面向对象方法中的数据库设计

coffeewoo 2010-02-05 15:21:00 22416 已收藏 3

文章标签： 数据库 框架 hibernate delphi 扩展 正则表达式

版权

最近收到不少网友关于在面向对象的分析设计中如何进行数据库设计的疑问。在大象-thinking in uml一书里，我详细讲述了面向对象从需求到设计的整个过程，但确实对数据库设计着墨甚少。因此写这篇文章对这个问题详细说明，在第二版里应当也会加上这一部分。

先贴出一位网友的疑问以及我的回复，作为这篇文章的引子：

网友fdshxp问道：

在软件开发时要进行数据库设计，现在通常的做法是需求分析，做数据流图，画ER图，这些显然是面向过程的东西，而在面向对象分析设计时，只是提数据库设计的内容，具体怎样做？虽然可以将数据库操作封装起来，但要设计的数据库表不能通过拍脑袋的方法获得，总不能再去画一遍数据流图吧！

我回答：

 在面向对象中，是没有数据流这一说法的。业务的完成是由对象及消息来完成的，只有“对象流”，没有数据流。

    只是在现实中，绝大部分的对象持久化是用关系数据库实现的，我们还没有在性能上和查询上可以顶替关系数据库的对象数据库。设计数据库表的目的是不考虑所谓“流”的，考虑的是如何把对象高效的持久化。可以说，数据库设计和之前的面向对象设计是两个领域的问题，面向对象设计解决业务执行逻辑问题，数据库设计解决数据高效的问题（它根本不考虑流控制的概念），它们中间通过OR-mapping的机制结合起来。如果你对此一直有疑问，那说明你试图在设计数据库表时考虑通过数据库表设计表达业务逻辑问题，而不是考虑如何高效的持久化对象。

    假设，现在技术成熟到我们已经有性能不低于关系数据库的XML持久化机制和对象查询机制，任何对象都可以直接持久化而不需要OR-mapping，那么还需要设计数据库表么？

网友fdshxp继续问道：

谭老师，你好！很荣幸收到你的回信！

    你回信中说：面向对象设计解决业务执行逻辑问题，数据库设计解决数据高效的问题,说的真好。

    我的确是通过数据库表设计表达业务逻辑问题，因为在看许多数据库方面的书籍的时候，书上都是这么说的，例如人大的王珊教授的《数据库系统概论》中说：数据库设计是指对于一个给定的应用，构造优化的数据库逻辑模式和物理结构，并据此建立数据库及其应用系统，使之能够有效地存储和管理数据，满足各种用户的应用需求，包括信息管理要求和数据操作要求。我的理解就像你所说的，通过数据库表设计表达业务逻辑问题，比如可以设置约束限制数据，通过触发器对相应的业务进行自动化处理。

    我在软件开发方面的学习可谓是一波三折，一开始，学习了C/C++，什么也不能做，后来接触VC++，由于其体系庞大，也没有做出什么来，转到VB、Delphi上，很轻松，做一个小的项目，看到VB和Delphi由于其和当今技术的差别，现转到.NET上来。（始终没敢转到JAVA上）

    正像你大象书中说的那样，我用面向对象纯粹是为了改进开发效率，通过封装、继承、多态等手段重构代码。进行数据库主要是为了存储业务数据，为了更好、更全面地管理业务，我按照数据库书上讲的，进行了面向过程的分析，画数据流图和数据字典，以求得对业务数据全方位的认知，接下来，整合业务数据画ER图，将其转化为数据库表。

    说实话，Hibernate的orm映射，我没用过。在进行面向对象分析和设计时，如果不用ORM，如何将数据库设计融入其中，我现在还是一头雾水。现在许多书上的做法是：先建立数据库表，然后将其封装，设计类。这种做法看起来很好。但是表是怎样推出来的，究竟设计几个表合适，还是不知道。

    通过谭老师你的书，我认识到设计的重要性，认识到编程的可推导性，从分析设计中推导出编码的要素。但是现在大部分的系统都需要数据库的支持，用数据库来存放业务数据。

    面向对象设计中，如何设计数据库的问题，请谭老师在不忙的情况下，给我一个设计程式。

    我想是时候谈谈面向对象数据库设计的一些想法了，在回答这位网友的同时更多的讲讲面向对象方法里如何设计数据库。

    首先想说的是面向过程的的数据流分析方法不是不正确，只是它不符合对象分析方法。两者的出发点是不同的，就象向两个不同方向前进的队伍，是无法调合的。而现在很普遍的所谓面向对象设计时“先建立数据库表，然后将其封装，设计类”则是彻头彻尾的错误! 套上一个面向对象的马甲，干的是完全不面向对象的事情。面向过程方法下的表设计还有数据流为推导，而这种伪对象方法为了穿上面向对象的画皮而抛弃了数据流的马甲，却又不按照对象分析方法行事，就更不知道数据库表是如何推导出来的了。

    运用最广的Hibernate在实际中有太多的误用，OR-mapping被仅仅当成数据库物理表和对象之间的简单一一对应，其本质还是先设计数据库再设计类。再强调一次“面向对象设计解决业务执行逻辑问题，数据库设计解决数据高效的问题”，它们本质上是两个领域的设计，只是由OR-mapping来连接它们。要采用面向对象方法，首先要忘记数据库的存在，采用对象分析方法，先把对象分析和定义出来，保证业务执行逻辑能够被这些对象很好的完成。达到这一点后，再来考虑对象持久化的问题。依据数据库的三大范式以及性能要求来把对象持久化。注意！！这时我们设计数据库要解决的问题是“对象数据高效持久化”，而不是业务逻辑！它不是从需求中推导出来的! 例如面向过程的设计中，一张申请表很可能被设计成一张物理表；而面向对象设计中，很可能没有申请表这么一张物理表，而只有“用户资料”、“申请流程”、“申请资质”等对象表，所谓的申请对象，是在运行期由这些对象聚合而成的。

    每个对象都有自己的属性和状态，我们需要把这个对象的属性和状态保存在数据库中，那么最理想最最简单的情况，就是一个对象对应一张物理表，而对象之间的关联关系（一对一，一对多，多对多）也可以简单的映射成数据库的主-外键关系。 但还有很多非数据库关系需要考虑，如：继承、聚合、依赖等。一张表如何继承自另一张表呢？关系数据库显然没有这样的定义，这就需要用OR-mapping来完成这种语义的转换。例如，当实例化一个子对象时，OR-mapping负责从代表了“父”对象的表中读出父对象属性并将其赋值给子对象，并且当父对象变化时，OR-mapping需要把这一变化反映到所有子对象实例（这只是一种OR-mapping方案，也有在所有子表里冗余存贮父对象属性来实现的）。再比如聚合对象，一个公司对象由公司基本信息以及一个部门List构成，那么在持久化这个对象时显然需要把它分成公司表和部门表（一对多关系），在业务逻辑执行过程中操作公司对象时它们始终是一体的对象，但当CUDR这个对象时OR-mapping要负责将对对象的操作转化为对两张表的操作。而依赖表示了两个对象之间相互依存的关系，当一个变化时另一个相应的要变化。这在数据库中可以由Insert/update/delete 引发的trigger来实现，但更好的做法显然是由OR-mapping来实现这种关系的管理。

    实际上我们所遇到的情况只会更加复杂，一个复杂的业务对象可能对应的数据库中的许多张表；一些简单的对象也可能只对应数据库中某张表的一部分。现在我们应该明白OR-mapping的作用了，它不是负责将数据表直接翻译成为对象那么简单，它负责的是将对象关系语义转化成数据关系语义。换言之，OR-mapping负责的是“数据”和“表现”的分离，数据如何存贮和查询是一回事（由三大范式和性能优化考虑决定），数据如何表现又是另一回事（由业务执行逻辑和高效面向对象设计决定）。如果一个or-mapping做得足够好，能完美支持对象关系和数据关系的转换的话，你就可以独立的更改对象和数据库，之后只需要重新配置一下mapping关系即可。

    一个典型的例子，在面向对象的设计中，业务逻辑和控制逻辑通常是分离的。比如一个定单对象，在业务执行逻辑上，除了业务数据，它还需要一些状态属性来标识流程控制进程；但是流程控制进程通常都不是业务数据的一部分，它只是系统的控制逻辑。在好的面向对象设计中，这种控制逻辑是可以分离出来用另一组对象来标识，再通过对象的聚合或者对象之间的依赖注入来将两者动态绑定的。在以数据流为基础的数据库设计中，通常的做法是将状态控制字段与业务字段设计在同一张表里的。其结果是控制逻辑与业务逻辑被静态绑定，这意味着两者都不能独立变化。只要查看一下现在的很多系统中，当流程变化时导致要更改业务表，或当业务数据变化要改流程，就说明该设计不是一个面向对象的设计，或者至少是一个糟糕的面向对象设计。真正好的面向对象设计会分离业务逻辑和控制逻辑，在运行过程中业务对象与流程控制对象是独立加载并在流程控制框架下动态绑定的。这意味着两者都获得了独立变更的能力。在此基础下持久化业务对象和流程控制对象的结果是必然会形成一组流程控制表和一组业务数据表，它们两者之间是没有静态依赖关系的，某个流程实例的控制状态只会存在于流程控制表而不会存在于业务表中。因此，流程控制与业务得以解耦而独立变更。

    如果将革命进行得更彻底一些，我们甚至可以仅仅将数据库视为保存数据的一种手段，而放弃数据库的约束，如主-外键关系。在笔者以前进行的一个项目中进行了这样的尝试，所有数据库表之间均没有主-外键关系，没有trigger，没有约束，每张表都是独立的，每张表都是直接对象的持久化的结果，数据库甚至不管理对象之间的关联关系，每张表仅由一个唯一的主键ID来标识对象实例。而对象之间的关系全部抽象出来用一组对象关系表来管理，一条关系表记录表示两个对象ID之前的一种关系，由一个对象关系管理框架来管理它们。对象关系管理框架管理对象之间的“关联”、“继承”、“依赖”等简单关系，同时经过扩展，这些关系可以扩展成为更复杂的对象关系，例如可以在关系当中加入时间因素，表示某两个对象在一定时间之内是“关联”的或“继承”的；也加入版本因素，表示某两个对象在某个版本当中是“关联”的；甚至可以加入条件因素，用一个正则表达式来表达在什么条件下两个对象产生“关联”关系。在这个管理框架下，对象理论上拥有无限的扩展能力，而这种能力却不依赖于数据库。一张数据库表的变化仅仅影响它对应的持久化的那个对象而已。我们完全可以在程序中动态的创造出对象关联（向关系管理框架中加入一个关系实例）从而动态的创造出一个全新的对象，我们也可以扩展关系管理框架中的关系而得到更加复杂的对象组合。

   但是彻底的革命也并不是完美的，这种与数据库关系彻底的决裂意味着我们同时放弃了数据库的高效，完全由程序来管理对象关系不但引入了一个复杂的框架，同时整体性能也大受影响！例如，一个拥有子对象的对象在采用数据库关系管理时，我们可以用一条SQL语句来加载这个对象；在采用对象关系管理框架以后，我们必须先得到一个对象，然后向关系管理框架咨询它所关联的对象ID，然后再加载它，这个过程必然产生多条SQL调用。CURD所有操作都需要额外的向关系管理框架咨询和操作，得到扩展能的同时牺牲了性能。但现实就是这样，人生不如意十之八九，得到一些总是会失去一些的。好在在性能要求不太高的场合，这个框架是相当有效的！以致于在项目过程中我们从未对数据库修改头疼过，因为我们的程序逻辑、显示逻辑等与数据库是无耦合的，我们使用的是一种称为ValueObject的 POJO来作为业务实体对象和显示对象；而这个ValueObject是由关系管理框架根据对象关系将持久对象（Persistence Object）动态组合出来的。等效于我们解耦了实体对象和实体对象的持久化结果，自然的，数据库的修改就变得轻松很多了。

    今天的文章里详细讨论了面向对象方法里数据库的设计方法。如果你是一个面向对象的革命者或愤青，那么你可以宣称面向对象不需要数据库设计（估计这是少数派）！如果你是一个面向过程的保守派，那么你可以宣称数据库设计是一切的核心（估计这是多数派）！然而我们还是现实一些，站在实用主义的角度，承认:

面向对象方法是非常行之有效的；数据库设计应当围绕着对象的高效持久化进行而不是以数据库设计为核心；

关系数据库的高效及方便不是对象数据库模式在短期内可以轻易达到的，我们不能因为倒脏水把婴儿也泼掉了；

最好的方法是根据实际项目对性能和扩展性的要求，在性能要求高的场合可以适当牺牲面向对象的特性来达到性能要求，在扩展性要求高的场合则可以适当牺牲数据库性能来满足扩展性。

————————————————

版权声明：本文为CSDN博主「coffeewoo」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/coffeewoo/article/details/5291582