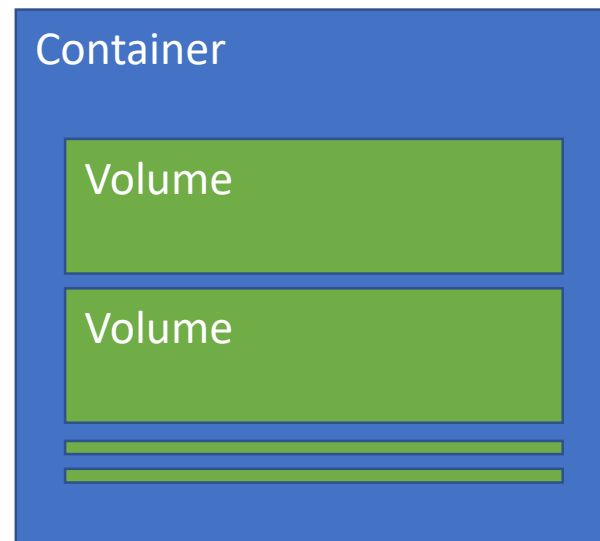


APFS 101

# ContainerとVolume

- APFSのパーティションはContainerと呼ばれ、内部に複数のVolumeを定義できる。
- 各VolumeはContainer内の空き領域を共有する。



# メタデータとobject

- APFSではメタデータ（file名、タイムスタンプ、パスやデータへの参照など）はobjectと呼ばれる。3種類のobjectが定義されており、共通ヘッダ（次頁参照）のo\_typeフィールドのObject Type Flagsの値で識別することができる。
  - Physical object (OBJ\_PHYSICAL)
    - ストレージ上に保持されるobject。更新時は新たな複製が作られ、その複製には異なるobject-id(o\_oid) が付与される。o\_oid = 物理アドレス（container superblockの先頭からの offset）となる。
  - Ephemeral object (OBJ\_EPHEMERAL)
    - メモリに保持されるobject。同時にストレージ上にも存在し、更新されるたびに上書きされる。
  - Virtual object (OBJ\_VIRTUAL)
    - ストレージ上に保持されるobject。更新時は新たな複製が作られ、同じo\_oidが付与される。Virtual objectの物理アドレスはObject map（OBJECT\_TYPE\_OMAP）から参照する（後述）。

# objectヘッダ(checksum, oid, xid)

- objectは32バイトの共通のヘッダ (obj\_phys) を持っている。
- Transaction id(o\_xid)を使ってobjectの世代管理を行う。同じo\_oidを持つobjectが複数存在する場合、xidの値が最も大きいobjectが最新の状態を示す。
- 共通ヘッダ先頭にはobjectのFletcher-64 checksum (o\_cksum) が保持されており、この検証に失敗した場合は同じo\_oidで古いo\_xidを持つobjectが利用される。
- Fletcher-64 checksumの計算については本資料P.29に記載している。
- type/sub\_typeとobject type flagの定義については資料[1]のP.13およびP.18参照。

```
struct obj_phys {  
    uint8_t      o_cksum[MAX_CKSUM_SIZE];  
    oid_t        o_oid;  
    xid_t        o_xid;  
    uint32_t     o_type;  
    uint32_t     o_subtype;  
};  
typedef struct obj_phys obj_phys_t;
```

- sample.rawの先頭にあるobjectのヘッダをバイナリエディタで開いた例

00000000	EF	D8	AA	CF	B4	DF	E0	89	01	00	00	00	00	00	00	00
00000010	16	00	00	00	00	00	00	00	01	00	00	80	00	00	00	00

```
o_chksum:      \xEF\xD8\xAA\xCF\xB4\xDF\xE0\x89  
o_oid:         0x00000000000000000001  
o_xid:         0x00000000000000000016  
o_type:        0x80000001  
o_subtype:     0x00000000
```

メタデータの参照例

Container内のVolumeを参照する

# Container内のVolumeを参照する (1)

- Volume Superblock (apfs\_superblock\_t) のmapを探す(1)
  - Container冒頭のContainer Superblock (nx\_superblock\_t) のnx\_omap\_oidフィールドから参照されているomap\_phys\_t (Containerのobject map) をたどる。Containerのobject mapのom\_tree\_oidフィールドの参照先に、OMAPのBTREE (btree\_node\_phys\_t) が配置されている。
  - sample.rawではブロック0x17ECにomap\_phys\_tが、0x17EDにbtree\_node\_phys\_t があることがわかる。
  - nx\_superblock\_tの定義は資料[1]のP.25、omap\_phys\_tの定義はP.41参照。

## Object Type: NX\_SUPERBLOCK

```
nx_superblock_t
...
0x20 u32 nx_magic
0x24 u32 nx_block_size
...
0xA0 u64 nx_omap_oid
...
```

## Object Type: OMAP

```
omap_phys_t
...
0x30 u64 om_tree_oid
...
```

## Object Type: BTREE

### Object Sub Type: OMAP

```
btree_node_phys_t
```

00000000	EF D8 AA CF	B4 DF E0 89	01 00 00 00	00 00 00 00
00000010	16 00 00 00	00 00 00 00	01 00 00 80	00 00 00 00
00000020	4E 58 53 42	00 10 00 00	98 1C 00 00	00 00 00 00
00000030	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000040	02 nx_magic	00 nx_block_size	AD 55 67 E2 46 D2	
00000050	9B 4B 49 E1	74 DF DE DA	09 04 00 00	00 00 00 00
00000060	17 00 00 00	00 00 00 00	08 00 00 00	7C 00 00 00
00000070	01 00 00 00	00 00 00 00	09 00 00 00	00 00 00 00
00000080	04 nx_omap_oid	00 00 00 00	02 00 00 00	02 00 00 00
00000090	52 00 00 00	04 00 00 00	00 04 00 00	00 00 00 00
000000A0	EC 17 00 00	00 00 00 00	01 04 00 00	00 00 00 00
000000B0	00 00 00 00	01 00 00 00	02 04 00 00	00 00 00 00

17EC000	C6 20	om_tree_oid	E 40
17EC010	16 00	0 00	
17EC020	01 00 00 00	00 00 00 00	
17EC030	ED 17 00 00	00 00 00 00	

# Container Superblockのバックアップを探す (1)

- 前頁ではContainer冒頭のContainer Superblockを参照したが、APFSでは最新を含む複数の世代のContainer Superblockの複製がcheckpoint descriptor areaに保存されているので、必要に応じてそれらを探し、利用することができる。
- checkpoint descriptor areaを探すためには、Container先頭のContainer Superblockのnx\_xp\_desc\_baseフィールドと nx\_xp\_desc\_blocks フィールドを参照する。
- sample.rawではcheckpoint descriptor areaがブロック1(nx\_xp\_desc\_base)から始まり、全部で8(nx\_xp\_desc\_blocks)ブロック存在することがわかる。このブロック1というのはブロックアドレスのことで、ここではblock sizeが0x1000なので、物理アドレス（Container先頭からのオフセット）0x1000を示す。

```
nx_superblock_t
```

```
...
```

```
0x20 u32 nx_magic
```

```
0x24 u32 nx_block_size
```

```
...
```

```
0x68 u32 nx_xp_desc_blocks
```

```
...
```

```
0x70 u64 nx_xp_desc_base
```

```
...
```

00000000	EF	nx_magic	4	nx_block_size	00	00	00	00	00	00	00	00
00000010	16	00	00	00	00	80	00	00	00	00	00	00
00000020	4E	58	53	42	00	10	00	00	98	1C	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00
00000040	02	00	00	00	00	00	00	00	nx_xp_desc_blocks	D2	00	00
00000050	9B	4B	49	E1	74	DF	DE	DA	00	00	00	00
00000060	17	00	00	00	00	00	00	00	08	00	00	00
00000070	01	00	00	00	00	00	00	00	09	00	00	00
00000080	00	00	00	00	00	00	00	00	02	00	00	00
00000090	nx_xp_desc_base	00	00	00	00	04	00	00	00	00	00	00
000000A0	EC	17	00	00	00	00	00	00	01	04	00	00



# Container Superblockのバックアップを探す (2)

- checkpoint descriptor areaにはcheckpoint\_map\_phys\_tまたはnx\_superblock\_tが保存される。ここにあるnx\_superblock\_tが最新および過去のContainer Superblockの複製。このnx\_superblock\_tのうち、最大のtransaction id(o\_xid)を持つnx\_superblock\_tが最新のContainer Superblockの複製。
- sample.rawでは0x1000から8ブロックがcheckpoint descriptor areaなので、8ブロックそれぞれのヘッダを調べ、nx\_superblock\_tかどうか確認する (nx\_magicまたはOBJECT\_TYPEを確認)。
- sample.rawでは、4つのnx\_superblock\_tが見つかる (0x2000, 0x4000, 0x6000, 0x8000)。このうちo\_xidの値が一番大きいのは0x4000のnx\_superblock\_tなので、これが最新のContainer Superblockだとわかる。これはContainer先頭のnx\_superblock\_tのコピーでもある。

0001000	53 92 C1 3E	EC 1B 3E 81	01 00 00 00	00 00 00 00	o_type	00
0001010	15 00 00 00	00 00 00 00	0C 00 00 40	00 00 00 00		
0001020	01 00 00 00	04 00 00 00	05 00 00 80	00 00 00 00		

(Type: OBJECT\_TYPE\_CHECKPOINT\_MAP)

0002000	0F 4E AB CF	36 6A E0 89	01 00 00 00	00 00 00 00	o_type	00 00
0002010	15 00 00 00	00 00 00 00	01 00 00 80	00 00 00 00		
0002020	4E 58 53 42	00 10 00 00	1C 00 00 00	00 00 00 00	nx_magic	

o\_xid nx\_superblock\_t、xid: 0x15

0004000	0F 4E AB CF	6F CA F0 89	01 00 00 00	00 00 00 00	o_type	00 00
0004010	16 00 00 00	00 00 00 00	01 00 00 80	00 00 00 00		
0004020	4E 58 53 42	00 10 00 00	1C 00 00 00	00 00 00 00	nx_magic	

o\_xid nx\_superblock\_t、xid: 0x16

# Container Superblockのバックアップを探す (3)

- 本資料では、Container Superblockの「バックアップを探す手段」としてcheckpoint descriptor areaの探索を紹介した。しかし、資料[1]には、Container先頭のContainer Superblockではなく、checkpoint descriptor areaにある最新世代のContainer Superblockを利用するよう記載されている（Container先頭のContainer Superblockは最新世代のものとは保障されていない）。
- ただし、Appleの実装（macOSのdisk utility）や、代表的なAPFSマウント用ツールであるapfs-fuseでは、Container先頭に配置されているContainer Superblockに矛盾などが無い場合、そのContainer先頭のContainer Superblockが最新のContainer Superblockとして使用される。
  - 例えば、Container先頭Container Superblockの一部（nx\_xp\_desc\_base、nx\_xp\_desc\_blocks以外のフィールド）が破損している場合、checkpoint descriptor areaにある最新のnx\_superblock\_tが正常であっても対象のイメージをマウントできなくなる。
- 現時点ではAPFSパーティションマウント時の動作を再現するにはContainer先頭のContainer Superblockを利用するのが適していると考えられる。

# Container内のVolumeを参照する (2)

- Volume Superblock (apfs\_superblock\_t) のmapを探す(2)
  - BTREE (btree\_node\_phys\_t) のB-Tree構造にはContainer内に存在するVolumeを示すVolume Superblock (apfs\_superblock\_t) が列挙されている。
  - まず、B-Treeを解析する必要がある。
  - B-Treeについては資料[1]のP.107、 btree\_node\_phys\_tについてはP.108に記載されているが、次頁以降で簡単に解説する。

# B-Treeの構造 (1) ROOT NODE

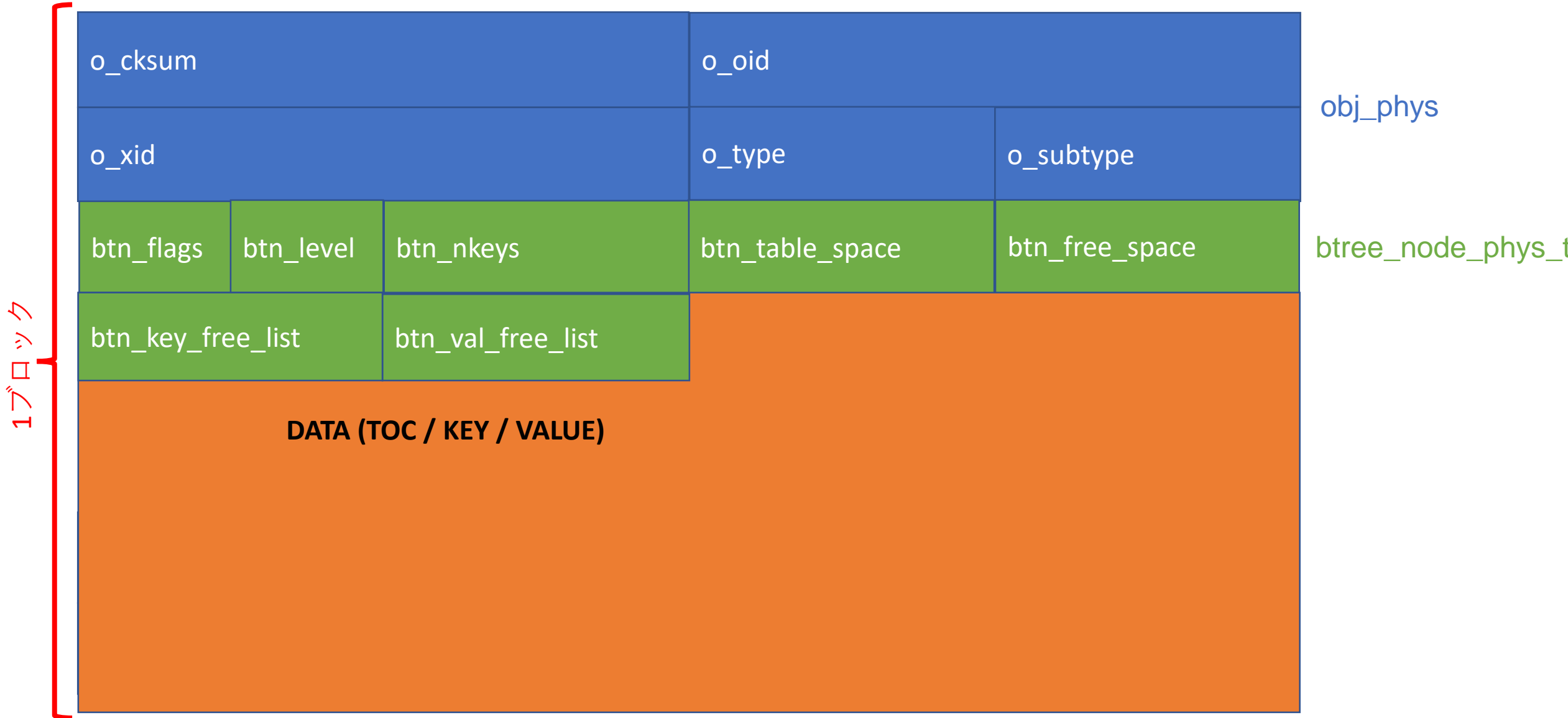
btn\_flagsにBTNODE\_ROOT(0x0001)  
が設定されている

1ブロック

o_cksum			o_oid		obj_phys
o_xid			o_type	o_subtype	
btn_flags	btn_level	btn_nkeys	btn_table_space	btn_free_space	btree_node_phys_t
btn_key_free_list		btn_val_free_list	DATA (TOC / KEY / VALUE)		
			bt_flags	bt_node_size	btree_info_t
bt_key_size		bt_val_size	bt_longest_key	bt_longest_val	
bt_key_count			bt_node_count		

# B-Treeの構造 (2) LEAF NODE

btn\_flagsにBTNODE\_ROOT(0x0001)  
が設定されていない



# B-Treeの構造 (3) TOC/KEY/VALUE (1)

- `btree_node_phys_t`の`btn_data[]`はTOC（Table of Contents）、KEY、VALUEという3種類のデータを含んでいる。
- TOCは`btn_data[]`の先頭から始まる。KEYはTOCの末尾から始まる。
- 各KEY/VALUEの配置はTOCに保持されている（後述）。TOCでは各KEYはKEYフィールド先頭からのオフセット、各VALUEはVALUEフィールド末尾からオフセットで表現される。
- VALUEは`btn_data[]`の末尾に配置される。`btree_node_phys_t`の`flags`に`BTNODE_ROOT`が設定されている場合は、`btree_node_phys_t`で始まるblockの末尾-0x28バイトが`btn_data`の末尾となる。`BTNODE_ROOT`が設定されていない場合は、`btree_node_phys_t`で始まるblockの末尾が`btn_data`の末尾となる。



# B-Treeの構造 (4) TOC/KEY/VALUE (2)

- TOC、KEY、VALUEフィールドの配置はbtn\_table\_spaceの値から計算する。
- 図のsample.rawの例では以下のようになる。
  - TOCのオフセット(btn\_table\_space.off) : 0x0、TOCの長さ(btn\_table\_space.len) : 0x1C0
  - TOCはbtree\_node\_physの末尾+上記オフセットから始まるので、物理アドレスは以下のように計算できる。
    - ブロック先頭 + sizeof(btree\_node\_phys) + btn\_table\_space.off = 0x17ED000 + 0x38 + 0x0 = 0x17ED038
  - btn\_nkeysが1なので、KEYの数は1つ (=VALUEも1つ)。KEYフィールドはTOCの末尾から始まるので、物理アドレスは以下の様になる。
    - TOCの先頭 + TOCの長さ(btn\_table\_space.len) = 0x17E038 + 0x1C0 = 0x17ED1F8
  - VALUEフィールドbtn\_data[]の末尾から（先頭に向かって）始まるので、フィールド先頭は以下の様になる。
    - ブロック先頭 + ブロック長 - sizeof(btree\_info\_t) = 0x17ED000 + 0x1000 - 0x28 = 0x17EDFD8
  - なお、B-TreeがBTNODE\_ROOTでない場合、VALUEフォールドはブロック末尾（ブロック先頭 + ブロック長）から始まる。

btn\_flags:

- BTNODE\_ROOT(1)
- BTNODE\_LEAF(2)
- BTNODE\_FIXED\_KV\_SIZE(4)

17ED000	17	65	btn_nkeys				1E	45	btn_table_space.off				0	btn_table_space.len			
17ED010	16	00	00	00	00	00	00	00	00	02	00	00	40	00	00	00	00
17ED020	07	00	00	00	00	01	00	00	00	00	00	C0	01	20	00	A0	0D
17ED030	10	00	10	00	20	00	10	00	00	00	10	00	00	00	10	00	00

# B-Treeの構造 (5) TOC/KEY/VALUE (3)

- btree\_node\_phys\_tのflagsにBTNODE\_FIXED\_KV\_SIZEが設定されている場合、TOCはKEY、VALUEのオフセットの組み合わせ（KEYのオフセット(u16)、VALUEのオフセット(u16)）を保持する。このとき、KEY、VALUEとも長さは0x10バイトに固定される。
- BTNODE\_FIXED\_KV\_SIZEが設定されていない場合、TOCはKEY、VALUEのオフセットと長さを保持する（KEYのオフセット(u16)、KEYの長さ(u16)、VALUEのオフセット(u16)、VALUEの長さ(u16)）。
- 図のsample.rawの0x17ED000のbtree\_node\_phys\_tではBTNODE\_FIXED\_KV\_SIZEが設定されているので以下ようになる。

1つ目のKEYのオフセット

17ED020	07 00 00 00	01 00 00 00	00 00 C0 01	20 00 A0 0D
17ED030	10 00 10 00	20 00 10 00	00 00 10 00	00 00 00 00
17ED040	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

1つ目のVALUEのオフセット

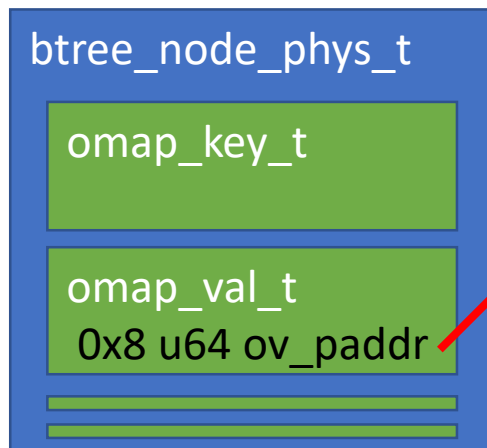
- 1番目のKEYはKEYフィールドの先頭から始まる。長さは0x10。 0x17ED1F8 ~
- 1番目のVALUEはVALUEの末尾-0x10バイトから始まる。長さは0x10。 0x17EDFC8 ~
- KEYのオフセットはKEYフィールドの先頭からのオフセット、VALUEのオフセットはVALUEフィールドの末尾からのオフセット。複数KEYがある場合、例えば2番目のKEYの方が、1番目のKEYよりKEYフィールドの先頭側に配置されている場合などもある。



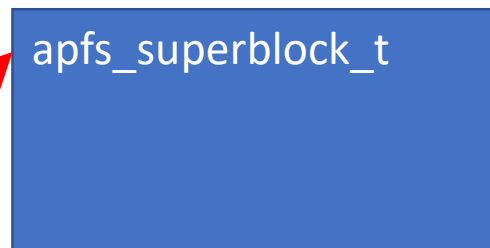
# Container内のVolumeを参照する (3)

- Volume Superblock (apfs\_superblock\_t) のmapを探す(3)
  - sample.rawにはvolumeが1つしかないので、OMAPのBTREEにはomap\_key\_tとomap\_val\_tの組み合わせが1つ保持されている。
  - omap\_key\_tとomap\_val\_tの定義は資料[1]のP.43参照。

Object Type: BTREE  
Object Sub Type: OMAP



Object Type: FS



omap_key_t	17ED1F0	00	ok_xid				ok_oid			
	17ED200	16	00	00	00	00	02	04	00	00
omap_val_t	17EDFC0	E8	17	00	00	00	ov_flags			
	17EDFD0	EB	17	00	00	00	00	00	00	00
							ov_size			
							00	10	00	00
							ov_paddr			
							12	00	00	00
							00	10	00	00

つまり、対象のVolume Superblockは

- object id: 0x402
- transaction id: 0x16
- size: 0x1000
- physical address block: 0x17EB -> **0x17EB000**

Volume内のfileを参照する

# Volume内のfileを参照する (1)

- Container内のVolumeを探した時と同様に、Volume Superblock(apfs\_superblock\_t) のapfs\_omap\_oid フィールドからOMAP (omap\_phys\_t) をたどってB-Tree (subtype:OMAP) を探す。
- Volume Superblockの定義は資料[1]のP.48参照。

Object Type: FS

apfs\_superblock\_t

...

0x20 u32 apfs\_magic

...

0x80 u64 apfs\_omap\_oid

...

Object Type: OMAP

omap\_phys\_t

0x30 u64 om\_tree\_oid

Object Type: BTREE

Object Sub Type: OMAP

btree\_node\_phys\_t

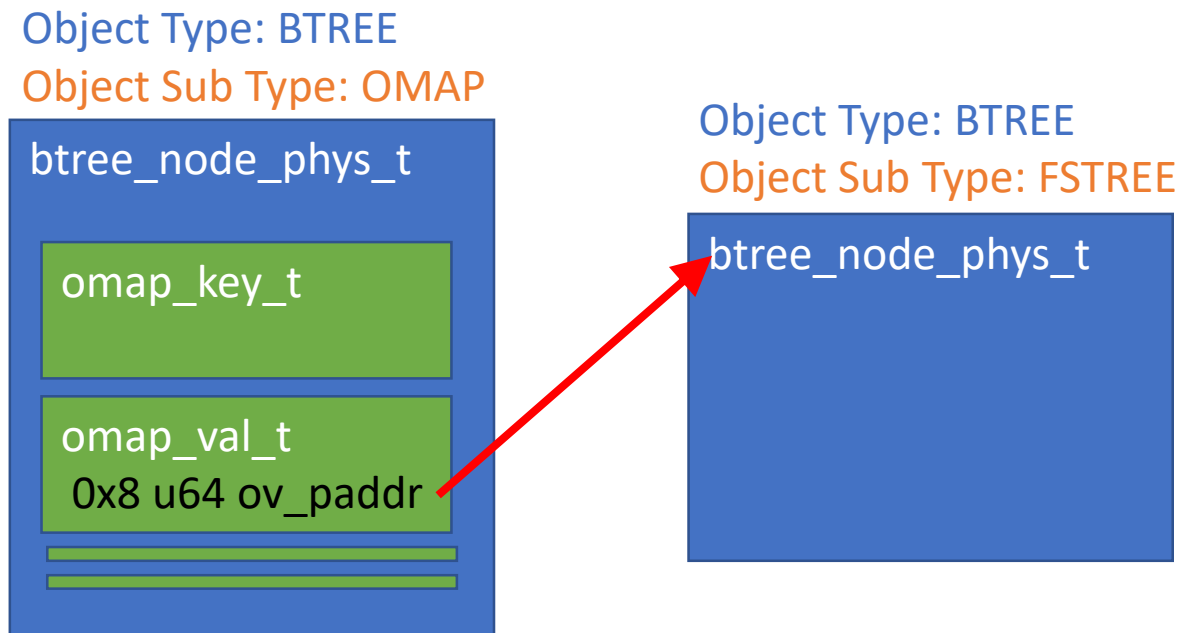
17EB000	54 42	apfs_magic	C 13 9C	02 04 00 00	00 00 00 00
17EB010	16 00		00 00 00	0D 00 00 00	00 00 00 00
17EB020	41 50 53 42		00 00 00 00	00 00 00 00	00 00 00 00
17EB080	E6 17 00 00		00 00 00 00	apfs_omap_oid	00 00 00

17E6000	74 58 41 FF	99 77 BE 40
17E6010	15	om_tree_oid 0 00 00 00
17E6020	00	00 00 00 00
17E6030	E7 17 00 00	00 00 00 00

sample.rawの例では、ブロック0x17E7 (=0x17E7000バイト) にB-Tree (subtype:OMAP) が配置されていることがわかる。

# Volume内のfileを参照する (2)

- B-Tree (subtype:OMAP) にはfileやdirectory情報のB-Tree (subtype: FSTREE) への参照が保持されている。
- Volumeに複数のFSTREEが存在する場合は複数のomap\_key\_t/omap\_key\_val\_tの組み合わせが保持されている。



# Volume内のfileを参照する (3)

- sample.rawのケースでは、4つのFSTREEが見つかる。

17E7020	07 00 00 00	04 00 00 00	00 00 C0 01	50 00 40 0D
17E7030	30 00 10 00	40 00 10 00	20 00 30 00	10 00 20 00
17E7040	40 00 50 00	00 00 10 00	00 00 00 00	00 00 00 00

3 ← 4 ← TOC 1 2

btn\_nkeys (keyの数)

17E71F0	00 00 00 00	00 00 00 00	08 04 00 00	00 00 00 00
17E7200	14 00 00 00	00 00 00 00	06 04 00 00	00 00 00 00
17E7210	15 00 00 00	00 00 00 00	04 04 00 00	00 00 00 00
17E7220	14 00 00 00	00 00 00 00	FF FF 10 00	00 00 00 00
17E7230	14 00 00 00	00 00 00 00	07 04 00 00	00 00 00 00
17E7240	14 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

KEY

17E7F80	00 00 00 00	00 00 00 00	00 00 00 00	00 10 00 00
17E7F90	AE 00 00 00	00 00 00 00	FF FF 10 00	00 10 00 00
17E7FA0	B0 00 00 00	00 00 00 00	00 00 00 00	00 10 00 00
17E7FB0	AF 00 00 00	00 00 00 00	00 00 00 00	00 10 00 00
17E7FC0	E5 17 00 00	00 00 00 00	00 00 00 00	00 10 00 00
17E7FD0	AB 00 00 00	00 00 00 00	00 00 00 00	00 10 00 00

VALUE

1番目のKEY(omap\_key\_t):

- ok\_oid: 0x404

- ok\_xid: 0x14

1番目のVALUE(omap\_val\_t):

- ov\_flags: 0x0

- ov\_size: 0x1000

- ov\_paddr: 0xAF

3番目のKEY(omap\_key\_t):

ok\_oid: 0x407

ok\_xid: 0x14

3番目のVALUE(omap\_val\_t):

ov\_flags: 0x0

ov\_size: 0x1000

ov\_paddr: 0xAE

2番目のKEY(omap\_key\_t):

- ok\_oid: 0x406

- ok\_xid: 0x15

2番目のVALUE(omap\_val\_t):

- ov\_flags: 0x0

- ov\_size: 0x1000

- ov\_paddr: 0x17E5

4番目のKEY(omap\_key\_t):

ok\_oid: 0x408

ok\_xid: 0x14

4番目のVALUE(omap\_val\_t):

ov\_flags: 0x0

ov\_size: 0x1000

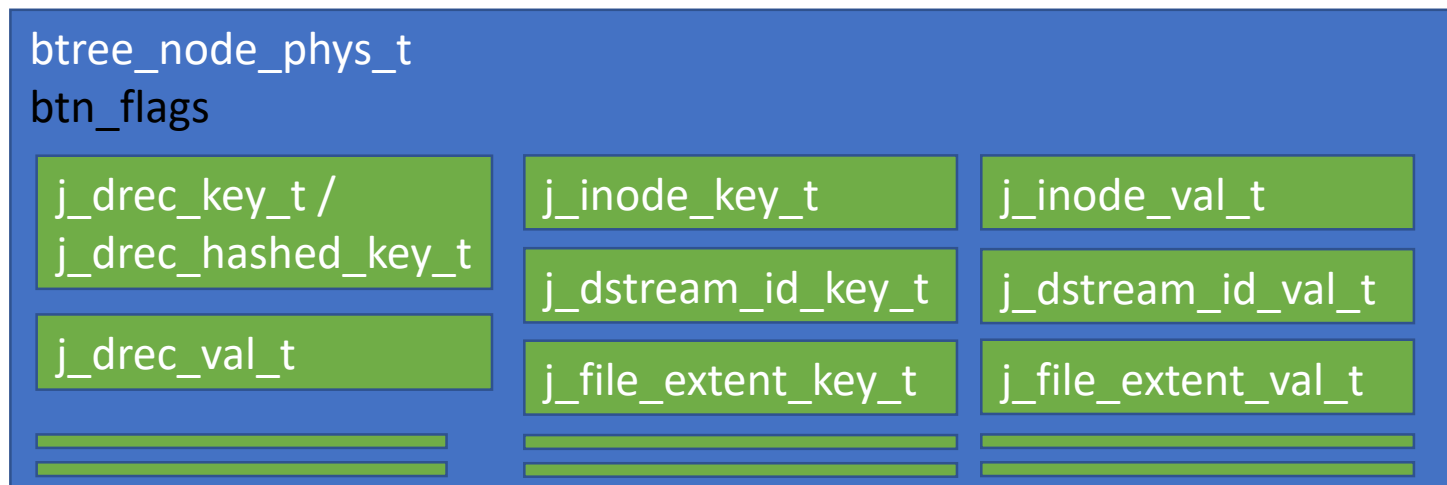
ov\_paddr: 0xAB

# File System Object (1)

- FSTREE (btree\_node\_phys\_t) には、fileやdirectoryに関する情報がkey/valueの形で保持されている。
- ボリューム内のfileやdirectoryの情報を得るためには、あらかじめすべてのFSTREEを解析してB-Treeに含まれる情報を保持しておく必要がある。

Object Type: BTREE

Object Sub Type: FSTREE (root node / leaf node)



# File System Object (2)

- KEYはj\_key\_tヘッダを持っており、そのなかのOBJ\_TYPEの値により、key/valueの種類が識別できる。
- j\_key\_tの定義は資料[1]のP.64、OBJ\_TYPE (j\_obj\_types) の定義はP.75参照。



# File System Object (3)

- 例として、sample.rawの3つ目のFSTREE（paddr: 0xAEのbtree\_node\_phys\_tから始まるB-Tree）のKEYを取り上げる。まず、このFSTREEを解析してKEYのオフセットを特定する必要がある。本資料P.14から解説しているB-Tree解析と同じ手順で解析できる。ただし、このFSTREEはBTNODE\_FIXED\_KV\_SIZEが定義されていないため可変長なので、TOCの読み方に注意する必要がある。また、BTNODE\_ROOTが定義されていないためLEAF NODEなので、btree\_info\_tがない（ブロック末尾がbtn\_data[]の末尾となる）。
- sample.rawの3つ目のFSTREE（paddr: 0xAEのbtree\_node\_phys\_tから始まるB-Tree）の1番目のKEYは、j\_key\_tに0x9000000000000001が格納されているので、TYPE->9 (APFS\_TYPE\_DIR\_REC)、ID->1 となる。

1A8A1F0	00	00	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00	90	.....
1A8A200	0C	8C	A6	AC	70	72	69	76	61	74	65	2D	64	69	72	00			....private-dir.

- TYPEから、このKEYはj\_drec\_key\_tまたはj\_drec\_hashed\_key\_tであることがわかる。なお、これはj\_key\_t末尾から4バイトのhashが格納されているのでj\_drec\_hashed\_key\_t。
- j\_drec\_hashed\_key/j\_drec\_valueは対応するdirectory配下のfileの情報（IDやdirectoryへの追加日時など）を格納している。ここではid: 1のdirectory（最上位のdirectory。"root"というdirectoryとは異なるので注意）配下の"private-dir"というdirectoryに関する情報が含まれている。



# File System Object (4)

- APFSは以下のようなdirectory構成をとる。

    / （最上位のdirectory）

    └ private-dir

    └ root

        └ （ユーザが配置するdirectoryやfile）

- 資料[1]では上記の"root"がマウントポイントとされており、実際にmacOSのDiskUtilityでマウントすると、"root"がマウントポイントとなる。ただし、apfs-fuseでマウントした場合は、最上位のdirectory（上記の"/"）がマウントポイントとなるので注意が必要。

# Volume内のfileを参照する (4)

- 例えばsample.raw内にある"iir\_vol40.pdf"という名称のfileは、ID:16のdirectory配下にあり、idが13であることがわかる。
- j\_drec\_hashded\_key\_t、j\_drec\_val\_tの定義は資料[1]のP.70、P.71参照。

j\_drec\_hashded\_key\_t

00AE580	6F 72 65 00	16 00 00 00	00 00 00 90	0E 2C 84 2E	ore.....,..
00AE590	69 69 72 5F	76 6F 6C 34	30 2E 70 64	66 00 16 00	iir_vol40.pdf...

j\_key\_t

name\_len\_and\_hash

name[]

j\_drec\_val\_t

00AE880	00 00 00 00	93 64 0E DC	57 9D 91 15	08 00 13 00
00AE890	00 00 00 00	00 00 D8 CB	E3 DB 57 9D	91 15 08 00

file\_id

flags

xfield[]

# Volume内のfileを参照する (5)

- 例えばsample.raw内にある"iir\_vol40.pdf"という名称のfile (ID:13) に関するタイムスタンプなどの情報はID:13のj\_inode\_val\_tに格納されている。
- j\_inode\_key\_t、j\_inode\_val\_tの定義は資料[1]のP.65、P.66参照。

j\_inode\_key\_t

00AE460	00	13	00	00	00	00	00	00	00	30	13	00	00	00	00	00	00
---------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

j\_key\_t -> OBJ\_TYPE: 3 (APFS\_TYPE\_INODE)

j\_inode\_val\_t

00AEB10	00	00	00	5F	99	11	00	00	00	00	00	16	00	00	00	00	00	...
00AEB20	00	00	13	00	00	00	00	00	00	00	00	00	44	D7	C2	FF	6D	...
00AEB30	58	15	00	44	D7	C2	FF	6D	58	15	78	F1	E3	DB	57	9D	...	
00AEB40	91	15	0F	F7	2A	57	54	9D	91	15	00	80	00	00	00	00	...	
00AEB50	00	00	01	00	00	00	00	00	00	00	00	00	00	00	00	00	...	
00AEB60	00	00	63	00	00	00	63	00	00	00	A4	01	00	00	00	00	..C...C.....	
00AEB70	00	00	00	00	00	00	02	00	00	00	00	00	00	00	08	20	.....8.....	
00AEB80	28	00	69	69	72	5F	76	6F	00	00	00	00	00	00	4	66	00	(.iir_vol40.pdf.
00AEB90	00	00	F1	6B	9F	00	00	00	00	00	00	00	70	9F	00	00	00	...k.....p...
00AEBA0	00	00	00	00	00	00	00	00	00	00	F1	6B	9F	00	00	00	00	.....k....
00AEBB0	00	00	F1	6B	9F	00	00	00	00	00	00	20	00	00	00	00	00	...k.....

private\_id (ID)

create\_time (u64)

change\_time (u64)

mod\_time (u64)

access\_time (u64)

# Volume内のfileを参照する (6)

- 例えばsample.raw内にある"iir\_vol40.pdf"という名称のfile (ID:13) のコンテンツデータへの参照はID:13のj\_file\_extent\_val\_tに格納されている。
- j\_file\_extent\_key\_t、j\_file\_extent\_val\_tの定義は資料[1]のP.91、P.92参照。

j\_file\_extent\_key\_t

00AE490	00	00	00	00	60	13	00	00	00	00	00	00	80	00	00	00
00AE4A0	00	00	00	00	00	14	00	00	00	00	00	00	30	14	00	00

j\_key\_t -> OBJ\_TYPE: 8 (APFS\_TYPE\_FILE\_EXTENT)

logical\_addr(u64)

ファイルがフラグメント化していると、1つのファイルに対してフラグメントの数だけj\_file\_extentが作成される。logical\_addrは、そのj\_file\_extent (=フラグメント) の、ファイル中でのオフセットを示す。

j\_file\_extent\_val\_t

00AEAE0	00	00	65	3F	81	00	00	00	00	00	00	00	70	9F	00	00	00
00AEAF0	00	00	B4	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00AEB00	00	00	01	00	00	00	02	00	10	00	4F	D6					

len\_and\_flags (u64/上位1バイトがflags)

flag: 0x0、len: 0x9F700

phys\_block\_num: 0xB4 -> 0xB4000

00B4000	25	50	44	46	2D	31	2E	36	0D	25	E2	E3	CF	D3	0D	0A	%PDF-1.6.%.....
00B4010	31	33	33	38	20	30	20	6F	62	6A	0D	3C	3C	2F	46	69	1338 0 obj.<</Fi
00B4020	6C	74	65	72	2F	46	6C	61	74	65	44	65	63	6F	64	65	lter/FlateDecode
00B4030	2F	46	69	72	73	74	20	32	35	36	2F	4C	65	6E	67	74	/First 256/Lengt
00B4040	68	20	31	32	33	38	2F	4E	20	32	39	2F	54	79	70	65	h 1238/N 29/Type

Fletcher-64 checksumの計算

# Fletcher-64 checksumの計算

- objectを変更した場合、object先頭のFletcher-64(o-cksum)を再計算する必要がある（checksumが検証できないobjectは利用されない）。
- 下記URLからダウンロードできるfletcherNbit.pyを使って計算することができる。
  - <https://github.com/njaladan/hashpy>
- 例えばsample.rawの0x2000から始まるblockのchecksumは左下のサンプルコード(f64\_calc.py)を使う。

```
import fletcherNbit

target_image="path/to/sample.raw"
target_object=0x2000
object_length=0x1000

f = open(target_image, "rb")
f.seek(target_object+8)
data_for_checksum = f.read(object_length-8)
checksum = fletcherNbit.Fletcher64()
checksum.update(data_for_checksum)
print(checksum.cb_hexdigest())
```

f64\_calc.py

左のサンプルコードを実行すると、以下の様な結果が得られる。  
この結果の上位4バイト、下位4バイトをそれぞれリトルエンディアンで結合すると、APFSで使われているFletcher-64 Checksumが得られる。

```
> python f64_calc.py
0xcfab4e8f39e06a36
```

0002000	8F 4E AB CF	36 6A E0 89
---------	-------------	-------------

apfs-fuse

# apfs-fuseのコンパイル (1)

- APFSマウント用ツール。以下のURLからダウンロードしてコンパイルして使う。
  - <https://github.com/sgan81/apfs-fuse>
- 以下で、SIFT (<https://digital-forensics.sans.org/community/downloads>) 環境でのコンパイル手順を紹介する。
  - aptコマンドで依存ライブラリをインストールする（gitに記載されているものと一部異なるので注意）。

```
> sudo apt update  
> sudo apt install fuse libfuse-dev bzip2 libbz2-dev cmake git libattr1-dev cmake-curses-gui
```

- Gitからダウンロードしてsubmoduleを更新する。

```
> git clone https://github.com/sgan81/apfs-fuse.git  
> cd apfs-fuse  
> git submodule init  
> git submodule update
```

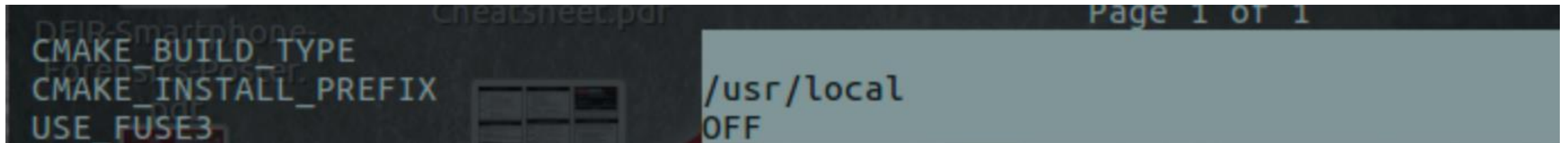


# apfs-fuseのコンパイル (2)

- Buildフォルダを作り、buildする。

```
> mkdir build
> cd build
> cmake ..
> ccmake . # <- FUSE3をOFFにする
> make
```

- なお、ccmakeでは以下のような画面になる。矢印キーでUSE\_FUSE3の行に移動し、ENTERキーを押下してONをOFFに変更する。その後cキー、gキーの順に押下するとconfigが更新されてccmakeが終了する。



# apfs-fuseの使い方

- sample.rawイメージは以下のコマンドでマウントできる。

```
# apfs-fuse -s 0 <image file> <mount dir>
```

- アンマウント時は次のコマンドを使う。

```
# fusermount -u <mount dir>
```

# apfs-dump

- apfs-fuseに含まれるapfs-dumpコマンドを使うと、イメージに含まれるすべてのobjectをdumpすることができる。

```
> apfs-dump <image file> <output.txt>
```

- apfsイメージの解析には極めて便利。

[paddr]	cksum	oid	xid	type	subtype	descript
0000000000000000	EFD8AACFB4DFE089	0000000000000001	0000000000000016	80000001	00000000	Contain
0020	u32 magic	: 4253584E				
0024	u32 block_size	: 00001000				
0028	u64 block_count	: 00000000000001C98				
0030	u64 features	: 0000000000000000	☐			
0038	u64 ro_compat_feat's	: 0000000000000000	☐			
0040	u64 incompat_feat's	: 0000000000000002	[NX_INCOMPAT_VERSION2]			

関連資料

# [1] Apple File System Reference

- <https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf>
- 構造体の定義だけでなく、以下の各章の冒頭の説明は有用。
  - P.9 (Objects)
  - P.24 (Container)
  - P.41 (Object Maps)
  - P.64 (File-System Objects)
  - P.107 (B-Trees)

## [2] APFS Internals

- [https://objectivebythesea.com/v1/talks/OBTS\\_v1\\_Levin.pdf](https://objectivebythesea.com/v1/talks/OBTS_v1_Levin.pdf)
- OS X Internalsの著者がAPFSの概要をまとめた資料。ポイントが簡潔にまとめられている。

# [3] Decoding the APFS file system

- [https://cyberforensicator.com/wp-content/uploads/2017/11/DIIN\\_698\\_Revisedproof.1-min-ilovepdf-compressed.pdf](https://cyberforensicator.com/wp-content/uploads/2017/11/DIIN_698_Revisedproof.1-min-ilovepdf-compressed.pdf)
- 資料[2]の次に読み易い解説。ただし**Apple**公式資料の公開前に書かれたため、用語が異なっている。以下に一部の対応表を示す。

当該論文	公式
Volume Checkpoint Superblock	OBJECT_TYPE_FS
Volume Block	OBJECT_TYPE_BTREE - OBJECT_TYPE_OMAP
Table	OBJECT_TYPE_BTREE
Node ID#	Transaction id
B-Tree Object Map	OBJECT_TYPE_BTREE - OBJECT_TYPE_OMAP
B-Tree Root Node	OBJECT_TYPE_BTREE - OBJECT_TYPE_FSTREE (root node)
B-Tree Leaf Node	OBJECT_TYPE_BTREE - OBJECT_TYPE_FSTREE (leaf node)

# その他

- [4] Apple File System (APFS)
  - [https://github.com/libyal/libfsapfs/blob/master/documentation/Apple%20File%20System%20\(APFS\).asciidoc](https://github.com/libyal/libfsapfs/blob/master/documentation/Apple%20File%20System%20(APFS).asciidoc)
  - 簡潔で構造体の定義が追い易い。
- [5] APFS filesystem format
  - <https://blog.cugu.eu/post/apfs/>
  - B-Treeの構造が比較的わかりやすく図示されている。
- [6] njaladan/hashpy
  - <https://github.com/njaladan/hashpy>
  - 各種hashライブラリのpython実装。Fletcherも含まれている。
- [7] Mounting an APFS image in Linux
  - <http://az4n6.blogspot.com/2018/01/mounting-apfs-image-in-linux.html>
  - apfs-fuseの使い方ガイド。