Perl 5 e programmazione object-oriented: un'applicazione reale

Il Perl 5 ha introdotto rispetto alla versione precedente la possibilità di utilizzare oggetti e strutture dati complesse; queste caratteristiche hanno reso il linguaggio ancora più potente e versatile. In questo articolo vedremo come vengono utilizzati gli oggetti in Perl su una applicazione reale, un server realizzato per il CRS4 (Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna)

di Marco Marongiu

Fino alla versione 4 la potenza del Perl era concentrata nelle espressioni regolari e in tutte quelle funzioni che consentivano di manipolare stringhe meglio di ogni altro linguaggio. Ma un grave punto debole era costituito dalla impossibilità di creare strutture dati complesse, se non ricorrendo a dei veri e propri trucchi.

Il Perl 5 ha portato due importanti innovazioni riguardanti le strutture dati e la sintassi:

- *sono state introdotte le* reference, *che consentono di annidare le strutture dati con un livello arbitrario di complessità;*
- *si può scegliere se programmare in modo procedurale o, finalmente, a oggetti.*

La programmazione ad oggetti

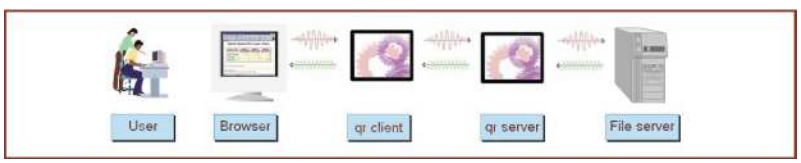
A grandi linee (e col rischio di scandalizzare i puristi!) potremmo definire gli oggetti come aggregati di dati e di funzioni che li manipolano; tali funzioni prendono il nome di *metodi*; i dati interni dell'oggetto prendono il nome di *attributi*, e non possono essere modificati se non attraverso i metodi dell'oggetto stesso. Una classe non è altro che un "prototipo" di oggetti, un modello su cui gli oggetti vengono effettivamente costruiti.
Per fare un esempio vicino del mondo reale, un orologio è un oggetto fatto in un certo modo che serve a misurare il tempo, e un qualsiasi modello di "Rolex" è un oggetto della classe orologio.
Il fatto di non poter manipolare direttamente i dati ma di dover ricorrere a delle funzioni che, in un certo senso, lo facciano per noi ha delle implicazioni importanti. La strutturazione dei dati all'interno dell'oggetto e l'implementazione stessa dei metodi rimangono mascherate all'utilizzatore.
Così se il nostro oggetto "orologio" ha un metodo *imposta_data* che accetta un qualunque numero da 1 a 31, per reimpostare la data al primo del mese sarà sufficiente solo invocare quel metodo sull'oggetto, passandogli il parametro 1, senza necessità di sapere come verrà eseguita l'azione richiesta.
Quindi, se scrivo una applicazione che legge da una certa sorgente il giorno del mese e lo reimposta sull'orologio, il mio programma continuerà a funzionare senza modifiche anche se chi ha implementato l'oggetto decide di modificarne il codice (sempre che non vengano cambiati il nome al metodo che imposta la data o il modo di passargli i parametri).
Ciò premesso, entriamo nel dettaglio dell'applicazione e, soprattutto, di come in essa sono utilizzati gli oggetti.

Il quota report server

Nell'azienda in cui lavoro la maggior parte dei dati degli utenti sono attestati su un file server; la quantità dei dati che un utente può scrivere è limitata attraverso quote.
Il concetto di quota su un file server è grossomodo lo stesso di UNIX, che è stato descritto molto bene su questa rivista [1]. Dall'analisi del problema è apparso subito chiaro che per dare la possibilità agli utenti di monitorare la loro quota, senza dare accesso al file server, era necessario in primo luogo avere un server che pubblicasse i report delle quote, e in secondo luogo un'applicazione (ad esempio una CGI) che mostrasse ad ogni utente solo i dati che lo riguardavano.
Uno schema di funzionamento è riportato in **Figura 1**.
Il quota report server doveva essere fortemente modulare tale che, per modificarne il modo di eseguire l'autenticazione o il numero e la sintassi di comandi accettati, fossero sufficienti solo modifiche di dettaglio. Una implementazione object-oriented, che utilizzasse oggetti per

FIGURA 1

Un esempio di utilizzo del quota report server attraverso un browser web: collegandosi ad una pagina si attiva un client che interroga il quota report server; a sua volta il quota report server richiede le statistiche al file server, le elabora e le restituisce al client; il client formatta i dati in HTML per consentire la visualizzazione attraverso il browser (Grafica: I.Marcialis)



i compiti specifici, sembrava la scelta migliore. Decidemmo quindi di seguire questa strada.
Il listato dello script (senza il codice delle classi) è riportato nel **Listato 1**. L'analisi che faremo ha due obbiettivi: il primo è mostrare come gli oggetti vengono utilizzati in Perl, il secondo è mostrare come una attenta analisi dei problemi e una soluzione object-oriented azzeccata permettano di costruire delle applicazioni versatili e più facilmente manutenibili.

Moduli, classi e librerie utilizzati

Nello script sono utilizzate

- librerie che fanno parte della distribuzione ufficiale del Perl, come *POSIX*
- moduli opzionali come *IO::Socket*
- moduli creati appositamente per questa applicazione (*QRServer::Auth::Fixed*, *QRServer::Parser::Base*, *QRServer::Tokenizer*, *QRServer::Reporter*).

Vediamo brevemente cosa sono e a cosa servono.

- *POSIX*: viene qui utilizzato solo per importare il simbolo *WNOHANG*, che viene successivamente utilizzato nei signal handler dello script (**Riquadro 1**);
- *IO::Socket*: nella sua variante *INET* è una classe di oggetti che rappresentano socket, utilizzati dallo script per creare le connessioni di rete;
- *QRServer::Auth::Fixed*: è l'oggetto che si occupa dell'autenticazione del client; è detto "Fixed" perché riconosce un unico identificativo con una sola password;
- *QRServer::Parser::Base*: è la classe degli oggetti che effettuano il parsing delle linee di comando trasmesse dal client al server;
- *QRServer::Tokenizer*: è la classe degli oggetti che spezzano una stringa nelle sue componenti, secondo un pattern descritto da un'espressione regolare;
- *QRServer::Reporter*: è la classe degli oggetti che collegano il quota report server al file server, leggono le statistiche sulle quote e ne estraggono le parti relative all'utente richiesto dal client.

RIQUADRO 1

Come ben sanno coloro che programmano in ambiente UNIX, i segnali sono un mezzo per forzare l'esecuzione di una funzione fuori dal normale ciclo di un programma; tali funzioni prendono il nome di «signal handler», cioè "gestori di segnale".
Un segnale può essere generato all'interno del processo che lo dovrà gestire, o da un altro processo o dall'intervento di un utente.
Un esempio del primo caso è il segnale *SIGALRM*, che nel programma illustrato nell'articolo viene utilizzato per forzare la chiusura della connessione con un client dopo un certo periodo di inattività. Superato il timeout, il programma genera un segnale *SIGALRM* che viene intercettato dalla routine descritta nel **Listato 1** alle righe 41-45 che, com'è facile intuire, chiudono il processo child.
Un esempio del secondo caso è costituito da un processo child che, chiudendosi, informa il parent dell'evento generando un segnale *SIGCHLD*. Nel **Listato 1** questo evento è gestito dal parent mediante la breve subroutine della linea 40.
Un esempio dell'ultimo caso è la pressione della classica sequenza CTRL-C, che genera un segnale *SIGINT*. Questo produce solitamente l'interruzione del processo in esecuzione.

Segnali e Signal Handler

Il codice

Il comando per importare nello script il codice delle classi, moduli di libreria e dichiarare dei pragma (direttive di compilazione) è la stessa: *use*.
Questo ha una sua ragione d'essere; infatti in Perl la distinzione fra modulo e classe è puramente formale. Così alla riga 9 importiamo dalla libreria *POSIX* il simbolo *WNOHANG*, e nelle linee 10-15 importiamo il codice di tutte le classi che utilizzeremo nel nostro script. Le righe dalla 17 alla 19 creano, nell'ordine, un "autenticatore" del client (*$auth*), un tokenizer (*$tokenizer*) e un parser dei comandi (*$parser*). Per convenzione, il metodo costruttore delle classi viene chiamato *new*, ma non è obbligatorio; niente vieterebbe ad un programmatore italiano di chiamare *nuova* il costruttore di un oggetto *Automobile*; un nuovo oggetto di tale classe verrebbe quindi creato con una istruzione del tipo:

```
my $p106 = nuova Automobile
                ("Peugeot",106);
```

La notazione appena vista per invocare un metodo su un oggetto (o su una classe) è detta "notazione indiretta", e consiste nello scrivere nell'ordine il nome del metodo, il nome o un oggetto di una classe e i parametri da passare al metodo. Perl supporta anche un'altra no-

LISTATO 1

Il listato dello script del quota server

```
 1 #!/usr/local/bin/perl
 2
 3 use strict ;
 4 use vars qw(
 5      %config
 6      $child_socket
 7     ) ;
 8
 9 use POSIX q(WNOHANG) ;
10 use IO::Socket::INET ;
11
12 use QRServer::Auth::Fixed ;
13 use QRServer::Parser::Base ;
14 use QRServer::Tokenizer ;
15 use QRServer::Reporter ;
16
17 my $auth   = new QRServer::Auth::Fixed qw(sandokan orzowey) ;
18 my $tokenizer = new QRServer::Tokenizer '\S+' ;
19 my $parser = new QRServer::Parser::Base ;
20
21 ################################################
22 ### CONFIGURATION ##############################
23
24 %config =
25  (
26   Socket => {
27      LocalHost => "come.see.it",
28      LocalPort => 12345,
29      Listen    => 5,
30      Proto     => "tcp",
31      Reuse     => 1
32      },
33   Server => {
34       TimeOut => 120,
35      },
36  ) ;
37
38 $| = 1 ; # Set autoflush
39
40 $SIG{CHLD} = sub { waitpid(-1,WNOHANG) > 0 } ;
41 $SIG{ALRM} = sub {
42   print $child_socket "-ERR Server timeout, closing connection\n" ;
43   warn "child $$: timeout expired, closing connection\n" ;
44   exit 1 ;
45 } ;
46
47 ### YOU SHOULD NOT NEED TO EDIT *ANYTHING* BELOW THIS LINE ##
48 ################################################
49
50 my $timeout = $config{Server}->{TimeOut} ;
51 my $listen_socket = new IO::Socket::INET %{$config{Socket}} ;
52 die "Cannot create listen socket: $!\n" unless $listen_socket ;
53
54 $listen_socket->autoflush(1) ;
55
56 warn "Quota Report Server listening at port "
57   ,$config{Socket}->{LocalPort},", PID $$\n" ;
58
59 while (1) {
60   local $child_socket = $listen_socket->accept() ;
61   next unless $child_socket ; # Needed for Solaris 2.6+
62
63   # Fork a new process for this connection
64   my $pid = fork() ;
65   die "Cannot fork: $!\n" unless defined $pid ;
66   if ($pid == 0) {
67    # Close parent socket
68    $listen_socket->close() ;
69
70    # Set autoflush on child socket
71    $child_socket->autoflush(1) ;
72    print $child_socket "+OK Quota Report Server ready to talk\n" ;
73    alarm $timeout ;
74
75    # Authentication cycle ;
76    while (not $auth->accepted()) { # accepted() returns true if
77                                    # authentication is good
78      my $input = <$child_socket> ;
79      my ($command,@params) = $tokenizer->tokenize($input) ;
80      $command = uc $command ;
81      my ($code,$msg) = $auth->exec($command,@params) ;
82      if ($code == 0) {
83         # Void command
84         print $child_socket "+OK (Void line)\n" ;
```

tazione, quella "diretta"; in notazione diretta l'istruzione precedente si scriverebbe:

```
my $p106 = Automobile->nuova
                ("Peugeot",106);
```

Ovvero: il nome o l'oggetto di una classe, una freccia, il nome del metodo e i parametri. Le due notazioni sono equivalenti ed intercambiabili. La parte fra le righe di cancelletti # contiene la configurazione dello script. Le righe dalla 27 alla 31 contengono i parametri per la creazione del socket che gestirà le connessioni: il server funzionerà sulla macchina come.see.it, sulla porta 12345, e sfrutterà il protocollo TCP. La riga 34 stabilisce il periodo di inattività dei client oltre il quale il server chiuderà la connessione. Le linee dalla 40 alla 45 sono dei signal handler. Per una spiegazione sulla loro funzione rimandiamo al **Riquadro 1**.
Si osservi ora la linea 51:

```
my $listen_socket = newIO::
Socket::INET %{$config{Socket}};
```

come detto poc'anzi, stiamo invocando il metodo new della classe IO::Socket::INET con la notazione indiretta; i parametri sono quelli definiti nelle righe 27-31.
L'oggetto risultante è *$listen_socket*, che rappresenta il socket sul quale il nostro server si metterà in ascolto, in attesa di connessioni dai client. Alla linea 54 viene invocato su questo nuovo oggetto un altro metodo, *autoflush*, passando il parametro numerico 1.
Alla riga 60 invochiamo sull'oggetto un altro metodo, accept, senza parametri. Questo metodo mette *$listen_socket* in ascolto. Appena una nuova connessione viene rilevata il metodo restituisce un nuovo oggetto della stessa classe, *$child_socket*, collegato al client che ha fatto la richiesta. Il metodo accept permette quindi all'oggetto di "clonarsi", e restituire un nuovo oggetto della stessa classe e configurato allo stesso modo.
Alla linea 64 il server stesso genera un processo gemello, che nel linguaggio UNIX è detto "child", cioè "figlio". Allo stesso modo il processo generatore è detto "parent", che

in italiano è stato tradotto come "padre". Sia nel parent che nel child l'esecuzione continua alla riga 65; lo stato e tutte le variabili che erano definite nel parent sono ricopiate uguali nel child, tranne la variabile *$pid*, che nel child vale zero.
Soffermiamoci ora sul cuore del processo child, ossia l'intervallo di righe che vanno dalla 76 alla 156, sezione in cui entrano in gioco i moduli di autenticazione e di parsing. Alla linea 76 comincia il ciclo di autenticazione; su *$auth* viene invocato il metodo *accepted*, che restituisce un valore non nullo se il client ha fornito una coppia (utente, password) corretta.
Alla linea 78 viene letta una linea dal client, che viene spezzata nelle sue componenti (comando e parametri del comando) dal metodo *tokenize* del tokenizer nella linea 79. Alla linea 80 invochiamo un altro metodo dell'oggetto *$auth*: il metodo *exec* prende come parametri un comando del client e i parametri di esso e cerca di eseguirlo, restituendo un codice di stato (che viene conservato nella variabile *$code*) e un eventuale messaggio informativo (che viene conservato nella variabile *$msg*). Il codice restituito da *$auth->exec* viene verificato nei condizionali "if" successivi, e il server esegue delle azioni descritte nella **Tabella 1**.
Si noti che per ogni azione riuscita il messaggio inviato al client inizia con +OK, mentre in caso di errore esso inizia con -ERR (analogamente a quanto avviene con il protocollo POP3).
Il programma esce dal ciclo while alla linea 101 quando l'autenticazione è avvenuta con successo (il metodo *$auth->accepted* restituisce un valore non nullo); la notifica dell'avvenuta autenticazione viene inviata sullo standard error alla linea 103.
Alla linea 107 entra in gioco un nuovo oggetto, *$qr* della classe *QRServer::Reporter*. Gli oggetti di questa classe hanno un solo metodo, *report*, che accetta come parametro il nome di un utente o un identificativo numerico, e restituisce le corrispondenti statistiche sulle quote (come si può notare alla linea 131).
Alla linea 108 inizia il ciclo co-

LISTATO 1

Prosecuzione del Listato 1

```
85     } elsif ($code == -1) {
86   # Command not understood
87   print $child_socket "-ERR Command $command not understood\n" ;
88     } elsif ($code == -2) {
89   # Generic error
90   print $child_socket "-ERR $msg\n" ;
91     } elsif ($code == -127) {
92   print $child_socket "+OK $msg\n" ;
93   warn "child $$: user ".$auth->user().'@'.$child_socket->peerhost().
94     " signing off\n" ;
95   $child_socket->close() ;
96   exit 0 ;
97     } else {
98   print $child_socket "+OK $msg\n" ;
99   alarm $timeout ;
100    }
101  }
102
103    warn "child $$: user ".$auth->user().'@'.$child_socket->peerhost().
104      " authenticated\n" ;
105    # Command cycle
106    # Create a quota reporter
107    my $qr    = new QRServer::Reporter ;
108    while ($parser->status()) {
109      my $input = <$child_socket> ;
110      my ($command,@params) = $tokenizer->tokenize($input) ;
111      $command = uc $command ;
112      my ($code,$msg) = $parser->exec($command,@params) ;
113      if ($code == 0) {
114        # Void command
115        print $child_socket "+OK (Void line)\n" ;
116        alarm $timeout ;
117    } elsif ($code == -1) {
118        # Command not understood
119        print $child_socket "-ERR Command $command not understood\n" ;
120    } elsif ($code == -2) {
121        # Generic error
122        print $child_socket "-ERR $msg\n" ;
123    } elsif ($code == 1) {
124  # Asking for a report
125  alarm $timeout ;
126  my $user = shift @params ;
127  warn "child $$: sending report about $user\n" ;
128  print $child_socket "+OK Sending data\n" ;
129
130  # @params contains one element, the quota id to report
131  foreach my $data ($qr->report($user)) {
132   my ($tree,$kused,$klimit) = @{$data}[3,4,5] ;
133   my $perc ;
134   if ($klimit == 0) {
135     $perc = 999 ;
136   } else {
137     $perc = int(100*$kused/$klimit) ;
138   }
139
140   printf $child_socket "%15s  %8d  %8d  %3d%%\n"
141     ,($tree,$kused,$klimit,$perc) ;
142  }
143
144  print $child_socket "+OK You can use USER (or ID) again, or QUIT\n" ;
145    } else {
146  $parser->quit() ;  # turns $parser's status to off
147                     # ($parser->status is turned to off)
148  warn "child $$: user ".$auth->user().'@'.$child_socket->peerhost().
149    " signing off\n" ;
150  print $child_socket "+OK Closing connection\n" ;
151    }
152   }
153
154   # Child is off. Close connection and return to main cycle
155   $child_socket->close() ;
156   exit 0 ;
157  } else {
158    # PARENT PROCESS
159
160    warn "child $pid forked; connection from ".$child_socket->peerhost().
161     "\n" ;
162
163    # Close child's socket and return back to accept
164    $child_socket->close() ;
165  }
166 }
167
168 die "Se sei arrivato fin qui, hai problemi seri!\n" ;
```

TABELLA 1

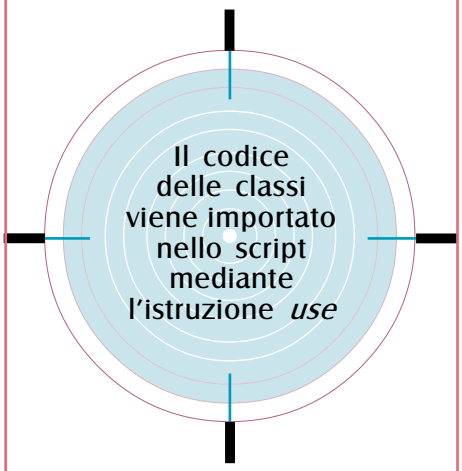
Codice	Significato	Azione
0	Linea di input vuota	Nessuna
-1	Comando incomprensibile o non implementato	Nessuna
-2	Errore generico	Nessuna
-127	Richiesta di chiusura della connessione	Chiudere la connessione e il processo child
Altro	Comando generico eseguito con successo	Inviare al client il messaggio restituito dal metodo exec e continuare il ciclo di autenticazione

Codici di ritorno del metodo exec della classe QRServer::Auth

mandi: vengono lette le linee trasmesse dal client, una per volta, e intraprese le azioni richieste. L'oggetto *$parser* ha un metodo, *status*, che restituisce un valore vero (non nullo) fintantoché dal client non arriva una richiesta di disconnessione. A quel punto l'oggetto restituisce un valore falso (ad esempio zero, o una stringa vuota), e l'iterazione successiva terminerà il ciclo while.
Le linee dalla 109 alla 157 sono analoghe alle 76-101, e ne omettiamo la descrizione.

Considerazioni finali

Come detto, il fatto di aver progettato e programmato la nostra applicazione utilizzando oggetti offre il vantaggio di una aumentata flessibilità; dimostriamo questo fatto con un semplice esempio. Supponiamo di voler supportare sul server una autenticazione diversa dalla "Fixed", in modo che utenti e password vengano verificati su un database Oracle.
Dovremo allora implementare una classe che potremmo chiamare *QRServer::Auth::Oracle*, i cui oggetti abbiano la stessa identica interfaccia di *QRServer::Auth::Fixed*. Questo significa che gli oggetti della classe *QRServer::Auth::Oracle* dovranno implementare gli stessi metodi di *QRServer::Auth::Fixed*, accettare gli stessi parametri e restituire lo stesso tipo di valori. Fatto questo, sarà sufficiente modificare nel **Listato 1** le righe 12 e 17, sostituendo la stringa Fixed con Oracle e fornendo eventualmente altri parametri al costruttore (per esempio il nome del server di database, il nome utente e la password per l'accesso ai dati) per ottenere un nuovo server con l'autenticazione basata su Oracle!



Ma questo vantaggio è poca cosa se confrontato con uno ben più importante: chi deve implementare l'oggetto non ha necessità di sapere dove verrà utilizzato. Tutto quello che è tenuto a fare è costruire un codice che rispetti le specifiche dell'interfaccia richiesta (cioè: deve avere un metodo *accepted* che non accetti parametri in ingresso e restituisca un valore non nullo se il client si è autenticato, un metodo *exec* e così via). Questo consente, date le specifiche dell'interfaccia di ogni oggetto, di commissionare a persone diverse parti diverse dell'applicazione; non programmando ad oggetti questo è solitamente più complicato. Vale quindi la pena di lasciare il porto sicuro del Perl 4 e avventurarsi nella programmazione OO e nel Perl 5: a medio-lungo termine i vostri script non potranno che trarne vantaggio!

Bibliografia

[1] P. Orsoni, *Installare ed usare "quota"*, in Login 8, Gennaio/Febbraio 1998, Edizioni Infomedia.
[2] L.D. Stein, *Client-server applications*, in "The Perl Journal", Issue #15, Fall 1999, EarthWeb,
[3] S. Srinivasan, *Advanced Perl Programming*, O'Reilly & Associates, 1997, ISBN 1-56592-220-4; capitoli 7 (Object-Oriented Programming) e 12 (Networking with sockets).
[4] G. Barr, *Network Programming with Perl*, handout del tutorial omonimo tenuto alla Open Source Software Convention 1999 / Perl Conference 3.0.
[5] L. Wall, R. Schwartz, T. Christiansen, Programming *Perl, 2nd Edition*, O'Reilly & Associates, 1996, ISBN 1-56592-149-6.
[6] Documentazione del modulo IO::Socket.
[7] P. de Mauro, *Un demone dalla A alla Z, in Login* n. 18, Settembre/Ottobre 1999, Gruppo Editoriale Infomedia.
[8] G. Lehey, *Porting UNIX software*, O'Reilly & Associates, 1995, ISBN 1-56592-126-7.

Marco Marongiu è laureato in Matematica. È amministratore e programmatore di Sistema presso il Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna (CRS4) nel gruppo "Computers & Network Services". Si occupa di tecnologie Internet dal 1994. È co-fondatore e coordinatore del Gruppo Utenti Linux di Cagliari (GULCh). Può essere contattato all'indirizzo marongiu@infomedia.it.