BDKについてわかったこと

バージョン

以前は大きい「bdk」の中にいろいろとプログラムが入っていた(V0)が、V1 と呼ばれるこれを書いている現在では Rustのクレートとして複数に分かれることになった。

https://github.com/bitcoindevkit#stable

Stable

Project	Lead Maintainer	Secondary Maintainer
bdk_wallet	ValuedMammal	oleonardolima
bdk_core	evanlinjin	oleonardolima
bdk_chain	evanlinjin	oleonardolima
bdk_esplora	oleonardolima	luisschwab
bdk_electrum	evanlinjin	oleonardolima
bdk_bitcoind_rpc	evanlinjin	ValuedMammal
bdk_testenv	tvpeter	luisschwab
esplora-client	oleonardolima	luisschwab
electrum-client	oleonardolima	luisschwab
bdk-ffi	thunderbiscuit	reez
bdk-jvm	thunderbiscuit	ltoroD
bdk-android	thunderbiscuit	reez
bdk-swift	reez	thunderbiscuit
book-of-bdk	thunderbiscuit	reez

GitHubの主要なリポジトリは `bdk` と `bdk_wallet` 。

注意すべき点は、**現在は "bdk" という Rustクレートは使われていない**ということだ。docs.rs にはこのように残っているのだが、v0.30.2 を最後に DEPRECATED である。クレートとしては使用しなくてよいが現在でもリポジトリ名 "bdk" で各プロジェクトが開発しているので非常に紛らわしい。

実装としては基本的には `bdk_wallet` を使い、必要に応じて各プロジェクトのクレートを参照

することになる。

同期について

同期にかかる時間

同期にはフル同期と部分同期がある。ウォレットの起動時にフル同期し、運用中はウォレットのアドレスに関係あるところだけ部分同期するというやり方が一般的である。
といっても、フル同期 `full_scan()` と部分同期 `sync()` は内部で呼び出しているメソッドは同じである。 `full_scan()` はHDウォレットのインデックスにしたがってアドレスでスキャンしていき、未使用すなわち送受金がないアドレスが指定したインデックス以上続いたらそこでスキャンを打ち切る。 `sync()` は引数の `SyncRequest` で与えられた内容だけスキャンする。

部分同期を簡単に使うなら、 `start_sync_with_revealed_spks()` で同期要求 `SyncRequestBuilder` を作ってもらい、バックエンドの `sync()` に与える。 `bdk_electrum` のsync場合、Electrum Serverにまとめて処理要求する `batch_size` とウォレットへのINPUTトランザクション手数料を計算するためにトランザクションを取得するかどうかの `fetch_prev_txouts` がある。手数料計算に関係するなら `fetch_prev_txouts=true` が必要そうだが、そうではない。これはウォレットのINPUTトランザクションの手数料に関するもので、今のウォレットが作るトランザクションの手数料とは関係ない。

同期は時間がかかる。UTXO数もそうだがウォレットのアドレス数にも影響される。千アドレスで1〜2秒くらいだったが 3千アドレスで6秒くらいかかった。UTXO数は減らすことができるがHDウォレットのインデックス値は減ることがないので `SyncRequestBuilder` に伝えるアドレス数をなんとかしないといけない。

- 1ユーザ1アカウントにして極力アドレスを増やさない
- 同期対象のアドレスを制限する

BDK bookでもアドレス数が多い場合には後者を勧めている。
Additional Considerations — Really Big Wallets

同期対象を制限

同期対象を制限する場合は `sync()` を使う。引数で与える `SyncRequest` を制限することでスキャン対象を制御できる。
SyncRequestBuilder in bdk_chain::spk_client - Rust

- spks(), spks_with_indexes()
  - アドレス、正確には scriptPubKey を登録
  - "with_indexes()" はインデックスとセットでscriptPubKeyを登録する。

インデックスはHDウォレットのインデックスでなくてもよい。TxGraphに保存されるわけではなくウォレット側のデータになるようだ。

インデックス値は `KeychainKind` の値、つまり `External` か `Internal` とセットである。

deepwiki

txids()

TXIDを登録

outpoints()

OutPointを登録

unused_spks_from_indexer()

未使用アドレスだけを集めて内部で spks_with_indexes() を呼ぶ便利関数

deepwiki

chain_tip()

同期開始するブロックを指定する。指定しない場合はブロックデータは更新せずトランザクションデータの更新のみ行う。

deepwiki

ウォレットに紐づいているかどうかは考慮せず同期して内部のグラフ構造(TxGraph)を更新する。

同期対象を求めるヘルパー関数はこうなっている。自分で管理するのであれば `revealed_spks_from_indexer()` や `expected_spk_txids()` 、ここには出ていないが `unused_spks_from_indexer()` は使わないだろうと思う。

start_sync_with_revealed_spks

```
    pub fn start_sync_with_revealed_spks(&self) ->
SyncRequestBuilder<(KeychainKind, u32)> {
        use bdk_chain::keychain_txout::SyncRequestBuilderExt;
        SyncRequest::builder()
            .chain_tip(self.chain.tip())
            .revealed_spks_from_indexer(&self.indexed_graph.index, ..)
            .expected_spk_txids(self.indexed_graph.list_expected_spk_txids(
                &self.chain,
                self.chain.tip().block_id(),
                ..,
            ))
    }
```

UTXO数は直接同期時間には影響しない

同期メソッド `sync()` は `SyncRequest` に沿って同期するだけなので、UTXO数が増えても `SyncRequest` にそれを反映させない限りは同期時間に影響しない。

`SyncRequest` はSPK, OutPoint, TXIDを指定するのでUTXOをそこに含めれば影響してくるのだが、同期対象を細かく制御していけばUTXOを無条件でスキャンする必要はなくなるはずである。

TX数と処理時間の増加

BDKのウォレットを運用していると処理時間がだんだんかかるようになっていく。

https://claude.ai/share/15c32c85-351b-41fe-bf60-53fa1bbdcc5d

同期 : バックエンドに問い合わせるため件数に応じて時間がかかる。

DB探索 : `list_unspent` のようにUTXOを探したりトランザクション作成したりするときに時間がかかるようになる。これは送受信したトランザクション数が影響するようだ。

`wallet.transactions()` のようなトランザクション取得も同様に時間がかかる

Coin Selection : 引数に「必須のUTXO」「オプショナルのUTXO」があり、オプショナルのために内部で `list_unspent` 相当の処理をしている。manually_selected_only()によって抑制できるという情報もあったが試していない。自分でUTXOを選択するならCoin Selectionを自分で実装せずUTXO選択だけ自分でやるようにした方が実装としては面倒が少ないだろう。

残高取得 : これもTxGraphを使うからか時間がかかる。

このように、思ってもない処理に時間がかかるようになる。

`ElectrumClient` を作ったりするのは時間がかからない。

現在DB(正しくは TxGraph)からトランザクションを削減する方法がないので、どうしようもない。ウォレットの再構築によってTxGraphも再構築されて整理することができるそうだ。そのときはノードアプリを再起動させることになるか、少なくともその間はBitcoinに関する操作はできない。ノードが定足数より多く運用しているなら1台ずつ再構築していけばサービスを停止しなくて済むかもしれないが、ノードが連携しているわけではないので手動で行うなどした方が無難か。

大量のトランザクションを裁くのにbdk_walletは向いていない

TX数と処理時間の増加のように、TX数が増えるとなんやかやと処理に時間がかかるようになる。

`sync` 関連で5秒、 `list_unspent` 関連で20秒、のような時間のかかり方をしている。

計測してみると、 `list_unspent()` そのものは4秒程度だった。UTXOのアドレスやconfirmation数を追加で設定しているのだが、そこまで行うと13秒くらいまで膨らんでいた。

`rust-bitcoin` のラッパーみたいな `bdk_chain` や `bdk_electrum` はともかくTxGraphを減らすことができない `bdk_wallet` は大量のトランザクションを扱うことになるアプリには向いていないのだろう。
https://claude.ai/share/0b8f0fb8-40d7-482d-8dd3-d6dd42a1a5fc

syncを活かしてlist_unspentに頼らない

同期はsync()の戻り値を bdk_wallet に `apply_update()` している。これも TxGraph がかかわるので時間がかかるのだが、自分で同じものを作るのはかなり大変そうなのでここは活かしたままにしたい。

`list_unspent()` のデータがいるのはPeg-Inするトランザクションがあるかどうかを調べるときなので、UTXOすべてはいらない。トランザクションは相手にBitcoinアドレスを伝えて送金してもらうだけなのでOutPointがわかっているわけではない。 bdk_wallet にアドレスからUTXOを取得する仕組みがあればよいのだが、ない。なので `list_unspent()` で全部取得しているという次第である。

そうなると `sync()` の戻り値SyncResponseを自分で管理して `list_unspent()` 相当のことができればよい。 `full_scan()` の戻り値はFullScanResponseでちょっと型が違う。これにも対応せねばなるまい。違いは `last_active_indices` だけのようだ。

FullScanResponse

```
pub struct FullScanResponse<K, A = ConfirmationBlockTime> {
    pub tx_update: TxUpdate<A>,
    pub last_active_indices: BTreeMap<K, u32>,
    pub chain_update: Option<CheckPoint>,
}
```

SyncResponse

```
pub struct SyncResponse<A = ConfirmationBlockTime> {
    pub tx_update: TxUpdate<A>,
    pub chain_update: Option<CheckPoint>,
}
```

bdk_core::TxUpdate

```
pub struct TxUpdate<A = ()> {
    pub txs: Vec<Arc<Transaction>>,
    pub txouts: BTreeMap<OutPoint, TxOut>,
    pub anchors: BTreeSet<(A, Txid)>,
    pub seen_ats: HashSet<(Txid, u64)>,
    pub evicted_ats: HashSet<(Txid, u64)>,
}
```

:bdk_core::CheckPoint

```
pub struct CheckPoint(/* private fields */);
```