

# ブロックチェーン公開講座 第3回

## ビットコインその2

---

芝野恭平

東京大学大学院工学系研究科技術経営戦略学専攻

ブロックチェーンイノベーション寄付講座

特任研究員

[shibano@tmi.t.u-tokyo.ac.jp](mailto:shibano@tmi.t.u-tokyo.ac.jp)



# トランザクション

---



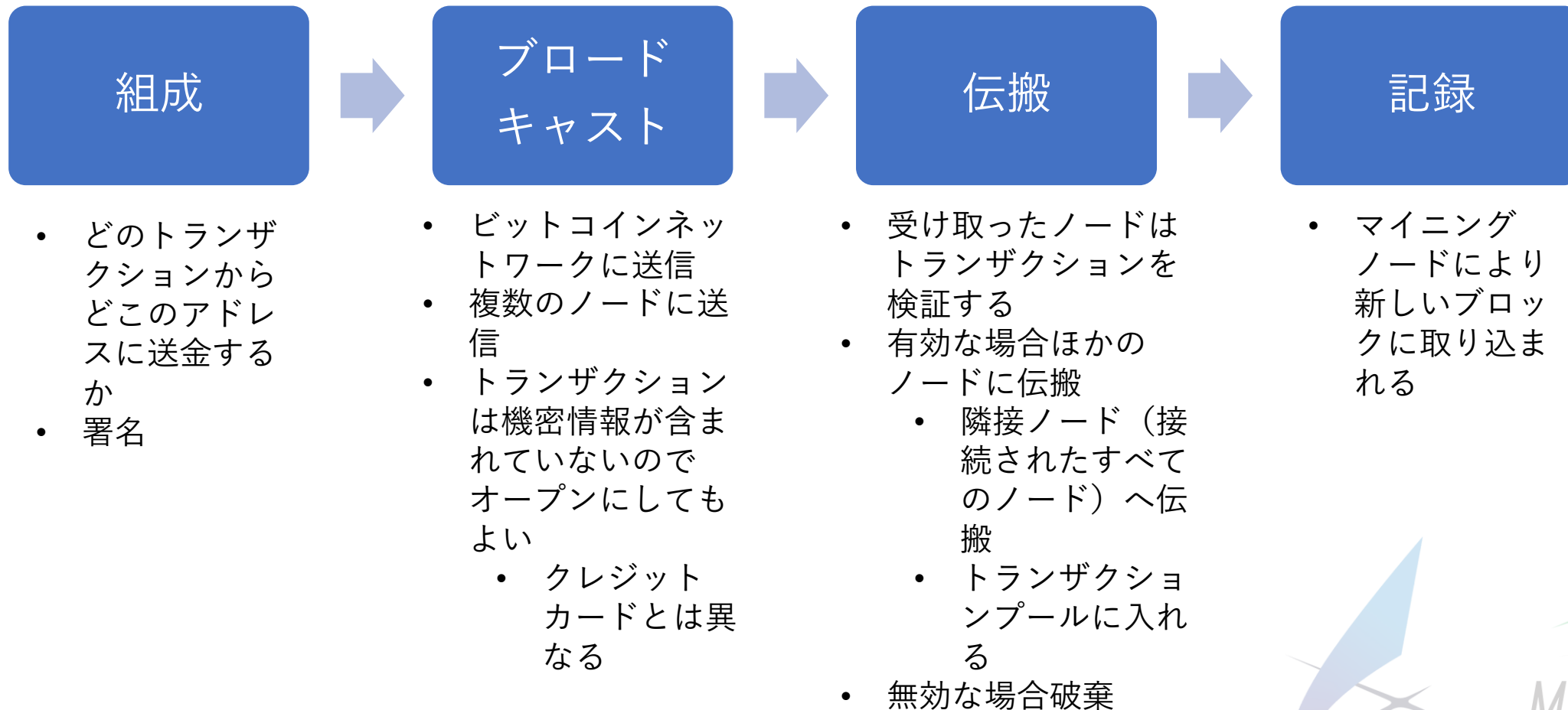
## 概要

---

- UTXOモデル
- トランザクションデータ
  - トランザクションID
  - インプット, アウトプット
  - 手数料
- 鍵の使い方
  - ロッキングスクリプト, アンロックスクリプト
- 様々なトランザクション5種類
  - Pay-to-Public-Key-Hash (P2PKH)
  - Pay-to-Public-Key
  - データアウトプット (OP\_RETURN)
  - Multi-Signature
  - Pay-to-Script-Hash (P2SH)

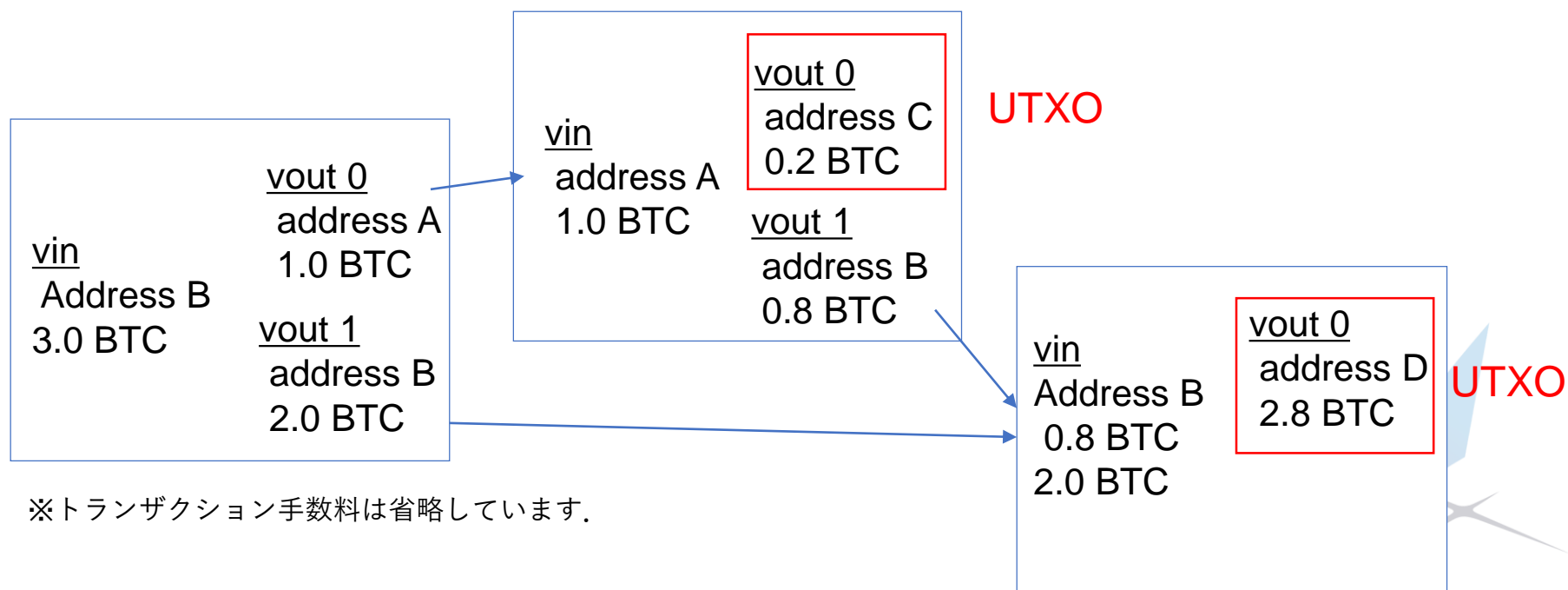


# トランザクションのライフサイクル



## トランザクション間の関係例

- トランザクションはブロックに含まれる, 送金情報
- どのトランザクションからどのアドレスに対していくら送金したか
- 未処理のトランザクションアウトプット (UTXO) から新しくトランザクションを生成できる
  - 元のトランザクションで送金された額を残高として, さらにほかのアドレスに送金する



# トランザクションデータ形式

- トランザクションデータの形式：固定長データ形式, hexadecimal notation

トランザクションID：

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000

Example 1. Alice's transaction, serialized and presented in hexadecimal notation



見やすく変換

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4
"sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

60e316000000000000が送金額  
小さい数から並んでいる（リトルエンディアン）.  
0x16e360=1,500,000 satoshi = 0.015 BTC  
(1億 satoshi = 1 BTC)

# トランザクションID

- トランザクションを一意に識別する識別子
- トランザクションデータをダブルハッシュ（SHA256を2回）した値バイトオーダー逆順にしたもの
- トランザクションハッシュとも呼ばれる.
- 前ページの例だと以下：

トランザクションID：  
0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef80000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000

shibano@DESKTOP-940THE0:~\$ [bx sha256](#)

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef80000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000|[bx sha256](#)  
**f2c245c38672a5d8fba5a5caa44dcef277a52e916a0603272f91286f2b052706**

バイトオーダーを逆順にするとトランザクションIDと一致していることが確認できる.

# トランザクションのデータ構造

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig":
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[
ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```

この例では、1つのInput、2か所のOutputで構成されます。



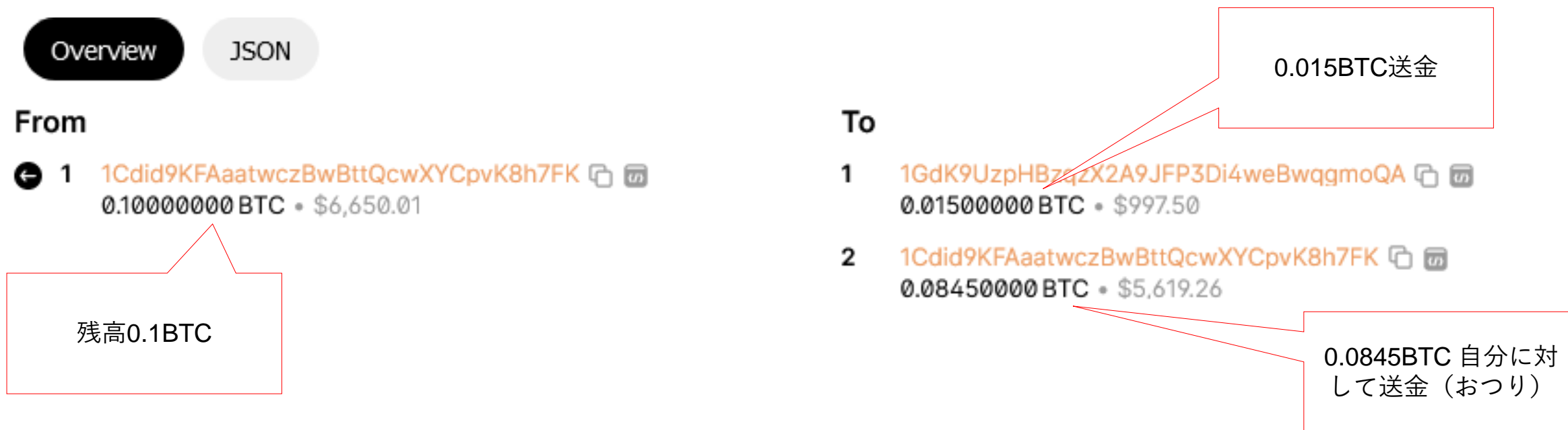
# TransactionのInputとOutput

```
“vin”: [  
  {  
    “txid”:  
    “7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18”,  
    “vout”: 0,  
    . . .  
  },  
  
  {  
    “vout”: [  
      {  
        “value”: 0.01500000,  
        . . .  
      },  
      {  
        “value”: 0.08450000,  
        . . .  
      }  
    ]  
  }  
]
```

インプットの  
TransactionID

インプットランザ  
クション内のアウト  
プットインデックス

## ブロックエクスプローラーでトランザクションをしてみる



Overview JSON

**From**

1 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK 0.10000000 BTC • \$6,650.01

残高0.1BTC

**To**

1 1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA 0.01500000 BTC • \$997.50

0.015BTC送金

2 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK 0.08450000 BTC • \$5,619.26

0.0845BTC 自分に対して送金（おつり）

<https://www.blockchain.com/ja/btc/tx/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2>

送金額合計は  
 $0.015 + 0.0845 = 0.0995$  BTC  
足りない0.0005BTCはトランザクション手数料となる。

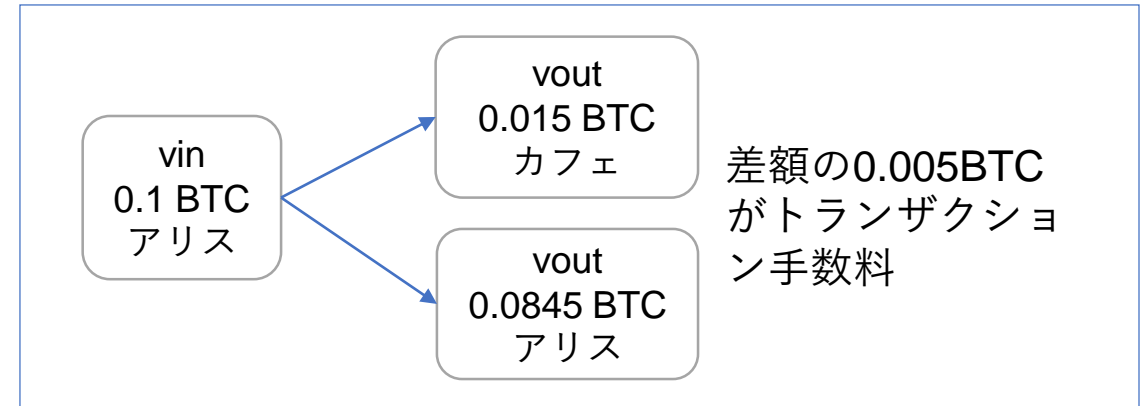
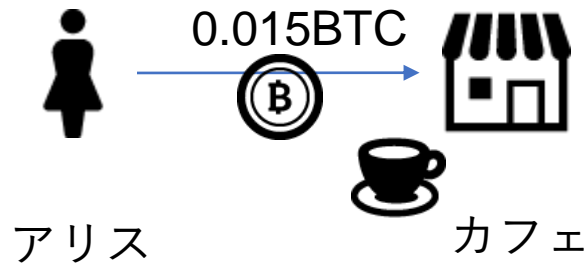


## トランザクション手数料

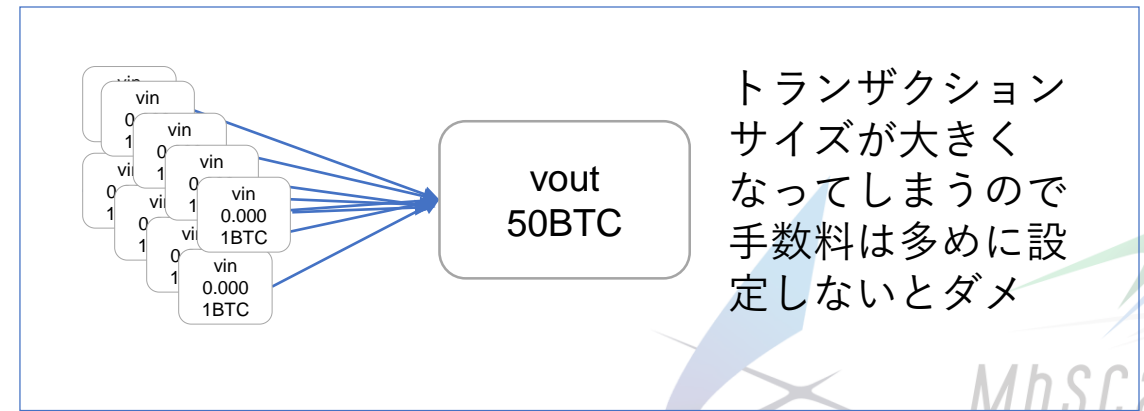
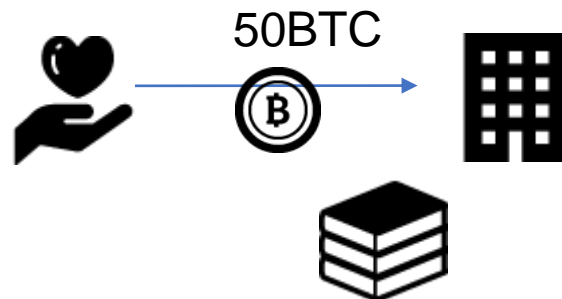
- トランザクション手数料はトランザクションの作成者によって決められる。
  - ほとんどのウォレットは、トランザクション手数料を自動的に計算して入れる。
  - 手動で決めることもできる。
  - トランザクションの大きさに基づいて計算され、送金金額とは関係ない
- 手数料はマイナーに報酬として与えられる。
- マイナーは、どのトランザクションをブロックに含めるかを手数料によって決めている
  - ブロックサイズ 1MB
  - トランザクションの大きさ
  - 手数料
- サイズあたりの手数料を高く設定すればするほどブロックに取り込まれるのが速くなる傾向がある。

## トランザクション手数料の高い・低い例

## アリスのコーヒー代金支払い

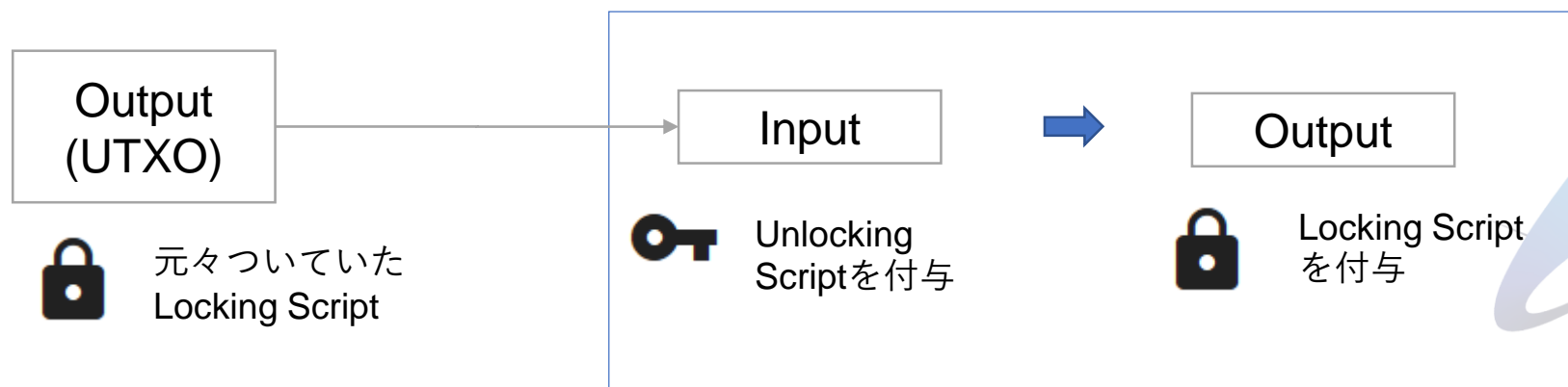


## チャリティ団体が募金で教科書を買う



# Locking Script / Unlocking Script

- Outputのトランザクションは将来, UTXO (未処理のトランザクション) となり, そのアドレスの所有者が送金できる.
- 所有者「だけ」が送金できるようにしなければならない.
- これをLocking Script / Unlocking Scriptという仕組みで解決する.
  - Pay-to-Public-Key-Hashというのが標準的な送金の仕組み
- OutputトランザクションにはLocking Scriptがついている
- そのトランザクションをInputにするには対応するUnlocking Scriptがないとダメ.
  - 不正なUnlocking Scriptのトランザクションはノード到達時に検証に失敗するため伝搬されないしブロックに取り込まれない.
  - 正しいUnlocking Script作成には秘密鍵が必要



# Locking Script/Unlocking Script

Transactionをもう一度見てみる.

```
{  
  "vin": [  
    {  
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
      "scriptSig": "30450221008 . . . .",  
    },  
  ],  
  "vout": [  
    {  
      "value": 0.01500000,  
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"  
    },  
  ],  
}
```

voutにあるvalueは送金金額で, scriptPubKeyというのがLocking Script  
vinにあるscriptSigがtxidで指定しているトランザクションのUnlocking Script

Locking, Unlocking ScriptはBitcoin Scriptで書かれている.

Locking script と Unlocking scriptの関係はBitcoin Scriptで以下がTrueになればよい

Unlocking script

Locking script

Inputトランザクションでは参照しているUTXOのLocking Scriptに対してのUnlocking Scriptを書く

# Bitcoin Script

- 例えば、以下の例を見てみる。

2 3 OP\_ADD 5 OP\_EQUAL

結果はTrueとなる。

Locking Scriptが以下だったとする

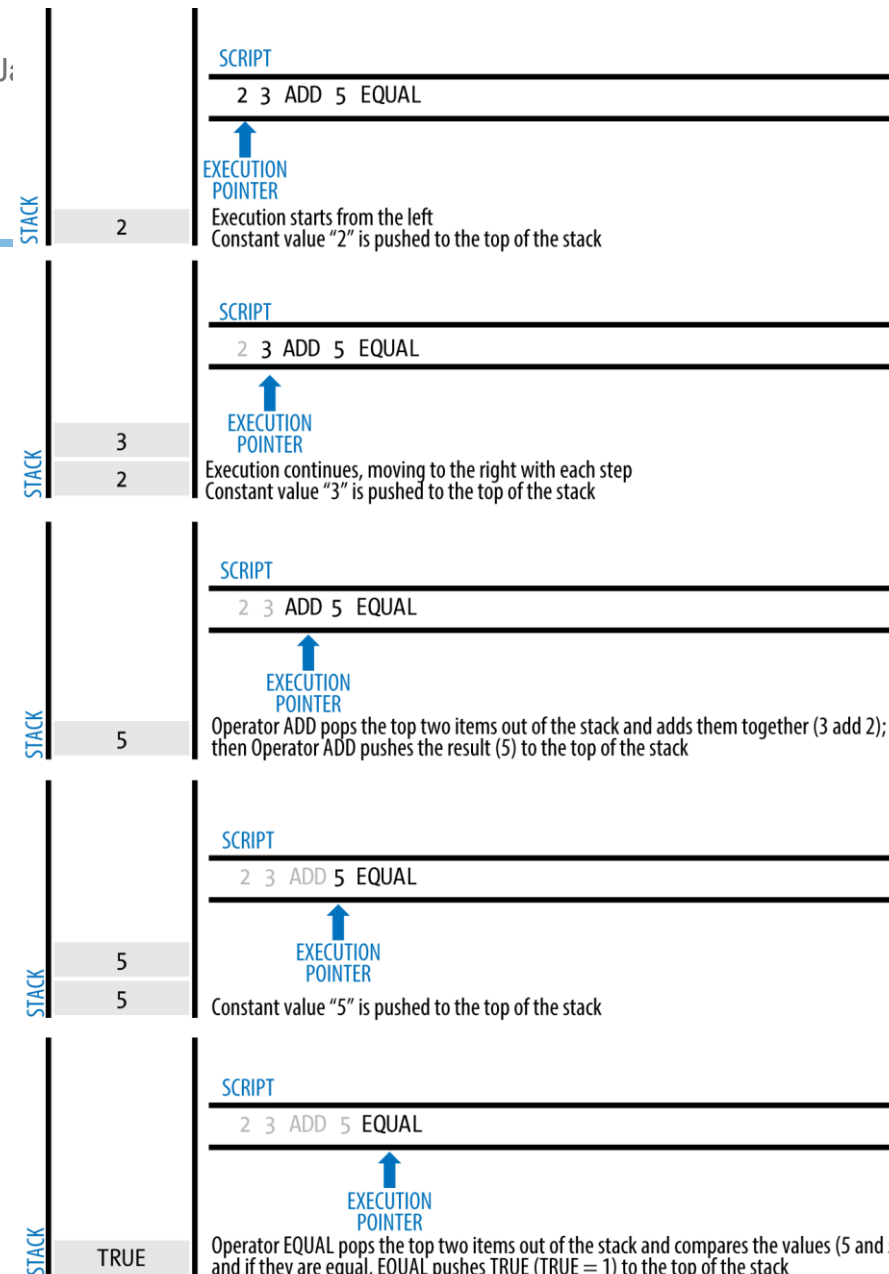
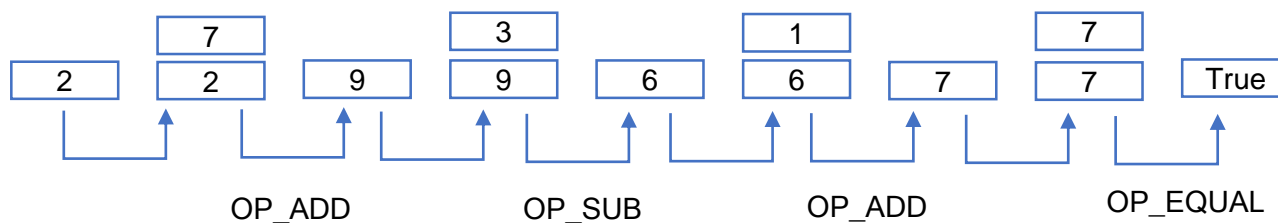
7 OP\_ADD 3 OP\_SUB 1 OP\_ADD 7 OP\_EQUAL

この場合、Unlocking Scriptは次

2

すなわち、以下のScriptがTrueになる

2 7 OP\_ADD 3 OP\_SUB 1 OP\_ADD 7 OP\_EQUAL



[https://github.com/bitcoinbook/bitcoinbook/blob/second\\_edition\\_print3/ch06.asciidoc](https://github.com/bitcoinbook/bitcoinbook/blob/second_edition_print3/ch06.asciidoc)

Figure 4. Bitcoin's script validation doing simple math

## 標準的なトランザクション

- 5つの標準的なトランザクションがある。
  - Locking/Unlocking scriptの種類が5種類
- Pay-to-Public-Key-Hash (P2PKH)
  - 標準的な送金トランザクション
- Pay-to-Public-Key
  - Coinbaseトランザクション
- データアウトプット (OP\_RETURN)
  - 支払いとは関係ないトランザクション用
- Multi-Signature
  - 複数人の署名が必要なトランザクション
- Pay-to-Script-Hash (P2SH)
  - より複雑な処理が可能に





# Pay-to-Public-Key-Hash -- Locking ScriptとUnlocking Script

Unlocking script

Locking script

がTrueになるように、Inputトランザクションではすでに書かれているLocking Scriptに対してUnlocking Scriptを書く

Aliceがカフェで0.015BTC支払う場合、そしてカフェがそれをUnlockすることを考えてみる。

Locking Script：（Aliceが作成）

OP\_DUP OP\_HASH160 <Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

Unlocking Script：（カフェが作成）

<Cafe Signature> <Cafe Public Key>

Public Key Hash (160bit)は、アドレスをBase58でデコードして抽出

つなぎあわせた以下のScriptがTrueになればOK

<Cafe Signature> <Cafe Public Key> OP\_DUP OP\_HASH160 <Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

**Unlocking Script (カフェが作成)**

**Locking Script (Aliceが作成)**

**Script:** `<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG`

**Stack:**

- Initial: `<sig>`
- After `<PubK>` push: `<PubK>`, `<sig>`
- After `DUP`: `<PubK>`, `<PubK>`, `<sig>`
- After `HASH160`: `<PubKHash>`, `<PubK>`, `<sig>`
- After `EQUALVERIFY`: `<PubKHash>`, `<sig>`
- After `CHECKSIG`: `TRUE`

**Execution Steps:**

- Execution starts. Value `<sig>` is pushed to the top of the stack.
- Execution continues, moving to the right with each step. Value `<PubK>` is pushed to the top of the stack, on top of `<sig>`.
- `DUP` operator duplicates the top item in the stack, the resulting value is pushed to the top of the stack.
- `HASH160` operator hashes the top item in the stack with `RIPEMD160(SHA256(PubK))` the resulting value (`PubKHash`) is pushed to the top of the stack.
- The value `PubKHash` from the script is pushed on top of the value `PubKHash` calculated previously from the `HASH160` of the `PubK`.
- The `EQUALVERIFY` operator compares the `PubKHash` encumbering the transaction with the `PubKHash` calculated from the user's `PubK`. If they match, both are removed and execution continues.
- The `CHECKSIG` operator checks that the signature `<sig>` matches the public key `<PubK>` and pushes `TRUE` to the top of the stack if true.

**OP\_EQUALVERIFY:** → 2つの値が等しかったら処理を続行. 異なっていたら異常として終了.

**公開鍵から計算されるHASH160がLocking Scriptに含まれるHASH160（アドレスから抽出）と一致しているか**  
→ アドレスと対応する公開鍵を署名検証で用いていることを証明.

**Unlock Scriptに含まれる署名と公開鍵の検証**  
→ 公開鍵に対応する秘密鍵の持ち主であることを証明

Figure 5. Evaluating a script for a P2PKH transaction

## Pay-to-Public-Key-Hash -- CHECKSIGでやっていること

- 署名とそれを検証するってどういうことか？
- ECDsaでの署名検証.
  - 公開鍵を用いて、対象メッセージに対する署名が、公開鍵に紐づく秘密鍵でなされていることを検証できる.
  - 参考：<https://ja.wikipedia.org/wiki/%E6%A5%95%E5%86%86%E6%9B%B2%E7%B7%9ADSA>
- 署名の対象は、そのトランザクションに含まれるInput, Outputを含むデータ (SIGHASH\_ALL)
  - 送金先や金額を他の人が変更できないように.
  - 参考：[https://en.bitcoin.it/wiki/OP\\_CHECKSIG](https://en.bitcoin.it/wiki/OP_CHECKSIG)

“scriptSig” :

```
“3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf”,
```

署名

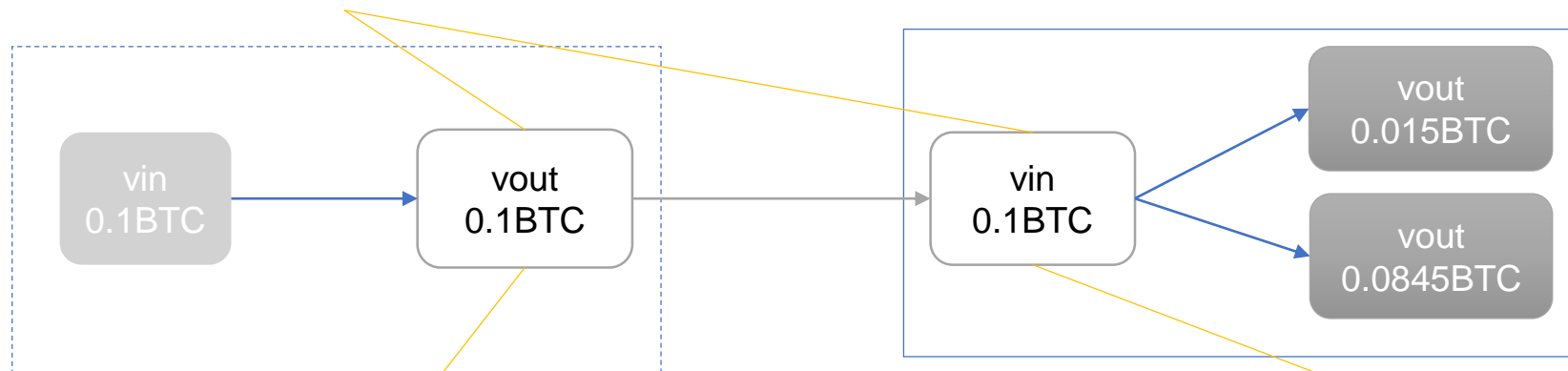
公開鍵

(詳細は割愛します)  
要するに、秘密鍵を用いて署名を生成でき、その署名を公開鍵を用いて検証できる。

# Pay-to-Public-Key-Hashまとめ

送金トランザクション

アドレス：1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK



Output script (Locking script)

```
OP_DUP OP_HASH160
7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG
```

青：公開鍵ハッシュ=アドレスの一部

※ 公開鍵ハッシュはアドレス中に埋め込まれているため、送金先アドレスを知っていればロックングスクリプトの作成ができる。

```
$ bx base58check-decode 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
wrapper
{
  checksum 3326777089
  payload 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
  version 0
}
```

Input script (Unlocking script)

```
483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae2
4cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301410
484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767
89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf
```

緑：署名  
青：公開鍵

<https://btc.com/7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18>  
<https://btc.com/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2>

# Pay-to-Public-Key-Hashまとめ

## Output script (Locking script)

```
OP_DUP OP_HASH160
7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc0
25a8 OP_EQUALVERIFY
OP_CHECKSIG
```

青：公開鍵ハッシュ = アドレスの一部

## Input script (Unlocking script)

```
483045022100884d142d86652a3f47ba4746ec719bbfbd040
a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f
6addac960298cad530a863ea8f53982c09db8f6e381301410
484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade
8416ab9fe423cc5412336376789d172787ec3457eee41c04f
4938de5cc17b4a10fa336a8d752adf
```

緑：署名  
青：公開鍵

```
483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b
9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30
928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f493
8de5cc17b4a10fa336a8d752adf
```

```
OP_DUP OP_HASH160
7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
OP_EQUALVERIFY OP_CHECKSIG
```

が実行される。

やっていることをまとめると

- Inputスクリプトにある公開鍵がOutputスクリプトのものと同じかどうかチェックされる
- その公開鍵をもって、署名の検証が可能か  
→ 秘密鍵をもってそのTxを作ったかどうかのチェックにつながる



# Pay-to-Public-Key

- P2PKHよりシンプルな送金形式
- ロッキングスクリプト内には公開鍵ハッシュではなく公開鍵そのものを使用
  - 送金する側が，送金先アドレスの公開鍵も必要になる
  - アドレスだけでは作れない
- coinbaseトランザクションで使用する。
  - ブロック内に存在するマイナーへのマイニング報酬のトランザクション

Unlocking script

Locking script

ロッキングスクリプト：

<Public Key A> OP\_CHECKSIG

アンロッキングスクリプト：

<Signature from Private Key A>

検証用の結合されたスクリプト：

<Signature from Private Key A>

<Public Key A> OP\_CHECKSIG

## データアウトプット (OP\_RETURN)

- ビットコインはタイムスタンプ付きの分散台帳
  - タイムスタンプはブロックに含まれます
- 支払い以外の情報の記録でビットコインブロックチェーンを使用したい。
  - ファイルの存在証明など
- ビットコインの開発者の中で意見が分かれる
  - 賛成派：ビットコインの支払い以外にも応用の幅を広げるべき
  - 反対派：そんなことするとブロックチェーンが肥大化してしまう
    - 支払いとは関係ないトランザクションが増える
    - UTXOが増えてしまう
    - UTXOは別DBで管理
- 妥協案としてOP\_RETURNコードが導入された

OP\_RETURN <data>

- このdataは80バイト
  - 多くの使用例では、何かのハッシュ値のみ保存する
- OP\_RETURNを使用するためのアンロッキングスクリプトは存在しない
  - UTXOにカウントされない

## マルチシグネチャ

- ロッキングスクリプトにいくつかの公開鍵を登録しておき，そのうちのいくつかの署名をアンロッキングスクリプトで必要とするトランザクション
- 全部でN個の公開鍵を登録し，送金にはそのうち少なくともM個の署名が必要 (M of N).
  - $M \leq N$
- 例えば2-of-5 マルチシグネチャの場合は5個の公開鍵のうち2つの署名が必要.
- 会社のアドレスなど，複数人の署名がないと送金できなくしたい場合に使用される.
- 2-of-5マルチシグのロッキングスクリプト例：
  - とある会社では会社のアドレスからの送金に，社長と役員3人，弁護士の計5人のうちの2人の署名を必要としている
- マルチシグネチャscriptでは最大15個の公開鍵が使用可能.
  - 1-of-1から15-of-15の任意の組み合わせで使用できる





# マルチシグネチャ

## 2-of-5マルチシグの例

Unlocking script

Locking script

## ロッキングスクリプト：

```
2 <Public Key A> <Public Key B> <Public Key C> <Public Key D> <Public Key E> 5 CHECKMULTISIG
```

## アンロッキングスクリプト：

```
0 <Signature B> <Signature C>
```

## 検証用の結合されたスクリプト：

```
0 <Signature B> <Signature C>
```

```
2 <Public Key A> <Public Key B> <Public Key C> <Public Key D> <Public Key E> 5 CHECKMULTISIG
```

※ アンロッキングスクリプトの最初の「0」はCHECKMULTISIGにバグが含まれており，そのために必要な要素，処理では無視されるが空にはできないので慣例的に0を置いて代用する。

## Pay-to-Script-Hash (P2SH)

- 複雑なscriptを単純化できるようにしたもの
- 2-of-5マルチシグのロッキングスクリプト例：

```
2 <自分の Public Key> <役員1の Public Key> <役員2の Public Key> <役員3の Public Key> <弁護士の Public Key> 5 CHECKMULTISIG
```

- マルチシグネチャの問題点：
  - 支払い前にマルチシグのスクリプトを共有する必要がある.
  - 複数の公開鍵を含むためスクリプトが長い.
    - 公開鍵は圧縮された形式でも264bit
  - したがって**トランザクションサイズも大きくなり**，送金手数料も増大する.
- 受け取り側の都合でマルチシグを導入しているのにも関わらず，送金側にその負担を強いてしまう.
  - ロッキングスクリプトは，その受取側に送金するときのトランザクション内に入る.

## Pay-to-Script-Hash (P2SH) - Redeem scriptの導入

- P2SHでは、スクリプトそのものの代わりにそのスクリプトのハッシュ値をロックングスクリプトに用いる
- そのスクリプトをリディーム（引き換え）スクリプト（Redeem script）と呼ぶ
- これにより、長いスクリプトはアンロックングスクリプト内に含まれることになる。

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 1. Complex script without P2SH

Sig1 Sig2 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Locking Script	HASH160 <20-byte hash of redeem script> EQUAL
Unlocking Script	Sig1 Sig2 <redeem script>

Table 2. Complex script as P2SH

Sig1 Sig2 <redeem script> HASH160 <20-byte hash of redeem script> EQUAL

## どれくらい短くなるか

- マルチシグでP2SHを使用していない状態と使用した状態でのロッキングスクリプトを比較する。

2 <自分の Public Key> <パートナー1の Public Key> <パートナー2の Public Key> <パートナー3の Public Key> <弁護士の Public Key> 5 CHECKMULTISIG

P2SHを使用していない時のロッキングスクリプトは

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF
5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78
D0E34224858008E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3
020DECDBC3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645AB
E8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022
DD618DA774D207D137AAB59E0B000EB7ED238F4D800 5 CHECKMULTISIG
```

この長いスクリプト全体をダブルハッシュ（SHA256後RIPEMD160）する。

```
$ echo ¥
> 2 ¥
> [04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] ¥
> [04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] ¥
> [047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] ¥
> [0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] ¥
> [043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] ¥
> 5 CHECKMULTISIG ¥
> | bx script-encode | bx sha256 | bx ripemd160
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

P2SHを使用した時のロッキングスクリプトは以下。

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

サイズをずいぶん小さくできることがわかる。

# アンロッキングスクリプトは？

P2SHを使用していない時のアンロッキングスクリプトは

0 <Sig 1> <Sig 2>

P2SHを使用した時のアンロッキングスクリプトは

<Sig1> <Sig2> 2  
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900  
A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4  
D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B  
9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D99779650421D65CBD7149B255382  
ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2A  
FD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA7  
74D207D137AAB59E0B000EB7ED238F4D800 5 CHECKMULTISIG

となり、長いスクリプトがアンロッキング側に移っていることがわかる

検証用のスクリプトは

Unlocking script

Locking script

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

※ Redeemスクリプトのハッシュ値が一致しているかの確認後、Redeemスクリプト内部が実行される

<https://learnmeabitcoin.com/technical/p2sh>

## P2SHのアドレス

- P2SHに関してスクリプトハッシュをBase58Checkエンコードしてアドレスを生成できる.

```
$ echo ¥  
> '54c557e07dde5bb6cb791c7a540e0a4796f5e97e'¥  
> | bx address-encode -v 5  
39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw
```

バージョンプレフィックスに「5」  
を指定

```
$ bx address-decode 39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw  
wrapper  
{  
  checksum 4100823282  
  payload 54c557e07dde5bb6cb791c7a540e0a4796f5e97e  
  version 5  
}
```

プレフィックスが3となる.

このアドレスを顧客に教えておけば、顧客は通常の支払いと同様にウォレットを使用して送金できる。  
(ウォレットはP2SHに対応している必要がある)





## P2SH利点

---

- 送金者のトランザクションのデータサイズを小さくする
- スクリプトハッシュはアドレスとして表現される.
- 長いScriptを含むトランザクションは使用者のアンロッキングスクリプト側になるため, 高額なトランザクション手数料はP2SHの使用者側に移る





## まとめ

---

- UTXOモデル
- トランザクションデータ
  - トランザクションID
  - インプット, アウトプット
  - 手数料
- 鍵の使い方
  - ロッキングスクリプト, アンロックスクリプト
- 様々なトランザクション5種類
  - Pay-to-Public-Key-Hash (P2PKH)
  - Pay-to-Public-Key
  - データアウトプット (OP\_RETURN)
  - Multi-Signature
  - Pay-to-Script-Hash (P2SH)





# ブロックチェーン

---



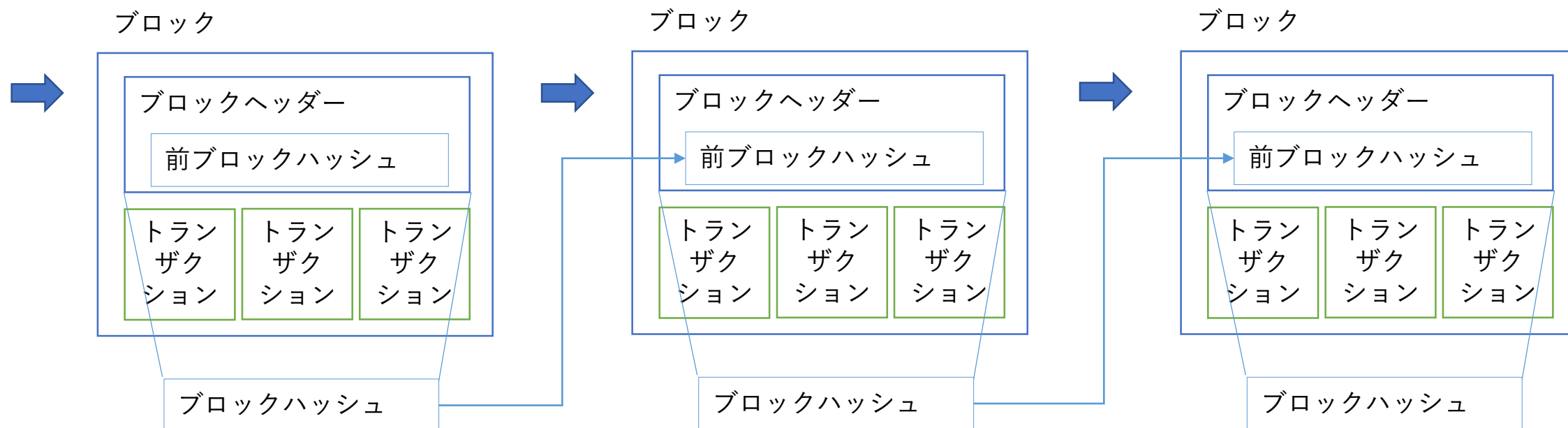
## 概要

---

- ブロックチェーンの構造の説明
  - ブロック + チェーン
- マークルツリー
- ブロック内に入っているデータ
- ブロック識別子
- Genesisブロック

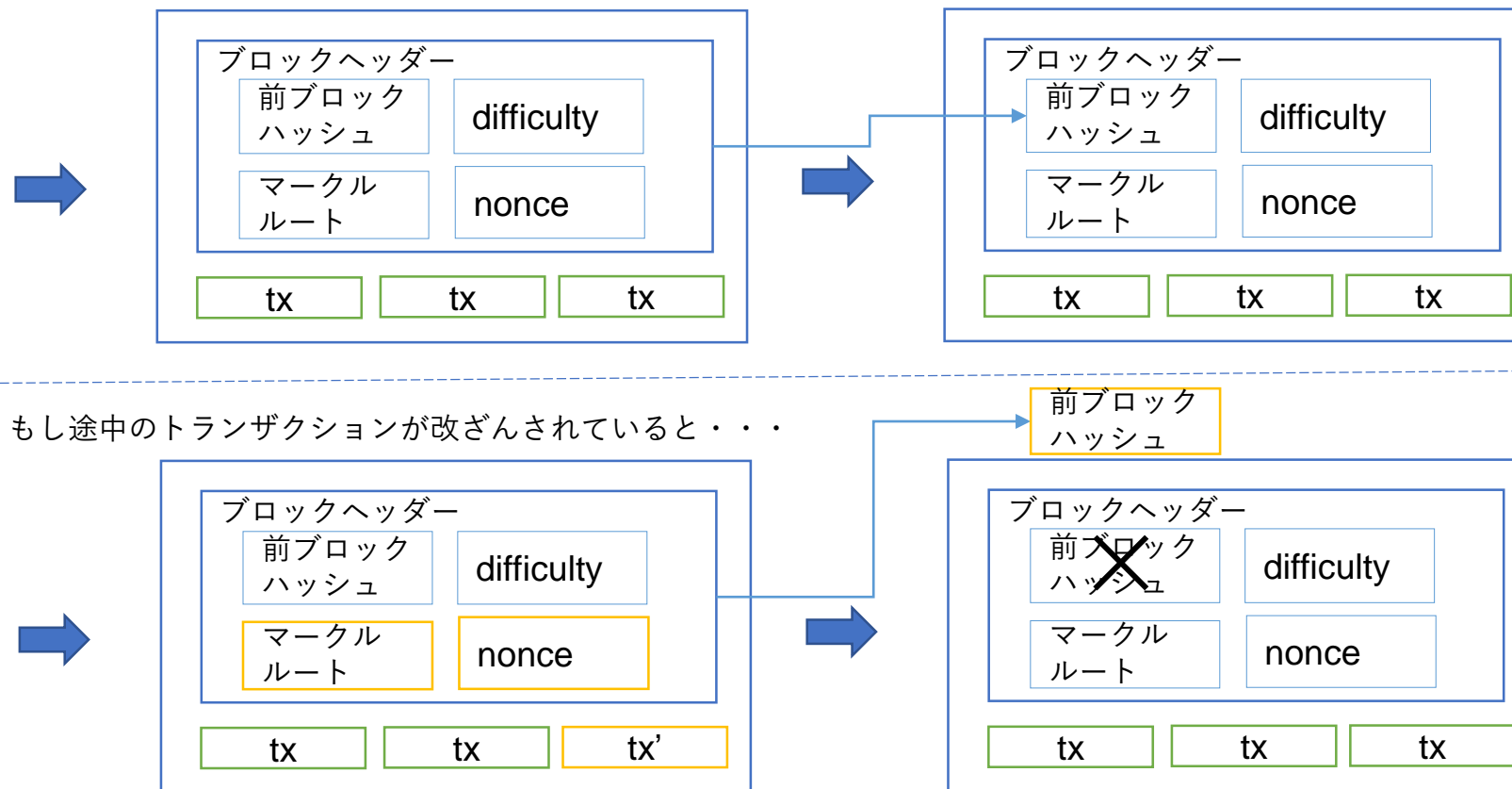


## ブロックの連鎖



前ブロックのハッシュ値が自身のブロックに含まれる。  
それが連鎖状に連なっている。  
最初のブロック（0番目，Genesisブロック）まで連鎖が続く。

# ブロックハッシュによる改ざん困難さ



まずは改ざんしようとしているトランザクションを含むブロックの検証を通るようにしなければならない

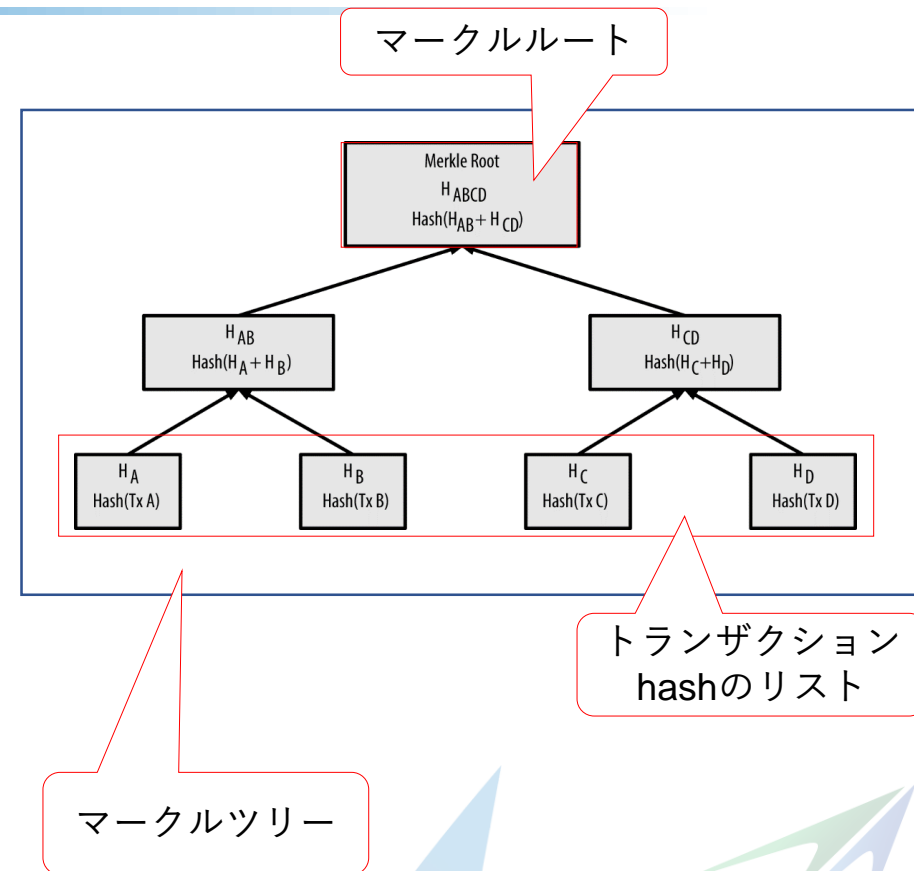
- 新たなナンス値を求める必要

それができたとしても次のブロックに含まれるハッシュ値と相違してしまうため、つながらなくなる。

最長ブロックのチェーンを選択するようになっているため連鎖的にそのブロック以降のナンス値をすべて改めて求めていく必要が出てしまう。

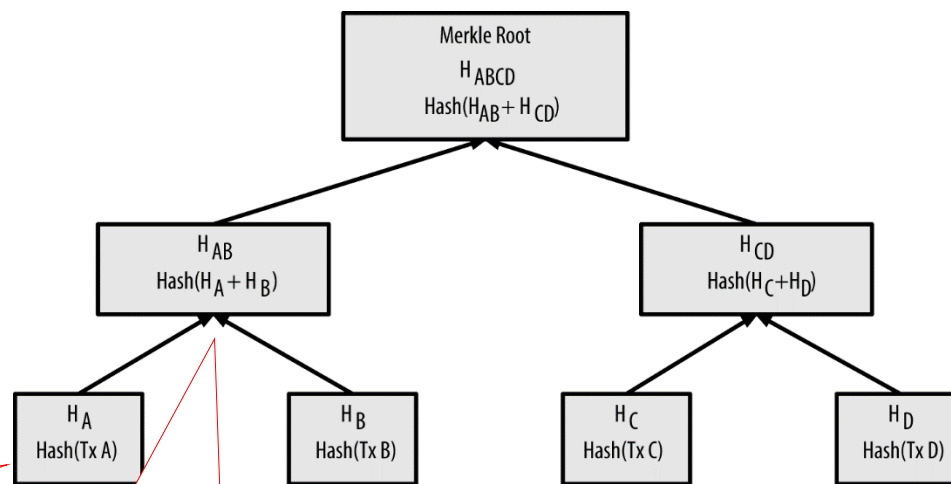
# マークルツリー

- ブロック内に含まれるトランザクション全体（値や順番）の要約を行う。
- マークルルートという32バイトの値で、ブロック内の全トランザクションを表現できる。
  - データの転送量を削減しつつ、特定のトランザクションがブロック内に含まれていることを効率的に確認できるように。
- マークルルート
  - マークルツリーのルート：ハッシュ値
  - トランザクションリストのフィンガープリント
    - トランザクションの順番や値が変更されるとマークルルートの値も変わる
  - ブロックヘッダーに含まれている。
  - そのブロックに含まれているトランザクションリストを一意に表現できる。
- マークルツリーを使うことで・・・
  - とあるトランザクションがそのブロックに含まれているかどうかを全トランザクションデータを用いなくても検証できる。
    - マークルツリーの一部のみ使用すれば検証可能
    - SPVノードで使用される
      - ネットワーク転送量の削減につながる



## マークルツリー

- 二分ハッシュ木(binary hash tree)とも呼ばれる。
- SHA256を2回 (double-SHA256)
- データ構造上, 一番上が「根」(root), 一番下が「葉」(leaf).



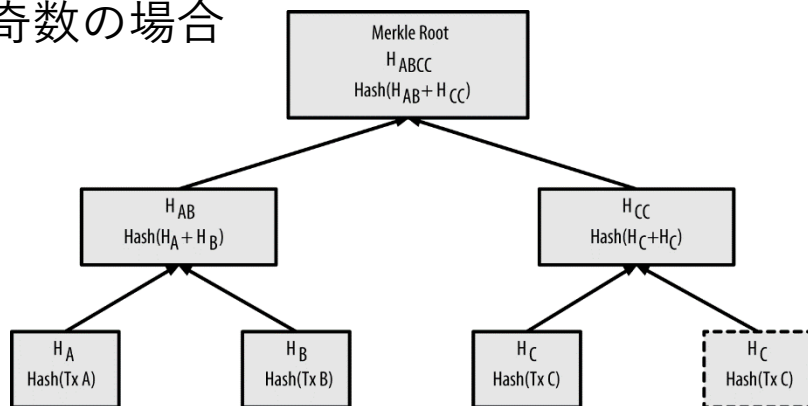
$H_A = \text{SHA256}(\text{SHA256}(\text{Transaction A}))$

Figure 2. Calculating the nodes in a merkle tree

$H_{AB} = \text{SHA256}(\text{SHA256}(H_A + H_B))$   
ここで「+」は値の連結

## 足りない葉は，データ要素を二重に使うことで解消している

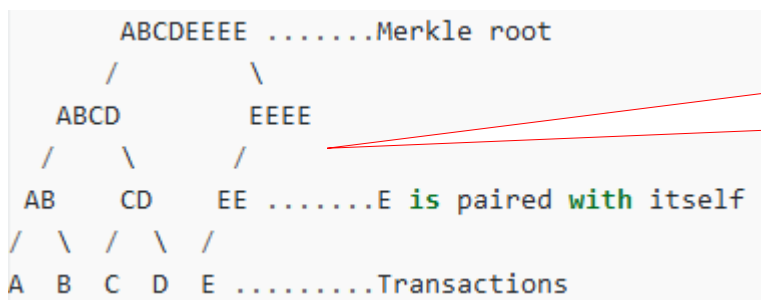
要素数が奇数の場合



データ構造上，となりのTx\_Cをコピーして使用する。

Figure 3. Duplicating one data element achieves an even number of data elements

一般的には・・・



要素数が2のべき乗個でない場合は，足りない葉はすべて同じTx\_Eが重複して存在することにして計算

[https://developer.bitcoin.org/devguide/block\\_chain.html](https://developer.bitcoin.org/devguide/block_chain.html)

## どれだけ多くのトランザクションでも表現できる

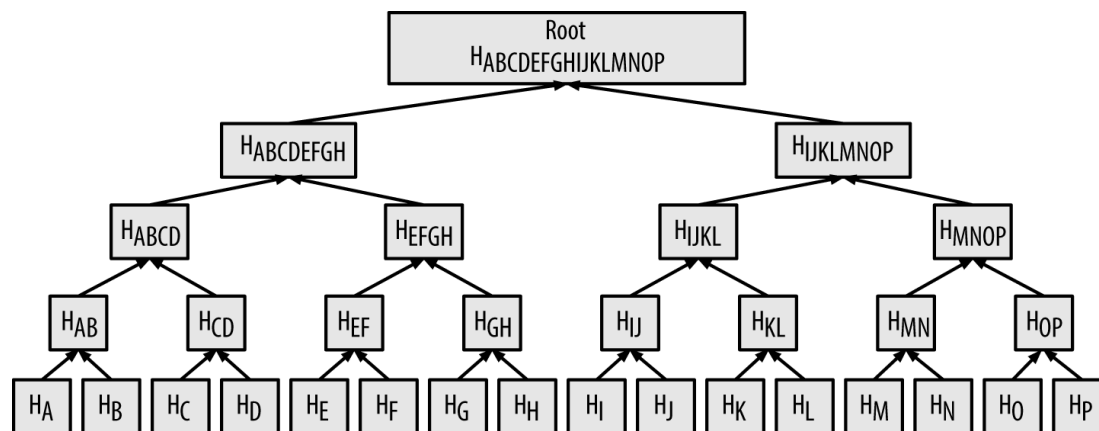


Figure 4. A merkle tree summarizing many data elements

トランザクションの数が1つでも100万でもマークルルートの値は同じ32バイト。  
トランザクションの一つでも値が違ったり順番が違うと異なるマークルルートの値となる。  
→ トランザクションリストをユニークな値として表現できる。



## マークルパスを与えるだけで検証可能

- マークルパスとはマークルツリーの一部で、そのトランザクションの検証に必要な途中の葉
- マークルルートが既知の状態、あるトランザクションがそのマークルツリーに含まれているかを確認したいときに使用される。
  - SPVノードはブロックヘッダーのみ保持。すなわちマークルルートを持っている。
  - そのブロックに含まれるTxを安全に他のフルノードから取得するにはトランザクションとともにマークルパスを取得。

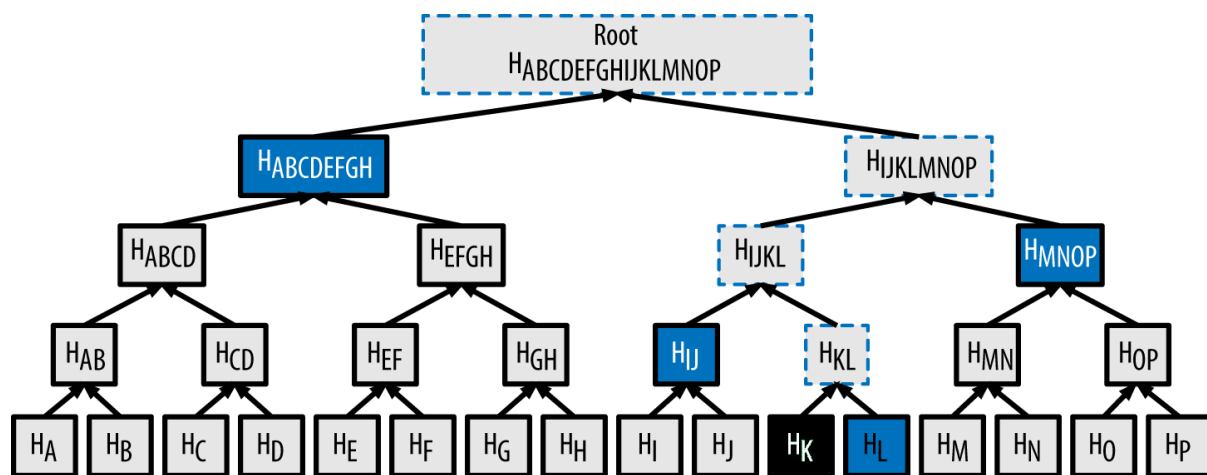


Figure 5. A merkle path used to prove inclusion of a data element

トランザクションKがブロックに含まれているかどうかを検証する。  
マークルパスとしてH\_ABCDEFGH, H\_MNOP, H\_IJ, H\_Lの4つ値があれば検証可能。

$H_K \rightarrow K$ から計算

$H_{KL} \rightarrow H_K$ と $H_L$ から計算

$H_{IJKL} \rightarrow H_{IJ}$ と $H_{KL}$ から計算

$H_{IJKLMNOP} \rightarrow H_{IJKL}$ と $H_{MNOP}$ から計算

Root  $\rightarrow H_{ABCDEFGH}$ と $H_{IJKLMNOP}$ から計算

計算されたRootが既知のマークルルートと一致しているかどうかを確認する。

## 補足) 手計算を試みる

- トランザクションID
  - トランザクションID (トランザクションハッシュ) もSHA256を2回かけたもの
  - ビッグエンディアン
- ブロックヘッダ内のマークルルート
  - ビッグエンディアン
- 上記に注意して手計算を行う.

### エンディアンを変換するコマンド

```
$ echo cd6f845e1f09e9a265df38deae2977dbe0baca06669ccaf216a9a3d02cfdd07a | fold -w2 | tac | tr -d '\n'
7ad0fd2cd0a3a916f2ca9c6606cabae0db7729aede38df65a2e9091f5e846fcd
```

### ダブルハッシュのコマンド (bitcoin explorerを使用)

```
$ echo 7ad0fd2cd0a3a916f2ca9c6606cabae0db7729aede38df65a2e9091f5e846fcd | bx sha256 | bx sha256
ca8671ed5d393f90f4640244c85d917ba708a9a3ccc3555437b2ad31e042a2d8
```

ハッシュの計算中の値はリトルエンディアンで, トランザクションハッシュ, マークルルートの各値はビッグエンディアンであることに注意

## 補足) 実データのマークルルートを計算してみる

- <https://www.blockstream.info/testnet/block/000000002b35701826c61e9657b464e6418338f11f236c65903efe142a298636>
- を実際に計算してみる。 (bitcoinテストネットのブロック)
- トランザクションが3つ入っており、それぞれのトランザクションからマークルツリーを構築し、ブロックヘッダーのマークルルートと値が一致するか確認する。

### トランザクションID

e9623f9e8637d1ffd26c6815ac3581577cc637  
6d2a7d7d4d6fed813df5f4b09f

5a47195b6f29ee5fa84b47adccc0037c875ee  
ef95039e63e1cab0fa588b1ab53

40c047626b4e87dda92b4bb0b09dca3d1d8b  
db94077d80dd092d97dd13e72a01



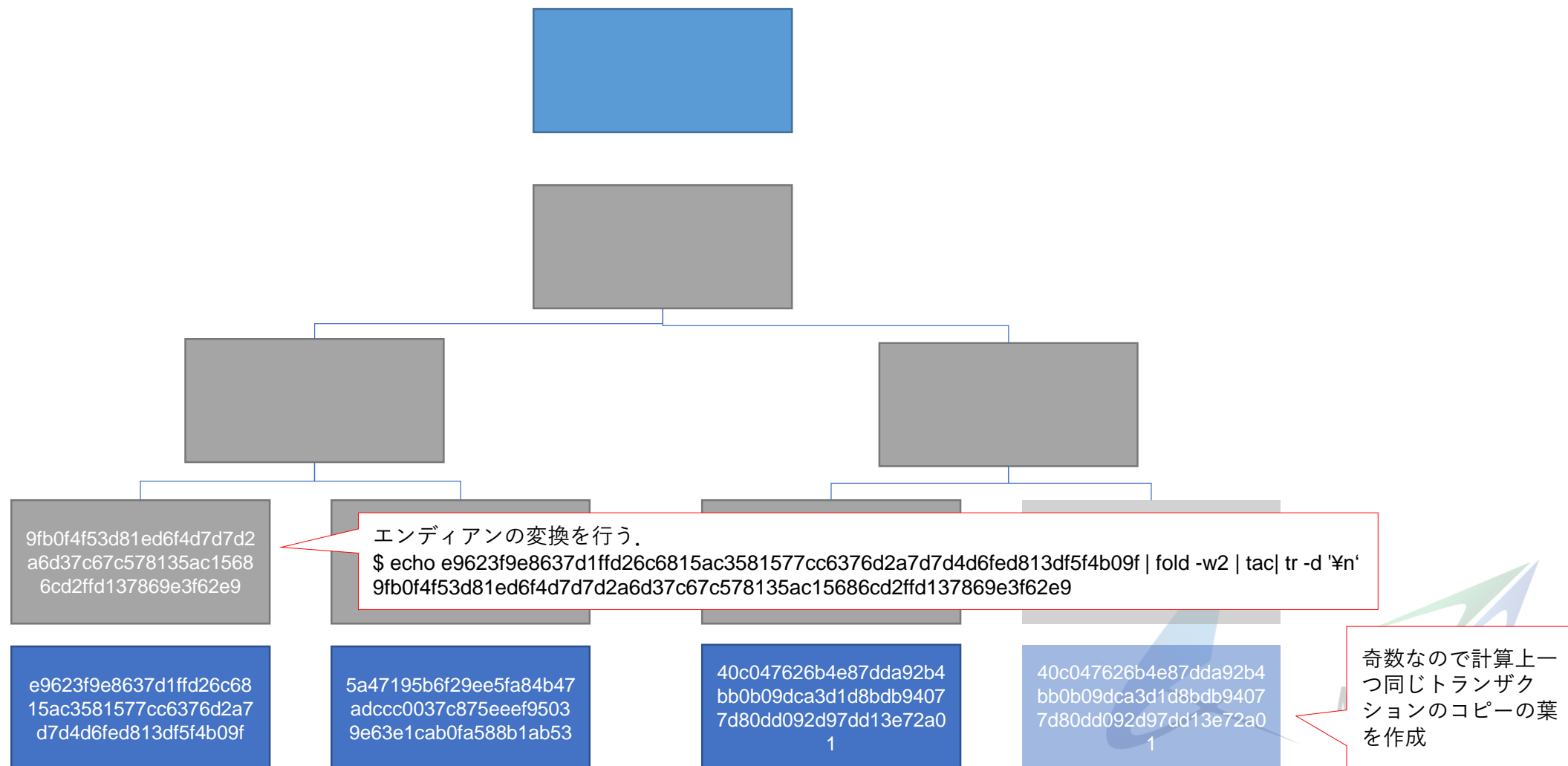
実際に計算してみて値が  
一致するか確認してみる。

### マークルルート

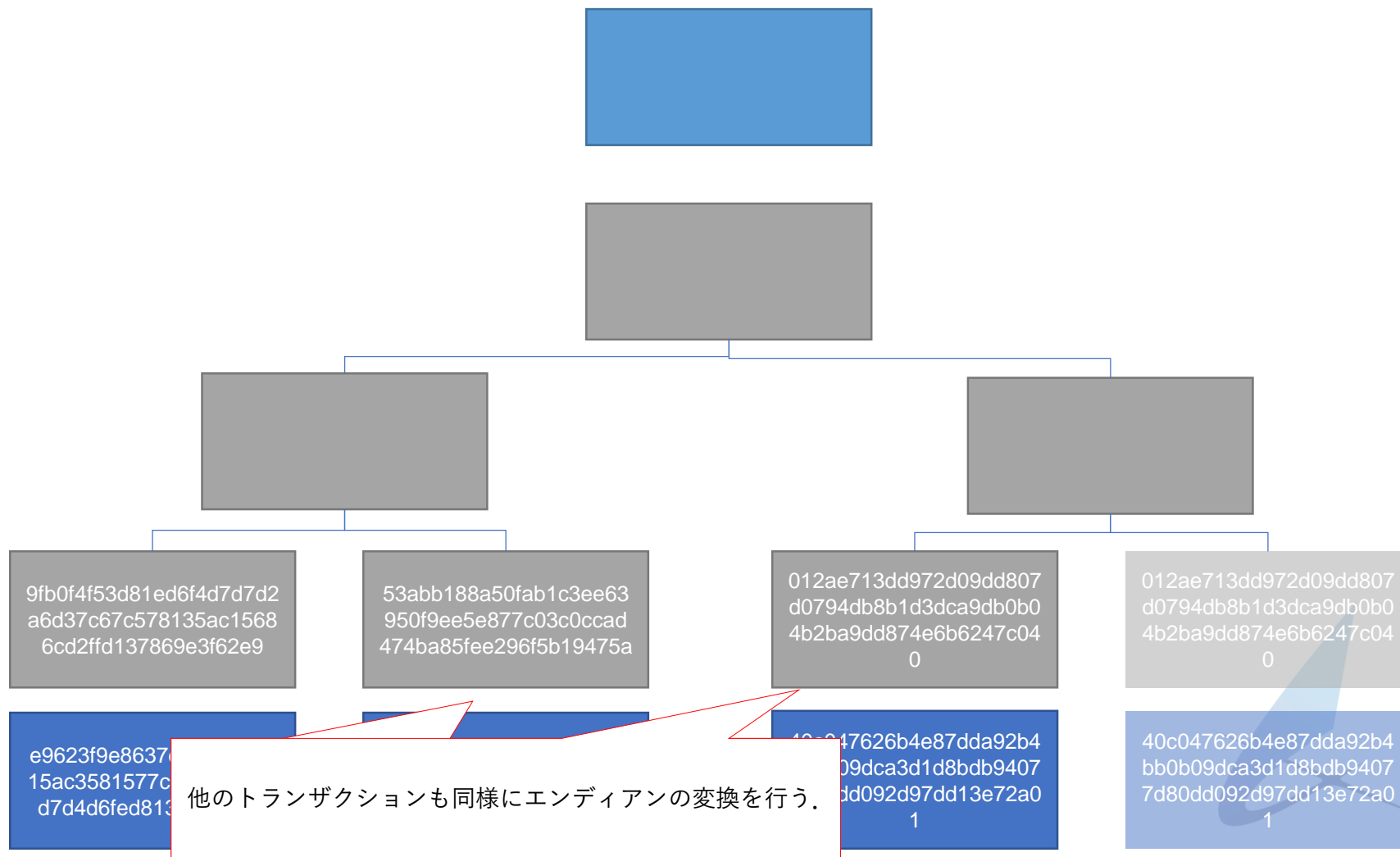
7ad0fd2cd0a3a916f2ca9c6606cabae0db772  
9aede38df65a2e9091f5e846fcd



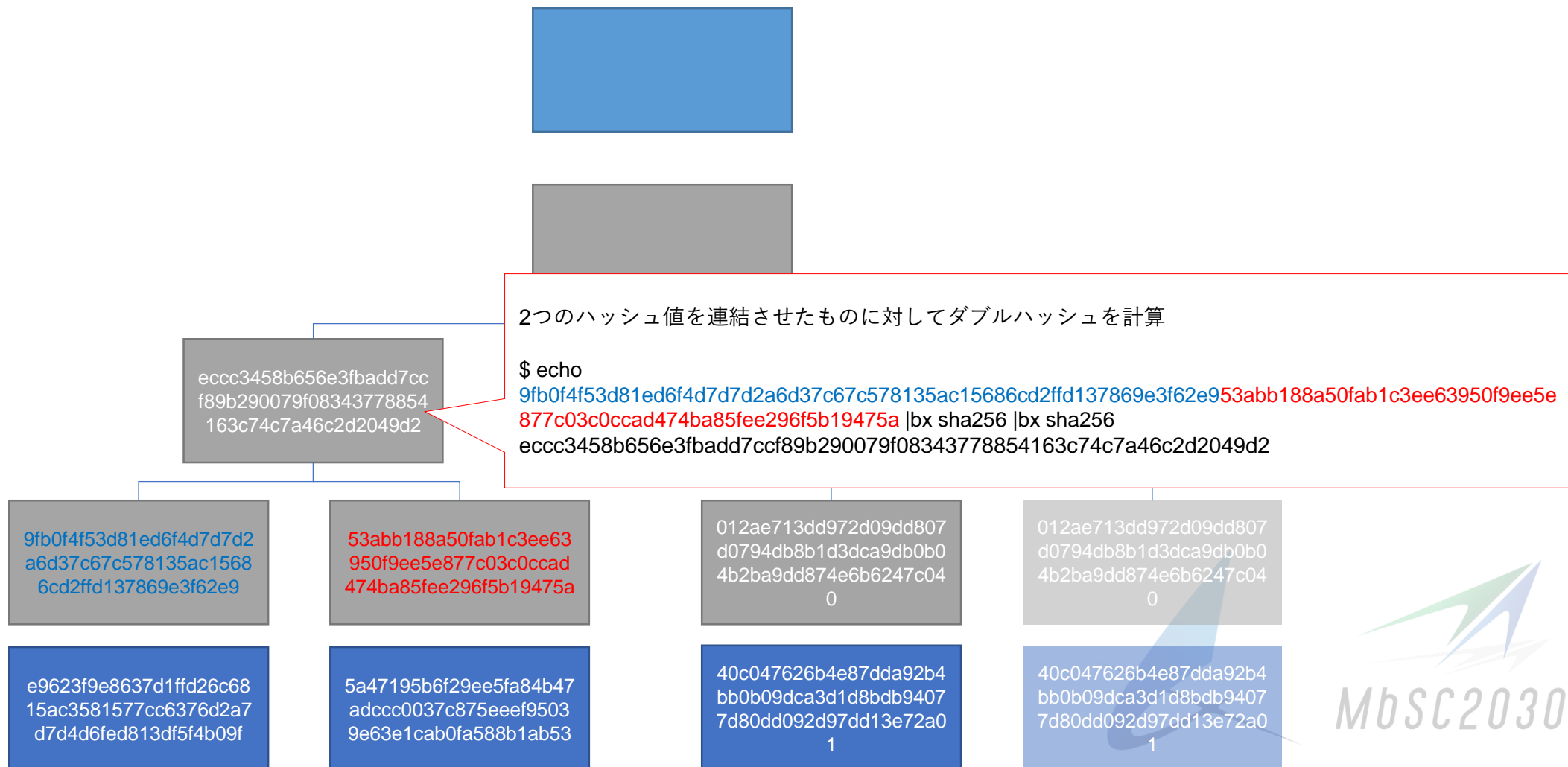
## 補足) マークルツリーを完成させる



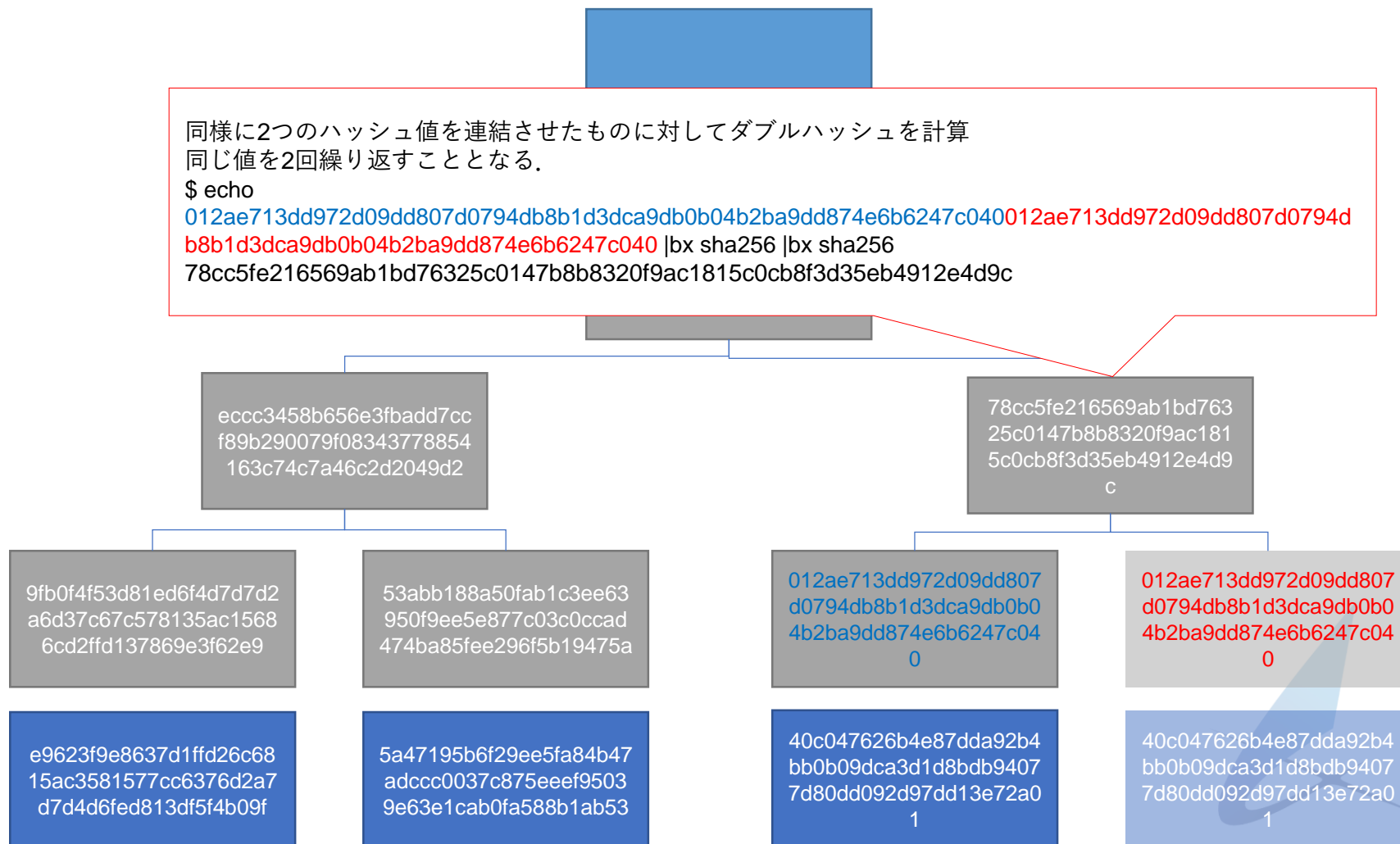
## 補足) マークルツリーを完成させる



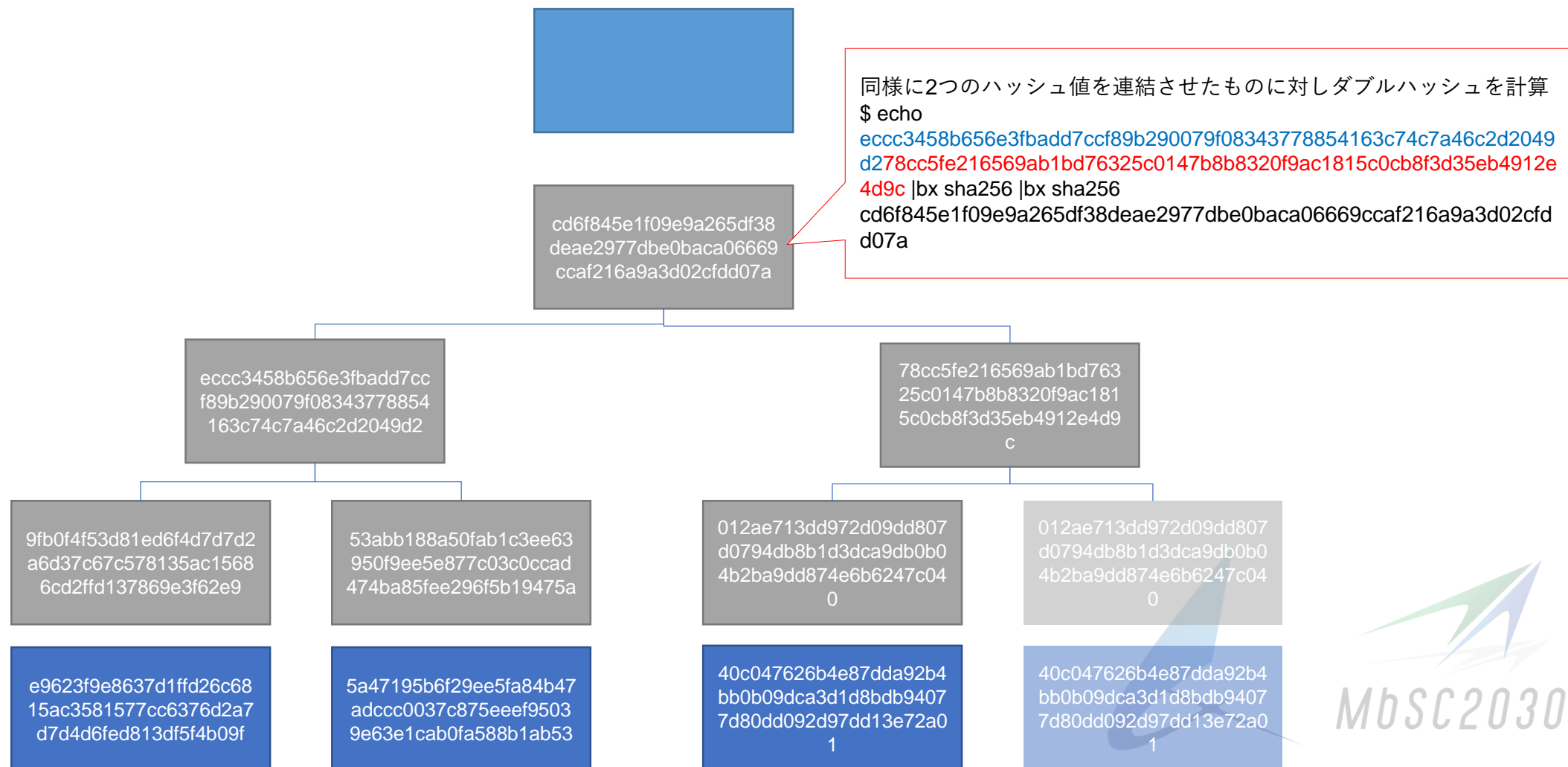
## 補足) マークルツリーを完成させる



## 補足) マークルツリーを完成させる

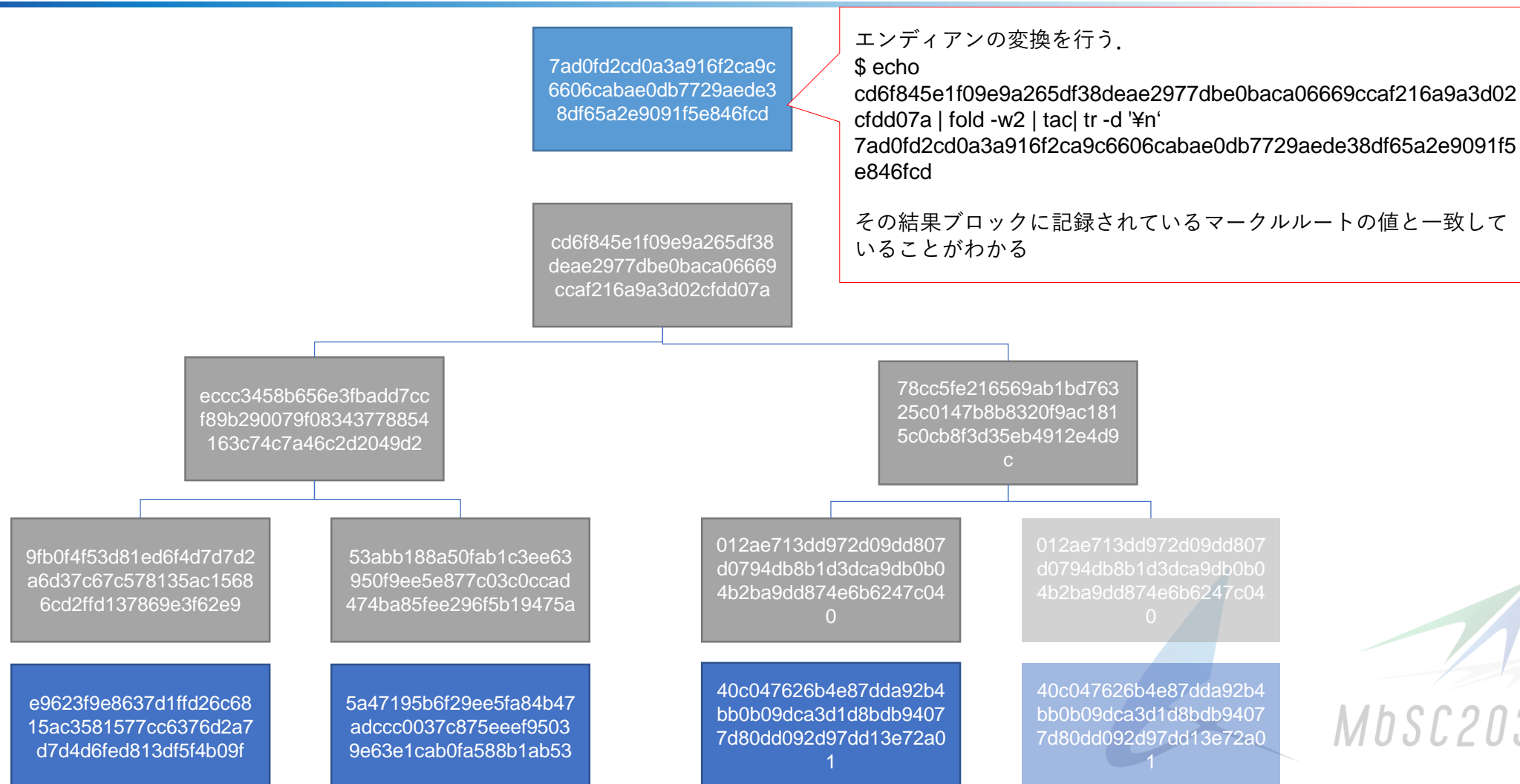


## 補足) マークルツリーを完成させる





## 補足) マークルツリーを完成させる



# ブロックの構造：詳細

## ブロックの構造

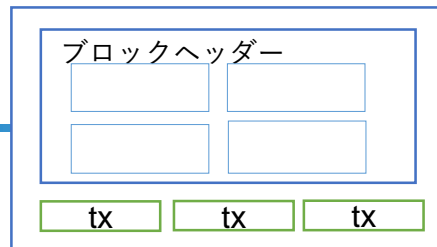
### 固定長データ

サイズ	内容	詳細
4 bytes	ブロックサイズ	このフィールドを含まない、このブロックのサイズ
80 bytes	ブロックヘッダー	いくつかのフィールド
1-9 bytes (VarInt)	トランザクション数	このブロックに含まれるトランザクション数
可変	トランザクション	トランザクションのリスト

Table 1. The structure of a block

### ブロックサイズについて

- 最大1MB
- 1つのブロックには平均1,900トランザクションが含まれる

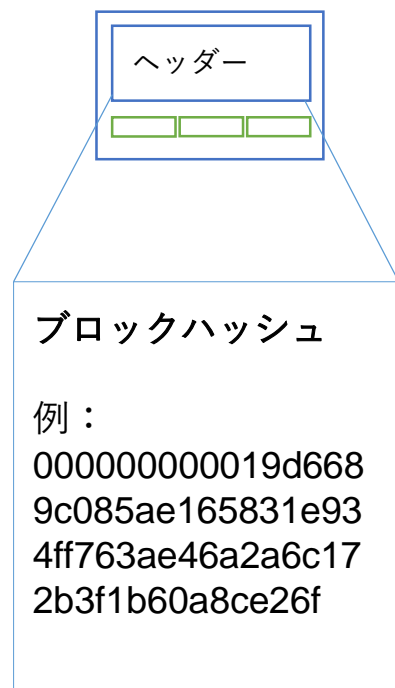


## ブロックヘッダの構造

Size	Field	Description
4 bytes	Version	ソフトウェア、プロトコルバージョン情報
32 bytes	Previous Block Hash	前ブロックのブロックハッシュ
32 bytes	Merkle Root	ブロックの全トランザクションに対するマークルツリーのルートハッシュ
4 bytes	Timestamp	ブロックの生成時刻 (Unix時間)
4 bytes	Difficulty Target	ブロック生成時のPoWのdifficulty
4 bytes	Nonce	ナンス値

Table 2. The structure of the block header

## ブロック識別子：ブロックヘッダハッシュとブロック高



- ブロックを指定するとき、ブロック識別子としてブロックハッシュを使用する。
  - 各ブロックをユニークに識別したい。
  - ブロック高で指定できそうだが、ブロックチェーンは枝分かれ（フォーク）している場合があり、高さだけでは一意にブロックを指定できない。
- ブロックハッシュ値はブロックヘッダに対してSHA256を用いて2回ハッシュ化した値（32バイト）
- ブロック内にその値は保存されていない。
  - つまり、ブロックチェーン内にどのブロックハッシュがどのブロックを表現しているのかを紐づける情報は保存されていない。
  - ブロックハッシュ値は対応するブロックのブロックヘッダから計算にすることでのみ導ける。
  - クライアントソフト内で、検索高速化のために別DBで管理されていることはあり得る。
- 前のブロックのブロックハッシュ値はヘッダ内に含まれている。
- マイニングの際に計算しているハッシュ値と同じもの

# Genesisブロック

- ビットコインブロックチェーンの最初のブロックは2009年1月4日に作られた
- これをgenesisブロックと呼ぶ
- ブロックハッシュ：000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
- genesisブロックは他のノードから取得する必要はなく，クライアントソフト（bitcoin core）にハードコードされている。
  - <https://github.com/bitcoin/bitcoin/blob/master/src/kernel/chainparams.cpp#L61>
- Coinbaseトランザクションにメッセージが含まれている。
  - Coinbaseトランザクションとはマイナーにマイニング報酬を支払うトランザクション。
  - マイナーが自由にデータを入れられる領域がある。
    - 詳細は「マイニングとコンセンサス」で。
  - 「The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.」という文字列が入っている
    - 英国タイムズ紙の2009年1月3日の見出し。
    - Genesisブロックがこれ以前には存在しなかったことの証明としてサトシ・ナカモトが入れた。
- genesisブロックのTimestampの値は2009-01-04 03:15

## まとめ

---

- ブロックチェーンの構造
  - 前ブロックのハッシュ値がブロックに含まれることで改ざん不可能なチェーンを構成している
- マークルツリー
  - マークルルートがブロックヘッダに保管
  - SPVノードで使用.
    - マークルパスから、あるトランザクションがブロックに含まれてるかの検証ができる。
- ブロック内に入っているデータ
  - ブロックヘッダとトランザクションリスト
- ブロック識別子
  - そのブロックハッシュ値. ブロックヘッダから求められるがその識別子そのものはどこにも保存されていない。
- Genesisブロック
  - ブロックチェーンの最初のブロックのこと

- 本スライドの著作権は、東京大学ブロックチェーンイノベーション寄付講座に帰属しています。 自己の学習用途以外の使用、無断転載・改変等は禁止します。
- ただし、
  - Mastering Bitcoin 2nd edition  
[https://github.com/bitcoinbook/bitcoinbook/tree/second\\_edition\\_print3](https://github.com/bitcoinbook/bitcoinbook/tree/second_edition_print3)
  - ビットコインとブロックチェーン --- 暗号通貨を支える技術 （著：Andreas M. Antonopoulos  
訳：今井崇也，鳩貝 淳一郎）
- を使用した部分の使用のみ，CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0/deed.ja> のライセンスを適用するものとします。