Ethereumのローカル開発環境

マルチシグ方式のPeg-inを作る

Bitcoin–Ethereumのブリッジ試作をしている。

ということは Ethereumアプリが必要ということになる。

まったく記憶にないので、一から調べる。

- foundry をインストール Ethereumのローカル開発環境
  - Ethereumのローカル開発環境#2826e1
    - `anvil` でローカルEVM環境を立ち上げ、 `cast` で JSON-RPC を使わず操作する
      - `anvil` は複数ノードで同期する機能はない
        - geth は clique が deprecacted になったのでできなさそう
        - besu がよいか Private networks - Besu documentation
    - スマコンの開発は `forge` を使うとよい
      - `--rpc-url http://localhost:8545` とすると `anvil` にアクセスしてチェーンを読み込む
      - `forge test` は単体テスト用にテストを実行してデプロイはしない前提
      - `forge deploy` は `--broadcast` をつけると `--rpc-url` で指定したチェーンにデプロイする
        - `--rpc-url` は現在 `--fork-url` の方が正式名称
  - Rust 開発は Alloy を使う
    - Ethereumのローカル開発環境#28267e

よく使うコマンド

- `cast` コマンド
  - Ethereumのローカル開発環境#9c1199
- `forge script` でデプロイ
  - Ethereumのローカル開発環境#3669d0
  - コントラクトのサイズが `24576` byte を超えると警告が出るし続けても失敗した。
    - EIP-170 による 0x6000 超過が理由だそうで mainnet でも同様。ライブラリにするなどの対処がいるらしい。

Ethereumの基本

今はPoWではなくPoS

数年(毎年？)でハードフォークしている

2022年のParisがThe Mergeという大きい変更でPoSになった

どの設定か忘れたが、どのハードフォーク以降にするかで機能が使えたり使えなかったりしたことがあったように思う

1ブロックが提案されるまでの時間を**スロット**と呼び、mainnet は12秒

バリデータの割り当てやファイナリティの判定を行う「区切り」を**エポック**と呼び、mainnet は32スロット(=6.4分)

トランザクションに状態がある

Finalized になるのは2エポック通過後。経過ではなく通過なので、承認されたスロットを含むエポックが終わってさらに2エポック経過したあと。さらに「ジャスティファイ」しないと Finalized にならないらしい。

Safe は「だいたい大丈夫だろう」という状態で、おおよその取引は Finalized までだと長すぎるのでこちらを使うらしい。トランザクションが承認されたブロックの次のブロックが承認されたらSafeになる。なので1〜2スロット程度の時間になる。

今は web3.js や ethers.js などはもうない。viem というのが有名なようだ。hardhat はまだある。

イーサリアムのローカル開発のセットアップ

開発言語としては、C、TypeScript、Rust の 3つが載っている。

foundry

foundry - Ethereum Development Framework

"Rust"と書いてあるが、このツール群が Rust で書かれているだけで、Rust 用のライブラリではなかった。

```
$ curl -L https://foundry.paradigm.xyz | bash
```

`$HOME/.foundry` にファイルが置かれるので、別の場所にしたかったら移動してシンボリックリンクを張っておけばよいだろう。

インストールはコマンド1つ。

```
$ foundryup
```

`ls`

```
$ ls ~/.foundry/bin
anvil  cast  chisel  forge  foundryup
```

`forge` helps you build, test, debug, deploy and verify smart contracts.

`anvil` is your local development node that complies with the Ethereum JSON-RPC spec.

`cast` is your swiss army knife for interacting with onchain applications from the command line.

`chisel` : Fast Solidity REPL for rapid prototyping and debugging

REPL : 対話型実行環境

anvilregtest的なノード

anvil

リファレンス

`anvil --fork-url` で URL を指定するとそのチェーンを fork したような設定をしてくれる。
forkした場合、デフォルトの設定はforkしたデータに色々上書きされる。Chain IDは上書きされ、nonce などもコピーされる。chainIDをanvilのものにしたければ `--chain-id 31337` で上書きする。実行時にコピーされるのではなく、必要に応じてURLから取得するとのこと。コントラクトもデプロイしていなくても呼び出せたりする。
anvil のデフォルトアカウントは使用できず、コピーしたサイトの設定が使用される。

`anvil` だけでデフォルト設定で起動できる。
ETH がある秘密鍵と EOA のセットは起動ログに出力されている。
デフォルトのChain ID は `31337`

デフォルトのアドレスと秘密鍵

```
Available Accounts
==================

(0) 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000.000000000000000000 ETH)
(1) 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000.000000000000000000 ETH)
(2) 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000.000000000000000000 ETH)
(3) 0x90F79bf6EB2c4f870365E785982E1f101E93b906 (10000.000000000000000000 ETH)
(4) 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65 (10000.000000000000000000 ETH)
(5) 0x9965507D1a55bcC2695C58ba16FB37d819B0A4dc (10000.000000000000000000 ETH)
(6) 0x976EA74026E726554dB657fA54763abd0C3a0aa9 (10000.000000000000000000 ETH)
(7) 0x14dC79964da2C08b23698B3D3cc7Ca32193d9955 (10000.000000000000000000 ETH)
```

```
(8) 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f (10000.000000000000000000 ETH)
(9) 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720 (10000.000000000000000000 ETH)

Private Keys
==================

(0) 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
(1) 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d
(2) 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a
(3) 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6
(4) 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a
(5) 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba
(6) 0x92db14e403b83dfe3df233f83dfa3a0d7096f21ca9b0d6d6b8d88b2b4ec1564e
(7) 0x4bbbf85ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcdbf7cbf4356
(8) 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97
(9) 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

Wallet
==================
Mnemonic:          test test test test test test test test test test test junk
Derivation path:   m/44'/60'/0'/0/
```

stateの保存

オプションに `--state <PATH or JSON file>` をつけると、anvil の終了時にステートを保存し、anvil の起動時にそのステートを読み込むという動作をする。

mining mode

デフォルトではトランザクションを投げると自動でブロック生成。

`--block-time <second>` で定期的にブロック生成。

`--no-mining` でマイニングしない。

regtest のように好きなタイミングでマイニングすることはできないようだ。

finalizedを縮める

PoS になってブロックの確定が safe と finalized

cast

https://getfoundry.sh/cast/reference/cast

いろいろコマンド。

たくさんあるのでサイト参照。

balance

```
$ cast block-number
0
  $ cast balance 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
10000000000000000000

$ cast mktx 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 --value 1000 --private-key
0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
0x02f86a827a6980018477359401825208947099797 0c51812dc3a010c7d01b50e0d17dc79c88203e88
0c001a0baec6b6502c9dbf7f9b15c03411a911654cc8e8cca5a5830b6b84c05be194deea0702535e5c8
457c534185d601e5caed30a98e7932c6a1249a2e6601a2bba61d81

$ cast publish 0x02f86a827a6980018477359401825208947099797 0c51812dc3a010c7d01b5
0e0d17dc79c88203e880c001a0baec6b6502c9dbf7f9b15c03411a911654cc8e8cca5a5830b6b84c
05be194deea0702535e5c8457c534185d601e5caed30a98e7932c6a1249a2e6601a2bba61d81
{"status":"0x1","cumulativeGasUsed":"0x5208","logs":
[],"logsBloom":"0x00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000","type":"0x2","transactionHash":"0xfb2a711f8c07b8e
17870ee5c2798258f8259d2e527d413ef650ee76762e05895","transactionIndex":"0x0","blockH
ash":"0xaf072b9519bef807e99434b46e3d71360fc43cd6f51441069134837ea6526c83","blockNum
ber":"0x1","gasUsed":"0x5208","effectiveGasPrice":"0x3b9aca01","blobGasPrice":"0x1"
,"from":"0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266","to":"0x70997970c51812dc3a010c
7d01b50e0d17dc79c8","contractAddress":null}

$ cast block-number
1
$ cast balance 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
999999978999999978000
$ cast balance 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
10000000000000000001000
```

forge

https://getfoundry.sh/forge/overview

Solidityのコンパイルやデプロイの環境を作る。

`solc` をバージョンにあわせてインストールしたり lint も走らせたりするので楽。

`init`

サンプルの `Counter` スマコンが入ったプロジェクト作成。

`install`

`git` リポジトリから取得するようだ。 `git submodule` として追加されるようなので自分のリポジトリはそこまで気にしなくてよさそう。

```
$ forge install OpenZeppelin/openzeppelin-contracts@v5.4.0
```

`clone`

etherscan の APIキー(アカウントを作る必要あり)とスマコンのアドレスを与えるとforge を使ってデプロイできるプロジェクトを作ってくれる。

と思って Alloy の Getting Startedにあったコントラクトを指定したのだが `forge build` で失敗してしまった。

`pragma solidity ^0.4.18` だったので `forge build --use 0.4.18` などでバージョンは指定できるが lint はバージョンが指定できないようでコンパイル後に失敗していた。 `foundry.toml` でlintを無視すれば通った。

clone

```
$ forge clone 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2 -e
SZGNBBE7IJR28KE3DZMI2P42SUFCCMB3QA
Downloading the source code of 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 from
Etherscan...
Initializing /mnt/wsl/home2/ueno/Ethereum/forge...
Installing forge-std in /mnt/wsl/home2/ueno/Ethereum/forge/lib/forge-std (url:
Some("https://github.com/foundry-rs/forge-std"), tag: None)
Cloning into '/mnt/wsl/home2/ueno/Ethereum/forge/lib/forge-std'...
remote: Enumerating objects: 2386, done.
remote: Counting objects: 100% (1225/1225), done.
remote: Compressing objects: 100% (239/239), done.
remote: Total 2386 (delta 1098), reused 1015 (delta 983), pack-reused 1161 (from 3)
Receiving objects: 100% (2386/2386), 838.87 KiB | 4.82 MiB/s, done.
Resolving deltas: 100% (1585/1585), done.
```

```
    Installed forge-std tag=v1.11.0@8e40513d678f392f398620b3ef2b418648b33e89
    Initialized forge project
Collecting the creation information of 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
from Etherscan...
[⠒] Compiling...
[⠢] Compiling 1 files with Solc 0.4.19
[⠁] Solc 0.4.19 finished in 8.41ms
Compiler run successful!
```

create

`forge create` でデプロイする。以下のオプションは必須のようだ。

- スマコン名
  - ここでは `WETH9`
    - `WETH9` はスマコンの名前(solファイルに `contract WETH9` と書いている)。
- デプロイする鍵
  - ここでは `--private-key`

`--broadcast` を付けないと実際のデプロイは行わない。

deploy

```
$ forge create WETH9 --private-key
0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 --broadcast
[⠒] Compiling...
No files changed, compilation skipped
Deployer: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Deployed to: 0xCf7Ed3AccA5a467e9e704C703E8D87F634fB0Fc9
Transaction hash:
0xab493350eb507b3b9c82de21b3770ac7412f2a169884f7edee80a4b7cfb80f8e
```

script

Solidityファイルを直接チェーン上で実行するコマンド。主な用途はコントラクトのデプロイで、複数のコントラクトがあるなど単純にデプロイできないときは `script/` の下にデプロイ用の sol ファイルを作って実行するとよい。

```
$ forge script script/Deploy.s.sol --rpc-url http://127.0.0.1:8545 --broadcast --
private-key 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
```

ドキュメントには `--rpc-url` とあるのでデプロイするチェーンのRPC URLを示すオプションのように見えるが、

rpc-url

```
  -f, --fork-url <URL>
          Fetch state over a remote endpoint instead of starting from an empty
state.

          If you want to fetch state from a specific block number, see --fork-
block-number.

          [aliases: --rpc-url]
```

デプロイしたコントラクトのアドレスは実行したディレクトリの `broadcast/Deploy.s.sol/<チェーンID>/run-latest.json` を参照すると良い。 `contractAddress` に現れるがコントラクトが複数ある場合は `contractName` で探してから参照すると良い。
パスの `31337` は anvil の ChainIDなので、他のチェーンなら別の値になる。なおJSONファイルは履歴を残していくのでテスト環境ではファイルが増えすぎることがあるので気にしておくとよい。

また、デプロイするアドレスは計算することも可能である。ただ複数コントラクト展開の場合は `nonce` がわかりづらい

`compute-address`

```
$ cast compute-address 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 --nonce 2
Computed Address: 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0
```

`test`

https://getfoundry.sh/forge/tests/overview

https://getfoundry.sh/forge/tests/forge-std

`./test/*.sol` に書いたテストを実行する。

`import {console2} from "forge-std/console2.sol";` などで `console2.log()` と書いておくと `forge test -vv` で出力される。

よくわかっていないが、

- `setUp()` は各テスト `test_XXX()` 実行前に呼び出される
- それぞれの `msg.sender` は同じだが `setUp()` でスマコンを `new` した先の `msg.sender` は一致しない
  - Solidity がスマコン内で別のスマコンを呼び出すと別スマコンでの `msg.sender` はスマコンのアドレスになるそうだからそれかもしれない(`msg.origin` は元の EOA になる)。
- revertすることを期待する `vm.expectRevert()` はコントラクト呼び出しの直前が安全。他

の `vm.xxx()` が挟まっているだけで効果がないことがあった。

よく使う anvil 向けコマンド

anvil の基本データ(アカウント、秘密鍵、Chain ID、base fee)

Ethereumのローカル開発環境#9c1052

(0) 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266

0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

(1) 0x70997970C51812dc3A010C7d01b50e0d17dc79C8

0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

(2) 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC

0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

注意

`cast` のオプション `--rpc-url` を省略すると `http://127.0.0.1:8545` を指定した扱いになる。

残高

```
$ cast balance 0xアカウント
```

nonce

```
$ cast nonce 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
```

ETH送金

```
$ cast send 0xアカウント --value 100wei --private-key 0x秘密鍵 [--rpc-url http://〜]
```

```
$ cast send 0x702415923f9bEf4859E6948C2Bf3295F367e3645 --value 1ether --private-key
0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
```

TX情報

```
$ cast tx 0xTXハッシュ
```

スマコンの関数呼び出し

```
$ cast send <アドレス>"シグネイチャ(引数1,引数2,...)"  引数1 引数2 ... [--value xether]
--private-key <KEY>
$ cast call <アドレス> "シグネイチャ(引数1,引数2,...)"  引数1 引数2 ... [--from 送信者]
```

```
$ cast call 0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0 "balanceOf(address)"
0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
```

Rustデベロッパーのためのイーサリアム

- paradigmxyz/reth: Modular, contributor-friendly and blazing-fast implementation of the Ethereum protocol, in Rust
  - フルノード
- bluealloy/revm: Rust implementation of the Ethereum Virtual Machine.
  - EVM
- Alloy
  - おそらくクライアントライブラリ
  - ethers-rs を吸収した？
- sigp/lighthouse: Ethereum consensus client in Rust
  - コンセンサスクライアント
- a16z/helios: A fast, secure, and portable multichain light client for Ethereum
  - クライアントライブラリ

AIによる比較

- Grok
- ChatGPT

3つしか挙げていないが、lighthouse がちょっと違いそうなので Alloy か Helios のどちらか。
スターは Helios の方が多いが Alloy は ethers-rs を書き直したもの
マルチシグウォレットが目的であればモバイルは考慮不要だろうか。
ユーザは MetaMask を使ってもらうだけでなくモバイルアプリに組み込む想定もいるかもしれない。が、管理者が使うのだからPC前提で良いか。
Alloy を使いつつ、同期が取りたいなら Helios と組み合わせる、という感じか。

Alloy

Ithaca という Web3インフラ会社が開発しているツール。元Paradigmの幹部の人が作った会社らしい。ケンカ別れして作った会社というわけではなく投資したり協力関係にあるらしい(Grok)。
他にも Reth や Foundry を提供しているので、そこそこ Ethereum で影響力がありそうだ。2024年設立だが Paradigm のオープンソースツールを引き継いだからだろう。

alloy · Rust Interface for Ethereum

ウォレットサンプル

Signers vs Ethereum Wallet – alloy

https://github.com/alloy-rs/examples?tab=readme-ov-file

https://github.com/alloy-rs/examples/blob/main/examples/wallets/examples/ethereum_wallet.rs

Gas計算

送金可能な最大額は `残高 - ガス代最大額` になっている。実際に使われるガス代はそれよりも小さいことが多いので、残額をゼロにするのは難しい。

https://deepwiki.com/search/basefee_1f107cd2-2571-4807-8b23-126762601a05?mode=fast

rust-alloy

```
        let est = provider.estimate_eip1559_fees().await?;
        println!("estimate max_fee_per_gas: {}", est.max_fee_per_gas);
        println!("estimate max_priority_fee_per_gas: {}",
    est.max_priority_fee_per_gas);
```

`baseFee` はトランザクションに直接には現れないのでいらないように思うが、EVM がチェックする残額は `effectiveGasPrice = min(maxFeePerGas, baseFee + maxPriorityFeePerGas)` ではなく `max()` の方になる。それを満たさないとそもそも送金が開始できない。

rust-alloy

```
        let block_number = provider.get_block_number().await?;
        let fee_history = provider.get_fee_history(
            block_number,
            BlockNumberOrTag::Number(block_number),
            &[alloy_provider::utils::EIP1559_FEE_ESTIMATION_REWARD_PERCENTILE]).await?;
        let base_fee = fee_history.next_block_base_fee().unwrap_or(0);
        println!("base_fee: {}", base_fee);
```

Grok

`effectiveGasPrice = min(maxFeePerGas, baseFee + maxPriorityFeePerGas)`

Gas に単位は特にないらしい

`maxPriorityFeePerGas` などの単位は Gwei

`10^9` Gwei = `10^18` Wei (`1` ETH = `10^18` wei)

最大ETH = `gasLimit × maxFeePerGas × 10^{-18}` ETH

最小ETH = `gasUsed × baseFee × 10^{-18} ETH` （priorityFee= `0` の場合）。

実際は `gasUsed × (baseFee + priorityFee) × 10^{-18}` ETH。

Alloy で `with_max_fee_per_gas()` と `with_max_priority_fee_per_gas()` を指定すると `6,325,187,862,000` weiだった。

base_fee: `301,199,421`

estimate max_fee_per_gas: `688,318,155`

estimate max_priority_fee_per_gas: `1`

実txから

effectiveGasPrice: `301,199,422` = `base_fee + max_priority_fee_per_gas` **一致！**

gasLimit: `21,000`

maxFeePerGas : `688,318,155`

maxPriorityFeePerGas: `1`

gasUsed x effectiveGasPrice = `6,325,187,862,000` **一致！**

困ったこと

2025/12/02: 不安定

1回目から失敗することがしばしばあった。

2回目に呼び出すと失敗する。

Exampleレベルの単発で動かす範囲では問題が無かったので実装を進めていたのだが、スマートコントラクトを call ではなく send するシーンが出てきて上に書いたような現象が起きることが分かった。
今の作りは、マルチシグ管理しているEthereumのトークンを他のユーザに転送するために担当者が他のノードから署名を集めて転送するスマートコントラクトを呼び出す、としている。
P2Pで署名を集めていて、署名が来たら取りあえずスマートコントラクトを呼び出している。
今は 3-of-3 にしているので 1名から届いただけで呼び出しても revertし、もう1名から届いてようやく成功する。

まず、1回目に成功すべきところで失敗することがしばしばあった。
これはログを出している間は起きなかったため、起きるのに気付いたのはログのON/OFFができるようにしたあとだった。

最初に気付いたのは 2回目の失敗、つまり 1回目に revert → 成功、2回目に revert → 失敗、という流れである。

こちらはログを出していても発生するので、現在(2025/12/02)はこちらの調査を進めている。
"失敗"というのは、Alloy API でトランザクションを展開する際は `.send().await` した結果を .get_receipt()なり .watch() なりして結果が戻ってくるまで待つのだが、それが戻ってこない。 `eth_blockNumber` と `eth_getTransactionReceipt` を繰り返しているように見える。 `.get_receipt()` でタイムアウトさせるようにすると戻ってくるが、再送しても結果は変わらない。
anvil と geth で試したがどちらも同じ結果。geth で "nonce too high" になったが nonceがずれている？ 今度は "nonce too low" になった。

https://chatgpt.com/share/692e3aa1-b1a0-8001-b7f2-01e7cffe5030

- nonce の基礎知識
  - 0から始まる
  - `eth_getTransactionCount` でnonceを取得
    - これは次のトランザクションで使う値
  - アカウント(EOA、コントラクト)ごとに所持する
  - トランザクションがブロックに取り込まれたときにインクリメントされる。
    - `eth_getTransactionCount` は次に展開予定の nonce ということになる
    - call 系はブロックに取り込まれないのでインクリメントされない。
    - send系は revert もブロックに取り込まれるのでインクリメントする

Hyperledger Besu

anvil は開発には良いが複数ノード接続したりはできないのでちょっとした時差などを考慮した環境を作ることができない。
Ethereumでは Clique が deprecated になって geth でプライベートチェーンを作ることができなさそうなので Hyperledger Besu を使うことにした。

環境設定

バイナリは docker コンテナを使うので設定だけする必要がある。

anvil はトランザクションを投げると自動でブロックを作るし、一定間隔でブロックを生成する機能もある。besu でブロックを生成するには validator が必要。自分が validator になることもできるので 1ノードで運用が可能である。
validator になるには鍵を持つ必要がある。その鍵を起動オプションに指定するだけでなくブロックのgenesis情報に組み込む。その両方を行うコマンドが用意されている。

```
$ besu operator generate-blockchain-config --config-file=/config/qbftConfigFile.json
```

```
--to=/config/generate --private-key-file-name=key
```

今回はコンセンサスアルゴリズムQBFTを使った。特にこだわりはない。JSONファイルに genesis の情報とそれ以外の情報を入れてコマンドを実行すると出力先ディレクトリ(`--to`)を作って genesis.json と鍵ファイル(key, key.pub) を生成する(ディレクトリが存在したらエラーになるので実行前に削除する)。

qbftConfigFile.json

```
{
  "genesis": {
    "config": {
      "chainId": 31337,
      "homesteadBlock" : 0,
      "daoForkBlock" : 0,
      "eip150Block" : 0,
      "eip155Block" : 0,
      "byzantiumBlock" : 0,
      "constantinopleBlock" : 0,
      "petersburgBlock" : 0,
      "istanbulBlock" : 0,
      "muirGlacierBlock" : 0,
      "berlinBlock" : 0,
      "londonBlock" : 0,
      "arrowGlacierBlock" : 0,
      "grayGlacierBlock" : 0,
      "mergeNetsplitBlock" : 0,
      "shanghaiTime" : 0,
      "cancunTime" : 0,
      "qbft" : {
        "epochlength" : 30000,
        "blockperiodseconds" : 5,
        "requesttimeoutseconds" : 10,
        "zeroBaseFee" : true
      }
    },
    "nonce": "0x0",
    "timestamp": "0x58ee40ba",
    "gasLimit": "0x1fffffffffffff",
    "difficulty": "0x1",
    "mixHash":
```

```
  "0x63746963616c2062797a616e74696e65206661756c7420746f6c6572616e6365",
    "coinbase": "0x0000000000000000000000000000000000000000",
    "alloc": {
      "0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266": {
        "privateKey":
"0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80",
        "comment": "anvil account#0",
        "balance": "9000000000000000000000"
      }
    }
  },
  "blockchain": {
    "nodes": {
      "generate": true,
      "count": 1
    }
  }
}
```

大半がどこかから持ってきた値である。 `alloc` は anvil のデフォルトと同じにした。

QBFT - Besu documentation

面倒だったのがコントラクトとEVMの不一致だった。
OpenZeppelin v5系を使っているのだが、QBFTのサイトに書いてある設定を使うと EVMバージョンが古いため OpenZeppelin のコードが生成した命令(`mcopy`)がなくてデプロイに失敗した。
OpenZeppelin v4では使われていないようなのでバージョンを下げようとしたが、そうするとMultiSigで使っていた `Account.sol` がなかったのでやめた。
では Solidity のバージョンを下げると生成するコマンドも対応する EVMに合わせるということだったので下げようとしたが、OpenZeppelin v5 が 8.0.24 以降を指定していたのでそれもできず。
OpenZeppelin のv5.0.0 のような細かい指定をすればもしかしたら、という提案もあったが面倒なので、それよりは Hyperledger Besu の EVMバージョンを上げる方向に変更した。
最終的に効いたのは `"cancunTime": 0` だった。 `shanghaiTime` も必要だったかもしれない。
空ブロック自動生成のためには `qbft.zeroBaseFee` も必要だった。

生成された genesis.json と key を使いたいディレクトリに置き、起動オプションに与える。

`docker-compose`

```
  besu:
    image: hyperledger/besu:latest
    container_name: besu
    ports:
      - "8545:8545"
    volumes:
      - ./besu:/config
    entrypoint: ["besu"]
    command:
      - --logging=DEBUG
      - --data-path=/data
      - --genesis-file=/config/genesis.json
      - --p2p-port=30303
      - --host-allowlist=*
      - --rpc-http-enabled
      - --rpc-http-host=0.0.0.0
      - --rpc-http-port=8545
      - --rpc-http-api=ETH,NET,WEB3,ADMIN,DEBUG
      - --network-id=31337
      - --node-private-key-file=/config/key
    networks:
      shared:
        aliases:
          - ethereum.local
```

anvilとの違い

Hyperledger Besu というか QBFT Proof of Authority、PoAである。そのためブロックの生成＝ファイナライズになる。Ethereumなどの PoS では「まあまあ安全」という Safe や「ほぼ確実」という Finalized という概念がある。ブロックやブロック番号の取得でブロック番号を指定する代わりにタグとして `Safe` や `Finalized` を指定できるのだが PoA では使用できない。