ASP.NET MVC

Title: Nancy框架
Date: 2014-09-12 23:39
Category: .Net
Tags: Nancy, C#, .Net, web
Author: 刘理想

[TOC]

Nancy框架

一、创建第一个Nancy应用

1. 安装Nancy项目模板
2. 创建`Nancy Empty Web Application with ASP.NET Hosting`
3. 添加`Nancy module`,它是一个标准C#类，通过添加下面几行代码定义了web应用的路由处理方法。
4. 编译并运行。

```
public class HelloModule : NancyModule
{
    public HelloModule()
    {
        Get["/"] = parameters => "Hello World";
    }
}
```

二、探索Nancy的module

Module继承自`NancyModule`类。Module是必不可少的.它不仅定义了路由，还提供了许多其他信息，比如请求、上下文、构造响应的辅助方法、视图渲染等等。

1. 模块能够在全局被发现

可以在任意地方定义module，比如外部的dll等，这为代码的复用带来很大的方便。不用担心效率问题，扫描module只在程序启动时发生。

2. 使用模块为路由创建一个根

类似命名空间的概念，在创建构造方法时传给base一个名称。

```
public class ResourceModule : NancyModule
```

```
{
    public ResourceModule() : base("/products")
    {
        // would capture routes to /products/list sent as a GET request
        Get["/list"] = parameters => {
            return "The list of products";
        };
    }
}
```

三、定义路由

路由是在module的构造方法中定义的。为了定义一个路由，你需要声明 `方法` + `模式` + `动作` +(可选) `条件`

比如：

```
public class ProductsModule : NancyModule
{
    public ProductsModule()
    {
        Get["/products/{id}"] = _ =>
        {
            //do something
        };
    }
}
```

或者异步

```
public class ProductsModule : NancyModule
{
    public ProductsModule()
    {
        Get["/products/{id}", runAsync: true] = async (_, token) =>
        {
            //do something long and tedious
        };
    }
}
```

1. 方法

支持HTTP常见方法：`DELETE`，`GET`，`HEAD`，`OPTIONS`，`POST`，`PUT`，`PATCH`

2. 模式

模式能够自定义，Nancy提供了一些常用的：

1. 字面量 - `/some/literal/segments`
2. 捕获片段 - `/{name}`，获取URL的片段，并传给路由的Action
3. 捕获可选片段 - `/{name?}`，添加了一个问号，片段就是可选的了
4. 捕获可选/默认片段 - `/{name?default}`
5. 正则片段 - `/(?<age>[\d]{1,2})`，使用命名捕获组来捕获片段，如果不需要捕获，使用非捕获组，比如`(?:regex-goes-here)`
6. 贪心片段 - `/{name*}`，从/处开始捕获
7. 贪心正则捕获 - `^(?<name>[a-z]{3, 10}(?:/{1})(?<action>[a-z]{5, 10}))$`
8. 多个捕获片段 - `/{file}.{extension}`或者`/{file}.ext`

3. 模式的优先级

4. 动作

动作时一个lambda表达式`Func<dynamic, dynamic>`，输入时`DynamicDictionary`，详见此处.

响应可以使任意的model，最终的结果会被Content Negotiation处理。但是如果返回值是`Response`类型，则原样返回。

`Response`对象有几个隐形转换操作：

1. `int`变为Http的状态
2. `HttpStatusCode`枚举值
3. `string`直接是相应的body
4. `Action<Stream>`则写道response stream中

5. 条件

路由条件用来过滤（比如登录非登录）。使用`Func<NancyContext, bool>`的lambda表达式定义.

```
Post["/login", (ctx) => ctx.Request.Form.remember] = _ =>
{
     return "Handling code when remember is true!";
}

Post["/login", (ctx) => !ctx.Request.Form.remember] = _ =>
{
     return "Handling code when remember is false!";
```

```
}
```

6. 路由片段约束

```
Get["/intConstraint/{value:int}"] = _ => "Value " + _.value + " is an integer.";
```

只有为int的才会匹配。

约束：

- `int`
- `decimal`
- `guid`
- `bool`
- `alpha`
- `datetime`
- `datetime(format)`
- `min(minimum)`
- `max(maximum)`
- `range(minimum, maximum)`
- `minlength(length)`
- `maxlength(length)`
- `length(minimum, maximum)`

6.1 自定义约束

实现`IRouteSegmentConstraint`接口，或者继承自

- `RouteSegmentConstraintBase<T>` - Base class for a named constraint.
- `ParameterizedRouteSegmentConstraintBase<T>` - Base class for a named constraint that accepts arguments.

例子

一个email约束

```
public class EmailRouteSegmentConstraint : RouteSegmentConstraintBase<string>
{
    public override string Name
    {
        get { return "email"; }
    }

    protected override bool TryMatch(string constraint, string segment, out string
matchedValue)
```

```
    {
        if (segment.Contains("@"))
        {
            matchedValue = segment;
            return true;
        }

        matchedValue = null;
        return false;
    }
}
```

用法

```
Get["/profile/{value:email}"] = _ => "Value " + _.value + " is an e-mail address.";
```

7. 选择去调用路由的秘诀

一个请求有时符合多个模式，此时记住：

1. module的顺序在启动时不定
2. 同一module中的路由是按顺序来的
3. 多个匹配中，得分最高的匹配
4. 得分相同的匹配按照启动时的顺序匹配

8. 疯狂的路由

一些可能的用法：

```
// would capture routes like /hello/nancy sent as a GET request
Get["/hello/{name}"] = parameters => {
    return "Hello " + parameters.name;
};

// would capture routes like /favoriteNumber/1234, but not /favoriteNumber/asdf as
a GET request
Get["/favoriteNumber/{value:int}"] = parameters => {
    return "So your favorite number is " + parameters.value + "?";
};

// would capture routes like /products/1034 sent as a DELETE request
Delete[@"/products/(?<id>[\d]{1,7})"] = parameters => {
    return 200;
```

```
};

// would capture routes like /users/192/add/moderator sent as a POST request
Post["/users/{id}/add/{category}"] = parameters => {
    return HttpStatusCode.OK;
};
```

四、自定义路由

http://www.phillipHaydon.com/2013/04/nancyfx-implementing-your-own-routing/

五、异步

1. 语法

Before/After管道、主路由委托都可以使用async.语法绝大部分与同步代码一致，但需要注意下面的变化：

- before/after钩子接受两个参数，context和cancellation token(取消令牌)，而不仅仅是context
- 路由定义有一个附加的bool参数，并且委托接受两个参数，一个捕获的参数，另一个cancellation token.

2 语法例子

```
public MainModule()
{
    Before += async (ctx, ct) =>
        {
            this.AddToLog("Before Hook Delay\n");
            await Task.Delay(5000);

            return null;
        };

    After += async (ctx, ct) =>
        {
            this.AddToLog("After Hook Delay\n");
            await Task.Delay(5000);
            this.AddToLog("After Hook Complete\n");

            ctx.Response = this.GetLog();
        };
```

```
    Get["/", true] = async (x, ct) =>
        {
            this.AddToLog("Delay 1\n");
            await Task.Delay(1000);

            this.AddToLog("Delay 2\n");
            await Task.Delay(1000);

            this.AddToLog("Executing async http client\n");
            var client = new HttpClient();
            var res = await client.GetAsync("http://nancyfx.org");
            var content = await res.Content.ReadAsStringAsync();

            this.AddToLog("Response: " + content.Split('\n')[0] + "\n");

            return (Response)this.GetLog();
        };
}
```

六、查看DynamicDictionary

`DynamicDictionary`类似字典，但功能更多.从请求中获取的值都保存到它里面。可以使用属性或者index来使用捕获的值。

```
Get["/hello/{name}"] = parameters => {
    return "Hello " + parameters.name;
};

Get["/goodbye/{name}"] = parameters => {
    return "Goodbye " + parameters["name"];
};
```

存储的值可以显示或者隐式的转换为基础类型或者特殊属性.使用`HasValue`决定是否被赋值。值已经实现了`IEquatable<>`和`IConvertible`接口。

七、module的before/after钩子

除了为特定的路由定义处理程序,module还可以拦截匹配某个路由的请求,请求前后都能做到。重要的是要理解,只有传入的请求匹配模块的路由之一，这些拦截器才会被调用。

1. 在路由被调用前拦截请求

Before拦截器能让你修改请求，甚至可以通过返回一个response来放弃请求。

```
Before += ctx => {
    return <null or a Response object>;
};
```

定义Before拦截器的语法与定义路由有些不同。因为它是定义在module上，被所有路由调用，所以不需要匹配模式。

传给拦截器的是当前请求的NancyContext实例。

最后的不同就是拦截器的返回值，如果返回`null`，拦截器将主动权转给路由；如果返回`Response`对象，则路由不起作用。

2. After拦截器

与定义Before烂机器相同，但是没有返回值。

```
After += ctx => {
    // Modify ctx.Response
};
```

Before拦截器可以修改Request，相应的，After拦截器可以修改Response。

八、Application的Before,After和OnError管道

应用管道能在所有的路由上执行，是全局性的。

1.Before拦截

应用级的`Before`钩子通过`Func<NancyContext, Response>`函数定义：

```
pipelines.BeforeRequest += (ctx) => {
    return <null or a Response object>;
};
```

异步版本的：

```
pipelines.BeforeRequest += async (ctx, token) => {
    return <null or a Response object>;
};
```

2. After拦截

After拦截器通过`Action<NancyContext>定义：

```
pipelines.AfterRequest += (ctx) => {
    // Modify ctx.Response
};
```

3. 错误拦截器

`OnError`拦截器用来拦截路由发生的错误。通过它可以获取`NancyContext`和发生的异常。

`OnError`拦截器通过`Func<NancyContext, Exception, Response>`函数定义：

```
pipelines.OnError += (ctx, ex) => {
    return null;
};
```

System.AggregateExceptions在OnError管道中的注意事项：

路由是通过许多嵌套的Task(`System.Threading.Tasks.Task`)来执行的。如果那个任务出现了问题，异常会被包装到`System.AggregateException`。`System.AggregateException`可以持有任意个异常。

如果只有一个异常，Nancy会解包异常并且交给`OnError`管道。如果发生多个异常，Nancy会使用`System.AggregateException`，以避免吞异常。

4. 构建自己的钩子

在Bootstrapper中创建系统级的钩子.可以在`ApplicationStartup`或者`RequestStartup`方法中定义它们。这是因为也许你需要在钩子中使用容器中的一些东西。两个方法的不同之处在于范围不同。

```
protected override void ApplicationStartup(TinyIoCContainer container, IPipelines p
ipelines)
{
}

protected override void RequestStartup(TinyIoCContainer requestContainer, IPipeline
s pipelines, NancyContext context)
{
}
```

通过使用`pipelines`中适当的属性来创建钩子。它允许你获取`BeforeRequest`，`AfterRequest`和`OnError`属性。

九、模型绑定

发送数据给Nancy可以有多种方法，比如Query String, 路由捕获参数、请求体request body。手工处理这些不同的方法也可以，但是还有一种方法就是统一处理，绑定到`model`。

Nancy只用一行代码就能处理上述的所有情况，并且能接受`JSON`和`XML`形式的请求。

也可以扩展Nancy的模型绑定。

Nancy的模型绑定在`NancyModule`中被定义为一个单独的扩展方法。该扩展在`Nancy.ModelBinding`命名空间里，并且添加了Bind()和BindTo()方法

```
Foo f = this.Bind();

var f = this.Bind<Foo>();

var f = this.BindTo(instance);
```

上面3个有着相同的功能，他们提供了做同一事物的不同方法。前两个使用Bind()重载来创建`Foo`类型的实例，并且绑定；BindTo()则绑定到现有实例。

1. 屏蔽不想要的信息

```
var f = this.Bind<Foo>(f => f.id, f => f.creator, f => f.createddate);
```

或者

```
var f = this.Bind<Foo>("id", "creator", "createddate");
```

当绑定到到arrary, list或者ienumerable时，屏蔽的是序列中的元素。

2. 绑定配置

使用`BindingConfig`实例来修改model binder的默认行为。

下面是`BindingConfig`提供的一些配置项：

属性	描述	默认
BodyOnly	是否只绑定request body。这种情况下，request和context参数都不会被绑定。如果没有body并且没有选项，那么绑定就不会放生	false
IgnoreErrors	是否忽略绑定错误并且继续下一个属性	false
Overwrite	丙丁是否可以覆盖没有默认值的属性	true

不准Overwrite还有一个快捷方法：`BindingConfig.NoOverwrite`

3. 反序列化rich request body payloads(负载)

有时你像在请求中发送结构化的数据，比如`JSON`或者`XML`，并且绑定到模型。模型绑定器支持这种反序列化。

Nancy支持两种反序列化：JSON和XML。绑定器根据Http的`Content-type`头来决定使用哪一种反序列化。

默认使用JSON反序列化来处理`application/json`，`text/json`和`application/vnd....+json`。同样的使用XML反序列化来处理`application/xml`，`text/xml`和`application/vnd....+xml`

对于其他模型绑定器，你可以使用自己的反序列化，并且Nancy会自动检测他们，任何用户定义的绑定器的优先级都高于内建的。
注意：如果你使用Nancy.Json.JsonSetting.MaxJsonLength Exceeded错误，那是因为你的payloads太高了，在Bootstrapper中更改限制：`ApplicationStartup`中设置`Nancy.Json.JsonSettings.MaxJsonLength=int.MaxValue`

4. 模型绑定Checkbox

要绑定复选框到bool值，确定设置`value=true`：

```
<input type="checkbox" name="rememberMe" value="true"/>
```

```
public class LoginModel
{
    public bool RememberMe { get; set; }
}
```

5. 绑定到list

5.1 绑定arrary到单独的对象

如果有一个form:

```
<form action="/ArrayOnObject" method="post">
  <input type="text" name="Tags" value="Tag1,Tag2,Tag3"/>
  <input type="text" name="Ints" value="1,2,3,4,4,5,6,3,2,21,1"/>
  <input type="submit" value="Submit"/>
</form>
```

而且有一个类：

```
public class Posts
{
  public string[] Tags { get; set; }
  public int[] Ints { get; set; }
}
```

使用一个简单的语句：

```
var listOfPosts = this.Bind<Posts>();
```

5.2 绑定到对象的list

```
<form action="/SimpleListDemo" method="post">
      User 1:<input type="text" name="Name[0]" value="thecodejunkie" />
      Commits <input type="text" name="Commits[0]" value="1068"/>
      <br />
      User 2:<input type="text" name="Name[1]" value="grumpydev" />
      Commits <input type="text" name="Commits[1]" value="1049"/>
      <br />
      User 3:<input type="text" name="Name[2]" value="jchannon" />
      Commits <input type="text" name="Commits[2]" value="109"/>
      <br />
      User 4:<input type="text" name="Name[3]" value="prabirshrestha" />
      Commits <input type="text" name="Commits[3]" value="75"/>
      <br />
      User 5:<input type="text" name="Name[4]" value="phillip-haydon" />
      Commits <input type="text" name="Commits[4]" value="40"/>
      <br />
      <input type="submit" value="Test the binding thingy"/>
</form>
```

可以使用`this.Bind<List<User>>();`来绑定对象列表：

```
public class User
{
   public string Name { get; set; }
   public int Commits { get; set; }
}
```

5.3 HTML form中的List分隔符

两种分隔符

- 下划线(`Name_1`，`Name_2`等)
- 括号(`Name[1]`，`Name[2]`等)

十、Bootstrapper

bootstrapper负责自动发现模型、自定义模型绑定、依赖等等。可以被替换掉。

1. 简单的修改bootstrapper

```
public class CustomBootstrapper : DefaultNancyBootstrapper
{
    protected override void ApplicationStartup(TinyIoCContainer container, IPipelin
es pipelines)
    {
         // your customization goes here
    }
}
```

2. 找到合适的bootstrapper

应用启动时，它会寻找自定义的bootstrap，如果没有找到，则使用`DefaultNancyBootstrap`。每个应用只能有**一个**bootstrapper. 如果有多个，则Nancy寻找最底层的bootstrapper。

3. 使用自动注册

注入自己的依赖到NancyModule中

```
public class Home : NancyModule
{
    public Home(IMessageService service)
    {
        //If there is only one implementation of IMessageService in the application
,
        // TinyIoC will resolve the dependency on its own and inject it in the modu
le.
    }
}
```

十一、视图引擎

视图引擎就是输入“模板”和“模型”，输出HTML（大部分情况下）到浏览器。

Nancy默认使用`SuperSimpleViewEngine`。它支持一些必要的功能：layout布局、partials部分、models模型、conditions条件和iterations循环。你可以使用这个而不无需其他依赖。它支持`.html`和`.sshtml`文件。

```
@Master['MasterPage']

@Section['Content']
    <p>This content from the index page<p>
    <h3>Partials</h3>
    <p>Login box below rendered via a partial view with no model.</p>
    <div id="login">
        @Partial['login'];
    </div>
    <p>Box below is rendered via a partial with a sub-model passed in.</p>
    <p>The submodel is a list which the partial iterates over with Each</p>
    <div id="users">
        @Partial['user', Model.Users];
    </div>
    <h3>Encoding</h3>
    <p>Model output can also be encoded:</p>
    <p>@!Model.NaughtyStuff</p>
@EndSection
```

除此之外，Nancy还支持Razor, Spark, NDjango和dotLiquid引擎。通过添加引用，Nancy会自动的根据文件后缀名调用对应的引擎。

1. 在路由中渲染视图

```
Get["/products"] = parameters => {
    return View["products.html", someModel];
};
```

模板说明：

1. 视图文件名: "products.html"
2. 如果没有后缀，而且有多个同名模板，则会收到`AmbigiousViewsException`错误。
3. 一个相对于跟的路径(比如：`products/products.html`)

更多参见视图位置约定

2.从模型中解析视图的名称

如果值传递给View一个模型，Nancy会用模型名（去掉"Model"后缀）作为视图名。

```
Get["/products"] = parameters => {
    return View[new ProductsModel()];
};
```

如果找不到，就会报406 Not Acceptable.

十二、超简单视图引擎

SSVE基于正则，支持`sshtml`，`html`，`html`文件后缀。

模型可以是标准类型，或者`ExpandoObjects`（或者实现了`IDynamicMetaObjectProvider`实现了`IDictionary<string, object>`的对象）。

所有的命令都可以有分号，但不是必须的。`[.Parameters]`这样的参数可以使任意层级的，比如`This.Property.That.Property`。

注意：所有引号都是单引号.

1. 标准变量替换

如果变量不能替换，则使用`[Err!]`替换。

语法：

```
@Model[.Parameters]
```

例子：

```
Hello @Model.Name, your age is @Model.User.Age
```

2. 循环

循环不能嵌套

语法：

```
@Each[.Parameters]
   [@Current[.Parameters]]
@EndEach
```

`@Each`表示循环；`@Current`表示当前变量，使用方法同`@Model`。

例子：

```
@Each.Users
   Hello @Current.Name!
@EndEach
```

3. 条件

参数必须是bool，或能隐式转化。嵌套的@If @IfNot不支持。

语法：

```
@If[Not].Parameters
   [contents]
@EndIf
```

例子：

```
@IfNot.HasUsers
   No users found!
@EndIf
```

4. 隐式条件

如果module实现了`ICollection`，那你就能使用隐式转换。使用`Has`前缀。

语法：

```
Has[CollectionPropertyName]
```

例子：

```
@If.HasUsers
   Users found!
@EndIf
```

5. HTML编码

`@Model`和`@Current`都可以有一个`!`，用来编码HTML：

语法：

```
@!Model[.Parameter]
```

```
@!Current[.Parameter]
```

例子：

```
@!Model.Test

@Each
   @!Current.Test
@EndEach
```

6. 部分Patials

语法：

```
@Partial['<view name>'[, Model.Property]]
```

例子：

```
// Renders the partial view with the same model as the parent
@Partial['subview.sshtml'];

// Renders the partial view using the User as the model
@Partial['subview.sshtml', Model.User];
```

7. Master页和section

可以声明master页和节。不必为每个节提供内容。Master能用`@Module`，并且扩展名可以省略。

可以多次使用`@Section`

语法

```
@Master['<name>']

@Section['<name>']
@EndSection
```

例子：

```
// master.sshtml
<html>
<body>
```

```
@Section['Content'];
</body>
</html>

// index.sshtml
@Master['master.sshtml']

@Section['Content']
   This is content on the index page
@EndSection
```

8. 防止伪造token

防止CSRF

语法：

```
@AntiForgeryToken
```

例子：

```
@AntiForgeryToken
```

9. 路径扩展

扩展相对路径为整体路径。

语法：

```
@Path['<relative-path>']
```

例子：

```
@Path['~/relative/url/image.png']
```

10. 扩展SSVE

十二、Razor引擎

这个Razor引擎跟ASP.NET MVC的有点不一样。

注意，Nancy仍然绑定模型到`@Model`，而不是ASP.NET中的`@model`

1. 安装Razor

只需要添加`Nancy.ViewEngines.Razor.dll`（使用nuget安装`Nancy.ViewEngines.Razor`）。然后试图模板以`cshtml`或`vbhtml`结尾即可。

2. 配置Razor

十三、实现自己的视图引擎需要注意的地方

十四、视图位置约定

1. 查看默认约定

视图位置的约定通过`Func<string, dynamic, ViewLocationContext, string>`方法以及下面的一些默认约定来定义。

1.1 根约定

```
(viewName, model, viewLocationContext) => {
    return viewName;
}
```

这个约定会在根目录里寻找视图。但是如果视图包含一个相对路径，视图名称执行对应于根路径的路径。比如，视图`admin/index`会在`admin/index`目下寻找视图。

1.2 视图文件夹约定

```
(viewName, model, viewLocationContext) => {
    return string.Concat("views/", viewName);
}
```

很简单，视图`admin/index`会在`views/admin/index`下查找对应的视图。

1.3 视图和模块路径约定

```
(viewName, model, viewLocationContext) => {
    return string.Concat("views/", viewLocationContext.ModulePath, "/", viewName);
}
```

对于模块products的视图`admin/index`，会在`views/products/admin/index`中查找视图。

1.4 模块路径约定

```
(viewName, model, viewLocationContext) => {
    return string.Concat(viewLocationContext.ModulePath, "/", viewName);
}
```

这个约定会在与模块名相同的文件夹中查找视图。

1.5 模块名称约定

```
(viewName, model, viewLocationContext) => {
    return string.Concat(viewLocationContext.ModuleName, "/", viewName);
}
```

查找以模块名为前缀的对应视图。

1.6 视图模块名称约定

```
(viewName, model, viewLocationContext) => {
    return string.Concat("views/", viewLocationContext.ModuleName, "/", viewName);
}
```

查找views文件夹下以模块名为前缀的对应视图。

2. 从模型类型推断是退名

如果没有提供视图名而只提供了视图，那么：

- `Customer`类型的模型->`Customer`视图名
- `CustomerModel`类型的模型->`Customer`视图名

3. 自定义约定

自定义一个bootstrapper，然后添加约定到`Conventions.ViewLocationConventions`集合。

比如：

```
public class CustomConventionsBootstrapper : DefaultNancyBootstrapper
{
    protected override void ApplicationStartup(TinyIoCContainer container, Nancy.Bo
```

```
otstrapper.IPipelines pipelines)
    {
        this.Conventions.ViewLocationConventions.Add((viewName, model, context) =>
        {
            return string.Concat("custom/", viewName);
        });
    }
}
```

比如这个会查找custom文件夹下的视图名称。

`ViewLocationConventions`是一个标准的列表，可以进行修改。

3. 使用IConventions定义自己的约定

你也可以实现`IConvention`接口，并在`Initialise`方法中添加约定到`ViewLocationConventions`属性中。

Nancy会定位所有接口的实现，并且执行约定，这些发生在他们被传递给bootstrapper的`ConfigureConventions`方法之前。

十五、本地化

Nancy内建了本地化。有一系列的约定描述了如何决定当前文化，还有一些根据文化选择视图的约定。

所以，对于`de-DE`的文化他会寻找`Home-de-DE`的视图。

不仅如此，还会有rese文件，比如`Text.resx`， `Text.de-DE.resx`（可以被重写).

Razor本地化的例子

十六、测试应用

使用NuGet来安装`Nancy.Testing`。

测试应当与主应用分开。

为了测试路由，使用helper类`Browser`。使用bootstrap实例化Browser。

```
[Fact]
public void Should_return_status_ok_when_route_exists()
{
    // Given
    var bootstrapper = new DefaultNancyBootstrapper();
    var browser = new Browser(bootstrapper);

    // When
```

```
    var result = browser.Get("/", with => {
        with.HttpRequest();
    });

    // Then
    Assert.Equal(HttpStatusCode.OK, result.StatusCode);
}
```

十七、根路径

Nancy通过`IRootPathProvider`接口的唯一方法`GetRootPath`来确定根路径。

1. 改变跟路径

改变根路径需要做两件事：

首先，自定义一个类实现`IRootPathProvider`：

```
public class CustomRootPathProvider : IRootPathProvider
{
    public string GetRootPath()
    {
        return "What ever path you want to use as your application root";
    }
}
```

注意，根路径是绝对路径。

其次，在自定义的Bootstrapper中重写`RootPathProvider`属性。

```
public class CustomBootstrapper : DefaultNancyBootstrapper
{
    protected override IRootPathProvider RootPathProvider
    {
        get { return new CustomRootPathProvider(); }
    }
}
```

2. 上传文件

在Nancy中要上传文件，你需要接受上传文件的content stream, 在磁盘上创建文件，并将stream写入到磁盘。

```
var uploadDirectory =  Path.Combine(pathProvider.GetRootPath(), "Content", "uploads
");

if (!Directory.Exists(uploadDirectory))
{
    Directory.CreateDirectory(uploadDirectory);
}

foreach (var file in Request.Files)
{
    var filename = Path.Combine(uploadDirectory, file.Name);
    using (FileStream fileStream = new FileStream(filename, FileMode.Create))
    {
        file.Value.CopyTo(fileStream);
    }
}
```

上例中的`pathProvider`是在模块的构造函数中传递进来的，通过它的`GetRootPath()`来获取跟路径。

```
public HomeModule(IRootPathProvider pathProvider)
```

十八、管理静态内容

简而言之：把东西都放到`/Content`文件夹内，仅此而已

十九、诊断

Nancy自带诊断功能：`http://<address-of-your-application>/_Nancy/`

1. 配置到dashboard的访问

添加密码：

```
public class CustomBootstrapper : DefaultNancyBootstrapper
{
    protected override DiagnosticsConfiguration DiagnosticsConfiguration
    {
        get { return new DiagnosticsConfiguration { Password = @"A2\6mVtH/XRT\p,B"}
; }
    }
}
```

2. 去除诊断

```
public class CustomBootstrapper : DefaultNancyBootstrapper
{
    protected override void ApplicationStartup(TinyIoc.TinyIoCContainer container,
IPipelines pipelines)
    {
        DiagnosticsHook.Disable(pipelines);
    }
}
```

3. 有哪些工具呢？

`Information`, `Interactive Diagnostics`, `Request Tracing`, `Configuration`

3.1 信息

3.2 配置

Nancy中`StaticConfiguration`可以用来配置程序的行为，配置页面提供了配置方法。

注意，系统重启后配置页面的内容失效。

要想永久保存配置，请在bootstrapper的`ApplicationStartup`中设置。

3.3 请求跟踪

请求跟踪因为性能原因默认关闭，可以再`Configuration`页开启，也可以这样：

```
public class CustomBootstrapper : DefaultNancyBootstrapper
{
    protected override void ApplicationStartup(TinyIoC.TinyIoCContainer container,
IPipelines pipelines)
    {
        StaticConfiguration.EnableRequestTracing = true;
    }
}
```

跟踪日志可以通过`NancyContext`中得到。和容易添加自己的内容：

```
public class HomeModule : NancyModule
{
```

```
    public HomeModule()
    {
        Get["/"] = parameters => {
            this.Context.Trace.TraceLog.WriteLog(s => s.AppendLine("Root path was c
alled"));
            return HttpStatusCode.Ok;
        };
    }
}
```

`WriteLog`方法是用一个接受`StringBuilder`的函数是为了调试关闭时直接不调用函数，从而避免性能损耗。

3.4 交互式的诊断

只要实现了`IDiagnosticsProvider`接口，Nancy诊断会自动发现它，并且把它暴露给交互工具。

（1）IDiagnosticsProvider接口

```
/// <summary>
/// Defines the functionality a diagnostics provider.
/// </summary>
public interface IDiagnosticsProvider
{
    /// <summary>
    /// Gets the name of the provider.
    /// </summary>
    /// <value>A <see cref="string"/> containing the name of the provider.</value>
   string Name { get; }

   /// <summary>
   /// Gets the description of the provider.
   /// </summary>
   /// <value>A <see cref="string"/> containing the description of the provider.</v
alue>
   string Description { get; }

   /// <summary>
   /// Gets the object that contains the interactive diagnostics methods.
   /// </summary>
   /// <value>An instance of the interactive diagnostics object.</value>
   object DiagnosticObject { get; }
}
```

（2）可诊断的对象

任何公共方法都会暴露给交互诊断面板。方法可以是能被JSON序列化的任意类型。类型的返回值会被返回成JSON Report Format

（3）提供描述给方法

两种方法：
1、使用attribute: `Nancy.Diagnostics.DescriptionAttribute`

2、使用property：使用与方法同名但添加了`Description`后缀的属性，比如`NameOfYourMethodDescription`描述了`NameOfYourMethod`方法。

（4）自定义模板输出

（5）创建诊断提供者

二十、添加自己的favicon

1. 替换默认的FavIcon

在应用中防止一个*favicon*的文件，名称以`.icon`或`.png`结尾即可。

2. 使用内嵌icon

在Bootstrapper中重写`FavIcon`属性：

```
public class Bootstrapper : DefaultNancyBootstrapper
{
    private byte[] favicon;

    protected override byte[] FavIcon
    {
        get { return this.favicon?? (this.favicon= LoadFavIcon()); }
    }

    private byte[] LoadFavIcon()
    {
        //TODO: remember to replace 'AssemblyName' with the prefix of the resource
        using (var resourceStream = GetType().Assembly.GetManifestResourceStream("A
ssemblyName.favicon.ico"))
        {
            var tempFavicon = new byte[resourceStream.Length];
            resourceStream.Read(tempFavicon, 0, (int)resourceStream.Length);
            return tempFavicon;
        }
```

```
    }
}
```

3. 移除ICON

设置Bootstrapper的`FavIcon`属性为`null`。

二十一、添加自定义的错误页面

第一篇:http://mike-ward.net/blog/post/00824/custom-error-pages-in-nancyfx

第二篇：https://blog.tommyparnell.com/custom-error-pages-in-nancy/

二十二、加密帮助方法

命名空间:`Nancy.Cryptography`

1. IEncryptionProvider 接口

```
/// <summary>
/// Provides symmetrical encryption support
/// </summary>
public interface IEncryptionProvider
{
    /// <summary>
    /// Encrypt and base64 encode the string
    /// </summary>
    /// <param name="data">Data to encrypt</param>
    /// <returns>Encrypted string</returns>
    string Encrypt(string data);

    /// <summary>
    /// Decrypt string
    /// </summary>
    /// <param name="data">Data to decrypt</param>
    /// <returns>Decrypted string</returns>
    string Decrypt(string data);
}
```

Nancy提供了两个默认实现

- `NoEncryptionProvider`:没有加密，仅仅是base64
- `RijndaelEncryptionProvider`: 使用Rijndael算法，使用256位的key和128为的初始向量，加密base64字符串。

2. IHmacProvider 接口

用来签名，防止篡改。

```
/// <summary>
/// Creates Hash-based Message Authentication Codes (HMACs)
/// </summary>
public interface IHmacProvider
{
    /// <summary>
    /// Gets the length of the HMAC signature in bytes
    /// </summary>
    int HmacLength { get; }

    /// <summary>
    /// Create a hmac from the given data
    /// </summary>
    /// <param name="data">Data to create hmac from</param>
    /// <returns>Hmac bytes</returns>
    byte[] GenerateHmac(string data);

    /// <summary>
    /// Create a hmac from the given data
    /// </summary>
    /// <param name="data">Data to create hmac from</param>
    /// <returns>Hmac bytes</returns>
    byte[] GenerateHmac(byte[] data);
}
```

Nancy也提供了一个默认实现：`DefaultHmacProvider`，使用`IKeyGenerator`来产生一个key来用SHA-256来进行hash。

3. IKeyGenerator 接口

用来产生key来加密和数字签名。

```
/// <summary>
/// Provides key byte generation
/// </summary>
```

```
public interface IKeyGenerator
{
    /// <summary>
    /// Generate a sequence of bytes
    /// </summary>
    /// <param name="count">Number of bytes to return</param>
    /// <returns>Array <see cref="count"/> bytes</returns>
    byte[] GetBytes(int count);
}
```

Nancy提供了两个默认实现。

- `RandomKeyGenerator`使用`RNGCryptoServiceProvider`产生了一个随机定长的key
- `PassphraseKeyGenerator`使用密码、静态盐以及可选循环数字，以及`Rfc2898DeriveBytes`来产生一个key

注意，如果使用`PassphraseKeyGenerator`，它的初始化应当在应用启动时使用，因为它太慢了。这意味着盐是静态的，因此密码一定要足够长和复杂。

4. 加密配置类型CryptographyConfiguration

这是一个存储`IEncryptionProvider`和`IHmacProvider`的简便方法。它有两个静态属性：

- `Default`使用`RijndaelEncryptionProvider`和`DefaultHmacProvider`，两个都使用`RandomKeyGenerator`。
- `NoEncryption`使用`NoEncryption`和`DefaultHmacProvider`，两个也都使用`RandomKeyGenerator`.

可以单独使用`CryptographyConfiguration`，也可以在bootstrapper中配置一个：

```
/// <summary>
/// Gets the cryptography configuration
/// </summary>
protected virtual CryptographyConfiguration CryptographyConfiguration
{
    get { return CryptographyConfiguration.Default; }
}
```

二十三、Content negotiation(内容协商)

当返回不是`Response`类型时，使用response processor来根据请求的`Accept`来处理。

1. Response Processor

```
public interface IResponseProcessor
{
    /// <summary>
    /// Gets a set of mappings that map a given extension (such as .json)
    /// to a media range that can be sent to the client in a vary header.
    /// </summary>
    IEnumerable<Tuple<string, MediaRange>> ExtensionMappings { get; }

    /// <summary>
    /// Determines whether the the processor can handle a given content type and mo
del.
    /// </summary>
    ProcessorMatch CanProcess(MediaRange requestedMediaRange, dynamic model, NancyC
ontext context);

    /// <summary>
    /// Process the response.
    /// </summary>
    Response Process(MediaRange requestedMediaRange, dynamic model, NancyContext co
ntext);
}
```

Response Processor是自发现的，也可以在Bootstrap中配置。

```
public class Bootstrapper : DefaultNancyBootstrapper
{
    protected override NancyInternalConfiguration InternalConfiguration
    {
        get
        {
            var processors = new[]
            {
                typeof(SomeProcessor),
                typeof(AnotherProcessor)
            };

            return NancyInternalConfiguration.WithOverrides(x => x.ResponseProcesso
rs = processors);
        }
    }
}
```

1.1 匹配优先级

当相应准备转化请求媒体的格式时，Nancy会查询所有的processor的`CanProcess`方法，并且会聚合`ProcessorMatch`的返回值。

`ProcessorMatch`类型确保每个processor让Nancy知道它们对媒体类型的支持程度。

```
public class ProcessorMatch
{
    /// <summary>
    /// Gets or sets the match result based on the content type
    /// </summary>
    public MatchResult RequestedContentTypeResult { get; set; }

    /// <summary>
    /// Gets or sets the match result based on the model
    /// </summary>
    public MatchResult ModelResult { get; set; }
}
```

`MatchResult`枚举了匹配程度：

```
public enum MatchResult
{
    /// <summary>
    /// No match, nothing to see here, move along
    /// </summary>
    NoMatch,

    /// <summary>
    /// Will accept anything
    /// </summary>
    DontCare,

    /// <summary>
    /// Matched, but in a non-specific way such as a wildcard match or fallback
    /// </summary>
    NonExactMatch,

    /// <summary>
    /// Exact specific match
    /// </summary>
    ExactMatch
}
```

所有的`ProcessorMatch`会按照Match程度降序排列，最匹配的被执行。如果有两个匹配程度相同，Nancy会选择其中一个。

1.2 默认响应处理器

Nancy提供了一些默认响应处理器

- `JsonProcessor` - 当请求类型为`application/json`或者`application/vnd.foobar+json`时，转化返回值为json；
- `ViewProcessor` - 当请求类型为`text/html`时，使用返回值作为model，返回视图。视图使用视图位置约定；
- `XmlProcessor` - 当请求为`application/xml`或者为`application/vnd.foobar+xml`时，返回xml。

2. 控制协商

`Nancy.Responses.Negotiation`命名空间中的`Negotiator`用来控制协商。`Negotiator`有一个属性：`NegotiationContext`. `NegotiationContext`可以用来控制响应的协商。

但是一般不会直接使用`Negotiator`和`NegotiationContext`，因为`NancyModule`包含了一个帮助方法`Negotiate`，用来更好的创造`Negotiator`实例。

在路由中使用`Negotiator`的例子：

```
Get["/"] = parameters => {
    return Negotiate
        .WithModel(new RatPack {FirstName = "Nancy "})
        .WithMediaRangeModel("text/html", new RatPack {FirstName = "Nancy fancy pan
ts"})
        .WithView("negotiatedview")
        .WithHeader("X-Custom", "SomeValue");
};
```

`Negotiator`包含了用来配置返回`Negotiator`实例的一些方法。

- `WithHeader` - 添加一个Http头；
- `WithHeaders` - 添加一个Http的头集合；
- `WithView` - 使用视图；
- `WithModel` - 使用模型；
- `WithMediaRangeModel` - 使用特定的媒体类型和模型，如果失败了，就使用`WithModel`指定的模型；
- `WithFullNegotiation` - 设置允许媒体类型为`*/*`的帮助方法；
- `WithAllowedMediaRange` - 指定允许的媒体范围。默认是"/",但是一旦指定一个特定的内容类型，通配符就会被移走。
- `WithStatusCode` - 状态码

3. 支持文件扩展名

Nancy支持基于扩展名来设置协商的处理，此时传递正常的可接受的头。

例子：

```
Get["/ratpack"] = parameters => {
    return new RatPack {FirstName = "Nancy "});
};
```

它既可以通过`/ratpack`和设置的`application/json`头来调用，也可以使用`/ratpack.json`并且不设置`application/json`来调用，两个结果一样。

内部Nancy是通过检测扩展名，并查询可用的响应处理器的`ExtensionMappings`属性来查看是否有支持的扩展。如果有，就调用并且设置对应的头信息，但是如果有更优先的处理器，则用更优先的处理器，除非更优先的处理器失败了，才会使用扩展。

4. 强制可接受的头(Accept Header)

约定的格式：

```
Func<
   IEnumerable<Tuple<string, decimal>>,
   NancyContext,
   IEnumerable<Tuple<string, decimal>>>
```

这个函数接受`NancyContext`和当前头，并且期望你返回修改后的可接受头列表。

默认情况下，Nancy在`Nancy.Conventions.BuiltInAcceptHeaderCoercions class`中提供了如下约定，其中加*的表示是默认默认被转换的：

- `BoostHtml` (*) - 如果text/html的优先级低于其他内容类型，则提高优先级；
- `CoerceBlankAcceptHeader` (*) - 如果没有指定请求头，就分配一个默认的；
- `CoerceStupidBrowsers` - 对于老浏览器，替换请求头，即使它们说是请求xml还是返回html。

更改哪一个强制起作用时在bootstrapper中的`ConfigureConventions`来设置的：

```
public class Bootstrapper : DefaultNancyBootstrapper
{
    protected override void ConfigureConventions(NancyConventions nancyConventions)
    {
        base.ConfigureConventions(nancyConventions);

        this.Conventions.AcceptHeaderCoercionConventions.Add((acceptHeaders, ctx) =
> {

            // Modify the acceptHeaders by adding, removing or updating the current
            // values.
```

```
            return acceptHeaders;
        });
    }
}
```

当然你也可以继承你自己的bootstrapper。

5. 使用IConventions来定义自己的约定

可以通过实现`IConventions`接口来创造一个类，并在它的`Initialise`方法中添加自己的约定到传递进来的参数的`AcceptHeaderCoercionConventions`属性中。

在所有的接口被传递给bootstrapper的`ConfigureConventions`的方法之前，Nancy会定位所有的接口实现，并且激发这些约定。

6. 自动协商头

Nancy会自动添加链接和各种各样的头到协商响应中。链接头链接。连接头会连接到根据文件扩展来的其他代表中。

7. 更多信息

- Nancy and Content Negotiation
- Revisting Content Negotiation and APIs part 1
- Revisting Content Negotiation and APIs part 2
- Revisting Content Negotiation and APIs part 3

二十四、使用转换器来扩展序列化

二十五、授权

Nancy中的验证使用扩展点：比如应用管道、模块管道、`NancyContext`和其他的一些扩展方法。所以你可以写自己的验证来替换默认提供的验证。

Nancy提供了以下几种验证，通过Nuget安装：

- 表单(`Nancy.Authentication.Forms`)
- 基本(`Nancy.Authentication.Basic`)
- 无状态(`Nancy.Authentication.Stateless`)

1. 了解用户

Nancy中用户使用`IUserIdentity`接口代表，它提供了一些用户的基本信息：

```
public interface IUserIdentity
{
    /// <summary>
    /// Gets or sets the name of the current user.
    /// </summary>
    string UserName { get; set; }

    /// <summary>
    /// Gets or set the claims of the current user.
    /// </summary>
    IEnumerable<string> Claims { get; set; }
}
```

你应当提供基于自己应用需求的类来实现自己的用户接口。

要获得当前用户，只需要获取`NancyContext`的`CurrentUser`属性。返回`null`值表明当前请求未认证，其他的则表示已认证。

context在Nancy的大部分地方都能获取，所以不必担心能否获取当前请求的用户身份。

2. 保护你的资源

可以在模块级和应用级来保护资源，方法是检测`NancyContext.CurrentUser`属性不为null。

这个任务可以通过在模块管道的`Before`中实现。这个钩子允许我们终结当前请求的执行，返回其它资源，比如当未验证用户视图访问安全资源时：

```
public class SecureModule : NancyModule
{
    public SecureModule()
    {
        Before += ctx => {
            return (this.Context.CurrentUser == null) ? new HtmlResponse(HttpStatus
Code.Unauthorized) : null;
        };

        // Your routes here
    }
}
```

在每个模块上添加安全代码违反了DRY原则，更是一个无聊的任务。使用扩展方法！

Nancy有一些扩展方法包装了这些任务，彻底的减少了要写的代码量。

下面是一些可用的扩展方法：

- `RequiresAuthentication` - 确保验证用户是可用的，或者返回`HttpStatusCode.Unauthorized`. 对于认证的用户，`CurrentUser`不能为`null`，而且`UserName`不能为空；
- `RequiresClaims` - 用户必须满足声明列表中所有的条件才能获取资源；
- `RequiresAnyClaim` - 见上一条，但是只需满足任意一条；
- `RequiresValidatedClaims` - 通过自定义函数，来全部自我掌控验证流程，函数格式`Func<IEnumerable<string>, bool>`；
- `RequiresHttps` - 只允许https访问；

这些都是`NancyModule`类的扩展方法，要使用它们需要添加`Nancy.Security`命名空间。

使用扩展方法，前面的例子可以这样写：

```
public class SecureModule : NancyModule
{
    public SecureModule()
    {
        this.RequiresAuthentication();
    }

    // Your routes here
}
```

当然还可以这样写：

```
public class SecureModule : NancyModule
{
    public SecureModule()
    {
        this.RequiresHttps();
        this.RequiresAuthentication();
        this.RequiresClaims(new [] { "Admin" });
    }

    // Your routes here
}
```

用户必须通过https，被授权，而且拥有Admin claim才能访问上面的路由。

3. 创造你自己的安全扩展

为了创造自己的安全扩展，你只需要添加扩展方法到`NancyModule`，并且绑定到`Before`管道，并检查证书。

比如，下面说明了`RequiresAuthentication`如何工作的：

```
```

```
public static class ModuleSecurity
{
    public static void RequiresAuthentication(this NancyModule module)
    {
        module.Before.AddItemToEndOfPipeline(RequiresAuthentication);
    }

    private static Response RequiresAuthentication(NancyContext context)
    {
        Response response = null;
        if ((context.CurrentUser == null) ||
            String.IsNullOrWhiteSpace(context.CurrentUser.UserName))
        {
            response = new Response { StatusCode = HttpStatusCode.Unauthorized };
        }

        return response;
    }

}
```

4. 实现自己的验证provider

实际的验证provider实现根据不同的需求变化很大，但是基本模式如下：

1. 应用管道的`Before`钩子用来检查请求的证书（比如cookie, headers等等）。如果发现证书，则验证用户并授权给`NancyContext`的`CurrentUser`属性。
2. 模块管道的`Before`钩子用来确认当前的请求是被认证的用户执行，如果不是，则拒绝并返回`HttpStatusCode.Unauthorized`
3. 应用管道的`After`钩子用来检查请求是否因为认证失败而被丢弃，比如检查`HttpStatusCode.Unauthorized`(401)状态码。如果检测到了就帮助用户去认证，比如重定向到login表单或者使用header的帮助通知客户端。

5. 无状态认证

无状态认证就是在每个请求中进行检查，根据请求的一些信息，来决定是否应该被确认为一个已认证的请求。

比如你检查请求来确认查询字符串的参数是否传递了api key，或者是否包含某些head， 有或者请求是否来自某些特定的ip。

使用无状态认证需要做下面几件事：

1. 安装`Nancy.Authentication.Stateless`包
2. 配置并开启无状态认证
3. 保护资源

5.1 配置并开启无状态认证

在bootstrapper中添加：

```
StatelessAuthentication.Enable(pipelines, statelessAuthConfiguration);
```

被传递到`StatelessAuthentication.Enable`方法中的`statelessAuthConfiguration`变量，是一个`StatelessAuthenticationConfiguration`类型的实例，它能够让你自定义无状态认证提供者的行为。

定义`StatelessAuthenticationConfiguration`类型实例的时候，需要有一个`Func<NancyContext, IUserIdentity>`类型的参数。这个函数用来检查请求或者context中的其他相关内容，并且在请求未通过验证时返回`null`，否则返回合适的`IUserIdentity`.

5.2 简单配置

```
var configuration =
    new StatelessAuthenticationConfiguration(ctx =>
    {
        if (!ctx.Request.Query.apikey.HasValue)
        {
            return null;
        }

        // This would where you authenticated the request. IUserApiMapper is
        // not a Nancy type.
        var userValidator =
            container.Resolve<IUserApiMapper>();

        return userValidator.GetUserFromAccessToken(ctx.Request.Query.apikey);
    });
```

6. Form认证

详细例子见Nancy解决方案中`Nancy.Demo.Authentication.Forms`例子

为了开启form认证，需要完成：

1. 安装`Nancy.Authentication.Forms`包
2. 实现`IUserMapper`
3. 实现路由来处理login和logout
4. 配置并开启Form认证

6.1 User mapper

User mapper用来负责从标示符identifier映射到用户。标示符是一个令牌，被存储在认证cookie中，用来代表执行请求的用户身份，避免每次请求时输入证书。

使用GUID来做标示符，如果用username来做标示符容易被嗅探并攻击。GUID还很难读取，而且每个GUID都不一样，增加了嗅探的难度。

注意，需要知道标示符对每个用户来说都是永久的并且是唯一的。

`IUserMapper`接口的定义：

```
public interface IUserMapper
{
    /// <summary>
    /// Get the real username from an identifier
    /// </summary>
    /// <param name="identifier">User identifier</param>
    /// <param name="context">The current NancyFx context</param>
    /// <returns>Matching populated IUserIdentity object, or empty</returns>
    IUserIdentity GetUserFromIdentifier(Guid identifier, NancyContext context);
}
```

6.2 修改应用，处理form认证

有了`IUserMapper`后，下一步就是在不需要认证的地方添加login和logout了。

下面是一个模块的基础框架。请注意资源的路径和模块的名称可以使任意的：

```
public class LoginModule : NancyModule
{
    public LoginModule()
    {
        Get["/login"] = parameters => {
            // Called when the user visits the login page or is redirected here bec
ause
            // an attempt was made to access a restricted resource. It should retur
n
            // the view that contains the login form
        };

        Get["/logout"] = parameters => {
            // Called when the user clicks the sign out button in the application.
Should
            // perform one of the Logout actions (see below)
        };

        Post["/login"] = parameters => {
```

```
            // Called when the user submits the contents of the login form. Should
            // validate the user based on the posted form data, and perform one of
the
            // Login actions (see below)
        };
    }
}
```

`Nancy.Authentication.Forms` 命名空间中有一些扩展方法可供使用：

- `LoginAndRedirect` - 登录用户并重定向用户到他们来时的url。或者也可以提供一个预留的url，用来在没有重定向url时使用。如果使用form提交，注意使用action=""，因为它会保留returnUrl原封不动。
- `LoginWithoutRedirect` - 登录用户，并且返回响应和状态码200(ok)
- `Login` 会调用当前请求的 `IsAjaxRequest` 的扩展方法，并且如果不是Ajax调用，则执行 `LoginAndRedirect` 方法，否则执行 `LoginWithoutRedirect` 方法
- `LogoutAndRedirect` - 登出用户，并提供重定向
- `LogoutWithoutRedirect` - 登出用户并返回状态码为200(OK)的响应
- `Logout` 会调用当前请求的 `IsAjaxRequest` 方法，如果不是ajax请求，则执行 `LogoutAndRedirect` ，否则执行 `LogoutWithoutRedirect`

注意1： `Nancy.Extensions.RequestExtensions` 中的 `IsAjaxRequest` 扩展方法会检查 `X-Requested-With` 头，并且在其包含值 `XMLHttpRequest` 时返回true

注意2： 请确认路径的定义login和logout的页面没有要求使用登录。

6.3 启用form认证

在bootstrapper中添加：

```
FormsAuthentication.Enable(pipelines, formsAuthConfiguration);
```

既可以在 `ApplicationStartup` 中又可以在 `RequestStartup` 中添加。到底在何处加，取决于 `IUserMapper` ，即user mapper到底是有应用级的生命周期还是请求级的生命周期。

传递给 `FormsAuthentication.Enable` 方法的 `formsAuthConfiguration` 变量是 `FormsAuthenticationConfiguration` 类型，它能让你自定义form认证提供者的行为。

比如，下面是一个基本的认证配置：

```
var formsAuthConfiguration =
new FormsAuthenticationConfiguration()
{
    RedirectUrl = "~/login",
    UserMapper = container.Resolve<IUserMapper>(),
};
```

下面是一些配置项：

- `RedirectingQuerystringKey`：默认名是`returnUrl`
- `RedirectingUrl`：未认证的用户应当被重定向的url，一般是登录页面`~/login`
- `UserMapper`：`IUserMapper`在认证时应该被使用
- `RequiresSSL`：SSL
- `DisableRedirect`：遇到未认证时，是否重定向到登陆页
- `CryptographyConfiguration`：`CryptographyConfiguration.Default`与form认证cookie配合使用。`CryptographyConfiguration.Default`是默认的。

6.4 关于加密，还有一些话

默认使用`RandomKeyGenerator`，这意味着每次程序启动时会产生一个新的秘钥，那么应用重启回到这认证cookie失效，在多台机器负载均衡时也会出现这种问题，别怕，看看加密配置

下面是一个例子：

```
var cryptographyConfiguration = new CryptographyConfiguration(
    new RijndaelEncryptionProvider(new PassphraseKeyGenerator("SuperSecretPass", ne
w byte[] { 1, 2, 3, 4, 5, 6, 7, 8 })),
    new DefaultHmacProvider(new PassphraseKeyGenerator("UberSuperSecure", new byte[
] { 1, 2, 3, 4, 5, 6, 7, 8 })));

var config =
    new FormsAuthenticationConfiguration()
    {
        CryptographyConfiguration = cryptographyConfiguration,
        RedirectUrl = "/login",
        UserMapper = container.Resolve<IUserMapper>(),
    };
```

6.5 跟多

- Forms authentication with nancyfx
- Multiple forms authentication sections

7. 令牌认证

详细例子在Nancy解决方案中的`Nancy.Demo.Authentication.Token`中。

7.1 认识Nancy的令牌认证

Nancy令牌认证工程是为了多种客户端(iOS, Android, Angular SPA等等)能与统一后台Nancy应用而创建的。

7.2 基本原理

令牌认证与授权在下面这些需求下应运而生：

- 没有cookie（不适所有的客户端都是浏览器）
- 避免一旦用户被认证/授权后，从后端数据存储中取回用户和权限信息
- 允许客户端应用在第一次授权后保存令牌，以便为后续请求使用
- 通过单向加密算法确保令牌没有被篡改，阻止嗅探冒充令牌攻击
- 使用有期限的可配置的key来进行令牌生成
- 使用server端的文件系统来存储私钥，这样即使应用重启也能恢复。注意：可以使用内存存储作为测试。

7.3 使用

7.3.1 Nancy配置

令牌认证可以像form认证那样：

```
public class Bootstrapper : DefaultNancyBootstrapper
{
    protected override void RequestStartup(TinyIoCContainer container, IPipelines p
ipelines, NancyContext context)
    {
        TokenAuthentication.Enable(pipelines, new TokenAuthenticationConfiguration(
container.Resolve<ITokenizer>()));
    }
}
```

令牌从`IUserIdentity`和`NancyContext`中，通过实现`ITokenizer`接口产生。默认实现是`Tokenizer`，它提供了一些可配置的方法。默认情况下，它产生一个令牌包含下面部分：

- 用户名
- Pipe separated list of user claims
- UTC当前时间
- 客户端的"User-Agent"头（必须）

建议配置Tokenizer，使用其他附加能代表用户唯一设备的信息。

下面举例说明了如何初始化用户认证，并且返回生成的令牌给客户端：

```
public class AuthModule : NancyModule
{
    public AuthModule(ITokenizer tokenizer)
        : base("/auth")
    {
        Post["/"] = x =>
            {
                var userName = (string)this.Request.Form.UserName;
```

```
                var password = (string)this.Request.Form.Password;

                var userIdentity = UserDatabase.ValidateUser(userName, password);

                if (userIdentity == null)
                {
                    return HttpStatusCode.Unauthorized;
                }

                var token = tokenizer.Tokenize(userIdentity, Context);

                return new
                    {
                        Token = token,
                    };
            };

        Get["/validation"] = _ =>
            {
                this.RequiresAuthentication();
                return "Yay! You are authenticated!";
            };

        Get["/admin"] = _ =>
        {
            this.RequiresClaims(new[] { "admin" });
            return "Yay! You are authorized!";
        };
    }
}
```

7.3.2 客户端配置

一旦你的客户端接收到了token，那么你必须使用token来设置HTTP头：

```
Authorization: Token {your-token-goes-here}
```

8. 幕后的工作

https://github.com/NancyFx/Nancy/commit/9ae0a5494bc335c3d940d730ae5d5f18c1018836