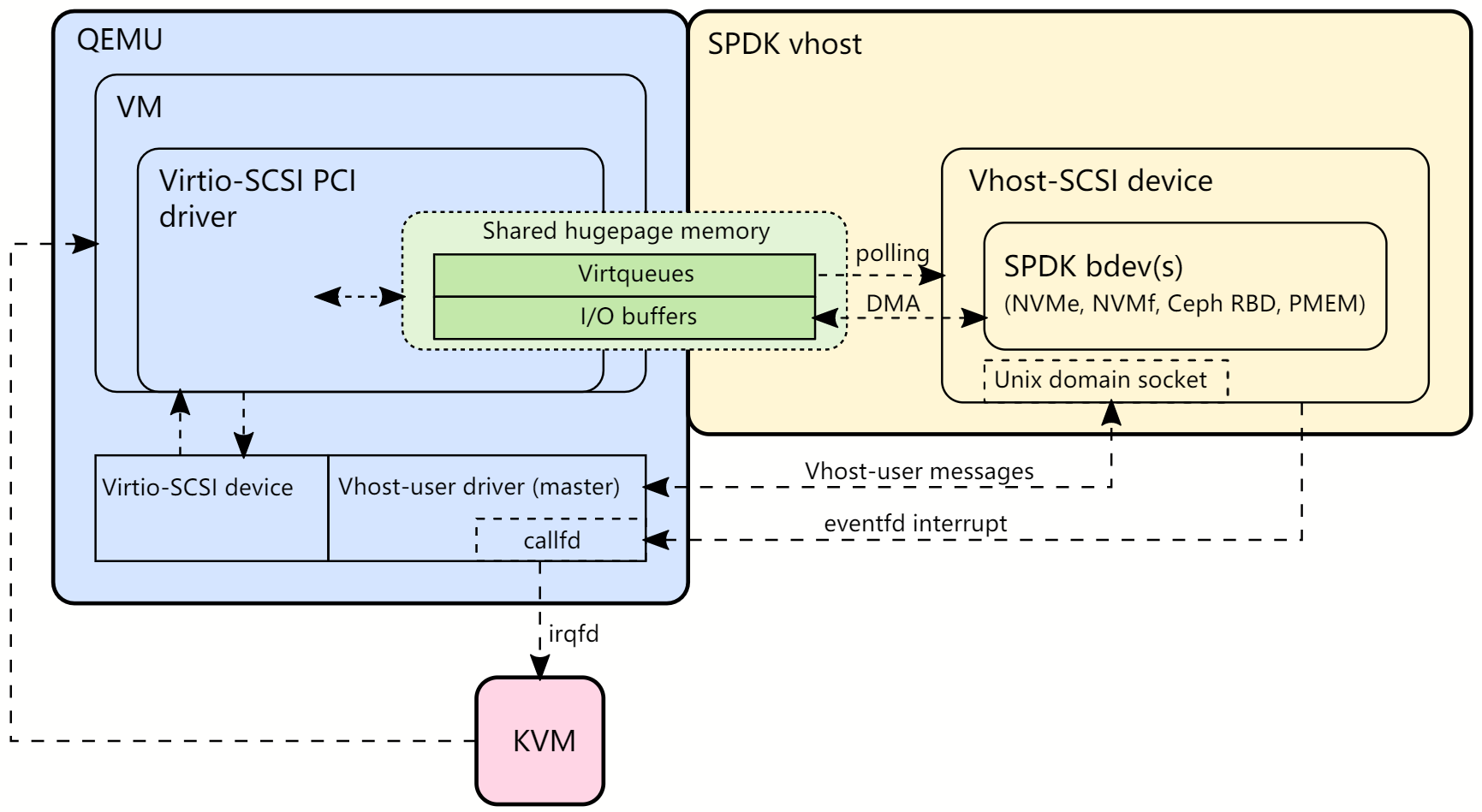
在当前的实现中，QEMU是Master，Slave是用户空间中运行的软件以太网交换机，例如Snabbswitch。

主设备和从设备可以是套接字通信中的客户端（即连接）或服务器（监听）。

SPDK vhost是Vhost用户的slave服务器。它公开了Unix域套接字（domain sockets）并允许外部应用程序连接。

### 2.6.2 QEMU

主要的Vhost用户用例之一是QEMU中的网络（DPDK）或存储（SPDK）卸载。下图显示了基于QEMU的VM如何与SPDK Vhost-SCSI设备通信。



QEMU / SPDK vhost数据流

### 2.6.3 设备初始化

使用Vhost用户消息交换所有初始化和管理信息。连接始终以功能协商开始。Master和Slave都公开了他们实现的功能列表，并且在协商时他们选择了一组共同​​的功能。这些功能中的大多数都与实现相关，但也考虑了多队列支持或实时迁移。

协商后，Vhost用户驱动程序共享其内存，以便vhost设备（SPDK）可以直接访问它。内存可以分段为多个物理上不连续的区域，Vhost用户规范对其数量进行限制 - 当前为8.驱动程序使用以下数据为每个区域发送单个消息：

* 文件描述符 - 用于mmap
* 用户地址 - 用于Vhost用户消息中的内存转换（例如，转换vring地址）
* 访客地址 - 缓冲区地址vrings中的翻译（对于QEMU，这是访客内部的物理地址）
* 用户偏移量 - mmap的正偏移量
* 大小

每次更换内存后，Master将发送新的内存区域 - 通常是hotplug / hotremove。之前的映射将被删除。

驱动程序也可以请求设备配置，包括例如磁盘几何结构。但是，Vhost-SCSI驱动程序不需要实现此功能，因为它们使用通用SCSI I / O来查询底层磁盘。

然后，驱动程序请求最大支持队列的数量并开始发送virtqueue数据，其中包括：

* 独特的虚拟队列
* 最后处理的vring描述符的索引
* vring地址（来自用户地址空间）
* call描述符（用于在I / O完成后中断驱动程序）
* kick描述符（监听I / O请求 - SPDK未使用）

如果已协商多队列功能，则驱动程序必须为要轮询的每个额外队列发送特定的ENABLE消息。其他队列在初始化后立即进行轮询。

### 2.6.4 I / O路径

Master通过在共享内存中分配适当的缓冲区，填充请求数据，并将这些缓冲区的访客地址放入virtqueue来发送I / O。

Virtio-Block请求如下所示。

struct virtio\_blk\_req {

uint32\_t type; // READ，WRITE，FLUSH（只读）

uint64\_t offset; //磁盘中的偏移量（只读）

struct iovec buffers[]; // scatter-gatter list（读/写）

uint8\_t status; // I / O完成状态（只写）

};

并且Virtio-SCSI请求如下。

struct virtio\_scsi\_req\_cmd {

struct virtio\_scsi\_cmd\_req \* req; //请求数据（只读）

struct iovec read\_only\_buffers[]; // WRITE I / O的scatter-Gatter列表

struct virtio\_scsi\_cmd\_resp \* resp; //响应数据（只写）

struct iovec write\_only\_buffers[]; // READ I / O的scatter-Gatter列表

}

Virtqueue通常由一系列描述符组成，每个I / O需要转换为这样的描述符链。单个描述符可以是可读的或可写的，因此每个I / O请求至少包含两个（请求+响应）。

struct virtq\_desc {

le64 addr; / \*地址（客户 - 物理的）。\* /

le32 len; /\* 长度。\* /

/ \*这标志缓冲区继续通过下一个字段。\* /

#define VIRTQ\_DESC\_F\_NEXT 1

/ \*这将缓冲区标记为设备只写（否则设备为只读）。\* /

#define VIRTQ\_DESC\_F\_WRITE 2

le16 flags; / \*如上所示的标志。\* /

le16 next; / \*下一个字段if flags＆NEXT \* /

};

Legacy Virtio实现使用名称vring和virtqueue，名称vring仍然在代码中的virtio数据结构中使用。而不是struct virtq\_desc，struct vring\_desc更有可能找到。

轮询此描述符链后，设备需要转换并将其转换回原始请求结构。它需要预先知道请求布局，因此每个设备后端（Vhost-Block / SCSI）都有自己的轮询virtqueues实现。对于每个描述符，设备在Vhost用户内存区域表中执行查找，并进行gpa\_to\_vva转换（客户物理地址到vhost虚拟地址）。SPDK强制请求和响应数据包含在单个内存区域中。I / O缓冲区没有这样的限制，如果需要，SPDK可以自动执行额外的iovec拆分和gpa\_to\_vva转换。在形成请求结构后，SPDK将此类I / O转发到底层驱动器并轮询完成。一旦I / O完成，SPDK vhost使用适当的数据填充响应缓冲区，并通过在调用描述符上执行eventfd\_write来中断guest虚拟机以获得正确的virtqueue。涉及多个中断合并功能，但本文档不讨论它们。

### 2.6.5 SPDK优化

由于其轮询模式特性，SPDK vhost消除了对I / O提交通知的要求，大大增加了vhost服务器吞吐量并降低了提交I / O的访客开销。存在几种不同的解决方案来缓解I / O完成中断开销（irqfd，vDPA），但本文档不讨论这些解决方案。为了获得最高性能，可以使用轮询模式[Virtio驱动程序](https://spdk.io/doc/virtio.html)，因为它可以抑制所有I / O完成中断，从而使I / O路径完全绕过QEMU / KVM开销。

## 2.7. SPDK目录结构概述

**概观**

SPDK由一组存放在lib公共接口头文件中的C库组成include/spdk，另外还有一组由这些库构建的应用程序app。用户可以在其软件中使用C库或部署完整的SPDK应用程序。

SPDK是围绕消息传递而不是锁定而设计的，并且大多数SPDK库对它们嵌入的应用程序的底层线程模型做出了若干假设。然而，SPDK竭尽全力地保持对实际使用的特定消息传递、事件、协同例程或轻量级线程框架的不可知性。为此，所有SPDK库都与lib/thread（public interface at [include/spdk/thread.h](https://spdk.io/doc/thread_8h.html)）中的抽象库进行交互。任何框架都可以初始化线程抽象并提供回调以实现SPDK库所需的功能。有关此抽象的更多信息，请参阅[消息传递和并发](https://spdk.io/doc/concurrency.html)。

SPDK建立在POSIX之上，适用于大多数操作。为了更容易地移植到非POSIX环境，所有POSIX标头都被隔离到[include/spdk/stdinc.h](https://spdk.io/doc/stdinc_8h.html)中。但是，SPDK需要POSIX不提供的许多操作，例如枚举系统上的PCI设备或分配对DMA安全的内存。这些附加操作都是在名为env的库中抽象出来的，该库的公共头位于include/spdk/env.h。默认情况下，spdk使用基于dpdk的库来实现env接口。但是，可以替换该实现。有关其他信息，请参阅[SPDK移植指南](https://spdk.io/doc/porting.html)。

**Applications**

app顶级目录包含全面的应用程序，内置了SPDK组件。有关完整概述，请参阅[SPDK应用程序概述](https://spdk.io/doc/app_overview.html)。

通常可以使用少量配置选项启动SPDK应用程序。然后使用JSON-RPC执行应用程序的完整配置。有关其他信息，请参阅[JSON-RPC方法](https://spdk.io/doc/jsonrpc.html)。

**Libraries**

lib目录包含SPDK的真正核心。每个组件都是一个C库，其下有自己的目录lib。一些关键库是：

* [块设备用户指南](https://spdk.io/doc/bdev.html)
* [NVMe驱动程序](https://spdk.io/doc/nvme.html)

**Documentation**

doc顶级目录包含所有SPDK的文档。API文档是使用Doxygen直接从代码创建的，但更多的一般文章和更长的解释都驻留在此目录中，以及Doxygen配置文件。

要构建文档，只需在文档目录中键入make。

**Examples**

examples示例顶级目录包含一组用于参考的示例。它们与应用程序不同，应用程序正在执行一个可以合理部署的“实际”任务。相反，这些示例要么是为了演示SPDK的某些方面而精心设计的，要么被认为不够完整，无法保证将它们标记为完整的SPDK应用程序。

这是一个了解SPDK工作原理的好地方。特别是，查看examples/nvme/hello\_world。

**Include**

include目录是所有头文件所在的位置。公共API都放在spdk子目录中include，我们强烈建议应用程序将其包含路径设置为顶级include目录，并通过前缀包括标题spdk/：

#include“ [spdk / nvme.h](https://spdk.io/doc/nvme_8h.html) ”

这里的大多数标题对应于lib目录中的库。但是，有一些标题是独立的。他们是：

* [assert.h](https://spdk.io/doc/assert_8h.html)
* [barrier.h](https://spdk.io/doc/barrier_8h.html)
* [endian.h](https://spdk.io/doc/endian_8h.html)
* [fd.h](https://spdk.io/doc/fd_8h.html)
* [mmio.h](https://spdk.io/doc/mmio_8h.html)
* queue.h 和 queue\_extras.h
* [string.h](https://spdk.io/doc/string_8h.html)

还有一个spdk\_internal目录，其中包含SPDK中的库广泛包含的头文件，但它们不是公共API的一部分，不会安装在用户的系统上。

**Scripts**

scripts目录包含许多操作的便捷脚本。最重要的两个是check\_format.sh，它将使用astyle和pep8来检查我们定义的约定的C，C ++和Python编码风格，以及setup.sh，它将设备从内核驱动程序中绑定和解除。

**Tests**

test目录包含SPDK组件的所有测试，子目录镜像整个存储库的结构。测试是单元测试和功能测试的混合。

## 2.8. SPDK移植指南

通过实现env库接口，spdk被移植到新的环境中。env接口为驱动程序提供API，用于分配物理上连续的和固定的内存、执行PCI操作（配置周期和映射栏）、虚拟到物理地址转换和管理内存池。env API在include/spdk/env.h中定义。

SPDK包括基于数据平面开发工具包（[DPDK](http://dpdk.org/)）的env库的默认实现。此dpdk实现可以在lib/env\_dpdk中找到。

DPDK目前仅支持Linux和FreeBSD。想要在其他操作系统或DPDK以外的用户空间驱动程序框架中使用SPDK的用户，需要实现新版本的env库。通过更新CONFIG中的以下行，可以将新实现集成到SPDK构建中：

CONFIG\_ENV？= $（SPDK\_ROOT\_DIR）/lib/env\_dpdk

# 第三章 用户指南

## 3.1. 系统配置用户指南

本系统配置指南介绍了如何配置与SPDK配合使用的系统。

### 3.1.1 IOMMU配置

IOMMU可以在许多平台上存在并启用。当存在并启用IOMMU时，建议使用vfio-pci内核驱动程序部署SPDK应用程序。在这种情况下，SPDK的scripts/setup.sh脚本将自动选择vfio-pci。

但是，某些设备在绑定时无法正常运行vfio-pci，而必须连接到uio\_pci\_generic内核驱动程序。在这种情况下，用户应注意在运行之前禁用IOMMU或将其设置为直通模式scripts/setup.sh。

要禁用IOMMU或将其放入直通模式，请将intel\_iommu=off或amd\_iommu=off或intel\_iommu=on iommu=pt添加到x86\_64系统的GRUB命令行，或在arm64系统上添加iommu.passthrough=1。

## 3.2. SPDK应用程序概述

SPDK主要是一个开发工具包，它提供库和头文件，供其他应用程序使用。但是，SPDK还包含许多应用程序。这些应用程序主要用于测试库，但许多都功能齐全且质量高。SPDK的主要应用包括：

* iSCSI Target
* NVMe over Fabrics Target
* vhost Target
* SPDK Target（结合以上三者的统一应用）

目录中还有许多工具和示例examples。

SPDK目标都基于一个公共框架，因此它们有很多共同点。框架定义了一个称为子系统（subsystem）的概念，所有功能都在不同的子系统中实现。子系统具有统一的初始化和拆卸路径。

### 3.2.1 配置SPDK应用程序

命令行参数

SPDK应用程序框架为使用它的所有应用程序定义了一组基本命令行标志。特定应用程序可能实现其他标志

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参数 | 说明 | 类型 | 默认 | 描述 |
| -c | -config | string |  | [配置文件](https://spdk.io/doc/app_overview.html#cmd_arg_config_file) |
| -d | -limit-coredump | flag | false | [限制coredump](https://spdk.io/doc/app_overview.html#cmd_arg_limit_coredump) |
| -e | -tpoint-group-mask | integer | 0x0 | [Tracepoint组掩码](https://spdk.io/doc/app_overview.html#cmd_arg_limit_tpoint_group_mask) |
| -g | -single-file-segments | flag |  | [只创建一个hugetlbfs文件](https://spdk.io/doc/app_overview.html#cmd_arg_single_file_segments) |
| -h | -help | flag |  | 显示所有可用参数并退出 |
| -i | -shm-id | integer |  | [多进程模式](https://spdk.io/doc/app_overview.html#cmd_arg_multi_process) |
| -m | -cpumask | CPU mask | 0x1 | 应用程序[CPU掩码](https://spdk.io/doc/app_overview.html#cpu_mask) |
| -n | -mem-channels | integer | all channels | 用于DPDK的内存通道数 |
| -p | -master-core | integer | CPU掩码中的第一个核心 | DPDK的主（主）核心 |
| -r | -rpc-socket | string | /var/tmp/spdk.sock | RPC监听地址 |
| -s | -mem-size | integer | 所有hugepage 内存 | [内存大小](https://spdk.io/doc/app_overview.html#cmd_arg_memory_size) |
|  | -silence-noticelog | flag |  | 禁用通知级别日志记录 stderr |
| -u | -no-pci | flag |  | [禁用PCI访问](https://spdk.io/doc/app_overview.html#cmd_arg_disable_pci_access)。 |
|  | -wait-for-rpc |  |  | [延迟初始化](https://spdk.io/doc/app_overview.html#cmd_arg_deferred_initialization) |
| -B | -pci-blacklist | B:D:F |  | [PCI地址黑名单和白名单](https://spdk.io/doc/app_overview.html#cmd_arg_pci_blacklist_whitelist)。 |
| -W | -pci-whitelist | B:D:F |  | [PCI地址黑名单和白名单](https://spdk.io/doc/app_overview.html#cmd_arg_pci_blacklist_whitelist)。 |
| -R | -huge-unlink | flag |  | [初始化后取消链接hugepage文件](https://spdk.io/doc/app_overview.html#cmd_arg_huge_unlink) |
|  | -huge-dir | string | 第一次发现 | 从特定的挂载分配大页面 |
| -L | -logflag | string |  | [调试日志](https://spdk.io/doc/app_overview.html#cmd_arg_debug_log_flags) |

详细说明：

**配置文件**

过去，SPDK应用程序是使用配置文件配置的。这仍然受支持，但被认为已弃用，现在主张使用JSON RPC配置。有关详细信息，请参阅[JSON-RPC方法](https://spdk.io/doc/jsonrpc.html)。

需要注意的是--config和--wait-for-rpc不能同时使用。

**限制coredump**

默认情况下，SPDK应用程序会将核心文件大小的资源限制设置为RLIM\_INFINITY。指定--limit-coredump不会设置资源限制。

**Tracepoint group掩码**

SPDK有一个实验性的低开销跟踪框架。这个框架中的跟踪点被组织成跟踪点组。默认情况下，禁用所有跟踪点组。--tpoint group mask可用于在应用程序中启用跟踪点组的特定子集。

注意：有关跟踪点框架的其他文档正在进行中。

**延迟初始化**

SPDK应用程序在一组状态中进行，从STARTUP开始到RUNTIME时结束。

如果提供了--wait for rpc参数，SPDK将在启动子系统初始化之前暂停。这种状态被称为STARTUP。JSON RPC服务器已准备就绪，但只有一个子系统的命令可用于设置初始化参数。SPDK应用程序进入RUNTIME状态后，无法更改这些参数。当客户端完成配置SPDK子系统时，需要发出[start\_subsystem\_init](https://spdk.io/doc/jsonrpc.html#rpc_start_subsystem_init) 的RPC命令以开始初始化过程。rpc\_start\_subsystem\_init返回true后，spdk将进入运行时状态，可用命令列表将变大。

要查看当前状态中所有可用的RPC方法，请使用rpc\_get\_methods，并且将参数-c（current）设置为true。

有关更多详细信息，请参阅[JSON-RPC方法](https://spdk.io/doc/jsonrpc.html)文档。

**只创建一个hugetlbfs文件**

该选项不必每页创建一个hugetlbfs文件，而是让SPDK为每个socket的每个大页面创建一个文件。[Virtio驱动](https://spdk.io/doc/virtio.html)需要与超过8个大页面一起使用。请参考[2MB大页面](https://spdk.io/doc/virtio.html#virtio_2mb)章节。

**多进程模式**

当指定--shm id时，应用程序以多进程模式启动。使用相同shm-id的应用程序共享其内存和[NVMe设备](https://spdk.io/doc/nvme.html#nvme_multi_process)。以给定id开头的第一个应用程序成为主要进程（primary process），其余的称为辅助进程（secondary processes），只附加到它上面。当主进程退出时，辅助进程继续运行，但此时不能附加新进程。同一个shm-id组中的所有进程必须使用相同的[--single-file-segments设置](https://spdk.io/doc/app_overview.html#cmd_arg_single_file_segments)。

**内存大小**

要保留的Hugepages内存的总大小。如果使用DPDK env层，它将从所有可用的hugetlbfs挂载中保留内存，从页面大小最大的挂载开始。此选项接受具有可能的二进制前缀的字节数，例如1024、1024M、1G。默认单位是兆字节。

从DPDK 18.05.1开始，可以在运行时保留Hugepages，这意味着SPDK应用程序可以使用0个预先保留的内存启动。与在应用程序启动时预先保留的Hugepages不同，运行时保留的Hugepages一旦不再使用，就会被释放到系统中。

**禁用PCI访问**

如果在禁用PCI访问的情况下运行SPDK，则不会检测到任何PCI设备。这主要包括NVMe和IOAT设备。此外，在此模式下不需要VFIO和UIO内核模块。

**PCI地址黑名单和白名单**

如果使用黑名单，则将忽略具有提供的PCI地址的所有设备。如果使用白名单，则仅探测列入白名单的设备。-B或者-W可以多次使用，但不能混合在一起。也就是说，-B和-W不能同时使用。

**初始化后取消HugePage文件的链接**

默认情况下，每个基于DPDK的应用程序都会尝试在初始化期间删除任何孤立的hugetlbfs文件。此选项在创建时立即删除当前进程的hugetlbfs文件，但与--shm-id不兼容。

**调试日志**

启用特定的调试日志类型。此选项可以多次使用。在--help输出中提供了所有可用类型的列表，其中--logflag all启用了所有类型。调试日志仅在SPDK的调试版本中可用。

**CPU掩码**

每当提到CPU掩码时，它是以下格式之一的字符串：

* 不区分大小写的十六进制字符串，带或不带“0x”前缀。
* 逗号分隔的CPU列表或CPU范围列表。使用' - '定义范围。

例

以下CPU掩码相等，对应于CPU 0,1,2,8,9,10,11和12：

0x1f07

0x1F07

1f07

[0,1,2,8-12]

[0,1,2,8,9,10,11,12]

## 3.3. iSCSI Target

### 3.3.1. iSCSI Target入门指南

存储性能开发工具包iSCSI目标应用程序命名为iscsi\_tgt。以下部分介绍如何从克隆包中运行iscsi。

**先决条件**

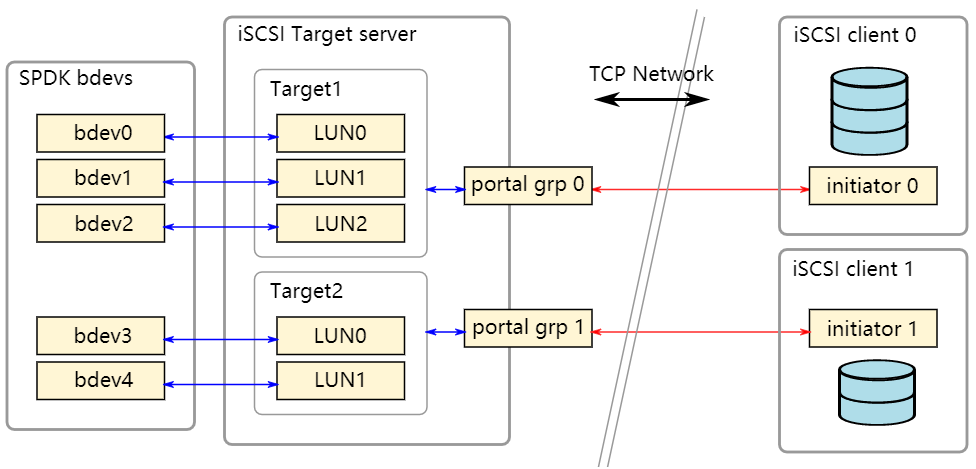
本指南首先假设您已经可以在平台上构建标准的SPDK发行版。

一旦构建，二进制文件将存在于app/iscsi\_tgt中。

如果想通过使用信号来杀死应用程序，请确保使用SIGTERM，然后应用程序将在退出之前释放所有共享内存资源，SIGKILL将使共享内存资源无法被应用程序释放，您可能需要手动释放资源。

**介绍**

下图显示了本文档中描述的iSCSI结构的不同部分之间的关​​系。



iSCSI结构

### 3.3.2. 通过配置文件配置iSCSI Target

使用特定于iscsi的配置文件来配置iSCSI Target。完整记录的示例配置文件位于etc/spdk/iscsi.conf.in。

配置文件用于配置SPDK iSCSI Target。此文件定义以下内容：用作iSCSI portal的TCP端口; 常规iSCSI参数; 启动器名称和地址，以允许访问iSCSI目标节点; 通过iSCSI LUN导出的存储后端的数量和类型; portal组，启动器组和LUN之间的iSCSI目标节点映射。

您应该复制示例配置文件，修改它以适合您的环境，然后运行iscsi\_tgt应用程序并使用-c选项将配置文件传递给它。现在，目标需要提升权限（root）才能运行。

app/iscsi\_tgt/iscsi\_tgt -c /path/to/iscsi.conf

**将CPU核心分配给iSCSI Target**

SPDK使用[DPDK环境抽象层](http://dpdk.org/doc/guides/prog_guide/env_abstraction_layer.html)（DPDK EAL）来访问硬件资源，例如巨大的内存页面和CPU内核。DPDK EAL提供了将线程分配给特定内核的功能。要确保SPDK iSCSI Target具有最佳性能，请将NIC和NVMe设备放在同一NUMA节点上，并将目标配置为在与该节点关联的CPU核心上运行。以下命令行选项用于配置SPDK iSCSI目标：

-m 0xF000000

这是CPU核心的十六进制位掩码，iSCSI Target将开始轮询线程。在这个示例中，将使用CPU核心24,25,26和27。

**在iSCSI Target中配置LUN**

iSCSI Target节点中的每个LUN都与SPDK块设备相关联。有关配置SPDK块设备的详细信息，请参见“ [块设备用户指南”](https://spdk.io/doc/bdev.html)。块映射到LUN映射在配置文件中指定为：

[TargetNodeX]

LUN0 Malloc0

LUN1 Nvme0n1

磁盘是一个RAM磁盘，它是由iSCSI在用户空间中分配的一块内存。如果系统有足够的DMA通道，它将使用卸载引擎来完成拷贝工作而不是memcpy。

### 3.3.3. 通过RPC方法配置iSCSI Target

除配置文件外，还可以通过JSON-RPC调用配置iSCSI目标。有关详细信息，请参阅[JSON-RPC方法](https://spdk.io/doc/jsonrpc.html)。

**portal组**

* add\_portal\_group - 添加portal组（pg）。
* delete\_portal\_group - 删除现有portal组。
* add\_pg\_ig\_maps - 将启动器组（ig）添加到现有iSCSI目标节点的portal组映射。
* delete\_pg\_ig\_maps - 从现有iSCSI目标节点删除启动器组到portal组映射。
* get\_portal\_groups - 显示有关所有可用portal组的信息。

/path/to/spdk/scripts/rpc.py add\_portal\_group 1 10.0.0.1:3260

**启动器组(initiator group)**

* add\_initiator\_group - 添加启动器组。
* delete\_initiator\_group - 删除现有的启动器组。
* add\_initiators\_to\_initiator\_group - 将启动器添加到现有启动器组。
* get\_initiator\_groups - 显示有关所有可用启动器组的信息。

/path/to/spdk/scripts/rpc.py add\_initiator\_group 2 ANY 10.0.0.2/32

**Target节点**

* construct\_target\_node - 添加iSCSI目标节点。
* delete\_target\_node - 删除iSCSI目标节点。
* target\_node\_add\_lun - 将LUN添加到现有iSCSI目标节点。
* get\_target\_nodes - 显示有关所有可用iSCSI目标节点的信息。

/path/to/spdk/scripts/rpc.py construct\_target\_node Target3 Target3\_alias MyBdev：0 1：2 64 -d

### 3.3.4. 配置iSCSI启动器

Linux发起程序是open-iscsi。

安装open-iscsi软件包Fedora：

yum install -y iscsi-initiator-utils

Ubuntu的：

apt-get install -y open-iscsi

**建立Setup**

编辑/etc/iscsi/iscsid.conf

node.session.cmds\_max = 4096

node.session.queue\_depth = 128

必须重新启动iscsid或接收SIGHUP才能使更改生效。要发送SIGHUP，请运行：

killall -HUP iscsid

对/etc/sysctl.conf的建议更改

net.ipv4.tcp\_timestamps = 1

net.ipv4.tcp\_sack = 0

net.ipv4.tcp\_rmem = 10000000 10000000 10000000

net.ipv4.tcp\_wmem = 10000000 10000000 10000000

net.ipv4.tcp\_mem = 10000000 10000000 10000000

net.core.rmem\_default = 524287

net.core.wmem\_default = 524287

net.core.rmem\_max = 524287

net.core.wmem\_max = 524287

net.core.optmem\_max = 524287

net.core.netdev\_max\_backlog = 300000

**发现Discory**

假设target IP是10.0.0.1

iscsiadm -m discovery -t sendtargets -p 10.0.0.1

**连接到Target**

iscsiadm -m node --login

此时，iSCSI目标应显示为SCSI磁盘。检查dmesg以查看它们出现的情况。

**断开与Target的连接**

iscsiadm -m node --logout

**删除Target节点缓存**

iscsiadm -m node -o delete

这将导致启动器忘记所有先前发现的iSCSI目标节点。

**查找iSCSI LUN的/ dev / sdX节点**

iscsiadm -m session -P 3 | grep "Attached scsi disk" | awk '{print $4}'

这将显示所有已登录iSCSI会话中每个SCSI LUN的/ dev节点名称。

**调整**

连接目标后，可以调整它们。例如，如果/dev/sdc是iSCSI磁盘，则可以执行以下操作：将noop设置为scheduler

echo noop>/sys/block/sdc/queue/scheduler

禁用合并/合并（对于精确的工作负载测量非常有用）

echo“2”>/sys/block/sdc/queue/nomerges

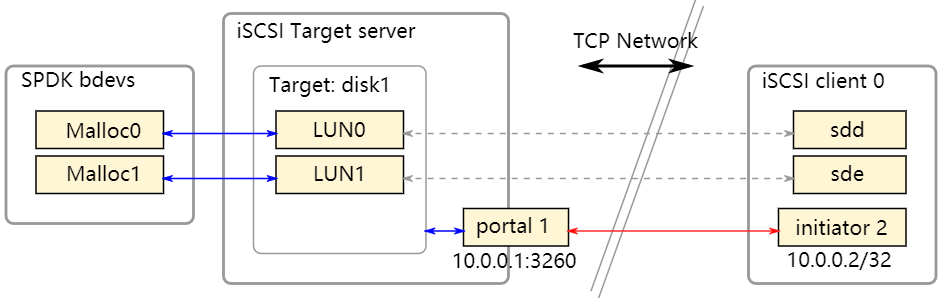
增加块队列的请求

echo“1024”>/sys/block/sdc/queue/nr\_requests

### 3.3.5. rpc配置示例\*\*\*

**使用一个portal和两个LUN配置简单iSCSI Target**

假设我们有一个iSCSI目标服务器，地址端口为10.0.0.1:3260，两个LUN（Malloc0和Malloc），并接受10.0.0.2/3260上的启动器，如下图所示：



iSCSI配置示例

**配置iSCSI目标**

启动iscsi\_tgt应用程序：

$ ./app/iscsi\_tgt/iscsi\_tgt

构建两个64MB Malloc块设备，512B扇区大小“Malloc0”和“Malloc1”：

$ ./scripts/rpc.py construct\_malloc\_bdev -b Malloc0 64 512

$ ./scripts/rpc.py construct\_malloc\_bdev -b Malloc1 64 512

创建ID为1且地址为10.0.0.1:3260的新portal组：

$ ./scripts/rpc.py add\_portal\_group 1 10.0.0.1:3260

创建一个ID为2的启动器组以接受来自10.0.0.2/3260的任何连接：

$ ./scripts/rpc.py add\_initiator\_group 2 ANY 10.0.0.2/32

最后使用先前创建的bdev构建一个目标，如LUN0（Malloc0）和LUN1（Malloc1），名称为“disk1”，别名为“Data Disk1”，使用portal组1和发起者组2。

$ ./scripts/rpc.py construct\_target\_node disk1“Data Disk1”“Malloc0：0 Malloc1：1”1：2 64 -d

**配置启动器（initiator）**

发现Target

$ iscsiadm -m discovery -t sendtargets -p 10.0.0.1

10.0.0.1:3260,1 iqn.2016-06.io.spdk:disk1

连接到Target

$ iscsiadm -m node --login

此时，iSCSI目标应显示为SCSI磁盘。

检查dmesg -c以查看它们出现的情况。在此示例中，它可能如下所示：

...

[630111.860078] scsi host68: iSCSI Initiator over TCP/IP

[630112.124743] scsi 68:0:0:0: Direct-Access INTEL Malloc disk 0001 PQ: 0 ANSI: 5

[630112.125445] sd 68:0:0:0: [sdd] 131072 512-byte logical blocks: (67.1 MB/64.0 MiB)

[630112.125468] sd 68:0:0:0: Attached scsi generic sg3 type 0

[630112.125926] sd 68:0:0:0: [sdd] Write Protect is off

[630112.125934] sd 68:0:0:0: [sdd] Mode Sense: 83 00 00 08

[630112.126049] sd 68:0:0:0: [sdd] Write cache: enabled, read cache: disabled, doesn't support DPO or FUA

[630112.126483] scsi 68:0:0:1: Direct-Access INTEL Malloc disk 0001 PQ: 0 ANSI: 5

[630112.127096] sd 68:0:0:1: Attached scsi generic sg4 type 0

[630112.127143] sd 68:0:0:1: [sde] 131072 512-byte logical blocks: (67.1 MB/64.0 MiB)

[630112.127566] sd 68:0:0:1: [sde] Write Protect is off

[630112.127573] sd 68:0:0:1: [sde] Mode Sense: 83 00 00 08

[630112.127728] sd 68:0:0:1: [sde] Write cache: enabled, read cache: disabled, doesn't support DPO or FUA

[630112.128246] sd 68:0:0:0: [sdd] Attached SCSI disk

[630112.129789] sd 68:0:0:1: [sde] Attached SCSI disk

...

您还可以使用简单的bash命令在所有已记录的iSCSI会话中查找每个iSCSI LUN的/ dev / sdX节点：

$ iscsiadm -m session -P 3 | grep ”Attached scsi disk”| awk'{print $ 4}'

sdd

sde

### 3.3.6. iSCSI 热插拔

在iSCSI级别，我们为Hotplug提供以下支持：

bdev / nvme：在bdev / nvme级别，我们启动一个hotplug监视器，它将定期调用[spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)以获取hotplug事件。我们为[spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)提供私有attach\_cb和remove\_cb 。对于attach\_cb，我们将在附加的NVMe设备上创建块设备，对于remove\_cb，我们将取消注册块设备，该设备也将通知上层堆栈（对于iSCSI目标，上层堆栈为scsi / lun） ）处理热删除事件。

scsi / lun：当LUN从块设备层收到热删除通知时，LUN将被标记为已删除，此后的所有IO将返回检查条件状态。然后LUN启动一个轮询器，它将等待已经提交给块设备的所有命令返回; 在所有命令返回后，LUN将被删除。

**已知的错误和限制**

对于write命令，如果要使用write命令测试hotplug将导致r2t，例如1M大小IO，则会使iscsi tgt崩溃。对于read命令，如果要测试具有大的read IO的热插拔，例如1M大小的IO，则可能会使iscsi tgt崩溃。

也可以看看[spdk\_nvme\_probe](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)。

## 3.4. NVMe over Fabrics Target

也可以看看

* + 1. NVMe over Fabrics主机支持

1. NVMe-oF目标跟踪点

**NVMe-oF Target入门指南**

SPDK NVMe over Fabrics Target是一个用户空间应用程序，通过以太网，Infiniband或光纤通道等结构呈现块设备。SPDK目前支持RDMA和TCP传输。

NVMe over Fabrics规范定义了可以通过不同传输导出的子系统。SPDK选择将导出这些子系统的软件称为“Target”，这是用于iSCSI的术语。规范指的是作为“主机”连接到Target的“客户端”。许多人也会将主机称为“启动器”，这与iSCSI术语中的内容相同。SPDK将尝试坚持使用术语“Target”和“主机”来匹配规范。

Linux内核还实现了NVMe-oF目标和主机，并且测试了SPDK与Linux内核实现的互操作性。

如果要使用signal杀死应用程序，请确保使用SIGTERM，然后应用程序将在退出之前释放所有共享内存资源，SIGKILL将使共享内存资源无法被应用程序释放，您可能需要手动释放资源。

**RDMA传输支持**

它需要一个支持RDMA的NIC，并安装相应的OFED（OpenFabrics Enterprise Distribution）软件包才能运行。也许OS发行版提供了包，但OFED也可以[在这里找到](https://downloads.openfabrics.org/OFED/)。

**先决条件**

要使用RDMA传输构建nvmf\_tgt，还有一些其他依赖项。

Fedora的：

dnf install libibverbs-devel librdmacm-devel

Ubuntu的：

apt-get install libibverbs-dev librdmacm-dev

然后在启用RDMA的情况下构建SPDK：

./configure --with-rdma <其他配置参数>

make

一旦构建，二进制文件将在app/nvmf\_tgt生成。

**InfiniBand / RDMA动词的先决条件**

在使用RDMA传输启动NVMe-oF目标之前，我们必须加载InfiniBand和RDMA模块，以允许用户空间进程直接使用InfiniBand / RDMA谓词。

modprobe ib\_cm

modprobe ib\_core

＃请注意，ib\_ucm在较新版本的内核中不存在，不是必需的。

modprobe ib\_ucm || True

modprobe ib\_umad

modprobe ib\_uverbs

modprobe iw\_cm

modprobe rdma\_cm

modprobe rdma\_ucm

**RDMA NIC的先决条件**

在启动我们的NVMe-oF目标之前，我们必须检测RDMA网卡并为其分配IP地址。

**查找RDMA NIC和关联的网络接口**

ls /sys/class/infiniband/\*/device/net

**Mellanox ConnectX-3 RDMA网卡**

modprobe mlx4\_core

modprobe mlx4\_ib

modprobe mlx4\_en

**Mellanox ConnectX-4 RDMA网卡**

modprobe mlx5\_core

modprobe mlx5\_ib

**为RDMA NIC分配IP地址**

ifconfig eth1 192.168.100.8 netmask 255.255.255.0 up

ifconfig eth2 192.168.100.9 netmask 255.255.255.0 up

**RDMA限制**

由于RDMA NIC对注册的内存区域数量施加了限制，因此SPDK NVMe-oF Target应用程序最终可能无法分配更多可DMA的内存。这是DPDK动态内存管理的一个缺陷，并且最有可能发生在运行时保留的2MB大页面。一种类型的存储器瓶颈是NIC存储器区域的数量，例如，对于最大存储器区域数量，一些NIC报告多达2048个。这为我们提供了4GB内存限制，总内存区域有2MB大页面。可以通过使用1GB大页面或在应用程序启动时使用--mem-size或-s选项预先保留内存来克服它。所有预先保留的内存都将注册为单个区域，但在SPDK应用程序终止之前不会返回到系统。

**TCP传输支持**

默认情况下，传输内置于nvmf\_tgt中，并且它不需要任何特殊库。

**在Fabrics Target上配置SPDK NVMe**

可以使用JSON RPC配置NVMe over Fabrics Target。配置NVMe-oF子系统所需的基本RPC详述如下。有关使用NVMe over Fabrics特定RPC的更多信息，请参考[NVMe-oF Target](https://spdk.io/doc/jsonrpc.html#jsonrpc_components_nvmf_tgt) RPC部分章节。

不建议使用.ini样式配置文件来配置NVMe-oF目标，并应使用基于JSON的RPC替换。.ini样式配置文件可以通过新脚本转换为json格式scripts/config\_converter.py。

**FC传输支持**

要使用FC传输构建nvmf\_tgt，还有一个额外的FC LLD（低级驱动程序）代码依赖项。有关获取FC驱动程序模块的说明，请与您的FC供应商联系。

**Broadcom FC LLD代码**

适用于Broadcom FC NVMe的适配器的FC LLD驱动程序可从[https://github.com/ecdufcdrvr/bcmufctdrvr获取](https://github.com/ecdufcdrvr/bcmufctdrvr)。

获取FC LLD模块，然后在启用FC的情况下构建SPDK：

克隆SPDK repo并初始化子模块后，构建FC LLD库，然后可以与fc传输链接。

git clone https://github.com/spdk/spdk spdk

git clone https://github.com/ecdufcdrvr/bcmufctdrvr fc

cd spdk

git submodule update --init

cd ../fc

make DPDK\_DIR = ../spdk/dpdk/build SPDK\_DIR = ../spdk

cd ../spdk

./configure --with-fc = ../fc/build

make

**使用RPC**

使用提升的权限启动nvmf\_tgt应用程序。一旦目标启动，nvmf\_create\_transport rpc可用于初始化给定的传输。下面是一个示例，其中目标启动并配置了两个不同的传输。RDMA传输配置为I / O单元大小为8192字节，每个控制器4个最大qpairs，以及0字节的封装数据大小。TCP传输配置为I / O单元大小为16384字节，每个控制器8个最大qpairs，以及8192字节的封装数据大小。

app/nvmf\_tgt/nvmf\_tgt

scripts / rpc.py nvmf\_create\_transport -t RDMA -u 8192 -p 4 -c 0

scripts / rpc.py nvmf\_create\_transport -t TCP -u 16384 -p 8 -c 8192

下面是创建malloc bdev并将其分配给子系统的示例。根据您自己的情况，使用RDMA传输调整bdevs，NQN，序列号和IP地址。如果将“rdma”替换为“TCP”，则子系统将添加具有TCP传输的侦听器。

scripts/rpc.py construct\_malloc\_bdev -b Malloc0 512 512

scripts/rpc.py nvmf\_subsystem\_create nqn.2016-06.io.spdk：cnode1 -a -s SPDK00000000000001 -d SPDK\_Controller1

scripts/rpc.py nvmf\_subsystem\_add\_ns nqn.2016-06.io.spdk：cnode1 Malloc0

scripts/rpc.py nvmf\_subsystem\_add\_listener nqn.2016-06.io.spdk：cnode1 -t rdma -a 192.168.100.8 -s 4420

**NQN正式定义**

NVMe限定名称或NQN在[NVMe规范的](http://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf)第7.9节中定义。SPDK试图使用[Extended Backus-Naur形式](https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form)来形式化该定义。在验证NQN时，SPDK模块使用此正式定义（下面提供）。

基本类型

year = 4 \* digit ;

month = '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09' | '10' | '11' | '12' ;

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

hex digit = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

NQN定义

NVMe Qualified Name = ( NVMe-oF Discovery NQN | NVMe UUID NQN | NVMe Domain NQN ), '\0' ;

NVMe-oF Discovery NQN = "nqn.2014-08.org.nvmexpress.discovery" ;

NVMe UUID NQN = "nqn.2014-08.org.nvmexpress:uuid:", string UUID ;

string UUID = 8 \* hex digit, '-', 3 \* (4 \* hex digit, '-'), 12 \* hex digit ;

NVMe Domain NQN = "nqn.", year, '-', month, '.', reverse domain, ':', utf-8 string ;

请注意，以上定义中的以下类型在其他地方定义：

1. utf-8 string：在[rfc 3629中](https://tools.ietf.org/html/rfc3629)定义。
2. 反向域：等同于[rfc 1034中](https://tools.ietf.org/html/rfc1034)定义的域名。

虽然在正式定义中没有说明，但SPDK强制要求“最大名称长度为223个字节”的规范。在定义nqn的长度时，SPDK不包括空终止字符，并且将接受包含多达223个有效字节的nqn和一个额外的空终止符。确切地说，SPDK遵循与c标准库函数[strlen（）](http://man7.org/linux/man-pages/man3/strlen.3.html)相同的约定。

**NQN比较**

SPDK逐个字节比较NQNs，不进行大小写匹配或unicode规范化。这对基于uuid的NQN具有特定的意义。例如，在SPDK NVMe-oF目标中进行比较时，以下一对NQN将不匹配：

nqn.2014-08.org.nvmexpress：uuid：11111111-aaaa-bbdd-ffee-123456789abc   
nqn.2014-08.org.nvmexpress：uuid：11111111-AAAA-BBDD-FFEE-123456789ABC

为了在使用SPDK时确保基于uuid的NQN的一致性，用户在NQN中表示字母十六进制数字时应使用小写。

将CPU核心分配给NVMe over Fabrics Target

SPDK使用[DPDK环境抽象层](http://dpdk.org/doc/guides/prog_guide/env_abstraction_layer.html)来访问硬件资源，例如巨大的内存页面和CPU内核。DPDK EAL提供了将线程分配给特定内核的功能。要确保SPDK NVMe-oF目标具有最佳性能，请将NIC和NVMe设备配置为位于同一NUMA节点上。

该-m核心屏蔽选项指定SPDK被允许上执行工作项目的CPU内核的位掩码。例如，要允许SPDK使用核心24,25,26和27：

app/nvmf\_tgt/nvmf\_tgt -m 0xF000000

**在Fabric Fabric主机上配置Linux NVMe**

Linux内核和SPDK都在Fabric架构主机上实现NVMe。Linux内核NVMe-oF RDMA主机支持由nvme-rdma驱动程序提供（支持RDMA传输）和nvme-tcp（支持TCP传输）。以下显示了用于加载驱动程序的两个不同命令。

modprobe nvme-rdma

modprobe nvme-tcp

nvme-cli工具可用于与Fabrics主机上的Linux内核NVMe连接。有关discover，connect和disconnect命令的示例，请参见下文。在所有这三种情况下，通过将'rdma'换成'tcp'，可以将传输更改为TCP。

发现：

nvme discover -t rdma -a 192.168.100.8 -s 4420

连接：

nvme connect -t rdma -n“nqn.2016-06.io.spdk：cnode1”-a 192.168.100.8 -s 4420

断开：

nvme disconnect -n“nqn.2016-06.io.spdk：cnode1”

**启用NVMe-oF目标跟踪点以进行脱机分析和调试**

SPDK有一个跟踪框架，用于在运行时捕获低级事件信息。[NVMe-oF目标跟踪点](https://spdk.io/doc/nvmf_tgt_tracepoints.html)可以分析性能和应用程序崩溃。

## 3.5. Vhost Target（略）

## 3.6 块设备用户指南

### 3.6.1 bdev介绍

SPDK块设备层（通常简称为*bdev*）是一个C库，旨在等同于操作系统块存储层，该层通常位于传统内核存储堆栈中设备驱动程序的正上方。具体来说，此库提供以下功能：

* 一种可插拔模块API，用于实现与不同类型的块存储设备接口的块设备。
* NVMe，malloc（ramdisk），Linux AIO，virtio-scsi，Ceph RBD，Pmem和Vhost-SCSI Initiator等驱动程序模块。
* 用于枚举和声明SPDK块设备，然后在这些设备上执行操作（读取，写入，取消映射等）的应用程序API。
* 堆栈块设备以创建复杂I / O管道的工具，包括逻辑卷管理（lvol）和分区支持（GPT）。
* 通过JSON-RPC配置块设备。
* 请求排队，超时和重置处理。
* 多个无锁队列，用于将I / O发送到块设备。

Bdev模块创建抽象层，为所有设备提供通用API。用户可以使用可用的bdev模块或使用下面任何类型的设备创建自己的模块（有关详细信息，请参阅[编写自定义块设备模块](https://spdk.io/doc/bdev_module.html)）。SPDK还提供vbdev模块，用于在现有bdev上创建块设备。例如[逻辑卷](https://spdk.io/doc/bdev.html#bdev_ug_logical_volumes)（lvol）或[SPDK GPT分区表](https://spdk.io/doc/bdev.html#bdev_ug_gpt)。

**先决条件**

本指南假设您已经可以在平台上构建标准SPDK分发版。块设备层是一个C库，其中包含一个名为[bdev.h的](https://spdk.io/doc/bdev_8h.html)公共头文件。以下章节中描述的所有SPDK配置都是使用JSON-RPC命令完成的。SPDK提供了一个基于python的命令行工具，用于发送位于的RPC命令scripts/rpc.py。用户可以通过使用-h或--help标记运行此脚本来列出可用命令。此外，用户可以通过运行直接从SPDK应用程序检索当前支持的RPC命令集scripts/rpc.py rpc\_get\_methods。通过添加-h flag作为命令参数，可以显示每个命令的详细帮助。

### 3.6.2 通用RPC命令

**get\_bdevs**

可以使用get\_bdevs的RPC命令获取当前可用的块设备列表，包括有关它们的详细信息。用户可以添加可选参数name以获取该名称bdev指定的详细信息。

响应示例

{

"num\_blocks": 32768,

"assigned\_rate\_limits": {

"rw\_ios\_per\_sec": 10000,

"rw\_mbytes\_per\_sec": 20

},

"supported\_io\_types": {

"reset": true,

"nvme\_admin": false,

"unmap": true,

"read": true,

"write\_zeroes": true,

"write": true,

"flush": true,

"nvme\_io": false

},

"driver\_specific": {},

"claimed": false,

"block\_size": 4096,

"product\_name": "Malloc disk",

"name": "Malloc0"

}

**set\_bdev\_qos\_limit**

用户可以使用set\_bdev\_qos\_limit的RPC命令，在现有bdev上启用、调整和禁用速率限制。支持两种类型的速率限制：IOPS和带宽。可以随时为指定的bdev启用、调整和禁用速率限制。Bdev name是此RPC命令的必需参数，必须至少指定rw\_ios\_per\_sec 和 rw\_mbytes\_per\_sec中的一个。当两个速率限制都启用时，第一个满足的限制将生效。可以指定值0以禁用相应的速率限制。用户可以使用-h或运行此命令以--help获取更多信息。

**Histograms**

enable\_bdev\_histogram命令允许指定BDEV启用或禁用聚集等待时间数据。用户可以get\_bdev\_histogram使用scripts / histogram.py脚本调用和解析直方图。

示例命令

rpc.py enable\_bdev\_histogram Nvme0n1 --enable

该命令将启用在Nvme0n1设备上收集直方图的数据。

rpc.py get\_bdev\_histogram Nvme0n1 | histogram.py

该命令将下载收集的直方图数据。该脚本将解析数据并显示包含延迟范围的IO计数的表。

rpc.py enable\_bdev\_histogram Nvme0n1 --disable

该命令将禁用Nvme0n1设备上的直方图。

### 3.6.3 Ceph RBD

SPDK RBD-BDEV驱动程序提供SPDK块层对CEPH Rados块设备（RBD）的访问。CEPH RBD设备通过访问librbd和libados库，以访问CEPH导出的Rados块设备。若要创建ceph-bdev rpc命令，应使用construct\_rbd\_bdev。

示例命令

rpc.py construct\_rbd\_bdev rbd foo 512

此命令将创建一个bdev，表示来自名为'rbd'的池中的'foo'图像。

要删除块设备表示，请使用delete\_rbd\_bdev命令。

rpc.py delete\_rbd\_bdev Rbd0

### 3.6.4 压缩虚拟Bdev模块

压缩bdev模块（compression vbdev），可以配置为为底层精简配置的逻辑卷（lvol）提供压缩/解压缩服务。虽然底层模块可以是任何东西（即NVME bdev），但除非存储在磁盘上的数据被适当放置，否则将无法实现整体压缩优势。压缩vbdev模块依赖于调用的内部SPDK库reduce来完成此操作，有关详细信息，请参阅4.7 [SPDK“Reduce”块压缩算法](https://spdk.io/doc/reduce.html)。

vbdev模块依赖于DPDK CompressDev Framework来提供所有压缩功能。该框架为许多不同的软件压缩模块提供支持，并为英特尔QAT提供硬件辅助支持。现在，vbdev模块同时支持ISAL和QAT的DPDK驱动程序。

持久性存储器用于存储与备份设备上的数据布局相关联的元数据。SPDK依靠[PMDK](http://pmem.io/pmdk/)来连接持久性内存，因此PMDK支持的任何硬件都可以工作。如果在创建vbdev时，提供的PMEM目录未指向持久性内存（即常规文件系统），则性能将受到严重影响。vbdev模块和reduce库旨在将持久内存用于任何生产用途。

示例命令

rpc.py construct\_compress\_bdev -p /pmem\_files -b myLvol

在此示例中，使用持久性内存创建压缩vbdev，该内存映射到pmem\_files现有精简配置逻辑卷之上的目录myLvol。生成的压缩bdev将被命名为COMP\_LVS/myLvol，其中LVS是myLvol驻留的逻辑卷存储的名称。

逻辑卷称为后备设备，一旦创建了压缩vbdev，它就不能与将在指定目录中创建的持久内存文件分离。如果持久性内存文件不可用，则压缩vbdev也将不可用。

默认情况下，如果硬件和驱动程序可用并已加载，则vbdev模块将选择QAT驱动程序。如果没有，它将恢复为仅使用软件的ISAL驱动程序。通过使用以下命令，可以指定驱动程序，但这不是持久性的，因此必须在创建时或在加载基础逻辑卷之前完成它。在下面的示例中，

* 0告诉vbdev模块使用QAT（如果可用），否则使用ISAL，这是默认值，如果足够，则不需要该命令。
* 传递值1表示驱动程序使用QAT，如果该值不可用，则创建或加载vbdev将无法创建或加载。
* 如下所示的值“2”告诉模块使用ISAL，如果由于某种原因它不可用，则vbdev应该无法创建或加载。

rpc.py set\_compress\_pmd -p 2

要删除压缩vbdev，请使用以下命令，该命令也将删除PMEM文件。如果删除逻辑卷，则不会删除PMEM文件，compression vbdev将不可用。

rpc.py delete\_compress\_bdev COMP\_LVS/myLvol

### 3.6.5 加密虚拟Bdev模块

加密虚拟bdev模块（crypto virtual bdev），可以配置为为任何底层bdev提供静态数据加密。该模块依赖于DPDK CryptoDev Framework来提供所有加密功能。该框架仅支持许多不同的软件加密模块以及英特尔QAT板的硬件辅助支持。该框架还提供对密码、散列（hash）、身份验证和AEAD功能的支持。现在SPDK虚拟bdev模块仅支持密码，如下所示：

* AESN-NI多缓冲加密轮询模式驱动程序：RTE\_CRYPTO\_CIPHER\_AES128\_CBC
* 英特尔（R）QuickAssist（QAT）加密轮询模式驱动程序：RTE\_CRYPTO\_CIPHER\_AES128\_CBC（注意：QAT功能正常，但在硬件与SPDK CI系统完全集成之前，标记为实验性。）

为了支持使用bdev块偏移量（LBA）作为初始化向量（IV），加密模块将所有I / O分解为大小等于底层bdev的块大小的加密操作。例如，具有512B块大小的bdev的4K I / O将导致8次加密操作。

对于读取，提供给加密模块的缓冲区将用作未加密数据的目标缓冲区。但是，对于写入，临时暂存缓冲区用作加密的目标缓冲区，然后作为写缓冲区传递给底层bdev。这样做是为了避免加密原始源缓冲区中的数据，这可能会在某些用例中引起问题。

示例命令

rpc.py construct\_crypto\_bdev NVMe1n1 CryNvmeA crypto\_aesni\_mb 0123456789123456

此命令将在NVMe bdev“NVMe1n1”之上创建一个名为“CryNvmeA”的加密vbdev，并将使用DPDK软件驱动程序“crypto\_aesni\_mb”和密钥“0123456789123456”。

要删除vbdev，请使用delete\_crypto\_bdev命令。

rpc.py delete\_crypto\_bdev CryNvmeA

### 3.6.6 延迟vbdev模块

延迟vbdev模块（delay vbdev），用于在较低级别的bdev上实现预定的额外延迟。这使得能够在SPDK应用程序的功能性测试或可伸缩性测试期间模拟设备的延迟特性。例如，为了模拟处理I/O时驱动器延迟的影响，可以配置一个空的bdev（NULL bdev），并在其上配置一个延迟bdev（delay vbdev）。

delay-bdev模块的目的不是提供特定NVMe驱动器延迟的高保真复制，而是提供对一般延迟如何影响给定应用程序的“全局”理解。

使用bdev\_delay\_create 的rpc命令创建延迟bdev。这个命令接受6个参数，一个用于延迟bdev的名称，另一个用于基本bdev的名称。其余四个参数表示以下延迟值：平均读取延迟、平均写入延迟、P99读取延迟和P99写入延迟。在延迟的情况下，bdev p99延迟意味着在完成上层协议之前，I / O的百分之一将至少延迟p99延迟的值。所有延迟值都是以微秒为单位测量的。

示例命令：

rpc.py bdev\_delay\_create -b Null0 -d delay0 -r 10 --nine-nine-read-latency 50 -w 30 --nine-nine-write-latency 90

此命令将创建一个延迟bdev，平均读写延迟分别为10和30微秒和p99读写延迟分别为50和90微秒。

可以使用bdev\_delay\_delete rpc删除延迟bdev。

示例命令：

rpc.py bdev\_delay\_delete delay0

### 3.6.7 GPT（GUID分区表）

默认情况下启用GPT虚拟bdev驱动程序，不需要任何配置。它将自动检测任何连接的bdev上的[SPDK GPT分区表](https://spdk.io/doc/bdev.html#bdev_ug_gpt)，并可能创建多个虚拟bdev。

**SPDK GPT分区表**

SPDK分区类型GUID是7c5222bd-8f5d-4087-9c00-bf9843c7b58c。现有的SPDK bdev可以通过NBD显示为Linux块设备，然后使用标准分区工具进行分区。分区后，需要删除bdev并再次附加，以便GPT bdev模块查看任何更改。必须先加载NBD内核模块。要创建NBD，bdev用户应该使用start\_nbd\_disk命令。

示例命令

rpc.py start\_nbd\_disk Malloc0 /dev/nbd0

这将把SPDK bdev  Malloc0公开到在/dev/nbd0块设备。

要删除NBD设备，用户应使用stop\_nbd\_disk命令。

示例命令

rpc.py stop\_nbd\_disk /dev/nbd0

要显示完整或指定的nbd设备列表，用户应使用get\_nbd\_disksRPC命令。

示例命令

rpc.py stop\_nbd\_disk -n /dev/nbd0

使用NBD创建GPT分区表

＃通过JSON-RPC将bdev Nvme0n1显示为内核块设备/dev/nbd0

rpc.py start\_nbd\_disk Nvme0n1 /dev/nbd0

＃创建GPT分区表。

parted -s /dev/nbd0 mklabel gpt

＃添加占用50％可用空间的分区。

parted -s /dev/nbd0 mkpart MyPartition'0％''50％'

＃将分区类型更改为SPDK GUID。

#sgdisk是gdisk包的一部分。

sgdisk -t 1：7c5222bd-8f5d-4087-9c00-bf9843c7b58c /dev/nbd0

＃停止NBD设备（停止导出/dev/nbd0）。

rpc.py stop\_nbd\_disk /dev/nbd0

#至此，Nvme0n1配置成功GPT分区表，并且第一个分区将自动显示为SPDK应用程序中的Nvme0n1p1。

### 3.6.8 iSCSI bdev

SPDK iSCSI bdev驱动程序依赖于libiscsi，因此默认情况下不启用。要使用它，请使用额外的--with-iscsi-initiator配置选项构建SPDK 。

以下命令从给定的iSCSI URL处公开的单个LUN中创建一个iSCSI0 bdev，iqn.2016-06.io.spdk:init作为报告的发起程序IQN。

rpc.py construct\_iscsi\_bdev -b iSCSI0 -i iqn.2016-06.io.spdk:init --url iscsi://127.0.0.1/iqn.2016-06.io.spdk:disk1/0

URL的格式如下： iscsi://[<username>[%<password>]@]<host>[:<port>]/<target-iqn>/<lun>

### 3.6.9 Linux AIO bdev

SPDK AIO bdev驱动程序通过Linux AIO(异步读写模型)提供对Linux内核块设备或linux文件系统上文件的SPDK块层访问。请注意，使用了O\_DIRECT，因此绕过了Linux页面缓存。此模式可能与基于内核的典型目标接近，因为用户空间目标可以在不使用用户空间驱动程序的情况下获得。要创建AIO，construct\_aio\_bdev应使用bdev RPC命令。

示例命令

rpc.py construct\_aio\_bdev /dev/sda aio0

此命令aio0将从/dev/sda 创建设备。

rpc.py construct\_aio\_bdev /tmp/file file 8192

此命令file将从/tmp/file 创建块大小为8192的设备。

要删除aio bdev，请使用delete\_aio\_bdev命令。

rpc.py delete\_aio\_bdev aio0

### 3.6.10 OCF虚拟bdev

OCF虚拟bdev模块基于[Open CAS Framework](https://github.com/Open-CAS/ocf) —— 一种高性能块存储缓存元库。要启用该模块，请使用--with-ocf标志配置SPDK 。OCF bdev可用于为任何底层bdev启用缓存。

以下是创建OCF bdev的示例命令：

rpc.py construct\_ocf\_bdev Cache1 wt Malloc0 Nvme0n1

此命令将创建新的OCF bdev Cache1，其中bdev Malloc0作为缓存设备，Nvme0n1作为核心设备，Write-Through作为初始缓存模式。Malloc0将被用作Nvme0n1的缓存，因此写入Cache1的数据最终将出现Nvme0n1上。默认情况下，OCF将配置行大小等于4KiB的缓存，并且禁用非易失性元数据。

删除Cache1：

rpc.py delete\_ocf\_bdev Cache1

在删除过程中， OCF缓存将停止，并将所有缓存的数据写入核心设备。

请注意，OCF对每个设备的RAM要求约为56000 + 缓存设备大小 \* 58 / 缓存行大小（以字节为单位）。要获得有关OCF的更多信息，请访问[OCF文档](https://open-cas.github.io/)。

### 3.6.11 Malloc bdev

Malloc bdevs是ramdisks(通过软件将一部分内存RAM模拟为硬盘来使用的一种技术)。由于它的性质，它们是不稳定的。它们是从为SPDK应用程序提供的巨大页面内存创建的。

### 3.6.12 NULL bdev

SPDK null bdev驱动程序是一个虚拟块I / O目标，它会丢弃所有写入并返回未定义的读取数据。它可以用最小的块设备开销对bdev I / O堆栈的其余部分进行基准测试，并用于测试使用Malloc bdev无法轻松创建的配置。要创建Null bdev，应该使用construct\_null\_bdev命令。

示例命令

rpc.py construct\_null\_bdev Null0 8589934592 4096

此命令将创建一个块大小为4096，总大小为8 PB的Null0设备。

要删除null bdev，请使用delete\_null\_bdev命令。

rpc.py delete\_null\_bdev Null0

### 3.6.13 NVMe bdev

有两种方法可以在SPDK中基于NVMe设备创建块设备。第一种方法是连接本地PCIe驱动器，第二种方法是连接NVMe-oF设备。在这两种情况下，用户都应使用construct\_nvme\_bdev命令来实现这一点。

示例命令

rpc.py construct\_nvme\_bdev -b NVMe1 -t PCIe -a 0000:01:00.0

此命令将在系统中创建物理设备的NVMe bdev。

rpc.py construct\_nvme\_bdev -b Nvme0 -t RDMA -a 192.168.100.1 -f IPv4 -s 4420 -n nqn.2016-06.io.spdk:cnode1

此命令将创建NVMe-oF资源的NVMe bdev。

要删除NVMe控制器，请使用delete\_nvme\_controller命令。

rpc.py delete\_nvme\_controller Nvme0

此命令将删除名为Nvme0的NVMe控制器。

### 3.6.14 逻辑卷Lvol

Logical Volumes库是一个灵活的存储空间管理系统。它允许在其他bdev之上创建和管理可变大小的虚拟块设备。SPDK逻辑卷库建立在[Blobstore程序员指南](https://spdk.io/doc/blob.html)之上。有关详细说明，请参阅[逻辑卷](https://spdk.io/doc/logical_volumes.html#lvol)。

**Lvol store**

在创建逻辑卷（lvols）之前，必须首先在选定的块设备上创建一个lvol store。Lvol store是lvols的容器，负责管理lvol bdevs的底层bdev空间分配和存储元数据。要创建lvol存储，用户应该使用construct\_lvol\_store命令。

示例命令

rpc.py construct\_lvol\_store Malloc2 lvs -c 4096

这将在名为Malloc2的bdev之上创建名为lvs的lvol store，集群（cluster）大小为4096。作为响应，将向用户提供uuid，这是唯一的lvol store标识符。

用户可以使用get\_lvol\_stores的RPC命令获取可用的lvol store列表（没有可用的参数）。

响应示例

{

“uuid”：“330a6ab2-f468-11e7-983e-001e67edf35d”，

“base\_bdev”：“Malloc2”，

“free\_clusters”：8190，

“cluster\_size”：8192，

“total\_data\_clusters”：8190，

“block\_size”：4096，

“name”：“lvs”

}

要删除lvol store，用户应该使用destroy\_lvol\_store命令。

示例命令

rpc.py destroy\_lvol\_store -u 330a6ab2-f468-11e7-983e-001e67edf35d

rpc.py destroy\_lvol\_store -l lvs

**Lvol bdev**

要在现有lvol store上创建lvol，用户应使用construct\_lvol\_bdev命令。每个创建的lvol都将由新的bdev表示。

示例命令

rpc.py construct\_lvol\_bdev lvol1 25 -l lvs

rpc.py construct\_lvol\_bdev lvol2 25 -u 330a6ab2-f468-11e7-983e-001e67edf35d

### 3.6.15 RAID

RAID虚拟bdev模块提供，将任何SPDK bdev组合成一个RAID bdev的功能。目前，SPDK仅支持RAID 0。

RAID功能不会在成员磁盘上存储磁盘元数据，因此用户必须在重新启动应用程序时重建RAID卷。如果成员磁盘尚不存在，用户可以指定成员磁盘来创建RAID卷事件。—— 由于成员磁盘稍后注册，RAID模块将声明它们并在所有成员磁盘可用后显示RAID卷。允许使用不同大小的磁盘 ——最小的磁盘大小是每个成员磁盘上使用的空间量。

示例命令

rpc.py construct\_raid\_bdev -n Raid0 -z 64 -r 0 -b "lvol0 lvol1 lvol2 lvol3"

rpc.py get\_raid\_bdevs

rpc.py destroy\_raid\_bdev Raid0

### 3.6.16 Passthru

SPDK Passthru虚拟块设备模块用作如何编写虚拟块设备模块的示例。它实现了vbdev模块所需的功能，并演示了一些其他基本功能，例如每个I / O上下文的使用。

示例命令

rpc.py construct\_passthru\_bdev -b aio -p pt

rpc.py delete\_passthru\_bdev pt

### 3.6.17 Pmem

SPDK pmem bdev驱动程序使用pmemblk池作为块I / O操作的目标。有关Pmem内存的详细信息，请参阅[http://pmem.io](http://pmem.io/)网站上的PMDK文档。首先，用户需要配置SPDK以包含PMDK支持：

configure --with-pmdk

要创建用于SPDK用户的pmemblk池，应使用create\_pmem\_pool命令。

示例命令

rpc.py create\_pmem\_pool /path/to/pmem\_pool 25 4096

要获取有关已创建的pmem池文件的信息，用户可以使用pmem\_pool\_info命令。

示例命令

rpc.py pmem\_pool\_info /path/to/pmem\_pool

要删除pmem池文件，用户可以使用delete\_pmem\_pool命令。

示例命令

rpc.py delete\_pmem\_pool /path/to/pmem\_pool

要基于pmemblk池文件创建bdev，用户应使用construct\_pmem\_bdev命令。

示例命令

rpc.py construct\_pmem\_bdev /path/to/pmem\_pool -n pmem

要删除块设备表示，请使用delete\_pmem\_bdev命令。

rpc.py delete\_pmem\_bdev pmem

### 3.6.18 Virtio Block

Virtio-Block驱动程序允许从Virtio-Block设备创建SPDK bdev。

以下命令创建一个名为VirtioBlk0的Virtio-Block设备，该设备来自 SPDK [vhost Target](https://spdk.io/doc/vhost.html)直接显示的用户套接字/tmp/vhost.0。可选的vq-count和vq-size参数，指定要使用的请求队列数和队列深度。

rpc.py construct\_virtio\_dev --dev-type blk --trtype user --traddr /tmp/vhost.0 --vq-count 2 --vq-size 512 VirtioBlk0

该驱动程序也可以在基于QEMU的VM中使用。以下命令创建一个名为VirtioBlk0Virtio PCI设备的Virtio Block设备0000:00:01.0。将从PCI配置空间自动读取整个配置。它将反馈的所有参数传递给QEMU的vhost-user-scsi-pci设备。

rpc.py construct\_virtio\_dev --dev-type blk --trtype pci --traddr 0000:01:00.0 VirtioBlk1

可以使用以下命令删除Virtio-Block设备

rpc.py remove\_virtio\_bdev VirtioBlk0

### 3.6.19 Virtio SCSI

Virtio-SCSI驱动程序允许从Virtio-SCSI LUN创建SPDK块设备。

Virtio-SCSI bdev的构造方式与Virtio-Block的构造方式相同。

rpc.py construct\_virtio\_dev --dev-type scsi --trtype user --traddr /tmp/vhost.0 --vq-count 2 --vq-size 512 VirtioScsi0

rpc.py construct\_virtio\_dev --dev-type scsi --trtype pci --traddr 0000:01:00.0 VirtioScsi0

每个Virtio-SCSI设备最多可以导出64个名为VirtioScsi0t0~VirtioScsi0t63的块设备，每个SCSI设备一个LUN（LUN0）。以上2个命令将输出所有可见的bdev的名称。

可以使用以下命令删除Virtio-SCSI设备

rpc.py remove\_virtio\_bdev VirtioScsi0

删除Virtio-SCSI设备将破坏其所有bdev。

## 3.7 BlobFS（Blobstore文件系统）

BlobFS入门指南

### 3.7.1 RocksDB集成

按照<https://github.com/spdk/spdk>克隆并构建SPDK存储库

git clone https://github.com/spdk/spdk.git

cd spdk

./configure

make

将RocksDB存储库从SPDK GitHub fork克隆到单独的目录中。请务必查看spdk-v5.14.3分支代码。

cd ..

git clone -b spdk-v5.14.3 https://github.com/spdk/rocksdb.git

构建RocksDB。只有db\_bench基准测试工具与BlobFS集成在一起。

cd rocksdb

make db\_bench SPDK\_DIR = path/to/spdk

或者您也可以添加DEBUG\_LEVEL=0发布版本（需要打开USE\_RTTI）。

export USE\_RTTI = 1 && make db\_bench DEBUG\_LEVEL = 0 SPDK\_DIR = path / to / spdk

使用SPDK的gen\_nvme.sh脚本在配置文件中创建NVMe部分。

scripts/gen\_nvme.sh > /usr/local/etc/spdk/rocksdb.conf

验证配置文件是否指定了正确的NVMe SSD。如果您不想使用任何NVMe SSD进行RocksDB / SPDK测试，请将其从配置文件中删除。

确保为大页面分配至少5GB的内存。默认情况下，SPDK setup.sh脚本仅分配2GB。以下将分配5GB的大页面内存（除了将NVMe设备绑定到uio / vfio之外）。

HUGEMEM=5120 scripts/setup.sh

创建一个空的SPDK blobfs进行测试。

test/blobfs/mkfs/mkfs /usr/local/etc/spdk/rocksdb.conf Nvme0n1

此时，RocksDB已准备好使用SPDK进行测试。三个db\_bench参数用于配置SPDK：

1. spdk - 定义SPDK配置文件的名称。如果省略，RocksDB将使用默认的PosixEnv实现而不是SpdkEnv。（需要）
2. [spdk\_bdev](https://spdk.io/doc/structspdk__bdev.html) - 定义包含要用于测试的BlobFS的SPDK块设备的名称。（需要）
3. spdk\_cache\_size - 定义SPDK使用的用户空间缓存内存量。以兆字节（MB）表示。默认值为4096（4GB）。（可选的）

SPDK有一组脚本，这些脚本将针对各种工作负载运行db\_bench，并捕获性能和分析数据。主要脚本是test/blobfs/rocksdb/rocksdb.sh

### 3.7.2 FUSE插件

BlobFS提供了一个FUSE插件，用于将SPDK BlobFS作为内核文件系统安装，以便进行检查或调试。FUSE插件需要fuse3，并在系统上检测到fuse3时自动构建。

test/blobfs/fuse/fuse /usr/local/etc/spdk/rocksdb.conf Nvme0n1 /mnt/fuse

请注意，FUSE插件有一些限制 - 请参阅下面的列表。

**限制**

* 到目前为止，BlobFS主要使用RocksDB进行了测试，因此任何不同于RocksDB使用文件系统的用例都可能遇到问题。在初始版本发布后，BlobFS将在更广泛的用例中进行测试。
* 目前仅支持同步API。已开发出异步API但尚未经过全面测试，因此尚未成为公共接口的一部分。这将在以后的版本中添加。
* 文件重命名不是原子的。这将在以后的版本中修复。
* BlobFS目前仅支持没有目录支持的文件的平面命名空间。每个blob中，文件名当前存储为xattrs。这意味着文件名查找是O（n）操作。一个SPDK btree实现正在进行中，这将是未来版本中BlobFS目录支持的基础。
* 写入文件必须始终附加到文件的末尾。在将来的版本中将添加对文件中任何位置的写入支持。

## 3.8 JSON-RPC方法（略）

# 第四章 程序员指南

## 4.1. Blobstore程序员指南

**目标观众**

程序员指南适用于编写使用SPDK Blobstore的应用程序的开发人员。它旨在补充源代码，以全面了解如何将Blobstore集成到应用程序中，并提供有关Blobstore如何在幕后工作的高级洞察。它不是用作设计文档或API参考，在某些情况下，将讨论源代码片段和高级序列; 有关最新的源代码参考，请参阅[repo](https://github.com/spdk)。

### 4.1.1 介绍

Blobstore是一种持久的电源故障安全块分配器，旨在用作支持更高级别存储服务的本地存储系统，通常代替传统的文件系统。这些更高级别的服务可以是本地数据库或键/值存储（MySQL，RocksDB），它们可以是专用设备（SAN，NAS）或分布式存储系统（例如Ceph，Cassandra）。但是，它并非设计为通用文件系统，并且故意不符合POSIX标准。为了避免混淆，我们避免引用文件或对象，而使用术语“blob”。Blobstore旨在允许对名为“blobs”的块设备上的块组进行异步、非缓存、并行读取和写入。Blob通常很大，至少数百千字节为单位，并且总是底层块大小的倍数。

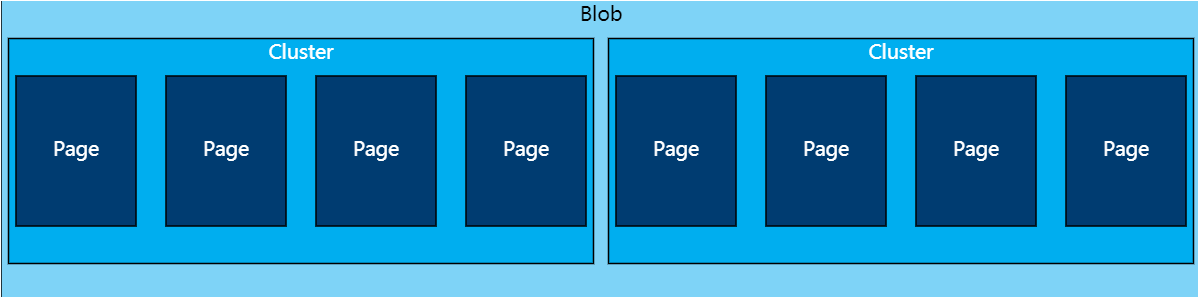
Blobstore主要设计用于在“下一代”媒体上运行，这意味着该设备支持快速随机读取和写入，无需任何背景垃圾收集。然而，在实践中，设计也将在NAND上运行良好。

### 4.1.2 运作理论

**抽象：**

Blobstore定义了存储抽象的层次结构，如下所示。

* 逻辑块：逻辑块由磁盘本身公开，磁盘编号从0到N，其中N是磁盘中的块数。逻辑块通常是512B或4KiB。
* 页面：页面定义为在Blobstore创建时定义的固定数量的逻辑块。组成页面的逻辑块始终是连续的。页面也从磁盘的开头编号，使得第一页的块是第0页，第二页是第1页，等等。页面的大小通常是4KiB，因此在实践中这是8或1个逻辑块。SSD必须能够执行至少页面大小的原子读取和写入。
* Cluster群集：群集是在Blobstore创建时定义的固定页数。组成群集的页面始终是连续的。群集也从磁盘的开头编号，其中群集0是第一个群集页面，群集1是第二组页面等。群集通常是1MiB大小，或256页。
* Blob：blob是一个有序的集群列表。应用程序操纵（创建，调整大小，删除等）Blob，并在电源故障和重新启动时保持不变。应用程序使用Blobstore提供的标识符来访问特定blob。通过指定从blob开头的偏移量，以页为单位读取和写入Blob。应用程序还可以以键/值对的形式存储元数据，每个blob我们将其称为xattrs（扩展属性）。
* Blobstore：已由基于Blobstore的应用程序初始化的SSD称为“Blobstore”。Blobstore拥有整个底层设备，该设备由私有Blobstore元数据区域和由应用程序管理的blob集合组成。



**原子性**

对于所有关于原子性的Blobstore操作，存在对底层设备的依赖性以保证至少一个页面大小的原子操作。这里的原子性可以指多个操作：

* 数据写入：对于数据写入的情况，原子性单位是一页。因此，如果正在进行大于一页的写入操作并且系统出现电源故障，则介质上的数据将以页面大小的粒度保持一致（如果在断电时单个页面正在更新中，该页面位置的数据将与电源恢复后的写入操作开始之前的数据相同。）
* Blob元数据更新：每个blob都有自己的元数据集（xattrs，size等）。出于性能原因，此元数据的副本保留在RAM中，并且仅在应用程序显式调用时或者在卸载Blobstore时才与磁盘版本同步。因此，例如，在调用同步它（稍后讨论）之前，设置xattr是不一致的，然而，这是以原子方式执行的。
* Blobstore元数据更新：Blobstore本身具有自己的元数据，就像每个blob元数据一样，它在RAM和磁盘上都有一个副本。但是，与每个blob元数据不同，Blobstore元数据区域不会通过blob同步调用保持一致，只有在通过API正确卸载Blobstore时才会同步它。因此，如果Blobstore元数据被更新（blob创建，删除，调整大小等）并且未正确卸载，则需要在下次加载时执行一些额外的步骤，这将花费比它如果有更多的时间。干净地关闭，但不会有任何不一致。

**回调**

Blobstore是回调驱动; 如果任何Blobstore API无法前进，它将不会阻塞，而是在此时返回控制，并在原始调用完成时调用API中提供的回调函数以及参数。回调将在与调用相同的线程上进行，稍后将在线程上进行。但是，某些API不提供回调参数; 在这些情况下，调用是完全同步的。使用回调的异步调用示例包括涉及磁盘IO的异步调用，例如，在IO完成之前需要进行一定量的轮询。

**后端支持**

Blobstore需要可以使用该bdev层集成的后备存储设备，或者直接将设备驱动程序集成到Blobstore。blobstore通过调用在初始化时提供给它的函数指针来对后备块设备执行操作。为方便起见，可以使用这些将I / O路由到bdev层的函数指针的实现bdev\_blob.c。或者，例如，可以绕过少量的bdev层开销直接集成SPDK NVMe驱动器。这些选项将在下一个关于示例的部分中进一步讨论。

**元数据操作**

由于Blobstore设计为无锁，因此需要将元数据操作隔离到单个线程，以避免在内存数据结构中进行锁定，从而在blob定义的布局（以及其他数据）上维护数据。在Blobstore中，这实现为the metadata thread并且被定义为应用程序与其进行元数据相关调用的线程。应用程序需要设置一个单独的线程来进行这些调用，并确保它不会将相关的IO操作与元数据操作混合在一起，即使它们位于不同的线程上也是如此。这将在“设计注意事项”部分中进一步讨论。

**线程**

例如，使用Blobstore和SPDK NVMe驱动程序的应用程序可以支持各种线程方案。最简单的是单线程应用程序，其中应用程序，Blobstore代码和NVMe驱动程序共享一个核心。在这种情况下，单个线程将用于提交元数据操作以及IO操作，并且由应用程序决定是否一次只发出一个元数据操作而不与受影响的IO操作混合。

**通道channel**

通道是SPDK范围的抽象，使用Blobstore考虑它们的最佳方式是，为了执行IO，它们是必需的。应用程序将对通道执行IO，并且最好将通道视为与线程1：1关联。

**Blob标识符**

当应用程序创建blob时，它不像许多其他类似存储系统那样提供名称，而是由Blobstore返回一个唯一标识符，它需要在后续API上使用它来对Blobstore执行操作。

### 4.1.3 设计注意事项

**初始化选项**

初始化Blobstore时，需要考虑多个配置选项。选项及其默认值为：

1. 群集cluster大小：默认情况下，此值为1MB。群集大小必须是页面大小的倍数，并且应根据应用程序在分配方面的使用模型进行选择。回想一下blob是由簇组成的，所以当blob被分配/解除分配或大小发生变化时，磁盘LBA将以簇大小的组进行操作。如果应用程序期望主要处理非常大（总是多GB）的blob，那么将簇大小更改为1GB可能是有意义的。
2. 元数据页面数量：默认情况下，Blobstore假设可以存在与元数据页面一样多的集群，这是元数据使用方面最糟糕的情况，可以在此处覆盖，但空间效率不高。
3. 最大同时元数据操作：确定为执行元数据操作预留了多少内部预分配的内存结构。不太可能需要更改此值（默认值为32）。
4. 每个通道的最大同时操作数：确定为通道操作留出多少内部预分配的内存结构。对该值的更改将取决于应用程序，并且最好通过典型使用模型的知识，对所使用的SSD类型的理解以及经验数据来确定。默认值为512。
5. Blobstore类型：此字段是应用程序使用的字符数组，需要识别此处找到的Blobstore是否适合声明。默认值为NULL，除非在使用相同磁盘的多个应用程序有可能无意中使用错误的Blobstore的环境中部署应用程序，否则无需设置此值。但是，它可以设置为任何有效的字符集。

**子页面大小的操作**

Blobstore只能执行页面大小的读/写操作。如果应用程序需要更精细的粒度，则必须自己适应。

**线程**

如前所述，Blobstore可以与应用程序共享一个线程，或者应用程序可以在资源限制内定义任意数量的线程，这是有意义的。必须遵循的基本考虑因素是：

* 元数据操作（名称中包含MD的API）应彼此隔离，因为受这些API影响的内存结构没有内部锁定。
* 元数据操作应与冲突的IO操作隔离（冲突的IO的示例将是读取/写入元数据操作解除分配的blob区域的IO）。
* 异步回调将始终在调用线程上进行。
* 无论在发布中涉及多少线程或哪些线程，都不能对IO排序做出任何假设。

**数据缓冲存储器**

与所有基于SPDK的应用程序一样，Blobstore要求使用SPDK API分配用于数据缓冲区的内存。

**错误处理**

异步Blobstore回调都包含应检查的错误号; 非零值表示和错误。如果适用，同步调用通常会返回错误值。

**异步API**

异步回调不会立即返回控制，但是在执行时不能阻塞的情况下不能再进行前进。因此，在回调完成之前，不能对异步调用的进度做出假设。

**Xattrs**

在Blobstore中设置和删除xattrs是元数据操作，xattrs存储在每个blob元数据中。因此，在进行blob同步调用并完成之前，不会持久保持xattrs。具有用于持久化每个blob元数据的步骤过程允许应用程序执行批量的xattr更新，例如，仅需要一个更昂贵的调用来同步和保持值。

**同步元数据**

如前所述，Blobstore中有两种类型的元数据，每个blob有一个元数据，Blobstore本身有一个全局元数据。只能通过API显式同步每个blob元数据。全局元数据在运行时将不一致，仅在正确关闭时同步。然而，不正确关闭的含义只会对下次启动造成性能损失，因为需要根据每个blob元数据的解析来重建全局元数据。为了保持一致的启动时间，始终通过API正确关闭Blobstore非常重要。

**迭代Blob**

示例代码和工具中包含有关如何遍历blob的多个示例。但是，值得注意的是，如果通过iter API浏览现有的blob，如果你的应用程序发现它正在查找的blob，则需要显式关闭它（因为它是由Blobstore在内部打开的）或者完整地遍历完整列表。

**超级Blob**

super blob只是一个blob ID，可以存储为全局元数据的一部分，作为“root”blob的一部分。应用程序可以选择使用此blob来存储它所需的任何信息，或者在理解Blobstore上的任何类型的结构时发现它们。

### 4.1.4 例子

在[repo中](https://github.com/spdk/spdk)有多个Blobstore用法示例：

1. Hello World：实际上hello\_blob.c这是一个单线程应用程序的一个非常基本的例子，它只是演示了非常基本的API。尽管Blobstore针对NVMe进行了优化，但此示例使用RAM磁盘（malloc）后端，以便可以在任何开发环境中轻松执行。malloc后端是一个bdev模块，因此该示例不仅使用SPDK框架，还使用该bdev层。
2. CLI：blobcli.c示例是命令行实用程序，不仅用作示例代码，还用作Blobstore本身的测试和开发工具。它也是一个简单的单线程应用程序，它依赖于SPDK框架和bdev层，但提供多种操作模式来完成一些实际任务。在命令模式下，它接受单次命令，如果有许多命令可以通过，这可能需要一些时间，因为每个命令都需要几秒钟等待DPDK初始化。因此它有一个shell模式，允许开发人员到达blob>提示然后使用简单的命令快速与Blobstore交互，这些命令包括从/向常规文件导入/导出blob的功能。最后，还有一种脚本模式可以自动执行一系列任务，同样适用于开发和/或测试类型活动。

### 4.1.5配置

Blobstore配置选项在“ [设计注意事项”](https://spdk.io/doc/blob.html#blob_pg_design)下的初始化选项部分中进行了描述。

### 4.1.6 组件细节

本节中的信息不一定与设计与Blobstore一起使用的应用程序有关，但是对内部构件的更多了解可能很有趣，并且也包含在那里，以便为那些想要为Blobstore工作本身做出贡献的人提供帮助。

**媒体格式**

Blobstore拥有整个存储设备。设备从头开始被分成簇，使得簇0从第一个逻辑块开始。

LBA 0 LBA N.

+ ----------+ ----------+ ----- + --------- +

| Cluster 0 | Cluster 1 | ......| Cluster N |

+ ----------+ ----------+ ----- + --------- +

Cluster 0是特殊的，具有以下格式，其中第0页是Cluster的第一页：

+--------+-------------------+

| Page 0 | Page 1 ... Page N |

+--------+-------------------+

| Super | Metadata Region |

| Block | |

+--------+-------------------+

超级块是位于分区开头的单个页面。它包含有关Blobstore的基本信息。元数据区域是集群0的剩余部分，并且可以扩展到其他集群。有关超级块和元数据区域的完整结构详细信息，请参阅最新的源代码。

每个blob在元数据区域内为其元数据分配一组非连续的页面。这些页面形成一个链表。列表中的第一页将在更新时写入，而所有其他页面将写入新的位置。这要求后备设备支持大于或等于页面大小的原子写入大小，以保证操作是原子操作。有关详细信息，请参阅原子性部分。

**序列和批处理**

内部Blobstore使用序列和批处理的概念分别以串行方式或并行方式将IO提交给底层设备。两者都使用以下结构定义：

struct spdk\_bs\_request\_set;

这些请求集基本上是簿记机制，以帮助Blobstore有效地处理相关的IO组。它们仅是内部构造，并且基于每个通道预先分配（前面讨论了通道）。当启动集（序列或批处理）时，它们将从通道关联的链表中删除，然后在完成时返回到列表。

**关键内部结构**

blobstore.h包含Blobstore内部工作的许多关键结构。这里仅审查一些值得注意的。请注意，这blobstore.h是一个内部头文件，Blobstore的头文件定义了公共API [blob.h](https://spdk.io/doc/blob_8h.html)。

struct spdk\_blob

这是一个内存中的数据结构，包含blob标识符，其当前状态和blob的可变元数据的两个副本等关键元素; 一个副本是当前元数据，另一个是写入磁盘的最后一个副本。

struct spdk\_blob\_mut\_data

这是一个每个blob结构，包括struct spdk\_blob实际定义blob本身的结构。它具有blob的大小和构成的特定信息（即为此blob分配了多少个簇以及哪些簇。）

struct spdk\_blob\_store

这是整个Blobstore的主要内存结构。它定义全局磁盘元数据区域并维护与整个系统相关的信息 - 初始化选项，如簇大小等。

struct spdk\_bs\_super\_block

超级块是一个磁盘上的结构，它包含刚才讨论的内存Blobstore结构中的所有相关信息以及人们期望在这里看到的其他元素，如签名，版本，校验和等。

**代码布局和通用约定**

一般而言，Blobstore.c布局中包含与描述性注释一起被阻止的相关功能组。例如，

/ \* START spdk\_bs\_md\_delete\_blob \* /

<完成blob删除的相关功能>

/ \* END spdk\_bs\_md\_delete\_blob \* /

在大多数情况下，遵循以下惯例：

以下划线开头的函数仅在内部调用

具有字母的函数或变量与cplset或callback completions相关

## 4.2. 块设备层编程指南

**目标观众**

本编程指南适用于编写使用SPDK bdev库访问块设备的应用程序的开发人员。

**介绍**

块设备是支持以固定大小的块读取和写入数据的存储设备。这些块通常为512或4096字节。设备可以是软件中的逻辑构造，或者对应于诸如NVMe SSD的物理设备。

块设备层由单个通用库lib/bdev和多个可选模块（作为单独的库）组成，这些模块实现了各种类型的块设备。通用库的公共头文件是[bdev.h](https://spdk.io/doc/bdev_8h.html)，它是与任何类型的块设备交互所需的全部API。本指南将介绍如何使用该API与bdev进行交互。有关实现bdev模块的指南，请参阅[编写自定义块设备模块](https://spdk.io/doc/bdev_module.html)。

除了为所有块设备提供通用抽象之外，bdev层还提供了许多有用的功能：

* 自动排队I / O请求以响应队列满或内存不足的情况
* 即使在I / O流量发生时，也可以热删除支持。
* 带宽和延迟等I / O统计信息
* 设备重置支持和I / O超时跟踪

**基本原语**

bdev API的用户与许多基本对象进行交互。

struct [spdk\_bdev](https://spdk.io/doc/structspdk__bdev.html)，本指南将其称为*bdev*，表示通用块设备。struct spdk\_bdev\_desc，此前称为*描述符*，表示给定块设备的句柄。描述符用于建立和跟踪使用底层块设备的权限，非常类似于UNIX系统上的文件描述符。对块设备的请求是异步的，由[spdk\_bdev\_io](https://spdk.io/doc/structspdk__bdev__io.html)对象表示。请求必须在关联的I / O通道上提交。[消息传递和并发中](https://spdk.io/doc/concurrency.html)描述了I / O通道的动机和设计。

Bdev可以是分层的，这样一些bdev通过将请求路由到其他bdev来服务I / O。这可用于实现缓存、RAID、逻辑卷管理等。将I/O路由到其他BDEV的BDEV通常称为虚拟BDEV，简称为*vbdevs*。

**初始化库**

bdev层依赖于头文件[include/spdk/thread.h](https://spdk.io/doc/thread_8h.html)抽象的通用消息传递基础[结构](https://spdk.io/doc/thread_8h.html)。有关完整说明，请参阅[消息传递和并发](https://spdk.io/doc/concurrency.html)。最重要的是，只能通过调用spdk\_thread\_create（）从已经分配了SPDK的线程调用bdev库。

从分配的线程中，可以通过调用[spdk\_bdev\_initialize（）](https://spdk.io/doc/bdev_8h.html#a854eb960c06597051071023a3d70e887)来初始化bdev库，这是一个异步操作。在调用完成回调之前，不能调用其他bdev库函数。同样，要拆除bdev库，请调用[spdk\_bdev\_finish（）](https://spdk.io/doc/bdev_8h.html#a7263aeb5393b46fa5af8e04760f62ea4)。

**发现块设备**

所有块设备都有一个简单的字符串名称。在任何时候，都可以通过调用[spdk\_bdev\_get\_by\_name（）](https://spdk.io/doc/bdev_8h.html#a89c759c53d028e3e408d19156f0e1eb1)来获取指向设备对象的指针，或者可以使用[spdk\_bdev\_first（）](https://spdk.io/doc/bdev_8h.html#a578b0a9d444ddbb23c8181d9a5ea4f46)和[spdk\_bdev\_next（）](https://spdk.io/doc/bdev_8h.html#aa1c0b9cb8dde1177df3442206cf87d2d)及其变体来迭代整个bdev集。

一些块设备也可以给出别名，也是字符串名称。别名的行为类似于符号链接 - 它们可以与实名互换使用以查找块设备。

**准备使用块设备**

为了将I / O请求发送到块设备，必须首先通过调用[spdk\_bdev\_open（）](https://spdk.io/doc/bdev_8h.html#aeacdfd5debd3917d84d18be0277203ff)来打开它。这将返回描述符。多个用户可能同时打开bdev，并且用户之间的读写协调必须由bdev层之外的某些更高级别的机制来处理。如果虚拟bdev模块声明了bdev，则打开具有写入权限的bdev可能会失败。虚拟bdev模块实现RAID或逻辑卷管理之类的逻辑，并将其I / O转发到较低级别的bdev，因此它们将这些较低级别的bdev标记为声称防止外部用户发出写入。

打开块设备时，可以提供可选的回调和上下文，如果删除了为块设备提供服务的底层存储，则将调用该回调和上下文。例如，当NVMe SSD热插拔时，将在物理NVMe SSD支持的bdev的每个打开描述符上调用remove回调。回调可以被认为是关闭打开描述符的请求，因此可以释放其他内存。当存在开放描述符时，不能拆除bdev，因此强烈建议提供回调。

当用户完成描述符时，他们可以通过调用[spdk\_bdev\_close（）](https://spdk.io/doc/bdev_8h.html#aaf5fc830851a2eb0aa34bbb40cf5eac6)来释放它。

描述符可以同时传递给多个线程并从中使用。但是，对于每个线程，必须通过调用[spdk\_bdev\_get\_io\_channel（）](https://spdk.io/doc/bdev_8h.html#ab5bb9e746b030d7c4c89cbcf60076c7e)获得单独的I / O通道。这将分配必要的每线程资源，以便在不接受锁定的情况下向bdev提交I / O请求。要释放频道，请调用[spdk\_put\_io\_channel（）](https://spdk.io/doc/thread_8h.html#a717571a5011db0cdd730498146f25f9a)。在销毁所有相关通道之前，不能关闭描述符。

**发送I / O**

一旦一个描述符和一个信道已经获得，I / O可以通过调用各种I / O功能提交诸如发送[spdk\_bdev\_read（）](https://spdk.io/doc/group__bdev__io__submit__functions.html#ga4b500ce84df1d2551f76e635c9dafbd5)。这些调用都将回调作为参数，稍后将使用[spdk\_bdev\_io](https://spdk.io/doc/structspdk__bdev__io.html)对象的句柄调用该参数。响应完成，用户必须调用[spdk\_bdev\_free\_io（）](https://spdk.io/doc/bdev_8h.html#a9ac23d38eb3b56b58197e0111ef68c6b)来释放资源。在此回调中，用户还可以使用函数[spdk\_bdev\_io\_get\_nvme\_status（）](https://spdk.io/doc/bdev_8h.html#afe43254747db3cd072dfb596be3bbf2f)和[spdk\_bdev\_io\_get\_scsi\_status（）](https://spdk.io/doc/bdev_8h.html#ac3c213b4317d5077197950253af1ac50)以他们选择的格式获取错误信息。

通过调用[spdk\_bdev\_read（）](https://spdk.io/doc/group__bdev__io__submit__functions.html#ga4b500ce84df1d2551f76e635c9dafbd5)或[spdk\_bdev\_write（）](https://spdk.io/doc/group__bdev__io__submit__functions.html#gaa740a114ef34d6a2f126d4e3a9dd9e9b)等函数来执行I / O提交。这些函数将一个指向内存区域的指针或一个描述将被传输到块设备的内存的分散集合列表SGL作为参数。必须通过[spdk\_dma\_malloc（）](https://spdk.io/doc/env_8h.html#a0874731c44ac31e4b14d91c6844a87d1)或其变体分配此内存。有关内存必须来自特殊分配池的完整说明，请参阅[从用户空间直接内存访问（DMA）](https://spdk.io/doc/memory.html)。在可能的情况下，内存中的数据将使用[直接内存访问](https://en.wikipedia.org/wiki/Direct_memory_access)直接传输到块设备。这意味着它不会被复制。

所有I / O提交功能都是异步和非阻塞的。它们不会因任何原因阻塞或停止线程。但是，I / O提交功能可能会以两种方式之一失败。首先，它们可能会立即失败并返回错误代码。在这种情况下，将不会调用提供的回调。其次，它们可能异步失败。在这种情况下，关联的[spdk\_bdev\_io](https://spdk.io/doc/structspdk__bdev__io.html)将传递给回调，它将报告错误信息。

某些I / O请求类型是可选的，给定的bdev可能不支持。要查询bdev以获取其支持的I / O请求类型，请调用[spdk\_bdev\_io\_type\_supported（）](https://spdk.io/doc/bdev_8h.html#a7cd5dff692dc162459801d5649fe7655)。

**重置块设备**

为了处理意外的故障情况，bdev库提供了一种通过调用[spdk\_bdev\_reset（）](https://spdk.io/doc/group__bdev__io__submit__functions.html#gaca574d3e94be6150343823dd2e7ecf46)来执行设备重置的机制。这会将消息传递给bdev存在I / O通道的每个其他线程，暂停它，然后将重置请求转发到底层bdev模块并等待完成。完成后，I / O通道将恢复，重置将完成。bdev模块中的特定行为是特定于模块的。例如，NVMe设备将删除所有队列对，执行NVMe重置，然后重新创建队列对并继续。最重要的是，无论设备类型如何，块设备的所有未完成的I / O都将在复位完成之前完成。

## 4.3 编写自定义块设备模块

目标观众

本编程指南适用于编写自己的块设备模块以与SPDK的bdev层集成的开发人员。有关如何使用bdev层的[指南](https://spdk.io/doc/bdev_pg.html)，请参见“ [块设备层编程指南”](https://spdk.io/doc/bdev_pg.html)。

### 4.3.1 介绍

块设备模块是SPDK，相当于传统操作系统中的设备驱动程序。该模块提供了一组函数指针，这些函数指针被调用以服务块设备I / O请求。SPDK提供了许多块设备模块，包括NVMe，RAM-disk和Ceph RBD。但是，一些用户希望自己编写以与自定义硬件或现有存储软件堆栈进行交互。本指南旨在准确演示如何编写模块。

### 4.3.2 创建一个新模块

块设备模块今天位于lib/bdev下的子目录中。目前无法将bdev模块的代码放在其他位置，但可以对构建系统进行更新，以便将来启用它。要创建模块，请使用单个C文件和Makefile添加新目录。一个很好的起点是复制现有的'null'bdev模块。

bdev模块将与之交互的主要接口位于[include/spdk/bdev\_module.h中](https://spdk.io/doc/bdev__module_8h.html)。在该标头中定义了一个宏，用于注册新的bdev模块 - SPDK\_BDEV\_MODULE\_REGISTER。此宏将指针[spdk\_bdev\_module](https://spdk.io/doc/structspdk__bdev__module.html)结构作为参数，用于注册新的bdev模块。

该[spdk\_bdev\_module](https://spdk.io/doc/structspdk__bdev__module.html)结构描述像初始化模块属性（module\_init）和拆卸（module\_fini）函数，该函数返回上下文大小（函数get\_ctx\_size） -将在用于每个I / O请求由该模块被分配临时空间，这将是一个回调每次另一个模块（examine\_config和examine\_disk）注册新的bdev时调用。有关更多详细信息，请查看struct [spdk\_bdev\_module](https://spdk.io/doc/structspdk__bdev__module.html)的文档。

**创建Bdevs**

通过调用[spdk\_bdev\_register（）](https://spdk.io/doc/bdev__module_8h.html#a5fef71f7761986298bb4aaf345736bb3)在模块中创建新的bdev 。模块必须分配一个struct [spdk\_bdev](https://spdk.io/doc/structspdk__bdev.html)，适当地填充它，然后将它传递给寄存器调用。要填写的最重要的字段是fn\_table，指向此数据结构：

/ \*

\*块设备后端的功能表。提供了一组允许与后端通信的API。主要命令是通过submit\_request对I/O进行读/写API调用。

\* /

struct [spdk\_bdev\_fn\_table](https://spdk.io/doc/structspdk__bdev__fn__table.html) {

int（\* [destruct](https://spdk.io/doc/structspdk__bdev__fn__table.html#a8a76c50a2b20205ec9fffe4afb9e3a15)）（void \* ctx）; / \*销毁后端块设备对象\* /

void（\* [submit\_request](https://spdk.io/doc/structspdk__bdev__fn__table.html#ae80a9002371becdc6fb1c13f8f5173bf)）（struct [spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html) \* ch，struct [spdk\_bdev\_io](https://spdk.io/doc/structspdk__bdev__io.html) \*）; / \*处理IO\* /

/ \*检查块设备是否支持特定的I / O类型。\* /

bool（\* [io\_type\_supported](https://spdk.io/doc/structspdk__bdev__fn__table.html#a20ec74cad9e56c5ac8ee9db239d8f46e)）（void \* ctx，enum [spdk\_bdev\_io\_type](https://spdk.io/doc/bdev_8h.html#a633029e24ab5ae4a689bffa2565a519f)）;

/ \*获取调用线程的特定bdev的I / O通道。\* /

struct [spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html) \*（\* get\_io\_channel）（void \* ctx）;

/ \*将特定于驱动程序的配置输出到JSON流。可选 - 可能为NULL。JSON写入上下文将使用打开的对象初始化，因此bdev驱动程序应该写一个名称（基于驱动程序名称），后跟一个JSON值（很可能是另一个嵌套对象）。 \* /

int（\* dump\_config\_json）（void \* ctx，struct spdk\_json\_write\_ctx \* w）;

/ \*获取每I / O通道的旋转时间（以微秒为单位）。 \*可选 - 可能为NULL。\* /

uint64\_t（\* [get\_spin\_time](https://spdk.io/doc/structspdk__bdev__fn__table.html#aafc336b4a42290b7e944fa0e73558a62)）（struct [spdk\_io\_channel](https://spdk.io/doc/structspdk__io__channel.html) \* ch）;

};

bdev模块必须实现这些函数回调。

destruct（）：当系统不再需要时，调用该功能以拆除设备。析构函数的作用取决于模块 - 它可能只是释放内存，或者它可能正在关闭一块硬件。

io\_type\_supported（）返回是否支持特定的I / O类型。可用的I / O类型是：

enum [spdk\_bdev\_io\_type](https://spdk.io/doc/bdev_8h.html#a633029e24ab5ae4a689bffa2565a519f) {

SPDK\_BDEV\_IO\_TYPE\_INVALID = 0，

SPDK\_BDEV\_IO\_TYPE\_READ，

SPDK\_BDEV\_IO\_TYPE\_WRITE，

SPDK\_BDEV\_IO\_TYPE\_UNMAP，

SPDK\_BDEV\_IO\_TYPE\_FLUSH，

SPDK\_BDEV\_IO\_TYPE\_RESET，

SPDK\_BDEV\_IO\_TYPE\_NVME\_ADMIN，

SPDK\_BDEV\_IO\_TYPE\_NVME\_IO，

SPDK\_BDEV\_IO\_TYPE\_NVME\_IO\_MD，

SPDK\_BDEV\_IO\_TYPE\_WRITE\_ZEROES，

};

对于最简单的BDEV模块，只有SPDK\_BDEV\_IO\_TYPE\_READ和SPDK\_BDEV\_IO\_TYPE\_WRITE有必要的。SPDK\_BDEV\_IO\_TYPE\_UNMAP通常称为“修剪”或“解除分配”，并且是将一组块标记为不再包含有效数据的请求。SPDK\_BDEV\_IO\_TYPE\_FLUSH是一个请求，使所有以前完成的写入持久。许多设备不需要冲洗。SPDK\_BDEV\_IO\_TYPE\_WRITE\_ZEROES就像常规写入一样，但不提供数据缓冲区（它只包含全0）。如果不支持，则通用bdev代码能够通过发送常规写入请求来模拟它。

SPDK\_BDEV\_IO\_TYPE\_RESET是中止所有I / O并将底层设备返回其初始状态的请求。在以某种方式完成所有I / O之前，请勿完成重置请求。

SPDK\_BDEV\_IO\_TYPE\_NVME\_ADMIN，SPDK\_BDEV\_IO\_TYPE\_NVME\_IO和，SPDK\_BDEV\_IO\_TYPE\_NVME\_IO\_MD都是通过SPDK bdev层传递原始NVMe命令的机制。它们是严格可选的，如果后备存储设备能够处理NVMe命令，那么实现它们可能是有意义的。

get\_io\_channel（）应返回I / O通道。有关I / O通道的详细说明，请参阅[消息传递和并发](https://spdk.io/doc/concurrency.html)。通用bdev层将为get\_io\_channel每个线程调用一次，缓存结果并将结果传递给submit\_request。它将使用它调用的线程的相应通道，叫做submit\_request（）。

submit\_request（），调用该函数向块设备提交I / O请求。

完成I / O请求后，模块必须调用[spdk\_bdev\_io\_complete（）](https://spdk.io/doc/bdev__module_8h.html#a3247574caabc2b31796861a10b2c9cab)。I / O不必在调用上下文中完成submit\_request。

### 4.3.3创建虚拟Bdev

如果块设备通过将I / O路由到其他块设备来处理I / O请求，则它们被视为虚拟设备。规范示例将是实现RAID的bdev模块。虚拟bdev的创建方式与常规bdev相同，但需要另外一步。模块可以使用[spdk\_bdev\_get\_by\_name（）](https://spdk.io/doc/bdev_8h.html#a89c759c53d028e3e408d19156f0e1eb1)查找它希望路由I / O的底层bdev ，其中字符串名称由用户在配置文件中或通过RPC提供。然后，通过打开bdev以获得描述符，并为bdev创建I / O通道（可能响应于get\_io\_channel回调），模块可以继续进行。最后一步是让模块使用其开放描述符来调用[spdk\_bdev\_module\_claim\_bdev（）](https://spdk.io/doc/bdev__module_8h.html#a43188aae62750556bba04f4ddc81fa94)，表明它正在消耗底层的bdev。这可以防止其他用户使用写入权限打开描述符。这有效地“促进”描述符写入独占，并且是仅对bdev模块可用的操作。

## 4.4 NVMe over Fabrics目标编程指南

目标观众

本编程指南适用于编写使用SPDK NVMe-oF目标库（lib/nvmf）的应用程序的开发人员。它旨在提供背景上下文，架构洞察和设计建议。本指南不会介绍如何使用SPDK NVMe-oF目标应用程序。有关如何按原样使用现有应用程序的指南，请参阅[NVMe over Fabrics Target](https://spdk.io/doc/nvmf.html)。

### 4.4.1 介绍

SPDK NVMe-oF目标库位于lib/nvmf。该库实现了创建NVMe-oF目标应用程序所需的所有逻辑。它用于实现示例NVMe-oF目标应用程序app/nvmf\_tgt，但旨在单独使用。

本指南的编写假设读者熟悉NVMe和NVMe over Fabrics。熟悉这些的最好方法是阅读它们的协议。

### 4.4.2 原语结构体

该库公开了许多原语 - 用户创建和交互的基本对象。他们是：

struct spdk\_nvmf\_tgt：NVMe-oF target。令人惊讶的是，这个概念没有出现在NVMe-oF规范中。SPDK将此定义为具有关联命名空间的子系统集合，以及传输集及其关联的网络连接。这将作为**target**在本指南中引用。

struct spdk\_nvmf\_subsystem：NVMe-oF子系统，由NVMe-oF规范定义。子系统包含名称空间和控制器并执行访问控制。这将作为**subsystem**在本指南中引用。

struct spdk\_nvmf\_ns：NVMe-oF命名空间，由NVMe-oF规范定义。命名空间是bdevs。有关SPDK bdev层的说明，请参见“ [块设备用户指南”](https://spdk.io/doc/bdev.html)。这将在本指南中称为**namespace**。

struct spdk\_nvmf\_qpair：NVMe-oF队列对，由NVMe-oF规范定义。这些将1：1映射到网络连接。这将作为**qpair**在本指南中引用。

struct spdk\_nvmf\_transport：NVMe-oF规范定义的网络结构的抽象。该规范旨在允许许多不同的网络结构，因此代码镜像并实现插件系统。目前，只有RDMA传输可用。这将在本指南中称为**transport**。

struct spdk\_nvmf\_poll\_group：可以作为一个单元轮询的网络连接集合的抽象。这是SPDK定义的概念，未出现在NVMe-oF规范中。通常，网络传输具有检查连接组上的传入数据的效率，而不是单独检查每个连接数据（例如epoll），因此轮询组为此提供了通用抽象。这将在本指南中称为**poll group**。

struct spdk\_nvmf\_listener：目标将接受新连接的网络地址。

struct spdk\_nvmf\_host：表示主机（发起方）系统的NVMe-oF NQN。这用于访问控制。

### 4.4.3 基础函数

所述NVMe-的目标库的用户开始通过创建使用目标[spdk\_nvmf\_tgt\_create（）](https://spdk.io/doc/nvmf_8h.html#a4c301a333bb0a069af54caf6e35e6d43)，设置了一组在其上的地址以接受通过调用连接[spdk\_nvmf\_tgt\_listen（）](https://spdk.io/doc/nvmf_8h.html#a2c31e57f4634daadf4238bf213a989d6)，然后使用创建一个子系统[spdk\_nvmf\_subsystem\_create（）](https://spdk.io/doc/nvmf_8h.html#ad1b7f34d31e34d784fb54b70b02bdb15)。

子系统以非活动状态开始，必须通过调用[spdk\_nvmf\_subsystem\_start（）](https://spdk.io/doc/nvmf_8h.html#a0584bea11782561b13355b89de25b6c0)来激活。子系统可以在运行时修改，但仅在处于暂停或非活动状态时。一个正在运行的子系统可以通过调用暂停[spdk\_nvmf\_subsystem\_pause（）](https://spdk.io/doc/nvmf_8h.html#a7a8b1ca226480e9582975bd03d3b232d)，并通过调用恢复[spdk\_nvmf\_subsystem\_resume（）](https://spdk.io/doc/nvmf_8h.html#aac21e50a3e893cf2629955dd553e9e32)。

当子系统处于非活动状态或暂停时，可以通过调用[spdk\_nvmf\_subsystem\_add\_ns（）](https://spdk.io/doc/nvmf_8h.html#a81b63417dea01e5a4fe15a4f3322c63c)将命名空间添加到子系统。命名空间是bdevs。有关SPDK bdev层的更多信息，请参见“ [块设备用户指南”](https://spdk.io/doc/bdev.html)。

可以通过调用[spdk\_bdev\_get\_by\_name（）](https://spdk.io/doc/bdev_8h.html#a89c759c53d028e3e408d19156f0e1eb1)获得[bdev](https://spdk.io/doc/bdev_8h.html#a89c759c53d028e3e408d19156f0e1eb1)。

一旦子系统存在且目标正在侦听地址，则轮询[spdk\_nvmf\_tgt\_accept（）](https://spdk.io/doc/nvmf_8h.html#aaffa201e13c1a8ee55c74fc0a928908e)可以接受新连接。

子系统的所有I / O都由轮询组驱动，轮询组轮询传入的网络I / O。可以通过调用[spdk\_nvmf\_poll\_group\_create（）](https://spdk.io/doc/nvmf_8h.html#a3b1de2ab69c992ce2c732c354a08238f)来创建轮询组。它们会自动请求在创建它们的线程上开始轮询。最重要的是，只能从创建它的线程访问一个轮询组。

当[spdk\_nvmf\_tgt\_accept（）](https://spdk.io/doc/nvmf_8h.html#aaffa201e13c1a8ee55c74fc0a928908e)检测到新连接时，它将构造一个新的struct spdk\_nvmf\_qpair对象，并用new\_qpair\_fn为每个新的qpair 调用用户提供的回调。响应此回调，用户必须通过调用[spdk\_nvmf\_poll\_group\_add（）](https://spdk.io/doc/nvmf_8h.html#a37eea41a3c3856e0ea2cd383af20183d)将qpair分配给轮询组。请记住，只能从创建它的线程访问一个轮询组，因此调用[spdk\_nvmf\_poll\_group\_add（）](https://spdk.io/doc/nvmf_8h.html#a37eea41a3c3856e0ea2cd383af20183d)可能需要将消息传递给相应的线程。

### 4.4.4访问控制

通过将允许的侦听地址和主机添加到子系统，在子系统级别执行访问控制（请参阅[spdk\_nvmf\_subsystem\_add\_listener（）](https://spdk.io/doc/nvmf_8h.html#a46521b556107a0ab58998adfeee8795f)和[spdk\_nvmf\_subsystem\_add\_host（）](https://spdk.io/doc/nvmf_8h.html#a19b3442b7eb75c234c3e3090d2b5add3)）。默认情况下，子系统不接受来自任何主机或任何已建立的侦听地址的连接。监听器和主机只能添加到非活动或暂停的子系统。

### 4.4.5发现子系统

根据NVMe-oF规范定义的发现子系统是为构建的每个NVMe-oF目标自动创建的。与发现子系统的连接的处理方式与任何其他子系统相同 - 创建新的qpairs以响应[spdk\_nvmf\_tgt\_accept（）](https://spdk.io/doc/nvmf_8h.html#aaffa201e13c1a8ee55c74fc0a928908e)，并且必须将它们分配给轮询组。

### 4.4.6 传输

NVMe-oF规范定义了多个网络传输（NVMe over Fabrics中的“Fabrics”），并且具有可扩展的系统，用于在未来添加新的结构。SPDK NVMe-oF目标库实现了用于网络传输的插件系统，以镜像规范。新传输必须实现的API位于lib/nvmf/transport.h中。在撰写本文时，仅实施了RDMA传输。

SPDK NVMe-oF目标旨在同时处理来自多个结构的I / O.

### 4.4.7选择线程模型

SPDK NVMe-oF目标库并不严格规定线程模型，但轮询组在创建它们的线程上执行所有轮询和I / O处理。鉴于此，在应用程序中使用的每个线程创建一个轮询组几乎总是有意义的。为响应[spdk\_nvmf\_tgt\_accept（）](https://spdk.io/doc/nvmf_8h.html#aaffa201e13c1a8ee55c74fc0a928908e)而创建的新qpairs 可以循环发送到轮询组。这就是SPDK NVMe-oF目标应用程序当前的功能。

用于将qpairs分发到轮询组的更高级算法是可能的。例如，NUMA感知算法将是对基本循环的改进，其中NUMA感知意味着将qpairs分配给在CPU核心上运行的轮询组，该CPU核心与网络适配器和存储设备位于相同的NUMA节点上。负载感知算法也可能有好处。

### 4.4.8 跨CPU核心扩展

轮询组将轮询其分配的qpair来接收传入的I / O请求。对于常规NVMe命令（例如READ和WRITE），I / O请求在初始线程上从开始到提交到后备存储设备的点处理，而不会中断。通过轮询后备存储设备并在轮询线程上处理完成来发现完成。常规NVMe命令（READ，WRITE等）不需要任何跨线程协调，因此不需要锁定。

NVMe ADMIN命令用于管理NVMe设备本身，可以修改子系统中的全局状态。例如，NVMe ADMIN命令可以执行命名空间管理，例如缩小命名空间。对于这些命令，子系统将通过向系统中的每个线程发送消息来暂时进入暂停状态。在此期间，任何针对子系统的线程上的所有新传入I / O都将排队。子系统完全暂停后，将发生状态更改，并将消息发送到每个线程以释放排队的I / O并恢复。管理命令很少见，因此这种协调方式比强制所有命令在I / O路径中获取锁定更可取。

### 4.4.9 零拷贝支持

对于RDMA传输，数据从RDMA NIC传输到主机存储器，然后将主机存储器传输到SSD（反之亦然），无需任何中间副本。数据永远不会从主机内存中的一个位置移动到另一个位置。未来的其他运输可能需要数据副本。

### 4.4.10 RDMA

SPDK NVMe-oF RDMA传输在libibverbs和rdmacm库之上实现，这些库在大多数Linux发行版中打包并可用。它不通过DPDK使用用户空间RDMA驱动程序堆栈。

为了扩展到大量连接，SPDK NVMe-oF RDMA传输为每个轮询组分配一个RDMA完成队列。分配给轮询组的所有新qpairs都有自己的RDMA发送和接收队列，但共享此公共完成队列。这允许轮询组轮询单个队列以查找传入消息，而不是遍历每个队列。

每个RDMA请求由状态机处理，该状态机将请求遍历多个状态。这样可以保持代码的有序性，并使所有的角落情况更加明显。

RDMA SEND，READ和WRITE操作是按彼此的顺序进行的，但RDMA RECV不一定按SEND确认进行排序。例如，可以在检测到包含NVMe完成的先前SEND的确认之前检测包含新NVMe-oF封壳的传入RDMA RECV消息。例如，在检测到包含NVME完成的上一次SEND的确认之前，可以检测到包含了RDMA RECV消息的新NVMe-oF数据包。这在整个队列深度上是有问题的，因为可能还没有空闲的请求结构。为了解决这个问题，RDMA请求结构分为两部分 - rdma\_recv和rdma\_request。新的RDMA RECV将始终获取一个空闲的rdma\_recv，但可能需要在队列中等待SEND确认才能获取完整的rdma\_request对象。

此外，RDMA NIC为READ / WRITE操作公开不同于SEND / RECV操作的队列深度。RDMA传输基于SEND / RECV操作限制报告可用队列深度，并将根据需要在软件中排队以适应（通常较低）READ / WRITE操作的限制。

## 4.5 Flash传输层

Flash传输层库在实现开放式通道接口的非块SSD之上提供块设备访问。它处理逻辑到物理地址映射，响应异步媒体管理事件，并管理碎片整理过程。

### 4.5.1 术语

#### 4.5.1.1 逻辑到物理地址映射

* 简写：L2P

包含逻辑地址（LBA）到磁盘上物理位置（PPA）的映射。LBA是连续的，其范围从0到表面块的数量（在器件形成期间计算备用块的数量并从可用地址空间中减去）。备用块考虑了在设备的整个生命周期中脱机的块，以及为数据[碎片整理](https://spdk.io/doc/ftl.html#ftl_reloc)提供必要的缓冲区。

#### 4.5.1.2 Band

Band描述了一组块（chunk），每个块都属于不同的并行单元。对band的所有写入都遵循相同的模式 - 一批逻辑块写入一个块，另一批写入下一个块，依此类推。这确保了写操作的并行性，因为它们可以在不同的块上独立执行。每个band都会跟踪它所包含的LBA以及它们的有效性，因为一些数据将通过后续写入同一逻辑地址而失效。通过从最旧的band到最新的band按顺序读取此信息，可以从SSD恢复L2P映射。

+--------------+ +--------------+ +--------------+

band 1 | chunk 1 +--------+ chk 1 +---- --- --- --- --- ---+ chk 1 |

+--------------+ +--------------+ +--------------+

band 2 | chunk 2 +--------+ chk 2 +---- --- --- --- --- ---+ chk 2 |

+--------------+ +--------------+ +--------------+

band 3 | chunk 3 +--------+ chk 3 +---- --- --- --- --- ---+ chk 3 |

+--------------+ +--------------+ +--------------+

| ... | | ... | | ... |

+--------------+ +--------------+ +--------------+

band m | chunk m +--------+ chk m +---- --- --- --- --- ---+ chk m |

+--------------+ +--------------+ +--------------+

| ... | | ... | | ... |

+--------------+ +--------------+ +--------------+

parallel unit 1 pu 2 pu n

地址映射和有效映射以及其他几个东西（例如，它所属的设备的UUID，浮出水面的LBA的数量，band的序列号等），band的元数据的一部分。元数据分为两部分：

* 头部，包含打开band时已知的信息（设备的UUID，band的序列号等），位于band的起始块，
* 尾部，包含地址图和有效地图，位于band的末尾。

头元数据 band的数据 尾元数据

+ ------------------------- + ------------------+ ------ + ---------------------- +

| chk 1 | ... | chk n | ... | ... | chk 1 | ... | | ... | chk m-1 | chk m |

| lbk 1 | | lbk 1 | | | lbk x | | | | lblk y | lblk y |

+ ------------------------- + ----------------- + ------ + ---------------------- +

Band 按顺序写入（以前面描述的方式）。在可以写入band之前，需要擦除其所有块。在此期间，band被认为处于PREP状态。在此之后，band转换到OPENING正在写入头元数据的状态。然后band移动到该OPEN状态，并且可以将实际用户数据写入band。填充完整个可用空间后，将写入尾部元数据并将band转换为CLOSING状态。当完成band成为CLOSED。

#### 4.5.1.3 环写缓冲区

简写：RWB

由于SSD可能支持的最小写入大小可以是块大小的倍数，因此为了支持对单个块的写入，需要缓冲数据。写缓冲区是此问题的解决方案。它由许多预先分配的称为批处理的缓冲区组成，每个缓冲区的大小允许单次传输到SSD。单个批次分为块大小的缓冲区条目。

写的缓冲区

+ ------------------------------------- +

|批次1 |

| + --------------------------------- + |

| | rwb | rwb | ... | rwb | |

| | entry 1 | entry 2 | | entry n | |

| + --------------------------------- + |

+ ------------------------------------- +

| ...... |

+ ------------------------------------- +

|批量m |

| + --------------------------------- + |

| | rwb | rwb | ... | rwb | |

| | entry 1 | entry 2 | | entry n | |

| + --------------------------------- + |

+ ------------------------------------- +

当调度写入时，它需要为每个块获取一个条目并将数据复制到该缓冲区。一旦复制了所有块，就可以将写入信号发送给用户。在此期间，对rwb已填充的批次进行轮询，如果找到，则将其发送到SSD。完成该操作后，可以释放整批。在数据存在的整个时间内rwb，L2P指向缓冲区条目而不是SSD上的位置。这允许服务来自缓冲区的读取请求。

#### 4.5.1.4 碎片整理和重定位

速记：碎片整理，重定位

由于对同一LBA的写入使其先前的物理位置无效，因此某个band上的某些块可能包含基本上浪费空间的旧数据。由于无法覆盖已写入的块，因此该数据将保留在那里直到整个块（chunk）被重置。这可能会产生一种情况，其中所有band都包含一些有效数据，并且不能擦除任何band，因此不再执行任何写操作。因此，需要一种机制来移动有效数据并使整个band无效，以便可以重复使用它们。

band band

+-----------------------------------+ +-----------------------------------+

| \*\* \* \* \*\*\* \* \*\*\* \* \* | | |

|\*\* \* \* \* \* \* \* \*| +----> | |

|\* \*\*\* \* \* \* | | |

+-----------------------------------+ +-----------------------------------+

有效块（blocks）标有星号'\*'。

数据重定位的另一个原因可能是来自SSD的事件告诉我们如果数据没有重新定位，数据可能会损坏。这可能是由于它的老化（如果它是很久以前写的）或由于读干扰（媒体特性，在读操作期间导致相邻块的损坏）而发生的。

调用负责数据重定位的模块reloc。当选择频带进行碎片整理或接收到ANM（异步NAND管理）事件时，将适当的块标记为需要移动。该reloc模块采用标记了一些此类块的频段，检查其有效性，如果它们仍然有效，则复制它们。

选择用于碎片整理的波段取决于几个因素：其有效比率（1）（有效块与所有用户块的比例），其年龄（2）（何时写入）及其块的写入计数/磨损等级指数（ 3）（乐队写了多少次）。比率（1）越低，其年龄（2）越高，其写入次数（3）越低，为磁盘碎片整理选择频带的可能性越大。

### 4.5.2 使用方法

#### 4.5.2.1 先决条件

为了使用FTL模块，需要一个开放式通道SSD。获得一个的最简单方法是使用QEMU模拟它。带有提供开放通​​道支持的补丁的QEMU可以在[spdk-3.0.0](https://github.com/spdk/qemu/tree/spdk-3.0.0)分支上的SPDK QEMU fork上[找到](https://github.com/spdk/qemu/tree/spdk-3.0.0)。

#### 4.5.2.2 配置QEMU

为了模拟开放式通道设备，QEMU需要描述SSD特性和几何形状的参数：

* serial - 序列号，
* lver - 版本的OCSSD标准（0 - 禁用，1 - “1.2”，2 - “2.0”），libftl仅支持2.0，
* lba\_index - 默认LBA格式。可能的值（libftl仅支持lba\_index> = 3）：

|  |  |  |
| --- | --- | --- |
| lba\_index | 数据 | 元数据 |
| 0 | 512B | 0B |
| 1 | 512B | 8B |
| 2 | 512B | 16B |
| 3 | 4096B | 0B |
| 4 | 4096B | 64B |
| 5 | 4096B | 128B |
| 6 | 4096B | 16B |

* lnum\_ch - groups数量，
* lnum\_lun - 并行单元数
* lnum\_pln - 平面数（所有平面的逻辑块构成一个块）
* lpgs\_per\_blk - 每块的页数（最小可编程单位）
* lsecs\_per\_pg - 页面中的扇区数
* lblks\_per\_pln - 并行单元中的块（chunk）数
* laer\_thread\_sleep - 异步事件之间的超时（以毫秒为单位），请求主机根据媒体反馈重新定位数据
* lmetadata - 元数据文件

有关可用选项的更多详细说明，请参阅hw/block/nvme.cQEMU存储库中的文件。

例：

$/path/to/qemu [OTHER PARAMETERS] -drive format = raw，file =/path/to/data/file，if = none，id = myocssd0

-device nvme，drive = myocssd0，serial = deadbeef，lver = 2，lba\_index = 3，lnum\_ch = 1，lnum\_lun = 8，lnum\_pln = 4，

lpgs\_per\_blk = 1536，lsecs\_per\_pg = 4，lblks\_per\_pln = 512，lmetadata = /path/to/MD/file

在上面的示例中，创建了具有1个通道，8个并行单元，每个并行单元512个块的设备，每个块中的24576（lnum\_pln\* lpgs\_per\_blk\* lsecs\_per\_pg）个逻辑块，逻辑块为4096B。因此，数据文件的大小至少需要为384G（8 \* 512 \* 24576 \* 4096B），并且可以使用以下命令创建：

$ fallocate -l 384G /path/to/data/file

#### 4.5.2.3 配置SPDK

要验证驱动器是否正确模拟，可以检查NVMe识别应用程序的输出（假设scripts/setup.sh之前已调用并且该驱动程序已针对该设备进行了更改）：

$ examples/nvme/identify/identify

=====================================================

NVMe Controller at 0000:00:0a.0 [1d1d:1f1f]

=====================================================

Controller Capabilities/Features

================================

Vendor ID: 1d1d

Subsystem Vendor ID: 1af4

Serial Number: deadbeef

Model Number: QEMU NVMe Ctrl

... other info ...

Namespace OCSSD Geometry

=======================

OC version: maj:2 min:0

... other info ...

Groups (channels): 1

PUs (LUNs) per group: 8

Chunks per LUN: 512

Logical blks per chunk: 24576

... other info ...

与其他bdev类似，可以基于配置文件或通过RPC创建FTL bdev。两个接口都需要相同的参数，这些参数由RPC调用--help选项描述construct\_ftl\_bdev，它们是：

* bdev的名字
* 设备的传输类型（例如PCIe）
* 设备的运输地址（例如00:0a.0）
* 并联单位范围
* FTL设备的UUID（如果要从SSD恢复FTL）

示例配置：

[FTL]

TransportID“trtype：PCIe traddr：00：0a.0”nvme0“0-3”00000000-0000-0000-0000-000000000000

TransportID“trtype：PCIe traddr：00：0a.0”nvme1“4-5”e9825835-b03c-49d7-bc3e-5827cbde8a88

以上将导致创建两个设备：

* nvme0上00:0a.0使用并行单元0-3，从头开始创建
* nvme1 使用并行单元4-5在同一设备上，使用提供的UUID从SSD恢复

使用以下两个RPC调用可以实现相同的目的：

$ scripts/rpc.py construct\_ftl\_bdev -b nvme0 -l 0-3 -a 00:0a.0

{

"name": "nvme0",

"uuid": "b4624a89-3174-476a-b9e5-5fd27d73e870"

}

$ scripts/rpc.py construct\_ftl\_bdev -b nvme1 -l 0-3 -a 00:0a.0 -u e9825835-b03c-49d7-bc3e-5827cbde8a88

{

"name": "nvme1",

"uuid": "e9825835-b03c-49d7-bc3e-5827cbde8a88"

}

## 4.6 GDB宏用户指南

### 4.6.1 介绍

使用gdb调试spdk应用程序时，我们可能需要查看列表中的数据结构，例如有关bdevs或线程的信息。

例如，如果我有几个bdev，并且我希望通过名称'test\_vols3'获取有关bdev的信息，我将需要手动迭代列表，如下所示：

(gdb) p g\_bdev\_mgr->bdevs->tqh\_first->name

$5 = 0x7f7dcc0b21b0 "test\_vols1"

(gdb) p g\_bdev\_mgr->bdevs->tqh\_first->internal->link->tqe\_next->name

$6 = 0x7f7dcc0b1a70 "test\_vols2"

(gdb) p

g\_bdev\_mgr->bdevs->tqh\_first->internal->link->tqe\_next->internal->link->tqe\_next->name

$7 = 0x7f7dcc215a00 "test\_vols3"

(gdb) p

g\_bdev\_mgr->bdevs->tqh\_first->internal->link->tqe\_next->internal->link->tqe\_next

$8 = (struct spdk\_bdev \*) 0x7f7dcc2c7c08

在这个阶段，我们可以开始查看我们知道的地址为0x7f7dcc2c7c08的bdev的相关字段。

如果有100个bdev，这可能有些麻烦，我们需要的是列表中的第56个......

相反，我们可以使用gdb宏来获取有关所有设备的信息。

例子：

打印bdevs：

(gdb) spdk\_print\_bdevs

SPDK object of type struct spdk\_bdev at 0x7f7dcc1642a8

((struct spdk\_bdev\*) 0x7f7dcc1642a8)

name 0x7f7dcc0b21b0 "test\_vols1"

---------------

SPDK object of type struct spdk\_bdev at 0x7f7dcc216008

((struct spdk\_bdev\*) 0x7f7dcc216008)

name 0x7f7dcc0b1a70 "test\_vols2"

---------------

SPDK object of type struct spdk\_bdev at 0x7f7dcc2c7c08

((struct spdk\_bdev\*) 0x7f7dcc2c7c08)

name 0x7f7dcc215a00 "test\_vols3"

---------------

按名称查找bdev：

(gdb) spdk\_find\_bdev test\_vols1

test\_vols1

SPDK object of type struct spdk\_bdev at 0x7f7dcc1642a8

((struct spdk\_bdev\*) 0x7f7dcc1642a8)

name 0x7f7dcc0b21b0 "test\_vols1"

打印spdk线程：

(gdb) spdk\_print\_threads

SPDK object of type struct spdk\_thread at 0x7fffd0008b50

((struct spdk\_thread\*) 0x7fffd0008b50)

name 0x7fffd00008e0 "reactor\_1"

IO Channels:

SPDK object of type struct spdk\_io\_channel at 0x7fffd0052610

((struct spdk\_io\_channel\*) 0x7fffd0052610)

name

ref 1

device 0x7fffd0008c80 (0x7fffd0008ce0 "nvmf\_tgt")

---------------

SPDK object of type struct spdk\_io\_channel at 0x7fffd0056cd0

((struct spdk\_io\_channel\*) 0x7fffd0056cd0)

name

ref 2

device 0x7fffd0056bf0 (0x7fffd0008e70 "test\_vol1")

---------------

SPDK object of type struct spdk\_io\_channel at 0x7fffd00582e0

((struct spdk\_io\_channel\*) 0x7fffd00582e0)

name

ref 1

device 0x7fffd0056c50 (0x7fffd0056cb0 "bdev\_test\_vol1")

---------------

SPDK object of type struct spdk\_io\_channel at 0x7fffd00583b0

((struct spdk\_io\_channel\*) 0x7fffd00583b0)

name

ref 1

device 0x7fffd0005630 (0x7fffd0005690 "bdev\_mgr")

---------------

打印nvmf子系统：

(gdb) spdk\_print\_nvmf\_subsystems

SPDK object of type struct spdk\_nvmf\_subsystem at 0x7fffd0008d00

((struct spdk\_nvmf\_subsystem\*) 0x7fffd0008d00)

name "nqn.2014-08.org.nvmexpress.discovery", '\000' <repeats 187 times>

nqn "nqn.2014-08.org.nvmexpress.discovery", '\000' <repeats 187 times>

ID 0

---------------

SPDK object of type struct spdk\_nvmf\_subsystem at 0x7fffd0055760

((struct spdk\_nvmf\_subsystem\*) 0x7fffd0055760)

name "nqn.2016-06.io.spdk.umgmt:cnode1", '\000' <repeats 191 times>

nqn "nqn.2016-06.io.spdk.umgmt:cnode1", '\000' <repeats 191 times>

ID 1

### 4.6.2 加载gdb宏

将gdb宏复制到要调试的主机。最好将文件复制到PYTHONPATH中的某个位置，或将目标目录添加到PYTHONPATH。这不是强制性的，可以解决，但在将模块加载到gdb时可以节省一些步骤。

从gdb开始，打开应用程序核心，调用python并加载模块。

在下面的示例中，我将宏复制到了不在PYTHONPATH中的/ tmp目录，因此我不得不手动将目录添加到路径中。

(gdb) python

>import sys

>sys.path.append('/tmp')

>import gdb\_macros

>end

(gdb) spdk\_load\_macros

### 4.6.3 使用gdb数据目录

在大多数系统上，数据目录是/ usr / share / gdb。应该将python脚本复制到数据目录下的python / gdb / function（或python / gdb / command）目录中，例如/ usr / share / gdb / python / gdb / function。

如果python脚本在那里，那么启动gdb时你唯一需要做的就是键入“spdk\_load\_macros”。

### 4.6.4 使用.gdbinit加载宏

.gdbinit也可用于在启动gdb之前自动运行上面的手动步骤。

例如.gdbinit：

source /opt/km/install/tools/gdb\_macros/gdb\_macros.py

启动gdb时，您仍然需要调用spdk\_load\_macros。

### 4.6.5 为什么我们需要显式调用spdk\_load\_macros

原因是宏需要使用spdk提供的全局变量来迭代spdk列表并构建列表对象的可迭代表示。如果这些数据不可用，这将导致错误，如果使用gdb的原因不是调试spdk核心转储，则很可能会导致错误。

在下面的示例中，我尝试在全局变量不可用时加载宏，导致gdb无法加载gdb\_macros：

(gdb) spdk\_load\_macros

Traceback (most recent call last):

File "/opt/km/install/tools/gdb\_macros/gdb\_macros.py", line 257, in invoke

spdk\_print\_threads()

File "/opt/km/install/tools/gdb\_macros/gdb\_macros.py", line 241, in \_\_init\_\_

threads = SpdkThreads()

File "/opt/km/install/tools/gdb\_macros/gdb\_macros.py", line 234, in \_\_init\_\_

super(SpdkThreads, self).\_\_init\_\_('g\_threads', SpdkThread)

File "/opt/km/install/tools/gdb\_macros/gdb\_macros.py", line 25, in \_\_init\_\_

['tailq'])

File "/opt/km/install/tools/gdb\_macros/gdb\_macros.py", line 10, in \_\_init\_\_

self.list = gdb.parse\_and\_eval(self.list\_pointer)

RuntimeError: No symbol table is loaded. Use the "file" command.

Error occurred in Python command: No symbol table is loaded. Use the "file"

command.

### 4.6.6 以上可用的宏总结

* spdk\_load\_macros：加载宏（使用-reload以重新加载它们）
* spdk\_print\_bdevs：有关bdevs的信息
* spdk\_find\_bdev：找一个bdev（子串搜索）
* spdk\_print\_io\_devices：有关io设备的信息
* spdk\_print\_nvmf\_subsystems：有关nvmf子系统的信息
* spdk\_print\_threads：有关线程的信息

### 4.6.7 添加新宏

列表迭代宏通常由3层构建：

* SpdkPrintCommand：继承自gdb.Command并调用列表迭代
* SpdkTailqList：根据tailq成员实现执行tailq列表的迭代
* SpdkObject：提供str函数，以便列表迭代可以打印对象

其他有用的对象：

* SpdkNormalTailqList：表示将“tailq”作为tailq对象的列表
* SpdkArr：对数组的迭代（而不是链表）

## 4.7 SPDK “Reduce”块压缩算法

### 4.7.1 介绍

SPDK“减少”块压缩方案基于使用SSD来存储压缩的存储块和用于元数据的持久存储器。该元数据包括用户请求到SSD上的压缩块的逻辑块的映射。本文档中描述的方案是通用的，不依赖于任何特定的块设备框架，例如SPDK块设备（bdev）框架。该算法将在名为“libreduce”的库中实现。可以在此库之上构建更高级别的软件模块，以在特定的块设备框架中创建和呈现块设备。对于SPDK，bdev\_reduce模块将充当libreduce库的包装器，将压缩块设备呈现为SPDK bdev。

此方案仅描述压缩块（compressed block）如何存储在SSD上以及用于跟踪这些压缩块的元数据。它依赖于更高的软件模块来执行压缩算法本身。对于SPDK，bdev\_reduce模块将利用DPDK compressdev框架代表libreduce库执行压缩和解压缩。

（请注意，在某些情况下，存储块可能不可压缩，或者压缩不足以实现压缩的节省。在这些情况下，数据可以未压缩存储在磁盘上。短语“压缩存储块”包括这些未压缩的块。）

压缩块设备是构建在类似大小的后备存储设备之上的逻辑实体。后备存储设备必须进行精简配置，以便通过本文档后面所述的原因实现压缩的任何节省。该算法不直接了解后备存储设备的实现，只是它总是使用后备存储设备上可用的编号最小的块。这将确保在精简配置的后备存储设备上使用此算法时，在实际需要之前不会分配块。

后备存储设备的大小必须适用于最糟糕的情况，即无法压缩数据。在这种情况下，后备存储设备的大小将与压缩块设备相同。由于此算法通过永不覆盖数据来确保原子性，因此在更新关联的元数据之前，需要一些额外的后备存储来临时存储正在进行的写入的数据。

后备存储设备的存储将以4KB为单位进行分配，读取和写入，以获得最佳NVMe性能。这些4KB单元称为“后备IO单元”。它们的索引从0到N-1，索引称为“支持IO单位索引”。开始时，完整的索引集代表“空闲后备IO单元列表”。

压缩块设备（compressed block device）以块为单位压缩和解压缩数据，其中块是至少两个4KB后备IO单元的倍数。每个块的后备IO单元数决定了块大小，并在创建压缩块设备时指定。块在1之间消耗大量4KB后备IO单元，并且块中的4KB单元数量。例如，一个16KB的块会消耗1,2,3或4个备用IO单元。支持IO单元的数量取决于块能够被压缩的程度。与块相关联的磁盘上的块存储在永久存储器中的“块映射”中。每个块映射由N个64位值组成，其中N是块中的最大后备IO单元数。每个64位值对应一个后备IO单元索引。特殊值（例如，2 ^ 64-1）用于支持由于压缩而不需要的IO单元。分配的块映射的数量等于压缩块设备的大小除以其块大小，加上一些额外的块映射。这些额外的块映射用于确保写入时的原子性，稍后将在本文档中进行说明。首先，所有块映射都代表“空闲块映射列表”。

最后，压缩块设备的逻辑视图由“逻辑映射”表示。逻辑映射是块偏移到压缩块设备到相应块组映射的映射。逻辑映射中的每个条目都是64位值，表示关联的块映射。如果没有关联的块映射，则使用特殊值（UINT64\_MAX）。通过将字节偏移除以块大小来确定映射以获得索引，该索引用作块映射条目阵列中的数组索引。开始时，逻辑映射中的所有条目都没有关联的块映射。请注意，虽然对后备存储设备的访问是以4KB为单位，但逻辑视图可能允许4KB或512B单元访问，并且应该执行类似的操作。

### 4.7.2 例子

为了说明这个算法，我们将以非常小的比例使用一个真实的例子。

压缩块设备的大小为64KB，块大小为16KB。这将实现以下目标：

* “备份存储”将包含一个80KB精简配置的逻辑卷。这对应于压缩块设备的64KB大小，再加上额外的16KB，以便在最坏情况下的压缩场景下处理额外的写操作。
* “空闲后备IO单元列表”将包含索引0到19（含）。这些代表后备存储中的20个4KB IO单元。
* “块映射”的大小为32字节。这相当于每个块（16KB / 4KB）有4个后备IO单元，每个后备IO单元索引对应8B（64b）。
* 将在160B的持久性内存中分配5个块映射。这对应于压缩块设备中4个块的4个块映射（64KB / 16KB），以及覆盖现有块时使用的额外块映射。
* “空闲后备IO单元列表”将包含索引0到4（包括）。这些代表5个分配的块映射。
* “逻辑映射”将在32B的持久性存储器中分配。这对应于压缩块设备中的4个块的4个条目和每个条目的8B（64b）。

在这些示例中，值“X”将表示上述特殊值（2 ^ 64-1）。

**初始的创建**

+--------------------+

Backing Device | |

+--------------------+

Free Backing IO Unit List 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

+------------+------------+------------+------------+------------+

Chunk Maps | | | | | |

+------------+------------+------------+------------+------------+

Free Chunk Map List 0, 1, 2, 3, 4

+---+---+---+---+

Logical Map | X | X | X | X |

+---+---+---+---+

**写入偏移量为32KB的16KB**

* 在逻辑映射中找到相应的索引。偏移量32KB除以块大小（16KB）为2。
* 逻辑映射中的条目2是“X”。这意味着尚未写入此16KB的任何部分。
* 在内存中分配16KB缓冲区
* 将传入的16KB数据压缩到此分配的缓冲区中
* 假设此数据压缩为6KB。这需要2个4KB的后备IO单元。
* 从可用后备IO单元列表中分配2个块（0和1）。始终使用可用后备IO单元列表中编号最小的条目 - 这可确保在保留后备存储的精简配置逻辑卷中不分配不必要的后备存储。
* 将6KB数据写入后备IO单元0和1。
* 从空闲块映射列表中分配块映射（0）。
* 将（0，1，X，X）写入块图。这表示仅使用2个后备IO单元来存储16KB数据。
* 将块映射索引写入逻辑映射中的条目2。

+--------------------+

Backing Device |01 |

+--------------------+

Free Backing IO Unit List 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

+------------+------------+------------+------------+------------+

Chunk Maps | 0 1 X X | | | | |

+------------+------------+------------+------------+------------+

Free Chunk Map List 1, 2, 3, 4

+---+---+---+---+

Logical Map | X | X | 0 | X |

+---+---+---+---+

**写入偏移量为8KB的4KB**

* 在逻辑映射中找到相应的索引。偏移量8KB除以块大小为0。
* 逻辑映射中的条目0是“X”。这意味着尚未写入此16KB的任何部分。
* 写入不是针对整个16KB块，因此我们必须为源数据分配16KB块大小的缓冲区。
* 将传入的4KB数据复制到此16KB缓冲区的偏移量8KB。将16KB缓冲区的其余部分归零。
* 分配一个16KB的目标缓冲区。
* 将16KB源数据缓冲区压缩到16KB目标缓冲区
* 假设此数据压缩为3KB。这需要1个4KB的后备IO单元。
* 从可用后备IO单元列表中分配1个块（2）。
* 将3KB数据写入块2。
* 从空闲块映射列表中分配块映射（1）。
* 将（2，X，X，X）写入块图。
* 将块映射索引写入逻辑映射中的条目0。

+--------------------+

Backing Device |012 |

+--------------------+

Free Backing IO Unit List 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

+------------+------------+------------+------------+------------+

Chunk Maps | 0 1 X X | 2 X X X | | | |

+------------+------------+------------+------------+------------+

Free Chunk Map List 2, 3, 4

+---+---+---+---+

Logical Map | 1 | X | 0 | X |

+---+---+---+---+

**以16KB的偏移量读取16KB**

* 偏移16KB映射到逻辑映射中的索引1。
* 逻辑映射中的条目1是“X”。这意味着尚未写入此16KB的任何部分。
* 由于没有数据写入此块，因此返回全0以满足读取I / O.

**写入偏移量为4KB的4KB**

* 偏移4KB映射到逻辑映射中的索引0。
* 逻辑映射中的条目0是“1”。由于我们没有覆盖整个块，我们必须进行读 - 修改 - 写。
* 块映射1仅指定一个后备IO单元（2）。分配一个16KB的缓冲区并将块2读入其中。这将被称为压缩数据缓冲区。请注意，分配16KB而不是4KB，以便我们可以重用此缓冲区来保存稍后将写回磁盘的压缩数据。
* 为此块的未压缩数据分配16KB缓冲区。将压缩数据缓冲区中的数据解压缩到此缓冲区中。
* 将传入的4KB数据复制到未压缩数据缓冲区的4KB偏移量。
* 将16KB未压缩数据缓冲区压缩到压缩数据缓冲区中。
* 假设此数据压缩为5KB。这需要2个4KB的后备IO单元。
* 从可用后备IO单元列表中分配块3和4。
* 将5KB数据写入块3和4。
* 从空闲块映射列表中分配块映射2。
* 将（3，4，X，X）写入块映射2. 请注意，此时，逻辑映射不会引用块映射。如果此时出现电源故障，则此块的先前数据仍将完全有效。
* 将块映射2写入逻辑映射中的条目0。
* 空闲块映射1返回到空闲映射列表。
* 空闲后备IO单元2返回到空闲后备IO单元列表。

+--------------------+

Backing Device |01 34 |

+--------------------+

Free Backing IO Unit List 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

+------------+------------+------------+------------+------------+

Chunk Maps | 0 1 X X | | 3 4 X X | | |

+------------+------------+------------+------------+------------+

Free Chunk Map List 1, 3, 4

+---+---+---+---+

Logical Map | 2 | X | 0 | X |

+---+---+---+---+

**跨越多个块的操作**

跨越块边界的操作在逻辑上被分成多个操作，每个操作与单个块相关联。

示例：偏移量为4KB时写入20KB

在这种情况下，写操作被分成偏移4KB的12KB写入（仅影响逻辑映射中的块0）和偏移16KB的8KB写入（仅影响逻辑映射中的块1）。使用上述算法独立地处理每个写入。在两个操作完成之前，不会完成20KB写入。

**取消映射操作**

通过从逻辑映射中删除块映射条目（如果有）来实现对整个块的取消映射操作。块映射返回到空闲块映射列表，并且与块映射关联的任何后备IO单元返回到空闲后备IO单元列表。

仅影响块的一部分的取消映射操作可被视为将零写入块的该区域。如果整个块通过多个操作未映射，则可以通过等于全零的未压缩数据来检测它。当发生这种情况时，可以从逻辑映射中移除块映射条目。

在取消映射整个块之后，对块的后续读取将返回全零。这类似于上面的“16KB偏移读取16KB”。

**写零运算**

写入零操作的处理方式与取消映射操作类似。如果写入零操作覆盖整个块，我们可以完全删除逻辑映射中的块条目。然后对该块的后续读取将返回全零。

**重新启动**

使用libreduce的应用程序将定期退出并需要重新启动。当应用程序重新启动时，它将重新加载压缩卷，以便可以从应用程序退出时的相同状态再次使用它们。

当重新加载压缩卷时，通过遍历逻辑映射来重建空闲块映射列表和空闲后备IO单元列表。逻辑映射仅指向有效的块映射，有效的块映射仅指向有效的后备IO单元。任何未引用的块映射和后备IO单元都会进入各自的空闲列表。

这确保了如果系统在写入操作的中间崩溃 - 即在更新块映射期间或之后，但在将其写入逻辑映射之前 - 在压缩卷重新启动后，与正在进行的写入相关的所有内容都将被忽略。

**在同一块上重叠操作**

实现必须注意处理同一块上的重叠操作。例如，操作1将一些数据写入块A，并且当这正在进行时，操作2也将一些数据写入块A.在这种情况下，操作2不应该开始直到操作1完成。进一步的优化超出了本文档的范围。

**精简配置后备存储**

必须对备份存储进行精简配置，以实现压缩带来的任何节省。该算法将始终使用（并重复使用）支持设备上最接近偏移0的IO单元。这确保即使后备存储设备的大小可能类似于压缩卷的大小，也实际上不会分配后备存储设备的存储，直到实际需要后备IO单元。

## 4.8 通知库

通知库（Notify library）实现了一个事件总线，允许用户注册，生成和监听事件。例如，bdev库可以为bdev创建注册新的事件类型。每次创建bdev时，它都会“发送”该事件。该事件的消费者可以定期轮询新事件以检索它们。事件总线实现为固定大小的圆环。如果事件消费者不经常轮询，则事件可能会丢失。所有事件都由单调递增的整数标识，因此可以检测到丢失的事件，尽管没有恢复。

**注册事件类型**

在初始化期间，发送方库应使用注册自己的事件类型spdk\_notify\_type\_register(const char \*type)。参数'type'是通知类型的名称。

**获取有关事件的信息**

消费者可以在运行时使用期间获取有关可用事件类型的信息spdk\_notify\_foreach\_type，这些信息迭代已注册的通知类型并在每个类型上调用回调，以便用户可以生成有关通知的详细信息。

**获取新事件**

消费者可以通过调用函数来获取事件spdk\_notify\_foreach\_event。调用者应指定最后收到的事件和最大调用次数。每个事件可能有多个消费者。事件总线实现为循环缓冲区，因此较旧的事件可能会被较新的事件覆盖。

**发送事件**

发生事件时，库可以spdk\_notify\_send使用两个字符串进行调用。一个包含事件类型，例如“spdk\_bdev\_register”，第二个包含上下文，例如“Nvme0n1”

**RPC调用**

请参阅[JSON-RPC文档](https://spdk.io/doc/jsonrpc.md/#rpc_get_notification_types)

# 第五章 基本信息

## 5.1 事件框架

SPDK提供了一个用于编写异步，轮询模式，无共享服务器应用程序的框架。事件框架是可选的; 大多数其他SPDK组件集成到应用程序中，而不依赖于SPDK事件库。该框架定义了几个概念 - 反应器（reactors），事件（events）和轮询器（pollers） - 将在以下各节中介绍。事件框架为每个核心（reactor）生成一个线程，并使用无锁队列连接线程。然后可以在线程之间传递消息（事件）。在现代CPU架构中，消息传递通常比传统锁定快得多。有关此框架的理论基础的讨论，请参阅2.3 消息传递和并发。

事件框架公共接口在[event.h中](https://spdk.io/doc/event_8h.html)定义。

### 5.1.1 事件框架设计注意事项

简单的服务器应用程序可以以单线程方式编写。这允许直接的代码可以维持状态而无需任何锁定或其他同步。但是，要向上扩展（例如，允许更多同时连接），应用程序可能需要使用多个线程。在理想情况下，每个连接都独立于所有其他连接，可以通过创建其他线程并为其分配连接来扩展应用程序，而不会引入跨线程同步。不幸的是，在许多实际情况中，连接并不是完全独立的，并且跨线程共享状态是必要的。SPDK提供了一个事件框架来帮助解决这个问题。

### 5.1.2 SPDK事件框架组件

事件（events）

为了在最小化同步开销的同时完成跨线程通信，框架以事件的形式提供消息传递。事件框架为每个CPU核心运行一个事件循环线程。这些线程称为反应器，它们的主要职责是处理队列中的传入事件。每个事件都包含一个捆绑的函数指针及其参数，目的地是特定的CPU核心。使用创建的事件[spdk\_event\_allocate（）](https://spdk.io/doc/event_8h.html#a885e8b94b71844c94f0b380afdfa0df9)，并使用执行[spdk\_event\_call（）](https://spdk.io/doc/event_8h.html#af17b7963b6a6964e5eb18480193acedb)。与每个连接的线程服务器设计不同，它通过依赖于操作系统来调度并发许多线程将阻塞I / O发布到有限数量的内核上来实现并发性，事件驱动模型需要使用显式异步操作来实现并发。可以使用非阻塞函数调用发出异步I / O，并且通常使用回调函数来发信号通知完成。

反应器（reactors）

每个反应器都有一个无锁队列，用于传入该核心的事件，来自任何核心的线程可以将事件插入任何其他核心的队列中。在每个核心上运行的reactor循环检查传入的事件，并在收到它们时以先进先出的顺序执行它们。事件函数永远不应该阻塞，并且最好应该非常快速地执行，因为它们是直接从目标核心上的事件循环中调用的。

轮询器（pollers）

该框架还定义了另一种称为轮询器的函数。可以使用[spdk\_poller\_register（）](https://spdk.io/doc/thread_8h.html#a6b3e883ddce01ae8d93541304d1e9126)函数注册轮询器。轮询器与事件一样，是具有可捆绑和执行的参数的函数。但是，与事件不同，轮询器会重复执行，直到未注册并在其注册的线程上执行。reactor事件循环通过其他事件处理散布对轮询器的调用。轮询器旨在轮询硬件以替代中断。通常，轮询器在主事件循环的每次迭代中执行。如果不需要低延迟，也可以安排轮询器在定时器上定期执行。

### 5.1.3 应用框架

框架本身被捆绑到一个更高级别的抽象中，称为“app”。一旦[调用spdk\_app\_start（）](https://spdk.io/doc/event_8h.html#ab4c22e2920f70becd9b2ef752efd7975)，它将阻塞当前线程，直到应用程序通过调用[spdk\_app\_stop（）](https://spdk.io/doc/event_8h.html#a40f3af1c0340f5c3eb915bc43e3cee83)终止，或者在调用调用者提供的函数之前，在[spdk\_app\_start（）](https://spdk.io/doc/event_8h.html#ab4c22e2920f70becd9b2ef752efd7975)内的初始化代码中发生错误情况。

## 5.2 逻辑卷

Logical Volumes库是一个灵活的存储空间管理系统。它提供了可变大小的虚拟块设备的创建和管理。SPDK逻辑卷库建立在[Blobstore程序员指南](https://spdk.io/doc/blob.html)之上。

### 5.2.1 术语

**逻辑卷存储**

* 速记：lvolstore，lvs
* 输入name：struct spdk\_lvol\_store

逻辑卷存储使用blobstore的超级blob功能来保存uuid（以及将来的其他元数据）。Blobstore类型在blobstore中实现，并保存在磁盘上。lvolstore将在创建时生成UUID，以便可以从其他lvolstores中唯一标识它。默认情况下，创建lvol存储数据区域时会取消映射。可选的-clear-method参数可以在创建时传递，以将该行为更改为写入零或不执行任何操作。

**逻辑卷**

* 速记：lvol
* 输入name：struct spdk\_lvol

逻辑卷实现为从lvolstore创建的SPDK blob。lvol由其UUID唯一标识。Lvol附加可以有别名。

**逻辑卷块设备**

* 速记：lvol\_bdev
* 输入name：struct spdk\_lvol\_bdev

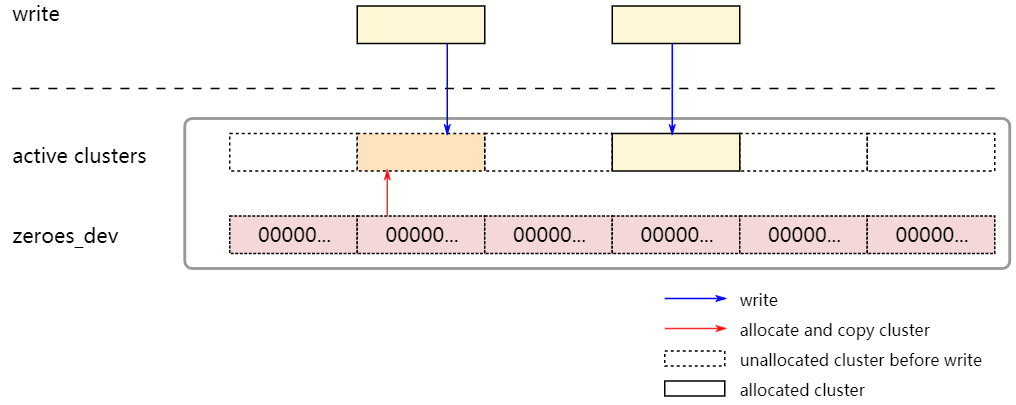
使用[lvol](https://spdk.io/doc/structspdk__bdev.html)实现表示SPDK块设备（[spdk\_bdev](https://spdk.io/doc/structspdk__bdev.html)）。逻辑卷块设备转换通用SPDK块设备I / O（[spdk\_bdev\_io](https://spdk.io/doc/structspdk__bdev__io.html)）操作进入等效的SPDK blob操作。lvol name和lvolstore name的组合以“lvs\_name / lvol\_name”形式提供lvol\_bdev别名。由于blobstore页面大小，创建的bdev的block\_size始终为4096。Cluster\_size可通过参数进行配置。新bdev的大小将向上舍入为cluster\_size的最接近倍数。默认情况下，lvol bdevs声称lvol store的一部分等于它们的设置大小。

启用精简配置选项后，在将数据写入lvol bdev之前，不会从lvol存储中获取空间。默认情况下，在删除lvol bdev或调整大小时，已分配的群集将被取消映射。可选的-clear-method参数可以在创建时传递，以将该行为更改为写入零或不执行任何操作。

**精简配置**

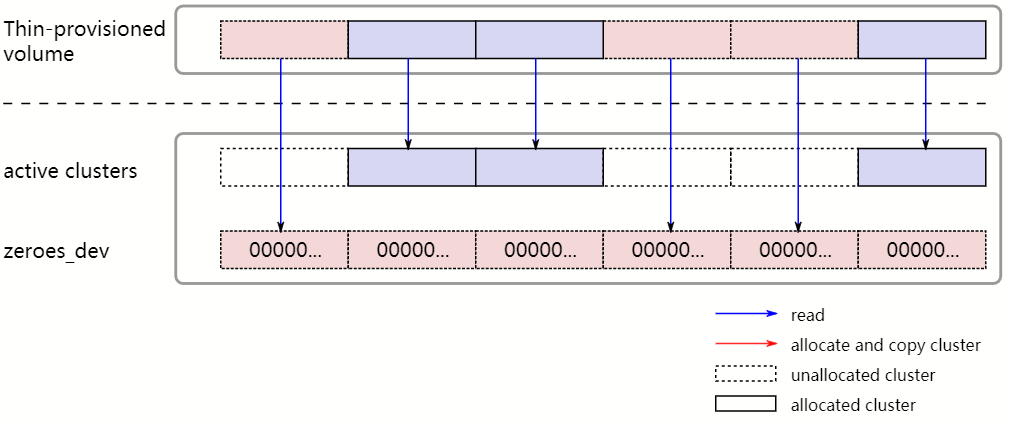
精简配置的lvol依赖于动态集群分配（例如，当执行集群上的第一次写入操作时），仅使用存储数据所需的空间，并且从底层设备获得未分配的集群（例如，zeroes\_dev）。

精简配置blob的示例写入操作如下图所示：



将群集（clusters）写入精简配置的blob

示例读取操作和精简配置blob的结构如下图所示：

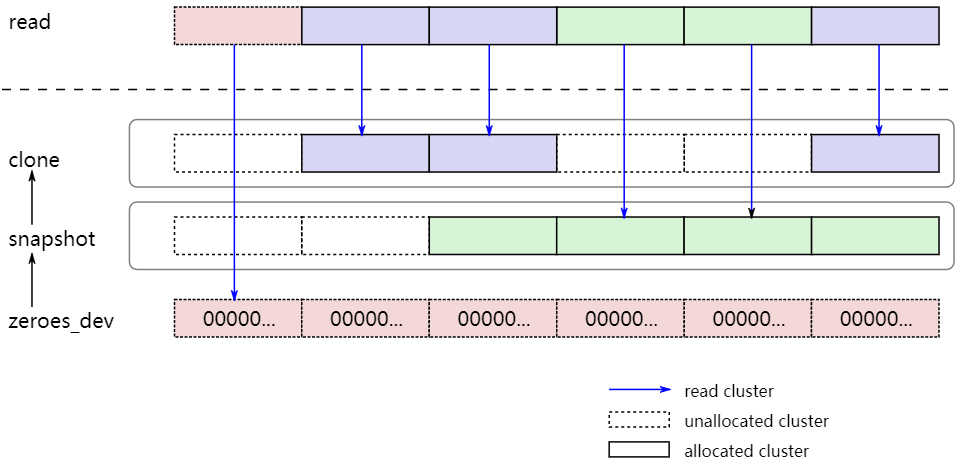


从精简配置的blob中读取群集

**快照和克隆**

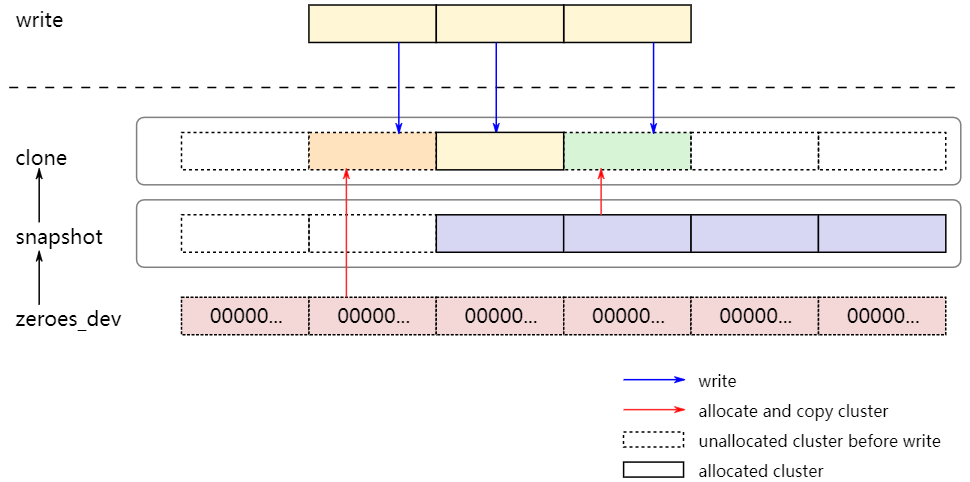
逻辑卷支持快照和克隆功能。用户可以在任何给定时间创建现有逻辑卷的快照以保存当前卷状态的备份。创建快照时，原始卷将变为精简配置，并仅保存与其基础快照的增量差异。这意味着来自未分配群集的每次读取实际上都是从快照读取的，并且每次写入未分配群集都会在实际写入发生之前触发从快照中的相应群集到逻辑卷中的新群集的新群集分配和数据复制。

读操作如下图所示执行：



从克隆中读取集群

写操作如下图所示执行：



将集群写入克隆

用户还可以创建将进行精简配置的现有快照的克隆，并且其行为方式与创建快照的逻辑卷的行为方式相同。只要逻辑卷存储上有足够的空间，就不会创建可以创建的克隆和快照的限制。快照是只读的。可以仅从快照或只读逻辑卷创建克隆。

只有在其上面有一个克隆时，才能删除快照。关系链将相应更新。将合并克隆和快照的集群映射，并使用快照集群映射中的地址更新克隆中未分配集群的条目。整个操作仅修改元数据 - 在此过程中不会复制任何数据。

**膨胀**

Blob可以膨胀以从后备设备复制数据（例如快照）并分配所有剩余的集群。作为此操作的结果，将删除blob的所有依赖项。

使用inflate调用删除支持blob和bdevs关系

**解耦**

通过从所有已分配群集的后备设备（例如快照）复制数据，可以将Blob与其父Blob分离。剩余的未分配群集将保留精简配置。注意：执行解耦时，仅删除单个依赖项。要根据彼此删除blob链中的所有依赖项，需要发出多个调用。

### 5.2.2 配置逻辑卷

逻辑卷没有可用的静态配置。所有配置都通过RPC完成。有关逻辑卷的信息保留在块设备上。

RPC概述

关于lvolstore的RPC：

construct\_lvol\_store [-h] [-c CLUSTER\_SZ] bdev\_name lvs\_name

Constructs lvolstore on specified bdev with specified name. During

construction bdev is unmapped at initialization and all data is

erased. Then original bdev is claimed by

SPDK, but no additional spdk bdevs are created.

Returns uuid of created lvolstore.

Optional parameters:

-h show help

-c CLUSTER\_SZ Specifies the size of cluster. By default its 4MiB.

--clear-method specify data region clear method "none", "unmap" (default), "write\_zeroes"

destroy\_lvol\_store [-h] [-u UUID] [-l LVS\_NAME]

Destroy lvolstore on specified bdev. Removes lvolstore along with lvols on

it. User can identify lvol store by UUID or its name. Note that destroying

lvolstore requires using this call, while deleting single lvol requires

using destroy\_lvol\_bdev rpc call.

optional arguments:

-h, --help show help

get\_lvol\_stores [-h] [-u UUID] [-l LVS\_NAME]

Display current logical volume store list

optional arguments:

-h, --help show help

-u UUID, --uuid UUID show details of specified lvol store

-l LVS\_NAME, --lvs\_name LVS\_NAME show details of specified lvol store

rename\_lvol\_store [-h] old\_name new\_name

Change logical volume store name

optional arguments:

-h, --help show this help message and exit

关于lvol和spdk bdev的RPC：

construct\_lvol\_bdev [-h] [-u UUID] [-l LVS\_NAME] [-t] [-c CLEAR\_METHOD] lvol\_name size

Creates lvol with specified size and name on lvolstore specified by its uuid

or name. Then constructs spdk bdev on top of that lvol and presents it as spdk bdev.

User may use -t switch to create thin provisioned lvol.

Returns the name of new spdk bdev

optional arguments:

-h, --help show help

-c, --clear-method specify data clusters clear method "none", "unmap" (default), "write\_zeroes"

get\_bdevs [-h] [-b NAME]

User can view created bdevs using this call including those created on top of lvols.

optional arguments:

-h, --help show help

-b NAME, --name NAME Name of the block device. Example: Nvme0n1

destroy\_lvol\_bdev [-h] bdev\_name

Deletes a logical volume previously created by construct\_lvol\_bdev.

optional arguments:

-h, --help show help

snapshot\_lvol\_bdev [-h] lvol\_name snapshot\_name

Create a snapshot with snapshot\_name of a given lvol bdev.

optional arguments:

-h, --help show help

clone\_lvol\_bdev [-h] snapshot\_name clone\_name

Create a clone with clone\_name of a given lvol snapshot.

optional arguments:

-h, --help show help

rename\_lvol\_bdev [-h] old\_name new\_name

Change lvol bdev name

optional arguments:

-h, --help show help

resize\_lvol\_bdev [-h] name size

Resize existing lvol bdev

optional arguments:

-h, --help show help

set\_read\_only\_lvol\_bdev [-h] name

Mark lvol bdev as read only

optional arguments:

-h, --help show help

inflate\_lvol\_bdev [-h] name

Inflate lvol bdev

optional arguments:

-h, --help show help

decouple\_parent\_lvol\_bdev [-h] name

Decouple parent of a logical volume

optional arguments:

-h, --help show help

## 5.3 矢量数据包处理（略）

# 第六章 杂项

[Peer-2-Peer DMAs](https://spdk.io/doc/peer_2_peer.html)

“对等网络”，又称[工作组](https://baike.baidu.com/item/%E5%B7%A5%E4%BD%9C%E7%BB%84/5103560)。具体来说，P2P是指网上各台计算机有相同的功能，无主从之分，一台计算机都是既可作为服务器，设定共享资源供网络中其他计算机所使用，又可以作为工作站，没有专用的服务器，也没有专用的工作站。对等网络是小型局域网常用的组网方式。请注意，本文档中讨论的功能目前标记为实验性的。

## 6.1 介绍

对等网络，是在不使用系统内存缓冲区的情况下将数据直接从一个PCI端点（EP）DMA到另一个PCI端点的概念。从SPDK的角度来看，最明显的例子是使用NVMe控制器内存缓冲区（CMB）在两个NVMe SSD之间直接复制数据。

在本文档的这一部分中，我们概述了如何在SPDK中执行P2P操作，并概述了执行P2P操作时可能出现的一些问题。

## 6.2 NVMe的P2P API

下表给出了为NVMe CMB提供P2P功能访问的功能。

|  |  |
| --- | --- |
| 关键功能 | 描述 |
| [spdk\_nvme\_ctrlr\_alloc\_cmb\_io\_buffer（）](https://spdk.io/doc/nvme_8h.html#ad63b25defbb0f1ccd52538e3b9a748df) | 从控制器内存缓冲区分配I / O缓冲区（实验）。 |
| [spdk\_nvme\_ctrlr\_free\_cmb\_io\_buffer（）](https://spdk.io/doc/nvme_8h.html#abb11b2a40d2cd45072f83624956b8b01) | 释放控制器内存I / O缓冲区（实验）。 |
| [spdk\_nvme\_ctrlr\_get\_regs\_cmbsz（）](https://spdk.io/doc/nvme_8h.html#a260b335960b9d81ac1a2601b003b99cd) | 获取NVMe控制器CMBSZ（控制器内存缓冲区大小）寄存器。 |

## 6.3 确定设备支持

SPDK的识别示例应用程序显示设备是否具有控制器内存缓冲区以及它支持的操作。运行如下：

./examples/nvme/identify/identify -r traddr：<pci id of ssd>

cmb\_copy：一个示例P2P应用程序

运行cmb\_copy示例应用程序。

./examples/nvme/cmb\_copy/cmb\_copy -r <pci id of write ssd>-1-0-1 -w <pci id of write ssd>-1-0-1 -c <pci id of the ssd with cmb>

这应该使用CMB作为DMA缓冲区，将读取NVMe SSD上的命名空间1中的单个LBA（LBA 0）复制到写入SSD上命名空间1上的LBA 0。

## 6.4 P2P问题

* 在某些系统中，当在直接连接到根联合体（RC）的PCIe EP之间执行对等网络DMA时，DMA可能会失败或性能可能不会很好。基本上你的milage可能会有所不同。建议您使用PCIe交换机（例如Broadcom或Microsemi提供的交换机），因为它们可以提供良好的性能。
* 即使使用PCIe交换机，也可能存在对等网络DMA无法工作的情况。这可能是由于BIOS和/或OS启用了PCIe访问控制服务（ACS）。您可以使用setpci或通过Internet上可以找到的树内核补丁来禁用ACS。
* 在涉及多个交换机的更复杂的拓扑中，可以在EP之间构建多个路径。这可能导致TLP排序问题。如果您在这些环境中工作，请小心！

# 第七章 驱动程序

## 7.1 NVMe驱动程序\*\*\*

### 7.1.1 介绍

NVMe驱动程序是一个C库，可以直接链接到一个应用程序，该应用程序提供与[NVMe SSD之间的](http://nvmexpress.org/)直接零拷贝数据传输。它完全是被动的，这意味着它不会产生任何线程，只会响应来自应用程序本身的函数调用而执行操作。该库通过直接将[PCI BAR](https://en.wikipedia.org/wiki/PCI_configuration_space)映射到本地进程并执行[MMIO](https://en.wikipedia.org/wiki/Memory-mapped_I/O)来控制NVMe设备。I / O是通过队列对异步提交的，一般流程与Linux的[libaio](http://man7.org/linux/man-pages/man2/io_submit.2.html)并不完全不同。

最近，该库已经过改进，还可以通过NVMe over Fabrics连接到远程NVMe设备。用户现在可以在本地PCI总线上和远程NVMe上通过Fabrics发现服务调用[spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)。API没有改变。

### 7.1.2 例子

**开始使用Hello World**

提供了许多示例来演示如何使用NVMe库。它们都在存储库的[examples / nvme](https://github.com/spdk/spdk/tree/master/examples/nvme)目录中。最好的起点是[hello\_world](https://github.com/spdk/spdk/blob/master/examples/nvme/hello_world/hello_world.c)。

**使用Fio插件运行基准测试**

SPDK为非常流行的[fio](https://github.com/axboe/fio)工具提供了一个插件，用于运行一些基本的基准测试。有关详细信息，请参阅fio启动[指南](https://github.com/spdk/spdk/blob/master/examples/nvme/fio_plugin/)。

**使用Perf工具运行基准测试**

[示例/ nvme / perf](https://github.com/spdk/spdk/tree/master/examples/nvme/perf)中的NVMe perf实用程序是也可用于性能测试的示例之一。fio工具被广泛使用，因为它非常灵活。但是，这种灵活性增加了开销并降低了SPDK的效率。因此，SPDK提供了一个性能基准测试工具，在基准测试期间具有最小的开销。使用具有4K 100％随机读取工作负载的性能与fio时，我们测得的IOPS /核心数量增加了2.6倍。perf基准测试工具提供了多个运行时选项，以支持最常见的工作负载。以下示例演示了如何使用perf。

示例：使用perf for 4K 100％Random Read工作负载到本地NVMe SSD 300秒

perf -q 128 -o 4096 -w randread -r'trtype：PCIe traddr：0000：04：00.0'-t 300

示例：使用perf for 4K 100％随机读取工作负载到通过NVMe-oF通过网络导出的远程NVMe SSD

perf -q 128 -o 4096 -w randread -r'trtype：RDMA adrfam：IPv4 traddr：192.168.100.8 trsvcid：4420'-t 300

示例：将用于4K 70/30随机读/写混合工作负载的perf用于所有本地NVMe SSD 300秒

perf -q 128 -o 4096 -w randrw -M 70 -t 300

示例：使用perf进行扩展LBA格式CRC保护测试到本地NVMe SSD，用户必须先写入SSD才能从SSD读取LBA

perf -q 1 -o 4096 -w write -r'trtype：PCIe traddr：0000：04：00.0'-t 300 -e'PRACT = 0，PRCKH = GUARD'

perf -q 1 -o 4096 -w read -r'trtype：PCIe traddr：0000：04：00.0'-t 200 -e'PRACT = 0，PRCKH = GUARD'

### 7.1.3 公共接口

[spdk / nvme.h](https://spdk.io/doc/nvme_8h.html)

|  |  |
| --- | --- |
| 关键功能 | 描述 |
| [spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d) | 枚举传输ID指示的总线，并将用户空间NVMe驱动程序附加到找到的每个设备（如果需要）。 |
| [spdk\_nvme\_ctrlr\_alloc\_io\_qpair（）](https://spdk.io/doc/nvme_8h.html#a13f745d239dab9b8f934fae2ad4984a2) | 分配I / O队列对（提交和完成队列）。 |
| [spdk\_nvme\_ctrlr\_get\_ns（）](https://spdk.io/doc/nvme_8h.html#a6f01fb8a5f404e41e6fa224a7942be08) | 获取给定控制器的命名空间句柄。 |
| [spdk\_nvme\_ns\_cmd\_read（）](https://spdk.io/doc/nvme_8h.html#a084c6ecb53bd810fbb5051100b79bec5) | 向指定的NVMe名称空间提交读取I / O. |
| [spdk\_nvme\_ns\_cmd\_readv（）](https://spdk.io/doc/nvme_8h.html#acb47ce7de6b6e963ec9fb8de261466ae) | 将读取I / O提交到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_read\_with\_md（）](https://spdk.io/doc/nvme_8h.html#aa2913b93326e636eca6dfe7b42e349fe) | 向指定的NVMe名称空间提交读取I / O. |
| [spdk\_nvme\_ns\_cmd\_write（）](https://spdk.io/doc/nvme_8h.html#a3065f669d8b605efdcadffbf94a50538) | 将写入I / O提交到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_writev（）](https://spdk.io/doc/nvme_8h.html#adfcbb5d31f0b572847cc8ae8b07dfcfb) | 将写入I / O提交到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_write\_with\_md（）](https://spdk.io/doc/nvme_8h.html#adc2aa2be0d657be0c63d5abc02b274ec) | 将写入I / O提交到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_write\_zeroes（）](https://spdk.io/doc/nvme_8h.html#af02e8c701c5496f163a69917ad5397dd) | 提交写入将I / O归零到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_dataset\_management（）](https://spdk.io/doc/nvme_8h.html#ac0c646dd18675c54ffcf834ce699658d) | 将数据集管理请求提交到指定的NVMe名称空间。 |
| [spdk\_nvme\_ns\_cmd\_flush（）](https://spdk.io/doc/nvme_8h.html#aed0b134e140121bb9bd8664d4a43a5c6) | 向指定的NVMe名称空间提交刷新请求。 |
| [spdk\_nvme\_qpair\_process\_completions（）](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782) | 处理在队列对上提交的I / O的任何未完成的完成。 |
| [spdk\_nvme\_ctrlr\_cmd\_admin\_raw（）](https://spdk.io/doc/nvme_8h.html#afe2a9d3b715649b4d0a0e89196a13e6d) | 将给定的admin命令发送到NVMe控制器。 |
| [spdk\_nvme\_ctrlr\_process\_admin\_completions（）](https://spdk.io/doc/nvme_8h.html#a10282695461985f58f54de022911745e) | 处理管理命令的任何未完成的完成。 |
| [spdk\_nvme\_ctrlr\_cmd\_io\_raw（）](https://spdk.io/doc/nvme_8h.html#a1e3def668122e76abbfb74305f118291) | 将给定的NVM I / O命令发送到NVMe控制器。 |
| [spdk\_nvme\_ctrlr\_cmd\_io\_raw\_with\_md（）](https://spdk.io/doc/nvme_8h.html#afd2d8453bb2478c7b1a70c0a09c7ef4b) | 将带有元数据的给定NVM I / O命令发送到NVMe控制器。 |

### 7.1.4 NVMe驱动程序设计

**NVMe I / O提交**

使用nvme\_ns\_cmd\_xxx函数将I / O提交到NVMe命名空间。NVMe驱动程序将I / O请求作为命令中指定的队列对上的NVMe提交队列条目提交。该函数在命令完成之前立即返回。应用程序必须通过调用[spdk\_nvme\_qpair\_process\_completions（）](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782)在具有未完成I / O的每个队列对上轮询I / O完成，以接收完成回调。

也可以看看

[spdk\_nvme\_ns\_cmd\_read](https://spdk.io/doc/nvme_8h.html#a084c6ecb53bd810fbb5051100b79bec5)，[spdk\_nvme\_ns\_cmd\_write](https://spdk.io/doc/nvme_8h.html#a3065f669d8b605efdcadffbf94a50538)，[spdk\_nvme\_ns\_cmd\_dataset\_management](https://spdk.io/doc/nvme_8h.html#ac0c646dd18675c54ffcf834ce699658d)，[spdk\_nvme\_ns\_cmd\_flush](https://spdk.io/doc/nvme_8h.html#aed0b134e140121bb9bd8664d4a43a5c6)，[spdk\_nvme\_qpair\_process\_completions](https://spdk.io/doc/nvme_8h.html#aa331d140870e977722bfbb6826524782)

**扩展性能**

NVMe队列对（struct spdk\_nvme\_qpair）为I / O提供并行提交路径。可以从不同线程同时在多个队列对上提交I / O。 但是，队列对不包含锁或原子，因此给定的队列对一次只能由单个线程使用。NVMe驱动程序不强制执行此要求（这样做需要锁定），违反此要求会导致未定义的行为。

允许的队列对数量由NVMe SSD本身决定。规范允许数千个，但大多数设备支持32到128。规范不保证每个队列对可用的性能，但实际上，只使用一个队列对几乎总是可以实现设备的完整性能。例如，如果一个设备声称能够在队列深度128处能达到每秒450000个I/O，实际上，无论驱动程序使用4个队列对（每个队列对的队列深度为32），还是单个队列对的队列深度为128），这都无关紧要。

鉴于上述情况，使用SPDK的应用程序最简单的线程模型是在池中生成固定数量的线程，并将单个NVMe队列对专用于每个线程。进一步的改进是将每个线程固定到一个单独的CPU内核，并且SPDK文档通常会交替使用“CPU内核”和“线程”，因为我们考虑到了这个线程模型。

NVMe驱动程序在I/O路径中不带锁，因此只要队列对和CPU核心专用于每个新线程，它就可以在每个线程的性能上进行线性扩展。为了充分利用这种伸缩性，应用程序应考虑组织其内部数据结构，以便将数据专门分配给单个线程。所有需要该数据的操作都应该通过向拥有线程发送请求来完成。这导致了消息传递体系结构，而不是锁定体系结构，并且将导致在CPU核心之间具有更好的伸缩性。

**NVMe驱动程序内部内存使用**

SPDK NVMe驱动程序提供零拷贝数据传输路径，这意味着I / O命令没有数据缓冲区。但是，某些Admin命令具有数据副本，具体取决于用户使用的API。

每个队列对都有许多跟踪器，用于跟踪调用者提交的命令。I / O队列的数字跟踪器取决于用户对队列大小的输入以及从控制器功能寄存器字段读取的值支持的最大队列条目（MQES，基于0的值）。每个跟踪器的固定大小为4096字节，因此每个I / O队列使用的最大内存为：（MQES + 1）\* 4 KiB。

I / O队列对可以在主机内存中分配，这适用于大多数NVMe控制器，一些支持Controller Memory Buffer控制器内存缓冲的NVMe控制器可以将I / O队列对放在控制器的PCI BAR空间，SPDK NVMe驱动程序可以将I/O提交队列放进控制器内存缓冲区，它取决于用户的输入和控制器功能。每个提交队列条目（SQE）和完成队列条目（CQE）分别消耗64个字节和16个字节。因此，每个I / O队列对使用的最大内存为（MQES + 1）\*（64 + 16）字节。

### 7.1.5 NVMe over Fabrics主机支持

NVMe驱动程序支持连接到远程NVMe-oF目标，并以与本地NVMe SSD相同的方式与它们进行交互。

**在Fabric结构目标上指定远程NVMe**

连接到远程NVMe-oF目标的方法与本地PCIe连接的NVMe设备的常规枚举过程非常相似。要连接到远程NVMe over Fabrics子系统，用户可以使用指定目标NVMe地址的trid参数调用spdk\_nvme\_probe（）。

调用者可以手动填写[spdk\_nvme\_transport\_id](https://spdk.io/doc/structspdk__nvme__transport__id.html)结构体，或使用[spdk\_nvme\_transport\_id\_parse（）](https://spdk.io/doc/nvme_8h.html#ac37484cc5d14777e4ae1fde031d0edf2)函数将人类可读的字符串表示转换为所需的结构。

该[spdk\_nvme\_transport\_id](https://spdk.io/doc/structspdk__nvme__transport__id.html)可能包含发现服务或单个NVM子系统的地址。如果指定了发现服务地址，则NVMe库将为每个发现的NVM子系统调用[spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d) probe\_cb，这允许用户选择要附加的所需子系统。或者，如果地址直接指定单个NVM子系统，则NVMe库将probe\_cb仅调用该子系统; 这允许用户跳过发现步骤并直接连接到具有已知地址的子系统。

**RDMA限制**

请参阅NVMe-oF target的[RDMA限制](https://spdk.io/doc/nvmf.html#nvmf_rdma_limitations)

### 7.1.6 NVMe多进程

此功能使SPDK NVMe驱动程序能够支持访问同一NVMe设备的多个进程。NVMe驱动程序从共享内存中分配关键结构，以便每个进程可以映射该内存并创建自己的队列对或共享管理队列。每个NVMe控制器的I / O队列对数量有限。

此功能的主要动机是支持管理工具，这些工具可以附加到长时间运行的应用程序，执行一些维护工作或收集信息，然后分离。

**配置**

DPDK EAL允许生成不同类型的进程，每个进程对应用程序使用的巨大页面内存具有不同的权限。

有两种类型的过程：

* 初始化共享内存并具有完全权限的主进程
* 辅助进程，可以通过映射其共享内存区域并执行NVMe操作（包括创建队列对）来附加到主进程。

默认情况下启用此功能，并通过选择共享内存组ID的值来控制此功能。此ID是正整数，具有相同共享内存组ID的两个应用程序将共享内存。具有给定共享内存组ID的第一个应用程序将被视为主要应用程序，和所有其他辅助应用程序。

示例：相同的shm\_id和非重叠核心掩码

./perf options [AIO device(s)]...

[-c core mask for I/O submission/completion]

[-i shared memory group ID]

./perf -q 1 -o 4096 -w randread -c 0x1 -t 60 -i 1

./perf -q 8 -o 131072 -w write -c 0x10 -t 60 -i 1

**限制**

共享内存的两个进程可能不共享其核心掩码中的任何核心。

如果主进程在辅助进程仍在运行时退出，则这些进程将继续运行。但是，无法创建新的主进程。

应用程序负责协调对逻辑块的访问。

如果进程意外退出，则在最后一个进程退出时将释放分配的内存。

也可以看看

[spdk\_nvme\_probe](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)，[spdk\_nvme\_ctrlr\_process\_admin\_completions](https://spdk.io/doc/nvme_8h.html#a10282695461985f58f54de022911745e)

### 7.1.7 NVMe Hotplug

在NVMe驱动程序级别，我们为Hotplug提供以下支持：

* Hotplug事件检测：NVMe库的用户可以定期调用[spdk\_nvme\_probe（）](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)来检测hotplug事件。将为检测到的每个新设备调用probe\_cb，后跟attach\_cb。如果先前连接的NVMe设备不再存在于系统上，则用户可以可选地还提供将被调用的remove\_cb。删除设备的所有后续I / O都将返回错误。
* 使用IO负载热删除NVMe：在I / O发生时热删除设备时，对PCI BAR的所有访问都将导致SIGBUS错误。NVMe驱动程序通过安装SIGBUS处理程序并将PCI BAR重新映射到新的占位符内存位置来自动处理此情况。这意味着在热删除期间运行中的I / O将以适当的错误代码完成，并且不会使应用程序崩溃。

也可以看看

[spdk\_nvme\_probe](https://spdk.io/doc/nvme_8h.html#a225bbc386ec518ae21bd5536f21db45d)

## 7.2 I/OAT驱动程序

### 7.2.1 公共接口

* [spdk / ioat.h](https://spdk.io/doc/ioat_8h.html)

### 7.2.2 关键功能

|  |  |
| --- | --- |
| 功能 | 描述 |
| [spdk\_ioat\_probe（）](https://spdk.io/doc/ioat_8h.html#a784c1a69962e0964cf6988badd945b6f) | 枚举连接到系统的I / OAT设备，并根据需要将用户空间I / OAT驱动程序附加到它们。 |
| [spdk\_ioat\_get\_dma\_capabilities（）](https://spdk.io/doc/ioat_8h.html#a87ce4a1c8bdd3fb69079ac51e00f92e5) | 获得DMA引擎功能。 |
| [spdk\_ioat\_submit\_copy（）](https://spdk.io/doc/ioat_8h.html#ac1de22182996edecb435f9583665008d) | 构建并提交DMA引擎内存复制请求。 |
| [spdk\_ioat\_submit\_fill（）](https://spdk.io/doc/ioat_8h.html#a6025f251c715e93ea27ee03b5ab9557c) | 构建并提交DMA引擎内存填充请求。 |

## 7.3 Virtio驱动程序

### 7.3.1 介绍

SPDK Virtio驱动程序是一个允许与Virtio设备通信的C库。它允许任何SPDK应用程序成为（SPDK）vhost目标的发起者。

该驱动程序支持两种不同的使用模型

* PCI -这是在客户端虚拟机中使用的标准操作模式，其中QEMU将virtio控制器作为虚拟PCI设备提供。
* vhost-user - 可用于直接在同一主机上连接到vhost套接字。

与SPDK [vhost Target](https://spdk.io/doc/vhost.html)一样，驱动程序使用轮询器而不是标准中断来检查I / O响应。如果在VM内部使用，它会绕过QEMU和客户端内核的中断和上下文切换开销，从而显着提高整体I / O性能。

该Virtio库目前用于实现两个bdev模块：[Virtio SCSI](https://spdk.io/doc/bdev.html#bdev_config_virtio_scsi)和[Virtio Block](https://spdk.io/doc/bdev.html#bdev_config_virtio_blk)。这些模块将导出任何SPDK应用程序可使用的通用SPDK块设备。

### 7.3.2 2MB大页面

用户vhost规范，对使用的“内存区域”的数量进行了限制（8个）。每个区域对应一个文件描述符，DPDK - 作为SPDK的内存分配器 - 默认情况下每个hugepage使用一个文件。因此，默认情况下，这使得spdk virtio只具有1GB的大页面。要使用virtio initiator和2MB Hugepages运行SPDK应用程序，需要传递“-g”命令行选项。这将强制DPDK为其所有内存创建一个单独的非物理连续的hugetlbfs文件。

# 第八章 工具

## 8.1 SPDK CLI

Spdkcli是SPDK的命令行管理应用程序。Spdkcli支持有限数量的应用程序和bdev模块，应该被视为v18.04版本的实验。这个实验版本是为v18.04添加的，以获得早期的反馈，因为spdkcli在下一个SPDK版本中变得更加全面。

### 8.1.1 安装所需的依赖项

所有依赖项都应由scripts / pkgdep.sh脚本处理。目前的包依赖包括：

* configshell

### 8.1.2 运行SPDK应用程序实例

./scripts/setup.sh

./app/vhost/vhost -c vhost.conf

### 8.1.3 运行SPDK CLI

应该使用与SPDK应用程序相同的权限运行Spdkcli。要在交互模式下使用SPDK CLI，请使用：

scripts/spdkcli.py

使用“help”命令获取每个树节点的可用命令列表。

也可以使用SPDK CLI只运行一个命令，只需使用该命令作为应用程序的参数。例如，要查看当前配置并立即退出：

scripts/spdkcli.py ls

### 8.1.4 可选 - 创建Python虚拟环境

如果您不想丢弃系统Python安装，则可以使用Python虚拟环境。

首先创建虚拟环境：

cd spdk

mkdir venv

virtualenv-3 ./venv

source ./venv/bin/activate

然后使用pip安装依赖项。这样，依赖关系将仅安装在虚拟环境中。

(venv) pip install configshell-fb

提示：如果您使用“sudo”而不是root帐户，建议在激活环境之前执行“sudo -s”。这是因为当使用sudo调用spdkcli时，venv可能无法正常工作，例如“sudo python spdkcli.py” - 某些环境变量可能无法传递，您将遇到错误。

## 8.2 nvme-CLI

### 8.2.1 nvme-cli with SPDK入门指南

现在，nvme-cli可以为大多数可用命令和Intel特定命令支持内核驱动程序和SPDK用户模式驱动程序。

1. 从SPDK GitHub fork中克隆nvme-cli存储库。确保你看看spdk-1.6分支。

git clone -b spdk-1.6 https://github.com/spdk/nvme-cli.git

1. 从nvme-cli文件夹下的<https://github.com/spdk/spdk>克隆SPDK存储库。
2. 请参阅SPDK文件夹下的“README.md”以正确构建SPDK。
3. 请参阅nvme-cli文件夹下的“README.md”以正确构建nvme-cli。
4. 使用“root”帐户执行“<spdk\_folder> /scripts/setup.sh”。
5. 更新nvme-cli文件夹下的“spdk.conf”文件以正确配置SPDK。注意事项如下：

spdk = 1

指示是否使用spdk。可以是0（关闭）或1（开启）。

默认为1，假定您已运行“<spdk\_folder> /scripts/setup.sh”，从内核解除绑定驱动器。

core\_mask为0x1。

一个位掩码，表示用于nvme-cli操作的核心。

默认为核心0。

mem\_size = 512

用于nvme-cli的保留巨页内存量（以MB为单位）。

默认为512MB。

shm\_id = 0

表示与NVMe驱动器关联的spdk应用程序的共享内存ID，并应相应调整。

默认为0。

1. 运行“./nvme list”命令以获取每个找到的NVMe SSD的域：bus：device.function。
2. 对于指定的设备，使用domain：bus：device.function而不是“/ dev / nvmeX”运行其他nvme命令。

示例：./ nvme smart-log 0000：01：00.0

1. 对英特尔NVMe SSD运行针对英特尔特定命令的“./nvme intel”命令。

示例：./ nvme intel internal-log 0000：08：00.0

1. 使用“root”用户执行“<spdk\_folder> /scripts/setup.sh reset”并在spdk.conf中更新“spdk = 0”以使用内核驱动程序（如果需要）。

### 8.2.2 使用场景

**作为系统上唯一的SPDK应用程序运行**

* 在spdk.conf中将spdk修改为1。如果系统具有较少的内核或较少的内存，请相应地更新spdk.conf。

**与共享NVMe SSD上的其他正在运行的SPDK应用程序一起运行**

* 对于其他正在运行的SPDK应用程序，从“-i 1”之类的参数开始，使其具有相同的“shm\_id”。
* 使用默认的spdk.conf设置，其中“shm\_id = 1”启动nvme-cli。
* 如果其他SPDK应用程序使用不同的shm\_id参数运行，请相应地更新“spdk.conf”。

**在非共享NVMe SSD上与其他正在运行的SPDK应用程序一起运行**

* 正确配置其他正在运行的SPDK应用程序。

a.仅访问所需的NVMe SSD。

b.分配固定数量的内存，而不是所有可用内存。

* 正确配置nvme-cli的spdk.conf设置。

a.不能从其他SPDK应用程序访问NVMe SSD。

b.将内存大小更改为适当的大小。

**注意**

要运行新构建的nvme-cli，要么显式运行为“./nvme”，要么将其添加到$ PATH中以避免调用其他已安装的版本。

要在任意目录中运行具有SPDK支持的新构建的nvme-cli，请将“spdk.conf”从nvme cli文件夹复制到该目录，并按照建议更新配置。

# 第九章 性能测试报告（略）

# 第十章NVMe-oF Target跟踪点\*\*\*

## 10.1 介绍

SPDK有一个跟踪框架，用于在运行时捕获低级事件信息。跟踪点提供了可在运行时访问的高性能跟踪机制。它们在共享内存中实现为循环缓冲区，可从其他进程访问。NVMe-oF目标配有跟踪点，可以分析性能和应用程序崩溃。（注意：SPDK跟踪框架应该仍然被认为是实验性的。正在进行正式化和记录框架的工作正在进行中。）

## 10.2 启用跟踪点

跟踪点分组放置。它们作为一个组启用和禁用。

要启用SPDK目标应用程序中所有跟踪点组的检测，请在-e参数设置为0xFFFF的情况下启动目标：

app/nvmf\_tgt/nvmf\_tgt -e 0xFFFF

要在SPDK目标应用程序中启用NVMe-oF RDMA跟踪点的检测，请在-e参数设置为0x10的情况下启动目标：

app/nvmf\_tgt/nvmf\_tgt -e 0x10

当目标启动时，将使用spdk\_trace应用程序记录一条消息，其中包含以人类可读格式查看跟踪点所需的信息。目标还将记录有关共享内存文件的信息。

app.c：527：spdk\_app\_setup\_trace：\* NOTICE \*：指定了Tracepoint Group Mask 0xFFFF。

app.c：531：spdk\_app\_setup\_trace：\*注意\*：使用'spdk\_trace -s nvmf -p 24147'在运行时捕获事件的快照。

app.c：533：spdk\_app\_setup\_trace：\* NOTICE \*：或复制/dev/shm/nvmf\_trace.pid24147进行离线分析/调试。

请注意，启用跟踪点时，应用程序退出时不会删除共享内存文件。这可确保在应用程序退出后可以使用该文件进行分析。在Linux上，共享内存文件位于/ dev / shm中，如果需要，可以手动删除以释放shm空间。系统重启还将释放所有/ dev / shm文件。

## 10.3 捕获事件的快照

将I / O发送到SPDK目标应用程序以生成事件。以下是perf通过RDMA网络接口将I / O发送到NVMe-oF目标10分钟的示例用法。

./perf -q 128 -s 4096 -w randread -t 600 -r'trtype：RDMA adrfam：IPv4 traddr：192.168.100.2 trsvcid：4420'

可以在app / trace目录中找到spdk\_trace程序。要分析运行NVMe-oF目标的同一系统上的跟踪点，只需执行日志中显示的命令行：

app/trace/spdk\_trace -s nvmf -p 24147

要分析不同系统上的跟踪点，请首先准备要传输的跟踪点文件。tracepoint文件可能很大，但通常压缩得很好。此步骤还可用于准备跟踪点文件以附加到GitHub问题以调试NVMe-oF应用程序崩溃。

bzip2 -c /dev/shm/nvmf\_trace.pid24147> /tmp/trace.bz2

将/tmp/trace.bz2跟踪点文件传输到其他系统后：

bunzip2 /tmp/trace.bz2

app/trace/spdk\_trace -f /tmp/trace

以下是示例跟踪捕获，显示每个I / O在每个RDMA状态下花费的累积时间。具有相同ID的所有跟踪捕获都针对相同的I / O.

28: 6026.658 ( 12656064) RDMA\_REQ\_NEED\_BUFFER id: r3622 time: 0.019

28: 6026.694 ( 12656140) RDMA\_REQ\_RDY\_TO\_EXECUTE id: r3622 time: 0.055

28: 6026.820 ( 12656406) RDMA\_REQ\_EXECUTING id: r3622 time: 0.182

28: 6026.992 ( 12656766) RDMA\_REQ\_EXECUTED id: r3477 time: 228.510

28: 6027.010 ( 12656804) RDMA\_REQ\_TX\_PENDING\_C\_TO\_H id: r3477 time: 228.528

28: 6027.022 ( 12656828) RDMA\_REQ\_RDY\_TO\_COMPLETE id: r3477 time: 228.539

28: 6027.115 ( 12657024) RDMA\_REQ\_COMPLETING id: r3477 time: 228.633

28: 6027.471 ( 12657770) RDMA\_REQ\_COMPLETED id: r3518 time: 171.577

28: 6028.027 ( 12658940) RDMA\_REQ\_NEW id: r3623

28: 6028.057 ( 12659002) RDMA\_REQ\_NEED\_BUFFER id: r3623 time: 0.030

28: 6028.095 ( 12659082) RDMA\_REQ\_RDY\_TO\_EXECUTE id: r3623 time: 0.068

28: 6028.216 ( 12659336) RDMA\_REQ\_EXECUTING id: r3623 time: 0.189

28: 6028.408 ( 12659740) RDMA\_REQ\_EXECUTED id: r3505 time: 190.509

28: 6028.441 ( 12659808) RDMA\_REQ\_TX\_PENDING\_C\_TO\_H id: r3505 time: 190.542

28: 6028.452 ( 12659832) RDMA\_REQ\_RDY\_TO\_COMPLETE id: r3505 time: 190.553

28: 6028.536 ( 12660008) RDMA\_REQ\_COMPLETING id: r3505 time: 190.637

28: 6028.854 ( 12660676) RDMA\_REQ\_COMPLETED id: r3465 time: 247.000

28: 6029.433 ( 12661892) RDMA\_REQ\_NEW id: r3624

28: 6029.452 ( 12661932) RDMA\_REQ\_NEED\_BUFFER id: r3624 time: 0.019

28: 6029.482 ( 12661996) RDMA\_REQ\_RDY\_TO\_EXECUTE id: r3624 time: 0.050

28: 6029.591 ( 12662224) RDMA\_REQ\_EXECUTING id: r3624 time: 0.158

28: 6029.782 ( 12662624) RDMA\_REQ\_EXECUTED id: r3564 time: 96.937

28: 6029.798 ( 12662658) RDMA\_REQ\_TX\_PENDING\_C\_TO\_H id: r3564 time: 96.953

28: 6029.812 ( 12662688) RDMA\_REQ\_RDY\_TO\_COMPLETE id: r3564 time: 96.967

28: 6029.899 ( 12662870) RDMA\_REQ\_COMPLETING id: r3564 time: 97.054

28: 6030.262 ( 12663634) RDMA\_REQ\_COMPLETED id: r3477 time: 231.780

28: 6030.786 ( 12664734) RDMA\_REQ\_NEW id: r3625

28: 6030.804 ( 12664772) RDMA\_REQ\_NEED\_BUFFER id: r3625 time: 0.018

28: 6030.841 ( 12664848) RDMA\_REQ\_RDY\_TO\_EXECUTE id: r3625 time: 0.054

28: 6030.963 ( 12665104) RDMA\_REQ\_EXECUTING id: r3625 time: 0.176

28: 6031.139 ( 12665474) RDMA\_REQ\_EXECUTED id: r3552 time: 114.906

28: 6031.196 ( 12665594) RDMA\_REQ\_TX\_PENDING\_C\_TO\_H id: r3552 time: 114.963

28: 6031.210 ( 12665624) RDMA\_REQ\_RDY\_TO\_COMPLETE id: r3552 time: 114.977

28: 6031.293 ( 12665798) RDMA\_REQ\_COMPLETING id: r3552 time: 115.060

28: 6031.633 ( 12666512) RDMA\_REQ\_COMPLETED id: r3505 time: 193.734

28: 6032.230 ( 12667766) RDMA\_REQ\_NEW id: r3626

28: 6032.248 ( 12667804) RDMA\_REQ\_NEED\_BUFFER id: r3626 time: 0.018

28: 6032.288 ( 12667888) RDMA\_REQ\_RDY\_TO\_EXECUTE id: r3626 time: 0.058

28: 6032.396 ( 12668114) RDMA\_REQ\_EXECUTING id: r3626 time: 0.166

28: 6032.593 ( 12668528) RDMA\_REQ\_EXECUTED id: r3570 time: 90.443

28: 6032.611 ( 12668564) RDMA\_REQ\_TX\_PENDING\_C\_TO\_H id: r3570 time: 90.460

28: 6032.623 ( 12668590) RDMA\_REQ\_RDY\_TO\_COMPLETE id: r3570 time: 90.473

28: 6032.707 ( 12668766) RDMA\_REQ\_COMPLETING id: r3570 time: 90.557

28: 6033.056 ( 12669500) RDMA\_REQ\_COMPLETED id: r3564 time: 100.211

## 10.4 捕获足够的跟踪事件

由于SPDK应用程序直接生成的跟踪点文件是共享内存中的循环缓冲区，因此它捕获的跟踪事件可能不足以进行进一步分析。可以在app / trace\_record目录中找到spdk\_trace\_record程序。spdk\_trace\_record用于轮询spdk tracepoint共享内存，从中记录新条目，并在SIGINT或SIGTERM关闭时将所有条目存储到指定的输出文件中。启动SPDK nvmf目标后，只需执行日志中显示的命令行：

app/trace\_record/spdk\_trace\_record -q -s nvmf -p 24147 -f /tmp/spdk\_nvmf\_record.trace

还将I / O发送到SPDK目标应用程序，以通过前一个perf示例生成事件10分钟。

./perf -q 128 -s 4096 -w randread -t 600 -r'trtype：RDMA adrfam：IPv4 traddr：192.168.100.2 trsvcid：4420'

完成perf示例后，通过信号SIGINT（Ctrl + C）关闭spdk\_trace\_record。要从spdk\_trace\_record分析tracepoints输出文件，只需运行spdk\_trace程序：

app/trace/spdk\_trace -f /tmp/spdk\_nvmf\_record.trace

## 10.5 添加新的跟踪点

SPDK应用程序和库提供了几个跟踪点。您可以将新跟踪点添加到现有跟踪组。例如，要将新跟踪点添加到SPDK RDMA库（lib / nvmf / rdma.c）跟踪组TRACE\_GROUP\_NVMF\_RDMA，请使用SPDK\_TPOINT\_ID宏定义跟踪点并为其分配唯一ID：

#define TRACE\_GROUP\_NVMF\_RDMA 0x4

#define TRACE\_RDMA\_REQUEST\_STATE\_NEW SPDK\_TPOINT\_ID（TRACE\_GROUP\_NVMF\_RDMA，0x0）

...

#define NEW\_TRACE\_POINT\_NAME SPDK\_TPOINT\_ID（TRACE\_GROUP\_NVMF\_RDMA，UNIQUE\_ID）

您还需要使用spdk\_trace\_register\_description函数在应用程序/库中的SPDK\_TRACE\_REGISTER\_FN宏调用中注册新的跟踪点，如下所示：

SPDK\_TRACE\_REGISTER\_FN（nvmf\_trace）

{

spdk\_trace\_register\_object（OBJECT\_NVMF\_RDMA\_IO，'r'）;

spdk\_trace\_register\_description（”RDMA\_REQ\_NEW”，””，

TRACE\_RDMA\_REQUEST\_STATE\_NEW，

OWNER\_NONE，OBJECT\_NVMF\_RDMA\_IO，1,1，”cmid： ”）;

...

spdk\_trace\_register\_description（”NEW\_RDMA\_REQ\_NAME”，””，

NEW\_TRACE\_POINT\_NAME，

OWNER\_NONE，OBJECT\_NVMF\_RDMA\_IO，0,1，”cmid： ”）;

}

最后，在应用程序/库中的适当位置使用spdk\_trace\_record函数来记录新跟踪点的当前跟踪状态。以下示例显示了spdk\_trace\_record函数用于记录多个跟踪点的当前跟踪状态的用法。

case RDMA\_REQUEST\_STATE\_NEW:

spdk\_trace\_record(TRACE\_RDMA\_REQUEST\_STATE\_NEW, 0, 0, (uintptr\_t)rdma\_req,

(uintptr\_t)rqpair->cm\_id);

...

break;

case RDMA\_REQUEST\_STATE\_NEED\_BUFFER:

spdk\_trace\_record(TRACE\_RDMA\_REQUEST\_STATE\_NEED\_BUFFER, 0, 0, (uintptr\_t)rdma\_req,

(uintptr\_t)rqpair->cm\_id);

...

break;

case RDMA\_REQUEST\_STATE\_TRANSFER\_PENDING\_HOST\_TO\_CONTROLLER:

spdk\_trace\_record(TRACE\_RDMA\_REQUEST\_STATE\_TRANSFER\_PENDING\_HOST\_TO\_CONTROLLER, 0, 0,

(uintptr\_t)rdma\_req, (uintptr\_t)rqpair->cm\_id);

...

[Tracepoint库文档](https://www.spdk.io/doc/trace_8h.html)中记录了所有跟踪功能。

（完）