这次想讨论的话题是有关领域驱动设计，和领域驱动设计中使用贫血、失血or充血模型的。在这之前我想讨论下当前很多应用的问题，想起这个话题的起因是因为我在InfoQ上面看到这样一篇文章《[Spring Web应用的最大瑕疵](http://www.infoq.com/cn/news/2013/11/spring-web-flaw)》，不得不说，这样的标题相当吸引人(′·ω·`)。内容和主要观点大概是这样的，现在大部分应用Spring框架的Java Web应用都相当关注[单一职责原则](http://en.wikipedia.org/wiki/Single_responsibility_principle)和[关注分离原则](http://en.wikipedia.org/wiki/Separation_of_concerns)，但是在此之上却诞生了一些不太好的反模式和设计原则，比如：

* 领域模型对象只是用来存储应用的数据。(领域模型使用了贫血模型这种反模式)
* 业务逻辑位于服务层中，管理域对象的数据。
* 在服务层中，应用的每个实体对应一个服务类。

这类设计原则的应用非常广泛，我现在所在的Java Web项目就是使用这样的设计原则进行架构设计的，基本都是常见的三层或多层架构，他们大概是什么样的呢？

1. Web层（俗称展现层吧，Presentation Layer)：接收用户输入，将数据传至服务层；
2. 服务层(Service Layer，可以叫Business Logic Layer)：事务边界，处理业务逻辑、权限管理与授权，并与存储层通信；
3. 存储层（Data access layer）：与数据库进行通信，对数据进行持久化。

但是发现什么没有？问题出在了服务层，他承受了太多的职责，像事务管理、业务逻辑、权限检查等等，这违反了单一职责原则和关注分离原则，并且产生了大量的依赖和循环依赖。当业务复杂度上升时，服务层所包含的代码将会非常庞大和复杂，直接导致了测试成本的上升。   
我这里正好有个例子，在现在的项目中，负责处理保险业务单的核心类中，包含了4000多行代码，它与数据库中某一关键表相关联，引用（注入）了十几个DAO。在数十个各类方法中，可以处理保单、再保、理赔等等各种不同的业务，同时它还深度依赖于hibernate，不但使用了ORM方法处理数据，甚至还直接用了HQL来获取数据。因为有众多其他服务类与他进行循环引用，项目后期这个庞然大物已经没有人敢轻易改动了，因为谁也不知道他到底都能做什么，重构更是不可能的事。

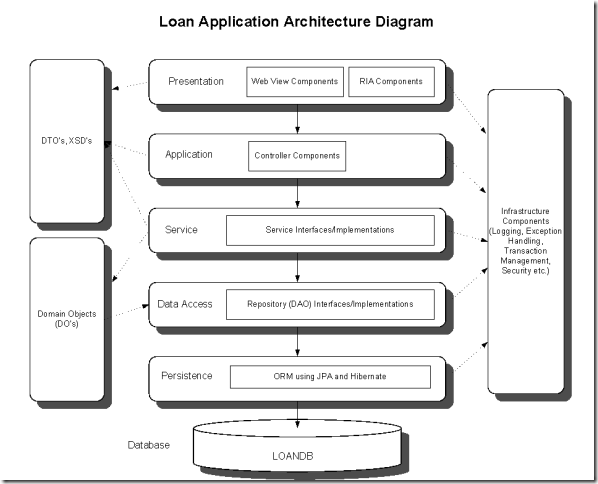
说了些服务层的坏话，那应该怎么改进呢？

* 首先，我们需要将业务逻辑从服务层移动到领域模型中，这样的好处是，服务层可以只负责应用逻辑（如数据有效性验证、授权检查、开始结束事务等），领域模型可以专门负责其相关的业务逻辑。还是以之前的保单系统来距离，架构设计时完全可以针对保单、再保、理赔等多个领域模型进行建模，相关的业务可以分别放到不同的领域模型中，一些很有可能重复的业务代码都会被集中到一处，从而降低了复制-粘贴的可能性。
* 其次，将服务类变得更小，使之只负责单一的职责。文章中有个例子，例如用户账户的CRUD和其他操作，就可以将其放到两个不同的服务类中，一个负责账户的CRUD操作，另外一个负责与用户账户相关的其他操作。

这样就能使服务类变得小巧、松散、可测试了，同时还能降低其他人理解与重用的成本。

接下来的问题就是，在实际的项目中，怎样实践这些设计原则呢？   
这里有一篇《[领域驱动设计和开发实践](http://www.infoq.com/cn/articles/ddd-in-practice)》非常值得一看，他所推崇的分层结构和上文所述类似，甚至提出了一些更细节的规则：

* 服务层需要包含应用逻辑、用户会话的管理，但不能包含领域逻辑、业务逻辑和数据访问逻辑；
* 领域层（领域对象）应该包含业务逻辑，可以处理与业务相关的会话状态.但作为商业应用的核心，应该具有良好的可移植性，不能对特定框架（如Struts、Hibernate、EJB等）产生依赖



说到这里，终于到了讨论的正题——贫血、失血和充血模型。什么是贫血失血充血模型呢？简单来说

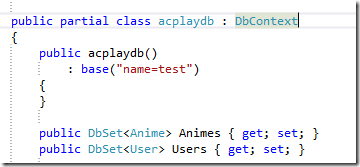
* 失血模型：模型仅仅包含数据的定义和getter/setter方法，业务逻辑和应用逻辑都放到服务层中。这种类在java中叫POJO，在.NET中叫POCO。
* 贫血模型：贫血模型中包含了一些业务逻辑，但不包含依赖持久层的业务逻辑。这部分依赖于持久层的业务逻辑将会放到服务层中。可以看出，贫血模型中的领域对象是不依赖于持久层的。
* 充血模型：充血模型中包含了所有的业务逻辑，包括依赖于持久层的业务逻辑。所以，使用充血模型的领域层是依赖于持久层，简单表示就是UI层->服务层->领域层<->持久层
* 胀血模型：胀血模型就是把和业务逻辑不想关的其他应用逻辑（如授权、事务等）都放到领域模型中。我感觉胀血模型反而是另外一种的失血模型，因为服务层消失了，领域层干了服务层的事，到头来还是什么都没变。

可以看出来，失血模型和胀血模型都是不可取的，现在的问题是，贫血模型和充血模型哪个更加好一些。很久很久以前，人们针对这个问题进行了旷日持久的争论，最后仍然没有什么结果。这里有一些帖子可供回味：   
[贫血，充血模型的解释以及一些经验](http://www.evernote.com/shard/s315/sh/a389cb28-9910-443b-8e30-4d1796dafe9d/2838ff81c0a65b966acda644ebb2300d)   
[总结一下最近关于domain object以及相关的讨论](http://www.iteye.com/topic/11712)

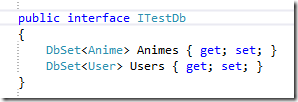
双方争论的焦点主要在我上面加粗的两句话上，就是领域模型是否要依赖持久层，因为依赖持久层就意味着单元测试的展开要更加困难（无法脱离框架进行测试，原文的讨论中这里专指Hibernate），领域层就更难独立，将来也更难从应用程序中剥离出来，当然好处是业务逻辑不必混放在不同的层中，使得单一职责性体现的更好。而支持者（充血模型）认为，只要将持久层抽象出来，即可减少测试的困难性，同时适用充血模型毕竟带来了不少开发上的便利性，除了依赖持久层这一点，拥有更多好处的充血模型仍然值得选择。最后，谁也没能说服谁，关于贫血模型和充血模型的选择，更多的要靠具体的业务场景来决定，并不能说哪一种更比哪一种好。设计模式这种东西不是向来都没有什么定论么。

我个人则倾向使用充血模型，因为充血模型更加像一个设计完善的系统架构，好在计算机世界里有很多的IOC和DI框架，唯一的缺陷依赖持久层可以通过各种变通的方法绕过，随着技术的进步，一些缺陷也会被慢慢解决。我的思路是这样的：先将持久层抽象为接口，然后通过服务层将持久层注入到领域模型中，这样领域模型仅仅会依赖于持久层的接口。而这个接口，可以利用现有框架的技术进行抽象。举例来说，Java版Hibernate我了解不多，就以.NET的Entity Framework来说吧：

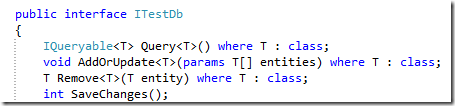
现在有这么一个DbContext，大家都懂得，DbContext和DbSet是非常不好Mock的两个类（我就是嫌麻烦而已，高手请无视），里面有两个表，一个叫Animes另一个叫Users



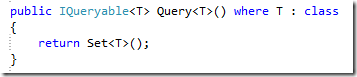
怎样设计接口才能使它既容易使用又可以方便测试呢？直接提取一个接口？DbSet不容易Mock的问题还是没有解决吧。



好在我们有LINQ和IQueryable<T>，随便改造一下，接口就变成了这样：



请注意Query<T>()方法，这个方法返回一个IQueryable<T>的对象，而实现了IQueryable的对象是支持LINQ操作的，也就是说，我们可以仍然可以将搜索的Expression交给真正的DbContext来做，而这个DbContext只需要简单一句话：



查询时从 from a in db.Anime.AsQueryable() 改成 from a in db.Query<Anime>()，一切都解决了。当你在单元测试中想要返回一个假的数据源的时候，直接让FakeDb.Query<T>()方法返回一个拥有假数据的List<T>.AsQueryable()就可以了。这样就实现了领域层和持久层的解耦，毕竟IQueryable是通用的嘛。