Tricks with DBI and Transactions

Paul Fenwick
<pjf@perltraining.com.au>

8th May, 2002

What is DBI?

- The most commonly used Perl module for database access.

- Supports a great many database back-ends.

- Appreciated for its consistant interface.

Some typical (but wrong) DBI code

```
use DBI;

my $customer = "pjf";
my $dbh = DBI->connect($data_source, $username, $auth);

my $sth = $dbh->prepare("SELECT spending,purchases
              FROM customer_records
              WHERE customer = ?");

$sth->execute($customer);

my ($spending, $purchases) = $sth->fetchrow_array;

$dbh->do("UPDATE customer_records SET spending = 0,
          purchases = 0 WHERE customer = ?",
          undef, $customer);

$dbh->do("DELETE from monthly_avg WHERE customer = ?",
         undef, $customer);

my $avg_purchase = $spending/$purchases;

$dbh->do("INSERT into monthly_avg (avg,spending)
         VALUES (?, ?)",undef, $avg_purchase,$customer);
```

Problems with the code

- No checking of return values. Mindlessly continues regardless of errors.

- Potential for interruption between clearing `customer_records` and `monthly_avg` and inserting the new records.

- Potential for the code to throw an exception if `$purchases` is zero.

Enter transactions...

- Transactions allow a block of SQL statements to be executed in an all-or-nothing affair.

- Supported by many database engines.

- Transactions can be "rolled back" in the case of failure. Great for our previous example.

DBI code revisited

```
use DBI;

my $customer = "pjf";
my $dbh = DBI->connect($data_source, $username, $auth,
                       {AutoCommit => 0});
$dbh->begin_work;

# All that SQL stuff goes here.

$dbh->commit;
```

Not the entire fix

- Much better. The transaction ensures that all statements are executed, or none of them are.

- Still no checking of return values for database access.

- No chance of recovery from errors.

The old way of error-checking...

```
use DBI;

my $customer = "pjf";
my $dbh = DBI->connect($data_source, $username, $auth,
                       {AutoCommit => 0});
        or die $DBI::errstr;

$dbh->begin_work or die $dbh->errstr;

my $sth = $dbh->prepare("SELECT spending,purchases
                FROM customer_records WHERE customer = ?")
        or die $dbh->errstr;

$sth->execute($customer) or die $sth->errstr;

# ... etc or die $dbh->errstr;
```

Error handling in DBI

- Checking every database access (with `or die ...`) can get awfully boring and repetitive.

- Makes code more difficult to read.

- Easy to forget an `or die` or two.

- Some error tests are little known. (Eg, testing `$sth->err` after fetching data.)

Error handling – the better way

```
my $dbh = DBI->connect($data_source, $username, $auth,
                       {RaiseError => 1, PrintError => 0});

# Do database things here on without checking returns.
```

- Setting `RaiseError` on causes any error to throw a die with a helpful error message.
- We turn `PrintError` off to avoid getting errors twice.
- Code is readable again. Weird subtle errors are checked for. Peace and harmony is restored.
- Doesn't necessarily tell you which statement caused the problem (neither did our previous `die` statements).

Better yet...

```
my $dbh = DBI->connect($data_source, $username, $auth,
                  {RaiseError => 1, PrintError => 0,
                   ShowErrorStatement => 1});
```

- `ShowErrorStatement` appends the SQL statement to the error message.
- No longer any need to put the statement in a variable first for easy printing.
- Combined with `RaiseError` (or even on its own), it's a fantastic time-saver.

Transactions and exception handling

```
use DBI;

my $customer = "pjf";
my $dbh = DBI->connect($data_source, $username, $auth,
          {RaiseError => 1, PrintError => 0,
           ShowErrorStatement => 1});

eval {
        $dbh->begin_work;

        # Database accesses and dangerous divide-by-zero
        # operations go here.

        $dbh->commit;
};

if ($@) {
        local $dbh->{RaiseError} = 0;
        $dbh->rollback;
        # Handle the error here.
}
```

How it works

- The `eval` catches any errors, be they from database accesses or caused by other errors.

- The handling block (which examines `$@`) tests to see if an error occurs and recovers appropriately.

- `RaiseError` is turned off in case the `rollback` might cause an error (which is unlikely).

- Extremely handy for processing a large number of records, where the failure of one record should not stop the others being processed.

Another way of handling errors

```
my $dbh = DBI->connect($data_source, $username, $auth,
          {RaiseError => 1, PrintError => 0,
           ShowErrorStatement => 1});

{
        local $dbh->{HandleError} = sub {
                my ($error, $dbh) = @_;
                return (error_is_ugly($error) ? 0 : 1);
        };

        # SQL statements which ignore non-ugly errors
        # go here.
}
```

Using `HandleError`

- `HandleError` lets you set your own database handler to deal with database errors.

- `HandleError` can "mute" an error by returning a true value. This causes the effects of `RaiseError` and `PrintError` to be ignored. The return value of the original call will still indicate that an error has occured, and DBI's `errstr` and `err` methods will still contain data about the error.

HandleError continued

- It's possible to (somewhat) make a failed DBI call turn into a successful looking one:

```
$dbh->{HandleError} = sub {
      $_[0]->set_err(0,"");
      $_[2] = "New result";
      return 1;
};
```

- $_[2] is the return value from the "failed" call.

- This *only* works for methods which return a single (scalar) value.

- If you're in a transaction, it might still have been terminated by the database, even if you do hide the error.

- Beware of infinite loops from executing SQL that might fail.

Putting it all together

```
my $dbh = DBI->connect($data_source, $username, $auth,
          {RaiseError => 1, PrintError => 0,
           ShowErrorStatement => 1, AutoCommit => 0});

my $gremlins = 0;  # Failure from gremlins.

# Read lots of records, insert them into the database.

{
    local $dbh->{HandleError} = sub {
        $gremlins++ if $_[0] =~ /gremlin/i;
        return 0;
    };

    while (<>) {
        eval {
             $dbh->begin_work;
             chomp;
             @fields = split;
             $dbh->do("INSERT into users
                       VALUES (?, ?, ?, ?)",
                       undef, @fields);
             $dbh->do("INSERT into credit VALUES (?, ?)",
                      undef, $fields[0],30);
             $dbh->commit;
        };

        if ($@) { print "FAILED: $_\n"; }
    }
}

print "$gremlins records failed due to gremlin attacks.\n";
```

Conclusion

- DBI is fantastic for accessing databases.
- Checking all statements for errors is a pain, and can be avoided by using `RaiseError`.
- It is possible to gracefully recover from failed transactions using `eval`.
- Using `HandleError` you can mute or hide the occurance of database errors.