前言

本教程是在 CPAN 上 Catalyst 的文档基础上编写的.

阅读本教程前你需要具备的知识:

1. 使用过 perl 编写 cgi 程序
2. 使用过 perl 的 DBI 编写 perl 程序进行数据库操作

编写者: 小生菜菜

联系邮箱: laomoi@163.com

目录

第 1 章　　Catalyst 的简介

使用 perl 进行 web 开发,传统的做法是写*.cgi 文件,然后把这些 cgi 文件配到 web server 里面,当 web server 收到客户端的 request 时,就调用对应的 cgi 进行处理. 用这种方法进行 web 开发, cgi 文件之间相对比较独立,代码里面可能含有大量的重复代码,整个系统的结构也是比较松散的, 可扩展性也不强.

Catalyst 是一套用来 web 开发的框架,基于 Catalyst 开发的 web 系统,具有比较强的可扩展性.

第1节 搭建 Catalyst 开发环境

1 一般方法:
在 linux 系统上安装 Catalyst 只需要使用 CPAN 安装即可(推荐)

```
cpan -i Catalyst
cpan -i Catalyst::Devel(用来运行 Catalyst 项目的机器不需要安装该模块,如果需要使用 Catalyst 进行开发则需要安装该模块)
```

在 windows 上安装可以使用 ppm,但是过程比较繁琐,不建议.

2 安装脚本
Matt Trout 写了一个安装脚本: http://www.shadowcatsystems.co.uk/static/cat-install
只需要从该链接下载脚本,然后 perl cat-install 即可按顺序安装相关模块,可用于 windows 跟 linux,如果用于 windows 的话,必须保证你的机器上有 c 编译器和 make 工具.

3 模块打包方法(推荐)
Chris Laco 把 Catalyst 相关的模块打成了一个包:http://handelframework.com/downloads/CatInABox.tar.gz
把该包下载下来不需要安装即可马上使用.可用于 windows 跟 linux 系统.

第2节 创建第一个 Catalyst 项目

给我们第一个项目起名为 X, 那么如下使用 Catalyst 创建我们的项目:

```
perl catalyst.pl X
```

如果这条命令出错了,那么可能是你的 Catalyst 没有安装好,请参考第一节把 Catalyst 装好.

键入命令后, catalyst.pl 会帮你创建如下文件:

```
created "X"
created "X\script"
created "X\lib"
created "X\root"
created "X\root\static"
created "X\root\static\images"
created "X\t"
created "X\lib\X"
created "X\lib\X\Model"
created "X\lib\X\View"
created "X\lib\X\Controller"
created "X\x.yml"
created "X\lib\X.pm"
created "X\lib\X\Controller\Root.pm"
created "X/README"
created "X/Changes"
created "X\t/01app.t"
```

```
created "X\t/02pod.t"
created "X\t/03podcoverage.t"
created "X\root\static\images\catalyst_logo.png"
created "X\root\static\images\btn_120x50_built.png"
created "X\root\static\images\btn_120x50_built_shadow.png"
created "X\root\static\images\btn_120x50_powered.png"
created "X\root\static\images\btn_120x50_powered_shadow.png"
created "X\root\static\images\btn_88x31_built.png"
created "X\root\static\images\btn_88x31_built_shadow.png"
created "X\root\static\images\btn_88x31_powered.png"
created "X\root\static\images\btn_88x31_powered_shadow.png"
created "X\root\favicon.ico"
created "X/Makefile.PL"
created "X\script/x_cgi.pl"
created "X\script/x_fastcgi.pl"
created "X\script/x_server.pl"
created "X\script/x_test.pl"
created "X\script/x_create.pl"
```

我们来看一下这些目录结构以及这些文件的含义:

X/	项目文件夹
X/x.yml	项目的配置文件,Catalyst 的插件 ConfigLoader 会使用该文件
X/t	test 文件夹,放置项目的各种测试脚本
X/script	辅助脚本文件夹,放置 catalyst 为项目准备的辅助脚本,开发时才会用到

X/root	root 文件夹,一般放置项目的静态文件,模板文件等
X/lib	lib 文件夹,放置项目的各种模块
X/lib/X.pm	以项目名字命名的应用程序类,在整个项目中起驱动作用
X/lib/View	文件夹,放置各种 View 模块
X/lib/Model	文件夹,放置各种 Model 模块
X/lib/Controller	文件夹,放置各种 Controller 模块

在这里，比较重要的文件夹是 root 跟 lib 。

root 文件夹,一般放置项目的静态文件,模板文件等

lib 文件夹一进去会看到一个 X.pm,它以项目名字命名的应用程序模块，它在整个项目中起驱动作用， 你会在第二章看到它的详细说明。

lib 文件夹再深一层你会看到 3 个文件夹， Controller, Model, View, 这就是 Catalyst 的 MVC 结构。整个 WEB 项目的绝大多数代码都集中在 3 个文件夹里面.

位于 View 文件夹里面的 View 模块主要负责数据的渲染和加工,一般就是根据 Controller 传过来的数 据生成 HTML 代码, 我们常用的一种 View 模块是 Catalyst::View::TT,它的内部会使用模块 TemplateToolkit(简称 TT),如果你们使用过 HTML::Template 这样的模块就会大概理解 TT 在程序里面的作用.

位于 Model 文件夹里面的 Model 模块主要负责对数据源的访问, 一般就是负责数据库的读取.我们常用的 Model 模块是 Catalyst::Model::DBIx::Class::Schema,它的内部会使用模块 DBIx::Class::Schema(一般叫做 DBIC).

位于 Controller 文件夹里面的模块称之为 Controller, Controller 里面有一些具有特殊属性的函数(称之为 action),在 Catalyst 里面,一个客户端的请求一般最后会被 Catalyst 分派 给某个 Controller 的某个 action 进行处理.所以 Controller 主要负责的就是处理用户的请求,控制程序的逻辑.

Controller, Model, View 都属于 Catalyst 组件,Controller 是必不可少的,而其他 2 种组件是可有可无的, 因为对一个客户端请求的处理不一定要访问 DB,也不一定要渲染数据,当然,一般的 Catalyst 应用程序都具有 MVC 3 种组件.

在 Catalyst 里面除了有不同的 View 组件跟 Model 组件可供选择外，Catalyst 还具有大量的插件可供使用，可用的 Catalyst 插件列表参看：
http://search.cpan.org/~jrockway/Catalyst-Manual-5.700501/lib/Catalyst/Manual/Plugins.pod

第 2 章　　剖析应用程序类

继续我们上一章的内容，在创建了 X 这个 Catalyst 项目之后，lib 下有一个 X.pm，称为应用程序类 (application class).

我们把 X.pm 里面的注释都删掉,代码如下:

```
package X;
use strict;
use warnings;
use Catalyst::Runtime '5.70';
use Catalyst qw/-Debug ConfigLoader Static::Simple/;
our $VERSION = '0.01';
__PACKAGE__->config( name => 'X' );
__PACKAGE__->setup;
1;
```

第 4 行代码: use Catalyst::Runtime '5.70';
这行代码无重要意义

第 5 行代码: use Catalyst qw/-Debug ConfigLoader Static::Simple/;
这里实际上就是加载 Catalyst 模块，而在对 Catalyst 模块进行 import 的时候，Catalyst 会让本身变成 X 的父类,于是，X.pm 就继承于 Catalyst 了，应用程序类也因此具有 Catalyst 的所有方法.
在 import Catalyst 的时候，Catalyst 得到了这个参数列表 qw/-Debug ConfigLoader Static::Simple/,于是 Catalyst 给自己设置了一个 Debug 的标志位,同时加载 2 个插件:
Catalyst::Plugin::ConfigLoader　　: 该插件用于读取项目目录下的 x.yml 文件而生成整个项目的配置信息
Catalyst::Plugin::Static::Simple　　: 当客户端请求一些静态文件时,该插件可以用来从磁盘读取静态文件然后直接发送响应给客户端,而无需经过 Catalyst 把请求分发给某个 Controller 的某个 action 的过程.

从这里我们可以看到，如果要在项目里面添加更多的插件,只需要把插件的名字写到 use Catalyst qw/.../里面即可．插件加载进来后在后面经过 setup 之后也会变成应用程序类的父类，所以使用了某个插件后，通过应用程序类可以使用该插件的所有方法．
需要注意的是，
use Catalyst qw/A B/; 跟
use Catalyst qw/B A/;
的效果可能是不一样的,前者导致应用程序类的父类是(A, B, Catalyst),而后者导致应用程序类的父类是(B, A, Catalyst),这样的话,当调用应用程序类的某个方法时，前者是优先调用 A 的该方法,后者则是优先调用 B 的该方法．

第 7 行代码: __PACKAGE__->config(name => 'X');
config 是 Catalyst 的方法,这里可以配置项目的任何信息．
这里只是配置了一下项目的名字为 X,其实如果使用了 ConfigLoader 这个插件,而且 x.yml 里面也配置了项目的名字,那么这里的配置会被 x.yml 覆盖．

第 8 行代码: __PACKAGE__->setup;

setup 也是 Catalyst 的方法,这行代码的运行会使整个项目进行初始化:

- 初始化插件
- 加载所有的 MVC 组件
- 加载所有 Controller 里面的 action 并构造相关的映射关系．

在 setup 过后,整个 Catalyst 算是初始化完毕了,可以开始处理客户端的请求．

第 3 章　　掌握 Controller

第1节 Controller 简介

上一章介绍了 X.pm,也就是应用程序类，当我们在 apache+mod_perl 里面把 X.pm 部署进去之后，当客户端有请求过来的时候,X.pm 选择某个 Controller 里面的某个 action 来处理这个请求，这个选择的过程称为分发(dispatch)，Catalatys 项目刚建立的时候默认已经有了一个 Controller: Root.pm.
在分析 Root.pm 的代码之前,我们先尝试一下在 Root.pm 里面添加一个名字为 help 的 action,整个模块的代码如下:(红色部分是我们刚添加的代码)

```
package X::Controller::Root;
use strict;
use warnings;
use base 'Catalyst::Controller';

__PACKAGE__->config->{namespace} = '';

sub default : Private {
    my ( $self, $c ) = @_;
    $c->response->body( $c->welcome_message );
}

sub help : Local {
    my ( $self, $c ) = @_;

    $c->response->body( "What can I do for you?" );
}
```

```
sub end : ActionClass('RenderView') {}

1;
```

我们在使用 Catalyst 进行开发的时候,如果要进行测试,看看页面效果什么的,没有必要把项目部署到 apache+mod_perl 里面, 因为 Catalyst 为每个项目都提供了一个脚本,你可以在 script/目录下看到 x_server.pl,它类似于一个小型的 web server, 运行这个脚本的话,它会 use X.pm, 从而加载你的整个项目相关的所有插件跟组件, 而且它会在某个端口上进行监听, 当有客户端往该端口发送请求时,它会把请求交给 X.pm 去分发.

所以, 在开发阶段为了测试或者调试一般都会使用这个小型的 server.

现在我们来运行它: perl x_server.pl
它会在加载 Catalyst 项目的相关组件跟插件时打印出一大串的信息,这些先不用看, 加载完成后你会看到:

```
You can connect to your server at http://computername:3000
```

表示它已经启动成功了,现在你可以通过 3000 号端口来访问它.

假设我们这个项目所在的机器 IP 为: 192.168.1.1
那么现在我们试着用浏览器访问: http://192.168.1.1:3000/help
你会看到页面上显示了 "What can I do for you?"

恩, 当客户端发起请求时, 请求的 URL 是: /help
于是 X.pm 根据这个 URL 找到了 Root.pm 里面的 一个名字为 help 的 action 来处理这个请求, 这个 action 做的所有工作就是往响应的主体(body)里面填写一段字符串"What can I do for you?".

同理,如果我们在 Root.pm 里面写多几个 action,名字分别叫 action1, action2, action3, 那么它们分别可以处理这些请求: /action1, /action2, /action3

其实,编写 action 就是编写函数,只是这个函数跟一般的函数不同,它有一个或者多个属性,我们一般只编写具有一个属性的 action, 你可以看到 help 这个 action, 它的属性是 Local, 属性跟函数名之间以冒号隔开.action 的属性决定了这个 action 可以处理什么样的 URL.
Local 这个属性的意思就是说,这个 action 所能处理的 url = action 所处的 controller 的 namespace + "/" + action 的名字.

help 这个 action 处于 Root.pm 这个 controller 里面，一个 controller 默认的命名空间是它名字的小写，
如果含有冒号则全部以"/"进行替换，
比如
X::Controller::A 它的默认命名空间为 a，
X::Controller::A::B 它的默认命名空间为 a/b，
那么 X::Controller::Root 它的默认命名空间是 root，
那么 help 这个 action 所能处理的 URL = root + "/" + help = root/help
但是你在 X::Controller::Root.pm 里面看到了这样的一行代码：

```
__PACKAGE__->config->{namespace} = '';
```

这行代码把 X::Controller::Root 的命名空间改成了 ''，所以，
help 这个 action 所能处理的 URL = '' + "/" + help = /help
所以，我们访问 /help 的时候， X.pm 会发现 help 这个 action 可以处理这样的 URL，所以就把这个请求交给了 help 进行处理。

注意 Controller 的命名空间我们一般是不做任何修改的，Root.pm 进行了修改只是因为它比较特殊。
如果 Controller 的命名空间不修改的话，根据 url 跟 action 的对应关系，从一个 url 我们就可以看出它应该是哪个 Controller 的哪个 action 进行处理。(假设没有多个 action 对应同一个 url 的情况)

每个 action 的第一行代码基本上就是：
my ($self, $c) = @_;
这是负责分发请求给 action 是传进来的参数，$self 无特别意义，指 Controller 本身的名字而已，$c 是 Catalyst 里面最重要的一个对象。

$c 是什么?

c 是指 context 上下文的意思.$c 它其实是应用程序类的一个对象，它内部主要包含了以下组成部分:

$c->request　　: Catalyst::Request 对象,用于取得请求中的各种参数

$c->response　　: Catalyst::Response 对象，是对客户端请求的响应

$c->stash　　: 一个 hash 引用,用于 Catalyst 每个组件或者 action 之间传递数据

这 3 个对象的生命周期为一个请求周期,也就是说,在处理每一次客户端请求的时候,这 3 个对象都会重新生成.

之前提到过，model，view 是 Catalalyst 项目的组件，那么通过$c 可以得到这些组件，比如

$c->model('Model::DBIC') 可以得到组件(MyApp::Model::DBIC)

$c->model('View::TT') 可以得到组件(MyApp::View::TT)

组件的生命周期并不是一个请求周期，它在整个应用程序一开始时加载进来，在应用程序停止时才进行销毁。

在 X::Controller::Root.pm 里面你还可以看到其他 2 个 action:　default 跟 end,

关于 action 更详细的讲解,我们会在后面讲到.

第2节 action 详解

Catalyst 项目里面，action 的属性决定了这个 action 可以处理什么样的 URL.根据它们的属性，大概可以划分成以下几种类型:

1. path 类型 (包含 Local，Path，Global)
2. regex 类型 (包含 LocalRegex，Regex)
3. 自定义的 private 属性的 action
4. Catalyst 内置的 private 属性的 action (这些 action 包括 auto，begin，end，index，default)
5. 其他(暂时不在本教程讨论范围内,比如新版本的 Catalyst 里面的 Chained 属性)

有一点要注意,在选择哪一个 action 来处理某个 URL 的过程中，主要看 2 个因素:

1. action 所处的 Controller 的 namespace
2. action 本身(包括属性的定义,也可能包括 action 的名字)

1. path 类型 (包含 Local，Path，Global)

先看下面的例子:

```
package X::Controller::A::B;
sub list : Local {
}
sub g_list :  Global {
}
sub p_list : path('blist') {
}
sub p2_list  : path('/alist') {
}
sub p3_list : path('blist/clist') {
}
1;
```

在上面的例子里面,Controller 的 namespace = 'a/b'.

Local 属性的 action 是最为常见的,例子里面的 list 能匹配的 URL 为:
'a/b' + '/' + 'list' = 'a/b/list'

Global 属性的 action 跟 Local 属性的类似,只是它无视所在 Controller 的 namespace 而已,
例子里面的 g_list 可以匹配的 URL 为:
'/' + 'g_list' = '/g_list'

所谓 path 属性, 是 Local 属性跟 Global 属性的结合而已,
当 path 里面的参数第一个字符是 / 时,表示它是一个 Global 类型的, 否则,它就是一个 Local 类型的.
Path 属性的 action 所能匹配的 URL 跟 action 的名字无关,而是跟 path 里面的参数有关.
例子里面:
p_list 这个 action 其实 等同于: sub blist : Local , 所以它能匹配的 URL 为:
'a/b' + '/' + 'blist' = 'a/b/blist'

p2_list 这个 action 其实 等同于: sub alist : Global, 所以它能匹配的 URL 为:
'/' + 'alist' = '/alist'

p3_list 这个 action 能匹配的 URL 为:
'a/b' + '/' + 'blist/clist' = 'a/b/blist/clist'

这时候可能你有疑问, 如果我们定义了多个 path 类型的 action, 这些 action 都可以映射同一个 URL,那么究竟最后会调用哪个 action 呢?
首先,在实际开发的时候, 一定要尽量避免这种情况的发生, 出现这种情况的话,可以说就是 BUG.
因为在这种情况下, 只有最后定义的那个 path 类型的 action 可以映射 URL, 其他 action 都白定义了.看这个例子:

```
package X::Controller::A::B;
sub list : Local {
}
sub p_list : path('list') {
}
sub p2_list  : path('list') {
}
1;
```

这 3 个 action 都可以映射 URL: a/b/list, 实际上只有最后一个 action: p2_list 可以实际映射, 前面定义的 2 个 action(list, p_list)会被无情的抛弃.

注意:

一般我们发送 get 请求的时候，如果要在 URL 里面附上我们这次请求的参数(称为 query_param),那么这个请求的参数是类似这样传递的： http://xxxx/?p=1&s=2 ， ?后面的就是我们这次请求的参数， 而 catalsyt 不仅仅支持这种传统的方法来传递参数(这些请求参数可以通过$c->request->query_param 来取到)， 它还支持通过 argument 的方式来传递参数， 比如:
客户端请求 http://a/b/list/xiao/sheng
但 是我们没有定义过 a/b/list/xiao/sheng 这个 action，我们只定义过 a/b/list 这个 action,那么 catalyst 会把这 个请求也交给 a/b/list 这个 action 来处理，而 xiao，跟 sheng 这 2 个字符串就变成 URL 里面附加的请求参数， 你可以通过$c->request->arguments 来得到这 2 个字符串。那么也就是说:

path 类型的 action，比如上面例子里面的 list : Local， 我们说它可以匹配 'a/b/list'， 并不是说它只能匹配 a/b/list，它是有可能会匹配到 a/b/list/xiao/sheng 这样的 url 的。 等所有类型的 action 都讲完后，我会把如何根据一个 URL 找到一个 action 来跟它匹配剖析给大家看。

2. regex 类型 (包含 LocalRegex，Regex)

LocalRegex('...') 类似于 Path('..'),只是 LocalRegex 更为灵活，它的参数可以是一个正则表达式.
Regex('...') 类似于 Path('/..'),只是 Regex 更为灵活，它的参数可以是一个正则表达式.

比如

```
package X::Controller::A::B;
sub a : Path('list'){
}
sub b : LocalRegex('^list') {
}
sub a2 : Path('/list2') {
}
sub b2 : Regex('^list2') {
}

sub x : Regex('xiao') {
}

1;
```

a 这个 action 它可以匹配的 URL 为:
a/b/list

b 这个 action 它可以匹配的 URL 为:

a/b/^list
也就是 a/b/list*,即 URL 中以 a/b/list 开头

a2 这个 action 它可以匹配的 URL 为:
list

b2 这个 action 它可以匹配的 URL 为:
list2
也就是 list2* ，即 URL 中以 list2 开头

x 这个 action 它可以匹配的 URL 为:
xiao
也就是 *xiao* ，即 URL 中含有 xiao 字符串

需要注意，在为一个 URL 寻找一个 action 来进行分发时，path 类型的 action 的优先级高于 regex 类型的 action.
关于 regex 属性的 action,你可以从$c->req->captures 里面得到正则表达式里面匹配到的$1,$2..比如 Regex('^list(\d+)') 可以匹配 list5 这样的 url，匹配到的 5 保存在$c->req->captures->[0]里面。

3. 自定义的 private 属性的 action

private 属性的 action 不能直接映射为 URL，只能用于 action 内部的调用.
一般来说,为了处理客户端请求，编写 path 类型或者 regex 类型的 action 已经足够了，但是在某些情况下,编写 private 属性的 action，可以实现代码重用,见下面的例子:

```
package X::Controller::A::B;
sub c1 : Local{
    my ($self, $c) = @_;
    $c->forward('init_data');
    #continue
    $c->res->body("continute");
}
sub c2 : Local {
    my ($self, $c) = @_;
    $c->detach('init_data');
    #not return
    $c->res->body("last");
}
sub init_data : Private {
```

```
    my ($self, $c) = @_;
    #init data code
}
1;
```

c1, c2 是 2 个可以对外映射 URL 的 action, 而 init_data 则不可以,
但是 c1, c2 内部可以通过 forward 或者 detach 来调用到 init_data,
forward 就相当于函数调用,(只不过这个函数调用外部包了一个 eval，所以 action 内部就算 die 掉也只是抛出一个异常， 不会导致整个系统的崩溃)， c1 内部执行完 init_data 这个 action 后,会继续返回 c1 继续执行 c1 下面的代码,
detach 跟 forward 唯一的区别就是 detach 执行完不再返回, c2 执行完 detach 之后不返回, 也就是说 $c->res->body("last"); 这行代码不会被执行到.

private 属性的 action, 我认为它的最大优点在于它可以跨组件调用, 也就是 Controller A 可以调用 Controller B 里面的一个 private 属性的 action.这跟我们以前写的 cgi 不一样, cgi 里面是不可以直接调用另外一个 cgi 文件里面定义的函数的.

从这点上, Catalyst 提倡的 donot repeat yourself 可见一斑.

这种跨组件调用可以见下面例子:

```
package X::Controller::A;
sub init_data : Private {
    my ($self, $c) = @_;
    #init data code
}
1;

package X::Controller::B;
sub c1 : Local{
    my ($self, $c) = @_;
    $c->forward('/a/init_data');
    #continue
    $c->res->body("continute");
}

1;
```

关于 forward 跟 detach,你会在后面章节看到他们更详细的说明.

4. Catalyst 内置的 private 属性的 action (这些 action 包括 auto, begin, end, index, default)

先来谈谈 begin 跟 end.
假设我们现在有这样的 Controller 结构:

```
X::Controller::Root;        ( namespace 为'')
X::Controller::A;           ( namespace 为默认的'a')
X::Controller::B;           ( namespace 为默认的'b')
X::Controller::A:A2;        ( namespace 为默认的'a/a2')
X::Controller::A:A2::A3;    ( namespace 为默认的'a/a2/a3').
```

假设现在客户端请求的 URL 为 a/a2/a3/list
刚好 X::Controller::A:A2::A3 里面定义了一个名为 list,属性为 Local 的 action,
那么 Catalyst 就会把 URL 分发给 X::Controller::A:A2::A3::list 这个 action 进行处理,
在 list 这个 action 开始执行之前, Catalyst 会检查有没有在 a/a2/a3 这个 namespace 下定义过名 字为 begin, 属性为 private 的 action,如果定义过,那么先执行 a/a2/a3/begin 这个 action,之后再执行 list 这个 action,如果没 有定义过, 那么会沿着 a/a2/a3 这个命名空间往上一层去找, 也就是检查有没有定义过 a/a2/begin 这个 action,如果有则执行, 然后再执行 list,如果没有则继续往上找,直到找到或者 namespace 为空也没找到.
也就是说它按照下面的顺序来查找有没有定义过 begin, 有则先执行 begin 再执行 list:

```
a/a2/a3/begin
a/a2/begin
a/begin
begin
```

用一句话来概括就是:
根据 URL 找到可以匹配的 action 后, 在执行该 action 之前,会在该 action 所处的命名空间的每个层次上, 从下往上寻找一个名为 begin,属性为 private 的 action,找到则执行,执行完不再往上找, 接下来执行匹配的 action.

在上面例子里面, 如果同时定义了 a/a2/a3/begin 跟 a/begin, 在执行 a/a2/a3/begin 之后, 接下来就去执行 list 了, 不会继续往上找到 a/begin 来执行.

在执行完 list 这个 action 之后, Catalyst 会去寻找和执行一个名为 end,属性为 private 的 action, 它寻找的顺序是这样的:

```
a/a2/a3/end
a/a2/end
a/end
end
```

看得出来寻找 end 跟寻找 begin 是一样的，只是一个是在执行 list 之后，一个是在执行 list 之前。对于 end， 用一句话概括就是：
根据 URL 找到可以匹配的 action 后，在执行该 action 之后，会在该 action 所处的命名空间的每个层次上，从下往上寻找一个名为 end，属性为 private 的 action，找到则执行，执行完不再往上找.

实际上， 在上面的例子中， 当执行完 begin 之后，在执行 list 之前， catalyst 还会去寻找一种名为 auto，属性为 private 的 action，在上面的例子中，它寻找的 auto 的顺序是：

```
auto
a/auto
a/a2/auto
a/a2/a3/auto
```

当寻找一个 auto 之后，catalyst 会执行它，然后判断 auto 这个 action 的返回值， 如果 auto 返回真值，那么继续往下寻找 auto， 继续执行，直到所有的 auto 都执行完毕且返回真值后， 才会去执行 list 这个 action。如果当中有某个 auto 返回了假值， 那么就不再往下寻找 auto，也不会再去执行 list，就会直接跳到寻找 end 的过程。
用一句话来概括的话就是：
根 据 URL 找到可以匹配的 action 后， 在执行 begin 之后，在执行该 action 之前，会在该 action 所处的命名空间的每个层次上，从上到下寻找一个名为 auto，属性为 private 的 action，找到则执行，如果 auto 返回真值就继续往下寻找执行 auto，当所有的 auto 都返回真值时，接下来才会去执行 list，否则则直接跳到寻找执行 end 的过程。

所以在找到 list 这个 action 后， catalyst 实际上执行了一系列的 action，这堆 action 就是所谓的反应链，在本例中，该反应链为：

```
a/a2/a3/begin or a/a2/begin or a/begin or begin
auto
a/auto
a/a2/auto
a/a2/a3/auto
a/a2/a3/list
a/a2/a3/end or a/a2/end or a/end or end
```

剩下还有 2 个很特殊的 private 属性的 action: index 跟 default.

我们之前讲到， 在为一个 URL 寻找一个 action 来进行分发时，path 类型的 action 的优 先级高于 regex 类型的 action. 那么，是不是 path 类型的 action 的优先级是最高的呢？ 不是的，有一种比 path 类型的 action 优先级更高，它就是 private 属性的 index。下面例子：

```
package X::Controller::A;
```

```
sub b : Local {
}
1;

package X::Controller::A::B;
sub index : private {
}
1;
```

X::Controller::A::b 这个 action 可以匹配 url: a/b,
X::Controller::A::B::index 可以而且只能匹配的 url 是 a/b,
我们之前讲到 X::Controller::A::b 不仅可以匹配 a/b 这个 url，当没有定义 a/b/c 这个 action 时，它可能可以匹配 a/b/c 这个 URL。
而 index 这种 action 它只能匹配所在 Controller 的 namespace，也就是 a/b。
当有客户请求 a/b 时， 优先考虑的是 X::Controller::A::B::index 这个 action，如果这个 action 没有定义，才会匹配到 X::Controller::A::b。
在说到 default 这个 action 之前，先看下面的一个例子：

```
package X::Controller::A;
sub d : Path('') {
}
1;
```

d 这个 action 它可以匹配 a 这样的 url,如果客户请求 a/xiaosheng 这样的 url 时，并没有定义 action: a/xiaosheng 时，那么 d 还可以用来处理这个 URL，xiaosheng 则变成请求的一个参数，从$c->req->arguments 可以得到它。
从这里可以看到，如果客户端访问 a/*的 URL 时，如果没有定义过相应的 a/*这个 action 的话， d 这个 action 就会派上用场。
那么其实， d 这个 action 可以改成名为 default， 属性为 private 的 action,如下：

```
package X::Controller::A;
sub default : Private {
}
1;
```

这样的话，default 这个 action 起到的作用跟 d 的作用是一样，主要用来当找不到可用的 action 来匹配 URL 时，就会调用这样 一个默认的 action 来进行处理，一般是显示一个错误页面，提示用户访问了一不存在的页面。 default 这个 action 跟 d 的区别在于， 当客户端访问 a/xiaosheng 时， default 内部得到的$c->req->arguments 并不是['xiaosheng'], 而是['a','xiaosheng']。当然还有一个区别，那就是 default 这种 action 的优先级是最低的，比 regex 类型的还低。

从这里我们可以知道， X::Controller::Root::default 这个 action 里面可以设置一个全站点的一个默认页面。

5．action 的继续剖析

客户端访问 a/b/c 时，
决定选择哪一个 action 来处理这个请求的过程如下：
1．$namespace=a/b/c， 到 2
2．在$namespace 这个命名空间下， 有没有定义过 index 这种 action,有则选之，无则到 3
3．在$namespace 这个命名空间下， 有没有定义过 path 类型的 action 可以处理 a/b/c 这个 URL,有则选之，无则到 4
4．在$namespace 这个命名空间下， 有没有定义过 regex 类型的 action 可以处理 a/b/c 这个 URL,有则选之，无则到 5
5．在$namespace 这个命名空间下， 有没有定义过 default action 可以处理 a/b/c 这个 URL,有则选之，无则到 6
6．如果$namespace 已经不可再往上，那么报错，退出，否则$namespace= $namespace 往上一层(比如 a/b/c 往上一层则是 a/b)，到 2

由此可见，如果没有定义 Root::default，那么将可能导致无法找到一个 action 来处理而导致出错。

第3节 如何编写 action

大家现在应该对 action 都有了一个大概的概念,那么这一节开始讲如何编写 action.

如果我们不想在 Root.pm 里面写 action,那么在编写 action 之前,我们需要先创建一个 Controller,假设它名字为 A,那么在 script 文件夹下面执行:
perl x_create.pl Controller A
这样你就可以在 lib/X/Controller/里面看见刚创建好的 A.pm 了.
我们在 A 里面写一个最简单的 action:

```
package X::Controller::A;
sub list : Local {
    my ($self, $c) = @_;
    my $id = $c->req->param('id');
    if ($id) {
        $c->res->body("your id is $id");
    } else {
        $c->res->body("no id input");
    }
}
```

如果客户端请求的 URL 是: a/list/?id=2007, 那么网页内容会是 your id is 2007,
如果客户端请求的 URL 是: a/list/, 那么网页内容会是 no id input.

下面先介绍$c 的一些常用方法:

1. $c->req

该 方法是$c->request 的一个别名,返回一个 Catalyst::Request 对象. 当客户端一个请求过来时,Catalyst 把该次请求相关的数据封装在$c->reqeust 这个对象里面, 然后使用$c->request 一些方法

可以得到该次请求里面涉及到的参数内容.
它常用的方法有: (更详细的内容请参看 CPAN 文档)

1. query_parameters
得到该次请求里面的 GET 参数.
客户请求 a/list/?id=2007, $c->request->query_parameters('id')会得到 2007 这个值

2. body_parameters, 别名 body_params
得到该次请求里面的 POST 参数.
客户端向 a/list/发送一个 POST 请求, post 参数为 id=2007&name=xiaosheng, 那么
$c->request->body_parameters('id')会得到 2007 这个值,$c->request->body_parameters('name')会得到 xiaosheng 这个值

3. param
这个方法比以上 2 个方法更常用, 该方法可以取到请求参数,而不管该参数是 GET 参数还是 POST 参数.
客户请求 a/list/?id=2007, $c->request->param('id')会得到 2007 这个值
客户端向 a/list/发送一个 POST 请求, post 参数为 id=2007, $c->request->param('id')一样会得到 2007 这个值

4. header
得到请求头部的某个值
$c->req->header('User-Agent')可以得到请求头部里面的 User-Agent 的值.

5. user_agent
其实就是$c->req->header('User-Agent')

6. address
得到该次请求的 IP 地址

7. arguments
得到传递给某 action 的参数.
客户端请求 a/b/list/xiao/sheng
但是我们没有定义过 a/b/list/xiao/sheng 这个 action, 我们只定义过 a/b/list 这个 action,那么 catalyst 会把这个请求也交给 a/b/list 这个 action 来处理,
在 action 里面
$c->request->arguments == ['xiao', 'sheng']
取 arguments 你还可以这样取:

```
package X::Controller::A::B;
sub list : Local {
    my ($self, $c, $x, $y) = @_;

}
```

$x 就是$c->req->arguments->[0],
$y 就是$c->req->arguments->[1].

如果你是在 action 内部 forward 到另外一个 Private 的 action,那么$c->req->arguments 在 forward 之前会先被保存起来，调用完再恢复.

```
package X::Controller::A::B;
sub list : Local {
    my ($self, $c, $x, $y) = @_;

    #这里, $c->req->arguments = [$x, $y] = ['xiao', 'sheng']


    $c->forward('p', ['1', '2']);
    $c->res->body('list' );
    #这里, $c->req->arguments 还是= [$x, $y] = ['xiao', 'sheng']

}


sub p: Private {
  my ($self, $c, $a, $b) = @_;
  #这里, $c->req->arguments= [$a, $b] = ['1', '2']
}
```

2. $c->res

该 方法是$c->response 的一个别名,返回一个 Catalyst::Response 对象.当处理完客户端的一个请求后,Catalyst 要 给客户端一个响应, 包含响应头部跟内容(header 和 body). $c->response 就是封装整个响应的一个对象.
它常用的方法有: (更详细的内容请参看 CPAN 文档)

1. header
设置响应头部, 比如
```
$c->res->header('Content-Type' => 'text/html');
```

2 content_type
```
$c->res->content_type('text/html');
```
其实就是
```
$c->res->header('Content-Type' => 'text/html');
```

3 body 跟 output 是一样的
设置响应的内容
```
$c->res->body('<html><body>hello world</body></html>');
```

4. redirect
重定向到另外一个 URL 里面
```
$c->res->redirect ('http://www.google.com');
```
这行代码导致客户端浏览器收到这个响应之后马上再发出一个请求到 http://www.google.com.
如果是本站点重定向的话, 经常跟$c->uri_for 一起使用:
```
$c->res->redirect ( $c->uri_for('/book/detail/', { id => 3} ) );
```
假设一开始的请求 URL 是 http://x.com/action1,那么 action1 这行代码会导致重定向到:
http://x.com/book/detail/?id=3

关于$c->uri_for 的详细说明请参看下文或者 CPAN 文档

3. $c->stash
该方法返回一个 hash 引用, 生命周期同$c->req, $c->res, 为一个请求周期. 在一个请求周期内可以把它当作一个垃圾堆来放置数据, 主要用于 action 之间传递数据或者组件之间传递数据, 在以后的例子会经常看到它.
```
$c->stash->{list} = ['a', 'b']; #往 stash 里面放置一个 key=list,value=['a','b'].
```

4. $c->log
该方法返回一个 Catalyst::Log 对象, 你可以调用该对象的方法来往标准错误里面输出一些 DEBUG 消息或者错误消息.

```
$c->log->error('test');
$c->log->info('test');
```
如果你开着 catalyst 自带的 server 来调试程序,就会看到屏幕上打印出字符串"test".
$c->log 一般用来输出一些变量的值进行调试用.

5. $c->config
该方法返回整个项目配置的一个 hashref.一般我们使用 Catalyst::Plugin::ConfigLoader 这个插件在项目初始化时读取 app.yml 文件, 读出来后就是一个哈希引用, 也就是$c->config.
$c->config->{home} 一般就是指 Catalyst 项目的 app 目录在磁盘上的路径.

6. $c->forward
跳转到另外一个 action 或者组件里面进行处理.
forward 到 action 里面的代码其实都会被包裹在 eval 里面, 这个时候就算你写 die 也不会导致整个应用程序崩溃.

它有几种参数调用方式:
1. $c->forward($action [, \@arguments])
$action 参数必须是一个 Controller 里面的 action 的名字,可以是绝对路径, 如果不是绝对路径那就是当前 controller 里面的 action:

```
my $foo = $c->forward('/foo');
$c->forward('index');
```

2. $c->forward($class, $method, [, \@arguments])
$class 是某个组件的名字, 可以是 Controller,也可以是 View, Model,$method 就是该组件内部定义的方法,如果没有$method 参数,默认就是调用组件内部的 process 方法.

7. $c->detach
跟 forward 类似,只是跳到另外一个 action 之后就不再返回执行原 action 下面的代码.

8. $c->path_to
假设项目所在位置为/home/xiaosheng/X/
$c->path_to('bin', 'do.pl')就是字符串 '/home/xiaosheng/X/bin/do.pl'.

9. $c->uri_for
它返回一个 URI 对象. 参数形式如下:
$c->uri_for($path, @args?, \%query_values?)
假设客户端请求 http://xiaosheng.com/a/list/ , 匹配这个 URL 的 action 是 X::Controller::A::list 在这个 action 里面调用

$c->uri_for('list2'); 将返回 URI 对象: url 为 http://xiaosheng.com/a/list2
$c->uri_for('/list2'); 将返回 URI 对象: url 为 http://xiaosheng.com/list2
$c->uri_for('list2', 'detail'); 将返回 URI 对象: url 为 http://xiaosheng.com/a/list2/detail
$c->uri_for('list2', { id => 2}); 将返回 URI 对象: url 为 http://xiaosheng.com/a/list2?id=2
$c->uri_for('list2', 'detail', { id => 2}); 将返回 URI 对象: url 为 http://xiaosheng.com/a/list2/detail?id=2

第 4 章　　掌握 View

第1节 简介

现在我们应该知道 controller 的编写方法了，这一章我们将开始介绍 View.

在 MVC 模式里面，View 可以有多个,它指的是数据的外在表现.
假设我们系统里面有 3 个 View:
ViewA: 负责把数据填到 HTML 里面生成 HTML 代码
ViewB: 负责把数据写到图片文件中从而生成一张含有数据的图片
ViewC: 负责处理数据从而生成 PDF 文件.
那么一样的数据，它经过 ViewA 表现出来的话它就是一张网页，它通过 ViewB 表现出来那么就是一张图片，它通过 ViewC 表现出来就是一个 PDF 文件.

一般的 Catalyst 项目， 都至少有一个 View 专门负责网页的 HTML 代码生成.

假 设我们在项目里面添加一个 View，它名字叫 MyView,那么它的模块名就是 X::View::MyView，它必须从 Catalyst::View 这个模块继承，这样它才是一个有效的 Catalyst 组件. 这样我们就可以通过 $c->view('MyView')来得到 X::View::MyView 这个组件.

当我们在 action 里面编写这样的代码:

```
$c->forward('View::MyView');
```

程序就会跳转到 X::View::V 这个组件的 Process 方法里面，所以,我们要定义这个组件的 Process 方法.

下面我们演示如何创建一个 View 以及如何在 Controller 里面使用这个 View.

首先先在 script 文件夹下，执行:
perl x_create.pl View MyView
这样就会自动创建模块 X::View::MyView，然后在该模块下编写一个 process 方法:

```
sub process {
    my ( $self, $c ) = @_;
```

```
    my $data = $c->stash->{data};

    unless ( $c->response->headers->content_type ) {
        $c->res->headers->content_type('text/html; charset=utf-8');
    }

    $c->response->body(qq#<html> <body> <font color ="red">$data</font></body></html>#);

    return 1;
}
```

这个 process 方法就是从$c->stash->{data}里面得到一个字符串,然后把这个字符串渲染成红色, 最后把 HTML 代码写到响应的内容里面,同时设置一下响应的头部.

然后我们在 Controller 里面这样写:

```
package X::Controller::A;
use strict;
use warnings;
use base 'Catalyst::Controller';

sub list : Local {
     my ($self, $c,) = @_;
     $c->stash->{data} = 'list';
}

sub list2 : Local {
```

```
    my ($self, $c,) = @_;
    $c->stash->{data} = 'list2';
}

sub end : Private {
    my ($self, $c) = @_;

    $c->forward('View::MyView');
}

1;
```

当 用户访问 a/list 时, list 这个 action 会往 stash 里面存放一个字符串'list',接着 Catalyst 会调用最近的 end action,这个 action 里面 forward 到了 MyView,于是调用了 MyView 的 process 函数进行处理,最后页面上看到一个红色的 'list'字符串,
同理,如果用户访问 a/list2,页面上可以看到一个红色的'list2'字符串.

那么到这里为止,你已经了解了一个最简单的 View 的工作原理.

在 大多数情况下, 我们并不会像例子这样自己从头写一个 View, 因为如果你要把数据渲染到 HTML 代码里面的话,CPAN 已经有非常多的模块做这些工作,我们没必要重新发明轮子. 所以我们一般是定义一个 View,然后这个 View 里面 use 一个 CPAN 上的模块, 然后在 process 里面调用 CPAN 该模块的方法生成 HTML 代码即可.
所以, 从这种意义上, 在 Catalyst 里面,View 只是一个接口而已.

CPAN 有一套功能强大的模块: Template::Toolkit(简称 TT),它可以做把数据渲染到 HTML 里面的工作, 而且, 已经有人为 TT 写了一个 Catalsyt::View::TT, 我们只需要把这个 Catalsyt::View::TT 直接拿来用即可, 如果你觉得 Catalsyt::View::TT 这个名字不爽, 你可以自己写一个 Catalsyt::View::MyTT,只要它是从 Catalsyt::View::TT 继承即可,你无需编写其他代码.

下面一节我们将开始介绍 TT 的使用.

第2节 TT 入门

Template::Toolkit，简称 TT，它本身是一套非常强大的模块,跟 Catalyst 是无关的,所以这一节是只讲 TT 的基本用法,下一节讲如何在 Catalyst 里面使用 TT.

一 关于 TT

TT 它是一套模板处理系统，它的主要功能简单来讲就是,给它一个模板以及模板内变量的值,它可以帮你把变量的值替换进模板,生成最后的文本．常用于生成 HTML 代码.
关于 TT 更详细的文档,请参看 CPAN.

二 TT 基本用法

1．替换变量:

```
use Template;
my $tt = Template->new;

#要替换的变量值:
my $vars = {name => 'xiaosheng'};

#模板可以是一个文件，也可以是一段字符串，也可以是一个文件句柄，

#方法 1
my $template_str = 'hi, my name is [% name %]';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";
#方法 2
my $template_file = 'a.tt';  #a.tt 的内容是'hi, my name is [% name %]'
$tt->process($template_file , $vars) || die $tt->error(), "\n";

#方法 3

my $template_str = 'hi, my name is [% name %]';
my $output;
$tt->process(\$template_str, $vars, \$output) || die $tt->error(), "\n";
```

在上面的例子里,方法 1,跟方法 2 都会往标准输出打印出'hi, my name is xiaosheng'

方法 3 里面，$output 的值为字符串'hi, my name is xiaosheng';

2 循环语句

```
use Template;
my $tt = Template->new;

#例 2.1
my $vars = {
  cai_list=> ['caicai1', 'caicai2', 'caicai3']
};
my $template_str = '
[% FOREACH cai IN cai_list %]
the name is [% cai %]
[% END %]
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
the name is caicai1

the name is caicai2

the name is caicai3
#例 2.2
$vars = {
  cai_list=> [
      { name => 'caicai1', id => 1 },
      { name => 'caicai2',id => 2 }
  ]
};
$template_str = '
[% FOREACH cai IN cai_list %]
the name is [% cai.name %], the id is [% cai.id %]
[% END %]
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";
#打印出
```

```
the name is caicai1, the id is 1

the name is caicai2, the id is 2

#例 2.3
$template_str = '
[% i = 10 %]
[% WHILE i > 0 %]
now is i=[% i %]
[% i = i - 1 %]
[% END %]
';

$tt->process(\$template_str, $vars) || die $tt->error(), "\n";
#打印出
now is i=3
now is i=2
now is i=1
```

3 条件语句

```
 my $vars = {
  cai_list=> [
      { name => 'caicai1', score => 100 },
      { name => 'caicai2', score => 85  },
      { name => 'caicai3', score => 95 },
      { name => 'caicai1', score => 75 },
      { name => 'caicai2', score => 65  },
      { name => 'caicai3', score => 55 },
  ]
};
my $template_str = '
[% FOREACH cai IN cai_list %]
    [% IF cai.score == 100  %]
         [% cai.name %] : 满分
    [% ELSIF cai.score >= 90 %]
         [% cai.name %] : 优秀
    [% ELSIF cai.score >= 80 %]
         [% cai.name %] : 良好
```

```
    [% ELSIF cai.score >= 70 %]
         [% cai.name %] : 一般
    [% ELSIF cai.score >= 60 %]
         [% cai.name %] : 及格
    [% ELSE %]
        [% cai.name %] : 不及格
    [% END %]
[% END %]
';

$tt->process(\$template_str, $vars) || die $tt->error(), "\n";
#打印出
caicai1 : 满分
caicai2 : 良好
caicai3 : 优秀
caicai1 : 一般
caicai2 : 及格
caicai3 : 不及格
```

4 去空格以及回车

```
use Template;
my $tt = Template->new;
#例 4.1
my $vars = {
  name => 'xiaosheng',
};
my $template_str = '
hi,
[% name %],
good morning
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
hi,
xiaosheng
,good morning

#例 4.2
```

```
$vars = {
  name => 'xiaosheng',
};

$template_str = '
hi,
[%- name %]
,good morning
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
hi,xiaosheng
,good morning

#例 4.3
$vars = {
  name => 'xiaosheng',
};

$template_str = '
hi,
[% name -%]
,good morning
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
hi,
xiaosheng,good morning

#例 4.4
$vars = {
  name => 'xiaosheng',
};

$template_str = '
hi,
[%- name -%]
```

```
,good morning
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";
#打印出
hi,xiaosheng,good morning
```

5 常用的虚方法(virtual method)

在上面例子我们看到,对于替换进模板的那些变量,
如果$var1 是一个 hashref, 如果在 TT 里面要得到$var1->{key1}的值, 在 TT 里面可以写 var1.key1.
如果$var1 是一个对象,它有一个方法名为 print,在 TT 里面要调用$var1->print()的话就得写成可以通过 var1.print.
也就是说, 在 TT 里面通过 . 可以调用一个对象的方法.

如 果 var1 是一个 scalar 或者是一个 list 又或者是 hash, 它本身没有 var1->defined 这个方法, 但是在 TT 里面你通过 var1.defined 来判断 var1 是否是 defined 的值, 这个 defined 的方法,是 TT 给这个变量加上的虚方法.
关于详细的 virtual method,请参考 CPAN 文档
http://search.cpan.org/~abw/Template-Toolkit-2.18/lib/Template/Manual/VMethods.pod

这里稍微介绍几个虚方法:

```
use Template;
my $tt = Template->new;
#defined
#适用于 scalar,判断一个 scalar 是否 defined
my $vars = {
  cai_list=> [
      { name => 'caicai1' },
      { name => '0' },
      { name => undef }
  ]
};
my $template_str = '
[% FOREACH cai IN cai_list %]
     [% IF cai.name.defined %]
        [% cai.name %]
     [% ELSE %]
        not defined
```

```
    [% END %]
[% END %]
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
caicai1
0
not defined

#length
#适用于 scalar,得到一个字符串含有的字符个数
my $vars = {
  cai_list=> [
      { name => 'caicai1' },
      { name => 'xiaosheng' },
  ]
};
my $template_str = '
[% FOREACH cai IN cai_list %]
    length of name is [%  cai.name.length %]
[% END %]
';$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
length of name is 7
length of name is 9

#size
#适用于 list,得到 list 包含的元素个数
my $vars = {
  cai_list=> [
      { name => 'caicai1' },
      { name => 'xiaosheng' },
  ]
};
my $template_str = '
[% cai_list.size %]
';
```

```
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
2
```

6 过滤器 filter

对于要替换进模板的变量,在显示之前可以给它加上一个或者多个过滤器，把过滤后的结果再替换到模板里面.

关于所有的 filter,请参考

http://search.cpan.org/~abw/Template-Toolkit-2.18/lib/Template/Manual/Filters.pod

这里稍微介绍几个

```
use Template;
my $tt = Template->new;
#html
#起到 html escape 的作用
my $vars = {
  var => '<font color=red>xiaosheng</font>',
};
my $template_str = '
[% var | html %]
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
&lt;font color=red&gt;xiaosheng&lt;/font&gt;

#format
#格式化输出
my $vars = {
    var => '100.345',
};
my $template_str = '
[% var | format("%.02f")%]
';
$tt->process(\$template_str, $vars) || die $tt->error(), "\n";

#打印出
100.35
```

第3节 把 TT 作为 View 使用

要把 TT 作为 View 在 Catalyst 里面使用,只需要:
1. 建立一个 X::View::MyView, 使之继承自 Catalyst::View::TT.
2. 配置这个 View.

现在继续拿出我们前面几章里面的示例项目: X, 键入命令:
perl script/x_create.pl View MyView TT
这条命令执行完, 会在 lib/X/View/下面创建 X::View::MyView.pm.

接下来我们进行配置.
假设我们现在要给这个 View 配置模板的默认扩展名为.tt.

对于 Catalyst 的组件, 配置主要有 3 种方法,

第一种是直接在组件内部配置, 也就是在 Catalyst::View::MyView.pm 里面写:
__PACKAGE__->config(TEMPLATE_EXTENSION => '.tt');

第二种是在应用程序类内配置, 也就是在 MyApp.pm 里面, 在 setup 这行代码前写:
__PACKAGE__->conifg('View::MyView' => { TEMPLATE_EXTENSION => '.tt', });

第三种是在 x.yml 里面配置,前提是项目使用了 ConfigLoader 这个插件. 在 x.yml 里面写:
View::MyView:
  TEMPLATE_EXTENSION: .tt

第 3 种方法是比较推荐的做法, 整个项目的组件跟插件的配置,或者是其他的配置, 都可以写到 x.yml 里面.

关于这个 View 更详细的配置,请参看 CPAN 文档. 现在我们整个 x.yml 里面内容如下:

```
---
name: X

View::MyView:
  INCLUDE_PATH:
```

```
    - __HOME__/root/tt
  TEMPLATE_EXTENSION: .tt
```

__HOME__是 ConfigLoader 支持的一个变量,它实际上指整个项目的 home 目录
这几行配置使到 View::MyView 在找模板的时候会去__HOME__/root/tt 目录下找, 当我们没有指定模板名时,这个 View 会根据当前匹配的 action 名以及配置的扩展名.tt 去找模板.

下面我们在 root/tt/下面建立一个 test.tt,内容如下:

```
hello, this is only a test  <br>

[% var1 %] <br>
[% var2 %] <br>
 our project name is :[% c.config.name %]
```

在 Root.pm 里面修改 action 代码如下:

```
sub default : Private {
    my ( $self, $c ) = @_;
    $c->response->body( $c->welcome_message );
}

sub t :Local {
    my ( $self, $c ) = @_;
    $c->stash->{template} = 'test.tt';
}

sub end : Private  {
     my ( $self, $c ) = @_;
    $c->forward('View::MyView') unless $c->res->body;
}
```

启动我们的 x_server.pl,访问 http://xxxx/t

你会看到页面:

```
hello, this is only a test
this is var1
this is var2
our project name is :X
```

当你访问 http://xxxx/t 时，t 这个 action 匹配这个 URL.
我们准备在这个 action 里面使用一个叫 test.tt 的模板，所以我们先把模板名字存到 stash 里面，待会 View::MyView 会把从 stash 里面得到这个模板名字,然后去磁盘读取这个模板.
我们还往 stash 里面放了一些变量: var1,var2,这些 stash 里面的变量，待会 View::MyView 都会从 stash 里面把它们读出来,然后全部替换到模板里面.
在 end 这个 action 里面，程序 forward 到 View::MyView 这个组件，该组件的 process 方法会从 stash 读出 template 的名字，然后从磁盘加载模板，再把 stash 里面的变量都替换到模板里面，最后把输出保存到$c->res->body 里面.

然后你会发现,我们并没有往$c->stash 里面放入一个名为 c 的变量,为什么模板里面却有这样的语句呢: [% c.config.name %]，那是因为，View::MyView 会自动的把$c 这个变量放到模板里面去替换变量 c.所以我们可以在 action 里面省略这样的代码: $c->stash->{c} = $c;

第 5 章　　掌握 Model

第1节 简介

这一章讲 MVC 的最后一块：Model．前面已经介绍过 Model,它主要负责对数据源的访问，一般就是负责数据库的读取．

Model 跟 View 差不多,在大多数情况下，它只是一个接口．

CPAN 有一套模块：DBIx::Class::Schema． 它使用了所谓的对象关系映射（Object Relational Mapping，简称 ORM）的技术， 把数据库的每一张表都映射成一个 class，对数据库的操作无需直接写 SQL，只需要对 class 进行操作即可．
已经有人为它写了一个 Catalyst::Model::DBIC::Schema， 我们只需要把这个 Catalyst::Model::DBIC::Schema 直接拿来用即可，同样的,你可以写自己的 Catalyst::Model::MyDB 只需要它从 Catalyst::Model::DBIC::Schema 继承即可．

下一节会讲 DBIx::Class::Schema 的基本用法．学习 DBIx::Class::Schema 会比较累，需要大家多读文档,多写代码．

第2节 DBIx::Class::Schema 入门

这一节将以 mysql 数据库为例.

创建数据库,名为 X.初步创建 5 个表:
book: 字段有 id, author_id, name
author: 字段有 id, name, is_male
chapter: 字段有 id, name, book_id, subject, content
type: 字段有 id, name
book_type: 字段有 id, book_id, type_id

一个 author 有多个 book, 一个 book 有多个 chapter,
type 指 book 的类别
一本 book 可能属于多个类别, book 跟 type 的关系存在 book_type 这张关系表里.

建数据库语句如下:

```
CREATE DATABASE `X` CHARACTER SET utf8;
USE `X`;
CREATE TABLE `book` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `author_id` int(10) unsigned default NULL,
  `name` varchar(255) default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `author` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `is_male` tinyint(1) default NULL,
  `name` varchar(255) default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `chapter` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `book_id` int(10) unsigned default NULL,
  `name` varchar(255) default NULL,
  `subject` varchar(255) default NULL,
  `content` text default NULL,
  PRIMARY KEY (`id`),
  KEY `x_book_id` (`book_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `type` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `name` varchar(255) default NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `book_type` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `book_id` int(10) unsigned default NULL,
  `type_id` int(10) unsigned default NULL,
  PRIMARY KEY (`id`),
  KEY `x_book_id` (`book_id`),
  KEY `x_type_id` (`type_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

首先创建一个 MySchema.pm,使之从 DBIx::Class::Schema 继承,如下:

```
package MySchema;
use base 'DBIx::Class::Schema';
use strict;
use warnings;

__PACKAGE__->load_classes();

1;
```

接下来为每个表创建一个 class,这种 class 称为 DBIx::Class::ResultSource.

建立 MySchema/Book.pm, MySchema/Author.pm, MySchema/Type.pm, MySchema/Chapter.pm, MySchema/BookType.pm,如下:

```
package MySchema::Book;
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('book');
__PACKAGE__->add_columns(qw/id name author_id/);
__PACKAGE__->set_primary_key('id');
1;
package MySchema::Author;
```

```
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('author');
__PACKAGE__->add_columns(qw/id name is_male /);
__PACKAGE__->set_primary_key('id');
1;


package MySchema::Chapter;
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('chapter');
__PACKAGE__->add_columns(qw/id name book_id subject content/);
__PACKAGE__->set_primary_key('id');
1;


package MySchema::Type;
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('type');
```

```
__PACKAGE__->add_columns(qw/id name/);
__PACKAGE__->set_primary_key('id');
1;


package MySchema::BookType;
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('book_type');
__PACKAGE__->add_columns(qw/id book_id author_id/);
__PACKAGE__->set_primary_key('id');
1;
```

MySchema.pm 它继承自 DBIx::Class::Schema，它代表的是对数据库的连接，在 MySchema.pm 里面的这行代码:
__PACKAGE__->load_classes();
表示它会去加载 MySchema::*.pm，也就是 5 个表对应的 5 个 ResultSource.

现在我们已经可以通过 MySchema 来访问数据库了,下面我们写一个测试脚本:

```
#!perl
use strict;
use MySchema;
my $schema = MySchema->connect(
    'dbi:mysql:x;localhost',
    'root',
```

```
    '',
    { AutoCommit => 1 },
);
$schema->storage->debug(1);
$schema->resultset('Book')->find(1);
```

MySchema->connect 内部其实就是 DBI->connect，连接数据库。
$schema->storage->debug(1) 把 DBIC 的 DEBUG 模式打开，这样的话，下面的 DBIC 语句如果执行了 SQL 操作,将会把执行的 SQL 语句打印出来。
$schema->resultset('Book')->find(1); 这行代码执行后，屏幕上会打印出这样的 SQL 语句：

```
SELECT me.id, me.name, me.author_id FROM book me WHERE ( ( me.id = ? ) ): '1'
```

这行 SQL 实际上就是 SELECT * FROM book WHERE id=1
$schema->resultset('Book') 它实际上是 SELECT * FROM book 的一个结果集。

在这样的一个结果集(resultset)上我们一般可以使用这些方法：

◆ find: 基于主键或者 UNIQUE 键的搜索，返回一条记录。(一条记录的对象称为 RowObject.)
上例中 id 是 BOOK 表的主键，find(1)就是 SELECT * FROM BOOK WHERE id=1。

next: 逐条迭代结果集中的记录，比如要打印结果集中所有记录的 id,可以这样用：

```
while(my $row_object = $resultset->next) {
    print $row_object->id, "\n";
}
```

上面的这行代码：$row_object->id
对于某个 RowObject,要得到它的某个字段的值，可以通过 $row_object->字段名()来调用， 要修改它的某个字段的值，可以通过 $row_object->字段名($new_value); 来修改它的值。

◆ search: 最常用的数据库检索方法，在标量上下文它返回检索过后的 resultset,在列表上下文它返回所有记录对象的一个数组(也就是一个 RowObject 数组)。
search 方法它有 2 个参数，2 个参数都是 HASH 引用，第一个用来构造 SQL 语句中 WHERE 的条件，第二个用来产生 SQL 语句的其他属性：比如 order by，group by 等，具体请参考
http://search.cpan.org/~nwiger/SQL-Abstract-1.22/lib/SQL/Abstract.pm

- update: 更新结果集的一些字段.
- delete: 删除结果集的一些记录.
- create: 创建一条新记录

下面是一些 DBIC 语句跟它产生的 SQL:

```
$schema->resultset('Book')->search(); 等同于$schema->resultset('Book')

 SELECT *  FROM BOOK;

$schema->resultset('Book')->search(
    {
        id => 2,
        name => { 'like' => '%perl' }
    }
);
SELECT *  FROM BOOK  WHERE  id=2 AND name like '%perl';

$schema->resultset('Book')->search(
    {
        id => {'-in' => [1,2,3]},
    }
);
SELECT *  FROM BOOK WHERE id IN (1,2,3);

$schema->resultset('Book')->search(
    {
        '-or' => [
            id => 1,
            name => 'perl',
        ],
    }
);
SELECT *  FROM BOOK WHERE id=1 OR name='perl';

$schema->resultset('Book')->search(
```

```
    {
       id => { '>' => 10}
    },
    {
        order_by => 'id desc'
    }
);
SELECT *  FROM BOOK WHERE id>=10 order by id desc;

$schema->resultset('Book')->search(
    {
        author => \"is not null",
    },
    {
        select => ['author', 'count(*)'],
        as => ['author', 'book_count'],
        group_by => 'author'
    }
);

SELECT author, count(*) FROM BOOK WHERE author is not null group by author;

$schema->resultset('Book')->update({ name => 'xiaosheng'});
UPDATE book SET name='xiaosheng';

$schema->resultset('Book')->search({ id => 2})->update({ name => 'xiaosheng'});
UPDATE book SET name='xiaosheng' WHERE id=2;

$schema->resultset('Book')->delete();
DELETE FROM book;

$schema->resultset('Book')->search( {name => 'perl'} )->delete();
DELETE FROM book WHERE name='perl';

$schema->resultset('Book')->create( {name => 'catalyst', author_id => 2} );
INSERT INTO book (author_id, name) values(2, 'catalyst');
```

现在大家可能会发现,上面的 SQL 全部是单表操作.
如果要实现从 Book 表 join 到 author 表, 那么首先要定义从 Book 到 Author 的关系.

DBIC 常用的关系有:
◆ belongs_to: 比如我们这里的 Book 表的 author_id 实际上是 Author 表的主键,那么可以定义 Book 到 Author 的 belongs_to 关系. 同理可以定义从 Chapter 到 Book 的 belongs_to 关系
◆ has_many: 跟 belongs_to 的关系正好反过来, 这里我们可以定义 Author 到 Book 的 has_many 的关系,也可以定义 Book 到 Chapter 的 has_many 关系.
◆ 其他的关系: might_have, many_to_many, has_one 等不在讨论范围内. 更详细的信息可参考 CPAN 文档.

注意我们如果要从 Book 表 join Author 表的话, 需要而且仅需要在 Book 里面定义 belongs_to 关系, 并不需要 Author 里面也定义关系.

那么现在我们在 Book 里面定义 belongs_to:

```
package MySchema::Book;
use base 'DBIx::Class';
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('book');
__PACKAGE__->add_columns(qw/id name author_id/);
__PACKAGE__->set_primary_key('id');
__PACKAGE__->belongs_to( 'author', 'MySchema::Author', 'author_id' );
1;
```

__PACKAGE__->belongs_to('author', 'MySchema::Author', 'author_id');
从这行代码里面, 可以得知
MySchema::Book belongs_to MySchema::Author
'author'指的是这个关系的名字,
'author_id' 指 book.author_id 是一个外键, 它是 MySchema::Author 的主键。

现在来测试一下这个关系：

```
#!perl
use strict;
use MySchema;
my $schema = MySchema->connect(
    'dbi:mysql:x;localhost',
    'root',
    '',
    { AutoCommit => 1 },
);
$schema->storage->debug(1);
my @book_list = $schema->resultset('Book')->search(
    {},
    {
        join =>['author']
    }
);

#打印出来的 SQL:

SELECT me.id, me.name, me.author_id FROM book me  JOIN author author ON ( author.id =
me.author_id ):


假设 book 表有 3 条记录

id, author_id, name
```

```
1, 1, 'book1'

2, 1, 'book2'

3, 2, 'book3'
```

假设 author 表有 2 条记录

```
id, is_fale, name
1,   1,  'author1'
2,   0,  'author2'
```

继续执行下面的代码：

```
for my $row_object_book (@book_list) {
    print 'book: ', $row_object_book->id, ', author name is ';
    print $row_object_book->author->name;
    print "\n";
}
```

会打印出来的 SQL：

```
SELECT me.id, me.name, me.is_male FROM author me WHERE ( ( ( me.id = ? ) ) ): '1'

SELECT me.id, me.name, me.is_male FROM author me WHERE ( ( ( me.id = ? ) ) ): '1'

SELECT me.id, me.name, me.is_male FROM author me WHERE ( ( ( me.id = ? ) ) ): '2'
```

我们发现每次执行$row_object_book->author->name 这行代码， 就会产生一条 SQL：
SELECT * FROM author WHERE id = ?
这条 SQL 只是为了得到 author 的 name 而已， 其实我们可以在把 book 跟 author join 起来的时候就把 author 的 name 取出来了，无需到后来才一条一条去取， 我们把上例中的 join 改成 prefetch：

```
my @book_list = $schema->resultset('Book')->search(
    {},
```

```
    {
        prefetch =>['author']
    }
);
```

#打印出来的 SQL:

```
SELECT me.id, me.name, me.author_id, author.id, author.name, author.is_male FROM book
me  JOIN author author ON ( author.id = me.author_id )
```

继续执行下面的代码:

```
for my $row_object_book (@book_list) {
    print 'book: ', $row_object_book->id, ', author name is ';
    print $row_object_book->author->name;
    print "\n";
}
```

不会再打印出 SQL， 因为我们上面的 prefetch 已经把 book 跟 author 一起取了出来，所以

$row_object_book->author->name; 不会再进行数据库查询

上例中，
$row_object_book 是一本书的 row object,
$row_object_book->author， 是这本书的 author 的 row object.
如果我们定义了从 book 到 chapter 的 has_many 关系（名为 chapters），那么，通过
$row_object_book->chapter 可以得到这个属于这本 book 的所有 chapter。

见下例:

```
package MySchema::Book;
use base 'DBIx::Class';
```

```
use strict;
use warnings;
__PACKAGE__->load_components(qw/PK::Auto Core/);
__PACKAGE__->table('book');
__PACKAGE__->add_columns(qw/id name author_id/);
__PACKAGE__->set_primary_key('id');
__PACKAGE__->belongs_to( 'author', 'MySchema::Author', 'author_id' );

__PACKAGE__->has_many( 'chapters', 'MySchema::Chapter', 'book_id' );
1;
```

继续测试一下：

```
假设 chapter 表有 3 条记录

id, book_id, name

1, 1, 'chapter1'
2, 1, 'chapter2'
3, 1, 'chapter3'


my $book = $schema->resultset('Book')->find(1);

#打印出来的 SQL:

SELECT me.id, me.name, me.author_id FROM book me WHERE ( ( me.id = ? ) ): '1'


继续执行下面的代码：
my $rs_chapters = $book->chapters;
```

```
while(my $row_object = $rs_chapters->next) {
    print $row_object->name, "\n";
}
#while 迭代之后打印出来的 SQL:
SELECT me.id, me.name, me.book_id, me.subject, me.content FROM chapter me WHERE ( me.book_id
= ? ): '1'



my @chapter_list = $book->chapters;

#打印一样出来的 SQL:

SELECT me.id, me.name, me.book_id, me.subject, me.content FROM chapter me WHERE ( me.book_id
= ? ): '1'

#如果要迭代每一个 chapter，要用 for 循环

for my $row_object (@chapter_list) {

    print $row_object->name, "\n";

}
```

上例中， $book->chapters 跟 resultset->search 类似， 它的返回值也是看上下文的，**在标量上下文它返回一个 resultset,在列表上下文它返回一个数组(也就是一个 RowObject 数组).**

关于 DBIx::Class::Schema 的介绍到此为止， 建议大家阅读 CPAN 上的 DBIC 文档。

第3节 .把 DBIC::Schema 作为 Model 使用

前面一节我们学习了 DBIx::Class::Schema, 它是由 MySchema.pm 跟 MySchema/*.pm 这些模块组成的，要在 Catalyst 里面使用这套模块， 只需要
1. 把 MySchema.pm 跟 MySchema/*.pm 拷贝到项目 X 的 lib 目录
2. 建立一个 X::Model::MyModel,使之继承自 Catalyst::Model::DBIC::Schema，并且使之跟 MySchema 关联起来。
3. 配置 Catalyst::Model::MyModel，比如数据库连接等。

现在继续拿出我们前面几章里面的示例项目: X, 键入命令:
perl x_create.pl Model MyModel DBIC::Schema MySchema
这条命令执行完, 会在 lib/X/Model/下面创建 X::Model::MyModel.pm，并且该 Model 跟 MySchema 是关联的，通过 MyModel 可以访问 MySchema 进而使用整套 DBIC::Schema.

打开 X::Model::MyModel.pm， 会看到：
里面有这样的代码：

```
__PACKAGE__->config(
    schema_class => 'MySchema',
);
```

这行代码把 MyModel 跟 MySchema 关联了起来。
跟第 4 章里面讲到的一样， Catalyst 的组件的配置可以有多种方法，这次我们还是选择老方法， 把配置写到 x.yml 里面， 所以我们把

```
__PACKAGE__->config(
    schema_class => 'MySchema',
);
```

这行代码从 Mydel.pm 里面删掉， 然后在 x.yml 里面写：

```
Model::MyModel:

  schema_class: MySchema
```

接下来配置这个 Model 的数据库连接参数,现在整个 x.yml 如下：

```
---

name: X

View::MyView:
```

```
  INCLUDE_PATH:
    - __HOME__/root/tt
  TEMPLATE_EXTENSION: .tt

Model::MyModel:

  schema_class: MySchema

  connect_info:
    - 'dbi:mysql:x;localhost'
    - 'root'
    - ''
    - AutoCommit: 1
      PrintError: 1

[这里有空行]
```

配置工作到此结束， 接下来我们结合之前用到的 **TT** 在 **action** 里面来使用这个 **Model:**

在 **root/tt/**下面建立一个 **list.tt,**内容如下**:**

```
hello, this is book list  <br>
[% FOREACH book IN book_list %]
  id: [% book.id %]<br>
  name: [% book.name %]<br>
  author_name: [% book.author.name %]<br>
  chapters of this book:<br>
        [% chapters = book.chapters %]
        <!-- now the chapters is array ref -->
        [% FOREACH chapter IN chapters %]
            [% chapter.name %]<br>
        [% END %]
 <hr>
[% END%]
```

在 Root.pm 里面添加 action 如下:

```
sub default : Private {
    my ( $self, $c ) = @_;
    $c->response->body( $c->welcome_message );
}

sub t :Local {
    my ( $self, $c ) = @_;
    $c->stash->{template} = 'test.tt';
}

sub list : Local {
    my ( $self, $c ) = @_;
    my @book_list = $c->model('MyModel')->resultset('Book')->search(
        {},
        {prefetch => ['author']}
    );
    $c->stash->{book_list} = \@book_list;
    $c->stash->{template} = 'list.tt';
}

sub end : Private  {
     my ( $self, $c ) = @_;
    $c->forward('View::MyView') unless $c->res->body;
}
```

启动我们的 x_server.pl,访问 http://xxxx/list

你会看到页面:

```
hello, this is book list
id: 1
name: book1
author_name: author1
chapters of this book:
chapter1
chapter2
```

```
chapter3
```

```
id: 2
name: book2
author_name: author1
chapters of this book:
```

```
id: 3
name: book3
author_name: author2
chapters of this book:
```

`$c->model('MyModel')` 返回的对象有点类似于上一节里面的`$schema` 对象，如果你想从 `Model` 得到它里面包含的 `schema` 对象，可以使用 `$c->model('MyModel')->schema`
`$c->model('MyModel')->resultset('Book')` 这行代码可以缩写为：
`$c->model('MyModel::Book')`

第 6 章　　调试 Catalyst 程序

通过 **x_server.pl** 调试 **Catalyst** 程序一般有 **2** 种方法：

1. 利用**$c->log** 是最常用的方法。
perl x_server.pl -d 会打开 **x_server** 的 **debug** 标志，
X 整个项目加载进来的时候，你可以从屏幕上看到相关的信息，
当有 **HTTP** 请求进来的时候， 你也可以从屏幕上看到 **action** 的跳转和执行。

你还可以往**$c->log** 里面写一些 **Debug** 信息， 然后通过这些输出的 **Debug** 信息， 进行分析。
比如想观察某个变量**$var** 的值，那么在 **action** 里面编写：
$c->log->debug($var);
或者
use Data::Dumper;
$c->log->debug(Dumper($var));

这样你可以从屏幕上看到**$var** 的值。

2. 给程序下断点进行调试。这个方法不是很常用。
在 **action** 里面加入： **$DB::single = 1;** 这行代码便可下一个断点。

假设现在的 **Root.pm** 如下：

```
package X::Controller::Root;
use strict;
use warnings;
use base 'Catalyst::Controller';
__PACKAGE__->config->{namespace} = '';
sub default : Private {
    my ( $self, $c ) = @_;
    $c->response->body( $c->welcome_message );
}
```

```
sub help : Local {
    my ( $self, $c ) = @_;

    my $var = { name => 'xiaosheng'};
    $DB::single = 1;
    $c->response->body( "What can I do for you?" );
}

sub end : ActionClass('RenderView') {}
1;
```

然后这样开启 x_server: perl -d x_server.pl
这个时候程序会停留在第一行待执行的代码上。

键入命令 c, 程序继续跑，直到整个 Catalyst 项目加载完毕， 这时 x_server 处于等待 HTTP 请求的状态。

访问 http://xxx/help, Catalyst 找到了 help 这个 action， 在执行这个 action 的时候， 当遇到 $DB::single = 1;这行代码时程序便停了下来， 这时如果要查看$var 的值，
那么键入命令： x $var
便可打印出$var 的值。
关于 perl debug 的详细命令，请参看相关文档， 这里稍微介绍几个：
c 是 continue 的意思， 让程序继续执行下面的代码
n 是 step over, next 的意思， 让程序执行当前这行代码
s 是 step into 的意思， 让程序继续跳进当前这行代码， 比如当前行代码是一个函数调用，那么 s 命令会跳入所调用的函数内部。