**EC-Central项目**

**开发事项Guideline**

Benny.T.Yang

2012-3-30

# 一、 编写目的

本文档的目的是用于指导开发人员了解整个项目中需要遵守的一些开发规范，同时了解项目中提供的一些基础工具类。

# 二、注意事项介绍

## （一）Domain之间交互访问通过接口隔离；

## （二）业务相关操作不能直接跨Domain访问数据库；（具体执行方法请参见《EC-Central项目整体技术架构设计原则和思路》）

## （三）业务层次使用业务相关类型时必须使用我们的工厂来获取实例对象；

## （四）业务相关操作和非业务查询需要隔离；

* 业务相关的操作 --- 所有的Action（包含写的），以及业务过程需要的Load等读操作（如根据SysNo来Load Master或者Load Transaction List）
* 业务无关的查询 --- UI上由“查询条件” + “查询按钮”这类所发起的单向读取的请求操作

在系统中的体现：

* 业务无关的查询：Restful 🡪 IDataAccess 🡪 SqlDataAccess ，将直接绕过AppService和BizProcessor，并且在DataAccess层直接使用DataTable，在Restful Service返回QueryResult，在Silverlight端直接使用dynamic对象
* 业务相关的操作: Restful 🡪 AppService 🡪 BizProcesser 🡪 IDataAccess 🡪 SqlDataAccess，在Service端各个层之间使用BizEntity来传输数据，在Silverlight端也有BizEntity的映射对应实体

## （五）Appservice，BizProcesser层中的public method，严禁使用静态方法，这三层中所有public方法都需要加上virtual修饰符；

## （六）多语言支持，无论UI还是Service端，在使用需要展示给用户看的文本信息时，都需要考虑多语言支持；（会提供相应的工具类）

## （七）在整个系统中，如果没有特殊需求的情况下，都不能用try{ }catch{ }消化掉异常，而需要向上抛出异常；

## （八）对于方法执行，通过有无异常（Exception）抛出来通知调用者方法执行成功或失败；

一般情况下，对于方法执行不允许用方法的返回值或者出参来告诉调用者方法执行成功或失败，而是通过有误异常抛出的方式来告诉调用者方法执行成功或失败；对于业务异常，请使用BizException，并且将需要提示用户的业务信息通过构造函数放到Message属性中；

## （九）业务异常需要明确清晰，能告诉客户出错原因以及如何修改输入以保证下次执行成功；

## （十）在Service端对于非业务相关的查询需要单独隔离，不能和业务相关操作混在一起；并且注意下述的注意事项：

对非业务相关的查询在DataAccess层请直接返回DataTable，在Restful Service上则请返回ECCentral.Service.Utility.WCF.QueryResult或QueryResultList对象；QueryResult对象会有个属性指向装载数据的DataTable；对应的在UI Portal端，请使用RestClient的QueryDynamicData方法来获得一个dynamic对象使用，对于没有交换的界面可以直接UI绑定该dynamic对象，只是xaml的绑定属性需要加上方括号，如[Name]，而如果有交互的UI那么还是请定义相应的ViewModel，并做dynamic对象和VM直接的数据转换；**（会提供相应的工具类）**

如果服务端需要一次返回多个二维表的数据，或者是除了返回一张表数据外还需要返回其他一些附加数据，那么此时可以使用ECCentral.Service.Utility.WCF.QueryResultList对象，该对象继承于List<QueryResult>，对应的在UI Portal端，仍然使用RestClient的QueryDynamicData方法来获得一个dynamic对象使用，只是返回的dynamic对象本身就是一个数组，需要通过索引器来访问。

## （十一）在整个系统中，都请使用强类型的枚举类型；

使用强类型的枚举而不是字符code或int值的方式来表示业务上的枚举量；在UI上会使用EnumConverter来实现枚举值和具体显示文本的转换，在DataAccess上会通过数据转换来将枚举值映射到DB字段的相应类型的值上；

## （十二）枚举类型和配置的使用区分：

在我们的业务范畴内不变的，建议用枚举类型，否则请使用业务配置，如果配置需要有UI并允许最终客户来修改，那么需要配置到数据库，否则可以配置到数据库也可以直接配置到业务配置文件中，建议在这种情况配置到文件，代码会简单些；

## （十三）枚举类型的归属和公用问题：

和业务相关的枚举的归属定义应该和实体的归属定义一样，需要按Domain来划分归属，每个Domain只能定义属于自己业务范围的实体的相关枚举，而不能去定义属于其他Domain的枚举，比如客户等级的枚举，也许在SO或者RMA里都会用到，但这个业务明显属于Customer Domain，所以这个枚举应该由CustomerDomain去定义，然后在SO和RMA Domain里可以引用来使用；保证这种业务相关的枚举，只有一份，并且定义在自己业务所属的Domain里；

**注：如果2个Domain都有用到类似的业务枚举项，那么如何判断它们两个枚举项是不是业务意义上的同一个呢？最简单的判断方式就是假设其中一个Domain的枚举项有了变化（比如增加选项），那么另外一个Domain的类似的枚举项是否需要跟随变化，如果需要，那么它们就是一个枚举，就应该只定义在某一个Domain里就好了；如果不是跟随变化，那么说明2个枚举项只是碰巧名字相似而已，但本质业务上是不同的2个东西，那么此时就应该分别定义2个枚举类型了；**

而对于没有业务的一些枚举，比如单纯的有效无效、是与否、性别之类的枚举，这些枚举都以Domain为范围来处理共用，每个Domain考虑自己内部的共用即可，而不用考虑说需要所有Domain都来共用；

## （十四）枚举类型与数据库存储的字符或数字之间的映射关系处理：

每个Domain都需要关注自己Domain所定义的枚举和DB存储数据直接的转换映射关系：（其他Domain所定义的枚举的映射关系由其他Domain自己去处理）

(1). 如果数据库中存储的就是枚举自身的int值，那么这个映射不用另外专门再定义了

(2). 如果数据库中存储的是char或者所存的int的值与枚举自身的int值不一致，那么就需要定义一个枚举和数据库具体存储数据之间的映射关系，这个映射关系只能也必须定义在每个Domain自己的SqlDataAccess工程下EnumCodeMapRegister类型中，通过EnumCodeMapper.AddMap<TEnum>方法来定义具体的映射关系；

这样我们的ECCentral.Service.Utility.DataAccess中的DataCommand类型的void SetParameterValue<T>(T entity)方法和ExecuteEntity<T>方法就可以直接根据这个映射关系来完成数据库数据和枚举直接的相互转换工作；

**注：有个原则，一个枚举是由哪个Domain定义的，那么就只能在这个Domain的DataAccess工程里用EnumCodeMapRegister进行注册，不能在某个Domain的DataAccess工程里去注册定义在其他Domain里的Enum；**

## （十五）业务字典类的对象，必须直接使用该对象，以示语义明确，不能用CodeNamePair或KeyValuePair<T1，T2>代替，比如Company，WebChannel等；

## （十六）标准化产品，所以代码里编码出现硬编码的和Newegg、Ozzo相关的字符：

在IPP3中有些枚举值的名字、对象属性的名字里会出现NEG、Newegg、OZZO等相关字符，这次我们要避免再使用；

**（十七）Portal端各Domain内需要调用其它Domain的Restful Service的地方，统一放在OtherDomainDataFacade.cs 中（也就是说，每个Domain的Facade下都有一个OtherDomainDataFacade类）**

## （十八）关于Restful Service：

1. 因为所选择的技术工具原因，当前的Restful Service的方法只支持单个入参，并且不支持out、ref的参数；
2. 在Restful Service的方法定义时，如果入参和返回值使用已有的BizEntity是足够的时候，请直接使用BizEntity；
3. 当UI上对入参和返回值要求的数据比较复杂时，没有BizEntity能满足时，请按需定义ReqeustMsg和ResponseMsg；
4. 禁止因为为了满足UI上对Restful Service的入参和出参的数据要求而来修改BizEntity的属性定义；

## （十九）服务端的异常调试方法：

对于服务端发生比较奇怪的异常时，比如服务无效或者服务找不到之类的错误（因为对于WCF框架底层发生的异常，是无法被正确传输到客户端，在客户端看到的错误信息可能都是服务无效或者服务找不到的错误），可以用debug方式，在ECCentral.Service.Utility.WCF工程的ServiceBehavior目录下的RestServiceErrorHandler类型的HandleError方法里加断点来调试，在这里就能看到所有的异常，包括WCF框架自己底层抛出的异常，如前面提到的JSON序列的错误等都能看到真实信息，以方便大家定位和修复错误。

## （二十）BizEntity中定义的实体类型的属性，其属性类型如果是.net的值类型的，都需要定义为其对应的Nullable<>泛型类型，比如int的就应该定义为int?；

## （二十一）针对BizEntity有5个基本的接口：IIdentity、ICompany、ILanguage，IWebChannel、IMarketingPlace，设计目的如下说明：

* IIdentity接口，里面就一个属性SysNo，实现该接口的类型应该都是领域模型里的实体，拥有唯一标识和自己的状态，对于非实体的领域模型，也就是值，可以不用实现该接口，因为值模型不需要唯一标识（这也意味着值模型不能单独进行持久化，必须跟随聚合了它的实体模型一起进行持久化）
* ICompany接口，里面就一个属性CompanyCode，实现该接口的实体类型表示其业务数据都需要按Company进行切分的，对于不需要按Company进行切分数据的（比如“全国的行政区域”这种客观唯一的数据）就不用实现该接口
* ILanguage接口，里面就一个属性LanguageCode，实现该接口的实体类型表示数据都需要按照语言进行切分的，比如Product的BasicInfo，而对于不需要分语言的，就不用实现该接口
* IWebChannel接口，里面就一个属性WebChannel，实现该接口的实体表示其数据都要按照WebChannel来切分，比如Product的Price信息
* IMarketingPlace接口，里面就一个属性Merchant，实现该接口的实体就表示其数据都要按照Merchant来切分，比如订单SO

这样我们整个实体模型会更清晰，使用起来也会更得当更灵活；

当然我们这样做，可能会带来的一个问题，就是我们的实体在持久化的时候，可能会和DB的字段对不上，因为我们的DB里不管三七二十都加了CompanyCode、LanguageCode的字段，而我们的业务实体可能根据业务分析后决定不需要LanguageCode，此时持久化的时候就会缺个字段的数据；我们建议这个时候可以在DataAccess层里直接为缺的这个字段填个默认值好了（比如填入当前线程的语言），因为既然我们在领域实体上都没定义这个字段，那就说明这个字段对我们的业务确实没有用，所以填什么值都无所谓了。

## （二十二）Silverlight UI的相关要求：

详见*$/ECCentral/02\_Design/01\_设计与开发指南*下的《7.EC-Central项目界面设计规范与注意事项.docx》

## （二十三）服务端的上下文对象ServiceContext，可以用来获取当前执行上下文的用户唯一编号和用户客户端IP地址；

因为我们现在Service端使用的WCF应用的宿主环境，所以内部使用的WCF的OperationContext来实现的，所以如果是更换了程序运行宿主环境，比如Service端程序是运行在ASP.Net的应用，或者Winform应用里（比如做UnitTest的时候），那么此时需要根据新的环境重新实现一个IContext（如果是asp.net应用那基本上就用session或cookie来实现，winform程序用static变量就可以了），然后再在web.config或app.config文件里的appSetings下加一个配置：<add key="ServiceContextType" value="类型带程序集的全名" />

## （二十四）关于常量的定义和使用：

* 在Service端，常量都定义在ECCentral.BizEntity.Enum工程里；该工程里为每个Domain都建立了一个专门用来定义自己Domain的Const的静态类文件，请大家把自己Domain需要用到的Const常量都定义到该静态类里。（另外要注意，在Service端跨Domain不要共享常量，宁可每个Domain都冗余定义一次。）
* 在Portal端，所有Domain的常量都放在ECCentral.Portal.Common下的ConstValue中。

## （二十五）关于事务的使用：

当前我们采用的All in one的方式部署我们的Service，同时我们的Transaction Central DB也都是在同一个DB Server实例上（不同的库），所以我们现在在代码里直接采用微软.Net类库中的TransactionScope类的方式来做事务保证（因为是同一个DB Server Instance，所以我们会使用同一个SqlConnection，保证本地事务，而不会打开分布式事务）；  
我们只能在每个Domain的BizProcessor和AppService这2个工程里使用事务，请不要在程序的其他地方再引入事务；

后期如果我们要横向扩展Service（不同的Domain的Service运行在不同进程里）或拆分DB，那么可能需要找出所有使用了TransactionScope的地方，然后根据扩展或拆分方案，逐个来Review这些事务的地方该如何来处理（比如可以异步操作的地方，考虑用消息队列处理方式，如MSMQ或SSB；而必须要同步操作的地方，要考虑即时的PreCheck加Retry机制等等，需要具体问题具体考虑）；

## （二十六）关于缓存的使用：

我们可以通过配置文件来配置相应的缓存提供者（可参考VSTS中*$/ECCentral/02\_Design/01\_设计与开发指南*目录下的《6.EC-Central项目配置文件说明.docx》文档中相关说明），当前我们提供了基于.Net 4.0的MemoryCache作为本地缓存和基于Memcached 1.4.5的分布式缓存；

在代码中使用缓存有2种方式：（具体使用请参考VSTS中*$/ECCentral/02\_Design/02\_工具类帮助说明*目录下的文档《Cache的使用说明.docx》）

1. 在需要使用缓存的地方，调用相应的API来控制缓存，这个将会是侵入到代码中的方式，需要开发人员来编写代码的同时就需要考虑缓存的具体使用地方和使用方式；好处是可以非常灵活、灵活粒度的来控制缓存，灵活、有针对性得来设计缓存的Key；
2. 在需要缓存的方法上添加相关的Attribute特性，这种方式的好处是对代码编写没有侵入性，写代码的时候不用关心缓存问题，可以在最后了统一考虑哪些方式需要加上缓存；不好的地方是，缓存使用比较死板，只能按照方法+指定的参数值的方式来设计缓存项，并只能在方法级别做缓存；

建议做法是：大部分情况（通用情况）下会采用2的方式，在一些特殊的地方，需要更灵活控制缓存的，则采用1的方式；

## （二十七）关于日志的处理：

我们把Service端的日志，分为业务日志和非业务日志两种

1. **业务日志**：主要是和业务数据紧密相关的，其目的主要是用来追溯业务数据变化情况的；有些业务日志还会通过UI展现给用户看的，对这种业务日志处理，我们会有3种模式：
2. 只写统一的系统业务日志，那么请调用上面所提到的在Common Domain的ICommonBizInteract接口里的CreateOperationLog方法即可；
3. 只写自己单独的Log，这种情况就各个Domain自己各自实现即可，比如Customer这边也会有针对设置Bad Customer的Log，这种Log就是Customer只会写入自己的对应日志表中；
4. 即会写入统一的系统业务日志，同时还会有自己单独的Log，比如SO Domain会有SOLog，所以即需要写入自己的对应日志表中（SO Domain自己实现即可），同时也需要调用在Common Domain的ICommonBizInteract接口里的CreateOperationLog方法来写入统一的系统业务日志；

那么现在我们的代码中，对于需要写业务日志的地方依然延续之前IPP3中所用的业务日志处理模式，比如某个地方的代码，在IPP3里用的第3种的模式，那么在ECCentral里也继续保持使用第3种的模式（即自己实现自己的日志，不用再调用统一日志接口写统一业务日志）

1. **非业务日志**：主要是用来跟踪程序运行状态的一些信息，比如记录异常（对于一般的异常框架都会统一处理和记录）、性能、调用频率、某些程序变量的值或者环境上下文等等信息，其目的是用来监控系统程序的运行状况或者调试跟踪一些代码问题的；对于非业务日志，请使用ECCentral.Service.Utility.Logger这个静态类的相关方法；

## （二十八）关于权限控制的处理：

因为我们是Portal 🡪 Service的架构，所以对于功能权限的控制，我们会涉及到Portal端以及Service端分别的权限控制:

1. Portal端的权限控制：

(1). 页面级的权限控制  
这个在Oversea的SliverlightFramework框架里已经处理好了，所以开发人员Coding时不用再关心；

(2). 控件级的权限控制  
基本做法延续IPP3里的方式，在需要进行权限控制的地方，通过代码来控制相应控件的状态，具体做法是：  
a). 首先在ECCentral.Portal.Basic工程下的AuthKeyConst.cs文件中，在静态类AuthKeyConst里定义相关功能权限的Key值，如：  
public readonly static AuthKey Customer\_AgentInfo\_Add = new AuthKey("NeweggAgentInfoAdd");  
b). 在需要判断权限的地方调用ECCentral.Portal.Basic.AuthMgr的HasFunctionPoint方法，如：  
bool hasRight = AuthMgr.HasFunctionPoint(AuthKeyConst.Customer\_AgentInfo\_Add);

1. Service端的权限控制：

对于Service的权限控制，可以统一由ECCentral.Service.Utility.WCF里的相关ServiceBehavior来做处理，所以不需要开发人员Coding时再关心了；  
**当前我们并没有真正去实现服务端每个Service方法的权限控制**，如果将来要做的话，具体做法如下：

(1). 做一个类来实现接口ECCentral.Service.Utility.WCF.IAuthorize，该接口里只定义了一个方法：  
bool Check(string userIdentity, string methodName, string urlTemplate, string url)  
参数userIdentity为用户唯一标示（UserSysNo的字符串形式）；  
参数methodName为所调用的Service的方法名（如：GetCompanysByUser）；  
参数urlTemplate则是Restful Service的方法上所标记的UriTemplate（如：/Company/{userSysNo}）；  
参数url则是该次请求的url路径（如/CommonService/Company/2）；  
在该方法的实现里，可以去读取相关的用户权限配置（这个方法权限配置的整理才会是主要的工作量所在！！！这也是当前还暂时不做实现的原因），然后根据传入的参数来进行用户访问服务方法的权限控制；

(2). 在WCF Restful Service的配置文件里（文件默认是放在WebHost根目录下的Configuration/RestService.config），在根节点serviceList上添加一个属性authorizerType，将该属性的值设为第(1)步里所实现的类的AssemblyQualifiedName；

完成了这两步就可以为Service端添加对于每个Service方法的权限控制了；

## （二十九）关于业务代码中需要发送邮件（包括给外部消费者和内部工作人员的）：

请统一使用ECCentral.Service.Utility中的EmailHelper所提供的相关发送方法；关于邮件模板发送邮件的具体说明，请参看VSTS中*$/ECCentral/02\_Design/02\_工具类帮助说明*目录下的《邮件配置及邮件发送使用说明.docx》文档；

对于发送邮件时，多语言的使用指导说明请参看VSTS中*$/ECCentral/02\_Design/01\_设计与开发指南*目录下的《附6.EC-Central中针对多语言的开发指导说明.docx》文档中的第四点；

## （三十）关于业务代码中需要发送短信：

请统一使用ECCentral.Service.IBizInteract中Common Domain所提供的ICommonBizInteract接口中的SendSMS方法；

## （三十一）关于IPP3中遗留下来的一个SysConfig：

在IPP3里，这是一个Key-Value的配置功能，该配置是存储在DB里的，其中的配置是可以和前台Website共享，但不支持多语言，在IPP3里已经有大量的配置信息存储在该处了；所以在ECCentral里也需要继续使用该配置，各个Domain读取该配置的方式统一为：调用ECCentral.Service.IBizInteract中Common Domain所提供的ICommonBizInteract接口中的方法： string GetSystemConfigurationValue(string key, string companyCode)

## （三十二）导出Excel的处理：

对于导出数据为Excel的需求，有提供了相关的类方法来处理，具体可参见*$/ECCentral/02\_Design/02\_工具类帮助说明*下的文件《将数据导出Excel文件功能的实现的说明.docx》；

另外需要强调注意的一点**：在导出全部时，请还是要给查询条件Filter的PageSize设置一个不能太大的值（现在请统一都设置为一个常量**：**ECCentral.Portal.Basic.ConstValue.MaxRowCountLimit**），以防止查询拖垮DB Server；

## （三十三）SP（存储过程）的重构标准：

如果SP中有跨Domain的数据表/视图的访问或操作或跨Domain的SP调用，那么这个SP必须拆掉，目的是要实现跨Domain的数据隔离，只能通过IBizInteract来实现业务部分的跨Domain数据访问操作；

而对于一个SP是否需要重构到C#代码里，当前阶段暂不做强行要求，各个开发人员可自行决定是否将SP逻辑抽到C#里来，建议是尽量把业务逻辑从SP抽到C#中，提高业务内聚性；

## （三十四）Portal端UI上Oversea提供的DataGrid组件需要保证其GridID是唯一的：

Portal端UI上Oversea提供的DataGrid组件需要保证其GridID属性在整个系统里是唯一的，建议用Guid来做GridID，每次生成一个新的Guid（可以在SQLServer里用SELECT NEWID()生成或用VS.net的Guid生成工具）

## （三十五）上传文件并进行处理的统一做法：

使用框架提供的统一方法，具体见*$/ECCentral/02\_Design/02\_工具类帮助说明*下的文档：《上传组件FileUplodaClient以及上传控件FileUploader.docx》；

## （三十六）下载文件的统一做法：

1. 如果需要下载的文件是固定的，比如导入数据的模板文件等，那么直接把需要被下载的文件放到Service端的WebHost的Download目录下所属的Domain目录里，然后在Portal端调用ECCentral.Portal.Basic.Utilities.UtilityHelper的public static void OpenWebPage(string uri) 方法即可打开浏览器进行下载了；
2. 如果下载的文件的路径是动态获取的（比如要读取数据库或者临时生成的等），那么需要定义一个Restful Service，用来先把动态获取到的文件url相对路径返回给Portal端，然后Portal端再使用ECCentral.Portal.Basic.Utilities.UtilityHelper的public static void OpenWebPage(string uri) 方法即可打开浏览器进行下载了；

## （三十七）弹出浏览器，并打开一个url地址的网页的做法：

1. 如果只是采用Get方式来访问url页面，那么直接调用ECCentral.Portal.Basic.Utilities.UtilityHelper的public static void OpenWebPage(string uri) 方法即可；
2. 如果是要采用Post方式来访问打开url页面，那么直接调用ECCentral.Portal.Basic.Utilities.UtilityHelper的public static void OpenWebPageByPost(string uri, Dictionary<string, string> postData) 方法即可；

## （三十八）在Portal端预览Html的做法：

调用ECCentral.Portal.Basic.Utilities.HtmlViewHelper的public static void ViewHtmlInBrowser相关方法即可（需要先构建好要预览的html的字符串）；

## （三十九）在ECCentral中对于打印的统一做法：

1. 在Service端实现接口ECCentral.Service.Utility.IPrintDataBuild，并在相应配置文件中进行配置，并给一个唯一的名称；
2. 在Portal端使用ECCentral.Service.Utility中的HtmlViewHelper所提供的public static void WebPrintPreview相关发送方法；

关于打印模板和打印接口的具体相关说明，请参看VSTS中*$/ECCentral/02\_Design/02\_工具类帮助说明*目录下的《统一打印方案说明.docx》文档；

对于打印时，打印模板的多语言的使用指导说明请参看VSTS中*$/ECCentral/02\_Design/01\_设计与开发指南*目录下的《附6.EC-Central中针对多语言的开发指导说明.docx》文档中的第五点；

## （四十）关于邮件模板和打印模板的模板规则：

参看VSTS中*$/ECCentral/02\_Design/02\_工具类帮助说明*目录下的《邮件模板及打印模板的统一规则.docx》

## （四十一）在Portal端和Service端使用多线程的异步线程时，需要注意在异步线程里手工设置一下线程当前语言，否则异步线程的线程语言并不一定和主线程语言一致（默认是和Windows控制面板中的区域设置一致）：

在.net 4.5以后，可以通过设置CultureInfo类的DefaultThreadCurrentCulture和DefaultThreadCurrentUICulture两个静态属性来解决这个问题，但.net 4.0及以前版本，还是只能手工为每个异步线程设置线程的当前语言

# 三、工具类介绍

## （一）服务端工具类

### 1. 对象工厂-ObjectFactory



### 2. 服务端异常信息多语言工具类-ResouceHelper



### 3. 服务端业务数据配置读取工具类-CodeNamePairManager



### 4. 高效反射工具-FastInvoke

### 5. 序列化反序列化工具-SerializationUtility

### 6. 其他工具-StringUtility、CommonUtility

### 7. 数据映射工具-DataMapper

直接通过SQL字段名以及Parameter名来完成和数据实体属性的映射定义；具体映射规则和第8点的DataAccess中的映射规则一致（在DataAccess内部就是用的DataMapper来做的映射）；

### 8. SQLServer持久化工具-ECCentral.Service.Utility.DataAccess



### 9. ECCentral.Service.Utility.WCF中的IConvert和QueryResult

### 10. RetryHelper

### 11. 日志工具Log

### 12. 异常处理工具ExceptionHelper

### 13. 应用配置读取工具AppSettingManager



### 14. 关于缓存Cache的使用



### 15. 根据枚举值获得其显示文本-EnumHelper

### 16. 加/解密工具类-CryptoManager

//加密

string rst = CryptoManager.GetCrypto(CryptoAlgorithm.DES).Encrypt(password);  
//解密

string password = CryptoManager.GetCrypto(CryptoAlgorithm.DES).Decrypt(rst);

CryptoAlgorithm为枚举，用来指定加/解密方式，现在支持DES, RC2, Rijndael, TripleDES, RSA, DSA, MD5, SHA1这些加密方式（其中MD5, SHA1只支持加密，不支持解密）；

### 17. 邮件发送工具类EmailHelper

参加文档《邮件配置及邮件发送使用说明.docx》

### 18. 上传文件管理工具类FileUploadManager

参加文档《上传组件FileUplodaClient以及上传控件FileUploader.docx》中第二大点的第3点；

## （二）UI Portal端工具类

### 1. Restful Service调用工具类-RestClient



### 2. 根据Entity自动生成ViewModel的工具（一个外部的exe工具）



### 3. 高效反射工具- FastInvoke

### 4. 数据实体与ViewModel的数据转换工具-EntityConverter



### 5. dynamic对象与数据实体转换工具-DynamicConverter



### 6. 枚举值转换为显示文本的工具-EnumConverter



### 7. 获取服务端业务配置数据的工具-CodeNamePairHelper



### 8. UI控件数据合法性验证工具-ValidationHelper

### 9. 其它工具-UtilityHelper

### 10. 系列格式转换工具-Converters



### 11. Service端的应用配置读取工具AppSettingHelper



### 12. BooleanConverter



### 13. 公共UI控件-Components

参见VSTS中$\EC\_Products\ECCentral\02\_Design\03\_公共信息与数据\公共控件列表.xlsx

# 四、开发相关的Tips

1. 每个Domain会有自己的客户端UI，既可以访问自己Domain的Service所提供的Restful Service，同时也可以访问其他Domain的Service。但为了避免这种交叉引用引起混乱，所以这里给出调用的一个指导原则：

* 如果自己Domain和其他Domain都有能够符合UI调用要求的Restful Service，那么优先考虑调用本Domain的Restful Service；
* 如果所需要调用其他Domain的服务，是在一系列的服务方法行为中的一个步骤，并且这个步骤的前后不会有UI行为中断，那么建议将这些服务行为步骤调用的逻辑封装到自己Domain的Service端，包装一个更大力度的Service服务方法，那么对其他Domain服务的交互调用也就直接由自己Domain的Service端来负责了，此时UI客户端就只用和自己Domain的Service端交互了； （这样也可以减少HTTP请求和网络传输，提高性能，同时也更好的将业务逻辑封装到Service端，而不是分散在Client UI上了）
* 如果所需要调用其他Domain的服务是一个单独动作，其前后需要在UI上交互（比如获取的数据需要先在页面上显示等用户点击按钮确认等），那么该调用则可以直接调用其他Domain已有的Restful Service。

1. 对于在UI界面上，需要同时调用多个Restful Service的时候，需要考虑这多个调用之间的相互关系：
   1. 如果这多个调用之间没有任何关系和制约，那么可以全部异步方式调用即可；

ServiceA(\*\*,\*\*,(obj,args)=>{ …….});

ServiceB(\*\*,\*\*,(obj,args)=>{ …….});

ServiceC(\*\*,\*\*,(obj,args)=>{ …….});

* 1. 如果这多个调用之间会有先后顺序的依赖，比如需要第一个service调用的返回值，作为第二个service调用的入参，那么就只能在前一个异步callback中再发起后一个service的异步调用请求，从而达到控制先后顺序的目的；

ServiceA(\*\*,\*\*,(obj1,args1)=>{

ServiceB(\*\*,\*\*,(obj2,args2)=>{

ServiceC(\*\*,\*\*,(obj3,args3)=>{ ……. });

});

});

* 1. 如果多个调用中，可能有一个service的调用，需要依赖多个service的调用结果，而这多个service调用之间则没有先后顺序需求可以并发，那么此时可以采用计数器方式：



1. 在非业务查询的代码中，非业务查询的Service端的代码写法基本都是：  
   QueryResult result = new QueryResult();   
   int dataCount = 0;   
   result.Data = ObjectFactory<ISOQueryDA>.Instance.Query(filter, out dataCount);   
   result.TotalCount = dataCount;   
   return result;  
   针对这种相同使用模式的非业务查询的调用，可以采用下面的简化：
2. 设计一个委托：

delegate DataTable NoBizQueryListHanding<T>(T filter, out int dataCount) where T:class, new();

1. 定义一个通用方法：  
   public QueryResult QueryList<T>(T filter, NoBizQueryListHanding<T> handing)   
   {   
       QueryResult result = new QueryResult();   
       int dataCount = 0;   
       result.Data = handing(filter, out dataCount);   
       result.TotalCount = dataCount;   
       return result;   
   }
2. 具体使用就只需一句代码：  
   return QueryList<SOInvoiceChangeLogQueryFilter>(filter, ObjectFactory<ISOQueryDA>.Instance.InvoiceChangeLogQuery);