**[ASP.NET Core 之 Identity 入门（一）](https://www.cnblogs.com/savorboard/p/aspnetcore-identity.html)**

**前言**

在 ASP.NET Core 中，仍然沿用了 ASP.NET里面的 Identity 组件库，负责对用户的身份进行认证，总体来说的话，没有MVC 5 里面那么复杂，因为在MVC 5里面引入了OWIN的东西，所以很多初学者在学习来很费劲，对于 Identity 都是一头雾水，包括我也是，曾经在学 identity 这个东西前后花了一个多月来搞懂里面的原理。所以大部分开发者对于 Identity 并没有爱，也并没有使用它，会觉得被绑架。

值得庆幸的是，在 ASP.NET Core 中，由于对模块的抽象化逐渐清晰，以及中间件的使用，这使得 Identity 的学习和使用路线变得更加平易近人，下面就让我们一起来看看吧。

**Getting Started**

在开始之前，让我们先忘记它和Entity Framework的关系，也忘记它和Authentication的关系，我们先学习几个英语单词。

有这么几个“**单词**”你可能需要弄明白：

**# 1： Claims**

大家应该都知道身份证长什么样子的，如下：



其中，**姓名**：奥巴马；**性别**：男；**民族**：肯尼亚；**出生**：1961.08.04，等等这些身份信息，可以看出都是一个一个的**键值对**，那如果我们想在程序中存这些东西，怎么样来设计呢？对，你可能想到了使用一个字典进行存储，一个Key，一个Value刚好满足需求。但是Key，Value的话感觉不太友好，不太面向对象，所以如果我们做成一个对象的话，是不是更好一些呢？最起码你可以用vs的智能提示了吧，我们修改一下，改成下面这样：

//我给对象取一个名字叫`Claim`你没有意见吧

public class Claim

{

public string ClaimType { get; set; }

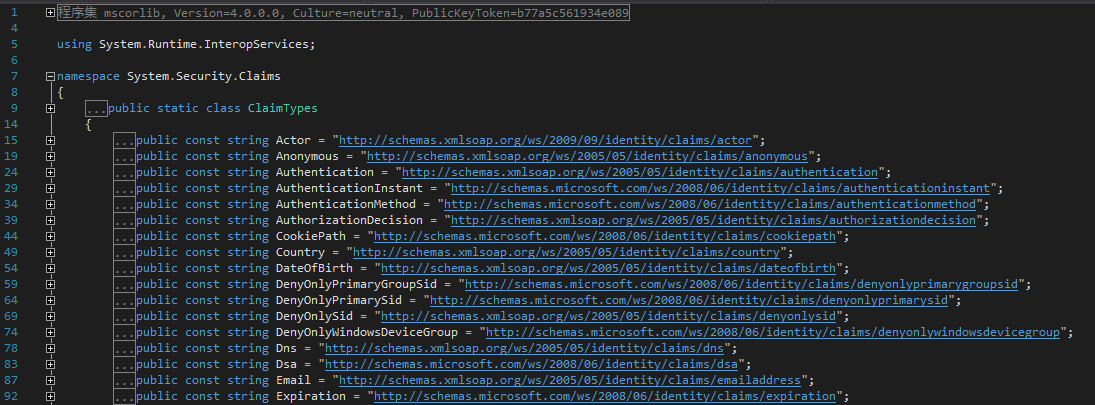
public string ClaimValue { get; set; }

}

ClaimType 就是Key，ClaimValue就代表一个Value。这样的话，刚好可以存储**一个**键值对。这时候姓名：奥巴马是不是可以存进去了。

微软的人很贴心，给我们准备了一些默认的ClaimType呢？很多常用的都在里面呢，一起看看吧：

这里延伸第一个知识点：ClaimTypes



为了阅读体验，截图我只放了一部分哦。可以看到有什么Name，Email，Gender，MobilePhone等常用的都已经有了，其他的还有很多。细心的读者可能注意了，它的命名空间是System.Security.Claims，那就说明这个东西是.net 框架的一部分，嗯，我们暂时只需要知道这么多就OK了。

Claim 介绍完毕，是不是很简单，其他地方怎么翻译我不管，在本篇文章里面，它叫 “**证件单元**”。

**# 2： ClaimsIdentity**

在有了“**证件单元**”之后，我们就用它可以制造一张身份证了，那么应该怎么样制造呢？有些同学可能已经想到了，对，就是新建一个对象，然后在构造函数里面把身份证单元传输进去，然后就得到一张身份证了。我们给这张身份证取一个英文名字叫 “**ClaimsIdentity**”，这个名字看起来还蛮符合的，既有 Claims 表示其组成部分，又有表示其用途的 Identity（身份），很满意的一个名字。

实际上，在现实生活中，我们的身份证有一部分信息是隐藏的，有一部分是可以直接看到的。比如新一代的身份证里面存储了你的指纹信息你是看不到的，这些都存储在身份证里面的芯片中，那能看到的比如姓名啊，年龄啊等。我们在设计一个对象的时候也是一样，需要暴露出来一些东西，那这里我们的 ClaimsIdentity 就暴露出来一个 Name，Lable等。

我们造的身份证（ClaimsIdentity）还有一个重要的属性就是类型（AuthenticationType），等等，AuthenticationType是什么东西？看起来有点眼熟的样子。我们知道我们自己的身份证是干嘛的吧，就是用来证明我们的身份的，在你证明身份出示它的时候，其实它有很多种形式载体的，什么意思呢？比如你可以直接拿出实体形式的身份证，那也可以是纸张形式的复印件，也可以是电子形式的电子码等等，这个时候就需要有一个能够表示其存在形式的类型字段，对，这个AuthenticationType就是干这个事情的。

然后我们在给我们的身份证添加一些润色，让其看起来好看，比如提供一些方法添加 Claims 的，删除 Claims的，写到二进制流里面的啊等等，最终我们的身份证对象看起来基本上是这样了：

public class ClaimsIdentity

{

public ClaimsIdentity(IEnumerable<Claim> claims){}

//名字这么重要，当然不能让别人随便改啊，所以我不许 set，除了我儿子跟我姓，所以是 virtual 的

public virtual string Name { get; }

public string Label { get; set; }

//这是我的证件类型，也很重要，同样不许 set

public virtual string AuthenticationType { get; }

public virtual void AddClaim(Claim claim);

public virtual void RemoveClaim(Claim claim);

public virtual void FindClaim(Claim claim);

}

嗯，到这里，我们的**身份证**看起来似乎很完美了，但是从面向对象的角度来说好像还少了点什么东西？ 对~，还是抽象，我们需要抽象出来一个接口来进行一些约束，约束什么呢？既然作为一个证件，那么肯定会涉及到这几个属性信息：  
1、名字。2、类型。3、证件是否合法。  
反应到接口里面的话就是如下，我们给接口取个名字叫：“**身份(IIdentity)**”：

这里延伸第二个知识点：IIdentity接口。

// 定义证件对象的基本功能。

public interface IIdentity

{

//证件名称

string Name { get; }

// 用于标识证件的载体类型。

string AuthenticationType { get; }

//是否是合法的证件。

bool IsAuthenticated { get; }

}

所以我们的 ClaimsIdentity 最终看起来定义就是这样的了：

public class ClaimsIdentity : IIdentity

{

//......

}

ClaimsIdentity 介绍完毕，是不是发现也很简单，其他地方怎么翻译我不管，在本篇文章里面，它叫 “**身份证**”。

**# 3： ClaimsPrincipal**

有了身份证，我们就能证明我就是我了，有些时候一个人有很多张身份证，你猜这个人是干嘛的？ 对，不是黄牛就是诈骗犯。

但是，有些时候一个人还有其他很多种身份，你猜这个人是干嘛的？这就很正常了对不对，比如你可以同时是一名教师，母亲，商人。如果你想证明你同时有这几种身份的时候，你可能需要出示教师证，你孩子的出生证，法人代表的营业执照证。

在程序中，一个身份证不仅仅代表你这个人了，而是代表一个身份，是证明你自己的主要身份哦。如果一个人还有其他很多种身份，这个时候就需要有一个东西（载体）来携带着这些证件了对吧？OK，我们给需要携带证件的这个对象取一个贴切点的名字，叫“**证件当事人（ClaimsPrincipal）**”吧。

以下是 Principal 这个单词在词典给出的解释，我用它你应该没意见吧：

principal ['prɪnsəpl]

adj. 主要的；资本的

n. 首长；校长；资本；当事人

这个时候可能有同学会问了，是不是应该叫ClaimsIdentityPrincipal比较好呢？嗯，我也觉得应该叫 ClaimsIdentityPrincipal 可能更好一点，或许微软的人偷懒了，简写成了ClaimsPrincipal。

知道其功能后，代码就很好写了，和上面ClaimsIdentity一样的套路：

public class ClaimsPrincipal

{

//把拥有的证件都给当事人

public ClaimsPrincipal(IEnumerable<ClaimsIdentity> identities){}

//当事人的主身份呢

public virtual IIdentity Identity { get; }

public virtual IEnumerable<ClaimsIdentity> Identities { get; }

public virtual void AddIdentity(ClaimsIdentity identity);

//为什么没有RemoveIdentity ， 留给大家思考吧？

}

当时人看起来也几乎完美了，但是我们还需要对其抽象一下，抽象哪些东西呢？ 作为一个当事人，你应该有一个主身份吧，就是你的身份证咯，可能你还会用到角色（角色后面会详细介绍，这里你知道有这么个东西就行了）。

这里延伸第三个知识点：IPrincipal 接口。

public interface IPrincipal

{

//身份

IIdentity Identity { get; }

//在否属于某个角色

bool IsInRole(string role);

}

然后，我们的 **证件当事人** 看起来应该是这样的：

public class ClaimsPrincipal : IPrincipal

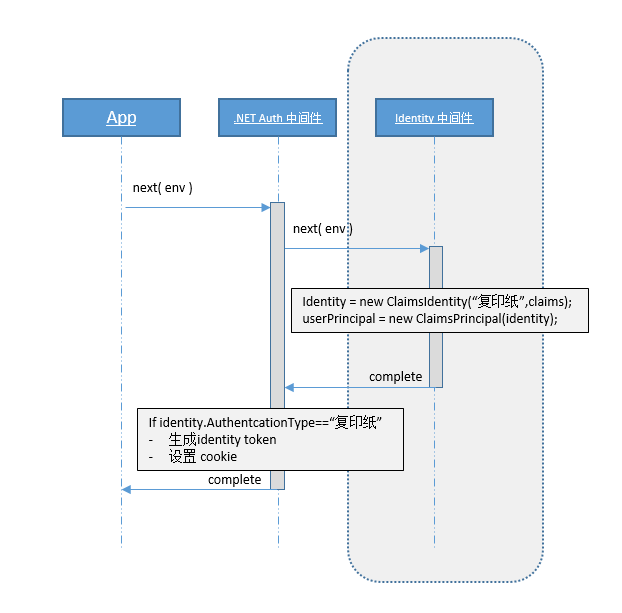
{

//...

}

ClaimsPrincipal 介绍完了，也很简单吧？ 其他地方怎么翻译我不管，在本篇文章里面，它叫 “**证件当事人**”。

想在，我们已经知道了 “证件单元（Claims）” ， “身份证（ClaimsIdentity）” ， “证件当事人（ClaimsPrincipal）”，并且整理清楚了他们之间的逻辑关系，趁热打铁，下面这个图是一个identity登入部分的不完全示意图，虚线圈出来的部分应该可以看懂了吧：



可以看出，首先我们在app这边有一些**证件单元**，然后调用ClaimsIdentity把证件单元初始化为一个**身份证**，然后再把身份证交给**证件当事人**由其保管。

才把 Getting Started 写完，发现已经这么长了，所以打算写成一个系列了，可能3 - 4篇吧。

**总结**

好了，本篇就先介绍到这里，在本篇博客中，我们学会了几个英文单词，并且知道了这些英文单词在程序中是扮演这怎么样一个对象。并且根据图我们知道了这些对象在整个认证系统种处在怎么样一个位置。 我发现如果想把 identity 讲清楚仅仅靠这一篇博客是不够的，下一篇我们将对.NET Authentication中间件进行抽丝剥茧，直到掌握.NET的整个认证系统后，我们再来看一下 Identiy 到底和 Entity Framework 有着怎样的爱恨情仇。

这仅仅是一个开始，大家如果觉得本篇博客对您有帮助的话，感谢您的【推荐】，如果你对 .NET Core 感兴趣可以关注我，我会定期在博客分享关于 .NET Core 的学习心得。

本文地址：[http://www.cnblogs.com/savorboard/p/aspnetcore-identity.html](http://www.cnblogs.com/savorboard/p/aspnetcore-identity.html" \t "_blank)  
作者博客：[Savorboard](http://www.cnblogs.com/savorboard" \t "_blank)  
欢迎转载，请在明显位置给出出处及链接

# [ASP.NET Core 之 Identity 入门（二）](https://www.cnblogs.com/savorboard/p/aspnetcore-identity2.html)

### 前言

在 [上篇文章](http://www.cnblogs.com/savorboard/p/aspnetcore-identity.html) 中讲了关于 Identity 需要了解的单词以及相对应的几个知识点，并且知道了Identity处在整个登入流程中的位置，本篇主要是在 .NET 整个认证系统中比较重要的一个环节，就是 认证（Authentication），因为想要把 Identity 讲清楚，是绕不过 Authentication 的。

在之前写过一篇关于 ASP.NET Core 中间件的文章，里面有一部分（怎么样自定义自己的中间件）是具体关于认证系统的一个具体使用，有兴趣的朋友可以看一下[这篇文章](http://www.cnblogs.com/savorboard/p/5586229.html)。

其实 Identity 也是认证系统的一个具体使用，大家一定要把 Authentication 和 Identity 当作是两个东西，一旦混淆，你就容易陷入进去。

下面就来说一下 ASP.NET Core 中的认证系统是怎么样一回事。不要怕，其实很简单，全是干货~

### Getting Started

大家应该还记得在上一篇中的奥巴马先生吧，他现在不住在华盛顿了，他到中国来旅游了，现在住在北京，这几天听说西湖风景不错，于是在 12306 定了一张北京到杭州的高铁票。取到票之后，他向我们展示了一下：



今天是11.11号，奥巴马很开心，原因你懂的。快到出发的时间了，于是，拿着票走到了火车站检票口，刚把身份证和火车票递给检票员。“cut”，导演喊了一声。尼玛原来是在拍电影呢~  
导演说：奥巴马，你演的太烂了，别演了，你来演检票员吧，让旁边小李来演要出行路由的奥巴马吧。奥巴马不情愿的说了一声：“好吧，希望小李能够受的了你”。

“action”，导演又喊了一声，故事开始了~

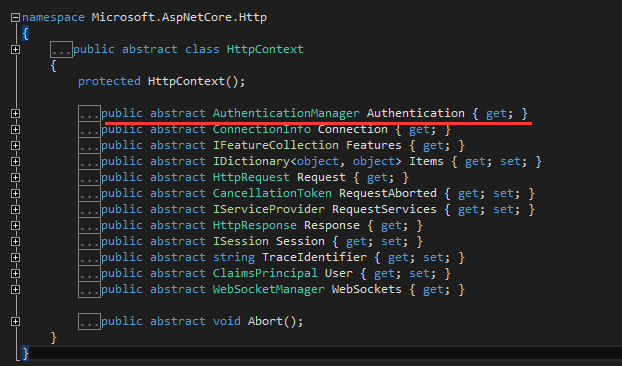
### AuthenticationManager

奥巴马当了检票员以后，特别高兴，因为他有权利了呀，他可以控制别人能不能上车了，说不定还能偷偷放几个人进去捞点外快呢。

得知了他能干什么以后，他觉得检票员这个名字简直太 low 了，很快，他就有了一个新的高大上的名字，叫：认证管理员（AuthenticationManager），而且，他觉得他自己应该处在整个核心位置，为什么呢？你想想看，那么庞大的一套铁路载人系统，能不能有收入有钱赚，全靠他给不给放人进去，如果一个人都不放进去，另外那一大帮人只能去喝西北风了。

到这里，聪明的同学可能已经知道奥巴马把他自己放在怎么样一个核心位置了。对，他把自己放到了 HttpContext 里面。怎么样？ 够核心吧。

这里延伸第一个知识点：AuthenticationManager 所处的位置



有同学在上面的截图里面发现了 public abstract ClaimsPrincipal User { get; set; }， 这不就是我们上一篇中讲到的 **“ 证件当事人 ”** ，现在小李扮演的那个角色么？ 对，这个 User 就是本文中的小李，被你提前发现他躲着这里了，嘿嘿。

还有一个知识点，就是 AuthenticationScheme，什么意思呢？ 且看  
奥巴马敢把自己放在这么核心的位置也是有他的能力的，怎么讲呢？ 比如说在检票的时候，别人递过来一张身份证和一张火车票，那怎么样验证这两个证件是合法的呢？ 以下就是奥巴马提出的针对两种证件的验证方案：

方案1、针对身份证的验证，可以查看其本人是否和身份证头像是否一致，年龄是否符合当事人具体年龄。  
方案2、针对火车票的验证，可以看车次，时间是否符合发车目标，另外可以看车票上的身份号码是否和身份证一致。

其中，这每一种方案，就对应一个 AuthenticationScheme（验证方案名称），是不是明白了。

这就是第二个知识点 AuthenticationScheme 很重要。

知道了奥巴马的职责后，就很容易的把代码写出来了：

public abstract class AuthenticationManager

{

//AuthenticateContext包含了需要认证的上下文，里面就有小李

public abstract Task AuthenticateAsync(AuthenticateContext context);

//握手

public abstract Task ChallengeAsync(string authenticationScheme, AuthenticationProperties properties, ChallengeBehavior behavior);

//登入

public abstract Task SignInAsync(string authenticationScheme, ClaimsPrincipal principal, AuthenticationProperties properties);

//登出

public abstract Task SignOutAsync(string authenticationScheme, AuthenticationProperties properties);

}

奥巴马做为一个检票员，有一个认证方法，AuthenticateAsync() ，注意这是其一个核心功能，其他几个都可以没有，但是唯独不能没有这个功能，没有的话他就不能称之为一个检票员了。

然后还有一个握手ChallengeAsync，登入SignInAsync和登出SignOutAsync，下面说说笔者对这三个方法的理解吧。

**ChallengeAsync**：是社区协议文件 [RFC2167](https://tools.ietf.org/html/rfc2617) 定义的关于在HTTP Authentication 过程中的一种关于握手的一个过程，主要是摘要认证（digest authentication），更多信息查看[这里](http://blog.csdn.net/gl1987807/article/details/6090811)。

是不是有点专业，看不懂，没事，有通俗版本的。 小李要进站了，这个时候小李问了一下我们的检票员奥巴马先生。

* **小李**：“你好，检票员，我可以进站吗？”
* **检票员奥巴马**：“要赶火车吗？可以啊，请出示你的证件？”
* **小李**：“好的，这是我的证件，你检查一下？”
* **检票员奥巴马**：“嗯，证件没问题，进去吧”

这样一个过程就是握手（digest-challenge）或者叫问答的一个过程，明白了 ChallengeAsync 的原理了吧? 是不是很简单。

**SignInAsync**，**SignOutAsync**：个人觉得这两个不应该放在这里，因为并不属于认证的职责，也不属于协议规定的内容。但是这两个方法确实需要抽象，应该单独抽取一个接口存放，至于为什么这样做，或许是因为以下原因：

1、对登入登出的抽象是和认证紧密结合的，大多数情况下认证资料的保存是需要在SignIn进行的，比如 Cookies Authentication 中间件就在SignIn方法里面做了Cookie的保存。

2、 AuthenticationManager 这个对象是处在 HttpContext  
上下文里面的，本着面向抽象和封装的原则，放到其里面是合适的，这样能够很方便的用户对其调用。

关于 AuthenticationManager 已经介绍完了，是不是很简单呢？

### IAuthenticationHandler

有些同学可能会问了，如果 AuthenticationManager 不提供接口的话，只是一个抽象类的话，那如果自定义认证方法就必须继承它，这对于开发者来说是不友好的，也违背了面向接口编程的理念。嗯，确实是这样，那么接口来了：

public interface IAuthenticationHandler

{

void GetDescriptions(DescribeSchemesContext context);

Task AuthenticateAsync(AuthenticateContext context);

Task ChallengeAsync(ChallengeContext context);

Task SignInAsync(SignInContext context);

Task SignOutAsync(SignOutContext context);

}

这个接口是在 AuthenticationManager 实现类 DefaultAuthenticationManager 中延伸出来的，所以大家不用再去看里面的源码了，记住以后如果需要重写认证相关的东西，实现IAuthenticationHandler就可以了。

### Authentication 中间件

对 IAuthenticationHandler 的初步实现，封装了 AuthenticationHandler 这个抽象类，把具体的核心功能都交给下游去实现了，下面的CookieAuthentication 中间件核心类 CookieAuthenticationHandler 就是继承自AuthenticationHandler， 知道这么多就够了。

### CookieAuthentication 中间件

故事还要继续，奥巴马在接到小李递来的身份证和火车票之后，首先拿着火车票在一个二维码机器上扫描了一下，然后又拿着身份证在一个机器上刷了一下，经过核查，发现都没有问题。于是拿起印章在上面盖了一个 “ 验讫 ”。

这中间都发生了什么呢？

首先，在二维码扫描的过程，这个过程二维码机器会解析你火车票上的二维码，如果发现解析失败，会直接响应认证失败。也就是你别想进站了。

如果解析成功，就会得到你这个票据中的信息了，然后拿到你票据里面的的当事人信息进行验证是否被列为了铁路局黑名单中。

如果验证通过，则会给你颁发一个识别码，把符合你身份的一个识别码写入到你的火车票中和检票员旁边的电脑系统中，即 “ 验讫 ”。

话说这个验讫有点高级，它会向你的火车票芯片中写入一些信息，那么都写入些什么信息呢？ 1、奥巴马个人的信息。2、验证途中的一些上下信息。3、使用的验证方案。

知道了，这些之后，那么就很容易实现这个验证方法了，对吧？ 以下是 CookieAuthentication 中间件中的核心类 CookieAuthenticationHandler 的里面的核心方法HandleAuthenticateAsync()，同样你可以理解为实现的 IAuthenticationHandler 接口的 AuthenticateAsync：

protected override async Task<AuthenticateResult> HandleAuthenticateAsync()

{

// 解析二维码

var result = await EnsureCookieTicket();

if (!result.Succeeded)

{

return result;

}

// 从二维码中拿当事人信息进行验证

var context = new CookieValidatePrincipalContext(Context, result.Ticket, Options);

await Options.Events.ValidatePrincipal(context);

if (context.Principal == null)

{

return AuthenticateResult.Fail("No principal.");

}

if (context.ShouldRenew)

{

RequestRefresh(result.Ticket);

}

// 验讫， 写入芯片

return AuthenticateResult.Success(new AuthenticationTicket(context.Principal, context.Properties, Options.AuthenticationScheme));

}

#### **HandleSignInAsync**

我们故事继续……

奥巴马检票完成之后，把票就交给了小李，小李拿到票之后，导演又喊了一声：“ cut ”……

怎么又停了，小李和奥巴马一肚子的疑惑，导演说：“ 奥巴马呀，你检票员演的不错，还是继续扮演你的本职角色吧，演好了中午盒饭给你双份，小李，你来演检票员吧 ”。  
可以吃两份盒饭了，奥巴马听后心里还是很开心。

“action” 导演喊了一声……

奥巴马接过票，向着车站里面的列车停车处走去，走到了列车门口要进去的时候，又出现了一个人，奥巴马知道，这个人就是做车内乘客登记的（ps: 一般情况下，做乘客登记都是在列车行驶的过程中，在这里我们假设这个做乘客登记的人比较勤快，就在车门口守着），登记完成之后就让奥巴马进去了。

那么，登记这个过程中都干了些什么呢？

首先，登记员的手持设备会解析火车票票里面写入芯片中的信息，发现没有问题，就开始向自己手里面的登记本登记信息了，主要包含车票主人信息，过期时间，审核人等。

这样整个过程就是 HandleSignInAsync 的一个过程，换成程序术语就是，组装 Cookie 登入上下文信息，写入到 Http 流的 header 中，也就写入到了客户端浏览器cookie。

至此，整个过程就完了，我们来看一下代码：

//方法里面的流程，我只列出了核心部分，影响阅读的全删了

protected override async Task HandleSignInAsync(SignInContext signin)

{

// 解析芯片中的信息

var result = await EnsureCookieTicket();

// 组织登入上下文，设置过期时间等

// 使用 data protected 加密登记本上的信息

var cookieValue = Options.TicketDataFormat.Protect(ticket);

// 写入到浏览器header

await ApplyHeaders(cookieValue);

}

不想深入了解的可以忽略这部分内容：  
在 HandleSignInAsync 这个函数的源码中，其中有一个很巧妙的设计, 就是 await Options.Events.SignedIn(signedInContext); 这样一句代码，干什么用的呢？ 而且前后一共调用了两次，有同学知道是为什么吗？ 我准备在下一篇中给出答案。

还记得前面 HttpContext 中的ClaimsPrincipal User吗？ 就是小李临时顶替的那个角色，现在有值了，他就是是奥巴马了。

奥巴马在座位上坐好之后，经过6个小时的路程就从北京到杭州了，不得不佩服中国高铁的速度呀

至此，CookieAuthentication 中间件的整个工作流程已经讲完了，故事也结束了。

以上，就是这两行代码背后的故事：

var user = new ClaimsPrincipal(new ClaimsIdentity(new[] { new Claim(ClaimTypes.Name, "奥巴马") }, CookieAuthenticationDefaults.AuthenticationScheme));

await HttpContext.Authentication.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, user);

### 总结

在本篇中我们知道了 AuthenticationManager，也知道了 IAuthenticationHandler 并且简单的介绍了一下 Authentication 中间件和 CookieAuthentication 中间件，其中 CookieAuthentication 中间件是我们以后使用最多的一个中间件了，本篇也对其做了一个详细的介绍，我想通过本篇文章在以后使用的过程中应该问题不大了。

有同学可能会问了，讲了这么多认证的东西它和 Identity 有什么关系呢？ 难道我通篇都在隐藏他和 Identity 的关系你没看出来？。。。。真的想知道？ 看下一篇吧。

顺便给自己打个广告：成都地区有推荐工作的么？有意联系右上角。

如果觉得本篇博客对您有帮助的话，感谢您的【推荐】，如果你对 .NET Core 感兴趣可以关注我，我会定期在博客分享关于 .NET Core 的学习心得。

本文地址：<http://www.cnblogs.com/savorboard/p/aspnetcore-identity2.html>  
作者博客：[Savorboard](http://www.cnblogs.com/savorboard)  
欢迎转载，请在明显位置给出出处及链接

分类: [ASP.NET Core](https://www.cnblogs.com/savorboard/category/840273.html)

标签: [asp.net core](https://www.cnblogs.com/savorboard/tag/asp.net%20core/), [asp.net core identity](https://www.cnblogs.com/savorboard/tag/asp.net%20core%20identity/), [cookie中间件](https://www.cnblogs.com/savorboard/tag/cookie%E4%B8%AD%E9%97%B4%E4%BB%B6/)

# [ASP.NET Core 之 Identity 入门（三）](https://www.cnblogs.com/savorboard/p/aspnetcore-identity3.html)

### 在[上一篇](http://www.cnblogs.com/savorboard/p/aspnetcore-identity2.html" \t "_blank)文章中，我们学习了 CookieAuthentication 中间件，本篇的话主要看一下 Identity 本身。

最早2005年 ASP.NET 2.0 的时候开始， Web 应用程序在处理身份验证和授权有了很多的变化，多了比如手机端，平板等，所以那个时候为了适应这种变化就引入了ASP.NET Membership，但是随着时间的发展一些社交网站或者程序聚集了大量的用户，比如Facebook，Twitter，QQ等，这个时候用户希望能够使用他们在这些社交站点身份来登陆当前网站，这样可以免除注册这些琐碎而又必要的操作，用户也不必记住大量的账户密码。

又随着互联网的发展，越来越多的开发者不只是关注具体业务代码的编写，转变为开始关注应用程序代码的单元测试，这已经是开发者关注的核心。所以在2008年，ASP.NET 团队引入了 MVC 框架，这样来帮助开发者很方便的构建单元测试，同时开发者希望他们的 Membership 系统也能够做到这一点。

基于以上，ASP.NET Identity 应运而生。

### Identity 要解决的问题

很多开发人员说他们不愿意使用Identity，自己实现要方便的多，OK，那么需求来了？以下就是我针对此次任务给你提出来的需求。

**身份系统**

* 可以同时被所有的ASP.NET 框架使用（Web MVC，Web Forms，Web Api，SignalR）
* 可以应用于构建 Web, 手机，存储，或者混合应用。

**能够对用户资料（User Profile）很方便的扩展**

* 可以针对用户资料进行扩展。

**持久化**

* 默认把用户信息存储在数据库中，可以支持使用EF进行持久化。（**可以看到，EF 其实只是Identity的一个功能点而已**）
* 可以控制数据库架构，更改表名或者主键的数据类型（int，string）
* 可以使用不同的存储机制（如 NoSQL，DB2等）

**单元测试**

* 使WEB 应用程序可以进行单元测试，可以针对ASP.NET Identity编写单元测试

**角色机制**

* 提供角色机制，可以使用不同的角色来进行不同权限的限制，可以轻松的创建角色，向用户添加角色等。

**要支持基于Claims**

* 需要支持基于 Claims 的身份验证机制，其中用户身份是一组Claims，一组Claims可以比角色拥有更强的表现力，而角色仅仅是一个bool值来表示是不是会员而已。

**第三方社交登陆**

* 可以很方便的使用第三方登入，比如 Microsoft 账户，Facebook， Twitter，Google等，并且存储用户特定的数据。

**封装为中间件**

* 基于中间件实现，不要对具体项目产生依赖
* 基于 Authorzation 中间件实现，而不是使用 FormsAuthentication 来存储cookie。

**NuGet包提供**

* 发布为 Nuget 包，这样可以容易的进行迭代和bug修复，可以灵活的提供给使用者。

以上，就是我提出来的需求，如果让你来封装这样一个用户身份认证组件，你会不是想到以上的这些功能点，那针对于这些功能点你又会怎么样来设计呢？

下面来看一下 Identity 怎么样设计的吧。

### Getting Started

抽丝剥茧，我们先从入口看一下其使用方式。 首先我们打开 Startup.cs 文件，然后添加如下代码：

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));

services.AddIdentity<ApplicationUser, IdentityRole>(options => {

options.Cookies.ApplicationCookie.AuthenticationScheme = "ApplicationCookie";

options.Cookies.ApplicationCookie.CookieName = "Interop";

})

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

}

public void Configure(IApplicationBuilder app)

{

// 使用了 CookieAuthentication 中间件做身份认证

app.UseIdentity();

}

}

在 ConfigureServices 中，先是注册了数据库上下文，然后又 services.AddIdentity() 我们看一下里面都注册了哪些服务呢？

public static IdentityBuilder AddIdentity<TUser, TRole>(

this IServiceCollection services,

Action<IdentityOptions> setupAction)

where TUser : class

where TRole : class

{

// 这个就是被 Identity 使用的

services.AddAuthentication(options =>

{

// This is the Default value for ExternalCookieAuthenticationScheme

options.SignInScheme = new IdentityCookieOptions().ExternalCookieAuthenticationScheme;

});

// 注册 IHttpContextAccessor ，会用到

services.TryAddSingleton<IHttpContextAccessor, HttpContextAccessor>();

// Identity services

services.TryAddSingleton<IdentityMarkerService>();

services.TryAddScoped<IUserValidator<TUser>, UserValidator<TUser>>();

services.TryAddScoped<IPasswordValidator<TUser>, PasswordValidator<TUser>>();

services.TryAddScoped<IPasswordHasher<TUser>, PasswordHasher<TUser>>();

services.TryAddScoped<ILookupNormalizer, UpperInvariantLookupNormalizer>();

services.TryAddScoped<IRoleValidator<TRole>, RoleValidator<TRole>>();

// 错误描述信息

services.TryAddScoped<IdentityErrorDescriber>();

services.TryAddScoped<ISecurityStampValidator, SecurityStampValidator<TUser>>();

//身份当事人工厂

services.TryAddScoped<IUserClaimsPrincipalFactory<TUser>, UserClaimsPrincipalFactory<TUser, TRole>>();

//三大对象

services.TryAddScoped<UserManager<TUser>, UserManager<TUser>>();

services.TryAddScoped<SignInManager<TUser>, SignInManager<TUser>>();

services.TryAddScoped<RoleManager<TRole>, RoleManager<TRole>>();

if (setupAction != null)

{

services.Configure(setupAction);

}

return new IdentityBuilder(typeof(TUser), typeof(TRole), services);

}

看了以上代码后，基本上知道了 Identity 他的设计的一个架构了，通过此结构我们也能够学习到我们自己封装一个中间件的时候，该怎么样来组织我们的代码结构，怎么样的利用 ASP.NET Core 给我们提供的依赖注入来更好的解耦，下面我们来看一下通过以上代码我们能够学到什么东西：

1、 在 public static IdentityBuilder AddIdentity<TUser, TRole>(this IServiceCollection services, Action<IdentityOptions> setupAction) 这个扩展方法中，提供了一个参数 Action<IdentityOptions>，这个是干什么用的呢？ 这个是我们在设计一个中间件的时候，有需要外部提供的参数，就会设计一个 Options 的类用来接受外部的参数，然后封装为一个Action委托的方式提供。在使用到的地方就可以以 IOption xxx 的形式注入进来使用了。

2、services.TryAddScoped<Interface,Implement>()，这个注册方式就是说，如果检测到DI容器里面已经有了当前要注册的Interface或者Service，就不会再次注册，没有才会注册进去。 那么为什么此处要这样用呢？ 这是因为如果用户已经实现了此接口并且已经注册的容器当中的话，就使用用户注册的，而不是中间件自身的。用户就能很好的对中间件提供的功能进行自定义了，这就是OO中的多态性，这就是**里氏替换**原则。

3、如果你能理解第2条的话，那么你应该知道为什么会在服务中注册 错误描述信息 那个IdentityErrorDescriber 了吧，也能够解决你想提示账号密码错误，无奈Identity输出是英文问题的提示。

4、三大对象，这个是 Identity 的核心了，所以学习 Identity 的话，在看完博客 ASP.NET Core 之 Identity 入门（[一](http://www.cnblogs.com/savorboard/p/aspnetcore-identity.html)，[二](http://www.cnblogs.com/savorboard/p/aspnetcore-identity2.html)）之后，学这三个对象就够了。

**SignInManager**： 主要处理注册登录相关业务逻辑。

**UserManager**： 处理用户相关添加删除，修改密码，添加删除角色等。

**RoleManager**：角色相关添加删除更新等。

有些同学可能很好奇，都没有依赖具体的数据库或者是EF，是怎么样做到的增删改查的呢？

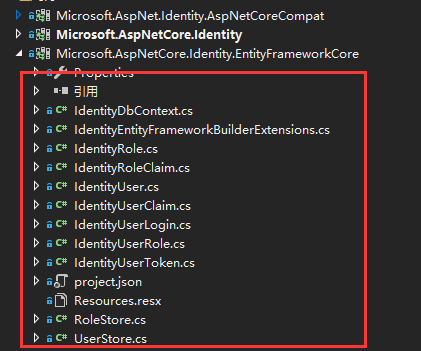
这个时候，就需要几个 Store 接口派上用场了。以下是Identity中定义的Store接口：

* IQueryableRoleStore
* IQueryableUserStore
* IRoleClaimStore
* IRoleStore
* IUserAuthenticationTokenStore
* IUserClaimStore
* IUserEmailStore
* IUserLockoutStore
* IUserLoginStore
* IUserPasswordStore
* IUserPhoneNumberStore
* IUserRoleStore
* IUserSecurityStampStore
* IUserStore
* IUserTwoFactorStore

有了这些接口之后，是不是豁然开朗了，原来 Identity 是通过这种方式实现的持久化机制，依赖抽象接口而不是依赖具体的细节实现，这就是面向对象中的**依赖倒置**原则呀。

### Identity 和 EntityFramework

Identity 和 EntityFramework的关系，相信上个章节看懂了之后，就很容易明白了，对的，EF 只是针对于上述 Store 接口的实现，不信你看截图的源码：



Identity 打头的那些类文件都是定义的需要持久化的Entity对象，Store结尾的那些就是接口的实现啦。

### 第三方的 Identity 实现

除了 EF 是官方默认提供的持久化库之外，还有一些第三方的库，当然你也可以自己使用 ADO.NET 或者 Drapper 实现。

MangoDb 针对于 Identity 提供的实现： <https://github.com/tugberkugurlu/AspNetCore.Identity.MongoDB>

LinqToDB 针对于 Identity 提供的实现：<https://github.com/linq2db/LinqToDB.Identity>

### 总结

这篇博文写了蛮久的时间的，一方面是因为在构思怎么样的思路来让大家更好的理解，而不仅仅是使用。因为有太多的文章介绍Identity 的使用方式以及代码了，但是最后大家还是不会用。后来想到如果让别人想要理解你的库也好代码也好，让其知道诞生的背景是很重要的，因为这才是设计的初衷。另一方面是因为Connect() 2016 大会上，.NET Core 发布了 1.1 版本，除了把项目升级到1.1之外，也在学习1.1新的一些东西，以便更好给大家分享。

授人以鱼不如授人以渔， 一篇好的博文确实要花费作者更多的心血和时间。如果你觉得这篇博文对您有帮助，感谢您的【推荐】。

同时，ASP.NET Core 之 Identity 系列也结束了，感谢支持的朋友们。如果你对 .NET Core 感兴趣可以关注我，我会定期在博客分享关于 .NET Core 的学习心得。

本文地址：<http://www.cnblogs.com/savorboard/p/aspnetcore-identity3.html>  
作者博客：[Savorboard](http://www.cnblogs.com/savorboard)  
欢迎转载，请在明显位置给出出处及链接