第 3 章

Bitcoin ノードとの通信

本章では 2 章で受け取った Bitcoin の情報を Bitcoin ネットワークから取得するためのノードへの通信方法について解説します。Bitcoin ノードを見つける方法である Seed Node と DNS Seed について解説し、実際に通信するためのプロトコルについて紹介します。

3.1 最初の Bitcoin ノードの見つけ方

Bitcoin ネットワークには中央管理者がいないため特定の ip にアクセスすれば接続できるというものではありません。そのため、有志で公開している ip アドレスに対して接続しに行くのが普通です。ip アドレスを知る方法は DNS Seed と Seed Node の 2 つがあるので順に紹介します。

3.1.1 DNS Seed

DNS Seed は稼働している Bitcoin ノードの IP アドレスリストを返してくれます。DNS はソースコードに記載されおり、その中から利用できるノードに対して接続を行います。

btcd で使われる DNS Seed は下記設定ファイル[*1]に定義されており、次のようなものがあります。

1. testnet-seed.bitcoin.jonasschnelli.ch
2. testnet-seed.bitcoin.schildbach.de
3. seed.tbtc.petertodd.org

[*1] https://github.com/btcsuite/btcd/blob/master/chaincfg/params.go#L408

4. testnet-seed.bluematt.me

設定ファイルに記載されている DNS に対して dig コマンドを実行すると稼働している ip アドレスリストが返ってきます。

```
$ dig +short testnet-seed.bitcoin.jonasschnelli.ch
63.141.242.58
13.124.186.174
167.179.98.113
159.65.121.59
118.167.26.167
212.227.82.212
```

3.1.2 Seed Node

基本的にノードは DNS Seed から ip アドレスリストを取得し、接続を試みますが DNS が応答しない場合接続先が分からなくなってしまいます。そこで予め接続可能な ip アドレスリストをソースコード上にハードコードしておきます。そのノードのこと Seed Node と言います。

btcd は Seed Node を設定していないため bitcoind[*2]で使われている Seed Node[*3]を紹介します。ip アドレスリストは nodes_main.txt と nodes_test.txt にそれぞれメインネットとテストネットで稼働している ip アドレスリストが記載されています。全てのノードが正常に稼働しているかは接続してみるまで分からないという不便さはありますが、これが管理者がいないネットワークということを実感できるかと思います。

▼リスト 3.11　https://github.com/bitcoin/bitcoin/tree/master/contrib/seeds/nodes_main.txt

```
2.24.141.73:8333
5.8.18.29:8333
5.43.228.99:8333
5.145.10.122:8333
5.166.35.47:8333
```

3.1.3 ノードとの接続

DNS Seed を使って実際に接続を行います。TCP 接続は net パッケージの Dial 関数を使います。ノードへの TCP 接続はこれだけで完了しますが、データをやり取りするには

[*2] https://github.com/bitcoin/bitcoin
[*3] https://github.com/bitcoin/bitcoin/tree/master/contrib/seeds

ハンドシェイク処理が必要になります。次は実際にデータをやり取りするための方法について解説します。

▼リスト 3.11　pkg/protocol/network/client.go

```
type Client struct {
  Conn net.Conn
}

func NewClient(dns string) *Client {
  conn, err := net.Dial("tcp", dns)
  if err != nil {
    log.Fatal(err)
  }
  return &Client{Conn: conn}
}
```

3.2　通信プロトコル

Bitcoin ノードの通信プロトコルは Bitcoin Wiki の Protocol Documentation[*4]にまとめられています。

3.2.1　Message Protocol

メッセージの基本構成は送信するメッセージ情報をまとめた Message Header とメッセージ本文である Payload で構築され、次の表のようになります。

▼表 3.1　Message Protocol

名称	バイト数	フォーマット	内容
マジックナンバー	4	VarInt	ネットワークを示すマジックバイト
コマンド	12	[]byte	メッセージタイプを示すバイト列
ペイロードのサイズ	4	VarInt	ピアに送信するペイロードのバイト数
ペイロードのチェックサム	4	[4]byte	ペイロードを SHA256 で 2 回ハッシュした先頭 4 バイトをチェックサムとして使用
ペイロード	可変	VarInt	送信するメッセージ

コマンドは送信するメッセージの名称をバイト列にしたものです。次にペイロードの長さ、ペイロードをダブルハッシュしたチェックサムと続きます。最後に実際のメッセージ

[*4] https://en.bitcoin.it/wiki/Protocol_documentation

をバイト列にしたペイロードがきます。

マジックナンバーはメインネットかテストネットを判別する値でそれぞれ次のように定義されています。

▼表 3.2　マジックナンバー

ネットワーク	マジックナンバー
メインネット	0xD9B4BEF9
テストネット	0xDAB5BFFA

ノードにはこれらをまとめてバイト列としたものをメッセージとして送信します。

3.2.2　プロトコルで使用する基本的な型（VarInt/VarStr）

ピアとの通信で使用する基本的な型である VarInt と VarStr について解説します。

VarInt

VarInt は Variable Length Integer の略で可変長整数を定義しています。VarInt はデータ量を節約するために使われており、整数値の大きさによりデータを格納するバイト列の長さを変えます。整数値はリトルエンディアンとして格納されます。Bitcoin では基本的に全ての数値をリトルエンディアンで扱いますが、ポート番号のみビッグエンディアンとなります。

▼表 3.3　VarInt Protocol

数値	バイト数	フォーマット
< 0xfd（253）	1	数値のみ
<= 0xffff（65535）	3	先頭に fd、後ろにリトルエンディアンの数値
<= 0xffff ffff	5	先頭に fe、後ろにリトルエンディアンの数値
0xffff ffff <	9	先頭に ff、後ろにリトルエンディアンの数値

例えば 253（0xfd）以上、65535（0xffff）以下の数値の場合、先頭に 1 バイトの fd、数値を格納する場所に 3 バイトの計 4 バイトを使って数値を表現します。Bitcoin プロトコルではこの VarInt がよく登場するのではじめに定義しておきます。

▼リスト 3.11　pkg/protocol/common/varint.go

```
type VarInt struct {
  Data uint64
}

func NewVarInt(u uint64) *VarInt {
  return &VarInt{
    Data: u,
  }
}

func (v *VarInt) Encode() []byte {
  if v.Data < 0xfd {
    return []byte{byte(v.Data)}
  }
  if v.Data <= 0xffff {
    b := make([]byte, 3)
    b[0] = byte(0xfd)
    binary.LittleEndian.PutUint16(b[1:], uint16(v.Data))
    return b
  }
  if v.Data <= 0xffffffff {
    b := make([]byte, 5)
    b[0] = byte(0xfe)
    binary.LittleEndian.PutUint32(b[1:], uint32(v.Data))
    return b
  }
  if v.Data <= 0xffffffffffffffff {
    b := make([]byte, 9)
    b[0] = byte(0xff)
    binary.LittleEndian.PutUint64(b[1:], v.Data)
    return b
  }
  return []byte{byte(v.Data)}
}
```

VarStr

VarInt の String 版で、先頭に文字列の長さを表す VarInt を置き、その後ろに文字列を格納します。

▼表 3.4　VarStr Protocol

バイト数	フォーマット	内容
1 以上	VarInt	文字列の長さ
0 以上	[]byte	文字列

▼リスト 3.11　pkg/protocol/common/varstr.go

```
type VarStr struct {
  Length *VarInt
  Data   []byte
}

func NewVarStr(b []byte) *VarStr {
  len := uint64(len(b))
  length := NewVarInt(len)
  return &VarStr{
    Length: length,
    Data:   b,
  }
}

// Encode encode VarStr to byte slice.
func (s *VarStr) Encode() []byte {
  return bytes.Join([][]byte{
    s.Length.Encode(),
    s.Data,
  },
    []byte{},
  )
}
```

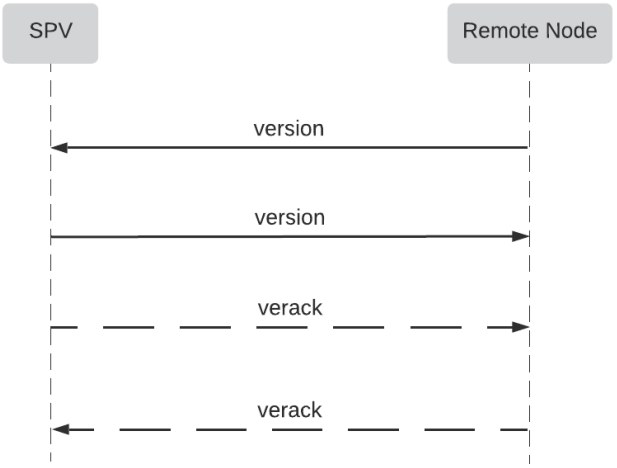
3.3 ノードとのハンドシェイク（Version/Verack）

ノードと通信を始めるにはお互いの対応プロトコルを確認する必要があります。この確認をハンドシェイク通信[*5]で行います。ハンドシェイク通信ではお互いのノードが対応しているバージョン情報などを送信します。送信するメッセージは Version メッセージです。

Version メッセージには自身のノードの対応プロトコルを記載し、相手ノードが対応可能であれば Verack というメッセージが返ってきます。相手ノードも同様に Version メッセージを送信してくるので、メッセージをハンドリングし、対応可能であれば Verack メッセージを返します。お互いが Verack メッセージを返すことでハンドシェイクが完了し、以降は自由にメッセージのやり取りを行うことができます。

ハンドシェイクの処理の流れは次のようになります。

[*5] https://en.bitcoin.it/wiki/Version_Handshake



▲図 3.1　ハンドシェイク通信

Version

バージョンメッセージの中身は次のようになっています。

▼表 3.5　Version Protocol

名称	バイト数	フォーマット	内容
バージョン	4	uint32	プロトコルバージョンを指定
サービス	8	uint64	サービス
タイムスタンプ	8	uint64	タイムスタンプ
通信先ネットワークアドレス	26	NetworkAddr	接続ノードの IP アドレス
自身のネットワークアドレス	26	NetworkAddr	自身のノードの IP アドレス
ノンス	8	uint64	ランダムな数値
ユーザーエージェント	可変	VarStr	ノードの表示名、空文字でも可
取得開始ブロック	4	uint32	取得開始ブロック高
リレーフラグ	1	bool	false の場合、フィルタコマンドを 受信するまでデータが送られてこない

バージョン情報は Bitcoin Developer Reference[*6]にまとまっています。このあと説明

[*6] https://bitcoin.org/en/developer-reference#constants-and-defaults

する filterload メッセージを使うには 70001 以上を指定するようにしてください。執筆時点での最新バージョンは 70015 です。サービスはノードがサポートしている機能を設定しますが、今回は 1 を設定しておけば問題ないです。サービスの詳細は Bitcoin Wiki の Version 項目[*7]に記載されています。

NetworkAddr の中身は次のようになっており、通信している IP とポートを指定します。ローカルノードの IP は 127.0.0.1 を設定しておきます。

▼表 3.6　NetworkAddr Protocol

名称	バイト数	フォーマット	内容
サービス	16	uint64	version メッセージで指定している service と同じ
IP	8	[]byte	IP アドレス
ポート	2	uint16	通信ポート

バージョンメッセージのソースコードは次のようになります。

▼リスト 3.11　pkg/protocol/message/version.go

```
type Version struct {
  Version     uint32
  Services    uint64
  Timestamp   uint64
  AddrRecv    *common.NetworkAddress
  AddrFrom    *common.NetworkAddress
  Nonce       uint64
  UserAgent   *common.VarStr
  StartHeight uint32
  Relay       bool
}

func NewVersion() protocol.Message {
  addrFrom := &common.NetworkAddress{
    Services: uint64(1),
    IP: [16]byte{
      0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0xFF, 0xFF,
      0x7F, 0x00, 0x00, 0x01,
    },
    Port: 18333,
  }
  return &Version{
    Version:     uint32(70015),
    Services:    uint64(1),
    Timestamp:   uint64(time.Now().Unix()),
    AddrRecv:    addrFrom,
    AddrFrom:    addrFrom,
```

[*7] https://bitcoin.org/en/developer-reference#version

```
    Nonce:       uint64(0),
    UserAgent:   common.NewVarStr([]byte("")),
    StartHeight: uint32(0),
    Relay:       false, // falseの場合、ブロードキャストするトランザクションはフィルタ
コマンド（load,add,clear）を受信するまでアナウンスされない
  }
}
```

NetworkAddress は次のようになります。

▼リスト 3.11　pkg/protocol/message/networkaddress.go

```
type NetworkAddress struct {
  Services uint64
  IP       [16]byte
  Port     uint16
}

func (addr *NetworkAddress) Encode() [26]byte {
  var b [26]byte
  binary.LittleEndian.PutUint64(b[0:8], addr.Services)
  copy(b[8:24], addr.IP[:])
  binary.BigEndian.PutUint16(b[24:26], addr.Port)
  return b
}
```

Verack

Verack メッセージは Header 情報だけで payload は空になります。Version メッセージに対して了承する場合、Command を Verack として空のメッセージを返すだけです。

▼リスト 3.11　pkg/protocol/message/verack.go

```
func NewVerack() protocol.Message {
  return &Verack{}
}
```

3.3.1　ハンドシェイク処理

ハンドシェイクを行うためまずはメッセージの送信と受信の処理を実装します。network パッケージがノードの役割で言うところのネットワークルーティング機能を担います。

▼リスト 3.11　pkg/network/client.go

```
type Client struct {
  Conn net.Conn
}

func NewClient(address string) *Client {
  conn, err := net.Dial("tcp", address)
  if err != nil {
    log.Fatal(err)
  }
  return &Client{Conn: conn}
}

func (c *Client) SendMessage(msg protocol.Message) (int, error) {
  message := common.NewMessage(msg.Command(), msg.Encode())
  log.Printf("send    : %s", string(message.Command[:]))
  return c.Conn.Write(message.Encode())
}

func (c *Client) ReceiveMessage(size uint32) ([]byte, error) {
  buf := make([]byte, size)
  _, err := c.Conn.Read(buf)
  if err != nil {
    return nil, err
  }
  return buf, nil
}
```

SendMessage で送信するメッセージはインターフェースで定義されており、各メッセージのコマンド名とエンコード結果を返すメソッドを実装するようにしています。メッセージインターフェースは次のようになります。

▼リスト 3.11　pkg/protocol/message.go

```
type Message interface {
  Command() [12]byte
  Encode() []byte
}
```

では実際にハンドシェイク処理の一連の流れを載せます。ハンドシェイク処理は network パッケージを使用して spv.go で行います。

▼リスト 3.11　internal/spv/spv.go

```
func (s *SPV) Handshake() error {
  v := message.NewVersion()
  _, err := s.Client.SendMessage(v)
  if err != nil {
    return err
  }

  var recvVerack, sendVerack bool
```

```
  for {
    if recvVerack && sendVerack {
      log.Printf("success handshake")
      return nil
    }
    buf, err := s.Client.ReceiveMessage(common.MessageLen)
    if err != nil {
      log.Printf("handshake Receive message error: %+v", err)
      return err
    }

    var header [24]byte
    copy(header[:], buf)
    msg := common.DecodeMessageHeader(header)
    _, err = s.Client.ReceiveMessage(msg.Length)
    if err != nil {
      return err
    }

    if bytes.HasPrefix(msg.Command[:], []byte("verack")) {
      recvVerack = true
    } else if bytes.HasPrefix(msg.Command[:], []byte("version")) {
      _, err := s.Client.SendMessage(message.NewVerack())
      if err != nil {
        return err
      }
      sendVerack = true
    } else {
      log.Printf("receive : other")
    }
  }
}
```

Version メッセージをどちらが先に送るかという決まりはないため一番最初に Version メッセージを送信しておきます。ReceiveMessage がメッセージを受け取る処理になりますが、必ず Version メッセージが一番最初に送られてきます。Command の内容が Version であればこちらから空の Verack メッセージを送信します。こちらが送信した Version メッセージの内容に対して相手が了承し Verack メッセージを送信してくればハンドシェイクが完了となります。

3.3.2 ハートビート通信（keepalive）

接続したピアとの通信が有効かどうか定期的にハートビート通信が行われます。ピアからは 2 分おきくらいに ping メッセージが送られてくるので pong メッセージを返すだけです。その際、ping メッセージには毎回ランダムな nonce 値が入っているので、それを pong メッセージに含めて送り返すことで接続を維持できます。こちらから ping メッセージを送らなくとも送られてくる ping メッセージに対して pong メッセージを返すだけでも接続は維持されます。

▼表 3.7　Ping Protocol

名称	バイト数	フォーマット	内容
nonce	8	uint64	8 バイトの乱数

▼表 3.8　Pong Protocol

名称	バイト数	フォーマット	内容
nonce	8	uint64	ping メッセージで受け取った nonce

次が ping メッセージをハンドリングする処理になります。やっていることは非常にシンプルで ping メッセージ含まれる nonce を pong メッセージ詰めて送信するだけです。

▼リスト 3.11　internal/spv/spv.go

```
func (s *SPV) MessageHandler() error {
  var transaction *message.Tx
  for {
    buf, err := s.Client.ReceiveMessage(common.MessageHeaderLength)
    if err != nil {
      log.Printf("ReceiveMessage: %+v", err)
      return err
    }
    var header [24]byte
    copy(header[:], buf)
    msg := common.DecodeMessageHeader(header)
    b, err := s.Client.ReceiveMessage(msg.Length)
    if err != nil {
      return err
    }
    if !common.IsTestnet3(msg.Magic) {
      log.Printf("not testnet3")
      continue
    }
    if !common.IsValidChecksum(msg.Checksum, b) {
      log.Printf("invalid checksum")
      continue
    }

    if bytes.HasPrefix(msg.Command[:], []byte("ping")) {
      ping := message.DecodePing(b)
      pong := message.NewPong(ping.Nonce)
      s.Client.SendMessage(pong)
    } else {
      log.Printf("receive : other")
    }
  }
}
```