Mails löschen mit Ruby

CHRISTOPH LANGE

Die Mails der letzten Tage lasse ich aus Sicherheitsgründen immer noch auf dem POP3-Server liegen, damit ich übers Web von außerhalb darauf zugreifen kann. Nur, wie löscht man solche Mails, ohne das Web-Interface (oft langsam), direkt POP3 über Telnet (unbequem) oder »fetchmail --flush« (wenig flexibel) zu verwenden? Hier hilft ein kleines Ruby-Programm.

Ruby ist eine moderne Skriptsprache, die den Anspruch stellt, Perl und Python zu ersetzen. Ruby verbindet die besten Elemente dieser Sprachen mit intelligenten Konzepten vieler anderer Sprachen. Für unsere Aufgabe ist Ruby wegen der flexiblen und einfachen Syntax sowie der zahlreichen von der Klassenbibliothek unterstützten Datenstrukturen und Netzwerk-Protokolle interessant.

Arbeitsweise

PopDel ist ein Kommandozeilen-Programm, dem die Anzahl der zu löschen oder zu behaltenden Mails sowie die Zugangsdaten für den Server übergeben werden. Wenn Benutzername und Paßwort nicht gegeben sind, versucht das Programm, sie aus »~/.fetchmailrc« zu lesen. Das liegt näher als eine eigene Konfigurationsdatei, weil der typische PopDel-Anwender Mails auch ohne Klick&Bunt-Oberfläche lesen kann und sie deshalb wahrscheinlich sowieso mit *fetchmail* abholt. Die Funktion *help* zeigt die Online-Hilfe, die das bei einem falschen Aufruf des Programms oder gesetzter Option *-h* beziehungsweise *--help* noch einmal erklärt:

```
def help
  print <<EOF
Aufruf:
popdel [-hv] [+-]n SERVER [USER PASSWD]
Löscht die ersten +n Mails oder alle bis
auf -n Mails vom SERVER (POP3).
Sind USER und PASSWORD nicht gegeben,
versucht popdel, sie aus der Datei
\`~/.fetchmailrc\' zu lesen

  -v, --verbose Ausführliche Hinweise
  -h, --help    Diese Hilfe
EOF
end
```

Strings von mehreren Zeilen lassen sich am besten mit einem Here-Dokument bilden. Das Dokument beginnt mit *<<ENDE* und endet bei einer Zeile, die genau aus dem Text »ENDE« besteht. Wenn mit dem Here-Dokument noch etwas gemacht werden soll, steht das in der Zeile davor. So kann man ein Here-Dokument durch Einrückung übersichtlicher gestalten und vor der Ausgabe wieder ausrücken. *\t* steht dabei für ein Tabulatorzeichen:

```
print <<EOF.gsub(/^\t/, '')
        Ein mit Tabulatoren eingerücktes
        Here-Dokument, das normal
        ausgegeben wird.
EOF

# Ausgabe:

Ein mit Tabulatoren eingerücktes
Here-Dokument, das normal
ausgegeben wird.
```

Nun einige Beispiele für den Aufruf des Programms:

```
$ popdel -v -1 pop.my.server
Angemeldet als user@pop.my.server
3 Mails auf dem Server
Lösche Mail 1
Lösche Mail 2
$ popdel -v +1 pop.server gast k6X9vdbx
Angemeldet als gast@pop.server
9 Mails auf dem Server
Lösche Mail 1
$ popdel -5 pop.illegal.server hack key
Anmeldung als User hack nicht möglich
$ popdel +10 pop.not.conf.server
Benutzername und Passwort konnten nicht
bestimmt werden
$ popdel -2 pop.not.ready.server
Kann keine Verbindung zu Server
pop.not.ready.server herstellen
```

Kommandozeilen-Parameter verarbeiten

Das Programm versteht zwei Kommandozeilen-Optionen; darauf folgen als weitere Argumente mindestens die Anzahl der zu löschenden oder zu behaltenden Mails und der Name des Servers. Die Verarbeitung von Optionen automatisiert das Modul *getoptlong*, eine Nachbildung der C-Bibliotheksfunktion *getopt_long*.

```
require 'getoptlong'

parser = GetoptLong.new
parser.quiet = true
parser.set_options(['-h', '--help',
        GetoptLong::NO_ARGUMENT],
        ['-v', '--verbose',
        GetoptLong::NO_ARGUMENT])
```

Die Eigenschaft *quiet* wird gesetzt, damit *getoptlib* bei fehlerhaften Optionen keine eigenen Fehlermeldungen ausgibt; das machen wir selbst. *set_options* erhält beliebig viele Optionen als Argumente, für jede ein Array. Dessen erste Elemente sind beliebig viele lange und kurze Namen für die Option; das letzte legt fest, ob die Option ein Argument hat. Möglich wären hier auch *GetoptLong::REQUIRE_ARGUMENT* und *GetoptLong::OPTIONAL_ARGUMENT*.
Der einfachste Weg zur Verarbeitung der Optionen ist der Iterator *each_option*, dessen Code-Block zu jeder gefundenen Option den Namen und das eventuell vorhandene Argument erhält:

```
begin
  parser.each_option do |name, arg|
    print "Option #{name}"
    print " mit Argument #{arg}" \
          if arg
    print "\n"
  end
rescue
  puts 'Optionen sind fehlerhaft'
  exit 1
end
```

Der Name einer Option ist immer das erste Element, das im Array zur jeweiligen Option an *set_options* übergeben wurde.
Die Verarbeitung der Optionen kann verschiedene Exceptions (objektorientierte Laufzeitfehler) auslösen, die mit einem *begin-rescue*-Block abgefangen werden sollten (Tabelle 1). PopDel akzeptiert aber nach den Optionen auch ein Argument der Form *-[0-9]+* (regulärer Ausdruck). Wir müssen verhindern, daß dieses Argument als ungültige Option erkannt wird und deshalb zugunsten der manuellen Methode *get_option* auf den Iterator verzichten:

```
ERR_SYNTAX = 1

begin
  loop do
    break
       if ARGV[0] =~ /^-[0-9]+$/
        name, arg = parser.get_option
    case name
    when '-h'
      opt_help = true
    when '-v'
      opt_verbose = true
    when nil
      break
    end
  end
rescue
  help
  exit ERR_SYNTAX
end
```

Wenn eine Zahl als Option erkannt wird, brechen wir die Verarbeitung der Optionen ab. Die Zeile *name, arg = parser.get_option* ist legitim, weil *get_option* ein Array mit zwei Elementen zurückliefert. Sind keine Optionen mehr vorhanden, liefert *get_option* als Name *nil* zurück. Dieser Fall ist ausgeschlossen, wenn Sie den Iterator *each_option* verwenden.
Im Gegensatz zu anderen *getopt*-Implementationen bleiben in Ruby nach dem Verarbeiten der Optionen im Array *ARGV* nur noch die Kommandozeilen-Parameter nach den Optionen zurück. Wenn das keine mehr sind, liegt ein Fehler vor. In diesem Fall und wenn die Option *-h* gesetzt ist, zeigen wir den Hilfetext an und beenden das Programm:

```
if opt_help or ARGV.size == 0
  help
  exit opt_help ? 0 : ERR_SYNTAX
end
```

Nach der Optionsverarbeitung muß *ARGV* noch zwei (Anzahl, Server) oder vier (zusätzlich Benutzername und Paßwort) Elemente enthalten. Alles andere ist ein Fehler. Da der Fehler an mehreren Stellen auftreten kann, machen wir den Code mit einer eigenen Exception-Klasse übersichtlicher.

```
class CommandLineError < \
      StandardError
end

begin
  if ARGV.size > 0 and
     ARGV[0] =~ /^[+-][0-9]+$/
    numdel = ARGV[0].to_i -
             (ARGV[0][0] == ?- ? 1 : 0)
    ARGV.shift
  else
    raise CommandLineError.new
  end

  if ARGV.size > 0
    server = ARGV[0]
    ARGV.shift
  else
    raise CommandLineError.new
  end

  if ARGV.size > 0
    if ARGV.size == 2
      username, password = ARGV[0, 2]
    else
      raise CommandLineError.new
    end
  end
rescue CommandLineError
  help
  exit ERR_SYNTAX
end
```

Wenn das erste Argument eine Zahl

Exception	Bedeutung
GetoptLong::AmbigousOption	Eine lange Option wurde nicht ganz ausgeschrieben und ist dadurch mehrdeutig.
GetoptLong::InvalidOption	Eine ungültige Option wurde angegeben.
GetoptLong::MissingArgument	Einer Option fehlt das erforderliche Argument.
GetoptLong::NeedlessArgument	Zu einer Option ohne Argument wurde eins angegeben.

Tabelle 1: getoptlong-Exceptions

mit Vorzeichen ist, wird sie in einen *Integer* umgewandelt. Wichtig ist hier, daß *+0* und *-0* nicht verwechselt werden, denn *+0* löscht keine Mails (die ersten Null Mails), *-0* aber alle (alle bis auf Null). Deshalb ziehen wir von einer negativen Zahl zusätzlich Eins ab. Der Ausdruck *ARGV[0][0] == ?- ? 1 : 0* gibt *1* zurück, wenn das erste Zeichen (Index 0, auch bei Strings) des ersten noch verbliebenen Kommandozeilen-Parameters ein Minuszeichen (Konstante *?-*) ist, sonst *0*. Der nächste Parameter ist der Name des Servers, optional folgen Benutzername und Paßwort. *ARGV[0, 2]* gibt ein Array mit den ersten beiden Elementen (von Index 0 bis Index 2, nicht einschließlich) aus *ARGV* zurück, was sich wieder an zwei Variablen zuweisen läßt. Der folgende *rescue*-Block fängt nur unsere Exception ab.

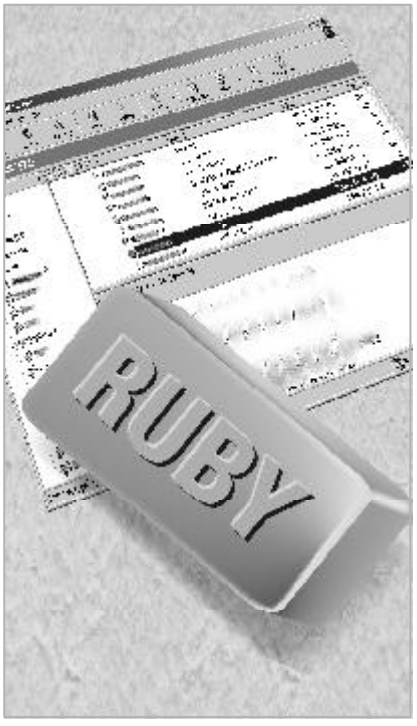
Konfigurationsdatei einlesen

Wenn Benutzername und Paßwort nicht auf der Kommandozeile angegeben wurden, müssen sie in »~/.fetchmailrc« gesucht werden. Die Syntax dieser Datei ist sehr frei; wir verlassen uns darauf, daß sie einigermaßen konventionell gehalten ist. Wir suchen eine Zeile mit der Anweisung »poll SERVER« und in dieser Zeile die in doppelte Anführungszeichen eingeschlossenen Wörter nach »user« beziehungsweise »username« sowie »pass« beziehungsweise »password«. Wenn das nicht erfolgreich war, beendet sich das Programm mit einer Fehlermeldung.

```
unless username
  confline = File.new("#{ENV['HOME']
    }/.fetchmailrc").readlines.
    grep(/poll #{Regexp.quote(server)}/)
    [0]
  username = (confline =~ /user(?:name)?
              \"([^\"]+)\"/ && $1)
  password = (confline =~ /pass(?:word)?
              \"([^\"]+)\"/ && $1)


  unless username and password
    STDERR.print 'Benutzername'
                 unless username
    STDERR.print ' und ' unless
                 username or password
    STDERR.print 'Passwort' unless
                 password
    STDERR.print ' konnte'
    STDERR.print 'n' unless username
                 or password
    STDERR.puts ' nicht bestimmt werden'
    exit ERR_CONFIG
  end
end
```

Zur Bestimmung der richtigen Zeile genügt in Ruby eine einzige Anweisung. *File.new* öffnet die angegebene Datei, wobei *ENV['HOME']* im Dateinamen für den Wert der Umgebungsvariablen *HOME*, also das Home-Verzeichnis steht. *readlines* liefert alle Zeilen der Datei als Array zurück (in Perl als »slurp mode« bekannt). *grep* (Achtung, Zeilenumbruch innerhalb der Anweisung!) gibt ein Array aller Zeilen mit der Anweisung »poll SERVER« zurück, und davon nehmen wir auf gut Glück die erste. *Regexp.quote* quotet alle Sonderzeichen in einem String, damit er als konstanter Bestandteil eines regulären Ausdrucks verwendet werden kann. In einem Server-Namen wird dadurch beispielsweise ».« als »\.« dargestellt und verliert damit seine besondere Bedeutung.



Code-Wettbewerb

Es gibt einen »Obfuscated C Contest« und dasselbe für Perl. Ziel dieser Wettbewerbe ist es, möglichst unleserlichen, aber dennoch effektiven Code zu schreiben.

Der Benutzername ist ein in doppelte Anführungszeichen eingeschlossenes Wort, das nach »user« oder »username« steht. *(?:name)?* steht für »name« oder nichts; *?:* verhindert dabei, daß der zum Klammerinhalt passende Text in einer der Variablen *$1*, *$2* und so weiter gespeichert wird. Die Klammer dient hier nicht zum Finden eines Texts, sondern nur zum Gruppieren für den Wiederholungsoperator *?*. Wohl aber gespeichert wird das in den Anführungszeichen gefundene Wort. Wenn die Mustersuche erfolgreich war, wird *$1* zurückgegeben, sonst *nil*.
Wenn *username* oder *password* nach dem Einlesen der Konfigurationsdatei immer noch *nil* sind, liegt ein Fehler vor. Bestimmte Ausschnitte der Fehlermeldung werden nur dann ausgegeben, wenn eine Bedingung zutrifft. Eine Bedingung nach einer einzigen Anweisung kann einfach wie in Perl in derselben Zeile nachgestellt werden.

POP3-Zugriff

Der bisherige Code diente fast nur der Benutzerfreundlichkeit; die eigentliche Logik macht den kleinsten Teil des Programms aus. Verwendet wird die Klasse *Net::POP3* aus dem Modul *net/pop*. Die Methode *start* mit den Parametern Servername, Port (*nil* nimmt den Standardwert 110), Benutzername und Paßwort ist ein Iterator, der als Variable ein *POP3*-Objekt zur Arbeit innerhalb der POP3-Sitzung zurückgibt. Das ist nur der Übersichtlichkeit halber so gelöst, weil der Iterator ja nicht über eine Menge oder Liste iteriert.

```
require 'net/pop'

Net::POP3.start(server, nil,
```

```
    username, password) do |pop|
  # ...
end
```

Die Methode *POP3#mails* gibt dann ein Array von *POPMail*-Objekten zurück. Viel einfacher wäre es noch, alle Mails auf einmal vom Server zu löschen, weil es schon in der Klasse *POP3* einen Iterator über alle Mails gibt:

```
Net::POP3.foreach(server, nil,
    username, password) do |mail|
  mail.delete
end
```

Wir aber müssen die Anzahl der Mails auf dem Server bestimmen (*pop.mails.size*) und dann nur die gewünschten Mails löschen. Zu löschen ist ein Bereich von Mail Nr. a bis Mail Nr. b, und da es in Ruby keine arithmetische *for*-Zählschleife wie in C und Perl gibt, erzeugen wir ein *Range*-Objekt, das den Bereich der zu löschenden Mails enthält.

```
count = pop.mails.size
if opt_verbose
  puts \
  "Angemeldet als #{username}@#{server}"
  puts "#{count} Mails auf dem Server"
end

if numdel != 0
  delrange = if numdel > 0
    0 .. numdel - 1
    else
      0 .. count + numdel
    end
  delrange.each do |i|
    puts "Lösche Mail #{i + 1}" \
        if opt_verbose
    pop.mails[i].delete
  end
end
```

Zu löschen ist nur etwas, wenn die Variable *numdel* nicht als Resultat des Kommandozeilen-Parameters *+0* den Wert *0* hat. In diesem Fall sind bei positivem *numdel* (Parameter *+n*) die Mails mit den Array-Indizes *0* bis *numdel - 1* zu löschen, bei negativem *numdel* die Mails mit den Indizes *0* bis *count + numdel*. Bei zehn Mails löscht beispielsweise die Option *+3* die ersten drei Mails (Indizes 0 bis 2), die Option *-3* (die in der Variablen *numdel* als *-4* gespeichert wird) aber alles bis auf die letzten drei (Indizes 0 bis 6, übrig bleiben 7 bis 9). Beachten Sie, daß in Ruby jeder Block einen Rückgabewert hat. Daher können wir elegant mit *if* etwas schreiben, was in anderen Sprachen nur mit dem kryptischen *?:*-Operator möglich wäre. So wirkt es am übersichtlichsten; am kürzesten wäre die Schreibweise *delrange = 0..(numdel + (numdel > 0 ? -1 : count))*. Gibt es eigentlich schon einen »Obfuscated Ruby Contest«?
Das Iterieren über den Bereich könnten wir uns schenken, aber wir brauchen für die Ausgabe des Hinweistextes einen Schleifenzähler. Sonst könnten wir das *Range*-Objekt direkt auf das Array anwenden:

```
pop.mails[delrange].each do |mail|
  mail.delete
end
```

Um die ganze POP3-Sitzung fassen wir einen *begin-rescue*-Block. Mögliche Exceptions sind ein *SocketError* oder *Errno::ECONNREFUSED* (die Exceptions *Errno::...* sind die Ruby-Pendants zu den unter *man errno* beschriebenen Laufzeitfehlern, die bei Unix-Systemaufrufen auftreten), wenn der Server kein POP3 kann, oder *Net::ProtoAuthError*, wenn die Zugangsdaten falsch sind.

```
begin
  Net::POP3.start(server, nil,
      username, password) do |pop|
    # ...
  end
  rescue SocketError,
      Errno::ECONNREFUSED
  STDERR.puts "Kann keine Verbindung " \
```


```
      "zu Server #{server} herstellen"
  exit ERR_CONNECT
rescue Net::ProtoAuthError
  STDERR.puts "Anmeldung als User " \
      "#{username} nicht möglich"
  exit ERR_LOGIN
end
```

Fazit

Die eigentliche Logik dieses Programms umfaßt kaum mehr als zehn Zeilen. Das geht in kaum einer Sprache so einfach und doch verständlich. Viele Aufgaben der Internet-Programmierung sind heute schon mit Ruby lösbar. Auf andere Lösungen muß man noch warten, weil die Klassenbibliothek noch lange nicht so umfangreich ist wie die von Perl.
Auf der freeX-CD im Verzeichnis »freeX/popdel« finden Sie das Programm Ruby in der aktuellen Version 1.6.4 und die Dokumentation zum Modul *getoptlong*.

Literatur

[1] Andrew Hunt, David Thomas: Programming Ruby – The Pragmatic Programmer's Guide. Addison-Wesley 2000. ISBN 0201710897. 592 Seiten. http://www.rubycentral.com/book/index.html

[2] Christoph Lange: Perl++ – Die Programmiersprache Ruby. Toolbox 4'2001, Seiten 73-81. Toolbox-Verlag 2001.