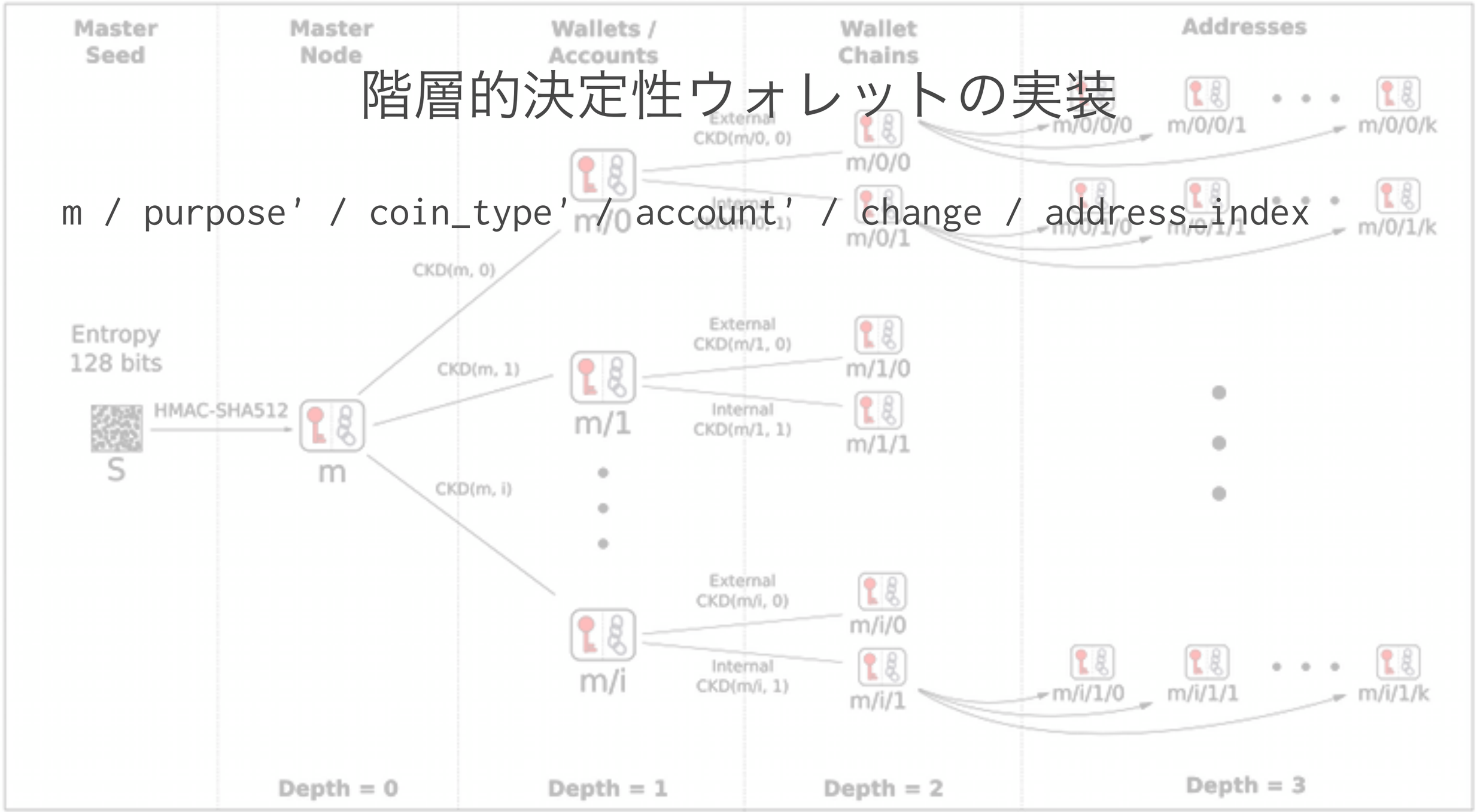
階層的決定性ウォレットの実装

```
m / purpose' / coin_type' / account' / change / address_index
```



Abount Me

- Twitter @kogane5513
- 2018/10〜　マネーフォワードフィナンシャル
  - 仮想通貨取引所の新規開設に向けて奮闘中！
- 〜2018/9
  - （株）SmartDrive
    - 新規事業開発
  - freee 株式会社（フリー）
    - 決済、認証基盤チーム

本日の内容

- 階層的決定性ウォレットとは(BIP32,44)
- 階層的決定性ウォレットの実装
- MultiSig Addressの実装

使用言語：Golang

使用ライブラリー：btcsuite/btcutil

```
import (
 "github.com/btcsuite/btcd/chaincfg"
 "github.com/btcsuite/btcutil/hdkeychain"
 "github.com/btcsuite/btcd/txscript"
)
```

階層的決定性ウォレットとは(BIP32,BIP44)

マスターなるシード値から、m/i/0/kのような階層構造的に秘密鍵を生成・管理できる標準規格です。
（Hierarchical Deterministic Wallet = HD Wallet）
通常のウォレットでは、使用済みの秘密鍵と公開鍵のペアを定期的に、すべてバックアップしておく必要がありましたが、BIP32で標準化されたプロトコルを利用することで、マスターシードさえ保存されていれば、そこから派生する秘密鍵を持っていなくてもインデックスを指定することで、異なるシステムからいつでも再利用することができます。
ビットコインコアにもv0.13.0から導入され、正式にサポートされるようになりました。

つまり、HD Walletの規格で生成された秘密鍵であれば、Seedさえあれば復元、再利用可能となり運用が楽になるというもの。
1個の乱数からツリー構造的に多数アドレス（秘密鍵、公開鍵）を生成できます。
下記は、階層化されたパスのイメージです。

- BIP32 Path level:マスター / アカウント' / 支払い or お釣り / アドレス

- BIP44 Path level: マスター / 仕様(BIP)' / 通貨' / アカウント' / 支払い or お釣り / アドレス

階層的決定性ウォレットとは(BIP32,BIP44)

階層別による仕様は下記になります。

m: マスター鍵
purpose: 目的階層 44に設定された定数
coin*type*: コインの種類階層。通貨毎にスペースが決められている
account: 使用目的階層。寄付目的 / 貯蓄目的 / 共通経費 など使用するユーザー側で決めることができる
change: 受取階層。外部送金者（*External*）からの受取りが*0* / 自身（*Internal*）のトランザクションからのおつりの受取りが*1*
*address*index: アドレス階層。インデックス値が振られる

Coin types
Path Examples

シードの生成

基本的にHD Walletの実装では、hdkeychainパッケージに用意されています。
シードの生成もこのパッケージにあるGenerateSeedファンクションを使えばよくて、
シードの長さはBIP-0032で推奨されているバイト単位の長さ(256bits：128bits〜512bits)も定数として定義されています

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

seed, err := hdkeychain.GenerateSeed(hdkeychain.RecommendedSeedLen)
if err != nil {
   return err
}
```

マスター鍵

シードを利用してマスター鍵を生成します。
また、テストネットの場合、第二引数に渡すものはchaincfgパッケージに用意されているchaincfg.TestNet3Paramsをセットしてください。

```
import (
  "github.com/btcsuite/btcd/chaincfg"
    "github.com/btcsuite/btcutil/hdkeychain"
)

master, err := hdkeychain.NewMaster(seed, &chaincfg.MainNetParams)
if err != nil {
   return err
}
```

Purpose

ここではどの仕様(BIP)に従っているかを示す。また、強化鍵導出のためインデックス値に2^{31}を加算します。

強化鍵導出？
1つでもビットコインアドレスの秘密鍵が流出してしまうと、同じブランチの秘密鍵を全て算出されてしまう脆弱性があり、強化導出関数をセットすると、この脆弱性を回避できるらしい。

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

// m/44'
// BIP-49(P2SHでネストされたP2WPKHアドレス導出する場合は、49??)
purpose, err := master.Child(44 + hdkeychain.HardenedKeyStart)
if err != nil {
    return err
}
```

CoinType

ここではどの通貨に従っているかを示す。また、ここでも強化鍵導出のためインデックス値に2^{31}を加算します。

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

// m/44'/0'
coinType, err := purpose.Child(0 + hdkeychain.HardenedKeyStart)
if err != nil {
   return err
}
```

Account

アカウントを示すインデックスを指定する。また、強化鍵導出のためインデックス値に2^{31}を加算する

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

// m/44'/0'/0'
account, err := coinType.Child(1 + hdkeychain.HardenedKeyStart)
if err != nil {
   return err
}
```

エクスターナル/インターナル

ここでは支払い=0、おつり用=1のいづれかを示す

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

// m/44'/0'/0'/0
change, err := account.Child(0)
if err != nil {
   return err
}
```

Addess Index

ここが末端となりここで生成された公開鍵を使ってアドレスを生成する

```
import (
    "github.com/btcsuite/btcutil/hdkeychain"
)

// m/44'/0'/0'/0/0
addressIndex, err := change.Child(0)
if err != nil {
   return err
}
addressIndex.ECPubKey() // 公開鍵
```

MultiSig Addressの実装

```
// AddressPubKeyを使ってredeemScriptを作成する
addressPubKeys := []*btcutil.AddressPubKey{addressPubKey1, addressPubKey2}
// 2 of 2
redeemScript, err := txscript.MultiSigScript(addressPubKeys, len(addressPubKeys))
if err != nil {
    return err
}
// redeemScriptよりMultiSig adddressを作成する
ad, err := btcutil.NewAddressScriptHash(redeemScript, &chaincfg.MainNetParams)
if err != nil {
    return err
}
addr := ad.EncodeAddress()
```

MultiSig Addressの実装

```
func MultiSigScript(pubkeys []*btcutil.AddressPubKey, nrequired int) ([]byte, error) {
    if len(pubkeys) < nrequired {
        str := fmt.Sprintf("unable to generate multisig script with "+
            "%d required signatures when there are only %d public "+
            "keys available", nrequired, len(pubkeys))
        return nil, scriptError(ErrTooManyRequiredSigs, str)
    }

    builder := NewScriptBuilder().AddInt64(int64(nrequired))
    for _, key := range pubkeys {
        builder.AddData(key.ScriptAddress())
    }
    builder.AddInt64(int64(len(pubkeys)))
    builder.AddOp(OP_CHECKMULTISIG)

    return builder.Script()
}
```