

Starting NEO

NEO Keymakers Japan 著

2019-01-23 版 発行

前書き

NEO の技術書を書くにあたって

スマートエコノミーの実現を目指すブロックチェーンプロジェクト NEO は、世界各地に拠点があり、コミュニティベースの運営が行われています。中国には NEL^{*1}、アメリカをはじめとする英語圏では、CityOfZion と呼ばれる NEO のディベロッパーコミュニティがあり、活発な議論と開発が進んでいます。しかしながら、日本においては開発者向けのそのようなコミュニティはなく、国内における NEO の普及は、それほど進んでいないのが現状です。本プロジェクトは、NEO Keymakers Japan を中心に行われる、NEO の日本人開発者向けの手引書作成プロジェクトです。NEO 上で開発を行うにあたり、現状では日本語文献が圧倒的に少ないので、国内における NEO の普及はそれほど進んでいないのが実情です。そこで、NEO 上での開発参入の敷居を下げるべく、ブロックチェーンに触れたことがある方がスムーズに NEO について学習していくとともに、Dapps 開発等に取り組む上で役に立つような文献を作成します。

想定している読者

この本は、主にプログラマ向けに書かれています。もしプログラム言語を使えるのであれば、どのように NEO というブロックチェーンが動くのか、どのように NEO のプラットフォームを利用するのか、どのようにブロックチェーンのソフトウェアを開発するのかが分かります。最初のいくつかの章は、NEO やブロックチェーンの内部の動きを理解したいものの、プログラマーではない方向けの入門として最適でしょう。

^{*1} 参考：neonewstoday より <https://neonewstoday.com/development/neweconolab-nel/>

コード例

例は C#や Golang、Python で説明されており、Linux または Mac OS X のような Unix ライクなオペレーティングシステムのコマンドラインを使って実行できます。本書で使用したコードは GitHub repository のメインリポジトリの code ディレクトリにあります。この本にあるコードをフォークして、コード例を試してみてください。そして、修正点があれば GitHub を通してご連絡をお願いします。すべてのコードスニペットは、ほとんどのオペレーティングシステムにおいて、対応する言語のコンパイラとインタプリタを最小限インストールすることで、置き換えられます。本書では、必要に応じて、基本的なインストールの命令と、その命令のアウトプットについて順を追った例を提供します。すべてのコードスニペットは、読者が本書と同じ計算をすれば同じ結果となることを確認できるよう、できる限り現実の数値と計算を使用しています。たとえば、秘密鍵と、これと対応する公開鍵およびアドレスは、すべて現実に存在するものです。本書に掲載されたトランザクション、ブロック、ブロックチェーンの事例は、実際の NEO のブロックチェーンに存在しており、公開された台帳の一部ですので、読者は実際にこれらを確認することができます。

ライセンス

原作者のクレジット（氏名、作品タイトルなど）を表示し、改変した場合には元の作品と同じ CC ライセンス（このライセンス）で公開することを主な条件に、営利目的での二次利用も許可します。

Creative Commons Attribution-ShareAlike 4.0 International License

用語集

この用語解説では、NEO あるいはブロックチェーンに関連して使われる用語の多くを解説しています。これらの用語はこの本全体を通じて使われますので、タップクリアランスとして参考にしてください。

NEO

通貨単位または通貨そのもの（NEO）で、ネットワークおよびソフトウェアの総称でもあります。

NEOGas (Gas)

通貨単位または通貨そのもの（Gas）で、スマートコントラクトのデプロイや実行に使われます。

ブロックチェーン

検証されたブロックの連なり。各ブロックはひとつ前のブロックと繋がっており、genesis ブロックに至るまで続いています。

genesis ブロック

NEO ブロックチェーンにおける最初のブロック。

ネットワーク

トランザクションやブロックをすべての NEO ノードに拡散する、P2P ネットワークのこと。

アドレス

NEO アドレスとは、ASH41gtWftHvhuYhZz1jj7ee7z9vp9D9wk といった、「A」から始まる文字と数字の連なりです。ユーザーは、email を自分の email アドレスに送るよう誰かに依頼するのと同じように、NEO または、NEOGas を自分の NEO アドレスに送るよう依頼します。

秘密鍵

紐づけられたアドレスに送られた NEO を解錠するための秘密の番号で、936a185caaa266bb9cbe981e9e05cb78cd732b0b3280eb944412bb6f8f8f07afといった形をとります。

ハッシュ

二進法の入力に対する、デジタルなフィンガープリントのこと。

ウォレット

NEO アドレスと秘密鍵を格納するソフトウェアのこと。NEO または Gas を送ったり受けたり保有したりすることに用います。

トランザクション

NEO をあるアドレスから他のアドレスに送ること。より正確にいえば、トランザクションとは、価値の転移を表した、署名されたデータ構造です。トランザクションは、NEO ネットワークに伝えられ、コンセンサスノードによって集められ、ブロックにまとめられ、ブロックチェーンに格納されます。

ブロック

グループにまとめられたトランザクションのことで、タイムスタンプとひとつ前のブロックの Hash 値などが含まれています。ブロックは、コンセンサスノードによって、それによってトランザクションが検証されます。検証されたブロックは、ネットワークにおける合意によりブロックチェーンに加えられます。

手数料

スマートコントラクトのデプロイまたは実行にかかる費用のこと。NEO や Gas の送受信には手数料は発生しません。

dBFT

NEO ネットワークにおけるコンセンサスアルゴリズムのこと。限られた参加者によって、コンセンサスをとる仕組み。

NEP

NEO 改善提案 (NEO Enhancement Proposals) の略称で、NEO コミュニティのメンバーが NEO を改善するために提出してきた一連の提案のことを指します。たとえば、NEP5 は、トークンの仕様に関する提案です。

目次

前書き	2
NEO の技術書を書くにあたって	2
想定している読者	2
コード例	3
ライセンス	3
用語集	4
第 1 章 イントロダクション	11
1.1 ブロックチェーンとは	11
1.2 P2P 技術	11
1.3 ブロックチェーンで何ができるのか	13
1.3.1 金融系	14
1.3.2 ポイント/リワード	15
1.3.3 資金調達	15
1.3.4 コミュニケーション	16
1.3.5 資産管理	16
1.3.6 ストレージ	17
1.3.7 認証	17
1.3.8 シェアリング	17
1.3.9 物流管理	18
1.3.10 コンテンツ	18
1.3.11 公共	18
1.3.12 医療	19
1.3.13 IoT	19
第 2 章 NEO の仕組みと概要	21

2.1	NEOについて	21
2.2	NEOのトークン設計について	22
第3章	キー、アドレス	23
3.1	公開鍵暗号	23
3.2	ハッシュ	24
3.3	秘密鍵	24
	公開鍵と秘密鍵	25
	安全な秘密鍵とは	26
3.4	公開鍵の生成	26
3.4.1	楕円曲線とは	26
3.4.2	楕円曲線暗号	28
3.4.3	楕円曲線暗号を利用した公開鍵の生成	28
	アドレスと公開鍵	28
3.5	Base58とBase58Checkエンコード	29
3.6	NEOのアドレスを作つてみよう！	30
3.6.1	秘密鍵から公開鍵の生成	32
3.6.2	RIPEMD-160で公開鍵のハッシュを生成	33
3.6.3	base58でエンコード	34
3.6.4	最終コード	35
	アドレスの最初の文字「A」の理由	42
第4章	ウォレット	43
4.1	この章でカバーすること	43
4.1.1	ハードウェアウォレットで管理する方法	43
4.1.2	NEON Wallet	43
4.1.3	NEO GUI	46
4.1.4	NEO Tracker	47
4.1.5	NEOウォレット	48
4.1.6	NEO CLI	49
第5章	トランザクション	50
5.1	ブロック	50
5.2	トランザクションの内容	50
5.2.1	トランザクションのタイプ (Type)	52
5.2.2	トランザクションの属性 (Attributes)	56

5.2.3	トランザクションの入力	57
5.2.4	トランザクションの出力	57
NEO3.0 では Transaction の種類を簡素化する方向へ		58
5.3	NEO scan	58
5.4	悪意のあるトランザクションに対する取り組み	58
第 6 章	NEO ネットワーク	60
6.1	オフチェーンガバナンス	60
6.2	オンチェーンガバナンス	60
6.3	コンセンサスノードへのなり方	61
6.3.1	要件調査	61
6.4	プレーヤー詳細	62
6.4.1	NEO Foundation	62
6.4.2	CITY OF ZION	62
6.4.3	KPN	62
6.4.4	Swisscom	63
6.5	ブロック高	63
6.6	NEO テストネット	63
6.7	テストネットの特徴	63
6.8	NEO クライアントをダウンロードする	63
6.9	テスト GAS とテスト NEO の獲得方法	64
6.9.1	リクエストフォームに回答する	65
6.9.2	マルチパーティーアドレスを作成する	65
6.9.3	他のアカウントのアセットを送信する	67
6.10	クライアントをすばやく同期する	67
6.10.1	ステップ 1	67
6.10.2	ステップ 2	68
6.10.3	ステップ 3	69
6.11	ネットワーク・プロトコル	70
6.12	NEO のノードでプライベートチェーンを構築する	70
6.12.1	Virtual Machine をセッティングする	71
6.12.2	ウォレットを作成する	71
6.12.3	configuration ファイルを修正する	72
6.13	無料の NEO を手に入れる	75
第 7 章	コンセンサスアルゴリズム	77

7.1	用語一覧	77
7.2	概要	77
7.3	ビサンチン將軍問題 (BFT)	78
7.4	dBFT (Delegated Byzantine Fault Tolerant)	79
7.5	システムモデル	80
7.6	一般手順	80
7.7	View の変更	81
7.8	フローチャート	82
7.8.1	役割	83
7.8.2	理論	83
7.8.3	Honest Speaker	84
7.8.4	Dishonest Speaker	85
7.9	実装	86
7.10	アルゴリズム	87
7.10.1	定義	87
7.10.2	必要事項	88
7.10.3	アルゴリズム概要	88
7.11	障害耐性度	93
第 8 章 SDK や開発について		95
8.1	neo-local	95
8.1.1	インストール	95
8.1.2	起動	96
8.2	neo-local-faucet	97
8.3	neonjs	97
8.4	neo-python	98
8.4.1	プロンプト操作	98
8.5	Neoscan	100
8.6	その他のツール	100
第 9 章 スマートコントラクト		101
9.1	NEO Smart Contract 2.0	101
スマートコントラクトの変更について		102
9.2	スマートコントラクトの対応言語	102
9.3	スマートコントラクトの実例	103
9.3.1	音楽コンテンツの自己出版	103

9.3.2 コンテンツ作成者・所有者の管理と証明	103
9.4 手数料形態	103
デプロイに必要な手数料	105
第 10 章 はじめての dApps 開発	106
10.1 概要	106
10.2 NEO のローカル環境の構築と起動	106
10.3 スマートコントラクトの開発	107
10.3.1 スマートコントラクトの作成	108
10.3.2 作成したスマートコントラクトのコンパイル	109
10.3.3 コンパイルしたスマートコントラクトのデプロイ	110
10.3.4 デプロイスマートコントラクトへのアクセス	113
10.4 フロントエンドからのスマートコントラクト呼び出し	114
第 11 章 NEP	116
11.1 概要	116
11.2 タイプ	116
11.2.1 スタンダードトラック型	117
11.2.2 インフォーメーション型	117
11.2.3 メタ型	117
11.3 NEP の出し方	118
11.3.1 アイデア	118
11.3.2 執筆者	118
11.3.3 公開	118
11.3.4 プルリクエスト	119
11.3.5 編集者の承認	119
11.3.6 スタンダードトラック型 NEP の出し方	119
11.3.7 承認とステータス	120
11.3.8 フォーマットとテンプレート	121
11.4 実際の NEP	123
11.4.1 NEP5	124
11.4.2 NEP6	125
11.4.3 NeoX	125
後書き	127
Github 上での貢献	127

第 1 章

イントロダクション

この章では、ブロックチェーンについて概念的に理解できるように解説し、ブロックチェーンの利用が期待されている分野について俯瞰していきます。

1.1 ブロックチェーンとは

ブロックチェーンは、金融分野にとどまらず、広く「分散型台帳」として各分野での応用が期待されている、汎用性の高い技術です。

1.2 P2P 技術

P2P (Peer-to-Peer) とは、対等な者同士が繋がるネットワークという意味を持ちます。P2P 型のネットワークモデルには、ネットワークを構成するコンピュータ（ノードと呼ぶことがあります）が相互につながりながら、サービスを提供し、同時に受け入れているのです。これに対応したネットワークモデルとして、クライアントサーバ型があります。クライアントサーバ型では、ネットワークを構成するコンピュータに役割を明確に分担させています。つまり、サーバと呼ばれる特定の（高負荷な）処理を行いサービスを提供する端末と、クライアントと呼ばれる、サーバにサービスを要求し、それを享受する端末が存在していて、サーバを中心としたネットワーク構成になっています。

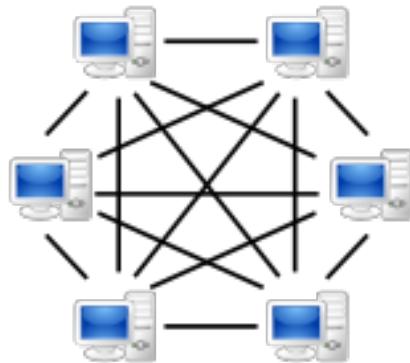


図: P2P ネットワークの構成図



図: クライエントサーバー型の構成図

ふたつの型が抱えるメリット・デメリットを見てていきましょう。まず、クライアントサーバーモデルでは、サーバが中心に位置しているため、サーバーの障害がシステム全体

の障害に直結してしまいます。一方、P2P 型のモデルの場合は、全てのノードが対等に振る舞うケースが多いため、このような障害は発生しません。一方、P2P ではその特性上、各ノードは、他の多数のノードと通信経路を確保する必要があります。全てのノードをつなぐ回線の品質が保証されていれば、経路にかかわらず、高品質な通信を享受できますが、現実的な問題として難しいところがあります。すなわち、P2P のネットワーク構成それ自体に相応の工夫が必要になってきます。また、クライアントサーバ型では、通信相手はつねにサーバーと特定できる一方、P2P 型では通信相手の確認を常に使う必要があります。

1.3 ブロックチェーンで何ができるのか

ブロックチェーン技術は、改ざんが実質的に不可能であり、ゼロダウンタイムなシステムを提供することができます。クライアントサーバ型のシステムを抜け出して、このブロックチェーン技術を適応できる分野としては次のようなことが考えられます。

ブロックチェーン技術活用のユースケース

- ビットコイン発祥のブロックチェーン技術を改良しながら、金融以外の分野にもユースケースが広がっており、「ビットコイン2.0」と呼ばれている

金融系	ポイント／リワード	資産管理	商流管理	公共
決済 (SETL、FactoryBanking)	ギフトカード交換 (GyftBlock)	bitcoinによる資産管理 (Uphold)(旧Bitreserve)	サプライチェーン (Skuchain)	市政予算の可視化 (Mayors Chain)
為替・送金・貯蓄等 (Ripple, Stellar)	アーティスト向けリワード (PopChest)	土地登記等の公証 (Factom)	トラッキング管理 (Provenance)	投票 (Neutral Voting Bloc)
証券取引 (Overstock, Symbiont, BitShares, Mirror, Hedgy)	プリベートカード (BuyAnyCoin)	データの保管 (Stroj, BigchainDB)	マーケットプレイス (OpenBazaar)	バーチャル国家/宇宙開発 (BitNation/Spacechain)
bitcoin取引 (itbit, Coinffeine)	リワードトーカン (Ribbit Rewards)		金保管 (Bitgold)	ベーシックインカム (GroupCurrency)
ソーシャルバンкиング (ROSCA)			ダイヤモンドの所有権 (Everledger)	
移民向け送金 (Toast)			デジタルアセット管理・移転 (Colu)	
新興国向け送金 (Bitpesa)				
イスラム向け送金/シャリア遵法 (Abra, Blossoms)				
資金調達				
アーティストエクイティ取引 (PeerTracks)				
クラウドファンディング (Swarm)				
認証				
デジタルID (ShoCard, OneName)				
アート作品所有権/真偽証明 (Ascribe/VeriSart)				
薬品の真偽証明 (Block Verify)				
コンテンツ				
ストリーミング (Streamium)				
ゲーム (Spells of Genesis, Voxelnauts)				
コミュニケーション				
SNS (Synereo, Reveal)				
メッセンジャー、取引 (Getgems, Sendchat)				
シェアリング				
ライドシェアリング (La'ZooZ)				
将来予測				
未来予測、市場予測 (Augur)				
医療				
医療情報 (BitHealth)				
IoT				
IoT (Adept, Filament)				
マイニング電球 (BitFury)				
マイニングチップ (21 Inc.)				

図：ブロックチェーンを利用したユースケース例

出典：経産省 4/28 勉強会

次にそれぞれの分野でどのような利用方法が期待されているのかを概観してみましょう。

1.3.1 金融系

- 貯蓄

主な暗号通貨には「中央銀行」が存在しない為、勝手に通貨を発行されることなくインフレが起こりにくい通貨といえます。そのためビットコインはデジタルゴールド、つまり「価値の保存」ができる電子データとして考えられています。また、分散型ネットワークが堅牢なセキュリティとなっているため、暗号通貨の資産は消すことも、盗むこともできないため、安心して保有しておくことができます。

- 送金

スマホやパソコンがあれば、個人間で相手に送金するのに銀行は必要ありません。数百円の振込手数料や、数千円かかる国際送金手数料も数円～数十円で送金することが可能で、さらに、ブロックチェーンが銀行と同じ役割をしているので、24時間相手に送金することができます。暗号通貨の ATM は世界中にあり、受け取り側は 24 時間いつでもビットコインを法定通貨（円・ドルなど）に換えて引き出すことが可能です。また、日本の銀行も国際送金ビジネスに名乗りを上げています。現在の送金システムでは国際送金を行う場合、いくつもの金融機関を介さねばならず、少額の送金であっても数千円の手数料を払う必要があります。これに暗号通貨を利用すれば【円→暗号通貨→ドル】といったように複数の金融機関を介すことなく他国の通貨と変えることができるようになります。

- 証券取引

株券は電子データ化されているので、証券会社が巨額のコストをかけてデータサーバーなどで集中管理することで、正確性や安全性を担保しています。この株券をブロックチェーンで生成されたトークンに置き換えることで、低コストで高いセキュリティを担保することができるため「取引手数料を抑えることができる」と期待されています。また、株主へ保有株分の議決権を付与し、チェーン上で決議を行うことで議決の管理（足し算）も簡単に行うことができます。これにより少額の株主がいても容易に決議できるため、単元は 100 株ではなく、0.1 株からでも投資できるようになり株式市場の活性化が期待されます。

1.3.2 ポイント/リワード

- ギフトカード

たとえばギフトカードにブロックチェーンを活用すると、作成されたアセット（トークン）の複製は不可能なので、偽造される心配はありません。また、チケットショップなどの買取業者も本物であるかを確認する手間もがからず容易に換金できるようになります。ユーザーへの配布や譲渡は、わざわざ郵送する必要がなくメール感覚で送りたい相手に送ることもできるようになります。

- チケット

チケットは転売されることが多く、有名なアーティストの LIVE になると正規の値段よりも何十倍も高く売買されています。これは主催者側の意図とは大きく反しています。これにブロックチェーンを活用すると、購入者を会場で認証することができるようになるので、転売が不可能になります。したがって、本当にコンサートに行きたい人がチケットを買えるようになります。

- プリペイドカード

難民を抱える地域で、ブロックチェーンを活用したプリペイドカードが発行されています。ブロックチェーン技術を活用したプリペイドカードは、銀行などの仲介業者を交えず個人間で送金できる仕組みとなっているため、身分証明書を持っていない難民でも使用できることが特徴です。また、このカードそのものが銀行の口座としても機能するため、給与の振り込みとしても利用できるようになっています。

1.3.3 資金調達

- クラウドファンディング

ICO はトークンを販売し資金調達を行いますが「本人確認の必要がなく、世界中の企業・個人が誰でも参加することができる」というのが大きな特徴になります。

- 寄付

寄付文化が根付いているアメリカでは 30 兆円規模の寄付市場があります。また、日本でも 1 兆円を超えるといわれています。現在では世界の恵まれない国へ寄付をする場合は、日本赤十字など団体を通して寄付をしますが、多くの団体が寄付金の中から活動資金を捻出しています。これもブロックチェーンを活用すれば、さきほどのプリペイドカード

などに直接寄付ができるようになりますので、誰にも手数料を取られることなく現地の方々へ寄付を送ることができます。また、第三者を経由するにしても「誰が、いつ寄付をしたか」だけではなく「いつ、誰に、いくらが届いたか」まで確認できるようになりますので市場の活性化にも繋がると思われます。

1.3.4 コミュニケーション

- SNS

Twitter・Facebook・Instagramなどはユーザーが発信する内容の質の低下が問題になっています。主にSNSはニックネームで登録をするので「虚偽の情報」を流すハードルが低いのです。また、国家機密や企業秘密が流出した場合、本人を特定するのも簡単ではありません。これを解決するには、全てのアカウントで本人認証を行えば「誰が発信した情報なのか」を明確にすることができます。有効な手段かと思われますが、企業が個人情報を取り扱うにはリスクが伴います。それはセキュリティにかかるコストです。その堅牢なセキュリティであるブロックチェーン技術を活用すれば、個人情報の流出を防ぐことができ、ユーザーも安心して個人情報を入力できるようになります。

1.3.5 資産管理

- 登記

管理者がいなくても正確性や安全性の担保ができるようになるのがブロックチェーン技術ですから「個人・法人・動産・不動産・物権・債権」など不動産登記法や商業登記法などで定められた登記手続きの効率化に繋がります。また、多額の人件費やデータサーバー費用も削減できるため、国家予算にも大きく影響してくるでしょう。

- 債権の移転

ブロックチェーン技術は、信頼を管理するのに適しているため、紙ベースで行っていた作業は大幅に削減することができます。たとえば、外航貨物海上保険などの保険証券は輸出入貨物に対して保険をかけるため、国をまたいで保険証券を譲渡しています。この譲渡は主に紙で行われているため、時間がかかるばかりか紛失リスクも伴います。これをデジタルアセットで譲渡すれば、時間もかからず紛失リスクは軽減します。これはゴルフ会員権・車・高級腕時計・ダイヤモンドにいたるまで価値や契約の譲渡が素早く簡単にできるようになります。

1.3.6 ストレージ

- データの保管

企業間のデータベースシステムとして、プライベートチェーンまたは、コンソーシアムチェーンのブロックチェーンを活用することができます。

1.3.7 認証

- 本人確認

これまでのサービスは、各サービスごとに本人確認が必要でした。これはサービスの「提供側・受ける側」共に面倒な手続きです。これもブロックチェーンで作成されたアセットを個人情報と紐づけることで解決できます。

- 著作権

自分で作曲した楽曲、歌詞などをチェーン上に記録しておけば、誰にも盗用されることなく著作権を主張することができます。また、スマートコントラクトを活用すれば、再生時やダウンロード時に使用料の請求を行うことができます。著作権者がグループだったとしても、利益配分もあらかじめ決められた分配方法で自動的に分配することができます。これは、音楽だけではなく「アート・写真・文章・動画・特許」など、さまざまな領域において活用することができるのです。このようにメジャー・デビューを目指さなくても、個人で小さな経済圏を作ることもできる可能性もあります。

- 公証人

ブロックチェーンを活用すれば、公証人を介さずとも公正証書を発行できます。たとえば、遺書です。たとえ本人が亡くなっていたとしても、それが本物であることが証明されます。さらに、資産はスマートコントラクトにより自動的に相続税を計算し、あらかじめ指定された相続人へ相続させることができます。

1.3.8 シェアリング

現在、エアビー・アンド・ビーのような民泊はシェアリングエコノミーと呼ばれていますが、実際には第三者が仲介しているためシェアリングというよりは「レンタル」に近いものです。これにブロックチェーンを活用すれば、部屋のシェアや、車のシェアも相手の身元が分かるので安心して取引を行うことができます。つまり第三者に手数料を取られるこ

となくシェアリングすることが可能になるのです。

1.3.9 物流管理

- サプライチェーン

どんな原材料で、どのように調達し、どうやって製造されたのか、そして出荷された後の追跡など、商品がエンドユーザーに届くまでのプロセスを全て遠隔で監視、記録することができます。さらに、リアルタイムで参加者全員に共有できるのでボトルネックも可視化でき業務の効率化が期待できます。

- マーケットプレイス

ネットショッピングで【Aさん→Bさん】へ商品を売る場合、BさんはAさんに代金を支払います。しかし、代金を支払っても商品が届く保証もありませんし、先に商品を届けてもお金を支払ってもらえる保証はありません。ここで活躍するのがスマートコントラクトです。スマートコントラクトは「あらかじめ定めた条件が満たされた時に、自動的に処理を実行する」というものです。自動販売機の例が有名ですが、自動販売機は「120円を入れて押されたボタンの商品を排出する」このようなスマートコントラクトになっています。同じように、ネットショッピングに応用すると「支払いと同時に商品を発送する」と条件を決めておけば自動的に商品の発送が完了します。これをチェーン上で行えば、改ざんはほぼ不可能であるため相手を信頼せずとも安心してエスクロー取引を行うことが可能になります。つまり、現在では楽天、アマゾンなどの第三者を通して商品を購入していましたが、ユーザーが個人間で気楽に取引ができるようになります。

1.3.10 コンテンツ

- ゲーム

ブロックチェーンで作成されたトーカンをゲームのアイテムとして使用するのであれば、改ざんは実質不可能なので、そのアイテムトーカンをユーザー間で交換したり、他のゲームアイテムと交換したりすることが可能になります。

1.3.11 公共

* 市政予算の可視化ブロックチェーンは、全ての取引内容を分散型のネットワークで誰もが閲覧できるようになるため、ブロックチェーン技術で市政予算を管理すれば「私たちの税金が適切に使われているのか?」「不正はないのか?」などを市民、国民に可視化

することができます。また、監査機関も監査する手間やコストが削減できるため、大きな予算削減が期待できます。

- 投票

たとえば、選挙の場合、開票は人の手で行う為、改ざんや意図的な集計ミスなどがあります。また、インターネット投票を行うにしてもハッキングの恐れがあります。これにブロックチェーン技術を活用すると、本人認証により正確性は担保されるうえ、堅牢なセキュリティによりデータの改ざんは不可能になります。また、全国民が取引記録を閲覧できるため従来のシステムに比べ透明性が高くなります。

- 税金

日本円がトークンに置き換われば、送金時やアプリケーション使用時に税金を徴収することができます。家を買う時、車を買う時、住民税から所得税・消費税に至るまで、項目ごとに異なる税率で計算し自動的に徴収することが可能になるのです。たとえば「賃料を送金する時は住民税〇%」というような手法で徴収でき、脱税は不可能であるため平等に税金を徴収できるようになります。

1.3.12 医療

- 医療情報

最近ではセカンドオピニオンと呼ばれる複数の医師に診察をしてもらうよう促す動きもあります。これまで診療された医療記録は各病院のカルテに保管されていますが、この医療記録を医療機関同士で共有しようという動きが国内でも進んでいます。たとえば、旅行先で診療する場合も、過去の病歴やアレルギー、現在服用している薬などを速やかに把握できるので、診療に役立てるすることができます。また、事故などで救急搬送された場合も、患者本人の意識が無かったとしても免許証やマイナンバーを所持していれば、輸血を行うのに血液型を調べる必要も無く速やかに診療を始めることができます。こういった高度なセキュリティが必要な情報の管理も、ブロックチェーンを活用すれば人的リソース、人件費、その仕組みの構築・維持にかかる膨大なコストをかけることなく、運用できるようになります。

1.3.13 IoT

- IoT

IoT（モノのインターネット）では、全てのモノとインターネットが繋がれば接続先が

増え、トラフィックが膨大になるにつれてサーバ側に処理が集中してボトルネックが発生しやすくなります。これを、ひとつデータベースに集中させるのは負荷大きいため、分散型ネットワークで管理することにより負荷を分散することができます。また、分散型ネットワークは複数のノードが機能を停止しても、残りのノードが稼働している限りゼロダウンタイムで稼働できるため、多少の障害ではサービスが停止しないという「耐障害性」を持ち合わせます。

第 2 章

NEO の仕組みと概要

この章では、NEO がもつ基本的な特徴について触れていきます。

2.1 NEO について

NEO は、スマートコントラクトを実行するプラットフォームの構築を目指すオープンソースプロジェクトです。上海の開発チームは 2014 年頃からブロックチェーン技術の研究開発をスタートしました。このとき開発していたブロックチェーンは Antshares と呼ばれました。



図: Antshares のロゴ

開発が続けられた Antshares はリブランディングを行いプロジェクト名を NEO に変え、2016年秋に ICO を実施します。

▼表 2.1 ICO の概要

期間	資金調達額（BTC）
2016年9月7日～10月17日	6069 BTC

NEO は完全なオープンソースのパブリックブロックチェーンであり、コミュニティによって運営されています。

次のリンクはコミュニティによってまとめられた NEO に関するツールや wallet、有益な情報源などについてまとめられていますので、参考にしてみてください。

<https://github.com/City0fZion/awesome-neo>

2.2 NEO のトークン設計について

ネットワークに流通するネイティブトークンが 2 つ存在します。

NEO

- ネットワークを維持するためのガバナンストークン
- Validator 選出のための投票機能 (governance)
- NEO 保有者に自動的に Gas を分配
- ICO に配布
- 総発行量は、1 億 NEO で、5000 万 NEO が ICO で販売され、残りの 5000 万 NEO は Foundation が管理
- 1NEO が最小単位

Gas

- ブロック生成時に発生
- Network 維持のための手数料 (スマートコントラクトの deploy/invoke)
- Network 内のペイメントトークン
- 少数点以下でも分割できる

現在 (2018/12)、メインネットでの NEO や Gas の送金手数料は無料になっています。

第3章

キー、アドレス

この章では、前半に公開鍵暗号方式や楕円曲線暗号といった暗号技術の基本を説明するとともに、後半では実際に NEO のアドレス生成の過程をコードベースで辿っていき、より NEO のアドレスへの理解を深められるように解説していきます。

3.1 公開鍵暗号

公開鍵暗号は、秘密鍵と公開鍵という2つの鍵を使った暗号のこと、1つの鍵を使う共通鍵暗号^{*1}と対をなす暗号技術です。秘密鍵は自分以外に知らないようにし、公開鍵はオープンにしても問題ない暗号方式です。代表的なものに RSA 暗号が挙げられます。また、通信を暗号化する SSL は公開鍵暗号と共通鍵暗号の両方を利用しています。

公開鍵暗号は次の性質を持ちます。

- 公開鍵で暗号化した文は秘密鍵のみで復号可能
- 秘密鍵で暗号化した文は公開鍵のみで復号可能（出来ない公開鍵暗号も存在）
- 秘密鍵から公開鍵を知ることが可能、公開鍵から秘密鍵を知ることは不可能
- 秘密鍵でサインした文は公開鍵で本人が作成した文であると証明可能（デジタル署名）

以上の性質をもつ公開鍵暗号ですが、暗号通貨ではアドレスの作成とアドレスをもつ本人である証明（デジタル署名）を使われます。デジタル署名は、主に暗号通貨の送金時に本人が送金した証明に使われます。デジタル署名がないと、第三者が誰かの暗号通貨を不

^{*1} 共通鍵暗号は送信者と受信者が共通の鍵を持ち合うことにより秘密通信を行う方式のこと。厳密に分類するとブロックごとに暗号化するブロック暗号と、乱数を用いるストリーム暗号に分けられる。

正に送金し放題になり通貨として機能しません。

暗号通貨では Bitcoin の慣例に倣い楕円曲線デジタル署名アルゴリズム ECDSA (Elliptic Curve Digital Signature Algorithm) が用いられることが多いです。

3.2 ハッシュ

次に、暗号化技術を知る上で欠かせない知識であるハッシュについてみていきます。平文全体に対して秘密鍵で暗号化⇒公開鍵で復号化という処理をすると、平文が長大であるほど演算に時間が掛かってしまい、実用的ではなくなってしまいます。そこでハッシュ関数を利用し、平文を一定の長さに変換してから署名する仕組みが一般的になっています。ハッシュはあらゆる入力データから固定長の小さな出力値を一意に求める仕組みのことを指し、この仕組みで得られた出力値をハッシュ値と言います。代表的なものに MD5 や SHA-1 があります。ハッシュの主な用途は受信したデータの破損や改ざんの検証や公開鍵暗号のデジタル署名です。他にも、パスワードをデータベースに保存するときも普通はパスワードにハッシュを施してから保存します。ハッシュは次の性質を持ちます。

- あらゆる長さの入力データから固定長のハッシュ値を出力する
- 同じ入力データからは必ず同じハッシュ値が得られる
- 似た入力データからでも、まったく違うハッシュ値が得られる
- ハッシュ値から入力データを求ることは出来ない
- 違う入力データから同じハッシュ値が出力される（衝突する）ことはほぼあり得ないがごく稀にあり得る

暗号通貨では、お馴染みブロックチェーンやアドレスの作成、取引（トランザクション）の検証、マーケルルートの作成そしてマイニングなどさまざまところで利用されます。

暗号通貨でよく使われるのは SHA-256 (出力 32byte) や RIPEMD-160 (出力 20byte) で、あとで NEO のコードを見る時にも実際に現れます。NEO アドレスは一方向暗号学的ハッシュ化を使うことで公開鍵から生成されます。暗号学的ハッシュ関数は NEO の中で広範囲に活用されます。

3.3 秘密鍵

さて、公開鍵暗号とハッシュについての概要がつかめたところで秘密鍵について掘り下げていきます。秘密鍵は意図的に指定できるものではなく無作為に選ばれる数値です。秘密鍵による所有権管理は NEO アドレスに結びついた全ての資金の根幹をなします。秘密鍵は署名を生成するときに使われ、この署名は資金を使うときに所有権の主張に必要とな

ります。秘密鍵はいつでも極秘に保っておかなければいけません。もし他者に秘密鍵が漏れると、秘密鍵に対応した NEO アドレスの資金のコントロールを他者に与えることになってしまうためです。秘密鍵をバックアップをしておくだけでなく、秘密鍵の偶発的な紛失からも保護しておかなければいけません。というのは、もし秘密鍵をなくしてしまうと、秘密鍵を復元することはできず、秘密鍵によってセキュリティを担保していた資金も永遠に失ってしまうためです。

NEO の秘密鍵は、ランダム性がある 256bit の数値で、ある文字列を先ほどの SHA256 関数を使って 16 進数で表現すると次のような 4bit ずつ 64 個の 16 進数で表現されている 256bit の整数になります。

```
936a185caaa266bb9cbe981e9e05cb78cd732b0b3280eb944412bb6f8f8f07af
```

ちなみに先ほどの秘密鍵の $2^{4 \times 64} = 1157920892373161954235709850086879078532699846656405640394575840079131296$ 39936

NEO のネットワーク上では、秘密鍵は、NEO や Gas の送信、または、スマートコントラクトのデプロイをする時のトランザクションに署名するのに使われています。公開鍵と署名生成に使用する秘密鍵は、ある式（後述）に基づいて導き出されており、このときの署名は、秘密鍵を公開することなく、公開鍵に対する正当性を検証できるようになっています。

NEO 所有者は、NEO を送信する時に、公開鍵と署名をトランザクションに記載します。NEO ネットワークに参加している参加者は、この公開鍵と署名を確認することで、トランザクションを検証し有効なものとみなします。

公開鍵と秘密鍵

公開鍵は、秘密鍵から計算できるため、秘密鍵だけ保存しておけばいつでも公開鍵を導き出すことができます。

安全な秘密鍵とは

キーを生成する上で重要かつ一番最初にしなればいけないことは、キー生成の安定的なエントロピー源、つまり十分なランダムさを確保することです。NEO キーを作ることは、"1 から 2^{256} までの間の数字を選ぶ"ということと本質的に同じです。数字を選ぶ厳格な方法は、予測可能であったり再現可能性があったりしない方法です。NEO の秘密鍵は先述したように単なる数値なので、コインや鉛筆、紙だけを使ってランダムに秘密鍵を選ぶことができます。たとえば、コインを 256 回投げて NEO ウォレットで使うランダムな秘密鍵の二進数を作り出すことができます。公開鍵はこの秘密鍵から生成することができます。

3.4 公開鍵の生成

公開鍵暗号では、「秘密鍵から公開鍵を知ることが可能、公開鍵から秘密鍵を知ることは不可能」という特徴をもつ公開鍵をどのように生成するかが大きな課題になります。そこには、楕円曲線を利用した暗号化の技術が関わっています。

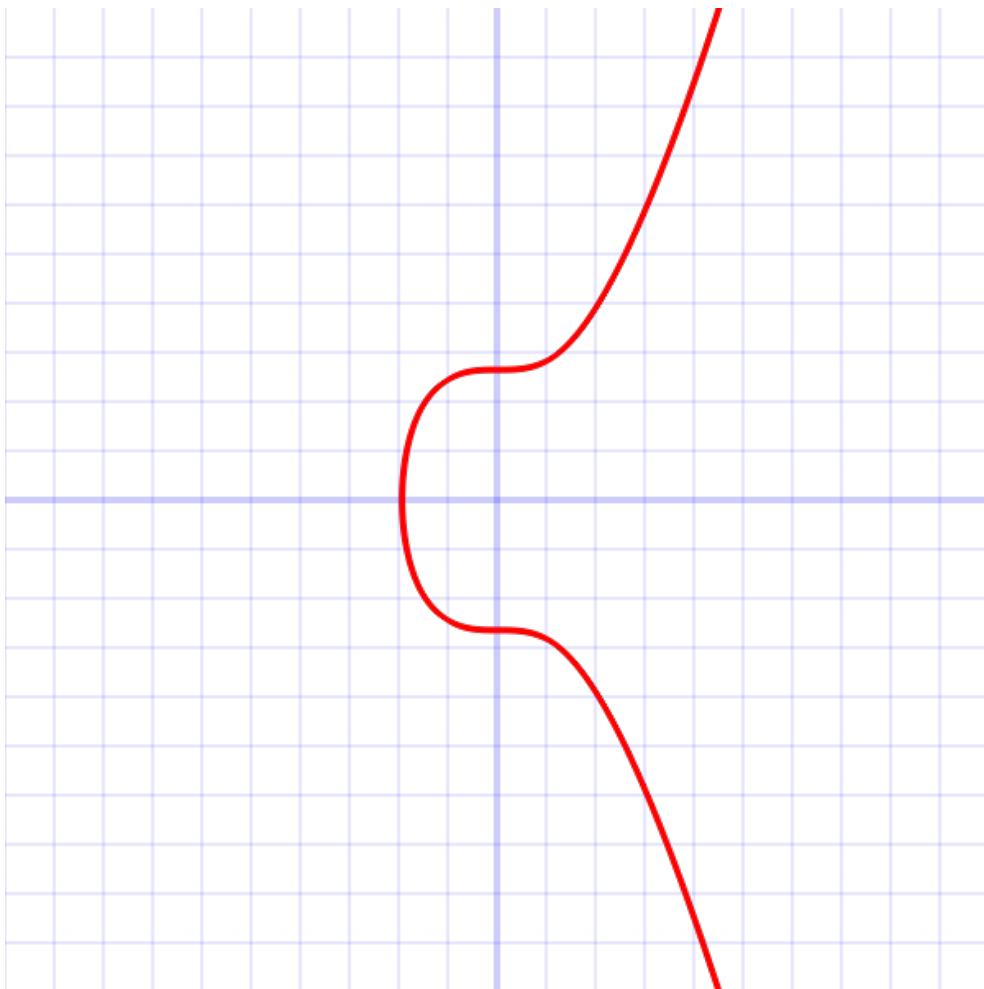
3.4.1 楕円曲線とは

楕円曲線とは次の形の方程式により定義される平面曲線です。

$$y^2 = x^3 + ax + b \quad (1)$$

図: 楕円曲線を表した数式

この a, b の値を変えることで、楕円曲線はさまざまな曲線の形状になります。楕円曲線及び楕円曲線を用いた暗号化にはいくつかのパラメータが存在し、たとえば、Bitcoin で使用されている Secp256k1 では、それらの各パラメーター ($a=0, b=7$) が決まっています。



▲図 3.1 Secp256k1

注意；これは、実数部分における Secp256k1 のグラフですが、実際は体 Z_p 上で定義されるランダムに散らばった小さい点です。

一方、NEO では、楕円曲線の中でも Secp256r1 と呼ばれる楕円曲線を利用しています。これは、各パラメーターが次のようになっており、先ほどの Secp256k1 の楕円曲線と比べて複雑なであることが分かると思います。

$$a = -3$$

$$b = 41058363725152142129326129780047268409114441015993725554835256314039467401291$$

注意：パラメータが異なるので、Secp256r1 のグラフは、図 3.1 とまったく同じではありません。

3.4.2 楕円曲線暗号

楕円曲線暗号 ((Elliptic Curve Cryptography : ECC)) は、楕円曲線上の点に対する加法とスカラ一倍算で表現される離散対数問題をベースに作られた非対称型暗号方式または公開鍵暗号方式です。もう少し噛み砕いていうと、楕円曲線上の離散対数問題 (EC-DLP) の困難性を安全性の根拠とした、楕円曲線を利用した暗号方式のことをさします。ここでいう離散対数問題とは、素数 p と定数 g が与えられたとき、 $y=g^x \bmod p$ を x から計算することは容易だが、 y から x を求めることは困難であるということです。

3.4.3 楕円曲線暗号を利用した公開鍵の生成

秘密鍵を、ランダムに生成された数値 ' k ' とすると、あらかじめ決められた生成元 ' G ' を k に掛けることで楕円曲線上のもう 1 つの点を得ます。このもう 1 つの点は公開鍵 K に対応するものです。次の式では、 k を秘密鍵、 G を生成元、 K は結果として算出される公開鍵となります。

$$K = k * G$$

ベースポイントの G は secp256r1 標準で決められており、NEO での全ての公開鍵に対して常に同じです。したがって、秘密鍵 k が同じであれば、公開鍵は同じになります。また、公開鍵から生成された NEO アドレスから秘密鍵 k を求めることはできないので、アドレスを他人と共有することも可能ですが、この公開鍵から秘密鍵に戻すことができないのは、公開鍵生成プロセスが上述したように数学的に一方向になっているためです。

アドレスと公開鍵

NEO アドレスは公開鍵は同じではありません。NEO アドレスは公開鍵から一方向関数を使って導出されるものです。

3.5 Base58 と Base58Check エンコード

ここから先は、生成した公開鍵をアドレスに変換していく過程になります。公開鍵そのものをアドレスとして使うのではなく、人間がみてもわかりやすい形式に変換する作業になります。多くのコンピュータでは、大きな数字をコンパクトに表すためにいくつかの記号を使うことで 10 以上を基底とするアルファベットと数字を混ぜた表現を使っています。たとえば、伝統的な 10 進数では 0 から 9 までの 10 個の数字を使う一方、16 進数では A から F の文字を使うことで 16 個の数字を使います。16 進数で表される数字は 10 進数で表すよりも短くなります。バイナリデータを email のようなテキストベースの通信で送るために、Base-64 では 26 個の小文字、26 個の大文字、10 個の数字、 "+" や "/" のような 2 種類の文字を使います。Base58 は Bitcoin で使うために開発されたテキストベースのエンコード形式で、他の暗号通貨でも使われています。これはコンパクトな表現、可読性、エラー発見および防止のためです。Base58 は Base64 の部分集合でアルファベットの大文字小文字、数字が使われます。しかし、あるフォントで表示したときに同じように見えて、よく間違えてしまういくつかの文字は省かれています。Base58 は Base64 から 0 (数字の 0)、O (大文字 o)、1 (小文字 L)、I (大文字 i)、記号 "+" や "/" を除いたもの、つまり、(0, O, l, I, +, /) を除いたアルファベットの大文字小文字、数字の集合になっています。

書き間違いや転写間違いをさらに防ぐため、Base58Check はチェックサムを加えた Base58 エンコード形式になっていて NEO で頻繁に使われています。チェックサムは、エンコードされようとしているデータの最後に追加される 4byte です。このチェックサムはエンコードされたデータのハッシュから作られ、転写間違いやタイミング間違いを検出したり防いだりするのに使われます。Base58Check でエンコードされたデータが与えられた場合、デコードソフトウェアはエンコードされようとしているデータのチェックサムを計算し、含まれているチェックサムと比較します。もし 2 つが一致しなかった場合、これはエラーが混入してしまったか Base58Check データが無効だということを示しています。これによって、たとえば、ウォレットが有効な送り先だと判断して受け付けてしまった打ち間違い NEO アドレスを無効と判断し、資金を失ってしまうということを防ぐことができます。

公開鍵から NEO アドレスを作るときに使うアルゴリズムは、Secure Hash Algorithm (SHA) と RACE IntegrityPrimitives Evaluation Message Digest (RIPEMD) で、中でも SHA256 と RIPEMD160 が使われます。公開鍵 K の SHA256 ハッシュを計算し、さらにこの結果の RIPEMD160 ハッシュを計算することで、160bit (20byte) の数字を作り出します。

NEO アドレスは、"Base58Check" と呼ばれる形にエンコードされた状態で通常使わ

れます。"Base58Check"では、58 個の文字 (Base58) とチェックサムを使いますが、これは人間にとって読みやすくしたり、曖昧さを避けたり、転写時のエラーを防いだりするためです。さらに、"Base58Check"で prefix を追加することによって、エンコードしたデータの本体の最初に固有の文字 (A) が現れるため、NEO のアドレスだと判断しやすくなります。

3.6 NEO のアドレスを作ってみよう！

NEO のアドレスは、'A'から始まる 34 桁の文字列の羅列になります。

```
AXJAtEWGNW3EbgJZxoYDe9tL7CafDqdYKY
```

これから、City Of Zion が Golang で実装した Neo-Go のパッケージを元に A から始まる NEO アドレスの生成方法を辿っていきたいと思います。参考にした City Of Zion のパッケージは次のとおりです。

Neo-Go: <https://github.com/CityOfZion/neo-go>

また、本章で使用したコードは、全て github 上に掲載していますので、そちらも合わせてご確認ください。

<https://github.com/keymakers/neoJPbook/blob/master/code/address/main.go>

NEO のアドレス生成は、次の手順で達成されます。

1. 秘密鍵 (PrivateKey) の生成
2. 楕円曲線暗号の仕組みを利用して秘密鍵から公開鍵 (PublicKey) の生成
3. RIPEMD-160 で公開鍵のハッシュを生成
4. version byte をつける (NEO アドレスの'A'に該当する)
5. checksum をつける
6. base58 で encode する

ここでは、すでに準備されている橢円曲線上の点を選びます。

▼ ランダムな秘密鍵の生成

```

c.G.Y, _ = new(big.Int).SetString(
    "4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5", 16,
)
c.N, _ = new(big.Int).SetString(
    "FFFFFFFF00000000FFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551", 16,
)
c.H, _ = new(big.Int).SetString("01", 16)

return c
}

```

実際に実行してみます。

```

func main() {
    privateKey, err := NewPrivateKey()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(hex.EncodeToString(privateKey.b))
}

```

秘密鍵が生成されたことがわかります。前述したとおり 256bit の秘密鍵は、16進数では 64 桁で表示されます。

```
49dac28b59c9127ede34be6ec3a2ab68475e4619c6f289eef67b0b6ba0014ca6
```

3.6.1 秘密鍵から公開鍵の生成

```

// PublicKey derives the public key from the private key.
func (p *PrivateKey) PublicKey() ([]byte, error) {
var (
    c = NewEllipticCurve()
    q = new(big.Int).SetBytes(p.b)
)

point := c.ScalarBaseMult(q)
if !c.IsOnCurve(point) {
    return nil, errors.New("failed to derive public key using elliptic curve")
}

bx := point.X.Bytes()
padded := append(
    bytes.Repeat(
        []byte{0x00},
        32-len(bx),
    ),
)

```

```

        bx...,
)

prefix := []byte{0x03}
if point.Y.Bit(0) == 0 {
prefix = []byte{0x02}
}
b := append(prefix, padded...)

return b, nil
}

```

実際にこの関数を実行してみます。

```

func main() {
privateKey, err := NewPrivateKey()
if err != nil {
log.Fatal(err)
}

publicKey, err := privateKey.PublicKey()
if err != nil{
log.Fatal(err)
}

fmt.Println(hex.EncodeToString(publicKey))
}

```

公開鍵が生成されました。もし、これが Secp256r1 で定義された楕円曲線上にない場合、エラーが帰ってくるはずなので、正しい公開鍵だと確認できます。

```
0222c1b83dcfc3565a09e6beede84732ee14b9c8e2bb5effb2d1c53f575fce16a
```

3.6.2 RIPEMD-160 で公開鍵のハッシュを生成

```

// Signature creates the signature using the private key.
func (p *PrivateKey) Signature() ([]byte, error) {
b, err := p.PublicKey()
if err != nil {
    return nil, err
}

b = append([]byte{0x21}, b...)
b = append(b, 0xAC)

sha := sha256.New()

```

```
sha.Write(b)
hash := sha.Sum(nil)

ripemd := ripemd160.New()
ripemd.Reset()
ripemd.Write(hash)
hash = ripemd.Sum(nil)

return hash, nil
}
```

関数名は、signature となっていますが、関数内で実行していることは、秘密鍵から公開鍵を生成（前述）して、それを元に公開鍵の 20byte のハッシュを生成しています。実際にこの関数を実行してみます。

```
func main() {
//ランダムなプライベートキーの生成
privateKey, err := NewPrivateKey()
if err != nil {
log.Fatal(err)
}
publicKeyHash, err := privateKey.Signature()
if err != nil {
log.Fatal(err)
}
fmt.Println(hex.EncodeToString(publickeyHash))
}
```

20byte の公開鍵ハッシュが生成されました。

```
2dd673f4cdf38741eb0e73c51025e6002fcc44b5
```

3.6.3 base58 でエンコード

```
// Address derives the public NEO address that is coupled with the private
// key, and returns it as a string.
func (p *PrivateKey) Address() (string, error) {
b, err := p.Signature()
if err != nil {
return "", err
}

b = append([]byte{0x17}, b...)
sha := sha256.New()
```

```

sha.Write(b)
hash := sha.Sum(nil)

sha.Reset()
sha.Write(hash)
hash = sha.Sum(nil)

b = append(b, hash[0:4]...)

address := Base58Encode(b)

return address, nil
}

```

先ほどの'Signature'関数以降をみてみると、base58 における A に当たる 0x17 を先頭につけるとともに、checksum として、4byte 分を後半にくっつけていることがわかります。最後に、これを base58 でエンコードすることで、NEO のアドレスが生成されます。

3.6.4 最終コード

```

package main

import(
    "fmt"
    "log"
    "io"
    "crypto/rand"
    "math/big"
    "crypto/sha256"
    "golang.org/x/crypto/ripemd160"
    "bytes"
    "errors"
    "encoding/hex"
)

type (
    // EllipticCurve represents the parameters of a short Weierstrass equation
    // elliptic curve.
    EllipticCurve struct {
        A *big.Int
        B *big.Int
        P *big.Int
        G ECPPoint
        N *big.Int
        H *big.Int
    }

    // ECPPoint represents a point on the EllipticCurve.
    ECPPoint struct {
        X *big.Int
        Y *big.Int
    }
)

```

```

)

type PrivateKey struct {
b []byte
}

func main() {
//ランダムなプライベートキーの生成
privateKey, err := NewPrivateKey()
if err != nil {
log.Fatal(err)
}
fmt.Println(hex.EncodeToString(privateKey.b))

//プライベートキーからアドレスの生成
//1. PrivateKey → PublicKey
//2. PublicKeyHash の生成 (20byte に圧縮)
//3. version の追加
//4. checksum の追加
//5. base58 でエンコード
address, err := privateKey.Address()
if err != nil {
log.Fatal(err)
}
fmt.Println(address)
}

func NewPrivateKey() (*PrivateKey, error) {
c := NewEllipticCurve()
b := make([]byte, c.N.BitLen()/8+8)
if _, err := io.ReadFull(rand.Reader, b); err != nil {
return nil, err
}

d := new(big.Int).SetBytes(b)
d.Mod(d, new(big.Int).Sub(c.N, big.NewInt(1)))
d.Add(d, big.NewInt(1))

p := &PrivateKey{b: d.Bytes()}
return p, nil
}

// NewEllipticCurve returns a ready to use EllipticCurve with preconfigured
// fields for the NEO protocol.
func NewEllipticCurve() EllipticCurve {
c := EllipticCurve{}

c.P, _ = new(big.Int).SetString(
    "FFFFFFFF00000010000000000000000000000000000000FFFFFFFFFFFFFF", 16,
)
c.A, _ = new(big.Int).SetString(
    "FFFFFFFF000000100000000000000000000000000000000FFFFFFFFFFFFFFFC", 16,
)
c.B, _ = new(big.Int).SetString(
    "5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B", 16,
)
c.G.X, _ = new(big.Int).SetString(

```

```
        "6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296", 16,
    )
c.G.Y, _ = new(big.Int).SetString(
    "4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5", 16,
)
c.N, _ = new(big.Int).SetString(
    "FFFFFFFFF00000000FFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551", 16,
)
c.H, _ = new(big.Int).SetString("01", 16)

return c
}

// Address derives the public NEO address that is coupled with the private
// key, and returns it as a string.
func (p *PrivateKey) Address() (string, error) {
b, err := p.Signature()
if err != nil {
return "", err
}

b = append([]byte{0x17}, b...)

sha := sha256.New()
sha.Write(b)
hash := sha.Sum(nil)

sha.Reset()
sha.Write(hash)
hash = sha.Sum(nil)

b = append(b, hash[0:4]...)

address := Base58Encode(b)

return address, nil
}

// Signature creates the signature using the private key.
func (p *PrivateKey) Signature() ([]byte, error) {
b, err := p.PublicKey()
if err != nil {
return nil, err
}

b = append([]byte{0x21}, b...)
b = append(b, 0xAC)

sha := sha256.New()
sha.Write(b)
hash := sha.Sum(nil)

ripemd := ripemd160.New()
ripemd.Reset()
ripemd.Write(hash)
hash = ripemd.Sum(nil)

return hash, nil
}
```

```

// PublicKey derives the public key from the private key.
func (p *PrivateKey) PublicKey() ([]byte, error) {
    var (
        c = NewEllipticCurve()
        q = new(big.Int).SetBytes(p.b)
    )

    point := c.ScalarBaseMult(q)
    if !c.IsOnCurve(point) {
        return nil, errors.New("failed to derive public key using elliptic curve")
    }

    bx := point.X.Bytes()
    padded := append(
        bytes.Repeat(
            []byte{0x00},
            32-len(bx),
        ),
        bx...,
    )

    prefix := []byte{0x03}
    if point.Y.Bit(0) == 0 {
        prefix = []byte{0x02}
    }
    b := append(prefix, padded...)

    return b, nil
}

//Base58Encode encodes a byte slice to be a base58 encoded string.
func Base58Encode(bytes []byte) string {
    var (
        lookupTable = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
        x           = new(big.Int).SetBytes(bytes)
        r           = new(big.Int)
        m           = big.NewInt(58)
        zero        = big.NewInt(0)
        encoded     string
    )

    for x.Cmp(zero) > 0 {
        x.QuoRem(x, m, r)
        encoded = string(lookupTable[r.Int64()]) + encoded
    }

    return encoded
}

// ScalarBaseMult computes Q = k * G on EllipticCurve ec.
func (c *EllipticCurve) ScalarBaseMult(k *big.Int) (Q ECPoint) {
    return c.ScalarMult(k, c.G)
}

// ScalarMult computes Q = k * P on EllipticCurve ec.
func (c *EllipticCurve) ScalarMult(k *big.Int, P ECPoint) (Q ECPoint) {
    // Implementation based on pseudocode here:
    // https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication
}

```

```

var R0 ECPPoint
var R1 ECPPoint

R0.X = nil
R0.Y = nil
R1.X = new(big.Int).Set(P.X)
R1.Y = new(big.Int).Set(P.Y)

for i := c.N.BitLen() - 1; i >= 0; i-- {
    if k.Bit(i) == 0 {
        R1 = c.Add(R0, R1)
        R0 = c.Add(R0, R0)
    } else {
        R0 = c.Add(R0, R1)
        R1 = c.Add(R1, R1)
    }
}
return R0
}

// IsOnCurve checks if point P is on EllipticCurve ec.
func (c *EllipticCurve) IsOnCurve(P ECPPoint) bool {
if c.IsInfinity(P) {
    return false
}
lhs := mulMod(P.Y, P.Y, c.P)
rhs := addMod(
    addMod(
        expMod(P.X, big.NewInt(3), c.P),
        mulMod(c.A, P.X, c.P), c.P),
    c.B, c.P)

if lhs.Cmp(rhs) == 0 {
    return true
}
return false
}

// IsInfinity checks if point P is infinity on EllipticCurve ec.
func (c *EllipticCurve) IsInfinity(P ECPPoint) bool {
return P.X == nil && P.Y == nil
}

// addMod computes z = (x + y) % p.
func addMod(x *big.Int, y *big.Int, p *big.Int) (z *big.Int) {
z = new(big.Int).Add(x, y)
z.Mod(z, p)
return z
}

// subMod computes z = (x - y) % p.
func subMod(x *big.Int, y *big.Int, p *big.Int) (z *big.Int) {
z = new(big.Int).Sub(x, y)
z.Mod(z, p)
return z
}

// mulMod computes z = (x * y) % p.

```

```

func mulMod(x *big.Int, y *big.Int, p *big.Int) (z *big.Int) {
    n := new(big.Int).Set(x)
    z = big.NewInt(0)

    for i := 0; i < y.BitLen(); i++ {
        if y.Bit(i) == 1 {
            z = addMod(z, n, p)
        }
        n = addMod(n, n, p)
    }

    return z
}

// expMod computes z = (x^e) % p.
func expMod(x *big.Int, y *big.Int, p *big.Int) (z *big.Int) {
    z = new(big.Int).Exp(x, y, p)
    return z
}

// invMod computes z = (1/x) % p.
func invMod(x *big.Int, p *big.Int) (z *big.Int) {
    z = new(big.Int).ModInverse(x, p)
    return z
}

// Add computes R = P + Q on EllipticCurve ec.
func (c *EllipticCurve) Add(P, Q ECPoint) (R ECPoint) {
    // See rules 1-5 on SEC1 pg.7 http://www.secg.org/collateral/sec1_final.pdf
    if c.IsInfinity(P) && c.IsInfinity(Q) {
        R.X = nil
        R.Y = nil
    } else if c.IsInfinity(P) {
        R.X = new(big.Int).Set(Q.X)
        R.Y = new(big.Int).Set(Q.Y)
    } else if c.IsInfinity(Q) {
        R.X = new(big.Int).Set(P.X)
        R.Y = new(big.Int).Set(P.Y)
    } else if P.X.Cmp(Q.X) == 0 && addMod(P.Y, Q.Y, c.P).Sign() == 0 {
        R.X = nil
        R.Y = nil
    } else if P.X.Cmp(Q.X) == 0 && P.Y.Cmp(Q.Y) == 0 && P.Y.Sign() != 0 {
        num := addMod(
            mulMod(big.NewInt(3),
                   mulMod(P.X, P.X, c.P), c.P),
            c.A, c.P)
        den := invMod(mulMod(big.NewInt(2), P.Y, c.P), c.P)
        lambda := mulMod(num, den, c.P)
        R.X = subMod(
            mulMod(lambda, lambda, c.P),
            mulMod(big.NewInt(2), P.X, c.P),
            c.P)
        R.Y = subMod(
            mulMod(lambda, subMod(P.X, R.X, c.P), c.P),
            P.Y, c.P)
    } else if P.X.Cmp(Q.X) != 0 {

```

```

num := subMod(Q.Y, P.Y, c.P)
den := invMod(subMod(Q.X, P.X, c.P), c.P)
lambda := mulMod(num, den, c.P)
R.X = subMod(
    subMod(
        mulMod(lambda, lambda, c.P),
        P.X, c.P),
    Q.X, c.P)
R.Y = subMod(
    mulMod(lambda,
        subMod(P.X, R.X, c.P), c.P),
    P.Y, c.P)
} else {
    panic(fmt.Sprintf("Unsupported point addition: %v + %v", P, Q))
}

return R
}

```

実際に、最終コードを実行してみると、次のような NEO のアドレスと秘密鍵が生成されました。

```

686adb9235d948078b1aa098e50f369fe4df90e96b3bd5abfb8c0cc972b4d359
AUFkwd7fqQo7uLVoygvFBys9kTCharg9Tr

```

このアドレスが機能するか、実際に import してみて NEO の送金をしてみたいと思います。今回使用する wallet は、CityOfZion が作っている neon-wallet です。

The screenshot shows a transaction record on the neoscan website. At the top, it says "Transaction Information". Below that, it shows a "Contract Hash: 1d3783c59b43197ad33a9ce1868dfa8c365fea3ed4de4e5050cb6c724a8821ca". It has a link to "全ての取引に戻る". The transaction details are as follows:

送信元	送信先
AHgMNEOMTLim7eqXr15Tw7vdzAFJydkDN 1 NEO	AUFkwd7fqQo7uLVoygvFBys9kTCharg9Tr 1 NEO

Below the table, there are two rows of information:

時間	ネットワーク料金	システム料金
38 seconds ago	0	0

At the bottom, it says "ブロックに含む 2,996,799" and "サイズ 202 バイツ".

図: neoscan の取引履歴

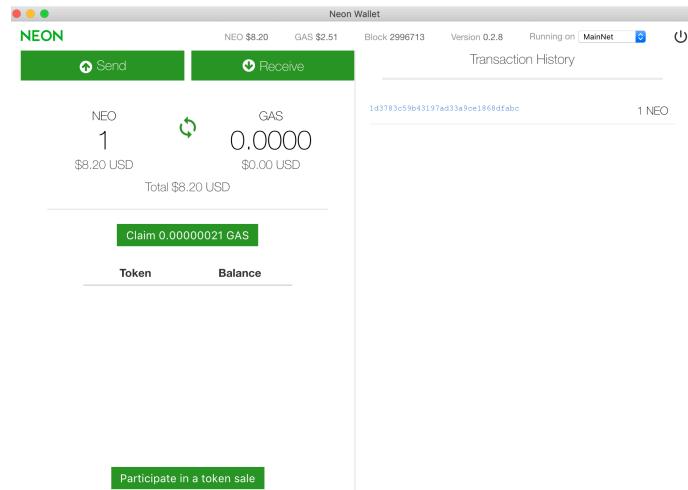


図: neon-wallet の中身

無事送信することができました！ 取引履歴は、neoscan で確認することができます。

アドレスの最初の文字「A」の理由

NEO のアドレスが A から始まるのは、NEO の前身である AntShare の A からきているといわれています。

第 4 章

ウォレット

4.1 この章でカバーすること

NEO を管理するにあたっていくつかの方法を紹介します。

4.1.1 ハードウェアウォレットで管理する方法

NEO などの暗号資産を管理する方法は大きく分けて 3 種類あります。Ledger 社などが提供しているハードウェアウォレットで管理する方法、bitFlyer や Coinbase のような取引所のアカウントで管理する方法、Ginco や Coinbase Wallet などが提供しているウォレットで管理する方法です。その中でもっとも安全性が高いのがハードウォレットで管理する方法です。これは、ハードウェアウォレットで管理することによりオフライン環境で資産が管理できるようになるので、プライベートキーを安全に守ることができます。

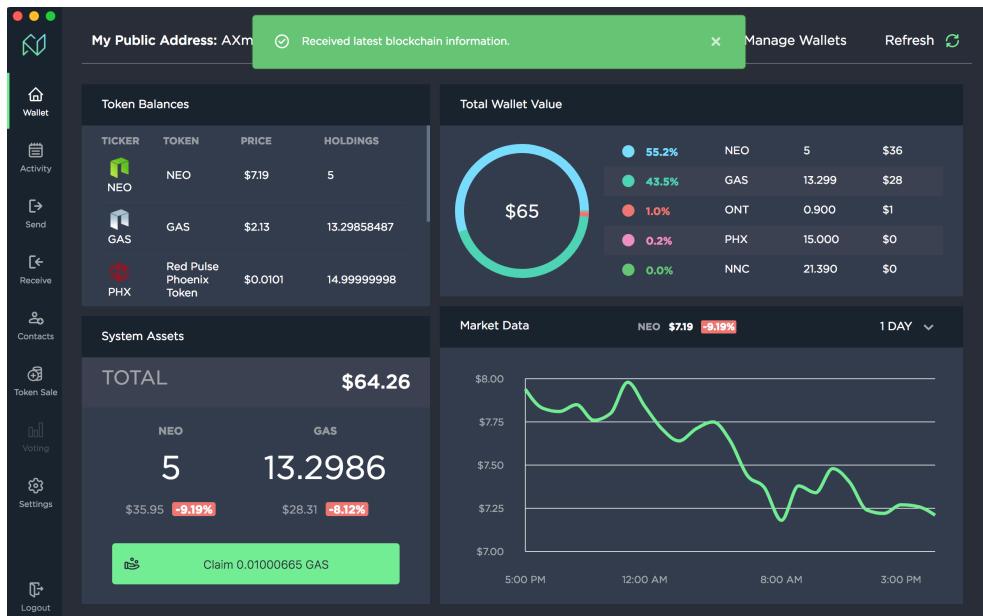
4.1.2 NEON Wallet

NEON Wallet はデスクトップウォレットです。NEON ウォレットでは NEO と GAS の送金、保有、受け取りを行うことができます。

現在の NEON ウォレットでできることは次のとおりです。

- ウォレットを作成する
- プライベートキーの秘匿化
- ウォレットアカウントのインポートとエクスポート
- 残高の表示
- GAS と NEO の価格表示
- NEO,GAS,NEP5 トーケンの送金

- ネットワークの変更
- 一斉送金
- NEP9 での QR サポート
- NEO トーカンセールへの参加
- など



▲図 4.1 ブロック同期 1

インストール

NEON ウォレットは、次の URL よりダウンロードすることができます。

<https://github.com/CityOfZion/neon-wallet/releases>

NEON を手動でインストールをする際に、Node v6.11.0 と yarn が必要となります。実際に開発を行うときには、プロジェクトのルートディレクトリーにて、

- Setup
 - yarn install
 - * node のディペンデンシーをインストールします。エラーが発生してし

またの場合には、次のリンクを確認することでエラーの発生源を特定することができます。

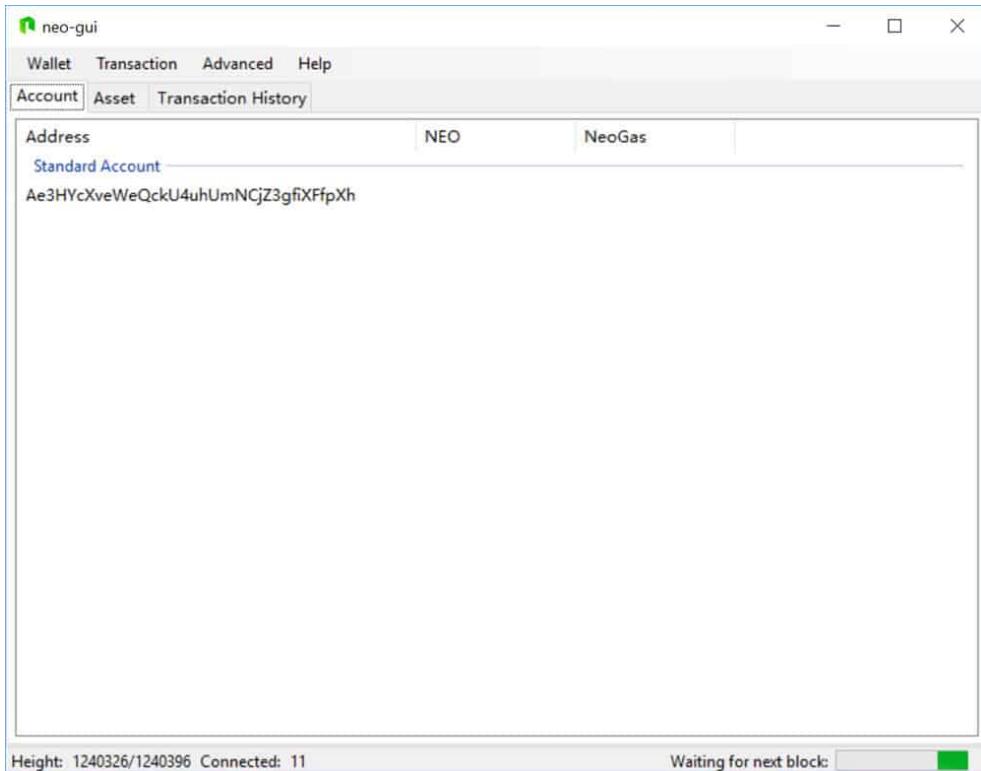
<https://github.com/node-hid/node-hid#compiling-from-source>

- ./node_modules/.bin/electron -v
 - バージョンが 1.8.4 であることを確認します。起動するまでに 10~15 秒ほど の時間がかかります。
- Developing
 - yarn dev
- Running
 - yarn assets
 - yarn start
- Testing
 - yarn test もしくは、yarn run test-watch でテストを行います。

NEON ウォレットは開発アップデートが順次更新されていくので、github のアカウントより確認ください。

<https://github.com/City0fZion/neon-wallet>

4.1.3 NEO GUI

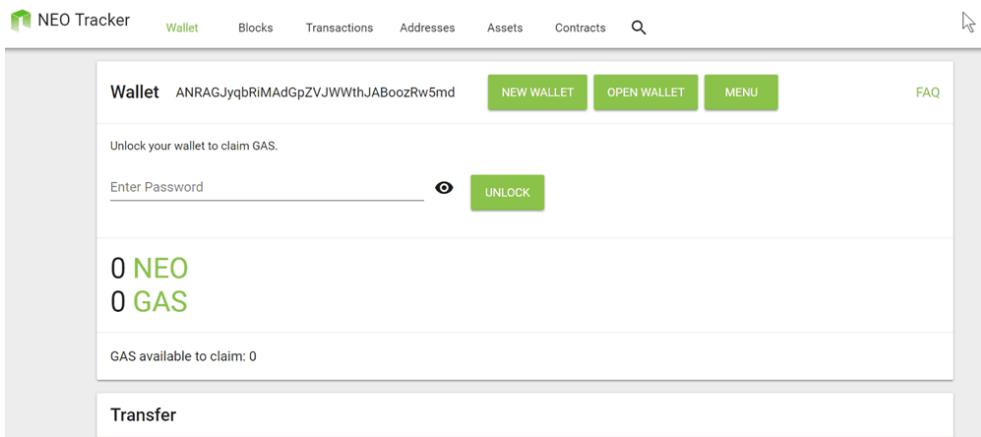


▲図 4.2 NEO-GUI

NEO GUI は NEO 開発者向けのデスクトップウォレットです。これによってユーザーはグラフィカルユーザーインターフェーズとやり取りを行いながら開発をすることができます。現在サポートされている OS は、windows のみです。NEO-GUI のインストールは次のリンクより可能です。

<https://github.com/neo-project/neo-gui/releases>

4.1.4 NEO Tracker

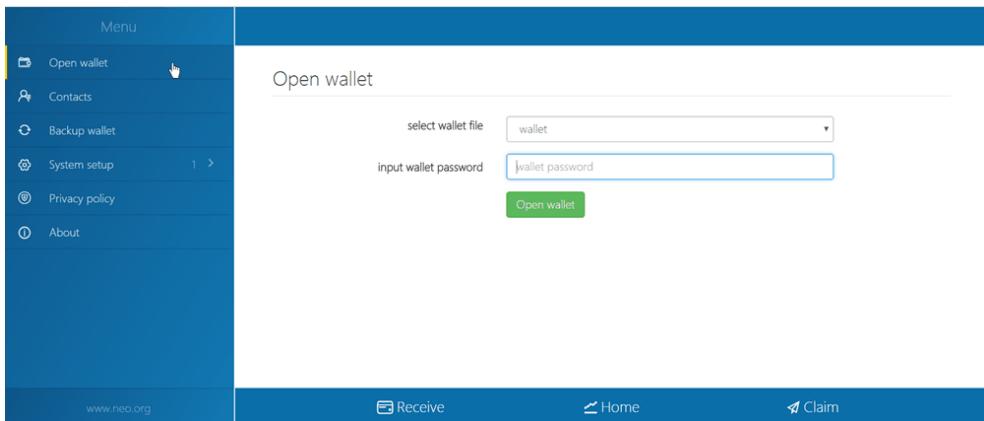


▲図 4.3 NEO-Tracker

NEO Tracker ウォレットはブラウザベースのウォレットです。Javascript によって開発されており、オープンソースプロジェクトとなっています。NEO と GAS の受信、送信、保存ができ、ユーザーフレンドリーなアプリケーションとなっています。NEO Tracker ウォレットの使用は次のリンクより可能です。

<https://neotracker.io/>

4.1.5 NEO ウォレット

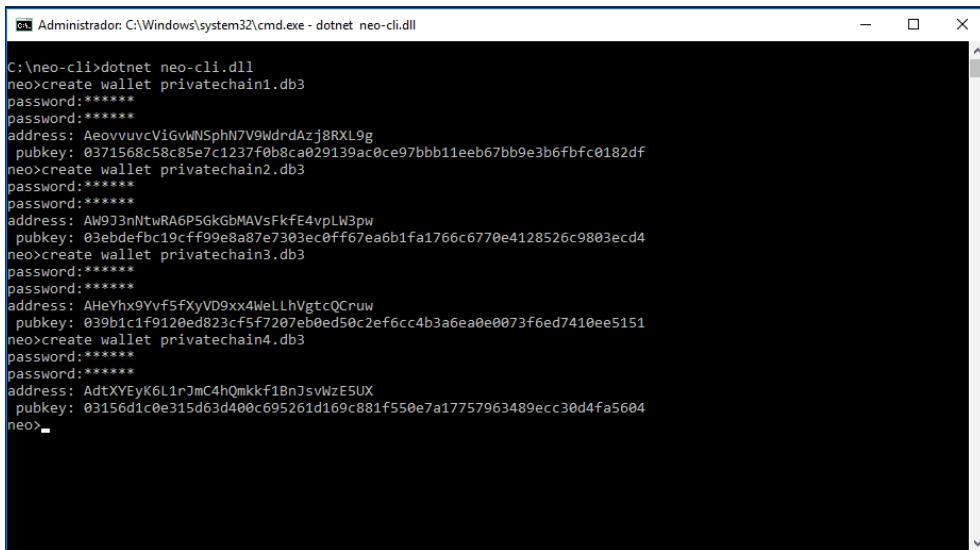


▲図 4.4 NEO ウォレット

NEO ウォレットは NEO の開発者コミュニティによって開発されているブラウザ wallet です。プライベートキーをローカル環境で管理できるので、安全性が比較的高くなっています。NEO ウォレットの使用は次のリンクより可能です。

<https://neowallet.cn/>

4.1.6 NEO CLI



```
C:\neo-cli>dotnet neo-cli.dll
neo>create wallet privatechain1.db3
password:*****
password:*****
address: AeovuvvcViGvWNsPhN7V9WdrdAzj8RXL9g
pubkey: 0371568c58c85e7c1237f0bb8ca029139ac0ce97bbb11eeb67bb9e3b6fbfc0182df
neo>create wallet privatechain2.db3
password:*****
password:*****
password:*****
address: AW9J3nNtwRA6P5GkGbMAVsFkfe4vpLW3pw
pubkey: 03ebdefbc19cff99e8a87e7303ec0ff67ea6b1fa1766c6770e4128526c9803ecd4
neo>create wallet privatechain3.db3
password:*****
password:*****
password:*****
address: AHeYhx9Yvf5fXyVD9xx4WeLLhVgtcQCruw
pubkey: 039b1c1f9120ed823cf5f7207eb0ed50c2ef6cc4b3a6ea0e0073f6ed7410ee5151
neo>create wallet privatechain4.db3
password:*****
password:*****
address: AdtXYeyK6L1rJmC4hQmkkf18nJsvWzE5UX
pubkey: 03156d1c0e315d63d400c695261d169c881f550e7a17757963489ecc30d4fa5604
neo>~
```

▲図 4.5 NEO CLI

NEO CLI は NEO の開発者コミュニティによって開発されています。こちらはコマンドライン上での操作となるため、上級者むけのウォレットです。NEO CLI のインストールは次のリンクより可能です。

<https://github.com/neo-project/neo-cli/releases>

第 5 章

トランザクション

トランザクションとは、コインやトークン、アセットなどの取り引きを指します。NEO のネットワークプロトコルはビットコインに似ていますが、ブロックやトランザクションのデータ構造は大きく異なります。

5.1 ブロック

トランザクションやアセットなど、ネットワーク全体のデータはブロックチェーンに格納されます。ブロックはトランザクションをまとめて格納でき、トランザクションがブロックに格納される際、ブロックの検証用スクリプトによって検証が行われます。検証のあと、ブロックはコンセンサスアルゴリズム（dBFT）を経てブロックチェーンに格納されます。

ブロックの構造を表 5.1 に示します。

ブロックは前のブロックのハッシュ値を持っており、ブロックチェーン上でブロック同士の「チェーン」が形成されます。ブロックのハッシュ値を計算するとき、ブロック全体を計算するのではなく、Version, PrevBlock, MerkleRoot, Timestamp, Height, Nonce, NextMiner の 7 つの値のみ計算されます。MerkleRoot には全てのトランザクションのハッシュ値が含まれており、トランザクションの変更はそのブロックのハッシュの値に影響します。

5.2 トランザクションの内容

トランザクションの内容をみてみましょう。（表 5.2）トランザクションの内容を大きく示すと、Type, Attributes, Input, Output となります。その他にもプロトコルやライブラリの中で、Scripts, Witness, Exclusive Data などのデータをもつことがあります。

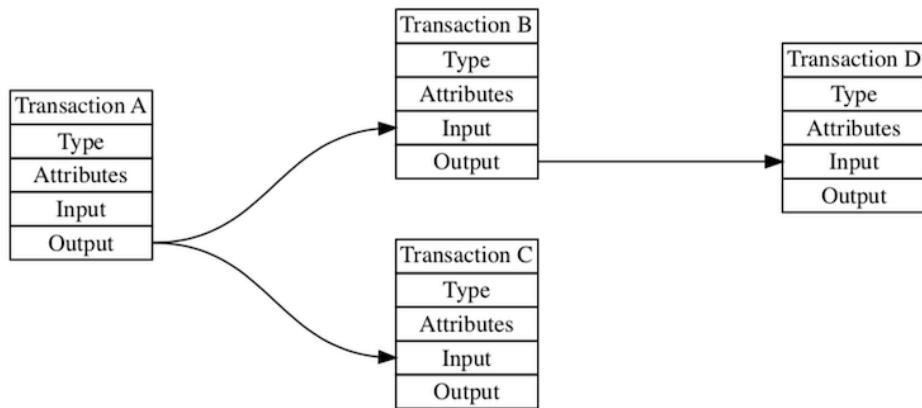
▼表 5.1 ブロック

サイズ	名称	データ型	説明
4	Version	uint32	ブロックのバージョンを示します。現在は 0 が使用されています。
32	PrevBlock	uint256	前のブロックのハッシュ値を示します。
32	MerkleRoot	uint256	トランザクションリストのルートのハッシュ値
4	Timestamp	uint32	タイムスタンプ。Timestamp は、前のブロックよりも後の値が入る必要があります。一般に、前後のブロックでタイムスタンプの差は約 15 秒程度となります。
4	Height	uint32	ブロックの高さ。Height は、前のブロックの Height に 1 を加えた値が入ります。
8	Nonce	uint64	ランダムな数値
20	NextMiner	uint160	次の採掘者（マイナー）のコントラクトアドレス
1	-	uint8	固定値 (1)
x	Script	script	ブロックの検証用のスクリプト
x*x	Transactions	tx[]	トランザクションのリスト。リストの最初には必ず後述の MinerTransaction が入ります。

NEO ではコインの管理方法として、ビットコインなどでも使用されている UTXO (Unspent Transaction Output) という仕組みを利用しています。UTXO とは、ブロックチェーン上でまだ使用されていないトランザクションのアウトプットであり、残高を示します。簡単にいうと、取り引きデータから残高を算出する方法になります。図 5.1 はトランザクションの流れの一例を示しています。過去のトランザクションのアウトプットが、次のトランザクションのインプットとなります。

▼表 5.2 トランザクションの内容

名前	説明
Type	トランザクションのタイプ
Attributes	トランザクションの属性
Input	トランザクションの入力
Output	トランザクションの出力
(Scripts)	NEO のネットワークプロトコルにおいて、トランザクションの検証に使用されるスクリプト。Neon-js においては Witness に当たります。
(Exclusive Data)	Neon-js において、各トランザクションの Type ごとに持つユニークな値。



▲図 5.1 UTXO

5.2.1 トランザクションのタイプ (Type)

トランザクションのタイプには表 5.3 のような種類があります。
いくつか代表的なトランザクション^{*1}について説明します。

^{*1} 参考 : validator の候補者として登録を行う EnrollmentTransaction は廃止予定になっています。また、VotingTransaction/AgencyTransaction も neo-python ではまだサポートされていますが、NEO-GUI などの C# で実装されているクライアントでは 2016 年ごろから廃止されました。

▼表 5.3 トランザクションのタイプ

名称	説明
MinerTransaction	コンセンサストランザクション、バイトチャージ割当を行う
IssueTransaction	アセット分配を行います。
ClaimTransaction	GAS 配布を行います。
RegisterTransaction	アセットを登録します。version2 以降は Asset.CreateAsset 関数に置き換わっています。
ContractTransaction	最も一般的に使用されるコントラクトトランザクションです。
StateTransaction	validator の状態の取得/更新時に使用されるトランザクションです。
PublishTransaction	スマートコントラクトトランザクション。version2 以降は Contract.Create 関数に置き換わっています。
InvocationTransaction	スマートコントラクトトランザクションの呼び出しを行います。

MinerTransaction は、各ブロックの Transactions のリストの最初に格納され、そのブロックの全ての手数料をブロックの検証を行うバリデーターの報酬にします。Nonce (uint32) をもち、Nonce の値はハッシュ値の衝突を避けるのに使用します。(表 5.4)

▼表 5.4 MinerTransaction

サイズ	フィールド	データ型	説明
4	Nonce	uint32	ランダムな数値

次に、IssueTransaction についてです。アセットの管理者は、IssueTransaction によってアセットを作成し、NEO のブロックチェーンに登録して任意のアドレスへ送ることができます。発行されているアセットが NEO の場合、トランザクションの手数料はかかりません。IssueTransaction も Nonce の値を持ちます。(表 5.5)

▼表 5.5 IssueTransaction

サイズ	フィールド	データ型	説明
4	Nonce	uint32	ランダムな数値

ClaimTransaction は分配するための GAS を Input として持ちます。(表 5.6)

NEO ブロックチェーンに新しいアセットを登録する場合は、RegisterTransaction (表 5.7) を使用します。RegisterTransaction は、NEO の version2 以降は Asset.CreateAsset

▼表 5.6 ClaimTransaction

サイズ	フィールド	データ型	説明
34*x	Claims	tx_in[]	分配する GAS

関数（リスト 5.1、表 5.8）に置き換わっています。RegisterTransaction の構造を示します。

▼表 5.7 RegisterTransaction

サイズ	フィールド	データ型	説明
1	AssetType	uint8	アセットタイプ
x	Name	varstr	アセットの名前
8	Amount	int64	アセットの金額（正の値をとる場合は制限モード、 -10^{-8} の場合は無制限モード）
33	Issuer	ec_point	発行者の PublicKey
20	Admin	uint160	発行者のコントラクトのハッシュ値

▼リスト 5.1 Asset.CreateAsset

```
public static extern Neo.SmartContract.Framework.Services.Neo.Asset
Create(byte asset_type,
       string name,
       long amount,
       byte precision,
       byte[] owner,
       byte[] admin,
       byte[] issuer)
```

▼表 5.8 Asset.CreateAsset

名称	データ型	説明
asset_type	byte	アセットタイプ
name	string	アセットの名前
amount	long	アセットの合計。入力値は、100,000,000 を掛けた値。
precision	byte	アセットが取ることができる小数点以下の桁数。
owner	byte[]	長さが 33 のバイト配列の、所有者のパブリックキー
admin	byte[]	長さが 20 のバイト配列の、管理者のコントラクトアドレス
issuer	byte[]	長さが 20 のバイト配列の、発行者のコントラクトアドレス
戻り値	Asset	新たに登録されたアセット

AssetType には表 5.9 の値が格納されます。それぞれのアセットには固有の制限があ

り、NEO および Gas はシステムの組み込みアセットのため Genesis ブロック（高さが 0 のブロック）でのみ作成することができます。equity のようなアセットを作成するときは、アセットの合計額を制限し、送信者と受信者の両方がトランザクションに署名しなくてはなりません。Currency のアセット作成時には、アセットの合計額は制限することができません。

▼表 5.9 AssetType

値	名前	説明
0x40	CreditFlag	クレジットフラグ
0x80	DutyFlag	Duty フラグ。このフラグが ON の場合、検証が行われます。
0x00	SystemShare	NEO
0x01	SystemCoin	NEO GAS
0x08	Currency	通貨
0x60	Token	カスタムアセットのトークン。CreditFlag 0x20 で計算されます
0x90	Share	equity/シェア。DutyFlag 0x10 で計算されます
0x98	Invoice	インボイス。DutyFlag 0x18 で計算されます

ContractTransaction は、NEO や GAS などのアセットを送るためのもっとも一般的なトランザクションです。Inputs と Outputs のトランザクションフィールドは、このトランザクションにおいて重要です。（たとえば、NEO をどれだけ、どのアドレスに対して送信するかが設定されます。）

PublishTransaction は、version2 以降において Contract.Create 関数（リスト 5.2、表 5.10）に置き換わっています。

▼リスト 5.2 Contract.Create

```
public static extern Neo.SmartContract.Framework.Services.Neo.Contract
CreateContract(byte[] script,
               byte[] parameter_list,
               byte return_type,
               bool need_storage,
               string name,
               string version,
               string author,
               string email,
               string description)
```

InvocationTransaction 表 5.11 に示します。

▼表 5.10 Contract.Create

名称	データ型	説明
script	byte[]	コントラクトのバイトコード
parameter_list	byte[]	パラメータリスト
return_type	byte	戻り値の型
need_storage	bool	コントラクトが永続化ストアを必要とするかどうか
name	string	コントラクトの名前
version	string	バージョン
author	string	署名者の名前
email	string	署名者の e メールアドレス
description	string	コントラクトの説明
戻り値	Contract	新たに作成されたコントラクト

▼表 5.11 InvocationTransaction

サイズ	フィールド	データ型	説明
x	Script	uint8[]	スマートコントラクトのスクリプトにより呼び出されます
8	Gas	int64	スマートコントラクトを実行するのに必要なコストです

5.2.2 トランザクションの属性 (Attributes)

トランザクションの属性は、トランザクションが使用される目的に合わせたデータを格納している Usage と、目的外のデータ用に確保された Data の 2 つの領域に分類されます。(表 5.12)

▼表 5.12 トランザクションの属性

サイズ	名称	データの型	説明
1	Usage	uint8	トランザクションの目的に関連したデータ
0 1	length	uint8	データ長 (Usage の値によっては省略されます)
length	Data	uint8[length]	トランザクションの目的外のデータ

Usage では、表 5.13 のようなデータが格納されます。

ContractHash、ECDH02-ECDH03、Hash1-Hash15において、データ長は 32 固定であり length フィールドは省略されます。CertUrl、DescriptionUrl、Description、Remark-Remark15において、データ長を 255 以下の長さで明確に定義する必要があり、省略することはできません。

▼表 5.13 トランザクションの用途

値	名称	説明
0x00	ContractHash	コントラクトのハッシュ値
0x02-0x03	ECDH02-ECDH03	ECDH 鍵交換のための公開鍵
0x20	Script	トランザクションの追加のバリデーション
0x30	Vote	投票に使用します
0x80	CertUrl	証明書の URL アドレス
0x81	DescriptionUrl	トランザクションの説明 URL
0x90	Description	簡単な説明
0xa1-0xaf	Hash1-Hash15	カスタムハッシュ値を保管するのに使用
0xf0-0xff	Remark-Remark15	注意事項

5.2.3 トランザクションの入力

トランザクションの入力は、以前に存在していた別のトランザクションの出力から来なければなりません。1つ前のトランザクションの出力を確認するために、Input では次の2つのデータを持ちます。

1. 参照される1つ前のトランザクションのハッシュ (PrevHash)
2. 1つ前のトランザクションの出力リストにある、この入力のインデックス (PrevIndex)

5.2.4 トランザクションの出力

トランザクションの出力には、表 5.14 に示す3つのフィールドがあります。各トランザクションは最大 65536 の Output をもつことができます。

▼表 5.14 トランザクションの出力

サイズ	フィールド	データ型	説明
32	AssetId	uint256	アセット ID
8	Value	int64	アセットの値
20	ScriptHash	uint160	宛先のアドレス

NEO3.0 では Transaction の種類を簡素化する方向へ

今まで見てきたように、NEO 2.x では、次のようにさまざまな種類のトランザクションがあります。

- MinerTransaction
- IssueTransaction
- ClaimTransaction
- EnrollmentTransaction
- RegisterTransaction
- ContractTransaction
- StateTransaction
- PublishTransaction
- InvocationTransaction

しかしながら、それらのほとんどは時代遅れであり、多くのタイプのトランザクションはスマートコントラクトで簡単に実装することができます。保持する必要がある唯一のものは InvocationTransaction です。

参考:<https://github.com/neo-project/neo/issues/327>

5.3 NEO scan

ブロックチェーン上の情報を取り出すために、ブロックチェーンのデータを同期させると膨大な時間がかかります。NEO scan を利用すると、Web 上で瞬時にトランザクションの確認を行うことができます。

NEO SCAN API では Web API を提供しており、プログラムに組み込んで NEO のトランザクションを参照させることができます。NEO SCAN API の仕様については下記ページを参照してください

<https://neoscan.io/docs/index.html>

5.4 悪意のあるトランザクションに対する取り組み

NEO ではネットワークのトランザクション数が大幅に増加しており、その中には NEO のネットワークに対する攻撃も含まれます。しかしながら、NEO は高いパフォーマンス

をもち、1秒あたりに多くのトランザクションを処理することが可能です。

NEO では MaxTransactionsPerBlock というパラメーターが、ネットワークからトランザクションが溢れるのを防ぐために追加されました。また、MaxFreeTransactionsPerBlock というパラメーターがあり、これによって1つのブロックで手数料なしで送信できるトランザクションの数を制限しています。MaxFreeTransactionsPerBlock を超えてしまったトランザクションは、ノードのメモリプールに残り、順番を待ちます。MaxTransactionsPerBlock は 500、MaxFreeTransactionsPerBlock は 20 に設定されています。

MaxFreeTransactionsPerBlock が 20 で、各ブロックのトランザクションの先頭は必ず MinerTransaction となるため、ブロックごとのトランザクションの合計は 21 になります。ブロックの中の 21 のトランザクションのうち、いずれかのトランザクションが手数料を払って送信された場合、そのノードは他のノードより高い優先度で処理されます。

トランザクションの手数料の額は、どのぐらい優先度をあげて早く取引を完了したいかによって変わりますが、一般には 1 satoshi gas 程度です。トランザクションの手数料を払うかどうかの決定には、ノードのメモリプールの状態を確認してもよいでしょう。ノードのメモリプールの状態は、NEO-CLI の getrawmempool 関数によって取得することができます。

<http://docs.neo.org/en-us/node/cli/2.7.6/api/getrawmempool.html>

NEO チームでは、ユーザーが取引所から NEO を引き出す際に、取引所が手数料を追加することを推奨しています。また、ウォレットに手数料を設定する機能をつけることを推奨しており、これによってユーザーは簡単に手数料を使用した取引が行うことが可能です。

第 6 章

NEO ネットワーク

NEO では、ブロックを生成を行う上でコンセンサスノードと呼ばれる複数のノードによって合意形成を行います。本章では、現在のコンセンサスノードの概要やこれからコンセンサスノードになるための手順について解説していきます。

6.1 オフチェーンガバナンス

オフチェーンガバナンスは通常 NEO ファウンデーションによって行われます。現在のコンセンサスノードをマネジメントするだけではなく、NEO のプロモーションをしたりコアプロジェクトをサポートしたりしています。

6.2 オンチェーンガバナンス

現在、メインネットではコンセンサスノードは 7 つあり、NEO Foundation が 5 つのコンセンサスノードを持ち、City of Zion と KPN が 1 つずつコンセンサスノードを持っています。そのコンセンサスノードになる前のノードを候補ノード (candidate node) と呼び、現在 4 つの候補ノードがあります。

NEO には 2 つのトーカンがあります。1 つは NEO、もう 1 つは GAS と表される Neo Gas です。NEO の総量は 100,000,000 トーカンであり、分割することはできません。

トーカンはネットワークをマネジメントするためのトーカンとして使われており、NEO のプロトコルのアップデートや block keeping の投票に用いられています。

NEO トーカンは 2 つに分割されており、50% はクラウドセールにて販売され、残りの 50% は NEO が保有しています。これは NEO が長期的なデベロップメントを行いエコシステムをマネジメントするために必要だからです。50% の内訳は以下です。

- 10% : NEO の開発者と NEO ファウンデーションのメンバーに対するインセンティブ
- 10% : NEO のエコシステム内で開発者に対するインセンティブ
- 15% : 不遇の自体に対する備え
- 15% : NEO 上のプロジェクトに対する投資

<https://github.com/neo-ngd/reference/blob/master/How-To-Become-NEO-Consensus-Nodev1.4.md>

6.3 コンセンサスノードへのなり方

オンチェーン、オフチェーンでのコンセンサスノードのなり方があります。

- オンチェーンの場合
 - 要件調査
 - ノードになり投票
 - 投票を集め、意思決定のサポート
- オフチェーンの場合
 - 要件調査
 - NEO ファウンデーションとのパートナーシップ
 - ノードになり投票
 - 投票を集め、意思決定のサポート

6.3.1 要件調査

コンセンサスノードになる場合、次の要件を提出することが求められ、コンセンサスノードページと、オフィシャルページに情報が乗ることになります。

- 組織情報
 - ウェブサイトもしくは SNS アカウント
 - 組織の名前と本拠地
 - 少なくとも 2/3 以上のチームメンバーの写真と詳細
 - コンタクト (email アドレスなど)
- サーバータイプ
- 次の課題に対する対応策
 - ノードの安全性

- メンテナンス
- 長期安定性
- 単一障害性/リカバリー
- 予算
- ハードウェアのスケーリング/アップデートに対するプラン
 - ハードウェアの推奨スペック
 - * 4 Core CPU
 - * 8 GB RAM
 - * 10M Bandwidth
 - * 100G Hard Drive
- NEO のコミュニティに対する貢献

6.4 プレーヤー詳細

6.4.1 NEO Foundation

Public Key: 024c7b7fb6c310fccf1ba33b082519d82964ea93868d676662d4a59ad548df0e7d
NEO ブロックチェーン運営主体。NPO 法人であり Smart Economy (智能経済) を実現するために開発を行っています。NEO ファウンデーションは NEO Global Development と NEO Global Capital により設立されました。City of Zion や NewEconoLab、NeoResearch などに経済的な資金提供を行っています。

twitter: https://twitter.com/neo_blockchain

facebook: <https://www.facebook.com/NEOSmartEcon>

github: <https://github.com/neo-project>

6.4.2 CITY OF ZION

Public Key: 025bdf3f181f53e9696227843950deb72dcd374ded17c057159513c3d0abe20b64

Website: <http://cityofzion.io/>

E-mail: council@cityofzion.io

City of Zion はオープンソース開発、デザイン、翻訳などを行い NEO コアチームとエコシステムを開発しているグループです。

6.4.3 KPN

Public Key: 035e819642a8915a2572f972ddbdbe3042ae6437349295edce9bdc3b8884bbf9a3

Website: <https://www.kpn.com>

E-mail: DLT@KPN.com

KPN はオランダに拠点をもつ通信会社です。

6.4.4 Swisscom

Swiss に拠点をもつ通信会社です。テストネットにノードをもっています。

6.5 ブロック高

NEO ブロックチェーンにおいて、ブロックの生成速度は 15~20 秒に 1 つです。執筆時（2018 年 11 月）で過去に生成されてきたブロックの数は 2922556 となっています。

6.6 NEO テストネット

テストネットとは開発者・ユーザーが自由に開発しプログラムをテストすることができる環境です。テストネットで開発する際にも GAS と呼ばれる手数料がかかりますが、これは実際の GAS ではなく価値はありません。テストネットのブロックは完全にメインネットと隔離されています。なので、実際に開発をする際には、まずテストネットでプログラムを検証してからメインネットにデプロイするという手順を取ります。このステップを踏む理由は、メインネットにデプロイされたプログラムはたとえ開発者でも変更することができないためです。

6.7 テストネットの特徴

1. 無料でやりとりをすることができる
2. グローバルな環境で開発できる
3. ブロックチェーンブラウザにて簡単に過去の取引をみることができる
4. スマートコントラクトをデプロイすることができ、誰でも参照することができる
5. テストネットは商用では使われない

6.8 NEO クライアントをダウンロードする

テストネットとメインネットのクライアントは同じものを使います。クライアントの configuration ファイルを変更することによってメインネットとテストネットを入れ替え

ることができます。

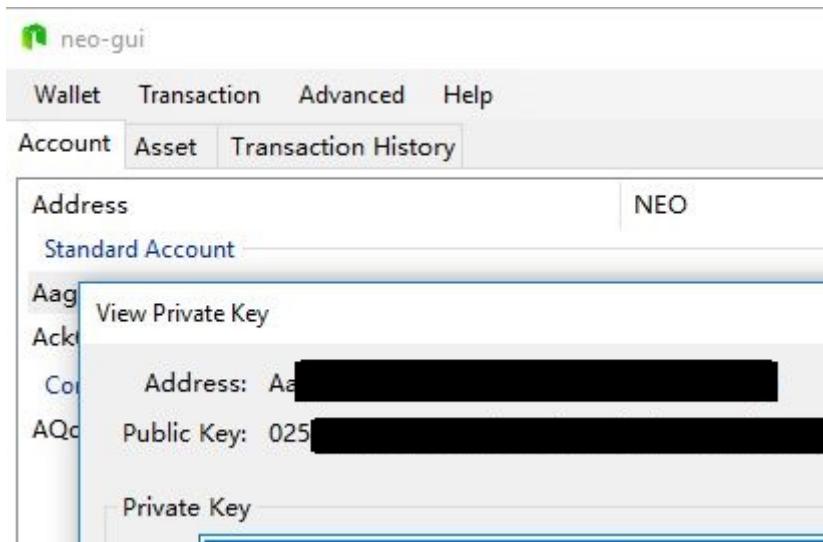
- リリース
 - NEO-GUI
 - NEO-CLI
- ソースコード
 - NEO-GUI
 - NEO-CLI

6.9 テスト GAS とテスト NEO の獲得方法

1. リクエストフォーム（<https://www.neo.org/Testnet/Create>）を埋めますその際に、EMAIL と PUBLIC KEY を入力してください
2. その後、1日ほどした後、コントラクトアドレスと2つ目のプライベートキーを受け取ります
3. 自分のウォレットにマルチシグアドレスを作ります
4. アドレスを特定し自分のウォレットからアセットを送信します

パブリックキーを確認する

アドレスとパブリックキーはことなります。パブリックキーはプライベートキーを表示している時に見ることが出いますが、プライベートキーをシェアするとアドレスの残高から通貨を引き出せるようになるので、他の人にシェアしないようにしましょう。



▲図 6.1 パブリックキーの確認

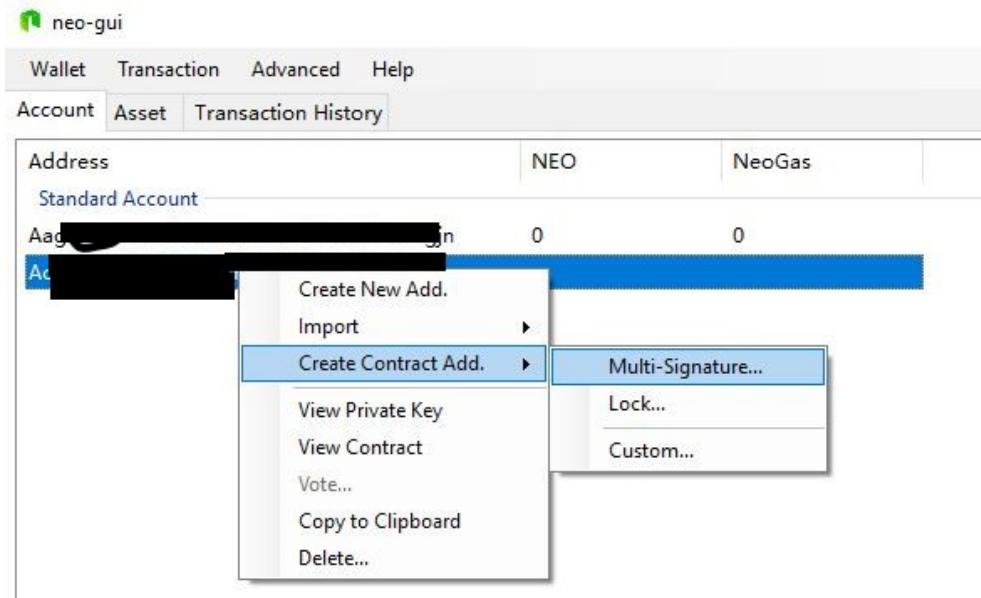
6.9.1 リクエストフォームに回答する

こちらの URL (<https://www.neo.org/Testnet/Create>) から EMAIL アドレスとパブリックキーを指定します。数日ご EMAIL が届きます。

6.9.2 マルチパーティーアドレスを作成する

アセットにアクセスするために、自身の NEO-GUI でウォレット内に "Multi-party signed address" を作成します。

NEO-GUI 上で Create Contract Address を押し Multi-Signature を選択します。



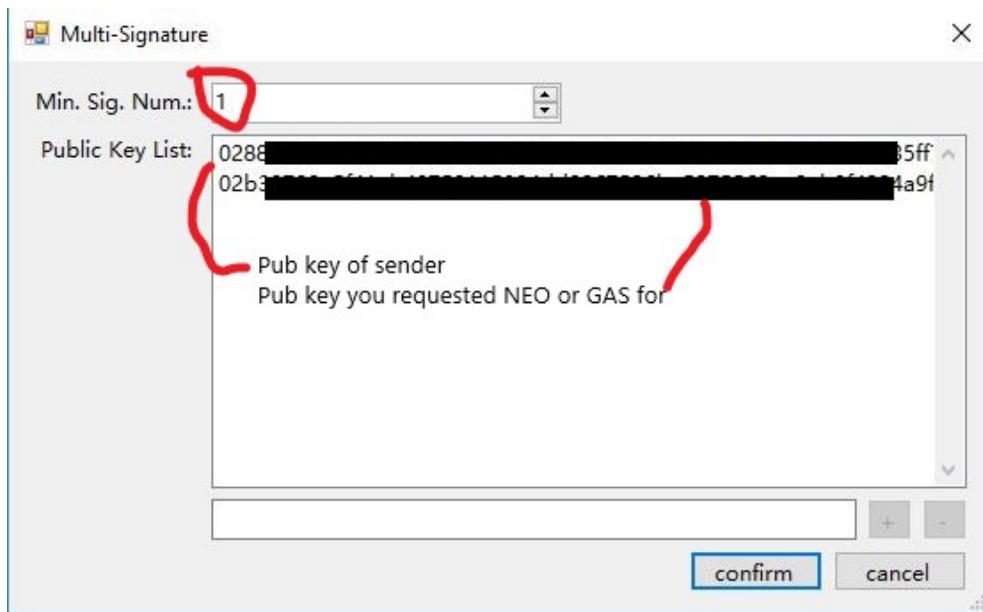
▲図 6.2 NEO-GUI での操作 1

パブリックキーのリストから、署名に必要なキーを選択します。

最低限必要な署名の数を設定します。

confirm を押します。

指定した Email に紐付いた contract アドレスが作成され、アカウントのページに表示されます。



▲図 6.3 NEO-GUI での操作 2

6.9.3 他のアカウントのアセットを送信する

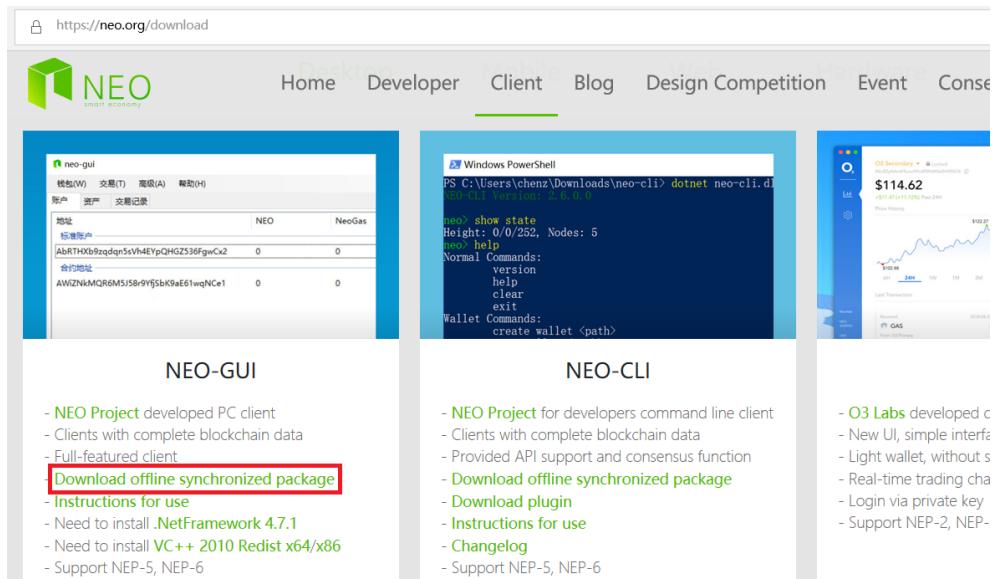
1. NEO-GUI 上で Contract Address を選択します
2. NEO-GUI メニューから Transaction を押し Transfer を選択します
3. アセット、送りたい量、送信先のアドレスを指定します

6.10 クライアントをすばやく同期する

クライアントは使用する実際に使用する前に完全に同期する必要があります。過去のブロックをすべてダウンロードする代わりに必要な部分だけをダウンロードすることで同期をすばやく行うことができます。

6.10.1 ステップ 1

Offline synchronized package をダウンロードする



▲図 6.4 ブロック同期 1

6.10.2 ステップ 2

オフラインパッケージをダウンロードするページで、ネットワークに応じて Mainnet もしくは Testnet を選択し、パッケージをダウンロードします。

The screenshot shows a user interface for a blockchain client. At the top, there are two tabs: '主网 Mainnet' (highlighted with a green border) and '测试网 Testnet'. Below the tabs, there are two sections: 'Full Offline Package 全量离线同步包' and 'Increment Offline Package 增量离线同步包'. Each section contains a table with the following columns: Date / 日期, Starting Height / 开始高度, Ending Height / 结束高度, File Size / 文件大小, Estimate Time / 预计时间, and Download / 下载 (with a 'download' link). A red box highlights the 'download' link for the first package.

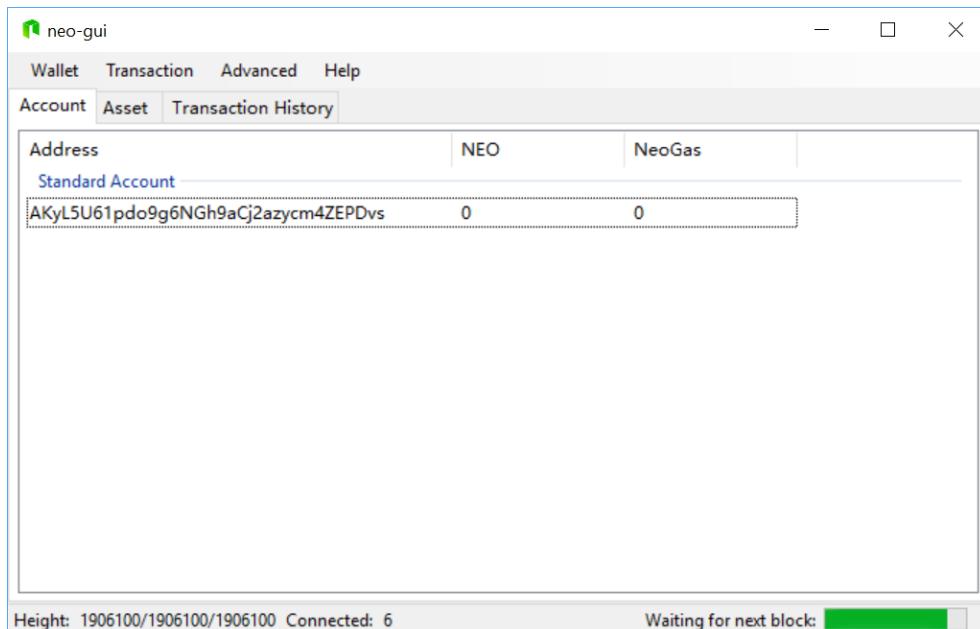
Date 日期	Starting Height 开始高度	Ending Height 结束高度	File Size 文件大小	Estimate Time 预计时间	Download 下载
2018/10/15	1895770	1895770	955.25M	1.5h	download MD5:5F86D8FF094B6F88CBC80152AD0B1F47

Date 日期	Starting Height 开始高度	Ending Height 结束高度	File Size 文件大小	Estimate Time 预计时间	Download 下载
2018/10/17	1855770	1905770	25.25M	15min	download MD5:A924DC996D80FD7C378D7E01C6B824F

▲図 6.5 ブロック同期 2

6.10.3 ステップ 3

クライアントを起動してブロックのダウンロードしてステータスを確認します。



▲図 6.6 ブロック同期 3

* NEO-GUI では、画面からダウンロードの状態を確認することができます。

6.11 ネットワーク・プロトコル

NEO は P2P (Peer to Peer) ネットワーク構造を採用しています。この構造では、ノードが TCP/IP プロトコルを通じてお互いにやりとりをします。このプロトコルでは Peer ノードと Validating ノードという 2 種類のノードがあります。Peer ノードは取引やブロックを拡散し受け取り、交換します。一方で Validating ノードはブロックを作り出すことができます。

6.12 NEO のノードでプライベートチェーンを構築する

プライベートチェーンの構築方法と GAS と NEO の使用方法を見ていきましょう。

6.12.1 Virtual Machine をセッティングする

NEO のプライベートブロックチェーンでは、コンセンサスを行うために少なくとも 4 つのノードが必要です。今回はデモンストレーションなので、Standard DS1 v2 1 core, 3.5 GB RAM サイズの Windows Virtual Machine を Azure 上に構築します。

NAME	TYPE	STATUS	RESOURCE GROUP	LOCATION
PrivateChain1	Virtual machine	Running	PrivateChain	West Europe
PrivateChain2	Virtual machine	Running	PrivateChain	West Europe
PrivateChain3	Virtual machine	Running	PrivateChain	West Europe
PrivateChain4	Virtual machine	Running	PrivateChain	West Europe

▲図 6.7 プライベートチェーンの構築

クラウドのサーバーに virtual machine を構築する場合、virtual machine の管理画面にログインして network security group を作成してください。

"network interface" "network security group" "inbound security rules" "add" を port 10331-10334 に加えてください。

virtual machine の作成が完了したら、4 つの virtual machine の IP address を保存してください。このアドレスは後使用します。

6.12.2 ウォレットを作成する

最初に 4 つのウォレットを作成します。それらには、wallet1.db3, wallet2.db3, wallet3.db3, wallet4.db3 という名前をつけます。このステップはローカルの PC 画面上もしくはコマンドラインで実行することができます。次のスクリーンショットでどのようにウォレットをコマンドライン上で作成するかが分かると思います。

```
C:\neo-cli>dotnet neo-cli.dll
neo>create wallet privatechain1.db3
password:*****
password:*****
address: AeovvvvcViGvWNSpnN7V9WdrdAzj8RXL9g
pubkey: 0371568c58c85e7c1237f0b8ca029139ac0ce97bbb11eeb67bb9e3b6fbfc0182df
neo>create wallet privatechain2.db3
password:*****
password:*****
password:*****
password:*****
address: AW913NtwRA6P5GkGbMAVsFkfE4vpLw3pw
pubkey: 03ebdefbc19cff99e8a87e7303ec0ff67ea6b1fa1766c6770e4128526c9803ecd4
neo>create wallet privatechain3.db3
password:*****
password:*****
password:*****
password:*****
address: AHeYhx9Yvf5fxYVD9xx4WeLLhVgtcQCruw
pubkey: 039b1c1f9120ed823cf5f7207eb0ed50c2ef6cc4b3a6ea0e0073f6ed7410ee5151
neo>create wallet privatechain4.db3
password:*****
password:*****
address: AdtXYEyK6L1rJmC4hQmkf1BnJsvWzE5UX
pubkey: 03156d1c0e315d63d400c695261d169c881f550e7a17757963489ecc30d4fa5604
neo>
```

▲図 6.8 wallet の生成

ウォレットを作成することができ、対応するパブリックキーをテキストファイルに保存します

その後、ウォレットを 4 つの Virtual Machine のノードにコピーします。

6.12.3 configuration ファイルを修正する

ノードの設定ファイルである protocol.json を開きます。Magic の value を修正します。Magic は通常、メッセージのネットワークを識別するために用いられています。Magic は uint 型なので、値は [0 - 4294967295] のいずれかをとります。

StandbyValidators を修正し、さきほど保存した 4 つのパブリックキーの値を入力します。最後に SeedList を修正します。ここでは、以前保存した IP address 入力し port は変更を加えないようにします。次のコードを参考にしてください。

▼ configuration

```
{
  "ProtocolConfiguration": {
    "Magic": 1704630,
    "AddressVersion": 23,
    "StandbyValidators": [
      "02f27545181beb8f528d13bbb66d279db996ecb56ed9a324496d114acb48aa7a32",
      "02daa386d979ae6643869a365294055546023acb332ee1a74a5ae5d54774a97bac",
      "0306f12f7217569cdbe9dde9ff702d0040e0a4570873eee63291adaa658128e55c",
```

```
        "035781b4d55dc58187f61b5d9277afbaae425deacc5df57f9891f3a5c73ecb24df"
    ],
    "SeedList": [
        "13.75.112.62:10333",
        "137.116.173.200:10333",
        "168.63.206.73:10333",
        "137.116.171.134:10333"
    ],
    "SystemFee": {
        "EnrollmentTransaction": 0,
        "IssueTransaction": 0,
        "PublishTransaction": 0,
        "RegisterTransaction": 0
    }
}
```

SystemFee は手数料を表しています。次の作業にかかる手数料は* book-keepers の登録 - 1000 * アセットの分配 - 500 * スマートコントラクト 500 * アセットの登録 - 10000 となっています。

この設定は SeedList 内で変更することができます。

その後、4 つのバーチャルマシーンで次のコマンドを入力してノードを動かします。ウォレットを開き、コンセンサスプロセスを開始します。CLI Command Reference を参考してください。

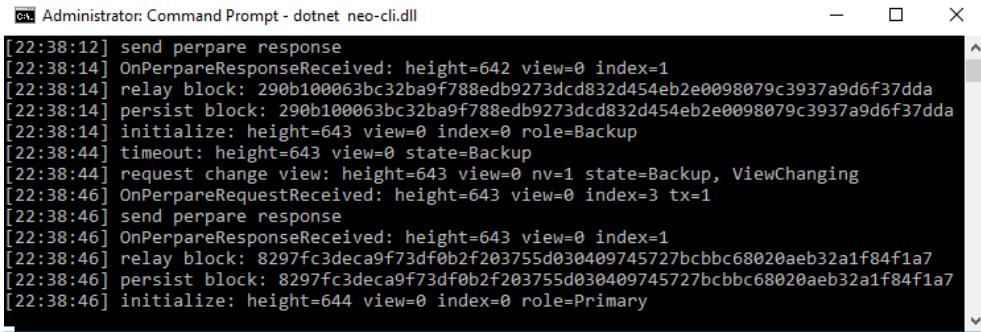
ノードをスタートする : Dotnet neo-cli.dll

ウォレットを開く : Open wallet wallet1.db3

注意 : すべてのノードで wallet1 を開くわけではないので、ノードによって番号を変えてください。

コンセンサスをスタートする : Start consensus

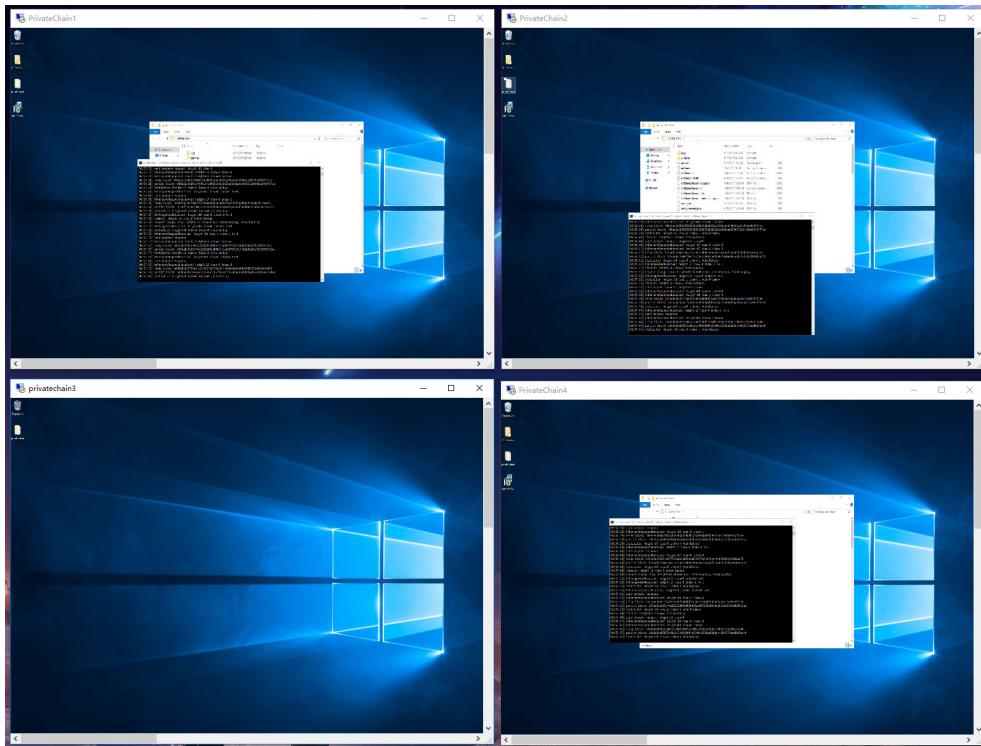
上記のコマンドが正常に動作すれば、4 つのノードは次のようなコンセンサスプロセスを表示します。



```
[22:38:12] send perpare response
[22:38:14] OnPerpareResponseReceived: height=642 view=0 index=1
[22:38:14] relay block: 290b100063bc32ba9f788edb9273dc832d454eb2e0098079c3937a9d6f37dda
[22:38:14] persist block: 290b100063bc32ba9f788edb9273dc832d454eb2e0098079c3937a9d6f37dda
[22:38:14] initialize: height=643 view=0 index=0 role=Backup
[22:38:44] timeout: height=643 view=0 state=Backup, ViewChanging
[22:38:46] OnPerpareRequestReceived: height=643 view=0 index=3 tx=1
[22:38:46] send perpare response
[22:38:46] OnPerpareResponseReceived: height=643 view=0 index=1
[22:38:46] relay block: 8297fc3deca9f73df0b2f203755d030409745727bccbc68020aeb32a1f84f1a7
[22:38:46] persist block: 8297fc3deca9f73df0b2f203755d030409745727bccbc68020aeb32a1f84f1a7
[22:38:46] initialize: height=644 view=0 index=0 role=Primary
```

▲図 6.9 コンセンサスプロセス

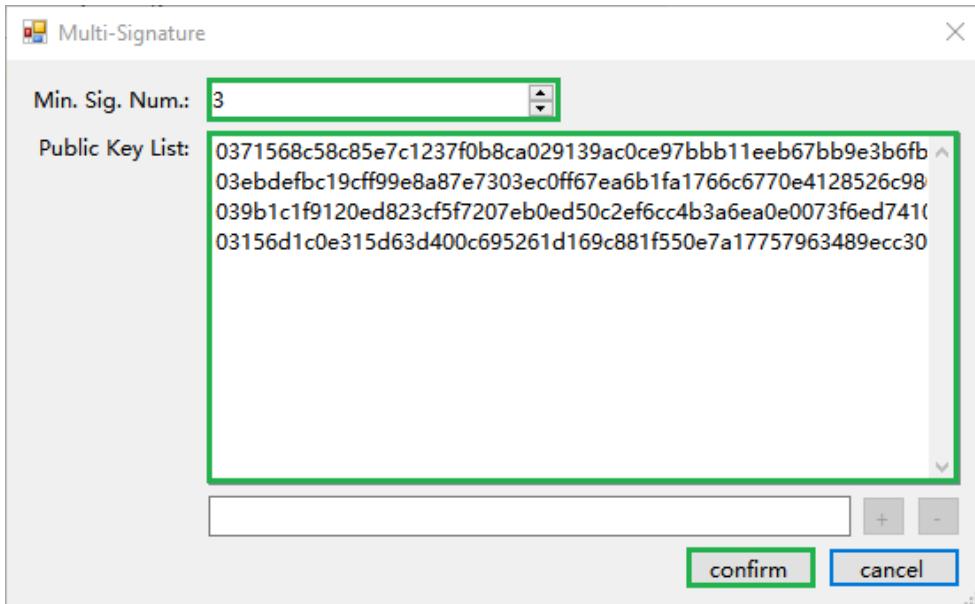
4つのノードは次に見れるように1つの環境が停止しても動き続ける。



▲図 6.10 コンセンサスの継続

6.13 無料の NEO を手に入れる

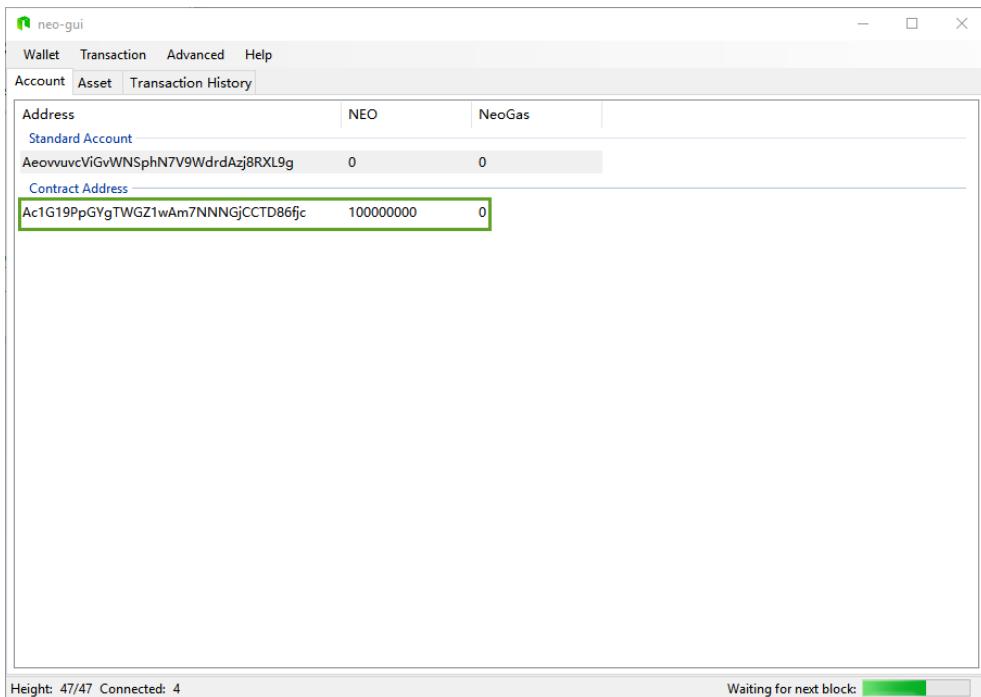
Neo-GUI をインストールし、プライベートブロックチェーンにつなぐために、protocol.json を修正します。Wallet を開き、左上の文字が 0 でなく、つぎのように表示されていました場合、クライアントはプライベートチェーンに接続されています。



▲図 6.11 NEO-GUI でプライベートチェーンに接続

PC クライアントの wallet1.db3 を開き、multi-party signature アドレスを追加します。その後、protocol.json に 4 つのパブリックキーを入力します。マルチシグのミニマムの鍵数を選択（この場合は 3 つ）します。

Confirm をクリックしメニューバーの wallet をクリックします。そうすると、コントラクトアドレスに 100000000NEO が入っていることを確認することができます。



▲図 6.12 NEO 獲得

第 7 章

コンセンサスアルゴリズム

こちらの章は

<http://docs.neo.org/en-us/basic/consensus/whitepaper.html>

<http://docs.neo.org/en-us/basic/consensus/consensus.html>

の一部を日本語訳し、補足を加えたものです。

7.1 用語一覧

- Proof of Stake PoS - 障害許容のあるネットワークコンセンサスを使ったアルゴリズムの一種。
- Proof of Work PoW - 障害許容のある計算能力を使ったアルゴリズムの一種。
- Byzantine Fault BF - ノードは機能しつつも悪意を持って行動する失敗
- Delegated Byzantine Fault Tolerance DBFT - 障害許容を保証した NEO ブロックチェーンに実装されたコンセンサスアルゴリズム
- View v - NEO DBFT のコンセンサス形成において使われるデータセット

7.2 概要

ブロックチェーンは分散型台帳システムで、デジタル資産の登録や発行、財産権証明書、クレジットポイントなどに利用することができ、P2P で決済やトランザクションを送ることを可能にします。ブロックチェーン技術は Cryptography Mailing List で Satoshi Nakamoto によって Bitcoin としてはじめて提唱されました。Bitcoin の提唱以降 e-cash などたくさんのアプリケーションがブロックチェーン上で作られています。従来の中央集

権的な社会においてブロックチェーンはオープンで改ざんが不可能であり、二重送金が防止でき、第三者に依存せずトラストレスに機能するという点で極めて革新的です。しかし、すべての分散型台帳がそうであるようにブロックチェーンもネットワークはの遅延、送信エラー、ソフトウェアのバグ、セキュリティループホール、悪質なハッカーによる攻撃など多くの問題に対処する必要があります。さらに、トラストレスな分散型であるがゆえに参加者の全員を信用することができません。つまり、悪意にあるノードが現れる危険性もあれば、利権を争ってデータの分裂も生まれる可能性があります。このような潜在的な欠陥に対処するためにブロックチェーン技術は、常にすべてのノードが承認した最新バージョンの台帳を全員が共有していることを保証する必要があります、さらに台帳を更新していくための効率的なコンセンサスメカニズムを必要とします。従来の障害許容メカニズムではこのような問題を完全に対処することができません。ユニバーサルな規模ですべての欠陥に対処できる策が必要です。ビットコインで採用されている Proof-of-Work (PoW) メカニズムは上記であげた問題を見事に解決します。PoW 使った方式はマジョリティの計算能力が善良のノードによるものとすることで障害許容が保証されます。しかし、計算能力に依存するこのスキームではマイニングをする際に電力の大量消費やハードウエアの使用により非常に効率が悪いです。このような依存は PoW のネットワークにさまざまな制限をもたらし、特にスケーラビリティ問題は深刻な問題になっています。さらに、ビットコインマイナーの圧倒的ハッシュパワーの影響を受けないようにするために、新しく作られるブロックチェーンはビットコインと異なるハッシュアルゴリズムを見つける必要があります。たとえば Litecoin はビットコインで使われている SHA256 ではなく SCRYPT を採用しています。

7.3 ビサンチン将軍問題 (BFT)

ビサンチン将軍問題とは分散型台帳において昔から存在する問題のひとつで、相互に通信しあう集団において、個々のプレイヤーに対して悪意のある人による攻撃や通信の故障によって偽りの情報が伝達される可能性がある場合に、全体として正しい合意形成を行えるかという問題である。ビサンチン障害許容メカニズム (BFT) は分散型台帳において生じるこのような欠陥に対する新しい解決策です。特に NEO で採用されているコンセンサスアルゴリズム dBFT はビサンチン障害許容メカニズムの中でも、パブリックブロックチェーン用に工夫が施されています。

7.4 dBFT (Delegated Byzantine Fault Tolerant)

ブロックチェーン間のもっとも根本的な性質の違いのひとつはどのようにして障害耐性を保証するかです。NEO は dBFT というコンセンサスアルゴリズムを採用しています。dBFT は Delegated Byzantine Fault Tolerant の頭文字をとったものであり、大規模な議決権の代理行使を通じてコンセンサス形成をするビザンチン障害許容メカニズムです。dBFT は通常の BFT とは異なり、NEO トークン保持者による投票を通じて支持する代表者を選ぶことができます。delegated と呼ばれる所以はこの議決権の代理行使の仕組みにあります。BFT アルゴリズムを通じて選ばれた代表者のグループはコンセンサス形成を行い、新しいブロックを生成します。dBFT コンセンサスアルゴリズムはこのようにして PoS の特徴（NEO 保持者はコンセンサスノードの採決を行う）を一部取り入れる一方、最小限のリソースを用いてネットワークをビザンチン将軍問題のような欠陥からシステムを守ります。NEO のネットワークにおける投票はリアルタイムで継続しています。このコンセンサスアルゴリズムは障害耐性に欠陥をもたらすことなく現在のブロックチェーンにおける性能やスケーラビリティ問題を対処することができます。

dBFT は n 個のコンセンサスノードからなるシステムにおいて $f = (n-1) / 3$ までの障害許容を提供します。これは $f = (n-1) / 3$ まで個の悪意のあるノードまでシステムは耐性があるということを意味します。このメカニズムはビザンチン将軍問題に耐性があり、可用性を含んでいるためどんなネットワーク環境にも適しています。dBFT は一度確認が済むとブロックは分岐できなくなり、トランザクションは取り消されたり戻されたりしないのでファイナリティが高いと言えます。NEO の dBFT コンセンサスマカニズムでは、ブロックを生成するのに 15~20 秒程度なのでトランザクションの処理数は毎秒 1,000 トランザクションにまで登ります。この数字はパブリックブロックチェーンにしては極めて高いです。たとえばビットコインのブロックは 10 分に一度生成され、1 ブロックは 1MB が上限なため理論上毎秒処理できるトランザクションは 6~7 個です（実際は若干変動する）。さらに dBFT は最適化を通じて処理できるトランザクション数を毎秒 10,000 トランザクションにまで増加させるポテンシャルがあります。このように大量のトランザクションを処理できるシステムは大規模な商業にも応用できます。また、dBFT はデジタルアイデンティティ技術を取り入れており、記録者は実名や会社名を利用することができます。ゆえに、司法的決断によりフリーズ、取り消し、相続、回復、権限の送付が可能となっています。これは NEO ネットワークに金融資産を登録することを促進させます。

7.5 システムモデル

ブロックチェーンとは分散型台帳システムであり、参加者は P 2 P のネットワークを通じて繋がっており、メッセージは台帳に書き込まれていくように共有されます。一般にノードは二種類存在しひとつは普通のノード、もうひとつは記録ノードです。普通のノードは台帳のデータを受領しながら、情報の伝達や価値の移動の手段としてブロックチェーンを利用しています。一方記録ノードは経理のような業務を行うことで台帳を管理維持しています。ブロックチェーンをはじめとした分散型台帳システムではメッセージの消失や破損、遅延の繰り返しが考えられ、さらに送信した順番は必ずしも受信する順番と一致しないことが起こります。また、ノードの行動は恣意的で自由にネットワークを出入りでき、時に誤った情報を流すこともあるれば、急に機能しなくなることも想定されます。さらに、機械であろうが人間であろうがエラーは必ずつきものになってきます。このような問題に対処すべくブロックチェーンでは送信者がメッセージのハッシュ値に署名をすることによって情報の整合性や確実性は暗号学で保証します。今 $\langle m \rangle \sigma_i$ を、ノード i からのメッセージ m の電子署名だと定義し、また $D(m)$ をメッセージ m のハッシュ値だと定義します。特別な断りがない限りすべての署名はメッセージのハッシュ値への署名であるとします。

7.6 一般手順

通常時のブロック生成時間を t とするとアルゴリズムは次の流れで実行されます。

1. ノードがトランザクションデータに署名をつけてネットワーク全体に伝達する
2. すべての記録ノードはトランザクションを各自で監視し、各自メモリーに保存する
3. 時間 t 経過後議長（スピーカー）は $\langle \text{prepareRequest}, h, v, p, \text{block}, \langle \text{block} \rangle \sigma_p \rangle$ を送る□
4. 提案を受けると議員 i が $\langle \text{prepareResponse}, h, v, i, \langle \text{block} \rangle \sigma_i \rangle$ を送る
5. 受信したノード数が $n - f$ $\langle \text{block} \rangle \sigma_i$ に到達するとコンセンサスが形成され完全なブロックが作られる
6. 完全なブロックを受け取るとメモリー内のトランザクションを消し次のラウンドのコンセンサス形成を開始

全てのコンセンサスノードにとって少なくとも $n - f$ 個のノードが同じオリジナルの状態にいる必要があります。具体的には全てのノード i にとってブロック高 h と View の数字 v は一致していないといけないということです。しかしこれを実現するのは非常に難しいです。 h の値の共有はブロックを同期することで保つことができます。一方 v の値の共有は View を変更することで保つことができます。ブロックの同期はここでは扱いま

せんが、View の同期については次のセクションで扱うことにします。各ノードは伝達を監視し、提案の受領を見届けた後、トランザクションを検証します。一度提案が晒されてしまうとノードはメモリーに不正なトランザクションを新しく書くことができなくなります。もし不正なトランザクションが提案に初めから含まれていた場合、このラウンドのコンセンサス形成は断念され、View の値が直ちに変更されます。検証の過程は次のとおりである。

1. トランザクションデータのフォーマットがルールを守っているか?もしそうでなかったらトランザクションは不正である。
2. トランザクションはもうブロックチェーンにあるか?もしあればトランザクションは不正。
3. トランザクションのコントラクトスクリプトがちゃんと実行されているか?もし実行できていなかったらトランザクションは不正。
4. トランザクション内に二重送金があるか?もしあればトランザクションは不正。
5. 上記の過程でトランザクションが不正とみなされなかった場合、正しいトランザクションとみなされる。

7.7 View の変更

もし $2^{(v+1)} * t$ 回のインターバルの後コンセンサス形成ができない場合、あるいは不正なトランザクションを含む提案を受領した時 view の値が変更されます。 $k=1, vk = v + k$ とする。ノード i が View 変更リクエストを送る $\langle \text{ChangeView}, h, v, i, vk \rangle$ 。どのノードからでもよいので少なくとも $n - f$ 個の同じ vk を異なる i から受信すると、View の変更が完了する。 $v = vk$ としコンセンサス形成が再び始まる。もし $2^{(v+1)} * t$ 回のインターバルの後 View の変更がなされない時 k が増加してステップ 2 に戻る。

k が増加すると全体の待ち時間が指数関数的に長くなる。したがって頻繁な View の変更は避けられノードはコンセンサスを形成することが強いられます。View の変更が完了する前にオリジナルの View v はまだ有効であるためネットワークの遅延などによる不要な View の変更が避けられます。

7.8 フローチャート

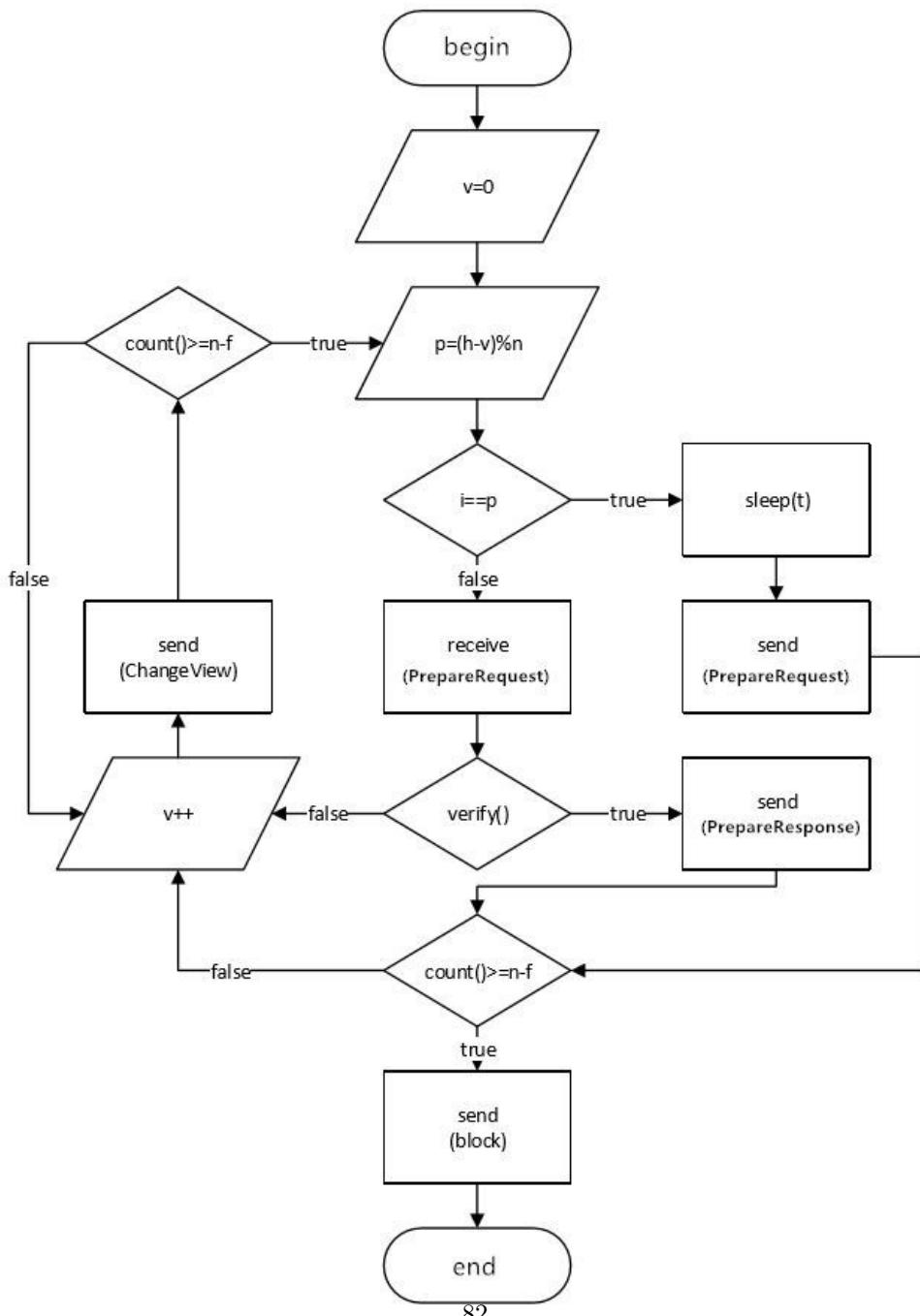


図: フローチャート

7.8.1 役割

NEO のコンセンサスアルゴリズムではコンセンサスノードは NEO 保有者の中から選ばれ、選ばれたものはトランザクションの正当性を判断し、投票を行います。このようなノードは上記で述べた記録ノードと同じノードを指します。以下では彼らをコンセンサスノードと呼びことにします。



図: コンセンサスノード

- コンセンサスノードこのノードはコンセンサス形成に大きく関与する。コンセンサス形成時コンセンサスノードは次のふたつの役職に分かれる。



図: スピーカー (One)

- スピーカー (One) スピーカーはシステムにブロックの提案を行う。



図: 代表者 (Multiple)

- 代表者 (Multiple) 代表者はトランザクションのコンセンサス形成の責任をもつ

7.8.2 理論

まずスピーカーの順序にしたがって必ずコンセンサス形成を行ういくつかの代表を決めます。このシステムではスピーカーやどんな代表者も悪者になり得るので気をつけなければなりません。悪意のあるノードがいる場合の攻撃方法を色々検討して万全な障害耐性を整える必要があります。たとえば、悪意のあるノードは受信者に対してバラバラのメッ

セージを送るかもしれません。これは考えられる問題の中でももっとも深刻な問題です。この問題ん対処法はスピーカーが善良かどうかを代表者が見極め、グループとして正しく昨日するために必要な行動が何かを認識することです。dBFT が正しく機能するかを確かめるために 66.66% consensus rate の検証を行う。ここで悪意のあるノードはアクティブに悪事を働くとは限らないことを頭に留めておかなければなりません。つまり突然的に善良なノードが悪意をもつようになるかもしれないということです。また善良なノードが過失的に誤って機能してしまう恐れもあります。以下ではコンセンサスアルゴリズムがうまく機能するかどうかを確かめるためにいくつかのシナリオを想定し、検証することにします。具体的には n が小さい値のときの簡単な例における各ノードとスピーカー間のメッセージの送受信を想定します。このメカニズムは dBFT でも使われており、システムの中枢を担っています。今回はうまく機能する場合と機能しない場合のみを比較することにします。より詳細な説明は参照をご覧ください。

7.8.3 Honest Speaker

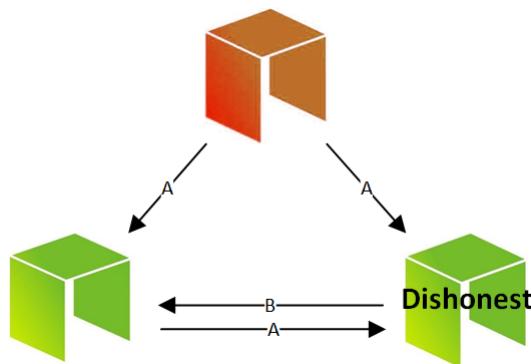


図: $n = 3$ で悪意のある代表者がいる時の例

図 1 ではひとつの善良な代表者 (50%) がいる。両代表者は善良なスピーカーから同じメッセージを受け取る。しかし悪意のある代表者によって善良な代表者は悪意のある代表者がいることを確認することしかできず、それがスピーカーなのかもう一方の代表者かはわからない。これにより代表者は View を変更し、投票を拒みます。

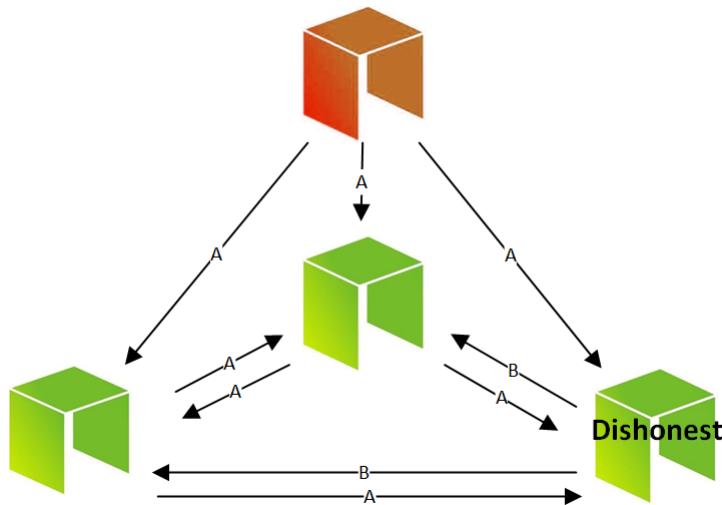
図: $n = 4$ で悪意のある代表者がいる時の例

図2において2つの善良な代表者がいる(66%)。全ての代表者は善良なスピーカーから同じメッセージを受信し、検証結果を他の代表者に送信する。ふたつの善良な代表者によるコンセンサス形成により私たちは、スピーカーか右の代表者が悪意があると分かる。

7.8.4 Dishonest Speaker

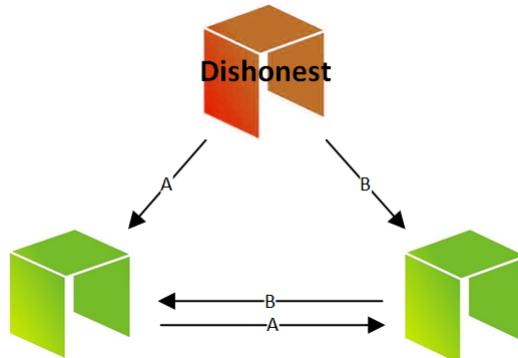
図: $n = 3$ で悪意のあるスピーカーがいる時の例

図3の場合の結論は図1に想定されるな結論がある。代表者はどのノードが悪意があるかはわからない。

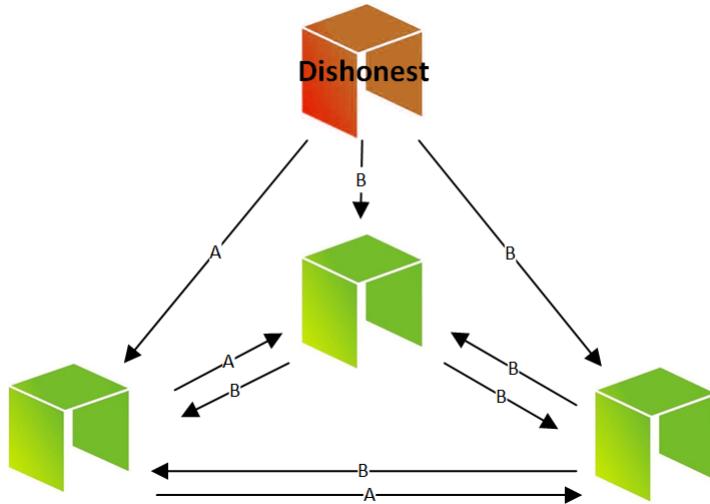


図: $n = 4$ ので悪意のあるスピーカーがいる時の例

図4で示された例では真ん中と右のノードから受信したブロックは検証不可能である。これによって善良な代表者全体の 66% を占めるため、view が変更され、新しいスピーカーを探求する。この例において悪意のあるスピーカーが代表者の 3 分の 2 に正しいデータを送った場合、view を変更することなく検証することができます。

7.9 実装

NEO の dBFT の実装は一定の善良な代表者のメッセージが一致するまで反復的にコンセンサス形成を行うメソッドによりコンセンサスが正しく形成されることを保証します。アルゴリズムのパフォーマンスはシステム内の善良なノード割合に依存します。図5は悪意のあるノードの割合の働きに応じて予想される反復を示します。図5は善良のノードの割合は 66.66% を超えない場合を想定します。66.66% という決定的な値から 33.33 %までの間にはコンセンサス形成ができない'No-Man's Land'があります。33.33% 次の場合は悪意のあるたちはノードはコンセンサス形成が可能になり、システム内で悪意のあるノードによる合意形成が”正しい”とみなされた情報として書き込まれることになります'。

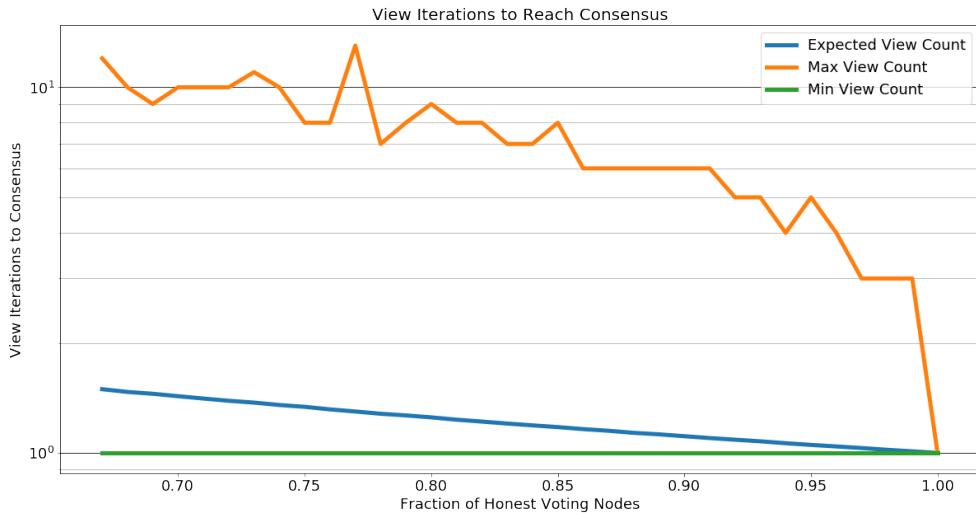


図: Monto-Carlo による DBFT アルゴリズムのシミュレーション

各コンセンサス形成に必要な反復数を示す{100 Nodes; 100,000 Simulated Blocks with random honest node selection}

7.10 アルゴリズム

7.10.1 定義

アルゴリズムにおいて以下を定義する:

- t: ブロック生成に配分された時間 (秒)
- 現在: $t = 15$ seconds
- この値はコンセンサス形成活動とコミュニケーションイベントが時間に顕著に関係しているれば、ひとつの view の反復の期間を近似するのに利用できる。
- n: アクティブなコンセンサスノードの数
- f: システム内における最小限の悪意のあるノード
- $f = (n - 1) / 3$
- h :現在のブロック高
- i : コンセンサスノードインデックス

- v : コンセンサスノードの view。view はノードがコンセンサス形成のラウンド内で受信した情報の集合を含む。これはすべての代表者による投票を含む。 (prepareResponse or ChangeView)。
- k : view v のインデックス。コンセンサス形成アクティビティは複数のラウンドを要求できる。コンセンサス形成が失敗した時、 k は 1 増加し、次のラウンドが始まる。
- p : スピーカによって選ばれたコンセンサスノードのインデックス。この計算メカニズムは一つのコンセンサスノードが独裁的に働くことを防ぐためである。
- $p = (h - k) \bmod (n)$
- s : 安全なコンセンサスの threshold。これを下回るとネットワークに欠陥が生じる。
- $s = ((n - 1) - f)$

7.10.2 必要事項

NEO では consensus fault tolerance のために 3 つの主要な必要事項がある。

1. s 個の代表者ノードはブロックがコミットされるようになる前にトランザクションのコンセンサス形成を行う必要がある。
2. 悪意のあるコンセンサスノードは善良なコンセンサスノードに偽ったトランザクションをするように唆すことはできない。
3. コンセンサス形成アクティビティを始めるにあたって少なくとも s 個の代表者は同じ状態 (h, k) でなければならない。

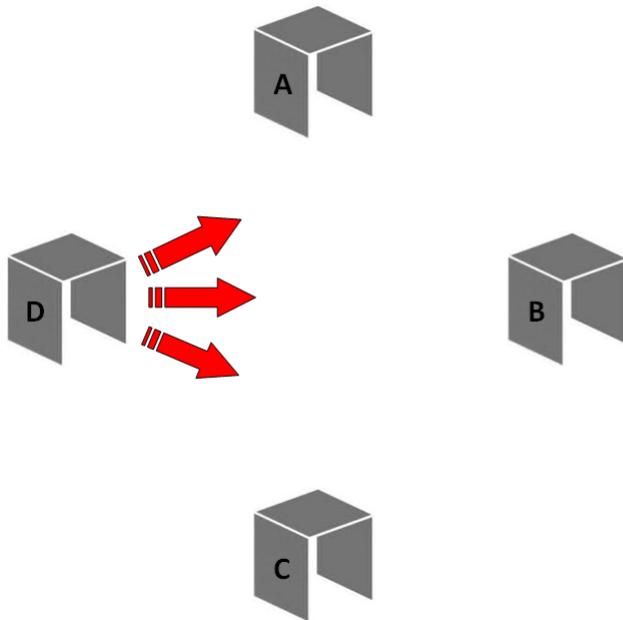
7.10.3 アルゴリズム概要

BFT はセキュリティとユーザビリティ両方をカバーします。BFT では、合意形成にジョインしたノードの合計を $n = |R|$ (R はコンセンサスノードのセットを表す) すると、悪意のあるノードが $(n - 1) / 3$ 以下の場合システムの安定性は確保されます。 f をシステムが耐えられる悪意のあるノードの最大数とすると $f = (n - 1) / 3$ における。実際、台帳全体は記録ノードによって維持され、普通のノードはコンセンサス形成には関与しません。また、すべてのコンセンサスノードは最新の状態遷移を記録する必要があります。コンセンサスに使われる初めから現在までのデータセットを View と呼びます。もある View でコンセンサスを形成することができなければ View の変更が必要になります。私たちはそれぞれの View を 0 から番号をつけて識別し、この番号はコンセンサスが

取れるまで一つずつ増加します。 n 個のコンセンサスノードは 0 から $n-1$ までの番号がつけられます。毎回コンセンサスを形成する際、あるノードが議長（スピーカー）をやり、そのほかのノードは議員の役割をします。議長（スピーカー）の番号 p は以下のようなアルゴリズムによって決定します。仮に現在のブロック高を h とし、 $p = (h - v) \bmod n$ とすると p の値は $0 \leq p < n$ となります。ブロックは少なくとも記録するノードの $n-f$ 個の署名コンセンサス形成時に毎回生成されます。ブロックが生成されると $v=0$ にリセットされ、新しいコンセンサス形成のラウンドが始まります。

アルゴリズムは次のようにして機能します。

1. コンセンサスノードはトランザクションをネットワーク全体に送信者の署名付きで伝達する。



図：コンセンサスノードはトランザクションを受信しシステム全体に伝達する。

2. コンセンサスノードはトランザクションデータをローカルメモリーに記録する
3. コンセンサス形成の初めの view v は初期化されている
4. スピーカが確認される。 t 秒待つ

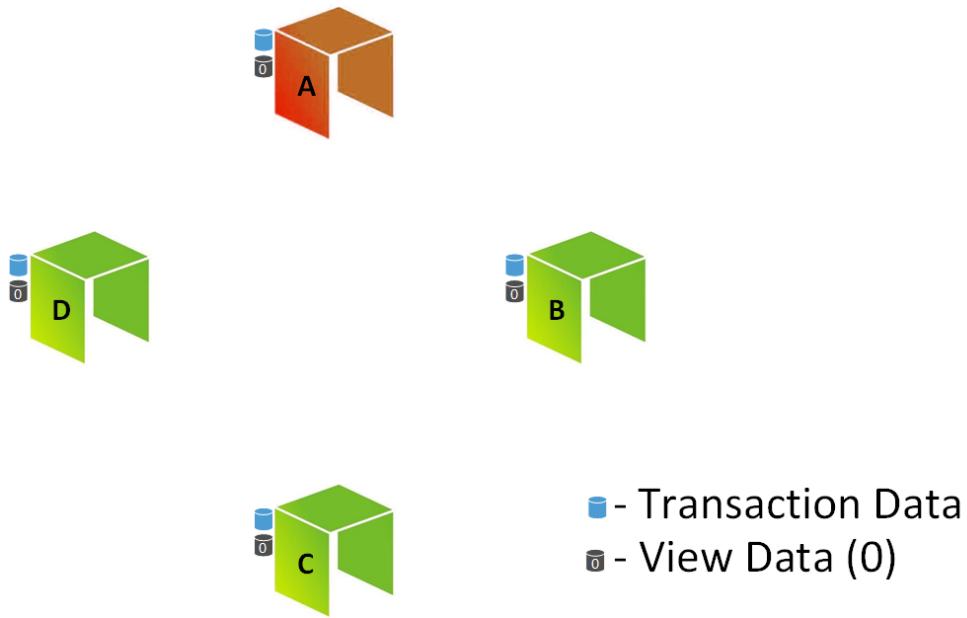


図: スピーカーが確認され、view がセットされる。

5. スピーカーが提案を伝達する: <prepareRequest, h, k, p, bloc, [block]sign>

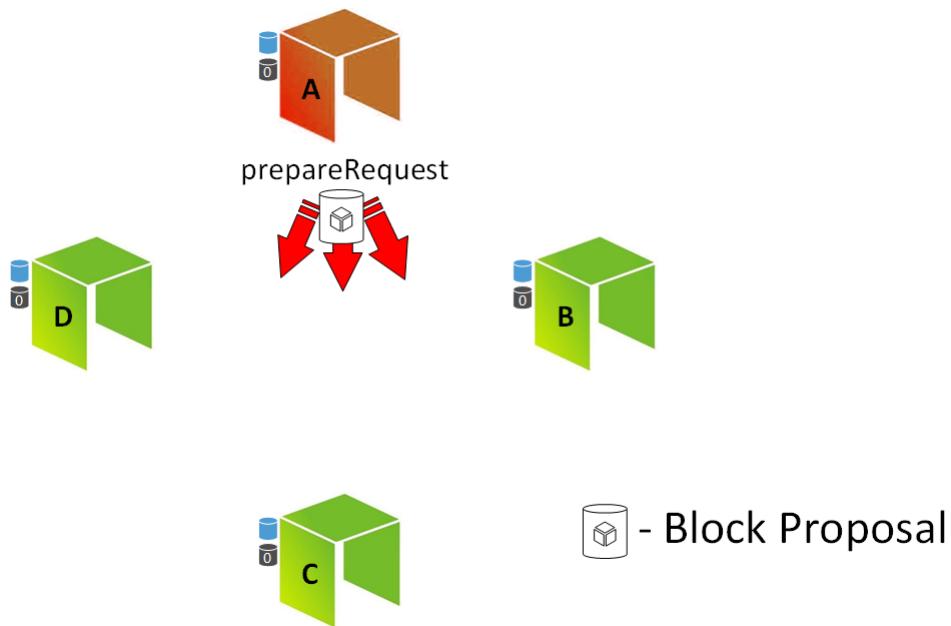
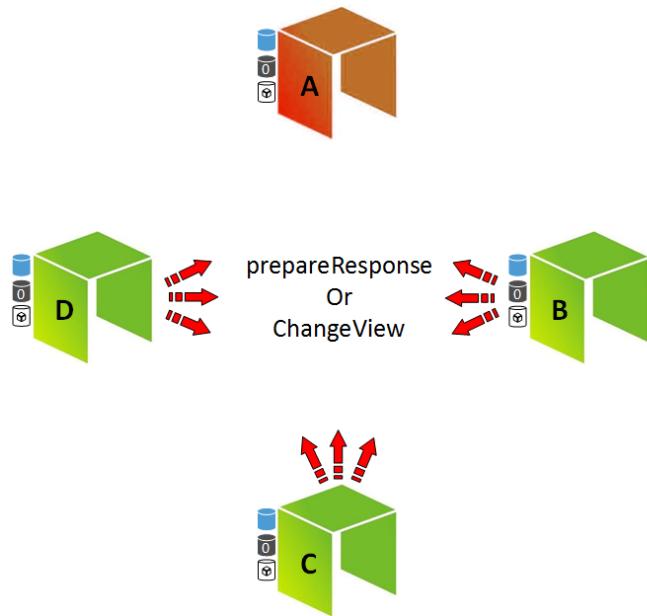


図: スピーカーが代表者によるレビューのためにブロックの提案を作る。

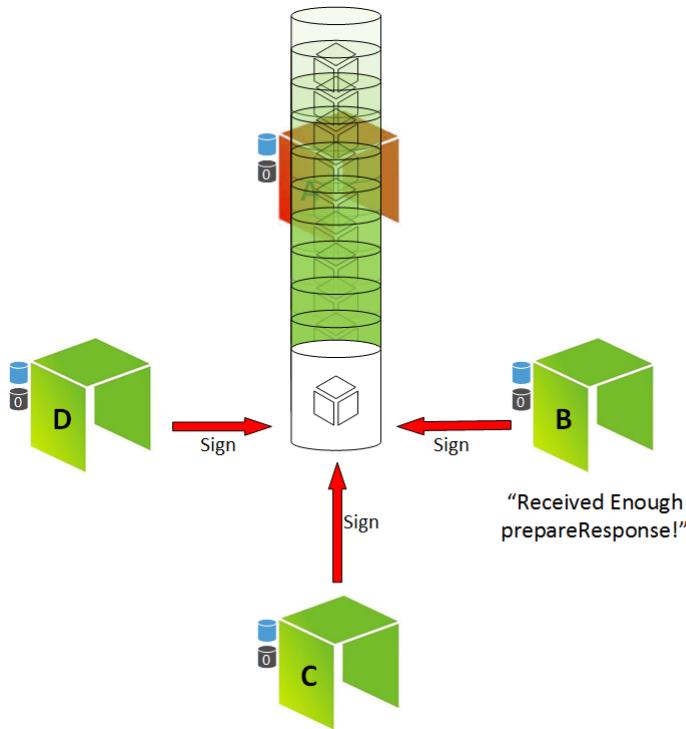
6. 代表者は提案を受信し検証する

- データのフォーマットがシステムのルールと合致しているか。
- トランザクションはすでにブロックチェーンに格納されているか
- コントラクトのスクリプトは正しく実行されているか
- トランザクションは二重送金になっていないか
- もし正しく検証されれば <prepareResponse, h, k, i, [block]sign> とネットワークに伝達する
- もし正しく検証されなければ <ChangeView, h,k,i,k+1> とネットワークに伝達する



図：代表者がブロック提案をレビューして返答する

7. s 個の'prepareResponse' の伝達を受信したのち、代表者はコンセンサスを形成し、ブロックを発行する。
8. 代表者はブロックにサインする。



図：コンセンサスが形成され、賛同する代表者がブロックにサインしてブロックチェーンに格納する。

9. コンセンサスノードが完全なブロックを受信したとき、そのときの view のデータは一掃され、次のラウンドのコンセンサス形成が始まる。

7.11 障害耐性度

BFT のアルゴリズムは n 個のノードを含むコンセンサスシステムに $f = (n - 1) / 3$ の障害耐性を提供します。この障害耐性度はセキュリティとユーザビリティ含み、どんなネットワーク環境にも適しています。ノードからのリクエストデータは送信者の署名を含むので、悪意のある記録ノードは偽ってリクエストできません。その代わり、悪意のあるノードはシステムをフォークさせることで情報を過去のものに戻そうとします。仮にこのシステムのネットワークにおいてコンセンサスノードが 3 つの部分 R_1 、 R_2 、 F に分かれ、それぞれの間には $R = R_1 \cup R_2 \cup F$ 、 $R_1 \cap R_2 = \emptyset$ 、 $R_1 \cap F = \emptyset$ 、 $R_2 \cap F = \emptyset$ が成り立つとします。さらに、 R_1 と R_2 が善良な記録ノードで自分と同じ

セットにいるノードとしかコミュニケーションが取れないようなサイロにいるとします。そして F は全て悪意のあるノードだとします。さらに F によるネットワークのコンディションによって F は R1、R2 を含む全てのノードとコミュニケーションが取れるとします。もし F がフォークしようとするならば R1 とコンセンサス形成を行い、ブロックを作ります。さらに F は R2 にはこのことを知らせないまま R2 と次のコンセンサス形成を行い、R1 とのコンセンサスを取り消します。この状況に至るためには $|R1| + |F| \geq n - f$ と $|R2| + |F| \geq n - f$ が必要です。最悪のシナリオでは $|F| = f$ 、すなわちシステムが耐えられる悪意のあるノードの数が最大になるのは $|R1| \leq n - 2f$ と $|R2| \geq n - 2f$ であり、全て合わせると $|R1| + |R2| \geq 2n - 4f$ すなわち $n \leq 3f$ である。つまり $f = (n - 1) / 3$ であるとすると、システムはフォークされることはないのです。

第 8 章

SDK や開発について

この章では、NEO での開発を促進するための SDK や開発ツールの操作方法について解説します。

8.1 neo-local

neo-local とは、自身の PC やサーバ上で瞬時に NEO ブロックチェーンとその周辺ツールを展開できる、NEO dApp 向けのブロックチェーン構築ツールです。本ツールを使用することによって、環境構築を完了し NEO での開発をすぐに始めることができます。対応 OS は MacOS、Linux、Windows です。本ツールは、Ethereum の Ganache に相当します。

neo-local を使用すると、複数の Docker イメージが展開されます。

neo-local は、次のサービスによって構成されます。

- neo-local-faucet (開発用フォーセット)
- neo-privatenet (ローカル実行用のプライベートネット)
- neo-python (開発用 CLI)
- neo-scan-api (ブロック参照ツールの API)
- neo-scan-sync (ブロック参照ツールのブロックチェーンとの同期)
- postgres (neoscan 用のデータベース)

8.1.1 インストール

本項では、neo-local をインストールする方法を解説します。それぞれのプラットフォームにおいて、neo-scan を稼働させるには Docker、Docker Compose、Git が必要なので

事前にインストールをしてください。

git コマンドを使用して、neo-local のリポジトリをクローンします。

```
git clone https://github.com/CityOfZion/neo-local.git
```

8.1.2 起動

本項では、neo-local を起動する方法を解説します。

Linux, MacOS

neo-local をクローンしたディレクトリに移動します。

```
cd ./neo-local
```

neo-scan を起動します。

```
make start
```

neo-scan を停止させる場合は次のコマンドを実行します。

```
make stop
```

Windows

neo-local をクローンしたディレクトリに移動します。

```
cd ./neo-local
```

neo-scan を起動して、neo-python にアクセスします。

```
docker-compose up -d --build --remove-orphans  
docker exec -it neo-python np-prompt -p -v
```

neo-scan を停止させる場合は次のコマンドを実行します。

```
docker-compose down
```

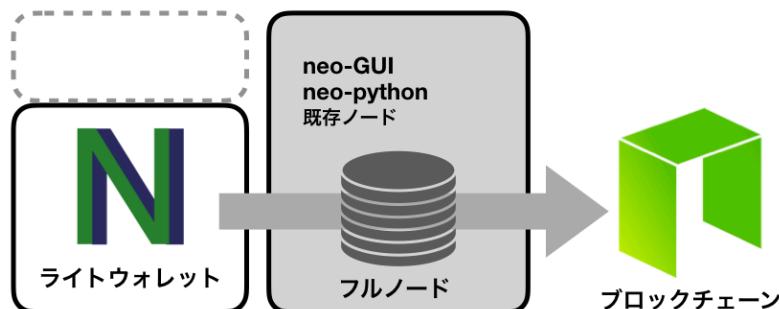
8.2 neo-local-faucet

ブロックチェーンの操作や、スマートコントラクトをデプロイする際には、手数料として NEOGas が必要になります。NEO には、開発の際の手数料を補充する手段として、ファーセット（蛇口）が用意されています。neo-local に含まれているので、単体でインストールする必要はありません。

8.3 neonjs

neon-js とは、ウォレット機能、トランザクションの操作機能、その他便利機能を搭載した NEO ブロックチェーンのための Javascript SDK です。City Of Zion によってオープンソースで開発されており、NeonWallet という NEO 用ウォレットアプリで実際に使われています。

neon-js は、ノードとして機能しないライトウォレットです。前章までで使用した neo-python には、フルノードとして多くの機能が搭載されています。図 8.1 にフルノードとライトウォレットの違いを示します。



▲図 8.1 ライトウォレットとフルノードの違い

ライトウォレットとフルノードには、決定的な違いとしてブロックチェーンとの同期を行うか否かという点があります。フルノードは、ブロックチェーンとの同期のためにブ

ロックチェーンのすべてのデータを内部に取り込み、自身がノードとしてブロックチェーンのやり取りを行います。これに対し、ライトウォレットでは同期をせずブロックチェーンとのやり取りをフルノードに委任します。よって、ライトウォレットでは内部にブロックチェーンのデータをもつことなく、さまざまな処理を行うことが可能となっています。ライトウォレットは、機能がシンプルで必要とする容量が少ないとことから、スマートフォンやブラウザのアプリケーションで展開しやすいという特徴があります。

8.4 neo-python

neo-python とは、City Of Zion によって開発された NEO ブロックチェーンのための Python SDK です。neo-python では、主に次の機能を備えています。neo-local に含まれているので、単体でインストールする必要はありません。

- Python ベースの P2P ノード機能
- インタラクティブな CLI
- スマートコントラクトのビルド、デプロイ、実行
- Python バーチャルマシンのブロックチェーン上でスマートコントラクトを実行
- 基本的なウォレット機能
- NEP2、NEP5 に対応したウォレット機能
- RPC クライアント機能
- RPC サーバ機能
- Notification Server 機能
- Runtime.Log や Runtime.Notify によるイベントモニタリング機能

8.4.1 プロンプト操作

wallet 関連の操作

```
#ウォレットを開く
open wallet {wallet_path}

#ウォレットを作成する
create wallet

#ウォレット情報を表示する
wallet

#ウォレットを閉じる
wallet close
```

```

#ウォレットを再構築する
wallet rebuild

#指定したブロックの高さからウォレットを再構築する
wallet rebuild {number}

#コントラクトによるトークンの送信
wallet tkn_send {token symbol} {address_from} {address to} {amount}

#ウォレットによるトークンの送信
wallet tkn_send_from {token symbol} {address_from} {address to} {amount}

#ウォレットからトークンを削除する
wallet delete_token {token_contract_hash}

#GAS の配当を請求する
wallet claim

#wif キーからアカウントをインポートする
import wif {wif}

#nep2 暗号化済みキーからアカウントをインポートする
import nep2 {nep2_encrypted_key}

#wif を出力する
export wif {address}

#nep2 暗号化済みキーを出力する
export nep2 {address}

```

トランザクションに関するコマンド

```
#JSON 形式で書かれたトランザクションに署名する
sign {transaction in JSON format}
```

スマートコントラクトに関するコマンド

```

#スマートコントラクトのイベントログを表示するように設定する
config sc-events {on/off}

#ブロックチェーンに影響しないテスト用の Invoke を行う
testinvoke {contract hash}

#オプション:
--attach-neo={amount}
--attach-gas={amount}
--from-addr={addr}
--no-parse-addr (parse address strings to script hash by bytearray)

```

```
#ストレージへの書き込み結果の表示設定（標準は on になっている）
debugstorage {on/off/reset}
```

その他のコマンド

```
#現在のノード情報を表示
open wallet

#プロンプトを終了する
quit または exit

#コマンドリストの表示をする
help

#指定したアカウントの情報を表示する
account {address}
```

8.5 Neoscan

neoscan は、Web アプリケーションと API サーバによって構成されたソフトウェアです。NEO ブロックチェーンとの同期をとり、ブロックチェーン上で行われたトランザクションや、デプロイしたスマートコントラクトの情報を即座に参照することができます。

<https://neoscan.io/> にアクセスすることで、NEO の Mainnet に接続している neoscan を利用することができます。

現在の neo-scan は、次の複数のサービスにより構成されています。

- neo-scan-api (ブロック参照ツールの API)
- neo-scan-sync (ブロック参照ツールのブロックチェーンとの同期)
- postgres (データベース)

8.6 その他のツール

これまでに解説したツールの他にも、NEO dApp 開発に役に立つツールや情報は存在します。Awesome NEO(<https://github.com/CityOfZion/awesome-neo>)という Github のリポジトリでは、NEO の関連ツールへのリンクがまとめられており、多くの NEO 技術者の助けとなるでしょう。

第 9 章

スマートコントラクト

スマートコントラクトの概念自体は、ビットコインが生まれる前から存在しており、1990 年代に暗号学者の Nick Szabo によって提唱されたのが始まりとされています。Nick Szabo は、スマートコントラクトの例として自動販売機を挙げています。自動販売機は、「ユーザが必要な金額を自動販売機に導入する」「ユーザが購入したい商品のボタンを押す」という 2 つの条件が満たされたときに「ユーザに商品を提供する」という売買契約が自動的に行われます。このように、あらゆる契約行動を当事者の関与無しに自動的に実行されるように定義し、契約そのものが執行を実行するという概念が、広義のスマートコントラクトとして知られています。

本書では、狭義のスマートコントラクトとして「ブロックチェーン上で動作する契約の自動執行プログラム」と定義します。

9.1 NEO Smart Contract 2.0

NEO は、堅牢性、高パフォーマンスを重視し、さらに拡張性をもつよう設計されています。

パフォーマンスの観点から、NEO ではスマートコントラクトの実行環境として、NeoVM (NEO Virtual Machine) を使用しています。NeoVM は軽量かつ高速に起動することから、スマートコントラクトのような短いプログラムには適しており、JIT と呼ばれるリアルタイムコンパイラによって、スマートコントラクトを環境に合わせてコンパイルしています。

スマートコントラクトをブロックチェーン上に展開し、利用可能な状態にすることをデプロイ (Deploy) といいます。展開されたスマートコントラクトは、スクリプトハッシュと呼ばれる文字列に紐付けられ、以降はスクリプトハッシュを介してスマートコントラクトを呼び出せるようになります。

NEOは、NeoVMによって外部とのデータのやり取りができるように設計されており、スマートコントラクト内で外部システムや他のブロックチェーンのデータにアクセスが可能です。また、NeoVMはAPIを介してブロックチェーンの操作やスマートコントラクトの情報の取得を可能にしています。詳細は<http://docs.neo.org/en-us/sc/reference/api.html>を参照してください。

スマートコントラクトの変更について

ブロックチェーン上に展開されたスマートコントラクトは、たとえ定義した本人であっても後からルールを変更する事はできないように設計されています。一度公開したプログラムが修正できないという制約に最初は抵抗を感じるかもしれません。しかし、スマートコントラクトが契約そのものであるならば当たり前のことです。現実世界において、一度規定し誰かが施行している可能性のある契約を、都合が悪いから変更するという行為は許されることではありません。内容の変更を要する場合であれば、契約内容を修正した書類などを新たに用意して、現行の契約の代わりに新しい契約に同意してもらう必要があります。ブロックチェーンにおいてもこれは同じで、内容に変更を加える場合には修正したスマートコントラクトを別のスマートコントラクトとして新たに展開する必要があります。

9.2 スマートコントラクトの対応言語

NEOでは、作成したスマートコントラクトをコンパイルし、共通の命令言語に変換してからNeoVMに渡すため、複数の言語に対応しています。

現在、NEOでは次の言語に対応しています。

- C#, VB.Net, F#
- Java, Kotlin

その他にも、City of Zionの提供するSDKによって次の言語でもスマートコントラクトの記述が可能となっています。ただし、これらの言語では対応状況によって、NEOの正式に対応している言語よりも制約がある場合が多いです。

- Python
- Javascript, Typescript

- Go

9.3 スマートコントラクトの実例

本節では、NEO のスマートコントラクトを使用した実例を紹介します。

9.3.1 音楽コンテンツの自己出版

アーティストのコンテンツ配信にも、スマートコントラクトが応用されています。現状の音楽コンテンツは、アーティストがレーベル会社と契約を締結することで、コンテンツ配信企業がエンドユーザーにコンテンツを届けるという構造になっています。これに対して imusify (<https://imusify.com>) では、ブロックチェーン上に音楽コンテンツを自己出版できる音楽経済プラットフォームを構築しています。このプラットフォームでは、アーティストはレーベル会社といった中間業者無しに、所有権を明確にしたままコンテンツをエンドユーザーに販売することが可能となっています。また、今後のロードマップではアーティスト同士のコラボレーションやチケットの販売、クラウドファンディングなどが可能になる見込みです。

9.3.2 コンテンツ作成者・所有者の管理と証明

Red Pulse (<https://www.redpulse.com>) は、中国市場に関する質の高い情報を、経済に関心をもつ投資家やアナリスト、アドバイザーに提供することを目指した事業です。Red Pulse の提供するプラットフォームでは、投稿されたコンテンツに対して作成者や所有者といった情報がスマートコントラクトによりブロックチェーン上に記録されます。スマートコントラクトを介してブロックチェーン上にデータを読み書きすることによって、透明性の高く信頼性のある権利の証明や管理を可能にしています。

9.4 手数料形態

スマートコントラクトを実行する際には、手数料として GAS をブロックチェーンに対して支払う必要があります。手数料はスマートコントラクトの実行時に呼び出される NeoVM のシステムコールの種類と数によって決定されますが、現在の NEO ブロックチェーンでは、毎回のスマートコントラクトの実行に対し 10GAS 分の無料枠が設定されています。このため、10GAS を超えないスマートコントラクトの実行には手数料が発生しません。システムコールに対する手数料は、表 9.1 に示すとおりです。この他にも、

スマートコントラクトの呼び出しを優先的に行うために、ユーザは追加で任意の手数料(1GAS～)を支払うことができます。

▲表 9.1 システムコールに対する手数料

システムコール	役割	手数料(GAS)
Runtime.CheckWitness		0.2
Blockchain.GetHeader	ヘッダーの取得	0.1
Blockchain.GetBlock	ブロック情報の取得	0.2
Blockchain.GetTransaction	トランザクション情報の取得	0.1
Blockchain.GetAccount	アカウント情報の取得	0.1
Blockchain.GetValidators	バリデーターの取得	0.2
Blockchain.GetAsset	アセットの取得	0.1
Blockchain.GetContract	コントラクト情報の取得	0.1
Transaction.GetReferences	参照の取得	0.2
Account.SetVotes	投票	1
Validator.Register	バリデーターの登録	1000
Asset.Create (system asset)	アセットの作成	5000
Asset.Renew (system asset) [per year]	アセットの更新(年)	5000
Contract.Create	スマートコントラクトの作成	100~1000
Contract.Migrate	スマートコントラクトのデプロイ	100~1000
Storage.Get	ブロックチェーン上のストレージからデータを取得	0.1
Storage.Put	[per KB]* ストレージにデータを書き込む(KB毎)	1
Storage.Delete	ストレージのデータを削除	0.1
(Default)	必ず発生する手数料	0.001

デプロイに必要な手数料

スマートコントラクトのデプロイ時に必要な手数料は 100~1000GAS となっています。この手数料は、スマートコントラクトをブロックチェーンにデプロイする通常手数料 100GAS に加えて、定義したスマートコントラクトがストレージ領域やダイナミックコールを使用するか否かによって変動します。例えば、そのスマートコントラクトがストレージ領域を必要とするのであれば追加で 400GAS、通常手数料と合わせて合計 500GAS が必要です。また、コントラクト内で他のスマートコントラクトを呼び出すダイナミックコールを必要とするのであれば追加で 500GAS、通常手数料と合わせて合計 600GAS が必要です。ストレージ領域とダイナミックコールを両方とも必要とするのであれば、手数料は合計で 1000GAS になります。なお、実際にスマートコントラクトをデプロイする際には、上記の手数料から無料枠 10GAS 分を差し引いたものが請求されます。

第 10 章

はじめての dApps 開発

この章では、実際に NEO を使った dApps の開発手法について説明します。シンプルなコントラクトを実行できるための手順についてまとめています。

10.1 概要

NEO を使った dApps の開発として、今回はこの手順で開発を行っていきます。なお、「フロントエンドからのスマートコントラクトの呼び出し」では、Javascript でのフロントエンドの開発を行ったことがある方向けに、作成したスマートコントラクトとのやり取りを解説しています。

- NEO のローカル環境の構築
- スマートコントラクトの開発
- スマートコントラクトのコンパイル
- スマートコントラクトの Deploy
- スマートコントラクトの Invoke
- フロントエンドからのスマートコントラクトの呼び出し

まずは簡単なコントラクトを実行できることを目指にしてみましょう。それでは、NEO のローカル環境の構築から始めていきます。

10.2 NEO のローカル環境の構築と起動

ここでは環境構築を簡単にするために neo-local を使用します。neo-local のインストールについては、8 章の「SDK や開発について」の neo-local の節に詳しくまとめてありますので、ここでは割愛します。

それでは、neo-local を起動し実行環境を準備しましょう。

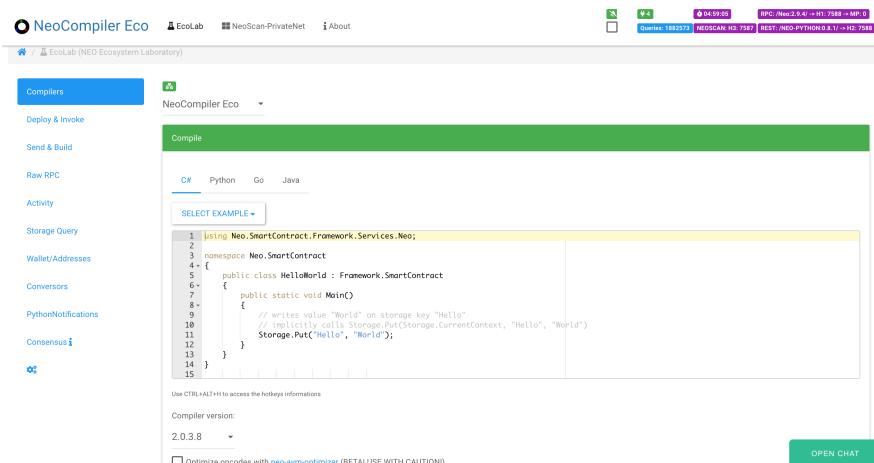
```
# neo-local の起動
$ cd ./neo-local
$ make start

[neo-local] Fetching Docker containers...
~省略~
[neo-local] Starting Docker containers...
~省略~
[neo-local] Waiting for network.....
[neo-local] Network running!
[neo-local] Attaching terminal to neo-python client
~省略~
neo>
```

複数のコンテナが立ち上がったのち、プロンプトが起動します。ローカルに NEO の実行環境を構築できたら早速スマートコントラクトの開発を行ってみましょう。

10.3 スマートコントラクトの開発

今回はできるだけ簡単にスマートコントラクトの開発を行うために、NeoResearch コミュニティが開発した neocompiler<<https://neocompiler.io>>を使用します。



▲図 10.1 neocompiler.io

Neocompiler ではブラウザ上で次の操作を行うことができます。

- スマートコントラクトの作成 (C#, Python, Go, Java)
- 作成したスマートコントラクトのコンパイル
- コンパイルしたスマートコントラクトのデプロイ
- デプロイスマートコントラクトへのアクセス

シンプルなスマートコントラクトであれば、Neocompiler のみで開発を行うことができます。

それでは実際にスマートコントラクトの作成を始めていきましょう。

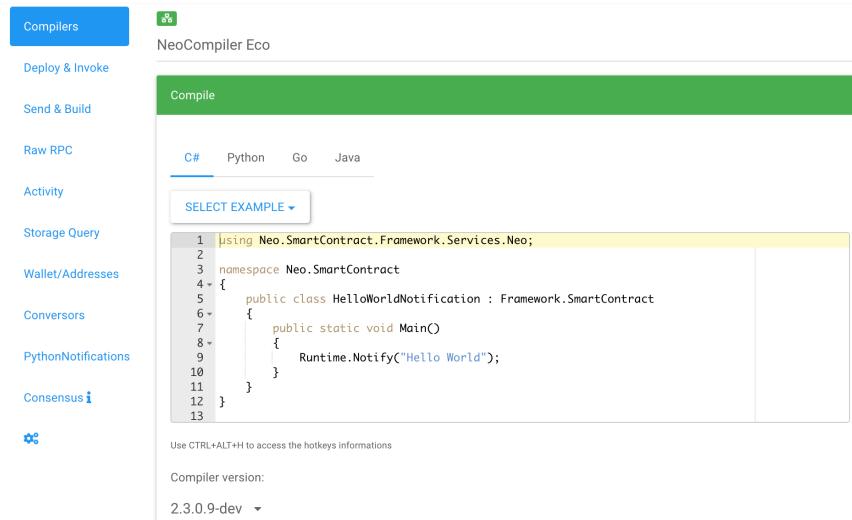
10.3.1 スマートコントラクトの作成

まずは Neocompiler にアクセスします。

Neocompiler

<https://neocompiler.io/#/ecolab>

今回は Example (サンプルコード) の中から、C#の"HelloWorldNotification.cs"を使用して手順を説明します。



▲図 10.2 "Select Example" から "HelloWorldNotification.cs"を選択する

"HelloWorldNotification.cs"のコントラクトの中身は次のとおりです。

```

using Neo.SmartContract.Framework.Services.Neo;

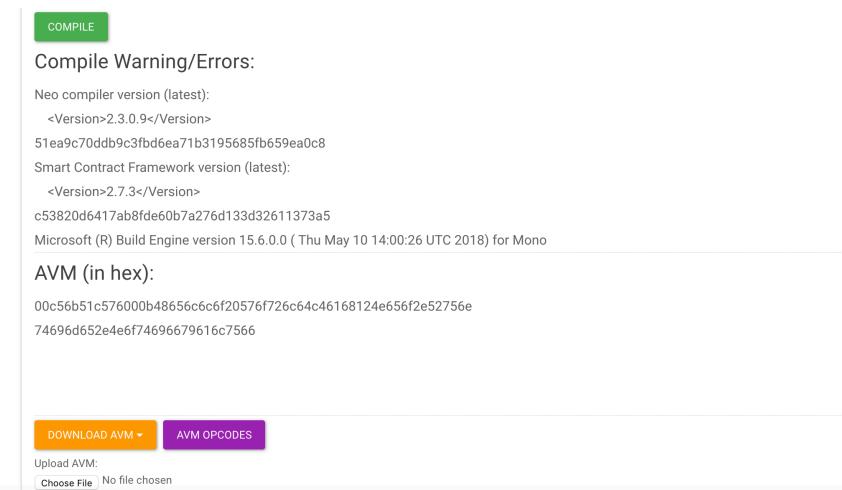
namespace Neo.SmartContract
{
    public class HelloWorldNotification : Framework.SmartContract
    {
        public static void Main()
        {
            Runtime.Notify("Hello World");
        }
    }
}

```

これは、実行時に"Hello World"と表示するための簡単なコントラクトです。
それでは、この作成したコントラクトをコンパイルしてみましょう。

10.3.2 作成したスマートコントラクトのコンパイル

画面の下の方に移動すると [Compile] と書かれたボタンがあるのでクリックします。このとき、BaseNetwork に"NeoCompiler Eco"が選択されていることを確認してください。



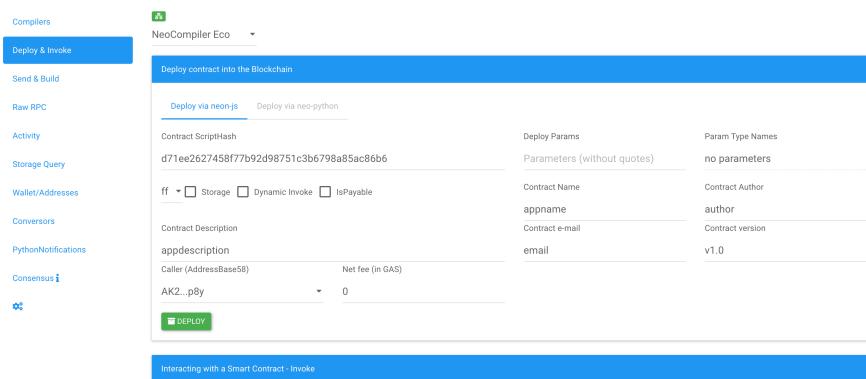
▲図 10.3 コンパイル終了後の画面

C#で書かれた言語を NEO が理解できるように AVM 形式に変換しています。コンパイルが終了すると上記のように AVM が出力されるのでしばらくお待ちください。
コンパイルが終了したら、コントラクトのデプロイを行ってみましょう。

10.3.3 コンパイルしたスマートコントラクトのデプロイ

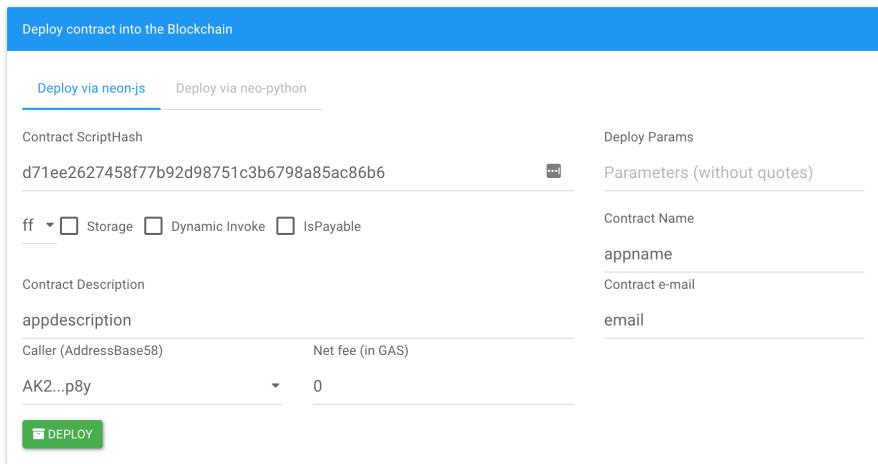
スマートコントラクトをブロックチェーンに展開し、使用可能な状態にしましょう。

左側の [Deploy & Invoke] を選択し、BaseNetwork を Localhost に設定します。これによって、Neocompiler が neo-local で構築したローカルの環境に対して操作を行えるようになります。



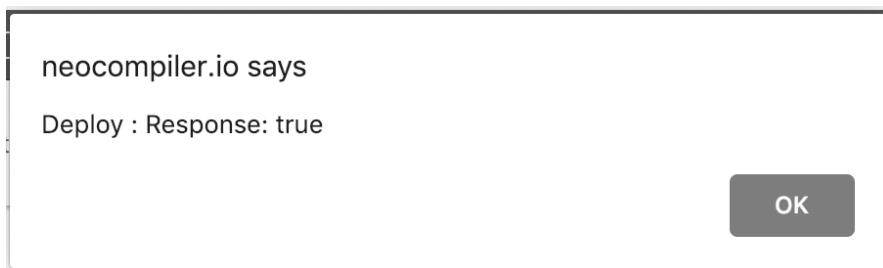
▲図 10.4 Deploy & Invoke

画面の下の方にある"Deploy contract into the Blockchain"の"DEPLOY"をクリックします。



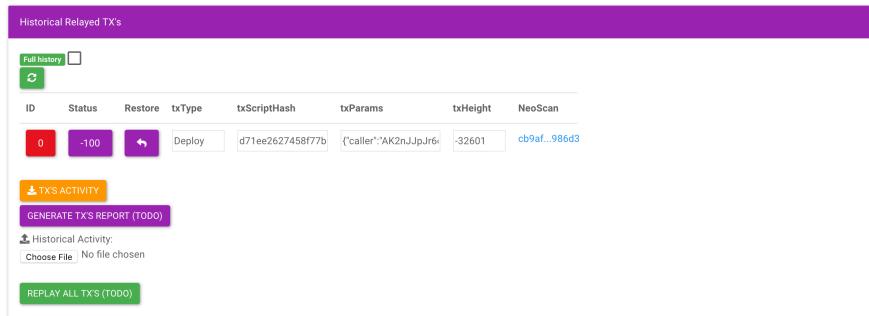
▲図 10.5 Deploy Button をクリック

次のようなポップアップメッセージが表示され、Activity ページに遷移します。



▲図 10.6 Deploy が成功

ポップアップメッセージの OK を押した後、Activity ページの "Historical Relayed Tx's" にトランザクションが表示されます。



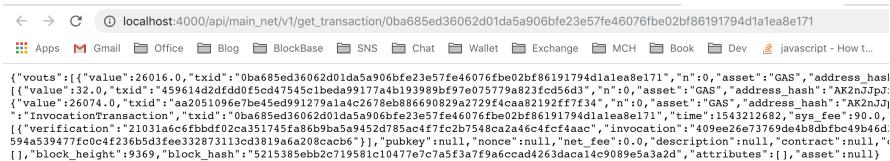
▲図 10.7 トランザクションの状態はここで確認可能

デプロイ時には neo-local にもメッセージが表示されます。

```
neo> [I 181126 06:11:26 EventHub:62] [SmartContract.Contract.Create][9369] [d71ee2627458f77b92d98751c3b6798a85ac86b6] 
[{'version': 0, 'code': {'hash': '0xd71ee2627458f77b92d98751c3b6798a85ac86b6', 'script': '00c56b51c576000b48656c6c6f29f', 'appname': 'code_version': 'v1.0', 'author': 'author', 'email': 'email', 'description': 'appdescription', 'properties': {}}, 'txType': 'Deploy', 'txScriptHash': 'd71ee2627458f77b92d98751c3b6798a85ac86b6', 'txParams': {'caller': 'AK2nJpJr6v'}, 'txHeight': 32601, 'NeoScan': 'cb9af...986d3'}]
[I 181126 06:11:26 EventHub:58] [test_mode][SmartContract.Runtime.Notify] [d71ee2627458f77b92d98751c3b6798a85ac86b6]
[I 181126 06:11:26 EventHub:58] [test_mode][SmartContract.Runtime.Notify] [d71ee2627458f77b92d98751c3b6798a85ac86b6]
[I 181126 06:11:26 EventHub:58] [test_mode][SmartContract.Runtime.Notify] [d71ee2627458f77b92d98751c3b6798a85ac86b6]
[I 181126 06:11:26 EventHub:58] [test_mode][SmartContract.Execution.Success] [d71ee2627458f77b92d98751c3b6798a85ac86b6]
[b'name'], {'type': 'ByteArray', 'value': bytarray(b'')}, {'type': 'ByteArray', 'value': b'symbol'], {'type': 'ByteArray', 'value': b'symbol'}]
[I 181126 06:11:26 EventHub:62] [SmartContract.Execution.Success][9369] [e8726be255c865aff903b7ef3427bf02a008] [txDropInterface'] 'value': {'version': 0, 'code': {'hash': '0xd71ee2627458f77b92d98751c3b6798a85ac86b6', 'script': '00c56b51c576000b48656c6c6f29f', 'appname': 'code_version': 'v1.0', 'author': 'author', 'email': 'email', 'description': 'appdescription', 'properties': {}}, 'txType': 'Deploy', 'txScriptHash': 'd71ee2627458f77b92d98751c3b6798a85ac86b6', 'txParams': {'caller': 'AK2nJpJr6v'}, 'txHeight': 32601, 'NeoScan': 'cb9af...986d3'}]
```

▲図 10.8 Deploy 時の NEO 環境のメッセージ

デプロイが終了すると、"Histrical Relayed Tx's" の "TX on NeoScan" にもトランザクションを確認することができます。

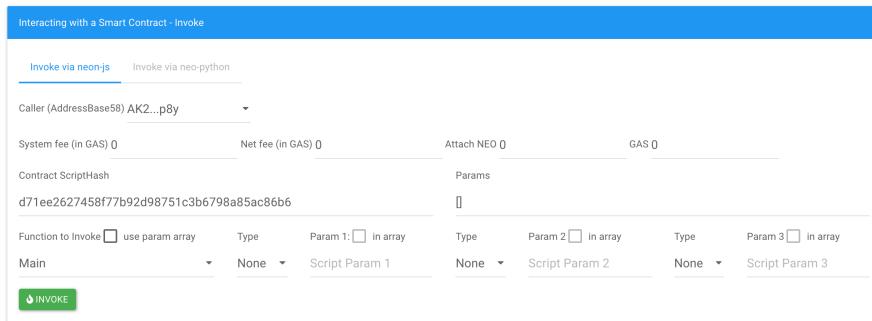


▲図 10.9 NeoScan のメッセージ

これでスマートコントラクトのデプロイは完了です。それでは、デプロイスマートコントラクトへのアクセスをしてみましょう。

10.3.4 デプロイスマートコントラクトへのアクセス

左側の [Deploy & Invoke] を選択し、"Interacting with a SmartContract"から Invoke をクリックします。



▲図 10.10 Invoke Button をクリック

コントラクトが実行されると、NEO 環境に次のようなメッセージが出力されていると思います。

```
[SmartContract.Runtime.Notify][9385] [d71ee2627458f77b92d98751c3b6798a85ac86b6]
[tx 82df6bf2d20410c77c87e1f02c339be13184ace3d1669ae8aef912d39c426ea7]
{'type': 'Array', 'value': [{'type': 'ByteArray', 'value': b'Hello World'}]}

,
['type': 'ByteArray', 'value': bytearray(b'')}, {'type': 'ByteArray', 'value': b'symbol'}, {'type': 'ByteArray', 'value': b'face'}, {'value': {'version': 0, 'code': {'hash': '0xd71ee2627458f77b92d98751c3b6798a85ac86b6', 'script': '0xc56b51c57600004865', 'name': 'appname', 'code_version': 'v1.0', 'author': 'author', 'email': 'email', 'description': 'appdescription', 'properties': {}}}, 9
neo> [I 181126 06:11:26 EventHub:62] [SmartContract.Execution.Success][9369] [e8726be255c865aff903b7ef3427bf02a008] [tx 0ba685]
[I 181126 06:16:15 EventHub:62] [SmartContract.Runtime.Notify][9385] [d71ee2627458f77b92d98751c3b6798a85ac86b6] [tx 82df6bf2d20410c77c87e1f02c339be13184ace3d1669ae8aef912d39c426ea7]
[I 181126 06:16:15 EventHub:62] [SmartContract.Execution.Success][9385] [d71ee2627458f77b92d98751c3b6798a85ac86b6] [tx 82df6bf2d20410c77c87e1f02c339be13184ace3d1669ae8aef912d39c426ea7]
13  }
14  }
15 }
```

▲図 10.11 neo 環境上のデバッグログ

以上がスマートコントラクトの簡単な開発手順です。それでは、次にデプロイしたコントラクトにアクセスするためのフロントエンドを Javascript で作成してみましょう。

10.4 フロントエンドからのスマートコントラクト呼び出し

ここでは NEO 用 JavascriptSDK である Neon.js を使用します。npm または CDN でインストールすることができます。

```
# npm
npm i @cityofzion/neon-js
# CDN
<script src="https://unpkg.com/@cityofzion/neon-js" />
```

サンプルコードは次のとおりです。

▼ sc-access.js

```
//Neon.js の読み込み
var Neon = require('@cityofzion/neon-js');

//API provider を読み込んで API の作成
//localhost の NeoScan の API を指定
var apiProvider = new Neon.api.neoscan.instance('http://localhost:4000/
    api/main_net')

//Private Key(WIF) の読み込んで Account のインスタンスを作成
var acct = new Neon.wallet.Account(
    //WIF
    "KxDgvEKzgSBPPfuVfw67oPQBSjidEiqTHURKSDL1R7yGaGYAeYnr"
);

//Script の作成
var script = Neon.default.create.script({
    //Contract デプロイ時に生成される ScriptHash
    scriptHash: "d71ee2627458f77b92d98751c3b6798a85ac86b6",
    //今回は Method がないのでブランク
    operation: "",
    //今回は arguments がないのでブランク
    args: []
})

//api, account の用意
var config = {
    api: apiProvider, //The network to perform the action, MainNet or TestNet
    account: acct, // This is the address which the assets come from.
    script: script,
    gas: 1
};

Neon.default.doInvoke(config)
    .then(config => {
        console.log(config.response);
    })
    .catch(config => {
```

```
    console.log(config);
});
```

neo-local を立ち上げたのち、上記のコードを実行すると、コンソール画面に "Hello world" と表示されるはずです。これで、先ほど作成した NEO のコントラクトを呼び出すことができました。

以上が、はじめての dApps 開発となります。

Neon-js には NEO の開発を促進するさまざまな機能があるので、詳しい使い方は下記を参考にしてみてください。

neon-js

<http://cityofzion.io/neon-js/docs/en/installation.html>

第 11 章

NEP

この章では、前半に NEP の概要やタイプ・その出し方の基本について説明するとともに、後半では実際の NEP の内容を紹介し、NEP への理解を深められるように解説していきます。

11.1 概要

NEP とは、NEO Enhancement Proposal の略称であり、NEO のユーザーとコミュニティに対して、NEO に関する情報を提供したり、NEO の実装環境を改善する新しいアイデアを提案するために、git 上にデザインされたドキュメントのことです。NEP には、NEO に実装されている既存デザインの詳細やそれを改善するためのアイデアがまとめられているので、ユーザーが NEO の実装環境やその変更内容を確認するのに便利です。NEP を執筆することは誰でも可能ですが、次のような前提があります。

- 内容の技術的な部分についてその根拠とともに明記すること
- NEO のコミュニティに対して合意（コンセンサス）を形成しつつ、その反対意見についてもドキュメント化すること

理想は、NEO の情報をまとめたり、実装環境を改善したユーザーが完了済みの NEP をリスト化し、NEO を取りまく現状について全てのエンドユーザーが簡単に理解できるようにすることです。

11.2 タイプ

次は NEP のタイプについて説明していきます。NEP には次の 3 つがあります。

11.2.1 スタンダードトラック型

スタンダードトラック (Standards Track) 型 NEP は、NEO の実装環境の大部分または全部に影響を与える何かしらの変更を提案しています。たとえば、次のような内容です。

- ネットワークプロトコルに対する変更
- ブロックやトランザクションの評価ルールにおける変更
- アプリケーションの標準 (standards) / 系統 (conventions) の提案
- NEO を用いたアプリケーションの相互運用性に影響を与えるその他の変更内容や追加機能

11.2.2 インフォーメーション型

インフォーメーション (Information) 型 NEP は、NEO の既存デザインの詳細についてまとめていたり、NEO のユーザーとコミュニティに対して一般的なガイドラインを提供していますが、NEO を改善する新しいアイデアを提案するわけではありません。インフォーメーション型 NEP には次のような性質があります。

- 必ずしも NEO のコミュニティに対して合意を形成したり、レコメンデーション (推奨) を提供しなくてもいい
- ユーザーには基本的にそのアドバイスに従う義務がない

11.2.3 メタ型

メタ (Meta) 型 NEP は、NEO を取り巻くプロセスや環境についてまとめていたり、その変更内容を提案しています。たとえば、次のような内容です。

- 手続きやガイドライン
- 意思決定プロセスに対する変更
- NEO の開発ツールや開発環境に対する変更

メタ型 NEP には、次のような性質があります。

- スタンダードトラック型 NEP と違い、NEO のプロトコル自体を超える領域に対応している
- インフォーメーション型 NEP と違い、変更内容の提案が NEO のコードベースに

対してではないので、コミュニティの合意が求められる場合が多い

- ユーザーには基本的にそのアドバイスに従う義務がある

11.3 NEP の出し方

11.3.1 アイデア

NEP を出すプロセスは、NEO を改善する新しいアイデアを思いつくところから始まります。この際に次のことが意識されなければなりません。

- NEP にはキーとなる提案や新しいアイデアがひとつ含まれること
- NEP が必要とされる変更は、単一のクライアントではなく複数のクライアントに影響を及ぼしたり、複数のアプリケーションで利用される標準を定義するものであること

NEP の編集者（エディター）は、焦点が絞りきれていない提案を拒否する権利をもっており、NEP の提案内容が疑わしい場合は、焦点をさらに絞ったいくつかの NEP に分けることが好ましいです。

11.3.2 執筆者

NEP には執筆者（チャンピョン）が必要です。執筆者とは、後述するスタイルとフォーマットにそって NEP をデザインし、公開されている適切な議論を追ってアイデアに関するコミュニティの合意を形成しようとする人のことです。

11.3.3 公開

NEP を執筆する前にアイデアを公開することが好ましいです。本来、すでに公開された議論を踏まえてアイデアが拒否されないようにするには、インターネット上の下調べも含めて大量の労力や時間を割く必要があります。そこで、NEP として執筆する前にアイデアを公開して、その有用性を NEO のコミュニティに確認することでそれらを節約できます。NEO のコミュニティ全体にとっても、新しいアイデアの有用性を確かめることは大きなメリットです。

11.3.4 プルリクエスト

アイデアの有用性を NEO コミュニティから認められた後には、NEP の執筆者は草稿状態の NEP をプルリクエストとして提出しましょう。こうすることで、執筆者は草稿状態の NEP を断続的に編集でき、追加で外部から寄せられるコメントなど、アイデアへのフィードバックを活用できます。

11.3.5 編集者の承認

NEP の執筆に協力する編集者がプルリクエストを承認すれば、執筆者はその NEP に番号をつけ、スタンダードトラック型・インフォーメーション型・メタ型のどれかのタイプにラベリングし、「Draft (草稿状態)」とステータスを決めた後に、新しい NEP として git のレポジトリに追加することができます。NEP の編集者が理由なしに NEP を拒否することはないので、NEP のステータスが拒否された場合は、次のような理由が当てはまっているかを検討しましょう。

- 既存の NEP とアイデアが重複している
- アイデアの技術的な部分が不明瞭である
- 適切な動機付け (Motivation) が提供されていない
- 下位互換性 (Backwards Compatibility) への対処がされていない
- NEO の理念に沿っていない

11.3.6 スタンダードトラック型 NEP の出し方

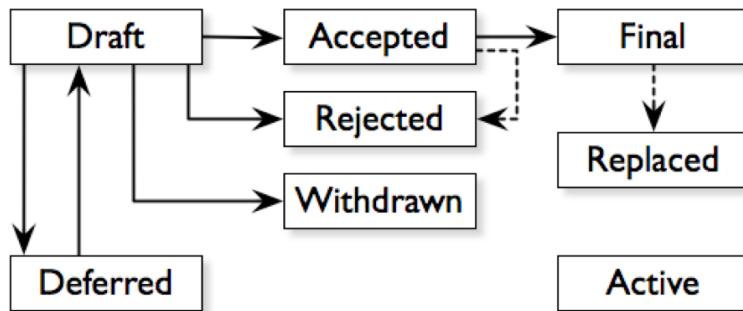
スタンダードトラック型 NEP は次の 3 つの部分で構成されます。

- 設計内容 (デザインドキュメント)
- 実装方法 (インプリメンテーション)
- 最終的な公式版へのアップデート内容

スタンダードトラック型 NEP は、アイデアを実装する前に設計内容を承認されなければなりません。ただし、実装されること自体がそのアイデアの学習と議論を促進する場合は例外となります。その後、コードやパッチ・URL などの形式で実装方法が記載された状態で、最終的な公式版へのアップデートが検討されます。アイデアの実装方法は、堅実なものでありプロトコルを複雑化しすぎないことが重要です。

11.3.7 承認とステータス

NEP は承認の可否やその進捗具合によってステータスが変わります。想定されるステータスの変化と状態を図 11.1 でみてみましょう。



▲図 11.1 NEP のステータスの分岐

ここでは次のような主要なステータスを取り上げます。

- Draft (草稿状態)
 - 草稿の段階である NEP のステータスです。NEP の執筆者のプルリクエストが承認された後につけられます。
- Final (最終状態)
 - 最終的な公式版となった NEP のステータスです。NEP の承認・実装の後、それが NEO のコミュニティに受け入れられるとつけられます。
- Deferred (縹越状態)
 - 該当する NEP に一切の進捗が生まれていない場合につけられるステータスです。ただし、一度このステータスがつけられても執筆者と編集者はそれを「Draft」に戻すことができます。
- Rejected (拒否された状態)
 - NEP が編集者によって拒否された場合につけられるステータスです。いいアイデアではなかったものの、それ自体の記録が残ることには価値があります。
- Active (アクティブな状態)
 - 決して完了することがない NEP に対してつけられるステータスです。イン

フォーメーション型 NEP とメタ型 NEP の場合につけられます。

- Replaced (代替された状態)
 - 新しい NEP に代替された古い NEP につくステータスです。NEP は既存の NEP に依拠したりそれを置き換えることができます。

11.3.8 フォーマットとテンプレート

NEP を出す際には、守らなければならないフォーマットやテンプレートがあります。前提として次のふたつが求められます。

- メディアウィキ (mediawiki) またはマークダウン (markdown) のフォーマットを用いること
- イメージファイルが NEP のサブディレクトリに含まれること

さらに、NEP には次の要素が記載されなければなりません。ただし「Motivation (動機付け)」についてはオプション (選択可能) です。

- Preamble (前文)
 - NEP に関する基礎的情報を RFC822 スタイルのヘッダーで記載します。
- Abstract (概説)
 - アイデアにより対処される技術的な課題を 200 文字以内で記載します。
- Motivation
 - NEO のプロトコルの変更を提案する NEP の場合に、新しい NEP が解決できる課題をなぜ既存のプロトコルでは解決できないのかを説明し、適切な動機付けを行います。
- Specification (仕様)
 - 構文や意味など、新しい変更や機能の技術仕様を記載します。
- Rationale (根拠)
 - アイデアの設計や提案内容の根拠を記載します。NEO のコミュニティの合意形成のためにも重要になるので、議論の過程で生じた反論も考慮しましょう。
- Backwards Compatibility
 - 下位互換性の課題と想定される対処法について記載します。課題へ対処するために十分な論文を参照することが重要です。

- Test Cases (テストケース)
 - アイデアや提案内容の実装をテストした事例について記載します。合意内容に影響を与える事例については記載することが必要不可欠です。それ以外の場合には記載の有無を選択できます。
- Implementations (実装方法)
 - アイデアや提案内容を実装するための方法をコードやパッチ・URL などの形式で記載します。

最後に、NEP を出すために前文を記載する際に守るべきフォーマットとテンプレートについて補足していきます。

```
NEP: 5
Title: Token Standard
Author: Tyler Adams <lllwvlvwlll@gmail.com>, luodanwg <luodan.wg@gmail.com>,
        tanyuan <tanyuan666@gmail.com>, Alan Fong <anfn@umich.edu>
Type: Standard
Status: Final
Created: 2017-08-10
```

2018 年時点の NEP5 では実際にこのように表示されています。

```
NEP: <NEP number>(this is determined by the NEP editor)
Title: <NEP title>
Author: <list of authors' real names and optionally, email address>
*Discussions-To: <email address>
Status: <Draft | Active | Accepted | Deferred | Rejected | Withdrawn | Final
        | Superseded>
Type: <Standard | Informational | Meta>
Created: <date created on, in ISO 8601 (yyyy-mm-dd) format>
*Replaces: <NEP numbers>
*Superseded-By: <NEP number>
*Resolution: <url>
```

これは前文のテンプレートです。RFC822 スタイルのヘッダーで情報が記載されています。*印のついたヘッダーはオプションですが、その他は記載が必須です。

「NEP」と「Title」にはそれぞれ NEP の番号とタイトルを記載します。そして「Author」には、全ての執筆者と管理者（オーナー）の名前を記載しなければなりません。オプションでそれぞれの E メールアドレスを記載する場合もあります。たとえば次のように表示されます。

▼ E メールアドレスを記載する場合

```
Random J. User <address@dom.ain>
```

▼ E メールアドレスを記載しない場合

Random J. User

その他のヘッダーは次のようなものです。

- Resolution
 - スタンダードトラック型 NEP の場合に、NEP に関する宣告が記載された E メールメッセージの URL やその他のウェブリソースを記載します。
- Discussions-To
 - 「Draft」ステータスの NEP など、非公開で議論されている NEP の場合に、議論が行われているメーリングリストや URL を記載します。ただし、執筆者と私的に NEP の議論が行われている場合は「Discussions-To」のヘッダーは必要とされません。
- Type
 - NEP のタイプ（スタンダードトラック型・インフォーメーション型・メタ型）を記載します。
- Created
 - NEP に番号がつけられたに日付を記載します。たとえば NEP5 では 2017-08-10 と記載されています。
- Replaces
 - 既存の NEP を代替した新しい NEP の場合に、代替した古い NEP の番号を記載します。
- Superseded-By
 - 新しい NEP により古くなった NEP の場合に、NEP を代替する新しい NEP の数を記載します。

11.4 実際の NEP

実際の NEP の中で代表的なものをみてみましょう。

ここでは NEP5 と NEP6、そして今後新しく実装される可能性の高い NEOX を取り上げ紹介します。

▼表 11.1 代表的な実際の NEP

番号	タイトル	タイプ	ステータス
1	NEP Purpose and Guidelines	Meta	Active
2	Passphrase-protected private key	Standard	Final
3	NeoContract ABI	Standard	Final
4	Dynamic Contract Invocation	Standard	Replaced
5	Token Standard	Standard	Final
6	Wallet Standard	Standard	Final
7	Triggers for NeoContract	Standard	Final
8	Stack Isolation for NeoVM	Standard	Final
9	URI Scheme	Standard	Final
10	Composite Smart Contracts	Standard	Final
11	Non-fungible Token	Standard	Accepted
	Neoid Standard Stub		
	Neofs Standard Stub		
	Neox Standard Stub		
	Neoqs Standard Stub		

11.4.1 NEP5

NEP5 の概要

NEP5 は、トークン化されたスマートコントラクトが統一的に対応する仕組みを可能にするための、NEO のブロックチェーンのトークン規格を提案しています。

NEP5 の動機

NEO のブロックチェーンが拡大するにつれて、スマートコントラクトの配備と発動はますます重要になってきます。しかし、各コントラクトが互いに対応する規格がなければ、互いに類似性を持っているはずのコントラクトのそれぞれに特有な API に対してメンテナンスが行われなければなりません。トークン化されたコントラクトが動作する基本的な仕組みは全て同じなので、それ自体がこのメンテナンスの有効な実例となります。そこで、トークンに対応する基本的な方法があれば、全体の仕組みの中で、トークンを利用するスマートコントラクト全てに対する基本的動作のメンテナンスを行わずに済みます。

11.4.2 NEP6

NEP6 の概要

NEP6 は、NEO が実装された環境間でウォレットファイルをシェア（共有）することを可能にするためのウォレット規格を提案しています。

NEP6 の動機

現在では異なるクライアントプログラムが異なるウォレットファイルを生成しています。そのため、これらのファイルの形式と保存方法、暗号化方法は全て異なっており、ユーザーが異なるクライアントプログラム間を移動することは難しく、プライベートキーを生成することで移動できた場合でも、複数のキーをもつ種々のウォレットは処理し切れません。そこで、統一的なウォレットの形式が必要とされています。これがあれば、ウォレットのファイルを変えたりプライベートキーを生成する必要もなく、ユーザーは全てのプラットフォームを安全かつ簡単に移動できます。

11.4.3 NeoX

NeoX は、クロスチェーン型の相互運用性を実装するプロトコルへの拡張を提案しています。提案内容は「クロスチェーン型資産交換プロトコル (cross-chain assets exchange protocol)」と「クロスチェーン型分配取引プロトコル (cross-chain distributed transaction protocol)」のふたつに分けられます。

クロスチェーン型資産交換プロトコル

クロスチェーン型資産交換プロトコルとは、図 11.2 のように二本鎖で不可分な資産を交換するプロトコルのことです。



▲図 11.2 クロスチェーン型資産交換プロトコルのイメージ

これは、NEO のプラットフォームに参加する複数のユーザーが異なるチェーン間で資産を交換でき、トランザクションの全プロセスにおいて、ユーザーの一方のみが成功したり失敗することを防ぐためのアイデアです。これが実装されると、両方の資産交換者が確実に結果を共有するようになります。

クロスチェーン型分配取引プロトコル

クロスチェーン型分配取引プロトコルとは、図 11.3 のようにトランザクションの種々のステップが異なるブロックチェーンを介して分散されつつも、トランザクションの全プロセスの一貫性が確保されるプロトコルのことです。



▲図 11.3 クロスチェーン型分散取引プロトコルのイメージ

これはクロスチェーン型資産交換を拡張し、資産交換という行為を任意なものへと変えるアイデアといえます。

後書き

Github 上での貢献

多くの方々が、コメントや、訂正や、加筆を Github 上の初期のドラフトに寄せてくださいました。この本に貢献して下さったすべての皆様に感謝します。Github 上で多大な貢献をして下さった方々は、次のとおりです（GitHub の ID を記載）。

- EG-easy
- genm
- ichikak
- SotaWatanabe
- azusa-tarola
- keigominami
- tomoaki25official
- ookimaki

Starting NEO

2018年12月16日 NEO技術書1版 v1.0.0

著者 NEO Keymakers Japan

イラスト 未定

印刷所 未定

Creative Commons Attribution-ShareAlike 4.0 International License