RPCアプリの作成

@DG Lab - Karl-Johan Alm

このセッションについて

0104. RPCアプリの作成：注意点とベストプラクティス（前半）

0202. RPCのコードを触ろう

0302. RPCアプリの作成：注意点とベストプラクティス（後半）

Agenda

- Task 1〜3の解答例
- Task以外

Task 1

Task 1

場所：payments.js, 25行目（createInvoice）

内容：model.invoiceの作成

詳細：関数の足りない部分を補う（具体的には、bitcoin addressを獲得し、それをinvoiceのDBに登録する）

ヒント：非同期プログラミングを調べるとやり易くなる（http://bc-2.jp/materials/0103_追加書類.pdfを参考に）

Task 1

リファレンス実装では5行（4ステップ）。

テスト（npm test）でPASSすること（13/40→22/40）:

```
Invoice ✓ can be created
reorg unconfirm then... / double spend (replace...):
  ✓ can create an invoice
  ✓ can pay an invoice
double spend replace (payment to same address):
  ✓ submits invoice
```

Task 1 (1/2)

まず、ビットコインの新しいアドレスを手に入れる。

```
bitcoin.getNewAddress((err, response) => {
    if (err) return cb(err);
    // ...
});
```

エラーが出ると、それをcbに渡して終了。

Task 1 (1/2)

ビットコインの新しいアドレスを手に入れる方法その2。

```
const addr = bitcoin.getNewAddressS().result;
```

これでも大丈夫だけど、

```
try … catch (e) { cb(e); }
```

が必要。

Task 1 (2/2)

アドレスを獲得したら、**model.invoice.create**を呼んで、結果をcbに飛ばせば完了。

```
bitcoin.getNewAddress((err, response) => {
    if (err) return cb(err);
    const addr = response.result;
    model.invoice.create(satoshi, content, addr,
cb);
});
```

Task 2

Task 2

場所：payments.js, ~100行目（updateInvoice）

内容：paymentの対応

詳細：paymentの確認と

total/final/disabledAmountの計算

（上の方に変数がある）

Task 2

リファレンス実装では25行（13ステップ）。

テスト（npm test）でPASSすること（22/40→25/40）

ヒント1：reorgが発生した場合、以下が必要：

```
model.payment.setStatus(payment._id, 'reorg', (err) => {
  model.history.create(invoice._id, payment._id, {
    action: 'reorg',
  …
```

Task 2

ヒント2：reorgが発生したかどうかは、txがブロックに入っているかどうかで分かる。

そして、transaction.blockhashがnullであれば、そのtxはブロックに入っていないということになる！

Task 2

ここでやりたいことは：

① paymentに付いているtransactionを確認

② totalAmountに金額を足す

③ 確認済みかどうかを決めてfinalAmountに金額を足す

④ 一番低いconfirmations（ブロック数）を確かめる

Task 2

TxがOKかどうか：

・ノードからtxを入手して、

i エラーが出た→txが見つからない→駄目

ii blockhashがない→txがブロックから外れた
→恐らくreorgが起きた。
ダブルスペンド（等）の可能性→駄目

Task 2

i は既に入っているので何もしなくても良い

ii は発生した場合、model.paymentのステータスを'reorg'にして次のpayment（asyncCallback）へ。

ユーザー的に上の履歴を残さないと駄目なので、
model.history.createを使ってreorgというactionを入れる。

Task 2 (1/4)

```
if (!blockhash) {
  // reorgが起きた
  disabledAmount += amountSatoshi;
  if (payment.status === 'reorg') {
    // 既に知っているから何もしない
    return asyncCallback();
  }
  // …
```

Task 2 (2/4)

続き（reorgが起きたケース）:

```
    model.payment.setStatus(payment._id, 'reorg', (err) => {
        model.history.create(invoiceId, payment._id, {
            action: 'reorg',
            invoiceId: invoiceId,
            paymentId: payment._id,
        }, 'paymentはreorgされた', asyncCallback);
    });
    return;
}
```

Task 2 (3/4)

これでpaymentはOKなので、計算に入る。

```
   const isFinal = confirmations >=
config.requiredConfirmations;
   if (minConfirms > confirmations) minConfirms =
confirmations;
   if (maxConfirms < confirmations) maxConfirms =
confirmations;
```

Task 2 (4/4)

```
totalAmount += amountSatoshi;
if (isFinal) finalAmount += amountSatoshi;

model.payment.setStatus(
   payment._id,
   isFinal ? 'confirmed' : 'pending',
   asyncCallback);
```

Task 3

Task 3

場所：payments.js, ~140行目（updateInvoice）

内容：invoiceのアップデート

詳細：ここまでに確認・集計したpaymentの情報を元に、invoiceをアップデートする

Task 3

リファレンス実装では23行（21ステップ）。

テスト（npm test）でPASSすること（25/40→40/40）

つまり、

Task 3が完成すれば**全てのテストがPASSするはず！**

Task 3

ヒント：（過不足なく）丁度支払ったという事を

`Math.abs(支払った金額 - 要求した金額) < config.thresholdSatoshi`

というように確かめる。

Task 3

ここでやりたいことはただ1つ:

invoiceのステータスを確定すること。

そのステータスは7つある。

Task 3

7つのステータス

場合	確認済み	未確認
まだ払ってない	unpaid	---
払ったが足りない	partial	pending_partial
払い過ぎた	overpaid	pending_overpaid
払った	paid	pending_paid

Task 3

- 確認済みかどうかを判断する為に、confirmationsの最小値が必要
- 丁度払ったかどうかのフラグがあればやりやすい
- 確認済みの場合とそうでない場合を区別する必要もある

Task 3 (1/3)

上の方で求められる変数:

```
const confirmations = Math.min(minConfirms, maxConfirms);
const pendingAmount = totalAmount - finalAmount;
const threshold = config.thresholdSatoshi;
const finalRem = invoice.amount - finalAmount;
const totalRem = invoice.amount - totalAmount;
const finalMatch = Math.abs(finalRem) < threshold;
const totalMatch = Math.abs(totalRem) < threshold;
```

Task 3 (2/3)

下の方：

```
let status;
```

というヘルプ変数を入れて、

Task 3 (3/3)

7つのケース（順番はあくまで例の一つ）:

```
if (finalMatch && totalMatch)          status = 'paid';
else if (totalAmount === 0)            status = 'unpaid';
else if (finalRem < -threshold)        status = 'overpaid';
else if (totalRem < -threshold)        status = 'pending_overpaid';
else if (totalMatch)                   status = 'pending_paid';
else if (totalAmount === finalAmount)  status = 'partial';
else                                   status = 'pending_partial';

model.invoice.updateStatus(invoiceId, status, cbwrap);
```

Task以外

タスク以外にも、しなければいけない、ややこしい事がいくつかある。

scannerは恐らく一番ややこしいので、終わる前にそれの流れにちょっと触れる。

Scanner: **scanner.js**

目的：tx、invoiceの状況が変化した際にアップデート

① scan関数がbitcoinノードから最新ブロックリスト取得

② ブロックの中身（tx）を見て、reorgが起きたか判断

③ それぞれのtxをチェック

④ reorgが起きた場合、watched invoiceもチェック

Scanner: 最新ブロックリスト取得

```
bitcoin.listSinceBlock(<前回scanした最新ブロックハッシュ>, (err, info) => {
   if (err) return cb(err);
   // info.result.transactions等がある
   // ...
});
```

Scanner: reorgの判断

txをみて、reorgが起きたか判断するにはcategoryを見る。

```
for (const tx of result.transactions) {
   if (tx.category === 'orphan') {
      autoCheckWatched = true;
      break;
   }
}
```

Scanner: transactionのチェック

- bitcoinノードからtxを求める
- 関係ないtxなら無視
- **payment.updatePayment**を実行
- **checkWalletConflict**を実行
- invoiceが付いたtxなら、**affectedInvoices**に追加

Scanner: watched invoices

invoice.confirmations = min(payments.confirmations)

invoice.confirmationsが100を超えるまでwatchedというフラグがオンになる。

つまり、払われて100ブロックが経つまで見続ける。

Reorgが起きたら全てのwatched invoiceを再確認する。

（100ブロック≒1000分≒17時間）

Scanner: **checkWatchedInvoices**

・DBから**getWatchedInvoices**

・全てのpaymentを取り出して、それぞれのtxに対するwallet conflictを確認

・最後に**payment.updateInvoice**を実行

Scanner: **checkWalletConflicts**

- bitcoinノードからtxを求める
- tx.walletconflictsが空っぽなら終わり
- tx.walletconflictsの中からtxidを一個ずつ取り出して、
  - bitcoinノードから**ctx**を求める
  - **payment.updatePayment(ctx)**を実行

@DG Lab - Karl-Johan Alm