**EC-Central项目**

**整体技术架构设计原则和思路**

Benny.T.Yang

2012-3-30

我们在“如果可以设计简单，我们一定简单设计”这个大的设计指导方向上，我们会有几个大原则，也是我们整个系统设计的基础：

1. **Domain之间的切割与隔离**

(1). 业务切割

具体做法：各个Domain之间的交互调用，都通过接口来做

(2). 业务相关操作的数据切割

具体做法：把数据库表、存储过程、Function都归属到某个Domain里去，每个Domain在业务相关操作的数据持久化时只能访问自己Domain的数据库表和存储过程（无论读或写）；对于其他Domain的数据需要做业务相关读写时，都必须通过Domain间的业务交互接口来实现。

* 对使用到的表的切割：根据需要使用的表的划分归属，来决定是直接访问，还是通过其他Domain提供的交互来访问；
* 对使用到的存储过程、Function的切割：
  + 首先根据该存储过程、Function的划分归属，来决定是直接访问，还是通过其他Domain提供的交互来访问；
  + 如果是自己Domain的存储过程或Function，则需要再检查该存储过程里，是否有访问了归属为其他Domain的表、视图、存储过程或Function，如果有则需要把对其他Domain的表、视图、存储过程或Function的部分从存储过程中抽到C#代码里，来通过调用其他Domain提供的交互接口来实现（这里很有可能会需要对这个存储过程进行拆分）；
* 对使用到View的切割：根据View中有用到的表的归属来识别，如果View中只要有所用到的表是其他Domain的，那么就不能直接访问该View了，而需要通过调用其他Domain提供的交互接口来实现；

在系统中的体现：我们会定义公共的业务交互接口IBizInteract

1. **Domain内部的业务操作和非业务查询操作的隔离**

我们把Domain内部的操作分为2种：

* 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

针对这2种操作，我们会进行隔离，并且使用完全不同的处理方式；对于业务相关的操作我们会严格按照上面提到的业务切割和数据切割的原则来做；而对于业务无关的查询，我们则不作任何数据访问限制，可以允许跨Domain的查询其他Domain的表或视图，比如可以在SQL里直接做JOIN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

1. **Domain内部业务的内聚**

在Domain内部把相关业务内聚在一起，提高复用性，也同时避免一个需求点变化引出多个代码修改点的情况；对于后期维护代码的人员，能更容易从代码设计上看懂整个业务的设计；

具体做法：

* 从整体来分析业务，抽象出业务领域模型，整理出模型的数据与行为，梳理清模型间的关系，在系统里反应为我们的BizEntity[数据]和BizProcessor[行为]（因为一些实际情况我们采用了反模式，强行把数据与行为拆开了）；
* 逐步把散落在数据库中的业务逻辑和操作，都尽量整合到我们的业务模型上来；
* 逐步把散落在JOB里的业务逻辑和操作，都整合到我们的业务模型上来，让JOB只是一个单纯的触发器而已，真正的业务逻辑都通过调用EC Central的Restful Service来做；

1. **数据持久化的切割**

让业务的实现不以数据库的设计而受限，同时让数据库也不再受制于业务代码，可以更灵活的让DBA从更专业的角度去进行优化，甚至切换数据存储方案；

具体做法：

* 在设计业务逻辑代码时，不再以DB为中心，而以本身的业务领域模型为中心，在需要做持久化的地方，将领域模型的数据字段和DB的数据字段做Mapping，所以DB结构的变化，只需要调整Mapping关系即可，上层的业务模型和交互逻辑不用受影响；
* 对数据持久化操作抽象出数据持久化接口，上层建筑只依赖于持久化接口，而不用知道具体的持久化方案，这样为切换持久化方案提供了可能（比如将来切换NoSQL的代价和成本就更小了）；

1. **积木式的搭建系统 ----通过MEF框架来组装各个层和对象**

可以灵活替换或升级每一个模型对象，为它重新定义新的数据或者处理逻辑；

具体做法：

* 对于public类型的public方法，都定义为virtual的，方便后期做override替换；

（之前有人提到2个问题：（1）虚方法带来的性能问题 --- 其实虚方法调用速度比接口方法、委托都要快，在.net平台下是仅次于非虚方法调用；写一个什么都不做的虚方法，调用10000次的执行时间在0.1毫秒多点，那么即使是非虚方法的调用时间为0，那么10000次调用的性能差距也就在0.1毫秒多点了，这个性能损耗我们可以接受，我们在DB或者http传输或者业务算法上多花点心思得到的性能提升远远超过这里的性能损失；（2）关于方法的粒度，这是个业务分析和代码设计的问题，涉及到本身一个方法该如何来切割，这应该和业务相关的，这里建议说没有必要为了这个virtual而且刻意去考虑或调整方法的粒度，就和没有virtual的时候一样来定义方法好了，因为即使没有virtual我们也要考虑避免出现几百上千行的难以维护的大方法）

* 在Biz相关的层次里，所有的对象的实例化都通过我们统一的工厂来做，而不要直接用new方法（这里也做过性能比较了，10000次工厂创建和直接实例化的性能差距在1毫秒以内）

在后期，发生变化时：

* 如果业务逻辑发生变化，我们可以识别出需要override的行为的对象，然后从它们来派生新的子类，并复写新的业务逻辑；
* 如果数据字段需要新增，那么我们可以从原有BizEntity上继承出新的实体，并添加新的字段，然后在对应的override的新逻辑出使用新的BizEntity；

最后通过配置来升级相应的接口或基类的具体实现类，同时可以支持一个接口或基类的多版本实现的并行，只需切换配置就可以切换版本

以上5点原则都是对系统做横向与纵向的切割，通过功能层次做横向切割，通过领域划分做纵向切割，以达到内聚和解耦，可以灵活替换升级不同的粒度。