
Working with TCP Sockets
なるほど
TCP
ソケット
Rubyで学ぶソケットプログラミングの基礎
JESSE STORIMER　著
島田浩二　訳

なるほど TCP ソケット

Ruby で学ぶソケットプログラミングの基礎

Jesse Storimer　著
島田 浩二　訳

2024-09-23 版　　発行

はじめに

ソケットはデジタル世界へとつながっている。

話はコンピュータの黎明期にさかのぼる。当時、コンピュータは科学者の使う道具だった。コンピュータは数学的な計算やシミュレーションなどに利用されていた。

時が過ぎ、コンピュータが人々をつなげられるようになると、科学者でない人々もコンピュータに関心を持つようになった。そして、今日では科学者が使うよりもはるかに多くの台数のコンピュータが、一般人によって使われている。どこの誰とでも情報共有や連絡ができるようになったことで、コンピュータは一般人の興味をひくものになった。

それを実現したのはネットワークプログラミングだ。もっと具体的に言うなら、特定のソケットプログラミング API の普及だ。本書を読んでいるのなら、おそらくあなたはオンラインで人々とつながり、コンピュータ同士が接続されていることを前提に構築されたテクノロジーと共に毎日を過ごしていることだろう。

ネットワークプログラミングとは、突き詰めると共有と通信のことだ。本書は、あなたがネットワークプログラミングの基本的な仕組みを理解して、その本質により良く貢献できるようにするためにある。

私の話

ソケットの世界に初めて触れたときのことは、今でも覚えている。それは、あんまりいいものではなかった。

Web エンジニアだった私は、当時あらゆる種類の HTTP API を扱っていたし、REST や JSON のような高度な概念の扱いにも慣れていた。

そんな折、ドメインレジストラの API を扱う必要が生じた。

API ドキュメントを開いて、私は衝撃を受けた。ドキュメントには、いくつかのプライベートホスト名とランダムなポートを指定して TCP ソケットを開くよう書かれていた。この API は Twitter API のようには動きそうになかった！

API は TCP ソケットを要求してきただけでなく、データを JSON や XML などの形式にエンコードもしていなかった。API には遵守すべき独自の回線プロトコルが定められていた。ソケット経由で厳密に書式化されたテキスト行を送り、その次に空行、続けてキーと値のペアを引数として送っていき、最後にリクエストが完了したことを示す空行を 2 つ送る必要があった。

その後には同様の方法で応答を読み込む必要があった。そして思った。「いったい何……」

同僚にこのことを伝えると、彼は私の不安をわかってくれた。彼もまたこうした API を使った経験はなかった。そして、彼はすぐに警告してくれた。「C のプログラムでならソケットを使ったことがあるんだ。注意した方がよいよ。終了前には必ずソケットを閉じる必要があるからね。さもないと、それらはずっと開きっぱなしになるんだ。プログラムが終了したあとにそいつらを閉じるのは難しいよ」

なんだって！？ ずっと開きっぱなし？ プロトコル？ ポート？ 私はあぜんとした。

それから、別の同僚が API ドキュメントをちらっと見て、言った。「マジ

で？ ソケットを扱う方法がわからないって？ Web ページを読むたびに毎回ソケットを開いているって知ってるだろ？ どんな仕組みになっているのかは知っておかないといけないよ」

私はチャレンジだと考えて、この仕事に取り組んだ。最初は概念を整理するだけでも大変だった。けれど、とにかく続けた。たくさんミスもした。そして、最終的にはうまく扱うことができた。おかげで、ソケットを扱うことに関して、私はマシなプログラマになれたと思う。この経験は自分の仕事の背景にある技術への理解を深めてくれた。とても良い気分だ。

本書では、当時私が感じたような痛みを受けることなく、一連の技術への深い理解をあなたに届けたいと願っている。

対象読者

対象読者は、Unix や Unix ライクな システム で開発をしている Ruby 技術者だ。

ある程度 Ruby を知っていることを前提としているので、Ruby の基礎については特に触れない。逆に、ネットワークプログラミングの考え方についての知識を持っている必要はない。ネットプワークプログラミングについては基礎から触れていく。

すべてのコードは Ruby 1.9 を使って書かれており、それ以前のバージョンでは動作確認は行っていない。

本書の構成

本書は 3 部構成となっている。

第 1 部では、ソケットプログラミングの基礎について紹介する。ソケットを作り、接続し、データを共有する方法を学ぶ。

第 2 部では、ソケットプログラミングの応用的なトピックを紹介する。ここでは「Hello, World」スタイルのソケットプログラミングから先に進むた

めに必要な内容を扱う。

第 3 部では、ここまででやってきたことすべてを「現実世界」のシナリオに適用する。ここではソケットを一旦忘れ、ネットワークプログラムに並行処理を適用する方法を示す。いくつかのアーキテクチャパターンを実装し、同じ問題を解決した場合の比較を行う。

バークレーソケット API

本書はバークレーソケット API とその使い方に焦点をあてている。バークレーソケット API は、1983 年 BSD オペレーティングシステムのバージョン 4.2 で初めて登場した。それはその後で新たに提案された伝送制御プロトコル（TCP）の最初の実装だった。

バークレーソケット API は時の試練に耐え続けている。バークレーソケット API は、本書であなたが触れることになる API だ。モダンなプログラミング言語のほとんどでサポートされている API でもある。そして、この API は 1983 年に世界に登場したときと変わらない API だ。

バークレーソケット API が古びれない要因の 1 つは、**基礎となるプロトコルの詳細を知らなくてもソケットが使用できる**ということだろう。この点は鍵であり、本書の中で詳しく述べるつもりだ。

バークレーソケット API は、実際のプロトコル実装よりも上のレベルで動作するプログラミング API だ。パケットを連番で整理するというよりはむしろ、2 つのエンドポイントを接続してそれらの間でデータを共有するための API だ。

バークレーソケット API の実装は C で書かれている。しかし、C で書かれているほとんどのモダンな言語には、低レベルのインターフェイスへのバインディングが含まれている。本書でも、得られた知識を他にも持ち運べるよう随所で努力している。

具体的には、単にソケット API 周りの Ruby ラッパーを示すのではなく必ずそのラッパークラスが基にしている低レベルの API を示すことから始

めるようにしている。これによって、本書で得た知識は他の言語でも応用できるはずだ。

Ruby 以外の言語で作業することになっても、ここで学んだ基礎を適用して、目的に応じた低レベルの部品を使えるはずだ。

本書で扱わないこと

前の節で示したように、バークレーソケット API の強みの 1 つは、API を使用するのに「基礎となるプロトコルの詳細を知る必要がない」ことだ。本書ではその思想を全面的に踏襲する。

ネットワークプログラミングの本には、基礎的なプロトコルとその複雑さを説明することに焦点をあてているものや、UDP のような別のプロトコルや生のソケットの上で TCP を再実装するような内容を扱っているものがある。しかし、本書ではそうした内容は扱わない。

「基礎となるプロトコルの実装を知らなくても使用できる」というバークレーソケット API の考えを踏襲し、目的を実現するために API をどう使うかに焦点をあてる。そして、可能な限り、実際の仕事をどう行うかに焦点をあてることに専念する。

しかし、パフォーマンスの最適化を行う場合などは、基礎的なプロトコルへの理解の欠如がその妨げとなる原因となる。そのため、概念が理解できるような最低限必要なことについては説明するつもりだ。

プロトコルに話を戻そう。すでに言ったように、本書では TCP について詳細は説明しない。同じことは、HTTP や FTP などのアプリケーション・プロトコルにも当てはまる。例としていくつかは扱うが、詳細については触れない。

もしプロトコル自体に興味を持ったのなら、スティーヴンスの『詳解 TCP/IP〈Vol.1〉プロトコル』[*1]を読んでみることをお勧めする。

[*1] http://www.amazon.co.jp/dp/4894713209/

netcat

コードの動作確認で任意の接続を行なうために、本書では至るところで `netcat` を使用している。`netcat`（コマンド名は `nc`）は、TCP（と UDP）の任意の接続と待ち受けを行うための Unix ユーティリティだ。`netcat` はソケットを使う際にとても便利なツールだ。

Unix システムを使っているならこのコマンドはおそらくすでにインストールされているので、サンプルを進める際には何の問題もないだろう。

謝辞

まずはじめに、家族に感謝する。Sara と Inara へ。彼女達は、テキストは書いていないが、独自の方法で貢献してくれた。この仕事をするための時間と空間を与えてくれなかったら、そして重要なことのリマインドをしてもらえなかったら、本書は確実に存在していなかっただろう。

次に、素晴らしいレビュアーにも感謝する。次の方々は、本書の草稿を読み、改善案を提供してくれた。大いに感謝している。Jonathan Rudenberg、Henrik Nyh、Cody Fauser、Julien Boyer、Joshua Wehner、Mike Perham、Camilo Lopez、Pat Shaughnessy、Trevor Bramble、Ryan LeCompte、Joe James、Michael Bernstein、Jesus Castello、Pradeepto Bhattacharya。

目次

目次

第 1 章

はじめてのソケット

それでは、例とともに始めていこう。

1.1 Ruby のソケットライブラリ

Ruby のソケットクラス群はデフォルトでは読み込まれない。必要なものはすべて、`require 'socket'` とすることで読み込める。ソケットライブラリは、TCP ソケットや UDP ソケットをはじめとする、必要なプリミティブのためのさまざまなクラスを含んでいる。そのうちのいくつかは、本書を通じてどんなものかわかるはずだ。

> 'socket' ライブラリは Ruby の標準ライブラリの 1 つだ。'openssl' や 'zlib'、'curses' などと同様に、依存する C ライブラリへの薄いバインディングを提供していて、長きに渡って安定して Ruby のリリースに含まれている。

ソケットを作る際には、必ず `require 'socket'` とするように。

1.2 はじめてのソケットを作る

それでは、ソケットを作っていこう。

```
1 require 'socket'
2
3 socket = Socket.new(Socket::AF_INET, Socket::SOCK_STREAM)
```

これで INET ドメインで STREAM タイプのソケットが作られる。INET はインターネットの略で、具体的には IPv4 プロトコルファミリのソケットを参照する。

STREAM の部分はストリームを使って通信すると指定している。このストリームは TCP によって提供される。ここでもし STREAM の代わりに DGRAM（ダイアグラム）を指定した場合は、UDP ソケットを参照する。このタイプの部分は、作成するソケットの種類をカーネルに伝える。

1.3 エンドポイントを理解する

IPv4 について説明する中で、いくつかの新しい用語を使った。話を続ける前に、ここで IPv4 とアドレスについて理解しておこう。

通信したい 2 つのソケットがある場合、それらのソケットは、お互いを見つけられるように互いの場所を知っている必要がある。これは電話と一緒だ。もし誰かと電話をしたいと思ったら相手の電話番号を知っている必要があるだろう。

ソケットは IP アドレスを使って特定のホストへとメッセージを送る。ホストは固有の IP アドレスによって識別される。IP アドレスはホストを指す「電話番号」だ。

先ほど具体的に言及したのは IPv4 アドレスだった。IPv4 アドレスは、「192.168.0.1」のように、ドットでつなげた 255 以下の 4 つの数字で表さ

れる。IP アドレスとは一体何をするものだろうか？ IP アドレスを備えることによって、ホストは特定の IP アドレスを持つ別のホストへデータを送信できる。

IP アドレス電話帳

通信したいホストのアドレスがわかっている場合は、ソケット通信を想像するのは容易だ。けれど、そのアドレスはどうやって取得するんだろう？ 暗記しておかなければいけないのだろうか？ 書き留めておく？ うーん。

DNS という言葉を、これまでにどこかでは耳にしたことがあるのではないだろうか。DNS は IP アドレスとホスト名を対応付けるシステムだ。これを使うと、話したいホストの特定のアドレスを覚えておく必要はなくなる。その代わりに、あなたはホスト名を覚えておく必要がある。ホスト名を覚えておけば、そのホスト名に対応するアドレスの解決を DNS に依頼できる。そうすれば、たとえアドレスが変更になったとしても、ホスト名は常に正しい場所を指すようになる。

1.4 ループバック

IP アドレスは、常にリモートホストを参照しているわけではない。とくに開発時には、ローカルホスト上のソケットへ接続したいと思うだろう。

ほとんどのシステムは、ループバックのインターフェイスを備えている。だいたいは仮想的なインターフェイスであり、ネットワークカードへのインターフェイスと違って、どのハードウェアともつながっていない。ループ

バックインターフェイスへ送ったすべてのデータは、同じインターフェイス上ですぐに受け取れる。ループバックアドレスを使ったネットワークはローカルホストに制約されている。

ループバックインターフェイスのホスト名は `localhost` と呼ばれており、ループバックの IP アドレスは一般的に `127.0.0.1` となっている。これらはシステム内の「hosts」ファイルに定義されている。

1.5 IPv6

IPv4 については先ほど触れたが、IPv6 については言及しなかった。IPv6 は IP アドレスの新しいアドレス指定方式だ。

IPv6 が何故存在しているのだろうか？ それは私たちが IPv4 アドレスを使い切ってしまったからだ。IPv4 は 0-255 の範囲の 4 つの数値からなる。これらの 4 つの数値は 8 ビットで表現でき、最大で 32 ビット分のアドレスを持てる。つまり 2 の 32 乗、すなわち 430 億個のアドレスを持てるということだ。これは大きな数ではある。けれど、想像してみてほしい。あなたが毎日目にしている、ネットワークへと接続されている個々のデバイスはどれくらいの数になるだろう..... アドレスを使い切ってしまったとしても不思議ではないはずだ。

IPv6 は現時点では少しばかり重要な問題だ。IPv4 が使い果たされてしまった今[*1]、IPv6 は価値を持ってくる。IPv6 は桁外れにたくさんの IP アドレスを実現するため、IPv4 とは異なる形式を持つ。

しかし、だいたいの場合それを直接手で打つ必要はない。また、どちらのアドレスの仕組みを使ったとしても通信の仕方には変わりはない。

[*1] `http://www.nro.net/news/ipv4-free-pool-depleted`

1.6 ポート

エンドポイントには、もう 1 つの重要な部分がある。それがポート番号だ。電話番号の例えを続けよう。もしオフィスビル内の誰かと会話をしようとした場合は、内線番号もダイヤルする必要があるだろう。ポート番号はソケットのエンドポイントの「内線番号」だ。

IP アドレスとポート番号の組み合わせは、各ソケットに対して一意でなければならない。つまり、IPv4 アドレスを持つソケットと、IPv6 アドレスを持つソケットが同じポート番号を持つことは可能だが、同じ IPv4 アドレスを持つ 2 つのソケットが同じポート番号を持つことはできない。

ポートがなかったとしたら、ホストは一度に 1 つのソケットしかサポートできない。それぞれのソケットを特定のポート番号に結びつけることで、ホストは同時に数千のソケットをサポートできる。

どのポートを使うべき？

この問題は DNS では解決できないが、定義されているポート番号のリストを使用することで解決できる。

たとえば、HTTP 通信のデフォルトポートは 80 で、FTP 通信のデフォルトポートは 21 だ。このリストの維持に責任がある組織[*2]は実際に存在している。他のポート番号については、次の章で述べる。

[*2] http://www.iana.org/

1.7　2 つ目のソケットを作る

ここでは、Ruby が提供するシンタックスシュガーの最初の一口を味わうことにする。

Ruby にはもっと高度に抽象化したソケットを作る方法もあるが、さまざまなオプションを定数ではなくシンボルで表現する方法も提供している。たとえば、`Socket::AF_INET` は `:INET` 、`Socket::SOCK_STREAM` は `:STREAM` というシンボルでも表現できる。IPv6 ドメインの TCP ソケットを作成する例は次のようになる。

```
1 require 'socket'
2
3 socket = Socket.new(:INET6, :STREAM)
```

これでソケットを作成できた。けれど、まだ他のソケットとデータ交換する準備はできていない。次の章では、ソケットを使って実際の作業を行う準備をしていこう。

1.8　ドキュメント

ドキュメントの話をするのには、今がちょうど良いタイミングだろう。ソケットプログラミングをする際の良いところは、助けとなる多くのドキュメントがマシン上にすでに揃っているところだ。そうしたドキュメントを見つけるための基本的な場所は 2 つ。man ページと ri だ。

順番にそれぞれ簡単に見ていこう。

1. **Unix マニュアルページ（man ページ）**は、基盤となるシステム関数（C コード）についてのドキュメントを提供する。システム関数は、Ruby のソケットライブラリの基盤となるプリミティブだ。man ページは徹底しているけれども、とても低レベルだ。man ページは、Ruby のドキュメント

に不足している、システムコールが何をするかについての知識を与えてくれる。また、エラーコードの内容も調べられる。

たとえば、上記のサンプルコードでは `Socket.new` を使用した。このメソッドは `socket()` というシステム関数に対応している。次のコマンドを使用することで、このシステム関数の man ページを参照できる。

```
$ man 2 socket
```

2 について教えて欲しいって？ これは `man` プログラムに man ページ中のセクション 2 を探すよう指示している。man ページは複数のセクションに分割されている。

- セクション 1: だれもが実行できるユーザーコマンド
- セクション 2: システムコール
- セクション 3: C ライブラリ関数
- セクション 4: デバイス
- セクション 5: ファイルフォーマット
- セクション 7: その他についての概要。tcp(7) は重要

本書では socket(2) のような表記で man ページを参照する。これはセクション 2 にある socket の man ページを参照することを表す。いくつかのコマンドはセクションをまたいで存在することがあるため、このように記述する必要がある。たとえば stat(1) と stat(2) なんかがそうだ。

socket(2) の「SEE ALSO」セクションを見ることで、探しているかもしれない他のシステムコールを見つけられるだろう。

2. **ri** は Ruby のコマンドラインドキュメンテーションツールだ。Ruby をインストールするとコアライブラリのドキュメントは一緒にインストールされる。

Ruby のいくつかの部分は十分にドキュメント化されていない。しかし、

ソケットライブラリは十分カバーされていると思う。ri コマンドを使って Socket.new に関するドキュメントを見てみることにしよう。次のコマンドを打ってみて欲しい。

```
$ ri Socket.new
```

ri は便利で、インターネット接続も必要としない。説明や例が必要なときには、きっと役に立つはずだ。

1.9 この章で扱ったシステムコール

各章では、その章で紹介した新しいシステムコールを一覧していくことにする。詳しく調べたい場合には、ri コマンドか man ページを使って参照してほしい。

- Socket.new -> socket(2)

第 2 章

接続の確立

TCP 接続は 2 つのエンドポイント間で行われる。エンドポイントは同一マシン上かもしれないし、世界のどこか、異なる場所にある別々のマシン上かもしれない。けれど、背後にある原則は同じだ。

ソケットを作ると、そのソケットは「接続を開始するソケット」か「接続を待機するソケット」のいずれかの役割を担う。どちらの役割も必須だ。「接続を待機するソケット」がなければソケットは接続を開始できない。同様に、「接続を開始するソケット」がなければソケットが接続を待機する必要もなくなる。

ネットワークプログラミングで一般的に使われる用語だと、「**接続を待機するソケット**」**がサーバー**、「**接続を開始するソケット**」**がクライアント**となる。順番にそれぞれのライフサイクルを見ていこう。

第 3 章

サーバーのライフサイクル

サーバーソケットは接続を開始せず、待機（リッスン）する。典型的なライフサイクルは次のようになる。

1. 作成（create）
2. 割り当て（bind）
3. 待機（listen）
4. 受付（accept）
5. クローズ（close)

1. についてはすでに説明した。ここでは残りの部分について見ていこう。

3.1　サーバーの割り当て

サーバーソケットのライフサイクルにおける 2 番目のステップは、接続を待ち受けるポート番号への**割り当て（バインド）**だ。

```
1  require 'socket'
2
3  # まず、新しい TCP ソケットを作成する
4  socket = Socket.new(:INET, :STREAM)
5
```

```
# 待機用のアドレスを保持する C 構造体を作成する
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')

# C 構造体をソケットに割り当てる
socket.bind(addr)
```

上記に示したコードは、ローカルポートに TCP ソケットを割り当てる方法を示した低レベルの実装だ。実際、同じことを C 言語で実装したコードと、ほぼ一致する内容となっている。

このコードでは、作成したソケットをローカルホストの 4481 番ポートに割り当てている。これで他のソケットをこのポートに割り当てることはできなくなった。もしそれをした場合には、Errno::EADDRINUSE 例外が発生する。クライアントソケットがこのポートを使ってソケットに接続できるようになるには、さらにいくつかのステップを完了する必要がある。

このコードを実行すると、プログラムは直ちに終了してしまうだろう。コードは機能している。けれど、接続を実際に待機するには、手続きがまだ十分ではないんだ。サーバーを実際に listen モードにする方法は、もう少し待って欲しい。

整理すると、サーバーは、クライアントソケットが接続できる特定のポート番号にソケットを割り当てる。

もちろん、Ruby はシンタックスシュガーを提供していて、それを使えば直接 Socket.pack_sockaddr_in や Socket#bind を使う必要はない。けれど、シンタックスシュガーを学ぶ前に、それがしていることを身をもって経験しておくことは重要だ。

3.1.1 どのポートを割り当てるべき？

これはサーバー実装者にとって重要な判断だ。ランダムなポート番号を選択してもよいのだろうか？ 他のプログラムが特定のポート番号を自分のものとしてすでに主張しているかを、どうやって識別できるだろうか？

1〜65,535 のうちの任意のポートが**使用可能**ではある。けれど、ポートを選ぶ前に考慮すべき重要な慣習がある。

1 番目のルール。**0〜1024 の範囲のポート番号は極力使用しない**。これらのポート番号は「ウェルノウンポート[*1]」と呼ばれ、システムが利用するために予約されている。たとえば、HTTP 通信は 80、SMTP 通信は 25、rsync は 873 がデフォルトのポート番号となっている。通常、これらのポートを割り当てるにはルート権限が求められる。

2 番目のルール。**49,000〜65,535 の範囲のポート番号は使用しない**。これらは短命なポート番号だ。通常これらのポート番号は、あらかじめ定義されたポート番号を必要としない、一時的にポート番号を必要としているサービスによって使われる。また、これらのポート番号は次の節で説明する接続確立プロセスで必要とされるものの 1 つだ。この範囲のポート番号を選ぶと、一部のユーザーにとって問題を引き起こす可能性がある。

これらを除く、**1025〜48,999 内の任意のポートが、公平に利用できるポート番号となる**。もし、サーバーのポート番号としてこれらのうちの 1 つのポート番号を要求するつもりなら、IANA の登録済みポート番号リスト[*2]を確認して、他の人気のあるサーバーと競合しないことを確認する必要がある。

3.1.2 どのアドレスを割り当てるべき？

先ほどの例では、ソケットを `0.0.0.0` に割り当てていた。このとき、`127.0.0.1` や `1.2.3.4` に割り当てた場合とはどのような違いがあるだろうか？ その答えはインターフェイスに関係している。

システムが IP アドレス `127.0.0.1` で表されるループバックインターフェイスを備えていることは前で説明した。システムはまた、異なる IP ア

[*1] よく知られた（well-known）ポート番号

[*2] `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt`

ドレス（たとえば 192.168.0.5）によって表現される、物理的なハードウェアとつながったインターフェイスも備えている。IP アドレスによって表現される特定のインターフェイスへ bind した場合、ソケットはそのインターフェイス上で待機するだけとなり、それ以外のものを無視するようになる。

127.0.0.1 に割り当てた場合は、ソケットは専用のループバックインターフェイス上でだけ待機する。この場合、localhost か 127.0.0.1 に対して作られた接続のみが、サーバーソケットへ送信される。このインターフェイスはローカルでのみ使用可能なため、外部からの接続は許可されない。

192.168.0.5 に割り当てた場合は、ソケットはそのインターフェイス上でだけ待機する。そのインターフェイスを指定できるクライアントであれば接続は待機される。しかし、localhost 上のいかなる接続もサーバーソケットには送信されない。

すべてのインターフェイス上で待機したい場合は 0.0.0.0 を使用できる。0.0.0.0 を使えば、利用できるあらゆるインターフェイスやループバックに割り当てられる。だいたいはこれで困らないはずだ。

```
require 'socket'

# このソケットはループバックインターフェイスに割り当てられ、
# localhost 上のクライアントからの接続だけを待機する
local_socket = Socket.new(:INET, :STREAM)
local_addr = Socket.pack_sockaddr_in(4481, '127.0.0.1')
local_socket.bind(local_addr)

# このソケットは利用できるすべてのインターフェイスに割り当てられ、
# そこにメッセージを送信できるあらゆるクライアントからの接続を待機する
any_socket = Socket.new(:INET, :STREAM)
any_addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
any_socket.bind(any_addr)

# このソケットは不明なインターフェイスに割り当てられようとし、
# その結果 Errno::EADDRNOTAVAIL 例外が発生する
error_socket = Socket.new(:INET, :STREAM)
```

```
error_addr = Socket.pack_sockaddr_in(4481, '1.2.3.4')
error_socket.bind(error_addr)
```

3.2 接続の待機

ソケットを作成してポートに割り当てた後は、やってくる接続を待機するようソケットに指示する必要がある。

```
require 'socket'

# ソケットを作成し、ポート番号 4481 に割り当てる
socket = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
socket.bind(addr)

# やってくる接続を待機するようソケットに指示する
socket.listen(5)
```

前節のコードに付け加えたのは、ソケットへの `listen` の呼び出しだけだ。

このコードを実行しても、やはりまだすぐに終了してしまうだろう。接続を処理できるようになるには、サーバーソケットのライフサイクルとして必要なステップがあと 1 つ必要だ。それについては次節で説明する。ここではまず、`listen` について詳しく触れていこう。

3.2.1 Listen キュー

`listen` メソッドに整数の引数が渡されていることに気付いただろうか。この数値は、サーバーソケットが保留可能な接続の最大数を表している。この保留可能な接続のリストは **Listen キュー**と呼ばれる。

サーバーにやってきた新しいクライアント接続がいずれも Listen キューに格納されるような状況を、クライアント接続の処理によって「サーバーが

ビジー状態である」と言う。もし、新しいクライアント接続があって Listen キューがいっぱいだった場合は、クライアントは Errno::ECONNREFUSED 例外を発生する。

3.2.2 Listen キューはどれくらい大きくすべき？

Listen キューの大きさは何だかマジックナンバーのようだ。なぜキューの最大数を 10,000 に設定しないのだろう？ なぜ接続を拒否しないといけないのだろう？ なるほど。よい質問だ。

まず、制限値について話すことにする。実行時に Socket::SOMAXCONN を評価することで、Listen キューの設定可能な制限値を取得できる。私の Mac 上では、この値は 128 だった。なので、私はそれよりも大きい値は設定できない。root ユーザは、必要に応じて、システムレベルでサーバーのこの制限値を上げられる。

サーバーを動かしていて Errno::ECONNREFUSED 例外が発生したとする。そのときは、Listen キューのサイズを大きくするところから出発してみるとよいだろう。しかし、最終的には Listen キューに接続を溜めていたくはないはずだ。それはサービスのユーザーがレスポンスを待たされていることを意味している。つまり、そうなったときには、サーバーのインスタンスを増やす、あるいはアーキテクチャを変更する、といった対策が必要な可能性がある。

通常は接続を拒否したくはないはずだ。server.listen(Socket::SOMAXCONN) とすることで、Listen キューのサイズをシステムとして許可された最大値に設定できる。

3.3 接続の受付

最後に、サーバーが接続を実際に処理する部分について説明する。接続の受付は accept メソッドを使って行う。作成したソケットを待機させ、最初

の接続を受け付けるには次のようにする。

```
require 'socket'

# サーバーソケットを作成する
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 接続を受け付ける
connection, _ = server.accept
```

このコードを実行してみると、すぐには終了しないことに気付くはずだ。そう、accept メソッドは接続を受け付けるまで処理をブロックするんだ。それでは、netcat を使って接続を与えてみることにしよう。

```
$ echo ohai | nc localhost 4481
```

このコマンドを実行すると、nc(1) が実行され、Ruby プログラムが正常に終了することを確認できるはずだ。これまで見たことがないような劇的な結末ではなかったかもしれない。けれど、これですべてが接続され正しく機能したことが証明された。おめでとう！

3.3.1 accept は処理をブロックする

accept はブロッキング呼び出しだ。呼び出されると、新しい接続を受信するまで現在のスレッドを無期限にブロックする。

前節で説明した Listen キューを覚えているだろうか？ accept はそのキューから保留中の次の接続を単にポップする。そして、もし取り出すべき接続が存在していなければ、接続がプッシュされるのを待つ。

3.3.2 accept は配列を返す

先ほどの例では、accept 呼び出しから 2 つの値を受け取っていた。accept メソッドは、実際には Array オブジェクトを返却する。Array オブジェクトは 2 つの要素を含んでいる。1 つ目が接続そのもので、 2 つ目が Addrinfo オブジェクトだ。2 つ目の要素である Addrinfo オブジェクトは、クライアント接続のリモートアドレスを表現している。

Addrinfo

Addrinfo は、ホストとポート番号を表現する Ruby のクラスだ。エンドポイントの表現をうまい具合に閉じてくれる。いくつかの場所で Socket の標準インターフェイスの一部のように扱える。

このオブジェクトは、Addrinfo.tcp('localhost', 4481) のように書くことで構築できる。オブジェクトが持つメソッドで便利なのは#ip_address と#ip_port だ。詳細は$ ri Addrinfo で確認してみてほしい。

それでは、accept から返された接続情報とアドレス情報を詳しく見ていこう。

```ruby
require 'socket'

# サーバーソケットを作成する
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 新しい接続を受け付ける
connection, _ = server.accept

print 'Connection class: '
p connection.class

print 'Server fileno: '
p server.fileno

print 'Connection fileno: '
p connection.fileno

print 'Local address: '
p connection.local_address

print 'Remote address: '
p connection.remote_address
```

（先ほどの netcat コマンドなどを使って）サーバーが接続を受け付けると、出力は次のようになる。

```
Connection class: Socket
Server fileno: 5
Connection fileno: 8
Local address: #<Addrinfo: 127.0.0.1:4481 TCP>
Remote address: #<Addrinfo: 127.0.0.1:58164 TCP>
```

このちょっとした実行結果は、TCP 接続がどのように行なわれるかについて、多くのことを教えてくれる。少しだけ深く潜ってみよう。

3.3.3 接続クラス

accept は「接続」を返す。けれど、先ほどの実行結果は、特別な「接続クラス」が存在しないことを示している。接続は、実際には Socket クラスのインスタンスとして表現される。

3.3.4 ファイルディスクリプタ

accept が Socket クラスのインスタンスを返すのはわかったが、この接続はサーバーソケットとは異なる、ファイルディスクリプタ番号（fileno）を持っている。ファイルディスクリプタ番号とは、カーネルが提供している現在のプロセスで開かれているファイルを追跡するための番号だ。

> **ソケットはファイル？**
>
> そう。少なくとも Unix の世界では、すべてがファイル[*3]として扱われる。これには、ファイルシステム上にあるファイルだけではなく、パイプやソケット、プリンタなどすべてのものが含まれる。

これは、accept がサーバーソケットとは異なる、まったく新しい Socket を返したことを示している。この Socket インスタンスは接続を表している。これは重要だ。それぞれの接続は、新しい Socket オブジェクトによって表現される。これによって、サーバーソケットはそのままの状態で、新しい接続を受け付け続けられる。

3.3.5 接続アドレス

接続オブジェクトは 2 つのアドレスを持つ。ローカルアドレスとリモートアドレスだ。リモートアドレスは accept の 2 つめの返り値だ。また、リモートアドレスは接続情報オブジェクトの remote_address からも参照できる。

接続オブジェクトの local_address はローカルマシン上のエンドポイントへの参照となる。接続オブジェクトの remote_address はもう一方の端のエンドポイントへの参照となる。もう一方の端のエンドポイントは別のホスト上にある可能性もあるが、今回のケースでは同じマシン上にある。

TCP 接続は、ローカルホスト、ローカルポート、リモートホスト、リモートポートからなる固有のグループによって定義される。これらの 4 つのプロパティの組み合わせは、TCP 接続ごとに必ず一意でなくてはならない。

細かく見ていこう。リモートポートが一意なら、ローカルホストはリモートホストへの接続を 2 つ同時に開始できる。同様に、ローカルポートが一意なら、リモートホストは同一のローカルポートからの接続に 2 つ同時に応答できる。しかし、ローカルポートやリモートポートが同じである場合、同時に 2 つは接続**できない**。

3.3.6 接続受付のループ

accept は 1 つの接続を返す。前述のコード例では、サーバーは 1 つの

[*3] http://ph7spot.com/musings/in-unix-everything-is-a-file

接続を受け付けて終了していた。実際のサーバーを書く場合は、だいたいが可能な限り長く接続を待機し、処理をし続けることになる。これはループを使って簡単に実現できる。

```
require 'socket'

# サーバーソケットを作成する
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)

# 接続を受け付ける無限ループに入り、
# 接続を処理する
loop do
  connection, _ = server.accept
  # 接続を処理する
  connection.close
end
```

上記は Ruby を使って何らかのサーバーを書く際の一般的なやり方だ。なので、Ruby では、こうした処理の上にいくつかのシンタックスシュガーを提供している。この章の最後では、そうした Ruby が提供するラッパーメソッドについて見ていくつもりだ。

3.4 接続のクローズ

接続を受け付けて処理を終え、サーバーが最後にするのは接続の `close` だ。これで接続の「開始ー処理ー終了」ライフサイクルが締めくくられる。

ここでは特に新しいコード例は示さない。先ほどのコード例を見て欲しい。新しい接続を受け付ける前に、接続オブジェクトの `close` を呼び出すようになっている。

3.4.1 終了時のクローズ

なぜ close が必要なのだろうか？ プログラムを終了するときには、オープンしているすべてのファイルディスクリプタ（ソケットを含む）はクローズされるはずだ。では、なぜ明示的にクローズしなければならないのだろう？ これには、いくつかのもっともな理由がある。

1. リソースの使用状況。もし、ソケットを使って処理をしたにも関わらず、それをクローズしない場合、使われなくなったソケットへの参照が保持されたままになる可能性がある。Ruby の GC は、参照されなくなった接続をあなたに代わって掃除してくれる。けど、リソースの使用状況をしっかりと制御し続け、不要なものは取り除いておくことをお勧めする。GC は回収したものをすべてクローズするということに注意しよう。
2. オープンできるファイルの上限。これはまあ、1 つ目の理由の拡張だ。すべてのプロセスはオープンできるファイル数に制限を持っている。それぞれの接続がファイルであることを覚えてるだろうか？ 不要な接続をプロセスで保持しつづけると、この制限に少しずつ近づいていき、後で問題を引き起こす可能性がある。

許可されているプロセス毎にオープンできるファイル数の上限を取得するには `Process.getrlimit(:NOFILE)` を使おう。ソフトリミット（ユーザが設定した値）とハードリミット（システム上の制限値）を格納した配列が返される。
もし最大まで制限値を引き上げたければ、`Process.setrlimit(Process.getrlimit(:NOFILE)[1])` を使うとよい。

3.4.2 さまざまなクローズの仕方

ソケットは双方向通信（読み込み／書き込み）が許されている。そのため、それらのチャンネルのうちの 1 つだけをクローズするということも実際には可能だ。

```
require 'socket'

# サーバーソケットを作成する
server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(128)
connection, _ = server.accept

# データの書き込みは必要なくなったが、読み込みはまだ続けたい
connection.close_write

# 読み込みも必要なくなった
connection.close_read
```

出力ストリームが閉じられると、ソケットのもう一方のエンドポイントに EOF が送られる（EOF についてはこの後すぐに触れる）。

> **接続のコピーはどのように存在する？**
>
> Socket#dup を使うと、ファイルディスクリプタのコピーを作成できる。これは dup(2) を使ってオペレーティング・システムのレベルでファイルディスクリプタを複製する。しかし、これはかなり特殊な操作であり、おそらく見かけることはないだろう。

> より一般的なのは、Process.fork を使ってファイルディスクリプタのコピーを得る方法だ。このメソッドは、現在のプロセスの完全なコピーを持ったまったく新しいプロセス（Unix のみ）を作成する。メモリ内のすべてのコピーを提供するのに加え、開いているすべてのファイルディスクリプタを dup(2) する。そうして、新しいプロセスはコピーを取得できる。

close は、呼び出したソケットのインスタンスを閉じる。システム内にソケットの他のコピーが存在する場合は、ソケットは**閉じられず**、失ったリソースも取り戻せない。実際、たとえ 1 つのインスタンスが閉じられたとしても、接続の他のコピーがデータ交換を続けている可能性がある。

close とは異なり、shutdown は現在のインスタンスとそのすべてのコピー上で起こっている通信を無効にし、ソケットとそのすべてのコピーの通信を完全にシャットダウンする。しかし、ソケットによって使用されたリソースは取り戻せない。ライフサイクルを完了するには、それぞれのソケットのインスタンスは必ず close しなければならない。

```
1  require 'socket'
2
3  # サーバーソケットを作成する
4  server = Socket.new(:INET, :STREAM)
5  addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
6  server.bind(addr)
7  server.listen(128)
8  connection, _ = server.accept
9
10 # 接続のコピーを作成する
11 copy = connection.dup
12
13 # 接続のすべてのコピーの通信をシャットダウンする
14 connection.shutdown
```

```
15
16  # オリジナルの接続をクローズする。
17  # GC に収集されたタイミングで、コピーはクローズする
18  connection.close
```

3.5 Ruby ラッパー

私たちは皆、Ruby が提供するエレガントな構文に馴染み、そして愛している。サーバーソケットを作り、扱うための拡張も例外ではない。それらの便利なメソッド群は、カスタムクラスの中に典型的なコードをラップし、可能な場所では Ruby のブロックの力を活かしている。それでは、見ていこう。

3.5.1 サーバーの構築

最初に取り上げるのは TCPServer クラスだ。これは、プロセス中の「サーバー本体」部分を抽象化したうまいやり方だ。

```
1  require 'socket'
2
3  server = TCPServer.new(4481)
```

うん、Ruby らしいコードだ。これは次のコードを効率的に置き換えたものだ。

```ruby
require 'socket'

server = Socket.new(:INET, :STREAM)
addr = Socket.pack_sockaddr_in(4481, '0.0.0.0')
server.bind(addr)
server.listen(5)
```

使いたくなるコードがどちらかは明らかだろう。

3.5.2 接続処理

Ruby はサーバー本体の準備に加えて、接続処理に対しても、より抽象化したインターフェイスを提供している。

複数の接続を処理するために loop を使っていたのを覚えているだろうか？ loop を使うなんてバカらしい。次のようにしよう。

```ruby
require 'socket'

# ソケットを作成し待機する
server = TCPServer.new(4481)(4481)

# 接続を受け付ける無限ループに入り、
# 接続を処理する
Socket.accept_loop(server) do |connection|
  # 接続を処理する
  connection.close
end
```

ブロックの終わりで接続が自動的に**閉じられない**ことに注意しよう。ブロック引数には accept を呼び出した時の戻り値と同じ接続が渡される。

Socket.accept_loop は、実は待機しているソケットを複数渡すことができ、そのいずれかで接続を受け付けられるという、追加の利点を持っている。これは、次のように用いることができ、Socket.tcp_server_sockets

と本当に相性がいい。

```
require 'socket'

# ソケットを作成し待機する
servers = Socket.tcp_server_sockets(4481)

# 接続を受け付ける無限ループに入り、
# 接続を処理する
Socket.accept_loop(servers) do |connection|
  # 接続を処理する
  connection.close
end
```

Socket.accept_loop にソケットの集合を渡し、それらが美しく処理している部分に注目してほしい。

3.5.3 すべてを 1 つに

Ruby ラッパーの極めつけは Socket.tcp_server_loop だ。これは、上記のすべての手順を 1 つにラップする。

```
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 接続を処理する
  connection.close
end
```

このメソッドは実際のところ、Socket.tcp_server_sockets と Socket.accept_loop の単なるラッパーだ。けれど、これ以上簡潔には書くことはできないだろう！

3.6 この章で扱ったシステムコール

- `Socket#bind` -> bind(2)
- `Socket#listen` -> listen(2)
- `Socket#accept` -> accept(2)
- `Socket#local_address` -> getsockname(2)
- `Socket#remote_address` -> getpeername(2)
- `Socket#close` -> close(2)
- `Socket#close_write` -> shutdown(2)
- `Socket#shutdown` -> shutdown(2)

第 4 章

クライアントのライフサイクル

先に触れたとおり、ネットワーク接続を構成する重要な役割は 2 つある。サーバーは「待機する」役割を担い、やってくる接続を処理するために待機する。一方、クライアントはサーバーとの接続を「開始する」役割を担う。つまり、クライアントはサーバーの場所を識別して、そのサーバーへの外部接続を作成する。

はっきりと言えることは、クライアント無しにサーバーは成り立たないということだ。

クライアントのライフサイクルは、サーバーのライフサイクルよりも少しだけ短い。こんな感じだ。

1. 作成（create）
2. 割り当て（bind）
3. 接続（connect）
4. クローズ（close）

1. はサーバーとクライアントで共通だ。なので、ここではクライアントにおける `bind` から見ていくことにする。

4.1 クライアントの割り当て

クライアントソケットはサーバーソケットと同様に、bind と共に人生を歩み始める。サーバーの章では、bind を特定のアドレスとポートを指定して呼び出していた。サーバー側では bind の呼び出しを省略することは稀だが、**クライアント側では bind を呼び出す方が稀だ**。クライアントソケット（サーバーも同様）が bind 呼び出しを省略した場合は、一時的に利用可能な範囲から無作為にポート番号が割り当てられる。

なぜ bind を呼びださない？

クライアントが bind を呼び出さないのは、ポート番号でアクセスされる必要がないからだ。逆に、サーバーはクライアントから特定のポート番号でアクセスできることを期待されているため、特定のポート番号を割り当てる必要がある。

FTP を例にすると、FTP でよく使われるポート番号は 21 だ。したがって、FTP サーバーはそのポート番号を割り当てて、クライアントから見つけられるようにする必要がある。けれど、クライアントはどのポート番号からでも接続できる。クライアントのポート番号が何であるかはサーバーに影響を与えない。

クライアントが bind を呼び出さないのは、誰もクライアントのポート番号を必要としないからだ。

オススメは「ポートを割り当てない！」なので、ここではコード例は特に記述しない。

4.2 クライアントの接続

サーバーとクライアントの処理が本当に分かれるのは、connect 呼び出しからだ。connect 呼び出しは、リモートソケットへの接続を初期化する。

```
require 'socket'

socket = Socket.new(:INET, :STREAM)

# ポート番号 80 番での google.com への接続を初期化する
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

ふたたび低レベルのプリミティブを使って説明するため、ここでは接続先を C 構造体に格納する必要がある。

このコード例は、一時的に利用可能な任意のローカルポートを使って TCP 接続を初期化し、google.com のポート番号 80 のソケットを待機している。bind 呼び出しを行っていないことに注目してほしい。

4.3 接続にしくじる

クライアントのライフサイクルにおいて、サーバーが接続を受け付けられるようになる前に、クライアントソケットがサーバーに接続しにいくことは十分に起こりうる。存在しないサーバーに接続しにいく可能性もある。こうした場合、実際にはどちらの状況でも同じ結果となる。TCP は楽観的なプロトコルなので、リモートホストからの応答をできる限り長く待つ。

それでは、存在しないエンドポイントに接続してみよう。

```
require 'socket'

socket = Socket.new(:INET, :STREAM)

```

```
# gopher 用として知られるポート番号で google.com に接続しようとした
remote_addr = Socket.pack_sockaddr_in(70, 'google.com')
socket.connect(remote_addr)
```

このコードを実行すると、connect 呼び出しから戻るのに**長い**時間がかかるはずだ。connect のタイムアウト設定は、デフォルトでは長めに設定されている。

これはクライアントにとって実に理にかなっている。帯域幅に影響され接続を確立するまでに長い時間がかかる可能性があるからだ。帯域幅が潤沢なクライアントであれば、リモートアドレスがすぐに接続を受け付けるだろう。

とはいえ、もし時間内に接続できなかった場合には Errno::ETIMEDOUT 例外が発生することになる。これはソケットを扱うときの一般的なタイムアウト例外で、要求された操作がタイムアウトしたことを示している。もしソケットのタイムアウト時間の調整に興味があれば、第 15 章「タイムアウト」を参照してほしい。

同様の事象は、クライアントがサーバーへ接続する際に、サーバー側で bind と listen の呼び出しだけをして accept を呼び出さない場合にも発生する。connect がうまく返るのは、リモートサーバーが接続を受け付けた時だけだ。

4.4 Ruby ラッパー

サーバーソケットを作るコードと同様、クライアントソケットを作るコードも低レベルで冗長だった。ご想像のとおり、Ruby にはこれらをうまくやってくれるラッパーが用意されている。

4.4.1 クライアントの構築

プログラマに優しい Ruby らしいコードをお見せする前に、まずは比較のために、低レベルで冗長なコードをお見せする。

```
require 'socket'

socket = Socket.new(:INET, :STREAM)

# google.com のポート 80 番に接続するよう初期化する
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
socket.connect(remote_addr)
```

Ruby が用意しているラッパーを使うとこうなる。

```
require 'socket'

socket = TCPSocket.new('google.com', 80)
```

だいぶ良い感じだ。2 つのオブジェクトを作っていろんなことをしていた 3 行の処理が、1 つのオブジェクト作成にまとめられた。

ブロックを取る `Socket.tcp` を使っても同様のクライアントを構築できる。

```ruby
require 'socket'

Socket.tcp('google.com', 80) do |connection|
  connection.write "GET / HTTP/1.1\r\n"
  connection.close
end

# ブロックを省略すると、
# TCPSocket.new() と同じように振る舞う。
client = Socket.tcp('google.com', 80)
```

4.5 この章で扱ったシステムコール

- Socket#bind -> bind(2)
- Socket#connect -> connect(2)

第 5 章

データ交換

ここまでは接続の確立、すなわち 2 つのエンドポイントを接続することについて説明してきた。それ自体は興味深いものの、本当に面白いのはその接続を通して行うデータ交換だ。この章からは、その部分へと入っていく。最後には、サーバーとクライアントを結びつけて、会話をさせられるようになっているはずだ。

本題に入る前に、1 つ伝えておきたいことがある。それは、TCP 接続のことをリモートソケットとローカルソケットをつないだ管のようなものだと考えると、とても役立つということだ。私たちはその管を使ってデータの固まりを送ったり受け取ったりできる。バークレーソケット API は、そうした世界観でうまくやれるように設計された。

実際には、すべてのデータは TCP/IP パケットにエンコードされ、目的地に至るまで多くのルータやホストを経由する。これは少しばかりとんでもないことなので、もし何かがうまく動かないというときには、このことを思い出す必要がある。しかし、幸いなことに、多くの人がそのとんでもない部分を見ないで済むように一生懸命働いてくれている。そのおかげで、私たちは単純なメンタルモデルのままでいられる。

5.1 ストリーム

本題に入る前に、もう 1 つ伝えることがある。それは、ここまでは触れてきていない、TCP のストリーム指向という性質についてだ。

本書のはじめで最初のソケットを作成したとき、ストリームソケットを使うと言って、:STREAM オプションを渡していた。TCP はストリーム指向のプロトコルだ。ソケットを作成するときに:STREAM オプションを渡さなければ、そのソケットは TCP ソケットにはならない。

これは何を意味しているのだろう？ コードにはどんな影響があるだろうか？

章のはじめでパケットという用語をそれとなく出していた。下位のプロトコルレベルでは、TCP はネットワーク越しにパケットを送信する。

けど、ここでパケットについて話すつもりはない。アプリケーションコードの観点でみると、TCP 接続は開始も終了もない通信の順次ストリームを提供する。そこには、単にストリームがあるだけだ。

いくつかの擬似コードと共に、これを見ていこう。

```
# このコードは 3 つのデータ片をネットワーク越しに 1 つずつ送信する
data = ['a', 'b', 'c']

for piece in data
  write_to_connection(piece)
end

# このコードは 1 回の操作で 3 つのデータを受信する
result = read_from_connection #=> ['a', 'b', 'c']
```

ここからわかるのは、ストリームにはメッセージ境界の概念がないということだ。クライアントがデータを 3 つに分割して送ったのに対し、サーバー

はデータを読み込んだときにそれを 1 つのデータとして受け取った。クライアントがデータを 3 つのデータとして送信したという事実を、サーバーは知る由もない。

ただ、注意してほしいのは、メッセージ境界は保持されないけれども、ストリーム上のコンテンツ順は保持されているということだ。

第 6 章

ソケットの読み込み

ここまで、接続についてたくさん話してきた。ここからは「ソケット同士の接続を通してどうやってデータを受け渡すか」という、本当に面白い部分に入っていく。当然のことながら、ソケットを使ってデータを読み書きする方法は複数あり、そして Ruby はその上に素敵で便利なラッパーを提供している。

この章では、データを読み込むさまざまな方法を見ていく。

6.1 単純な読み込み

ソケットからデータを読み込む最も単純な方法は `read` メソッドだ。

```
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # もっとも単純に接続からデータを読み込む方法。
  puts connection.read

  # 一度データを読み込んだら接続を閉じる。
  # クライアントに書き込みを待つのを止めてよいと知らせられる。
  connection.close
end
```

ターミナルから上記のスクリプトを実行し、別のターミナルを開いて、次に示す `netcat` コマンドを実行して、サーバー側の出力を確認しよう。

```
$ echo gekko | nc localhost 4481
```

Ruby で `File` クラスの API を使ったことがあれば、上記のコードには馴染みがあるだろう。Ruby のさまざまなソケットクラスは、`File` クラスと同様に、`IO` クラスを親に持つ。Ruby におけるすべての入出力オブジェクト（ソケット、パイプ、ファイルなど）は共通のインターフェイスを持っていて、`read` や `write`、`flush` といったメソッドをサポートしている。

これは Ruby による発明というわけではない。その基となっている read(2)、write(3) といったシステムコールは、ファイルやソケット、パイプなどで、すべて同じように機能する。この抽象化は OS 自体のコアとして組み込まれている。Unix では「**すべてがファイルである**」ことを覚えておいてほしい。

6.2 世の中はそう単純じゃない

単純にデータを読み込めるものの、この方法は脆弱だ。もし、先ほどのコードを再び実行して、次のような netcat コマンドを実行して放置した場合、サーバーはデータを読み終えることなく、永遠に終了しない。

```
$ tail -f /var/log/system.log | nc localhost 4481
```

この挙動の原因は、EOF(End-Of-File) にある。EOF については次の節で詳しく説明する。ここでは詳細はひとまず詳細は置いたまま、ざっくりとした理由を見ていくことにする。

問題の要点は、tail -f はデータ送信を終了しないということだ。末尾に残りのデータが存在しない場合、tail コマンドはそれらが追加されるのを待つ。そのため、tail コマンドは netcat へのパイプを開いたままとなり、netcat もサーバーへのデータ送信を終えることがなくなってしまう。

サーバーの read 呼び出しは、クライアントがデータを送信し終えるまでブロックしつづける。この場合、サーバーは待って、待って、待ちつづける。その間受信したデータをメモリ内にバッファリングしつづけ、それをプログラムに戻すことはない。

6.3 読み込む長さ

上記の問題を回避する方法の 1 つは、読み込む長さを指定することだ。そうすることで、クライアントが終了するまでデータを読み込み続ける代わりに、一定量のデータを read して処理を戻すようサーバーに指示できる。

```
1 require 'socket'
2 one_kb = 1024 # bytes
3
```

```
4  Socket.tcp_server_loop(4481) do |connection|
5    # 1 KB のデータを読み込む
6    while data = connection.read(one_kb) do
7      puts data
8    end
9
10   connection.close
11 end
```

この例を先ほどのコマンドと一緒に実行してみる。

```
$ tail -f /var/log/system.log | nc localhost 4481
```

今度は、netcat コマンドを実行している間、サーバーはデータを出力するはずだ。1 キロバイト分のデータが出力されるだろう。

違いは read に整数値を渡しているかどうかだ。この整数値は、その量までデータを読み込んだら、読み込みを止めて処理を戻すようサーバーに指示している。ここでは、可能な限りデータの取得を継続したいので、read がデータを返さなくなるまでループを繰り返している。

6.4 ブロックの性質

read 呼び出しは、データがすべて到着するまで処理をブロックして待つ。1 キロバイトずつデータを読み込んでいた、先ほどの例を見てみよう。これを数回実行してみると、次のことが明らかになる。読み取ったデータ量が 1 キロバイト未満の場合は、1 キロバイト分のデータを読み込むまで read 呼び出しはブロックし続ける。

実際、この方法を使用するとデッドロック状態に陥る可能性がある。サーバーが接続から 1 キロバイトずつデータを読み込もうとしているときに、クライアントが 500 バイトだけデータを送信してから待機すると、サーバーは

1 キロバイト受信するまでデータを待ち続けることになる。

この状況は 2 つの方法で改善できる。

1. クライアントが 500 バイトのデータを送ったあとで EOF を送信する、
2. サーバーがデータの部分読み込みを行う。

6.5 EOF

データを読み込み中の接続が EOF を受け取った場合、接続はこれ以上データが送られて来ないものと判断して読み込みを終了する。いずれの IO 操作を理解するにも、EOF は重要な概念だ。

まずはじめに、EOF が「end of file」を意味しているという歴史的な経緯について少しだけ触れる。「でも、ここではファイルなんて扱ってないんだけど.....」うん、その気持ちはわかる。けれど、ここで思い出してほしいのは「すべてはファイルである」という考え方だ。

「EOF 文字」への参照を見たことがあるかもしれない。しかし、実際にはそのようなものはない。EOF は文字コードの表現ではない。**EOF はそれよりももっと状態イベントに近いものだ**。ソケットは、これ以上書き込むデータがない場合に、`shutdown` か `close` を使ってそれを示せる。その結果、もう一端の読み込み側に EOF が送られ、データがこれ以上送られてこないと伝えられる。

この周期を使って、先ほどのデッドロック問題を解決していこう。サーバーが 1 キロバイトのデータを期待しているときに、クライアントは 500 バイトのデータだけを送っていた。

この問題を解決するには、500 バイトのデータを送信した後でクライアントが EOF を送ればよい。サーバーは EOF を受け取ると、読み込んだ量がたとえ 1 キロバイトに達していなくても、データの読み込みを止める。EOF はこれ以上のデータが来ないことを伝えるものだ。

したがって、次のようにすればコードはうまく動作する。

```
require 'socket'
one_kb = 1024 # bytes

Socket.tcp_server_loop(4481) do |connection|
  # 1 キロバイトのデータを読み込む
  while data = connection.read(one_kb) do
    puts data
  end

  connection.close
end
```

クライアント接続は次のようにする。

```
require 'socket'

client = TCPSocket.new('localhost', 4481)
client.write('gekko')
client.close
```

EOF を送信する最も単純な方法は、そのソケットを閉じることだ。閉じられれば、間違いなくそのソケットはそれ以上データを送信しない！

EOF という名称が適切である簡単な説明はこうだ。`File#read` を呼び出したとする。これは `Socket#read` みたいなものだ。`File#read` は読み込むものがなくなるまでデータを読み込みつづける。そして、ファイル全体を読み終わったところで EOF を受け取り、読み込んだデータを返す。

6.6 部分読み込み

少し前に「部分読み込み」という用語に触れた。部分読み込みは、前節の最後と同じような状況で使える方法だ。これについて見ていこう。

最初に見たデータの読み込み方は怠惰だった。`read` を呼び出すと、指定

された長さのデータを受け取るか EOF を受け取るかしてデータが返るまで、できるだけ長く待つ。逆のアプローチを取りデータを読み込む代替方法がある。それが readpartial だ。

readpartial 呼び出しは、ブロックするのではなく利用可能なデータをすぐに返す。readpartial を呼び出す場合は、最大長を指定する整数の値を引数に渡す必要がある。readpartial はその長さまでデータを読み込む。もし、1 キロバイトずつデータを読み込むよう指定している状態でクライアントが 500 バイトしかデータを送ってこなかったとしても、readpartial はブロックしない。直ちにそのデータを返す。

次のサーバーを実行してみてほしい。

```
require 'socket'
one_hundred_kb = 1024 * 100

Socket.tcp_server_loop(4481) do |connection|
  begin
    # 100 キロバイトかそれ以下のデータを読み込む
    while data = connection.readpartial(one_hundred_kb) do
      puts data
    end
  rescue EOFError
  end

  connection.close
end
```

クライアントは次のようにする。

```
$ tail -f /var/log/system.log | nc localhost 4481
```

実行結果は、サーバーが 100 キロバイトの固まりを待つのではなく、利用可能な少量のデータの束を流していることを示す。readpartial は、デー

タが存在していれば最大長に達していなくても喜んでそれを返す。

EOF の扱いについて、readpartial の振る舞いは read と異なる。read が EOF を受け取ったときに単に戻るのに対して、readpartial は EOFError 例外を発生させる。

繰り返しになるが、要するに read は怠けものだ。できるだけ多くのデータを返すために、できるだけ長く待つ。逆に、readpartial は頑張り屋だ。利用可能なデータをできるだけはやく戻す。

この後は、write の基本を見た後、バッファについて触れる。そこまで行けば、次のような興味深い疑問のいくつかについての答えが手に入るはずだ。一度に読み込むことのできる最大のデータサイズはどれくらいだろう？細かくたくさん読み込むのと、一度に多く読み込むのとでは、どちらが良いのだろうか？

6.7 この章で扱ったシステムコール

- Socket#read -> read(2)。より fread(3) に近い振る舞いをする。
- Socket#readpartial -> read(2)

第 7 章

ソケットへの書き込み

勘の良い読者なら、次にこの話題が来ることを予想しただろう。あるソケットがデータを読み込むには、別のソケットがデータを書き込む必要がある！

ソケットに書き込むために使う道具はただ 1 つ、write メソッドだ。write メソッドの使い方は、とても簡単で直感的だ。

```
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 接続にデータを書き込む単純な方法
  connection.write('Welcome!')
  connection.close
end
```

write の使い方の基本はこれですべてだ。書き込み周りの興味深い疑問のいくつかについては、次の章でバッファリングについて見ていく中で答えていくことにする。

7.1 この章で扱ったシステムコール

- `Socket#write` -> write(2)

第 8 章

バッファリング

この章で、いくつかの重要な質問に答えていく。一度の呼び出しで、一体どれくらいのデータを読み書きすべきだろう？ `write` 呼び出しから無事に返った場合、それは接続しているもう一端のソケットがデータを受信したことを意味しているのか？ 大きいサイズでの `write` 呼び出しは、小さいサイズでの `write` 呼び出しに分割すべきだろうか？ それはどんな影響があるのだろう？

8.1 ライトバッファ

TCP 接続でデータを `write` すると実際には何が起きるのか、というところから話を始めることにしよう。

`write` を呼び出して、例外が発生することなく戻ったとする。これは、ネットワークを超えてデータが送られて無事にクライアントソケットに受信された、ということを意味しては**いない**。`write` から正常に戻った場合、それは Ruby の IO システムとその基盤である OS のカーネルにデータが託されたことを示している。

アプリケーションとネットワークハードウェアの間には、バッファのための層が少なくとも 1 つは存在している。それらがどこにあり、実際に何を

しているのかを見ていこう。

`write` からうまく戻ったときに唯一保証されるのは、データがカーネルの手に渡ったということだけだ。カーネルはデータをすぐに送信するかもしれないし、あるいは、効率化のためにデータを保持し、他のデータと組み合わせて送信しようとするかもしれない。

Ruby のソケットは、デフォルトでは `sync` を `true` に設定する。この設定は、Ruby の内部でのバッファリング[*1]をスキップする。内部バッファリングは、データを貯めておくバッファ層を別に追加する。

そもそもなぜバッファするのか？

IO バッファリング層が存在する理由はパフォーマンスだ。通常、それによってパフォーマンスは**大きく**改善する。

ネットワーク経由でのデータ送信は本当に本当に遅い[*2]。バッファリングすることで、`write` 呼び出しをすぐに返すように実装できる。そして、その舞台裏では、カーネルが保留中の書き込みを集め、グループ化し、最大のパフォーマンスを引き出せるようにデータ送信を最適化できる。ネットワークレベルで小さなパケットをたくさん送信することはオーバーヘッドが大きい。そのため、カーネルは小さなデータをまとめて大きなパケットにしてから書き込む。

[*1] `http://jstorimer.com/2012/09/25/ruby-io-buffers.html`

[*2] `https://gist.github.com/2841832`

8.2 どのくらい書き込むのが良いの？

バッファリングについてわかったところで、あらためてこの質問について考えよう。私たちは小さな `write` 呼び出しをたくさんすべきだろうか。それとも 1 回の大きな `write` 呼び出しをすべきだろうか。

バッファのおかげで、実際のところはそれを考える必要はない。通常はカーネルがデータの分割や統合を行い、パフォーマンスを調整してくれるため、どのような書き込みでも良いパフォーマンスが得られるようになっている。もしもファイルやビッグデータといった**本当に**大きな `write` をおこなう場合は、データを分割して RAM に移動されないようにした方が良い。

一般的には、**書き込む必要のあるデータをすべて書き込むことで**、カーネルがそのデータをどのようにまとめるかを判断してくれ、それによって最高のパフォーマンスが得られるはずだ。

8.3 リードバッファ

書き込みだけでなく、読み込みもまたバッファされる。

TCP 接続からデータを `read` する際に最大読み込み長を指定した場合、実際には Ruby はあなたが指定した長さよりも多くのデータを受信できる可能性がある。

この場合、その「余分な」データは Ruby の内部リードバッファに格納される。そして、次に `read` が呼び出されると、Ruby はカーネルに新しいデータを要求する前に、まず保留中のデータが無いか内部バッファを確認する。

8.4 どのくらい読み込むのが良いの？

この質問の答えは、ライトバッファのように単純にはいかない。そのため、課題とベストプラクティスを見ていくことにする。

TCP はデータのストリームを提供するので、送信側からどのくらいデータが送られてくるかはわからない。これは、読み込み長を考える際は常に推測して決定するしかないことを意味している。

なぜ、単純に大きな読み込み長を指定してしまって、利用可能なデータをすべて取得するということをしないのだろうか？ 読み込み長を指定すると、カーネルはそのためにいくつかのメモリを割り当てる。ということは、必要以上のメモリを指定すると、使われることのないメモリを割り当てることになってしまう。これはリソースの無駄使いだ。

逆に、すべてのデータを取得するのに読み込みが何回も発生するような小さな長さを指定してしまった場合には、システムコールごとのオーバーヘッドが発生する。

多くの場合がそうであるように、最良のパフォーマンスを得るには、受信データにしたがってプログラムを調整する必要がある。大きなデータの固まりをたくさん受信するとしたら？ そのときは、大きな読み込み長を指定するべきだろう。

銀の弾丸となる答えはない。しかし、**少しチートして**、ソケットを使用しているさまざまな Ruby プロジェクトを調査し、この質問の答えについて統計をとってみた。

Mongrel、Unicorn、Puma、Passenger、Net::HTTP を調査したところ、これらのプロジェクトでは `readpartial(1024 * 16)` となっていた。つまり、読み込み長として 16 キロバイトを採用していた。

Redis プロジェクトだけは、読み込み長として 1 キロバイトを採用していた。

読み込み長を 16 キロバイトとすることに疑問を感じた場合は、手元の

データに従ってサーバーをチューニングすれば、いつでも最高のパフォーマンスを得られるだろう。

第 9 章

はじめてのクライアント/サーバー

やれやれ。

ここまで、接続の確立とデータ交換ついての多くの情報について見てきた。これまでの説明は、とても小さな、自己完結したコード片と一緒に進んできた。そろそろ、ここまで見てきたすべてを、ネットワークサーバーとクライアントに詰め込んで動かしても良い頃合いだ。

9.1 サーバー

今回作るサーバーは、まだ誰も聞いたことのない、次世代の NoSQL ソリューションとする。Ruby のハッシュにネットワーク層の皮を被せてこれを実現することにする。名前は CloudHash が適切だろう。

次に CloudHash サーバーの全実装を示す。

```
1  require 'socket'
2
3  module CloudHash
4    class Server
5      def initialize(port)
```

```
6         # サーバーソケットを作成する。
7         @server = TCPServer.new(port)
8         puts "Listening on port #{@server.local_address.ip_port}"
9         @storage = {}
10      end
11
12      def start
13        # 接続受付のループ。
14        Socket.accept_loop(@server) do |connection|
15          handle(connection)
16          connection.close
17        end
18      end
19
20      def handle(connection)
21        # EOF を受けるまで、接続から読み込む。
22        request = connection.read
23
24        # ハッシュ操作の結果を書き戻す。
25        connection.write process(request)
26      end
27
28      # 補助コマンド:
29      # SET key value
30      # GET key
31      def process(request)
32        command, key, value = request.split
33        case command.upcase
34        when 'GET'
35          @storage[key]
36        when 'SET'
37          @storage[key] = value
38        end
39      end
40    end
41  end
42
43  server = CloudHash::Server.new(4481)
```

```
server.start
```

9.2 クライアント

次に、単純なクライアントの実装を示す。

```
require 'socket'

module CloudHash
  class Client
    class << self
      attr_accessor :host, :port
    end

    def self.get(key)
      request "GET #{key}"
    end

    def self.set(key, value)
      request "SET #{key} #{value}"
    end

    def self.request(string)
      # 各操作ごとに新しい接続を作成する。
      @client = TCPSocket.new(host, port)
      @client.write(string)

      # リクエストを書き込んだら、EOF を送る。
      @client.close_write

      # レスポンスを得るために、EOF まで読み込む。
      @client.read
    end
  end
end

```

```
31 CloudHash::Client.host = 'localhost'
32 CloudHash::Client.port = 4481
33 puts CloudHash::Client.set 'prez', 'obama'
34 puts CloudHash::Client.get 'prez'
35 puts CloudHash::Client.get 'vp'
```

9.3 すべてをつなぎ合わせる

それでは、すべてをつなぎ合わせよう！

まずサーバーを起動する。

```
$ ruby code/cloud_hash/server.rb
```

データ構造がハッシュだということを思い出しておこう。次の操作を行うことで、クライアントが実行される。

```
$ tail -4 code/cloud_hash/client.rb
puts CloudHash::Client.set 'prez', 'obama'
puts CloudHash::Client.get 'prez'
puts CloudHash::Client.get 'vp'

$ ruby code/cloud_hash/client.rb
```

9.4 考察

ここでやったことは何だろう？ Ruby のハッシュをネットワーク API を使ってラップした。けれど、Hash が提供する API すべてをラップしたわけではない。ゲッターとセッターだけだ。コードのかなりの部分は、ネットワークに絡んだ定型のものだ。この例を拡張して、もっとたくさんの Hash API をラップすることも簡単だろう。

コード中のコメントから、コードの意図はわかってもらえると思う。コメントは、接続の確立や EOF など、これまで見てきた考え方と必ず結びつくようにしてある。

しかしながら、CloudHash は全体的には間に合わせの実装だ。これまでのいくつかの章で、接続の確立とデータ交換の基本を見てきた。ここではそれら両方を適用した。この例から欠けているのは、アーキテクチャパターンや設計のベストプラクティス、いくつかのまだ見ぬ高度な機能だ。

たとえば、クライアントは要求ごとに新しい接続を開始しなければならない。そのため、連続するリクエストをまとめて送りたい場合でも、行ごとにそれぞれ別々の接続が必要となる。現在のサーバーの設計では、そのようにせざるをえない。サーバーは、クライアントソケットからのコマンドを 1 つ処理したら、それを閉じるようになっている。

このようにしなければいけない理由があるわけではない。接続の確立はオーバーヘッドなので、それを無くすために同じ接続で複数のリクエストを処理することも可能だ。

それとは異なる方法でも、これを改善することは可能だ。クライアントとサーバーは、単純なメッセージプロトコルを使って通信することも可能だ。単純なメッセージプロトコルを使えば、EOF を区切り文字として送信する必要はない。これによって、1 つの接続で複数のリクエストをさばくことが可能になる。けれども、サーバーは以前として 1 つずつクライアント接続を処理するままだ。1 つのクライアントがたくさんのリクエストを送信したり長時間接続を保持してしまうと、他のクライアントはサーバーとやりとりすることはできないだろう。

サーバーに並行性の形を持たせることで、この問題を解決できる。本書の残りの部分では、効果的で理解しやすいネットワークプログラムを書けるようになるための基礎を構築していく。CloudHash は、現状ではソケットプログラミングがどうあるべきかについての良い例では**ない**。

第 10 章

ソケットオプション

高度なテクニックを扱う章に入る手始めとして、この章ではソケットオプションについて見ていく。ソケットオプションは、ソケットにシステム固有の振る舞いを設定するための低レベルな方法だ。低レベルなため、Ruby もソケットオプションに関するシステムコールについては派手なラッパーを提供していない。

10.1 SO_TYPE

それでは、ソケットオプションについて見ていこう。まずはソケットタイプだ。

```ruby
require 'socket'

socket = TCPSocket.new('google.com', 80)
# ソケットの種類を表現する Socket::Option インスタンスを取得する。
opt = socket.getsockopt(Socket::SOL_SOCKET, Socket::SO_TYPE)

# オプションを表現する整数値と Socket::SOCK_STREAM の値を比較する。
opt.int == Socket::SOCK_STREAM #=> true
opt.int == Socket::SOCK_DGRAM #=> false
```

getsockopt 呼び出しは、Socket::Option のインスタンスを返す。このレベルでは、すべての作業を整数値で解決する。SocketOption#int を使うことで、返ってきたインスタンスに紐付いた整数値を取得できる。

この場合はソケットタイプ（最初にソケットを作成したときに指定したことを覚えているだろうか？）を取得しているので、ソケットタイプ定数と int の値を比較することで、返ってきたのが STREAM ソケットであるとわかる。

こうした定数の代わりに、Ruby は常にメモ化されたシンボルを提供している。上記は次のようにも書ける。

```ruby
require 'socket'

socket = TCPSocket.new('google.com', 80)
# 定数ではなくシンボルを使用する。
opt = socket.getsockopt(:SOCKET, :TYPE)
```

10.2 SO_REUSE_ADDR

これは**すべての**サーバーで設定されるであろう一般的なオプションだ。

SO_REUSE_ADDR オプションは、**ソケットが** TCP_TIME_WAIT **状態のときに**、サーバーの使っているローカルアドレスを別のソケットに割り当ててもよいことをカーネルに伝える。

TIME_WAIT 状態？

この状態は、データをバッファ内に保留しているソケットを close したときに発生する。write 呼び出しが保証するのは、データがバッファに入ったことだけだったことを覚えているだろうか？ ソケットを close しても、バッファに保留されているデータは破棄され**ない**。

その裏で、カーネルは保留中のデータを送信するのに十分な時間、接続を開いたままほったらかす。これは実際にデータを送信して受信側からの受信応答を待ち、時にはデータの再送まで行う必要があることを意味している。

保留中のデータがある状態でサーバーを close し、すぐに同じアドレスに（サーバーをすぐに再起動するなどして）別のソケットを割り当てようとしたとする。すると、保留中のデータが無効になっていない限りは、Errno::EADDRINUSE 例外が発生する。SO_REUSE_ADDR を設定することで、この問題を回避でき、TIME_WAIT 状態でも別のソケットに使われているアドレスを割り当てられるようになる。

設定する方法を次に示す。

```ruby
require 'socket'

server = TCPServer.new('localhost', 4481)

server.setsockopt(:SOCKET, :REUSEADDR, true)
server.getsockopt(:SOCKET, :REUSEADDR) #=> true
```

TCPServer.new と Socket.tcp_server_loop およびその親戚では、このオプションはデフォルトで有効になっている。

ソケットオプションの全リストは、システム上の setsockopt(2) で確認できる。

10.3 この章で扱ったシステムコール

- Socket#setsockopt -> setsockopt(2)
- Socket#getsockopt -> getsockopt(2)

第 11 章

ノンブロッキング IO

この章ではノンブロッキング IO を扱う。ノンブロッキング IO は、非同期やイベント IO とは異なるものだ。もしこれらの違いがわからなくても問題ない。本書の残りを読み進めることでわかるはずだ。

ノンブロッキング IO は次の章で扱う**多重接続**と密接に絡んでいる。しかし、単体でも役に立つ知識なので、まず本章で独立して扱う。

11.1 ノンブロッキング読み込み

少し前に説明した `read` の挙動を覚えてるだろうか？ `read` は EOF を受けるか、最小バイト数を受け取るまで処理をブロックする。この挙動は、クライアントが EOF を送らないときに長時間のブロックを引き起こす可能性がある。

使用可能なデータをすぐに返す `readpartial` を使うことで、ブロックは部分的には回避できる。しかし、使用可能なデータがない場合は、`readpartial` を使っていたとしてもブロックする。**絶対に**ブロックせずに読み込み操作を行なうには、`read_nonblock` が必要だ。

`readpartial` と同様に、`read_nonblock` は最大読み込みバイト数を指定する整数値を引数に取る。もし使用可能なデータがある場合には、その最大

バイト数に達していなくても戻る。read_nonblock は、次のような感じで動作する。

```
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  loop do
    begin
      puts connection.read_nonblock(4096)
    rescue Errno::EAGAIN
      retry
    rescue EOFError
      break
    end
  end

  connection.close
end
```

前のときと同じように、接続を閉じないクライアントを起動しよう。

```
$ tail -f /var/log/system.log | nc localhost 4481
```

サーバーに送られたデータがない場合でも、read_nonblock 呼び出しはすぐに戻る。実際には、その場合は Errno::EAGAIN 例外が発行される。man ページで EAGAIN を引くと、こう書かれている。

> ファイルはノンブロッキング IO によって注意を払われたが、読み込み可能なデータはなかった。

そのとおり。これが、同じ状況だとブロックされたであろう、readpartial との違いだ。

ソケットがブロックしない代わりにこの例外を捕捉した場合は、一体何を

すべきだろうか？ 上記の例では、ビジーループに入って何度も retry するようにした。これは、動作を確認する目的でこうしているだけで、実際に処理する際の適切な方法ではない。

ブロックされた読み込みをリトライする適切な方法は、IO.select を使用することだ。

```
begin
  connection.read_nonblock(4096)
rescue Errno::EAGAIN
  IO.select([connection])
  retry
end
```

この例は、read_nonblock と retry を何度も繰り返すのと同じ動きを、無駄なサイクルを減らして実現している。ソケットの配列を第一引数にして IO.select を呼び出すと、ソケットのいずれかが読み込み可能になるまでブロックする。そのため、retry はソケットが読み込み可能な状態になったときにのみ呼び出される。IO.select についての詳細は、次の章で説明する。

この例では、読み込み時にブロックしていた方法を、ノンブロッキングなメソッドを使って再実装した。それ自体はそんなに有用なことではない。しかし、IO.select を使うことで、複数ソケットを監視したり、読み込みチェックを定期的に行うといった、柔軟性が提供されるということは示すことができた。

読み込みはいつブロックする？

read_nonblock メソッドは、保留中のデータが無いかどうか、ま

ず Ruby の内部バッファをチェックする。もしデータがあった場合には、その段階ですぐに戻る。

そうでなかった場合は、次に select(2) を使い、読み込み可能なデータが無いかどうかをカーネルに尋ねる。もし、カーネルのバッファかネットワーク越しに使用可能なデータがあった場合には、そのデータを取得して戻る。そして、上記のいずれの条件も満たさなかった場合には、read(2) によってブロックされ、read_nonblock から例外を発行することになる。

11.2 ノンブロッキング書き込み

ノンブロッキング書き込みは、前に見た write 呼び出しとの重要な違いがいくつかある。最も注目すべき違いは、write が渡されたすべてのデータを常に書き込もうとするのに対して、write_nonblock は部分書き込みをして返ることが可能だという点だ。

この振る舞いを確認するために、netcat を使って使い捨てのサーバーを起動しよう。

```
$ nc -l localhost 4481
```

続いて、write_nonblock を使った次のクライアントを実行する。

```
1 require 'socket'
2
3 client = TCPSocket.new('localhost', 4481)
4 payload = 'Lorem ipsum' * 10_000
5
6 written = client.write_nonblock(payload)
```

```
written < payload.size #=> true
```

2 つのプログラムを互いに実行すると、クライアント側には true が出力されるはずだ。これはつまり、完全長のペイロードデータ未満の値が返ってきているということだ。write_nonblock メソッドは、ブロックされるとそれ以上データを書き込まない。そうして、どれくらい書き込んだかを示す値を返す。書き込まれなかった残りのデータを書き込むのは、呼び出し側の責任となる。

write_nonblock の振る舞いは、write(2) システムコールと同じだ。書き込める分だけ書き込んで、書き込んだサイズを返す。これは、要求されたすべてのデータを write(2) を必要な回数だけ呼び出して書き込む、Ruby の write メソッドとは異なっている。

一回の呼び出しで、要求されたすべてのデータを書き込めなかったら、一体どうすべきだろうか？ その場合は、書き込めなかった分の書き込みに再挑戦する以外はない。けれど、すぐにそれをしてはいけない。裏で動いている write(2) がまだブロックしていた場合には、Errno::EAGAIN 例外が発行されてしまうからだ。IO.select を利用しソケットが書き込み可能であることがわかったら、ブロックされずに書き込めることを意味している。

```
require 'socket'

client = TCPSocket.new('localhost', 4481)
payload = 'Lorem ipsum' * 10_000

begin
  loop do
    bytes = client.write_nonblock(payload)
    break if bytes >= payload.size
    payload.slice!(0, bytes)
    IO.select(nil, [client])
   end

rescue Errno::EAGAIN
```

```
15    IO.select(nil, [client])
16    retry
17  end
```

ここでは、IO.select の 2 番目の引数にソケットの配列を渡すと、そのいずれかが書き込み可能になるまでブロックするという振る舞いを利用している。

コード中のループ処理は、部分書き込みを正しく扱っている。write_nonblock がペイロードのサイズよりも小さな値を返したときは、ペイロードのデータを分割し、ソケットが再び書き込み可能になるまで待ってからループを繰り返す。

書き込みはいつブロックする？

裏で動く write(2) は、次に挙げる 2 つの状況でブロックする。

1. TCP 接続の受信側がデータを受信しきっていない状況で、許容以上のデータを送った。TCP が採用している輻輳制御のアルゴリズムのおかげで、幸いにもネットワークがパケットで溢れることはない。TCP 接続の受信側にデータが届くまで長い時間がかかるときは、許容以上のデータでネットワークが溢れないよう制御される。
2. TCP 接続の受信側がデータを扱うことができない状況の場合。一度受信したデータを受け入れると、データの受信側は残りのデータを埋めるために、データ「窓」を順にクリアする必要がある。これはカーネルの読み込みバッファのことを指している。もし、受け取ったデータを受信側が処理していない場合、輻輳制御アルゴリズムでは、クライアント側がデータ受信の準備ができるまで、送信側にデータ送信をブロックすることを強制する。

11.3 ノンブロッキング accept

read や write に加え、他のよく使用されるメソッドも、ノンブロッキング版が存在している。

accept_nonblock は、通常の accept にとても似ている。accept が待ち受けキューから接続を単に取り出すと説明したことを覚えてるだろうか。このとき、accept_nonblock はブロックするのではなく、Errno::EAGAIN 例外を発生させる。

次に例を示す。

```
require 'socket'

server = TCPServer.new(4481)

loop do
  begin
    connection = server.accept_nonblock
  rescue Errno::EAGAIN
    # 他の重要な作業を行う。
    retry
  end
end
```

11.4 ノンブロッキング connect

ここまで来ると、connect_nonblock メソッドが何をするかはだいたい推測できると思っただろうか？ そうであれば、少し驚くことになるかもしれない。connect_nonblock は、他のノンブロッキング IO メソッドとは少しばかり違った振る舞いをする。

他のメソッドが処理を完了するか、完了せずに適切な例外を発生させるの

に対して、connect_nonblock は処理を残したまま例外を発生させる。

リモートホストに即座に接続できない場合、connect_nonblock は処理をバックグラウンドで継続し、処理が進行中であることを知らせるために Errno::EINPROGRESS 例外を発生させる。次の章では、バックグラウンドで行っていた処理が完了した際の通知について説明する。ここでは、次の簡単な例のみを示しておく。

```
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  # google.com のポート 80 番へのノンブロッキング接続を初期化。
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  # ノンブロッキング接続の試行を開始。
rescue Errno::EALREADY
  # ノンブロッキング接続の試行中。
rescue Errno::ECONNREFUSED
  # リモートホストは接続を拒否した。
end
```

第 12 章

多重接続

接続の多重化とは、複数のソケットと同時にやりとりすることを指す。これは並列処理を指してはいないし、マルチスレッドとも関係ない。どういうことかは、コード例を見ればわかるはずだ。

これまでに見てきたテクニックを踏まえて、そのときどきに複数の TCP 接続上のデータを処理しなくてはならいサーバーを、どのように実装するかを想像してみよう。おそらく、特定のソケットでブロックされるのを防ぐために、新しく得たばかりのノンブロッキング IO の知識を使うことになるだろう。

```
# 接続の配列を与える。
connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

# 無限ループに入る。
loop do
  # コネクション毎の処理...
  connections.each do |conn|
    begin
      # ノンブロッキングな方法で各接続から読み込みを行い、
      # データを受信していれば処理し、そうでなければ次の接続を処理する。
      data = conn.read_nonblock(4096)
      process(data)
```

```
13      rescue Errno::EAGAIN
14      end
15    end
16  end
```

これはうまくいくだろうか？ 動いた！ けれど、かなり無駄が多い。

read_nonblock を呼び出すたびに、少なくとも 1 つのシステムコールを使う。それに、データが無いときでもサーバーはムダに多くの回数データの読み取りを行うだろう。read_nonblock は select(2) を使って利用可能なデータがあるかどうかをチェックする、と言ったのを覚えているだろうか？ こうした目的のために select(2) を直接使える Ruby ラッパーがあるんだ。

12.1 select(2)

複数の TCP 接続上にあるデータを処理するには、次のようにする。

```
1   # 接続の配列を与える。
2   connections = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
3
4   loop do
5     # 接続が読み込みの準備を完了しているかを select(2) を使って調べる
6     ready = IO.select(connections)
7
8     # 読み込み可能な接続だけからデータを読み込む。
9     readable_connections = ready[0]
10    readable_connections.each do |conn|
11      data = conn.readpartial(4096)
12      process(data)
13    end
14  end
```

この例では、多くの接続を処理する際のオーバーヘッドを減らすために、IO.select を使用している。IO.select の用途は、いくつかの IO オブジェクトを取り、その中で読み書きが可能になっているものを教えることだ。こ

れを使えば、暗闇の中で手を伸ばすようなことはする必要はない。

では、IO.select のいくつかのプロパティを見ていこう。

IO.select は**ファイルディスクリプタが読み書き可能になったことを教えてくれる**。上記の例では、IO.select には 1 つの引数が渡されている。しかし、この引数は実際には 3 つの重要な配列となっている。

```
1 for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
2 for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
3
4 IO.select(for_reading, for_writing, for_reading)
```

1 つ目は読み出したい IO オブジェクトの配列、2 つ目は書き込みたい IO オブジェクトの配列、そして 3 つ目は例外的な条件に興味のある IO オブジェクトの配列だ。帯域外データ（詳しくは 第 18 章「緊急データ」 で説明する）に興味がないなら、ほとんどの場合は第 3 引数は無視して構わない。たとえ単一の IO オブジェクトからの読み取りにしか興味がない場合でも、IO.select に渡す際には配列に入れる必要があることに注意しよう。

IO.select は**多重配列を返す**。IO.select は、引数リストに対応する 3 つの配列を要素に持つ配列を返す。最初の要素にはブロックせずに読み取れる IO オブジェクトが含まれる。これは 1 つ目の引数として渡された IO オブジェクトの配列のサブセットとなる。2 つ目の要素はブロックせずに書き込める IO オブジェクトが含まれ、3 つ目の要素は該当する例外的な条件を持っている IO オブジェクトが含まれる。

```
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

ready = IO.select(for_reading, for_writing, for_writing)

# 引数として渡された各配列用の配列が返される。
# 今回は for_writing 中に書き込み可能な接続がない。
# そして、for_reading 内の接続の 1 つが読み込み可能となっている。
puts ready #=> [[<TCPSocket>], [], []]
```

IO.select は**ブロックする**。IO.select は同期メソッド呼び出しだ。すでに見てきたように、このメソッドは渡された IO オブジェクトの状態が 1 つでも変化するまで処理をブロックする。どれか 1 つでも状態が変化すればメソッドはすぐに返り、複数の状態が変更となっていた場合はネストされた配列を通してその結果が伝えられる。

IO.select は 第 4 の引数にタイムアウト値を秒単位で指定することもできる。これによって、IO.select が永遠にブロックされつづけるのを防げる。タイムアウト値には整数、または浮動小数点値を指定する。IO の状態が変わる前にタイムアウトに達した場合は、IO.select は nil を返す。

```
for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]

timeout = 10
ready = IO.select(for_reading, for_writing, for_writing, timeout)

# このケースでは IO.select は 10 秒以内にいかなる状態の変化も
# 検出できなかったので、二重配列の代わりに nil が返却される。
puts ready #=> nil
```

また、IO オブジェクトを返す to_io メソッドを持っていさえすれば、素の Ruby オブジェクトであっても IO.select にそのまま渡せる。IO オブジェクトとドメインオブジェクトの対応付けを維持しなくてよいので、この

方法はとても便利だ。IO.select は、to_io メソッドを実装してさえいれば、素の Ruby オブジェクトと共に動作できる。

12.2 読み込み／書き込み以外のイベント

ここまで、IO.select を使った、読み書きの状態監視について見てきた。しかし、実際にはそれを他のいくつかの場所に押し込めることも可能だ。

12.2.1 EOF

もし、読み込み用にソケットを監視していて EOF を受信した場合、EOF は読み込み用のソケット配列の一部として返される。その場合、使用している read(2) のバリエーションに応じて、読み込みを行った段階で EOFError か nil を受け取るかもしれない。

12.2.2 応答

もし、読み込みのためにソケットを監視していて、やってきた接続を受信した場合、それは読み込み用のソケット配列の一部として返される。これについては、これらのソケットを扱うロジックを特別持っている必要があり、read よりも、むしろ accept を使う必要がある。

12.2.3 接続

これは、ここで紹介している中で最も興味深いものだ。前の章で、connect_nonblock を使ったときにすぐに接続できなかった場合には、Errno::EINPROGRESS 例外があがると説明した。IO.select を使うことで、そのバックグラウンド接続が完了しているかどうかを把握できる。

```ruby
require 'socket'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')

begin
  # google.com の 80 番ポートへの接続を
  # ノンブロッキング接続を初期化する。
  socket.connect_nonblock(remote_addr)
rescue Errno::EINPROGRESS
  IO.select(nil, [socket])

  begin
    socket.connect_nonblock(remote_addr)
  rescue Errno::EISCONN
    # 成功！
  rescue Errno::ECONNREFUSED
    # リモートホストによって拒否される。
  end
end
```

このコードの最初の部分は、前章のものと一緒だ。connect_nonblock を実行し、バックグラウンドで接続試行が開始していることを示す Errno::EINPROGRESS が発生したら、その例外を補足する。ここでは、その中に新しいコードを追加している。

IO.select を呼び出して、ソケットの書き込み状態に変更がないかどうかを監視している。これによって、変更があった時点で、接続が完了したことを把握できる。そして、状態を確認するために、ここでもう一度 connect_nonblock を実行する！ もし Errno::EISCONN が発生した場合は、すでにリモートホストに接続されている。成功だ！ 別の例外は、リモートホストへの接続におけるエラー状態を示す。

この派手なコード片は、実際には**ブロッキング** connect 呼び出しを実際にエミュレートする。何故だろう？ どんな理由かをある程度は示すことも

できるが、この処理の実装を想像することも難しくないはずだ。connect_nonblock を開始し、いったん止めて別のことを行う。そして、タイムアウトと共に IO.select を呼び出す。裏の接続が終了していなければ、他の処理を継続し、また後で IO.select を確認する。

この小さなテクニックを使うことで、非常に単純なポートスキャナ[*1]を実装できる。ポートスキャナは、リモートホストのあるポートの範囲へ接続を確立しようとする。そして、どのポート接続が開いていたかを伝える。

```
require 'socket'

# パラメータの設定。
PORT_RANGE = 1..128
HOST = 'archive.org'
TIME_TO_WAIT = 5 # 秒

# 各ポートごとにソケットを作成し、
# ノンブロッキングな接続を初期化する。
sockets = PORT_RANGE.map do |port|
  socket = Socket.new(:INET, :STREAM)
  remote_addr = Socket.sockaddr_in(port, 'archive.org')

  begin
    socket.connect_nonblock(remote_addr)
  rescue Errno::EINPROGRESS
  end

  socket
end

# 期限を設定する。
expiration = Time.now + TIME_TO_WAIT

loop do
  # 期限を超えて待つことがないよう、都度タイムアウトを
```

[*1] http://ja.wikipedia.org/wiki/ポートスキャン

```
27    # 調整しながら、IO.select を呼び出す。
28    _, writable, _ = IO.select(nil, sockets, nil, expiration - Time.now)
29    break unless writable
30
31    writable.each do |socket|
32      begin
33        socket.connect_nonblock(socket.remote_address)
34      rescue Errno::EISCONN
35        # ソケットが既に接続されていたら、成功とみなしてカウントできる。
36        puts "#{HOST}:#{socket.remote_address.ip_port} accepts
          ↪  connections..."
37        # ソケットをリストから削除する。
38        # これによって書き込み可能であると選択されることはなくなる。
39        sockets.delete(socket)
40      rescue Errno::EINVAL
41        sockets.delete(socket)
42      end
43    end
44  end
```

このコードでは、connect_nonblock の利点を活用して、100 数個の接続を一度に初期化している。IO.select を使ってすべての接続を監視し、正常に接続できたかを最終的に確認している。次に示すのは、archive.org に向けてこのコードを実行した結果の出力だ。

```
archive.org:25 accepts connections...
archive.org:22 accepts connections...
archive.org:80 accepts connections...
archive.org:443 accepts connections...
```

結果は必ずしも順番通りではないことに注意してほしい。プロセスを終了した接続から出力されている。ここで開いているポートのグループはとても一般的だ。ポートの 25 番 は SMTP、22 番は SSH、80 番は HTTP、そして 443 番は HTTPS 用にそれぞれ予約されている。

12.3 ハイパフォーマンス多重化

IO.select は Ruby のコアライブラリに同梱されている。そして、その中で唯一の、Ruby プログラムで多重化を行うためのソリューションだ。最近のモダンな OS のカーネルは、さまざまな多重化の方法をサポートしている。select(2) はその中で最も古く、できることが小さいものだ。

IO.select は接続が少ない間であればうまく機能する。しかし、性能は監視する接続の数に線形的だ。より多くの接続を監視しようとすると、性能は低下していく。また、IO.select は、FD_SETSIZE という C ライブラリの中で定義されている C マクロによって制限されている。select(2) は、FD_SETSIZE(だいたいのシステムでは 1024) を越える番号のファイルディスクリプタを監視できない。つまり、IO.select は最大で 1024 個の IO オブジェクトしか監視できないよう制限されている。

代替案はもちろん存在する。

poll(2) システムコールは select(2) とは若干の違いはあるが、だいたい同じようなものだ。(Linux の) epoll(2) システムコールと (BSD の) kqueue(2) システムコールはより高い性能を提供する、select(2) と poll(2) の代替だ。たとえば、EventMachine のような高性能なネットワーク用のツールキットは、可能な所では epoll(2) や kqueue(2) を優先して利用する。

これら特定のシステムコールの例をあげることもできる。けれど、ここでは nio4r[*2]を紹介したい。nio4r は、使用するシステム上で最も性能のでる多重化の方法を用いつつ、多重化ソリューションに対する共通のインターフェイスを提供する gem パッケージだ。

[*2] https://github.com/celluloid/nio4r

第 13 章

Nagle アルゴリズム

Nagle アルゴリズム[*1]は、すべての TCP 接続にデフォルトで適用されている、いわゆる最適化だ。

この最適化は、バッファリングを行わずに一度にとても少量のデータを送信するようなアプリケーションに最も適している。この基準にあわない場合は、しばしばサーバーによって無効化されている。それでは、アルゴリズムを確認していこう。

1. ローカルバッファに TCP パケットを構成するのに十分なデータが存在する場合は、すぐにデータをすべて送信する。
2. ローカルバッファに保留中のデータがなく、受信側からも保留の通知がない場合には、すぐにデータを送信する。
3. 受信側から保留の通知を受け、かつ TCP パケットを構成するのに十分なデータがまだ無い場合には、ローカルバッファにデータを置く。

このアルゴリズムは、小さな TCP パケットをたくさん送信することを防ぐ。これはもともと、telnet のようなプロトコルとうまくやるために設計された。telnet では、一度に 1 つずつキーが入力され、一方でネットワークを

[*1] 訳注：https://ja.wikipedia.org/wiki/Nagle アルゴリズム

通して遅延なくそれぞれの文字が送信される。

もし、リクエストやレスポンスが 1 つの TCP パケットで包むには十二分に大きい、HTTP のようなプロトコルを扱っている場合には、このアルゴリズムは最後のパケット送信を遅らせる以外の効果はないだろう。Nagle アルゴリズムは、telnet を実装するといった、かなり特定の状況で自分の足を打ち抜かないようにするためのものだ。Ruby のバッファリングであったり、TCP 上の一般的なプロトコルを考えると、このアルゴリズムを禁止したいと思うかもしれない。

たとえば、Ruby で作られているすべての Web サーバーは、このオプションを禁止している。オプションを禁止するには、次のようにすると良い。

```
require 'socket'

server = TCPServer.new(4481)

# Nagle アルゴリズムを禁止する。サーバーに遅延なく送信するよう伝える。
server.setsockopt(Socket::IPPROTO_TCP, Socket::TCP_NODELAY, 1)
```

第 14 章

メッセージのフレーミング

サーバーとクライアントでやりとりするメッセージをどうやってフォーマットするかについて、まだ説明していなかった。

`CloudHash` が抱えている問題の 1 つは、クライアントがサーバーに何かコマンドを送りたいと思うたびに新しい接続を開く必要があるということだった。この主な理由は、クライアントとサーバーがメッセージをフレーミングする方法を持っていなかったことによる。そのため、メッセージの終わりを示すには EOF をフォールバックする必要があった。

これは技術的には「目的を達している」。けれど、理想的な実現の仕方ではない。コマンドを実行するたびに接続を開くとすると、不要なオーバーヘッドがかかる。また、同じ TCP 接続で複数のメッセージを送信することも確かに可能であるが、接続を開いたままにする場合には、メッセージの区切りを通知する何らかの方法が必要となる。

複数のメッセージに渡って接続を再利用するフレーミングというアイデアは、馴染み深い HTTP の `keep-alive` 機能と同じコンセプトだ。複数のリクエスト（フレーミングしたメッセージ）のために接続を開いたままにすると、新しい接続を開かないことでリソースを節約できる。

メッセージをフレーミングするための選択肢は文字通り無限にある。簡単なものもあれば、複雑なものもある。すべてはメッセージをどうフォーマッ

トしたいかによる。

> **プロトコル vs メッセージ**
>
> この章では、プロトコルとは異なるものとして、メッセージについて触れる。たとえば、HTTP プロトコルは、リクエスト行やヘッダなどを含むメッセージ内容のための約束事だけでなく、メッセージ境界（改行の種類）も定義している。プロトコルはメッセージをどうフォーマットすべきかを定義する。それに対して、この章は TCP ストリーム上でメッセージとメッセージをどう分けるかということを扱っている。

14.0.1 改行の使用

改行を使用することは、メッセージをフレーミングする最も簡単な方法だ。もし、クライアントとサーバーが同じオペレーティングシステム上で動いているなら、ソケット上で `IO#gets` や `IO#puts` を使うことで、改行付きのメッセージを送信できる。

それでは、`CloudHash` サーバーの該当する個所を書き換えてみよう。EOF ではなく、改行を使ってメッセージをフレーミングする形へと書き換えることにする。

```
def handle(connection)
  loop do
    request = connection.gets
    break if request == 'exit'
    connection.puts process(request)
```

```
  end
end
```

ここでは、loop を追加し、read から gets に変更している。この方法では、クライアントが「終了」要求を送るまでの間、サーバーはクライアントが望むように多くのリクエストを処理する。

より堅牢なアプローチは、IO.select を使って接続上のイベントを待つことだ。もし「終了」要求を送ることなしにクライアントソケットが切断をしたなら、現状のサーバーはクラッシュすることになるだろう。

クライアントは次のようにリクエストを送ることになる。

```
def initialize(host, port)
  @connection = TCPSocket.new(host, port)
end

def get
  request "GET #{key}"
end

def set
  request "SET #{key} #{value}"
end

def request(string)
  @connection.puts(string)

  # 応答を得るために新しい行を受信するまで読み込む。
  @connection.gets
end
```

クライアントがクラスメソッドを使わなくなっていることに注目してほしい。接続はリクエストを通して継続するようになっている。1 つの接続をインスタンス内にカプセル化して持っていて、そのオブジェクトのメソッドを呼び出せば良いようになっている。

改行とオペレーティングシステム

クライアントとサーバーが同じオペレーティングシステム上で実行されるのであれば、gets と puts を使えると説明したのを覚えているだろうか。その理由を説明していこう。

gets と puts のリファンレンスを読むと、デフォルトの改行文字に環境変数$/が指定されているとわかる。この変数には Unix では \n が格納されている。Windows の場合は\r\n だ。つまり、あるシステム上での puts は別のシステム上での gets と互換性がない。もしこの方法を使用する場合には、互換性を持つ改行文字を明示的に示して、メソッドの引数に渡すことを保証する必要がある。

現実世界において、フレームメッセージに改行を使用するプロトコルの 1 つが HTTP だ。次に示すのは短い HTTP リクエストの例だ。

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
\r\n
```

この例では、改行にはエスケープシーケンス\r\n が明示的に付けられている。この改行のシーケンスはオペレーティングシステムによらず、任意の HTTP クライアント／サーバーで遵守されなければならない

この方法でも確かに動く。しかし、やり方は 1 つじゃない。

14.0.2　コンテンツ長の使用

メッセージをフレーミングするやり方にはもう 1 つ、コンテンツ長を指定する方法がある。

この方法では、メッセージの送信者は最初に固定幅の整数表現でメッセージのサイズを示す。そして接続を通じて、メッセージを後ろにつけてそれを送信する。メッセージの受信者は、固定幅の整数を読み取ってメッセージのサイズを取得し、続いてそのサイズのバイト数を読み取り、メッセージを取得する。

CloudHash サーバーの関連する部分をこのやり方に変更すると、次のようになる。

```
# 固定幅整数型の大きさを取得する。
SIZE_OF_INT = [11].pack('i').size

def handle(connection)
  # メッセージ長は固定幅にパックされている。
  # それを読み込んでアンパックする。
  packed_msg_length = connection.read(SIZE_OF_INT)
  msg_length = packed_msg_length.unpack('i').first

  # 与えられた長さ分、メッセージをフェッチする。
  request = connection.read(msg_length)
  connection.write process(request)
end
```

クライアントがリクエストを送信する部分はこんな感じだ。

```
payload = 'SET prez obama'

# メッセージ長を固定幅整数にパックする。
msg_length = payload.size
packed_msg_length = [msg_length].pack('i')

# メッセージ長と、それに続けて
# メッセージ自体を書き込む。
connection.write(packed_msg_length)
connection.write(payload)
```

クライアントはバイト順序がビッグエンディアン[*1]の整数型としてメッセージ長をパックする。これによって、どの整数であっても同じバイト数にパックされることが保証される。この保証がないと、サーバーは 2 か 3 か 4 かそれ以上か、メッセージ長が何桁で表現されているのかを知ることができない。この方法を使用しているクライアントとサーバーはメッセージサイズとして常に同じバイト数を使用している。

コードをよく見るとわかるが、このメソッドでは、puts や gets といったラッパーメソッドは使用せず、read や write などの基本的な IO 操作だけを使用している。

[*1] http://ja.wikipedia.org/wiki/エンディアン

第 15 章

タイムアウト

タイムアウトとは、つまり忍耐のことだ。ソケットの接続を待てるのは一体どれくらいだろうか？ 読み込みには？ 書き込みには？

あなたが待てる範囲、それがこれらに対する答えだ。高性能なネットワークプログラムは、終わらない処理を待つことを大抵は望まない。最初の 5 秒でデータを書き込めなければ、ソケットはそれを問題と認識して、別の振る舞いへと処理を切り替えるだろう。

15.1 未使用のオプション

もし、Ruby のコードをいくらかでも読んだことがあれば、標準ライブラリに付属する `timeout` ライブラリを見たことがあるかもしれない。このライブラリは、Ruby によるソケットプログラミングでよく使われるものだ。けれど、ここではその話をするつもりは**ない**。もっとうまいやり方があるんだ！ `timeout` ライブラリは汎用的なタイムアウトの仕組みを提供する。しかし、オペレーティングシステムには、より高性能、より直感的なソケット固有のタイムアウトが付属している。

オペレーティングシステムはまた、ネイティブのソケット・タイムアウトも提供している。これは、`SNDTIMEO` や `RCVTIMEO` というソケットオプショ

ンを通して設定が可能だ。**しかし**、これは Ruby 1.9 からは機能しなくなっている。スレッド中でブロッキング IO を処理するため、Ruby は poll(2) 周りのすべてのソケット操作をラップしている。そのため、これらは Ruby からは使用できない。

他にどんな方法が残っているだろう？

15.2 IO.select

それじゃあ、古くて信頼できる `IO.select` を使うことにしようか。え？実際それはたくさん使われているんだ。

`IO.select` をどう使うかについては、前の章ですでに見た。ここでは、それをタイムアウトのためにどう使用するかを見ていこう。

```
require 'socket'
require 'timeout'

timeout = 5 # seconds

Socket.tcp_server_loop(4481) do |connection|

  begin
    # 最初の read(2) を初期化する。read(2) はソケット上で要求されるデータを必要
    ↪ とし、
    # 読み込み可能なデータが既にあるときは select(2) を回避する。
    connection.read_nonblock(4096)

  rescue Errno::EAGAIN
    # 接続が読み込み可能になったかどうかを監視する。
    if IO.select([connection], nil, nil, timeout)
      # IO.select は実際にはソケットを返すが、
      # 戻り値は気にしないでもよい。 nil を返さなかったという事実が
      # ソケットが読み込み可能であることを意味している。
      retry

    else
```

```
22        raise Timeout::Error
23      end
24    end
25
26    connection.close
27  end
```

ここでは、Timeout::Error を使用するために timeout を必要とした。

15.3 受付タイムアウト

前に、accept が IO.select とうまく動作するのを説明した。accept 周りでタイムアウトを行う必要がある場合も、read と同じようになる。

```
server = TCPServer.new(4481)
timeout = 5 # 秒

begin
  server.accept_nonblock
rescue Errno::EAGAIN
  if IO.select([server], nil, nil, timeout)
    retry
  else
    raise Timeout::Error
  end
end
```

15.4 接続タイムアウト

接続周りでタイムアウトを行う場合も、他の例と同様だ。

```ruby
require 'socket'
require 'timeout'

socket = Socket.new(:INET, :STREAM)
remote_addr = Socket.pack_sockaddr_in(80, 'google.com')
timeout = 5 # 秒

begin
  # google.com の 80 番ポートへのノンブロッキング接続を開始する。
  socket.connect_nonblock(remote_addr)

rescue Errno::EINPROGRESS
  # 接続の途中であることを示している。
  # ソケットが書き込み可能になるのを監視し、接続が完了したことを通知する。
  #
  # ブロックを再試行すると、EISCONN を捕捉するブロックへと飛び、
  # この begin ブロックの外側、ソケットを使用できる場所へと処理がうつる。
  if IO.select(nil, [socket], nil, timeout)
    retry
  else
    raise Timeout::Error
  end

rescue Errno::EISCONN
  # 接続が完全に完了したことを示している。
end

socket.write("ohai")
socket.close
```

こうした IO.select ベースのタイムアウトの仕組みは一般的に使われている。もちろん、Ruby の標準ライブラリ内でも同様だ。そして、ネイティブのソケットタイムアウトよりも高い安全性を提供している。

第 16 章

DNS ルックアップ

タイムアウトはコードを制御するうまいやり方ではある。しかし、それは制御できることが少ないことに起因する。

クライアント接続を例に取ろう。

```
1 require 'socket'
2
3 socket = TCPSocket.new('google.com', 80)
```

コンストラクタの中で Ruby が `connect` を呼び出すことは既に見た。ここでは IP アドレスではなくホスト名を渡している。つまり、ホスト名から接続できる固有のアドレスを解決するために、Ruby は DNS ルックアップを行う必要がある。

それがどうしたって？ 遅い DNS サーバーは Ruby のプロセス全体をブロックするかもしれない。これは、マルチスレッド環境下では残念な結果になる。

16.1 MRI と GIL

標準の Ruby 実装（MRI）はグローバルインタプリタロック（GIL、Global Interpreter Lock）と呼ばれる仕組みを持つ。GIL は安全のための仕組みだ。GIL は、Ruby インタプリタが危険を及ぼす可能性のある操作を一度に 1 つずつしか行わないことを保証する。これはまさにマルチスレッド環境下で使われる。1 つのスレッドがアクティブである間、**他のすべてのスレッドがブロックされる**。これによって、MRI ではより安全にシンプルなコードを書けるようになっている。

ありがたいことに、GIL はブロッキング IO を理解**する**。もしブロッキング IO（たとえば `read` によるブロックなど）を行っているスレッドがある場合は、MRI は GIL を解放して別のスレッドが実行を継続できるようになる。ブロッキング IO 呼び出しが終了して再度動作が可能となったスレッドは、実行を再開するための列へと並ぶ。

MRI は、C 拡張の実行を特別厳しく扱う。ライブラリが C 拡張の API を使用すると、GIL はその間他のコードの実行をブロックする。これはブロッキング IO が発生したとしても例外ではない。C 拡張の実行が IO でブロックされると、他のすべてのスレッドがブロックされる。

この辺で手の内を明かし、ここでのポイントを述べることにしよう。Ruby は DNS ルックアップのために C 拡張を使用している。したがって、DNS ルックアップが長時間ブロックしている場合、MRI は GIL を解放しないということになる。

16.2 resolv

ありがたいことに、Ruby は標準ライブラリでこの解決策を提供している。`resolv` ライブラリは、DNS ルックアップの純粋な Ruby 実装による代替を提供する。これによって、DNS ルックアップ時に長くブロックする

ような場合でも、MRI が GIL を解放できるようになる。マルチスレッド環境でも、これで大勝利だ。

resolv ライブラリは独自の API を持つ。けれど、標準ライブラリでは resolv を使うために Socket をモンキーパッチするライブラリも提供している。

```
require 'resolv' # ライブラリ
require 'resolv-replace' # モンキーパッチ
```

マルチスレッド環境下でソケットプログラミングを行う際には、これを使うことをお薦めする。

第 17 章

SSL ソケット

SSL は、公開鍵を使いソケット経由でデータを安全にやりとりする仕組みを提供する。

SSL ソケットは TCP ソケットを置き換えるものではない。しかし、簡素で安全ではない従来のソケットを、安全な SSL ソケットに「アップグレード」できる。いうなれば、TCP ソケットの上にセキュアなレイヤーを追加できるものだ。

TCP ソケットを SSL にアップグレードすることはできる。しかし、1 つのソケットで SSL と非 SSL の両方の通信ができるわけではない、ということに注意してほしい。SSL を使う際は、受信側とのエンドツーエンド通信はすべてを SSL を使って行う。そうでなければセキュアではない。

SSL 経由で通信するためには、従来の TCP ソケットと同様、2 つのポート (および 2 つのソケット) が必要となる。HTTP はこの一般的な例だ。HTTP 通信はデフォルトで 80 番ポートが使われる。これに対して、HTTPS（HTTP over SSL）通信はデフォルトでは 443 ポートを使って行われる。

どのような TCP ソケットでも SSL ソケットに変換できる。Ruby ではほとんどの場合、標準ライブラリに含まれる `openssl` ライブラリをそのために用いる。次に示すコードはその例だ。

```
require 'socket'
require 'openssl'

def main
  # TCP サーバーを作成する。
  server = TCPServer.new(4481)

  # SSL コンテキストを作成する。
  ctx = OpenSSL::SSL::SSLContext.new
  ctx.cert, ctx.key = create_self_signed_cert(
    1024,
    [['CN', 'localhost']],
    "Generated by Ruby/OpenSSL"
  )
  ctx.verify_mode = OpenSSL::SSL::VERIFY_PEER

  # TCP サーバー上に SSL ラッパーを作成する。
  ssl_server = OpenSSL::SSL::SSLServer.new(server, ctx)

  # Accept connections on the SSL socket.
  # SSL ソケット上で接続に応答する。
  connection = ssl_server.accept

  # 他の接続と同じように扱う。
  connection.write("Bah now")
  connection.close
end

# このコードは webrick/ssl から拝借した。
# このメソッドは Context オブジェクトで使用する自己署名 SSL 証明書を生成する。
def create_self_signed_cert(bits, cn, comment)
  rsa = OpenSSL::PKey::RSA.new(bits){|p, n|
    case p
    when 0; $stderr.putc "."  # BN_generate_prime
    when 1; $stderr.putc "+"  # BN_generate_prime
    when 2; $stderr.putc "*"  # searching good prime,
      # n = #of try,
```

```
38        # but also data from BN_generate_prime
39      when 3; $stderr.putc "\n" # found good prime, n==0 - p, n==1 - q,
40        # but also data from BN_generate_prime
41      else;   $stderr.putc "*"  # BN_generate_prime
42      end
43    }
44    cert = OpenSSL::X509::Certificate.new
45    cert.version = 2
46    cert.serial = 1
47    name = OpenSSL::X509::Name.new(cn)
48    cert.subject = name
49    cert.issuer = name
50    cert.not_before = Time.now
51    cert.not_after = Time.now + (365*24*60*60)
52    cert.public_key = rsa.public_key
53
54    ef = OpenSSL::X509::ExtensionFactory.new(nil,cert)
55    ef.issuer_certificate = cert
56    cert.extensions = [
57      ef.create_extension("basicConstraints","CA:FALSE"),
58      ef.create_extension("keyUsage", "keyEncipherment"),
59      ef.create_extension("subjectKeyIdentifier", "hash"),
60      ef.create_extension("extendedKeyUsage", "serverAuth"),
61      ef.create_extension("nsComment", comment),
62    ]
63    aki = ef.create_extension("authorityKeyIdentifier",
64                              "keyid:always,issuer:always")
65    cert.add_extension(aki)
66    cert.sign(rsa, OpenSSL::Digest::SHA1.new)
67
68    return [ cert, rsa ]
69  end
70
71  main
```

コードの一部では、自己署名 SSL 証明書を生成し、SSL 接続にそれを使用している。証明書は SSL におけるセキュリティの基礎となるものだ。こ

れが無いと、ただの安全ではない通信を使用しているに過ぎなくなってしまう。

同様に、verify_mode = OpenSSL::SSL::VERIFY_PEER という設定は、セキュアな接続には不可欠だ。多くの Ruby ライブラリでは、この値はデフォルトでは OpenSSL::SSL::VERIFY_NONE となっている。これは甘い設定と言わざるを得ない。この設定では、未検証な SSL 証明書を許す可能性がある。未検証の SSL 証明書が許されてしまうと、証明書が提供するセキュリティの多くが見かけだけになってしまう。この問題は、Ruby コミュニティの中でも長いこと議論されてきている[*1]。

それではサーバーを起動し、netcat を使って通常の TCP 接続を試してみよう。

```
$ echo hello | nc localhost 4481
```

サーバーは OpenSSL::SSL::SSLError と共にクラッシュするはずだ。これは期待通りの動作だ！

サーバーは安全でないクライアントからの接続を拒否し、そのため例外をあげた。サーバーを立ち上げ直し、今度は SSL で保護された Ruby クライアントを使用して接続してみることにしよう。

```
1 require 'socket'
2 require 'openssl'
3
4 # TCP クライアントソケットを作成する。
5 socket = TCPSocket.new('0.0.0.0', 4481)
6
7 ssl_socket = OpenSSL::SSL::SSLSocket.new(socket)
8 ssl_socket.connect
```

[*1] http://www.rubyinside.com/how-to-cure-nethttps-risky-default-https-behavior-4010.html

```
9
10 ssl_socket.read
```

今度は両方のプログラムが正常に終了したはずだ。今回は、サーバーとクライアントの間で正常に SSL 通信が行われた。

典型的なプロダクション環境の設定では、自己署名証明書は生成しないだろう。自己署名証明書は開発やテストでの使用にのみ適している。通常、証明書は信頼できる提供者から購入する。SSL 証明書提供者は安全な通信に必要な `cert` と `key` を提供する。そして、上記の例で自己署名証明書を使っていたところで、その証明書を使うことになる。

第 18 章

緊急データ

少し前に、TCP ソケットは順序付けられたデータストリームを提供している点を強調した。言い換えると、TCP データストリームはキューのようなものだ。たとえば、ソケット接続の端が接続にいくつかのデータを書き込む。これはキューにデータをプッシュするようなものだ。データはさまざまな段階（ローカルバッファ、ネットワーク転送、リモートバッファ）を経て移動していき、受信側のソケットによってこの「キュー」からポップされる。

このメンタルモデルは、一般的な TCP 通信に当てはまる。帯域外データとして参照される TCP 緊急データは、すでにあるデータをバイパスしてでも、キューの先頭にすべてのデータをプッシュすることを許す。それによって、できるだけはやく接続のもう一端に受信させられる。

そのためのメソッドが、`Socket` ライブラリでまだ登場していないメソッド、`Socket#send` だ。

`Socket#send` は、（`IO` から継承した） `Socket#write` を特殊化したようなメソッドだ。実際、引数なしで呼ばれた場合には、ただの `write` のように振る舞う。

```
# これらは同じように振る舞う。
socket.write 'foo'
socket.send 'foo'
```

write メソッドが任意の IO オブジェクトと一緒に使えるよう一般化されているのに対し、send メソッドはソケット**でだけ**動くように特殊化されている。この特殊化によって、Socket#send は「フラグ」という第 2 引数を受け取られる。この「フラグ」によって、緊急データのようないくつかのデータを send に指定できる。

18.1 緊急データの送信

では、実際に見ていこう。

```
1  require 'socket'
2
3  socket = TCPSocket.new 'localhost', 4481
4
5  # 標準の方法を使っていくつかのデータを送信する。
6  socket.write 'first'
7  socket.write 'second'
8
9  # いくつかの緊急データを送信する。
10 socket.send '!', Socket::MSG_OOB
```

緊急データを送信するために、send メソッドを呼び出した際に、フラグとして Socket::MSG_OOB 定数を渡している。OOB は帯域外（アウト・オブ・バンド、out-of-band）を意味している。

これで緊急データを送ったことになるが、受信側が緊急データを最初に受信するにはまだ十分ではない。つまり、送信側と受信側とで、これが機能するよう協力し合う必要がある。

18.2 緊急データの受信

次に、受信側が Socket#recv を使ってどのように緊急データを受信するかを示す。

```
require 'socket'

Socket.tcp_server_loop(4481) do |connection|
  # 最初に緊急データを受信する。
  urgent_data = connection.recv(1, Socket::MSG_OOB)

  data = connection.readpartial(1024)
end
```

緊急データを受信するには、送信側が緊急データを送る際に使用しているものと同じフラグを指定して Socket#recv を呼び出す必要がある。Socket#send がソケットに特化したデータを書き込む方法だったように、Socket#recv はソケットに特化したデータを読み込む方法だ。そしてもちろん、フラグを受け取ることもできる。

最初に「通常の」データが書き込まれているにも関わらず、そのデータの前に緊急データを受信できたことに注目してほしい。これが緊急データを使ってできることだ。また、緊急データを明示的に受け取っていることにも注目してほしい。もし recv を呼び出していなかったら、サーバーは緊急データに気がついてなかっただろう。言い換えると、もし緊急データを求めていなければ受信側はそれを受け取らないでいい。この場合の対処の仕方は既にやった通りだ。

もし緊急データが届いていなければ、connection.recv(1, Socket::MSG_OOB) 呼び出しは、Errno::EINVAL 例外をあげて失敗する。

18.3 制限

先ほどの例で、1 バイトだけの緊急データを送信していたことに気がついただろうか。**これは意図的だ**。TCP の実装は、1 度に 1 バイトだけ送信できる、という形で緊急データを限定的にサポートしている。緊急データを複数バイト送信した場合は、最後のバイトのみが緊急データとみなされる。それ以前のバイトは「通常の」TCP データストリームの一部として扱われる。

18.4 緊急データと IO.select

他と同じように、IO.select を使用することで緊急データ用のソケットを監視できる。しかし、重大な注意点がある。

IO.select の第 3 引数が帯域外データに関連する IO オブジェクトの配列だと言ったのを覚えているだろうか。そのときは次のような感じで使用していた。

```
1 for_reading = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
2 for_writing = [<TCPSocket>, <TCPSocket>, <TCPSocket>]
3
4 IO.select(for_reading, for_writing, for_reading)
```

読み込みを監視するソケットの配列を、第 3 引数に渡していることに注目してほしい。これは、もしそれらのソケットのいずれかが緊急データを受信した場合、それらは IO.select から返された第 3 引数の要素に含まれることを意味する。

やみくもに recv を呼び出さなくても、緊急データ用のソケットを監視できるという点で、これはとても素晴らしい。けれど、私の経験上では IO.select は緊急データを処理したあとでも、その緊急データがあると言い続けてくるんだ！「通常の」TCP データストリームの一部が処理されるまで、

だいたいの場合ローカルの recv バッファが空になるまでそれは継続する。このことは、すでに処理した緊急データを無視するような、いくつかの余分なエラー処理や状態管理を追加する必要があることを意味している。

この注意点と 1 バイト制約を考えると、緊急データは滅多には使用されない TCP の機能だ。Ruby の標準ライブラリ内（net/ftp 内）で唯一使われている箇所でも、間違って複数バイトの緊急データを送信しようとしている[*1]。

18.5 SO_OOBINLINE オプション

緊急データを扱うもう 1 つの方法は、通常のデータストリームの中でそれに注目するというシンプルな方法だ。帯域内でも帯域外データの受信を可能にするには、SO_OOBINLINE というオプションを使用する。これによって、緊急データは「通常の」データストリーム内に順序通りに結合される。このオプションを有効にすると、緊急データはもはや緊急データとしては扱われない。片方が書き込みを行ったのと同じ順序でキューから読み取られる。

このオプションを設定する方法を次に示す。

```
1  require 'socket'
2
3  Socket.tcp_server_loop(4481) do |connection|
4    # 「通常の」データ中で緊急データを受信する。
5    connection.setsockopt :SOCKET, :OOBINLINE, true
6
7    # 緊急データを検出すると読み込みを止めることに
8    # 気をつけること。
9    connection.readpartial(1024) #=> 'foo'
10   connection.readpartial(1024) #=> '!'
11 end
```

[*1] http://www.ruby-forum.com/topic/201519

この例では、もはや緊急データは最初には受信されていない。foo データが!データの前に受信される。

コードの中で read 一族が緊急データをどう認識しているかを示した。foo データと!データは 1024 バイトの制限内のデータに納まっているにもかかわらず、緊急データと遭遇したところで読み込みを止めてそこまでのデータを戻している。

このオプションは、送信側のソケットではなく受信側のソケットにだけ効果がある。

第 19 章

ネットワークアーキテクチャパターン

ここまでの章では、基礎と「知る必要のあること」についてカバーしてきた。ここからは、ベストプラクティスと現実世界の例を見ていく。ここまでの部分はリファレンス文書のようなものだ。何を探すかさえわかれば、機能の使い方や問題の解決方法を参照できる。

Ruby で FTP サーバーを作るのであれば、本書の前半は確かに役に立つはずだ。しかし、偉大なソフトウェアを作れるよう、あなたを導いてくれるわけではない。

構成要素についての知識は得たけれど、ネットワークアプリケーションの一般的な構築方法についてはまだ触れていない。並行処理はどう処理すべきだろうか？ エラー処理は？ 遅いクライアントに対処する最善の方法は何だろうか？ リソースをもっとも効率的に使うにはどうしたらよいだろうか？

ここからの章は、これらの質問に回答していくためにある。まず 6 つのネットワークアーキテクチャパターンを説明し、サンプルプロジェクトにそれらを適用していく。

19.1 ミューズ

単に図を使って抽象的に話すのではなく、実際のサンプルプロジェクトを異なるパターンを使って実装していきたい。パターン毎の違いに実際に飛び込んでいく必要があるんだ。

このため、FTP のサブセットを話すサーバーを書いていくことにする。なぜサブセットなのかって？ それは、ここでフォーカスしたいのはアーキテクチャのパターンであって、プロトコルの実装ではないからだ。なぜ FTP かって？ FTP ならわざわざクライアントを書かなくてもテストできるからだ。FTP クライアントはすでに数多く存在している。

あまり親しみがない人のために解説すると、FTP はファイル転送プロトコル（File Transfer Protcol）だ。FTP は 2 つのコンピュータ間でファイルを転送するために、TCP 上にテキストベースのプロトコルを定義している。

ご存知のとおり、FTP はファイルシステムをブラウズするようなものだ。FTP は 2 つの TCP ソケットを同時に使用する。1 つの「制御」ソケットが、サーバーとクライアント間で FTP コマンドとその引数を送信するために使われる。ファイル転送が行われるたびに、それとは別の新しい TCP ソケットが使用される。これは、転送が進行している間でも、制御ソケットによって FTP コマンドを継続して処理できるようにする素晴らしいハックだ。

FTP サーバーのプロトコル実装を次に示す。この実装では、接続ごとの個々のコマンド処理を `CommandHandler` クラスによってカプセル化している。同じサーバーに対する個々の接続は、それぞれ別の作業ディレクトリを持つ可能性がある。このクラスは、そうした個々の情報を扱う。

```
module FTP
  class CommandHandler
    CRLF = "\r\n"

    attr_reader :connection
    def initialize(connection)
      @connection = connection
    end

    def pwd
      @pwd || Dir.pwd
    end

    def handle(data)
      cmd = data[0..3].strip.upcase
      options = data[4..-1].strip

      case cmd
      when 'USER'
        # 匿名を受け付ける
        "230 Logged in anonymously"

      when 'SYST'
        # このシステムが何と言う名前か？
        "215 UNIX Working With FTP"

      when 'CWD'
        if File.directory?(options)
          @pwd = options
          "250 directory changed to #{pwd}"
        else
          "550 directory not found"
        end

      when 'PWD'
        "257 \"#{pwd}\" is the current directory"

```

```
38        when 'PORT'
39          parts = options.split(',')
40          ip_address = parts[0..3].join('.')
41          port = Integer(parts[4]) * 256 + Integer(parts[5])
42
43          @data_socket = TCPSocket.new(ip_address, port)
44          "200 Active connection established (#{port})"
45
46        when 'RETR'
47          file = File.open(File.join(pwd, options), 'r')
48          connection.respond "125 Data transfer starting #{file.size}
            ↪  bytes"
49
50          bytes = IO.copy_stream(file, @data_socket)
51          @data_socket.close
52
53          "226 Closing data connection, sent #{bytes} bytes"
54
55        when 'LIST'
56          connection.respond "125 Opening data connection for file list"
57
58          result = Dir.entries(pwd).join(CRLF)
59          @data_socket.write(result)
60          @data_socket.close
61
62          "226 Closing data connection, sent #{result.size} bytes"
63
64        when 'QUIT'
65          "221 Ciao"
66
67        else
68          "502 Don't know how to respond to #{cmd}"
69        end
70      end
71    end
72  end
```

このプロトコル実装では、ネットワークや並行性について多くは語らない。次の章以降でそれらを扱っていくことにする。

第 20 章

シリアル

最初に見ていくネットワークアーキテクチャパターンは、リクエスト処理の「シリアル *(Serial)*」モデルだ。FTP サーバーの実装を見ながら進めていこう。

20.1 説明

シリアルアーキテクチャでは、すべてのクライアント接続はシリアルに処理される。並行性を持たないため、複数のクライアントが同時に捌かれることはない。

このアーキテクチャの流れは単純だ。

1. クライアントが接続する。
2. クライアント／サーバーでリクエストとレスポンスをやりとりする。
3. クライアントが切断する。
4. 1 に戻る。

20.2 実装

```
require 'socket'
require_relative '../command_handler'

module FTP
  CRLF = "\r\n"

  class Serial
    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      loop do
        @client = @control_socket.accept
        respond "220 OHAI"

        handler = CommandHandler.new(self)

        loop do
          request = gets

          if request
            respond handler.handle(request)
          else
            @client.close
            break
          end
```

```
38          end
39        end
40      end
41    end
42  end
43
44  server = FTP::Serial.new(4481)
45  server.run
```

このクラスがネットワーク処理と並行性だけにしか責任を持っていない点に注目してほしい。具体的なプロトコルの処理については、CommandHandler のメソッドに任せている。パターンを詳しく見ていこう。最初の数行を取り上げる。

```
1   class Serial
2     def initialize(port = 21)
3       @control_socket = TCPServer.new(port)
4       trap(:INT) { exit }
5     end
6
7     def gets
8       @client.gets(CRLF)
9     end
10
11    def respond(message)
12      @client.write(message)
13      @client.write(CRLF)
14    end
```

これら 3 つのメソッドは、この種の実装ではお決まりだ。initialize メソッドでは、クライアント接続を受け付けるソケットを開く。

gets メソッドは、現在のクライアント接続へ gets を委譲する。区切り文字を明示的に渡していることに注目してほしい。これによって、区切り文字の異なるプラットフォーム間でも移植できるようにしている。

respond メソッドは、フォーマットされた FTP レスポンスを書き出す。message は整数のレスポンスコードと文字列のメッセージの組み合わせだ。キャリッジ・リターン\r とライン・フィード\n の組み合わせを受信すると、クライアントはレスポンスが完了していることがわかる。

```
def run
  loop do
    @client = @control_socket.accept
    respond "220 OHAI"

    handler = CommandHandler.new(self)
```

これが、このサーバープログラムのメインループだ。見て分かるように、すべての処理はこのメインループの中で行われる。

ループ中**唯一**の accept 呼び出しが、このメソッドの最初にある。ここで、initialize の中で初期化された@control_socket から接続を受け付ける。220 を返している箇所が、プロトコルを実装している部分だ。FTP では新しいクライアント接続を受け入れた後、必ず返事をする必要がある。

最後に行っているのは、この接続用の CommandHandler の初期化だ。CommandHandler は接続ごとのサーバーの状態（現在の作業ディレクトリ）をカプセル化する。サーバーはこの handler オブジェクトに受信したリクエストを渡し、適切なレスポンスを受け取る。

この部分が、このパターンでの並行性の障壁となる。ここを処理している間、サーバーは他の接続に応答**しない**。そのために、このパターンは並行性を持つことができない。他のパターンがこの部分をどう扱っているかを見ることで、ここの違いがより明らかになっていくだろう。

```
loop do
  request = gets

  if request
    respond handler.handle(request)
  else
    @client.close
    break
  end
end
```

ここは、シリアル実装した FTP サーバーの仕上げ部分だ。

ループに入り、明示的に区切り文字を指定して、クライアントソケットからリクエストを取得している。handler はこれらのリクエストが渡ってくると、クライアント向けに適切なレスポンスを作成する。

（FTP のサブセットしかサポートしていないとはいえ）これが完全に機能する FTP サーバーであるとすると、実際にこのサーバーを実行し、標準的な FTP クライアントからサーバーに接続し、次のような動作を行える。

```
$ ruby code/ftp/arch/serial.rb
```

```
$ ftp -a -v 127.0.0.1 4481
cd /var/log
pwd
get kernel.log
```

20.3 考察

開発者のニーズに**完全**に依存するため、各パターンの長所と短所を見極めるのは難しい。ここでは、それぞれのパターンが秀でている箇所と、それを実現するためのトレードオフについて説明することにとどめたい。

シリアルアーキテクチャが提供する最大の利点は、「シンプルさ」だ。状態のロックも共有もない。接続が混線するようなこともない。リソースの使用に関しても同様だ。1 つのインスタンスが 1 つの接続を処理することは、たくさんのインスタンスやたくさんの接続ほどにはリソースを消費しない。

明らかな欠点は、並行性を持たないということだ。後から来て待たされている接続は、たとえ現在の接続がアイドル状態のときでも処理されない。リクエストの中継に時間がかかっていたり、リクエストの送信中に一時停止している場合でも、接続が閉じられるまでサーバーは処理をブロックする。

このシリアル実装は、この後に続くもっと興味深いパターンのための基準となる。

第 21 章

コネクション毎のプロセス

これが、リクエストの並列処理を可能にする、本書で扱う最初のネットワーク・アーキテクチャだ。

21.1 説明

並行処理を実現するため、このアーキテクチャはシリアルアーキテクチャを少しだけ変更する必要がある。接続を受け付ける部分とソケットからデータを読み込む部分は変わらない。

変更するのは、接続を受け付けた後だ。サーバーは受け付けた新しい接続を処理するために、子プロセスを `fork` する。子プロセスは接続を処理し、そして終了する。

fork の基礎

`$ruby myapp.rb` を実行してプログラムを起動すると、コードをロードして実行する新しい Ruby プロセスが生成される。

プログラム中で fork をすると、実行時にさらに新しいプロセスを生成できる。fork の効果は、2 つのプロセスを完全なコピーとしてしまう点だ。新しく生成されたプロセスは子とみなされ、元のプロセスは親とされる。fork が終わると、必要に応じて別々の道を行ける 2 つのプロセスが手に入る。

これはとてつもなく便利だ。たとえば、接続を accept した後でプロセスを fork すると、子プロセスは自動的にクライアント接続のコピーを手にすることになる。したがって、余計なセットアップやデータの共有、ロックなどをせずに並列処理を開始できる。

それでは、処理の流れを明らかにしよう。

1. 接続がサーバーにやってくる。
2. サーバーの主プロセスが接続を受け付ける。
3. サーバーの主プロセスは新しい子プロセスを fork する。子プロセスはサーバープロセスの完全なコピーを持つ。
4. 子プロセスは接続に対する処理を継続し、それと平行して、サーバーの主プロセスはステップ 1 に戻る。

カーネルの恩恵により、これらのプロセスは並列で実行される。新しい子プロセスが接続を処理している間、親である元のプロセスは新しい接続を受け付け、それを処理する子プロセスを fork していく。

どの時点でも、常に 1 つの親プロセスが接続に受け付けるために待機している。また、複数の子プロセスによって、それぞれの接続が処理される。

21.2 実装

```
require 'socket'
require_relative '../command_handler'

module FTP
  class ProcessPerConnection
    CRLF = "\r\n"

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      loop do
        @client = @control_socket.accept

        pid = fork do
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = gets

            if request
              respond handler.handle(request)
            else
              @client.close
```

```
38                break
39              end
40            end
41          end
42
43          Process.detach(pid)
44        end
45      end
46    end
47  end
48
49  server = FTP::ProcessPerConnection.new(4481)
50  server.run
```

見てわかるように、ほとんどは元のコードのままだ。主な違いは、ループ中に fork の呼び出しが加わっていることだ。

```
1  @client = @control_socket.accept
2
3  pid = fork do
4    respond "220 OHAI"
5
6    handler = CommandHandler.new(self)
```

接続を受け付けると、サーバープロセスは直ちにブロック付き fork を呼び出す。新しい子プロセスは、ブロックで与えられた処理を実行して終了する。

これは、それぞれの接続が独立した個別のプロセスに処理されることを意味する。親プロセスはブロック内のコードは実行せずに処理を継続する。

```
1 Process.detach(pid)
```

最後に Process.detach を呼び出していることに気が付いただろうか。プロセスが終了したあと、親からその終了ステータスを尋ねられるまでは、そのプロセスは完全に掃除されない。前述のコードでは、親は子プロセスの終了ステータスを特に気にしていない。そのため、はやい段階で親プロセスから子プロセスをデタッチすることで、プロセスが終了した際に完全にリソースが掃除されることを保証している[*1]。

21.3 考察

このパターンには利点がいくつかある。まずはシンプルだということ。シリアル実装のはじめにほんの少しコードを追加するだけで、複数のクライアントを並行的に処理できる。

2 つ目の利点は、並行処理を行う際のハードルが低いことだ。fork が子プロセスにすべてのコピーを提供することは前述した。見た目も複雑でなく、競合状態やロックもないし、コードの分離もちょっとしたものだ。

このパターンの明らかな欠点は、fork にとっては嬉しいことだけれど、生成可能な子プロセスの数に上限がないことだ。相手が少数のクライアントなら、これは問題ないかもしれない。しかし、数千数百のプロセスを生成することになったら、システムはすぐにダウンしてしまうだろう。この懸念は、後の章で述べる「prefork」パターンを採用することで解決できる。

環境によっては、fork の使用が問題になるかもしれない。fork は Unix 系のオペレーティングシステムでのみしかサポートされていない。つまり、Windows や JRuby ではサポートされて**いない**。

他には、スレッドを使用するかプロセスを使用するかという議論がある。

[*1] もし子プロセスの生成やゾンビプロセスなどについてもっと知りたいと思ったなら、本シリーズの 1 冊目である『なるほど Unix プロセス』を読んでみてほしい。

これについては、次の章で実際にスレッドによる実現方法を見たあとで触れることにする。

21.4 実例

- shotgun
- inetd

第 22 章

コネクション毎のスレッド

22.1　説明

このパターンは、前の章の「コネクション毎のプロセス」パターンに**とても**良く似ている。違い？ プロセスを生成する代わりに、スレッドを生成するんだ。

スレッド vs プロセス

スレッドもプロセスも、どちらも並列実行を提供する。けれど、やり方はまったく異なる。どちらも銀の弾丸ではない。どちらを使用するかはユースケースに依る。

生成。スレッドを生成するコストはとても低い。プロセスの生成は、元のプロセスが持つすべてのものをコピーする。スレッドはプロセス単位で管理される。つまり、複数のスレッドを生成すると、それらはすべて同じプロセス内に存在することになる。スレッドはメモリをコピーするのではなく共有する。そのため、スレッドの生成

はとても高速だ。

同期。スレッドはメモリを共有する。そのため、複数のスレッドからアクセスされるようなデータについては細心の注意を払う必要がある。これは通常、ミューテックスやロックといったスレッド間のアクセスの同期化を意味する。プロセスはすべて独自のコピーを持っている。そのため、これについては気にする必要はない。

並列処理。どちらの提供する並列実行も、カーネルによって実装されている。MRI の並列スレッドで気をつけなくてはならない重要な点は、インタプリタは実行中のコンテキストに対して**グローバルロック**を使用しているという点だ。スレッドはプロセス毎に存在するので、すべてのスレッドは同じインタプリタによって実行される。そのため、たとえマルチスレッド処理を行っていたとしても、MRI では真の並列処理は実現することができない。JRuby や Rubinius 2.0 といった Ruby 実装では、この辺りの事情は異なっている。

プロセスでは、このことは問題にならない。プロセスがコピーされると、新しいプロセスには Ruby インタプリタのコピーも含まれている。そのため、グローバルロックは存在しない。MRI では、プロセスだけが真の並列処理を提供している。

並列処理とスレッドに関して、もう 1 つ。MRI は GIL を使っているものの、スレッドの実装はかなりスマートだ。第 16 章「DNS ルックアップ」で述べたように、Ruby では実行中のスレッドが IO でブロックした場合、他のスレッドが実行されるようになっている。

まとめると、スレッドは軽量、プロセスは重量ということになる。どちらも並行実行を提供していて、どちらにも適したユースケースがある。

22.2 実装

```
require 'socket'
require 'thread'
require_relative '../command_handler'

module FTP
  Connection = Struct.new(:client) do
    CRLF = "\r\n"

    def gets
      client.gets(CRLF)
    end

    def respond(message)
      client.write(message)
      client.write(CRLF)
    end

    def close
      client.close
    end
  end

  class ThreadPerConnection
    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def run
      Thread.abort_on_exception = true

      loop do
        conn = Connection.new(@control_socket.accept)

```

```
35          Thread.new do
36            conn.respond "220 OHAI"
37
38            handler = FTP::CommandHandler.new(conn)
39
40            loop do
41              request = conn.gets
42
43              if request
44                conn.respond handler.handle(request)
45              else
46                conn.close
47                break
48              end
49            end
50          end
51        end
52      end
53    end
54  end
55
56  server = FTP::ThreadPerConnection.new(4481)
57  server.run
```

このコードは、これまでの 2 つの実装例と微妙に違いがある。同じメソッドも多少あるものの、違った整理がされている。

```
Connection = Struct.new(:client) do
  CRLF = "\r\n"

  def gets
    client.gets(CRLF)
  end

  def respond(message)
    client.write(message)
    client.write(CRLF)
  end

  def close
    client.close
  end
end
```

このコードは、これまでの例にもあったお決まりのメソッドだ。しかし、サーバークラスで直接定義されておらず、Connection クラス中に定義されている。

```
def run
  Thread.abort_on_exception = true

  loop do
    conn = Connection.new(@control_socket.accept)

    Thread.new do
      conn.respond "220 OHAI"

      handler = FTP::CommandHandler.new(conn)
```

この箇所には、重要な違いが 2 つある。1 つ目の違いは、前の例でプロセスを生成していたところで、スレッドを生成していること。2 つ目の違いは、accept から返ったクライアントソケットを、Connection.new の引数

に渡していることだ。これによって、スレッドは専用の Connection インスタンスを手に入れられる。

スレッドを使う際に、この処理はとても重要だ。単純にインスタンス変数にクライアントソケットを割り当ててしまうと、それはすべてのスレッド間で共有されてしまうことになる。スレッドは FTP サーバーの共有インスタンス中で生成される。そうすると、インスタンスの内部状態をすべてのスレッドが共有できてしまう。

このことは、プロセスを使ったプログラミングとはまったく異なっている。プロセスでは、それぞれのプロセスがメモリ内容のコピーを持っている。内部状態の共有は、スレッドによるプログラミングが難しいと言われる理由の 1 つだ。経験上、スレッドを使ってソケットプログラミングを行う際は、従うべき簡単なルールがある。それは、各スレッドはそれぞれの接続オブジェクトを持つようにするということだ。そして、頭痛の種はそこに閉じ込めておくんだ。

22.3 考察

このパターンは、これまでのパターンと同じ利点を多く共有している。ほんの少しコードの変更が必要となり、理解しておかなければならないことが多少増えているだけだ。スレッドを使用すると、ロックや同期などの問題と向き合わなければならない。けれども、それぞれの接続は単独のスレッドによって処理されるので、そこまで心配する必要はない。

このパターンが「コネクション毎のプロセス」に勝っている点の 1 つは、スレッドがリソース的に軽量だということだ。そのため、プロセスよりも多く生成できる。このパターンは、プロセスを使った場合よりもたくさんのクライアントを並列で処理できる可能性がある。

けど、ちょっと待って。そこには MRI の GIL が立ちふさがることを忘れないようにしよう。最終的に、どちらのパターンも銀の弾丸ではない。それぞれについて検討し、試し、テストする必要がある。

このパターンは「コネクション毎のプロセス」パターンとデメリットを共にする。スレッド数はシステムをダウンさせるまで増やせる。サーバーの接続数が増加していくと、すべての有効なスレッド間で処理を切り換えようとして、システムがダウンする可能性がある。これは最大スレッド数を制限することで解決できる。詳しくは第 24 章「スレッドプール」で触れることにする。

22.4 実例

- WEBrick
- Mongrel

第 23 章

prefork

23.1 説明

このパターンは、既に見た「コネクション毎のプロセス」アーキテクチャを思わせる。

このパターンもまた、並列処理の手段としてプロセスを扱う。けれど、接続の到着ごとには子プロセスを生成しない。サーバーを起動した際、まだ接続が到着する前の段階で、プロセスの束を `fork` する。

ワークフローを確認しよう。

1. メインのサーバープロセスは接続を待機するソケットを作成する。
2. メインのサーバープロセスは子プロセスの群れを `fork` する。
3. それぞれの子プロセスは、共有しているソケット上の接続を受け付け、個々にそれらを処理する。
4. メインのサーバープロセスは子プロセスを監視する。

重要なコンセプトは、メインのサーバープロセスは接続を待機するソケットを開くが受け付けはしない、ということだ。あらかじめ決められた数の子プロセスを `fork` することで、それぞれの子プロセスは待機しているソケットのコピーを持つことになる。そして、子プロセスは待機しているソケット

に対して、accept を呼び出す。

このパターンの優れている点は、子プロセス間で接続の同期やバランスなどを考慮しなくて良いということだ。そうしたことはカーネルがうまくやってくれる。複数のプロセスが同じソケットのコピーを使って接続を受け付けようとすると、カーネルは負荷を分散し、その中の 1 つのコピーだけが特定の接続に応答することを保証してくれる。

23.2 実装

```
require 'socket'
require_relative '../command_handler'

module FTP
  class Preforking
    CRLF = "\r\n"
    CONCURRENCY = 4

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      child_pids = []

      CONCURRENCY.times do
```

```
27          child_pids << spawn_child
28        end
29
30        trap(:INT) {
31          child_pids.each do |cpid|
32            begin
33              Process.kill(:INT, cpid)
34            rescue Errno::ESRCH
35            end
36          end
37
38          exit
39        }
40
41        loop do
42          pid = Process.wait
43          $stderr.puts "Process #{pid} quit unexpectedly"
44
45          child_pids.delete(pid)
46          child_pids << spawn_child
47        end
48      end
49
50      def spawn_child
51        fork do
52          loop do
53            @client = @control_socket.accept
54            respond "220 OHAI"
55
56            handler = CommandHandler.new(self)
57
58            loop do
59              request = gets
60
61              if request
62                respond handler.handle(request)
63              else
64                @client.close
```

```
65                break
66              end
67            end
68          end
69        end
70      end
71    end
72  end
73
74  server = FTP::Preforking.new(4481)
75  server.run
```

この実装は、これまで見てきた 3 つの実装とはまったく異なっている。特徴的な 2 か所のコードについて、順に説明していく。

```
1   def run
2     child_pids = []
3
4     CONCURRENCY.times do
5       child_pids << spawn_child
6     end
7
8     trap(:INT) {
9       child_pids.each do |cpid|
10        begin
11          Process.kill(:INT, cpid)
12        rescue Errno::ESRCH
13        end
14      end
15
16      exit
17    }
18
19    loop do
20      pid = Process.wait
21      $stderr.puts "Process #{pid} quit unexpectedly"
22
```

```
23      child_pids.delete(pid)
24      child_pids << spawn_child
25    end
26  end
```

このメソッドは、CONCURRENCY の数だけ spawn_child メソッドを呼び出す所から始まっている。（ずっと下で定義されている）spawn_child メソッドは、新しいプロセスを実際に fork し、そのプロセス ID（pid）を返している。

子プロセスを生成した後、親プロセスは INT シグナル用のシグナルハンドラを定義している。INT シグナルは、たとえば Ctrl-C キーを入力したときなどにプロセスが受信するシグナルだ。ここでは、受け取った INT シグナルを単に子プロセスに転送している。子プロセスが親プロセスから独立して存在していると、たとえ親プロセスが死んだとしても勝手に生き続けてしまう。そのため、親プロセスは自分が終了する際に、自分の子プロセスもしっかりクリーンアップする必要がある。

シグナルを処理した後、親プロセスは Process.wait を持つループへと入る。これによって、親プロセスはいずれかの子プロセスが終了するまでブロックされる。Process.wait は終了した子プロセスの pid を返す。通常は子プロセスは終了しないように実装してあるので、これは異常終了したと判断できる。そのため、STDERR にメッセージを出力し、終了した子プロセスに代わる新しい子プロセスを生成する。

いくつかの prefork サーバー、とりわけ Unicorn [*1] は、子プロセスをより積極的に監視する役割を親プロセスに与えている。たとえば、リクエストの処理に長い時間かかっているかどうかを監視し、この場合は強制的にそのプロセスを終了して代わりの子プロセスを生成する。

[*1] http://unicorn.bogomips.org

```
def spawn_child
  fork do
    loop do
      @client = @control_socket.accept
      respond "220 OHAI"

      handler = CommandHandler.new(self)

      loop do
        request = gets

        if request
          respond handler.handle(request)
        else
          @client.close
          break
        end
      end
    end
  end
end
```

このメソッドの根幹部分には馴染みがあるだろう。今回、このコードは fork と loop に包まれている。子プロセスは accept を呼び出す**前**に fork されている。外側のループでは各接続を処理し、閉じて、新しい接続を処理する。こうして子プロセスは自身の接続受付ループを繰り返す。

23.3 考察

この優れたパターンには、語るべきポイントがいくつかある。

「コネクション毎のプロセス」アーキテクチャと比べて、「prefork」は接続毎に fork のコストを払う必要はない。プロセスの生成コストは安くないが、「コネクション毎のプロセス」だと接続の開始ごとに毎回そのコストを

支払う必要がある。

このパターンでは事前にすべてのプロセスを生成している。そのため、プロセスを生成しすぎるという問題を防げる。

このパターンが「コネクション毎のスレッド」に勝っている点は、完全な分離だ。それぞれの子プロセスは、Ruby インタプリタも含めてすべてのコピーを独自に持っている。これによって、あるプロセスの障害が他のプロセスに影響を与えることはない。スレッドではメモリ空間を同一プロセス内で共有することになるので、あるスレッドの障害が他のスレッドに影響を与える可能性がある。

「prefork」の欠点は、多くのプロセスを `fork` することで、サーバーがより多くのメモリを消費するという点だ。プロセスはけっして安くない。`fork` した各プロセスがすべてのコピーを持つことを考えると、メモリ使用量は `fork` したプロセスが親プロセスの 100% の大きさまで増加すると予測できる。

親プロセスが 100 メガバイトだとすると、4 つの子プロセスを `fork` するということは、500 メガバイトを占有することになる。そして、これで 4 つだけ同時接続が可能となる。

この点についてはこれ以上ここでは述べない。けど、このコードは実にシンプルでもある。コンセプトをいくらかは理解する必要があるが、総合的に見てシンプルだし、実行時にこんがらがるような心配もほとんどない。

23.4 実例

- Unicorn

第 24 章

スレッドプール

24.1　概要

このパターンでは、「prefork」パターンのスレッド版だ。サーバーの起動時にスレッドを一定の数だけ生成して、それぞれの独立したスレッドが接続を処理する。

アーキテクチャの流れは前の章と同じで、「プロセス」の代わりに「スレッド」を用いる。

24.2　実装

```
require 'socket'
require 'thread'
require_relative '../command_handler'

module FTP
  Connection = Struct.new(:client) do
    CRLF = "\r\n"

    def gets
      client.gets(CRLF)
    end
```

```
12
13      def respond(message)
14        client.write(message)
15        client.write(CRLF)
16      end
17
18      def close
19        client.close
20      end
21    end
22
23    class ThreadPool
24      CONCURRENCY = 25
25
26      def initialize(port = 21)
27        @control_socket = TCPServer.new(port)
28        trap(:INT) { exit }
29      end
30
31      def run
32        Thread.abort_on_exception = true
33        threads = ThreadGroup.new
34
35        CONCURRENCY.times do
36          threads.add spawn_thread
37        end
38
39        sleep
40      end
41
42      def spawn_thread
43        Thread.new do
44          loop do
45            conn = Connection.new(@control_socket.accept)
46            conn.respond "220 OHAI"
47
48            handler = CommandHandler.new(conn)
49
```

```
50            loop do
51              request = conn.gets
52
53              if request
54                conn.respond handler.handle(request)
55              else
56                conn.close
57                break
58              end
59            end
60          end
61        end
62      end
63    end
64  end
65
66  server = FTP::ThreadPool.new(4481)
67  server.run
```

ふたたび、2 つのメソッドが登場している。1 つはスレッドを生成するメソッド、もう 1 つは生成したスレッドの振る舞いをカプセル化したメソッドだ。スレッドを使うことになるため、ここではまた Connection クラスを使用する。

```
1   CONCURRENCY = 25
2
3   def initialize(port = 21)
4     @control_socket = TCPServer.new(port)
5     trap(:INT) { exit }
6   end
7
8   def run
9     Thread.abort_on_exception = true
10    threads = ThreadGroup.new
11
12    CONCURRENCY.times do
13      threads.add spawn_thread
```

```
14      end
15
16      sleep
17    end
```

run メソッドは ThreadGroup を生成し、すべてのスレッドを監視する。ThreadGroup はスレッド管理配列のようなものだ。ThreadGroup にスレッドを追加しておき、もし実行が終了したらグループから取り除く。

ThreadGroup#list を使うと、グループ内のすべてのスレッドのリストを取得できる。この実装では実際に使用していないが、もしすべてのスレッドを扱いたいような場合（たとえばそれらを join するなど）には、ThreadGroup は便利だ。

ここでは、前の章と同じように、CONCURRENCY 回数分 spawn_thread メソッドを呼び出している。CONCURRENCY の回数が「prefork」の時よりも多くなっていることに気がついただろうか。改めて言っておくと、スレッドはプロセスに比べると軽量だ。そのため、ここで CONCURRENCY の回数を増やせる。ただし、MRI GIL がメリットの一部を妨げることになることに留意してほしい。

このメソッドの最後では、終了を防ぐために sleep を呼び出している。プールが作業をしている間、メインスレッドはアイドル状態のままだ。理論的にはプールを監視することもできるが、ここでは終了しないように念のため sleep している。

```
def spawn_thread
  Thread.new do
    loop do
      conn = Connection.new(@control_socket.accept)
      conn.respond "220 OHAI"

      handler = CommandHandler.new(conn)

      loop do
        request = conn.gets

        if request
          conn.respond handler.handle(request)
        else
          conn.close
          break
        end
      end
    end
  end
end
```

このメソッドはとくに新鮮みはない。「prefork」パターンと同じだ。すなわち、接続を処理する処理を繰り返すスレッドを生成する。繰り返しになるが、カーネルは 1 つのスレッド中では 1 つの接続のみ accept することを保証する。

24.3 考察

このパターンで考察できることの多くは、これまでのものと同様だ。

スレッドかプロセスかのトレードオフ以外でいうと、このパターンは接続を処理するたびにスレッドを生成する必要がなく、変なロックや競合状態もないが、それでも並列処理を提供する。

24.4 実例

- Puma

第 25 章

イベント (Reactor)

25.1 概要

これまで見てきたパターンは、実際にはすべて「シリアル」パターンのバリエーションだった。すべてのパターンは同じ構造を持ち、「シリアル」パターン以外のパターンはそれをプロセスかスレッドで包んでいた。

ここで紹介するパターンは、他のパターンとはまったく異なるアプローチを取る。

25.2 実装

(Reactor パターン[*1] に基づく)イベントパターンは話題沸騰中だ。EventMachine や Twisted、Node.js、Nginx などのライブラリの核となっている。

このパターンは、シングルスレッド、シングルプロセスだ。にもかかわらず、これまで紹介してきたパターンと、少なくとも同程度の並行性を担保する。

このパターンは (Reactor と名付けられた) 接続多重化装置を中心に置く。接続のライフサイクルの各段階は個々のイベントへと分解され、各イベン

[*1] http://en.wikipedia.org/wiki/Reactor_pattern

トは交互配置されて任意の順で処理される。接続の各段階は accept、read、write、close といった単純な IO 操作となる。

接続多重化装置はイベントを監視し、イベントに関連付けられたコードを実行する。

それでは、ワークフローを見ていこう。

1. サーバーは接続を待機するソケットを監視する。
2. 新しい接続がやってくると、サーバーは監視するソケットのリストにその接続を追加する。
3. 接続を待機するソケットと同様に、サーバーはアクティブな接続を監視する。
4. アクティブな接続が読み込み可能だという通知を受けると、サーバーは接続からデータの固まりを読み込み、関連するコールバックを呼び出す。
5. アクティブな接続が**まだ**読み込み可能だという通知を受けると、サーバーは再び接続からデータの固まりを読み込み、関連するコールバックを呼び出す。
6. サーバーは別の接続を受信すると、監視するソケットのリストにその接続を追加する。
7. サーバーは最初の接続が書き込み可能だという通知を受けると、その接続にレスポンスを書き込む。

注目してほしいのは、これらすべてがシングルスレッドで行われるということだ。最初の接続が読み書きの途中であったとしても、サーバーは新しい接続を `accept` できる。

サーバーは各操作を小さい単位に分割している。そのため、複数の接続に対するさまざまなイベントを交互配置させられる。

そろそろコードの中身に入ってもよい時間だろう。

```
require 'socket'
require_relative '../command_handler'

module FTP
  class Evented
    CHUNK_SIZE = 1024 * 16

    class Connection
      CRLF = "\r\n"
      attr_reader :client

      def initialize(io)
        @client = io
        @request, @response = "", ""
        @handler = CommandHandler.new(self)

        respond "220 OHAI"
        on_writable
      end

      def on_data(data)
        @request << data

        if @request.end_with?(CRLF)
          # リクエストが完了した。
          respond @handler.handle(@request)
          @request = ""
        end
      end

      def respond(message)
        @response << message + CRLF

        # すぐに書き込めることを書き込む。
        # 残りはソケットが次に書き込み可能になったときに
        # リトライする。
        on_writable
```

```
38        end
39
40        def on_writable
41          bytes = client.write_nonblock(@response)
42          @response.slice!(0, bytes)
43        end
44
45        def monitor_for_reading?
46          true
47        end
48
49        def monitor_for_writing?
50          !(@response.empty?)
51        end
52      end
53
54      def initialize(port = 21)
55        @control_socket = TCPServer.new(port)
56        trap(:INT) { exit }
57      end
58
59      def run
60        @handles = {}
61
62        loop do
63          to_read =
          ↪  @handles.values.select(&:monitor_for_reading?).map(&:client)
64          to_write =
          ↪  @handles.values.select(&:monitor_for_writing?).map(&:client)
65
66          readables, writables = IO.select(to_read + [@control_socket],
          ↪  to_write)
67
68          readables.each do |socket|
69            if socket == @control_socket
70              io = @control_socket.accept
71              connection = Connection.new(io)
72              @handles[io.fileno] = connection
```

```
73
74            else
75              connection = @handles[socket.fileno]
76
77              begin
78                data = socket.read_nonblock(CHUNK_SIZE)
79                connection.on_data(data)
80              rescue Errno::EAGAIN
81              rescue EOFError
82                @handles.delete(socket.fileno)
83              end
84            end
85          end
86
87          writables.each do |socket|
88            connection = @handles[socket.fileno]
89            connection.on_writable
90          end
91        end
92      end
93    end
94  end
95
96  server = FTP::Evented.new(4481)
97  server.run
```

これまで見てきた実装とは異なるリズムに従っていることがわかるだろう。1 つずつ見ていく。

```
1  class Connection
```

この部分では、「イベント」を処理する Connection クラスを定義している。

スレッドを基盤にしたパターンでは、Connection クラスはスレッドと状態を分離するためのものだった。この例ではスレッドは使っていない。では、Connection クラスが必要な理由は何だろう？

プロセスを基盤にしたパターンはすべて、接続を互いに独立させるためにプロセスを使用していた。プロセスをどう使うかに関わらず、各接続は独立した 1 つのプロセスによって処理されていた。つまり、接続はプロセスとして表現されていた。

「イベント」パターンはシングルスレッドではあるけれど、複数の接続を並行に処理する。そのため、クライアントからの各接続はお互いの状態に踏み入らない、独自のオブジェクトを使った表現を必要とする。

```
1  class Connection
2    CRLF = "\r\n"
3    attr_reader :client
4
5    def initialize(io)
6      @client = io
7      @request, @response = "", ""
8      @handler = CommandHandler.new(self)
9
10     respond "220 OHAI"
11     on_writable
```

Connection クラスの最初の部分には、馴染みがあるものがいくつかある。

Connection オブジェクトは、実際の処理を行う IO オブジェクトを、@client インスタンス変数に格納する。そして、attr_accessor を使って、そのインスタンス変数に外からアクセスできるようにしている。

Connection オブジェクトは、これまでと同様、CommandHandler のインスタンスを初期化時に生成している。その後、FTP の必須の慣習である「hello」レスポンスを書き込む。しかし、クライアント接続に対して直接書き込むわけではなく、response 変数に単にレスポンス内容を設定している。次節で説明するが、これは Reactor がクライアントにデータ送信するためのトリガーとなる。

```
def on_data(data)
  @request << data

  if @request.end_with?(CRLF)
    # リクエストが完了した。
    respond @handler.handle(@request)
    @request = ""
  end
end

def respond(message)
  @response << message + CRLF

  # すぐに書き込めることを書き込む。
  # 残りはソケットが次に書き込み可能になったときに
  # リトライする。
  on_writable
end

def on_writable
  bytes = client.write_nonblock(@response)
  @response.slice!(0, bytes)
```

Connection クラスのこの部分では、Reactor コアと協調するライフサイクルのメソッド群を定義している。

たとえば、Reactor はクライアント接続からデータを読み込むと、そのデータを引数に on_data メソッドを呼び出す。メソッドの内部では、リクエストを完全に受信したかをチェックし、受信が完了していれば@handler

にレスポンスの構築を要求して、それを response に割り当てる。

on_writable メソッドは、クライアント接続に書き込む準備が完了すると呼び出される。このメソッドは@response 変数を扱う場所だ。このメソッドでは、@response の内容をクライアント接続に書き込める。書き込めたバイト数に応じて、@response をスライスして書き込めた部分を取り除く。

書き込めなかった@response の残りについては、次のタイミングで書き込まれる。もしすべてが書き込み済みの場合は、@response は空文字列でスライスされて何も書き込まれない。

残りの 2 つのメソッド、monitor_for_reading?と monitor_for_writing?は、読み書きのために特定の接続の状態を監視すべきかを、Reactor が問い合わせるためのメソッドだ。今回の例では、新しいデータが存在すれば常にそれを読み込みたいが、書き込む側で言えば、書き込まれたる@response があるかどうかだけを監視したい。@response が空の場合には、Reactor はクライアント接続が書き込み可能かを通知しない。

```
  def monitor_for_writing?
    !(@response.empty?)
  end
end

def initialize(port = 21)
  @control_socket = TCPServer.new(port)
  trap(:INT) { exit }
```

ここが Reactor コアのメイン処理部だ。

Hash オブジェクト@handles は、\{6 => #<FTP::Evented::Connection:xyz123>} のような構造をしていて、キーはファイルディスクリプタ番号、値は Connection オブジェクトとなる。

メインループの最初の行では、先ほど見たライフサイクルメソッドを使用している。アクティブな各接続に対して、読み込みや書き込みを監視すべき

かを尋ねている。そして、それに適した各接続の IO オブジェクトへの参照を取得している。

次に、Reactor は間髪いれずに、その IO オブジェクトを IO.select へと渡している。この IO.select 呼び出しは、監視しているソケットのいずれかが処理すべきイベントを受け取るまでロックする。

読み込みを監視するために、Connection オブジェクトの中に@control_socket を忍び込ませていることにも注目してほしい。これによって、新しいクライアント接続の到着を検出できる。

```
1  def run
2    @handles = {}
3
4    loop do
5      to_read =
       ↪  @handles.values.select(&:monitor_for_reading?).map(&:client)
6      to_write =
       ↪  @handles.values.select(&:monitor_for_writing?).map(&:client)
7
8      readables, writables = IO.select(to_read + [@control_socket],
       ↪  to_write)
9
10     readables.each do |socket|
11       if socket == @control_socket
12         io = @control_socket.accept
13         connection = Connection.new(io)
14         @handles[io.fileno] = connection
15
16       else
17         connection = @handles[socket.fileno]
18
19         begin
20           data = socket.read_nonblock(CHUNK_SIZE)
21           connection.on_data(data)
22         rescue Errno::EAGAIN
23         rescue EOFError
```

Reactor のこの部分は、IO.select から受け取ったイベントに適したメソッドを呼び出している。

はじめに、Reactor は「読み込み可能」とみなしたソケットを処理する。もし@control_socket が読み込み可能なら、それは新しいクライアント接続だということだ。なので、Reactor はそれを accept して新しい Connection オブジェクトを生成し、@handles に格納する。これによって、次のループからそれを監視できる。

次は、「読み込み可能」とみなしたソケットが、通常のクライアント接続であった場合の処理だ。この場合、データを読み込み、適切な Connection オブジェクトの on_data メソッドを呼び出す。読み込みがブロックされた場合（Errno::EAGAIN が発生する）は、特別なことはせず単にイベントを落とす。クライアントが切断された場合（EOFError が発生する）は、@handles からエントリを削除する。それによって、その接続用のオブジェクトは GC され、監視されないようになる。

最後の部分は、「書き込み可能」とみなしたソケットを、適切な Connection オブジェクトの on_writable メソッドを呼び出して処理している。

25.3 考察

このパターンは他のパターンとはまったく異なっている。そのため、他とはまったく異なった利点と欠点を持つ。

まず第一に、このパターンは数千、数万の数の同時接続を処理できるような、高い並行性を持つと評価されている。他のパターンはプロセスやスレッドに制約されているため、単純にはこれに近づくことはできない。

5000 の接続を処理するためにスレッドを 5000 個作成したなら、サーバーは停止してしまうことだろう。このパターンは、同時接続を処理する点においては他のパターンに楽々と勝利する。

このパターンの主な欠点は、それによって強制されるプログラミングモデルだ。その代わり、プロセスやスレッドを扱わないことで、モデルはずっと

シンプルになる。プロセスやスレッドを扱わないということは、共有メモリや同期、プロセスの暴走などに頭を悩ます必要がないということだ。しかし、シングルスレッドの内部ですべての並行処理が起きることになるため、1 つの重要なルールに従う必要がある。それは、Reactor を決してブロックしてはいけないということだ。

関連する実装を見ながら、これを説明していこう。`CommandHandler` クラスの内側を見てみる。FTP ファイル転送コマンド（`RETR`）を扱うときは、ソケットを開き、データを転送し、ソケットを閉じる。ここで重要なのは、このソケットは Reactor のメインループの外で使用されていて、Reactor はそれについて把握していないということだ。

ファイル転送を要求しているクライアントとの接続環境が遅かったとすると、それは Reactor にどんな影響を与えるだろうか。

すべてが同じスレッドで実行されるとすると、この遅い 1 つの接続によって Reactor 全体がブロックされる！ `Connection` オブジェクトのメソッドを呼び出すと、Reactor 全体がメソッドが返るまでブロックされてしまう。`on_data` メソッドは、`CommandHandler` に委譲される。そのため、クライアントにファイルを転送している間 Reactor 全体がブロックされる。その間は他のデータを読み込めないし、新しい接続も受け入れられない。

したいことをアプリケーションが迅速にできることは、とても重要だ。Reactor を使って遅い接続を処理するにはどうしたら良いのだろうか。答えはこうだ。Reactor そのものを使用するんだ！

もしこのパターンを使用しているなら、Reactor によってブロッキング IO が処理されないようにしなければいけない。この例で言うと、`CommandHandler` が使用するソケットは、`on_data` メソッドと `on_writable` メソッドが定義された `Connection` のサブクラス内にカプセル化する必要があることを意味する。

遅い接続へデータが書き込めるようになると、Reactor はしかるべき `on_writable` メソッドを呼び出す。すると、**ブロックせずに**クライアントへ書き込める。この方法なら、Reactor が遅いリモート接続を待っている間も、

その準備が整うまで他の接続の処理を継続できる。

まとめると、このパターンは明らかな利点をいくつか提供する。そして、ソケットプログラミングの抽象のいくつかを単純化する。一方、アプリケーションが行っている入出力はすべて見直す必要がある。遅いコードや IO をブロックする外部ライブラリなどを使ってしまうと、実にたやすく享受している利点のすべてが失われてしまうことになる。

25.4 実例

- EventMachine
- Celluloid::IO
- Twisted

第 26 章

ハイブリッド

ネットワークパターンのパートも、この章で最後だ。ここでは具体的なパターンは取り上げない。代わりに、ここまでに説明してきた 2 つ以上のパターンを使った、ハイブリッドなパターンのつくり方を取り上げる。

ここまで説明してきたアーキテクチャは、どの種類のサービス（ここまでの章では FTP を見てきた）にも適用できる。けれども、昨今は HTTP サーバーに対して注目があつまっている。これはびっくりするくらいの Web の流行によるものだ。Ruby コミュニティはこの Web の流行の最前線にいて、そこではさまざまな HTTP サーバーが公正なシェアを持っている。なので、この章で扱う実世界の例は、すべて HTTP サーバーとなっている。

それでは、いくつか例を見ていこう。

26.1 nginx

nginx [*1]は、C で書かれた高性能なネットワークサーバーだ。プロジェクトの公式サイトでは、1 台のサーバーで **1 万の同時要求**を処理できると主張している。nginx は、Ruby の世界でしばしば Web アプリケーション

[*1] `http://nginx.org`

サーバーの HTTP プロキシとして使用される。しかし、実際には HTTP や SMTP、さらに他のプロトコルも処理できる。

では、nginx はどうやって並行処理を実現しているのだろうか？

nginx は、その中核[*2]として「prefork」パターンを使用している。しかし、`fork` したそれぞれのプロセスの内側は「イベント」パターンとなっている。いくつかの理由から、これは理にかなった効果的な選択だ。

第一に、すべての生成コストは nginx が子プロセスを生成する起動時に支払われる。これによって、nginx は複数コアとサーバーリソースを最大限に活用できることが保証される。第二に、「イベント」パターンは何も生成せず、スレッドも使用しないという点で注目に値する。スレッドを使用する際の 1 つの問題は、カーネルがアクティブなスレッド間のコンテキストを切り替え、管理するためにかかるオーバーヘッドを要求されることだ。

nginx には高速な実行を可能にする機能が他にも多く詰まっている。それには C 言語でだけ行えるようなタイトなメモリ管理も含まれる。そして、それらの機能の中核には、これまでの章で説明してきたパターンがハイブリッドで用いられている。

26.2 Puma

Puma は「並行処理のために作られた Ruby Web サーバー[*3]」を提供する。Puma は、GIL **無し**の Ruby 実装（Rubinius もしくは JRuby）向けの次世代 Web サーバーとして設計された。Puma の README [*4] には、スレッドにおける GIL の影響を思い出させてくれる良い概要が記されている。

では、Puma はどうやって並行性を担保しているのだろうか？

Puma は高い並行性を実現するために**スレッドプール**を使用する。メインスレッドは常に新しい接続を `accept` し、接続を処理用のスレッドプール

[*2] `http://www.aosabook.org/en/nginx.html`
[*3] `http://puma.io`
[*4] `https://github.com/puma/puma#description`

にキューイングする。これは keep-alive を使用しない場合の HTTP 接続に関する話だ[*5]。Puma は HTTP keep-alive をサポートしているので、もし接続を処理した際の最初のリクエストで kept-alive を要求された場合には、Puma はこれを尊重して接続を閉じない。

しかしながら、Puma は単に接続を `accept` するだけではない。新しいリクエスト用に接続を監視し、それらをうまく処理しなくてはいけない。Puma はこれを 「イベント（Reactor）」パターンを用いて行う。keep-alive されている接続への新しいリクエストを受信すると、そのリクエストはまた処理用のスレッドプールにキューイングされる。

つまり、Puma のリクエスト処理は、常にスレッドプールによって行われる。この処理は、永続的な任意の接続を監視する Reactor によってサポートされている。

Puma は他の最適化にも満ちているが、nginx と同様に、その中核ではこれまでの章で説明したパターンがハイブリッドで用いられている。

26.3 EventMachine

EventMachine は、Ruby 界でイベント駆動 I/O ライブラリとして良く知られている。このライブラリは、高い安定性とスケーラビリティを提供するために「Reactor」パターンを使用している。内部は C で書かれているが、C 拡張として Ruby のインターフェイスも提供している。

では、EventMachine はどうやって並行性を実現しているのだろうか？

EventMachine は中核として「イベント」パターンを実装している。

EventMachine の中核は、多重接続のネットワークイベントを処理できる、シングルスレッドのイベントループになっている。EventMachine はまた、実行時間の長い操作や他を遅くするようなブロックする操作を猶予するための**スレッドプール**も提供している。

[*5] `http://en.wikipedia.org/wiki/HTTP_persistent_connection`

EventMachine は生成プロセスの監視やネットワークプロトコルの実装、その他多くの機能をサポートしている。この章で紹介したハイブリッドなアーキテクチャは、並行性を改善する方法の 1 つでしかない。

第 27 章

おわりに

これでソケットプログラミングについての基礎はわかったはずだ。Ruby やこれから出会うであろうプログラミング環境において、ここで得たことを活用できるはずだ。本書で示したのは、これからもずっと使い続けられる有効な知識だ。

本書を最後まで読んでくれてありがとう。本書が、あなたの仕事、そしてそれと共にある技術に対する深い理解への糧となることを願っている。私のメールアドレスは jesse@jstorimer.com だ。本書の内容やプログラミングのことについて、ぜひ気軽に連絡してほしい。

ソケットプログラミングについて理解したなら、次は Unix プロセスを使ったプログラミングを学ぶことで、その基礎を固めよう。

『なるほど Unix プロセス』は、Ruby プログラマに Unix プロセスを使ったプログラミングの内と外を教える唯一の書籍だ。Unix プロセスを使ったプログラミングは、一般的に C プログラマの領域だとみなされている。しかし、Ruby はプロセスを扱う低レベルのシステムコールへの強力なインターフェイスを提供している。

ファイルディスクリプタが何でどのように動くか。プロセスをどうやってデーモンにするか。有名な Ruby プロジェクトの内部がどう作られているか。『なるほど Unix プロセス』は、そうした多くのことを教えてくれるは

ずだ。

読者からはこんな声が届いている。

> 本書は今日の多くの開発者のギャップを埋めるものだ。Unix の基礎を学ぶために C 言語を書く必要はない。本書はそれが真実であることを証明する。
>
> デビット・ブライアント・コープラン

> 本書を読み終えたときには、恋に落ちていた！　本書は、Unix プログラミングがどれだけシンプルで強力なものであるか、そして自分がいかに無知であったかを気付かせてくれた。
>
> マーク・アンドレ・コウノイヤー

詳細は `http://tatsu-zine.com/books/naruhounix` で[*1]。ぜひ手に取って貰えると嬉しい。

[*1] 訳注：日本語版は `https://tatsu-zine.com/books/naruhounix` で購入できる。

なるほど TCP ソケット
Ruby で学ぶソケットプログラミングの基礎

2024 年 9 月 23 日　初版

著　者　Jesse Storimer
翻　訳　島田 浩二

Original English language title: 'Working With Tcp Sockets'. Published by Jesse Storimer. Copyright (C) 2012 Jesse Storimer.