

# ブロックチェーン技術を用いたマネジメント法の 提案

プロジェクトマネジメントコース  
ソフトウェア開発管理グループ  
矢吹研究室  
1442068  
鈴木 博文

# 目次

第 1 章	序論	2
第 2 章	背景	3
2.1	ビットコインを始めとした仮想通貨の歴史 . . . . .	3
2.2	ブロックチェーン技術の歴史と応用例 . . . . .	5
2.3	Colony 社によるブロックチェーン技術の応用例 . . . . .	7
第 3 章	目的	8
第 4 章	手法	9
4.1	ソフトウェアの導入 . . . . .	9
第 5 章	結果	37
5.1	Naivechain によるブロックの作成 . . . . .	38
5.2	Ethereum を用いたブロックチェーン環境構築 . . . . .	44
5.3	プロトタイプモデルの実装 . . . . .	52
第 6 章	考察	62
第 7 章	結論	63
	参考文献	64
	謝辞	65

# 第 1 章

## 序論

当研究では，急速に発展が拡大するブロックチェーン技術を，プロジェクトマネジメント（以下，PM）学科内の研究室において利用した際の利点・難点を調査する．

ファイナンスとテクノロジーを掛け合わせた造語である「フィンテック」の分野における企業買収や設立が昨年から緩和された [1]．中でも世界的に流通が拡大しているのがビットコインを始めとした「仮想通貨」である．これが通貨として機能し，サービスが成り立つために必要な技術がブロックチェーンだ．

ブロックチェーンは「仮想通貨」だけでなく投資や投票など，他の分野でも活用が試みられている．

Jack du Rose 氏が運営する Colony 社は，ブロックチェーンを会社経営・マネジメントに応用し，インターネット上での組織の自律的な運営を試みている [2]．

## 第 2 章

# 背景

### 2.1 ビットコインを始めとした仮想通貨の歴史

#### ビットコインの歴史

ビットコインは、2008 年 10 月に中本哲史 (Satoshi Nakamoto) と名乗る人物がインターネット上に投稿した論文によって、提唱された。

それからわずか 3 ヶ月後、2009 年 1 月には、ビットコインの理論を実現するためのソフトウェアがオープンソースで開発され、公開された。そしてすぐに、ビットコインの最初の取引が行われた。開発者や、この時からビットコインを所有している人たちは、膨大な価値を持ったビットコインを所有していると言われている。

それからおよそ 1 年後の 2010 年 2 月に、ビットコイン両替が出来る最初の取引所が誕生した。そして同年 5 月、はじめて現実社会でビットコインを使った決済が行われている。

ビットコインとは、ひとことで言えば「仮想通貨」である。その名の通り「仮想」の「通貨」である。まず「通貨」ということから、「コイン」ということから分かる通り、ビットコインはお金である。円やドルのように「お金」であることには変わらない。「お金」なので円やドルのように通貨の単位が存在する。ビットコインの単位は、BTC (ピーティーシー) と表記される。1 円や 1 ドルのように、1BTC (1 ビットコイン) と数えることが出来る。

#### 一般的な仮想通貨の例

仮想通貨としてイメージしやすいのは、オンラインゲーム内の通貨である。円やドルを支払って、ゲーム内で使われている通貨を手に入れば、そのゲーム内で使われている通貨が「使える」ゲーム内のお店で、アイテムを買うことができる。ゲーム内でのみ使える、仮想の通貨である。

他にも、特定の Web サイトでのみ使える仮想通貨も存在する。利用登録の際に 1,000 円で 1,000 ポイントを購入し、その 1,000 ポイントで有料サービスを購入するというしくみを利用している Web サイトが存在する。そのポイントが「使える」Web サイト内でのみ有効な、仮想の通貨である。

## ビットコインと仮想通貨の違い

仮想通貨とビットコインは何が違うのか。実は存在している理由が全く違うのである。特定のゲームや Web サイト内でのみ使える仮想通貨は、企業単位で作られており、利用者を囲い込むことによって仮想通貨の運営主体が利益を上げることが目標としている。

一方のビットコインは、国家単位で運営されている円やドルと同じく、経済活動を円滑に進めるために作られた仮想通貨なのである。

ビットコインは、世界中で日常生活に「使える」ようにすることを目指して作られている。仮想通貨なので紙幣や硬貨は存在しないが、代わりにパソコンやスマートフォンをお財布代わりにして、物の売買が実現できるように作られている。

まだまだ発展途上ではあるが、円やドル以上に利便性が高く、安定し、世界中で利用できる次世代の通貨をめざして作られた仮想通貨なのである。

## ビットコインとブロックチェーン

ビットコインは銀行のような中央を経由せず、直接 1 対 1 で通貨のようなものを取引出来る仕組みである。これはつまりサーバークライアントモデルに基づいた信用によらず、取引ができるということである。

この仕組みは P2P(Peer-to-peer) 技術と、公開鍵暗号などの暗号技術を用いて実現されている。重要なのは、ネットワーク全体で、特定のどちらかの一方のみを一貫して、正しい取引であると決定できることである。そこでビットコインに導入されたのが「ブロックチェーン」という仕組みである。それぞれの「ブロック」は多数のトランザクションと、あとで説明する「ナンス」と呼ばれる特別な値、そして直前のブロックのハッシュを持っている。「ブロック」に含まれた取引のみを「正しい取引」と認めることにする。そして、ネットワーク全体で「唯一のブロックの鎖」を持つようにする。これによって、一貫した取引履歴を全体が共有できる、というのがブロックチェーンのコンセプトである。

## 2.2 ブロックチェーン技術の歴史と応用例

### ブロックチェーン技術の概要

ブロックチェーンとは、信頼性、説明責任、透明性を担保しながらビジネス・プロセスを効率化出来る、次世代のトランザクション・アプリケーションの基盤となる技術である。ブロックチェーンはビットコインで有名になった設計パターンだが、その用途にはビットコインにとどまらない可能性がある。ブロックチェーンを活用すれば、これまで全世界のビジネスの根幹を支えてきたビジネス上のやりとりの方法を刷新し、新たなデジタル式のやりとりの方法を構築することができる。ブロックチェーンには、企業間ビジネス・プロセスのコストと複雑性を大幅に削減する可能性がある。分散台帳に基づき低コストで簡単に構築できるビジネス・ネットワーク上で、これまでのような集中管理拠点を設けることなく、価値を持つすべてのモノを追跡、交換できるようになる。

この最新技術が秘めている可能性は、幅広いビジネス・アプリケーションへの応用が期待されている。

たとえば、ブロックチェーンを活用すれば証券の決済にかかる期間を数日から数分に短縮することができる。ほかにも、企業の物流とそれに伴う支払いの管理や、製造メーカー、OEM の委託先メーカー、規制当局間で製造ログを共有して商品のリコール件数を削減するなど、さまざまな用途が考えられる。

### ブロックチェーンの重要な概念

ブロックチェーンにおける 2 つの重要な概念は、参加メンバーが価値あるモノを交換する場となるビジネス・ネットワークと、各メンバーが所有し、メンバー間で常に内容が同期される台帳である。

ビジネス・ネットワークは複数のノードで構成される分散型 P2P アーキテクチャで、各ノードは市場の参加者である。これはプロトコルを共有する複数のピアがトランザクションを検証しコミットすることで合意を形成する。

共有台帳はブロックチェーン上で取引を行う企業に正しい情報を提供する唯一の機能である。ビジネス・ネットワーク上の全てのトランザクションを記録し、参加者間で共有される。各参加者に複製され、それぞれの参加者が独自にコピーを所有するが、自分が許可されたトランザクションしか参照できない。

#### ブロックチェーン技術の応用事例

ブロックチェーンは、仮想通貨以外への応用が始まっている。ブロックチェーンを応用し、送金や証券などの金融取引や資産管理をはじめとし、ポイントや公共サービスなどでも活用しようとする動きがある。

米 Nasdaq 社の未公開株式取引市場である Nasdaq Private Market の「Nasdaq Linq」は、株式未公開企業の従業員らが、自身で保有している株式を売買でき、その取引の「台帳」を実装する技術としてブロックチェーンを使用しているのだ。

金融以外の分野では米 Gyft Block 社がブロックチェーンを活用して、ポイント交換システムを立ち上げており、安価で信頼性の高い、ギフトカードを交換する仕組みを作り出している。

また、豪 Flux 社が政治に活かそうと選挙システムを構築するほか、英 Everledger 社はダイヤモンドの形状をセンサーで読み取りデジタル指紋に変換し、ブロックチェーンにダイヤモンドの認定書を記録する。そのダイヤモンドが消費者に販売されるまでの取引ルートを追跡してブロックチェーンに記録することにより、消費者が盗品を購入してしまうことを防いでいる。

上記以外にも、不動産取引・食品管理・農業支援・個人特定・履歴書管理など金融部門のみに限らず様々な分野での応用が進んでおり、ブロックチェーン技術の活用は今後さらに拡大が発展していくと考えられる。

## 2.3 Colony 社によるブロックチェーン技術の応用例

Colony 創業者兼最高経営責任者である Jack du Rose ( ジャック・デュ・ローズ ) 氏は分散型ネットワークを活用し、インターネット上に存在する組織を自立分散的に運営することで注目を集めている。もともとは宝石デザイナーとして活躍していた。

Colony 社 [3] の経営は、一言で言えば、分散型のプラットフォームであり、ブロックチェーンを活用することで、インターネット上での組織の自律的な運営を可能にするものである。

「信用」「信頼」「価値」を分散型で担保するオープンなプラットフォームである Colony は、分散型の企業や非営利組織の概念を提唱しており、「会社」のあり方そのものを買えうる可能性を秘めている。

それは「ソーシャルコラボレーションプラットフォーム」と言い換えることも可能だ。利用方法としてはたとえば「Web サイトをつくりたい」と考えたとき、その Web サイトを作るための「Colony」を立ち上げ、デザイナーやエンジニアなど参加者を募るというもの。「Colony」に参加するか、立ち上げるか。このどちらかで利用できる。

これだけ聞くとクラウドソーシングを思い浮かべるが、大きく違うのは、その対価・報酬と意思決定のあり方だ。自律分散型のガバナンスの重要性を指摘するデュ・ローズは、フリーランスをマネジメントするための方法として「ブロックチェーン」が有効だと指摘する。ジョブのアサイン、組織の意思決定に分散型ネットワークを活用するというわけだ。つまり参加者同士が直接つながり、提案と投票によって意思決定がなされる。その「Colony」への貢献度に応じ、民主的に評価される真にフラットな組織を志向する。

デザインやプログラミングといった作業への対価は、Colony 内の独自のトークンで支払われる。トークンは企業や株式のように「Colony」の所有権を表すもので、現金と取引することができる。



## 第 3 章

# 目的

ブロックチェーン技術を，マネジメントに応用した例を参考に，PM 学科内の研究室において同技術を利用した際の利点・難点を研究する．出欠情報・プロジェクト内での作業時間・成績情報・経歴情報など，存在証明が必要とされるドキュメントを管理するプロトタイプを実装し記録改ざんの複雑化と存在証明の効率化を図る．

## 第 4 章

# 手法

### 4.1 ソフトウェアの導入

本研究において、いくつかのソフトウェアを利用する。Windows 用ソフトウェアパッケージ管理ソフトである Chocolatey を始めとし、仮想環境を構築するために利用される Vagrant、VirtualBox、LinuxOS である Ubuntu16.04 LTS についても解説を行う。

研究内では、ブロックチェーンの基本的な概念を理解するために NaiveChain を用い、その後ブロック作成に関する契約行動を自動実行出来る、スマート・コントラクトが構築可能な Ethereum を利用する。ブロックチェーンについての基本実装についてはこの 2 つのソフトウェアについて解説を行う。

### 4.1.1 Chocolatey

Chocolatey[4] は Windows のコマンドライン上で利用することができる，Windows 用のパッケージマネージャである．Linux で用いられる apt-get や yum のようなパッケージ管理コマンドと同様に使用を行うことができる．図 4.1 は Chocolatey の公式ホームページである．

Chocolatey は，バージョン管理と依存関係の両方の要件を理解しているパッケージフレームワークを利用して，Windows ソフトウェアの管理のあらゆる側面を容易に処理できるように設計された単一の統一されたインターフェイスである．

Chocolatey 本体のインストールはコマンドプロンプト（管理者権限）を起動し以下のスクリプトを入力し実行する．

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile  
-InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object  
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1  
'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

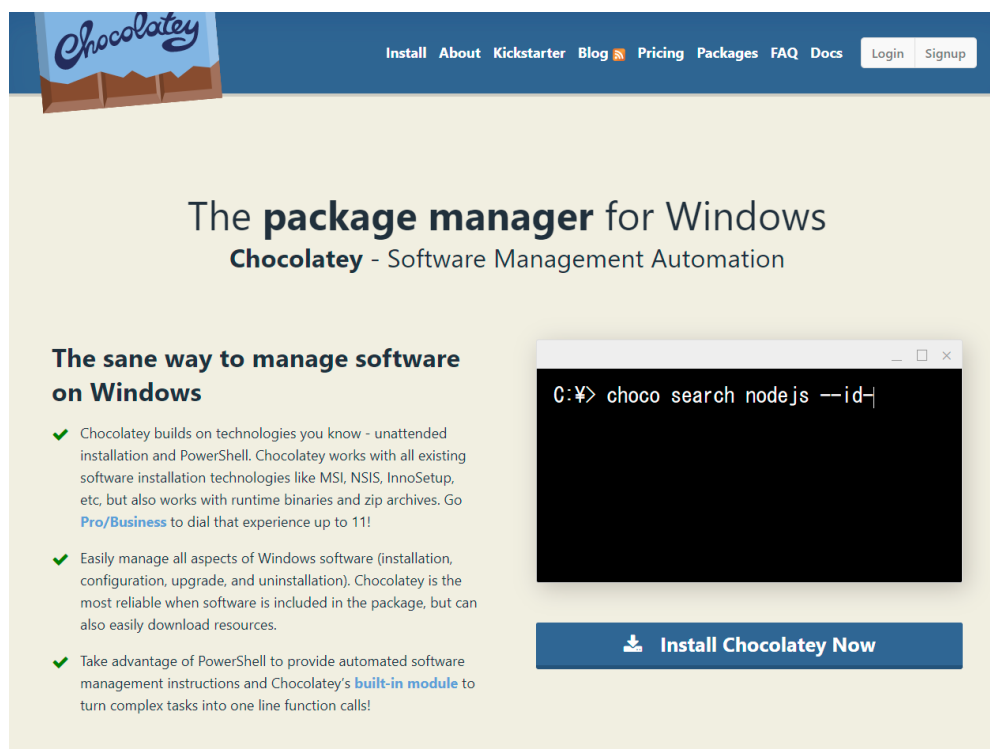


図 4.1 Chocolatey の公式ホームページ

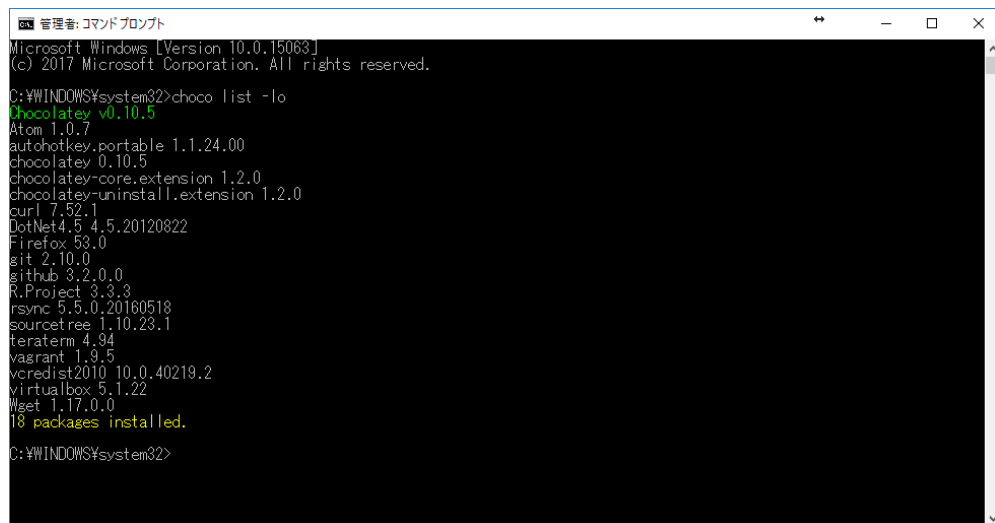
インストール完了後は `choco install [packageName]` などとして任意のパッケージをインストールする。 `choco install [packageName1] [packageName2] ...` とすると複数コマンドを一括してインストールできるほか、`-y` を追加することによりインストールへの同意を一括して `Yes` とすることが出来る。研究において利用する主なコマンドの一覧を表 4.1 として下記に記載する。

表 4.1 Chocolatey で使用できるコマンド

<code>choco install [packageName]</code>	指定パッケージをインストールする。
<code>choco list</code>	インストール可能なパッケージの一覧を表示する。
<code>choco list -lo</code>	インストール済みのパッケージ一覧を取得する。
<code>choco update [packageName]</code>	指定パッケージをアップデートする。
<code>choco update all</code>	インストール済みの全てのパッケージをアップデートする。
<code>choco uninstall [packageName]</code>	指定パッケージをアンインストールする。

当研究で利用するソフトウェアは以下のコマンドを Chocolatey で実行することで導入できる。図 4.2 はローカルにインストールされたソフトウェアの一覧を表示した結果である。

`choco install vagrant VirtualBox git rsync -y`



```

管理: コマンドプロンプト
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>choco list -lo
Chocolatey v0.10.5
Atom 1.0.7
autohotkey.portable 1.1.24.00
chocolatey 0.10.5
chocolatey-core.extension 1.2.0
chocolatey-uninstall.extension 1.2.0
curl 7.52.1
DotNet4.5 4.5.20120822
Firefox 53.0
git 2.10.0
github 3.2.0.0
RProject 3.3.3
rsync 5.5.0.20160518
sourcetree 1.10.23.1
teraterm 4.94
vagrant 1.9.5
vc_redist2010 10.0.40219.2
virtualbox 5.1.22
wget 1.17.0.0
18 packages installed.

C:\WINDOWS\system32>

```

図 4.2 Chocolatey を用いて導入したソフトウェア一覧の表示

### 4.1.2 VirtualBox

VirtualBox[5] は x86 ベース・システム用の強力なクロスプラットフォーム仮想化ソフトウェアである。「クロスプラットフォーム」とは、Windows, Linux, Mac OSX さらに Solaris x86 コンピュータにインストール可能であることを意味する。また、「仮想化ソフトウェア」とは、同じコンピュータ上で同時に複数の OS を実行する複数の仮想マシンを生成、実行できることを意味する。たとえば、Mac 上で Windows と Linux を実行したり、Windows PC 上で Linux と Solaris を実行したり、Linux システム上で Windows を実行したりすることができる。以下に VirtualBox を利用するにおいて用いられる主な用語の説明を記載する。

#### ハードウェアシミュレーション

ハードウェアによる仮想化支援機能として VT-x と、AMD-V への対応を含む。対応当初はデフォルトでどちらも有効となっていなかったが、現在のバージョンで提供される機能の一部には、これらの仮想化支援機能を必要とするものがある。バージョン 5.0 より KVM が選択可能になり、Linux においてハードウェアエミュレーションのオーバーヘッドが削減可能になった。

#### ハードディスク

ハードディスクドライブは、通常「仮想ディスク・イメージ (Virtual Disk Images)」と呼ばれる他の仮想化ソリューションとは互換性のない特別なコンテナ・フォーマットとしてエミュレートされる。これらは、ホスト OS 上のシステムファイル (拡張子.vdi) として格納される。別の方法として、VirtualBox は iSCSI の対象との接続が可能で、それらを仮想ハードディスク群として使用することが出来る。この他、他の仮想マシンソフトウェアで用いられる、vmdk (VMware), vhd (Microsoft Virtual PC), hdd (Parallels) などの仮想ディスクイメージにも対応する。ただし、これらのディスクイメージは本来 VirtualBox 向けのフォーマットではない為、フォーマットのバージョンと VirtualBox のバージョンの対応など、利用に当たっては互換性の面における注意が必要であるが、有志によりコンバートユーティリティがいくつか開発されている。

#### 光学ドライブ

CD や DVD ドライブとして ISO イメージが使用できる。例えば、Linux ディストリビューションの DVD イメージをダウンロードして、直接 VirtualBox で使用することが出来る。その場合、ISO イメージを CD-R や DVD-RW といった物理メディアに焼き込む必要がない。また、物理的ディスクを仮想マシンから直接的にマウントすることも可能である。

## ネットワーク機能

イーサネット・アダプタとして、AMD PCnet-PCI (Am79C970A),AMD PCnet-FAST (Am79C973),Intel PRO/1000 MT Desktop(82540EM),Intel PRO/1000 T Server(82543GC),Intel PRO/1000 MT Server(82545EM) のいずれかを仮想化する。これらの仮想化されたアダプタによる外部との接続手段として、NAT (ホスト OS による NATP 機能)、ブリッジアダプタ (ホスト OS の物理インターフェースとのブリッジ機能)、内部ネットワーク (ゲスト OS 同士を接続する内部的なネットワーク)、ホストオンリーアダプタ (ホスト OS 上の仮想 Ethernet アダプタと直接的に接続する) が提供される。新規作成されるマシンは、いずれかのアダプタと NAT の組み合わせが設定される。ゲスト OS 上のアプリケーションは、これによりホスト OS を経由して外部との通信が可能となる。NAT を提供するホスト OS は、一般的なブロードバンドルータと同様の動作を行う。バージョン 5.0 から準仮想化機能が搭載され始め、準仮想化ネットワークが選択可能になった。この仮想ネットワークインターフェースを利用することで、VirtualBox が virtio-net のドライバを持つ OS のカーネルと協調して VirtualBox 上のゲスト OS と物理ネットワークインターフェースの間で直接データを受け渡しすることが可能になり、ネットワークにおけるエミュレーションのオーバーヘッドを削減することが可能になる。

図 4.3 は VirtualBox の起動画面である。

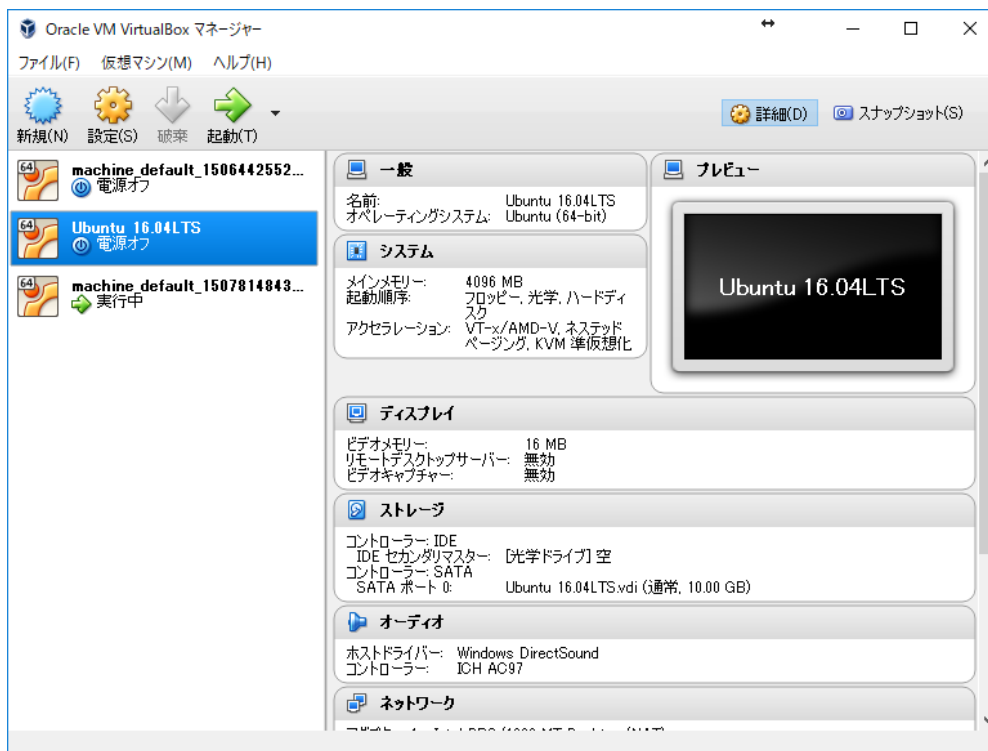


図 4.3 VirtualBox の起動画面

### 4.1.3 Vagrant

Vagrant[6] は、FLOSS の仮想開発環境構築ソフトウェアである。VirtualBox をはじめとする仮想化ソフトウェアや Chef や Salt, Puppet といった構成管理ソフトウェアのラッパーとみなすこともできる。

Vagrant を用いると、構成情報を記述した設定ファイルをもとに、仮想環境の構築から設定までを自動的に行うことが出来る。通常の仮想化ソフトウェアでは環境構築のために複雑なコマンド操作を行うが、Vagrant では数行のコマンドで構築が完了する。Vagrantfile というテキストファイルで仮想マシンの構成を管理しており、ファイル内に仮想マシン起動に構成管理ツールを実行するよう指定する箇所がある。この Vagrantfile と構成管理ツールの設定ファイルを共有することで、どこでも同じ仮想マシンを再現することができるようになる。以下に Vagrant を利用するにおいて用いられる主な用語の説明を記載する。

#### Box ファイル

仮想マシン起動時にテンプレートとなるイメージファイルであるが、一般的な ISO イメージファイルとは異なる。Box ファイルの一覧は <https://atlas.hashicorp.com/search> に公開されている。同サイトから Box ファイルを取得して利用するほか、自身で Box ファイルを作成することも可能である。

#### Vagrantfile

仮想マシンに構成を記述するファイル。主に以下の項目を指定する。

- 起動する仮想マシンの指定
- ネットワークの設定
- 共有フォルダの設定
- CPU やメモリの割当などマシンスペックの設定
- Shell や構成管理ツールの実行を指定するプロビジョニング

#### プロバイダ

仮想化ソフト、いわゆる仮想環境のこと。Vagrant ではデフォルトで VirtualBox をサポートするが、プラグインをインストールすれば VMware や Parallels, AWS(EC2) などにも対応可能である。

#### プロビジョニング

ミドルウェアのインストールや設定を行うツール。ここでは Shell や構成管理ツール (Chef, Puppet, Ansible) のことを指す。Vagrantfile 内で指定し、呼び出す。本研究では provision.sh としファイルを作成する。

## 共有フォルダ

ローカルと仮想マシンの間でファイルを同期する機能のこと。デフォルトでは、ローカルの Vagrantfile があるフォルダと仮想マシンの `$/vagrant` が同期されている。他のフォルダを指定したい場合は Vagrantfile の `config.vm.synced_folder` に指定する。

## プラグイン

`vagrant plugin install` で後からインストール可能な追加機能。

## コマンド一覧

Vagrant において使用される頻度の高いコマンドを表 4.3 に記載する。

表 4.2 Vagrant で使用できるコマンド

<code>vagrant box add &lt;name&gt;</code>	Box を追加する。
<code>vagrant box remote &lt;name&gt;</code>	Box を削除する。
<code>vagrant box list</code>	Box の一覧を表示する。
<code>vagrant init [name]</code>	Vagrantfile の作成をする。
<code>vagrant up</code>	仮想マシンを起動する。
<code>vagrant halt</code>	仮想マシンを停止する。
<code>vagrant reload</code>	仮想マシンを再起動する。
<code>vagrant destroy</code>	仮想マシンを削除する。
<code>vagrant package</code>	仮想マシンを Box 形式で出力する。
<code>vagrant plugin install &lt;name&gt;</code>	プラグインを追加する。
<code>vagrant plugin list</code>	プラグイン一覧を表示する。

上記以外の Vagrant で利用できるコマンドは `vagrant -h` で確認ができる。すべてのコマンドは `-h` オプションを付けて実行（例 `vagrant up -h`）することで、各コマンドの詳細なヘルプが表示される。



## 研究用仮想マシンの構築

本研究においては所属研究室である，矢吹研究室の公式仮想マシンを利用して開発を行う．研究室公式仮想マシンのベースとなる Box ファイル，プロビジョニングを作成する時点からの解説を行う．

まず，Guest Addition を更新し，ディスクサイズの変更を簡易化するためのプラグインを導入する．

```
vagrant plugin install vagrant-vbguest  
vagrant plugin install vagrant-disksize
```

C ドライブ上に Vagrant 専用のディレクトリを作成する．

```
cd :  
cd /  
mkdir vagrant  
cd vagrant  
git clone https://github.com/yabukilab/machine.git
```

研究室外や Vagrantfile を初期設定で利用したい場合などは Box ファイルを直接利用する．特に制限がなければ Ubuntu の最新 OS である Ubuntu16.04 LTS(ubuntu/xenial64) を利用する．

```
mkdir ubuntu  
cd ubuntu  
vagrant init ubuntu/xenial64
```

Vagrantfile をテキストエディタ等で開き , 下記スクリプトが有効であることを確認する .  
Box を取得した際に記載されているスクリプトはコメントアウトを外し , 記載がないものは下記を参考に追加する .

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/xenial64"
  if Vagrant.has_plugin?("vagrant-disksize")
    config.disksize.size = "10GB"
  end
  config.vm.network:forwarded_port, guest: 80, host: 80
  config.vm.network:forwarded_port, guest: 443, host: 443
  config.vm.network:forwarded_port, guest: 5432, host: 5432
  config.vm.network:forwarded_port, guest: 8888, host: 8888
  config.vm.synced_folder "./html", "/var/www/html"
  config.vm.provider "VirtualBox" do |vb|
    vb.memory = "4096"
    vb.cpus = 2
  end
  config.vm.provision "shell", path: "provision.sh"
  config.vbguest.auto_update = false
end
```

Vagrantfile があるディレクトリと同階層上に provision.sh を作成し下記スクリプトを記載する。2 行目は研究室にあるパッケージのキャッシュサーバを利用するための設定のため、研究室外で使うときは、この行を削除する。

```
#Timezone
timedatectl set-timezone Asia/Tokyo

echo 'Acquire::http::Proxy "http://10.100.192.4:3142/";' >
/etc/apt/apt.conf.d/02proxy

apt update

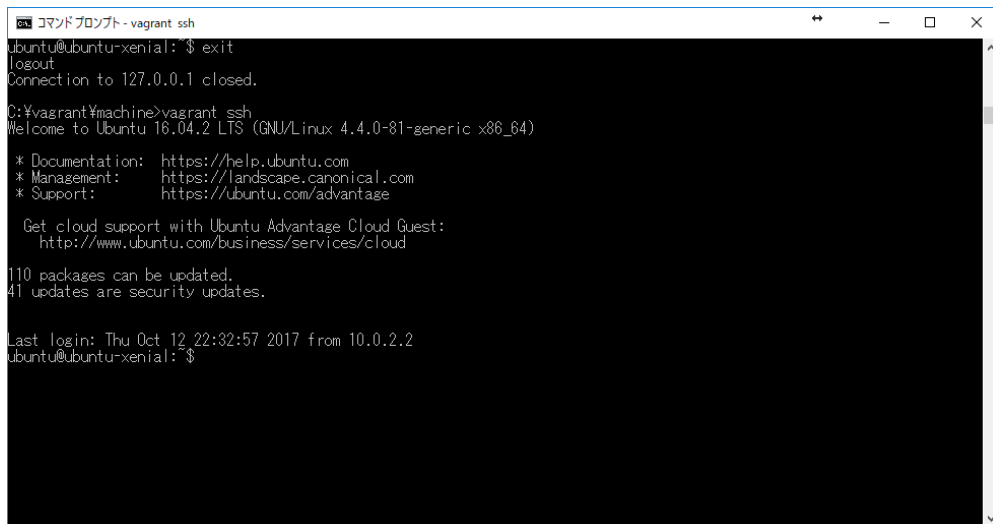
echo "alias ls='ls -F'" >> /home/ubuntu/.bashrc
echo "alias R='R --no-save'" >> /home/ubuntu/.bashrc
echo "alias jn='jupyter notebook --no-browser --ip=0.0.0.0'" >>
/home/ubuntu/.bashrc
chown ubuntu:ubuntu /home/ubuntu/.bashrc

echo '. ~/.bashrc' >> /home/ubuntu/.bash_profile
echo 'export PATH="$HOME/anaconda3/bin:$PATH"' >>
/home/ubuntu/.bash_profile
chown ubuntu:ubuntu /home/ubuntu/.bash_profile

echo "options(repos = 'https://cloud.r-project.org/')" >>
/home/ubuntu/.Rprofile
chown ubuntu:ubuntu /home/ubuntu/.Rprofile
```

vagrant up を実行し仮想マシンを起動する。VirtualBox 以外で利用する場合は vagrant up -provider virtulabox とする必要があるがデフォルトのプロバイダが VirtualBox の場合は特に指定する必要はない。指定した Box が登録されていない場合は、自動的にダウンロードして Box を追加 (vagrant box add) を行う。

仮想マシンが起動したら vagrant ssh として Windows のコマンドプロンプトから ssh ログインを行う。図 4.4 はその実行結果である。



```
コマンドプロンプト - vagrant ssh
ubuntu@ubuntu-xenial:~$ exit
logout
Connection to 127.0.0.1 closed.
C:\Vagrant\machine>vagrant ssh
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-81-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

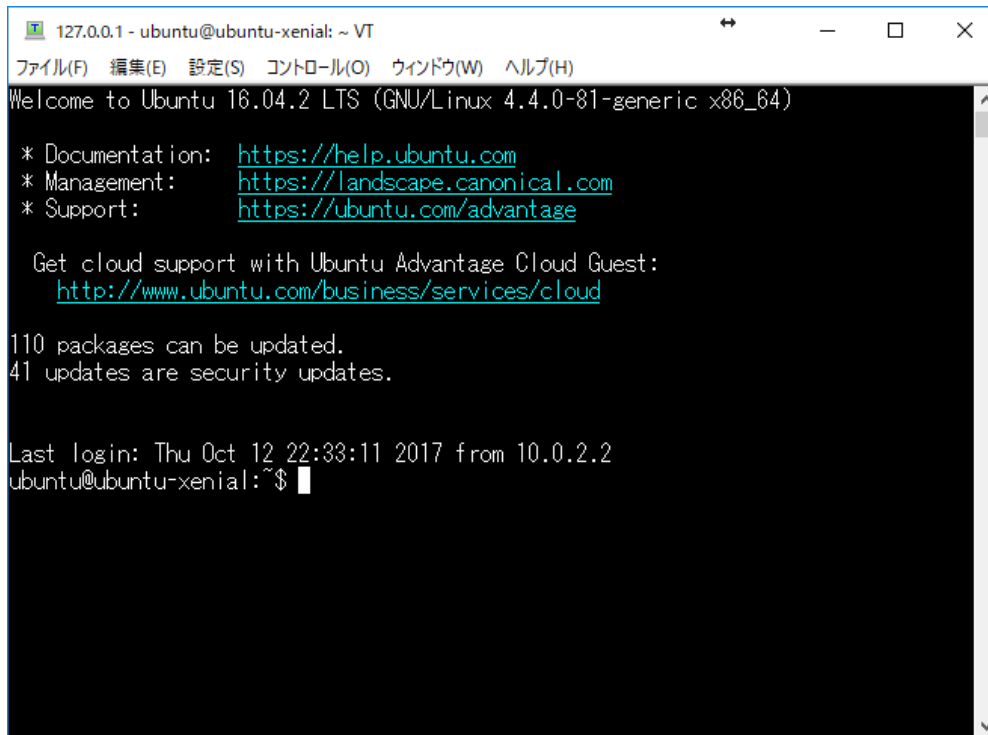
Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

110 packages can be updated.
41 updates are security updates.

Last login: Thu Oct 12 22:32:57 2017 from 10.0.2.2
ubuntu@ubuntu-xenial:~$
```

図 4.4 コマンドプロンプトから ssh 接続で表示した起動画面

PuTTY や TeraTerm などの SSH クライアントを使用してログインすることも出来る．接続先ホスト名は 127.0.0.1 , 接続先ポートは 2222 を指定する．ユーザ名は ubuntu パスワードは C:\Users\<username>\.vagrant.d\boxes\ubuntu-VAGRANTSLASH-xenial64\20XXMMDD.0.0\VirtualBox\Vagrantfile に記載がある．パスワードは passwd として任意に変更できるため，必要に応じて変更する．図 4.5 はその起動画面である．



```
127.0.0.1 - ubuntu@ubuntu-xenial: ~ VT
ファイル(F) 編集(E) 設定(S) コントロール(O) ウィンドウ(W) ヘルプ(H)
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-81-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

110 packages can be updated.
41 updates are security updates.

Last login: Thu Oct 12 22:33:11 2017 from 10.0.2.2
ubuntu@ubuntu-xenial:~$
```

図 4.5 TeraTerm から ssh 接続で表示した起動画面

#### 4.1.4 Ubuntu

Ubuntu[7] はコミュニティにより開発されているオペレーティングシステムである。ラップトップ、デスクトップ、そしてサーバーに利用することが出来る。Ubuntu には、家庭・学校・職場で必要とされるワープロやメールソフトから、サーバーソフトウェアやプログラミングツールまで、あらゆるソフトウェアが含まれている。

Ubuntu は現在、そして将来的に渡って無償で提供されている。ライセンス料を支払う必要はない。Ubuntu をダウンロードすれば、友人や家族と、あるいは学校やビジネスに、完全に無料で利用できる。

Ubuntu は、セキュリティに配慮して設計されています。デスクトップおよびサーバーの無償セキュリティアップデートが、少なくとも 9 ヶ月間に渡って提供される。長期サポート (LTS) 版を利用すれば、5 年間に渡りセキュリティアップデート提供される。もちろん、LTS 版を利用するために追加の費用は必要ない。図 4.6 は Ubuntu Desktop の起動画面である。

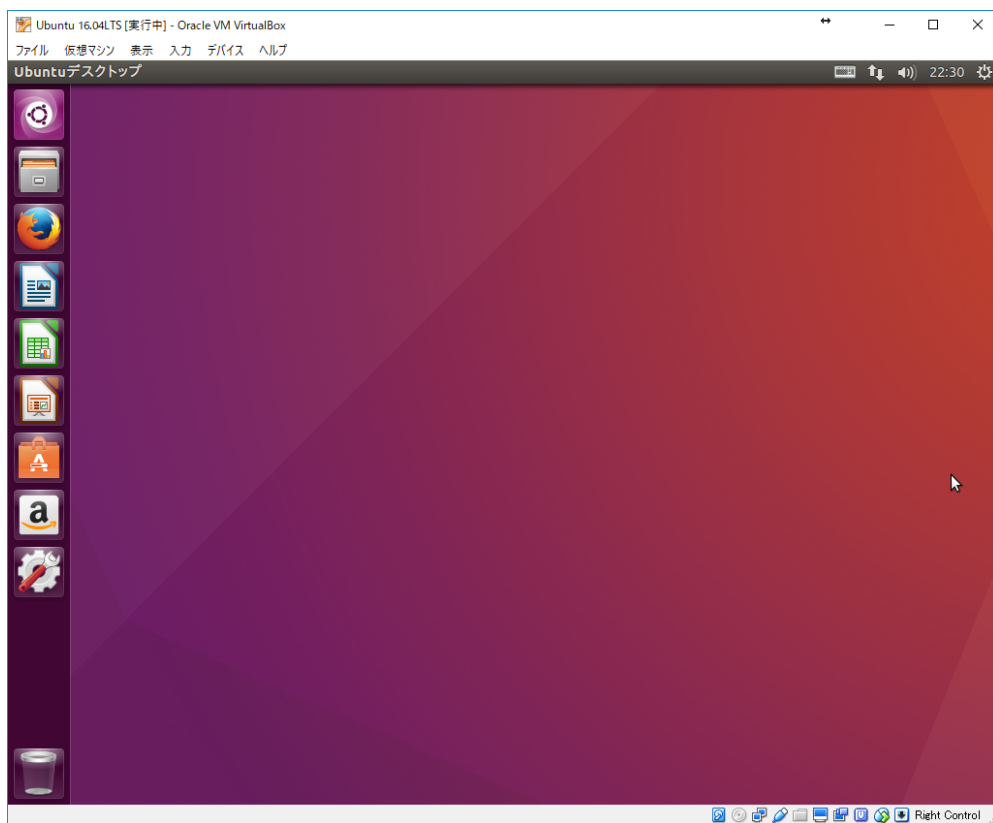


図 4.6 Ubuntu Desktop の起動画面

#### 4.1.5 Docker

Docker[8] はもともと，DotCloud 社（現 Docker Inc.）が開発者や IT 部門をターゲットとしたアプリケーションや OS の開発・配備を行うための基盤ソフトウェアとして開発され，2013 年にリリースされた．このソフトウェアは，オープンソースソフトウェアの「Docker」として公開され，その使い勝手の良さから，多くの開発 y 差，IT 部門の管理者で瞬く間に利用されることになった．Docker は，仮想化ソフトウェアに比べ，極めて集約度の高い IT システムを実現することができる．

Docker は，高性能なアプリケーションの開発・実行環境の提供だけでなく，それらの OS やアプリケーションを世界中で共有し，IT システムがある目的を達成するために必要な工程を自動化するサービスを提供する点が革新的と言える．世界中の開発者や IT 部門のシステム管理者が共通して理解できる同じ IT システムを使えば，他者が作ったシステムを少し変える，または，そのまま使うことで，開発，システムテスト，実システムへの配備の工数を大幅に削減することができるようになる．具体的には，例えば，Web アプリケーションの開発において，その動作に必要となるライブラリや，起動，停止，監視用のスクリプトを新たに調査・別途開発するといった“追加の作業”を極力おさえることができる．

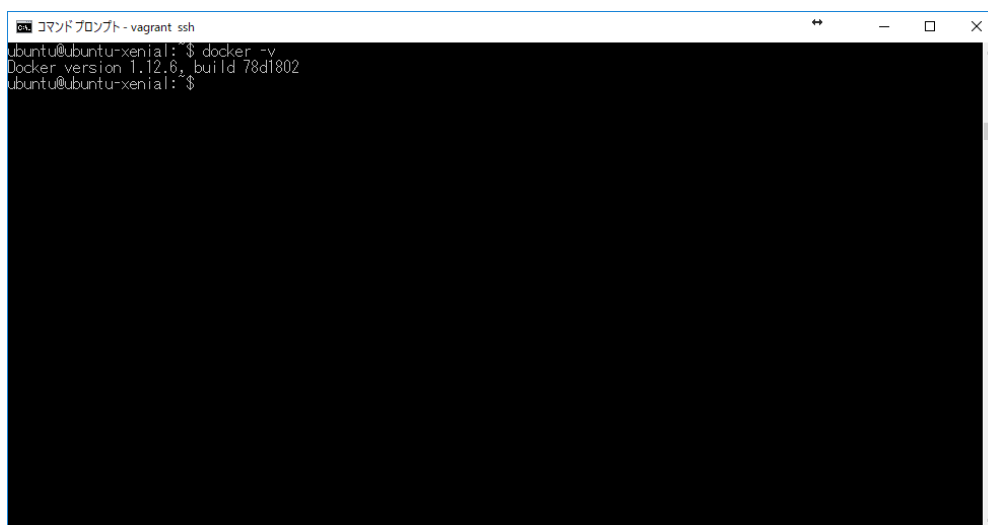
表 4.3 Docker で使用できるコマンド

<code>docker pull &lt;imagename&gt;</code>	イメージを参照してくる．
<code>docker build --no-cache=true</code>	キャッシュを無効にする．
<code>docker run -h example -i -t &lt;image-name&gt; /bin/bash</code>	ホスト名（例：example）を付けて起動する．
<code>docker run -name example -i -t &lt;imagename&gt; /bin/bash</code>	コンテナ名（例：example）を付けて起動する．
<code>docker start -a &lt;Container ID&gt;</code>	コンテナを起動してログインする．
<code>docker attach &lt;Container ID&gt;</code>	起動中のコンテナにログインする．
<code>docker ps</code>	起動中のコンテナの一覧を表示する．
<code>docker images</code>	イメージの一覧を表示する
<code>docker rmi &lt;Image ID&gt;</code>	イメージを削除する．

## Docker の導入

仮想環境内に Docker を導入するために以下の手順を実行する。

1. 管理者でないコマンドプロンプトを起動し `cd /vagrant/machine` などとして仮想環境のあるディレクトリへ移動する。
2. `vagrant up` を実行し `vagrant ssh` で仮想環境へ ssh 接続する。
3. `sudo apt-get update` を実行し、apt のアップデートを行う。
4. `sudo apt install docker.io -y` とし Docker の導入を行なう
5. `docker -v` と入力後実行し、図 4.7 のようにバージョン確認ができれば導入は完了である。なお、執筆時点でのバージョンは 1.13.1 である。

A screenshot of a terminal window titled 'コマンドプロンプト - vagrant ssh'. The terminal shows the command 'docker -v' being executed, resulting in the output 'Docker version 1.12.6, build 78d1802'. The prompt 'ubuntu@ubuntu-xenial:~\$' is visible before and after the command.

```
ubuntu@ubuntu-xenial:~$ docker -v
Docker version 1.12.6, build 78d1802
ubuntu@ubuntu-xenial:~$
```

図 4.7 Docker の起動画面



## Docker Compose

複数のコンテナを組み合わせた構成でサービスを運用している場合、コンテナに対し特定のパラメータを指定したり、特定の順序での起動が必要となるケースがある。こういった作業を自動化するツールが、「Docker Compose」だ。

Docker Compose は Docker が開発するコマンドラインツールで、あらかじめ用意しておいた設定ファイルに従ってコンテナを起動するツールだ。設定ファイルには複数のコンテナに関する記述が可能で、コンテナの起動オプションやコンテナに与える環境変数など、さまざまな設定も同時に記述できる。また、コンテナ同士の依存関係を設定することも可能で、これによって関連するコンテナを複数まとめて起動することも可能だ。

この場合、設定された依存関係に応じて適切な順番でコンテナが起動されるようになっており、コマンド 1 つで簡単に必要なサービスを開始できるようになっている。逆にコンテナを停止させる際も、自動的に適切な順番でコンテナを停止させるようになっている。また Docker Swarm との統合機能も提供されており、コンテナを複数実行させる（スケールさせる）操作も可能だ。コンテナの実行だけでなくビルド操作もサポートされている。

- build  
サービスのビルドを実行する。サービスとは「Web」や「DB」を指し、yml ファイルに image の情報がある場合でローカルにそのイメージ名がなければリモートからプルしてくる。
- config  
docker-compose.yml で書かれている内容が表示される。
- exec  
docker exec コマンドと同等のことができる。引数にサービス名と実行するコマンドを指定して実行する。
- images  
対象のイメージの情報を表示する。
- kill  
コンテナを強制停止する。
- ps  
コンテナの一覧を表示する。docker ps でも表示される。
- run  
引数で指定したサービスのコンテナ内でコマンドを実行する。
- start  
サービスを開始する。既にコンテナがある状態でのみ実行できる。
- stop  
サービスを停止する。
- up  
コンテナを作成して、起動する。-d をつけることでバックグラウンドで実行することができる。また-build をつけることで起動前にイメージも構築する。

## Docker-Compose の導入

仮想環境内に Docker-Compose を導入するために以下の手順を実行する。

1. 管理者でないコマンドプロンプトを起動し `cd /vagrant/machine` などとして仮想環境のあるディレクトリへ移動する。
2. `vagrant up` を実行し `vagrant ssh` で仮想環境へ ssh 接続する。
3. `sudo apt-get update` を実行し、`apt` のアップデートを行う。
4. `sudo apt install docker-compose -y` とし Docker の導入を行なう
5. `docker-compose -v` と入力後実行し、図 4.8 のようにバージョン確認ができれば導入は完了である。なお、執筆時点でのバージョンは 1.8.0 である。

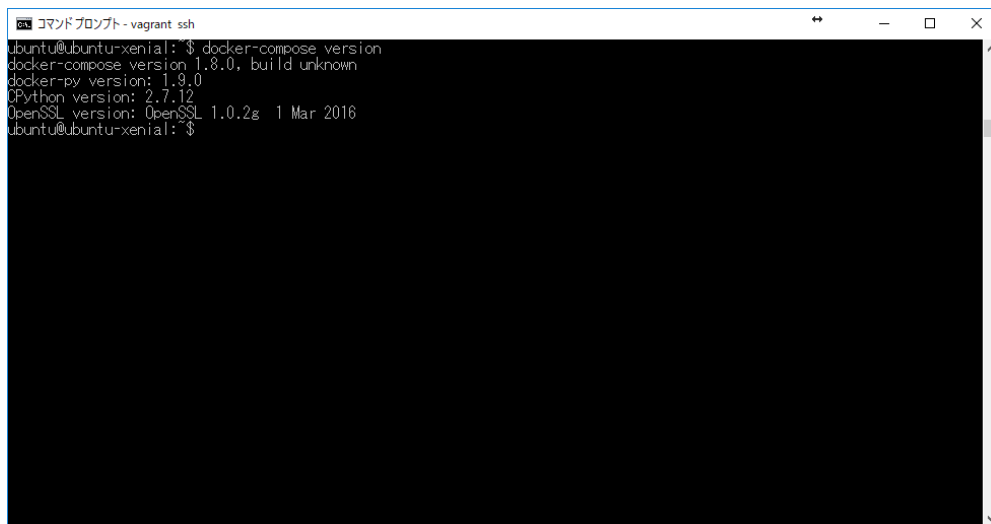
A terminal window titled 'コマンドプロンプト - vagrant ssh' showing the output of the 'docker-compose version' command. The output lists the Docker Compose version as 1.8.0 (build unknown), Docker-pty version as 1.9.0, Python version as 2.7.12, and OpenSSL version as OpenSSL 1.0.2g 1 Mar 2016. The prompt returns to 'ubuntu@ubuntu-xenial: \$'.

図 4.8 Docker Compose の起動画面

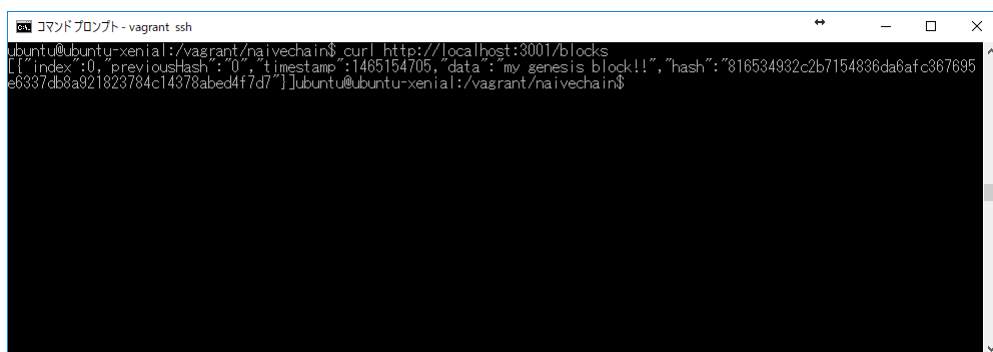
#### 4.1.6 NaiveChain

ブロックチェーンは基本的な概念は非常にシンプルである。分散型データベースで、順序付けられたレコードのリストが連続的に増加していく。しかしシンプルとは言え、ブロックチェーンやそれを使うことで解決しようとしている問題について話をする際にボトルネックとなることがある。

そのため、本来シンプルであるべきブロックチェーンの理解がより困難になってしまっている。抜け目のないソースコードであれば尚更である。そこで開発されたのが Lauri Hartikka 氏による NaiveChain[9] である。このソフトウェアは 200 行の JavaScript によって実装された、非常にシンプルなブロックチェーン作成ソフトウェアである。

以下の手順で導入とその確認を行なう。

1. 管理者でないコマンドプロンプトを起動し `cd /vagrant/machine` などとして仮想環境のあるディレクトリへ移動する。
2. NaiveChain の GitHub リポジトリを `git clone https://github.com/lhartikk/naivechain.git` としてローカルへクローンする。
3. `vagrant up` を実行し `vagrant ssh` で仮想環境へ ssh 接続する。
4. `$ cd /vagrant/naivechain` としホスト OS との共有フォルダ内へディレクトリを移動する。
5. `$ sudo docker-compose up -d` として NaiveChain を起動する。初回の起動には構築に時間がかかる。
6. NaiveChain においてブロックの作成・参照には cURL を用いる。NaiveChain の起動を確認するため `curl http://localhost:3001/blocks` を実行し、作成済みブロックの一覧を表示する。図 4.9 のように "my genesis block!!" のコメントを含んだ配列を取得できることを確認する。



```
ubuntu@ubuntu-xenial:/vagrant/naivechain$ curl http://localhost:3001/blocks
[{"index":0,"previoushash":"0","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7"}]ubuntu@ubuntu-xenial:/vagrant/naivechain$
```

図 4.9 NaiveChain でのブロック参照の結果

#### 4.1.7 Ethereum

Ethereum[10] は分散アプリケーションのためのプラットフォームであり，2013 年 12 月以降からオープンソースプロジェクトとして開発が進められているものだ．

インターネットの登場以来，メール，SNS，電子決済，クラウド・ファンディングなど数えられないほどの Web サービスを利用して日々を生活している．これらのサービスのほぼ全てにおいて，その運営に何らかの中央管理システムの存在が必須だった．

例えば SNS では，個人がアップロードしたデータを Facebook や Twitter といった企業は中央で一元管理することでサービスが成り立っている．クラウド・ファインディングでは，Kickstarter のような企業が，資金調達の仲立ちをし，その中で集まった資金についての管理を置こうなうことで，サービスが成り立っている．また，インターネットの基盤であるドメイン名も，ICANN を中心とした管理組織によって管理がなされている．

このような中央管理システムの存在するサービスは以下の点で欠点がある．

- 可用性：中央管理システムが存在する以上，その中央組織が何らかの理由で潰れなければサービスは継続しない．また，組織が存続している場合でも障害によるデータ消失の危険が避けられない．
- プライバシー：例えば SNS など，個人の生活のデータを私的企業が一手に握ることは，データ漏洩の危険性，企業によるデータの不正利用の危険性を考えると好ましいものではない．
- 検閲：中央の組織により管理されたサービスは，そのサービス提供者による独自の検閲が少なからず入る．検閲するかしないかは「サービス提供者」の手に握られ，たとえそれが公序良俗に「反しない」ものであっても検閲の対象になる可能性がある．

Ethereum は「ブロックチェーン」と呼ばれる技術をベースに，なんら特別な管理者のいない P2P システム上で様々なサービスを実現するための基盤を提供するものである．つまり Facebook や Twitter，Kickstarter や ICANN といったような，中央で管理する機関の存在を必要とせずと同様のサービスを実現する基盤を提供している．

Ethereum とは

ブロックチェーン技術を用いた新たな応用サービス公開するためには、2つの選択肢がある。1つは新たなサービスのために新しいブロックチェーンを構築しそれを使ってサービスを行うこと。2つめは、ビットコインのような既存のブロックチェーンを利用しその上にサービスを構築するというものだ。

新しいブロックチェーンを構築することは、非常に敷居の高い方法だ。実装とテストに相当の工数がかかる。さらに重要なのは、ブロックチェーンを用いた合意形成がサービスの公開当初から安定して動作するためには、事前に相当数の参加者が集まっている必要がある。これは需要の少ないニッチなサービスを展開する際には、非常に致命的な問題となる。

一方で、既存のビットコインのブロックチェーンを利用したサービス提供する場合は、あくまでビットコイン自体の設計上の制限に従う必要があり、非常に不自由なものになってしまう。

Ethereum は上記のようなブロックチェーンを利用した分散アプリケーションを開発しサービス提供を行う際の障壁を取り除くことを目的とした「分散アプリケーションプラットフォーム」だ。そのプラットフォームを形作るためのプロトコル定義や実装がオープンソース・プロジェクトとして行われている。

Ethereum は独自の P2P のブロックチェーンネットワークを構築し、分散アプリケーションが動作する実行環境の役割を果たす。様々な分散アプリケーションが Ethereum のブロックチェーンを共有して利用することで利用者の少ないニッチな分散アプリケーションでも、ブロックチェーンを利用した「合意形成」が安定して動作する環境を提供する。

また、分散アプリケーションのコードは、ブロックチェーンに組み込まれプルーフ・オブ・ワークの仕組みにより、改ざん不可能になる。このコードは Ethereum ネットワークに参加する各ノード上で実行され、その結果の状態もブロックチェーンに組み込まれ、やはり改ざんが不可能になる。

ビットコインはブロックチェーンの技術を用いて悪意のある参加者が参加する可能性のある P2P ネットワーク上で「取引」を正しく動作させる環境でした。一方で Ethereum は、取引だけでなく任意のアプリケーションをこのような P2P ネットワーク上で正しく動作させることを可能にする環境を提供する。

## Ethereum の仕組み

### 内部通貨：ether

Ethereum では「ether」という独自の内部通貨が規定されている。ビットコインと同様、それ自体が価値を持つ通貨としての利用も可能だが、より重要な事は ether が Ethereum 内で分散アプリケーションやスマート・コントラクトを実行するための「燃料」の役割を果たすということだ。Ethereum は上述のように、このプラットフォーム上で動作する分散アプリケーションに対して任意の処理を可能にしており、それぞれの分散アプリケーションの間でその動作に必要な計算資源の量が異なってくる。そこで Ethereum では、分散アプリケーションを実行するためには、その処理の重さに応じた燃料が必要とすることによって、Ethereum 上で動作する分散アプリケーション間での計算資源の割り当ての平等性を確保している。

### アカウント

Ethereum には「アカウント」と呼ばれるオブジェクトが既定されている。アカウントは 20Byte のアドレス（例：0x4c84598c919d82bfde24cdd13cc41f2aad9c33d3）により参照される。アカウントは主に次の 4 つのフィールドを持つ。

- nonce：そのアカウントが送信した累積トランザクション数
- ether balance：そのアカウントが所有する ether 量
- contract code：コントラクト・コード（EOA の場合は空）
- storage：そのアカウントが保持する任意のデータ

このフィールドのデータは、アカウント間でトランザクションが発生することにより変化する。つまり、アカウントの「状態」がトランザクションによって変化していく。「アカウント」には 2 つのタイプが存在する。1 つは「Externally Owned Account (EOA)」もう一つは「Contract」である。EOA は、我々ユーザにより生成されコントロールされるアカウントである。ユーザの任意のタイミングでトランザクションを生成し、他の EOA への ether の送金、新しい Contract の生成やコントラクト・コードの実行を行う。

一方で Contract は EOA はからトランザクションを介して生成される。Contract は一種の自動エージェントであり、EOA が発信するトランザクションをトリガに、コントラクト・コードを実行する。

## トランザクション

Ethereum では EOA から任意のタイミングでトランザクションを送信することで、各アカウントの状態が変化する。EOA がトランザクションを生成しそれを Ethereum ネットワーク上に送信する。採掘者は受信したトランザクションの正当性をチェックし問題なければ、そのトランザクションの情報とトランザクションの内容に基づいて変化した最新のアカウントの状態をブロックチェーンに埋め込む。トランザクションには主に以下の情報が含まれる。

- ether 送金額
- 相手先アドレス
- 送信アカウント署名
- 任意データ
- STARTGAS 値
- GASPRICE 値

最初の 3 つはビットコインのような暗号通貨のトランザクションと同じで、それぞれ、Ethereum の内部通貨である ether の送金額と相手のアドレス、そしてトランザクションの送信者が ether 送金元アドレスの所有者であることを証明するデジタル証明だ。

「任意データ」はトランザクションの相手先が Contract である場合に、そのコントラクト・コードに引き渡すデータを格納する。

## トランザクションの処理の流れ

トランザクションによりどのようにアカウントの状態が変更されていくのか大まかに以下になる。

1. EOA がトランザクションを生成し，Ethereum ネットワーク上に送信する．
2. ネットワーク内の採掘者がトランザクションを受信する．
3. 採掘者は，署名の正当性など，受信したトランザクションのデータに問題がないかをチェック．問題がある場合にはエラーとして以降の処理を行わない．
4. 採掘者はトランザクション内の STARTGAS 値と GASPRICE 値を参照．「STARTGAS 値 \* GASPRICE 値」の量の ether を，前払い手数料として，トランザクションを送信したアカウントの保有する ether から引く．もし ether の保有量が「STARTGAS 値 \* GASPRICE 値」よりも少なければエラーとして以降の処理を行わない．
5. 残り GAS = STARTGAS 値とする．
6. トランザクション・データの大きさ 1 バイト当たり 5gas を残り GAS から引く．
7. トランザクション内で指定された相手に対して，指定された額の ether を送金．また「トランザクションの相手」が Contract の場合は，Contract の持つコードを実行．
8. 送金額の ether を送金者が保有していない，または，コードを実行中に残り GAS がゼロになった場合には，手数料の支払情報のみを残し，元の状態にロールバックさせる．トランザクション実行前の状態から，トランザクション実行のための手数料分だけトランザクションの送信者の保有 ether から引き，採掘者の保有額にそれを足した状態を終状態とする．
9. 送金，またはコードの実行が正常に終了し，GAS が余っている場合は，その余った GAS をトランザクションの送信者に対して戻す．



## ブロックチェーンと採掘

ビットコインのシステムで、ブロックチェーンはビットコイン・ネットワーク上で発生したすべてのトランザクションを記録した、誰でも参照可能な公開取引元帳の役割を果たしていた。

Ethereum のブロックチェーンも同様に公開元帳の役割を果たします。ただビットコインの場合とは異なり、Ethereum のアカウントのブロックチェーンには、トランザクションだけでなく Ethereum ネットワークの全アカウントの最新の状態に関する情報も記録される。

つまり、ブロックチェーンにはトランザクションとアカウントの状態が記録されていき、そのブロックチェーンに書き込まれた状態を「正」とする Ethereum ネットワーク内の合意が形成されていく。

Ethereum ネットワーク内の採掘者は、ether の報酬を目当てに、ブロックの採掘競争を続ける。Ethereum では、新しいブロックが平均して 12 秒に 1 回採掘されるように動的プルーフ・オブ・ワークの難易度を調整されるように設計されている。

Ethereum では、採掘が成功すると

- 1 採掘あたり固定で 5ether。
- ブロックに含まれる全ての contract のコードを実行した際に消費した gas に相当する ether。
- ブロックに含んだ 1 つの Uncle ブロック当たり 1/32 ether。

の報酬が採掘者に与えられる。

## 4.1.8 Ethereum の導入

研究において構築するプロトタイプを作成するため，Ethereum の導入を行なう．以下に本研究において利用したソフトウェアのバージョンを掲載する．

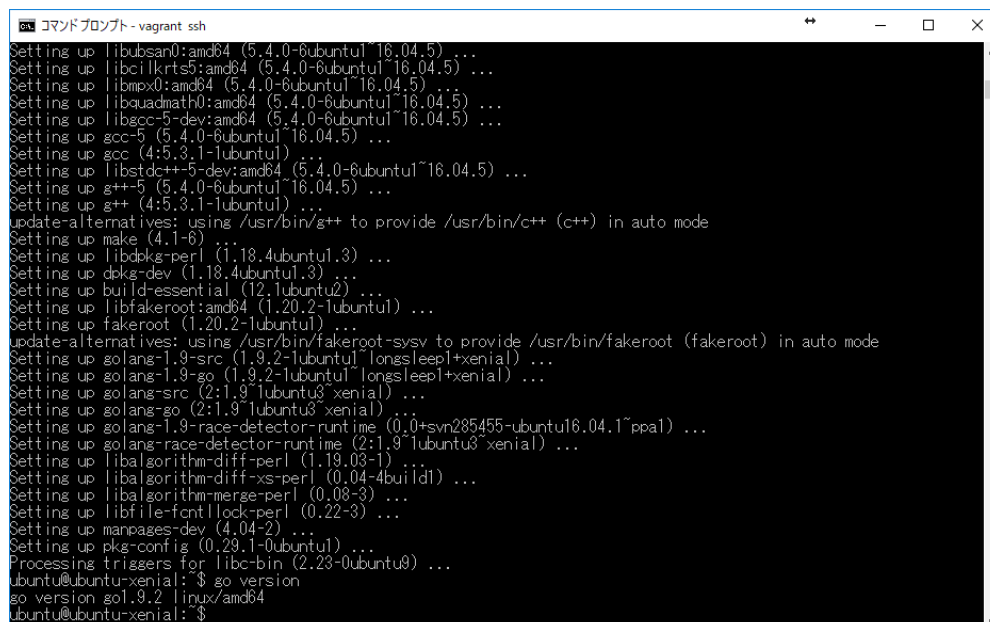
- ホスト OS : Windows10 64bit
- ゲスト OS : Ubuntu 16.04LTS 64bit
- Vagrant 2.0.1
- VirtualBox 5.2.0
- rsync 5.5.0

### Go の導入

Ethereum にて用いられるプログラミング言語である Go を導入する．以下にスクリプトを一覧として記載する．

```
sudo add-apt-repository ppa:longsleep/golang-backports
sudo apt-get update
sudo apt-get install golang-go -y
go version
```

Go の導入が完了すると，go version によって導入された Go のバージョンが図 4.10 のように表示される．



```
コマンドプロンプト - vagrant ssh
Setting up libubsan0:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up libcilkrts5:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up libmpx0:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up libquadmath0:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up libgcc-5-dev:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up gcc-5 (5.4.0-6ubuntu1~16.04.5) ...
Setting up gcc (4:5.3.1-1ubuntu1) ...
Setting up libstdc++-5-dev:amd64 (5.4.0-6ubuntu1~16.04.5) ...
Setting up g++-5 (5.4.0-6ubuntu1~16.04.5) ...
Setting up g++ (4:5.3.1-1ubuntu1) ...
update-alternatives: using /usr/bin/g++ to provide /usr/bin/c++ (c++) in auto mode
Setting up make (4.1-6) ...
Setting up libdkg-perl (1:18.4ubuntu1.3) ...
Setting up dkg-dev (1:18.4ubuntu1.3) ...
Setting up build-essential (12.1ubuntu2) ...
Setting up libfakeroot:amd64 (1.20.2-1ubuntu1) ...
Setting up fakeroot (1.20.2-1ubuntu1) ...
update-alternatives: using /usr/bin/fakeroot-sysv to provide /usr/bin/fakeroot (fakeroot) in auto mode
Setting up golang-1.9-src (1.9.2-1ubuntu1~longsleep1~xenial) ...
Setting up golang-src (2:1.9~1ubuntu3~xenial) ...
Setting up golang-go (2:1.9~1ubuntu3~xenial) ...
Setting up golang-1.9-race-detector-runtime (0.0+svn285455-ubuntu16.04.1~ppa1) ...
Setting up golang-race-detector-runtime (2:1.9~1ubuntu3~xenial) ...
Setting up libalgorithm-diff-perl (1.19.03-1) ...
Setting up libalgorithm-diff-xs-perl (0.04-4build1) ...
Setting up libalgorithm-merge-perl (0.08-3) ...
Setting up libfile-fcntllock-perl (0.22-3) ...
Setting up manpages-dev (4.04-2) ...
Setting up pkg-config (0.29.1-0ubuntu1) ...
Processing triggers for libc-bin (2.23-0ubuntu9) ...
ubuntu@ubuntu-xenial:~$ go version
go version go1.9.2 linux/amd64
ubuntu@ubuntu-xenial:~$
```

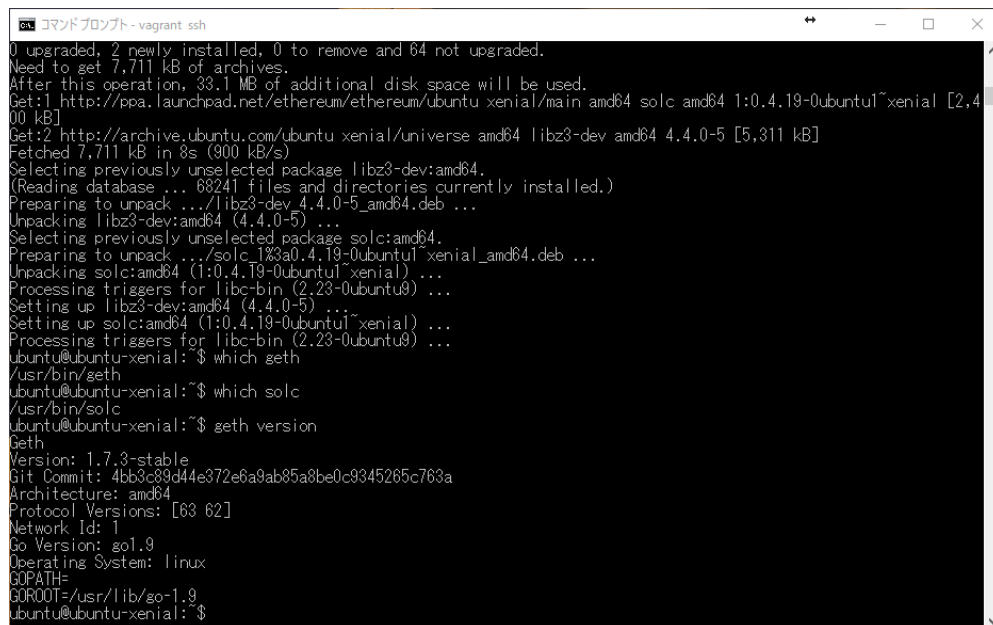
図 4.10 Go の導入とバージョンの確認

## Ethereum・Geth・Solc の導入

Ethereum にて用いられるプラグインの Geth と Solc を導入する．以下にスクリプトを一覧として記載する．

```
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum -y
sudo apt-get install solc -y
which geth
which solc
geth version
```

Geth・Solc の導入が完了すると，geth version によって導入された Geth のバージョンが図 4.11 のように表示される．



```
コマンドプロンプト - vagrant ssh
0 upgraded, 2 newly installed, 0 to remove and 64 not upgraded.
Need to get 7,711 kB of archives.
After this operation, 33.1 MB of additional disk space will be used.
Get:1 http://ppa.launchpad.net/ethereum/ethereum/ubuntu xenial/main amd64 solc amd64 1:0.4.19-0ubuntu1~xenial [2,400 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial/universe amd64 libz3-dev amd64 4.4.0-5 [5,311 kB]
Fetched 7,711 kB in 8s (900 kB/s)
Selecting previously unselected package libz3-dev:amd64.
(Reading database ... 68241 files and directories currently installed.)
Preparing to unpack .../libz3-dev_4.4.0-5_amd64.deb ...
Unpacking libz3-dev:amd64 (4.4.0-5) ...
Selecting previously unselected package solc:amd64.
Preparing to unpack .../solc_1%3a0.4.19-0ubuntu1~xenial_amd64.deb ...
Unpacking solc:amd64 (1:0.4.19-0ubuntu1~xenial) ...
Processing triggers for libc-bin (2.23-0ubuntu9) ...
Setting up libz3-dev:amd64 (4.4.0-5) ...
Setting up solc:amd64 (1:0.4.19-0ubuntu1~xenial) ...
Processing triggers for libc-bin (2.23-0ubuntu9) ...
ubuntu@ubuntu-xenial:~$ which geth
/usr/bin/geth
ubuntu@ubuntu-xenial:~$ which solc
/usr/bin/solc
ubuntu@ubuntu-xenial:~$ geth version
Geth
Version: 1.7.3-stable
Git Commit: 4bb3c89d44e372e6a9ab85a8be0c9345265c763a
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.9
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.9
ubuntu@ubuntu-xenial:~$
```

図 4.11 Geth の導入とバージョンの確認

## Geth の起動

Geth を起動するためには、ローカルプライベートテストネットで次の準備をする必要がある。

- データディレクトリ
- Genesis ファイル

データディレクトリは、送受信したブロックのデータや、アカウント情報を保存するためのディレクトリである。データディレクトリを指定することで、異なるブロックチェーンネットワークを共存させることが出来るようになる。ホームディレクトリに data\_testnet ディレクトリを作成する。

```
$ mkdir /data_testnet
$ cd data_testnet
$ pwd
/home/ubuntu/data_testnet
```

pwd にて取得したディレクトリは開発において何度も参照することとなるので、控えておくと良い。

続いて「Genesis ファイル」を用意する必要がある。Genesis ファイルは、ブロックチェーンの Genesis ブロック（0 番目のブロック）の情報を書いた json 形式のテキストファイルである。同じブロックチェーンネットワークに参加するノードは、同じ Genesis ブロックから連なるブロックチェーンを共有することとなる。ローカルプライベートネットを構築する場合には、ゼロからブロックチェーンを作ることになるため、Genesis ブロックの情報を書いた Genesis ファイルが必要となる。先ほど作成したディレクトリに genesis.json を配置する。

```
$ vim genesis.json
{
  "config": {},
  "nonce": "0x000000000000000042",
  "timestamp": "0x0",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "gasLimit": "0x8000000",
  "difficulty": "0x4000",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "alloc": {}
}
```

データディレクトリと Genesis ファイルが準備できたら , Geth を初期化する . 上記 pwd のパスを利用し実行する .

```
$ geth --datadir /home/ubuntu/data_testnet init /home/ubuntu/data_testnet/genesis.json
```

初期化が完了したら Geth を起動する . 以降もパスは任意で変更して対応する . 問題なく起動すると図 4.12 のように Welcome メッセージが表示される .

```
$ geth --networkid 4649 --nodiscover --maxpeers 0 --datadir /home/ubuntu/data_testnet console 2>> /home/ubuntu/data_testnet/geth.log
```

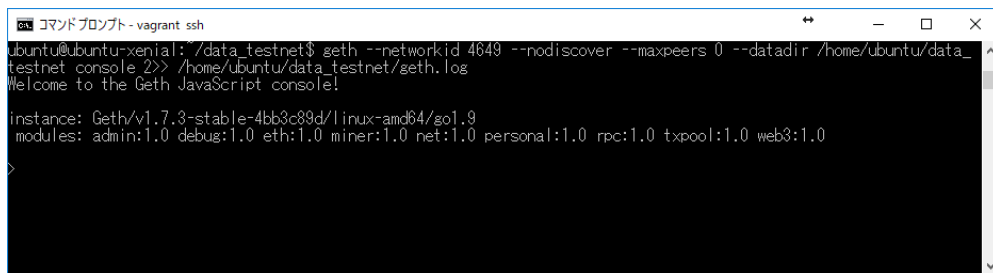


図 4.12 Geth の起動画面

各オプションの意味は以下のとおりである .

- --networkid 4649  
ネットワーク識別子 . 0 から 3 は予約済みである . (0=Olympic, 1=Frontier, 2=Morden, 3=Ropsten) . それ以外の数値であれば問題ない .
- --nodiscover  
利用のノードを , 他のノードから検出できないようにするオプションである . ノード追加は手動になる . 指定しないと , 同じ Genesis ファイルとネットワーク ID のブロックチェーンネットワークに利用中のノードが接続してしまう可能性がある .
- --maxpeers 0  
利用中のノードに接続できるノード数である . 0 を指定すると , 他のノードは接続しなくなる .
- --datadir /home/ubuntu/data\_testnet  
データディレクトリを指定する . 指定しないと , デフォルトのディレクトリが使用される . 環境に応じて置き換えて指定する必要がある .
- console  
対話型の JavaScript コンソールを起動する .
- 2>> /home/ubuntu/data\_testnet/geth.log  
ログファイルを作成するため , エラー処理をリダイレクトする .

## 第 5 章

# 結果

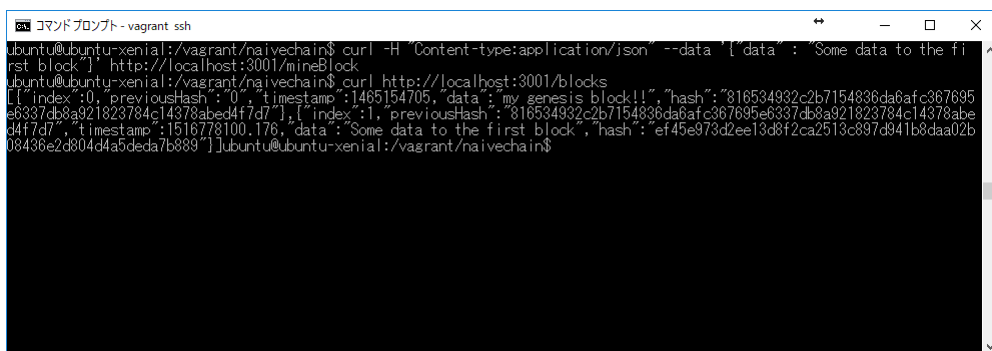
NaiveChain を用いてブロックの作成と確認を行なうことによって、ブロックチェーンにおけるブロックの繋がりとハッシュ値の関係性について理解を深める。また、Ethereum のテストネットワーク内にて Ethereum 内の仮想通貨である Ether の送金と、研究において構築するプロトタイプの作成を行った結果を記載する。

なお、全スクリプトは執筆時点での内容であり今後のバージョン変更に伴ってスクリプトの訂正が必要となる場合があることをあらかじめ断っておきたい。

## 5.1 Naivechain によるブロックの作成

4.1.6 を参照にディレクトリなどを変更したあと、以下の手順で作業を進め、NaiveChain を用いブロック作成を行なうことを通して、ブロックチェーンへの理解を深める。

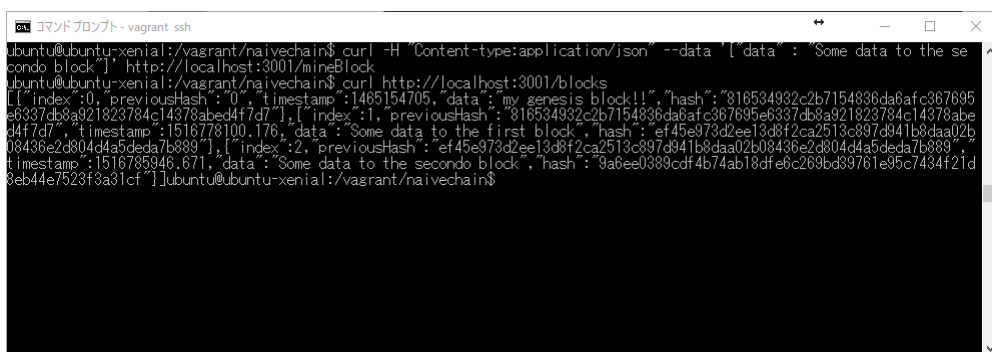
1. 新規ブロックを追加するため `curl -H "Content-type:application/json" -data '{"data": "Some data to the first block"}' http://localhost:3001/mineBlock` を実行する。その後、再度 `curl http://localhost:3001/blocks` を実行し、作成済みブロックの一覧を表示する。図 5.1 のように "index": "1" を含んだ 2 つ目のブロックが作成されていることを確認する。



```
ubuntu@ubuntu-xenial:/vagrant/naivechain$ curl -H "Content-type:application/json" --data '{"data": "Some data to the first block"}' http://localhost:3001/mineBlock
ubuntu@ubuntu-xenial:/vagrant/naivechain$ curl http://localhost:3001/blocks
[[{"index":0,"previousHash":"0","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abe44f7d7"},{"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abe44f7d7","timestamp":1516778100.176,"data":"Some data to the first block","hash":"ef45e973d2ee13d8f2ca2513c897d941b8daa02b08436e2d804d4a5deda7b889"}]]ubuntu@ubuntu-xenial:/vagrant/naivechain$
```

図 5.1 2 つ目のブロックを作成した結果

2. もうひとつブロックを追加するため `curl -H "Content-type:application/json" -data '{"data": "Some data to the second block"}' http://localhost:3001/mineBlock` を実行する。その後、再度 `curl http://localhost:3001/blocks` を実行し、作成済みブロックの一覧を表示する。図 5.2 のように "index": "2" を含んだ 3 つ目のブロックが作成されていることを確認する。



```
ubuntu@ubuntu-xenial:/vagrant/naivechain$ curl -H "Content-type:application/json" --data '{"data": "Some data to the second block"}' http://localhost:3001/mineBlock
ubuntu@ubuntu-xenial:/vagrant/naivechain$ curl http://localhost:3001/blocks
[[{"index":0,"previousHash":"0","timestamp":1465154705,"data":"my genesis block!!","hash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abe44f7d7"},{"index":1,"previousHash":"816534932c2b7154836da6afc367695e6337db8a921823784c14378abe44f7d7","timestamp":1516778100.176,"data":"Some data to the first block","hash":"ef45e973d2ee13d8f2ca2513c897d941b8daa02b08436e2d804d4a5deda7b889"},{"index":2,"previousHash":"ef45e973d2ee13d8f2ca2513c897d941b8daa02b08436e2d804d4a5deda7b889","timestamp":1516789946.671,"data":"Some data to the second block","hash":"9a6ee0369cdf4b74ab18dfe6c269bd39761e95c7434f21d8eb44e7523f3a31cf"}]]ubuntu@ubuntu-xenial:/vagrant/naivechain$
```

図 5.2 3 つ目のブロックを作成した結果

### ブロックの構造

まず行すべき作業は、ブロックの構造を決めることである。可能な限りシンプルにするため、最低限必要となる、インデックス、タイムスタンプ、データ、ハッシュ値、そして1つ前のブロックのハッシュ値のみを構造に含める。

```
class Block {
  constructor(index, previousHash, timestamp, data, hash) {
    this.index = index;
    this.previousHash = previousHash.toString();
    this.timestamp = timestamp;
    this.data = data;
    this.hash = hash.toString();
  }
}
```

### ブロックのハッシュ

ブロックはデータの安全性を確保するため、ハッシュ化されている必要があり、SHA-256によってブロック内のコンテンツが引き継がれる。

```
var calculateHash = (index, previousHash, timestamp, data) => {
  return CryptoJS.SHA256(index + previousHash + timestamp + data).toString();
};
```



## ブロックの生成

ブロックを生成するには、1 つ前のブロックのハッシュ値を知っていなければならない、その上でその他必要なコンテンツ(インデックス、ハッシュ値、データ、タイムスタンプ)を作成する。ブロックデータはエンドユーザによって提供される。

```
var generateNextBlock = (blockData) => {
  var previousBlock = getLatestBlock();
  var nextIndex = previousBlock.index + 1;
  var nextTimestamp = new Date().getTime() / 1000;
  var nextHash = calculateHash(nextIndex, previousBlock.hash,
    nextTimestamp, blockData);
  return new Block(nextIndex, previousBlock.hash, nextTimestamp,
    blockData, nextHash);
};
```

## ブロックの保存

ブロックチェーンを保存するには、インメモリの JavaScript の配列が使用される。ブロックチェーンの最初のブロックは常に「ジェネシスブロック」と呼ばれ、ハードコーディングされている。

```
var getGenesisBlock = () => {
  return new Block(0, "0", 1465154705, "my genesis block!!",
    "816534932c2b7154836da6afc367695e6337db8a921823784c14378abed4f7d7");
};

var blockchain = [getGenesisBlock()];
```

### ブロックの安全性の確認

いかなる場合でも、ブロックもしくはブロックのチェーンが安全性を満たしているか確認しなければならない。特に他のノードから新しいブロックを受け取ったとき、承認すべきかどうかの決断をしなければならない場合は重要となる。

```
var isValidNewBlock = (newBlock, previousBlock) => {
  if (previousBlock.index + 1 !== newBlock.index) {
    console.log('invalid index');
    return false;
  } else if (previousBlock.hash !== newBlock.previousHash) {
    console.log('invalid previoushash');
    return false;
  } else if (calculateHashForBlock(newBlock) !== newBlock.hash) {
    console.log('invalid hash: ' + calculateHashForBlock(newBlock) + ' '
      + newBlock.hash);
    return false;
  }
  return true;
};
```

### 最長チェーンを選択

いかなる場合でも、チェーンには明確なブロックのセットが1つでなくてはならない。矛盾する点があった場合はブロックの数が多いチェーンを選択する。

```
var replaceChain = (newBlocks) => {
  if (isValidChain(newBlocks) && newBlocks.length > blockchain.length) {
    console.log('Received blockchain is valid. Replacing current blockchain
      with received blockchain');
    blockchain = newBlocks;
    broadcast(responseLatestMsg());
  } else {
    console.log('Received blockchain invalid');
  }
};
```

### 他のノードとの連絡

ノードに不可欠なのは、ブロックチェーンを他のノードと共有、また同期する点である。  
以下は、ネットワークが常に同期されるために必要なルールである。

- ノードが新しいブロックを生成したら、ネットワークへと送信する。
- ノードが新しいピアに連結したら、最新のブロックに対してクエリを行う。
- ノードが、最新として知られるブロックのインデックスよりも大きいインデックスのブロックに出くわしたら、最新のチェーンにそのブロックを追加するか、全ブロックチェーンに対してクエリを行う。

### ノードの制御

何らかの方法で、ユーザはノードを制御する必要がある。これは HTTP サーバを設定することで可能である。

```
var initHttpServer = () => {
  var app = express();
  app.use(bodyParser.json());

  app.get('/blocks', (req, res) => res.send(JSON.stringify(blockchain)));
  app.post('/mineBlock', (req, res) => {
    var newBlock = generateNextBlock(req.body.data);
    addBlock(newBlock);
    broadcast(responseLatestMsg());
    console.log('block added: ' + JSON.stringify(newBlock));
    res.send();
  });
  app.get('/peers', (req, res) => {
    res.send(sockets.map(s => s._socket.remoteAddress + ':' +
      s._socket.remotePort));
  });
  app.post('/addPeer', (req, res) => {
    connectToPeers([req.body.peer]);
    res.send();
  });
  app.listen(http_port, () => console.log('Listening http on port: ' +
    http_port));
};
```

以下の方法で、ユーザはノードとの情報交換が可能となる。

- 全てのブロックを記載する。
- ユーザから与えられたコンテンツを基に新しいブロックを作成する。
- ピアを記載するか追加する。

```
[{"index":0,"previousHash":"","timestamp":1465154705,
"data":"my genesis
block!!","hash":"816534932c2b7154836da6afc367695e633
7db8a921823784c14378abed4f7d7"},
{"index":1,"previousHash":"816534932c2b7154836da6afc
367695e6337db8a921823784c14378abed4f7d7","timestam
p":1505904883.755,"data":"Some data to the first
block","hash":"75deb75c4631b562cd1a2f88f2ea553f3fa8
8f27d3bcdcaf92b645e1546c46f7"},
{"index":2,"previousHash":"75deb75c4631b562cd1a2f88f
2ea553f3fa88f27d3bcdcaf92b645e1546c46f7","timestam
p":1505904920.496,"data":"Some data to the second
block","hash":"2f4695c7f0a702eb85eb773089f9c3bba2ea
c8ef4454642ca421da182c782407"}]
```

図 5.3 NaiveChain を用いたブロック作成の実行結果

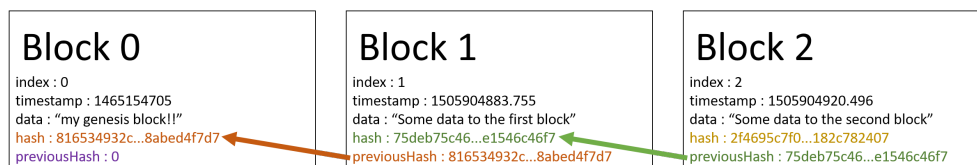


図 5.4 NaiveChain を用いたブロック作成結果の可視化

NaiveChian はデモンストレーションおよび学習目的で作成された。このブロックチェーンには「マイニング」アルゴリズムは含まれていないので、パブリックネットワークで使用することはできないが、ブロックチェーンを動かすための基本的な機能は実装されている。図 5.3 はブロック作成の実行結果。図 5.4 はその結果を可視化したものである。

## 5.2 Ethereum を用いたブロックチェーン環境構築

### 5.2.1 Ether の送金

Ethereum には EOA と Contract の 2 種類のアカунツがあることを説明した．今回は EOA を作成する．

```
> personal.newAccount("pass0")
"0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2"
```

"pass0"はアカウントノパスフレーズで，半角英数記号を使用した任意の文字列を指定できる．非常に用意なパスフレーズを指定したが，本番環境で使用する際には適切な長さで複雑なパスフレーズを指定する必要がある．ただし，忘れると復元することが出来ない．

"0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2"は，作成されたアカウントのアドレスである．このアドレスを指定して送金などを行うことになる．ここで，アドレスはユニークになるように作成されるので，実行の結果は環境によって異なる．アカウントの確認は `eth.accounts` コマンドで行う．このコマンドで表示されるアドレスは，当該ノードで管理しているアカウントのアドレスとなる．

```
> eth.accounts
["0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2"]
```

アカウントの作成が確認できたら，Ether の送金を行うために，もうひとつアカウントを作成する．作成後は `eth.accounts` コマンドで表示されるアカウントが増えたことを確認する．

```
> personal.newAccount("pass1")
"0x8a122d7d3283126ee0e10bb2b9e6b4ddd0a7b32b"
> eth.accounts
["0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2",
"0x8a122d7d3283126ee0e10bb2b9e6b4ddd0a7b32b"]
```

`exit` コマンドで Geth を終了する．

Geth を起動していない状態でもアカウントを作成することが出来る．パスフレーズは `pass2` としてアカウントを新規作成してみる．

```
$ geth --datadir /home/ubuntu/data_testnet account new
Your new account is locked with a password. Please give a password.
Do not forget this password.
Passphrase:pass2
Repeat passphrase:pass2
Address: {a92c91ea62cfa0d0a0570d46da51ff589b60b0c9}
```

引き続いて geth コマンドでアカウントを確認してみる .

```
$ geth --datadir /home/ubuntu/data_testnet account list
Account #0: {37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2}
keystore:///home/ubuntu/data_testnet/keystore/UTC--2018-01-25T
02-05-03.861297301Z--37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2
Account #1: {8a122d7d3283126ee0e10bb2b9e6b4ddd0a7b32b}
keystore:///home/ubuntu/data_testnet/keystore/UTC--2018-01-25T
02-18-08.652245148Z--8a122d7d3283126ee0e10bb2b9e6b4ddd0a7b32b
Account #2: {a92c91ea62cfa0d0a0570d46da51ff589b60b0c9}
keystore:///home/ubuntu/data_testnet/keystore/UTC--2018-01-25T
02-23-55.511872679Z--a92c91ea62cfa0d0a0570d46da51ff589b60b0c9
```

アカウントの確認が出来たら , 再度 Geth を起動する起動のコマンドは先ほどと同じものである .

起動が完了したら送金するための Ether をマイニングしていく . Ethereum では , マイニング成功時に報酬を受け取るアカウントを Etherbase と言う . Etherbase は , デフォルトでは eth.account[0] が設定されている . eth.coinbase コマンドで Etherbase を確認することが出来る .

```
> eth.coinbase
"0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2"
```

Etherbase は miner.setEtherbase コマンドで変更できる .

```
> miner.setEtherbase(eth.accounts[1])
true
```

変更ができたことを確認してみる .

```
> eth.coinbase
"0x8a122d7d3283126ee0e10bb2b9e6b4ddd0a7b32b"
```

miner.setEtherbase コマンドで変更できることが確認できたら , 以降の作業のため元のアカウントに戻しておく .

```
> miner.setEtherbase(eth.accounts[0])
true
> eth.coinbase
"0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2"
```

現在のアカウントの残高を確認する．残高の確認は `eth.getBalance` コマンドで、引数にアカウントノアドレスを渡す．作成直後のアカウントは Ether を所有していないため、今の段階ではどのアカウントでも実行結果は 0 となる．

```
> eth.getBalance(eth.accounts[0])
0
> eth.getBalance(eth.accounts[1])
0
> eth.getBalance(eth.accounts[2])
0
```

Ethereum もビットコインと同様にマイニングによって仮想通貨 Ether を報酬として獲得することができる．マイニングは `miner.start(threadnum)` コマンドで開始できる．ここで `threadnum` はマイニングを行うスレッド数である．

`eth.accounts[0]` から `eth.accounts[1]` に 10ether を送金してみる．送金は、`sendTransaction` コマンドで行う．`from` に送信元のアドレス、`to` に送信先のアドレス、`value` に送金金額を `wei` で指定する．

```
> eth.sendTransaction({from: eth.accounts[0], to: eth.accounts[1],
value: web3.toWei(10, "ether")})
```

```
Error: authentication needed: password or unlock
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1
```

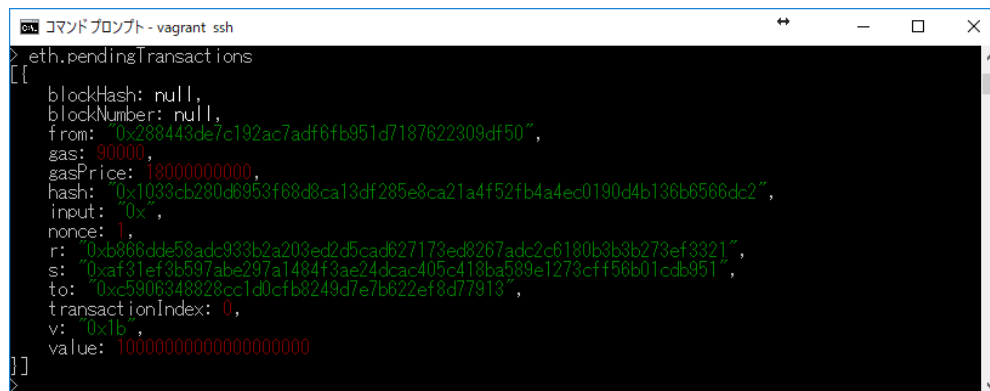
トランザクションの発行は手数料になるため、誤って実行できないように通常はロックされており、使用時にロックを解除する必要がある．`personal.unlockAccount` コマンドにアンロックするアカウントのアドレスを指定して実行する．

```
> personal.unlockAccount(web3.eth.accounts[0])
Unlock account 0x37ca4013f17e6fd8062afd8f9f66218d7a5ebfa2
Passphrase:
true
```

アンロックしたら再度 `sendTransaction` を実行する．結果は発行したトランザクションの ID である．環境によって変化するユニークなものなので、本論と異なる場合がある．

```
> eth.sendTransaction({from: eth.accounts[0], to: eth.accounts[1],
value: web3.toWei(10, "ether")})
"0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4
b136b6566dc2"
```

sendTransaction を実行しただけでは、処理は実行されない。ブロックチェーンでは、ブロックの中にそのトランザクションが取り込まれるときに、トランザクションの内容が実行される。トランザクションの状態を確認してみると、blockNumber が null となっている。ブロックにはっていない、つまり未処理であることが eth.pendingTransaction コマンドで図 5.5 のように確認することが出来る。

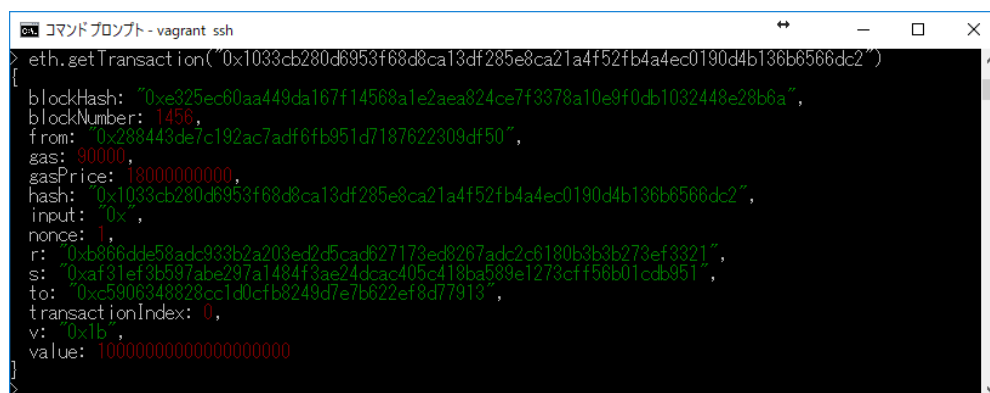
A terminal window titled 'コマンドプロンプト - vagrant ssh' showing the output of the 'eth.pendingTransactions' command. The output is a JSON array containing one transaction object. The object has fields: blockHash (null), blockNumber (null), from (0x288443de7c192ac7adf6fb951d7187622309df50), gas (90000), gasPrice (18000000000), hash (0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4b136b6566dc2), input (0x), nonce (1), r (0xb866dde58adc933b2a203ed2d5cad627173ed8267adc2c6180b3b3b273ef3321), s (0xaf31ef3b597abe297a1484f3ae24dcac405c418ba589e1273cff56b01cbb951), to (0xc5906348828cc1d0cfb8249d7e7b622ef8d77913), transactionIndex (0), v (0x1b), and value (1000000000000000000).

```
> eth.pendingTransactions
[[{"blockHash": null,
  "blockNumber": null,
  "from": "0x288443de7c192ac7adf6fb951d7187622309df50",
  "gas": 90000,
  "gasPrice": 18000000000,
  "hash": "0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4b136b6566dc2",
  "input": "0x",
  "nonce": 1,
  "r": "0xb866dde58adc933b2a203ed2d5cad627173ed8267adc2c6180b3b3b273ef3321",
  "s": "0xaf31ef3b597abe297a1484f3ae24dcac405c418ba589e1273cff56b01cbb951",
  "to": "0xc5906348828cc1d0cfb8249d7e7b622ef8d77913",
  "transactionIndex": 0,
  "v": "0x1b",
  "value": 1000000000000000000}]]
```

図 5.5 ブロックに入っていない未処理のトランザクションを表示した結果

miner.start コマンドでマイニングを再開して新しくブロックを作成する。eth.pendingTransactions を実行してトランザクションが表示されなくなったことを確認したら eth.getTransaction コマンドでトランザクションを確認する。先ほどは blockNumber が null であったが値が追加されていることが図 5.6 のように確認できる。

```
> miner.start()
null
> eth.pendingTransactions
[ ]
>miner.stop()
true
```

A terminal window titled 'コマンドプロンプト - vagrant ssh' showing the output of the 'eth.getTransaction' command with a specific transaction hash. The output is a JSON object with fields: blockHash (0xe325ec60aa449da167f14568a1e2aea824ce7f3378a10e9f0db1032448e28b6a), blockNumber (1456), from (0x288443de7c192ac7adf6fb951d7187622309df50), gas (90000), gasPrice (18000000000), hash (0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4b136b6566dc2), input (0x), nonce (1), r (0xb866dde58adc933b2a203ed2d5cad627173ed8267adc2c6180b3b3b273ef3321), s (0xaf31ef3b597abe297a1484f3ae24dcac405c418ba589e1273cff56b01cbb951), to (0xc5906348828cc1d0cfb8249d7e7b622ef8d77913), transactionIndex (0), v (0x1b), and value (1000000000000000000).

```
> eth.getTransaction("0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4b136b6566dc2")
{"blockHash": "0xe325ec60aa449da167f14568a1e2aea824ce7f3378a10e9f0db1032448e28b6a",
  "blockNumber": 1456,
  "from": "0x288443de7c192ac7adf6fb951d7187622309df50",
  "gas": 90000,
  "gasPrice": 18000000000,
  "hash": "0x1033cb280d6953f68d8ca13df285e8ca21a4f52fb4a4ec0190d4b136b6566dc2",
  "input": "0x",
  "nonce": 1,
  "r": "0xb866dde58adc933b2a203ed2d5cad627173ed8267adc2c6180b3b3b273ef3321",
  "s": "0xaf31ef3b597abe297a1484f3ae24dcac405c418ba589e1273cff56b01cbb951",
  "to": "0xc5906348828cc1d0cfb8249d7e7b622ef8d77913",
  "transactionIndex": 0,
  "v": "0x1b",
  "value": 1000000000000000000}
```

図 5.6 ブロックを新たに追加することに成功した結果



## 5.2.2 スマート・コントラクト

### スマート・コントラクトの説明

スマート・コントラクトは、ブロックチェーン上で動作するアプリケーションの位置づけとなる。スマート・コントラクトの開発の大きな流れは、Web アプリケーション開発と同様である。開発者はコードを書き、サーバ（ブロックチェーン）にデプロイする。ユーザはブラウザでサーバにアクセスし、目的の処理を行う。

開発者は高レベルの言語でコントラクトを記述する。そしてそれを EVM コンパイラでコンパイルして EVM バイトコードにし、ブロックチェーンにデプロイする（EVM とは Ethereum Virtual Machine の略）。EVM バイトコードは、個々の EVM 上で実行される。これはちょうど Java プログラム、Java バイトコード、JVM の関係に似ている。なお「ブロックチェーンにデプロイする」とは、ブロック内に EVM バイトコードを保存することを意味する。ブロックチェーンネットワークに参加する全ノードは同じブロックを保持するので、全ノードが EVM バイトコードを保持して実行できることになる。

Ethereum のコントラクトを書くためのプログラム言語は、現在、次の 3 種類がある。

- Solidity

Solidity は、JavaScript に似た構文を持つ言語で、現時点における Ethereum のコントラクト開発における主要言語であり、最も人気がある。

- Serpent

Serpent は Python に似た構文の言語である。インターネット上の情報はそれほど多くない。

- LLL(Lisp Like Language)

アセンブリに似た低レベルの言語である。

## スマート・コントラクトの利用

開発中の環境においてスマート・コントラクトが利用できることを確認するために、"Hello, World!"などの簡単な文字列を返すだけのコントラクトを実行してみる。プログラムの内容は以下のとおりである。

```
pragma solidity ^0.4.8;
contract HelloWorld {
    string public greeting;
    function HelloWorld(string _greeting) {
        greeting = _greeting;
    }
    function setGreeting(string _greeting) {
        greeting = _greeting;
    }
    function say() constant returns (string) {
        return greeting;
    }
}
```

プログラム内の変数は以下の意味を示す。なお、重要な点のみ示す。

- コントラクトの宣言：contract HelloWorld {  
contract でコントラクトを宣言する。コントラクトは Java などのオブジェクト指向プログラミング言語のクラスと似ており任意の名前を付けることが出来る。
- 状態変数の宣言：string public greeting;  
コントラクト内で有効な変数を宣言できる。Ethereum では、これを状態変数と呼ぶ。今回はユーザから渡された文字列を保持する変数 greeting を宣言した。

コントラクト実行は以下の手順で行なう，データ部やトランザクションの内容を全文掲載すると膨大な量となるため一部割愛する．

1. 任意の場所にコントラクトプログラムを配置し，コントラクトプログラムビルド用のデータを出力する．

```
$ solc -o ./ --bin --optimize HelloWorld.sol
$ cat HelloWorld.bin
6060604052341561000f57600080fd5b60405161046d38038061.....
.....61e4c1bf9257fd6542dbf1628bf82a571b89769cc4c11f5be0029
```

2. コントラクトの情報を取得する．

```
$ solc --abi HelloWorld.sol
[{"constant":true,"inputs":[],"name":"say","outputs":[{"name":
"", "type":"string"}],"payable":false,"stateMutability":"view",
(割愛) e,"stateMutability":"nonpayable","type":"constructor"}]
```

3. Geth を起動する．

```
$ geth --networkid 4649 --nodiscover --maxpeers 0 --datadir /home/
ubuntu/data_testnet console 2>> /home/ubuntu/data_testnet/geth.log
```

4. コントラクト登録者のロックを解除する．

```
> personal.unlockAccount(eth.accounts[0])
Unlock account 0x288443de7c192ac7adf6fb951d7187622309df50
Passphrase:
true
```

5. コントラクトをブロックチェーンに登録する．

```
> HelloContract = web3.eth.contract([{"constant":true,"inputs":[],
(割愛) se,"stateMutability":"nonpayable","type":"constructor"}]);
```

6. 0x をデータ部に入れて 16 進数であることを明確にする．

```
> Hello = HelloContract.new({from: eth.accounts[0], data: '0x6060
604052341561000f57600080fd5b60 (割愛) 161046d380aa7098a17
57fd6542dbf1628bf82a571b89769cc4c11f5be0029', gas: 3000000})
```

7. マイニングを開始する．

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

8. コントラクトにアクセスするための変数を定義する .

```
> contractObj = eth.contract(Hello.abi).at(Hello.address)
```

9. コントラクト登録者のロックを解除する .

```
> personal.unlockAccount(web3.eth.accounts[0])
Unlock account 0x288443de7c192ac7adf6fb951d7187622309df50
Passphrase:
true
```

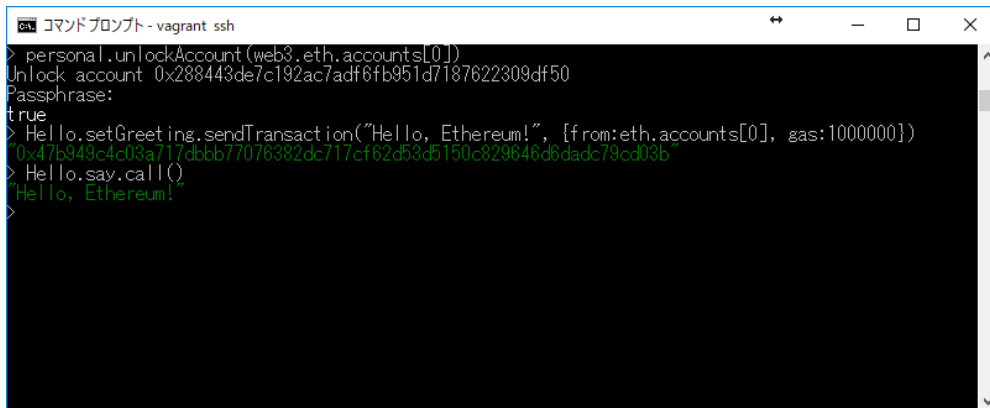
10. setGreeting を実行して呼び出す変数を更新する .

```
> Hello.setGreeting.sendTransaction("Hello, Ethereum!",
{from:eth.accounts[0], gas:1000000})
"0x0d9b04455fa0e7672ad20b9d2698834f18af4c07ed5b
1e9efce3ff0a35b4d8c7"
```

11. say を実行して更新されたかを確認する .

```
> Hello.say.call()
"Hello, Ethereum!"
```

スマート・コントラクトがここまで問題なく実行できたら , 図 5.7 のように自身で設定した変数を呼び出すことが成功している . 以降は動作の確認が出来たため , スマート・コントラクトをもちいたプロトタイプ作成を開始した .



```
コマンドプロンプト - vagrant ssh
> personal.unlockAccount(web3.eth.accounts[0])
Unlock account 0x288443de7c192ac7adf6fb951d7187622309df50
Passphrase:
true
> Hello.setGreeting.sendTransaction("Hello, Ethereum!", {from:eth.accounts[0], gas:1000000})
"0x47b949c4c03a717dbbb77076382dc717cf62d53d5150c829646d6dadcf9cd03b"
> Hello.say.call()
"Hello, Ethereum!"
>
```

図 5.7 スマート・コントラクトをもちいた簡単な変数の設定と出力結果

### 5.3 プロトタイプモデルの実装

本研究において、構築したプロトタイプモデルは「出欠情報の管理」「経歴情報の管理」を行なうスマート・コントラクトで実装されたプログラムである。

「出欠情報の管理」を行なうプログラムは、授業およびゼミ等への出席が間違いなく本人によるものであることの証明・授業やゼミが実施された日時において本人が出席していたか欠席していたかの情報の改ざんの防止などを図るために作成したものである。

記録情報は、本人氏名、日時、出欠の有無であり本人氏名と日時を参照することによって出欠情報をブロックチェーンから取り出すことが出来る。

「経歴情報の管理」を行なうプログラムは、記録者本人の所属している研究室や大学、社会人である場合には企業や機関など任意の値を登録することが出来る。記録する本人のアカウントを始め、記録される側のアカウントも作成することによって P2P ネットワークによって本人が間違いなくその組織に所属していることを確認するものである。

記録確認は本人が可能であることはもちろん、閲覧者となる本人以外のアカウントは記録者本人によって閲覧の有無とその期間を決定することができ、ブロックチェーンでの管理のため改ざんの防止も図るよう作成している。

### 5.3.1 出欠情報の管理

任意の値を用いて登録が可能な出欠情報を管理するプログラムを作成した．attendance.sol として作成し，プログラムの内容を以下に記載する．

```
pragma solidity ^0.4.8;

contract attendance {
    mapping (bytes32 => mapping (bytes32 => string)) public tranlog;

    function setTransaction(bytes32 name, bytes32 date,
    string atten_data) {
        if(bytes(tranlog[name][date]).length != 0) {
            throw;
        }
        tranlog[name][date] = atten_data;
    }

    function getTransaction(bytes32 name, bytes32 date)
        constant returns (string atten_data) {
        return tranlog[name][date];
    }
}
```

ハッシュ化した name と date の他に文字列を atten\_data として保存する．setTransaction で任意の値を設定し，getTransaction で情報を取得する．

実行方法は以下に記載するが 5.2.2 と大きく異なる点はない．

1. 任意の場所にコントラクトプログラムを配置し，コントラクトプログラムビルド用のデータを出力する．

```
$ solc -o ./ --bin --optimize attendance.sol
$ cat attendance.bin
6060604052341561000f57600080fd5b60405161046d38038061.....
.....61e4c1bf9257fd6542dbf1628bf82a571b89769cc4c11f5be0029
```

2. コントラクトの情報を取得する．

```
$ solc --abi attendance.sol
[{"constant":true,"inputs":[],"name":"say","outputs":[{"name":
"", "type":"string"}],"payable":false,"stateMutability":"view",
(割愛)e,"stateMutability":"nonpayable","type":"constructor"}]
```

3. Geth を起動する．

```
$ geth --networkid 4649 --nodiscover --maxpeers 0 --datadir /home/
ubuntu/data_testnet console 2>> /home/ubuntu/data_testnet/geth.log
```

4. コントラクト登録者のロックを解除する .

```
> personal.unlockAccount(eth.accounts[0])
Unlock account 0x288443de7c192ac7adf6fb951d7187622309df50
Passphrase:
true
```

5. コントラクトをブロックチェーンに登録する .

```
> AttenContract = web3.eth.contract([{"constant":true,"inputs":[{"
(割愛) payable":false,"stateMutability":"view","type":"function"}]);
```

6. 0x をデータ部に入れて 16 進数であることを明確にする .

```
> Atten = AttenContract.new({from: eth.accounts[0], data: '0x6060
604052341561000f57600080fd5b60 (割愛) 161046d380aa7098a17
57fd6542dbf1628bf82a571b89769cc4c11f5be0029', gas: 3000000},
function(e, contract){console.log(e, contract); if (typeof contract.
address != 'undefined') { console.log('Contract mined! address: ' +
contract.address + ' transactionHash: ' + contract.transaction
Hash) ; }})
```

7. マイニングを開始する .

```
> miner.start()
null
> null [object Object]
Contract mined! address: 0x77913d7e41ab1bbae59bf2a0d07492
4f50b01adb transactionHash: 0x0e3cf64b29132e843b23d5591fd
13e0d3684ae897fc09b5ed46b871fa514153a
```





### 5.3.2 出欠情報の管理

任意の値を用いて登録が可能な経歴情報を管理するプログラムを作成した . person.sol として作成し , プログラムの内容を以下に記載する .

```
pragma solidity ^0.4.8;

contract person {
    address admin;

    struct AppDetail {
        bool allowReference;
        uint256 approveBlockNo;
        uint256 refLimitBlockNo;
        address applicant;
    }

    struct PersonDetail {
        string name;
        string birth;
        address[] orglist;
    }

    struct OrganizationDetail {
        string name;
    }

    mapping(address => AppDetail) appDetail;

    mapping(address => PersonDetail) personDetail;

    mapping(address => OrganizationDetail) public orgDetail;

    function PersonCertification() {
        admin = msg.sender;
    }

    function setPerson(string _name, string _birth) {
        personDetail[msg.sender].name = _name;
        personDetail[msg.sender].birth = _birth;
    }

    function setOrganization(string _name) {
        orgDetail[msg.sender].name = _name;
    }

    function setBelong(address _person) {
        personDetail[_person].orglist.push(msg.sender);
    }

    function setApprove(address _applicant, uint256 _span) {
        appDetail[msg.sender].allowReference = true;
        appDetail[msg.sender].approveBlockNo = block.number;
        appDetail[msg.sender].refLimitBlockNo = block.number + _span;
        appDetail[msg.sender].applicant = _applicant;
    }
}
```

```

function getPerson(address _person) public constant returns(
    bool _allowReference,
    uint256 _approveBlockNo,
    uint256 _refLimitBlockNo,
    address _applicant,
    string _name,
    string _birth,
    address[] _orglist) {
    _allowReference = appDetail[_person].allowReference;
    _approveBlockNo = appDetail[_person].approveBlockNo;
    _refLimitBlockNo = appDetail[_person].refLimitBlockNo;
    _applicant = appDetail[_person].applicant;
    if ((msg.sender == _applicant)
        && (_allowReference == true)
        && (block.number < _refLimitBlockNo))
    || (msg.sender == admin)
    || (msg.sender == _person)) {
        _name = personDetail[_person].name;
        _birth = personDetail[_person].birth;
        _orglist = personDetail[_person].orglist;
    }
}
}

```

実行方法を以下に記載する．複数のアカウントを用い，異なるアカウントで Geth を起動し変数を回すため，コマンドラインの他に並行して TeraTerm などの ssh クライアントを用いて作業を行いたいのが良い．

本論において作成したアカウントを次に一覧としておく．

- eth.accounts[0]  
0x288443de7c192ac7adf6fb951d7187622309df50
- eth.accounts[1]  
0xc5906348828cc1d0cfb8249d7e7b622ef8d77913
- eth.accounts[2]  
0x5d91af951dcf13690ae9448bfa264d91e3c7f740
- eth.accounts[3]  
0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40
- eth.accounts[4]  
0xac8952ec60e771e11a484a5ecab9fbfd181d358a

1. Contract のアカウントを合計で 5 つ作成する . パスフレーズとアドレスを忘れないよう記載しておく .

```
> personal.newAccount("pass0")
"0x288443de7c192ac7adf6fb951d7187622309df50"
> personal.newAccount("pass1")
"0xc5906348828cc1d0cfb8249d7e7b622ef8d77913"
> personal.newAccount("pass2")
"0x5d91af951dcf13690ae9448bfa264d91e3c7f740"
> personal.newAccount("pass3")
"0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40"
> personal.newAccount("pass4")
"0xac8952ec60e771e11a484a5ecab9fbfd181d358a"
```

2. コントラクトプログラムビルド用のデータ部を出力する .

```
$ solc -o ./ --bin --optimize person.sol
$ cat person.bin
6060604052341561000f57600080fd5b60405161046d38038061.....
.....61e4c1bf9257fd6542dbf1628bf82a571b89769cc4c11f5be0029
```

3. コントラクトの情報を取得する .

```
$ solc --abi person.sol
[{"constant":false,"inputs":[{"name":"_name","type":"string"}],
( 割愛 )orgDetail","outputs":[{"name":"name","type":"string"}],
"payable":false,"stateMutability":"view","type":"function"}]
```

4. TeraTerm など外部の ssh クライアントから Geth を起動する .

```
$ geth --datadir /home/ubuntu/data_testnet --networkid 15
--mine --minerthreads=1 --etherbase=0x288443de7c192a
c7adf6fb951d7187622309df50 --rpc --rpcport 8545 --rpcaddr
"0.0.0.0" --rpccorsdomain "*" --rpcapi "admin,db,eth,debug,
miner,net,shh,txpool,personal,web3"
```

5. Geth をコマンドラインなどから Attach する .

```
$ geth attach rpc:http://localhost:8545 console
```

6. コントラクトをブロックチェーンに登録する .

```
> PersonContract = web3.eth.contract([{"constant":true,"inputs":[{"
( 割愛 )payable":false,"stateMutability":"view","type":"function"}]);
```

7. 0x をデータ部に入れて 16 進数であることを明確にする .

```
> Person = PersonContract.new({from: eth.accounts[0], data: '0x6060
604052341561000f57600080fd5b60 (割愛) 161046d380aa7098a17
50bc0b06d4acb0fcd2c9575304ae5eade2e4aaf5b', gas: 30000000},
function(e, contract){console.log(e, contract); if (typeof contract.
address != 'undefined') { console.log('Contract mined! address: ' +
contract.address + ' transactionHash: ' + contract.transaction
Hash) ; }})
> null [object Object]
Contract mined! address: 0xaa64c4842108bb6db68e2b88ec8cef54
85e0bfa2 transactionHash: 0xc9ef23f545cb43eb20aca3c0f8d2208
c9197a9c91012f741bd2566f7dea4aa73
```

8. コントラクトにアクセスするための変数を定義する .

```
> contractObj = eth.contract(Person.abi).at(Person.address)
```

9. eth.accounts[1] で Geth を起動する . ssh クライアントで実行中の Geth を停止し , 新たに起動する . コマンドラインで exit する必要はない .

```
$ geth --datadir /home/ubuntu/data_testnet --networkid 15
--mine --minerthreads=1 --etherbase=0xc5906348828cc1
d0cfb8249d7e7b622ef8d77913 --rpc --rpcport 8545 --rpcaddr
"0.0.0.0" --rpccorsdomain "*" --rpcapi "admin,db,eth,debug,
miner,net,shh,txpool,personal,web3"
```

10. 組織情報を入力する .

```
> personal.unlockAccount(web3.eth.accounts[1])
> contractObj.setOrganization.sendTransaction("chiba institute of
technology",{from:eth.accounts[1]})
> contractObj.orgDetail.call("0xc5906348828cc1d0cfb8249d7e7b62
2ef8d77913",{from:eth.accounts[1]})
"chiba institute of technology"
```

11. 記録者が通学していた実績を登録する .

```
> contractObj.setBelong.sendTransaction("0xac8952ec60e771e11a
484a5ecab9fbfd181d358a",{from:eth.accounts[1]})
```

12. eth.accounts[2] で Geth を起動する . ssh クライアントで実行中の Geth を停止し , 新たに起動する . コマンドラインで exit する必要はない .

```
$ geth --datadir /home/ubuntu/data_testnet --networkid 15
--mine --minerthreads=1 --etherbase=0x5d91af951dcf136
90ae9448bfa264d91e3c7f740 --rpc --rpcport 8545 --rpcaddr
"0.0.0.0" --rpccorsdomain "*" --rpcapi "admin,db,eth,debug,
miner,net,shh,txpool,personal,web3"
```

13. 組織情報を入力する .

```
> personal.unlockAccount(web3.eth.accounts[2])
> contractObj.setOrganization.sendTransaction("C.I.T Service
",{from:eth.accounts[2]})
> contractObj.orgDetail.call("0x5d91af951dcf13690ae9448bf
a264d91e3c7f740",{from:eth.accounts[2]})
"C.I.T Service"
```

14. 記録者が通学していた実績を登録する .

```
> contractObj.setBelong.sendTransaction("0xac8952ec60e771e11a
484a5ecab9fbfd181d358a",{from:eth.accounts[2]})
```

15. eth.accounts[4] で Geth を起動する . ssh クライアントで実行中の Geth を停止し , 新たに起動する . コマンドラインで exit する必要はない .

```
$ geth --datadir /home/ubuntu/data_testnet --networkid 15
--mine --minerthreads=1 --etherbase=0xac8952ec60e771e
11a484a5ecab9fbfd181d358a --rpc --rpcport 8545 --rpcaddr
"0.0.0.0" --rpccorsdomain "*" --rpcapi "admin,db,eth,debug,
miner,net,shh,txpool,personal,web3"
```

16. 本人情報を登録する .

```
> personal.unlockAccount(web3.eth.accounts[4])
> contractObj.setPerson.sendTransaction("Taro YAMADA",
"19850101",{from:eth.accounts[4]})
>contractObj.getPerson.call("0xac8952ec60e771e11a484a5ecab
9fbfd181d358a",{from:eth.accounts[4]})
[false, 0, 0, "0x00000000000000000000000000000000",
"Taro YAMADA", "19850101",
["0xc5906348828cc1d0cfb8249d7e7b622ef8d77913",
"0x5d91af951dcf13690ae9448bfa264d91e3c7f740"]]
```

17. 記録者が経歴を見る人に閲覧許可を出す .

```
> contractObj.setApprove.sendTransaction("0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40",250,{from:eth.accounts[4],
gas:3000000})
> contractObj.getPerson.call("0xac8952ec60e771e11a484a5ecab9fbfd181d358a",{from:eth.accounts[4]})
[true, 1134, 1384, "0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40","Taro YAMADA", "19850101",
["0xc5906348828cc1d0cfb8249d7e7b622ef8d77913",
"0x5d91af951dcf13690ae9448bfa264d91e3c7f740"]]
```

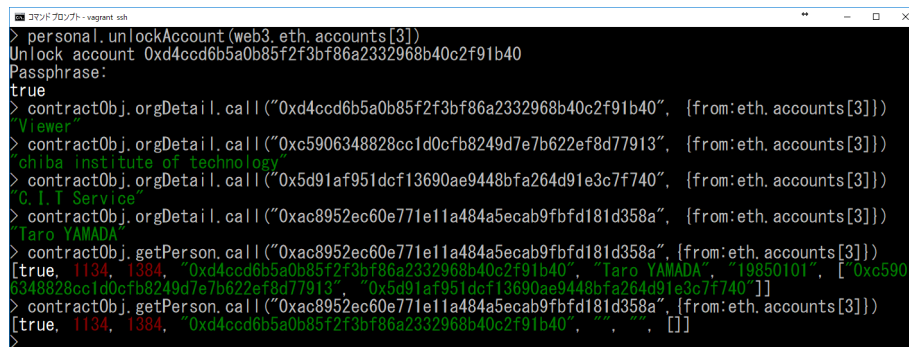
18. eth.accounts[3] で Geth を起動する . ssh クライアントで実行中の Geth を停止し , 新たに起動する . コマンドラインで exit する必要はない .

```
$ geth --datadir /home/ubuntu/data_testnet --networkid 15
--mine --minerthreads=1 --etherbase=0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40 --rpc --rpcport 8545 --rpcaddr
"0.0.0.0" --rpccorsdomain "*" --rpcapi "admin,db,eth,debug,
miner,net,ssh,txpool,personal,web3"
```

19. 本人確認情報を閲覧する .

```
> contractObj.getPerson.call("0xac8952ec60e771e11a484a5ecab9fbfd181d358a",{from:eth.accounts[4]})
[true, 1134, 1384, "0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40","Taro YAMADA", "19850101",
["0xc5906348828cc1d0cfb8249d7e7b622ef8d77913",
"0x5d91af951dcf13690ae9448bfa264d91e3c7f740"]]
```

閲覧情報の取得が成功すると図 5.9 のように表示される . 閲覧期限内に情報を取得しようとした場合には , 本人の氏名と生年月日が正しく表示される . 閲覧期限が切れた場合に情報を取得しようとした場合には確認できないという結果も得ることが出来た .



```
コンソールプロンプト - vagrant ssh
> personal.unlockAccount(web3.eth.accounts[3])
Unlock account 0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40
Passphrase:
true
> contractObj.orgDetail.call("0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40", {from:eth.accounts[3]})
"Viewer"
> contractObj.orgDetail.call("0xc5906348828cc1d0cfb8249d7e7b622ef8d77913", {from:eth.accounts[3]})
"chiba institute of technology"
> contractObj.orgDetail.call("0x5d91af951dcf13690ae9448bfa264d91e3c7f740", {from:eth.accounts[3]})
"C.I.T. Service"
> contractObj.orgDetail.call("0xac8952ec60e771e11a484a5ecab9fbfd181d358a", {from:eth.accounts[3]})
"Taro YAMADA"
> contractObj.getPerson.call("0xac8952ec60e771e11a484a5ecab9fbfd181d358a", {from:eth.accounts[3]})
[true, 1134, 1384, "0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40", "Taro YAMADA", "19850101", ["0xc5906348828cc1d0cfb8249d7e7b622ef8d77913", "0x5d91af951dcf13690ae9448bfa264d91e3c7f740"]]
> contractObj.getPerson.call("0xac8952ec60e771e11a484a5ecab9fbfd181d358a", {from:eth.accounts[3]})
[true, 1134, 1384, "0xd4ccd6b5a0b85f2f3bf86a2332968b40c2f91b40", "", "", []]
```

図 5.9 登録された経歴情報の結果

## 第 6 章

# 考察

ブロックチェーンを用いた存在証明を行なうプログラムをスマート・コントラクトにて構築することが出来た．存在証明が必要となるドキュメントを管理することが多い PM 学科内において，幅広く利用する価値があるのではないかと考えた．

電子記録の存在証明はプロジェクト内の成果物において利用することも重要であるため，構築したプロトタイプの強化も有用と考える．

## 第 7 章

# 結論

証明が必要となるドキュメントをブロックチェーンで管理することで、改ざんを複雑化しデータの信頼性を向上させることが出来た。

マネジメントに応用する点で独自性が低いため、具体的に利用する内容の検討が必要である。PMBOKなどを参考にしながら存在証明を電子的に行なうべきである成果物やその他の情報を洗い出すことによって、今回作成したプロトタイプモデルの更なる有効的な活用法を見つけることが出来るだろう。



## 参考文献

- [1] 東洋経済新聞社. フィンテックで何が起こるか知っていますか. <http://toyokeizai.net/articles/-/166765> (2018.01.19 閲覧).
- [2] 北田淳. ブロックチェーンは世界を変える. In *WIRED VOL.25*, pp. 54–55. コンデナスト・ジャパン, 2016.
- [3] Colony. Colony.io. <https://colony.io/> (2017.10.10 閲覧).
- [4] Chocolatey. Chocolatey - the package manager for windows. <https://chocolatey.org/> (2017.10.10 閲覧).
- [5] Virtualbox. Oracle vm virtualbox. <https://www.virtualbox.org/> (2017.10.10 閲覧).
- [6] Vagrant. Vagrant by hashicorp. <https://www.vagrantup.com/> (2017.10.10 閲覧).
- [7] Ubuntu. Homepage | ubuntu japanese team. <https://www.ubuntulinux.jp/> (2017.10.10 閲覧).
- [8] Docker. Docker - build, ship, and run any app, anywhere. <https://www.docker.com/> (2017.10.10 閲覧).
- [9] NaiveChain. Naivechain - a blockchain implementation in 200 lines of code. <https://github.com/lhartikk/naivechain> (2017.10.10 閲覧).
- [10] Ethereum. Ethereum project. <https://www.ethereum.org/> (2017.10.10 閲覧).

# 謝辞

本研究を進めるにあたって，ご指導を頂いた卒業論文指導教員の矢吹太朗准教授を始め，プロジェクトマネジメント学科の堀内俊幸教授，下田篤教授，田隈広紀准教授に感謝致します．矢吹准教授には研究外にも多くのことを教えていただきました．

また，研究や日常の情報交流の場を通して多くの知識と方法などを教えていただいた矢吹研究室の皆様，その他本研究や大学生活全体を通してかかわっていただいた皆様にも感謝の意をここに表します．