ORM によるデータベース操作 (DBIx::MoCo を使った開発)

今日は何をしますか

- データベース・OR マッパーの役割を理解し、MoCo を使えるようになる
- 来週以降の Web アプリのための下地づくり

カリキュラムについて

- Perl & OOP
- **OR マッパーによるデータベース操作 ← いまここ**
  - DBIx::MoCo / データベース
- WAF によるウェブアプリケーション開発
  - Ridge / Web アプリケーション (サーバー側)
- JavaScript で学ぶイベントドリブン
  - JavaScript / Web アプリケーション (クライアント側)
- インターネットサービスの企画 (課題なし)
- はてなのインフラストラクチャについて (課題なし)

今日の講義

課題

- MoCo を用いて、コマンドラインインターフェースで日記を書けるツールを作成してもらいます

構成

- 基本編
  - ORM、MoCo の基本的な概念や使い方を紹介します
- 実践編
  - MoCo を使った簡単なブックマーク管理ツールの作り方をなぞります
- 課題の解説
- 駆け足で進みますのでがんばってついてきてください
- 質問があれば途中でも聞いてください

データの永続化

- アプリケーションが扱うデータ
  - ユーザ情報・何らかの投稿・後から集計したデータなど
- メモリ (変数) にデータを持っておくだけでは当然ダメ
  - アプリケーションの複数のインスタンスは物理的・時間的制約によりメモリを共有できない
- というわけでデータの永続化 (データストレージ) が必要です

簡単なデータストレージの例

- 2ちゃんねるの機能
    - 多くの鯖に分散されたスレッド
    - スレッドを閲覧
    - スレッドの最後にレスを追加
    - スレッドを立てる
- データストレージ = dat ファイル (1 行 1 レス)

```
 名無しさん<>sage<>2011/08/19(金) 06:19:10.13 <> &gt;&gt;1乙 <>
名無しさん<>sage<>2011/08/19(金) 06:21:30.21 <> こんにちは <>
```

2ちゃんねる+αを考えてみる

ふつうのウェブサービスっぽくしてみる

- ユーザログイン: 過去の書き込みを一覧
    - → dat ファイルにユーザ名を記録して、一覧するときに全部検索？
- 耐障害性: マシンが故障してもサービスが継続できるように
    - → dat ファイルを複数のマシンにコピーする？
- 面倒

一般にウェブサービスは成長します

- データは大量・増える一方
    - データの量に (あまり) 依らずに、効率よくデータを取得できる必要がある
- サービスは 24 時間 365 日提供したい
    - アプリケーション・データの冗長化

そのための データベース です

- それなりの規模のウェブサービスを作るときに想像されること……
  - データを構造化
  - 取得に効率のよいデータの保管
  - 複数のノードにデータを持って耐障害性を確保
- などなどいい感じにまとまっていて
- ライブラリがあり
- 運用実績も豊富

使ったことありますか?

- 関係 (リレーショナル) データベース / SQL
- OR マッパー
- DBI
- MoCo

関係データベースとは？

- 関係モデルに基づくデータベース

関係モデル

- 関係は属性と組 (タプル) の集合で表される
- 関係代数に基づき演算が定義される

R: (ID, 名前, 誕生日) = { (1, 初音ミク, 2007-08-31), (2, 鏡音リン, 2007-12-27), (3, 鏡音レン, 2007-12-27), (4, 巡音ルカ, 2009-01-30) }

関係データベース

- データベースに複数のテーブルが属する
- データは表で表される
  - 表 = 関係、カラム = 属性、行 (レコード) = タプル

例) artist テーブル:

id	name	birthday
1	初音ミク	2007-08-31
2	鏡音リン	2007-12-27
3	鏡音レン	2007-12-27
4	巡音ルカ	2009-01-30

album テーブル:

id	artist_id	name	released_on
1	1	みくのかんづめ	2008-12-3

SQL

- SQL という言語により表のデータの問い合わせ、更新などを行う

```
 SELECT birthday FROM artist WHERE name = '初音ミク';
SELECT * FROM artist WHERE birthday < '2009-01-01' ORDER BY birthday DESC;
```

```
INSERT INTO artist (id, name, birthday) VALUES (5, '重音テト', '2008-04-01');
```

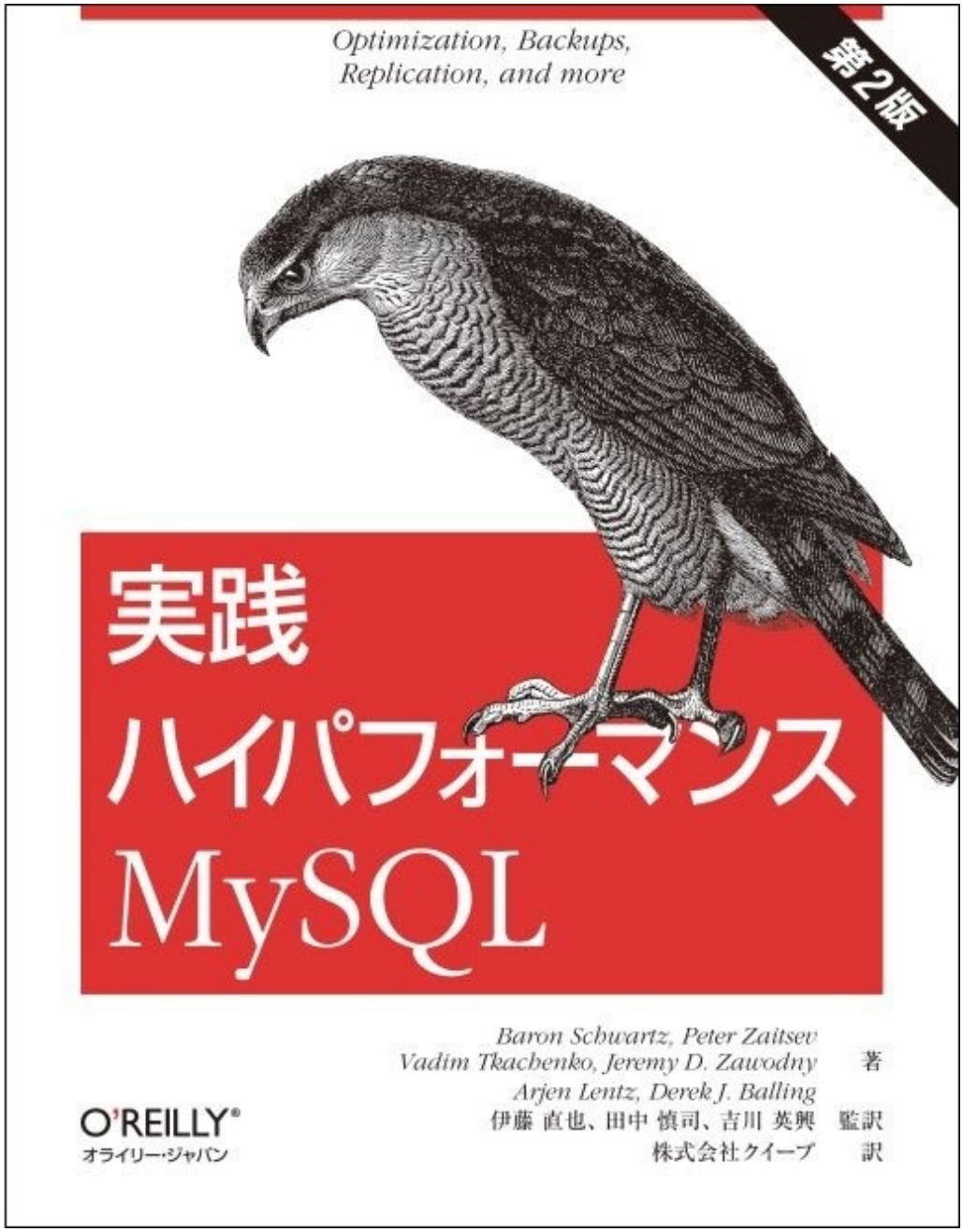
```
UPDATE artist SET birthday = '2008-07-18' WHERE name LIKE '鏡音%';
```

- 動詞 (SELECT, INSERT, UPDATE, DELETE)
- 対象: WHERE …

キー (インデックス)

- カラムの組み合わせについてインデックス (索引) を作成することができる
- 普通のインデックス
    - そのカラムについてソートしたい時に
- プライマリキー (主キー)
    - テーブル内で一意なキー
    - まず変わらない値を設定
- ユニークキー
    - テーブル内で一意なキー (の組み合わせ)

前提本



RDBMS

- Relational DataBase Management System
- 関係データベースの実装
    - いろいろ面倒を見てくれる
    - ファイルの配置、ロック
    - SQL の解釈
    - レプリケーション
    - トランザクション
- はてなでは MySQL を採用

Perl から RDB を使う: DBI

- Perl からデータベースに接続するモジュール
    - OR マッパーではない

- DBI
  - (DBD::*) MySQL、PostgreSQL、SQLite、…

DBI を用いる

```
 use DBI;
use DateTime::Format::MySQL;

my $dbh = DBI->connect('dbi:mysql:dbname=vocaloid', 'root', '') or die $DBI::e
rrstr;
my $artists = $dbh->selectall_arrayref('SELECT * FROM artist WHERE birthday <
? ORDER BY birthday ASC', { Slice => {} }, '2008-01-01');

# [
#   {
#     'id' => '1',
#     'name' => '初音ミク',
#     'birthday' => '2007-08-31'
#   },
#   {
#     'id' => '2',
#     'name' => '鏡音リン',
#     'birthday' => '2007-12-27'
#   },
#   {
#     'id' => '3',
#     'name' => '鏡音レン',
#     'birthday' => '2007-12-27'
#   }
# ]

my $artist = $artists->[0];
$artist->{name}; # '初音ミク'
DateTime::Format::MySQL->parse_date($artist->{birthday})->strftime('%x'); # 'A
ug 31, 2007'

my $albums = $dbh->selectall_arrayref('SELECT * FROM album WHERE artist_id = ?
', { Slice => {} }, $artist->{id});
```

- DB とのコネクションの面倒を見なきゃいけない
- SQL を直に書かなきゃいけない
- 結果がただのハッシュ

OR マッパーの登場です

- ORM: **O**bject-**R**elational **mapping** (オブジェクト関係マッピング)
  - Object: オブジェクト指向でいうオブジェクト
  - Relational: 関係データベース (relational database; RDB)

オブジェクト指向とは？

OR マッパーを使った書き方 (DBIx::MoCo)

```
 use Vocaloid::MoCo::Artist;

my $artists = Vocaloid::MoCo::Artist->search(
    where => {
        birthday => { '<' => '2008-01-01' },
    },
    order => 'birthday ASC',
);

my $artist = $artists->first;

$artist->name; # '初音ミク'
$artist->birthday->strftime('%x'); # 'Aug 31, 2007'

my $albums = $artist->albums;
```

- DB への接続、SQL の生成、型変換などは OR マッパーが隠蔽してくれる
  - ただし必要な情報が入ったモジュール (Vocaloid::MoCo::Artist) は書かないといけない
  - 詳しくは後で

MoCo (もこ) とは?

- DBIx::MoCo
- はてなのサービスで使われている OR マッパー
- はてな社内で開発
  - jkondo (社長です) 作、現在は motemen (わたしです) がメンテナ
- はてなスター、はてなブックマーク、はてなハイク、うごメモはてな、はてなココ、...

MoCo で何ができたか

```
my $artists = Vocaloid::MoCo::Artist->search(...);
```

- SQL の自動生成
- 得られた行たちのオブジェクト化

```
$artist->name;
$artist->birthday->strftime('%x');
```

- 行のカラムへのアクセス
- ちなみに書き換えもこのメソッドでできます

```
my $albums = $artist->albums;
```

- 意味的に関連する別のテーブルへのアクセスをメソッドとして定義できる

OR マッパーの機能

- DB に格納されているレコードとコード中で活躍するオブジェクトとの変換
- 接続情報の管理
- SQL の生成
- 型の自動変換

など

MoCo の概要

- 1 クラスが 1 テーブルに対応 (Active Record パターン)
  - 1 インスタンス = テーブルの 1 行
  - テーブルに対する操作 → クラスメソッド
  - `MoCo::Artist->search()`
  - レコードに対する操作 → インスタンスメソッド
  - `$artist->name()`

```
package Vocaloid::MoCo::Artist;
use base 'DBIx::MoCo';

__PACKAGE__->table('artist'); # MoCo::Artist は artist テーブルに対応

package Vocaloid::MoCo::Album;
use base 'DBIx::MoCo';

__PACKAGE__->table('album'); # MoCo::Album は album テーブルに対応
```

- 以下、MoCo のメソッドをざっと見ていきます！
- CRUD のうちどれに当たるかを意識しておきましょう

$class->find

- 条件に合う行を取得

```
my $artist = Vocaloid::MoCo::Artist->find(name => '初音ミク');
print $artist->id, "\n";
print $artist->name, "\n";
```

```
print $artist->birthday, "\n";
```

```
SELECT * FROM artist WHERE name = '初音ミク' LIMIT 1;
```

id	name	birthday
1	初音ミク	2007-08-31

$class->search

- 条件に合う行を複数取得

```
my $artists = Vocaloid::MoCo::Artist->search(
    where => {
        name => { -like => '鏡音%' },
    },
    order => 'id ASC',
    # offset => 0,
    limit => 10,
);

$artists->each(sub {
    print $_->name, "\n";
});
```

- where => {} で条件を指定する。[cpan:SQL::Abstract] 形式で指定できる
- order で順序を指定する (カラム名 ASC|DESC)
- offset, limit で範囲を絞る

```
SELECT * FROM artist WHERE name LIKE '鏡音%' ORDER BY id ASC LIMIT 10;
```

id	name	birthday
2	鏡音リン	2007-12-27
3	鏡音レン	2007-12-27

- DBIx::MoCo::List オブジェクトが返ってくる

DBIx::MoCo::List

- search メソッドで返ってくるオブジェクト
- ほとんどただの配列だけど、操作を Ruby っぽく書ける
- DB 操作に関わるメソッドはありません

- c.f. [cpan:List::Rubyish]

- **push, pop, shift, unshift, add, append, prepend**

- **length, size**

- **first, last**

- **map**, collect, **each**

- **grep**

- slice, zip
- compact
- flatten
- delete, ~~deleteif~~, deleteat
- inject
- find
- join
- reduce
- sum
- uniq
- dup
- dump

$class->create

- 行の挿入

```
my $new_artist = Vocaloid::MoCo::Artist->create(
    id => 5,
    name => '重音テト',
    birthday => '2008-04-01',
);
print $new_artist->name;
```

```
INSERT INTO artist (id, name, birthday)
    VALUES (5, '重音テト', '2008-04-01');
```

id	name	birthday
1	初音ミク	2007-08-31
2	鏡音リン	2007-12-27
3	鏡音レン	2007-12-27
4	巡音ルカ	2009-01-30
5	重音テト	2008-04-01

$instance->$column()

- 得られた行の 1 つのフィールドへのアクセス (読み取り/書き込み)

```
$artist = Vocaloid::MoCo::Artist->find(name => '初音ミク');

## 現在の値を取得
print $artist->name;
```

```
## 新しい値を設定
$artist->name('弱音ハク');
```

```
UPDATE artist SET name = '弱音ハク' WHERE id = 1;
```

id	name	birthday
1	初音ミク弱音ハク	2007-08-31
2	鏡音リン	2007-12-27
3	鏡音レン	2007-12-27
4	巡音ルカ	2009-01-30
5	重音テト	2008-04-01

$instance->delete

- 行の削除

```
$artist = Vocaloid::MoCo::Artist->find(name => '初音ミク');
$artist->delete;
```

```
DELETE FROM artist WHERE id = 1;
```

id	name	birthday
2	鏡音リン	2007-12-27
3	鏡音レン	2007-12-27
4	巡音ルカ	2009-01-30

has-many

- 「ひとつの artist に複数の album が対応する」

```
package Vocaloid::MoCo::Artist;

# __PACKAGE__->has_many(
#     albums => 'Vocaloid::MoCo::Album', {
#         key => { id => 'artist_id' }
#     }
# );

sub albums {
    my $self = shift;
    return Vocaloid::MoCo::Album->search(
        where => {
            artist_id => $self->id,
        },
    );
```

```
}
```

とメソッドを定義しておくと

```
$artist = Vocaloid::MoCo::Artist->find(name => '初音ミク');
$artist->albums->each(sub {
    print $_->title, "\n";
});
```

とかできる。

```
SELECT * FROM album WHERE artist_id = 1;
```

has-a

- 「ひとつの album にひとつの artist が対応する」

```
package Vocaloid::MoCo::Album;

# __PACKAGE__->has_a(
#     albums => 'Vocaloid::MoCo::Artist', {
#         key => { artist_id => 'id' }
#     }
# );

sub artist {
    my $self = shift;
    return Vocaloid::MoCo::Artist->find(id => $self->artist_id);
}
```

とメソッドを定義しておくと

```
$album = Vocaloid::MoCo::Album->find(title => 'supercell');
$artist = $album->artist;
print $artist->name, "\n";
```

とかできる。

```
SELECT * FROM album WHERE title = 'supercell' LIMIT 1;
```

inflate/deflate

- 行の特定のカラムについて、「スカラー値 ⇔ Perl のオブジェクト」の変換ルールを指定
- 例: DATE 型 ⇔ DateTime
- 例: 文字列 ⇔ URI

```
package Vocaloid::MoCo::Artist;
```

```
...

__PACKAGE__->inflate_column(
    birthday => {
        inflate => sub {
            if ($_[0] & $_[0] ne '0000-00-00') {
                return DateTime::Format::MySQL->parse_date($_[0]);
            }
        },
        deflate => sub {
            if ($_[0]) {
                return DateTime::Format::MySQL->format_date($_[0]);
            } else {
                return '0000-00-00';
            }
        },
    },
);

...

$artist->birthday->strftime('%x');
```

SQL を書く: SQL::Abstract を使った書き方

- もうすぐ休憩です
- [cpan:SQL::Abstract]

```
 my $name = '初音ミク';
my $artists = Vocaloid::MoCo::Artist->search(
    where => {
        name => $name,
        birthday => { '>', $dt },
    },
);
```

```
 SELECT * FROM artist WHERE name = ? AND birthday > ?;
```

SQL::Abstract を使わない書き方

```
 # 悪い例
my $name = '初音ミク';
my $artists = Vocaloid::MoCo::Artist->search(
    where => qq(name = '$name'), # 危険!
);
```

```
 SELECT * FROM artist WHERE name = '初音ミク';
```

セキュリティー

- 文字列連結は一般に安全ではない！
  - $name がユーザの入力だった場合
  - $name = q('; DROP TABLE artist; ');
- 書いたプログラムが安全かは常に気にしましょう

```
SELECT * FROM artist WHERE name = ''; DROP TABLE artist; '';
```

語られなかったこと

- 透過キャッシュ
- has*a*, *has*many
- retrieve
- muid

休憩

bookmark.pl を作ってみよう

- 実践編です
- 小さなブックマークアプリを書いていく過程を見ていきます

できること (大雑把に)

- ユーザは URL (エントリ) を個人のブックマークに追加し、コメントを残せる
- エントリはユーザに共通の情報を持つ (ページタイトルなど)
- とりあえず一人用で (マルチユーザも視野にいれつつ)

add, list, delete

3 操作くらいできるようにしてみたい

- bookmark.pl add <*url*> [コメント]
  - ブックマークを追加

```
 % ./bookmark.pl add http://www.yahoo.co.jp/ ヤッホー
bookmarked [8] Yahoo! JAPAN <http://www.yahoo.co.jp/>
  @2011-08-16 ヤッホー
```

- bookmark.pl list
  - ブックマークの一覧を出力

```
 % ./bookmark.pl
 *** motemen's bookmarks ***
[8] Yahoo! JAPAN <http://www.yahoo.co.jp/>
  @2011-08-16 ヤッホー
[7] The CPAN Search Site - search.cpan.org <http://search.cpan.org/>
  @2011-08-16 くぱん
[6] はてな <http://www.hatena.ne.jp/>
  @2011-08-16 はてー
[4] Google <http://www.google.com/>
  @2011-08-16 ごー
[1] motemen <http://motemen.appspot.com/>
```

```
  @2011-08-15 モテメンドットコム
```

- bookmark.pl delete <*id*>
  - ブックマークを削除

```
 % ./bookmark.pl delete 4
deleted [4] Google <http://www.google.com/>
  @2011-08-16 ごー
```

...という bookmark.pl を作ってみよう

コードを手元に

```
 % git clone https://github.com/hatena/Intern-Bookmark-2011.git
% cd Intern-Bookmark-2011
% git checkout -t origin/moco
% git submodule update --init
% mysqladmin -uroot create intern_bookmark
% mysql -uroot intern_bookmark < db/schema.sql
```

以降こんな感じでいきます

- テーブルの設計
- モデル (MoCo) の設計
- DataBase.pm を必要最低限書く
- MoCo.pm を必要最低限書く
- アプリケーションのロジックを書く

テーブルの設計

- どんな概念が登場するか？
  - 何が一意であるべきか

user

id	name
1	antipop
2	motemen
3	cho45

- UNIQUE KEY (name)

entry

ユーザに共通の、URL に関する情報

id	url	title
1	http://www.example.com/	IANA — Example domains
2	http://www.hatena.ne.jp/	はてな
3	http://motemen.appspot.com/	motemen

- UNIQUE KEY (url)

bookmark

ユーザが URL をブックマークした情報 (ユーザ×エントリ)

id	user*id*	entryid	comment
1	1 (= antipop)	1 (= example.com)	例示用ドメインか〜。
2	1	2 (= はてな)	はてな〜。
3	2 (= motemen)	3 (= motemen.com)	僕のホームページです
4	3 (= cho45)	3	モテメンさんのホームページですね
5	3	1	example ですね

- UNIQUE KEY (user*id*, *entry*id)

コードの設計

- ロジックをモデル (MoCo) に集約
  - プログラムがすっきりする
  - テスト書きやすい

最初に利用例を考えてみるといいです

```
# ブックマーク一覧
$user->bookmarks;

# ブックマーク追加
$user->add_bookmark(
    url => $url,
    comment => $comment,
);

# ブックマーク削除
$user->delete_bookmark($entry);
```

- とりあえずテストを書いてみる
- とりあえず一番外側のスクリプトを書いてみる

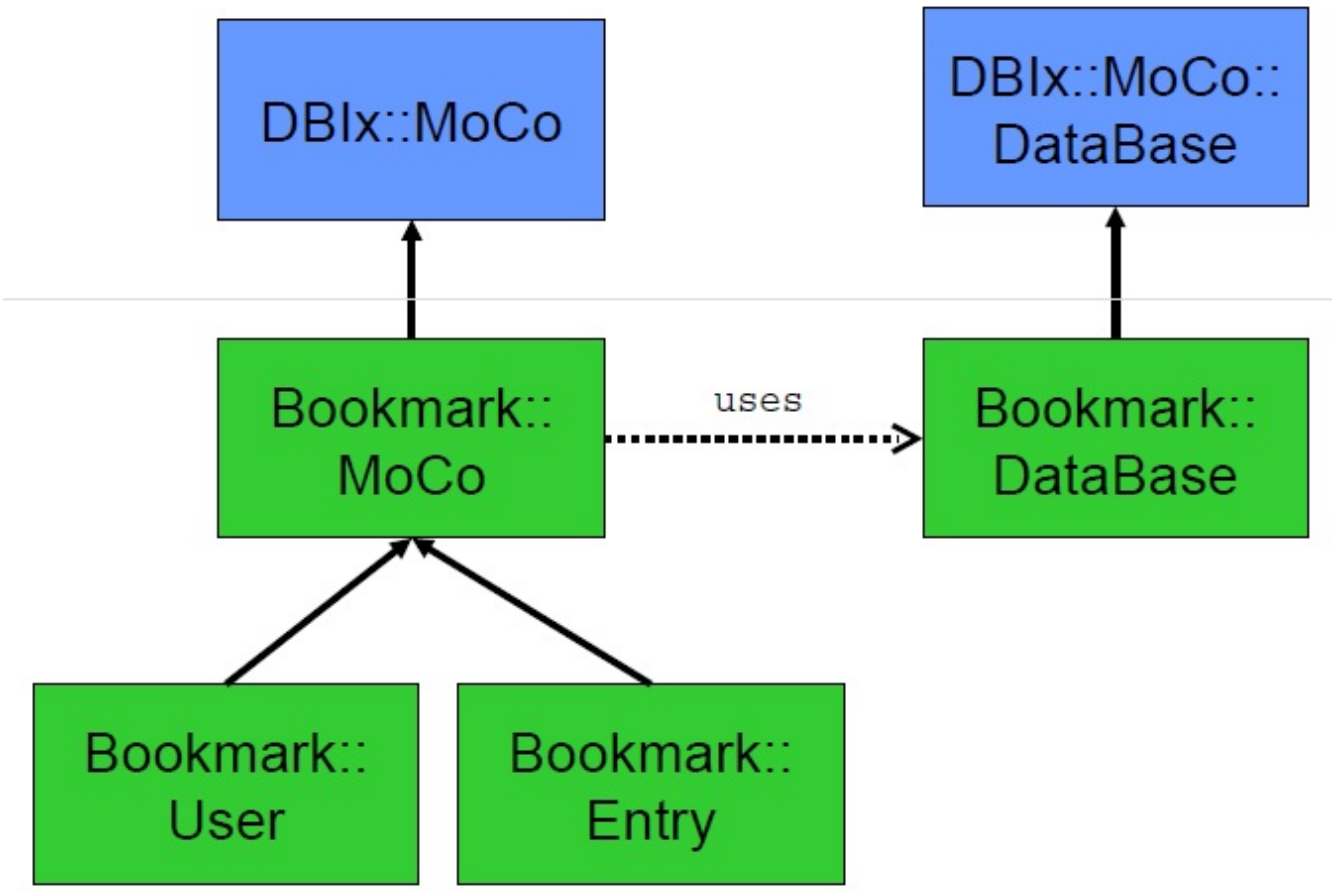
bookmark.pl

- アプリケーションのロジックはモデルクラス (MoCo) に集約
- コマンドライン周りの処理だけ記述

```
#!/usr/bin/env perl
use strict;
use warnings;
use FindBin;
use lib "$FindBin::Bin/lib", glob "$FindBin::Bin/modules/*/lib";
use Intern::Bookmark::MoCo;
use Pod::Usage; # for pod2usage()
use Encode::Locale;
```

```
binmode STDOUT, ':encoding(console_out)';

my %HANDLERS = (
    add  => e188397e11e5c74fe75b4bfa435fa7a5a1f2f734amp;add_bookmark,
    list => e188397e11e5c74fe75b4bfa435fa7a5a1f2f734amp;list_bookmarks,
);

my $command = shift @ARGV || 'list';

my $user = moco('User')->find(name => $ENV{USER}) || moco('User')->create(name
 => $ENV{USER});
my $handler = $HANDLERS{ $command } or pod2usage;

$handler->($user, @ARGV);

exit 0;

sub add_bookmark {
    my ($user, $url, $comment) = @_;

    my $bookmark = $user->add_bookmark(
        url => $url,
        comment => $comment,
    );
    print 'bookmarked ', $bookmark->as_string, "\n";
}

sub list_bookmarks {
    my ($user) = @_;

    printf " *** %s's bookmarks ***\n", $user->name;

    my $bookmarks = $user->bookmarks;
    foreach my $bookmark (@$bookmarks) {
        print $bookmark->as_string, "\n";
    }
}
```

MoCoを使う流れ



- DBIx::MoCo::DataBase を継承した Bookmark::DataBase
- DBIx::MoCo を継承した Bookmark::MoCo
- Bookmark::MoCo を継承した Bookmark::MoCo::***

submodule として追加

```
% git submodule add https://github.com/hatena/DBIx-MoCo.git modules/DBIx-MoCo
```

まずは DataBase.pm を書いてみるか

- DB の接続情報を設定
- dbi:mysql:dbname=intern_bookmark
  - DBD::mysql を使用して MySQL の intern_bookmark データベースに接続
- このクラスは書いたら終わり

```
 package Intern::Bookmark::DataBase;
use strict;
use warnings;
use base 'DBIx::MoCo::DataBase';

__PACKAGE__->dsn('dbi:mysql:dbname=intern_bookmark');
__PACKAGE__->username('root');
__PACKAGE__->password('');
```

```
1;
```

それから MoCo.pm

- モデルクラス共通の振る舞いを記述

```
 package Intern::Bookmark::MoCo;
use strict;
use warnings;
use base 'DBIx::MoCo';
use Bookmark::DataBase;

__PACKAGE__->db_object('Bookmark::DataBase');

1;
```

moco()

- Intern::Bookmark::MoCo には moco という関数が

```
 use Bookmark::MoCo::User;
use Bookmark::MoCo::Entry;
Bookmark::MoCo::User->find(name => 'onishi')
Bookmark::MoCo::Entry->find(id => 1)
```

と書く代わりにこう書ける

```
 use Bookmark::MoCo;
moco('User')->find(name => 'onishi')
moco('Entry')->find(id => 1)
```

各モデルクラス

- DataBase クラスの指定は親クラスで行われているので不要

```
 package Intern::Bookmark::MoCo::User;
use strict;
use warnings;
use base 'Intern::Bookmark::MoCo';

__PACKAGE__->table('user');

1;
```

- その他 MoCo::Entry, MoCo::Bookmark も同じように
- 3 つのテーブルに対応する 3 つのクラス

relationship 的なもの

```
```

```
$user->bookmarks;
```

ってやりたい

```
package Intern::Bookmark::MoCo::User;

...

sub bookmarks {
    my $self = shift;
    return moco('Bookmark')->search(
        where => {
            user_id => $self->id,
        },
        order => 'created_on DESC',
    );
}
```

カラムをオブジェクトとして扱う (inflate / deflate)

- $bookmark->created_on が DateTime オブジェクトになってると何かと便利

```
__PACKAGE__->inflate_column(
    created_on => {
        inflate => sub {
            my $value = shift;
            return $value eq '0000-00-00 00:00:00' ? undef : DateTime::Format:
:MySQL->parse_datetime($value);
        },
        deflate => sub {
            my $dt = shift;
            return DateTime::Format::MySQL->format_datetime($dt);
        }
    }
);
```

トリガ

```
package Intern::Bookmark::MoCo;

__PACKAGE__->add_trigger(
    before_create => sub {
        my ($class, $args) = @_;
        foreach my $col (qw(created_on updated_on)) {
            if ($class->has_column($col) && !defined $args->{$col}) {
                $args->{$col} = $class->now . '';
            }
        }
```

```
    }
);
```

- create メソッドで INSERT するときに自動的に created_on を補う
- create メソッドに渡された引数が $args に入ってます
- 親クラスに書いておけばすべての MoCo::*** で有効

トリガのフックポイント

- before_create ($class, $args) ※1
- after_create ($class, $self)
- before_update ($class, $self, $args) ※2
- after_update ($class, $self)
- before_delete ($class, $self)

before*create, b*efore*u*pdate だけ引数が異なるので注意

- トリガは (使用するなら) デフォルト値を埋めるくらいの用途に留めましょう
    - トリガの中でさらに DB 操作を行うようになると収集がつかなくなる

utf8_columns

- 特定カラムを文字列としてアクセス
- inflate_column と同じようなことをやっている

```
package Intern::Bookmark::MoCo::Bookmark;

__PACKAGE__->utf8_columns(qw(comment));
```

- Perl の文字列 (的なもの) にはバイト列 (byte string) と utf8 文字列 (utf8 character string) の2種類があります (ちょっと面倒です)
    - 外界との入出力はつねにバイト列、プログラムの内部では文字列
    - 文字化けとか "Wide character ..." の警告を見かけたら自分がどちらを扱っているかに注意しましょう

ロジックの実装

- ここが一番楽しいところですね！

```
package Intern::Bookmark::MoCo::User;

sub add_bookmark {
    my ($self, %args) = @_;

    my $url = $args{url} or croak q(add_bookmark: parameter 'url' required);

    my $entry = moco('Entry')->find(url => $url);
    if (not $entry) {
        $entry = moco('Entry')->create(url => $url);
        $entry->update_title; # ここでページのタイトルを取得しにいく (HTTP)
    }

    if (my $bookmark = $self->bookmark_on_entry($entry)) {
        $bookmark->comment($args{comment});
        return $bookmark;
```

```
    } else {
        return moco('Bookmark')->create(
            user_id  => $self->id,
            entry_id => $entry->id,
            comment  => $args{comment},
        );
    }
}
```

- croak: use [cpan:Carp] すると使えます
    - die と似てるけど呼び出し元で死ぬ

開発のお供に

- [cpan:Devel::KYTProf] を使うのがオススメ
    - use するだけ

```
 croquis.local% perl -MDevel::KYTProf ./bookmark.pl add http://www.hatena.ne.j
p/ はてー
    0.251 ms  [DBI::st]  SELECT * FROM user WHERE ( name = ? ) LIMIT 1 (bind:
motemen) (1 rows)  | DBIx::MoCo::DataBase:344
    0.207 ms  [DBI::st]  SELECT * FROM entry WHERE ( url = ? ) LIMIT 1 (bind:
http://www.hatena.ne.jp/) (0 rows)  | DBIx::MoCo::DataBase:344
   21.194 ms  [DBI::st]  DESCRIBE `entry` '%'  (5 rows)  | DBD::mysql::db:426
    0.228 ms  [DBI::st]  INSERT INTO entry ( created_on, updated_on, url) VALU
ES ( ?, ?, ? ) (bind: 2011-08-16 02:42:44, 2011-08-16 02:42:44, http://www.hat
ena.ne.jp/) (1 rows)  | DBIx::MoCo::DataBase:344
    2.012 ms  [DBI::st]  DESCRIBE `entry` '%'  (5 rows)  | DBD::mysql::db:426
    0.290 ms  [DBI::st]  SHOW INDEX FROM entry  (2 rows)  | DBIx::MoCo::DataBa
se:344
 5435.445 ms  [LWP::UserAgent]  GET http://www.hatena.ne.jp/  | Intern::Bookma
rk::MoCo::Entry:42
    0.240 ms  [DBI::st]  UPDATE entry SET title = ? WHERE ( id = ? ) (bind: はて
な, 6) (1 rows)  | DBIx::MoCo::DataBase:344
    0.192 ms  [DBI::st]  SELECT * FROM bookmark WHERE ( ( entry_id = ? AND use
r_id = ? ) ) LIMIT 1 (bind: 6, 1) (0 rows)  | DBIx::MoCo::DataBase:344
    2.800 ms  [DBI::st]  DESCRIBE `bookmark` '%'  (6 rows)  | DBD::mysql::db:4
26
    0.180 ms  [DBI::st]  INSERT INTO bookmark ( comment, created_on, entry_id,
 updated_on, user_id) VALUES ( ?, ?, ?, ?, ? ) (bind: はてー, 2011-08-16 02:42:
49, 6, 2011-08-16 02:42:49, 1) (1 rows)  | DBIx::MoCo::DataBase:344
    0.848 ms  [DBI::st]  DESCRIBE `bookmark` '%'  (6 rows)  | DBD::mysql::db:4
26
    0.390 ms  [DBI::st]  SHOW INDEX FROM bookmark  (5 rows)  | DBIx::MoCo::Dat
aBase:344
    0.212 ms  [DBI::st]  SELECT * FROM entry WHERE ( id = ? ) LIMIT 1 (bind: 6
) (1 rows)  | DBIx::MoCo::DataBase:344
bookmarked [6] はてな <http://www.hatena.ne.jp/>
  @2011-08-16 はてー
```

- または環境変数 `MOCO_DEBUG` を真にする
- それから… [cpan:Data::Dumper] いいです

```
 use Data::Dumper;
my $x = { foo => [1,2,3] };
print Dumper($x);
# $VAR1 = {
#            'foo' => [
#                       1,
#                       2,
#                       3
#                     ]
#          };
```

- repl もいいですね ([cpan:Eval::WithLexicals], [cpan:Devel::REPL])

テスト

- 書いたプログラムが正しいかどう確かめるか？
  - 小規模なら実際に動かしてみるのでもやっていける
  - = 大規模だとムリ
  - コードの変更の影響を完全に把握するのは無理
  - 意図せず別の機能に不具合を引き起こしていないか
  - 他人のコードの意図は把握できない
  - 昔の自分も他人です (だいたい一晩から)
- 今回は単体テストを書きましょう

テストすべきこと

- 正しい条件で正しく動くこと
- おかしな条件で正しく動くこと (エラーを吐くなど)
- 境界条件で正しく動くこと

テスト例

```
 package t::Intern::Bookmark::MoCo::User;
use strict;
use warnings;
use base 'Test::Class';
use Test::More;
use t::Bookmark;

...

sub add_bookmark : Test(5) {
    # テストしたいメソッドの登場人物を用意
    ok my $user = Intern::Bookmark::MoCo::User->create(name => 'test_user_1'),
 'create user';

    # メソッド実行前の状態をテスト
    is_deeply $user->bookmarks->to_a, [];
```

```
    # テストしたいメソッドを実行
    my $bookmark = $user->add_bookmark(url => 'http://www.example.com/', comme
nt => 'nice page');

    # メソッドの返り値をテスト
    isa_ok $bookmark, 'Intern::Bookmark::MoCo::Bookmark';
    is $bookmark->entry->url, 'http://www.example.com/', '$bookmark url';

    # メソッド実行後の状態をテスト
    is_deeply
        $user->bookmarks->map(sub { $_->entry->url })->to_a,
        [ 'http://www.example.com/' ],
        '$user->bookmarks';
}

...

__PACKAGE__->runtests;
```

- [cpan:Test::Class] という JUnit ライクなテストフレームワークを使っています

テスト用モジュールを書いておくと便利

```
 package t::Bookmark;
use strict;
use warnings;
use lib 'lib', glob 'modules/*/lib';
use Intern::Bookmark::DataBase;

Intern::Bookmark::DataBase->dsn('dbi:mysql:dbname=intern_bookmark_test');

$Intern::Bookmark::MoCo::Entry::NO_HTTP = 1;

sub truncate_db {
    Intern::Bookmark::DataBase->execute("TRUNCATE TABLE $_") for qw(user entry
 bookmark);
}

1;
```

- すべてのテスト用スクリプトから use する
- 本番とは別のテスト用データベースの dsn を設定する
- HTTP アクセスしないフラグを立てる、等々

心構え: テストは安心して実行できるように

- 本番の DB にアクセスしないようにする
    - テスト専用の DB を用意して、テストでは必ずそちらを使うようにする
    - [cpan:DBIx::RewriteDSN] を使うのもよいです
- 外部との通信を発生させない

    - テストの高速化にもつながります

ディレクトリ構成

```
 - bookmark.pl
- lib       - Bookmark  - MoCo.pm
                        - DataBase.pm
                        - MoCo          - User.pm
                                        - Entry.pm
                                        - Bookmark.pm

- t         - Bookmark.pm
            - Intern-Bookmark-MoCo.t
            - Intern-Bookmark-MoCo-User.t

- modules   - DBIx-MoCo
```

以上

- かけ足で説明してきましたが、全容はもっと深いので、ソースや pod (perldoc) を読んでみてください
- 試行錯誤もいいですが人に訊くのが一番楽!!!

課題

- コマンドラインインターフェースで日記を書けるツール diary.pl を作成してください (必須)
- diary.pl に機能を追加してください (記事のカテゴリ機能など)

基本機能

- 記事の追加
- 記事の一覧表示
- 記事の編集
- 記事の削除
- マルチユーザー (ただし今回はシングルユーザーでしか利用しない)

テーブル設計

とりあえず 2 テーブルでいってみましょうか

- user
- entry

実行例

```
 % ./diary.pl add タイトル   # 記事追加
% ./diary.pl list           # 記事を一覧
% ./diary.pl edit 記事ID    # 記事を編集
% ./diary.pl delete 記事ID # 記事を削除
```

スキーマ設計 (例)

- 望むように独自のスキーマを設計してよいです
- プライマリキーには AUTO_INCREMENT を指定しておくと便利
- データベース名は intern*diary*ユーザ名 としてください

```
CREATE TABLE user (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARBINARY(32) NOT NULL,
    created DATETIME NOT NULL,
    PRIMARY KEY (id),
    UNIQUE KEY (name)
);

CREATE TABLE entry (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    user_id INT UNSIGNED NOT NULL,
    title VARBINARY(255) NOT NULL,
    body BLOB NOT NULL,
    created DATETIME NOT NULL,
    updated DATETIME NOT NULL,
    PRIMARY KEY (id),
    KEY (user_id, created)
);
```

オプション課題 独自機能

- アプリケーションに独自の機能を追加してみてください
  - 記事のカテゴリ分け機能
  - 検索
  - などなど

評価基準

- 基本機能 5 点
  - 記事の追加・一覧 3 点
  - 記事の編集・削除 2 点
  - 来週の課題につながりますので必須で！
- 追加機能 2 点
- 設計 2 点
- テスト 1 点

諸注意

- コミット先こんな感じで

```
Intern-Diary
           /lib
           /t
```

- DBIx-MoCo を submodule として追加しましょう
  - "modules/DBIx-MoCo/lib" を use lib (@INC に追加) する必要があります

```
git submodule add https://github.com/hatena/DBIx-MoCo.git Intern-Diary/module
s/DBIx-MoCo
```

- DB 名
  - 評価・検証する際にバッティングしないように database 名を intern*diary*{はてな ID} としてください。

hitode909 さんだったら、

```
CREATE DATABASE intern_diary_hitode909;
```

mysqldump お願い

評価のため mysqldump もお願いします。

```
% mysqldump -uroot -Q tablename > tablename.sql
```

保存先は mysqldump ディレクトリに hitode909 さんだったら、

```
 % mkdir mysqldump
% mysqldump -uroot -Q intern_diary_hitode909 > mysqldump/intern_diary_hitode90
9.sql
```

これも commit, push してください。

ヒント(& BK)

今日書くコードは明日以降も利用します！

- CLI 以外の利用も見据えた設計を
- アプリケーションに必要な機能は MoCo クラス内に書きましょう

コマンドラインに関すること

- `@ARGV` 変数
  - ./diary.pl hoge fuga として起動すると `@ARGV = ('hoge', 'fuga')` となります
- コマンドライン引数をパーズするには [cpan:Getopt::Long]
- 標準入力からの読み取り

```
my $data = join "\n", <STDIN>;
```

ご清聴ありがとうございました

- 分からないことはメンターか隣りのインターンに尋ねましょう！
  - 人気の質問に関してはあとでまとめて補足をするかもしれないのでどんどん訊いてください
- 今日は歓迎会です

補足編

SQL

```
SELECT * FROM table_name
    WHERE column1 = 'value1' AND column2 < 40
    ORDER BY column3 ASC
    LIMIT 10;
```

FROM で指定されたテーブルから、WHERE で指定された条件のレコードを、 ORDER BY の順序で、 LIMIT の個数だけ取得する。

- WHERE ... OR ...

- WHERE column1 IN ('value1', 'value2', 'value3')
- ORDER BY column2 ASC -- 昇順
- ORDER BY column2 DESC -- 降順

```
INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');
```

```
UPDATE table_name SET column1 = 'value1' WHERE column2 = 'value2';
```

```
DELETE table_name WHERE column1 = 'value1';
```

DELETE table_name; だけだと大変なことに!

```
CREATE TABLE table_name (
    column1 INT UNSIGNED NOT NULL, -- 整数
    column2 VARCHAR(127) NOT NULL, -- 可変長文字列
    column3 TEXT NOT NULL, -- もっと長い文字列
    column4 TIMESTAMP DEFAULT 0, -- 日時
    PRIMARY KEY (column1),
    KEY (column2),
    KEY (column4)
) DEFAULT CHRASET=BINARY;
```

テーブルを作る。

- PRIMARY KEY (主キー) には他のどのレコードとも異なる ID が入るカラムを指定しておく
- KEY には検索に使うカラム (WHERE で使うカラム) を指定しておく
    - 検索高速化のためのインデックスが作られる

SQL::Abstract

詳しくは [cpan:SQL::Abstract] をみて

```
  # WHERE column = 'value'
  where => {
      column => 'value',
  },
```

```
  # WHERE column < 40
  where => {
      column => {'<', 40},
  },
```

```
  # WHERE column != 'value'
  where => {
      column => {'!=, 'value'},
  },
```

```
  # WHERE column IS NULL
```

```
  where => {
      column => undef,
  },
```

```
   # WHERE column1 = 'value1' AND column2 IN ('a', 'b', 'c')
  where => {
      column1 => 'value1',
      column2 => {-in => ['a', 'b', 'c']},
  },
```

IN で指定する値が空でも SQL::Abstract 的には問題ないけど MySQL 的には構文エラーなので注意。

```
   # WHERE column IN () -- 構文エラー
  where => {
      column => {-in => []},
  },
```

Placeholder

SQL 文に「?」(placeholder) を埋め込むとその部分に入る値を別に渡せる。

```
my $sql = 'select album.id, name, title, released_on '
        . 'from artist inner join albums on artist.id = album.artist_id '
        . 'where artist.name = ?';

my $result = $dbh->selectall_hashref($sql, 'id', undef, $artist_name);
```

「?」に $artist_name が代入される。 MoCo SQL::Abstract も実際は placeholder を使っている。

Placeholder を使うと、 * SQL インジェクションの心配がなくなる * MySQL が SQL 文の構文解析結果をキャッシュしやすくなる * 「SELECT * FROM artist WHERE name = '初音ミク'」と「SELECT * FROM artist WHERE name = '鏡音リン'」のどちらも「SELECT * FROM artist WHERE name = ?」になる ... というメリットがある。

ORM

- オブジェクト生成のオーバーヘッド
  - Perl のオブジェクト生成 (bless) は結構コストが高い
  - 数千個のオブジェクトを生成するよりSQL 1 回で済ませた方がいいかも
- 無駄なSQL
  - オブジェクト指向的に綺麗な書き方をしても、必ずしも良い SQL にはならない
  - むしろパフォーマンス悪化の要因にすらなる
  - 一度にまとめられる問い合わせがばらばらに発行されるとか
  - ORマッパー内部である程度は最適化できるけど限界がある
  - → オブジェクト指向インターフェイスでSQLを隠蔽しているのに、どんなSQLが発行されているか気にしないといけないという矛盾

```
sub has_a_method {
    my $self = shift;
    return $related_class->find(id => $self->has_a_id);
}

# こうすると @$list の要素数分 "SELECT * FROM related_table WHERE id = ?" が実行され
```

る

```
my @related;
for my $obj (@$list) {
    pus @related, $obj->has_a_method;
}

# こっちなら "SELECT * FROM related_table WHERE id IN (?, ?, ...)" 1回で済む
my $related = $related_class->where(
    where => {
        id => {-in => [map { $_->has_a_id } @$list]},
    },
);
```

byte string & utf8 string

perldoc utf8, perlunicode

- Perl の文字列にはバイト列 (byte string) と文字列 (utf8 character string) の2種類がある
- バイト列は 0x00-0xFF のバイトの並びを表す
- 文字列は U 0000〜U-FFFFFFFF (32ビット環境の場合) の文字の並びを表す
    - 一般的な計算機では UTF-8 で符号化されているので utf8 文字列と呼ばれる
- バイト列と区別するとき、文字列のことを「(utf8) フラグが立っている」という

```
# 何もしないとバイト列になる（ファイルが UTF-8 なら、UTF-8 を表すバイト列になる）
$bytes = 'あいうえお';
warn length $bytes; # 15

# use utf8; プラグマの効力が及ぶ範囲では文字列になる（ファイルは UTF-8 にしておく）
use utf8;
$chars = 'あいうえお';
warn length $chars; # 5
```

byte/character convertion

[cpan:Encode]

- 入出力は基本的にバイト列になっている
    - ファイル、DB、HTTP、...
- Perl で文字の列を扱いたいときは原則として utf8 文字列を使うべき
    - → 入力はできるだけ早い段階で文字列に変換し、出力はできるだけ遅い段階でバイト列に変換する
- バイト列と文字列の相互変換には Encode モジュールを使う
    - encode/decode はどちらがどちらか覚えにくいけど、
    - 人間が読める文字列を機械が読めるバイト列にするのが符号化 (encode)
    - 機械が読めるバイト列を人間が読める文字列にするのが復号 (decode)

```
use Encode;

$bytes = encode 'utf8', $chars& # 文字列を符号化してバイト列に
$chars = decode 'utf8', $bytes& # バイト列を復号して文字列に
```

DateTime

[cpan:DateTime], [cpan:DateTime::Format::MySQL]

- Perl で日時を表すときは DateTime がよく使われる

```
 $dt = DateTime->now(time_zone => 'UTC');
$dt = DateTime->new(year => 2010, month => 8, day => 3, hour => 10, minute =>
0, second => 0, time_zone => 'UTC');

$dt->add(days => 3);
$dt->subtract(hours => 4);

warn $dt->ymd('-');
warn $dt->hms(':');
```

- データベースでは UTC (協定世界時) で保存し、表示するときに必要ならタイムゾーン変換するのが好ましい

```
 warn $dt->time_zone;
$dt->set_time_zone('Asia/Tokyo');
```

- MySQL 形式との変換には DateTime::Format::MySQL を使う

```
 use DateTime::Format::MySQL;

my $dt = DateTime::Format::MySQL->parse_datetime('2010-01-01 02:02:02');
warn DateTime::Format::MySQL->format_datetime($dt);
```

- なお、本当の Perl ネイティブの時刻形式は time 関数の形式
    - ほとんどの環境では Unix の time_t = 1970年1月1日0時0分0秒 (UTC) からの秒数

```
 $time = time;
warn $time;

use DateTime;
$dt = DateTime->from_epoch(epoch => $time);
warn $dt->epoch;
```