

# Solidity 入門 by yum

At 2022/11/17; Bcali Group MTG

はじめに

目的

今回やること

今回使うツール・ライブラリなど

事前準備

1. スマートコントラクトとは

1.1. スマートコントラクトの概念

元々の概念：

Ethereum のスマートコントラクト：

1.2. Ethereum 上のスマートコントラクトのライフサイクル

2. Solidity 入門

2.1. Solidity とは

2.2. 環境構築

GitHub のテンプレートレポジトリをクローンする

クローンしたレポジトリを開発用コンテナ内で開く

Dockerfile の説明

3. はじめてのコントラクト作成

3.1. Solidity を記述

3.2. ローカルテストネットワークにデプロイしてみる

4. 文法の簡単な説明

4.1 Pragma

4.2. コメント

4.3. 変数

4.3.1. 状態変数 と メモリ変数

4.3.2. 変数の型

4.4. 関数

4.5. Contract

5. 演習

5.1. カウンターコントラクト

5.2. Crowdfunding コントラクトを作ってみる

5.2.1. コンストラクタを実装する

5.2.2. コントラクトに送金可能にする

5.2.3. コントラクトに預けられた資金を引き出す関数を実装する

5.2.4. Hardhat ローカルネットワークにデプロイする

5.2.5. Sepolia テストネットワークにデプロイしてみる

5.2.5. おまけ課題（時間があればやる）

6. 参考文献

## はじめに

### 目的

- 一般に「スマートコントラクト」と呼ばれるものが、どのように作られ、動いているのか実際に手を動かして作ってみることで理解する
  - あくまで流れだけを簡単に触ってみる
- 軽くしかやらないので興味を持った場合は色々と調べてみてください！

### 今回やること

- Solidity を使ってクラウドファンディングコントラクトを作ってみる
  - 目標金額を設定してデプロイ
  - コントラクトに ETH を送信
  - 目標金額に達成したらコントラクトから ETH を取り出せる

## 今回使うツール・ライブラリなど

- Hardhat と周辺ツール
  - スマートコントラクトを作るためのフレームワーク & CLI でいい感じに開発進めれる
- Metamask
  - Chrome の拡張機能として使えるウォレット
- Docker & Visual Studio Code
  - Dev Container でクールに開発環境をセットアップしよう！

### ▼ フロントエンド（今回は直接には触らない）

- Next.js + TypeScript: ウェブアプリのフロントエンドフレームワーク。React を用いて表現する。
- ethers.js: ブラウザでブロックチェーンノードと通信するためのクライアントライブラリ。同様のものに web3.js がある。

## 事前準備

以下の準備を事前をお願いします。

- VSCode と Docker、そして devcontainer 拡張機能の準備
  - 前回にインストールした Docker と git はそのまま使います。
  - 拡張機能: <https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack>
  - cf. <https://code.visualstudio.com/docs/devcontainers/containers>
  - 宗教上の理由で VSCode を使いたくない方は、いい感じに Docker コンテナ内で開発できる環境を用意してください（この場合、開発環境に関するサポートは難しいです。）
- Metamask のセットアップ
  - Metamask とは？
    - Chrome などので使えるブラウザ拡張機能型のウォレットです。
    - 開発時のデバッグなどによく使われます。
    - スマホ版も存在するらしい。

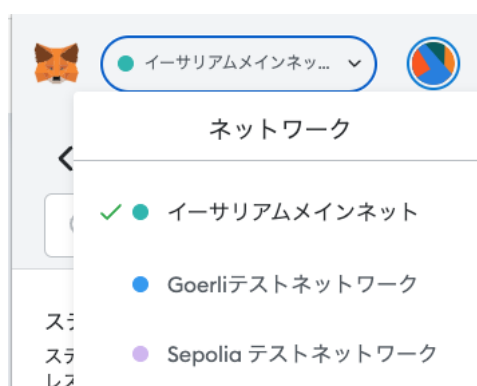
### ▼ セットアップ手順

#### 1. Metamask をインストール

- <https://metamask.io/download/>
- cf. <https://coincheck.com/ja/article/472>

#### 2. Metamask でテストネットを表示

- 右上のアイコンをクリック → 設定 → 高度な設定 から「テストネットワークを表示」をオンにする
- ネットワークリストのドロップダウンをクリックして、以下の画像のようにテストネットワークが表示されれば成功



3. **(任意・推奨)** メインのアドレスと開発用のアドレスはセキュリティ上の理由や利便性から分けた方が良いでしょう。2つ目の新しいアカウントを生成して、そちらを今回使用することを推奨します。
  - 右上のアイコンをクリック → 「アカウントを作成」
  - アカウント名は自由に決める（通常第三者に漏れることはない）
  - **注意：** 開発用アカウントを作成した場合、以下の Sepolia faucet の受け取り先を含め、この回では今後はそのアカウントを使い続けること。
- Sepolia テストネットワークのネイティブトークンである SepoliaETH を入手しておく
  - faucet と呼ばれる無料でトークンが貰えるサービスから入手する
  - Sepolia FaucETH: <https://faucet.sepolia.dev/>
    - 先ほど作った Metamask のアドレスを入力し、Captcha を解いた後で、「Request funds」をクリック
    - 数分後に Metamask に入金が反映されれば完了
- ▼ なぜ faucet ?
 

通常の本番 Ethereum ネットワークのトークンは取引所などでも流通しているためそこで購入可能だが、テストネットワークのトークンはそれ自体に価値があるとはみなされていないので、取引所などでは大規模に流通していない。そのためテストネットのトークンは誰かから無料で貰う必要がある。faucet はその代表的な手段。

  - ちなみに yum はテストネットのバリデータノードを自分で建てて、それにステーキングして貰った報酬を faucet で配布する、みたいなシステムを作りたいなあなどと思っています。興味ある方はぜひ声をかけてください。一緒にやりましょう！

## 1. スマートコントラクトとは

※ 以下、この章は kekeho さん作の資料から全て引用、yum によって少し改変。

### 1.1. スマートコントラクトの概念

#### 元々の概念：

もともとの概念は、1996年にNick Szaboが提唱した：

<https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/>

<https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/>

- デジタル革命で、ネットワークで繋がったコンピュータと暗号技術を用いて行われるデジタル形式の、新しい形の契約(コントラクト)が出来るんじゃないか？という話
- 当事者が他の約束を実行する手順まで含んだ、デジタル形式で規定された一連の約束
- (オフラインの)例: 自動販売機
  - 硬貨を入れると、価格に応じて自動的に商品を提供する
  - 物理的ロックをかけることで、攻撃者から保護している
- 後述する現在のスマートコントラクトの概念とは乖離があることに注意
- 第三者へのトラストレスなエスクロー取引を実装できるのでは？とも。

#### Ethereum のスマートコントラクト：

Ethereum によって導入されたスマートコントラクトはブロックチェーン上で動くプログラム

- ざっくりとしたイメージ

[https://viewer.diagrams.net/?border=0&tags=%7B%7D&highlight=0000ff&edit=\\_blank&layers=1&nav=1&title=Untitled%20Diagram.drawio&open=R3Zddb9MwFIZ%2FTSS4YMpH04%2FLrh1wARJSEWYXXuwl1pw4ctwm2a%2FHxuwkjpO1BQgDm8rn9Vf9vMfHrRNs0uoDA3nymUJEHN%2BFIRNsHd8Pw4X4lELdCHPfbYSYYdhIXifs8BNSoh62xxAVxkBOKeE4N8WIZhmKuKEBxmhpDnugxNw1BzGyhF0EiK1%2Bx5Anjbr0F53%2BEeE40Tt781XTkwI9WJ2kSACKZU8KbpxgwyjITsutNohldppLM%2B%2F9RG%2F7xRjK%2BCkTct97Kg4w4mzlrcvIYY%2FfXunVjKAslcHviY0epSUdV%2FBaw2D0X0GkVzPdYlRMsEc7XIQyd5SuC%2B0hKdERJ5oPtCM79RcGR8Q41iAXRMcZ0LjVE5Q%2B4s%2BVE0ezGtxiTRDNEWc1WKImjBTgFWGtXHZ80tJSc%2BqhdKAypC4XbiDKBqK4xIM%2FWmm%2FwhR331ISGcWUUnHfOQMRv6qe2Q25irNzE17BGX1EG0ooE0pGMySJYkIGEIAwl8EKsRHKKYZQbjPqlunBZJ7cZoT%2FqWc0HW5Z4UFH2VwLUuypEhAUeDltMKEJECw%2BrYf3MngKtThtup3bmsdVZjfqhVluzdLRN0kGdSGIQhaNX9ghzgO3bMIHa%2BcHLAY8WPVwLa351844p%2FWGCKA44P5dcdMVTi8oVgcpM0eLzTTJ5gN8qI5pprVfzwGC1kVYTVYqQFgLfScY%2B2xfyHt7IfKqADh%2F1sBLPR%2FvQTYD5xFPxYY8snDqx9r4F4Pd8%2BFspzI6x4Tbz4CZX4pKMvjTM56609KnBfsebHi%2FFk0i%2BmrCyB8E759pXe3%2F3PL%2Ff1%2BqGWCwe0O7UR2R9wKLuxWaiSR50TacS8asWx8ZSArhH2YZrpPbNV2v1I7L3qjxgrwT1gkwu4PYfN2dv%2Bqg5sf](https://viewer.diagrams.net/?border=0&tags=%7B%7D&highlight=0000ff&edit=_blank&layers=1&nav=1&title=Untitled%20Diagram.drawio&open=R3Zddb9MwFIZ%2FTSS4YMpH04%2FLrh1wARJSEWYXXuwl1pw4ctwm2a%2FHxuwkjpO1BQgDm8rn9Vf9vMfHrRNs0uoDA3nymUJEHN%2BFIRNsHd8Pw4X4lELdCHPfbYSYYdhIXifs8BNSoh62xxAVxkBOKeE4N8WIZhmKuKEBxmhpDnugxNw1BzGyhF0EiK1%2Bx5Anjbr0F53%2BEeE40Tt781XTkwI9WJ2kSACKZU8KbpxgwyjITsutNohldppLM%2B%2F9RG%2F7xRjK%2BCkTct97Kg4w4mzlrcvIYY%2FfXunVjKAslcHviY0epSUdV%2FBaw2D0X0GkVzPdYlRMsEc7XIQyd5SuC%2B0hKdERJ5oPtCM79RcGR8Q41iAXRMcZ0LjVE5Q%2B4s%2BVE0ezGtxiTRDNEWc1WKImjBTgFWGtXHZ80tJSc%2BqhdKAypC4XbiDKBqK4xIM%2FWmm%2FwhR331ISGcWUUnHfOQMRv6qe2Q25irNzE17BGX1EG0ooE0pGMySJYkIGEIAwl8EKsRHKKYZQbjPqlunBZJ7cZoT%2FqWc0HW5Z4UFH2VwLUuypEhAUeDltMKEJECw%2BrYf3MngKtThtup3bmsdVZjfqhVluzdLRN0kGdSGIQhaNX9ghzgO3bMIHa%2BcHLAY8WPVwLa351844p%2FWGCKA44P5dcdMVTi8oVgcpM0eLzTTJ5gN8qI5pprVfzwGC1kVYTVYqQFgLfScY%2B2xfyHt7IfKqADh%2F1sBLPR%2FvQTYD5xFPxYY8snDqx9r4F4Pd8%2BFspzI6x4Tbz4CZX4pKMvjTM56609KnBfsebHi%2FFk0i%2BmrCyB8E759pXe3%2F3PL%2Ff1%2BqGWCwe0O7UR2R9wKLuxWaiSR50TacS8asWx8ZSArhH2YZrpPbNV2v1I7L3qjxgrwT1gkwu4PYfN2dv%2Bqg5sf)

- ステートマシンを思い浮かべる
  - ステートマシン: 以下の流れを繰り返すことによって複雑な処理を進めていくことを可能にした機械
    1. あるステート（状態）がある
    2. 外部からあるイベントがエミットされる
    3. 該当ステートが更新され、それによって処理が切り替わる
- ブロックチェーンをメモリ（ステート）として見立てる
  - プログラムのバイトコードもブロックチェーン上に格納されている
- トランザクションによって関数呼び出しを行い、状態遷移を引き起こす。
  - コードはそれぞれのノードの上にあるEVM（イーサリアムに特化したVM）で実行される

## 1.2. Ethereum 上のスマートコントラクトのライフサイクル

1. Solidity 等のスマートコントラクト用言語で実装することが多い

```
pragma solidity ^0.8.10;

contract SampleContract {
    uint256 internal state = 0;

    constructor() {}

    function add(uint256 x) external {
        state = this.state + x;
    }
}
```

2. デプロイする前に、EVM用のバイトコードにコンパイルする

```
$ solc --bin sample.sol
===== test.sol:SampleContract =====
Binary:
60806040526000805534801561001457600080fd5b50610186806100246000396000f3fe608060405234801561001057600080fd5b5060004361061002b57600035
```

3. チェーンにデプロイ
  - アドレス `0x0` 宛のトランザクションを送信することでデプロイできる
4. EOA（秘密鍵に紐づくアカウント）からのトランザクションによってメソッドが呼び出され、状態遷移が起こる
  - 実行が失敗したらロールバックされる。アトミック性を持つ。
  - コントラクトアカウントはそもそも秘密鍵を持たないため、コントラクトからトランザクションを開始することはできない。
5. （コントラクト内から `SELFDESTRUCT` 命令を実行することで、コントラクトの削除を行うことができる）

- よっぽどことがない限り実行してはいけない
- 仮にコントラクトに Ether がある状態で実行すると Ether ごと消滅する（burned される）ので注意
- そもそもブロックチェーン上には存在の履歴が残るのであまり意味がない

## 2. Solidity 入門

### 2.1. Solidity とは

- Ethereum（あるいはEVM互換チェーン）で稼働するスマートコントラクトを記述するためのDSL（ドメイン固有言語）
- Gavin Woodらを中心に開発された
- オブジェクト指向言語におけるクラスに似た形でコントラクトを記述できる
- 言語の仕様の的にはチューリング完全である
  - 実際はEVM上で gas による制約を受ける

（※ kekeho さん作の資料から引用、yum によって少し改変。）

### 2.2. 環境構築

前回、せっかく Docker と git について学んだので、今回はその2つを用いて開発環境を爆速で構築することにします。VSCode には公式で Dev Containers という拡張機能が配布されており、これを導入すると、Docker コンテナ内の開発環境に直接繋いで VSCode で開発ができるようになります。

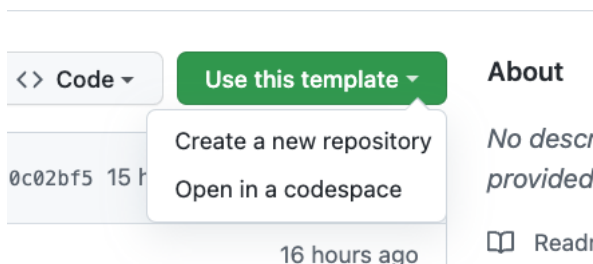
1. GitHub のテンプレートリポジトリをクローンする
2. クローンしたリポジトリを開発用コンテナ内で開く

#### GitHub のテンプレートリポジトリをクローンする

1. テンプレートリポジトリにアクセス

`https://github.com/ymsg19/solidity-handson`

2. 「Use this template」 から 「Create a new repository」 を選択



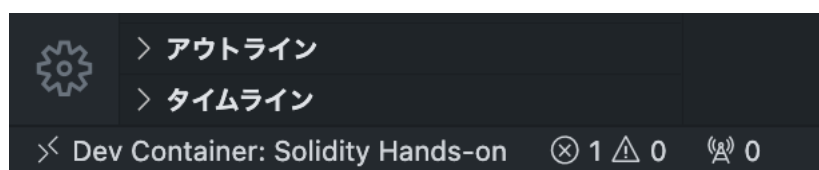
3. リポジトリ作成画面が出てくるので、いつもの流れでリポジトリを作成する
  - 任意：「Include all branches」にチェックを入れると僕が作った完成品の実装があるブランチ “full-implementation” もコピーできる。
4. クローンしたいディレクトリに移動した上で、`git clone <さっき作ったリポジトリのurl>` を実行する

#### クローンしたリポジトリを開発用コンテナ内で開く

1. クローンしたプロジェクトフォルダを VSCode で開く
2. `⌘ + Shift + P` or `Ctrl + Shift + P` を押してコマンドパレットを開く
3. `dev` と入力して予測に出てきた `Dev Containers: Open Folder in Container...` を選択



- エラーなくコンテナに接続できれば完了。初回起動時は少し時間がかかります。左下の表示が以下の画像のように「Dev Container: ...」になっていれば成功です。



- 以下のコマンドを実行して、今回利用するツール・ライブラリ類をインストールする

```
$ yarn
```

### ▼ Dockerfile の説明

Dockerfile はコンテナのセットアップ内容について記述するファイルです。専用のDSLを用いて記述します。

```
ARG VARIANT="focal"
# Microsoft が用意している devcontainers 用の Docker イメージを利用
# 今回は base:focal のタグを指定しているので、Ubuntu 20.04 LTS の環境を利用します。
FROM mcr.microsoft.com/vscode/devcontainers/base:${VARIANT}

# apt のリポジトリを更新
RUN apt update && apt upgrade -y
# curl, git, gpg をインストール
RUN apt install curl git gnupg2

# Node.js v18 のインストール
RUN curl -fsSL https://deb.nodesource.com/setup_18.x | \
  sudo -E bash - && \
  apt install -y nodejs

# 今回は Node.js のパッケージマネージャに yarn を用いるので、yarn を有効化
RUN corepack enable yarn && \
  corepack prepare yarn@stable --activate
```

## 3. はじめてのコントラクト作成

- Hello, Worldコントラクトを作ります

### 3.1. Solidity を記述

- `contracts/HelloWorld.sol` を作成し、以下のコードを書く

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract HelloWorld {
    function sayHello() external pure returns (string memory) {
        return "Hello, World!";
    }
}
```

- コンパイル

```
$ yarn hardhat compile
# 出力
Generating typings for: x artifacts in dir: typechain-types for target: ethers-v5
Successfully generated x typings!
Compiled x Solidity file successfully
Done in xx.
```

### 3.2. ローカルテストネットワークにデプロイしてみる

- Hardhat Console を使って Hardhat のローカルテストネットにデプロイ & 実行

```
$ yarn hardhat console // hardhat のコンソールモードを起動。ノードを立ててるので少し時間がかかる。
Compiled 1 Solidity file successfully
Welcome to Node.js v18.12.1.
Type ".help" for more information.

> const Contract = await ethers.getContractFactory("HelloWorld") // コントラクトの情報を読み込み
undefined
> const contract = await Contract.deploy() // コントラクトのデプロイTXを送信
undefined
> contract.address // コントラクトのアドレスを出力
'0x5FbDB2315678afecb367f032d93F642f64180aa3'
> await contract.sayHello() // sayHello 関数を呼び出し
'Hello, World!'
```

- ここでの学びポイント
  - `contract` キーワードでコントラクトを書くことができる
    - 一つのファイルに複数個のコントラクトを書くことも可能
    - コントラクトはオブジェクト指向プログラミングのようなカプセル化や再利用をすることもできる
  - `function` キーワードでメソッド定義ができる
    - `return` で値を返すことができる(返さなくても良い)
  - Hardhat & JS を用いて、コントラクトのデプロイ・呼び出しができる

## 4. 文法の簡単な説明

### 4.1 Pragma

- コンパイラのバージョンを指定する。
- 基本的にファイルの先頭行に記述する。
  - 指定しないとコンパイルエラーが発生する。
- `^` を付けることで、メジャーバージョンが同一な範囲内での最新版を指定することができる
  - セマンティックバージョンing：ソフトウェアのバージョンing手法の1つ。雑に説明すると、アップデートをメジャーアップデート、マイナーアップデート、パッチアップデートの3つに分類して、それぞれのバージョンを `MAJOR.MINOR.PATCH` で繋ぐ。アップデートの度にそれに対応する番号をインクリメントし、それよりも下位のバージョンの種類を 0 にする。
    - e.g. `v1.2.3` のメジャーバージョンは 1、マイナーバージョンは 2、パッチバージョンは 3 となる

- メジャーアップデートの場合、次のバージョンは通常 `v2.0.0` になる
- マイナーアップデートの場合、次のバージョンは通常 `v1.3.0` になる
- パッチアップデートの場合、次のバージョンは通常 `v1.2.4` になる
- cf. <https://semver.org/lang/ja/>
- 色々と派生形があり、現場の開発スタイルによって様々なカスタマイズされることも多々ある
- Solidity は npm のセマンティックバージョンング規約を準用している
- コード例

```
pragma solidity 0.8.17; // バージョン0.8.17のコンパイラでコンパイルしなさい
pragma solidity ^0.8.0; // バージョン0.8.0か、0.8.0よりも新しく、1.0.0 よりも小さいバージョンのコンパイラでコンパイルしなさい
```

## 4.2. コメント

- `//` 以降は無視されるので、コメントを書くことができる
- 例

```
uint256 count = 0 // カウンター変数
```

## 4.3. 変数

### 4.3.1. 状態変数 と メモリ変数

- 状態変数 (State variables) : チェーン上 (storage) に記録される。  
型 `visibility` 変数名 (= 初期値) で宣言
  - コントラクト内で宣言した変数は、State variablesになる。
  - いくつかの Visibility を設定できる
    - `public`: コントラクトの外から直接アクセスできる。(自動的に同名のゲッターが生成される)
    - `external`: 外部からの呼び出しのみ可能。ただし、`this`を使用することで内部から外部的に呼び出すことは可能。(gas が安くなるという噂は聞くが要検証)
    - `internal`: コントラクト内のメソッドと、その継承コントラクト内からのみ使用できる。デフォルト。
    - `private`: `internal`のもっと厳しいバージョン。継承コントラクトからも使用できない。
    - `constant`: 定数

```
contract Sample {
    uint256 public a = 10;
    uint256 internal b = 10;
    uint256 c = 10; // デフォルトは internal
    uint256 private d = 10;
    uint256 constant e = 10;
}
```

- メモリ変数 (Memory variables): 一時的な変数。メソッドの実行終了時に破棄される。
  - メソッド内で宣言した変数は、Memory variablesになる。
  - 一時変数に状態変数を使うのは無駄なので、なるべくメモリ変数を使う
    - 状態変数の操作には関数実行時に gas (手数料) がさらにかかる

```
contract Sample {
    uint256 stateVal = 10; // 状態変数

    function hoge() {
        uint256 memoryVal = 20; // メモリ変数
    }
}
```



### 4.3.2. 変数の型

<https://docs.soliditylang.org/en/latest/types.html>

- 値型

- 常に値として渡される
- `bool`: 真偽値

```
bool hoge = false;
bool fuga = true;

!hoge // 論理否定

# ANDとORは短絡評価
hoge && fuga // AND
hoge || fuga // OR

hoge == fuga // 等価
hoge != fuga // 不等価
```

- `uint`: 符号なし整数
  - `uint8`, `uint16`, `uint24`, ..., `uint256` と, 8つつ用意されている

```
uint256 hoge = 123;
uint8 fuga = 10;

hoge + fuga // 足し算
```

- `int`: 符号付き整数
  - `int8`, `int16`, `int24`, ..., `int256` と, 8つつ用意されている

```
int256 hoge = -123;
int112 fuga = 21;

hoge + fuga // 足し算
hoge - fuga // 引き算
hoge * fuga // 掛け算
hoge / fuga // 割り算

hoge % fuga // mod (割り算の余り)
hoge ** fuga // 累乗

// 比較
hoge < fuga
hoge <= fuga
hoge > fuga
hoge >= fuga
hoge == fuga
hoge != fuga

// ビット演算子
hoge & fuga // AND
hoge | fuga // OR
hoge ^ fuga // XOR
~hoge // NOT

// シフト
hoge << 1 // 1bit 左シフト
hoge >> 1 // 1bit 右シフト
```

- `byte`:
  - ビット列の操作ができる
  - `bytes1` から `bytes32` まで定義されている

```
bytes32 hoge = bytes32(uint256(123))
bytes32 fuga = bytes32(uint256(456))

// インデックスアクセス
hoge[1]
```

```
// 比較
hoge < fuga
hoge <= fuga
hoge > fuga
hoge >= fuga
hoge == fuga
hoge != fuga

// ビット演算子
hoge & fuga // AND
hoge | fuga // OR
hoge ^ fuga // XOR
~hoge // NOT

// シフト
hoge << 1 // 1bit 左シフト
hoge >> 1 // 1bit 右シフト
```

- **address**

- 20byte (Ethereumのアドレスと同じ幅)

```
address yum1 = 0xe38e47c5912effFDf04537CB1eE8E73CD4C4e5Fd;
address payable yum2 = 0xe38e47c5912effFDf04537CB1eE8E73CD4C4e5Fd;

yum1.balance // 残高

// payableを付けると, transferメソッドを使えるようになる
yum2.transfer(10) // 10weiをコントラクトからyum2に送金
// (ちなみに 1wei = 1e-18 ETH だったはず)
```

- **enum**

- 列挙型

```
enum Choices {
    Shimekiri,
    Harakiri
}

contract Sample {
    Choices choice = Choices.Shimekiri;
}
```

- 参照型

- ▼ 参照型のデータロケーション

全ての参照型には、データロケーションというアノテーションがついている。これらは参照型のデータがどこに保存されているのかを示す。

- memory
  - 一時的な保存領域
  - 指定できる型に条件なし
  - ミュータブル
- calldata
  - 一時的な保存領域
  - 配列、構造体、マッピングに適用可能
  - イミュータブル
- storage
  - 永続的な保存領域 = 状態変数

- array

- 配列. **型名[(サイズ)] 変数名** で定義

```
uint256[] counts; // 動的配列。サイズは可変。

counts.push(1);

peopleNum = 100;
counts = new uint256[](peopleNum) // 実行時にサイズを決定することができる

uint256[3] xyz; // 固定配列。サイズは固定。
xyz = [3, 3, 4];
```

#### ◦ string

- 文字列。ダブルクォーテーションで囲う。

```
string name = "yum"; // ASCIIのみ
string name = unicode"曾我 悠真" // UTF-8。明示的に宣言が必要。
```

#### ◦ mapping

- 他の言語のハッシュテーブル, 辞書などに相当. `mapping(keyの型名 => valueの型名)`

```
mapping (uint256 => string) nametable;

nametable[1] = 'yum'; // 代入
nametable[1]; // 'yum' // アクセス
```

#### ◦ struct

- 構造体。複数の変数を纏めるもの。

```
enum Gakubu {
    Kankyo,
    Sogo,
    Kango
}

struct Student {
    int8 enterYear;
    string name;
    Gakubu gakubu;
}

Student yum = Student(21, "Yuma", Gakubu.Kankyo);
yum.enterYear // 22
yum.name // "Yuma"
yum.gakubu = Gakubu.Sogo // 代入
```

## 4.4. 関数

- メソッド(関数: function)を定義できる。
- `function 関数名 (引数) visibility modifier [returns (<return types>)]` の形で書く
  - Visibility
    - 変数に同じ。
    - デフォルトも `internal`
  - オプションの修飾子
    - 以下は予約済みの修飾子の例
      - `pure`: 純粋関数。状態変数へのアクセスをしない。外部からの呼び出しに関してガス代がかからない。
      - `view`: 定数関数。状態変数を書き換えない。外部からの呼び出しに関してガス代がかからない。
      - `payable`: ETHを受け取ることができるメソッド。
    - 自分で修飾子を実装することもできる
      - cf. <https://docs.soliditylang.org/en/latest/contracts.html#function-modifiers>

- 返り値の型
  - `returns (int256)` みたいな感じで指定.

```
contract Sample {

  function add(int256 a, int256 b) external pure returns (int256) {
    return a + b;
  }

}
```

## 4.5. Contract

- Ethereumにデプロイするプログラムの基本単位
- 変数とメソッドを束ねたもの
  - オブジェクト指向言語におけるクラスのようなもの。カプセル化や再利用をすることもできる。
  - 変数とメソッドの他にも、event、modifier なども含めることができるが、今回のハンズオンの範囲外なので、気になる方は調べてみてください

```
contract SampleContract {
  int256 internal stateVar = 10;

  function currentState() external view returns (int256) {
    return stateVar;
  }

  function add(int256 a, int256 b) external pure returns (int256) {
    return a + b;
  }
}
```

## 5. 演習

### 5.1. カウンターコントラクト

- メソッドを呼び出すたびにカウントが1増える、カウンターを実装してみよう
- ちなみに本番環境でインクリメント処理する場合は致命的なバグが発生しやすいので、十分に検証されたライブラリを利用するのが良い。
  - cf. <https://docs.openzeppelin.com/contracts/4.x/api/utils#Counters>
- `contracts/counter.sol` を新しく作成して以下のように編集。

```
pragma solidity ^0.8.0;

contract Counter {
  uint256 public count = 0;

  function increment() external {
    count += 1;
  }
}
```

#### ▼ コンパイル & 実行スクリプト

```
$ yarn hardhat console
Compiled 1 Solidity file successfully
Welcome to Node.js v18.12.1.
Type ".help" for more information.

> const Contract = await ethers.getContractFactory("Counter")
undefined
> const contract = await Contract.deploy()
undefined
> contract.address
'0x5FbDB2315678afecb367f032d93F642f64180aa3'
```

```

> await contract.count() // count を取得 (public な変数なので getter が自動で追加される)
BigNumber { value: "0" }
> await contract.increment() // increment を実行
{ ... } // tx
> await contract.count()
BigNumber { value: "1" }
> await contract.increment()
{ ... } // tx
> await contract.increment()
{ ... } // tx
> await contract.count()
BigNumber { value: "3" }

```

## 5.2. Crowdfunding コントラクトを作ってみる

ここからは予め用意されている `contracts/Crowdfunding.sol` を編集していく。

実装する機能は以下の通り。

- デプロイ時に目標金額（やメタ情報）を設定機能
- コントラクトでのETHの受け取り機能
- 目標金額に達成したらコントラクトから ETH を取り出せる機能

### 5.2.1. コンストラクタを実装する

Solidity におけるコンストラクタは、コントラクトのデプロイ時に呼び出される。ここで変数の初期化などを行うことができる。

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract Crowdfunding {
    bool private accomplished = false; // 目標完遂かどうかのフラグ
    uint private target; // 目標金額
    string private title; // タイトル
    string private description; // 説明
    address payable private toAddr; // 引き出し先アドレス

    constructor(
        string memory _title,
        string memory _description,
        uint _target,
        address payable _toAddr
    ) payable {
        title = _title;
        description = _description;
        target = _target;
        toAddr = _toAddr;
        accomplished = false;
    }
}

```

### 5.2.2. コントラクトに送金可能にする

`payable` 修飾子を付けた関数を実装することで関数呼び出しの際に ETH を一緒に送金できるようになる。通常、単なる送金 tx（コントラクトの関数呼び出し tx ではないアカウント間の資金移動tx）で ETH を受け取るには `receive()` を用いる。

```

pragma solidity ^0.8.9;

contract Crowdfunding {
    bool private accomplished = false;
    uint private target;
    string private title;
    string private description;
    address payable private toAddr;

    // ...中略...

    receive() external payable {}
}

```

### 5.2.3. コントラクトに預けられた資金を引き出す関数を実装する

現在のコントラクトの残高が `target` を満たしている場合、コントラクトに預けられた資金を `toAddr` に送金する関数を実装する。

- 関数内に `require(bool condition, string reason)` を記述することによって、`condition` が `false` であった場合、tx は revert される。
  - アトミック性を持つので revert された場合、tx 開始以降の状態変数の変更は取り消される。
  - revert された場合、gas は消費されたもの以外は返却される。
- コントラクト自身の残高は `address(this).balance` で参照できる。
- `address.transfer(uint amount)` で送金できる。

```
pragma solidity ^0.8.9;

contract Crowdfunding {
    // ...中略...

    receive() external payable {}

    function withdraw() public {
        require(target <= address(this).balance, "insufficient balance to withdraw");
        toAddr.transfer(address(this).balance);
    }
}
```

## 5.2.4. Hardhat ローカルネットワークにデプロイする

今回はデプロイスクリプトを使ってデプロイしてみます。

- `scripts/deploy.ts` と `scripts/send.ts` を開き、修正する。

### ▼ `scripts/deploy.ts`

```
import { ethers } from "hardhat";

async function main() {
    const [deployer] = await ethers.getSigners();

    console.log("Deploying contracts with the account:", deployer.address);
    console.log("Account balance:", (await deployer.getBalance()).toString());

    const Crowdfunding = await ethers.getContractFactory("Crowdfunding");

    // ここから修正
    const crowdfunding = await Crowdfunding.deploy(
        "Test Funds", // title: タイトル
        "Test Desc", // description: 説明
        ethers.utils.parseEther("1"), // target: 目標金額
        "0x..." // toAddr: 回収資金の送金先アドレス
    );
    // ここまで修正

    console.log(`The contract was successfully deployed to ${crowdfunding.address}`);
}

main().catch((error) => {
    console.error(error);
    process.exitCode = 1;
});
```

### ▼ `scripts/send.ts`

```
import { ethers } from "hardhat";

async function main() {
    const [owner] = await ethers.getSigners();

    const tx = await owner.sendTransaction({
        to: "0x...", // ここをさっき作った Metamask の自分のアドレスにする
        value: ethers.utils.parseEther("100.0")
    });

    await tx.wait().then(() => {
        console.log("100.0 ETH has been sent.");
    }).catch(e => console.error(e))
}
```

```
// We recommend this pattern to be able to use async/await everywhere
// and properly handle errors.
main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

2. 以下を実行する。このシェルのプロセスはテストする間はずっと起動させ続けること。

```
$ yarn hardhat node # ネットワーク立ち上げ
Started HTTP and WebSocket JSON-RPC server at http://0.0.0.0:8545/

Accounts
=====
...略...
```

3. 次に、別のシェルを立ち上げて以下を実行する。コントラクトアドレスはどこかに控えておく。

```
# コントラクトのデプロイ
$ yarn hardhat run scripts/deploy.ts --network localhost
Compiled x Solidity file successfully
Deploying contracts with the account: 0x...
Account balance: ...
The contract was successfully deployed to <デプロイ先コントラクトアドレス>
Done in 22.21s.

# テスト用の ETH を自分の手元に送信
$ yarn hardhat run scripts/sendETH.ts --network localhost
100.0 ETH has been sent.
Done in 16.81s.

# フロントエンドサーバの起動
$ yarn dev
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

ブラウザで <http://localhost:3000/> を開くと、フロントエンドが表示される。



4. きちんと想定した動作がされるか色々と試してみよう！

### 5.2.5. Sepolia テストネットワークにデプロイしてみる

今までデプロイしていた先はローカルネットワークであり、このままでは他の人からアクセスできない。一方で Ethereum の本番ネットワークに投入するのは実際のお金を扱うので、開発途中の不完全なコントラクトのテストに使うのは大変危険。そんな時に使うのがテストネットワーク。本番ネットワークに似た環境でテストできる。

1. `sample.env` をコピーして、`.env` にファイル名を改名する
2. `.env` に SepoliaETH を保有するアカウントの秘密鍵を記入
  - 秘密鍵の取り扱いには厳重に注意！
  - `.env` は git の差分に含まない。（`.gitignore` に追加済）
3. 以下のコマンドを実行

```
$ yarn hardhat run scripts/deploy.ts --network sepolia
Deploying contracts with the account: 0x...
Account balance: ...
The contract was successfully deployed to <デプロイ先コントラクトアドレス>
Done in 22.21s.
```



### 5.2.5. おまけ課題（時間があればやる）


- このままでは誰でも withdraw を実行できてしまうので、withdraw は送金先のアドレスの持ち主のみ実行可能にする。
  - `msg.sender` で tx 送信者のアドレスを取得できる
- このままでは完了後にも ETH が送金できてしまうので、
  1. withdraw されたタイミングで `accomplished` フラグを `true` に設定するように実装する。
  2. `accomplished` が `false` である場合のみ送金を受け付けるように実装する。
- 新たに状態変数を加えて、最低送金限度額・最高送金限度額を実装してみる
- 新たに状態変数を加えて、各アカウントの寄付額を記録し外部から参照できるようにする

## 6. 参考文献

- SolidityとEthereumによる実践スマートコントラクト開発

#### SolidityとEthereumによる実践スマートコントラクト開発

Ebook Storeで電子版を購入:価格2,992円 本書は、ブロックチェーン開発をこれから始めるエンジニア向けに、スマートコントラクト開発をわかりやすく解説します。EVM（イーサリアム仮想マシン）上で実行可能なSolidity言語と、人気の高いフレームワークTruffle Suiteを使って、独自のスマートコントラクト

 <https://www.oreilly.co.jp/books/9784873119342/>

#### コントラクト開発


Truffle Suiteを用いた開発の基礎からデプロイまで



- kekeho さん作の資料（ありがとうございました!!）
- 本格的かつ実践的に学んでみたい方は CryptoZombies もおすすめです

#### #1 Solidity Tutorial & Ethereum Blockchain Programming Course | CryptoZombies

CryptoZombies is The Most Popular, Interactive Solidity Tutorial That Will Help You Learn Blockchain Programming by Building Your Own Fun Game with Zombies - Master Blockchain Development with Web3, Infura, Metamask & Ethereum Smart Contracts and Become a Blockchain Developer in Record

 <https://cryptozombies.io/>

Learn to Code  
Blockchain DApps By  
Building Simple Games

Learn • Build • Earn

