

# 'Mastering Bitcoin' への賛辞

「私が一般の人にBitcoinについて話すと、『で、実際どうやって動いているの?』と聞かれることがよくある。今や私は、その質問への素晴らしい回答を手にした。なぜなら、*Mastering Bitcoin* を読めば誰でも、Bitcoinがどのように動いているかについて深い理解に達し、次世代のわくわくするような暗号通貨のアプリケーションを作るのに十分な力が身に付くからだ」

— Gavin Andresen, Chief Scientist Bitcoin Foundation

「Bitcoinとブロックチェーン技術は、次世代のインターネットの基本的な構成要素になりつつある。シリコンバレーの"ベスト・アンド・ブライテスト"がこれに取り組んでいる。Andreasの著作は、この金融の世界のソフトウェア革命に加わるための、良い助けとなるだろう」

— Naval Ravikant, Co-founder AngelList

「*Mastering Bitcoin* は、Bitcoinに関して今日手に入る、技術面で最高の文献だ。Bitcoinは、2010年代の最も重要な技術として振り返られるようになるだろう。本書は、開発に携わる者にとって、特にBitcoinプロトコルでのアプリケーション構築に関心がある者にとって、究極の必読図書である。強くお勧めする」

— Balaji S. Srinivasan (@balajis), General Partner; Andreessen Horowitz

「Bitcoinブロックチェーンの発明は、インターネットそのものと同じくらい幅広く多様性に富んだ生態系を実現する、全く新たなプラットフォームが打ち立てられたことを意味している。卓越した思想家の一人であるAndreas Antonopoulosは、本書を書く人間として完璧な人選である」

— Roger Ver, Bitcoin Entrepreneur and Investor

# 目次

# 序文

## Bitcoinの本を書くにあたって

私は、Bitcoinに2011年の中頃に偶然出会いました。その時の私の反応は、「ふふっ！オタクの通貨じゃないか！」というもので、それ以上でも以下でもありませんでした。そしてその後6ヶ月間、その重要性を理解することなく無視していました。多くの賢い人々がこのような反応をすることを繰り返し見てきましたが、このことは私にいくばくかの慰めを与えてくれます。私は、メーリングリストでのディスカッションで2度目にBitcoinに出会った時、Satoshi

Nakamotoによって書かれたホワイトペーパーを読むことにしました。正式な情報源から学んで、それが何であるのかを知るためにです。9ページを読み終えた瞬間のことを、未だに覚えています。私は、Bitcoinが単なるデジタル通貨ではなく、通貨を含む様々なものに基盤を提供する信用のネットワークなのだ、とその時に理解したのです。"これはお金ではない。これは分散型の信用ネットワークなのだ"という認識に至ってから4ヶ月間、私は見つけられる限りどんな細かいBitcoinの情報をも貪り読みました。私は取り憑かれ、魅了されました。そして毎日12時間以上、可能な限り、画面に食いつき、読み、書き、プログラムをし、学びました。熱狂状態から戻って来た時には、それまでまともに食事も取らなかったのが9キロ以上も痩せていていましたが、Bitcoinに専念することに決めました。

2年後、Bitcoin関連のサービスやプロダクトを調べるための小さなスタートアップをいくつか作った後に、まさに今が私の最初の本を書く時だ、と決心しました。Bitcoinは、私を創造力の狂乱へ導き、頭の中を一杯に埋め尽くしたテーマです。そして、Bitcoinはインターネット以降に出会った中で、最もエキサイティングなテクノロジーです。より多くの人と、この素晴らしいテクノロジーへの情熱を共有する時でした。

## 想定している読者

この本は、主にプログラマ向けに書かれています。もしプログラム言語を使えるのであれば、どのように暗号通貨が動くのか、どのように暗号通貨を利用するのか、どのように暗号通貨のソフトウェアを開発するのかかわかるでしょう。最初のいくつかの章は、Bitcoinや暗号通貨の内部の動きを理解したいものの、プログラマではない方向けのBitcoin入門として最適でしょう。

## この本で使用される表記規約

この本では以下の表記規約を使用します。

### イタリック

新しい用語、URL、Eメールアドレス、ファイル名、ファイル拡張子を表す

### 等幅(*Constant width*)

プログラムを表示するときに使用され、また変数や関数名、データベース、データタイプ、環境変数、ステートメント、キーワードなどプログラムの一要素を表現するときにも使用される

### 等幅太字(**Constant width bold**)

ユーザに文字通り入力されるべきコマンドやその他テキストを表す

### 等幅イタリック(*Constant width italic*)

ユーザ側の環境や文脈によって変わる値によって置き換えられるべきテキストを表す

**TIP** | このアイコンは、ヒントや提案、一般的な示唆を表します。

**WARNING** | このアイコンは、警告や注意を表します。

## コード例

例はPythonやC++で説明されており、LinuxまたはMac OS Xのような Unixライクなオペレーティングシステムのコマンドラインを使って実行できます。全てのコードスニペットは [GitHub repository](#) のメインリポジトリの `code` ディレクトリにあります。この本にあるコードをフォークして、コード例を試してみてください。そして、修正点があればGitHubを通してご連絡をお願いします。

すべてのコードスニペットは、ほとんどのオペレーティングシステムにおいて、対応する言語のコンパイラとインタプリタを最小限インストールすることで、置き換えられます。本書では、必要に応じて、基本的なインストールの命令と、その命令のアウトプットについて順を追った例を提供します。

コードスニペットやコードアウトプットには、紙上で読むのに適するよう、ある行がバックスラッシュ(\)で分けられて改行されているものがあります。読者がこれらを書き移すときは、バックスラッシュを除いて行をつなげば、この本で示した通りの正しい結果を得られます。

すべてのコードスニペットは、読者が本書と同じ計算をすれば同じ結果となることを確認できるよう、できる限り現実の数値と計算を使用しています。例えば、秘密鍵と、これと対応する公開鍵およびアドレスは、すべて現実に存在するものです。本書に掲載されたトランザクション、ブロック、ブロックチェーンの事例は、実際のBitcoinのブロックチェーンに存在しており、公開された台帳の一部ですので、読者は実際にこれらを確認することができます。

## 謝辞

この本は、多くの人の努力と献身によってできています。暗号通貨とBitcoinについての技術的な本を書くことに協力してくださった友人、同僚、そして面識がないにも関わらず協力してくださった方々に感謝しています。

Bitcoinのテクノロジーと、Bitcoinのコミュニティを切り分けて考えることは不可能です。そしてこの本はそのコミュニティによる本でもあります。私の本書への取り組みは、まさに本を書き終わる時まで、Bitcoinのコミュニティ全体から励まされ、応援され、支えられ、称賛を頂きました。これは何よりもかけがえのないことであり、この本を書くことで私は2年間その素晴らしいコミュニティの一部でいることができました。私を受け入れてくれたこのコミュニティに対して私はいくら感謝してもしきれません。そして名前をあげるにはあまりにも多くの方々の支え、例えばカンファレンスやイベント、セミナー、ミートアップ、ピザを食べる集まり、小さな個人的な集まり、Twitter、Reddit、bitcointalk.orgで私と交流して下さった方々がいらっしゃいます。あらゆるアイデアやアナロジー、質問、回答、そして貴方がこの本でご覧になった説明のいくつかは、私のコミュニティとの交流の中でインスパイアされ、検証され、改善されてきました。支えて下さった皆様ありがとうございます。あなた方無しにこの本は完成し得なかった。私の一生の喜びです。

勿論、最初の本の著者となるずっと前から著者となるための道のりは始まっていました。私の母語(と学校教育)はギリシャ語で、それ故大学の最初の年に英語の筆記をより良いものにするためのコースを取らなければなりませんでした。Diana

Kordasは私の英語筆記の先生で自信とスキルを付ける手伝いをしてくださいました。そしてプロフェッショナルとして私は、データセンターについての技術的なライティングのスキルを磨き、そして *Network World* magazineに寄稿しました。私はJohn Dix と、John Gallant に感謝しています。彼らは私に *Network World*

にて初めての仕事をくれ、そして編集者のMichael Cooney と同僚の Johna Till Johnson は私の原稿を編集して出版できるものにしてくれました。500 の言葉を1週間の間に書きそれを4年間続けたことは、私が遂に著者になるために十分な経験を与えてくれました。Jean de Vera は、私が執筆者となることを早い頃から促してくれ、私が自分の中に既に本を持っているということを信じ、そのように言ってくれました。

私がこの本をO'Reillyに提案する際に、推薦をしてくれたり、プロポーザルのレビューをしてくれたりして、私をサポートして下さった方々にも御礼申し上げます。とりわけ、John Gallant, Gregory Ness, Richard Stiennon, Joel Snyder, Adam B. Levine, Sandra Gittlen, John Dix, Johna Till Johnson, Roger Ver, Jon Matonisに、感謝しています。Richard Kagan、Tymon Mattoszkolaは、早期のプロポーザルをレビューしてくれました。Matthew Owain Taylorは、プロポーザルの編集をしてくれました。

Cricket Liu (O'Reilly title `_DNS and BIND_`の著者) は、私にO'Reilly を紹介してくれました。Michael Loukides と、O'ReillyのAllyson MacDonaldは、幾月もこの本を実現するために手伝ってくれました。締め切りが過ぎて、私達が予定していたスケジュールから納品が遅れた時にも、Allysonは極めて忍耐強く待ってくれました。

最初の幾つかの章のドラフトが、最も大変でした。なぜならば、Bitcoinは分かりやすく説明することが難しいからです。bitcoinの技術に関して、ある話題をひとつ扱おうとすると、全ての内容が関わってくるということが常でした。私は、何とかこのトピックを簡単に理解できるようにできないかと格闘し、何度もつまずき落ち込みながら、高度に技術的なテーマについての物語を作りました。最終的に、Bitcoinを実際に使う人々を通じて、Bitcoinの物語を語ることにしたことで、本書全体がとても書きやすくなりました。私の友人でありメンターでもあるRichard Kaganに感謝します。彼は、物語を分かりやすくし、行き詰まりを乗り越えることを手伝ってくれました。Pamera

Morganさんは、早い段階のドラフトのレビューをしてくれ、この本をより良いものにするために難しい質問をしてくれました。San Francisco Bitcoin Developer Meetup Groupと、同グループの共同創設者Taariq Lewisは、早い段階で内容を検証して下さいました。

私は、早い段階のドラフトをGithub上に公開してコメントをもらいつつ、この本を執筆しました。何百ものコメント、提案、訂正、支援を頂きました。こうした支援に対するお礼は、私の感謝と共に [初期にリリースしたドラフト \(Github上での貢献\)](#) に述べられています。特にMinh T. Nguyenは、Github上の貢献をボランティアで管理してくれ、彼自身が多大な貢献をもたらしてくれました。Andrew Nauglerは、インフォグラフィックデザインを担当してくれました。

ドラフトの完成後、技術的なレビューを全体を通して何度も行いました。Cricket LieとLorne Lantxの全てのレビューとコメント、そして支援に感謝します。

何人かのBitcoin開発者の方は、コードのサンプル、レビュー、コメント、そして励ましの言葉をくださいました。Amir Taaki、Erik Voskuilは、幾つかのコードのスニペットと、多くの重要なコメントをくださいました。Vitalik Buterin、Richard Kissは、楕円曲線とコードについて手伝ってくださいました。Gavin Andresenは、訂正とコメントと励ましの言葉をくださいました。Michalis Kargakisは、コメントと貢献とbtcdの記事を、Robin Ingeは、第二版ための誤記の訂正をして頂きました。

私の母、Theresaのおかげで、言葉や本に愛着を持つことができました。母は、壁一面に本が並ぶ家で私を育

ててくれました。母は、自称ハイテク恐怖症であるにもかかわらず、1982年に初めてのコンピューターを買ってくれました。私の父、Manelaosは、80歳の時に初の著書を出版した市井のエンジニアでしたが、私に、論理的で分析的な思考、科学とエンジニアリングへの愛を教えてくださいました。

この旅を支えて下さったすべての方々に、感謝します。

## 初期にリリースしたドラフト（**Github**上での貢献）

多くの方々が、コメントや、訂正や、加筆をGithub上の初期のドラフトに寄せてくださいました。この本に貢献して下さったすべての皆様に感謝します。Github上で多大な貢献をして下さった方々は、以下の通りです。カッコ内は、Github IDです。

- Minh T. Nguyen, GitHub contribution editor (enderminh)
- Ed Eykholt (edeykholt)
- Michalis Kargakis (kargakis)
- Erik Wahlström (erikwam)
- Richard Kiss (richardkiss)
- Eric Winchell (winchell)
- Sergej Kotliar (ziggamon)
- Nagaraj Hubli (nagarajhubli)
- ethers
- Alex Waters (alexwaters)
- Mihail Russu (MihailRussu)
- Ish Ot Jr. (ishotjr)
- James Addison (jayaddison)
- Nekomata (nekomata-3)
- Simon de la Rouviere (simondlr)
- Chapman Shoop (belovachap)
- Holger Schinzel (schinzelh)
- effectsToCause (vericoïn)
- Stephan Oeste (Emzy)
- Joe Bauers (joebauers)
- Jason Bisterfeldt (jbisterfeldt)
- Ed Leafe (EdLeafe)

===オープン版

この本は、<http://creativecommons.org/licenses/by-sa/4.0/>[Creative Commons Attribution Share-Alike

License (CC-BY-SA) ]の下で、翻訳のために発行された"Mastering Bitcoin"のオープン版です。このライセンスは、以下の条件を満たす方に対し、本書または本書の一部について、読み、共有し、複製し、印刷し、販売し、再利用することを許可するものです。

- 同じライセンス(Share-Alike)を適用すること
- 出典を明記すること

===出典

Mastering Bitcoin by Andreas M. Antonopoulos LLC <https://bitcoinbook.info>

Copyright 2016, Andreas M. Antonopoulos LLC

===翻訳

あなたが英語以外の言語で、この本を読んでいるとしたら、それはボランティアによって翻訳されたものです。以下の人々が、翻訳に貢献しました。

- 今井 崇也(Takaya Imai) 日本語翻訳リーダー(Translation Leader) Email: [takaya.imai@frontier-ptnrs.com](mailto:takaya.imai@frontier-ptnrs.com)
- 鳩貝 淳一郎(Junichiro Hatogai)
- Jonathan Underwood, Tomoaki Sato, Akira Mitani

## ==用語解説

この用語解説では、Bitcoinに関連して使われる用語の多くを解説しています。これらの用語はこの本全体を通じて使われますので、クイックリファレンスとしてブックマークしてください。

### アドレス

Bitcoinアドレスとは、`1DSrfjdB2AnWaFNgSbv3MZC2m74996JafV` といった、「1」から始まる文字と数字の連なりです。ユーザーは、emailを自分のemailアドレスに送るよう誰かに依頼するのと同じように、bitcoinを自分のBitcoinアドレスに送るよう依頼します。

### *bip*

Bitcoin改善提案(Bitcoin Improvement Proposals)の略称で、BitcoinコミュニティのメンバーがBitcoinを改善するために提出してきた一連の提案のことを指します。例えば、BIP0021はbitcoin uniform resource identifier (URI) スキームを改善するための提案です。

### *bitcoin* または *Bitcoin*

通貨単位または通貨そのもの(bitcoin)、ネットワークおよびソフトウェアの総称(Bitcoin)。

### ブロック

グループにまとめられたトランザクションのことで、タイムスタンプとひとつ前のブロックのフィンガープリントが刻印されています。ブロックヘッダのハッシュ値が求められることでproof of workが作られ、それによってトランザクションが検証されます。検証されたブロックは、ネットワークにおける合意によりブロックチェーンに加えられます。

### ブロックチェーン

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

### 承認(*confirmation*)

トランザクションがブロックに含まれると、承認数は1となります。同じブロックチェーンにおいて、そのブロックにもうひとつのブロックが繋がるとすぐに、トランザクションの承認数は2となります。6またはそれ以上の承認数があることは、トランザクションが覆されない十分な証拠とみなされます。

### *difficulty*

proof of workを作るために必要な計算量を制御する、ネットワーク全体に適用される設定のこと。

### *difficulty*ターゲット

ネットワークにおける計算によって、概ね10分毎に新たなブロックが加わるような、*difficulty*の設定のこと。

### *difficulty retargeting*

ネットワーク全体にわたる*difficulty*の再計算のことを指します。2,106ブロック毎に一度、直前の2,106ブロックにおけるハッシュ値を算出するパワーを考慮して、再計算を行います。

### 手数料

トランザクションの送り手は、そのトランザクションの処理のために、ネットワークへの手数料をしばしば追加します。ほとんどのトランザクションは、0.5mBTCという最少の手数料で処理されます。



## ハッシュ

二進法の入力に対する、デジタルなフィンガープリントのこと。

## genesisブロック

この暗号通貨を起動するのに使われた、ブロックチェーンにおける最初のブロック。

## マイナー

ハッシュ値の算出を繰り返すことで、新しいブロックのための有効なproof of workを見つけ出すネットワークノード。

## ネットワーク

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

## Proof-Of-Work

見つけるのに相当量の計算を要するデータ。Bitcoinにおいてマイナーは、ネットワーク全体に設定されたdifficultyターゲットを満たすSHA256アルゴリズムに対し、解を見つけなければなりません。

## 報酬

新たなブロックのproof-of-workとなる解を発見したマイナーに対し、ネットワークから払われる報酬のこと。報酬は当該ブロックに含まれ、現在の金額は1ブロックに対し25BTCとなっています。

## 秘密鍵

紐づけられたアドレスに送られたbitcoinを解錠するための秘密の番号で、+5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh+といった形をとります。

## トランザクション

bitcoinをあるアドレスからほかのアドレスに送ること。より正確に言えば、トランザクションとは、価値の転移を表した、署名されたデータ構造です。トランザクションは、Bitcoinネットワークに伝えられ、マイナーによって集められ、ブロックにまとめられ、ブロックチェーンに固定されます。

## ウォレット

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

# イントロダクション

## Bitcoinとは

Bitcoinはデジタルマネーエコシステムの基礎となるコンセプトと技術の集合体です。Bitcoinネットワークの参加者の間で、価値の保有と輸送がbitcoinという名の通貨単位で行われます。ユーザー間の通信は主にBitcoinプロトコルに基づいて、インターネットを通じて行われます。Bitcoinのプロトコルスタックはオープンソースとして利用可能であり、ノートパソコンからスマートフォンまで、様々なデバイス上で動作します。

ユーザーはネットワークを通じてbitcoinをやり取りすることで、従来の通貨で行うほぼ全てのこと、つまり物品の売買から個人や組織への送金、融資まで行うことが可能になります。bitcoin自体も売買が可能であり、専門の両替機関で他の通貨とも両替することも可能です。bitcoinは高速かつ安全であり、国境を越えて取引が可能であることから、ある意味でインターネットに使うための最適な通貨ともいえます。

伝統的な通貨と異なりbitcoinは完全に仮想的なものです。物理的なコインは存在せず、またデジタルコイン自体が存在するわけでもありません。コインは「送信者から受信者へある一定量の額面を移動させる」という取引(トランザクション)の中で暗に示されるものです。

Bitcoinのユーザーはトランザクションの所有権を証明する鍵を所有し、そのトランザクションの中に記載された額面を使用したり新しい所有者に送金することができます。この鍵はそれぞれの利用者のPC上の電子財布(ウォレット)に保持されます。この鍵の保有がbitcoinを使用する唯一の条件であり、そのコントロールは各ユーザーにゆだねられています。

Bitcoinは分散されたpeer-to-peerのシステムであり、何らかの"中央"サーバや管理者が存在するものではありません。bitcoinは取引の過程で行われる"マイニング"と呼ばれる数学的な解を見つけ出す競争により新たに生み出されます。どの(フルプロトコルスタックを動作させている)Bitcoinネットワーク参加者も、自身のコンピューターリソースを用いて取引の記録処理と検証処理を行うことでマイナーとなることができます。平均して10分に1回の頻度で誰かが数学的な解を見つけることで取引が検証され、その解の発見者にはbitcoinが新たに与えられます。つまるところbitcoinのマイニングとは、中央銀行が行う必要があった通貨の発行と決済の機能を、世界的な競争で代替したものなのです。

Bitcoinプロトコルにはマイニングの機能を規定するアルゴリズムが組み込まれています。マイナーが行わなければならないタスクの難易度は、マイナーの数(またはCPUの数)が変動しても平均的に10分に1回ほど解が見つかるように自動的に調整されます。またプロトコルではbitcoinが新たに発行される頻度は4年毎に半減されるように規定されており、またbitcoinの総発行量は2100万bitcoinを超えないように規定されています。結果、bitcoinの流通数は容易に予想可能で2140年に2100万に到達するカーブを描くこととなります。長期的にbitcoinの発行レートが減少していくため、bitcoin通貨はデフレーション傾向となります。さらに、予期された通貨発行レートよりも多く通貨が"発行"されることがないため、インフレーション状態になることはありません。

Bitcoinはプロトコル名でもありネットワーク名でもあり、さらには分散コンピューティングのイノベーションの名称でもあります。通貨としてのbitcoinはこのイノベーションの単なる最初の応用であるだけです。開発者の視点から、Bitcoinを通貨のインターネット、つまり分散コンピューティングによって価値やデジタル資産の所有権のセキュアなやりとりを行うためのネットワーク基盤と考えています。Bitcoinは見た目よりも大きな可能性にあふれているのです。

この章では主な概念と用語を説明することから始め、必要なソフトウェアを用意し、単純な取引の中でbitcoinを使ってみます。その後の章でBitcoinが動作する技術的な部分を明らかにし、Bitcoinネットワークとプロ

トコルの仕組みを見ていきます。

## Bitcoin以前のデジタル通貨

持続可能なデジタルマネーの出現には暗号技術の発展が欠かせません。これはデジタル情報を、モノやサービスと交換可能な価値を持つものとして扱う際の課題を考えると理解できます。デジタルマネーを使おうとする人は次の2つの基本的な疑問を持つでしょう。

1. このデジタルマネーは本物か？偽物ではないのか？
2. 自身が所有するデジタルマネーは自身だけのものか？他者が同時にそのデジタルマネーを所有していることはないか？(二重支払問題として知られる)

紙幣の発行者は、紙製造技術と印刷技術をより高度にすることで偽造問題に対処しています。また物理的なお金は、同じものが2か所に存在することもないため二重支払問題とは無縁です。もちろん伝統的なお金の貯蓄や送金などが電子的に処理されることもあります。この場合、世界のすべての電子取引を管理する中央機関が決済することにより偽造問題や二重支払問題に対処しています。特殊なインクやホログラムのストライプを利用することができないデジタルマネーにとって、暗号技術はユーザーに価値正当性を担保する基本となります。特に、ユーザーはデジタル資産や電子取引データに、暗号によるデジタル署名を施すことでその所有権を保障することができます。また適切なシステム設計を行えばデジタル署名を用いることで二重支払問題も対処可能です。

1980年代後半に暗号技術の理解が進みより広く利用されるようになった際、多くの研究者が暗号技術を利用したデジタル通貨の開発を試みました。これらの初期のデジタル通貨プロジェクトは国の通貨やゴールドのような貴金属を裏付けにデジタルマネーが発行されるというものでした。

これらの初期のデジタル通貨は機能したものの中央管理されたものであり、その結果、政府やハッカー達に容易に攻撃されるものでした。これらの通貨は伝統的な銀行システムと同様に、中央手形交換所を導入し定期的な取引を決済する仕組みをとっていました。残念ながらこのような黎明期のデジタル通貨は、政府による訴訟の末ほとんどの場合で廃止に至りました。また親会社の突然の破たんにより劇的に消滅したもののケースもあります。正当な政府であれ犯罪分子であれ、何らかの敵対者の介入に対して、より強固な通貨であるためには、単一攻撃点の無い分散化された仕組みが必要となります。Bitcoinはまさにそのようなシステムであり、完全に分散化され、単一攻撃点として狙われたり破綻する可能性のあるような中央機関も持たずに動作するよう、デザインされています。

Bitcoinは何十年にも渡る暗号技術や分散システムの研究の集大成であると共に、次の4つのカギとなるイノベーションを独創的かつパワフル組み合わせにより成り立っています。

- 分散化されたpeer-to-peerネットワーク(Bitcoinプロトコル)
- 公開取引元帳(ブロックチェーン)
- 数学的かつ決定論的な通貨発行(分散マイニング)
- 分散取引検証システム(トランザクションscript)

## Bitcoinの歴史

Bitcoinは "Bitcoin: A Peer-to-Peer Electronic Cash System. (Nakamoto)" (Bitcoin: Peer-to-Peer電子マネーシステム)というSatoshi Nakamoto名義による論文の発表と共に、2008年に発明されました。Nakamotoは

b-

moneyやハッシュキャッシュといった先行の発明を組み合わせることで、完全に分散され、いかなる中央機関を持たずに取引の合意と検証を行える電子通貨システムを作り出しました。最も重要なイノベーションは、分散ネットワークの中で取引状態の合意を形成するために10分毎にグローバルな"選出"を実行していく"proof of work"と呼ばれる分散計算システムを取り入れたことです。これにより、これまで中央手形交換所を通じて決済する方法でしか解決できなかった、一つのお金を2か所で使用するというデジタル通貨の弱点である二重支払問題が巧みに解決されました。

BitcoinネットワークはNakamotoがリリースしたリファレンス実装をベースに2009年にスタートし、その以降、多くのプログラマーにより改良され続けています。Bitcoinにセキュリティと堅牢性を与える源となる分散コンピューターの規模は増大し続け、現在では世界トップのスーパーコンピューターの処理能力を超えます。bitcoinの市場価格はその交換レートにもよりますが50億~100億米ドルの見積もられ、Bitcoinネットワークで行われたこれまでに最大の取引は1億5000万米ドルであり、これが一瞬に手数料もかかわらず送金されました。

Satoshi Nakamotoは2011年の4月頃から、開発の責任を活発なボランティアグループの一つに引き渡し、公共の場から身を引きました。Nakamotoが個人なのかグループなのかも含めて現在までその正体は不明です。ただし、Bitcoinのシステムは、Nakamotoや特定のグループが運営しているわけではなく完全に透明な数学的な原則に則って動作しています。この発明自体が極めて革新的なものであり、既に分散コンピューティング学、経済学、計量経済学などの分野に派生して新たな研究が始まっています。

## 分散コンピューティングにおける難問への解

Satoshi Nakamotoの発明は、"ビザンチン將軍問題"と呼ばれる分散コンピューティングの分野での難問に対して、初めて現実的な解を示したものです。この問題は、潜在的に信用できない危ういネットワークの中で、ネットワーク全体の行動指針について各ノードの情報交換によりどのように合意をとるのか?というものです。proof of workというコンセプトを導入することで、いかなる中央機関も必要とせず合意形成を可能とするNakamotoの発明は、分散コンピューティングの世界で革命的なものであり通貨という枠にとどまらない応用が予想されます。例えば選挙、宝くじ、資産登記、デジタル公証などの正当性の証明などに応用されていくでしょう。

## Bitcoinを誰がどのように使うのか

Bitcoin自体はテクノロジーでもあり、人々の間で価値を交換するための言葉、つまりお金でもあります。ここで、Bitcoinを利用する人の例をいくつか挙げ、彼らがどのような用途で使うのかを見ていきましょう。今後本書の中でここで示した例を用いて、実世界の中でのBitcoinの利用用途とそれを実現するテクノロジーがどのように動作しているのかを見ていきます。

### 北アメリカでの少額物品の販売

北カリフォルニアのベイエリアに住んでいるAliceは、エンジニアの友達からbitcoinについて聞きそれを使ってみたいと思っています。彼女がbitcoinについて学び、それを入手し、パロアルトにあるBobのカフェでそれを使ってコーヒーを買う例を見ていきます。この例を用いて、Bitcoinのソフトウェアや両替について、また消費者側の視点でどのようにbitcoinがやり取りされるのかを紹介します。

## 北アメリカでの高額物品の販売

Carolはサンフランシスコで画廊を経営しており、高額な絵画をbitcoinで販売しています。この例を用いて、"51%"合意攻撃というものが高額商品を扱う業者にとってどのようなリスクとなるのかを紹介します。

## 海外請負サービス

パロアルトでカフェを経営しているBobは新しいウェブサイトを作ろうと考えています。彼はインドのバンガロールに住むウェブサイト開発者のGopeshと契約し、bitcoinで開発料を支払うことにしました。この例を用いてbitcoinを用いたアウトソーシングや請負サービスと国際電信送金を紹介します。

## チャリティー募金

Eugeniaはフィリピンで児童へのチャリティー活動を行っています。彼女は最近bitcoinについて知り、これを用いて新たな国内外のグループに寄付金を募集したいと考えています。またbitcoinを用いて寄付金を必要な地域に素早く分配したいと考えています。この例で、bitcoinを用いた国境や通貨を超えた世界的な資金調達や、オープンな元帳を用いたチャリティー活動の透明性の向上の方法を紹介します。

## 輸出入

Mohammedはドバイで電化製品の輸入業を営んでおり、アメリカや中国からUAEへ電化製品を輸入する際の支払処理をより速やかに進めるためにbitcoinを利用しています。この例を用いて、大規模なBtoBビジネスでの物品の国際売買で、どのようにbitcoinが利用できるのかを紹介します。

## bitcoinの採掘

Jingは上海に住むコンピューターエンジニアを目指す学生で、副収入を得るためにBitcoin"マイニング用"のシステムを構築しています。この例でBitcoinの"産業"基盤であり、Bitcoinネットワークを堅牢にし通貨の発行を担う、専用のシステムを見ていきます。

ここに示したそれぞれの例は、新たな市場や産業、またはグローバル経済問題に対する革新的なソリューションを創造するためにBitcoinを利用している、実際の人々や産業をベースにしました。

# Bitcoinをはじめよう

専用のアプリケーションをダウンロードするかウェブアプリケーションを利用することで、Bitcoinネットワークに参加しbitcoinが利用可能になります。Bitcoin自体は規格であり、その規格に準拠するクライアントソフトウェアは数多く存在します。その中には、Satoshi Nakamotoのオリジナルの実装から派生しオープンソースプロジェクトとして開発チームが管理している、Satoshiクライアントと呼ばれるリファレンス実装もあります。

Bitcoinクライアントには主に次の3つのタイプがあります。

## フルクライアント

フルクライアントまたは"フルノード"と呼ばれるタイプのクライアントはBitcoinが開始されて以来の全てのユーザーの全ての取引情報の保持します。またユーザーのウォレットの管理も行い、フルノードだけでBitcoinネットワーク上のトランザクションを直接開始できます。これは、他のサーバや第三者のサービスに依存することなくEメールの送受信全般を管理することが可能なスタンドアロンのメールサーバに似ているといえるでしょう。

## 軽量クライアント

軽量クライアントと呼ばれるクライアントはユーザーのウォレットは保持しますが、Bitcoinの取引情報やBitcoinネットワークへのアクセスは第三者が管理するサーバを介して行います。フルクライアントのように全トランザクション情報を保持しないため、取引の正当性の認証は第三者サーバを信用したうえでそれに依存することになります。これは、メールサーバにあるメールボックスを介してEメールを送受信するメールクライアントに似ています。

## ウェブクライアント

ウェブクライアントと呼ばれるクライアントはウェブブラウザから第三者が保持しているウォレットにアクセスする形態のものです。これは全メールのデータを第三者のサーバ内で管理しているウェブメールに似ているといえるでしょう。

## モバイルでのBitcoin

アンドロイド端末のようなスマートフォン上で動作するクライアントがあり、やはりフルクライアント、軽量クライアント、ウェブクライアントのどれかの形体をとります。いくつかのクライアントでは、ウェブまたはデスクトップのクライアントと同期し、同じウォレットにある資金を複数のデバイスに跨って使用することが可能になっています。

どのタイプのクライアントを選ぶかは、ユーザーが自分の資金の管理やコントロールをどの程度自分で行いたいかによります。フルクライアントを利用することでユーザーは最大限コントロールと独立性を確保できますが。一方でバックアップやセキュリティ確保を自身で行う必要があります。対極にあるのがウェブクライアントです。セットアップも利用も最も簡単ですが、セキュリティやコントロールはサービス提供者の提供するものとなるため相手方リスクが避けられません。これまで幾度となくあったように、万が一ウェブ・ウォレットサービスに不正アクセスが発生した場合には、そのサービスを利用するユーザーは全資金を失う可能性もあります。ただ、逆に言えばフルクライアントを利用しているユーザーも、自分自身で適切な運用が行われていなければ、自身の全資金を失うリスクがあります。

本書の中で、リファレンス実装(Satoshiクライアント)からウェブウォレットまで、幾つかのタイプの実際のクライアントについてその使用例をデモンストレーションしていきます。Bitcoinのプログラム用インターフェースを見ていく中で、リファレンス実装のクライアントが必要になる例も出てきます。これはリファレンス実装が、フルクライアントであることに加え、ウォレットやネットワーク、取引サービスに対してAPIを公開しているからです。

## クイックスタート

[Bitcoinを誰がどのように使うのか](#)の節の中で紹介したAliceについて見ていきましょう。彼女は技術者でなく、最近友達からbitcoinについて聞きbitcoinを知りました。彼女はBitcoinの公式サイトである [bitcoin.org](https://bitcoin.org) に訪問し、いくつかのBitcoinのクライアントの種類があることを知ります。そこで彼女はサイトのアドバイスに従い、軽量クライアントである Multibitを選びました。

Aliceは [bitcoin.org](https://bitcoin.org) サイトからリンクをたどりMultibitをダウンロードし彼女のデスクトップPCにインストールします。MultibitはWindows、MAC OS、LinuxのデスクトップPCで利用可能なクライアントです。

## WARNING

Bitcoinウォレットは必ずパスワードまたはパスフレーズで保護されていなければなりません。多数の悪意ある人間が脆弱なパスワードの解読を試みており、簡単には破れないパスワードを利用すべきです。大文字・小文字・数字・記号の組み合わせでパスワードを生成し、誕生日や名前、スポーツチームのようなものは避け、言語に関わらず辞書に載っている一般的な単語は避けてください。可能であれば、完全にランダムなパスワードを生成するパスワードジェネレーターのようなものを利用し、12文字以上のパスワードにするべきです。Bitcoinがお金であること、そして世界のどこにでも一瞬のうちに移動させることができること、適切に守られていなければ簡単に盗まれてしまうことを是非とも忘れないでください。

AliceがMultibitクライアントをダウンロード・インストールした後、アプリケーションを起動するとMultibitクライアントのウェルカム画面のようなウェルカム画面が表示されます。

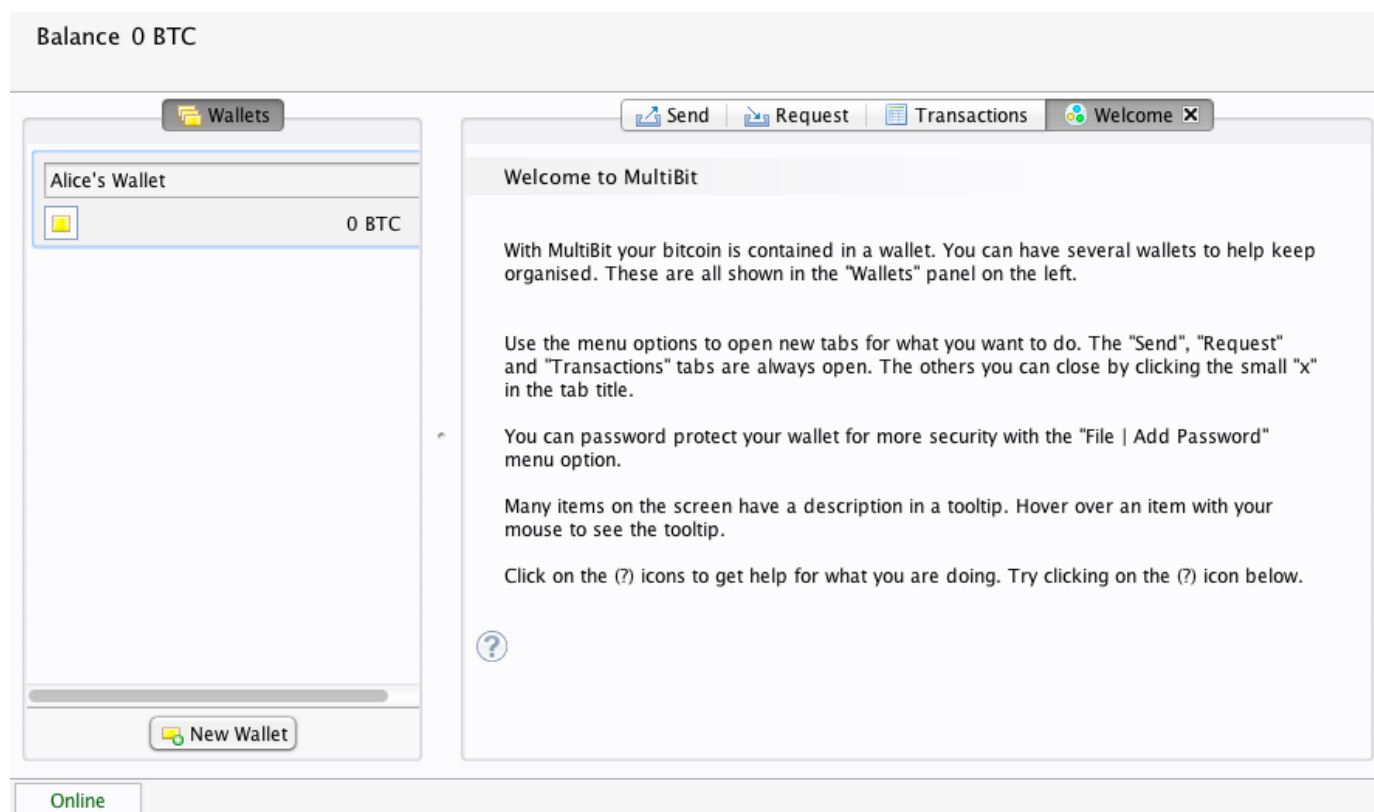


Figure 1. Multibitクライアントのウェルカム画面

Multibitは自動的にアリスのためにBitcoinアドレスとウォレットを生成し、そのアドレスは [\[multibit-request\]](#) 図に示されたようにRequestタブに表示されます。

Requestタブに表示されたAliceのBitcoinアドレス `image::images/msbt_0102.png["MultibitReceive"]`

この画面で最も重要なのはAliceの `Bitcoin` アドレス `1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK` です。Eメールのアドレスと同様、他者とこのアドレスを共有することで、その人が彼女のウォレットに直接お金を送信することが可能になります。アドレスはスクリーン上に `1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK` という数字とアルファベットで構成された文字列として表示されています。Bitcoinアドレスの横にはQRコードが表示されます。これはスマートフォンのカメラで読み取り可能なように白黒の四角形で表されるバーコードの形式にBitcoinアドレスを変換したものです。Bitcoinアドレス、またはそのQRコードをそれぞれの横にあるコピーボタンをクリックすることでクリップボードにコピーできます。QRコード自体をクリックするこ

とで、スマートフォンで容易に読み取れるように拡大表示することも可能です。

また、他の人が長い文字と数字の列を打ち込む必要が無いようにQRコードを印刷することも可能です。

#### TIP

Bitcoinアドレスは数字の1か3で始まります。Eメールアドレスと同様に、他者にこのBitcoinアドレスを教えてあなたのウォレットにbitcoinを直接送金してもらうことが可能です。一方で、Eメールアドレスとは異なり、Bitcoinアドレスは無制限に作ることができ、それらをあなたのウォレットに紐づけることができます。ウォレットは単にBitcoinアドレスとそのアドレスの持つbitcoinを使うためのキーの集合です。例えば、取引の度にアドレスを生成することも可能で、そうすればよりあなたのプライバシーがより守られるでしょう。

さあ、アリスは新しいBitcoinウォレットを使う準備ができました。

## 最初のbitcoinを手に入れる。

現在のところ、bitcoinを銀行や外貨両替所で買うことは出来ず、2014年時点ではまだかなりの国でbitcoinを手に入れることは難しい状態です。ただ、例えば以下のような各国の通貨でbitcoinが売買可能なウェブサービスは数多く存在します。

### Bitstamp

電信送金により、ユーロ（EUR）や米ドル（USD）を含む幾つかの通貨をサポートするヨーロッパのbitcoin市場です。

### Coinbase

米国に拠点がある、bitcoinで商品が売買可能なプラットフォームを持つBitcoinウォレットです。ACHシステムを通じた当座預金口座にアクセスを可能にすることでbitcoinの売買が容易になっています。

これらの暗号通貨両替所は国の通貨と暗号通貨の両替を行います。この場合、国または国際的な規制に従う必要があるため、これらのサービスは、しばしば特定の国または地域のみを専門に扱います。通貨取引所はあなたの国の司法権が及ぶ範囲の通貨に限定されます。銀行で口座を開設するのと同様、これらのサービスに口座を開設するのは、KYC (know your customer)やAML (anti-money laundering) というような規制に従い、様々な本人確認を行う必要があるため、数日から数週間ほどの時間がかかります。ただし、Bitcoin取引所口座の開設が完了すると、外貨取引口座での外貨取引同様、bitcoinを素早く売買することが可能になります。

より詳細な取引所のリストは [bitcoin charts](#) を参照するとよいでしょう。ここでは様々なbitcoin通貨取引所での相場情報・マーケット情報が掲載されています。

その他にもbitcoinを入手する方法が4つあります。

- bitcoinを持つ友人から直接bitcoinを買う。多くのユーザーはこの方法から始めます。
- localbitcoins.com のようにあなたの住む地域でbitcoinを売ってくれる人を探すサービスを利用する。
- 商品やサービスをbitcoinで売る。例えばあなたがプログラマーであれば、あなたのプログラミングスキルを売ることもできます。
- Bitcoin [CoinDesk](#) ATMを利用する。あなたの住む町のATMは [CoinDesk](#) のような地図サービスを利用することで見つけることもできます。



Aliceはカリフォルニアの通貨市場に自分の口座を開設されるのを待つまでの間、bitcoinを紹介してくれた友人から最初のbitcoinを手に入れることにします。

## bitcoinを送る / 受け取る

AliceはBitcoinウォレットを既に作成しており、bitcoinを受け取る準備は整っています。ウォレットアプリケーションは秘密鍵(詳細は[\[private\\_keys\]](#)節を参照)をランダムに生成し、同時にそれに対応したBitcoinアドレスも生成しました。この時点では彼女のBitcoinアドレスはBitcoinネットワークに伝えられていないし何らかの登録が行われたわけでもありません。アドレスとそのアドレスに紐づく資金をコントロールするためのキーが単に存在するだけです。受け手としてこのアドレスが指定されたBitcoinトランザクションが公開元帳(ブロックチェーン)に書き込まれるまでは、Bitcoinネットワークにとってこのアドレスは単なるビットコイン上で有効なアドレスの候補の一部であるだけです。トランザクションが公開元帳に書き込まれるとネットワークはそのアドレスを認知し、Aliceもそのアドレスの資金の残高を元帳から参照できるようになります。

Aliceは米ドルとbitcoinを交換するために、bitcoinを紹介してくれたJoeにレストランで会うことにしました。彼女はアドレスとそのQRコードをプリントアウトして持参しています。Bitcoinアドレスを他人に見せることはセキュリティ面で懸念することはありません。Bitcoinアドレスをどこに書き込んででも特にリスクはありません。

Aliceは、この新しいテクノロジーに多くのリスクをかけることを避けるため10米ドルだけbitcoinに替えたいと思っています。彼女は10米ドル相当のbitcoinを送ってもらうよう、Joeに10米ドルと彼女のアドレスを渡しました。

次にJoeは適切な額のbitcoinをAliceに送るために交換レートを確認します。アプリケーションやWebサイトなどで市場の交換レート情報を提供するサービスは数多く存在し、例えば下記のものが有名です。

### Bitcoin Charts

bitcoin市場の情報サイト。世界各国の取引所における各国通貨との交換レート情報を各国の通貨建てで提供。

### Bitcoin Average

各国通貨に対してのbitcoinの売買高加重平均価格をシンプルに表示してくれるサイト。

### ZeroBlock

幾つかの取引所でのbitcoin価格を表示してくれるフリーのAndorid/iOSアプリ(詳細は [ZeroBlock](#) : bitcoinの市場情報を提供してくれるAndoroid/iOSアプリ 図を参照)。

### Bitcoin Wisdom

もう一つの市場情報サイト。

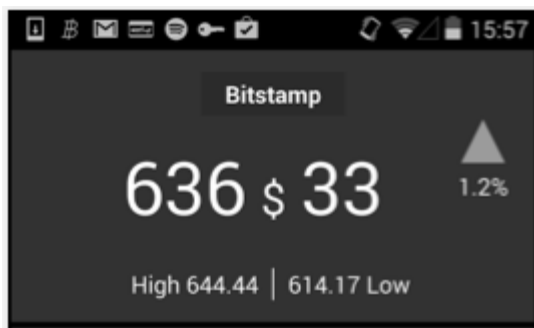


Figure 2. ZeroBlock : bitcoinの市場情報を提供してくれるAndoroid/iOSアプリ

上記に挙げたアプリやWebサイトをチェックしてJoeは交換レートを、1 bitcoinあたり100米ドルと決めます。このレートの場合、Aliceから渡された10米ドルは0.10 bitcoin(100 millibitともいう)に相当します。

適切な交換価格を決定した後、Joeはモバイルのウォレットアプリケーションを起動しbitcoinを"send"メニューを選択します。例えば彼がアンドロイドのBlockchain mobile walletアプリを使用していたとしたら、彼はBlockchain mobile walletの送信画面に示されたような2つの入力を求める画面が表示されます。

- 今回のトランザクションでの送信先Bitcoinアドレス
- 送信するbitcoinの量

Bitcoinアドレスを入力する欄には、QRコードのアイコンがあります。長くて入力しづらいAliceのBitcoinアドレス(1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK)を直接入力せず、カメラでQRコードを読み取ることでアドレスが入力可能です。JoeはQRコードのアイコンをタップし、スマートフォンのカメラを起動してAliceが印刷したQRコードを読み取ります。Joeは読み込まれて自動で入力されたアドレスの文字列の幾つかの部分をAliceが印刷したアドレスと比較して、正しくアドレスが読み込まれていることを確認します。

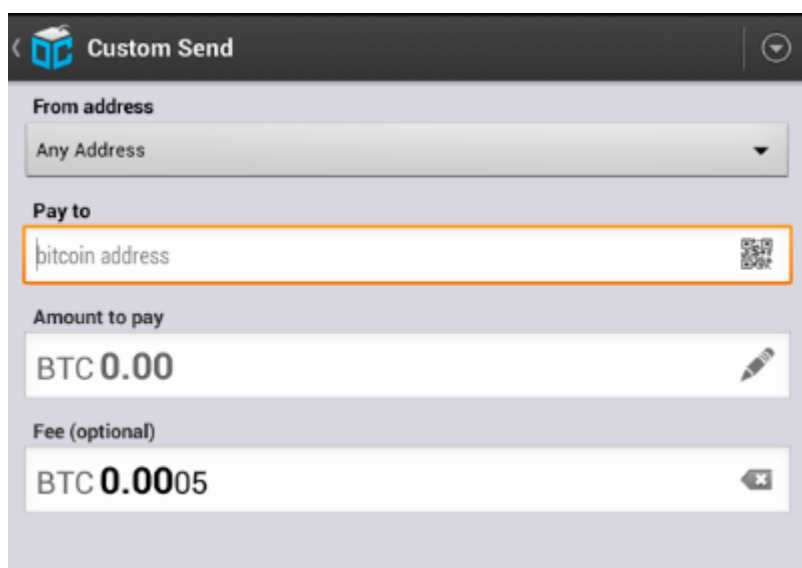


Figure 3. Blockchain mobile walletの送信画面

Joeは今回のトランザクションで送信するbitcoinの量、つまり0.10bitcoinを入力します。bitcoinはお金のため、送信先や量を間違える訳にはいきません。そのため、Joeは入力が正しいことを入念に確認した後、送信

ボタンを押してトランザクションを送信します。Joeのアプリケーションは、Joeのアドレスが持つ資金から0.10bitcoinをAliceのアドレスに渡すというトランザクションにJoeの秘密鍵で署名をし、トランザクションを送信します。署名によりネットワークはこのトランザクションはJoeの承認がある正当なものとして認知することが可能になります。peer-to-peerプロトコルにより、このトランザクションの情報がBitcoinネットワーク上に素早く伝搬されます。1秒以内に、ネットワーク上で密につながったノードのほとんどが、このトランザクション情報を受信し、またAliceのアドレスを初めて認知します。

Alice側でもスマートフォンやノートパソコンから、このトランザクションを確認することができます。Bitcoinが始まって以来発生した全てのトランザクションの情報を記録し続けている公開元帳は文字通り公開されており、Aliceもこの元帳で自分のアドレスを検索し、bitcoinを受け取ったかを調べることができます。これは [blockchain.info](http://blockchain.info)

というサイトで検索ボックスに自分のアドレスを入力することで簡単に行うことができます。 <http://bit.ly/1u0FFKL>[[blockchain.info](http://blockchain.info)のページ] では、Aliceのアドレスから送信された、またはそのアドレスに送られた全てのトランザクションが表示されます。Aliceがこのページを見ていると、Joeがbitcoinを送信した後すぐに0.10bitcoinが送信されたトランザクションが新しく表示されます。

## トランザクションの承認

表示されたJoeからのトランザクションは最初、"未承認"と記載されます。これはトランザクションの情報がネットワーク上に伝搬したがブロックチェーンと呼ばれる公開元帳にまだ記載されていないことを示しています。マイナーがこのトランザクションを拾い上げブロックチェーンに記載します。トランザクションは、おおよそ10分毎に生成されるブロックに記載されて初めてネットワーク上で"承認済み"として受け入れられ、Aliceは受け取ったbitcoinを使用できるようになります。つまりトランザクション自体は一瞬で閲覧可能になりますが、新しいブロックに記載されて初めて"信用ある"取引として認められるということです。

これで、Aliceは晴れて0.10bitcoinの所有者となり、このbitcoinが使用できるようになりました。次の章ではAliceがbitcoinを使って初めて商品を購入するところを見ていき、その中でトランザクションとその伝搬を担う技術を詳しく解説します。

# Bitcoinの仕組み

## トランザクション、ブロック、マイニング、ブロックチェーン

今までであるような銀行サービスや支払いの方法と違って、Bitcoinは特定の機関に管理されないde-centralized trustを基礎にしています。特定の機関による管理の代わりに、Bitcoinでは、Bitcoinのシステムに参加する参加者の相互協力から生まれ、個に還元できない全体としての性質によって信用管理をしています。この章では、Bitcoinの1つのトランザクションを追うことで詳細にBitcoinの仕組みを調べ、また、トランザクションがBitcoinの分散化合意形成の仕組みによって"信用"され受け入れられ、最後に全てのトランザクションの分散元帳であるブロックチェーンに記録される過程を見ます。

以下の例を使って、実際に行われているトランザクションをシミュレートしてみましょう。この例の登場人物はJoe、Alice、Bobで、それぞれの間であるウォレットからあるウォレットへ資産を送るというものです。Bitcoinネットワークのトランザクションやブロックチェーンを追うとき、個々の詳細なステップを可視化するために [blockchain explorer](#) ウェブサイトを使います。blockchain explorerはウェブアプリケーションで、Bitcoinアドレスやトランザクション、ブロックの変化を追うことができる検索エンジンのように使えます。

ポピュラーなblockchain explorerは以下です。

- [Blockchain info](#)
- [Bitcoin Block Explorer](#)
- [insight](#)
- [blockr Block Reader](#)

それぞれのblockchain explorerではBitcoinアドレスやトランザクションハッシュ、ブロック番号を元に検索でき、Bitcoinネットワークやブロックチェーン上にあるデータと同じデータを見つけることができます。また、関連するウェブサイトを直接参照できるようにそれぞれの例にはURLを載せますので、詳細を確認することができます。

## Bitcoin概観

[Bitcoin概観](#)図にあるように、Bitcoinの仕組みは秘密鍵を含むウォレットを持っているユーザやBitcoinネットワークを伝わるトランザクション、全てのトランザクションを保持している元帳であるブロックチェーンを作り出すマイナーで構成されています。この章では、Bitcoinネットワークに沿って1つのトランザクションを追い、各ステップを説明します。続く章では、ウォレット、マイニング、決済システムの背後にあるテクノロジーを掘り下げます。

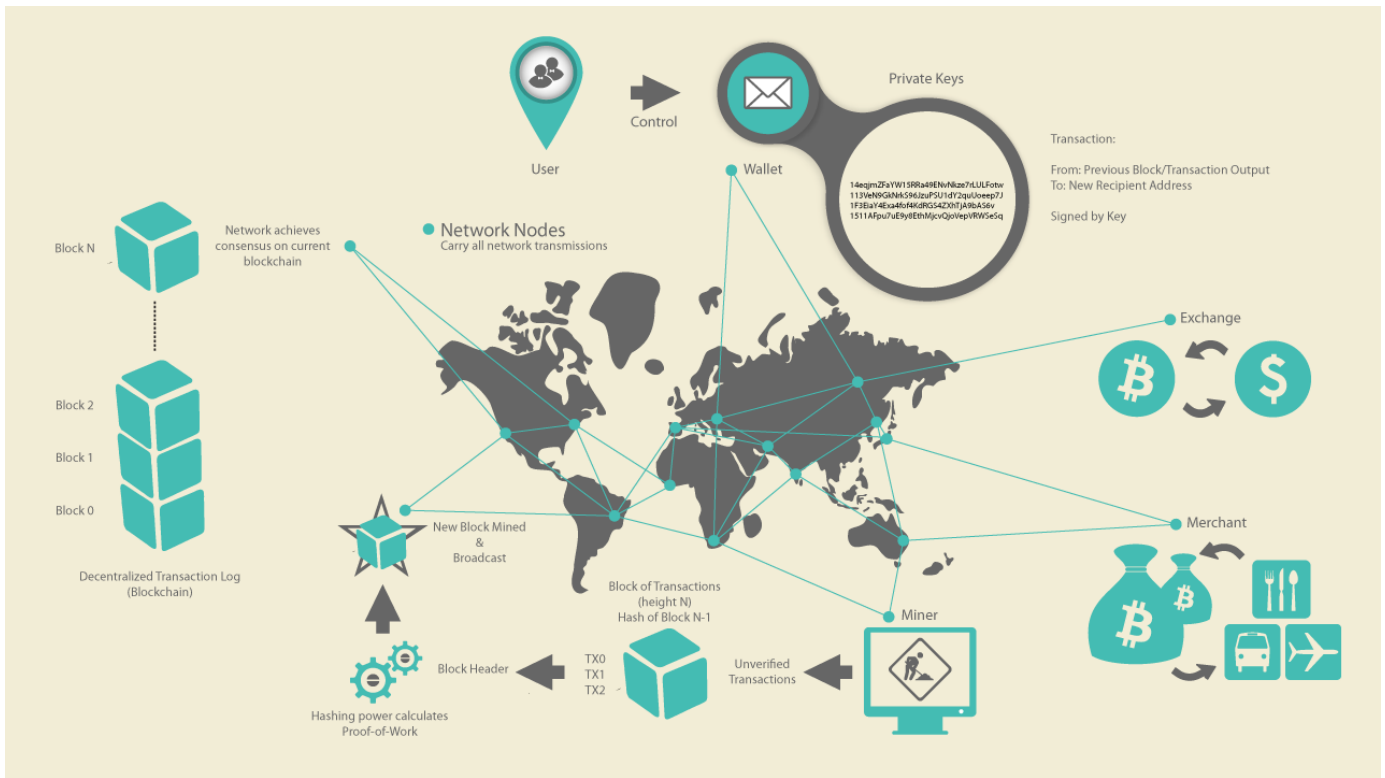


Figure 1. Bitcoin概観

## コーヒー代金の支払い

前の章に出てきたAliceは初めてbitcoinを手にしたところです。[getting\_first\_bitcoin]でお分かりの通り、Aliceは友達Joeと会って、現金をbitcoinと交換したのです。JoeからAliceは0.10BTCを受け取りました。今ちょうどAliceは最近bitcoinでの支払いを始めたPalo AltoのBobのコーヒーショップでbitcoinでの支払いをるところです。Bobのコーヒーショップでは現地通貨(米ドル)の値段表示しかありませんが、支払いをするときに米ドルで払うのかbitcoinで払うのかを決められるのです。Aliceはコーヒーを注文しBobはレジにこの注文を入力しました。すると、POSシステムは直近のレートで米ドルでの金額をbitcoinでの金額に変換して、両方の金額を表示してくれます。このとき、図にあるようなQRコードも一緒に表示してくれます。(支払いリクエストQRコード(スキャンしてみてください!))[図参照](#))

総額:  
\$1.50 USD  
0.015 BTC



Figure 2. 支払いリクエストQRコード(スキャンしてみてください!)

この支払いリクエストQRコードはBIP0021にあるプロトコルに沿って次のようなURLに変換されます。

```
bitcoin:1GdK9UzpbHzqzX2A9JFP3Di4weBwqgmoQA?  
amount=0.015&  
label=Bob%27s%20Cafe&  
message=Purchase%20at%20Bob%27s%20Cafe
```

#### URLの構成要素

Bitcoinアドレス: "1GdK9UzpbHzqzX2A9JFP3Di4weBwqgmoQA"  
支払い総額: "0.015"  
支払い先アドレスのラベル: "Bob's Cafe"  
支払い説明文: "Purchase at Bob's Cafe"

#### TIP

送り先Bitcoinアドレスだけを含んでいるQRコードと違って、支払いリクエストは、送り先Bitcoinアドレス、支払い総額、"Bob's Cafe"のような一般的な説明文を含んでいる、QRコードでエンコードされたURLです。これによって、Bitcoinウォレットが、人間が読める形式の説明文をユーザに表示しながら、支払いを行うのに用いる情報を、空欄にあらかじめ記入しておくことができます。このQRコードをBitcoinウォレットでスキャンすると、Aliceが見ているものを見ることができます。

Bobは言いました。"15ドルです。bitcoinでの支払いであれば15mBTCですよ。"

Aliceがスマートフォンを使って表示されているQRコードをスキャンすると、スマートフォンに 0.0150BTC と表示され Bob's Cafe への支払いをするのに彼女は send ボタンを押しました。数秒後レジにトランザクションが表示されBobはトランザクションの完了を確認しました(処理時間はクレジットカードでの承認に必要な時間と同じくらいです)。

この後の節では、もっと詳細にトランザクションの内容を説明し、Aliceのウォレットがどのようにしてトランザクションを実行したのか、トランザクション情報はどのようにしてBitcoinネットワークに流れ、どのように検証されたのか、送られたbitcoinをBobは次回どのように使うことができるのか、を見ていきます。

**NOTE** Bitcoinネットワークでは様々な額で取引ができます。例えば、ミリbitcoin (1/1000 bitcoin)から satoshiとして知られている1/100,000,000 bitcoinまでです。この本を通して、最も小さい単位(1satoshi)から今後採掘される全てのbitcoinの総額(21,000,000bitcoin)まで、bitcoin通貨の量を表現するために“bitcoin”という用語を使っていきます。

## Bitcoinトランザクション

シンプルに言って、トランザクションとはbitcoinの所有者が他の人にbitcoinを送ったと認めたことをBitcoinネットワークに示すことです。このため、新しい所有者が受け取ったbitcoinを使うには、新しい所有者が他の人にbitcoinを送ったと認めたことを示す別のトランザクションを作らなければいけません。

トランザクションは複式簿記の個々の取引行のようなものです。それぞれのトランザクションは1個または複数の"インプット"を持っているため、トランザクションの借り方にこの"インプット"が記載されています。また、それぞれのトランザクションは1個または複数の"アウトプット"を持っているため、トランザクションの貸し方にこの"アウトプット"が記載されています。インプットとアウトプット(それぞれ借り方と貸し方)は同じ額になるようにはならず、わずかにインプットのほうが大きくなります。この差がトランザクション手数料であり、元帳の中にあるトランザクションからマイナーがかき集めることになるものです。[複式簿記としてのトランザクション](#)図には、bitcoinトランザクションを簿記的に書いてあります。

トランザクションにはそれぞれのインプットごとにbitcoinの所有権証明も含まれています。この所有権証明はデジタル署名の形になっており、このデジタル署名は所有者とは独立に誰か他の人によって検証されるようになっています。Bitcoinの用語で、"bitcoinを使う"とはトランザクションに署名することです。

**TIP** トランザクション は トランザクションインプット から トランザクションアウトプット に価値を移動します。インプットはどこからbitcoinが来たかを示し、通常は前のトランザクションのアウトプットになります。トランザクションアウトプットは、このbitcoinを鍵と紐付けることで新しい所有者にこのbitcoinを割り当てます。この鍵は 解除条件 と呼ばれるものです。解除条件は、資金を将来トランザクションで使用するときに必要なとされる署名に対する必要条件になります。1つのトランザクションからのアウトプットは新しいトランザクションの中でインプットとして使用され、これにより、アドレスからアドレスに価値が移転するときに、所有の連鎖が作られるのです([トランザクションの連鎖](#)。[あるトランザクションのアウトプットは次のトランザクションのインプットになる](#)。図参照)。

Transaction as Double-Entry Bookkeeping			
Inputs	Value	Outputs	Value
Input 1	0.10 BTC	Output 1	0.10 BTC
Input 2	0.20 BTC	Output 2	0.20 BTC
Input 3	0.10 BTC	Output 3	0.20 BTC
Input 4	0.15 BTC		
...			
Total Inputs:	0.55 BTC	Total Outputs:	0.50 BTC
...			
	<i>Inputs</i>		<i>0.55 BTC</i>
-	<u><i>Outputs</i></u>		<u><i>0.50 BTC</i></u>
	<i>Difference</i>		<i>0.05 BTC (implied transaction fee)</i>

Figure 3. 複式簿記としてのトランザクション



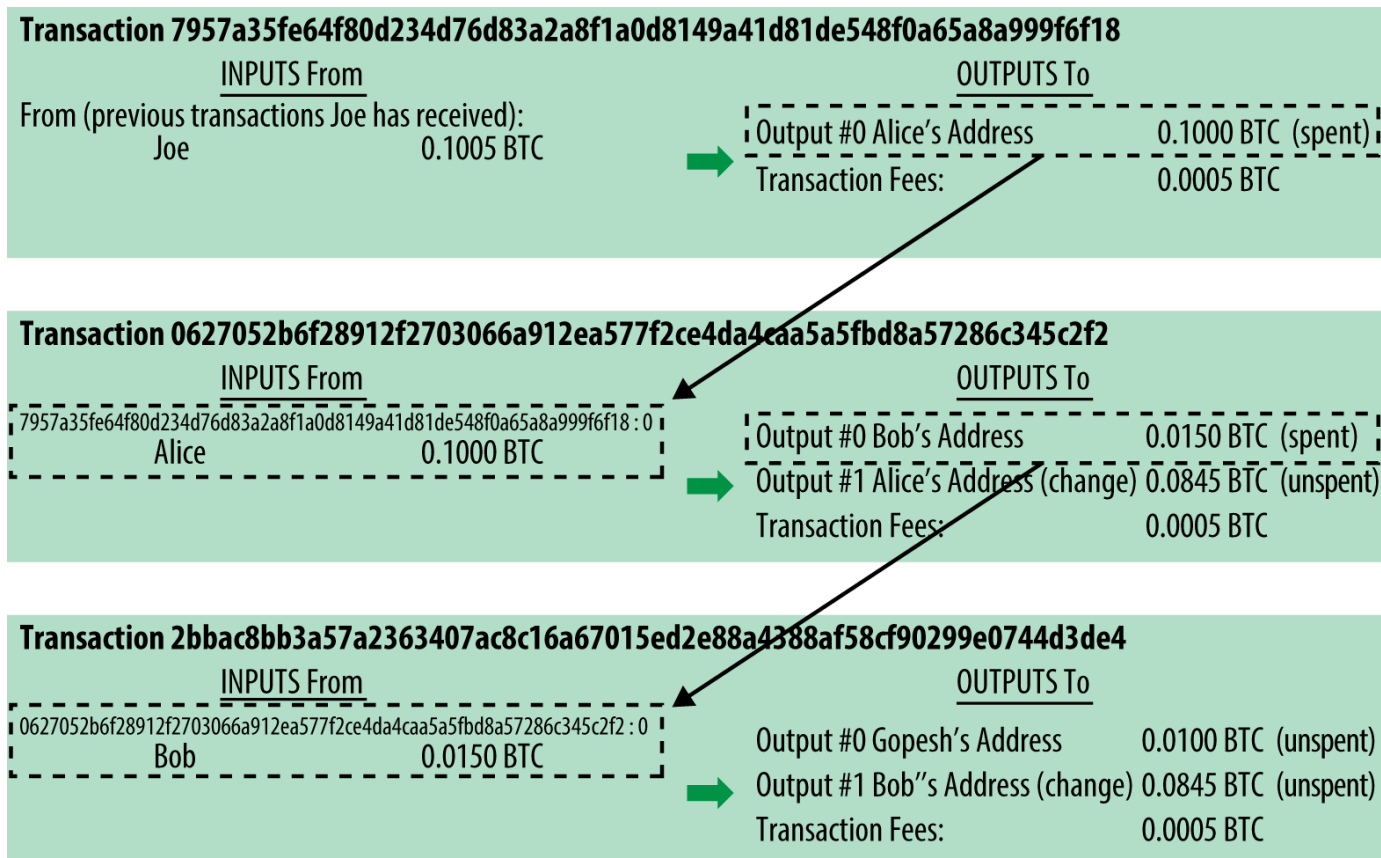


Figure 4.

トランザクションの連鎖。あるトランザクションのアウトプットは次のトランザクションのインプットになる。

AliceがBobのコーヒーショップで支払いをするときは、JoeからAliceへのトランザクションをこのトランザクションのインプットとして用います。前の章で、Aliceは現金と引き換えにJoeからbitcoinを受け取りました。このトランザクションはAliceの秘密鍵でロックされています。AliceからBobへの新しいトランザクションは、JoeからAliceへのトランザクションの内容をインプットとして参照し、コーヒー代の支払いとおつりの受け取りのトランザクションアウトプットを作成します。トランザクションはチェーンの形を取っていて、最新のトランザクションのインプットは前のトランザクションのアウトプットに対応しています。Aliceの秘密鍵は前のトランザクションのアウトプットを解錠し、それによってこのアウトプットにある資金がAliceのものであるとBitcoinネットワークに示すのです。Aliceは、コーヒー代の支払いをBobのBitcoinアドレスに紐づけます。このことによって、このアウトプットを使うためには、Bobは署名を生成しなければなりません。このことは、この価値の移転がAliceとBobの間のものであるということを表しています。トランザクションの連鎖。あるトランザクションのアウトプットは次のトランザクションのインプットになる。図が、Joe、Alice、Bobの一連のトランザクションの連鎖を説明しています。

## 一般的なトランザクション形式

最も一般的なトランザクションの形式は、1つのBitcoinアドレスからもう1つへの単純な支払いという形式をしており、元の持ち主に戻されるおつりが通常含まれます。このタイプのトランザクションは、一般的なトランザクション図に示されているように、1つのインプットと2つのアウトプットを持っています。

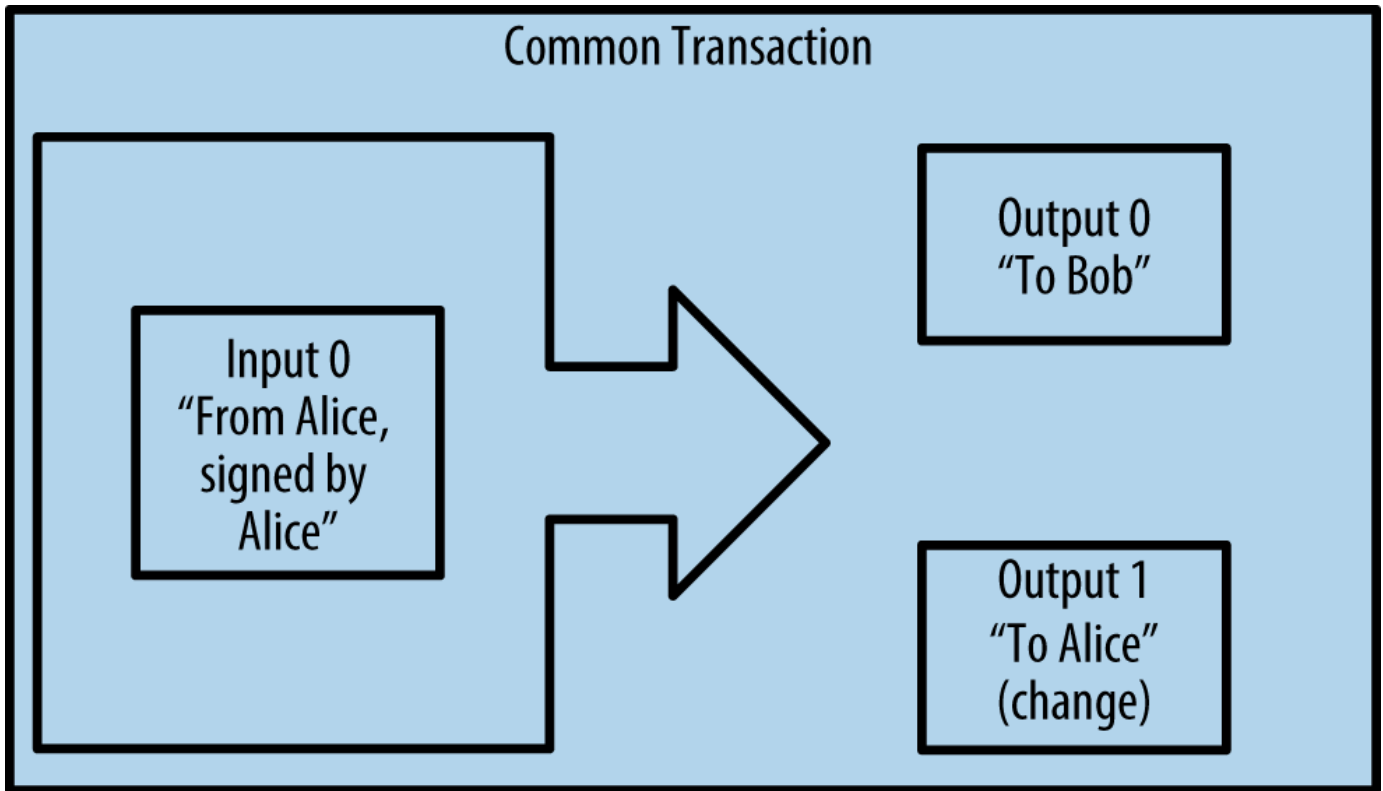


Figure 5. 一般的なトランザクション

別のトランザクション形式は、いくつかのインプットを集めて1つのアウトプットにまとめる形です(集約型トランザクション図参照)。これは現実にあるコインや少額紙幣をまとめて大きな額の紙幣にするトランザクションと同じです。これらのトランザクションはときどきウォレットで作られ、おつりとして受け取った小さな額をまとめるために使われます。

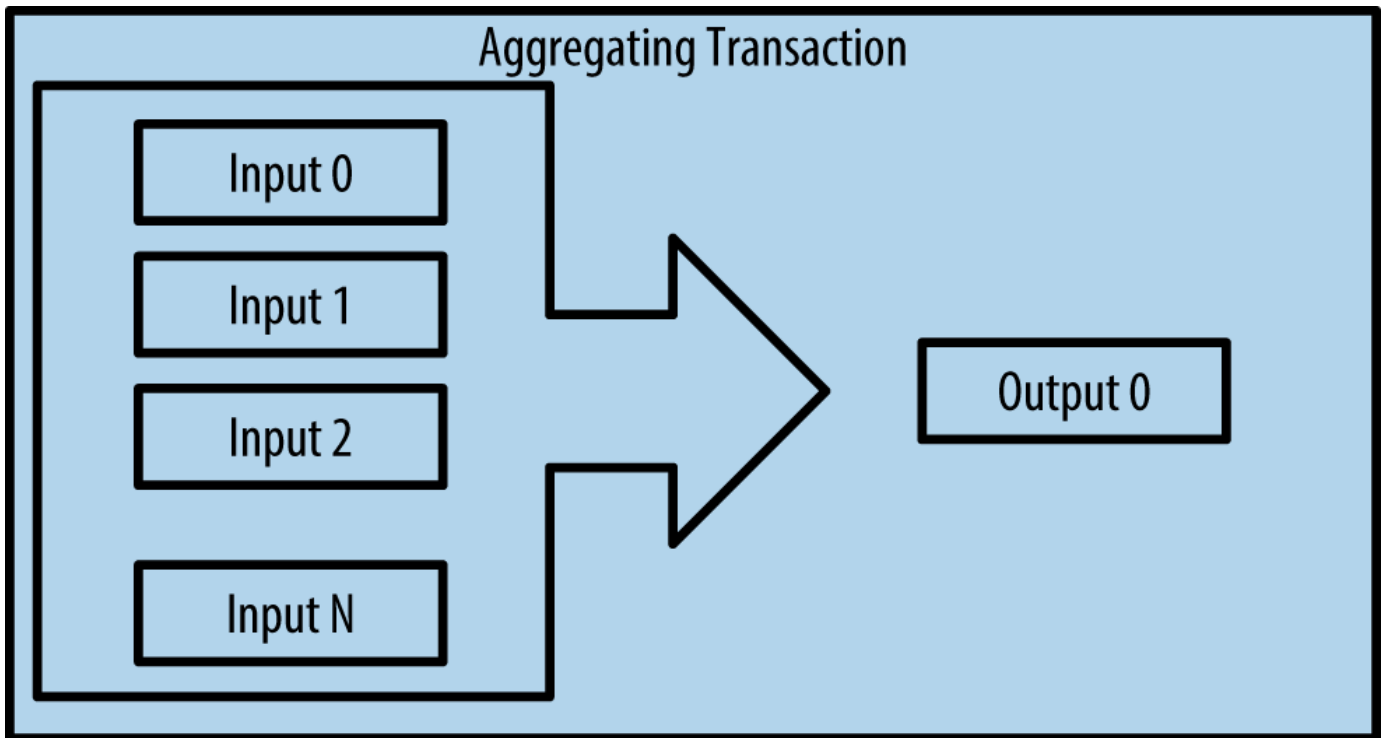


Figure 6. 集約型トランザクション

もう1つの別のトランザクションの形式は1つのインプットを複数のアウトプットに分けて複数の受取人に使う場合です(分配型トランザクション図参照)。このタイプのトランザクションは、企業内での給与の支払いでときどき使われます。

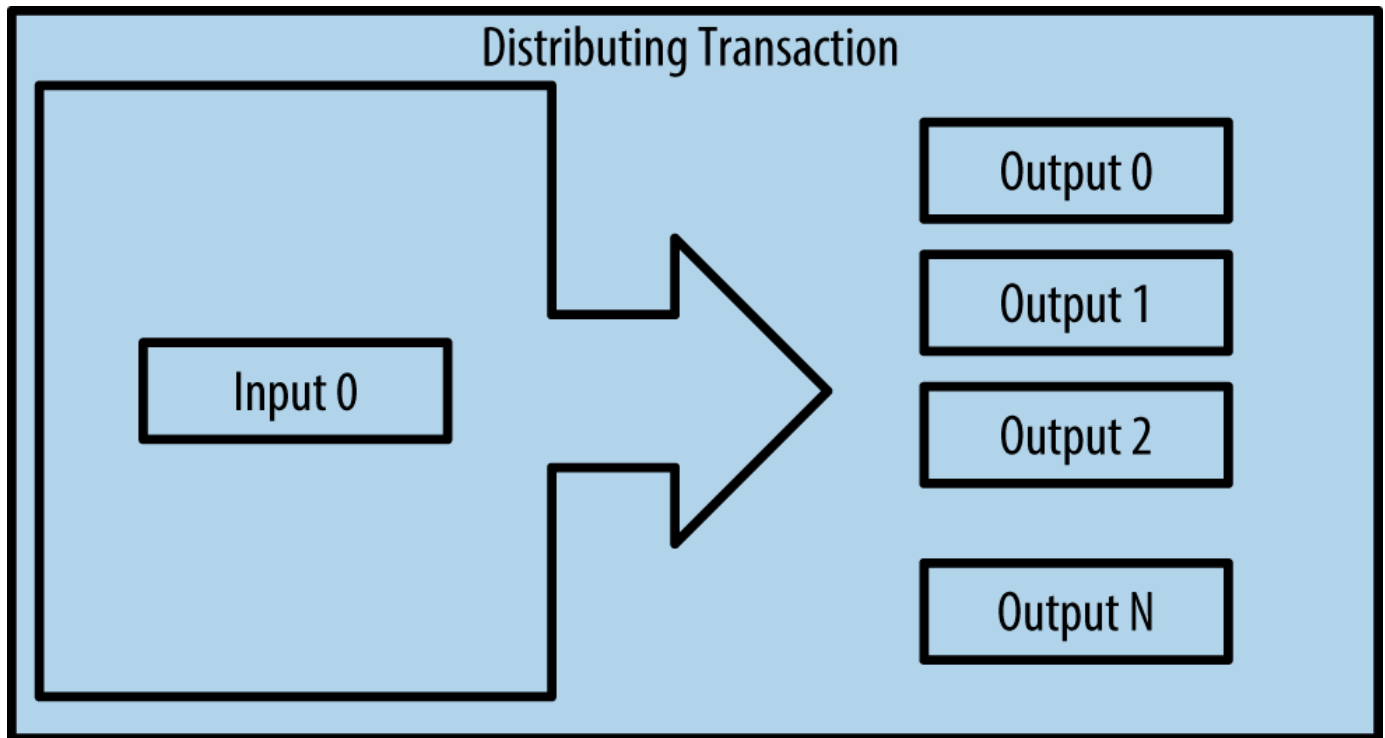


Figure 7. 分配型トランザクション

## トランザクションの構築

Aliceのウォレットで、適切なインプットとアウトプットを選ぶ処理はすでに実装されています。Aliceが決めなければいけないのは、どこに送るかということと、いくら送るかということだけで、残りはウォレットが自動的に実行してくれます。重要なこととして、ウォレットはネットワークに繋がってなくても、トランザクションを組めるということがあります。家で小切手を書いてあとで銀行に郵送できるのと同様に、トランザクションを作ったりこれに署名したりするのに、Bitcoinネットワークに繋がっている必要はないのです。取引が実行されるには、最終的に送られるだけでよいのです。

### 正しいインプットをどのように得るか

Aliceのウォレットはインプットを最初に見つけます。というのは、Bobに送ることができる金額がウォレットにあるかを確認しなければいけないためです。ほとんどのウォレットは未使用トランザクションアウトプットを保持するデータベースを持っていて、ウォレットの秘密鍵でロックされています。AliceのウォレットはJoeから送金された時のアウトプットのコピーを持っています([getting\_first\_bitcoin]図参照)。フルインデックスを持っているBitcoinウォレットは、ブロックチェーンにある全てのトランザクションアウトプットのコピーを保持しています。これはウォレットがトランザクションインプットを作成するとともにすばやく支払える金額の未使用アウトプットがあるかどうかを確認するためです。しかし、フルインデックスウォレットは多くのデータ容量を持っている必要があるため、ほとんどのウォレットは、ウォレットの所有者の未使用アウトプットのみを保持している、“軽量(lightweight)”ウォレットと呼ばれるものです。

もしウォレットが未使用アウトプットのコピーを保持していない場合、この情報を取得するためにBitcoinネットワークに問い合わせることができます。この場合いろいろな種類のAPIを通して問い合わせたり、フルインデックスを持っているノードにJSON RPC APIを通して問い合わせたりできます。Aliceの

## Bitcoinアドレスに対する全ての未使用アウトプットの参照はRESTful

APIを使って問い合わせに行ったものです。RESTful APIというのは特定のURLに対してHTTP GETコマンドを発行して情報を得るための仕組みです。このURLは、あるBitcoinアドレスが持っている未使用トランザクションアウトプットを全て返します。そして、この情報を元にウォレットはトランザクションインプットを作成します。以下では、`cURL` というRESTful APIを使うためのシンプルなコマンドを使っています。

### Example 1. AliceのBitcoinアドレスに対する全ての未使用アウトプットの参照

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
```

### Example 2. 参照URLからのレスポンス

```
{
  "unspent_outputs": [
    {
      "tx_hash": "186f9f998a5...2836dd734d2804fe65fa35779",
      "tx_index": 104810202,
      "tx_output_n": 0,
      "script": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
      "value": 10000000,
      "value_hex": "00989680",
      "confirmations": 0
    }
  ]
}
```

参照URLからのレスポンスにある通りRESTful APIから返ってきたレスポンスには1つの未使用アウトプットがあります。これは、AliceのBitcoinアドレス1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FKが所有しているものです。このレスポンスにはトランザクションの詳細が含まれていて、未使用アウトプットがsatoshiという単位で書かれています(1000万satoshiは0.10bitcoinに相当)。この情報を元に、Aliceのウォレットは他のBitcoinアドレスに送るためのトランザクションを作ることができるのです。

**TIP** | [JoeからAliceへのトランザクション](#)を見てみましょう。

今まで見てきたように、Aliceのウォレットは、コーヒー代の支払いに十分な額の、単一の未使用アウトプットを持っています。そうでなければ、Aliceのウォレットは、コーヒーの支払いができる額になるまで財布からコインを取り出すように、少額の未使用アウトプットをかき集める必要があるでしょう。どちらの場合でも

、ウォレットがトランザクションアウトプット(支払い)を作成するときには、次の節でみるように、Aliceにおつりを戻す必要があるかもしれません。

## アウトプットの作成

トランザクションアウトプットはscriptの形で作成されます。このscriptは、資金を使用する際の解除条件であり、これに対する解を導入することのみ解除されます。要するに、このscriptが意味しているのは、「Bobのパブリックアドレスに対応する秘密鍵から作られた署名を示す人であれば誰でも、このアウトプットが支払われる」ということです。Bobのみが、対応する秘密鍵を含むウォレットを持っているため、このウォレットのみがこのアウトプットを復号する署名を示すことができます。従って、Aliceがアウトプットを復号しようとしても、Bobの署名を要求され、邪魔されてしまいます。

Aliceの資金は0.10BTCのアウトプットの形をとっており、この額は0.015BTC分のコーヒーへの支払いには大きすぎるので、このトランザクションは、2つ目のアウトプットを含むこととなります。Aliceは、0.085BTCのおつりを受け取る必要があります。Aliceへのおつりの支払い処理は、Aliceのウォレットによって、Bobへの支払い処理を含むものと同じトランザクションにおいて、作られます。Aliceのウォレットは、彼女の資金を2つの支払い処理に分けます。1つは、Bobへの支払い、もう1つは彼女自身に支払うものです。そうすることで、彼女はそのおつりのアウトプットを、その後のトランザクションにおいて使うことができ、従って後の支払いにあてることができるのです。

最終的に、トランザクションがBitcoinネットワークで早く処理されるために、Aliceのウォレットは少額のトランザクション手数料を加えます。手数料は支払いの際ははっきり示されるのではなく、トランザクションにおけるインプットとアウトプットとの差額として暗に示されます。Aliceがおつりとして0.085BTCではなく0.0845BTCのアウトプットを作ると、0.0005BTC(1mBTCの半分)が使われずに残ることとなります。インプットとしての0.10BTCは、2つのアウトプットで全て使われるわけではないのです。アウトプットを全て足しても0.10BTCより小さいからです。この差額がトランザクション手数料となり、マイナーがトランザクションをブロックに含め、ブロックチェーンに組み込むための手数料として、マイナーによって徴収されます。

このようにして作られたトランザクションは、[BobのコーヒーショップへのAliceのトランザクション図](#)にある通り、blockchain explorerを使って見ることができます。

# Transaction View information about a bitcoin transaction

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK (0.1 BTC - Output)



1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA  
- (Unspent) 0.015 BTC  
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK -  
(Unspent) 0.0845 BTC

97 Confirmations

0.0995 BTC

Summary		Inputs and Outputs	
Size	258 (bytes)	Total Input	0.1 BTC
Received Time	2013-12-27 23:03:05	Total Output	0.0995 BTC
Included In Blocks	277316 (2013-12-27 23:11:54 +9 minutes)	Fees	0.0005 BTC
		Estimated BTC Transacted	0.015 BTC

Figure 8. BobのコーヒーショップへのAliceのトランザクション

## TIP

AliceからBobのコーヒーショップへのトランザクションを見てみましょう。

## トランザクションを元帳にどうやって取り込むか

Aliceのウォレットで作られるトランザクションは258byteで、資金の所有者を確認し新しい所有者を割り当てるのに必要な全てがこれに含まれています。このトランザクションがBitcoinネットワークに送られてはじめて、分散元帳であるブロックチェーンの一部になります。この節では、どのようにトランザクションが新しいブロックの一部になるのか、どのようにブロックが "マイニング" されるのか、を確認します。また、新たにブロックチェーンに加えられたブロックへの信用が、その後ブロックが追加されるとともに、ネットワークによって高められる様子を、最後に見ていきます。

## Bitcoinネットワークへのトランザクションの送信

トランザクションはブロックチェーンに取り込まれるための情報を全て持っているため、どこで、どのようにBitcoinネットワークに送信されても構いません。Bitcoinネットワークはpeer-to-peerネットワークであり、個々のBitcoinクライアントは、いくつかの他のBitcoinクライアントと繋がることで、Bitcoinネットワークに参加しています。Bitcoinネットワークの目的は、トランザクションとブロックを全てのBitcoinクライアントに伝えることです。

## どのようにBitcoinネットワークを伝わって行くのか

Aliceのウォレットは、有線、WiFi、モバイル、何であれインターネットに繋がっていれば、どのBitcoinクライアントに対しても、新しいトランザクションを送ることができます。AliceのウォレットはBobのウォレットと直接繋がっている必要はなく、Bobのカフェが提供しているインターネットアクセスポイントを使う必要もないのです。Bitcoinネットワークのノード(クライアント)は、有効な見たことのないトランザクションを受け取ると、繋がっている他のノードに即座にそのトランザクションを転送します。これによって、このトラン

ザクションは迅速にpeer-to-peerネットワークを伝わって行き、数秒以内に大半のノードに到達するのです。

### Bobの視点で見たときは

Bobのウォレットが直接Aliceのウォレットと繋がっている場合、Aliceから送られるトランザクションを最初に受けるノードは、Bobのウォレットかもしれません。しかし、たとえAliceのウォレットが他のノードを通してトランザクションを送ったとしても、トランザクションは数秒以内にBobのウォレットに到達するでしょう。BobのウォレットはすぐにこのトランザクションをBobへの支払いであると認識します。というのは、このトランザクションはBobの秘密鍵で復号できるアウトプットになっているからです。Bobのウォレットは独立に、このトランザクションが正規の形式であるか、未使用のインプットを使っているか、トランザクションをブロックに取り込んでもらうのに十分なトランザクション手数料を含んでいるかの確認も行います。この時点でBobは、多少のリスクはありますが、このトランザクションがすぐにブロックに含められ承認されるとみなせます。

#### TIP

Bitcoinトランザクションに関するよくある誤解は、“承認”のために新しいブロックが生成されるまで10分間待たなければならないとか、完全な6回の承認のために60分間待たなければならないといったことです。承認は、トランザクションがBitcoinネットワーク全体に受け入れられたことを保証しますが、このように待つことはコーヒー一杯のような少額の商品には必要ありません。店舗側は、承認がない場合でも、いつも彼らが受け入れている個人IDや署名がないクレジットカードよりリスクが大きいなら、有効な少額のトランザクションを受け入れるでしょう。

## Bitcoinマイニング

トランザクションはBitcoinネットワークに伝えられました。しかし、\_マイニング\_と呼ばれるプロセスを通して検証されブロックに取り込まれるまで、共有されている元帳である\_ブロックチェーン\_の一部になることはできません。詳細な説明は[\[ch8\]](#)を参照してください。

Bitcoinにおける信用の仕組みは、計算によって成り立っています。トランザクションがブロックの中に取り込まれるためには膨大な計算を必要としますが、取り込まれていることを検証するにはわずかな計算しか必要ありません。このマイニングは、以下の2つの目的のために行うものです。

- マイニングは、それぞれのブロックの中に新しいbitcoinを作り出します。これは、あたかも中央銀行が新しいお金を印刷するようなものです。ブロックごとに作り出されるbitcoinの量は決められており、時間とともに減少していきます。
- マイニングは、信用を作り出します。マイニングは、「トランザクションを含むブロックに十分な計算量がつぎ込まれた場合にのみ、そのトランザクションが承認される」ということを保証することによって、信用を作り出します。より多くのブロックがあるということは、より多くの計算量を要したことを意味し、従って、より多くの信用を得ていることを意味するのです。

マイニングを説明するためには、数独パズルを考えるのが分かりやすいです。それは、誰かが解法を見つけるごとにリセットされて、約10分間で解けるように難しさが自動的に調整されるような数独パズルです。数千の行と列を持つ、巨大な数独パズルを想像してください。私があなたに完成したパズルを見せたら、完成したことを確認することはすぐにできます。しかし、パズルがある部分だけ完成していて他が全て空欄であれば、解くためにとても多くの時間がかかってしまいます。数独パズルの難しさは、行や列の数を増やしたり減らしたりすることで調整できます。しかし、完成したかを確認することについては、パズルの大きさによらず短時間でできます。Bitcoinで使っているこのような“パズル”は、暗号化ハッシュに基づいており、上記の数独パズルと似た特徴を持っています。すなわち、解くのはとても大変なのに確認するのは簡単という非対称性と、

難しさを調整できるという特徴です。

[[user-stories](#)]では、上海でコンピュータエンジニアリングを学ぶ学生であるJingを例として挙げました。JingはマイナーとしてBitcoinネットワークに参加しています。Jingは、世界中の数千のマイナーとともに、約10分毎に解法を見つけるレースに参加しているのです。このような解法を見つける作業は"proof of work"と呼ばれ、秒間数千兆回のハッシュの生成処理を必要とします。proof of workのアルゴリズムは、前もって決められたパターンに合う解法が現れるまで反復的に、ブロックのヘッダとランダム値をSHA256暗号化アルゴリズムでハッシュ化することです。そのような解法を最初に見つけたマイナーがそのブロックの勝者となり、解法を見つけたブロックをブロックチェーンに組み込みます。

Jingは2010年に、新しいブロックのproof of workを見つけるために、非常に速いデスクトップコンピュータを使って、マイニングを始めました。多くのマイナーがBitcoinネットワークに参加するにつれ、解法を得る難しさはどんどん増していきました。Jingと他のマイナーは、すぐにさらに特殊なハードウェア、例えば、ゲーム用デスクトップコンピュータに用いられるような、ハイエンドの専用グラフィック処理装置(GPU)などにアップグレードしました。この本を書いている時点で、数百のマイニングアルゴリズムを組み込んだハードウェアであるASICという回路を、複数同時に稼働させて、ようやく利益が出るくらいに難しいものになっています。Jingは、"マイニングプール"にも参加しました。これは、解法を見つける作業の負担を何人かで分担し、報酬も参加者で分けるという、宝くじの共同購入のようなものです。Jingは現在、24時間マイニングを行うためにUSB接続のASICマシンを2台使っています。彼はマイニングで得たbitcoinを売ることによって電気代を支払いながら、収益をあげています。彼のコンピュータ上では、Bitcoinクライアントのリファレンス実装であるbitcoindのコピーを走らせており、これを特殊なマイニングソフトウェアのバックエンドとして使っています。

## ブロック内のトランザクションのマイニング

Bitcoinネットワークに送られたトランザクションは、グローバルに分散した元帳であるブロックチェーンの一部となるまでは、検証されたことにはなりません。平均して10分ごとに、マイナーはブロックチェーンに取り込まれていないトランザクションを含むブロックを生成します。新しいトランザクションは、ウォレットやその他のソフトウェアから常にBitcoinネットワークに流れ込みます。Bitcoinネットワークのノードがこの新しいトランザクションを見つけると、各ノードの中にある、未検証のトランザクションを一時的にとどめておくトランザクションプールに加えます。マイナーは、新しいブロックを作るとき、未検証のトランザクションをこのプールから取り出して新しいブロックに追加します。その上で、新しいブロックの有効性を証明するために(proof of workとして知られる)非常に難しい問題を解きます。このマイニングの過程の詳細は[[mining](#)]で説明されています。

トランザクションは新しいブロックに追加されますが、この新しいブロックには、最も高い手数料が設定されているものが最初に処理されるという基準や、その他のいくつかの基準によって、処理の優先順位が付けられています。マイナーは、Bitcoinネットワークからマイニングされたブロックを受け取ったことで競争に負けたことを知るとすぐに、新しいブロックのマイニングに取りかかります。マイナーはすぐに新しいブロックの箱を作り、それにトランザクションと前のブロックのハッシュ値を入れて、その新しいブロックのためのproof of workの計算を開始します。マイナーは自分が作るブロックに、特別なトランザクションを含めます。これは、マイナー自身のBitcoinアドレスに、新たに作られたbitcoinの報酬(現在は1ブロックあたり25BTC)を支払うトランザクションです。マイナーは、ブロックが有効であることを示す解法を見つけると、報酬を勝ち取ります。このマイナーが解法を見つけたブロックがグローバルなブロックチェーンに追加され、報酬を得るために含めたトランザクションがマイナーにとって利用可能となるからです。マイニングプールに参加しているJingは、彼のソフトウェアをあらかじめ設定しておきます。これによって提供した計算量に応じて分けられた報酬が、プールのアドレスからJingやその他のマイナーに配られます。



AliceのトランザクションはBitcoinネットワークによって回収され、未検証のトランザクションのプールに入れます。そのトランザクションは十分な手数料を含んでいたため、Jingが参加するマイニングプールが作ったブロックに入ることになりました。Aliceのウォレットがトランザクションを送ってから約5分後に、JingのASICマイナーがブロックの解法を見つけ、他に419のトランザクションを含むそのブロックを #277316としてビットコインネットワーク上に放出しました。他のマイナーがそれを検証し、それが終わると、次のブロックを作るレースが始まりました。

Aliceのトランザクションを含むブロックを、見ることができます([Aliceのトランザクション](#))。

数分後に、新しいブロック #277317  
が別のマイナーによってマイニングされました。この新しいブロックは、Aliceのトランザクションを含んだ直前のブロック #277316 を前提にしているため、#277316がマイニングされたときよりもさらに多くの計算がブロックチェーンに注ぎ込まれ、それによりAliceのトランザクションの信用が強化されることとなります。Aliceのトランザクションを含んでいるブロックは、  
"承認" 1回とカウントされます。  
Aliceのトランザクションを含むブロックの上に、新たにブロックが積み重ねられるごとに、承認が積み重ねられることとなります。ブロックが積み重ねられるにつれて、トランザクションの取り消しが指数関数的に難しくなり、このことで、そのブロックのBitcoinネットワークにおける信用がますます増えるのです。

ブロック #277316 に含まれているAliceのトランザクション図には、Aliceのトランザクションを含むブロック #277316 が示されています。ブロック #277316 の下には (#0を含めて)277316個のブロックあり、genesisブロックとして知られる#0まで、全てのブロックがブロックの連鎖(ブロックチェーン)として繋がっているのです。時間の経過とともにブロックの"高さ"が高くなると、計算の難易度はより高くなります。チェーンが長くなるほど計算量が積み重なることになるため、Aliceのトランザクションが含まれるブロックの後にマイニングされたブロックは、さらなる保証として働きます。慣例的に、6回より多くの検証がなされたブロックは改変ができないと考えられています。というのは、6個のブロックを無効にし、再計算するためには、膨大な計算量が必要だからです。マイニングの過程やマイニングが信用を作り出す方法は、第8章で詳しく説明します。

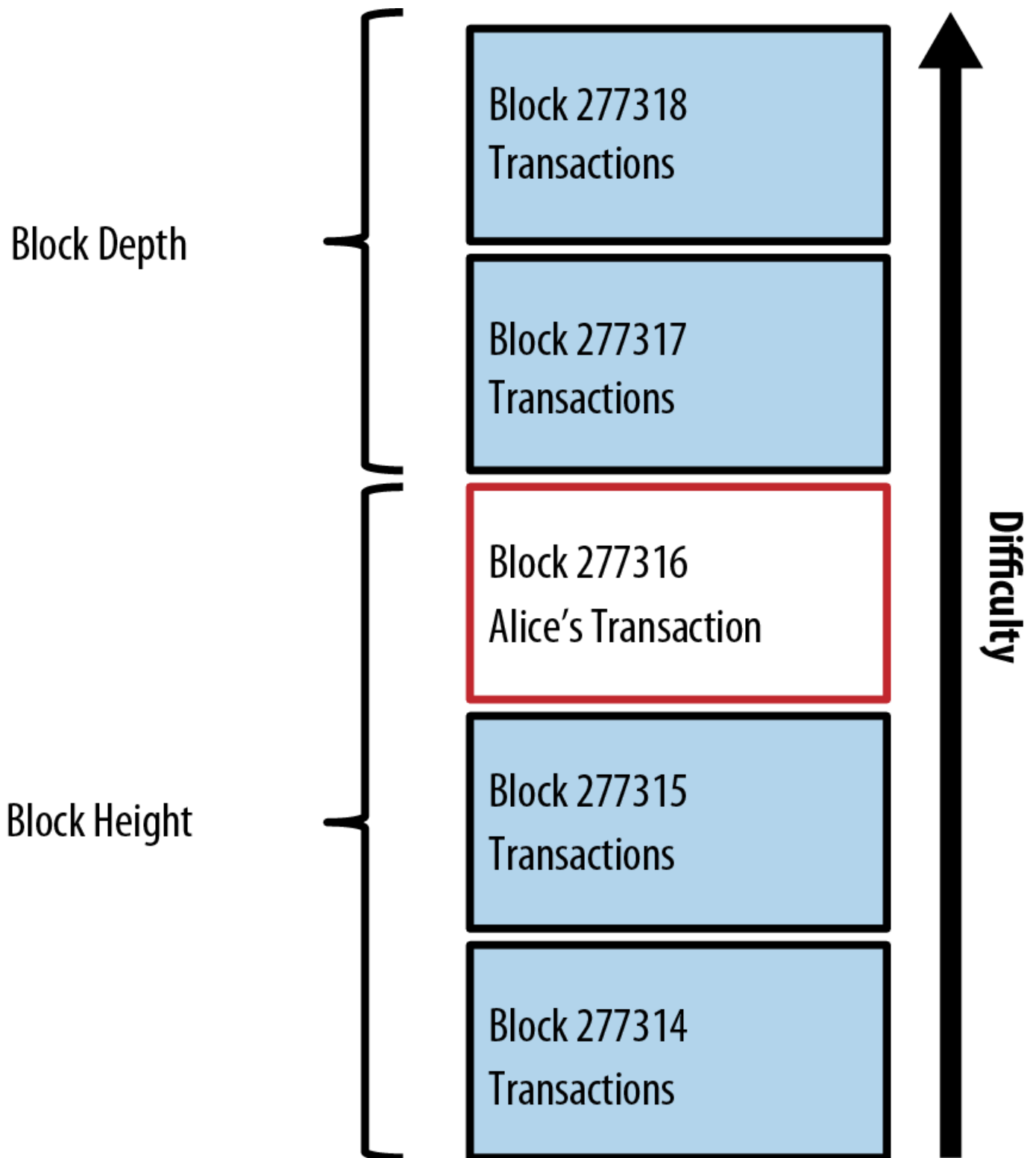


Figure 9. ブロック #277316 に含まれているAliceのトランザクション

## トランザクションの使用

Aliceのトランザクションはブロックの一部としてブロックチェーンに埋め込まれ、Bitcoinの分散された元帳の一部として全てのBitcoinアプリケーションから参照できるようになりました。それぞれのBitcoinクライアントは、独立にトランザクションが有効で使用可能かを検証できます。フルインデックスクライアントは、Bobに支払ったbitcoinが最初に生成され、ひとつひとつのトランザクションを経て、Bobのアドレスにたどり

着くまでの、全ての軌跡を追うことができます。軽量クライアントは"SPV(simplified payment verification)"([spv\_nodes]参照)と呼ばれる検証を行います。すなわち、トランザクションがブロックチェーンに含まれ、そのトランザクションの後にマイニングされたブロックがいくつかあることを確認し、トランザクションが有効であるとネットワークが受け入れるようにします。

Bobは今や、新たなトランザクションを作り、Aliceや他の人とのトランザクションで得たアウトプットをインプットとして参照し新しい所有者に割りあてることで、これらのアウトプットを支払いに使うことができます。例えば、Bobは、Aliceから支払われたコーヒーの代金を送ることで、契約者やサプライヤーに支払いができるのです。BobのBitcoinソフトウェアは、たくさんの少額の支払いを、ひとつのより大きい金額の支払いにまとめるでしょう。もしかしたら、一日のbitcoin収入全てをまとめて1つのトランザクションに集約しているかもしれません。このトランザクションは、いろいろな支払いを、店舗の"決済"口座として使っている単一のBitcoinアドレスに移します。集約型トランザクションで、この集約型トランザクションを解説しています。

Bobが、Aliceや他のお客さんから受け取った支払いを使うほど、トランザクションの連鎖を伸ばすことになります。それは、参加者全員が確認し信用するグローバルなブロックチェーンに、これらのトランザクションが追加されることを意味します。Bobは新しいウェブページを作るためにBangaloreにいるウェブデザイナーGopeshに支払いをすることを考えてみましょう。トランザクションの連鎖はJoeからGopeshへのトランザクションの連鎖の一部としてのAliceのトランザクション図のようになっています。

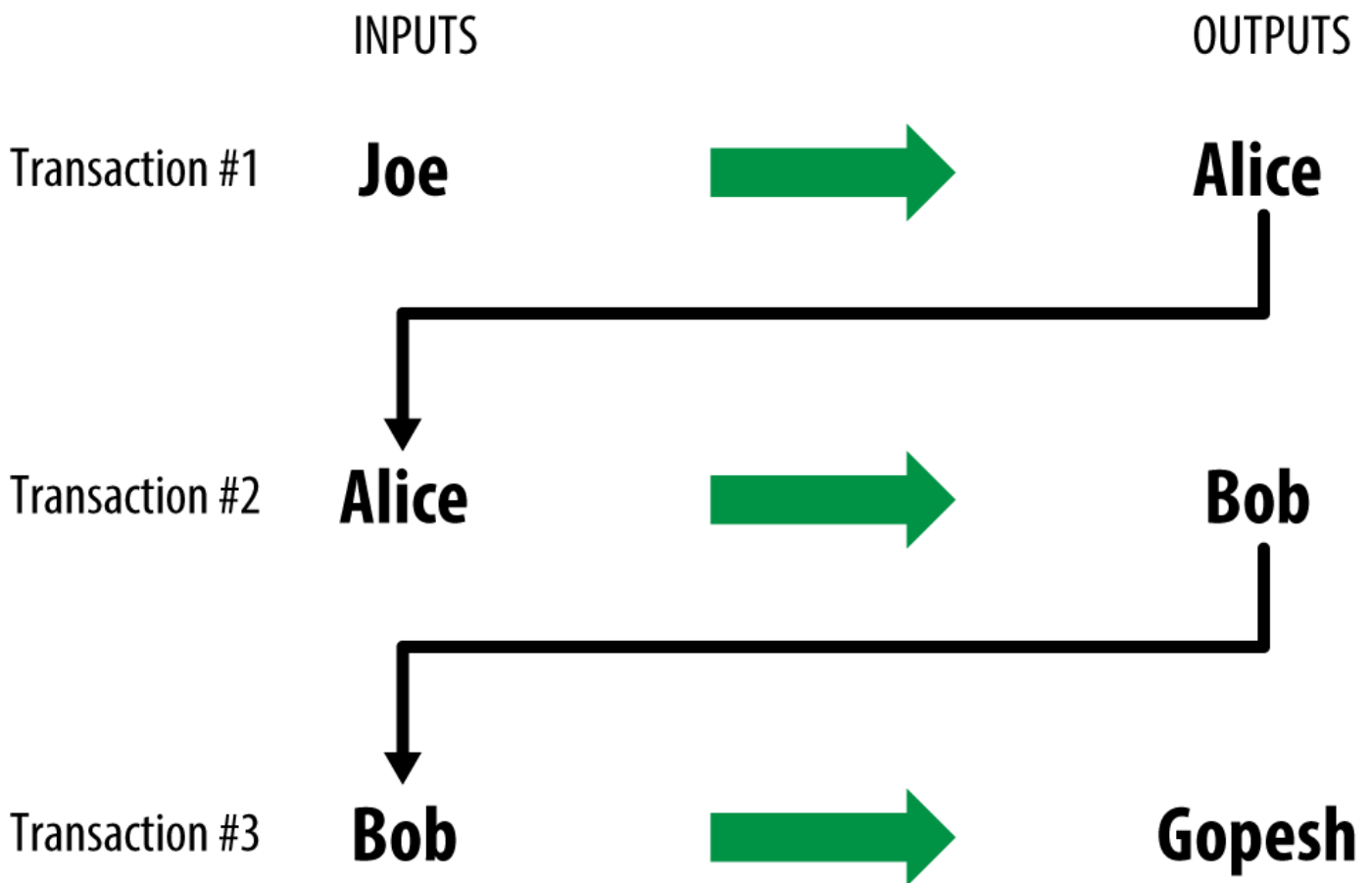


Figure 10. JoeからGopeshへのトランザクションの連鎖の一部としてのAliceのトランザクション

# Bitcoinクライアント

## Bitcoin Core：リファレンス実装

BitcoinリファレンスクライアントであるBitcoin Core("Satoshiクライアント")は bitcoin.orgからダウンロードすることができます。このクライアントはBitcoinの仕組みの全ての機能、ウォレット、トランザクション確認エンジン(全てのブロックチェーンのコピーを保持)、peer-to-peer Bitcoinネットワークノード、を実装しています。

Bitcoinリファレンスクライアントをダウンロードするために [ウォレット選択ページ](#) でBitcoin Coreを選んでください。ご自身で使っているOSに合わせてインストーラーをダウンロードできます。もしWindowsなら、ZIPアーカイブか.exe実行ファイルです。もしMac OSなら、.dmgディスクイメージ。Linuxなら、UbuntuのPPA/パッケージかtar.gzアーカイブです。 [bitcoin.org](#) [でのBitcoinクライアント選択図](#)にあるように、bitcoin.orgには推奨されるBitcoinクライアントがリストアップされています。

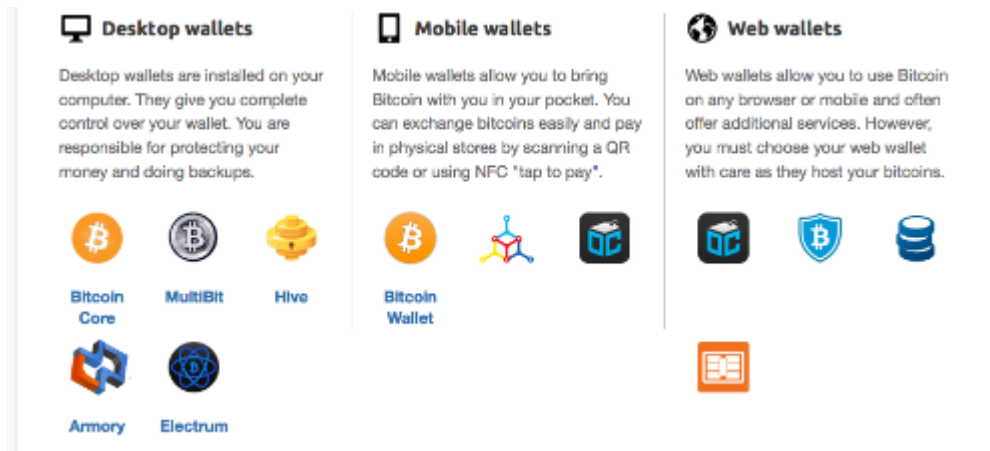


Figure 1. bitcoin.org でのBitcoinクライアント選択

### 最初にBitcoin Coreを実行するには

まずインストール可能なパッケージファイル(.exe, .dmg, PPAなど)をダウンロードしてください。Windowsなら、.exeファイルを実行しステップごとのインストールプロセスに従います。Mac OSなら、.dmgファイルを実行しBitcoin-QT

アイコンをアプリケーションフォルダにドラッグしてください。Ubuntuなら、PPAファイルをダブルクリックするとパッケージマネージャーが起動します。一度インストールが完全に終わると、アプリケーションリストにBitcoin-QTというソフトウェアがインストールされているはずで

Bitcoinクライアントを起動するためにアイコンをダブルクリックしてみてください。

### Bitcoin

Coreを最初に起動したとき、まずブロックチェーンをダウンロードし始めます。これには数日間かかるかもしれませんが([ブロックチェーン初期構築時のBitcoin Coreスクリーン図参照](#))。画面に"同期完了(Synchronized)"と出るまでバックグラウンドで動かしておいてください。"同期完了"と出ると、もう"同期されていません(out of sync)"とは表示されなくなります。

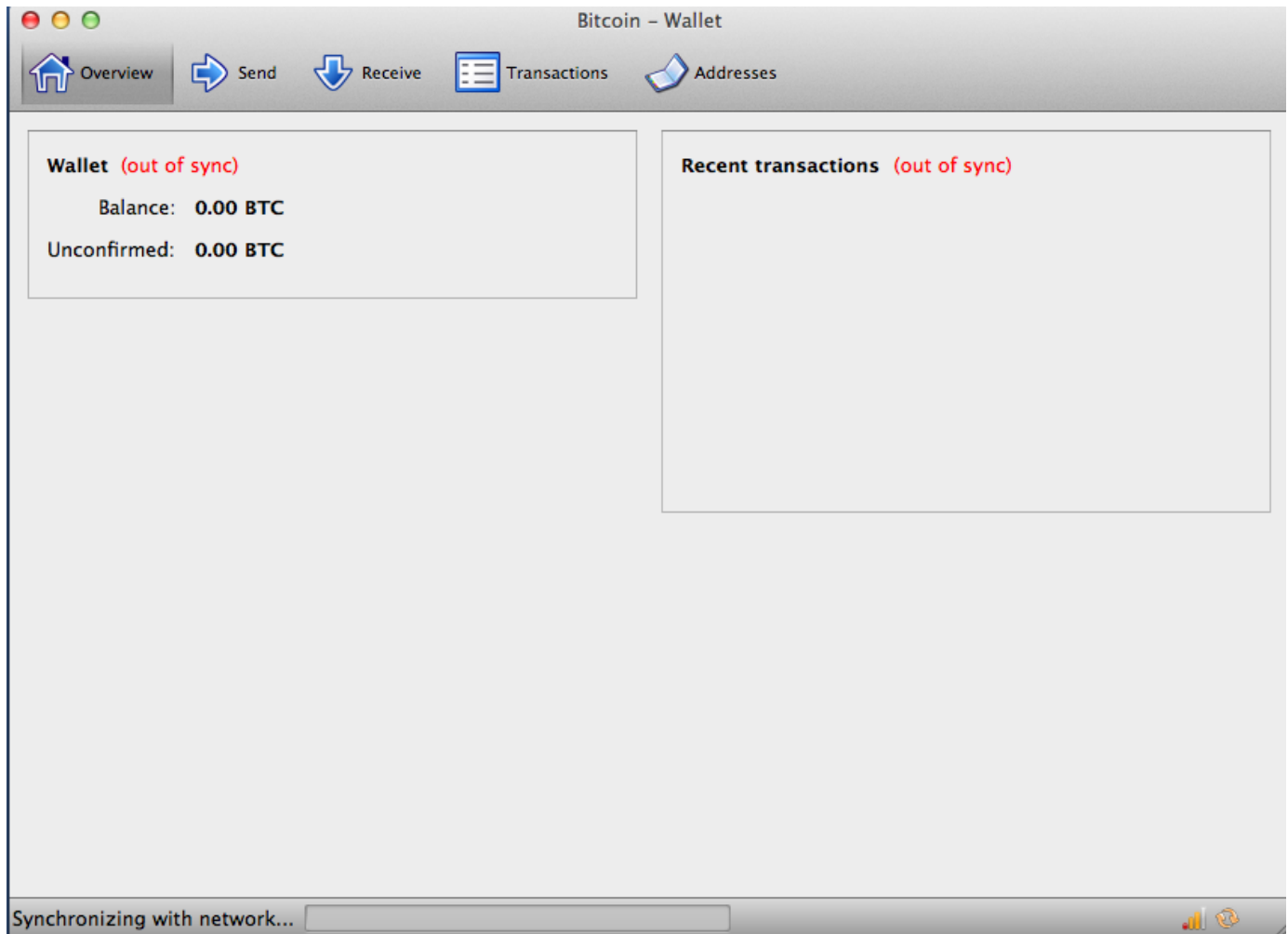


Figure 2. ブロックチェーン初期構築時のBitcoin Coreスクリーン

**TIP**

Bitcoin Coreはトランザクション元帳(ブロックチェーン)の完全なコピーを保持しており、2009年にBitcoinネットワークが稼働し始めてからBitcoinネットワークで起きた全てのトランザクションを含んでいます。このデータセットは数十GB(2013年後半の時点で約16GB)あり、数日かけてダウンロードされます。フルブロックチェーンデータセットをダウンロードするまで、クライアントはトランザクションを処理することも口座残高を更新することもできません。クライアントの画面には、ダウンロードの間ずっと口座残高の横に"同期されていません(out of sync)"と表示され、下部に"同期中(Synchronizing)"と表示されます。最初の同期を完了するために十分な空きディスク容量、帯域幅、時間があるかを事前に確認しておいてください。

## ソースコードからのBitcoin Coreコンパイル

開発者向けに、ZIPアーカイブとしてソースコードをダウンロードする、またはGitHubからソースコードをcloneしてすることもできます。

[GitHub](#)

[bitcoin](#)ページのサイドバーから

ZIPアーカイブのダウンロードが選べます。また、gitのコマンドを使うことでローカルコピーを作ることができます。次の例は、LinuxやMac OSなどUnixのようなOS上でコマンドを実行してソースコードをcloneしています。

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

#### TIP

導入手順とスクリーンに表示される結果はバージョンによって変わるかもしれません。もしここで見た導入手順と違うことがあれば、ダウンロードしてきたソースコードに付いているドキュメントに従ってください。また、スクリーンに表示される結果がここにある例とわずかに違っていても驚かないでください。

`git` `clone`が終わると、`bitcoin` というディレクトリの中に完全なソースコードがダウンロードされます。プロンプトの次に `cd bitcoin` と打ち込んでディレクトリを移ってください。

```
$ cd bitcoin
```

デフォルトでローカルコピーは最新のソースコードと同期されているので、そのソースコードは不安定であったり、またベータ版であったりするかもしれません。ソースコードをコンパイルする前に、("release tags" リリースタグを確認して特定のバージョンを選択します。これは、タグが付けられたソースコードのスナップショットとローカルコピーを同期するという事です。タグは、特定のリリースに目印をつけるために開発者がソースコードに付与したバージョン番号です。まず、`git tag` コマンドを実行して同期できるタグを確認してみましょう。

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12
```

[... 多くのタグがあるため省略 ...]

```
v0.8.4rc2
v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

このタグリストはBitcoinの全てのリリースタグを示しています。慣習に沿って、テストが必要な

リリース候補

には

"rc"という接尾詞がついています。商用環境で使用できる安定なソースコードにはこの接尾詞はつきません。

前のリストから最も大きいバージョンのタグ `v0.9.0rc1` (執筆時点

)を選択しましょう。ローカルコピーとこのバージョンのソースコードを同期するために、`git checkout`コマンドを実行します。

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

ソースコードにはドキュメントファイルも入っています。 `more` `README.md`  
とプロンプトのところに入力して `README.md`  
というドキュメントファイルを読んでみましょう。スペースキーを押すことで次のページに移ることができます。この章では、コマンドラインから操作できるLinux上で動作するBitcoinクライアント `bitcoind`  
をビルドしてみます。 `more` `doc/build-unix.md` を入力して  
`bitcoind`のコンパイル説明書を読んでみましょう。またMac OS XやWindowsに対しても同様の `doc`  
ディレクトリがあり、それぞれ `build-osx.md`、`build-msw.md` というコンパイル説明書があります。

コンパイル説明書の最初の部分を注意深く読むと、ビルド必須事項があります。これらは、コンパイルを始める上で必要なライブラリです。もし欠けているようであれば、コンパイルは途中で失敗してしまいます。もしビルド必須事項にあるものが足りなければ、それをインストールし失敗してしまったところからコンパイルをやり直してください。ビルド必須事項が全てインストールされていると考えて、 `autogen.sh`  
というスクリプトを使って`bitcoind`のビルドを始めてみましょう。

**TIP** Bitcoin Coreのビルド手順は、version 0.9から `autogen/configure/make`を使う形に変わりました。昔のバージョンだと簡単なMakefileを使い、例とわずかに違った形の手順になっています。あなたがコンパイルしたいバージョンの導入手順に従ってください。0.9で導入された`autogen/configure/make`は今後のバージョンで使われるビルド方法です。

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

`autogen.sh`

というスクリプトは自動的に設定スクリプトを生成するスクリプトです。あなたのシステムの設定やコンパイルに必要なライブラリがあるかをチェックしてくれます。最も重要なスクリプトは、`bitcoind`をビルドする上での数多くのカスタマイズ方法を設定できる `configure` スクリプトです。 `./configure --help` と入力していろいろなオプションを見てみてください。

```
$ ./configure --help
```

```
`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.
```

```
Usage: ./configure [OPTION]... [VAR=VALUE]...
```

```
To assign environment variables (e.g., CC, CFLAGS...), specify them as  
VAR=VALUE. See below for descriptions of some of the useful variables.
```

```
Defaults for the options are specified in brackets.
```

```
Configuration:
```

```
-h, --help          display this help and exit  
  --help=short      display options specific to this package  
  --help=recursive  display the short help of all the included packages  
-V, --version       display version information and exit
```

```
[... さらに多くのオプション、変数が以下に表示されます ...]
```

```
Optional Features:
```

```
--disable-option-checking ignore unrecognized --enable/--with options  
--disable-FEATURE        do not include FEATURE (same as --enable-FEATURE=no)  
--enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
```

```
[... 多くのオプションが出てくるため省略 ...]
```

```
Use these variables to override the choices made by `configure' or to help  
it to find libraries and programs with nonstandard names/locations.
```

```
Report bugs to <info@bitcoin.org>.
```

```
$
```

configure スクリプトによってbitcoind  
のある機能を有効化したり無効化したりできます。これを行うには、configureスクリプトの後ろに --enable  
-FEATURE や --disable-FEATURE といった形でフラグを設定して実行します。この FEATURE  
のところにはhelpを実行したときに出ている各機能の名前が入ります。この章では、全てのデフォルト機能を入れたbitcoindクライアントをビルドすることにします。ここは設定フラグを使いませんが、どんな機能を付けられるのか確認してみた方がよいでしょう。次にconfigureスクリプトを実行して、必要なライブラリを自動的にチェックしカスタマイズされたビルドスクリプトを生成していきます。



```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
```

[... システムが持っている多くの機能がチェックされます ...]

```
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

全てがうまくいったら、`configure` コマンドはカスタマイズされたビルドスクリプトを生成して終了します。何か足りないライブラリがあったりエラーがあったりすると、`configure` コマンドはビルドスクリプトを生成することなくエラーを出して終了してしまいます。もしエラーが出たら、おそらくライブラリ自体がないか、またはライブラリがあってもそのバージョンのライブラリと `bitcoind` の相性が悪いかでしょう。構築ドキュメントをもう一度確認し足りないものをインストールし `configure` コマンドを実行するとエラーが解消されます。次にソースコードをコンパイルします。これには1時間ほどかかります。コンパイルが実行されている間、数秒に1回、数分に1回は表示されるメッセージを確認した方がよいです。というのは何かエラーが起きて止まってしまうことがあるからです。コンパイルはいつでも止まってしまったところから再開できます。 `make` と入力してコンパイルを始めてみてください。

```

$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX    addrman.o
  CXX    alert.o
  CXX    rpcserver.o
  CXX    bloom.o
  CXX    chainparams.o

[... 多くのコンパイルメッセージが続きますが省略 ...]

  CXX    test_bitcoin-wallet_tests.o
  CXX    test_bitcoin-rpc_wallet_tests.o
  CXXLD  test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$

```

makeがエラーなく実行されると、コンパイルされたbitcoindが生成されます。最後に、この実行可能なbitcoind(コンパイルされたbitcoind)をシステム上の適切なところにインストールするために `make install` コマンドを使ってインストールします。

```

$ sudo make install
Making install in src
Making install in .
  /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make install-am
  /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$

```

bitcoindが正常にインストールされたかは、以下のように2つのコマンドがどこに配置されているかを表示するコマンドを使うことで確認できます。

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

デフォルトでbitcoindは `/usr/local/bin` に配置されます。最初にbitcoindを実行したときに、JSON-RPCを使うための強力なパスワードの設定を含む設定ファイルを作るようにbitcoindから言われます。bitcoind とプロンプトのところに入力してbitcoindをスタートさせてください。

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configuration file:
/home/ubuntu/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

好きなエディタを使って設定ファイルを編集し、bitcoindに言われたようにパスワードを強力なものにしてください。以下に書かれたパスワードを使っては いけません。 `.bitcoin` ディレクトリの中に `.bitcoin/bitcoin.conf` というファイルを作成し、ユーザ名とパスワードを入力してください。

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
```

設定ファイルを編集しているときに、いくつか別のオプションを設定したいかもしれません。例えば、`txindex` とかです([トランザクションデータベースインデックスとtxindexオプション参照](#))。他の設定可能なオプションについては、`bitcoind --help` と入力して実行することで表示されます。

## Bitcoin

Coreクライアントを実行してみましょう。最初に全てのブロックをダウンロードしてブロックチェーンを構築し始めます。ブロックチェーンのファイルは1GB以上あるため平均してダウンロードに2日くらいかかります。BitTorrentクライアントを使って部分的な `blockchain.dat` ブロックチェーンを [SourceForge](#) からダウンロードしてブロックチェーン初期化処理を短くすることもできます。

bitcoindをバックグラウンドで実行したい場合は、オプションとして `-daemon` を付けて実行してください。

```

$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... 多くの起動メッセージがありますが省略 ...]

```

## コマンドラインからのBitcoin Core JSON-RPC APIの使用

Bitcoin CoreクライアントにはJSON-RPC APIが実装されており、`bitcoin-cli` コマンドでJSON-RPC APIにアクセスすることもできます。 `bitcoin-cli` コマンドを使うことで`bitcoind`と1つ1つコミュニケーションをとるようにJSON-RPC APIを使って利用することができます。また、プログラムからJSON-RPC APIを使うこともできます。始めるために以下のように `help` を使って利用できるRPCコマンドを確認してみましょう。

```

$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )

```

```

getblockchaininfo
getblockcount
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount

```

```
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" (
[{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...]
["privatekey1",...] sighashtype )
stop
submitblock "hexdata" ( "jsonparametersobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numblocks )
verifymessage "bitcoinaddress" "signature" "message"
wallelock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

## Bitcoin Coreクライアントのステータスの取得

コマンド: getinfo

Bitcoinのgetinfo RPCコマンドは、  
Bitcoinネットワークノード、ウォレット、ブロックチェーンデータベースについての基本的な情報を表示します。以下の bitcoin-cli コマンドを実行してみてください。

```
$ bitcoin-cli getinfo
```

```
{
  "version" : 90000,
  "protocolversion" : 70002,
  "walletversion" : 60000,
  "balance" : 0.00000000,
  "blocks" : 286216,
  "timeoffset" : -72,
  "connections" : 4,
  "proxy" : "",
  "difficulty" : 2621404453.06461525,
  "testnet" : false,
  "keypoololdest" : 1374553827,
  "keypoolsize" : 101,
  "paytxfee" : 0.00000000,
  "errors" : ""
}
```

このデータは Javascriptオブジェクト記法(JSON)で返されます。  
JSONは多くのプログラミング言語で簡単に利用できるとともに、人間にとっても読みやすいものです。このデータの中に、Bitcoinクライアントのバージョン情報(90000)やプロトコルバージョン情報(70002)、ウォレットバージョン情報(60000)があります。ウォレットにある現在の残高を見ると0になっています。"blocks"と

あるところには、現在何ブロックをBitcoinクライアントが把握しているかというブロック高(286216)があります。また、Bitcoinネットワークに関するいろいろな統計情報も見ることができ、このクライアントの設定内容も見ることができます。この章の残りの部分で、これらの設定値についてもっと詳細に分け入ってみます。

#### TIP

bitcoindクライアントが他のBitcoinクライアントからブロックをダウンロードしながら、現在のブロックチェーンの高さに"追いつく"ためにおそらく1日以上かかるでしょう。getinfoコマンドを使うことで追い付いたブロック高を把握することができ、現在の進捗を知ることができます。

## ウォレットセットアップと暗号化

コマンド: encryptwallet, walletpassphrase

秘密鍵の生成やその他のコマンドに進む前に、まず最初にウォレットをパスワードで暗号化しておくべきです。この例では、"foo"というパスワードとともに encryptwallet コマンドを使います。言うまでもないことですが、必ず"foo"をもっと強力で複雑なパスワードに置き換えてください!

```
$ bitcoin-cli encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The
keypool has been flushed, you need to make a new backup.
$
```

ウォレットが暗号化されたかどうかはgetinfoコマンドを実行することで確認できます。暗号化すると getinfo コマンドの実行結果に unlocked\_until という新しい項目が表示されるようになります。これはどれだけの時間メモリの中にパスワードを保持しておくかというカウンターを表します。暗号化直後にgetinfoコマンドを実行するとunlocked\_untilは0になっています。つまり、ロックされています。

```
$ bitcoin-cli getinfo
```

```
{
  "version" : 90000,
  #[...] その他の情報は省略 ...]
  "unlocked_until" : 0,
  "errors" : ""
}
$
```

ウォレットのロックを解除するためには、 walletpassphrase コマンドが必要になります。このコマンドは、パスワードとウォレットを再びロックするまでの秒数という2つのパラメータが必要です。

```
$ bitcoin-cli walletpassphrase foo 360
$
```

getinfoコマンドを再度実行することでウォレットのロックが解除されていること、および再びロックされる時刻が確認できます。

```
$ bitcoin-cli getinfo
```

```
{
  "version" : 90000,
  #[...] その他の情報は省略 ...]
  "unlocked_until" : 1392580909,
  "errors" : ""
}
```

## ウォレットバックアップ、プレーンテキストダンプ、リストア

コマンド: backupwallet, importwallet, dumpwallet

次にウォレットバックアップを作り、このバックアップからウォレットをリストアする練習をしてみます。パラメータとしてバックアップファイル名を指定することで `backupwallet` コマンドを使ってバックアップをすることができます。ここでは、バックアップファイルとして `wallet.backup` というファイルを作成します。

```
$ bitcoin-cli backupwallet wallet.backup
$
```

バックアップファイルを用いてリストアするには `importwallet` コマンドを使います。ウォレットがロックされている場合には、最初にロックを解除しなければいけません(1つ前の `walletpassphrase` 参照)。

```
$ bitcoin-cli importwallet wallet.backup
$
```

`dumpwallet` コマンドは、ウォレットを人間が読めるテキストとしてダンプするときに使うことができます。



```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(0000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 #
addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 #
addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[...他にも多くのキーが出てきます...]
```

\$

## ウォレットアドレスと受信トランザクション

コマンド: `getnewaddress`, `getreceivedbyaddress`, `listtransactions`, `getaddressesbyaccount`, `getbalance`

BitcoinリファレンスクライアントはBitcoinアドレスプールやこのプールのデータサイズを管理しており、このプールのデータサイズは `getinfo` コマンドで表示される`keypoolsize`として表示されます。これらのBitcoinアドレスは自動的に生成され、パブリックなBitcoin受信アドレス、おつり受信アドレスとして使われます。Bitcoinアドレスを作るためには、`getnewaddress` コマンドを使ってください。

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

他のウォレット(例えば取引所やウェブウォレット、その他のbitcoindウォレット)からbitcoindウォレットに少額を送るためにこのBitcoinアドレスを使うことができます。この例では、さきほど作ったBitcoinアドレスに50mbits(0.050bitcoin)を送ってみることにしましょう。

今bitcoindクライアントに問い合わせることで、このBitcoinアドレスにbitcoinが届いたか、0.050bitcoinになるまでに何回の承認が必要とされたかを知ることができます。承認が0回のもものだけを見てください。こうすると、送金の数秒後にbitcoinがウォレットに反映されたことを確認することができます。承認回数を0回に指定しさきほどのBitcoinアドレスに対して`getreceivedbyaddress`コマンドを実行してみましょう。

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

もしコマンドの最後にある0を省略すると、最低でも`minconf`で設定されている回数だけ承認されないと送られたbitcoinが残高に反映されないようになっています。`minconf`とは、残高にトランザクションを表示する前に行う承認の回数の設定値です。`minconf`はbitcoindの設定ファイルの中で設定できます。今回のbitcoin

送付のトランザクションは送金から数秒経ってもまだ承認されておらず、よって残高が0と表示されているのです。

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

bitcoindにあるBitcoinアドレス全てに対して送られたトランザクションは  
コマンドを使うことで参照できます。

listtransactions

```
$ bitcoin-cli listtransactions
```

```
[
  {
    "account" : "",
    "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "category" : "receive",
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908
  }
]
```

全てのBitcoinアドレスは getaddressesbyaccount コマンドを使うことで参照できます。

```
$ bitcoin-cli getaddressesbyaccount ""
```

```
[
  "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
  "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
  "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
  "1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",
  "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
  "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
  "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
  "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
  "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
  "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
  "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
  "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
  "1LqJZz1D9yHxG4cLkdujngG5jNNGmPeAMD"
]
```

最後に、`getbalance` コマンドはこのウォレットにある総残高を表示します。ここに表示される数字は、`minconf`で設定された回数以上承認されたトランザクションについてのもののみです。

```
$ bitcoin-cli getbalance
0.05000000
```

**TIP**

もしトランザクションがまだ承認されていなければ、`+getbalance+`が返す残高は0になります。  
"`minconf`"オプションは、何回トランザクションが承認されれば残高に表示するかを設定しています。

## トランザクションのデコード & 解読

コマンド: `gettransaction`, `getrawtransaction`, `decoderawtransaction`

`gettransaction` コマンドを使ってさきほど表示された受信トランザクションを探索してみましょう。さきほど `listtransactions` コマンドのときに出てきた `txid` というトランザクションハッシュと、`gettransaction` コマンドでトランザクションをさらに細かく見ていきます。

```
{
  "amount" : 0.05000000,
  "confirmations" : 0,
  "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
  "time" : 1392660908,
  "timereceived" : 1392660908,
  "details" : [
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "category" : "receive",
      "amount" : 0.05000000
    }
  ]
}
```

#### TIP

トランザクションIDはトランザクションが承認されるまで信用できるものではありません。ブロックチェーン内にトランザクションハッシュがないというだけで、トランザクションがまだ処理されていないと判断することはできません。これは "トランザクション展性(transaction malleability)" と呼ばれるもので、トランザクションハッシュはブロック内で承認される前に変更され得るのです。承認された後、txidは不変になり信用できるものになります。

#### gettransaction

コマンドで表示されるトランザクション形式は簡略化されたものです。さらに細かいトランザクションの内容を見るためには、`getrawtransaction` コマンドと `decoderawtransaction` コマンドを使います。最初に、トランザクションハッシュ `txid` を引数とする `getrawtransaction` コマンドを実行すると、トランザクションが"生"の16進数テキストとして表示されます。

この16進数テキストを `decoderawtransaction` コマンドを使ってデコードしてみます。さきほどの16進数テキストをコピーして `decoderawtransaction` コマンドの1つ目の引数として貼付けて実行するとJSON形式として解釈された文字列が出てきます(16進数テキストになっているのは、以下の例にある長いJSONを短く格納しておくためです)。

デコードされたトランザクションには、トランザクションインプット/アウトプットを含む全ての項目が表示されます。この場合トランザクションには、さきほど作った新しいBitcoinアドレスへの50mbitsの送付に対応した1つのインプットとそれに対して生成された2つのアウトプットが含まれます。インプットは前に承認されたトランザクションのアウトプットだったので、`d3c7` で始まるvinのtxidのところに書かれています。2つのアウトプットは50mbits分の送金と送付元に送り返されるおつりを示します。

`gettransaction` コマンドなどを通して、ブロックチェーンの中にあるこのtxid(9ca8から始まる)の中身をさらに分け入っていくことができます。トランザクションからトランザクションへ次々に見ていくと、ある所有者からある所有者へのbitcoinが転送されていくトランザクションのチェーンをたどることができるのです。

一度受け取ったトランザクションが承認されると、`gettransaction`コマンドはこのトランザクションが含まれることになったブロックハッシュ(識別子)も返すようになります。

ここで、`ブロックハッシュ` と `ブロックインデックス` という新しい情報について見ていきます。`ブロックハッシュ` はトランザクションが含まれることになったブロックのハッシュ値で、`ブロックインデックス` は含まれることになったトランザクションがブロックの中の何番目のトランザクションとして入っているかを表し今の場合は18です(このトランザクションがブロック内の18番目にあったということを表します)。

## トランザクションデータベースインデックスと`txindex`オプション

デフォルトでBitcoin Coreは自身のウォレットに関係したトランザクションのみを含むデータベースを構築します。もし`gettransaction`コマンドなどで任意のトランザクションを見れるようにしたいなら、Bitcoin Coreに`txindex`というオプションを設定する必要があります。Bitcoin Core設定ファイル(通常はあなたのホームディレクトリの下に配置されている `.bitcoin/bitcoin.conf`)で `txindex=1` に設定してください。一度この変更をするとbitcoindを再起動して、`index`が再構築されるまで待たなければいけません。

## ブロック探索

コマンド: `getblock`, `getblockhash`

今やどのブロックに実行したトランザクションが含まれていたかが分かったので、そのブロックの中身を見てみましょう。ブロックハッシュを指定して `getblock` コマンドを実行してみてください。

このブロックには367個のトランザクションが含まれていて、見て分かるように18番目(9ca8f9...)のトランザクションが、我々のBitcoinアドレスに50mbitsを送金したトランザクションのtxidです。 `height` パラメータはこのブロックがブロックチェーンの286384番目のブロックであることを示しています。

また、 `getblockhash` コマンドの引数でブロック高を指定することでブロックハッシュを取得することもできます。

"genesisブロック"のブロックハッシュを取り出してみましょう。genesisブロックという一番最初のブロックはブロック高が0で、Satoshi Nakamotoによって掘り出されたものです。

`getblock`、`getblockhash`、`gettransaction` コマンドはプログラムからブロックチェーンを探索するために使われます。

## <phrase role="keep-together">未使用アウトプット</phrase>に基づくトランザクションの生成、署名、送信

コマンド: `listunspent`, `gettxout`, `createrawtransaction`, `decoderawtransaction`, `signrawtransaction`, `sendrawtransaction`

Bitcoinのトランザクションは、所有権の転送というトランザクションチェーンを作ります。このために前のトランザクションの結果である"アウトプット"を利用する形になっています。今やこのウォレットはBitcoin

アドレスにアウトプットに紐づけることができるトランザクションを受け取りました。一度これが承認されるとそのアウトプットを使うことができますようになります。

まずは `listunspent` コマンドを使って、このウォレットにある全ての未使用 アウトプットを表示してみましょう。

```
$ bitcoin-cli listunspent
```

トランザクション `9ca8f9...` には、Bitcoinアドレス `1hvzSo...` に紐づけられた50mbitのアウトプット(voutインデックスが0)が付加されていて、この時点でこのトランザクションは7回の承認を受けています。一般にトランザクションは前に作られたアウトプットを前のtxidとvoutインデックスを参照することでアウトプットを取得しインプットとして使います。これから、txid `"9ca8f9..."` の0番目のvoutをインプットとして使うトランザクションを作成し、新しい別のBitcoinアドレスにbitcoinを送る新しいアウトプットをこのインプットに紐づけます。

まず、このアウトプットの細かいところを探してみましょう。 `gettxout` コマンドを使うと未使用アウトプットの詳細を知ることができます。トランザクションアウトプットは常にtxidとvoutの組み合わせで指定でき、これらは `gettxout` コマンドを実行するときの引数となります。

`gettxout` コマンドで表示されたものは、Bitcoinアドレス `1hvz...` に紐づけられた50mbitのアウトプットです。このアウトプットを使うには、新しいトランザクションを作成します。まずこの50mbitのお金を送る新しいBitcoinアドレスを生成しましょう。

```
$ bitcoin-cli getnewaddress  
1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb
```

ウォレットが今作った新しいBitcoinアドレス `1LnfTn...` に25mbitを送ってみましょう。新しいトランザクションでは、前に出てきた50mbitのアウトプットを使い、この新しいBitcoinアドレスに25mbitを送ります。前のトランザクションから `1hvz...` 全てのアウトプットを使う必要があるため、Bitcoinアドレス `1hvz...` に返すおつり分も作らなければいけません。また、このトランザクションのトランザクション手数料も払う必要があります。この手数料を支払うために、0.5mbitだけおつりから差し引き、24.5mbitをおつりとして返します。新しいアウトプットのbitcoinの総和(25 mBTC + 24.5 mBTC = 49.5 mBTC)とインプットのbitcoin(50 mBTC)の差は、マイナーによってトランザクション手数料としてかき集められます。

`createrawtransaction` コマンドを使ってこのトランザクションを作成します。`createrawtransaction` コマンドの引数として、トランザクションのインプット(さきほど送った承認済トランザクションから来た50mbitの未使用アウトプット)と、新しいアドレスに送るお金と元のBitcoinアドレスに戻ってくるおつりの2つのアウトプットを指定します。

`createrawtransaction` コマンドは生成されたトランザクションをエンコードした16進数テキストを作ります。`decoderawtransaction` コマンドを使って、このトランザクションが正しいのか、人間には読みにくいこの16進数テキストをデコードして確認してみましょう。

これは正しそうです！この新しいトランザクションは、承認されたトランザクションから未使用アウトプット

を"消費"し、2つのアウトプットとして使用しました。1つは送付先の新しいBitcoinアドレスへの25mbit、もう1つは送付元のBitcoinアドレスに返ってくるおつりの24.5mbitです。差の0.5mbitはトランザクション手数料であり、この新しいトランザクションが含まれたブロックを探したマイナーに割り当てられます。

もう気づいたかもしれませんが、トランザクションには空の `scriptSig` が含まれています。なぜなら、まだ署名をしていないからです。署名がないと、このトランザクションは意味をなしません。未使用アウトプットが置かれていたBitcoinアドレスを `scriptSig` 所有しているということはまだ証明していないのです。署名によって、このアウトプットの解除条件を満たしこのアウトプットの所有者であることを証明すると、このアウトプットを使うことができます。 `signrawtransaction` コマンドを使うことでこのトランザクションに署名をします。このコマンドの引数として、署名をしたいトランザクションの16進数テキストを指定します。

#### TIP

暗号化されたウォレットでは、トランザクションを署名する前にウォレットのロックを解除しなければいけません。というのは、署名をするにはウォレットの秘密鍵にアクセスする必要があります。

この `signrawtransaction` コマンドはもう1つの16進数テキストを返します。 `decoderawtransaction` コマンドでこの16進数テキストをデコードして何が変わったのかを見てみましょう。

再度トランザクションを見てみると、インプットの `scriptSig` が空ではなくテキストが含まれていることが分かります。この `scriptSig` (はBitcoinアドレス `1hvz...` の所有者を証明している電子署名で、アウトプットの解除条件を満たしアウトプットが使用できるようになります。この署名が、Bitcoinネットワーク上のいかなるノードでもこのトランザクションを検証できるようにしているのです。

Bitcoinネットワークに新しく作ったトランザクションを送信する時が来ました。 `sendrawtransaction` コマンドを使って送信してください。 `sendrawtransaction` コマンドでは `signrawtransaction` コマンドで返ってきた16進数テキストを引数として指定します。これはさきほどデコードした16進数テキストです。

`sendrawtransaction` コマンドは、Bitcoinネットワークにトランザクションが送信されるとトランザクションハッシュ(`txid`) を返します。 `gettransaction` コマンドでこのトランザクションIDについて問い合わせることができるようになりました。

```

{
  "amount" : 0.00000000,
  "fee" : -0.00050000,
  "confirmations" : 0,
  "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
  "time" : 1392666702,
  "timereceived" : 1392666702,
  "details" : [
    {
      "account" : "",
      "address" : "1LnFTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "category" : "send",
      "amount" : -0.02500000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "category" : "send",
      "amount" : -0.02450000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1LnFTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "category" : "receive",
      "amount" : 0.02500000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "category" : "receive",
      "amount" : 0.02450000
    }
  ]
}

```

前にやったように、`getrawtransaction` コマンドと `decodetransaction` コマンドを使ってこのトランザクションのもっと詳細も調べることができます。これらのコマンドは、トランザクションをBitcoinネットワークに送る前に生成しデコードした16進数テキストと全く同じテキストを返します。

## 代替のBitcoinクライアント、ライブラリ、ツールキット

Bitcoinネットワークと通信するのに、Bitcoinリファレンスクライアントであるbitcoind以外に他にもBitcoinクライアントやライブラリがあります。これらは様々なプログラミング言語で実装されていて、プログラマー



にそれぞれのプログラミング言語ごとのインターフェイスを提供しています。

代替実装は以下です。

### *libbitcoin*

Bitcoin Cross-Platform C++ Development Toolkit

### *bitcoin explorer*

Bitcoin Command Line Tool

### *bitcoin server*

Bitcoin Full Node and Query Server

### *bitcoinj*

A Java full-node client library

### *btcd*

A Go language full-node bitcoin client

### *Bits of Proof (BOP)*

A Java enterprise-class implementation of bitcoin

### *picocoin*

A C implementation of a lightweight client library for bitcoin

### *pybitcointools*

A Python bitcoin library

### *pycoin*

Another Python bitcoin library

他にもいろいろなプログラミング言語で実装された多くのライブラリがあり、常に新しいライブラリが作り出されています。

## **LibbitcoinとBitcoin Explorer**

libbitcoinライブラリはクロスプラットフォームなC++ツールキットで、libbitcoin-server full-node とBitcoin Explorer (bx)コマンドラインツールをサポートしています。

bxコマンドはこの章で説明したbitcoindクライアントコマンドと同じ機能を多く持っています。bxコマンドはまた、bitcoindでは提供されていないいくつかの重要な管理ツールや操作ツールも持っており、ステルスアドレスやステルス支払い、ステルススクエリのサポートだけでなくtype-2決定性キーやmnemonicキーエンコーディングも提供されています。

### **Bitcoin Explorer**インストール

Bitcoin Explorerを使うには、単に [あなたが使用している](#)

OSに合った署名済み実行ファイルをダウンロードしてください。Linux、OS  
Windowsであれば、ビルドはmainnetとtestnetともに可能です。

X、

引数なしで `bx` と入力すると、利用可能な全てのコマンドリストが表示されます([[appdx\\_bx](#)]参照)。

Bitcoin Explorerはまた [WindowsのためのVisual Studio](#)プロジェクトだけでなく、LinuxやOS X上でソースコードから構築するインストーラも提供されています。Autotoolsを使って手動でソースコードからビルドすることもできます。これらを使うとlibbitcoinの依存ライブラリも一緒にインストールされます。

**TIP**

Bitcoin Explorerは、アドレスのエンコードやデコード、異なったフォーマットや表現方法への変換のための多くの有用なコマンドを提供しています。Base16(16進数) やBase58、Base58Check、Base64などいろいろなフォーマットをこれらのコマンドを使って調べてみてください。

### libbitcoinインストール

libbitcoinライブラリは [Linux、OS XまたはWindowsでのVisual Studio](#)プロジェクトとしてビルドするためのインストーラを提供しています。また、Autotoolsを使ってソースコードから手動でビルドすることもできます。

**TIP**

Bitcoin Explorerインストーラはbxとlibbitcoinライブラリ両方をインストールします。もしbxをソースコードからビルドした場合はこのステップをスキップすることができます。

## pycoin

Richard Kiss によって元々作成され管理されているPythonライブラリ [pycoin](#) は、Bitcoinのキーやトランザクションの扱いをサポートしているPythonベースライブラリであるだけでなく、規格外のトランザクションを適切に扱うスクリプト言語をもサポートしているライブラリです。

pycoinライブラリはPython 2系(2.7.x)とPython 3系(3.3以降)をサポートしており、kuやtxという便利なコマンドラインツールも付属しています。pycoin 0.42を仮想環境上(venvを利用)でPython 3系を使ってインストールするには、以下に沿ってやってみてください。

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Downloading/unpacking pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded
  Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info for package
pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

以下は、pycoinライブラリを使って情報取得をしたりbitcoinを使ったりするPythonスクリプトサンプルです。

```

#!/usr/bin/env python

from pycoin.key import Key

from pycoin.key.validate import is_address_valid, is_wif_valid
from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address)
print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False),
key.address(use_uncompressed=True)):
    print("** WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())

```

もしkuやtxというコマンドラインユーティリティを使うには、[\[appdxbitcoinimpproposals\]](#)の例を参照してみてください。

## btcd

btcdは Go 言語で書かれているフルノードのBitcoin実装です。現在btcdは、Bitcoinリファレンス実装であるbitcoindのように、厳密なブロックの受け入れルール(バグは含まれていますが)を使ってブロックチェーンのダウンロード、検証、サーブをしています。また、適切に新しく掘り出されたブロックをリレーしたり、トランザクションプールを管理したり、ブロックにまだ組み込まれていない個別のトランザクションをリレーしたりもします。トランザクションプールに入っている全ての個別トランザクションは、先ほどの厳密な受け入れルールに従わなければなりません。また、トランザクションをフィルターするマイナー必要条件("標準"トランザクション)に基づく厳密検証もしています。この厳密検証は広く使われている検証ルールです。

btcdとbitcoindの違いはbtcdがウォレットの機能を持っていないことで、これは計画的に決められたデザイン方針です。これは、あなたがbtcdを通して直接支払いをしたり受け取ったりできないことを意味します。ウォレット機能はbtcwalletとbtcdguiプロジェクトによって提供されており、これらプロジェクトは両方とも活発に開発されています。その他の特筆すべき違いは、btcdがHTTP POSTリクエスト(これはbitcoindもサポートしています)とWebsocketの両方をサポートしていることで、事実btcdのRPC接続はデフォルトでTLS-enableになっています。

### btcdのインストール

btcdをWindowsにインストールするには、[GitHub](#) から msiファイルをダウンロードし実行してください。もしくは、Linux上で以下のコマンドを実行してください。ただし、Go言語がすでにインストールされていることを前提にしています。

```
$ go get github.com/conformal/btcd/...
```

btcdを最新バージョンにアップデートするには、以下を実行するだけです。

```
$ go get -u -v github.com/conformal/btcd/...
```

### btcdのコントロール

btcdは多くの設定オプションを持っています。このオプションは以下を実行することで見ることができます。

```
$ btcd --help
```

btcdは、btcctlのようにすぐに使える形で提供されており、RPC経由でbtcdをコントロールしたりbtcdに問い合わせを出したりするコマンドラインツールが同封されています。ただbtcdはデフォルトでRPCサーバが使える状態にはなっておらず、最低でも以下の設定ファイルにRPCユーザ名とパスワードの両方を設定しなければいけません。

- *btcd.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

- *btctl.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

もしくは、もしコマンドラインで設定ファイルを上書きしたければ以下を実行してください。

```
$ bitcoind -u myuser -P SomeDecentp4ssw0rd
$ btctl -u myuser -P SomeDecentp4ssw0rd
```

実行可能なオプションのリストを表示するには、以下を実行してみてください。

```
$ btctl --help
```

# キー、アドレス、ウォレット イントロダクション

bitcoinの所有権は、デジタルキー、Bitcoinアドレス、デジタル署名で規定されます。デジタルキーはBitcoinネットワークの中に保持されておらず、ユーザによって作成された個々のユーザのファイルの中、またはウォレットという簡単なデータベースの中に保持されています。ユーザのウォレットの中にあるデジタルキーは完全にBitcoinプロトコルとは独立しており、ブロックチェーンへの参照またはインターネットへのアクセスを必要とすることなくユーザのウォレットでデジタルキーの生成および管理ができるようになっています。このキーによってBitcoinの多くの興味深い性質が実現可能になっています。この性質にはde-centralized trustや資産コントロール/所有権の証明、暗号学的証明によるセキュリティモデルが含まれます。

全てのBitcoinトランザクションには有効な署名が必要であり、これによってトランザクションをブロックチェーンに含めることができます。有効な署名は有効なデジタルキーのみによって生成できます。よって、このデジタルキーのコピーを持つ人であればどんな人でも、このデジタルキーに紐づく口座にあるbitcoinをコントロールできます。銀行の口座番号と似た公開鍵と、銀行のPINコードに似た秘密鍵、また小切手への署名について考えてみましょう。これらのデジタルキーをごく稀にユーザが確認することがありますが、ほとんどの場合ウォレットのファイルの内部に格納されウォレットによって管理されるため目にすることはありません。

Bitcoinトランザクションの支払い情報の一部には受取人の公開鍵が表示されており、この公開鍵はBitcoinアドレスと呼ばれるデジタルフィンガープリントの形で表示されています。このBitcoinアドレスは小切手にある受取人名(米國小切手にある"Pay to the order of")と同じようにも使われます。多くの場合、Bitcoinアドレスは公開鍵から生成され1つの公開鍵に対応しています。しかし、全てのBitcoinアドレスが公開鍵を表現しているわけではありません。この後の章でも見る通り、scriptのような形でも他の受取人を表現できます。この方法では、Bitcoinアドレスは資金の受取人を要約しているだけであり、小切手用紙のように取引の相手方を柔軟に変更できるようにしています。これは、支払い手段として、個人の口座への支払い、企業の口座への支払い、請求書に対する支払い、現金としての支払いなどがあるようなものです。Bitcoinアドレスはユーザが繰り返し見るキーの1つの表現です。なぜなら、BitcoinアドレスはBitcoinネットワークの中で共有される必要があるものだからです。

この章では、暗号学的キーを含むウォレットを紹介します。どのようにキーが生成され、格納され、管理されているのかを見ていき、秘密鍵や公開鍵、Bitcoinアドレス、scriptアドレスを表すいろいろなエンコード形式を説明します。最後に、メッセージに署名したり所有権を証明したり文字列指定のあるBitcoinアドレスやペーパーウォレットを作成したりするためのキーの特別な使用方法を見ていきます。

## 公開鍵暗号と暗号通貨

公開鍵暗号は1970年代に発明され、計算機および情報セキュリティ分野の数学的基礎になっていました。

公開鍵暗号が発明されてから、いくつかの適した数学的関数が発見されていきました。この数学的関数は、素数のべき乗や楕円曲線上のスカラー倍算などです。これらの数学的関数は現実な時間で考えると不可逆ものになっています。不可逆とは、一方向への計算は簡単でも逆方向の計算は実行不可能なことを言います。これらの数学的関数に基づく暗号は、デジタルシークレットや偽造不可能なデジタル署名の生成を可能にしています。Bitcoinは公開鍵暗号として楕円曲線上のスカラー倍算を使っています。

Bitcoinでは、bitcoinへのアクセスコントロールをするキーペアを生成するために公開鍵暗号を使っています。このキーペアは秘密鍵とこの秘密鍵から一意に生成される公開鍵で構成されています。この公開鍵はbitcoi

nを受け取るために使われ、秘密鍵はBitcoinトランザクションに署名するのに使われます。

公開鍵と署名生成に使われる秘密鍵の間には数学的な関係があります。この署名は秘密鍵を公開することなく公開鍵だけで検証できるようになっています。

bitcoinを使うとき、bitcoin所有者は公開鍵と署名(生成するごとに異なるものになりますが、同じ秘密鍵から生成されます)をBitcoinトランザクション内に記載します。この公開鍵と署名を通して、Bitcoinネットワークの全ての人はこのトランザクションが有効なものであると検証でき、bitcoinを送った人が送る時点でこのbitcoinを所有しているというを確認できるのです。

**TIP** ほとんどのウォレット実装では、利便性のため秘密鍵と公開鍵がキーペアとして一緒に保存されています。しかし、この公開鍵は秘密鍵から計算できるため、秘密鍵だけを保存しておくことも可能です。

## 秘密鍵と公開鍵

Bitcoinウォレットにはキーペアリストを持っており、それぞれのキーペアは秘密鍵と公開鍵で構成されています。秘密鍵( $k$ )は数値で、通常ランダムに選ばれます。秘密鍵から不可逆関数である楕円曲線上のスカラー倍算を用いて公開鍵( $K$ )を生成します。そして、この公開鍵( $K$ )から不可逆ハッシュ関数を用いてBitcoinアドレス( $A$ )を生成します。この節では、秘密鍵の生成から始めて、公開鍵の生成に使われる楕円曲線の数学を見ていき、最後にBitcoinアドレスを公開鍵から生成します。秘密鍵と公開鍵、Bitcoinアドレスの関係は[公開鍵](#)、[秘密鍵](#)、[Bitcoinアドレス図](#)に書かれています。

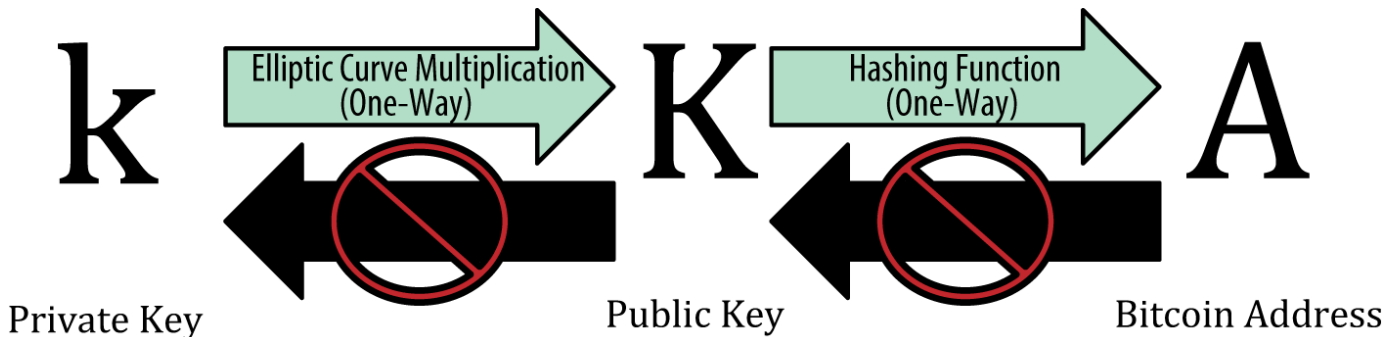


Figure 1. 公開鍵、秘密鍵、Bitcoinアドレス

## 秘密鍵

秘密鍵は意図的に指定できるものではなく無作為に選ばれる数値です。秘密鍵による所有権管理はBitcoinアドレスに結びついた全ての資金の根幹をなします。秘密鍵は署名を生成するときに使われ、この署名は資金を使うときに所有権の主張に必要となります。秘密鍵はいつでも極秘に保っておかなければいけません。もし他者に秘密鍵が漏れると、秘密鍵に対応したBitcoinアドレスの資金のコントロールを他者に与えることになってしまうためです。秘密鍵をバックアップをしておくだけでなく、秘密鍵の偶発的な紛失からも保護しておかなければいけません。というのは、もし秘密鍵をなくしてしまうと、秘密鍵を復元することはできず、秘密鍵によってセキュリティを担保していた資金も永遠に失ってしまうためです。

**TIP** Bitcoin秘密鍵は単なる数値で、コインや鉛筆、紙だけを使ってランダムに秘密鍵を選ぶことができます。例えば、コインを256回投げたBitcoinウォレットで使うランダムな秘密鍵の二進数を作り出すことができます。公開鍵はこの秘密鍵から生成することができます。



## ランダムな数値からの秘密鍵の生成

キーを生成する上で重要かつ一番最初にしなければならないことは、キー生成の安定的なエントロピー源、つまり十分なランダムさを確保することです。Bitcoinキーを作ることは、"1から  $2^{256}$  までの間の数字を選ぶ"ということと本質的に同じです。数字を選ぶ厳格な方法は、予測可能であったり再現可能性があったりしない方法です。Bitcoinソフトウェアは、256bitのエントロピー(ランダムさ)を作り出すOSのランダム値生成器を使っています。通常OSのランダム値生成器は人間を元にしたランダムさを使って初期化されます。数秒間だけマウスを小刻みに動かしてくださいとOSからお願いされたことがあるかもしれませんが、それはこのランダム値生成が理由です。病的なほどに疑り深い人に対しては、サイコロやペンや紙を使うのが一番よいでしょう。

さらに正確に言うと、秘密鍵は  $1$  と  $n - 1$  の間の任意の整数で、 $n$ はBitcoinで使われている楕円曲線の位数として定義されている定数です( $n = 1.158 * 10^{77}$ で $2^{256}$ よりわずかに小さい)(楕円曲線暗号図参照)。このようなキーを作るために、256bitの数値からランダムに数値を選び、選んだ数値が  $n - 1$  より小さいかをチェックしています。プログラミング用語で言うとこれは通常、暗号学的に安定なランダム源から集められた任意の長さを持ったより大きい文字列を取得し、256bitの数値を作る便利なSHA256ハッシュアルゴリズムに放り込むことで達成できます。もし結果が  $n - 1$  よりも小さかった場合、それが適切な秘密鍵です。もしそうでなければ、単にもう一度別のランダムな数値を取得してうまくいくまで同じことを繰り返します。

### TIP

自身でランダムな数値を作り出すコードを書いたり、使っているプログラミング言語が提供している"簡単な"ランダム数値生成器を使ったりしないでください。十分なエントロピー源からのシードを伴った暗号学的に安全な疑似乱数生成器(CSPRNG)を使ってください。選んだ乱数生成器ライブラリのドキュメントを読んで暗号学的に安全かどうかを確認してください。CSPRNGの正しい実装はキーのセキュリティにとって決定的に重要な部分になります。

以下はランダムに生成された秘密鍵( $k$ )で、16進数形式(それぞれ4bitずつ64個の16進数整数で表されている256bit整数)になっています。

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

### TIP

Bitcoinの秘密鍵スペースのサイズは  $2^{256}$  であり、途方もなく大きな数字です。10進数で約  $10^{77}$  になります。この数字の分かりやすい比較として原子の数を考えてみましょう。観測可能な宇宙には  $10^{80}$  個の原子があると見積もられており、秘密鍵が取り得るパターン数はこの原子の数くらいの数になります。

Bitcoin Coreクライアントで新しい鍵を生成するために前に説明した `getnewaddress` コマンドを使います([ch03\_bitcoin\_client]図参照)。セキュリティのため、公開鍵だけを表示しており秘密鍵は表示していません。`bitcoind`に秘密鍵を表示させるには、`dumpprivkey` コマンドを使ってください。`dumpprivkey` コマンドは `Wallet Import Format (WIF)` と呼ばれるBase58エンコード形式で秘密鍵を表示します。この詳細については秘密鍵フォーマット図で説明します。`getnewaddress` コマンドと `dumpprivkey` コマンドを使った秘密鍵の生成例と表示例を以下に示します。

```
$ bitcoind getnewaddress
1J7mdg5rbQyUHENYdx39VWVK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39VWVK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

`dumpprivkey` コマンドはウォレットを開き、さきほどの `getnewaddress` コマンドで生成された秘密鍵を抽出します。このウォレットに秘密鍵と公開鍵の両方が格納されていなければ、`bitcoind`が公開鍵を秘密鍵から知ることはできません。

#### TIP

+`dumpprivkey`+コマンドは公開鍵から秘密鍵を生成しているわけではありません。これは不可能だからです。このコマンドは単にウォレットがすでに知っている秘密鍵を表示しているだけで、この秘密鍵は`getnewaddress`コマンドで生成されたものです。

また、秘密鍵を生成および表示するためにBitcoin Explorerコマンドラインツール([libbitcoin](#) 参照)の `seed` コマンド、`ec-new` コマンド、`ec-to-wif` コマンドを使うこともできます。

```
$ bx seed | bx ec-new | bx ec-to-wif
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
```

## 公開鍵

公開鍵は楕円曲線上のスカラー倍算を使って秘密鍵から計算されるもので、この処理は不可逆な処理になっています。 $(K = k * G)$  但し  $k$  は秘密鍵、 $G$  は ベースポイント と呼ばれる定点、 $K$  は結果として出てくる公開鍵です。逆操作は、離散対数問題 -  $K$  を知っていたときに  $k$  を導出する問題 - として知られ、この難しさは  $k$  の全ての可能な値を総当たりで調べるのと同じくらい時間がかかる問題です。秘密鍵から公開鍵を生成する方法を説明する前に、楕円曲線暗号をもうちょっと細かく見てみましょう。

## 楕円曲線暗号

楕円曲線暗号は、楕円曲線上の点に対する加法とスカラー倍算で表現される離散対数問題をベースに作られた非対称型暗号方式または公開鍵暗号方式です。

[楕円曲線](#)図はBitcoinで使われているものと似ている楕円曲線の例です。

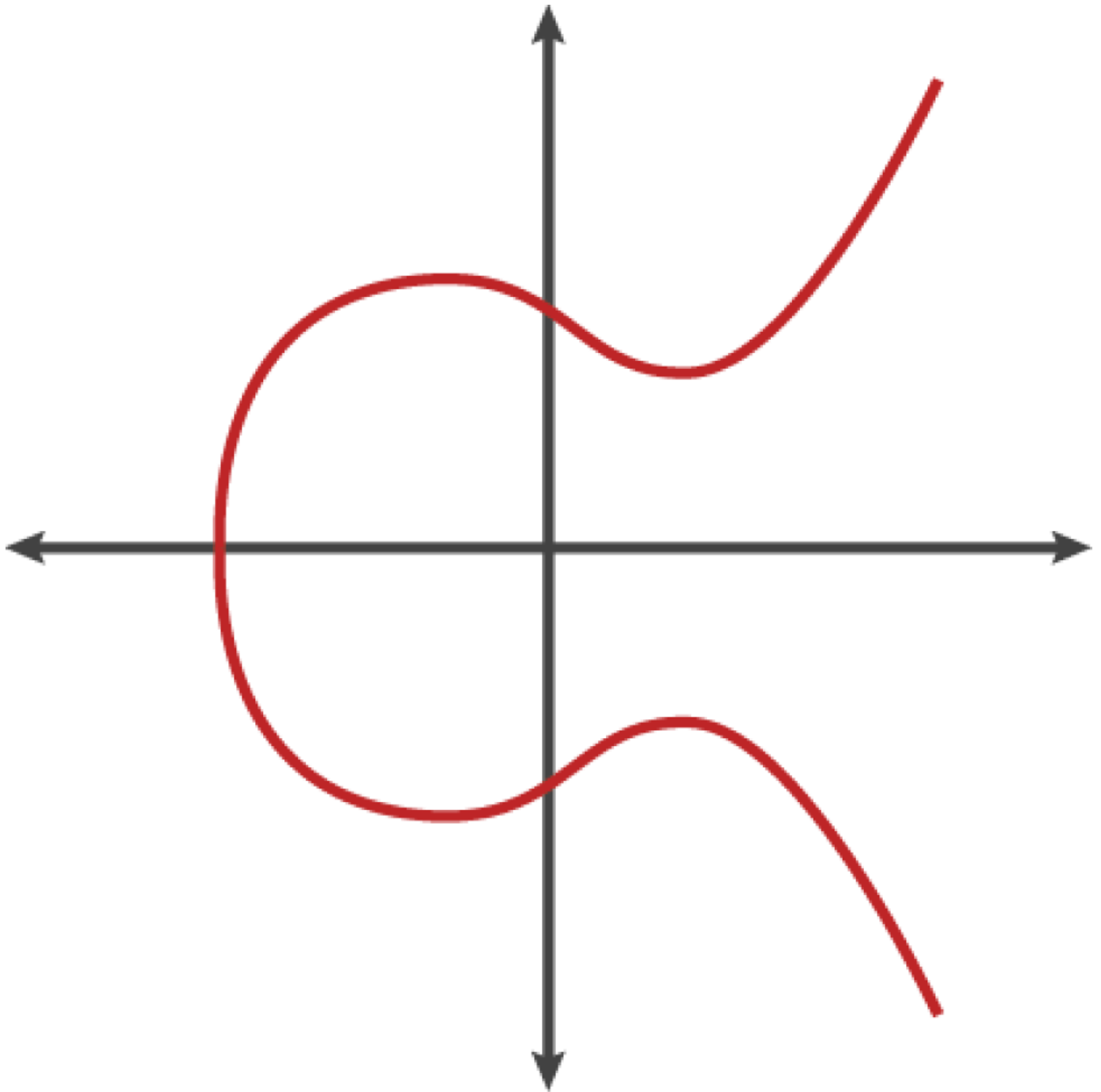


Figure 2. 楕円曲線

Bitcoinは特別な楕円曲線を使っており、National Institute of Standards and Technology (NIST)で標準化された  $\text{secp256k1}$  と呼ばれる集合を使っています。  $\text{secp256k1}$  曲線は次のような関数で定義されており、この関数は楕円曲線になっています。

または

$\text{mod } p$  (素数  $p$  を法とした剰余)とは、この曲線が、位数が素数  $p$  の有限体をなしていることを示しており、 $(\mathbb{F}_p)$  但し  $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$  (とても大きい素数) のようにも書かれます。

この曲線は実数ではなく素数位数の有限体をなしているため、二次元に散りばめられたドットパターンのよう

に見えます。しかし、この数学は実数上に定義された楕円曲線の数学と同等です。例として、楕円曲線暗号:  $p=17$ の有限体 $F(p)$ をなしている楕円曲線を可視化したもの図はより小さい位数17の有限体をなす楕円曲線を示していて、グリッド上のドットパターンになっています。 `secp256k1`

Bitcoin楕円曲線は、底が知れないほど大きなグリッド上にもっと複雑に描かれたドットパターンと考えることができます。

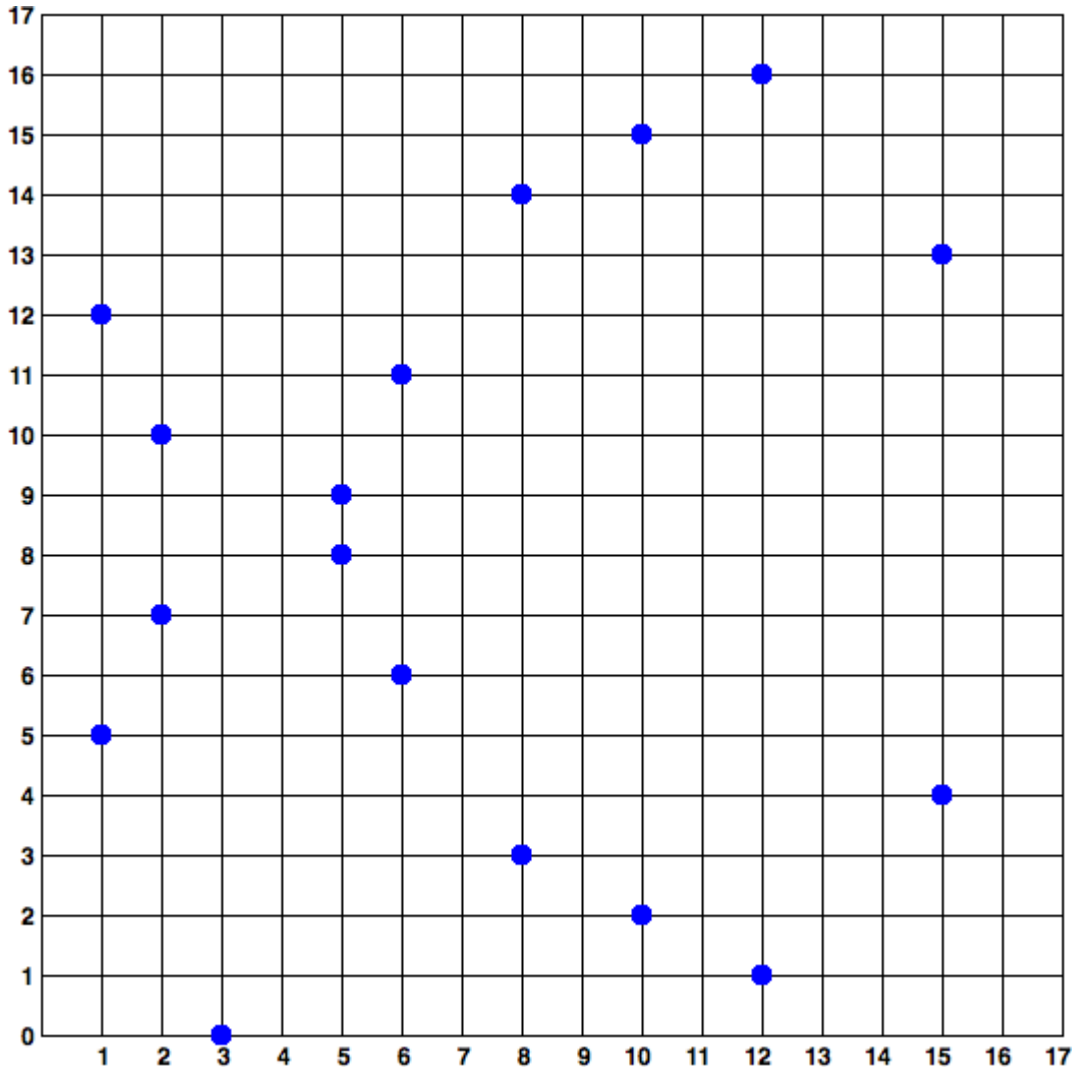


Figure 3. 楕円曲線暗号:  $p=17$ の有限体 $F(p)$ をなしている楕円曲線を可視化したもの

例えば、以下は `secp256k1` 曲線上の点Pです。これが `secp256k1` 曲線上にあることは以下のPythonコードを使って自身で確かめることができます。

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240,
32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

```

Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0

```

楕円曲線では、"無限遠点"と呼ばれる点があります。この点は簡単に言って足し算での0に対応しています。コンピュータ上では、ときどき  $x = y = 0$  と表現されます (これは楕円曲線方程式を満たしていませんが、特別な場合として簡単に確かめることができる解です)。

+  
演算子は加法と呼ばれ、小学校で習う実数に対する加法と似たいくつかの性質を持ちます。楕円曲線上にある2つの点  $P_1$  と  $P_2$  が与えられたとき、3つ目の点  $P_3 = P_1 + P_2$  があり楕円曲線上にあります。

幾何学的には、3つ目の点  $P_3$  は  $P_1$  と  $P_2$  を通る直線を描くことによって計算されます。この直線は必ず楕円曲線と別のところで交差します。この交差した点を  $P_3' = (x, y)$  とすると、 $P_3$  はX軸に対して反転したところにある点  $P_3 = (x, -y)$  になります。

無限点の必要性を示す、2つの特別な場合があります。

もし  $P_1$  と  $P_2$  が同じ点だったとすると、 $P_1$  と  $P_2$  を"通る"直線は  $P_1$  で曲線に接する接線となるはずですが、この接線は曲線と新しい点と必ず交わります。接線の傾きを決めるのに微積分のテクニックを使うことができます。奇妙にも、実数ではなく整数座標で構成される曲線にしたとしても微積分のテクニックはうまくいくのです！

いくつかの場合 ( $P_1$  と  $P_2$  が同じX座標を持ちY座標が違う場合など)、 $P_1$  と  $P_2$  を結ぶ直線は厳密に垂直となり  $P_3$  は無限遠点となります。

もし  $P_1$  が無限遠点である場合、和は  $P_1 + P_2 = P_2$  となります。同様に、もし  $P_2$  が無限遠点である場合、 $P_1 + P_2 = P_1$  となります。これが示していることは、無限遠点が0の役割をしているということです。

+ が  $(A + B) + C = A + (B + C)$  という結合則を満たすことがわかります。これは括弧をなくとも何のあいまいさもなく  $A + B + C$  と書けることを意味します。

加法が定義されたので、加法を拡張する標準的な方法に沿ってスカラー倍算を定義できます。楕円曲線上の点  $P$  に対して、もし  $k$  が整数だとすると、 $kP = P + P + P + \dots + P$  ( $k$ 回)。紛らわしいことにこの  $k$  がときどき "べき指数" と呼ばれるのです。

## 公開鍵の生成

秘密鍵をランダムに生成された数値  $k$  とすると、あらかじめ決められた生成元  $G$  を  $k$

に掛けることで楕円曲線上のもう1つの点を得ます。このもう1つの点は公開鍵  $K$  に対応するものです。ベースポイントは  $\text{secp256k1}$  標準で決められており、Bitcoinでの全ての公開鍵に対して常に同じです。

但し、 $k$ は秘密鍵、 $G$ はベースポイント、 $K$ は結果として出てくる公開鍵で楕円曲線上にある点です。ベースポイントはどのBitcoinユーザでも同じであるため、秘密鍵 $k$ が同じであれば公開鍵は同じになります。 $k$ と $K$ の関係は固定されていますが、 $k$ から $K$ という一方向でのみ計算ができます。これは、( $K$ から得られる)Bitcoinアドレスがどの人にも共有され得るからで、Bitcoinアドレスから秘密鍵 $k$ を知ることはできません。

**TIP**

秘密鍵は公開鍵に変換できますが、公開鍵は秘密鍵に戻すことはできません。これは秘密鍵から公開鍵への変換プロセスが一方向だけにしか使えないからです。

楕円曲線上のスカラー倍算を実装すると、前に生成した秘密鍵 $k$ をベースポイント $G$ に掛けることで公開鍵 $K$ が得られます。

$$K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G$$

公開鍵 $K$ は点  $K = (x,y)$  として定義されます:

$$K = (x, y)$$

但し

$$\begin{aligned} x &= F028892BAD7ED57D2FB57BF33081D5CF6F9ED3D3D7F159C2E2FFF579DC341A \\ y &= 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB \end{aligned}$$

点への整数のスカラー倍算を可視化するために、数理的な構造が同じ実数に対する簡単な楕円曲線を使います。ゴールは、ベースポイント $G$ から決められる $kG$ を見つけることです。これは、 $G$ 自身を $k$ 回足すことと同じです。楕円曲線で自分自身に足すとは、点に接する接線を描きその接線がもう一度楕円曲線に交わる点、 $X$ 軸に対して対称な点、を見つけることです。

楕円曲線暗号: 楕円曲線上での整数  $k$  によるベースポイント  $G$  へのスカラー倍算を可視化したもの図は、楕円曲線上で幾何学的な操作を繰り返すことで $G, 2G, 4G$ を導きだすプロセスを表しています。

**TIP**

ほとんどのBitcoin実装では、[OpenSSL](#) 暗号学的ライブラリを使って楕円曲線での数学の計算をしています。例えば、公開鍵を導出するために関数 `EC_POINT_mul()` が使われています。

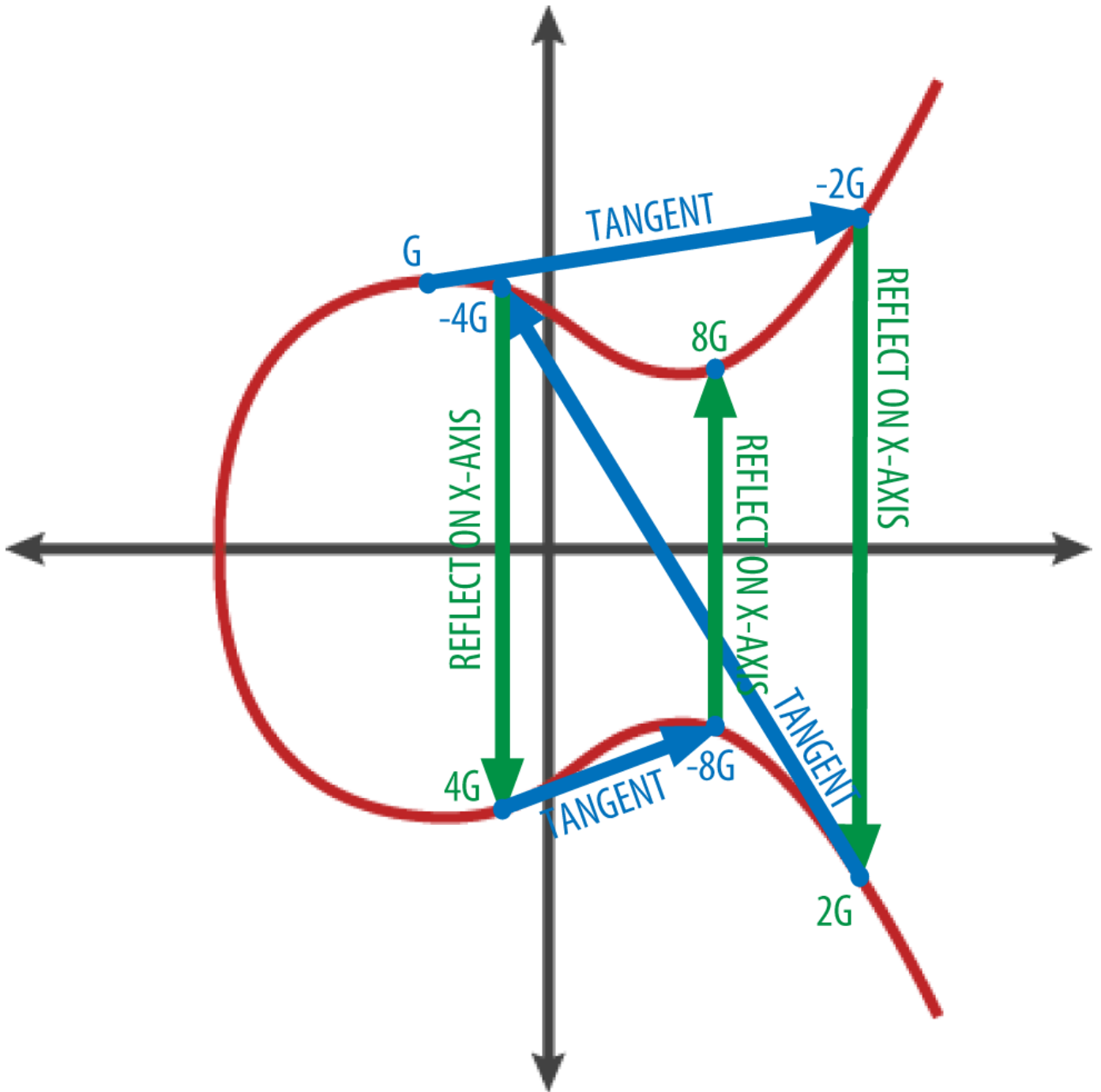


Figure 4. 楕円曲線暗号: 楕円曲線上での整数  $k$  によるベースポイント  $G$  へのスカラー倍算を可視化したもの

## Bitcoin アドレス

Bitcoinアドレスは、あなたにお金を送りたい人誰にでも共有されえる、数値と文字で構成された文字列です。公開鍵から作られるBitcoinアドレスは数値と文字による文字列になっており、1からはじまるものです。以下はBitcoinアドレスの例です。

1J7mdg5rbQyUHENYdx39VWVK7fsLpEoXZy

Bitcoinアドレスは、資金の受取人としてトランザクションに共通に現れるものです。小切手(英語)とBitcoinトランザクションを比べてみましょう。Bitcoinアドレスは米国小切手の"Pay to the order of"のあとに書かれる受益者名です。小切手上では、この受益者は銀行口座を持っている人の名前にもなりますが、企業名や機関名、現金になることもあります。小切手では口座を特定する必要はないので、むしろ資金の受取人として口座を指す抽象名を使います。この抽象名を使うことで小切手での支払いがしやすくなります。BitcoinトランザクションではBitcoinアドレスという小切手に似た抽象名を使い、Bitcoinトランザクションを使いやすいものになっています。Bitcoinアドレスは秘密鍵/公開鍵の所有者、または[p2sh]にある支払いscriptのような他の何かを表すものです。とりあえずBitcoinアドレスが公開鍵から生成される簡単な場合から説明していきましょう。

Bitcoinアドレスは一方暗号的ハッシュ化を使うことで公開鍵から生成されます。"ハッシュ化アルゴリズム"または簡単に"ハッシュアルゴリズム"は、フィンガープリントまたは任意のサイズの"ハッシュ"を作り出す一方関数です。暗号的ハッシュ関数はBitcoinの中で広範囲に活用されます。具体的にはBitcoinアドレス、scriptアドレス、proof-of-workマイニングアルゴリズムの中で使われます。公開鍵からBitcoinアドレスを作るときに使うアルゴリズムは、Secure Hash Algorithm (SHA) と RACE Integrity Primitives Evaluation Message Digest (RIPEMD) で、中でも SHA256 と RIPEMD160 が使われます。

公開鍵KのSHA256ハッシュを計算し、さらにこの結果のRIPEMD160ハッシュを計算することで、160bit(20byte)の数字を作り出します。

但し、Kは公開鍵、Aは結果として出てきたBitcoinアドレスです。

**TIP**

Bitcoinアドレスは公開鍵と同じではありません。  
Bitcoinアドレスは公開鍵から一方関数を使って導出されるものです。

Bitcoinアドレスは、"Base58Check" と呼ばれる形にエンコードされた状態で通常使われます(Base58とBase58Checkエンコード参照)。  
"Base58Check"では、58個の文字(Base58)とチェックサムを使いますが、これは人間にとって読みやすくしたり、曖昧さを避けたり、転写時のエラーを防いだりするためです。Base58Checkはまた、Bitcoinで他の用途でも使われます。Bitcoinアドレス、秘密鍵、暗号化された鍵、scriptハッシュなど、ユーザが読む必要があったり、数字を正しく転写しなければいけなかったりするときにはいつでもです。次の節では、Base58Checkのエンコード、デコードの仕組みと結果として出てくる表現について説明します。[公開鍵からBitcoinアドレスへ](#)、[公開鍵をBitcoinアドレスに変換するプロセス図](#)は公開鍵からBitcoinアドレスへの変換を説明しています。



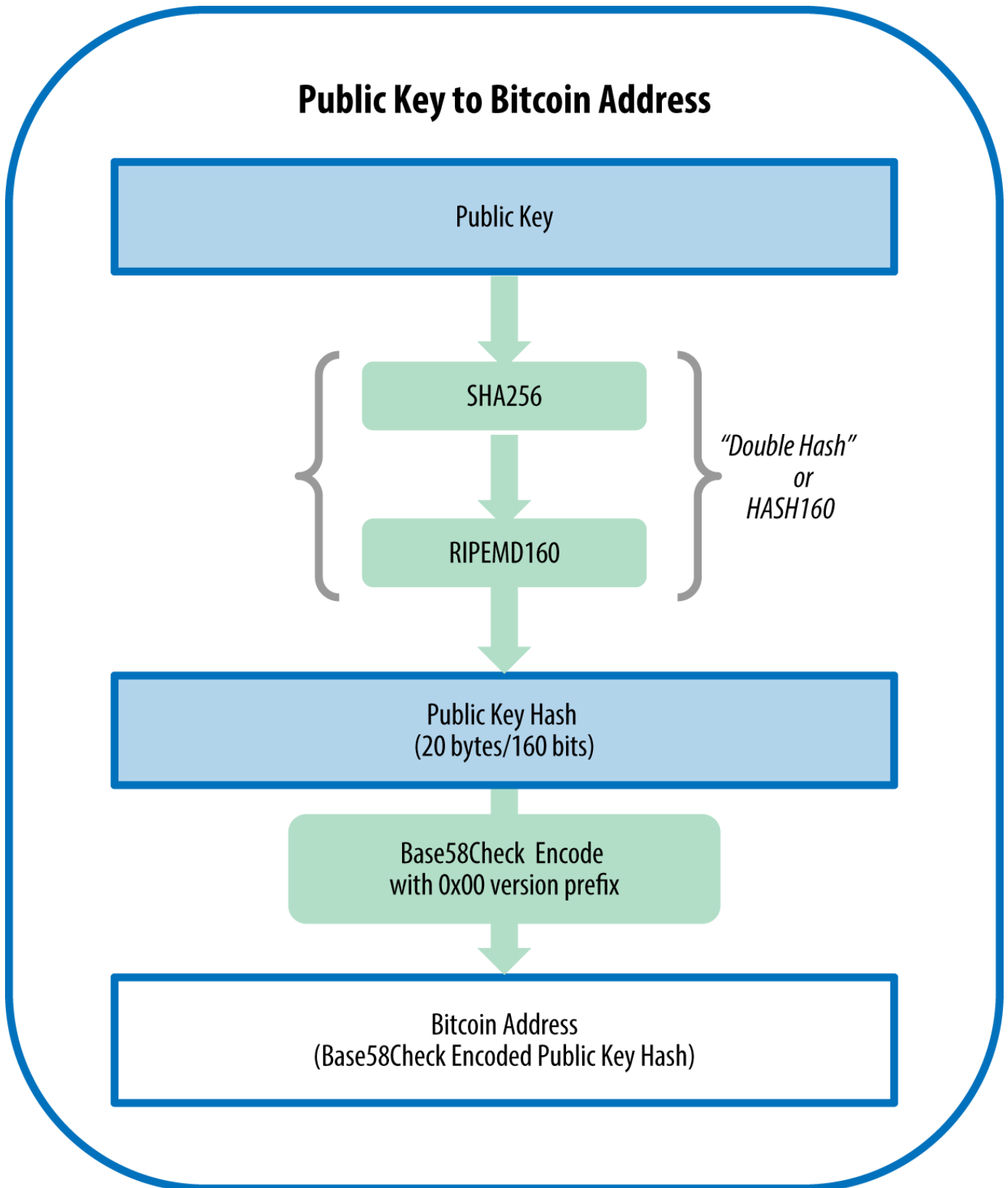


Figure 5. 公開鍵からBitcoinアドレスへ: 公開鍵をBitcoinアドレスに変換するプロセス

### Base58とBase58Checkエンコード

大きな数字をコンパクトに表すために、多くのコンピュータではいくつかの記号を使うことで10以上を基数とするアルファベットと数字を混ぜた表現を使っています。例えば、伝統的な10進数では0から9までの10個

の数字を使う一方、16進数ではAからFの文字を使うことで16個の数字を使います。16進数で表される数字は10進数で表すよりも短くなります。

バイナリデータをemailのようなテキストベースの通信で送るために、Base-64 では26個の小文字、26個の大文字、10個の数字、"/" や " " のような2種類の文字を使います。Base-64 はemailにバイナリデータを添付するのによく使われます。Base58 はBitcoinで使うために開発されたテキストベースのエンコード形式で、他の暗号通貨でも使われています。これはコンパクトな表現、可読性、エラー発見および防止のためです。Base58はBase64の部分集合でアルファベットの大文字小文字、数字が使われます。しかし、あるフォントで表示したときに同じように見えて、よく間違えてしまういくつかの文字は省かれています。Base58はBase64から0(数字の0)、O(大文字o)、l(小文字L)、I(大文字i)、記号"/"や" "を除いたもの、つまり、(0, O, l, I)を除いたアルファベットの大文字小文字、数字の集合になっています。

Example 1. BitcoinにおけるBase58のアルファベット

```
123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

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

数値データをBase58Check形式に変換するために、まずデータの先頭に"version byte"と呼ばれている文字を追加します。このversion byteは、簡単にエンコードされたデータの種類を特定できるように付加されています。例えば、Bitcoinアドレスの場合先頭はゼロ(16進数で0x00)で、一方秘密鍵をエンコードするときには先頭は128(16進数で0x80)です。一般的に使われているversion byteのリストは[Base58Checkのversion prefixとエンコードされた結果の例](#)を参照してください。

次に、"double-SHA"チェックサムを計算してみましょう。これは、SHA256ハッシュアルゴリズムを前の結果(プレフィックスとデータ)に2回適用するという意味です。

```
checksum = SHA256(SHA256(prefix+data))
```

結果として出てくる32byteハッシュ(ハッシュのハッシュ)から最初の4byteだけを取り出します。これら4byteをエラーチェックコードとして、またはチェックサムとして使用されます。このチェックサムは最後に付加されます。

結果は3つの部分、プレフィックス、データ、チェックサムによって構成されていて、前に書いたBase58のアルファベットを使ってエンコードされています。[Base58Checkエンコーディング](#): [曖昧さなく](#)

Bitcoinデータをエンコードするために、version byte、チェックサム付加しBase58変換をしたフォーマット図はBase58Checkエンコードのプロセスを説明しています。

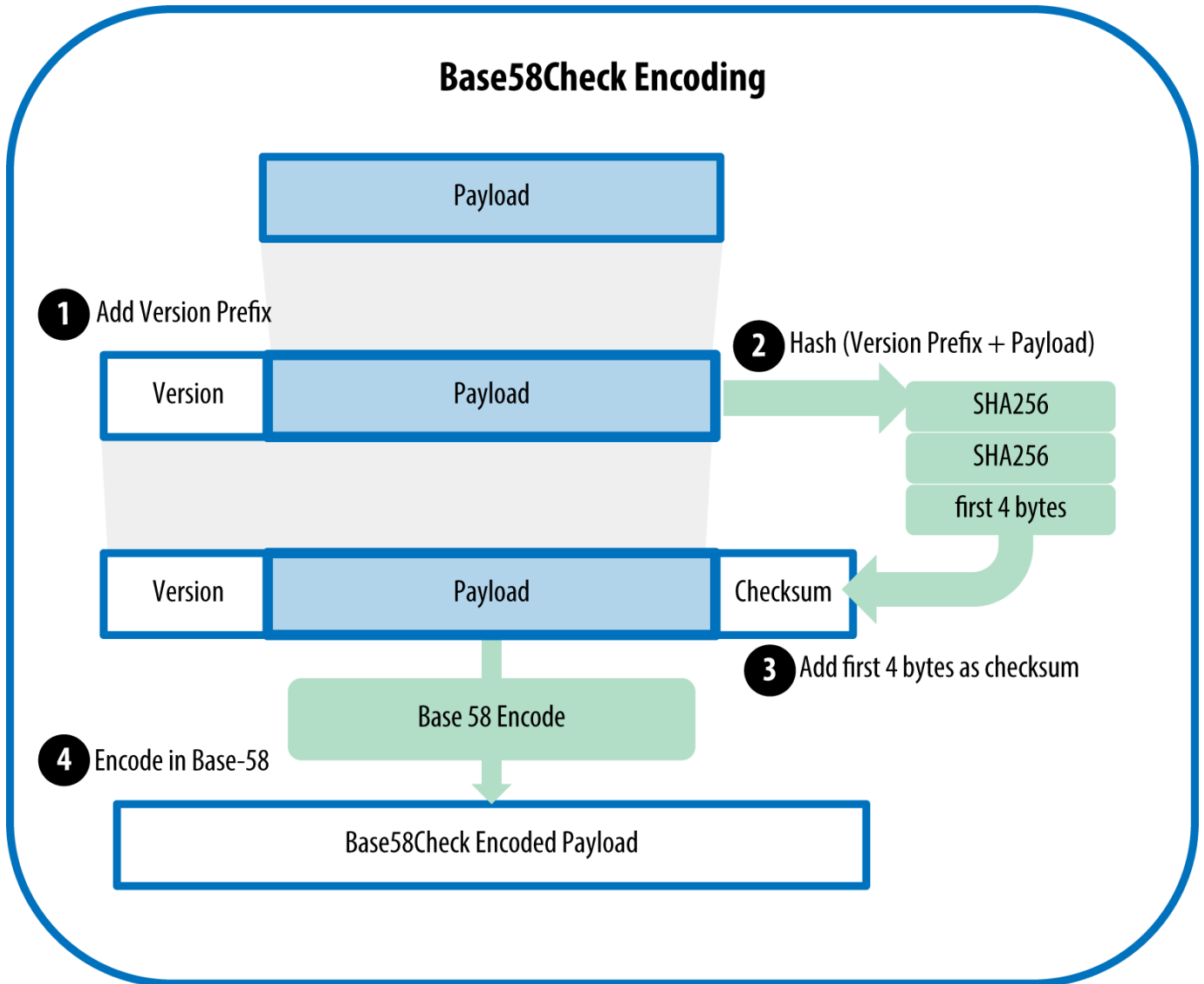


Figure 6. Base58Checkエンコーディング: 曖昧さなくBitcoinデータをエンコードするために、version byte、チェックサム付加しBase58変換をしたフォーマット

Bitcoinでは、データをコンパクトにするためや読みやすくするため、エラーを検知しやすくするために、ユーザに渡されるほとんどのデータをBase58Checkエンコードにしています。Base58Checkでのversion prefixは、簡単に形式を区別するために使われており、Base58でエンコードされるときにBase58Checkエンコードpayloadの最初に特定の文字を付けられています。これらの文字は、どんな種類のデータをエンコードしたのか、データをどう使うのかを人間が分かるようにしています。1から始まるものはBase58CheckエンコードされたBitcoinアドレス、5から始まるものはBase58Checkエンコードされた秘密鍵WIF形式です。Base58Checkのversion prefixとエンコードされた結果の例に、いくつかのversion prefixとBase58エンコードされた文字の例を示します。

Table 1. Base58Checkのversion prefixとエンコードされた結果の例

種類	Version prefix (16進数)	Base58出力文字列の先頭に付けられるプレフィックス
Bitcoinアドレス	0x00	1
Pay-to-Script-Hashアドレス	0x05	3
Bitcoin Testnet アドレス	0x6F	m or n
秘密鍵 WIF形式	0x80	5, K or L
BIP38 暗号化秘密鍵	0x0142	6P
BIP32 拡張公開鍵	0x0488B21E	xpub

Bitcoinアドレスを生成する完全な手順を見てみましょう。秘密鍵から始めて、公開鍵(楕円曲線上の点)を作り、二重ハッシュ化アドレスを作り、最後にBase58Checkエンコードします。[秘密鍵からのBase58CheckエンコードされたBitcoinアドレスの作成](#)にあるC++コードは、秘密鍵からBase58Checkエンコード済みBitcoinアドレスまでの手順を完全な形で逐一示しています。このサンプルコードは、いくつかの補助関数を使うために[\[alt\\_libraries\]](#)で紹介したlibbitcoinライブラリを使っています。

## Example 2. 秘密鍵からのBase58CheckエンコードされたBitcoinアドレスの作成

```
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Private secret key.
    bc::ec_secret secret;
    bool success = bc::decode_base16(secret,
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
    assert(success);
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //   bc::payment_address payaddr;
    //   bc::set_public_key(payaddr, public_key);
    //   const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);

    bc::data_chunk unencoded_address;
    // Reserve 25 bytes
    // [ version:1 ]
    // [ hash:20   ]
    // [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).
    unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash.
    bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding
    assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);

    std::cout << "Address: " << address << std::endl;
    return 0;
}
```

このコードは事前に決められた秘密鍵を使っており、動作させるたびに毎回同じBitcoinアドレスが生成され

るようになっています。具体的な動かし方は[このaddrコードのコンパイルと実行](#)に示されている通りです。

*Example 3.* このaddrコードのコンパイルと実行

```
# Compile the addr.cpp code
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# Run the addr executable
$ ./addr
Public key: 0202a406624211f2abbd6c68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6EhcdU1fpEdX7913CK
```

## キーフォーマット

秘密鍵も公開鍵も多くの違った形式で表現されています。見た目が違っていても、これらの表現は全て同じ数値にエンコードされます。これらの形式は基本的に人々が読みやすくなる、間違うことなく転写できるようにするために使われます。

### 秘密鍵フォーマット

秘密鍵は多くの違った形式で表現されていて、これらは全て同じ256bitの数値に対応しています。[秘密鍵の表現一覧\(エンコーディングフォーマット\)](#)に秘密鍵を表現するために使われる3つの形式を示します。

*Table 2.* 秘密鍵の表現一覧(エンコーディングフォーマット)

種類	プレフィックス	説明
16進数	なし	64個の16進数
WIF	5	Base58Checkエンコーディング: 128のversion prefixと、 32bitチェックサムを伴ったBase58
圧縮WIF	K または L	上にあるように、エンコード前に サフィックス 0x01 が追加されています

**例:** [同じキーの違ったフォーマット](#)は、これら3つの形式で生成された秘密鍵を示しています。

*Table 3.* 例: 同じキーの違ったフォーマット

フォーマット	秘密鍵
16進数	1e99423a4ed27608a15a2616a2b0e9e52ced330ac53 0edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpbnk eyhfsYB1Jcn
WIF圧縮形式	KxFC1jmwWCoACiCAWZ3eXa96mBM6tb3TYzGmf 6YwgdGWZgawvrtJ

これら全ての表現は、同じ数値、同じ秘密鍵を表す違った形式になっています。ぱっと見は違っていますが、どれも他の形式に簡単に変換できます。

Bitcoin Explorer([libbitcoin](#)参照)の `wif-to-ec` コマンドを使うことで、さきほどの両方の WIFキーが同じ秘密鍵を表すことを示すことができます。

```
$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

$ bx wif-to-ec KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
```

## Base58Checkからのデコード

Bitcoin Explorerコマンド([libbitcoin](#)参照)は、Bitcoinキーやアドレス、トランザクションを操作するシェルスクリプトやコマンドラインの "パイプライン" を簡単に書けるようにするツールです。Bitcoin Explorerを使うとコマンドラインで Base58Checkをデコードできます。

`base58check-decode` コマンドを使うことで圧縮されていないキーをデコードできます。

```
$ bx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
wrapper
{
  checksum 4286807748
  payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
  version 128
}
```

出力された結果には、`payload`としてのキーと `Wallet Import Format (WIF)`の `version prefix 128`、チェックサムが含まれています。

圧縮されたキーの "payload"にはサフィックスが追加されており、出力された公開鍵が圧縮されたものであることを表しています。

01

```
$ bx base58check-decode KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
wrapper
{
  checksum 2339607926
  payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01
  version 128
}
```

## 16進数からBase58Checkエンコード

Base58Checkにエンコード(前のコマンドの逆)するために、Bitcoin Explorer の base58check-encode コマンド([libbitcoin](#)参照)を使います。このとき、16進数秘密鍵 とともに、WIF (Wallet Import Format) を示すversion prefix 128を入力します。

```
bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
--version 128
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
```

## 16進数(圧縮されたキー)からBase58Checkへのエンコード

"圧縮された"秘密鍵としてBase58Checkにエンコードする([圧縮された秘密鍵](#)参照)ためには、16進数鍵の末尾に 01 を追加し上記のようにエンコードします。

```
$ bx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 --version 128
KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

生成されたWIF圧縮形式は"K"から始まります。これは、元の秘密鍵の最後に"01"が付いていることを意味し、圧縮された公開鍵のみを生成するために使われます([圧縮された公開鍵](#)参照)。

## 公開鍵フォーマット

公開鍵もまた違った形で表現され、圧縮された 公開鍵または 圧縮されていない 公開鍵があります。

前に見たように、公開鍵は楕円曲線上の点であり、(x,y)

というペアの形で構成されます。これは普通プレフィックスに 04 が伴って表されます。この 04 のあとに256bitの2つの数字、1つは x 座標、もう1つは y 座標、が続きます。プレフィックス 04 は圧縮されていない公開鍵を圧縮された公開鍵と区別するために使われます。圧縮された公開鍵は 02 、03 から始まります。

ここで、さきほど作った秘密鍵から公開鍵を生成してみましょう。以下に x 座標と y 座標を示します。

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

以下は、520bitの数値(130桁の16進数整数)として表したさきほどと同じ公開鍵です。これは 04 のプレフィックスが付いており、そのあとに 04 x y のように x 座標と y 座標が続きます。

```
K =
04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A07CF33DA18BD734C600B96A
72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```



## 圧縮された公開鍵

<?dbhtml

orphans="4">

圧縮された公開鍵は、トランザクションのサイズの削減やブロックチェーンを保持しているBitcoinノード上のディスクスペースの保護のためにBitcoinに導入されました。ほとんどのトランザクションは、所有者の証明書を検証したりbitcoinを使うために公開鍵を含んでいるのです。それぞれの圧縮されていない公開鍵は520 bit(プレフィックス  $0x$   $0y$ )を必要とするため、ブロックごとに数百トランザクション、1日に数万トランザクションというトランザクションを重ねると巨大なデータがブロックチェーンに追加されていくことになってしまいます。

公開鍵で見えてきたように、公開鍵は楕円曲線上の点  $(x,y)$  です。楕円曲線は数学的な関数として表現されるため、楕円曲線上の点は公開鍵にある方程式の解であり、もし  $x$  座標が分かるとすると  $y$  座標は  $y^2 \bmod p = (x^3 + 7) \bmod p$  を解くことで計算できます。このため、公開鍵の点として単に  $x$  座標だけを保持すればよく、 $y$  座標を省略して鍵のサイズと256bitを保持するのに必要なスペースを削減することができます。これによりトランザクションのデータサイズにして50%弱の削減ができます。

圧縮されていない公開鍵が  $04$  から始まるのに対して、圧縮されている公開鍵は  $02$  または  $03$  から始まります。なぜ2つのプレフィックスがあるのかというと、方程式の左側には  $y^2$  があるので、 $x$ 座標から  $y$ 座標を導こうとすると  $y$ の解は正負それぞれの符号を持った平方根になってしまい1つに決めることができません。イメージ的には、 $y$  座標が  $x$ 軸の上側と下側の両方の値を取り得ることを意味します。

楕円曲線図にある楕円曲線から分かるように、楕円曲線は  $x$ 軸に対して鏡のように折り返す操作に対して対称です。このため、 $y$  座標の絶対値は省略できても  $y$ 座標の符号は省略できず保存しておく必要があります。別のいい方をすると、これら符号はそれぞれ違った点、違った公開鍵を表すので、点が  $x$ 軸の上にあったか下にあったかを覚えておかなければいけません。素数位数  $p$  の有限体上で二進数演算を用いて楕円曲線を計算すると、 $y$  座標は偶数または奇数になり、これらはさきほど説明した正/負に対応しています。よって、 $y$ の2つの可能な値を区別するには、 $y$  が偶数なら  $02$  を圧縮された公開鍵の先頭に付与、 $y$  が奇数なら  $03$  を先頭に付与するようにします。これによって、ソフトウェアは  $y$  座標を  $x$  座標から正しく導くことができ、公開鍵を圧縮することができます。公開鍵の圧縮は公開鍵の圧縮図で説明されています。

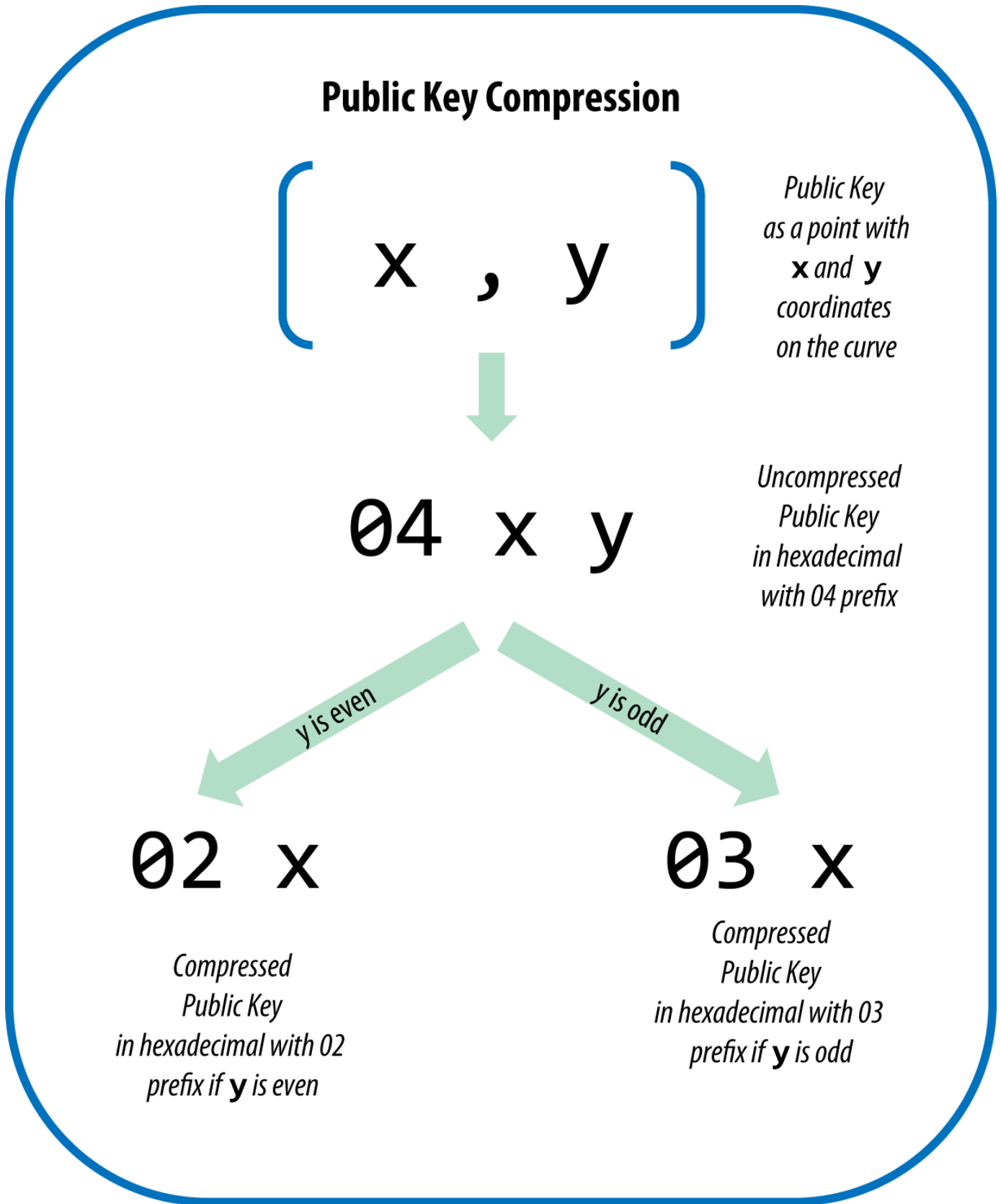


Figure 7. 公開鍵の圧縮

前に生成した公開鍵と同じものを以下に示します。以下は、 $y$  座標が奇数であることを示している +03+ をプレフィックスとして持つ 264bit (66桁の16進数) の圧縮された公開鍵です。

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

圧縮されていない公開鍵と違うように見えますが、この圧縮された公開鍵は同じ秘密鍵から生成されたものです。重要なこととして、もし圧縮された公開鍵を二重ハッシュ化関数 (RIPEMD160(SHA256(K))) を使ってBitcoinアドレスに変換したとすると、違った Bitcoinアドレスが生成されるでしょう。これは1つの秘密鍵が2つの形式(圧縮と非圧縮)の公開鍵を生成し、それぞれの公開鍵を用いて別々のBitcoinアドレスを生成してしまうためですが、これは混乱させる元になってしまいます。

圧縮された公開鍵は次第にBitcoinクライアントでデフォルトになりつつあります。圧縮された公開鍵に対応することで、トランザクションのサイズおよびブロックチェーンのサイズを削減することに十分なインパクトがあるのです。しかし、全てのクライアントが圧縮された公開鍵に対応している訳ではありません。圧縮された公開鍵に対応している最近のクライアントは圧縮された公開鍵に対応していない古いクライアントから来たトランザクションを解釈しなければいけません。これはウォレットがもう1つのウォレットから秘密鍵をインポートするときに特に重要です。というのは、新しいウォレットがインポートされた秘密鍵に対応したトランザクションを見つけるためにブロックチェーンをスキャンしなければいけないためです。ウォレットはどのBitcoinアドレスをスキャンするべきでしょうか？圧縮されていない公開鍵から生成されたBitcoinアドレスでしょうか？それとも、圧縮された公開鍵から生成されたBitcoinアドレスでしょうか？両方とも有効なアドレスですが、これらは異なったBitcoinアドレスです！

この問題を解決するために、秘密鍵をウォレットからエクスポートする場合、今までと異なり最近のウォレットではWallet Import Formatで出力されます。Wallet Import Formatは、圧縮された公開鍵が秘密鍵から作られたということを示しており、Bitcoinアドレスは圧縮されたものということが分かるのです。これによって、インポートされる側のウォレットは古いウォレットから来た秘密鍵か新しいウォレットから来た秘密鍵かを区別でき、公開鍵が圧縮されているかいないかに対応したBitcoinアドレスが含まれているトランザクションをブロックチェーンから探し出すことができます。次の節で、どのようにこれが動いているのか詳細を見てみましょう。

## 圧縮された秘密鍵

皮肉にも、"圧縮された秘密鍵"という言葉は誤解を与えてしまいます。というのは、秘密鍵がWIF圧縮形式でエクスポートされた場合、実際には"圧縮されていない"秘密鍵より1byte 長い からです。これは、01を秘密鍵の最後に付加しているためで、01は秘密鍵が新しいウォレットから来たこと、秘密鍵から圧縮された公開鍵を生成すべきであることを意味しています。秘密鍵自体は圧縮されておらず、また圧縮することはできません。"圧縮された秘密鍵"という言葉は、本当は"圧縮された公開鍵を生成するために使うべき秘密鍵"という意味です。WIF圧縮形式やWIF形式をエクスポートをするときの形式という意味でのみ使うべきで、混乱を避けるために、"圧縮された"という言葉は秘密鍵に対して使うべきではないのです。

WIF圧縮形式とWIF形式は片方しか使えないということを覚えておいてください。圧縮された公開鍵を実装した新しいウォレットでは、秘密鍵はWIF圧縮形式(先頭がKまたはL)としてのみエクスポートされます。もしウォレットが古い実装のもので圧縮された公開鍵が使えないのであれば、秘密鍵はWIF形式(先頭が5)としてのみエクスポートされます。ここでのゴールは、圧縮された公開鍵またはBitcoinアドレスでブロックチェーンを探索しなければいけないか、圧縮されていないもので探索しなければいけないかをウォレットに教えることです。

もしウォレットが圧縮された公開鍵を実装していれば、全てのトランザクションで圧縮された方の方法を使うでしょう。秘密鍵は楕円曲線上の公開鍵点を導出し、この公開鍵は圧縮されます。圧縮された公開鍵はBitcoinアドレスを生成することに使われ、Bitcoinアドレスはトランザクションの中で使われます。圧縮された公開鍵を実装した新しいウォレットから秘密鍵をエクスポートするとき、Wallet Import

Formatは秘密鍵の最後に1byteの `01` が付加される形に修正されます。結果として出力される Base58Checkエンコード秘密鍵は"圧縮されたWIF形式"と呼ばれ、古いウォレットでのWIF形式(非圧縮)の場合の"5"の代わりにKまたはLから始まります。

例: 同じキーの違ったフォーマットは同じキーを表しており、WIF形式とWIF圧縮形式でエンコードされています。

Table 4. 例: 同じキーの違ったフォーマット

| フォーマット   | 秘密鍵  |
|----------|--|
| 16進数     | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD   |
| WIF      | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpbnk<br>eyhfsYB1Jcn            |
| 16進数圧縮形式 | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01 |
| WIF圧縮形式  | KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf<br>6YwgdGWZgawvrtJ            |

**TIP**

"圧縮された秘密鍵"は誤った名称です！これらは圧縮されていません。むしろ、WIF圧縮形式は圧縮された公開鍵やこれに対応したBitcoinアドレスを導出するためのみ使われるべきということを表しています。皮肉にも、"WIF圧縮形式"にエンコードされた秘密鍵は1byteだけ長いのです。というのは、"圧縮されていない"秘密鍵と区別するためにサフィックス `01` が追加されているためです。

## PythonでのキーとBitcoinアドレスの実装

Pythonで書かれた最も総合的なBitcoinライブラリは Vitalik Buterin によって書かれた [pybitcointools](#) です。pybitcointoolsライブラリを使った、キーとアドレスの生成と各フォーマット生成の中で、pybitcointoolsライブラリ("bitcoin"としてimportされています)を使ってキーとBitcoinアドレスをいろいろな形式で生成しています。

Example 4. pybitcointoolsライブラリを使った、キーとアドレスの生成と各フォーマット生成

```
import bitcoin

# Generate a random private key
valid_private_key = False
while not valid_private_key:
    private_key = bitcoin.random_key()
    decoded_private_key = bitcoin.decode_privkey(private_key, 'hex')
    valid_private_key = 0 < decoded_private_key < bitcoin.N

print "Private Key (hex) is: ", private_key
```

```

print "Private Key (decimal) is: ", decoded_private_key

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
print "Private Key (WIF) is: ", wif_encoded_private_key

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print "Private Key Compressed (hex) is: ", compressed_private_key

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print "Private Key (WIF-Compressed) is: ", wif_compressed_private_key

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
print "Public Key (x,y) coordinates is:", public_key

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key, 'hex')
print "Public Key (hex) is:", hex_encoded_public_key

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key
if (public_key_y % 2) == 0:
    compressed_prefix = '02'
else:
    compressed_prefix = '03'
hex_compressed_public_key = compressed_prefix + bitcoin.encode(public_key_x, 16)
print "Compressed Public Key (hex) is:", hex_compressed_public_key

# Generate bitcoin address from public key
print "Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key)

# Generate compressed bitcoin address from compressed public key
print "Compressed Bitcoin Address (b58check) is:", \
    bitcoin.pubkey_to_address(hex_compressed_public_key)

```

[key-to-address-ecc-example.py](#)の実行はこのコードを実行した結果を示しています。

*Example 5. key-to-address-ecc-example.py*の実行

[Bitcoinのキー生成に使われる楕円関数数学のデモスクリプト](#)  
 1つのコード例です。このコードでは、楕円曲線での計算にPython

はもう

ECDSAライブラリを使い、いかなる特別なBitcoinライブラリも使っていません。

*Example 6. Bitcoinのキー生成に使われる楕円関数数学のデモスクリプト*

```
import ecdsa
import os
from ecdsa.util import string_to_number, number_to_string

# secp256k1, http://www.oid-info.com/get/1.3.132.0.10
_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2FL
_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L
_b = 0x000000000000000000000000000000000000000000000000000000000000007L
_a = 0x00000000000000000000000000000000000000000000000000000000000000L
_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L
_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L
curve_secp256k1 = ecdsa.ellipticcurve.CurveFp(_p, _a, _b)
generator_secp256k1 = ecdsa.ellipticcurve.Point(curve_secp256k1, _Gx, _Gy, _r)
oid_secp256k1 = (1, 3, 132, 0, 10)
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve_secp256k1, generator_secp256k1,
oid_secp256k1)
ec_order = _r

curve = curve_secp256k1
generator = generator_secp256k1

def random_secret():
    convert_to_int = lambda array: int("".join(array).encode("hex"), 16)

    # Collect 256 bits of random data from the OS's cryptographically secure random
    generator
    byte_array = os.urandom(32)

    return convert_to_int(byte_array)

def get_point_pubkey(point):
    if point.y() & 1:
        key = '03' + '%064x' % point.x()
    else:
        key = '02' + '%064x' % point.x()
    return key.decode('hex')

def get_point_pubkey_uncompressed(point):
    key = '04' + \
        '%064x' % point.x() + \
        '%064x' % point.y()
    return key.decode('hex')
```

```

# Generate a new private key.
secret = random_secret()
print "Secret: ", secret

# Get the public key point.
point = secret * generator
print "EC point:", point

print "BTC public key:", get_point_pubkey(point).encode("hex")

# Given the point (x, y) we can create the object using:
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec_order)
assert point1 == point

```

## Python

## ECDSAライブラリのインストールとec\_math.py

スクリプトの実行はこのコードを実行した結果を示しています。

### NOTE

上記の例コードでは `os.urandom` を使用しており、これは裏で動作しているオペレーティングシステムによって提供されている暗号的に安全な乱数生成器(CSRNG)の値を反映しています。LinuxのようなUNIX-likeなオペレーティングシステムの場合これは `/dev/urandom` から乱数を取得し、Windowsの場合 `CryptGenRandom()` を呼び出すことで乱数を取得します。もし適した乱数発生源がない場合、`NotImplementedError` が発生します。ここで使われている乱数生成器はデモ目的のものであり、十分なセキュリティを持ったように実装されていないので商用レベルのクオリティを持ったBitcoinキーを生成するには適切ではありません。

### Example 7. Python ECDSAライブラリのインストールとec\_math.pyスクリプトの実行

```

$ # Install Python PIP package manager
$ sudo apt-get install python-pip
$ # Install the Python ECDSA library
$ sudo pip install ecdsa
$ # Run the script
$ python ec-math.py
Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873

```

# ウォレット

ウォレットは秘密鍵のためのコンテナであり、通常構造化されたファイルまたは簡単なデータベースとして実装されています。キーを作るもう1つの方法は、決定性鍵生成です。ここでは、一方向ハッシュ関数を使って前に出てきた秘密鍵から順々に新しい秘密鍵を作ってみます。秘密鍵を再生成するために必要なのは最初の鍵(シード または マスター キーとして知られているもの)だけです。この節では、キー生成の方法と決定性キーによって構築されたウォレットの構造を説明します。

## TIP

Bitcoinウォレットはキーを保持していますが、bitcoinは保持していません。それぞれのユーザはbitcoinではなくキーを含むウォレットを持つこととなります。ウォレット自体は本当に秘密鍵/公開鍵のペアを含むまさにキーホルダーなのです(秘密鍵と公開鍵参照)。ユーザはトランザクションにキーを使って署名をし、トランザクションアウトプット(ユーザのbitcoin)を所有していることを証明します。このbitcoinはトランザクションアウトプット(よくvoutまたはtxoutと書かれます)の形でブロックチェーン上に保存されています。

## 非決定性(ランダム)ウォレット

最初のBitcoinクライアントでは、ウォレットは単にランダムに生成された秘密鍵の集まりでした。このタイプのウォレットを *Type-0* 非決定性ウォレット と呼びます。例えば、Bitcoin Coreクライアントは、初回起動のときにあらかじめ100個のランダムな秘密鍵を作成し、個々のキーは一度しか使われないためその後必要に応じてさらにキーを作ります。このタイプのウォレットは"Just a Bunch Of Keys"またはJBOKというニックネームがついています。ランダムなキーの不利な点は、もし多く生成したとするとそれら全てのコピーも保持しなければいけなくなる点です。つまり、ウォレットは頻繁にバックアップされなければならないということです。もしバックアップされていないければウォレットがアクセス不可能になって管理資金が永久に失われてしまうのです。このため、トランザクションごとに1回だけBitcoinアドレスを使うというBitcoinアドレス再利用回避方法は使えなくなってしまうのです。Bitcoinアドレスの再利用は、Bitcoinアドレスが多くのトランザクションに結びつくことでプライバシーの低下に繋がります。特にBitcoinアドレスの再利用を避けたいのであれば、*Type-0* 非決定性ウォレットを選ぶことはやめたほうがよいでしょう。多くのキーを管理することを意味し、頻繁にバックアップを作る必要が生じてしまうからです。Bitcoin Coreクライアントは*Type-0* ウォレットを含んでいますが、このウォレットの使用はBitcoin Coreの開発者たちから反対されています。[Type-0非決定性\(ランダム\)ウォレット: ランダムに生成されたキーのコレクション](#)は非決定性ウォレットを示しており、ランダムなキーの緩い集まりを表現しています。

## 決定性(Deterministic , Seeded)ウォレット

決定性(Deterministic)または"Seeded"ウォレットは、全て共通のシードから生成される秘密鍵を持っているウォレットです。共通のシードからの生成は一方向ハッシュ関数を使います。このシードはランダムに生成される数値で、この数値は指数または"chain code"(階層的決定性ウォレット(BIP0032/BIP0044)参照)のような秘密鍵を生成するためのその他のデータと結びついています。決定性ウォレットでは、このシードを使えば生成された全てのキーを復活できるため、このシードを生成した時点で一回バックアップを取っておけば十分です。このシードはまたウォレットのエクスポートやインポートでも重要で、異なったウォレットの間でのキーの移行を簡単にしてくれるのです。



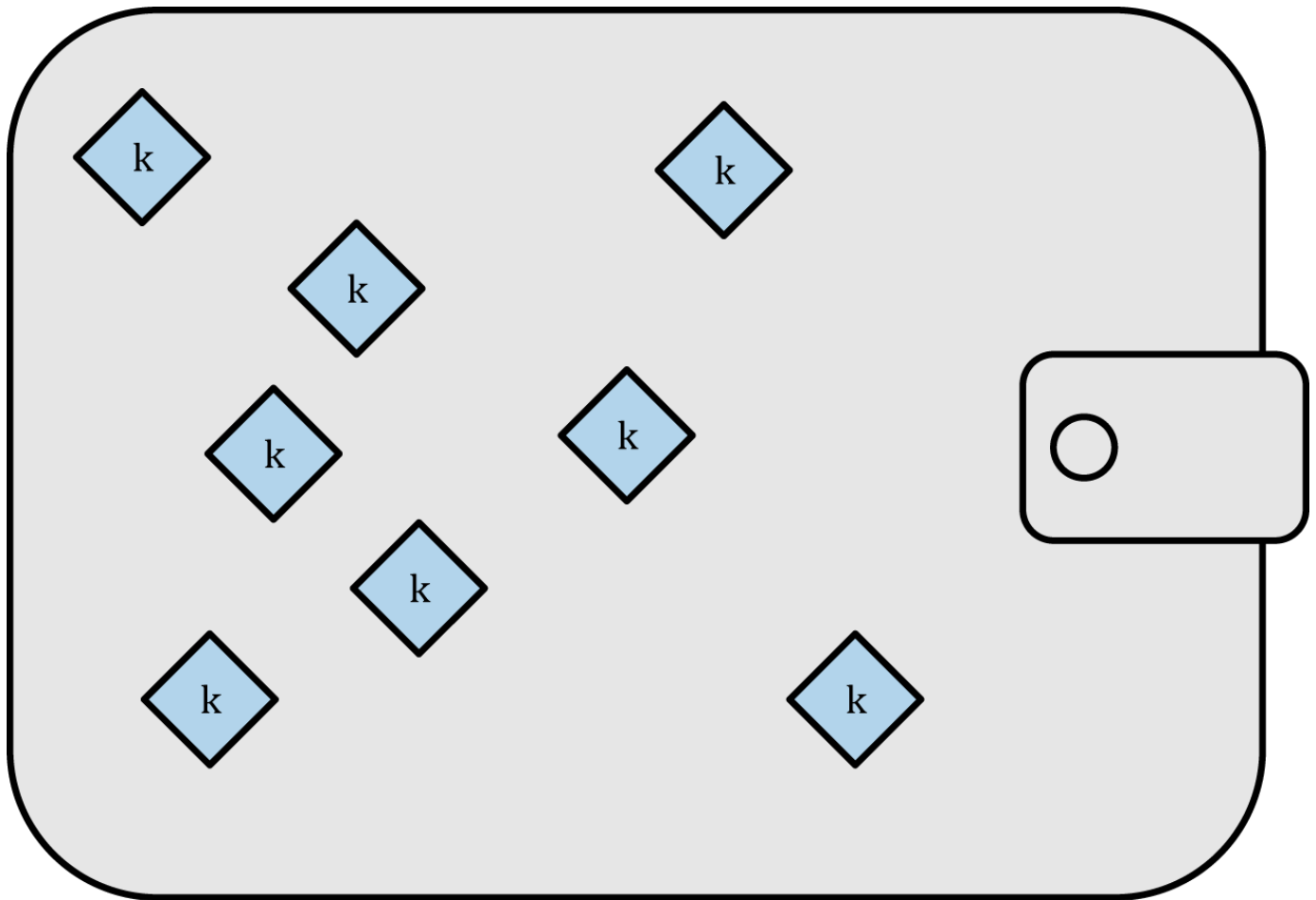


Figure 8. Type-0非決定性(ランダム)ウォレット: ランダムに生成されたキーのコレクション

## Mnemonic Code Words

mnemonic codeは、決定性ウォレットのシードとして使われるランダムな数値を表す(エンコードする)英単語の配列です。この英単語配列は、シードを再生成するために使われ、このシードからウォレットと全てのキーを再生成します。mnemonic

codeを伴った決定性ウォレットは、ウォレットの初期設定時にユーザに12から24個の英単語の配列を示します。この英単語配列はウォレットのバックアップであり、全てのキーを復活させるまたは再生成するのに使用できます。mnemonic

codeによってユーザはウォレットのバックアップを取りやすくなります。なぜなら、ランダムな数字と比べてこの英単語配列は読みやすく、また正確に転写できるからです。

mnemonic codeは Bitcoin Improvement Proposal 39 ([bip0039]参照)で定義されており、現在草案段階にあります。BIP0039は草案段階提案であり、まだ標準ではないことに注意してください。特にどの英単語群を使うかについて異なる標準があり、ElectrumウォレットはBIP0039に先行して別の英単語群を使っています。BIP0039はTrezorウォレットやいくつかの他のウォレットでも使われていますが、Electrumの実装とは相容れないものになっています。

BIP0039は以下のようにmnemonic codeとシードの生成を定義しています。

1. 128bitから256bitのランダムな配列(entropy)を生成

2. ランダムな配列のSHA256ハッシュの先頭4bitを取得し、ランダムな配列のチェックサムを生成
3. このチェックサムをランダムな配列の最後に付加
4. 2048個のあらかじめ決められた英単語の辞書のインデックスとして使うために、この配列を11bitずつの部分に分割
5. mnemonic codeを表す12から24個の英単語を生成

Mnemonic codes: エントロピーと単語長はエントロピーデータサイズとmnemonic codeの単語数の関係を示しています。

Table 5. Mnemonic codes: エントロピーと単語長

| エントロピー (bits) | チェックサム (bits) | エントロピー+チェックサム | 単語数 |
|---------------|---------------|---------------|-----|
| 128           | 4             | 132           | 12  |
| 160           | 5             | 165           | 15  |
| 192           | 6             | 198           | 18  |
| 224           | 7             | 231           | 21  |
| 256           | 8             | 264           | 24  |

mnemonic codeは128から256bitになっており、PBKDF2というキー拡張関数を使うことでmnemonic codeからより長いシード(512bit)が生成されます。結果として出てくるシードは決定性ウォレットと全ての鍵を生成することに使われます。

表 [<xref linkend="table\\_4-6" xrefstyle="select: labelnumber"/>](#) と [<xref linkend="table\\_4-7" xrefstyle="select: labelnumber"/>](#) に、mnemonic codeとそれらが作り出したシードのいくつかの例を示します。

Table 6. 128bitエントロピーから得たmnemonic codeと出力されたシード

|                        |  |
|------------------------|--|
| エントロピーインプット (128 bits) | 0c1e24e5917779d297e14d45f14e1a1a   |
| Mnemonic (12 単語)       | army van defense carry jealous true garbage claim echo media make crunch   |
| シード (512 bits)         | 3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf7 |

Table 7. 256bitエントロピーから得たmnemonic codeと出力されたシード

|                        |  |
|------------------------|--|
| エントロピーインプット (256 bits) | 2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c |
|------------------------|--|

|                         |   |
|-------------------------|---|
| <b>Mnemonic (24 単語)</b> | cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige |
| <b>シード (512 bits)</b>   | 3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343                    |

## 階層的決定性ウォレット(BIP0032/BIP0044)

決定性ウォレットは1つの"シード"から多くのキーを生成しやすくするために開発されました。決定性ウォレットの最も進んだ形は、BIP0032で定義されている 階層的決定性ウォレット または HDウォレットです。階層的決定性ウォレットはツリー構造をなしているキーを含んでいて、このツリー構造は親キーが子キー群を作り、それぞれの子キーが孫キー群を作り出すような感じに無限に続いていきます。Type-2階層的決定性ウォレット: 1つのシードから生成されたキーツリー 図でツリー構造を説明しています。("hierarchical deterministic wallets (HD wallets)", "tree structure for"

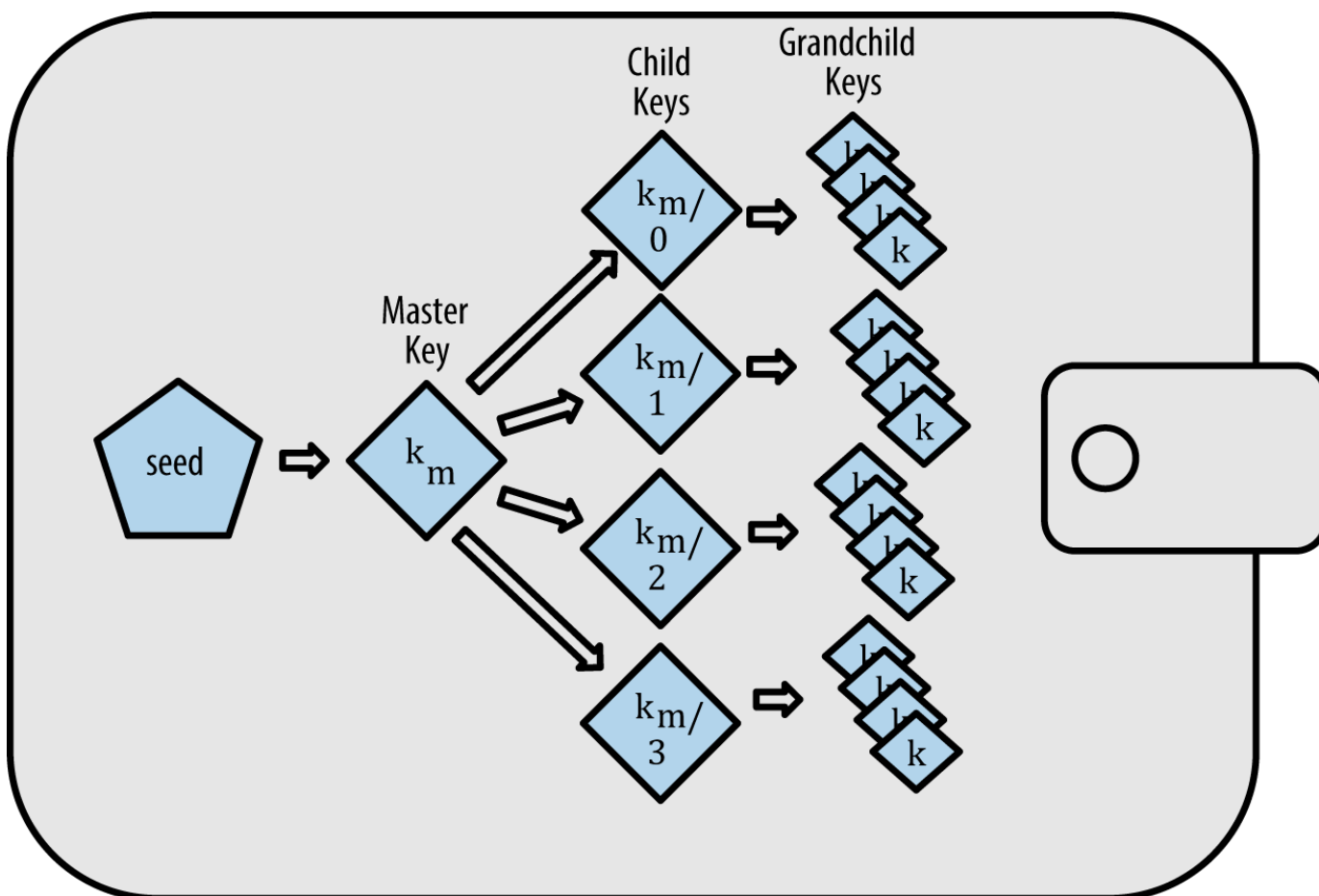


Figure 9. Type-2階層的決定性ウォレット: 1つのシードから生成されたキーツリー

### TIP

もしBitcoinウォレットを実装するのであれば、BIP0032とBIP0044標準に従ったHDウォレットとして構築するべきです。

HDウォレットは、ランダムな(非決定性)キーに比べて2つの主な利点があります。1つ目としては、ツリー構

造は付加的な組織的意味を表すのに使うことができる点です。例えば、サブキーの特定のブランチを支払いを受け取ることに使う場合や、異なるブランチをおつりを受け取ることに使う場合です。キーのブランチはまた企業内の状況に合わせて使われます。例えば、部署ごと、課ごと、特定の機能集団ごと、口座種類ごとにブランチを割り当てるような場合です。

2つ目の利点としては、ユーザが秘密鍵に触れることなく公開鍵を生成できる点です。それぞれのトランザクションごとに異なる公開鍵を発行することで、安全でないサーバや受信用にしかしていないサーバも使うことができます。公開鍵をあらかじめこれらのサーバに置いておいたり先に生成しておいたりする必要はありませんが、これらのサーバは資金を使うときに必要な秘密鍵を持つことはできません。

### シードからのHDウォレット作成

HDウォレットは単一の `root seed` から作られ、ルートシードは128bit、256bit、512bitのランダムな数値です。HDウォレットにある他の全ては、決定性的にルートシードから導出され、このルートシードからHDウォレット全体を再生成できます。ルートシードだけを単にコピーすれば数千個、または数百万個のキーがあってもHDウォレットをバックアップ、リストア、エクスポートしやすくなるということです。ルートシードは前の節 `Mnemonic Code Words` で書いたように `mnemonic word sequence` によってしばしば表現され、ルートシードを転写、保存をしやすくしています。

HDウォレットでマスターキーとマスターchain codeを生成するプロセスを `root seedからマスターキーとマスターchain codeの生成` を示します。

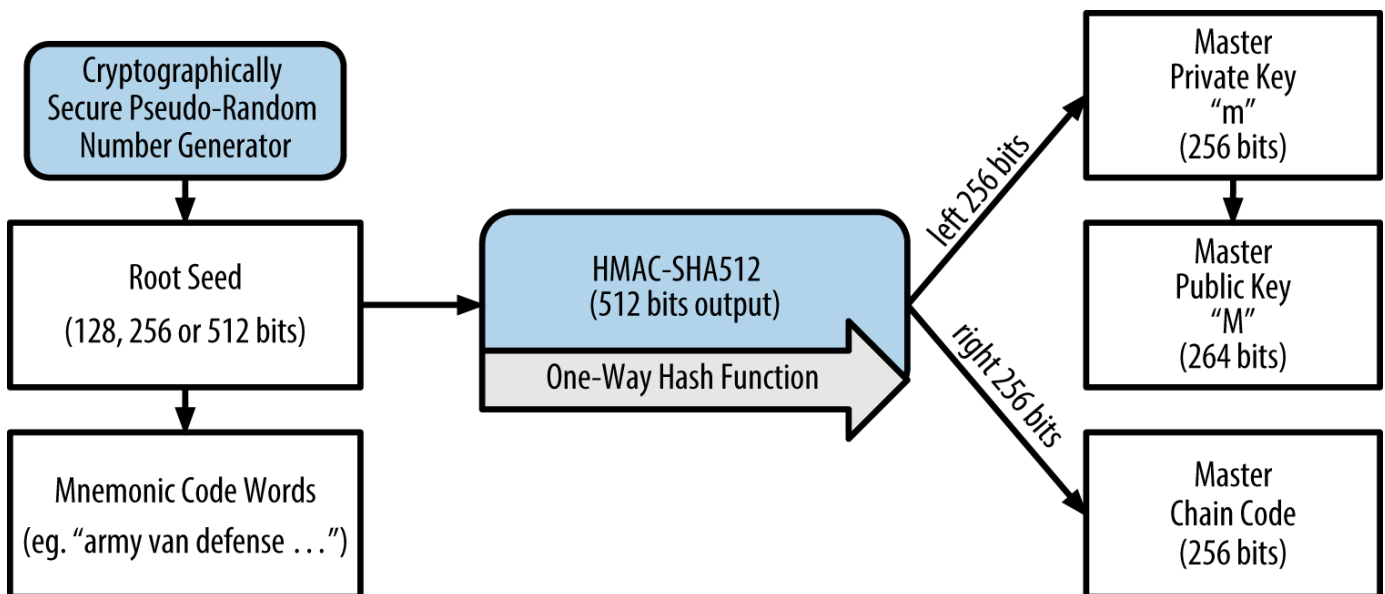


Figure 10. ルートシードからのマスターキーとマスターchain codeの生成

ルートシードはHMAC-SHA512アルゴリズムに対するインプットであり、結果として出力されたハッシュはマスター秘密鍵 `m` と マスターchain code を生成するために使われます。マスター秘密鍵 `m` は対応したマスター公開鍵 `M` を生成し、このときこの章の最初に見た標準的な楕円曲線上のスカラー倍算プロセス  $m * G$  が使われています。次の章で見るように、このchain codeは親キーから子キーを生成するプロセスの中でエントロピー(乱雑さ)を導入するために使われます。

### 子秘密鍵の導出(private child key derivation)

階層的決定性ウォレットは `子キー導出(child key derivation)`

(CKD)関数を使って親キーから子キーを導出します。

child

key

derivation関数は一方向ハッシュ関数をベースにしており、以下の組み合わせによって成っています。

- 子秘密鍵または子公開鍵(ECDSA非圧縮鍵)
- chain code(256bit)と呼ばれるシード
- インデックス(32bit)

表面的には、このchain

codeによって生成される子秘密鍵に十分なランダムさが含まれることとなります。そして、chain

codeなくインデックスだけではその他の子キーを生成することはできません。子キーを保持していたとしても、もしchain codeも持っていなければ他の子キーを見つけることはできないのです。最初のchain codeシード(ツリー構造のルート)はランダムなデータから作られ、次のchain codeはそれぞれの親chain codeから作られます。

以下のように、これらの3つの要素は組み合わせられ子キーを生成するためにハッシュ化されます。

親公開鍵、chain code、インデックスは組み合わせられ、512bitハッシュを生成するためにHMAC-SHA512アルゴリズムでハッシュ化されます。出力されたハッシュは2つの部分に分けられます。右半分の256bitは子chain codeになり、左半分の256bitとインデックスは親秘密鍵と合わせて子秘密鍵となります。子秘密鍵を生成するための親秘密鍵拡張図は、インデックスが0のときの0番目の子を作り出す状況を表しています。

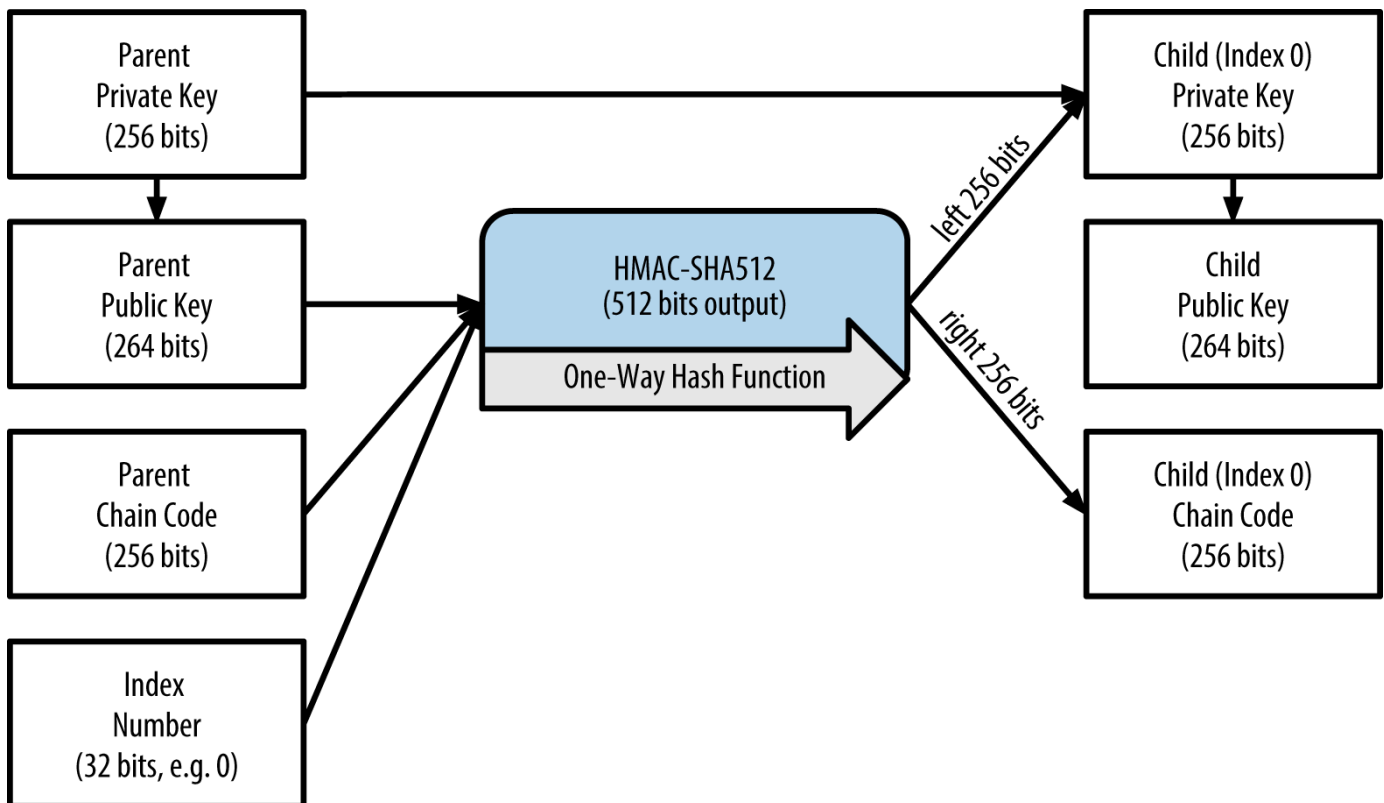


Figure 11. 子秘密鍵を生成するための親秘密鍵拡張

インデックスを変えることで、子0、子1、子2...と子を作り出していくことができ、それぞれの親キーは20億個の子キーを持つことができます。

このプロセスを繰り返すことで、ツリー構造を無限に下っていくことができ、それぞれの子が次々に親となり自身の子供を作っていきます。

### 生成された子キーの使用

それぞれ子秘密鍵たちは互いにどういう関係にあるかを知ることができません。HMAC-SHA512は一方関数であるため、子秘密鍵は親秘密鍵を探すことができずまた、子秘密鍵は自分の兄弟を探すこともできません。もしn番目の子があったとしても、n-1番目の子やn+1番目の子、それ以外のいかなる子も探すことができません。ただ唯一親秘密鍵とchain codeだけが全ての子を作り出すことができます。もし子chain codeがないと、子秘密鍵は孫秘密鍵を作ることもできません。子秘密鍵と子chain codeの両方があることで初めて孫秘密鍵を作り出すことができるのです。

子秘密鍵は何のために使われるのでしょうか？子秘密鍵は公開鍵とBitcoinアドレスを作ることに使われます。また、支払いに使われるトランザクションに署名をするときにも使われます。

#### TIP

子秘密鍵と、これに対応した子公開鍵およびBitcoinアドレスは、HDウォレットではない全てランダムに生成されたキーやアドレスと見分けがつかず、これらが生成列の一部であるという事は自分自身からは分からないのです。これらが一度生成されると、"普通"のキーと全く同じように使えます。

### 拡張鍵

前に見たように、キー導出関数は子供を作ることに使われ、キー、chain code、インデックスという3つのインプットに基づいています。本質的に必要なインプットはキーとchain codeで、これらを組み合わせたものは **拡張鍵** と呼ばれています。拡張鍵という言葉はまた、"拡張可能鍵"と呼ばれることもあります。なぜなら、このキーで子供を生成できるからです。

拡張鍵は単に256bitのキーと256bitのchain codeを単にくっつけて512bitにしたものです。拡張鍵には2つの種類があります。拡張秘密鍵は秘密鍵とchain codeの組み合わせで、子秘密鍵(そして、これから子公開鍵も)を生成するために使われます。拡張公開鍵は公開鍵とchain codeの組み合わせで、子公開鍵を生成するために使われます。詳細については[公開鍵の生成](#)に記載しています。

拡張鍵をツリー構造のブランチのルートと考えてみましょう。ブランチのルートを使って、ブランチの残りを生成することができます。拡張秘密鍵は完全なブランチを作ることができますが、一方拡張公開鍵は公開鍵のブランチしか作ることができません。

#### TIP

拡張鍵は秘密鍵または公開鍵、chain codeで構成されています。拡張鍵は子供を生成し、ツリー構造の中に子供自身のブランチを作り出していくことができます。拡張鍵を共有することで、ブランチ全体を参照することができます。

拡張鍵はBase58Checkでエンコードされ、BIP0032互換ウォレットの間でエクスポートとインポートを簡単に行うことができます。拡張鍵のBase58Checkはプレフィックスとして"xprv"と"xpub"という特別なversion byteを使います。拡張鍵は512または513bitなので、前に見たBase58Checkエンコード文字列よりも長くなっています。

ここにBase58Checkでエンコードされた拡張秘密鍵の例を示します。

```
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMK
Uga5biW6Hx4tws2six3b9c
```

以下はこの拡張秘密鍵に対応した拡張公開鍵です。これもBase58Checkでエンコードされています。

```
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMst
weyLXhRgPxdp14sk9tJPW9
```

### 子公開鍵の導出(public child key derivation)

前に書いたように、階層的決定性ウォレットには秘密鍵を使うことなく、親公開鍵から子公開鍵を作り出せるというとても有用な特徴を持っています。この性質から、子公開鍵を作る方法には2種類あることが分かります。子公開鍵を子秘密鍵から作るか、親公開鍵から作るか、です。

このため、拡張公開鍵は全ての公開鍵(そして公開鍵のみ)を生成することができます。

これにより、拡張公開鍵のコピーだけをもち全く秘密鍵を持たないサーバやアプリケーションで、とても安全な公開鍵のみの生成ができるようになります。この仕組みを使うと、制限なく公開鍵とBitcoinアドレスを作り出すことができますが、これらのBitcoinアドレスに送られるお金を使うことができません。一方、より安全なサーバでは拡張秘密鍵を使ってさきほどのBitcoinに対応した秘密鍵を生成でき、トランザクションに署名することでお金を使うことができます。

この応用として、Eコマースを提供するwebサーバに拡張公開鍵を置く場合が考えられます。webサーバは公開鍵のderivation関数を使ってBitcoinアドレスをトランザクションごと(例えば顧客のショッピングカートごと)に新しいBitcoinアドレスを作ることができます。しかし、このwebサーバは盗難の攻撃を受けやすい秘密鍵を持っていません。HDウォレットを使わない場合、これを実行する唯一の方法は、切り離された安全なサーバで数千のBitcoinアドレスを生成し、あらかじめEコマースサーバ上にBitcoinアドレスを読み込んでおく方法です。ただしこの方法は、webサーバがBitcoinアドレスを"払い出せ"ないため、Bitcoinアドレスが枯渇しないように定期的なメンテナンスが必要となります。

別の応用としては、コールドストレージまたはハードウェアウォレットへの応用です。この応用では、拡張秘密鍵はペーパーウォレットまたはハードウェアデバイス(例えば、Trezor hardware wallet)に保存されます。一方、拡張公開鍵はオンライン上に保持されます。ユーザは"受け取り用"のBitcoinアドレスを自由に作ることができますが、秘密鍵は安全にオフラインに保存されます。資金を使うには、ユーザはオフラインの署名用Bitcoinクライアント上で秘密鍵を使って行るか、ハードウェアウォレットデバイス上でトランザクションに署名するかをしなければいけません。[子公開鍵を生成するための親公開鍵拡張図](#)は親公開鍵から子公開鍵を導出するメカニズムを説明しています。

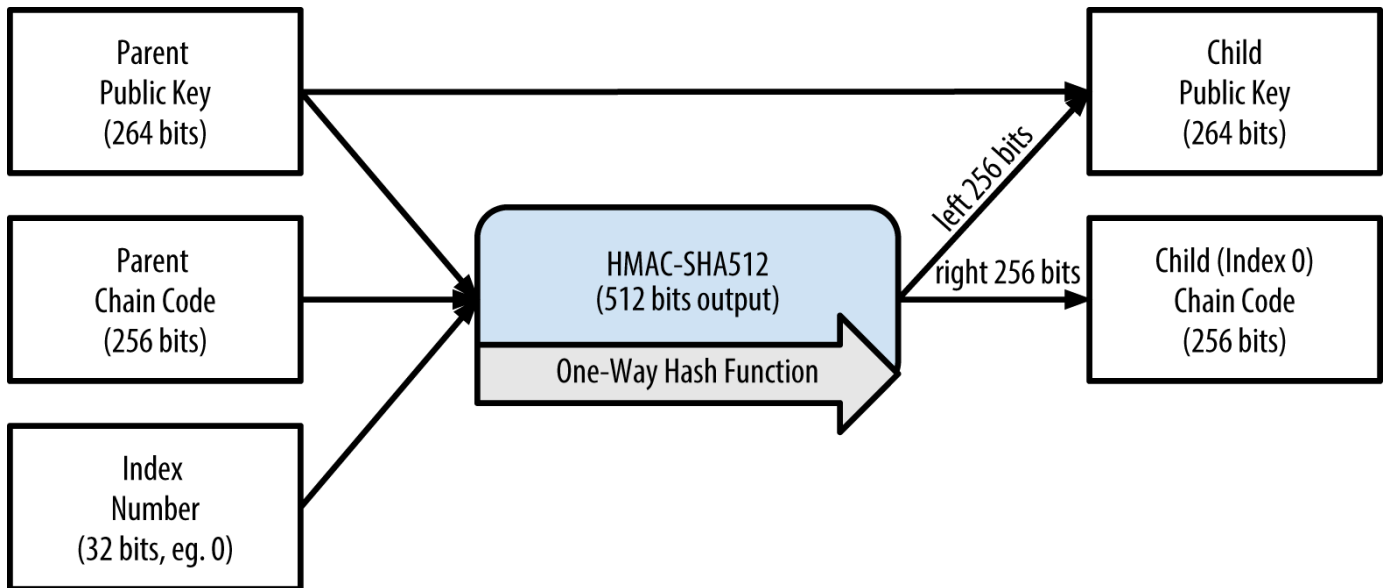


Figure 12. 子公開鍵を生成するための親公開鍵拡張

### 強化子公開鍵導出(hardened child key derivation)

拡張公開鍵から公開鍵のブランチを導出する方法はとても有用ですが、潜在的リスクも持っています。拡張公開鍵にさわられても子秘密鍵にはさわられません。しかし、拡張公開鍵はchain codeを含んでいるため、もし子秘密鍵が知られているまたは漏洩してしまった場合、このchain codeを使ってその他全ての子秘密鍵を導けてしまうのです。親chain codeを伴った1つの子秘密鍵の漏洩により、全ての子供の秘密鍵が明らかになってしまいます。悪いことに、親chain codeを伴った子秘密鍵は親秘密鍵を推測することに使うことができるのです。

このリスクへの解決策として、HDウォレットは *hardened derivation*(強化導出) と呼ばれるもう1つのderivation関数を使っています。この関数は、親公開鍵と子chain codeの間の関係を"壊す"ものです。hardened derivation関数は親公開鍵の代わりに親秘密鍵を使って子chain codeを導出します。これは親秘密鍵または兄弟秘密鍵が漏洩しないようなchain codeを使って親と子の間に"ファイヤーウォール"を作ります。hardened derivation関数は通常のchild key derivationとほとんど同じように見えますが、[子キーのhardened derivation: 親公開鍵の省略](#)図に示すように親公開鍵の代わりに親秘密鍵がハッシュ関数のインプットとして使われる点が異なります。



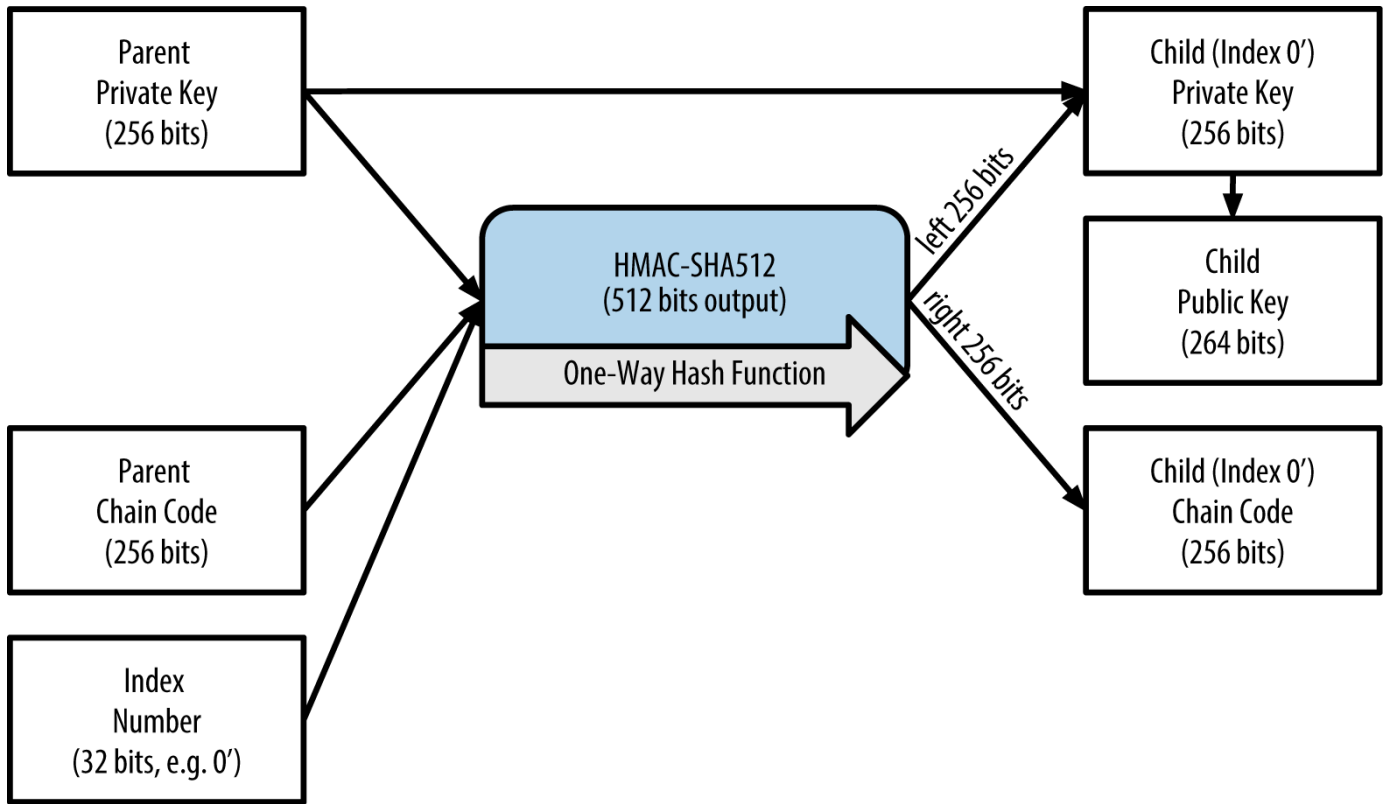


Figure 13. 子キーのhardened derivation: 親公開鍵の省略

子秘密鍵hardened derivation関数で出力される子秘密鍵とchain codeは、通常のderivation関数から得られる結果とは完全に異なります。結果として出てくる鍵の"ブランチ"は脆弱ではない拡張公開鍵を生成します。なぜなら、この拡張公開鍵に含まれているchain codeは、いかなる秘密鍵も攻撃できないようになっているからです。よって、hardened derivationは拡張公開鍵が使われる階層よりも上のツリーに行けないようにする"ギャップ"を作り出すのです。

もし拡張公開鍵の利便性を使い、しかもchain codeの漏洩リスクを回避したいのであれば、通常の親(親公開鍵)ではなく強化された親(親秘密鍵)から拡張公開鍵を導出すべきです。ベストプラクティスとしては、マスターキーの1階層目の子供を常にhardened derivationを通して導出されるようにしておくことがよいでしょう。

### 通常およびhardened derivationのインデックス

derivation関数で使われているインデックスは32bitの整数です。通常のderivation関数を通して得られたキーと強化されたderivation関数を通して得られたキーを簡単に区別するために、このインデックスを2つの範囲に分けておきます。0から $2^{31}-1$ (0x0 から 0x7FFFFFFF)までのインデックスは通常のderivation関数にのみ使われます。 $2^{31}$ から $2^{32}-1$ (0x80000000 から 0xFFFFFFFF)はhardened derivation関数にのみ使われます。よって、もしインデックスが $2^{31}$ より小さければ、子は通常のderivation関数から生成されたもの、一方もしインデックスが $2^{31}$ と等しいかそれより上であれば、子は強化されたderivation関数から生成されたものです。

読んだり表示したりしやすいように、強化された子供に対するインデックスは0から始まるダッシュ付きの数字で表されます。最初の通常derivation関数による子キーは0と表示され、一方最初のhardened derivation関数による子キー(インデックス 0x80000000)は

<markup>0'</markup>と表示されます。次に、二番目のhardenedキーはインデックス 0x80000001 を持ち 1' を表示されます。HDウォレットのインデックス i' を見たときは、これは  $2^{31} + i$  を表していると考えてください。

### HDウォレットキー識別子(path)

HDウォレットにあるキーは、"path"命名規則を使って一意に指定されます。このpath命名規則は、ツリーの階層をスラッシュ(/)で区切って表します([HDウォレットのpath例参照](#))。マスター秘密鍵から得られた秘密鍵は"m"から始まります。マスター公開鍵から得られた公開鍵は"M"から始まります。このため、マスター秘密鍵の最初の子秘密鍵はm/0で、最初の子公開鍵はM/0、最初の子の二番目の孫はm/0/1などとなります。

キーの"先祖"をたどるには右から左に読み、それが得られたマスターキーに到達するまで読んでいきます。例えば、識別子 m/x/y/z はz番目の m/x/y の子キーを表し、また m/x/y は m/x のy番目の子キー、m/x は m のx番目の子キーを表します。

Table 8. HDウォレットのpath例

| HD path     | 説明                               |
|-------------|----------------------------------|
| m/0         | マスター秘密鍵(m)から生成された最初(0)の子秘密鍵      |
| m/0/0       | 最初の子供(m/0)の最初の孫秘密鍵               |
| m/0'/0      | 最初の強化された子供 (m/0')の最初のノーマルな孫秘密鍵   |
| m/1/0       | 二番目の子供(m/1)の最初の孫秘密鍵              |
| M/23/17/0/0 | 24番目の子供の、18番目の孫の、最初の曾孫の、最初の玄孫公開鍵 |

### HDウォレットツリー構造をたどってみよう

HDウォレットツリー構造は非常に大きな柔軟性を持っています。それぞれの親拡張鍵は40億個の子供を持つことができます(20億個の通常の子供と20億個の強化された子供)。それぞれの子供は、もう1つ40億個の子供を持つことができ、これがどんどん続きます。ツリーは好きなだけ深くすることができ、制限なく世代を作っていくことができます。制限なく作っていくことはできますが、無限のツリーをたどろうとするととても時間がかかってしまいます。特にHDウォレットを他のウォレットに移そうとするときとても大変です。

この複雑さを解決する2つのBitcoin Improvement Proposal (BIP)が提案されています。2つ目のBIP0043は、ツリー構造の"目的"が明確化された特別な識別子を最初の拡張された子インデックスとして使うということを提案しています。BIP0043に基づけば、HDウォレットはツリーの1階層目に1つだけブランチを持ち、その他の階層は目的が定義された構造や名前空間を持つインデックスを伴うはずでです。例えば、1階層目にm/i/ のブランチだけを持っている HDウォレットは特定の目的に沿ったウォレットとして使われることを想定されており、この目的はインデックス "i" で指定される目的になります。

このBIPを拡張することで、BIP0044はBIP0043の元での"目的"を表す数字 44' を定義しており、これは複数の口座を保持する目的を表すものです。ウォレットがBIP0044の構造に従っているかどうかは、1階層目が m/44'/だけになっていることから確認できます。

BIP0044では、以下のように5つの事前に定義された階層構造を提案しています。

m / purpose' / coin\_type' / account' / change / address\_index

最初の階層"purpose"は常に 44' になります。第2階層 "coin\_type"は暗号通貨コインの種類を表し、個々の通貨が第2階層以下に独自のサブツリーを持つような複数の通貨を扱えるHDウォレットを作ることができるようになっていています。現在は3つの通貨が定義されており、Bitcoinは m/44'/0'、Bitcoin Testnetは <markup>m/44'/1'</markup>、Litecoinは<markup>m/44'/2'</markup>になっています。

第3階層"account"は、ユーザが複数の口座を使えるようにし、会計や組織的な目的で使えるようにしています。例えば、HDウォレットは2つのBitcoin "口座" <markup>m/44'/0'/0'</markup>、<markup>m/44'/0'/1'</markup>を持つかもしれません。それぞれの口座はそれぞれ自身のサブツリーのルートになっています。

第4階層"charge"では、HDウォレットは2つのサブツリーを持つことができ、1つは受取用アドレスで1つはおつり用アドレスです。前の階層ではhardened derivationが使われたのですが、この階層では通常の導出が使われています。これは、拡張公開鍵を安全ではない環境で使うようにするためです。使うことのできないアドレスが第4階層の子供としてHDウォレットから導出され、第5階層"address\_index"を作ります。例えば、最初の口座の三番目のbitcoin受け取り用アドレスはM/44'/0'/0'/0/2になります。 [BIP0044のHDウォレット構造例](#)にいくつかの例を示します。

Table 9. BIP0044のHDウォレット構造例

| HD path          | 説明                                     |
|------------------|--|
| M/44'/0'/0'/0/2  | 最初のBitcoin口座に対する3番目の受信公開鍵              |
| M/44'/0'/3'/1/14 | 4番目のBitcoin口座に対する15番目のおつり用公開鍵          |
| m/44'/2'/0'/0/1  | トランザクションに署名するための、Litecoinメイン口座の2番目の秘密鍵 |

### Bitcoin Explorerを使ったHDウォレットの実験

[\[ch03\\_bitcoin\\_client\]](#)で紹介したBitcoin

Explorerコマンドラインツールを使うと、異なった形式での表現と同時にBIP0032決定性キーを生成したり拡張したりする実験をしてみることができます。:

```

$ bx seed | bx hd-new > m # create a new master private key from a seed and store in
file "m"
$ cat m # show the master extended private key
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaRgY5uPwV7RpEgHs7cvdg
fiSjLjjbuGKGcjRyU7RGGSS8Xa
$ cat m | bx hd-public # generate the M/0 extended public key
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunS
DMstweyLXhRgPxdp14sk9tJPW9
$ cat m | bx hd-private # generate the m/0 extended private key
xprv9tyUQV64JT5qs3RSTJKXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6G
oNMKUga5biW6Hx4tws2six3b9c
$ cat m | bx hd-private | bx hd-to-wif # show the private key of m/0 as a WIF
L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN
$ cat m | bx hd-public | bx hd-to-address # show the bitcoin address of M/0
1CHCnCjgMNb6digimckNQ6TBVcTWBAmPHK
$ cat m | bx hd-private | bx hd-private --index 12 --hard | bx hd-private --index 4 #
generate m/0/12'/4
xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqqwoS6YZBLatRZRy
UeAkRPe1kLR1P6Mn7jUrXFquUt

```

## 高度なキーとアドレス

次の節では、暗号化秘密鍵、scriptとマルチシグネチャアドレス、文字列指定のあるBitcoinアドレス(vanity address)、ペーパーウォレットなど高度なキーとアドレスを見ていきます。

### 暗号化秘密鍵(BIP0038)

秘密鍵は極秘にしておかなければいけません。ただ秘密鍵の機密性は、実用上達成することが大変難しいことはよく知られていることです。というのは、機密性といつでも安全に使えることの両立が難しいためです。秘密鍵を秘密に保つことは、秘密鍵のバックアップを保持し紛失を避けるようにするとさらに難しくなります。ウォレットにあるパスワードで暗号化された秘密鍵は安全かもしれませんが、ウォレットはバックアップしておかなければいけないのです。例えばウォレットをアップグレードする、または別のウォレットに変えるといったときに、ユーザは鍵を1つのウォレットから別のウォレットに移動する必要があります。秘密鍵のバックアップもまた紙(ペーパーウォレット参照)、またはUSBフラッシュメモリのような外部ストレージに保存されるかもしれませんが、しかし、バックアップそのものが盗まれたり紛失してしまったらどうなるのでしょうか？これらの両立が難しいセキュリティ問題を解決するために、持ち出し可能で便利な暗号化秘密鍵が考案されました。この暗号化秘密鍵は、多くのウォレットやBitcoinクライアントに実装されており、Bitcoin Improvement Proposal 38またはBIP0038([bip0038](#)参照)で標準化されています。

BIP0038は、安全なバックアップメディアへの保存およびウォレット間の転送ができるように、またキーが晒される可能性のある状況にも対応できるように、パスフレーズで秘密鍵を暗号化しさらにBase58Checkでエンコードする標準規格を提案しています。この暗号化基準は、Advanced Encryption Standard (AES)を採用しています。AESはNational Institute of Standards and Technology

(NIST)によって開発され、商用または軍用アプリケーションの暗号化実装に広く使われているものです。

BIP0038暗号化スキームでは、インプットとしてBitcoin秘密鍵を取り、通常プレフィックス "5" を伴ったBase58Checkを使いWallet Import Format (WIF)にエンコードされます。さらにBIP0038暗号化スキームでは、アルファベットと数字が混ざった複雑な文字列で構成されるパスコードという長いパスワードを使います。BIP0038暗号化スキームの結果として、6P から始まるBase58Checkでエンコードされた暗号化秘密鍵が得られます。もしキーが6P から始まっていれば、それは暗号化されており、どのウォレットでも使えるWIF形式秘密鍵(5 から始まる)に戻すためにはパスフレーズが必要だということが分かります。現在多くのウォレットはBIP0038暗号化秘密鍵を実装しており、キーをインポートし復号化するときにパスフレーズを求められるでしょう。非常に使いやすいブラウザベースの Bit Address (Wallet Detailsタブ参照)などのサードパーティーのアプリケーションでもBIP0038キーを復号化することができます。

BIP0038暗号化キーの最も多い利用用途は、秘密鍵を紙にバックアップするときです。ユーザが協力的なパスフレーズを選んでいる限りBIP0038暗号化秘密鍵を伴ったペーパーウォレットは非常に安全で、オフラインBitcoinストレージ("コールドウォレット"と呼ばれてもいます)を作るための最高の方法です。

パスフレーズを入れることでどのように復号化されたキーを得るかを理解するために [bitaddress.org](http://bitaddress.org) を使ってBIP0038暗号化秘密鍵の例にある暗号化されたキーをテストしてみてください。

Table 10. BIP0038暗号化秘密鍵の例

|                |  |
|----------------|--|
| 秘密鍵 (WIF)      | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbncyhfsYB1Jcn         |
| パスフレーズ         | MyTestPassphrase   |
| 暗号化鍵 (BIP0038) | 6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ |

## Pay-to-Script Hash (P2SH) とマルチシグネチャアドレス

知っての通り、初期から使われているBitcoinアドレスは“1”から始まり、秘密鍵から得られた公開鍵から得られます。誰もがbitcoinを“1”から始まるBitcoinアドレスに送れますが、このbitcoinは対応する秘密鍵署名と公開鍵ハッシュを使うことでのみ使用できます。

“3”から始まるBitcoinアドレスはpay-to-script hash (P2SH)アドレスであり、ときどき間違ってマルチシグネチャアドレスまたはマルチシグアドレスと呼ばれます。これらではBitcoinトランザクションの受取人を、公開鍵の所有者の代わりにscriptのハッシュを使って指定します。この特徴は、2012年にBitcoin Improvement Proposal 16または BIP0016 ([[bip0016](#)]参照)で導入され、幅広く採用されています。というのは、この特徴により、アドレス自体に機能を追加することができるようになったためです。“1”から始まるBitcoinアドレスに資金を“送る”トランザクション pay-to-public-key-hash (P2PKH)とも呼ばれる)と違って、“3”で始まるBitcoinアドレスに送った資金のトランザクションでは、公開鍵ハッシュと秘密鍵署名以外の別のものも要求されます。要求されるものはBitcoinアドレスを作ったときにscriptの中で指定され、このBitcoinアドレスへの全てのインプットでそれらが要求されます。

pay-to-scriptハッシュアドレスはトランザクション

scriptから生成され、トランザクションアウトプットを誰が使えるかということを定義しています(詳細については[\[p2sh\]](#)参照)。pay-to-scriptハッシュアドレスのエンコードはBitcoinアドレスを生成するときに使われる二重ハッシュ関数と同じ関数を使って行われ、公開鍵の代わりにscriptにのみ適用されます。

```
script hash = RIPEMD160(SHA256(script))
```

出力された"script ハッシュ"はversion prefix 5でBase58Checkエンコーディングされ、これは3から始まるものになっています。P2SHアドレスの例は、32M8ednmuyZ2zVbes4puqe44NZumgG92sMで、これはBitcoin Explorerコマンド script-encode、sha256、ripemd160、base58check-encode ([\[libbitcoin\]](#)参照) を使って生成することができます。

```
$ echo dup hash160 [ 89abcdefabbaabbaabbaabbaabbaabbaabbaabba ] equalverify checksig >
script
$ bx script-encode < script | bx sha256 | bx ripemd160 | bx base58check-encode --version
5
3F6i6kwkevjr7AsAd4te2YB2zZyASEm1HM
```

#### TIP

P2SHはマルチシグネチャの標準トランザクションというわけではありません。P2SHアドレスはほぼ常にマルチシグネチャscriptを表しますが、他のトランザクションタイプのscriptを表す可能性もあるのです。

## マルチシグネチャアドレスとP2SH

現在、P2SHの最も一般的な実装は、マルチシグネチャアドレスscriptです。この名前が示しているように、scriptは所有権を証明し資金を利用するために1つ以上の署名を要求します。Bitcoinマルチシグネチャは、N個のキーから作られるM個の署名("threshold"とも呼ばれます)を要求し、これはM-of-N multisigと呼ばれています。ここで、MはNと等しいか小さい数です。例えば、[\[ch01\\_intro\\_what\\_is\\_bitcoin\]](#)に出てきたコーヒーショップのオーナーのボブは、彼のキーと彼の奥さんのキーから作られる1-of-2シグネチャーを要求するマルチシグネチャーアドレスを使うことができ、このアドレスに紐づいたトランザクションアウトプットを使うには彼または奥さんのどちらか一方のキーで署名すればよいのです。Gopesh(ボブのウェブサイトを作ったウェブデザイナー)がビジネス用に2-of-3マルチシグネチャアドレスを持っていたとすると、少なくともビジネスパートナーの2人がトランザクションに署名しなければこのアドレスに紐づく資金を使うことができません。

[\[transactions\]](#)で、P2SH(とマルチシグネチャ)でのトランザクションの作成方法について説明します。

## Vanity Address

文字列指定のあるBitcoinアドレス(Vanity Address)は人間が読むことができるメッセージを含んだBitcoinアドレスです。例えば、1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33には、最初の四文字に"Love"という単語が含まれています。Vanity addressでは、Bitcoinアドレスに望んだパターンが出るまで何回も秘密鍵を生成しチェックしなければいけません。vanity addressを生成するいくつかの効率的な方法があるものの、基本的に秘密鍵をランダムに生成して公開鍵、Bit



| 長さ | パターン         | 頻出度         | 探索時間    |
|----|--------------|-------------|---------|
| 5  | 1KidsC       | 6億5600万個に1個 | 1時間     |
| 6  | 1KidsCh      | 380億個に1個    | 2日      |
| 7  | 1KidsCha     | 2.2兆個に1個    | 3~4ヶ月   |
| 8  | 1KidsChar    | 128兆個に1個    | 13~18年  |
| 9  | 1KidsChari   | 7000兆個に1個   | 800年    |
| 10 | 1KidsCharit  | 40京個に1個     | 46,000年 |
| 11 | 1KidsCharity | 2300京個に1個   | 250万年   |

このように、たとえ数千台のコンピュータが使えたとしてもEugeniaは"1KidsCharity"を含むvanity addressをすぐに作ることはできないでしょう。望むパターンの文字数が1個増えると見つける難しさは58倍になります。7文字より多いパターンだと通常特別に設計されたハードウェアが必要になり、複数の graphical processing units (GPUs)を積んだカスタムデスクトップが必要になります。特別に設計されたハードウェアとして、Bitcoinマイニングではもう利益を生まなくなったBitcoinマイニング"マシン"をvanity address生成に転用することがよくあります。GPUで組まれたものを使うと、汎用CPUに比べてはるかに速く計算できるのです。

vanity addressを見つけるもう1つの方法は、vanityマイナープールに依頼することです。例えば、<http://vanitypool.appspot.com>[Vanityプール]です。これはGPUハードウェアを使ってvanity addressを見つけ出すことで儲けを出そうとしている集団です。少ない費用(0.01bitcoin、この執筆段階では約\$5)でEugeniaは7文字のパターンを持つvanity addressの探索を外注に出すことができ、数時間後に結果を得ることができるようになります。

vanity addressの生成は、1つずつを確認していく総当たり方式です。1個1個ランダムにキーを作ってみて、望んだパターンに合っているか確認します。vanity addressマイナーは"vanityマイナー"の例で、C++で書かれたvanity addressを探すプログラムです。例では[\[alt\\_libraries\]](#)で紹介した libbitcoinライブラリを使っています。

#### Example 8. vanity addressマイナー

```
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine);
// Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
```



```

bool match_found(const std::string& address);

int main()
{
    // random_device on Linux uses "/dev/urandom"
    // CAUTION: Depending on implementation this RNG may not be secure enough!
    // Do not use vanity keys generated by this example in production
    std::random_device random;
    std::default_random_engine engine(random());

    // Loop continuously...
    while (true)
    {
        // Generate a random secret.
        bc::ec_secret secret = random_secret(engine);
        // Get the address.
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_hex(secret) << std::endl;
            return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value...
    for (uint8_t& byte: secret)
        byte = engine() % std::numeric_limits<uint8_t>::max();
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr;
    bc::set_public_key(payaddr, pubkey);
}

```

```

// Return encoded form.
return payaddr.encoded();
}

bool match_found(const std::string& address)
{
    auto addr_it = address.begin();
    // Loop through the search string comparing it to the lower case
    // character of the supplied address.
    for (auto it = search.begin(); it != search.end(); ++it, ++addr_it)
        if (*it != std::tolower(*addr_it))
            return false;
    // Reached end of search string, so address matches.
    return true;
}

```

#### NOTE

上記の例コードでは `std::random_device` を使用しており、これは裏で動作しているオペレーティングシステムによって提供されている暗号的に安全な乱数生成器(CSRNG)の値を反映しています。LinuxのようなUNIX-likeなオペレーティングシステムの場合だとこれは `/dev/urandom` から乱数を取得しています。ここで使われている乱数生成器はデモ目的のものであり、十分なセキュリティを持ったように実装されていないので商用レベルのクオリティを持ったBitcoinキーを生成するには適切ではありません。

この例コードでは、Cコンパイラとlibbitcoinライブラリを使ってコンパイルする必要があります。例コードを動作させるには、`vanity-miner++` 実行ファイルを引数なしで実行([vanity-minerコード例のコンパイルと実行参照](#))し、実行すると"1kid"から始まるvanity addressを見つけ始めます。

## Example 9. vanity-minerコード例のコンパイルと実行

```
$ # Compile the code with g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the example
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Run it again for a different result
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Use "time" to see how long it takes to find a result
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real    0m8.868s
user    0m8.828s
sys    0m0.035s
```

例コードは3文字のパターン"kid"と合うBitcoinアドレスを探すのに数秒かかります。これはUnixコマンド `time` で計測したものです。ソースコードの中にある `search` パターンを変えて、4文字または5文字パターンにするとどれくらい処理に時間がかかるようになるかやってみてください！

### Vanity addressのセキュリティ

`vanity` `address`はセキュリティを増す方向にも下げる方向にも働きます。まさに諸刃の剣なのです。セキュリティを改善する方向に使われるとすると、特色があり見て分かりやすいアドレスであるため、悪意ある者があなたのお客さんをだまし自身のアドレスに支払いをさせることが難しくなります。他方セキュリティを下げる方向に使われるとすると、`vanity address`では誰でも似たアドレスを作ることができるため、似たアドレスを使ってお客さんをだますこともできてしまいます。

Eugenialは、ランダムに生成されたBitcoinアドレス(例えば、1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy)を使って寄付を募ることもできます。また、1Kidsから始まる`vanity address`を作って、区別しやすいアドレスにすることもできます。

どちらの場合でも、1つだけのアドレス(寄付者ごとに分けられたアドレスではなく)を使うことのリスクの1つは、侵入者があなたのウェブサイトへ侵入しBitcoinアドレスを侵入者のBitcoinアドレスに置き換えてしまうことです。もし寄付用のBitcoinアドレスをすでに多くの場所に貼り出しているとしたら、寄付者は寄付をする前に、前にウェブサイトやメール、チラシで見たBitcoinアドレスと同じであることを確認するかもしれません。+1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy+のようなランダムなBitcoinアドレスを使っている場合、大方の人はおそらく最初の数文字、"1J7mdg"だけを見てBitcoinアドレスが合っているか判断するでしょう。`vanity address`

を使っていると、見た目が似ているBitcoinアドレスですりかえようとしている誰かが、最初の数文字だけ合っているBitcoinアドレスをすばやく作ることができてしまいます。

Table 13. ランダムなアドレスに先頭が一致するvanity addressの生成

|                        |                                    |
|------------------------|------------------------------------|
| オリジナルのランダムなアドレス        | 1J7mdg5rbQyUHENYdx39WVVK7fsLpEoXZy |
| <b>Vanity (4文字が一致)</b> | 1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy |
| <b>Vanity (5文字が一致)</b> | 1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n |
| <b>Vanity (6文字が一致)</b> | 1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX |

vanity addressはセキュリティを向上させるのでしょうか？Eugeniaがvanity address 1Kids33q44erFfpeXrmDSz7zEqG2FesZEN を作ったとすると、人々はvanityパターンの単語<em>と次の数文字</em>だけを見て正しいかどうかを見ます。例えば、"1Kids33"だけです。悪意ある者は最初の6文字か8文字だけ合っているvanity addressを作りますが、2文字多く一致しているvanity addressを生成する労力はEugeniaが4文字のvanity addressを作るために使った労力の3,364倍(58 &#x00D7; 58)です。本質的に、Eugeniaがつぎ込んだ(またはvanity poolにアドレス生成を頼んだ)労力が多ければ、この悪意ある者はさらに長いvanityパターンの作成を"強いられる"こととなります。もしEugeniaが8文字のvanity addressの生成を頼んでいたとすると、この悪意ある者は10文字のvanity addressを作らなければいけません。これはパーソナルコンピュータ上では実行できずカスタマイズされたvanity-mining rig やvanity poolでさえ高価になってしまいます。特にもし悪意ある者が詐欺を行うことで得られる報酬がvanity addressの生成コストをカバーするほど高くないなら、Eugeniaには入手可能なアドレスでもこの攻撃者には入手不可能になります。

## ペーパーウォレット

ペーパーウォレットは秘密鍵を紙の上に印刷したものです。よくペーパーウォレットは利便性のため秘密鍵に対応するBitcoinアドレスも含んでいます。しかし、Bitcoinアドレスは秘密鍵から導出できるためこれは必須ではありません。ペーパーウォレットはバックアップやオフラインBitcoinストレージを作るとても効率的な方法で、"コールドストレージ"とも呼ばれています。ペーパーウォレットはハードドライブの破損、盗難、また間違っってデータを削除してしまった場合などによるキーの紛失に対するバックアップとして機能します。もしペーパーウォレットのキーがオフラインで生成されてコンピュータ上に保存されていないとすると、ペーパーウォレットはハッカーやキーロガー、その他のオンライン上の脅威などに対する安全性が増す"コールドストレージ"として機能します。

ペーパーウォレットにはいろいろな形、大きさ、デザインがありますが、基本的に秘密鍵とBitcoinアドレスが紙に印刷されているだけのものです。[ペーパーウォレットの最もシンプルな形-Bitcoinアドレスと秘密鍵の印刷](#)は最も簡単なペーパーウォレットを示しています。

Table 14. ペーパーウォレットの最もシンプルな形-Bitcoinアドレスと秘密鍵の印刷

|                                    |  |
|------------------------------------|--|
| 公開アドレス                             | 秘密鍵 (WIF)  |
| 1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpbk<br>eyhfsYB1Jcn |

ペーパーウォレットは [bitaddress.org](http://bitaddress.org) でのクライアント側 JavaScriptツールなどを使って簡単に作ることができます。このページには、インターネットに接続していなくてもペーパーウォレットが作れるコードが含まれています。それを使うために、HTMLページをローカルドライブ、または外部USBフラッシュドライブに保存してください。インターネットから切り離された状態で、ブラウザで保存したHTMLページを開いてみてください。より最適な状況を作り出すためには、CD-ROMで起動できるLinux OSのようなもっと簡素なオペレーティングシステムを使って起動し直してください。ツールを使って生成したどんなキーでもUSBケーブル(無線ではなく)で繋がれたローカルプリンタで印刷することができます。これにより、オンラインから切り離されたペーパーウォレットを作ることができます。これらのペーパーウォレットを耐火金庫に入れ、bitcoinをペーパーウォレット上のBitcoinアドレスに"送り"ます。ペーパーウォレットはとてもシンプルですが、極めて効果的な"コールドストレージ"です。



Figure 14. [bitaddress.org](http://bitaddress.org)から持ってきたシンプルなペーパーウォレットの例

簡単なペーパーウォレットの不利な点は、盗難される可能性があることです。盗難者は紙を盗む、またはペーパーウォレットの写真を撮ることでこれらのキーに紐づいているbitcoinをコントロールできるようになります。より洗練されたペーパーウォレットストレージはBIP0038暗号化秘密鍵を使う方法です。ペーパーウォレットに印刷されたキーは、所有者が記憶しているパスフレーズによって守られています。パスフレーズなしでは、暗号化されたキーを使うことはできません。暗号化秘密鍵を使ったペーパーウォレットは単なるパスフレーズで保護されたウォレットよりも安全です。というのは、オンラインに晒されることがないということと、金庫またはその他の安全なストレージから物理的に取り出さなければいけないということがあるためです。 [bitaddress.org](http://bitaddress.org)から持ってきた暗号化されたペーパーウォレットの例。パスフレーズは"test"。図はbitaddress.org上で作られた暗号化秘密鍵(BIP0038)によるペーパーウォレットを示しています。



Figure 15. bitaddress.orgから持ってきた暗号化されたペーパーウォレットの例。パスフレーズは"test"。

### WARNING

何回かペーパーウォレットに資金を預けることはできますが、ペーパーウォレットから引き出すときは全ての資金を一度に引き出すべきです。これは、資金のロックを解除して使用するプロセスでいくつかのウォレットはおつり用のアドレスを生成するかもしれないからです。さらに、もしトランザクションに署名するために使ったコンピュータに脆弱性があった場合、秘密鍵が漏洩してしまうかもしれません。ペーパーウォレットの残高全てを一度に使うことによってキーが漏洩してしまうリスクを減らすことになります。もし小さい額だけ必要なのであれば、同じトランザクション内で新しいペーパーウォレットに残りの資金を送ってください。

ペーパーウォレットは多くのデザイン、大きさがあり、また多くの異なる特徴をそれぞれ持っています。いくつかはギフトとして使われることを想定したものであり、クリスマスや新年など季節ごとのテーマを持ったデザインが施されています。また、他のいくつかは銀行の格納庫、または金庫に置くことを想定されたもの、削るスクラッチがついたものなどがあります。図 `<xref linkend="paper_wallet_bpw" xrefstyle="select: labelnumber"/>` から 図 `<xref linkend="paper_wallet_spw" xrefstyle="select: labelnumber"/>` はいろいろな種類のペーパーウォレットを紹介しています。

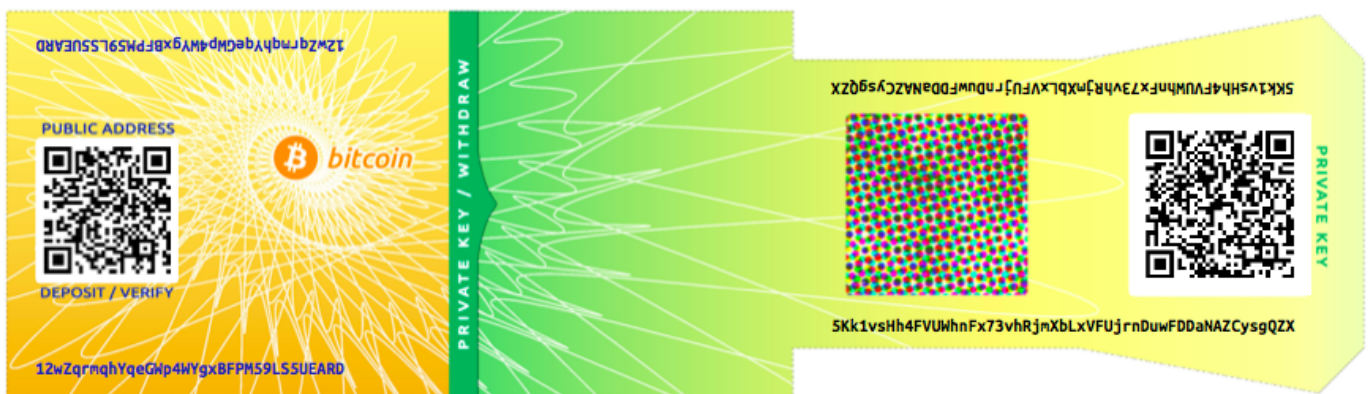


Figure 16. 折りたたみ部分に秘密鍵が置かれた bitcoinpaperwallet.com ペーパーウォレット例



Figure 17. 秘密鍵部分が覆われて見えないようになっている *bitcoinpaperwallet.com* ペーパーウォレット

他のデザインでは、チケットのように切り離し可能になっていて、火事や洪水など自然災害があってもいいようにキーとBitcoinアドレスの複数のコピーが持てるようになっているものもあります。



Figure 18. バックアップ用の“切り離し部分”にキーのコピーがあるペーパーウォレットの例

# トランザクション

## イントロダクション

トランザクションはBitcoinシステムの中で最も重要な部分です。他のものは全て、トランザクションが作成されBitcoinネットワークを伝搬し検証そして最後にグローバルなトランザクション元帳(ブロックチェーン)に追加される、という流れを支えるように作られています。トランザクションはBitcoinシステムに参加している者の間での価値の転送をデジタル化したデータの集合体です。それぞれのトランザクションはBitcoinのブロックチェーン(グローバルな複式簿記元帳)の中にある誰でも見ることができる取引です。

この章では、いろいろな形式のトランザクションや、トランザクションは何を含んでいるのか、トランザクションはどのように作られるのか、またどのように検証されるのか、どのように永続的な記録の一部になるのか、ということを説明していきます。

## トランザクションのライフサイクル

トランザクションのライフサイクルはまず *origination* と呼ばれるトランザクションの生成から始まります。このときトランザクションは1つまたは複数の署名で署名されます。これら署名はトランザクションによって参照されている資金を使う許可を意味します。署名後にトランザクションはBitcoinネットワークにブロードキャストされます。それぞれのネットワークノード(Bitcoinネットワークへの参加者)はトランザクションを検証し、(ほぼ)全てのノードに行き渡るまでトランザクションがどんどんBitcoinネットワーク内を伝搬していきます。最後に、トランザクションはマイニングノードによって検証され、ブロックチェーンの中のブロックに記録されます。

一度ブロックチェーンに記録され十分なブロックによって承認される(confirmation)と、トランザクションはBitcoin元帳の永続的な一部となり、全ての参加者によって有効なものとして受け入れられます。トランザクションによって新しい所有者に割り当てられた資金は次に新しいトランザクションの中で使用することができ、所有者が変わって、再びトランザクションのライフサイクルが始まることとなります。

### トランザクションの生成

小切手と同じやり方を使ってトランザクションを考えてみると考えやすいです。小切手と同じように、トランザクションはお金を転送するという意思を表す道具で、実際に使われるまで目に見える形にはなりません。また小切手と同じように、トランザクションの発行人はトランザクションに署名している人である必要はありません。

トランザクションを作った人が仮に口座の正規署名者でなかったとしても、オンラインでもオフラインでもトランザクションを作ることができます。例えば、口座へのアクセス権を持った事務員はCEOによる署名が入った小切手を作ることができます。同様に口座へのアクセス権を持った事務員はBitcoinトランザクションを作ることができ、トランザクションを有効にするデジタル署名をトランザクションに適用することができます。小切手は資金がある特定の口座を参照している一方、Bitcoinトランザクションは口座ではなく以前実行された特定のトランザクションを参照することとなります。

トランザクションが一度作られると、資金の所有者(または所有者たち)によって署名されます。もしトランザクションが正規の形式を保持しつつ署名されていれば、このトランザクションは有効になり資金の転送を実行するために必要な全ての情報を含んでいることとなります。最終的に、有効なトランザクションはBitcoinネットワークを伝搬していき、マイナーによって公的な元帳(ブロックチェーン)に格納されます。



## Bitcoinネットワークへのトランザクションのブロードキャスト

まず、トランザクションはブロックチェーンに記録されるためにBitcoinネットワークに放出される必要があります。Bitcoinトランザクションは300から400byteのデータを持ち、数万ものBitcoinノードのうちどれでもいいので1つに辿り着かなければいけません。2つ以上のBitcoinノードにブロードキャストするので、送信者はBitcoinノードを信用する必要はありません。ノードは送信者を信用する必要はなく、また送信者が誰なのかを特定する必要もありません。トランザクションは署名されており、また一切の機密情報(秘密鍵や証明書)も含まれていないため、いかなる転送手段を使って公にブロードキャストしても構いません。例えば、暗号化されたネットワークでしかデータの転送ができないセンシティブな情報が含まれているクレジットカードのトランザクションと違って、Bitcoinのトランザクションはどんなネットワークを通してでも送ることができます。トランザクションがどれか1つのBitcoinノードにたどり着くことができるのであれば、最初の転送方法はいつでもよいのです。

このためBitcoinトランザクションは、WiFiやBluetooth、NFC、光通信、バーコード、Bitcoinアドレスのウェブフォームへのコピペのような安全でないネットワーク、を通してでも転送することができます。安全でない極端な場合として、パケット通信(アマチュア無線)や衛星中継、バースト転送を用いた短波通信、周波数ホッピングなどのスペクトラム拡散(無線通信)などがあります。Bitcoinトランザクションは、絵文字としてでさえ表現でき、公的なフォーラムへの投稿、またテキストメッセージ、Skypeチャットメッセージとして送ることもできます。Bitcoinはお金をデータの形に変え、これにより誰もトランザクションを作成や実行を阻止できないようにしたのです。

## Bitcoinネットワーク上でのトランザクションの伝搬

一度トランザクションがBitcoinネットワークに接続されたノードに送られると、このトランザクションは送られたノードで有効なものか検証されます。有効なものだと確認されると、そのノードは接続している他のノードにこのトランザクションを伝搬します。同時に、成功メッセージが発行ノードに返却されます。もしこのトランザクションが無効なものであればノードはこのトランザクションを棄却し、同時に棄却メッセージを発行ノードに返却します。

Bitcoinネットワークはpeer-to-peerネットワークであり、それぞれのBitcoinノードは数個のノードに接続されています。この数個のノードはpeer-to-peerプロトコルに従ってノードを起動したときに発見したノードです。全Bitcoinネットワークは緩やかに接続されたメッシュであり、固定されたトポロジーや構造を持つことなく全てのノードは平等に扱われます。トランザクションやブロックを含んだメッセージはそれぞれのノードから接続されている他のピアに伝搬します。このプロセスは"flooding"と呼ばれています。有効だと確認された新しいトランザクションは接続された全てのノード(隣接ノード)に送られ、それぞれの隣接ノードはまた全ての隣接ノードにこのトランザクションを送ります。このような方法で、全ての接続されたノードがこのトランザクションを受け取るまで波紋のようにBitcoinネットワーク内を伝わっていき数秒以内に全体に広がっていきます。

Bitcoinネットワークは、攻撃に強く、また効率的なルールに従って全てのノードにトランザクションとブロックを伝搬できるように設計されています。厄介なBitcoinシステムに対するDOS攻撃のような強制的なデータの送りつけを防ぐために、全てのノードはトランザクションを次のノードに送る前に全てのトランザクションが有効なものか確認しています。このため、おかしなトランザクションが次のノードに送られることはありません。この方法について[[tx\\_verification](#)]で詳細に説明します。

## トランザクションの構造

トランザクションは、資金源(     インプット     と呼ばれる)から送り先(     アウトプット     と呼ばれる)

データの集合体)への価値の転送を記号化した  
 データの集合体  
 です。トランザクションのインプットやアウトプットは、アカウントやIDなど、個人を特定できる情報と結び  
 ついているわけではありません。代わりにこれらを、所有者だけがもっている秘密鍵でロックされているbit  
 coinの固まりとして考えるべきです。トランザクションはトランザクションの構造に示すようないくつかのフ  
 ィールドを含んでいます。

Table 1. トランザクションの構造

| サイズ               | フィールド名         | 説明                         |
|-------------------|----------------|----------------------------|
| 4 byte            | Version        | このトランザクションがどのルールに従っているかを指定 |
| 1-9 byte (VarInt) | Input Counter  | いくつのインプットが含まれているか          |
| 可変サイズ             | Inputs         | 1つまたは複数のトランザクションインプット      |
| 1-9 byte (VarInt) | Output Counter | いくつのアウトプットが含まれているか         |
| 可変サイズ             | Outputs        | 1つまたは複数のトランザクションアウトプット     |
| 4 byte            | Locktime       | Unixタイムスタンプ、またはブロック高       |

## トランザクションLocktime

locktimeはトランザクションが検証されたり、Bitcoinネットワーク内でリレーされたり、またブロッ  
 クチェーンに追加されたりした最も早い時刻です。これは、リファレンス実装であるBitcoin  
 Coreの中でnLockTimeとしても知られていたものです。ほとんどのトランザクションではすぐに伝搬  
 されたことを表すためにlocktimeが0に設定されます。もしlocktimeが0でないかまたは500,000,000よ  
 り下になっているときはlocktimeをブロック高として解釈し、このブロック高より前のブロックではこ  
 のトランザクションがブロックチェーンに取り込まれていないということを意味します。もし500,000,  
 000よりも大きいときはlocktimeをUNIX Epochタイムスタンプ(1970/1/1からの秒数  
 )として解釈し、この時刻よりも前にこのトランザクションが有効ではなかったということを意味しま  
 す。locktimeが将来のブロックまたは時刻になっている場合は発行システムによってトランザクション  
 が保持されていなければならず、トランザクションが有効になってからのみBitcoinネットワークに送  
 信されなければいけません。locktimeは先日付小切手の日付のようなものです。

## トランザクションアウトプットとインプット

Bitcoinトランザクションの基本的な構成要素は、未使用トランザクションアウトプット  
 または  
 UTXO(unsent transaction output)です。UTXOは、特定の所有者にロックされた分割不可能な  
 bitcoinの固まりです。これはブロックチェーンに記録されており、Bitcoinネットワーク全体によって通貨の  
 単位として捉えられているものです。Bitcoinネットワークは全ての利用可能(未使用)なUTXOを追跡しており  
 、現在数百万に達するほどの量があります。ユーザがbitcoinを受け取る時はいつでも、UTXOとしてブロッ  
 クチェーンに記録されます。このため、ユーザのbitcoinは数百個のトランザクションまたは数百個のブロッ

クの中にUTXOとして散り散りな状態になってしまっているかもしれません。実際には、Bitcoinアドレスまたは口座の残高として記録されている訳ではないのです。あるのはただ散り散りになり特定の所有者に利用が制限されたUTXOだけです。ユーザのbitcoin残高という概念は、ウォレットによって作り上げられたものにすぎません。ウォレットはブロックチェーンをスキャンしてユーザに属している全てのUTXOを掻き集め残高を計算しているのです。

**TIP** Bitcoinに口座も残高もありません。あるのは単にブロックチェーンの中に散らばった未使用トランザクションアウトプット (UTXO)だけです。

UTXOは satoshi を単位とした任意の値を持つことができます。ドルがセントというさらに下の2桁の十進数を持つように、bitcoinはsatoshiという8桁の十進数を持ちます。UTXOは任意の値ですが、一度作られるとコインのように2つに切ることができません。別の言い方をすると、もし20bitcoinのUTXOを持っていて1bitcoinだけ使いたいとすると、トランザクションは20bitcoinのUTXOを消費しなければならないため2つのアウトプットを作らなければいけません。1つは支払った1bitcoin、もう1つはあなたのウォレットに戻ってくるおつりの19bitcoinです。結果として、ほとんどのBitcoinトランザクションはおつりを生成します。

\$1.50の飲み物を買う人を想像してみましょう。彼女の財布から\$1.50になるコインと紙幣の組み合わせを探しだします。もし財布にあるならおつりのいらぬちょうど金額(1ドル札と2つの25セントコイン、または6つの25セントコイン)、無理であれば5ドル札のような大きな単位の紙幣を選びます。もし多くのお金を持っているとすると、\$5をショップオーナーに支払い\$3.50のおつりが帰ってくると考えるでしょう。彼女はこの\$3.50を財布に戻し、将来の買い物のときに使うことができます。

同様に、Bitcoinトランザクションはユーザが使用可能なUTXOから作られます。ユーザはUTXOを半分に割ることはできません。ウォレットは通常ユーザの利用可能なUTXOを選び、トランザクションに必要な金額以上になるように組み合わせます。

実用上は、Bitcoinアプリケーションは購入額を満たすためにいくつかのやり方を使うことができます。いくつかのより小さい単位の額を組み合わせる、おつりがなくなきっちりした金額を選ぶ、またはトランザクションに必要な金額より大きい額を使っておつりを作るなどです。このやり方はウォレットのほうで自動的に実行され、ユーザには見えないようになっています。関係してくるとしたら、生トランザクションをUTXOからプログラムを通して手で構成する場合だけです。

トランザクションによって消費されたUTXOはトランザクションインプットと呼ばれ、トランザクションによって作られたUTXOをトランザクションアウトプットと呼びます。UTXOを消費し作成するトランザクション連鎖の中で、bitcoinの固まりはある所有者からある所有者に移っていきます。トランザクションは現在の所有者の署名を使って解錠されることでUTXOを消費します。

インプットとアウトプットのチェーンの例外は、 *coinbase* トランザクションと呼ばれる特殊なトランザクションです。これは、それぞれのブロックの一番最初のトランザクションです。このトランザクションはマイニングに"勝った"マイナーによってブロックの一番最初に置かれ、マイニングに対する報酬としてマイナーにbitcoinが支払われるトランザクションとなります。このようにしてマイニングプロセスを通してBitcoinのお金が供給されていきます。詳細は[[ch8](#)]で説明します。

**TIP** トランザクションのチェーンの一番最初には何が来るでしょうか？インプットそれともアウトプット、鶏それとも卵？厳密に言って、アウトプットが最初に来ます。なぜなら、新しいbitcoinを生成するcoinbaseトランザクションはインプットを持っておらず、何も無いところからアウトプットを作るからです。

## トランザクションアウトプット

全てのBitcoinトランザクションはアウトプットを作ります。このアウトプットはBitcoin元帳上に記録され、ほとんど全てのアウトプット(1つの例外を除いて。[データアウトプット\(OP\\_RETURN\)](#)参照)は、未使用トランザクションアウトプット または UTXO と呼ばれる使用可能なbitcoinの固まりを作ります。UTXOはBitcoinネットワーク全体によって認識されており、所有者が将来の取引でこれを使うことができます。誰かにbitcoinを送ることは、送り先のBitcoinアドレスと紐づけられた未使用トランザクションアウトプット(UTXO)を作り出すことです。このUTXOは受信者が使うことが可能なトランザクションアウトプットです。

UTXOは全てのフルノード Bitcoinクライアントによって追跡され、 *UTXOセット* または *UTXOプール* と呼ばれるメモリに持っているデータベースで管理されています。そして、新しいトランザクションはUTXOセットにある1つまたは複数のアウトプットを消費(使用)することになります。

トランザクションアウトプットは以下2つの部分で成り立っています。

- bitcoinの最小単位である *satoshi* 単位で表されたbitcoin金額
- アウトプットと使用するにあたって満たさなければいけない条件である"解除条件(encumbrance)"として知られている *locking script*

*locking script*の中で使われているトランザクションScript言語の詳細については[トランザクションscript](#)と[Script言語](#)で説明します。[トランザクションアウトプットの構造](#)はトランザクションアウトプットの構造を示しています。

Table 2. トランザクションアウトプットの構造

| サイズ               | フィールド名              | 説明                                  |
|-------------------|---------------------|-------------------------------------|
| 8 byte            | Amount              | satoshi単位( $10^8$ bitcoin)のbitcoin額 |
| 1-9 byte (VarInt) | Locking-Script Size | 次に続くlocking scriptのバイト長             |
| 可変サイズ             | Locking-Script      | アウトプットを使用するために必要な条件を定義したscript      |

あるBitcoinアドレスに関連したUTXOを見つけ出す [blockchain.info](#) APIを呼び出すスクリプトで[blockchain.info](#) APIを使って特定のBitcoinアドレスの未使用アウトプット(UTXO)を調べています。

Example 1. あるBitcoinアドレスに関連したUTXOを見つけ出す *blockchain.info* APIを呼び出すスクリプト

```
# get unspent outputs from blockchain API

import json
import requests

# example address
address = '1Dorian4RoXcnBv9hnQ4Y2C1an6NJ4UrxX'

# The API URL is https://blockchain.info/unspent?active=<address>
# It returns a JSON object with a list "unspent_outputs", containing UTXO, like this:
#{ "unspent_outputs":[
#   {
#     "tx_hash":"ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167",
#     "tx_index":51919767,
#     "tx_output_n": 1,
#     "script":"76a9148c7e252f8d64b0b6e313985915110fcfefcf4a2d88ac",
#     "value": 8000000,
#     "value_hex": "7a1200",
#     "confirmations":28691
#   },
#   ...
# ]}

resp = requests.get('https://blockchain.info/unspent?active=%s' % address)
utxo_set = json.loads(resp.text)["unspent_outputs"]

for utxo in utxo_set:
    print "%s:%d - %ld Satoshis" % (utxo['tx_hash'], utxo['tx_output_n'],
    utxo['value'])
```

このスクリプトを実行すると、"トランザクションID":"特定の未使用トランザクションアウトプット(UTXO)のインデックス" - "UTXOの satoshi 単位での金額" という形のリストが表示されます。 [get-utxo.py スクリプトの実行](#)のアウトプットにlocking scriptは表示されていません。

## Example 2. get-utxo.py スクリプトの実行

```
$ python get-utxo.py
ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167:1 - 8000000 Satoshi
6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 - 16050000
Satoshi
74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0:0 - 5000000 Satoshi
b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4:0 - 10000000
Satoshi
...
```

### 使用条件(解除条件)

トランザクションアウトプットはbitcoin(satoshi単位で表された)を、特定の解除条件(*encumbrance*)またはbitcoinを使うにあたって満たさなければいけない条件を定義したlocking scriptと関連づけています。多くの場合、locking scriptは特定のBitcoinアドレスにアウトプットをロックし、これにより所有権が新しい所有者に移ります。AliceがBobのカフェにコーヒー代を支払ったとき、彼女のトランザクションのアウトプットにはカフェのBitcoinアドレスにロックされた0.015bitcoinアウトプットが含まれていました。この0.015bitcoinアウトプットはブロックチェーンに記録され、カフェのBitcoinアドレスに紐づいた未使用トランザクションアウトプットセットの一部になったのです。Bobがこの0.015bitcoinアウトプットを支払いに使うときに、彼のトランザクションはBobの秘密鍵による署名を含むunlocking scriptを提示することでこの0.015bitcoinアウトプットのロックを外すのです。

### トランザクションインプット

簡単に言って、トランザクションインプットはUTXOへのポインタです。トランザクションインプットは、トランザクションハッシュとUTXOが記録されているブロックチェーン内の場所を示すシーケンス番号を使って特定のUTXOを指定します。UTXOを使うために、トランザクションインプットはunlocking scriptというUTXOのロックを解除するscriptも持っています。unlocking scriptは通常locking scriptの中にあるBitcoinアドレスの所有権を証明している署名です。

ユーザが支払いをするとき、ウォレットは使用可能なUTXOを選びトランザクションを構成します。例えば、0.015bitcoinの支払いをするのであれば、ウォレットは0.01bitcoinのUTXOと0.005bitcoinのUTXOを選び支払いに必要な金額になるようにするかもしれません。

支払いに総額いくらが必要となるかを計算するためのスクリプトでは"貪欲(greedy)"アルゴリズムを使うことで、ある金額を満たすようにUTXOを選ぶ例を示しています。この例では、UTXOをあらかじめ決められた配列で与えています。しかし、現実ではUTXOはBitcoin CoreのRPC APIを使って集めてくるか、またはあるBitcoinアドレスに関連したUTXOを見つけ出す [blockchain.info](https://blockchain.info) APIを呼び出すスクリプトにあるようなサードパーティAPIを使って集めてきます。

### Example 3. 支払いに総額いくらが必要となるかを計算するためのスクリプト

```

# Selects outputs from a UTXO list using a greedy algorithm.

from sys import argv

class OutputInfo:

    def __init__(self, tx_hash, tx_index, value):
        self.tx_hash = tx_hash
        self.tx_index = tx_index
        self.value = value

    def __repr__(self):
        return "<%s:%s with %s Satoshis>" % (self.tx_hash, self.tx_index,
                                             self.value)

# Select optimal outputs for a send from unspent outputs list.
# Returns output list and remaining change to be sent to
# a change address.
def select_outputs_greedy(unspent, min_value):
    # Fail if empty.
    if not unspent:
        return None
    # Partition into 2 lists.
    lessers = [utxo for utxo in unspent if utxo.value < min_value]
    greater = [utxo for utxo in unspent if utxo.value >= min_value]
    key_func = lambda utxo: utxo.value
    if greater:
        # Not-empty. Find the smallest greater.
        min_greater = min(greater)
        change = min_greater.value - min_value
        return [min_greater], change
    # Not found in greater. Try several lessers instead.
    # Rearrange them from biggest to smallest. We want to use the least
    # amount of inputs as possible.
    lessers.sort(key=key_func, reverse=True)
    result = []
    accum = 0
    for utxo in lessers:
        result.append(utxo)
        accum += utxo.value
        if accum >= min_value:
            change = accum - min_value
            return result, "Change: %d Satoshis" % change
    # No results found.
    return None, 0

def main():
    unspent = [

```

```

OutputInfo("ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167", 1,
8000000),

OutputInfo("6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf", 0,
16050000),

OutputInfo("b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4", 0,
10000000),

OutputInfo("7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1", 0,
25000000),

OutputInfo("55ea01bd7e9afd3d3ab9790199e777d62a0709cf0725e80a7350fdb22d7b8ec6", 17,
5470541),

OutputInfo("12b6a7934c1df821945ee9ee3b3326d07ca7a65fd6416ea44ce8c3db0c078c64", 0,
10000000),

OutputInfo("7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818", 0,
16100000),
]

if len(argv) > 1:
    target = long(argv[1])
else:
    target = 55000000

print "For transaction amount %d Satoshis (%f bitcoin) use: " % (target,
target/10.0**8)
print select_outputs_greedy(unspent, target)

if __name__ == "__main__":
    main()

```

もし `select-utxo.py` スクリプトをパラメータなしで実行すると、55,000,000satoshi(0.55bitcoin)の支払いに対してUTXOの組み合わせ(とおつりも)を構成しようとしています。パラメータとして支払額を指定すると、スクリプトは指定した支払額を満たすようにUTXOを選びます。[select-utxo.py](#) [スクリプトの実行](#)では、0.5bitcoinまたは50,000,000satoshiの支払い額を指定してスクリプトを実行しています。



#### Example 4. select-utxo.py スクリプトの実行

```
$ python select-utxo.py 50000000
For transaction amount 50000000 Satoshis (0.500000 bitcoin) use:
([<7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1:0 with 25000000 Satoshis>, <7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818:0 with 16100000 Satoshis>, <6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 with 16050000 Satoshis>], 'Change: 7150000 Satoshis')
```

一度UTXOが選ばれると、ウォレットはそれぞれのUTXOに対して署名を含んでいるunlocking scriptを作ります。このunlocking scriptによってlocking scriptの条件を満たすためUTXOが使用可能になります。ウォレットはこれらのUTXOへの参照とunlocking scriptをインプットとしてトランザクションに追加します。トランザクションインプットの構造はトランザクションインプットの構造を示しています。

Table 3. トランザクションインプットの構造

| サイズ               | フィールド名                | 説明  |
|-------------------|-----------------------|---|
| 32 byte           | Transaction Hash      | 使われるUTXOを含むトランザクションハッシュ                   |
| 4 byte            | Output Index          | 使われるUTXOのトランザクション内インデックス、一番最初のアウトプットの場合は0 |
| 1-9 byte (VarInt) | Unlocking-Script Size | unlocking-scriptのバイト長                     |
| 可変サイズ             | Unlocking-Script      | UTXOのlocking scriptを満たすscript             |
| 4 byte            | Sequence Number       | 現在トランザクション置き換えは使用不可になっている、0xFFFFFFFF に設定  |

#### NOTE

sequence numberは、locktimeが示す時間またはブロック高に到達する前にトランザクションを書き換えるために使われますが、現在この機能は使用不可になっています。ほとんどのトランザクションではこの値を整数最大値(0xFFFFFFFF)に設定し、この場合Bitcoinネットワークで無視されます。もしトランザクションが0ではないlocktimeを持っているとすると、locktimeを有効にするために少なくともこのトランザクションのインプットのうちの1つが0xFFFFFFFFよりも小さいsequence numberを持たなければいけないのです。

#### トランザクション手数料

ほとんどのトランザクションはトランザクション手数料を含んでいて、この手数料はBitcoinマイナーに与えられます。マイニングと手数料、マイナーによって集められた報酬についての詳細は[ch8]で説明することに

します。この節では、どのようにしてトランザクション手数料がトランザクションに含められるかを説明します。ほとんどのウォレットはトランザクション手数料を自動的に計算しトランザクションに含めます。しかし、もしトランザクションをプログラムを通して構築する、またはコマンドラインを使って構築する場合は、手動でこれらの手数料をトランザクションに含めなければいけません。

トランザクション手数料はトランザクションを次のブロックに含める(マイニングする)ことのインセンティブとして働き、また少額でも手数料をトランザクションに入れなければいけないため"スパム"トランザクションやBitcoinシステムを悪用することに対する逆のインセンティブとして働きます。トランザクション手数料はトランザクションを記録しているブロックをマイニングしたマイナーによって集められます。

トランザクション手数料はトランザクションのデータサイズ(KB)に基づいて計算され、いくらの支払いをしたかによって手数料は決まりません。トランザクション手数料はBitcoinネットワーク内での市場原理に基づいて決められます。マイナーごとにどのトランザクションを優先的に選ぶかの判断条件は違っており、この判断条件には手数料の大きさも含まれます。手数料が含まれていないトランザクションも状況によってはマイナーに選ばれるかもしれません。しかし、トランザクション手数料はマイナーによって処理される優先順位に影響し、十分な手数料をもっているトランザクションがマイニングされている次のブロックに含まれる可能性が高くなり、一方十分な手数料を持っていないか、手数料がないトランザクションはブロックに取り込まれることが遅れてしまいます。数ブロック後に取り込まれる、そもそも処理されないということになるかもしれません。トランザクション手数料は必須ではなく、ときどき手数料がないトランザクションもマイナーに処理されますが、トランザクション手数料を含めることは処理の優先順位をあげることに繋がるのです。

時間とともに、トランザクション手数料の計算方法やトランザクションの優先順位付け方法が発展してきました。最初、トランザクション手数料は固定されており、Bitcoinネットワーク全体で一定でした。次第に手数料制限は緩和され、Bitcoinネットワークのキャパシティやトランザクション量に基づく市場の力関係にトランザクション手数料が影響されるようになってきました。現在の最小トランザクション手数料はトランザクションのデータサイズ1KBあたり0.0001bitcoin、0.1ミリbitcoinに固定されており、最近1ミリbitcoinに減らされました。多くのトランザクションは1KBより小さいですが、いくつかのインプットまたはアウトプットを持っているとより大きな手数料になります。Bitcoinプロトコルの将来の改定で、ウォレットが最近のトランザクションの手数料平均値に基づき統計的に最適な手数料を決定できるようになると予想されています。

マイナーがトランザクションの優先順位付けをする際に使っている現在のアルゴリズムについては[ch8]で詳細に説明します。

## トランザクションへの手数料の追加

トランザクションのデータ構造には手数料に対するフィールドはありません。代わりに、手数料はインプットの総和とアウトプットの総和との差として暗に含められる形になっています。全てのインプットの総和から全てのアウトプットの総和を引いて残った余分な額がマイナーによって集められる手数料です。

トランザクション手数料はインプットとアウトプットの差として暗に含められています。

$$\text{Fees} = \text{Sum}(\text{Inputs}) - \text{Sum}(\text{Outputs})$$

これは、幾分トランザクションを理解する上で混乱してしまうところですが重要なポイントです。というのは、もし自身でトランザクションを構築するとしたときに、うっかり大きな額の手数料を含めないようにしないといけないためです。つまり、全てのインプットを把握しておかなければいけません。そして、必要であればおつりを送るアウトプットを作成しなければいけません。さもなければ、マイナーにとっても大きなチップをあげることになってしまうのです！

例えば、20bitcoinのUTXOを消費して1bitcoinの支払いをしようとするなら、19bitcoinのおつりがアウトプットに含まれていなければいけません。そうしないと、19bitcoinの"残り物"はトランザクション手数料とし

てカウントされてしまい、あなたのトランザクションを含むブロックをマイニングしたマイナーによって19bit  
tcoinが集められてしまうのです。

## WARNING

手動でトランザクションを構築したときにもしおつりのアウトプットを追加し忘れてしま  
うと、おつり分をトランザクション手数料として払ってしまうこととなります。"おつ  
りは不要です!"というのは、通常の支払いの感覚からすると不思議に感じるかもしれま  
せん。

再度Aliceのコーヒー代支払いの例を使って実用上どのように動作するかを見ていきましょう。Aliceは0.015b  
itcoinをコーヒー代として支払おうとしています。分かりやすくするために彼女は0.001bitcoinをトランザク  
ション手数料として含めようとしていてみましょう。これはトランザクションの総コストが0.016bitco  
inになることを意味しています。よって、彼女のウォレットは0.016bitcoinかまたはそれより多い額になるよ  
うにUTXOを集め、必要ならおつりを作らなければいけません。彼女のウォレットが0.2bitcoinのUTXOが使用  
可能だとしてみると、このUTXOを消費することになります。アウトプットとしては、Bobのカフェ店への  
支払いとして0.015bitcoinのアウトプットを作り、そして2つ目のアウトプットとして自分自身のウォレット  
に返ってくる0.184bitcoinのおつりのアウトプットを作ります。0.001bitcoinが残っていますが、これが暗に  
トランザクションに含まれているトランザクション手数料になります。

違ったシナリオを考えてみましょう。フィリピンの子供チャリティーディレクターのEugeniaは子供のために  
学校の教科書を購入するための支援金集めが完了し、全世界の人々からいただいた数千個の小さな寄付を受け  
取りました。総額にして50bitcoinです。このため、彼女のウォレットは小さな支払い(UTXO)でいっぱいにな  
ってしまいました。彼女は数百冊の学校の教科書を地元の出版社から購入したいと考えていて、支払いをbitc  
oinでするつもりでいます。

Eugeniaのウォレットは1個の大きなトランザクションを作ろうとしたため、多くの小さな額のUTXOで占め  
られているUTXOセットからUTXOを集めてこなければいけません。結果として作られるトランザクションに  
はインプットとして数百個の小さな額のUTXOと出版社に支払われるたった1個のアウトプットで構成される  
こととなります。多くのインプットを伴ったトランザクションのデータサイズは1KBよりも大きく、おそらく  
2、3KBです。結果的に、最小手数料0.0001bitcoinよりも高いトランザクション手数料が必要になります。

Eugeniaのウォレットはトランザクションのデータサイズと1KBあたりの手数料を掛け合わせて適切な手数料  
を計算することになります。多くのウォレットは大きなトランザクションに対して手数料を多めに払っていま  
す。これは、トランザクションを迅速に処理してもらうためです。高い手数料を払うのはEugeniaが多くのお  
金を使っているからではなく、トランザクションがより複雑でよりデータサイズが大きいからです。トランザ  
クション手数料の額はトランザクションのbitcoin額とは無関係なのです。

## トランザクションチェーンとOrphanトランザクション

今まで見てきたように、トランザクションはチェーンを形成します。このチェーンというのは、1つのトラン  
ザクションは前のトランザクションアウトプット(親と呼ばれる)を使い、また次のトランザクションのために  
アウトプット(子と呼ばれる)を作る、というチェーンです。ときどきトランザクションのチェーン全体(親トラン  
ザクション、子トランザクション、孫トランザクション)が一度に作られることがあります。これは親トラン  
ザクションが署名される前に、署名された有効な子トランザクションが必要な場合です。例えば、これは  
CoinJoin トランザクションの場合のテクニックに使われます。  
CoinJoinトランザクションは、複数の人のトランザクションを別のトランザクションに加えて混ぜることでプ  
ライバシーを守るために使われます。

トランザクションのチェーンはBitcoinネットワークを通して放出されたとき、順番通りにノードに届くわけではなく、もしかすると親よりも先に子が届いてしまうかもしれません。この場合、子を最初に見つけたノードは、この子が参照している親トランザクションのことはまだ知りません。子供を拒否するよりもむしろ、一時的なプールに子を置いておき、親が届くことを待ちます。親がないトランザクションのプールを *orphan* トランザクションプール と呼びます。一度親が届くと、親のUTXOを参照している *orphan* は全てプールから取り出され、再帰的に再確認されます。このとき、トランザクションのチェーン全体が *orphan* トランザクションプールからトランザクションプールに取り込まれ、ブロックに取り込まれる準備が整います。トランザクションのチェーンは多くの世代が伴ったとしてもどれだけでも長くでき、同時に送信できます。 *orphan* プールに *orphan* トランザクションを保持しておく方法を使うことで、親の到着が遅れたとしても子を放棄することなく、かつ正しい順番でトランザクションのチェーンを構築できるのです。

メモリに保持できる *orphan* トランザクションの数には制限があります。これは、BitcoinノードからのDOS攻撃を防ぐためです。制限数は、 `MAX_ORPHAN_TRANSACTIONS` というBitcoinリファレンスクライアントのソースコード内にあるパラメータで定義されています。もしプールにある *orphan* トランザクションの数が `MAX_ORPHAN_TRANSACTIONS` を越えると、ランダムに選ばれたいくつかの *orphan* トランザクションがプールから追い出され、プールにある *orphan* トランザクション数が制限以内になるように調整されます。

## トランザクションscriptとScript言語

Bitcoinクライアントはscriptを実行することでトランザクションの有効性をチェックします。UTXOにあるlocking script(解除条件)と通常署名を含んでいるunlocking scriptはこのScript言語で書かれています。トランザクションが有効かチェックされるときは、資金の使用条件を満たしているかどうかをみるためにそれぞれのインプットにあるunlocking scriptが対応したlocking scriptとともに実行されます。

今日、Bitcoinネットワークを通して処理される多くのトランザクションは"AliceがBobに支払う"というような形式になっており、Pay-to-Public-Key-Hash scriptと呼ばれるscriptに基づいています。しかし、アウトプットをロックしインプットを解錠するscriptを使うことは、プログラミング言語を通してトランザクションに無限個の条件を含められることを意味します。つまり、Bitcoinトランザクションは"AliceがBobに支払う"という形式に制限されているわけではないのです。

これは単にこのScript言語によって表現できる可能性の氷山の一角を見せているに過ぎません。この節では、Bitcoinトランザクションのscript言語の要素を説明し、どのように資金の使用に対する完全な条件を表現するのか、どのようにunlocking scriptは条件を満たすことができるのかを説明していきます。

### TIP

Bitcoinトランザクションの有効性チェックは静的なパターンに基づいているわけではなく、Script言語の実行を通して行われています。この言語はほとんど無限個の条件を表現することができます。このようにしてBitcoinは"プログラム可能な通貨"を実現しているのです。

## scriptの構築(Lock + Unlock)

Bitcoinのトランザクション有効性チェックエンジンは二種類のscriptによって成り立っています。1つはlocking script、もう1つはunlocking scriptです。

locking

scriptはアウトプットに置かれている解除条件で、将来アウトプットを使用する際に満たさなければいけない条件を指定しています。歴史的に、locking scriptは scriptPubKey と呼ばれていました。というのは、locking scriptに通常公開鍵またはBitcoinアドレスが含まれているからです。この本では、Scriptテクノロジーの可能性をより多く得るためにそれを"locking script"と呼ぶことにします。多くのBitcoinアプリケーションでは、locking scriptと呼んだものが scriptPubKey としてソースコードに出てきます。

unlocking scriptは、locking scriptによってアウトプットに置かれた条件を"解く"または満たすscriptで、アウトプットを使用できるようにします。unlocking scriptは全てのトランザクションインプットの一部であり、ほとんどの場合秘密鍵からウォレットが作り出したデジタル署名を含んでいます。歴史的に、unlocking scriptが通常デジタル署名を含んでいるためunlocking scriptは scriptSig と呼ばれています。多くのBitcoinアプリケーションのソースコードの中では、unlocking scriptを scriptSig と呼んでいます。この本では、これを"unlocking script"と呼ぶことにしています。というのは、全てのunlocking scriptが署名を含んでいなければいけないわけではなく、locking scriptの解除をするために必要なものは署名以外にもあるということに気づいてもらうためです。

全てのBitcoinクライアントはlocking scriptとunlocking scriptを一緒に実行することでトランザクションが有効であることをチェックします。トランザクションにあるそれぞれのインプットに対して、有効性チェックソフトウェアは最初にインプットによって参照されているUTXOを取得しようとします。このUTXOは、UTXOを使うときに必要な条件が定義してあるlocking scriptを含んでいます。有効性チェックソフトウェアはこのときこのUTXOの資金を使おうとしているインプットに含まれているunlocking scriptを取り出し、locking scriptとunlocking scriptを実行します。

オリジナルのBitcoinクライアントでは、unlocking scriptとlocking scriptは結合されており順番に実行されていました。セキュリティの観点からこれは2010年に変更されました。悪意あるunlocking scriptがスタックにデータをプッシュし、locking scriptを意味がないものにしてしまうという弱点が生じてしまうためです。現在の実装では、次に説明するようにunlocking scriptとlocking scriptは別々に実行され、スタックがこれらの間で転送される形で実行されるようになっています。

最初に、スタック実行エンジンを使ってunlocking scriptが実行されます。もしunlocking scriptがエラーなく(例えば、"まだ実行されていない"オペレータが何も残っていないなど)実行されると、メインスタック(代替スタックではなく)がコピーされlocking scriptが実行されます。もしunlocking scriptからコピーされたスタックデータとともに実行されたlocking scriptの結果が"TRUE"なら、unlocking scriptはlocking scriptによって課されていた条件を解くことに成功したということです。よって、インプットはUTXOを使用する有効な権限を持っているということになります。もし"TRUE"以外が実行結果に残ってしまっているのであれば、インプットは有効ではありません。UTXOに置いてある使用条件を満たすことができなかったからです。重要なこととして、UTXOはブロックチェーンに永遠に記録され続けるため成功しなかった場合UTXOであるという状態は変化せず、何度新しいトランザクションがUTXOを不正に使用しようとしてもUTXOは全く影響を受けません。UTXOの条件を正しく満たす有効なトランザクションだけがUTXOに"使用済み"という印をつけ、使用可能な(未使用)UTXOのセットから削除させることができるのです。

トランザクションscriptを評価するためのscriptSigとscriptPubKeyの結合はunlocking scriptとlocking

scriptのよくある例(公開鍵ハッシュへの支払い)です。これは、scriptの有効性チェックの前のunlocking scriptとlocking scriptが連結されたscriptを示しています。

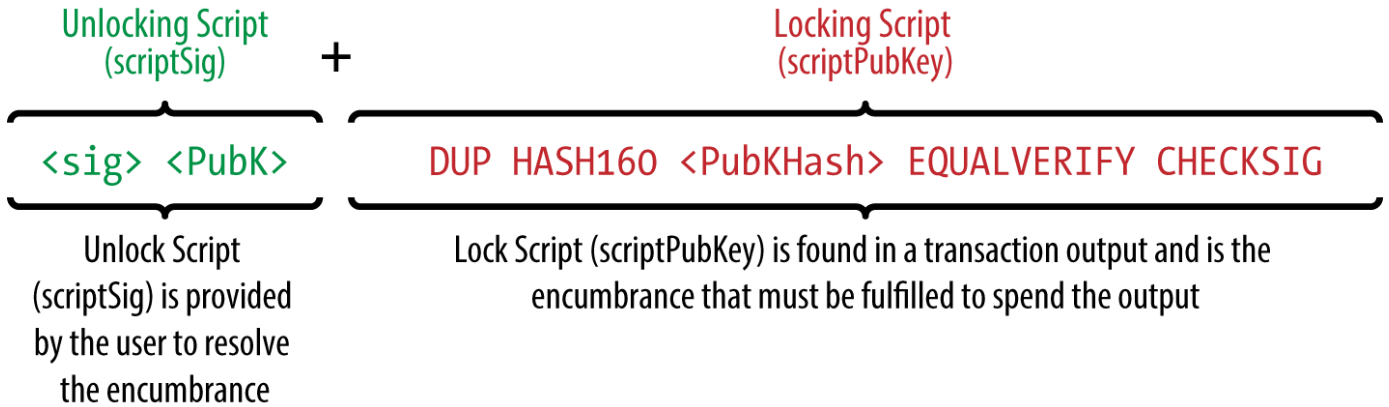


Figure 1. トランザクションscriptを評価するためのscriptSigとscriptPubKeyの結合

## Script言語

Scriptと呼ばれるBitcoinトランザクションスクリプト言語は、Forth言語のような逆ポーランド記法の言語です。もしちんぷんかんぷんに聞こえるとしたら、あなたはおそらく1960年代のプログラミング言語を勉強したことがないのでしょう。scriptはとてもシンプルな言語で、限られたハードウェアで動くように設計された言語です。おそらく電卓のような組み込みデバイスくらい簡単なハードウェアです。これは最小の処理のみを必要とし、最近のプログラミング言語でできるようなことの多くはできません。

BitcoinのScript言語はスタックベース言語と呼ばれています。というのは、スタックと呼ばれるデータ構造を使っているからです。スタックとはとても簡単なデータ構造で、イメージ的にはカードを重ねたもののようなものです。スタックは2つの操作を許しています。pushとpopです。pushはアイテムをスタックの一番上に加えます。popは一番上にあるアイテムをスタックから除きます。

Script言語はそれぞれのアイテムを左から右に処理することでscriptを実行していきます。数値(定数)がスタックにpushされます。オペレーターは1つまたは複数の値をスタックに対してpushまたはpopし、またはそれらに対して何らかの操作をします。場合によっては操作した結果をスタックにpushするかもしれません。例えば、`OP_ADD` はスタックから2つのアイテムをpopして、2つのアイテムを加え合わせ結果をスタックにpushします。

条件オペレーターは条件を評価して、TRUEかFALSEというブール型の結果を作り出します。例えば、`OP_EQUAL` はスタックから2つのアイテムをpopして、もしそれらが等しいならTRUE(TRUEは数値の1によって表現されます)をpushし、等しくなければFALSE(数値の0で表します)をpushします。Bitcoinトランザクションscriptは通常有効なトランザクションを示すTRUEの結果を生成するために条件オペレーターを含んでいます。

[Bitcoinのscript検証を使って簡単な算数をやってみる](#)。図では `2 3 OP_ADD 5 OP_EQUAL` というscriptで、加法オペレーター `OP_ADD` を実行し2つの数値を加え結果をスタックに置き、次に `OP_EQUAL` の結果と5が等しいかをチェックする条件オペレーター `OP_EQUAL` を実行しています。簡潔に言えば、`OP_` というプレフィックスは [Bitcoinのscript検証を使って簡単な算数をやってみる](#) で省略されています。

以下はちょっとだけ複雑なscriptで、`2 + 7 - 3 + 1` を計算しています。

scriptがいくつかのオペレーターを含んでいるとき、スタックの性質上1つのオペレーターの結果を次のオペレーターだけが使うことができます。

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

鉛筆と紙を使ってこのscriptが有効かあなた自身でやってみましょう。scriptの実行が終わった段階で、スタックにTRUEが残っているはずで

ほとんどのlocking scriptは資金の使用にあたっての所有権の証明のためBitcoinアドレスや公開鍵を参照していますが、locking scriptは複雑である必要はありません。結果としてTRUEが出力されるlocking scriptとunlocking scriptのどんな組み合わせも有効とみなされます。Script言語の例として使った簡単な算数も、トランザクションアウトプットをロックするために使うことができるきちんとしたlocking scriptです。

locking scriptとしてさきほどの算数のscript例の一部を使ってみましょう。

```
3 OP_ADD 5 OP_EQUAL
```

これは以下のunlocking scriptを持つインプットがトランザクションにあれば満たすことができます。

```
2
```

有効性を確認するソフトウェアはlocking scriptとunlocking scriptをくっつけて以下のscriptを作ります。

```
2 3 OP_ADD 5 OP_EQUAL
```

[Bitcoinのscript検証を使って簡単な算数をやってみる](#)。図で見たように、このscriptが実行されるとこの結果はOP\_TRUEになりトランザクションは有効であると分かります。この場合、有効なトランザクションアウトプットのlocking scriptだけでなく、算数の計算ができる人であれば誰でも2がこのscriptを満たすことができるので誰でもUTXOを使うことができます。

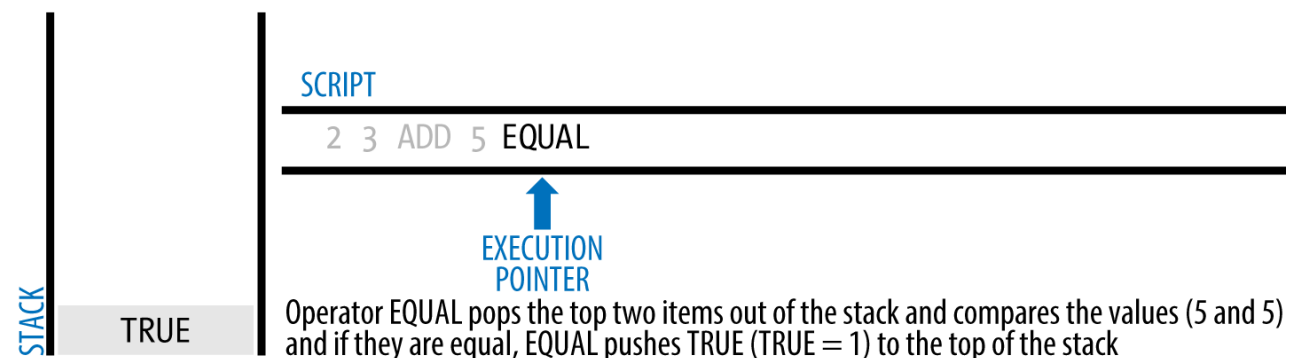
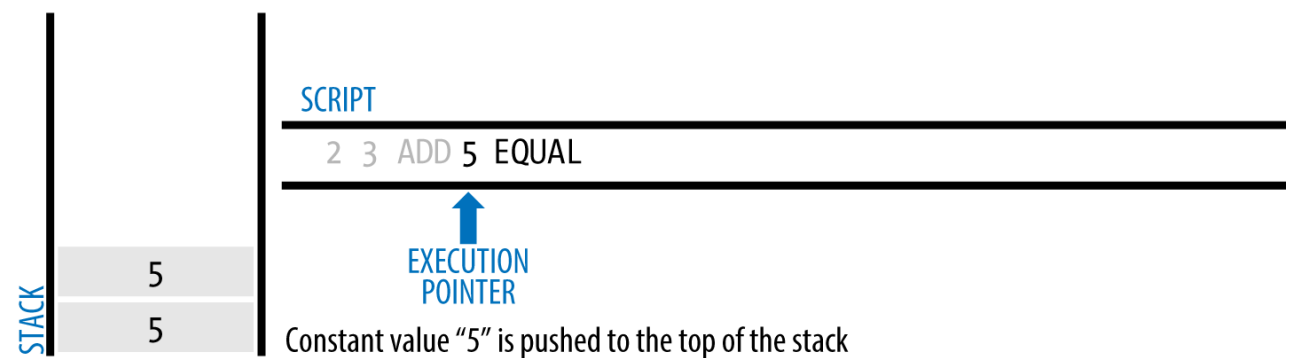
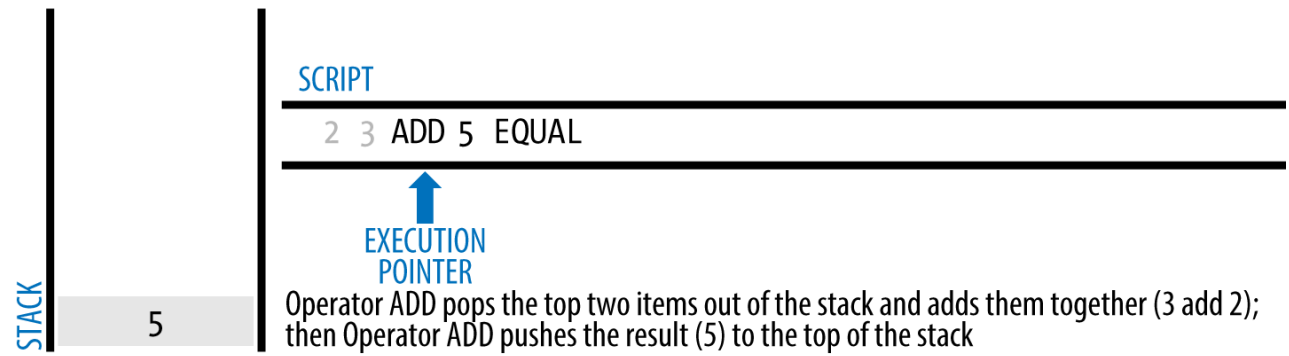
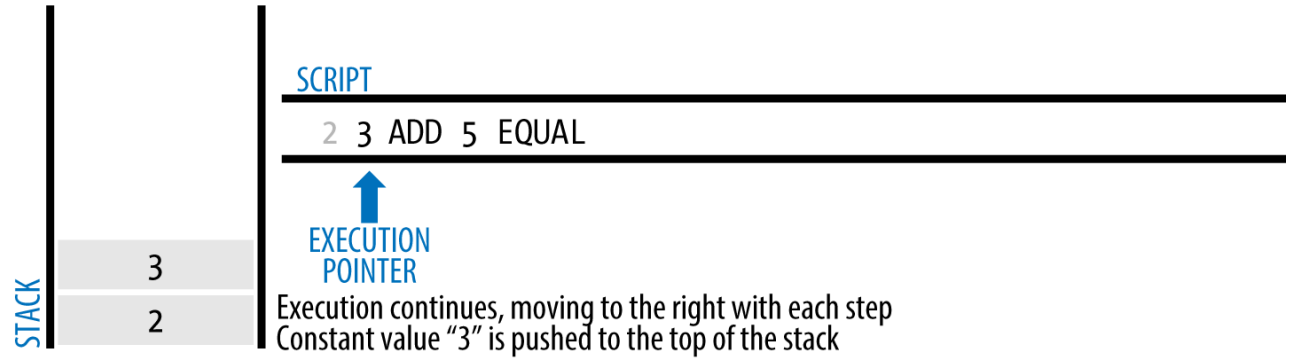
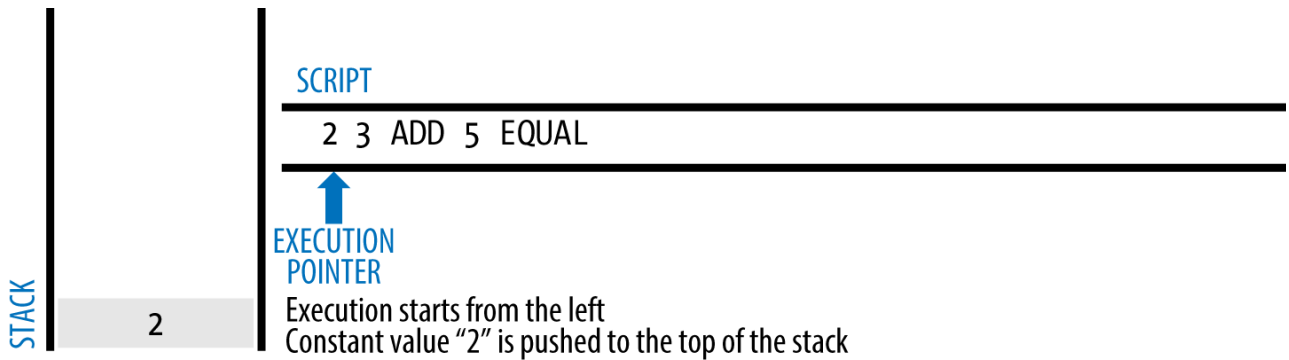




Figure 2. Bitcoinのscript検証を使って簡単な算数をやってみる。

#### TIP

もしスタックの一番上にTRUE ( `&#x7b;0x01&#x7d;` のように表現されます)、またはTRUEではないが0以外の値があればトランザクションは有効と検証されたこととなります。または、script実行後にスタックに空値ではなく何も残っていなかった場合もトランザクションは有効と検証されたこととなります。トランザクションが無効になってしまう場合は、スタックの一番上に偽(`&#x7b;&#x7d;` のように表現される長さ0の空値)がある場合や、OP\_VERIFYやOP\_RETURNまたはOP\_ENDIFのような条件付き終了オペレータなどによって明示的にscript実行が終了させられる場合です。詳細については [\[tx\\_script\\_ops\]](#) を参照してください。

## チューリング不完全

BitcoinトランザクションScript言語は多くのオペレーターを持っています。しかし、意図的にループやif文などの分岐処理がないように制限されています。これは言語が `&#x7b;&#x7d;` チューリング完全ではないということを書いており、このようになっているのはscriptの簡潔さや実行時間を予測できるようにすることがあります。Script言語は汎用言語ではないのです。これらの制限によって、無限ループを作ることやBitcoinネットワークを使ったDOS攻撃を起こすようなトランザクションに内在する"論理爆弾(logic bomb)"などを作ることができなくなっています。思い出してみてください、全てのトランザクションはBitcoinネットワーク上の全てのフルノードによって有効性が確認されているので、トランザクションの有効性確認処理に問題があれば簡単に脆弱性が作れてしまいます。言語が制限されているためにトランザクションの有効性確認メカニズムが脆弱性を生むことを防いでいるのです。

## ステートレスな検証

BitcoinトランザクションScript言語はステートレスです。これは、scriptの実行前の状態を何も保持しない、またはscriptの実行後の状態を一切保存しないということです。このため、scriptを実行するために必要な全ての情報はscriptの中に含まれていることになり、scriptはどんなシステム上でも同じプロセスで実行できることが予測できるということになります。もしあなたのシステムがscriptを検証できるなら、確実にBitcoinネットワーク内の他全てのシステムもまたscriptを検証でき、有効なトランザクションは全ての人に対して有効なのです。この結果の予測可能性はBitcoinシステムの本質的な利点です。

## 標準的なトランザクション

Bitcoinの発展の最初の数年は、開発者たちがBitcoinリファレンスクライアントによって処理されるscriptの種類にいくつかの制限を加えていました。これらの制限は `isStandard()` と呼ばれる関数の中にあり、5つの"標準的なトランザクション"の種類が定義されています。これらの制限は一時的なもので、あなたがこれを読むときまでに解除されているかもしれません。これらの制限が解除されるまでは、5つの標準的なトランザクションスクリプトだけがBitcoin

Coreクライアントやこれを動作させている多くのマイナーに受け入れられています。非標準的なトランザクションを作るとは可能ですが、このトランザクションをブロックに入れてくれるマイナーを見つけなければいけません。

どんなscriptが有効なトランザクションscriptとして現在許可されているかを見るためにBitcoin Coreクライアント(リファレンス実装)のソースコードをチェックしてみましょう。

トランザクションscriptの5つの標準的な種類は、`pay-to-public-key-hash` (P2PKH), `public-key, multi-signature` (最大15個のキーまで), `pay-to-script-hash` (P2SH), `data output` (OP\_RETURN)

です。これらの詳細な説明は次の節で行います。

## Pay-to-Public-Key-Hash (P2PKH)

Bitcoinネットワーク上で処理されているトランザクションの多くはP2PKHトランザクションです。これはBitcoinアドレスとして知られている公開鍵ハッシュを伴ったトランザクションアウトプットを拘束しているlocking scriptを含んでいます。Bitcoinアドレスへの支払いをするトランザクションはP2PKH scriptを含んでおり、このP2PKH scriptでロックされているアウトプットは公開鍵とこの公開鍵に対応したデジタル署名を提示することで解除(資金の使用)ができます。

例えば、AliceがBobのカフェに支払う場面を再度見てみましょう。AliceはこのカフェのBitcoinアドレスに0.015bitcoinの支払いをします。このトランザクションアウトプットには以下のようなlocking scriptが含まれています。

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

Cafe Public Key Hash はこのカフェのBitcoinアドレス (Base58Checkエンコーディングが施されていないもの)と同じものです。多くのアプリケーションでは公開鍵ハッシュ を16進数で表したものを使っており、なじみのある"1"から始まるBase58Check形式のBitcoinアドレスではありません。

このlocking scriptは以下のunlocking scriptで条件を満たすことができます。

```
<Cafe Signature> <Cafe Public Key>
```

この2つのscriptを合わせることで、以下の検証scriptの形になります。

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160  
<Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

これを実行するとき、unlocking scriptがlocking scriptの条件を満たしかつその場合に限り、この結合されたscriptはTRUEと評価されます。他の言い方をすると、もしunlocking scriptがBobのカフェの秘密鍵から作られた有効な署名を持っていれば結果はTRUEになります。

☒ `<xref linkend="P2PubKHash1" xrefstyle="select: labelnumber"/>` と ☒ `<xref linkend="P2PubKHash2" xrefstyle="select: labelnumber"/>` は結合されたscriptの逐次実行を(2つの部分で)表しており、この結合されたscriptが有効なトランザクションであることを証明することになります。

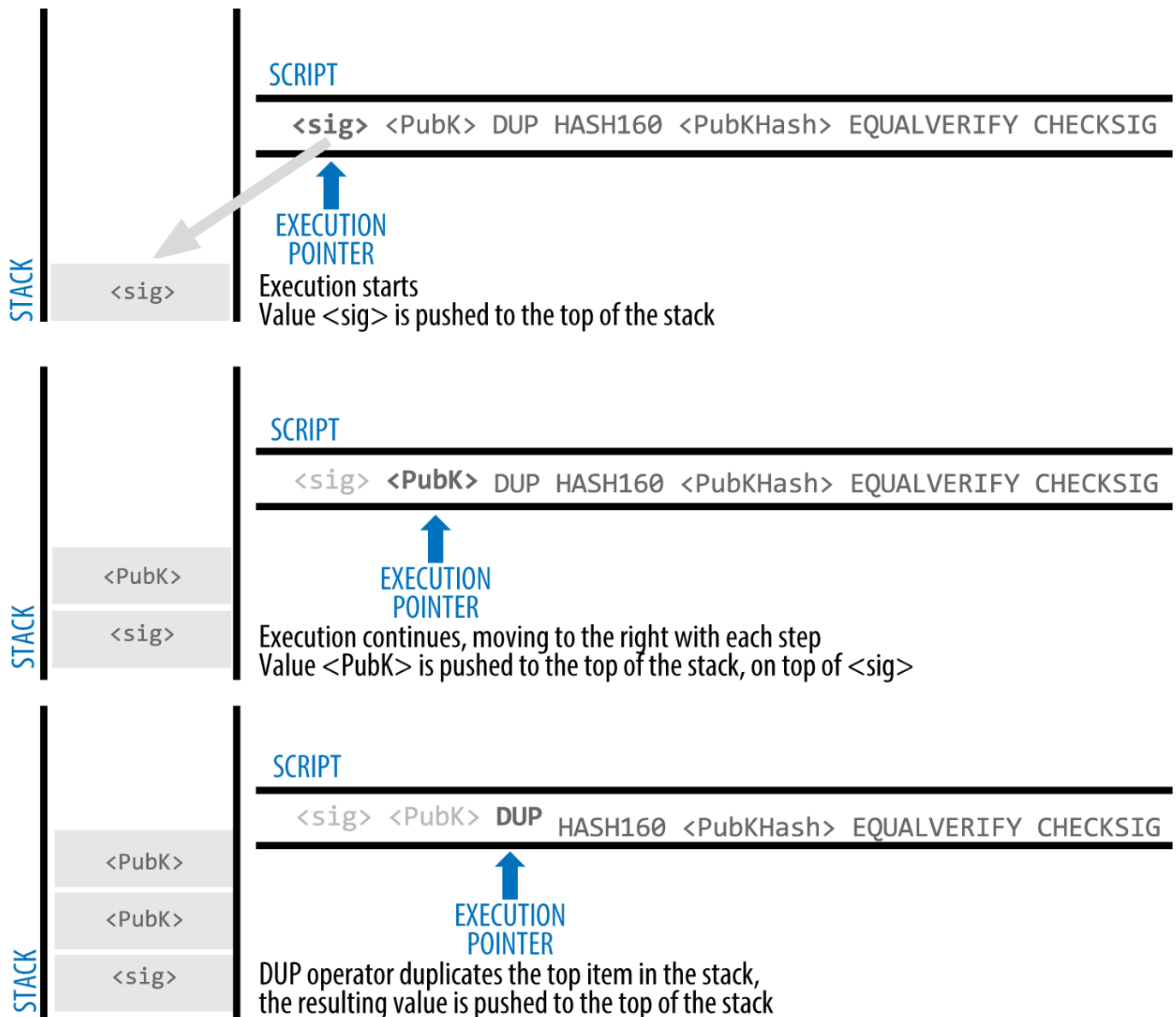


Figure 3. P2PKHトランザクションにおけるscriptの評価(1/2)

## Pay-to-Public-Key

pay-to-public-keyはpay-to-public-key-hashよりもよりシンプルなBitcoin支払いの形式です。このscript形式は、前に出てきたP2PKHでのとても短い公開鍵ハッシュではなく、公開鍵そのものをlocking scriptに配置しています。pay-to-public-key-hashはBitcoinアドレスをより短くし使いやすくするためにSatoshi Nakamotoによって発明されたものです。現在pay-to-public-keyはcoinbaseトランザクションでよく見られるもので、P2PKHが使用できるように更新されていない古いマイニングソフトウェアで使われています。

pay-to-public-key locking scriptは以下のようなものです。

```
<Public Key A> OP_CHECKSIG
```

アウトプットを解除するために提示されなければならないunlocking scriptは以下のようなシンプルな署名です。

```
<Signature from Private Key A>
```

トランザクションの有効性検証に使われる結合されたscriptは以下です。

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

このscriptは `CHECKSIG` オペレーターを使ったシンプルな scriptで、正しい秘密鍵に紐づく署名を検証しています。

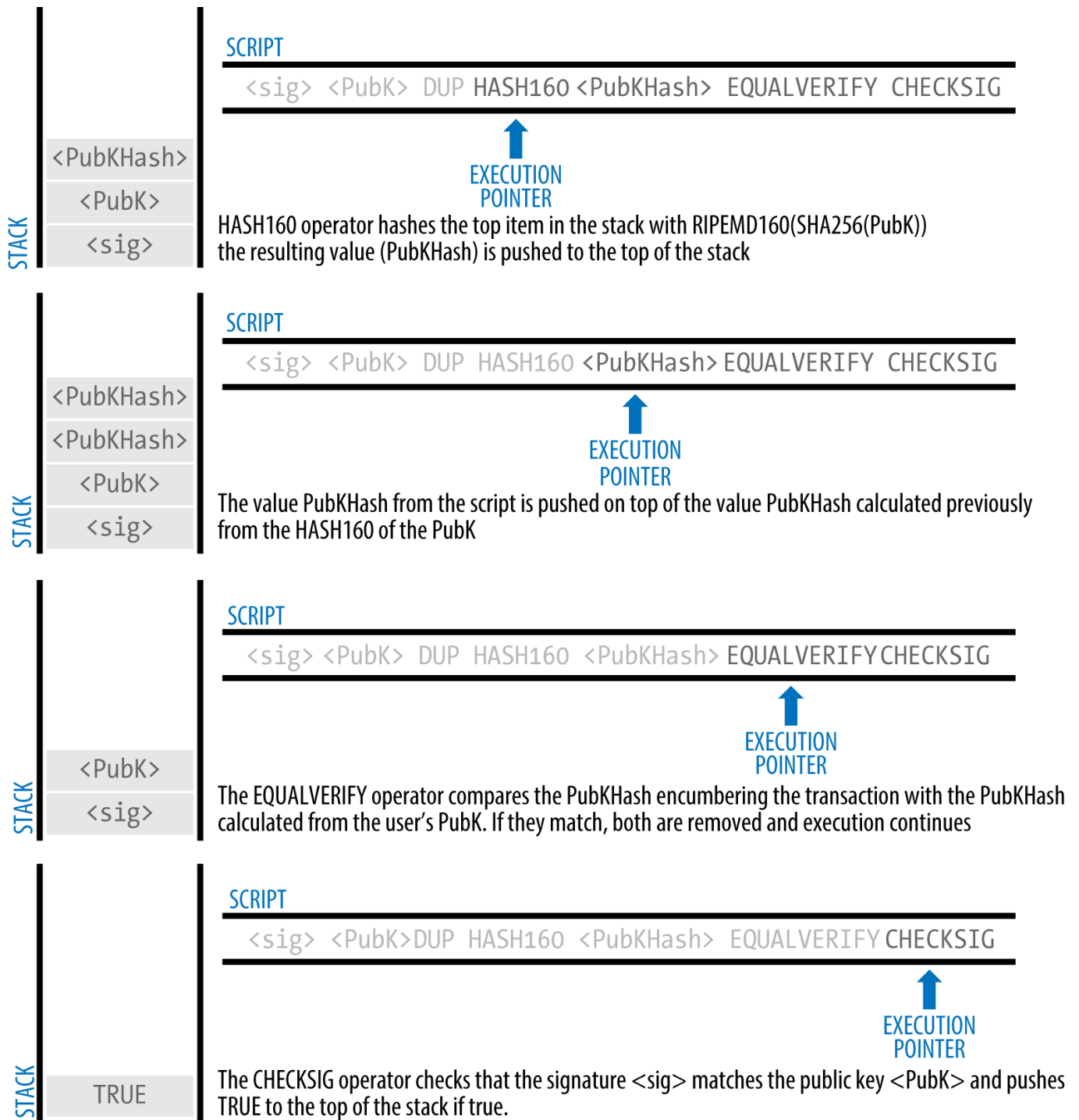


Figure 4. P2PKHトランザクションにおけるscriptの評価(2/2)

## マルチシグネチャ

マルチシグネチャscriptはN個の公開鍵と、解除条件を解放する少なくともM個の署名が入っている条件を設定しています。これはM-of-Nスキームとしても知られており、Nはキーの総数、Mは検証に必要な署名数です。例えば2-of-3マルチシグネチャでは、あらかじめ登録しておいた署名者の3つの公開鍵があり、これらのうち2つを使って有効なトランザクションに対する署名を作らなければいけません。このとき、標準的なマルチシグネチャscriptは多くても15個の公開鍵だけが使用できるように制限されており、これは1-of-1から15-of-

15までのマルチシグネチャ、またはそれぞれの組み合わせを使用できることを示しています。この制限はこの本が出版されるまでに引き上げられるかもしれません。Bitcoinネットワークによって現在何が許可されているかを見るために `isStandard()` 関数をチェックしてみてください。

M-of-Nマルチシグネチャ条件を設定しているlocking scriptの一般形式は以下です。

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

ただし、Nは登録されている公開鍵の総数、Mはアウトプットを使うにあたって必要な署名数です。

2-of-3マルチシグネチャ条件を設定しているlocking scriptは以下のようなものです。

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

このlocking scriptは署名と公開鍵のペアを含む以下のunlocking scriptで条件を満たすことができます。

```
OP_0 <Signature B> <Signature C>
```

または、登録されている3つの公開鍵に対応する秘密鍵から作られる署名であれば、どんな2つの組み合わせでも使うことができます。

#### NOTE

最初に置かれている `OP_0` は `CHECKMULTISIG` のオリジナルの実装にバグがありそれを補完するために必要となっています。このバグというのは、`CHECKMULTISIG` を実行した時に処理に関係のないスタック上のアイテムを余分に1つpopしてしまうというバグです。`CHECKMULTISIG` 処理は事実 `OP_0` を無視して実行され、`OP_0` は単なる空箱のようなものになっています。

この2つのscriptは以下の結合された検証scriptを形作ります。

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3  
OP_CHECKMULTISIG
```

これを実行するとき、unlocking scriptがlocking scriptの条件を満たしかつその場合に限り、この結合されたscriptはTRUEと評価されます。この場合、解除条件に設定してある3つの公開鍵のうち2つに対応した秘密鍵から作られる有効な署名をunlocking scriptが持っているかどうか条件になります。

## データアウトプット(OP\_RETURN)

Bitcoinの時刻が刻印された分散化元帳であるブロックチェーンは支払い以上のポテンシャルを持っています。多くの開発者たちはデジタル公証人サービス、株券、スマートコントラクトのようなアプリケーションのためにトランザクション

Script言語を使うことでより進んだシステムの安全性や可用性を確保しようとしてきました。当初BitcoinのScript言語をこれらの目的に使っていく場合、ブロックチェーン上に記録されたデータであるトランザクションアウトプットを利用することが考えられました。これで行う例としては、ファイルの存在証明があります。これは、このトランザクションが参照している特定の日付を利用して、あるファイルの存在証明(proof-of-existence)を誰でもできるようにします。このことで、ファイルのデジタルフィンガープリントを記録するのです。

bitcoinの支払いと無関係なデータをブロックチェーン上に記録することは物議を巻き起こしました。多くの開発者たちはこのようなブロックチェーンの使い方を汚いものと考え、思いとどませようと考えました。またある人たちはこれがブロックチェーンテクノロジーの強力な拡張性を示すものと感じ、このような実験を押し進めようとしてきました。支払いと関係のないデータを含めることに反対な人たちはこれにより"ブロックチェーンの肥大化"を引き起こすと考えており、ブロックチェーンが本来運ぶ必要のなかったデータのためにディスクストレージのコストが増大しフルノードを動作させているサーバのコストが増えてしまうと考えました。さらに、このようなトランザクションは使用されないUTXOを作り出し、送り先Bitcoinアドレスの領域20byteを自由に使える領域として使ってしまう。このBitcoinアドレスはデータのために使われるので、これは秘密鍵に対応しておらず決して使われないUTXOを結果として生み出してしまうのです。これはあたかも偽物の支払いのようになってしまいます。決して使われないこれらのトランザクションはUTXOセットから決して削除されず、永遠にUTXOデータベースのサイズを大きくし続け、"肥大化"させてしまいます。

Bitcoin Coreクライアントのバージョン0.9では、妥協策として OP\_RETURN オペレーターが導入されました。 OP\_RETURN は開発者たちが支払いに関係のない80byteのデータをトランザクションアウトプットに追加できるようにしています。"偽物の"UTXOと違って、 OP\_RETURN オペレーターはUTXOセットに保持される必要がない 明示的使用不可アウトプットを作り出します。 OP\_RETURN アウトプットはブロックチェーン上に記録されるためディスク容量を消費しブロックチェーンのデータサイズ増大を促してしまいますが、UTXOセットに保存されないためUTXOメモリプールと高価なRAMのコストの肥大化にはならないようになっています。

OP\_RETURN scriptは以下のようなものです。

```
OP_RETURN <data>
```

このdataは80byteに制限され、多くの場合SHA256アルゴリズム(32byte)の出力結果のようなハッシュになっています。多くのアプリケーションはアプリケーションを示すidをプレフィックスとしてdataの前に置いています。例えば、 [Proof of Existence](#) というデジタル公証人サービスは8byteのプレフィックス "DOCPROOF" を使っていて、ASCIIコードで表すと16進数で 44f4350524f4f46 になります。

OP\_RETURN アウトプットを"使用する"ための"unlocking script"がないことを覚えておいてください。つまり、OP\_RETURN ではこのアウトプットでロックされている資金を使うことはできないのです。そして、使用可能なものとしてUTXOセットに保持しておく必要はありません。このアウトプットに割り当てられているどんなbitcoinも永遠に失われてしまうため、 OP\_RETURN アウトプットは通常0bitcoinを持ちます。もしscript検証ソフトウェアが OP\_RETURN を見つけた場合には、検証scriptの実行を直ちに停止しトランザクションを無効にします。このため、もし偶然 OP\_RETURN アウトプットをトランザクションインプットが参照した場合は、このトランザクションは無効になります。

標準的なトランザクション( `isStandard()` )を確認してみてください)はたった1つだけしか `OP_RETURN` アウトプットを持つことができませんが、 `OP_RETURN` アウトプットは他の種類のアウトプットを持つトランザクションと結合することができます。

現在のBitcoin `Core`バージョン0.10では2つの新しいコマンドラインオプションが追加されました。 `datacarrier` オプションは `OP_RETURN` トランザクションのリレーとマイニングを行うかどうかをコントロールしており、デフォルトは "1"でリレーとマイニングの実行を許可するものになっています。 `datacarriersize` オプションは数値を引数として取り `OP_RETURN` データの最大バイトサイズを指定します。この最大バイトサイズのデフォルトは40byteです。

#### NOTE

`OP_RETURN`は最初80byteの制限を付けた形で提案されていましたが、この機能が実際にリリースされた時に制限が40byteに削減されました。2015年2月にリリースされたBitcoin `Core`バージョン0.10の中で、この制限は80byteに引き上げられました。ノードは `OP_RETURN` トランザクションをリレー、マイニングしないか、または単にリレーだけして80byteより小さいデータを持つ `OP_RETURN` トランザクションだけをマイニングするか、を選べるようになっています。

## Pay-to-Script-Hash (P2SH)

`pay-to-script-hash` (`P2SH`)は2012年に導入されたもので、複雑なトランザクション `script`をはるかに簡単化した新しい種類のトランザクションです。`P2SH`を説明するために、実用的な例を見てください。

[[ch01\\_intro\\_what\\_is\\_bitcoin](#)]で、ドバイで電子機器の輸入業をやっているMohammedを紹介しました。Mohammedの会社は会社の口座のためにBitcoinのマルチシグネチャ機能を利用しています。マルチシグネチャ `script`はBitcoinの先進的なScript言語の主要な使い方の1つで、とても強力な機能です。Mohammedの会社は全ての顧客からの支払い(会計用語で"売掛金")にマルチシグネチャを使っています。マルチシグネチャスキームを使い、顧客による全ての支払いは次のような方法で安全性を担保しています。支払いを実行するには少なくとも2つの署名が必要であり、登録されている人はMohammed、彼のパートナーのうちの1人、バックアップキーを持っている彼の代理人です。このようなマルチシグネチャスキームはコーポレートガバナンスを提供し、盗難、横領、または紛失を防ぐ役割があります。

このための `script`はとても長く、以下のようなものです。

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

マルチシグネチャ `script`はとても強力な機能ですが、これは扱いにくいものなのです。というのは、Mohammedは支払いをする前にこの `script`について全ての顧客に説明する必要があるためです。それぞれの顧客は特別なトランザクション `script`を作ることができる特別なウォレットを使う必要があり、また特別な `script`を使ってどのようにトランザクションを作ればよいか理解する必要があります。さらに、この `script`がとても長い公開鍵を含んでいるため、作られたトランザクションは単純な支払いトランザクションと比べて約5倍も大きいのです。そのため、余分に大きいトランザクションデータサイズの負担が顧客ごとのトランザクション手数料として乗っかってきます。最終的に、このような大きなトランザクション `script`は使用されるまで全てのフルノードのRAM内のUTXOセットに保持されます。このような問題点によって実用上、複雑なアウトプット `sc`



riptの使用が難しくなってしまうのです。

pay-to-script-hash (P2SH) はこれらの実用的な難点を解決するために開発され、Bitcoinアドレスでの支払いと同じくらい簡単に複雑なscriptを使えるようにしたのです。P2SHでの支払いで、複雑なlocking scriptは暗号学的なハッシュであるデジタルフィンガープリントに置き換えられます。UTXOを使おうとするトランザクションがのちに作られたとき、このトランザクションはunlocking scriptだけでなくこのハッシュとマッチするscriptを含んでいなければいけません。簡単に言って、P2SHは"このハッシュとマッチするscriptに対して支払い、このscriptはのちほどこのアウトプットが使用されるときに与えられます"という意味です。

P2SHトランザクションでは、ハッシュによって置き換えられたlocking scriptは *redeem script* と呼ばれます。なぜなら、これがlocking scriptとしてよりはむしろ回収時にシステムに提供されるからです。P2SHを使用しない複雑なscriptはP2SHではないscriptを示し、P2SHを使用した複雑なscriptはP2SHでエンコードされた同じscriptを示しています。

Table 4. P2SHを使用しない複雑なscript

|                  |  |
|------------------|--|
| Locking Script   | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG |
| Unlocking Script | Sig1 Sig2  |

Table 5. P2SHを使用した複雑なscript

|                  |  |
|------------------|--|
| Redeem Script    | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG |
| Locking Script   | OP_HASH160 <20-byte hash of redeem script><br>OP_EQUAL       |
| Unlocking Script | Sig1 Sig2 redeem script                                      |

テーブルにある通りP2SHのほうは、アウトプットを使用するための詳細条件が書かれた複雑なscriptがlocking scriptにはありません。その代わりに、redeem scriptのハッシュだけがlocking scriptにはあり、redeem script自身はアウトプットが使用される時のunlocking scriptの一部としてあとで提供されます。

Mohammedの会社の場合の複雑なマルチシグネチャscriptとP2SH scriptを見てみましょう。

まず、Mohammedの会社が顧客からの支払いに使っているマルチシグネチャscriptは以下です。

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

上記の大なり小なり括弧を実際の公開鍵(04から始まる520bitの数値)に置き換えてみると、以下のようにとても長くなってしまふことが分かるはずです。

2

```
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984
D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308
EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49047E632
48B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC1
0F1E8E8F3020DECDBC3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9
D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1E
CED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D1
37AAB59E0B000EB7ED238F4D800 5 OP_CHECKMULTISIG
```

このscript全体は20byteの暗号的ハッシュで表現できます。このハッシュは最初にSHA256ハッシュ化アルゴリズムを適用しその後この結果にRIPEMD160アルゴリズムを適用することで作成されます。前のscriptに対してのこのハッシュ化して得た20byteのハッシュは以下になります。

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

P2SHトランザクションは、以下のような長いscriptの代わりにこのハッシュを含めたlocking scriptでアウトプットをロックしています。

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

見て分かるように前のlocking scriptよりもずいぶん短いことが分かります。"5つのキーのマルチシグネチャscriptに対する支払い"ではなく、P2SHでは"このハッシュを持ったscriptへの支払い(pay to a script with this hash)"になります。Mohammedの会社への支払いをする顧客は、とても短いlocking scriptを含めるだけで支払いができるのです。MohammedがこのUTXOを使いたいときは、オリジナルのredeem scriptと解除するための署名を以下のように提供しなければいけません。

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>
```

2つのscriptは2つの段階で結合されます。まず、このハッシュが合っているかを確認するためにlocking scriptに対してredeem scriptがチェックされます。

```
<2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 <redeem scriptHash> OP_EQUAL
```

もしredeem scriptハッシュが合っていれば、unlocking scriptはredeem scriptを解除するために実行されます。

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG
```

## Pay-to-script-hashアドレス

P2SHに関してもう1つの重要な点は、BIP0013で定義されているようにscriptハッシュをエンコードしてアドレスとして使えるようにする点です。P2SHアドレスはscriptの20byteハッシュをBase58Checkエンコードしたものです。これはちょうど公開鍵の20byteハッシュのBase58CheckエンコードをしたBitcoinアドレスのようなものです。P2SHアドレスはversion prefixとして"5"が使われており、Base58Checkエンコードしたアドレスは"3"から始まるものになっています。例えば、Mohammedの複雑なscriptでは、P2SHとして20byteにハッシュ化されてBase58Checkエンコードを施されたP2SHアドレスは39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw になります。Mohammedはこの"アドレス"を彼の顧客に送ることで、彼の顧客はBitcoinアドレスに対する支払いとあたかも同じようにウォレットを使うことができます。3というプレフィックスはこのアドレスが特別な種類のアドレスであることを示します。

P2SHアドレスは全ての複雑な点を隠蔽し、支払う人がscriptを見ることなく支払いができるようにしています。

## pay-to-script-hashの利点

pay-to-script-hashはlocking scriptを複雑なまま直接扱うことに比べて以下の利点があります。

- より短いフィンガープリントで複雑なscriptを置き換えることで、トランザクションのデータサイズを小さくする
- scriptがアドレスとして実装されることで、送り主と送り主のウォレットはP2SHに関する複雑な実装をする必要がない
- P2SHは、scriptを構成する負担を送り手ではなく受け手側に移している
- P2SHは、アウトプットが持つ長いscript(これはUTXOセットに含まれるためメモリを圧迫する)をインプット側(ブロックチェーン上にのみ保存される)に移すことでデータストレージの負担をインプット側に移している
- P2SHは、支払い時に生じる長いscriptを資金が使われる時点で生じるようにすることで、データストレージの負担が生じる時刻を移している
- P2SHは、送り手が長いscriptを伴って資金を送るときに負担するトランザクション手数料を、受け手がredeem scriptを使って資金を使うときに負担するように変更している

## redeem scriptとisStandard検証

Bitcoin Coreクライアントのバージョン0.9.2より前では、pay-to-script-hashはisStandard()関数による標準的なBitcoinトランザクションスクリプトに含められていませんでした。これは、トランザクションを使用するときに提供されるredeem scriptが、OP\_RETURNとP2SH自身を除くP2PK、P2PKH、マルチシグネチャのうちのどれか1つに統合されるしかないということを意味しています。

現在のBitcoin Coreクライアントのバージョン0.9.2では、P2SHトランザクションは有効なscriptはなんでも含むことができ、P2SHはもっとフレキシブルになりました。そして、多くの斬新で複雑なトランザクションの実験が許可されています。

P2SH redeem scriptの中にP2SHを置くことはできないという点に注意してください。というのは、P2SHの設計で再帰ができないようになっているためです。また、まだredeem scriptの中でOP\_RETURNを使用することはできません。OP\_RETURNは定義によってあとでこれを含むアウトプットを使用することができないためです。

注意点として、redeem scriptはP2SHアウトプットを使用しようとするまでBitcoinネットワークに提供されません。このため、もし間違ったredeem scriptハッシュを伴ったアウトプットをロックしてしまった場合も、このトランザクションはお構いなしにBitcoinノードの検証をパスします。しかし、このアウトプットを使うことはできません。なぜなら、redeem scriptをこのアウトプットを使用するときに提示しても、このredeem scriptハッシュが間違っているためredeem scriptが受理されないのです。これはリスクを生み出してしまいます。というのは、使用することができないP2SHにbitcoinがロックされてしまうためです。redeem scriptハッシュだけではこのredeem scriptハッシュがredeem scriptを表しているか分からないため、たとえ無効なredeem scriptだとしてもBitcoinネットワークはP2SH解除条件をパスしてしまうでしょう。

**WARNING**

P2SH locking scriptはredeem scriptのハッシュを含んでいます。このハッシュは、redeem scriptそのものに関して一切ヒントを与えてくれません。もしこのredeem scriptが無効だったとしてもP2SHトランザクションは有効だと考えられ受け入れられてしまいます。あとで使えないような形で間違っ Bitcoinをロックしてしまうかもしれません。

# Bitcoinネットワーク

## Peer-to-Peer ネットワーク設計

Bitcoinはインターネット上のpeer-to-peerネットワークとして構築されています。peer-to-peer、またはP2Pという言葉は、ネットワークに参加しているコンピュータがそれぞれ同等の立場を持ち、"特別な"ノードがなく、全てのノードがサービス負荷を負担し合っていることを指します。ネットワークノードは"フラット"なトポロジーを持つメッシュネットワークの中で互いに繋がっています。ここにサーバ、どこかに中央を持つサービス、ネットワーク内の階層はありません。peer-to-peerネットワーク内のノードは、サービスを提供もした同時に消費もすることで互いに利益を保っており、これがネットワークへ参加することのインセンティブになっています。peer-to-peerネットワークは本質的に柔軟であり、非中央的でオープンです。P2Pネットワークの代表的な例は初期インターネットそのもので、IPネットワーク上のノードは全て平等でした。今日のインターネットの構造はより階層的になりましたが、インターネットプロトコルはまだフラットポロジエのエッセンスを保っています。Bitcoinが現れる前にP2Pテクノロジーを使った最も大きく最も成功したサービスはファイル共有であり、パイオニアとしてはNapster、最近の革新例としてはBitTorrentがあります。

Bitcoinは計画的に構築されたpeer-to-peerのデジタルキャッシュシステムです。コントロールの分散化はBitcoinの中心的な設計原則であり、フラットで分散的なP2Pコンセンサスネットワークによってメンテナンスされています。

"Bitcoinネットワーク"という言葉は、Bitcoin P2Pプロトコルが動作しているBitcoinノード全体を指します。Bitcoin P2Pプロトコルに加えて、Stratumのような他のプロトコルもあります。Stratumはマイニング、軽量またはモバイルのウォレットに使われたりしています。これらのBitcoin P2Pプロトコル以外のプロトコルは、Bitcoin P2Pプロトコルを使っているBitcoinネットワークにアクセスするサーバをルーティングするゲートウェイで使われています。また、これにより他のプロトコルで動くノードにネットワークを拡張することができます。例えば、StratumサーバはStratumマイニングノードをStratumプロトコルを通してメインのBitcoinネットワークに接続させ、StratumプロトコルとBitcoin P2Pプロトコルの橋渡しをします。Bitcoin P2Pプロトコル、プールマイニングプロトコル、Stratumプロトコル、およびBitcoinシステムの各要素を繋げるその他の関連したプロトコルを全て含む全ネットワークを"拡張されたBitcoinネットワーク"という言葉で表します。

## ノードタイプと役割

Bitcoin P2Pネットワーク内のノードは平等ではありませんが、これらはいくつかの役割に分かれています。Bitcoinノードは、ルーティング、ブロックチェーンデータベース、マイニング、ウォレットという機能の集合体です。これらの機能4つを全て持つものがフルノードであり4種類のBitcoinネットワークノード機能:ウォレット、マイナー、フルブロックチェーンデータベース、ネットワークルーティング図で示されています。

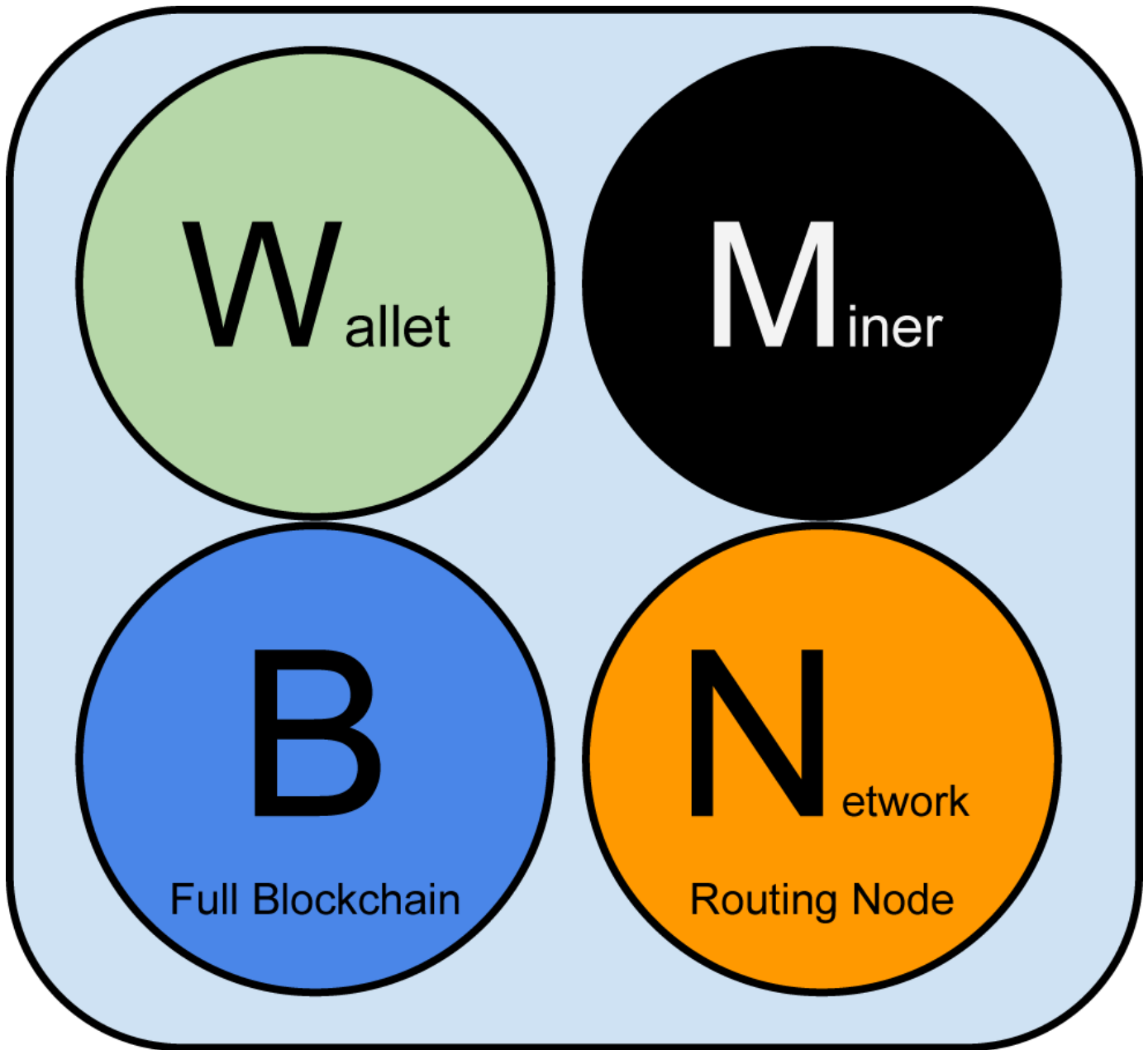


Figure 1. 4種類のBitcoinネットワークノード機能:

ウォレット、マイナー、フルブロックチェーンデータベース、ネットワークルーティング

全てのノードはBitcoinネットワークに参加するためにルーティング機能を必ず持っていて、その他の機能は持っていたり持っていなかったりします。全てのノードはトランザクションとブロックを検証して伝搬し、その他のピアを発見してコネクションを常に保っています。4種類のBitcoinネットワークノード機能: ウォレット、マイナー、フルブロックチェーンデータベース、ネットワークルーティング図のフルノード例では、ルーティング機能を"Network Routing Node"と書いてあるオレンジの円で示しています。

フルノードと呼ばれるいくつかのノードは、完全で最新のブロックチェーンの管理もしています。フルノードは外部への参照をすることなく閉じられた形で自律的かつ厳然にトランザクションを検証します。いくつかのノードはブロックチェーンの一部の管理のみを行っており、*simplified payment verification* または SPV と呼ばれている方法でトランザクションを検証します。これらのノードはSPVまたは軽量ノードと呼ばれています。さきほどの図にあったフルノード例では、フルノードのブロックチェーンデータベースを"Full Blockchain"と書いてあるブルーの円で示しています。いろいろなノードタイプやゲートウェイ、プロトコルを表した拡張されたBitcoinネットワーク全体図では、SPVノードがブルーの円がない形で描かれており、

これはブロックチェーンの完全なコピーを持たないということを表しています。

マイニングノードは新しいブロックを作り出す競争をしており、proof-of-workアルゴリズムを解くための特別なハードウェアを動作させて行っています。いくつかのマイニングノードはフルノードでもあり、ブロックチェーンの完全なコピーを管理しています。一方それ以外はマイニングプールに参加している軽量ノードであり、フルノードを管理しているプールサーバに依存しています。マイニング機能は、フルノードの中にある"マイナー"と書かれた黒い円で示されています。

ウォレットは一部がフルノードになっており、通常デスクトップBitcoinクライアントという形でフルノードになっています。スマートフォンなどリソースが限られているデバイスでは多くのウォレットがSPVノードになっています。ウォレット機能は4種類のBitcoinネットワークノード機能:  
ウォレット、マイナー、フルブロックチェーンデータベース、ネットワークルーティング図にある"Wallet"と書かれた緑色の円で示されています。

## Bitcoin

P2Pプロトコル上の主なノードタイプに加えて、その他のプロトコルで動作しているノードもあります。例えば、マイニングプール特化型プロトコルや軽量クライアントアクセスプロトコルなどです。

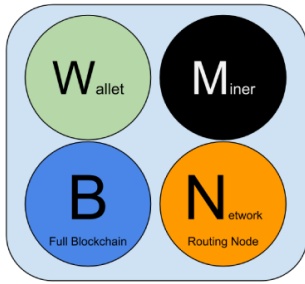
拡張されたBitcoinネットワーク上の様々なノードタイプは拡張されたBitcoinネットワーク上の主なノードタイプを示しています。

# 拡張されたBitcoinネットワーク

Bitcoin P2Pプロトコルが動作しているメインのBitcoinネットワークは7000から10000個のノードから構成されており、それぞれBitcoinリファレンスクライアント(Bitcoin Core)のいろいろなバージョンが動作しています。また、数百個のノードはBitcoin P2Pプロトコルのいろいろな実装である BitcoinJ、Libbitcoin、およびbtcdなどで動作しています。Bitcoin P2Pネットワーク上の少数のノードはマイニングノードも兼ねていて、マイニング、トランザクション検証、新ブロック生成の競争をしています。いろいろな大きな企業は、Bitcoin Coreクライアントをベースとするフルノードクライアントを使ってBitcoinネットワークと通信をしており、これらはブロックチェーンの完全なコピーやネットワークノードとしての機能は持っているもののマイニングやウォレットの機能は持ちません。これらのノードはネットワークエッジルーターとして機能しており、いろいろなその他のサービス(交換所、ウォレット、ブロックエクスプローラ、決済システム)を構築できるようにしています。

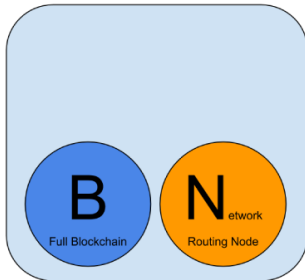
以前説明したように拡張されたBitcoinネットワークはBitcoin P2Pプロトコルが動作しているネットワークを含んでいますが、他にも一部分に特化したプロトコルで動作しているノードもあります。メインのBitcoin P2Pネットワークに接続しているノードは、多くのプールサーバや、その他のプロトコルで動作しているノードをつないでいるプロトコルゲートウェイです。これらの他のプロトコルノードはほとんどプールマイニングノード([ch8]参照)や軽量ウォレットクライアントであり、ブロックチェーンのフルコピーは持っていません。

いろいろなノードタイプやゲートウェイ、プロトコルを表した拡張されたBitcoinネットワーク全体図は拡張されたBitcoinネットワークを示しており、ノードのいろいろなタイプ、ゲートウェイサーバ、エッジルーター、およびウォレットクライアント、またそれぞれが接続し合うために使っているいろいろなプロトコルを示しています。



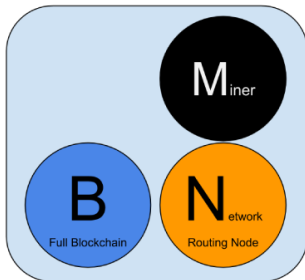
## Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



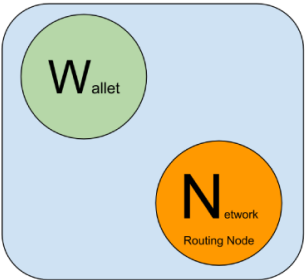
## Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



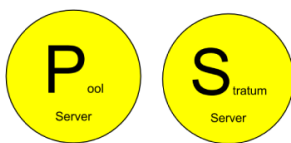
## Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



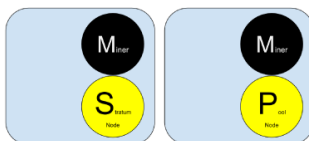
## Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.



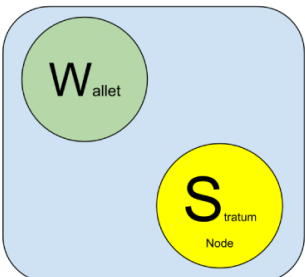
## Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.



## Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



## Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.



Figure 2. 拡張されたBitcoinネットワーク上の様々なノードタイプ

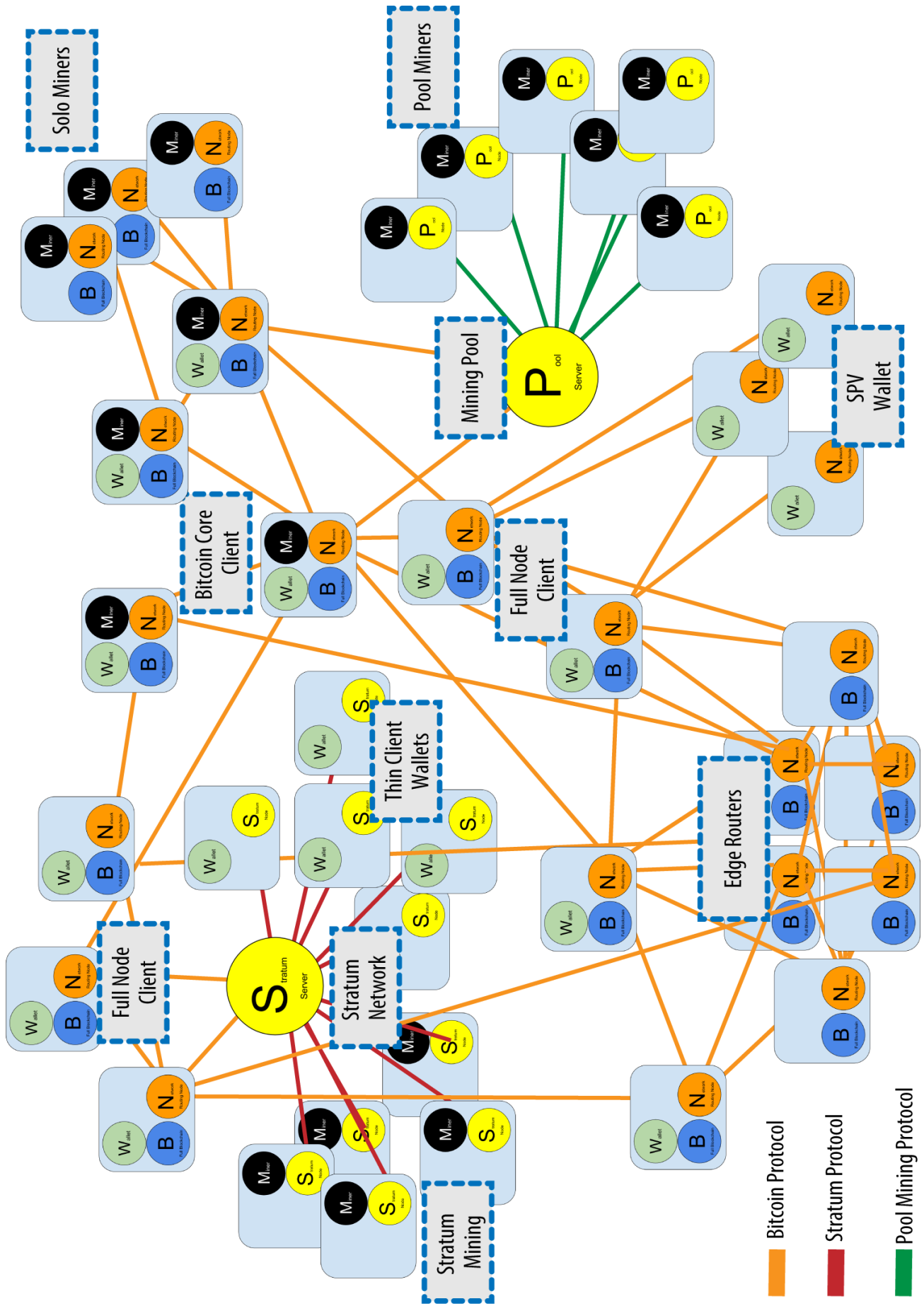


Figure 3. いろいろなノードタイプやゲートウェイ、プロトコルを表した拡張されたBitcoinネットワーク全体図

## ネットワークをどのように発見するのか

新しいノードが立ち上がったとき、Bitcoinネットワークに参加するには他のBitcoinノードを見つけなければいけません。そして、このノードは少なくとも1個のノードを見つけ接続しなければいけません。他のノードの地理的な位置は関係ありません。というのは、Bitcoinネットワークのトポロジは地理と関連づけて決められてはいないからです。このため、ランダムにノードが選ばれます。

知られているピアに接続するために、ノードはTCPコネクションを確立し、通常8333番ポート(一般にBitcoinによって使われているポート)または指定されているなら代替のポートを使います。コネクションを確立すると、ノードは `version` messageを送信することで"ハンドシェイク"を始めます(ピア同士の最初のハンドシェイク参照)。`version` messageとは、以下のような基本的な識別情報を含んでいるものです。

### *PROTOCOL\_VERSION*

クライアントが"会話をする"Bitcoin P2Pプロトコルバージョンを示す定数 (例えば 70002)

### *nLocalServices*

ノードがサポートしているローカルサービスのリスト、現状 `NODE_NETWORK` のみ

### *nTime*

現在時刻

### *addrYou*

このノードから見えるリモートノードのIPアドレス

### *addrMe*

ローカルノードのIPアドレス

### *subver*

このノード上で動作しているソフトウェアの種類を示すサブバージョン (例えば `"/Satoshi:0.9.2.1/"`)

### *BestHeight*

このノードのブロックチェーンのブロック高

(`version network message`の例については [GitHub](#) 参照)

ピアノードはコネクションを承認し確立するために`+verack+`を返します。場合によっては、もしコネクションのお返しにピアとして接続し直す場合は自身の `version message`を送ります。

新しいノードはどのようにしてピアを見つけるのでしょうか？一番最初に行うことは"DNSシード"にあるDNSに問い合わせることです。DNSシードはBitcoinノードのIPアドレスリストを提供するDNSサーバです。DNSシードのうちいくつかは安定的にリクエストを受け付けているBitcoinノードの静的なIPアドレスを返却しています。また、いくつかのDNSシードは、クローラや長期的に稼働しているBitcoinノードによって集められたBitcoinノードのリストからランダムにいくつかを選んで返却するカスタマイズされたBIND(Berkeley

Internet Name Daemon)で実装されています。Bitcoin Coreクライアントは5つのDNSシードを含んでいます。これらは所有者やDNSシードの実装が多様になるように構成され、確実に初期動作プロセスが実行できるようになっています。Bitcoin Coreクライアントでは、DNSシードを使うかどうかを `-dnsseed` オプションでコントロールできるようになっています(1がデフォルトで、デフォルトでDNSシードを使用するようになっています)。

DNSシードを使わない場合、初期動作プロセス中のノードにはBitcoinネットワークについて何も知らないため、少なくとも1つのBitcoinノードのIPアドレスが与えられなければいけません。その後、このノードはさらに他のノードとのコネクションを確立します。コマンドラインオプション `-seednode` は一番最初のシードBitcoinノードとコネクションを確立するために使われます。初期動作プロセスで最初のシードノードが使われた後、Bitcoinクライアントはこのシードノードとのコネクションを切り、新たに発見したピアを使うようになります。

# Node A

# Node B

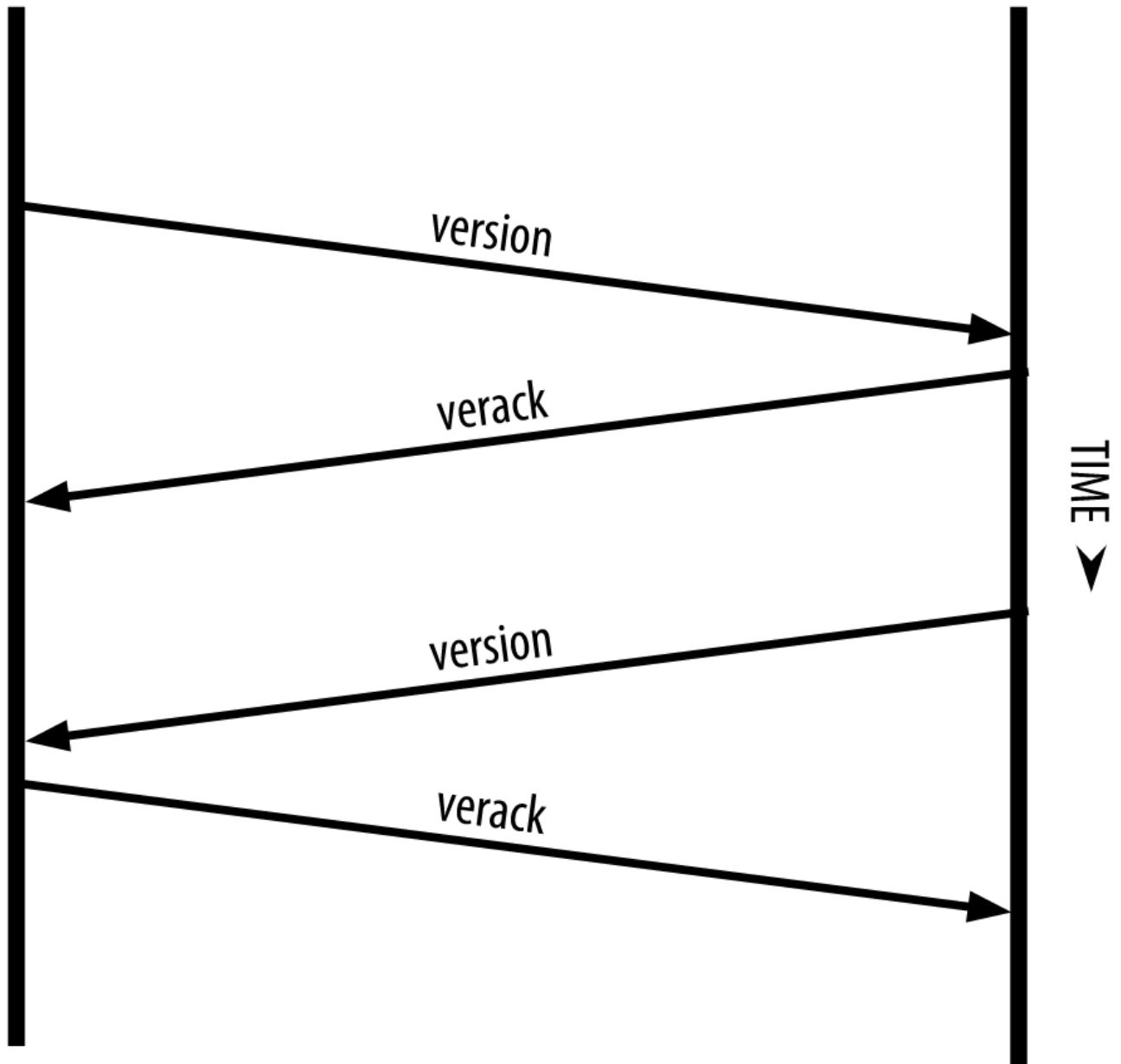


Figure 4. ピア同士の最初のハンドシェイク

一度1つまたはそれ以上のコネクションを確立すると、新しいノードは `addr` `message` という自身の IP アドレスが含まれた情報を隣接ノードに送信します。隣接ノードは次々に `addr` `message` を彼らの近くのノードに転送し、新しく接続されたノードが確実に `well known` になるようにします。また、新しく接続されたノードは `getaddr` を隣接ノードに送ることができ、他のピアの IP アドレスリストを返してもらうようにお願いすることもできます。そうすれば、ノードは接続するピアを新たに見つけることができ、その存在を他のノードに知らせることができるのです。自身の IP アドレスの伝搬と他の IP アドレスの発見図はアドレスを発見する手順を示しています。

# Node A

# Node B

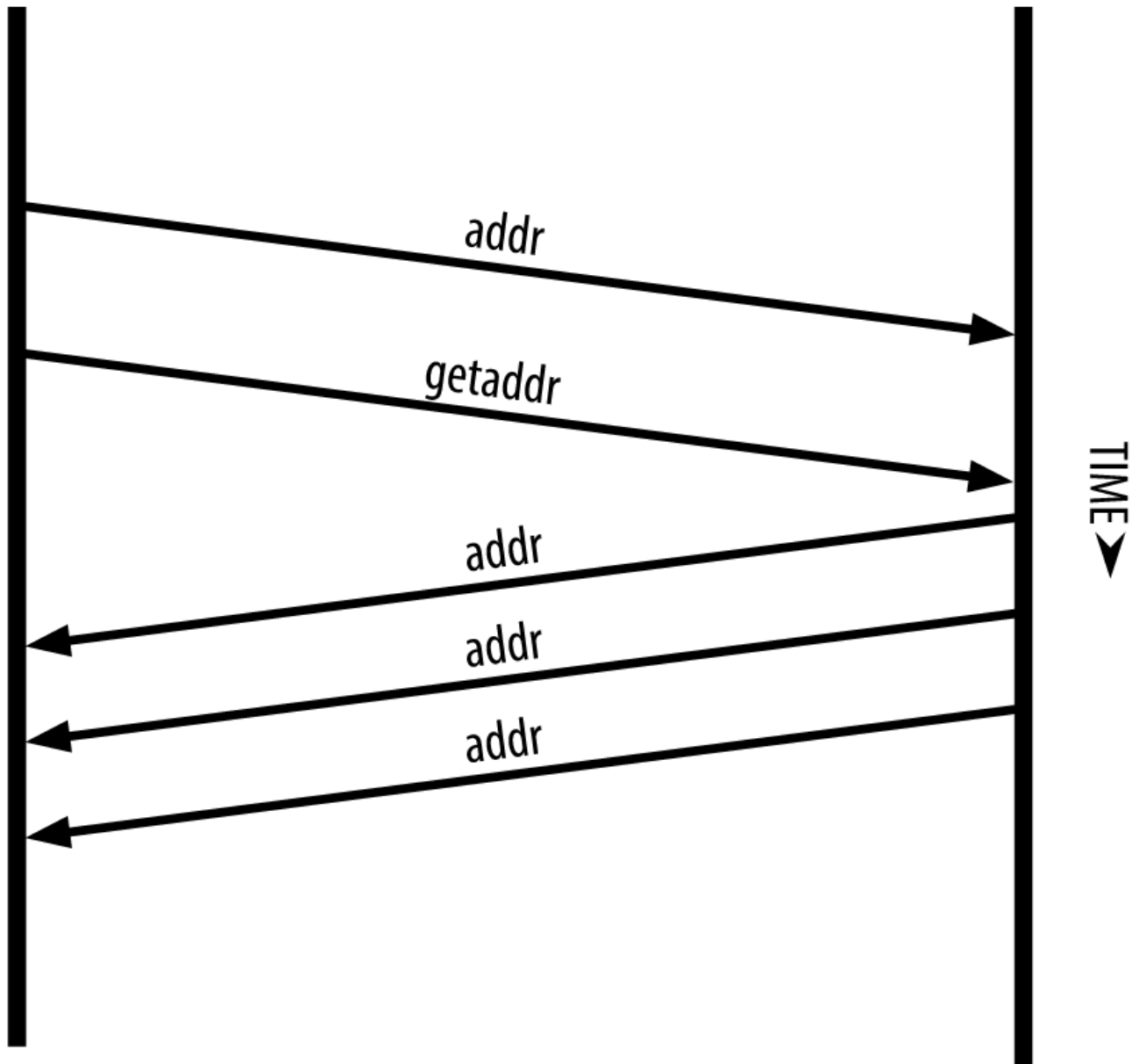


Figure 5. 自身のIPアドレスの伝搬と他のIPアドレスの発見

ノードは2、3個の異なるピアと接続し、Bitcoinネットワークへの多様なパスを確立しておかなければいけません。というのは、このパスは信用できるものではないためです。ノードは連絡なく通信が切れたり復活したりするのです。このため、他のノードの初期動作プロセス時に助けるという目的だけでなく、古い接続を失ったときのためにもノードは常に新しいノードを見つけ続けなければいけません。ただ初期動作プロセスを行うにはたった1個だけ別のノードへの接続があれば十分です。というのは、最初に繋いだノードが初期起動時に必要な情報を接続しているいくつかのピアに要求し、要求を受けたピアがまた別の接続しているピアに問い合わせるからです。また、これは多くのノードに接続してネットワークリソースを無駄に消費しないためでもあります。初期動作プロセスを終えた後ノードは直近でうまく接続を張れたピアを覚えておき、リポートしたときにすばやく覚えておいたピアと接続を張ります。以前繋がっていたどのピアも接続リクエストに答えなければ、そのノードは再度DNSシードノードを使って初期動作プロセスを行うこととなります。

Bitcoin

Coreクライアントが動作しているノードでは、

getpeerinfo

のコマンドを使ってピア接続を表示することができます。

```
$ bitcoin-cli getpeerinfo
```

```
[
  {
    "addr" : "85.213.199.39:8333",
    "services" : "00000001",
    "lastsend" : 1405634126,
    "lastrecv" : 1405634127,
    "bytessent" : 23487651,
    "bytesrecv" : 138679099,
    "conntime" : 1405021768,
    "pingtime" : 0.00000000,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.2.1/",
    "inbound" : false,
    "startingheight" : 310131,
    "banscore" : 0,
    "syncnode" : true
  },
  {
    "addr" : "58.23.244.20:8333",
    "services" : "00000001",
    "lastsend" : 1405634127,
    "lastrecv" : 1405634124,
    "bytessent" : 4460918,
    "bytesrecv" : 8903575,
    "conntime" : 1405559628,
    "pingtime" : 0.00000000,
    "version" : 70001,
    "subver" : "/Satoshi:0.8.6/",
    "inbound" : false,
    "startingheight" : 311074,
    "banscore" : 0,
    "syncnode" : false
  }
]
```

自動的に行われるピア管理ではなく特定のピアのIPアドレスを指定するために `-connect=<IPAddress>` オプションが用意されていて、1つまたは複数のIPアドレスを指定できます。このオプションが使われると、自動的にピアを見つけたりすることはせずにノードは選択されたIPアドレスにしか接続しないようになります。

コネクション上に何もトラフィックがない場合、ノードは定期的にコネクション維持のためメッセージを送ります。90分以上何の通信もしなかったコネクションがあった場合、ノードはコネクションが切れたとみなし

新しいピアを探し始めます。このように、Bitcoinネットワークは常に一時的なノードやネットワークの問題を調整しながら、中央のコントロールなしに必要なに応じて有機的に成長または縮小を繰り返します。

## フルノード

フルノードは全てのトランザクションを含む完全なブロックチェーンを管理しているノードです。もっと正確に言うと、フルノードはおそらく"フルブロックチェーンノード"と呼ばれるべきです。Bitcoinの初期の頃全てのノードはフルノードでしたが、現在はBitcoin Coreがフルブロックチェーンノードです。これは2年前からBitcoinクライアントの新しい形が導入されてきたためです。新しい形というのは完全なブロックチェーンを管理する形ではなく軽量クライアントとして動かすという形です。次の節でこの詳細を説明します。

フルブロックチェーンノードは完全で最新のブロックチェーンコピーを管理しており、これらノードは他のノードと独立に最初のブロック(genesisブロック)から最新のブロックまでを構築し検証します。また、フルブロックチェーンノードは他のノードや情報源に頼ることなく独立的かつ厳然にどんなトランザクションでも検証します。フルブロックチェーンノードはBitcoinネットワークに頼ることで新しいトランザクションのブロックをBitcoinネットワークから受け取り、それらを検証した後ブロックチェーンのローカルコピーに追加していきます。

フルブロックチェーンノードを動作させてみると分かるように、他のノードを全く信用することも頼ることもなく全てのトランザクションの検証が独立に進められていきます。フルブロックチェーンを保持するために20GB強のストレージが必要であるため、フルブロックチェーンノードを走らせるには多くのディスク容量とBitcoinネットワークからブロックチェーンをダウンロードするための2、3日の時間が必要です。

いくつかのフルブロックチェーンBitcoinクライアントの代替実装があり、別のプログラミング言語やソフトウェア設計で構築されています。しかし、主な実装はBitcoinリファレンスクライアント Bitcoin Coreであり、Satoshiクライアントと呼ばれています。Bitcoinネットワーク上の90%以上のノードがBitcoin Coreのいろいろなバージョンで動作しています。このバージョンは+/Satoshi:0.8.6/+のように表示され、"Satoshi"のあとに、前に見た+getpeerinfo+コマンドの結果に出てくるサブバージョンが付加された形になっています。

## "Inventory"の交換

フルノードがピアと接続して最初にやることは、完全なブロックチェーンを構築することです。もしノードが新しくできたもので全くブロックチェーンを持っていなければ、Bitcoin Coreに埋め込まれている1個のブロック、genesisブロック、しか知りません。このため、新しいノードは数十万ブロックものブロックをBitcoinネットワークからダウンロード&同期して、フルブロックチェーンを再構築しなければいけません。

ブロックチェーンの同期プロセスは、version messageから始まります。というのは、version messageにノードの現在のブロックチェーン高(ブロック数)を示す+BestHeight+が含まれているからです。ノードは+version+

messagesを見て相手のピアが何ブロック保持しているかを知ること、自身のブロックチェーンと比較できるようになります。次にピアノードは互いにローカルブロックチェーンの一番上のブロックハッシュ(フィンガープリント)を含む getblocks

messageを交換します。もしあるピアが持っている一番上のブロックのハッシュと受け取ったハッシュが違っていたとしても、古いブロックのハッシュと受け取ったハッシュが一致することが分かったとすると、自身の持っているブロックチェーンが相手のピアよりも長いということを知ることができます。



より長いブロックチェーンを持っているピアは他のノードよりも多くのブロックを持っています。このため、どのブロックを他のノードが欲しているかを特定することができます。他のノードと共有すべき最初の500ブロックを特定すると、これらブロックそれぞれのハッシュを `inv` (inventory、一覧) messageを使って他のノードに送ります。これらのブロックを持っていないノードは、`inv` messageにあるハッシュから自身のブロックチェーンに足りないブロックのハッシュを選んだのち `getdata` messageを使ってフルブロックデータを送ってもらうようにリクエストを出します。

例えば、あるノードがgenesisブロックしか持っていないとしましょう。genesisブロックの次の500ブロックのハッシュを含む `inv` messageを他のピアから受け取ります。このノードは接続しているピア全てに次の500ブロックに関するブロックデータ送信リクエストを送りますが、このリクエストを送りすぎることによってBitcoinネットワークが破綻しないようになっています。このノードはピアごとに何ブロックがまだ送られてきていない"送信中"状態にあるかをトラッキングし続けており、1ピアに対する送信中状態最大ブロック数 (`MAX_BLOCKS_IN_TRANSIT_PER_PEER`)

)を越えないようにチェックし続けています。この方法により、もし多くのブロックが必要だったとしても、前のデータ送信リクエストが完了してから次のリクエストを送るようになっています。これによって、ピアはペースをコントロールでき、ネットワーク全体に対する負荷も時間的に分散されます。[\[blockchain\]](#)図で見るように、受け取ったそれぞれのブロックはブロックチェーンに追加されていきます。ローカルブロックチェーンが徐々に構築されていくにつれて、より多くのブロックのリクエスト&受信がされていき、このノードのブロックチェーンがBitcoinネットワークのブロックチェーンに追いつくまでこのプロセスは続きます。

ローカルブロックチェーンと他のピアのブロックチェーンとの比較および不足ブロックの取得プロセスは、ノードがどれくらいの時間オフラインになっていても継続されます。ノードが数分オフラインであったために数ブロックが不足してしまったりしても、または数ヶ月オフラインであったために数千ブロックが不足してしまったりしても、このノードはまず `getblocks` を送り `inv` レスポンスを受け取り、足りないブロックのダウンロードを開始します。[ピアからブロックを取得することによってブロックチェーンと同期するノード](#)は、`inventory`とブロック伝搬プロトコルを示しています。

## Simplified Payment Verification (SPV) ノード

全てのノードがフルブロックチェーンを保持する能力を備えているわけではありません。多くのBitcoinクライアントはディスク容量や計算スピードが限られているスマートフォンやタブレット、組み込みシステムなどのデバイス上で動作するように設計されています。このようなデバイスに対しては、フルブロックチェーンを保持することなしに前節で説明したプロセスを実行できるように *simplified payment verification* (SPV)が使われます。この方法を用いるクライアントをSPVクライアントまたは軽量クライアントと呼びます。このクライアントが多く採用されるにつれて、SPVノードがBitcoinノードの主要な形、Bitcoinウォレット、になっています。

SPVノードはブロックヘッダだけをダウンロードしトランザクション自体はダウンロードしません。トランザクションがないヘッダだけのブロックチェーンはフルブロックチェーンの1/1000くらいの大きさになります。SPVノードはBitcoinネットワーク上の全てのトランザクションについて知っているわけではないため、使用可能な全てのUTXOを構築できません。SPVノードは、必要に応じてブロックチェーンの関連した部分のみを提供するピアに頼るという方法を用いてトランザクションを検証します。

**Node A**

**Node B**

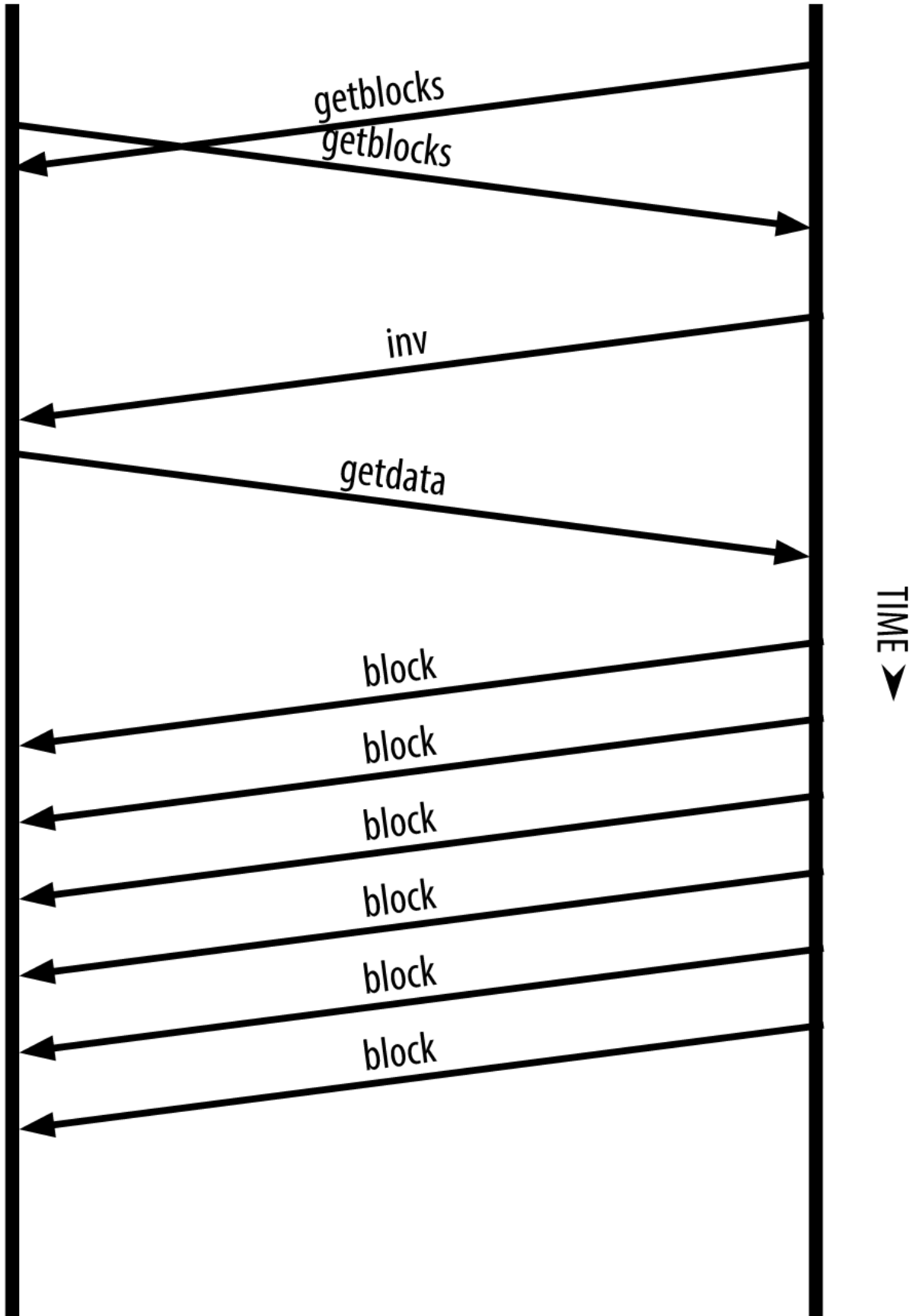


Figure 6. ピアからブロックを取得することによってブロックチェーンと同期するノード

アナロジーとして、フルノードは行ったことのない町の全てのストリート、住所についての詳細な地図を持っている観光客に似ています。これに対して、SPVノードはメイン通りしか知らず行き当たりばったりで進む観光客のようなものです。両方の観光客ともメインストリートが確認できる点は同じですが、地図を持っていない観光客はメインストリートにどんな横道があるか、他にどんなストリートがあるかは分かりません。単に2

3 Church Streetというストリートにいただけでは、地図を持っていない観光客は同じ名前のストリートが同じに町に他にも多くあるかどうか、目の前のストリートが行きたいストリートなのかどうかを知ることはできないのです。地図を持たない観光客が取れる最も良い方法は、十分な数の人々に尋ねることです。そのうちの何人かが彼をだまそうとしないことを祈りましょう。

simplified payment verificationはブロックチェーンの 高さ の代わりにブロックチェーンの 深さを参照することでトランザクションを検証します。フルブロックチェーンノードが完全に検証された数千ブロックのブロックチェーンや全てのトランザクションを構築する一方、SPVノードは全てのブロックチェーン(しかし全てのトランザクションではないです)とこのSPVノードと関連のあるトランザクションだけを検証します。

例えばブロック300,000にあるトランザクションを調べる場合、フルノードは300,000個のブロックをgenesisブロックまで結びつけUTXOのフルデータベースを構築しUTXOが使用されていないことを確認することでトランザクションを検証していきます。SPVノードはUTXOが使用されていないかどうか検証できません。その代わりに、SPVノードは `merkle path`([\[merkle\\_trees\]](#)参照)を使うことでトランザクションとこのトランザクションを含んでいるブロックとの間を結びつけていきます。ブロック300,000のトランザクションを使用する場合、SPVノードは6個のブロック、300,001番目から300,006番目まで、を確認するまで待ちます。これはネットワーク上の他のノードが300,000番目のブロックを受け取りそしてさらに6ブロック以上が上に作られることで、トランザクションが二重に使用されたものではないことが代理ノードによって証明されることを待つためです。

あるブロックに、あるトランザクションがあるかどうかをSPVノードに問い詰めることはできません。SPVノードは、代理ノードに対してmerkle path証明を要求し、ブロックチェーンにあるproof of workを検証することでブロックの中のトランザクションの存在を確認できます。しかし、トランザクションの存在はSPVノードには"隠されて"いるのです。SPVノードは確実にトランザクションは存在することを証明できますが、同じUTXOの二重使用(double-spend)のようなトランザクションが存在しているかどうかは検証できません。なぜなら、全てのトランザクションの記録を持っている訳ではないからです。これらの弱点は、DOS攻撃または二重使用攻撃に利用されてしまいます。これに対抗するために、SPVノードはランダムにいくつかのノードと接続するようしておく必要があります。これは、できるだけ信用できるノードと接続するようしておくためです。ランダムに接続することで、ネットワーク分割攻撃またはSybil攻撃を回避することができます。というのは、SPVノードが攻撃者のノードまたは攻撃者のネットワークにのみに接続してしまうと、信用できる正しいBitcoinネットワークに接続できなくなってしまうためです。

実用上、バランスよくコネクションを持っているSPVノードは十分に安全で、必要なリソース量、実用性、安全性のよいバランスがとられています。しかし、絶対に確実なセキュリティという点では、フルブロックチェーンノードが最も良いです。

**TIP**

フルブロックチェーンノードは、あるトランザクションより下の全てのブロックのチェーンをチェックすることでこのトランザクションを検証します。これは、このUTXOが未使用であることを保証するためです。一方、SPVノードはこのブロックよりも上の一握りだけのブロックを確認することで、このブロックがどれだけ深く埋められているかを確認しています。

ブロックヘッダを得るために、SPVノードは `getblocks message`の代わりに `getheaders message` を使います。`getheaders message` を受け取ったピアは2,000個までのブロックヘッダを1個の `headers message` で返送します。このプロセスはフルノードがブロックを集めるプロセスと同じです。また、SPVノードはピアが送信したブロックやトランザクションをフィルタリングしています。関連あるトランザクションを取得する際には `getdata request` を使います。ピアはトランザクションが含まれている `tx message` を生成し返却します。[SPVノードのブロックヘッダ同期図](#)はブロックヘッダの同期を示しています。

# Node A

# Node B

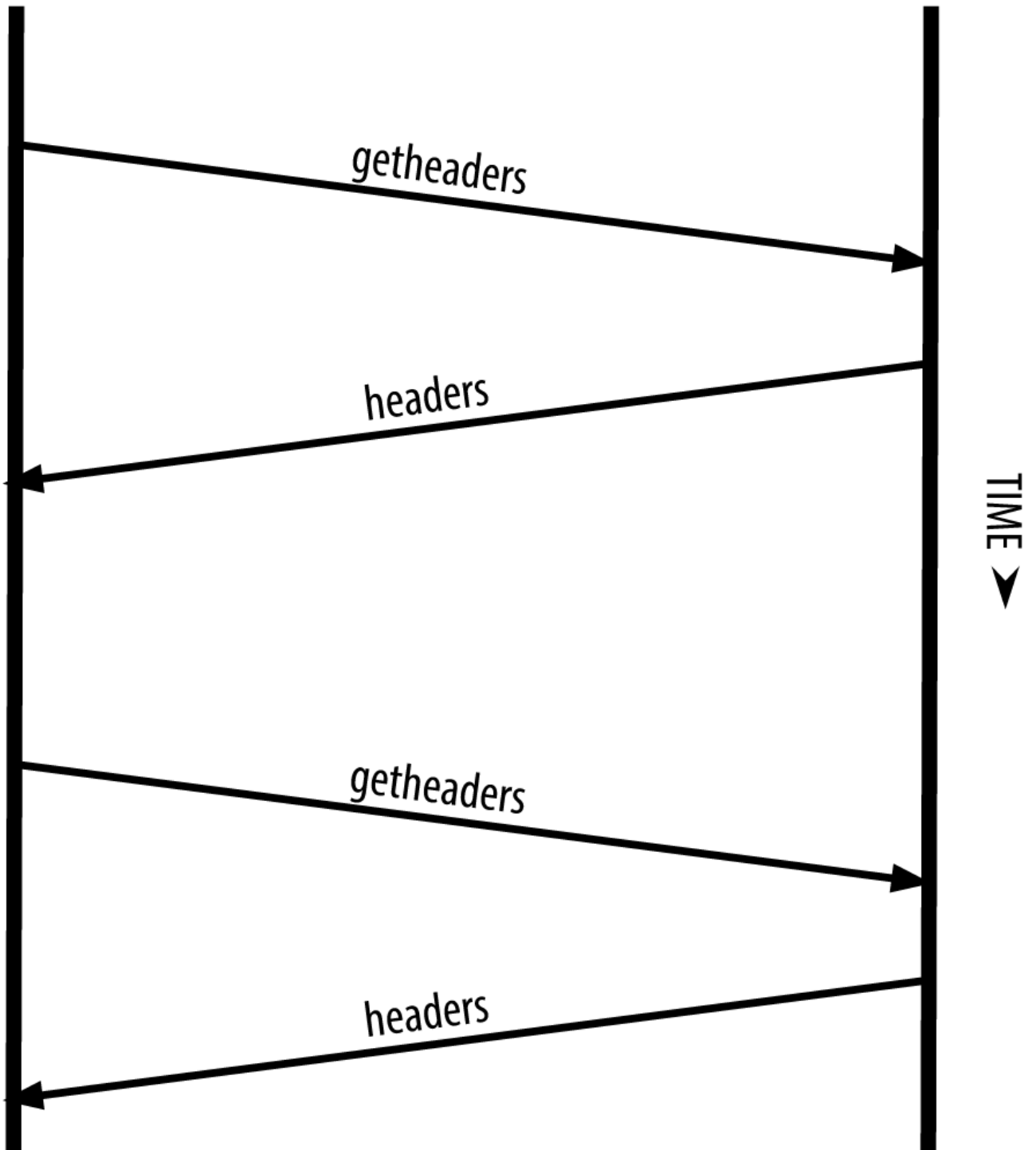


Figure 7. SPVノードのブロックヘッダ同期

SPVノードは関連あるトランザクションのみを取得するので、プライバシーリスクが生じてしまいます。フルブロックチェーンノードと違って、全てのトランザクションを取得するわけではなく関連あるデータだけを取得するためウォレットのBitcoinアドレスがもれてしまうのです。例えば、第三者のモニタリングツールはSPVノード上のウォレットからリクエストされたトランザクションを全て追跡することができ複数のBitcoinアドレスをウォレットのユーザと結びつけることができます。

SPV/軽量ノードが導入されたあと少しして、Bitcoinの開発者たちは `bloom` `filter` と呼ばれるプライバシーを漏らさない機能を追加しました。`bloom filter`は、SPVノードと関連あるBitcoinアドレスがどれかを漏らすことなくトランザクションの部分集合を取得する方法です。ただし、このフィルタリングメカニズムは正確なものではなく確率を利用したものです。

## Bloom Filter

`bloom filter`は確率的探索フィルタで、欲しいパターンを正確に特定しなくてもよい方法です。`bloom filter`はプライバシーを漏らさないような探索パターンを作り、SPVノードは特定のパターンに合ったトランザクションが含まれているかを他のピアに確認することができるのです。

前の節でのアナロジーとして、地図を持っていない観光客は人にある住所 "23 Church St." への方向を尋ねます。もし彼女がこのストリートへの方向を知らない人に尋ねたら、情報を得ることなくうっかり彼女が行こうとしているところを明かしてしまうことになるのです。`bloom filter`は「この近くにRCHで終わるストリートはありますか？」と尋ねるようなものです。このような質問をすることで、少しは行き先を明かさずに済みます。このテクニックを使って、さらに詳しい質問「この近くにURCHで終わるストリートはありますか？」、またはもっと粗い質問「この近くにHで終わるストリートはありますか？」によって観光客は行きたい場所を特定していくことができるかもしれません。質問の仕方を変えることで、観光客は正確な返答ではありませんが住所を特定できる可能性のある多くの結果とプライバシーを守ることができるのです。もっと直接的に質問すれば、もっと少ない質問で行きたい場所に行けますが、プライバシーを失ってしまいます。

`bloom filter`は、この例と同じことをSPVノードがトランザクションを探るときに使えるようにし、正確性とプライバシーのバランスを取ることができるようにします。より正確な**`bloom filter`**は正確な結果を返しますが、どのBitcoinアドレスをウォレットが使っているかを明かすことでプライバシーを犠牲にします。代わりに、より粗い**`bloom filter`**はこのBitcoinノードに関係しないより多くのトランザクションに関する多くのデータを返しますが、プライバシーを保てるようにします。

SPVノードは**`bloom filter`**を"空"の状態初期化しますが、この状態ではどんなパターンもマッチしません。次にSPVノードはウォレットが持っている全てのBitcoinアドレスのリストを作成し、それぞれのBitcoinアドレスに紐づいたトランザクションアウトプットごとに探索パターンを作成します。通常、探索パターンは `pay-to-public-key-hash script` です。これは、`public-key-hash(Bitcoinアドレス)`への支払いをするトランザクションに提供されるlocking scriptです。もしSPVノードが `P2SH` アドレスの残高をトラッキングしているのであれば、探索パターンは**`pay-to-public-key-hash script`**の代わりに `pay-to-script-hash script` になります。次に、SPVノードは**`bloom filter`**が探索パターンを認識できるようにこれらの探索パターンを**`bloom filter`**に追加します。最後に、SPVノードは**`bloom filter`**をピアに送り、ピアは送られてきた**`bloom filter`**を使ってどのトランザクションが探索パターンにマッチするかを調べます。

`bloom filter`はN個のビット列とM個のハッシュ関数で構成されています。ハッシュ関数はいつも1からNの間の値を生成するようになっており、この数はビット列の場所に対応しています。どのノードでも同じハッシュ関数を使い特定の入力に対して同じ結果を得られるように、ハッシュ関数は決定性的なものになります。`bloom filter`の長さ(N)とハッシュ関数の数(M)として違ったものを選ぶことで**`bloom filter`**をチューニングすることができ、正確さのレベルおよびプライバシーの確保度合いを調整できます。

16bitのフィールドと3つのハッシュ関数を持った極端にシンプルにしたbloom filterの例図では、bloom filterがどのように動くかのデモンストレーションとしてとても小さい16個のビット列と3個のハッシュ関数を使っています。

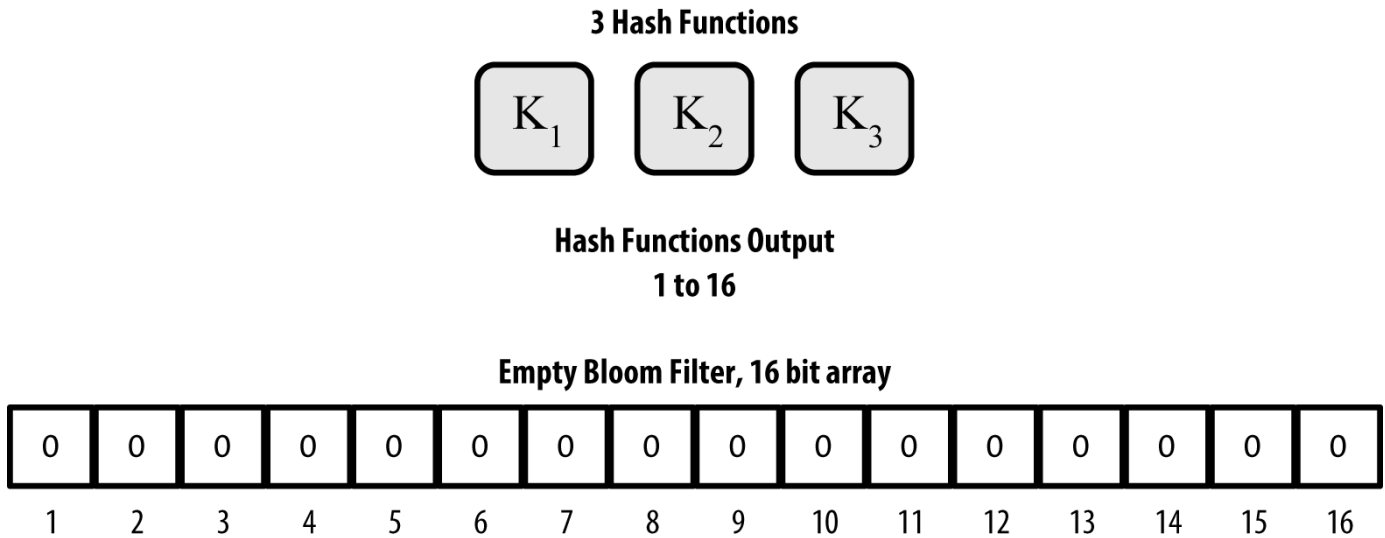


Figure 8. 16bitのフィールドと3つのハッシュ関数を持った極端にシンプルにしたbloom filterの例

bloom filterはまず全てのビット列が0のなるように初期化されます。bloom filterにパターンを追加するために、パターンをそれぞれのハッシュ関数で次々にハッシュ化しbloom filterに追加していきます。インプットパターンを最初のハッシュ関数に通して1からNまでの間の数を得ます。この数に対応したビット列(1からNまでのindexが振ってある)のビットを見つけ 1を立てます。次のハッシュ関数に対しても同様に行いM個のハッシュ関数全てに対して行くと、ビットが 0から 1に変わった模様としてトランザクションに対する探索パターンがbloom filterに "記録" されます。

前に示したシンプルなbloom filterにパターン"A"を与えた場合図はパターン"A"を16bitのフィールドと3つのハッシュ関数を持った極端にシンプルにしたbloom filterの例図のbloom filterに記録した例です。

2つ目のパターンを追加するプロセスは、1つ目のプロセスを繰り返すだけです。2つ目に対してもそれぞれのハッシュ関数を使ってハッシュ化し、ビット列の特定の場所のビットに 1を立てることでパターンを記録します。多くのパターンを記録していくにつれて、すでに 1のビットが立っている場所をもう一度 1に立てようとするかもしれませんが、この場合このビットは変化しません。本質的に、bloom filterに多くのパターンを記録すればするほど 1 が立っている場所が増え飽和していき、bloom filterの正確さは衰えていきます。これが、bloom filterが確率的なデータ構造、パターンを追加すればするほど正確性が失われる、になっている理由です。正確さはパターンの数が多くなればなるほど減り、逆に、ビット列の大きさ(N)とハッシュ関数の数(M)が大きくなればなるほどこの減り度合いを抑制できます。より大きなビット列と多くのハッシュ関数を使うことで多くのパターンをより正確に記録できるのです。

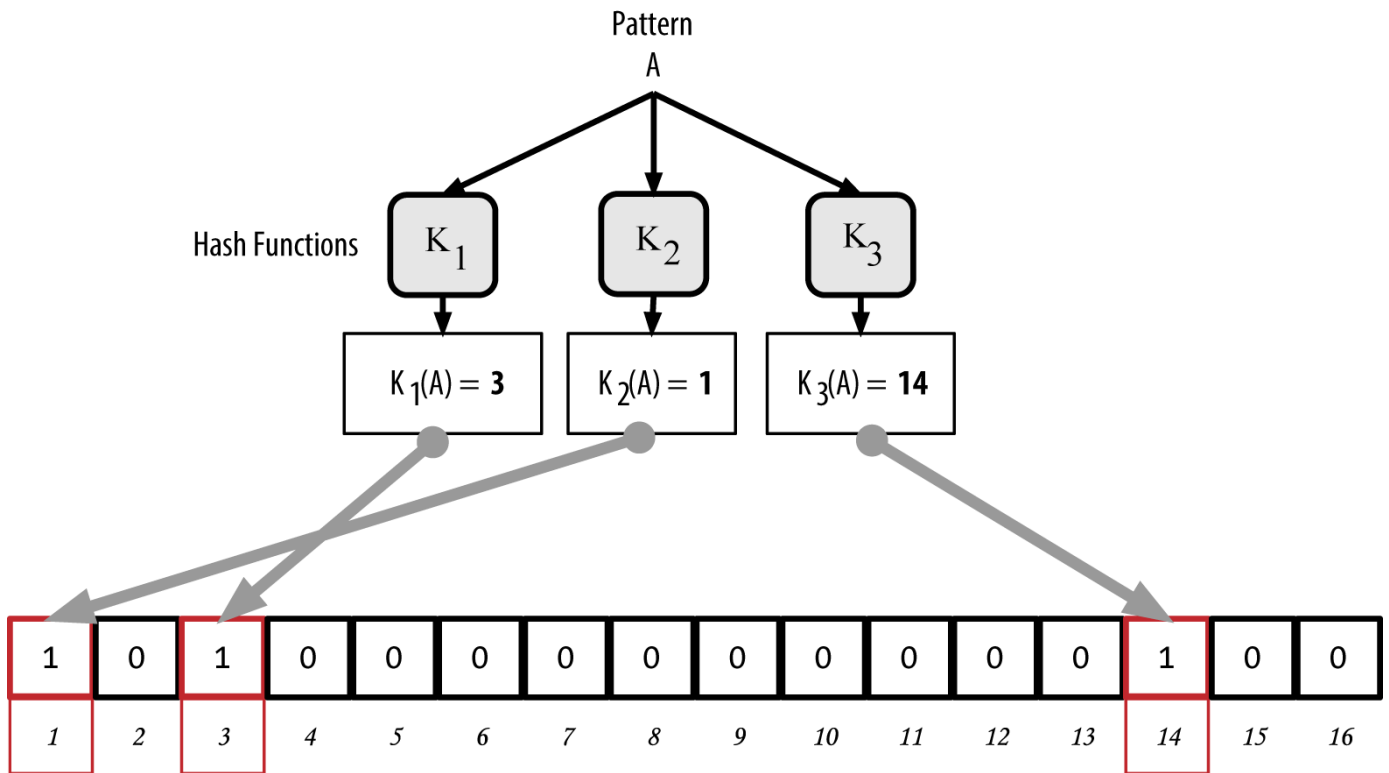


Figure 9. 前に示したシンプルなbloom filterにパターン"A"を与えた場合

前に示したシンプルなbloom filterに2番目のパターン"B"を与えた場合図はパターン"B"をbloom filterに記録する例です。

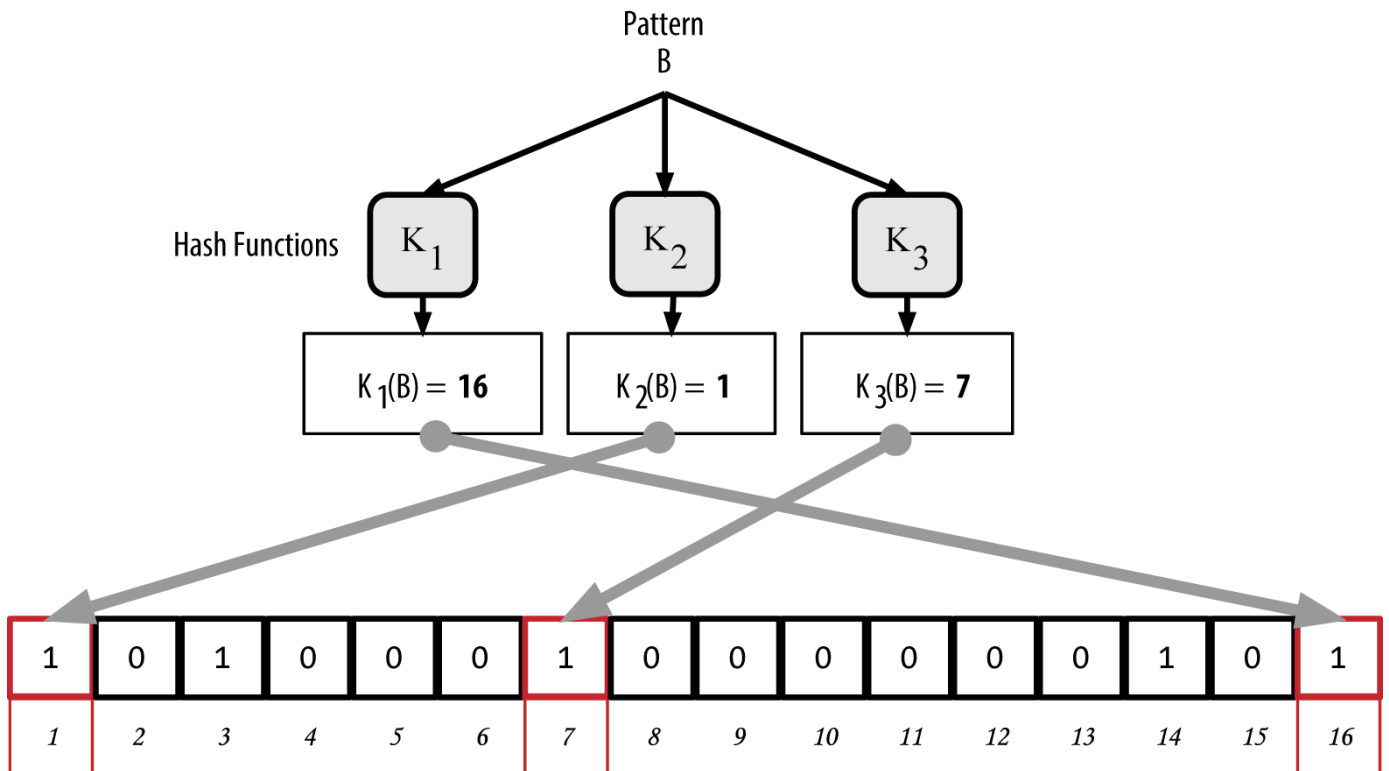


Figure 10. 前に示したシンプルなbloom filterに2番目のパターン"B"を与えた場合

あるパターンがbloom



filterの一部にあるかどうかチェックするために、このパターンをそれぞれのハッシュ関数でハッシュ化し得られたビットパターンとbloom filterのビット列を比較します。あるパターンのビットパターンの中で 1 になっている場所がbloom filterのビット列でも 1 になっていれば、あるパターンが おそらく bloom filterに含まれているだろうと推察できます。bloom

filterのビット列のあるビットは複数のパターンによる重複で 1 になっているかもしれないので、答えとしては確実ではないですが、むしろ確率的な答えになります。簡単に言うと、bloom filterは"たぶん、含まれる"と答えるだけです。

bloom filterを使ってパターン"X"が存在するかチェック。その結果は確率的な陽性、つまり"たぶんある"。図はパターン"X"がbloom filterに含まれているかチェックする例です。対応したビットは 1 になっており、よっておそらくパターン"X"を含むということになります。

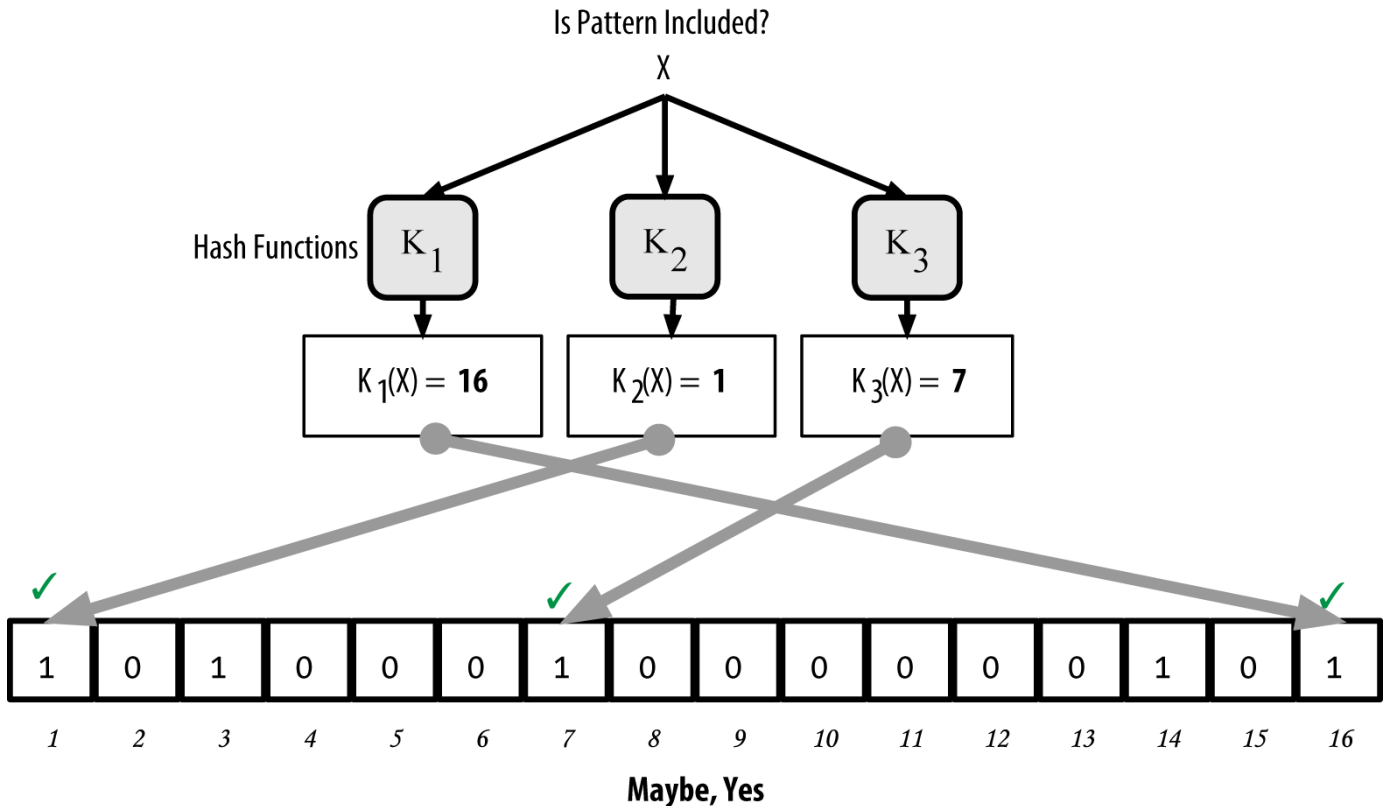


Figure 11. bloom filterを使ってパターン"X"が存在するかチェック。その結果は確率的な陽性、つまり"たぶんある"。

逆に、あるパターンがbloom filterに含まれていないということをチェックする場合は、対応したbloom filterのビット列のどれか1つが 0 であることを確認すればよく、このことであるパターンがbloom filterには含まれていないということを証明することができます。含まれていないというチェックに対しては確率的ではなく、確実なものです。簡単に言うと、bloom filterは"絶対に含まれない!"と答えることができます。

bloom filterを使ってパターン"Y"が存在するか確認。その結果は正確な陰性、つまり"確実でない!"。図はパターン"Y"がbloom filterに含まれているかチェックする例です。対応したビットの1つが 0 になっており、よってパターン"Y"は全体に含まれないということになります。

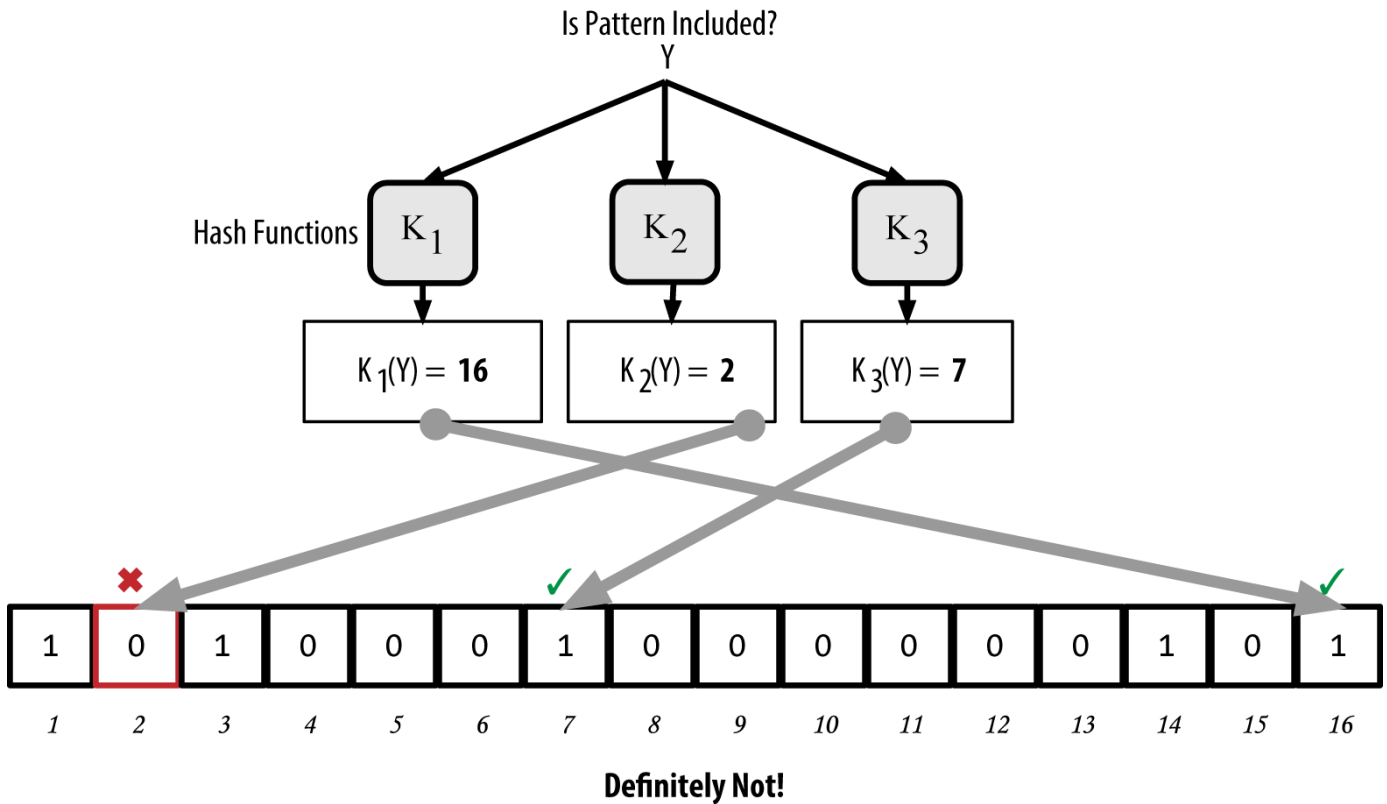


Figure 12. bloom filterを使ってパターン"Y"が存在するか確認。その結果は正確な陰性、つまり"確実にない！"。

bloom filterのBitcoinでの実装は Bitcoin Improvement Proposal 37 (BIP0037) に記述されています。[appdxbitcoinimpproposals]を参照するか、または GitHub に行ってみてください。

## Bloom FilterとInventory更新

bloom filterはSPVノードが受け取るトランザクション(およびそれらを含んでいるブロック)をフィルタリングするために使われます。SPVノードはSPVノードのウォレットにあるBitcoinアドレスのみにマッチするフィルタを作成します。SPVノードはbloom filterを含んでいる filterload messageをピアに送ります。bloom filterが送られると、ピアはそれぞれのトランザクションのアウトプットを送られてきたbloom filterでチェックします。bloom filterにマッチしたトランザクションだけがSPVノードに送られます。

getdata messageに対するレスポンスとして、ピアはbloom filterにマッチしたブロックのヘッダとマッチしたトランザクションそれぞれに対するmerkle path([merkle\_trees]参照)を含む merkleblock message をSPVノードに送ります。ピアはまたbloom filterにマッチしたトランザクションを含む tx message も送ります。

SPVノードが新たにパターンを増やす場合は filteradd message をピアに送ることでパターンをbloom filterに追加できます。またbloom filterを削除するためには、 filterclear message をピアに送ります。bloom filterからあるパターンだけを削除することはできないので、この場合SPVノードは一度bloom filterを削除してから新しいbloom filterを送り直します。

# トランザクションプール

Bitcoinネットワーク上のほとんどのノードは メモリプール または トランザクションプールと呼ばれる未検証トランザクションの一時リストを持っています。ノードはこのプールを使って、Bitcoinネットワークに伝わっていてもまだブロックチェーンに含まれていないトランザクションをトラッキングしています。例えば、ウォレットを持っているノードは、Bitcoinネットワークに伝わっていてもまだ承認されていないウォレットへの入金トランザクションを一時的にこのトランザクションプールに保持しています。

トランザクションが到着し検証されると、これらはトランザクションプールに追加されたあと隣接ノードに中継され、そしてBitcoinネットワーク上を伝搬していきます。

いくつかのノードはorphan(孤児)になっているトランザクションを入れておく別のプールも持っています。もしトランザクションインプットがまだノードが知らないトランザクションを参照していた場合、親トランザクションが到着するまでorphanトランザクションは一時的にorphanプールに保存されます。

トランザクションがトランザクションプールに追加されるとき、ノードはorphanプールにあるトランザクションが追加されるこのトランザクションアウトプットを参照していないかチェックします。もし参照していれば、orphanプールから削除してトランザクションプールに追加されます。このプロセスはorphanプールにあるトランザクション全てに対して行われ、トランザクションが到着することが起点となり全トランザクションのチェーンが再構築されていきます。

トランザクションプールもorphanプール(もし実装されていれば)もローカルメモリに保持され、永続的なストレージには保存されません。これらは常にBitcoinネットワークからmessageが届くごとに書き変わっていくため、むしろローカルメモリのほうがよいのです。ノードが起動するときどちらのプールも空になっていて、Bitcoinネットワークからトランザクションが届くと次第に混み合ってきます。

いくつかのBitcoinクライアントの実装ではUTXOデータベースまたはUTXOプールも管理しています。このプールはブロックチェーン上の全ての未使用アウトプットを集めたものです。"UTXOプール"という名前の響きがトランザクションプールと似ていますが、別のデータの集まりです。トランザクションプールやorphanプールと違って、UTXOプールの初期状態は空ではなく最初から数百万個の未使用トランザクションアウトプット(2009年からのトランザクションアウトプット)を持っています。UTXOプールはローカルメモリまたは永続ストレージのデータベースに保持されています。

トランザクションプールとorphanプールはそれぞれのノードでの状態が異なりノードがいつ起動したか再起動したかによって変わってきますが、UTXOプールはBitcoinネットワーク内で合意されたものであり、ノードごとの違いはわずかです。さらに、トランザクションプールとorphanプールは未検証トランザクションのみを含み、UTXOプールは検証済アウトプットのみを含みます。

## アラートメッセージ

アラートメッセージは稀にしか使われない機能ですが、それにも関わらずほとんどのノードに実装されています。アラートメッセージはBitcoinの"緊急放送システム"で、コアのBitcoin開発者たちが緊急メッセージを全てのBitcoinノードに送れます。この機能を使うことで、コアのBitcoin開発者たちがBitcoinネットワーク内の重大な問題を全てのBitcoinユーザに通知できるようになっています。例えばユーザが何らかのアクションをとらなければならないクリティカルなバグのようなものを通知するためです。このアラートシステムはほんの数回だけしか使われておらず、最も大きなものとしては2013年初期にあったクリティカルなデータベースバグのときで、ブロックチェーンの分岐が起きてしまったときに使用されています。

アラートメッセージは

alert

messageによって伝搬されます。アラートメッセージは以下にあるフィールドを含んでいます。

#### *ID*

アラートを一意に指定するID

#### *Expiration*

アラートが失効するまでの時間

#### *RelayUntil*

アラートが中継されなくなるまでの時間

#### *MinVer, MaxVer*

アラートが適用されるBitcoinプロトコルバージョンの範囲

#### *subVer*

アラートが適用されるクライアントバージョン

#### *Priority*

アラートの優先レベル、現在使用されていない

アラートは公開鍵で暗号的に署名されています。公開鍵に対応した秘密鍵は何人かの選ばれたコア開発メンバーによって保持されています。このデジタル署名によってBitcoinネットワークを嘘のアラートが伝搬しないようになっています。

アラートメッセージを受け取ったノードはそれを検証し、有効期間をチェックし、全てのピアにアラートメッセージを伝搬します。このため、Bitcoinネットワーク上をすばやく伝搬することができるようになっています。

Bitcoin Coreクライアント内に、このアラートを表示することができるコマンドラインオプション `-alertnotify` があり、アラートを受け取ったときに実行する動作を指定できます。アラートメッセージは `alertnotify` コマンドにパラメーターとして渡されます。よくある設定は、`alertnotify` コマンドにノードの管理者にアラートメッセージを含むEメールを送る設定です。このアラートはまたグラフィカルなユーザインターフェイス(bitcoin-Qt)が動いていればポップアップダイアログとしても表示されます。

Bitcoinプロトコルの他の実装では、アラートを別の形で受け取られているかもしれません。多くのハードウェアに埋め込まれたBitcoinマイニングシステムではアラートメッセージ機能は実装されていません。というのは、ユーザインターフェイスがないためです。このようなマイニングシステムを動作させているマイナーは、マイニングプールオペレーターを通してアラートを受け取るか、アラートのためだけに軽量ノードを動作させておくことを強く推奨します。

# ブロックチェーン

## イントロダクション

ブロックチェーンのデータ構造は、トランザクションを含むブロックが数珠つなぎに並べられており、一つ前のブロックへのリンクを持つような構造になっています。

ブロックチェーンはフラットファイルまたはシンプルなデータベース内に保持されており、Bitcoin Coreクライアントの場合、GoogleのLevelDBを使ってブロックチェーンのメタデータが保存されています。ブロックは"一つ前"のブロックへのリンクを持っています。このため、ブロックチェーンはよく垂直スタックとして表現されることが多いです。この垂直スタックは、ブロックが積み重ねられ下のブロックがその上にあるブロックの土台となっているようなものです。このような垂直に積み重ねていく例えから、一番最初のブロックからあるブロックまでの距離を表現するのに"高さ(height)"という言葉を使い、一番最後のブロックを"トップ(top)"または"先端(tip)"というような言葉で表します。

ブロックチェーン内のそれぞれのブロックは、そのヘッダにSHA256暗号的ハッシュアルゴリズムを適用することで得られるハッシュ値をIDとして持ちます。また、それぞれのブロックはヘッダの"previous block hash"のフィールドを通して一つ前のブロックを参照しており、この参照されているブロックを `_親_ブロック` と呼びます。別の言い方をすると、それぞれのブロックは親ブロックのハッシュを自身のヘッダに持っているのです。ブロックをその親ブロックに繋いでいくことで形成されるハッシュの列は、最終的には一番最初に生成された `_genesisブロック_` と呼ばれるブロックにまで繋がっていきます。

ブロックは必ず1つの親ブロックを持ちますが、一時的に1つの親ブロックに複数の子ブロックができるケースがあります。これは、それぞれの子ブロックヘッダの"previous block hash"フィールドに同一の親ブロックのハッシュ値が格納されている状態です。このようなケースは複数のマイナーがほぼ同時に新しいブロックを採掘した場合に発生し、ブロックチェーンは一時的に"フォーク(分岐)"することになります(詳細は [\[forks\]](#) を参照)。しかし最終的には複数の子ブロックの中の1つがブロックチェーンの一部となり、フォークは解消されることになります。ブロックが複数の子ブロックを持ったとしても、それぞれのブロックの親ブロックは必ず1つです。これは、ヘッダ内の"previous block hash"フィールドに単一の親ブロック情報が入っているためです。

"previous block hash"フィールドがブロックヘッダにあるため、現在のブロックのハッシュは"previous block hash"フィールドの影響を受けます。親ブロックのハッシュ値が変更された場合は子ブロックのハッシュ値が変わってしまうのです。親ブロックの内容を修正すると親ブロックのハッシュ値が変わり、親ブロックのハッシュ値が変わると子ブロックの"previous block hash"フィールドの値が変わり、結果、子ブロックのハッシュ値も変わります。同様に子ブロックのハッシュ値が変更されることで孫ブロック、孫ブロックの変更で曾孫ブロックのハッシュ値が変更というように続きます。

つまり、多くの世代が後に続くあるブロックの内容をなんらかの理由で変更するためには、その後の世代の全てのハッシュ値を再計算しないといけないということになります。この再計算は非常に計算量を要するために、古い世代のブロックは誰にも変更が出来ず、この変更不可能性がBitcoinの安全性の鍵となっています。

ブロックチェーンは地層や氷河のようなものです。表層部分は季節や気候の変化によって変化しやすいものの、十数センチ下の層では状態はより安定し、さらに数十メートル下の層では数百万年前の状態がそのままの形で残っているのが見てとれます。ブロックチェーンでも同様です。ほんの最近の2、3ブロックはフォークによる再計算を行って書き換えられるかもしれませんが、先頭の6ブロックは表土の2、3インチのようなもので、一度6ブロックより深くのブロックチェーンの奥に入ってしまうとブロックはほぼ変更されないようになりま

す。100ブロックまで来るともっと安定的になり、この100ブロック目にあるcoinbaseトランザクション(新たにマイニングされたbitcoinが含まれているトランザクション)を使うことができるようになります。3000~4000ブロックまで奥に入ってしまう(1ヶ月分のブロックが積み重なる)と、このブロックは歴史に刻まれ確固たるものになります。ここまで行くと、現実には生じるどんな目的の支払いにも使えるレベルでしょう。ところが一方で、ブロックチェーンは常により長いブロックチェーンによって置き換えられる可能性があり、どんなにブロックが積み重ねられてもブロックが書き換えられる可能性は常にあります。この可能性は時間が経つにつれて徐々に減っていき、完全に可能性が0になるには無限の時間がかかることになります。

## ブロックの構造

ブロックは、公開元帳であるブロックチェーンに含めるいくつかのトランザクションを集めたコンテナ型データ構造になっています。ブロックはメタデータを含むヘッダと、ブロックのサイズの大半を占める大量のトランザクションのリストによって構成されています。ブロックのヘッダサイズは80バイトである一方、1つのトランザクションのサイズは最低でも250byteあり、平均して500個のトランザクションが1つのブロックに含まれます。つまりブロック全体のサイズは、ヘッダサイズの1000倍程度になります。

[ブロック構造](#)にブロックの構造を示しています。

Table 1. ブロック構造

| サイズ              | フィールド名              | 説明                                   |
|------------------|---------------------|--------------------------------------|
| 4byte            | Block Size          | この次のフィールドからブロックの最後までのデータサイズ(byte単位)。 |
| 80byte           | Block Header        | nonceなどいくつかのフィールドがこのヘッダフィールドに含まれます。  |
| 1-9byte (VarInt) | Transaction Counter | ブロックに含まれるトランザクション数                   |
| 可変サイズ            | Transactions        | ブロックに記録されるトランザクションのリスト               |

## ブロックヘッダ

ブロックのヘッダには3種類のメタデータで構成されています。1つ目は一つ前のブロックのハッシュ値であり、ブロックチェーンの中で前ブロックを示す情報になります。2つ目は *difficulty*、*timestamp*、*\_nonce\_* といったマイニング競争に関係するメタデータです。マイニングについては[\[ch8\]](#)に詳しく説明します。そして3つ目は、ブロック内の全トランザクションデータを効率的に要約するためのデータ構造であるmerkle treeのルートハッシュです。 [ブロックヘッダ構造](#) にブロックヘッダの構造を示します。

Table 2. ブロックヘッダ構造

| サイズ    | フィールド名              | 説明                  |
|--------|---------------------|---------------------|
| 4byte  | Version             | ソフトウェア/プロトコルバージョン番号 |
| 32byte | Previous Block Hash | 親ブロックのハッシュ値         |

| サイズ    | フィールド名            | 説明                                    |
|--------|-------------------|---------------------------------------|
| 32byte | Merkle Root       | ブロックの全トランザクションに関するmerkle treeのルートハッシュ |
| 4byte  | Timestamp         | ブロックの生成時刻(Unix時間)                     |
| 4byte  | Difficulty Target | ブロック生成時のproof of workのdifficulty      |
| 4byte  | Nonce             | proof of workで使われるカウンタ                |

マイニングの過程で使われる、nonce、difficulty、timestampの詳細については[\[ch8\]](#)章で説明します。

## ブロック識別子：ブロックヘッダハッシュとブロック高

最も重要なブロックの識別子はブロックヘッダに対してSHA256アルゴリズムを2回適用して生成される暗号的ハッシュ、つまりデジタルフィンガープリントです。この32byteのハッシュ値は `<em>ブロックハッシュ</em>` または、より正確に `<phrase role="keep-together">_ブロックヘッダハッシュ_</phrase>` と呼ばれます。これはハッシュ値の計算にヘッダのデータのみ利用されていることによります。

例えば `000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f` は一番最初に生成されたブロックのブロックハッシュです。ブロックハッシュは各ブロックにユニークに与えられる識別子であり、各ブロックのヘッダのみの情報から個々のノードごとに独立に計算される識別子となっています。

ブロックがBitcoinネットワーク内で伝送される際も、ブロックチェーンとしてストレージ内に格納される場合にも、ブロックがそのデータ構造内に自身のブロックハッシュを持っていないのです。代わりに、各ノードがブロックを受け取ったときにそのブロックのブロックハッシュを計算します。ただブロックハッシュは各ノード内でブロックのメタデータの取り出しを高速化するための索引情報として別テーブルに保持されていることはあります。

ブロックを識別するもう1つの方法は、 `<phrase role="keep-together"><em>ブロック高</em></phrase>` であり、これはブロックチェーン内での位置を表します。最初に作られたブロックのブロック高は0(ゼロ)であり、これは先ほど `+000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f+` というブロックハッシュ値で参照したブロック `<phrase role="keep-together">_ブロック高_</phrase>` と同一のブロックを指し示します。つまり

1つのブロックはブロックハッシュと、ブロック高の2通りの方法で識別することが可能です。後続のブロックは最初のブロックの"上に"積み重ねられ、積み重ねられる度にブロック高は1つ"高く"なっていきます。これはちょうど重なり合って積み重ねられた箱のようなものです。2014年1月1日現在でブロック高は約278,000であり、これは2009年1月に最初のブロックが生成されて以来278,000個のブロックが積み上げられたことを意味します。

ブロックハッシュと違って、ブロック高はユニークな識別子ではありません。1つのブロックは特定のブロック高が割り当てられていますが、逆は真ではありません。2つまたはもっと多くのブロックが同じブロック高を持っているかもしれないからです。これは、ブロックチェーン内の同じ場所をマイナーが競争して取得しようとしているためです。どのようにしてこれが起こるのかについては[\[forks\]](#)節で詳細に説明します。ブロック高はまたブロックのデータの一部でもありません。それぞれのノードは、Bitcoinネットワークからブロックを受け取ったときにこのブロックがブロックチェーン内のどこの位置(ブロック高)にあるのかをブロック高を使うことなくブロックハッシュから動的に特定します。ただこのブロック高は、ブロックチェーンからすば

やくブロック情報を取得する目的でデータベースにメタデータとして保存されている可能性があります。

**TIP** ブロックの  
1つのブロックを一意に指定します。ブロックはまた常に特定の  
は常に  
ブロック高  
を持っている。しかし、常に特定のブロック高は1つのブロックを指定できるというわけでは  
ないのです。これは、2つまたはもっと多くのブロックがブロックチェーン内の1つ位置の取り  
合いをしているかもしれないためです。

## genesisブロック

ブロックチェーンの一番最初のブロックはgenesisブロックと呼ばれており、これは2009年に作られたものです。これはブロックチェーンにある全てのブロックの祖先であり、どんなブロックからスタートしてブロックチェーンを過去にさかのぼっていても結局genesisブロックにぶつかります。

全てのノードはいつも1つのブロックのブロックチェーンから始まります。というのは、genesisブロックは変更できないようにBitcoinクライアントにハードコーディングされているためです。全てのノードはgenesisブロックのハッシュとデータ、作成された日時、1つのトランザクションが含まれていることを"知って"おり、これにより信用されたブロックチェーンを構築するときの安全な"根幹"を持つことができるのです。

Bitcoin Coreクライアントの内部にハードコーディングされたgenesisブロックを見るには、[chainparams.cpp](#) を参照してみてください。

以下のハッシュがgenesisブロックのハッシュです。

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

blockchain.infoのようなブロック探索サイトでブロックハッシュを検索することができ、以下のハッシュを含むURLを参照することでgenesisブロックの内容が書かれたページを見ることができます。

<https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

<https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

Bitcoin Coreリファレンスクライアントの以下のコマンドを実行することでもgenesisブロックの内容を確認することができます。

```
$ bitcoind getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```



```
{
  "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "confirmations" : 308321,
  "size" : 285,
  "height" : 0,
  "version" : 1,
  "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
  "tx" : [
    "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
  ],
  "time" : 1231006505,
  "nonce" : 2083236893,
  "bits" : "1d00ffff",
  "difficulty" : 1.00000000,
  "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

genesisブロックには隠されたメッセージが含まれています。coinbaseトランザクションインプットには"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks."(リーマンショックの影響を受けた銀行へのイギリス政府からの資金援助に関する記事)という文章が記載されています。このメッセージはイギリスの新聞 [タイムズ紙](#) のヘッドラインを参照することでgenesisブロックが2009年1月3日以前になかったことの証明になっています。これはまた、前例のない、世界規模の金融危機と同時期にビットコインが稼働を始めたという事実をもって、独立した金融システムの重要性を想起させる、皮肉なリマインダになっています。このメッセージはBitcoinの創造者であるSatoshi Nakamotoによって最初のブロックに埋め込まれたものです。

## ブロックチェーン内でのブロック連結

Bitcoinフルノードは、genesisブロックから始まるブロックチェーンのローカルコピーを保持しています。このブロックチェーンのローカルコピーは、新しいブロックが見つかりチェーンが拡張されるたびに定期的にアップデートされます。ノードがBitcoinネットワークからブロックを受け取ったとき、これらのブロックの検証を行いすでに保持しているブロックチェーンにこれらを連結します。連結するために、ノードは受け取ったブロックヘッダを調べ"previous block hash"を探します。

例として、あるノードが277,314個のブロックをブロックチェーンのローカルコピーに持っていると仮定してみましょう。ノードが知っている一番最後のブロックはブロック277,314で、ブロックヘッダのハッシュは000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249 です。

Bitcoinノードは新しいブロックをBitcoinネットワークから受け取りました。このブロックは以下のようなものです。

```

{
  "size" : 43560,
  "version" : 2,
  "previousblockhash" :
    "00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
  "merkleroot" :
    "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
  "time" : 1388185038,
  "difficulty" : 1180923195.25802612,
  "nonce" : 4215469401,
  "tx" : [
    "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

    #[...] 中略 ...]

    "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
  ]
}

```

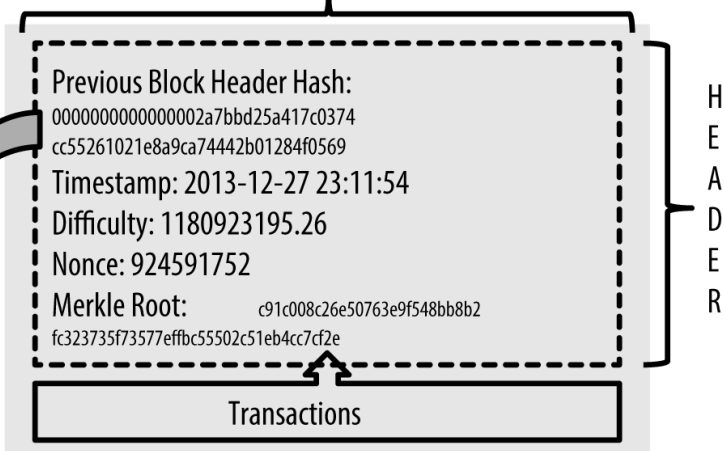
このノードがこの新しいブロックの `previousblockhash` フィールドを調べてみると、ここにある親ブロックのハッシュがノードが持っているブロックチェーンの一番最後のブロック高 `277,314` のハッシュであることがわかりました。このため、ノードはこの新しいブロックが一番最後のブロックの子ブロックであると分かり、このノードは新しいブロックをブロックチェーンの最後に追加することにしました。最終的に新しいブロック分だけ長いブロック高 `277,315` を持ったブロックチェーンができたこととなります。ブロックヘッダにある `previous block hash` を通してチェーン内で連結されているブロック。図は3つのブロックのチェーンを示していて、それぞれのブロックは `previousblockhash` フィールドを通して連結されています。

## Merkle Trees

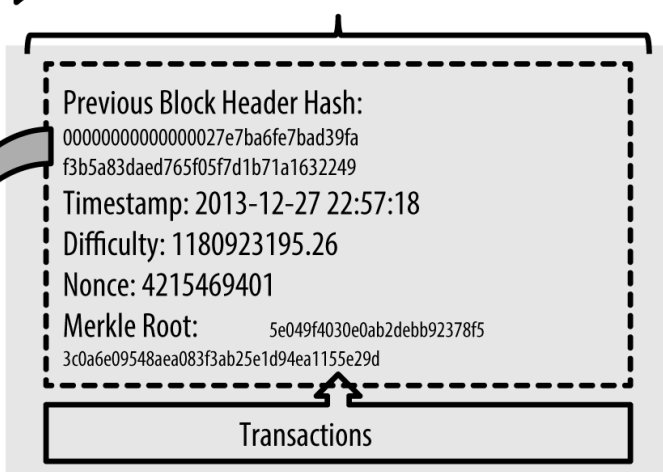
ブロックチェーンのそれぞれのブロックには `merkle tree` を使ったブロックに含まれる全てのトランザクションのサマリが含まれています。

`merkle tree` は ("binary hash tree" 二分ハッシュ木) と呼ばれるもので、効率的に大きなデータをまとめ、データ全体を検証できるようにしています。`merkle tree` は暗号学的なハッシュを含む二分木です。"tree"という言葉は、コンピュータサイエンスの分野で使われる枝葉を持つデータ構造を表す言葉として使われています。しかし、これらの木はあとで出てくる例で見るように通常上下が逆の状態を表され、上方向が"根っこ(root)"で下方向が"葉(leaves)"になっています。

Block Height 277316  
Header Hash:  
00000000000001b6b9a13b095e96db  
41c4a928b97ef2d944a9b31b2cc7bdc4



Block Height 277315  
Header Hash:  
000000000000002a7bbd25a417c0374  
cc55261021e8a9ca74442b01284f0569



Block Height 277314  
Header Hash:  
0000000000000027e7ba6fe7bad39fa  
f3b5a83daed765f05f7d1b71a1632249

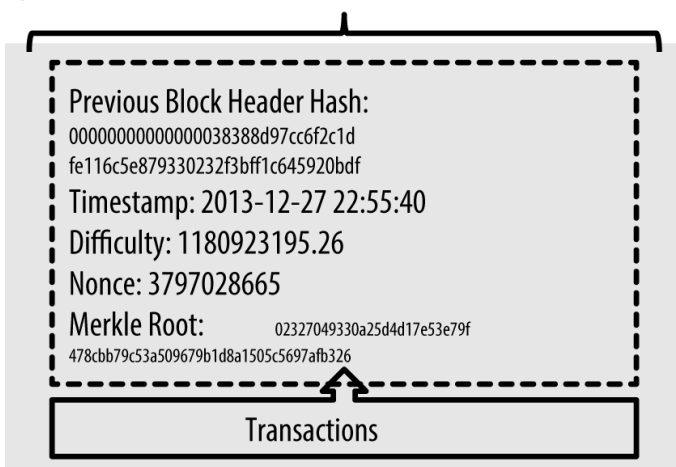


Figure 1. ブロックヘッダにある *previous block hash* を通してチェーン内で連結されているブロック。

merkle

treeはブロックに含まれている全てのトランザクションをまとめるために使われ、トランザクション全体のデジタルフィンガープリントを作成することであるトランザクションがブロックに含まれているかどうかをとっても効率的に確認する方法を提供します。merkle treeは再帰的に葉ノードのペアから1つのハッシュ値を計算し、ハッシュが1つだけ残るまで続けます。この残ったハッシュを root または merkle root と呼びます。Bitcoinのmerkle treeに使われている暗号的なハッシュアルゴリズムはSHA256を2回適用したもので、double-SHA256とも呼ばれています。

N個のデータ要素がハッシュ化されmerkle treeの中にまとめられているとき、多くても  $2 \cdot \log_2(N)$  回の計算をすることでデータ要素がmerkle treeに含まれているかどうかを知ることができます。

merkle treeは底部から作られていきます。次の例で見ると、A、B、C、Dの4つのトランザクションから始めてみましょう。これらはmerkle tree内の各ノードのハッシュ値計算図に示されている通りmerkle treeの葉を構成するものです。これらのトランザクションがmerkle treeに保存されている訳ではなく、トランザクションのデータをハッシュ化したものが葉ノードである  $H_A$ ,  $H_B$ ,  $H_C$ ,  $H_D$  に保存されます。

$$H_{A\sim} = \text{SHA256}(\text{SHA256}(\text{Transaction A}))$$

葉ノードの隣同士のペアは、ペアそれぞれのハッシュをくっつけたもののハッシュを取り親ノードにまとめられます。例えば、親ノード  $H_{AB}$  を作るためには2つの子ノードの32byteハッシュをくっつけて64byteの文字列を作ります。その後この文字列は2回ハッシュ化され親ノードのハッシュが作り出されます。

$$H_{AB\sim} = \text{SHA256}(\text{SHA256}(H_{A\sim} + H_{B\sim}))$$

このプロセスはノードが1つになるまで続けられ、この最後の1つのノードを merkle root と呼びます。32byteハッシュはブロックヘッダに保存され、4つの全トランザクションのデータがmerkle rootのハッシュにまとめられます。

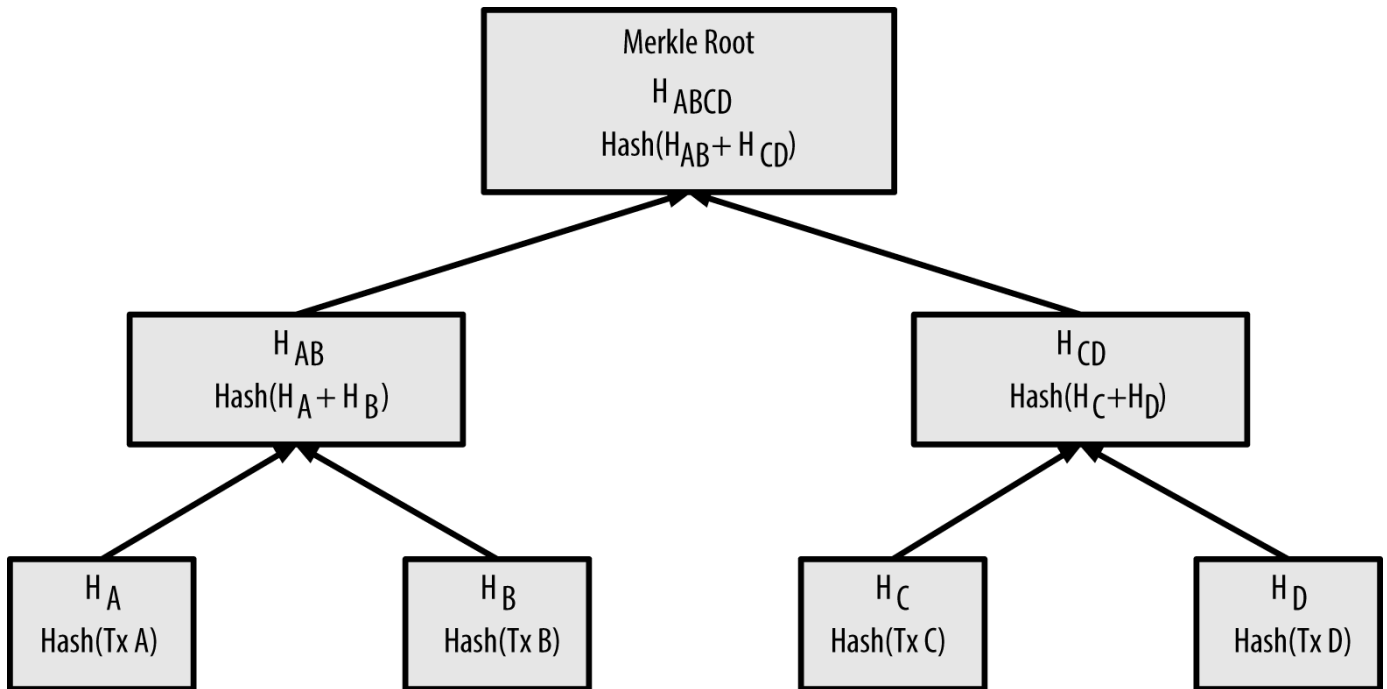


Figure 2. merkle tree内での各ノードのハッシュ値計算

merkle

treeは二分木であるため、葉ノードが偶数個になる必要があります。もしトランザクションの数が奇数個である場合は、最後のトランザクションハッシュは自分自身とくっつけてハッシュを作りバランス木として知られている偶数個の葉ノードができるようにします。これは1つのデータ要素を二重で使うことで偶数個のデータ要素を持ったmerkle treeを構成図に示されており、トランザクションCが二重になっています。

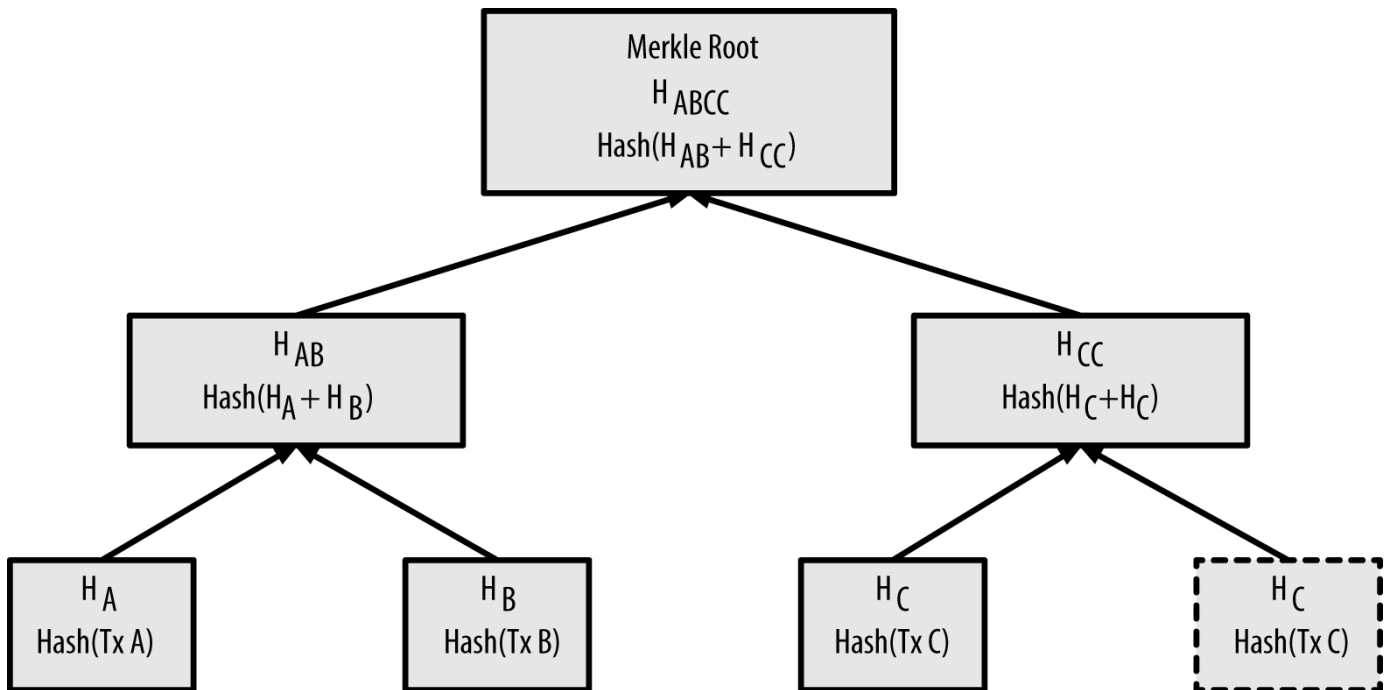


Figure 3. 1つのデータ要素を二重で使うことで偶数個のデータ要素を持ったmerkle treeを構成

4つのトランザクションから木を作る方法は、どんなサイズの木にも一般化できます。Bitcoinでは、1つのブロックに数百から千個以上のトランザクションを持つことはよくあり、さきほどの方法と全く同じ方法でmer

kle rootの32byteハッシュとしてまとめられます。多くのデータ要素をまとめているmerkle tree図には、16個のトランザクションからなるmerkle treeが書かれています。図の中でmerkle rootは葉ノードよりも大きく見えますが、厳密に同じサイズで32byteです。ブロックの中に1つのトランザクションしかないのか、10万個のトランザクションがあるのかに関わらず、merkle rootは常に32byteのハッシュにまとめられます。

特定のトランザクションがブロックに含まれていることを証明するために、Bitcoinノードはたった  $\log_2(N)$  個の32byteハッシュを作り出すだけでよく、これにより特定のトランザクションをmerkle tree rootに繋ぐ authentication path または merkle path を構成します。これはブロックに含まれるトランザクションの数が多くなるにつれて特に重要になっていきます。というのは、トランザクション数に対して2を底とする対数を計算すると、トランザクション数が増えてもほとんど大きくなりませんからです。このことで、データサイズが数MBにもなるブロックに含まれる千個以上のトランザクションから1個のトランザクションを特定するためのmerkle pathを、たった10個から12個のハッシュ(320-384バイト)で効率的に作り出すことができます。

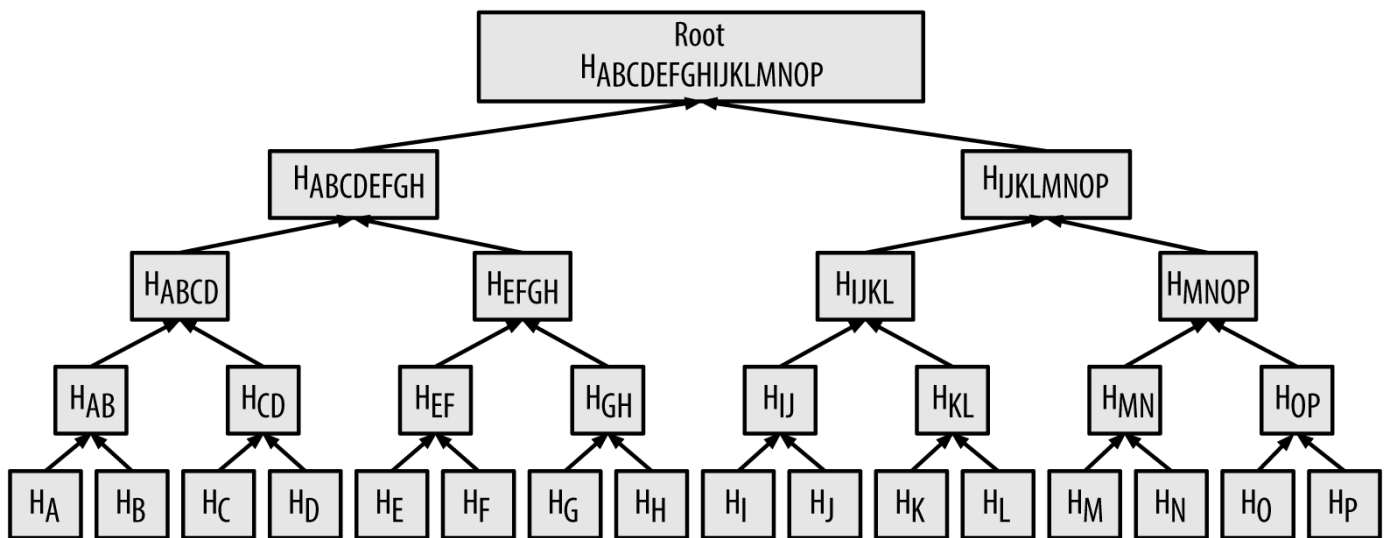


Figure 4. 多くのデータ要素をまとめているmerkle tree

データ要素が含まれていることの証明に使われるmerkle path図にある通り、Bitcoinノードはたった4つの32byteハッシュ(全部で128バイト)を使ったmerkle pathを作り出すことで、あるトランザクションKがブロックに含まれていることを証明できるのです。このmerkle pathは4つのハッシュ  $H_L$ 、 $H_I$ 、 $H_{MNOP}$ 、 $H_{ABCDEFGH}$  から構成されます(これらハッシュはデータ要素が含まれていることの証明に使われるmerkle path図に青い四角で記されています)。これらの4つのハッシュがauthentication pathとして提示されると、あらゆるBitcoinノードは  $H_k$  (データ要素が含まれていることの証明に使われるmerkle path図に緑の四角で表示)がmerkle rootに含まれているということを対となる追加の4つのハッシュ、 $H_{KL}$ 、 $H_{IJKL}$ 、 $H_{IJKLMNOP}$ 、merkle tree root を計算することで示すことができます(データ要素が含まれていることの証明に使われるmerkle path図に点線で縁取られた四角で表示)。

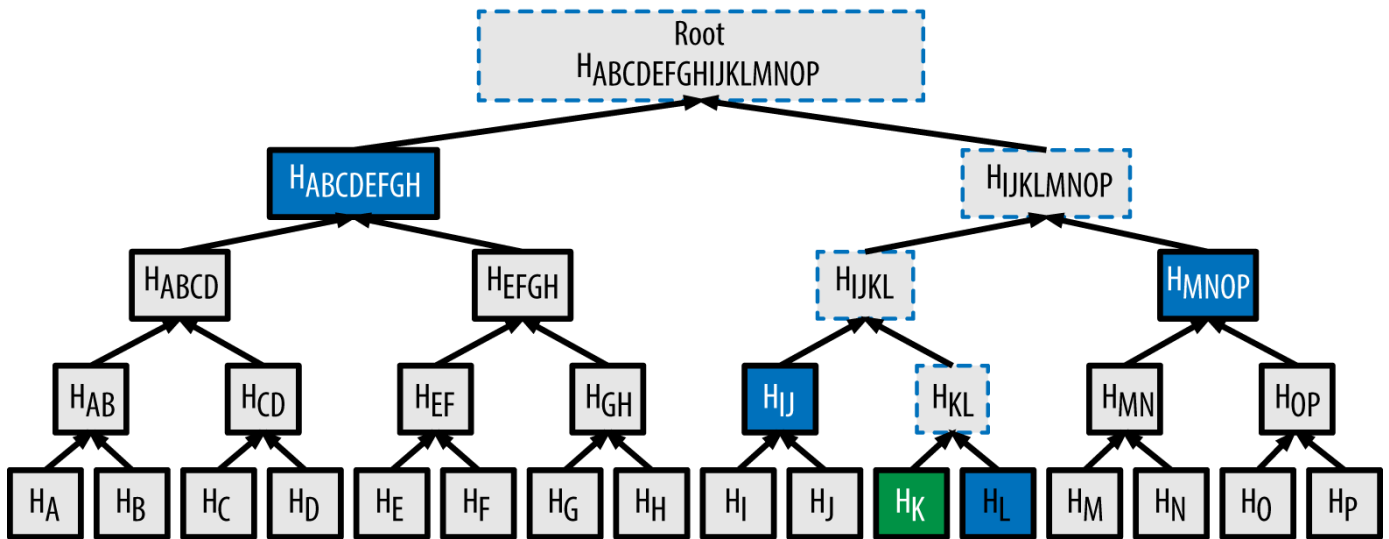


Figure 5. データ要素が含まれていることの証明に使われるmerkle path

merkle treeの構築にあるコードは、葉ノードからmerkle rootまでのmerkle treeを作り出すプロセスをデモンストレーションであり、いくつかの補助関数でlibbitcoinライブラリを使っています。

### Example 1. merkle treeの構築

```

#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
        return bc::null_hash;
    else if (merkle.size() == 1)
        return merkle[0];

    // While there is more than 1 hash in the list, keep looping...
    while (merkle.size() > 1)
    {
        // If number of hashes is odd, duplicate last hash in the list.
        if (merkle.size() % 2 != 0)
            merkle.push_back(merkle.back());
        // List size is now even.
        assert(merkle.size() % 2 == 0);

        // New hash list.
        bc::hash_list new_merkle;
        // Loop through hashes 2 at a time.
        for (auto it = merkle.begin(); it != merkle.end(); it += 2)
        {
            // Join both current hashes together (concatenate).

```

```

        bc::data_chunk concat_data(bc::hash_size * 2);
        auto concat = bc::make_serializer(concat_data.begin());
        concat.write_hash(*it);
        concat.write_hash(*(it + 1));
        assert(concat.iterator() == concat_data.end());
        // Hash both of the hashes.
        bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
        // Add this to the new list.
        new_merkle.push_back(new_root);
    }
    // This is the new list.
    merkle = new_merkle;

    // DEBUG output -----
    std::cout << "Current merkle hash list:" << std::endl;
    for (const auto& hash: merkle)
        std::cout << " " << bc::encode_hex(hash) << std::endl;
    std::cout << std::endl;
    // -----
}
// Finally we end up with a single item.
return merkle[0];
}

int main()
{
    // Replace these hashes with ones from a block to reproduce the same merkle root.
    bc::hash_list tx_hashes{
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000000"),
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000011"),
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000022"),
    };
    const bc::hash_digest merkle_root = create_merkle(tx_hashes);
    std::cout << "Result: " << bc::encode_hex(merkle_root) << std::endl;
    return 0;
}

```

[merkle tree構築例コードのコンパイルと実行](#)にさきほどのコードをコンパイルし実行した結果を示します。



## Example 2. merkle tree構築例コードのコンパイルと実行

```
$ # Compile the merkle.cpp code
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the merkle executable
$ ./merkle
Current merkle hash list:
 32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
 30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Current merkle hash list:
 d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3
```

merkle treeの効率性はブロックのデータサイズスケールが大きくなるほど顕著になります。merkle treeの効率性は、ブロックのデータサイズ、ブロック内トランザクション数に応じて必要なmerkle pathの大きさを示しています。

Table 3. merkle treeの効率性

| トランザクション数       | ブロック平均データサイズ | パスサイズ (ハッシュ数) | パスサイズ (byte) |
|-----------------|--------------|---------------|--------------|
| 16 トランザクション     | 4 KB         | 4 ハッシュ        | 128 byte     |
| 512 トランザクション    | 128 KB       | 9 ハッシュ        | 288 byte     |
| 2048 トランザクション   | 512 KB       | 11 ハッシュ       | 352 byte     |
| 65,535 トランザクション | 16 MB        | 16 ハッシュ       | 512 byte     |

表から分かるように、ブロックのデータサイズが急速に大きく(16個のトランザクションの時は4KBですが、65,535個のトランザクションだと16MB)なっていても、merkle pathのデータサイズはトランザクション数が大きくなるよりもゆっくり大きくなります(128バイトから512バイトにしか増えない)。Bitcoinノードはmerkle treeとともにブロックヘッダ(1ブロックあたり80byte)を取得しフルノードから小さなmerkle pathを取得することで、ブロックの中にあるトランザクションが含まれているかどうかを知ることができます。これには、数GBもあるブロックチェーンの大半を保存したり、また受け渡してもらう必要もありません。simplified payment verification(SPVノード)と呼ばれるフルブロックチェーンを持っていないBitcoinノードは、merkle pathを使うことで全てのブロックをダウンロードすることなくトランザクションを検証しているのです。

## merkle treeとSimplified Payment Verification (SPV)

merkle

treeはSPVノードによってよく利用されます。

SPVノードは全てのトランザクションを持たず完全なブロックチェーンをダウンロードすることはありません。SPVノードはauthentication pathまたはmerkle pathを使って、あるトランザクションがブロックに含まれているかどうか確認します。

例として、ウォレット内にあるBitcoinアドレスへの支払いにだけ関心のあるSPVノードを考えてみましょう。SPVノードは、ウォレット内のBitcoinアドレスを含むトランザクションだけを取得するためピアにbloom filterを送ります。ピアはbloom filterに合致するトランザクションを確認し、merkleblock messageを使ってブロック情報を送り返します。この merkleblock messageにはブロックヘッダとmerkle pathが含まれており、このmerkle pathはSPVノードにとって関心のあるトランザクションからmerkle rootへの経路です。SPVノードはこのmerkle pathを使って関心のあるトランザクションがブロック内に含まれていることを確認します。またこのブロックヘッダを使いこのブロックをすでに保持しているブロックチェーン情報と結びつけます。2つの連結の組み合わせ、トランザクションとブロック、ブロックとブロックチェーン、を使うことでこのトランザクションがブロックチェーンに記録されているということを確認しています。SPVノードはトランザクションの確認にブロックヘッダに対するデータとmerkle pathに関するデータという1KB以下のデータを受け取ることで済ますことができ、フルノードと比べて1000分の1以下のデータを保持するだけでこれが可能になっています(現在だと1MB程度)。

# マイニングとコンセンサス(合意形成)

## イントロダクション

マイニングとは貨幣供給において新しいbitcoinが追加される処理のことです。マイニングはまた、不正なトランザクションまたはダブルスPEND(二重使用)と呼ばれる同じ量のbitcoinが二度使用されるような不正からBitcoinのシステムを保護します。マイナーはbitcoinを報酬として受け取れる可能性と引き換えに、Bitcoinネットワークに処理能力を提供します。

マイナーは新しいトランザクションを検証し、これらをグローバルな元帳に記録します。一番最後のブロック以降に生じたトランザクションが含まれる新しいブロックは10分ごとに"マイニング"され、ブロックチェーンにこれらのトランザクションが追加されます。ブロックチェーンに追加された、またはブロックの一部になったトランザクションは"承認済(confirmed)"となり、これらのトランザクションで送付されたbitcoinを新しい所有者が使用することができるようになります。

マイナーはマイニングに対する報酬を二種類受け取ります。1つは新しいブロックを作った際に新しく発行されたbitcoin、もう1つはブロックに含まれている全トランザクションから得られるトランザクション手数料です。これらの報酬を得るために、マイナーたちは暗号学的ハッシュアルゴリズムに基づいた難解な数学的な問題を競って解決しなければいけません。proof of workと呼ばれるこの問題への解法は新しいブロックに含められ、マイナーが十分なコンピュータリソースを注ぎ込んだことの証明として機能することになります。報酬を稼ぐためのproof-of-workアルゴリズムの解法とトランザクションをブロックチェーンに記録する権利をマイナーたちが競争するという仕組みは、Bitcoinのセキュリティモデルの基礎をなしています。

新しいbitcoinの生成プロセスはマイニングと呼ばれています。なぜなら、貴金属を採掘するように報酬が減っていくように設計されているからです。bitcoinの供給はマイニングを通して行われ、これはあたかも紙幣を刷ることで新しいお金を発行している中央銀行のようなものです。マイニングを通して新たに作られるbitcoinの量はおよそ4年ごと(正確には210,000ブロックごとに)減っていきます。2009年1月の時点で1ブロックあたり50bitcoinから始まり、2012年11月には半分の1ブロックあたり25bitcoinになりました。次は2016年のどこかで1ブロックあたり12.5bitcoinとさらに半分になります。この公式に当てはめると、Bitcoinマイニングの報酬は全てのbitcoin(20,999,999.98 bitcoin)が発行され終わるおよそ2140年まで指数関数的に減少していきます。2140年以降はもう新しいbitcoinが発行されることはありません。

Bitcoinマイナーはまたトランザクションから手数料を得ます。全てのトランザクションにはトランザクション手数料が含まれている可能性があり、トランザクションインプットとトランザクションアウトプットの差として与えられます。マイニング競争に勝ったマイナーは、生成したブロックに含まれるトランザクションにある"置いていかれたおつり"を報酬として得ることになります。今日、この手数料はBitcoinマイナーの収入の0.5%以下であり、主な収入は新しくマイニングされたbitcoinになっています。しかし、時間が経つごとにこの報酬は減っていき1ブロックに含まれるトランザクション数が増えていくと、Bitcoinマイニング収入の多くの部分はトランザクション手数料からになるでしょう。2140年以降になると、全てのBitcoinマイナーの収入はトランザクション手数料の形で得られるようになります。

"マイニング"という言葉は、ちょっと紛らわしい言葉です。貴金属を採掘するという意味で、マイニングで得られる報酬に注意が向いてしまうためです。確かにマイニングはこの報酬がインセンティブとなりますが、元々のマイニングの目的は新しいbitcoinを生成することによって与えられる報酬ではないのです。もしマイニングをbitcoinを作り出すプロセスだと思っているのであれば、このプロセスの意味を取り違えています。マ

マイニングは分散的な手形交換所であり、マイニングによってトランザクションが検証されます。マイニングはBitcoinシステムを安全なものにし、またマイニングがあることで中央当局なしにネットワーク上での合意形成が可能になったのです。

マイニングはBitcoinを特別なものに行っている発明であり、peer-to-peerのデジタルキャッシュを基礎とした分散的セキュリティ機構です。新たなbitcoinやトランザクション手数料という報酬は、マイナーたちにBitcoinネットワークのセキュリティを守らせるインセンティブの枠組みであり、また一方では同時に通貨供給の目的も果たしているのです。

この章では、最初に通貨供給メカニズムとしてのマイニングを説明し、その後マイニングの最も重要な性質でありBitcoinセキュリティの土台であるdecentralized emergent consensus(分散型創発コンセンサス。ノードは個々で独立にトランザクションなどの検証をし、隣接ノードと相互作用をしているだけであり直接的にビットコインネットワーク全体の合意形成を行っているわけではありません。しかし、結果としてビットコインネットワーク全体でのコンセンサスが取られることとなります。)について説明します。

## Bitcoin経済と通貨の発行

bitcoinは、1ブロックごとのbitcoin発行量公式に基づきブロックごとに"鑄造"されます。この新たなbitcoinはおおよそ10分ごとに生成されるブロックに含まれ、何も無いところから生成されます。210,000ブロックごとに、または約4年ごとに、通貨発行量は半分に減ります。Bitcoinネットワークが稼働を始めてからの最初の4年間はそれぞれのブロックが50bitcoinを発行していました。

新しいbitcoin発行量は2012年11月に1ブロックあたり25bitcoinに減り、420,000ブロックがマイニングされる2016年のどこかで更に12.5bitcoinに減ります。この1ブロックあたりの発行量は指数関数的に減少し、64回の半減を繰り返して13,230,000ブロック(おおよそ2137年にマイニングされるはずのブロック)まで減少していきます。これ以降はbitcoinの最小通貨単位である1satoshiを下回ってしまうため新しくbitcoinを発行できなくなり、最終的には2140年あたりに1344万ブロック(13.44Mブロック)でほぼ2100万bitcoin(2,099,999,997,690,000satoshi)が発行されることとなります。この後、ブロックには新しいbitcoinが含まれなくなり、マイナーは単にトランザクション手数料を通して報酬を得るようになります。[幾何級数的に減少する発行率に基づく、時間経過ごとのbitcoin通貨発行量](#)図は通貨発行量が減るにつれて時間とともにbitcoin総量がどのように増えていくかを示しています。

## Bitcoin Money Supply

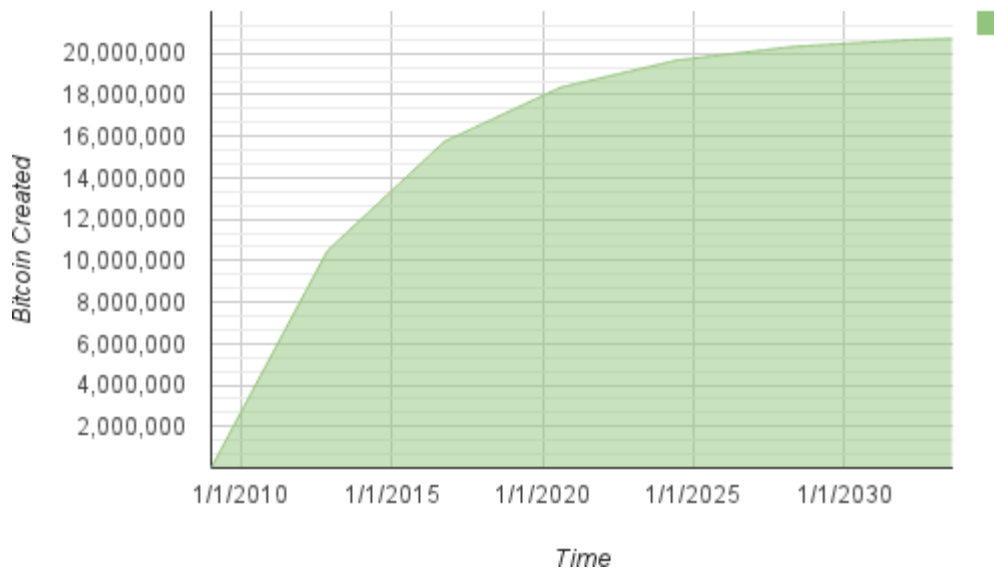


Figure 1. 幾何級数的に減少する発行率に基づく、時間経過ごとのbitcoin通貨発行量

### NOTE

採掘されるbitcoinの最大値は、bitcoinに対する可能なマイニング報酬の上限です。実際マイナーはブロックを採掘しても、得る報酬を故意に完全な報酬より少なくするかもしれません。そのようなブロックの可能性が既存または将来にわたってあるため、結果的な通貨総発行総量はより少なくなります。

支払いに総額いくら bitcoinが必要となるかを計算するためのスクリプト図にある例コードで将来発行されるbitcoin総量を計算しています。

Example 1. 支払いに総額いくら bitcoinが必要となるかを計算するためのスクリプト

```
# Original block reward for miners was 50 BTC
start_block_reward = 50
# 210000 is around every 4 years with a 10 minute block interval
reward_interval = 210000

def max_money():
    # 50 BTC = 50 0000 0000 Satoshis
    current_reward = 50 * 10**8
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total

print "Total BTC to ever be created:", max_money(), "Satoshis"
```

max\_money.pyスクリプトの実行図にはこのスクリプトによって出力される結果を示しています。

## Example 2. max\_money.pyスクリプトの実行

```
$ python max_money.py
Total BTC to ever be created: 2099999997690000 Satoshis
```

1ブロックあたりの通貨発行量を減少させbitcoinの総量を有限にすることでインフレを防ぐことができます。中央銀行によって無限に発行可能な法定通貨と違い、bitcoinは通貨発行によってインフレが起きることは決してありません。

### デフレ通貨

とても重要なこととして、bitcoin総量が有限であるためこの通貨は本質的に デフレ傾向が生じることとなります。デフレは通貨価格(または通貨交換レート)をつり上げようとする需要側と、供給側のミスマッチによって生じる適正価格の調整現象です。インフレと逆に、通貨価格のデフレはお金の購買力が時間とともに上がるということを意味します。

多くのエコノミストたちはデフレ経済はどんなに費用をかけてでも回避すべき災害なのかを議論しています。というのは、急激なデフレが生じると人々はお金を使わず蓄積しようとし、より物価が下落することを望むようになってしまうためです。このような需要がなくなってしまうデフレスパイラルに突入すると、日本の"失われた10年"のような現象が生じるようになります。

Bitcoinの専門家たちはデフレそれ自体は悪いものではないと話しています。むしろ、需要がなくなることに伴いデフレが生じるということは、デフレを学習する上での単なる例でしかないと考えています。無限に発行可能な法定通貨の場合、完全に需要がなくなり、かつ追加の通貨発行をしないようにしなければデフレスパイラルは簡単には生じません。Bitcoinでのデフレは需要がなくなることによって起こるものではなく、あらかじめ決めておくことができる制限された供給側によって引き起こされるものです。

現実には、デフレの場合ベンダーが価格を下げてでも人々の貯蓄が優先されます。そして、買う側が貯蓄をやめるくらいに価格が下がるまで確実にデフレが続くこととなります。売り側も貯蓄をしたいと考えるため、価格の下落は買う側と売り側の双方の思惑が均衡した価格になるまで続きます。bitcoinの場合価格が30%下がっても、多くのbitcoin販売者は貯蓄よりも利益を得ることを考えます。素早い経済的後退が起きなければ、この通貨のデフレ的側面が本当に問題であるかどうか分からないのです。

## 分散化されたコンセンサス(合意形成)

前の章でブロックチェーンというグローバルな公的な元帳を説明しました。これは、Bitcoinネットワークに参加している人全員が所有権の権威レコードとして認めているものです。

しかし、Bitcoinネットワークに参加している皆さんは、共通認識である「何を誰が所有しているのか」ということの"真偽"を、誰も信用することなくどのようにして認めているのでしょうか。今までの全ての支払いシステムは全ての取引を検証しクリアリングする手形取引サービスを提供する中央集権的信用モデルに依存しています。Bitcoinには中央集権的な仕組みはありません。しかし、ほぼ全てのフルノードが権威レコードとして信用できる公的な元帳の完全なコピーを持っています。このブロックチェーンは中央権力によって作られる訳ではありませんが、Bitcoinネットワークに属している全てのノードによって独立に組み立てられます。そして、どういうわけかBitcoinネットワークに属する全てのノードは結果的に他の全てのノードと同じ公的な

元帳のコピーを組み立てることになります。信用できないネットワークから送られた情報に基づき動作するにも関わらず。この章では中央権力なくBitcoinネットワークがグローバルなコンセンサスに到達するプロセスを説明します。

Satoshi Nakamotoの主要な発明は*emergent consensus*(創発的コンセンサス)に対する分散メカニズムです。emergentというのは、決められた行動による明示的な同意ではないという意味です。ここでの同意は数千の独立したノードの非同期的相互作用の結果として生まれた人工物で、全ノードは次のシンプルなルールに従っています。通貨、トランザクション、支払い、および中央権力や信用に依存しないセキュリティモデルといったBitcoinの全ての特徴はこの発明から導かれます。

Bitcoinの分散化されたコンセンサスは、Bitcoinネットワークを通して独立的に各ノードで起こる以下4つのプロセスの相互作用から生じてきます。

- 独立したトランザクション検証(全てのフルノードによる判断条件の包括的なリストに基づく検証)
- 独立したトランザクション集積(proof-of-workアルゴリズムによる計算と結びついた、マイニングノードによるブロックへのトランザクションの集積)
- 独立した新規ブロック検証とブロックチェーンへの埋め込み(全てのノードによって新しいブロックが検証され、このブロックがブロックチェーンに取り込まれる)
- 独立したブロックチェーン選択(個々のノードで、proof of workを通して証明された最も多くの累積計算量を持っているブロックチェーンが選ばれる)

次のいくつかの節で、これらのプロセスについて説明し、どのようにしてノードが相互作用をしてBitcoinネットワーク全体のemergent consensusをしているのかを説明します。

## 独立したトランザクション検証

[[transactions](#)]では、どのようにしてウォレットがUTXOを集めてトランザクションを作り、適切なunlocking scriptを付与して新しい所有者に割り当てられた新しいアウトプットを作るかを説明しました。結果的に作られたトランザクションはBitcoinネットワーク全体に伝搬できるようにBitcoinネットワーク内の隣接ノードに送られることとなります。

しかし、隣接ノードにトランザクションが転送される前に、トランザクションを受け取った全てのBitcoinノードは最初にトランザクションを検証します。これによって有効なトランザクションだけがBitcoinネットワーク内を伝搬することを保証しており、無効なトランザクションは最初にこのトランザクションに会ったノードによって破棄されます。

それぞれのノードは全てのトランザクションを以下の長いチェックリストを判断基準として検証します。

- トランザクションの文法またはデータ構造は正しいか
- インプットとアウトプットのいずれも空でないか
- byte単位のトランザクションデータサイズが MAX\_BLOCK\_SIZE よりも小さいか
- それぞれのアウトプットvalueおよびtotal valueは許されている値の範囲内(0より大きく、2100万bitcoinよりも小さい)にあるか
- インプットのいずれもhash=0, N=-1でないか(coinbaseトランザクションはリレーされていくべきでない)
- nLockTime は INT\_MAX より小さいかまたは等しいか

- byte単位でのトランザクションデータサイズは100より大きいかまたは等しいか
- トランザクションに含まれている署名オペレーション数は、署名オペレーション回数上限よりも小さいか
- unlocking script( scriptSig )はスタックに数字をpushすることだけででき、locking script( scriptPubkey )は isStandard 形式に合っているか(これにより"非標準"トランザクションは拒否されます)
- トランザクションプールか、またはメインブランチブロックチェーンのブロックに、同じトランザクションがあるか
- それぞれのインプットに対して、もしこのインプットが参照しているアウトプットをトランザクションプールの他のトランザクションも参照していた場合、このトランザクションを拒否する
- それぞれのインプットに対して、メインブランチブロックチェーンかトランザクションプールにインプットが参照しているトランザクションアウトプットが見つかるかを確認する。もし参照しているアウトプットが見つからなければ、これはorphan(孤児)トランザクションです。orphanトランザクションプールにまだこのトランザクションがなければ、orphanトランザクションプールにこのトランザクションを追加する
- それぞれのインプットに対して、もしインプットが参照しているアウトプットがcoinbaseアウトプットだった場合、このアウトプットは少なくとも COINBASE\_MATURITY (100) の承認数を持っているか
- それぞれのインプットに対して、参照しているアウトプットがすでに使用されて使用不可になっていないか
- 参照しているアウトプットを使って、それぞれのインプットvalueとその総和が許されている値の範囲内(0より大きく、2100万bitcoinよりも小さい)にあるか
- もしインプットvalueの総和がアウトプットvalueの総和よりも小さければ拒否する
- もしトランザクション手数料が少なすぎて空ブロックに入れることができない場合は拒否する
- それぞれのインプットにあるunlocking scriptは、対応したアウトプットのlocking scriptを解除できるか

これらの条件はBitcoin リファレンスクライアントにある `AcceptToMemoryPool`、`CheckTransaction`、`CheckInputs` 関数を見ることで詳細を確認できます。この条件は、新しい種類のDOS攻撃に対応したり、またさらにトランザクションの種類を増やすためにときどき緩めたりと、時間とともに変わっていくことに注意してください。

トランザクションを受け取ったときや他のノードに伝搬させたりする前に独立にそれぞれのトランザクションを検証することによって、全てのノードが トランザクションプール、メモリプール または *mempool* と呼ばれるプールを構築します。

## マイニングノード

いくつかのBitcoinノードは **マイナー** と呼ばれる特別なBitcoinノードです。[\[ch01\\_intro\\_what\\_is\\_bitcoin\]](#)で、上海にいるコンピュータエンジニアの学生でBitcoinマイナーであるJingを紹介しました。Jingは、bitcoinをマイニングするために作られた特別なコンピュータ "マイニング専用マシン" を走らせてbitcoinを稼いでいます。Jingの特別なマイニングハードウェアはフルBitcoinノードが走っているサーバに接続されています。Jingと違い、マイナーの中には**マイニングプール**で見えるようにフルノードを使うことなくマイニングをしているマイナーもいます。他のフルノードと同様、Jingのノードは承認されていないBitcoinネットワーク上のトランザクションを受け取って伝搬しています。しかし、Jingのノードはそれだけでなくいくつかのトランザクションを新しいブロックに集積することもしているのです。

JingのBitcoinノードは、全てのBitcoinノードがするようにBitcoinネットワーク上を伝搬している新しいプロ



ックを待っています。しかし、新しいブロックが来ることはマイニングノードにとって特別な意味を持ちます。新しいブロックが伝搬してくるということは事実上マイナー同士の競争が終わったということです。この伝搬は競争の勝者を伝えることになるからです。マイナーに新しいブロックが届くということは他の誰かが競争に勝ちそれ以外の人は負けたということを意味します。しかし、このラウンドの終わりは次のラウンドの始まりです。新しいブロックはレースの終わりを示す単なるチェッカーフラッグではなく、次のブロックに対するスターティングピストルでもあります。

## ブロックへのトランザクション集積

トランザクションを検証した後、Bitcoinノードはメモリプールまたはトランザクションプールにそれらのトランザクションを追加します。このプールにあるトランザクションはブロックに含められる(マイニングされる)までこのプールで待機しています。JingのBitcoinノードは他のBitcoinノードと同じようにトランザクションを集め検証し新しいトランザクションをリレーします。しかし、他のBitcoinノードと違うのは、JingのBitcoinノードはこれらのトランザクションを候補ブロック(candidate block)に集めておくということです。

AliceがBobのコーヒーショップでコーヒー代を払ったときに作られたブロックを追ってみましょう([cup\_of\_coffee]参照)。Aliceのトランザクションはブロック277,316に含まれていました。この章でやりたいことを説明しやすくするために、このブロックがJingのマイニングシステムによって採掘され、Aliceのトランザクションがこの新しいブロックの一部になっているとしましょう。

JingのマイニングBitcoinノードはブロックチェーンのローカルコピーを保持していて、このブロックチェーンには2009年にBitcoinシステムが稼働を始めてから作られた全てのブロックが含まれています。Aliceがコーヒー代を支払うまでにJingのBitcoinノードはブロック277,314までブロックチェーンを組み立てました。JingのBitcoinノードはトランザクションを待っていたり、新しいブロックをマイニングしたり、または他のBitcoinノードが発見したブロックを待っていたりしています。JingのBitcoinノードがマイニングしているときにBitcoinネットワークからブロック277,315を受け取りました。このブロックが到着したということは、ブロック277,315の競争が終わり、ブロック277,316を作る競争が始まったということを示します。

ブロック277,315が到着する前の10分間、JingのBitcoinノードはブロック277,315に対する解を探しながら、同時に次のブロックの準備のためトランザクションを集めていました。現在まで数百個のトランザクションをメモリプールに集めました。ブロック277,315を受け取り検証するとすぐに、JingのBitcoinノードはメモリプールにある全てのトランザクションをチェックしブロック277,315に含まれていたトランザクションをメモリプールから削除していきます。

JingのBitcoinノードはすぐにブロック277,316の候補となる新しい空ブロックの構築を始めました。このブロックは候補ブロックと呼ばれており、有効なproof of workが含まれていないブロックです。このブロックは、マイナーがproof-of-workアルゴリズムへの解を見つけたときにのみ有効になるのです。

## トランザクション年齢、トランザクション手数料、トランザクション優先度

候補ブロックを構築するために、JingのBitcoinノードはメモリプールからトランザクションを選びました。この選び方はそれぞれのトランザクションごとに優先度を計算し、最も優先度が高いものから先に選びます。トランザクション優先度は未使用トランザクションであるUTXOの"年齢"に基づき計算され、新しく、より小さいvalueインプットを持つトランザクションよりも古く大きなvalueインプットを持つトランザクションが優先されます。優先トランザクションで、しかもこのブロックに十分なスペースがあれば、トランザクション手数料がないトランザクションも送られます。

トランザクションの優先度は、インプットのvalueと年齢(Input Age)の積の総和をトランザクションの総データサイズ(Transaction Size)で割ったもので計算しています。

$$\text{Priority} = \text{Sum} (\text{Value of input} * \text{Input Age}) / \text{Transaction Size}$$

この方程式にあるインプットのvalueはsatoshi単位(bitcoinの1億分の1)で計られます。UTXOの年齢はUTXOがブロックチェーンに記録されてから積み重ねられたブロック数で、このUTXOが含まれているブロックがブロックチェーンのトップから何ブロック"深い"か"で計ります。トランザクションのデータサイズはbyte単位で計られます。

トランザクションが"優先度が高い(High Priority)"と判断されるようになるためには、優先度が57,600,000よりも大きくなければいけません。これは、インプットのvalueが1bitcoin(1億satoshi)、年齢が1日(144ブロック)、トランザクションのデータサイズが250byteに相当します。

$$\text{High Priority} > 100,000,000 \text{ satoshis} * 144 \text{ blocks} / 250 \text{ bytes} = 57,600,000$$

ブロック内のトランザクションスペースの最初の50KBは、優先度が高いトランザクションのために取ってあります。このため、JingのBitcoinノードは、トランザクション手数料に関わらず、最初の50KBを最も優先度が高いトランザクションで埋めます。優先度が高いトランザクションはトランザクション手数料がゼロであっても処理されます。

その後、JingのマイニングBitcoinノードはブロックの残りをブロックサイズの最大値(コード内のMAX\_BLOCK\_SIZE

)までトランザクションで埋めます。ここで埋められるトランザクションは最低トランザクション手数料以上を持つものであり、埋められる優先度はトランザクション手数料をトランザクションのデータサイズ(KB単位)で割った値の高い順で決められます。

もしブロックにまだスペースがあれば、JingのマイニングBitcoinノードはトランザクション手数料がないトランザクションを残りブロックスペースに埋めることを選択するかもしれません。マイナーの中には最善努力としてトランザクション手数料を持たないトランザクションをマイニングすることを選ぶマイナーもいますが、他のマイナーはトランザクション手数料がないトランザクションは無視するかもしれません。

このブロックが全て埋められた後、メモリプールに残されたトランザクションは次のブロックに含めるためにメモリプールに残されます。トランザクションがメモリプールに残る時間が長くなるにつれて、このトランザクションインプットの年齢はどんどん上がっていきます。これは、新しいブロックがブロックチェーンの上になんども追加されるためです。トランザクションの優先度はこのトランザクションのインプットの年齢に依るので、メモリプールに残ったままになっているトランザクションは古くなり優先度が上がっていきます。結局トランザクション手数料を持たないトランザクションは十分に高い優先度になり無料でブロックに取り込まれます。

Bitcoinトランザクションに期限はありません。現在有効なトランザクションは永遠に有効です。しかし、もしトランザクションがBitcoinネットワーク内を1ノード分しか伝搬されないとすると、このトランザクションはマイニングBitcoinノードのメモリプールに保持されている間だけしか存在することはできません。マイニングBitcoinノードが再起動されたとき、そのメモリプールは初期化されデータが削除されます。というのは、メモリプールが一時的なストレージであるためです。有効なトランザクションはBitcoinネットワークを通じて伝搬されるかもしれませんが、結局もし伝搬されなければ長期間メモリプールに居続けることはできないこととなります。ウォレットには、そのようなトランザクションを再送信したり、また適度な時間以内でうまく処理されないようであればより高いトランザクション手数料を設定してトランザクションを再構築するよう



## Generationトランザクション

ブロックに最初の追加されたトランザクションは特別なトランザクションで、 *generation*トランザクション または *coinbase*トランザクション と呼ばれています。このトランザクションはJingのBitcoinノードによって構築され、マイニング努力に対する彼への報酬になります。JingのBitcoinノードは彼自身のウォレットへの支払いとしてgenerationトランザクションを作ります。具体的には "JingのBitcoinアドレスに25.09094928bitcoinを支払う" というようなものです。結局、Jingがブロックをマイニングしたことに対する報酬総額は、coinbase報酬(新規発行分の25bitcoin)と、ブロックに含まれている全てのトランザクションから得られたトランザクション手数料(0.09094928bitcoin)の和になります。これは、 [Generationトランザクション](#) に示されています。

```
$ bitcoin-cli getrawtransaction  
d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1
```



## coinbase報酬と手数料

generationトランザクションを構築するためにJingのBitcoinノードは最初にトランザクション手数料の総額(Total Fees)を計算します。この総額は、ブロックに追加された418個のトランザクションのインプットとアウトプットから計算され、トランザクション手数料は以下のようになります。

$$\text{Total Fees} = \text{Sum}(\text{Inputs}) - \text{Sum}(\text{Outputs})$$

ブロック277,316にあるトランザクション手数料の総額は0.09094928bitcoinです。

次に、JingのBitcoinノードは新しいブロックに対する正しい報酬を計算します。この報酬はブロック高に基づいて計算され、最初は1ブロックあたり50bitcoinから始まり210,000ブロックごとに半分に減っていきます。このブロックのブロック高は277,316であるため、この報酬は25bitcoinです。

この半分に減っていく計算は、[ブロックの報酬計算](#) — [Bitcoin Coreクライアントのmain.cppにあるGetBlockSubsidy関数](#)に示すようにBitcoin CoreクライアントのGetBlockSubsidy関数で行われます。

*Example 5.* ブロックの報酬計算 — *Bitcoin Core*クライアントのmain.cppにあるGetBlockSubsidy関数

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)
{
    int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
    // Force block reward to zero when right shift is undefined.
    if (halvings >= 64)
        return 0;

    CAmount nSubsidy = 50 * COIN;
    // Subsidy is cut in half every 210,000 blocks which will occur approximately
    every 4 years.
    nSubsidy >>= halvings;
    return nSubsidy;
}
```

初期報酬は、COIN定数(100,000,000satoshi)に50を掛けてsatoshi単位で表したものになっています。これによって初期報酬(nSubsidy)が50億satoshiになっています。

次に、この関数は半減ブロック間隔(SubsidyHalvingInterval)で現在のブロック高を割ることで半減数halvingsを計算しています。ブロック277,316の場合、半減ブロック間隔が210,000ブロックごとであるため半減数は1回となります。

許されている半減数の最大値は64回で、もし64回の半減数を越えるとこのコードでは新しいブロックに対す

る報酬が0になります(単にトランザクション手数料のみが返る)。

次に、この関数は2進数右シフト演算子を使って報酬(nSubsidy)を半減が起こるたびに2で割っています。ブロック277,316の場合、これは50億satoshiの報酬に対して1回だけ2進数右シフト演算(1回半減)を行い報酬は25億satoshi(25bitcoin)となります。2進数右シフト演算子は、整数または浮動小数点での割り算よりもより効果的に2で割ることができるために使われています。

最終的に、coinbase報酬( nSubsidy )にトランザクション手数料( nFees )が加えられ、この和が返されます。

## Generationトランザクションの構造

これらの計算を行うことで、JingのBitcoinノードは彼自身に25.09094928bitcoinを支払うgenerationトランザクションを構築します。

**Generationトランザクション**を見ると分かるように、generationトランザクションは特別なフォーマットをしています。使用する前のUTXOを特定するトランザクションインプットと違って、これは"coinbase"インプットを持っています。[tx\_in\_structure]の中でトランザクションインプットを説明しました。ここでは、通常のトランザクションインプットとgenerationトランザクションインプットを比較してみましょう。"通常"の**トランザクションインプットの構造**は通常のトランザクションの構造を示していて、**generationトランザクションインプットの構造**はgenerationトランザクションインプットの構造を示しています。

Table 1. "通常"のトランザクションインプットの構造

| サイズ              | フィールド名                | 説明  |
|------------------|-----------------------|---|
| 32byte           | Transaction Hash      | 使われるUTXOを含むトランザクションハッシュ                   |
| 4byte            | Output Index          | 使われるUTXOのトランザクション内インデックス、一番最初のアウトプットの場合は0 |
| 1-9byte (VarInt) | Unlocking-Script Size | unlocking-scriptのbyte長                    |
| 可変サイズ            | Unlocking-Script      | UTXOのlocking scriptを満たすscript             |
| 4byte            | Sequence Number       | 現在トランザクション置換は使用不可になっていて、0xFFFFFFFFに固定     |

Table 2. generationトランザクションインプットの構造

| サイズ              | フィールド名             | 説明                                  |
|------------------|--------------------|-------------------------------------|
| 32byte           | Transaction Hash   | 全てのbitが0であり、他のトランザクションハッシュの参照はしていない |
| 4byte            | Output Index       | 全てのbitが1: 0xFFFFFFFF                |
| 1-9byte (VarInt) | Coinbase Data Size | Coinbase Dataサイズの長さ(2から100byte)     |

| サイズ   | フィールド名          | 説明   |
|-------|-----------------|--|
| 可変サイズ | Coinbase Data   | バージョン2 ブロックのextra nonceやmining tagのために使われる任意のデータであり、ブロック高から始まらなければならない |
| 4byte | Sequence Number | 0xFFFFFFFF に固定   |

generationトランザクションでは、最初の2つのフィールドはUTXOへの参照を表現していない値が設定されています。通常のトランザクションの"Transaction Hash"の代わりに、最初のフィールドは全てが0の32byteで埋められています。"Output Index"は全てが0xFF(10進数で255)に設定された4byteで埋められています。"Unlocking Script"はCoinbase Dataで置き換えられており、マイナーによって使われる任意のデータを入れられるフィールドになっています。

## Coinbase Data

generationトランザクションはunlocking scriptフィールド(a.k.a, scriptSig)を持っていません。その代わりに、このフィールドはCoinbase Dataフィールドで置き換えられています。このフィールドに入るデータは2byteから100byteの間のデータになっていなければいけません。

例えばSatoshi Nakamotoは、genesisブロックのCoinbase Dataフィールドに日付と"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"というテキストを加えました。Satoshi Nakamotoはこれを日付の証明と伝えたいメッセージのために使用しています。以下の節で見ると、現在マイナーはCoinbase Dataフィールドをextra nonceやマイニングプールを特定する文字列を含めることに使っています。

Coinbase Dataフィールドの最初の数byteは任意に使われていましたが、現在はもはやこのようにはなっていません。Bitcoin Improvement Proposal 34 (BIP0034)にある通り、version-2 ブロック(ブロックのversionフィールドが2)では、scriptの"push"オペレーションのようにブロック高をCoinbase Dataフィールドの最初に入れておかなければいけません。

ブロック277,316では、このCoinbase Dataフィールド([Generationトランザクション参照](#))に03443b0403858402062f503253482f という16進数が含まれています。これはトランザクションインプットの"Unlocking script"または scriptSigフィールドにあります。これをデコードしてみましょう。

最初の1byte 03 はscript実行エンジンに次の3byteをscriptスタックにpushするという意味です([\[tx\\_script\\_ops\\_table\\_pushdata\]](#)参照)。次の3byte 0x443b04 は、リトルエンディアン(逆読み、最下位バイトが最初に来る)でエンコードされたブロック高です。バイトの順番を逆にして0x043b44にし、これを10進数で読むと277,316になります。

次の数個の16進数(03858402062)は、proof of work適切解を探すためのextra nonce ([extra nonceによる方法参照](#))、またはランダムな値をエンコードするために使われます。



Coinbase Dataフィールドの最後の部分(2f503253482f)は /P2SH/ の ASCIIコードで、このブロックを採掘したマイニングBitcoinノードがBIP0016で定義されている pay-to-script-hash (P2SH) をサポートしていることを示しています。P2SHの導入には、BIP0016 またはBIP0017のいずれを支持するかを示すためのマイナーによる"投票"が必要でした。BIP0016の実装は /P2SH/ をCoinbase Dataフィールドに含めることであり、BIP0017の実装は p2sh/CHV をCoinbase Dataフィールドに含めることでした。結果的にBIP0016の実装が勝者として選ばれ、多くのマイナーはP2SHの支持を示すために /P2SH/ をCoinbase Dataフィールドに含める続けることになったのです。

[genesisブロックからのCoinbase Dataフィールドの抽出](#)は[\[alt\\_libraries\]](#)で紹介した libbitcoinライブラリを使っており、genesisブロックからCoinbase Dataフィールドを取り出して Satoshiのメッセージを表示するものです。libbitcoinライブラリはgenesisブロックの静的なコピーを持っており、この例コードではこのライブラリから直接genesisブロックを取得することができます。

*Example 6. genesisブロックからのCoinbase Dataフィールドの抽出*

```
/*
   Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::block_type block = bc::genesis_block();
    // Genesis block contains a single coinbase transaction.
    assert(block.transactions.size() == 1);
    // Get first transaction in block (coinbase).
    const bc::transaction_type& coinbase_tx = block.transactions[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs.size() == 1);
    const bc::transaction_input_type& coinbase_input = coinbase_tx.inputs[0];
    // Convert the input script to its raw format.
    const bc::data_chunk& raw_message = save_script(coinbase_input.script);
    // Convert this to an std::string.
    std::string message;
    message.resize(raw_message.size());
    std::copy(raw_message.begin(), raw_message.end(), message.begin());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}
```

[satoshi-words例コードのコンパイルと実行](#)では、このコードをGNUコンパイラでコンパイルし、出力される実行ファイルを実行しています。

C++

## Example 7. satoshi-words例コードのコンパイルと実行

```
$ # Compile the code
$ g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the executable
$ ./satoshi-words
^D    <GS>^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

## ブロックヘッダの構築

ブロックヘッダを構築するために、このマイニングBitcoinノードは[ブロックヘッダの構造](#)にリストアップしてある6つのフィールドを埋める必要があります。

Table 3. ブロックヘッダの構造

| サイズ    | フィールド名              | 説明                                      |
|--------|---------------------|---|
| 4byte  | Version             | ソフトウェア/プロトコルバージョン番号                     |
| 32byte | Previous Block Hash | 一つ前のブロック(親ブロック)のハッシュ                    |
| 32byte | Merkle Root         | ブロック内の全トランザクションに関するmerkle treeのrootハッシュ |
| 4byte  | Timestamp           | ブロックのおおよその生成時刻(Unix秒)                   |
| 4byte  | Difficulty Target   | ブロック生成時のproof of workアルゴリズムのdifficulty  |
| 4byte  | Nonce               | proof of workアルゴリズムで用いられるカウンタ           |

ブロック277,316が採掘された時点でこのブロック構造を記述しているversionは2で、このブロックには4byteをリトルエンディアンでエンコードした 0x02000000 が入っています。

次に、このマイニングBitcoinノードは "1つ前のブロックハッシュ"を追加する必要があります。これはブロック277,315のブロックヘッダのハッシュです。このBitcoinネットワークから受け取った1つ前のハッシュは、JingのBitcoinノードが候補ブロック277,316の親として選んだブロックです。ブロック277,315のブロックヘッダハッシュは以下です。

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

次のステップは全てのトランザクションをmerkle treeにまとめることです。これはmerkle

rootをブロックヘッダに加えるためです。generationトランザクションはブロックの最初のトランザクションになっています。418個のトランザクションはこのgenerationトランザクションのあとに追加され、全部で419個のトランザクションがブロックの中にあることとなります。[\[merkle\\_trees\]](#)で見たように、merkle treeは偶数個の"葉"ノードを持たなければいけません。このため、最後のトランザクションは重複することになり、420個の葉ノードが作られます。それぞれ葉ノードはトランザクションのハッシュを保持しています。トランザクションハッシュはペアを組んで結びつけられ、merkle treeの階層を作っていく、全てのトランザクションがmerkle treeの"root(根幹)"にまとめられるまで続きます。merkle treeのrootは全てのトランザクションを1つの32byteの値にまとめています。この値は[ブロック277,316](#)にある"merkle root"を見ることで確認でき、その値は以下になっています。

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

このときマイニングBitcoinノードはUnixの"Epoch"タイムスタンプのようにエンコードされた4byteのタイムスタンプを追加します。Unixの"Epoch"タイムスタンプは、1970年1月1日 深夜0:00 UTC/GMTから経過した秒数に基づいています。時刻 1388185914 は2013年12月27日金曜日 23:11:54 UTC/GMTと同じです。

この後このBitcoinノードはdifficulty targetを埋めます。これはこのブロックを有効なブロックにするために必要なproof-of-work difficultyを定義しています。このdifficultyはブロック内に"difficulty bits"として保存されていて、指数表記の形でエンコードされています。このエンコーディングは1byteの指数部、3byteの仮数部(係数)を持っています。例えばブロック277,316の場合、difficulty bitsの値は 0x1903a30c です。最初の部分 0x19 は16進数指数部で、次の部分 0x03a30c は係数です。difficulty targetのコンセプトについては [Difficulty TargetとRetargeting](#) で説明し、"difficulty bits"表現については [Difficultyの表現](#) で説明します。

最後のフィールドはnonceで、初期値は0です。

全ての他のフィールドを埋めると、ブロックヘッダは完全なものとなりマイニングプロセスを始めることができます。このゴールはdifficulty targetよりも小さいブロックヘッダハッシュになります。nonceを見つけることです。必要条件を満たすようなnonceを見つける前に、このマイニングBitcoinノードは10億個または1兆個のnonceの値を調べてみる必要があるのです。

## ブロックのマイニング

今や候補ブロックはJingのBitcoinノードによって構築され、Jingのハードウェアマイニング専用マシンがブロックを"採掘"しブロックを有効にするproof-of-workアルゴリズムに対する解を見つけるときです。この本を通して、Bitcoinシステムのいろいろな面で暗号学的ハッシュ関数を使い学んできました。ハッシュ関数SHA256はBitcoinのマイニングプロセスの中で使われている暗号学的ハッシュ関数です。

最もシンプルに言うと、マイニングとは1つのパラメータを変えながらブロックヘッダを繰り返しハッシュ化するプロセスで、出力されるハッシュが特別な条件を満たすまで行われます。ハッシュ関数の結果を先に立て決めることはできず、また特別なハッシュ値を作り出すためのパターンを作り出すこともできません。この

ハッシュ関数の特徴は次のことを意味しています。特別な条件に合うハッシュを作り出すただ1つの方法は、入力をランダムに修正しながら偶然に欲しいハッシュが現れるまで試行を繰り返し繰り返し行うことです。

## Proof-Of-Workアルゴリズム

ハッシュアルゴリズムは任意の長さのデータ入力を取り、このデータ入力のデジタルフィンガープリントとして固定長の、決定性を持った結果を作り出します。どんな入力に対しても、入力と同じであれば出力されるハッシュは常に同じです。また、たやすく計算することができ、同じハッシュアルゴリズムを実装している人なら誰でも検証できます。暗号的ハッシュアルゴリズムのキーとなる特徴は、同じフィンガープリントを作り出す違った2つの入力を探すことは至難の業であることです。ランダムな入力を試す以外に出力としてほしいフィンガープリントを作り出す入力を選ぶということも考えられますが、この特徴の自然な帰結としてこのような入力を選ぶこともまた至難の技なのです。

SHA256だと出力結果は常に256bitの長さになり、これは入力のデータサイズに関わらず決まります。[SHA256での例](#)では、Pythonインタプリタを使ってフレーズ "I am Satoshi Nakamoto." のSHA256ハッシュを計算しています。

*Example 8. SHA256での例*

```
$ python
```

```
Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

[SHA256での例](#)は "I am Satoshi Nakamoto" のハッシュ 5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e を計算した結果を示しています。この256bitの数字はこのフレーズの ハッシュ または ダイジェストであり、このフレーズの全ての部分に依っています。1個の文字、句読点、または他のいかなる文字でも追加すると異なるハッシュが生成されます。

今、もしこのフレーズを変えると完全に違ったハッシュが生成されるはずですが、フレーズの最後に数字を追加して作った新しいフレーズのハッシュが全く違うハッシュになることを試してみましょう。ハッシュ生成には [nonce生成を繰り返すことで多くのSHA256ハッシュを生成するスクリプト](#) にあるシンプルな Pythonスクリプトを使います。

Example 9. nonce生成を繰り返すことで多くのSHA256ハッシュを生成するスクリプト

```
# example of iterating a nonce in a hashing algorithm's input

import hashlib

text = "I am Satoshi Nakamoto"

# iterate nonce from 0 to 19
for nonce in xrange(20):

    # add the nonce to the end of the text
    input = text + str(nonce)

    # calculate the SHA-256 hash of the input (text+nonce)
    hash = hashlib.sha256(input).hexdigest()

    # show the input and hash result
    print input, '=>', hash
```

これを実行すると、テキストの最後に数字を追加して違った形に作られたいくつかのフレーズのハッシュが生成されます。[nonce生成を繰り返すことで多くのSHA256ハッシュを生成するスクリプト実行出力](#)に示している通り、数字を1つずつ増やしていくと違ったハッシュを得ることができます。



targetより小さいハッシュを見つけることはさらに難しくなります。

簡単にイメージしやすくするために、プレイヤーが2つのサイコロを繰り返し投げ、特定のtargetよりも和が小さくなるようにするゲームを想像してみましょう。最初のラウンドではtargetは12です。6を2つ出さなければあなたの勝ちです。次のラウンドではtargetを11にしましょう。プレイヤーは10かそれより小さくしなければいけません、これも簡単です。数回やってみたとtargetを5に下げてください。今、サイコロを投げたうち半分以上が5よりも大きくなってしまいました。targetがより低くなればなるほど、勝つために投げるサイコロの回数は指数関数的に大きくなっていきます。結局、targetが2(最低限の数値)では、36回サイコロを投げるうちたった1回、つまり全体の2%だけがゲームに勝つ結果を生成することになります。

nonce生成を繰り返すことで多くのSHA256ハッシュを生成するスクリプト実行出力では、勝利できる "nonce" は13でこの結果は誰でも独立に確認することができます。誰でも数値13をさきほどのフレーズ "I am Satoshi Nakamoto" の最後に追加し、ハッシュを計算し、targetより小さいことを検証することができます。そして、この検証結果はproof of workでもあるのです。というのは、これがあのnonceを見つける作業をしたことの証明だからです。検証するためにはたった1回のハッシュ計算でよい一方、うまくいくnonceを見つけるためには13回のハッシュ計算が必要になります。もしより低いtarget(より高いdifficulty)を使ったとすると、適したnonceを探すためにさらにたくさんのハッシュ計算が必要になるのですが、検証するには誰でも1回のハッシュ計算で済みます。さらに、targetを知ることによって、統計を使って誰でもdifficultyの見積もりをすることができ、よってどれだけの仕事かそのようなnonceを見つけるために必要だったかを知ることができるのです。

Bitcoinのproof of workはnonce生成を繰り返すことで多くのSHA256ハッシュを生成するスクリプト実行出力に示した課題ととても似ています。まずマイナーはトランザクションで埋められた候補ブロックを構築します。次に、ブロックのヘッダのハッシュを計算し、現在のtargetより小さいかどうかを確認します。もしそのハッシュがtargetよりも小さくなければ、マイナーはnonceを修正し(通常は1つ増加させるだけです)、再びハッシュを計算します。現在のBitcoinネットワークのdifficultyでは、ブロックヘッダハッシュが十分小さくなるnonceを見つける前にマイナーは1000兆回ハッシュを計算する必要があります。

とても簡単化されたproof-of-workアルゴリズムは簡略化されたproof-of-work実装にPythonで実装されています。

Example 11. 簡略化されたproof-of-work実装

```
#!/usr/bin/env python
# example of proof-of-work algorithm

import hashlib
import time

max_nonce = 2 ** 32 # 4 billion

def proof_of_work(header, difficulty_bits):

    # calculate the difficulty target
    target = 2 ** (256-difficulty_bits)
```

```

for nonce in xrange(max_nonce):
    hash_result = hashlib.sha256(str(header)+str(nonce)).hexdigest()

    # check if this is a valid result, below the target
    if long(hash_result, 16) < target:
        print "Success with nonce %d" % nonce
        print "Hash is %s" % hash_result
        return (hash_result,nonce)

print "Failed after %d (max_nonce) tries" % nonce
return nonce

if __name__ == '__main__':

    nonce = 0
    hash_result = ''

    # difficulty from 0 to 31 bits
    for difficulty_bits in xrange(32):

        difficulty = 2 ** difficulty_bits
        print "Difficulty: %ld (%d bits)" % (difficulty, difficulty_bits)

        print "Starting search..."

        # checkpoint the current time
        start_time = time.time()

        # make a new block which includes the hash from the previous block
        # we fake a block of transactions - just a string
        new_block = 'test block with transactions' + hash_result

        # find a valid nonce for the new block
        (hash_result, nonce) = proof_of_work(new_block, difficulty_bits)

        # checkpoint how long it took to find a result
        end_time = time.time()

        elapsed_time = end_time - start_time
        print "Elapsed Time: %.4f seconds" % elapsed_time

        if elapsed_time > 0:

            # estimate the hashes per second
            hash_power = float(long(nonce)/elapsed_time)
            print "Hashing Power: %ld hashes per second" % hash_power

```



このコードを実行すると、欲しいdifficultyを設定でき(difficultyはbit単位。左から数えた桁が何桁0でなければいけないか)、あなたのコンピュータが解を探すためにどれくらい時間がかかるかを確認できます。そして、[いろいろなdifficultyに対するproof of work](#)例コードの実行では平均的なノートパソコンを使うとどのようになるかを見ることができます。

Example 12. いろいろなdifficultyに対するproof of work例コードの実行

```
$ python proof-of-work-example.py*
```

```
Difficulty: 1 (0 bits)
```

```
[...]
```

```
Difficulty: 8 (3 bits)
```

```
Starting search...
```

```
Success with nonce 9
```

```
Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1
```

```
Elapsed Time: 0.0004 seconds
```

```
Hashing Power: 25065 hashes per second
```

```
Difficulty: 16 (4 bits)
```

```
Starting search...
```

```
Success with nonce 25
```

```
Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148
```

```
Elapsed Time: 0.0005 seconds
```

```
Hashing Power: 52507 hashes per second
```

```
Difficulty: 32 (5 bits)
```

```
Starting search...
```

```
Success with nonce 36
```

```
Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903
```

```
Elapsed Time: 0.0006 seconds
```

```
Hashing Power: 58164 hashes per second
```

```
[...]
```

```
Difficulty: 4194304 (22 bits)
```

```
Starting search...
```

```
Success with nonce 1759164
```

```
Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cefc3
```

```
Elapsed Time: 13.3201 seconds
```

```
Hashing Power: 132068 hashes per second
```

```
Difficulty: 8388608 (23 bits)
```

```
Starting search...
```

```
Success with nonce 14214729
```

```
Hash is 000001408cf12dbd20fcb6372a223e098d58786c6ff93488a9f74f5df4df0a3
```

```
Elapsed Time: 110.1507 seconds
```

```
Hashing Power: 129048 hashes per second
```

```
Difficulty: 16777216 (24 bits)
Starting search...
Success with nonce 24586379
Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95
Elapsed Time: 195.2991 seconds
Hashing Power: 125890 hashes per second
```

[...]

```
Difficulty: 67108864 (26 bits)
Starting search...
Success with nonce 84561291
Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a
Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second
```

見て分かるように、difficultyが1bit増えると解を見つけるためにかかる時間が指数関数的に大きくなります。256bitの数値空間全体を考えてみると、0にしなければいけない制約が1bit増えるとたびに解となる空間が半分になってしまうのです。いろいろなdifficultyに対するproof of work例コードの実行では、左から26bitまでが0になっているようなハッシュを作り出すnonceを見つけるために、8400万回の試行が必要になります。仮に1秒間に12万回以上ハッシュ計算ができるコンピュータがあったとしても、この解を見つけるために10分間もかかってしまいます。

執筆している時点で、Bitcoinネットワークは

```
0000000000000004c296e6376db3a241271f43fd3f5de7ba18986e517a243baa7
```

より小さいヘッダハッシュを持つブロックが試行されています。見て分かるように、ハッシュの最初に多くの0があります。これは、許容されるハッシュ範囲がとても小さくなっているということを意味し、よって有効なハッシュを見つけることがより難しくなっているということになります。次のブロックをBitcoinネットワークが発見するために平均的に秒間15京回(150 quadrillion hash)以上のハッシュ計算が必要になっています。これは不可能なことに見えますが、幸運なことにBitcoinネットワークは秒間100ペタハッシュ(PH/sec、1ペタは1000兆)の演算処理能力を提供しており、これにより平均的に約10分間ごとにブロックを見つけることができます。

## Difficultyの表現

ブロック277,316では、ブロックが"difficulty bits"または単に"bits"と呼ばれる記法で書かれたdifficulty targetを含んでいることを確認しました。ブロック277,316では+0x1903a30c+という値がdifficulty bitsに入っています。この記法はdifficulty targetを係数部/指数部形式で表すもので、最初の2桁の16進数が指数部(exponent)、次の6桁の16進数が係数(coefficient)です。このブロックでは、指数部が 0x19、係数が 0x03a30c となっています。

この記法からdifficulty targetを計算する数式は以下になります。

$$\text{target} = \text{coefficient} * 2^{(8 * (\text{exponent} - 3))}$$



ます。実際にかかった時間と求められる時間との比が計算され、適した調整(difficultyを上げるまたは下げる)が行われます。もしBitcoinネットワークが10分毎よりも速くブロックを見つけていればdifficultyは上がります。もしブロックの発見が予想よりも遅ければ、difficultyは下がります。

この関係式は以下のようにまとめられます。

$$\text{New Difficulty} = \text{Old Difficulty} * (\text{Actual Time of Last 2016 Blocks} / 20160 \text{ minutes})$$

[proof-of-workのdifficulty retergeting](#) — [pow.cppのCalculateNextWorkRequired\(\)関数](#)はBitcoin Coreクライアントのなかで使われているコードを示しています。

*Example 13. proof-of-workのdifficulty retergeting — pow.cppのCalculateNextWorkRequired()関数*

```
// Limit adjustment step
int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
LogPrintf(" nActualTimespan = %d before bounds\n", nActualTimespan);
if (nActualTimespan < params.nPowTargetTimespan/4)
    nActualTimespan = params.nPowTargetTimespan/4;
if (nActualTimespan > params.nPowTargetTimespan*4)
    nActualTimespan = params.nPowTargetTimespan*4;

// Retarget
const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
arith_uint256 bnNew;
arith_uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= params.nPowTargetTimespan;

if (bnNew > bnPowLimit)
    bnNew = bnPowLimit;
```

#### NOTE

difficultyの調整は2016ブロックに1回起きます。オリジナルのBitcoin Coreクライアントにあるoff-by-oneエラーのため、difficultyの調整は前の2015ブロックの総時間に基づいています(本来すべき2016ブロックの総時間ではなく)。この結果、difficultyは0.05%だけ高くなるようなretargetingバイアスが生じています。

Interval(2,016ブロック)とTargetTimespan(2週間、1,209,600秒)は *chainparams.cpp* に定義されています。

difficultyが極端に動きすぎないように、retargetingは調整ごとに4倍または1/4以内になるようになっています。つまり、もし必要なdifficultyの調整が4倍よりも大きいまたは1/4よりも小さい場合は、最大でも4倍、最小でも1/4になりそれを越えたものにはなりません。不均衡が次の2,016ブロックの間続いてしまうため、さらなる調整は次のretargetingのときに行われます。このため、ハッシュ生成速度とdifficultyの大きな食い違い

は数回のretargetingを経て均衡するようになります。

#### TIP

Bitcoinブロックを発見するdifficultyは、前の2,016ブロックを発見するためにかかった時間に基づいて計算され、ネットワーク全体でだいたい'10分間'になっています。difficultyは、2,016ブロックごとに調整されていきます。

#### difficulty

targetはトランザクションの数やトランザクションに含まれる資金には依存しないことを注意しておきます。これは、ハッシュ生成速度、つまりはBitcoinをセキュアに保つために費やされる電気代もまたトランザクションの数に全く依存しないということです。これにより、今日のハッシュ生成速度が増加しなかったとしても、Bitcoinはより広く採用されスケールアップすることができ、セキュアに保たれるということです。ハッシュ生成速度が大きくなるということは、マーケットに参入した新しいマイナーに厳しい報酬競争を強いることとなります。十分なハッシュ生成速度が率直に報酬を狙うマイナーによってコントロールされている限り、"買収(takeover)"攻撃を防ぎBitcoinを安全に保つことができるのです。

#### difficulty

targetは電気代および

bitcoinと電気代を払うための通貨の交換レートに密接に関係しています。ハイパフォーマンスなマイニングシステムは最近のシリコン製造技術を用いて可能な限り効率化されており、電気をできる限り最高のレートでハッシュ生成計算に転換しています。マイニングマーケット上の主要な影響は1KW/hあたり何bitcoinかかるかです。なぜなら、これがマイニングの収益を決定しマイニングマーケットに参入するか撤退するかを決めるからです。

## うまくいったブロックの採掘

前に見たように、JingのBitcoinノードは候補ブロックを構築しマイニングの準備が整いました。JingはASIC(application-specific integrated circuits)で作られたいくつかのハードウェアのマイニング専用マシンを持っています。ASICは数十万個の集積回路で並行してSHA256アルゴリズムを計算するもので、信じられないほどのハッシュ生成速度を出します。これらの特別なマシンは彼のマイニングノードにUSBを通して接続されています。次に、Jingのデスクトップで動いているマイニングノードはブロックヘッダをマイニングハードウェアに送信し、ここから秒間10億回ものnonceの試行が始まります。

ブロック277,316の採掘を初めてから11分後くらいに1つのハードウェアマイニング専用マシンが解を見つけ、マイニングノードにそれを送り返しました。ブロックヘッダにそれを入れてみると、nonce 4,215,469,401が以下のブロックハッシュを生成することがわかりました。

```
00000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

これは以下のtargetよりも小さいものです。

```
00000000000000003A30C0000000000000000000000000000000000000000000
```

すぐに、Jingのマイニングノードはブロックを全てのピアに送信しました。彼らはこのブロックを受け取り、検証し、この新しいブロックを次に伝搬します。このブロックがBitcoinネットワークを波紋のように伝搬していくときに、それぞれのBitcoinノードはこのブロックを自身のブロックチェーンのコピーに追加しブロッ

ク高を277,316に増やします。マイニングノードがこのブロックを受け取り検証したとき、マイニングノードは同じブロック高のブロックの発見を諦め、すぐに次のブロックの計算を始めます。

次の節では、ブロックを検証し最も長いブロックチェーンを選ぶことで、分散されたブロックチェーンで合意形成を作るプロセスを見ていきます。

## 新しいブロックの検証

Bitcoinのコンセンサスメカニズムの3つ目のステップは、独立した全てのBitcoinノードによる新しいブロック検証です。新しく解決されたブロックがBitcoinネットワークを移動する時、それぞれのBitcoinノードはブロックをピアに送信する前に有効なブロックかどうかを確認するテストを実行します。これは、有効なブロックだけがBitcoinネットワークを伝搬するようにするためです。また、独立した検証は、誠実なマイナーがそれらのブロックをブロックチェーンに合体させ報酬を稼ぐことを保証しています。悪意のあるマイナーが作ったブロックがあった場合他のBitcoinノードから拒否されてしまうため報酬が得られないだけでなくproof-of-workで費やした努力を無駄にし何の埋め合わせもなく電気代を負うこととなります。

Bitcoinノードが新しいブロックを受け取った時、全てのブロックが満たすべき長い条件リストに照らし合わせて検証されます。もし条件を満たさなければブロックは拒否されます。これらの条件はBitcoin

Coreクライアントの中の `CheckBlock` 関数や `CheckBlockHeader` 関数で確認でき、以下の条件を含んでいます。

- ブロックのデータ構造が構文的に有効であること
- ブロックヘッダハッシュがdifficulty targetよりも小さいこと(proof of workを強制する)
- ブロックのTime Stampがノードが持つ時刻より2時間未来の時刻よりも小さいこと(ノードごとの時刻エラー(時刻違い)のある程度の許容)
- ブロックサイズが受け入れられる制限内であること
- 最初のトランザクション(そして、最初のトランザクションのみ)がcoinbase generationトランザクションであること
- ブロックに含まれる全てのトランザクションが独立したトランザクション検証で説明したチェックリストを満たすこと

全てのBitcoinノードによって独立に行われる検証によって、マイナーがごまかして不正をできないようになっています。前の節で、どのようにマイナーが新しいbitcoinをマイナーに与えるトランザクションをブロック内に書き、トランザクション手数料を要求するのかが確認しました。なぜマイナーは正しい報酬の代わりに彼ら自身で数千bitcoinを自身に与えるトランザクションを書かないのでしょうか。これは、全てのBitcoinノードが同じルールに従ってブロックを検証しているからです。不正なcoinbaseトランザクションがあった場合ブロック全体が無効になってしまい、結局このブロックは拒否されブロックチェーンの一部にはならないことになってしまいます。マイナーは全てのBitcoinノードが従っている共有ルールに基づく完全なブロックを構築する必要があり、しかもproof of workの正しい解を伴った形で採掘しなければいけません。これを行うために、マイニングに多大な電気を使います。もし彼らが不正を行えば、全ての電気と努力は無駄になってしまいます。これが、独立した検証が分散された合意形成(コンセンサス)のキーポイントである理由なのです。

## ブロックのチェーン組み立てと選択

Bitcoinの分散されたコンセンサスメカニズムの最後のステップは、ブロックをチェーンに組み込むことと、最も多くのproof of workを含むチェーンの選択です。一度

Bitcoinノードが新しいブロックを有効であると確認すると、このノードが持っている既存のブロックチェーンに新しいブロックを結びつけてチェーンを再構成しようとします。

Bitcoinノードは3種類のブロックセットを持っています。1つはメインのブロックチェーンにひも付けられたブロック。1つはメインのブロックチェーンから枝分かかれたブロック (セカンダリーチェーン)。もう1つはすでに知っているブロックチェーンに親がないブロック (orphans)です。無効なブロックは検証条件を満たさなかった時点ですぐに拒否されるためどのチェーンにも含まれません。

"メインチェーン"はどんなときでも、累積difficultyが最も多くなっているチェーンになっています。同じ長さのチェーンがあり片方がより多くのproof of workを持っている場合を除き、ほとんどの状況下ではこれは最も多くのブロックを持っているチェーンということになります。メインチェーンは、メインチェーンのブロックに繋がった"sibling(兄弟)"ブロックのブランチを持つこともあります。これらのブロックは有効ですがメインチェーンの一部ではありません。これらが保持されているのは、将来これらのチェーンのうちの1つがメインチェーンをdifficultyで上回りsiblingブロック側のチェーンに拡張されていく場合に参照できるようにするためです。次の節(Blockchainフォーク)では、ほとんど同時に同じブロック高を持つブロックが採掘された結果としてどのようにセカンダリーチェーンが生じるかを説明します。

新しいブロックを受け取った時、Bitcoinノードはすでにあるブロックチェーンにブロックを追加しようとします。このBitcoinノードはブロックの"previous block hash"フィールドを確認します。previous block hashフィールドは新しいブロックの親を参照しています。このBitcoinノードは対応した親を探そうとします。ほとんどの場合、親はメインチェーンの"先頭"にあり、これが意味することはメインチェーンが新しいブロックで拡張されるということです。例えば、新しいブロック 277,316 が親ブロック 277,315 のブロックハッシュを参照しているような場合です。ブロック 277,316 を受け取ったほとんどのBitcoinノードはすでにメインチェーンの先頭ブロックとしてブロック 277,315 を持っており、よって新しいブロックを連結しメインチェーンを拡張することになるのです。

[Blockchainフォーク](#)で見ると、時々新しいブロックがメインチェーン以外のチェーンを拡張するときがあります。この場合、Bitcoinノードは新しいブロックをセカンダリーチェーンにくっつけ、セカンダリーチェーンとメインチェーンのdifficultyを比較します。もしセカンダリーチェーンの累積difficultyがメインチェーンの累積difficultyを上回ってれば、Bitcoinノードはセカンダリーチェーンに 中心を移し(reconverge)ます。このことは、セカンダリーチェーンを新しいメインチェーンとして選び、古いメインチェーンをセカンダリーチェーンにするということを意味しています。もしこのBitcoinノードがマイナーであれば、この新しくより長いチェーンを拡張していくようにブロックを構築していくことになります。

もし有効なブロックを受け取っても既存のチェーンに親が見つからなかった場合、このブロックは"orphan(孤児)"とみなされます。orphanブロックは、これらの親を受け取るまでorphanブロックプールに保持されます。この親をBitcoinネットワークから受け取り既存のチェーンに連結すると、orphanブロックはorphanプールから取り出されてこの親に連結され、チェーンの一部となります。orphanブロックは、ほぼ同時に採掘された2つのブロックを逆順(親より前に子)で受け取ったときに通常生じるものです。

最も大きいdifficultyを持つチェーンを選ぶことで、全てのBitcoinノードは結局ネットワーク全体のコンセンサスに到達し、一時的なチェーン同士の不一致はより多くのproof of workが追加されるにつれて結局は解決されます。マイニングノードはどのチェーンが拡張されていくかを選ぶことでマイニングパワーで(チェーンに)"投票"していることになります。このマイニングノードが新しいブロックを採掘しチェーンを拡張するとき、新しいブロックそのものがマイニングノードの投票を表すのです。

次の節では、最も長いdifficultyチェーンを独立に選ぶことでどのようにしてチェーン同士(フォーク同士)の競争による不一致を解決しているかを見ていきます。

## Blockchain フォーク

ブロックチェーンは分散化されているため、ノードごとの異なったコピーが常に一致しているわけではありません。ブロックが異なったBitcoinノードに別々のタイミングで到着するかもしれず、ノードごとにブロックチェーンの状態は変わってしまうのです。これを解決するために、それぞれのBitcoinノードは常に最も多くのproof of workを持っているブロックのチェーンを選び拡張しようとしています。このブロックのチェーンは最長チェーン(the longest chain)または最大累積difficultyチェーン(the greatest cumulative difficulty chain)とも呼ばれています。チェーンの各ブロックに記録されているdifficultyを足し合わせることで、Bitcoinノードはこのチェーンを作るために使われたproof of workの総量を計算できます。全てのBitcoinノードが最大累積difficultyチェーンを選んでいる限り、グローバルなBitcoinネットワークは結果的に矛盾のない状態に収束します。フォークはブロックチェーンの異なるバージョン間での一時的な不一致によって生じますが、多くのブロックがフォークのうちの一つに追加されることで結果的に不一致が解消されるようになります。

次のいくつかの図を使って、Bitcoinネットワークの中でどのように"フォーク"が生じるのかを見ていきます。これらの図はグローバルに広がるBitcoinネットワークを簡略化した表現です。実際には、Bitcoinネットワークのトポロジーは地理的に組織されているわけではなく、むしろBitcoinノード間のメッシュネットワークとして構成されています。ネットワーク的に近くても地理的にはとても離れているかもしれません。地理的トポロジーの表現はフォークを図解する目的に対して用いられている簡略化なのです。実際のBitcoinネットワークではBitcoinノード間の"距離"をノードからノードへの"ホップ数"によって測っており、物理的な距離ではありません。図解の目的のため、異なったブロックは異なった色で表され、異なったブロックがBitcoinネットワークを通して広がり通過していったコネクションはブロックの色と同じ色に塗られています。

最初の図(ブロックチェーンのフォークが生じる過程の可視化 - フォークが生じる前)では、Bitcoinネットワークが同じブロックチェーンを持っている状態を表し、青色で表されたブロックはメインチェーンの先端ブロックを表します。



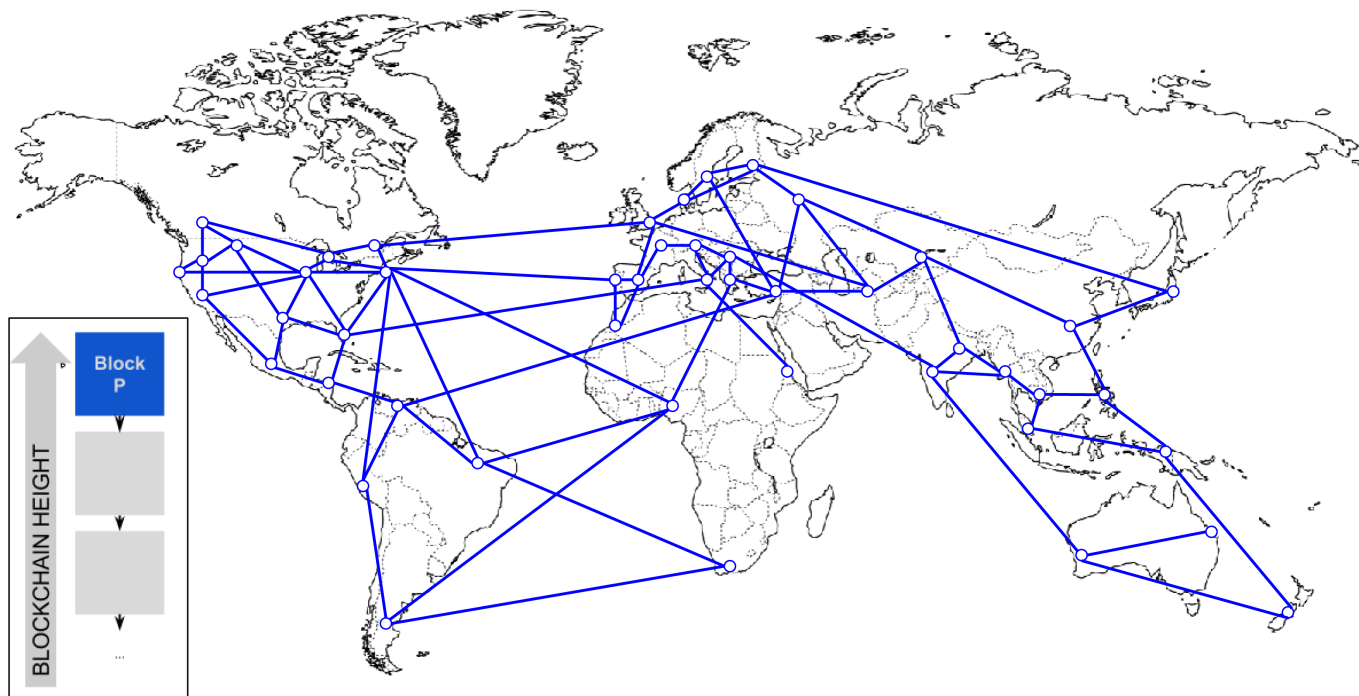


Figure 2. ブロックチェーンのフォークが生じる過程の可視化 - フォークが生じる前

"フォーク"は競争している2つの候補ブロックがあればいつでも生じる可能性があり、通常2人のマイナーが互いにほぼ同時刻にproof-of-workの解を見つけると起こります。両方のマイナーがそれぞれの候補ブロックに対する解を見つけると、彼らは直ちに自身の"勝ち取った"ブロックを近接ノードにブロードキャストし、近接ノードはこのブロックを次々にBitcoinネットワークに伝搬させていきます。有効なブロックを受け取ったBitcoinノードはこのブロックをローカルのブロックチェーンに追加し、1ブロックだけブロックチェーンを拡張します。もしこのBitcoinノードが同じ親を持つ別の候補ブロックをのちに見つけた場合は、セカンダリーチェーンに後から来た候補ブロックをつなげます。結果として、いくつかのBitcoinノードは最初の候補ブロックを"見て"、他のBitcoinノードは別の候補ブロックを見ることになるため、互いにぶつかる2つのブロックチェーンが生じることになるのです。

### ブロックチェーンのフォークが生じる過程の可視化

2

2つのブロックが同時に見つかったでは、ほぼ同時に異なるブロックを採掘した2人のマイナーを表しています。これらのブロックは両方とも青色のブロックの子供で、青色のブロックの上に追加しチェーンを拡張します。ブロックを追跡しやすくするために、カナダで作られたブロックを赤色、オーストラリアで作られたブロックを緑色にしてあります。

例として、カナダのマイナーが"赤色"のブロックに対するproof-of-workの解を見つけたとします。ほぼ同時に、オーストラリアのマイナーは"緑色"のブロックに対する解を見つけました。この時点で、2つの可能なブロックがあり、カナダで作られた方を"赤色"、オーストラリアで作られた方を"緑色"と呼ぶことにします。両方のブロックが有効であり、proof-of-workに対する有効な解を持っており、また同じ親ブロックを拡張するブロックとなっています。両方のブロックがおそらく大方同じトランザクションを持っており、違いとしてはトランザクションの順番くらいです。

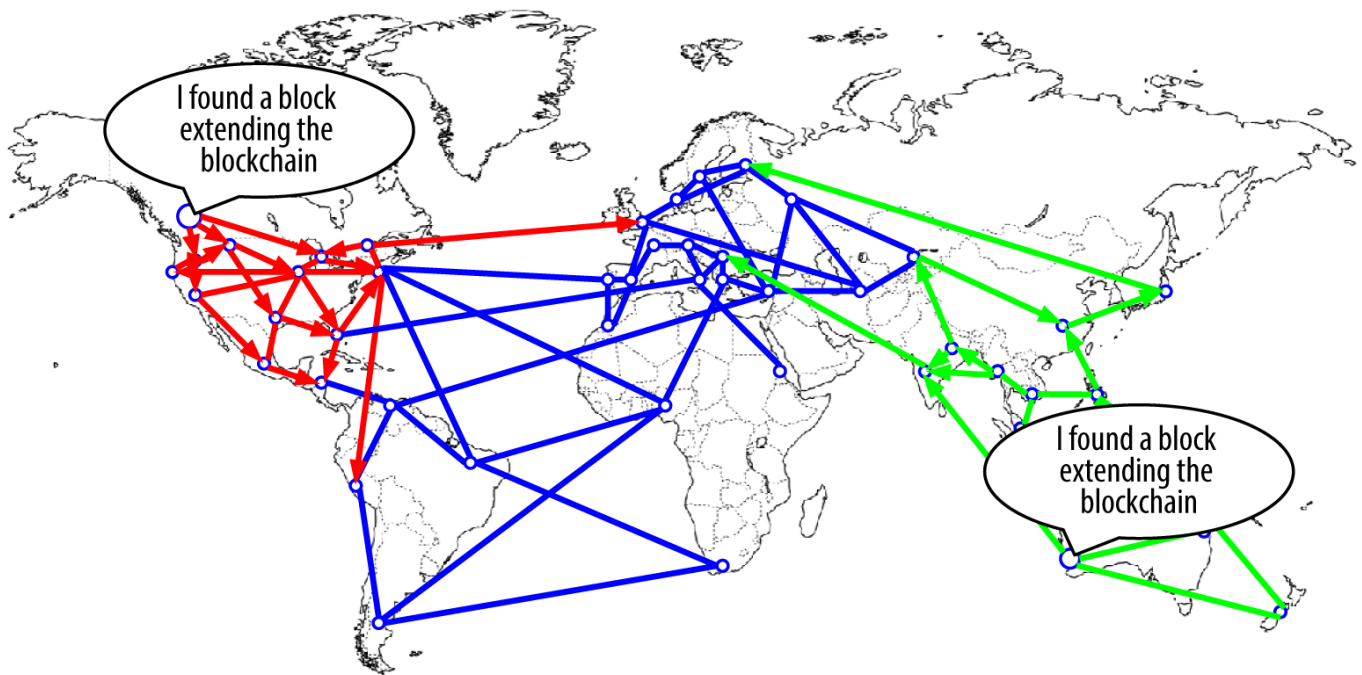


Figure 3. ブロックチェーンのフォークが生じる過程の可視化 - 2つのブロックが同時に見つかった

2つのブロックが伝搬するときに、いくつかのBitcoinノードは"赤色"のブロックを最初に受け取り、いくつかのBitcoinノードは"緑色"のブロックを最初に受け取ります。ブロックチェーンのフォークが生じる過程の可視化 - 2つのブロックが伝搬しBitcoinネットワークを2つに分割しているように、Bitcoinネットワークは2つのブロックチェーンに分かれてしまい、片側は赤色のブロックが先端にあり、もう1つは緑色のブロックが先端にあるようになっています。

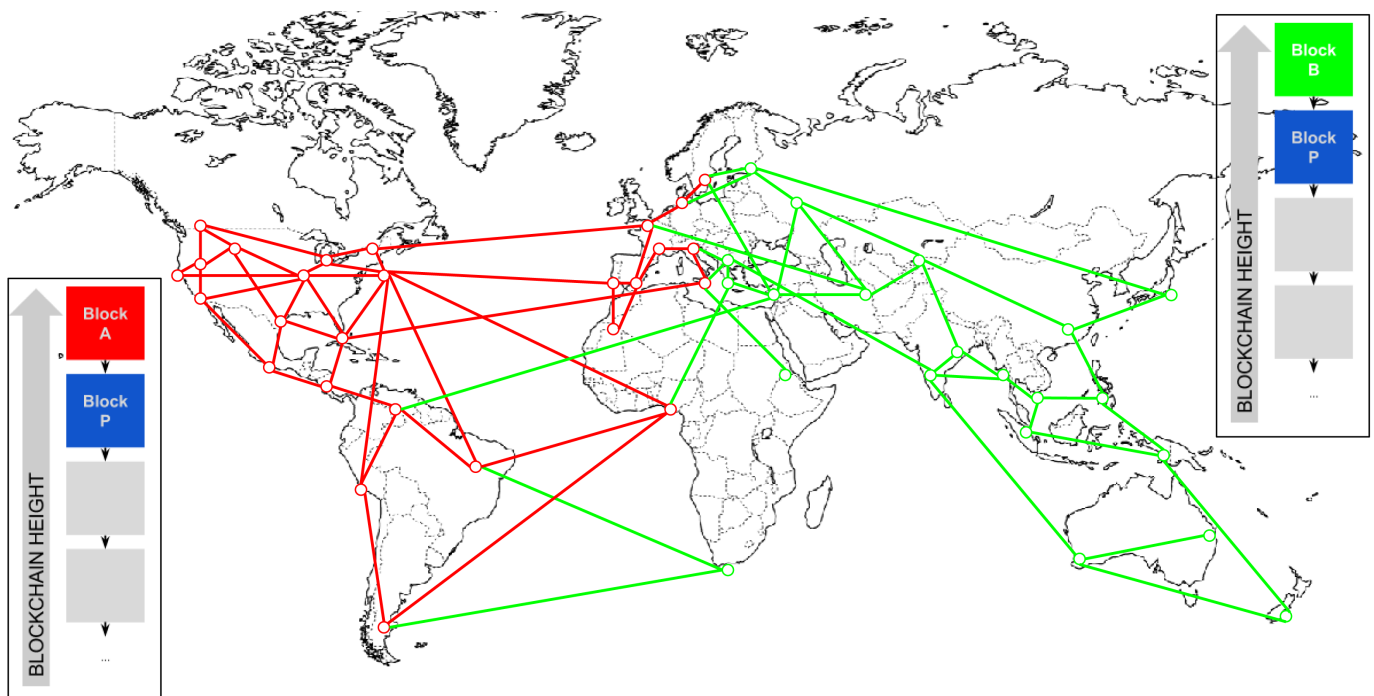


Figure 4. ブロックチェーンのフォークが生じる過程の可視化 - 2つのブロックが伝搬しBitcoinネットワークを2つに分割している

その瞬間から、カナダのBitcoinノードに(地理的ではなくトポロジー的に)最も近いBitcoinノードは最初に"赤

色"のブロックを受け取り、ブロックチェーンの最新のブロックとして"赤色"のブロックを持った新しいブロックチェーン(最も大きい累積difficultyを保持)を生成します(例えば、青-赤と繋がるブロックチェーン)。そして、少し後に届いた"緑色"の候補ブロックは無視することになります。一方、オーストラリアのBitcoinノードに近いBitcoinノードはオーストラリアのBitcoinノードが発見したブロックを受け取り、最新のブロックとして"緑色"のブロックを付ける形でブロックチェーンを拡張します(例えば、青-緑と繋がるブロックチェーン)。そして、数秒あとに届いた"赤色"のブロックは無視することになります。"赤色"のブロックを最初に見たどんなマイナーもすぐに親として"赤色"のブロックを参照する候補ブロックを構築し、これらの候補ブロックに対するproof of workを解き始めます。一方、"緑色"のブロックを受け入れたマイナーは"緑色"のブロックを頂点とするブロックチェーンを構築しこのブロックチェーンを拡張し始めます。

フォークはほとんど常に1ブロック以内で解決されます。"赤色"のブロックを親とする一部のBitcoinネットワークのハッシングパワーが"赤色"を親とするブロックチェーンの構築に投じられ、また別のBitcoinネットワークのハッシングパワーは"緑色"を親とするブロックチェーンの構築に投じられます。たとえハッシングパワーがほぼ均等に分割されてしまったとしても、あるマイナーが解を見つけ他の解を見つけたマイナーよりも前にそれを伝搬することになります。例えば、"緑色"のブロックを頂点に持つブロックチェーンを構築しているマイナーが"ピンク色"の新しいブロックを見つけてブロックチェーンを拡張する(例えば、青-緑-ピンクと繋がるブロックチェーン)と考えてみましょう。彼らはすぐにこの新しいブロックを伝搬し、[ブロックチェーンのフォークが生じる過程の可視化](#) - [新しいブロックが1つのフォークを拡張にあるように](#) Bitcoinネットワーク全体がこのブロックを有効な解として確認するようになります。

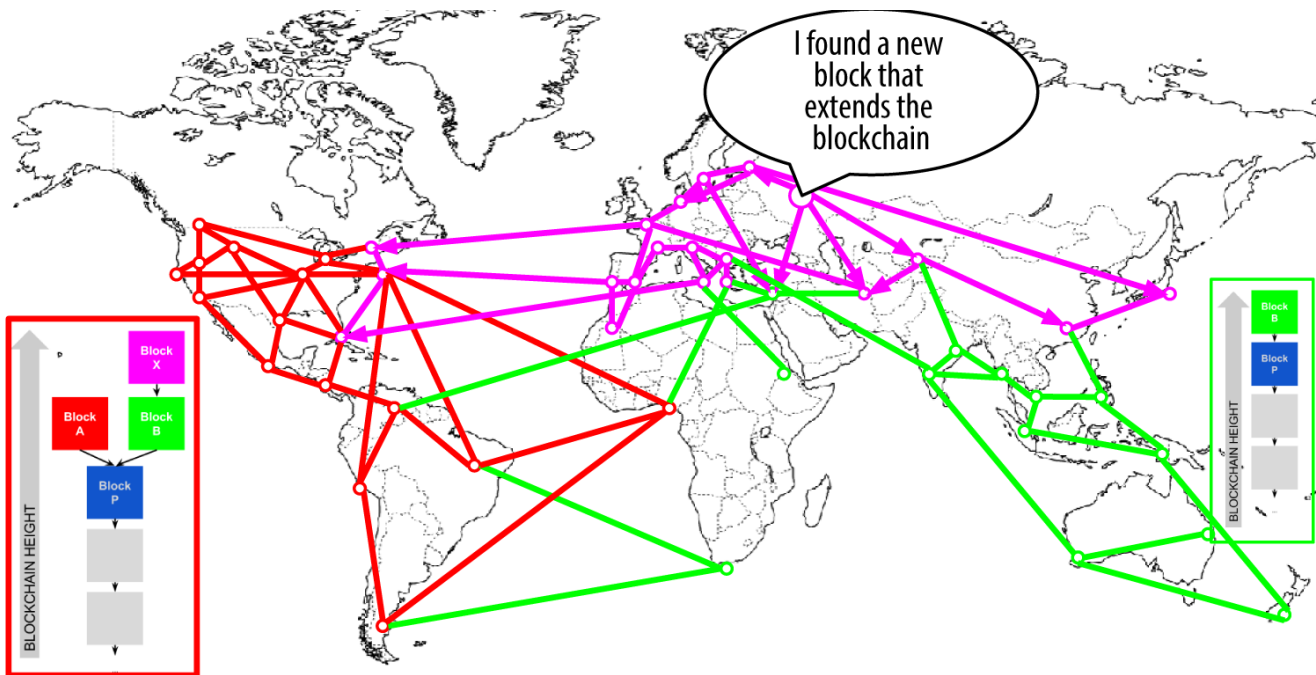


Figure 5. ブロックチェーンのフォークが生じる過程の可視化 - 新しいブロックが1つのフォークを拡張

前のラウンドで"緑色"のブロックの勝者として選んだ全てのBitcoinノードは、単にさらに1個ブロックをブロックチェーンに拡張していただくだけです。しかし、"赤色"のブロックを勝者として選んだBitcoinノードは2つのブロックチェーンを見ることになります。青-緑-ピンクのブロックチェーンと、青-赤のブロックチェーンです。今では青-緑-ピンクのブロックチェーンは青-赤のブロックチェーンよりも長くなっています(より多くの累積difficultyを持っている)。[ブロックチェーンの](#)

## フォークが生じる過程の可視化

ネットワークが新しい最長ブロックチェーンに再収縮するにあつたように、結果として、これらのBitcoinノードは青-緑-ピンクのブロックチェーンをメインチェーンとして選び、青-赤のブロックチェーンをセカンダリーチェーンに変更します。これがブロックチェーンの再収縮(reconvergence)で、より長いブロックチェーンの新しい情報を吸収するためにこれらのBitcoinノードがブロックチェーンの見方を変更することを強制されることで起こります。青-赤のブロックチェーンを拡張しようとしているどんなマイナーもこの拡張をやめます。というのは、彼らの候補ブロックの親がもはや最長ブロックチェーン上にはなく、この候補ブロックが"孤児(orphan)"になってしまったためです。"赤色"ブロックの中にあつたトランザクションは次のブロックの中で処理されるために再度マイニング対象になります。"赤色"ブロックはもはやメインチェーンにはないのです。Bitcoinネットワーク全体が青-緑-ピンクの1つのブロックチェーンに再収縮すると、"ピンク色"のブロックがブロックチェーンの最新ブロックとなります。青-緑-ピンクのブロックチェーンを拡張するために、全てのマイナーがすぐに"ピンク色"のブロックを親として参照している候補ブロックで作業を開始します。

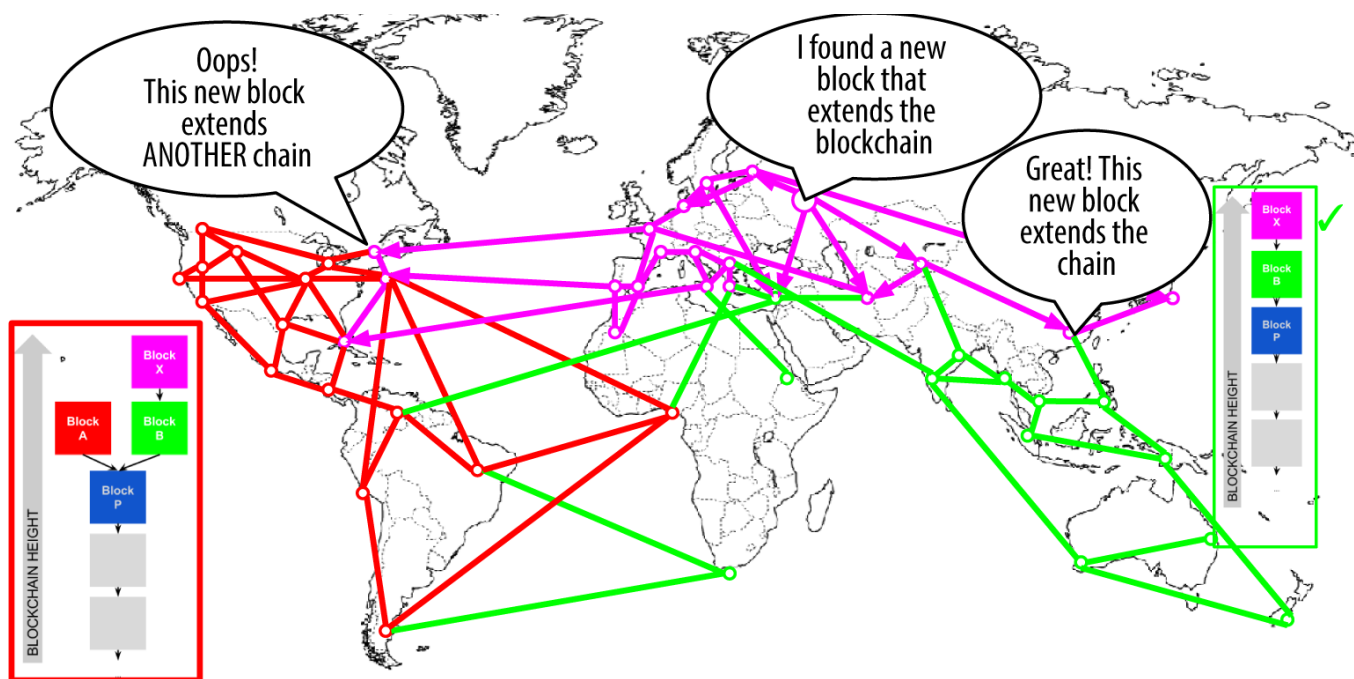


Figure 6. ブロックチェーンのフォークが生じる過程の可視化 - Bitcoinネットワークが新しい最長ブロックチェーンに再収縮する

もし2つのブロックがほぼ同時にフォークの"両方の端"で見つければ、理論的にはフォークが2ブロック分拡張することは可能です。しかし、これが生じる可能性はとても低いです。1ブロックのフォークは毎週起こりえますが、2ブロックのフォークは極めて稀です。

10分間というBitcoinのブロック間隔は、承認までにかかる時間(トランザクションの確定)とフォークが生じる確率の間の妥協点なのです。ブロック間隔をより短くすればトランザクションをより早く確定できますが、ブロックチェーンのフォークがより頻繁に起こってしまいます。一方、ブロック間隔を長くすればフォークの数は減りますが、トランザクションの確定に時間がかかることになります。

## マイニングとハッシュ化競争

Bitcoinマイニングは極度に競争が激しい業界です。ハッシングパワーはBitcoinが現れてから毎年指数関数的

に増加してきています。ここ数年の成長はテクノロジーの進化を反映しており、2010年、2011年には多くのマイナーがCPUマイニングから、GPUマイニングとフィールドプログラマブルゲートアレイ(FPGA)マイニングに変えていきました。2013年はASICマイニングが始まり、マイニングパワーの急激な上昇が起きました。ASICマイニングは、マイニング目的に特化するようにSHA256関数を直接シリコンチップに記述する方法です。これを使った最初のチップは、2010年にBitcoinネットワーク全体が出したマイニングパワーよりも多くのマイニングパワーを提供することができました。

以下のリストは最初の5年間におけるBitcoinネットワークの総ハッシングパワーの推移を表しています。

2009

0.5 MH/sec–8 MH/sec (16倍; 成長)

2010

8 MH/sec–116 GH/sec (14,500倍; 成長)

2011

16 GH/sec–9 TH/sec (562倍; 成長)

2012

9 TH/sec–23 TH/sec (2.5倍; 成長)

2013

23 TH/sec–10 PH/sec (450倍; 成長)

2014

10 PH/sec–150 PH/sec in August (15倍; 成長)

総ハッシングパワー(GHash/秒、過去2年間)のチャートにある通り、Bitcoinネットワークのハッシングパワーは過去2年間で増加しています。見てわかるように、マイナーとBitcoinの成長の間の競争によってハッシングパワー(Bitcoinネットワーク全体の秒間総生成ハッシュ数)が指数関数的に増加してきています。

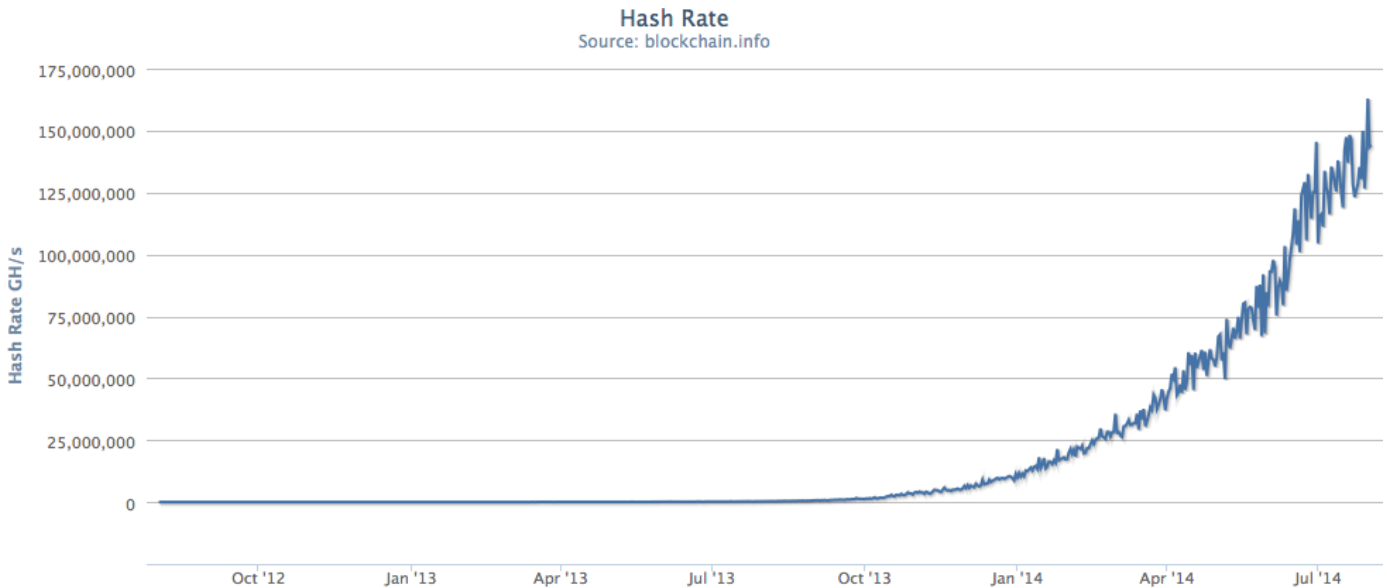


Figure 7. 総ハッシングパワー(GHash/秒、過去2年間)

マイニングに注ぎ込まれるハッシングパワーの量が爆発的に増えてきたため、difficultyもそれに合わせて上昇してきました。過去2年間のBitcoinのマイニングdifficulty推移に示されているチャートにあるdifficultyの数値は、現在のdifficultyを最小difficulty(最初のブロックのdifficulty)で割った率で計算されています。

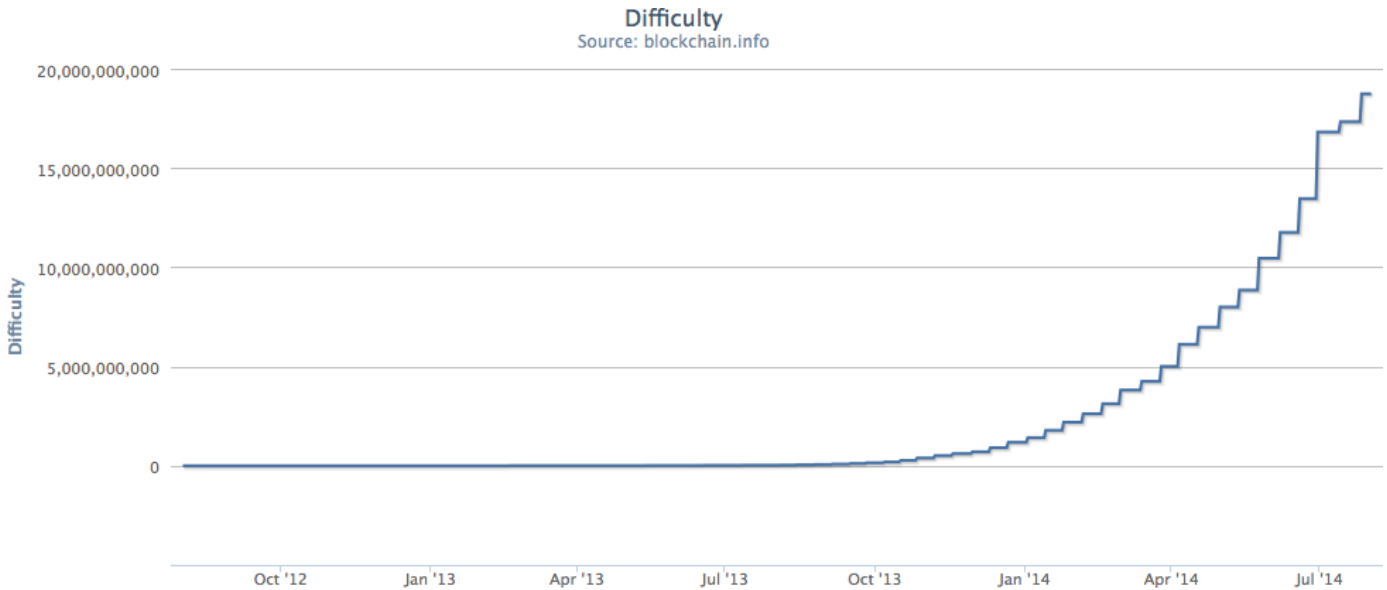


Figure 8. 過去2年間のBitcoinのマイニングdifficulty推移

最近の2年間はASICマイニングチップがより高密度になってきており、シリコン製作における22ナノメートル(nm)の加工寸法(分解能)限界値に近づいてきています。現在、ASICメーカーは汎用CPUチップメーカーを追い越そうとしており、16nm加工寸法チップを設計しています。マイニングの収益性が高いため汎用計算機よりも一層強くこの業界を引っ張っているのです。ただBitcoinマイニングに関してさらなる急激な上昇は残されていません。というのは、18ヶ月毎に半導体の集積密度が約2倍になるという

ムーアの法則の先端にまで達してしまっているためです。ただチップではなく数千チップを配置できるより高密度なデータセンターの競争によってまだより高密度な集積の余地があります。このため、まだBitcoinネットワークのマイニングパワーは指数関数的なペースで進化し続けています。もはや1つのチップでどれだけのマイニングができるかではなく、熱をうまく散らして十分なパワーを提供しつついくつかのチップをどれだけデータセンターに詰め込むことができるかの競争になってきています。

## extra nonceによる方法

2012年からBitcoinマイニングはブロックヘッダ構造にある制限を解決しながら発展してきました。Bitcoinの初期、difficulty targetが低くnonceを使って解を得られるまではマイナーはnonceを繰り返し使うことでブロックを発見できました。difficultyが大きくなっていくにつれて、マイナーはブロックを発見することなくnonceの4億通り全てを使ってしまうことが頻繁に起きるようになっていました。しかし、これはマイニング経過時間を把握するためのブロックのTime Stampを更新することで簡単に解決されました。このTime Stampはヘッダの一部であるため、Time Stampが変わることでマイナーはnonceの値を繰り返し使い、異なるハッシュ値を得ることができるのです。しかし、一度マイニングハードウェアの処理速度が4GH/秒を超えると、この方法は難しくなってきました。というのは、nonceが1秒以内使い尽くされてしまうからです。ASICマイニングが始まるとハッシュレートはTH/秒を超え、マイニングソフトウェアは有効なブロックを見つけるためにより広いnonceスペースが必要になってきました。Time Stampを少し引き延ばすことはできましたが、Time Stampを将来に移動することはブロックを無効にしてしまうこととなります。ブロックヘッダの中のどこかに"変更"が必要になってきました。これに対する解はcoinbaseトランザクションにextra nonceを入れるというものです。coinbase scriptは2byteから100byteのデータを記録できるため、マイナーはこのスペースをextra nonceとして使い始め、より大きいブロックヘッダの範囲を探索してブロックを見つけることができるようになりました。coinbaseトランザクションはmerkle treeに含められているため、coinbase scriptにあるどんな変更もmerkle rootを変化させることとなります。8byte extra nonceと"標準"の4byte nonceを使って、マイナーはTime Stampを変えることなく 秒間  $2^{96}$  個(8のあとに28個の0が続く数)の探索ができるようになりました。もし将来マイナーがこれら全ての可能性を調べ尽くせるようになれば、Time Stampを修正してマイニングをするようになるでしょう。また、将来のextra nonceスペースの拡張のためcoinbase scriptにはまだスペースが残されています。

## マイニングプール

この激しい競争環境の中で、一人でやっている個人のマイナー(ソロマイナーと呼ばれています)は勝ち目がありません。ブロックを見つけて電気やハードウェアのコストを相殺しようとするの見込みは低すぎて、宝くじを買うようなギャンブルになってしまいます。速い消費者向けASICマイニングでさえ、水力発電所の近くの巨大な倉庫に数万個のチップを積み重ねて作った商用システムには追いつきません。今ではマイナーはマイニングプールを作って協力するようになっており、マイニングプールでは個々のマイナーのハッシングパワーを貯め、報酬を数千人の参加者と分けるということをしています。マイニングプールに参加することで、マイナーは総報酬の小さな一部だけを得ることになりますが、毎日平均的に報酬を得ることができるようになり、不確実性を減らすことができます。

具体的な例を見てみましょう。マイナーが秒間6000ギガハッシュ(GH/s) または6TH/sの総ハッシングレートを持つマイニングハードウェアを購入したとします。2014年の8月時点で、この装置は約10000ドルします。このハードウェアは動作時に3キロワット(kW)の電力を消費し、1日に72kW時、金額にして1日平均7ドルか8ドルかかります。現在のBitcoin difficultyでは、マイナーは平均的に155日毎(5ヶ月)に1回ブロックを1人で採掘できます。もしマイナーがこの時間間隔で1つのブロックを見つけたとすると、25bitcoinの支払い(1bitcoinあたり約600ドル)は一回あたり15000ドルになり、ハードウェアやこの期間に消費した電気代のコスト全体を差し引くと約3000ドルの正味利益が残ります。しかし、5ヶ月間に1ブロックを見つけるかどうかはマイナーの運にかかっています。5ヶ月間に2ブロックを見つけて大きな利益を得るかもしれません。もしかしたら、10ヶ月間ブロックを見つけないことができず経済的なロスを受けてしまうかもしれません。さらに悪いことに、Bitcoinのproof-of-workアルゴリズムの

difficultyはおそらく時間が経つにつれて現在のハッシングパワーの成長率に従って著しく上がっていきだろうと考えられます。このことは、ハードウェアが実質的に時代遅れになる6ヶ月間が経つ前に、さらにパワフルなマイニングハードウェアで置き替えなければならないということを意味します。もしこのマイナーが5ヶ月に1回の棚ぼた的な15000ドルを待っている代わりにマイニングプールに参加していれば、一週間に約500ドルから750ドルを稼ぐことができるでしょう。マイニングプールからの定期的な支払いを使うことで大きなリスクを負うことなくハードウェアや電気代のコストの償却ができます。ハードウェアが6ヶ月から9ヶ月後に時代遅れになるためリスクはまだ高いですが、少なくとも収入はこの期間の間確実に定期的に入ることになるのです。

マイニングプールは特別なプールマイニングプロトコルを通して数十万人ものマイナーを束ねています。個々のマイナーはマイニングプールにアカウントを作成した後、マイニング機器をプールサーバに接続するように設定します。マイニングハードウェアはマイニングの最中このプールサーバに接続されたままになっており、他のマイナーとマイニング結果を同期しています。このため、マイニングプールマイナーはブロックを採掘した結果を共有し、これによって得られた報酬を分配します。

採掘に成功したブロックの報酬はプールサーバのBitcoinアドレスに支払われます。個々のマイナーではありません。報酬の分配総額がある閾値に達したら、プールサーバは繰り返しマイナーのBitcoinアドレスに支払いを行います。通常的に、プールマイニングサービスを提供するためプールサーバは報酬の一定パーセントを手数料として徴収しています。

マイニングプールに参加しているマイナーたちは候補ブロックに対する解を探す仕事を分割し、マイニングに対する寄与によって"分配"を稼ぎます。マイニングプールは分配を稼ぐためにより低いdifficulty targetを設定します。典型的に、Bitcoinネットワークのdifficultyの1000分の1以下のdifficultyになっています。マイニングプールの誰かがブロックを採掘するとまずこの報酬はプールによって受け取られ、寄与した仕事量に比例した分の分配が全てのマイナーに配られます。

マイニングプールは全てのマイナーに対して公開されています。大きい、小さい、プロ、アマチュアは関係ありません。このため、マイニングプールには単一の小さなマイニングマシンを持った参加者もいれば、ハイエンドマイニングハードウェアをガレージにいっぱい入れてマイニングをしている参加者もいます。一部の参加者は数十KWの電気代を使ってマイニングをしており、また別の参加者は1メガワットを消費してデータセンターを運用している参加者もいます。どのようにしてマイニングプールは個々の寄与を測定し、いかさまができないようにしながら平等に報酬を配っているのでしょうか？答えは、プールマイナーの個々の寄与を測るためにBitcoinのproof-of-

workアルゴリズムを使うことです。ただし、最も小さいプールマイナーでさえも頻繁に分配を受けられ、やりがいを感じられるようにより低いdifficultyに設定しておきます。低いdifficultyは分配金を稼ぐやすくするためのものですが、マイニングプールはこの低いdifficultyを使ってそれぞれのマイナーによって完了した仕事の量を測定します。マイニングプールが設定したdifficultyよりも低いdifficultyのブロックヘッダハッシュを見つけるたびに、プールマイナーはハッシュ化作業を行ったことを証明することになるのです。さらに重要なこととして、ネットワーク全体のdifficulty

targetよりも低いハッシュを見つけるマイニングプール全体の努力に、この分配金を稼ぐ作業が統計的に測定可能な形で貢献することになります。低いdifficultyのハッシュを見つけようとしている数千のマイナーが偶然Bitcoinネットワークのdifficulty targetを満たすハッシュを見つけることになるのです。

前に書いたサイコロゲームとの類似性に戻ってみましょう。もし4よりも小さい値(Bitcoinネットワーク全体のdifficulty)をサイコロを投げて出そうとするなら、プールはより簡単なtargetを設定し、何回プールプレイヤーが8よりも小さい値をサイコロを投げて出したかをカウントします。プールプレイヤーが8よりも小さい値(マイニングプールでの共有difficulty)を出したとき、プールプレイヤーは分配量を得ますがゲームには勝っていません。なぜなら(4より小さい値を出すという)ゲームの水準に達していないからです。ゲームに勝てる水準のdifficulty targetに達しなかったとしても、プールプレイヤーはより簡単なdifficulty



targetをより頻繁に満たすことで、定期的に分配量が彼らに割り振られるようにします。ときどきプールプレイヤーのうちの一人が二つのサイコロの目を足して4より小さい値にした場合、このプールが勝ちます。このとき、このときの報酬はプールプレイヤーが得た分配量に基づいてプールプレイヤーに分配されます。8かそれより小さい値を出すという水準がゲームに勝つようなものではなかったとしても、これはプールプレイヤーがサイコロを振ったということを知る公平な方法であり、時折4よりも小さい値を出すことがあるのです。

同様に、マイニングプールは個々のプールマイナーがプールのdifficultyよりも低いdifficultyのブロックヘッダハッシュをとる頻繁に発見できるようにプールのdifficultyを設定します。ときどきこれらの試行のうちの一つがBitcoinネットワークでのdifficulty

targetよりも低いブロックヘッダハッシュを作り出し、有効なブロックを作りプール全体が勝つこととなります。

## マネージドプール

ほとんどのマイニングプールは"管理された"もので、プールサーバを動かしている会社か個人がいます。このプールサーバの所有者はプールオペレータと呼ばれており、プールマイナーの稼ぎのうちの一定パーセントを手数料としプールマイナーに課しています。

プールサーバでは、特別なソフトウェアやプールマイナーの活動を調整するプールマイニングプロトコルを動作させています。プールサーバはまた、一つまたは複数のフルBitcoinノードと接続を張り、ブロックチェーンデータベースの完全なコピーに直接アクセスできるようになっています。これによって、プールサーバはプールマイナーのためにブロックやトランザクションの検証をすることができ、プールマイナーがフルノードを動かす負荷を軽減しています。プールマイナーにとって、これは重要なことです。なぜなら、フルノードには少なくとも15GBから20GBの永続的なストレージ(ディスク)と2GBのメモリ(RAM)を持っている専用コンピュータが必要になるからです。さらに、フルノードで動作しているBitcoinソフトウェアを監視し、メンテナンスし、頻繁にアップグレードをする必要があります。メンテナンスの欠如、またはリソースの欠如によって生じたどんなダウンタイムもマイナーの利益を減らしてしまいます。多くのマイナーにとって、フルノードを動作させることなく採掘ができるということは、マネージドプールに参加するもう一つの大きな利点なのです。

プールマイナーは Stratum (STM) や GetBlockTemplate (GBT) のようなマイニングプロトコルを使ってプールサーバに接続しています。少し前の標準的なプロトコルであった GetWork (GWK) は2012年の終わりからほぼ時代遅れになっています。というのは、4GH/sよりも大きいハッシュレートでのマイニングをサポートしていないからです。STMもGBTも候補ブロックヘッダのテンプレートを含む ブロック テンプレートを 作り出します。プールサーバはトランザクションを集めることで候補ブロックを構築し、coinbaseトランザクション(extra nonceスペースを含む)を追加し、merkle rootを計算し、前のブロックハッシュに連結します。候補ブロックのヘッダはこのときテンプレートとしてプールマイナーそれぞれに送られます。それぞれのプールマイナーはブロックテンプレートを使ってBitcoinネットワークのdifficultyよりも低いdifficultyで採掘をし、どんな成功した結果もプールサーバに送り返し分配量を稼ぐこととなります。

## P2Pool

マネージドプールではプールオペレータによってイカサマをされる可能性があります。プールオペレータはプールに対する労力を二重使用トランザクションやブロックの無効化(コンセンサス攻撃参照)に仕向けるかもしれません。さらに、中央化されたプールサーバが単一障害点になることがあります。もしDOS攻撃でプールサーバがダウンしたり遅延したりした場合、プールマイナーは採掘ができません。2011年に、これらの中央化

の問題点を解決するために、新しいプールマイニング方法が提案され実装されました。P2Poolはpeer-to-peerのマイニングプールで、中心的なオペレータがいません。

P2Poolはプールサーバの機能を分散化することで動作し、[シェアチェーン](#)と呼ばれるブロックチェーンのような並列システムで実装されています。シェアチェーンはBitcoinのブロックチェーンよりも低いdifficultyで動作しているブロックチェーンです。シェアチェーンによってプールマイナーは分散化されたプール内で協力することができるようになり、30秒毎に1シェアブロックの割合でシェアチェーン上の割り当て分を採掘します。シェアチェーン上のそれぞれのブロックは仕事に寄与したプールマイナーに対して仕事量に比例する形で割り当てた報酬を記録し、前のシェアブロックから先頭の方にシェアを運んでいきます。シェアブロックのうち1つでもBitcoinネットワークのdifficulty targetに達するものがあれば、それが伝搬されBitcoinのブロックチェーン上に埋め込まれ、勝ったシェアブロックを率いた全てのシェアに貢献したプールマイナーに報酬が与えられます。本質的には、プールマイナーのシェアと報酬を記録しているプールサーバの代わりに、シェアチェーンが全てのプールマイナーが全てのシェアを追跡できるようにしており、この追跡にBitcoinのブロックチェーンコンセンサスメカニズムのような分散されたコンセンサスメカニズムが使われています。

P2Poolマイニングはプールマイニングよりも複雑です。なぜなら、フルBitcoinノードとP2Poolノードソフトウェアをサポートするための十分なディスクスペース、メモリ、インターネット帯域を持った専用コンピュータをプールマイナーが動作させる必要があるためです。P2Poolマイナーは自身のマイニングハードウェアをローカルのP2Poolノードに接続し、このローカルP2Poolがマイニングハードウェアにブロックテンプレートを送るプールサーバの機能を真似ることになります。P2Pool上では、個々のプールマイナーが自身で候補ブロックを構築しソロマイナーのようにトランザクションを集めますが、このときシェアチェーン上で共同で採掘をします。P2Poolはソロマイニングより粒子が小さい支払いができるという有利な点がありつつ、マネージドプールのようなプールオペレータにとっても大きなコントロールを与えることがないというハイブリッドなアプローチになっています。

最近、マイニングプールへのマイニング集中が [51%攻撃](#) ([コンセンサス攻撃](#)参照)への懸念を引き起こすレベルにまでなっており、P2Poolへの参加が著しく増えてきています。さらなるP2Poolプロトコルの開発によってフルノードを走らせる必要性がなくなることが期待され続けており、結果的に分散化されたマイニングがさらに使いやすくなるでしょう。

P2Poolがマイニングプールオペレータによるパワーの集中を削減することはありますが、おそらくシェアチェーンそのものに対する51%攻撃の脆弱性はありえます。P2Poolがとて広く採用されてもBitcoinそのものに対する51%攻撃の解決はしないのです。むしろ、マイニングエコシステムを多様化させる一部分としてP2PoolはBitcoinを全体的により堅牢にすることになります。

## コンセンサス攻撃

Bitcoinのコンセンサスメカニズムは、少なくとも理論的には、ハッシングパワーを使って不正なまたは破壊的な方向に持って行こうとするマイナー(またはマイニングプール)による攻撃に対して脆弱です。今まで見たように、コンセンサスメカニズムは自己の興味に対して正直に行動するマイナーが大多数いるということに依存しています。しかし、もしマイナーやマイナーの集団がマイニングパワーの十分なシェアを取り得たとすると、彼らはBitcoinネットワークのセキュリティや有用性を破壊するようにコンセンサスメカニズムを攻撃できるのです。

コンセンサス攻撃は将来の合意形成に影響を与えることができるだけで、過去に対してはせいぜい少し過去(10ブロック前)に影響を与えられるくらいです。このことはとても重要なことなのです。Bitcoinの元帳は時間が過ぎれば過ぎるほど、どんどん不変になっていきます。理論上フォークしたブロックチェーンはどんな深さにでも達することができますが、実際にはとても深いフォークを作るには古いブロックを変更できないように

しておく必要があるため莫大な計算量が必要です。コンセンサス攻撃はまた秘密鍵や署名アルゴリズム(ECDSA)のセキュリティに全く影響を与えません。コンセンサス攻撃はbitcoinを盗むことも、署名なしにbitcoinを使うことも、bitcoinの支払先を書き換えることも、過去のトランザクションや記録の所有者を変えることもできません。コンセンサス攻撃は単に直近のブロックに影響を与え、将来のブロック生成に対してDOS攻撃による破壊を引き起こすだけなのです。

コンセンサスメカニズムに対する一つの攻撃シナリオは"51%攻撃"と呼ばれています。このシナリオでは、全Bitcoinネットワークのハッシングパワーの大多数(51%)をコントロールしているマイナーのグループが共謀してBitcoinへの攻撃をするというものです。ブロックの大部分を採掘する能力を持つことで、攻撃マイナーはブロックチェーンに故意の"フォーク"を作り出し、トランザクションを二重に使用したり、DOS攻撃を特定のトランザクションまたはアドレスに対して実行したりできます。フォーク/ダブルスペンド(二重使用)攻撃では、攻撃者が事前にある承認済みブロックよりも下からフォークすることでこの承認済みブロックを無効化し、攻撃者が作った代替りのチェーンにブロックチェーンを再収縮させます。十分なハッシングパワーを持っていれば、攻撃者は6つまたはそれ以上のブロックを無効化でき、変更不可能だと考えられている(6回の承認が行われた)トランザクションを無効化できるのです。ダブルスペンドが攻撃者自身のトランザクション上で実施できてしまうということはとても重要です。このために、攻撃者は有効な署名を作り出すことができずしてしまいます。もしトランザクションを無効化することで、攻撃者が支払いをすることなく両替や商品の取得ができるなら、このダブルスペンドは有益なものになるのです。

51%攻撃を具体的な例で説明してみましょう。第1章で、一杯のコーヒー代の支払いに使われたAliceとBobの間のトランザクションをみました。カフェのオーナーであるBobは承認(ブロックの採掘)を待つことなくコーヒー代を喜んで受け入れています。なぜなら、コーヒー代のダブルスペンドのリスクは、素早い顧客サービスを提供することの利便性と比べると低いからです。これは25ドル以下の支払いに対して署名なくクレジットカードの支払いを受け付けるコーヒーショップと同様で、署名のために生じる取引の遅延コストの方がクレジットカードの請求取り消しのリスクより比較的大きいからです。反対に、bitcoinでもっと高額な商品を買う場合はダブルスペンド攻撃の大きなリスクがあります。購入者は競合するトランザクション(販売者への支払いに使ったトランザクションインプット(UTXO)を使って、販売者への支払いをキャンセルするトランザクション)をブロードキャストします。ダブルスペンド攻撃は二つの場合に生じ得ます。一つは、トランザクションが承認される前、もう一つはもし攻撃者がいくつかのブロックを元に戻せるような優位性を持っている場合です。51%攻撃によって、攻撃者は自身で新しく作ったブロックチェーン上で自身のトランザクションをダブルスペンドをすることができるようになるため、古いブロックチェーン上にある販売者への支払いトランザクションを元に戻し、販売者への支払いをなかったことにできるのです。

例として、悪意ある攻撃者MalloryがCarolの画廊に行き、プロメテウスとしてSatoshi Nakamotoを描いた美しい三連祭壇画を購入することを考えてみましょう。Carolはこの"The Great Fire"の絵画を250,000ドルでMalloryにbitcoinで売りました。トランザクションの6回またはそれ以上の承認を待たずに、Carolはたった1回の承認後に絵画をラッピングしてMalloryに手渡しました。Malloryは共犯者Paulとともに共謀しており、Paulは巨大なマイニングプールを運用しています。この共犯者PaulはMalloryのトランザクションがブロックに取り込まれるとすぐに51%攻撃を実行しました。Paulはマイニングプールを操ってMalloryのトランザクションを含んでいるブロックと同じブロック高を再採掘し、MalloryからCarolへの支払いトランザクションを、Malloryが支払いに使ったインプットと同じインプットをダブルスペンドするトランザクションで置き換えます。このダブルスペンドトランザクションは同じUTXOを消費し、Carolへの支払いの代わりにMalloryのウォレットに支払い戻すようにし、本質的にMalloryがbitcoinを使う前の状態のままにしておくのです。このときPaulはマイニングプールを操りもう一つのブロックをマイニングし、元々のブロックチェーンよりも長いダブルスペンドトランザクションを含んだブロックチェーンを作ります(MalloryからCarolへの支払いトランザクションが含まれたブロックより下のブロックが同じようなフォークを作り出します)。新しい(元々のブロックチェーンより長い)ブロックチェーンが選ばれることでブロック

チェーンのフォークが解消されると、ダブルスペンドトランザクションはCarolへの元々の支払いトランザクションを置き換えることとなります。Carolは三つの絵画を失い、しかもbitcoinが支払われていないのです。この全ての行動に関して、Paulのマイニングプールに参加している参加者は幸せなことにダブルスペンドトランザクションが行われたことに気づかないままにいるかもしれません。というのは、彼らは自動化されたマイナーでマイニングを行っており、全てのトランザクションまたはブロックを追跡することはできないからです。

この種類の攻撃を防ぐために、大きな金額の商品を売る商人は、バイヤーに商品を渡す前に少なくとも6回の商人を待たなければいけません。または、エスクローマルチシグアカウントを使い、このエスクローアカウントに入金されたあと数回の承認を再び待たなければいけません。承認数を増やせば増やすほど、51%攻撃でトランザクションを無効化することが難しくなっていきます。バイヤーが24時間待たなければいけないとしても、高い商品に対しては、bitcoinでの支払いが便利であり効率的です。24時間は約144回承認に相当します。

ダブルスペンド以外のコンセンサス攻撃のシナリオは、特定のBitcoin参加者(特定のBitcoinアドレス)に対するサービスを拒否するようになってしまうことです。マイニングパワーの大多数を占める攻撃者は単に特定のトランザクションを無視することができます。もし他のマイナーによって採掘されたブロックがこれらのトランザクションを含んでいた場合、攻撃者はわざとフォークをしてこのブロックを再採掘することができ、再び特定のトランザクションを除外することができるのです。攻撃者がマイニングパワーの大多数をコントロールできる限り、このタイプの攻撃によって特定のBitcoinアドレスまたはBitcoinアドレスの集合に対して持続的DOS攻撃を引き起こすことができます。

この名前にも関わらず、51%攻撃シナリオは実際にハッシングパワーの51%が必要というわけではありません。事実、このような攻撃はハッシングパワーの51%より小さい割合でも起こすことができます。51%という閾値は、単にこのくらいの割合にならないとそのような攻撃がほとんど成功しないという意味です。コンセンサス攻撃は本質的に次のブロックに対する主導権争いであり、"より強い"グループがより勝ちやすいのです。ハッシングパワーがより少なければ成功確率はさがります。というのは、他のマイナーが"信用している"他のハッシングパワーによって同じブロックの生成がコントロールされるからです。もう一つの側面として、より多くのハッシングパワーを攻撃者が持っていれば持っているほど、攻撃者はわざとより長いフォークを作ることができます。このため、攻撃者が無効化できる直近のブロック数、または攻撃者がコントロールできる将来のブロック数も多くなります。セキュリティ研究グループは、統計学的モデリングを使って30%程度のハッシングパワーの占有率でいろいろなタイプのコンセンサス攻撃が可能になるということを主張しています。

総ハッシングパワーの大幅な増加によって、Bitcoinに対する単一マイナーによる攻撃はほとんど実行しにくくなっています。ソロマイナーが総マイニングパワーの大多数をコントロールすることは不可能なのです。しかし、マイニングプールによるハッシングパワーの中央コントロールによって、マイニングプールオペレータによる営利目的攻撃を引き起こすリスクが生じてきています。マネージドプールのマイニングプールオペレータは候補ブロックの構築をコントロールし、またどのトランザクションをブロックに含めるかをもコントロールします。これによって、トランザクションを除外するまたはダブルスペンドトランザクションを含められるパワーをマイニングプールオペレータに与えることになるのです。もしハッシングパワーを制限された形または気付かれないような微妙な形で悪用したとすると、おそらく気づかれることなくマイニングプールオペレータはコンセンサス攻撃から利益を上げることができるでしょう。

しかし、全ての攻撃者が利益に動機づけられているわけではありません。一つのありえる攻撃シナリオとして、攻撃者がBitcoinネットワークを破壊するつもりで攻撃を行うこともあります。このような破壊から利益を上げられる可能性がなかったとしても、Bitcoinに大きな損害を与えることを目指している悪意ある攻撃には莫大な投資や密かな計画が必要です。あるとしたら、おそらく州が支援しているような資金が十分にある攻撃者によって開始されるはずですが。あるいは、資金が十分にある攻撃者であれば、マイニングハードウェアを大量に集め、マイニングプールオペレータに歩み寄って他のマイニングプールに対してDOS攻撃を仕掛けることでBitcoinのコンセンサスメカニズムを攻撃するはずですが。これらのシナリオは全て理論的には可能ですが、B

Bitcoinネットワークの全体的なハッシングパワーが指数関数的に成長し続けているためだんだんと非現実的になっています。

確かに、深刻なコンセンサス攻撃は短期間にBitcoinに対する信用を腐食し、もしかすると深刻な価格衰退を招くかもしれません。しかし、Bitcoinネットワークとソフトウェアは一定速度で発展しており、コンセンサス攻撃に対してすぐにBitcoinコミュニティによって対応策が取られ、Bitcoinは今までより、より強力に、より匿名性が高く、より頑強になっていくことでしょう。

# Alt chain、通貨、<phrase role="keep-together">、アプリケーション</phrase>

Bitcoinは20年間にわたる分散型システムと通貨の研究の結果であり、Bitcoinはこれらの分野にproof of workに基づく分散型のコンセンサスメカニズムという新しい技術革新をもたらしました。この発明はBitcoinのコアとなる発明であり、通貨、金融サービス、経済学、分散型システム、投票システム、コーポレートガバナンス、そして契約の分野において、革命の波をもたらしてきたのです。

この章では、Bitcoinとブロックチェーンの発明から生まれた多くの派生物を見ていきます。これらの派生物とは、2009年にこのテクノロジーの革新が始まってから構築されたAlt chain(オルトチェーン、代替チェーン)、通貨、アプリケーションのことで、大部分は通貨である代替コイン、または Alt coin(オルトコイン、代替通貨) と呼ばれるコインであり、これらのコインはBitcoinと同じ設計原則で実装されていますが、異なるブロックチェーンとネットワークで運用されるコインとなっています。

Alt coin製作者やファンへの配慮のためにあらかじめお断りしておきますが、この章で紹介されるAlt coinの中には名前が挙げられていないものが50個以上あります。この章の目的はAlt coinの質を評価することではありません。また最も重要なものを主観でお話することでもありません。ここでは、いくつかの例を出しBitcoinエコシステムの幅広さと種類の豊富さを紹介し、今までに類を見ないそれぞれのイノベーションや重大な違いを紹介してつもりでいます。実際いくつかのAlt coinの例の中にはお金として考えるには大きな欠点を持っているものもあります。これらは研究という視点で考えるととても興味深いAlt coinになっています。ただし、この章は投資にあたってのガイドを意図していないことにご注意ください。

新しいコインは毎日新しく生み出されているため、重要なコイン、それも歴史を変えるかもしれないコインを全て把握していくことはできません。イノベーションの速度は極めて速く、この章はすぐに時代遅れなものになってしまうでしょう。

## Alt coinとAlt chainの分類

Bitcoinはオープンソースのプロジェクトで、そのコードは他の多くのソフトウェアプロジェクトのベースとして使われています。Bitcoinのソースコードから生まれたソフトウェアで最も一般的なものは分散型の Alt coin です。Alt coinはデジタル通貨の実装にBitcoinと同じブロックを構築する手段を使っています。

Bitcoinのブロックチェーン上にはいくつかのプロトコルのレイヤーがあります。これらレイヤーであるメタコイン、メタチェーン、ブロックチェーンアプリケーションは、ブロックチェーンをアプリケーションのプラットフォームとして拡張する、もしくはBitcoinのプロトコルに別のプロトコルレイヤーを加えて拡張することで実現しています。例としては、Colored coin、Mastercoin、NXT、Counterpartyがあります。

次の節ではいくつかの特徴的なAlt coinを調べていきましょう。例えば、LitecoinやDogecoin、Freicoin、Primecoin、Peercoin、Darkcoin、Zerocoinです。これらのAlt coinは歴史的な理由で特筆する価値があるのです。というのは、最も価値がある、また"最良"のAlt coinであるがためというわけではなく、特定の分野のAlt coinのイノベーションの例としてこれらがよい例であるためです。

Alt coinに加えて、"コイン"ではないいくつかのブロックチェーンの別実装もあります。これらは私が *Alt chain* と呼ぶものです。これらのAlt chainは、契約のためのプラットフォーム、名前登録、もしくは他のアプリケーションとしてコンセンサスの仕組みと分散型台帳を実装しています。Alt chainはBitcoinと同じブロックの構築方法を用いて、通貨もしくはトークンをその支払いに使用しています。ですが、主な目的は通貨ではありません。のちほどAlt chainの例としてNamecoinとEthereumを見ていくことにしましょう。

最後に、Bitcoin以外にもデジタル通貨もしくはデジタル決済システムのネットワークを提供している競合があります。競合にはRippleなどがあり、proof of workに基づく分散型台帳やコンセンサスのメカニズムを使うことなしに動いているものです。これらのブロックチェーンのテクノロジーでないものは、この本が扱う範囲外であるため、この章では扱わないことにします。

### ===メタコインのプラットフォーム

メタコインとメタチェーンはBitcoin上に建てられたソフトウェアのレイヤーであり、通貨内通貨、もしくはBitcoinシステムの内側のプラットフォーム、プロトコルのオーバーレイとして実装されています。これらの機能のレイヤーはBitcoinのコアの機能を拡張し、Bitcoinトランザクションとアドレス内にメタコインなどに必要な追加的なデータ(メタデータ)を記録することによって機能と特性を加えています。メタコインを最初に実装する際に使われたメタデータをBitcoinのブロックチェーンに載せる方法は、例えばBitcoinのアドレスを使ってデータをエンコードすることや、使われていないトランザクションのフィールド(例えばトランザクションのSequenceフィールド)を使うような方法です。トランザクションのScript opcodeに OP\_RETURN が導入されてからは、ブロックチェーンにより直接的にメタデータを書き込むことができるようになりました。多くのメタコインはこのOP\_RETURNによる方法に移行していきようとしています。

### ====Colored Coin

*Colored coin*(カラードコイン) は少額のbitcoin上にメタ情報を積み重ねるメタプロトコルです。"色がついた(colored)"コインというのは、別のアセットを表現するために転用された少額のbitcoinなのです。例えば1米ドル札を取り出して、そこに"これはAcme Incの1株の証明です"という意味のスタンプを押したとを考えてみてください。今この1米ドル札は2つの意味を持っています：一つは通貨としての意味、そして、もう一つは株式の証明としての意味です。株式としての価値の方が大きいので、誰もこの株式をキャンディを買うために使おうとはしないでしょう。つまり、もはや事実上通貨として使用していないのです。Colored coinはこれと同じやり方を行っています。特定のとても少額のbitcoinを他のアセットを表しているトレード可能な証明書に変換しているのです。"Color"の言葉が示しているのは、色のように属性を追加することで特別な意味を与えるという意味です。色はメタファーであり、実際に色とは関連はありません。実際にはColored coinに色は無いのです。

Colored coinは特別なウォレットによって管理されています。このウォレットは色付きbitcoinにメタデータを付加したり解釈したりできるウォレットです。そのようなウォレットを使い、ユーザーは特別な意味を持つラベルをbitcoinに追加して、色がついていないbitcoinから色がついているbitcoinへ変換します。例えば、株式の証明、クーポン、不動産、日用品、収集可能なトークンを表すラベルです。特定のコインに付けた"色"にどんな意味を加えて解釈するかは完全にユーザーに委ねられています。色をつけるためにユーザは、発行タイプ、より小さな単位に切り分けられるかどうか、シンボル、説明、その他の関連した情報といったメタ情報を定義することになります。一旦色が付くと、これらのコインは購入、販売、分割、収集ができるようになり、そして配当の支払いを受け取る事ができるようになります。色付きコインから紐付けられた情報を"取り除き"、bitcoinとしての元々の価値を取り戻すこともできます。

Colored coinを使ってみるために、"MasterBTC"のシンボルを持つ20個のcolored coinを作成しておきました。これは[\[example\\_9-1\]](#)にある通りこの本の無料コピーのクーポンを表しています。このcolored coinを表現しているMasterBTCのそれぞれのクーポンは、colored coinを使えるウォレットを持っていればあらゆるbitcoinユーザーに対して販売したり与えたりすることができ、MasterBTCを得たユーザーはこれをさらに別の人に送ったり、またはMasterBTCを本の無料コピーに使い発行者に戻すこともできます。このcolored coinの例は、[here](#)で見ることができます。

このcolored coinのメタデータは、この本の無料コピークーポンという意味を持っています。

```
{
  "source_addresses": [
    "3NpZmvSPLmN2cVFW1pY7gxEAVPCVfnWfVD"
  ],
  "contract_url":
  "https://www.coinprism.info/asset/3NpZmvSPLmN2cVFW1pY7gxEAVPCVfnWfVD",
  "name_short": "MasterBTC",
  "name": "Free copy of \"Mastering Bitcoin\"",
  "issuer": "Andreas M. Antonopoulos",
  "description": "This token is redeemable for a free copy of the book \"Mastering Bitcoin\"",
  "description_mime": "text/x-markdown; charset=UTF-8",
  "type": "Other",
  "divisibility": 0,
  "link_to_website": false,
  "icon_url": null,
  "image_url": null,
  "version": "1.0"
}
```

## Mastercoin

Mastercoin(マスターコイン)はBitcoin上のプロトコルレイヤーであり、Bitcoinシステムを拡張する様々なアプリケーションのためのプラットフォームとなっています。MastercoinはMSTという通貨をトランザクションを行うためのトークンとして使用しています。しかし、それは通貨というよりもむしろ、ユーザー通貨やスマートプロパティのトークン、分散型のアセット取引所などを構築するためのプラットフォームです。MastercoinをBitcoinトランザクションを送るトランスポートレイヤー上にあるアプリケーションレイヤーとして考えると、それはHTTPがTCP層の上を動いているようなものなのです。

Mastercoinは主に特別なBitcoinアドレスを使ったトランザクションの送受信を通して動作しており、この特別なBitcoinアドレスは "exodus" アドレス (1EXoDusjGwvnjZUyKkxZ4UHEf77z6A5S4P)と呼ばれています。これはHTTPが他のTCPトラフィックからHTTPのトラフィックを区別するために特別なTCPポート(ポート80番)を使って動作しているようなものです。Mastercoinのプロトコルは、特別なExodusアドレスとマルチシグネチャを使う方法から、トランザクション



のメタデータをエンコードするためにOP\_RETURNを使うプロトコルへとだんだん移行してきています。

====Counterparty

Counterparty(カウンターパーティー)は、Bitcoin上にあるMastercoinとは別のプロトコルレイヤーとして実装されています。Counterpartyは、ユーザー通貨や交換可能なトークン、ファイナンシャルツール、分散型アセット取引所などを可能にしています。CounterpartyはBitcoinのScript言語にある OP\_RETURN オペレーターを主に使って実装されており、このオペレーターはBitcoinのトランザクションに追加の意味を持たせるメタデータを記録するために使われています。Counterpartyは、Counterpartyのトランザクションを実行するためにXCPをトークンとして用いています。

## Alt Coin

大半のAlt coinはBitcoinのソースコードをベースにしており、この方法は"フォーク"とも呼ばれています。他にはBitcoinのソースコードを使わずに、ブロックチェーンのモデルに基いて"スクラッチから"実装されているものもあります。Alt coinとAlt chain(次の節で説明します)は、どちらも別々のブロックチェーンテクノロジーの実装であり、どちらも自身のブロックチェーンを使っています。Alt coinとAlt chainの定義の違いは、Alt coinが通貨として主に使われている一方で、Alt chainは通貨を主目的としない他の目的のために使われていることにあります。

厳密にいうと、Bitcoinのソースコードの一番最初のメジャーな"代替"フォークはAlt coinではなくAlt chainである *Namecoin* です。これについては次の節で説明します。

発表された日付からすると、Bitcoinのフォークとしての最初のAlt coinは 2011年8月に登場した *IXCoin* です。IXCoinはいくつかのBitcoinのパラメーターを修正したもので、取り分け1ブロックごとに96コインずつ報酬を加えていくという方法で通貨の生成を加速させようとしていました。

2011年9月には *Tenebrix* がローンチされました。Tenebrixはproof of workアルゴリズムの代替手段を初めて実装した最初の暗号通貨です。この代替アルゴリズムは `_scrypt_` であり、元々(ブルートフォースアタック対策としての)パスワードストレッチングのために作られたアルゴリズムです。Tenebrixが目指すゴールはメモリを多く使うアルゴリズムを用いることによってGPUやASICによるマイニングがしにくくなるコインを作ることでした。Tenebrixは通貨としては成功しませんでした。Litecoinの基礎となり何百ものクローンを生み出すことになったのです。

*Litecoin* はscryptをproof-of-workアルゴリズムとして使うことに加えてブロックの生成速度を速くする実装を行っており、Bitcoinの10分毎の代わりに2.5分毎にブロックが生成されるようになっています。litecoinは"bitcoinが金ならばlitecoinは銀"と謳われており、より軽量な代替通貨としての意味を持っています。より速い承認時間とより多い840万1 *itecoin* という総発行量があるため、小売業のトランザクションに適しているというのがLitecoinの大きな魅力となっています。

Alt coinは2011年、2012年の間にもどんどん増えていきました。これらはBitcoinまたはLitecoinを基礎とする仕組みを持っているもので、2013年までには20個ものAlt coinが現れ競争がどんどん激しくなっていました。2013年の終わりには200個に達し、2013年は"Alt coinの年"と呼ばれるようになりました。Alt coinの成長は2014年まで続き、執筆している時点で500個以上ものAlt coinが存在する状態になっています。半数以上のAlt coinはLitecoinのクローンです。

Alt coinを作り出すことは自体は簡単であるため、500個以上ものAlt coinが存在しています。大半のAlt coinはBitcoinとわずかだけしか違いがなく学ぶ価値がないものが多いです。つまり、単に製作者が儲けようとしているだけのものが実際には多いのです。しかし、人真似や虚偽の情報を流して人を騙そうとしているコインの中には注目すべき例外もあり、非常に重要なイノベーションとなっているものがあります。それらのAlt coinは急進的でBitcoinとは異なるアプローチを取っており、重要なイノベーションをBitcoinの設計原則に加えています。これらのAlt coinがBitcoinと異なっている部分には主に3つの分野があります。

- 異なる通貨ポリシー
- 異なるproof of workもしくはコンセンサスアルゴリズム
- 強力な匿名性のような特徴

詳細な情報は [graphical timeline of alt coins and alt chains](#) に記載してあるので見てみてください。

## Alt coinの価値評価

たくさんのAlt coinが生まれてくる中で、どのようにしてどれが注目すべきものかを決めればよいのでしょうか？いくつかのAlt coinは広く普及させることを目的としており、通貨として使うことを念頭に置いています。他には、異なる機能や通貨モデルを試すための試金石として作られているものもあります。ただ多くのAlt coinは、ただ製作者がすぐに儲けようとしているだけのものです。Alt coinの価値を評価するために私はそれらのAlt coinが持っている特徴と市場指標を見ることにしています。

ここに、あるAlt coinはどんな点でBitcoinと異なっているのか？ということの評価するための質問を挙げてみます。

- Alt coinは大きなイノベーションをもたらしているのか
- ユーザーがBitcoinを離れてAlt coinに引きつけられるだけの競争力のある違いを持っているのか？
- そのAlt coinはニッチなマーケットやアプリケーションに対して魅力を訴えているのか？
- そのAlt coinはコンセンサス攻撃に対する安全性を持てるほどマイナーにとって魅力があるのか？

ここにキーとなる市場指標を挙げてみます。

- Alt coinの時価総額はいくらか？
- どれだけのユーザーとウォレットを持っていると見積もられているか？
- どれだけの企業がAlt coinを受け入れているか？
- 毎日どれだけのトランザクション(出来高)がAlt coinで行われているか？
- どれだけの価値が毎日取引されているか？

この章では、特に技術的な特徴と先の質問によって代表されるイノベーションの潜在価値について説明していきます。

## 通貨パラメータ代替案: Litecoin、Dogecoin、Freicoin

Bitcoinは発行量が固定されているデフレ通貨の特徴を2、3個持っています。まず通貨総発行量が2100万bitcoin(または2100兆satoshi)に制限されていること。そして一定比率で発行レートが減少すること。そしてトランザクションの承認速度と通貨生成をコントロールする10分毎のブロック”ハートビート(鼓動)”があること。

多くのAlt

coinは異なる通貨ポリシーを取るためにこれらの主要な特徴を少しずつ調整してきたのです。何百ものAlt coinの中で、最も特徴的な例をいくつか次に紹介していきます。

### Litecoin

まず1つ目は2011年にリリースされたLitecoin(ライトコイン)であり、Bitcoinに次いで二番目に成功しているデジタル通貨です。主要なイノベーションは *scrypt* (Tenebrixから受け継いでいる)をproof of workのアルゴリズムとして使用していることであり、これにより高速で軽量のブロック生成速度になっています。

- ブロック生成時間 2.5分
- 通貨総発行量: 2140年までに8400万litecoin
- コンセンサスアルゴリズム: Scrypt proof of work
- 時価総額: 2014年中旬に1億6000万米ドル

### Dogecoin

Dogecoin(ドージコイン)は2013年12月にリリースされたもので、Litecoinのフォークに基づくものです。Dogecoinは速い通貨発行速度と高い通貨時価総額を通貨ポリシーとして持っています。これはDogecoinがチップとして使われることを推奨しているためであり、これがためにDogecoinには特筆すべき価値があります。初めはジョークとして始まりましたが、2014年に急速に価値が下がる前は大変有名になり活発で大きなコミュニティになっていました。

- ブロック生成時間: 60秒
- 通貨総発行量 : 2015年までに 100,000,000,000 (1000億) doge
- コンセンサスアルゴリズム: Scrypt proof of work
- 時価総額: 2014年中旬に1200万米ドル

### Freicoin

Freicoin(フレイコイン)は2012年7月に発表されたもので、 *demurrage*通貨(時間とともに減価する通貨)になっています。demurrage通貨は保存している価値に対してマイナスの利子率を持っているものです。Freicoinでは消費することを奨励しお金を所持し続けることを抑制するために、Freicoinを持っていると年率4.5%の手数料が課されるようになっています。FreicoinはBitcoinのデフレ通貨ポリシーとは正反対の通貨ポリシーを導入しているという点で特筆すべき価値があります。Freicoinは通貨としては成功してはいませんが、Alt coinによるいろいろな通貨ポリシーの興味深い例となっています。

- ブロック生成時間: 10分

- 通貨総発行量: 2140年までに 1億コイン
- コンセンサスアルゴリズム: SHA256 proof of work
- 時価総額: 2014年中旬で13万米ドル

## コンセンサスのイノベーション: Peercoin, Myriad, Blackcoin, Vericoïn, NXT

BitcoinのコンセンサスメカニヰムはSHA256アルゴリズムを用いたproof of workに基づいています。最初のAlt coinはproof of workに代わる手段としてscryptを導入し、マイニングプロセスをよりCPUに優しくなるようにしASICSによる中央集権化にならないようにしました。それ以降コンセンサスメカニヰムにおけるイノベーションは激しいペースで続いていきました。いくつかのAlt coinはscryptのような様々なアルゴリズムを採用していきました。("Blake algorithm")) Scrypt-N、Skein、Groestl、SHA3、X11、Blakeなどなどです。いくつかのAlt coinは様々なproof of workアルゴリズムと結びついていきました。そして2013年にはproof of workの代替手段を発明するに至りました。それは proof of stakeと呼ばれるもので、多くの最近のAlt coinはこれをベースに構成されています。

proof of stakeは既存の通貨所有者が利付き担保のように通貨の"所有を主張(stake)"することができるシステムです。それはなんとなく譲渡性預金(CD)に似ていて、参加者は投資のリターンを新しい通貨(利子の支払いとして発行される)またはトランザクションの手数料という形で稼ぎながら、通貨の一部を保有することができます。

### Peercoin

Peercoin(ピアコイン)は2012年8月に発表され、proof of workとproof of stakeのアルゴリズムを両方取り入れた通貨として最初に発行されたAlt coinです。

- ブロック生成時間: 10分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム: (ハイブリッド)proof of workで始まり、あとでproof of stake
- 時価総額: 2014年中旬で1400万米ドル

### Myriad

Myriad(ミリアド)は2014年2月に発表され、5つの異なるproof of workアルゴリズム(SHA256d、Scrypt、Qubit、Skein、Myriad-

Groestl)を同時に使っており、マイナーの参加状態に応じてそれぞれのアルゴリズムごとに異なるdifficultyを持っています。その意図は、多数のマイニングアルゴリズムを同時に攻撃しなければならないようにすることによって、MyriadをASICによる専業化と中央集権化に対して耐久性のあるものにし、さらにコンセンサス攻撃に強くすることにあります。

- ブロック生成時間: 30秒が平均(マイニングごとに2.5分がターゲット)
- 通貨総発行量 2024年までに20億コイン
- コンセンサスアルゴリズム: 複数proof of workアルゴリズム
- 時価総額: 2014年中旬に12万米ドル

## Blackcoin

Blackcoin(ブラックコイン)は2014年の冬に発表され、proof of stakeのコンセンサスアルゴリズムを用いています。特筆すべき点は"multipools"を導入したことです。multipoolsは、利益に応じてどのAlt coinをマイニングするかを自動的に変更するマイニングプールの一種です。

- ブロック生成時間: 1分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム: proof of stake
- 時価総額: 2014年月中旬で370万米ドル

## VeriCoin

VeriCoin(ベリコイン)は2014年5月に始まり、供給と需要のマーケットバランスに基いて動的に変更される金利レートを持つproof of stakeコンセンサスアルゴリズムを用いています。また、これはウォレットからbitcoinで支払いをする際に、自動的にbitcoinに両替をする機能を導入した初めてのAlt coinでもあります。

- ブロック生成時間: 1分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム: proof of stake
- 時価総額: 2014年月中旬で110万米ドル

## NXT

NXT (発音は"Next")は、proof of workによるマイニングを用いていないという点で"純粋な"proof of stakeのAlt coinです。NXTはスクラッチから実装された暗号通貨で、Bitcoinのフォークでも、他のAlt coinのフォークでもありません。NXTはたくさんの先進的な機能を実装しており、例えば名前登録機能Namecoinと似ている)や、分散型アセット取引所(Colored Coinsと似ている)、安全な統合型分散メッセージング("Bitmessage" Bitmessageと似ている)、そしてstakeデリゲーション(proof of stakeを他人に委任すること)があります。NXTに惹き付けられている人はNXTを"次世代"または暗号通貨2.0と呼んでいます。

- ブロック生成時間: 1分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム: proof of stake
- 時価総額: 2014年月中旬で3000万米ドル

## 2つの目的を持ったマイニングのイノベーション: Primecoin, Curecoin, Gridcoin

Bitcoinのproof of workアルゴリズムには目的が一つだけあります。それはBitcoinネットワークをセキュアに保つことです。伝統的な決済のシステムのセキュリティコストに比べて、マイニングのコストは高くはありません。しかし、多くの人に"無駄が多い"として批判されています。次世代

のAlt coinは、この問題に対処しようとしています。2つの目的を持つproof of work アルゴリズムは、Bitcoinネットワークをセキュアに保つproof of workである一方、特定の"有用な"問題を解決することができます。ただ通貨のセキュリティを保つということ以外の目的を導入することのリスクは、通貨の需要供給曲線に別の目的による外的影響も加えられてしまうことがあります。

## Primecoin

Primecoin(プライムコイン)は2013年7月に始まりました。Primecoinで使っているproof of workアルゴリズムでは素数の探索も行うのです。素数探索にはカニンガム鎖(第1カニンガム鎖と第2カニンガム鎖)とbi-

twinチェーンという素数列を計算することで素数を探索しています。素数はさまざまな科学の領域でとても有用なものとなっています。Primecoinのブロックチェーンにはproof of workで発見された素数が含まれており、このためパブリックなトランザクション元帳の生成と並行して科学的な発見の公的記録を生成することにもなるのです。

- ブロック生成時間: 1分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム: 素数列の発見を伴うproof of work
- 時価総額: 2014年中旬に130万ドル

## Curecoin

Curecoin(キュアコイン)は2013年5月に発表されました。SHA256のproof of work アルゴリズムを使って、Folding@Homeプロジェクトのタンパク質フォールディング研究と結び付けようとするものです。タンパク質フォールディングとは、病気を治す新薬の発見のために使われている、タンパク質の化学的な相互作用の解析をコンピューターによる膨大なシミュレーション計算で行うことです。

- ブロック生成時間: 10分
- 通貨総発行量: 無制限
- コンセンサスアルゴリズム; タンパク質フォールディングの研究を伴うproof of work
- 時価総額: 2014年中旬に58,000米ドル

## Gridcoin

Gridcoin(グリッドコイン)は2013年10月に発表されました。Gridcoinではscryptベースのproof of workを行っており、BOINCオープングリッドコンピューティングへの参加報奨金を伴うproof of workである。BOINCはBerkeley Open Infrastructure for Network Computingであり、科学的なグリッドコンピューティングに対してのオープンプロトコルです。参加者はコンピューティングリソースが必要な幅広い科学的な研究のために空いている演算処理リソースを共有できます。GridcoinはBOINCを、例えば素数やタンパク質フォールディングのような特定の科学問題を解決するためにだけでなく、一般的な目的のコンピューティングプラットフォームとして用いていることができます。

- ブロック生成時間: 150秒
- 通貨総発行量: 無制限

- コンセンサスアルゴリズム: BOINCのグリッドコンピューティングに対する参加報酬金を伴うproof of work
- 時価総額: 2014年中旬に122,000米ドル

## 匿名性に集中したAlt coin: CryptoNote Bytecoin Monero, Zerocash/Zerocoin, Darkcoin

Bitcoinはよく"匿名"のコインとして間違われます。実際には、BitcoinのアドレスとIDを紐付けることは簡単です。そしてビッグデータ解析を使えば、ある人がどのようにbitcoinを使っているかの包括的に見ていくことでBitcoinアドレスと個人を結び付けることができます。いくつかのAlt coinでは、強い匿名性を目的としているものがあります。その最初の取り組みはおそらく Zerocoin です。Zerocoinは、2013年のIEEEシンポジウム IEEE Symposium on Security and Privacy で発表された論文にある、Bitcoin上で匿名性を保つためのメタコインです。本書を執筆しているときはまだ開発中になっていますが、Zerocoinは将来Zerocoinと完全に分けられたZerocashというAlt coinとして実装されるでしょう。他の匿名性へのアプローチとしては、2013年10月に発表された論文にある\_CryptoNote\_があります。CryptoNoteは根本的な技術で、次に説明していく数多くのAlt coinでCryptoNoteを実装しています。ZerocashとCryptoNoteに加え、他にもいくつかの独立した匿名のコインが存在しています。例えば、Darkcoinはステルスアドレスと、匿名性のためのトランザクションのre-mixingを行っています。

### Zerocoin/Zerocash

Zerocoin(ゼロコイン)は、Johns Hopkins大学の研究者によって2013年に導入されたデジタル通貨で、匿名性に対する理論的な試みです。ZerocashはZerocoinのAlt coinとしての実装になっています。Zerocashはまだ開発段階でありリリースはされていません。

### CryptoNote

CryptoNote(クリプトノート)は、匿名性を持つデジタルキャッシュの基礎技術を提供する2013年10月に発表されたオルトコインのリファレンス実装です。異なる様々な実装にフォークできるように設計されており、一定周期で通貨として利用できなくなるようにリセットするメカニズムを備えています。いくつかのオルトコインはCryptoNoteから生まれてきたものです。例えば、Bytecoin (BCN)、Aeon (AEON)、Boobycoin (BBR)、duckNote (DUCK)、Fantomcoin (FCN)、Monero (XMR)、MonetaVerde (MCN)、そして Quazarcoin (QCN)です。CryptoNoteはBitcoinのフォークではなく、別枠として完全な形でまとめられた暗号通貨の実装であるということに特筆すべき価値があります。

### Bytecoin

("Graphical Processing Units (GPUs)"Bytecoin(バイトコイン)はCryptoNoteから生まれた最初の実装であり、CryptoNoteの技術に基づく匿名性のある通貨を提供しています。Bytecoinは2012年7月に始まりました。BTEという通貨シンボルを持つBytecoinが以前もありました。一方、CryptoNoteに由来するBytecoinの通貨シンボルはBCNです。Bytecoinが使用しているCryptonight proof of workアルゴリズムでは、少なくとも2MBのRAMをインスタンスごとに必要としており、GPUやASICのマイニングには適さないようになっています。BytecoinはCryptoNoteのリング署名、リンク不可能なトランザクションを使用しており、ブロックチェーン解析に対する対策としての匿名性を持っています。

- ブロック生成時間: 2分
- 通貨総発行量: 1840億BCN
- コンセンサスアルゴリズム: Cryptonight Proof of work
- 時価総額 2014年中旬に300万米ドル

## Monero

Monero(モネロ)はCryptoNoteの実装の一つです。これはBytecoinよりも幾分フラットな発行曲線を持っていて、80%の通貨は最初の4年間の間に発行されるようになっています。そして、CryptoNoteから継承したBytecoinと同様の匿名性の機能を持っている。

- ブロック生成時間: 1分
- 通貨総発行量: 1840万XMR
- コンセンサスアルゴリズム: Cryptonight Proof of work
- 時価総額: 2014年中旬の500万米ドル

## Darkcoin

Darkcoin(ダークコイン)は2014年1月に始まりました。Darkcoinは匿名性を持つ通貨として実装されており、DarkSendと呼ばれる全てのトランザクションに対してre-mixingを行って匿名化するプロトコルを使っています。Darkcoinはまた、proof of workアルゴリズムに11個のハッシュ関数 (blake、bmw、groestl、jh、keccak、skein、luffa、cubehash、shavite、simd、echo)を用いていることも特筆すべき価値になっています。

- ブロック生成時間: 2.5分
- 通貨総発行量: 最大2200万DRK
- コンセンサスアルゴリズム: 複数のラウンド関数を持った複数proof of workアルゴリズム
- 時価総額: 2014年中旬に1900万米ドル

## 通貨ではないAlt chain

### Alt

chainは、ブロックチェーンを別の設計思想で実装したものであり、それは主に通貨としては使われていないものです。多くは通貨を含みますが、その通貨は他の何かを導入するためのトークンとして使われます。例えば、リソースや契約としてです。言い換えれば、Alt chainにおける通貨は、プラットフォームの主要な点ではなく、2番目の機能なのです。

## Namecoin

Namecoin(ネームコイン)はBitcoinコードの最初のフォークでした。Namecoinはブロックチェーンを使った分散型のキーバリューの登録・移管プラットフォームです。グローバルなドメイン名のレジストリとなっており、インターネット上のドメインネーム登録システムと似ている。Namecoinは現在のDNS(domain name service)の代替として作成されており、ルートレベルのドメイン名は .bit となっています。



Namecoinはまた、ドメイン名の登録と別の名前空間におけるキーバリューペアの登録に使われています。Emailアドレスや、暗号キー、SSL証明書、ファイル署名、投票システム、株式証明書、そして他の数えきれない程のアプリケーションに使われています。

NamecoinシステムはNamecoinの通貨(シンボルはNMC)を含んでおり、登録と名前の移管のためにトランザクション手数料を払います。現在の価格では、トランザクション手数料は0.01NMCかまたは約1米セントになっています。Bitcoinと同じように、この手数料はNamecoinのマイナーによって集められます。

Namecoinの基本的な通貨パラメーターはBitcoinと同一です。

- ブロック生成時間: 10分
- 通貨総発行量: 2140年までに2100万NMC
- コンセンサスアルゴリズム: SHA256 proof of work
- 時価総額: 2014年中旬に1000万米ドル

Namecoinの名前空間は何も制限されておらず、誰でもどんな名前でもどんな方法でも登録できます。しかし、名前がブロックチェーンから読まれる時に、アプリケーションレベルのソフトウェアが名前をどのように読みだしていけばいいかを知っているようにしなければいけないため、特定の名前空間はあらかじめ決められた仕様になっています。もし形式が間違っていれば、特定の名前空間から名前を読み出そうとするときにどんなソフトウェアでもエラーが吐き出されることとなります。よく知られている名前空間は以下のようなものです。

- `d/` は `.bit` ドメインのための名前空間です。 `id/` は個人のIDを保存するための名前空間です。例えば emailアドレスやPGPのキーなどです。 `u/` (は追加的なもので、(openspecに基づく)IDを保存するためのより構造化された名前空間です。

NamecoinのクライアントはBitcoinのコアととても良く似ています。というのは、同じソースコードからできていたためです。インストールの際に、クライアントはNamecoinのフルブロックチェーンをダウンロードし、クエリを発行したり名前の登録したりできるようになります。3つの主要なコマンドは以下です。

#### `name_new`

クエリを発行するか、または名前の事前登録を行います。

#### `name_firstupdate`

名前を登録し、登録した名前を公開します。

#### `name_update`

名前の詳細を変更するか、または登録した名前を再読み込みします。

例えば、 `mastering-bitcoin.bit+` のドメインを登録しようと思ったら、以下のように `+name_new` のコマンドを使用します。

```
$ namecoind name_new d/mastering-bitcoin
```

```
[
  "21cbab5b1241c6d1a6ad70a2416b3124eb883ac38e423e5ff591d1968eb6664a",
  "a05555e0fc56c023"
]
```

`name_new` コマンドは Namecoin ブロックチェーンに名前登録要求を登録し、ランダムなキーとともに名前のハッシュを作成します。 `name_new+` の戻り値となる2つの文字列は名前のハッシュとランダムキーです(ランダムキー `+a05555e0fc56c023` はさきほどのコマンド実行例の出力結果にあります)。このランダムキーは名前をパブリックにするために使います。一旦名前登録要求が Namecoin のブロックチェーン上に登録されると、ランダムキーを `name_firstupdate` コマンドに渡して実行することで名前がパブリックになります。

```
$ namecoind name_firstupdate d/mastering-bitcoin a05555e0fc56c023 '{"map": {"www": {"ip": "1.2.3.4"}}}'
b7a2e59c0a26e5e2664948946ebeca1260985c2f616ba579e6bc7f35ec234b01
```

この例では `www.mastering-bitcoin.bit` を IP アドレスの `1.2.3.4` にマッピングしています。戻り値となるハッシュはトランザクションIDであり、これはこの登録を追跡するために使われます。そして、 `name_list` コマンドを実行するとどの名前が登録されているかを確認することができます。

```
$ namecoind name_list
```

```
[
  {
    "name" : "d/mastering-bitcoin",
    "value" : "{map: {www: {ip:1.2.3.4}}}",
    "address" : "NCccBXrRUahAGrisBA1BLPWQfSrups8Geh",
    "expires_in" : 35929
  }
]
```

Namecoinの登録は36,000ブロックごとに更新される必要があります(およそ200日から250日ごと)。

`name_update` コマンドを実行するには何も手数料を払う必要がなく、無料で Namecoin のドメインを更新できます。サードパーティープロバイダに頼めば、少ない手数料で登録、自動更新、またはウェブインターフェイスを通じた更新をハンドリングしてくれます。サードパーティープロバイダにお願いしておけば Namecoin クライアントを走らせておこななくても済みますが、Namecoin による分散型名前登録という独立したコントロールを失うこととなります。

## Ethereum

Ethereum(エセリウム)はブロックチェーンの台帳上において、チューリング完全な契約処理と、それを実行するプラットフォームです。Bitcoinの複製ではなく、完全に独立した仕様と実装を持っています。Ethereumは、組み込みの通貨を持っており、それは *ether*(イーサ) と呼ばれています。etherは契約実行のために必要なものです。Ethereumのブロックチェーンの記録は *contract*(コントラクト) であり、それは低位の言語、バイトコードのようなチューリング完全なコードで書かれています。本質的に、contractはEthereumシステム上で動くプログラムです。Ethereumのcontractでは、データを保存することができ、etherの支払いをしたり受け取ったりできます。そして計算可能な長さ無制限の処理(それ故にチューリング完全なのです)を分散型で自律的なソフトウェアのエージェントとして実行することができます。

Ethereumではとても複雑な、他のAlt chain自身では実装できないようなシステムを実装することができます。例えば、Namecoinのような名前登録のcontractは、下記のようにEthereumで書けてしまいます(より正確に言えば、Ethereumコードにコンパイルされる高級言語で書かれます)。

```
if !contract.storage[msg.data[0]]: # Is the key not yet taken?
    # Then take it!
    contract.storage[msg.data[0]] = msg.data[1]
    return(1)
else:

    return(0) // Otherwise do nothing
```

## 未来の通貨

次世代の暗号通貨は全般的に、将来のBitcoinよりも見通しが明るくさえあります。Bitcoinは分散型組織と分散型コンセンサスの全く新しい形を導入し、何百もの驚くべきイノベーションを生み出しました。これらの発明は分散システム科学から、金融、経済学、通貨、中央銀行、そしてコーポレートガバナンスまで経済の幅広い領域に影響をもたらしていくでしょう。以前は権威ある、または信用されている管理中枢のような中央集権的な組織や機構が、多くの人間が絡む活動で必要とされてきました。しかし、今やそれらを分散させることができるようになりました。ブロックチェーンと全く新しいコンセンサスシステムの発明は、権力の集中や腐敗、規制による封じ込めが起きる機会を取り除き、大規模なシステムで生じてしまう組織と協調のコストを大きく削減することになるのです。

# Bitcoinの安全性

Bitcoinは、銀行口座とは異なり、残高の数値を参照しているものではないため、安全性を保つことは大変です。bitcoinは、デジタルな現金や金に極めて近いものです。"実際に持っている者が9割方勝つ"という言い回しを聞いたことがあるかもしれませんが、bitcoinでは実際に持っている者が10割方勝ちます。Bitcoinの鍵を持っていることは、現金や貴金属の塊を持っていることと同じです。なくすことも、置き忘れることも、盗まれることも、誰かに誤った量を渡してしまうこともあります。これらのどのケースでも、歩道に現金を落としってしまったときと同様に、ユーザーはbitcoinを使うことができなくなります。

しかしながら、Bitcoinは、現金や金や銀行口座にはできないことができます。鍵が納められたBitcoinウォレットは、ファイルのようにバックアップが可能です。いくつもコピーを作って保存できますし、バックアップとして紙に印刷することもできます。現金や金や銀行口座は、"バックアップ"することはできません。Bitcoinは既存のどのようなものとも異なるので、私たちはBitcoinの安全性について考える時も、今までにない新しい方法で行う必要があります。

## 安全性の原則

Bitcoinの重要な原則は分散化であり、このことはBitcoinの安全性について重要なインプリケーションを導きます。伝統的な銀行や支払ネットワークのような集中化モデルは、アクセス制限と審査によって、悪意のある主体をシステムから遠ざけます。一方、Bitcoinのような分散化システムは、責任と管理権をユーザーに付与します。ネットワークの安全性はproof of workに基づくものであってアクセス制限によるものではないため、ネットワークはオープンであり、Bitcoinのトラフィックに暗号化は必要ありません。

クレジットカードのような伝統的な支払ネットワークでは、ユーザーのプライベートな識別情報(クレジットカード番号)を含んでいるので、支払いは無制限です。最初に課金してからも、その識別情報にアクセスできる者であれば、何度でもユーザーに課金してお金を"引き出す"ことができます。したがって支払ネットワークは、暗号化により徹底的に保護されなければなりませんし、支払いのトラフィックが、プロセスの途中で安全に保存された状態であれ、盗聴者や媒介者により漏洩され得ないことを保証しなければいけません。もし、悪意のある主体がシステムにアクセスできたら、現在のトランザクションと支払トークンの\_両方を\_手に入れ、これらを用いて新たなトランザクションを作り出すことが可能になります。さらに悪いことに、顧客情報が漏洩した場合には、その顧客はなりすましにさらされ、アカウントの不正使用を防ぐために行動を起こさなければならなくなります。

Bitcoinは全く異なります。Bitcoinのトランザクションは、特定の受取人に対する特定の金額のみを承認するものであって、捏造されることも変更されることもありません。また、取引主体を特定するようなどんなプライベート情報も明かしませんし、追加的に別の支払いの承認に用いることもできません。従って、Bitcoinの支払ネットワークには、暗号化も盗聴者からの保護も必要ありません。実際、ユーザーは、厳格に保護されていない、WiFiやBluetoothといったオープンな公共チャネルを通じて、安全性を損なうことなくトランザクションをブロードキャストできます。

Bitcoinの分散化された安全性モデルは、ユーザーの手に強い力を授けますが、その力は、鍵の秘匿性の維持に対する責任とともに与えられるものです。ほとんどのユーザーにとって、鍵の秘匿性の維持は簡単なことではありません。インターネットに接続したスマートフォンやPCといった、一般的な用途のコンピュータデバイスでは、特にそうです。Bitcoinの分散化モデルはクレジットカードで起こるような大規模な漏洩は防げるものの、多くのユーザーは鍵を適切に保護できておらず、ひとつずつハックされえます。

## Bitcoinのシステムを安全に開発する

Bitcoin関連のソフトウェア開発者にとって、最も重要な原則は分散化です。ほとんどの開発者は集中化された安全性モデルに慣れており、開発しているBitcoinアプリケーションにこのモデルを適用する誘惑に駆られることですが、それは悲惨な結果を生むことになります。

Bitcoinの安全性は、鍵の分散管理と、マイナーによる独立したトランザクション認証に基づいています。Bitcoinならではの安全性を活用したいのであれば、Bitcoinの安全性モデルの考え方から離れないでいる必要があります。つまり、鍵の管理はユーザーに任せよ、トランザクションはブロックチェーンに任せよ、ということです。

たとえば、初期のbitcoin交換所は、すべてのユーザーの資金をひとつの"ホットな"ウォレットに集め、鍵とともにひとつのサーバーに保存しました。こうした設計は、ユーザーから鍵の管理を取り上げ、単一のシステムに集約するものです。こうした多くのシステムはハックされ、顧客にとって悲惨な結果を招いています。

よくあるもうひとつの誤りは、トランザクション手数料を削減するため、またはトランザクションの処理プロセスを早めるための間違った努力のうちに、"ブロックチェーン外"でトランザクションを作ることです。"ブロックチェーン外"システムは、内部の集中化された元帳にトランザクションを記録し、Bitcoinのブロックチェーンとたまに同期するといったものです。こうした実践もまた、分散化されたBitcoinの安全性を、独占され集中化されたアプローチで置き換えてしまうものです。トランザクションがブロックチェーンの外で作られると、適切に保護されていない集中化された元帳は改竄される可能性があり、その場合、気づかれないうちに、資金が流用され蓄えが使い切られることとなります。

操作上の安全性、多重のアクセス制限、(伝統的な銀行が行っているような)監査といったものに多額の投資を行う用意がない限り、Bitcoinの分散化された安全性の考え方に反した状態で資金を持つことには、よほど慎重でなければいけません。たとえ、頑健な安全性モデルを実装するだけの資金と規律があったとしても、なりすましや買収や横領に苦しめられてきた、伝統的な金融ネットワークという脆弱なモデルを、複製しているに過ぎないのです。Bitcoinのユニークな分散化された安全性モデルの強みを活かすためには、親しみは感じるけれども結局はBitcoinの安全性を脅かすことになる、集中化されたアーキテクチャへの誘惑を断たねばなりません。

## 信用の根源

伝統的な安全性のアーキテクチャは「信用の根源」という概念に基づいています。それは、システムやアプリケーション全体の安全性の礎となる、信用の中核です。安全性のアーキテクチャは、信用の根源を中心にして同心円状に(タマネギのように)構成され、中心から外向きに信用が広がっていく形をとります。各層は、より信用度の高い内側の層の上に成り立つもので、内側の層は、アクセス制限、デジタル署名、暗号化といった、安全性に関する基本要素を用いています。ソフトウェアシステムが複雑化するにつれ、システムの安全性が脅かされるようなバグを含むことが多くなってきています。結果、ソフトウェアがより複雑なものになると、それを安全な状態に保つことはより難しくなります。信用の根源の概念をもとにすると、信用のほとんどは、システムのうちで最も複雑でない部分、従って最も脆弱でない部分に置かれ、その周囲に複雑なソフトウェアが層を成すこととなります。この安全性のアーキテクチャは、異なる規模で繰り返されます。すなわち、最初は単一のシステムのハードウェアに信用の根源が据えられ、その信用の根源が、OSを通じてより高いレベルのシステムの働きにまで拡張され、最終的には、外に向かって信用の度合いが低下する同心円上に配置された、多くのサーバーまで広がるのです。

Bitcoinの安全性のアーキテクチャは異なります。Bitcoinでは、合意形成システムは、信用できる公開された元帳を作り出し、その元帳は完全に分散化されたものです。正しく認証されたブロックチェーンはgenesisブロックを信用の根源として用い、最新のブロックまで続く信用の連鎖を作り上げます。Bitcoinシステムはブロックチェーンを信用の根源として用いることができますし、またそのようにしなければいけません。サービスの提供先が多くの異なるシステムであるような、複雑なBitcoinアプリケーションを設計するときには、ど

ここに信用が置かれようとしているかを明確にするため、安全性のアーキテクチャを注意深く精査する必要があります。結局、明確に信用されるべき唯一のものは、完全に認証されたブロックチェーンなのです。明示的であってもそうでなくても、あるアプリケーションがブロックチェーン以外のなにかに信用の基礎を置いている場合、脆弱性を招き入れることになり、心配の種になるに違いありません。開発中のアプリケーションの安全性のアーキテクチャを評価する良い方法は、個々の構成要素を考え、その構成要素が悪意のある主体の管理下に置かれ、完全に毀損されるという仮説のシナリオを吟味することです。構成要素を一つ一つとりあげ、その構成要素が毀損されたとしたときの、全体の安全性に与える影響度を算定するのです。もし、構成要素が毀損されたときにアプリケーションが安全でなければ、それらに誤って信頼を置いていたことが分かります。脆弱性のないbitcoinアプリケーションとはつまり、Bitcoinの合意形成メカニズムが毀損されることに対してのみ、脆弱なアプリケーションです。それは、信用の根源が、Bitcoinの安全性のアーキテクチャの中でも、最も強い部分に基づいていることを意味します。

ハックされたbitcoin交換所の数多くの事例を見れば、この点ははっきり分かります。なぜなら、彼らの安全性のアーキテクチャと設計は、誰からも全くと言っていいほど監督されていないからです。これらの集中化されたアーキテクチャやデザインの実装は、ブロックチェーンの外にある多くの構成要素、すなわち、ウォレット、集中化された元帳データベース、脆弱な暗号キーといったものを、信用できるものとして明確に位置付けていました。

## 安全性に関するベストプラクティス

人類は、物理的な脅威から安全を守る手段を、何千年も用いてきました。これに対して、デジタルの世界の安全性に関する私たちの経験は、50年にも満たないものです。近頃の一般的な目的のOSは、非常に安全とは言えず、特にデジタルマネーの保存に適しているというわけでもありません。私達のコンピューターは、インターネットへの常時接続を通じ、外界からの脅威に常に晒されています。コンピューターは、多くのプログラマーによって作られたソフトウェアを何千と走らせ、そうしたソフトウェアは時として何の制約もなくユーザーのファイルにアクセスします。コンピューターにインストールされた何千ものソフトウェアのうち、詐欺ソフトがひとつでも紛れていたら、キーボードとファイルに障害が起り、ウォレットアプリケーションに保管されたbitcoinが盗まれるかもしれません。コンピューターをウィルスやトロイの木馬がない状態に保つために、必要なメンテナンスの水準は、大多数のユーザーの技術を超えています。

何十年にも亘る情報セキュリティに関する研究と進歩にも関わらず、デジタル資産は、明確な敵対者に対し、いまだに情けないほど脆弱です。金融機関や諜報機関、防衛産業における、最も高度に保護され機密性の高いシステムでさえ、頻繁に破られます。Bitcoinが作り出すデジタル資産は、固有の価値を持っており、盗難されてすぐに新たな所有者に送られ、戻ってこなくなり得るものです。このことは、ハッカーにとって大きなインセンティブとなります。これまでハッカーは、クレジットカードや銀行口座といった、個人情報や口座にアクセスするためのトークンを、盗み出した後に価値あるものに変換する必要がありました。盗んだ金融情報の換金やロンダリングは困難であるにもかかわらず、盗難は増える一方でした。Bitcoinはそれ自体が価値であるため、換金やロンダリングの必要がなく、この問題をより大きくしています。

幸いなことに、Bitcoinはまた、コンピューターのセキュリティを改善するインセンティブをもたらします。以前は、コンピューターを危機に晒すリスクは、漠然としていて間接的でしたが、Bitcoinはこのリスクを明確なものにしています。bitcoinをコンピューターに保持することは、そのユーザーの関心を、コンピューターのセキュリティ改善の必要性に集中させることにつながります。Bitcoinや他のデジタル通貨の激増と拡がりの直接的な帰結として、私たちは、ハッキング技術とセキュリティソリューションの両方が発展していく様子を見てきました。簡単に言えば、ハッカーは今やとても美味しいターゲットを見つけ、ユーザーは自分自身を守る明確なインセンティブを持っているわけです。

社会におけるBitcoinの受容の直接的な帰結として、過去3年にわたって、ハードウェア暗号化、鍵保管、ハードウェアウォレット、マルチシグネチャ技術、デジタルエスクローといった、情報セキュリティ分野における素晴らしいイノベーションを、私たちは目の当たりにしてきました。次節では、実践的なユーザーセキュリティのための、多様なベストプラクティスを詳しくみていくことにしましょう。

## 物理的なbitcoinの保管

多くのユーザーは、情報として安全が保たれていることよりも、物理的に安全が保たれていることのほうに、はるかに安心を感じるので、bitcoinを物理的な形に変換するという事は、bitcoinの安全な保持のために大変有効な方法です。Bitcoinの鍵は単なる長い数字の羅列です。このことは、鍵が、紙への印刷や、金属のコインへの刻印といった、物理的な形態で保存され得ることを意味しています。従って、鍵を安全に保つことは、Bitcoinの鍵の紙のコピーを安全に保つことになります。紙に印刷されたBitcoinの鍵は"ペーパーウォレット"と呼ばれ、これを作るためのフリーツールが数多くあります。私は個人的に、自分のbitcoinのほとんど(99%以上)を、ペーパーウォレットに保存しています。そのウォレットはBIP0038で暗号化され、複数のコピーが鍵をかけた金庫に閉じ込んであります。Bitcoinをオフラインにしておくことは\_コールドストレージ\_と呼ばれ、最も有効なセキュリティのテクニックのひとつです。コールドストレージシステムでは、鍵はオフラインシステム(インターネットに一度も接続したことがないシステム)で生成され、紙であれUSBメモリスティックのようなデジタルメディアであれ、オフラインで保存されます。

## ハードウェアウォレット

長い目で見れば、Bitcoinの安全性は、改竄に耐性のあるハードウェアウォレットの形態を、ますますとるようになるでしょう。スマートフォンやデスクトップコンピューターとは異なり、bitcoinのハードウェアウォレットには、bitcoinを安全に保有するというただひとつの目的しかありません。漏洩の原因となり得る一般的なソフトウェアがなく、インターフェースも限られるために、ハードウェアウォレットは、専門家でないユーザーに絶対と言っていいほど確実なセキュリティをもたらします。私は、ハードウェアウォレットは、bitcoin保有の方法として広く用いられるようになるかと予想しています。このようなハードウェアウォレットの一例は、[Trezor](#) をご覧ください。

## リスク配分の適正化

ほとんどのユーザーは、bitcoinの盗難について適切に注意を払っていますが、盗難よりも大きなリスクが存在します。データファイルはいつでも失われます。もしそのファイルがbitcoinを含んでいたら、極めて悲惨なことになります。Bitcoinウォレットを安全に保護しようと努力するあまり、やり過ぎてbitcoinを失うことにならないよう、ユーザーは注意深くならなければいけません。2011年7月、有名なBitcoinの教育啓発プロジェクトが、ほぼ7,000

bitcoinを失いました。盗難防止の努力の一環として、彼らは複雑に暗号化されたバックアップを実施していました。結局、彼らは暗号化鍵を誤って失くし、バックアップは無価値となり、財産を失いました。砂漠に埋めてお金を隠すように、bitcoinを安全にし過ぎると、二度と見つけれなくなるかもしれません。

## リスク分散

あなたは、全財産を現金で財布の中に入れて持ち運びますか？ほとんどの人はこのようなことは向こう見ずと考えるのにもかかわらず、Bitcoinユーザーは、全てのbitcoinをひとつのウォレットに入れてしまうことがよくあります。そうではなくて、ユーザーは、複数の多様なBitcoinウォレットに、リスクを分散しなければなりません。慎重なユーザーは、自分のbitcoinのうちほんの少しだけ、おそらく5%に満たない程度を、オンラインまたはモバイルウォレットに"小銭"として持つようにしています。残りのbitcoinは、デスクトップウォレットやオフライン(コールドストレージ)のような、複数の異なる仕組みで保管されなければなりません。

## マルチシグネチャと管理

企業や個人が多額のbitcoinを保管するときにはいつでも、マルチシグネチャbitcoinアドレスを用いることを考慮すべきです。マルチシグネチャアドレスは、支払いに複数の署名を要求することで、資金を安全に守るもの

です。署名のための鍵は、複数の異なる場所に保管され、別々の人によって管理されなければなりません。たとえば、企業では、署名のための鍵が別々につくられ、複数の幹部によって保持され、誰であれ一人では資金に手を出せないようになっていないといけません。マルチシグネチャアドレスを用いることで、冗長性を得る、すなわち、一人の人が複数の鍵を別々の場所に保持することも、可能となります。

## サバイバビリティ

安全性に関して、見過ごされがちですが考慮すべき重要な点として、鍵の持ち主が動けなくなったり死亡したりした場合に、どうやってbitcoinを手にするかということがあります。Bitcoinユーザーは、複雑なパスワードを用い、鍵を安全で秘匿された状態に保ち、誰とも共有してはいけなく、と言われてきました。不幸なことに、こうしたプラクティスによって、ユーザー自身が鍵を使えないときには、ユーザーの家族が資金を再び手にするのはほぼ不可能です。実際、Bitcoinユーザーの家族は、bitcoinの資産の存在に全く気付かないことがほとんどでしょう。

多額のbitcoinを持っているのであれば、アクセス方法の詳細を、信頼のおける親戚や弁護士と共有することを考えるべきです。より複雑なサバイバビリティの仕組みは、マルチシグネチャアクセスと、"デジタル資産の遺言執行"の専門弁護士を通じた資産計画から、構成することができます。

## 結び

Bitcoinは全く新しい、前例のない、そして複雑なテクノロジーです。いずれは、専門家でないユーザーにも使いやすい、安全性のためのツールやプラクティスが開発されることでしょう。差し当たり、Bitcoinユーザーは本書で扱った多くのTIPSを用いることで、安全でトラブルのないBitcoinエクスペリエンスを楽しむことができます。



# Appendix A: Bitcoin Explorer (bx) コマンド

Usage: bx COMMAND [--help]

Info: The bx commands are:

address-decode  
address-embed  
address-encode  
address-validate  
base16-decode  
base16-encode  
base58-decode  
base58-encode  
base58check-decode  
base58check-encode  
base64-decode  
base64-encode  
bitcoin160  
bitcoin256  
btc-to-satoshi  
ec-add  
ec-add-secrets  
ec-multiply  
ec-multiply-secrets  
ec-new  
ec-to-address  
ec-to-public  
ec-to-wif  
fetch-balance  
fetch-header  
fetch-height  
fetch-history  
fetch-stealth  
fetch-tx  
fetch-tx-index  
hd-new  
hd-private  
hd-public  
hd-to-address  
hd-to-ec  
hd-to-public  
hd-to-wif  
help  
input-set

```
input-sign
input-validate
message-sign
message-validate
mnemonic-decode
mnemonic-encode
ripemd160
satoshi-to-btc
script-decode
script-encode
script-to-address
seed
send-tx
send-tx-node
send-tx-p2p
settings
sha160
sha256
sha512
stealth-decode
stealth-encode
stealth-public
stealth-secret
stealth-shared
tx-decode
tx-encode
uri-decode
uri-encode
validate-tx
watch-address
wif-to-ec
wif-to-public
wrap-decode
wrap-encode
```

さらに詳しい情報については、 [Bitcoin Explorer home page](#) と [Bitcoin Explorer user documentation](#) を参照してください。

## bxコマンドの使用例

Bitcoin Explorerコマンドによるいくつかの例を使って、キーとアドレスの実験をしてみましょう。

seed コマンドを使ってランダムな"シード"を生成してみましょう。この seed コマンドは、オペレーティングシステムの乱数生成器を使います。この生成したシードを `ec-new` コマンドに渡して、新しい秘密鍵を生成します。標準出力に出てきたものを `private_key` というファイルに記録しておきます。

```
$ bx seed | bx ec-new > private_key
$ cat private_key
73096ed11ab9f1db6135857958ece7d73ea7c30862145bcc4bbc7649075de474
```

次に、 `ec-to-public` コマンドを使って秘密鍵から公開鍵を生成してみましょう。 `private_key` を標準入力に渡してあげて標準出力に出てきたものを新しいファイル `public_key` に記録します。

```
$ bx ec-to-public < private_key > public_key
$ cat public_key
02fca46a6006a62dfd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

`public_key` を `ec-to-address` コマンドを使ってBitcoinアドレスに形式を変換できます。 `public_key` をコマンドに渡してあげます。

```
$ bx ec-to-address < public_key
17re1S4Q8ZHYP8Kw7xQad1Lr6XUzWUnkG
```

上記の方法で生成した鍵を使うとtype-0非決定性ウォレットができます。この意味は、それぞれの鍵が独立なシードから生成されているということです。Bitcoin ExplorerコマンドはまたBIP0032に従って決定的に鍵の生成ができます。この場合、"マスター"キーはシードから生成され、決定的に拡張されサブキーツリーを生成します。この形で作られたものがtype-2決定性ウォレットです。

最初に、 `seed` と `hd-new` コマンドを使って鍵の階層を導出する基礎として使われるマスターキーを生成します。

```
$ bx seed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1

$ bx hd-new < seed > master
$ cat master
xprv9s21ZrQH143K2BEhMYpNQoUvAgiEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBqkYo
WMMCRwzfuJk8RwF1SVEpAQ
```

`hd-private` コマンドを使って変数"account"にhardenedキーを入れ、accountに紐づく2つの秘密鍵の列(index=0, index=1)を作ります。

```

$ bx hd-private --hard < master > account
$ cat account
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwgcS8PYbg
oR2NWHelyvg4DhoEE68A1n

$ bx hd-private --index 0 < account
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVESuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F3DW
PkX7Tp1V1u97HKG1FJwUpU

$ bx hd-private --index 1 < account
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdsjsuMVERadbgDbziCRJru9n6tzEWR
ASVpEdrZrFidt1RDfn4yA3

```

次に、 `hd-public` コマンドを使って秘密鍵に対応した2つの公開鍵を含む列(index=0, index=1)を生成します。

```

$ bx hd-public --index 0 < account
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz
5wzaSW4FiGrseNDR4LKqTy

$ bx hd-public --index 1 < account
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQD
hohEcDFTEYmVYzwrD7Juf8

```

公開鍵はまた `hd-to-public` コマンドで導き出すこともできます。

```

$ bx hd-private --index 0 < account | bx hd-to-public
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz
5wzaSW4FiGrseNDR4LKqTy

$ bx hd-private --index 1 < account | bx hd-to-public
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQD
hohEcDFTEYmVYzwrD7Juf8

```

決定性チェーンの中で生成できる鍵の数には実用上制限はなく、全て単一のシードから導出されます。この手法は多くのウォレットで使用され、シードをバックアップしておき、シードからリストアしたりすることができる鍵を生成しています。

このシードは `mnemonic-encode` コマンドを使ってmnemonic codeの形でエンコードしておくことができます。

```

$ bx hd-mnemonic < seed > words
adore repeat vision worst especially veil inch woman cast recall dwell appreciate

```

このシードは逆に  
codeをデコードして得ることができます。

mnemonic-decode

コマンドを使ってmnemonic

```
$ bx mnemonic-decode < words  
eb68ee9f3df6bd4441a9feadec179ff1
```

mnemonicエンコードはシードを記録しておきやすく、また思い出しやすくするために作られます。

# Appendix A: Bitcoin改善提案(BIP, Bitcoin Improvement Proposals)

BIPはBitcoinコミュニティに情報を提供するための設計書であり、Bitcoinまたはその処理手順や環境に関する新しい機能を記述しているものです。

BIP0001 BIPの目的とガイドライン にある通り、BIPには3種類のタイプがあります。

## Standard BIP

ほとんど全てのBitcoin実装に影響を与える変更を記述しています。例えば、Bitcoinネットワークプロトコルに関する変更や、ブロックやトランザクションの検証ルールに関する変更、Bitcoinを使ったアプリケーションの相互運用に影響を与える変更や追加など。

## Informational BIP

Bitcoin設計に関する問題点、またはBitcoinコミュニティへの一般的なガイドラインや情報を記述しています。しかし、ここでは新しい機能は提案されません。Informational BIPではBitcoinコミュニティでの合意内容や推薦事項が必ずしも出てくるわけではないため、ユーザや実装者はinformational BIPを無視するかもしれませんし、従うかもしれません。

## Process BIP

Bitcoinでの処理手順を記述したり、その処理手順(または処理中のイベント)に関する変更を提案しています。Process BIPはStandard BIPに似ていますが、Bitcoinプロトコルそのもの以外の領域に対しても適用されます。ここで実装の提案をするかもしれませんが、コードをベースとしたものではありません。ここではBitcoinコミュニティ内での合意が必要な内容が頻繁に出てきます。Informational

BIPと違ってこれらは単なる推薦ではなく、一般的にユーザはこれらを無視することはできません。ここには例えば、処理手順やガイドライン、意思決定プロセスについての変更、Bitcoin開発で使うツールや環境についての変更が含まれます。

BIPは [GitHub](#) 上のバージョン管理リポジトリに記録されています。[BIPのスナップショット](#) は2014年秋の時点でのBIPを示しています。存在しているBIPやこれらの内容の最新の情報については信用できるリポジトリを調べてみてください。

Table 1. BIPのスナップショット

| BIP番号 | リンク   | タイトル          | 所有者        | 種類       | 状態                                |
|-------|---|---------------|------------|----------|-----------------------------------|
| 1     | <a href="https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki</a> | BIPの目的とガイドライン | Amir Taaki | Standard | アクティブ(完了させる目的で書かれたものではないBIPに付く状態) |

| BIP番号 | リンク   | タイトル                       | 所有者                           | 種類            | 状態    |
|-------|---|----------------------------|-------------------------------|---------------|-------|
| 10    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki</a> | マルチシグトランザクション配布            | Alan Reiner                   | Informational | 草案    |
| 11    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki</a> | M-of-N標準トランザクション           | Gavin Andresen                | Standard      | 承認済   |
| 12    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki</a> | OP_EVAL                    | Gavin Andresen                | Standard      | 取り下げ済 |
| 13    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki</a> | pay-to-script-hashでのアドレス形式 | Gavin Andresen                | Standard      | 最終版   |
| 14    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki</a> | プロトコルバージョンとユーザーエージェント      | Amir Taaki, Patrick Strateman | Standard      | 承認済   |
| 15    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki</a> | エイリアス                      | Amir Taaki                    | Standard      | 取り下げ済 |
| 16    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki</a> | Pay To Script Hash         | Gavin Andresen                | Standard      | 承認済   |

| BIP番号 | リンク   | タイトル                        | 所有者                          | 種類       | 状態    |
|-------|---|-----------------------------|------------------------------|----------|-------|
| 17    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki</a> | OP_CHECKHASHVERIFY (CHV)    | Luke Dashjr                  | Standard | 取り下げ済 |
| 18    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki</a> | hashScriptCheck             | Luke Dashjr                  | Standard | 草案    |
| 19    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki</a> | M-of-N 標準トランザクション(低SigOp)   | Luke Dashjr                  | Standard | 草案    |
| 20    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki</a> | URIスキーム                     | Luke Dashjr                  | Standard | 置き換え済 |
| 21    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki</a> | URIスキーム                     | Nils Schneider, Matt Corallo | Standard | 承認済   |
| 22    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki</a> | getblocktemplate - 基礎       | Luke Dashjr                  | Standard | 承認済   |
| 23    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki</a> | getblocktemplate - プールマイニング | Luke Dashjr                  | Standard | 承認済   |



| BIP番号 | リンク   | タイトル                               | 所有者            | 種類            | 状態  |
|-------|---|------------------------------------|----------------|---------------|-----|
| 30    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki</a> | 二重トランザクション                         | Pieter Wuille  | Standard      | 承認済 |
| 31    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki</a> | Pong message                       | Mike Hearn     | Standard      | 承認済 |
| 32    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki</a> | 階層的決定性ウォレット                        | Pieter Wuille  | Informational | 承認済 |
| 33    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki</a> | Stratizedノード                       | Amir Taaki     | Standard      | 草案  |
| 34    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki</a> | ブロックバージョン2, coinbaseトランザクション内ブロック高 | Gavin Andresen | Standard      | 承認済 |
| 35    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki</a> | mempool message                    | Jeff Garzik    | Standard      | 承認済 |
| 36    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki</a> | Custom Services                    | Stefan Thomas  | Standard      | 草案  |

| BIP番号 | リンク   | タイトル                    | 所有者                         | 種類       | 状態          |
|-------|---|-------------------------|-----------------------------|----------|-------------|
| 37    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki</a> | Bloom filtering         | Mike Hearn and Matt Corallo | Standard | 承認済         |
| 38    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki</a> | パスフレーズ保護秘密鍵             | Mike Caldwell               | Standard | 草案          |
| 39    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki</a> | 決定性キーを生成するmnemonic code | Slush                       | Standard | 草案          |
| 40    |   | Stratumワイヤープロトコル        | Slush                       | Standard | BIP番号割り当て済み |
| 41    |   | Stratumマイニングプロトコル       | Slush                       | Standard | BIP番号割り当て済み |
| 42    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki</a> | bitcoinの有限マネーサプライ       | Pieter Wuille               | Standard | 草案          |
| 43    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki</a> | 決定性ウォレットのpurposeフィールド   | Slush                       | Standard | 草案          |
| 44    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki</a> | 階層的ウォレットの複数アカウント階層構造    | Slush                       | Standard | 草案          |

| BIP番号 | リンク   | タイトル  | 所有者            | 種類            | 状態         |
|-------|---|---|----------------|---------------|------------|
| 50    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki</a> | 2013年3月に起きたブロックチェーンフォークに関する事後分析                       | Gavin Andresen | Informational | 草案         |
| 60    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki</a> | "version" messageのフィールド数の固定 (Relay-Transactionsフィールド) | Amir Taaki     | Standard      | 草案         |
| 61    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki</a> | "reject" P2P message                                  | Gavin Andresen | Standard      | 草案         |
| 62    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki</a> | トランザクション展性に対する対処                                      | Pieter Wuille  | Standard      | 草案         |
| 63    |   | ステルスアドレス  | Peter Todd     | Standard      | BIP番号割り当て済 |
| 64    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki</a> | getutxos message                                      | Mike Hearn     | Standard      | 草案         |
| 70    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki</a> | 支払いプロトコル  | Gavin Andresen | Standard      | 草案         |

| BIP番号 | リンク   | タイトル                         | 所有者            | 種類       | 状態 |
|-------|---|------------------------------|----------------|----------|----|
| 71    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki</a> | 支払いプロトコルMIMEタイプ              | Gavin Andresen | Standard | 草案 |
| 72    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki</a> | 支払いプロトコルURI                  | Gavin Andresen | Standard | 草案 |
| 73    | <a href="https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki">https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki</a> | 支払いリクエストURIに伴う"Accept"ヘッダの使用 | Stephen Pair   | Standard | 草案 |

# Appendix A: pycoin、ku、tx

Pythonライブラリ [pycoin](#) はBitcoinにおけるキーやトランザクションの操作をサポートしているPythonベースライブラリです。元々はRichard Kissによって書かれメンテナンスされているライブラリです。また、このライブラリは非標準トランザクションを扱うことができるScript言語をもサポートしています。

このpycoinライブラリはPython 2 (2.7.x)とPython 3 (after 3.3)を両方ともサポートしており、使いやすいコマンドラインツールkuとtxが付属しています。

## キーユーティリティ(KU)

コマンドラインユーティリティ `ku` ("キーユーティリティ")は、キーを操作するためのスイス・アーミーナイフのようなものです。これはBIP32キー、WIFおよびアドレス(Bitcoinとaltcoin)をサポートしており、以下にいくつかの使用例を示します。

GPGと `/dev/random` のデフォルトのエントロピー源を用いBIP32キーを作成:

```

$ ku create

input          : create
network       : Bitcoin
wallet key    : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJBsY7VqXUG7hipgdWaU
               m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version : xpub661MyMwAqRbcFpYYiuvZpKjKhJDZYAKWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
               DGcPfT56AMFeo8M8KPkFMfLUTvwjwb6WPv8rY65L2q8Hz
tree depth    : 0
fingerprint  : 9d9c6092
parent f'print : 00000000
child index   : 0
chain code    : 80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key   : yes
secret exponent :
112471538590155650688604752840386134637231974546906847202389294096567806844862
hex          : f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif         : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
uncompressed : 5KhoEavGNNH4GHKoy2PtU4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x :
76460638240546478364843397478278468101877117767873462127021560368290114016034
public pair y :
59807879657469774102040120298272207730921291736633247737077406753676825777701
x as hex     : a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
y as hex     : 843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
y parity     : odd
key pair as sec : 03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
uncompressed : 04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
               843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160      : 9d9c609247174ae323acfc96c852753fe3c8819d
uncompressed : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
uncompressed   : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM

```

パスフレーズからBIP32キーを作成:

### WARNING

この例で使われているパスフレーズは極めて予想しやすいものを使っています。このパスフレーズは絶対に使わないでください。

```
$ ku P:foo
```

```
input          : P:foo
network        : Bitcoin
wallet key     : xprv9s21ZrQH143K31AgNK5pyVvW23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
                ZoY5eSJMj2Vbyvi2hbmQnCuHBujZ2WXGTux1X2k9Krdtq
public version : xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtS
                VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
tree depth    : 0
fingerprint   : 5d353a2e
parent f'print : 00000000
child index   : 0
chain code    : 5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key   : yes
secret exponent :
65825730547097305716057160437970790220123864299761908948746835886007793998275
  hex         : 91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif          : L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
  uncompressed : 5JvNzA5vXDoKYJdw8SwwLHxUxaWvn9mDea6k1vRPCX7KLUVWa7W
public pair x :
81821982719381104061777349269130419024493616650993589394553404347774393168191
public pair y :
58994218069605424278320703250689780154785099509277691723126325051200459038290
  x as hex    : b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
  y as hex    : 826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
  y parity    : even
key pair as sec : 02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
  uncompressed : 04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
                826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160        : 5d353a2ecdb262477172852d57a3f11de0c19286
  uncompressed : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
  uncompressed  : 1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT
```

情報をJSON形式で取得:

```
$ ku P:foo -P -j
```

```
{
  "y_parity": "even",
  "public_pair_y_hex":
"826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
  "private_key": "no",
  "parent_fingerprint": "00000000",
  "tree_depth": "0",
  "network": "Bitcoin",
  "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT",
  "key_pair_as_sec_uncompressed":
"04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1
c1d3ae68541d9a04df54a2c48cc241c2983544de52",
  "public_pair_x_hex":
"b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f",
  "wallet_key":
"xpub661MyMwAqRbcFVF9ULcLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFi
dhjFj82pVShWu9curWmb2zy",
  "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
  "child_index": "0",
  "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
  "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
  "fingerprint": "5d353a2e",
  "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
  "input": "P:foo",
  "public_pair_x":
"81821982719381104061777349269130419024493616650993589394553404347774393168191",
  "public_pair_y":
"58994218069605424278320703250689780154785099509277691723126325051200459038290",
  "key_pair_as_sec":
"02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}
```

公開BIP32キー:

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFid
hjFj82pVShWu9curWmb2zy
```

サブキーを生成:



```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6i
ogWKxx6mefEw4M8EroLgKj
```

hardenedサブキー:

```
$ ku -w -s3/2H P:foo
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8iDk
BN45k67UKsJUXM1JfRcdn1
```

WIF:

```
$ ku -W P:foo
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

アドレス:

```
$ ku -a P:foo
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

サブキーのブランチを生成:

```
$ ku P:foo -s 0/0-5 -w
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3bhX
xKWB89Ggn2dzLcoJsuEdRK
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPNyAmvc77gUQvej5QHckc5Adtwwa28ACmANi9XhCrRvtFqQcUxt8r
UgFz3souMiDdWxJDZnQxxz
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sJx9JpCiJEiBwSMQyRxaN6srXUPbtj3PTxQFkZJAiwoUpmvtrxKZu
4zfsnr3pqqy2vthpkwuoVq
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtFXdiEY8UsRNJfqK6DAd
yZXoMvtaLHyWQx3FS4A9zw
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKd84Mm7Wtc2xmzFuKg7iYf7XFHkkSsaYKWKJbR5
4bnyAD9GzjUYbAYTtN4ruo
```

対応したアドレスの生成:

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkdHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb
116AXZc4bDVQrqmc inzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

対応したWIFの生成

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQ6xLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnMTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuVFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

BIP32文字列 (サブキー 0/3 に対応) を1つ選び、うまく動作するかチェック:

```
$ ku -W
xprv9xWkBDfyBXmZsA85GyWj9uYpYoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnMTDMrqemY8UF
$ ku -a
xprv9xWkBDfyBXmZsA85GyWj9uYpYoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmc inzu4aaPdrYqvuiBEK
```

思った通り、前に見たことがあるものが出てきました。

secret exponentから作成(秘密鍵を指定して作成):

\$ ku 1

```
input          : 1
network        : Bitcoin
secret exponent : 1
  hex          : 1
wif            : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn
  uncompressed : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf
public pair x  :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y  :
32670510020758816978083085130507043184471273380659243275938904335757337482424
  x as hex     : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  y as hex     : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
  y parity     : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  uncompressed : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
  uncompressed : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
  uncompressed  : 1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm
```

Litecoin/バージョン:

```
$ ku -nL 1
```

```
input          : 1
network        : Litecoin
secret exponent : 1
  hex          : 1
wif            : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdwUwyfRDeGZm76aUjV
  uncompressed : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x  :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y  :
32670510020758816978083085130507043184471273380659243275938904335757337482424
  x as hex      : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  y as hex      : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
  y parity      : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  uncompressed  : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                  483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
  uncompressed  : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6sUkLkQqDEVUnJ
  uncompressed  : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

Dogecoin WIF:

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qr ioRbQmjxac5TVoTtZuot
```

公開鍵ペア(テストネット上)から生成:

```
$ ku -nT
55066263022277343669578718895168534326250603453777594175500187360389116729240,even

input          :
550662630222773436695787188951685343262506034537775941755001873603
                89116729240,even
network        : Bitcoin testnet
public pair x  :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y  :
32670510020758816978083085130507043184471273380659243275938904335757337482424
x as hex       :
79be667ef9dcbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y as hex       :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity       : even
key pair as sec :
0279be667ef9dcbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
uncompressed   :
0479be667ef9dcbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
uncompressed   : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCybB6J1vRfbwM5hemdJz73FwDBC8r
uncompressed   : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

hash160から作成:

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6

input          : 751e76e8199196d454941c45d1b3a323f1433bd6
network        : Bitcoin
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

Dogecoinアドレスとして作成:

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6
```

```
input           : 751e76e8199196d454941c45d1b3a323f1433bd6
network         : Dogecoin
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

## トランザクションユーティリティ(TX)

コマンドラインユーティリティ `tx` は、人間が読める形でのトランザクション表示、pycoinのトランザクションキャッシュまたはウェブサービス(現在blockchain.infoとblockr.io、biteasy.comに対応)からのベーストランザクション取得、トランザクションのマージ、インプットまたはアウトプットの追加削除、トランザクションへの署名、ができます。

以下はいくつかの例です。

有名な "ピザ"トランザクション[PIZZA]を表示します:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache to
cache transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment variable
PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
        [-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
        [--remove-tx-in tx_in_index_to_delete]
        [--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
        [-b BITCOIND_URL] [-o path-to-output-file]
        argument [argument ...]
tx: error: can't find Tx with id
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
```

おっと！ウェブサービスの設定をしていませんでした。それを今からやりましょう。

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

これらの設定は自動的に行われていません。あなたがどのトランザクションに興味を持っているかという個人情報、コマンドラインツールがサードパーティのウェブサイトに漏洩しないようにするためです。もし気にしないのであれば、`.profile` にこれらの行を入れておき毎回設定しなくてもよいようにできます。

もう一度やってみましょう:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Version: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: (unknown) from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
including unspents in hex dump since transaction not fully signed
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e00000004a4
93046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e
9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901fffffffff010010a5d4e8000
0001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

** can't validate transaction as source transactions missing
```

最後の行にトランザクションの署名検証に関するメッセージが出ています。署名検証をするには元のトランザクション情報が必要なのです。このため、コマンドラインオプションとして `-a` を追加して、元のトランザクション情報を付加したトランザクションを取得してみましょう:

```
$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may be
incorrect
warning: transaction fee lower than (casually calculated) expected value of 0.1 mBTC,
transaction might not propogate
Version: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0 10000000.00000
mBTC sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input 10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees      0.00000 mBTC

010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a4
93046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e
9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901fffffffff010010a5d4e8000
0001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

all incoming transaction values validated
```

ここで、特定アドレスに対する未使用トランザクションアウトプットを見てみましょう。ブロック番号1には12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX へのcoinbaseトランザクションが見えます。 fetch\_unspent を使ってこのアドレスにある全てのbitcoinを見てみましょう:



```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/0/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/31/76a914119b098e2e9
80a229e139a9ed01a469e518e6f2688ac/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/86/76a914119b098e2e9
80a229e139a9ed01a469e518e6f2688ac/10000
a66ddd42f9f2491d3c336ce5527d45cc5c2163aaed3158f81dc054447f447a2/0/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/10000
ffd901679de65d4398de90cfe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcfd4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/5/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/1/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/100000
fd87f9adebb17f4ebb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/1
dfd0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/2000000

0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0/410496b538e853519c
726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141781e622947
21166bf621e73a82cbf2342c858eeac/5000000000
```

# Appendix A: トランザクション Script言語オペレータ、定数、シンボル

値をスタックの上にpushするでは、値をスタックの上にpushするオペレータをリストアップしています。

Table 1. 値をスタックの上にpushする

| シンボル              | 値(16進)       | 説明   |
|-------------------|--------------|--|
| OP_0 または OP_FALSE | 0x00         | 空配列がスタック上にpushされる                          |
| 1-75              | 0x01-0x4b    | 次のNバイトをスタック上にpushする、Nは1から75バイト             |
| OP_PUSHDATA1      | 0x4c         | 次のscriptバイトがNを含んでいれば、そのNバイトをスタック上にpush     |
| OP_PUSHDATA2      | 0x4d         | 次の2つのscriptバイトがNを含んでいれば、そのNバイトをスタック上にpush  |
| OP_PUSHDATA4      | 0x4e         | 次の4つのscriptバイトがNを含んでいれば、そのNバイトをスタック上にpush  |
| OP_1NEGATE        | 0x4f         | "-1"をスタック上にpush                            |
| OP_RESERVED       | 0x50         | 停止 - まだ実行されていない<br>OP_IF内でなければ不正なトランザクション  |
| OP_1 or OP_TRUE   | 0x51         | "1"をスタック上にpush                             |
| OP_2 to OP_16     | 0x52 to 0x60 | OP_Nに対して値"N"をスタック上にpush、例えば OP_2 は"2"をpush |

条件分岐制御では、条件分岐制御オペレータをリストアップしています。

Table 2. 条件分岐制御

| シンボル   | 値(16進) | 説明  |
|--------|--------|---|
| OP_NOP | 0x61   | 何もしない                                     |
| OP_VER | 0x62   | 停止 - まだ実行されていない<br>OP_IF内でなければ不正なトランザクション |
| OP_IF  | 0x63   | もしスタックの一番上に0がなければ次のステートメントを実行             |

| シンボル        | 値(16進) | 説明                                     |
|-------------|--------|--|
| OP_NOTIF    | 0x64   | もしスタックの一番上に0があれば次のステートメントを実行           |
| OP_VERIF    | 0x65   | 停止 - 不正なトランザクション                       |
| OP_VERNOTIF | 0x66   | 停止 - 不正なトランザクション                       |
| OP_ELSE     | 0x67   | 前のステートメントが実行されていない場合のみ実行               |
| OP_ENDIF    | 0x68   | OP_IF、OP_NOTIF、OP_ELSEブロックを終わらせる       |
| OP_VERIFY   | 0x69   | スタックの一番上をチェックし、真でなければ停止しトランザクションを無効化する |
| OP_RETURN   | 0x6a   | 停止しトランザクションを無効化する                      |

スタックオペレータでは、スタックを操作するためのオペレータをリストアップしています。

Table 3. スタックオペレータ

| シンボル            | 値(16進) | 説明                                    |
|-----------------|--------|---------------------------------------|
| OP_TOALTSTACK   | 0x6b   | スタックから一番上のアイテムをpopし、代替のスタックにpush      |
| OP_FROMALTSTACK | 0x6c   | 代替のスタックから一番上のアイテムをpopし、スタックにpush      |
| OP_2DROP        | 0x6d   | スタックの一番上から2つのアイテムをpop                 |
| OP_2DUP         | 0x6e   | スタックの一番上にある2つのアイテムを複製                 |
| OP_3DUP         | 0x6f   | スタックの一番上にある3つのアイテムを複製                 |
| OP_2OVER        | 0x70   | スタックの中の一番上から3番目と4番目のアイテムをスタックの一番上にコピー |
| OP_2ROT         | 0x71   | スタックの中の一番上から5番目と6番目のアイテムをスタックの一番上に移動  |
| OP_2SWAP        | 0x72   | スタックの一番上の2つのアイテムペアを交換                 |
| OP_IFDUP        | 0x73   | もし0でなければ、スタックの中の一番上のアイテムを複製           |
| OP_DEPTH        | 0x74   | スタック上のアイテム数をカウントし、カウント数をpush          |

| シンボル    | 値(16進) | 説明                                      |
|---------|--------|---|
| OP_DROP | 0x75   | スタックの中の一番上のアイテムをpop                     |
| OP_DUP  | 0x76   | スタックの中の一番上のアイテムを複製                      |
| OP_NIP  | 0x77   | スタックの中の二番目のアイテムをpop                     |
| OP_OVER | 0x78   | スタックの中の二番目のアイテムをコピーし、それをスタックの一番上にpush   |
| OP_PICK | 0x79   | スタックの一番上から値Nをpopし、N番目のアイテムをスタックの一番上にコピー |
| OP_ROLL | 0x7a   | スタックの一番上から値Nをpopし、N番目のアイテムをスタックの一番上に移動  |
| OP_ROT  | 0x7b   | スタックの中の一番上の3つのアイテムを回転                   |
| OP_SWAP | 0x7c   | スタックの中の一番上の3つのアイテムを交換                   |
| OP_TUCK | 0x7d   | 一番上のアイテムをコピーし、一番上と二番目の間にそれを挿入           |

文字列結合オペレータでは、文字列オペレータをリストアップしています。

Table 4. 文字列結合オペレータ

| シンボル      | 値(16進) | 説明                     |
|-----------|--------|------------------------|
| OP_CAT    | 0x7e   | 使用不可(一番上の2つのアイテムを結合)   |
| OP_SUBSTR | 0x7f   | 使用不可(部分文字列を返却)         |
| OP_LEFT   | 0x80   | 使用不可(左側部分文字列を返却)       |
| OP_RIGHT  | 0x81   | 使用不可(右側部分文字列を返却)       |
| OP_SIZE   | 0x82   | 一番上の文字列の長さを計算し、結果をpush |

2進数算術と条件では、2進数算術およびブーリアン論理オペレータをリストアップしています。

Table 5. 2進数算術と条件

| シンボル      | 値(16進) | 説明                    |
|-----------|--------|-----------------------|
| OP_INVERT | 0x83   | 使用不可(一番上のアイテムのbitを反転) |

| シンボル                  | 値(16進) | 説明   |
|-----------------------|--------|--|
| <i>OP_AND</i>         | 0x84   | 使用不可(一番上の2つのアイテムのANDをとる)                                     |
| <i>OP_OR</i>          | 0x85   | 使用不可(一番上の2つのアイテムのORをとる)                                      |
| <i>OP_XOR</i>         | 0x86   | 使用不可(一番上の2つのアイテムのXORをとる)                                     |
| <i>OP_EQUAL</i>       | 0x87   | もし一番上の2つのアイテムが完全に等しければ真(1)をpushし、それ以外なら偽(0)をpush             |
| <i>OP_EQUALVERIFY</i> | 0x88   | <i>OP_EQUAL</i> と同じですが、もし真でなければ停止のためあとで <i>OP_VERIFY</i> を実行 |
| <i>OP_RESERVED1</i>   | 0x89   | 停止 - まだ実行されていない<br><i>OP_IF</i> 内でなければ不正なトランザクション            |
| <i>OP_RESERVED2</i>   | 0x8a   | 停止 - まだ実行されていない<br><i>OP_IF</i> 内でなければ不正なトランザクション            |

数値的オペレータでは、数値的(算術的)オペレータをリストアップしています。

Table 6. 数値的オペレータ

| シンボル                | 値(16進) | 説明                                      |
|---------------------|--------|---|
| <i>OP_1ADD</i>      | 0x8b   | 一番上のアイテムに1を足す                           |
| <i>OP_1SUB</i>      | 0x8c   | 一番上のアイテムから1を引く                          |
| <i>OP_2MUL</i>      | 0x8d   | 使用不可(一番上のアイテムに2を掛ける)                    |
| <i>OP_2DIV</i>      | 0x8e   | 使用不可(一番上のアイテムを2で割る)                     |
| <i>OP_NEGATE</i>    | 0x8f   | 一番上のアイテムの符号を反転                          |
| <i>OP_ABS</i>       | 0x90   | 一番上のアイテムの符号をプラスに変更                      |
| <i>OP_NOT</i>       | 0x91   | もし一番上のアイテムが0または1ならブーリアンとして反転、それ以外なら0を返却 |
| <i>OP_0NOTEQUAL</i> | 0x92   | もし一番上のアイテムが0なら0を返却、それ以外なら1を返却           |

| シンボル                  | 値(16進) | 説明                                       |
|-----------------------|--------|--|
| OP_ADD                | 0x93   | 一番上の2つのアイテムをpopし、2つを加え合わせた結果をpush        |
| OP_SUB                | 0x94   | 一番上の2つのアイテムをpopし、2番目から1番目を引いた結果をpush     |
| OP_MUL                | 0x95   | 使用不可(一番上の2つのアイテムを掛け合わせる)                 |
| OP_DIV                | 0x96   | 使用不可(2番目のアイテムを1番目のアイテムで割る)               |
| OP_MOD                | 0x97   | 使用不可(2番目のアイテムを1番目のアイテムで割ったときの余り)         |
| OP_LSHIFT             | 0x98   | 使用不可(2番目のアイテムを最初のアイテムのbit数だけ左にシフト)       |
| OP_RSHIFT             | 0x99   | 使用不可(2番目のアイテムを最初のアイテムのbit数だけ右にシフト)       |
| OP_BOOLAND            | 0x9a   | 一番上の2つのアイテムのANDをとる                       |
| OP_BOOLOR             | 0x9b   | 一番上の2つのアイテムのORをとる                        |
| OP_NUMEQUAL           | 0x9c   | 一番上の2つのアイテムが同じ数値であれば真を返却                 |
| OP_NUMEQUALVERIFY     | 0x9d   | NUMEQUALと同じだが、もし真でなければ停止のためにOP_VERIFYを実行 |
| OP_NUMNOTEQUAL        | 0x9e   | 一番上の2つのアイテムが同じ数値でなければ真を返却                |
| OP_LESSTHAN           | 0x9f   | 2番目のアイテムが1番目のアイテムよりも小さい場合真を返却            |
| OP_GREATERTHAN        | 0xa0   | もし2番目のアイテムが1番目のアイテムよりも大きい場合真を返却          |
| OP_LESSTHANOREQUAL    | 0xa1   | もし2番目のアイテムが1番目のアイテムよりも小さいか等しければ真を返却      |
| OP_GREATERTHANOREQUAL | 0xa2   | もし2番目のアイテムが1番目のアイテムよりも大きいか等しければ真を返却      |
| OP_MIN                | 0xa3   | 1番目と2番目のアイテムのうちより小さいアイテムを返却              |
| OP_MAX                | 0xa4   | 1番目と2番目のアイテムのうちより大きいアイテムを返却              |

| シンボル      | 値(16進) | 説明                                   |
|-----------|--------|--------------------------------------|
| OP_WITHIN | 0xa5   | もし3番目のアイテムが2番目と1番目の間(または等しい)であれば真を返却 |

暗号的オペレータとハッシュ化オペレータでは、暗号的関数オペレータをリストアップしています。

Table 7. 暗号的オペレータとハッシュ化オペレータ

| シンボル                   | 値(16進) | 説明  |
|------------------------|--------|---|
| OP_RIPEMD160           | 0xa6   | 1番目のアイテムのRIPEMD160ハッシュを返却   |
| OP_SHA1                | 0xa7   | 1番目のアイテムのSHA1ハッシュを返却  |
| OP_SHA256              | 0xa8   | 1番目のアイテムのSHA256ハッシュを返却  |
| OP_HASH160             | 0xa9   | 1番目のアイテムのRIPEMD160(SHA256(x))ハッシュを返却  |
| OP_HASH256             | 0xaa   | 1番目のアイテムのSHA256(SHA256(x))ハッシュを返却   |
| OP_CODESEPARATOR       | 0xab   | 署名チェック済みのデータの最初に印を置く  |
| OP_CHECKSIG            | 0xac   | 公開鍵と署名をpopしたのち、トランザクションのハッシュ化データに対して署名が有効であるかを検証し、有効であれば真を返却  |
| OP_CHECKSIGVERIFY      | 0xad   | CHECKSIGと同じだが、もし真でなければ停止のためにOP_VERIFYを実行  |
| OP_CHECKMULTISIG       | 0xae   | 与えられたそれぞれの署名と公開鍵のペアに対してCHECKSIGを実行。結果は全て真でなければならない。この実装には余分な値をpopしてしまうというバグがあり、回避策としてOP_NOPをOP_CHECKMULTISIGの前に置く |
| OP_CHECKMULTISIGVERIFY | 0xaf   | CHECKMULTISIGと同じだが、もし真でなければ停止のためにOP_VERIFYを実行   |

非オペレータでは、非オペレータシンボルをリストアップしています。

Table 8. 非オペレータ

| シンボル             | 値(16進)    | 説明          |
|------------------|-----------|-------------|
| OP_NOP1-OP_NOP10 | 0xb0-0xb9 | 何もしない、無視される |

script/パーサの内部使用のために予約されているOPコードでは、内部scriptパーサによって使用されるために予約されているオペレータコードをリストアップしています。

Table 9. script/パーサの内部使用のために予約されているOPコード

| シンボル             | 値(16進) | 説明                      |
|------------------|--------|-------------------------|
| OP_SMALLDATA     | 0xf9   | 小さいデータフィールドを表す          |
| OP_SMALLINTEGER  | 0xfa   | 小さい整数データフィールドを表す        |
| OP_PUBKEYS       | 0xfb   | 公開鍵フィールド(複数)を表す         |
| OP_PUBKEYHASH    | 0xfd   | 公開鍵ハッシュフィールドを表す         |
| OP_PUBKEY        | 0xfe   | 公開鍵フィールドを表す             |
| OP_INVALIDOPCODE | 0xff   | 現在割り当てられていない任意のOPコードを表す |