NAME

Higher Order Perl Review - Chapter 1

AUTHOR

Ken Youens-Clark <kclark@cshl.edu>

REVIEW

Recursion Example

Gramene::Ontology::OntologyDB::get_children_recursive

Used to get all the terms descended from a given term, e.g., all the species under ``Oryza sativa,'' which can then be used in the markers search in an ``IN (...)'' clause.

```
  sub get_children {
      my $self     = shift;
      my $id       = shift or die 'No term id';
      my $dbh      = $self->db;
      my $children = $dbh->selectcol_arrayref(
          q[
              select term2_id
              from   term_to_term
              where  term1_id=?
          ],
          {},
          ( $id )
      );
```

```
      return wantarray ? @$children : $children;
  }
```

```
  sub get_children_recursive {
    my $self   = shift;
    my $tax_id = shift;
    my $db     = $self->db;
```

```
    my @children;
    for my $id ( $self->get_children( $tax_id ) ) {
        push @children, $id, $self->get_children_recursive( $id );
    }
```

```
    return uniq( @children );
}
```

Gramene::CDBI::_extract

When calling the ``get_related'' method on a Gramene::CDBI (Class::DBI) object, I originally wrote the code recursively:

```
# ------------------------------------------------
sub _extract {
# Internal subroutine called recursively by "get_related" to
# drill into the schema to get objects

    my ( $from, $path ) = @_;

    return if !defined $from;

    my $next = shift @$path or return;

    return if !$from->can( $next );

    my @return;
    for my $object ( $from->$next() ) {
        next if !defined $object;

        if ( @$path ) {
            push @return, _extract( $object, $path );
        }
        else {
            push @return, $object;
        };
    }

    return @return;
}
```

E.g., you might have a ``marker'' and you wanted it's related ``mappings.'' The code would figure out the path from the starting table to the ending table and then get each object from the first table in the path, use that to get all of it's children from the next table in the path, on down (depth-first) and finally return the results. It turned out to be horribly inefficient, and the results had to be unique'd, so I rewrote it iteratively such that I used the path to construct a single SQL statement that would give me the IDs for the ending table's related data and then instantiate objects using those IDs. This takes a fraction of the time.

Callback Example

SQL::Translator::format_package_name

Allows a callback to mutate the ``package'' for each class created from a schema. As you can also do this for table names and primary/foreign key names, these methods internally call ``_format_name,'' a further level of abstraction:

```
sub format_package_name {
    return shift->_format_name('_format_package_name', @_);
}

# ----------------------------------------------------------------------
# The other format_*_name methods rely on this one.  It optionally
# accepts a subroutine ref as the first argument (or uses an identity
# sub if one isn't provided or it doesn't already exist), and applies
# it to the rest of the arguments (if any).
# ----------------------------------------------------------------------
sub _format_name {
    my $self = shift;
    my $field = shift;
    my @args = @_;

    if (ref($args[0]) eq 'CODE') {
        $self->{$field} = shift @args;
    }
    elsif (! exists $self->{$field}) {
        $self->{$field} = sub { return shift };
    }

    return @args ? $self->{$field}->(@args) : $self->{$field};
}
```

E.g., maybe you want table names to be UPPERCASE:

```
my $translator = SQL::Translator->new(
    ...
    format_table_name => sub {my $tablename = shift; return uc($tablename)},
);
```

SQLT also allows the insertion of completely different ``parsers'' and ``producers'' via hooks/callbacks:

```
# Invokes My::Groovy::Parser::parse()
$tr->parser("My::Groovy::Parser");

# Invoke an anonymous subroutine directly
$tr->parser(sub {
  my $dumper = Data::Dumper->new([ $_[1] ], [ "SQL" ]);
  $dumper->Purity(1)->Terse(1)->Deepcopy(1);
  return $dumper->Dump;
});
```

Or you can filter data:

```
my $translator  = SQL::Translator->new(
        ...
        filters => [
            sub { ... },
            [ "Names", table => 'lc' ],
        ],
        ...
    );
```

CMap's ``object plugin'' system

Via CMap's configuration file, installers can customize the data shown on the various object details pages by installing custom code:

```
    <object_plugin>
        map_set_info <<EOF
            sub {
                my $map_set = shift;
                $map_set->{'foo'} = 'bar';
                push @{ $map_set->{'xrefs'} }, {
                    xref_name => 'Google',
                    xref_url  => "http://www.google.com/?q=";.
                        $map_set->{'map_set_short_name'}
                };

                push @{ $map_set->{'attributes'} }, {
                    attribute_name  => 'Favorite Color',
                    attribute_value => 'Blue. No, red! Ahhhhh!',
                };
            }
        EOF
        feature Gramene::Marker::Marker2CMap::new
    </object_plugin>
```

Text::RecordParser::field_compute:

A callback applied to the fields identified by position (or field name if `bind_fields` or `bind_header` was called).

The callback will be passed two arguments:

1. The current field
2. A reference to all the other fields, either as an array or hash reference, depending on the method which you called.

If data looks like this:

```
    parent    children
    Mike      Greg,Peter,Bobby
    Carol     Marcia,Jane,Cindy
```

You could split the ``children'' field into an array reference with the values like so:

```
    $p->field_compute( 'children', sub { [ split /,/, shift() ] } );
```

The field position or name doesn't actually have to exist, which means you could create new, computed fields on-the-fly. E.g., if you data looks like this:

```
    1,3,5
    32,4,1
    9,5,4
```

You could write a field_compute like this:

```
    $p->field_compute( 3,
        sub {
            my ( $cur, $others ) = @_;
            my $sum;
            $sum += $_ for @$others;
            return $sum;
        }
    );
```

Field ``3'' will be created as the sum of the other fields. This allows you to further write:

```
    my $data = $p->fetchall_arrayref;
    for my $rec ( @$data ) {
        print "$rec->[0] + $rec->[1] + $rec->[2] = $rec->[3]\n";
    }
```

Prints:

```
    1 + 3 + 5 = 9
    32 + 4 + 1 = 37
    9 + 5 + 4 = 18
```