令和2年度「専修学校による地域産業中核的人材養成事業」
スマートコントラクトを使用したシステム開発人材の育成

スマートコントラクト開発入門

学校法人 麻生塾 麻生情報ビジネス専門学校

目次

2.3 プラットフォームとなるブロックチェーン

2.3.1 Ethereum

2.3.2 NEO

2.3.3 EOS

3章　Ethereum

3.1　Ethereumの概要

3.1.1 歴史

3.1.2 WorldComputer

3.1.3 Bitcoinとの相違点

3.2 Ethereumの構成要素

3.2.1 ネットワーク

3.2.2 ノード

3.2.3 ブロック

3.2.4 トランザクション

3.2.5 アカウント

3.2.6 状態

3.3 Ethereumの処理の流れ

3.3.1 コントラクトの登録

3.3.2 コントラクトの呼び出し

4章　DApps(分散型アプリケーション)

4.1 DAppsの仕組み

4.1.1 既存のアプリケーションの構成

4.1.2 DAppsの構成

環境構築

この教材で行う演習のための、環境構築の方法を説明します。

利用するツール

- Virtual Box
- Ubuntu

Virtual Box

Virtual Boxとは使用しているPCに仮想環境を構築し、他のOSをインストールすることができる仮想化ソフトです。Virtualboxを導入することにより、複数のOSを切り替えて使用することが可能になります。

インストール

- VirtualBoxの公式サイトである、「https://www.virtualbox.org/」を開く
- 「Download VirtualBox 6.0」をクリックする
- ダウンロードを行った後は、VirtualBoxのインストーラを開き、画面の指示に従いVirtualBoxをインストールする

VirtualBoxはバージョン5.0以上で問題なく演習を行うことができると確認できています。また、VirtualBoxを起動する際に、仮想マシンにメモリの割り当てを行います。その際に、メモリを4GB以上割り当てることを推奨します。

Ubuntu

インストール

- Ubuntuのダウンロードページである、「https://jp.ubuntu.com/download」を開く
- Ubuntu Desktop 18.04.3 LTSのダウンロードボタンをクリックする

- Ubuntuのイメージのダウンロードが完了すると、VirtualBoxでUbuntuを起動する

以上で演習に必要な環境の構築は終了です。追加で準備が必要なツールに関しては、その都度導入方法を説明します。

1. ブロックチェーンの概要

この章ではスマートコントラクトの仕組みや特徴、スマートコントラクトを作成する際に必要となるブロックチェーンに関する語句について説明します。

1.1 ブロックチェーン

ここではブロックチェーンとはどのようなものであり、それが持っている特徴や種類について説明します。

1.1.1 ブロックチェーンとは

ブロックチェーンとは1つの大きな技術革新によって新たに作られたものではなく、既存のITや経済学、暗号学などの様々な学問により作られた新しい「仕組み」です。つまり、ブロックチェーンに用いられている個別の要素自体は以前から使われていたものであり、決して新しいものではありません。

では、具体的にブロックチェーンがどのようなものであるか説明します。

ブロックチェーンは一台のコンピュータの中で動かすものではなく、複数のコンピュータで**ネットワークを構築**することで機能します。このネットワークの中では仮想通貨の取引や様々なプログラムが実行され、これらの結果がネットワークの**参加者全員に共有**されます。この記録を**改ざんや削除されることなく、半永久的に保存**することができる点がブロックチェーンの大きな特徴です。これは決して難しいことのようには思えませんが、これまでのシステムで実現することは極めて難しいことでした。

この点について具体例を用いながら説明します。「半永久的保存する」といっても現実味がありませんので、ここではデータを100年間改ざんされることなく、安全に保存することを考えます。

考えることのできる選択肢の1つとして個人が保有するPCにデータを保存するという方法があります。この場合、PCの破損や紛失によりデータが失われてしまうこと

や、パスワードの流出によりデータが改ざんされてしまう可能性を考えることができます。仮に、いくら厳重に保存していたとしても、PCを管理していた人が亡くなってしまった場合には、データの安全は保証されません。

次に企業にデータを預けるという選択肢を考えます。個人でデータを管理するよりも安全なように思え、このようなサービスを利用している人も多いのではないでしょうか。しかし、企業にデータを預ける場合、その企業がデータの保管サービスを辞めることや、企業が倒産するなどしてデータが失われる状況を考えることができます。また、サイバー攻撃や内部犯によりデータが改ざんされる可能性もあります。

これらの方法以外にもデータの保存方法を考えることができますが、確実にデータを保存することは極めて難しいということがわかるのではないかと思います。

以上から、これまでの**個人や企業などがデータを管理する方法では、データが半永久的に安全に保存されることは保証されません**。つまり、特定の”誰か”によってデータが管理されている限り、データには改ざんや紛失の可能性が十分にあります。しかし、これまでの技術では個人や企業などの管理者の存在なしに、記録を保存する仕組みは存在しませんでした。

ブロックチェーンではこの問題を解決し、特定の”誰か”のような**管理者を必要とせずに、データを保存**することができます。これにより、データを**改ざんや削除されることなく、半永久的に保存**することができるようになりました。この点がブロックチェーンに大きな注目が集まっている理由の１つです。具体的なデータの管理方法については、1.2章で説明します。

ブロックチェーンの種類

ブロックチェーンには管理者が存在しないと説明しましたが、管理者の存在するブロックチェーンもあります。この管理者の有無という点からブロックチェーンを2つの種類に分類することができます。1つは管理者の存在しないブロックチェーンである**パブリックブロックチェーン**であり、もう1つは管理者の存在するブロックチェ

ーンであるパーミッションドブロックチェーンです。これらの特徴についてそれぞれ説明します。

パブリックブロックチェーン

パブリックブロックチェーンとは「パブリック」という言葉の通り「開かれたブロックチェーン」であり、誰でもブロックチェーンのネットワークに参加することができます。ブロックチェーンのネットワークに参加するための許可や、使用するコンピュータの性能、OSなどの制約はありません。

パブリックブロックチェーンに分類されるブロックチェーンには「価値の移転に特化したブロックチェーン」であるBitcoinや、「分散型アプリケーションのプラットフォーム」であるEthereumなどを挙げることができます。

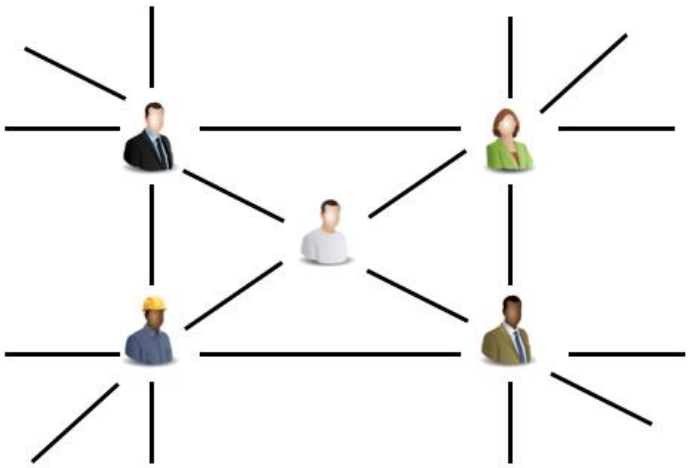


図1.1　パブリックブロックチェーンのイメージ

パブリックブロックチェーンのネットワークに参加するためにはWebサイトなどで配布されているクライアントソフトを利用します。クライアントソフトとはそれぞれのブロックチェーンの仕様を満たして作られたソフトウェアです。以下にBitcoinとEthereumのクライアントソフトを入手することのできるWebサイトを紹介します。今回紹介する「BitociCore」と「Go Ethereum」はそれぞれBitcoinとEthereumのネットワークで最もシェアが高いクライアントソフトです。

- BitcoinCore(https://bitcoincore.org/ja/)
- Go Ethereum(https://geth.Ethereum.org/)

パーミッションドブロックチェーン

パーミッションドブロックチェーンは、パブリックブロックチェーンと異なり、**ネットワークへの参加に管理者の許可が必要**となるブロックチェーンです。企業や団体などの組織内や組織間で利用されます。パーミッションドブロックチェーンに分類されるブロックチェーンとしては、Hyperledger Fabricが挙げられます。Hyperledger Fabricは具体的なブロックチェーンの名前ではなく、Hyperledgerというプロジェクトの中の1つであり、目的に応じたブロックチェーンを作成するためのプラットフォームです。Hyperledger Fabricを目的に応じて改変し、独自のブロックチェーンを作成することができます。

パーミッションドブロックチェーンはブロックチェーンの管理者の数によって「**プライベートブロックチェーン**」と「**コンソーシアムブロックチェーン**」の2種類に分類することができます。

パーミッションドブロックチェーンであるプライベートブロックチェーンと、コンソーシアムブロックチェーンについて説明します。

プライベートブロックチェーン

プライベートブロックチェーンとはブロックチェーンを**管理者または、管理団体が1つ**であるパーミッションドブロックチェーンです。企業や団体内での利用が想定されます。

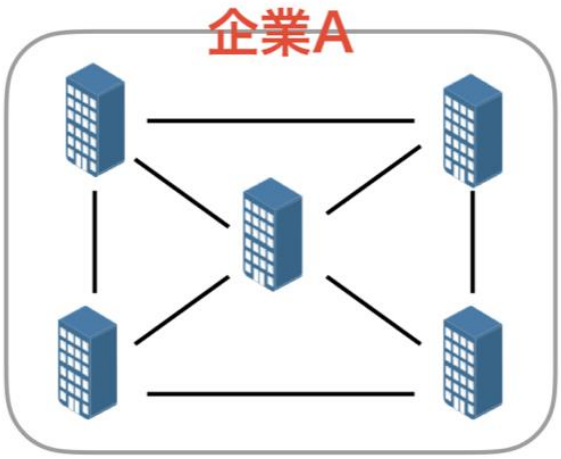


図1.2　プライベートブロックチェーンのイメージ

コンソーシアムブロックチェーン

　コンソーシアムブロックチェーンとは、ブロックチェーンを**管理者または、管理団体が複数存在**するパーミッションドブロックチェーンです。複数の企業や団体間での利用が想定されます。

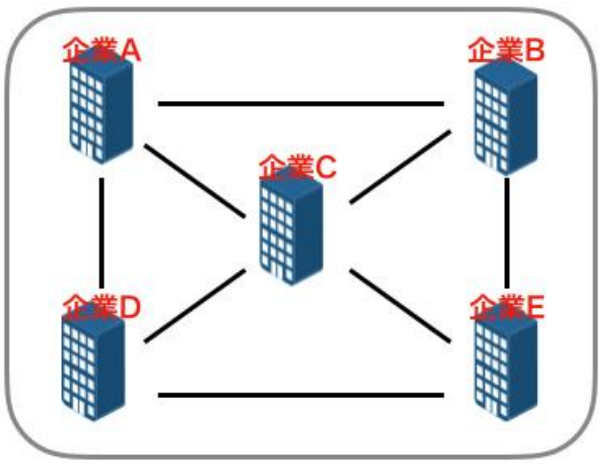


図1.3　コンソーシアムブロックチェーンのイメージ

ブロックチェーンのトリレンマ

　ブロックチェーンにはパブリックブロックチェーンやプライベートブロックチェーン、コンソーシアムブロックチェーンなどの種類があることを説明しました。そ

れぞれのブロックチェーンには特徴があり、一概に「ブロックチェーンにはこのような特徴がある」と言うことはできません。その理由についてブロックチェーンのトリレンマという概念を使って説明します。

トリレンマとは3つの要素が存在し、その中の2つの要素を重視すると、残りの1つの要素が犠牲になることを言います。トリレンマの例としては製品開発の際の「時間」「コスト」「品質」が挙げられます。開発時間を短く、コストを小さく抑えようとすると、品質が低下してしまい、品質を高い水準で満たしながら、開発時間を短縮すると、コストがかかるようになることなどを考えることができます。

ブロックチェーンのトリレンマとは、Ethereumの開発者であるヴィタリック・ブテリンにより提唱された概念であり、以下の3つを同時に満たすことはできないと考えられています。

- Security(安全性)
- Scalability(処理能力)
- Decentralization(分散性)

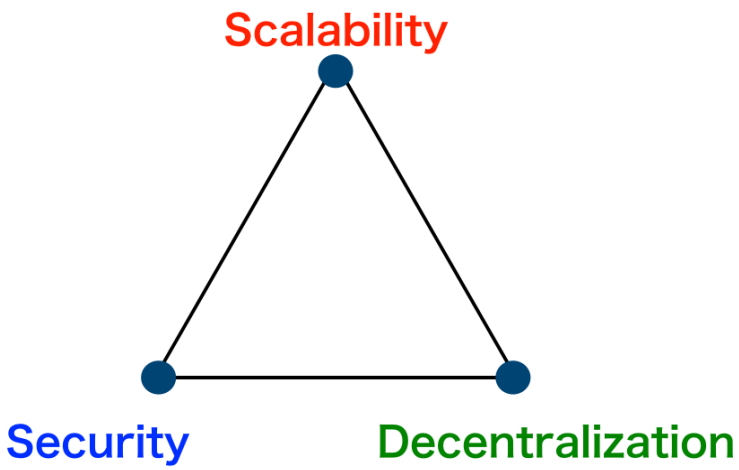


図1.4　ブロックチェーンのトリレンマ

ブロックチェーンのトリレンマを元にパブリックブロックチェーンとパーミッションドブロックチェーンの特徴について考えます。

この2種類のブロックチェーンは仕様上、**分散性に大きな違い**があります。パブリックブロックチェーンへは誰でも参加することができるため、分散性が極めて高い状態になっています。これに比べてパーミッションドブロックチェーンは参加者が限定されているため、分散性が高いとは言えません。なお、基本的にはどちらのブロックチェーンも安全性はある程度満たされているとします。

つまり、パブリックブロックチェーンでは分散性と安全性を維持する代わりに処理能力が低く、パーミッションドブロックチェーンでは安全性と処理能力を高い水準で維持する代わりに分散性が低くなっています。

このように、2つのブロックチェーンには分散性だけでなく、それ以外の部分にも大きな違いがあります。この性質の違いについて知った上で、ブロックチェーンの特徴について考える必要があります。

このテキストでは、パブリックブロックチェーンに焦点を当ててブロックチェーンとスマートコントラクトについて説明をします。理由は2点あります。

1つ目に初めて実用化されたブロックチェーンであるBitcoinはパブリックブロックチェーンであり、**ブロックチェーンの本来の特徴はパブリックブロックチェーンに出てくる**と考えるためです。

2つ目に現在スマートコントラクトの開発に頻繁に利用されるブロックチェーンはEthereumであり、このEthereumがパブリックブロックチェーンであるためです。

1.1.2 ブロックチェーンの特徴

ここからはパブリックブロックチェーンの特徴を3つ取り上げて説明します。

非中央集権

パブリックブロックチェーンは特定の企業や団体が管理しているのではなく、ブロックチェーンの参加者により**自律的にシステムが維持管理**されています。このよ

うに特別な権限を持った管理者が存在しない状態を**非中央集権的な状態**と呼びます。

これに対して、管理者が存在する状態を**中央集権的な状態**と言います。現在の社会の仕組みや様々なサービスについて考えた時、その多くが中央集権的な仕組みで成り立っていることがわかるのではないかと思います。具体的には国や地方自治体や企業、私たちが普段利用しているWebサービスなど日常で利用しているものが該当します。

現在、ブロックチェーンが持つ非中央集権性に大きな注目が集まっていますが、これは決して**非中央集権的な仕組みが中央集権的な仕組みに比べて優れているためではありません**。

これまでは中央集権的な方法でしか実現することができなかったシステムに対して、非中央集権という**新しい選択肢**が生まれたことに対して多くの注目が集まっています。どちらが優れ、どちらが劣っているということではありません。今後は中央集権と非中央集権を使い分けることで、より良いシステムを作ることができると期待されています。

高い改ざん耐性

ブロックチェーンは既存のデータベースシステムに比べ、極めて高い改ざん耐性を持っています。この特徴は主にブロックチェーンの2つの特性によって生まれています。

1つ目はブロックチェーンの**独自のデータ構造**によるものです。ブロックチェーンでは複数の取引がまとめられ、**ブロック**というデータが作られます。

ブロックを作る際には、取引をまとめるだけではなく、大量の計算コストが必要になる作業を行わなければなりません。
それぞれのブロックは大量の計算資源が投じられて作られています。
不正を目的として、ブロックの取引記録を書き換えた際には、そのブロックを再び正しいものとして、作るためには再度大量の計算資源を投じる必要があります。

それぞれのブロックは自身の前に作られた親のブロックの情報を保有しており、１つのブロックに対して書き換えを行うと、そのあとに続く全てのブロックで再度計算が必要になります。
このように計算資源にかかるコストにより、ブロックチェーンの高い改ざん性は実現されています。

2つ目に**大量のノードが取引記録を保存**していることが挙げられます。具体例を挙げるとパブリックブロックチェーンであるBitcoinでは世界中の1万近くのコンピュータでこれまでに行われた全ての取引の記録を保存しています。更に重要な点として、これらのコンピュータが特定の管理者によって運営されているのではなく、**独立した参加者によって管理**されているということです。これによりまとまった書き換えや紛失などが発生する可能性が低くなります。

高い可用性

ブロックチェーンの中でも特にパブリックブロックチェーンは可用性が高いことが特徴です。具体例を挙げるとBitcoinは2009年から運営が始まり、2020年の1月までの10年以上一度も停止していません。これは、ブロックチェーンのネットワークが**独立した多数のコンピュータにより構成**されていることと、ブロックチェーンのネットワークには、**誰でも簡単に参加することができる**点が挙げられます。

ここでブロックチェーンの仕組みを完全に停止させることを考えます。停止させる方法としてはブロックチェーンのネットワークを構成するコンピュータを1台ずつ停止させる必要があります。1台1台停止させ、ブロックチェーンのネットワークを構成する全てのコンピュータを停止させたとき、ブロックチェーンは完全に停止します。

しかし、ブロックチェーンを構成するノードはソフトウェアを持っていれば誰でも新たなノードを立ち上げることができます。そのため、コンピュータを停止させている間に次々と新たなノードが立ち上がります。これは植物に例えられることがあり、世界中に自生する特定の植物を根絶やしにするためには、世界中を回って1つ1つ抜く必要があります。この間にも世界のどこかでは新しい芽が生え始め、根絶することは極めて難しいと言えます。

このような仕組みでブロックチェーンの高い可用性は実現されています。

1.2 ブロックチェーンの構成要素

ここからはブロックチェーンの仕組みについて理解する上で重要な「トランザクション」、「ブロック」、「ブロックチェーン」などの構成要素と、ブロックチェーンの処理の流れについて説明します。

1.2.1 トランザクション

トランザクションとは、ブロックチェーンネットワークで行う**処理が記述されたデータ**です。具体的には送金処理や、プログラム実行の要求などが記述されています。このトランザクションが**ブロックチェーン上で行われる処理の最小単位**になります。

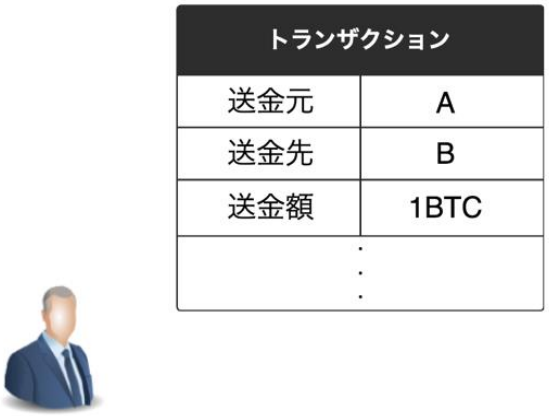


図1.5　トランザクションの作成

ブロックチェーン上で処理を行ってもらうためには、トランザクションを作成し、それをブロックチェーンネットワークを構成するノードに伝達します。トランザクションを受け取ったノードは、更に隣のノードへトランザクションを伝達します。これが繰り返されることで**トランザクションはブロックチェーンネットワーク全体に伝達されます**。

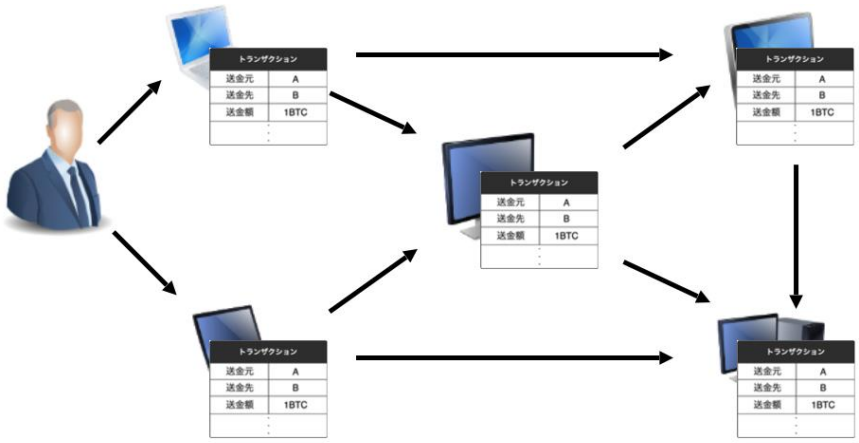


図1.6　トランザクションの伝達

それぞれのノードは受け取ったトランザクションを隣のノードに伝達する前に、トランザクションの**検証作業**を行います。これは、ブロックチェーンではトランザクションは誰でも発行することができるため、全てが正しいものであると断定はできず、不正なトランザクションが含まれる可能性があるためです。

それぞれのノードは自律的にトランザクションの検証を行い、正しいトランザクションであると判断すると、隣のノードへ伝達し、なおかつ自身の手元にも保存します。

1.2.2 ブロック

ブロックチェーンのネットワークではトランザクションが次々に発行され、それらは検証された後に、それぞれのノードの元に蓄積されます。トランザクションはそのままの状態でそれぞれのノードが保存しておくのではなく、一定時間経過するとそれぞれのノードは手元のトランザクションをまとめて**ブロック**と呼ばれるデータのまとまりを作ります。このトランザクションをまとめてブロックを作成する作業を**マイニング**と呼び、マイニングを行うノードを**マイナー**と呼びます。

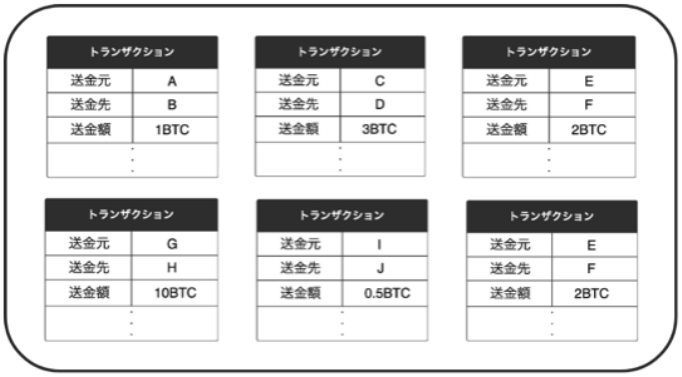


図1.7　ブロックの作成

ここで考えなければならないのが、誰がブロックを作るのかということです。基本的に1つのトランザクションは1つのブロックにしか含めることはできません。そのため、ネットワーク内で無数にブロックを作ることはできず、誰がブロックを作るのかという点が重要になります。

コンセンサスアルゴリズム

ブロックチェーンのネットワークからブロックを作成する代表者を決定するための手法を**コンセンサスアルゴリズム**と言います。ここでは初めて実用的に利用することができるようになったコンセンサスアルゴリズムである**Proof of Work**について紹介します。Proof of WorkはBitcoinやEthereumで採用されているコンセンサスアルゴリズムです。

Proof of Work

Proof of Workではブロックを作成する際に、トランザクションをまとめるだけではなく、ブロック固有のデータが記録される**ブロックヘッダーのハッシュ値が規定以下になるように、ヘッダーの構成要素の1つであるナンスの値を調整する**必要があります。ハッシュ関数の特性上入力値がわずかでも異なると、出力されるハッシュ値は大きく変わり、入力値からハッシュ値を予測することはできません。

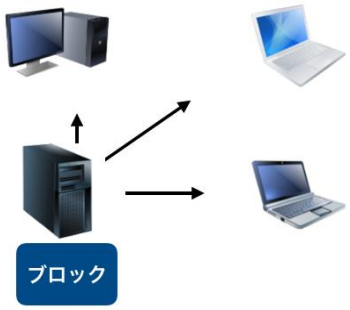
このような特性から、繰り返しナンスの値を調整することで、条件を満たすハッシュ値を探します。**最も早く条件を満たすハッシュ値を求め、最も早く作成されたブロック**を採用します。

以下にProof of Workの流れを示します。

1. マイナーがナンスの計算をする

2. あるマイナーがナンスを見つける

3. ブロックを作成し、伝達する



4. ブロックを検証し、保存する

図1.8 Proof of Workの流れ

今回はコンセンサスアルゴリズムの例としてProof of Workを紹介しましたが、これ以外にも通貨の保有額やデポジット額に応じてブロック作成の権利が与えられるProof of Stakeや、通貨の保有量、取引量、取引相手などによってマイナーに重要度という指標が与えられ、それによってブロックの作成の権利が与えられるProof of Importanceなどがあります。

更に同じProof of Workを使っているブロックチェーンでも、利用しているハッシュ関数が異なるなど、中身のアルゴリズムには様々なものがあります。

ここからはブロックの構成について説明します。ブロックを構成する項目はブロックチェーンによって異なりますが、多くのブロックチェーンのブロックは**ブロックヘッダー**と、**トランザクションリスト**により構成されています。ブロックヘッダーはブロックの固有のデータを保存している領域です。ブロックの識別子やタイムスタンプ、マイニングに関する情報などが記録されています。トランザクションリストには発行されたトランザクションがまとめられています。ブロックのデータ容量の大半をトランザクションリストが占めています。

以下のWebサイトでリアルタムに作成されるブロックの詳細を確認することができます。

- **Bitcoin** (https://www.blockchain.com/explorer)
- **Ethereum** (https://www.blockchain.com/explorer?currency=ETH)

パブリックブロックチェーンでは、ブロックチェーンの仕組みを動かし続けるためにマイニングを行うことや、ブロックチェーン上で行われた取引などのデータを保持し続けることに対する**責任は誰にもありません**。仮に自分がブロックチェーンに記録していたデータが、ブロックチェーンというシステムが止まることによって失われても、管理者が存在しない仕組みであるため、誰を責めることもできません。では、マイナーは何のために記録を保存し、マイニングを行っているのでしょうか。

ブロックチェーンではトランザクションをまとめ、ハッシュ計算を行い、ブロックを作成すると、**報酬を得る**ことができます。この報酬は新たに発行された通貨と、トランザクションを発行する際に必要となる手数料から支払われます。

また、ブロックを作るためには、正しいトランザクションを集める必要があり、そのためにはトランザクションの検証作業が必要になります。検証作業を行うためには、これまでにブロックチェーン上で行われた取引の履歴を保存し、新たな取引がそれと矛盾していないか確認する必要があります。そのため**過去の取引履歴を保存しておくことが必要**になります。このような報酬の仕組みにより、ブロックチェーンは稼働し続けることができます。

1.2.3 ブロックチェーン

マイナーによりブロックが作成されると、トランザクションが作成された時と同様に、**ネットワーク全体にブロックが伝達**されます。それぞれのマイナーは手元にブロックが届くと、そのブロックが正当なものであるかの検証作業を行います。検証の結果正しいブロックであると判断した際には、更に隣のノードへ伝達を行い、自身の手元にもブロックを保存します。これが繰り返されることで、ブロックはネットワーク内の全ての参加者に届きます。

Bitcoinでのブロックの検証内容の例について紹介します。

- ブロックの構造は正しいか
- ブロックに含まれているトランザクションは全て正しいものか
- ブロックのサイズは適切か

これ以外にも数十個の検証項目があり、全ての検証に通貨すると正しいブロックであると判断されます。

それぞれのブロックはそのブロックの**親となるブロックのハッシュ値をヘッダーに保有**しています。そのため、手元に届いたブロックは親ブロックに繋がり、それが連鎖することで、**ブロックがチェーン構造になり、ブロックチェーンが形成**されます。これが狭義のブロックチェーンであり、ブロックチェーンという名前はここから付けられています。

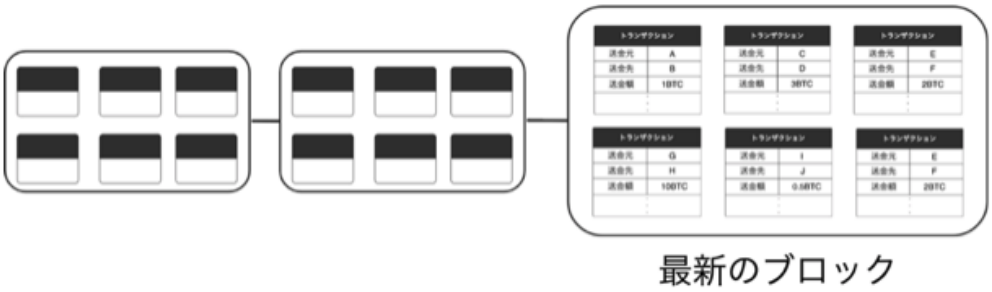


図1.9　ブロックチェーンの構造

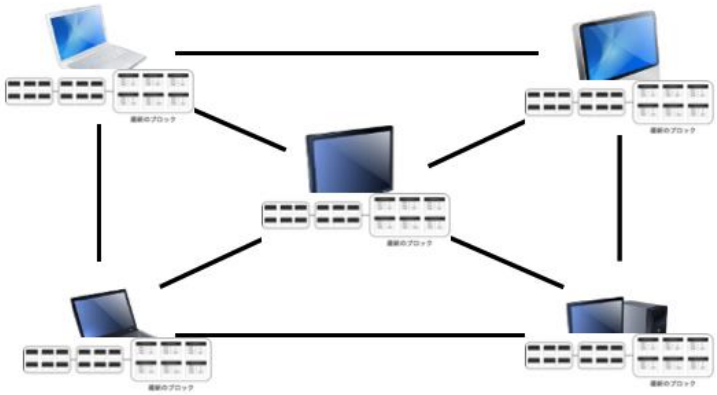


図1.10 ブロックの共有

以上により、ネットワーク内の全てのノードは、同じブロックチェーンを保有した状態になります。同じブロックチェーンを保有しているということは、同じブロック、同じトランザクションを保有していると言い換えることができます。つまり、**ネットワークの参加者全体で同じ記録が共有された状態**が出来上がります。

Nakamotoコンセンサス

ブロックチェーンは基本的に1つのブロックに1つのブロックが繋がることでチェーン構造になります。しかし、稀に1つのブロックに対して、複数のブロックが紐付きブロックチェーンが分岐することがあります。ブロックチェーンは分岐が発生し、それぞれにブロックが繋がり伸びていく状態となると、システムの安全性が低下してしまいます。そのため、分岐したブロックチェーンがそれぞれ伸びるのでは

なく、分岐が発生した際にも、1つのブロックチェーンのみが伸び、他方はそれ以上伸びなくなる仕組み作りがされています。

この仕組みはブロックチェーンにより異なりますが、今回はBitcoinで採用されているNakamotoコンセンサスという方法について説明します。

Nakamotoコンセンサスでは最も長く伸びているチェーンを採用する方式を採っています。これはブロックを1つ作る際には大量の計算資源、つまりコストが投じられることから、そのブロックには信頼があると考え、最も長いブロックチェーンには、より多くの信頼があると考えられることからこのような方式が取られています。

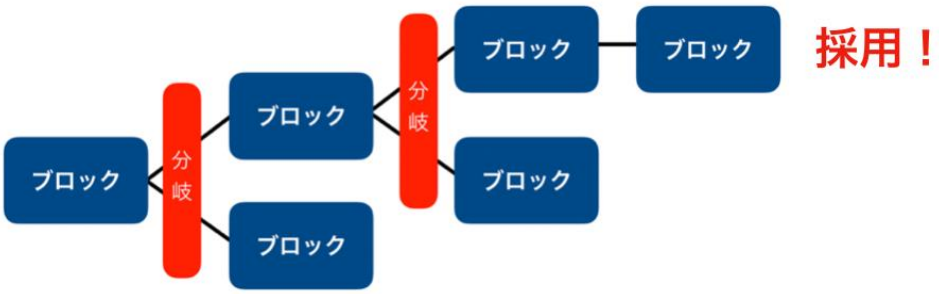


図1.11 Nakamotoコンセンサスによるチェーンの選択

2. スマートコントラクト

この章ではブロックチェーンの活用例の1つであるスマートコントラクトについて説明します。

2.1 ブロックチェーンとスマートコントラクト

2.1.1 ブロックチェーンの利用

まずはスマートコントラクトの概要と特徴について説明します。一言にスマート「コントラクト」と言ってもその解釈は様々あり、例として以下が挙げられます。

- 賢い契約
- 契約の自動化
- プログラム化された契約
- 自動執行権のある契約
- スマートコントラクト・プラットフォーム上で動く契約

紹介したスマートコントラクトの解釈からわかるようにスマートコントラクトには厳密な定義はありません。これらの共通点としてはコンピュータによって自動的に取引が行われる点です。そこで、ここでは人が介在しない全ての商取引をスマートコントラクトとします。

スマートコントラクトは決して特別な取引方法ではなく、私たちが普段から利用している馴染みのある技術です。具体的な例として、自動販売機、オンライン決済、自動改札などの身近なサービスを挙げることができます。つまり、スマートコントラクトはブロックチェーンにより作られた新しい技術ではなく、ブロックチェーンが誕生するより前から使われている技術です。決してブロックチェーンによって初めて成り立った技術ではありません。

従来のスマートコントラクトでは、契約の履行処理を自動的に行うために**サービスを提供する企業がサーバを準備**し、その中で契約処理が行われてきました。これは、私たちが現在利用しているサービスを思い浮かべるとわかるのではないかと思います。

契約の履行処理が行われるサーバでは取引内容が正しいかどうかの**検証作業**も行います。取引の内容を確認することで、不正な取引が行われることを防ぎ、利用者は安全な取引を行うことができます。これには、サーバでこれまでに行われた取引記録を保存しておく必要があります。

このような点からわかるように、従来のスマートコントラクトは管理者が存在することによって成り立つ技術です。しかし、この仕組みにはいくつか問題があります。

1つ目にスマートコントラクトを提供する企業が倒産することやサービスの終了により、**取引を管理していたサーバが停止**し、一切の取引を行うことができなくなる点を挙げることができます。当然、サーバが機能しなくなるので、これまで行なった取引の記録も失われてしまいます。

2つ目に**管理者が不正**を行う点があります。企業が管理を行っているサーバでは過去の記録や検証作業において不正を行うことができてしまいます。この点も管理者がいるからこそ発生する問題であるということができます。

ここで挙げた問題はブロックチェーンの管理者を必要とせずに自律的に動くことができるという特徴を利用することで解決することができます。

2.1.2 スマートコントラクトの仕組み

ここでは、スマートコントラクトの処理の流れについて説明します。以下に従来のスマートコントラクトの構成を示し、それぞれの構成要素について説明します。

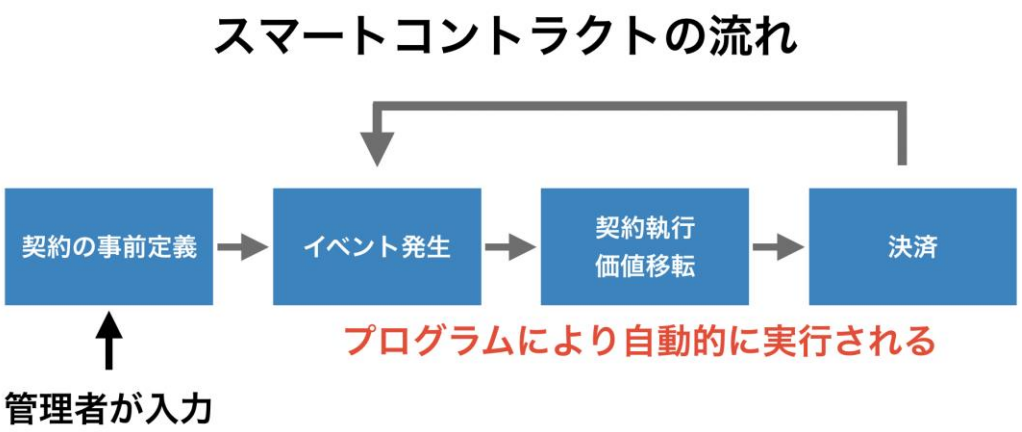


図2.1　スマートコントラクトの処理の流れ

契約の事前定義

スマートコントラクトにより行われる契約の内容を決定し、それをプログラムします。作成したスマートコントラクトはコンピュータ上で実行可能な状態にします。

イベントの待機

実行可能な状態になったスマートコントラクトは、発動のトリガーとなるイベントを待ちます。

契約実行　価値移転

あらかじめ定められたイベントが発動すると、定められたプログラムに従って契約を行うための処理が実行されます。

決済

契約内容に基づいて価値の移転が行われます。

次に、ブロックチェーンを用いたスマートコントラクトについて説明します。処理の流れを以下の図に示しました。

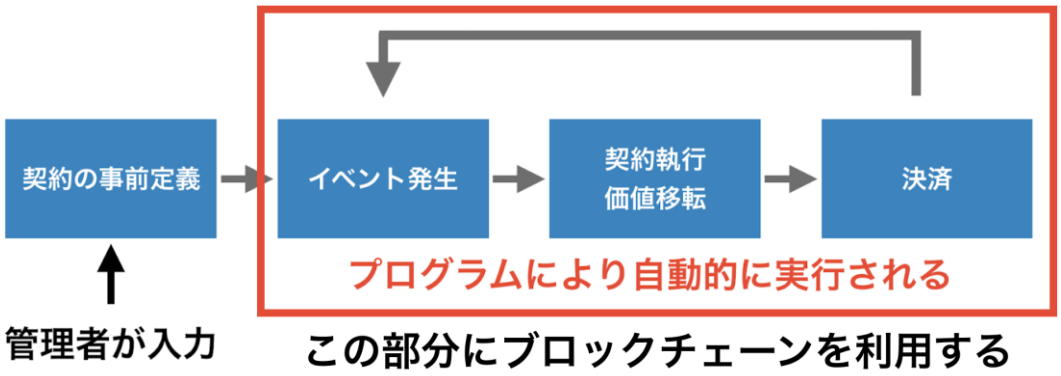


図2.2 ブロックチェーンを用いたスマートコントラクトの処理の流れ

これまでは、サービスを提供する企業が準備したサーバで行われていた処理をブロックチェーンにより行います。これによって、管理者を必要としないスマートコントラクトを実現することができました。

2.2 スマートコントラクトの特徴

ここからは、ブロックチェーンを用いたスマートコントラクトと、従来のスマートコントラクトを比較しながら、スマートコントラクトにブロックチェーンを用いることの利点について説明します。

2.2.1 利点

管理者不在

スマートコントラクトにブロックチェーンを用いることで、ブロックチェーンの特徴の1つである**管理者を必要としない**という特徴を活かしたスマートコントラクトを作成することができます。従来のスマートコントラクトでは管理者は取引の信頼点として存在していましたが、ブロックチェーンを用いることで、ブロックチェーンのプロトコルにより信頼点が必要なくなりました。このため、従来のスマートコントラクトのように管理者によってシステムが止められてしまうことや、管理者の判断で取引を行うことができなくなることはありません。

また、管理者が必要なくなったことから仲介により発生する手数料が必要なくなります。これによって、利用者は**より安い手数料**で取引を行うことができます。ここで注意しなければならない点として、仲介者が必要なくなり、仲介手数料は必要なくなりましたが、無料でブロックチェーンを用いたスマートコントラクトを利用することができる訳ではありません。これは、Ethereumをはじめとしたパブリックブロックチェーンではトランザクションを発行する際にトランザクションに手数料を付加しなければならないためです。

透明性の高い取引を行うことができる

パブリックブロックチェーンはその特性上、記録を世界中のノードで共有することで、それぞれが自律的に新たな取引を検証することができます。これはパブリックブロックチェーンに書かれた内容は**世界中の誰でも見ることができる**と言い換えることもできます。ブロックチェーンの中身を参照することのできるブロックチェ

ーンエクスプローラーもBitcoinやEthereumの記録が世界中に公開されているからこそ存在するサービスです。

スマートコントラクトにおいてブロックチェーンの、この特性を利用することで、**透明性の高い取引**を行うことができます。具体的にブロックチェーンに書き込まれる内容はスマートコントラクトを実行するために必要になるコードや、取引の結果です。パブリックブロックチェーンで取引を行なったのであれば、これらは世界中に公開され、透明性の高い取引を行うことができます。

しかし、これには難点もあります。様々な取引を行なったときに当然その中には世間に公開したくないものもあります。パブリックブロックチェーンを用いると、全ての取引が公開されてしまうため非公開で行いたい取引には向いていません。

改ざん耐性が高い

スマートコントラクトにブロックチェーンを用いることで、ブロックチェーンの特徴の1つである**改ざん耐性の高さ**を利用することができます。スマートコントラクトの実行に必要なコードや、取引の実行記録などを改ざんされることなく保存し続けることができます。これにより、**契約プログラムの改ざんによる不正**な取引や、**契約結果の書き換え**などが発生するリスクを大きく低下させることができます。

2.2.2 課題点

スケーラビリティ問題

BitcoinやEthereumをはじめとしたパブリックブロックチェーンでは**一定時間に処理することのトランザクション数に制限**があります。具体的にはBitcoinでは1秒間に最大7件、Ethereumでは最大15件のトランザクションしか処理することができません。これに対して、大手のクレジットカード会社では1秒間に数千件の処理を行うことができます。これと比較すると、いかにBitcoinやEthereumが一定時間に処理することのできるトランザクションの数が少ないかわかると思います。

パブリックブロックチェーンの処理能力はブロックチェーンネットワーク内のノードの数を増加させることや、より処理能力の高いノードを立てることで解決する問題ではなく、ブロックチェーンの**仕様が作成された時点である程度定まっていま**

す。このようにブロックチェーンの処理能力は一般利用を考えると、極めて低く、更に簡単に処理能力を高めることも難しい問題をスケーラビリティ問題と呼びます。

ブロックチェーンのスケーラビリティ問題はブロックの生成間隔と、ブロックのデータサイズがあらかじめ仕様として定められていることが原因として発生します。1つのブロックのデータサイズが定まると、当然そのブロックに含めることのできるトランザクションの数にも制約が発生します。これに加えてブロックの時間的な生成間隔も定められると、一定時間に処理することのできるトランザクション数は自然に決まります。

この問題はブロックのサイズを大きくすることや、ブロックの生成間隔を短くすることで簡単に解決することができるように思えます。しかし、このような仕様の変更は、ブロックチェーンのセキュリティの低下に直結します。そのため、このような仕様の変更によるスケーラビリティ問題の解決はBitcoinやEthereumでは行われることはないと考えられます。

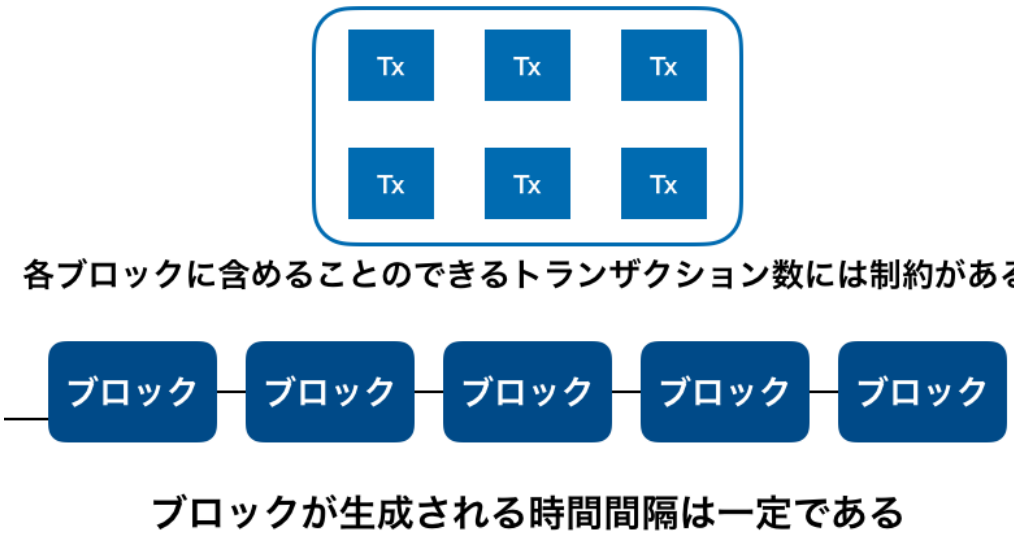


図2.3 ブロックチェーンのスケーラビリティ問題

鍵の管理の問題

スマートコントラクトを利用するためには、利用するブロックチェーンでアカウントを作成しなければなりません。アカウントとはブロックチェーン上で仮想通貨

を管理するための口座のような役割を果たすものです。それぞれのアカウントには、固有の秘密鍵が紐付き、そのアカウントからトランザクションを発行する際には、この**秘密鍵で署名を行う必要**があります。この鍵の管理が一般の利用者にとって大きな課題となります。

銀行口座のパスワードは比較的桁数が少なく、覚えている人も多いのではないかと思います。また、仮にパスワードを忘れた際にも、銀行で口座を開設した際の個人情報などからパスワードを再発行してもらうことが可能になります。

これに対して、ブロックチェーンのアカウントに紐づく秘密鍵はアルファベットと数字が混じった文字列になっており、更に桁数も多く銀行のパスワードなどに比べると覚えることは現実的ではありません。

アカウントを作る際には銀行のような機関で、作ってもらうのではなく、専用のアプリを利用することで個人でも簡単にアカウントを作ることができます。つまり、ブロックチェーンの**アカウントは一切の個人情報に結びついていません**。

また、ブロックチェーンは管理者が存在しない分散自律型の仕組みであることから、鍵を紛失した際に**鍵を再発行してくれる機関は存在しません**。つまり、アカウントにどれだけ大量の通貨を保有していたとしても、秘密鍵を紛失してしまうと、その通貨は二度と使うことができなくなります。

秘密鍵の管理の方法には、様々な方法がありますが、このようなアカウントの特徴を理解した上で、目的に応じた鍵の管理が必要になります。この特徴をスマートコントラクトを利用する一般のユーザーに理解してもらった上で、鍵を管理してもらうことが重要になります。

手数料の問題

ブロックチェーンを利用したスマートコントラクトを利用するためにはトランザクションを発行する必要があります。この際にパブリックブロックチェーンでは**トランザクション手数料が必要**になります。このトランザクション手数料はそれぞれのブロックチェーンの**内部通貨で支払う必要**があります。つまり、Bitcoinを用いたスマートコントラクトであればbitcoinを、Ethereumを用いたブロックチェーンであればEtherによって手数料を支払わなければなりません。

ここで考えなければならないのが一般ユーザーの仮想通貨の保有率です。仮に、今日ブロックチェーンを用いた素晴らしいサービスが発表されたとします。しかし、多くの人は仮想通貨を保有していないためこのサービスを利用することができません。このように一般のユーザーが仮想通貨を保有していないため、ブロックチェーンを用いたサービスを利用することができず、普及させる際のボトルネックになります。

この問題の解決策はいくつか提案され、実現されつつあります。方法としてはブロックチェーンを用いたサービスを提供する企業が**仮想通貨の支払いを肩代わり**します。ここでは、2つの方法を紹介します。

1つ目にそれぞれの**ユーザーに紐づいた秘密鍵を企業が管理**する方法です。企業がユーザーの代わりに通貨の保有やトランザクションの発行を行うことで、ユーザーが直接仮想通貨を保有する必要がなくなります。トランザクションの発行にかかった手数料は法定通貨に換算して、企業がユーザーに請求することができます。この方法ではユーザーが仮想通貨の購入や、鍵の管理などを一切行う必要がありません。その反面、特定の企業が大量の秘密鍵を管理することになるため、仮想通貨取引所からの鍵の流出による通貨の盗難事件と同じような事件が発生するリスクについて考える必要があります。

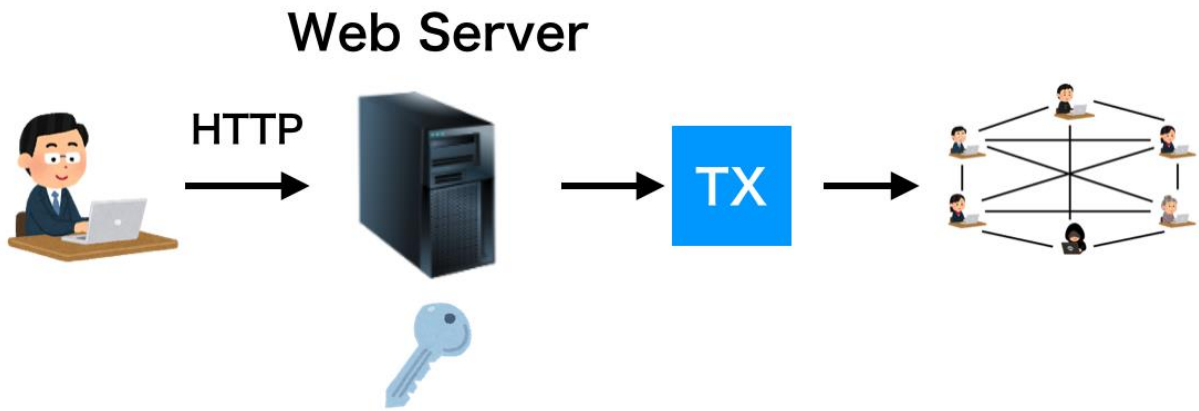


図2.4 鍵の管理を企業が行う仕組み

2つ目にEthereumで実現される**メタトランザクション**という仕組みがあります。メタトランザクションでは、ユーザーがEthereumのアカウントを作成し、処理に応じて必要なトランザクションを作成します。通常であれば、このトランザクションに

手数料をつけて、Ethereumのネットワークに伝達します。しかし、メタトランザクションの仕組みを利用する場合には、作成したトランザクションを**Ethereumネットワーク外部のサービスの提供者に対して送信**します。ここでサービスの提供者は、ユーザーから送信されたトランザクションを内部に含んだトランザクションを発行します。このトランザクションをEthereumのネットワークに伝達します。これによりマイナーにより内部のトランザクションが取り出され、ネットワークに伝達されることで、ユーザーが元々発行したトランザクションが実行されます。このとき、トランザクション手数料を支払う必要があるのは、トランザクションをEthereumネットワークに投げ込んだサービスの提供者のみになります。そのため、ユーザーは仮想通貨を保有することなく、鍵を自身で管理しながらトランザクションの署名を行い、ブロックチェーンを利用したサービスを利用することができます。サービスを提供した企業が代払いした手数料に関しては、企業がユーザーに対して請求することができます。

1つ目に紹介した代払いの仕組みとの最大の違いは、ユーザー自身が鍵を管理する点です。

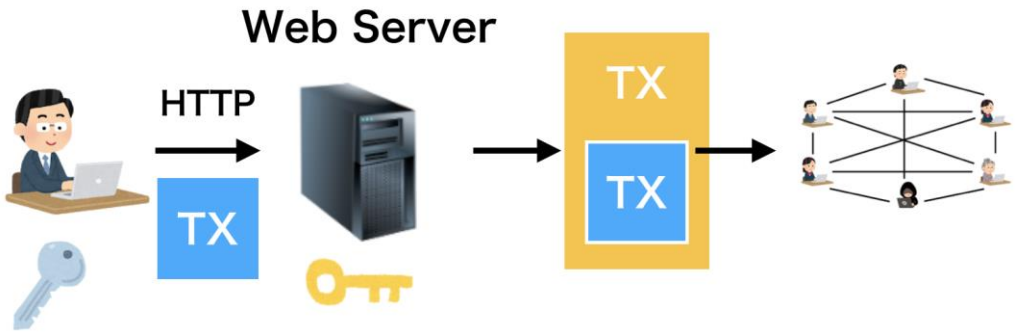


図2.4 メタトランザクションの仕組み

処理速度の問題

ブロックチェーンは従来のデータベースシステムに比べて改ざん耐性が高い特徴がありますが、これには条件があります。

ブロックチェーン上で取引を行う際には、トランザクションが発行されます。このトランザクションはマイナーによって処理され、ブロックに取り込まれます。トランザクションがブロックに取り込まれた段階で、このトランザクションに対して

行われる処理は全て終わりますが、これで安全に取引が完結した訳ではありません。

トランザクションは発行した直後は、トランザクションがマイナーの手元に保存されているものの、ブロックには取り込まれていない状態になります。これを承認0の状態と言います。この状態から時間の経過とともにトランザクションがブロックに取り込まれると、承認1の状態となります。更に時間の経過に伴って、トランザクションが取り込まれたブロックの後ろに更にブロックが繋がると承認2に、更にブロックが繋がると承認3となり、時間の経過に連れて承認数は増えていきます。**承認数が増加するに従って、取引の安全性が上昇**し、取引が消されてしまうことや、内容が書き換えられてしまう可能性は確率的に0に近似されていきます。

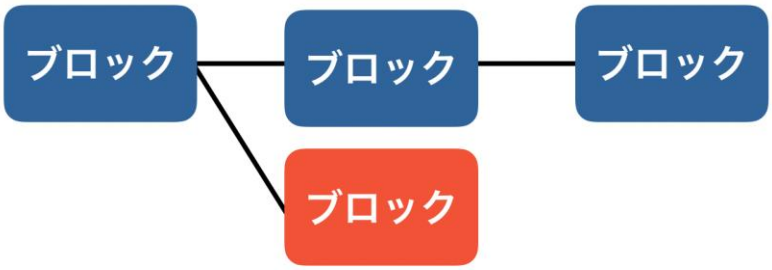
このようにブロックチェーンでは取引が行われてからの経過時間により取引の安全性が高まる仕組みになっており、取引を行なった直後は決して安全であるということはできません。そのため、**即時性が重視される取引にはブロックチェーンは向いていません**。

このような特徴は、ブロックチェーンが直鎖上のブロックに含まれる取引のみを参照する仕組みや、分岐が発生した際のブロックチェーンの選択などが関係します。ここではBitcoinに用いられている最も長いブロックチェーンを採用するNakamotoコンセンサスを例に説明します。

ブロックチェーンに対する攻撃の手法として、**51%攻撃**と呼ばれるものがあります。この攻撃はブロックチェーンに意図的に分岐を発生させ、分岐させた後のブロックチェーンに大量の計算資源を投じることで、メインのチェーンのよりも長いチェーンを作ることでチェーンの切り替えを行う攻撃です。このような攻撃を防ぐために、承認数が重要になります。

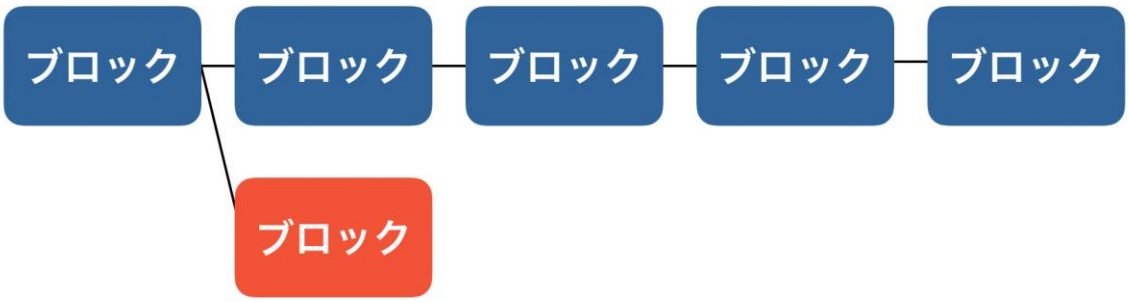
承認数が多いブロック、つまりブロックチェーンの先頭からより遠い場所にあるブロックから分岐を発生させ、先頭のブロックに追いつこうとすると、攻撃者が作成しなければならないブロックの数が多く、分岐したブロックチェーンがメインのブロックチェーンに追いつくことは確率的に難しくなります。承認数が少ないブロック、つまり先頭のブロックから近いブロックでは、メインのブロックに追いつくためのブロック数は比較的少なく、分岐したブロックチェーンが追いつく確率は比較的高くなります。

このような理由からブロックチェーン上で取引を行う際には、取引が行われてからの**経過時間が重要**になり、即時性と安全性を両立させる必要のある取引には向いていません。



最新のブロックから近い場所での分岐の際にはメインのチェーンの追いつくためのブロック数は少なくてよい

図2.5　浅い場所からのブロックチェーンの切り替わり



最新のブロックから遠い場所で分岐が発生するとメインのチェーンに追いつくためのブロック数が多く、追いつくことが難しい

図2.6　深い場所からのブロックチェーンの切り替わり

プログラムの修正

ブロックチェーンの特徴の1つとして**一度ブロックチェーンに記録された内容は削除することや、変更することが基本的にはできない**点が挙げられます。この点は大

きな利点とも捉えることができ、また場合によっては難点として捉えることもできます。

ブロックチェーンに記録されたプログラムは誰にも内容を書き換えることはできません。つまり、サイバー攻撃などによりコードが改変されることや、消されてしまうことはありません。この点は正しいプログラムをブロックチェーンにデプロイした場合には、大きな利点として働きます。しかし、作成したコードにバグがあった場合には、この特徴が大きな問題になります。仮にバグにより、自身の通貨が他者によって盗まれるなどの問題があった場合にも、このプログラムを改変することや、消してしまうことはできません。

このような場合に備えて、あらかじめプログラムに特定の関数を停止する機能や、プログラム自体を利用できないようにする機能を持たせておく必要があります。また、プログラムをアップデートする際にも事前に工夫が必要になります。

このように、これまでの開発と異なり、ブロックチェーンに関する知識をしっかりと持った上で開発を行うことが重要になります。

2.3 プラットフォームとなるブロックチェーン

スマートコントラクトは全てのブロックチェーン上で実行できる訳ではなく、あらかじめスマートコントラクトのプラットフォームとして開発されたブロックチェーンの上で実行することができます。ここでは、スマートコントラクトのプラットフォームとなる3つのブロックチェーンについて紹介します。

2.3.1 Ethereum

2015年にリリースされたブロックチェーンであり、**管理者を必要としないスマートコントラクトを実現**するために生まれたブロックチェーンです。Ethereumには内部通貨としてEtherが存在し、スマートコントラクトの利用料を支払うために利用されます。Bitcoinと同様にEtherは通貨としての利用も可能です。

現在、**スマートコントラクトやDAppsを作成する際の最もメジャーなブロックチェーン**であり、日本語の情報も多いことから、このテキストではEthereumを用いて、スマートコントラクトの開発を行います。Ethereumの詳細については次の章で確認します。

2.3.2 NEO

NEOは中国で開発されたブロックチェーンであり、スマートコントラクト作成のためのプラットフォームです。 プロジェクトが始まった当時はAntSharesと呼ばれていましたが、2017年に現在のNEOに改名されました。多くのパブリックブロックチェーンと同様にNEOのプラットフォームにも内部通貨が存在し、プロジェクトと同名のNEOと名付けられています。

NEOではPython、Java、C#など開発者が既に使い慣れた様々な言語でコントラクトを書くことができます。今後更にサポートする言語を増加させることが計画されています。

2.3.3 EOS

EOSはオープンソースのスマートコントラクトのプラットフォームです。EOSの最大の特徴は0.5秒という高速なブロック生成間隔と、DPoS(Delegated Proof of Stake)と呼ばれるコンセンサスアルゴリズムによる高速なトランザクション処理です。DPoSはブロック生成の権限を信頼するノードに委任するProof of Stakeアルゴリズムです。少数かつ高性能なノードのみで、ブロック生成を担うことで、パフォーマンスを向上させています。またスマートコントラクトを実行するためのトランザクションの発行時に、利用者が手数料を支払うのではなく、開発者が手数料を支払うことも特徴です。EOSのコントラクトはC言語もしくはC++言語で実装することができます。

3. Ethereum

ここからはスマートコントラクトを作成するためのプラットフォームであるEthereumについて、その概要と仕組みについて説明します。

3.1 Ethereumの概要

3.1.1 歴史

Ethereumの構想は2013年に当時大学生であったヴィタリック・ブテリンにより示されました。この構想を元に2014年にProof of Conceptの最初のフェーズとして、C++で実装されたEthereumのクライアントソフトがリリースされました。

以下に、時系列に沿ったEthereumに関する出来事について紹介します。

2013年11月　ホワイトペーパーが発表

ヴィタリックが作成したホワイトペーパーで、Ethereumプロジェクトの内容や技術について説明され、Ethereumの目的はブロックチェーンを利用した分散型アプリケーションの開発を行うためのプラットフォームを構築することであると発表されました。

2015年5月　テスト環境へリリース

Ethereumが、テスト環境であるRopstenにリリースされました。このテスト環境では内部通貨であるEtherの取引はできたものの、まだマイニングを行うことはできませんでした。

2016年6月　THE DAO事件

Ethereumを用いたTheDAOというプロジェクト内のコードのバグにより、360万ETHが流出する事件が起こりました。この事件はEthereumのブロックチェーンを攻撃が行われるよりも前に巻き戻すことによって、流出した通貨を取り戻し、解決されました。しかし、この事件はEthereum上の1つのプロジェクトのためにEthereum自体に

変更を加えたことに対して批判があり、大きな問題となりました。ここで重要なのが、Ethereum自体にバグがあった訳ではなく、Ethereum上のコントラクトにバグがあったという点です。

2019年3月、コンスタンティノープルの実行

Ethereumでは度々仕様の変更を行うためのアップデートが行われています。アップデートには2種類あり、1つは後方互換性のあるアップデートであり、もう1つは後方互換性のないアップデートです。ブロックチェーンにおいて後者のアップデートのことをハードフォークと呼びます。Ethereumではプロジェクトが立ち上がった当時から4つの大きなハードフォーク が計画されており、これらにはそれぞれ名前がつけられ、時系列順に以下のように呼ばれています。

- Frontier
- Homestead
- Metropolis
- Serenity

3つ目のハードフォークであるMetropolisは複数の段階に分けられて実行されており、上記のコンスタンティノープルはMetropolisの中の段階の１つです。

最後のハードフォークである、SerenityでEthereumのアップデートが全て完了する訳ではなく、Serinityが行われた後にはEthereum2.0というフェーズに入り、更にアップデートは続きます。

Ethereumでは現在、コンセンサスアルゴリズムにBitcoinと同様にProof of Workが利用されています。しかし、SerinityアップデートでProof of Stakeに移行される予定となっています。

3.1.2 WorldComputer

Ethereumでは参加者によって構成されるネットワーク全体で**仮想的なコンピュータ**が構築され、このコンピュータは**EVM(Ethereum Virtual Machine)**と呼ばれます。

また、Ethereumはパブリックブロックチェーンであり、世界中のコンピュータによって１つの仮想マシンが構築されている点に着目してWorldComputerとも呼ばれます。Ethereum上にプログラムを乗せ、プログラムを利用するだけ、つまりEthereumを利用するだけであれば、Ethereumは既存のクライアント・サーバー型ネットワークのサーバの様に振る舞い、プログラムの実行結果はブロックチェーン記録されます。以下にEVMのイメージ図を示します。

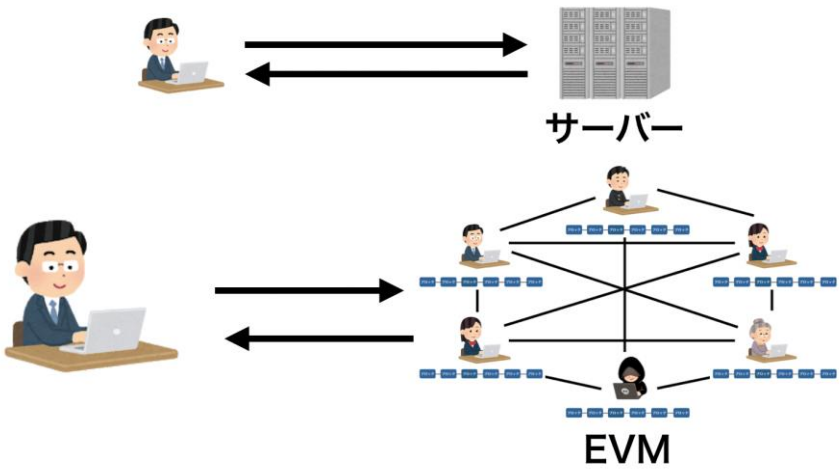


図3.1　EVMのイメージ

次に、Ethereumにおける「コントラクト」という言葉について説明します。Contract(コントラクト)という英単語には、「契約」や「契約する」などの意味があります。しかし、Ethereumにおける「コントラクト」は少し意味が異なるため注意が必要になります。Ethereumでは価値や権利が絡む取引、つまり一般的に契約と呼ばれるもののみをコントラクトと呼ぶのではなく、1つのプログラムのまとまりをコントラクトと呼びます。つまり、数字と数字を足す処理、登録した文字列を参照、変更する処理などの仮想通貨の取引や権利などには関連しない処理を動かすプログラムもコントラクトと呼ばれます。

3.1.3 Bitcoinとの相違点

ここからは共にパブリックブロックチェーンである、BitcoinとEthereumの相違点について確認していきます。

まず、これらのブロックチェーンが作られた目的と特徴について説明します。

Bitcoinは**誰にも止められることなく当事者同士で通貨を取引**することを目的に作られたブロックチェーンです。つまり**価値の移転に特化したブロックチェーン**であると言うことができます。また、Bitcoinはその特徴から金(きん)に例えられ、**DegitalGold**とも呼ばれます。ブロックを作成し、新たな通貨を発行することがマイニング(採掘する)と呼ばれるのは、このような背景があります。

これに対して、Ethereumは特定の目的に特化したブロックチェーンではなく、**コンピュータのように何でもできるブロックチェーン**の実現のために作られました。そのため、先程も説明した通り**WorldComputer**と呼ばれます。

このように2つのブロックチェーンはトランザクションが作成され、それがブロックに取り込まれ、最終的にブロックチェーンが形成されるといった部分は同じですが、開発された目的が異なることから、それぞれの目的を達成するために細かな部分には違いがあります。

ここからは具体的にBitcoinとEthereumの違いについて説明します。

ブロックチェーンに記録される情報

Bitcoinでは通貨が「どのアドレス」から「どのアドレス」に「どれだけの通貨」が送られたのかといった情報がトランザクションとしてブロックチェーンに記録されており、**取引の台帳のような役割**を果たしています。

これに対してEthereumでは大きく分けて3つの情報をブロックチェーンに保存しています。

1つ目にEthereum上で実行される**スマートコントラクトのコード**です。

2つ目はコントラクトの実行要求です。ブロックチェーン上のどのスマートコントラクトのどの部分を実行するかといった情報が保存されます。

3つ目にBitcoinと同様に送金に関する情報が保存されます。

複雑なプログラムを動かせるかどうか

Bitcoinは内部通貨を用いて価値の移転を行うことが目的で作られたブロックチェーンです。そのため、ブロックチェーン上でコンピュータのようにあらゆるプログラムを動かすことを目的にしておらず、**複雑なプログラムを動かすことはできません**。

これに対して、EthereumはWorldComputerと呼ばれている通り、**一般的なプログラミング言語で記述できる処理の大半を実行することができます**。

今回紹介した以外にも細かい点に注目すれば、トランザクションやブロックを構成する項目の違いや、ブロックが生成される時間的な間隔も異なります。

3.2 Ethereumの構成要素

ここからはEtehreumの構成要素について説明します。Ethereumのネットワークという大枠から捉えて徐々に詳細について説明していきます。

3.2.1 ネットワーク

Ethereumでは世界中のコンピュータによりネットワークが形成されます。ネットワークに参加しているそれぞれのコンピュータはノードと呼ばれます。それぞれのノードは独立した個人や企業により管理されており、Etheremに関して完全に平等な権限を持っています。

3.2.2 ノード

Ethereumネットワークにノードの1つとして参加するためには、Erthereumの仕様に従って作成されたクライアントソフトを利用します。Ethereumのクライアントソフトで最も利用されているものにGeth(Go Ethereum)があります。言葉の通り、Go言語で作成されています。これ以外にもParityというクライアントソフトがあり、Pythonによって作成されています。ここで紹介した2つのクライアントソフト以外にも、多数のクライアントソフトが存在しており、Etheruemの**仕様に従って作られたソフトウェアであればどのようなものでもネットワークに参加することができます**。

次に、ネットワーク内のそれぞれのノードが保有している情報について説明します。

1つ目はこれまでに発行されたブロックが連なった**ブロックチェーン**です。Bitcoinと同様にブロックにはトランザクションがまとめられ、コントラクトのコードもブロックチェーンに記録されています。

2つ目に**アカウントリスト**です。アカウントリストとは、「どのアカウントがいくら持っているか」といった情報や、「コントラクト内の変数にどのような値が入っているか」など情報がまとめられたものです。アカウントの詳細に関しては、後ほど説明します。

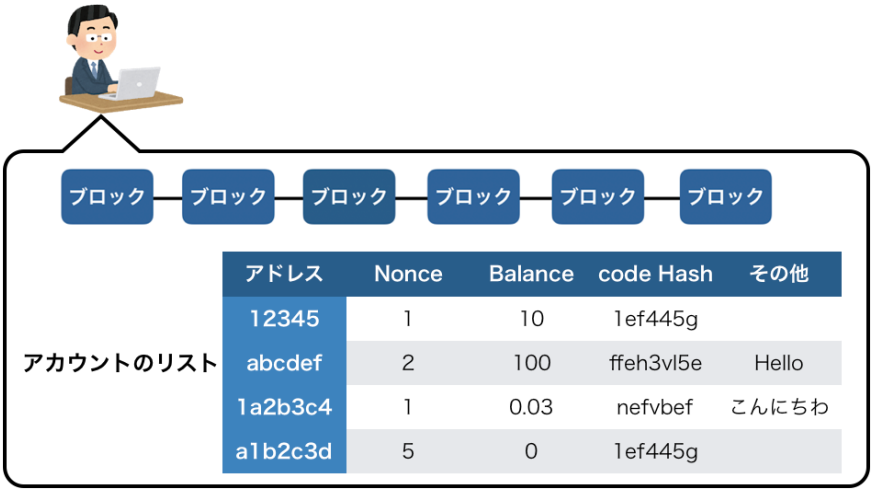


図3.2　それぞれのノードが持っている情報

ここからは、それぞれのノードが持っている2つの情報の内、ブロックチェーンに焦点を当てて、より詳しく説明します。

3.2.3 ブロック

Ethereumのブロックは3つの要素で構成されています。

1つ目が**ブロックヘッダー**です。この部分にはブロックの固有の情報が記録されています。以下にブロックヘッダーの項目の一部を紹介します。

ParentHash

自身の親となるブロックのハッシュ値

Beneficiary

ブロックを作ったマイナーのアドレス

TransactionRoot

ブロックに含まれる全てのトランザクションのRootHash

StateRoot

Ethereumネットワークに存在する全てのアカウントのRootHash

Diffuculty

マイニングの難易度を調整するパラメータ

Number

このブロックがGenesisBlockから数えて何番目のブロックであるか

GasLimit

このブロックで処理することのできる最大のガス量。ガスについては、後ほど説明します。

GasUsed

このブロックに含まれるトランザクションを全て実行するためにかかったガスの量。ガスについては、後ほど説明します。

Nonce

ブロックのハッシュ値を規定以下にするために求められた数値

2つ目にトランザクションリストです。名前の通りこの部分には発行されたトランザクションがまとめられています。

3つ目にOmmerBlockのヘッダーのリストがあります。この部分はマイニングに関連する項目です。今回は詳しい説明は割愛します。

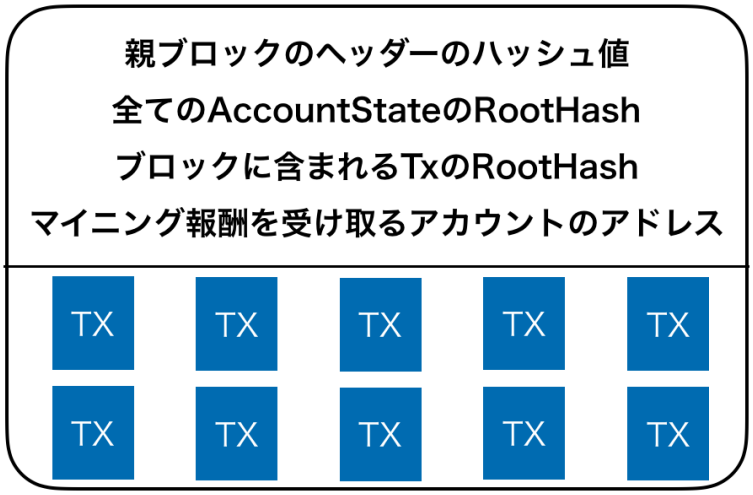


図3.3 ブロックのイメージ

次に、ブロックの中のトランザクションに焦点を当てて説明します。

3.2.4 トランザクション

Ethereumを始めとしてブロックチェーンでは、全ての処理がトランザクションが発行されることから始まります。Ethereumにおけるトランザクションは役割によって2つの種類に分けることができます。

1つ目にContractCreationと呼ばれるトランザクションです。このトランザクションは作成したコントラクトをブロックチェーンに登録し、実行可能な状態にするためのトランザクションです。

2つ目がContractCreation以外のトランザクションであり、これはMessageCallと呼ばれます。仮想通貨の送金や、既にブロックチェーンに登録されているコントラクトを実行するために発行されるトランザクションです。

この2種類のトランザクションは全く同じデータ構造をしており、それぞれの項目に記述される内容のみが異なります。以下にEthereumのトランザクションの構成要素を示します。

Nonce

これまでにこのアカウントから発行されたトランザクションの数。この値により、トランザクションに実行の順序をつけることができる。

GasPrice

ガスの価格。ガスについては、後ほど説明します。

GasLimit

このトランザクションの実行に支払う最大のガスの量。ガスについては、後ほど説明します。

To

トランザクションの種類がMessageCallであれば宛先のアドレスが入り、Contract Creationであればこの項目には何も記述されません。

Value

このトランザクションでのEtherの送金額。Ethereum内の通貨の最小単位である「wei」で記録されます。

Init

トランザクションの種類がContractCreationの場合、作成したコントラクトをコンパイルしたEVMコードがこの項目に記録されます。EVMコードとはEVM(Ethereum Vitural Machine)で実行することができる形式のバイトコードです。MessageCallの場合はコントラクトを登録する必要はありませんので、空欄になっています。

Data

トランザクションにはあらかじめ定められている項目に関連することしか書き込むことが出来ないのではなく、自由なデータを書き込むこともできます。そのようなデータはこの項目に書き込まれます。具体的には、コントラクトを実行する際に、コントラクト内のどの関数にどんなパラメータを与えて実行するのかなどコントラクトの呼び出しに必要な情報などを記録します。これだけでなく、好きな数字や文字列など何でも書き込むことができます。

Ethereum上でコントラクトを作成し、それを実行するために理解しなければならない項目の1つにガスがあります。パブリックブロックチェーンであるEthereumを利用するためには、Bitcoinと同様にトランザクションを発行する際に手数料がかかります。この手数料をEthereumではガスと呼びます。必要となるガスはEVMに要求する処理の複雑さや、ブロックチェーンに対しての書き込みの量などから計算されトランザクション毎に一意に決まります。

次にガスが存在する目的について説明します。1つ目にDoS攻撃を防ぐ目的があります。トランザクションを手数料なしに発行することができると、Ethereumのネットワークに対して、悪意の有無に関わらず大量のトランザクションが発行されることにより、ネットワークが混雑し、処理に遅延が発生することが考えられます。これを防ぐために、トランザクションの発行、つまりEthereumネットワークの利用に対して、手数料を課しています。

2つ目にブロックチェーンのストレージの問題です。Ethereumではブロックチェーンにコントラクトのコードや、その実行要求などが記録されており、これらはEthereumネットワークの参加者により保存されています。Ethereumのブロックチェーンに対しての書き込みに手数料がかからない場合には、トランザクションの発行者は何も考えることなく、ブロックチェーンにデータを保存します。これにより、保存するデータ量が肥大化し、データを保存する参加者の負担が増大することが考えられます。これを防ぐために、ブロックチェーンに対する書き込みに、手数料を課しています。

3つ目にマイナーの負担の削減です。EthereumではBitcoinと異なり、複雑なプログラムを実行することができます。このプログラムはトランザクションを処理するマイナーによって実行されます。複雑で処理に時間のかかるトランザクションが無数に発行されると、マイナーの負担が増え、処理に時間がかかりネットワークには遅延が発生します。これを防ぎ、処理量を最低限に抑えるためにプログラムの処理毎に手数料が定められています。

トランザクションを構成する項目にガスに関する項目が2つあります。それらについて説明します。

GasPrice

GasPriceは支払う**ガスの価格を設定**する項目です。トランザクションによってEthereumに要求する処理には、あらかじめ仕様により定められた計算方法によって、必要となるガスの量が定められます。ガスの単位はgasと表示され、GasPriceの設定により、1gasあたり何ether支払うのかを指定します。設定額については、トランザクションの発行者が決めることができます。GasPriceの高低はトランザクションの処理の優先度に関わります。GasPriceが高く設定されたトランザクションほど、マイナーから優先的に処理を行ってもらえるため、ブロックに取り込まれるまでの待ち時間が短くなります。また、GasPriceの高低は絶対的なものではなく、ネットワーク内にあるトランザクションに設定されたGasPriceから相対的に決まります。ネットワーク内に大量のトランザクションが発行され、遅延が発生している場合には、比較的高いGasPriceを設定しなければ処理に時間がかかります。適正なGasPriceについては、以下のサイトで確認することができます。

EtheGasStation https://ethgasstation.info/

GasLimit

GasLimitは発行したトランザクションで使用する**最大のガス量を設定**する項目です。この項目はトランザクションを発行したアカウントから必要以上の通貨が使われることを防ぐために設定されます。あるコントラクトを実行するためにトランザクションを発行した時、そのコントラクトのバグで無限ループなどの処理が始まった場合、処理量が無限に増加し、それに伴って必要になるガスの量も増加してしまいます。これにより、トランザクションを発行したアカウントの保有する通貨が使い果たされてしまう可能性があります。このような事態を防ぐためにトランザクションの発行者がトランザクション毎に「このトランザクションではこれ以上のガスは使用しません」という意思表示のためにGasLimitを設定します。

3.2.5 アカウント

Ethereumではアカウントという単位で通貨の管理を行います。それぞれのアカウントは通貨量などの複数の項目とアカウントの識別子であるアドレスを持っています。

Bitcoinにおいてアカウントは通貨を保有するための銀行口座のような役割を持っています。EthereumではBitcoinと同様に口座のような役割を持ったアカウントもありますが、少し特徴の異なるアカウントも存在します。これらについて説明します。

外部アカウント

私たちがEtherを保有するために作るアカウントは、外部アカウントと呼ばれます。Bitcoinのアカウントと同様に識別子であるアドレスを持ち、銀行口座のような役割を果たすアカウントです。

コントラクトアカウント

Ethereum上でコントラクトを作成した際に、それぞれのコントラクトに関連づくアカウントが作成されます。このアカウントをコントラクトアカウントと呼びます。コントラクトアカウントでも外部アカウントと同様に通貨を保有することもできます。

外部アカウントとコントラクトアカウントでは、トランザクションを構成する要素に違いはなく、記録される内容のみが異なります。ここからはアカウントの構成要素について説明します。

Address

ハッシュ関数を利用して作られるアカウントの識別子

Nonce

このアカウントからこれまでに発行されたトランザクションの数

Balance

保有する通貨の量

Codehash

コントラクトコードのハッシュ値

StorageRoot

上記以外のパラメータを木構造によりまとめたハッシュ値

外部アカウントとコントラクトアカウントで記録される内容が異なるのが、CodeHashの部分です。コントラクトアカウントでは、**コントラクトのコードのハッシュ値がCodeHashに記録**されます。これに対して外部アカウントでは、空文字のハッシュ値が記録されています。実際のEthereumのトランザクションを以下から確認してみてください。

Etherscan(https://Etherscan.io/)

これ以外に**トランザクションを自ら発行することが出来るのは外部アカウント**のみであるという違いがあります。Ethereumネットワーク上で処理を行う時には外部アカウントからトランザクションを発行することが必要になります。

しかし、厳密にはコントラクトアカウントからもトランザクションを発行することができます。Ethereum上のコントラクトには、自身以外のコントラクトを呼び出す機能を持たせることができます。コントラクトAを外部アカウントから発行したトランザクションで呼び出した時、コントラクトAのコードの中にコントラクトBを呼び出す処理があった場合、コントラクトAからコントラクトBを呼び出すためのトランザクションが発行されます。このトランザクションを内部トランザクションと呼びます。内部トランザクションの発行はあくまでも、外部アカウントが発行したトランザクションがトリガーとなっており、コントラクトが勝手にトランザクションを発行し、Ethereum上で勝手に処理を行うことはありません。

アカウントリスト

ネットワーク内に存在するそれぞれのノードが保有している情報について説明を行なった際に、それぞれのノードはブロックチェーン以外にアカウントリストという情報を保有していると説明しました。アカウントリストとはEthereum内にある**全てのアカウントについて、それぞれが保有する通貨量などのパラメータなどの一覧**が記録されたリストです。このアカウントリストはEthereumの0番目のブロックであるジェネシスブロックから順に最新のブロックまで、ブロックチェーンに含まれるトランザクションを実行することで得ることができます。

3.2.6 状態

Ethereumについて学ぶ際に「状態」や「State」と言った単語が頻繁に登場します。Ethereumにおいては2つの状態が存在します。それらについて説明します。

AccountState

AccountStateとは**それぞれのアカウントの状態**であり、アカウントの保有額などの全ての項目を全体のことを状態と呼びます。外部アカウントもコントラクトアカウントも同様です。

WorldState

WroldStateは**Ethereum全体の状態**であり、**全てのアカウントの状態を考慮した状態**になります。1つでもアカウントの状態が遷移すると、WorldStateもこれに伴って遷移します。

3.3 Ethereumの処理の流れ

ここからはEthereum上で行われる様々な処理の流れについて説明します。

3.3.1 コントラクトの登録

まず、コントラクトがブロックチェーンに登録されるまでの流れについて説明します。

コントラクトコードの作成

どのようなコントラクトをEthereumに登録したいかを考え、コードを作成します。このとき使用する言語はEthereum上でコントラクトを作成するための専用の言語を用いる必要があります。

作成したコントラクトは専用のコンパイラでコンパイルします。コンパイル後のコードのことをEthereumではEVMバイトコードと呼びます。

トランザクションの作成

Ethereum上にコントラクトを登録するためのトランザクションを作成します。このトランザクションはContractCreationです。このときトランザクションのinitの項目にEVMバイトコードを記入します。Ethereumではコードはブロックチェーンに登録されるものの、コンパイル後の状態であるため人間からは可読性の低いものになっています。

図3.4 ContractCreationのイメージ

トランザクションの伝達

作成したトランザクションはEthereumネットワークに伝達します。この際に、それぞれのノードは受け取ったトランザクションを検証し、その結果正しいものだけを自身の手元に保存し、かつ他のノードに伝達します。

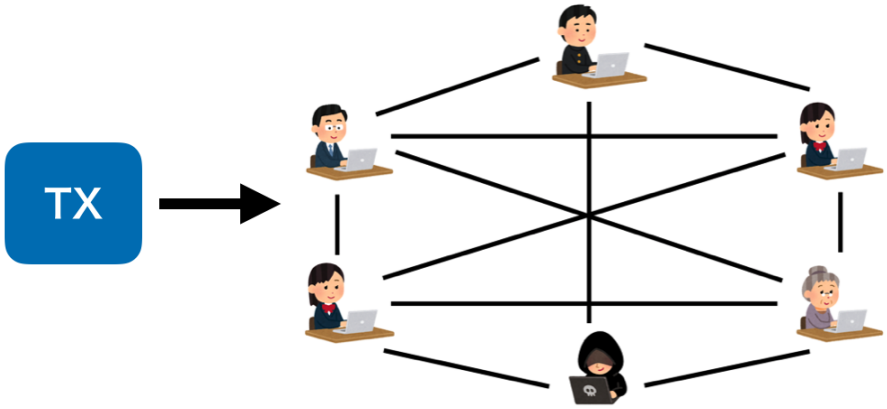
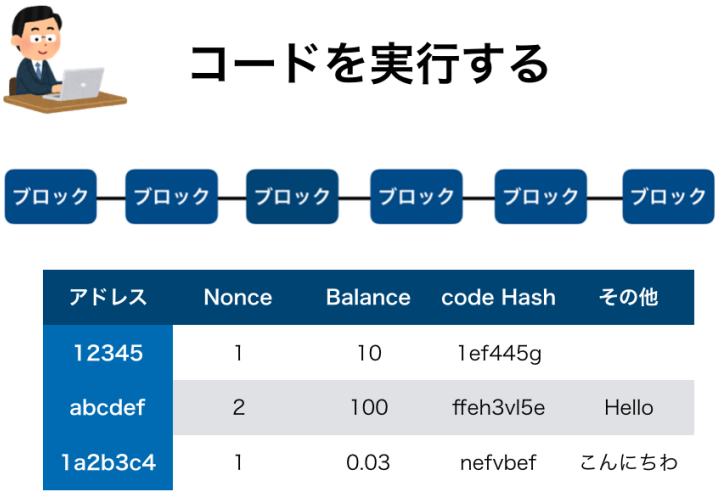


図3.5 トランザクションの伝達

トランザクションの実行

トランザクションを受け取ったマイナーはそのトランザクションで指示された処理を行います。ContractCreationを受け取ったマイナーは、新たにコントラクトアカウントを作成し、アカウントリストにアカウントの情報を記録します。



アドレス	Nonce	Balance	code Hash	その他
12345	1	10	1ef445g	
abcdef	2	100	ffeh3vl5e	Hello
1a2b3c4	1	0.03	nefvbef	こんにちわ

図3.6　コードを実行する

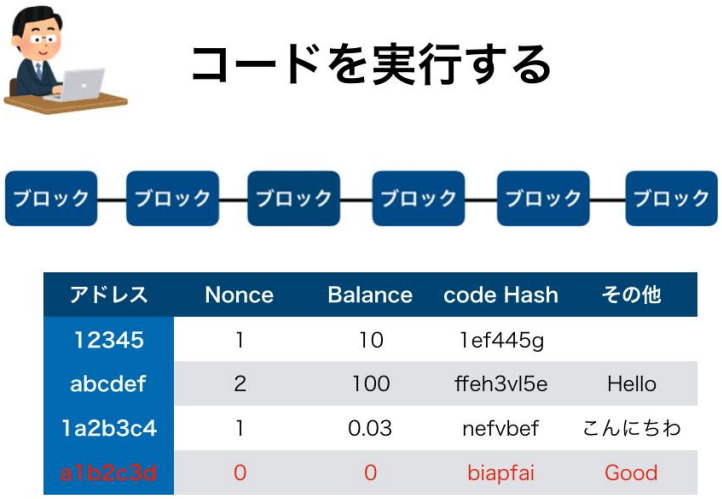


図3.7 新たにアカウントにコントラクトアカウントが作成される

ブロックの作成

Etheruemのネットワークには続々とトランザクションが投げ込まれており、マイナーは次々に処理を行います。それと同時に、現在の全てのアカウントの状態を踏まえてWorldStateを作り、ブロックを作成します。

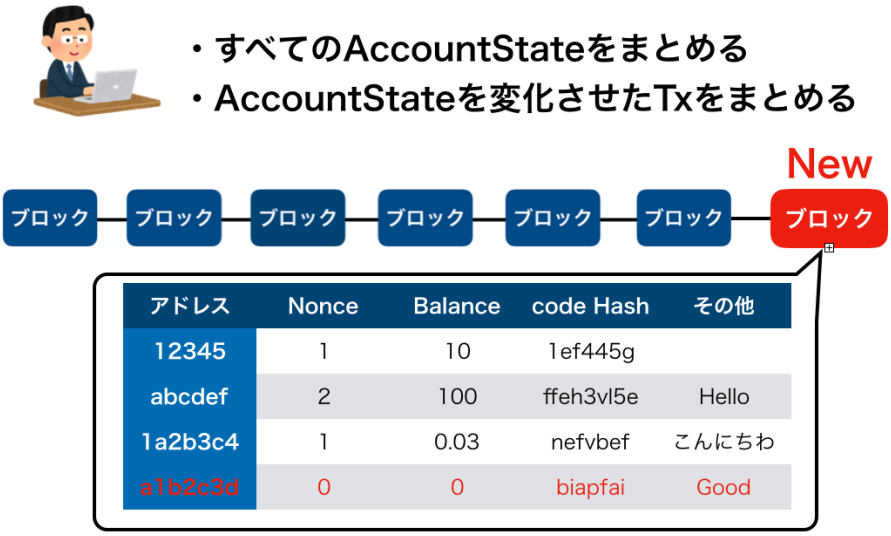


図3.8 ブロックの作成

以上で、作成されたトランザクションはブロックに取り込まれ、コントラクトをブロックチェーンに登録することができました。これによって、誰でもこのコントラクトを実行できるようになります。

3.3.2 コントラクトの呼び出し

次に登録したコントラクトを呼び出す際の処理の流れについて説明します。また、通常のEtherの送金に関しても同じ処理が行われます。

トランザクションの作成

既にEthereumのブロックチェーンに登録されている**コントラクトアカウントのアドレスをトランザクションのToの項目に記入**します。また、コントラクト内のどの関数にどのようなパラメータを渡すかといった情報もトランザクションに記述する必要があります。この情報はトランザクションのDataの項目に記述します。また、外部アカウントに対しての送金や、コントラクトに対して送金を行う必要がある際には、トランザクションのValueの項目に送金額を記入します。

図3.9 MessageCallのイメージ

トランザクションの伝達

作成されたトランザクションはEthereumネットワークに伝達します。トランザクションの伝達に関しては、ContactCreationと同様に、それぞれのノードで検証の結果正しいと判断されたトランザクションのみが伝達、保存します。

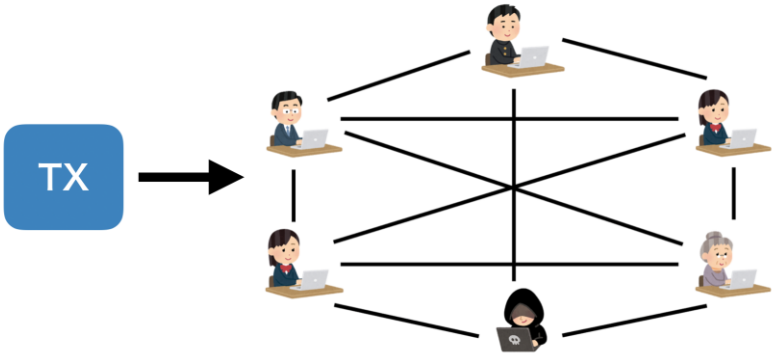


図3. 10　トランザクションの伝達

アカウントの状態の遷移

トランザクションで指定された**コントラクトをマイナーが実行**します。マイナーはまず、指定されたコントラクトのコードをブロックチェーン上にあるトランザクションの中から探します。その後、トランザクションに含まれている情報とコードを元に、**コントラクトの記述通りにアカウントの状態を遷移**させます。具体的には、送金作業であれば、送金者のアカウントのbalanceが減り、受金者のアカウントのbalanceが増やすことや、コントラクトに登録されている変数の値を変えることです。このようにトランザクションを実行することで1つ以上のアカウントの状態が遷移します。

図3. 11　ブロックチェーンから指定されたコントラクトを探す

コードを実行する

アドレス	Nonce	Balance	code Hash	その他
12345	1	10	1ef445g	
abcdef	2	100	ffeh3vl5e	Good Night
1a2b3c4	1	0.03	nefvbef	こんにちわ
a1b2c3d	5	0	1ef445g	

図3.12 トランザクションで渡された情報とコントラクトコードを踏まえてアカウントの状態を遷移させる

ブロックの作成

複数のトランザクションを実行し、アカウントの状態を遷移させた後に、ブロックの作成を行います。

以上で、作成したトランザクションがブロックに取り込まれ、コントラクトの実行、または送金作業が終了します。

4. DApps(分散型アプリケーション)

ここではブロックチェーンを利用したアプリケーションであるDApps(Decentralized Applications,分散型アプリケーション)の概要と、その仕組みについて紹介します。

4.1 DAppsの仕組み

まずは既存のアプリケーションと比較しながら、分散型アプリケーションの仕組みについて説明します。

4.1.1 既存のアプリケーションの構成

既存のアプリケーションはクライアント・サーバー型のネットワークが多く利用されています。このようなアプリケーションは、大きく2つの要素に分けることができます。１つはフロントエンドと呼ばれる部分で、もう１つはサーバーサイド、バックエンドと呼ばれる部分です。

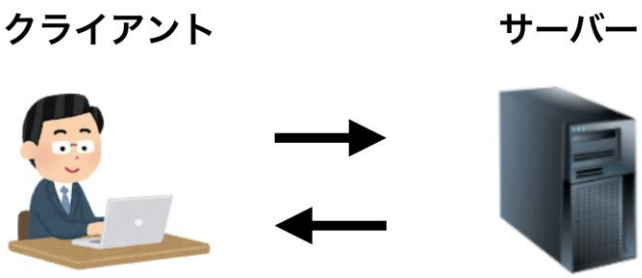


図4.1 クライアント・サーバー型のネットワーク

フロントエンド

ユーザーが直接目にし、操作する部分のことをフロントエンドと呼びます。具体的にはWebアプリケーションであればWebブラウザに表示される部分、スマートフォンアプリであればスマートフォンに表示される部分がこれに当たります。

サーバーサイド

フロントエンドの入力データや指示をもとに、処理を行う部分や、データを保存する処理を行う部分をサーバーサイドと呼びます。一般的なシステムの利用者は基本的にはサーバを目にすることはありません。

4.1.2 DAppsの構成

既存のアプリケーションに対して、ブロックチェーンを用いたサービスの構成について説明します。ブロックチェーンを利用したアプリケーションでは、既存のアプリケーションの、**サーバーサイドの処理や、データの保存など一部の機能をブロックチェーンで置き換え**ています。

図4.2 ブロックチェーンを用いたアプリケーション

ここで注意しなければならないのが、サーバーサイドの処理の全てを置き換えることは現在のブロックチェーンではできないということです。これにはいくつかの理由があります。

1つ目に**トランザクションに書き込むことのできるデータの容量には制約**があることが挙げられます。このような制約により、画像や動画などのデータ容量の大きいデータをブロックチェーンに保存することは基本的にはできません。データを複数のトランザクションに分割し、ブロックチェーンに保存することは可能ではありますが、大量のデータを保存するため、多くのGasが必要になりDAppsの利用者が高額の手数料を支払う必要が出てきます。

2つ目にトランザクション手数料が挙げられます。既存のアプリケーションで行なっていたような、複雑な処理をコントラクトに記述し実行した場合、多くのガスが必要になり、DAppsの利用者が高額の手数料を支払う必要が出てきます。

3つ目に処理能力と処理速度の問題があります。Ethereumを始めとした多くのパブリックブロックチェーンではその仕様上、即時性の必要とされる処理を安全に行うことができません。また、処理能力に関しても仕様が定まった段階である程度決められており、現在のパブリックブロックチェーンでは大量の処理を裁くことはできません。

このような点から基本的には、従来と同様のアプリケーションを準備し、必要な部分だけブロックチェーンを利用することが推奨されます。

4.2 DAppsの利点と課題点

ここからはブロックチェーンを利用してアプリケーションを作成することの利点と課題点について説明します。

ここでは改ざん耐性の高さ、可用性の高さなどのブロックチェーン自体の利点や処理能力や処理速度、手数料などのようなブロックチェーン自体の課題点については触れません。

4.2.1 利点

決済の仕組みを簡単に作ることができる

私たちが普段利用するアプリケーションの中には、サービスに対して支払いの仕組みが組み込まれているものが多くあります。決済の方法としては、クレジットカード決済、銀行での振込などいくつかの方法を考えることができます。これらの方法は一般的ではあるものの、決済の仕組みを組み込んだアプリケーションを作ることは非常にコストがかかります。

これに対して、ブロックチェーンを利用したアプリケーションでは、仮想通貨を利用することによって**簡単に決済の仕組みを導入**することができます。このような決済目的でブロックチェーンが利用されることもあります。

情報の公共化

これまでのアプリケーションでは、そのサービスに関する全ての情報はシステムを提供する企業が管理していました。この場合、企業がサービスの提供をやめることや、企業が破綻することでサービスは終了し、管理していたデータもクライアントの意思とは無関係に失われてしまうことになります。例えば、オンラインゲームのようなサービスで時間をかけて育成したキャラクターの情報もサービス終了と同時に全て失われることになります。

このような問題に対して、データをブロックチェーンに保存することで、自身の**データはサービスが終了した後も保存し続ける**ことができます。これにより、その情報を引き継いだサービスを作成することもできるようになります。

4.2.2 課題点

次にブロックチェーンをDAppsに利用する際の課題点について説明します。

システムが中央集権的になる

ブロックチェーンの特徴の1つに管理者を必要とせずにシステムが稼働する非中央集権性が挙げられます。多くのDAppsではこの特徴を活かすことができていません。ブロックチェーン自体には、管理者が存在していませんが、図4.2からわかるようにクラアントと、ブロックチェーンの間にサーバが存在し、その部分に**管理者が存在**します。つまり、DAppsのシステム全体は中央集権的な仕組みになっていると言えます。このため、DAppsサービスの提供者がサービスを終えてしまうと、ブロックチェーン上にサービスに関連するコントラクトや情報は残っているにも関わらず、サービスを利用することができなくなってしまいます。また、管理者の権限によりDAppsに急な仕様の変更などが加えられる可能性もあります。

可用性

ブロックチェーンの特徴の1つに高い可用性を挙げられます。しかし、DAppsのシステム全体を見ると、**従来のアプリケーションと可用性については変わらない**ことがわかるのではないかと思います。これは、システムの一部を切り取ると従来のクライアント・サーバー型の特徴を持った部分が存在し、ブロックチェーン自体は高い可用性を持っていても、そこにアクセスするサーバが停止してしまう可能性があるためです。

このようにブロックチェーンを利用しているからといって、ブロックチェーンの特徴を全て活かすことは現在はできません。

4.3 DAppsの事例

ここからは現在利用することのできるDAppsや、これからの利用が期待されているDAppsについて、いくつかの分野に分けて紹介します。

4.3.1 ゲーム

DAppsの事例として、まず一番に挙げることができるのがゲームです。いくつか具体的なゲームを紹介します。

CryptoKitties

公式サイト　https://www.cryptokitties.co/

Ethereum上に構築される仮想的な猫を育成するゲームです。ゲーム内で購入した猫同士を交配すると新たな子猫が生まれ、それらをユーザー間で売買することができます。この売買の際にはEthereumの内部通貨であるEtherが用いられ、Etherは仮想通取引所で法定通貨に交換することができます。このため、ギャンブルのようなゲームになっています。

子猫は基本的には両親が持つパラメータを引き継いで生まれてきますが、稀に突然変異により珍しい猫が生まれてきます。このような猫は高値で取引される場合が多く、過去には数百万円で取引されたこともあります。

このゲームではデータの全てをEthereumで行っている訳ではなく、猫の所有権や特徴の管理、猫の所有権の移転の取引にブロックチェーンを利用しています。そのほかの処理やデータの保存に関しては、既存のオンラインゲームと同様に、管理者が立てたサーバにより行われています。

クリプ豚

公式サイト　https://www.crypt-oink.io/landing/ja/index.htm

CryptoKittiesと同様に豚を育成し、売買することのできるゲームです。このゲームは日本初のブロックチェーンゲームとして大きな注目を浴びました。このゲームでは育成した豚を使ったレースなども行うことができます。

このゲームも豚の所有権やパラメータの管理、所有権の取引に関連する部分に、ブロックチェーンを利用しており、それ以外の部分に関しては従来のアプリケーションと同様にサーバによって管理が行われています。

4.3.2 分散型取引所

私たちが仮想通貨を入手する方法で最も簡単な方法が、仮想通貨取引所に法定通貨を支払い、仮想通貨を購入する方法です。一般的に仮想通貨取引所は企業により運営されています。これに対して、ブロックチェーンを用いて**非中央集権的に運営される仮想通貨取引所**を分散型取引所と言います。

これまでのような企業により運営される中央集権的な取引所を利用する際には、ユーザーは自身が保有するアカウントの秘密鍵を取引所に預ける必要があります。この状態は取引所という1つの企業を完全に信用し、自らの資産の管理を委任している状態であると言い換えることができます。取引所がこのような信用に依存した仕組みであるため、ブロックチェーンや仮想通貨の仕組み自体は安全であっても、ユーザーの秘密鍵を一括して管理している取引所が攻撃されると、ユーザーが預けた多額の資産が流出することになります。

非中央集権的な仕組みのブロックチェーンに対して、仮想通貨を取引する中央集権的な取引所に大きなリスクがあることは、とても好ましい状態であるとは言えません。

このような仕組みに対して、分散型取引所ではブロックチェーンを基盤とすることで、管理者が存在しない状態で、秘密鍵の管理はユーザー自身で行ったまま取引を行うことができます。

Ether Delta

公式サイト　https://Etherdelta.com/

EtherDeltaはEthereum上に構築される分散型取引所であり、ICO銘柄を素早く上場することで有名です。Ethereum上にあるコントラクトの利用率で常に上位を保っていることからも、最も活発なDEXであると言うことできます。

EtherDeltaを利用するために必要になる手数料には「Ethereumネットワークに対する手数料」と「プラットフォームに対する手数料」の2つがあり、EtherDeltaは「プラットフォームに対する手数料」によりマネタイズを行っています。

4.3.3 身分証明

私たちはオンライン、オフラインに限らず様々なサービスを、個人情報を提供することにより利用しています。例としては、ネットショップを利用するときには住所やクレジットカード番号などの様々な情報を企業に対して提供しなければならないことなどが挙げられます。また、利用するサービスによっては、サービス利用時における身分証明に長い時間がかかることもあります。加えて、サービスにより個人情報を管理する企業が異なるため、繰り返し同じ個人情報を登録しなければならないという難点もあります。これにより、複数の企業に個人情報が保有されることになり、情報漏洩リスクも高くなります。

uPort

公式サイト　https://www.uport.me/

uPortでは**個人情報をブロックチェーンで一元的に管理**し、あらゆるサービスを利用するときスマホ等のデバイスでuPortを介して利用すると、簡単に身分証明ができるようになることを目指しています。開発はEthereumブロックチェーンに特化した分散型アプリケーション開発スタジオConsenSys社により行われています。最終的には名前や生年月日などの情報だけでなく、税金の支払い情報、医療機関でのカルテ情報、車の所有情報などある特定の個人に関するあらゆる情報を管理することを目標としています。

個人情報の管理において**適切な相手に対して適切な情報だけを公開**するという柔軟な情報公開を行うことも重要です。具体的には、保険会社に個人情報を求められたときに必要なデータだけ保険会社に提供し、その他のデータは公開しないことや、年齢確認を求められた際には、年齢のみを公開することなどが挙げられます。uPortでは、身分証明を行う際に登録されている全ての情報を用いるのではなく、必要最低限の情報のみを相手に伝えることができます。

5. コントラクトの作成

ここからはEthereum上で動くコントラクトの作成と、その際に必要となる環境や開発言語について説明します。

5.1 Solidity

ここではEtheruem上でコントラクトを作成するための言語であるSolidityや、コントラクトを作成するための環境について説明します。

SolidityはEthereum上でコントラクトを作成するために作られた、オブジェクト指向型の言語です。C++やJavascriptの影響を受けて設計され、現在Ethereum上でコントラクトを作成する際の最もメジャーな言語であると言えます。

5.1.1 開発環境

今回はコントラクトを作成するためにRemixという統合開発環境を利用します。Remixではコードの作成からコンパイル、ブロックチェーンへのデプロイといった一連の流れの全てを行うことができます。

Remix公式サイト　https://github.com/Ethereum/remix

Remixを利用する方法は2つあり、1つはローカルに環境を構築する方法で、もう1つはオンラインで利用する方法です。今回はオンライン版を使用し、コントラクトの作成から、作成したコントラクトの利用までの流れを説明します。

はじめにオンライン版を使用する際の注意点として、作成したファイルはブラウザのローカルストレージに保存されるため、永続的に保存する必要がある場合は、コードをコピーし、自身のPCにファイルを作成して保存してください。

Remixを開く

- 「https://remix.Ethereum.org」 にアクセスします。

Remixの設定

- 言語の選択項目で Solidity を選択します。
- NewFile を選択し、ファイル名を Hello.sol にします。
- 画面右のエディタ部分に以下のコードを記述します。

```
pragma solidity ^0.5.10;

contract HelloWorld {
    string message = "HelloWorld";

    function getMessage() public view returns(string memory) {
        return message;
    }

    function changeMessage(string memory _message) public {
        message = _message;
    }
}
```

- コードの作成が終了すると、画面左のコンパイルボタンをクリックします。コンパイルが成功すると、緑のチェックマークが出てきます。
- コンパイルが終了すると、コンパイルボタンの１つ下のデプロイボタンをクリックします。

ここでは世界中のノードによって構成されているEthereumネットワークに作成したコントラクトをデプロイするのではなく、Remixの中で動くブロックチェーンのシミュレータに対して、作成したコントラクトをデプロイします。また、Remixのブロックチェーンのシミュレータでは、立ち上げた段階でアカウントが5つ作成され、それぞれのアカウントが100ether持っています。そのため、トランザクションを発行する際に必要となるトランザクション手数料を支払うために、通貨を取得する必要はありません。

ここまでの作業で、作成したコントラクトはブロックチェーンに登録され、利用できる状態になっています。

次に登録したコントラクトのRemixからの利用方法について説明します。

- デプロイボタンがある欄の下に「Deployed Contracts」という項目があり、この部分をクリックするとデプロイされたコントラクト一覧が表示されます。
- 今回デプロイしたHelloWorldコントラクトを選択します。選択すると、「changeMessage」と「getMessage」の2つのボタンが表示されます。

これらのボタンは、コントラクト内で作成した関数の名前に紐づいており、ボタンをクリックすると、それぞれの関数を実行することができます。

- まずはgetMessage関数を利用します。getMessageボタンをクリックし、そうするとボタンの下にコントラクト内で設定したmessage変数の初期値である、HelloWorldが表示されます。
- 次に、changeMessage関数を利用します。コントラクト内の関数を見てわかるように、関数を実行する際には新たな文字列を渡す必要があります。この新たな文字列をボタンの横のテキストボックスに入力します。注意しなければならない点が、Remixで文字列をテキストボックスに入力する際には、シングルクオテーションまたは、ダブルクオテーションで囲う必要があります。今回は例として、新たに”Good Morning”とテキストボックスに入力します。その後、changeMessageボタンをクリックすることで、文字列を変更することができます。その後、getMessageボタンをクリックすると、Good Morningが表示され、正しく文字列が変更されたことがわかります。

Remixの操作について

今回の例で紹介していなかったRemixの操作について説明します。

アカウントの切り替え

Remixでブロックチェーンのシミュレータを利用する場合には、あらかじめ5つのアカウントが作成されており、それらのアカウントを切り替えて利用することができます。この方法について説明します。

コントラクトをデプロイしたボタンの上に「Account」という項目があります。その横にアカウントのアドレスが表示されている部分があるので、その部分を選択します。ここで任意をアカウントに切り替えることができます。

トランザクションに通貨を付加する

トランザクションを発行し、コントラクトを実行する際にコントラクトや、特定の相手に対して送金を行う場合があります。このような場合、アカウントの切り替えを行なった項目の2つ下に「Value」という項目があり、ここから通貨量を指定します。このときに送金する通貨量の単位も選択することができます。

5.1.2 Solidityの基本

ここからはEthereum上でのコントラクトの開発言語であるSolidityを用いてコントラクトの作成を行います。

Helloコントラクトの作成

以下のコードの例にSolidityについて説明を行います。

```
1. pragma solidity ^0.5.10;
2.
3. contract HelloWorld {
4.     string message;
5.
6.    constructor(string memory _message) {
7.       message = _message;
8.    }
9.
10.    function getMessage() public view returns(string memory) {
```

```
11.         return message;
12.     }
13.
14.    function changeMessage(string memory _message) public {
15.      message = _message;
16.    }
17.}
```

1行目では使用する言語とそのバージョンについての設定を行っています。例では**Solidityのバージョン0.5.10以上のコンパイラを指定**しています。

```
pragma solidity ^0.5.10;
```

3行目の「contract HelloWorld」の部分では「HelloWorld」 という名前のコントラクトを宣言しています。**コントラクトに関する処理はこの括弧の中に記述**する必要があり、**言語の設定以外をコントラクトのブロックの外に書くことはできません**。

```
contract HelloWorld { }
```

4行目では変数を宣言しています。このようにコントラクトのブロックの直下で宣言された変数は**状態変数**と呼ばれ、ブロックチェーンに記録されます。ここでは文字列型の message という名前の状態変数を作成しています。変数については、関数の内部でも宣言することはできますが、そのような変数は関数の終了ともに破棄され、ブロックチェーンには記録されません。Solidityの変数の型については、後ほど説明します。

```
string message;
```

コンストラクタ

6行目の「constructor」から始まる行では、コントラクトのコンストラクタを作成しています。コンストラクタはコントラクトをデプロイする時に一度だけ実行される関数であり、状態変数の初期値の設定などを行うことができます。今回はコントラクトが持つ状態変数である、message にコンストラクタで初期値を与えます。

```
constructor(string memory _message) {
    message = _message;
}
```

9行目では登録した文字列を呼び出すための関数を作成しています。関数名は「getMessage」です。関数の宣言の行の最後にあるreturnsの括弧の中では関数の返り値の型を指定しています。

```
  function getMessage() public view returns(string memory) {
       return message;
   }
```

関数の可視性

関数名の直後に記述されている「public」は、関数の可視性を宣言しているものです。Solidityにおいて関数の可視性は4種類存在します。

- external：外部のコントラクトから実行可能な関数になる。実装されたコントラクト自身からは外部呼び出しとして呼び出すことができる。
- public：コントラクト内部からやメッセージ経由で外部から呼び出すことができる。
- internal：コントラクトの内部やコントラクトを継承したコントラクトから呼び出すことができる。
- private：コントラクト内部からのみ呼び出すことができる。

関数修飾子

関数修飾子は、**関数を実行する前に実行したい処理**を宣言することができます。今回利用しているview修飾子はブロックチェーンに対して書き込みは行わず、データを参照するだけの関数とすることを宣言しています。これ以外にもブロックチェーンに対して、書き込みや参照を行わない関数を作成する際に利用するpure修飾子や、関数内でEtherのやり取りを行うためのpayable修飾子などがあります。

関数修飾子はviewやpure、payableのようにあらかじめ利用することのできるものもありますが、以下のように自身で作成することもできます。

```
modifier onlyOwner {
    require(msg.sender == owner);
     _;
 }
```

modifier というキーワードを用いることで、関数の修飾子を作成することができます。上記ではあらかじめ登録された状態変数ownerに格納されたアドレスと、関数を実行しようとしているアドレスを比較します。2つのアドレスが同じであれば、関数を実行することができ、異なれば関数を実行することはできません。また、ここで利用したrequireは()の中がtrueであれば、そのまま処理を実行し、falseであればそこで処理を止め、トランザクションの処理を終えます。

最後のアンダースコアは、modiferは最後に「 _; 」で終わることが決められています。

上で作成したonlyOwner修飾子の利用例を以下に示します。

```
function sayHello() public pure onlyOwner returns(string memory) {
        return "Hello";
 }
```

14行目では、状態変数であるmessageの値を変更するための関数を宣言しています。10行目の文字列を参照するための関数と異なる点として、引数として文字列を

渡している点があります。また、ブロックチェーンに記録されているmessage変数の書き換えを行う必要があるため、先ほどの関数で利用したview修飾子は利用しません。

```
function changeMessage(string memory _message) public {
    message = _message;
}
```

Typeコントラクトの作成

Solidityは**静的型付き言語**であり、変数の宣言時に型を指定する必要があります。

ここでは、Solidityの変数の型について、以下のSolidityで使うことのできる様々な型を含めたコントラクトを用いて説明します。また、Solidityのそれぞれの型はその型に準じた初期値を持ちます。それぞれの型の初期値についても、紹介します。

```
1.  pragma solidity ^0.5.10;
2.
3.  contract Type {
4.      bool boolean = true;
5.
6.      uint number = 100;
7.
8.      address to = address(this);
9.
10.     string message =  “Hello” ;
11.
12.     uint[3] uintArray= [1, 2, 3];
13.     address[] addressArray = [address(this), address(0)];
14.
```

```
15.    mapping(address => string) addressToName;
16.    addressToName(address(this)) =  “This contract Address” ;
17.
18.     struct Human {
19.         uint age;
20.         string name;
21.         bool admin;
22.     }
23.}
```

4行目のbool型は論理値を扱う型です。初期値にはfalseが入っています。

```
bool boolean = true;
bool booleanDefault; // false
```

6行目のuintは符号なし整数型です。初期値には0が入っています。uint型はunit8からuint256まで8bit刻みで型が存在します。uintはuint256のエイリアスになっています。

```
uint number = 100;
uint initNumber; // 0
```

8行目はアドレス型です。Ethereumのアカウントの識別子であるアドレスを管理するための型です。初期値には 0x0 が入っています。「address(this)」でコントラクト自身のアドレスを取得することができます。

```
address send = 0x9124429e9fDB12fC3D4FBC110B40A2DDc39D59eC;
address to; // 0x0
address contractAddress = address(this);
```

10行目は文字列型です。初期値には空文字が入っています。

```
string message =  “Hello”;
string greeting; //  “”
```

12行目と13行目は配列です。Solidityでは静的な配列、動的な配列を利用することができます。例では静的な整数型の配列と、動的なアドレス型の配列を示しています。

```
uint[3] uintArray= [1, 2, 3];
 address[] addressArray = [address(this), address(0)];
```

配列

Solidityでは配列に対して2つのメソッドが用意されています。1つ目に配列の要素数を取得するメソッドである「length」、2つ目に配列の最後に要素を追加する「push」です。使い方に関しては以下に示します。

```
uint[] uintArray = [0, 1, 2];

uintArray.push(3) // [0, 1, 2, 3]
uintArray.length  // 4
```

配列に対して、任意の要素を削除することや、配列の並べ替えを行うためのメソッドは準備されていません。これらの処理を行いたい場合には、自身で実装する必要があります。

15行目はmappingで、16行目はその利用例です。一般的にハッシュ型や辞書型と呼ばれる型です。例では、addressをkeyとして文字列を値として持つaddressToNameという名前のmappingを作成しています。

```
mapping(address => string) addressToName;
addressToName(address(this)) = “This contract Address”;
```

18行目は構造体です。文字列や整数などの様々な型を持つ構造体を定義することができます。

```
struct Human {
        uint age;
        string name;
        bool admin;
    }
```

変数の型には大きく分けて値型と参照型があります。値型の例としては、uint型、bool型、address型などがあり、参照型はstring型、配列型があります。この2つの型にはいくつか使い分けが必要な場合があります。１つの例として、関数に値を渡す時と、関数の返り値の型を指定する時があります。以下に例を示します。

```
● function getUint(uint number) public view returns(uint) {}
● function getString(string memory string) public view returns(string memory) {}
```

上記のgetUint関数では、uint型の変数を受け取り、uint型の変数を返しています。このように値型の変数を関数に渡す時、受け取る時は変数の型のみを記述するだけでよいです。

次に、getString関数では、string型の変数を受け取り、string型の変数を返しています。この時、uint型の受け渡しと異なり型を指定するstringの後に、「memory」と書いてあります。これは参照型の変数の受け渡しの際に必要となります。

これ以外にコントラクトを作成する際に頻繁に利用するメソッドなどを以下に紹介します。

msg.sender

トランザクションを発行したアドレスを取得することができます。以下の例では、状態変数のownerに対してコンストラクタ内で、コントラクトをデプロイしたアドレスを代入しています。

```
address owner;

constructor() public {
        owner = msg.sender;  // コントラクトをデプロイしたアドレス
 }
```

msg.value

トランザクションに含まれる通貨量を取得することができます。単位はEthereumの内部通貨の最小単位であるweiです。以下にトランザクションに通貨が含まれている場合と、含まれていない場合で処理を分ける関数を例に挙げています。

```
function sayHello() public payable returns(string memory) {
        if(msg.value != 0) {
            return "Hello";
        } else {
            return "See you";
        }
    }
```

address.transfer(value)

指定したアドレス宛に通貨を送金することができます。以下にアカウントと、額を指定して送金を行う関数を例に示します。このとき、宛先として指定するアドレ

スは単なるaddress型ではなく、address型にpayable修飾子をつけた、「address payable型」にする必要があります。

```
function transferEther(address payable _to, uint _value) public payable {
        _to.transfer(_value);
    }
```

address.balance

アドレスが保有している通貨の量を取得することができます。

```
mapping(address => bool) richAddress;

    function addRichAddress(address _richAddress) public {
        require(_richAddress.balance >= 100 Ether);
        richAddress[_richAddress] = true;
    }
```

5.2 Solidity演習

ここからはSolidityを用いてコントラクトの作成を行っていきます。提示された仕様に従ってRemixでコントラクトを作成し、動作の確認まで行ってください。

演習1

1. Solidityのバージョンを0.5.10以上に設定し、Sampleという名前でコントラクトを作成してください。
2. 文字列型で、greetingという状態変数を作成してください。
3. コンストラクタを作成し、状態変数greetingに初期値Helloを与えてください。
4. greeteという名前の関数を作り、状態変数greetingを返してください。
5. changeGreetingという名前の関数を作り、状態変数greetingの値を変更できるようにしてください。
6. コントラクトをデプロイしたアドレスを管理者として状態変数に設定し、そのアドレスのみがchangeGreetingを実行することができるようにしてください。

演習2

1. Solidityのバージョンを0.5.10以上に設定し、Sampleという名前でコントラクトを作成してください。
2. キーがアドレス型、値が文字列型のmapping型の状態変数をaddressToNameという名前で作成してください。
3. コンストラクタを作成し、コンストラクタ内で状態変数addressToNameにキーをトランザクションの発行者のアドレス、値を文字列でOwnerをとして登録できるようにしてください。
4. setMyNameという名前の関数を作成し、状態変数addressToNameに関数を呼び出したアドレスと、任意の名前を登録できるようにしてください。
5. getNameという名前の関数を作成し、アドレスを渡すと状態変数addressToNameに登録された名前を返すようにしてください。

6. setNameという名前の関数を作成し、状態変数addressToNameに任意のアドレスと名前を同時に登録することができるようにしてください。ただし、この関数は10ether以上持っているアドレスのみ実行することができるようにしてください。

演習3

1. Solidityのバージョンを0.5.10以上に設定し、Bankという名前でコントラクトを作成してください。
2. キーがアドレス型、値が符号なし整数型のmapping型の状態変数をaddressToAmountという名前で作成してください。
3. depositという関数を作成し、ETHをコントラクトに対して送金すると、送金を行なったアドレスとそのアドレスのこれまでに預けた額が状態変数addressToAmountに保存されるようにしてください。ただし、一度に預けることができる額は1ETH以上の整数値のみです。
4. getMyAmountという関数を作成し、状態変数addressToAmountから自身がこれまでにコントラクトに対して送金した額を参照できるようにしてください。
5. withDrawという関数を作成し、自身がコントラクトに預けた額以下の通貨を引き出すことができるようにしてください。これに伴って、状態変数addressToAmountの値も変更してください。ただし、引き出すことができるのは1ETH以上の正の整数値のみです。

解答例

演習1.1

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4. }
```

演習1.2

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4.     string greeting;
5. }
```

演習1.3

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4.     string greeting;
5.
6.     constructor() public {
7.         greeting = "Hello";
8.     }
9. }
```

演習1.4

```
pragma solidity ^0.5.10;

contract Sample {
    string greeting;

    constructor() public {
        greeting = "Hello";
    }

    function greete() public view returns(string memory) {
        return greeting;
    }
}
```

演習1.5

```
pragma solidity ^0.5.10;

contract Sample {
    string greeting;

    constructor() public {
        greeting = "Hello";
    }

    function changeGreeting(string memory _greeting) public {
        greeting = _greeting;
  }
}
```

演習1.6

```
pragma solidity ^0.5.10;
contract Sample {
    string greeting;
    address owner;

    constructor() public {
        greeting = "Hello";
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(owner == msg.sender);
        _;
    }

    function greete() public view returns(string memory) {
        return greeting;
    }

    function changeGreeting(string memory _greeting) public onlyOwner{
        greeting = _greeting;
    }

}
```

演習2.1

```
pragma solidity ^0.5.10;

```

```
3. contract Sample {
4. }
```

演習2.2

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4.     mapping (address => string) addressToName;
5. }
```

演習2.3

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4.     mapping (address => string) addressToName;
5.
6.     constructor() public {
7.         addressToName[msg.sender] = "Owner";
8.     }
9. }
```

演習2.4

```
1. pragma solidity ^0.5.10;
2.
3. contract Sample {
4.     mapping (address => string) addressToName;
```

```
5.
6.      constructor() public {
7.          addressToName[msg.sender] = "Owner";
8.      }
9.
10.     function setMyName(string memory _name) public {
11.         addressToName[msg.sender] = _name;
12.     }
13.}
```

演習2.5

```
1.  pragma solidity ^0.5.10;
2.
3.  contract Sample {
4.      mapping (address => string) addressToName;
5.
6.      constructor() public {
7.          addressToName[msg.sender] = "Owner";
8.      }
9.
10.     function setMyName(string memory _name) public {
11.         addressToName[msg.sender] = _name;
12.     }
13.
14.     function getName(address _address) public view returns(string memory) {
15.         return addressToName[_address];
16.     }
17.}
```

演習2.6

```
pragma solidity ^0.5.10;

contract Sample {
    mapping (address => string) addressToName;

    constructor() public {
        addressToName[msg.sender] = "Owner";
    }

    function setMyName(string memory _name) public {
        addressToName[msg.sender] = _name;
    }

    function getName(address _address) public view returns(string memory) {
        return addressToName[_address];
    }

    function setName(address _address, string memory _name) public {
        require(msg.sender.balance >= 10 ether);
        addressToName[_address] = _name;
    }
}
```

演習3.1

```
pragma solidity ^0.5.10;

contract Bank {}
```

演習3.2

```
pragma solidity ^0.5.10;

contract Bank {
    mapping(address => uint) addressToAmount;
}
```

演習3.3

```
pragma solidity ^0.5.10;

contract Bank {
    mapping(address => uint) addressToAmount;
}
```

演習3.4

```
pragma solidity ^0.5.10;

contract Bank {
    mapping(address => uint) addressToAmount;

    function deposit() public payable {
        addressToAmount[msg.sender] += msg.value;
    }

    function getMyAmount() public view returns(uint) {
        return addressToAmount[msg.sender];
    }

}
```

演習3.5

```
pragma solidity ^0.5.10;

contract Bank {
    mapping(address => uint) addressToAmount;

    function deposit() public payable {
        addressToAmount[msg.sender] += msg.value;
    }

    function getMyAmount() public view returns(uint) {
        return addressToAmount[msg.sender];
    }

    function withDraw(uint _withDrawValue) public payable{
        require(addressToAmount[msg.sender] >= _withDrawValue);
        msg.sender.transfer(_withDrawValue * 10 ** 27);
        addressToAmount[msg.sender] -= _withDrawValue * 10 ** 27;
    }

}
```

6. 演習①

この章では5章で作成したコントラクトを、Remix上からだけでなく、Webブラウザから操作できるようにしていきます。

6.1 Geth

Geth(Go Ethereum)はEthereumノードを立ち上げるためのEthereumのクライアントソフトです。ブロックのマイニングや、アカウントの作成など**Ethereumノードの管理に必要となる全ての作業を行うことができます**。開発段階でGethを利用することの利点は、Ethereumネットワークにデプロイする前に、ローカル環境でアプリケーションを**実際の環境と同じ状態でテストすることができる**という点があります。

公式サイト　https://geth.Ethereum.org/

6.1.1 プライベートネットワークの構築

この演習では作成したコントラクトをGethを用いて作成したプライベートネットワークにデプロイします。

まずはGethの導入方法から説明します。今回使用するGethのバージョンは「1.9.10-stable-58cf5686」です。

- 巻頭で作成した仮想マシンを立ち上げ、ターミナルを開いてください。
- ターミナルで以下のコマンドを順に実行してください。

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
$ sudo apt-get update
$ sudo apt-get install ethereum
```

以下のコマンドを実行し、GethのバージョンやGethコマンドの種類などが表示されるとGethがインストールされていることが確認できます。

```
$ geth --help
```

プライベートネットワークを作成する

次にGethを用いてEthereumのプライベートネットワークを立ち上げます。

プライベートネットワーク用のフォルダを作成する

プライベートネットワークの作成に必要なファイルや、ネットワーク内の情報を保存するためのファイルを配置するフォルダを作成します。フォルダを作成する場所は任意の場所で構いません。

```
$ mkdir geth
$ cd geth
```

Genesisファイルを作成する

Genesisファイルとはブロックチェーンの始まりであり、0番目のブロックである**Genesisブロックを作成するための情報が記載されたファイル**です。プライベートブロックチェーンを自身で作成するためには、Genesisブロックからブロックが積み重なるため、このファイルが必要になります。今回ファイル名は「genesis.json」にします。Genesisファイルは先ほど作成したフォルダの中に以下の内容で作成してください。

```
{
  "config": {
    "chainId": 10,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0
  },
  "alloc": {},
  "coinbase": "0x0000000000000000000000000000000000000000",
  "difficulty": "0x0000",
  "extraData": "",
  "gasLimit": "0x2fefd8",
  "nonce": "0x0000000000000042",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp": "0x00"
}
```

Gethの起動

以下のコマンドを実行し、GenesisファイルからGenesisブロックを作成します。「--datadir オプション」では、トランザクションやブロックに関して情報の保存場所を指定することができます。

```
$ geth init ./genesis.json --datadir ./
```

次に、以下のコマンドでブロックチェーンをネットワークを作成します。

```
$ geth --networkid "10"  --datadir ./  --allow-insecure-unlock --rpc --rpcaddr "localho
st" --rpcport "8545" --rpccorsdomain "*" console 2>> ./geth.log
```

Gethの起動のために利用したオプションについて説明します。

- --networkid: Ethereumのそれぞれのネットワークは各々の識別子であるネットワークIDを持っています。このネットワークIDを指定し、接続するネットワークを指定するのが、networkidオプションです。今回は、genesis.json内に記述したネットワークIDである10を指定します。
- --datadir: ノードが持つ情報の保存先を指定します。
- --allow-insecure-unlock: Geth外部からアカウントのunlockを可能にします。
- --rpc: GethのHTTP-RPC サーバを有効にします。
- --rpcaddr ： HTTP-RPC サーバのドメインまたはIPアドレスを指定します。初期値として「localhost」が指定されていますが、ここではあえて明記しています。
- --rpcport ： HTTP-RPC サーバがリクエストを受けつけるポート番号を指定します。初期値として「8545」が指定されていますが、ここではあえて明記しています。
- --rpccorsdomain: クロスoriginリクエストを受け付けるドメインを指定します。
- console: Gethのコンソールを立ち上げます。

コマンドを実行し、以下のように表示されると、プライベートネットワークが立ち上がっています。

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.2-stable/darwin-amd64/go1.12.7
at block: 0 (Thu, 01 Jan 1970 09:00:00 JST)
```

```
 datadir: /geth
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0
txpool:1.0 Web3:1.0

>
```

6.1.2 Gethの操作

ここからは作成したプライベートネットワークへコントラクトをデプロイするための準備を行っていきます。

アカウントを作成する

Ethereumではアカウントを持っていなければ、通貨を保有することができません。つまり、マイニングを行うことも、コントラクトを作成することもできません。そのため、まずはアカウントの作成を行います。

まずは、アカウントが1つもないことをアカウントの一覧を取得することで確認します。以下のコマンドを実行してください。

```
> eth.accounts
[]
```

コマンドの実行結果が上のようになり、括弧の中に1つもアカウントが入っていません。これでアカウントが1つも存在しないことを確認できました。

では、アカウントの作成を行います。以下のコマンドを実行してください。パスワードを設定する必要がありますので、任意のパスワードを設定してください。

```
> personal.newAccount()
Password:
Repeat password:
```

```
"0xa03161765df271a5ba21a62e1ad04a13fef25572"
```

これでアカウントを作成することができました。コマンドの実行結果として表示されている文字列がこのアカウントのアドレスになります。再度アカウントの一覧を確認します。

```
> eth.accounts
["0xa03161765df271a5ba21a62e1ad04a13fef25572"]
```

先程は括弧の中身が空でしたが、今回は作成したアカウントのアドレスが確認できます。またアカウントは作られた順にindexが与えられ、今回作ったアカウントは最初のアカウントですので 0 が与えられています。この情報を元に、以下のコマンドを実行すると任意のアカウントのみを取得することができます。ここからさらに、アカウントを作成すればindexは1,2,3...と増えていきます。

```
> eth.accounts[0]
"0xa03161765df271a5ba21a62e1ad04a13fef25572"
```

次に、このアカウントの通貨の保有量を確認します。以下のコマンドを実行してください。

```
> eth.getBalance(eth.accounts[0])
0
```

結果として0が表示されています。単位はweiです。つまり、このアカウントはEtherを保有していないということになります。このままではコントラクトをデプロイするための手数料を支払うことができませんので、通貨を入手する必要があります。

マイニングを行う

現在、Gethを使って立ち上げたプライベートネットワークには参加者が自身が操作しているノード以外には存在しません。つまり、自身でマイニングを行わなければ、誰もブロックを作ってくれず、トランザクションは処理されません。また、マイニングを行うことで報酬として通貨を得ることもできます。

以下のコマンドを実行し、マイニングを始めます。

```
> miner.start()
null
```

これで自身が作成したノードがマイニングを始めました。ノードがマイニングを行っているかどうかは、以下のコマンドで確認することができます。

```
> eth.mining
true
```

trueと返ってくると、マイニングが行われています。マイニングを止める際には以下のコマンドを実行してください。

```
> miner.stop()
null
```

今回作成したプライベートネットワークには参加者が自身が立ち上げたノードしか存在していません。そのため自身がマイニングを止めてしまうと発行したトランザクションが処理されません。そのため、**常にマイニングさせた状態**にしておくか、または**トランザクションを発行した後には、必ずマイニングを行う**必要があります。

ブロック高の確認

マイニングが行われることで、ブロックが作成されます。以下のコマンドでブロック高を取得することができるため、ブロックが作成されていることを確認することができます。環境によってはマイニングを始めてから最初のブロックができるまでに、数十分程度かかることがあります。

```
> eth.blockNumber
320
```

現在、プライベートネットワークでは320個のブロックが作成されていることがわかります。

また、マイニングを行いブロックを作成すると、マイニング報酬を得ることができます。以下のコマンドで、アカウントの通貨の残高を確認します。

```
> eth.getBalance(eth.accounts[0])
1.6e+21
```

アカウントが保有する通貨が増えていることが確認できました。

送金を行う

コントラクトのデプロイには直接は関係ありませんが、GethからEtherの送金を行う方法について説明します。送金を行うためには2つ以上のアカウントが必要になりますので、もう1つアカウントを作成してください。手順は1つ目のアカウントを作った時と同じです。

送金を行う前には、アカウントをunlockする必要があります。以下のコマンドを実行してください。コマンドを実行するとパスワードが要求されるので、アカウントの作成の際に設定したパスワードを入力してください。

```
> personal.unlockAccount(eth.accounts[0])
Password:
true
```

次に送金を行うために以下のコマンドを実行します。今回は初めに作成して、マイニングを行い通貨を保有しているアカウントから、2つ目に作成したアカウントに向けて5etherの送金を行います。コマンド内で送金額を指定する「value」の項目には「wei」で送金額を指定する必要があります。今回はEther単位の送金ですので、Etherをweiに変換するために、「Web3.toWei(5, "Ether")」を利用しています。

```
>eth.sendTransaction({from: eth.accounts[0], to: eth.accounts[1], value: Web3.toWei(5, "Ether")})

"0x0c151702a9997b3cfa1bdb7d59b001f3523d5fef3905f6efa1bc2a5f48f09f67"
```

送金の実行結果として、送金を行うために発行されたトランザクションのIDが返ってきます。実際に、どのようなトランザクションが発行されたのか確認します。以下のコマンドでトランザクションの詳細を取得します。ダブルクオテーションの中身には自身のターミナルに出力されたトランザクショIDを記述してください。

```
>eth.getTransaction("0x0c151702a9997b3cfa1bdb7d59b001f3523d5fef3905f6efa1bc2a5f48f09f67")
  {
  blockHash: "0xea1f0369ba95ff9ffc36a87bd11274d85cbcd9a937588c0423cd1ca138780b0c",
  blockNumber: 321,
  from: "0xa03161765df271a5ba21a62e1ad04a13fef25572",
  gas: 21000,
  gasPrice: 1000000000,
  hash: "0x0c151702a9997b3cfa1bdb7d59b001f3523d5fef3905f6efa1bc2a5f48f09f67",
  input: "0x",
  nonce: 0,
```

```
  r: "0x4bd3e02658fdc31d028a2759dfca1ad65cc267ad5f8ae1ad64188a3ffa95c729",
  s: "0x4c72ff5d7505aa2da7042d1f78129cd7c80d39f41b511f11f1f432f9bd2fd111",
  to: "0xa3802ea2da3bfcba78b6a69af11931882b59e106",
  transactionIndex: 0,
  v: "0x65",
  value: 5000000000000000000
}
```

トランザウション内のfromの項目に送金元のアドレス、toの項目に送金先のアドレス、valueの項目に送金額が入っていることが確認できます。

次に、このトランザクションが含まれているブロックを確認します。トランザクションの項目にblockNumberがあります。これはこのトランザクションが含まれるブロック高です。この情報を元にブロックの詳細を確認します。以下のコマンドを実行してください。コマンドの引数にはトランザクションの詳細で確認した、ブロック高を入れてください。

```
> eth.getBlock(321)
{
  difficulty: 145663,
  extraData: "0xd983010902846765746888676f312e31322e378664617277696e",
  gasLimit: 98085330,
  gasUsed: 21000,
  hash: "0xea1f0369ba95ff9ffc36a87bd11274d85cbcd9a937588c0423cd1ca138780b0c",
  logsBloom: "0",
  miner: "0xa03161765df271a5ba21a62e1ad04a13fef25572",
  mixHash: "0xbaa6186cb31ae484a5d126ff393bd602e1202bd93ca94520eb8c25af59cfffd8",
  nonce: "0x008c9cd08b5b9529",
  number: 321,
  parentHash: "0xe03e48ffe70d3ff935ce2cab5c364e1ce875dda7a9e693f2a398fe7102044f57",
  receiptsRoot: "0x04709d145bbbee58330bd77e745b8d0fb7872b3a10d6198b04d806f233b13a27",
  sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  size: 652,
  stateRoot: "0x6127c097e6b9837c3e5c2e6e2e81c3e606afde5bc66c10f1250e7a54a624d06a",
```

```
  timestamp: 1570153851,
  totalDifficulty: 45505191,
  transactions: ["0x0c151702a9997b3cfa1bdb7d59b001f3523d5fef3905f6efa1bc2a5f48f09f67"],
  transactionsRoot: "0x99fa034f702b0a5bba029fbaf3ff1f05a6fa58e50ecb3c8ba0123d4633ad4ccd
",
  uncles: []
}
```

以上で、Gethの操作方法についての説明を終えます。今回紹介した以外にも様々なコマンドがありますので、公式サイトで確認してみてください。

6.2 コントラクトの作成

6.2.1 コントラクトの作成

今回は文字列を登録し、その文字列の参照と変更を行うことができるコントラクトを作成し、プライベートブロックチェーン上で利用できるようにします。

任意の場所にフォルダを作成し、Hello.solという名前でファイルを作成してください。コントラクトのコードは以下になります。

```
pragma solidity ^0.5.10;

contract Hello {

 string message;

  constructor() public {
     message = "Hello";
  }

   //message変数の値を取得する関数
   function getMessage() public view returns(string memory) {
       return message;
   }

   //message変数の値を変更する関数
   function changeMessage(string memory newMessage) public {
       message = newMessage;
   }
}
```

6.2.2 コントラクトのデプロイ

作成したコードをコンパイルして、デプロイするためにはSolidityのコンパイラである、Solcをインストールする必要があります。今回利用するSolcのバージョンは「0.5.14」です。入手できたsolcのバージョンが異なりましたら、Solidityファイルの一行目のSolidityのバージョンを書き換えてください。

Solcのインストール方法

- 巻頭で作成した仮想マシンを立ち上げ、ターミナルを開く。
- 以下のコマンドを実行する。

```
$ sudo apt install solc
```

Solidityコードのコンパイル

作成したコードをコンパイルします。新しくターミナルを立ち上げて、以下のコマンドを実行してください。コマンドはSolidityのコードを作成したディレクトリ内で実行してください。

```
$ solc --bin --abi Hello.sol
```

実行すると、「Binary」と「Contract JSON ABI」という2つの項目が表示されます。BinaryはEVMバイトコードのことであり、ABI(Application Binary Interface)はコントラクトのインターフェイス情報です。

ブロックチェーンへのデプロイ

コンパイル時に結果として出力したBinaryとABIを使って、プライベートブロックチェーン上にHelloコントラクトのデプロイを行います。

ここからは再び、Gethのコンソールでの操作に戻ります。以下のコマンドを順に実行してください。

```
// コンパイル結果に出力されたBinaryの先頭に0xをつけたものをbinに代入する
> bin = “0x…….”

// コンパイル結果に出力されたContract JSON ABIを代入する
> abi = []

> contract = eth.contract(abi)
> Hello = contract.new({ from: eth.accounts[0], data: bin, gas: 1000000 })
```

上記の4つ目のコマンドで、コントラクトのデプロイが完了します。

6.3 フロントエンドの作成

6.3.1 Web3

Web3とは、Ethereumネットワーク内のノードと通信するため仕様です。このWeb3の仕様に則り作成されたWeb3.jsはEthereumのノードと通信するJavaScript APIライブラリです。Web3.jsを活用することで、デプロイされたスマートコントラクトへブラウザからのアクセスが可能になります。今回作成するDAppsではWeb3.jsを利用して、ブラウザからEthereumネットワーク内のノードと通信することで、コントラクトをブラウザから操作するDAppsを作成します。

今回、Ethereumのノードと通信するためにWeb3.jsを利用しますが、これだけでなく、Rubyで作られたEthereum.rbや、Pythonで作られたWeb3.pyなど様々種類があります。

6.3.2 コードの作成

作成したコントラクトをWebブラウザから操作することができるようにフロントエンドを作成します。フロントエンドのファイル用にフォルダを作成し、以下の2つのファイルをフォルダの直下に作成してください。

index.html

```
<!doctype html>
<html>
<head>
    <title>Blockchain Hello World</title>
    <meta charset="utf-8" />
</head>

<body style="margin: 0 auto; text-align: center; width: 80%; ">
    <header style="text-align: center; margin-top: 20px; font-weight: bold; font-size: 30px">
        Blockchain App
    </header>
    <div style="margin-top: 20px">
        <div id="message" style="font-size: 20px"></div>
        <button onclick="getMessage()" style="margin-top: 10px">Refresh message</button>
    </div>
    <hr>
    <div style="margin-top: 20px">
        <input id="value" type="text" style="width:200px"><br>
        <button onclick="changeMessage()" style="margin-top: 10px">Update message</button>
```

```
    </div>
    <script src="./main.js"></script>
</body>

</html>
```

main.js

```
let abi;
var Web3;

let ContractAddress = "";
var ContractInstance;

function initApp() {
  let myContract = Web3.eth.contract(abi);
  ContractInstance = myContract.at(ContractAddress);
}

window.getMessage = () => {
  ContractInstance.getMessage((err, result) => {
    if (!err) {
      document.getElementById("message").innerText = result;
      console.log(result);
    } else {
      console.log(err);
    }
  });
};

window.changeMessage = () => {
  let value = document.getElementById("value").value;
  ContractInstance.changeMessage(value, (err, result) =>{
    if (!err) {
```

```
27.       console.log(result);
28.     } else {
29.       console.log(err);
30.     }
31.   });
32.};
33.
34.window.addEventListener("load", () => {
35.   if (typeof window.ethereum !== "undefined" || typeof window.Web3 !== "undefine
    d") {
36.     let provider = window["ethereum"] || window.Web3.currentProvider;
37.     Web3 = new Web3(provider);
38.     ethereum.enable();
39.   } else {
40.     console.log("Metamaskが認識されません");
41.   }
42.
43.   initApp();
44.});
```

main.jsの1行目のabiにはコントラクトのデプロイの際に利用したabiを、3行目のsmartContractAddressにはコントラクトをデプロイした際に表示されたコントラクトのアドレスを代入してください。

main.jsに記述されているコードについて説明します。ブラウザでmain.jsファイルが読み込まれたタイミングで、33行目から始まるブロックの処理が行われます。ここでは、主にブラウザからブラウザの拡張機能であるMetaMaskを接続するための設定を行っています。Ethereumネットワーク内のノードと通信するためのライブラリWeb3.jsもMetaMaskがあらかじめ持っているものを利用します。42行目のinitApp関数(関数の処理が書いてあるのは7行目から10行目)では、今回利用するコントラクトのアドレスとabiから、指定したコントラクトに対してContractInstance変数を通じてアクセスできるようにしています。

実際にコントラクトを実行している部分は、それぞれ以下の部分になります。

- 13行目：コントラクトのgetMessage関数を呼び出しています。

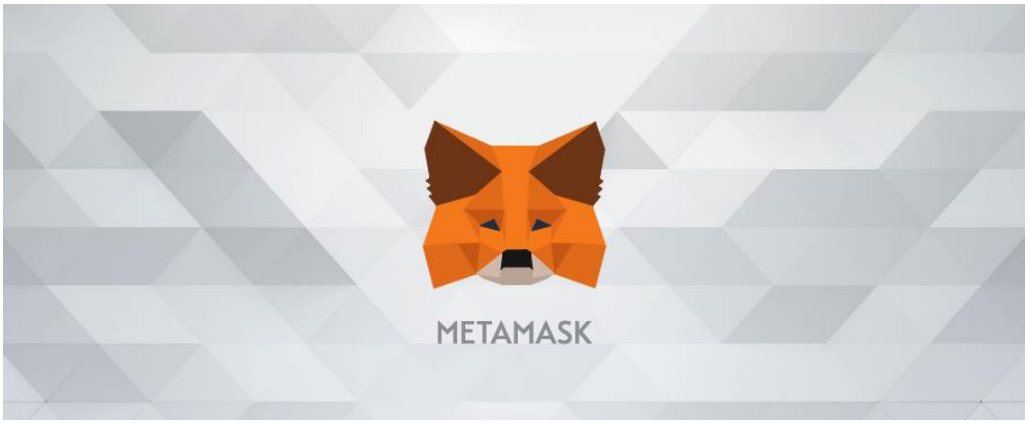
```
ContractInstance.getMessage((err, result) =>  {};
```

- 25行目：コントラクトのchangeMessage関数を呼び出しています。

```
ContractInstance.changeMessage(value, (err, result) => {};
```

これら以外の部分に関しては、index.html、main.js共にEthereumを利用する上で特別な記述は必要ありません。

6.4 MetaMask



MetaMaskとは、Google Chromeのプラグインとして使うことができるEthereumウォレットです。通常のブラウザでは、仮想通貨の管理や、Ethereumネットワーク内のノードと通信を行うことはできませんが、MetaMaskを用いることでEthereumネットワーク上のノードと通信し、Etherの管理から、送金やコントラクトの実行のためのトランザクションの発行まで行うことができます。またMetaMaskをブラウザのプラグインとして導入すると、Web3.jsがブラウザから利用できるようになります。

6.4.1 MetaMaskの導入

MetaMaskの導入方法について説明します。利用するブラウザはGoogleChromeを想定して説明を行います。

https://chrome.google.com/Webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=ja を開き、右上のChromeに追加を選択します。

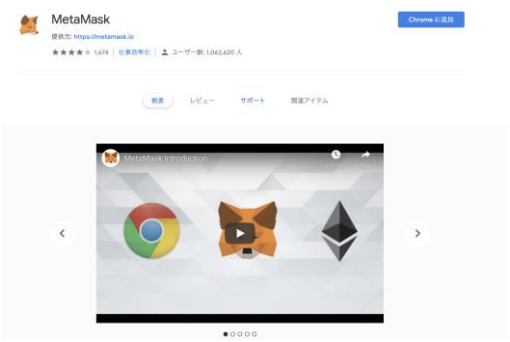


図6.1 Metamaskの導入①

中央の「Get Started」を選択します。

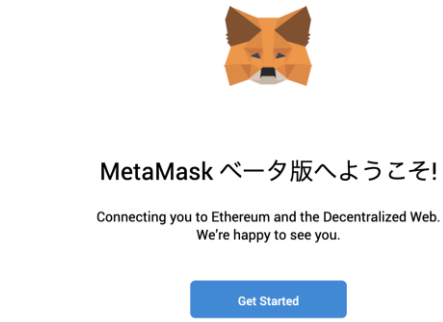


図6.2 MetaMaskの挿入②

右側の「Create Wallet」を選択します。

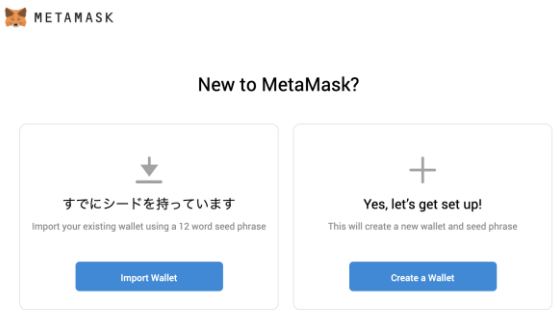


図6.3 MetaMaskの導入③

右側の「I agree」を選択します。

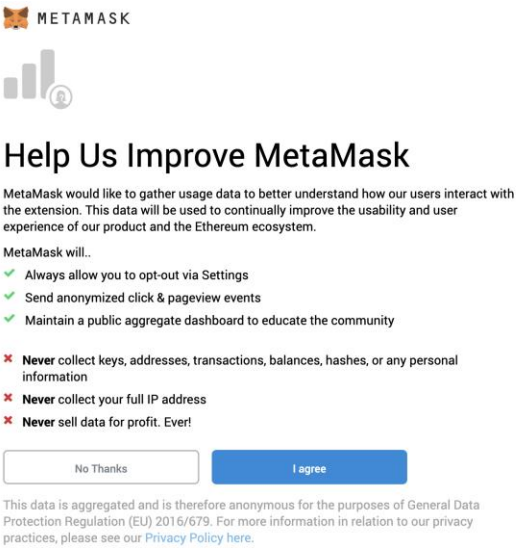


図6.4 MetaMaskの導入④

パスワードを入力し、作成を選択してください。

図6.5 MetaMaskの導入⑤

ウォレットのバックアップのためのバックアップフレーズが表示されます。表示される文字列のメモをノートなどに取ってください。このバックアップフレーズを利用することで、MetaMask以外のウォレットでも今回作成したウォレットを再現することができます。

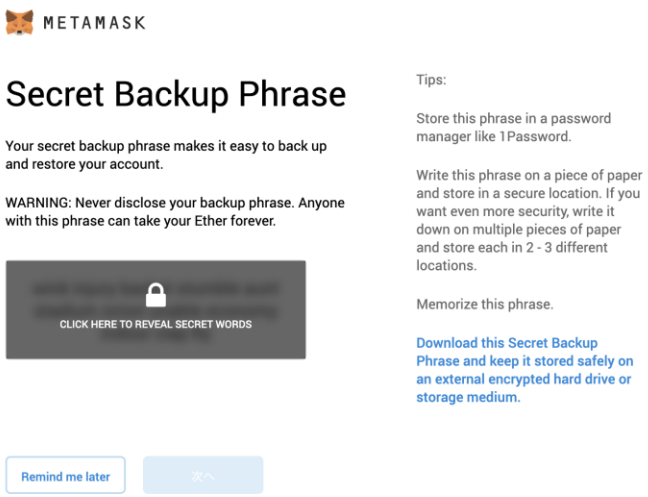


図6.6 MetaMaskの導入⑥

メモをとったバックアップフレーズを順に入力してください。

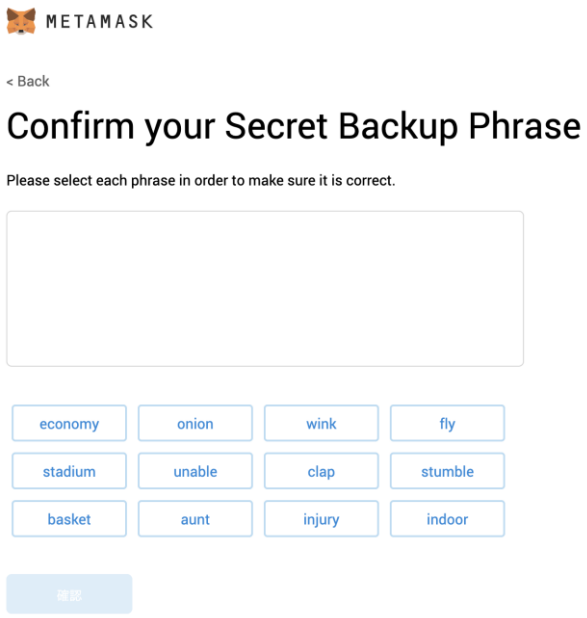


図6.7 MetaMaskの導入⑦

以上で、MetaMaskの設定は終了になります。全て完了を選択してください。

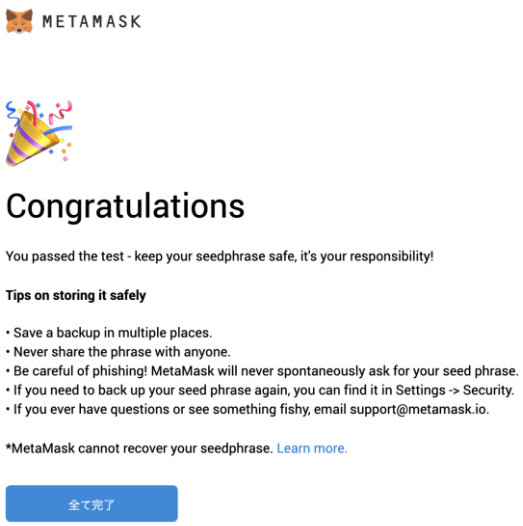


図6.8 MetaMaskの導入⑧

全ての設定が終了し、以下のような画面が開きMetaMaskを利用することができるようになります。

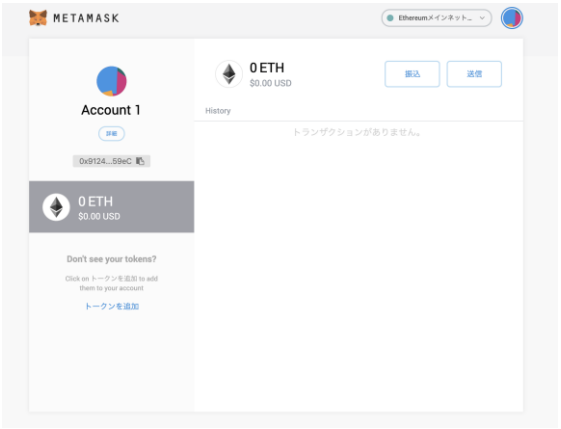


図6.9　MetaMaskの導入⑨

6.4.2 MetaMaskの利用

ここからはブラウザからMetaMaskを通じてEthereumのプライベートネットワークにアクセスするためにMetaMaskの設定を行います。

MetaMaskの画面右上の「Ethereumメインネットワーク」と表示されている部分をクリックし、「カスタムRPC」を選択します。

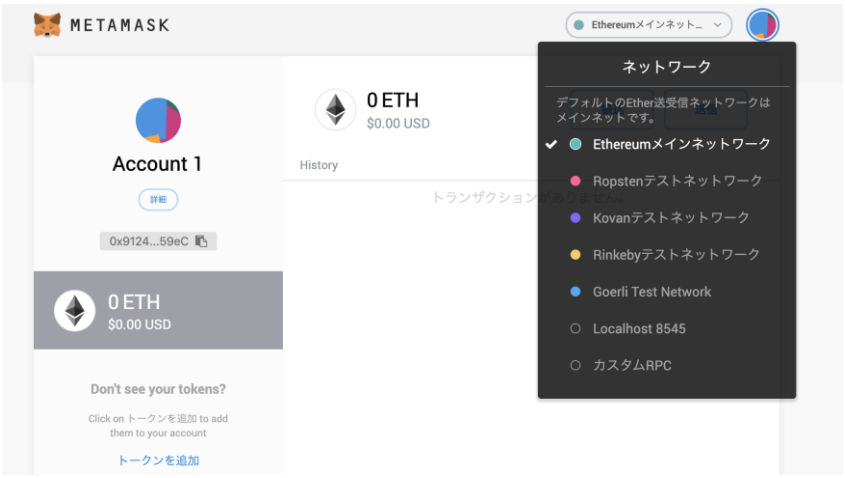


図6.10 MetaMaskの設定①

カスタムRPCを選択すると、以下のような設定画面が開きます。

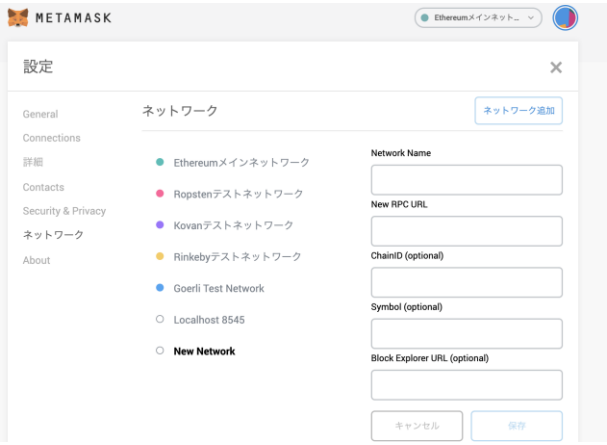


図6.11 MetaMaskの設定②

New RPC URLの部分に「http://localhost:9545」 と入力し、右下の保存ボタンをクリックしてください。このURLは先程プライベートネットワークを作成した際に立ち上げたノードのURLです。Gethでプライベートネットワークを起動させた際には、設定を行わない限りこのURLが初期値となります。

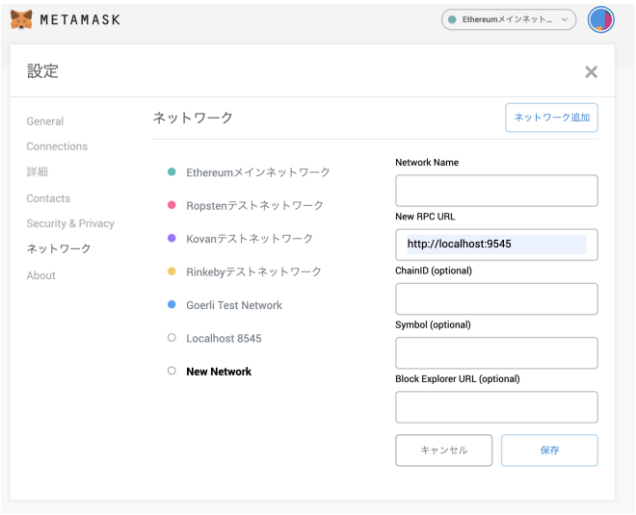


図6.12 MetaMaskの設定③

以上でGethを利用して作成したプライベートネットワークとMetaMaskの接続が完了します。画面右上のEthereumメインネットワークと表示されていた部分に、「http://localhost:9545」と表示されていれば問題ありません。

次に、Gethで作成したアカウントの情報をMetaMaskに反映します。ここで利用するアカウントは、Gethでノードを立ち上げた際に最初に作り、マイニングを行なって通貨を保有しているアカウントです。

Metamaskの画面の右上の丸いボタンをクリックし、アカウントのインポートを選択します。

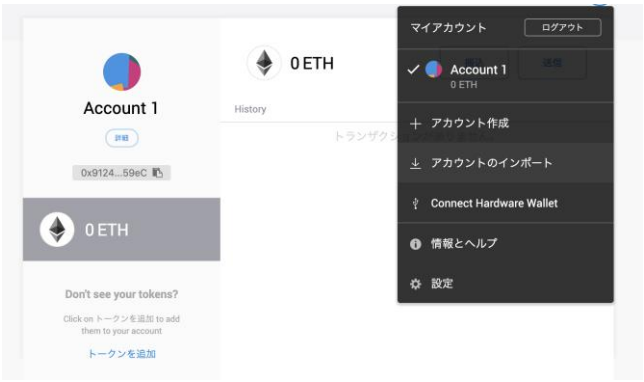


図6.13 アカウントのインポート①

次に、キーの種類の選択項目で、JSONファイルを選択します。

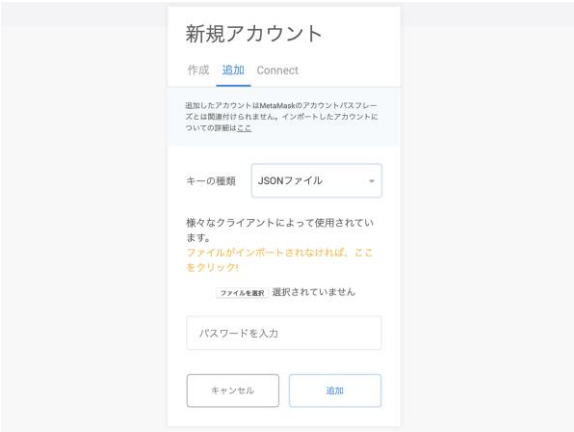


図6.14 アカウントのインポート②

インポートするファイルはGethを起動する際に作成したフォルダ内のkeystoreフォルダ内に、それぞれのアカウントに関連する情報が記述されたJSON形式のファイルがありますので、そのファイルを選択してください。

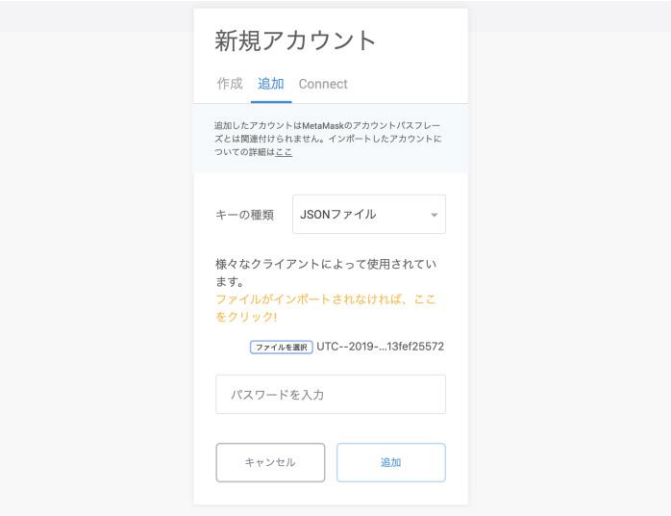


図6.15 アカウントのインポート③

ファイルを選択し、追加ボタンをクリックすると、アカウントがインポートされ、Gethで作成したアカウントがMetMaskに反映されます。Gethで作成したアカウントのアドレスと、MetaMaskに表示されているアカウントのアドレスが同じものであれば、問題なくアカウントがインポートされています。

DAppsを利用する

ここまでで、DAppsを利用する準備ができましたので、実際にブラウザで利用します。そのために、index.htmlとmain.jsを置いておくWebサーバを立ち上げる必要があります。ここで利用するWebサーバはなんでも構いませんが、今回は例として、簡易的なWebサーバを立ち上げることのできるアプリケーションを利用します。

https://chrome.google.com/Webstore/detail/Web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb をGoogleChromeで開く。

図6.16 Webサーバの立ち上げ方①

以下のような画面が開き、CHOOSE FOLDERでindex.htmlが入っているフォルダを選択し、GoogleChromeで「http://127.0.0.1:8887」を開く。

図6.17 Webサーバの立ち上げ方②

そうすると、以下のような画面が開きます。ここからは実際にDAppsを使っていきます。

画面中央のRefresh messageボタンをクリックしてください。

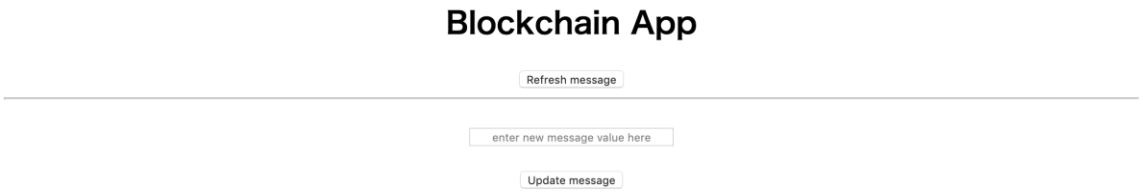


図6.18 DAppsの画面①

クリックすると、以下のようにコントラクト内で変数に与えたHelloという文字列が表示されます。

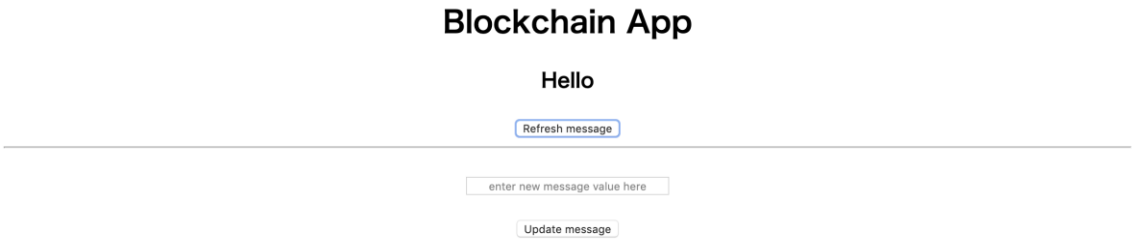


図6.19 DAppsの画面②

次に、文字列の変更を行います。下部のテキストボックスに好きな文字列を入れて、Update Messageボタンをクリックしてください。そうすると、以下のようなMetaMaskの画面が開きます。これは、Metamaskがコントラクトを実行するためのトランザクションを発行することに対しての確認画面です。確認ボタンをクリックしてください。

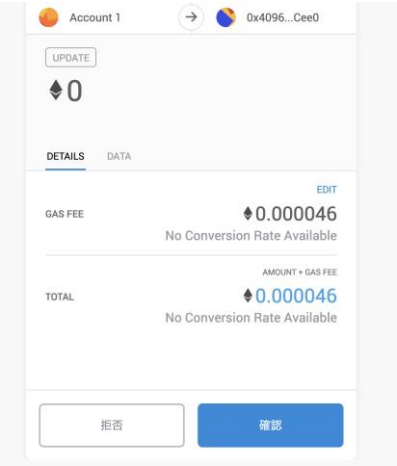


図6.20 DAppsの画面③

その後、再度Refresh messageボタンをクリックすると、先程変更した新しいメッセージに変わっていることが確認できます。

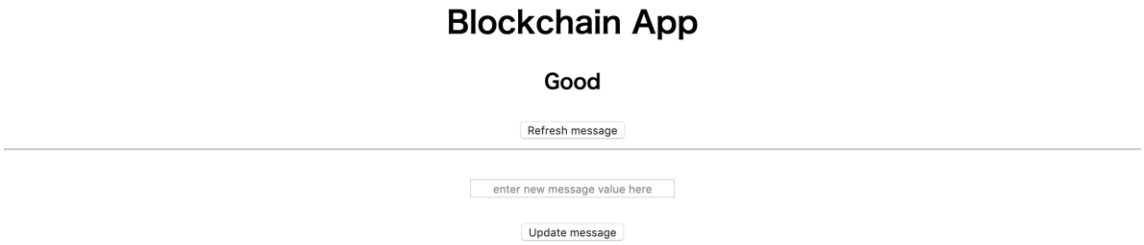


図6.21 DAppsの画面④

以上で、Gethを用いたプライベートネットワークの作成と、ブラウザから利用することのできるDAppsの作成方法についての説明を終えます。

7.演習②

この章では、DApps作成のためのフレームワークを用いて、ブラウザから操作することのできるDAppsを作成します。

7.1 Truffle

7.1.1 Truffleとは

TruffleはEtheruem上でDAppsを開発するためのフレームワークです。ここからはTruffleを用いてDAppsの開発を行います。

公式サイト　https://www.trufflesuite.com

環境構築

Truffleを利用するための環境構築を行います。以下のコマンドを順に実行してください。使用するバージョンは以下になります。

- Truffle: 5.1.12
- npm: 3.5.2
- node: 8.10.0

```
$ sudo apt-get install -y nodejs npm
$ sudo npm install -g truffle
```

7.1.2 コントラクトの作成

まずはプロジェクトを作成するためのフォルダを作ります。ここでは、名前はMyDAppsにします。

```
$ mkdir MyDApps
```

フォルダを作成すると、そのフォルダの中に入り、以下のコマンドを実行してください。

```
$ cd MyDApps
$ truffle init
```

このコマンドにより、フォルダ内にDAppsの作成に必要なフォルダやファイルが作成されます。

ここからはコントラクトの作成を行います。プロジェクト内のcontractsフォルダの中にHello.solという名前のファイルを作成し、以下の内容を書き込んでください。TruffleでDAppsを作成する際には、contractsフォルダ内にコントラクトを記述したSolidityのファイルを作成します。

```
1.  pragma solidity ^0.5.10;
2.
3.  contract Hello {
4.
5.      string message;
6.
7.      constructor(string memory initialMessage) public {
8.          message = initialMessage;
9.      }
10.
11.     //message変数の値を取得する関数
12.     function getMessage() public view returns(string memory) {
13.         return message;
14.     }
15.
```

```
    //message変数の値を変更する関数
    function changeMessage(string memory newMessage) public {
        message = newMessage;
    }
}
```

次に作成したコントラクトのコンパイルを行います。以下のコマンドを実行してください。

```
$ truffle compile

Compiling your contracts...
===========================
> Compiling ./contracts/Hello.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to MyDApps/build/contracts
> Compiled successfully using:
   - solc: 0.5.10+commit.1d4f565a.Emscripten.clang
```

これにより、作成したコードのコンパイルが終了します。

7.1.3 コントラクトのデプロイ

Truffleではマイグレーションファイルを用いて、コントラクトのデプロイを行います。Hello.solをデプロイするためのマイグレーションフォルダを作成するために以下のコマンドを実行してください。

```
$ truffle create migration DeployHello
```

このコマンドにより、プロジェクトの直下のmigrationsフォルダの中に名前がdeploy_hello.jsで終わるマイグレーションファイルが作成されます。次にmigrationファイルの中身の書き換えを行います。以下のように中身を書き換えてください。

```
const HelloContract = artifacts.require('Hello.sol');

module.exports = function(deployer) {
  deployer.deploy(HelloContract, 'Hello');
};
```

コントラクトのコンストラクタに引数を渡す必要がある場合には、migrationファイルに記述します。例では、以下の部分で初期値Helloを渡しています。

```
deployer.deploy(HelloContract, 'Hello');
```

これにより、作成したコントラクトをブロックチェーンにデプロイする準備が整いました。

今回はまずTruffleが提供するローカル環境で完結するBlockchainのシミュレータに対して、コントラクトをデプロイします。

コントラクトをデプロイするために、以下のコマンドを実行し、Truffleのコンソールに入ってください。これにより、TruffleがEthereumのプライベートネットワークを立ち上げてくれます。これ以降の作業はこのコンソールを閉じずに作業を行なってください。

```
$ truffle develop

Truffle Develop started at http://127.0.0.1:9545/

Accounts:
```

```
(0) 0x142391b6ee6e8d973b69ad90f44477634d05427f
(1) 0x1548ce4677bb26927ed67444a036324a9a93171d
(2) 0x6db5c17ffdb477d5e9b2eb669574679ce16cbd3e
(3) 0x76a28e9722da27cee9fa53a0a75baf9b750d74a7

// 秘密鍵は後ほど必要になるので、始めの１つをメモしておく
Private Keys:
(0) dee7c86854da45dba3173e2a9bb8e9bd00d340492066259c871dfb5d8ed4b65f
(1) 70ba736b21e9d6a280df2220e7b61f33688eaee9ef71064cdad6713ec4b645fd
(2) b1982758a28160c3afc213402153cd5a62f66224eed60849562cdc9da2f011af
(3) 8495fcc93d035151d1a609c495ff0d5f85a7293743ccf3362cc858cc593da164

Mnemonic: task empty liar there muscle armed random neutral neither outside trap grab

⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.
```

次にTruffleコンソール内で以下のコマンドを実行し、作成したマイグレーションファイルを元にブロックチェーンにコントラクトをデプロイします。

```
> migrate

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.

Starting migrations...
======================
> Network name:    'develop'
> Network id:      5777
> Block gas limit: 0x6691b7

1_initial_migration.js
======================
```

```
   Replacing 'Migrations'
   ----------------------
   > transaction hash:    0x574b4c2c8d55d80ec69a95820c0b5c883211265304811e85f5ff1f1625630daa
   > Blocks: 0            Seconds: 0
   > contract address:    0x095F743DA718155926fbe4eA8027Ae873849C8A1
   > block number:        1
   > block timestamp:     1569833529
   > account:             0x142391B6Ee6e8d973b69AD90f44477634D05427f
   > balance:             99.99430184
   > gas used:            284908
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00569816 ETH

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00569816 ETH

1549262571_deploy_hello.js  // 作成したmigrationファイルを元にデプロイされる
=========================
   Replacing 'Hello'
   -----------------
   > transaction hash:    0x3beae15ac5d03bff5d0d945afa4e3fdadecf9766ff227fa2a366deda97061b9f
   > Blocks: 0            Seconds: 0
// コントラクトアドレスは後ほど必要になるのでメモしておく
   > contract address:    0x9cC8f00e54BB2F63662847D06cd20F04cF355B39
   > block number:        3
   > block timestamp:     1569833529
   > account:             0x142391B6Ee6e8d973b69AD90f44477634D05427f
   > balance:             99.9872387
   > gas used:            311123
   > gas price:           20 gwei
```

```
   > value sent:          0 ETH
   > total cost:          0.00622246 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.00622246 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.01192062 ETH
```

以上のような、結果が出力されると、無事にブロックチェーンに対してデプロイが完了したことになります。上記の出力結果にデプロイしたコントラクトのアドレスが表示されます。このアドレスは後ほど利用しますので、メモしておいてください。

これで、コントラクトのデプロイ作業が終了します。

7.2 フロントエンドの作成

ここからは、ブラウザに表示されるフロントエンドの作業に入ります。今回はWebブラウザからブロックチェーン上のコントラクトが利用できるようにします。

7.2.1 コードの作成

DAppsのプロジェクトの直下にsrcフォルダを作成します。この中にHTMLファイルやJavascriptファイルなどを作成します。今回はsrcフォルダの中に以下の2つのファイルを作成してください。

- index.html
- main.js

ファイルの中身はそれぞれ以下のようにしてください。

index.html

```
<!doctype html>
<html>
<body style="text-align: center">
    <div style="font-size: 36px">Blockchain App</div>
    <div style="text-align: center">
        <div id="message" style="font-size:20px">Hello</div>
        <button onclick="getMessage()">Get message</button>
        <hr>
        <input id="value" type="text" placeholder="enter new message " /><br>
        <button onclick="changeMessage()">Change message</button>
    </div>
    <script type="text/javascript" src="main.js"></script>
</body>
</html>
```

main.js

```
let abi;
var Web3;

let ContractAddress = "";
var ContractInstance;

function initApp() {
  let myContract = Web3.eth.contract(abi);
  ContractInstance = myContract.at(ContractAddress);
}

window.getMessage = () => {
  ContractInstance.getMessage((err, result) => {
    if (!err) {
      document.getElementById("message").innerText = result;
      console.log(result);
    } else {
      console.log(err);
    }
  });
};

window.changeMessage = () => {
  let value = document.getElementById("value").value;
  ContractInstance.changeMessage(value, (err, result) =>{
    if (!err) {
      console.log(result);
    } else {
      console.log(err);
    }
  });
};
```

```
33.
34. window.addEventListener("load", () => {
35.   if (typeof window.ethereum !== "undefined" || typeof window.Web3 !== "undefine
    d") {
36.     let provider = window["ethereum"] || window.Web3.currentProvider;
37.     Web3 = new Web3(provider);
38.     ethereum.enable();
39.   } else {
40.     console.log("Metamaskが認識されません");
41.   }
42.
43.   initApp();
44. });
```

main.jsの1行目には、コントラクトのabiを代入します。abiはTruffleで作成したフォルダ内の「build/contracts」フォルダ内の「Hello.json」に記述されています。この「Hello.json」ファイルはHelloコントラクトをコンパイルした際に作成されました。

次に、3行目には利用するコントラクトのアドレスを記述する必要があります。アドレスはTruffleのコンソールでmigrateコマンドを実行した際に、表示されていますので、それを代入してください。

次に、MetaMaskの設定を行います。6章で行なったように、MetaMaskをTruffleで作成したネットワークに接続し、アカウントをインポートします。

MetaMaskとTruffleの接続

MetaMaskとTruffleの設定については、6.4章を参考にしてください。接続先はTruffleのコンソールを立ち上げた際に表示されたURLになり、基本的には「http://127.0.0.1:9545/ 」になります。

MetaMaskへのアカウントのインポート

Truffleでネットワークを立ち上げた際には、自動的にアカウントが10個作成され、それぞれが100etherずつ保有している状態になっています。このアカウントをMetaMaskから利用できるようにします。こちらの作業も6.4章を参考に行い、Truffleのコンソールを立ち上げた際にTruffleのコンソールに表示された秘密鍵を入力してください。

DAppsの利用

以上で、DAppsを利用する準備が整いました。ここで、HTMLファイルとJavascriptファイルを置くための、Webサーバを立ち上げます。手順については6.4章と同様で、DAppsプロジェクト内のsrcディレクトリを指定して、Webサーバを立ち上げてください。

DAppsの構成自体は、6章で作成したものと同じですので、操作してみてください。

7.3 テストネットワークの利用

7.3.1 Ethereumのネットワークの種類

Ethereumには大きく分けて3つの種類のネットワークが存在します。

メインネットワーク

一般的にEthereumのネットワークといった時に指されるのがこのメインネットワークです。**価値を持ったEtherが取引**されています。

テストネットワーク

メインネットワークと同様に世界中に広がるネットワークです。メインネットワークとは、テストネットワーク内で**取引されるEtherは他の通貨に変換することができない**と言う点が異なります。Bitcoinや法定通貨をはじめとして、メインネットワークのEtherとも交換することができません。つまり、基本的にはテストネットワークに存在する通貨は価値を持ちません。さらに、Ethereumのテストネットワークにはいくつか種類があります。

Ropsten

Ethereumのテストネットワークの中でも最初に作成されたネットワークです。メインネットワークのEthereumと同じ振る舞いをします。

Kovan

メインネットワークのEtheruemとほとんど同じように動いていますが、利用されているコンセンサスアルゴリズムが異なります。現在EthereumではProof of Workが採用され、誰でもマイニングすることができます。これに対して、KovanではProof of Authorityというコンセンサスアルゴリズムが採用されています。Proof of Authorityでは誰でもマイニングを行うことができるのではなく、あらかじめ定められたノードがマイニングを請け負う仕組みになっています。テストネットワークを安定して運用するためにこのような方法が採られています。

プライベートネットワーク

メインネットワークとテストネットワークは世界中のコンピュータで構成されているのに対し、プライベートネットワークはEthereumのクライアントソフトを用いることで、誰でも作成することのできるネットワークです。一台のコンピュータで構成してもいいですし、複数のコンピュータを繋げてネットワークを作成することもできます。

7.3.2 テストネットワークの利用

ここからはテストネットワークを利用します。先程はTruffleで作成したネットワークにコントラクトをデプロイしましたが、ここからはテストネットワークの１つであるであるRopstenに対してのデプロイの方法について説明します。

テストネットワーク上の通貨の入手

テストネットワークもメインネットワークと同様に、トランザクションを発行する際には手数料が必要になります。そのため、コントラクトをデプロイする前に通貨を入手する必要があります。方法については以下に示します。

MetaMaskをRopstenに接続する

Metamaskが接続するネットワークをRopstenに変更する

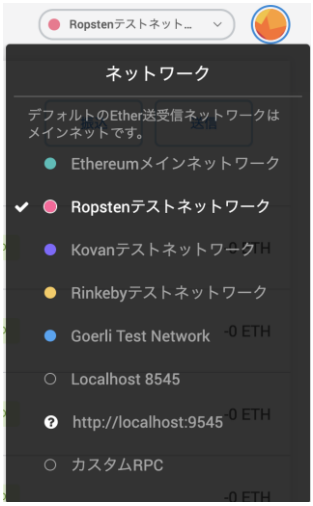


図7.1 Ropstenの利用①

「https://faucet.ropsten.be/」に接続する

テストネットワークの通貨は上記のようなWebサイトで無料配布されています。

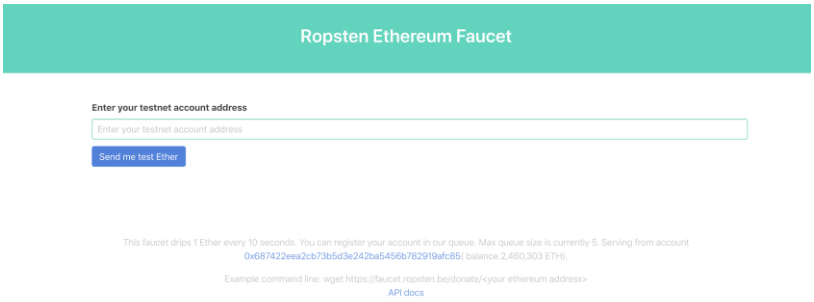


図7.2 Ropstenの利用②

通貨を保存するアドレスを入力する

Metamaskで表示されているアカウントのアドレスをテキストボックスに入力します。その後、「Send me test Ether」をクリックすると、入力したアドレス宛に1.5 ether振り込まれます。MetaMaskで確認してください。

図7.3 Ropstenの利用③

テストネットワーク用の通貨を配布するWebサイトは複数ありますが、そのほとんどのサイトで配布に制限がつけられています。具体的には、「1IPアドレスあたり、24時間に1回のみ」「1Ether以上持っているアドレスには配布しない」などがあります。今回利用した配布サイトでは、「1IPアドレスあたり24時間に1回のみ」かつ、「1アドレスあたり24時間に1回のみ」という条件があるので、注意が必要になります。

INFURAへの登録

ここからはTruffleからコントラクトをRopstenにデプロイするための準備を行います。ここで問題になるのが、「どこへデプロイするのか?」ということです。Ethereumのネットワークへは誰でも参加することができ、逆にいつでもネットワークから抜けることができます。このため、現在ネットワークに参加しているノードを探し、そのノードに対してデプロイする必要があります。しかし、この情報は簡単に取得できるものではありません。そのため、この部分を簡約化するために今回はINFURAというサービスを使用します。INFURAはEtheruemのメインネットワークやテストネットワークの現在稼働しているノードの情報を保有しており、INFURAから指定されたURLにアクセスすることで、INFURAを経由してコントラクトをテストネットワークにデプロイすることができます。

もちろん、自身でGethなどのクライアントソフトを立ち上げて、メインネットワークやテストネットワークに直接コントラクトをデプロイすることも可能ですが、ネットワークの同期に時間とPCリソースを費やす必要があります。簡単にコントラクトをデプロイしたい場合にはINFURAを利用することをおすすめします。

- INFURA(https://infura.io/dashboard)にアクセスし、トップページの「Get Started FOR FREE」をクリックします。
- 続いて表示される画面で必要事項を入力し、「SIGN UP」をクリックします。
- 登録したメールアドレスにINFURAから確認メールが届くので、メール内の「CONFIRM EMAIL ADDRESS」をクリックすると、ユーザー登録が完了します。
- 登録が完了すると、INFURAのダッシュボード画面に遷移します。
- ダッシュボード画面で「CREATE NEW PROJECT」をクリックし、任意の名前でプロジェクトを作成します。
- プロジェクトを作成すると、PROJECT IDとPROJECT SECRET, ENDPOINTが発行されます。

以上でINFURAの設定が終わります。

Truffleの設定ファイルの変更

次にTruffleでテストネットワークに接続するように、truffle-config.jsの設定を変更します。以下の部分を変更してください。

```
//  以下の部分のコメントアウトを外してください
// infuraKeyの部分にはINFURAのプロジェクトIDを記述してください
const HDWalletProvider = require('truffle-hdwallet-provider');
const infuraKey = "";

const fs = require('fs');
const mnemonic = fs.readFileSync(".secret").toString().trim();

//  以下の部分のコメントアウトを外してください
ropsten: {
      provider: () => new HDWalletProvider(mnemonic, `https://ropsten.infura.io/${infur
aKey}`),
      network_id: 3,       // Ropsten's id
      gas: 5500000,        // Ropsten has a lower block limit than mainnet
      confirmations: 2,    // # of confs to wait between deployments. (default: 0)
      timeoutBlocks: 200,  // # of blocks before a deployment times out  (minimum/defau
lt: 50)
      skipDryRun: true     // Skip dry run before migrations? (default: false for publi
c nets )
    },
```

更に、プロジェクトの直下に「.secret」という名前のファイルを作り、MetaMaskのパスフレーズをMetaMaskの「Expand View」→「設定」→「Security&Privacy」からコピーし、ファイルにペーストしてください。

これでRopstenに接続する準備ができましたので、Truffleを使ってRopstenに接続します。以下のコマンドを順に実行してください。

```
$ sudo apt-get install git
$ npm install truffle-hdwallet-provider
$ truffle console --network ropsten
truffle(ropsten)>
```

次に、Ropstenに対して作成したコントラクトをデプロイします。以下のコマンドを実行してください。

```
truffle(ropsten)> migrate
```

コントラクトが無事にEthereumのRopstenテストネットワークにデプロイされたかどうかは、Etherscan(https://Etherscan.io/)の右上から表示するネットワークをRopstenとして確認してください。

次にフロント側のファイルの操作に移ります。コマンドの実行結果にデプロイしたコントラクトのアドレスが表示されるので、そのアドレスをmain.jsのsmartContractAddress変数に記述してください。その後、6章と同様にWebサーバを立てて、index.htmlを開きます。これで、テストネットワーク上にあるコントラクトをWebブラウザから利用できるようになります。

以上で、DApps作成のためのフレームワークであるTruffleを用いたDAppsの作成方法と、テストネットワークの利用方法についての説明を終えます。

演習1

テストネットワークは、世界中の誰でもアクセスすることができるネットワークであり、その上にあるコントラクトは誰でも利用することができます。そこで、この演習ではRemixからテストネットワーク上に自身のオリジナルのトークンを発行し、そのトークンを相互にやりとりします。

ERCトークン

Ethereum上では自身で独自のトークンを発行することができます。ここではオリジナルのトークンをERCという仕様に従って発行する方法について説明します。

ERC

ERCとはEthereum Request for Commentsの略であり、Ethereumのアプリケーションレベルの標準化や企画が定義されています。ERCの中の1つにERC20 Token Standardがあります。これは2015年に提案された最初のERC規格であり、Etheruem上で発行されるトークンの標準的なインターフェイスを定義する規格です。ERC20が登場するまでは、新しいトークンが発行されるたびに、通貨を管理するウォレットに、その通貨に対応するための実装を追加する必要がありました。ERC20が登場した以後は、同じインターフェイスでトークンを扱うことができるようになりました。また、新しいトークンの発行も簡単に行うことができるようになりました。

では、ここからはERC20を利用して独自のトークンの発行を行います。以下にトークンを発行するためのコードを示します。Ethereum上でのコントラクト開発の統合開発環境であるRemixで新たにファイルを作成し、以下のコードを記述してください。

```
1. pragma solidity ^0.5.10;
2.
3. import "https://github.com/OpenZeppelin/openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
4. import "https://github.com/OpenZeppelin/openzeppelin-solidity/contracts/token/ER
```

```
    C20/ERC20Detailed.sol";
5.
6.  contract SampleToken is ERC20, ERC20Detailed {
7.      string private _name = "SampleToken";
8.      string private _symbol = "SPT";
9.      uint8 private _decimals = 18;
10.
11.     address account = msg.sender;
12.     uint private value = 10 * (10 **18);
13.
14.     constructor() ERC20Detailed(_name, _symbol, _decimals) public {
15.         _mint(account, value);
16.     }
17.}
```

3行目と4行目は改行されていますが、一行で記述してください。ここで、外部からERCトークンの発行に必要なファイルを取得しています。これらのコードはOpenZeppelin(https://github.com/OpenZeppelin/openzeppelin-contracts)で確認することができます。OpenZeppelinとはEtehreum上で安全なコードを書くためのライブラリです。

6行目でcontractの宣言をしています。今回作成するSampleTokenコントラクトは3行目と4行目でインポートした2つのコントラクトを継承したコントラクトになります。

7行目から12行目で状態変数を作成しています。いずれも、ERC20でトークンを発行する際に必要な項目です。以下にそれぞれの変数に説明を加えていますので、発行する際には、書き換えてみてください。11行目で発行したトークンの宛先のアドレスを指定しています。今回は、このコントラクトを発行したアカウントに発行された全ての通貨が振り込まれるようになっています。この部分に、他のアカウントのアドレスを入れることもできます。

```
string private _name = "SampleToken";   //  トークンの名前
string private _symbol = "SPT";   //  トークンのシンボル
uint8 private _decimals = 18;   // トークンの小数点以下の桁数

address account = msg.sender;   // 発行したトークンの宛先
uint private value = 10 * 10 ** 18;   //  トークンの総発行量
```

14行目でコンストラクタを作成しています。コンストラクタの宣言の行にあるERC20DetaildではERC20Detaildコントラクトのコンストラクタを呼び出しています。コンストラクタ内にある_mint関数はERC20コントラクト内で作成されたもので、この関数により通貨の発行が行われています。

ERC20は発行された段階で通貨の送金機能や残高の確認など複数のインターフェイスが提供されています。それらはRemixで確認することができますので、発行した後に利用してみてください。

MetaMaskでのERCトークンの取引

MetaMaskでは、Etherだけでなく ERCに準拠したトークンであれば扱うことができます。ここではRopstenにデプロイしたトークンの利用方法について説明します。

MetaMaskを開き、設定ボタンから「ExpandView」を開く

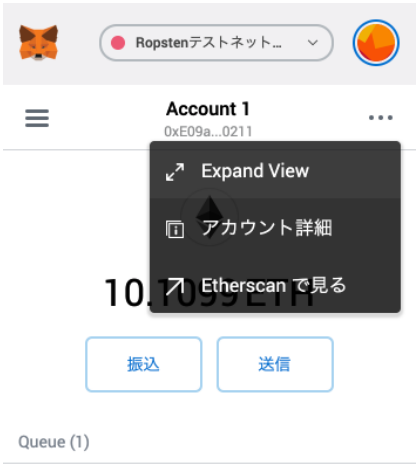


図7.4 ERCトークンの利用①

左下の「トークンの追加」をクリックする

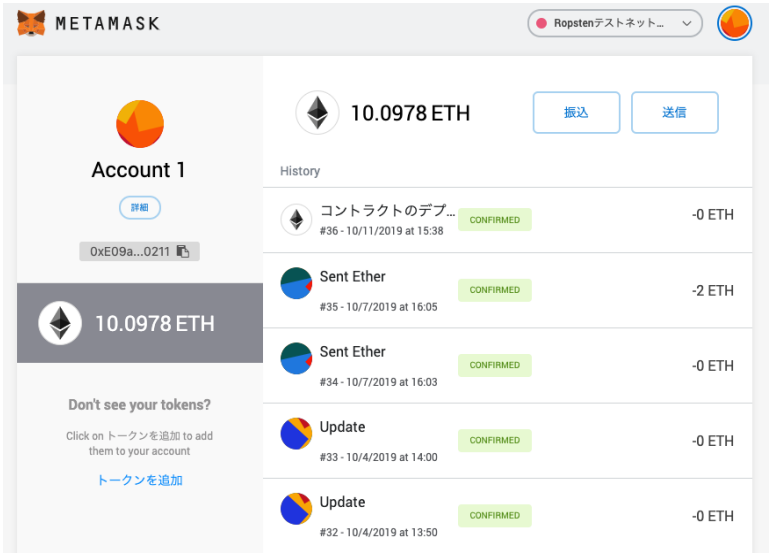


図7.5 ERCトークンの利用②

「カスタムトークンの追加」を選択し、Token Contract Addressにコントラクトのアドレスを入力し、「次へ」をクリックする

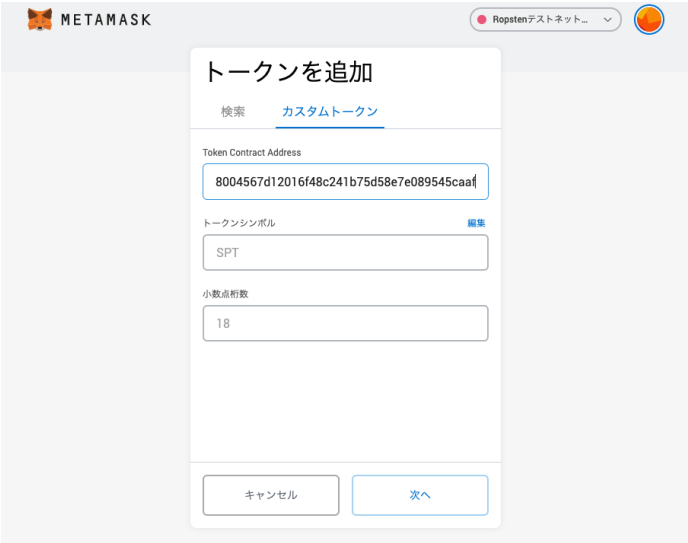


図7.6 ERCトークンの利用③

「トークンを追加」をクリックする

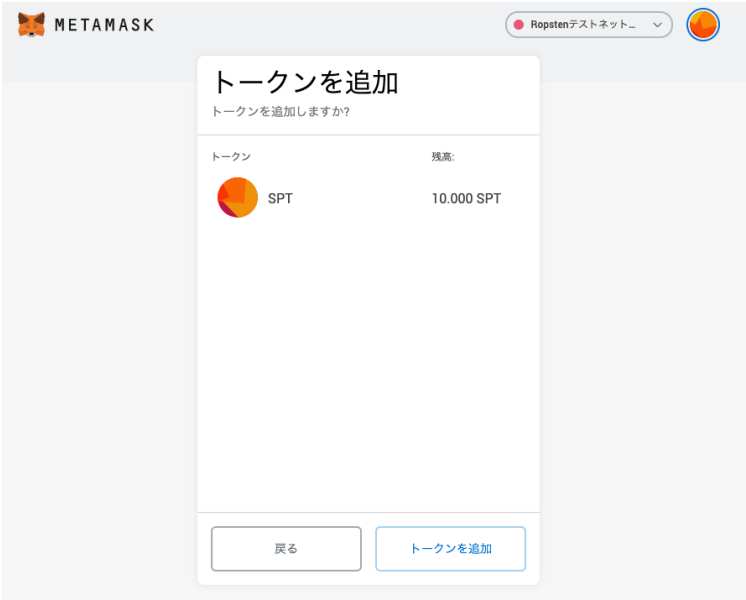


図7.7 ERCトークンの利用④

以上で、発行したERCトークンをMetaMaskから利用することができるようになります。

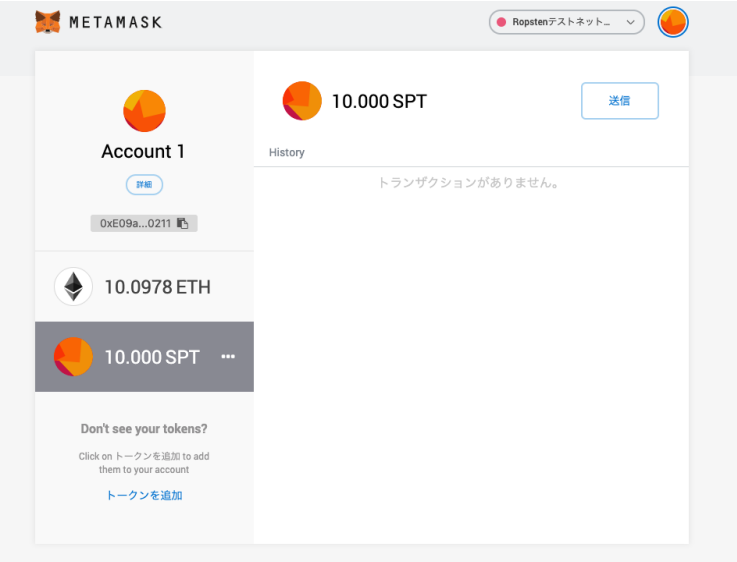


図7.8 ERCトークンの利用⑤

ここまでの設定が終わると、図7.8の右上にある「送信」から、送金相手のアドレスと、送金額を指定し作成したトークンを送信することができます。

相手から送られてきたトークンをMetaMaskで確認するためには、相手のトークンのアドレスを図7.4から図7.8までの流れに沿って、自身のMetaMaskに登録する必要があります。

ここまでで、自身のオリジナルのトークンを発行し、MetaMaskでそのトークンを利用することができるようになりました。今回作成したトークンには、基本的な送金機能しかありませんが、コントラクトに独自の状態変数や、関数を持たせることで機能を追加することができます。

演習2

以下のコントラクトをTruffleを用いてテストネットワーク上にデプロイし、ブラウザから操作できるようにしてください。

```
1.  pragma solidity ^0.5.10;
2.
3.  contract auction {
4.      uint public value;
5.      uint minValue = 10 ** 16;
6.      address payable public owner;
7.      address public winner;
8.      string public message;
9.
10.     modifier onlyOwner() {
11.         require(owner == msg.sender, "管理者のみ実行することできます");
12.         _;
13.     }
14.
15.     // コンストラクタ
16.     constructor() public {
17.         value = minValue;
18.         owner = msg.sender;
19.         winner = owner;
20.         message = "Hello";
21.     }
22.
23.      // messageを変更するための関数
24.     function changeMessage(string memory _message) public payable {
25.         require(value <= msg.value,"送金額が不足しています");
26.         message = _message;
27.         value = msg.value;
28.         winner = msg.sender;
29.     }
```

```
30.
31.     // コントラクトが保有する通貨量を取得する関数
32.     function getContractBalance() public view onlyOwner returns(uint) {
33.         return address(this).balance;
34.     }
35.     // オーナーがコントラクトから通貨を引き出す関数
36.     function withdraw() public payable onlyOwner {
37.         owner.transfer(address(this).balance);
38.     }
39. }
```

令和2年度「専修学校による地域産業中核的人材養成事業」
スマートコントラクトを使用したシステム開発人材の育成

スマートコントラクト開発入門

令和3年2月

学校法人 麻生塾　麻生情報ビジネス専門学校
〒812-0016　福岡県福岡市博多区博多駅南２丁目１２-３２