
Rubyによる
クローラー開発技法
巡回・解析機能の実装と21の運用例
佐々木拓郎／るびきち 著
SB Creative

Rubyによる
クローラー開発技法

巡回・解析機能の実装と21の運用例

々木拓郎／るびきち著

3 Creative

Rubyによるクローラー開発技法

巡回・解析機能の実装と21の運用例

佐々木拓郎／るびきち 著

SB Creative

■本書サポートページ

本書内で紹介しているサンプルスクリプトは、本書サポートサイトからダウンロードすることができます。下記のURLをご参照ください。
また、本書をお読みになったご感想、ご意見など、お気づきになった点がございましたらお寄せください。

サポートサイト
http://isbn.sbcr.jp/80354/

はじめに

本書はRubyでクローラーを開発・運用できるようにするための本です。クローラーを使えば情報収集のスピードが速くなり、最新情報をいち早く入手できるようになります。本書を通じてクローラーを実際に開発・運用して、それを体感してください。

クローラーというのは、自動的にWebサイトを巡回して情報収集するプログラムのことです。Google botなどの検索エンジンロボットが代表的ですが、個人・ビジネスに関わらず、さまざまな場面で使われています。そう聞くと難しそうに感じるかもしれませんが、まったくそんなことはなく、小規模なクローラーはRuby初級者でも書けます。実は、ファイルから情報を抜き出す普通のRubyスクリプトに、たった1行加えるだけでクローラーに変身してしまいます。もしかしたら知らず識らずのうちにあなたもクローラーを作っていたのかもしれません。クローラーは身近な存在です。

とはいえクローラーを作るにはいくつか注意点が必要です。関連する法律を守る必要がありますし、一歩間違えたら相手サイトを攻撃することになってしまいます。なんといっても岡崎図書館事件でクローラー開発者が逮捕されました。そのことから、より一層礼儀正しいクローラーを作る技術が世に求められています。

第1章では小さなスクリプトをクローラー化することを通じて、クローラー開発の全体像を学んでいただきます。標準ライブラリのみで完結するのでお手軽です。

第2章ではさまざまな外部ライブラリ（Gem）を用いて本格的なクローラー開発に入っていきます。それぞれのライブラリの特徴を知ったうえで用途に応じて使い分けていきます。

第3章ではデータの解析の話です。手っ取り早く正規表現を使うか保守を考えて構文解析を使うかの選択について、日本語文字コードの扱い、自然言語処理をとりあげます。

第4章では高度なクローラーを作るにあたってぶつかる問題に対処します。データの保存方法、データベース活用、効果的な開発・デバッグ方法、Anemoneの高度な機能についてです。

第5章ではGoogle、Amazon、Twitter、Facebookなど実際のサイトを使ってクローラーを作成していきます。画像を収集したり、トレンドや株価などのビジネスに役立つ情報も扱います。

第6章ではクローラーの運用ノウハウを余すことなく紹介します。サーバ・クラウドの活用、サイト変更に伴うメンテナンス、その他高度な内容を含みます。

2014年8月　るびきち

Contents

Chapter 1 10分クローラーの作成

Chapter 2 クローラー作成の基礎

Chapter 6 クローラーの運用

Chapter 1

10分クローラーの作成

1-1 イントロダクション

クローラーとは、Webページから自動で情報収集するプログラムです。本章ではRubyの標準ライブラリのみで小規模なクローラーを作っていきます。

1-1-1　クローラーとは

クローラーとは、システムが自動的にWebページを巡回して情報を収集するプログラムです。クローラーとして最も有名なのは、Googleなどの検索エンジンです。

クローラーはビジネスの場面でも使われています。マーケティング分析では、掲示板やSNSの書き込みをクローラーで見て回ります。この時、商品名などのキーワードに引っかかる書き込みを自動で見つけます。SNSに自動ログインして情報収集するクローラーも存在します。

個人的用途であってもクローラーは使われます。特定のページを定期的にアクセスして更新を自動チェックしたり、ページの内容を整形して表示するプログラムも広い意味でのクローラーになります。それらを駆使すれば、情報収集の時間が短縮できます。フィードがなかった時代には、筆者も日記サイトの最新記事のみを切り出して自分にメールするスクリプトをcronで走らせていました。

本書ではクローラー開発の奥深くまで触れます。本章ではその入口として、ニュースサイトの記事一覧を取ってくる簡単なクローラーを作成することから始めます。本格的なクローラーであってもその根底となる技術は変わらず、「ページを取得→解析→抽出→加工→出力」という流れは同じです。本章では小さなプログラムを通して一連の流れを見ていくことにします。

1-1-2　Rubyとは

Rubyは1993年から開発されている国産オブジェクト指向スクリプト言語です。代表的な汎用スクリプト言語Perlに可読性の高い構文とシンプルかつ強力なオブジェクト指向を加え、Lisp風の味つけをしたものがRubyです。1行から使えるスクリプト言語の世界にオブジェクト指向を導入したため、オブジェクト指向が一気に身近なものになりました。Rubyは本当にうまく設計されており、プログラミングを楽しいものにしてくれます。

RubyはPerl同様テキスト処理が大得意です。そのため、Webの世界でRubyは広く使われています。クローラーもWebを扱うものなのでRubyに向いています。

Rubyは標準で付属するライブラリに加え、Ruby固有のライブラリパッケージであるGemも充実しているので、目的を手早く達成できます。クローラーの作成においても、Nokogiriという強力なHTML・XMLパーサー、Anemoneというクローラーフレームワークが Gemに存在しているので、Rubyでクローラーを作ることは難しくないです。本章はあくまで導入部なので、Gemを使わずに標準ライブラリだけで幅広い処理ができることをお見せしましょう。

1-2 クローラー「GNU Wget」

実際にRubyでクローラーを作成する前に、「GNU Wget」を通じてクローラーとは何をするソフトウェアなのかを見ていきましょう。

1-2-1 Wgetとは

本書はRubyでクローラーを作成する方法を学ぶための本です。その前に、既に優秀なクローラーソフトウェアが存在するので、それを先に紹介しておきたいと思います。新しく作らなくても、このソフトウェアを使うか、そこから得られる実行結果を加工するだけで目的を果たせるかもしれません。

そのクローラーとは「GNU Wget」(以下、Wget)と言います。HTTP・FTP経由で自動ダウンロードを行うソフトウェアです。UNIX系OSはもちろんのこと、MacやWindowsでも使えます。Wgetはダウンローダとして有名で、技術系のブログを読んでいれば、インストールの説明などでしばしば登場してきます。

Wgetはただのダウンローダではなく、再帰ダウンロード(→p.9)やリンク変換(→p.10)ができる立派な「クローラー」です。サイトの内容を丸ごとローカルにダウンロードすることが簡単にできます。その際、リンク変換機能を使えばオフラインでも問題なくダウンロードしたサイトのリンクを渡り歩けます。拡張子指定を使えば画像・動画ファイルのみを集められます。再度実行する時に同じファイルをダウンロードしないような設定もできます。クローラー避けのための設定が書かれたrobots.txt(→p.31)を遵守し、ダウンロード間隔を空けられるなど、クローラーとしての礼儀もわきまえています。

そういうわけで、Wgetとはどのようなものなのかを見ていきましょう。その後でオリジナルクローラーを開発するか、Wgetの実行結果を処理するかを決断してください。

1-2-2　インストール

Wgetは定番かつ小規模なプログラムなのでインストールは難しくありません。

GNU/Linux（以下、Linux）では既にインストールされていることが多いです。インストールされていなければディストリビューション付属のパッケージシステムでインストールしてください。インストールされているかどうかはwhichコマンドで調べることができます。コマンド実行でパスが出力されていれば使えます。

以後、プロンプト「$」から始まるコマンドラインはLinux（UNIX）のシェルでの表記です。パスの設定などは、各自の環境に合わせて変更してください。なお、一部の出力結果はページに収まるように整形してあります。

◆ Wgetの確認

```
$ which wget
/usr/bin/wget
```

Wget

Macではインストールする必要がありますが、Homebrewなどのパッケージシステムを使えば簡単です。インストール後の確認はLinux同様にwhichで行えます。

◆ Wgetのインストール（Mac/Homebrew）

```
$ brew install wget
```

Linux・Macともにソースコードからコンパイルできます。執筆時点（2014年7月）では最新版は1.15なので、以下のコマンドを実行します。なお、MacではダウンローダcURLが最初からインストールされています。

◆ Wgetのインストール（ソースコードからコンパイル）

```
$ curl -O http://ftp.gnu.org/pub/gnu/wget/wget-1.15.tar.gz
$ tar zxvf wget-1.15.tar.gz
$ ./configure --with-ssl=openssl
$ make
$ sudo make install
```

Windowsではかなりバージョンが古く（1.11.4）なりますが（2014年7月現在）、バイナリーインストーラかChocolateyを使うのが簡単です。Cygwin環境ならば、デ

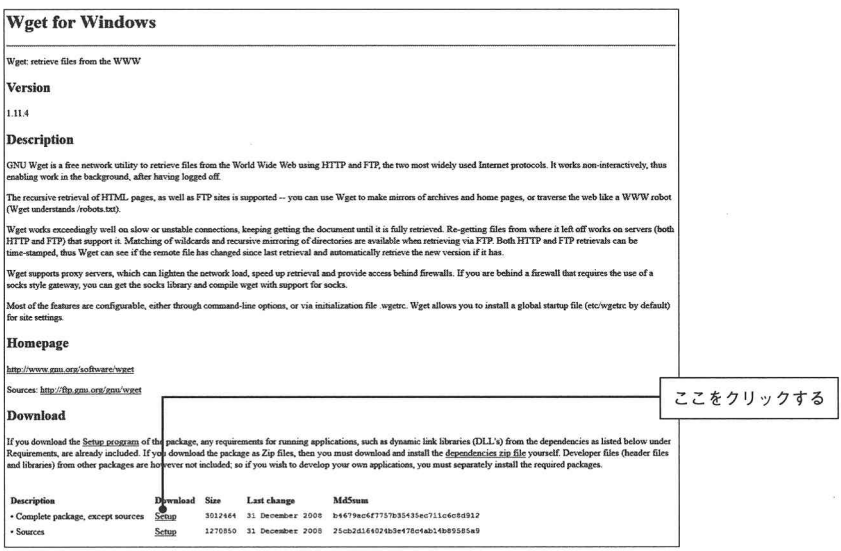
フォルトでインストールされます。

バイナリーインストーラは、以下のダウンロードサイトで「Complete package, except sources」を選択して実行します。

■ Wgetのダウンロードサイト（Windows）

URL **http://gnuwin32.sourceforge.net/packages/wget.htm**

Wgetのダウンロード（Windows）



Setupウィザードが出てくるので、そのままインストールしてください。

Wgetのインストール

インストール先を「C:¥wget」にした場合、「wget.exe」のパスは「C:¥wget¥bin¥wget.exe」になります。その時、「C:¥wget¥bin」を環境変数PATHに加えるとよいです。コマンドプロンプトからC:¥wget¥bin¥wget.exe --versionを実行してバージョンが出力されれば、正常にインストールされています。

◆ Wgetの確認（Windows）

```
C:¥work> C:¥wget¥bin¥wget.exe --version
SYSTEM_WGETRC = c:/progra~1/wget/etc/wgetrc
syswgetrc = c:¥wget/etc/wgetrc
GNU Wget 1.11.4
```

Chocolatey

Chocolateyは待望のWindowsパッケージシステムです。Windowsでの環境構築を劇的に簡単にします。

■ Chocolatey
URL http://chocolatey.org/

LinuxのAPTのように依存関係も解決してくれるし、登録パッケージ数も1,500を超えています。Wget以外にもRuby、MySQL、Git、Cygwinなどが登録されているので、開発者ならばぜひとも導入するべきです。

下記のコマンド（本家サイトから引用）をコマンドプロンプトから実行すればChocolateyをインストールできます。

◆ Chocolateyのインストール

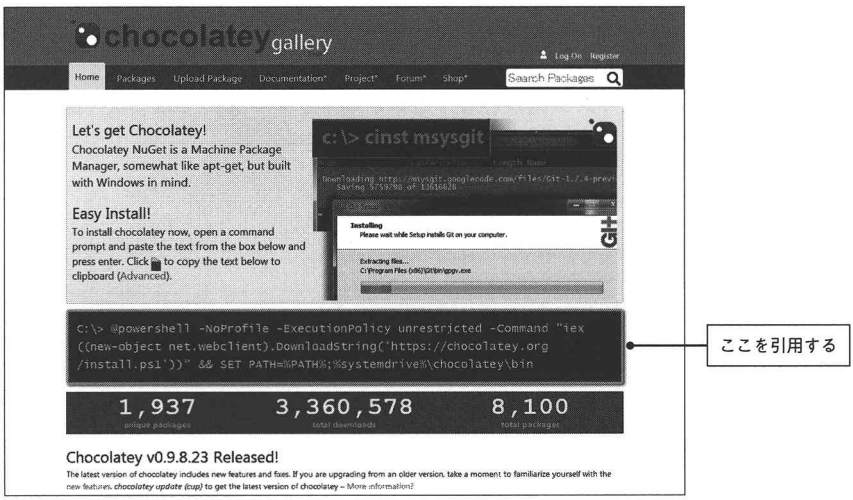
```
C:¥work> @powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.
webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH%;%systemdrive%
¥chocolatey¥bin
```

一度Chocolateyをインストールしたら後は簡単です。cinstコマンドでWgetをインストールできます。

◆ Wgetのインストール（Windows/Chocolatey）

```
C:¥work> cinst Wget
```

Chocolateyのサイト



1-2-3 Wgetの簡単な使い方

ダウンローダとしてのWgetはとても簡単に使えます。コマンドライン引数にURLを指定するだけです。例えば、初期のRubyのソースコードをダウンロードするには次のように実行します。

◆ **Wgetによるダウンロード**

```
$ wget http://ftp.ruby-lang.org/pub/ruby/1.0/ruby-0.49.tar.gz
```

すると、カレントディレクトリに「ruby-0.49.tar.gz」というファイル名でダウンロードされます。

ダウンロードするファイル名はデフォルトではURLからドメインとディレクトリを取り除いたものになりますが、-Oオプションで指定すれば任意のファイル名にできます。-O-と指定すれば標準出力に書き出します。

パイプでGNU tarの標準入力に渡してダウンロードしながらアーカイブを展開する応用例も定番です。その場合は、中間ファイルは作成されません。

◆ 展開しながらダウンロードする

```
$ wget -O- http://ftp.ruby-lang.org/pub/ruby/1.0/ruby-0.49.tar.gz | tar xzvf -
```

ダウンロードせずにリンク先が存在するかどうかをチェックするには--spiderオプションを指定します。ファイルサイズも確認できます。

◆ リンク先の確認

```
$ wget --spider http://ftp.ruby-lang.org/pub/ruby/1.0/ruby-0.49.tar.gz
Spider mode enabled. Check if remote file exists.
--2014-06-25 19:31:00--  http://ftp.ruby-lang.org/pub/ruby/1.0/ruby-0.49.tar.gz
Resolving ftp.ruby-lang.org... 221.186.184.75
Connecting to ftp.ruby-lang.org|221.186.184.75|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 214390 (209K) [application/octet-stream]
Remote file exists.
```

主なオプション

オプション	機　能
-O FILE	ダウンロードするファイル名を指定
--spider	ダウンロードせずに存在をチェック
-o FILE	メッセージをログファイルに出力
-q	メッセージを抑制する（quiet）
-nv	よけいな出力をしない
-i FILE	ファイルからURLを読み込む
-t N	再試行の回数を指定
-c	継続ダウンロード
--user USER	ユーザー名を指定
--password PW	パスワードを指定
--referer URL	refererを指定
--no-proxy	プロキシを使わない
--header STR	ヘッダ行を指定

1-2-4　クローラーとしてのWget

先ほどは普通のダウンローダとしてのWgetを紹介しましたが、ここからはクローラーとしてのWgetの使い方を紹介します。Wgetがクローラーであると言えるのは、再帰ダウンロード機能があるからです。

再帰ダウンロードとは、ページ内で指定されたURLのリンクを辿って、リンク先をもダウンロードすることです。これを使えばページをローカルに保存してオフラインで読めます。画像や音声ファイルを全自動で収集することもできます。もちろん、サイトを丸ごとミラーリングすることも可能です。

再帰ダウンロードを行うには、ダウンロードしたHTMLを解析してリンクを抽出する作業が含まれるので、Wgetはまさしくクローラーなのです。再帰ダウンロードにおける最重要オプションは、「-r」と「-l」です。

再帰ダウンロードの実行

-rは再帰ダウンロードを行うことを指示し、-lはどの階層（最初のページからリンクを辿る回数）まで潜るかを指定します。-rのみを付けた時は、-l5（再帰レベル5）が自動的に付けられます。しかし5階層も潜るのはかなり深いので、-rと-lはセットと覚えてください。-linf（無限階層）を指定すれば、そのサイトの最深部まですべてダウンロードできます。

-l1（再帰レベル1）は指定されたURL内のリンク先もダウンロードしますが、リンク先のページで指定されたリンク先まではダウンロードしません。多くの場合、目次ページを-l1で再帰ダウンロードするだけでこと足ります。

以下の例は、「http://www.example.com/」内のリンク先もダウンロードします。後ほどテストサーバを使ってオプションを検証していきますので、今は実行する必要はありません。

◆ リンク先のダウンロード

```
$ wget -r -l1 http://www.example.com/
```

ダウンロード間隔の設定

再帰ダウンロードはアクセスを集中させるので、適度にウェイト（待機時間）を入れることも大事です。サイトに対する攻撃と見なされないように、ゆっくりダウンロードすることが礼儀正しいクローラーの作法です。それを実現するオプションが、「-w」と「--random-wait」です。

-wは次のダウンロードに入る前に秒数だけ待ちます。-wだけを指定した時は等間隔でウェイトが入りますが、--random-waitを組み合わせると待ち時間をランダムにできます。--random-waitでの待ち時間は、-wで指定した値の0.5 ～ 1.5倍です。

以下の例は、ダウンロードの間隔を1 ～ 3秒空けます。

◆ 1 ～ 3秒間隔でダウンロードする

```
$ wget -r -l1 -w2 --random-wait http://www.example.com/
```

拡張子を指定してダウンロードする

-Aを指定すると、特定の拡張子のファイルのみをダウンロードできます。画像・音声・動画ファイルの収集もWgetでできます。

以下の例は、jpg/png/gifファイルを、間隔を1 ～ 3秒空けてダウンロードします。

◆ 画像ファイルをダウンロードする

```
$ wget -r -l1 -w2 --random-wait -A jpg,png,gif http://www.example.com/
```

コマンドを実行すると、カレントディレクトリにドメイン名のディレクトリが作成され、そのなかに画像ファイルがダウンロードされます。

Wgetの再帰ダウンロードはデフォルトでは同一ドメインに限定されています。つまり、外部サイトまでクロールしないようになっています。しかしそれでは困る場合があります。例えば画像用に別サーバを用意しているケースがあるからです。「images.example.com」に画像を置いている場合、上のコマンドでは画像はダウンロードされません。

この場合、-Hオプションで他のドメインもクロールできるようにする必要があります。

◆ 他のドメインからもダウンロードする

```
$ wget -r -l1 -w2 --random-wait -A jpg,png,gif -H http://www.example.com/
```

リンクの変換

サイトを丸ごとダウンロードして、オフラインで閲覧したい場合にもWgetは有用です。その場合は「-p」と「-k」オプションを使います。これらを使わないとオフライン閲覧に必要な素材が不足するなど、リンクが辿れなくなる問題が生じてしまいます。

-pはそのHTMLを表示するのに必要な画像・音声やCSSなどもダウンロードします。それらが別ドメインであってもダウンロードしてくれます。-r (→p.9) を使わ

ない場合は、そのURLのみが対象になります。

-kはリンク変換機能です。ハイパーリンクなどでURLを指定する箇所で、絶対パスから相対パスに変換します。例えば「http://www.example.com/1.html」にて「/2.html」へのリンクがある時、オンライン上であれば問題なく「http://www.example.com/2.html」へジャンプしてくれます。しかし、オフライン上では閲覧しているコンピュータのルートディレクトリの「2.html」を見てしまうため、リンクが辿れなくなってしまいます。そこで-kを使うと、「/2.html→2.html」と変換してくれるので、問題なくリンクが辿れるようになります。リンク変換の対象はハイパーリンクにかぎらず、インラインイメージ、CSSなどへのリンクも含みます。

リンク変換はすべてのダウンロードが終わった後で行われます。なぜなら、Wgetがどのリンク先をダウンロードしたのかを知っておく必要があるからです。

以下の例は、「http://www.example.com/」をダウンロードし(「index.html」がダウンロードされます)、表示に必要な素材もダウンロードします。

◆ サイトを丸ごとダウンロードする

```
$ wget -p http://www.example.com/
```

以下の例ではダウンロードの間隔を1 ～ 3秒空け、各リンク先および表示に必要な素材もダウンロードし、最後にリンク変換処理を施します。「index.html」直下のオフラインブラウジングが保証されます。

◆ ダウンロードとリンク変換

```
$ wget -r -l1 -w2 --random-wait -p -k http://www.example.com/
```

Wgetもれっきとしたクローラーであることが理解できたでしょうか？ クローラーは短時間に大量のリクエストを送り、通信量も多くなるため、悪用すれば攻撃にもなるので挙動をしっかり理解して、注意して実行する必要があります。

挙動を理解するためにWgetを試しに実行する必要がありますが、かといって試運転のために他のサイトに迷惑はかけられません。その解決方法を後ほど紹介します。

再帰ダウンロード関係のオプション

オプション	機　能
-r	再帰ダウンロードを有効にする
-l	階層の指定
-np	親ディレクトリに遡らない
-w N	ダウンロードごとにN秒待つ
--random-wait	-wで指定した数の0.5 ～ 1.5倍の時間待つ
-A LIST	拡張子で指定したファイルのみ取得
-R LIST	拡張子で指定したファイルを除外
-H	他のドメインもクロールする
-D LIST	-Hでクロールするドメインを指定
-p	URLの表示に必要な素材も取得する
-k	相対URLに変換する
-nd	ディレクトリを作成しない
-x	-rでなくてもディレクトリを作成する
-P PREFIX	PREFIXで指定したディレクトリに保存する
-N	タイムスタンプが新しいものを取得する
-m	ミラー。「-r -N -linf --no-remove-listing」と等価

1-3 クローラーを作るにあたってのRubyの基礎

Rubyとはどういう言語なのかをかけ足で見ていきます。そして、クローラー作成において重要とされる文字列・ファイル処理に触れます。

1-3-1　Rubyの特性

本節ではRubyの基礎について軽く触れておきます。

クローラーを作るにあたってのRubyの基礎と言っても、本当に基礎の知識があるだけで作れてしまいます。確かに複雑なクローラーはHTMLパーサーやクローラーフレームワークを使った方がよいのですが、簡単なものならばテキスト処理とたいして難度が変わらないです。ファイルを開くopenメソッドの引数にURLが使えるように拡張するライブラリが存在し、ファイル名のかわりにURLを指定すればHTTPアクセスができてしまいます。その状態のopenはHTTPクライアントであり、ファイルを開いて内容を読み取って加工して出力するごく普通のRubyスクリプトがクローラーに変身します。

また、取得したHTMLから情報を抽出して加工出力する処理はまさにテキスト処理そのものです。そのため、文字列や配列やハッシュなどの基本的なオブジェクトの扱い方がわかればなんとかなります。

Rubyのオブジェクト指向

Rubyはオブジェクト指向スクリプト言語です。ですが、ちょっとしたRubyプログラミングをするのにクラスを定義する必要はありません。豊富な組み込みクラスが用意されており、組み込みクラスを使うだけでもある程度プログラミングできてしまいます。

組み込みクラスを使うことでもメソッド呼び出しが起きるので、たとえクラスを定義しなくてもれっきとしたオブジェクト指向プログラミングです。本書で作成する最初のクローラーは関数的メソッドを数個定義するスクリプトですが、後のリファクタリングでクラスを定義します。

Rubyはすべてのデータがオブジェクトです。ユーザー定義クラスのオブジェクト以外に、整数や文字列もオブジェクトです。クラスさえもオブジェクトとして扱えます。

オブジェクト指向プログラミングにおいて重要なのは、自分自身のことをあまり語らないことです。内部構造をなるべく隠蔽して、外に公開するのは最低限に抑えることです。Rubyプログラミングは、正しいオブジェクト指向へ自然と誘導してくれます。

Rubyのメソッド呼び出し

オブジェクトのインスタンス変数は、デフォルトでは外からは(簡単には)見えない状態になっています。外にインスタンス変数を公開するには、インスタンス変数の値を返す・設定するメソッド(アクセサ)を定義します。アクセサによって、「オブジェクトにアクセスするにはメソッド呼び出しで」と一貫した姿勢になります。

Rubyでのメソッド呼び出しは、括弧を書く必要がありません。そのため、アクセサなど引数を持たないメソッドを呼び出す時には、単にメソッド名を書くだけになります。無引数メソッドの呼び出しは字面上ローカル変数と区別がつきません。変数への代入があるか、あるいはメソッドの引数である時はローカル変数ですが、そうではない場合はメソッド呼び出しとなります。

この巧妙に仕掛けられた設計により、メソッド定義の中身を書き換えることなくローカル変数をアクセサにすりかえることができます。「変数のように見えたが実

はメソッドだった」というオチです。クローラーのサンプルをリファクタリングする際でもその現象が起きるので、Rubyの妙味を感じてください。

引数のメソッド化の例　test.rb

```
def area(x, y) x*y end
class Rect
  def initialize(x, y) @x, @y = x, y end
  attr_reader :x, :y

  # 引数がメソッド化！
  def area() x*y end
end
area(10,5)            # => 50
Rect.new(10,5).area   # => 50
```

なお、式の後に「# =>」と書いてあるのはその式の値です。リファレンスマニュアルでよく使われている記法です。irb test.rbのようにirbコマンドでスクリプトを実行すれば各式の値が表示されます。irbを使わない場合はKernel#pを使って式の値を出力します。例えば、「p area (10,5)」のように使います。

gem install rcodetoolsでインストールされる「xmpfilter」というスクリプトを使えば、「# =>」の後ろに式の値が注釈されたスクリプトが出力されます。出力結果にさらにxmpfilterを通すことで、注釈結果が更新されます。xmpfilter test.rbのように実行します。

test.rbの実行 (irb)

```
$ irb test.rb
DL is deprecated, please use Fiddle
test.rb(main):001:0> ?def area(x, y) x*y end
=> nil
test.rb(main):002:0> class Rect
test.rb(main):003:1>   def initialize(x, y) @x, @y = x, y end
test.rb(main):004:1>   attr_reader :x, :y
test.rb(main):005:1>   # 引数がメソッド化！
test.rb(main):006:1*   def area() x*y end
test.rb(main):007:1> end
=> nil
test.rb(main):008:0> area(10,5)          # => 50
=> 50
```

```
test.rb(main):009:0> Rect.new(10,5).area    # => 50
=> 50
test.rb(main):010:0>
```

◆ test.rbの実行（xmpfilter）

```
$ xmpfilter test.rb
···def area(x, y) x*y end
class Rect
 def initialize(x, y) @x, @y = x, y end
 attr_reader :x, :y
 # ·····························
 def area() x*y end
end
area(10,5)          # => 50
Rect.new(10,5).area    # => 50
```

ブロック機能

Ruby版高階関数とも言える「ブロック」はとても強力な機能です。簡潔な表記でメソッドに処理を渡すことができます。

ブロックにはさまざまな用途があります。ブロック付きメソッドはかつてイテレータと呼ばれていたように、eachメソッドなどの繰り返し用途が代表的です。他にも、Enumerable#mapのようにブロックの返り値を使うもの、Enumerable#selectのように条件を指定するものもあります。特にEnumerableモジュールのメソッドはeachメソッドを使って書かれているのでイテレータです。それでも繰り返し処理であることを考えずに使えるのは、まさにブロックの強みそのものです。

eachメソッドによる繰り返し処理

```
ary = []; [1,10,100].each{
  |x| ary << x*3 }; ary       # => [3, 30, 300]
[1,10,100].map{|x| x*3 }      # => [3, 30, 300]
[2,7,8].select{|x| x.odd? }  # => [7]
```

キーワード引数

Ruby 2.0で念願のキーワード引数が導入されました。以前のバージョンでもハッシュを渡すことで同じようなことができましたが、キーワード引数を使えば最初からローカル変数に代入されているので、もっと使いやすくなりました。

次の例でもキーワード引数を使っています。

キーワード引数の例

```
def f(a:1, b:"foo", c:nil) [a,b,c] end
f                       # => [1, "foo", nil]
f a:10                  # => [10, "foo", nil]
f c:1.2, a:10           # => [10, "foo", 1.2]
f({:a=>10})             # => [10, "foo", nil]
f unknown:1 rescue $!   # => #<ArgumentError:
                              unknown keyword: unknown>
```

このように、Rubyは魅力いっぱいの楽しい言語です。本書はRubyの基礎知識が前提ですが、自信がない人は基礎力をきっちり身につけましょう。文法やRubyのシステムを理解したら、String、Regexp、Array、Hash、Enumerableなどの基本的なクラスとモジュールを抑えましょう。irbやxmpfilterを使って片っ端からメソッドを試してみると新たな発見があることでしょう。

スクリプトの文字コード

本書で紹介するスクリプトはUTF-8エンコーディングが前提となっています。Windowsにおいては環境変数RUBYOPTに「-Eutf-8」と入れるか、コマンドをruby -Eutf-8として実行してください。コマンドプロンプトのプロパティにてフォントを「MS ゴシック」にしてから「chcp 65001」を実行すれば、コマンドプロンプトがUTF-8対応になります。

1-3-2　文字列処理

クローラーを作るならば文字列を処理することが中心になります。ここでは文字列処理や正規表現について軽く触れておきます。文字列はStringオブジェクトであり、メソッドを渡すことで文字列処理を行います。

文字列の結合

文字列を結合するには「+」演算子（メソッド）を使います。数値と同じ演算子が使えるのは直感的ですね。破壊的に結合するには「<<」演算子を使います。非破壊的結合の「+」、破壊的結合の「<<」は配列でも使えます。

Rubyにおける「破壊的」とは元のオブジェクトを変更することを意味します。破

壊的メソッドは思わぬ落とし穴があるので、Rubyに慣れていないうちは使わない方がよいです。破壊的メソッド名は非破壊的メソッド名に「!」を付けたものが多いのですが、「!」が付いていない破壊的メソッドもいくつかあります。ここで登場するString#force_encodingもその1つです。

文字列の結合

```
# -*- coding: utf-8 -*-
# 文字列の結合
r = "ruby"
r + "ist"   # => "rubyist"
r           # => "ruby"

# 破壊的に結合
r << "ist"  # => "rubyist"
r           # => "rubyist"
```

部分文字列の取得

String#[]は部分文字列を取得します。例えば0から数えてN番目の文字を得るには、[N]と指定します。

文字列を指定して取得する

```
# -*- coding: utf-8 -*-
j = "るびぃ"  # => "るびぃ"

# (0から数えて)1番目の文字を得る
j[1]          # => "び"

# 0番目から2つの文字を得る
j[0,2]        # => "るび"

# 1~2番目の文字を得る
j[1..2]       # => "びぃ"
```

エンコーディング

かつてのRubyの文字列は単なるバイト列でしたが、今は文字の列として扱えます。バイト列から取り出す文字は、エンコーディング(ASCII、Shift_JIS、UTF-8など)で決定づけられます。Rubyでは文字列オブジェクト1つひとつにエンコーディング情報を持たせてあるので、正しく文字単位で処理してくれます。また、複

数のエンコーディングの文字列を混在できます。

エンコーディングが異なる文字列を処理しようとするとエラーになります。ただし、ASCII文字のみで構成されていればそのかぎりではありません。文字列処理の際はエンコーディングに気をつけてください。

文字列とエンコーディング

```
# -*- coding: utf-8 -*-
j = "るびぃ"              # => "るびぃ"

# エンコーディングを得る
j.encoding               # => #<Encoding:UTF-8>

# 文字数を得る
j.length                 # => 3

# バイト数を得る
j.bytesize               # => 9

# エンコーディングを変換する
e = j.encode("EUC-JP")   # => "¥x{A4EB}¥x{A4D3}¥x{A4A3}"
e.encoding               # => #<Encoding:EUC-JP>

# 別のエンコーディングで結合などを行うとエラーになる
j+e rescue $!  # => #<Encoding::CompatibilityError:
               #      incompatible character encodings:
               #      UTF-8 and EUC-JP>

# ASCII文字のみで構成された文字列ならば
# 別エンコーディングでもエラーにならない
"ruby".encode("UTF-8") + "ist".encode("EUC-JP")
# => "rubyist"

# 破壊的にエンコーディング情報をバイナリー
# (ASCII-8BIT)に変更する
j.force_encoding("ascii-8bit")
# => "¥xE3¥x82¥x8B¥xE3¥x81¥xB3¥xE3¥x81¥x83"

j.encoding  # => #<Encoding:ASCII-8BIT>

# ASCII-8BITからUTF-8に変換できないので、kconvを使う
j.encode("UTF-8") rescue $!
# => #<Encoding::UndefinedConversionError: "¥xE3"
#      from ASCII-8BIT to UTF-8>
```

```
require 'kconv'
j = j.toutf8              # => "るびぃ"
j.encoding                # => #<Encoding:UTF-8>

# toutf8はNKF.nkf("-w", str)と同じだが
# MIME解読をするので、それが嫌ならば-m0を付ける
NKF.nkf("-w -m0", j)      # => "るびぃ"
```

メソッドの表記方法

Rubyでのメソッドの表記方法には慣習が存在します。Klassクラスのインスタンスメソッドmethは「Klass#meth」、クラスメソッドは「Klass.meth」あるいは「Klass::meth」と表記します。メソッドの説明を表示するriコマンドではこの表記が使えます。

1-3-3 正規表現

文字列処理において特に重要となるのが正規表現です。文字列処理における正規表現とは、文字列のパターンを記述するためのミニ言語です。文字列が正規表現にマッチするかどうか検査したり、マッチした部分を取り出したり置換したりするのが主な使い方です。最近のRubyの正規表現はパワーアップしているので、今一度リファレンスマニュアルを再読してみるとよいでしょう。

文字列のマッチ

正規表現にマッチするかどうかを検査するには「=~」を使います。String#[]は正規表現を指定でき、その正規表現にマッチする文字列を取り出せます。

正規表現置換はString#subあるいはString#gsubを使います。前者はマッチした最初の1箇所のみ、後者はすべてが置換対象となります。ブロックを指定した時は、その評価結果に置換されます。

正規表現は通常「//」で囲みますが、そのなかに「/」を含む場合は「¥/」とエスケープする必要があります。HTMLやXMLの終了タグには「/」が含まれるので、エスケープ不要な%r記法がしばしば使われます。

メタ文字「*」と「+」と「?」はできるだけ長い文字列にマッチするようになっています。これを「最長マッチ」と言います。それとは反対に「*?」と「+?」と「??」はなるべくマッチしないようにする「最短マッチ」です。タグにマッチさせるには最短マッチを使いましょう。

主な正規表現の使い方

```
# -*- coding: utf-8 -*-
# 正規表現にマッチした位置を得る
"abc123" =~ /¥d+/             # => 3
"abc123" =~ /xyz/             # => nil

# 正規表現にマッチした文字列を得る
"abc123"[/¥d+/]               # => "123"
"abc123"[/xyz/]               # => nil

# 最初に括弧にマッチした文字列を得る
"abc123"[/^[a-z]+(¥d+)/, 1]  # => "123"

# 最長マッチvs最短マッチ
"abc123"[/(.?).+/,1]          # => "a"
"abc123"[/(.??).+/,1]         # => ""
"abc123"[/(.+).+/,1]          # => "abc12"
"abc123"[/(.?+).+/,1]         # => "a"

# シンプルな正規表現置換
"abc123".sub(/abc/, 'def')    # => "def123"

# マッチした部分を評価して置換
"abc123".sub(/[a-z]+/){|s| s.upcase}  # => "ABC123"

# ハッシュで置換パターンを設定できる
"abc123".gsub(/[a-z]/, 'a'=>'x', 'b'=>'y', 'c'=>'z')
# => "xyz123"

# case式で場合分け
type = case "abc123"
       when /^def/
         :def
       when /^abc/
         :abc
       else
         :other
       end
type                          # => :abc

## HTMLから最初のリンクを取り出す
html = '<a href="a.html">a</a><a href="b.html">b</a>'

# 【誤例】「+」は最長マッチなので、
# よけいなものがマッチしてしまう
html[%r!<a href="(.+)">(.+)</a>!, 1]
# => "a.html¥">a</a><a href=¥"b.html"
```

```
# 最短マッチ「+?」や否定文字クラス「[^]」を
# 使うのが正しい
html[%r!<a href="(.+?)">(.+?)</a>!, 1]
# => "a.html"

html[%r!<a href="([^¥"]+)">([^<]+)</a>!, 1]
# => "a.html"
```

情報の抜き出し

クローラーを作るにあたっては情報を抜き出す必要があります。HTMLパーサーを使う方法もありますが、シンプルに正規表現で抜き出すこともできます。

正規表現で情報を抜き出すにはString#[]以外にもString#scanが強力です。正規表現でクローラーを作る時にはほぼ毎回使うことになるほど重要なメソッドです。

String#scanは文字列に対して正規表現を繰り返し適用して、文字列配列を返します。このメソッドは正規表現に括弧(グルーピング)が含まれるか否かで挙動が変わります。括弧なしの場合は、正規表現全体にマッチした文字列の配列を返します。括弧付きの場合は、すべての括弧にマッチした部分の配列の配列を返します。

String#scanに括弧付き正規表現を指定すればまとめて情報を抜き出せますが、必ずしもscan一発ですべての情報が抜き出せるとはかぎりません。その時は2回以上scanを使って別個に抜き出してからインデックスごとに統合する必要があります。

```
(0...dates.length).map{|i| コード}
```

のようにインデックスでループすれば、dates[i]、links[i]で日付とリンクが得られるのですが、この場合にうってつけのメソッドがあります。

Array#zipは、自身と引数に渡した配列の各要素からなる配列の配列を得ます。これを使えばインデックスごとに各配列をループできます。String#scanで別々に情報抽出して、Array#zipでまとめるというパターンはよく使われるので覚えておきましょう。このパターンは後ほど作成する「10分クローラー」において登場します。

String#scanとArray#zip

```
# -*- coding: utf-8 -*-
html = <<XXXX
<p>1993年2月24日
<a href="http://www.ruby-lang.org/ja/">
  Ruby's birthday</a></p>
```

```
<p>2014年1月1日
<a href="http://www.example.com/">元日</a></p>
XXXX

## 情報を抜き出す
# 日付を抜き出す
dates = html.scan(/(¥d+)年(¥d+)月(¥d+)日$/)
# => [["1993", "2", "24"], ["2014", "1", "1"]]

# リンクを抜き出す
links = html.scan(%r!<a href="(.+?)">(.+?)</a></p>!)
# => [["http://www.ruby-lang.org/ja/",
#       "Ruby's birthday"],
#     ["http://www.example.com/", "元日"]]

## 情報をまとめる
# インデックスでループ
(0...dates.length).map{|i| [dates[i], links[i]]}
# => [[["1993", "2", "24"],
#       ["http://www.ruby-lang.org/ja/", "Ruby's birthday"]],
#     [["2014", "1", "1"], ["http://www.example.com/",
#       "元日"]]]

# Array#zipでまとめる
dates.zip(links)
# => [[["1993", "2", "24"],
#       ["http://www.ruby-lang.org/ja/",
#        "Ruby's birthday"]],
#     [["2014", "1", "1"],
#       ["http://www.example.com/", "元日"]]]
```

その他の主な文字列処理を列挙しておきます。

その他の文字列処理

```
# -*- coding: utf-8 -*-
# 書式文字列
format("%s %d URLs", "Download", 23)
# => "Download 23 URLs"

# 改行を取り除く
"abc¥n".chomp         # => "abc"
"abc¥n¥n".chomp       # => "abc¥n"
"abc¥n¥n".chomp("")   # => "abc"

# 文字の出現回数を数える
```

```
"abcd".count("a")    # => 1
"abcd".count("ab")   # => 2
"abcd".count("a-c")  # => 3

# a～cのうちbを含まない文字(ac)を数える
"abcd".count("a-c", "^b")  # => 2

# 行に分割する
s = "ab¥ncd¥n"
s.lines.to_a          # => ["ab¥n", "cd¥n"]
s.lines.map(&:chomp)  # => ["ab", "cd"]
s.split("¥n")         # => ["ab", "cd"]

# 各行ごとに繰り返す
a = []; s.each_line{|l| a << l}; a
# => ["ab¥n", "cd¥n"]

# 文字列を空白で分割
s = " ab cd ef "
s.split               # => ["ab", "cd", "ef"]

# 分割数を2に制限する
s.split(nil,2)        # => ["ab", "cd ef "]

# 先頭と末尾の空白を取り除く
s.strip               # => "ab cd ef"

# 文字に分割する
"abc".chars.to_a      # => ["a", "b", "c"]
```

1-3-4 ファイルを開く

ファイルを読み書きするにはopenメソッドを使います。ファイルの内容をそのまま文字列で得るには主に以下の書き方があります。後々filenameにURLを渡せるようにしたいならば、openを使いかつ短い3番目の記法がおすすめです。3番目は2番目を縮めた書き方です。4番目の記法はファイルが開きっぱなしになります。

```
File.read(filename)
open(filename){|f| f.read}
open(filename, &:read)
open(filename).read
```

ファイル入出力においてはエンコーディングに気をつけてください。入力ファイルがスクリプトのエンコーディングと一致していない場合は、エンコーディング変

換をする必要があります。

ファイルの入出力

```
# -*- coding: utf-8 -*-
TMP="/tmp/test.txt"

# Shift_JIS文字列をファイルに書き込む
open(TMP, "w"){|f|f.puts "あいうえお".encode("Shift_JIS")}

# 読み込むがエンコーディングがマッチしていない
File.read(TMP)
# => "¥x82¥xA0¥x82¥xA2¥x82¥xA4¥x82¥xA6¥x82¥xA8¥n"

unmatched = open(TMP, &:read)
# => "¥x82¥xA0¥x82¥xA2¥x82¥xA4¥x82¥xA6¥x82¥xA8¥n"

unmatched.encoding
# => #<Encoding:UTF-8>

# エンコーディング変換する必要がある
unmatched.force_encoding("Shift_JIS").encode!("UTF-8")
# => "あいうえお¥n"

unmatched
# => "あいうえお¥n"

# 入力ファイルのエンコーディングを指定する
sjis = open(TMP, "r:Shift_JIS", &:read)
# => "¥x{82A0}¥x{82A2}¥x{82A4}¥x{82A6}¥x{82A8}¥n"

sjis.encoding          # => #<Encoding:Shift_JIS>
sjis.encode("UTF-8")   # => "あいうえお¥n"

# 内部エンコーディングも指定すれば
# openの段階でUTF-8になる
utf8 = open(TMP, "r:Shift_JIS:UTF-8", &:read)
# => "あいうえお¥n"
utf8.encoding
# => #<Encoding:UTF-8>

# エンコーディングを推測して変換する
# NKF.nkfはString#toutf8と違いMIME decodeをしない
require 'kconv'
File.read(TMP).toutf8             # => "あいうえお¥n"
NKF.nkf("-wmO", File.read(TMP))   # => "あいうえお¥n"
File.unlink TMP
```

1-4 Rubyでテストサーバを立てる

よそのサイトに迷惑をかけずにWgetの挙動を検証するため、Rubyでテストサーバを立ち上げます。

1-4-1 標準ライブラリ「WEBrick」でお手軽httpd

本書で登場する最初のサンプルはWebサーバです。Rubyの基礎に触れたばかりでいきなりサーバというのは敷居が高く思えるかもしれませんが、まったく難しくはないので安心してください。今やサーバといえども単なるテキスト処理にすぎません。

本書はクローラー開発のための本であり、代表的なクローラーであるWgetを取り上げました。Wgetのオプションは多岐にわたり、細かい設定もできるようになっています。やりたいことがWgetのみで完結する場合、わざわざRubyで新たに開発する必要はありません。クローリングをWgetにまかせて、Rubyでその結果を処理する方法も考えられます。

方針を決定するためにはWgetについて熟知する必要があります。しかし、クローラーは集中アクセスを伴うため、よそのサイトに対してむやみに試運転するのは迷惑がかかります。とはいえ動作を理解するには何度も手を動かして実行する必要があります。

このジレンマを解決するのが、ローカルでWebサーバを立ち上げることなのです。ローカルのサーバならばいかに苛酷なアクセスをしてもまったく問題がありません。

幸いにもRubyには「WEBrick」というWebサーバが標準で付いてきます。つまり、Rubyをインストールした時点でWebサーバが使えることになります。

それではさっそくWEBrickを使ってみましょう。

ディレクトリの公開

カレントディレクトリをポート番号7777でローカルに公開するには、以下のスクリプトを使用します。WEBrickスレッドを1つ作り、メインスレッドでは標準入力を待っています。実行後は、[Enter]キーを押せば終了します。

カレントディレクトリを公開する　webrick0.rb

```
#!/usr/bin/env ruby
require 'webrick'
Thread.start{
  WEBrick::HTTPServer.new(DocumentRoot:".",
    Port:7777, BindAddress:"127.0.0.1").start
}
gets
```

◆ webrick0.rbの実行例

```
$ ruby webrick0.rb
[2014-06-25 21:17:17] INFO  WEBrick 1.3.1
[2014-06-25 21:17:17] INFO  ruby 2.1.2 (2014-05-08) [x86_64-linux]
[2014-06-25 21:17:17] INFO  WEBrick::HTTPServer#start: pid=4460 port=7777
```

bashやzshならばスクリプトを作成することなくワンライナーでできます。

◆ スクリプトファイルなしでカレントディレクトリを公開する

```
$ ruby -rwebrick -e 'Thread.start{ WEBrick::HTTPServer.new(DocumentRoot:".", Port:7777,
BindAddress:"127.0.0.1").start };gets'
```

もしカレントディレクトリに「test.html」があるのならば、「http://127.0.0.1:7777/test.html」でアクセスできます。

Webブラウザからも見られますが、Wgetでダウンロードもできます。ダウンロードする時にファイル名を指定すると、test.htmlの内容をコピーするのと同じになります。カレントディレクトリに「test-copy.html」としてコピーされます。

◆ 公開したディレクトリにアクセスする

```
$ wget -O test-copy.html http://127.0.0.1:7777/test.html
```

組み込みWebサーバを簡単に実現できる雰囲気を感じ取っていただけたでしょうか。もちろんこれだけではあまり意味がないので、動作検証用のWebサーバを作ります。

1-4-2　URLから規則的な内容のページを表示する

WEBrickの素晴しい点は、目的に特化したWebサーバを手軽に構築できること

です。今回の目的はWgetの動作検証なので、規則的な内容のページを表示するサーバを立ち上げます。難しいことはなく、テンプレートとなるスクリプトを書き換えるだけで実現できます。

規則的な内容を表示するWebサーバ webrick-template.rb

```
#!/usr/local/bin/ruby
require 'webrick'

class TestContentServlet <
  WEBrick::HTTPServlet::AbstractServlet ●―――――❶

  def do_GET(req, res)
    res.body = "This is #{req.path}<br>"
    res.content_type = WEBrick::HTTPUtils.mime_type(
      req.path_info,
      WEBrick::HTTPUtils::DefaultMimeTypes)
  end
end

srv = WEBrick::HTTPServer.new(
  :BindAddress => '127.0.0.1', :Port => 7777)
srv.mount('/', TestContentServlet) ●―――――――❷
trap("INT"){ srv.shutdown } ●―――――――――❸
srv.start
```

Webサーバが返す内容を定義するには、WEBrick::HTTPServerにServlet（サーバ上で動くプログラム）をマウントします。

ServletはWEBrick::HTTPServlet::AbstractServletのサブクラスを定義し（❶）、do_GETメソッドで内容を記述します。HTTP GETリクエストがきたらdo_GETメソッドを実行し、res.bodyにページ内容、res.content_typeにContent-Typeを設定します。ここではパスからContent-Typeを自動判別させています。

早い話、res.bodyにページ内容を代入してしまえば、それだけでWebサーバができてしまうのです。何も難しいことはありません。他はすべておまじないです。

srv.mountはServletの割り当て先を指定します（❷）。ここでは❶で定義したTestContentServletのみ立ち上げるので「/」に割り当てていますが、任意のディレクトリを指定すれば複数のServletを共存できます。

「trap("INT")」は［Ctrl］＋［C］キーで終了した時に実行されるコードを指定します（❸）。［Ctrl］＋［C］でサーバを終了したいのでこのように書いています。

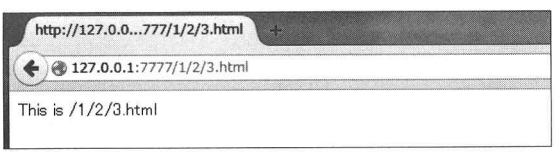
このテンプレートはそのまま実行できます。以下のコマンドでサーバを実行します。

◆ webrick-template.rbの実行例

```
$ ruby webrick-template.rb
[2014-06-25 21:23:19] INFO  WEBrick 1.3.1
[2014-06-25 21:23:19] INFO  ruby 2.1.2 (2014-05-08) [x86_64-linux]
[2014-06-25 21:23:19] INFO  WEBrick::HTTPServer#start: pid=5492 port=7777
```

ブラウザで「http://127.0.0.1:7777/1/2/3.html」にアクセスすれば、「This is/1/2/3.html」と表示されます。他のパスを指定しても同様です。このサーバを終了させるには［Ctrl］+［C］キーを押してください。

▼ ブラウザで表示する



このテンプレートサーバを実行することで、req.pathにはURLのうちポート番号の後ろの文字列が代入されることがわかりました。それではres.bodyの内容を決定しましょう。以下の仕様にします。

▼ res.bodyの仕様

要　素	概　要
Path	URLからドメイン部を取り除いた文字列
Node	パスから拡張子を取り除いた文字列
#{Node}.htmlからのリンク	#{Node}/1.html,
	#{Node}/2.html,
	#{Node}.txt,
	http://localhost:7777#{Node}.org,
	/1.html
#{Node}.txtの内容	"This is #{Path}"
その他のPath	"dummy"

例えば、「http://127.0.0.1:7777/1.html」のパスは「/1.html」で、Nodeは「/1」です。「/1.html」からは、「/1/1.html」「/1/2.html」「/1.txt」「http://localhost:7777/1.org」

「/1.html」へのリンクがあります。「/1.txt」の内容は「This is /1.txt」です。

同様に「/1/1.html」からは「/1/1/1.html」「/1/1/2.html」へリンクできるようになります。まるで地下に無限に続いていく階段のように、どんどん下の階層まで潜れるような構造です。それではスクリプトを見てみましょう。

テストサーバ test-webserver0.rb

```
#!/usr/local/bin/ruby
# -*- coding: utf-8 -*-
require 'webrick'

class TestContentServlet <
  WEBrick::HTTPServlet::AbstractServlet

  def do_GET(req, res)
    # 拡張子で分岐
    res.body = case req.path
               when /¥.html$/; html_content req.path
               when /¥.txt$/;  txt_content req.path
               else;           "dummy"
               end
    res.content_type = WEBrick::HTTPUtils.mime_type(
      req.path_info, WEBrick::HTTPUtils::DefaultMimeTypes)
  end

  def html_content(path)
    node = path[0..-6]
    <<HTML
<html><head><title>#{path}</title></head>
<body><p>
<a href="#{node}/1.html">#{node}/1.html</a><br>
<a href="#{node}/2.html">#{node}/2.html</a><br>
<a href="#{node}.txt">#{node}.txt</a><br>
<a href="http://localhost:7777#{node}.org">
  #{node}.org</a><br>
<a href="/1.html">/1.html</a>
</p></body></html>
HTML
  end

  def txt_content(path)
    "This is #{path}"
  end
end

srv = WEBrick::HTTPServer.new(:BindAddress => '127.0.0.1', :Port => 7777)
```

```
srv.mount('/', TestContentServlet)
trap("INT"){ srv.shutdown }
srv.start
```

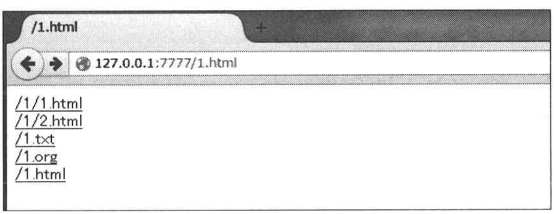
長いですが、パスを分岐して該当するメソッドを呼ぶようにres.bodyを書き換えただけです。

上のスクリプトを以下のコマンドで動かしてください。そして、「http://127.0.0.1:7777/1.html」を開いて正しくリンクが表示されれば成功です。

◆ test-webserver0.rbの実行例

```
$ ruby test-webserver0.rb
[2014-06-25 21:45:43] INFO  WEBrick 1.3.1
[2014-06-25 21:45:43] INFO  ruby 2.1.2 (2014-05-08) [x86_64-linux]
[2014-06-25 21:45:43] INFO  WEBrick::HTTPServer#start: pid=4980 port=7777
```

◆ テストサーバにアクセスする



なおセキュリティ上、見知らぬ人と同時に使うコンピュータでは動かさないでください。このテストサーバでもwget -r -linfを実行されたら無限にダウンロードしてしまう弱点があります。ここでは簡単にするため個人用PCで動かすことを前提としています。

テストサーバは、[Ctrl] + [C] キーで終了します。

1-4-3　Wgetのオプションを検証する

これでやっとWgetの細かい動作検証ができるようになりました。このテストサーバであれば誰にも迷惑はかからないので、じっくり試せます。

再帰ダウンロードの検証

それでは再帰ダウンロードの挙動の検証をしてみましょう。「http://127.0.0.1:

7777/1.html」を再帰ダウンロードした結果を示します。-nvオプションでよけいな出力を省いておきます。

◆ 再帰ダウンロードの検証

```
$ wget -nv -r -l1 http://127.0.0.1:7777/1.html
2014-07-01 09:43:40 URL:http://127.0.0.1:7777/1.html [262/262] -> "127.0.0.1:7777/1.html" [1]
2014-07-01 09:43:40 URL:http://127.0.0.1:7777/robots.txt [19/19] -> "127.0.0.1:7777/robots.txt" [1]
2014-07-01 09:43:40 URL:http://127.0.0.1:7777/1/1.html [280/280] -> "127.0.0.1:7777/1/1.html" [1]
2014-07-01 09:43:40 URL:http://127.0.0.1:7777/1/2.html [280/280] -> "127.0.0.1:7777/1/2.html" [1]
2014-07-01 09:43:40 URL:http://127.0.0.1:7777/1.txt [14/14] -> "127.0.0.1:7777/1.txt" [1]
FINISHED --2014-07-01 09:43:40--
Total wall clock time: 0.2s
Downloaded: 5 files, 855 in 0.2s (5.26 KB/s)
```

見てのとおり、4つのリンクと「/robots.txt」をダウンロードし、URLに対応するディレクトリに格納されていることがわかります。Windowsでは「:」がドライブレターで使えないため、「127.0.0.1+7777」というディレクトリになります。

「http://localhost:7777/1.org」は辿っていません。「127.0.0.1」と「localhost」は同一ホストですが、文字列上異なるのでWgetでは別ドメインと見なされるからです。検証用Webサーバにおいてはあえてこのように作り、別ドメインでの挙動をシミュレーションできるようにしました。

robots.txt

「/robots.txt」はサーバ側がクローラー(ロボット)の動きを制御するための設定ファイルです。Wgetは礼儀正しいクローラーなので/robots.txtも読み込みます。なお、このテストサーバは/robots.txtに意味のある内容は書いていません。

リンク変換の検証

保存された「127.0.0.1:7777/1.html」(Windowsでは127.0.0.1+7777/1.html)をブラウザで開いても、絶対パスでリンクが書かれているため、以降のリンクを辿れなくなっています。そこでリンク変換オプション-kを付けてみます。

◆ ダウンロード後にリンク変換を行う

```
$ wget -nv -r -l1 -k http://127.0.0.1:7777/1.html
```

出力は同じなので省きます。相対パスに書き換えられたため、実行後再びブラウザで「127.0.0.1:7777/1.html」を開くとリンクが辿れるようになります。

127.0.0.1:7777/1.html

```
<html><head><title>/1.html</title></head>
<body><p>
<a href="1/1.html">/1/1.html</a><br>
<a href="1/2.html">/1/2.html</a><br>
<a href="1.txt">/1.txt</a><br>
<a href="http://localhost:7777/1.org">/1.org</a><br>
<a href="1.html">/1.html</a>
</p></body></html>
```

「1.org」もダウンロードするには-Hを付けます。

◆ 別ドメインのファイルもダウンロードする

```
$ wget -nv -r -l1 -H -k http://127.0.0.1:7777/1.html
2014-07-01 09:59:37 URL:http://127.0.0.1:7777/1.html [262/262] -> "127.0.0.1:7777/1.html" [1]
2014-07-01 09:59:37 URL:http://127.0.0.1:7777/robots.txt [19/19] -> "127.0.0.1:7777/robots.txt" [1]
2014-07-01 09:59:37 URL:http://localhost:7777/robots.txt [19/19] -> "localhost:7777/robots.txt" [1]
2014-07-01 09:59:37 URL:http://127.0.0.1:7777/1/1.html [280/280] -> "127.0.0.1:7777/1/1.html" [1]
2014-07-01 09:59:37 URL:http://127.0.0.1:7777/1/2.html [280/280] -> "127.0.0.1:7777/1/2.html" [1]
2014-07-01 09:59:37 URL:http://127.0.0.1:7777/1.txt [14/14] -> "127.0.0.1:7777/1.txt" [1]
2014-07-01 09:59:37 URL:http://localhost:7777/1.org [5/5] -> "localhost:7777/1.org" [1]
FINISHED --2014-07-01 09:59:37--
Total wall clock time: 0.2s
Downloaded: 7 files, 879 in 0.2s (3.62 KB/s)
```

ウェイトの検証

ウェイトのオプション（→p.9）も検証しましょう。時刻に注目してみると、ダウンロードごとにまばらな秒数だけ待っていることがわかります。

◆ ウェイトを設定してダウンロードする

```
$ wget -nv -r -l1 -H -k -w4 --random-wait http://127.0.0.1:7777/1.html
2014-07-01 10:00:03 URL:http://127.0.0.1:7777/1.html [262/262] -> "127.0.0.1:7777/1.html" [1]
2014-07-01 10:00:08 URL:http://127.0.0.1:7777/robots.txt [19/19] -> "127.0.0.1:7777/robots.txt" [1]
2014-07-01 10:00:12 URL:http://localhost:7777/robots.txt [19/19] -> "localhost:7777/robots.txt" [1]
2014-07-01 10:00:17 URL:http://127.0.0.1:7777/1/1.html [280/280] -> "127.0.0.1:7777/1/1.html" [1]
2014-07-01 10:00:20 URL:http://127.0.0.1:7777/1/2.html [280/280] -> "127.0.0.1:7777/1/2.html" [1]
2014-07-01 10:00:22 URL:http://127.0.0.1:7777/1.txt [14/14] -> "127.0.0.1:7777/1.txt" [1]
```

```
2014-07-01 10:00:28 URL:http://localhost:7777/1.org [5/5] -> "localhost:7777/1.org" [1]
FINISHED --2014-07-01 10:00:28--
Total wall clock time: 25s
Downloaded: 7 files, 879 in 0s (55.2 MB/s)
```

Wgetには他にもたくさんのオプションが存在します。それらをすべて検証するのは本書の狙いからは外れるのでやりませんが、テストWebサーバを書き換えて検証ページを作成すればできます。テストサーバと二人三脚でWgetのオプションについて学んでみましょう。思いもよらぬ便利なオプションが発見できるかもしれません。

オプションを確認する

Wgetは幅広く使われていますが、あまりに多くのオプションがあり、どう使えばいいのかわからなくなるかもしれません。幸いコマンドラインツールの使い方のデータベースを検索する「cmdline-fu」「bro」「cheat」などのツールが存在するので、それらを使えば具体的な使用例がわかるようになります。いずれもGemで提供されているので、インストールも使い方も簡単です。

◆「cmdline-fu」「bro」「cheat」のインストール

```
$ sudo gem install cmdline-fu bropages cheat
```

◆「cmdline-fu」「bro」「cheat」のインストール(Windows)

```
C:work> gem install cmdline-fu bropages cheat
```

◆「cmdline-fu」の実行例

```
$ cmdline-fu matching wget
true
# Download an entire website
wget --random-wait -r -p -e robots=off -U mozilla http://www.example.com

# Extract tarball from internet without local saving
wget -qO - "http://www.tarball.com/tarball.gz" | tar zxvf -
# Google Translate
translate(){ wget -qO- "http://ajax.googleapis.com/ajax/services/language/translate?v=1.0&q=
$1&langpair=$2|${3:-en}" | sed 's/.*"translatedText":"¥([^"]*¥)".*}/¥1¥n/';}
```

```
~省略~

## Page(1):0-25 http://www.commandlinefu.com/commands/matching/wget/d2dldA==/
sort-by-votes/plaintext/0
# commandlinefu.com by David Winterbottom
```

1-5 超簡単！ 10分で作るクローラー

クローラー開発の手始めとして、記事リンクのURLとタイトルを出力する簡単な「10分クローラー」を作ってみましょう。

1-5-1　概略

ここまでは身近なクローラーとしてWgetを取り上げてきました。検証したように、Wgetはクローラーとして立派な仕事をしてくれます。しかし、Wgetはあくまでダウンロードするためのツールであり、クローラーとしての機能は再帰ダウンロードやリンク変換くらいです。もっと特化した処理を行いたいならば、Rubyでオリジナルのクローラーを作る必要があります。

そのためには、まず簡単なクローラーを作成することから始めましょう。今回ここで作るクローラーは、SBCRトピックスの記事リンクを抜き出すものです。

■ SBCRトピックス
URL **http://www.sbcr.jp/topics/**

元のURLは「http://www.sbcr.jp/topics/」ですが、クローラーはサイトのデザイン変更に振り回されてしまうのが宿命です。このスクリプトは執筆時点（2014年7月現在）では元のURLでも動作しますが、将来も動作する保証はありません。よって、本書の解説のために用意した別のURLに、サイトの内容をコピーしたHTMLを使っていきます。

■ コピーしたサイト
URL **http://crawler.sbcr.jp/samplepage.html**

クローラーの処理の流れは、大まかに以下のようになります。

- ①対象ページをダウンロードする
- ②ダウンロードしたページを解析する
- ③そこから必要なデータを抜き出す
- ④データを加工する
- ⑤出力する

ここでは単一のHTMLページを処理するクローラーの作成を通じて、その流れを追うことにします。ここで作成する「10分クローラー」はHTMLページの記事リンクを取り出し、各記事のタイトルとURLを標準出力に出力するものです。HTTPアクセスの部分は、最初はWgetを呼び出すことにします。

そして次節では、以下のように10分クローラーを拡張していきます。それらはすべてRubyの標準ライブラリでできます。標準ライブラリだけでもここまでできるのかと感じ取ってください。

- Wgetを呼ばずにRubyでダウンロード処理をする
- 機能をクラスに分けるリファクタリングを行う
- プレインテキスト以外にもRSSでも出力できるようにする
- クローラー自体をWEBrickでRSS配信サーバにする

このスクリプトは実用面も考え、カスタマイズしやすいように作っていきます。少し手を加えるだけで、RSS配信されていないサイトがRSSリーダーで読めるようになります。

1-5-2 HTMLを解析する

それでは、10分クローラーの心臓部であるHTMLを解析するスクリプトから作り始めましょう。これができてしまえば、ほとんど完成したようなものです。まずは解析対象ページを「samplepage.html」という名前で保存します。

◆ 対象ページを保存する

```
$ wget -O samplepage.html http://crawler.sbcr.jp/samplepage.html
```

カレントディレクトリに「samplepage.html」が保存されます。

samplepage.html

クローラーで最重要となる部分が対象ページの解析です。ここが一番難しく、逆にここさえできれば完成したようなものです。それではsamplepage.htmlを解析しましょう。

エンコーディングの統一

まずはsamplepage.htmlのエンコーディングを見てください。多くのWebサイトではUTF-8ですが、ここも例に漏れずUTF-8です。

UTF-8ならばそのまま処理できますが、それ以外の場合は、String#encodeやString#toutf8でスクリプトのエンコーディングを揃える必要があります。

情報の確認

そしてテキストエディタやブラウザでHTMLのソースを眺めてください。取り出したい情報がどこにあるのか、どのように書かれているのかを見るのです。

たいていの場合、情報は規則的に並んでいます。HTMLの癖を観察して、どう取り出すかの手段を決定します。HTMLパーサーを使ってXPath指定で取り出すこともできますし、正規表現で取り出すこともできます。

「samplepage.html」より抜粋

```
<div class="topicsCenterColumnTopicsTitle">
  <div class="topicsCenterColumnTopicsListDate">
    2014年2月21日<br /> ●―――――❶
  </div>
  <div class="topicsCenterColumnTopicsListTitle">

<a href="http://www.sbcr.jp/topics/11719/">
  最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】 キャンペーン開催中
  </a><br /> ●―――――❷

  </div>
</div>
```

幸い、samplepage.htmlには、すごくわかりやすい癖があります。

- 日付は「YYYY年MM月DD日
」と書かれている（❶）
- 記事リンクは行頭のA要素になっていて
で終わっている（❷）

それならばわざわざHTMLパーサーを使うまでもなく、正規表現だけで簡単に取り出せます。ここでは正規表現で取り出すことにします。

日付と記事リンクの取り出し

それではさっそくString#scanで日付と記事リンクを取り出してみましょう。

テキスト処理では正規表現の微調整など試行錯誤する必要があります。そのため、ラフスケッチとして小さいスクリプトから作っていきます。ここでは日付・記事リンクのそれぞれ最初の4つだけ取り出し、それらが対応していることを確認します。

日付と記事リンクの取得

sbcr0.rb

```
page_source = open("samplepage.html", &:read)
dates = page_source.scan(%r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
dates[0,4]
# => [["2014", "2", "21"],
#     ["2014", "2", "20"],
#     ["2014", "2", "14"],
#     ["2014", "2", "12"]]
url_titles = page_source.scan(
  %r!^<a href="(.+?)">(.+?)</a><br />!)
url_titles[0,4]
# => [["http://www.sbcr.jp/topics/11719/",
#      "最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】
```

```
#       キャンペーン開催中"],
#      ["http://www.sbcr.jp/topics/11712/",
#       "【新刊情報】2014年2月17日 23日
#         「コンセプト」の作り方がわかるビジネス書など12点"],
#      ["http://www.sbcr.jp/topics/11710/",
#       "『数学ガール』電子書籍版がAmazon Kindleで配信開始！
#         キャンペーンも同時開催！！"],
#      ["http://www.sbcr.jp/topics/11703/",
#       "【新刊情報】2014年2月10日 16日
#         アニメ化決定『ワルブレ』最新刊など11点"]]

# datesとurl_titlesの個数が一致している
dates.length       # => 68
url_titles.length  # => 68
```

String#scanを使って日付、記事URL・記事タイトルを取得できます。しかし、両者は別々の配列であり、各インデックスに対応しています。こんな時はArray#zipを使えばよいです。ページの都合上、2つの配列の最初の2要素のみで確認しています。

Array#zipで取得する

```
dates[0,2].zip(url_titles[0,2])
# => [[["2014", "2", "21"],
#       ["http://www.sbcr.jp/topics/11719/",
#        "最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】
#        キャンペーン開催中"]],
#     [["2014", "2", "20"],
#       ["http://www.sbcr.jp/topics/11712/",
#        "【新刊情報】2014年2月17日～23日
#        「コンセプト」の作り方がわかるビジネス書など12点"]]]
```

日付は文字列配列ではなくて1つのTimeオブジェクトにした方が応用範囲が広がります。Time.localは年月日時分秒を表す数字または文字列を取り、日本時間のTimeオブジェクトを作成します。

```
Time.local "2014", "2", "20"  # => 2014-02-20 00:00:00 +0900
Time.local 2014, 2, 20        # => 2014-02-20 00:00:00 +0900
```

HTMLを正規表現で解析する時に、よく忘れがちなのがHTMLアンエスケープです。HTMLでは特定の記号には独自の記法が定めてあります。正規表現で抜き出しただけだとこれらが残ってしまうので、CGI.unescapeHTMLで必ずアンエスケープしておきます。

```
require 'cgi'
CGI.unescapeHTML "&lt;A&amp;B&gt;"  # => "<A&B>"
```

これらを総合したparseメソッドは次のようになります。年月日の配列をそのままTime.localの引数にするので、「*」で渡しています。

parseメソッド sbcr1.rb

```
# -*- coding: utf-8 -*-
require 'cgi'

def parse(page_source)
  dates = page_source.scan(
    %r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
  url_titles = page_source.scan(
    %r!^<a href="(.+?)">(.+?)</a><br />!)
  url_titles.zip(dates).map{|(aurl, atitle),
    ymd|[CGI.unescapeHTML(aurl),
    CGI.unescapeHTML(atitle), Time.local(*ymd)]
  }
end
x = parse(open("samplepage.html", &:read))
x[0,2]
# => [["http://www.sbcr.jp/topics/11719/",
#       "最強の布陣で挑む！ GA文庫電子版
#       【俺TUEEEEE】キャンペーン開催中",
#       2014-02-21 00:00:00 +0900],
#      ["http://www.sbcr.jp/topics/11712/",
#       "【新刊情報】2014年2月17日～23日
#       「コンセプト」の作り方がわかるビジネス書など12点",
#       2014-02-20 00:00:00 +0900]]
```

1-5-3 WgetをRubyから呼ぶ

parseメソッドだけでは、クローラーではなくてただのテキスト処理プログラムです。クローラーは、プログラムのなかでHTTPアクセスを行うものです。

このスクリプトをクローラーにするのは簡単です。バッククォートを使いWgetをRubyから呼び出してしまえばよいです。バッククォートは、囲まれたコマンドの標準出力を文字列として取り出します。さらに-qオプションを付けてよけいな出力を抑制して、次の置き換えを行います。もちろん、Wgetのパスは各自正しく指定しておいてください。

```
x = parse(open("samplepage.html", &:read))
```

上記のparseメソッドを、以下のように書き換えます。

```
x = parse(`/usr/bin/wget -q -O- http://crawler.sbcr.jp/samplepage.html`)
```

実行結果は変わりません。

parseメソッド(Wget版)　sbcr1.rb

```
# -*- coding: utf-8 -*-
require 'cgi'
def parse(page_source)
  dates = page_source.scan(
    %r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
  url_titles = page_source.scan(
    %r!^<a href="(.+?)">(.+?)</a><br />!)
  url_titles.zip(dates).map{|(aurl, atitle),
    ymd|[CGI.unescapeHTML(aurl), CGI.unescapeHTML(atitle),
    Time.local(*ymd)]
  }
end
x = parse(`/usr/bin/wget -q -O- http://crawler.sbcr.jp/samplepage.html`)
x[0,2]
# => [["http://www.sbcr.jp/topics/11719/",
#       "最強の布陣で挑む！ GA文庫電子版
#       【俺TUEEEEE】キャンペーン開催中",
#       2014-02-21 00:00:00 +0900],
#      ["http://www.sbcr.jp/topics/11712/",
#       "【新刊情報】2014年2月17日～23日
#       「コンセプト」の作り方がわかるビジネス書など12点",
#       2014-02-20 00:00:00 +0900]]
```

1-5-4　最新記事をテキストで出力する

Webページはparseメソッドで記事リンク配列の配列に変換されました。10分クローラー最後の仕上げは、それをわかりやすく表示することです。ここではformat_textメソッドで出力文字列を作成します。

最新記事を取得する　sbcr2.rb

```
# -*- coding: utf-8 -*-
require 'cgi'

def parse(page_source)
```

```
  dates = page_source.scan(
    %r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
  url_titles = page_source.scan(
    %r!^<a href="(.+?)">(.+?)</a><br />!)
  url_titles.zip(dates).map{|(aurl, atitle),
    ymd|[CGI.unescapeHTML(aurl),
    CGI.unescapeHTML(atitle), Time.local(*ymd)]
  }
end

def format_text(title, url, url_title_time_ary)
  s = "Title: #{title}¥nURL: #{url}¥n¥n"
  url_title_time_ary.each do |aurl, atitle, atime|
    s << "* (#{atime})#{atitle}¥n"
    s << "    #{aurl}¥n"
  end
  s
end

puts format_text("WWW.SBCR.JP トピックス",
  "http://crawler.sbcr.jp/samplepage.html",
  parse(`/usr/bin/wget -q -O-
    http://crawler.sbcr.jp/samplepage.html`))
```

実行結果は以下のようになります。実行の際には、Wgetのパスは各自の環境に合わせて変更してください。

◆ sbcr2.rbの実行例

```
$ ruby sbcr2.rb

Title:WWW.SBCR.JP トピックス
URL: http://crawler.sbcr.jp/samplepage.html

* (2014-02-21 00:00:00 +0900)最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】キャンペーン開催中
  http://www.sbcr.jp/topics/11719/
* (2014-02-20 00:00:00 +0900)【新刊情報】2014年2月17日～23日 「コンセプト」の作り方がわかるビジネス書など12点
  http://www.sbcr.jp/topics/11712/

~省略~
```

これが10分クローラーです。ページ取得にWgetを使っているものの、プログラム自体でWebページを処理しているこのスクリプトは、れっきとしたクローラー

です。筆者も複雑なHTTPアクセスを伴う時にはRubyからWgetやcURLを呼び出すことはよくやります。試行錯誤したコマンドラインをそのまま使えるので、個人的用途であれば手っ取り早い手段です。

このスクリプトを作るうえで一番苦労したのは、HTMLを観察して正規表現を作成するところだと思います。そこさえ乗り切れば、Webページから情報を抜き出すスクリプトを作成することは難しくありません。テキスト処理に強いRubyの力です。次節はこのスクリプトをもっと発展させていきます。

1-6 クローラーを拡張する

本章最後は、先ほど作成した10分クローラーを改良して実用レベルにまで持っていきます。

1-6-1　open-uriに対応させる

まずは10分クローラーからWgetへの依存を取り除きましょう。そのためにはRubyでHTTPアクセスを行う必要がありますが、ここで紹介するopen-uriライブラリは一番簡単な方法です。Rubyでシンプルに HTTPアクセスをするならばこれを試しましょう。

open-uriの特徴は、普段使っているKernel#openをURLも扱えるように拡張しているところです。よって、ファイル名の部分をそのままURLに置き換えるだけで、openがそのままHTTPクライアントになります。open-uriで簡単なクローラーを書くことは、ほとんどの場合、通常のファイル処理を書くこととなんら変わりないのです。

10分クローラーのスクリプト（sbcr2.rb、→p.40）に対し、以下の置き換えを行います。

```
`/usr/bin/wget -q -O- http://crawler.sbcr.jp/samplepage.html`
```

Wgetの呼び出しを行っている部分を、open-uri対応に書き換えます。

```
require 'open-uri'
open("http://crawler.sbcr.jp/samplepage.html", &:read)
```

open-uriはopenを拡張しているだけなので、「File.read(IO.read)」には手をつけていません。ファイルの内容を丸ごと取得するのにFile.readを使っている人は、

```
open(ファイル名,&:read)
```

に書き換えればopen-uriでの拡張の恩恵を受けます。

しかし、このままではHTMLのエンコーディングが単なるバイト列を意味するASCII-8BITになるので、切り出し処理が動作しません。open-uriの仕様として、HTTPレスポンスでContent-Typeのcharsetが指定されていない時はASCII-8BITになってしまいます。crawler.sbcr.jpは（元のwww.sbcr.jpも）そのケースなので、openの引数に明示的にUTF-8を指定する必要があります。よって、次のようになります。呼び出し先を「samplepage.html」に置き換えても動作します。

```
open("http://crawler.sbcr.jp/samplepage.html", "r:UTF-8", &:read)
```

また、別解としてString#toutf8を使ってエンコーディングを変換する方法もあります。crawler.sbcr.jpはUTF-8で書かれているので文字列の内容は変更されず、文字列オブジェクトのエンコーディングがUTF-8に設定されます。この方法ならばEUC-JPやShift_JISで書かれたサイトでもUTF-8に統一して処理が行えます。

```
require 'kconv'
open("http://crawler.sbcr.jp/samplepage.html", &:read).toutf8
```

1-6-2 RSS2.0での出力に対応させる

次は、プレインテキスト（Plain text）で出力された記事リンクを、RSSでも出力するように拡張しましょう。Rubyならばrss標準ライブラリを使えばとても簡単です。

rss/2.0.rbの説明には、以下のようなサンプルが書いてあります。このサンプルは直接実行可能なので、実行結果も載せておきます。

rss/2.0.rbのサンプル rsstest.rb

```
require "rss"

rss = RSS::Maker.make("2.0") do |maker|
  # フィードの言語 日本語ならja
  maker.channel.language = "en"

  # フィード作成者
  maker.channel.author = "matz"

  # フィードの更新時刻
  maker.channel.updated = Time.now.to_s
```

```
  # フィードURL
  maker.channel.link =
    "http://www.ruby-lang.org/en/feeds/news.rss"

  # フィードの名前
  maker.channel.title = "Example Feed"

  # フィードの要旨
  maker.channel.description =
    "A longer description of my feed."

  # 項目はmaker.items.new_itemの
  # ブロックで定義する
  maker.items.new_item do |item|
    # 記事のURL
    item.link =
      "http://www.ruby-lang.org/en/news/
      2010/12/25/ruby-1-9-2-p136-is-released/"

    # 記事のタイトル
    item.title = "Ruby 1.9.2-p136 is released"

   # 記事の更新時刻
    item.updated = Time.now.to_s
  end
end

puts rss
```

◆ **rss/2.0.rbのサンプルの実行結果**

```
$ ruby rsstest.rb
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0"
 xmlns:content="http://purl.org/rss/1.0/modules/content/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"
 xmlns:trackback="http://madskills.com/public/xml/rss/module/trackback/">
 <channel>
  <title>Example Feed</title>
  <link>http://www.ruby-lang.org/en/feeds/news.rss</link>
  <description>A longer description of my feed.</description>
  <language>en</language>
  <pubDate>Sat, 08 Mar 2014 03:35:02 +0900</pubDate>
  <item>
```

```
  <title>Ruby 1.9.2-p136 is released</title>
  <link>http://www.ruby-lang.org/en/news/2010/12/25/ruby-1-9-2-p136-is-released/</link>
  <pubDate>Sat, 08 Mar 2014 03:35:02 +0900</pubDate>
  <dc:date>2014-03-08T03:35:02+09:00</dc:date>
 </item>
 ~省略~
 <dc:date>2014-03-08T03:35:02+09:00</dc:date>
</channel>
</rss>
```

サンプルと実行結果を見比べると、ブロックの階層とXMLの階層が一致している宣言的なドメイン特化言語（DSL）だとわかります。各記事は「maker.items.new_item」でどんどん追加していきます。今回はこのサンプルをほぼそのまま使ってあげればよいことになります。なお、一部の要素は省略しています。

ここでは、コマンドライン引数に「text-output」を指定した時にはプレインテキストで出力し、「rss-output」を指定した時にはRSSで出力するようにしましょう。引数が同じメソッドを分岐させるので、Object#__send__でシンボルとして指定したメソッドを呼び出しています。ファイル名はrssserver.rbにして、順次書き加えていく形になります。

10分クローラー改（RSS出力対応） rssserver.rb

```
# -*- coding: utf-8 -*-
require 'cgi'
require 'open-uri'
require 'rss'

def parse(page_source)
  dates = page_source.scan(
    %r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
  url_titles = page_source.scan(
    %r!^<a href="(.+?)">(.+?)</a><br />!)
  url_titles.zip(dates).map{|(aurl, atitle),
    ymd|[CGI.unescapeHTML(aurl), CGI.unescapeHTML(atitle),
    Time.local(*ymd)]
  }
end

def format_text(title, url, url_title_time_ary)
  s = "Title: #{title}¥nURL: #{url}¥n¥n"
  url_title_time_ary.each do |aurl, atitle, atime|
```

```
    s << "* (#{atime})#{atitle}¥n"
    s << "    #{aurl}¥n"
  end
  s
end

def format_rss(title, url, url_title_time_ary)
  RSS::Maker.make("2.0") do |maker|
    maker.channel.updated = Time.now.to_s
    maker.channel.link = url
    maker.channel.title = title
    maker.channel.description = title
    url_title_time_ary.each do |aurl, atitle, atime|
      maker.items.new_item do |item|
        item.link = aurl
        item.title = atitle
        item.updated = atime
        item.description = atitle
      end
    end
  end
end

parsed = parse(open(
  "http://crawler.sbcr.jp/samplepage.html",
  "r:UTF-8", &:read))

formatter = case ARGV.first
            when "rss-output"
              :format_rss
            when "text-output"
              :format_text
            end
puts __send__(formatter,
  "WWW.SBCR.JP トピックス",
  "http://crawler.sbcr.jp/samplepage.html", parsed)
```

RSS出力されるように実行してみましょう。

◆ rssserver.rbの実行例

```
$ ruby rssserver.rb rss-output
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0"
 xmlns:content="http://purl.org/rss/1.0/modules/content/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
```

```
 xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"
 xmlns:trackback="http://madskills.com/public/xml/rss/module/trackback/">
 <channel>
  <title>WWW.SBCR.JP トピックス</title>
  <link>http://crawler.sbcr.jp/samplepage.html</link>
  <description>WWW.SBCR.JP トピックス</description>
  <pubDate>Tue, 01 Jul 2014 21:51:12 +0900</pubDate>
  <item>
   <title>最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】キャンペーン開催中</title>
   <link>http://www.sbcr.jp/topics/11719/</link>
   <description>最強の布陣で挑む！ GA文庫電子版【俺TUEEEEE】キャンペーン開催中</description>
   <pubDate>Fri, 21 Feb 2014 00:00:00 +0900</pubDate>
   <dc:date>2014-02-21T00:00:00+09:00</dc:date>
  </item>

  ~省略~

  <dc:date>2014-07-01T21:51:12+09:00</dc:date>
 </channel>
</rss>
```

1-6-3 リファクタリング

本節の目標は、10分クローラーをRSS配信サーバに変身させることです。それと同時に、手を加えれば他のサイトでも使えるようにしたいと思います。そのために、そろそろオブジェクト指向らしいプログラムに書き換えましょう。

まずはSiteクラスとそのサブクラスSbcrTopicsを定義します。open-uriでページの内容を取得するのはどのサイトでも使えるので、Site#page_sourceにします。parseメソッドはsbcrトピックスにローカルなのでSbcrTopicsで定義します。他のサイトを解析したければ、そのサイト用のサブクラスを定義すればよいことになります。

出力の方もformat_textとformat_rssの2つのメソッドが登場してるので、これらもクラスに分けておきたいです。Formatterクラスとそのサブクラス TextFormatter、RSSFormatterを定義しましょう。

この状況下でRSSフィードを出力するには、次のコードになります。SbcrTopics.newはキーワード引数「url」「title」を取ります。そして、Site#outputでFormatterのクラスを指定して、対応する出力を得るようにします。Rubyにおいて、クラスはオブジェクトなのでメソッドの引数に渡せます。

```
site = SbcrTopics.new(
  url:"http://crawler.sbcr.jp/samplepage.html",
  title:"WWW.SBCR.JP トピックス")
puts site.output RSSFormatter
```

site.output RSSFormatterは、内部で、

```
RSSFormatter.new(site).format(site.parse)
```

が呼ばれるようにします。

この改造により、次のようにメソッドが移動します。

- parse　→ SbcrTopics#parse
- format_text　→ TextFormatter#format
- format_rss　→ RSSFormatter#format

メソッドをクラスに配属させることで、引数の数を減らせるメリットもあります。元のparseメソッドではpage_sourceが引数になっていましたが、ここではSite#page_sourceとメソッド化されたので、page_source引数はなくなります。

Siteクラス

Site#page_sourceは一度読み込んだらキャッシュしておきたいので、インスタンス変数に記憶しておきます。SbcrTopics#parseにて2回page_sourceが出てきているので、キャッシュしないと再び読み込んでしまいます。インスタンス変数にキャッシュするには「||=」がよく使われます。「@x ||= v」は「@x = @x || v」の略記であり、@xがnilかfalseの時は「v」を返し、それ以外の時は「@x」を返します。未定義のインスタンス変数はnilに初期化されているのでこの手法が使えます。ただし、キャッシュする内容がnilかfalseの時は使えません。

エンコーディングはスクリプトのエンコーディングであるUTF-8に統一するので、String#toutf8で変換しておきます。

10分クローラー改2（Siteクラス）　rssserver2.rb

```
require 'cgi'
require 'open-uri'
require 'rss'
require 'kconv'

class Site
```

```
  def initialize(url:"", title:"")
    @url, @title = url, title
  end
  attr_reader :url, :title

  def page_source
    @page_source ||= open(@url, &:read).toutf8
  end

  def output(formatter_klass)
    formatter_klass.new(self).format(parse)
  end
end

class SbcrTopics < Site
  def parse
    dates = page_source.scan(
      %r!(¥d+)年 ?(¥d+)月 ?(¥d+)日<br />!)
    url_titles = page_source.scan(
      %r!^<a href="(.+?)">(.+?)</a><br />!)
    url_titles.zip(dates).map{|(aurl, atitle),
      ymd|[CGI.unescapeHTML(aurl),
      CGI.unescapeHTML(atitle), Time.local(*ymd)]
    }
  end
end
```

Formatterクラス

今度はFormatterクラスです。元のformat_text、format_rssメソッドの引数にtitleとurlがありますが、それぞれFormatter#title、Formatter#urlとメソッド(アクセサ)化されているので、これらの引数はなくなります。

Formatter#title、Formatter#urlはそれぞれSiteオブジェクトから取得します。そのため、Siteクラスでattr_readerを定義しておいたのです。

10分クローラー改2 (Formatterクラス) rssserver2.rb

```
class Formatter
  def initialize(site)
    @url = site.url
    @title = site.title
  end
  attr_reader :url, :title
end
```

```
class TextFormatter < Formatter
  def format(url_title_time_ary)
    s = "Title: #{title}¥nURL: #{url}¥n¥n"
    url_title_time_ary.each do |aurl, atitle, atime|
      s << "* (#{atime})#{atitle}¥n"
      s << "    #{aurl}¥n"
    end
    s
  end
end

class RSSFormatter < Formatter
  def format(url_title_time_ary)
    RSS::Maker.make("2.0") do |maker|
      maker.channel.updated = Time.now.to_s
      maker.channel.link = url
      maker.channel.title = title
      maker.channel.description = title
      url_title_time_ary.each do |aurl, atitle, atime|
        maker.items.new_item do |item|
          item.link = aurl
          item.title = atitle
          item.updated = atime
          item.description = atitle
        end
      end
    end
  end
end
```

スクリプトの呼び出し

呼び出し部は以下のように変化します。

10分クローラー改2（呼び出し部）　rssserver2.rb

```
site = SbcrTopics.new(
  url:"http://crawler.sbcr.jp/samplepage.html",
  title:"WWW.SBCR.JP トピックス")
case ARGV.first
when "rss-output"
  puts site.output RSSFormatter
when "text-output"
  puts site.output TextFormatter
end
```

実行方法と出力結果は変わりません。

元のスクリプト(rssserver.rb、p.45)は関数的メソッドがたった3つというシンプルなものだったので、一気にリファクタリングしてしまいました。その過程において、parse、format_text、format_rssメソッドは内容に手を加えることなくそのまますっぽりとクラスに収まりました。ローカル変数と無引数メソッド呼び出しでは字面上の区別がないのでそのような現象が起きるのです。

これでクラス階層ができたので、新しいサイトを扱いたければSiteのサブクラスでparseメソッドを定義すればよいことになります。同様に新しい出力フォーマットを扱いたければFormatterのサブクラスでformatメソッドを定義することになります。

1-6-4 RSSサーバにする

最後は10分クローラーをRSS配信サーバにしてしまいます。Wgetの動作検証で使ったWebサーバ(→p.25)も、RSSFormatterもこのための布石だったのです。

Wgetのところで示したように、WEBrickのテンプレートを使えばWebサーバを立ち上げるのは難しくありません。WEBrick::HTTPServlet::AbstractServletのサブクラスRSSServletにて、do_GETメソッドを定義しましょう。そこで、res.bodyにRSSの文字列、res.content_typeに「"application/xml; charset=utf-8"」を指定します。

ただし1つだけ問題があります。RSSServletは外から与えられるサイト情報に依存しているので、どのように情報を渡せばいいのかということです。実はAbstractServletには外からパラメータを渡す仕組みが存在していて、この要求に応えてくれます。WEBrickのテンプレート(→p.26)で触れたように、WEBrick::HTTPServerオブジェクトでサーバを立ち上げ、mountメソッドでServletをマウントするようになっています。このmountメソッドの第3引数以降に任意のオブジェクトを渡せます。これらのオブジェクトはAbstractServlet内では@optionsというインスタンス変数で参照できます。

ただし、AbstractServletにそのままSiteオブジェクトを渡してはいけません。これはWebサーバなので何度も実行されます。Siteオブジェクトは一度page_sourceにアクセスしたらそれを記憶するため、サイトの内容が変化した時にも古い情報を元にRSSが作られてしまいます。よって、クラス、URL、タイトルをmountメソッドで渡し、RSSServlet#do_GETでSiteオブジェクトを作ります。URLとタイトルはハッシュで、SbcrTopics#initializeにそのまま渡します。

あとはテンプレート（→p.26）にあるサーバ定義をメソッド化するだけです。なお、クラス定義部分はp.45で掲載した内容と同じなので省略しています。

10分クローラー改2　rssserver2.rb

```
# -*- coding: utf-8 -*-
require 'cgi'
require 'open-uri'
require 'rss'
require 'kconv'
require 'webrick'

class Site
  ~省略~
end

class SbcrTopics < Site
  def parse
    ~省略~
  end
end

class Formatter
  ~省略~
end

class TextFormatter < Formatter
  ~省略~
end

class RSSFormatter < Formatter
  ~省略~
end

class RSSServlet <
  WEBrick::HTTPServlet::AbstractServlet

  def do_GET(req, res)
    klass, opts = @options
    res.body =
      klass.new(opts).output(RSSFormatter).to_s
    res.content_type =
      "application/xml; charset=utf-8"
  end
end

def start_server
```

```
  srv = WEBrick::HTTPServer.new(:BindAddress =>
    '127.0.0.1', :Port => 7777)
  srv.mount('/rss.xml', RSSServlet, SbcrTopics, ●1
    url:"http://crawler.sbcr.jp/samplepage.html",
    title:"WWW.SBCR.JP トピックス")
  # 場合によってはsrv.mount行を追加する
  trap("INT"){ srv.shutdown }
  srv.start
end

if ARGV.first == 'server'
  start_server
else
  site = SbcrTopics.new(
    url:"http://crawler.sbcr.jp/samplepage.html",
    title:"WWW.SBCR.JP トピックス")
  case ARGV.first
  when "rss-output"
    puts site.output RSSFormatter
  when "text-output"
    puts site.output TextFormatter
  end
end
```

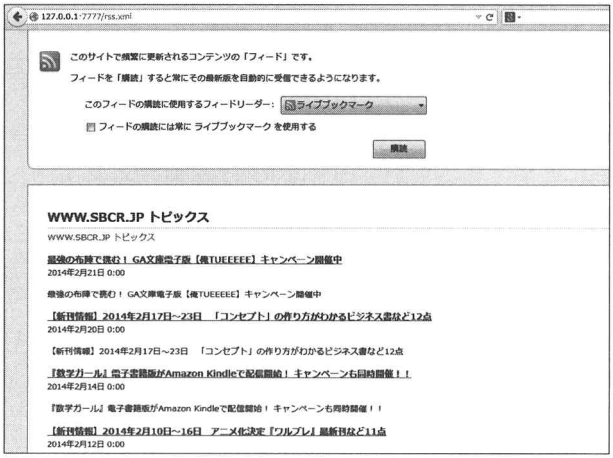
スクリプト内でアミかけで示した部分をターゲットに合わせて書き換えれば、他のサイトに対しても汎用的に使うことができます。

◆ rssserver2.rbの実行例

```
$ ruby rssserver2.rb server
[2014-06-26 17:33:08] INFO  WEBrick 1.3.1
[2014-06-26 17:33:08] INFO  ruby 2.1.2 (2014-05-08) [x86_64-linux]
[2014-06-26 17:33:08] INFO  WEBrick::HTTPServer#start: pid=4768 port=7777
```

「server」を指定して実行すると、RSSサーバが立ち上がります。そして、ブラウザから「http://127.0.0.1:7777/rss.xml」にアクセスすれば、RSSが出てきます。[Ctrl]＋[C]キーで終了します。

ブラウザからRSSを確認する



「rss-output」か「text-output」を指定すれば出力後そのまま終了します。

start_serverメソッドを見れば予想できるように、srv.mountを複数個書けば、他のサイトのRSSを表示できます(❶)。この「10分クローラー改」を他のサイトでも使うには、Siteのサブクラスとparseメソッドを定義し、srv.mount行を増やすだけです。これでRSSが提供されていないサイトの最新情報をRSSリーダーで読むことができます。

WEBrickは組み込みWebサーバなので、常駐型クローラーの動作状況をWebインターフェースやRSSで報告するという応用例もあります。次章ではより本格的なクローラーを作っていきます。

Chapter 2

クローラー作成の基礎

2-1 クローラーの目的と構造

本書では、基本的なクローラーの構造と作り方を学び、その後で目的のサイトごとに特化したクローラーの作り方を紹介します。最終的には、読者が自分の目的に沿った実践的なクローラーを作れるようになることをゴールとします。

このセクションでは、まずは一般的なクローラーの構造を学ぶことで、クローラーの機能や動作の基本を理解をします。その次にクローラーが利用するモジュールを知ることで、より詳細な動きを理解します。最後に、本書で利用するクローラーの紹介をします。

2-1-1　クローラーの目的

クローラー（Crawler）は、Web上の文章や画像を自動的に取得するプログラムのことを指します。もともとは、検索エンジンによって世界中のWebサイトをデータベース化、インデックス化する目的で開発されたもので、Googleのグーグルボットなどが有名です。クローラー以外にもさまざまな呼び名があり、「ボット（Bot）」「スパイダー（spider）」「ロボット（robot）」などとも呼ばれます。またWeb以外を対象に、ファイルサーバやデータベース内を巡回し、インデックス化する目的のプログラムもクローラーと呼ばれます。本書では、Webサイトを巡回して文章や画像を取得するプログラムを対象とし、呼び名を「クローラー」に統一します。

それではクローラーの目的とは何でしょうか？　検索エンジンのクローラーは、広くあまねくWebサイトを巡回してデータベース化することを目的とします。その成果として、我々は検索エンジンを利用してさまざまな情報を収集できます。一方で検索エンジンを利用するだけでは、自分が欲しい情報だけ抜き出して、定期的に取得するといったことはできません。そこで、自分専用のクローラーを作ることにより、取得元の対象サイトを絞り、必要とするデータのみを効率的に取得することができるのです。

2-1-2　クローラーの構造

クローラーは対象のWebサイトに対して、コンテンツをダウンロードして保存し、次の取得先を見つけていくといった形で巡回していきます。これらのクローラーの動きを構造として着目すると、大きく分けて次の3つに分類できます。

- コンテンツの取得
- データの解析
- データの保存

実はこの構造は、Webブラウザと大差はありません。ブラウザの構造は、データの「取得・解析・描写」です。つまりクローラーとブラウザの違いは、コンピュータか人間、どちらが利用するかでしかありません。その点を踏まえたうえで、クローラーの構造や動きを見ると理解が早くなります。それでは、クローラーのそれぞれの機能を見ていきましょう。

コンテンツの取得(クローリング)

コンテンツの取得は、サイトにアクセスしてWebページなどをダウンロードする機能です。クローラーは、HTMLをダウンロードして解析し、そのなかに含まれるリンク先を見つけてそのページをダウンロードするというプロセスを繰り返します。この一連のプロセスを「クローリング」(crawling)と呼びます。

またクローラーが扱うリンク先には、<A>タグ(アンカータグ)で示されているページや、<Img>タグ(画像タグ)や<Form>タグ(フォームタグ)のAction属性で指定された遷移先などがあります。本書では、クローラーが扱うリンク先としては、特別な指定がない場合は<A>タグを指すものとします。それ以外を対象にする場合は、明示して記載します。

クローラーを最も狭義な意味で定義すると「コンテンツを取得し巡回する機能」となります。クロール対象となるWebページには、大きく分けると次の3つの種類があります。

- ステートレスなページ(状態を持たないページ)
- ステートフルなページ(状態を持つページ)
- JavaScriptを元にクライアント側でページを組み立てるタイプ

ステートレスなページは、URLを指定すると単純にHTMLが返ってくるページです。JavaScriptを元にクライアント側でページを組み立てることもしません。ブログやニュースページなど多くのサイトが、このステートレスなページにあたります。

このパターンはURLに対応したHTMLを取得するだけで目的のデータを得ることができるので、クローラーで最も対応しやすいです。単純なクローラーはこのようなページを想定して作られ、1URLで1要素のみ取得します。つまりURLを指定

すると、そのHTMLのみを取得し付随のCSSやJavaScript、画像は取得しません。

ステートフルなページは、ログインずみの状態でないと参照できないページや、POSTなどで事前に送られた情報をサーバ側が保持し、その前提でないと参照できないパターンです。サーバサイドで動的に生成されるページの多くはこの形式です。クローリングする際は、認証やPOSTでのパラメータの送信の問題を解決するために、クローラー側で一工夫が必要です。対処法として、ブラウザをエミュレートするライブラリを利用したうえで、ログインやフォームの入力など特定ページに対する操作に特化して対応することが多いです。

JavaScriptを元にクライアント側でページを組み立てるタイプは、JavaScriptの指示の元にデータの取得や処理をブラウザ内で行い、ページを組み立てて表示するタイプです。目的のデータを取得するにはブラウザと同じようにJavaScriptを解釈して描写する必要があるので、通常のクローラーで対応するのは非常に困難です。

これに対応するには、2つの方法があります。JavaScriptの動作を人間が解釈して同様の動きをプログラム側に組み込む方法と、クローラーがJavaScriptの解釈をできるようにする方法です。前者は対象とするページの構造・複雑さにより難易度は格段に変わります。後者のJavaScriptを解釈させるためには、クローラーの内部に簡易ブラウザのエンジンを組み込んだり、クローラーがブラウザそのものを起動させるなどして、あたかも人間がブラウジングしているようにシミュレートする必要があります。

この他にもクローラーがコンテンツを取得する際に、付随的に必要になる機能がいくつもあります。ケーススタディを元に、順次解説していきます。

- robots.txtや<Meta>タグなどを参考に、コンテンツを取得してもよいかの判断
- プロキシサーバの指定
- ユーザーエージェント名の指定
- Cookieの受け入れ
- 処理を並列化して、効率的にデータを取得

なお、本書のなかで「ページ」はWebサイトで表示される個々の文章を指します。またHTMLについては、サーバサイドで動的に生成されたものか、サーバ上に置かれたHTMLファイルかを問わずに、最終的にクライアント側にダウンロードされたHTML文章を指すものとします。

robots.txtと<Meta>タグ

robots.txtとは、サイトの所有者がクローラーに対して命令を記述するためのファイルです。強制力はないものの、検索エンジンにクローリングの禁止などの指示を出すことができます。同様のことを、<Meta>タグで記述することが可能です。

クローラーの開発者は、対象ページのrobots.txtや<Meta>タグに注意し、そこに書いてある指示に従うように心がける必要があります。他にも、クローリングをするうえで従うべきことがいくつかあります。詳細は、それぞれのセクションで説明します。

データの解析(スクレイピング)

データの解析は、ダウンロードしたページを解析して特定のデータを引き抜く機能です。データを引き抜く部分は、特に「スクレイプ」(抜き出す)と呼ばれます。

解析の実装方法としては、以下の2つがあります。

- 正規表現を利用して、パターンマッチングする
- HTMLやXMLの文法を理解して、構文解析する

正規表現を利用する方法は、目的とするデータもしくはその周辺のデータの特徴を元に、パターンマッチングする方法です。この方法は単一の要素を抜き出す場合は手軽で非常に簡単に実装できるケースが多いです。構造化されていないHTMLの場合にも威力を発揮します。一方で複数の要素を取得する場合は、ループや条件分岐を駆使した複雑なプログラムになりがちです。また、取得先のページのデザイン変更のたびに対応する必要がある可能性があります。

もう1つの手法である構文解析は、取得したHTMLやXMLを構文解析ツール(パーサー)を利用し、CSSセレクタやXPathで要素を指定して抜き出します。この方法は構文を解析したうえで順番に辿っていくために、解析の処理コストが大きくなります。しかし、取得先が構造化されたHTMLの場合であれば、簡潔に処理が記述できるうえに取得結果の正確性が高いです。

昨今のWebサイトは、ブログやCMSあるいは動的システムなどから生成されるものが多く、構造化されたHTMLが主流になっています。またHTML5では文章構造がより厳密化され、検索エンジンを始めとするコンピュータから解釈しやすい方向に向かっています。これらの点を考えると、構文解析型のスクレイピングはより優位になると予想されます。そこで本書では、構文解析型のスクレイピングを中心

に解説していきます。

この他にも、HTML中の<A>タグの抽出や指定キーワードの出現の検出など、目的に特化して作り込む場合もあります。

データの保存

データの保存は、取得したデータをメモリ内、もしくはファイルやデータベースなどに保存しておく機能です。メモリへの保存は、巡回・解析のための一時的なものです。ファイルやデータベースへの保存は、データを永続化するために利用します。

永続化することにより、定期的なクローリングでは訪問間隔や多重度の調整や取得ずみのデータをスキップするなど、効率的な巡回が可能になります。また、データを保存することにより、巡回工程と解析工程の分離が可能になり、それぞれの機能が疎結合で運用保守しやすいプログラムになります。

一度かぎりのクローリングであれば、データ保存機能は必要ありません。一方で、クローラーを使い定期的に運用していく場合は、データ保存機能が重要になります。

2-1-3　クローラーが利用するライブラリ

ここまでで、クローラーの構成要素が一通り理解できたと思います。次はRubyでクローラーを作るために、どのようなモジュールを使うのかを見ていきましょう。

クローラーのそれぞれの機能をどのように作ればよいのでしょうか？　もちろんRubyを使って、ダウンロードや構文解析などそれぞれの機能を作ることは可能です。しかし、その方法は非常に手間がかかります。プログラマの世界では、その行為を「車輪の再発明」と呼び、広く受け入れられ確立されている技術や解決法を知らずに、同様のものを一から作ることを避ける傾向があります。

Rubyの世界では、世界中のRubyプログラマ達が作ったライブラリをGemという形式で広く公開し、誰でも利用できるようになっています。本書で紹介するクローラーが利用しているライブラリや、カスタマイズ時に利用するライブラリを、一部でありますがここで紹介します。

ダウンロードのライブラリ

Rubyのダウンロードライブラリの定番としては、「open-uri」があります。

open-uriは、Rubyの組み込みライブラリであるKernel#openを再定義したもので、ファイルと同様の操作でHTTP/FTPにアクセスすることができます。またopen-uriは標準添付ライブラリであり、Rubyが利用可能であればすぐに使うことができ

ます。下記のサンプルスクリプトのようにURLを指定するだけで(❶)、ファイルと同じように扱うことができます。

open-uriの利用例 open-uri.rb

```
require 'open-uri'

open(
  "http://docs.ruby-lang.org/ja/2.1.0/doc/index.html") {  ❶
  |f| f.each_line {|line| p line}
}
```

その他にも、プロキシやベーシック認証への対応やリダイレクト機能など、HTMLをダウンロードする際に利用する多くの機能を備えています。

似たようなライブラリに「httpclient」があります。こちらは、さらにCookieの対応や、HTTPのHEADメソッドやPOSTメソッドにも対応しています。

HTML構文解析のライブラリ

構文解析とは、対象の文章を定義された文法に従って解釈し分析することを指します。この構文解析を行うプログラムを、構文解析器やパーサー(parser)と呼びます。HTMLの構文解析に特化したプログラムであれば「HTMLパーサー」です。クローラーはHTMLやXMLの分析したうえで目的のデータを取得するので、パーサーが必須となります。

RubyのHTML構文解析のライブラリとしては、「Nokogiri」が定番です。NokogiriはHTMLやXMLを解釈できるパーサーで、XPathとCSSセレクタを使っての分析が可能です。下記の例では、Nokogiriの公式トップページを対象にCSSセレクタを利用して、ページ中の<A>タグのhref属性をすべて取得しています。なお、下記のスクリプトの実行にはNokogiriのインストールが必要です(→p.74、80)。

Nokogiriの利用例 nokogiri.rb

```
require 'nokogiri'
require 'open-uri'

doc = Nokogiri.HTML(open("http://nokogiri.org/"))

doc.css('a').each do |element|
  puts element[:href]
end
```

Nokogiriはhpricot

Nokogiriはhpricotというhtml構文解析ライブラリの影響を受けていて、機能面や構文などに類似点が多くあります。しかしhpricotの開発は既に終了し、hpricotの開発者自身もNokogiriの利用を推奨している状態です。このため、特別な理由がないかぎり、Nokogiriを使うのがよいでしょう。

一方で、Nokogiriとhpricotで1つだけ大きな違いがあります。Nokogiriは文字コードをUTF-8として扱うことを前提で作られています。hpricotはその前提はなく、文字コードを意識することなく、そのまま返すように作られています。そのため、NokogiriでUTF-8以外の文字コードを利用する場合は、一定の考慮が必要です。このあたりの事情は、「3-3 文字コードの対処法」にて紹介します（→p.156）。

補助的なライブラリ

ダウンロードや構文解析などクローラーの中核をなすライブラリの他にも、単機能だけど利用することで開発が楽になるライブラリがたくさんあります。例えば、「robotex」です。このライブラリはrobots.txtを参照しクロールの可否を確認します。なお、以下のスクリプトの実行にはrobotexのインストールが必要です（→p.144）

robotexの利用例　robotex.rb

```
require 'robotex'

robotex = Robotex.new
  "My User Agent Name Like WebCrawler"
p robotex.allowed?("http://www.yahoo.co.jp")
p robotex.allowed?("https://www.facebook.com/")
```

他にも便利なライブラリはたくさんあります。本書では利用する段階で、随時紹介することとします。

2-1-4　Ruby製のクローラー

先ほど「車輪の再発明」を避けるということで、Rubyのライブラリを紹介してきました。それでは、Ruby製のクローラーは存在しないのでしょうか？　もちろんクローラーを作るためのライブラリやフレームワークは既に多数存在しています。一方で、それらのライブラリやフレームワークは、クローラーを作るための機能が提供されているだけです。クローラーを作るということは、どのWebサイトから

何の情報を取得するかを定義し、それに沿って処理を追加していくということです。言い換えると、目的を注入することです。

本書ではクローラーを一から自作するということをしません。用途に応じて既存のライブラリを選択し、目的に応じた処理を追加するという形式を取ることとします。それでは、「2-1-2 クローラーの構造」(→p.56)で分類した3種類のWebページを元に、それぞれに最適なクローラーのライブラリを紹介します。

ステートレスなページを得意とするクローラー

ステートレスなページ(→p.57)は、クローラーが最も得意とするものです。多くのクローラーが、このタイプに対処するように特化しています。例えば、ニュースサイトを巡回して記事のタイトルを取得したり、ランキングページを定期的にクロールして順位の変動を観測したりといった用途で使われることが多いです。代表的なライブラリとして「Anemone」があります。

Anemoneは、対象のサイトに対して自動的に巡回します。巡回に際して必要な、次の巡回先の取得やページの解析、サイト側がクローラーに対しての許諾を定めるrobots.txtの検出と対応など、クローラーに必要な機能を一通り揃えています。下記のサンプルスクリプトは、Anemoneを利用してYahoo!ニュースから記事の一覧を取得する例です。実行にはAnemoneのインストールが必要です(→p.74、78)。

Yahoo!ニュースから記事の一覧を取得する anemone.rb

```
require 'anemone'

Anemone.crawl("http://news.yahoo.co.jp/", :depth_limit => 0) do |anemone|
  anemone.on_every_page do |page|
    page.doc.xpath(
      "//*[@id=¥"editorsPick¥"]/div/ul[1]/li/div/p/a").each do |title|

      puts title.text
    end
  end
end
```

Anemoneを利用すると、わずか10行足らずのコードでHTMLのダウンロードから解析まで行えます。クローラーを上手く利用することにより、取得先を指定するだけで目的の情報を簡単に取得できます。Anemoneの使い方については「2-2 Anemoneを利用する」(→p.66)以降で詳しく説明します。

ステートフルなページを得意とするクローラー

ステートフルなページ(→p.58)に対処するには、<Form>タグのAction属性を利用し、POSTメソッドでパラメータを送ることや、Cookieの受け入れが必要になります。Anemoneを始めとするステートレスなページに特化したクローラーには、対応できないものが多いです。

このような場合には、あたかも人間が操作しているような対話型の処理を行う機能が必要です。Rubyで対話型処理を行うライブラリとしては、代表的なもので「Mechanize」があります。Mechanizeは、Webサイトとの対話を自動化するためのライブラリであり、そのために必要なCookieの保存や<Form>タグへの入力や送信といった機能を備えています。

次のサンプルスクリプトは、Mechanizeを利用してAmazonのアソシエイト(アフィリエイト)ページにログインして、売上を取得する例です。実行にはMechanizeのインストールが必要です(Gemからインストールできます)。また、Amazonアソシエイトのユーザー名(❶)とパスワード(❷)も必要になります。

Amazonアソシエイトの売上を取得する mechanize.rb

```
require 'mechanize'

uri=URI.parse('https://affiliate.amazon.co.jp/')
agent = Mechanize.new
agent.user_agent_alias = 'Mac Safari'
page = agent.get(uri)
next_page = page.form_with(:name => 'sign_in') do |form|

  form.username = 'your_username' ●―――❶
  form.password = 'your_password' ●―――❷
end.submit
puts next_page.search('//*[@id="mini-report"]/div[5]/div[2]').text
```

Mechanizeを利用すると、ページに必要な事項を入力して次のページに遷移し、次のページのなかから必要なデータを抜き出すといったことが可能になります。一方で対話型ですので、ページごとの処理を記述する必要があります。そのため、サイト内を自動巡回してすべての情報を取得するといった処理には不向きです。しかし、Webページにログインして、特定の情報のみ取得するといった処理には最適です。

JavaScriptに対応できるクローラー

MechanizeはWebページと対話型の処理が可能です。しかし、取得したHTMLがJavaScriptでクライアント側で文章を組み立てることを前提としている場合、HTML内から目的のデータを取得するのは非常な困難を伴うことが多いです。なぜなら、MechanizeにはブラウザのようにJavaScriptを解釈・処理する機能がないからです。

JavaScriptの解釈・処理が必要なページ(→p.58)には、クローラー自身にブラウザエンジンを組み込んでいるもの、もしくはブラウザそのものを起動して操作できるものが必要になります。Rubyの場合は「Capybara」というライブラリがそのような機能を有しています。

もともとCapybaraは、Webシステムのテストを効率化するために開発されています。テストのために、ブラウザを利用して所定のパラメータを入力し、その結果が正しいかといった検証を自動化するために利用されています。また「Selenium」というツールも、Capybaraと同様のことができます。CapybaraとSeleniumをそれぞれ単体で使うことも可能ですし、組み合わせて使うことも可能です。

このブラウザを操作するという機能を転用することで、JavaScriptにも対処できるクローラーが作れます。下記のサンプルスクリプトでは、Seleniumを使いTwitterからトレンド情報を取得しています。実行にはSeleniumのインストールが必要です(→p.116)。

Twitterからトレンド情報を取得する selenium.rb

```
require 'selenium-webdriver'

# ブラウザの起動
driver = Selenium::WebDriver.for :firefox

# Waitの指定
driver.manage.timeouts.page_load = 10

driver.navigate.to 'https://twitter.com/search-home'

elements = driver.find_elements(
  :xpath, "//ul[@class='trend-items js-trends']/li/a")
elements.each do |element|
  # Tweetの表示
  puts element.text
```

```
end

driver.quit
```

TwitterのWebページでは、JavaScriptの遅延ロードを利用して、ツイートの表示をしています。遅延ロードとは、JavaScriptを利用して画面表示後に非同期に情報を取得する処理です。そのため、JavaScriptの解釈および実行ができないと、ページを構成する情報を取得することができません。CapybaraやSeleniumのようなブラウザを利用するタイプのクローラーであれば、遅延ロードのページも処理できます。

本書では、CapybaraやSeleniumを利用するクローラーを「ブラウザタイプのクローラー」と呼ぶこととします。CapybaraおよびSeleniumについては、「2-8 ブラウザタイプのクローラー」(→p.114)で詳しく説明します。

※

ここまでで4つのライブラリを紹介しました。このなかで明確にクローリングを目的として作られているものは、Anemoneだけです。MechanizeとCapybara、Seleniumはクローリングを主な用途として設計されていません。しかし、クローラーの基本機能であるダウンロードが可能であれば、解析や巡回機能は後付けで対処することは容易です。

本書では、これらのライブラリを利用して、目的ごとに使い分け効率的に対処する方法を紹介します。

2-2 Anemoneを利用する

Anemoneは、2009年にChris Kiteによりクローラーのフレームワークとして開発されたRubyのライブラリです。クローラーが必要とするデータ取得、解析、保存のすべての機能を備え、Rubyのクローラーライブラリとしては最も完成度が高いものの1つです。2014年現在では開発が停滞しているものの、現在もさまざまなユーザーに利用され続けています。

このセクションでは、Anemoneを使ってクローラーを作るために、まずAnemoneの主な機能と処理の流れを学びます。その後に、WindowsもしくはMacへインストールし、クローラーの開発準備を行います。

2-2-1 Anemoneの機能

先述のとおり、Anemoneはクローラーに必要なデータ取得、解析、保存のすべての機能を備えています。クローラーの作成に先立って、まずはAnemoneの機能ごとの主要なメソッドと使い方を学びましょう。

Anemoneの巡回機能

Anemoneのメインの機能は、Anemone::Coreに実装されています。Webサイトの巡回と取得したページの処理に関する機能もここで実装されていて、4つのインスタンスメソッドが用意されています。巡回先のURLを操作するfocus_crawlとskip_links_like、取得したページに対する処理であるon_every_pageとon_pages_likeです。

Anemoneの巡回に関するメソッド

メソッド名	機　能
focus_crawl (&block)	ページごとに、どのリンク先を巡回するか指定する
skip_links_like (*patterns)	巡回しないURLを正規表現で指定する
on_every_page (&block)	取得したページのすべてのページの処理をする
on_pages_like (*patterns, &block)	取得したページのうち、正規表現で一致したページのみの処理をする

focus_crawlもしくはskip_links_likeは、巡回するURLを指定します。取得対象から外す場合は、skip_links_likeを正規表現のパターンで指定します。取得対象のURLを絞り込む場合は、focus_crawlを利用します。focus_crawlはURLの配列を渡すことを求められていますが、実際の使い方としてはリンク先の一覧を取得して正規表現で絞り込むと便利です。

skip_links_likeの指定パターン　　anemone-skip_links_like.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

Anemone.crawl("http://www.yahoo.co.jp") do |anemone|
  # adminを含むURLは除外
  anemone.skip_links_like /¥/r¥//
  anemone.on_every_page do |page|
    puts page.url
  end
end
```

◆ anemone-skip_links_like.rbの実行例

```
$ ruby anemone-skip_links_like.rb
http://www.yahoo.co.jp/
```

■ focus_crawlの指定パターン（正規表現で指定）　anemone-forcus_crawl.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

Anemone.crawl(
  "http://www.amazon.co.jp/gp/bestsellers/", :depth_limit => 1) do |anemone|

  anemone.focus_crawl do |page|
    page.links.keep_if { |link|
      # bestsellersを含むURLのみ取得
      link.to_s.match(/¥/bestsellers/)
    }
  end
  anemone.on_every_page do |page|
    puts page.url
  end
end
```

◆ anemone-forcus_crawl.rbの実行例

```
$ ruby anemone-forcus_crawl.rb
http://www.amazon.co.jp/gp/bestsellers/
http://www.amazon.co.jp/gp/bestsellers/ref=zg_bs_tab/377-0656698-0299050
http://www.amazon.co.jp/gp/bestsellers/electronics/ref=zg_bs_electronics_home_all/377-0656698-
0299050?pf_rd_m=AN1VRQENFRJN5&pf_rd_s=center-2&pf_rd_r=1F4ZZQK1AS7J44Q9EQE7&pf_rd_
t=2101&pf_rd_p=99273989&pf_rd_i=home
http://www.amazon.co.jp/gp/bestsellers/videogames/ref=zg_bs_videogames_home_all/377-
0656698-0299050?pf_rd_m=AN1VRQENFRJN5&pf_rd_s=center-1&pf_rd_
r=1F4ZZQK1AS7J44Q9EQE7&pf_rd_t=2101&pf_rd_p=99274349&pf_rd_i=home

~省略~
```

取得対象のURLを絞り込む、もしくは除外した後で、on_every_pageメソッドもしくはon_pages_likeメソッドで取得したページに対しての処理を記述します。on_every_pageは渡されたページすべてに対して処理を実行します。on_pages_likeは正規表現で一致したパターンのみに処理をします。

focus_crawlとon_pages_likeの使い分け

focus_crawlとon_pages_likeは、実質的に果たす機能としては似ています。しかし、on_pages_likeの場合はページを取得してから処理をするかどうかの判断を行います。focus_crawlは対象に一致しない場合は、そもそもページの取得処理も行いません。そのため、focus_crawlで取得対象のページを絞り込む方が、効率的なクローリングが行えます。

on_pages_likeの使い方としては、重複しているページの除去などに使います。例えば、「http://example.com」と「http://example.com/」の2つのURLがあった場合、Anemoneは別のページとして捉えます。この2つはHTTPの301リダイレクトされる前と後ですので、どちらか一方を処理すればよいです。そのような処理に、on_pages_likeメソッドで、最後にスラッシュで終わるURLのみを対象にするといったことが可能になります。

Anemoneのページ解析機能

Anemoneが取得したページの情報は、Anemone::Pageを実体化したオブジェクトに格納されます。オブジェクトから参照できる属性は、以下のとおりです。読み書きの権限の「R」は読み込み可能、「W」は書き込み可能を示しています。

Anemone::Pageの属性一覧

属性名	読み書きの権限	説　明
body	R	レスポンスで返ってきたHTMLそのもの
code	RW	HTTPのレスポンスコード
data	RW	OpenStruct形式の構造体に保存されたデータ
depth	RW	クローリングを開始したURLを起点にした階層の深さ
error	R	エラー
headers	R	HTTPのレスポンスヘッダー
redirect_to	R	リダイレクトされた場合のURL
referer	RW	リファラー
response_time	RW	レスポンスの応答時間（ミリ秒単位）
url	R	ページのURL
visited	RW	訪問ずみか否か（Boolean）

またAnemone::Pageの主なインスタンスメソッドは、以下のとおりです。

Anemone::Pageの主なインスタンスメソッド

メソッド名	機　能
content_type()	ページのコンテンツタイプ（種類）を返す
doc()	HTMLのbodyを、Nokogiri形式にして返す
links()	ページ中の<A>タグのリンク先の一覧をリストで返す

Anemone::Pageに関するメソッドで一番多く利用するのは、linksです。このメソッドは、ページ中に含まれている<A>タグのリンク先をすべて取得し、配列として返します。リンク先の取得や巡回先の選択など、さまざまな用途で使います。

なお、Anemone内に組み込まれたNokogiriを利用するdocメソッドについては、UTF-8以外の日本語の文字コードを処理する場合には文字化けが発生する可能性があり、別途対処が必要です。詳細は「3-3 文字コードの対処法」（→p.156）にて説明します。

Anemoneのストレージ機能

Anemoneは取得したページを保存したうえで処理します。ストレージは複数利用可能で、目的に応じて使い分けることになります。ストレージの指定をしない場合は、取得したデータはメモリ内に保存されます。処理対象が多いほどメモリを利用し、プログラムを実行しているPCにも影響を与えます。

メモリ以外のストレージに保存したページは、クローラーのプログラムが終了した後も再利用可能です。取得対象が多い場合や、定期的・継続的にクローラーを動かす場合は、ストレージを利用するのがよいでしょう。

Anemone標準で利用可能なストレージ一覧

ストレージ名	概　要
PStore	標準添付ライブラリのPStoreを利用して、バイナリー形式で保存
Sqlite	DB（Sqlite）形式で保存
MongoDB	NoSQLであるMongoDBに保存
Redis	KeyValueStoreであるRedisに保存
Tokyo Cabinet	Memcached互換のKeyValueStoreであるTokyo Cabinetに保存
Kyoto Cabinet	Memcached互換のKeyValueStoreであるKyoto Cabinetに保存

Anemoneにはストレージに対するインターフェースが用意されています。あらかじめ用意されているストレージ以外でも、インターフェースの仕様に従ってカスタムライブラリを用意することにより、独自にストレージを追加することも可能で

す。ストレージについては、「4-1 データの保存方法」(→p.198)で取り上げます。

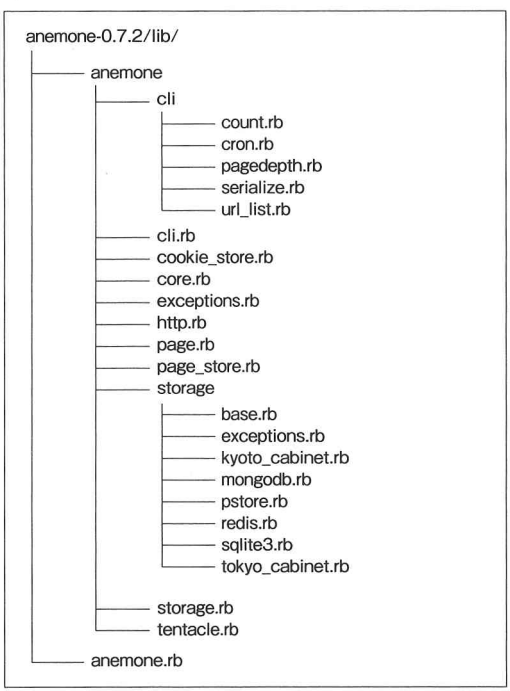
なお、ストレージに対する処理については、Anemone::Coreのafter_crawlメソッドで記述します。

2-2-2 Anemoneの内部構造

Anemoneは、各種制御および巡回の制御・データ取得を行うAnemone::Coreと、データ解析を行うAnemone::Page、データ保存を行うAnemone::Storageを中心に構成されています。

本書では、それぞれのソースコードについては詳しく説明はしません。しかし、Anemoneのソースコードは簡潔な記述で動作が記載されています。ぜひ一度、ソースコードを読むことをお勧めします。 Anemone-0.7.2でのディレクトリとファイル構成は、下記のとおりとなっています。

Anemone-0.7.2でのディレクトリとファイル構成



2-2-3　Anemoneの実行モデル

Anemoneの主要機能を理解したうえで、次は実行モデルを学びましょう。

Anemoneには、クローラーを作るうえで必要となる巡回・解析・保存といった基本機能を実装するメイン処理と、メインの処理を補完する細かい処理があります。まずはメイン処理の流れを理解することで、Anemoneの使い方を覚えましょう。

Anemoneのメイン処理の実行順序

Anemoneのメイン処理は、下記のような流れになります。

- ①巡回対象サイトのURLを指定
- ②除外対象ページのURLパターンを指定
- ③巡回対象ページのURLを指定
- ④取得したページに対して、正規表現で一致したページのみ処理
- ⑤取得したページすべてに対しての処理
- ⑥ストレージに対する処理

この流れに対応するスクリプトは、次のような構成になります。

Anemoneのメイン処理

```
require 'anemone'
  .
# ①巡回対象サイトのURLを指定
Anemone.crawl("http://example.com/") do |anemone|
  # ②除外対象ページのURLパターンを指定
  anemone.skip_links_like /除外対象のURLパターン/

  # ③巡回対象ページのURLを指定
  anemone.focus_crawl do |page|
    page.links  # => Array of links
  end

  # ④正規表現で一致したページのみ処理
  anemone.on_pages_like(/処理対象のURLパターン/) do |page|
    # ページに対する処理の記述
  end

  # ⑤すべてのページに対しての処理
  anemone.on_every_page do |page|
    # ページに対する処理の記述
  end

  # ⑥ストレージに対する処理
```

```
  anemone.after_crawl do |page|
    # ストレージに対する処理の記述
  end
end
```

まず「①巡回対象サイトのURLを指定」で、起点となるURLを指定します。対象は1つでも複数でも可能です。次に「②除外対象ページのURLパターンを指定」で、スキップするURLを正規表現で指定します。そして「③巡回対象ページのURLを指定」で、ページ中のリンク先から次に巡回するページの絞り込みを行います。この処理が不要な場合は記述しません。

取得したページに対して、「④正規表現で一致したページのみ処理」、もしくは「⑤すべてのページに対しての処理」でページに対する処理を記述します。この処理が、実際のクローラーを作る際にメイン処理となる部分です。通常、記述するのはどちらか一方ですが、どちらかを必ず記述することになります。

最後に「⑥ストレージに対する処理」で、後処理としてストレージに保存する際のロジックを記述します。後処理が不要の場合は記述も不要です。

全体の処理としては巡回対象ページの指定からストレージに対する処理の部分をループして、対象ページがなくなるまで繰り返します。

その他の処理の記述

メイン処理以外にも、Anemoneはいくつかのメソッドが利用可能です。また、パラメータとしていくつかの初期値を与えることができます。次のスクリプトは、オプション項目を指定して、Anemoneを実行している例です。

Anemoneのメイン以外の処理

```
require 'anemone'

opts = {
  :user_agent => "MyCrawler/0.00",
  :skip_query_strings => true,
  :delay => 1,
  :storage => Anemone::Storage.MongoDB,
  :depth_limit => 1,
}
Anemone.crawl("http://example.com/", opts) do |anemone|

  # 処理

end
```

メイン処理以外のメソッドおよびパラメータの使い方については、この後の実際のサイトを対象にクローラーを作成する際に、順次説明していきます。またオプションについては、「4-5 Anemoneのオプション一覧」(→p.240)ですべて紹介します。

2-3 Anemoneのインストール(Windows編)

このセクションでは、WindowsへのAnemoneのインストールを説明します。既にWindows上にRubyの環境が準備されていることを前提とします。筆者は、Windows7(64ビット版)上にRuby 2.0.0-p451(32ビット版)環境を用意し検証しています。RubyはRubyInstallerが提供しているビルドずみのWindows版バイナリーを利用しています。Rubyのインストールがまだの場合は、下記URLよりダウンロードし、インストールを行っておいてください。

■ RubyInstaller
URL http://rubyinstaller.org/

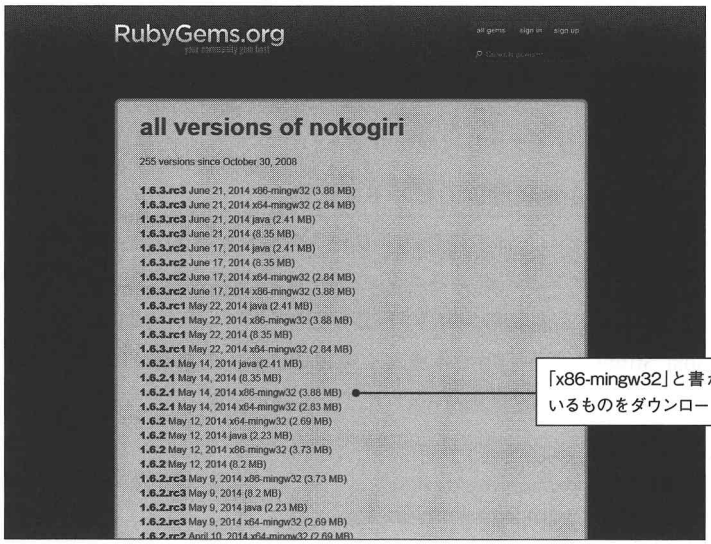
2-3-1　Nokogiriのインストール

Anemoneのインストールに先立って、Nokogiriをインストールします。通常、Gemからインストールする場合は、依存関係のあるライブラリが一括でインストールされます。しかし、Nokogiriはインストールの段階でエラーが起こる可能性が高いです。Nokogiriのビルドには、native extensionという形でlibxml2とlibxsltという2つのDLLとそれぞれのヘッダーファイルを利用します。この2つのライブラリがWindows上にない場合は、自身でインストールしたうえでパスの設定などが必要になります。インストールに慣れていない人には、非常に難易度が高いと思われます。

32ビット版Ruby限定ですが、Nokogiriのインストールを比較的簡単に行う方法があります。Windows用にビルドずみのNokogiriのバイナリーファイルをダウンロードして、そのままインストールする方法です。Nokogoriのバイナリーファイルは RubyGemsのサイトからダウンロードできます。最新バージョンのNokogiriから、「x86-mingw32」と書かれているものをダウンロードします。

■ Nokogiriのダウンロードページ(RubyGems)
URL https://rubygems.org/gems/nokogiri/versions

RubyGemsサイト(Nokogiriダウンロードページ)



インストールの際には、ダウンロードしたファイルのパスを指定します。下記の例は、モジュールのダウンロード完了後に、ダウンロードフォルダーに移動し、gem installコマンドでターゲットにダウンロードしたモジュールを指定してインストールする例です(フォルダーならびにバイナリーファイルのバージョンは各自の設定に合わせてください)。

Nokogiriのインストール

```
C:¥work> cd C:¥Users¥YourUserNmae¥Downloads
C:¥Users¥YourUserNmae¥Downloads> gem install nokogiri-1.6.1-x86-mingw32.gem
```

2-3-2 Anemoneのインストール

Nokogiriのインストールが完了した後に、Anemoneのインストールを行います。今回は、gemで指定するだけなので、簡単に終わります。

gemでのインストールは、バージョンを指定しないと自動的に最新のバージョンがインストールされます。2014年7月現在のAnemoneの最新バーションは、0.7.2です。次の例は、バージョン指定で「0.7.2」をインストールしています。

◆ Anemoneのインストール

```
C:¥work> gem install anemone -v 0.7.2
```

エラーもなく最後に「gem installed」と表示されれば、無事成功です。環境によってインストールされるライブラリは異なります。

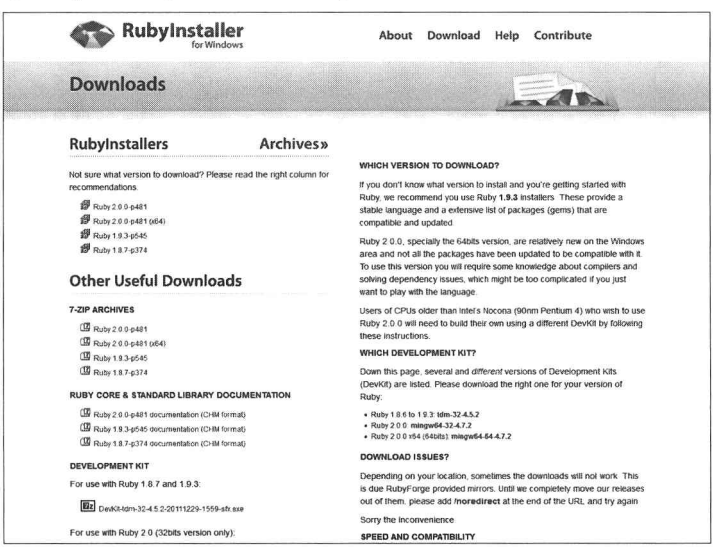
2-3-3　コンパイルツールを利用してビルドする場合

64ビット版のRubyを利用している場合は、ソースからビルドする必要があります。ビルドのためには、コンパイルするためのツールが必要です。Windowsの場合は、RubyInstaller (→p.74) にある「DEVELOPMENT KIT」という開発ツールやMicrosoftのVisual Studioなどが利用できます。本書ではビルド手順は割愛します。比較的手軽という点では、DEVELOPMENT KITがお勧めです。

■ DEVELOPMENT KITのダウンロード

URL **http://cdn.rubyinstaller.org/archives/devkits/DevKit-mingw64-32-4.7.2-20130224-1151-sfx.exe**

◆ RubyInstaller (ダウンロードページ)



DEVELOPMENT KITは、後の章で使用するライブラリのインストールの際に必要なるケースがあります。ここでぜひ、インストールをしておいてください。

以下は、DEVELOPMENT KITを「C:¥tools¥development-kit」に解凍した場合の例です。フォルダーは各自の環境に合わせてください。

◆ DEVELOPMENT KITのインストール

```
C:¥work> cd C:¥tools¥development-kit
C:¥tools¥development-kit> ruby dk.rb init
C:¥tools¥development-kit> ruby dk.rb review
C:¥tools¥development-kit> ruby dk.rb install
```

インストール後は、環境変数のPATHに「C:¥tools¥development-kit¥bin;」を追加してください(フォルダーは各自の環境に合わせてください)。

バイナリーファイルの名前

Windowsのバイナリーファイルは、ビルド方法やアーキテクチャによって名前が異なります。mswin32やmingw32はビルド方法を表しています。「mswin」は、Microsoft Visual StudioのVisual C++によってビルドされたものです。「mingw」は、Minimalist GNU for Windowsの略でありGNU GCCによりビルドされています。後ろの32は、Win32 APIのヘッダーを利用していることを表しています。また、x86やx64は命令セットアーキテクチャであり、x86は32ビット環境を示します。つまりx86-mingw32は、GNU GCCによって32ビット版でビルドされたWindows向けバイナリーということになります。

WindowsのSSLのルート証明書の配置

RubyからHTTPSのサイトにアクセスする際には、相手先の証明書の検証が行われます。その際に、Rubyに対してルート証明書を指定しておかないと、証明書の検証が行えずにエラーになります。

対処法としては、証明書の検証をスキップする方法と、ルート証明書をダウンロードして指定しておく方法があります。基本的には、後者の方法がよいでしょう。ルート証明書はいろいろな所から取得できますが、今回は次のサイトから取得します。

■ SSLのルート証明書の入手先
URL http://curl.haxx.se/ca/cacert.pem

取得した後に、任意のフォルダーに保存しておきます。今回は「C:¥tools¥Ruby200¥」の下に保存してあるとします(フォルダーは各自の環境に合わせてください)。ファイル名は「cacert.pem」とします。

まず、現在のルート証明書へのパスを確認します。

```
C:¥work> ruby -ropenssl -e "p OpenSSL::X509::DEFAULT_CERT_FILE"
```

次のように表示されるはずです。

```
"C:/Users/Luis/Code/luislavena/knap-build/var/knapsack/software/x86-windows/
openssl/1.0.0l/ssl/cert.pem"
```

ほぼ100％存在しないパスだと思うので、パスを変更します。パスの変更には、環境変数で「SSL_CERT_FILE」を追加し、SSLルート証明書のパス（この例ではC:¥tools¥Ruby200¥cacert.pem）をセットします。

2-4 Anemoneのインストール（Mac編）

このセクションでは、MacへのAnemoneのインストール手順を説明します。既にMac上にRubyの環境が準備されていることを前提とします。筆者の環境は、Mac OS X Mountain Lion上のRuby 2.0.0-p353（64ビット版）で検証しています。RubyはRVMを利用してインストールしています。

2-4-1　libxmlとlibxslt、libiconvのインストール

Windows同様に、Anemoneのインストールには Nokogiriが必要となります。Nokogiriはlibxmlとlibxsltおよびlibiconvを利用します。Windows版と同様にRubyGemsのサイトからビルドずみのモジュールはダウンロード可能なものの、インストール時にnative extensionsのビルドが発生します。そのため、MacでNokogiriを利用する場合は、libxmlなど必要なモジュールのインストールが必要となります。

Macに該当モジュールをインストールするには、いくつかの方法があります。本書では「Homebrew」を利用します。なお、Homebrew自体のインストール方法および利用方法は、割愛します。

◆ Nokogiriに必要なモジュールのインストール

```
$ brew install libxml2 libxslt
$ brew link libxml2 libxslt
```

Homebrew

Homebrewは、Mac OS X用のパッケージ管理ソフトです。一般的にソフトウェアをインストールする場合は、それぞれごとの手順に従ってインストールする必要があります。また、ソフトウェアの依存関係のため、他のソフトウェアを先にインストールする必要がある場合もあります。Homebrewなどのパッケージ管理ソフトを利用することにより、インストールに関わる作業をコマンド1つで実行することができます。

Macの場合、Homebrewに先行してMacPortsというパッケージ管理ソフトが主流でした。MacPortsは、システムを依存関係を最小にするために、システムにインストールずみのソフトウェアがあっても利用せずMacPortsで管理しているソフトウェアのみを使うような仕組みになっています。そのため、インストールするソフトウェアの数が増え、導入への時間もかかります。これに対して、Homebrewは既存のソフトウェアをできるだけ利用するような設計になっていて、インストール数も少なく手軽に導入できるようになっています。その軽量さが好評で、利用者が大幅に増えています。

■ Homebrew
URL http://brew.sh/index_ja.html

また、いくつかのRubyのGemライブラリをインストールする際には、ビルドツールが必要になります。Macのビルド環境は、Xcodeをインストール後にコマンドラインツールをインストールすることで可能となります。しかし、Xcodeは巨大なアプリケーションなので、インストールや更新が大変です。OS X Maverickであれば、Homebrewを使うことにより、次のコマンド1つでビルド環境を構築することができます。

```
$ brew install apple-gcc42
```

MacのSSLのルート証明書の配置

Macの場合にも、Rubyに対してルート証明書が必要になります。Windowsの時と同様に、所定の場所に証明書を配置すれば検証ができるようになります。

```
$ ruby -ropenssl -e "p OpenSSL::X509::DEFAULT_CERT_FILE"
```

現在の証明書のパスを確認すると、次のように表示されるはずです。

```
"/etc/openssl/cert.pem"
```

この「/etc/openssl」の下に「cert.pem」というファイル名で配置します。

```
$ wget http://curl.haxx.se/ca/cacert.pem
$ sudo mv cacert.pem /etc/openssl/cert.pem
```

2-4-2　Anemoneのインストール

libxmlとlibxslt、libiconvのインストールが完了した後に、Anemoneのインストールを行います。Windows版のインストール方法と違い、Nokogiriはバイナリー版を利用しません。そのため、gemでAnemoneを指定して、Nokogiriと一緒にインストールします。gemでのインストールは、バージョンを指定しないと自動的に最新のバージョンがインストールされます。2014年7月現在のAnemoneの最新バージョンは、「0.7.2」です。

◆ Anemoneのインストール

```
$ gem install anemone
Fetching: nokogiri-1.6.1.gem (100%)
Building native extensions. This could take a while...
Successfully installed nokogiri-1.6.1
Fetching: robotex-1.0.0.gem (100%)
Successfully installed robotex-1.0.0
Fetching: anemone-0.7.2.gem (100%)
Successfully installed anemone-0.7.2
Parsing documentation for anemone-0.7.2
Installing ri documentation for anemone-0.7.2
Parsing documentation for nokogiri-1.6.1
unable to convert "¥xCF" from ASCII-8BIT to UTF-8 for ../../extensions/x86_64-darwin-12/2.0.0-static/
nokogiri-1.6.1/nokogiri/nokogiri.bundle, skipping
unable to convert "¥xCF" from ASCII-8BIT to UTF-8 for lib/nokogiri/nokogiri.bundle, skipping
Installing ri documentation for nokogiri-1.6.1
Parsing documentation for robotex-1.0.0
Installing ri documentation for robotex-1.0.0
3 gems installed
```

エラーもなく最後に「gem installed」と表示されれば、無事成功です。環境によりインストールされるライブラリが異なるので、上記のメッセージと異なっていても問題はありません。上記の例ですと、マニュアルの一部の文字コードがコンバー

トできないために警告が発生しています。実害はないので、無視しても問題ありません。

※

クローラーを開発する環境が整いました。いよいよ次のセクションからは、実践的なクローラーの開発を学びます。

2-5 基本的なクローラーを作成する

ここまでクローラーの構造およびAnemoneの基本的な動作について学んできました。このセクションでは実際のWebサイトを対象にして、実用的なクローラーの作成を行います。

クローリングの対象は「Amazon.co.jp」を利用し、「本」ならびに「Kindleストア」カテゴリを対象にジャンルごとのベストセラーを取得します。対象は実際に稼働しているWebサイトなので、サイトの構造などが変化する可能性があります。これから記載する内容も、無効になる可能性があります。しかし、クローラー作成とは変化に対応していくことです。素早く変化に対応するためには、取得対象のサイトの構造を的確に把握することと、サイトに対して効率的にクローラーを巡回させて構造変化に強い方法でデータを抜き出す手法が必要です。実際にクローラーを作る過程で、その経験を積みます。

それでは、実際にクローラー作成を通じて、必要な知識やノウハウを習得していきましょう。

2-5-1 Amazonからジャンルごとのベストセラーを取得する

Amazon.co.jpのベストセラー（http://www.amazon.co.jp/gp/bestsellers/）から、「本」と「Kindleストア」のカテゴリを対象に、総合ランキングと「ビジネス・経済」や「コンピュータ・IT」など各ジャンルごとのランキングを取得します。

2014年7月現在、Amazon.co.jpのベストセラーは本やKindleストアのベストセラーの他に、DVDや家電・カメラ、おもちゃなど31のカテゴリにわたり、それぞれ1時間ごとに刻々と更新されています。

Amazon.co.jp（ベストセラー）

取得の目的と対象データ

今回作成するクローラーは、売れ筋書籍を継続的に取得し、売れ筋の変化および傾向を分析するための情報を収集することを目的とします。そのためにAmazon.co.jpの ベストセラーから定期的にジャンルごとの順位を取得します。

取得したデータはデータベースに格納し、分析などで活用しやすい形で保存します。データは、日毎にカテゴリ名と書籍名・ASIN・順位をそれぞれ保存します。なお、ASINとはAmazon内で商品を一意に識別するためのコードです。

収集したデータは、下記の表のような形で集計できるようにデータを収集します。

取得データの形式

カテゴリ名	書籍名	ASIN	順位
本/コンピュータ・IT	第五の権力―Googleには見えている未来	4478017883	1
本/コンピュータ・IT	マッチ箱の脳（AI）―使える人工知能のお話	B00DT4DY0M	2
本/コンピュータ・IT	闘うプログラマー[新装版] ビル・ゲイツの野望を担った男達	B00GSHI04M	3
本/ビジネス・経済	嫌われる勇気―自己啓発の源流「アドラー」の教え	4478025819	1
本/ビジネス・経済	惣菜弁当の殿堂 味付けは親心、盛り付けは活け花の心得 主婦の店さいち惣菜弁当全集	4889271635	2
本/ビジネス・経済	まんがでわかる7つの習慣	4800215315	3
Kindleストア/ビジネス・経済	捨てる生き方　幸運、金運、人運、引き寄せの法則	B00INTIUS4	1
Kindleストア/ビジネス経済	「原因」と「結果」の法則	B008BCC9YO	2
Kindleストア/ビジネス経済	ダンナ様はＦＢＩ	B009CTXC8W	3

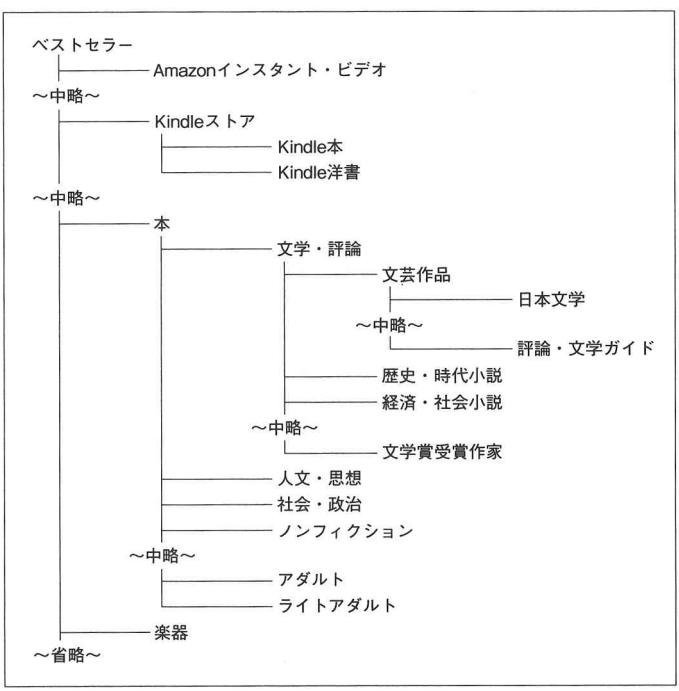
ベストセラーのサイトの構造

クローリングするには、まず対象とするサイトの構造がどうなっているのかを把握する必要があります。ここでのサイトの構造とは、階層構造とURLの体系を指します。

クローリングの対象とするAmazon.co.jpのベストセラーページは、カテゴリごとに構成されています。カテゴリは、「Kindleストア」や「本」、「DVD」など30以上あり、ベストセラーページのトップページの下にカテゴリトップという形で存在します。また、カテゴリ内でもさらに階層構造となっていて、例えば「本」カテゴリの場合は「文学・評論」や「人文・思想」など30以上のサブカテゴリを持ち、それぞれがさらにサブカテゴリを持つという構造になっています。サブカテゴリの数や階層はカテゴリによってもまちまちですが、3層ほどの構造になっていることが多いです。

ベストセラーの階層構造をツリー形式で表すと、次のような形になります。

▼ベストセラーのサイトの階層構造



上記のような階層構造に対して、URL体系はどうなっているのでしょうか。まずトップであるベストセラーページのURLは、

- http://www.amazon.co.jp/gp/bestsellers/

であり、Kindleストアや本、DVDといったカテゴリごとのトップページは、次のような形になります。

- http://www.amazon.co.jp/gp/bestsellers/digital-text/
- http://www.amazon.co.jp/gp/bestsellers/books/
- http://www.amazon.co.jp/gp/bestsellers/dvd/

つまり、「digital-text」「books」「dvd」などカテゴリを表す単語がベストセラーページのURLの後に付加されます。

これに対して、カテゴリの下層のサブカテゴリの場合は、その後にサブカテゴリを表すIDが追加されます。またツリー形式で3層目のサブカテゴリの場合でも、「カテゴリ名/サブカテゴリIDと」いった形になります。「本」カテゴリ配下の「文学・評論」の「経済・社会小説」の場合は、次のようになります。

- http://www.amazon.co.jp/gp/bestsellers/books/507214/

つまりサブカテゴリIDは階層にかかわらず内部的にはフラットな形で保持され、URL構造としてはカテゴリおよび2層のサブカテゴリしか存在しません。クローラー作成の際には、上記のURL体系を念頭に置いたうえで、URLに対する巡回先の追加・除外といった形でプログラミングを組んでいきます。

またブラウザで巡回していた場合は、URLのなかに上記のカテゴリ名とサブカテゴリIDの他に「ref」という参照元を指し示す引数が付きます。これはAmazonが分析に利用していると推測され、特に付加しなくても影響はありません。

厳密に言うとURLパラメータは、

- http://指定のURL ～パラメータ

といった形で、「～」以降の部分がパラメータになります。

一方で、URLの一部に意味を持たせてパラメータのように利用する方法もあります。このような場合、クローラーの挙動に影響を与える可能性もあります。

ベストセラーのページの構造

ベストセラーのサイトの構造に続き、ページの構造です。カテゴリやサブカテゴリ共通で、1ページに1位から20位まで20件表示されます。ページの切り替えを行うことで、最大100位まで表示可能です。また順位やタイトルといった要素は、それぞれCSSのクラスという形で識別可能になっています。

一方でKindleストアのベストセラーについては、一部構造が異なります。ページごとに20件表示という点では同一ですが、Kindleストアの場合は有料・無料それぞれのランキングが同一カテゴリ内に並列してあります。 ページの構造については、「2-5-3 スクレイピング機能の作成」(→p.91)にて詳しく説明します。

Amazon.co.jp ベストセラーのカテゴリ別ページ

取得対象の決定

ここまででAmazon.co.jp ベストセラーのサイトと各ページの構造がある程度わかりました。巡回対象の絞り込みのためにクローリングの対象を、「本」と「Kindleストア」のカテゴリそれぞれの、「ビジネス・経済」と「コンピュータ・IT」の上位20件ずつと設定します。Kindleストアについては、有料と無料それぞれを取得することとします。次のセクション以降で、目的に沿ったクローラーの作成を開始します。

2-5-2　クローリング機能の作成

今回のAnemoneを利用したクローラーは、まず巡回（クローリング）機能から作ります。巡回機能を作るうえでは起点となるURLを決定し、その後にクローラーが目的とするページをスキなく巡回するように、除外対象のURLや巡回対象のURLの絞り込みの調整などを行っていきます。最初から解析機能の実装までを一度に行うのではなく、まずは想定どおりにクローラーが動いているかをURLのみ表示させるといった方法で、確認しながら作っていくとよいでしょう。

初期URLの指定と巡回パラメータの決定

Anemoneを起動するには、Anemone::Coreのクラスメソッドであるcrawlかnewを利用します。クローラーの一連の機能をモジュール化する場合は、newを利用するのが便利です。簡易クローラーを作るのであれば、crawlを使うのが手軽でお勧めです。今回は、crawlを利用します。

crawlメソッドの書式は、下記のようにクローラーの起点となるURLとオプションを引数とします。またURLについては、urlsという変数名のとおり複数のURLを受け付けます。複数の場合は、配列にして渡します。オプションについては後で詳細に説明します。

```
crawl(urls, opts = {}) {|core if block_given?| ...}
Convenience method to start a new crawl
```

構文 crawlメソッド

```
crawl(起点URL, オプション)
```

それでは、実際に巡回だけをするクローラーを作ってみましょう。クロールの起点および巡回の対象はAmazon.co.jpの「本」カテゴリと「Kindleストア」カテゴリの「和書」です。

巡回クローラー　just-crawling.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

#クロールの起点となるURLを指定
urls = [
  "http://www.amazon.co.jp/gp/bestsellers/books/",
```

```
  "http://www.amazon.co.jp/gp/bestsellers/digital-text/2275256051/"]

Anemone.crawl(
  urls,:depth_limit => 1, :skip_query_strings => true) ●―――❶
  do |anemone|

  # 巡回先の絞り込み
  anemone.focus_crawl do |page| ●―――❷
    page.links.keep_if { |link|
      link.to_s.match(
        /¥/gp¥/bestsellers¥/books|¥/gp¥/bestsellers¥/digital-text/)
   }
  end

  # 取得したページに対する処理
  anemone.on_every_page do |page| ●―――❸
    puts page.url
  end
end
```

Anemone.crawlでURLとオプションを渡します(❶)。今回は、depth_limitパラメータで巡回する階層の深さを1層に制限しています。またskip_query_stringsパラメータを有効にしてURLパラメータによる区別をしないようにしています。URLパラメータの区別とは、「www.example.com?page=1」と「www.example.com?page=2」といった引数が違うURLを同じURLとするかしないかということです。skip_query_stringsをtrueにすると、引数の違いを無視して同じURLとして扱います。

巡回先の絞り込みはfocus_crawlメソッドで行います(❷)。取得したHTMLのリンク先を一覧化し、所定の条件に一致した場合のみ巡回先としています。今回は、「/gp/bestsellers/books」と「/gp/bestsellers/digital-text」のみが対象となります。

その後にon_every_pageメソッドを利用し、取得したページすべてに対してURLを表示させています(❸)。このURLが実際に巡回したURLとなります。

それでは、実行してみましょう。

◆ just-crawling.rbの実行例

```
$ ruby just-crawling.rb
http://www.amazon.co.jp/gp/bestsellers/books/
http://www.amazon.co.jp/gp/bestsellers/digital-text/2275256051/
http://www.amazon.co.jp/gp/bestsellers/books/571584/ref=zg_bs_nav_b_1_b/378-6122774-
0496919
```

```
~中略~

http://www.amazon.co.jp/gp/bestsellers/digital-text/2293031051/ref=zg_bs_nav_
kinc_2_2275256051/375-1791193-8865624
```

おそらく60以上のURLが表示されると思います。これらのURLは、「本」と「Kindleストア」カテゴリ内のサブカテゴリです。今回はサブカテゴリのなかから「ビジネス・経済」と「コンピュータ・IT」のみを対象とするため、このままでは対象外のページまで処理することになります。それを防ぐために、取得したページから処理対象のページを絞ることとします。

処理対象の絞り込み

取得したページのなかから処理対象のページを絞り込むのは、on_pages_likeメソッドを利用します。このメソッドの絞り込み方は、URLに対して正規表現で一致したもののみ処理をするという形です。

構文 on_pages_likeメソッド

```
on_pages_like(正規表現のパターン)
```

今回対象のURLのなかで絞り込みに使える情報としては、サブカテゴリを示しているであろうIDしかありません。ここでは仮に「サブカテゴリID」と呼びます。そこで、ブラウザで取得対象のページにアクセスして、URLからサブカテゴリIDを抜き出します。

「本」と「Kindleストア」のカテゴリそれぞれの、「ビジネス・経済」と「コンピュータ・IT」サブカテゴリのページにブラウザでアクセスし、URLを取得します。

■本：ビジネス・経済
URL http://www.amazon.co.jp/gp/bestsellers/books/466282/

■本：IT
URL http://www.amazon.co.jp/gp/bestsellers/books/466298/

■Kindleストア：ビジネス・経済
URL http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/

■ Kindleストア：コンピュータ・IT
URL http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/

それぞれ末尾の数字がサブカテゴリIDです。対象のサブカテゴリIDが特定できたところで、巡回クローラーのon_every_pageメソッドを、on_pages_likeメソッドに切り替えてみましょう。「just-crawling.rb」(→p.86)のon_every_pageメソッドを、下記のようにon_pages_likeメソッドに置き換えます(❶)。

絞り込み巡回クローラー just-crawling2.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

#クロールの起点となるURLに指定
urls = [
  "http://www.amazon.co.jp/gp/bestsellers/books/",
  "http://www.amazon.co.jp/gp/bestsellers/digital-text/2275256051/"]

Anemone.crawl (
  urls,:depth_limit => 1, :skip_query_strings => true)
  do |anemone|

  # 巡回先の絞り込み
  anemone.focus_crawl do |page|
    page.links.keep_if { |link|
      link.to_s.match(
        /¥/gp¥/bestsellers¥/books|¥/gp¥/bestsellers¥/digital-text/)
   }
  end

  # 取得したページに対する処理
  PATTERN =
    %r[466298¥/+|466282¥/+|2291657051¥/+|2291905051¥/+]
  anemone.on_pages_like(PATTERN) do |page| ●―――――――――――――――❶
    puts page.url
  end
end
```

このスクリプトを実行すると、次のように目的のページのみ取得できることが確認できます。

♥ just-crawling2.rbの実行例

```
$ ruby just-crawling2.rb
http://www.amazon.co.jp/gp/bestsellers/books/466282/ref=zg_bs_nav_b_1_b/378-8606758-
3657435
http://www.amazon.co.jp/gp/bestsellers/books/466298/ref=zg_bs_nav_b_1_b/378-8606758-
3657435
http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/ref=zg_bs_nav_
kinc_2_2275256051/376-7801353-1407519
http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/ref=zg_bs_nav_
kinc_2_2275256051/376-7801353-1407519
```

今回は練習の意味を兼ねて、巡回先を絞ったうえでさらに処理するページを絞るという形を取りました。しかし、最終的な目的は4ページのみです。この場合は、巡回させずに目標のページのみ取得する方が効率的です。その方法についても試してみましょう。

巡回させない方法

巡回させないで目的のページのみ取得する方法は、実は非常に簡単です。crawlメソッドの初期パラメータに目的のURLをすべて渡します。そのうえでdepth_limitに「0」を渡して、巡回しないように設定します。

ページ指定クローラー　pinpoint-crawling.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466298/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466282/")

Anemone.crawl(urls, :depth_limit => 0) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
  end
end
```

このスクリプトを実行すると、次のような結果となります。

◆ pinpoint-crawling.rbの実行例

```
$ ruby pinpoint-crawling.rb
http://www.amazon.co.jp/gp/bestsellers/books/466282/
http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/
http://www.amazon.co.jp/gp/bestsellers/books/466298/
http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/
```

目的のページが決まっていて少数であれば、ページ指定のクローラーを作った方が効率的です。一方で、不特定のページから条件に当てはまる情報を抜き出すという目的であれば、最初に紹介したとおりに巡回ルールを作る方が適しています。どちらの方法を選択するかは、クローラーの目的やサイトの構造に依存します。

このセクションで思いどおりにクローラーを巡回させることができるようになったと思います。次のセクションでは、取得したページから情報を抜き出すスクレイピング処理の実装をします。

2-5-3 スクレイピング機能の作成

巡回機能が完成したので、次は取得したページから目的とする情報を抜き出してみましょう。今回は、「カテゴリ名」「書籍名」「ASIN」「順位」を取得します。

ランキングページのHTMLの構造

ページのなかから目的の情報を抜き出すには、主に正規表現を使う方法と構文解析ツール（パーサー）を使う方法があります。いずれの方法も対象ページのHTMLの構造を理解しておく必要があります。まずはランキングページのHTMLの構造を見てみましょう。

ブラウザに標準もしくはアドオンの開発ツールを利用することで、簡単にHTMLの構造を見ることができます。Internet Explorerであれば［ツール］→［F12開発者ツール］メニュー、Firefoxであれば「Firebug」、Chromeであれば右クリックの［要素を検証］などで利用できます。ツールの利用については、「3-5-4 簡単なXPathの抽出方法について」（→p.183）で詳しく説明します。

Amazonベストセラーページの HTML構造

```
▼<div class="zg_itemRow">
  ▼<div class="zg_item_normal">
    ▶<div class="zg_image zg_itemLeftDiv_normal">…</div>
    ▼<div class="zg_itemRightDiv_normal">
      ▶<div class="zg_rankLine">…</div>
      ▶<div class="zg_title">…</div>
       <div class="zg_byline">




       エリック・シュミット, ジャレッド・コーエン, 櫻井 祐子</div>
      ▶<div class="zg_reviews">…</div>
       <div class="zg_bindingPlatform">単行本（ソフトカバー）</div>
      ▶<div class="zg_itemPriceBlock_normal">…</div>
      </div>
      <div class="zg_clear"></div>
    </div>
  </div>
```

ランキング部分については、CSSのClass属性「zg_itemRow」の<Div>タグ内にまとめられています。そのなかに、順位は「zg_rankLine」というClass属性の<Div>タグに、書籍名は「zg_title」というClass属性の<Div>タグに記載されています。そして、ASINですが「zg_reviews」というClass属性の<Div>タグのなかに記載されています。しかし、「zg_reviews」は読者からの評価やレビューの情報であり、評価が付いていない書籍については該当の<Div>タグが存在しません。確実に取得するには、商品名をクリックした時に表示されて、その商品の情報が一通り表示される商品詳細ページへのリンクURLからASINのみを抜き出す必要があります。

このようにスクレイピングを実施する場合は、例外ケースを考慮のうえで汎用的に使える方法で実装する必要があります。しかしながら、最初からすべてのケースを考慮して実装するのは難しいです。まず必要最小限の実装をしたうえで、実際に動かしてみて例外ケースを見つけるというトライ＆エラーの方法が有効です。

Anemone と Nokogiri の文字コード

ランキングページのHTMLの構造がわかったので、さっそくスクリプトを作りたいところです。しかし、構文解析を行うNokogiriには、文字コードによって上手く対処できず文字化けするケースがあります。文字化けの対処法としては、Nokogiriに渡す前に文字コード変換をするか、Nokogiriに対して正しい文字コードを教えるかの2つの方法があります。

AnemoneにはNokogiriが組み込まれているものの、文字コードに対する対処を

変更することができません。そのために、無駄が多いものの、Anemone内蔵のNokogiriとは別にNokogiriを別途定義して利用する必要があります。今回対象とするAmazon.co.jpのランキングページも、同様の対処が必要です。スクレイピングと文字コードの問題については、「3-3 文字コードの対処法」(→p.156)にて詳しく説明します。

文字コード変換とカテゴリ名の取得

「pinpoint-crawling.rb」(→p.90)を元に、スクレイピングの実装を行います。文字コードを明示的に「utf-8」と指定し、あわせてスクリプトのファイルの文字コードもUTF-8で保存されていることを確認してください。ファイルの文字コードの指定は利用しているエディタに依存しますが、ファイルの保存時に指定できるものが多いです。

requireの部分に下記のように「nokogiri」と「kconv」を追加します。kconvはRubyの標準添付ライブラリであり、日本語の文字コード変換のライブラリです。

```
require 'anemone'
require 'nokogiri'
require 'kconv'
```

次に文字コードをUTF-8に変換したうえ、Nokogiriでパースしたオブジェクトを生成します。通常の場合は、この処理は不要です。なぜならAnemoneにはAnemone::PageにNokogiriでパースしたオブジェクトを返すdocメソッドが用意されているからです。しかし、Version 0.7.2の時点のAnemoneには、下記の実装のとおり文字コードがUTF-8以外の場合の考慮がありません。そのため、対象ページの文字コードがUTF-8以外の場合は、自前で変換する必要があります。

Nokogiriの実装

```
#
# Nokogiri document for the HTML body
#
def doc
  return @doc if @doc
  @doc = Nokogiri::HTML(@body) if
    @body && html? rescue nil
end
```

自前で文字コードを変換するには、Anemone::Pageのbody属性から生のページデータを取得し、kconvのコンバートメソッド(toutf8)を利用してUTF-8に変換します。それをNokogiri::HTMLのparseメソッドを利用して、Nokogiriオブジェクトを作成します。

```
doc = Nokogiri::HTML.parse(page.body.toutf8)
```

それでは、準備が整ったところで、目的のデータを抜き出しましょう。目的のデータのうち、カテゴリおよびサブカテゴリ名はページに1つです。書籍名・ASIN・順位は、最大で20個まであります。よって、カテゴリ・サブカテゴリ名のみ一度取得し、後のデータはループ処理のなかで取得します。

まずはサブカテゴリの取得です。「コンピュータ・IT」や「ビジネス・経済」といったサブカテゴリ名は、HTMLを見るとID名「zg_listTitle」の<Div>タグ内の<Span>タグ内に配置されています。また、サブカテゴリ名の情報のみだとKindleストアと本の区別がつかないので、カテゴリの情報も取得します。同じくHTMLを見ると、ID名「zg_browseRoot」の<Div>タグ以下に格納されているのがわかります。

HTMLからXPathやCSSの構造を抜き出す方法については、「3-5-2 Nokogiri、XPathの使い方」(→p.172)および「3-5-4 簡単なXPathの抽出方法について」(→p.183)で詳しく解説します。

これまでの一連のスクリプトは以下のとおりです。いったん保存して、実行してみましょう。

カテゴリ名とサブカテゴリ名の取得　scraping.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'nokogiri'
require 'kconv'

urls = []
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466298/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466282/")

Anemone.crawl(urls, :depth_limit => 0) do |anemone|
  anemone.on_every_page do |page|
```

```
    # 文字コードをUTF8に変換したうえで、Nokogiriでパース
    doc = Nokogiri::HTML.parse(page.body.toutf8)

    category = doc.xpath(
      "//*[@id='zg_browseRoot']/ul/li/a").text

    # カテゴリ名の表示
    sub_category = doc.xpath(
      "//*[@id=¥"zg_listTitle¥"]/span").text

    puts category+"/"+sub_category
  end
end
```

文字化けが起こらなければ、下記のような結果が出ます。文字化けが起こった場合は、スクリプトファイル自体の文字コードの確認をしましょう。

◆ scraping.rbの実行例

```
$ ruby scraping.rb
Kindleストア/コンピュータ・IT
本/コンピュータ・IT
Kindleストア/ビジネス・経済
本/ビジネス・経済
```

書籍名と順位の抜き出し

次は、書籍名と順位の抜き出しです。前述のとおりランキングに関する情報は、Class属性「zg_itemRow」の<Div>タグにまとめられています(→p.92)。ここでの処理は、XPathの指定でランキング情報をまとめて取得して、配列に格納します。そのうえで、1件ずつ処理してデータを抜き出します。

なお、Kindleストアについては、有料・無料の2つのランキングが並列で存在します。構造を見ると、Class属性「zg_itemRow」の<Div>タグ以下に含まれているため、配列に追加するだけで取得できます。

有料・無料の書籍名と順位の取得

```
# 一般・Kindleストア有料
items =  doc.xpath(
  "//div[@class=¥"zg_itemRow¥"]/div[1]/div[2]")

# Kindleストア無料
```

```
items +=  doc.xpath(
  "//div[@class=¥"zg_itemRow¥"]/div[2]/div[2]")

items.each{|item|
  # 順位
  puts item.xpath("div[1]/span[1]").text

  # 書籍名
  puts item.xpath("div[¥"zg_title¥"]/a").text
}
```

ASINの取得

ASINの取得方法はいろいろあります。例えば、リンク先の個別ページにはHTML中に明示的にASINが記載されていて、Nokogiriを利用すると簡単に取得できます。しかし、個別ページも取得すると、ランキング分のページアクセスが発生してしまいます。他にもランキングページの<A>タグに記載されている個別ページのURLから取得する方法もあります。この場合は、ランキングページを一度取得するだけで、すべてのASINを取得できます。

今回は効率的にクロールするために、個別ページのURLから正規表現で取得することとします。URLの構造は、下記のとおり「dp/」と「/」に囲まれている部分がASINになっています。

- http://www.amazon.co.jp/商品名/dp/ASIN/

NokogiriのXPathと正規表現を利用して、ASINのみを抜き出します。URLはClass属性「zg_itemRow」の<Div>タグ以下の<A>タグに含まれています。NokogiriでClass属性指定で取得したうえで、そのなかのhref属性をテキストに変換後に正規表現で取得します。

「dp/」と「/」で囲まれた部分がASINなので、正規表現でその間の部分を抜き取るように記載します。今回は正規表現の詳しい解説はしませんので、必要であれば別途調べてください。

```
puts item.xpath(
  "div[¥"zg_title¥"]/a").attribute("href")
  .text.match(%r{dp/(.+?)/})[1]
```

ASIN抜き出し機能も組み合わせて、完成させたのが次のスクリプトです。

ベストセラー情報を取得する

scraping2.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'nokogiri'
require 'kconv'

urls = []
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291657051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/digital-text/2291905051/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466298/")
urls.push("http://www.amazon.co.jp/gp/bestsellers/books/466282/")

Anemone.crawl(urls, :depth_limit => 0) do |anemone|
  anemone.on_every_page do |page|
    #文字コードをUTF8に変換したうえで、Nokogiriでパース
    doc = Nokogiri::HTML.parse(page.body.toutf8)

    category = doc.xpath(
      "//*[@id='zg_browseRoot']/ul/li/a").text

    #カテゴリ名の表示
    sub_category = doc.xpath(
      "//*[@id=¥"zg_listTitle¥"]/span").text
    puts category+"/"+sub_category

    items =  doc.xpath(
      "//div[@class=¥"zg_itemRow¥"]/div[1]/div[2]")
    items +=  doc.xpath(
      "//div[@class=¥"zg_itemRow¥"]/div[2]/div[2]")
    items.each{|item|
      # 順位
      puts item.xpath("div[1]/span[1]").text

      # 書名
      puts item.xpath("div[¥"zg_title¥"]/a").text

      # ASIN
      puts item.xpath("div[¥"zg_title¥"]/a")
        .attribute("href").text.match(%r{dp/(.+?)/})[1]
    }
  end
end
```

実行すると、次のような形式で出力されます。もちろん書名および順位は、実行する時々で変わります。

◆ scraping2.rbの実行例

```
$ ruby scraping2.rb
Kindleストア/ビジネス·経済
1.
世界のエリートが学んできた「自分で考える力」の授業
B00E9QG0TG
2.
幸せな小金持ちという生き方？本田健初期作【完全版】
B00G48XK86
3.
最強の投資家バフェット (日経ビジネス人文庫)牧野 洋
B009SXCWAG
4.
ゼロ秒思考
B00HQ6O7BO
5.
黒本高城剛
B00I9JS2VI
6.
日本に殺されず幸せに生きる方法 (あさ出版電子書籍)
B00CRHZ9JQ

~省略~
```

想定どおりの結果が出てきたでしょうか。ここまでで、クローリングとスクレイピングの基礎について習得できました。次は、別の方法で同様のデータを取得してみましょう。

2-5-4　RSSを利用する方法

前のセクションで、クローリングおよびスクレイピングを駆使したクローラーを作成しました。しかし、Amazonのベストセラーの場合は、ほぼ同様のデータをもっと簡単に取得する方法があります。それはRSSを利用する方法です。

RSSとは、Webサイトの更新情報をまとめるための文章フォーマットで、ニュースやブログなどによく利用されています。RSSは人間よりコンピュータフレンドリーな構造であり、XML形式で記述され文章の1つ1つが意味づけられています。そのため、プログラムから取得や解釈が容易であり、クローラーを作成する際もできるだけRSSを利用する方が簡単で効率よく開発と巡回ができます。

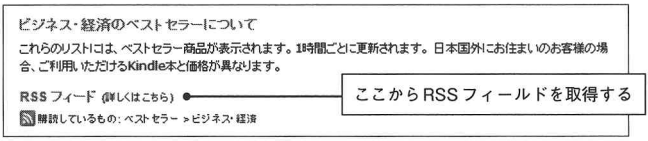
RSS 1.0とRSS 2.0があり、それぞれ別の規格として展開されています。また、

RSSに変わるものとしてAtomと呼ばれる別の規格も策定されています。現在は、それぞれが別の規格として普及しています。またRSS 1.0、RSS 2.0、Atomを総称して「フィード」と呼び、それらを利用してコンテンツ情報を伝えることをフィード配信と呼びます。

Amazon.co.jpのランキングページのRSSフィード

Amazon.co.jpのランキングページには、RSS 2.0のフィードが用意されています。ページの下部にRSSフィードへのリンクがあります。RSSの主要部分を抜き出すと、下記のような形になっています。

▼ランキングページのRSSフィードを取得する



ランキングページのRSSフィード

```
<rss version="2.0">
  <channel>
    <title>ページタイトル</title>
    <link>ページURL</link>
    <description>ページ説明</description>
    <pubDate>ページ生成日</pubDate>
    <lastBuildDate>ページ更新日</lastBuildDate>
    <ttl>有効期間</ttl>
    <generator>Amazon Community RSS 2.0</generator>
    <language>ja-jp</language>
    <copyright>Copyright 2014, Amazon.com</copyright>
    <item>
      <title>書籍1　タイトル</title>
      <link>書籍1　商品ページURL</link>
      <description>書籍1　説明</description>
    </item>
    <item>
      <title>書籍2　タイトル</title>
      <link>書籍2　商品ページURL</link>
      <description>書籍2　説明</description>
    </item>
  </channel>
</rss>
```

title要素にランキングページのカテゴリ名が記載され、<Item>タグの部分でランキング順に書籍データが格納されています。<Item>タグ内の<Title>タグに書名が、<Link>タグに個別ページへのURLが格納されています。この3つのタグを使うことにより、HTMLページのスクレイピング時と同様のデータが取得できます。

なお、格納される書籍数は上位10件のみで、Kindleストアの無料ランキングの情報はありません。このデータをNokogiriを使って解析します。

Nokogiriを利用してRSSフィードを解析する

RSSの実体はXMLです。NokogiriはHTMLの他にXMLの構文解析が可能です。使い方はHTMLの場合とほぼ同様で、最初にパースを行い要素を指定して目的とするデータを取得します。要素の指定の仕方はいくつかありますが、HTMLの時と同様にXPathを利用します。なお、HTTP経由のXMLの取得方法はいくつかありますが、今回はopen-uriを利用します。open-uriは標準添付ライブラリであり、インストール不要で利用できます。

AmazoneのRSSフィードの解析

rss-reader.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url = 'http://www.amazon.co.jp/gp/rss/bestsellers/digital-text/2291657051/'
xml = Nokogiri::XML(open(url).read)

puts xml.xpath('/rss/channel/title').text

item_nodes = xml.xpath('//item')  ●―――❶
item_nodes.each do |item|         ●―――❷
  puts item.xpath('title').text

  # ASIN
  puts item.xpath('link').text.match(%r{dp/(.+?)/})[1]
end
```

◆ rss-reader.rbの実行例

```
$ ruby rss-reader.rb
Amazon.co.jp: Kindleストア > Kindle本 > コンピュータ・ITのベストセラー
#1:「結果を出す人」はノートに何を書いているのか実践編 (Ｎａｎａブックス)
B009SXDIQI
```

```
#2: オブジェクト指向でなぜつくるのか 第2版
B00ISP0QB2
#3: Code Complete 第2版 上　完全なプログラミングを目指して
B00JEYPPOE
#4: 外資系金融のＥｘｃｅｌ作成術―表の見せ方&財務モデルの組み方
B00JM1Z8LY

~省略~
```

xpathの指定で「//item」はこのRSS内のすべてのitem要素を示します(❶)。この指定により、RSS内すべてのitem一覧を取得し配列に格納します。取得したitem一覧の配列を1件ずつ処理し、item内のtitleとlinkを取得しています(❷)。linkから個別ページのURLを取り出し、前の例と同様に正規表現でASINを抽出しています(→p.97)。RSSのパース方法については、「3-4 RSSの解析」(→p.163)でより詳しく解説します。

RSSフィードを利用することの利点と注意点

RSSフィードはその性質上、ページに対するダイジェスト版にあたります。そのため、HTMLページに比べて情報が少なく、全部の情報が載っていない場合が多いです。利用の際は、HTMLとRSSを比較したうえで、目的とするデータの有無を確認してHTMLとRSSのどちらからデータを取得するか決める必要があります。

RSSフィードを利用すると、クローラーの開発効率および巡回効率が高くなるという利点があります。またRSSは、もともとプログラムから取得されることを前提としているために、利用規約でクローラーからのアクセスが許可されていることが多いです。サイト運営側としても、クローラーがHTMLページを巡回するよりも、負荷の少ないRSSフィードを取得する方が好ましいと考えます。

2-6 クローリングができない場合の対処法

いままではクローラーが何の問題もなくサイトを巡回可能でデータ取得できる前提で考えていました。実際には、さまざまな要因でクローラーがデータを取得できないケースがあります。このセクションでは、クローラーが巡回できない場合の原因と対処法について学びます。

2-6-1　クローリングができない原因

クローリングができない場合の原因としては大きく2つあります。

- プロキシなどの利用者側のネットワーク環境の問題でインターネットへアクセスできない
- 認証やユーザーエージェントの問題でWebサイトにアクセス拒否される

ネットワーク環境の問題は、利用者側に起因します。ファイアウォールやプロキシサーバなどのネットワーク機器に起因する問題がほとんどですが、プログラム的に対処できるのはプロキシサーバのみです。本書では、クローラーのプロキシサーバ対応のみ記載します。

認証やユーザーエージェント起因で拒否される場合は、クローラーがWebサイト側のルールやポリシーに従っていないために発生します。この場合は、クローラーの挙動をWebサイト側に合わせることで対処ができます。いくつか代表的なケースを例に原因と対応策を学びます。

2-6-2　プロキシサーバ

プロキシ(Proxy)は代理という意味があり、プロキシサーバは外部のネットワーク接続を行う際に中継サーバとして働きます。一般的には、企業などで内部ネットワークからインターネット接続を行う際に利用され、目的としてはセキュリティの向上や有害サイトの遮断、キャッシュを利用することによる高速通信などがあります。プロキシのなかでWeb閲覧に関するものについては、特にHTTPプロキシと呼ばれます。

企業内のネットワークからクローラーを稼働する場合は、プロキシサーバの設定が必要な場合があります。Anemoneを利用する際のプロキシの設定を見てみましょう。

Anemoneでプロキシサーバを利用する

Anemoneには、起動オプションでプロキシを指定できます。Anemone.crawlの起動オプションの「:proxy_host」にプロキシサーバ名を、「:proxy_port」にプロキシサーバが利用するポート番号を記載します。

```
Anemone.crawl("http://example.com",{
  :proxy_host => 'your_proxy_server',
  :proxy_port => '8888'}) do |anemone|
```

構文 プロキシサーバの指定

```
Anemone.crawl("起点URL",{
 :proxy_host => 'プロキシサーバ名',
 :proxy_port -> 'ポート番号'}
```

オプションの項目が増えたので、オプションを変数に格納したうえで引数に渡してみましょう。渡し方は、オプションの項目名と値をハッシュ（連想配列）として作成し（❶）、Anemone.crawlの第2引数に渡すだけです（❷）。

▌オプションを変数で渡す

```
require 'anemone'

opts = { ●―――――❶
  :proxy_host => 'your_proxy_server',
  :proxy_port => '8888'
}

Anemone.crawl(
  "http://example.com", opts) do |anemone| ●―――――❷
end
```

クローラーの調整をしていると、オプションの指定項目がどんどん増えていきます。可読性を高めるためにも、Anemoneのオプションは変数化するのがよいでしょう。

認証付きプロキシサーバの利用

プロキシサーバのなかには、ユーザーIDとパスワードでの認証が必要なものもあります。その場合はどうすればよいのでしょうか？　プロキシサーバの種類によって設定方法が異なる場合があるために一概には言えませんが、一般的なケースではプロキシサーバのURLに「ユーザーID」と「パスワード」を付加します。

次の例は、プロキシサーバ名が「http://example-proxy.com」で、ユーザーIDが「your_userid」、パスワードが「your_password」の例です。

▌プロキシサーバにユーザーIDとパスワードを渡す

```
require 'anemone'

opts = {
  :proxy_host =>
```

```
    'http://your_userid:your_password@example-proxy.com',
  :proxy_port => '8888'
}

Anemone.crawl("http://example.com", opts) do |anemone|
end
```

構文 ユーザーIDとパスワードの付加

```
:proxy_host => 'http://ユーザーID:パスワード@プロキシサーバ名',
```

今回のケースは、一般的なプロキシサーバでの利用例を記載しています。プロキシサーバの利用方法については、社内のシステム管理者に確認のうえで行ってください。

2-6-3　サイト側にアクセス拒否されるケース

クローリング時にデータが取得できないケースとしては、プロキシサーバなどの利用者側の問題の他に、Webサイト側でアクセスが拒否される場合があります。アクセス拒否のケースとしては、ユーザーエージェントやアクセス回数による制限などのクライアント側の挙動によって制限するものと、認証などのサイト側で制限するものがあります。

ユーザーエージェント

ユーザーエージェントとは、受け手に通知するソフトウェアの名前です。HTTPの場合だと、ブラウザの名前とほぼ同意になります。このユーザーエージェントによるアクセス制限は、2つのケースが考えられます。

1つ目は、サイト側が特定のブラウザから利用されるように、ホワイトリスト方式で利用可能なブラウザを指定する場合です。ホワイトリスト方式とは、アクセス可能なユーザーエージェントをあらかじめリスト化しておく方式です。例えば、「Internet ExplorerとFirefoxのみ許可する」といった使い方です。

2つ目は、ブラックリスト方式です。ブラックリスト方式は、拒否するユーザエージェントの一覧をリスト化する方式です。一般的には、どちらの方式もJavaScriptやCSSの問題で、正常に閲覧できないブラウザを除外するために利用されることが多いです。それ以外にも、特定のクローラーを拒否するために設定されることもあります。

Anemoneの場合は、「user_agent」オプションでユーザーエージェントを指定することができます。以下の例では、ユーザーエージェントとして「my first crawler」を指定しています。

ユーザーエージェントの指定

```
require 'anemone'

opts = {
  :user_agent => 'my first crawler'
}

Anemone.crawl("http://example.com", opts) do |anemone|
end
```

構文 user_agentオプション

```
:user_agent => 'ユーザーエージェント名'
```

Googleによる LWP の拒否

ユーザーエージェントでの拒否の代表的な例としては、GoogleによるLWPの拒否があります。LWPは、Perlの代表的なHTTPクライアントライブラリであり、デフォルトのユーザーエージェントは「libwww-perl/#.###」(#.###はバージョン番号)といった形です。Googleの検索結果へのアクセスに対して、Googleは「libwww-perl」を含むユーザーエージェントを拒否しています。GoogleがLWPを拒否する理由としては、LWPを利用してボットが作られるケースが多く、Googleにとって有害になっているからだと推測されています。LWPの利用者が多いが故の拒否ということです。

アクセス回数制限

Webサイトは、特定のクライアントからの急激なアクセスがあった場合に、単位時間あたりのアクセス数を元に制限することがあります。単位時間は秒間や分間で設定されることが多く、その閾値以下でクローラーのアクセス頻度を設定する必要があります。制限以前の問題として、サイト側に迷惑をかけることを避けるために、また攻撃的なアクセスと見なされないために、アクセス回数の制限は必ず行う必要があります。

Anemoneの場合は、アクセス間隔を「delay」オプションで指定することができます。単位は秒で、指定した秒数の間隔でアクセスします。

アクセス間隔の指定

```
require 'anemone'

opts = {
  :delay => 1
}

Anemone.crawl("http://example.com", opts) do |anemone|
end
```

構文 delayオプション

```
:delay => 秒数
```

アクセス間隔の目安

アクセス間隔は、どれくらいを目安にすればよいのでしょうか。前提としては、取得元のサイトに迷惑をかけないことです。そのうえで、考慮すべき点が2つあります。1つ目は、クローラーが巡回することによりサイトに過度な負荷がかからない程度にすることです。2つ目は、robots.txtなどでサイト側から通知されているアクセス頻度の要請に従うことです。

まずクローラーによる巡回の負荷についてです。巡回の負荷については利用者側からは予測しかできないものの、一般的には動的に生成されるサイトと静的サイトでは大きく違うということを知っておくべきです。動的生成の場合、サーバ側でキャッシュなどの負荷対策をしていないと、1サーバあたりの秒間リクエストの上限が20 ～ 30程度となる可能性もあります。GoogleやYahoo、Amazonなどの大手サイトならいざ知らず、小規模のサイトであれば個人の作ったクローラーで簡単に過負荷の状態に陥ることもあります。攻撃的なアクセスと見なされると、刑事責任も含めて責任を追求される可能性もあります。

robots.txtとは、サイト側からクローラーに対する要求をまとめたものです。そのなかの1つに、crawl-delayという項目があります。crawl-delayは、アクセス頻度に対する指定です。単位は秒で「crawl-delay:1」の場合は、サイトに対する最短のアクセス間隔が1秒となります。一方で、一部のクローラーについては、crawl-delayを秒ではなく分として解釈するものもあります。robots.txtについては、「2-7-1 robots.txt」(→p.110)にて詳しく解説します。

認証サイト

認証が必要なサイトに対してクローリングする際には、2つほどの対処法があります。Cookieに保存された認証情報を利用してログイン状態にしたうえでクローリングする方法と、ブラウザをエミュレートしてユーザーID、パスワードを入力したうえでログインしクローリングする方法です。ここでは、Anemoneを利用してCookieを利用する方法について解説します。ブラウザをエミュレートする方法は、「2-8 ブラウザタイプのクローラー」(→p.114)にて詳しく説明します。

Cookieを利用したログイン方法は、2つの前提が必要になります。1つ目は、対象のサイトに対してブラウザを利用してログインし、ブラウザのツールなどを利用してCookieを取得できるということです。2つ目は、対象のサイトがログイン後一定期間内であれば、Cookie内の情報のみで認証を行うことです。ページ遷移のたびに、前ページの情報が必要な場合には対応できません。

ブラウザのCookieの取得方法としては、ブラウザ所定のCookieの保存ディレクトリに移動しテキストエディタなどで直接開く方法と、ブラウザの拡張ツールなどを利用してCookie情報を抜き出す方法があります。初期の設定などをすれば、後者の方が手軽でしょう。例えばFirefoxの場合、「Firebug」という定番のプラグインを利用することによりページ閲覧中にCookieを確認することができます。Chromeの場合は、「Edit This Cookie」が定番です。下記の図は、FirebugでMixiのCookieを見た例です。

FirebugでCookieを確認する

Firebugで確認したかぎり、mixi.jpの場合はCookie内に「_auid」「_lcp」「session」「stamp」「emid」「vntgsync」という6つのパラメータがあります。このなかで、認証に関係する項目を探し当てる必要があります。

探し方としては、最初にすべての項目をセットするという方法があります。クローラーが認証されることを確認したうえで、1つずつ項目を削除していきましょう。認証ができなくなった場合は、その項目が必要だということです。mixi.jpの

場合は、「emid」「session」「stamp」の3つを利用して認証している模様です。

当然のことながら、どの項目を設定する必要があるのかは、Webサイトごとに違います。一般的には、ユーザーを特定するIDと、それに対応するSessionID、それらの値の正当性を保証する署名を検証するWebサイトが多いようです。

次のスクリプトは、「mixi.jp」に対してブラウザでログインした状態でCookieの情報を抜き出し、Anemoneの「cookies」オプションでセットした例です。Webサイトによっては、accept_cookiesオプションをtrueにする必要があります。必要に応じてセットしましょう。

Cookie情報によるアクセス　scraping-with-cookie.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'nokogiri'
require 'kconv'

urls = []
urls.push("http://mixi.jp/home.pl?from=h_logo")

cookies = {
  :_auid => "xxxxxxxxxxxxxxxxxxx",
  :emid => "yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy",
  :session => "99999_zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz",
  :stamp => "qqqqqqqqqqqqqqqqqqqqqqq",
  :vntgsync => "1"
}
opts = {
  :delay => 1,
  :accept_cookies => true,
  :cookies => cookies,
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    doc = Nokogiri::HTML.parse(page.body.toutf8)
    communityList.each {|community|
      puts community.content
    }
  end
end
```

なお、上記スクリプトでセットしているcookiesの値などは、実行の際に適当なものに置き換えてください。

◆ scraping-with-cookie.rbの実行例

```
$ ruby scraping-with-cookie.rb
WEB事業者の為の定期勉強会 (114)
昔滋賀　今東京 (530)
くたびれたアメリカが好き (3467)
不動産投資フォーラム (6274)
滋賀県なめたら水止める☆ (10172)
```

ここまでで、Webサイト側からのクローラー拒否を回避する方法について学びました。一方で、Webサイトの裏をかくクローラーを作っていたとしても、いずれは問題に発展します。クローラーを作成するうえで重要なのは、Webサイト側のルールに従いながら迷惑をかけず共存できることです。次のセクションでは、ブラウザを作るうえでのモラルやルール、クローラーを巡る事件や法的見解などの事例を紹介します。それを踏まえて、クローラーの巡回作法について学びます。

2-7 行儀のよいクローラーを作るには

クローリングの対象となるWebサイトは著作物です。当然のことながら、クローラーでデータを収集してそのままインターネットで公開することは、著作権違反になります。クローラーの作成・運用については、どういった用途でデータを収集し活用することが可能なのか、ある程度の法的な知識も必要とします。またWebサイト自体も、何らかの意図のもとに公開されているものです。クローラーの巡回が原因で、他の利用者からWebサイトの閲覧が不可能もしくは困難な状況になると、業務を妨害していることになります。最悪の場合は業務妨害に問われることになるので、そのような事態を引き起こさないように細心の注意が必要です。

このセクションでは、行儀のよいクローラーを作るために、基礎的な知識の整理を行います。クローラーの巡回に関する一般的なルールと見なされているrobots.txtと、クローリングで取得したデータの取扱、クローリングがWebサイトに与える影響についてです。なお、筆者は法律についての専門家ではないので、法解釈などは行いません。

2-7-1　robots.txt

Robots Exclusion Standard（RES）もしくは、Robots Exclusion Protocol（REP）は、クローラーがWebサイトを巡回する際に守るべき規約であり、robots.txtはその規約をまとめたものです。YahooやMicrosoftなど検索サイト大手が参加することにより、事実上の業界標準になっています。一方、紳士協定にすぎないので、クローラーに対する強制力はありません。しかしながら、クローラーを作る際にはトラブルを避けるためにも、最大限robots.txtを尊重すべきです。そのためには、基本的なrobots.txtの読み方を身につける必要があります。また、robots.txtに記述された内容に対する対処は、基本的にはクローラー製作者が自分で実装する必要があります。robots.txtによるアクセス禁止のURLを訪問対象外にするなど、robots.txtに従うためのGemライブラリもあるので、必要に応じて利用してみるのもよいでしょう。

robots.txtの基本構造は、対象とするユーザーエージェントの指定と拒否対象のファイル・ディレクトリの指定の2つだけです。以下、実際の具体例で確認しましょう。

すべてのクローラーを拒否

すべてを拒否する場合は、「User-agent」をワイルドカードで指定します。そのうえで、アクセス禁止を示す「Disallow」の対象を「/」（ルート）で指定します。

```
User-agent: *
Disallow: /
```

すべてのクローラーを許可

すべてを許可する場合は、Disallowに何も指定しません。もしくは、robots.txt自体も不要です。

```
User-agent: *
Disallow:
```

すべてのクローラーからjpg形式を拒否

ワイルドカードと拡張子を組み合わせることにより、特定のファイル形式のみ拒否することも可能です。次のように指定することで、jpg形式の画像のクローリングを禁止することができます。

```
User-agent: *
Disalow:/*.jpg
```

また、複数のファイル形式を指定する場合は、対象分だけDisallowを指定します。

```
User-agent: *
Disallow:/*.jpg
Disallow:/*.png
```

すべてのクローラーから特定のディレクトリを拒否

ディレクトリを拒否対象にする場合は、最後に「/」で終わります。そうしないと部分一致で、他のディレクトリやファイルも対象になる可能性があります。また複数の拒否対象を指定する場合は、対象分だけDisallowを指定します。

次のように記述することで、「cgi-bin」「images」「tmp」「private」ディレクトリに対するクローリングを禁止します。

```
User-agent: *
Disallow: /cgi-bin/
Disallow: /images/
Disallow: /tmp/
Disallow: /private/
```

特定のユーザーエージェントを拒否

User-agentに拒否したいユーザーエージェント名を入れると、部分一致で対象を選別します。次のように記述することで、Googlebotを拒否します。

```
User-agent: Googlebot
Disallow: /
```

複数のクローラーを拒否する場合は、空白行で区切りを入れます。

```
User-agent: Googlebot
Disallow: /

User-agent: bingbot
Disallow: /
```

組み合わせて拒否

上記の書式を組み合わせて記述することも可能です。User-Agentごとに違ったルールを作ることも可能です。

```
User-agent: *
Disallow:/*.jpg
Disallow:/*.png
Disallow: /folder1/

User-Agent: Googlebot
Disallow: /folder2/
```

クローリング間隔の指定

robots.txtは、User-agentとDisallow以外にもいくつかの拡張があります。そのなかで特に重要な意味を持つ「Crawl-delay」は覚えておきましょう。Crawl-delayはクローリング間隔を指定するものです。クローラーは、delayに示された間隔でサイトへのアクセスを行います。

一方でdelayの単位については、明確な規定がありません。クローラーによっては、秒と解釈するものや分と解釈するものがあります。傾向としては、秒と解釈することが多いようです。次のように記述することで、クローリング間隔を10秒ごと（クローラーの解釈によっては10分ごと）に制限します。

```
User-agent: *
Crawl-delay: 10
```

2-7-2　サイトの利用規約

クローリングする際は、まずは対象サイトの利用規約を探すべきです。大手のWebサイトであれば、フッターなどに「利用規約」や「サービス規約」といった名前で記載されていることが多いです。

利用規約を見つけたら、明示的な禁止事項と許諾事項を確認します。著作権や利用許可といったタイトルで言及されることが多いようです。

■ Amazon.co.jp 利用規約
URL http://www.amazon.co.jp/gp/help/customer/display.html/ref=footer_cou?ie=UTF8&nodeId=643006

例えば、Amazon.co.jpの「利用許可およびサイトへのアクセス」には、次のように記載されています（2014年7月現在）。

> 本規約の遵守および該当する価格の支払いを条件とし、アマゾンまたはコンテンツ提供者は、アマゾンサービスを限定的、非独占的、非商業的および個人的に利用する権

> 利をお客様に許諾します（譲渡およびサブライセンス不可）。この利用許可には、アマゾンサービスまたはそのコンテンツの転売および商業目的での利用、製品リスト、解説、価格などの収集と利用、アマゾンサービスまたはそのコンテンツの二次的利用、他社のために行うアカウント情報のダウンロードとコピー、データマイニング、ロボットなどのデータ収集・抽出ツールの使用は、一切含まれません。

個人利用については、限定的ながら許諾されています。一方で、商用利用については禁止されています。実際のところ、Amazon.co.jpのように許可の範囲を明確にしているケースは多くありません。その場合は少なくとも明示的に禁止されていることに反しないうえで、一般的なクローラーのモラルに従うのが安全でしょう。

2-7-3 取得したデータの取り扱いと著作権

Webサイトに公開されている情報は、基本的には著作物です。そのため、著作権法に従って取り扱う必要があります。まずクローラーで収集して保存する行為ですが、これは複製にあたります。複製したものを、そのまま公開すると著作権侵害にあたります。一方で、著作権では「私的使用のための複製」は認められています。私的使用とは家庭内など限られた範囲内で利用することです。そのため、クローラーが収集したデータを自分で使うことは問題ありません。一方で、個人の活動であっても収集した著作物をそのままブログやSNSで公開することは、私的使用の範囲としては認められず著作権違反になります。また、会社で業務として行う場合も、私的使用と認められません。

なお、取得したデータを解析・加工して利用する場合については、平成21年改正で導入された著作権法第47条の7で定義されています。この法律によれば、コンピュータなどを用いて情報解析を行うことを目的とする場合には、必要と認められる限度において記録媒体に著作物を複製・翻案することができるとされています。つまり情報解析目的であれば、クローリングも問題ないということです。一方で、解析を行うものを「業として行う者」として、別途政令として定めるとあるので、そちらの解釈が必要となります。

■ 文化庁 平成21年通常国会　著作権法改正等について
URL http://www.bunka.go.jp/chosakuken/21_houkaisei.html

2-7-4 Webサイトのリソース圧迫と業務妨害

クローラー運用時の注意点は、著作権を守るだけではありません。クローラーがWebサイトのリソースを専有すると、他の人がWebサイトを利用できなくなります。

その場合、業務妨害罪に該当する恐れがあります。そのため、Webサイトのリソースを圧迫しない範囲内でクローリングする必要があります。どの範囲であればリソースを圧迫しないかは、Webサイトごとによって異なるので一概には言えません。一般的には、1秒間に1アクセス程度であれば問題ないと見なされていました。

しかし、2010年3月頃に岡崎市立中央図書館事件（Librahack事件）と呼ばれるクローラーに起因する事件が起きて、アクセス間隔についての解釈に変更が必要かもしれないという状況になっています。これは、個人利用で作成したクローラーが原因で図書館の蔵書検索システムにアクセス障害が発生し、クローラーの作成者が逮捕されたという事件です。特記事項として、クローラーは1秒に1アクセス程度に調整されていたものの、Webサイト側の不具合によりサイト障害が発生したということです。業務妨害の強い意図が認められないということで起訴猶予処分となったものの、クローラーを運用するうえで大きな影響を与えています。

同様の事件を起こさないためにも、クローラーを作る際は必ず動作テストを行い、Webサイト側のリソースに影響を与えていないかの確認をすべきです。またサイト側の応答コードを取得し、エラーが続く場合は停止処理を実装すべきです。このあたりについては、「4-4-5 エラーコードに対する処理」（→p.238）で解説します。

2-7-5　クローラーとAPI

著作権をしっかり守り業務妨害を起こさない形で運用したとしても、クローラーはトラブルの元になりやすいのも事実です。本書の目的と矛盾しますが、基本的な方針としてはクローラーはできるかぎり作らないのがよいでしょう。

AmazonやTwitter、Facebookをはじめ昨今の大手サイトはAPIを利用することで、目的のデータを取得できるケースが増えています。APIには、細かく利用規約や制限が定められています。これに従うかぎりは、法律的な問題を起こすことはありません。データ収集したい場合は、まずはAPIの活用ができないかを検討してください。クローラーの作成は、APIが提供されていない場合や、APIではどうしても取得できないデータを集める時に限定するのがよいでしょう。

2-8 ブラウザタイプのクローラー

クローラーに関する基本的な知識が得られたので、次はまったく違うタイプのクローラーを作成しましょう。ブラウザタイプのクローラーです。

これまでは、Anemoneを利用してクローラーを作ってきました。Anemoneはフレームワークとして、クローラーを作るうえで必要な機能をほぼ備えています。しかし、フォーム入力による画面遷移・対話処理やJavaScriptの実行などは、サポートしていません。自動巡回に特化しているために対話処理ができず、またブラウザのレンダリングエンジンを搭載していないためにJavaScriptに対応していない故です。

一方で昨今のHTML5やJavaScriptの隆盛で、JavaScriptをサポートしないとデータが収集できないケースが増えてきています。そういった場合は、Anemone以外をベースにクローラーを作成する必要があります。しかし、クローラーというカテゴリで探しても、対話型処理やJavaScriptに対応しているライブラリはありません。目を少し他の分野に向けてみると、ブラウザを操作するライブラリ群があります。テスト自動化の分野です。

テスト自動化とは、テスト支援ツールを利用してソフトウェアテストを自動化することです。従来は、JavaのJUnitを代表とするプログラム内部を直接テストするユニットテストと呼ばれる領域が中心でした。最近では対象とする領域をどんどんと広げており、Webアプリケーションの分野では、Webブラウザ経由でのUI層のテストも対象となっています。その際に利用するのが「Capybara」や「Selenium」といったツールです。画面テストの際には、Webサイトの特定の項目に値を入力し、その結果が想定どおりの値が出ているかを検証します。つまりWebサイトのテスト支援ツールを利用することにより、従来型のクローラーが苦手とした領域についても対応できるようになります。

2-8-1 画面テストツール

画面テスト系のツールとしては、Seleniumが有名です。Seleniumは画面のテストをするために、直接FirefoxやChromeなどのブラウザを起動して操作します。プラグインを追加することによりInternet ExplorerやSafariなど、さまざまなブラウザを扱えます。ブラウザを利用するので、当然のことながらJavaScriptや対話型処理も可能です。

しかし実際のところ、画面のテスト自動化の際はSeleniumのみを利用して実装することは稀です。テスティングフレームワークと呼ばれる「Cucumber」や「RSpec」といったツールと一緒に利用します。またブラウザシミュレータもSelenium以外にもWebKitやPoltergeist、RackTestなどのライブラリを利用します。

WebKitはブラウザのレンダリングエンジンそのものであり、SafariやChrome、

Operaなどさまざまなブラウザに採用されていました。なお、2014年7月現在では、ChromeやOperaはWebKitから派生したBlinkをレンダリングエンジンとして利用しています。Capybaraは、テスティングフレームワークとブラウザシミュレータを抽象的に使うためのツールです。Capybaraを介すことにより、これらのツールをより直感的に利用することができます。

ブラウザタイプのクローラーのメリットとデメリット

CapybaraやSeleniumのような画面テストツールをクローラーとして利用するメリットは何でしょうか？　最大のメリットとしては、ブラウザを利用するために、人間が操作するのと同等のことをプログラミングで実現できることです。原理的には、どのような処理でも記述できます。また、ブラウザを利用するので、JavaScriptへの対応も可能です。

デメリットとしては、対話型の処理のため、特定のページごとに作り込む必要があることです。リンクを元に自動巡回して、特定のデータを取得するといった処理には向いていません。また、ブラウザやレンダリングエンジンを利用するために、CPUやメモリなどのコンピュータリソースを必要とします。

次のセクションで、CapybaraからSeleniumやPoltergeistを利用するクローラーを作成します。

2-8-2　ブラウザタイプのクローラー作成の準備

ブラウザタイプのクローラー作成のために、まず環境設定を行います。必要なモジュールは、RubyのGemとしては「selenium-webdriver」と「Poltergeist」の2つを中心に、それに付随するライブラリです。CapybaraもPoltergeistと一緒に自動的にインストールされます。またPoltergeistは「PhantomJS」というWebKitのエミュレータに依存するために別途インストールします。PhantomJSはGemライブラリではなく、独立したアプリケーションです。

Seleniumのデフォルトの設定では、ブラウザはFirefoxを利用します。Firefoxがインストールされていない環境の場合は、インストールが必要です。それでは、WindowsとMacごとにインストールと設定手順の確認をします。その後で、Capybaraの概念と使い方を学びます。

Windowsへのインストール

Poltergeistをインストールする際に、依存するライブラリは下記の8つです。

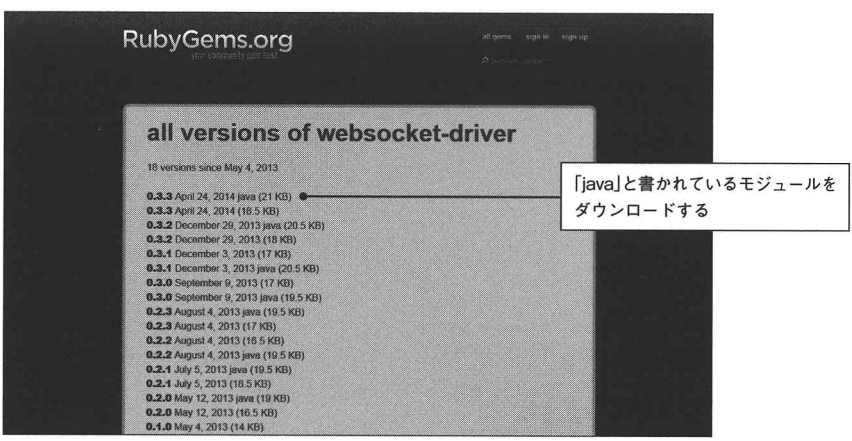
- xpath
- rack
- rack-test
- mime-types
- capybara
- websocket-driver
- multi_json
- cliver

このなかで、websocket-driverについてはビルドツールを必要とします。「2-3 Anemoneのインストール(Windows編)」(→p.74)でDEVELOPMENT KITなどのビルドツールを導入していれば、ソースコードからのビルドが可能です。

導入していない場合は、ビルドずみのGemをダウンロードするのが簡単です。RubyGemsのページから、ビルドずみである「java」と書かれたモジュールをダウンロードしてください。

■ WebSocket-Driver ダウンロードページ
URL **https://rubygems.org/gems/websocket-driver/versions/**

WebSocket-Driverのダウンロード



ダウンロード後に、ダウンロードフォルダーに移動し、gem installでwebsocket-driverをファイル指定でインストールします。

◆ WebSocket-Driverのインストール

```
C:¥work> cd C:¥Users¥YourUserNmae¥Downloads
C:¥Users¥YourUserNmae¥Downloads> gem install websocket-driver
```

その後に、gem installでPoltergeistとselenium-webdriverをインストールしましょう。

◆ Poltergeistのインストール

```
C:¥work> gem install poltergeist
```

◆ selenium-webdriverのインストール

```
C:¥work> gem install selenium-webdriver
```

インストール後に、gem listを入力して、poltergeistとselenium-webdriverが表示されていれば成功です。

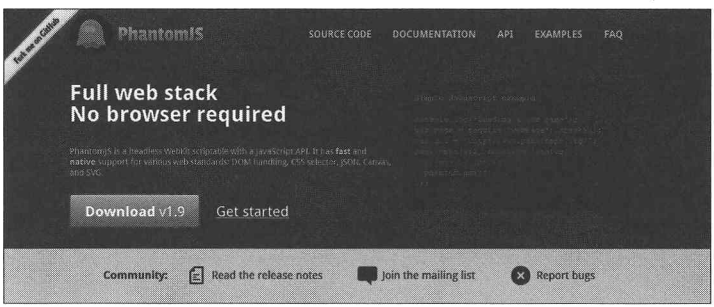
◆ インストールの確認

```
c:¥work> gem list
*** LOCAL GEMS ***

anemone (0.7.2)
bigdecimal (1.2.0)
bropages (0.1.0)
capybara (2.3.0)

~中略~

poltergeist (1.5.1)
psych (2.0.0)

~中略~

selenium-webdriver (2.42.0)
smart_colored (1.1.1)
test-unit (2.0.0.0)
websocket (1.0.7)
websocket-driver (0.3.3 java)
xpath (2.0.0)
```

PhantomJSについても、Windowsのビルドずみモジュールをダウンロードしてインストールします。PhamtomJSのビルドずみモジュールはzipファイルで提供されているため、解凍後に任意のフォルダーに移動させます。

■ PhantomJSのダウンロード
URL http://phantomjs.org/

PhantomJSのダウンロードサイト



その後に、Windowsの環境変数の設定を行います。ユーザー環境変数のPATHの末尾に、PhantomJSへのパスを追加します。

設定後にcmd.exe（コマンドプロンプト）を起動し、phantomjsと入力します。下記の図のようにphantomjsのプロンプトが始まれば、インストール成功です。

コマンドプロンプトの確認

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:¥work>phantomjs
phantomjs>
```

なお、PhantomJSを終了させる場合は、phantom.exit()と入力します。

Macへのインストール

Windowsと同様に、Gemからselenium-webdriverとPoltergeistをインストールします。インストール後に、gem listでselenium-webdriverとPoltergeistが表示されていることを確認してください。WebSocket-Driverは依存関係で自動でインストールされます。

◆ Poltergeistのインストール

```
$ gem install poltergeist
```

◆ selenium-webdriverのインストール

```
$ gem install selenium-webdriver
```

◆ インストールの確認

```
$ gem list
```

そして、PhantomJSをOSにインストールします。PhantomJSは、brew installで簡単にインストールできます。

◆ PhantomJS

```
$ brew install phantomjs
```

PhantomJSの他のインストール方法については、PoltergeistのGitHubサイトに「Installing PhantomJS」として記載されています。

■ GitHubのpoltergeistページ
URL https://github.com/jonleighton/poltergeist

PhantomJSのインストール後に、ターミナルを開きphantomjsと入力します。下記の図のように「phantomjs」のプロンプトが始まれば、インストール成功です。

Mac OS X ターミナルの確認

```
Last login: Sun Mar 16 21:06:38 on ttys004
Sasaki-no-MacBook-Pro:~ takuro$ phantomjs
phantomjs>
```

Capybara

準備が整ったところで、Capybaraの説明です。

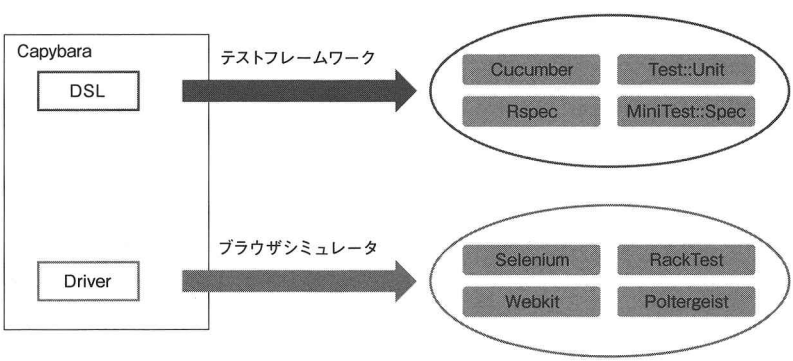
Capybaraは、WebシステムのUI層のテストを補助するためのライブラリです。WebシステムにおけるUI層のテストとは、ブラウザを通しての画面遷移や入力チェック、クロスブラウザによるデザイン・JavaScriptの挙動などの確認を行います。Capybaraはこれらのテストを補助するために、DSL機能とDriver機能の2つを

提供します。

DSL (Domain Specific Language) とはドメイン固有言語のことで、特定の問題に特化したコンピュータ言語です。Capybaraはテストフレームワークを操作する命令を、それぞれのフレームワークに依存しない形で提供します。つまりRubyの書式を拡張し、テスト用のメソッドを追加することにより、より簡潔な記述をできるようにすることです。なおテストフレームワークとは、プログラムのテスト自動化の際に利用するフレームワークです。代表的なものとしては、JavaのJUnitがあります。Rubyにおけるテストフレームワークとしては CucumberやRSpec、Test::Unitがあり、Capybaraはこれらのフレームワークの命令の差異を吸収し、透過的に扱えるようなDSLを提供します。

次にDriverの機能です。UI層のテストの際には、ブラウザであったりレンダリングエンジンやブラウザシミュレータを利用します。Capybaraは、それらをDriverとして扱い、テストフレームワークと同じように透過的に利用できるようにします。ブラウザやレンダリングエンジンを、モジュールとして扱うための仕組みになります。レンダリングエンジンやブラウザシミュレータには、それぞれ機能的な差異やメリット・デメリットがあります。Capybaraを使うことにより、実行する内容に応じて柔軟に切り替えて使うことが可能になります。

Capybara概念図



Capybara DSL

Capybaraは、それぞれのテストフレームワークの機能を抽象化するという観点でDSLを提供しています。テストフレームワークは、フォームに入力してボタンを

クリックして、次の画面の内容を確認するといった機能です。クローラーとして利用する際にも、ほぼ同様の機能が必要になります。

テストフレームワークの機能

機　能	概　要
画面遷移機能	GETメソッドでページを移動する
リンク&ボタンのクリック機能	さまざまな方法でリンクやボタンを指定し、クリックする
フォーム入力機能	フォームのテキストの入力やラジオボタン、チェックボックスの選択、画像の添付を行う
クエリー機能（確認機能）	画面中から特定の文字列やHTMLタグの存在確認を行いTrue or Falseで結果を返す
検索機能	特定の文字列やHTMLタグの検索、もしくはすべての要素から該当のタグの抽出を行う
スコープ機能	検索や操作のスコープを、特定のエレメント下にのみ限定して操作する
スクリプティング機能	JavaScriptを実行する。ただし、使用中のDriverがJavaScriptをサポートしていることが前提
デバッグ機能	デバッグ用に、実行時の状態を表示する

それでは、クローラーを作る際に必要な、Capybara DSLの主なメソッドについて確認してみましょう。

Capybara DSLの主なメソッド

メソッド名	概　要
visit('/path')	引数の値でページ遷移する
click_link('id-of-link')	ID指定でリンクを押す
click_link('Link Text')	リンクのテキスト名で押す
click_button('Save')	ボタン名で名前を押す
click_on('Link Text')	リンクかボタンどちらかをクリックする
click_on('Button Value')	ボタンの値指定で押す
fill_in('First Name', :with => 'John')	フォームテキストを埋める
fill_in('Password', :with => 'Seekrit')	パスワードフォームを埋める
fill_in('Description', :with => 'Really Long Text...')	TextAreaを埋める
choose('A Radio Button')	ラジオボタンを選択する
check('A Checkbox')	チェックボタンを選択する
uncheck('A Checkbox')	チェックボタンの選択を外す
attach_file('Image', '/path/to/image.jpg')	画像を添付する
select('Option', :from => 'Select Box')	セレクトボックスを選択する
find_field('First Name')	フィールド名指定で検索する

Capybara DSLの主なメソッド（続き）

メソッド名	概　要
find_link('Hello')	リンク名指定で検索する
find_button('Send')	ボタン名指定で検索する
find(:xpath, "//table/tr").click	XPath指定で検索する
find("#overlay").find("h1")	入れ子で検索する
all('a').each { |a| a[:href] }	すべての要素から<A>タグを抽出し、hrefを表示する

ここで紹介しているメソッドは、全体のなかのごく一部です。詳しくは、公式のRubyDocを参照してください。

■ Documentation for jnicklas/capybara
URL **http://rubydoc.info/github/jnicklas/capybara/master**

Capybara Driver

Capybaraは、Driverとしていくつかのブラウザシミュレータやブラウザエンジン、またはブラウザそのものを利用できます。どのDriverを利用するかにより、実現できることが異なります。また単純に機能が多いものを選べばよいというわけではありません。例えばSelenium経由でブラウザを起動すると、基本的にはすべてのことができます。しかし、ブラウザシミュレータを利用するのに比べると、処理が遅くなります。クローラーの処理を軽くするためには、できるだけブラウザシミュレータを利用するようにしましょう。

下記の表にCapybaraが利用できる主なDriverの一覧を記載します。今回は、PoltergeistとSeleniumの2つを使い分けることにします。

Capybaraが利用できる主なDriver

Driver名	区　分	JSが実行できるか	外部へのHTTP通信ができるか	ブラウザ起動の有無
RackTest	ブラウザシミュレータ	N	N	N
Capybara-webkit	ブラウザエンジン	Y	Y	N
Poltergeist	ブラウザシミュレータ	Y	Y	N
Selenium	ブラウザ利用	Y	Y	Y

Capybara-webkitとPoltergeistは、機能的にはほぼ同等です。Poltergeistが利用するPhantomJS自体がWebKitのシミュレータとして作られているためです。WebKitを導入するためには、XvfbやQTといったX Windowを仮想的に実行するた

めのライブラリが必要になります。このインストールが非常に難しいために、手軽に利用できる「Poltergeist＋PhantomJS」が人気になっています。

次のセクションでは、Capybaraを利用してログイン画面に対処するケースを考えてみます。

2-8-3　ログインが必要なページの対処

それではさっそくログインが必要なページに対応するクローラーを、Capybaraを利用して実装してみましょう。対象としては、「2-1-4 Ruby製のクローラー」(→p.62)でMechanizeのサンプルで使ったAmazonのアソシエイト(アフィリエイト)ページを再び利用します。このページを対象にログインして、前日の売上を取得する例を紹介します。

Amazonアソシエイトページの構造は、ページ遷移としては「ログインページ」→「ポータルページ」→「レポート全体の表示」と遷移します。それでは、順番に実装していきましょう。

ログイン処理の実装

まずログインページについては、以下のとおりです。メールアドレスとパスワードを入力して、サインインのボタンを押すという構造です。

■ Amazonアソシエイトサイト ログインページ
URL https://affiliate.amazon.co.jp/gp/associates/join/landing/main.html

Amazonアソシエイトサイト ログインページ

まずは、Capybaraの使い方を覚えましょう。利用するモジュールとしては、Capybara本体とcapybara/dsl、そしてDriverモジュールです。今回は、まずselenium-webdriverを利用します。理由としては、ブラウザが立ち上がり、実際の動作が目に見えるのでデバッグしやすいからです。

モジュールの導入

```
require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'
```

次にCapybaraの初期設定を行います。まずrun_serverを「false」にします。Capybaraはデフォルトで自前のRackアプリケーションを起動しようとします。Rackアプリケーションとは、自前のアプリケーションとWebサーバを繋ぐインターフェースです。今回の場合、自前のアプリケーションはクローラーで、WebサーバはAmazonアフィリエイトサイトになります。CapybaraのいくつかのDriverでは、このRackアプリケーションが必要になりますが、SeleniumとPoltergeistは必要ないのでfalseを設定します。

current_driverには「:selenium」を指定します。Capybaraのデフォルトは「:rack_test」であり、そのままではJavaScriptの実行や外部とのHTTP通信などができません。Capybara.app_hostは、対象サイトのURLを指定します。

Capybaraの初期設定

```
Capybara.run_server = false
Capybara.current_driver = :selenium
Capybara.app_host = "https://affiliate.amazon.co.jp/"
```

初期設定が終わったので、プログラム内にCrawlerモジュールとAmazonクラス、loginメソッドを作ります。Amazonクラス内では、Capybara::DSLを有効にしています。これにより、Amazonクラス内ではCapybaraの書式や変数が利用可能になります。

ログイン処理の実装としては、まず「visit('/')」で対象サイトに遷移します。そのうえで、フォームのメールアドレスとパスワードに、fill_inメソッドを利用して値を入力します。フォーム内の項目の指定方法としては、<Input>タグのname属性の値を指定します。その後に、サインインボタンの押下のため、click_buttonメ

ソッドを実装します。ボタンの指定は、<Input>タグのvalue属性の値を指定します。タグの指定方法はいくつかあるので、順次説明します。

なおログインには、Amazonアソシエイトのアカウントが必要です。「username」と「password」に自分のユーザー ID (❶) とパスワード (❷) を設定してください。

Crawlerモジュールの作成

```
module Crawler
  class Amazon
    include Capybara::DSL

    def login
      visit('/')
      fill_in "username",
        :with => 'YOUR_AMAZON_USER_ID' ●―――❶
      fill_in "password",
        :with => 'YOUR_AMAZON_PASSWORD' ●―――❷
      click_button "サインイン"
    end
  end
end
```

Capybaraの初期設定とCrawlerモジュールの作成ができたので、オブジェクトを作成しloginを実行します。上手くいけば、Firefoxが起動し、値が入力されて画面の遷移まで行われるでしょう。

オブジェクトの作成とログイン

```
crawler = Crawler::Amazon.new
crawler.login
```

スクリプトの全体は、以下のようになります。実行に際しては、ご自分のユーザー ID (❶) とパスワード (❷) を設定してください。

ログイン処理　login.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'

Capybara.current_driver = :selenium
```

```
Capybara.app_host = "https://affiliate.amazon.co.jp/"
Capybara.default_wait_time = 5

module Crawler
  class Amazon
    include Capybara::DSL

    def login
      visit('')
      fill_in "username",
        :with => 'YOUR_AMAZON_USER_ID' ❶
      fill_in "password",
        :with => 'YOUR_AMAZON_PASSWORD' ❷
      click_button "サインイン"
    end
  end
end

crawler = Crawler::Amazon.new
crawler.login
```

◆ **login.rbの実行例**

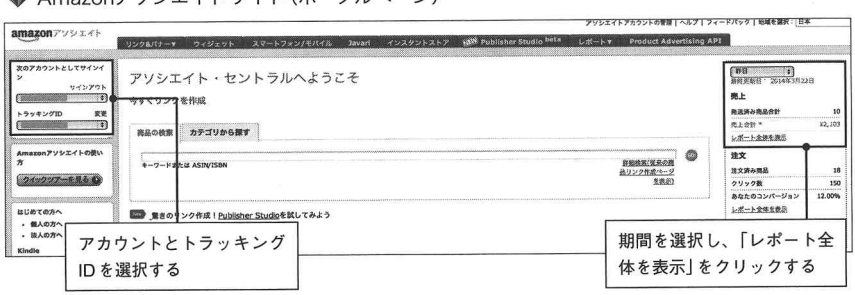
```
$ ruby login.rb
```

実行するとFirefoxが立ち上がり、IDとパスワードを自動で入力してログインします。

認証後のポータルページからレポートページへの遷移

ポータル画面では、画面左のプルダウンでアカウントとトラッキングIDを選択し、画面右のメニューで対象の期間を選んだうえで、売上もしくは注文の「レポート全体を表示」をクリックします。

◆ Amazonアソシエイトサイト（ポータルページ）



上記の動作をするportalメソッドを作りましょう。

portalメソッド

```
def portal
  select('YOUR_AFFILIATE_ID', ❶
    :from => 'idbox_store_id')
  select('昨日', :from => 'preSelectedPeriod')
  first('.line-item-links') ❷
    .click_link("レポート全体を表示")
end
```

プルダウンの処理は、Capybara DSL（→p.121）のselectメソッドを利用します（❶）。selectメソッドの利用方法は次のとおりです。

構文 Capybara DSLのselectメソッド

```
select('プルダウンの表記名',:from => 'Selectボックスのname')
```

それぞれ、ご自分のアカウントとトラッキングIDを設定してください。

次に「レポート全体を表示」リンクの押下です。画面中には、同一の名前のものが2つあります。そのため、単純に、

```
click_link("レポート全体を表示")
```

とすると、曖昧な指定方法という意味の「Ambiguous match, found 2 elements matching link」エラーが出ます。回避策としては、findメソッドを使い、XPathでどのタグかを明示的に指定する方法があります。

構文 Capybara DSLのfindメソッド

```
find(検索オプション)
```

次のように使います。

```
find(:xpath,"//table/tr")
find("#event_title")
```

それ以外には、1つ目に登場するエレメントを指定するといった方法もあります。今回の例では、リンク先の親タグのクラスがどちらもline-item-linksで、対象とす

るリンクが最初に登場します。そこでfirstメソッドを利用して、1つ目のline-item-linksを探し出し、そのなかのclick_linkを指定することとします(❷)。

構文 Capybara DSLのfirstメソッド

```
first(検索オプション)
```

メソッドの指定の仕方からわかるように、Capybaraでは検索や実行コマンドを入れ子で指定することができます。

レポートページからのスクレイピング

レポート画面では選択した条件のもと、集計結果がレポートとして表示されます。

▼Amazonアソシエイトサイト(レポートページ)

売上合計レポート 用語集

2014年3月22日 - 2014年3月22日

	発送済み商品	売上	紹介料
発送済み商品の合計 (Amazon.co.jp)	6	¥36,291	¥1,633
発送済み商品の合計 (マーケットプレイス) ①	4	¥10,444	¥470
発送済み商品合計	**10**	**¥46,735**	**¥2,103**
返品数合計	**0**	**¥0**	**¥0**
返金合計	**0**	**¥0**	**¥0**
紹介料合計	**10**	**¥46,735**	**¥2,103**

レポートページのなかから、売上合計レポートの「発送済み商品」「売上」「紹介料」を抜き出すreportメソッドを作ります。

これらの値は、同一の<Table>タグ・<Tr>タグの、横並びの<Td>タグのなかにあります。そういった場合は、withinメソッドを利用してスコープを定義すると、後々の記述量が少なくなり便利です。

構文 Capybara DSLのwithinメソッド

```
within(検索オプション)
```

次の例では、withinメソッドでXPath指定でスコープを選び(❶)、その下の<Td>タグを抜き出しています(❷)。

reportメソッド

```
def report
  within(:xpath, "//*[@class='totals']") do ❶
    # 発送ずみ商品
    puts "発送済み商品合計:"+all('td')[1].text ❷

    # 売上
    puts "売上:"+all('td')[2].text

    # 紹介料
    puts "紹介料:"+all('td')[3].text
  end
end
```

以下のスクリプトが、先ほどのログイン処理(login.rb)に、ポータルページとレポートページの処理を組み込んだものです。実行の際には、各自のアカウントなどを設定してください。

Capybaraクローラー

capybara-amazon.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'
require 'capybara/poltergeist'

Capybara.current_driver = :selenium
Capybara.app_host = "https://affiliate.amazon.co.jp/"
Capybara.default_wait_time = 5

module Crawler
  class Amazon
    include Capybara::DSL

    def login
      visit('')
      fill_in "username",
        :with => 'AMAZON_USER_ID'
      fill_in "password",
        :with => 'AMAZON_PASSWORD'
      click_button "サインイン"
    end

    def portal
      select('AMAZON_AFFILIATE_ID',
```

```
        :from => 'idbox_store_id')
      select('昨日', :from => 'preSelectedPeriod')
      first('.line-item-links')
      .click_link("レポート全体を表示")
    end

    def report
      within(:xpath, "//*[@class='totals']") do
        puts "発送済み商品:"+all('td')[1].text
        puts "売上:"+all('td')[2].text
        puts "紹介料:"+all('td')[3].text
      end
    end

  end
end

crawler = Crawler::Amazon.new
crawler.login
crawler.portal
crawler.report
```

♥ capybara-amazon.rbの実行例

```
$ ruby capybara-amazon.rb
発送済み商品:3
売上:¥10,055
紹介料:¥396
```

Seleniumから Poltergeistに切り替える

SeleniumでFirefoxなどのブラウザを利用する場合は、その都度ブラウザが起動します。そのため、レンダリングエンジンのみやブラウザシミュレーター内部での処理に比べて、処理時間がかかります。クローラー処理時間の短縮の手段として、スクリプトが完成した段階でPoltergeistなどのブラウザシミュレータに切り替えるのも1つの手です。高速性の他にも、クローラーをLinuxベースのサーバで稼働させる時などにX Windowなど画面システムが不要といったメリットがあります。サーバーサイドでの稼働については、「6-1 サーバサイドで動かす」(→p.362)で詳しく説明します。

Poltergeisへの切り替えは、基本的にはcurrent_driverとjavascript_driverに「:poltergeist」を指定するだけです(❶)。またAmazonアフィリエイトサイト固有

の問題で、ユーザーエージェントの変更が必要です。今回は、「"Mac Safari"」と設定します(❷)。

Poltergeistを利用する際は、「js_errors」というオプションの設定値をfalseにすることを推奨します(❸)。デフォルトはtrueですが、この場合JavaScriptのエラーが起こると処理中断します。実際のところ、Webサイト側の問題でJavaScriptのエラーが起きることは多々あるので、falseでスキップする方がよいでしょう。また、Capybaraには自動的にページロードするまで待つ機能(wait)があります。しかし、Poltergeistを利用する場合は、jQueryやページロードなどの利用時に一部想定どおりの動きをしない場合があります。下記のPoltergeistに切り替えた場合のサンプルでは、jQueryの処理待ちとページロード時のために独自のwaitメソッドを作成しています(❹)。

実行の際には、各自のアカウントなどを設定してください。

Capybaraクローラー(Poltergeist版) capybara-amazon-poltergeist.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'capybara/poltergeist'

Capybara.configure do |config|
  config.run_server = false
  config.current_driver = :poltergeist ●❶
  config.javascript_driver = :poltergeist ●
  config.app_host = "https://affiliate.amazon.co.jp/"
  config.default_wait_time = 5
end

Capybara.register_driver :poltergeist do |app|
  Capybara::Poltergeist::Driver.new(
    app, {:timeout=>120, js_errors: false}) ●❸
end

module Crawler
  class Amazon
    include Capybara::DSL

    def login
      puts "login start"
      page.driver.headers = { "User-Agent" => "Mac Safari" } ●❷
      visit('')
      fill_in "username", :with => 'AMAZON_USER_ID'
```

```
      fill_in "password", :with => 'AMAZON_PASSWORD'
      click_button "サインイン"
    end

    def portal
      wait_for_ajax
      select('AMAZON_AFFILIATE_ID',
        :from => 'idbox_store_id')
      page.execute_script("this.form.submit()")
      wait_for_ajax
      select('昨日', :from => 'preSelectedPeriod')
      page.execute_script("this.form.submit()")
      wait_for_ajax
      page.save_screenshot('screenshot2.png', :full => true)
      first('.line-item-links')
        .click_link("レポート全体を表示")
    end

    def report
      page.save_screenshot(
        'screenshot3.png', :full => true)
      within(:xpath, "//*[@class='totals']") do
        puts "発送済み商品:"+all('td')[1].text
        puts "売上:"+all('td')[2].text
        puts "紹介料:"+all('td')[3].text
      end
    end

    def wait_for_ajax ●———❹
      sleep 2
      Timeout.timeout(Capybara.default_wait_time) do
        active = page.evaluate_script('jQuery.active')
        until active == 0
          sleep 1
          active = page.evaluate_script('jQuery.active')
        end
      end
    end

  end
end

crawler = Crawler::Amazon.new
crawler.login
crawler.portal
crawler.report
```

◆ capybara-amazon-poltergeist.rbの実行例

```
$ ruby capybara-amazon-poltergeist.rb
updating cache[%s], but it was already cached 6

~省略~

updating cache[%s], but it was already cached 9
発送済み商品:3
売上:¥10,055
紹介料:¥396
```

2-8-4　JavaScriptを多用しているページの対処

次にCapybaraを利用して、JavaScriptを多用しているページへの対処を学びましょう。実はCapybaraとSeleniumを利用してブラウザを起動すれば、JavaScriptに対する対処はほとんど必要ありません。目的の処理を入れるタイミングの調整をするだけです。調整方法については、具体的な実例を元に試してみましょう。

遅延ロード

JavaScriptを多用している例として、JavaScriptによる遅延ロードを実装しているサイトを対象とします。遅延ロードとは、画面の初回読み込み時に一度に画像などのデータを読み込むのではなく、遅延させて順次読み込む方式のことです。遅延ロードを利用することにより、部分的であってもサイトの利用者にいち早く画面を見せることが可能となり、読み込みを待たされたという体感を緩和することができます。

利用のテクニックとしては、スクロールなしで表示する必要がある画像のみを先に取得し、スクロール時に都度画像を取得しにいく方法などがあります。またAjaxの通信のように、リクエストに応じて非同期で取得する方法も、遅延ロードの一種と言えます。

ブラックリストチェック

今回の対象は、IPアドレスとドメインのブラックリストチェックをするサイト(BLACLISTALERT.ORG)です。ブラックリストチェックサイトは、さまざまなセキュリティサイトやスパムデータベースに対して、メールサーバーのIPアドレスやドメインがスパムやウィルスの発信源になっていないかの確認を行います。その際の確認は、JavaScriptを利用して順次問い合わせを投げて、応答が返ってくるの

を待ちます。

確認の結果NGとなるということは、メールサーバーのレピュテーション（評価）が下がっているということで、何らかの理由が存在するということになります。原因を至急突き止め、対処が必要となります。

レピュテーションが下がる理由としては、スパムやウィルスメールを大量に配信したという理由の他にも、宛先不明のメールを大量に配信したということや、単位時間あたりに大量のメールを配信しすぎてブロックされたなどがあります。メールサーバの運用では、レピュテーションが低下していないか定期的に確認し、問題があれば早急な対応が必要です。その確認をクローラーが代用することで、運用負荷の低減をはかることができます。今回はそのケースを想定したクローラーを作成します。実際に運用するとなると、確認後にメールにて結果配信するなどが考えられるので、それについては「6-3 収集結果をメールで自動配信する」（→p.384）にて説明します。

BLACKLISTALERT.ORG

BLACLISTALERT.ORGの構造は、単純に調査対象のIPもしくはドメインを入力し、[check]ボタンを押下後にさまざまなサイトの評価を順次表示していくという形です。このサイトでは、評価が正常な場合は「OK」を、問題がある場合（NGの場合）は「Listed!」を表示します。今回の例は、「mail.google.com」を調査対象とし、問題が出た場合は「Alert!」として表示するとします。また、エビデンスとして画面のスクリーンショットを残します。

■ BLACKLISTALERT.ORG
URL http://www.blacklistalert.org/

BLACKLISTALERTサイト

スクリプトについては、「2-8-3 ログインが必要なページの対処」（→p.124）で作成したものとほぼ同じです。検査対象のIPアドレスもしくはドメインを引数とするようにしています（❶）。また問い合わせ先が多く時間がかかるため、タイムアウトの時間を長めに20秒取っています（❷）。NG判定は、withinメソッドで対象を絞っ

たうえで、そのなかの文言で「Listed!」があるかどうかで判定します(❸)。NGの場合は、アラート処理用に作成したメソッドであるalert_notice (❹)を呼び出し、アラート処理をさせます。今回の場合は、コンソールに「Alert!」と表示だけさせています。

また、スクリーンショットの取得は、save_screenshotメソッドを利用しています。save_screenshotは引数としてファイル名を受け取ります。

構文 Capybara DSLのsave_screenshotメソッド

```
save_screenshot(保存ファイル名,(オプション))

オプション
 :full => true
 :selector => '#id'
```

「:full」は全画面を取得するかどうか指定します(デフォルトはfalse)。「:selector」は#idで指定した位置のみを取得します。

ブラックリストチェック　blacklist.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'

Capybara.current_driver = :selenium
Capybara.app_host = "http://www.blacklistalert.org/"
Capybara.default_wait_time = 20 ●―――❷

module Crawler
  class BlackListAlert
    include Capybara::DSL

    def domain_check(target)
      visit('/')
      fill_in "q", :with => target
      click_button "check"

      within(:xpath, "/html/body/center/font/form") do ●―――❸

        alert_notice if page.has_content?('Listed!')
    end
    page.save_screenshot('screenshot.png') ●―――❶
  end
```

```
  def alert_notice ●──────❹
    # アラート処理
    puts "Alert!!"
    end
  end
end

crawler = Crawler::BlackListAlert.new
crawler.domain_check('mail.google.com')
```

◆ blacklist.rbの実行例（OK例）

```
$ ruby blacklist.rb
```

◥◤ 結果のスクリーンショット（OK例）

◆ blacklist.rbの実行例（NG例）

```
$ ruby blacklist.rb
Alert!!
```

◥◤ 結果のスクリーンショット（NG例）

NGのケースを見るには調査対象のIPアドレスもしくはドメインを、すでにブラックリスト判定されているものに変更して試してください。問題のあるIPアドレスの入手は、「DNSBL download list」などを検索条件に、DNSBL（DNS Blacklist）のリストをダウンロードしてください。そのなかに含まれているIPアドレスが、NG対象のリストです。次のURLは、SpamCop.netに報告されたIPアドレスのリストページです。

■ SpamCop.net
URL http://www.spamcop.net/w3m?action=inprogress

Capybaraを使ってJavaScriptによる遅延ロード、Ajaxへの対処例を紹介しました。同時にクローラーの用途が、あまねくデータを収集するだけでなく、日々定期的に特定のサイトを利用して、変化がないか確認するのにも使えるということもおわかりいただけたのではないでしょうか。

次は、Capybaraに足りない機能とそれに対する対処法を学びましょう。

2-8-5　足りない機能を補完する

Capybaraを利用して特定のページに特化したクローラーを作りました。Capybaraは、Webサイトを訪問してページを解析するという、クローラーとしての基本的な機能を備えています。一方で、細かい部分で機能を足したくなることは多々あります。そんな場合は、自分で補完していきましょう。

リンク抽出機能

Capybaraは対話型処理を基本とします。そのために、自動巡回に必要な機能は備えていません。例えば特定のページまでいって、その下の階層に対して、すべて何らかの処理をしたいといったケースは時々出てきます。

そこで、Capybaraにページ取得と自動巡回の機能を足してみましょう。下記の例は、ページ内のリンクの一覧を取得し（❶）、1階層のみ訪問してスクリーンショットを撮るという処理を実装しています（❷）。なお、サンプルのため、リンク先は10個以上収集しないようにしています。

ページ取得と自動巡回　capybara-link.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
```

```
require 'selenium-webdriver'
require 'URI'

Capybara.current_driver = :selenium
Capybara.app_host = "http://www.yahoo.co.jp/"
Capybara.default_wait_time = 20

module Crawler
  class LinkChecker
    include Capybara::DSL

    def initialize()
      visit('')
    end

    def find_links ●―――――❶
      @links = []
      all('a').each do |a|
        u = a[:href]
        next if u.nil? or u.empty?
        @links << u
        # 収集するリンクを10個までに抑える
        break if @links.size >= 10
      end

      @links.uniq!
      @links
    end

    def screenshot(link) ●―――――❷
      puts link
      visit(link)
      page.save_screenshot("screenshot.png")
    end

  end
end

base = URI.parse(Capybara.app_host)
crawler = Crawler::LinkChecker.new
links = crawler.find_links
links.each {|link|
  if URI.parse(link).host == base.host then
    crawler.screenshot(link)
  end
}
```

◆ capybara-link.rbの実行例

```
$ ruby capybara-link.rb
http://www.yahoo.co.jp/r/mbb;_ylt=A7dPhg1u9rBTsVYAlViJBtF7
http://www.yahoo.co.jp/r/mauc;_ylt=A7dPhg1u9rBTsVYAlliJBtF7
http://www.yahoo.co.jp/r/mmy;_ylt=A7dPhg1u9rBTsVYAl1iJBtF7
http://www.yahoo.co.jp/r/mtb;_ylt=A7dPhg1u9rBTsVYAmFiJBtF7
http://www.yahoo.co.jp/r/mshp;_ylt=A7dPhg1u9rBTsVYAmViJBtF7
http://www.yahoo.co.jp/r/mkid;_ylt=A7dPhg1u9rBTsVYAmliJBtF7
http://www.yahoo.co.jp/r/mcfp;_ylt=A7dPhg1u9rBTsVYAm1iJBtF7

~省略~
```

実行すると、スクリプトを格納したディレクトリにスクリーンショット(screenshot.png)が保存されます。スクリーンショットはページの巡回ごとに上書きされていって、最後に訪れたページのものが残ります。

このスクリプトでは、find_linksというメソッドを作り<A>タグからURLの一覧を取得しています(❶)。そして、そのリンク先がターゲットのホスト名と一致した場合のみ、巡回してスクリーンショットを撮るという処理を実行しています(❷)。スクリーンショット取得は、Capybara DSLのsave_screenshotメソッド(→ p.121)を利用しています。

次のページへ移動

2つ目のサンプルもリンク先を取得してページ移動する例です。最近のWebサイトは、1ページの分量を適切に抑えるという目的のもと、検索結果や長大なページは複数ページに分割されることが多いです。対話型のクローラーの場合、ページごとの処理をして次のページに移動するという機能があれば便利です。そこでクローラーにページ送り機能を追加します。

ページ送り機能は、RubyのGemライブラリである「autopagerize-ruby」を利用します。このライブラリがもともと想定している機能としては、すべてのページを取得のうえで1ページにまとめるという用途のようですが、次ページの検索機能やページごとのループなど必要な機能が揃っています。インストールは、gemから簡単に行えます。依存ライブラリも「addressable」と「httpclient」のみです。

■ GitHubのautopagerize-rubyページ
URL https://github.com/uu59/autopagerize-ruby

インストールは次のように行います。

◆ autopagerize-rubyのインストール

```
$ gem install autopagerize
```

インストール後に「Successfully installed autopagerize」と表示されれば成功です。

次のスクリプトは、Amazonの検索結果を1ページごと取得し、スクリーンショットを撮るという処理を行っています。なお、autopagerize-rubyは、Webサイトごとのページ送りの識別方法を定義ファイルから取得するという方法をとっています。そのために前処理として、定義ファイルの取得が必要です。

定義ファイルは、Wedataというサイトから取得可能です。Wedataは、データベース版のWikipediaのようなもので、データを誰でも登録編集可能で、そのデータをAPIを通じて取得することができます。また定義ファイル中にクロール対象のページがないのであれば、自分で定義を記述する必要があります。ここでは定義方法は省略します。

■ Wedata
URL http://wedata.net/

まずは、サイト情報の取得処理を行います。

◆ サイト情報の取得

```
$ curl http://wedata.net/databases/AutoPagerize/items_all.json > siteinfo.json
```

上記の例では、コマンドラインで取得しています。curlなどのダウンロードプログラムがインストールされていない場合は、ブラウザにURLを入力してダウンロードすることも可能です。ダウンロードしたファイルを、適当なファイル名に変更して利用してください。

▌取得した定義ファイル

```
[
{
  "resource_url": "http://wedata.net/items/74934",
  "database_resource_url":
    "http://wedata.net/databases/AutoPagerize",
  "data": {
```

```
    "pageElement": "//center/table",
    "nextLink": "//a[strong[contains(text(),'Old')]]",
    "url": "^http://up¥¥.mugitya¥¥.com/.+th¥¥.htm"
  },
  "created_by": "jak3",
  "name": "麦茶ろだ サムネイル",
  "created_at": "2014-06-30T04:34:09+09:00",
  "updated_at": "2014-06-30T04:37:02+09:00"
},
{
  "resource_url": "http://wedata.net/items/74932",
  "database_resource_url":
    "http://wedata.net/databases/AutoPagerize",
  "data": {
    "exampleUrl": "http://jav.h-era.org/",
    "pageElement": "id('bodyrow')",
    "nextLink": "//a[@class='nextpostslink'
      and contains(text(),¥"≫¥")]",
    "url": "^http://jav.h-era.org/page/"
  },
  "created_by": "Freaking Prime",
  "name": "Jav H-era",
  "created_at": "2014-06-29T04:07:24+09:00",
  "updated_at": "2014-06-29T15:52:05+09:00"
},

~省略~
```

Capybara.app_hostに対象ページとURLを指定します(❶)。ここでは、Amazonの本カテゴリで「Ruby」で検索した結果を設定しています。siteinfoに入手した定義ファイル(siteinfo.json)を指定しています(❷)。定義ファイルは、スクリプトと同じディレクトリに保存してあります。

ページ送り機能の追加　　capybara-autopager.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'selenium-webdriver'
require "multi_json"
require "autopagerize"

Capybara.current_driver = :selenium
Capybara.app_host = ●―――――❶
  "http://www.amazon.co.jp/s/?
```

```
    url=search-alias%3Dstripbooks&field-keywords=ruby"
Capybara.default_wait_time = 20

module Crawler
  class LinkChecker
    include Capybara::DSL

    def initialize()
      visit('')
      url = Capybara.app_host
      siteinfo = MultiJson.load(
        File.read("siteinfo.json")) ❷
      @page = Autopagerize.new(url, siteinfo) ❸
    end

    def get_nextlink
      page_number = 1
      @page.each do |page|
        visit(page.nextlink)
        save_screenshot(
          "screenshot#{page_number}.png")
        page_number += 1
      end
    end

  end
end

crawler = Crawler::LinkChecker.new
crawler.get_nextlink
```

◆ **capybara-autopager.rbの実行例**

```
$ ruby capybara-autopager.rb
DL is deprecated, please use Fiddle
```

実行すると、スクリプトを格納したディレクトリにスクリーンショット(screenshot.png)が保存されます。スクリーンショットはページごとに上書きされていって、最後に訪れたページのものが残ります。

クローラークラスの初期化時に、Autopagerizeオブジェクトを作成しインスタンス変数に格納しています。Autopagerize#eachメソッドで順次に次ページを取得するので(❸)、自動で巡回しています。なお、Autopagerizeは、初期値で10ページまで巡回するように設定されています。必要に応じて任意の順回数に変更してく

ださい。変更方法は、オブジェクト生成時に、maxpageの値を指定します。下記の例は、5ページ巡回するように設定しています。

```
Autopagerize.new(url, siteinfo, :maxpage => 5)
```

robots.txtへの対応

Seleniumを利用する場合、ユーザーエージェントはブラウザ自身になります。そのため、robots.txtが問題になることは少ないです。一方で、PhantomJSなどを利用の際は、念のため確認しておいた方が問題に巻き込まれる確率は小さくなります。サイト閲覧の可否は、Anemoneの作者のライブラリである「robotex」を利用するのがよいでしょう。Anemoneインストール時に同時にインストールされています。

robotexには、2つの機能があります。1つは、ユーザーエージェントの情報を元に、対象のサイトがrobots.txtで拒否されていないかをTrue/Falseで返します。もう1つは、ユーザーエージェントの情報を元に、delay時間を返します。robots.txtに従う場合は、この情報を元にクローラーの処理を記述します。

■ GitHubのchriskite/robotexページ
URL https://github.com/chriskite/robotex

robots.txtに対応する　　capybara-poltergeist-robots.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'capybara/poltergeist'
require 'robotex'

Capybara.configure do |config|
  config.run_server = false
  config.current_driver = :poltergeist
  config.javascript_driver = :poltergeist
  config.app_host = "http://www.yahoo.co.jp/"
  config.default_wait_time = 5
  config.automatic_reload = false
end
Capybara.register_driver :poltergeist do |app|
  Capybara::Poltergeist::Driver.new(app, {
    :timeout=>120, js_errors: false})
end
```

```
module Crawler
  class LinkChecker
    include Capybara::DSL

    def initialize()
      visit('')
      @robots = Robotex.new("Poltergeist")
    end

    def find_links
      @links = []
      all('a').each do |a|
        u = a[:href]
        next if u.nil? or u.empty?
        @links << u
      end
      @links.uniq!
      @links
    end

    def allowed?(link)
      @robots.allowed?(link)
    rescue
      false
    end

    def screenshot(link)
      puts link
      visit(link)
      page.save_screenshot("screenshot.png")
    end

  end
end

crawler = Crawler::LinkChecker.new
links = crawler.find_links
links.each {|link|
  if crawler.allowed?(link) then
    crawler.screenshot(link)
  end
}
```

◆ capybara-poltergeist-robots.rbの実行例

```
$ ruby capybara-poltergeist-robots.rb
http://brazil2014.yahoo.co.jp/
http://office.yahoo.co.jp/?_m=OfficeMode&_a=modeSwitch&copt4=1&nc=
http://search.yahoo.co.jp/
http://image.search.yahoo.co.jp/
http://search.yahoo.co.jp/video
http://dic.yahoo.co.jp/

～省略～
```

このスクリプトでは、クローラーオブジェクト生成時にRobotexオブジェクトを生成しています。そして、ページからリンク一覧を取得し、巡回可能かどうかの判断をしたうえで、スクリーンショットを取得するといった処理です。Robotex内部の処理で、robots.txtを参照し巡回の可否を判断しています。また、確認ずみのサイトについてはキャッシュし、2度目の確認をしないといった機能もあります。

Poltergeistのビルドインストール

「capybara-poltergeist-robots.rb」をWindowsで実行する際に、

```
C:/Ruby200/lib/ruby/2.0.0/rubygems/dependency.rb:296:in `to_specs': Could not find
'websocket-driver' (>= 0.2.0) - did find: [websocket-driver-0.3.3-java] (Gem::LoadError)
```

というエラーが出る場合があります。これはWindows版のPoltergeistの依存性の問題です。この問題を回避方法として、DEVELOPMENT KITを使って、Poltergeistをビルドしてインストールします。DEVELOPMENT KITのインストールはp.76をご参照ください。

Chapter 3

収集したデータを分析する

3-1 収集したデータを分析する

前章は、主にデータの収集を中心に解説してきました。この章では、収集したデータの分析についての解説を行います。解析の主な手段としては、正規表現と構文解析ツールによるデータ抽出があります。どちらも一長一短あるので、それぞれの特徴と基本的な使い方、使い分けを考えていきます。

構文解析ツールに関して言えば、最近のRubyではNokogiriが主流になっています。そこで本書でもNokogiriを取り扱います。Nokogiriのデータの指定の方法としては、XPathとCSSセレクタがあります。両者の基本的な構文と、ブラウザなどを活用して素早く目的のデータを抜き出す方法を学びます。

また、日本語のデータを解析する際には、切っても切れない文字コードの話と、コンピュータに日本語を解析させる自然言語処理の初歩として形態素解析についても簡単に紹介します。

3-1-1　正規表現と構文解析

HTMLもしくはXMLから目的のデータを抜き出す方法として、正規表現と構文解析があります。正規表現は、文章中のパターンに着目し、一定のルールに従うものを抜き出すという手法です。これに対して構文解析は文章中の文法に注目し、特定のタグや要素を抜き出すという形です。実際のところ、パーサー（構文解析器）のタグや要素を探すという実装は、内部的には正規表現を利用していることが多いです。つまり構文解析は、正規表現を使いやすくするために前処理をしてくれるラッパーと考えることも可能です。

正規表現を利用するメリットとしては、ライブラリ非依存で正規表現のみの知識で抽出が可能なことです。ライブラリ非依存だけでなく、ほぼプログラム言語非依存なので、RubyやRubyライブラリにあまり詳しくなくても正規表現がわかれば実装できるという利点はあります。

特定の項目、1点のみを取得する場合は、正規表現で切り出すという方法が手っ取り早いことが多々あります。一方で、複雑な構造のWebサイトからデータを取得する場合は、それに応じて正規表現も複雑になりがちです。複雑な正規表現を組み立てる時間や後々の保守の時間を考えると、継続的に運用するクローラーを作成する際には、正規表現は避けた方がよいのではないでしょうか。

では、構文解析の利用はどうでしょうか。まずデメリットとしては、専用のライブラリのインストールやそのライブラリの使い方の習得、XPathもしくはCSSセレクタを理解するなど、初期の学習コストはかなり高いです。一方で、どんな構造のデータに対しても、データを抜き出す難易度はあまり変わりません。つまり初期学習の壁を超えれば、あとは平坦な道が待っています。特に昨今のWebサイトは構造化されたものが多く、パーサーとの親和性が極めて高いです。後ほど説明しますが、ブラウザ付属の開発ツールで、HTML解析が簡単にできるようになっています。

この2点を考えると、多少の学習コストを払っても構文解析の仕方を身につけるべきです。特に、Rubyの場合はパーサーのデファクトスタンダードがNokogiriになっています。さまざまなツールの構文解析の実装は、Nokogiri依存になっています。そのためNokogiriの使い方を覚えるだけで、さまざまな局面で役に立つことになるでしょう。正規表現と構文解析については、「3-2 HTML解析と正規表現」(→p.149)と、「3-4 RSSの解析」(→p.163)、「3-5 HTMLの解析」(→p.172)で続けて学びます。

3-1-2 日本語の文字コードと日本語処理

Webサイトで利用される日本語の主な文字コードとしては、Shift_JISとECU_JP、UTF-8があります。Rubyは日本発祥のプログラム言語ということで、日本語の文字コードの扱いが他のプログラム言語よりも比較的に容易です。

一方でRubyのGemライブラリのなかには、文字コードの考慮がされていないものも多々あるのは事実です。自分で文字コードを変換するなどの対処が必要です。その際に必要となる最低限の文字コードの知識と、Rubyにおける文字コードの操作方法を、「3-3 文字コードの対処法」(→p.156)にて学びます。また日本語処理については、「MeCab」というライブラリを利用した形態素解析の紹介を「3-6 自然言語を使った日本語の処理」(→p.187)で行います。

3-2 HTML解析と正規表現

正規表現は、文字列のパターンを表現する表記法です。独特のルールで取っつきにくいものの、柔軟性や利便性に富み、短いプログラムでテキストデータをいかようにも扱うことができます。もともとRubyを含めたスクリプト言語は、テキスト処理を得意としています。正規表現をマスターすることによって、選択肢が広がり、

より効率的になります。クローラーで利用する、しないは別にしても、ぜひ身につけておきたい技術の1つです。

Rubyを使いながら正規表現がわからないというのはもったいないので、最低限の正規表現の使い方を記載します。また、HTMLの解析を想定したいくつかのサンプルパターンをあげておきます。

3-2-1　Rubyにおける正規表現の実装

Rubyは、言語レベルで正規表現が実装されています。例えば、文字列を扱うStringクラスのメソッドには、正規表現によるパターンマッチが用意されていて、「=~」もしくはmatchメソッドで正規表現を利用することが可能です。

構文 正規表現によるパターンマッチ

```
"文字列" =~ /正規表現リテラル/
"文字列".match(/正規表現リテラル/)
```

このように特別に正規表現のクラスを呼び出さなくても、Stringクラスから正規表現が利用できます。Rubyは他の言語に比べて、正規表現の利用の敷居が低いのが特徴です。

Rubyの正規表現クラス

Rubyにおける正規表現の実体は、RegExpクラスです。Stringクラスに実装されている正規表現のメソッドも、実体はRegExpクラスを利用しています。また正規表現の結果は、MatchDataクラスを実体化したオブジェクトに格納されます。この結果は、オブジェクトのみならず特殊変数にも格納されます。ちょっと取っつきにくいですが、覚えておくと冗長な記述をしなくてすむようになります。

Rubyの正規表現は、「/」で囲むと正規表現リテラルになります。また「=~」は正規表現にマッチするかの判定を行い、返り値はマッチ開始位置です。これに対して、String#matchはMatchDataオブジェクトを返します。一方で、どちらの場合も特殊変数「$&」にマッチしたテキスト全体を返します。

次の4つの式は、「$&」は同一の結果「Regular」を返します。式自体の結果は、マッチ開始位置である「17」とマッチした結果である「Regular」を返すものに別れます。

▌マッチ結果を返す式

```
puts "My First <strong>Regular</strong>Expression."
  =~ /Regular/       # => 17
puts $&              # => Regular

puts "My First <strong>Regular</strong>Expression"
  .match(/Regular/)  # => Regular
puts $&              # => Regular

puts /Regular/ =~
  "My First <strong>Regular</strong>Expression."  # => 17
puts $&              # => Regular

puts /Regular/.match(
  "My First <strong>Regular</strong>Expression.") # => Regular
puts $&              # => Regular
```

次にMatchDataクラスの使い方です。正規表現のマッチ結果が格納されるMatchDataオブジェクトには、マッチしたテキストのみならず、その前後の文字列も格納されます。また後述するキャプチャ（→p.152）を利用することで、括弧に囲まれた正規表現に一致する部分を順番指定で取得することも可能です。

▌MatchDataオブジェクトの値

```
str = "My First <strong>Regular</strong> Expression."
matchdata = str.match(/<strong>(.*?)<¥/strong>/)

puts matchdata.pre_match
# => My First（マッチしたテキストの手前の文字列）

puts matchdata.post_match
# => Expression.（マッチしたテキストの後ろの文字列）

puts matchdata[0]
# => <strong>Regular</strong>（マッチしたテキスト全体）

puts matchdata[1]
# => Regular（キャプチャの1つ目）
```

MatchDataオブジェクトのかわりに、特殊変数を利用することも可能です。正規表現の特殊変数は、スレッドごとに異なる値を持つスレッドローカルであり、かつメソッド内でのローカルな変数です。

特殊変数の値

```
str = "My First <strong>Regular</strong> Expression."
str.match(/<strong>(.*?)<¥/strong>/)

puts $`  # => My First(マッチしたテキストの手前の文字列)
puts $'  # => Expression.(マッチしたテキストの後ろの文字列)
puts $&  # => <strong>Regular</strong>(マッチしたテキスト全体)
puts $1  # => Regular(キャプチャの1つ目)
```

キャプチャ

Rubyの正規表現では、キャプチャという機能を使うことができます。キャプチャは、丸括弧()に囲まれた条件に一致する値を取得できます。HTMLを解析する際には、キャプチャを使いこなすことが特に重要です。なぜならば、キャプチャを利用することにより、タグで囲まれた文字列を簡単に抜き出すことが可能になるからです。

キャプチャで値を取得する

```
/¥<strong¥>(.+)¥<¥/strong¥>/ =~ "<div><strong>Hello!!</strong></div>"
puts $1  # => Hello!
```

この例では、「=~」の右辺にある文字列リテラルから、タグに囲まれた値(文字列)を取得しています。

なお、「=~」左右の正規表現リテラルと文字列リテラルは、位置を入れ替えても同じように動作します。

またRuby 1.9以降は「名前付きキャプチャ」という機能が利用可能です。これは括弧に名前を設定し、その名前で呼び出すことを可能にする機能です。下の例は、「tag」というキャプチャ名を設定しています。丸括弧内で「?<>」で囲むことで、キャプチャ名として認識されます。

名前付きキャプチャで値を取得する

```
/¥<strong¥>(?<tag>.+)¥<¥/strong¥>/ =~ "<div><strong>Hello!!</strong></div>"
puts tag  # => Hello!
```

名前付きキャプチャを利用することで、複雑な正規表現の可読性を上げることや、途中で順番がずれても修正不要といったメリットがあります。一方で、名前付き

キャプチャを利用する場合は、「=~」が必須で左辺は正規表現リテラルである必要があります。

メタ文字とリテラル

先ほどから何度かリテラルという言葉が出ています。リテラルとは、プログラム中に直接記述された文字列のことです。文字列であれば「文字列リテラル」、正規表現であれば「正規表現リテラル」と呼びます。これに対して、「メタ文字」という言葉があります。メタ文字は、プログラムのなかで特別な意味を持たせた文字のことです。

- ( ) [ ] { } . ? + * | ¥

メタ文字をリテラルとして扱いたい場合は、エスケープ文字「¥」を前に付けます(「¥」は英語環境ではバックスラッシュになります)。「(」をエスケープする場合は、「¥(」です。「¥」の場合は、「¥¥」です。またRegexp.escapeを利用することにより、メタ文字に対してエスケープ文字を付加した文字列を返します。

メタ文字を利用する

```
escaped_str = Regexp.escape("123()[]{}<>abc")
puts escaped_str  # => 123¥(¥)¥[¥]¥{¥}<>abc
```

文字と文字クラス

メタ文字以外にも、意味のある文字が存在します。「¥」と文字列を組み合わせることにより、特定のコードを表現します。

「¥」による文字コードの表現

表　記	意　味
¥t	水平タブ(horizontal tab)(0x09)
¥v	垂直タブ(vertical tab)(0x0B)
¥n	改行(newline)(0x0A)
¥r	復帰(return)(0x0D)
¥b	バックスペース(back space)(0x08)
¥f	改ページ(form feed)(0x0C)
¥a	ベル(bell)(0x07)
¥e	エスケープ文字(escape)(0x1B)
¥nnn	符号化バイト値の8進数表現(nnnの8進数3文字で表現)

「¥」による文字コードの表現（続き）

表　記	意　味
¥xHH	符号化バイト値の16進数表現（HHの16進数2文字で表現）
¥x{7HHHHHHH}	コードポイント値の16進数表現（7HHHHHHHの16進数で表現）
¥cx, ¥C-x	制御文字（xはaからzまでのいずれかの文字）
¥M-x	メタ（x|0x80）
¥M-¥C-x	メタ制御文字
¥uHHHH	ユニコード文字（HHHHの16進数4桁）
¥u{HHHH HHHH ....}	ユニコード文字列（¥uHHHHと同じ）

よく使われる表現については、文字クラスとして省略記法が存在します。例えば、

```
[a-zA-Z0-9]
```

は大文字・小文字の英字と数字全体を表します。省略記号として「¥w」を利用することで、同等の表現と見なされます。

省略記法

表　記	意　味
¥w	単語構成文字 [a-zA-Z0-9_]
¥W	非単語構成文字 [^a-zA-Z0-9_]
¥s	空白文字 [¥t¥r¥n¥f]
¥S	非空白文字 [^ ¥t¥r¥n¥f]
¥d	10進数字 [0-9]
¥D	非10進数字 [^0-9]
¥h	16進数字 [0-9a-fA-F]
¥H	非16進数字 [^0-9a-fA-F]

3-2-2　正規表現のオプション

正規表現には、いくつかのオプションがあります。そのなかで、「i」は大文字と小文字を区別しないというオプションになります。HTMLを解析する場合には、ほぼ必須となるので覚えておきましょう。

<Div>タグの表記として、 、<DIV>、<Div>の3つはHTMLの解釈としてはすべて等価にされます。一方で正規表現は、大文字と小文字は別ものとして扱います。組み合わせごとのパターンを記述するのは、非常な手間です。その際は、「i」オプションを利用します。これは大文字と小文字を区別しないというオプションです。HTML解析の際には、必須と言ってよいオプションの1つです。

「i」オプションを利用する

```
/div/i =~ "Div"
puts $&  # => Div
```

「i」オプションを使うことで、冗長な記載を避けることができました。また「m」オプションを使うことで改行を無視することができます。複数行にまたがるHTMLタグを抜き出す際は、「m」オプションが必要になります。

3-2-3 正規表現のパターン

それでは、HTMLを解析する際に使うであろう正規表現をいくつか例示します。なお、例文中の「html」という変数は、htmlソースが格納された文字列とします。

特定のタグを取得し、中身のみを抜き出して表示する

```
puts html.match(/<title>(.+?)<¥/title>/i)[1]
```

<A>タグ内のhref属性でリンクされているURL一覧を表示する

```
reg = /<a.+?href=['"](.+?)['"]/i
html.scan(reg).each do |item|
  puts item
end
```

id指定でタグ取得し表示する

```
puts html.match(/<[a-zA-Z].+?id=['"]id_name['"].+?>/i)
```

id指定でタグで囲まれた文字列を取得し表示する

```
puts html.match(/<[a-zA-Z].+?id=
  ['"]rhf_recs_error['"].+?>(.+?)<¥/.+?>/im)[1]
```

正規表現の視覚化

ここまで駆け足で正規表現の説明をしてきました。しかし、正規表現をマスターするには、まだまだ遠い道のりです。正規表現をマスターするには、使いながら少しずつテクニックの幅を広げていくしかありません。

一方で、正規表現を習得するのが難しい理由として、デバッグがしにくいという

点があります。つまり正規表現を書いてみて想定の結果が出なかった時に、どこが悪いのかがわからないということです。この壁のために、挫折したという人もいるでしょう。

そんななかで、知っておくと便利なのが正規表現の視覚化です。「正規表現 視覚化」のキーワードで検索するといくつもサイトが出てきますが、入力した正規表現のマッチングパターンを視覚化してくれるサイトがあります。思うとおりに動かない時は、どういった経路でマッチングされているか見てみるとすぐに解決することもあります。オンラインで利用できるので、一度試してみてください。

■ REGEXPER
URL http://www.regexper.com/

3-3 文字コードの対処法

クローラーを作成していると、一度は悩まされるのが文字コードの問題です。日本語を表すのに利用される文字コードは複数あります。代表的なものだけでも、次のように5種類ほどあります。またShift_JISに対するCP932や、それぞれのメーカー・キャリアごとの文字コードの実装などの亜種を含めると、数えきれないほどあります。

- ISO-2022-JP（JIS）
- Shift_JIS
- EUC_JP
- UTF-8
- UTF-16

このなかで、Webサイトでよく利用されるのは、「Shift_JIS」「EUC_JP」「UTF-8」の3種類です。この3つに絞って、どのように扱っていくのかをまとめます。

3-3-1　Rubyにおける文字コードの取り扱い

Ruby 2.0で文字コードを扱うには、主にKconv、NKF、IconvなどのOSにインストールされた文字コード変換ライブラリを利用する方法と、String#encodeメソッドやEncoding::ConverterなどRuby組み込みライブラリを利用する方法とがあります。

KconvはNKFのラッパークラスであり、実装としてはNKFです。このNKFは、nkf (Network Kanji code conversion Filter) をRubyから利用するためのライブラリで、日本語の文字コードの変換を行います。

それに対して、Ruby 1.9.1から M17N (Multilingualization) として多言語化した仕様になりました。その実装としてのEncodingがあります。古いバージョンからの互換性の問題がなければ、基本的にM17Nの実装であるString#encodeメソッドを利用するのがよいです。一方で、KconvやNKFを利用した実装も多いので、簡単に説明しておきます。なおIconvは、Ruby 1.9の時は非推奨だったのですが、Ruby 2.0からは標準添付ライブラリから削除されました。

Kconv と NKF

Kconvは日本語文字コードの変換を行うためのライブラリです。Kconvをrequireすると、Stringクラスに文字コード変換用のメソッドが追加されます。

▼文字コード変換用のメソッド

メソッド	機　能
String#iseuc	文字コードがEUCかどうか判定してTrue/Falseを返す
String#isjis	文字コードがJISかどうか判定してTrue/Falseを返す
String#issjis	文字コードがShift_JISかどうか判定してTrue/Falseを返す
String#isutf8	文字コードがUTF-8かどうか判定してTrue/Falseを返す
String#kconv	変換元の文字コードを変換先の文字コードに変換した文字列を返す
String#toeuc	EUCに変換した文字列を返す
String#tojis	JISに変換した文字列を返す
String#tolocale	ローケルエンコーディング（プログラムで使用中の文字コード）に変換した文字列を返す
String#tosjis	Shift_JISに変換した文字列を返す
String#toutf16	UTF-16に変換した文字列を返す
String#toutf32	UTF-32に変換した文字列を返す
String#toutf8	UTF-8に変換した文字列を返す

名前に「is」が付くメソッドは、文字コードかどうかの判定です。「to」は指定の文字コードに変換します。

▌文字コードの変換

```
require 'kconv'

# 何らかの文字コードの文字列
str = '日本語の文字コード'

puts str.toeuc   # => EUCの文字コードに変換
puts str.tosjis  # => Shift_JISの文字コードに変換
puts str.toutf8  # => UTF-8の文字コードに変換
```

元の文字コードの判別は自動で行われます。文字コードの判別は出現する文字列のバイト配列を読み取って、そのパターンがどの文字コードにあたるかをルールに従って行っています。しかし、そのパターンが、文字コード間で共通されている場合もあります。そのために完全な文字コード判定というのは不可能であり、一定の確率で失敗します。

クローラーの場合は、HTML中にCharSetやEncodingの指定が明示されている、もしくは対象のサイトの文字コードを知っている場合がほとんどなので、プログラムで明示的に指定するのがよいでしょう。明示的に変換する場合は、String#kconvメソッドを利用して、「変換先の文字コード, 変換元の文字コード」と指定します。

▌元の文字コードを指定して変換する

```
require 'kconv'

# 何らかの文字コードの文字列
str = '日本語の文字コード'

puts str.kconv(Kconv::SJIS, Kconv::UTF8)
# => UTF-8からShift_JISに変換

puts str.kconv(Kconv::EUC, Kconv::UTF8)
# => UTF-8からEUCに変換
```

Kconvを利用することにより、比較的簡単に日本語を扱うことができます。しかし、Kconvはあくまで日本語を扱うためのライブラリであり、多言語化対応ではありません。Rubyの国際化に伴い、本格的な多言語化が求められるようになってきました。そういった事情があり、Ruby 1.9.1にて本格的な多言語化対応がされました。

Ruby 1.9.1以降の多言語対応

Ruby 1.9.1以降、多言語化対応となっています。多言語化対応は、M17Nと呼ばれ「Multilingualization」の略です。MからNまで17文字あるために、そのように呼ばれています。

多くのプログラミング言語は、多言語化の際にはUCS Normalizationと呼ばれる方式で対応しています。これは内部的な文字コードをUTFなどの特定の文字コードに統一し、入出力の際にそれぞれ変換する方式です。これに対してRubyは、CSI(Code Set Independent)という方式を採用しています。CSI方式は内部の統一した文字コードは持たず、与えられた文字コードをそのまま扱います。

Rubyの多言語化対応の実装は、Encodingクラスで行われています。String#encodeの実体もEncodingクラスです。Encodingクラスは組み込みライブラリなため、Ruby 1.9以降であれば何も呼び出さなくてもそのまま使えます。

また、どの文字コードで記述しているかを明示することが必要になったので、Ruby 1.9で非アスキー文字列を扱う場合は、Magic Commentで文字コードを指定する必要があります。なおRuby 2.0では、何も指定しない場合のデフォルトの文字コードをUTF-8として扱うようになっています。

encodeメソッドで変換する

```
# coding: UTF-8

# 何らかの文字コードの文字列
str = '日本語の文字コード'

puts str.encode("Shift_JIS")
# => 暗黙的にUFT8から、Shift_JISに変換する

puts str.encode("Shift_JIS","UTF-8")
# => 明示的にUFT8から、Shift_JISに変換する
```

上記のとおりKconvと同等のことが実現できます。互換性の問題がないかぎり、String#encodeメソッドを使うのがよいでしょう。

■ Ruby 2.0.0 リファレンスマニュアル > 多言語化
URL http://docs.ruby-lang.org/ja/2.0.0/doc/spec=2fm17n.html

3-3-2　Nokogiriと文字コード

Rubyで構文解析ツールを利用してHTMLを解析する際は、もっぱらNokogiriを利用します。Nokogiriは内部的にはUTF-8を利用して処理しています。下記のコードは、Nokogiri 1.6.1におけるHTML Parseの実装です。encodingで文字コードを受け取るのですが、指定がない場合はUTF-8として扱います。

▌HTML Parseの実装

```
def self.parse tags, encoding = nil
  doc = HTML::Document.new

  encoding ||= tags.respond_to?(:encoding) ?
    tags.encoding.name : 'UTF-8'
  doc.encoding = encoding

  new(doc, tags)
end
```

このためNokogiriでUTF-8を扱う場合は、明示的に文字コードを指定して利用するか、UTF-8に変換のうえでNokogiriに渡すかのどちらかの対応が必要です。

下記の2つのサンプルスクリプトは、文字コードがShift_JISのWebサイトからHTMLを取得しています。1つ目のスクリプトは、Shift_JISの文字コードでそのまま受け取り、Nokogiri側で文字コードがShift_JISと宣言したうえで渡しています。この場合は、Nokogiri内部でUTF-8に変換されます。

▌そのままの文字コードで渡す　　jis.rb

```
require 'nokogiri'
require 'open-uri'

html = open(
  "http://www.amazon.co.jp/gp/bestsellers/","r:Shift_JIS")
doc = Nokogiri.HTML(html, nil, "Shift_JIS")

puts doc.xpath('//title').text
```

◆ jis.rbの実行例

```
$ ruby jis.rb
Amazon.co.jp ベストセラー: Amazon で最も人気のある商品です。
```

2つ目のスクリプトは、データ取得後にString#encodeメソッドを利用して、Shift_JISからUTF-8に変換しています。その後に、UTF-8の文字列としてNokogiriに渡しています。

文字コードを変換して渡す utf.rb

```
require 'nokogiri'
require 'open-uri'

html = open(
  "http://www.amazon.co.jp/gp/bestsellers/").read.encode("UTF-8","Shift_JIS")
doc = Nokogiri.HTML(html)
puts doc.xpath('//title').text
```

◆ utf.rbの実行例

```
$ ruby utf.rb
Amazon.co.jp ベストセラー: Amazon で最も人気のある商品です。
```

どちらの場合でも同じように動きます。一方で、String#encodeは、渡されたデータ中に不正なバイト列や未定義なバイト列がある場合に、例外を投げます。つまりインプットのデータ次第では、エラーが発生して止まります。

String#encodeのオプションで、その際に適当な文字列に置換することができます。下記のコードのうち、「:invalid」および「:undef」が不正なバイト列と未定義なバイト列を表します。「:replace」はその場合の対処で、置換します。デフォルトの動作はnilでErrorを投げます。

```
html = open(
  "http://www.amazon.co.jp/gp/bestsellers/")
  .read.encode("UTF-8","Shift_JIS",
  :invalid => :replace, :undef => :replace)
```

上記のような理由から、エラーハンドリングをしたうえで文字列をUTF-8に変換して、Nokogiriに文字列を渡すことを推奨します。

3-3-3 Anemoneと文字コード

「2-5-2 クローリング機能の作成」(→p.86)で解説したとおり、Anemoneは受け取ったHTMLを内部的にNokogiriで解析します。しかし、Version 0.7.2時点では文字コードがUTF-8以外の場合は文字化けします。対処としては、Nokogiriで変換した結果

を取得するdocオブジェクトを使わずに、自前でUTF-8に変換してNokogiriに渡すという方法がありますが、同じ処理を二度するので効率的ではないです。

気になるのであれば、Anemoneのソース自体を変更することで対処できます。Anemone内でNokogiriを利用しているのは「page.rb」内のdocメソッドです。ここで引き渡される文字列である@docを、明示的にUTF-8に変換することにより対処できます。

しかし、@docはASCII-8BITとして扱われているために、元のエンコードを探す必要があります。Anemoneには、content_typeというメソッドがあり、HTMLのヘッダーからcontent-typeを取得しています。content_typeから、charsetを抜き出すcharsetメソッドを実装し、String#encodeメソッドに渡します。

Anemoneを修正する

```
#
# Nokogiri document for the HTML body
#
def doc
  return @doc if @doc
  if @body && html?
    if charset == 'utf-8' || charset.nil?
      body = @body
    else
      body = @body.encode(
        "UTF-8", charset, :invalid => :replace,
        :undef => :replace) rescue nil
    end
    @doc = Nokogiri::HTML(body) if body
  end
end

#
# The content-type returned by the
#   HTTP request for this page
#
def content_type
  headers['content-type'].first
end

def charset
  matcher = content_type.match(
    /charset=[¥"]?([a-zA-Z¥_¥-¥d]*)[¥"]?/)
  matcher[1].downcase if matcher
end
```

Anemone本体を書き換えると、利用者側のプログラムで文字コードを意識する必要がなくなります。

文字コードを意識せずにクローリングする anemone-encode.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466282/")
urls.push("http://news.yahoo.co.jp")

Anemone.crawl(urls, :depth_limit => 0) do |anemone|
  anemone.on_every_page do |page|
    puts page.doc.xpath("//title")
  end
end
```

実行すると、文字化けも発生しません。

anemone-encode.rbの実行例

```
$ ruby anemone-encode.rb
<title>Yahoo!ニュース</title>
<title>Amazon.co.jp ベストセラー: ビジネス・経済 の中で最も人気のある商品です</title>
```

Anemoneと同様の問題が、Mechanizeの場合でも発生します。スクレイピングするだけであれば、同様の方法で対処できます。一方で、Mechanizeはフォームに入力してデータを送るといった処理もあるので、文字コードの問題が発生した場合は、Capybaraからブラウザを利用するのがよいでしょう。

3-4 RSSの解析

このセクションでは、RSSの解析方法とHTMLからのRSSフィードへのリンクの取得方法を学びます。RSSは、「2-5-4 RSSを利用する方法」(→p.98)で紹介したとおり、実体はXMLです。XMLはHTMLに比べてシンプルな構造で、解析しやすいです。またRSSは標準規格なのでフォーマットが決まっています。フィード名とその役割を理解すれば、すぐに欲しい情報が取れるでしょう。

3-4-1 名前空間（Namespace）

RSSの説明をする前に、名前空間について簡単に説明をします。

名前空間とは、それぞれの要素名が衝突しないようにするための修飾辞です。名前空間の宣言と要素の表現は、次のような形で行います。またデフォルトの名前空間として、1つだけ名前空間接頭辞を使用しない名前空間宣言も可能です。デフォルトの名前空間に属する要素は、名前空間識別子なしで要素を記述できます。

構文 名前空間の宣言

```
<要素名 xmlns:"デフォルトの名前空間識別子"
        xmlns:名前接頭辞="名前空間識別子">
  <要素名>
    <名前空間識別子:要素名></名前空間識別子:要素名>
  </要素名>
```

例えば、デフォルトの名前空間と「dc」という名前空間があり、それぞれ「item」という要素がある場合は、次のように表現します。名前空間が違うと同じ要素名でも、まったく別の要素と認識されます。そのため、XMLを解析する場合は名前空間の有無は重要になります。

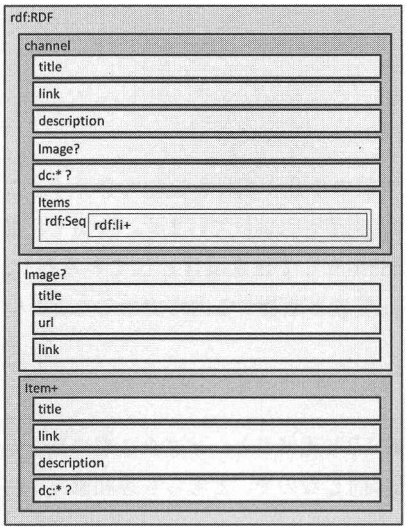
```
<xml xmlns:"デフォルトの名前空間識別子"
     xmlns:dc="名前空間識別子">
  <item />
  <dc:item />
</xml>
```

3-4-2 RSS 1.0

RSS 1.0の構造は、次の図のようになります。図中の「+」はその要素が繰り返すことを表し、「?」はその要素が存在する場合としない場合があります。また「dc:*」は任意の名前で項目を拡張できます。

channel要素内にWebサイトの基本情報、item要素内に記事情報を載せます。例えば、記事を扱うitemは、記事ごとに1つのitemができます。通常、複数の記事があるので、任意の回数だけitemを繰り返します。一般的なRSSでは、新しい順番に5から10ほどのitemを載せることが多いようです。

RSS 1.0の構造



個々の要素の詳細は、下記の表にまとめます。image部分については、実際のところほとんど設定されていないために割愛します。

RSS 1.0のchannel要素

要素名	詳　細
channel	属性のrdf:aboutにRSSのURLを記載する
title	Webサイトのタイトル
link	WebサイトのURL
description	Webサイトの説明
dc:date	最終更新日時。任意項目
dc:language	表示する言語。日本語ならja。任意項目
items	このなかで記事のURLを記載する
rdf:Seq	このなかで記事のURLを記載する
rdf:li	繰り返し項目。属性のrdf:resourceに記事のURLを記載する

RSS 1.0のitem要素

要素名	詳　細
item	記事情報。1つ以上の任意の数
title	Webサイトのタイトル
link	WebサイトのURL

RSS 1.0のitem要素（続き）

要素名	詳　細
description	Webサイトの説明
dc:date	更新日時。任意項目
dc:creator	作成者。任意項目

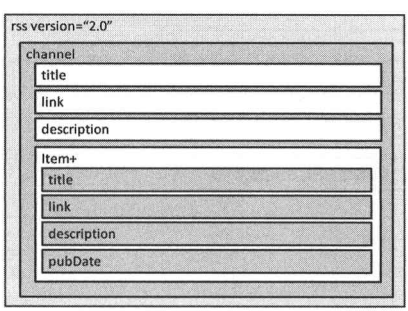
RSS 1.0は、dc（Dublin Core）という名前空間で任意の項目を追加できます。このため、非常に表現力が高く現在でも継続的に利用され続けています。一方で、更新日時など必須であろう項目についてもdc:dateとして任意項目としている点や、channel要素中のrdf:li要素は後に出てくるitem要素と重複する内容であるなど、いくつかの問題点があります。

3-4-3　RSS 2.0

次にRSS 2.0の構造も見てみましょう。RSS 1.0と比べるとシンプルな形になっています。一方でそれぞれの要素の名前や意味は同じなので、どちらも違和感なく利用できるでしょう。

主な変更点としては、channel要素内にサイト情報と記事情報を集約しています。また、RSS 1.0で重複していたchannel要素内のrdf:liとitem要素の意味の重複も解消されています。更新日を表すpubDateという要素が追加されています。それ以外にも、いくつものオプション項目が追加されています。しかし、実際のところ設定されるケースは少ないので、割愛します。また、RSS 1.0のように項目を独自に拡張できないため、表現力の弱さが問題となることがあります。

RSS 2.0の構造



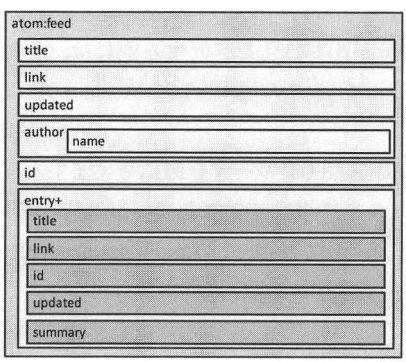
3-4-4 Atom 1.0

Atomは、RSS 2.0にかわるコンテンツ配信技術として規格化されました。Atomには、Atom配信フォーマット（Atom Syndication Format）とAtom出版プロトコル（Atom Publishing Protocol）の2つの仕様があります。Webサイトの更新には、Atom配信フォーマットが利用されます。単にAtomと表記した場合は、このAtom配信フォーマットを指すことが多いです。本書でも、Atom配信フォーマットを指すものとします。

Atomのポリシーの1つとして、誰でも拡張可能という点があります。そのため、Webサイトの更新通知にとどまらず、Gmailにおけるメールの配信などさまざまな用途で利用されています。

構造としては、feed要素中にサイト情報と個々の記事を表すentryが存在します。entryについては、規定ではトップレベル要素とすることも可能です。それ以外にも、コンテンツ要素を追加することも可能です。

Atom 1.0の構造



Atom 1.0のfeed要素

要素名	詳　細
title	Webサイトのタイトル
link	WebサイトのURL
subtitle	Webサイトの説明
updated	更新日時
author	作成者情報。名前やemail、URLなどを含められる

Atom 1.0のentry要素

要素名	詳　細
entry	記事情報。1つ以上の任意の数
title	記事のタイトル
link	記事のURL
summary	記事の説明
id	記事のユニークなID
pubDate	記事の公開日時
updated	記事の更新日時
content	記事の原文

3-4-5　RSS・Atomの解析

RSSもAtomもXMLを元に規格されたフォーマットです。そのため、XMLパーサーを利用すれば簡単に必要な情報を取得できます。NokogiriはHTMLパーサーだけでなくXMLパーサーの機能も兼ね備えてるため、ほぼHTMLと同じような方法でXMLも解析することができます。

解析方法としては、基本的には要素名を指定するだけでよいのですが、RSS 1.0とAtom 1.0については注意が必要です。NokogiriはXMLを解析する際には、名前空間を解釈します。そのため名前空間の指定が必要です。RSS 1.0とAtom 1.0には名前空間があります。

はてなブックマークのHotentryを配信するRSS 1.0フィードを例に考えます。下記は、RSSフィードの宣言部分です。デフォルトの名前空間として、「http://purl.org/rss/1.0/」が指定されています。それ以外にも「rdf」「content」「taxo」「opensearch」「dc」「hatena」などの名前空間が宣言されています。

はてなブックマークのHotentryを配信するRSS 1.0フィード

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns="http://purl.org/rss/1.0/" xmlns:content="http://purl.org/rss/1.0/
modules/content/" xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
xmlns:opensearch="http://a9.com/-/spec/opensearchrss/1.0/" xmlns:dc="http://
purl.org/dc/elements/1.1/" xmlns:hatena="http://www.hatena.ne.jp/info/xmlns#"
xmlns:media="http://search.yahoo.com/mrss">
```

RSS 1.0

それではRSS 1.0を解析してみましょう。NokogiriでXMLを解析する際は、Nokogiri::XMLを利用します（❶）。名前空間については、接頭辞と名前空間識別子をペアにし

たハッシュ配列を作成します(❷)。デフォルトの名前空間についても、接頭辞として仮の名前を与える必要があります。なお、名前空間の宣言は、必要な部分のみ宣言すれば問題ありません。すべての名前空間を宣言する必要はありません。

Nokogiriでは、XPathで検索時に引数として名前空間を渡せます(❸)。名前空間付きのXMLの場合は必須になります(❹)。名前空間を使わない方法としては、CSSセレクタで検索する方法もあります(❺)。

RSS 1.0の解析 rss10.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url = 'http://feeds.feedburner.com/hatena/b/hotentry'
xml = open(url).read

doc = Nokogiri::XML(xml) ●―――❶

namespaces = { ●―――❷
  # デフォルト名前空間
  "rss" => "http://purl.org/rss/1.0/",

  "rdf" => "http://www.w3.org/1999/02/22-rdf-syntax-ns#",

  "content" => "http://purl.org/rss/1.0/modules/content/",

  "dc" => "http://purl.org/dc/elements/1.1/",

  "feedburner" => "http://rssnamespace.org/feedburner/ext/1.0"
}

# channel
channel = doc.xpath('//rss:channel', namespaces) ●―――❸

# Xpathでtitleを検索
puts channel.xpath('rss:title', namespaces) ●―――❹
puts channel.xpath('feedburner:info', namespaces)
lis = channel.xpath('//rdf:li', namespaces)
lis.each {|li|
  puts li.attribute("resource")
}

# CSSセレクタでtitleを検索
puts doc.css('channel title') ●―――❺

items = doc.xpath('//rss:item', namespaces)
items.each {|item|
```

```
  puts item.xpath('rss:title', namespaces).inner_text
  puts item.xpath('content:encoded', namespaces)
  puts item.xpath('dc:date', namespaces).inner_text
}
```

◆ rss10.rbの実行例

```
$ ruby rss10.rb
<title>はてなブックマーク - 人気エントリー</title>
<feedburner:info xmlns:feedburner="http://rssnamespace.org/feedburner/ext/1.0" uri="hatena/b/
hotentry"/>
http://anond.hatelabo.jp/20140713042707
http://togetter.com/li/692199
http://www.huffingtonpost.jp/2014/07/13/mudenchuka-project_n_5581680.html
http://tm2501.hatenablog.com/entry/2014/07/13/130323

~省略~
```

名前空間の概念を理解していれば、XMLの解析はHTMLよりも簡単です。RSSやAtomが提供されている場合は、まずはそちらの利用から検討しましょう。

RSS 2.0

次にRSS 2.0の解析をする方法を見てみましょう。RSS 2.0の場合は、名前空間が定義されていない場合が多いので、HTMLと同じように解析できることが多いのが特徴です。Amazon.co.jpのカテゴリごとのランキングが、RSS 2.0でも配信されています。このRSSから、書名を抜き出してみましょう。

RSS 2.0の解析

rss20.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url =
  'http://www.amazon.co.jp/gp/rss/bestsellers/books/ref=zg_bs_books_rsslink'
xml = open(url).read

doc = Nokogiri::XML(xml)

items = doc.xpath('//rss/channel/item')
items.each { |item|
  #タイトルの表示
```

```
  puts item.xpath('title').text
}
```

◆ rss20.rbの実行例

```
$ ruby rss20.rb
#1: AKB総選挙! 水着サプライズ発表2014【ネット書店 初回入荷限定特典付】
#2: おかげさまで生きる
#3: 歪みの国のアリス(仮)
#4: 嫌われる勇気――自己啓発の源流「アドラー」の教え
#5: うまくいく (恋愛編)
#6: 弓張ノ月-居眠り磐音江戸双紙(46) (双葉文庫)
#7: 海賊とよばれた男(上) (講談社文庫 ひ 43-7)
#8: コミックマーケット 86 カタログ
#9: そのサラダ油が脳と体を壊してる (百年賢脳・健康法)
#10: やはり俺の青春ラブコメはまちがっている。6.5ドラマCD付き限定特装版 (ガガガ文庫)
```

Atom 1.0

最後にAtom 1.0です。Atom 1.0の構造は、RSS 2.0と同様に名前空間の指定が必要になります。取得方法も、構造の違いを除けばRSS 2.0と同じです。はてなブックマークのAtomAPIから、それぞれのエントリーのタイトルを取得する例で見てみましょう。

Atom 1.0の解析

arom10.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url = 'http://b.hatena.ne.jp/dkfj/atomfeed'
xml = open(url).read

doc = Nokogiri::XML(xml)

namespaces = {
  # デフォルト名前空間
  "atom" => "http://purl.org/atom/ns#",
  "dc" => "http://purl.org/dc/elements/1.1/"
}

entries = doc.xpath('//atom:entry', namespaces)
entries.each {|entry|
  puts entry.xpath('atom:title', namespaces).text
}
```

◆ atom10.rbの実行例

```
$ ruby atom10.rb
東京・大手町の地下から温泉採掘　NHKニュース
ギリシャ神話面白すぎクァロタwwwwwwwwwwwwwwwww:哲学ニュースnwk
「クラウドソーシングの衝撃」は、誰にとって衝撃なのか？ - プログラマになりたい
ルビコン河で溺れる韓国：日経ビジネスオンライン
運転し続けたい - NHK　クローズアップ現代
```

3-5 HTMLの解析

このセクションでは、Nokogiriを利用してHTMLを解析する方法を学びます。Nokogiriは、HTMLとXML、SAXとXSLTのパーサーです。Nokogiriの主な機能としては、XPath 1.0による探索、CSS3のセレクタによる探索、XML/HTMLのビルダー機能の3つがあります。

その一方でNokogiriは、10以上のモジュール、70以上のクラスで構成される比較的大きなライブラリです。やみくもにマニュアルを読んでも効率が悪いので、ここであらためてNokogiriのクラス構造を理解したうえで、使い方を学んでいきます。

3-5-1　Nokogiriのクラス構造

今回使おうとしているHTMLの解析機能は、Nokogiri::HTMLクラスにあります。このクラスは、クラスメソッドとして「parse」を持ちます。parseメソッドは、Nokogiri::HTML::Documentクラスのオブジェクトを返します。このクラスは、Nokogiri::XML::Documentを継承し、Nokogiri::XML::DocumentはNokogiri::XML::Nodeを継承しています。つまり、この3つのクラスのドキュメントやソースを読むと、だいたいの動作については理解できるようになります。

Nokogiri::HTML::DocumentとNokogiri::XML::Documentは、Nokogiriドキュメントの生成と格納の役割を果たします。そして、Nokogiri::XML::Nodeは、要素の操作や検索の機能を有しています。XPathやCSSでの検索機能も、ここに実装されています。それでは、次のセクションでそれぞれの動作を詳しく調べてみましょう。

3-5-2　Nokogiri、XPathの使い方

RubyでHTMLを解析する場合は、Nokogiriを利用するのが主流です。Nokogiriはここまでにも使ってきましたが、このセクションではあらためて、徹底的にNokogiriの使い方とXPathの書き方を紹介します。

NokogiriでHTMLファイルを開く

NokogiriでHTMLを解析する場合は、Nokogiri::HTMLもしくはNokogiri::HTML#parseを利用します。

引数としては、最低限、HTMLが文字列型もしくは読み込み可能な形式のオブジェクトとして格納された変数を渡せばよいです。どちらの方法で呼び出しても、内部的にNokogiri::HTML::Document#parseメソッドを実行します。返り値としては、Nokogiri::HTML::Documentが返ってきます。

下記のサンプルスクリプトは、open-uriを利用し「http://www.yahoo.co.jp」をダウンロードし、その結果を直接Nokogiriに渡しています。

ドキュメントを取得する nokogiri-parse.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))
p doc
```

◆ nokogiri-parse.rbの実行例

```
$ ruby nokogiri-parse.rb
#<Nokogiri::HTML::Document:0x809ff29c name="document" children=[#<Nokogiri::XML::DTD:0x809fee14 name="HTML">, #<Nokogiri::XML::Element:0x809fe9f0 name="html" children=[#<Nokogiri::XML::Element:0x809fe784 name="head" children=[#<Nokogiri::XML::Element:0x809fe554 name="meta" attributes=[#<Nokogiri::XML::Attr:0x809fe4c8 name="http-equiv"

~省略~
```

実行すると、「doc」オブジェクトのなかに、Nokogiriで解析された「http://www.yahoo.co.jp」のドキュメントが保存されます。

他の開き方としては、Nokogiri#parseメソッドを利用する方法があります。このメソッドは、受け取った値がHTMLかXMLかを自動判定して、それぞれNokogiri::HTML::Document#parseメソッドもしくはNokogiri::XML::Document#parseメソッドに渡します。判定方法は最初の512文字だけ見てHTMLかどうかの判定をしています。誤判定もしうるので、できるだけ使わない方がよいでしょう。

次の2つの書き方は、どちらも同じ結果が返ってきます。

```
doc = Nokogiri::HTML.parse(open('http://www.yahoo.co.jp'))
doc = Nokogiri.parse(open('http://www.yahoo.co.jp'))
```

Nokogiri::HTML::Document#parseメソッドの引数は、以下のとおりです。

```
parse(string_or_io, url = nil, encoding = nil,
  options = XML::ParseOptions::DEFAULT_HTML)
```

構文 parseメソッド

parse(文字列もしくはIOオブジェクト, URL, 文字コード, オプション)

第1引数にHTMLを格納するオブジェクトを受け取り、第2引数としてはURLを受け取ります。このURLについては、ほぼ意味をなさないので「nil」を渡すと覚えていて問題ないです。第3引数には文字コードを受け取ります。解析対象の文字コードがUTF-8以外の場合、指定しないとかなりの確率で正しく解析できないので注意しましょう。最後の第4引数には、オプションを受け取ります。第1引数以外はオプション項目なので、必要な場合のみ指定します。

オプションはNokogiri::XML::ParseOptionsに定義されています。デフォルトは、XML::ParseOptions::DEFAULT_HTMLでHTMLとして解釈するようになっています。ほとんどの場合は、特に変更する必要もないでしょう。parseメソッドの指定例を以下に示します。

```
doc = Nokogiri::HTML.parse(
  'http://www.amazon.co.jp', nil, "Shift-JIS")
```

Nokogiriで HTMLを解析する

生成したNokogiri::HTML::Documentから、特定のタグを抽出するにはNokogiri::XML::Nodeの検索系メソッドを利用します。Nokogiri::HTML::Documentは、Nokogiri::XML::Nodeを継承しているため、そのまま利用できます。

特定のタグを抽出する　nokogiri-tag.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))
```

```
title = doc.xpath('/html/head/title')
title = doc.css('title')
objects = doc.xpath('//a')

puts title
puts objects
```

◆ **nokogiri-tag.rbの実行例**

```
$ ruby nokogiri-tag.rb
Yahoo! JAPAN
<a href="r/mht"><nobr>ヘルプ</nobr></a>
<a href="r/c1">ショッピング</a>
<a href="r/c2">ヤフオク!</a>
<a href="r/c5">旅行、ホテル予約</a>

~省略~
```

検索には、主にXPathを使う方法とCSSセレクタを使う方法があります。XPathとCSSセレクタは由来は違うものの、どちらもHTMLやXMLの特定の部分を指し示すための言語構文です。タグ名やid名やclass名、もしくはその組み合わせで目的の部位を指し示します。

3-5-3 中心的な3つのクラス

Nokogiriを理解するうえでは、Nokogiri::XML::Nodeのメソッドと挙動を覚えるのが大事です。また、Nokogiri::HTML::DocumentとNokogiri::XML::Documentの継承関係のように、HTMLモジュールはXMLモジュールを継承しているものが多数あります。

- Nokogiri::HTML::Document
  - └Nokogiri::XML::Document
    - └Nokogiri::XML::Node

Nokogiriを素早く身につけるには、Nokogiri::XML::Nodeの挙動を徹底的に理解するのが早道です。そして、Nokogiri::XML::DocumentとNokogiri::HTML::Documentの使い方も理解しましょう。この3つのクラスを抑えておくと、Nokogiriができることがだいたい把握できます。あとは、NodeとNodeSetの関係を理解すればNokogiriを一通り使いこなせるはずです。

Node と NodeSet と Element

NodeとNodeSetは、共通のメソッドが多く挙動が似ているために、違いがわかりづらいです。両者の違いがわかっていないと、検索結果から値を取り出そうとする時にメソッドがないなどのエラーが出て混乱することがあると思います。しかし、NodeとNodeSetの関係がわかれば、何ができて何ができないかたちどころに理解できます。一言でいうとNodeSetは、Nodeを格納したリスト型配列です。

XPathやCSSの検索結果の多くは、Nokogiri::XML::NodeSetを返します。例えば、NodeSetのメソッドの1つであるNodeSet#inner_textは、リスト内のすべてのNodeのinner_textを返します。textは、inner_textのエイリアスです。HTMLの<Title>タグのようにドキュメント中に1つしかないタグの場合は、下記のようにすべて同じ結果を返します。

<Title>タグの取得　　nokogiri-title.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

nodesets = doc.xpath('//title')
puts nodesets.text              # => Yahoo! JAPAN
puts nodesets.inner_text        # => Yahoo! JAPAN
puts nodesets.first.inner_text  # => Yahoo! JAPAN

nodesets.each{|nodeset|
  puts nodeset.content()        # => Yahoo! JAPAN
  puts nodeset.text             # => Yahoo! JAPAN
  puts nodeset.inner_text       # => Yahoo! JAPAN
}
```

◆ nokogiri-title.rbの実行例

```
$ ruby nokogiri-title.rb
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
```

<A>タグのように複数の要素がヒットする場合は、結果が違ってきます。

<A>タグの取得 nokogiri-atag.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

nodesets = doc.xpath('//a')
puts nodesets.inner_text

nodesets.each{|node|
  puts node.inner_text  # => Yahoo! JAPAN
}
```

◆ nokogiri-atag.rbの実行例

```
$ ruby nokogiri-atag.rb

ヘルプショッピングヤフオク!旅行、ホテル予約ニュース天気スポーツファイナンステレビGyaO!Y!モバゲー地域地図路線食べログ求人、アルバイト不動産自動車掲示板ブログビューティー出会い電子書籍映画ゲーム占い>>サービス一覧関東甲信や東北 大雨に警戒諫早制裁金発生 農相「残念」ローソン 医薬品拡充に本腰歯削る機器使い回し 改善に壁日本のロボット産業 未来は?マー君完封目前で逃すも10勝優勝候補のブラジルに死角は?釈由美子 交際報道にうんざり最近の話題記事一覧巨大なアーチ雲ログイン新規取得メールメールアドレスを取得カレンダーカレンダーを活用ポイントを確認ログイン履歴を確認会社概要投資家情報社会的責任企業行動憲章広告掲載について採用情報利用規約セキュリティーの考え方プライバシーポリシー免責事項
ヘルプ
ショッピング
ヤフオク!
旅行、ホテル予約
ニュース
天気

~省略~
```

Nokogiri::HTML::Document

Nokogiri::HTML::Documentクラスで、HTMLの解析時に利用するDocument#parseメソッドを実装しています。それ以外にも、解析したHTMLの文字コードを抽出するDocument#encodingや、HTMLからタイトルを抜き出すDocument#titleなどのメソッドがあります。また、DocumentはNodeを継承しているので、Documentオブジェクトに対してNodeのメソッドが利用可能です。

Nokogiri::HTML::Documentクラスの利用

nokogiri-document.rb

```
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

puts doc.class     # => Nokogiri::HTML::Document
puts doc.title     # => Yahoo! JAPAN
puts doc.encoding  # => utf-8

# Nodeのメソッドも利用可能
puts doc.xpath('//title')
```

◆ nokogiri-document.rb実行例

```
$ ruby nokogiri-document.rb
Nokogiri::HTML::Document
Yahoo! JAPAN
utf-8
<title>Yahoo! JAPAN</title>
```

NodeとNodeSetのメソッド

NodeとNodeSetには、さまざまな検索方法があります。検索に関しては、ほぼ同じメソッドが利用可能です。NodeSetはNodeを継承していませんが、検索系のメソッドについては、Nodeと同様の名前を定義して内部的にNodeを呼び出して処理していることが多いです。下記の例は、すべて同じ結果を返します。

さまざまな検索

nokogiri-titles.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

puts doc%'//title'
puts doc/'//title'
puts doc.at('//title')
# => 検索にヒットした最初のノード

puts doc.at_xpath('//title')
# => xpathの検索にヒットした最初のノード
```

```
puts doc.at_css('title')
# => cssの検索にヒットした最初のノード

puts doc.css('title')
# => cssで検索。NodeSet

puts doc.css('title')[0]
# => cssで検索。NodeSetから最初のノード

puts doc.search('title')
# => xpathかcssで検索。NodeSet

puts doc.search('title')[0]
#=> xpathかcssで検索。NodeSetから最初のノード

puts doc.xpath('//title')
# => xpathで検索。NodeSet

puts doc.xpath('//title')[0]
# => xpathで検索。NodeSetから最初のノード

puts doc.xpath('//title').first
# => xpathで検索。NodeSetから最初のノード
```

◆ nokogiri-titles.rbの実行例

```
$ ruby nokogiri-titles.rb
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
```

NodeとNodeSetは、参照に関してもほぼ同じメソッドが使えます(一部NodeSetでは使えないメソッドがあります)。また、同一の結果を返すメソッドの多くはエイリアス(別名定義)として設定されているものです。

さまざまな参照

nokogiri-reference.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

## Nodeの参照
# HTMLタグ含む
puts doc.at('//title').to_html
puts doc.at('//title').to_xhtml
puts doc.at('//title').to_xml
puts doc.at('//title').to_s

# HTMLタグで囲まれた文字列
puts doc.at('//title').text
puts doc.at('//title').inner_html
puts doc.at('//title').inner_text
puts doc.at('//title').to_str

# 属性値の取得
puts doc.at('//a').[]('href')
puts doc.at('//a').attribute('href')
puts doc.at('//a').get_attribute('href')

## NodeSetの参照
# HTMLタグ含む
puts doc.xpath('//title').to_html
puts doc.xpath('//title').to_xhtml
puts doc.xpath('//title').to_xml
puts doc.xpath('//title').to_s

# HTMLタグで囲まれた文字列
puts doc.xpath('//title').text
puts doc.xpath('//title').inner_html
puts doc.xpath('//title').inner_text
```

◆ nokogiri-reference.rbの実行例

```
$ ruby nokogiri-reference.rb
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
Yahoo! JAPAN
```

```
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
r/mht
r/mht
r/mht
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
<title>Yahoo! JAPAN</title>
Yahoo! JAPAN
Yahoo! JAPAN
Yahoo! JAPAN
```

複数の検索結果が出るものは、NodeSetを返します。結果が1つだけのものは、Elementを返します。動作を理解するために、検索結果として返ってきたオブジェクトのクラスを見てみましょう。次の例では、<A>タグの検索をしています。1つ目の「xpath」の検索では、ドキュメント中のすべての<A>タグを取得します。そのため、返されるオブジェクトはNodeSetになります。2つ目の「at」の検索は、検索条件にヒットした最初の1件のみを返します。そのため、クラスはNokogiri::XML::Elementになります。

複数の検索結果を返す処理

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.yahoo.co.jp'))

node_set = doc.xpath('//a')
puts node_set.class  # => Nokogiri::XML::NodeSet

element = doc.at('//a')
puts element.class   # => Nokogiri::XML::Element
```

Nokogiriの各メソッドのエイリアス

初めてNokogiriを利用する場合に混乱しやすい点として、同じ処理をさまざまなパターンで記述できるところにあります。いろいろなサンプルを見ても、多種多様

で初めての人は面食らいます。その一因となっているのが、メソッドのエイリアスの多さです。基本的には、自分がなじんだ名前で使うのがよいでしょう。

◆ Nokogiriの検索・参照系メソッド

メソッド名	機　能	Node	NodeSet
at	CSSかXPathで検索し、ヒットした最初のElementを返す	○	○
at_css	CSSで検索し、ヒットした最初のElementを返す	○	○
at_xpath	XPathで検索し、ヒットした最初のElementを返す	○	○
css	CSSで検索し、NodeSetを返す	○	○
search	CSSかXPathで検索し、NodeSetを返す	○	○
xpath	XPathで検索し、NodeSetを返す	○	○
content	コンテンツ(タグで囲まれた部分)を返す	○	×
inner_html	タグ内のHTMLを返す	○	○
inner_text	contentのエイリアス	○	○
text	contentのエイリアス	○	○
to_html	要素全体を返す	○	○
to_s	要素全体を返す	○	○
to_str	textのエイリアス	○	×
to_xhtml	要素全体を返す	○	○
to_xml	要素全体を返す	○	○

XPathでの要素の指定方法

XPathは、ロケーションパスを利用して検索します。ロケーションパスはrootノード(HTMLの場合は、html)からタグ名で指定します。htmlタグ中の<Head>タグを指定する場合は、「¥/html¥/head」といった形で指定します。

その他にも、ワイルドカードとして指定できる「¥/¥/」やタグ内の属性での指定なども可能です。

XPathによる検索

nokogiri-xpath.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open('http://www.hatena.ne.jp/'))

# タグを順番に追ってタイトルタグを抜き出す
puts doc.xpath("/html/head/title")

# 文書内のすべてのtitleタグを抜き出す
```

```
puts doc.xpath("//title")

# id指定で特定のulの絞り込み後、
# その配下の3つ目のliタグを抜き出す
puts doc.xpath("//ul[@id='servicelist']/li[3]")

# class指定で検索 classがtitleのh2タグ
puts doc.xpath("//h2[@class='title']")
```

◆ **nokogiri-xpath.rbの実行例**

```
$ ruby nokogiri-xpath.rb
<title>はてな</title>
<title>はてな</title>
<li id="s-link-space"><a href="http://space.hatena.ne.jp/?via=201002">
        <img src="//www.hatena.ne.jp/images/favicon/space.png" alt="" class="service-favicon
for-pc"><img src="/images/hatenaportal/portal/2012/s-link-icon-space@2x.png" alt=""
class="service-favicon for-touch"><span class="servicelist-title">

~省略~
```

3-5-4 簡単なXPathの抽出方法

ここまででNokogiriの使い方は、ある程度身についているはずです。次は、XPathを抽出する方法の説明です。XPathは、基本的には上からタグを辿っていくとわかります。

ただし、昨今の複雑なHTMLを辿っていくには、かなりの根気が必要になります。しかし、FirefoxやChromeを使うことにより、選択しているノードのXPathを簡単に抽出できます。

ChromeでXPathを抽出する方法

Chrome付属の「開発/管理」ツールを利用すると、ブラウザで選択している部分のXPathを簡単に抽出できます。次の3つの手順でクリップボードにXPathがコピーされます。

- ①目的のノード（場所）を選択している状態で、右クリックする
- ②メニューのなかから［要素を検証］を選択する
- ③選択されているノード（青くなっている部分）の上で右クリックして、［Copy XPath］を選択する

ChromeでXPathを抽出

上の図では、「Amazon.co.jp」の個別商品ページから、価格を抜き出している例です。コピーされたXPathは、次のようになっています。

- //*[@id="actualPriceValue"]/b

Chromeで抜き出すXPathは、目的のノードに一番近いID属性からの相対パスです。実際のHTMLは、下記のようなものです（インデントなどを多少修正しています）。

抜き出したXPathのHTML

```
<td id="actualPriceContent">
  <span id="actualPriceValue">
```

```
    <b class="priceLarge">￥ 3,024  </b>
  </span>
  <span id="actualPriceExtraMessaging">
    <span></span>
    <b>通常配送無料</b>
    <a href="/gp/help/customer/display.html/ref=mk_sss_dp_1?ie=UTF8&amp;nodeI
d=642982&amp;pop-up=1" target="AmazonHelp" onclick="return amz_js_PopWin('/
gp/help/customer/display.html/ref=mk_sss_dp_1?ie=UTF8&amp;nodeId=642982&amp;p
op-up=1','AmazonHelp','width=550,height=550,resizable=1,scrollbars=1,toolbar=
0,status=0');">詳細</a>
  </span>
</td>
```

目的の情報である価格は<B>タグで囲まれ、その1つ上が<Span>タグということがわかります。この<Span>タグには「actualPriceValue」というID属性が付いています。HTMLにおけるID属性は、文章内で要素を一意に識別するための識別子です。取得したXPathは一意の要素であるactualPriceContent属性を持つ<Span>タグの下の<B>タグということを示しています。

FirefoxでXPathを抽出する方法

Firefoxの場合も、ほぼ同様の操作で取得できます。ただし前提として、Firebugをインストールしている必要があります。Firebugはアドオンなので、Firefoxのメニューの［ツール］→［アドオン］から、［アドオンの入手］の検索窓で「Firebug」を検索すれば簡単にインストールできます。

インストール後は下記の手順でXPathを取得できます。

- ①目的のノード（場所）を選択している状態で右クリックする
- ②メニューのなかから［Firebugで要素を調査］を選択する
- ③選択されているノード（青くなっている部分）の上で右クリックして、［XPathをコピー］を選択する

Firebugで抜き出すXPathは、ルートノード（html）からの絶対パスです。

- /html/body/div[2]/form/table[3]/tbody/tr/td/div/table/tbody/tr/td[2]/span/b

Chromeと挙動が違うので、目的に応じて使い分けるのがよいでしょう。

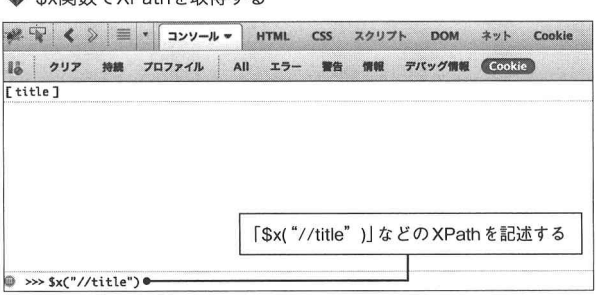
FirefoxでXPathを抽出

Firebug の Console の $x 関数

Firefoxには、XPathの抽出以外の機能があります。FireBugには、いくつもの組み込み関数が用意されています。そのなかの1つに、$x関数というものがあります。これは、任意のXPath要素を取得するというものです。

使い方は、Firebugの「コンソール」タブから一番下の「>>>」と表示されているところで、「$x('//title')」といった感じでXPathを記述します。指定に間違いがなければ、該当のノードが取得されます。自分でXPathを記述する必要がある場合は、想定どおりの動きをしているか、この機能を利用することで検証することができます。

$x関数でXPathを取得する



3-6 自然言語を使った日本語の処理

ここまでで、Nokogiriを使うための方法を一通り解説しました。次は、自然言語処理を使って、日本語のデータ処理を扱う方法です。

3-6-1 形態素解析と特徴語抽出

このセクションでは、形態素解析を行うために代表的な形態素解析ツールの使い方を解説します。形態素解析とは、文法のルールや辞書などを用いて、自然文を最小の単位(形態素)に分解する処理です。また分解した単語の品詞の判別などをすることもあります。形態素解析では最小単位まで分解されるため、複数の単語が繋がるキーワードを抽出できません。単語ごとの重要度というのは取り扱う範疇外のため、重要な単語(特徴語)の抽出もできません。そのあたりをどう解決するかを考えます。

形態素解析エンジン

オープンソースの形態素解析エンジンとしては、2014年時点での主流は「MeCab」です。特徴としては、言語、辞書に依存しない汎用的な設計となっていて、辞書を変えることにより日本語以外の言語にも対応できるようになっています。

■ MeCab
URL https://code.google.com/p/mecab/

インストール型以外の形態素解析エンジンとしては、APIとして利用できる「Yahoo! JAPAN 日本語形態素解析API」があります。リクエストの回数や1リクエストあたりのサイズ制限などがありますが、インストール不要で手軽に試せます。

■ Yahoo! JAPAN 日本語形態素解析API
URL http://developer.yahoo.co.jp/webapi/jlp/ma/v1/parse.html

キーワード抽出

形態素解析を利用すると最小単位まで分解されてしまうため、利用想定によっては実用的でない場合があります。例えば、「プロ野球」の出現頻度を調べたい時に、「プロ」と「野球」で分解されると目的を達成できません。その際は、ある一定のルールで単語を組み合わせる「キーワード抽出」が必要になります。

キーワード抽出の方法は、いろいろあります。ルールベースで名詞の連続は一語

と見なすといった手法や、品詞と品詞の繋がりやすさをコストという概念で考え、単語ごとのコスト表を元に計算する手法などが一例です。これらの手法については、自然言語処理の範疇ですので本書では割愛します。本書では、形態素解析ツールと同様に、キーワード抽出もキーワード抽出APIを利用します。

重要語抽出（特徴語抽出）

形態素およびキーワードの抽出ができたとしても、抜き出した語をそのまま扱うと不都合が生じる場合があります。例えば、一般的な語をそのまま収集し続けると、データ量が膨大になります。後々の解析フェーズで大量の計算リソースが必要になるので、重要なものに絞って保存するというのも1つの方法です。一方で、重要な語の抽出方法をどうするかという問題が出てきます。

重要な語は、文章中の出現回数の多い少ないで決まるものではありません。何度も繰り返し登場しているものの重要でない語や、一度しか登場していないものの重要な位置を占める語もあります。

文章中から重要な語を抜き出す手法の1つに「TF-IDF法」というものがあります。TF-IDF法は、語の出現頻度と出現確率を掛け合わせることにより、文章を特徴づける語を検出する手法です。出現頻度については、文章からカウントすればすぐに計算できます。TF-IDF法については、出現確率のデータをどうするかが肝になります。あらかじめ計算しておいた辞書セットを利用する方法や、簡易的に検索エンジンの検索結果数を利用して出現確率とするといった方法などがあります。

では、実際にYahoo! APIとRubyで出現頻度の計算をしてみましょう。

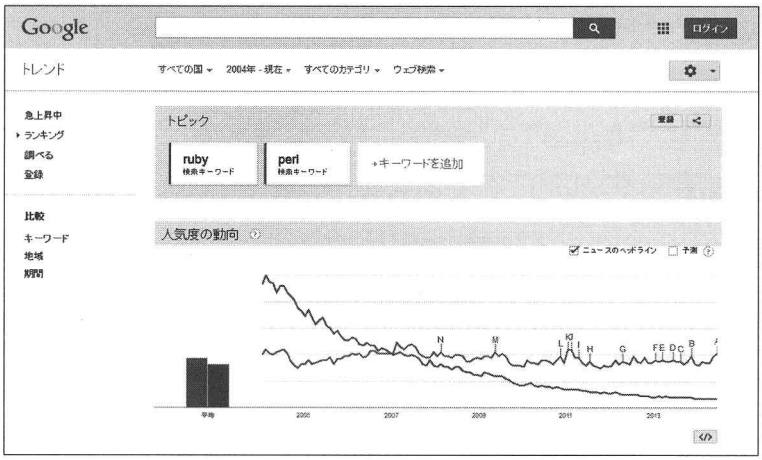
3-6-2　日本語処理

Googleトレンドは、検索されたワードによってトレンドの推移を視覚化しています。対立する2つの商品を比べてみると、どちらが優勢なのか、どのタイミングで何が分岐点になったのかが見えてきて、なかなか面白いサービスです。しかし、残念ながらすべて検索ワードが対象ではなく、ニッチなワードについては表示されません。

■ Google トレンド
URL http://www.google.com/trends/

次の図は「ruby」と「perl」で比べてみたものです。

Googleトレンドでトレンド推移を比較する



検索ワードは、Googleなど検索エンジンのみが持つデータなので一般の人には真似ができません。類似の方法として、日毎のニュースやブログなどを処理して、日時のキーワードの出現数の変化に注目することで、似たようなトレンドの推移を取得することができます。

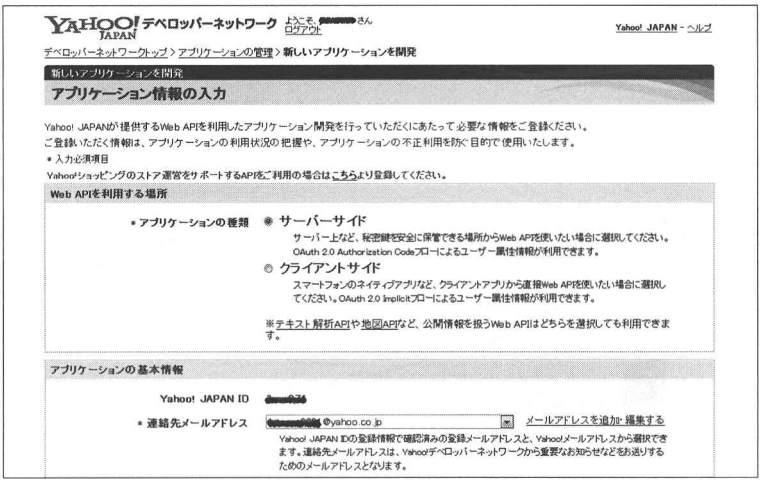
その際の解析元のデータとしては、Twitter全般といったカテゴリを絞らない方法や、Yahoo!の経済ニュースやはてなブックマークのITなど特定のカテゴリに絞るという方法もあります。目的とするデータを元に、データソースとなる対象を選びましょう。今回は、Yahoo! Japanの経済ニュースのカテゴリを対象データソースとします。

Yahoo!APIのアカウント作成

Yahoo!のテキスト解析APIを利用するには、Yahoo! Japan IDとアプリケーションIDが必要になります。アプリケーションIDは、Yahoo! Japan IDを取得のうえでログインした状態で、アプリケーション登録をすることで取得できます。

■ アプリケーション登録
URL https://e.developer.yahoo.co.jp/register

アプリケーション情報を入力する



アプリケーションの登録時に、アプリケーションの種類(サーバーサイド/クライアントサイド)と連絡先メールアドレス、アプリケーション名、サイトURL、アプリケーションの説明を登録します。

アプリケーションの種類は、プログラムが動く場所を示します。サーバ上で動かす場合はサーバサイド、PCなど手元のパソコンで動かす場合はクライアントサイドになります。テキスト解析APIについては、どちらを選んでも利用できます。

アプリケーションの基本情報の登録は、連絡先メールアドレスやアプリケーション名などを登録します。後々わかるように適切な名前を付けましょう。すべての情報を入力してガイドラインに同意すると、登録完了となりアプリケーションIDが発行されます。

形態素解析

Yahoo!の形態素解析APIは、リクエストパラメータおよびレスポンスをXML形式で返します。パラメータについては、それぞれ次の表に記載します。なお、2014年現在に提供されているサービスであるVersion1を前提としています。Yahoo!API側の仕様変更およびサービスの停止の影響を受ける可能性があります。

またAPIの制限として、24時間以内で1つのアプリケーションIDにつき50,000件のリクエストが上限となっています。また、1リクエストの最大サイズを100KBに制限されています。実験的に試すには十分なリクエスト数ですが、本格的なシステム構築を検討する場合は、制限について要件を満たせるかの検討が必要になってきます。

Yahoo! 形態素解析API　リクエストパラメータ

パラメータ	値	説　明
appid	string	アプリケーションID（必須）
sentence	string	解析対象のテキスト（必須）
results	string: ma, uniq	解析結果の種類をコンマで区切って指定（必須）。ma：形態素解析の結果をma_resultに返す。uniq：出現頻度情報をuniq_resultに返す。 無指定の場合は「ma」になる
response	string: surface, reading, pos, baseform, feature	ma_response,uniq_responseのデフォルト設定。wordに返される形態素情報をコンマで区切って指定する。 無指定の場合は「surface,reading,pos」になる
filter	string	ma_filter、uniq_filterのデフォルト設定。解析結果として出力する品詞番号を「|」で区切って指定する。 filterに指定可能な品詞番号:1（形容詞）、2（形容動詞）、3（感動詞）、4（副詞）、5（連体詞）、6（接続詞）、7（接頭辞）、8（接尾辞）、9（名詞）、10（動詞）、11（助詞）、12（助動詞）、13（特殊（句読点、括弧、記号など））
ma_response	string	ma_result内のwordに返される形態素情報をコンマで区切って指定。無指定の場合「response」の指定が用いられる
ma_filter	string	ma_result内に解析結果として出力する品詞番号を「|」で区切って指定する。無指定の場合「filter」の指定が用いられる
uniq_response	string	uniq_result内のwordに返される形態素情報をコンマで区切って指定する。無指定の場合「response」の指定が用いられる
uniq_filter	string	uniq_result内に解析結果として出力する品詞番号を「|」で区切って指定する。無指定の場合「filter」の指定が用いられる
uniq_by_baseform	string	このパラメータがtrueならば、基本形の同一性により、uniq_resultの結果を求める

Yahoo! 形態素解析API　レスポンスフィールド

フィールド	説　明	備　考
ResultSet	解析結果のすべてを含む	共通
ma_result	形態素解析の結果を含む	maのみ
uniq_result	形態素解析の結果から同一形態素の出現数を求めたものを返す	uniqのみ
total_count	形態素の総数を返す	共通
filtered_count	フィルタにマッチした形態素数を返す	uniqのみ
word_list	形態素のリストを返す	共通
word	形態素を返す	共通
surface	形態素の表記を返す	共通
reading	形態素の読みがなを返す	共通・response指定時のみ
pos	形態素の品詞を返す	共通
baseform	形態素の基本形表記を返す。活用のない形態素の場合は省略される	共通・response指定時のみ
feature	形態素の全情報を文字列で返す	共通
count	uniq_result中のword内に現れる、形態素の出現数を返す	uniqのみ

形態素解析APIのレスポンスであるXMLの解析には、XMLパーサーが必要です。Rubyには標準添付ライブラリとして「rexml」というXMLパーサーがあるので、これを利用します。

実行の際には、「APPLICATION_ID」に先ほど取得したアプリケーションIDを設定してください。

形態素解析APIの利用　　basic_morphological_analysis.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'

APPLICATION_ID = 'YAHOO_API_KEY'
BASE_URL = 'http://jlp.yahooapis.jp/MAService/V1/parse'

def request(text)
  app_id = APPLICATION_ID
  params =
    "?appid=#{app_id}&results=uniq&filter=9&uniq_filter=9"
  url = "#{BASE_URL}#{params}"+"&sentence="+URI.encode("#{text}")
  response = open(url)
  doc = REXML::Document.new(response).elements[
    'ResultSet/uniq_result/word_list/']
  doc.elements.each('word') {|element|
    text = element.elements["surface"][0]
    count = element.elements["count"][0]
    p "#{text}=#{count}"
  }
end

text = "隣の客はよく柿食う客だ"
request(text)
```

◆ **basic_morphological_analysis.rbの実行例**

```
$ ruby basic_morphological_analysis.rb
"客=2"
"柿=1"
"隣=1"
```

サンプルスクリプトでは、形態素解析をしたうえで形態素ごとにカウントして結果を返しています。引数の指定はparams変数で行い、解析対象のテキストはURI.encodeで文字列をURLで扱える形に変換しています。

キーワード抽出

キーワード抽出APIも、ほぼ形態素解析APIと同様の形で利用できます。キーワード抽出APIは、レスポンス形式としてXML以外にもJSON形式やPHP Serialize形式が利用できます。

◆ Yahoo! キーワード抽出API

パラメータ	値	説 明
appid	string	アプリケーションID（必須）
sentence	string	解析対象のテキスト（必須）
output	string	レスポンス形式を指定。指定のない場合、XML形式で返す。xml：XML形式で返す。json：JSON形式で返す。JSONP形式で返すには、合わせてcallbackパラメータで関数名を指定する。php：PHP Serialize形式で返す
callback	string	JSONPとして出力する際のコールバック関数名を指定するパラメータ。UTF-8でエンコードした文字列を入力する。関数名として英数字を使用する。output=json&callback=<callback関数名> のように指定

◆ Yahoo! キーワード抽出API　レスポンスフィールド

フィールド	説 明
ResultSet	すべてのキーワード抽出結果
Result	キーワードの結果セット。最大20件返す
Keyphrase	キーワード。重要度の順に並ぶ
Score	キーワードの重要度。100までの整数で高いほど重要度が高くなる

キーワード抽出APIのレスポンスの制約として、Resultは20件までです。

次のサンプルスクリプトが、キーワード抽出APIを使用する例です。実行の際には、「APPLICATION_ID」にp.189で取得したアプリケーションIDを設定してください。

キーワード抽出APIの利用　basic_keyphrase_service.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'

APPLICATION_ID = 'YAHOO_API_KEY'
BASE_URL =
  'http://jlp.yahooapis.jp/KeyphraseService/V1/extract'

def request(text)
  app_id = APPLICATION_ID
  params = "?appid=#{app_id}&output=xml"
  url = "#{BASE_URL}#{params}"+"&sentence="+URI.encode("#{text}")
```

```
  response = open(url)
  doc = REXML::Document.new(response).elements['ResultSet/']
  doc.elements.each('Result') {|element|
    text = element.elements["Keyphrase"][0]
    score = element.elements["Score"][0]
    p "#{text}=#{score}"
  }
end

text = "隣の客はよく柿食う客だ"
request(text)
```

◆ basic_keyphrase_service.rbの実行例

```
$ ruby basic_keyphrase_service.rb
"柿食う客=100"
"隣=40"
```

次のスクリプトは、Yahoo!ニュースから経済ニュースの一覧を取得し、ニュース本文のみを抽出し、キーワード分析をしています。データ保存まではやっていませんが、このようなプログラムで出現キーワードの数を日々記録することで、トレンドの推移を知ることができます。実行の際には、「APPLICATION_ID」にp.189で取得したアプリケーションIDを設定してください。

キーワード分析　　keyphrase_service.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'
require 'nokogiri'

APPLICATION_ID = 'YAHOO_API_KEY'
BASE_URL =
  'http://jlp.yahooapis.jp/KeyphraseService/V1/extract'

$word_list = Hash::new

def request(text)
  app_id = APPLICATION_ID
  params = "?appid=#{app_id}&output=xml"
  url = "#{BASE_URL}#{params}"+"&sentence="+URI.encode("#{text}")
  response = open(url)
  doc = REXML::Document.new(response).elements['ResultSet/']
  doc.elements.each('Result') {|element|
```

```
    text = element.elements["Keyphrase"][0]
    score = element.elements["Score"][0]
    $word_list["#{text}"] =
      $word_list["#{text}"].nil? ? 1 : $word_list["#{text}"]+1
  }
end

def get_urls(page_url)
  urls = Array.new()
  uri = URI.parse(page_url)
  doc = Nokogiri::HTML(open(page_url))
  doc.xpath("//*[@id='main']//ul[@class='list']//a")
    .each do |anchor|

    url = anchor[:href]
    url = uri.merge(url) if not uri =~ /^http/
    urls << url
  end
  return urls
end

def get_headline_text(page_url)
  text = ""
  doc = Nokogiri::HTML(open(page_url))
  if page_url.to_s.match(/dailynews/)
    text = doc.xpath("//*[@id='detailHeadline']").text
  else
    text =
      doc.xpath("//*[@id='main']//p[@class='hbody']").text
  end
  return text.gsub(/¥n/,"")
end

def get(page_url)
  urls = get_urls(page_url)
  urls.each {|url|
    text = get_headline_text(url)
    # p text
    request(text)
  }
end

page_url = 'http://news.yahoo.co.jp/list/?c=economy'
get(page_url)
$word_list.each{ |key,value|
  p "#{key}=#{value}"
}
```

◆ keyphrase_service.rbの実行例

```
$ ruby keyphrase_service.rb
"鍵盤ハーモニカ=1"
"ピアニカ=1"
"プラスチック部品=1"
"楽器店=1"
"小売り希望価格=1"
"朝日新聞=2"
"ヤマハ=1"
"演奏用パイプ=1"
"丸み=1"
"背面=1"
"名前シール=1"
"側面=1"

～省略～
```

特徴語抽出

TF-IDF法は、TF（単語の出現頻度）とIDF法（逆文書頻度）の2つの指標に基づいて計算されます。TFとは、文章中のその単語の出現総数をすべての単語の出現総数で割ったものです。そしてIDFは、Inverse Document Frequencyであり、DF（Document Frequency）の逆数であり、log_2(N/df)で計算しています。ここの「N」は文章総数を指します。これにより「明日」や「私」といった一般的な語に対してフィルターがかかり、重要な単語を浮き上がらせ、個々の単語の重要度を計算できます。つまりTF-IDF法は、単語の出現頻度と単語の重要度を掛け合わせたものになります。

今回利用した、Yahoo!キーワード抽出APIについては、キーワードごとの重要度も計算ずみで結果が返されます。重要度の計算アルゴリズムは不明ですが、TF-IDF法のようなものを利用していると推測されます。自前でTF-IDFを計算する場合は、TFは簡単に測定可能ですが、IDFについては一工夫必要です。あらかじめWikipediaのデータや検索エンジンの検索結果数などから、キーワードごとの出現数を測定し重要度を計算しておく必要があります。

Chapter 4

高度な利用方法

4-1 データの保存方法

クローラーを本格的に運用する場合は、データの保存方法に対する検討が必要です。小規模なクローラーであれば、特に考えなくても問題は起こりません。例えばAnemoneの場合では、デフォルトのデータ保存先はメモリです。そのままで何万件と巡回させると、メモリ溢れのためにエラーが起きるようになります。大規模で運用する場合は、ファイルやRDBMS、NoSQLなどのデータストレージを利用することが必須になります。NoSQLは、一般的にはRDBMSに対して、それ以外のデータベースの総称となります。ここでは分散型データベースやキーバリューストア型のデータベースを指すものとします。

4-1-1　データストレージ

データストレージの選択肢は、たくさんあります。それぞれ一長一短があり、どんなケースにでも最適なストレージといったものはありません。自分のクローラーの目的と、どれくらいのデータを保存するかによって、ストレージを選ぶ必要があります。

一般的には、小中規模であればメモリやファイルを、中大規模であればRDBMSを選ぶのがよいでしょう。クローラーが収集したデータで何らかのサービスを展開するくらいになると、NoSQLなどが選択肢に入ってくるかもしれません。

▼データストレージの比較

保存場所	メリット	デメリット
メモリ	手軽で高速	データの永続化ができない（プログラム起動時のみ利用可能）データ量とメモリ使用量がほぼ比例する
ファイル	手軽にデータの永続化ができる	検索性などで、データ活用時の利便性に欠ける
RDBMS	データ管理が容易で、他のプログラムからも利用しやすい	一定規模以上になると、拡張性が乏しくなる
NoSQL	データ管理が容易で、他のプログラムからも利用しやすい。RDBMSより大規模なデータも扱える場合もある	選択した製品次第だが、RDBMSに比べて何らかの制約がある

4-1-2　ファイルに保存

まずはAnemoneでファイルを使う方法について説明します。

Anemoneでは、ストレージのオプションとして「PStore」が利用可能です。PStoreはRubyの標準添付ライブラリで、データを外部ファイルに保存できます。

データは「Marshal.dmp」という形式のバイナリーで保存されます。

Marshalはシリアライズ(直列化)とほぼ同義で、オブジェクトをバイト列などに変換することを意味します。ここでのMarshal化は、メモリ上のデータをファイルに保存することを指します。

Anomone でPStore の利用

Anemoneでは、「:storage」オプションでAnemone::Storage.PStoreを指定することで使用できます。また指定時に、保存するファイル名も合わせて指定する必要があります(❶)。

ファイルで保存する

anemone-file.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
  :storage => Anemone::Storage.PStore('file.txt'), ❶
  :obey_robots_txt => true,
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("/head/title/text()").to_s if page.doc
  end
end
```

◆ anemone-file.rbの実行例

```
$ ruby anemone-file.rb
http://www.yahoo.co.jp/
```

上記の例は、「file.txt」という名前で「http://www.yahoo.co.jp」から取得した情報を保存するようになっています。ファイルはコマンドを実行したディレクトリに保存されます。また、ファイル名をテキスト形式にしているものの、上述のとおりバイナリー形式なのでほぼ読むことはできません。

それ以外にも、現状のAnemoneでは複数のファイルに分散して保存したり、保

存ずみのファイルを利用してクロールずみのURLをスキップするといったことはできません。プログラム起動時に、同名のファイルが既存の場合は削除し、新たに作成します。

Marshal形式で保存されているために、他のプログラムから利用することも可能です。しかし、検索性なども低く扱いにくい面は否めません。

4-1-3　データベースとの連携

Anemoneの場合、ストレージとしてPStore形式のファイル以外にも、データベース形式やKVS形式が利用できます。

データベース形式は、簡易的なRDBMSである「SQLite3」と分散データベースである「MongoDB」が利用可能です。またKVS形式は「Key-Value Store」のことで、プログラムから利用するハッシュと同様にKey（キー）とValue（値）のペアでデータを管理します。RDBMSほど多様な機能はありませんが、手軽に大量のデータが管理可能です。Anemoneでは、KVS形式としては、「Redis」「Tokyo Cabinet」「Kyoto Cabinet」が利用可能です。

データベース利用の利点

データベースの利用の利点としては、主に2つあります。

1つ目は、大量のデータを扱っても性能の低下が比較的少ないことです。もともとデータベースは、データを収集して管理することに特化しています。そのため、検索や抽出が容易でデータの再利用性が高いのです。また、大量のデータを扱うために、データを格納する方法についても最適化されています。実装方法については、どのようなデータをどれくらいの量で扱うのかという目的によって、RDBMSや分散データベース、KVSとそれぞれ方式は違ってきます。

2つ目は、他のプログラムからのデータ連携が非常に容易であることです。データベースを利用すると、クローラーが収集したデータを他のプログラムやアプリから簡単に検索・表示することが可能です。またクローラーが稼働中でも、リアルタイムに最新のデータを参照することが可能です。

データベースの種類の選択

クローラーを使ってサービスを展開するのであれば、最低限データベースの利用が必要になってきます。そのうえで、どれくらいのデータを扱うかによってデータベースの種類を選択する必要があります。

データベースはCAPの定理と言って、「一貫性(Consistency)」と「可用性(Availability)」と「分断耐性(Partition-tolerance)」の3つの内から2つを選ぶことになります。3つは同時に選べません。一般的には、RDBMSは「一貫性と可用性」を、分散データベースは「一貫性と可用性」もしくは「可用性と分断耐性」のどちらかを選んでいる製品が多いです。

分断耐性が高いと、複数のサーバに1つのデータベースを構築できるなど、拡張性が高くなります。クローラーが扱うデータの特徴として、一貫性は特に必要としません。そのため、RDBMSより分散データベースやKVSなどが好まれる傾向があります。

4-1-4 SQLite3に保存

前述のとおり、AnemoneはRDBも利用できます。公式に対応しているRDBは、「SQLite3」です。

インストール

RubyからSQLite3を利用する場合は、gemによるインストールが必要です。

◆ SQLite3のインストール

```
$ gem install sqlite3 --no-ri --no-rdoc
```

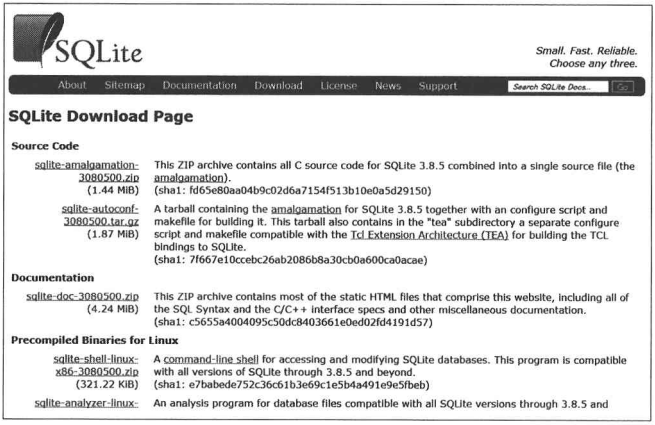
インストール後に「Successfully installed sqlite3-1.3.9」という文言が出れば成功です(バージョンは2014年7月現在です)。

SQLite3のライブラリのインストール

OSからSQLite3のデータベースを操作するには、SQLite3のライブラリが必要です。Windowsの場合は、コンパイルずみのバイナリーモジュールが利用可能です。SQLiteの公式ページから、「sqlite-shell-win32-x86-xxxxxxxxx.zip」をダウンロードします(xxxxxxxxは最新のバージョンを選択してください)。ダウンロード後に解凍し、「sqlite3.exe」を任意の場所に配置します。配置した後に、そこの場所にパスを通すことにより利用できます。

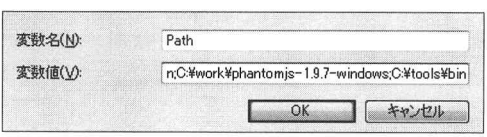
■ SQLiteの公式のダウンロードページ
URL http://sqlite.org/download.html

SQLiteの公式のダウンロードページ



下記の例は、「C:¥tools¥bin」の下に配置した例です。設定後にコマンドプロンプトから、sqlite3 --versionでバージョン名が表示されたら成功です。

Windows7パス登録



SQLite3のバージョン確認（Windows）

```
C:¥work> sqlite3 --version
3.8.5 2014-06-04 14:06:34 b1ed4f2a34ba66c29b130f8d13e9092758019212
```

Macの場合は、デフォルトでインストールされています。ターミナルからインストールずみか確認してください。何らかのバージョンが表示されれば、利用可能な状態です。

SQLite3のバージョン確認（Mac）

```
$ sqlite3 --version
3.7.13 2012-07-17 17:46:21 65035912264e3acbced5a3e16793327f0a2f17bb
```

SQLite3の利用

AnemoneからSQLite3を利用する場合は、「:storage」オプションで、

```
Anemone::Storage::SQLite3()
```

を指定します。ファイル名無指定の場合は、「anemone.db」が作成されます。ファイル名を変更する場合は、

```
Anemone::Storage::SQLite3('filename.db')
```

といった形で指定します。

データベースに保存する

anemone-sqlite3.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
  :storage => Anemone::Storage::SQLite3(),
  :obey_robots_txt => true,
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    p page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-sqlite3.rbの実行例

```
$ ruby anemone-sqlite3.rb
http://www.yahoo.co.jp/
"Yahoo! JAPAN"
```

スクリプトの実行後に、スクリプトを保存したディレクトリに「anemone.db」が作成されていれば成功です。

ところで、「anemone-sqlite3.rb」を実行した後に、もう一度実行してみましょう。二度目は、何も表示されないはずです。

◆ anemone-sqlite3.rbの実行例（二度目）

```
$ ruby anemone-sqlite3.rb
$
```

これはSQLite3のDBを検索し、訪問ずみのページはスキップするためです。ストレージにSQLite3を選択した場合、PStoreのように起動ごとに過去の履歴を消すということはしません。そのため、Anemone::Coreの内部メソッドであるvisit_link?メソッドで「@pages.has_page?(link)」にTrueが返ってくるために、訪れるべきページとしてはFalseが返ってくるようになります。

■ Anemone::Core#visit_linkの実装

```
#
# Returns +true+ if *link*
# has not been visited already,
# and is not excluded by a skip_link pattern...
# and is not excluded by robots.txt...
# and is not deeper than the depth limit
# Returns +false+ otherwise.
#
def visit_link?(link, from_page = nil)
  !@pages.has_page?(link) &&
  !skip_link?(link) &&
  !skip_query_string?(link) &&
  allowed(link) &&
  !too_deep?(from_page)
end
```

クローリングの際に、訪問ずみのページのスキップ機能は重要です。一方で、プログラムから選択ができないので注意が必要です。本来であれば、日付などの条件で再訪問の調整をしたいところですが、SQLite3が作るスキーマには日付の項目がありません。必要であれば、項目を追加してAnemoneを改良するのも1つの手です。デフォルトの状態では、Anemone::Storage::SQLite3クラスが訪問ずみかどうかの判断は、has_keyメソッドでURLのみを判定の条件としています。

■ has_keyメソッドの実装

```
def has_key?(url)
  !!@db.get_first_value(
    'SELECT id FROM anemone_storage WHERE key = ?',
    url.to_s)
end
```

作成したデータベースの確認

SQLite3利用時にAnemoneが作成したデータベースを参照するには、コマンドプロンプトもしくはターミナルからsqlite3コマンドで行います。専用のツールを利用している場合は、そちらをご利用ください。コマンド実行後はプロンプトが「sqlite」に変化します。

◆ 作成したDBの確認

```
$ sqlite3 anemone.db
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

バージョンは各自の環境に合わせて変化します。また、Anemoneが作成するテーブル名は「anemone_storage」です。

◆ テーブル名の確認

```
sqlite> .tables
anemone_storage
```

テーブル構造としては、「id」と「key」、そして実際のデータを格納する「data」の3つです。idは自動付番で、keyがURLになります。

◆ テーブル構造の確認

```
sqlite> .schema anemone_storage
CREATE TABLE anemone_storage (
        id INTEGER PRIMARY KEY ASC,
        key TEXT,
        data BLOB
      );
CREATE INDEX anemone_key_idx on anemone_storage (key);
```

データは、PStoreと同様に「Marshal.dmp」でバイナリー化されて格納されます。バイナリーデータを表示させても意味がないので、idとkeyを確認してみましょう。

◆ データの確認

```
sqlite> select id, key from anemone_storage where key = 'http://www.yahoo.co.jp/';
1|http://www.yahoo.co.jp/
```

なお、SQLite3から抜けるには.exitで抜けます。コマンドがわからなければ、.helpで調べられます。

◆ SQLite3を終了する

```
sqlite> .exit
```

4-1-5　MongoDBに保存

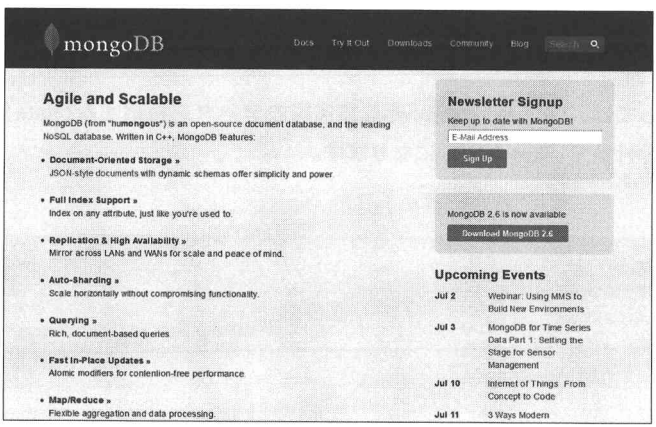
次に分散データベースを利用する例を解説します。分散データベースとしてはMongoDBが利用可能です。

Windowsへのインストール

Windowsへのインストールについては、公式ページでも細かく説明されています。

■ MongoDBの公式ページ
URL http://www.mongodb.org/

MongoDBの公式ページ



基本的な流れとしては、MongoDB用のフォルダーを作成し、ダウンロードしたバイナリーモジュールを配置するという流れです。

- ①「C:¥data」の作成
- ②「C:¥data¥db」の作成
- ③公式ページからダウンロードしてきたMongoDBを解凍

- ④MongoDBを「C:¥data」に配置
- ⑤解凍したMongoDBモジュールのbinフォルダー内のmongod.exeを実行
- ⑥mongodが起動。journalフォルダーなど必要なものが自動的に作成される

公式ページに従ってインストールした場合は、下記のようなフォルダー構成になります。

MongoDB Windowsのフォルダー構成

```
C:¥>tree data
フォルダー パスの一覧
ボリューム シリアル番号は 14EF-A945 です
C:¥DATA
├─db
│  └─journal
└─mongodb-win32-x86_64-2008plus-2.6.0
    └─bin

C:¥>
```

MongoDBのインストールが完了したら、次はRubyからMongoDBを利用するために、Gemのmongoモジュールをインストールします。付随して、bsonもインストールされます。bsonを利用する場合は、bson_extのインストールも勧められます。bson_extをインストールするには、ビルドツールが必要となるのでご注意ください。

◆ **mongoモジュールのインストール**

```
C:¥work> gem install mongo
```

MongoDBを利用するには、binフォルダーの「mongod.exe」を起動します。コマンドラインでmongod.exeの実行、もしくはエクスプローラーからダブルクリックで起動します。コマンドプロンプトが開いた間のみ利用可能です。永続的に利用するには、デーモンをバックエンドで起動する必要があります。MongoDBの永続的な起動についての詳細は、本書では割愛します。

Macへのインストール

HomeBrewを利用の場合は、installコマンドでインストール可能です。インストール終了後に、mongoコマンドでバージョン表示できれば成功です。

◆ **MongoDBのインストール**

```
$ brew install mongodb
```

◆ バージョンの確認

```
$ mongo --version
MongoDB shell version: 2.4.9
```

MongoDBのインストールが完了したら、次はRubyからMongoDBを利用するためのGemのインストールです。Gemのmongoモジュールをインストールします。付随して、bsonもインストールされます。またWindows版と同様に、bson_extのインストールを勧められます。インストールせずとも問題ないですが、必要であればインストールしてください。

◆ mongoモジュールのインストール

```
$ gem install mongo
```

MongoDBを利用するには、MongoDBのデーモンを起動する必要があります。mongodで起動することが可能ですが、下記の方法ではターミナルが開いている時のみ利用可能です。永続的に利用するには、デーモンをバックエンドで起動する必要があります。MongoDBの永続的な起動についての詳細は、本書では割愛します。

◆ MongoDBデーモンの起動

```
$ mongod
all output going to: /Users/username/tools/homebrew/var/log/mongodb/mongo.log
```

MongoDBの利用

AnemoneからMongoDBを利用する場合は、オプションの「:storage」で、

```
Anemone::Storage::MongoDB()
```

を指定します。初期化のオプションは、次の形式です。

```
Anemone::Storage::MongoDB(mongo_db, collection_name)
```

DB名とCollection名を指定します。Collectionは、RDBのテーブルとほぼ同じ意味です。無指定の場合は、DBは「anemone」を利用し、Collection名は「pages」を利用します。

MongoDBに保存する anemone-mongo.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
  :storage => Anemone::Storage::MongoDB(),
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ **anemone-mongo.rbの実行例**

```
$ ruby anemone-mongo.rb
http://www.yahoo.co.jp/
Yahoo! JAPAN
```

スクリプトを実行する際には、MongoDB（MongoDBを解凍したフォルダーのbinフォルダーのmongod.exe）を起動しておいてください。bson_extをインストールしていない場合は、警告が出るのでご注意ください。

なおストレージでMongoDBを利用した場合は、起動ごとにCollectionの初期化が行われます。そのために、SQLite3利用時のように、起動をまたいで訪問ずみのスキップをすることができません。

またDB名とCollection名を変更する場合は、次のように指定します（❶）。名前の変更の際は、mongoを呼び出す必要があります。

DB名とCollection名を変更する anemone-mongo-option.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'mongo'

urls = []
```

```
urls.push("http://www.yahoo.co.jp")

opts = {
  :storage => Anemone::Storage::MongoDB(
    Mongo::Connection.new.db('crawler'),"documents") ●―――――❶

}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-mongo-option.rbの実行例

```
$ ruby anemone-mongo-option.rb
http://www.yahoo.co.jp/
Yahoo! JAPAN
http://www.yahoo.co.jp/r/c5
http://www.yahoo.co.jp/r/c1
http://www.yahoo.co.jp/r/mht
http://www.yahoo.co.jp/r/c2
http://www.yahoo.co.jp/r/c12

~省略~
```

作成したデータベースの確認

Anemoneが作成したデータベースを参照するには、コマンドプロンプトもしくはターミナルからmongoコマンドで行います。専用のツールを利用している場合は、そちらをご利用ください。

◆ 作成したデータベースの参照

```
$ mongo
MongoDB shell version: 2.4.9
connecting to: test

~省略~

>
```

MongoDB内部のDBは、show dbsで確認できます。デフォルトの指定の場合「anemone」になります。先ほどのcrawlerという名前での実行も行っていれば、そちらのデータベースも作成されています。

◆ データベースの確認

```
> show dbs
anemone   0.203125GB
crawler   0.078125GB
local     0.078125GB
```

使用中のデータベースの変更は、useコマンドを利用します。

◆ データベースの変更

```
> use anemone
switched to db anemone
```

データベース中のCollectionの一覧の取得は、show collectionsで行えます。

◆ Collection一覧の取得

```
> show collections
pages
system.indexes
```

Collectionからレコードすべてを取得するのは、db.コレクション名.findコマンドです。

◆ レコードを取得する

```
> db.pages.find()
{ "_id" : ObjectId("536582567a78b0cffc477006"), "url" : "http://www.yahoo.co.jp/", "headers" : BinData(
0,"BAh7DyIJZGF0ZVsGIiJTYXQsIDAzIE1heSAyMDE0IDIzOjU3OjEwIE

~省略~
```

件数を取得するには、db.コレクション名.countコマンドです。

◆ 件数を取得する

```
> db.pages.count()
1
```

検索条件は、findの第1引数で指定します。引数は配列で渡し、「{項目名：検索条件}」といった形式です。

◆ 検索条件の指定

```
> db.pages.find({url : "http://www.yahoo.co.jp/"});
{ "_id" : ObjectId("53c7f92e19238815423bc844"), "url" : "http://www.yahoo.co.jp/", "headers" : BinData
(0,"BAh7ECILc2VydmVyWwYiCm5naW54IglkYXRlWwYiIlRodSwgMTcgSnVsIDIwMTQgMTY6MjY6MjIg
R01UIhFjb250ZW50LXR5cGVbBiIddGV4dC9odG1sOy

~省略~
```

Collectionの特定の項目のみ取得する場合は、findの第2引数で指定します。引数は配列で渡し、「{項目名：表示/非表示}」といった形式です。表示の場合は「1」を、非表示の場合は「0」で指定します。

◆ 特定の項目のみを取得する

```
> db.pages.find({},{url : 1});
{ "_id" : ObjectId("5365858d7a78b0cffc47700b"), "url" : "http://www.yahoo.co.jp/" }
```

MongoDBのコマンド一覧は、helpで表示できます。

◆ コマンド一覧の取得

```
> help
    db.help()           help on db methods
    db.mycoll.help()        help on collection methods
    sh.help()           sharding helpers

~省略~
```

MongoDBから抜ける場合は、exitコマンドを利用します。

◆ MongoDBを終了する

```
> exit
bye
```

4-1-6 MySQLに保存

Anemoneが利用可能なストレージには、MySQLはありません。しかし、必要であればプラグインとして追加することが可能です。ストレージの構成と、追加の仕方を見ていきましょう。

Anemoneのストレージの構造

Anemoneのストレージは、各ストレージの実装とそれを呼び出す「storage.rb」で構成されています。プラグインとしてストレージを追加する際は、実装クラスを追加したうえで、storage.rbに追加したストレージを記述します。

▼ storageの構成

```
├─ storage.rb
├─ storage
│   ├── base.rb
│   ├── exceptions.rb
│   ├── kyoto_cabinet.rb
│   ├── mongodb.rb
│   ├── pstore.rb
│   ├── redis.rb
│   ├── sqlite3.rb
│   └── tokyo_cabinet.rb
```

実装クラスに最低限必要なメソッドは、「base.rb」で定義されている下記のメソッドです。

▼ 実装クラスに必要なメソッド

メソッド	機　能
def initialize(adapter)	初期化
def [](key)	データの抽出
def []=(key, value)	データの登録・更新
def delete(key)	データの削除
def each	すべてのデータを抽出
def merge!(hash)	マージ
def close	クローズ
def size	登録データ数の表示
def keys	キー一覧の取得
def has_key?(key)	キーの存在確認

MySQLプラグインの追加

構造がわかれば、プラグインの実装はそれほど難しくありません。プラグインではまず、MySQLを扱うためのAnemoneのストレージライブラリのロードを行います。任意のライブラリを利用すればよいのですが、ここでは「mysql2」を利用します。mysql2の使い方については、GitHubのmysql2ページに記載されています。

■ GitHubのmysql2ページ
URL https://github.com/brianmario/mysql2

自作したライブラリをプラグインとして使用するためには、まずAnemoneのソースをダウンロードします。GitHubのAnemoneページから、Gitもしくは Zip形式で取得して展開します。

■ GitHubのAnemoneページ
URL https://github.com/chriskite/anemone

ソースからのGemライブラリのインストールは、gemコマンドのビルドとインストールで行います。ビルドの際は、依存関係の指定の仕方などでいくつか警告が出ます。バージョン指定の仕方の問題なので、ビルド自体には影響はありません。

ビルドは、Anemoneを解凍したディレクトリに移動して行います。

◆ ライブラリのビルド・インストール

```
$ cd anemone
$ gem build anemone.gemspec
WARNING: licenses is empty, but is recommended. Use a license abbreviation from:
http://opensource.org/licenses/alphabetical
WARNING: no description specified
WARNING: no email specified
WARNING: open-ended dependency on nokogiri (>= 1.3.0) is not recommended
  if nokogiri is semantically versioned, use:
    add_runtime_dependency 'nokogiri', '~> 1.3', '>= 1.3.0'

～省略～

$ gem install anemone-0.7.2.gem
Successfully installed anemone-0.7.2
Parsing documentation for anemone-0.7.2
Done installing documentation for anemone after 0 seconds
1 gem installed
```

それでは、自作のmysqlライブラリを作成します。実装クラス用のファイル「mysql.rb」を作成し、まずはライブラリがロードできないと、エラーを返すような実装から作ります。

ライブラリのロード

mysql.rb

```
begin
  require 'mysql2'
rescue LoadError
  puts "You need the mysql2 gem to
    use Anemone::Storage::MySQL"

  exit
end
```

次に初期化メソッドです。書き方としてはいろいろありますが、引数としてオプションをハッシュ値で受け取るようにしています。ハッシュ値のなかにデータベース関係のパラメータがあれば、それを設定します。なければデフォルト値を設定します。そして、データベースへ接続し、スキーマを作成します。データベースへの接続は、クラス内で使い回します。

初期化メソッド

mysql.rb

```
def initialize(opts = {})
  host = opts[:host] || 'localhost'
  username = opts[:username] || 'crawler'
  password = opts[:password] || 'anemone_pass'
  database = opts[:database] || 'anemone'
  @db = Mysql2::Client.new(:host => #{host},
    :username => #{username}, :password => #{password},
    :database => #{database})
  create_schema
end
```

次にテーブル作成です。Anemoneが利用するテーブルとして、anemone_storageテーブルを作成します。既に存在している場合はスキップします。テーブルの項目としては、管理IDとデータのキーとデータのみです。このcreate_schemaメソッドは、外部から直接呼ばれることはないので、プライベートメソッドとして追加します。

テーブルの作成　mysql.rb

```
private

def create_schema
  @db.query <<SQL
    create table if not exists anemone_storage (
      id int(11) NOT NULL auto_increment,
      page_key varchar(255),
      page_data BLOB,
      PRIMARY KEY (id),
      key (page_key)
    ) DEFAULT CHARSET=utf8;
  SQL
end
```

同様にプライベートメソッドとして、文字列からハッシュ値を生成するメソッドを追加します。データのキーとなるURLの文字列長が長いことと、SQLで扱いにくい文字を排除するためです。

ハッシュ値の生成　mysql.rb

```
def get_hash_value(key)
  puts "get_hash_value"
  Digest::SHA1.hexdigest(key)
end
```

データ存在確認メソッドは、データベースへの問い合わせを実装します。キーはURLをハッシュ化したものを使います。

データの存在確認　mysql.rb

```
def has_key?(url)
  key = get_hash_value(url)
  result = @db.query(
    "SELECT count(id) FROM anemone_storage WHERE page_key = '#{key}'")

  if result.first['count(id)'] > 0
    return true
  else
    return false
  end
end
```

主要な実装としては、データの抽出と登録・更新です。データ部分の保存方法はいろいろありますが、他のAnemoneのストレージに従い、Marshalダンプしたものを格納します。

データの抽出と登録・更新 mysql.rb

```
def [](url)
  value = @db.query(
    "SELECT data FROM anemone_storage WHERE page_key =
     '#{get_hash_value(url)}'").first['data']

  if value
    Marshal.load(value)
  end
end

def []=(url, value)
  data = Marshal.dump(value)
  key = get_hash_value(url)
  if has_key?(url)
    @db.query("UPDATE anemone_storage SET page_data =
      '#{data}' WHERE page_key = '#{key}'")

  else
    @db.query("INSERT INTO anemone_storage (
      page_key, page_data) VALUES('#{key}', '#{data}')")
  end
end
```

主要部分のみ紹介しましたが、基本的な考え方は他のプラグインと同様です。メソッドごとの実装を、MySQLの処理に置き換えるだけで作成できます。同じRDBということで、SQLite3プラグインを元にSQLの処理部分のみ変更すれば最小限のコーディングで作成できます。作成した実装クラスの全文は、サンプルファイル(mysql.rb)でご確認ください。サンプルファイルについては、p. ii をご参照ください。

最後に、storage.rbにMySQLのプラグインを追加すれば完成です。

MySQLプラグインの追加 storage.rb

```
def self.MySQL(opts = {})
  require 'anemone/storage/mysql'
  self::MySQL.new(opts)
end
```

作成したモジュールをGem形式にビルドするには、次のようにAnemoneのソースのルートディレクトリに移動し、buildコマンドを実行します。

◆ モジュールのビルド

```
$ cd anemone
$ gem build anemone.gemspec
```

ビルドすれば、あとはinstallコマンドで改造したAnemoneライブラリをインストールできます。

◆ ライブラリのインストール

```
$ gem install anemone-0.7.2.gem
```

MySQLを利用する

作成したライブラリを呼び出すスクリプトは次のような形です。MongoDBのスクリプト(→p.209)から、ストレージの部分を「MySQL ()」に変更するだけです(❶)。必要に応じて、オプションを追加して呼び出すようにします。

MySQLに保存する

anemone-mysql.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
  :storage => Anemone::Storage::MySQL(), ●―――――❶
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-mysql.rbの実行例

```
$ ruby anemone-mysql.rb
http://www.yahoo.co.jp/
Yahoo! JAPAN
http://www.yahoo.co.jp/r/mht
http://www.yahoo.co.jp/r/c2

~省略~
```

今回は、MySQLの例で紹介しました。必要なデータストアがなければ、MySQL以外にも追加してみましょう。

4-2 クローラーの開発とデバッグ方法

プログラムを開発する際は、試行錯誤が欠かせません。特にクローラーの場合は、外部からコンテンツをHTTPプロトコルで取得する処理があります。クローラーはプログラムの一度あたりの実行時間が長い傾向にあるため、効率的に開発する方法やデバッグする方法が必須です。プログラムのデバッグ方法全般と、コンテンツ取得部分に特化した方法を紹介します。

4-2-1 Rubyプログラムのデバッグ方法

クローラーの開発にかぎらず、Rubyプログラムのデバッグ方法についてです。ある程度以上の規模になると、Ruby用のIDE（統合開発環境）を利用するのが一番効率がよいのでしょうが、ここではIDEを使わずに、Rubyのみで利用可能なデバッグ方法を紹介します。

変数の中身を表示する

デバッグの重要な情報の1つが、実行時点での変数のなかに何が入っているかです。一番簡単なデバッグ方法として、標準出力に変数の中身を表示して値を確認するという方法があります。Rubyには標準出力に表示するメソッドがいくつかあります。代表的なものはputsやprint、またデバッグ時に便利なpやppがあります。それぞれの使い分けを見てみましょう。

まずputsとpの使い分けについてです。どちらもRuby標準のKernelモジュールのメソッドの1つです。putsメソッドは、引数と改行を標準出力に出力します。引数

のなかに改行コードが含まれていた場合は、改行されて出力されます。これに対してpメソッドは、標準出力に出力するという点は一緒ですが、主にデバッグでの使用を目的としています。

変数の中身を表示する debug_puts_p.rb

```
# -*- coding: utf-8 -*-
puts "文字列"        # => 文字列
p "文字列"           # => "文字列"

puts 15             # => 15
p 15                # => 15

puts "改行¥nコード"  # => 改行 コード
p "改行¥nコード"     # => "改行¥nコード"
```

pメソッドを利用すると、変数が文字列型の場合はダブルクォーテーションで囲まれて表示されます。数値型の場合は、そのままで出力されます。これに対して、putsメソッドは、すべてそのまま表示されます。

また、文字列中に改行コードが含まれていた場合は、putsメソッドの場合は改行されて表示されます(上記の例では、「改行」の後ろで改行されます)。pメソッドの場合は、改行コードを文字列としてそのまま表示します。これらの理由から、pメソッドはデバッグ時に利用しやすいメソッドとなっています。一方で、Windowsのコマンドプロンプトの場合、pで出力したものについては文字コードで表示されます。

◆ debug_puts_p.rbの実行例

```
$ ruby debug_puts_p.rb
文字列
"文字列"
15
15
改行
コード
"改行¥nコード"
```

ppはpretty printライブラリと言って、読みやすいインデントと改行をするライブラリです。標準添付ライブラリなので、インストール不要でRequireするだけで利用できます。pとppの違いは表示形式です。ppの場合は、自動的にインデントと

改行を調整して表示されます。配列やオブジェクトの中身を見る場合は、ppを使うと便利な場合があります。

ppで変数の中身を確認する debug_p_pp.rb

```
# -*- coding: utf-8 -*-
require 'pp'

ary = Array.new(6) { {:foo => :bar} }

p ary
pp ary
```

◆ debug_p_pp.rbの実行例

```
$ ruby debug_p_pp.rb
[{:foo=>:bar}, {:foo=>:bar}, {:foo=>:bar}, {:foo=>:bar}, {:foo=>:bar}, {:foo=>:bar}]
[{:foo=>:bar},
 {:foo=>:bar},
 {:foo=>:bar},
 {:foo=>:bar},
 {:foo=>:bar},
 {:foo=>:bar}]
```

トレースする

Gemのライブラリを利用していると、ライブラリの中身がどのような順番で呼び出されて、どのように動作しているのかがわからないことがあります。そういった場合には、トレースを行うことにより、ライブラリの動きがわかり、理解に繋がります。

Rubyの場合、Kernelモジュールにcallerという呼び出し元の情報を表示するメソッドが用意されています。ppと組み合わせて使うと、見やすいです。

ライブラリの情報を取得する anemone-caller.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'pp'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
```

```
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  # 呼び出し元の情報をスタックトレースで表示
  pp caller()
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

実行すると、次のようにAnemoneライブラリのなかで、どういった順番で呼び出されて実行されているのかが表示されます。

◆ anemone-caller.rbの実行例

```
$ ruby anemone-caller.rb
["/Users/takuro/.rvm/gems/ruby-2.0.0-p353/gems/anemone-0.7.2/lib/anemone/core.rb:91:in `block
in crawl'",
 "/Users/takuro/.rvm/gems/ruby-2.0.0-p353/gems/anemone-0.7.2/lib/anemone/core.rb:83:in
`initialize'",
 "/Users/takuro/.rvm/gems/ruby-2.0.0-p353/gems/anemone-0.7.2/lib/anemone/core.rb:90:in `new'",
 "/Users/takuro/.rvm/gems/ruby-2.0.0-p353/gems/anemone-0.7.2/lib/anemone/core.rb:90:in `crawl'",
 "/Users/takuro/.rvm/gems/ruby-2.0.0-p353/gems/anemone-0.7.2/lib/anemone/core.rb:18:in `crawl'",
 "anemone-caller.rb:11:in `<main>'"]
http://www.yahoo.co.jp/
Yahoo! JAPAN
```

もしくは起動オプションに-r tracerを付けることで、すべてのトレースをすることが可能です。すべてのライブラリの実行ログが出力されるために、膨大な量のスタックトレースが発行されます。

◆ すべてトレースする

```
$ ruby -r tracer anemone-caller.rb
```

デバッグする

Rubyには、コマンドラインから利用可能なデバッグツールがいくつかあります。標準添付ライブラリのdebugや、サードパーティのデバッガーです。サードパーティのデバッガーについては、Ruby 1.8時代にはruby-debugが有名でした。その

後に1.9や2.0などのバージョンアップがあるたびに、派生や類似ライブラリが出てきました。Debuggerやbyebug、pry-byebugなどです。ここでは標準添付ライブラリのdebugのみを紹介します。

debugの使い方は、起動時に-r debugを付けるだけです。

◆ **debugの実行**

```
$ ruby -r debug anemone-caller.rb
Debug.rb
Emacs support available.

/Users/yourhome/.rvm/rubies/ruby-2.0.0-p353/lib/ruby/site_ruby/2.0.0/rubygems/core_ext/
kernel_require.rb:57:
(rdb:1)
```

主なデバッグコマンドは次の表のとおりです。stepもしくはnextでステップイン実行、もしくはcontでスクリプト終了もしくはブレイクポイントまで移動します。ポイントごとに、var localなどのコマンドで変数の中身を表示します。またlistコマンドで周辺のソースを表示します。

◆ 主なデバッグコマンド

コマンド名	省略名	機　能
break [<file>:]<position>|<class>:<method>	b	ブレイクポイントの設定もしくは表示を行う
cont	c	スクリプトが終了するまで、もしくは次のブレイクポイントに到達するまで処理を実行する
step [nnn]	s	1行ずつ実行する。引数を指定した場合は、その行数分だけ実行する
next [nnn]	n	stepと同様だが、メソッド内は実行しない
list [(-|nn-mm)]	l	スクリプトを表示する。引数で範囲指定可能。また「-」の場合は前の行を表示する
quit	q	スクリプトを中断し、デバッガを終了する
var global	var g	グローバル変数を表示する
var local	var l	ローカル変数を表示する
var instance <object>	var i	オブジェクトのインスタンス変数を表示する
var const <object>	var c	オブジェクトの定数を表示する

■ Ruby 2.0.0 リファレンスマニュアル（debugライブラリ）
URL **http://docs.ruby-lang.org/ja/2.0.0/library/debug.html**

ある一定規模以上のプログラムの場合は、IDEのデバッガーを使うと使い勝手がよいでしょう。また、小さなプログラムの場合は、デバッガーも不要で標準出力のみで十分な場合もあります。なかなかコマンドラインからのデバッガーを使うタイミングは難しいところがありますが、1つの手段として覚えておくと役に立つこともあるでしょう。

4-2-2　開発プロキシを使ったクローラーの開発

クローラーは、外部のサーバと通信してコンテンツを取得します。そのため、開発の段階で何度もサーバへのアクセスを繰り返すこともあります。通信処理を含むプログラムは、ローカルのみで完結するプログラムに比べて、何倍も処理時間がかかります。

また、そもそもネットワークが通じていない状態では開発もできません。それ以外にも、何度もコンテンツを取得にいくことで、Webサイトの提供者に迷惑をかける可能性もあります。トラブルの原因にもなりかねません。そこで、開発段階では初回のみコンテンツを取得するように工夫することで、諸問題を解決することができます。

開発用プロキシ「CocProxy」

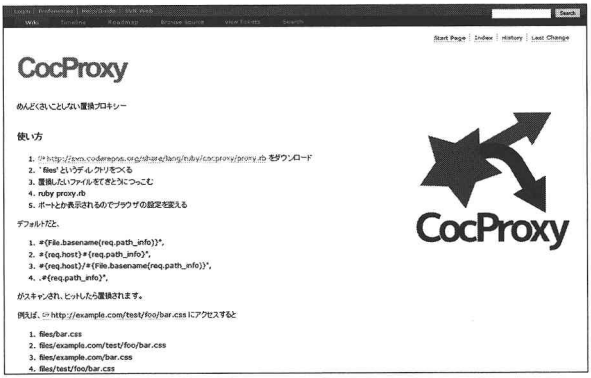
クローラーに必要な開発用プロキシとは何でしょうか？　初回アクセス時は、Webサイトを訪問しコンテンツを取得します。コンテンツを取得すると、ローカルにファイルを保存し、2回目以降のアクセス時にはローカルのキャッシュを返すようなものが理想的です。このような動作をするRubyのモジュールが「CocProxy」です。

CocProxyは、Rubyの標準ライブラリで実装された開発用プロキシです。また、基本的に1つのファイルで完結することを目標として作られています。このため、Ruby環境があれば簡単に使うことができます。Ruby 1.9をベースに開発されていますが、Ruby 2.0でも問題なく動作します。

CocProxyについては、以下のサイトでご確認ください。

■ CocProxyサイト
URL http://coderepos.org/share/wiki/CocProxy

CocProxyサイト



CocProxyは、プログラムなどの利用者がネットワーク経由でのファイル取得時に動作します。デフォルトの設定では、リクエストされたファイルに対して、まずローカルのファイルがないかを確認します。存在する場合は、それを返します。ない場合は次にキャッシュを確認します。その両方がない時に始めてWebファイルを取得しにいきます。Webからファイルを取得後に、キャッシュとして保存するために、2回目以降は取得しにいかないようになります。

この2つの動作により、クローラー開発に寄与します。ローカルにファイルを置いておくと、オフラインでも開発可能です。また、キャッシュを利用することで、Webサイトへのアクセス数を減らし、待ち時間を減らすことができ、取得側のWebサイトに負荷をかけません。

CocProxyのローカルファイルの参照ルールについては、以下のとおりです。

- ①ファイル名で参照
- ②「http¥https」なしで、FQDNなしのパスでの参照
- ③ドメイン名/ファイル名での参照
- ④絶対パスでの参照

Yahoo!のセキュリティガイドに二度参照しにいった場合の動作（参照するファイル）は、以下のとおりです。

- ①Checking files/1a.html
- ②Checking files/security.yahoo.co.jp/guide/1a.html

- ③Checking files/security.yahoo.co.jp/1a.html
- ④Checking files/./guide/1a.html
- ⑤Cached: http://security.yahoo.co.jp/guide/1a.html
- ⑥Checking files/1a.html
- ⑦Checking files/security.yahoo.co.jp/guide/1a.html
- ⑧Checking files/security.yahoo.co.jp/1a.html
- ⑨Checking files/./guide/1a.html
- ⑩From Cache: http://security.yahoo.co.jp/guide/1a.html

「http://security.yahoo.co.jp/guide/1a.html」ローカルファイルを参照した後で、ネットワーク経由でファイルを取得にいっています。初回取得時にキャッシュとして保存しています。2回目はキャッシュから返しています。

CocProxyのインストールと起動

CocProxyのインストールは、ダウンロードするだけです。ダウンロード後に、任意の場所に配置しましょう。

■ CocProxyのダウンロード
URL **http://svn.coderepos.org/share/lang/ruby/cocproxy/proxy.rb**

起動は、「proxy.rb」をrubyからキックするだけです。デフォルトでは、5432ポートを利用します。

◆ **CocProxyの実行**

```
$ ruby proxy.rb
Use default configuration.
Port : 5432
Dir  : files/
Cache: true
Rules:
  1. #{File.basename(req.path_info)}
  2. #{req.host}#{req.path_info}
  3. #{req.host}/#{File.basename(req.path_info)}
  4. .#{req.path_info}
```

また標準ライブラリ以外のモジュールや、設定ファイルで挙動を変更できるバージョンについても公開されています。

■ その他の開発プロキシ

URL http://svn.coderepos.org/share/lang/ruby/cocproxy/

プログラムからのCocProxyの利用

CocProxyをプログラムから利用するには、「:proxy」として指定することで可能です。Nokogiri (open-uri) とAnemoneでの利用例は、次のとおりです。なお、スクリプトの実行時は、CocProxyを起動させておいてください。

NokogiriでCocProxyを利用する

nokogiri-proxy.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri::HTML(open(
  'http://www.yahoo.co.jp', :proxy => 'http://localhost:5432'))

puts doc.title  # => Yahoo! JAPAN
```

◆ nokogiri-proxy.rbの実行例

```
$ ruby nokogiri-proxy.rb
Yahoo! JAPAN
```

AnemoneでCocProxyを利用する

anemone-proxy.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp/")

opts = {
  :proxy_host => 'localhost',
  :proxy_port => '5432',
  :obey_robots_txt => true,
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-proxy.rbの実行例

```
$ ruby anemone-proxy.rb
Yahoo! JAPAN
```

プロキシのログから、キャッシュが有効になっていることがわかります。ログは、最初の5行が「nokogiri-proxy.rb」、次の5行が「anemone-proxy.rb」のものです。

プロキシのログ

```
Checking files//
Checking files/www.yahoo.co.jp/
Checking files/www.yahoo.co.jp//
Checking files/./
From Cache: http://www.yahoo.co.jp/
Checking files//
Checking files/www.yahoo.co.jp/
Checking files/www.yahoo.co.jp//
Checking files/./
From Cache: http://www.yahoo.co.jp/
```

CocProxyは、[Ctrl] + [C] キーで終了します。

4-3 クローリングとスクレイピングの分離

クローラーの構造は、コンテンツの取得（クローリング）とデータの解析（スクレイピング）とデータの保存の3つの機能に分類できます。このなかで開発に占める割合は、クローリングとスクレイピングの比重が大きいです。また、Webサイトの性質上、HTMLは頻繁に変更される可能性が高いです。

HTMLが変更された場合は、スクレイピング部分を変更する必要があります。そのため、クローリング部分とスクレイピング部分を分離すると、変更がしやすく保守性が高いプログラムになります。AnemoneやNokogiriを利用して、分離した例を紹介します。

4-3-1　スクレイピング部分の分離

クローリングとスクレイピングの分離は、両者の境界線で考えるとよいです。クローリングの役割は、取得するURLを決定してダウンロードしたドキュメントを

渡すまでです。そしてスクレイピングは、取得したドキュメントを解析する部分となります。解析結果の利用用途に応じて考えればよいです。例えば、解析部分で表示する、変数に格納して返す、データベースなどに保存して永続化するなど、いろいろな方法があります。

次のサンプルスクリプトは、クローラー部分とスクレイピングを行うパーサー部分を分離した一番単純な例です。Anemoneは、ダウンロードしたコンテンツをAnemone::Pageのオブジェクトに格納します。そのオブジェクトごと渡して、スクレイピングの部分をparserメソッドで行っています。

クローリングとスクレイピングの分離 separate-parser.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

def crawl(url)
  Anemone.crawl(
    url, :depth_limit => 0) do | anemone |
    anemone.on_every_page do |page|
      parser(page)
    end
  end
end

def parser(page)
  puts page.doc.xpath("//title/text()")
end

url = "http://www.yahoo.co.jp/"
crawl(url)
```

◆ separate-parser.rbの実行例

```
$ ruby separate-parser.rb
Yahoo! JAPAN
```

4-3-2 分離度を上げる

先ほどの例では、クローリングとスクレイピングを分離しているものの、分離度のレベルは低いです。理由としては、クローラーとパーサーのデータの受け渡しが、Anemoneのオブジェクトでやり取りしているためです。また、クローラーとパー

サーが同一のプログラムで直列に動いています。

分離度を上げると、クローラープログラムとパーサープログラムの2つに分離できます。その場合のメリットは、2つのプログラムを並列に動かすことが可能になり、かつどちらか一方がボトルネックになっている場合は、そちら側のリソースを増やすことで効率的に動かすことが可能になることがあります。

次の2つのスクリプトは、クローラーとパーサーの2つに分離した例です。クローラーは、ファイルを取得して保存するだけです。パーサーは、所定のディレクトリ以下にファイルがあれば、スクレイピングを行います。サンプルなので、クローラーは1つのファイルしか取りませんが、実際の運用ではファイルを取得し続ける処理に特化すれば効率がよくなります。また、パーサーも一度ファイルの一覧を取得して、処理が完了すれば終了するようになっています。両者とも定期起動もしくはデーモンなどで常駐するようにしておけば、24時間動き続けるクローラーとなります。

分離クローラー structuration-crawler.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'digest/sha1'

def crawler(url)
  hash_str = Digest::SHA1.hexdigest(url)
  path="files/"+hash_str
  if !is_exist?(path)
    source = open(url).read
    open(path,'w+b') {|f| f.write(source)}
  end
end

def is_exist?(path)
  File.exist?(path)
end

url = "http://www.yahoo.co.jp/"
crawler(url)
```

実行前にスクリプトと同じ場所に「files」ディレクトリを作成しておいてください。そこにファイルを保存します。

◆ structuration-crawler.rbの実行例

```
$ ruby structuration-crawler.rb
$ ls files/
c421c316c0d603db79c62f10364d068aaf43137c
```

実行するとfilesディレクトリにファイルが作成されます

分離パーサー structuration-parser.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'

def parser(file)
  doc = Nokogiri::HTML(open(file))
  puts doc.title  # => Yahoo! JAPAN
end

def get_file_list(target,parsed)
  puts target
  Dir::foreach(target).each {|file|
    next if file == "." or file == ".."
    parser(target+"/"+file)
    move_file(target+"/"+file,parsed+"/"+file)
  }
end

def move_file(from,to)
  puts from
  puts to
  File.rename(from,to)
end

get_file_list("files","parsed_files")
```

同様に実行前にプログラムと同じ場所に「parsed」ディレクトリを作成しておいてください。そこにファイルを保存します。

◆ structuration-parser.rbの実行例

```
$ ruby structuration-parser.rb
files
Yahoo! JAPAN
files/c421c316c0d603db79c62f10364d068aaf43137c
parsed_files/c421c316c0d603db79c62f10364d068aaf43137c
```

4-4 クローラーを効率的に動かすには

クローラーを効率的に動かすには、2つの方法があります。1つは、単位時間あたりにできるだけ多くのコンテンツをダウンロードする方法です。もう1つは、無駄なダウンロードをできるだけ減らす方法です。

単位時間あたりのダウンロード数を増やすには、並列実行や、複数のクローラーを同時に起動して、プログラムの多重度を上げることで実現できます。一方で、他人が運営しているWebサイトについて短い間隔でダウンロードを繰り返すことは、サイトの負荷を含め迷惑をかけることに繋がりかねません。方法論として一通り説明しますが、実際に運用する際は十分な注意と配慮が必要です。

もう1つの無駄なダウンロードを減らす方法については、取得ずみのコンテンツは取得しないといった方法や、最終更新日もしくは最終取得日を見て取得の判断をするといった方法があります。この方法については、Webサイトの運営側の負荷を軽減できます。まずは、こちらの方法を検討すべきでしょう。

4-4-1　多重度を上げる

プログラムの多重度を上げるには、プログラム内で複数同時に実行する方法や、複数のクローラーを同時に起動する分散処理で実現できます。

プログラム内での複数同時実行には、処理方式について「並行処理」と「並列処理」があります。言葉としてよく似ているので混同しやすいのですが、並行処理については、実行の順番を制御しながら同期的に実行される処理のことです。並列処理は、複数の実行が非同期で実行される処理です。

一般的にクローラーの処理の場合、ボトルネックはクローリングしてデータを取得する部分になります。例えば、コンピュータ内部のみで完結する処理の場合、数ミリ秒程度で終了します。これに対して、ネットワークを通じて外部のシステムと通信する処理であれば、数十～数百ミリ秒や、場合によっては数秒という単位の時間がかかります。クローラーを並列処理する場合であれば、一番ボトルネックになるダウンロード処理を複数走らせて、ダウンロード完了したものから随時処理するというのが基本となります。

Rubyについては、「thread」という並行処理を扱う組み込みライブラリと、並列処理を行う「Parallel」という拡張ライブラリがあります。

threadによる並行処理

Anemoneは、threadを利用して並行処理を行っています。デフォルトでは4つのthreadを利用し、「:threads」パラメータで変更可能です(❶)。下記の例は、10個のURLをダウンロードするスクリプトです。オプションの指定で、thread数を10個に増やしています。

threadで並行してダウンロードを行う anemone-thread.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466284/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/571582/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492152/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466286/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466282/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492054/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466290/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492166/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466298/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466294/")

opts = {
  :obey_robots_txt => true,
  :threads => 10, ●―――――❶
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-thread.rbの実行例

```
$ ruby anemone-thread.rb
http://www.amazon.co.jp/gp/bestsellers/books/492152/
Amazon.co.jp &#131;x&#131;X&#131;g&#131;Z&#131;&#137;&#129;[: &#131;m&#131;&#147
;&#131;t&#131;B&#131;N&#131;V&#131;&#135;&#131;&#147; &#130;&Igrave;&#146;&#134;
&#130;&Aring;&#141;&Aring;&#130;&agrave;&#144;l&#139;C&#130;&Igrave;&#130; 
&#130;&eacute;&#143;&curren;&#149;i&#130;&Aring;&#130;&middot;
http://www.amazon.co.jp/gp/bestsellers/books/466284/

~省略~
```

上記のスクリプトで、threadsを1個と10個に変更したものを、それぞれ10回実行し、処理時間の平均時間を比較したのが下記の表です。如実に効果が出ているのがわかります。

thread数による処理時間の違い

thread数	1個	10個
処理時間	11.7459秒	2.6648秒

Parallelによる並列処理

同様にParallelで並列化した例です。in_threadsパラメータで、スレッド数を調整できます(❶)。

スクリプトの実行には、Parallelのインストールが必要です。gemからインストールすることができます。

◆ Parallelのインストール

```
$ gem install parallel
```

Parallelで並行してダウンロードを行う　anemone-parallel.rb

```
# -*- coding: utf-8 -*-
require 'parallel'
require 'nokogiri'
require 'open-uri'

urls = []
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466284/")
urls.push(
```

```
  "http://www.amazon.co.jp/gp/bestsellers/books/571582/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492152/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466286/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466282/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492054/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466290/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/492166/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466298/")
urls.push(
  "http://www.amazon.co.jp/gp/bestsellers/books/466294/")

Parallel.each(urls, in_threads: 10) {|url| ●──────❶
  doc = Nokogiri::HTML(open(url))
  puts doc.title
}
```

◆ anemone-parallel.rbの実行例

```
$ ruby anemone-parallel.rb
Amazon.co.jp ?x?X?g?Z???[:?????E?a?Z?E?i?D?o?c ?l???A?A?a?l?C?l???e???i?A?・
Amazon.co.jp ?x?X?g?Z???[:?r?W?l?X?E?o?l ?l???A?A?a?l?C?l???e???i?A?・
Amazon.co.jp ?x?X?g?Z???[:?A?[?g?E???z?E?f?U?C?? ?l???A?A?a?l?C?l???e???i?A?・
Amazon.co.jp ?x?X?g?Z???[:?a?w?E?o?w?E?A?i?w?E???E?w ?l???A?A?a?l?C?l???e???i?A?・

～省略～
```

分散処理の高速化

分散処理については、同じPC端末/サーバから多数のプログラムを実行しても、リソースを取り合うのであまり意味がありません。複数のサーバを利用することで効果を発揮します。

また分散処理の場合、実行の制御が重要になります。クローラーでは、キュー(Queue)やDBなどを利用して取得対象を制御することが可能です。並列・分散処理の他の方法については、「6-5 さらなる高速化の手法」(→p.410)でEventMachineを利用した例をあげます。

4-4-2 タイムアウトの調整

何らかの理由によりクロール対象のコンテンツが取得できない場合や、応答が極端に遅い場合は、データ取得のタイムアウト値を設定することにより全体でのクロール時間の短縮を図れます。Anemone、open-uriともに「read_timeout」オプションで、秒単位でタイムアウト値を設定可能です。

次のサンプルでは、それぞれ5秒に設定しています(❶)。

Anemoneのタイムアウト設定 anemone-timeout.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.yahoo.co.jp")

opts = {
  :obey_robots_txt => true,
  :read_timeout => 5, ●──────❶
  :depth_limit => 0
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    puts page.doc.xpath("//title/text()").to_s if page.doc
  end
end
```

◆ anemone-timeout.rbの実行例

```
$ ruby anemone-timeout.rb
http://www.yahoo.co.jp/
Yahoo! JAPAN
```

open-uriのタイムアウト設定 anemone-timeout-openuri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

urls = []
urls.push("http://www.yahoo.co.jp")

nokogiri_options = {
  :read_timeout => 5 ●──────❶
```

```
}

urls.each{ |url|
  html = open(url, nokogiri_options)
  doc = Nokogiri::HTML(html)
  puts doc.title
  puts url
}
```

◆ anemone-timeout-openuri.rbの実行例

```
$ ruby anemone-timeout-openuri.rb
Yahoo! JAPAN
http://www.yahoo.co.jp
```

4-4-3 HTTP Compressionによる通信データの圧縮

HTTP通信の効率化手段の1つにHTTP Compressionという手法があります。サーバ・クライアント間の通信を圧縮して、使用する帯域を小さくして効率的に通信する手法です。圧縮の方式としては、gzipやdeflateなどいくつかあります。圧縮の有無および方式は、サーバ側が指定します。クライアント側が受け入れ可能であれば、サーバ側が提示した圧縮方式で通信します。

クローラープログラムの場合でも通信データの圧縮は利用できます。AnemoneのHTTP通信の実装は、内部的にopen-uriを利用しています。そして、open-uriはNet::HTTPのラッパープログラムです。Net::HTTPは、HTTP Compressionに対応していて、サーバ側が利用可能な場合は自動的に選択するようになっています。そのため、Anemoneやopen-uriを使用する場合で、サーバ側がHTTP Compressionに対応している場合は、自動的に利用することになります。

4-4-4 未取得のデータのみ取得する

クローラーを効率的に動かすのに大切なのは、データのダウンロードにかかる時間をできるだけ短くすることです。そのなかで一番効率的なのは、データをダウンロードしないことです。つまり、取得ずみや不要なデータについて、ダウンロードをスキップすることです。

Anemoneの場合、基本的には取得ずみのデータはスキップします。しかし、データストアがSQLite3以外の場合、プログラムの起動時にデータストアの内容をクリアするので、起動をまたいでの取得ずみデータのスキップはできません。

Anemone自体を修正するか、別の方法を検討する必要があります。

取得ずみのデータのコントロール方法としては、プロキシサーバを利用する方法があります。プロキシサーバ側で取得ずみのデータをキャッシュさせ、2回目以降の取得についてはキャッシュのデータを返すことでダウンロードを抑制します。また、プロキシサーバ側でキャッシュ期間をコントロールすることで、例えば1日以上経過したデータについては、あらためてダウンロードし直すといったことも可能です。プロキシサーバを利用すると、プログラム側は何もすることもなくデータ取得のコントロールを行うことができます。

プロキシサーバを利用する方法については、p.227でサンプルを掲載しています。

4-4-5　エラーコードに対する処理

クローラーを効率的に動かす、またはWebサイト運営者と揉め事を起こさないために大切な点があります。それは、エラーコードを監視し、想定に反してエラーが起こった場合は、ただちにクローリングを中止するように作り込んでおくことです。

クローラーが受け取るべきHTTPのステータスコードは、200です。それ以外の場合は、何らかの問題が起こっている可能性が高いです。特に500番台はサーバ内部でエラーが起こっている状況です。クローラーからのアクセス過多が原因で500エラーが引き起こされた場合、業務妨害で訴えられる可能性すらあります。

クローラーが遭遇するであろう主なステータスコードが次の表です。400番台が、クライアント側起因のエラーです。500番台が、サーバ側で発生したエラーです。400、500番台のエラーコードが発生した時は、特に注意が必要です。

主なHTTPステータスコード一覧

コード	コード名	意　味
200	OK	リクエストは成功した
301	Moved Permanently	恒久的に移動した
302	Found	リクエストしたリソースが一時的に移動された
400	Bad Request	リクエストが不正
401	Unauthorized	認証が必要
404	Not Found	未検出。リソースが見つからない。もしくは、アクセス権がない
500	Internal Server Error	サーバ内部エラー
501	Not Implemented	実装されていないメソッドを使用した

エラーコードへの対処例

エラーコードへの対処としては、正常系のステータスコードに対しては処理続行、404など今後訪問する必要がないURLに対しては除外リストに追加、それ以外のエラーコードについては、エクセプションを発生させて処理を停止するなどがあります。

実行の際には、クロール先のURLを指定してください(❶)。

エラーコードへの対処 anemone-errorcode.rb

```
# -*- coding: utf-8 -*-
require 'anemone'

urls = []
urls.push("http://www.hatena.ne.jp/hogehogehoge") ●―――❶

opts = {
  :depth_limit => 1,
  :obey_robots_txt => true
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.on_every_page do |page|
    puts page.url
    raise '500 Error!:' + page.url.path.to_s if page.code = 500
  end
  anemone.after_crawl do |pages|
    puts "hoge"
  end

end
```

◆ anemone-errorcode.rbの実行例

```
$ ruby anemone-errorcode.rb
anemone-errorcode.rb:14:in `block (2 levels) in <main>': 500 Error!:/hogehogehoge (RuntimeError)

~省略~
```

4-5 Anemoneのオプション一覧

これまで個別に取り扱ってきたAnemoneのオプションについて、あらためてまとめ直します。対象としているバージョンは、0.7.2です。

4-5-1　Anemoneのオプション

次の表が、Anemoneのオプションをまとめたものです。

▼ Anemoneのオプション一覧

オプション名	デフォルト値	詳　細
threads	4	ページ取得の際に同時実行するスレッド数。数が多いほど多重度が高くなる
verbose	false	Anemone実行の詳細ログの表示
discard_page_bodies	false	NokogiriオブジェクトおよびHTML本文を破棄する
:user_agent	"Anemone/#{Anemone::VERSION}"	ユーザーエージェントの設定をする
:delay	0	クローリングの間隔（秒）を指定。0秒より大きい値を設定している場合は、threadsの値に1が指定される
:obey_robots_txt	false	robots.txtに従うかどうか
:depth_limit	false	探索の深さの設定。falseの場合は、無制限で探索
:redirect_limit	5	何回までHTTPのリダイレクトを許可するか
:storage	nil	どのストレージを利用するか。デフォルトはメモリ
:cookies	nil	クッキー名の指定
:accept_cookies	false	クッキーを受け入れるか
:skip_query_strings	false	URLの引数を無視するか
:proxy_host	nil	プロキシサーバのホスト名
:proxy_port	false	プロキシサーバのポート番号
:read_timeout	無制限	HTTPの読み込みタイムアウト（秒）

4-5-2　ストレージオプション（storage）

ストレージオプションの一覧です。ストレージごとに挙動が違うので、注意が必要です。

ストレージオプション一覧

ストレージ	引数(デフォルト値)	詳　細
Hash	なし	メモリを利用する(デフォルト)
PStore	ファイル名(null)	ファイルに保存する
TokyoCabinet	DBファイル名(anemone.tch)	TokyoCabinetに保存する
KyotoCabinet	DBファイル名(anemone.kch)	KyotoCabinetに保存する
MongoDB	MongoDBコネクション(anemone),コレクション名(pages)	MongoDBに保存する
Redis	Redisオプション配列(anemone)	Redisに保存する
SQLite3	DBファイル名(anemone.db)	SQLite3に保存する

4-5-3 クローリング間隔オプション(delay)

delayはクローリング間隔を指定するオプションです。単位は秒です。

注意点としては、値を0秒より大きい値で設定した場合は、threadsの値が1(同時実行しない)が設定されます。オプションでthreadsの値を設定していた場合でも、上書きされます。

4-5-4 巡回戦略オプション(skip_query_strings)

skip_query_stringsは、URLのパラメータ部分を無視するかどうかを指定するオプションです。

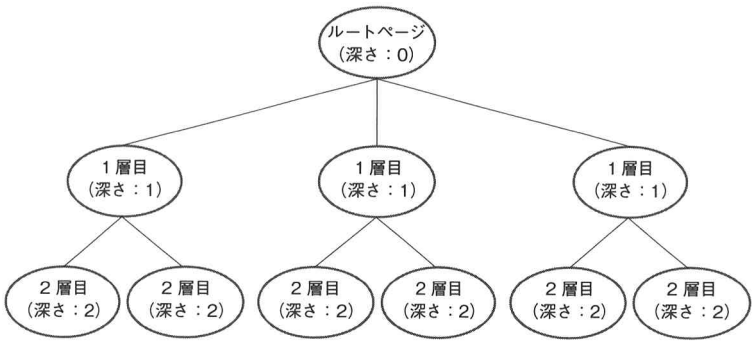
デフォルトはFalseで、パラメータ部分を無視しません。Trueの場合はパラメータ部分は無視され、下記の2つのURLは同じURLと見なされます。Falseの場合は、別物として扱われます。

- http://example.com/page?id=10
- http://example.com/page?id=20

4-5-5 探索戦略オプション(depth_limit)

depth_limitはページの巡回の深さを整数値で指定します。巡回の深さは、ルートページからの距離で計算されます。ルートページは「0」で、ルートページ内にあったリンク先は「1」になります。以下、1ずつ加算されていきます。デフォルトはFalseで、無限に巡回します。「0」を指定した場合は、ルートページのみ巡回します。

▼ページ階層の数え方



4-6 APIを利用した収集

本書では、クローラーでのデータ収集を中心に取り扱っています。しかし、データを収集するという目的に対しては、クローラー以外にもさまざまな選択肢があります。その1つがサイト運営者側が提供しているAPIです。

4-6-1　APIを利用するメリット

APIは、アプリケーション・プログラミング・インターフェース（Application Programming Interface）の略であり、ソフトウェア間でやり取りするための仕様です。

HTMLをスクレイピングしてデータを収集する場合は、HTML変更の影響を受けやすく、その都度メンテナンスが必要なことや、データのみではなく人間が理解できるような視覚的な表現を含むなど、データとして見ると転送効率が悪いことがあります。これに対してAPIは、ソフトウェアが利用することを前提としているために、一般的にはインターフェースの一貫性が高く、変更の影響を受けることは少ないです。

また、データを受け渡しするAPIの場合は、データのみやり取りするために、無駄な部分が少なく転送効率が高いです。サイト運営者としても、Webサイトをスクレイピングされるよりも API を利用される方がメリットが大きく、大手のサービスであればAPIが提供されていることが多いです。例えば、Amazonであれば商品情報や関連コンテンツを検索できるProduct Advertising API、Googleの検索や地図APIや、Twitter、Facebookの各種APIなどさまざまなAPIが提供されています。

クローラーによるスクレイピングについては、運営者側の判断により有害と見なされることがあり、グレーゾーンの部分があるのも否めません。それに対して、運営者が提供するAPIであれば、利用規約に従って利用するかぎりは問題になることはありません。また、一般的にはAPIを利用する方が、クローラーを作成するより簡単です。APIが提供されている場合は、まずはAPIの利用から検討するようにしましょう。

4-6-2 Amazon Product Advertising API

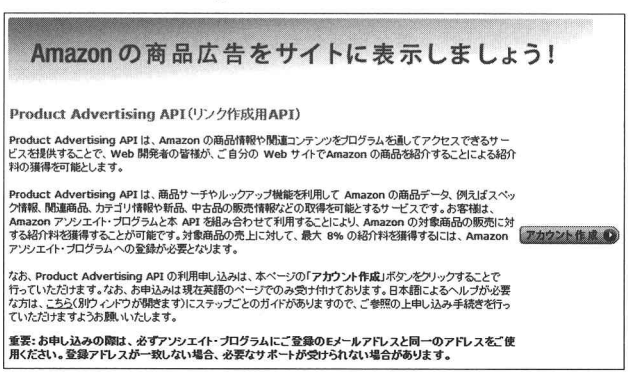
APIを利用したプログラムの例として、Amazonの「Product Advertising API」を使ってみます。このAPIを利用することにより、プログラムを通して商品の検索や商品情報の取得が可能となります。

Amazon Product Advertising APIの利用登録

Amazon Product Advertising APIを利用するには、米国のAmazon.comのアカウント作成のうえで、Product Advertising APIライセンス契約をする必要があります。アカウントの登録およびライセンス契約は、Product Advertising APIのトップページのアカウント作成から行います。

■ Product Advertising API
URL **https://affiliate.amazon.co.jp/gp/advertising/api/detail/main.html**

▼ Product Advertising APIトップページ

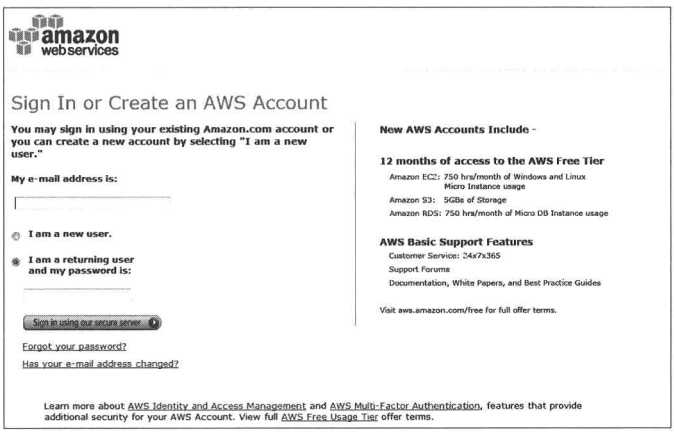


アカウント取得後に、Security CredentialのページでAccess KeyとSecret Access Keyを取得します。Secret Access Keyは、キーの作成時のみしか取得できないために、なくさないように大切に保存しておきましょう。

■ Security Credentialのページ

URL https://console.aws.amazon.com/iam/home?#security_credential

Security Credentialのページ



Amazon Product Advertising APIのラッパーライブラリ

Amazon Product Advertising APIの利用は、各種パラメータを埋め込んだURLでリクエストを送ります。レスポンスは、XML形式で返ってきます。直接APIを操作することも可能ですが、URLの生成やレスポンスフィールドのパースなどが多少煩雑なので、ラッパーライブラリを利用すると簡単に利用できます。いくつかモジュールはありますが、2014年現在で一番利用されているのは、「Amazon ECS」というサードパーティのモジュールです。

インストールは、gemで行えます。

◆ Amazon ECSのインストール

```
$ gem install amazon-ecs
```

インストール後に、gem list amazon-ecsで次のように出力されれば成功です。

◆ インストールの確認

```
$ gem list amazon-ecs

*** LOCAL GEMS ***

amazon-ecs (2.2.5)
```

Amazon Product Advertising APIの使用例

Product Advertising APIは、商品検索系のメソッドと商品購入系のメソッドがあります。クローラーの代替となるのは、商品検索系のメソッドです。下記のスクリプトは、一覧検索と個別商品の表示の例です。

実行の際には、先ほど取得したAccess KeyとSecret Access Keyを設定してください。また、「:associate_tag」に、登録時に指定したアプリケーション名を設定してください。

一覧検索と個別商品の表示

item-search.rb

```
# -*- coding: utf-8 -*-
require 'amazon/ecs'
require 'pp'

Amazon::Ecs.options = {
  :associate_tag => 'ASSOCIATE_TAG'
  :AWS_access_key_id => 'AWS_ACCESS_KEY'
  :AWS_secret_key => 'AWS_SECRET_ACCESS_KEY'
}

# 商品検索
opts = {
  :country => 'jp',
  :author => '北方謙三'
}
res = Amazon::Ecs.item_search('三国志', opts)
res.items.each do |item|
  puts item.get('ItemAttributes/Title')
end

# 個別商品の詳細表示
res = Amazon::Ecs.item_lookup(
  'B00JXEFT6Y', :response_group => 'Small,
  ItemAttributes, Images', :country => 'jp')
pp res
```

◆ item-search.rbの実行例

```
$ ruby item-search.rb
文庫版三国志完結記念セット(全14巻)
三国志 (1の巻) (ハルキ文庫―時代小説文庫)
三国志 10 (バンブーコミックス)
三国志〈2の巻〉参旗の星 (ハルキ文庫―時代小説文庫)
三国志　1 (バンブーコミックス)
三国志〈3の巻〉玄戈の星 (ハルキ文庫―時代小説文庫)

~省略~
```

APIを利用すると、クローラーを作成しAmazonのページから一覧と個別商品を抜き出すよりも、はるかに少ない労力で同じことができます。本書の主要テーマはクローラーの作成ですが、データを収集するという目的であれば積極的にAPIを利用すべきと考えています。

それでは次の章から、個別のサイトやサービスを対象に、より実践的なクローラーを作成していきましょう。

Chapter 5

目的別クローラーの作成

5-1 Googleの検索結果を取得する

さまざまなクローラーを作成するにあたり、まずは検索エンジンから情報を抜き取る方法をマスターします。検索エンジンから抜き出す情報としては、検索結果として返ってくるサイト名・URL・概要の他に、検索結果数というものも情報として価値があります。ここで紹介する方法は一番初歩の部分ですが、今後いろいろな組み合わせで利用できるでしょう。

5-1-1 Googleの検索結果のスクレイピング

検索エンジンとして欠かすことができないのがGoogleです。Googleの検索結果は少し癖があります。例えば、ユーザーエージェントにより検索結果のHTML自体が大幅に変わってくるため、ブラウザでIDなどを確認してスクリプトに落とし込もうとした時に、想定どおりに動かなくてとまどうことがあるかもしれません。しかし、検索方法自体はシンプルです。

- https://www.google.com/search?q=サーチキーワード

サーチキーワードのところに、URIエスケープした文字列を配置します。Rubyの場合では、URI#escapeメソッドで文字列をURIエスケープできます。例えば、「クローラー」という文字列をURIエスケープする場合には、次のような形になります。

```
escaped_url = URI.escape("https://www.google.com/search?q=クローラー")
```

URIエスケープを配置すると、次のようになります。

```
https://www.google.com/search?q=
  %E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9%E3%83%BC
```

ASCII文字列（日本語の文字列以外）は、エスケープ前と後でも同じなのでキーワードのみURIエスケープして繋ぎ合わせても問題ありません。次のスクリプトは、Nokogiriを利用してGoogleの検索結果をスクレイピングした例です。検索結果数と、検索結果のサイトとURLを表示しています。

なお、Googleの検索結果をブラウザ上でクリックした場合は、Googleのカウントプログラムを経由して実際のページが表示されることになります。そのため、遷移先のURLについては、URLのパラメータの一部として組み込まれています。そ

こで、CGI#parseメソッドを利用して、パラメータを分解のうえで取得しています。

Googleの検索結果を取得する nokogiri-google.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'
require 'uri'
require 'cgi'

escaped_url = URI.escape(
  "https://www.google.com/search?q=クローラー&oe=utf-8&hl=ja")
doc = Nokogiri::HTML(open(escaped_url))

# 検索結果の数
puts doc.xpath("//*[@id='resultStats']/text()")

doc.xpath('//h3/a').each do |link|
  puts CGI.parse(link[:href])["adurl"]
  puts link.content
end
```

なお、Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

◆ nokogiri-google.rbの実行例

```
$ ruby nokogiri-google.rb
約 552,000 件
クローラ - Wikipedia
クローラとは - IT用語辞典 e-Words
クローラー の画像検索結果
クローラーとは (クローラーとは) [単語記事] - ニコニコ大百科
クローラーとは／検索エンジンの仕組みをおさらい。クローラー対策とステータスコー ...
Google クローラ - ウェブマスター ツール ヘルプ
製品情報：クローラクレーン | コベルコクレーン株式会社
クローラクレーン | 日立住友重機械建機クレーン

~省略~
```

なお、Googleの検索結果の数については、それほど正確な値は出さない仕様になっています。あくまで参考程度に利用するのがよいでしょう。

また、Googleは短期間のうちに大量のアクセスを繰り返すと、警告とともにブロックされ一定時間、同一のIPからの検索が一切できなくなります。会社や学校などでプロキシサーバ経由でアクセスして同一のIPを利用している場合は、その組織全体に影響を与えます。利用が必要な場合は、細心の注意を払う必要があります。本来であればAPIを利用すべきですが、制約が厳しいので難しいところです。

5-1-2　Gemを利用する

上記と同じ処理をするGemのライブラリがいくつか公開されています。そのなかの1つに、「google-search」があります。このライブラリは、Web検索の結果以外にも画像検索やブログ、ニュース検索といろいろな検索に対応しています。インストールは、gemからライブラリ名を指定するだけで可能です。「Successfully installed google-search-1.0.3」と表示されれば成功です（バージョンは2014年7月現在です）。

◆ google-searchのインストール

```
$ gem install google-search
```

google-searchライブラリを使ったスクリプトは、次のとおりです。エンコードやパース処理をライブラリが行うために、短い記述量で利用できます。

■google-searchを使って情報を取得する　　google-search-api.rb

```
# -*- coding: utf-8 -*-
require 'google-search'

Google::Search::Web.new(:query => 'クローラー').each do |item|
  puts item.uri
  puts item.title
end
```

◆ google-search-api.rbの実行例

```
$ ruby google-search-api.rb
http://ja.wikipedia.org/wiki/%E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9
クローラ - Wikipedia
http://e-words.jp/w/E382AFE383ADE383BCE383A9.html
クローラとは - IT用語辞典 e-Words
http://dic.nicovideo.jp/a/%E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9
```

```
クローラーとは (クローラーとは) [単語記事] - ニコニコ大百科
https://support.google.com/webmasters/answer/1061943?hl=ja
Google クローラ - ウェブマスター ツール ヘルプ

~省略~
```

5-1-3 Google Custom Search APIを利用する

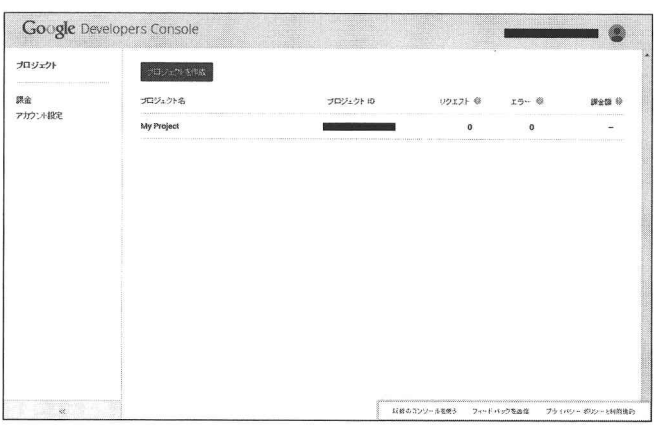
Googleは、いろいろなAPIを用意しています。検索に関するAPIの1つに、「Google Custom Search API」があります。無料で利用できるものの、1日あたりの使用制限があります。2014年7月現在では、1日200回の検索ができます。

APIキーの作成

GoogleのAPIを利用するには、Googleのアカウントを作成してログインのうえで、「Google Developers Console」でプロジェクトを作成する必要があります。そのうえで、Custom Search APIの利用設定と、API利用のためのキーの作成、検索エンジンIDの取得が必要です。

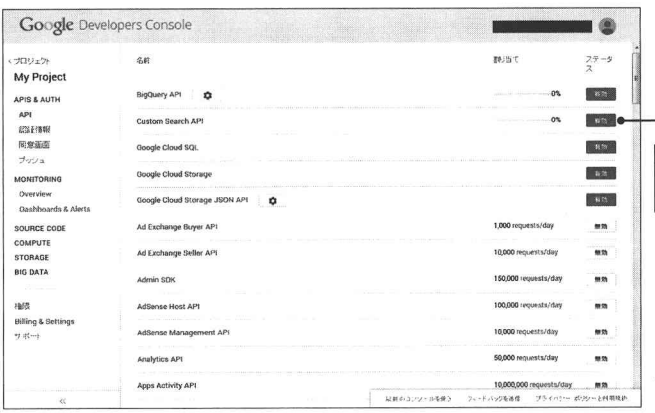
■ Google Developers Console
URL https://cloud.google.com/console/project

Google Developers Console



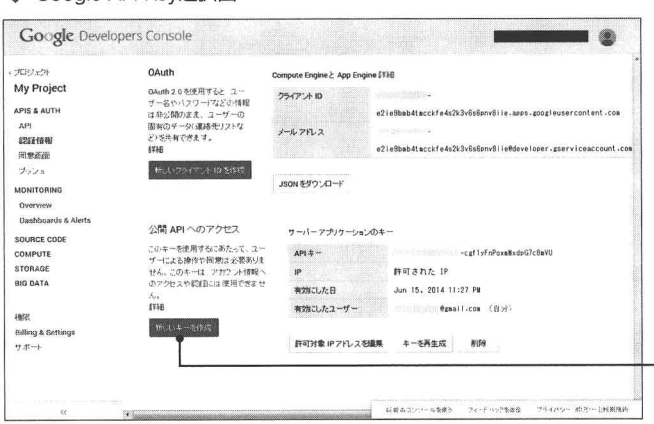
プロジェクト作成後にAPIのセッティング画面で「API」を選び、「Custom Search API」を有効にします。

Google APIセッティング画面

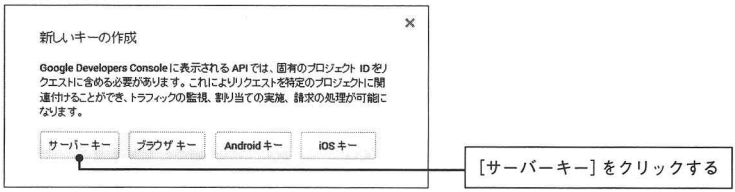


次に、「認証情報」を選択し、APIを利用するためのキーを作成します。「公開APIへのアクセス」の［新しいキーを作成］をクリックして作成します。キーの種類はいくつかありますが、［サーバーキー］を選択します。キー作成時に接続元IPの制限が可能ですが、特に必要はありません。

Google API Key選択画



Google API Key選択画面



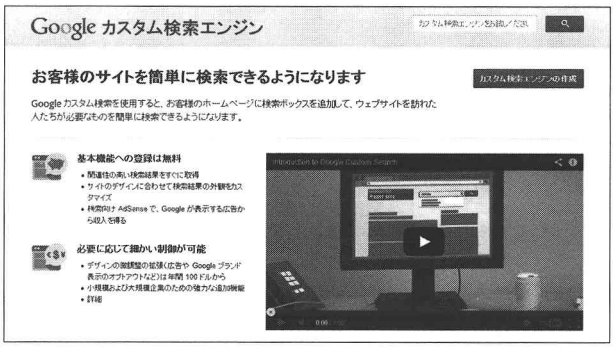
キーの作成が完了すると、「認証情報」画面にAPIキーが表示されます。APIキーはなくさないように保存しておきましょう。

カスタム検索エンジンの作成

最後に、カスタム検索エンジンを作成します。Googleの「カスタム検索エンジン」ページから作成します。[カスタム検索エンジンの作成]ボタンをクリックします。

■「カスタム検索エンジン」ページ
URL https://www.google.com/cse/

「カスタム検索エンジン」ページ



カスタム検索エンジン作成時に、検索対象ページを最低1つ以上指定する必要があります。作成後の設定変更で、それ以外のサイトも検索対象とするために、適当なURLを入れておけば問題ありません。

カスタム検索エンジンの作成ページ

作成後に検索エンジンの編集を行い、「検索するサイト」の項目で「追加したサイトを重視して、ウェブ全体を検索する」に変更します。そのうえで、[検索エンジンID] ボタンを押下して、IDを取得して保存します。

カスタム検索エンジンの編集ページ

APIを利用する

APIキーと検索エンジンIDが取得できたので、この2つを使いAPIを操作します。APIはHTTPS通信で行い、リクエストはURLで送り、結果はJSON形式で返ってきます。

実行にあたっては、「api_key」と「custom_search_engine_id」に、取得したAPIキーと検索エンジンIDを設定してください。

APIを使ってGoogleから検索結果を取得する google-api.rb

```
# -*- coding: utf-8 -*-
require 'json'
require 'uri'
require 'open-uri'

api_key = 'GOOGLE_API_KEY'
custom_search_engine_id = 'GOOGLE_ENGINE_ID'
search_word = URI.encode("クローラー")
url =
  "https://www.googleapis.com/customsearch/v1?key=
   #{api_key}&cx=#{custom_search_engine_id}&q=
   #{search_word}"

json = JSON.load(open(url))
json['items'].each do |item|
  puts item['title']
  puts item['link']
end
```

◆ google-api.rbの実行例

```
$ ruby google-api.rb
```

Google Custom Search APIのパラメータの詳細については、Google開発者ページを参照してください。

■ Google開発者ページ
URL https://developers.google.com/custom-search/?hl=ja

5-2 ブログへのクローリング

文章を解析して統計的なデータを取得する場合、そもそも元の文章をどこから取得するかというのは重大なテーマです。インターネット出現以前は、新聞のアーカイブデータがほぼ唯一のデータでした。現在は、目的に応じてニュースサイトやTwitter、ブログなどからデータを取得するという手法が取られています。ブログについては、適切な文章の長さと、比較的構造化されているために、クローリングしやすいというメリットがあります。また、タグやカテゴリというメタデータで、1つ1つの記事自体にある程度の意味づけがされています。そのため、特定カテゴリの文章を収集するという目的を達しやすくなっています。

ブログをクローリングする場合は、ブログのURLリストをどう収集するのかという点と、収集したブログから本文部分のみを抽出する方法が重要になります。それぞれ、いくつかの方法を紹介します。

5-2-1　個別ブログの記事取得

個別ブログの記事取得にはいくつかの方法があります。それぞれ一長一短あるので、目的に応じて使い分けることになります。代表的な方法として、次の3つをあげます。

- Anemoneなどで自動巡回する
- Sitemapsを参照する
- RSSから定期的に新着記事を取得する

1つ目の自動巡回について、ブログは比較的循環しやすい構造になっています。一方で、1つのブログエントリーが個別記事、カテゴリ別、月別などさまざまな切り口でまとめられているために、内容が重複する場合があります。例えば、ライブドアブログの「livedoor Blog 開発日誌」というブログでは、次のような構造になっています。

- トップページ　：http://blog.livedoor.jp/staff/
- 個別の記事　　：http://blog.livedoor.jp/staff/archives/ランダムな数字.html
- カテゴリ別記事：http://blog.livedoor.jp/staff/archives/cat_カテゴリ番号.html
- 月別記事　　：http://blog.livedoor.jp/staff/archives/YYYY-MM.html

個別の記事以外も収集すると、記事内容が重複してしまいます。また、「http://ブログドメイン/ブログ名/」という形式のため、何も指定しないで巡回すると、同じブログドメインの別ブログも収集してしまう可能性が高いです。そのため、巡回先を絞るなどの対応が必要となってきます。

巡回対象の絞り込み

下記のスクリプトは、Anemoneで「http://blog.livedoor.jp/staff/」の個別記事のみを処理する例です。巡回対象の絞り込みのみで、具体的なページの処理は記述していません。ブログのページ処理は、後ほど解説します。

巡回対象の絞り込み anemone-blog.rb

```
# -*- coding: utf-8 -*-
require 'anemone'
require 'open-uri'

urls = ["http://blog.livedoor.jp/staff/"]
opts = {
  :depth_limit => false,
  :delay => 1
}

Anemone.crawl(urls, opts) do |anemone|
  anemone.focus_crawl do |page|
    page.links.keep_if { |link|
      link.to_s.match(/blog.livedoor.jp¥/staff¥/archives¥/(¥d+)¥.html/)
    }
  end

  anemone.on_every_page do |page|
    # 処理対象とするURLの表示
    puts page.url
  end
end
```

◆ anemone-blog.rbの実行例

```
$ ruby anemone-blog.rb
http://blog.livedoor.jp/staff/
http://blog.livedoor.jp/staff/
http://blog.livedoor.jp/staff/archives/51856334.html
http://blog.livedoor.jp/staff/archives/51855648.html
http://blog.livedoor.jp/staff/archives/51854912.html

~省略~
```

スクリプトを実行すると、「http://blog.livedoor.jp/staff/」の個別記事の一覧(URL)を取得します。

Sitemapsの参照

次にSitemapsを参照する方法です。Sitemapsは、Webサイト側から検索エンジンなどに、サイト内でクロールすべきURLを通知するための規定です。XMLフォーマットで記述され、サイト直下に「sitemap.xml」という形で配置されています。またサイト内のページが多い場合は、トップの「sitemap」の下に複数のSitemapsを配置する階層構造になります。

次のスクリプトも、「livedoor Blog 開発日誌」のSitemapsから個別ページのURLを取得しています。URLのリストを作成できれば、別途データ取得と解析のプログラムの作成は容易になります。

SitemapsからURLを取得する

sitemap-parser.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'

def get_xml_doc(url)
  return REXML::Document.new(open(url))
end

base_url="http://blog.takuros.net/sitemap.xml"
base_url="http://blog.livedoor.jp/staff/sitemap.xml"
sitemaps = get_xml_doc(base_url)
sitemaps.elements.each('sitemapindex/sitemap/loc')do |element|
  sitemap = get_xml_doc(element.text)
  sitemap.elements.each('urlset/url/loc/')do |element|
    if /¥/staff¥/archives¥/(¥d+)¥.html/ =~ element.text
      # 巡回対象とするURLの表示
      puts element.text
    end
  end
end
```

◆ sitemap-parser.rbの実行例

```
$ ruby sitemap-parser.rb
http://blog.livedoor.jp/staff/archives/51856334.html
http://blog.livedoor.jp/staff/archives/51855648.html
```

```
http://blog.livedoor.jp/staff/archives/51854912.html
http://blog.livedoor.jp/staff/archives/51854614.html
http://blog.livedoor.jp/staff/archives/51854511.html
http://blog.livedoor.jp/staff/archives/51852661.html
http://blog.livedoor.jp/staff/archives/51851795.html
http://blog.livedoor.jp/staff/archives/51849248.html
http://blog.livedoor.jp/staff/archives/51848573.html
http://blog.livedoor.jp/staff/archives/51847851.html
```

Sitemapsが存在する場合は、巡回先の取得方法としては一番効率がよくなります。反面、ブログによってはSitemapsを備えていない場合もあるので、クローラーによる巡回と使い分けるのがよいでしょう。

ブログの新着記事の取得

先の2つスクリプトは、ブログ全体を対象として巡回する場合に有効な手段です。しかし、毎日クローラーを動かす場合は、差分のURLのみで十分です。そういった場合は、RSSを利用して巡回先を取得するといった方法が有効になります。

次のスクリプトは、RSSファイルを取得し、1日以内に追加されたエントリーがあれば、そのURLを表示します。追加エントリーがない場合は、何も表示されません。日数の調整をすることにより、その間の差分データを取得する仕組みを作れます。

RSSから巡回先を取得する

rss-parser.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'
require 'date'

url="http://blog.livedoor.jp/staff/index.rdf"

doc = REXML::Document.new(open(url))
doc.elements.each('rdf:RDF/item') do |item|
  dc_date = Date.parse(item.elements['dc:date'].text)

  # 1日以内に追加されたデータ、処理対象とする
  if (Date.today - dc_date).to_i <= 1
    # 巡回対象とするURLの表示
    puts item.elements['link'].text
  end
end
```

◆ rss-parser.rbの実行例

```
$ ruby rss-parser.rb
http://blog.livedoor.jp/staff/archives/51856334.html
```

なお、ブログの更新が前日からされていない場合は、結果出力はありません。

ブログサービスの全体の新着記事の取得

これまでのスクリプトは、個別ブログを対象に巡回先を取得する方法でした。では、不特定多数のブログから、記事を収集したい場合はどうするのでしょうか？

1つの方法としては、各ブログサービスの新着ページや新着RSSを利用する方法があります。RSSで配布されている場合は、個別ブログの新着記事の取得とまったく同じ方法で取得できます。

■ はてなダイアリー　日記一覧
URL http://d.hatena.ne.jp/diarylist?mode=rss

一方で、ブログサービスの提供者側が、新着記事のRSSの配布サービスを止める傾向にあります。

5-2-2　本文抽出

クロール対象のブログのURLが取得できれば、次は解析です。ブログの解析の場合は、本文のみを抽出して解析するといったケースが多いです。

方法は2種類あります。ブログごとに本文部分を指し示す部位を特定して抽出する方法と、個別ブログの構造にかかわらず汎用的に抽出する本文抽出モジュールを利用する方法です。抽出精度という点では、個別に作り込む前者の方が高くなります。一方で、複数のブログを抽出する場合は汎用的な仕組みが必要になります。

HTML構造による抽出

次のスクリプトは、はてなブログの記事から本文を抜き出している例です。対象となるブログは、スクリプト内で直接指定しています(❶)。

本文部分の特定は、FirefoxやChromeの開発者ツールを利用し、本文を選択している状態で要素のXPathを抜き出すと簡単に特定できます(→p.183)。ブログサービスごとにIDやクラス名は異なりますが、「content」などの名前であることが多いです。

ブログの本文を抽出する nokogiri-extract.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'

html = open(
  'http://blog.takuros.net/entry/20140104/1388788175').read ●―――❶
doc = Nokogiri::HTML(html)
puts doc.xpath("//div[@class='entry-content']").text
```

◆ nokogiri-extract.rbの実行例

```
$ ruby nokogiri-extract.rb

　　3年ほど前に、Ruby製のクローラー"anemone"を紹介しました。その当時から
完成度が高く、Rubyでクローラーを使う場合はanemoneを利用してきました。最近、他に
新しくて良いのがないか調べましたが、機能面の網羅性という意味でanemoneを超えるも
のは見つけられませんでした。そこで改めてanemoneのソースを読んでみたところ、クロ
ーラーが必要とする機能を必要最小限で実装され、やはり中々良い出来です。冬休みの宿
題ではないですが、勉強の意味を兼ねてソースを追っていくことにします。

~省略~
```

本文抽出モジュール

次にブログの本文抽出モジュールを利用した例です。今回紹介する「extractcontent」という本文抽出モジュールは、先ほどのHTMLの構造を見てルールベースで抽出するのではなく、自然言語処理を利用して本文らしい文章を計算して抽出しています。

■ Webページの本文抽出（nakatani @ cybozu labs）
URL http://labs.cybozu.co.jp/blog/nakatani/2007/09/web_1.html

残念ながらこのモジュールはRuby 1.8.5で実装されてるため、正規表現のエンジンが変更された関係でRuby 2.0ではエラーが発生します。別の人により、Ruby 2.0でも動作するように改良されたものが公開されています。そちらをダウンロードして、インストールすることが可能です。

■ GitHubのextractcontentページ
URL https://github.com/mono0x/extractcontent/

上記のページからZIPファイルをダウンロードし、任意のディレクトリに展開します。そのうえで、展開したディレクトリに移動し、gemでビルド・インストールします。なおビルドにはDEVELOPMENT KITが必要です(→p.76)。

◆ extractcontentのインストール

```
$ gem build extractcontent.gemspec
$ gem install extractcontent-0.0.1.gem
```

インストール時にWARNINGメッセージが出力されます。これは、ソース中にライセンスと説明表記がないための警告です。無視して先に進んでも問題ありません。

extractcontentは、HTMLを渡すとcontentとtitleの文字列を返します。内部的な処理としては、HTMLをブロックという単位で分離し、本文らしくないブロックに低いスコアを与え、スコアが高いブロックが並んでいる部分を本文として扱います。

スコアが低くなる要素としては、配置やアフィリエイトのリンク、フッターによく出るキーワードなどがあります。

■ extractcontentで本文を抽出する　extract.rb

```
# -*- coding: utf-8 -*-
require 'extractcontent'
require 'open-uri'

html = open(
  'http://blog.takuros.net/entry/20140104/1388788175').read
content, title = ExtractContent.analyse(html)
puts title
puts content
```

◆ extract.rbの実行例

```
$ ruby extract.rb

        3年ほど前に、Ruby製のクローラー"anemone"を紹介しました。その当時から
完成度が高く、Rubyでクローラーを使う場合はanemoneを利用してきました。最近、他に
新しくて良いのがないか調べましたが、機能面の網羅性という意味でanemoneを超えるも
のは見つけられませんでした。そこで改めてanemoneのソースを読んでみたところ、クロ
ーラーが必要とする機能を必要最小限で実装され、やはり中々良い出来です。冬休みの宿
題ではないですが、勉強の意味を兼ねてソースを追っていくことにします。

~省略~
```

5-3 Amazonのデータを取得する

Amazonの場合は「4-6 APIを利用した収集」(→p.242)で説明したとおり、データの取得にはクローリングではなくAPIを利用すべきです。API経由で取得できないデータに関してのみ、クローリングを併用して補完するのがよいでしょう。

APIで取得できないデータとしては、新着情報やランキング、セール情報などがあります。これらのページから商品IDのみを取得し、その後はAPI経由で商品データを取得しましょう。

5-3-1 商品ID「ASIN」

Amazon内での商品は、ASIN (Amazon Standard Identification Number)で識別されます。ASINは、英数字からなる10桁の番号です。本、CD、DVDなどカテゴリを問わずに、全体で一意の値となります。本の場合は、ASINはISBN (International Standard Book Number)となります。ISBNについては10桁もしくは13桁ですが、Amazon内では基本的には10桁のISBNを利用しているようです。

Amazon内で商品を扱う場合は、ASINが必要になります。HTMLやAPIからASINを収集し、APIを通じて商品を取得するのがクローリングの流れとなります。

5-3-2 商品IDの取得

Amazonで商品検索を行う場合は、APIを利用しましょう。画面から抜き出すメリットは、ほとんどないです。

次のスクリプトは、Amazon.co.jpから「本」カテゴリを対象に「クローラー」をキーワードにして検索した例です。検索結果のなかから、ASINを抜き出しています。

Amazonの検索結果から商品IDを取得する amazon-search.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'
require 'uri'

search_word = URI.escape("クローラー")
url=
  "http://www.amazon.co.jp/s/ref=nb_sb_noss?url=
   search-alias%3Dstripbooks&field-keywords=
   #{search_word}"
```

```
doc = Nokogiri::HTML(open(url))
doc.xpath("//h3[@class='newaps']/a").each {|item|
  # ASIN
  puts item[:href].match(%r{dp/(.+)})[1]
}
```

◆ amazon-search.rbの実行例

```
$ ruby amazon-search.rb
4873111870
4010550155
4873113024
4492973222
4844335359
4800710138

~省略~
```

次は同様の処理をAPIで行った例です。HTMLの変更の影響を受けないので、安定して使えます。実行の際には、「:associate_tag」「:AWS_access_key_id」「:AWS_secret_key」を設定してください(→p.243)。

■ APIでAmazonの商品IDを取得する　amazon-api-search.rb

```
# -*- coding: utf-8 -*-
require 'amazon/ecs'

Amazon::Ecs.options = {
  :associate_tag => 'sampleapp-22',
  :AWS_access_key_id => 'AWS_ACCESS_KEY'
  :AWS_secret_key => 'AWS_SECRET_ACCESS_KEY'
}

# 商品検索
res = Amazon::Ecs.item_search(
  'クローラー', :search_index => 'Books', :country => 'jp')
res.items.each do |item|
  puts item.get('ASIN')
end
```

◆ amazon-api-search.rbの実行例

```
$ ruby amazon-api-search.rb
4873111870
4010550155
477791206X
4873113024
4938335107

~省略~
```

5-3-3 商品データの取得

商品データを取得する際も、APIを利用することを強く勧めます。下記のスクリプトは、商品詳細ページから、商品ID「4873111870」の商品の、書籍名・価格・画像イメージのURLを抜き出しています。

Amazonの商品データを取得する amazon-detail.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'

asin = "4873111870"
url="http://www.amazon.co.jp/dp/#{asin}"

doc = Nokogiri::HTML(open(url))

# 書籍名
puts doc.xpath("//span[@id='btAsinTitle']").text

# 価格
puts doc.xpath("//span[@id='actualPriceValue']/b").text

# 画像イメージのURL
puts doc.xpath("//img[@id='prodImage']").attribute("src").text
```

◆ amazon-detail.rbの実行例

```
$ ruby amazon-detail.rb
Spidering hacks―ウェブ情報ラクラク取得テクニック101選 [単行本]
￥3,780

http://ecx.images-amazon.com/images/I/41AT4JG2KQL._BO2,204,203,200_PIsitb-sticker-arrow-
click,TopRight,35,-76_AA240_SH20_OU09_.jpg
```

次は同様の処理をAPIで行った場合のスクリプトです。Amazon::ECSモジュールでは、item_lookupメソッドでASIN指定でのデータ取得ができます。XMLが返されるので、任意の要素を取得します。実行の際には、「:associate_tag」「:AWS_access_key_id」「:AWS_secret_key」を設定してください（→p.243）。

APIで商品データを取得する　　amazon-api-detail.rb

```
# -*- coding: utf-8 -*-
require 'amazon/ecs'

Amazon::Ecs.options = {
  :associate_tag => 'sampleapp-22',
  :AWS_access_key_id => 'AWS_ACCESS_KEY',
  :AWS_secret_key => 'AWS_SECRET_ACCESS_KEY'
}

# 商品検索
res = Amazon::Ecs.item_lookup(
  '4873111870', :response_group =>
  'Small, ItemAttributes, Images', :country => 'jp')

# 書籍名
puts res.items.first.get("ItemAttributes/Title")

# 価格
puts res.items.first.get("//FormattedPrice")

# 画像イメージのURL
puts res.items.first.get("MediumImage/URL")
```

◆ amazon-api-detail.rbの実行例

```
$ ruby amazon-api-detail.rb
Spidering hacks―ウェブ情報ラクラク取得テクニック101選
￥ 3,780
http://ecx.images-amazon.com/images/I/41AT4JG2KQL._SL160_.jpg
```

5-3-4　新着・ランキング・セール

新着リストやランキング・セール情報は、残念ながらAPIから取得できません。HTMLを解析して、ASINを取得します。抜き出し方は、「2-5-3　スクレイピング機能の作成」（→P.91）で説明したとおりです。

■ Amazon.co.jp 新着ニューリリース
URL http://www.amazon.co.jp/gp/new-releases

■ Amazon.co.jp 新着ニューリリース: 本
URL http://www.amazon.co.jp/gp/new-releases/books/

■ Amazon.co.jp ベストセラー
URL http://www.amazon.co.jp/gp/bestsellers/

■ Amazon.co.jp: タイムセール
URL http://www.amazon.co.jp/%E3%82%BF%E3%82%A4%E3%83%A0%E3%82%BB%E3%83%BC%E3%83%AB/

5-4 Twitterのデータ収集

Twitterに流れるTweetのデータとしての強みは、即時性と属性情報です。まず即時性については、一度の投稿が140文字という制約のために、Webサイトやブログのようにまとめて書くという使われ方ではなく、今起きていることをすぐに書くというような利用のされ方が多いです。そのため、即時性の高いデータとして活用できます。

属性情報には、1つ1つのTweetにはユーザー名とつぶやきだけではなく、さまざまな付随情報が付いています。例えば、発言した人の情報に始まり、そのTweetがどれくらいお気に入り登録やRetweetされたかなどです。また、ユーザーの設定によっては位置情報も付いてきます。それ以外にも、ハッシュタグといったTweetを分類するための、ゆるい仕組みもあります。

Twitterのデータを上手く収集することで、今までできなかった分析が可能になります。Twitterのデータ収集の方法としては、HTMLからのクローリングと、APIからの収集の2つがあります。Twitterの場合もAPIを使うべきです。HTMLのクローリングでは、収集できるデータも限定され、かつAPIに比べて手間が多いです。本書では両方とも紹介しますが、APIを使うことをお勧めします。

5-4-1 HTMLからのクローリング

まずはHTMLからのクローリングです。Twitterのデータの収集の仕方としては、自分もしくは他人のタイムラインを取得する場合と、検索結果を取得する場合の2つのパターンがあります。

取得対象の指定

タイムラインの取得の場合、取得対象は、

- https://twitter.com/ユーザー ID

という形になります。例えば、日本語版Twitter公式アカウントの場合は、次のURLから取得できます。

- https://twitter.com/TwitterJP

次に検索ですが、これはいくつかオプションがあります。まず基本的な形ですが、

- search?q=検索語句&src=typed

でトップTweetを取得できます。トップTweetとは、Twitterによってフィルタリングされて、重要度が高いもののみ表示される形式です。検索語句については、URIエンコードされている必要があります。「クローラー」で検索した場合は、次のような形になります。URIエンコードについては、p.248をご参照ください。

- https://twitter.com/search?q=
%E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9%E3%83%BC&src=typd

トップTweetではなく、すべての検索結果を出したい場合は、「f=realtime」というオプションを付加します。

- https://twitter.com/search?f=realtime&q=
%E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9%E3%83%BC&src=typd

ハッシュタグの検索も、基本的には同じです。ハッシュタグ「#」もURIエンコードして「%23」に変換する必要があります。

- https://twitter.com/search?f=realtime&q=%23jawsug&src=typd

また、ユーザー名「mode=users」、画像「mode=photos」、動画「mode=videos」、ニュース「mode=news」などの検索オプションを付けることにより、検索結果を絞り込むことができます。次のURLは、画像検索の例です。

- https://twitter.com/search?q=%E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9%E3%83%BC&src=typd&mode=photos

タイムラインデータの取得

TwitterのHTMLは、タイムラインと検索結果の画面で見た目は同じですが、HTMLの構造は違っています。まず、タイムラインですが、1つ1つのTweetは、「data-component-term="tweet"」という属性を持つ<Div>タグのなかに格納されています。詳細を取得する場合は、まずはその<Div>タグをまとめて切り出すのがよいでしょう。

次のスクリプトは、日本語版Twitter公式アカウントのタイムラインから、Tweet時間、Tweet本文、Retweet数、お気に入り登録された数を取得しています。

Twitterのタイムラインからデータを取得する twitter-timeline-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri.HTML(open('https://twitter.com/TwitterJP'))
doc.xpath("//div[@data-component-term='tweet']").each { |tweet|
  # Tweet時間
  puts Time.at(tweet.xpath(
    ".//a[@class='ProfileTweet-timestamp js-permalink js-nav
     js-tooltip']/span").first['data-time'].to_i) ●―――――❶

  # Tweet本文
  puts tweet.xpath(
    ".//p[@class='ProfileTweet-text js-tweet-text u-dir']").text

  # Retweet数
  retweet = tweet.xpath(
    ".//li[@class='ProfileTweet-action ProfileTweet-action--retweet
     js-toggle-state js-toggle-rt']/button[@class=
     'ProfileTweet-actionCount js-actionCount
      js-tooltip']")
  if !retweet.empty?
    puts "Retweetされた数 : " +
      retweet[0]['data-tweet-stat-count'] ●―――――❷
  end

  # お気に入りされた数
  like = tweet.xpath(
```

```
    ".//li[@class='ProfileTweet-action ProfileTweet-action--favorite
     js-toggle-state']/button[@class=
     'ProfileTweet-actionCount js-actionCount js-tooltip']")
  if !like.empty?
    puts "お気に入りされた数 : " +
      like[0]['data-tweet-stat-count'] ●―――――❷
  end
}
```

◆ twitter-timeline-nokogiri.rbの実行例

```
$ ruby twitter-timeline-nokogiri.rb
2014-07-18 11:59:46 +0900
1週間でもっとも多くリツイートされたツイートからのご紹介です。今週は話題のテーマ
パーク施設オープンに関するツイートが盛り上がりました。この夏はたくさんの魔法のツ
イートをお待ちしています。 https://blog.twitter.com/ja/2014/0718rt?
Retweetされた数 : 25
お気に入りされた数 : 21
2014-07-17 14:00:28 +0900
1つのツイートに4枚まで写真を付けられるの、ご存知ですよね？お友達やご家族みんなで
出かけることが多くなりそうな夏、ぜひご活用ください。

https://blog.twitter.com/ja/2014/0717howto?…
Retweetされた数 : 105
お気に入りされた数 : 102
2014-07-15 10:51:56 +0900

~省略~
```

Tweetされた時間については、「ProfileTweet-timestamp js-permalink js-nav js-tooltip」というClass名の<A>タグ以下の<Span>タグのなかに、「data-time」という属性値で埋め込まれています(❶)。

直接<Span>タグを取ろうとすると1日以内の発言かどうかでClass名が変わるために、1つ上のタグから指定する方が簡単です。埋め込まれた値は、「1400201764」といった形のUnixTime形式です。表示する際は、Timeクラスなどを利用して変換が必要です。

また、Retweet数やお気に入りされた数は、<Li>タグの下の<Button>タグのなかの、「data-tweet-stat-count」という属性値として埋め込まれています(❷)。Retweetやお気に入り登録されているかにより、HTMLのタグが変わってきます。また、

Twitterの表示形式の変更に伴い、影響を受ける可能性は極めて高いでしょう。

なお、Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

検索結果の取得

次に検索結果を取得する例です。検索結果には、Retweetやお気に入りの情報は付加されていません。それらを取得する場合は、「開く」のリンクから再度HTMLを取得することになります。TweetごとにHTTPリクエストが発生するため、非常に効率が悪いです。後で説明する、REST APIを利用しましょう。

検索結果を取得する twitter-query-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

doc = Nokogiri.HTML(open(
  'https://twitter.com/search?f=realtime&q=
   %E3%82%AF%E3%83%AD%E3%83%BC%E3%83%A9%E3%83%BC&src=typd'))

doc.xpath("//li[@data-item-type='tweet']").each { |tweet|
  # Tweet時間
  puts Time.at(tweet.xpath(
    ".//a[@class='tweet-timestamp js-permalink js-nav js-tooltip']
     /span").first['data-time'].to_i)

  # Tweet本文
  puts tweet.xpath(".//p[@class='js-tweet-text tweet-text']").text
}
```

◆ twitter-query-nokogiri.rbの実行例

```
$ ruby twitter-query-nokogiri.rb
2014-07-18 09:47:04 +0900
htmlのソースが複雑になれば、その分クローラーが重要なキーワードを見落とす可能性が高まります。
もしスタイルシートがhtml内に直書きされているのであれば、外部ファイルに移動させるだけでグッと
SEO効果が上がる場合があります。一度確認してみてください。 #見落とす

~省略~
```

Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

以上が、HTMLを解析してTweet情報を取得する方法になります。これ以外の機能として、再帰的にデータを取得する方法などがありますが、そのためにはTwitterの遅延ロードの仕組みを解析したうえで処理を書く必要があり、非常に難解になります。また、一度作成したとしてもTwitterのHTMLの画面の変更によって、正常に動作しなくなる恐れがあります。上記のことを踏まえて、APIを利用しましょう。

5-4-2　TwitterのAPI

Twitterには、主に2種類のAPIが用意されています。「REST API」と「Streaming API」です。REST APIは、Tweetの取得や投稿、ユーザー操作に関するAPIです。Twitterのサードパーティ製のアプリのほぼすべてが、このTwitter REST APIを利用して作られています。画面で操作するのとほぼ同等の機能があります。

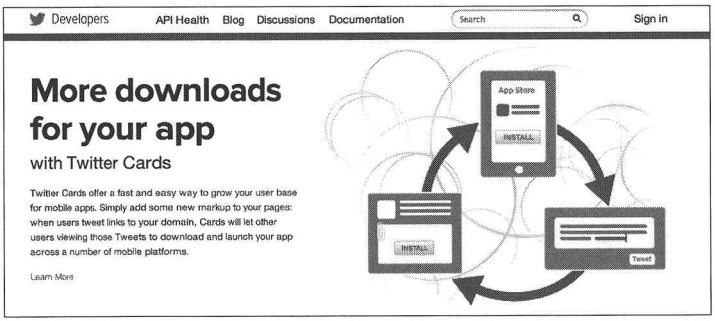
これに対してStreaming APIは、ユーザーもしくはパブリックタイムラインを取得するだけのAPIです。REST APIに比べて機能が大幅に少ないように思えますが、Streaming APIならではの特徴があります。一度接続するとTwitter側からデータが流し込まれるために、大量のデータを効率よく取得するというメリットがあります。用途の使い分けとしては、検索などの場合はREST APIを使用し、Twitterに流れるデータを何でも取得して解析したい場合はStreaming APIを利用するようにします。

5-4-3　Twitter REST API

REST APIを利用するには、まずTwitterの開発者ページでアプリケーション登録して、APIを利用するためのキーを取得します。開発者画面には、TwitterのIDとパスワードでログインできます。

■ Twitter開発者ページ
URL https://dev.twitter.com/

Twitter Developerページ

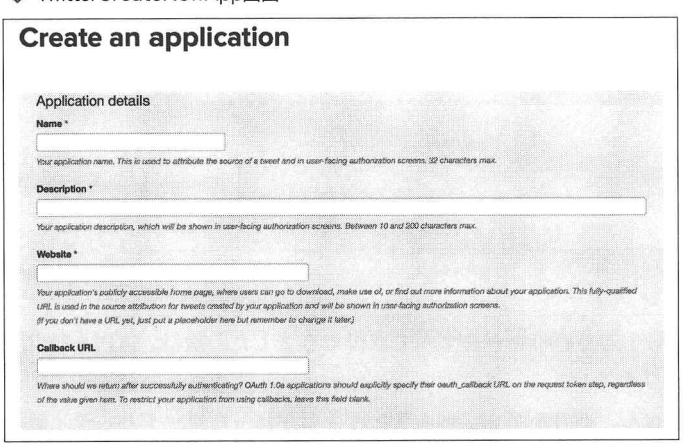


APIキーの取得

アプリケーションの登録は、右上の自分のアイコンから、[My Applications]を選び[CreateNewApp]ボタンをクリックして行います。

登録には、アプリケーション名と概要、WebサイトのURLの登録が必須になります。また任意で「Callback URL」を登録します。WebサイトのURLについては、必須となっているので適当なURLを選んで登録します。Callback URLについては、Webシステムから利用する場合は必須ですが、今回のケースでは必要ないです。

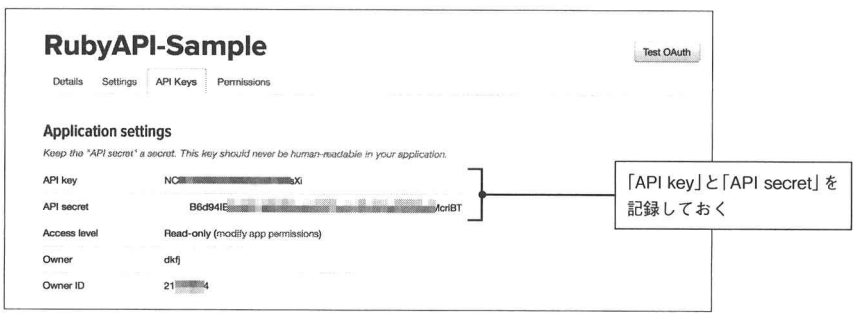
TwitterCreateNewApp画面



アプリケーションの登録の後は、APIキーを取得します。「My Applications」の一覧画面から作成したアプリのリンクを押下します。アプリケーションの詳細ペー

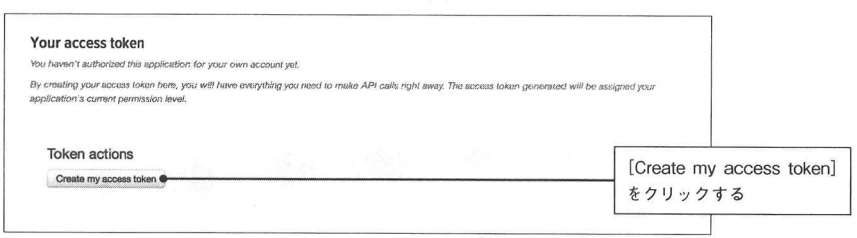
ジで、「API Keys」のタブを押すとAPIが表示されます。ここで、「API key」と「API secret」を取得できます。また「Access level」などの変更により、参照以外に更新ができるようになります。今回は、参照権限で十分です。不必要に高いレベルの権限は、与えないようにしましょう。

▼ TwitterアプリケーションのAPI Keyの確認画面



APIの利用には、さらに「Access token」と「Access token secret」が必要になります。API Keysのページを下にスクロールして、「Your access token」の「Token actions」から［Create my access token］ボタンを押下してトークンを生成します。

▼ TwitterアプリケーションのAccess Tokenの取得



ライブラリのインストール

これでAPIを利用するためのキーをすべて取得できました。次は、APIを利用するためのライブラリのインストールです。Twitterは、各言語ごとの公式ライブラリは提供していません。そのかわり、個人または企業が作ったサードパーティ製のライブラリが多数存在し、公式ページで紹介されています。このなかから用途にあったものを利用しましょう。

■ Twitter Libraries
URL https://dev.twitter.com/docs/twitter-libraries

今回は、@sferikによりメンテナンスされているライブラリを利用します。その名も「twitter」です。

■ GitHubのtwitterページ
URL https://github.com/sferik/twitter

インストール時にビルドが必要なため、開発者キットが必要になります。Windowsの場合は、Ruby Installerにある「DEVELOPMENT KIT」という開発ツールやMicrosoftのVisual Studioなどが利用できます。比較的手軽という点では、DEVELOPMENT KITがお勧めです。DEVELOPMENT KITのインストールはp.76をご参照ください。

インストールを開始すると、数多くのGemが付随的にインストールされます。完了後、gem list twitterでLOCAL GEMSのなかに表示されたら成功です。

◆ twitterのインストール

```
$ gem install twitter
```

◆ インストールの確認

```
$ gem list twitter
*** LOCAL GEMS ***

twitter (5.9.0)
```

APIの利用

APIの利用は準備までが大変です。しかし、その後はHTMLの解析に比べて簡単です。次のスクリプトは、REST APIを使用する例です。実行にあたっては、「:consumer_key」「:consumer_secret」「:access_token」「:access_token_secret」に、各自が取得したAPIキーなどを設定してください。

REST APIでTwitterのデータを取得する

twitter-timeline-api.rb

```
# -*- coding: utf-8 -*-
require 'twitter'
```

```
config = {
  :consumer_key => 'TWITTER_API_KEY',
  :consumer_secret => 'TWITTER_API_SECRET',
  :access_token => 'TWITTER_ACCESS_TOKEN',
  :access_token_secret => 'TWITTER_ACCESS_TOKEN_SECRET'
}
client = Twitter::REST::Client.new(config)
client.user_timeline('dkfj').each {|tweet|
  # Tweet時間
  puts tweet.created_at

  # Tweet本文
  puts tweet.text

  # Retweet数
  puts "Retweetされた数 : " + tweet.retweet_count.to_s

  # お気に入りされた数
  puts "お気に入りされた数 : " + tweet.favorite_count.to_s

  # 位置情報
  puts "位置情報 : " + tweet.geo if !tweet.geo.nil?
}
```

◆ twitter-timeline-api.rbの実行例

```
$ ruby twitter-timeline-api.rb
2014-07-18 12:26:09 +0900
今日のKindle日替わりセールは、「マーケティングの基本　この1冊ですべてわかる」。コトラーとか読むのしんどいので、試しに読んでみる http://t.co/ogcKxHLDrd
Retweetされた数 : 0
お気に入りされた数 : 0
2014-07-17 12:44:44 +0900
展示ブースの隅っこで、JAWSUGのブースで立ってます。ステッカー配ってます。暇なので、遊びに来てください #aws #AWSSummit #jawsug http://t.co/WwFdgXndWy
Retweetされた数 : 8
お気に入りされた数 : 3
2014-07-17 10:29:54 +0900
AWS Summitの会場についた。予想してたけど、凄い人だ。
Retweetされた数 : 0
お気に入りされた数 : 0

~省略~
```

APIが返す値の詳細については、Twitter公式のページで確認できます。TweetやRetweetなどの情報の他にも、位置情報やつぶやいたユーザーの属性情報なども細かく取得できます。

■ GET statuses/user_timelineページ
URL https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline

5-4-4 Twitter Streaming API

twitterライブラリを利用すれば、Streaming APIも簡単に使えます。Streaming APIは、他のAPIと比べて少し特殊で、起動するとプログラム終了までTweetが流れ続けてきます。今回利用するAPIは、Streaming APIのなかでも「Sample」と呼ばれるものです。これは、全Tweetのうちでフィルタリングされた数%だけが提供されます。それでも、膨大な量のTweetが流れてきます。スクリプトを実行すると、いかにTwitterが大量のデータを扱っているか実感できるでしょう。

実行にあたっては、「:consumer_key」「:consumer_secret」「:access_token」「:access_token_secret」に、各自が取得したAPIキーなどを設定してください。

Twitter Streaming APIでTweetを取得する twitter-streaming-api.rb

```
# -*- coding: utf-8 -*-
require 'twitter'

config = {
  :consumer_key => 'TWITTER_API_KEY',
  :consumer_secret => 'TWITTER_API_SECRET',
  :access_token => 'TWITTER_ACCESS_TOKEN',
  :access_token_secret => 'TWITTER_ACCESS_TOKEN_SECRET'
}
client = Twitter::Streaming::Client.new(config)
client.sample do |tweet|
  if tweet.is_a?(Twitter::Tweet)
    # 日本語のつぶやきだけ表示
    puts tweet.text if tweet.lang == "ja"
  end
end
```

♥ twitter-streaming-api.rbの実行例

```
$ ruby twitter-streaming-api.rb

富山やばすぎ
戦いを挑むペコ可愛い愛しい
カブトムシ似合ってます！
写真がカブトムシでホッとしました。
ツノがなくて、平べったかったらと思うとゾゾっとしますΣ(ﾟдﾟlll)

~省略~
```

このTweetオブジェクトに含まれている項目については、公式ページを参照してください。idやuser名、つぶやきの他にもさまざまな情報が埋め込まれています。

■ Tweetsページ
URL https://dev.twitter.com/docs/platform-objects/tweets

5-5 Facebookへのクローリング

クローラーという観点から見ると、FacebookはTwitterに比べるとクローズドなソーシャルネットワークサービスです。ログイン状態でないと、パブリックに公開されたタイムラインすら取得できません。収集できるデータも、あくまで自分を中心とした知人との繋がりが前提になります。一方で、認証APIを使わなくても、「いいね」やコメント数などは取得できます。これらの情報は、人の関心を示す行動「アテンション」の重要な指標となります。Facebookをデータとして活用する場合は、アテンションを中心に取り扱うとよいでしょう。

クローラーを作成するうえでの注意点として、Facebookは純粋なクローラーに対してはほとんど扉を閉ざしています。ブラウザなどでログインをしたうえで認証情報を利用するか、Capybaraなどのブラウザを操作するクローラーを利用することで、データを収集することも可能です。しかし、FacebookはAPIが充実しているため、わざわざクローラーを作成することにメリットはほとんどありません。このセクションでは、APIの利用を中心に解説します。

5-5-1 Facebook Graph APIとFQL

Facebook Platformとして、いくつかのAPIが提供されています。2014年7月現在

のバージョンはv2.0で、データ収集系のAPIは「Graph API」と「FQL」があります。

FQLについては、このv2.0をもってサービス提供終了となります。v2.0の提供期間は2016年までとなっています。Facebookとしては、FQLからGraph APIへの移行を勧めている状態なので、新規で作成する場合はGraph APIを選ぶべきでしょう。

5-5-2 認証が必要ないFacebook Graph API

Graph APIについては、認証が必要なものと不要なものがあります。ほとんどのものが認証が必要であり、写真やチェックインなど投稿者との関係で取得できるかどうかの判断が入るものです。認証不要で取得できるものとしては、「いいね」の数などURLに対する評価です。まず、この値を取得する方法から見てみましょう。

「いいね」の取得は、「http://graph.facebook.com/」の後に目的とするURLを組み合わせてHTTPリクエストします。ブラウザとスクリプト、どちらから取得しても大丈夫です。

- http://graph.facebook.com/http://www.apple.com/jp/

レスポンスはJSON形式で返ってきます。idは対象のURL、sharesは「いいね」の数で、commentsはコメントされた数です。「いいね」などされていない場合は、項目として出てきません。

「いいね」の取得結果

```
{
   "id": "http://www.apple.com/jp/",
   "shares": 52132,
   "comments": 1
}
```

「5-2 ブログへのクローリング」(←p.256)で作成したブログのSitemapsからURLを抜き出すスクリプトと組み合わせると、ブログ内で注目を集めた記事がわかります。

ブログ内の記事の「いいね」の数を取得する facebook-sitemap-like.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'
require 'json'

def get_xml_doc(url)
  return REXML::Document.new(open(url))
end
```

```
base_url="http://blog.takuros.net/sitemap.xml"
sitemaps = get_xml_doc(base_url)
sitemaps.elements.each('sitemapindex/sitemap/loc') do |element|

  sitemap = get_xml_doc(element.text)
  sitemap.elements.each('urlset/url/loc/')
    do |element|
    response = open(
      "http://graph.facebook.com/#{element.text}").read
    json = JSON.parse(response)
    puts json['id']  # => URLの表示
    puts "いいね:"+json['shares'].to_s if json.key?('shares')
  end
end
```

◆ facebook-sitemap-like.rbの実行例

```
$ ruby facebook-sitemap-like.rb
http://blog.takuros.net/
http://blog.takuros.net/about
http://blog.takuros.net/entry/2014/06/27/195514
http://blog.takuros.net/entry/2014/06/26/125108
いいね:1
~省略~
```

5-5-3　認証が必要なFacebook Graph API

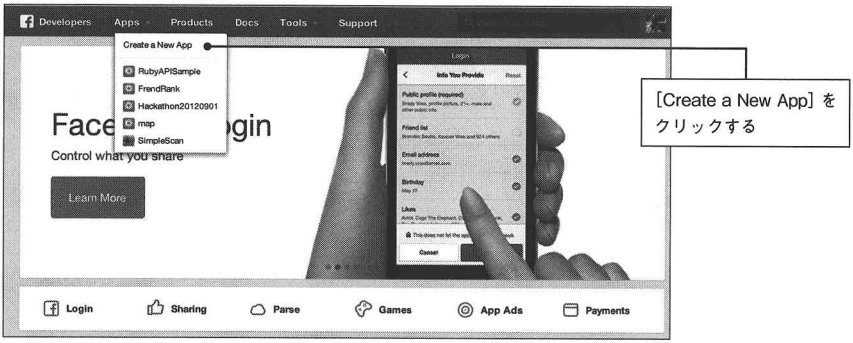
前述のとおり、Facebookから取得できるデータの多くは、自分との関係のなかから得られます。そのために、まず自分が何者かということを証明する必要があり、認証が必要です。認証のためには、開発者登録とアプリケーション登録が必要になります。開発者登録は、初回の1回だけ必要になります。アプリケーション登録は、新しいアプリケーションの登録の度に必要になります。

開発者登録

Facebookの開発者ページから、開発者登録していない状態で［Apps］ボタンを押し、［Create a New App］を押下すると、登録のナビゲーションが始まります。登録するには、クレジットカードの登録もしくは携帯電話でのSMSを使っての認証が必要になります。ここでは、SMSを使っての認証の仕方を説明します。

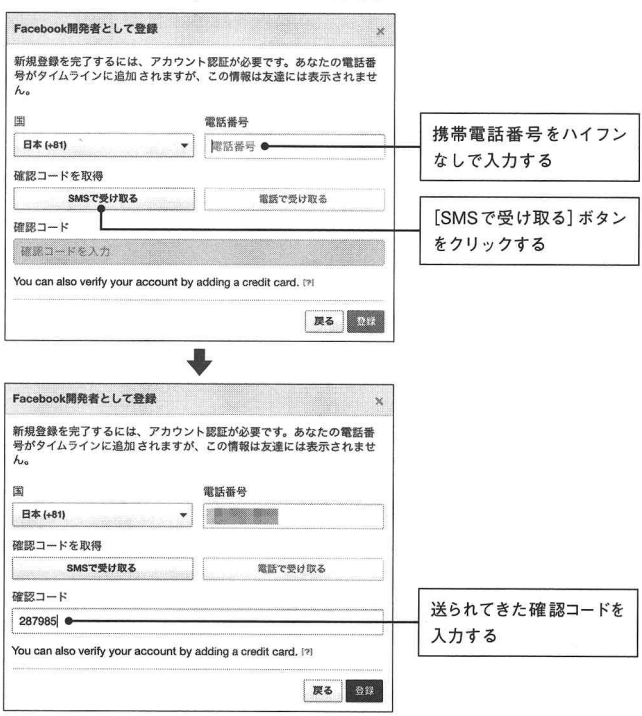
■ Facebook Developer画面
URL https://developers.facebook.com/

FacebookDeveloper画面



国番号に日本(+81)がされていることを確認し、携帯の電話番号をハイフンなしで続けて登録します。そして、[SMSで受け取る]ボタンを押下します。そうすると、1分以内にFacebookからショートメッセージが届きます。そこに6桁の数字が記載されているので、登録画面に入力します。

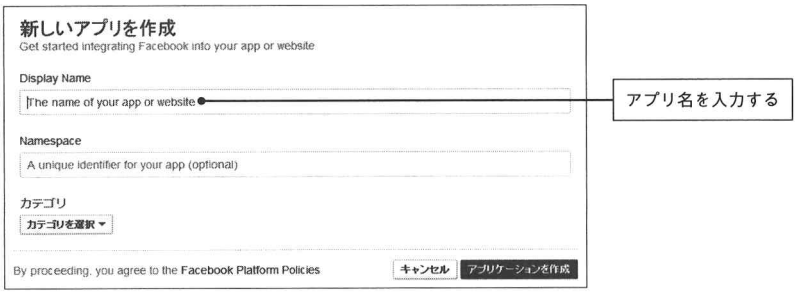
FacebookDeveloper　開発者登録画面



アプリケーションのキーの取得

認証用のキーの取得には、Facebookの開発者ページからアプリケーションの登録が必要です。開発者ページから［Apps］ボタンを押し、［Create a New App］を押下します。アプリ名を入力のうえでカテゴリを選んで、アプリケーションを作成します。アプリケーションの登録には、Facebookへのログインが必要です。

新しいアプリを作成

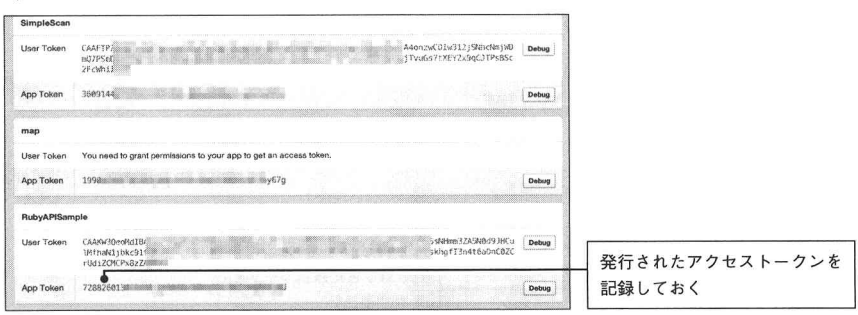


アプリケーション作成後は、アクセストークンを取得します。アクセストークンは、「Access Token」画面から取得できます。FacebookDeveloper画面上のツールバーから［Tools］を押して移動します。

アプリケーションを作成しただけでは、ユーザー向けのアクセストークンは発行されません。「Access Token」画面で「You need to grant permissions」のリンクを押して、権限を付与します。

■ Facebook Access Token画面
URL **https://developers.facebook.com/tools/access_token**

Facebook Access Token画面



なお、発行されたアクセストークンの有効期限は、1時間程度です。さらに権限を付与することにより、最大60日まで延ばすことは可能です。

ライブラリのインストール

RubyからFacebook Graph APIを扱うライブラリはいくつかあります。代表的なのは「FbGraph」です。それ以外にも「Koala」という軽量のラッパーライブラリがあります。今回は、Koalaを使うことにします。

■ GitHubのKoalaページ
URL https://github.com/arsduo/koala

Koalaのインストールは、gemで行えます。「Successfully installed koala-xxx」と表示されれば成功です(xxx部分はバージョン番号)。

◆ Koalaのインストール

```
$ gem install koala
```

APIの利用

データとしてのFacebookは、扱いが難しい部分があります。Twitterのように、全ユーザーからサンプリングしてデータを収集するということはできません。タイムラインの情報は、あくまで自分というバイアスがかかっています。Facebook中のパブリック情報から検索してデータを抽出するという方法もありますが、単純な検索だけではFacebookからデータを抽出しているという意味は少ないです。

そこで、特定のキーワードに対して検索し、その発言内容と発言主の属性情報を取得するというケースで考えてみましょう。Koalaは、Graph APIの軽量のラッパープログラムです。APIのメソッドを知らなくても、Graph APIのURLを知っていればそのまま使えます。

Graph APIのURLとは、「https://graph.facebook.com/」の後ろにリクエストURIを付けたものになります。この後ろに付ける部分をスクリプト内で指定することで、目的の情報が取得できます。

また、Graph APIのURLによるデータの取得結果は、「Graph API Explorer」で確認することができます。

■ Graph API Explorerページ
URL https://developers.facebook.com/tools/explorer/

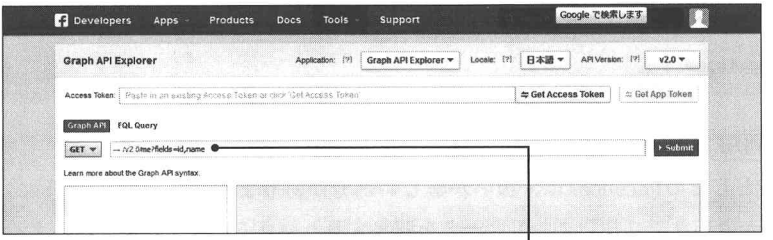
例えば、Facebookの自分の投稿を5件取得する場合は、次のようなクエリーを発行します。

• me?fields=posts.limit(5)

取得対象に対して、どういった情報を何件取るか指定しています。クエリーの書き方についてもっと知りたい場合は、公式ページのヘルプを参照してください。

■ Graph API公式ページ
URL https://developers.facebook.com/docs/graph-api/using-graph-api/v2.0

Graph API Explorerページ



確認するGraph APIのURLを入力する。クエリーも入力できる

発言内容と発言主の属性情報を取得する　facebook-graph-api.rb

```
# -*- coding: utf-8 -*-
require 'koala'

graph = Koala::Facebook::API.new('FACEBOOK_ACCESS_TOKEN')

# BRAVIAについて言及している発言を取得
search = graph.search('BRAVIA')
search.each {|result|
  puts result['message']
  # 発言主の情報を取得
  who = graph.get_object(result['from']['id'].to_s)
  puts "性別:" + who['gender'].to_s
  puts "生年月日:" + who['birthday'].to_s
}
```

◆ facebook-graph-api.rbの実行例

```
$ ruby facebook-graph-api.rb
Sony Bravia 32" LCD tv flat screen 1 and a half years old. worth $400 asking $300 firm need money
to buy a car to get to work. Also selling an xbox one for $400 2 controllers call of duty ghosts watch
dogs and forza 5. will sell tv and xbox as a package for $650 Insane deal
性別:male
生年月日:
Sony bravia 42" hdmi usb $4000
性別:female
生年月日:

~省略~
```

スクリプトを実行する際には、ご自分のアクセストークンを設定してください。

このスクリプトでは、発言内容と属性を抜き出すのみに留まっています。しかし、自然言語処理と組み合わせて、発言内容が好意的か否定的かの判別をして、属性ごとの統計を取ることにより、例えば商品に対する反応を自動的に集めることも可能です。データを収集する際は、目的と収集先の特徴を考慮すると、有効な分析となるでしょう。

5-6 画像を収集する

クローラーを作る目的として、画像を収集させたいという人も多いでしょう。画像収集の場合も、最終的な目的となるデータを抜き出すという点では同じです。画像を指し示すURLを抜き出したうえで、ファイルの保存機能を実装すれば目的のクローラーは作成できます。

5-6-1 Flickrからクローリングで収集する

Flickrなどの画像サービスをクローリングする際は、キーワード検索やタグ検索で画像の一覧を取得します。その次に、個々の画像のURLを取得します。

画像サービスの場合だと、サムネイル画像のようにサイズの小さな画像や、大きなサイズ、オリジナルのサイズなど複数の画像サイズが用意されている場合が多いです。目的に沿った画像サイズを集めるにはどうすればよいのか考える必要があります。

■ Flickr
URL https://www.flickr.com/

一般的には、画像一覧で出てくるのは小さな画像が多いです。ただし、画像の名前付けルールから、1つの画像がわかればHTML中にない場合でも推測できることが多いです。Flickrの画像ファイル名は、「ファイルID＋ランダムな英数字（＋サイズ符号）.jpg」と決まっています。1つのファイル名がわかると、サイズ符号を変えることですべてのサイズを取得できます。例えば、

• https://farm3.static.flickr.com/2485/3839885270_6fb8b54e06.jpg

というファイルの元画像サイズは、「_z」を付けた、

• https://farm3.static.flickr.com/2485/3839885270_6fb8b54e06_z.jpg

になります。ファイル名部分を「url」とすると、「url_m」「url_s」「url_t」「url_b」「url_z」「url_q」「url_n」「url_c」「url_o」という種類があります。

以下のスクリプトは、Flickrのサイトから「cat」をキーワードに検索して、サムネイル画像と大きな画像を取得します。

Flickrから画像を取得する　flickr-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'
require 'uri'
require 'cgi'

def save_image(url)
  filename = File.basename(url)
  open("files/"+filename.to_s,'wb') do |file|
    open(url) do |data|
      file.write(data.read)
    end
  end
end

search_word=URI.encode("cat")
doc = Nokogiri::HTML(open(
  "https://www.flickr.com/search/?q=#{search_word}"))
doc.xpath(
  "//a[@class='rapidnofollow photo-click']/img")
```

```
  .each {|link|

  url = link["data-defer-src"]

  # サムネイル画像
  puts url
  save_image(url)

  # 大きなファイル
  url = url.gsub('.jpg','_b.jpg')
  puts url
  save_image(url)
}
```

◆ **flickr-nokogiri.rbの実行例**

```
$ ruby flickr-nokogiri.rb
https://farm8.staticflickr.com/7419/13916738258_a6ea3662dc_b.jpg
https://farm6.staticflickr.com/5487/13607990214_34f9fea37f.jpg
https://farm6.staticflickr.com/5487/13607990214_34f9fea37f_b.jpg
https://farm6.staticflickr.com/5512/14410052054_6296fd600f.jpg
https://farm6.staticflickr.com/5512/14410052054_6296fd600f_b.jpg
https://farm4.staticflickr.com/3680/12954407273_2dcb60090c.jpg
https://farm4.staticflickr.com/3680/12954407273_2dcb60090c_b.jpg
https://farm4.staticflickr.com/3729/13138731845_f419ef6a98_z.jpg
https://farm4.staticflickr.com/3729/13138731845_f419ef6a98_z_b.jpg

~省略~
```

画像収集の場合は、テキストファイルに比べて格段にネットワークの帯域を必要とします。不用意なクローラーを使って、ネットワーク全体を専有しないように注意しましょう。

5-6-2 Flickr API

FlickrにもAPIが用意されています。まずはAPIキーを取得しましょう。

APIキーの取得

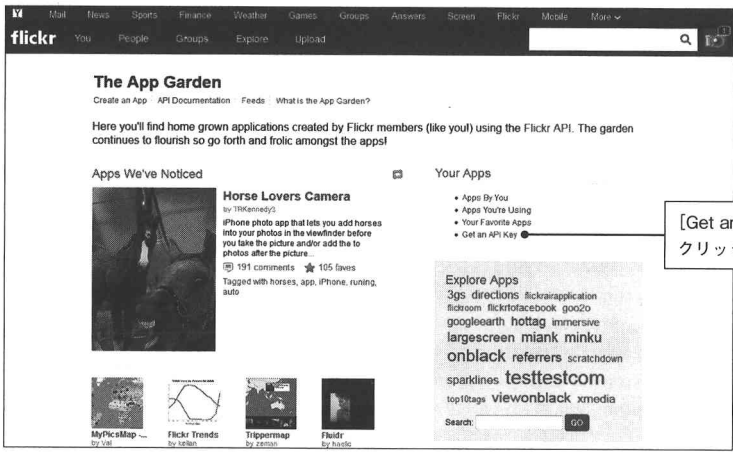
FlickrのAPIのキーは、米Yahoo.comのアカウントがあれば取得できます。米Yahoo.comアカウントでログインのうえで、アプリケーション登録画面に進みます。

■ Flickrのログインページ

URL https://www.flickr.com/services/apps/create/apply/?

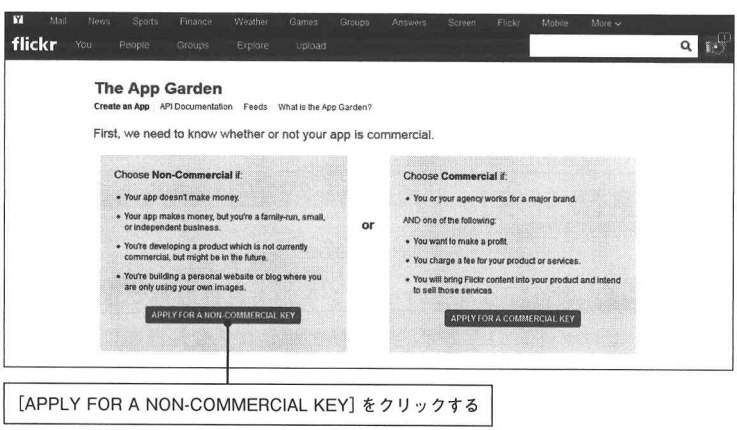
アプリケーションの登録画面は、ページ下のメニューから［App Garden］を選択します。

▼アプリケーションの登録画面



アプリケーションの登録画面で、「Your Apps」の［Get an API Key］をクリックします。

▼審査の申し込み画面



アプリケーションの登録は、商用と非商用で審査の過程が違います。今回は、非商用（Non-Commericial）で登録をします。

［APPLY FOR A NON-COMMERCIAL KEY］ボタンをクリックし、必要な事項を記入して申し込みをすれば、アクセスと秘密のキーが表示されるので、なくさないように保存しておきましょう。

ライブラリのインストール

FlickrのAPIを利用するためのGemライブラリは、サードパーティ製でいくつか出ています。今回は「Flickraw」を利用します。

■ GitHubのFlickrawページ
URL https://github.com/hanklords/flickraw

インストールは、gemから行えます。インストール後にgem list flickrawでバージョンが表示されるとインストール成功です。

◆ Flickrawのインストール

```
$ gem install flickraw
```

◆ インストールの確認

```
$ gem list flickraw

*** LOCAL GEMS ***

flickraw (0.9.8)
```

Flickrawは、多少癖のあるAPIです。検索結果のオブジェクトを取得した後に、さらにFlickrawのインスタンスメソッドで変換をしています。次のスクリプトは、「cat」というタグで検索している例です。

実行の際は、「FlickRaw.api_key」と「FlickRaw.shared_secret」に、先ほど取得したAPIキーを設定してください。

なお、Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

タグで検索して画像を取得する

flickr-api.rb

```
# -*- coding: utf-8 -*-
require 'flickraw'

FlickRaw.api_key='FLICKR_API_KEY'
FlickRaw.shared_secret='FLICKR_API_SECRET'

tag = "cat"

images = flickr.photos.search(
  tags: tag, sort: "relevance", per_page: 20)
images.each{|image|
  url = FlickRaw.url image;
  puts url
}
```

◆ flickr-api.rbの実行例

```
$ ruby flickr-api.rb
https://farm6.staticflickr.com/5593/14679746585_cd75a9c75f.jpg
https://farm4.staticflickr.com/3925/14675178614_f15e59fb19.jpg
https://farm6.staticflickr.com/5551/14490813239_7973aa4882.jpg
https://farm3.staticflickr.com/2925/14674156464_f4db93699f.jpg

~省略~
```

5-7 YouTubeから動画を収集する

画像と同様に、動画を収集するクローラーも作成できます。しかし、画像と比べても、動画は桁違いに大きなファイルです。画質によりますが、1分につき1MB以上の容量が必要となります。10分のファイルを10本ダウンロードしただけでも、1GBの容量が必要となります。そのため、動画向けのクローラーを作成する場合は、動画のダウンロードを行わずにメタ情報（動画のタイトル、説明）とそのURLを収集し、動画リストを作成するのが現実的でしょう。

5-7-1　動画のURLを収集する

それでは動画サイトのクローラーを作成してみましょう。最大規模の動画サイトであるYouTubeを対象にします。

クローラーの造りとしては、キーワード「新幹線」を元に検索し、一覧から動画ページのURLを抽出します。動画ページからは、タイトルと説明文を抽出します。

YouTubeの動画の情報を収集する youtube-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'
require 'uri'

urls = []
search_term = URI.encode("新幹線")

url = "http://www.youtube.com/results?search_query=#{search_term}"

doc = Nokogiri::HTML(open (url))
elements= doc.xpath("//h3[@class='yt-lockup-title']/a")
elements.each do |a|
  code = a.attributes['href'].value
  urls << "http://www.youtube.com" + code if code.include?('watch')
end


urls.each {|url|
  puts url
  doc = Nokogiri::HTML(open(url), nil,"UTF-8")
  title =
     doc.xpath("//h1['watch-headline-title']/span").text.gsub(/¥n/,'')
  description =
     doc.xpath("//p[@id='eow-description']").text
  puts title
  puts description
}
```

◆ youtube-nokogiri.rbの実行例

```
$ ruby youtube-nokogiri.rb
http://www.youtube.com/watch?v=zs-msd2YpTw
  東北新幹線 E6系 スーパーこまちデビュー初日 高速通過～連結 Shinkansen Debut

2013.3.16日撮影 3月16日、ついに営業運転を開始した新型車両E6系『スーパーこまち』
。その営業運転デビュー初日の賑わいを一番列車の高速通過、乗車、連結まで、東北新幹
線-くりこま高原駅・盛岡駅・秋田新幹線(田沢湖線)田沢湖駅で撮影しました。新型車両

～省略～
```

5-7-2 動画をダウンロードする

URLとタイトルや説明文のみでなく、動画も抽出したいという場合は、それぞれの動画向けのダウンロードアプリケーションと組み合わせるのがよいでしょう。例えば、YouTubeの場合は「youtube-dl」というアプリケーションがあります。youtube-dlは、URLを渡すとダウンロードしてmpegで保存します。

■ GitHubのyoutube-dlページ
URL https://github.com/rg3/youtube-dl

インストール方法は省略して、スクリプトとの連携方法のみ紹介します。先ほどのスクリプトのURL一覧表示の部分で、youtube-dlを呼び出す機能を追加します。インストールしたyoutube-dlに対して、事前にパスの設定を行いターミナル/コマンドプロンプトからyoutube-dlで呼び出せることを確認しておいてください。

ここで使うsystemメソッドは、OSのコマンドを直接呼び出す強力な機能です。

YouTubeの動画をダウンロードする youtube-download.rb

```
urls = []
urls.push("http://www.youtube.com/watch?v=HhClT9cqK8E")
urls.push("http://www.youtube.com/watch?v=-MfeittF-qI")

urls.each {|url|
  puts url
  system("youtube-dl -t #{url}")
}
```

youtube-download.rbの実行例

```
$ ruby youtube-download.rb
http://www.youtube.com/watch?v=HhClT9cqK8E
[youtube] Setting language
[youtube] HhClT9cqK8E: Downloading webpage
[youtube] HhClT9cqK8E: Downloading video info webpage
[youtube] HhClT9cqK8E: Extracting video information
[download] Destination: 北陸新幹線'E7系' 関東にお目見え@大宮駅・東京駅-HhClT9cqK8E.mp4
[download] 100% of 139.29MiB in 05:45
http://www.youtube.com/watch?v=-MfeittF-qI
[youtube] Setting language
[youtube] -MfeittF-qI: Downloading webpage
[youtube] -MfeittF-qI: Downloading video info webpage
[youtube] -MfeittF-qI: Extracting video information
```

```
[download] Destination: 山陽新幹線 高速通過集 MAX.speed 300km_h Japanese Bullet Train - Shinkansen--MfeittF-ql.mp4
[download] 100% of 327.04MiB in 14:36
```

ダウンロード対象のファイルが数十～数百MBになるため、回線の速度によりますがダウンロードの時間はそれなりにかかります。ダウンロードしたファイルは、プログラムの直下に保存され、ファイル名は「動画のタイトル名＋ハッシュ文字列.mp4」という形になります。

5-8 iTunes Storeの順位を取得する

iTunesの利用者は全世界で5億人以上、日本だけでも数千万人はいると言われています。膨大なユーザー数の行動結果であるアプリや音楽のランキングは、データとして多大な価値があります。iOSアプリ開発者のみならず、マーケティングに携わる人であるならば、iTunes Storeの順位の変動を抑えておきたいでしょう。毎日、iTunesの総合やカテゴリ別に順位を調べることは可能です。しかし、人力でやるにはあまりに手間がかかります。そこで、クローラーに自動的に働いてもらいましょう。

5-8-1 iTunes Storeのランキング

Web上のiTunes Storeのランキングページには、「ソング」「アルバム」「映画」「ブック」「無料アプリ」「有料アプリ」「ミュージックビデオ」の総合ランキングが公開されています。

◆iTunes Storeのランキングページ

一方で、カテゴリごとのランキングは、iTunesアプリからiTunes Storeを表示した場合にしか見ることはできません。しかし、このiTunesアプリが表示される内容も、実は単なるWebサイトから配信されるHTMLです。その証拠に、各アプリケーションや、ランキングを選択した状態で右クリックすると「リンクを開く」や「リンクをコピーする」の選択肢が出てきます。試しに、iPadのファイナンスのカテゴリの有料Appのランキングを開くと、次のようなURLがコピーできます。

- https://itunes.apple.com/WebObjects/MZStore.woa/wa/viewTop?genreId=6015&id=25177&popId=47

iTunesアプリで見るiTunesStoreランキング

しかし、残念ながらそのURLをブラウザに貼り付けて表示しても、「iTunesアプリを開きます」と表示されるだけです。

こういった場合の可能性の1つは、ユーザーエージェントによって接続元アプリの制限をしている場合があります。そこで、ローカルのPCとiTunes Storeとの通信を解析して、ユーザーエージェントを割り出します。通信解析のソフトはいくつかありますが、「Wireshark」がWindowsでもMacでも利用可能です。

■ Wireshark
URL http://www.wireshark.org/download.html

インストーラーをダウンロードして、指示に従いインストールします。Macの場合はX Windows Systemというソフトが必要となり、Mavericksの場合は付属していません。そのため、X互換のアプリを別途インストールが必要です。Macの場合、

「XQuartz」などが利用できます。

■ XQuartz
URL http://xquartz.macosforge.org/landing/

Wiresharkの使い方は割愛しますが、HTTP通信のみ取得するとiTunes Storeへの通信はiTunesというユーザーエージェントで通信していることがわかります。

• iTunes/11.2.1 (Macintosh; OS X 10.9.2) AppleWebKit/537.74.9¥r¥n

このユーザーエージェントを指定すると、ランキング情報が入った結果が返ってきます。

なお、Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

iTunes Storeのランキング情報を取得する iTunesRankList.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'

url="https://itunes.apple.com/WebObjects/
     MZStore.woa/wa/viewTop?genreId=6015&id=25177&popId=47"
user_agent=
  "iTunes/11.2.1 (Macintosh;  OS X 10.9.2)AppleWebKit/537.74.9¥r¥n"
puts open(url, "User-Agent" => user_agent).read
```

◆ iTunesRankList.rbの実行例

```
$ ruby iTunesRankList.rb
<!DOCTYPE html>
<html lang="en">
 <head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" /><meta name="keywords"
content="iTunes Store" /><meta name="description" content="iTunes Store" /><meta
name="platform-cache-id" content="17" />

~省略~
```

5-8-2 カテゴリIDとランキング種別

次に、iTunesのアプリのランキングの引数について調べてみましょう。カテゴリ別のランキングとそのURLの対応をいくつか見比べていると、「genreId」がカテゴリを指し示し、「popId」がランキングの種別を示すことがわかります。例えば、ファイナンスのカテゴリIDは「6015」です。そして、iPad有料は「47」です。組み合わせると、

- https://itunes.apple.com/WebObjects/MZStore.woa/wa/viewTop?genreId=6015&popId=47

になります。それ以外のカテゴリとランキング種別を表にまとめると、次のとおりです。

カテゴリのID

カテゴリ名(日本語)	カテゴリ名(英語)	genreID
Newsstand	Newsstand	6021
エンターテイメント	Entertainment	6016
カタログ	Catalog	6022
教育	Education	6017
ゲーム	Games	6014
仕事効率化	Productivity	6007
辞書/辞典/その他	Reference	6006
写真/ビデオ	Photography	6008
スポーツ	Sports	6004
ソーシャルネットワーキング	Social Networking	6005
天気	Weather	6001
ナビゲーション	Navigation	6010
ニュース	News	6009
ビジネス	Business	6000
ファイナンス	Finance	6015
ブック	Books	6018
ヘルスケア/フィットネス	Health & Fitness	6013
ミュージック	Music	6011
メディカル	Medical	6020
ユーティリティ	Utilities	6002
ライフスタイル	Lifestyle	6012
旅行	Travel	6003
全体	All	36

▼ ランキング種別のID

ランキング種別	popId
iPhone無料	27
iPhone有料	30
iPhoneトップセールス	38
iPad無料	44
iPad有料	47
iPadトップセールス	46

これ以外にも、国を表すIDがあります。これは引数ではなく、HTTP Requestのヘッダーに「X-Apple-Store-Front」という名前で指定します。日本を表す国IDは「143462-9」になります。

5-8-3 iTunesアプリのランキングを取得する

これらの情報を組み合わせると、次のようなスクリプトになります。カテゴリごと、ランキング種別ごとのランキングをすべて表示します。実際に日々のランキングの変動を保存する場合は、アプリ名ではなくアプリIDで保存する方が都合がよいでしょう。

なお、このスクリプトも実行の際に、SSLのルート証明書の設定が必要です(→p.77、79)。

iTunesアプリからランキングを取得する iTunesRank.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'

Categories = %w(6021 6016 6022 6017 6014 6007 6006
  6008 6004 6005 6001 6010 6009 6000 6015 6018 6013
  6011 6020 6002 6012 6003 36)
PopIdes = %w(27 30 38 44 47 46)

BASE_URL =
  'https://itunes.apple.com/WebObjects/MZStore.woa/wa/viewTop?'
USER_AGENT =
  "iTunes/11.2.1 (Macintosh;  OS X 10.9.2)AppleWebKit/537.74.9¥r¥n"
STORE  = '143462-9'

def getUrl(category,popId)
  return BASE_URL+"genreId="+category+"&popId="+popId
end

def getRanks(url)
```

```
    i = 0
    open(url,
       "User-Agent" => USER_AGENT,
       "X-Apple-Store-Front" => STORE) do |f|
      f.each do |line|
        next if !line.match(/Buy.*salableAdamId=(¥d+)/)
        i += 1
        line.match(/itemName="([^"]+)"/)
        puts i.to_s + ": " + $1
      end
    end
end

# main
Categories.each {|category|
  PopIdes.each {|popId|
    puts "category=#{category}, popID=#{popId}"
    url = getUrl(category,popId)
    getRanks(url)
  }
}
```

◆ iTunesRank.rbの実行例

```
$ ruby iTunesRank.rb
category=6021, popID=27
1:VIBES
2:Seventeen
3:クロスワードフレンズ
4:R25
5:non-no
6:dancyu
7:BE-PAL
8:ディズニーファン
9:ザ・マイカー
10:週プレDigital

~省略~
```

5-9 Google Playの順位を取得する

Google Playは、登録者数ではiTunesには及ばないものの、アプリケーションの多様度ではiTunesを凌ぐほどの規模があります。iTunesと同様に、総合ランキングおよびカテゴリごとのランキングが公開されています。Google Playについても、クローラーでデータを収集し、さらにiTunes Storeのデータの比較を加味すると、アプリケーションの開発やマーケティングの重要なデータとなるでしょう。

5-9-1 Google Playのランキング

Web上のGoogle Playには、「アプリ」「映画＆テレビ」「書籍」のランキングが公開されています。また、Web版にもそれぞれのカテゴリごとのランキングが公開されているため、iTunesに比べてクローリングの難易度は低いです。

アプリの場合は、「ニュース＆雑誌」「ビジネス」「ゲーム」など20以上のサブカテゴリに分かれています。そのなかで「ゲーム」のみは、「アクション」「アドベンチャー」などさらに細かいカテゴリに細分化されています。また、有料版・無料版という区分でのランキングも提供されています。特にユーザーエージェントなどによる制限はされていません。

Google Playのランキングページ

一方で、Google Playの場合、自動的に国を判別してランキングを表示しています。その判断の材料は、接続元のIPや接続アカウントの登録情報を元にしている模様です。そのため、アプリ側で設定を変えて各国のランキングを取得するのは難しいです。各国の公開プロキシを利用するなどして接続元IPを変更することも可能ですが、利用する公開プロキシが安全か確認するのも難しく、避けた方がよいでしょう。

その課題以外は、Google Playをクローリングするのは非常に簡単です。次のスクリプトは、アプリの「ファイナンス」カテゴリから、有償版のアプリのランキングを抽出しています。特筆点としては、Nokogiriの文字コードの自動判別が失敗するために、文字コードを明示的に指定していることだけです。

なお、Rubyに対してSSLのルート証明書の設定もしくは配置をしていない場合、実行時にSSL関係のエラーが発生します。対処方法としては、所定の位置にルート証明書を配置する必要があります。詳しくは、p.77、79を参照してください。

Google Playのランキングを取得する　nokogiri-google-play.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'

# ファイナンス·有償アプリ
url =
  'https://play.google.com/store/apps/category/
   FINANCE/collection/topselling_paid'
doc = Nokogiri::HTML(open(url), nil, 'UTF-8')
doc.xpath("//h2/a[@class='title']").each{ |item|
  puts item[:title]
}
```

◆ nokogiri-google-play.rbの実行例

```
$ ruby nokogiri-google-play.rb
THE RHYTHM OF FIGHTERS
モダンコンバット4:Zero Hour
ギャングスター ベガス
GTA III
Dokuro
グランド·セフト·オート·サンアンドレアス

~省略~
```

5-9-2 Google Playのクローラーライブラリ

Google Playをクローリングするライブラリも存在します。「market_bot」という名前です。これを利用して、Google Playの解析を行います。

■ GitHubのmarket_botページ
URL https://github.com/chadrem/market_bot

◆ market_botのインストール

```
$ gem install market_bot
```

market_botは、Nokogiriを利用してGoogle Playをスクレイピングします。ランキングの取得の他に、新着の取得や価格など付随情報の取得を行います。1アプリごとに個別ページの取得するため、処理時間が少しかかります。しかし、解析部分と対象ページの選択が抽象化され、使い勝手のよいスクリプトを作ることができるでしょう。

なお筆者の手元の環境では、残念ながらWindowsでmarket_botを動かすことはできませんでした。market_botは、ダウンロードのライブラリとしてtyphoeusを利用しています。このライブラリはOSにインストールしたcURLを利用するのですが、SSLの場合にルート証明書が上手く読み込めません。typhoeusのオプションでSSL関係の設定はあるので、market_botがtyphoeusを利用している部分(leaderboard.rb)を編集するとWindowsでも動くと思われます。

ライブラリを利用する play_bot.rb

```
# -*- coding: utf-8 -*-
require 'market_bot'

# ゲームカテゴリの無料アプリのランキング取得
lb = MarketBot::Android::Leaderboard.new(:topselling_free, :game)
lb.update

lb.results.each {|result|
  app = MarketBot::Android::App.new(result[:market_id])
  app.update
  puts "#{app.title} price: #{app.price}"
}
```

◆ play_bot.rbの実行例

```
$ ruby play_bot.rb
LINE:ディズニー ツムツム price:0
白猫プロジェクト price:0
ピコットキングダム price:0
モンスターストライク price:0
Summoners War:Sky Arena price:0

~省略~
```

5-9-3　カテゴリIDとランキング種別

最後に、Google Playのアプリのカテゴリとランキング種別について調べてみましょう。カテゴリ別のランキングとそのURLの対応をいくつか見比べていると、「category/」以下がカテゴリを指し示し、最後にランキングの種別を示すことがわかります。

例えば、ウィジェットのカテゴリは「APP_WIDGETS」です。そして、アプリ有料版ランキングは「topselling_paid」です。組み合わせると、

- https://play.google.com/store/apps/category/APP_WIDGETS/collection/topselling_paid

になります。それ以外のカテゴリとランキング種別を表にまとめると、次のとおりです。

Google Playのカテゴリ

カテゴリ名（日本語）	カテゴリID
書籍＆リファレンス	BOOKS_AND_REFERENCE
ビジネス	BUSINESS
コミック	COMICS
コミュニケーション	COMMUNICATION
教育	EDUCATION
エンターテイメント	ENTERTAINMENT
ファイナンス	FINANCE
ゲーム	GAME
アクションゲーム	GAME_ACTION
アドベンチャーゲーム	GAME_ADVENTURE
アーケードゲーム	GAME_ARCADE

▼ Google Playのカテゴリ（続き）

カテゴリ名（日本語）	カテゴリID
ボードゲーム	GAME_BOARD
カードゲーム	GAME_CARD
カジノゲーム	GAME_CASINO
カジュアルゲーム	GAME_CASUAL
教育ゲーム	GAME_EDUCATIONAL
家族向けゲーム	GAME_FAMILY
壁紙ゲーム	GAME_WALLPAPER
音楽ゲーム	GAME_MUSIC
パズルゲーム	GAME_PUZZLE
レーシングゲーム	GAME_RACING
ロールプレイングゲーム	GAME_ROLE_PLAYING
シミュレーションゲーム	GAME_SIMULATION
スポーツゲーム	GAME_SPORTS
健康＆フィットネス	HEALTH_AND_FITNESS
ライブラリ＆デモ	LIBRARIES_AND_DEMO
ライフスタイル	LIFESTYLE
壁紙	APP_WALLPAPER
メディア＆ビデオ	MEDIA_AND_VIDEO
医療	MEDICAL
音楽	MUSIC_AND_AUDIO
ニュース＆雑誌	NEWS_AND_MAGAZINES
パーソナライゼーション	PERSONALIZATION
写真	PHOTOGRAPHY
製品	PRODUCTIVITY
買い物	SHOPPING
社会	OCIAL
スポーツ	SPORTS
ツール	TOOLS
交通	TRANSPORTATION
旅行＆地域	TRAVEL_AND_LOCAL
天気	WEATHER
ウィジェット	APP_WIDGETS

Google Playのランキング種別

ランキング種別	ランキングId
アプリ有料版ランキング	topselling_paid
アプリ無料版ランキング	topselling_free
アプリ新着有料版ランキング	topselling_new_free
アプリ新着無料版ランキング	topselling_new_paid
おすすめアプリ	editors_choice
アプリ有料版ゲームランキング	topselling_paid_game
有力アプリ	movers_shakers
おすすめアプリ	featured
タブレット向けアプリ	tablet_featured
人気急上昇アプリ	topgrossing

5-10 SEOに役立てる

いくつかの種類のクローラーを作ってきましたが、次はSEO（検索エンジン最適化）に役立てるクローラーを作成します。

一般的にSEOとクローラーというキーワードであれば、検索エンジンのクローラーに対して、いかに巡回しやすくするかがポイントになります。今回は、逆に自作のクローラーを使って、SEOの手助けをするということを考えてみます。

5-10-1　検索順位を収集する

クローラーを使い、検索順位を定期的に計測します。通常、検索エンジン経由のアクセス向上を狙う場合、ターゲットとなるキーワードを設定し、そのキーワードでアクセスが増えるように施策を行います。この施策の部分がSEOになります。これに対して、今回作成しようとするSEO支援クローラーは、その成果確認のために動きます。

具体的には、調査対象となるドメインと検索キーワードを設定し、検索キーワード（❶）に対して対象のドメイン（❷）が何位に位置するか取得します。日々検索順位測定アプリを動かして結果を保存することによって、施策に対してどれくらいでどのような効果があったのかを確認できます。

キーワードごとの検索順位を取得する　google-rank.rb

```
# -*- coding: utf-8 -*-
require 'google-search'
```

```
def find_item uri, query
  search = Google::Search::Web.new do |search|
    search.query = query
    search.size = :large
    search.each_response { print '.'; $stdout.flush }
  end
  puts uri
  search.find { |item| item.uri =~ uri }
end

def rank_for domain,query
  print "%35s " % query
  if item = find_item(/#{domain}/, query)
    puts " #%d" % (item.index + 1)
  else
    puts " Not found"
  end
end

regular_expression_domain = 'takuros¥.net'  ❷
target_word = 'ruby クローラー'  ❶
rank_for(regular_expression_domain,target_word)
```

◆ google-rank.rbの実行例

```
$ ruby google-rank.rb
   ruby クローラー (?-mix:takuros¥.net)
.#1
```

5-10-2 被リンク

SEOの重要な指標の1つに、被リンク数(back links)やページランクというのがあります。自分のサイトが、他のサイトからどれくらいリンクを貼られているのか、またURLのGoogleのページランクはどれくらいなのかという指標です。単純に言うと、ページランクが高いサイトからリンクをたくさん集めれるのが、SEOとして効果的という話です。

最近では検索エンジンの進化で、被リンク数やページランクの意味は薄れてきています。しかし、被リンク数やページランクが増えているのかは、統計的には少し意味があるでしょう。そこで、定期的に収集するために、スクリプトから取得してみます。

被リンクの取得方法

被リンクの取得方法は、検索エンジンごとに差異があります。Googleの場合、「link:検索」を利用します。例えば、Googleに対する被リンクの調査は、「link:www.google.co.jp」という検索方法になります。しかし、Googleは被リンク情報を正確には開示しなくなっているため、参考程度にしかなりません。

同様に、インデックス数の調査は、「site:検索」を利用します。ページランクについては、Googleは少し特殊で「http://toolbarqueries.google.com/tbr」に対して、URLとそれに対するチェックサムを付加してリクエストを送ります。このような検索を各検索エンジンごとに行うことで、それぞれの指標の調査ができます。

ライブラリの利用

上記の機能を1つ1つ実装すれば、目的の指標は取得できます。しかし、実装対象が多いことと、それぞれの検索エンジンの挙動が不定期に変わる可能性もあります。そこで、方法の1つとしてメンテナンスされているGemライブラリを利用するという手もあります。「PageRankr」は、Ruby製の被リンク調査ライブラリです。Googleの他に、Bing、Yahoo、Alexaの被リンクを取得します。また、被リンクのみでなくインデックス数やページランクの取得も行えます。

■ GitHubのPageRankrページ
URL https://github.com/blatyo/page_rankr

PageRankrのインストールは、gemから行えます。ただし、依存ライブラリであるJSONなどでネイティブビルドが必要になります。そのため、インストールにはビルドツールが必要となります。ビルドツールのインストールについては、p.76をご参照ください。インストール後にgem list PageRankrでバージョンが表示されれば、インストール成功です。

◆ PageRankrのインストール

```
$ gem install PageRankr
```

◆ インストールの確認

```
$ gem list PageRankr

*** LOCAL GEMS ***
```

PageRankr (4.2.0)

PageRankrは、インスタンスメソッドのみ提供します。取りたい情報に対して、「クラス名#メソッド名」という形で取得します。引数も、取得対象のURLとオプションで対象の検索エンジンを指定するのみです。返り値は、ハッシュで返ります。

PageRankrで被リンクを取得する

page_rankr.rb

```
# -*- coding: utf-8 -*-
require 'page_rankr'

target_url = "d.hatena.ne.jp/dkfj"
# target_url = "blog.takuros.net"

puts "バックリンク数 チェック"
puts PageRankr.backlinks(target_url, :google, :bing, :yahoo, :alexa)
puts "インデックス数 チェック"
puts PageRankr.indexes(target_url, :google, :bing, :yahoo)
puts "ページランク チェック"
puts PageRankr.ranks(
  target_url, :alexa_us, :alexa_global, :google, :moz_rank, :page_authority)
```

page_rankr.rbの実行例

```
$ ruby page_rankr.rb
バックリンク数 チェック
{:google=>5, :bing=>2930, :yahoo=>nil, :alexa=>20}
インデックス数 チェック
{:google=>2090, :bing=>3060, :yahoo=>nil}
ページランク チェック
{:alexa_us=>468457, :alexa_global=>611733, :google=>nil, :moz_rank=>4, :page_authority=>27}
```

ツールから被リンクを取得する

被リンクやインデックス数については、プログラムから取得する以外にも検索エンジン各社が提供しているツールから取得できます。例えば、Googleの場合はウェブマスタツールとして提供されています。機械的に処理するのでなければ、専用ツールを利用しましょう。

■ Google　ウェブマスタツール
URL https://www.google.com/webmasters/tools/

5-11 Wikipediaのデータを活用する

フリーで利用できるテキストデータとして、最大級のものはWikipediaでしょう。日本語サイトのみでも、2014年3月で90万記事を超えました。詳細は統計ページで確認できますが、1万人以上の編集者、100万人近い利用者、随時編集される記事と、桁違いの規模です。

■ Wikipedia：統計
URL http://ja.wikipedia.org/wiki/%E7%89%B9%E5%88%A5:%E7%B5%B1%E8%A8%88

クローラーの目的がデータの収集であれば、このWikipediaのデータを活用しない手はありません。データの集合体という観点からWikipediaを考えて、どのように利用するか検討してみましょう。

5-11-1　Wikipediaからのクローリングとデータ

Wikipediaに対するクローリングは明示的に禁止されています。そのかわりに、Wikipediaのデータすべてがダウンロード可能になっています。

■ Wikipedia：データベースダウンロード
URL http://ja.wikipedia.org/wiki/Wikipedia:%E3%83%87%E3%83%BC%E3%82%BF%E3%83%99%E3%83%BC%E3%82%B9%E3%83%80%E3%82%A6%E3%83%B3%E3%83%AD%E3%83%BC%E3%83%89

ダウンロードは、次のURLから行えます。

■ Index of /jawiki/latest/
URL http://dumps.wikimedia.org/jawiki/latest/

データは、XML形式もしくはMySQLのダンプファイルとして提供されています。用途ごとに細分化して提供されているので、主要なもののみ簡単にまとめます。

Wikipediaのデータ

ファイルの説明	ファイル名	ファイルサイズ
全ページの要約	jawiki-latest-abstract.xml	1.4GB
全ページのタイトル	jawiki-latest-all-titles-in-ns0.gz	8.7MB
カテゴリの一覧	jawiki-latest-category.sql.gz	2.5MB
カテゴリのリンク情報	jawiki-latest-categorylinks.sql.gz	116.4MB

Wikipediaのデータ（続き）

ファイルの説明	ファイル名	ファイルサイズ
外部リンク情報	jawiki-latest-externallinks.sql.gz	158.7MB
画像情報	jawiki-latest-image.sql.gz	11.1MB
画像のリンク先情報	jawiki-latest-imagelinks.sql.gz	29.9M
ページ情報	jawiki-latest-page.sql.gz	90.4MB
ページ間のリンク情報	jawiki-latest-pagelinks.sql.gz	690.8MB
全ページの記事本文	jawiki-latest-pages-articles.xml.bz2	1.8GB
議論ページなどを含んだ全ページの記事本文	jawiki-latest-pages-meta-current.xml.bz2	2.1GB

どういった用途で活用するかにより、ダウンロードする対象が変わってきます。例えば、クローリング先を取得したいのであれば外部リンクの情報を含んだ「jawiki-latest-externallinks.sql.gz」が必要となります。キーワードなどの辞書作成をしたいのであれば、タイトル一覧などが役に立ちます。もっと踏み込んで、自然言語処理などで類似文章検索の元データとしたいのであれば、全ページの要約もしくは記事本文、ページ間のリンク情報などが利用できます。用途に応じて、使い分けましょう。「5-12 キーワードを収集する」（→p.311）や「5-13 流行をキャッチする」（→p.314）で、Wikipediaのデータを活用した例を紹介します。

5-11-2 Wikipediaのカテゴリの活用

Wikipediaは、全部のデータをダウンロードできます。一方で、数ページ分だけの情報が必要な場合には、手間がかかります。そういった場合は、クローラーを利用するのも1つの手でしょう。Wikipediaに対するクローラーの例として、カテゴリの活用があります。Wikipediaのカテゴリページは、分野ごとに人力で編集されています。例えば、人名一覧を作成する際に重宝します。

■ Wikipedia：カテゴリ

URL **http://ja.wikipedia.org/w/index.php?title=%E7%89%B9%E5%88%A5:%E3%82%AB%E3%83%86%E3%82%B4%E3%83%AA**

Wikipediaのカテゴリは、「http://ja.wikipedia.org/wiki/Category:」の後にカテゴリ名が続きます。またカテゴリに下に、さらにサブカテゴリがある場合があります。

次のスクリプトは、「日本の映画作品_（ジャンル別）」というカテゴリを起点に（❶）再帰的にサブカテゴリを抽出するプログラムです。Wikipediaに対して再帰的処理を実装するとほぼ無限に検索してしまうので、検索の深さも指定しています（❷）。

Wikipediaからカテゴリを取得する

wikipedia-category.rb

```
# -*- coding: utf-8 -*-
require "nokogiri"
require "open-uri"

@base_url = "http://ja.wikipedia.org"
# 検索対象のカテゴリのURL
# 日本の映画作品_(ジャンル別)
category_url = "/wiki/Category:%E6%97%A5%E6%9C%AC%E3%8 ●―――❶
  1%AE%E6%98%A0%E7%94%BB%E4%BD%9C%E5%93%81_(
  %E3%82%B8%E3%83%A3%E3%83%B3%E3%83%AB%E5%88%A5)"

def category_search(url,depth)
  return if depth >= 4 ●―――❷
  doc = Nokogiri::HTML(open(@base_url+url))
  doc.xpath("//div[@class='CategoryTreeItem']/a").each do |element|
    puts element.text
    puts element[:href]
    # 再帰的に取得
    category_search(element[:href],depth+1)
  end
end

category_search(category_url,1)
```

◆ wikipedia-category.rbの実行例

```
$ ruby wikipedia-category.rb
日本のテレビアニメの劇場版
/wiki/Category:%E6%97%A5%E6%9C%AC%E3%81%AE%E3%83%86%E3%83%AC%E3%83%93%E
3%82%A2%E
3%83%8B%E3%83%A1%E3%81%AE%E5%8A%87%E5%A0%B4%E7%89%88
宇宙戦艦ヤマトシリーズ
/wiki/Category:%E5%AE%87%E5%AE%99%E6%88%A6%E8%89%A6%E3%83%A4%E3%83%9E%E3
%83%88%E
3%82%B7%E3%83%AA%E3%83%BC%E3%82%BA
宇宙戦艦ヤマトシリーズの登場人物
/wiki/Category:%E5%AE%87%E5%AE%99%E6%88%A6%E8%89%A6%E3%83%A4%E3%83%9E%E3
%83%88%E

~省略~
```

上記スクリプトに取得した対象のページも追加すると、簡単に一覧からのデータ取得ができるようになります。

5-12 キーワードを収集する

クローラーで収集した文章を解析する際に、キーワード辞書を用意して、マッチングで出現文字列のカウントをする場合があります。この方式の場合は、辞書のメンテナンスが肝となります。初期キーワードを一括で作成し、日々追加すべきキーワードを探すのはなかなか手間です。集合知の力を借りることで、その負荷を軽減できます。

Wikipediaやはてなキーワードを利用し、日々追加されるキーワードをウォッチします。自分の辞書に追加すべきキーワードがあれば、追加するという方式です。

5-12-1 Wikipediaのタイトル

キーワード情報の生成の元データとして、Wikipediaのタイトルを使うという方法があります。タイトルは記事数分だけ存在するので、2014年7月現在で90万件と膨大なデータを利用できますが、残念ながら、タイトルのみの差分更新はしていません。タイトルのみの全データが配布されているので、初期データとして取り込むのもよいでしょう。

データは、1行1タイトルという形式で格納されています。ただのテキストファイルで、URLも固定なのでクローラーの作成の必要はありません。

■ Wikipedia：最新タイトル一覧
URL http://dumps.wikimedia.org/jawiki/latest/jawiki-latest-all-titles-in-ns0.gz

5-12-2 はてなキーワード

はてなキーワードは、株式会社はてなが提供するインターネット百科事典の一種です。Wikipedia同様に、編集はユーザーによって行われます。一括のダウンロードの他に、日毎の更新が取得可能です。Wikipediaに比べて身近な話題のキーワードが多く、多様性に富んでいます。

■ はてなキーワード一覧ファイル
URL http://developer.hatena.ne.jp/ja/documents/keyword/misc/catalog

日毎の差分データの取得は、新着順の並び指定で取得します。ただし、日の指定で取得する方法は存在しないので、最新の状態から順番に取得し当日分のデータがなくなるまで処理します。

はてなキーワードの新着ワードを取得する

hatena-keyword-new.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'
require 'date'

def get_nokogiri_doc(url)
  begin
    html = open(url,{"Cookie" =>
      "rk=475a389917c3594554b534074992be19bf79007"})
  rescue OpenURI::HTTPError
    return
  end
  Nokogiri::HTML(html.read)
end

def has_next_page?(doc)
  doc.xpath("//*[@id='main']/div/div[2]/a").each {|element|
    return true if element.text == "次の20件>"
  }
  return false
end

start_url=
  "http://d.hatena.ne.jp/keywordlist?s=created&r=1"
num=0
loop {
  url = "#{start_url}&of=#{num}"
  doc = get_nokogiri_doc(url)
  doc.xpath("//div[@class='keyword-list']/ul/li").each {|element|
    created = element.xpath("div/span[@class='created']/span").text
    day = Date.parse(created)
    exit if (Date.today - day).to_i >= 1
    puts element.xpath("h3/a").text
  }
  num = num+20
  break if !has_next_page?(doc)
}
```

◆ hatena-keyword-new.rbの実行例

```
$ ruby hatena-keyword-new.rb
果実
Granata
Montserrat
```

```
Granada

~省略~
```

5-12-3 Google Suggest API

特定のキーワードに関係するキーワードを取得したい場合があります。十分な学習データがあれば、自然言語処理で計算可能です。一方で手早く結果だけを取得したい場合は、Google Suggest APIを利用して結果だけ取得することも可能です。

次のスクリプトは、Googleの検索結果から「猫」に関連するキーワードを取得しています。

Googleの検索結果から関連キーワードを取得する google_suggest.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'uri'
require 'rexml/document'

base_url = 'http://www.google.com/complete/search?hl=ja&output=toolbar'
keyword=URI.encode('猫')
url="#{base_url}&q=#{keyword}"
doc = REXML::Document.new(open(url).read.encode("UTF-8"))
doc.elements.each('toplevel/CompleteSuggestion/suggestion')do |element|
  puts element.attributes["data"]
end
```

◆ google_suggest.rbの実行例

```
$ ruby google_suggest.rb
猫侍
猫
猫 里親
猫 画像
猫 種類
猫 アワビ
猫カフェ
猫戦車
猫ピッチャー
猫の恩返し
```

関連キーワードを取得することで、例えばどのようなキーワードで検索されやすいかを推測できます。コンテンツに反映させることで、より検索されやすいページになるでしょう。

5-13 流行をキャッチする

マーケティングを生業にしている人であるならば、クローラーを使ってトレンドの推移を知りたいと思うでしょう。GoogleやTwitterのデータを活用することで、きっとその手助けをすることができます。トレンドには、短期的なものと長期的なものがあります。それぞれ、どんな方法で取得できるか考えてみます。

5-13-1　瞬間的なトレンドをキャッチする

Twitterでつぶやかれているキーワードや、Googleで検索されている語から、今この瞬間のトレンドがわかります。TwitterもGoogleもそれぞれ、旬なキーワードとして提供しています。

Googleの旬なキーワード

Googleは、Googleトレンドのサービスの一部として、Atom形式で提供しています。使い方は簡単で、次のURLからデータを取得するだけです。

■ Googleの旬なデータ
URL http://www.google.co.jp/trends/hottrends/atom/hourly

上位20個の旬なキーワードと、Googleトレンドへのリンクの URLが提供されています。

Google Hot Trends

```
What are people searching for on Google today? http://www.google.com/trends/hottrends/atom/hourly,2007-08 2014-07-03T05:00:00Z
2014-07-03T05:00:00Z
• 小野恵令奈
• 会見
• 野々村竜太郎
• 野々村議員
• 集団的自衛権
• 野々村
• 後藤真希
• アルゼンチン
• メッシ
• VAIO
• 強襲
• 任天堂
• 集団的自衛権
• シンドカフカ
• オチョア
• きゃりーぱみゅぱみゅ
• 新宿 焼身自殺
• ブラジル ヂリ
• ネイマール
• 宝塚記念
```

Googleから旬なキーワードを取得する

google_trends_word.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'

atom = open('http://www.google.co.jp/trends/hottrends/atom/hourly')
doc = Nokogiri::HTML(atom)
#puts doc

doc.xpath("/html/body/feed/entry/content/li/span/a").each {|element|
  puts element.text
  puts element[:href]
}
```

◆ google_trends_word.rbの実行例

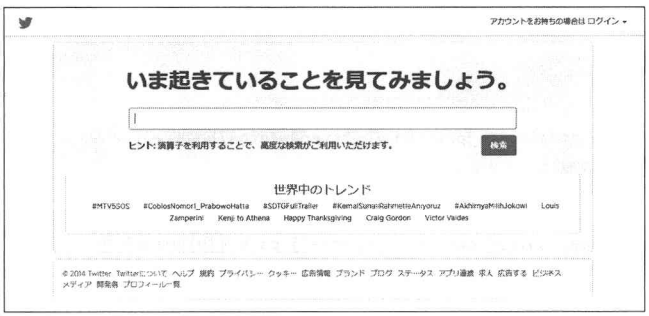
```
$ ruby google_trends_word.rb
小野恵令奈
http://www.google.co.jp/trends/hottrends?pn=p4#a=20140703-%25E5%25B0%258F%25E9%2
587%258E%25E6%2581%25B5%25E4%25BB%25A4%25E5%25A5%2588
号泣会見
http://www.google.co.jp/trends/hottrends?pn=p4#a=20140703-%25E5%258F%25B7%25E6%2
5B3%25A3%25E4%25BC%259A%25E8%25A6%258B
野々村竜太郎
http://www.google.co.jp/trends/hottrends?pn=p4#a=20140702-%25E9%2587%258E%25E3%2
580%2585%25E6%259D%2591%25E7%25AB%259C%25E5%25A4%25AA%25E9%2583%258E
野々村議員
http://www.google.co.jp/trends/hottrends?pn=p4#a=20140702-%25E9%2587%258E%25E3%2
580%2585%25E6%259D%2591%25E8%25AD%25B0%25E5%2593%25A1

~省略~
```

Twitterのトレンド

Twitterの場合も、タイムラインの横にトレンドが表示されます。しかし、タイムラインはログイン状態でないと、ログインもしくはアカウント登録を促されるページが表示され、トレンドも表示されません。そこで、「Twitter/検索」画面を利用します。

■ Twitter/検索
URL https://twitter.com/search-home

Twitter/検索画面



Twitterのトレンド情報は、遅延ロードで呼び出されます。そのため、open-uriなどでHTMLのみを取得しても、トレンドの情報は出てきません。そこで、Capybaraを利用して、遅延ロードにも対処します。Capybaraの遅延ロードについては、p.124をご参照ください。

なお、Windowsでcapybara-webkit（→p.123）を利用する場合は、qmake 4.8のインストールが必要となります。また、qmakeは、mingw-w64を必要とします。そのため、mingw-w64をインストール後に、qmakeをインストールしましょう。インストール後は、qmake.exeにパスを通すことにより、capybara-webkitがインストール可能になります。

■ qmake 4.8
URL http://qt-project.org/downloads

■ MinGW-w64
URL http://sourceforge.net/projects/mingw-w64/

◆ capybara-webkitのインストール

```
$ gem install capybara-webkit
```

▌Twitterのトレンドを取得する　　twitter_trends_capybara.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'capybara-webkit'

Capybara.default_selector = :xpath
```

```
Capybara.default_driver = :selenium
Capybara.app_host = "https://twitter.com/search-home"

module Spider
  class Twitter
    include Capybara::DSL

    def search
      visit('')
      all("//ul[@class='trend-items js-trends']/li/a").each do |element|
        puts element.text
      end
    end
  end
end

spider = Spider::Twitter.new
spider.search
```

◆ **twitter_trends_capybara.rbの実行例**

```
$ ruby twitter_trends_capybara.rb
#GazzedeBebekKatliam?
#ReplaceMovieTitleWithTurtle
#DearROBIN
#CocaColayaHay?r
#NRPAsk
RIP James Garner
Bukti Jokowi Menang Capres2014
She's Dating The Turtle
Chara
Philip Hammond
```

Capybaraを使うことで、遅延ロードにも対処できます。しかし、Twitterのトレンドは、ログインしていない状態だとデフォルトの全世界になります。日本のトレンドを取得したい場合は、言語設定を日本にしているアカウントでログインするか、検索語の画面でトレンドの設定を行う必要があります。処理として効率が悪いので、APIを利用しましょう。

APIでTwitterのトレンドを取得する

Trends APIを利用すると、与えられた地域IDごとのトレンドを取得できます。地域IDは、Yahoo!の「Where On Earth ID (woeid)」を使用しています。

実行に際しては、「:consumer_key」「:consumer_secret」「:access_token」「:access_token_secret」にTwitterのAPIキーなどを設定してください。APIキーについては、p.272をご参照ください。また、RubyからTwitterのAPIを利用するには、Gemのライブラリを利用すると効率がよいです。今回は、「5-4 Twitterデータの収集」(→p.267)で利用したtwitterライブラリを使います。

■ Twitter DevelopersのGET trends/placeページ
URL **https://dev.twitter.com/docs/api/1.1/get/trends/place**

■ Yahoo! GeoPoint
URL **https://developer.yahoo.com/geo/geoplanet/guide/concepts.html**

woeidを調べるには、米国のYahoo! IDを取得したうえで、開発者登録を行います。ApplicationIDを取得します。調べたい場所の略称とApplicationIDをURLにセットすることにより、結果のJSONを取得することができます。次の例は、サンフランシスコ空港を調べた場合です。

- http://where.yahooapis.com/v1/places.q(SFO)?appid=YourApplicationID

またwoeidだけであれば、TwitterのAPIからでも取得できます。ただし、それが指し示す場所は、別途調べる必要があります。

▌woeidを取得する　　**twitter_woeid_api.rb**

```
# -*- coding: utf-8 -*-
require 'twitter'

config = {
  :consumer_key => 'TWITTER_API_KEY',
  :consumer_secret => 'TWITTER_API_SECRET',
  :access_token => 'TWITTER_ACCESS_TOKEN',
  :access_token_secret => 'TWITTER_ACCESS_TOKEN_SECRET'
}

client = Twitter::REST::Client.new(config)
client.trends_available.each {|available|
  puts available.id
}
```

◆ twitter_woeid_api.rbの実行例

```
$ ruby twitter_woeid_api.rb
1
2972
3369
3444
3534
4118
8676

~省略~
```

APIでTwitterのトレンドを取得する

twitter_trends_api.rb

```
# -*- coding: utf-8 -*-
require 'twitter'

config = {
  :consumer_key => 'TWITTER_API_KEY',
  :consumer_secret => 'TWITTER_API_SECRET',
  :access_token => 'TWITTER_ACCESS_TOKEN',
  :access_token_secret => 'TWITTER_ACCESS_TOKEN_SECRET'
}
# 日本
place_id = 23424856
client = Twitter::REST::Client.new(config)
client.trends(place_id).each {|trend|
  puts trend.name
  puts trend.url
}
```

◆ twitter_trends_api.rbの実行例

```
$ ruby twitter_trends_api.rb
#か行の後に依存症つけてみた
http://twitter.com/search?q=%23%E3%81%8B%E8%A1%8C%E3%81%AE%E5%BE%8C%E3%81%AB%E4%BE%9D%E5%AD%98%E7%97%87%E3%81%A4%E3%81%91%E3%81%A6%E3%81%BF%E3%81%9F
#ま行のあとに自分の名前をつけるとなんか悪口
http://twitter.com/search?q=%23%E3%81%BE%E8%A1%8C%E3%81%AE%E3%81%82%E3%81%A8%E3%81%AB%E8%87%AA%E5%88%86%E3%81%AE%E5%90%8D%E5%89%8D%E3%82%92%E3%81%A4%E3%81%91%E3%82%8B%E3%81%A8%E3%81%AA%E3%82%93%E3%81%8B%E6%82%AA%E5%8F%A3

~省略~
```

5-13-2　長期的なトレンドをキャッチする

先ほどのTwitterやGoogleを使った方法は、短期的なトレンドをキャッチする方法です。マーケティングで使うのであれば、例えばこれからトレンドがきそうなものをキャッチしたいでしょう。残念ながら、そのものずばりの方法はありません。しかしながら、手がかりになるようなデータは収集できます。1つは、先ほどのGoogleトレンド(→p.314)を使って、時系列のデータを抜き出す方法です。

Googleトレンドのグラフは、Google Chart Toolsを利用してJavaScriptで描画されています。そのため、元となるデータはすべてHTML中に含まれています。これを利用して、検索数の推移を数値として取得することができます。

次のスクリプトは、「クラウド」というキーワード(❶)の検索数の推移を取得しています。検索開始日(❷)を指定することもできます。

Googleの検索数の推移を取得する　　google_trends_data.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'uri'

word = "クラウド" ●―――――❶

# 検索開始日の設定
day=Time.now
# 36ヶ月前
day=day - 1080*24*60*60 ●―――――❷
month = day.month
year = day.year

url = URI.encode("http://www.google.com/trends/
  fetchComponent?hl=en&q=#{word}&date=#{month}/
  #{year}+36m&cmpt=q&content=1&cid=
  TIMESERIES_GRAPH_0&export=5&w=500&h=330")

html = open(url)
raw_data_array = html.read.split("rows¥":")[1]
  .split("var htmlChart").first
split_raw = raw_data_array.split("],")
split_raw.each do |raw|
  source = raw.split("Date(")[1].split(",")
  date = source[0]+'-'+(source[1].gsub(/¥s/,"")
    .to_i+1).to_s+'-'+source[2].gsub(/¥s/,"")
    .gsub(/¥)¥}/,"") #.gsub(/)},/,'')
  num = source[5]
  puts "#{date}, #{num}"
end
```

◆ google_trends_data.rbの実行例

```
$ ruby google_trends_data.rb
2011-7-6,54
2011-7-13,64
2011-7-20,57

~中略~

2014-6-18,77
2014-6-25,76
2014-7-2,76
```

得られたデータをどのように活用するかは、工夫次第です。なお、Google Trends APIを認証なしでスクリプトから直接扱うと、非常に少ない回数で利用制限がされます。ご注意ください。

これ以外にも、「5-4 Twitterのデータ収集」(→p.267)で紹介したTwitter Streaming APIで毎日データを収集して、日毎の出現数をカウントすることでトレンドの推移はわかります。出現数の推移を統計的に計算することにより、トレンドの兆しを見つけることができるかもしれません。

5-14 企業・株価情報を収集する

企業・株価情報を収集する目的でクローラーを作成しようとする人は多いでしょう。定番の方法ですが、Yahoo!ファイナンスから企業情報と株価情報を取得する方法を試してみましょう。

5-14-1 証券コード一覧を取得する

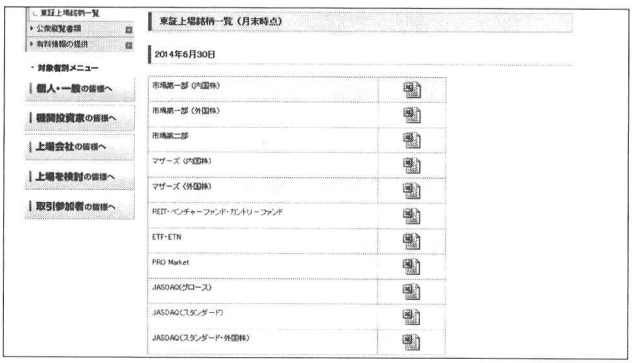
企業・株価情報を収集する際は、まずは証券コードの一覧を取得することから始めます。証券コードはいくつかありますが、一般的には銘柄コードのことを指します。銘柄コードは4桁の数字で、上場株式やその他の上場証券(ETFなど)に割り振られています。

クローリングの際に証券コードが必要になる理由は、金融情報などを扱うWebサイトの多くで、管理を証券コードで行うことが多いためです。一覧の取得は、東証サイトの東証上場銘柄一覧などからダウンロードできます。

■ 東証：東証上場銘柄一覧
URL http://www.tse.or.jp/market/data/listed_companies/index.html

別の方法として、企業情報を証券コードで扱っているWebサイトに対して、0001から9999まで試行すればすべて取得できます。ただし、対象のWebシステム側の負荷になるので、東証が一覧を公開しているかぎりは、そこからダウンロードするのが一番です。エクセル形式で公開しているため、ダウンロード後にシステムで扱いやすいテキストファイルに変換するとよいでしょう。

東証上場銘柄一覧ダウンロード



5-14-2　企業情報および当日の株価を収集する

証券コードを取得できたら、次は企業情報の取得です。収集対象として考えられるのが、代表者名や住所、業種などの属性情報と、発行株式数や単元株、年初来高値・安値などの株価に関する情報です。これらの情報が揃っているサイトは、日本経済新聞の日経会社情報とYahoo!ファイナンスです。

■ 日経会社情報
URL http://www.nikkei.com/markets/company/

■ Yahoo!ファイナンス
URL http://finance.yahoo.co.jp/

今回の例では、Yahoo!ファイナンスを対象とします。ページの特徴として、項目を一意に取得するidやclassなどはほとんどありません。そのため、相対的な位置などの指定で値を取得する必要があります。単純な構造なので解析は難しくないのですが、メンテナンス性に劣るのが難点です。

なお、今回のスクリプトは、会社情報の取得部分をクラス化しています。ここまでは単純なバッチ形式で処理を作ってきましたが、クラス化しておいた方が機能分割やテストが容易で保守性が高いスクリプトを作りやすいです。

このスクリプトは、証券コード「4689」(ヤフー(株))の情報を取得しています(❶)。証券コードを切り替えることで、さまざまな会社に対応できます。

Yahoo!ファイナンスから企業情報を取得する nokogiri-stock.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

class CompanyInfo
    def initialize(ticker_code)
      @baseUrl = "http://stocks.finance.yahoo.co.jp/stocks"
      @tickerCode = ticker_code
      scrape
    end
    attr_reader :name, :tickerCode, :category,
      :unit, :recentHighPrice, :recentLowPrice,
      :highPrice, :lowPrice, :price

    private
    def scrape_stock_info(html, index)
        get_content(
          html, "dd", "ymuiEditLink marO", index, "/strong").delete(",")
    end

    def get_company_info()
      url = "#{@baseUrl}/profile/?code=#{@tickerCode}"
      doc = get_nokogiri_doc(url)
      @name = doc.xpath("//th[@class='symbol']/h1").text
      @category =
        doc.xpath("//table[@class='boardFinCom marB6']/tr[6]/td").text
      @unit = doc.xpath(
        "//table[@class='boardFinCom marB6']/tr[13]/td").text
    end

    def get_stock_info()
      url = "#{@baseUrl}/detail/?code=#{@tickerCode}"
      doc = get_nokogiri_doc(url)
      @recentHighPrice =
        doc.xpath(
        "//div[11]/dl/dd[@class='ymuiEditLink marO']/strong").text
      @recentLowPrice =
        doc.xpath(
        "//div[12]/dl/dd[@class='ymuiEditLink marO']/strong").text
```

```
      @highPrice =
        doc.xpath("//div[@class='innerDate']
          /div[3]/dl/dd[@class='ymuiEditLink marO']
          /strong").text
      @lowPrice =
        doc.xpath("//div[@class='innerDate']
          /div[4]/dl/dd[@class='ymuiEditLink marO']
          /strong").text
      @price = doc.xpath("//td[@class='stoksPrice']").text
    end

    def get_nokogiri_doc(url)
      begin
        html = open(url)
      rescue OpenURI::HTTPError
        return
      end
      Nokogiri::HTML(html.read, nil, 'utf-8')
    end

    def scrape
      get_company_info
      get_stock_info
    end
end

company = CompanyInfo.new("4689") ●―――――❶
puts company.name
puts company.category
puts company.unit
puts "年初来高値:"+company.recentHighPrice
puts "年初来安値:"+company.recentLowPrice
puts "高値:"+company.highPrice
puts "安値:"+company.lowPrice
puts "株価:"+company.price
```

◆ nokogiri-stock.rbの実行例

```
$ ruby nokogiri-stock.rb
ヤフー(株)
情報・通信
100株
年初来高値:668
年初来安値:408
高値:483
安値:478
株価:480
```

5-14-3 株価の時系列データを収集する

過去の株価の推移を元に、株価の分析をする場合があります。Yahoo!ファイナンスでは、時系列データの取得も可能です。日付を遡ってデータを取得するスクリプトを作ってみましょう。時系列データ取得ページは、次のようなURLになっています。

- http://info.finance.yahoo.co.jp/history/
  ?code=4689.T&sy=1900&sm=1&sd=1&ey=2014&em=5&ed=27&tm=d&p=2

引数で証券コード、日付のFrom-To、ページ数を指定します。1ページにつき50件ずつ表示します。

スクリプト作成のポイントは、最終ページまで回帰的にデータを取得することです。ページを1ページずつ処理し、その後に次のページがあるかの判定をします。判定は、改ページのリストであるページャの部分に「次へ」の文言があるかどうかで判定しています。また、企業ごとに上場開始日が異なります。そのため、検索期間のFromを1900年1月1日に指定することにより、すべてのデータを取得できるようにしています。

時系列に沿って株価を取得する nokogiri-stock-history.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

def get_nokogiri_doc(url)
  begin
    html = open(url)
  rescue OpenURI::HTTPError
    return
  end
  Nokogiri::HTML(html.read, nil, 'utf-8')
end

def has_next_page?(doc)
  doc.xpath("//*[@id='main']/ul/a").each {|element|
    return true if element.text == "次へ"

  }
  return false
end
```

```
def get_daily_data(doc)
  doc.xpath(
    "//table[@class='boardFin yjSt marB6']/tr").each {|element|

    # 日付行および株式分割告知を回避
    if element.children[0].text !=
      "日付" && element.children[1][:class] != "through"

      # 日付
      day = element.children[0].text

      # 始値
      open_price = element.children[1].text

      # 高値
      hight_price =  element.children[2].text

      # 安値
      low_price = element.children[3].text

      # 終値
      closing_price = element.children[4].text

      # 出来高
      volume = element.children[5].text.gsub(/,/,'')

      puts "#{day},#{open_price},#{hight_price},
        #{low_price},#{closing_price},#{volume}"
    end
  }
end

# 証券コード
code="4689"

# 検索日
day=Time.now
ey=day.year
em=day.month
ed=day.day
start_url="http://info.finance.yahoo.co.jp/history
  /?sy=1900&sm=1&sd=1&ey=#{ey}&em=#{em}&ed=#{ed}&tm=
  d&code=#{code}"
num=1
puts "日付,始値,高値,安値,終値,出来高"
loop {
  url = "#{start_url}&p=#{num}"
  doc = get_nokogiri_doc(url)
```

```
  get_daily_data(doc)
  break if !has_next_page?(doc)
  num = num+1
}
```

◆ nokogiri-stock-history.rbの実行例

```
$ ruby nokogiri-stock-history.rb

日付,始値,高値,安値,終値,出来高
2014年7月3日,485,486,478,480,8037200
2014年7月2日,488,489,480,482,8357100
2014年7月1日,476,487,472,484,17689600

~中略~

1997年11月6日,1,036,288,1,093,632,1,028,096,1,028,096,440
1997年11月5日,966,656,1,085,440,876,544,1,019,904,1264
1997年11月4日,999,424,999,424,999,424,999,424,636
```

有償サービスの利用

上記のようにクローリングにより株価情報は取得できます。一方で、東証サイトにより、有償にて株価データのダウンロードやAPIの利用が可能です。法人のみならず個人でも利用可能なので、用途および頻度によっては検討すべきでしょう。

■ 東証データダウンロードサービス
URL http://ec.tse.or.jp/

5-15 為替情報・金融指標を収集する

株価に引き続き、為替などの金融指標も取得してみましょう。株価と金融指標を組み合わせて分析することにより、何か見えてくるかもしれません。

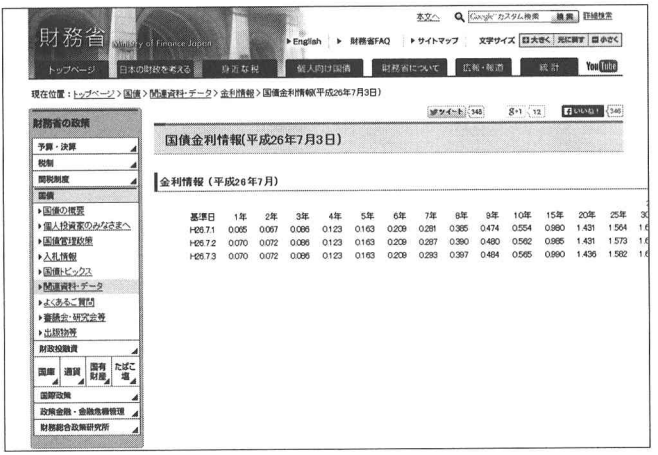
5-15-1 国債金利

国債金利については、財務省のページから取得できます。日々更新されており、CSVでも提供されています。データは日々更新され、月単位で切り替わります。

■ 国債金利情報：財務省

URL http://www.mof.go.jp/jgbs/reference/interest_rate/jgbcm.htm

国債金利情報



HTMLから取得する

最新のデータは各年ごとの表の最終行になります。HTML自体も入れ子のテーブル構造になっていて、ClassやIdなどの手がかりが少ないです。こういった場合は、配列で指定する方法しかないです。また、必要なのは最終行のみなので、Nokogiri::XML::Elementのlastメソッドを利用します。

国債金利の最新データを取得する　bond-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url="http://www.mof.go.jp/jgbs/reference/interest_rate/jgbcm.htm"
reg_pattern="//*[@id='index']/tbody/tr/td/ul/table[3]/tbody/tr"
doc = Nokogiri::HTML(open(url))
elements =  doc.xpath(reg_pattern).last
array=[]
elements.xpath("td").each {|element|
  array.push element.text
}

puts "基準日:#{array[0]}"
puts "1年:#{array[1]}"
```

```
puts "2年:#{array[2]}"
puts "3年:#{array[3]}"
puts "4年:#{array[4]}"
puts "5年:#{array[5]}"
puts "6年:#{array[6]}"
puts "7年:#{array[7]}"
puts "8年:#{array[8]}"
puts "9年:#{array[9]}"
puts "10年:#{array[10]}"
puts "15年:#{array[11]}"
puts "20年:#{array[12]}"
puts "25年:#{array[13]}"
puts "30年:#{array[14]}"
puts "40年:#{array[15]}"
```

◆ bond-nokogiri.rbの実行例

```
$ ruby bond-nokogiri.rb
基準日:H26.7.17
1年:0.049
2年:0.065
3年:0.079
4年:0.116
5年:0.154
6年:0.195
7年:0.270
8年:0.364
9年:0.458
10年:0.537
15年:0.967
20年:1.406
25年:1.569
30年:1.673
40年:1.779
```

CSVから取得する

スクリプトから使う場合はCSVを利用する方が便利かもしれません。RubyでCSVファイルを扱う場合、標準ライブラリであるCSVライブラリが利用できます。ダウンロードしたファイルを直接扱うには、IOインスタンスを扱えるCSVライブラリのインスタンスメソッドを選択する必要があります。newメソッドなど一部のみしか扱えないので注意してください。また、CSVの行数などは取得できないため、ループなどで最終行判定をする必要があります。

CSVから国債金利の最新データを取得する

bond-csv.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'csv'

url="http://www.mof.go.jp/jgbs/reference/interest_rate/jgbcm.csv"
csv = open(url)
csv_obj = CSV.new(open(url), {
  :encoding => "Shift_JIS", :headers => :first_row})
csv_obj.each do |row|
  # 最終行のみ表示
  if csv_obj.eof?
    # puts row
    puts "基準日:#{row[0]}"
    puts "1年:#{row[1]}"
    puts "2年:#{row[2]}"
    puts "3年:#{row[3]}"
    puts "4年:#{row[4]}"
    puts "5年:#{row[5]}"
    puts "6年:#{row[6]}"
    puts "7年:#{row[7]}"
    puts "8年:#{row[8]}"
    puts "9年:#{row[9]}"
    puts "10年:#{row[10]}"
    puts "15年:#{row[11]}"
    puts "20年:#{row[12]}"
    puts "25年:#{row[13]}"
    puts "30年:#{row[14]}"
    puts "40年:#{row[15]}"
  end
end
```

◆ bond-csv.rbの実行例

```
$ ruby bond-csv.rb
基準日:H26.7.3
1年:0.070
2年:0.072
3年:0.086
4年:0.123
5年:0.163
6年:0.209
7年:0.293
8年:0.397
9年:0.484
```

```
10年:0.565
15年:0.990
20年:1.436
25年:1.582
30年:1.683
40年:1.775
```

5-15-2 為替情報

為替については、Yahoo!ファイナンスで取得できます。

■米ドル/円-FXレート・チャート：Yahoo!ファイナンス
URL http://info.finance.yahoo.co.jp/fx/detail/?code=USDJPY=FX

◆米ドル/円 FXレート

Yahoo!ファイナンスでは、個々の主要な要素ごとにID属性が付いています。そのため、解析は非常に簡単に行えます。FXの場合は、Bid（売値）とAsk（買値）を取得します。

次のスクリプトは、実行時の米ドル/円-FXレートを取得します。

米ドル/円-FXレートを取得する　fx-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url = 'http://info.finance.yahoo.co.jp/fx/detail/?code=USDJPY=FX'
doc = Nokogiri::HTML(open(url))
bid = doc.xpath("//*[@id='USDJPY_detail_bid']").text
ask = doc.xpath("//*[@id='USDJPY_detail_ask']").text
puts "Bid(売値):#{bid}"
puts "Ask(買値):#{ask}"
```

◆ fx-nokogiri.rbの実行例

```
$ ruby fx-nokogiri.rb
Bid(売値):102.082
Ask(買値):102.086
```

5-15-3　その他の経済指標

国債の金利や為替は、経済の現在の状態を表すものです。それでは、市場は何をもって動いているのでしょうか。そのなかの1つに、各国の政府・中央銀行が発表する経済指標などがあります。例えば、雇用統計や貿易収支です。その値も取得してみましょう。Yahoo!ファイナンスでは、株価や為替情報のみならず、経済指標情報も取り扱っています。

■ 経済指標情報：Yahoo!ファイナンス
URL http://info.finance.yahoo.co.jp/fx/marketcalendar/

次のスクリプトは、Yahoo!ファイナンスから直近の経済指標を取得します。それぞれの経済指標には、指標自体の市場への影響の大きさや、市場の予想、実際の結果などがあります。どの指標を参考にするかは、個々人の判断となります。

経済指数情報を取得する　indicators-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

url = 'http://info.finance.yahoo.co.jp/fx/marketcalendar/'
doc = Nokogiri::HTML(open(url))
```

```
doc.xpath("//div[@class='ecoEventTbl02 marB20']/table/tr").each {|element|
  puts "指標名:#{element.xpath(".//td[@class='event']").text}"
  puts "予想:#{element.xpath(".//td[@class='expectation']").text}"
  puts "結果:#{element.xpath(".//td[@class='result']").text}"
}
```

◆ indicators-nokogiri.rbの実行例

```
$ ruby indicators-nokogiri.rb
指標名:5月 製造業新規受注 [前月比]
予想:-1.1%
結果:

~省略~
```

政府が発表する雇用統計などを扱う方法については、「5-20 官公庁のオープンデータを活用する」(→p.349)でもう少し詳しく解説します。

5-16 郵便番号と緯度経度情報を取得する

会員制のWebサイトなどを運営している場合、分析の一環として会員の地域ごとの分布を調べることがあります。その際は、郵便番号や住所を元に集計するといったことが多いでしょう。そして、集計結果はエクセルなどにまとめられるでしょう。ここで、エクセルなどのテキストベースの集計だけでなく、視覚化された情報があればより直感的に状況を把握できます。

住所情報に紐づく緯度経度の情報があれば、視覚化された情報を付与することができます。しかし、住所ごとの緯度経度の情報を持っている人は稀でしょう。そこで、Googleが提供している地図サービス「Google Maps」を利用し、緯度経度の情報を取得して活用してみましょう。

5-16-1 Google Maps APIによるジオコーディング

ジオコーディングとは、住所を地図上の地点に変換することを意味します。地点を指し示すには、緯度経度を利用します。必然的に、ジオコーディングをすると、緯度経度の情報を取得できます。Google Mapsは、住所や郵便番号からジオコーディングが行えます。

Google Mapsは住所や郵便番号から検索可能で、緯度経度を含む住所情報を返します。利用にあたってAPIキーは必須ではないものの、1つのIPあたり1日2500回のリクエストなどの制限があります。使用回数が多い「Maps API for Business」といった有償オプションもあります。

■ Google Maps API のライセンス - Google Maps API : Google Developers
URL https://developers.google.com/maps/licensing?hl=ja

Google Maps APIをRubyから利用する場合は、「geocoder」というGemライブラリがお勧めです。geocoderは、他にもYahoo!やBingなどの地図サービスの利用も可能です。インストールは、gemから行います。インストール後に、gem list geocoderでgeocoderが表示されれば成功です。

■ GitHubのalexreisner/geocoderページ
URL https://github.com/alexreisner/geocoder

◆ geocoderのインストール

```
$ gem install geocoder
```

◆ インストールの確認

```
$ gem list geocoder

*** LOCAL GEMS ***

geocoder (1.2.1)
```

5-16-2　郵便番号から緯度・経度を検索する

それでは、geocoderを使っての緯度・経度を取得します。住所もしくは郵便番号から検索できますが、住所の場合は表記ゆれが大きいので郵便番号を利用しましょう。郵便番号検索の場合、7桁の郵便番号を「3桁-4桁」の形にして検索します。

また、引数として国指定はありますが、場合によっては日本以外の結果を返す場合があります。そのため、最初にレスポンスが日本であるかの確認が必須になります。

次のスクリプトは、大阪市西区北堀江(550-0014)の位置情報を取得しています(❶)。指定する郵便番号を入れ替えると、その地点の情報が取得できます。

郵便番号から緯度・経度情報を取得する geocode.rb

```
# -*- coding: utf-8 -*-
require 'geocoder'

Geocoder.configure(:language =>
  "ja", :units => "km")

addresses = Geocoder.search("550-0014", ●1
  :params => {:countorycodes => "ja"})
addresses.each {|address|
  address.data["address_components"].each {|value|
    if value["short_name"] == "JP"
      lat = address.data
        ["geometry"]["location"]["lat"]
      lng = address.data
        ["geometry"]["location"]["lng"]
      puts "経度:#{lat}"
      puts "緯度:#{lng}"
    end
  }
}
```

◆ geocode.rbの実行例

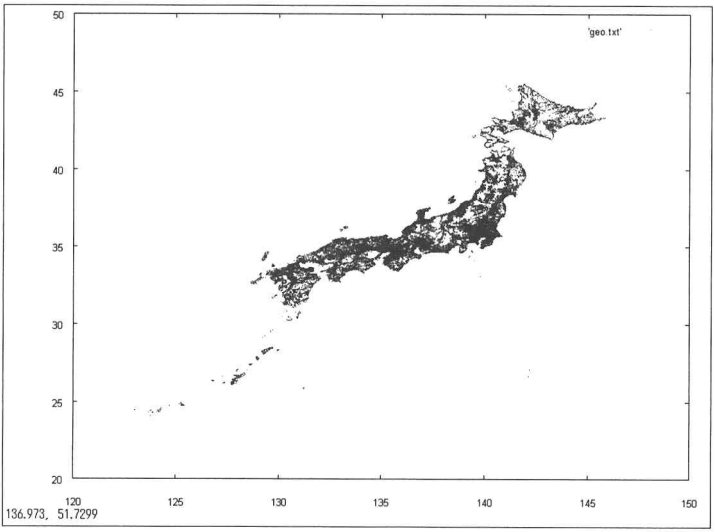
```
$ ruby geocode.rb
経度:34.6734205
緯度:135.4911445
```

5-16-3 郵便番号と緯度・経度データによる可視化

それでは、緯度・経度情報が集まれば、どのように視覚化できるのでしょうか。今回は、単純に郵便番号データの視覚化を行います。視覚化のツールについてはいろいろありますが、今回はWindows用の「Pajek」を利用しています。緯度経度情報は、Pajekの書式に従いポイントとして登録することにより次のような図を出力することができます。Pajekの使い方の説明は、省略させていただきます。

■ Pajek
URL http://vlado.fmf.uni-lj.si/pub/networks/pajek/

緯度・経度情報を視覚化する



単純に緯度経度情報を出すだけでは、あまり役に立ちません。実用的な例だと、位置情報(x,y)軸と売上情報(z)軸として組み合わせることにより、どの地域がどれくらい売れているのか立体的な地図に表すことができます。

5-17 新刊情報を収集する

クローラーの身近な利用例としては、書籍の新刊情報を収集することがあげられます。新刊情報のリストが定期的に届けば、面白そうな本を逃さず入手することもできます。また、お気に入りの作者やシリーズがあれば、アラートを出すことも可能です。それでは、Amazonから新刊情報を取得してみましょう。

5-17-1　Amazonの新刊・予約の検索パラメータ

Amazonの新刊・予約ページでは、カテゴリと期間を指定して検索を行うことができます。

■ Amazon.co.jpの新刊・予約ページ

URL http://www.amazon.co.jp/%E6%9C%AC-%E6%96%B0%E5%88%8A/b/ref=sv_b_2?ie=UTF8&node=2405051051

Amazon.co.jpの新刊・予約ページ

それでは、実際の検索URLはどのような形でしょうか。次のURLは、「文学・評論」の30日以内で検索した例です。

- http://www.amazon.co.jp/s/ref=sr_nr_n_0?rh=n%3A465392%2Cp_n_publication_date%3A2285541051%2Cn%3A%21465610%2Cn%3A466284&bbn=465610&ie=UTF8&qid=1401531073&rnid=465610

比較的難解です。検索パラメータに関する部分はrhパラメータのなかに詰め込まれて格納されています。rhパラメータ中の「%3A」や「%2C」など「%」から始まる3文字は、URIエンコードされたアスキーコードです。例えば%3Aは「:」、%2Cは「,」にあたります。その前提のうえで、カテゴリごとのURLを並べてみると、どの値が何を意味するかわかってきます。publication_dateが期間で、nがカテゴリになります。

検索期間のId

検索期間	Id
3日以内	2285919051
7日以内	2285539051
30日以内	2285541051
90日以内	82839051

◆検索期間のId（続き）

検索期間	Id
過去3日	2315442051
過去7日	2315443051
過去30日	82837051
過去90日	82838051

◆カテゴリ名のId

カテゴリ名	Id
文学・評論	571582
人文・思想	466284
社会・政治	571584
ノンフィクション	571584
歴史・地理	466286
ビジネス・経済	466282
投資・金融・会社経営	492054
科学・テクノロジー	466290
医学・薬学・看護学・歯科学	466298
コンピュータ・IT	466298
アート・建築・デザイン	466294
趣味・実用	466292
スポーツ・アウトドア	2400471051
資格・検定・就職	492228
暮らし・健康・子育て	466304
旅行ガイド・マップ	492090
語学・辞事典・年鑑	466302
教育・学参・受験	3148931
絵本・児童書	466306
コミック・ラノベ・BL	466280
タレント写真集	500592
ゲーム攻略・ゲームブック	92266
エンターテイメント	466296
楽譜・スコア・音楽書	746102
アダルト	10667101

5-17-2　新刊情報を取得する

パラメータの解析さえできると、データ取得はそれほど難しくありません。期間とカテゴリからURLを合成し、解析するだけです。ページングを再帰的に辿るた

めに、「次のページ」を取得するメソッドを用意します。

次のスクリプトは、過去30日間の「歴史・地理」カテゴリの新刊情報を取得します。

新刊情報を取得する new-books-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

# 検索期間 => 過去30日
search_term="82837051"

# 検索カテゴリ => 歴史・地理
category="466286"

base_url = "http://www.amazon.co.jp/s/
  ?rh=n%3A465392%2Cp_n_binding_browse-bin
  %3A86137051%2Cn%3A%2146561O%2C"
url="#{base_url}n%3A#{category}
  %2Cp_n_publication_date%3A#{search_term}"

def get_next_url(doc)
  url = nil
  element = doc.xpath("//a[@id='pagnNextLink']")
  if !element.empty?
    url = "http://www.amazon.co.jp/#{element.first[:href]}"
  end
  return url
end

loop {
  doc = Nokogiri::HTML(open(url))
  doc.xpath("//div[@id='atfResults']/div/h3/a").each {|element|
    puts element[:href]
    puts element.text
  }
  url = get_next_url(doc)
  break if url.nil?
}
```

♥ new-books-nokogiri.rbの実行例

```
$ ruby new-books-nokogiri.rb
原子野のトラウマ―被爆者調査再検証 こころの傷をみつめて
ttp://www.amazon.co.jp/%E4%B8%AD%E5%9B%BD%E3%82%A8%E3%82%B9%E3%83%8B%E3%83%83%E
%82%AF%E3%83%BB%E3%83%9E%E3%82%A4%E3%83%8E%E3%83%AA%E3%83%86%E3%82%A3%E3%81%AE%
5%AE%B6%E6%97%8F%E2%80%95%E5%A4%89%E5%AE%B9%E3%81%A8%E6%96%87%E5%8C%96%E7%B6%99
E6%89%BF%E3%82%92%E3%82%81%E3%81%90%E3%81%A3%E3%81%A6-%E6%97%A9%E7%A8%B2%E7%94%
0%E7%8F%BE%E4%BB%A3%E4%B8%AD%E5%9B%BD%E7%A0%94%E7%A9%B6%E5%8F%A2%E6%9B%B8-%E6%9
%B0%E4%BF%9D-%E6%95%A6%E5%AD%90/dp/4877912592
中国エスニック・マイノリティの家族―変容と文化継承をめぐって (早稲田現代中国研究
叢書)

~省略~
```

5-17-3　APIを利用する

新刊情報は、Amazon Product Advertising APIを利用すると、制約付きながらも取得できます。APIの検索オプションでpowerを指定すると、「during」というパラメータで月までの指定ができます。またsortオプションで日付順で指定できます。つまり当月もしくは翌月を指定すると、新刊情報を取得できます。

次のスクリプトは、Amazon Product Advertisingを扱うライブラリである、「amazon-ecs」を利用したサンプルです。「4-6 APIを利用した収集」(→p.242)で紹介したitem_searchメソッドでは、必ず検索ワードを指定する必要があります。そこで、よりローレベルのパラメータを投げられるsend_requestを利用し、検索を行っています。

amazon-ecsのインストールはgemから行えます。インストール後にgem list amazon-ecsでamazon-ecsが表示されれば成功です。

♥ amazon-ecsのインストール

```
$ gem install amazon
```

◆ インストールの確認

```
$ gem list amazon-ecs

*** LOCAL GEMS ***

amazon-ecs (2.2.5)
```

スクリプトを実行する際には、「:associate_tag」「:AWS_access_key_id」「:AWS_secret_key」を設定してください（→p.243）。

APIでAmazonの新刊情報を取得する

new-books-api.rb

```
# -*- coding: utf-8 -*-
require 'amazon/ecs'

Amazon::Ecs.options = {
  :associate_tag => 'sampleapp-22',
  :AWS_access_key_id => 'AWS_ACCESS_KEY',
  :AWS_secret_key => 'AWS_SECRET_ACCESS_KEY'
}

day = Time.now

power = "pubdate:during #{day.month}-#{day.year}"
res = Amazon::Ecs.send_request(
  {:operation => 'ItemSearch',:search_index => 'Books',:sort =>
'daterank',:country => 'jp', :power => power })

res.items.each do |item|
  puts item.get('ASIN')
  puts item.get('ItemAttributes/Title')
end
```

◆ new-books-api.rbの実行例

```
$ ruby new-books-api.rb
4862462332
1000 Dot-to-Dot 点つなぎ:街
4426606314
12日でマスター! "14~'15年版U-CANのFP技能士3級超速習レッスン&amp;amp;模試 (ユーキャンの資格試験シリーズ)
4426605393

~省略~
```

すべての新刊を取得するには、ページングなどの処理が必要です。ぜひ、実装を試してみてください。

5-18 荷物を追跡する

クローラーの技術を応用すると、荷物の追跡のようなこともスクリプトから実行できます。最近では、運輸会社各社がWebから簡単に追跡できるインターフェースを用意しています。そのため、ただスクリプトから追跡するだけでは、それほどメリットはありません。そこで、追跡後にその予定をGoogle Calendarに登録してみましょう。

5-18-1　ヤマト運輸の荷物を追跡する

ヤマト運輸を例にスクリプトを作ってみましょう。クロネコヤマトでは、次のURLに伝票番号を付加することにより、荷物の配送状況を問い合わせすることができます。

- http://jizen.kuronekoyamato.co.jp/jizen/servlet/crjz.b.NQ0010?id=伝票番号

問い合わせ後に、伝票番号とステータスと予定日の一覧が出てきます。まずは、それを取得してみましょう。問い合わせには、Postメソッドでデータを送る必要があります。そのため、Net::HTTPを利用して、パラメータをPostメソッドで送っています。

なお、実行時は「Input slip number」に伝票番号を設定してください（❶）。

ヤマト運輸の配送状況を取得する　yamato-nokogiri.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'net/http'

# コマンドライン引数から問い合わせ番号の取得
if ARGV.size == 1
  number =  ARGV[0]
else
  puts "Input slip number" ●―――――❶
  exit 1
end
```

```
Net::HTTP.version_1_2

url = "toi.kuronekoyamato.co.jp"
html = nil
Net::HTTP.start(url, 80) {|http|
  response = http.post('/cgi-bin/tneko',"number01=#{number}")
  html = response.body
}
if html.nil?
  exit
else
  html.encode!("UTF-8", "Shift_JIS")
end

doc = Nokogiri::HTML(html) if !html.nil?
doc.xpath("//table[@class='ichiran']/tr[3]").each {|element|

  day = element.xpath(".//*[@class='hiduke']/font").text
  status = element.xpath(".//*[@class='ct']/font").text
  puts day
  puts status
}
```

◆ yamato-nokogiri.rbの実行例

```
$ ruby yamato-nokogiri.rb 4106-9707-4561
07/18
配達完了
ヤマト配送:4106-9707-4561/配達完了 (h4no9065cgvq045fpi5qtv433o)
```

5-18-2 Google Calendarに登録する

それではデータを取得できたので、Google Calendarにデータを登録しましょう。RubyからGoogle Calendarの操作は、公式ライブラリである「Google API Client」を利用して行えます。しかし、かなり重量級のAPIなので、手軽に使うには少々面倒くさいです。そこで、Google Calendarの軽量ラッパーライブラリである「google_calendar」を利用してみましょう。

■ GitHubのgoogle_calendarページ
URL https://github.com/northworld/google_calendar

google_calendarのインストールはgemから行えます。インストール後にgem list

google_calendarで、google_calendarが表示されれば成功です。

◆ google_calendarのインストール

```
$ gem install google_calendar
```

◆ インストールの確認

```
$ gem list google_calendar

*** LOCAL GEMS ***

google_calendar (0.3.1)
```

google_calendarライブラリは、アクセストークンではなくGoogleのIDとパスワードで操作します。スクリプトの取り扱いは十分注意しましょう。先ほど作成したスクリプトに下記のメソッドを追加します。そして、日付とステータスを抜き出しているところから、呼び出すように変更します。

■配送情報をGoogle Calendarに登録する　yamato-google-calendar.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'net/http'
require 'google_calendar'

#コマンドライン引数から問い合わせ番号の取得
if ARGV.size == 1
  number =  ARGV[0]
else
  puts "Input slip number"
  exit 1
end

def set_google_calendar(number,status,day)
  cal = Google::Calendar.new(
    :username => 'GOOGLE_ID',
    :password => 'GOOGLE_PASS',
    :app_name => 'mycompany.com-googlecalendar-integration')
  today = Time.now
  day_of_month = day.split(/¥//)[1]
  month = day.split(/¥//)[0]
  date = Time.local(today.year, month, day_of_month, 12, 00, 00)
```

```
  event = cal.create_event do |e|
    e.title = "ヤマト配送:#{number}/#{status}"
    e.start_time = date
    e.end_time = date
  end

  # イベントの登録
  puts event
end

Net::HTTP.version_1_2

url = "toi.kuronekoyamato.co.jp"
html = nil
Net::HTTP.start(url, 80) {|http|
  response = http.post('/cgi-bin/tneko',"number01=#{number}")
  html = response.body
}
if html.nil?
  exit
else
  html.encode!("UTF-8", "Shift_JIS")
end

doc = Nokogiri::HTML(html) if !html.nil?
doc.xpath("//table[@class='ichiran']/tr[3]").each {|element|

  day = element.xpath(".//*[@class='hiduke']/font").text
  status = element.xpath(".//*[@class='ct']/font").text
  puts day
  puts status
  set_google_calendar(number,status,day)
}
```

◆ **yamato-google-calendar.rbの実行例**

```
$ ruby yamato-google-calendar.rb 4106-9707-4561
04/30
配達完了
ヤマト配送:9999-9999-9999/配達完了 (xxxxxxxxxxxxxxxxxxxxxxxxxxx)
            2014-04-30T03:00:00Z
            2014-04-30T03:00:00Z
```

5-19 不動産情報を取得する

過去の不動産の売買履歴を分析できれば、今買おうとしている物件が適正価格かどうかの判断の材料にできます。しかし、元データはどうすればよいのでしょうか？　不動産会社のサイトは、現在の物件しか公開していません。過去のデータが必要です。

不動産業界には、レインズと呼ばれる不動産会社各社の間で不動産情報をやり取りするネットワークシステムがあります。これを参照することで、過去分を含めて物件情報を見ることができます。

5-19-1　レインズからのデータ取得

レインズからデータを取得するには、基本的には会員不動産会社になる必要があります。一方で、限定的ながら一部の情報は誰でも取得可能になっています。以下のURLで、マンション・戸建住宅の売買価格・相場・取引事例の情報が公開されています。

■ 不動産取引情報提供サイト

URL http://www.contract.reins.or.jp/search/displayAreaConditionBLogic.do

▼ 不動産取引情報提供サイト

不動産取引情報提供サイトでは、全国の戸建て・マンションの売買履歴を取得できます。データの選択は、都道府県と地区を選択します。1ページにつき100件ずつ、

複数のページにわたって提供されます。そのなかで取得可能なデータは、次の表のとおりです。

◆ 不動産取引情報提供サイトの提供データ

項目名	概要・凡例
沿線	最寄りの沿線名
最寄り駅	最寄りの駅名
駅からの距離	徒歩○分以内
所在地	中央区日本橋本町
単価(万円/m^2)	○○万円
専有面積	60～80m^2
間取り	3LDK
築年	1980年から1981年
成約時期	2013年8月～2013年10月
用途地域	商業

動的サイトでかつセッションを元に動くサイトのため、Capybaraなどを利用してブラウザを操作してデータを取得します。

スクリプトの流れは、検索する都道府県と地域を選び、後はページがあるかぎりページングします。ページングは、プルダウンにより行われます。そのため、最初に全ページのリストを作成し、ループを元に1ページずつ処理するという実装にしています。

次のスクリプトでは、大阪府の北部を対象としています(❶)。都道府県・地区選択の部分も動的に選択するようにすると、全国のデータをすべて取得することも可能でしょう。しかし、サイト側の負荷も考慮して、必要なところのみ取得すべきです。

Capybaraのインストールについてはp.116、capybara-webkitについてはp.123をご参照ください。

■ 地域ごとの不動産取引情報を取得する　reins_capybara.rb

```
# -*- coding: utf-8 -*-
require 'capybara'
require 'capybara/dsl'
require 'capybara-webkit'

Capybara.default_selector = :xpath
Capybara.default_driver = :selenium
```

```
Capybara.app_host =
   "http://www.contract.reins.or.jp/search/displayAreaConditionBLogic.do"

module Spider
  class Reins
    include Capybara::DSL
    def initialize()
      @pages = []
      @current_page = 0
    end

    def crawle
      visit('')

      # 地域を選択する
      select('大阪府', :from => 'prefCodeA') ●―――――❶
      select('大阪市北部', :from => 'areaCodeA')
      click_on('検索する')

      # OKボタン
      page.driver.browser.switch_to.alert.accept

      # ページ番号の取得
      get_pages

      # ページごとに全データを取得する
      for i in 1..@pages.size-1
        scrape
        page_change(i)
      end
    end
    def scrape
      all("//*[@id='data05']/div[2]/table/tbody/tr").each {|element|

        if element.text !~ /^△沿線/
          # 物件情報の表示
          puts element.text
        end
      }
    end
    def get_pages
      all("//*[@id='data05']/div[1]/select/option").each {|element|
        @pages.push(element.text)
      }
    end
    def page_change(page_no)
      select(@pages[page_no], :from => 'listPageNum')
    end
```

```
  end
end

spider = Spider::Reins.new
spider.crawle
```

◆ reins_capybara.rbの実行例

```
$ ruby reins_capybara.rb
1 東海道・山陽新幹線 新大阪 徒歩5分以内 東淀川区東中島 13万円 0～20m2 ワンルーム 1974年から 1975年 2013年10月～ 2013年12月 -
2 東海道・山陽新幹線 新大阪 徒歩5分以内 淀川区宮原 39万円 0～20m2 1K 2000年から 2001年 2013年7月～ 2013年9月 -
3 東海道・山陽新幹線 新大阪 徒歩10分以内 東淀川区東中島 10万円 40～60m2 2DK 1973年から 1974年 2013年10月～ 2013年12月 -
4 東海道・山陽新幹線 新大阪 徒歩10分以内 淀川区木川東 18万円 60～80m2 3LDK 1979年から 1980年 2013年7月～ 2013年9月 一種住居
5 東海道・山陽新幹線 新大阪 徒歩10分以内 淀川区西宮原 23万円 20～40m2 ワンルーム 1972年から 1973年 2014年4月～ 2014年6月 一種住居

～省略～
```

築年数と坪当たりの単価まで取得できるので、経年での値下がりシミュレーションまで可能です。またパラメータとして駅からの距離を加えることにより、より詳細な分析ができることでしょう。

5-20 官公庁のオープンデータを活用する

「5-15 為替情報・金融指標を収集する」(→p.327)でも紹介しましたが、総務省統計局や独立行政法人統計センターには大量のデータが公開されています。これらのデータを活用すれば、わざわざクローリングせずにすむことも多いでしょう。一方で、公開されているデータの多くがExcel形式など、プログラムから扱いにくい形が多いのも事実です。しかし、最近ではオープンデータのかけ声とともに、次のような形で公開されるようになってきています。

- ①機械判読に適したデータ形式で、
- ②二次利用が可能な利用ルールで公開されたデータ

そのプロジェクトの1つに、次世代統計利用システムがあります。これは、数々の統計データをAPIを介して利用できるようにするシステムです。

■ 次世代統計利用システム
URL http://statdb.nstac.go.jp/

次世代統計利用システム

5-20-1　提供されているデータ一覧

次世代統計利用システムでは、さまざまなデータが提供されています。人口推移から労働力調査・国勢調査など各種の調査、消費者物価指数をはじめとする市況統計などが提供されています。

■ 提供データ（次世代統計利用システム）
URL http://statdb.nstac.go.jp/system-info/api/api-data/

提供される統計データは、次の表のとおりです。

▼政府統計コード一覧

政府統計コード	統計調査名
00200521	国勢調査
00200522	住宅・土地統計調査
00200523	住民基本台帳人口移動報告
00200524	人口推計
00200531	労働力調査
00200532	就業構造基本調査
00200533	社会生活基本調査
00200541	個人企業経済調査
00200543	科学技術研究調査
00200544	サービス産業動向調査
00200545	サービス業基本調査
00200551	事業所・企業統計調査
00200552	経済センサス－基礎調査
00200553	経済センサス－活動調査
00200561	家計調査
00200563	貯蓄動向調査
00200564	全国消費実態調査
00200565	家計消費状況調査
00200571	小売物価統計調査
00200572	全国物価統計調査
00200573	消費者物価指数
00200511	地域メッシュ統計
00200502	社会・人口統計体系（都道府県・市区町村のすがた）

5-20-2 次世代統計利用システムのAPI登録

次世代統計利用システムを利用するには、登録が必要です。登録には、メールアドレスが必要になります。まず仮登録で、メールアドレスを登録します。

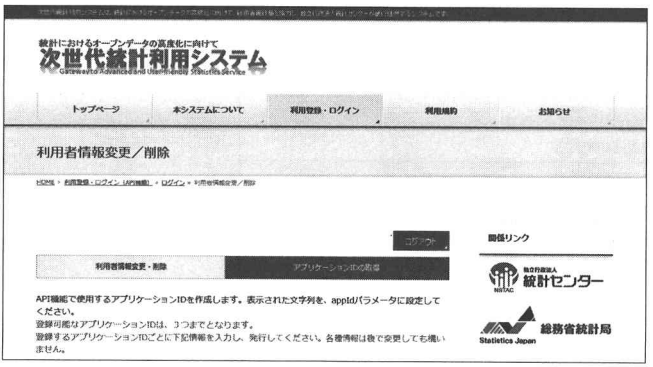
■次世代統計利用システムの利用登録
URL https://statdb.nstac.go.jp/apiuser/php/index.php?action=provisional

仮登録すると、登録したメールアドレスに本登録用のURLが送付されます。そのURLをブラウザで開いて、パスワードや利用者情報を入力すると登録完了となります。

また、登録後にアプリケーションIDの取得が必要になります。アプリケーション

IDは、ログイン後の「利用者情報変更/削除」画面から、［アプリケーションIDの取得］タブを選択することで取得できます。

▼次世代統計利用システムのアプリケーションID取得



アプリケーションIDの取得には、アプリケーション登録が必要です。アプリケーション名とURL・概要の登録が必要となります。URLについては、外部公開するものでなければ、「http://localhost/」などで登録しておけばよいようです。

入力完了して［登録］ボタンを押すと、アプリケーションIDが発行されます。今後利用するので、保存しておきましょう。

5-20-3　次世代統計利用システムのAPIの利用

アプリケーションIDが取得できれば、あとは比較的簡単にAPIを利用できます。APIは、基本的に次の形で利用できます。まずはブラウザなどで、URLを貼り付けて結果を確認してみましょう。

- http://statdb.nstac.go.jp/api/1.0b/app/getStatsList?appId=アプリケーションID

レスポンスは、XML形式で返ってきます。検索IDなどを指定しない場合は、すべての統計リストが返ってきます。今後、このレスポンス内の<LIST_INF>タグにあるIDを利用して、それぞれの統計を取得することになります。まずは、統計名とIDを一括で取得するスクリプトを作成してみましょう。

APIのレスポンス

この XML ファイルにはスタイル情報が関連づけられていないようです。以下にドキュメントツリーを表示します。

```
−<GET_STATS_LIST xsi:noNamespaceSchemaLocation="http://statdb.nstac.go.jp/api/1.0b/schema/GetStatsList.xsd">
 −<RESULT>
    <STATUS>0</STATUS>
    <ERROR_MSG>正常に終了しました。</ERROR_MSG>
    <DATE>2014-07-04T16:24:56.892+09:00</DATE>
  </RESULT>
 −<PARAMETER>
    <LANG>J</LANG>
  </PARAMETER>
 −<DATALIST_INF>
    <NUMBER>28208</NUMBER>
   −<LIST_INF id="0000030001">
      <STAT_NAME code="00200521">国勢調査</STAT_NAME>
      <GOV_ORG code="00200">総務省</GOV_ORG>
      <STATISTICS_NAME>昭和55年国勢調査 第1次基本集計 全国編</STATISTICS_NAME>
     −<TITLE no="00101">
        男女の別(性別)（3），年齢5歳階級（23），人口 全国・市部・郡部・都道府県（47），全域・人口集中地区の別
      </TITLE>
      <CYCLE>−</CYCLE>
      <SURVEY_DATE>198010</SURVEY_DATE>
      <OPEN_DATE>2007-10-05</OPEN_DATE>
      <SMALL_AREA>0</SMALL_AREA>
    </LIST_INF>
   −<LIST_INF id="0000030002">
      <STAT_NAME code="00200521">国勢調査</STAT_NAME>
      <GOV_ORG code="00200">総務省</GOV_ORG>
      <STATISTICS_NAME>昭和55年国勢調査 第1次基本集計 全国編</STATISTICS_NAME>
     −<TITLE no="00102">
        男女の別(性別)（3），年齢各歳階級（103），人口 全国・市部・郡部・都道府県（47），全域・人口集中地区の別
      </TITLE>
      <CYCLE>−</CYCLE>
      <SURVEY_DATE>198010</SURVEY_DATE>
      <OPEN_DATE>2007-10-05</OPEN_DATE>
      <SMALL_AREA>0</SMALL_AREA>
    </LIST_INF>
   −<LIST_INF id="0000030003">
      <STAT_NAME code="00200521">国勢調査</STAT_NAME>
      <GOV_ORG code="00200">総務省</GOV_ORG>
      <STATISTICS_NAME>昭和55年国勢調査 第1次基本集計 全国編</STATISTICS_NAME>
```

次世代統計利用システムから統計情報を取得する

statistics-list.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'rexml/document'

xml = open(
  "http://statdb.nstac.go.jp/api/1.0b/
   app/getStatsList?appId=#{'STATISTICS_API'}")

doc = REXML::Document.new(xml)

# puts doc
doc.elements.each(
  'GET_STATS_LIST/DATALIST_INF/LIST_INF') { |element|

  puts "ID:#{element.attributes["id"]}"
  puts "統計名:#{element.elements["STATISTICS_NAME"].text}"
  puts "タイトル:#{element.elements["TITLE"].text}"
}
```

statistics-list.rbの実行例

```
$ ruby statistics-list.rb
ID:0000030001
統計名:昭和55年国勢調査 第1次基本集計 全国編
タイトル:男女の別(性別)（3），年齢５歳階級（２３），人口 全国・市部・郡部・都道府県（４７），全域・人口
```

```
集中地区の別
ID:0000030002
統計名:昭和55年国勢調査 第1次基本集計 全国編
タイトル:男女の別(性別)(3)，年齢各歳階級(１０３)，人口 全国･市部･郡部･都道府県(４７)，全域･人口集中地区の別
ID:0000030003
統計名:昭和55年国勢調査 第1次基本集計 全国編
タイトル:総人口･日本人(2)，男女の別(性別)(2)，年齢各歳階級(１２２)，出生の月(5)，人口 全国･市部･郡部･都道府県(４７)，全域･人口集中地区の別

~省略~
```

実行すると、おそらく数分から数十分程度かかるでしょう。すべての統計リストを取得するために、膨大なサイズのXMLになっています。2014年7月時点で、3万件近い統計が登録されています。取得した統計のIDを利用すれば、個々の統計データが取得できます。統計IDは、次のように、statsDataIdを付加したURLからも取得できます。

- http://statdb.nstac.go.jp/api/1.0b/app/getStatsData?appId=アプリケーションID&statsDataId=統計ID

先ほど取得した統計IDのリストを元に、「昭和55年国勢調査 第1次基本集計 全国編」をブラウザで表示してみましょう。統計IDは「0000030001」なので、URLは次のような形になります。アプリケーションIDの部分を自分のIDに置き換えてください。

- http://statdb.nstac.go.jp/api/1.0b/app/getStatsData?appId=アプリケーションID&statsDataId=0000030001

レスポンスのXMLのなかには、データの定義から実際のデータまですべて記載されています。必要に応じて利用してみましょう。

5-21 新聞の見出しを集める

ニュースサイトからニュースのタイトルとURLを取得するスクリプトを作成しましょう。取得自体は、ここまでやってきたことと同じです。そこで各社ごとの差異を吸収し、メンテナンスしやすい形で作成してみます。

5-21-1 取得対象とプログラムの構造

取得対象のサイトは、読売新聞・朝日新聞・日本経済新聞の3社とします。各サイトから、新着情報のみ取得することとします。URLは、次のとおりとなっています。

■ 読売新聞
URL http://www.yomiuri.co.jp/latestnews/

■ 朝日新聞
URL http://www.asahi.com/news/

■ 日経新聞
URL http://www.nikkei.com/news/category/

後から対象とするサイトを追加する前提で、付け加えやすい構造にします。メイン処理を記述したスクリプトと、サイトごとのクローリングやスクレイピングの差異を記述したサイト定義のスクリプトという形にします。対象サイトが増えるとサイト定義スクリプトを追加するという形です。

- 呼び出し元スクリプト：news-main.rb
- サイトの親クラス：news-site.rb
- サイトごとの実装：news-asashi.rb/news-yomiuri.rb/news-nikkei.rb

5-21-2 親クラスの実装

まず各サイトの共通部分を抽出した親クラスを作成します。共通化する部分は、初期化メソッドとクロール先のURLとスクレイピングです。今回は、クロール自体は呼び出し元のスクリプトで実装するので、サイトに関する情報のみを記述します。

まず初期化メソッドでは、変数の初期化の他に、委譲される各サイトの実装が必

要なメソッドを備えているかのチェックを行います。ここでチェックをすることで、将来サイトを追加した際に必要な機能を備えていない場合に、問題を検知することができます。スクレイピング処理については、各サイトごとの実装に依存する部分が多いので、呼び出すだけにします。

親クラス　news-site.rb

```
class NewsSite
# アクセサメソッド
attr_reader :base_url
attr_reader :url

def initialize(site)
  @site = site
  @base_url = site.base_url
  @url = site.url

  # サイトごとの実装クラスが
  # 必要なメソッドを備えているかチェック
  methods.each do |method|
    if !@site.respond_to?(method.to_sym)
      raise "Site adapter must support method #{method}"
    end
  end
end

def scrape(doc)
  @site.scrape(doc)
end
```

5-21-3　各社別の記事・URLの抜き出し

読売新聞を例に、各サイトごとの実装を行います。ここでは、取得先のサイトの定義と、スクレイプ処理の実装を行います。

サイトごとの実装　news-yomiuri.rb

```
class Yomiuri
  attr_reader :base_url
  attr_reader :url

  def initialize()
    @base_url = "http://www.yomiuri.co.jp/latestnews/"
    @url = @base_url
  end
```

```
  def scrape(doc)
    titles={}
    reg_pattern =
      "//div[@class='pbNested row-contents']/div/div/ul/li/a"
    doc.xpath(reg_pattern).each {|element|
      url = element.text.gsub(/¥r¥n|¥t|¥s/,"")
      title = "#{element[:href]}"
      titles.store(url.to_s, title)
    }
    return titles
  end
end
```

5-21-4 呼び出し元の実装

最後に、呼び出し元のプログラムの実装を行います。呼び出し元では、与えられたURLを元にサイトのクロールを行います。スクレイピング処理の実装は、先ほど作成したクラスを利用します。news-site.rbおよびnews-yomiuri.rbは、同じディレクトリ内に配置してください。

呼び出し元のスクリプト news-main.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'
require 'nokogiri'

require './news-site.rb'
require './news-yomiuri.rb'

def get_nokogiri_doc(url)
  begin
    html = open(url)
  rescue OpenURI::HTTPError
    return
  end
  Nokogiri::HTML(html.read, nil, 'utf-8')
end

site = NewsSite.new(Yomiuri.new)
doc = get_nokogiri_doc(site.url)
titles = site.scrape(doc)
titles.each {|key,value|
  puts key
  puts value
}
```

◆ news-main.rbの実行例

```
$ ruby news-main.rb
「露からプク」証拠映像あるとウクライナ大統領（2014年07月20日21時56分）
http://www.yomiuri.co.jp/national/20140720-OYT1T50115.html
落雷の東急多摩川線、運転を再開…池上線も（2014年07月20日21時50分）
http://www.yomiuri.co.jp/world/20140720-OYT1T50130.html
撃墜機にマレーシア・ナジブ首相の祖父の妻も（2014年07月20日21時46分）
http://www.yomiuri.co.jp/politics/20140720-OYT1T50129.html
陸自調達のオスプレイ、佐賀空港配備へ…防衛相（2014年07月20日21時43分）
http://www.yomiuri.co.jp/national/20140720-OYT1T50128.html
関東に猛雨と落雷、交通乱れや停電相次ぐ（2014年07月20日21時32分）

～省略～
```

このような形にしておけば、処理対象の追加や変更は簡単です。それぞれのスクリプトにどの機能を持たすかが設計のポイントになりますが、機能・目的に依存する部分が大きくなります。試行錯誤しながら、最適な配置を目指しましょう。

5-21-5　ページング機能の追加

日経新聞については、複数のページに分かれています。今回作成したスクリプトに、ページングの機能を追加してみましょう。news-site.rbにページの取得メソッドを追加します。

```
def get_next_url(doc)
  @site.get_next_url(doc)
end
```

■ 親クラスにページング機能を追加する　　news-site-paging.rb

```
class NewsSite
  # アクセサメソッド
  attr_reader :base_url
  attr_reader :url

  def initialize(site)
    @site = site
    @base_url = site.base_url
    @url = site.url

    # サイトごとの実装クラスが
    # 必要なメソッドを備えているかチェック
```

```
    methods.each do |method|
      if !@site.respond_to?(method.to_sym)
        raise "Site adapter must support method #{method}"
      end
    end
  end

  def scrape(doc)
    @site.scrape(doc)
  end

  def get_next_url(doc)
    @site.get_next_url(doc)
  end
end
```

次に、各サイトのスクリプトに実装をします。改ページ処理がある日経新聞とそれ以外については、別々の実装をします。

コードの全文はサンプルファイル（→p. ii）をご参照ください。

日経新聞への対応 news-nikkei-paging.rb

```
def get_next_url(doc)
  next_url = nil
  element = doc.xpath("//li[@class='cmnc-next']/a")
  if !element.empty?
    next_url = "#{base_url}#{element.first[:href]}"
  end
  return next_url
end
```

その他の新聞への対応 news-yomiuri-paging.rb/news-asahi-paging.rb

```
def get_next_url(doc)
  return nil
end
```

最後に、呼び出し元プスクリプトを修正します。ループ処理を追加し、次のURLがあるかぎり処理を継続します。

呼び出し元スクリプトの修正　news-main-paging.rb

```
site = NewsSite.new(Nikkei.new)
url = site.url
loop {
  doc = get_nokogiri_doc(url)
  titles = site.scrape(doc)
  titles.each {|key,value|
    puts key
    puts value
  }
  url = site.get_next_url(doc)
  break if url.nil?
}
```

さらなる拡張としては、スクレイピングの結果を格納するクラスの作成や、サイトの文字コードの定義などいろいろあります。必要に応じて、追加してみましょう。

Chapter 6

クローラーの運用

6-1 サーバサイドで動かす

ここまで、WindowsやMacなどパーソナルコンピュータ上での実行を前提としてきました。クローラーを本格的に運用する場合、サーバサイドで動かすことのメリットが多数あります。サーバで動かすことの意味や、サーバを扱う際の考え方、テクニックを紹介します。

6-1-1　サーバで動かすメリット

サーバの言葉としての意味は3つあります。1つ目は、コンピュータ機器として、パーソナルコンピュータに対してのサーバです。2つ目は、ソフトウェアとしてクライアント・サーバモデルのサービスを提供する側のプログラムであるサーバです。HTTPサーバやFTPサーバなどがあります。

3つ目がオペレーションシステム(OS)として、Windows 7やMac OS Xなどクライアントに対しての、サーバOSです。Windows Server 2012などのWindows系と、RedHat Enterprise LinuxやCentOSといったLinux系などがあります。ここでは、主にOSとしてのサーバを対象に考えていきます。特にLinux系を対象とし、CentOSを前提とします。

柔軟なクローラー運営

それでは、サーバ上でクローラーを動かすメリットは何があるのでしょうか？　例えば、Linuxであればスケジューリングでの実行や、豊富なアプリケーションとの連携などがあります。サーバは基本的に、24時間の稼働を前提としています。そのため、1日中クローラーを動かすことや、毎日決まった時間にクローラーを動かして最新の情報を収集することができます。スケジューリングがしやすいということです。

また、スケジューリングするために、Crondデーモンのようなスケジュール管理プログラムが標準で組み込まれています。24時間稼働とスケジュール管理プログラムを組み合わせることで、柔軟なクローラー運営ができます。なお、Windows 7などWindows系クライアントOSでもタスクスケジューラというスケジュール管理プログラムが標準で組み込まれていて、ほぼ同様のことが行えます。またBSD UNIXベースであるMac OS Xも、同様にCrondデーモンが用意されています。

次に豊富なアプリケーションについてです。GUI系のアプリケーションであれば、

WindowsやMacの方が充実しています。一方でサーバサイドで動くアプリケーションであれば、Linux系サーバが充実しています。クローラーを運用する場合は、クライアントアプリではなくサーバサイドのアプリが必要な場合が多いです。そういった点でも、クローラーはサーバ上で動かすのがよいでしょう。

身近になったサーバOS

今までであれば、個人でサーバ環境を用意してLinuxなどを利用するのは、多少の困難を伴いました。しかし、昨今のクラウドの普及や、Vagrantのような仮想化環境の進化で、個人でも簡単にサーバが利用できるようになっています。

利用へのハードルが下がっているので、手軽に試してみるのがよいでしょう。次のセクションで、サーバOSのインストールからログインまでの手順を紹介します。

6-1-2 サーバへのインストール

ここからは、Linuxのディストリビューションの1つであるCentOS上でクローラーを動かします。しかし、実際にCentOSをインストールする環境を手元に用意できないケースもあるので、ローカルPCのなかに仮想化ソフトウェアである「VirtualBox」と、仮想化環境構築ツールである「Vagrant」を使って作業を進めます。Vagrantを使ってCentOSのインストールまで行えば、あとは実際のサーバであろうが仮想環境であろうが、同じ操作で扱えます。

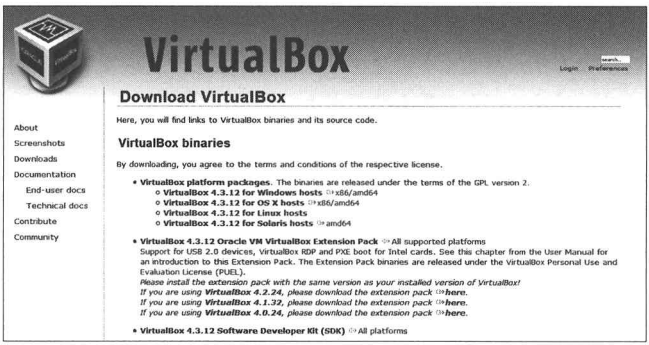
VirtualBox と Vagrant のインストール

まずVirtualBoxをインストールします。VirtualBoxは、Oracle社製の仮想化ソフトウェアで、既存のOS上で仮想のOS（ゲストOS）を実行できます。ゲストOSとしてサポートされているものは、LinuxやWindows、Mac OS Xなど多岐にわたります。なお、ゲストOSとして起動する場合も、OSのライセンスは必要となります。それぞれのライセンスに従って、正しく動かしましょう。

VirtualBoxは、次のURLからダウンロードできます。自分のクライアントPCのOSに従って、ダウンロードしましょう。

■ VirtualBox
URL https://www.virtualbox.org/wiki/Downloads

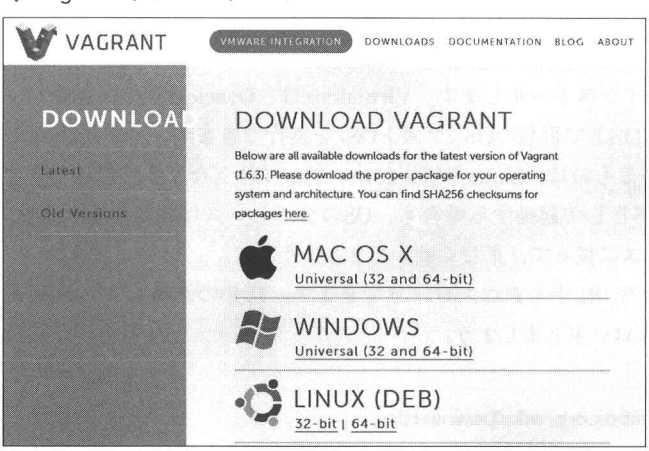
VirtualBoxのダウンロードページ



Vagrantは、仮想化ソフトウェアを操作するためのツールです。直接VirtualBoxを操作することも可能ですが、Vagrantを介することでコマンドラインで簡単に操作でき、プラグインと組み合わせることによりさまざまな機能を追加することができます。

■ Vagrant
URL **http://www.vagrantup.com/downloads.html**

Vagrantのダウンロードページ



手元に利用可能なサーバがある場合は、Rubyのインストール（→p.371）まで飛ばしてください。

Windowsへのインストール

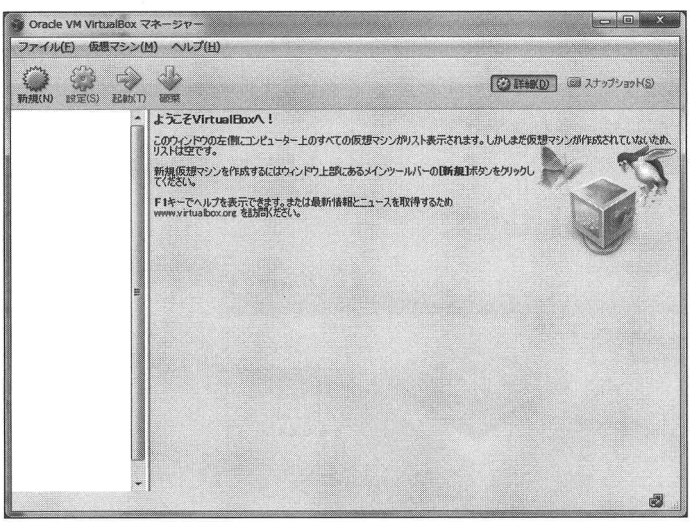
WindowsへのVirtualBoxのインストールは、ウィザードに従ってほぼ自動で終了します。インストール完了後に再起動が必要になります。

インストール完了後に、パスを通します。環境変数の設定画面で、Path変数にVirtualBoxのパスを追加します。通常であれば、次のようなURLです。インストール先のフォルダーに合わせて調整してください。

- C:¥Program Files¥Oracle¥VirtualBox

設定後にコマンドプロンプトを立ち上げて、**virtualbox**と入力して[Enter]キーを押します。GUIの画面が表示されるとインストールは成功です。

VirtualBoxの画面(Windows)



次にVagrantのインストールをします。Vagrantのインストールも、ウィザードに従って行えば、すぐに終わります。インストール後はシステムの再起動が求められます。

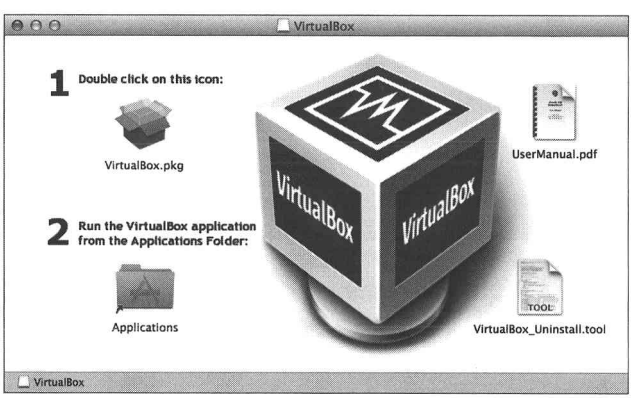
Vagrantのインストール(Windows)

Macへのインストール

Macへのインストールは、ダウンロード後にdmgイメージファイルをダブルクリックして起動します。そして、VirtualBox.pkgをダブルクリックしてインストールウィザードを開始します。ウィザードに従ってインストールすると、アプリケーションフォルダーにコピーされます。

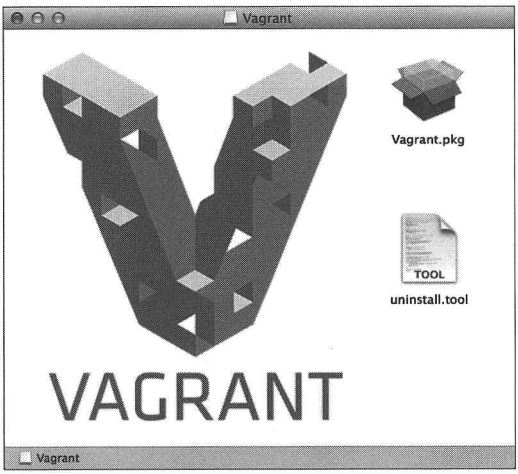
インストール後に、ターミナルからvirtualboxと入力して、GUIの画面が立ち上がってくると成功です。

VirtualBoxの画面(Mac)



続いて、Vagrantのインストールを行います。Vagrantの場合も、dmgイメージファイルをダブルクリックして起動します。そして、Vagrant.pkgをダブルクリックしてインストールウィザードを開始します。

Vagrantのインストール(Mac)



インストール完了後に、ターミナルでvagrant --versionと入力して、次のように表示されれば成功です。

◆ インストールの確認

```
$ vagrant --version
Vagrant 1.6.3
```

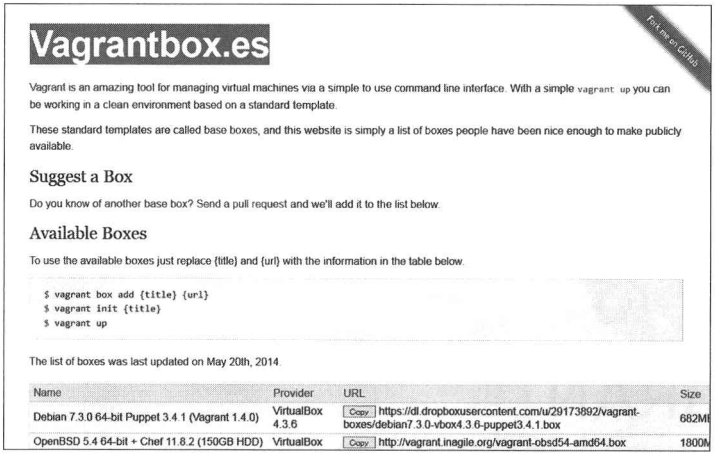
ゲストOSのインストール

準備ができたところで、Vagrantを使ってVirtualBox上に仮想OSをインストールしましょう。仮想OSを作成するには、元となるマシンイメージが必要となります。そのうえで、VirtualBoxの設定を行い、インストールする必要があります。

Vagrantを使うと、マシンイメージのダウンロードから設定までがコマンドラインから実行できます。それでは、まずはインストールするOSのマシンイメージを探しましょう。Vagrantで利用可能なマシンイメージは、次のサイトで一覧になっています。もちろん、ここ以外にもたくさん利用可能な仮想イメージは存在します。必要に応じて探すとよいでしょう。

■ Vagrantbox.esページ
URL http://www.vagrantbox.es/

マシンイメージを探す



今回は、CentOS 6.5を利用します。イメージ名は「CentOS 6.5 x86_64」でURLは、次のとおりです。

■ CentOS 6.5のイメージファイル

URL **https://github.com/2creatives/vagrant-centos/releases/download/v6.5.3/centos65-x86_64-20140116.box**

もし存在しない場合は、別の似たイメージを探して利用してください。

Vagrantの使い方は、マシンイメージの登録と設定、起動の3つのフェーズです。まず登録は、任意のイメージ名と仮想イメージのURLを設定します。次にinitでマシンイメージに対して設定を行います。完了すると、vagrant upで仮想マシンを起動できます。

構文 マシンイメージの登録

```
vagrant box add {title} {url}
```

構文 マシンイメージの設定

```
vagrant init {title}
```

構文 マシンイメージの起動

```
vagrant up
```

次のコマンドが実際の設定です。イメージ名は「centos65」にしています。

◆ マシンイメージの登録

```
$ vagrant box add centos65 https://github.com/2creatives/vagrant-centos/releases/download/v6.5.3/
centos65-x86_64-20140116.box
==> box: Adding box 'centos65' (v0) for provider:
    box: Downloading: https://github.com/2creatives/vagrant-centos/releases/download/v6.5.3/
centos65-x86_64-20140116.box
==> box: Successfully added box 'centos65' (v0) for 'virtualbox'!
```

◆ マシンイメージの設定

```
$ vagrant init centos65
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

◆ マシンイメージの起動

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'centos65'...
==> default: Matching MAC address for NAT networking...

~中略~

==> default: Mounting shared folders...
    default: /vagrant => /Users/takuro/Documents/centos
```

無事インストール完了できたら、さっそくログインして動作確認をしましょう。Macの場合は、vagrant sshでログインできます。

◆ ゲストOSへのログイン

```
[vagrant@vagrant-centos65 ~]$ vagrant ssh
[vagrant@vagrant-centos65 ~]$
```

エラーなくコマンドプロンプトが表示されれば成功です。

Windowsの場合は、vagrant sshコマンドが利用できません。TeraTermなどのSSHクライアントソフトが必要になります。また、IPアドレス指定で接続できるように、Vagrantの設定が必要になります。vagrant initで作成されたVagrantfile（カレントフォルダーに作成されます）をテキストエディタなどで開き、「config.vm.network」のコメントアウトを外します。これにより、Vagrantで立ち上げるOSのIPアドレスが、設定されます。デフォルトでは「192.168.33.10」になっているので、都合に合わせて変更しましょう。

```
config.vm.network "private_network", ip: "192.168.33.10"
```

設定後に仮想マシンの再起動を行います。再起動は、vagrant reloadを利用します。

◆ 仮想マシンの再起動

```
C:¥work vagrant reload
==> default: Attempting graceful shutdown of VM...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...

～省略～
```

あとは、SSHクライアントから「192.168.33.10」へ接続するように設定します。なお、デフォルトでは、接続時のユーザー名とパスワードともに「vagrant」となっています。以後は、SSHクライアント経由で動かします。

仮想マシンの停止は、vagrant haltコマンドを利用します。

◆ 仮想マシンの停止

```
[vagrant@vagrant-centos65 ~]$ vagrant halt
```

開発ツールとRubyのインストール

無事CentOSにログインできたら、次は開発ツールとRubyのインストールを行います。開発ツールとは、コンパイルするためのmakeやgccを中心とした必要なライブラリ群になります。1つずつインストールするのは大変なので、ここではグループインストール機能で一括でインストールします。

◆ **開発ツールのインストール**

```
[vagrant@vagrant-centos65 ~]$ sudo yum groupinstall "Development Tools"
```

開発者ツールがインストールできたら、次はRubyをインストールします。Rubyのインストール方法はいろいろありますが、今回は複数のバージョンを切り替えられる「rvm」を利用します。また、実行中のユーザーの領域にインストールされるので、root権限がない場合でも利用できます。rvmのインストールは、公式ページに従いインストールシェルをダウンロードして直接実行します。

■ rvm
URL **http://rvm.io/rvm/install**

◆ **rvmのインストール**

```
[vagrant@vagrant-centos65 ~]$ ¥curl -sSL https://get.rvm.io | bash
Downloading https://github.com/wayneeseguin/rvm/archive/master.tar.gz

~中略~

In case of problems: http://rvm.io/help and https://twitter.com/rvm_io

 * WARNING: You have '~/.profile' file, you might want to load it,
   to do that add the following line to '/home/vagrant/.bash_profile':

     source ~/.profile
```

インストールが完了したら、指示に従いbashの設定を再読み込みします。これでrvmへのパスが設定されます。rvm versionでバージョン番号が表示されれば成功です。

◆ **bashの設定を再の読み込み**

```
[vagrant@vagrant-centos65 ~]$ source ~/.profile
```

◆ インストールの確認

```
[vagrant@vagrant-centos65 ~]$ rvm version
rvm 1.25.27 (master) by Wayne E. Seguin <wayneeseguin@gmail.com>, Michal Papis <mpapis@
gmail.com> [https://rvm.io/]
```

最後に、Ruby自体のインストールを行います。インストールは、Rubyのコンパイルが行われるため、非常に時間がかかります。

◆ Rubyのインストール

```
[vagrant@vagrant-centos65 ~]$ rvm install 2.0.0
```

インストールが完了したら、rvm useコマンドで使用するRubyのバージョンを設定します。--defaultを付けることで、デフォルト設定ができます。次回以降、わざわざ使用するバージョンを選択する必要がなくなります。

◆ Rubyのバージョン設定

```
[vagrant@vagrant-centos65 ~]$ rvm use 2.0.0 --default
```

Rubyのバージョンを表示させて、エラーがないことを確認すれば完了です。

◆ インストールの確認

```
[vagrant@vagrant-centos65 ~]$ ruby --version
ruby 2.0.0p481 (2014-05-08 revision 45883) [x86_64-linux]
```

なお、これ以降のコマンド表記では、「[vagrant@vagrant-centos65 ~]」は省略します。

6-1-3　Linuxのコマンド

Linuxを扱うには、さまざまなコマンドを覚える必要があります。ここで最小限のコマンドを紹介します。

コマンド一覧

よく使うコマンドを紹介します。使い方は、マニュアルを参照してください。

◆よく使うコマンド

コマンド	機 能
ls	ファイル・ディレクトリの一覧を表示する
cp	ファイルをコピーする
pwd	今現在（カレント）のディレクトリを表示する
cd	ディレクトリを移動する
mkdir	ディレクトリを作成する
rm	ファイルを削除する。ディレクトリの場合は、-rを付ける
ps	実行中のプロセスを表示する
history	コマンド履歴を表示する
cat	ファイルを表示する
vi	ファイルを編集する
echo	文字列や変数を表示する
exit	ログアウトする
grep	正規表現で文字列を検索する
head	ファイルの先頭部分を表示する
tail	ファイルの末尾部分を表示する
man	コマンドのマニュアルを表示する
sudo	rootユーザー権限で実行する
wc	ファイルの行数をカウントする
which	コマンドのパスを表示する

コマンドの使い方を調べる

コマンドの使い方を表示するには、manコマンドを利用します。manの後に調べたいコマンドを入力することで、多くの場合はそのコマンドのマニュアルを表示させることができます。

◆ マニュアルを表示する

```
$ man cp

CP(1)       User Commands          CP(1)

NAME
    cp - copy files and directories

SYNOPSIS
    cp [OPTION]... [-T] SOURCE DEST
    cp [OPTION]... SOURCE... DIRECTORY
    cp [OPTION]... -t DIRECTORY SOURCE...

~省略~
```

これ以外にも、コマンドごとにヘルプオプションが用意されている場合があります。ただし、ヘルプオプションの指定の仕方は、コマンドごとに異なります。多くの場合は、--helpなので、まずはそのように入力してみましょう。例えオプションの指定方法が違っても、ヘルプの見方を教えてくれる場合があります。

◆ ヘルプオプションを表示する

```
$ cp --help
```

ディレクトリの移動

コマンドのなかで多く使うもののうちに、cdコマンドがあります。cdの後に対象ディレクトリを指定することで、任意の場所に移動できます。Linuxが初めての人は、まずはこのコマンドを使いこなしましょう。

ディレクトリを移動するコマンド

コマンド	機　能
cd ../	1つ上のディレクトリに移動する
cd /	rootディレクトリに移動する
cd ~/	ホームディレクトリに移動する
cd ~/tmp	ホームディレクトリの下のtmpディレクトリに移動する
cd -	直前にいたディレクトリに戻る

実行結果の表示

echoコマンドは、文字列や変数を表示するコマンドです。変数のなかには、バッシュなどの組み込みの変数も含まれています。これを利用することにより、アプリの実行結果などを確認できます。

実行結果を表示するコマンド

コマンド	機　能
./tmp.sh	何らかの処理を実行する（この例では「tmp.sh」を実行）
echo $?	直前の実行コマンドの返り値を表示する。「0」の場合は正常終了。それ以外は異常終了

6-2 定期的にデータを収集する

クローラーをサーバOSで動かすことのメリットの1つとして、スケジューリングのしやすさがあります。Linuxには、Crondをはじめとするスケジューリングプログラムや、常駐プログラム(デーモン)として、バックグランドプロセスで稼働させる方法が充実しています。

6-2-1 Crondでスケジューリングを登録する

Crondは、プログラムを指定した時間に実行するためのデーモンです。デーモン(Daemon)とは、バックグランドプロセスとして動くプログラムのことで、ユーザーが直接対話的に制御するのではなく、決められたルールに従って自動的に動きます。UNIXやLinuxは数々のデーモンが稼働可能で、例えば、Webサーバである Apache Httpdであったり、SSHの接続要求を受け付けるsshdなどがあります。Crondもデーモンの一種で、タスクスケジューラとして常駐し、Crontabという設定ファイルに従いプログラムを実行します。

CrontabはCrondが参照する設定ファイルで、crontabコマンドもしくは直接ファイルを編集することにより設定できます。スケジューリングは、月・曜日・日・時間・分単位で指定可能で、複数のタスクを登録できます。書式は次のとおりです。スケジューリングの登録・編集の際には、-eを付けて実行します。

構文 スケジューリングの登録

```
crontab -e
「分」「時」「日」「月」「曜日」[実行コマンド]
```

ワイルドカードで指定することもできます。例えば、毎時20分に実行(Testという文字列を表示)する場合は、次のような形になります。

```
20 * * * * echo test
```

毎日、13:20に一度だけ実行の場合は、時分の指定をします。

```
20 13 * * * echo test
```

曜日の指定も数字で行います。月曜日が「1」で、日曜日は「7」もしくは「0」で指定します。毎週火曜日の13時20分に実行の場合は、次のようになります。

```
20 13 * * 2 echo test
```

crontabコマンドは基本的な設定以外にも、間隔指定・リスト指定・範囲指定などさまざまな指定方法があります。例えば、10分ごとに実行する場合は、「/」の後に間隔を指定します。

```
*/10 * * * * echo test
```

範囲指定であれば、「-」で指定します。9時から17時の間まで、毎時10分に実行する例です。

```
10 9-17 * * * echo test
```

リスト指定は、「,」で区切り複数の値を設定します。毎週水曜日と金曜日の13時20分に実行する例です。

```
20 13 * * 3,5 echo test
```

複数の指定方法を組み合わせて記述することも可能です。6時、9時～17時の毎時、23時で実行する例です。

```
00 6,9-17,23 * * * echo test
```

それでは、実際にスケジューリング登録してみましょう。CentOSにログインしたうえで、crontabコマンドで登録します。crontabに-lの引数を付けると、現在の一覧が表示されます。-eで登録・編集ができます。編集の際は、vimが起動されます。

下記の例は、毎時40分に/tmpディレクトリに、実行日時を名前にファイルを作成する例です。コマンド中に「%」を含む場合は、「¥」（あるいはバックスラッシュ）でエスケープが必要になります。

◆ スケジューリングを確認する

```
$ crontab -l
no crontab for vagrant
```

◆ スケジューリングを登録する

```
$ crontab -e
40 * * * * /bin/touch /tmp/`date +"¥%Y¥%m¥%d¥%H¥%M"`
```

6-2-2 Crondで動かす際の注意点

Crontabを設定する際には、環境変数に注意する必要があります。有効な環境変数は、「HOME」「SHELL」「LOGNAME」のみです。ユーザーアカウントに設定している環境変数は、Crondでの実行時には引き継がれません。したがって、パスが設定されていないためにコマンドが実行されなかったり、環境変数に設定した各種設定情報が無効になるといったことが起こり得ます。Crondで実行時に環境変数を利用するには、Crontab内で宣言する方法や、コマンド実行時にシェルスクリプトを指定して、そのなかで宣言するなどの方法があります。

Crontab内で指定する

Crontab内で指定する場合は、「変数名=設定値」という形で指定します。次の例では、crontabで実行する際に利用するPATHを指定しています。他に必要な環境変数があれば、同じように登録します。

◆ Crontabから環境変数を設定する

```
$ crontab -e
PATH=/home/vagrant/.rvm/gems/ruby-2.0.0-p481/bin:/home/vagrant/.rvm/gems/ruby-2.0.0-p481@
global/bin:/home/vagrant/.rvm/rubies/ruby-2.0.0-p481/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/
usr/sbin:/sbin:/home/vagrant/.rvm/bin:/home/vagrant/bin

40 * * * * /bin/touch /tmp/`date +"¥%Y¥%m¥%d¥%H¥%M"`
```

一方で、crontabで実行するすべてのコマンドに、同一の環境変数を与えられないという場合もあります。そういった場合は、別の方法で指定する必要があります。

シェルスクリプトから指定する

Crondからシェルスクリプトを呼び出し、環境変数を宣言したうえでコマンドを実行する方法もあります。CrondではHOMEの環境変数は有効です。それを利用して、ユーザーの環境変数を読み込んだうえで、コマンドをキックするという方法も考えられます。

シェルスクリプト show_environment.sh

```
source /etc/profile
source $HOME/.bashrc
export > /tmp/environment
```

◆ シェルスクリプトから環境変数を指定する

```
$ crontab -e
10 * * * * bash $HOME/show_environment.sh
```

なお、サーバの時刻設定が日本時間ではなくUTC（協定世界時）の場合、下記の方法で日本時間の設定ができます。必要に応じて、設定してください。

```
$ sudo cp -p /usr/share/zoneinfo/Asia/Tokyo /etc/localtime
```

定期的に実行することで、クローラーの利便性は上がります。次はCrondと組み合わせて、クローラーを運用する方法について紹介していきます。

6-2-3　差分を検知する

Crondを使うことにより、簡単に定期実行が可能になりました。それでは、定点観測を行い結果を格納しましょう。データの格納先はいろいろありますが、後々の扱いやすさという点でリレーショナルデータベース（RDB）を利用します。今回はオープンソースのRDBであるMySQLを利用します。

MySQLのインストール

CentOSであればyumコマンドで、リモートから各種のパッケージを取得してインストールできます。yum listでコマンド取得可能なプログラムの一覧が表示されます。一方でデフォルトで提供されているパッケージが既に古い場合があります。その場合は、取得先を追加することにより、より新しいバージョンが入手可能となることがあります。

次の例は、MySQLのコミュニティが提供しているパッケージを、取得対象リストとして追加している例です。

◆ MySQLのyumリポジトリを追加する

```
$ sudo yum install -y http://repo.mysql.com/mysql-community-release-el6-5.noarch.rpm
```

◆ 追加したyumリポジトリの確認

```
$ ls /etc/yum.repos.d/
CentOS-Base.repo      CentOS-Vault.repo  mysql-community-source.repo
CentOS-Debuginfo.repo epel-testing.repo  mysql-community.repo
CentOS-Media.repo     epel.repo
```

それでは、MySQLを追加しましょう。今回はMySQLのデーモンと開発用のヘッダーをあわせて指定しています。yumでのインストール後に、自動起動の設定とmysqldの起動を行っています。

◆ MySQLのインストール

```
$ sudo yum install mysql-community-server mysql-devel
```

◆ 自動起動の設定

```
$ sudo chkconfig mysqld on
$ chkconfig --list | grep mysql
mysqld      0:off  1:off  2:on   3:on   4:on   5:on   6:off
```

◆ MySQLの起動

```
$ sudo service mysqld start
Starting mysqld:                           [  OK  ]
```

mysqldの起動ができたら、ログインしてみましょう。インストール直後はrootユーザーでパスワードなしでログインできます。実際に運用する際は、個別のユーザーの作成とパスワードの設定を行いましょう。

◆ MySQLへのログイン

```
$ mysql -uroot -p
```

MySQLログイン

```
centos — vagrant@vagrant-centos65:~ — ssh — 80×15
[vagrant@vagrant-centos65 ~]$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.6.19 MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

6-2-4 時系列で表示する

それでは、CrondとMySQLを利用して定期的にデータの収集・保存を行います。今回は、「5-14 企業・株価情報を収集する」(→p.321)で紹介した株価の収集スクリプトと組み合わせます。

ライブラリのインストール

まず、RubyからMySQLを扱うために、ライブラリのインストールをします。MySQLのライブラリはいくつかありますが、「mysql2」を利用します。

■ GitHubのmysql2ページ
URL https://github.com/brianmario/mysql2

◆ mysql2ライブラリのインストール

```
$ gem install mysql2
```

データベースの作成

MySQLにログインし(→p.379)、データベースとユーザー、テーブルの作成を行います。

crawler_dbデータベースを作成し、crawlerユーザーにパスワードを「crawler_pass」で作成しています。データの格納先であるstocksテーブルは、idの他に銘柄コード、日付、高値・安値を項目として持ちます。

◆ MySQLの設定

```
mysql> create database crawler_db default character set=utf8;
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON *.* TO crawler@localhost IDENTIFIED BY 'crawler_pass' WITH GRANT
OPTION;
Query OK, 0 rows affected (0.01 sec)

mysql> use crawler_db
Database changed

mysql> create table stocks (id int(11) not null auto_increment, ticker_code char(4) not null, day_str char(8)
not null, high_price int(4), low_price int(4), primary key (id), key (ticker_code,day_str));
Query OK, 0 rows affected (0.03 sec)
```

MySQLのコマンドの詳細については、公式ドキュメントを参照してください。

■ MySQL公式ドキュメント
URL http://dev.mysql.com/doc/#manual

スクリプトの作成

準備が整ったところで、「5-14 企業・株価情報を収集する」(→p.321)で作成したスクリプト「nokogiri-stock.rb」を修正します。スクレイピング部分に日付の取得と、プログラムの最終部分に、mysqlへのインサート機能を追加します。

追加するコード

```
require 'mysql2'

client = Mysql2::Client.new(:host => "localhost",
  :username => "crawler", :password => "crawler_pass",
  :database => "crawler_db")
count = client.query(
  "SELECT id FROM stocks WHERE day_str = #{company.day}").size
if count == 0 then
  client.query("INSERT INTO stocks (
    ticker_code,day_str,high_price,low_price)
    VALUES(#{company.tickerCode}, #{company.day},
    #{company.highPrice}, #{company.lowPrice})")
end
```

次に示すのが、「nokogiri-stock.rb」にコードを追加した状態です。

定期的に株価情報を取得する

stock.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

class CompanyInfo
    def initialize(ticker_code)
      @baseUrl = "http://stocks.finance.yahoo.co.jp/stocks"
      @tickerCode = ticker_code
      scrape
    end
    attr_reader :name, :tickerCode, :category,
      :unit, :recentHighPrice, :recentLowPrice,
      :highPrice, :lowPrice, :price, :day
```

```
  private
  def scrape_stock_info(html, index)
    get_content(html, "dd", "ymuiEditLink marO",
      index, "/strong").delete(",")
  end

  def get_company_info()
    url = "#{@baseUrl}/profile/?code=#{@tickerCode}"
    doc = get_nokogiri_doc(url)
    @name = doc.xpath("//th[@class='symbol']/h1").text
    @category = doc.xpath(
      "//table[@class='boardFinCom marB6']/tr[6]/td").text
    @unit = doc.xpath(
      "//table[@class='boardFinCom marB6']/tr[13]/td").text
  end

  def get_stock_info()
    url = "#{@baseUrl}/detail/?code=#{@tickerCode}"
    doc = get_nokogiri_doc(url)
    @recentHighPrice = doc.xpath(
      "//div[11]/dl/dd[@class='ymuiEditLink marO']/strong").text
    @recentLowPrice = doc.xpath(
      "//div[12]/dl/dd[@class='ymuiEditLink marO']/strong").text
    @highPrice = doc.xpath(
      "//div[@class='innerDate']/div[3]/dl/
       dd[@class='ymuiEditLink marO']/strong").text
    @lowPrice = doc.xpath(
      "//div[@class='innerDate']/div[4]/dl/
       dd[@class='ymuiEditLink marO']/strong").text
    @price = doc.xpath("//td[@class='stoksPrice']").text
    date=Time.now
    day_str=doc.xpath("//dd[@class='yjSb real']/span").text
    @day = "#{date.year}#{day_str.gsub(/¥//,'')}"
  end

  def get_nokogiri_doc(url)
    begin
      html = open(url)
    rescue OpenURI::HTTPError
      return
    end
    Nokogiri::HTML(html.read, nil, 'utf-8')
  end

  def scrape
    get_company_info
    get_stock_info
  end
```

```
end

company = CompanyInfo.new("4689")
require 'mysql2'
client = Mysql2::Client.new(:host => "localhost",
  :username => "crawler", :password => "crawler_pass",
  :database => "crawler_db")
count = client.query(
  "SELECT id FROM stocks WHERE day_str = #{company.day}").size
if count == 0 then
  client.query(
    "INSERT INTO stocks (ticker_code,day_str,
     high_price,low_price) VALUES(#{company.tickerCode},
     #{company.day}, #{company.highPrice},
     #{company.lowPrice})")
end
```

「/home/vagrant」の下に、「crawler/stock」というディレクトリを作成します。そして、先ほどの「stock.rb」と、次の「exec_stock.sh」を配置します。

◆ ディレクトリの作成

```
$ cd /home/vagrant
$ mkdir crawler
$ mkdir stock
```

実行のためのシェルスクリプト　exec_stock.sh

```
#!/bin/sh

ruby $HOME/crawler/stock/stock.rb
```

スクリプトの実行

月曜日から金曜日まで、毎日16時05分に実行します。環境変数を読み込むために、シェルスクリプト経由で起動しています。スクリプトのパスは、環境に合わせて変更してください。

取得した結果は、先に作成しておいた「crawler_db」データベースに蓄積されていきます。

◆ スケジューリングの登録

```
$ crontab -e
05 16 * * 1-5 /bin/bash -l /home/vagrant/crawler/stock/exec_stock.sh
```

これで、クローラーが定期的にデータを収集してくれます。収集したデータは、以下のように確認することができます。

◆ 取得したデータの確認

```
$ mysql -uroot -p
mysql> use crawler_db;
mysql> select * from stocks;
+----+-------------+----------+------------+-----------+
| id | ticker_code | day_str  | high_price | low_price |
+----+-------------+----------+------------+-----------+
|  1 | 4689        | 20140606 |        503 |       493 |
|  2 | 4689        | 20140609 |        512 |       503 |
|  3 | 4689        | 20140611 |        496 |       482 |
|  4 | 4689        | 20140612 |        486 |       478 |
```

6-3 収集結果をメールで自動送信する

Crondとデータベースを組み合わせれば、クローラーを自由自在に動かすことができます。次は、クローラーから結果をメールで受け取る方法について紹介します。

クローラーの起動を定期的に自動で実行するようになると、次はその実行結果を取得したくなります。結果通知の方法として、古典的ながら有効な手法の1つとして、メールで通知する方法があります。

6-3-1　どういった内容を送るのか

結果通知をメールで送る場合、どのようなケースで送るかのパターン分けがあります。想定どおりのクローリングができた時を成功、できなかった場合を失敗と定義すると、次のパターンが考えられます。

- 成功/失敗にかかわらず、すべて送る
- 成功の時のみ送る
- 失敗の時のみ送る
- 成功/失敗にかかわらず、すべて送らない

今回は、メール送信を行うことが前提ですから、4番目の送らないというパターンが外れます。また2番目の成功の時のみ送るでは、失敗した場合の対処が取りづ

らいです。現実的には、1番目もしくは3番目のパターンを取ることが多いです。今回は、1番目の「成功/失敗にかかわらずすべて送る」を前提とします。

メールを送るには、SMTPサーバが必要になります。しかし、昨今のスパムメールの増加の影響で、キャリアやプロバイダなど通信業者はメール送信についての制約を強めています。そのため、個人でSMTPサーバを立てて運用することが難しくなりつつあります。そこで、SMTPサーバを構築せずにメールを送信する方法を紹介します。Gmailを利用する方法と、AmazonのクラウドサービスのAWSのなかのメール送信サービスであるAmazon Simple Email Service (SES) を利用する方法です。サービスごとに、それぞれ解説します。

6-3-2 Gmailを使って結果通知する

Gmailを利用してメールを送信するには、認証方式や通信プロトコルでいくつかの選択肢があります。基本的な形としては、認証したうえでGmailのSMTPサーバを利用して送信するという形になります。

RubyからGmailを利用する

RubyからGmailを利用するには、さまざまな方法があります。Gmailも一般的なSMTPサーバとしての機能を提供しているので、それを利用してメール送信モジュールを利用する方法があります。それ以外にも、Gmailに特化したモジュールも多数存在します。

Gmail専用ライブラリを利用する方がコードの記述量も少ないので、今回はGmail専用のライブラリを利用します。Ruby向けのGmail専用ライブラリはいくつかありますが、一番実績があり開発も活発な「GMail for Ruby」を利用します。

■ GitHubのGMail for Rubyページ
URL https://github.com/nu7hatch/gmail

インストールは、gemから行えます。インストール後に、gem list gmailでバージョンが表示されれば成功です。

◆ GMail for Rubyのインストール

```
$ gem install gmail
```

◆ インストールの確認

```
$ gem list gmail
*** LOCAL GEMS ***
```

```
gmail (0.4.0)
gmail_xoauth (0.4.1)
```

認証方法

GmailのGemライブラリの認証方法は、IDとパスワードによる認証と、OAuth 1.0による認証の2種類があります。GoogleのOAuth 1.0は、2012年4月20日をもってサポート終了しています。かわりにOAuth 2.0が提供されていますが、RubyのGemのなかでOAuth 2.0に対応している有力なライブラリがないのが現状です。本来であればスクリプトから利用する場合は、ID・パスワードでの認証より、OAuthでの認証をすべきです。今回はID・パスワード認証のサンプルを提示します。本格的に使うのであれば、OAuth 2.0対応を取り込みましょう。

以下が、Gmailの認証を行うスクリプトです。実行の際には、「Gmail.connect」にご自分のアカウントのIDとパスワードを設定してください(❶)。

Gmailの認証 gmail-id.rb

```
# -*- coding: utf-8 -*-
require 'gmail'

gmail = Gmail.connect('GOOGLE_ID', 'GOOGLE_PASS') ●——❶
gmail.deliver do
  to "example@example.com"
  subject "日本語"
  text_part do
    body "日本語"
  end
end

gmail.logout
```

なおGoogleの2要素認証の設定をして、IDとパスワード以外にスマホなどでのパスコードが必要な場合は、別途対策が必要になります。

クローラーの結果を通知

メールが送信できることを確認できたら、次はクローラーの結果を通知しましょう。「5-14 企業・株価情報を収集する」(→p.321)で作成したスクリプト「nokogiri-stock.rb」に、結果をメールで送付する処理を追加します。

追加するコード

```
require 'gmail'

day=Time.now
day_str=
  "#{day.year}/#{sprintf("%02d",day.month)}/#{sprintf("%02d",day.day)}"
mail_body = <<"EOS"
  #{company.name}
  #{company.category}
  #{company.unit}
  年初来高値:#{company.recentHighPrice}
  年初来安値:#{company.recentLowPrice}
  高値:#{company.highPrice}
  安値:#{company.lowPrice}
  株価:#{company.price}
EOS

gmail = Gmail.connect('GOOGLE_ID', 'GOOGLE_PASS')
gmail.deliver do
  to "mail@example.com"
  subject "株価情報#{day_str}"
  text_part do
    body mail_body
  end
end

gmail.logout
```

取得した株価情報をメールで送付する

nokogiri-stock-mail.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

class CompanyInfo
    def initialize(ticker_code)
      @baseUrl = "http://stocks.finance.yahoo.co.jp/stocks"
      @tickerCode = ticker_code
      scrape
    end
    attr_reader :name, :tickerCode, :category,
      :unit, :recentHighPrice, :recentLowPrice,
      :highPrice, :lowPrice, :price

    private
    def scrape_stock_info(html, index)
```

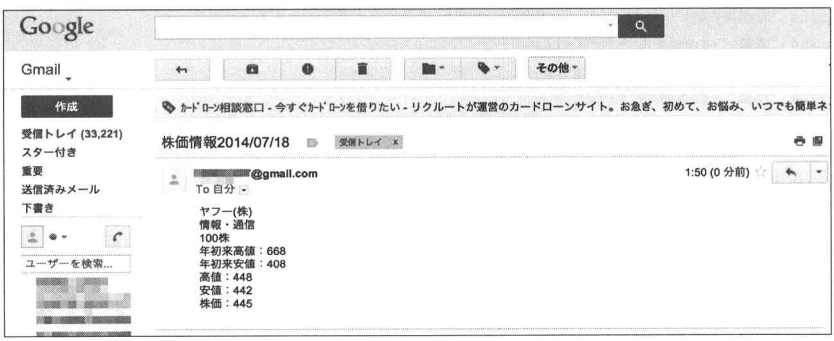
```
      get_content(html, "dd", "ymuiEditLink marO",
        index, "/strong").delete(",")
    end

    def get_company_info()
      url = "#{@baseUrl}/profile/?code=#{@tickerCode}"
      doc = get_nokogiri_doc(url)
      @name = doc.xpath("//th[@class='symbol']/h1").text
      @category = doc.xpath(
        "//table[@class='boardFinCom marB6']/tr[6]/td").text
      @unit = doc.xpath(
        "//table[@class='boardFinCom marB6']/tr[13]/td").text
    end

    def get_stock_info()
      url = "#{@baseUrl}/detail/?code=#{@tickerCode}"
      doc = get_nokogiri_doc(url)
      @recentHighPrice = doc.xpath(
        "//div[11]/dl/dd[@class='ymuiEditLink marO']/strong").text
      @recentLowPrice = doc.xpath(
        "//div[12]/dl/dd[@class='ymuiEditLink marO']/strong").text
      @highPrice = doc.xpath(
        "//div[@class='innerDate']/div[3]/dl/
         dd[@class='ymuiEditLink marO']/strong").text
      @lowPrice = doc.xpath(
        "//div[@class='innerDate']/div[4]/dl/
         dd[@class='ymuiEditLink marO']/strong").text
      @price = doc.xpath("//td[@class='stoksPrice']").text
    end

    def get_nokogiri_doc(url)
      begin
        html = open(url)
      rescue OpenURI::HTTPError
        return
      end
      Nokogiri::HTML(html.read, nil, 'utf-8')
    end

    def scrape
      get_company_info
      get_stock_info
    end
end

company = CompanyInfo.new("4689")
puts company.name
```

```
puts company.category
puts company.unit
puts "年初来高値:"+company.recentHighPrice
puts "年初来安値:"+company.recentLowPrice
puts "高値:"+company.highPrice
puts "安値:"+company.lowPrice
puts "株価:"+company.price

require 'gmail'

day=Time.now
day_str=
  "#{day.year}/#{sprintf("%02d",day.month)}/#{sprintf("%02d",day.day)}"
mail_body = <<"EOS"
 #{company.name}
 #{company.category}
 #{company.unit}
 年初来高値:#{company.recentHighPrice}
 年初来安値:#{company.recentLowPrice}
 高値:#{company.highPrice}
 安値:#{company.lowPrice}
 株価:#{company.price}
EOS

gmail = Gmail.connect('GOOGLE_ID', 'GOOGLE_PASS')
gmail.deliver do
  to "mail@example.com"
  subject "株価情報#{day_str}"
  text_part do
    body mail_body
  end
end

gmail.logout
```

スクリプトを定期的に実行する方法は、p.375をご参照ください。

無事、メールは送られてきたでしょうか。今回の株価情報のようにスマートフォンのアプリがあるようなものであれば、わざわざメールで送る必要はないですが、クローラーでしか取れない情報であればメールでスマートフォンに通知することで便利になることが多いです。

メール通知



メールの文字コード

メールで送信する場合は、文字コードの考慮がないと文字化けが発生するケースがあります。

今回は、メールの文面をUTF-8で作成し、そのまま送信しています。昨今のメールクライアントであれば、UTF-8で送られても文字化けせず表示できるケースが多いです。自分が使っているメールクライアントで文字化けする場合は、ISO-2022-JPでエンコードするなど対策をしてください。

6-3-3　Amazon Simple Email Service (SES) を使って結果通知する

Gmailを利用する他に、Amazonのクラウドサービスである AWS (Amazon Web Services) を利用する方法があります。AWSは、仮想サーバ以外にも多数のサービスが存在します。そのなかの1つに、「Amazon Simple Email Service (SES)」というメール送信のサービスがあります。SESは有料のサービスですが、一般的な利用の範囲であれば無料で利用できます。2014年7月現在では、1日2,000通までの送信は無料で利用できます。料金の詳細については、Amazon SESの料金ページで確認してください。

■ Amazon SES料金表
URL http://aws.amazon.com/jp/ses/pricing/

AWSのアカウントを作成する

AWSを利用するには、Amazon.comのアカウントが必要となります。「4-6-2 Amazon Product Advertising API」(→p.243)でアカウントを作成しているのであれば、そのアカウントを活用できます。アカウント作成の流れは、以下のとおりです。

- ①AWSアカウントでサインインもしくは新規アカウントの作成
- ②新規アカウント作成の場合は、名前やパスワードなどのAWSログイン情報の設定
- ③氏名・住所などの問い合わせ情報の入力
- ④クレジットカードの登録
- ⑤電話(自動音声)による身元確認
- ⑥AWSサポートプランの選択

登録方法については、Amazon公式ページに丁寧に説明されています。それを参照のうえで、登録してください。AWSサポートプランについては、ベーシック(無料)で問題ありません。

■ AWSアカウントの作成ページ
URL http://aws.amazon.com/jp/register-flow/

SESを利用する

AWSアカウントを用意し、プログラムから利用するためのAPIのキーを取得すればSESの利用はできます。APIキーは、「4-6-2 Amazon Produce Advertising API」(→p.243)と同じ方法で取得できますが、権限の範囲が広すぎるので限定したアカウントを発行します。

AWSには、「Identity and Access Management (IAM)」という認証と権限管理の機能があります。IAMを使うことにより、機能単位でのアクセス許可ができます。セキュリティを考慮すると、最小限の権限を持ったIAMアカウントで運用するのがよいでしょう。IAMアカウントの発行手順と権限付与は、次のとおりです。

- ①AWS管理コンソールにログインする
- ②サービスの一覧から「IAM」を選択する
- ③[Users]を選択し、[Create New Users]ボタンを押下する
- ④任意のユーザー名を入力し、IAMアカウントを作成する。その際に、「AccessKey」を保存しておく
- ⑤作成したユーザーを選択し、「Permissions」を選び、[Attach User Policy]を押下

する

- ⑥Select Policy Templateの「Amazon SNS Full Access」を選択し、［Apply Policy］を押下する

IAMアカウントの作成が完了したら、次はSESでのメールアドレスの登録です。Amazon SESでは、不正利用を防ぎ信頼性を高めるために、認証された送信元からのみメールを送信できます。認証の単位は、メールアドレスもしくはドメインです。今回は、メールアドレスの認証を行います。

- ①サービスの一覧から「SES」を選択する
- ②SESのダッシュボードから、「Email Addresses」を選択する
- ③[Verify a New Email Address］ボタンを押下して、利用したいメールアドレスを入力する
- ④入力したメール宛に確認メールが届くので、指示に従ってURLを開く
- ⑤認証完了

RubyからAWSの各サービスを利用する場合は、Amazon自身が出している公式のGemライブラリを利用するのが一番よいです。インストールはgemから行えます。インストール完了後にgem list aws-sdkでバージョンが表示されれば成功です。

◆ aws-sdkのインストール

```
$ gem install aws-sdk
```

◆ インストールの確認

```
$ gem list aws-sdk

*** LOCAL GEMS ***
aws-sdk (1.42.0)
```

ここまで用意ができれば、SESの利用は簡単です。アクセスキーとシークレットアクセスキーを使って認証をした後に、メールを送信します。なお送信元として設定できるメールアドレスは、認証したメールアドレスのみとなっています。

「5-14 企業・株価情報を収集する」(→p.321)で作成したスクリプト「nokogiri-stock.rb」に、以下のコードを追加します。実行の際には、「:access_key_id」と

「:secret_access_key」にIAMアカウントのAccess KeyとSecret Access Keyを指定してください。「:from」のアドレスには認証したメールアドレスを、「:to」のアドレスには送信先のメールアドレスを設定してください。

追加するコード

```
require 'aws-sdk'

ses = AWS::SimpleEmailService.new(
  :access_key_id => 'AWS_SES_KEY',
  :secret_access_key => 'AWS_SES_SECRET'
)

mail_body = <<"EOS"
 #{company.name}
 #{company.category}
 #{company.unit}
 年初来高値:#{company.recentHighPrice}
 年初来安値:#{company.recentLowPrice}
 高値:#{company.highPrice}
 安値:#{company.lowPrice}
 株価:#{company.price}
EOS

ses.send_email(
  :subject => "株価情報#{company.day}",
  :from => 'mail@example.com',
  :to => 'mail@example.com',
  :body_text => mail_body
)
```

AWSで株価情報をメール送信する nokogiri-stock-ses.rb

```
# -*- coding: utf-8 -*-
require 'nokogiri'
require 'open-uri'

class CompanyInfo
    def initialize(ticker_code)
      @baseUrl = "http://stocks.finance.yahoo.co.jp/stocks"
      @tickerCode = ticker_code
      scrape
    end
    attr_reader :name, :tickerCode, :category,
      :unit, :recentHighPrice, :recentLowPrice,
      :highPrice, :lowPrice, :price, :day
```

```
  private
  def scrape_stock_info(html, index)
    get_content(html, "dd", "ymuiEditLink marO",
      index, "/strong").delete(",")
  end

  def get_company_info()
    url = "#{@baseUrl}/profile/?code=#{@tickerCode}"
    doc = get_nokogiri_doc(url)
    @name = doc.xpath("//th[@class='symbol']/h1").text
    @category = doc.xpath(
      "//table[@class='boardFinCom marB6']/tr[6]/td").text
    @unit = doc.xpath(
      "//table[@class='boardFinCom marB6']/tr[13]/td").text
  end

  def get_stock_info()
    url = "#{@baseUrl}/detail/?code=#{@tickerCode}"
    doc = get_nokogiri_doc(url)
    @recentHighPrice = doc.xpath(
      "//div[11]/dl/dd[@class='ymuiEditLink marO']/strong").text
    @recentLowPrice = doc.xpath(
      "//div[12]/dl/dd[@class='ymuiEditLink marO']/strong").text
    @highPrice = doc.xpath(
      "//div[@class='innerDate']/div[3]/dl/
       dd[@class='ymuiEditLink marO']/strong").text
    @lowPrice = doc.xpath(
      "//div[@class='innerDate']/div[4]/dl/
       dd[@class='ymuiEditLink marO']/strong").text
    @price = doc.xpath("//td[@class='stoksPrice']").text
    date=Time.now
    day_str=doc.xpath("//dd[@class='yjSb real']/span").text
    @day = "#{date.year}/#{day_str}"
  end

  def get_nokogiri_doc(url)
    begin
      html = open(url)
    rescue OpenURI::HTTPError
      return
    end
    Nokogiri::HTML(html.read, nil, 'utf-8')
  end

  def scrape
    get_company_info
```

```
      get_stock_info
    end
end

company = CompanyInfo.new("4689")
puts company.name
puts company.category
puts company.unit
puts "日付:"+company.day
puts "年初来高値:"+company.recentHighPrice
puts "年初来安値:"+company.recentLowPrice
puts "高値:"+company.highPrice
puts "安値:"+company.lowPrice
puts "株価:"+company.price

require 'aws-sdk'

ses = AWS::SimpleEmailService.new(
  :access_key_id => 'AWS_SES_KEY',
  :secret_access_key => 'AWS_SES_SECRET'
)

mail_body = <<"EOS"
 #{company.name}
 #{company.category}
 #{company.unit}
 年初来高値:#{company.recentHighPrice}
 年初来安値:#{company.recentLowPrice}
 高値:#{company.highPrice}
 安値:#{company.lowPrice}
 株価:#{company.price}
EOS

ses.send_email(
  :subject => "株価情報#{company.day}",
  :from => 'mail@example.com',
  :to => 'mail@example.com',
  :body_text => mail_body
)
```

メールの送信までできれば、クローラーとの連携は「6-3-2 Gmailを使って結果通知する」(→p.385) と同じです。

6-4 クラウドを活用する

前節では、メール送信のためにAmazonのクラウドサービス（AWS）を利用しました。メール送信以外にも、クローラーを構築運用するうえで、クラウドの活用シーンはたくさんあります。このセクションでは、クラウドの活用方法を紹介していきます。

6-4-1　AWSのサービス

AWSには、仮想サーバであるAmazon EC2からストレージサービスのS3、データベースサービスのRDSやDynamoDB、通知サービスのSNSやキューサービスのSQSなどさまざまなレイヤーのサービスがあります。それぞれの役割がわかると、クローラーの構築と運用が非常に楽になります。

仮想サーバ「Amazon EC2」

筆者は簡単なクローラーであれば自宅のPCで動かしていますが、継続的に動かすものについてはAWS上のEC2を利用しています。EC2は「Elastic Compute Cloud」の略で、Amazonのクラウド内のコンピュータリソースを利用できるサービスです。

利用は1時間単位で、CPUやメモリのサイズによりますが、1時間あたり数円から利用できます。また、スポットインスタンスと呼ばれる入札制の仕組みを活用すると、さらに数分の1という値段で利用できる場合があります。それぞれのサーバにグローバルIPを割り振ることが可能なので、複数のIPアドレスが必要な場合などに重宝します。また複数台を同時に起動して、一気に処理するといった用途でも使えます。

ストレージサービス「Amazon S3」

EC2と並びAWSを代表するサービスの1つに、Amazon S3（Simple Storage Service）があります。ストレージのサービスですが、容量無制限でWebのどこからでも利用できるオンラインストレージです。データは3重以上に冗長化されて保存され、ユーザーはデータのバックアップの心配もほぼ無用です。

1GBあたり1ヶ月3円程度のコストで運用できます。純粋にハードディスクのGBあたりの単価と比べると割高に見えるかもしれませんが、オンラインで利用できて

かつ3重以上にバックアップされているということを考えると、驚異的な値段です。

S3は、データの保存先として非常に優れています。筆者は、クローリングで収集したHTMLやTwitter Streaming API (→p.277) で収集した過去数年分のTweetの保存などに利用しています。

データベースサービス「Amazon RDS」

PaaSはクラウドの提供形態の1つで、実行環境などを直接提供します。Amazon RDSはPaaSの形態で提供されているサービスで、ユーザーにOSなどを意識させることなくリレーショナルデータベース (RDB) を提供します。2014年現在で提供されているデータベースは、MySQL、Oracle、SQLServer、PostgreSQLと主要なものは一通り揃っています。

データベースを利用する時に、わざわざインストールから始める必要もなく、まさにサービスという形で利用できます。筆者も最近では、データベースを利用する場合は、ほぼRDSを利用しています。インストール不要という以外にも、利用シーンに応じて柔軟にリソースの増減が可能です。

クローラーの利用では、メタデータの管理はデータベースを利用すると検索可能になりデータとしての価値が上がります。クローラーとデータベースは、切っても切れない関係です。

通知サービス「Amazon SNS」

AWSでは、EC2のようなIaaSやRDSなどのPaaSのみならず、アプリケーションサービスも多数出しています。そのなかの1つが、Amazon SNS (Simple Notification Service) です。名前のとおり通知サービスで、メールやHTTP通知、iPhoneやAndroidなどのスマートフォンなど、さまざまな方法で通知できます。スマートフォンへのプッシュ通知を一から実装するのはなかなか手間ですが、SNSを使うことで比較的簡単に実装できます。

キューサービス「Amazon SQS」

Amazon SQS (Simple Queue Service) は、AWSのなかでもあまり目立たないサービスです。いわゆるキューのサービスを提供するのですが、従来のキューのように大がかりなシステムの用意も不要で、手軽に扱えます。

キューのシステムは、メッセージを登録して、順番に取得するだけのサービスです。機能は非常に単純ですが、上手く活用すると、クローラーとこれほど相性がよ

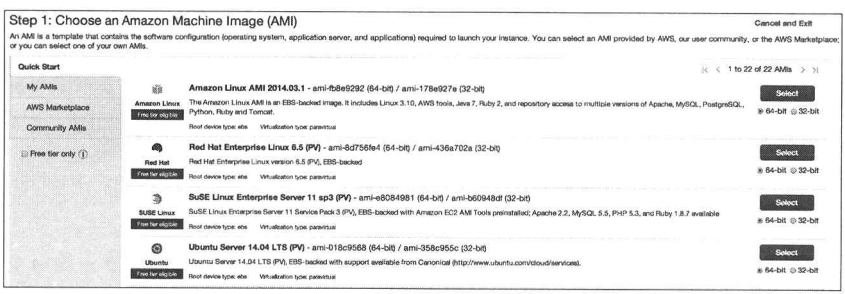
いサービスもないくらいのものです。例えば、複数のサーバを同時に立ち上げてクローラーを並列で動かす時に、処理する対象のURLの管理をどうするかという問題があります。この処理対象の管理をキューを利用することにより、100台のクローラーを同時に動かすということを簡単に実現できます。SQSを使った具体的な例は、「6-5-2 分散処理」(→p.415)で紹介します。

6-4-2　クラウド上のサーバを利用する

Amazon EC2は仮想サーバです。EC2の上で、Windows ServerやRed Hat Enterprise LinuxやUbuntuなどのLinux系OSが利用できます。

AWSではOSイメージをAMI(Amazon Machine Image)と呼びます。AMIはAmazon社自身が出している公式のものや、AWS利用者が作成したCommunity AMI、AWS上に出店するベンダーが作成したMarketplaceのAMIなどがあります。

Amazon AMI

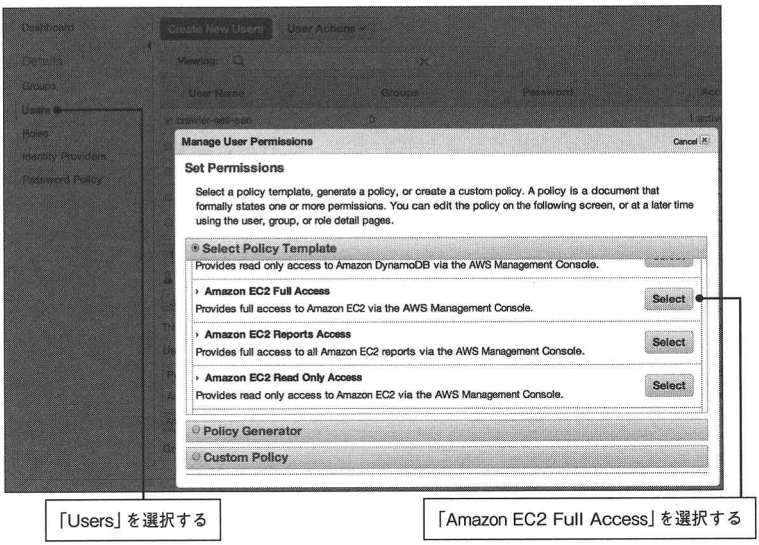


EC2を利用する準備

EC2を利用するには、いくつかの準備が必要です。まずは、「6-3-3 Amazon Simple Email Service (SES)を使って結果通知する」(→p.390)で作成したIAMアカウントに対して、EC2の利用権限を付与します。

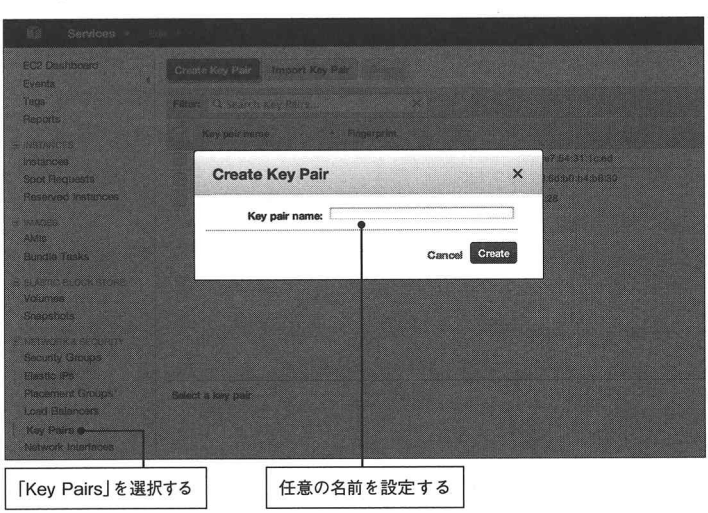
権限の付与は、AWSアカウントでログインしたうえで、AWSコンソールのIAMダッシュボードから行います。「Users」で既存のユーザーを選択し、Manager User Permissionsの「Set Permissions」で権限を追加します。Select Policy Templateの「Amazon EC2 Full Access」を選択し、[Select]を押下します。今後、EC2を操作する場合は、今回作成(設定)したIAMアカウントを利用することを推奨します。

EC2利用権限の付与



次にKey Pairを作成します。Key PairはEC2上に起動したサーバにログインするのに利用します。Key Pairの作成は、AWSコンソールのEC2ダッシュボードの「Key Pairs」から行います。[Create Key Pair]ボタンを押下して、任意の名前を入力すると作成できます。同時に秘密鍵がダウンロードされます。今後のサーバへのアクセスに必要となりますので、大切に保管しましょう。

Key Pairを作成する



サーバを起動する

ここまで準備ができるとAWSの利用は可能となります。EC2ダッシュボードから「Instances」を選択し、[Launch Instance] ボタンを押下します。AMIの選択画面が出てくるので任意のOSを選択しましょう。OSのなかに、「Amazon Linux AMI」という見慣れないものが出てくると思います。これは、Red Had Enterpriseの互換ディストリビューションです。よくメンテナンスされて使いやすいので、特別こだわりがないのであれば使ってみることをお勧めします。

OSを選択すると、次はインスタンスサイズの選択画面になります。インスタンスサイズごとに、CPUやメモリのサイズが違います。当然ながら高性能のインスタンスは時間あたりの料金が高いのでご注意ください。まず使ってみるだけであれば、「Micro Instance」をお勧めします。新規作成したアカウントであれば、無料枠もあります。インスタンスを選択して、[Preview and Launch] を選択すると起動の一歩手前になります。その後に、先ほど作成したKey Pairを選択すると起動できます。

他にも設定できるところはたくさんあるので、興味があればAmazonの公式のトレーニング資料を参考してください。

■ AWS クラウドサービス活用資料集
URL http://aws.amazon.com/jp/aws-jp-introduction/

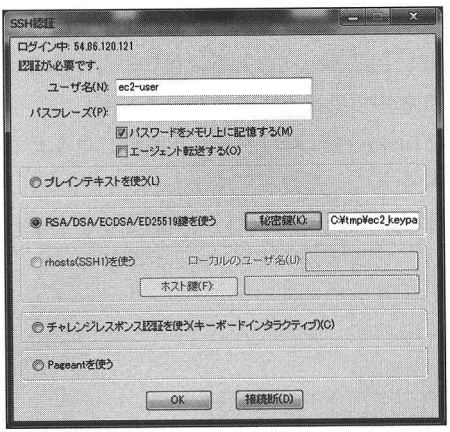
サーバにログインする

サーバを起動すると、Instancesの一覧にサーバが表示されます。選択すると、サーバの詳細情報が表示されます。詳細中にPublic IPがあるので、このIPを利用するとサーバにログインできます。デフォルトのOSユーザーは「ec2-user」という名前で作成されています。

WindowsからのログインはTera TermやPuTTYなどのSSHクライアントツールからログインします。IPアドレスを入力のうえで、ユーザーに「ec2-user」を入力し、パスワードではなく秘密鍵で認証します。

■ Tera Termホームページ
URL http://ttssh2.sourceforge.jp/

TeraTermからログインする



Macの場合は、ターミナルからsshコマンドでログインできます。注意点としては、KeyPair.pemファイルのパーミッションを本人のみにしておく必要があります。sshコマンドで、認証にKeyPair.pemファイルを指定するには、-iオプションを利用します。IPアドレスは、AWS管理コンソールに表示されたIPアドレスを入力してください。

◆ **サーバログイン(Mac)**

```
$ chmod 600 your_ec2_keypair.pem
$ ssh -i ssh -i your_ec2_keypair.pem ec2-user@99.99.99.99
```

Macのターミナルからログインする

ログインした後は、「6-1 サーバサイドで動かす」(→p.362)で紹介したLinuxの使い方と同じです。

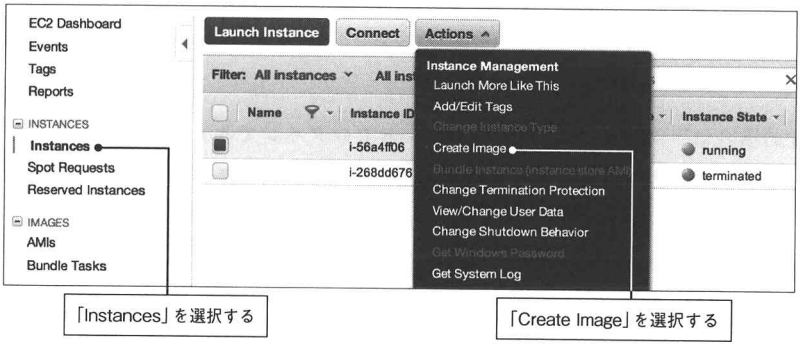
AMIとして保存する

AWSでは、設定したサーバをAMIという形で保存できます。一度保存しておくと、その情報を元に何台でも起動できます。同じサーバを簡単に複数作成できる利点を活かして、何十台も同時に起動させるといったようなことができます。

AMIの作成は、EC2ダッシュボードの「Instances」からサーバを選択したうえで、Actionsの「Create Image」から任意の名前を入力して作成できます。

作成したAMIから起動する場合は、起動画面で「My AMIs」を選択することで利用できます。

Create Image



EC2はかなり使い勝手のよい仮想サーバなので、積極的に使ってみましょう。ただし、従量課金制なので起動しっぱなしであれば、その分課金されます。不要な場合は、終了させることを忘れないようにしましょう。インスタンスを選択したうえで、「Terminate」することで終了にできます。

6-4-3　クラウド上のストレージを利用する

長くクローラーを運用していると、データの保存をどうするかという問題に直面します。クローラーは運用を続けると基本的にデータがどんどんと溜まっていくものです。そんな際に、必要な時に必要なだけ利用できるクラウドストレージは便利です。

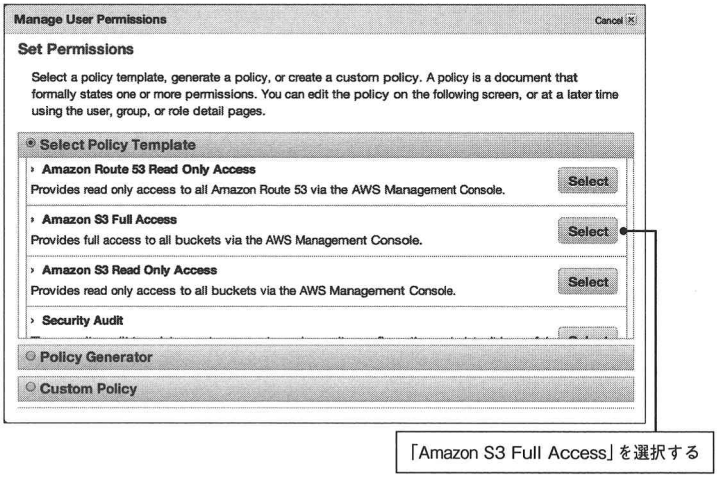
Amazon S3を利用する準備

S3を利用するにも、いくつかの準備が必要です。まずは、「6-3-3 Amazon Simple Email Service (SES)を使って結果通知する」(→p.390)で作成したIAMアカウント

に対して、S3の利用権限を付与します。

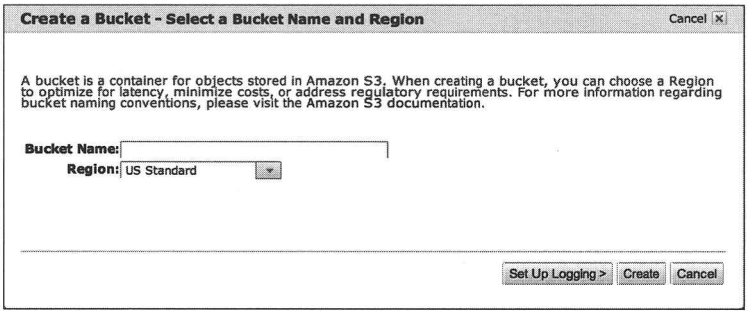
権限の付与は、AWSアカウントでログインしたうえで、AWSコンソールのIAMダッシュボードから行います。「Users」で既存のユーザーを選択し、Manage User Permissionsの「Set Parmissions」で権限を追加します。Select Policy Templateの「Amazon S3 Full Access」を選択し、[Select]を押下します。EC2やSESと同様に、S3を利用する場合はIAMアカウントを利用することを推奨します。

S3利用権限の付与



S3を利用する際は、まずはBucketを作成する必要があります。BucketはS3内で一意の名前である必要があります。つまりドメインと同じように、他のユーザーが既に使っていた場合は利用できません。

S3 Bucketの作成



Bucket作成時にバケット名とRegionの選択があります。Regionはデータを置く地域の選択です。Amazonのデータセンターは、北米・アジア・ヨーロッパと世界中に分散しています。ユーザーはそのなかから、任意のRegionを選択します。基本的には、自分が住んでいる地域から地理的に近いところを選ぶと、通信遅延が少なく便利でしょう。一方で、北米に比べて日本はS3の価格設定が高いといったデメリットがあります。

PCから利用する

S3は、PCからブラウザ経由で利用できます。ファイルのアップロードや一覧表示、ダウンロードなどができます。しかし、日常的に使うには、少し不便です。そこでサードパーティ製のツールを使うことにより、FTPライクに使えます。

サードパーティ製のツールとして、WindowsとMacどちらでも使えるFirefoxを拡張する「S3Fox」や、Windows専用の「CloudBerry」、WindowsやMac用で利用できる「CyberDuck」などがあります。

■ S3Fox

URL https://addons.mozilla.org/ja/firefox/addon/amazon-s3-organizers3fox/

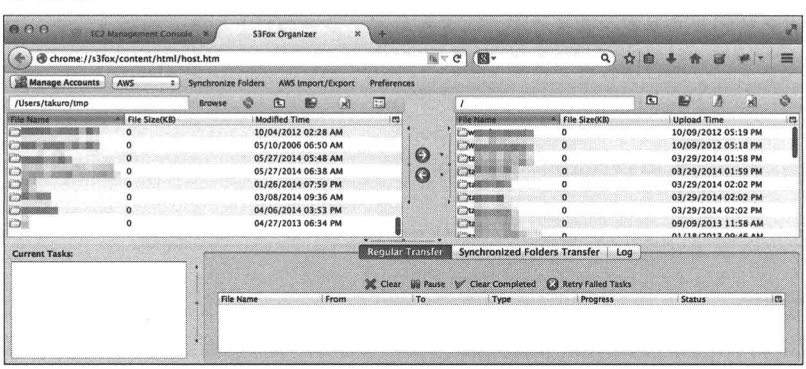
■ CloudBerry

URL http://www.cloudberrylab.com/

■ CyberDuck

URL http://cyberduck.softonic.jp/mac

S3Fox



スクリプトから利用する

スクリプトからS3を使う場合は、SESと同様に公式の「aws-sdk」を利用するとよいです(→p.392)。

◆ aws-sdkのインストール

```
$ gem install aws-sdk
```

RubyでS3を操作するには、以下のような形で一通り扱えます。

実行の際には、「:access_key_id」と「:secret_access_key」に取得したIAMアカウントのアクセスキーや秘密キーを設定してください。また、「your-bucket-name」部分にはバケット名(→p.403)を設定してください。その他、ディレクトリ名やファイル名などを各自の環境に合わせて設定してください。

RubyスクリプトからS3を利用する use-s3.rb

```
# -*- coding: utf-8 -*-
require 'aws-sdk'

AWS.config({
  :access_key_id => 'AWS_ACCESS_KEY',
  :secret_access_key => 'AWS_SECRET_ACCESS_KEY'
})
s3 = AWS::S3.new

bucket = s3.buckets['your-bucket-name']
tree = bucket.as_tree

# ディレクトリ一覧
directories =
  tree.children.select(&:branch?).collect(&:prefix)
directories.each{|directory|
  puts directory
}

# ファイル一覧
files = tree.children.select(&:leaf?).collect(&:key)
files.each{|file|
  puts file
}

# キー一覧(ファイル&ディレクトリ一覧)
puts "keys"
keys = bucket.objects.collect(&:key)
```

```
keys.each{|key|
  puts key
}

# サブディレクトリ下のファイルの一覧表示
puts "sub folder"
tree = bucket.as_tree({:prefix => 'subfoldername/'})
files = tree.children.select(&:leaf?).collect(&:key)
files.each{|file|
  puts file
}

# ファイル操作
obj =
  s3.buckets['your-bucket-name'].objects['filename']

# ファイルの書き込み
obj.write('Hello World!')

# ファイルの読み込み
obj.read
```

S3の速度

スクリプトからS3を扱う際の注意点としては、速度の遅さです。ローカルのストレージにあるファイルに比べ、S3はネットワーク上にあります。ファイル操作は都度HTTPプロトコルで行われるため、小さなファイルでも取得に数十〜数百ミリ秒ほどかかります。このため、頻繁に取得や更新するような処理をS3に対して行うのは向いていません。

クローラーとの連携

クローラーと連携するのであれば、「5-6 画像を収集する」(→p.285) で紹介したような画像収集系のクローラーと連携させるのがよいでしょう。

実行の際には、「:access_key_id」と「:secret_access_key」に取得したIAMアカウントのアクセスキーや秘密キーを設定してください。また、「your-bucket-name」部分にはバケット名(→p.403)を設定してください。

収集した画像をクラウド上に保存する s3-images.rb

```
# -*- coding: utf-8 -*-
require 'aws-sdk'
```

```
require 'open-uri'
require 'cgi'

AWS.config({
  :access_key_id => 'AWS_ACCESS_KEY',
  :secret_access_key => 'AWS_SECRET_ACCESS_KEY'
})
s3 = AWS::S3.new
@bucket = s3.buckets['your-bucket-name']

def save_image(url)
  filename = File.basename(url)
  obj = @bucket.objects[filename]
  open(url) do |data|
    obj.write(data.read)
  end
end

search_word=URI.encode("cat")
doc = Nokogiri::HTML(open(
  "https://www.flickr.com/search/?q=#{search_word}"))
doc.xpath(
  "//a[@class='rapidnofollow photo-click']/img").each {|link|

  url = link["data-defer-src"]
  # サムネイル画像
  save_image(url)
}
```

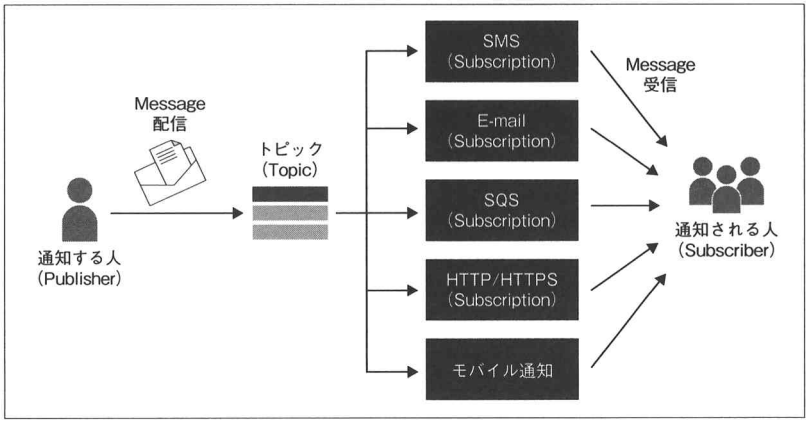
このスクリプトの場合、ファイルの取得に対してS3へのアップロードが圧倒的に時間がかかります。高速化の手段しては、S3へのアップロード部分を分離して別途動かすような形が望ましいでしょう。

6-4-4 Amazon SNSで通知する

少し前であれば、プログラムから通知するのであればE-mailを使うのが一般的でした。しかし、昨今ではスマートフォンへのプッシュ通知やTwilioのようなAPIを利用して直接電話をかけるようなこともできるようになっています。いろいろなデバイスに通知するのに、プロトコルごとに実装するのは手間がかかります。

そこで、AmazonのSimple Notification Service (SNS) のようなサービスを利用することで、プロトコルを透過的に扱うことができます。SNSはプッシュ型の通知サービスで、2014年7月現在では、HTTP/HTTPS、E-mail、SMSとSQS、モバイル通知と5種類があります。

▼ Amazon SNS

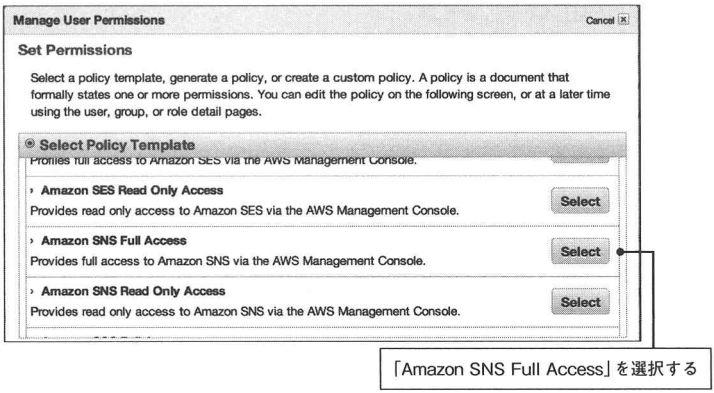


Amazon SNSを利用する準備

SNSを利用するにも、いくつかの準備が必要です。まずは、「6-3-3 Amazon Simple Email Service (SES) を使って結果通知する」で作成したIAMアカウントに対して、SNSの利用権限を付与します。

権限の付与は、AWSアカウントでログインしたうえでAWSコンソールのIAMダッシュボードから行います。「Users」で既存のユーザーを選択し、Manage User Permissionsの「Set Parmissions」で権限を追加します。Select Policy Templateの「Amazon SNS Full Access」を選択し、[Select] を押下します。

▼ SNS利用権限の付与

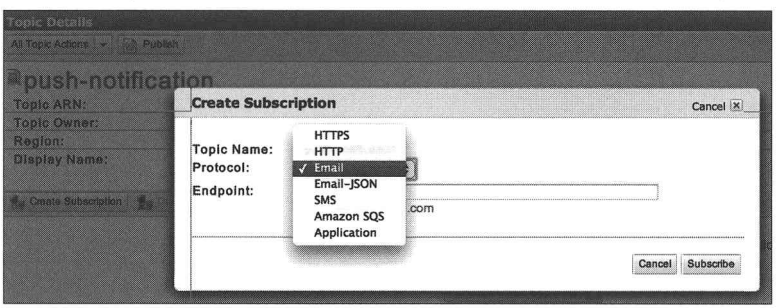


権限を付与したら、次はSNSの「Topic」の作成です。Topicとは、通知の名前です。AWS管理コンソールのSNSのダッシュボードから作成できます。例えば、その通知を使うアプリケーション名やイベント名を付けます。

Topicの次は、Topicに属する「Subscription」を作成します。Subscriptionは通知の方法です。E-mailやHTTPなどさまざまなプロトコルが利用できます。また、1つのTopicに複数のSubscriptionを作成できます。一度の通知で、そのSubscription分だけ通知ができるということになります。

スマートフォンへのプッシュ通知は、Subscriptionのアプリケーションからできます。ただし、iOSやAndroidなど受け手側での準備が必要です。例えば、iOSの場合はプッシュ通知用の証明書の作成などが必要になります。

SNS利用権限の付与



スクリプトから利用する

スクリプトからプッシュ通知するには、作成したTopic ARNを利用します。スクリプト自体は非常にシンプルです。Subscriptionとして用意しておけば、これだけでE-mailやHTTPなど複数のプロトコルに通知できます。

実行の際には、「:access_key_id」と「:secret_access_key」に取得したIAMアカウントのアクセスキーや秘密キーを設定してください。また、SNSのTopic作成時の設定として、エンドポイントを指定します。エンドポイントは、「arn:aws:sns:us-east-1:」に、AWSアカウントIDとトピック名を付与します。なお、エンドポイントは、AWS管理画面のSNSダッシュボードから、対象のTopicを選ぶと「Topic ARN」という項目で表示されています。

Amazon SNSで通知する sns.rb

```
# -*- coding: utf-8 -*-
require 'aws-sdk'

AWS.config({
  :access_key_id => 'AWS_ACCESS_KEY',
  :secret_access_key => 'AWS_SECRET_ACCESS_KEY',
})

topic = AWS::SNS::Topic.new(
  'arn:aws:sns:us-east-1:your_account_id:topic_arn')
topic.publish('Notification from SNS')
```

クローラーとの連携

クローラーと連携する場合は、「6-3-2 Gmailを使って結果通知する」(→p.385)で紹介したように、結果を通知するような形で利用するとよいでしょう。また、SQSのようなキュープログラムと連携できるので、結果を通知して次のプログラムを実行させるということも可能です。

6-5 さらなる高速化の手法

クローラー単体での処理を高速化する方法の1つは、処理時間がかかる部分を並列化することです。「4-4 クローラーを効率的に動かすには」(→p.232)で言及しましたが、クローラーの処理のなかで大きな割合を占めるのが、ダウンロードです。これを非同期処理で多重化することで、クローラー単体での高速化が実現できます。

6-5-1 非同期処理

ダウンロード処理を高速化する方法として、まずは非同期処理のやり方を見ていきましょう。

EventMachine

Rubyで非同期処理をする場合、複数のスレッドを利用するThreadがあります。スレッド方式は、多重度が増えると極端に効率が悪くなるという問題があります。そこで別の方法として、Reactorパターンを利用した「EventMachine」があります。

Reactorパターンは、イベントドリブン方式で、特定のイベント(読み込み可能な状態になる)が発生すると初めて処理を開始します。内部的にはシングルスレッ

ドで動作する、少し変わった非同期処理です。Reactorパターンは、特にネットワークなどのI/O待ち時間が多い場合に効果を発揮します。EventMachineはReactorパターンのRubyの実装の1つです。

■ EventMachine
URL http://rubyeventmachine.com/

EventMachineのインストールは、gemから行えます。インストール後は、gem list eventmachineでバージョンが表示されれば成功です。

◆ EventMachineのインストール

```
$ gem install eventmachine
```

◆ インストールの確認

```
$ gem list eventmachine

*** LOCAL GEMS ***
eventmachine (1.0.3 x86-mingw32)
```

また、EventMachineを利用した実装の1つに、「EM-HTTP-Request」というライブラリがあります。これは、その名のとおりEventMachineを使って非同期にHTTPリクエストするというライブラリです。

■ GitHubのEM-HTTP-Requestページ
URL https://github.com/igrigorik/em-http-request

EM-HTTP-Requestのインストールは、gemから行えます。インストール後は、gem list em-http-requestでバージョンが表示されれば成功です。

◆ EM-HTTP-Requestのインストール

```
$ gem install em-http-request
```

◆ インストールの確認

```
$ gem list em-http-request

*** LOCAL GEMS ***
em-http-request (1.1.2)
```

このライブラリを利用して非同期のダウンロードを実装してみましょう。このスクリプトは、URLのリストを含んだファイルを引数として渡すことで(❶)、リスト内のURLに対してダウンロードを行います。

非同期のダウンロード em_http_download.rb

```
# -*- coding: utf-8 -*-
require 'eventmachine'
require 'em-http'

class Downloader
  def download(url_list_file)
    pending = File.readlines(url_list_file).size
    EM.run do
      File.open(url_list_file) {|f|
        f.each do |url|
          http = EM::HttpRequest.new(url).get
          http.callback {
            puts
              "#{url}¥n#{http.response_header.status} -
               #{http.response.length} bytes¥n"

            pending -= 1
            EM.stop if pending < 1
          }
          http.errback {
            puts pending
            puts "#{url}¥n" + http.error

            pending -= 1
            EM.stop if pending < 1
          }
        end
      }
    end
  end
end

def main(url_list_file) ●―――――❶
  if url_list_file == nil || !File.exist?(url_list_file) then
    abort "set first arg as `url list file'"
  end

  downloader = Downloader.new
  downloader.download(url_list_file)
end
```

```
if __FILE__ == $0
  main(ARGV[0])
end
```

このプログラムに、200ほどのURLのリストを含んだファイルを引数に渡し、実行時間を計測してみます。「url_list.txt」はサンプルファイル（→p. ii）に用意してあるものをご使用ください。

◆ em_http_download.rbの実行例

```
$ time ruby em_http_download.rb url_list.txt
real    0m7.297s
user    0m0.882s
sys     0m0.954s
```

Linuxのtimeコマンドで測定してみます。このコマンドは、timeの後のコマンドの実行時間を測定できます。いくつか値が出てきますが、realが実測値になります。今回の実行時間は、7秒程度です。次に、比較のために、並列ではなく1件ずつ順次処理する場合と比べてみましょう。

■順次処理によるダウンロード　simple_http_download.rb

```
# -*- coding: utf-8 -*-
require 'open-uri'

class Downloader
  def download(url_list_file)
    File.open(url_list_file) {|f|
      f.each do |url|
        puts url
        puts open(url)
      end
    }
  end
end

def main(url_list_file)
  if url_list_file == nil || !File.exist?(url_list_file) then
    abort "set first arg as `url list file'"
  end

  downloader = Downloader.new
  downloader.download(url_list_file)
end
```

```
if __FILE__ == $0
  main(ARGV[0])
end
```

◆ simple_http_download.rbの実行

```
$ time ruby simple_http_download.rb url_list.txt
real    1m38.066s
user    0m0.997s
sys     0m0.291s
```

実行時間は1分38秒でした。圧倒的なスピード差があります。

cosmicrawler

通常の同期処理やスレッドを使った処理に比べると、EventMachineはどうしてもコードの記述量が多くなります。純粋にクローラーとしてEventMachineを利用したい場合は、「cosmicrawler」というRubyのGemライブラリがあります。cosmicrawlerを利用することによりEventMachineの煩雑な記述を隠蔽し、解析部分に注力できます。

■ GitHubのcosmicrawlerページ
URL https://github.com/bash0C7/cosmicrawler

cosmicrawlerのインストールは、gemから行えます。インストール後は、gem list cosmicrawlerでバージョンが表示されれば成功です。

◆ cosmicrawlerのインストール

```
$ gem install cosmicrawler
```

◆ インストールの確認

```
$ gem list cosmicrawler

*** LOCAL GEMS ***
cosmicrawler (0.0.1)
```

次のスクリプトはcosmicrawlerによる非同期処理で、はてなブックマークのHTMLを取得しています。

cosmicrawlerによる非同期処理 cosmicrawler-sample.rb

```
# -*- coding: utf-8 -*-
require 'cosmicrawler'

Cosmicrawler.http_crawl(%w(
  http://b.hatena.ne.jp/hotentry/it
  http://b.hatena.ne.jp/hotentry/life)) {|request|

  get = request.get
  puts get.response if
    get.response_header.status == 200
}
```

◆ cosmicrawler-sample.rbの実行

```
$ ruby cosmicrawler-sample.rb
<!DOCTYPE html>
<html lang="ja" data-page-class="hotentry">
<head>
<meta charset="UTF-8">

~省略~
```

6-5-2 分散処理

前のセクションでは、非同期処理での多重実行によるクローラーの高速化を図りました。この手法の本質はスクリプト内での効率化であり、スピードアップの上限があります。そこでもう1つの方法として、複数のサーバで同時に実行する方法も試してみましょう。いわゆる分散処理です。

キューの活用

分散処理をする場合、処理対象の制御が重要になります。せっかく複数のサーバで処理していても、同じ対象に対して処理をしていれば意味がないからです。

その対処方法は2つあります。1つは、ジョブ制御サーバが複数の処理サーバにタスクを投げる方法です。Hadoopのような分散処理フレームワークはこの形です。もう1つは、処理サーバの方からタスクを取得する方法です。タスクの管理にキュー(待ち行列)などが使われます。Hadoopを使ったクローラーの実装もあるようですが、実際に動かすのはなかなか大変です。

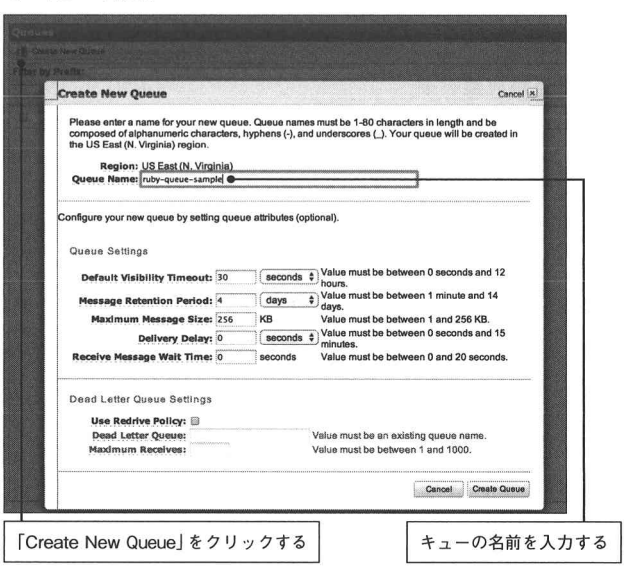
そこで比較的簡単なキューを使った分散処理を見てみましょう。キューとは待ち行列です。先入れ先出しで、先に入れられたデータから順に取り出していきます。キューの実装は、原理的にはファイルやデータベースを使ってもできます。しかし、キューは登録・削除処理や、処理中のキューの排他制御の実装が必要です。またデータが消失しないように可用性や、キュー自体がボトルネックにならないようにスケーラビリティが必要になります。ちゃんとしたキューのシステムを構築しようとすると、かなり大がかりになります。

Amazon Simple Queue Service

AWSには、Amazon Simple Queue Service (SQS) というキューのサービスがあります。SQSはAWSのサービスのなかでも最古参のものの1つです。これを使ってみましょう。

他のAWSサービスと同様に、SQSを利用する際には権限付与が必要です。まずは、「6-3-3 Amazon Simple Email Service (SES) を使って結果通知する」(→p.390) で作成したIAMアカウントに対して、SQSの利用権限を付与します。権限の付与は、AWSアカウントでログインしたうえで、AWSコンソールのIAMダッシュボードから行います。「Users」で既存のユーザーを選択し、Manage User Permissionsの「Set Parmissions」で権限を追加します。Select Policy Templateの「Amazon SQS Full Access」を選択し、[Select] を押下します。

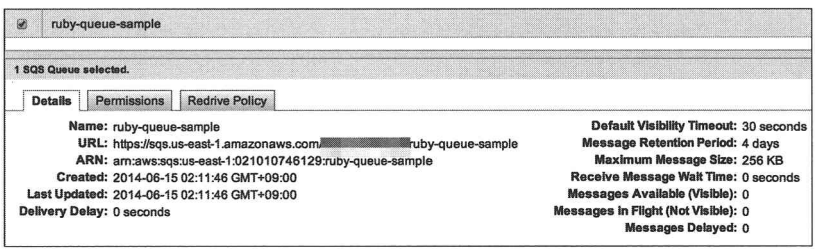
▼キューの作成



権限を付与したら、次はキューを作成します。キューの作成は、SQSダッシュボードから［Create New Queue］ボタンを押下します。作成時にいくつかパラメータ設定がありますが、キュー名を入力するだけで作れます。

無事作成できたら、選択して詳細を見てみましょう。「Details」タブにURLという項目があります。このURLがキューを操作するためのエンドポイントになります。

SQSの詳細



キューの登録

それでは、実際にキューに登録してみましょう。今回は、URLのリストを格納したファイル（url_list.txt）を用意し（❶）、そのなかのURLをすべてキューに格納します。

実行の際には、「:access_key_id」と「:secret_access_key」にIAMアカウントのサクセスキーと秘密キーを、「url」にエンドポイントを設定してください。URLのリストを格納したファイルは、サンプルファイル（→p.ii）にあるものをご使用ください。

また、aws-sdkは既にインストールしてあるものとします（→p.392）。

キューを登録する sqs-put-url.rb

```
# -*- coding: utf-8 -*-
require 'aws-sdk'

AWS.config(
  :access_key_id => 'AWS_ACCESS_KEY',
  :secret_access_key => 'AWS_SECRET_ACCESS_KEY',
  :sss_endpoint => 'sqs.ap-northeast-1.amazonaws.com'
)

url = 'AWS_SQS_ENDPOINT'

sqs = AWS::SQS.new
```

```
# URLリストを登録
f = open("url_list.txt") ●―――――❶
f.each {|line|
  sqs.queues[url].send_message(line)
}
```

キューを使った処理

キュー登録が完了したら、次はキューからURLを取得して処理をするスクリプトを作成してみましょう。基本的な流れとしては、次の3つの手順になります。

- ①キューからメッセージ（URL）の取得
- ②URLに対して処理
- ③（処理が成功したら）キューからメッセージ（URL）の削除

Amazon SQSは、キューからメッセージを取得した場合、そのメッセージについては一定時間他から見えなくなります。そのため、複数台のサーバから同時にキューを取得されても、同じメッセージを処理するといった問題は発生しません。SQSの接続URLは、AWS管理画面のSQSから、キューの詳細画面のURLを参照しセットしてください。

キューからメッセージを取得して処理を行う　sqs-pull-url.rb

```
# -*- coding: utf-8 -*-
require 'aws-sdk'

AWS.config(
  :access_key_id => 'AWS_ACCESS_KEY',
  :secret_access_key => 'AWS_SECRET_ACCESS_KEY',
  :sss_endpoint => 'sqs.ap-northeast-1.amazonaws.com'
)

url = 'AWS_SQS_ENDPOINT'

sqs = AWS::SQS.new

sqs.queues[url].poll do |msg|
  puts msg.body
  # 受け取ったURLを元に何らかの処理を実装
end
```

◆ sqs-pull-url.rbの実行例

```
$ ruby sqs-pull-url.rb
http://ja.wikipedia.org/wiki/Help:%E8%A8%98%E4%BA%8B%E3%81%A8%E3%81%AF%E4%BD%95%E3%81%8B
http://ja.wikipedia.org/wiki/%E7%B5%B1%E8%A8%88
http://ja.wikipedia.org/wiki/Wikipedia:%E7%A7%80%E9%80%B8%E3%83%94%E3%83%83%E3%82%AF%E3%82%A2%E3%83%83%E3%83%97
http://ja.wikipedia.org/wiki/%E3%83%AA%E3%83%93%E3%82%A2
http://ja.wikipedia.org/wiki/Wikipedia:%E8%89%AF%E8%B3%AA%E3%81%AA%E8%A8%98%E4%BA%8B
```

～省略～

なお、このスクリプトは実行しているかぎり、延々とキューを待ち続けます。必要に応じて、スクリプトを終了させる処理も追加しましょう。

※

AWS上に作るのであれば、SQSとEC2を組み合わせてスケーラビリティのあるクローラーを作成できます。取得先の負荷の問題がないのであれば、数十台のサーバを並べて並列で処理するといったことも簡単にできます。

6-6 変化に対応する

クローラーは、本質的には後追いの技術です。対象とするサイトの構造やHTMLが変更された場合、後手に回った対応しかできません。そのため、クローラーを運用をするには、いかに変化を検知し対処できるかをあらかじめ考えておく必要があります。

検知方法としては、想定の結果が出なかった場合にクローラー自身に気づかせることと、その結果を通知することが必要になります。また、検知後のクローラーの修正と、修正後の再実行についても考えていきます。

6-6-1 検知方法

クロール対象の変化のパターンとして、大きく2つに分けることができます。

- 対象のページがなくなる
- HTMLの構造が変化する

1つ目の対象のページがなくなることは、対処がしやすいです。スキップしておけば、基本的には影響はありません。あとは、ページがなくなったということに対しての通知だけをしておけばよいでしょう。

問題となるのが、2つ目のHTMLの構造が変化する場合です。これについては影響が大きくなることがあるので、ケースごとに考えていきましょう。

取得結果の検証

HTMLの構造が変化した場合、次の2つの影響が考えられます。

- 値が取れなくなる
- 間違った値が取れる

値が取れなくなる問題については、すぐに気がつけるので問題は少ないです。しかし、後者の間違った値が取れるというのは、あらかじめ想定しておかないと気がつきにくい問題です。

その対処方法としては、スクレイピングした値の検証をするということが考えられます。例えば、数値を取るはずの場合に、それ以外の値が入っていれば異常ということになります。いわゆるバリデーション(varidation)です。次のスクリプトは、価格を取得した結果が数字以外の場合はエラーとして扱うという例です。

数字以外だとエラーにする

```
# 価格
price = element.xpath("td[@class='price']").text
price = price.gsub(/円/,'').gsub(/,/,'')

# 数字以外はエラー
if price.is_a?(Integer)
  detail['price'] = price
else
  # エラー処理
end
```

値の判定のパターンとしては、文字列や数値の判定だけではありません。必要に応じて、取りうる値の範囲検証であったり、電話番号やチェックサムが合っているかなどのルールベースの検証などがあります。検証方法は、正規表現などを使って自前で行う方法や、ライブラリを使う方法があります。ライブラリの場合、Rails

のActive RecordのValidationsが有名です。それ以外にも単体でも使えるライブラリがいくつかあります。

Gemのリストコマンドに-rを付けると、利用可能なGemの一覧が表示されます。検索条件にvalidateを入れるとある程度絞り込めます。目的の検証用のライブラリがあるか、まずは探してみるとよいでしょう。

◆ 検証用のライブラリを探す

```
$ gem list -r validate
validate (1.3.3)
validate-response (0.0.5)
validate-website (0.7.9)
validate_as_email (2.1.0)
validate_as_seo_tag (0.0.3)
validate_as_url (0.0.10)
validate_block (0.2.0)

~中略~

validates_vat_number (0.2)
validates_website (0.1.0)
validates_xml (1.0.3)
```

安全に止める

取得結果の検証の結果、想定の値が取れなかった場合の対処はどうすればよいのでしょうか？　選択肢として、スキップしてそのまま続ける方法と、処理自体を止めてしまう方法があります。どちらが適切なのかは、クローラーの目的によって異なります。

複数サイトを巡回するようなクローラーであれば、1つのサイトの問題で全体を止めるのはよくありません。スキップするのが正解でしょう。逆に、特定サイトを巡回している場合は、スキップしてもまた同じ問題に遭遇する可能性が高いです。そのため、全体のクローリング処理を止めてしまう方がよいです。

問題が発生した場合の止め方も、いくつかあります。簡単な方法としては、rescueを使って例外処理にしてしまうというものがあります。Rubyの制御構文の1つであるrescueは、例外（exception）が発生した時に処理を引き受けます。また、raiseコマンドを使うことにより強制的に例外を発生させることも可能です。

rescueで処理を停止する

```
begin
  # クローリング処理本体
  # 値判定
  if price.is_a?(Integer)
    detail['price'] = price
  else
    # 例外を発生させる
    raise #=> RuntimeError:
  end
rescue
  # 例外時は、異常終了させる
  exit 1
end
```

通知

例外処理などを使ってクローラーを止めたとしても、その事実を通知しないと気がつかずにスルーされてしまいます。それを防ぐために、何らかの事態が起きた場合には通知するのが望ましいでしょう。

通知についても、2種類の方法があります。プログラム自体に通知させる方法と、プログラムを起動するジョブ管理のシステムに通知させる方法です。ジョブ管理のシステムとしては、Crondのようなものや、もっと高度なHinemosのような運用管理の仕組みもあります。プログラム自身に通知させる方法としては、「6-3 収集結果をメールで自動送信する」(→p.384)や「6-4-4 Amazon SNSで通知する」(→p.407)で紹介したようなメールや通知サービスを利用する方法があります。

6-6-2　修正＆再処理

クローラーの失敗を受け取った次は、原因を究明して対処する必要があります。原因の究明と対処については、新規のクローラー作成と同じなので時間をかければ対応できるでしょう。

問題は、再実行についてです。あらかじめ再実行のことを想定していないと、成功しているところを含めて全部再実行しないといけない場合もあります。その場合、非常に無駄が発生します。クローラーを作成する場合は、あらかじめ失敗したことを想定し、再実行方法を考えておく必要があります。

再実行方法としては、いくつかあります。まず、どこまで処理しているのかという情報があれば、そこから再実行する方法があります。また、単純に再実行してもデータ的に問題ないような作りにする方法もあります。それぞれのケースで考えていきます。

単純再処理

再実行方法の1つ目としては、単純に再実行した場合でも、データの不整合が発生せず補正する必要もない状態にすることです。そのためには、データベースなどに格納する前にデータのチェックを行い、取得ずみのデータが存在する場合と存在しない場合に分けて処理を記述します。

下記の例は、物件情報を収集し、取得ずみのデータの場合はデータ保存しないようにしています。このような作りであれば、再実行した際にも同じデータが2件重複して登録されることもありません。

取得ずみのデータを確認する

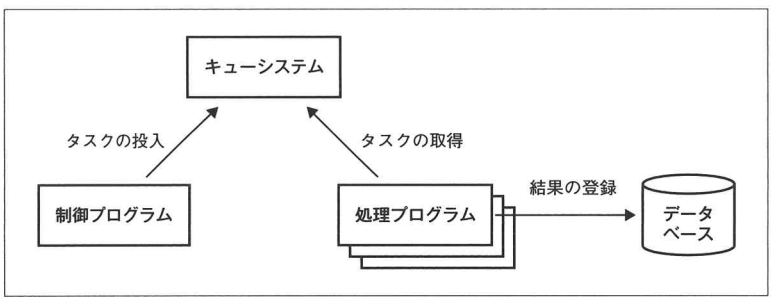
```
# 物件情報の登録
def set_bukken_spec(spec)
  id = get_bukken_id(spec['url'])
  if id.nil?
    query = "INSERT INTO housing(name,url,address,
             access,distance,age,floor_number)
             VALUES('#{spec['name']}','#{spec['url']}',
             '#{spec['address']}','#{spec['access']}',
             '#{spec['distance']}','#{spec['age']}',
             '#{spec['floor']}')"
    puts query
    @client.query(query)
    id = get_bukken_id(spec['url'])
  end
  return id
end

# 物件IDの取得
def get_bukken_id(url)
  id = nil
  @client.query(
    "select id from housing where url = '#{url}'").each do |row|
    id = row["id"]
  end
  return id
end
```

再実行を考える一番のポイントは、データのキーが何になるかです。それが日付であったりURLやそれ以外の場合もあります。そこを見極めるのがクローラーを構築・運用するうえでのコツになります。

処理進行情報の管理

もう1つの方法としては、タスクの進行情報をスクリプト自身以外の何かで管理させる方法です。例えば、処理予定のURLを別の仕組みで管理することにより、初回実行時か再実行時かは何も考えずに、与えられたURLを受け取って処理するような作りにします。

キューを使った構成



筆者がよく取る構成として、タスク投入部分(例えば取得対象のURLを登録)と処理部分を完全に分離します。そして、そのタスクの中継をキューのシステムで行います。AWSを利用する場合は、EC2とSQSを利用します(p.398、416の例をご参照ください)。

この構成にしておけば、たとえサーバ自体が途中でダウンしても、何のエラー処理もせずに別のサーバに引き継げます。また、複数台のサーバで処理することも可能になります。タスク情報を分離することは、システムを分離しそれぞれを疎結合にすることです。そのことが再実行性を高め、運用を簡単にします。

6-7 クローラーとそれに付随する技術

本書はクローラーに関するものです。そのため、データを収集する方法についてのみ解説してきました。しかし、クローラーを利用する背景には、データを収集してそれを活用することがあります。つまり、クローラーでデータを取得した後の工程の方がより重要なのです。データの活用については、昨今ではビックデータというキーワードで語られています。本書の終わりを迎えるにあたって、少しだけデータ解析の手法を紹介しておきましょう。

6-7-1 データを活用する方法

収集したデータを活用する方法として、データの可視化やデータマイニングなどがあります。

データの可視化は、数値などの人間にとって直感的にわかりづらいものを、グラフや図などに変換することにより視覚化してわかりやすくする技術です。大量に集めたデータを1枚の図に表現することにより、「セレンディピティ」別の価値の発見に繋がりやすくします。

データマイニングは、統計学などを駆使してデータの山のなかから新たな知識を抽出することです。抽出には数式などを駆使して機械的に行う面もありますが、仮説を元にデータを検証して発見するといったことも多いでしょう。それぞれの手法について、概要を紹介します。

6-7-2 データの可視化

一口に可視化と言っても非常に範囲が広いです。例えば、エクセル上のデータをグラフ化するのも可視化ですし、表にまとめるのも可視化の1つです。また日常の業務をフローに落とし込むのも可視化の1つでしょう。ここでは、クローラーと可視化という観点で考えてみます。

地図データとの組み合わせ

クローラーで集めたデータとの組み合わせの1つに地図があります。地図データと、市町村ごとの売上データや地価、賃貸の相場などを組み合わせると、とたんに数字が見やすくなります。しかし、地図データや地理情報のデータを自分で一から用意するのは大変です。そこでオープンソースのツールを使うことをお勧めします。

地図情報を扱うツール

地図情報を扱うためのツールに、「PostGIS」があります。PostgreSQLデータベースで地理空間情報を扱うための拡張で、GIS（地理情報システム）をオブジェクトとして格納することができます。

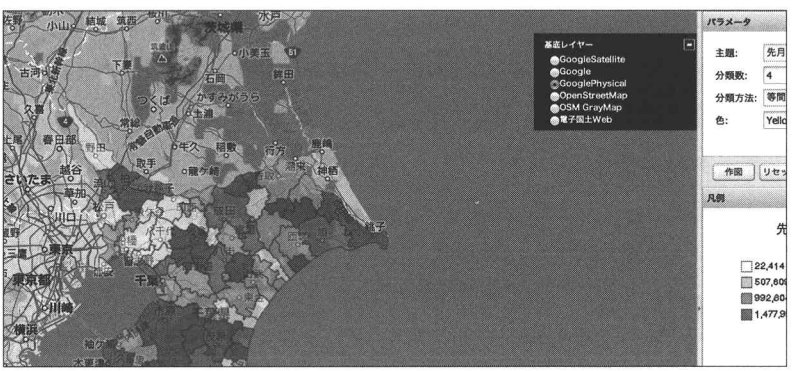
■ PostGIS
URL http://www.finds.jp/docs/pgisman/2.0.0/postgis.html

このPostGISを利用したアプリケーションを使うことにより、比較的簡単に地図との組み合わせができます。現在のところ、Ruby製の地図を使った可視化アプリケーションでよいものは少ないです。Ruby以外に目を向けると、「GeoFuse」などがあります。

■ GitHubのGeoFuseページ
URL https://github.com/mbasa/geofuse

GeoFuseはTomcatとPostgreSQLを利用するJavaアプリです。ローカル環境にインストール可能ですが、Javaに馴染みがない人には少々手間がかかるかもしれません。いくつかデモサイトも提供されているので、まずはそちらを試してみるのがよいでしょう。

▼ 地図データを利用した可視化



6-7-3　データマイニング

クローリングで収集できるデータは、文章などのテキスト情報と株価や気温などの数字情報があります。データマイニングは、どちらの情報も取り扱うことができます。

データ分析

クローラーを使ったデータ分析として、製品や技術に対する盛衰を調査するといったことができます。調査対象に関するキーワードについて、GoogleやTwitterでの日々の出現数、またそれに関連するメディアに対して記事タイトルや本文に含まれるかなどを計測します。取得した値に対して、重みづけして点数化した推移を見てると、意外なほど正確に流行り廃れが浮かび上がってきます。

これ以外にも、クラス分析や回帰分析、クラスタリングなどの統計処理など、データマイニングにはさまざまな手法が存在します。データマイニングには、どのケースにも適用できるという万能の手法は存在しません。いろいろな手法の存在を

知っていれば、選択の幅は広がります。すべての方法を身につけるのは難しいですが、代表的なアルゴリズムの存在を知っていれば、必要に応じて学習するということもできるでしょう。

自然言語処理

データマイニングのなかで、テキストを対象とするものを「テキストマイニング」と言います。また、コンピュータに人間が日常的に使っている言語を処理させることを「自然言語処理」と言います。クローラーが収集したブログがTwitterなどの自然言語を、統計的に処理することにより見えてくるものはたくさんあります。

自然言語処理の単純な使用例としては、「5-13 流行をキャッチする」(→p.314)で紹介したようなキーワードごとの出現頻度といったものがあります。形態素解析器などを組み合わせて、文章をキーワードに分割するといった方法を利用します。

もう少し高度な例としては、クラスタリングと組み合わせる手法があります。文章のクラス分類は、技術や政治・経済、暮らしなどカテゴリに対して、それぞれ元となる学習データ(文章)を用意します。文章内の単語の出現頻度で、どの単語がどのカテゴリに出現しやすいかのデータを作成します。そういったデータを蓄積することにより、新しい文章が出た時にも瞬時に分類することが可能になります。

分類のアルゴリズムとしては、単純ベイズ分類器(Naive Bayes classifier)やサポートベクターマシン(Support vector machine)などがあります。クローラーとの組み合わせ方としては、収集したデータを機械的に分類することにより人間の手間を省くなどが考えられます。

Index

◆ た行

◆ な行

◆ は行

おわりに

本書を執筆中に、筆者は東京に転勤することになりました。引越の際には、まずは住むところを決める必要があります。住居を決めるためには、周辺の環境や、通勤時間、家賃などいくつかのファクターがあります。

周辺の環境は、定性的な部分があるので実際に見てみないとわからない部分があります。しかし、通勤時間や家賃については、データを収集すればおおよそのことがわかります。そこで、筆者は賃貸サイトから候補地の物件情報をすべて抽出し、平方メートル辺りの家賃単価や築年数ごとの相場を調べました。その結果により、相場にくらべてコストメリットの高い地域を選定のうえで、効率的に物件を探すことができました。

卑近な例ですが、クローラーを使って実生活に活かす例の1つです。クローラーを作成する際に、筆者はいつもPerlを生み出したラリー・ウォールによるプログラマの3大美徳を思い出します。

1. 怠慢(Laziness)
2. 短気(Impatience)
3. 傲慢(Hubris)

怠惰というのは、全体の労力を減らすために、役立つプログラムを作成する気質を指します。また短気というのは、手作業で根気よく作業をするのではなく、プログラムで一気に解決するような気質を指します。このあたりを考えると、クローラーはまさにプログラマの美徳を体現するものです。

本書で紹介した手法も、クローラーのさわりの部分でしかありません。ぜひ、自分なりの利用方法を生み出して、自分が楽になるために、ひいては自分の生活を豊かにするためにクローラーを活用してください。そしてクローラーにとどまらず、さらに可視化やデータマイニングなどの技術を習得し、技術者として次のステップに進まれることを望みます。

2014年8月　佐々木拓郎

■本書サポートページ

本書内で紹介したサンプルスクリプトは、下記のURLよりダウンロード可能です。
また、本書をお読みいただいたご感想、ご意見をお寄せください。

http://isbn.sbcr.jp/80354/

○著者プロフィール

佐々木拓郎

本業は、Web系のシステムアーキテクト。企画から設計開発、運用まで幅広く担当。最近はクラウド×自動化をテーマに、できるだけ楽することを考えている。休日はワインを飲みながら、趣味でアプリ開発をしている。またAWSやRubyをはじめとする、いろいろなコミュニティに出没している。

るびきち

RubyとEmacsとw3mとScreenとratpoisonとLinuxがないと生きていけないガチガチCUI系フリーライター。テキストブラウザw3mで快適にWebを駆け回るために多数の個人用クローラーを開発。主な著書に『Ruby逆引きハンドブック（C&R研究所）』『Emacsテクニックバイブル（技術評論社）』。メルマガ『Emacsの鬼るびきちのココだけの話』毎週土曜日発行。

http://rubikitch.com/

Rubyによるクローラー開発技法
巡回・解析機能の実装と21の運用例

2014年8月28日　初版第1刷発行

著者……佐々木拓郎　るびきち
発行者……小川 淳
発行所……SBクリエイティブ株式会社
〒106-0032 東京都港区六本木2-4-5
TEL 03-5549-1201（営業）
http://www.sbcr.jp/

印刷……株式会社シナノ
組版……クニメディア株式会社
装丁……一瀬錠二（Art of NOISE）

Printed In Japan　ISBN978-4-7973-8035-4

ISBN978-4-7973-8035-4
C0055 ¥2980E

定価 本体2,980円 +税

Rubyによる クローラー開発技法

巡回・解析機能の実装と21の運用例