챕터 3. 풀 (Pools)

ZFS 풀 또는 zpool은 ZFS 스택의 중간을 형성하며, 하위 수준의 가상 장치를 사용자가 볼 수 있는 파일 시스템에 연결합니다. 풀 보기 시스템의 모든 풀을 보려면 스토리지 블록의 zpoolocating을 실행하세요. ZFS 풀 수준에서 ZFS 데이터 세트에 사용할 수 있는 공간을 늘리거나 특수 가상 장치를 추가하여 읽기 또는 쓰기 성능을 향상시킬 수 있습니다.

ZFS 블록

UFS 및 extfs와 같은 기존 파일시스템은 디스크에 데이터를 고정 크기 블록으로 저장합니다. 파일 시스템에는 아이노드 (inode)라고 하는 특수 블록이 있으며, 이 블록은 어떤 블록이 어떤 파일에 속하는지 색인합니다. NTFS나 FAT와 같은 비유닉스 파일시스템도 비슷한 구조를 사용합니다. 이는 업계 전반의 표준입니다.

ZFS는 특별한 인덱스 블록을 미리 구성하지 않습니다. 스트라이프라고도 하는 스토리지 블록만 사용합니다. 각 블록에는 블록을 디스크의 다른 블록과 트리로 연결하기 위한 인덱스 정보가 포함되어 있습니다. ZFS는 블록의 모든 정보에 대한 해시를 계산하고 해당 정보를 블록과 상위 블록에 저장합니다. 각 블록은 그 자체로 완전한 단위입니다. 파일이 부분적으로 누락될 수 있지만 존재하는 내용은 일관성을 유지합니다.

전용 인덱스 블록이 없어도 좋지만, ZFS는 분명 어딘가에서 시작해야 합니다! 모든 데이터 트리에는 루트가 필요합니다. ZFS는 우버블록이라는 특수 블록을 사용해 파일 시스템 루트에 대한 포인터를 저장합니다. ZFS는 디스크의 데이터를 변경하지 않으며, 오히려 블록이 변경되면 수정된 데이터로 블록의 완전히 새로운 복사본을 씁니다. (이 쓰기 시 복사본 동작에 대해서는 6장에서 자세히 설명합니다.) 데이터 풀은 128개의 블록을 우버블록용으로 예약하고, 기본 풀이 변경될 때 순차적으로 사용합니다. 마지막 우버블록이 사용되면 ZFS는 처음으로 돌아갑니다.

우버블록만이 중요한 블록은 아닙니다. ZFS는 파일시스템 메타데이터 및 풀 데이터와 같은 중요한 정보가 포함된 블록을 여러 개의 동일한 블록에 복사합니다. 메인 블록이 손상되면 ZFS는 디토 블록 (ditto block, 동일한 내용을 복사한)에서 백업 복사본을 확인합니다. 동일한 블록은 별도의 디스크 또는 단일 디스크의 별도 부분에 가능한 한 서로 멀리 떨어져 저장됩니다. (ZFS는 디스크 하드웨어의 레이아웃을 볼 수 있는 특별한 기능은 없지만 대충 짐작할 수 있습니다.)

ZFS는 트랜잭션 그룹 또는 txg의 스토리지 미디어에 변경 내용을 커밋합니다. 트랜잭션 그룹은 여러 개의 일괄 변경 내용을 포함하며 64비트 숫자가 증가합니다. 각 트랜잭션 그룹은 라인의 다음 우버블록을 사용합니다. ZFS는 128개의 트랜잭션 그룹 중에서 가장 높은 트랜잭션 번호를 가진 우버블록을 찾아서 가장 최신의 우버블록을 식별합니다.

ZFS는 인덱싱을 위해 일부 블록을 사용하지만, 이러한 z노드와 d노드는 풀에 있는 모든 스토리지 블록을 사용할 수 있습니다. 파일시스템을 만들 때 할당되는 UFS2 또는 extfs 인덱스 노드와는 다릅니다.

스트라이프, RAID 및 풀 (Stripes, RAID, and Pools)

스토리지와 관련하여 스트라이프라는 단어를 여러 번 들어보셨을 것입니다. ZFS 풀은 가상 장치에 걸쳐 데이터를 "스트라이프"합니다. 전통적인 RAID는 물리적 장치에 걸쳐 데이터를 "스트라이프"합니다. 스트라이프란 무엇이며, 풀에서 어떻게 작동할까요?

스트라이프는 단일 장치에 기록되는 데이터 덩어리입니다. 대부분의 기존 RAID는 128KB 스트라이프 크기를 사용합니다. 기존 RAID 장치에 파일을 쓸 때 RAID 소프트웨어는 보통 병렬로 128KB 단위로 각 드라이브에 씁니다. 마찬가지로 기존 RAID 어레이에서 읽는 작업도 스트라이프 크기 단위로 이루어집니다. 서버의 워크로드에 맞게 스트라이프 크기를 사용자가 지정할 수 있지만, 하드웨어의 용량과 소프트웨어의 한계로 인해 스트라이프 크기가 크게 제한됩니다.

스트라이프는 중복성을 제공하지 않습니다. 기존 RAID는 패리티 및/또는 미러링을 통해 중복성을 확보합니다. ZFS 풀은 기본 VDEV에서 중복성을 확보합니다.

ZFS는 로켓으로 구동되는 롤러스케이트에 줄무늬를 넣습니다. ZFS 데이터 세트는 기본 스트라이프 크기인 128KB를 사용하지만, ZFS는 장비와 워크로드에 맞게 이 스트라이프 크기를 동적으로 변경할 수 있을 만큼 똑똑합니다. 특정 데이터 청크에는 32KB 스트라이프 크기가 적합하지만 다른 데이터 조각에는 64KB가 적합할 경우, ZFS는 각 데이터에 적합한 크기를 사용합니다. ZFS 개발자들은 최대 1MB의 스트라이프 크기에 대한 지원을 완료했습니다. 이 기능은 이미 FreeBSD-CURRENT에서 사용할 수 있으며, FreeBSD 10.2 이상에 포함될 예정입니다.

ZFS 풀은 기존 RAID보다 훨씬 더 유연합니다. 전통적인 RAID는 고정적이고 유연하지 않은 데이터 레이아웃을 가지고 있습니다(일부 하드웨어 벤더는 유연성이 더 높은 자체 독점 RAID 시스템을 가지고 있습니다). RAID 소프트웨어는 결정론적 순서로 각 디스크에 씁니다. ZFS는 더 많은 유연성을 제공합니다. 5개의 디스크로 구성된 기존 RAID 어레이를 사용하는 경우, 해당 어레이는 항상 5개의 디스크로 구성됩니다. 디스크를 추가하여 어레이를 변경할 수 없습니다. 디스크를 더 큰 디스크로 교환할 수는 있지만, 그렇게 해도 어레이의 크기는 변경되지 않습니다. RAID 장치를 생성하면 어레이의 기본 특성이 석화됩니다.

ZFS 풀은 변경을 허용할 뿐만 아니라 추가도 쉽게 허용하도록 설계되었습니다. 다섯 개의 VDEV가 있는 ZFS 풀이 있는데 여섯 번째를 추가하려는 경우 괜찮습니다. ZFS는 해당 VDEV를 수락하고 깜박임 없이 해당 장치에서 데이터 스트라이핑을 시작합니다. RAID-Z VDEV에는 스토리지를 추가할 수 없고 풀에는 VDEV만 추가할 수 있습니다. RAID-Z VDEV의 공급자 수는 생성 시 고정되어 있습니다.

하지만 ZFS를 사용하면 해당 가상 장치는 ZFS가 지원하는 모든 유형의 VDEV가 될 수 있습니다. 미러 쌍인 두 개의 VDEV를 예로 들어보겠습니다. 하나의 zpool에 넣습니다. ZFS가 데이터를 스트라이핑합니다. 기존 RAID에서는 미러 상단의 스트라이프를 RAID-10이라고 합니다. 대부분의 사용 사례에서 RAID-10은 가장 높은 성능의 RAID입니다. 그러나 기존 RAID-10의 크기가 고정되어 있는 경우 풀에 VDEV를 추가할 수 있습니다. RAID-10을 확장한다는 것은 데이터를 백업하고, RAID 어레이에 디스크를 추가하고, 데이터를 복원하는 것을 의미합니다. zpool을 확장한다는 것은 풀에 더 많은 VDEV를 추가한다는 의미입니다. 또한 RAID-10은 최대 2개의 디스크 깊이를 허용하는 반면, ZFS는 최대 264개의 디스크 깊이를 허용합니다.

하지만 풀은 중복성을 제공하지 않는다는 점을 기억하세요. 모든 ZFS 중복성은 기본 VDEV에서 제공됩니다.

풀 보기 (Viewing Pools)

시스템의 모든 풀을 보려면 **zpool list**를 실행합니다.

```
$ zpool list
NAME   SIZE  ALLOC  FREE  EXPANDSZ  FRAG  CAP  DEDUP  HEALTH  ALTROOT
db    2.72T  1.16G  2.72T      -    0%   0%  1.00x  ONLINE  -
zroot  920G  17.3G  903G       -    2%   1%  1.00x  ONLINE  -
```

첫 번째 열은 풀 이름입니다 . 이 시스템에는 **db**와 **zroot**라는 두 개의 풀이 있습니다 .
다음 세 열은 각 풀의 크기와 사용량 정보를 제공합니다 . 크기 , 사용된 공간의 양 , 사용 가능한 공간의 양이 표시됩니다 .
EXPANDSZ 열은 기본 스토리지 공급업체에 여유 공간이 있는지 여부를 보여줍니다 . 5장에서 설명한 대로 이 풀의 공간을 확장할 수 있습니다 . 이 공간에는 패리티 정보로 이동하는 블록이 포함되어 있으므로 풀을 확장해도 사용 가능한 공간이 이만큼 확보되지는 않습니다 .
FRAG 아래에서 이 풀의 조각화 양을 볼 수 있습니다 . 조각화는 파일 시스템 성능을 저하시킵니다 .
CAP 열은 사용 가능한 공간의 몇 퍼센트가 사용되었는지 보여줍니다 .
DEDUP 항목은 파일 시스템에서 발생한 중복 제거의 양을 보여줍니다 . 6장에서는 중복 제거에 대해 다룹니다 .
풀의 **HEALTH** 열은 기본 VDEV의 상태를 반영합니다 . 스토리지 공급업체에 장애가 발생하면 첫 번째 힌트는 온라인 이외의 모든 상태입니다 . 5장에서는 풀 상태에 대해 설명합니다 .
마지막으로 , **ALTROOT**는 이 풀이 마운트된 위치 , 즉 "대체 루트 "를 보여줍니다 .

4장에서는 대체 루트에 대해 다룹니다 . 특정 풀에 대한 정보를 알고 싶으면 zpool list 뒤에 풀 이름을 나열합니다 .
이 명령은 스토리지 풀 **prod** 및 **test**의 출력만 표시합니다 .

```
$ zpool list prod test
```

기본 드라이브의 사용률을 포함하여 풀에 대한 자세한 정보를 원하면 **-v** 옵션을 추가합니다 . 이 옵션은 풀 이름 앞에 붙여야 합니다 .

```
$ zpool list -v zroot
```

-p 플래그는 사람이 읽기 쉬운 형식이 아닌 바이트 단위로 숫자를 인쇄하고 , **-H**는 열 헤더를 제거합니다 .
이러한 옵션은 자동화 및 관리 스크립트에 유용합니다 .

기본 VDEV 레이아웃을 포함하여 시스템 풀을 더 자세히 보려면 **zpool status**를 사용하세요 . 풀을 만들 때 zpool 상태의 많은 예를 보게 될 것입니다 .

풀을 간단히 확인하려면 zpool status -x를 실행합니다 .

```
$ zpool status -x
all pools are healthy
```

때로는 이것만으로 충분할 때도 있습니다 .

여러 개의 VDEV (Multiple VDEVs)

풀에는 여러 개의 VDEV가 포함될 수 있습니다. VDEV를 추가하면 풀에서 사용 가능한 공간이 늘어날 뿐만 아니라 성능도 향상됩니다. 풀은 모든 쓰기를 VDEV 간에 분할합니다. 작은 파일은 단일 스트라이프만 있으면 되지만, 여러 개의 작은 파일을 쓰는 경우 ZFS는 VDEV 간에 쓰기를 분할합니다.

2장에서는 다양한 VDEV 유형의 성능에 대해 설명합니다. 이러한 성능은 풀 레벨로 올라갑니다. 여러 VDEV에서 대용량 파일을 읽는 경우, 파일 읽기는 마지막(일반적으로 가장 느린) 드라이브가 해당 데이터의 일부 호출을 완료하면 완료됩니다. 하지만 풀에 여러 대의 VDEV가 포함된 경우, 가장 느린 드라이브에는 파일의 일부만 포함되어 있으므로 파일에 액세스하는 데 필요한 시간이 다소 줄어듭니다. 스토리지 공급업체에서 데이터를 읽을 때 가장 느린 부분은 데이터를 호출할 올바른 디스크의 헤드를 찾는 것이므로 시간을 VDEV의 수로 나누는 것만큼 간단하지는 않지만, 풀에 VDEV를 추가하면 성능이 향상됩니다.

모범 사례에서는 풀에 동일한 스토리지 VDEV만 사용하는 것이 좋습니다. 풀에 미러링된 VDEV가 여러 개 있는 경우 풀에 RAID-Z3 장치를 추가하지 마세요. 혼합 스토리지 VDEV는 풀 성능을 크게 떨어뜨리고 장치 간에 데이터를 최적으로 분산하는 ZFS의 작업을 더 힘들게 만듭니다. 이 작업을 수행할 수는 있지만, 하지 않는 것이 좋습니다.

> ZFS는 유닉스의 전통에 따라 멍청한 일을 하는 것을 막지 않는데, 이는 영리한 일을 하는 것도 막는 것이기 때문입니다.

VDEV 제거 (Removing VDEVs)

현재 풀에서 VDEV를 제거할 수 없습니다. 각 VDEV에는 데이터가 있습니다. 특정 유형의 VDEV에서 디스크를 제거할 수 있지만, VDEV 전체에는 중요한 풀 데이터가 남아 있습니다. 예를 들어, 미러 VDEV에서 디스크를 제거할 수는 있지만 풀에서 전체 VDEV를 제거할 수는 없습니다. 일반적으로는 장애가 발생한 경우에만 VDEV에서 디스크를 제거합니다. 스토리지 공급업체를 끌어당기는 등 풀에서 강제로 VDEV를 제거하면 풀이 파괴됩니다. 스트라이프 또는 미러 풀에서 VDEV를 제거하는 기능은 2015년 말에 OpenZFS에 제공될 예정이지만 아직은 불가능합니다. RAID-Z 장치 제거에 대한 지원은 로드맵에 있지만 아직 작업이 시작되지 않았습니다. 즉, 풀을 축소할 수 없습니다.

풀을 더 작게 만들려면 해당 풀의 데이터를 새롭고 더 작은 풀로 옮긴 다음 원래 풀에서 디스크를 재활용해야 합니다.

풀 정렬 및 디스크 섹터 크기 (Pools Alignment and Disk Sector Size)

ZFS는 기본 제공업체의 섹터 크기를 포함하여 스토리지 매체에 대한 심층적인 지식이 있어야 합니다. 풀이 올바른 섹터 크기를 사용하지 않거나 ZFS의 섹터가 디스크의 물리적 섹터에 정렬되지 않으면 스토리지 성능이 절반 이상 저하됩니다. 이 두 가지 문제는 직교하는 문제이지만, 둘 중 하나라도 계획하지 않으면 시스템이 망가질 수 있습니다.

파티션 정렬과 ZFS 섹터 크기에 대해서는 별도로 설명하겠습니다.

파티션 정렬 (Partition Alignment)

디스크는 섹터 크기를 보고하므로 문제가 되는 경우를 제외하고는 문제가 되지 않습니다. 많은 디스크가 512바이트 섹터를 가지고 있다고 보고하지만, 실제로는 4096바이트(4K) 섹터를 가지고 있습니다. FreeBSD Mastery: Storage Essentials에서 이에 대해 자세히 다루고 있으므로 여기서는 자세히 다루지 않겠습니다.

유서 깊은 마스터 부트 레코드(MBR)와 같은 오래된 파티션 관리 체계는 디스크 파티션이 디스크의 물리적 특성에 부합하는지 확인하기 위해 온갖 종류의 복잡한 수학을 포함했습니다. GPT(GUID Partition Table)와 같은 최신 파티션 체계는 물리적 디스크가 서로 다른 언어를 사용한다는 것을 알고 있으며, 이러한 오래된 MBR 기반 제한이 완전히 엉터리라는 것을 알기 때문에 파티션이 완전한 섹터를 채우도록 요구합니다.

하지만 디스크가 섹터 크기에 대해 거짓말을 하는 경우, **gpart(8)**을 사용하면 물리적 섹터의 중간에서 시작하거나 끝나는 파티션을 만들 수 있습니다. 디스크를 읽거나 쓸 때마다 두 개의 물리적 섹터를 건드려야 합니다. 이는 성능에 큰 영향을 미칩니다.

특정 SSD는 파티션이 128KB 또는 1MB 경계를 따라 정렬될 것을 기대하기도 합니다.

정렬 문제를 피하는 쉬운 방법은 모든 GPT 파티션을 메가바이트 경계에서 시작하고 끝내도록 하는 것입니다. gpart 추가 명령에 **-a 1m** 인수를 추가하세요.

ZFS 섹터 크기 (ZFS Sector Size)

ZFS는 기본적으로 512바이트 섹터 크기를 가정합니다. 물리적 512바이트 섹터가 있는 디스크에서 512바이트 파일 시스템 섹터 크기를 사용하는 것은 전혀 문제가 없습니다. 4K-섹터 디스크에 512바이트 파일 시스템 섹터를 사용하면 하드웨어가 더 많이 작동합니다. 이러한 디스크에 4KB의 데이터를 쓰고 싶다고 가정해 보겠습니다. 하드 드라이브에 하나의 물리적 섹터를 쓰도록 지시하는 대신 섹터의 첫 번째 8분의 1을 수정하고, 두 번째 8분의 1을 수정하고, 세 번째 8분의 1을 수정하는 식으로 수정하라는 지시를 내립니다. 4KB 섹터에 512바이트 쓰기를 수행한다는 것은 4KB 전체를 읽고, 작은 부분을 수정한 다음 다시 쓰는 것을 의미합니다. 이는 전체 섹터를 덮어쓰는 것보다 훨씬 느립니다. 성능이 급감합니다. 512바이트 섹터가 있는 디스크에서 ZFS가 4K 섹터 크기를 사용하는 경우, 디스크 하드웨어는 액세스 요청을 물리적 섹터 크기로 분할하여 성능 비용을 거의 들이지 않습니다.
더 큰 섹터 크기를 사용한다고 해서 성능에 영향을 주지는 않지만, 작은 파일을 많이 저장할 경우 공간 효율성이 떨어집니다. 1KB 파일이 여러 개 있는 경우 각 파일이 하나의 섹터를 차지합니다.

ZFS 섹터 크기는 풀에 있는 각 가상 장치의 속성입니다. 풀을 내보내거나 고장난 드라이브를 교체할 때에도 가상 장치의 섹터 크기는 변경할 수 없습니다.

이 두 가지 사실을 종합하면, 기본 디스크에서 보고하는 섹터 크기에 관계없이 ZFS가 4K-섹터를 사용하도록 강제하는 것이 거의 항상 바람직하다는 것을 의미합니다. 더 큰 ZFS 섹터 크기를 사용해도 특정 데이터베이스 작업을 제외하고는 성능이 저하되지 않으며, 그 경우에도 실제로 512바이트 섹터를 사용하는 디스크를 사용하는 경우에만 성능이 저하되지 않습니다.

ZFS는 가장 큰 섹터 크기를 보고하는 장치의 섹터 크기를 사용합니다. 모든 장치가 512바이트 섹터를 사용한다고 주장하고 더 큰 섹터 크기를 설정하지 않으면 해당 장치로 구축된 가상 장치가 512바이트 섹터를 사용합니다. 4096바이트 섹터를 가진 단일 디바이스를 VDEV에 포함하면 ZFS가 4096바이트 섹터를 사용하도록 강제합니다.

4K-섹터 디바이스가 섹터 크기를 보고한다고 믿지 마세요. 항상 4K-섹터를 사용하도록 ZFS에 지시하세요.

ashift라는 풀 변수가 섹터 크기를 제어합니다. ashift가 9이면 ZFS가 512바이트 섹터를 사용하도록 지시합니다. ashift가 12이면 ZFS가 4096바이트 섹터를 사용하도록 지시합니다. (왜 9와 12일까요? 2^9는 512이고, 2^{12}는 4096입니다.) ashift를 설정하는 방법은 FreeBSD 릴리스에 따라 다릅니다.

> 모두가 '9'를 보고 2^9를 생각하잖아요?

FreeBSD 10.1 및 새로운 Ashift (FreeBSD 10.1 and Newer Ashift)

/etc/sysctl.conf 또는 명령줄에서 **sysctl vfs.zfs.min_auto_ashift**를 사용하여 시스템의 기본 ashift를 설정합니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
```

설치 중에 명령줄을 사용하지만, 새 풀을 만들 때 잊지 않도록 **/etc/sysctl.conf**에 영구적으로 설정할 수도 있습니다.

이 책의 예제에서는 FreeBSD 10.1 이상을 사용한다고 가정합니다. 이전 FreeBSD 버전의 경우, sysctl을 설정하는 대신 매번 ashift를 설정해야 합니다.

구형 FreeBSD Ashift (Older FreeBSD Ashift)

10.1 이전의 FreeBSD 버전에는 최신 FreeBSD 버전에 있는 ashift sysctl이 없기 때문에, ZFS의 내부 섹터 크기 감지 코드에 의존해야 합니다. 이 코드는 기본 스토리지 매체, 즉 스토리지 공급자로부터 섹터 크기를 읽습니다.

이 사례는 공급자와 디스크의 중요한 차이점을 강조합니다. FreeBSD에서는 GEOM 모듈 **gnop(8)**을 사용하여 패스스루 장치를 만들 수 있습니다. gnop 모듈을 사용하면 저장 장치 사이에 임의의 데이터를 삽입할 수 있습니다(이 경우 섹터 크기를 강제 적용). "모든 것을 투명하게 통과시키되, 4096바이트 섹터 크기를 고집합니다."라는 gnop 장치를 생성합니다. 이 장치를 사용하여 zpool을 만듭니다. 여기서는 **/dev/gpt/zfs0**이라는 파티션에 gnop 장치를 추가합니다.

```
$ gnop create -S 4096 /dev/gpt/zfs0
```

이렇게 하면 **/dev/gpt/zfs0.nop** 장치가 생성됩니다. 이 공급자를 VDEV의 한 멤버로 사용하면 ZFS가 해당 VDEV의 섹터 크기를 선택합니다. 이 장의 나머지 부분에서는 다양한 ZFS 풀을 생성하는 방법에 대해 설명하지만, 여기서는 미러 풀을 생성할 때 이 장치를 사용하는 예제를 살펴보겠습니다.

```
$ zpool create compost mirror gpt/zfs0.nop gpt/zfs1
```

gnop(8)로 생성된 프로바이더는 임시적이며 재부팅 시 사라집니다. 하지만 **gnop(8)**이 모든 것을 디바이스로 전달하기 때문에 ZFS는 기본 디바이스에서 메타데이터를 찾습니다. 이미 섹터 크기가 설정되어 있으므로 ZFS는 더 이상 디스크의 섹터 크기를 감지하려고 시도하지 않습니다.

풀과 가상 디바이스 만들기 (Creating Pools and VDEVs)

zpool(8)을 사용하여 풀과 가상 디바이스를 동시에 생성합니다 . 또한 **zpool(8)**을 사용하여 기존 풀에 VDEV를 추가하고 고장난 디바이스를 교체할 수도 있지만 , 이 모든 내용은 5장에서 다루겠습니다 . 여기서는 각 RAID-Z 장치에 스트라이프 풀 , 미러 및 풀을 생성하겠습니다 . 2장에서는 각 VDEV 유형에 대해 설명합니다 .

원하는 만큼의 풀을 생성하기 전에 ashift를 한 번만 설정하면 됩니다 . 풀을 생성할 때마다 재설정할 필요는 없습니다 . 하지만 대부분의 독자가 자신이 생성하는 풀 유형에 대한 항목을 찾을 때까지 이 책을 건너뛸 것으로 예상되므로 모든 항목에 "set ashift"을 나열했습니다 .

샘플 드라이브 (Sample Drives)

0장에서는 고장난 하드웨어를 쉽게 식별할 수 있도록 물리적 위치와 일련 번호로 드라이브에 라벨을 붙일 것을 권장합니다 . 프로덕션에서는 매우 유용합니다 . 하지만 책에서는 장치 이름이 길면 이해하기 어렵습니다 . 이 예제에서는 zfs와 숫자로 된 GPT 레이블을 사용합니다 . 이 장에서는 각각 1GB 스왑 파티션과 **gpart(8)**로 만든 대형 ZFS 파티션이 있는 6개의 1TB 드라이브를 사용합니다 .

```
$ gpart create -s gpt da0
$ gpart add -a 1m -s1g -l sw0 -t freebsd-swap da0
$ gpart add -a 1m -1 zfs0 -t freebsd-zfs da0
```

결과 디스크의 파티션은 다음과 같습니다 .

```
$ gpart show -l da0
=>     40  1953525088  da0  GPT  (932G)
       40       2008    -  free -  (1.0M)
     2048    2097152    1  sw0  (1.0G)
  2099200  1951424512    2  zfs0  (931G)
1953523712       1416    -  free -  (708K)
```

우리는 GPT 레이블로 ZFS 풀을 관리하므로 예제에서는 gpt/zfs0부터 gpt/zfs5까지를 참조합니다 . 프로덕션 환경에서는 디스크를 실제 위치에 매핑하는 의미 있는 레이블을 사용하세요 .

스트라이프 풀 (Striped Pools)

일부 스토리지 풀은 중복성이 필요하지 않지만 많은 공간이 필요합니다 . 엔지니어링 및 물리 계산을 위한 스크래치 파티션이 이러한 종류의 스토리지의 일반적인 사용 사례입니다 . 풀 이름인 **zpool create**를 사용해 풀에 있는 장치를 나열합니다 . 풀을 만들기 전에 애쉬시프트를 설정하는 것을 잊지 마세요 . 여기서는 5개의 스토리지 공급자로 구성된 풀을 생성합니다 .

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create compost gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs4 gpt/zfs4
```

명령이 성공하면 아무런 출력도 돌아오지 않습니다 . 풀이 **zpool status**로 존재하는지 확인합니다 .

```
$ zpool status
  pool: compost
 state: ONLINE
  scan: none requested
config:

NAME       STATE READ WRITE CKSUM
compost    ONLINE   0     0     0
 gpt/zfs0  ONLINE   0     0     0
 gpt/zfs1  ONLINE   0     0     0
 gpt/zfs2  ONLINE   0     0     0
 gpt/zfs3  ONLINE   0     0     0
 gpt/zfs4  ONLINE   0     0     0
```

5개의 공급자가 모두 표시됩니다. 각 공급자는 자체 VDEV입니다.

이 정도 규모의 시스템에서는 상당히 큰 풀입니다. 이 풀은 모든 구성원 VDEV에 걸쳐 데이터를 스트라이핑하지만, VDEV에는 중복성이 없습니다. 대부분의 실제 애플리케이션에는 중복성이 필요합니다. 가장 간단한 종류의 중복성은 미러입니다.

미러 풀 (Mirrored Pools)

미러링된 장치는 모든 데이터를 여러 스토리지 공급업체에 복사합니다. 미러의 한 공급업체에 장애가 발생해도 풀에는 여전히 다른 데이터 사본이 있습니다. 기존 미러에는 두 개의 디스크가 있지만, 그 이상도 가능합니다.

동일한 **zpool create** 명령과 풀 이름을 사용합니다. **mirror** 키워드를 사용해 시스템 시프트 저장 장치를 설정합니다. 풀을 생성하기 전에 시스템 ashift를 설정하세요.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create reflect mirror gpt/zfs0 gpt/zfs1
```

zpool status로 풀의 구성을 확인합니다.

```
$ zpool status
  pool: reflect
 state: ONLINE
  scan: none requested
config:

NAME        STATE READ WRITE CKSUM
reflect    ONLINE   0     0     0
```

```
  mirror-0   ONLINE   0     0     0
   gpt/zfs0  ONLINE   0     0     0
   gpt/zfs1  ONLINE   0     0     0

errors: No known data errors
```

zpool 명령은 여기에 mirror-0이라는 새 계층을 만들었습니다. mirror-0 항목은 VDEV입니다. 이 VDEV에는 두 개의 장치, 즉 **gpt/zfs0**과 **gpt/zfs1**이 포함되어 있습니다. 필요에 따라 여러 개의 디스크로 미러를 구성할 수 있습니다. 사본이 너무 많은 것이 없는 것보다 낫습니다.

```
$ zpool create reflect mirror gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs3
```

그러나 이것은 너무 지나친 예가 될 수 있습니다 (FreeBSD Mastery: Advanced ZFS에서 미러를 여러 풀로 분할하는 것에 대해 설명합니다).

RAID-Z Pools

미러에서 얻을 수 있는 중복성은 빠르고 안정적이지만, 매우 복잡하거나 흥분되지는 않습니다. RAID-Z는 복잡성 대가로 더 큰 유연성을 제공하므로 더 흥미롭습니다. 다른 zpool과 마찬가지로 **zpool create**을 실행하고 풀 이름, 유형 및 저장 장치를 지정하여 RAID-Z 풀을 생성합니다. 여기서는 RAID-Z(또는 RAID-Z1) 풀을 생성합니다.

> 시스템 관리에서 '흥분'은 나쁜 단어입니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create raidz1 gpt/zfs0 gpt/zfs1 gpt/zfs2
```

새 풀의 상태에는 세 개의 공급자가 있는 raidz1-0이라는 새 VDEV가 표시됩니다.

```
$ zpool status bucket
  pool: bucket
 state: ONLINE
  scan: none requested
config:

NAME        STATE READ WRITE CKSUM
bucket      ONLINE   0     0     0
 raidz1-0   ONLINE   0     0     0
  gpt/zfs0  ONLINE   0     0     0
  gpt/zfs1  ONLINE   0     0     0
  gpt/zfs2  ONLINE   0     0     0
```

이 풀에 있는 디스크 중 하나에 장애가 발생해도 데이터는 그대로 유지됩니다. 다른 RAID-Z 레벨에는 훨씬 더 많은 중복성이 있습니다. 여기서는 6개의 공급자를 raidz3-Z3으로 끌어옵니다. RAID-Z3를 만들 때와 RAID-Z1을 만들 때의 유일한 차이점은 raidz3와 필요한 추가 장치의 사용 여부입니다.

```
$ zpool create bucket raidz3 gpt/zfs0 gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4 gpt/zfs5
```

지금쯤이면 짐작할 수 있겠지만, 풀의 상태에는 raidz3-0이라는 새 장치가 표시됩니다.

```
$ zpool status
  pool: bucket
 state: ONLINE
  scan: none requested
config:

NAME        STATE  READ WRITE CKSUM
bucket      ONLINE    0     0     0
 raidz3-0   ONLINE    0     0     0
  gpt/zfs0  ONLINE    0     0     0
  gpt/zfs1  ONLINE    0     0     0
...
```

이러한 풀에는 모두 단일 VDEV가 있습니다. 하지만 여러 개의 VDEV를 원한다면 어떻게 해야 할까요?

다중 VDEV 풀 (Multi-VDEV Pools)

여러 개의 VDEV가 있는 풀을 만들 수 있습니다. *mirror, raidz, raidz2, raidz3* 키워드는 모두 **zpool(8)**에 새 VDEV를 생성하도록 지시합니다. 이러한 키워드 중 하나 뒤에 나열된 모든 스토리지 공급자는 해당 VDEV의 새 인스턴스를 생성합니다. 키워드 중 하나가 다시 나타나면 **zpool(8)**은 새 VDEV로 시작합니다.

이 장의 시작 부분에서는 기존 RAID-10 설정을 시뮬레이션하여 여러 미러에 걸쳐 스트라이핑하는 방법을 다루었습니다. 여기서는 정확히 그렇게 합니다.

```
$ sysctl vfs.zfs.min_auto_ashift=12
$ zpool create barrel mirror gpt/zfs0 gpt/zfs1 mirror gpt/zfs2 gpt/zfs3
```

처음 세 단어인 **zpool create barrel**은 **zpool(8)**에 barrel이라는 이름의 새 풀을 인스턴스화하라고 지시합니다. **mirror** 키워드는 "미러 생성"을 의미합니다. 그러면 두 개의 스토리지 프로바이더, 즉 **gpt/zfs0**과 **gpt/zfs1**이 생깁니다. 이 스토리지 공급자는 첫 번째 미러로 이동합니다. **mirror**라는 단어가 다시 나타나면서 이전 VDEV가 완료되었고 새 VDEV에서 시작한다는 것을 **zpool(8)**에 알려줍니다. 두 번째 VDEV에도 두 개의 스토리지 프로바이더, 즉 **gpt/zfs2**와 **gpt/zfs3**가 있습니다. 이 풀의 상태는 이전에 보았던 것과는 다릅니다.

```
$ zpool status barrel
  pool: barrel
 state: ONLINE
  scan: none requested
config:


NAME       STATE READ WRITE CKSUM
barrel     ONLINE   0     0     0
 mirror-0  ONLINE   0     0     0
  gpt/zfs0 ONLINE   0     0     0
  gpt/zfs1 ONLINE   0     0     0
 mirror-1  ONLINE   0     0     0
  gpt/zfs2 ONLINE   0     0     0
  gpt/zfs3 ONLINE   0     0     0
```

풀에는 mirror-0과 mirror-1이라는 두 개의 VDEV가 있습니다. 각 VDEV에는 두 개의 저장 장치가 포함됩니다. ZFS는 모든 VDEV에 걸쳐 데이터를 스트라이핑합니다. 미러에 대한 스트라이핑은 RAID-10입니다. 일반적인 RAID와 동등한 방식이 없는 방식으로 다중 VDEV 풀을 배열할 수도 있습니다. FreeBSD의 일부 GEOM 클래스와 같은 소프트웨어 RAID 시스템을 사용하면 유사한 RAID를 구축할 수 있지만, 하드웨어 RAID 카드에서는 이러한 기능을 찾을 수 없습니다. 여기서는 두 개의 RAID-Z1 VDEV에 걸쳐 데이터를 스트라이핑하는 풀을 생성합니다.

```
$ zpool create vat raidz1 gpt/zfs0 gpt/zfs1 gpt/zfs2 raidz1 gpt/zfs3 gpt/zfs4 gpt/zfs5
```

첫 번째 RAID-Z1 VDEV에는 세 가지 스토리지 공급업체, 즉 **gpt/zfs0, gpt/zfs1, gpt/zfs2**가 포함됩니다. 두 번째는 **gpt/zfs3, gpt/zfs4, gpt/zfs5**를 포함합니다. zpool은 두 공급자에 걸쳐 데이터를 스트라이핑합니다. 이렇게 하면 두 개의 RAID-Z 장치가 포함된 풀이 생성됩니다.

```
$ zpool status vat
...
config:


NAME       STATE READ WRITE CKSUM
vat       ONLINE   0     0     0
 raidz1-0  ONLINE   0     0     0
  gpt/zfs0 ONLINE   0     0     0
  gpt/zfs1 ONLINE   0     0     0
  gpt/zfs2 ONLINE   0     0     0
 raidz1-1  ONLINE   0     0     0
  gpt/zfs3 ONLINE   0     0     0
  gpt/zfs4 ONLINE   0     0     0
```

```
gpt/zfs5 ONLINE   0     0     0
```

각 VDEV에는 고유한 중복성이 있습니다 .

미러가 RAIDZ보다 빠르긴 하지만 , 여러 대의 VDEV를 사용하면 속도가 빨라져 이 RAIDZ 기반 풀이 워크로드에 충분히 빠르고 훨씬 더 많은 공간을 제공할 수 있습니다 . 이를 확인할 수 있는 유일한 방법은 풀을 생성하고 워크로드를 테스트하는 것입니다 .

풀은 모든 쓰기 요청을 풀에 있는 VDEV 간에 분할한다는 점을 기억하세요 . 하나의 작은 파일은 하나의 VDEV에만 전송될 수 있지만 , 전체적으로는 쓰기 요청이 VDEV 간에 분할됩니다 . 여러 개의 VDEV를 사용하면 IOPS와 처리량 대역폭이 증가합니다 .

로그 장치 사용 (Using Log Devices)

2장에서 설명한 대로 , ZFS는 전용 쓰기 캐시 장치 및 /또는 전용 읽기 캐시 장치를 사용하여 성능을 향상시킬 수 있습니다 . 이러한 전용 장치는 일반적으로 매우 빠르고 내구성이 높은 SSD입니다 . **zpool(8)** 명령은 쓰기 캐시를 **log**, 읽기 캐시를 **cache**로 호출합니다 . 풀을 만들 때 **log** 및 **cache** 키워드를 사용하여 이러한 장치를 지정합니다 . 여기서는 읽기 및 쓰기 캐시가 모두 포함된 스크래치 라는 이름의 스트라이프 풀을 만듭니다 .

```
$ zpool create scratch gpt/zfs0 log gtp/zlog0 cache gpt/zcache1
```

로그 장치가 풀의 상태에 표시됩니다 .

```
$ zpool status scratch
...
config:

NAME         STATE READ WRITE CKSUM
scratch      ONLINE   0     0     0
 gpt/zfs0    ONLINE   0     0     0
logs
 gpt/zlog0   ONLINE   0     0     0
cache
 gpt/zcache1 ONLINE   0     0     0
```

고가용성이 필요한 시스템에서는 이러한 쓰기 캐시를 미러링할 수 있습니다 . 읽기 캐시를 미러링하는 것은 큰 의미가 없습니다 . 읽기 캐시가 손실되면 ZFS는 실제 풀에서 읽기로 돌아가기 때문입니다 . 그러나 ZIL 쓰기 로그가 손실되면 데이터 손실이 발생할 수 있으므로 미러링하는 것이 좋습니다 . 여기서는 *gpt/zfs0*부터 *gpt/zfs3*까지 장치를 사용해 두 개의 미러 스트라이프를 만들고 , 미러링된 로그 장치로 *gpt/zlog0*과 *gpt/zlog1*을 사용합니다 .

```
$ zpool create db mirror gpt/zfs0 gpt/zfs1 mirror gpt/zfs2 gpt/zfs3 log mirror gpt/zlog0 gpt/zlog1
```

기존 풀에 인텐트 로그 및 읽기 캐시 장치를 추가하거나 제거할 수 있습니다 . 이러한 장치의 성능 향상이 꼭 필요한지 확실하지 않은 경우 , 장치 없이 풀을 실행해 보세요 . 하지만 나중에 SSD 저장 장치를 추가할 수 있는 공간이 하드웨어에 있는지 확인하세요 !

일치하지 않는 VDEV (Mismatched VDEVs)

풀 내에서 서로 다른 VDEV 유형을 사용하는 것은 바람직하지 않으며 , **zpool(8)**은 이러한 장애를 방지하려고 시도합니다 .

```
$ zpool create daftie raidz gpt/zfs0 gpt/zfs1 gpt/zfs2 mirror gpt/zfs3 gpt/zfs4 gpt/zfs5
invalid vdev specification
use '-f' to override the following errors:
mismatched replication level: both raidz and mirror vdevs are present
```

zpool(8) 명령은 실수를 지적한 다음 어떻게 고쳐야 하는지 알려줍니다 . 일반적으로 이런 종류의 오류는 시스템 관리자에게 카페인이 더 필요하다는 의미로 받아들이지만 , 정말 의도한 것일 수도 있습니다 . 지정된 VDEV 유형과 스토리지 공급자로 **zpool create -f**를 실행하면 ZFS는 사용자가 완전히 잘못된 풀을 만들려고 했다는 것을 알 수 있습니다 . 이봐요 , 이건 당신 시스템이고 당신이 책임자예요 .

ZFS가 원하지 않는 작업을 수행하면 안 됩니다 . **-f**를 사용하면 ZFS가 처리하도록 설계되지 않은 것을 만들게 됩니다 . 제대로 작동하지 않고 복구할 수 없는 풀을 쉽게 만들 수 있습니다 .

공급자 재사용 (Reusing Providers)

올바른 풀을 만들기 위해 풀을 두 번 이상 생성하고 파괴하는 경우가 있습니다 . 한 머신에서 디스크를 가져와서 다른 머신에 마운트할 수도 있습니다 . 이전에 사용했던 디스크를 발견할 때도 있습니다 .

```
$ zpool create db gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4
invalid vdev specification
use '-f' to override the following errors:
/dev/gpt/zfs3 is part of exported pool 'db'
```

이 디스크를 나중에 내보낸 다른 풀에서 사용했습니다 (5장 참조). 문제가 발생한 디스크는 해당 풀에서 사용되었고 ZFS 레이블은 디스크에 그대로 남아있었습니다 . 파티션 테이블을 지우고 다시 만들었지만 , 새 파티션 테이블은 이전 파티션 테이블과 정확히 동일합니다 . 이 경우 ZFS는 이전 메타데이터를 쉽게 찾을 수 있습니다 .

이 공급업체에 중요한 정보가 없는 것이 확실하다면 지침에 따라 **-f**를 사용하여 새 풀을 강제로 생성하세요 .

```
$ zpool create -f db gpt/zfs1 gpt/zfs2 gpt/zfs3 gpt/zfs4
```

ZFS 프로그램은 명령줄 플래그가 어디로 가는지 매우 까다로울 수 있으므로 **-f**가 create 바로 뒤에 오는지 확인하세요 .

풀 무결성 (Pool Integrity)

ZFS에 대한 일반적인 불만 중 하나는 **fsck(8)**과 같은 파일 시스템 검사기가 없다는 것입니다. 온라인 풀 무결성 검사기가 **fsck(8)**이 검사하는 모든 것을 검증하기 때문에 오프라인 파일 검사기는 ZFS를 개선하지 못합니다. 또한 온라인 검사기는 기존 파일 시스템보다 훨씬 더 효과적입니다. 이제 ZFS가 파일 무결성을 보장하는 방법과 풀 스크러빙이 무결성을 유지하는 데 어떻게 도움이 되는지 알아보겠습니다.

ZFS 무결성 (ZFS Integrity)

저장 장치가 망가집니다. 어떤 종류의 디스크든 수조 개의 섹터가 있다면, 술에 취해 비틀거리며 돌아다니는 디스크에 우주선이 부딪힐 확률은 물론 쓰기 오류, 정전, 케이블 단락 등 다양한 문제가 발생할 확률이 훨씬 더 높아집니다. 어떤 파일 시스템도 기본 하드웨어의 오류를 막을 수는 없습니다.

ZFS는 거의 모든 곳에서 해시 를 사용합니다. 해시는 데이터 덩어리를 가져와서 고정 길이의 문자열을 계산하는 수학적 알고리즘입니다. 해시의 흥미로운 점은 원본 데이터를 조금만 변경해도 데이터의 해시가 크게 달라진다는 것입니다. 각 스토리지 블록에는 상위 블록의 해시가 포함되며, 각 상위 블록에는 모든 하위 블록의 해시가 포함됩니다.

ZFS는 스토리지 공급자 오류를 방지할 수는 없지만, 이러한 해시를 사용하여 오류를 감지합니다. 시스템이 데이터에 액세스할 때마다 체크섬을 확인합니다. ZFS는 운영 체제에 수정된 파일을 제공하기 전에 풀의 중복성을 사용하여 오류를 복구합니다. 이를 자가 치유 **(self-healing)**라고 합니다.

기본 VDEV에 중복성이 있는 경우, ZFS는 RAID-Z에서 손상된 블록을 재구성하거나 미러에서 온전한 사본을 가져옵니다. 미러의 양쪽 모두에 오류가 있는 경우, 두 디스크에 동일한 데이터가 손상되지 않는 한 ZFS는 파일을 복구할 수 있습니다. VDEV에 중복성이 없지만 데이터 세트에 데이터의 여분의 복사본이 있는 경우 (4장 참조), ZFS는 그 여분의 복사본을 대신 사용합니다.

기본 VDEV에 중복성이 없고 데이터 세트에 여분의 복사본이 보관되어 있지 않은 경우, 풀은 파일이 손상되었음을 알리고 잘못된 데이터를 반환하는 대신 오류를 반환합니다. 백업에서 해당 파일을 복원하거나 버릴 수 있습니다.

ZFS는 파일 무결성 검사를 수행하는 동시에 스토리지 블록 간의 연결도 확인합니다. 이 작업은 기존 파일 시스템에서 **fsck(8)**가 수행하는 작업입니다. 이 작업은 데이터 검증의 작은 부분이며, ZFS는 정상 작동의 일부로 이 작업을 지속적으로 수행합니다. ZFS는 사용되거나 사용되지 않는 아이노드가 아니라 실제로 존재하는 블록만 검사한다는 점에서 **fsck(8)**에 비해 추가적인 이점이 있습니다. 풀의 모든 데이터에 대해 전체 무결성 검사를 수행하려면 풀을 스크러빙하세요.

해시 기반 무결성 검사의 좋은 점은 모든 종류의 오류, 심지어 예상치 못한 오류까지 잡아낸다는 것입니다. 정상적인 파일시스템은 모두 비슷하지만, 불행한 파일시스템은 모두 나름의 방식으로 불행하다는 사실을 기억하세요.

Scrubbing ZFS

ZFS 풀의 스크럽 (scrub)은 풀에 있는 모든 데이터 블록의 암호화 해시를 확인합니다. 스크럽이 오류를 감지하면 충분한 복원력이 있는 경우 오류를 복구합니다. 스크럽은 풀이 온라인 상태이고 사용 중일 때 수행됩니다.

풀에서 데이터 오류가 확인된 경우 , 해당 데이터는 zpool의 상태에 표시됩니다 . 이전에 스크럽을 실행한 적이 있다면 스캔 라인에 해당 정보도 표시됩니다 .

```
...
scan: scrub repaired 0 in 15h57m with 0 errors on Sun Feb  8 15:57:55 2015
...
errors: No known data errors
...
```

이 풀은 액세스한 데이터에서 오류가 발생하지 않았습니다 . 오류가 발견되었다면 자체적으로 복구했을 것입니다 . 그러나 이 풀은 모든 데이터에서 오류를 검사한 것이 아니라 요청받은 데이터만 검사했습니다 . 전체 풀에서 오류를 체계적으로 검색하려면 스크럽을 사용합니다 . 풀 이름과 함께 zpool 스크럽을 실행합니다 .

```
$ zpool scrub zroot
```

스크럽은 백그라운드에서 실행됩니다 . **zpool status**를 실행하여 진행 상황을 확인할 수 있습니다 .

```
$ zpool status
...
scan: scrub in progress since Tue Feb 24 11:52:23 2015
12.8G scanned out of 17.3G at 23.0M/s, 0h3m to go
0 repaired, 74.08% done
...
```

스토리지를 스크러빙하는 ZFS 풀은 평소보다 느리게 실행됩니다 . 시스템이 이미 성능 한계에 도달한 경우에는 사용량이 많지 않은 시간에만 풀을 스크럽하세요 . 진행 중인 스크럽을 취소해야 하는 경우 , **zpool scrub -s**를 실행합니다 .

```
$ zpool scrub -s zroot
```

돌아가서 가능한 한 빨리 시스템에서 스크럽을 완료하도록 하세요 .

스크럽 빈도 (Scrub Frequency)

ZFS에 내장된 무결성 테스트와 복원력 덕분에 대부분의 오류는 복원력이 작동할 수 있을 만큼 조기에 발견된다면 수정할 수 있습니다 . 즉 , 하드웨어의 품질에 따라 호스트의 풀을 얼마나 자주 스크러빙해야 하는지가 결정됩니다 . 소위 "서버급 " 장비와 같이 신뢰할 수 있는 하드웨어를 사용하는 경우 분기별로 스크러빙하는 것으로 충분합니다 . 값싼 하드웨어를 남용하는 경우 한 달에 한 번 정도 스크러빙해야 합니다 .

이 장의 뒷부분에 있는 "ZFS 유지 관리 자동화 "에서 설명한 대로 FreeBSD는 정기적인 스크럽을 수행할 수 있습니다 .

풀 속성 (Pool Properties)

ZFS는 속성을 사용하여 풀의 특성을 표현합니다. zpool 속성은 데이터 집합의 속성과 매우 유사하게 보이고 작동하며 많은 속성이 둘 사이에 겹치는 것처럼 보이지만, 데이터 집합 속성은 풀 속성과 아무런 관계가 없습니다. 풀 속성에는 풀의 상태, 크기, 용량 및 풀별 기능과 같은 사실이 포함됩니다.

풀의 속성은 전체 풀에 영향을 줍니다. 풀의 일부에만 속성을 설정하려면 필요에 맞는 데이터 세트별 속성을 확인하세요.

풀 속성 보기 (Viewing Pool Properties)

시스템에 있는 모든 풀의 모든 속성을 보려면 **zpool get all**을 실행합니다. 특정 풀의 속성만 보려면 끝에 풀 이름을 추가하면 됩니다. 여기서는 **zroot** 풀에 대한 속성을 봅니다.

```
$ zpool get all zroot
NAME   PROPERTY  VALUE   SOURCE
zroot  size      920G    -
zroot  capacity  1%      -
zroot  altroot   -       default
zroot  health    ONLINE  -
...
```

처음 두 열은 풀 이름과 속성 이름을 나타냅니다.

세 번째 열에는 속성 값이 나열됩니다. 이 값은 사용 또는 사용 안 함, 켜짐 또는 꺼짐, 활성 또는 비활성 등과 같은 값일 수도 있고 값일 수도 있습니다. 이 풀의 크기 속성은 920G로, 이 풀에는 920GB의 공간이 있습니다.

SOURCE 열에는 이 속성이 설정된 위치가 표시됩니다. 이 열은 대시 하나 또는 기본 또는 로컬이라는 단어로 표시될 수 있습니다. 대시는 이 속성이 자체적으로 설정된 것이 아니라 어떻게든 풀에서 읽혀진 것임을 의미합니다. 풀의 크기나 해당 공간의 사용량에 대한 값은 설정하지 않습니다. FreeBSD는 풀에서 이러한 값을 계산합니다. 기본값의 출처는 이 속성이 기본값으로 설정되어 있음을 나타내며, 로컬은 이 속성이 이 풀에 특별히 설정되었음을 의미합니다.

단일 속성을 가져오려면 속성 이름과 함께 **zpool get**을 실행합니다.

```
$ zpool get size
NAME   PROPERTY  VALUE  SOURCE
db     size      2.72T  -
zroot  size      920G   -
```

마지막에 풀 이름을 지정하여 범위를 좁히세요.

풀 속성 변경 (Changing Pool Properties)

이 책 전체에서 풀 동작을 변경하기 위해 속성을 설정하겠습니다. **zpool set** 명령을 사용하여 풀의 속성을 변경합니다. 여기서는 풀의 **comment** 속성을 설정합니다.

```
$ zpool set comment="Main OS files" zroot
```

이제 이 코멘트가 속성 목록에 표시됩니다.

```
$ zpool get comment
NAME   PROPERTY  VALUE         SOURCE
db     comment   -             default
zroot  comment   Main OS files  local
```

여기서 SOURCE 열에 주목하세요. 기본적으로 풀에는 주석이 없습니다. 하지만 코멘트를 설정했으므로 소스가 로컬로 변경됩니다. 속성의 소스가 기본값에서 로컬로 변경되면 영원히 로컬로 유지됩니다. 속성을 기본값으로 설정해도 소스는 변경되지 않습니다.

```
$ zpool set comment="-" zroot
# zpool get comment
NAME   PROPERTY  VALUE  SOURCE
db     comment   -      default
zroot  comment   -      local
```

주석을 기본값으로 로컬 설정하므로 값의 소스는 로컬로 유지됩니다.

풀을 만들 때 **-o**를 사용하여 풀의 속성을 설정할 수 있습니다. 풀의 루트 데이터 세트에 대한 속성은 **-O**로 설정할 수 있습니다.

```
$ zpool create -o altroot=/mnt -O canmount=off -m none zroot /dev/gpt/disk0
```

이 풀의 **altroot** 속성은 */mnt*로 설정되어 있고, 이 풀의 루트 데이터 세트는 **canmount** 속성이 ***off***로 설정되어 있습니다. 속성으로 데이터 기록 방식이 변경되면 속성 변경 후 기록된 데이터만 영향을 받습니다. ZFS는 속성 변경에 따라 기존 데이터를 다시 쓰지 않습니다.

풀 히스토리 (Pool History)

모든 zpool은 풀이 생성된 시점부터 지금까지 풀에 적용된 모든 변경 사항의 사본을 보관합니다. 이 기록에는 시스템 전원 켜기 및 끄기와 같은 일상적인 이벤트는 포함되지 않지만, 속성 설정, 풀 업그레이드, 데이터 세트 생성은 포함됩니다.

기록에 액세스하려면 **zpool history**를 실행하고 풀 이름을 입력합니다.

```
$ zpool history zroot
History for 'zroot':
2014-01-07.04:12:05 zpool create -o altroot=/mnt -O canmount=off -m none zroot mirror /
dev/gpt/disk0.nop /dev/gpt/disk1.nop
2014-01-07.04:12:50 zfs set checksum=fletcher4 zroot
```

```
2014-01-07.04:13:00 zfs set atime=off zroot
...
```

숙련된 FreeBSD 사용자라면 FreeBSD 문서와 포럼에 있는 수많은 ZFS 튜토리얼을 통해 이 기능을 알아볼 수 있을 것입니다 .

기록은 다음과 같이 끝납니다 :

```
...
2015-03-12.14:36:35 zpool set comment=Main OS files zroot
2015-03-12.14:43:45 zpool set comment=- zroot
```

comment 속성을 변경했으므로 기록에 남습니다 . 영원히요 .

안타깝게도 풀 기록에서는 누가 각 변경을 수행했는지 추적할 수 없지만 , 변경 사항을 영구적으로 기록해 두면 문제 분석에 도움이 됩니다 .

Zpool 유지 관리 자동화 (Zpool Maintenance Automation)

FreeBSD는 **periodic(8)**로 실행되는 일일 유지 관리 작업의 일부로 각 시스템의 파일 시스템을 검사합니다 . 이 검사에 ZFS 풀 정보를 추가하면 풀 상태에 대한 정보를 얻을 수 있습니다 . *periodic.conf*의 **daily_status_zfs_enable** 옵션은 풀 검사를 활성화합니다 .

```
daily_status_zfs_enable="YES"
```

이제 매일 **periodic(8)** 출력에 일반적으로 "all pools are healthy."라는 한 줄의 출력인 **zpool status -x**가 포함됩니다 .

풀에 대한 더 자세한 정보를 원할 경우 일일 보고서에 zpool 목록 출력을 포함할 수도 있습니다 . 목록을 가져오려면 **daily_status_zfs_zpool_list**를 **yes**로 설정하세요 . 특정 풀의 상태만 표시하여 해당 출력을 잘라내려면 *periodic.conf*의 **daily_status_zpool** 변수에 원하는 풀을 나열하세요 .

FreeBSD가 풀 스크럽을 수행하도록 할 수도 있습니다 . 스크러빙 옵션을 설정하면 FreeBSD는 풀에 스크러빙이 필요한지 매일 확인하지만 , 설정된 간격으로만 스크러빙을 수행합니다 . 35일마다 모든 풀을 자동으로 스크러빙하려면 , *periodic.conf*에서 **daily_scrub_zfs_enable**을 **YES**로 설정하세요 .

```
daily_scrub_zfs_enable="YES"
```

FreeBSD는 기본적으로 모든 풀을 스크러빙합니다 . 일일 스크럽 검사에서 특정 풀을 명시적으로 제외할 수는 없습니다 . 그러나 **daily_scrub_zfs_pools**에 검사할 풀을 명시적으로 나열할 수 있습니다 . 목록에 없는 풀은 스크럽되지 않습니다 .

```
daily_scrub_zfs_pools="zroot prod test"
```

스크럽 사이의 일 수를 변경하려면 **daily_scrub_zfs_default_threshold**를 원하는 일 수로 설정합니다.

```
daily_scrub_zfs_default_threshold="10"
```

특정 풀을 다른 일정에 따라 스크러빙하려면 **daily_scrub_zfs_${poolname}_threshold**를 원하는 일수로 설정하세요. 여기서는 7일마다 풀 프로드를 스크러빙합니다.

```
daily_scrub_zfs_prod_threshold="7"
```

자체 임계값이 없는 풀은 기본 임계값을 사용합니다.

풀 제거 (Removing Pools)

풀을 제거하려면 **zpool destroy** 명령과 풀 이름을 사용합니다.

```
$ zpool destroy test
```

파괴는 풀의 기본 공급자를 파괴된 풀의 일부로 표시하여 다른 풀에 재사용할 수 있도록 합니다. 디스크를 지우지는 않으며, 5장을 읽은 사람이라면 누구나 풀을 복원하고 데이터에 액세스할 수 있습니다.

공급업체의 데이터를 안전하게 삭제하거나 덮어써야 하는 경우 디스크 덮어쓰기 또는 파쇄 프로그램이 필요합니다.

Zpool 기능 플래그 (Zpool Feature Flags)

ZFS와 풀에는 원래 풀에서 지원하는 기능을 나타내는 버전 번호가 있습니다. 시스템에서 익숙하지 않은 스토리지 공급업체를 보고 "이 풀은 ZFS 버전 20이므로 중복 제거나 기본 암호화를 지원하지 않습니다."라고 말할 수 있습니다. 풀을 릴리스에서 지원하는 최신 버전으로 업그레이드할 수도 있고 그렇지 않을 수도 있습니다.

그 후 오라클은 ZFS 소스를 폐쇄했고, 다양한 사람들이 마지막 오픈 소스 ZFS 릴리스를 가져와서 직접 유지 관리하도록 했습니다. 마지막 오픈 소스 오라클 ZFS 버전은 28이었습니다. 다양한 그룹이 자체 풀 기능을 구현함에 따라 서로 다른 그룹의 버전 번호가 상호 호환되지 않게 될 위험이 있었습니다. 각기 다른 ZFS 팀이 선택한 새로운 기능을 구현할 수 있었기 때문에, 예를 들어 FooZFS 버전 30은 BarZFS 버전 30과 호환되지 않을 수 있었습니다. ZFS의 주요 목표는 상호 운용성입니다.

OpenZFS 팀은 버전 번호로 기능을 추적하는 방식에서 벗어나는 것이 가장 좋은 방법이라고 판단했습니다. 이들은 OpenZFS 버전을 최대 5000개까지 늘려서 오라클이 새로운 버전을 추가할 수 있는 충분한 공간을 확보했습니다. 다양한 플랫폼의 다양한 OpenZFS 개발자를 모두 수용하기 위해 개발자들은 버전 번호를 기능 플래그로 효과적으로 대체하기로 결정했습니다.

FreeBSD를 포함하여 OpenZFS를 실행하는 모든 플랫폼에는 이 특정 설치가 지원하는 풀 기능을 나열하는 **zpool-features(7)** 매뉴얼 페이지가 포함되어 있어야 합니다. 최신 버전의 FreeBSD는 아마도 새로운 풀 기능을 지원할 것입니다.

기능을 사용하면 일반적으로 어떤 식으로든 온디스크 형식이 변경됩니다. 예를 들어, 스냅샷 지원을 추가한다는 것은 "이것은 스냅샷입니다."라는 새 필드와 메타데이터를 추가하는 것을 의미합니다. 해당 기능을 지원하지 않는 시스템에서는 이 풀을 보고 "이런, 이 데이터 구조를 인식하지 못합니다. 난 이걸 건드리지 않을 거야!"라고 말할 것입니다. 시스템 간에 디스크를 정기적으로 교체하는 경우, 새로운 기능 플래그를 업그레이드하거나 활성화하기 전에 여러 호스트에서 지원되는 기능 플래그를 주의 깊게 확인해야 합니다.

기능 플래그 보기 (Viewing Feature Flags)

풀에서 지원하는 기능 플래그와 해당 설정을 보려면 **"feature"**이라는 단어가 포함된 풀 속성을 찾습니다.

```
$ zpool get all zroot | grep feature
zroot  feature@async_destroy  enabled  local
zroot  feature@empty_bpobj    active   local
zroot  feature@lz4_compress   active   local
...
```

활성화된 풀 기능은 사용할 수 있지만 실제로는 아직 사용되지 않습니다. 시스템이 새로운 유형의 압축을 지원할 수 있지만 실제로 새 알고리즘을 사용하여 풀에 데이터를 쓰지 않은 경우입니다. 온디스크 형식이 해당 기능을 수용하도록 변경되지 않았기 때문에 이 풀은 해당 기능을 지원하지 않는 시스템에서 가져올 수 있습니다. 새 호스트는 놀라게 하는 어떤 것도 볼 수 없습니다.

비활성화된 풀 기능은 운영 체제에서 사용할 수 있지만 활성화되어 있지 않습니다. 풀에 이러한 기능을 사용할 수 있다고 표시되어 있지 않습니다. 비활성화된 기능이 있다는 것은 운영 체제에서 해당 기능을 사용할 수 있다는 의미입니다. 이 풀은 이 기능을 지원하지 않는 호스트에서도 확실히 사용할 수 있습니다.

이 기능이 활성화되어 있으면 해당 기능이 사용 중이므로 디스크 형식이 변경된 것입니다. 대부분의 경우 이 기능을 지원하지 않는 시스템에서는 이 풀을 가져올 수 없습니다. 기능이 활성화되어 있지만 해당 기능을 사용하는 모든 데이터 세트가 삭제된 경우 풀은 기능 설정을 활성화로 되돌립니다.

일부 기능은 "read-only compatible"입니다. 해당 기능을 사용 중인 경우, 해당 기능을 지원하지 않는 시스템으로 풀을 부분적으로 가져올 수 있습니다. 새 호스트는 풀에 있는 일부 데이터 집합을 볼 수 없고 풀에 데이터를 쓸 수 없지만, 데이터 집합에서 일부 데이터를 추출할 수는 있습니다.

풀을 만들면 해당 운영 체제의 ZFS 구현에서 지원하는 모든 기능이 활성화됩니다. **zpool create**와 함께 **-d** 플래그를 사용하여 새 풀의 모든 기능을 비활성화한 다음 기능을 보다 선택적으로 활성화할 수 있습니다.

이제 풀의 작동 원리를 이해했으니 실제 데이터를 풀에 입력해 보겠습니다.

Revision #1
Created 23 January 2024 14:19:33 by 고기만두 (MeatDumpling)
Updated 23 January 2024 14:19:46 by 고기만두 (MeatDumpling)