## 体系架构

遵循ddd4层

* **展现层(Presentation)**：向用户提供一个接口(UI)，使用应用层来和用户(UI)进行交互。
* **应用层(Application)**：应用层是表现层和领域层能够实现交互的中间者，协调业务对象去执行特定的应用任务
* **领域层(Domain)**：包括业务对象和业务规则，这是应用程序的核心层。
* **基础设施层(Infrastructure)**：提供通用技术来支持更高的层。例如基础设施层的仓储(Repository)可通过ORM来实现数据库交互，或者提供发送邮件的支持。

### 领域层（core 核心层）

领域层是实现所有业务规则的地方。

#### 实体（Entity）： 实体代表业务领域的数据和操作，在实践中，通常用来映射成数据库表。

#### **仓储（Repository）**： 仓储是像集合一样的对象，用来在数据源(数据库)上检索和存储实体。在领域层定义仓储，但是不实现它们。它们在基础设施层被实现。

#### **领域服务（Domain service）**： 当处理的业务规则跨越两个(及以上)实体时，应该写在领域服务方法里面。

#### **领域事件（Domain Event）**： 领域事件被用来定义特定于领域的事件，并且触发使用它们。领域服务与实体(以及其他领域对象)一起实现了不属于单个实体的业务规则。

#### **工作单元（Unit of Work）**： 工作单元是一种设计模式被用来管理数据库连接和事务，以及跟踪实体更改，并将这些更改保存到数据存储中。它被定义在领域层中，但是在基础设施层实现它们。

### 应用层（application）

应用层提供一些应用服务（Application Services）方法供展现层调用。一个应用服务方法接收一个DTO(数据传输对象)作为输入参数，使用这个输入参数执行特定的领域层操作，并根据需要可返回另一个DTO。在展现层到领域层之间，不应该接收或返回实体(Entity)对象，应该进行DTO映射。

一个应用服务方法通常被认为是一个工作单元（Unit of Work）。用户输入参数的验证工作也应该在应用层实现。ABP提供了一个基础架构让我们很容易地实现输入参数有效性验证。建议使用一种像AutoMapper这样的工具来进行实体与DTO之间的映射。

### 基础设施层

当在领域层中为定义了仓储接口，应该在基础设施层中实现这些接口。可以使用ORM工具，例如EntityFramework或NHibernate。ABP的基类已经提供了对这两种ORM工具的支持。数据库迁移也被用于这一层。

### Web与展现层

可分别用于多页面应用程序(MPA)和单页面应用程序(SPA)。

现在已经有很多SPA的JS框架，例如： AngularJs、 DurandalJs、BackboneJs、EmberJs。

1. ABP提供了一些帮助类，让我们更方便地使用AngularJs和DurandalJs。
2. SignalR是一种从服务器到客户端发送推送通知的完美工具。
3. ABP也实现了根据Web API接口自动创建 Javascript的代码函数，来简化JS对Web Api的调用。还有把服务器端的菜单、语言、设置等生成到JS端。

（但是在我自己的项目中，我是把这些自动生成功能关闭的，因为必要性不是很大，而这些又会比较影响性能）。

1. ABP会自动处理服务器端返回的异常，并以友好的界面提示用户。

### 其他

ABP使用Castle Windsor为整个程序框架提供依赖注入的功能。使用Log4Net日志记录组件，提供给其他各层调用以进行日志记录。

## 模块介绍

ABP框架提供了创建和组装模块的基础，一个模块能够依赖于另一个模块。在通常情况下，一个程序集就可以看成是一个模块。在ABP框架中，一个模块通过一个类来定义，而这个类要继承自AbpModule。

### Assembly程序集

Assembly程序集：Assembly是一个用来包含程序的名称，版本号，自我描述，文件关联关系和文件位置等信息的一个集合。最简单的理解就是：一个你自己写的类库生成的dll就可以看做是一个程序集，这个程序集可以包括很多类，类又包括很多方法等。

public class MyBlogApplicationModule : AbpModule //定义

{

public override void Initialize() //初始化

{

IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());

//这行代码的写法基本上是不变的。它的作用是把当前程序集的特定类或接口注册到依赖注入容器中。

}

}

ABP框架会扫描所有的程序集，并且发现AbpModule类中所有已经导入的类，如果你已经创建了包含多个程序集的应用，对于ABP，我们的建议是为每一个程序集创建一个Module（模块）。

### 生命周期事件

在一个应用中，ABP框架调用了Module模块的一些指定的方法来进行启动和关闭模块的操作。我们可以重载这些方法来完成我们自己的任务。

ABP框架通过依赖关系的顺序来调用这些方法，假如：模块A依赖于模块B,那么模块B要在模块A之前初始化，模块启动的方法顺序如下：

PreInitialize-B

* PreInitialize-A
* Initialize-B
* Initialize-A
* PostInitialize-B
* PostInitialize-A

#### PreInitialize（预初始化）

预初始化：当应用启动后，第一次运行会先调用这个方法。在依赖注入注册之前，你可以在这个方法中指定你需要注入的自定义启动类。举个例子吧：假如你创建了一个自定义的TestAbpSession : IAbpSession,那么请在这个方法中对TestAbpSession进行注册。

#### **Initialize(初始化)**

初始化：在这个方法中一般是来进行依赖注入的注册，一般我们通过IocManager.RegisterAssemblyByConvention这个方法来实现。如果你想实现自定义的依赖注入，那么请参考依赖注入的相关文档。

#### PostInitialize（提交初始化）

提交初始化：最后一个方法，这个方法用来解析依赖关系。

#### Shutdown

关闭：当应用关闭以后，这个方法被调用。

### 3．模块依赖

Abp框架会自动解析模块之间的依赖关系，但是我们还是建议你通过重载GetDependencies方法来明确的声明依赖关系。

[DependsOn(typeof(MyBlogCoreModule))]//通过注解来定义依赖关系

public class MyBlogApplicationModule : AbpModule

{

public override void Initialize()

{

IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());

}

}

我们就声明了MyBlogApplicationModule和MyBlogCoreModule的依赖关系（通过属性attribute），MyBlogApplicationModule这个应用模块依赖于MyBlogCoreModule核心模块，并且，MyBlogCoreModule核心模块会在MyBlogApplicationModule模块之前进行初始化。

### 插件模块

当模块从启动模块以及其依赖关系进行调查发现的时候，ABP也能够动态的加载其它指定模块。**AbpBootstrapper** 类定义了 **PlugInSources** 属性，我们能够是用该属性添加需要动态加载的模块。插件源可以是任何实现了 **IPlugInSource** 接口的类。**PlugInFolderSource** 类实现了该接口，它可以被用来加载指定文件夹下的程序集。

#### Net core 模块动态加载模块

你只需要在 **Startup** 类中使用已定义的扩展方法 **AddAbp**

services.AddAbp<MyStartupModule>(options =>

{

options.PlugInSources.Add(new FolderPlugInSource(@"C:\MyPlugIns"));

});

我们可以使用扩展方法 **AddFolder** 更方便的实现上述功能：

services.AddAbp<MyStartupModule>(options =>

{

options.PlugInSources.AddFolder(@"C:\MyPlugIns");

});

#### 自定义的模块方法如何调用

我们自己定义的模块中可能有方法被其他依赖于当前模块的模块调用，下面的例子，假设模块2依赖于模块1，并且想在预初始化的时候调用模块1的方法。这样，就把模块1注入到了模块2，因此，模块2就能调用模块1的方法了。

public class MyModule1 : AbpModule

{

public override void Initialize() //初始化模块

{

IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());//这里，进行依赖注入的注册。

}

public void MyModuleMethod1()

{

//这里写自定义的方法。

}

}

[DependsOn(typeof(MyModule1))]

public class MyModule2 : AbpModule

{

private readonly MyModule1 \_myModule1;

public MyModule2(MyModule1 myModule1)

{

\_myModule1 = myModule1;

}

public override void PreInitialize()

{

\_myModule1.MyModuleMethod1(); //调用MyModuleMethod1的方法。

}

public override void Initialize()

{

IocManager.RegisterAssemblyByConvention(Assembly.GetExecutingAssembly());

}

}

## 多租户

软件多租户是指一个软件架构的实例软件运行在一个服务器上，但存在多个租户。租户是一组共享一个公共的用户访问特定权限的软件实例。多租户架构,软件应用程序旨在提供每个租户专用的实例包括数据、配置、用户管理、租户个体功能和非功能属性。多租户与多实例架构,独立的软件实例代表不同的租户”操作多租户一般用来创建SaaS(软件即服务）应用程序（云计算）

### 开启多租户

默认多租户是被禁用的，我们需要在模块的 **PreInitialize** 方法中开启它，如下所示：

Configuration.MultiTenancy.IsEnabled = true;

### Host VS 租户

首先，我们先定义两个多租户系统中的术语：

* 租户：客户有它自己的用户,角色,权限,设置…并使用应用程序与其他租户完全隔离。多租户应用程序将有一个或多个租户。如果这是一个CRM应用程序,不同的租户也他们自己的帐户、联系人、产品和订单。所以,当我们说一个租户的用户,我们的意思是用户拥有的租户。
* Host: Host是单例的（只有唯一一个Host).Host负责创建和管理租户。所以Host用户独立与租户且可以控制租户。

### 3. Session

ABP定义IAbpSession接口来获取当前用户和租户id。这个接口中使用多租户当前租户的id。因此,它可以基于当前租户的id过滤数据。

这里有一些规则：

* 如果两个用户id和TenantId是null，那么当前用户没有登录到系统中。所以，我们不知道这是一个主机用户或租户的用户。在这种情况下，用户不能访问授权的内容。
* 用户id(如果不为空，TenantId为空的，然后我们可以知道当前用户是一个主机用户。
* 用户id(如果不为空，TenantId也不为空，我们可以知道当前用户是一个租户的用户。

有关更多的Session内容可查看：[Session](https://www.kancloud.cn/gaotang/abp/225828)

### 4. 数据过滤

当我们从数据库检索实体，我们必须添加一个TenantId过滤当前的租户实体。当你实现了接口：IMustHaveTenant或IMayHaveTenant中的一个时，ABP将自动完成数据过滤。

##### IMustHaveTenant Interface

这个接口通过TenantId属性来区分不同的租户的实体。示例：

public class Product : Entity, IMustHaveTenant

{

public int TenantId { get; set; }

public string Name { get; set; }

//...其它属性

}

因此，ABP能发现这是一个与租户相关的实体，并自动隔离其它租户的实体。

##### IMayHaveTenant interface

我们可能需要在Host和租户之间共享实体类型。一个实体可能属于租户或Host,IMayHaveTenant接口还定义了TenantId(类似于IMustHaveTenant),但在这种情况下可以为空。示例如下：

public class Role : Entity, IMayHaveTenant

{

public int? TenantId { get; set; }

public string RoleName { get; set; }

//...其它属性

}

我们可以使用相同的角色类存储主机角色和租户的角色。在这种情况下，TenantId属性会告诉我们这是一个Host实体还是一个租户实体。**null** 值意味着这是一个 **Host实体** ，一个 **非空值** 意味着这被一个租户拥有，该租户的Id是 **TenantId** 。

### 租户对数据库设置

当在多租户应用数据库上工作的时候，我们应该知道当前的租。默认获取租户ID的方式是从 IAbpSession 上获取的。我们可以改变这个行为并且切换到其它租户的数据库上

public class ProductService : ITransientDependency

{

private readonly IRepository<Product> \_productRepository;

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public ProductService(IRepository<Product> productRepository, IUnitOfWorkManager unitOfWorkManager)

{

\_productRepository = productRepository;

\_unitOfWorkManager = unitOfWorkManager;

}

[UnitOfWork]

public virtual List<Product> GetProducts(int tenantId)

{

using (\_unitOfWorkManager.Current.SetTenantId(tenantId))

{

return \_productRepository.GetAllList();

}

}

}

**SetTenantId** 方法确保我们得到的数据是指定租户的数据，这依赖于数据库架构：

* 1.如果给定的租户有特定的数据库，那么切换到这个数据库并且从该数据库中取得产品数据
* 2.如果给定的租户没有特定的数据库(例如：单数据库方式)，它会自动的添加TenantId条件到查询语句来过滤数据获取指定的租户的产品数据

如果我们没有使用SetTenantId方法，它会从Session中取得租户Id，如同之前所述。

这里有一些关于最佳实践的建议：

* 使用 **SetTenantId(null)** 切换到Host
* 如果没有特别的原因，你应该像上面示例所展示的一样，在using语句块中使用SetTenantId方法。因为它会在using语句块后且在 GetProducts 方法工作完成之前，自动的还原TenantId (也就是说using语句块运行完后，TenantId是从Session中获取的不会是来自于GetProducts的传入参数)
* 如果需要你可以嵌套使用SetTenantId方法
* 因为 **\_unitOfWorkManager.Current** 仅在工作单元中有效，请确保你的代码是在工作单元中运行

## 领域层-实体（Entity）

实体是 DDD（领域驱动设计）的核心概念之一。Eric Evans 是这样描述的“很多对象不是通过它们的属性定义的，而是通过一连串的连续性事件和标识定义的”（引用领域驱动设计一书）。

### 实体类

#### 继承于 Entity 类

（1）实体类重写了 **equality** (==) 操作符用来判断两个实体对象是否相等。

（2）还定义了一个 IsTransient()方法来检测当前 Id 的值是否与指定的类型的缺省值相等。

### 2.聚合根

#### 在领域驱动设计模式中聚合是一种设计模式

1. 聚合表示的是一组领域对象（包括实体和值对象）例如：订单和订单项，这都是单独的对象。但是，我们可以将订单(以及订单项)作为一个聚合来看待。
2. ABP不会强迫你使用聚合，你可以在你的应用中创建聚合以及聚合根。ABP定义了一个扩展自 **Entity** 的 **AggregateRoot** 类，用来创建聚合根实体。

(3) 领域事件 聚合根定义了DomainEvents的集合用来产生领域事件，在当前的工作单元完成之前，这些事件被自动触发。

(4) 事实上通过扩展IGeneratesDomainEvents接口，任何实体都能产生领域事件。但是，通常最佳实践是在聚合根中产生领域事件所以他被定义在聚合根中而不用是实体中。

### 3.接口约定

像实体一般具有像createTime的属性，用来指示该实体是什么时候创建的。Adp提供了一些有用的接口实现类似的功能。

#### (1)审计（Auditing）

1>.实现IHasCreationTime Adp自动设置该属性时间为当前时间。

public interface IHasCreationTime

{

DateTime CreationTime { get; set; }

}

public class Person : Entity<long>, IHasCreationTime

{

public virtual string Name { get; set; }

public virtual DateTime CreationTime { get; set; }

public Task()

{

CreationTime = DateTime.Now;

}

}

2> ICreationAudited 扩展自 IHasCreationTime 并且该接口具有属性 CreatorUserId

当保存一个新的实体时，ABP 会自动设置 CreatorUserId 的属性值为当前用户的 Id 。

public interface ICreationAudited : IHasCreationTime

{

long? CreatorUserId { get; set; }

}

3> 你可以很容易的实现 ICreationAudited 接口，通过派生自实体类 CreationAuditedEntity。它有一个实现不同 Id主键 数据类型的泛型版本。

当更新一个实体时，APB 会自动设置这些属性的值。你只需要在你的实体类里面实现这些属性。  
如果你想实现所有的审计属性，你可以直接扩展 IAudited 接口

作为一个快速开发方式，你可以直接派生自 AuditedEntity 类，不需要再去实现 IAudited 接口，AuditedEntity 类有一个实现不同 ID 数据类型的泛型版本(默认是 int)。

#### (2)逻辑删除（Soft delete）

逻辑删除是一个通用模式，它标记一个实体已经被删除了，而不是实际从数据库中删除记录。

我们可以实现该接口 ISoftDelete

public interface ISoftDelete

{

bool IsDeleted { get; set; }

}

ABP 会为我们自动过滤软删除的记录例如查询

实现IDeletionAudited 接口记录谁删除了这个数据

 IDeletionAudited 扩展自 ISoftDelete 接口。当一个实体被删除的时候 ABP 会自动的为这些属性设置值

#### (3)IFullAudited继承了所有审计接口

1>.所有的审计接口（例如：创建（creation），修改（modification）和删除（deletion）），你可以直接实现 IFullAudited 接口，因为该接口已经继承了这些接口

2>.快速开发可以直接从 FullAuditedEntity 类派生你的实体类，因为该类已经实现了 IFullAudited 接口。

## 领域层-值对象

用来描述领域的特殊方面，且没有标识的一个对象。

实体有自己的唯一标识，而值对象是没有标识的。如果两个实体的标识是不同的，那么它们是两个不同的实体，即使这两个实体的其它属性值是相同的

在DDD中，值对象是领域对象的另一种类型，它可以包含业务逻辑并且是领域中不可或缺的一部分。

### 1.DDD中的重要概念

实体entity 拥有唯一标识的一类对象，

但是对象不是通过他们的属性定义的，而是通过一连串的连续事件和标识定义的。

#### （1）值对象-value Object

1.对某个对象是什么不感兴趣，只关心它拥有的属性

2.用来描述领域的特殊方面、且没有标识符的一个对象，叫做值对象

3.能被简单的创建和丢弃，生命周期中不会被持久化

4.值对象可以被共享，值对象应该不可变

#### （2）服务-service（比webservice更细粒度服务描述）

1.领域中的一些动词，代表了领域中的一个重要的行为，却不属于任何对象

1)服务执行的操作涉及一个领域概念，这个领域概念通常不属于一个实体或者值对象

2)被执行的操作涉及到领域中的其他的对象

3)操作是无状态的

2.服务对象不再拥有内置的状态

3.服务对象担当重要的协调功能

4.开发通用语言时，领域中的主要概念被引入到语言中，语言中的名词很容易 被映射成对象

语言中对应那些名词的动词变成那些对象的行为。但是有些领域中的动作，它们是一些动词，看上去却不属于任何对象。它们代表了领域中的一个重要的行为，所以不能忽略它们或者简单的把它们合并到某个实体或者值对象中。给一个对象增加这样的行为会破坏这个对象，让它看上去拥有了本该属于它的功能。

#### （3）模块

1.将相关领域模型提炼分类，分而治之

2.将高关联度的模型分组到一个模块以提供尽可能大的内聚（以能完整完成任务为准）

3.分层是水平划分

4.模块是垂直划分(Domain内部)

#### （4）值对象基类

ABP有一个基类 **ValueObject<T>**，为了能够更方便的创建值对象，我们可以继承该基类。如：**Address** 值对象：

public class Address : ValueObject<Address>

{

public Guid CityId { get; private set; } //对城市实体的引用

public string Street { get; private set; }

public int Number { get; private set; }

public Address(Guid cityId, string street, int number)

{

CityId = cityId;

Street = street;

Number = number;

}

}

* 值对象应该被设计为不可变的(如上面的Address值对象)
* 构成一个值对象的属性应该形成一个概念上的整体。例如：CityId，街道和号码不应该是构成Person实体的一部分，应该分离出来作为Address值对象。这样也使Person实体更简单。

## 领域层-仓储（Repository）

“在领域层和数据映射层的中介,使用类似集合的接口来存取领域对象”(Martin Fowler)。

实际上，仓储被用于领域对象在数据库上的操作(实体Entity和值对象Value types)。一般来说,我们针对不同的实体(或聚合根Aggregate Root)会创建相对应的仓储。

### IRepository接口

在ABP中,仓储类要实现IRepository接口。最好的方式是针对不同仓储对象定义各自不同的接口。

针对Person实体的仓储接口声明的示例如下所示:

public interface IPersonRepository : IRepository<Person> { }

对于仓储类，IRepository定义了许多泛型的方法。比如: Select,Insert,Update,Delete方法(CRUD操作)。在大多数的时候,这些方法已足已应付一般实体的需要。如果这些方对于实体来说已足够,我们便不需要再去创建这个实体所需的仓储接口/类

#### 注意Load方法懒加载

Load并不会从数据库中检索实体,但它会创建延迟执行所需的代理对象。如果你只使用Id属性,实际上并不会检索实体,它只有在你存取想要查询实体的某个属性时才会从数据库中查询实体。

#### 自定义返回值

IQueryable<T>的延迟加载效果

## 领域层-领域服务（Domain service）

领域服务(或者服务，在DDD模式中)是被用来执行领域操作或者业务规则的。Eric Evans 在他的DDD书中这样说过：一个好的Service应该有

1．与领域概念相关的操作不是Entity或Value Object 的一个自然部分；

2. 接口是根据领域模型的其它元素定义的；

3. 操作是无状态的。

### （1）领域服务和Application Services 是不同的

1>.Application Services 返回的是DTO，而领域服务返回的是领域对象(实体或者值类型)。

2>领域服务可以被应用服务(application)和其它的领域服务调用，但是不可以被表现层直接调用(表现层(web)可以直接调用应用服务(application))。

### IDomainService和DomainService

定义了一个 IDomainService 接口，所有的领域服务都必须实现该接口(记住这是一个约定)，一旦实现了这个接口，那么领域服务就会通过Dependency Injection 自动的注册到系统中作为一个暂时对象(Transient)。

领域服务也可以继承自 DomainService 类(这是可选的)。  
因此，它可以用一些继承而来的属性来做日志记录，本地化等等；即使你不继承该类，如果你需要这些属性也是可以被注入的。

任务服务层（application）用给定的DTO和仓储资源去检索相关的Task和Person，并且将检索到的结果传递给TaskMananger(领域服务)（core service）。

## 领域层-工作单元（unit of work）

### ABP的连接和事务管理

#### 仓储类(Repository classes)

仓储是主要的数据库操作的类。ABP开启了一个数据库连接并且在进入到仓储方法时会启用一个事务。因此,你可以安全地使用连接于仓储方法中。在仓储方法结束后,事务会被提交并且会释放掉连接。假如仓储方法抛出任何异常,事务会被回滚并且释放掉连接。在这个模式中,仓储方法是单元性的(一个工作单元unit of work)。

如上所示,不需要撰写任何数据库连接操作(NHibernate中的Session)的程序代码。

假如仓储方法调用另一个仓储方法(一般来说,若工作单元方法调用另一个工作单元的方法),都使用同一个连接和事务。第一个被调用到的仓储方法负责管理连接和 事务,而其余被它调用的仓储方法则只单纯使用不管理。

#### 应用服务(Application service classes)

一个应用服务（**Application**）的方法也被考虑使用工作单元，应用服务自动开启事务

#### 工作单元(Unit of work)

##### 特性[UnitOfWork]

[UnitOfWork]

public void CreatePerson(CreatePersonInput input)

{

var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };

\_personRepository.Insert(person);

\_statisticsRepository.IncrementPeopleCount();

}

假如这是应用服务的方法则不需要添加UnitOfWork属性

##### 使用IUnitOfWorkManager.Begin(...)方法

public class MyService

{

private readonly IUnitOfWorkManager \_unitOfWorkManager;

private readonly IPersonRepository \_personRepository;

private readonly IStatisticsRepository \_statisticsRepository;

public MyService(IUnitOfWorkManager unitOfWorkManager, IPersonRepository personRepository, IStatisticsRepository statisticsRepository) {

\_unitOfWorkManager = unitOfWorkManager;

\_personRepository = personRepository;

\_statisticsRepository = statisticsRepository;

}

public void CreatePerson(CreatePersonInput input) {

var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };

using(var unitOfWork = \_unitOfWorkManager.Begin()) {

\_personRepository.Insert(person);

\_statisticsRepository.IncrementPeopleCount();

unitOfWork.Complete();

}

}

}

##### 禁用工作单元(Disabling unit of work)

想要禁用应用服务方法的工作单元(因为它默认是启用的)。要想做到这个,使用UnitOfWorkAttribute的IsDisabled属性

[UnitOfWork(IsDisabled = true)]

public virtual void RemoveFriendship(RemoveFriendInput input)

{

\_friendshipRepository.Delete(input.Id);

}

注意,如果工作单元方法调用这个RemoveFriendship方法,禁用被忽略且它和调用它的方法使用同一个工作单元。因此,使用禁用这个功能要很小心。同样地,上述程序代码工作的很好,因为仓储方法默认即为工作单元。

##### 工作单元调用其它工作单元(A unit of work method calls another)

若工作单元方法(一个贴上UnitOfWork属性标签的方法)调用另一个工作单元方法,他们共享同一个连接和事务。第一个方法管理连接,其它的方法只是使用它。这在所有方法都执行在同一个线程下是可行的(或是在同一个Web请求内)。实际上,当工作单元区域开始,所有的程序代码都会在同一个线程中执行并共享同一个连接事务,直到工作单元区域终止。这对于使用UnitOfWork属性和UnitOfWorkScope类来说都是一样的。如果你创建了一个不同的线程/任务,它使用自己所属的工作单元

##### 在startup configuration中改变所有工作单元的所有默认值

通常是用了我们模块中的PreInitialize方法来实现

public class SimpleTaskSystemCoreModule : AbpModule

{

public override void PreInitialize() {

Configuration.UnitOfWork.IsolationLevel = IsolationLevel.ReadCommitted;

Configuration.UnitOfWork.Timeout = TimeSpan.FromMinutes(30);

}

//...其它模块方法

}

## 领域层 - 领域事件(**Domain Event**)

一个类可以定义其专属的事件并且其它类可以注册该事件并监听，当事件被触发时可以获得事件通知。这对于对于桌面应用程序或独立的Windows Service来说非常有用。但是, 对于Web应用程序来说会有点问题,因为对象是根据请求(request)被创建并且它们的生命周期都很短暂。我们很难注册其它类别的事件。同样地,直接注册其它类别的事件也造成了类之间的耦合性

### 事件总线

事件总线为一个单体(singleton)的对象,它由所有其它类所共享,可通过它触发和处理事件。要使用这个事件总线,你需要引用它

#### 注入IEventBus事件接口

你可以使用依赖注入来获取对 **IEventBus** 的引用。这里使用的是属性注入的方式：

public class TaskAppService : ApplicationService

{

public IEventBus EventBus { get; set; }

public TaskAppService()

{

EventBus = NullEventBus.Instance;

}

}

#### 获取默认实例

如果你不能注入它，你可以直接使用 **EventBus.Default**。它是全局事件总线并且可以以如下方式使用:

EventBus.Default.Trigger(...); //触发事件

无论在什么地方，建议你尽可能的别直接使用EventBus.Default，因为那样使单元测试会很困难。

### 定义事件

触发事件之前，首先你应该定义该事件。事件是派生自 **EventData** 的类。假设我们想要触发一个事件，当某个任务完成的时候：

public class TaskCompletedEventData : EventData

{

public int TaskId { get; set; }

}

这个类包含了处理这个事件所需要包含的属性。**EventData** 定义了 **EventSource** (那个对象触发了该事件) 以及 **EventTime** (什么时候触发的) 属性。

### 处理异常

要进行事件的处理,你应该要实现IEventHandler<T>接口如下所示:

public class ActivityWriter : IEventHandler<TaskCompletedEventData>, ITransientDependency {

public void HandleEvent(TaskCompletedEventData eventData) {

WriteActivity("A task is completed by id = " + eventData.TaskId);

}

}

EventBus已集成到依赖注入系统中。就如同我们在上例中实现ITransientDependency那样,当TaskCompleted事件触发,它会创建一个新的ActivityWriter类的实体并且调用它的HandleEvent方法,并接着释放它。详情请见依赖注入(DI)一文

事件总线触发所有异常处理器，即使抛出异常是这些中的某个或者某些。如果你仅想抛出这些异常中的某一个，那么你可以直接通过该触发器方法抛出异常。如果超过一个的处理器抛出异常，那么事件总线会对这些异常抛出一个单独的 **AggregateException**。

### 注册处理器

必需注册处理器(handler)到事件总线中来处理事件。

#### 1. 自动型Automatically

ABP扫描所有实现IEventHandler接口的类,并且自动注册它们到事件总线中。当事件发生, 它通过依赖注入(DI)来取得处理器(handler)的引用对象并且在事件处理完毕之后将其释放。这是比较建议的事件总线使用方式于ABP中。

#### 2. 手动型(Manually)

也可以通过手动注册事件的方式,但是会有些问题。在Web应用程序中,事件的注册应该要在应用程序启动的时候。当一个Web请求(request)抵达时进行事件的注册,并且反复这个行为。这可能会导致你的应用程序发生一些问题,因为注册的类可以被调用多次。同样需要注意的是,手动注册无法与依赖注入系统一起使用。

ABP提供了多个事件总线注册方法的重载(overload)。最简单的一个重载方法是等待委派(delegate)或Lambda

EventBus.Register<TaskCompletedEventData>(eventData =>

{

WriteActivity("A task is completed by id = " + eventData.TaskId);

});

因此,事件:task completed会发生,而这个Lambda方法会被调用。第二个重载方法等待的是一个对象,该对象实现了IEventHandler<T>:

Eventbus.Register<TaskCompletedEventData>(new ActivityWriter());

相同的例子,如果ActivityWriter因事件而被调用。这个方法也有一个非泛型的重载。另一个重载接受两个泛化的参数:

EventBus.Register<TaskCompletedEventData, ActivityWriter>();

此时,事件总线创建一个新的ActivityWriter于每个事件。当它释放的时候,它会调用ActivityWriter.Dispose方法。

最后,你可以注册一个事件处理器工厂(event handler factory)来负责创建处理器。处理器工厂有两个方法: GetHandler和ReleaseHandler,范例如下:

public class ActivityWriterFactory : IEventHandlerFactory {

public IEventHandler GetHandler() {

return new ActivityWriter();

}

public void ReleaseHandler(IEventHandler handler) {

//TODO: 释放ActivityWriter实体(处理器)

}

}

ABP也提供了特殊的工厂类,IocHandlerFactory,通过依赖注入系统，IocHandlerFactory可以用来创建或者释放(dispose)处理器。ABP可以自动化注册IocHandlerFactory。因此,如果你想要使用依赖注入系统,请直接使用自动化注册的方式。

### 取消注册事件

当你手动注册事件总线,你或许想要在之后取消注册。最简单的取消事件注册的方式即为registration.Dispose()。举例如下:

//注册一个事件

Var registration = EventBus.Register<TaskCompletedEventData>(eventData => WriteActivity("A task is completed by id = " + eventData.TaskId));

//取消注册一个事件

registration.Dispose();

事件总线也提供取消注册方法。使用范例:

//创建一个处理器

var handler = new ActivityWriter();

//注册一个事件

EventBus.Register<TaskCompletedEventData>(handler);

//取消这个事件的注册

EventBus.Unregister<TaskCompletedEventData>(handler);

### 获取abp全局异常

ABP定义AbpHandledExceptionData事件，并且在异常发生的时候自动地触发这个事件。这在你想要取得更多关于异常的信息时特别有用(即便ABP已自动地纪录所有的异常)。你可以注册这个事件来发出通知当某个异常发生的时候

### 实体变更

对于实体变更，ABP也定义了一下事件的泛型版本: **EntityCreatingEventData<TEntity>, EntityCreatedEventData<TEntity>, EntityUpdatingEventData<TEntity>, EntityUpdatedEventData<TEntity>, EntityDeletingEventData<TEntity> and EntityDeletedEventData<TEntity>**。还有，对于泛型事件：**EntityChangingEventData<TEntity> 和 EntityChangedEventData<TEntity>** 是在某个实体发生插入，更新或者删除的时候出发。

#### Ing事件

指（EntityUpdating）该类事件被触发是在保存实体更改到数据库之前，所以你可以在这些事件里面抛出某个异常来回滚工作单元来阻止当前的更改操作

#### **ed** 事件

事件是指(例如：EntityUpdated) 该类事件被触发是在保存实体更改到数据库之后，这样就不可能有机会回滚工作单元。

#### 实体事件

实体更改事件被定义在 **Abp.Events.Bus.Entities** 命名空间，并且在某个实体被插入，更新或者删除的时候，ABP可以自动的触发它们

你有一个Person实体,可以注册 **EntityCreatedEventData<Person>** 事件来产生通知，当一个新的的Person实体被创建且插入到数据库的时候，ABP就会自动的触发该事件。

这些事件也支持继承。如果Student类继承自Person类，并且你注册了 **EntityCreatedEventData<Person>** 事件,接着你将会在Person或Student实体被插入后后收到触发。

#### 触发事件

public class TaskAppService : ApplicationService

{

public IEventBus EventBus { get; set; }

public TaskAppService() {

EventBus = NullEventBus.Instance;

}

public void CompleteTask(CompleteTaskInput input) {

//TODO: 已完成数据库上的任务

EventBus.Trigger(new TaskCompletedEventData { TaskId = 42 } );

}

}

EventBus.Trigger<TaskcompletedEventData>(new TaskCompletedEventData { TaskId = 42});

EventBus.Trigger(this, new TaskCompletedEventData { TaskId = 42 });

EventBus.Trigger(typeof(TaskCompletedEventData), this, new TaskCompletedEventData { TaskId = 42});

## 应用层-应用服务（application Service）

应用服务(application)用于将领域(业务)(domain \core)逻辑暴露给展现层(web)。展现层(web)通过传入DTO(数据传输对象)参数来调用应用服务(application)，而应用服务(application)通过领域对象(domain \core)来执行相应的业务逻辑并且将DTO返回给展现层。因此，展现层和领域层将被完全隔离开来。在一个理想的层级项目中，展现层应该从不直接访问领域对象。

### IApplicationService接口

在ABP中，一个应用服务需要实现 **IApplicationService** 接口。最好的实践是针对每个应用服务都创建相应的接口。所以，我们首先定义一个应用服务接口，如下所示：

public interface IPersonAppService : IApplicationService

{

void CreatePerson(CreatePersonInput input);

}

**IPersonAppService** 只有一个方法，它将被展现层调用来创建一个新的Person。**CreatePersonInput** 是一个DTO对象，如下所示：

public class CreatePersonInput

{

[Required]

public string Name { get; set; }

public string EmailAddress { get; set; }

}

接着，我们实现IPersonAppService接口:

public class PersonAppService : IPersonAppService

{

private readonly IRepository<Person> \_personRepository;

public PersonAppService(IRepository<Person> personRepository)

{

\_personRepository = personRepository;

}

public void CreatePerson(CreatePersonInput input)

{

var person = \_personRepository.FirstOrDefault(p => p.EmailAddress == input.EmailAddress);

if (person != null)

{

throw new UserFriendlyException("There is already a person with given email address");

}

person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };

\_personRepository.Insert(person);

}

}

1. PersonAppService通过IRepository<Person>来执行数据库操作。它通过构造器注入模式来生成。我们在这里使用了依赖注入。
2. PersonAppService实现了 **IApplicationService** (通过IPersonAppService继承IApplicationService)。ABP会自动地把它注册到依赖注入系统中，并可以注入到别的类型中使用
3. **CreatePerson** 方法需要一个 **CreatePersonInput** 类型的参数。这是一个作为输入的DTO，它将被ABP自动验证其数据有效性。可以查看DTO和数据有效性验证(Validation)文档获取相关细节。

### 应用服务类

继承: AsyncCrudAppService<Task, TaskDto>

可以实现基本的增删改查

public class TaskAppService : AsyncCrudAppService<Task, TaskDto>

{

public TaskAppService(IRepository<Task> repository)

: base(repository)

{

}

}

## 应用层-数据验证

### 使用数据注解

在这里 Description 属性被标记为 Required。AssignedPersonId 是可选的。在 System.ComponentModel.DataAnnotations 命名空间中，还有很多这样的特性 ( 例如： MaxLength, MinLength, RegularExpression 等等 )。如下所示：

### 自定义检验

如果数据注解的方式不能满足你的需求，你可以实现ICustomValidate接口，请看下面示例：

public class CreateTaskInput : ICustomValidate

{

public int? AssignedPersonId { get; set; }

public bool SendEmailToAssignedPerson { get; set; }

[Required]

public string Description { get; set; }

public void AddValidationErrors(CustomValidatationContext context)

{

if (SendEmailToAssignedPerson && (!AssignedPersonId.HasValue || AssignedPersonId.Value <= 0))

{

context.Results.Add(new ValidationResult("AssignedPersonId must be set if SendEmailToAssignedPerson is true!"));

}

}

}

**ICustomValidate** 接口声明了一个可被实现的 **AddValidationErrors** 方法。如果有验证错误的话，我们必须添加 **ValidationResult** 对象到 **context.Results** 列表。如果需要的话，你也可以使用 **context.IocResolver** 来解决依赖关系。

除了 **ICustomValidate** 接口，ABP也对.NET标准的 **IValidatableObject** 接口提供支持。你也可以实现它来执行自定义验证。如果你实现了这两个接口，那么这两个接口的方法都会被调用。

### 禁用验证

对于自动验证类，你可以使用这些特性来控制验证：

* **DisableValidation** 特性能够被用来对DTO的类，方法或者属性来禁用验证
* **EnableValidation** 特性仅能够被用来开启都某个方法的验证；如果对某个类禁用验证，但是想对该类的某个方法开启验证

### 标准化（数据组织完成后会调用的方法）

在验证数据输入后，我们需要执行一个额外的操作来组织DTO参数。ABP定义了一个 **IShouldNormalize** 接口，这个接口声明了一个 **Normalize** 的方法。如果你实现了这个接口，在数据验证后，**Normalize** 方法就会被调用。假设我们的DTO需要一个排序方向的属性。如果这个Sorting属性没有值，但是我们想要设置一个默认值；如下所示：

public class GetTasksInput : IShouldNormalize

{

public string Sorting { get; set; }

public void Normalize()

{

if (string.IsNullOrWhiteSpace(Sorting))

{

Sorting = "Name ASC";

}

}

}

## 应用层-权限认证

几乎所有的企业级应用程序都会有不同级别的权限验证。权限验证是用于检查用户是否允许某些指定操作。Abp有基础设施让你来实现权限验证。

### 定义权限

在使用验证权限前，我们需要为每一个操作定义唯一的权限。Abp的设计是基于模块化，所以不同的模块可以有不同的权限。为了定义权限，一个模块应该创建AuthorizationProvider的派生类。MyAuthorizationProvider继承自AuthorizationProvider，换句话说就是AuthorizationProvider派生出MyAuthorizationProvider

public class MyAuthorizationProvider : AuthorizationProvider

{

public override void SetPermissions(IPermissionDefinitionContext context)

{

var administration = context.CreatePermission("Administration");

var userManagement = administration.CreateChildPermission("Administration.UserManagement");

userManagement.CreateChildPermission("Administration.UserManagement.CreateUser");

var roleManagement = administration.CreateChildPermission("Administration.RoleManagement");

}

}

IPermissionDefinitionContext 有方法去获取和创建权限。

* Name：系统范围内的唯一名字。把它定义为一个字符串常量是个不错的注意。我们倾向于将“.”分割不同的层级，但并不要求这么做。你可以设置你任何喜欢的名字。唯一的规则就是这个名字必须是唯一的。
* Display Name：使用一个本地化的字符串去显示权限到UI。
* Description：和Display Name类似。
* MultiTenancySides：对租户应用程序，一个权限可以基于租户或者主机(原文:host)。这是个枚举标识，因此权限可以应用于不同方面(原文：Both Sides)。
* dependedFeature：能够被用来描述某个被依赖的功能。那么，只有在该依赖的功能被满足的情况下该权限才被授予

### 2.注册权限

一个权限可以有父权限和子权限。当然，这不会影响权限检查，它只是在UI层对权限归类有好处。创建authorizationprovider之后，我们应该在模块的PreIntialize方法对它进行注册。如下：

Configuration.Authorization.Providers.Add<MyAuthorizationProvider>()

authorizationprovider会自动注册到依赖注入系统中。因此，authorization provider可以注入任何依赖（像是Repository）从而使用其他资源去创建权限定义。

### 3.检查权限

#### 使用AbpAuthorize特性(Using AbpAuthorize attribute)

AbpAuthorize（AbpMvcAuthorize 对应 MVC Controllers and AbpApiAuthorize 对应 Web API Controllers）特性是最简单和常用的方法去检查权限

[AbpAuthorize("Administration.UserManagement.CreateUser")]

public void CreateUser(CreateUserInput input)

{

//A user can not execute this method if he is not granted for "Administration.UserManagement.CreateUser" permission.

}

没有获得“Administration.UserManagement.CreateUser”权限的用户不能够调用CreateUser。

AbpAuthorize 特性也检查当前用户是否登录 (使用 IAbpSession.UserId)。因此，如果我们将某个方法声明为AbpAuthorize 特性，它至少会检查用户是否登录

[AbpAuthorize]

public void SomeMethod(SomeMethodInput input)

{

//A user can not execute this method if he did not login.

}

##### **AbpAuthorize属性说明(AbpAuthorize attribute notes)**

* 不能应用于私有(private)方法
* 不能应用于静态(static)方法
* 不能应用于非注入(non-injected)类(我们必须用依赖注入)。

此外，

* AbpAuthorize特性可以应用于任何的Public方法，如果此方法被接口调用(比如在Application Services中通过接口调用)
* 方法是虚(virtual)方法，如果此方法直接被类引用进行调用([像是ASP.NET](http://xn--asp-ve1e609g.net/) MVC 或 Web API 的控制器)。
* 方式是虚(virtual)方法，如果此方法是protected。

#### **使用IPermissionChecker**

AbpAuthorize 适用于大部分的情况，但是某些情况下，我们还是需要自己在方法体里进行权限验证。我们可以注入和使用IPermissionChecker对象。如下边的代码所示：

public void CreateUser(CreateOrUpdateUserInput input)

{

if (!PermissionChecker.IsGranted("Administration.UserManagement.CreateUser"))

{

throw new AbpAuthorizationException("You are not authorized to create user!");

}

//如果该用户没有 "Administration.UserManagement.CreateUser" 的权限，那么它就不能运行到这个地方。

}

当然，你可以写入任何逻辑，由于IsGranted方法只是简单返回true或false（它还有异步版本哦）。如你简单的检查一个权限并抛出一个异常如上边代码那样，你可以用Authorize方法：

public void CreateUser(CreateOrUpdateUserInput input)

{

PermissionChecker.Authorize("Administration.UserManagement.CreateUser");

//如果该用户没有 "Administration.UserManagement.CreateUser" 的权限，那么它就不能运行到这个地方。

}

由于权限验证通常实现于Application层，ApplicationService基础类注入和定义了PermissionChecker属性。因此，权限检查器允许你在Application Service类使用，而不需要显式注入。

#### 客户端检验权限

我们可以使用定义在 **abp.auth** 命名空间中的API。在大多数情况下，我们需要检查当前用户时候具有指定的权限。例如：

abp.auth.isGranted('Administration.UserManagement.CreateUser');