MovableType プラグインを作る

2006 年 11 月 18 日
野田純生(junnama@alfasado.jp)
アルファサード有限会社

このドキュメント及び KOF2006 での「MovableType 作成入門」の内容については、主催者が独自に調査、実装した上で理解したことをベースとしたものであり、SixApart 社及び MovableType の公式なドキュメントではないことについて予めご了承ください。

MovableType は Blog ソフトです。ところがこの Blog ソフトを CMS のように使いたいというケースが増えています。
Blog は SEO に良いと言われていることや (←嘘です) 、導入の手軽さが受けているのだと思いますが、RSS を自動的に生成してくれることや、ブラウザベースでの管理ができること等、Blog を採用するメリットは確かにあると思います。
ここでは、実際にコードを書きながら MovableType Perl API について、またプラグイン開発について理解して行くことにしたいと思います。

MovableType API を利用した CGI スクリプト

まず、プラグイン作成の前に、以下の CGI スクリプトを見て下さい。

```
01.#!/usr/bin/perl -w
02.use strict;
03.
04.print "content-type: text/html\n\n";
05.
06.print <<'HTML';
07.<?xml version="1.0" encoding="UTF-8"?>
08.<html>
09.<head>
10.<body>
11.HTML
12.
13.use lib qw (/Applications/MAMP/htdocs/mt/lib/);
14.require MT;
15.
16.my $mt = MT->new(Config => 'mt-config.cgi');
17.my $iter = MT::Blog->load_iter(undef, { sort => 'id' });
18.
19.while (my $blog =$iter->()) {
20.	printf("<h1>Blog%s: %s</h1>\n", $blog->id, $blog->name);
21.	my @entries = MT::Entry->load({ blog_id => $blog->id }, {
22.	sort => 'modified_on',
23.	direction => 'descend',
24.	});
25.	for my $entry (@entries) {
26.		if ($entry->status == 2){
27.			printf("%s,%s,%s<br />\n",$entry->title,$entry->permalink,$entry->modified_on);
28.		}
29.	}
30.}
```

これは、MovableType に登録されているすべての Blog のエントリのうち、ステータスが「公開」のものを、Blog 毎且つ更新日順に表示する CGI スクリプトです。MovableType は Perl(一部は PHP)で書かれていますから、言語は Perl になります。

たった 30 行のスクリプトですが、「Blog を横断してエントリを抽出して表示する」処理をたった 30 行のスクリプトで実現しています。

MT に関する処理は 13〜14 行目以降です。13 行目、Perl の lib プラグマで Perl ライブラリのパスを指定します。
14 行目でライブラリをロードしています。

16 行目：MT API を使って Config => 'mt-config.cgi' として MT の設定ファイルを Config の引数にしてインスタンスを作成します。

17 行目。詳細は「Movable Type オブジェクト・リファレンス - MT::Object」を参照してください。
http://www.sixapart.jp/movabletype/manual/object_reference/archives/mt_object.html

『load メソッドを使って、データストアからオブジェクトを読み込むことができます。load メソッドは間違いなく最も複雑なメソッドです。というのは、オブジェクトの読み込みには、ID から、カラム値から、他のタイプのオブジェクトとの連結から、といったさまざまな方法があるからです。』

『さらに、オブジェクトの読み込みには、一括して配列に読み込む方法（load）と、反復子を使って一つずつオブジェクトを読み込む方法（load_iter）があります。load の一般的な書式は次のとおりです。』

ここでは、load_iter メソッドを使っています。

19 行目〜30 行目までのループで、Blog をひとつずつ参照して処理を行っています。
20 行目。Blog の ID と名前を見出しとして出力します。
21 行目。今度は load メソッドを使ってエントリを全て取り出して配列におさめています。
22〜23 行目で、更新日順に並べ変えています。
25 行目からのループで、エントリのタイトル、パーマリンク、更新日を取り出しています。
26 行目の if 文は、エントリのステータスを参照しています（1 が下書き、2 が公開）。

MovableType オブジェクト・リファレンスを活用する

Blog、Entry のオブジェクトリファレンスは以下のページを参照ください(※)。

Movable Type オブジェクト・リファレンス - MT::Blog
http://www.sixapart.jp/movabletype/manual/object_reference/archives/mt_blog.html

Movable Type オブジェクト・リファレンス - MT::Entry
http://www.sixapart.jp/movabletype/manual/object_reference/archives/mt_entry.html

※Movable Type オブジェクト・リファレンスを参照することで、多くのことが理解できます。
　http://www.sixapart.jp/movabletype/manual/object_reference/
　ただし、全てのリファレンス、メソッドについて記述されているわけではありません。
　例えば、MT::Entry の項では $entry->keyword について記載されていませんし、MT3.3 から採用された「タグ」をエントリから参照する手段については書かれていません。

　実際には$entry->tags でエントリに紐づいているタグの数が、$entry->get_tags でタグが配列として入るようです。
　つまり、「($entry->get_tags)[0]」とすれば一つ目のタグが得られます。

上記の MT::Entry のリファレンスにもありますが、Blog やエントリの値を参照するだけでなくエントリを作成することもできます。

```
use MT::Entry;
my $entry = MT::Entry->new;
$entry->blog_id(1);
$entry->status(2);
$entry->author_id(1);
$entry->title('KOF レポート!');
$entry->text('本日は大盛況?');
$entry->save or die $entry->errstr;
```

MS Excel 等のスプレッドシートアプリケーションでマスターデータを作り、書き出した CSV ファイルを読み込んで大量のページを Blog のエントリとして一気に登録する..ようなこともデータベースに直接アクセスすることなく出来てしまうことがわかると思います。

MovableType プラグインの開発

MT のプラグインには、テンプレート・タグ（変数タグ、コンテナ・タグ、条件タグ等）を新たに使えるようにする「テンプレート・タグ プラグイン」や、入力されたテキストに対して操作を行う「フィルター プラグイン」、管理画面の機能を拡張する「Transformer プラグイン」等があります。

今日は、「フィルター プラグイン」「Transformer プラグイン」を取り上げます。

プラグインのひな形

以下の 1～20 行目は、予めプラグインのテンプレートとしてひな形を作っておくと良いでしょう。
このプラグインは、何の動作もしませんが、mt/plugins フォルダに入れると MT のプラグインの管理画面に現れます。

```
01.package MT::Plugin::KOFPlugin;
02.use strict;
03.use MT;
04.
05.
06.@MT::Plugin::KOFPlugin::ISA = qw(MT::Plugin);
07.
08.my $plugin = new MT::Plugin::KOFPlugin({
09.             name => 'KOFPlugin',
10.             version => "0.1.0",
11.             description => "KOF 用初めてのプラグイン",
12.             author_name => '野田 純生',
13.             author_link => 'http://alfasado.net/',
14.             settings => new MT::PluginSettings([
15.                             ['KOFPlugin_setting1'],
16.             ]),
17.#            config_template => 'KOFPlugin_blog.tmpl',
18.#            system_config_template => 'KOFPlugin_system.tmpl',
20.});
```

17,18 行目はコメントアウトしていますが、mt/plugin/myplugin/ というディテクトリを作成してプラグインを置き、
mt/plugin/myplugin/templ/KOFPlugin_blog.tmpl と mt/plugin/myplugin/templ/KOFPlugin_system.tmpl というファイルを置くことによって、プラグイン管理画面に「設定を表示」リンクが現れ、フォーム等を使って設定項目を保存することができるようになります。
この管理画面から登録した設定へアクセスするための変数が 15 行目の「['KOFPlugin_setting1']」です。

※今日はこの部分は使いません。

2 つのラジオボタンで、0 と 1 の値をセットするためのテンプレートは以下のように書きます。

```
<input type="radio" name="KOFPlugin_setting1" value="0" <TMPL_UNLESS NAME=KOFPLUGIN_SETTING1>checked</TMPL_UNLESS> />
<input type="radio" name="KOFPlugin_setting1" value="1" <TMPL_UNLESS NAME=KOFPLUGIN_SETTING1>checked</TMPL_UNLESS> />
```

設定した情報へは、以下のようにして取り出すことができます（この場合は system_config_template から設定したシステム全体の設定となります）。

```
my $KOFPlugin_setting1 = $plugin->get_config_value('KOFPlugin_setting1', 'system');
```

フィルタープラグインの開発

ではまず最初に「フィルター プラグイン」を作成してみます。add_global_filter メソッドを使います。

MT::Template::Context->add_global_filter(KOFPlugin => \&_KOFPlugin);

今回は、機種依存文字の一例として、丸付き数字を変換するプラグインを作ることにします。

```
01.our @search_kisyuizon = split(/\,/,'①,②,③,④,⑤,⑥,⑦,⑧,⑨,⑩');
02.our @replace_kisyuizon = split(/\,/,'(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)');
03.
04.sub _KOFPlugin {
05.	my($text, $arg, $ctx) = @_;
06.	my $end = @search_kisyuizon;
07.
08.	for ( my $i = 0; $i < $end; $i++ ) {
09.		$text =~ s/$search_kisyuizon[$i]/$replace_kisyuizon[$i]/eg;
10.	}
11.	return $text;
12.}
```

テンプレートに以下のように記述することでこのフィルターが有効になります。

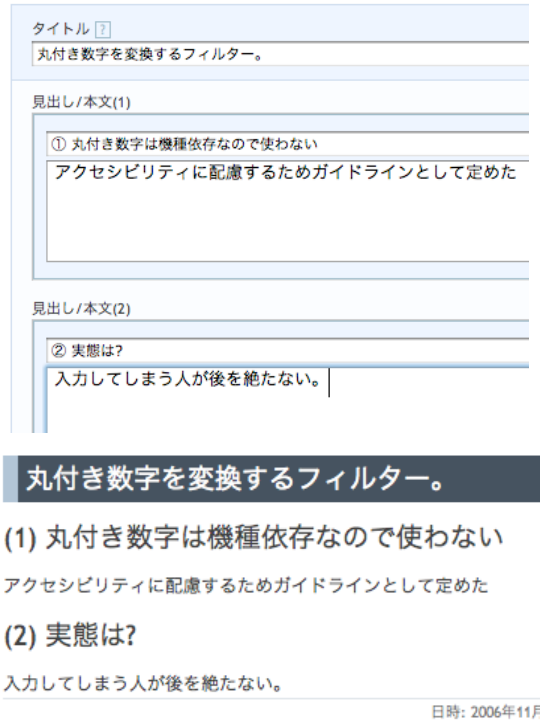
```
<$MTEntryBody KOFPlugin="1"$>
```

この場合、5 行目の「$arg」に値「1」が渡されますが、今回はこの値は使っていません。

$text には <$MTEntryBody$> に置き換わる文字列が渡されます。

フィルタープラグインの場合、$text に対して処理を行い、返り値に処理結果を渡してやればそれでおしまいです。

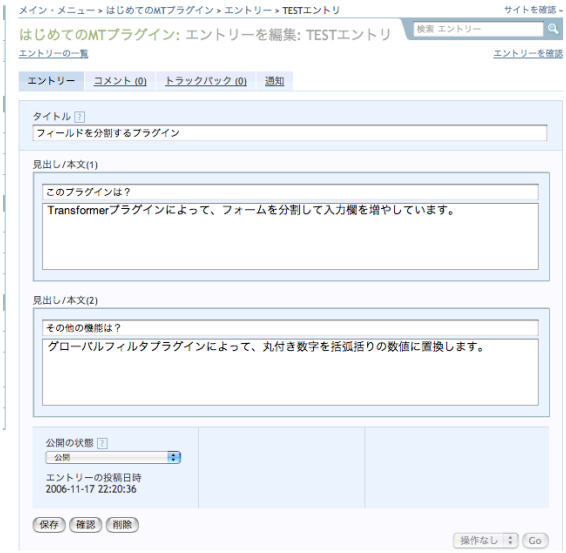
01,02 行目で登録した丸付き数字と置換文字列のリストを 08 行目から 10 行目のループ内で置換し、11 行目で返り値を return しています。

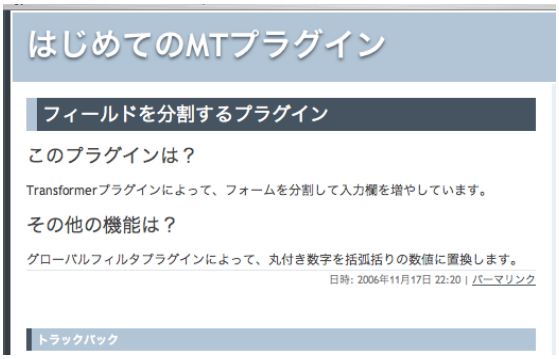


Transformer プラグインで管理画面をカスタマイズする

続いて「Transformer プラグイン」を開発してみましょう。

サンプルとして、エントリ登録画面の text フィールド（textarea）を見出し、本文 × ２組のフィールド（合計４つ）に分割して入力を支援するものを作ってみましょう。





エントリ登録画面のテンプレートは mt/tmpl/cms/edit_entry.tmpl です。
Transformer プラグインにおいては mt/tmpl/cms/ フォルダ以下のテンプレートを書き換える処理が多くなります。

1.edit_entry テンプレートを置換して text フィールド（textarea）を削除して、代わりに４つのフィールドと置換する

MT->add_callback('MT::App::CMS::AppTemplateSource.edit_entry', 9, $plugin, \&_app_template_source_entry);

2.4 つのフォールドから受け取った値は、保存前に HTML のタグを付けて一つに連結し、本来保存されるべき text の値としてセットする

MT->add_callback('CMSPreSave_entry', 9, $plugin,\&_cms_presave_entry);

3.再び 1.でフォームが表示される時には、逆に text を分割して４つのフォームにセットする

```
01.MT->add_callback('MT::App::CMS::AppTemplateSource.edit_entry', 9, $plugin, \&_app_template_source_entry);
02.
03.sub _app_template_source_entry {
04.        my ($eh, $app, $tmpl_ref) = @_;
05.        my $query = $app->param;
06.        my $entry_id = $query->param('id');
07.        my $entry = MT::Entry->load({ id => $entry_id });
```

```
08.		my $old = &entry_templ_old;
09.		$old = quotemeta($old);
10.		my $new = &entry_templ_new;
11.		$$tmpl_ref =~ s/$old/$new/;
12.1;
13.}
14.
15.sub entry_templ_old {
16.		return <<'HTML';
17. （ここに検索対象の文字列を貼付ける）
18.HTML
19.}
20.
21.sub entry_templ_new {
22.		return <<'HTML';
23. （ここに置換対象の文字列を貼付ける）
24.HTML
25.}
```

1 行目 AppTemplateSource.edit_entry で mt/tmpl/cms/edit_entry.tmpl が読み出された時にサブルーチン「_app_template_source_entry」で置換を行います。

3 行目から 13 行目までが実際の処理です。3 つめの $tmpl_ref がテンプレートの文字列です。参照渡しなので 11 行目のように「$$tmpl_ref =~ s/$old/$new/;」とすることでテンプレートの文字列 $old が $new に置換されます。
15 行目〜はテンプレートの中の文字列及び置換文字列を返すだけのサブルーチンです。

ここで改造したエントリ登録画面から「保存」をクリックすると、本文（text）フィールドがありませんから、そのままでは本文は表示されません。そこで、エントリ保存前に実行される「CMSPreSave_entry」でフィールドを連結して「本文」をセットします。

```
01.MT->add_callback('CMSPreSave_entry', 9, $plugin,\&_cms_presave_entry);
02.
03.sub _cms_presave_entry {
04.		my ($eh, $app, $obj, $original) = @_;
05.
06.		my $query = $app->param;
07.		my $fld01 = $query->param('fld01');
08.		my $fld02 = $query->param('fld02');
09.		my $fld03 = $query->param('fld03');
10.		my $fld04 = $query->param('fld04');
11.
12.		$fld01 =~ s/(<!--split-->)//sg;
13.		$fld02 =~ s/(<!--split-->)//sg;
14.		$fld03 =~ s/(<!--split-->)//sg;
15.		$fld04 =~ s/(<!--split-->)//sg;
16.
17.		my $sep = '<!--split-->';
18.		$obj->text('<h2>'.$fld01.'</h2>'.$sep.$fld02.$sep.'<h2>'.$fld03.'</h2>'.$sep.$fld04);
19.1;
20}
```

$obj, $original にはそれぞれ保存されるオブジェクト、保存前のオブジェクトが渡されます。
新規投稿の際には$original は空になります。

例えば、あるユーザーにはエントリのタイトルの変更を許可しない場合、以下のようにすればいくらタイトルフィールドの値を変更して「保存」してもタイトルが変わらないことになります。

$obj->title($original->title);

※フィールドはあるのに変更されないってのは、嫌がらせに過ぎませんが...

6 行目から 10 行目で、新規に作成した４つのフィールドの name を指定してポストされたデータから値を取得します。
ここでは、文字列「<!–split–>」を値のセパレータとして使いますので、「<!–split–>」がフィールドに入力されていた場合に備えて予め置換しています（書き方が冗長になってますね）。

text への値のセットは「$obj->text(セットする文字列)」です。

さて、これで改造したフォームから値を受け取り、<h2>タグを付けて連結し、本文を保存できるようになりました。

このままでは投稿はできますが、修正ができません。エントリ投稿後に表示される画面に（当然ですが）本文が表示されません。

そこで、_app_template_source_entry サブルーチンに、本文を逆に分割してフィールドにセットする処理を入れます。

具体的には、エントリを load した後で 本文（text）を取得し（$entry->text)、加えた<h2>タグを削除してからセパレータで分割し、４つのフォームにセットします。

```
if ($entry_id ne "") {
		my $entry = MT::Entry->load({ id => $entry_id });
		my $text = $entry->text;
		$text =~ s/<h2>//g;
		$text =~ s/<\/h2>//g;
		my @fields = split ('<!–split–>', $text);
...
```

さらに 10 行目の「my $new = &entry_templ_new;」を以下のように修正し、entry_templ_new サブルーチンもあわせて修正します。

my $new = &entry_templ_new($fields[0],$fields[1],$fields[2],$fields[3]);

サブルーチン entry_templ_new に分割した値を渡し、フォームの初期値としてセットしてページを生成します。

```
sub entry_templ_new {
		my ($fld01, $fld02, $fld03, $fld04) = @_;
		return <<"HTML";
 (略)
<p><input name="fld01" id="fld01" size="72" style="width:570px;border:gray" value="$fld01" onchange="setDirty()" /></p>
<p><textarea name="fld02" name="fld04" cols="72" rows="5" style="width:570px;border:gray" onchange="setDirty()">$fld02</textarea></p>
 (略)
HTML
}
```

別添のソースコードを参考にしてください。