実行例

```
> ruby nokogiri-detectrss.rb http://www.ruby-lang.org/ja/
detect: http://www.ruby-lang.org/ja/feeds/news.rss
2009/11/03:2009年11月・12月開催の地域Ruby会議
          (Regional RubyKaigi)のおしらせ
2009/10/07:rubyist.netのサービス停止
2009/10/06:2009年10月開催の地域Ruby会議(Regional RubyKaigi)の
          おしらせ
2009/09/13:Rubyist Magazine 0027号 発行
2009/09/04:RubyWorld Conference
2009/07/20:Ruby 1.9.2 preview1リリース
2009/07/20:Ruby 1.9.1-p243リリース
2009/06/10:BigDecimal の DoS 脆弱性
2009/05/12:Ruby 1.9.1-p129リリース
2009/04/16:Ruby 1.8.7-p160 リリース
```

トップページのURLを渡しただけで，そのページのRSSの中身を出力することができました。

メモ　Nokogiriにはここで紹介した以外にも多数の便利な機能が用意されています。詳しくはNokogiri公式サイト（http://nokogiri.org/）にあるチュートリアルなどを参考にしてください。

第23章 HTTPサーバのアクセスログ解析

23.1 Apacheのログを解析する

ネットワークで何かのサービスを提供するプログラムを，一般に「サーバ」と呼びます。たいていのサーバは「ログ」というデータを作っています。ログとは，そのサーバがどのように振る舞ったか，という記録で，何かトラブルが発生した際，またはサーバの能力を計る際などに，サーバ管理者はこれを参考にします。

WWWで，HTMLファイルやその他のデータを提供する「Webサーバ」も，もちろんサーバの一種なので，その動作の記録をログとして残しています。しかし，ログはアクセスがあるたびにどんどん吐かれていくので，すぐに数メガバイト，あるいは数ギガバイトまでたまってしまうサイトもあります。このようなサイトでは，大量のデータの中から欲しい情報だけを取り出して，見やすい形にすることが必要です。そのために行われるのがアクセスログ解析です。

この章ではアクセスログを解析するスクリプトを作っていきます。

23.2 アクセスログの概要

アクセスログを解析するには，アクセスログがどのような情報をどのように記録しているのかを知らなければなりません。ここでは，Apache Webサーバの標準的なアクセスログフォーマットの概要を説明します。

実際のアクセスログの1行は，次のようになっています。

```
192.168.100.34 - - [14/Jan/2010:02:16:10 +0900] "GET /top/
main.html HTTP/1.0" 200 26971 "http://www.example.org/"
"Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)"
```

このメッセージは，先頭から順番に表23.1の情報を表しています。

表 23.1 アクセスログの情報

例	意味
192.168.100.34	アクセス元のIPアドレス
-	identによるログネーム（省略時は「-」）
-	認証を行った際のアカウント（省略時は「-」）
[14/Jan/2010:02:16:10 +0900]	アクセスされた日時とタイムゾーン
"GET /top/main.html HTTP/1.0"	クライアントからのリクエスト
	例 / 意味
	GET / HTTPメソッド（GET,POSTなど）
	/top/main.html / URL
	HTTP/1.0 / HTTPバージョン
200	応答ステータス（「200」なら成功）
26971	ヘッダを除く応答として送信したバイト数（データを送信しなかった場合は「-」）
"http://www.example.com/"	リンク元のURL（リファラ）
"Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)"	使用ブラウザ（ユーザエージェント）

アクセスログの1行は，Webブラウザからの1アクセスに対応します。つまり，行数の数だけこのサイトにアクセスがあった，ということです。

もっとも，これだけ項目はありますが，アクセスログ解析を行う際に全部必要になるわけではありません。「アクセスの日時」と「クライアントからのリクエスト」の2つだけで十分ということもあるでしょう。場合によってはリファラやレスポンスコードからも有益な情報が得られます。

メモ　このログフォーマットは「Combined Log Format」といいます。仕様は以下のWebページで確認することができます。

- ログファイル － Apache HTTP サーバ
  http://httpd.apache.org/docs/2.2/ja/logs.html

23.3 アクセス数を調べる

一番簡単なことは，アクセス数，つまり全体の行数を調べることでしょうか。

先ほど述べたように，1行が1アクセスに該当します。ですから，その行数を調べることが，アクセス数を調べることに相当します。

Unixでは，第9章でも少し触れた，「wc」というファイルの行数を数えるためのコマンドがあります。このコマンドを使えば，プログラムを書く必要はありませんが，Windowsではこのようなツールが普及していません。そこで（ちょっと強引ですが）Rubyの出番となります（List 23.1）。

List 23.1 accesslog_count.rb

```
1: count = 0                          # 行数を初期化する
2: File.open(ARGV[0]) do |io|         # ファイルを開く
3:   io.each_line do |line|           # 行ごとに処理する
4:     count += 1                     # 行数を更新する
5:   end
6: end
7: puts count                         # 行数を表示する
```

実行例

```
> ruby accesslog_count.rb sample-access.log
1484
```

このプログラムは，まず，1行目で変数countに初期値として0を設定します。2行目でコマンドライン引数で与えられたファイルを開いて，3行目から5行目のeach_lineメソッドでデータを1行読み込むごとに1ずつ足しています。ファイルの最後まで読みったときには，アクセスログの行数，つまりサーバへのアクセス数が得られるというわけです。

23.4 アクセスログを解析する

List 23.1では行数数えるだけで，読み込んだデータを捨てていました。このデータを捨てずに解析することを考えましょう。先ほど紹介したログフォーマットを見ると，先頭のIPアドレス，ログネーム，アカウントは問題ないのですが，それ以降のデータには空白が含まれていることがわかります。単純にsplitメソッドを使って空白で区切るだけでは個々の値を取り出すことができませんから，今回は正規表現を使ってみます。

フォーマットの各要素にマッチするパターンを見ていきましょう。

23.4.1 IPアドレス，ログネーム，アカウントのパターン

これらの項目には通常は空白を含みません。空白を含まない文字列にマッチするパターンは次のようになります。

```
(\S+)
```

「空白以外の1文字以上の並び」を表すパターンです。キャプチャするために全体を括弧でくくっています。

23.4.2 時刻とタイムゾーンのパターン

時刻とタイムゾーンは「[時刻 タイムゾーン]」という形式になっています。「[ ]」の中は『「]」を含まない文字列』と表現すれば閉じ括弧の手前までマッチできます。

『「]」を含まない文字列』を表すパターンは次のようになります。「]」は文字のグルーピングに使われる文字なので，「\]」そのものを表すためにエスケープしています。

```
[^\]]+
```

これをキャプチャするために括弧で括って，前後に「[ ]」を追加すると次のようなパターンになります。


```
\[([^\]]+)\]
```

23.4.3 クライアントからのリクエストのパターン

リクエストは少し複雑です。この項目は「"メソッド URL バージョン"」という形式になっていて，「" "」で囲まれた部分がさらに空白で区切られています。個別に値を取得できるようにすると以下のようなパターンになります。

```
"(\S+)\s(\S+)\s(\S+)"
```

23.4.4 応答ステータス，送信バイト数のパターン

応答ステータスは数字の並びなので，次のパターンで十分です。

```
(\d+)
```

送信バイト数は数字の並びまたは「-」です。これも難しくはありません。

```
(\d+|-)
```

23.4.5 リファラ，ユーザエージェントのパターン

これらは「" "」で囲まれた文字です。「"」を含まない文字列のパターンは次のようになります。

```
"([^"]*)"
```

23.4.6 パターンをまとめる

ここまでのパターンをまとめると次のようになります。

```
CLF_REGEXP = /
  \A                          (?# 行頭)
  (\S+)\s                     (?# 1 address)
  (\S+)\s                     (?# 2 ident)
  (\S+)\s                     (?# 3 authuser)
  \[([^\]]+)\]\s              (?# 4 time)
  "(\S+)\s(\S+)\s(\S+)"\s     (?# 5 6 7 method url version)
  (\d+)\s                     (?# 8 status)
  (\d+|-)\s                   (?# 9 bytes)
  "([^"]*)"\s                 (?# 10 referer)
  "([^"]*)"                   (?# 11 user-agent)
  \Z                          (?# 行末)
/x
```

正規表現リテラルの最後の「x」は正規表現のオプションで，「正規表現の中の空白を無視する」という意味を持ちます。このオプションのおかげで，正規表現を改行したりインデントしたりして，読みやすいように整形できます。代わりに本当に空白にマッチさせたい場合は「\s」と記述しなければいけません。

パターン全体CLF_REGEXPには，ここまでに解説してきた各要素に対するパターンに，文字列の先頭を表す「\A」，文字列の末尾を表す「\Z」，各要素の間の空白を表す「\s」を加えています。

文字列の末尾にマッチする正規表現には「\Z」と「\z」があります。「\Z」は文字列の末尾に改行文字がある場合にその手前までマッチします。一方，「\z」は改行文字の有無に関わらず文字列の末尾にマッチします。今回はログファイルから取り出した1行に対してマッチングを行うので改行文字が含まれていますが，データとしては意味がないので無視するために「\Z」を使っています。

「(?# 〜)」の部分はコメントです。行ごとに何を表しているか，コメントを書いています。長い正規表現はあとで読み返すのが大変になりがちなので，パターンの区切りのよいところで改行したりコメントを入れたりしておくとよいでしょう。

なお，CLF_REGEXPという定数名の「CLF」は，フォーマット名の「Combined Log Format」を元にしています。

23.4.7 パターンを試す

さっそく簡単なプログラムでパターンを試してみましょう。List 23.2は，List 23.1と同じ要領でアクセスログを読み込んで，解析した結果を表示します。lineがCLF_REGEXPにマッチしたときに使っている$~は，$1，$2などと同様に直前にマッチした正規表現に関する情報を持ったオブジェクトです。詳しくは「コラム：MatchDataオブジェクト」(p.466)を参照してください。

List 23.2 accesslog_parse.rb

```
CLF_REGEXP = /
  \A                          (?# 行頭)
  (\S+)\s                     (?# 1 address)
  (\S+)\s                     (?# 2 ident)
  (\S+)\s                     (?# 3 user)
  \[([^\]]+)\]\s              (?# 4 time)
  "(\S+)\s(\S+)\s(\S+)"\s     (?# 5 6 7 method url version)
  (\d+)\s                     (?# 8 status)
  (\d+|-)\s                   (?# 9 bytes)
  "([^"]*)"\s                 (?# 10 referer)
  "([^"]*)"                   (?# 11 user_agent)
  \Z                          (?# 行末)
/x

count = 0                     # 行数を初期化する
File.open(ARGV[0]) do |io|    # ファイルを開く
  io.each_line do |line|      # 1行ごとに処理する
    if CLF_REGEXP =~ line     # 正規表現にマッチしたら
      p $~.captures           # キャプチャした部分を表示する
    end
    count += 1                # 行数を更新する
  end
end
puts count                    # 行数を表示する
```

実行例

```
> ruby accesslog_parse.rb sample-access.log
["10.250.64.178", "-", "-", "11/Feb/2010:01:55:25 -0500",
"GET", "/r/201002a.html", "HTTP/1.0", "200", "31062",
"http://www.rubycolor.org/r/topics.html", "Mozilla/4.0
(compatible; MSIE 5.5; Windows NT 4.0; T312461)"]
["10.41.149.19", "-", "-", "11/Feb/2010:01:56:59 -0500",
"GET", "/r/201002a.html", "HTTP/1.0", "200", "31062",
"http://www.rubycolor.org/r/topics.html", "Mozilla/4.0 (compa
tible; MSIE 5.5; Windows NT 4.0; T312461)"]
  ⋮
["10.80.135.114", "-", "-", "16/Feb/2010:01:43:59 -0500",
"GET", "/r/index.html", "HTTP/1.0", "200", "38848", "-",
"Mozilla/3.01 (compatible;)"]
1484
```

パターンによって解析された各項目が配列に格納されていることが確認できます。

Column

MatchDataオブジェクト

正規表現でキャプチャした部分には，$1，$2などの変数でアクセスできますが，数が多くなった場合にまとめて扱うのは少し不便です。正規表現のマッチングを行うと，「$~」という変数にマッチングの結果に関する情報が設定されます。この情報はMatchDataオブジェクトです。「$~.captures」で，$1，$2，……をまとめて配列として取得することができます。

```
/(\w+)-([\d+\.]+)/ =~ "ruby-1.8.7"
p $~.captures          #=> ["ruby", "1.8.7"]
```

$~はマッチングの際に自動的に設定されますが，正規表現オブジェクトのmatchメソッドの返り値としてMatchDataオブジェクトを得ることもできます。このメソッドはマッチしない場合はnilを返します。


```
if m = /(\w+)-([\d+\.]+)/.match("ruby-1.8.7")
  p m.captures          #=> ["ruby", "1.8.7"]
end
```

MatchDataオブジェクトのメソッドを表に挙げます。

表 MatchDataオブジェクトのメソッド

メソッド	説明
m[*n*]	*n*=0はマッチした部分全体，*n*=1は$1，*n*=2は$2，……に対応
m.begin(*n*)	*m*[*n*]の元の文字列上での開始位置
m.captures	キャプチャした文字列の一覧（マッチした全体は含まない）
m.end(*n*)	*m*[*n*]の元の文字列上での終了位置
m.length	*m*[*n*]でアクセスできるデータの総数
m.offset(*n*)	*m*.begin(*n*)と*m*.end(*n*)を要素とする配列
m.post_match	マッチした部分の前の文字列
m.pre_match	マッチした部分の後の文字列
m.regexp	比較に使用した正規表現（Ruby 1.9より）
m.size	*m*.lengthと同じ
m.string	比較に使用した文字列
m.values_at(*n*)	*m*[*n*]と同じ

23.5 AccessLogモジュール

正規表現を使いやすくするためにライブラリを作成します。先ほど作った正規表現を使ってログファイルを解析するときに便利なメソッドをモジュールにまとめます。モジュール名はAccessLogにしましょう。（List 23.3）

List 23.3 access_log.rb

```
1: module AccessLog
2:   CLF_REGEXP = /
3:     \A                    (?# 行頭)
4:     (\S+)\s               (?# 1 address)
5:     (\S+)\s               (?# 2 ident)
6:     (\S+)\s               (?# 3 user)
```

```
 7:     \[([^\]]+)\]\s              (?# 4 time)
 8:     "(\S+)\s(\S+)\s(\S+)"\s     (?# 5 6 7 method url version)
 9:     (\d+)\s                     (?# 8 status)
10:     (\d+|-)\s                   (?# 9 bytes)
11:     "([^"]*)"\s                 (?# 10 referer)
12:     "([^"]*)"                   (?# 11 user_agent)
13:     \Z                          (?# 行末)
14:   /x
15:
16:   Entry = Struct.new(    # 解析結果を保存するためのクラス
17:     :address, :ident, :user, :time,
18:     :method, :url, :version, :status, :byte,
19:     :referer, :user_agent
20:   )
21:
22:   module_function
23:
24:   def each_entry(file)
25:     file.each_line do |line|
26:       if entry = parse(line)
27:         yield(entry)
28:       end
29:     end
30:   end
31:
32:   def parse(line)
33:     if m = CLF_REGEXP.match(line)
34:       return Entry.new(*m.captures)
35:     end
36:     $stderr.puts("parse failure: #{line.dump}")
37:     return nil
38:   end
39: end
```

16行目のEntryは正規表現によって解析した結果を保存して，名前でアクセスできるようにするためのクラスの定義です。Structについては「コラム：Structクラス」(p.470)を参照してください。

22行目でmodule_functionメソッドを呼んで，それ以降に定義されたメソッドをモジュール関数として利用できるようにします。

24行目のAccessLog.each_entryメソッドは，引数fileとして渡されたIO（またはARGFやString）から読み取った行を，AccessLog.parseメソッドによって解析します。解析の結果得られたAccessLog::Entryオブジェクトをブロック引数としてブロックを起動します。

26行目のAccessLog.parseメソッドは，引数lineで渡された文字列を正規表現を使って解析します。正規表現にマッチした場合はAccessLog::Entryオブジェクトを生成して返します。lineが正規表現にマッチしなければ，エラーメッセージを出力してnilを返します。

List 23.2のaccesslog_parse.rbと同じような動作をするスクリプトはList 23.4のようになります。

List 23.4　accesslog_parse2.rb

```
require "access_log"        # access_log.rbを読み込む

count = 0
File.open(ARGV[0]) do |io|
  AccessLog.each_entry(io) do |entry|
    p entry.to_a            # エントリを表示する
    count += 1
  end
end
puts count
```

23.6 コマンドの実行

それでは，AccessLogモジュールを使ってアクセスログ解析を行うコマンドを作っていきます。内容としては，次の解析を考えます。

- 時間帯別のアクセス数を集計する
- URL別にアクセス数を集計する
- エラーになったアクセスを表示する

これらの処理を1つずつ記述したコマンドを3つ作ってもよいのですが，スクリプトが多くなると書き換えたりするのが面倒になるので，スクリプトは1つにまとめておいてコマンドラインで必要な処理を呼び分けられると便利です。今回はRakeを利用して，コマンドラインの処理を自分で書かかなくても呼び分けられるようにします。

23.6.1 Rakeとは

RakeはRubyで記述されたタスクを実行するためのプログラムです。タスクというと難しい感じがしますが，「繰り返し実行する必要のある定形の処理」という程度の意味で，普通のRubyスクリプトとして記述することができます。

column

Structクラス

List 23.3では，AccessLog::Entryクラスの作成にStructクラスを使っています。Structクラスは，データを格納するだけのクラスを作成する場合に便利です。

```
Struct.new(クラス名, メンバ名, …)
Struct.new(メンバ名, …)
```

のいずれかの形式で呼び出すと，メンバ名と同名のアクセスメソッドを持ったクラスを返します。クラス名はStringオブジェクトで，メンバ名はシンボルで指定します。Structクラスのインスタンスを返すのではない点に注意してください。

```
Foo = Struct.new(:a, :b, :c) # Fooクラスを生成
foo = Foo.new(1, 2, 3)       # new の引数は省略可
p foo.a                      #=> 1
p foo.b                      #=> 2
p foo.c                      #=> 3
foo.a = 10
p foo.a                      #=> 10
p foo.to_a      # 配列化     #=> [10, 2, 3]
```


Rakeのタスクはrakeコマンドによって実行されます。rakeコマンドがインストールされていない場合は，gemパッケージをインストールするのが簡単です。

実行例

```
> gem install rake
Successfully installed rake-0.8.7
1 gem installed
installing ri documentation for rake-0.8.7
installing RDoc documentation for rake-0.8.7
```

WindowsのActiveScriptRubyで「gem install」を実行するときは，管理者権限のコンソール「Ruby Console (Administrator)」を利用してください。

23.6.2 Rakefile

RakefileはRakeが実行するタスクの内容を記述するためのファイルです。rakeコマンドはカレントディレクトリにあるRakefileを読み込んで，その中に定義されたタスクを実行します。カレントディレクトリにRakefileがなければ，親ディレクトリにさかのぼってRakefileを探して読み込みます。Rakefileに複数のタスクを記述しておいて，コマンドライン引数によって呼び分けることができます。

簡単な例として，メッセージを表示するだけのタスクを作ってみます。

```
task :foo do
  puts "FOO!"
end

task :bar do
  puts "BAR!"
end
```

この例では2つのタスクfooとbarを定義しています。タスクを実行するには，以下のようにします。

実行例

```
> rake foo          ← fooタスクの実行
(in /home/ruby/ch23)
FOO!
> rake bar          ← barタスクの実行
(in /home/ruby/ch23)
BAR!
> rake foo bar      ← 両方のタスクを続けて実行
(in /home/ruby/ch23)
FOO!
BAR!
```

また，barタスクを実行するときは事前に:fooを実行したい，というふうに手順がある場合には以下のように，「**task タスク名 => [必要なタスク]**」という形式で記述します。必要なタスクの部分は配列です。複数のタスク名をならべて書けば，その順番に実行されます。

```
task :foo do
  puts "FOO!"
end

task :bar => [:foo] do
  puts "BAR!"
end
```

こうすると，barタスクを実行する前にfooタスクが実行されます。

実行例

```
> rake bar
(in /home/ruby/ch23)
FOO!
BAR!
```

rakeコマンドに引数を指定しなかった場合には，defaultというタスクが実行されます。引数を指定せずにタスクを実行する場合は，fooやbarと同様にdefaultというタスクを定義します。あるいは次のように，別のタスク名

を指定して，空のタスクを定義します。

```
task :default => [:foo]
```

タスクの中で利用するパラメータを変更するには環境変数を使います。コマンドラインからパラメータを与える場合は「**rake 変数名=値**」の形式で指定します。変数はいくつ指定しても構いません。指定された値は環境変数と一緒に定数ENVから取得することができます。

以下のようにタスクを定義して，

```
task :hello do
  name = ENV["NAME"] || "Ruby"
  puts "Hello, #{name}!"
end
```

以下のように実行することができます。

実行例

```
> rake hello NAME="Rake"
(in /home/ruby/ch23)
Hello, Rake!
```

Rakeには複雑なタスクを記述するために便利な機能が用意されていますが，ここまでの知識で当面の目的には間に合うので，Rakefileの説明はいったん切り上げます。

次の節からは，話をアクセスログの解析に戻してタスクを作っていきます。

メモ

Rakeはもともとはビルドツールとして開発されました（Rakeという名前はUnixのビルドツールであるmakeにちなんでいます）。「ビルド」というのは，複数のファイルに分割して記述されたプログラムや設定ファイルなどをまとめて1つのプログラムにしたり，インストール用のパッケージを作ったりする作業のことです。

RubyGemsのパッケージ形式であるgemファイルを作成する場合にもRakeが用いられます。

23.7 アクセスログ解析タスク

これから作成する解析の内容をもう一度挙げておきます。

- 時間帯別のアクセス数を集計する
- URL別にアクセス数を集計する
- エラーになったアクセスを表示する

いずれの処理もアクセスログ全体を読み込む必要があるので，まず，共通して利用できるログファイルを読み込むためのタスクを作成して，それから個々の解析のためのタスクを作っていくことにしましょう。

23.7.1 アクセスログの読み込み

アクセスログの読み込みには，この章のはじめに作ったAccessLogモジュール（List 23.3）を使います。Rakefileはrakeコマンドによって読み込まれますが，それ自体はRubyスクリプトなので，ライブラリを読み込むには通常どおりrequireメソッドを使います。

読み込みを行うタスクの名前はloadとしてタスクを作成します。

```
require "access_log"

entries = []
task :load do
  logfile = ENV["LOGFILE"] || "access.log" # ログファイル名
  puts "loading #{logfile}."               # メッセージを表示する
  File.open(logfile) do |log|              # ログファイルを開いて
    AccessLog.each_entry(log) do |entry|   # すべてのエントリを
                                           # 読み込む
      entries << entry
    end
  end
end
```


解析の対象となるアクセスログのファイル名は環境変数LOGFILEが設定されていればそれを使います。なければカレントディレクトリの"access.log"となります。ファイルを開いたら，List 23.4と同じ要領で，AccessLog.each_entryメソッドによって，解析されたエントリを取り出します。すべてのエントリは変数entriesに初期化された配列に追加されます。

エントリをすべて読み込むため，エントリが数百万件にものぼるような場合はメモリに収まりません。そのような場合は，解析した結果を配列ではなくデータベースに保存して，メモリの制限を回避しつつ高速にアクセスできるようにするといった見直しが必要となりますが，今回紹介するサンプルでは気にしないことにします。

23.7.2 時間帯別のアクセス数を集計する

0時から23時の1時間ごとのアクセス数の分布を調べるタスクを作ります。タスク名はtimeとして，集計の処理の前にタスクloadが実行されるようにします。

エントリに記録されたアクセス時刻は，AccessLog::Entryオブジェクトのtimeメソッドに格納されています。タスクloadで読み込んだ配列entriesの各要素から，アクセス時刻を取り出して，何時台の情報かを調べます。アクセス時刻は「14/Jan/2010:02:16:10 +0900」という形式なので，「/」か「:」か空白で文字列を分割すると，4番目（インデックスは3）の値から「時」を取得することができます。

```
task :time => [:load] do
  hour_count = Hash.new(0)                  # 集計用のハッシュ
  entries.each do |entry|                   # エントリを順に処理する
    times = entry.time.split(/[:\/ ]/) # 時刻を分割する
    hour_count[times[3]] += 1               # 「時」のカウントを追加する
  end
  hours = hour_count.keys.sort              # 「時」の一覧を取得する
  hours.each do |h|                         # 集計結果を順に表示する
    printf("%s: %s\n", h, "#" * (hour_count[h]/3))
  end
end
```

このタスクを実行すると，「時間帯」と「時間帯に対するアクセス数」を表示します。アクセスは単に数字を表示するのではなく，結果を棒グラフのように見せるために「#」をアクセス数に応じて表示します。

実行例は次のようになります。

実行例

```
> rake time LOGFILE=sample-access.log
(in /home/ruby/ch23)
loading sample-access.log.
00: ##############
01: ##################################
02: ##################################
03: ######################
04: ##################
05: ################
06: ############################
07: ##################
08: ######################
09: ####################
10: ##################
11: ##################
12: ################
13: ##########
14: ###########
15: ####################
16: ############
17: ########################
18: ####################
19: ########################
20: ######################
21: ############################
22: ################
23: ##############
```


23.7.3　URL別にアクセス数を集計する

時間帯別のアクセス集計を行う例では，時刻をキーにしてハッシュにアクセス数を集計しました。今度はURLをキーにしてハッシュにアクセス数を集計していきます。

```
task :url => [:load] do
  url_count = Hash.new(0)                    # 集計用のハッシュ
  entries.each do |entry|                    # エントリを順に処理する
    url_count[entry.url] += 1                # URLのカウントを追加する
  end
  ranking = url_count.sort_by{|url, count| -count }
        # アクセス数の降順になるようにハッシュの要素をソート
  ranking.each do |url, count|               # 集計結果を順に表示する
    printf("%d: %p\n", count, url)
  end
end
```

このタスクは，集計結果をソートして，アクセスの多い順にアクセス数とURLを表示します。基本的な構造は先ほどのtimeタスクと同じく，いったん集計したデータを順に出力しています。

注　ひとつ注意したいのは，出力の形式です。URLを出力する際に，printfメソッドに%pフォーマットを利用しています。

```
printf("%d: %p\n", count, url)
```

アクセスログファイルにはクライアントが送信したメッセージがそのまま書き込まれているため，コンソールで表示することが期待されてない文字が含まれているかもしれません。たとえば，表示が崩れたり，コンソールが制御不能になったりするといった予期しない結果が起こり得ます。
%pフォーマットを使うと，表示できない特殊な文字はpメソッドの出力のようにエスケープされます。

実行例は次のようになります。

実行例

```
> rake url LOGFILE=sample-access.log
(in /home/ruby/ch23)
loading sample-access.log.
loading sample-access.log.
406: "/r/"
301: "/r/index.html"
110: "/robots.txt"
20: "/r/200906c.html"
18: "/r/200907c.html"
18: "/r/201002a.html"
17: "/r/200907a.html"
16: "/r/topics_www.html"
16: "/rbsax/doc/"
15: "/r/200911a.html"
    ⋮
```

23.7.4　エラーになったアクセスを表示する

3つ目は，エラーになったリクエストの一覧を出力するタスクを紹介しましょう。

アクセスログのステータス項目によってリクエストが成功したか，エラーになったかを知ることができます。クライアントからのリクエストが，正常に処理された場合は，ステータスは「200」となります。

エラーの場合は，ステータスにより，もう少し情報を読み取ることができます。たとえば，クライアントが取得しようとしたファイルがなければ「404」，ファイルに適切なアクセス権がなければ「403」，CGIなどのプログラムがエラーになったら「500」などです。

エラーが起こった場合は，ステータスが4*xx*または5*xx*になります。


```
task :error => :load do
  entries.each do |entry|       # エントリを順に処理する
    if /^[45]/ =~ entry.status # ステータスが4xxか5xxなら表示する
      printf("%p %p %p\n", entry.time, entry.status, entry.url)
    end
  end
end
```

単純にステータスか「4」か「5」で始まる場合は，時刻とステータスとURLをコンソールに表示します。

実行例は次のようになります。

実行例

```
> rake error LOGFILE=sample-access.log
(in /home/ruby/ch23)
loading sample-access.log.
loading sample-access.log.
"11/Feb/2010:02:23:57 -0500" "404" "/robots.txt"
"11/Feb/2010:03:13:01 -0500" "404" "/robots.txt"
"11/Feb/2010:03:42:23 -0500" "404" "/robots.txt"
"11/Feb/2010:03:59:33 -0500" "404" "/r/201001b.html"
"11/Feb/2010:03:59:38 -0500" "404" "/r/201001c.html"
"11/Feb/2010:04:14:46 -0500" "404" "/robots.txt"
"11/Feb/2010:05:46:08 -0500" "404" "/r/200910b.html"
    ⋮
```

23.7.5　タスクの説明を追加する

完全なRakefile（List 23.5）を掲載しておきましょう。デフォルトのタスクは，time，url，errorをすべて実行することにします。

List 23.5 Rakefile

```
require "access_log"

entries = []
task :load do
  logfile = ENV["LOGFILE"] || "access.log" # ログファイル名
  puts "loading #{logfile}."               # メッセージを表示する
  File.open(logfile) do |log|              # ログファイルを開いて
    AccessLog.each_entry(log) do |entry|
                                           # すべてのエントリを読み込む
      entries << entry
    end
  end
end

desc "時間帯別のアクセス数を集計する"
task :time => [:load] do
  hour_count = Hash.new(0)                 # 集計用のハッシュ
  entries.each do |entry|                  # エントリを順に処理する
    times = entry.time.split(/[:\/ ]/)     # 時刻を分割する
    hour_count[times[3]] += 1              # 「時」のカウントを追加する
  end
  hours = hour_count.keys.sort             # 「時」の一覧を取得する
  hours.each do |h|                        # 集計結果を順に表示する
    printf("%s: %s\n", h, "#" * (hour_count[h]/3))
  end
end

desc "URL別にアクセス数を集計する"
task :url => [:load] do
  url_count = Hash.new(0)                  # 集計用のハッシュ
  entries.each do |entry|                  # エントリを順に処理する
    url_count[entry.url] += 1              # URLのカウントを追加する
  end

  ranking = url_count.sort_by{|url, count| -count }
        # アクセス数の降順になるようにハッシュの要素をソート
```



```
  ranking.each do |url, count|             # 集計結果を順に表示する
    printf("%d: %p\n", count, url)
  end
end

desc "エラーになったアクセスを表示する"
task :error => [:load] do
  entries.each do |entry|          # エントリを順に処理する
    if /^[45]/ =~ entry.status # ステータスが4xxか5xxなら表示する
      printf("%p %p %p\n", entry.time, entry.status, entry.url)
    end
  end
end

task :default => [:time, :url, :error]
```

各タスクを定義する直前に「**desc "タスクの説明"**」を書いています。こうしておくと，「rake --tasks」または「rake -T」でRakefileに含まれる実行可能なタスクの一覧を表示することができます。

実行例

```
> rake -T
(in /home/ruby/ch23)
rake error  # エラーになったアクセスを表示する
rake time   # 時間帯別のアクセス数を集計する
rake url    # URL別にアクセス数を集計する
```

Rakeを使うと，自分でコマンドをゼロから作成するのに比べて，ツールらしいスクリプトをかなり手軽に記述できます。紹介しきれませんでしたが，より高度な機能を知るにはRakeのパッケージに添付のドキュメントやサンプルコードを参考にしてください。

メモ

開発者による情報を以下のURLから得ることができます（ともに英語）。

- http://rake.rubyforge.org/
- http://github.com/jimweirich/rake

この原稿を書いている時点では日本語による公式な情報はありませんが，Web上ではRakeに関するたくさんの記事を読むことができます。