基于 AOP 的业务规则应用框架的研究与实现①

Research and Implementation of Application Framework of Business Rules Based on AOP

张　瑜　王　华　(杭州电子科技大学 计算机学院 浙江 杭州 310018)

摘　要： 构建基于 AOP（AOP 是 Aspect Oriented Programming 的缩写，意思是面向方面编程）的业务规则应用框架。该框架引入连接方面来模块化规则引擎调用，从而实现规则引擎调用与核心业务逻辑的分离，通过面向方面编程织入技术最终把二者整合起来。这样业务规则和核心业务逻辑就能够相互独立地进行设计和实现，业务规则和规则引擎的变化不会影响到核心业务逻辑代码，大大提高业务策略调整的灵活性。

关键词： 面向方面编程 业务规则 核心业务逻辑 规则引擎 连接方面

业务规则原理从业务策略的本质入手来研究业务策略调整问题，它在概念层次上把业务策略与核心业务逻辑分离开来。在实际应用中如何实现业务规则成为实现业务策略灵活调整的关键。

1　问题的提出

根据业务规则组织(Business Rules Group)对业务规则的定义，业务规则是描述和约束业务的语句，用来刻画业务的结构或控制和影响业务的行为。业务规则包含一组条件和在此条件下执行的操作，它们表示业务规则应用程序中的一段业务逻辑。而业务逻辑是指完整展现应用程序处理的过程，它是在分析阶段对软件的应用领域进行的分析总结，它的存在不依赖于软件的存在，相反，它先于软件存在并限制了软件应有的行为。核心业务逻辑是指对应用中最主要的业务流程的抽象描述，可以通过业务规则和数据流表现出来。

早期，业务规则是与核心业务逻辑交织在一起的，它直接通过核心业务逻辑中的部分程序代码来实现，如图 1 所示。

这里业务规则掺杂在核心业务逻辑中的业务对象之间，同样的业务规则可能存在于多个核心业务逻辑中。因此，业务规则无法独立于核心业务逻辑而变化，

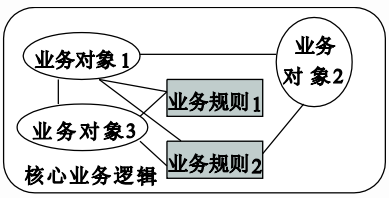

图 1　早期业务规则与核心业务逻辑交织示意图

而必须以修改核心业务逻辑代码的方式来实现。这不但增加了核心业务逻辑代码的不确定性，而且使得业务规则中的错误很容易扩散到核心业务逻辑代码中，给核心业务逻辑带来不可预知的问题。业务规则的重复性增加了修改代码的工作量，并且容易造成修改的不一致。

规则引擎的引入了实现业务规则与核心业务逻辑的分离，分离出的业务规则以规则引擎特定的形式表示出来，然后放入规则引擎中以备核心业务逻辑使用，如图 2 所示：

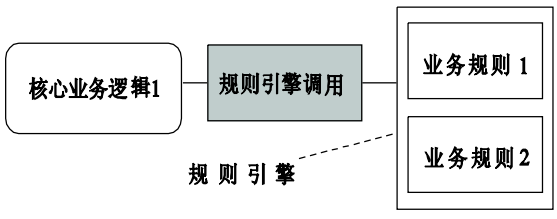

图 2　引入规则引擎后业务规则与核心业务逻辑的关系图

① 收稿时间:2008-10-29

分离出业务规则后，核心业务逻辑变得更为简洁、责任更加明确，但是业务规则与核心业务逻辑仍然是显式连接的，实现规则引擎调用的代码仍然交织在核心业务逻辑中。这样，当升级或更换规则引擎时，就必须修改核心业务逻辑代码。而且使用同一组业务规则的多个核心业务逻辑模块必然重复同样的规则引擎调用代码，这就使得规则引擎调用延续了业务规则的散播性，如图 3 所示：

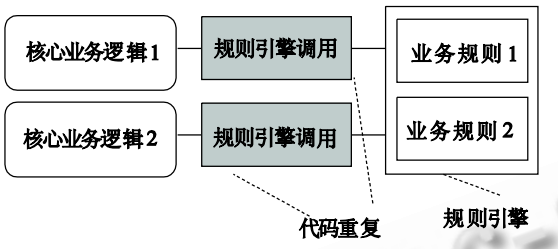

图 3　规则引擎调用的散播性

在当前的业务规则应用中，对核心业务逻辑代码的修改和规则引擎调用的散播性成为两大主要问题。本文正是从解决这两大问题入手，应用 AOP 思想和技术寻求一种新的业务规则应用框架。

2　解决问题的基本思想

业务规则应用演化中的一个重要推动力和出发点是关注点的分离，同样解决对核心业务逻辑代码的修改和规则引擎调用的散播性这两大主要问题的关键也是实现规则引擎调用与核心业务逻辑的分离。规则引擎调用与核心业务逻辑的交织是在源代码级别上的，同样的规则引擎调用代码可能存在于多个核心业务逻辑代码中，如图 4 所示：

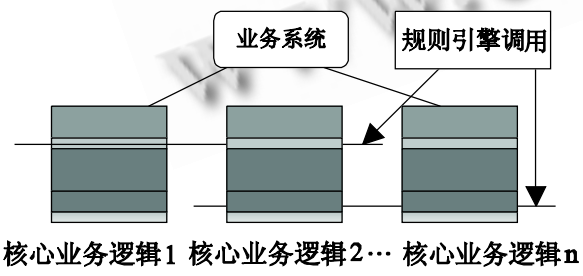

图 4　规则引擎调用与核心业务逻辑交织关系图

从图 4 中可以发现，规则引擎调用横切多个业务模块，属于横切关注点。这样实现规则引擎调用与核心业务逻辑的分离的关键就是实现这些规则引擎调用横切关注点的模块化。而 AOP 的引入正是着眼于横切关注点的模块化，它使得系统各个模块的职责更加清晰，各个模块能够相互独立地进行设计、实现以及维护。

AOP 是通过方面来实现横切关注点模块化的，本文把用来模块化规则引擎调用的方面称为连接方面。规则引擎调用和核心业务逻辑先分别用相互独立的模块来实现，然后再通过 AOP 的织入技术把二者有机地整合起来，这样就得到如图 5 所示的最终系统：

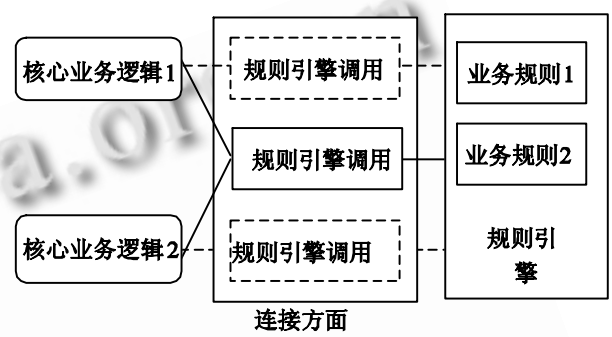

图 5　采用基于 AOP 的业务规则应用框架的系统示意图

在这里业务规则与核心业务逻辑是隐式连接的，对于核心业务逻辑来说业务规则的存在是透明的，业务规则的修改不会影响到核心业务逻辑代码，这就解决了对核心业务逻辑代码的修改问题。相同的规则引擎调用是统一在连接方面中实现的，直到编译阶段才和核心业务逻辑整合到一起，这就解决了规则引擎调用的散播性问题。所以，该业务规则应用框架有效地解决当前业务规则应用存在的两大问题，大大提高了业务规则应用的灵活性，进而有效地提高了业务策略调整的灵活性。

3　框架的实现

该框架主要由连接方面、规则引擎和规则文件三部分组成，其中连接方面是核心部分，也是最难实现的部分。连接方面负责确定核心业务逻辑中发生规则引擎调用的地方，并收集规则引擎执行规则评估时所需的信息。具体讲，连接方面通过定义专门的切入点来捕获核心业务逻辑中调用规则引擎的地方，在通知中使用切入点处获取的上下文信息构建规则引擎执行时的 Working Memory。这样就在核心业务逻辑中搭建起执行规则评估的环境，而且这一环境是动态织入的，不为核心业务逻辑所知，该环境的修改不会影响

到核心业务逻辑。连接方面在通知核心业务逻辑中特定连接点来完成规则引擎 Working Memory 的初始化后，就可以请求规则引擎对规则进行评估，如图 6 所示：

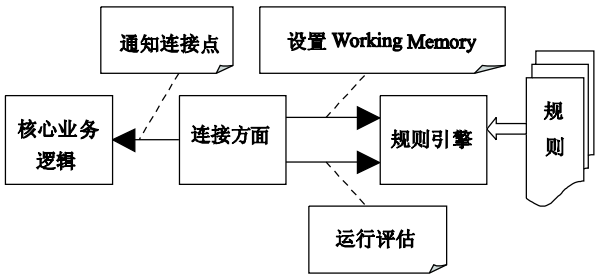

图 6　基于 AOP 的规则应用框架实现

具体的执行过程如序列图 7 所示：

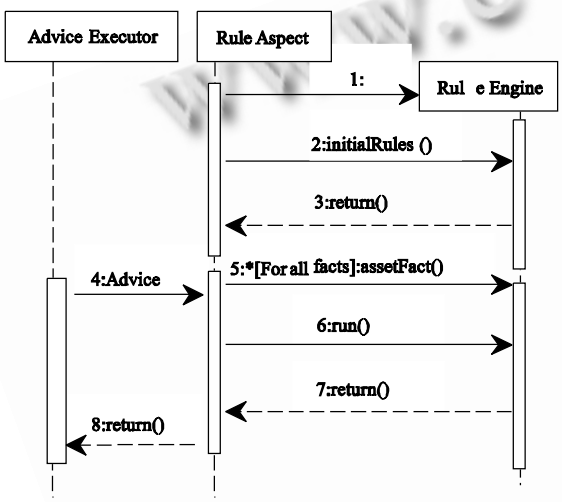

图 7　基于 AOP 的规则引擎与核心业务逻辑连接序列图

图 7 中 Advice Executor 是一个在织入过程中由 AspectJ 编译器产生的概念对象，织入过程把调用通知与业务对象中的相应操作关联起来。由该图可以看出业务对象组成的核心业务逻辑没有直接调用规则引擎，这一调用关系是通过 Advice Executor 以及 Rule Aspect 在织入过程实现的。

4　实现框架需要解决的主要问题

要想把该框架应用到实际业务系统中，需要解决规则引擎调用点的确认问题、规则执行时信息的获取问题以及连接方面的设计与实现，下面分别对这三个问题进行分析：

4.1 规则引擎调用点的确定问题

规则引擎调用通常发生在核心业务逻辑中某些定义好的执行点，特别是核心业务逻辑中某些行为被激发时或业务对象结构中某些操作被执行时，这些执行点称为规则引擎调用点，简称调用点。为了把规则引擎调用与核心业务逻辑有机地整合起来就必须准确暴露和定义核心业务逻辑中的这些调用点。

① 基本调用点 我们能够通过切入点来捕获调用点，由基本切入点所捕获的调用点称为基本调用点，当切入点捕获的情况发生时就可以调用规则引擎以执行规则评估。

② 静态上下文调用点 静态上下文调用是通过词汇结构切入点来捕获的，它把规则引擎调用限制在某个类、方面或方法代码范围内。

③ 动态上下文调用点 与静态上下文调用点不同，动态上下文调用是把规则引擎调用限定在代码运行时的某个范围，它是通过流程控制切入点 cflow (PointCut)或 cflowbelow (PointCut) 来捕获的。

4.2 规则执行时所需信息的获取问题

在规则引擎与核心业务逻辑分离后，规则执行时所需信息的获取称为一个突出的问题。这就要求连接方面不但要把规则引擎调用与核心业务逻辑连接起来，还应该捕获业务规则执行时所需的信息以构成规则引擎的 Working Memory，使得规则引擎能够进行规则评估。下面根据业务规则所需信息的类型分别进行说明：

① 环境信息 业务规则执行时可能需要系统的环境信息，这些信息往往是系统级别上的、具有全局性，例如时间日期等。不同的规则引擎对这些信息的获取有所不同：有些规则引擎支持规则直接获取这些信息，例如 Drools；而有些却不行，可以在连接方面中获取这些信息再传给规则引擎。

② 调用点可用的业务对象信息 许多业务规则需要特殊的业务对象，这就要求连接方面在切入点获取这些信息，然后再把它们传给规则引擎。可以通过参数切入点 args (Type) 或执行对象切入点 this (Type) 以及 target (Type) 来获取这些信息。

③ 调用点不可用的业务对象信息 有时候业务规则需要的信息在调用点已不存在，这就要求在这些信息消失之前先保存下来。可以通过方面关联(Aspect

Association)来实现，即把方面关联到一个对象或执行点上。通常在同一个虚拟机中方面只有一个实例，也就是说方面是无状态的。方面关联使得方面附着于某个对象或执行点上，这样方面就能够把该对象或执行点的信息保存下来。对象或执行点不同的值对应着不同的方面实例，方面也就有了状态值。

4.3 连接方面的设计与实现

在调用点确定好后就可以设计连接方面实现规则引擎调用与核心业务逻辑的整合。连接方面主要由三部分组成：调用点的定义、规则引擎的初始化、以及通知的定义，可以依据下面模板来设计连接方面：

```
public aspect ConnAspect [<association-specifier> (<Pointcut>)] {
        //定义规则引擎调用点
        pointcut connPoint() : call(* BusinessLogic.operate(..)) && within(BusinessLogic);
        // 规则引擎相关定义
        public ConnAspect (){
    //初始化规则引擎
   }
//连接规则引擎的通知
        after() returning(BuinessObject bo) : connPoint () {
        //启动规则引擎进行规则评估
        }
}
```

5 小结

本文分析了当前业务规则应用中存在的对核心业务逻辑代码的修改和规则引擎调用的散播性这两大问题，针对这两大问题提出基于 AOP 的业务规则应用框架，它的核心是用连接方面来模块化规则引擎调用使其从核心业务逻辑分离出来，最后再通过方面织入实现规则引擎调用与核心业务逻辑的低耦合整合。本文还分析了应用该框架时需要考虑的三个主要问题，并给出使用该框架的主要步骤。

参考文献

1 Willmor D, Embury SM. Testing the Implementation of Business Rules Using Intensional Database Tests. Proceedings of Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006, 2006,10:115－126.

2 Kamada, Aqueo. Business Rules and Services in the Context of Model Driven Architecture. Proceedings of Computational Science and Engineering Workshops, 2008.CSEWORKSHOPS'08, 2008:233－238.

3 Kamada A., Mendes M. Business rules in a service development and execution environment. Proceedings of Communications and Information Technologies, 2007,14:1366－1371.

4 Ross RG. IT systems perspective-The business rule approach. IEEE Computer, 2003,36(5):85－87.

5 Ali S, Soh B, Torabi T. A novel approach toward integration of rules into business processes using an agent-oriented framework. Industrial Informatics. IEEE Transactions, 2006,2(3):145－154.

6 Shepard, SJ. Policy-based networks: hype and hope. IT Professional, 2000,2(1):12－16.

7 Thomas ERL. Service-oriented Architecture：Concepts, Technology and Design. New York：Prentice Hall PTR, 2005.