摘要

现代信息系统已经被广泛使用于各行各业，每刻都产生着大量的数据。在这海量的数据背后蕴藏着丰富的有价值的信息，需要有相应的机制和系统来发现和提取。复杂事件处理机制（Complex Event Processing, CEP）是一种有效的办法。它从大量的数据中抽象出原始事件，并通过一定的事件操作符将不同的事件关联起来形成复合事件，表示新的意义，揭示数据之间隐含的信息。这些事件，特别是复合事件快速地传送到关注它的用户或者系统那里，以便他们进行决策，提高企业的响应能力，这就是事件驱动的思想。事件驱动体系结构上是对反应式系统的抽象，通过“推”模式的事件通信提供并发的反应式处理，特别适合松耦合通信和支持感知的应用，是解决SOA松耦合通信与协同处理的理想解决方案。

本文详细分析了事件驱动架构比之传统系统架构的优势，并介绍了可以实现此架构的复杂事件处理平台Apama及其MonitorScript语言。然后尝试着用它们设计和实现了金融交易清算系统T并给出了具体的介绍。最后，针对现有复杂事件处理平台缺乏可持久化能力的状况，提出了两种较通用的增强方案，实现了可恢复、可持久化的事件处理平台。

关键词： 事件驱动架构，复杂事件处理，Apama，交易清算

Abstract

Modern information systems have been widely used in all walks of life, every moment has produced a large number of data. In this mass of data is behind a wealth of valuable information, necessary to have appropriate mechanisms and systems to discover and extract. Complex Event Processing (CEP) is an effective approach. It is an abstract from a lot of data out of the original incident and the events through a certain operator associated with the different events together to form composite event, said the new meaning, between the data reveal the hidden information. These events, in particular, is a compound event and quickly sent to the attention of its users or the system there, so that they can make decisions, improve business responsiveness, which is event-driven ideas. Event-driven architecture is a reactive system of abstraction, through the "push" model of event communication to provide concurrent reactive processing, particularly suitable for loosely coupled communication and support for sensing applications to address communication and coordination of loosely coupled SOA vision processing solution.

This paper do the detailed analysis of the event-driven architecture than the traditional advantages of the system architecture, and describes the architecture can achieve this Apama complex event processing platform and its MonitorScript language. Then try to use them to design and implement a financial trading clearing system T and give a concrete description. Finally, existing complex event processing platform is lack of persistence ability, so we proposed two kinds of the common enhancements to achieve a resilient, sustainable-oriented event processing platform.

Keywords: Event Driven Architecture, Complex Event Processing, Apama, Trade Clearing

图目录

图 1.1 订单系统的事件交互 4
图 1.2 事件云集描述订单系统 5
图 1.3 传统数据处理方式中的订单发货 7
图 1.4 事件处理方式中的订单发货 8
图 2.1 事件驱动架构的分层结构 14
图 2.2 Apama 平台结构 16
图 2.3 Correlator 结构 18
图 2.4 Correlator 内部事件输入 / 输出队列 19
图 2.5 股票事件类型定义 20
图 2.6 monitor 的股票价格例子 21
图 2.7 monitor 组合事件联系的例子 22
图 3.1 当前的清算流程 27
图 3.2 系统 T 和抽象后的清算流程结构 28
图 3.3 系统 T 含未知可扩展的结构设计图 29
图 3.4 系统 T 最终的结构设计图 30
图 3.5 Correlator 内部各模块及事件处理流程 32
图 3.6 系统各模块启动流程 33
图 3.7 系统各模块关闭流程 34
图 3.8 BCService 内部各子模块启动关闭流程 37
图 3.9 SendMailService 与 Mail 的层次结构 38
图 3.10 载入系统 L 文件的事务过程 39
图 3.11 DBService 的结构 41
图 3.12 IAFStatusManager 管理 Adpater 状态的流程 42
图 3.13 FAdpater 工作流程 43
图 3.14 BMAdapter 工作流程 45
图 3.15 L 交易事务处理流程 47
图 3.16 系统 F 交易报告可靠性保证 48
图 4.1 状态转移图 52
图 4.2 非实时输出系统的处理流程 53

第1章 绪论

1.1 课题背景

过去几十年 IT 行业的飞速发展和 Internet 出现后所带来的深刻变化，使企业形成了极为复杂的企业 IT 应用实体和网络。这些变革也使得许多应用于金融、交通、通信、能源、国防等领域的实时系统也随之发生根本性的改变，整个实时系统越来越表现出分布、异构、松散耦合的特点，典型的例子如：分布式的金融业务处理系统、道路交通管理系统、加工控制系统、以及一些移动通信领域的控制系统。通常来说，今天很多跨国公司的业务处理大多是由分布的、基于信息的计算机系统来实现的。尽管对于不同的公司实体以及军用系统而言，组成整个系统的部件对象会有不同，但是整个系统的分布结构都是相似的：物理位置上呈分布状态的成百上千应用程序和系统部件，在各种各样的媒介中通过传输信息而完成彼此间的相互通信。

而现代信息系统更是已经被广泛使用于各行各业中，每时每刻都产生着大量的数据。在这海量的数据背后蕴藏着丰富的有价值的信息，需要有相应的机制和系统来发现和提取。

金融交易系统作为一种上述实时系统的典型,也随着 IT 业的发展由原先的人工交易方式逐渐向现在的电子交易方式发展。越来越多的交易客户放弃了传统的传真、电话的交易方式而采用在线电子交易[1]。当然，现在采用电子交易方式的交易系统一般不是孤立存在的，即它往往需要与系统外部的其他系统进行交互，而且这些系统通常是在不同的时期由不同的开发团队使用不同的技术来实现的。其中系统间的通信机制可能也是不同的，如传输协议、数据文件和消息机制等等。同时，对于一个实时股票系统而言，其所需要处理的股票或者面向的交易客户通常都是数以千计的，而交易事件更是要以十万、百万计，这就造成了系统内部数据量的庞大。另外，作为金融系统，系统响应时间的重要性也是不言而喻的。总而言之，一个实时股票系统不仅具有了分布、异构、松散耦合这些结构特点，而

且也不可避免地要求能处理大量的数据并同时给出及时的响应。

实时股票系统就是本文所要设计和实现的一种实时系统。

1.2 理论背景

根据上文提到的实时系统的结构特点和数据特点，本节将从这两个方面出发，分别探讨传统系统在系统架构和数据处理方式上的不足，比较得出更适合此类系统的系统架构和数据处理方式。

1.2.1 传统系统架构的不足

首先，传统的系统架构对于不同系统组件之间的交互设计往往都只关注于实现这些交互的系统组件的结构方面，而这种方式只适用于整个系统存在于单一的内存空间中并且在单一的系统开发团队的控制之下。而下文所设计的实时系统因为分布、异构的特征，使得传统系统架构根本无法满足这类系统的适用要求。并且在交互的代价和重要性都急剧上升的同时，传统系统却没有提供丰富的结构机制：如继承、多态等。所以，依靠关注系统组件结构的传统系统架构是较难以满足本文所要实现的实时系统的要求[2]。

其次，传统的系统架构因为利用了系统内部方法间的调用来实现交互，即一种方法调用另一种方法要求被调用的方法执行某些操作，并同时等待被调方法执行完毕后的返回结果，这使得传统系统有三个显著的特征：连续性、协调性和语境。协调性指的是系统的同步执行，即调用方法要等待被调用方法返回结果后才可继续执行。而连续性则保证在被调用方法执行完毕后，调用方法才可以往下继续执行。最后的语境特征是对协调性和连续性的保证，即通过调用堆栈来保存临时变量，使得被调用方法执行完毕之后，才可以恢复原来的状态，使得调用方法继续执行。显然，这些特征使得基于传统系统架构设计的整个系统的不同部分具有紧耦合的特点。而我们要实现的实时系统因为交互的不同部分在不同团队的控制下，因此必须尽可能地使得这些部分松耦合才能有更大的变化空间，而传统系统架构也是难以满足这点需求的。

最后，基于传统系统架构实现的系统很难真实地模拟现实世界的场景如：公司收到客户的订单、网络服务器收到用户的登录需求。对于这些场景而言，系统并没有真正地去触发这些行为，而是仅仅由于连接系统的外部环境所传输进来的一些信息，但是就是这些信息，引起了系统必须采取相应的行为对之进行监控、处理。

1.2.2 事件驱动架构（EDA，Event Driven Architecture）的提出

传统系统架构无法满足本文所要实现的实时系统的要求，但我们可以从其不足之处进行分析来构建出真正适合的新的系统架构：

传统系统架构中系统的交互通常是以类似调用的命令方式存在（如上文提到的一种方法去调用另一种方法），这种交互方式使得系统运转依赖于各部分不同的结构，系统耦合性太强。如果要改变这种方式，首先不妨认为系统中的每个部分都是独立的，而不同部分间的联系和交互则可通过间接的方式来进行：即在这些部分之间加入通道（Channel）的结构来联系它们；其次，交互的信息载体不再是命令，因为这样的命令信息往往要预先知道下一步所需执行的动作，并指明特定的命令执行者。相反的，交互方仅仅需要将交互的信息以一种事件的形式发布出来，并存放在之前的通道中，而由真正需要进行交互的另一方（或者是多方）从通道中去获取这些事件，并来进行交互行为的后续动作，而这些后续行动对于交互信息的发布方来说则是透明的。这便是事件驱动架构最初的构想。

1.2.2.1 事件无处不在

事件可以简单的定义为事物状态的一次值得关注的改变。我们将其作为交互信息的载体主要是因为事件在现实世界中是无处不在的：订单管理系统接收到一个新的订单，系统状态由“等待订单”变为“处理订单”；或者行驶中的汽车的油量低于某个临界值，油量仪表显示由“正常”转为“不足”；当用户购买完一件商品，这件商品的状态就由“待售”变为“已售”等等。以上这些状态的改变都是以事件的形式表现出来的。虽然现在已有很多计算机系统被设计出来对这些

事件进行监测、处理，如常见的一些嵌入式系统，但总体而言，这些已有系统可以监测到的大多是一些非普遍存在事件，甚至是一些人所无法看到的事件。而在上述各类实时系统中，随着系统中不同的应用模块间的交互的增加，越来越多的事件类型将在不同的程序和系统间被监测和传送。

以上文中的订单管理系统为例：当订单管理系统从网站上或者订单登录程序中收到一个订单事件，就要将此事件转发给其它关注该事件的应用程序或者系统，关注该订单事件的可能包括金融系统，该系统监测订单中是否包含信用卡信息，并且该信用卡能否有效地为订单付款；金融系统处理完订单事件后，发送账单支付事件给货运系统。关注订单事件的另一系统是仓库系统，该系统验证仓库中是否有足够的货源满足订单的需求，如有则发出存货分配事件给货运系统。此处的货运系统一直监听着存货分配和账单支付这两个事件，当货运系统收到这两事件之后即进行发货。图 1.1 是整个订单系统间的事件交互（蓝框矩形代表系统，红框矩形代表事件）：

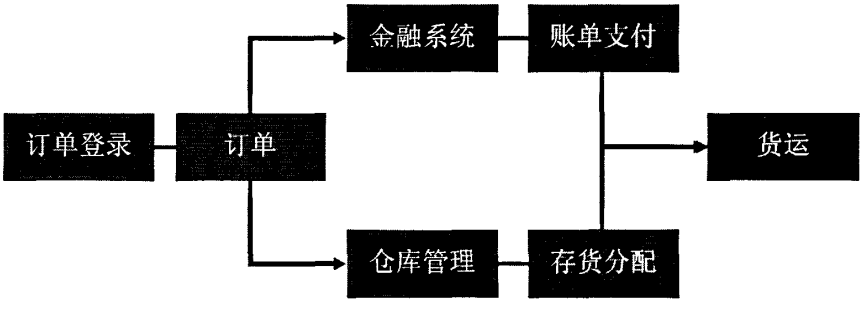

图 1.1 订单系统的事件交互

其实整个系统中事件的交互远远不止如此，如当金融系统验证到客户提供的信用卡过期时，要发送一个 email 事件给 email 网关，由此发送通知给用户要求其更换新的信用卡号。或者当仓库管理系统监测到仓库库存中已没有足够的货源来满足订单需求时要发送库存不足事件给进货系统，整个系统的事件交互行为将这样一直持续下去。David Luckham 用事件云集（Event Cloud）来描述类似的大量

事件在不同系统间的交互[3]。图 1.2 用事件云集来描述该订单系统：

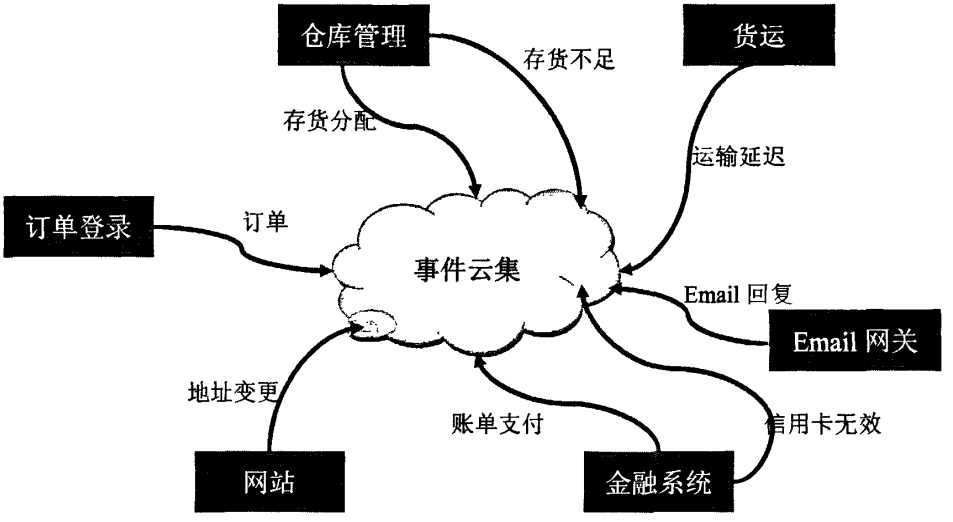

图 1.2 事件云集描述订单系统

1.2.2.2 事件抽象

确定事件作为系统交互的信息载体后，接下来面临问题的就是这种信息表示的有效性，即使得事件信息的表述形式可以被计算机所接受和理解的。因此我们就必须对现实中的事件进行抽象，以一种更为模式化的形式表述出来，为了完成关于事件的抽象描述，现引入了规格（Specification）和发生（Occurrence）这两个术语，所谓规格，就是事件的抽象定义（Definition），相当于面向对象编程语言中的类，而发生是事件的出现，则是一种具体的实例（Instance），相当于面向对象编程语言中的对象[4]。计算机正是通过获取事件的定义来识别事件，然后又通过获取事件的对象来最终得到现实中所发生事件的实际信息。

1.2.2.3 事件驱动架构

当明确了通道作为系统内部各个应用程序和部件间的连接中介，以及需要把事件当成交互信息的传输方式之后，事件驱动架构这种新型的系统架构便进入了架构研究人员的视野。此架构认为系统可以由事件的触发而采取相应的处理行

为。其主要的特点在于可以提升系统对事件的产生、反应、监控和处理能力[5]。EDA 作为一种系统架构其核心思想主要是通过发送与接收数据事件将系统中的各个部分集合关联起来。根据上面的分析以及这种系统架构本身的特点，将其应用在我们所研究和设计的实时系统中的设想就有了一定的可行性。

随着系统中不同应用程序间交互的不断增多，以及系统不断增加的与外部交互事件数据的能力，使得现有实时系统中事件数据的特征（无论是数量、获取的速度、复杂度、易变性）都是原有实时系统中所处理的数据所无法比拟的。那么，事件驱动架构中对于事件数据的处理和传统数据处理方式有什么不同？或者说我们要怎么样改进，才能使事件驱动架构真正适合我们所要实现的实时系统？

1.2.3 传统数据处理方式的不足

传统的数据处理方式通常侧重对静态数据的处理，这种处理方式可以归纳为三个步骤：

- 输入数据并保存；
- 对保存的数据建立索引（index）；
- 查询索引后的静态数据

在这种处理方式下，数据库组织、管理和索引应用系统的所有数据，以此控制对所有数据的处理：在数据没有在数据库中保存并建立索引之前，是无法对其进行操作的[6]。分析静态数据的实质就是查询处理过去保存的数据，然后根据查询所得结果来给出最后的结论。

还是以上文的订单管理系统的为例，当订单管理系统收到一个有效订单事件后，要求货运子系统将货物发给订单客户。在此之前，需要从数据库中获取用户的地址信息，再把地址信息连同订单事件发送给货运系统要求其进行发货。这个过程如图 1.3 所示，显然，对于每个订单事件的发货处理都有一段时间延迟。

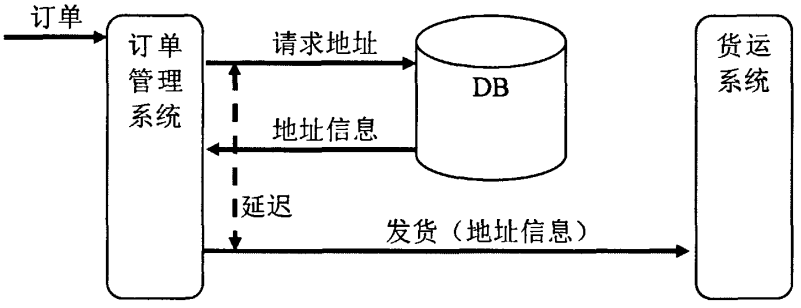

图 1.3 传统数据处理方式中的订单发货

如果在事件驱动架构中继续使用这种传统的数据处理方式，则当大量事件数据在很短的时间内涌入系统时，由于其以数据库为中心，为了使得该数据可以在未来进行查询，就必须不断地建立和更新相关数据的索引，这将极大地降低系统的响应时间，使得系统的处理速度无法跟上事件数据流入的速度。同时，由于数据库中对数据的访问只能通过查询来进行，所以当不断有新事件进入时，系统必须不断地查询数据库中的静态数据来验证新收到的数据是否满足系统所寻找的特征，显然，这样的一种数据处理方式十分的低效。

1.2.4 事件驱动架构中的事件处理

作为事件驱动架构的数据处理方式，事件处理将不再依赖于数据库，其改变了传统的以数据库为中心的数据处理方式。取而代之的是对事件进行处理的部分系统自身保留了必要数据的拷贝，同时，系统也可对这些拷贝数据进行更新[7]。如上文例子中的货运系统就需要保存客户的地址信息数据。所以，当订单管理系统收到订单事件时就可以直接发出发货事件，而货运系统则可根据发货事件中的订单信息自动找到订单用户的地址。这样就可以有效避免事件处理中的延迟，在处理大量数据时的改进将是非常可观的。图 1.4 描述了改进后的订单发货流程：

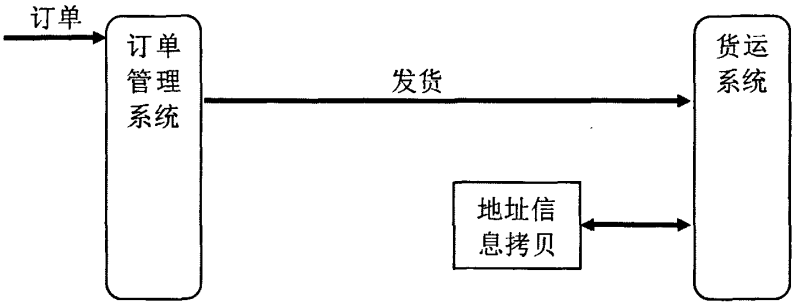

图 1.4 事件处理方式中的订单发货

1.3 本文工作及结构

本文主要是对交易清算系统T进行功能分析研究并实现，首先通过详细分析这类系统所固有的一些显著特征：有多样的快速变化的输入事件流，需要实时监控输入数据事件并做出快速决策，可以在较短时间内执行预先定义的处理行为。所以，保证系统有能力去监控并处理快速多变的输入事件数据流是系统成功的关键。其次，我们结合旧有系统架构的不足，深入探讨了事件驱动架构，并尝试将这一新型的系统架构应用到实时交易清算系统T的实现中去。在研究和实现系统T的过程中，使用了事件处理框架平台 Apama[8]，并采用其一种实现事件处理逻辑的面向事件编程语言 MonitorScript[9]。本文探讨了在系统 T 中使用 Apama 和 MonitorScript 的特点，并对 Apama 和 MonitorScript 应用在 T 实现上的优势和不足进行研究。最后，针对现有复杂事件处理平台缺乏可持久化能力的状况，提出并分析了两种较通用的增强方案。

本文的主要结构如下：

第一章， 绪论：介绍了本文的课题背景、理论背景和全文结构；

第二章， 事件驱动架构的研究和综述：介绍事件驱动架构的基础和研究内容，并和旧有的系统架构进行简单的比较；再分别介绍 Apama 和 MonitorScript 这两种用于系统T的事件处理平台和事件编程语言

第三章， 交易清算系统T的具体实现：详细分析了项目背景，系统的需求，

然后基于事件驱动架构思想应用 Apama 平台设计出整个系统的架构，包括系统与外部系统的连接接口，系统几个关键模块的设计，以及如何保证性能和异常恢复后的可用性；

第四章， 针对现有复杂事件处理平台缺乏可持久化能力的状况，通过对现行复杂事件处理平台处理流程的分析和抽象，提出了两种较通用的增强方案，实现了可恢复、可持久化的事件处理平台；

第五章， 全文总结与未来系统 T 的升级：总结本文所做的工作。

1.4 本章小结

本章首先介绍了文章的课题背景，并结合这一背景引出了事件驱动架构这个新型的系统架构。最后，阐述了本文的章节组织结构。

第2章 事件驱动架构、Apama、MonitorScript

系统由各种各样的模块部件和独立应用程序组成，如何关联这些部件和应用程序，并使它们可以进行交互是系统架构所要解决的问题。事件驱动架构适合应用在具有松散耦合的软件部件和服务的系统设计和实现中，其主要的优点在于可以提升系统对事件的产生、监控、反应和处理能力。事件驱动架构的核心思想在于通过发送与接收事件数据，将系统中的各个部件和应用程序关联集成起来，即系统中的某个部分（定义为生产者）产生了一个事件信息，而这个事件信息又是系统中另外若干个部分（定义为消费者）所需要关注的，此时，生产者需要能够及时的把这个事件信息发送给所有的消费者，然后消费者就可以做出适当的相应处理行为[10]。

2.1 事件驱动架构的定义

事件驱动架构是一种新型的软件系统架构，由通过松散耦合的不同应用程序之间交换事件来进行交互，这些应用程序就被称作事件驱动应用程序。事件驱动应用程序可以充当事件的生产者或消费者，或者同时充当这两种角色[11]。即当事件生产者生成一个事件后将其发布出去，此时可能有一个或者多个事件消费者在同时关注此事件，当这些事件消费者监听到这个事件后，就采取相应的处理动作。事件驱动架构的主要特点在于可以提升系统对事件的产生、监控、反应和处理能力。这种架构也因此主要应用于具有松耦合的服务和软件部件的系统设计和实现中。

2.2 事件驱动架构的特征

作为一种新型的系统结构，事件驱动架构并不仅仅只定义了一些步骤，然后根据它们以事件为交换信息载体在系统不同的应用程序之间进行交互，更重要的是，事件驱动架构本身要成为一种具有自身优势并日益成熟的系统结构，必须具有一些固有的优秀特征[12]：

- **异步性**

  事件驱动架构的初衷就是使得基于该架构所设计的系统能够适用在具有松散耦合特点的实际应用环境中，因此系统中事件的生产者与事件消费者的之间往往是相互透明的[13]，也就是说事件生产者不知道事件消费者所要执行的处理行为，甚至是不知道有多少事件消费者在等待某个事件的生成，所以事件生产者在将事件发送出去之后，就可以进行下一步的行为，而无须等待事件消费者对事件处理的结果返回；

- **术语定义**

  系统需要预先定义一些术语来更好的表达事件的定义，特别是用来表达事件的层次[14]。事件消费者可以根据层次来更加清晰的表述其所关注的某个单独的事件或者某类事件；

- **事件粒度的可控性**

  系统中不同的应用程序之间，对于不同事件的接受粒度总是有相应的物理限制[15]，往往越是底层的程序交互，这样的限制就越是明显。所以在定义事件的时候，必须根据相应的实际需求，对事件的粒度做出合理的选择；

- **时效性**

  系统可以在事件发生后即时的把它发布出去，而不是像传统的典型定时批处理程序，将事件先存储在本地，然后等待处理周期的到来再进行相应的处理；

- **广播通信**

  每一个系统中的事件，都可能有多个事件消费者在同时等待它的发生并等待处理，所以当事件的生产者在发送该事件时，可以如同广播的方式同时将该事件信息发送给所有的事件消费者[16]；

- **事件处理**

  系统在监听到所需关注的事件后，就需要对该事件进行相应的处理，由于事件可以抽象为简单事件和复合事件，以及不同事件间存在联系，如

因果、聚合等等，所以事件驱动框架可以采取不同的事件处理方式：简单事件处理、复杂事件处理、流事件处理。

事件驱动架构有着简单和精确的特点，因为基于这种架构的系统是通过对现实世界进行建模，抽象出发生的事件，所以整个系统能较容易的表示真实世界[17]，所有这些特征成为了事件驱动架构的优势，也使得了众多企业应用系统集成厂商坚定的认为事件驱动架构将成为下一代系统架构变革的目标所在。

2.3 事件驱动架构的构成

事件驱动架构是由不同的实现组件来构成的，可以将这些不同的实现组件细分成下面五个种类，这五类组件并非完全独立分开的，即某一个具体的实现组件（如事件处理）可能同时归属于这五个种类中的多个[18]：

2.3.1 事件资源

事件驱动架构里存在着各种各样的企业事件资源——业务处理、服务、应用程序、数据仓库，还包含人力资源以及一些自动的代理，这些代理可以用来生成事件并同时执行事件驱动的下级行为。

2.3.2 事件工具

事件工具包含了事件开发工具和事件管理工具。其中事件开发工具主要用来定义事件规格、事件的处理规则和管理事件的订阅者。而事件管理工具则是用来监控和管理事件消费者、事件生产者、事件流的动向和事件处理构件，同时，也可用来进行事件处理行为的统计分析。

2.3.3 事件元数据

一个定义良好的事件驱动架构必须有一个与之匹配的强大的事件元数据结构，事件元数据包括事件的规范定义和事件处理规则的设计。事件的规范定义又包含了事件格式，事件生产者，事件处理引擎，事件转换以及事件订阅者。虽然

目前还没有统一的事件定义和事件处理规则符号，但将这些统一起来应该只是一个时间问题。

2.3.4 事件处理

事件处理的核心是事件对象实例和事件处理引擎[19]。事件对象实例通常是持久的，可以保存来进行审计和一些事件趋势的分析。事件处理引擎则可以分为简单和复杂两种，其中的简单事件处理引擎通常都是自行研发，因需求的不同而不同。而复杂的事件处理引擎则是由专门的复杂事件处理引擎开发商来提供的。

事件驱动架构最初的原型是事务分析架构。现在的事件驱动架构中的整体流程逻辑也是基于这个原型所派生出来的。我们可以相应地把现在的事件处理流程分为四个逻辑层：事件生产者、事件传送通道、事件处理引擎和事件驱动的下级行为[20]。

2.3.5 企业集成

企业集成枢纽服务在整个事件驱动架构中扮演着非常重要的角色。有一些必须的集成枢纽服务如：事件传送通道，服务请求，事件处理（事件过滤、事件分发以及事件变换），业务处理请求以及企业信息的访问、订阅与发布。

另外，图2.1以分层的结构方式来描述这些不同的组件：

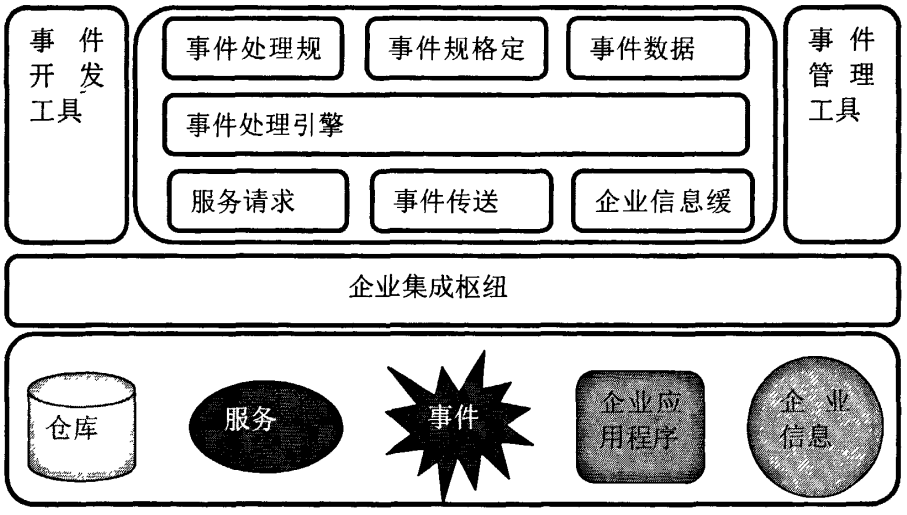

图 2.1 事件驱动架构的分层结构

2.4 Apama 和 MonitorScript

Apama 和 MonitorScript 是用于实现金融交易清算系统 T 的事件处理平台和事件编程语言（EPL，Event Programming Language），下面分别对 Apama 和 MonitorScript 进行简单的介绍。

2.4.1 Apama 平台的由来和特点

为了满足实时系统中事件处理的需求，需要一种新的完全不同于传统数据处理方式的平台。上世纪 90 年代，剑桥大学的 John Bates 意识到了这种需求并开始着手对这种新平台的研究。后来这些研究被商业化，就促使了 Apama 这个事件处理平台的诞生。如今，Apama 平台是 Progress 公司的一个商业产品。

Apama 平台是一种模块化的、可扩展的事件处理平台，这种平台的核心功能可以归纳为：

- 实时监控由消息设备或市场数据发送设施所传送过来的数据；
- 在内存中快速的分析这些事件，同时把多个存在特征相关或者次序相关等关联模式的事件联系起来；
- 触发新事件的生成，该事件表示对输入事件的处理行为，即对输入事件

的分析结果的响应动作。

为了达到以上的功能，Apama 平台颠覆了传统的数据处理流程，不再是如上一章中所描述的“存储—索引—查询”的模式，Apama 引入了一种称为 Correlator 的实时事件关联和处理引擎，可以预先定义事件的匹配模式（类似数据库中查询的作用）以及相应的处理行为，然后将这些模式和处理行为预先注入到 Correlator 里作为事件处理的 Scenario[21]，这些 Scenario 实现了下面的一些具体功能：

- 定位值得关注的事件：异常事件或达到某个预设阀值的事件，如当某只股票的价格超过当月的最高值时触发该事件；
- 进行事件分析和关联，然后生成可以被再次分析的事件，如不断的生成某只股票在过去五分钟的动态平均价格；
- 通过比较和联系过去一段时间内的多个事件的具体值，来判断事件的变化趋势；
- 在业务模式中组合时间有序的事件来表示一个复杂现象的产生或者一个业务过程的完成；
- 触发相应的处理行为作为某些预先定义的匹配模式的响应，如监测到某个合适的机会时自动的买卖股票。

除此之外，Apama 平台性能优越的另外一点是可以支持几千个类似的 Scenario 同时运行，当这么多的 Scenario 同时运行时，每个 Scenario 都会有本身所监控的事件、匹配模式以及处理行为，而且不同的 Scenario 之间必要时还可以进行交互（即某个 Scenario 的处理输出可作为另一个 Scenario 的输入）。

Apama 平台的这些特征使其非常适用在拥有快速变化的多样输入数据流，需要实时监控输入数据并快速做出决策以及采取相应的预期处理行为等需求的这类系统的实现。

2.4.2 Apama 平台结构

Apama 平台的总体结构如图 2.2 所示：

下面我们由下往上逐层来分析 Apama 平台中的组成部件：

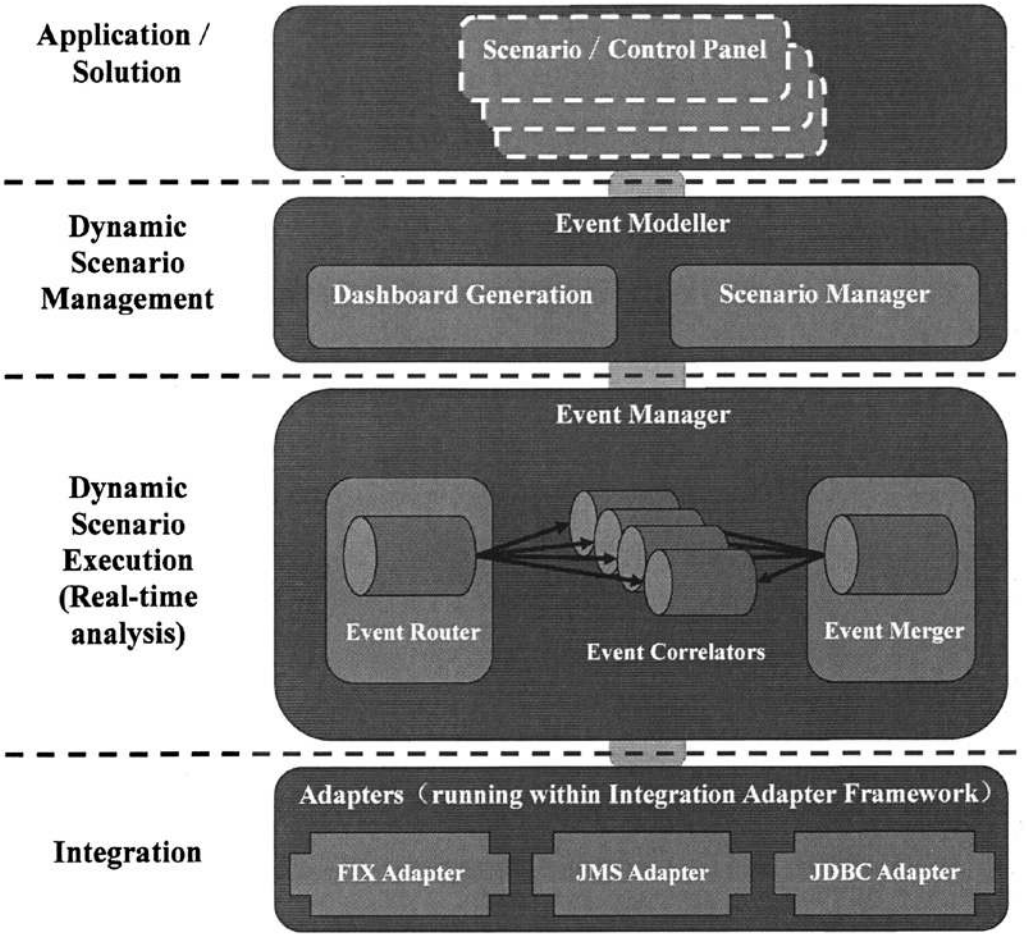

图 2.2 Apama 平台结构

1. 集成层（Integration）

通常，Correlator 所需要处理的事件数据来自不同的事件输入方如：股票报价设施、新闻发布设施、RFID 读卡器或者各种传感器，但是对于 Correlator 而言，其功能只是处理单一的事件格式，所以就必须有一种转换器来协助，它可以将外部的输入数据格式转化为 Correlator 可以识别的事件格式并同时可以将 Correlator 中的事件格式反向转换为外部服务可以识别的数据格式。集成层中的适配器（Adapter）就是用来实现这一功能

的。另外，自定义的 Adapter 必须遵循集成适配器框架（IAF，Integration Adapter Framework）来进行实现。

2. **应用 / 解决层（Application / Solution）**

   用户可以将多个相关的 Scenario 和管理这些 Scenario 的仪表板打包在一起，生成一个针对某类特殊问题的可配置的解决方案，这一层实际就是站在系统实际应用的角度来对动态场景管理层所提供的功能进行高级的整合。

2.4.3 Correlator

通过以上介绍，不难看出对于整个 Apama 平台而言，最为核心的部分莫过于 Correlator，因为这里不仅是 Event 和 Scenario 装载的地方，而且还是通过执行 Scenario 来对 Event 模式过滤、匹配和处理的地方。Correlator 的内部构成如图 2.3 所示：

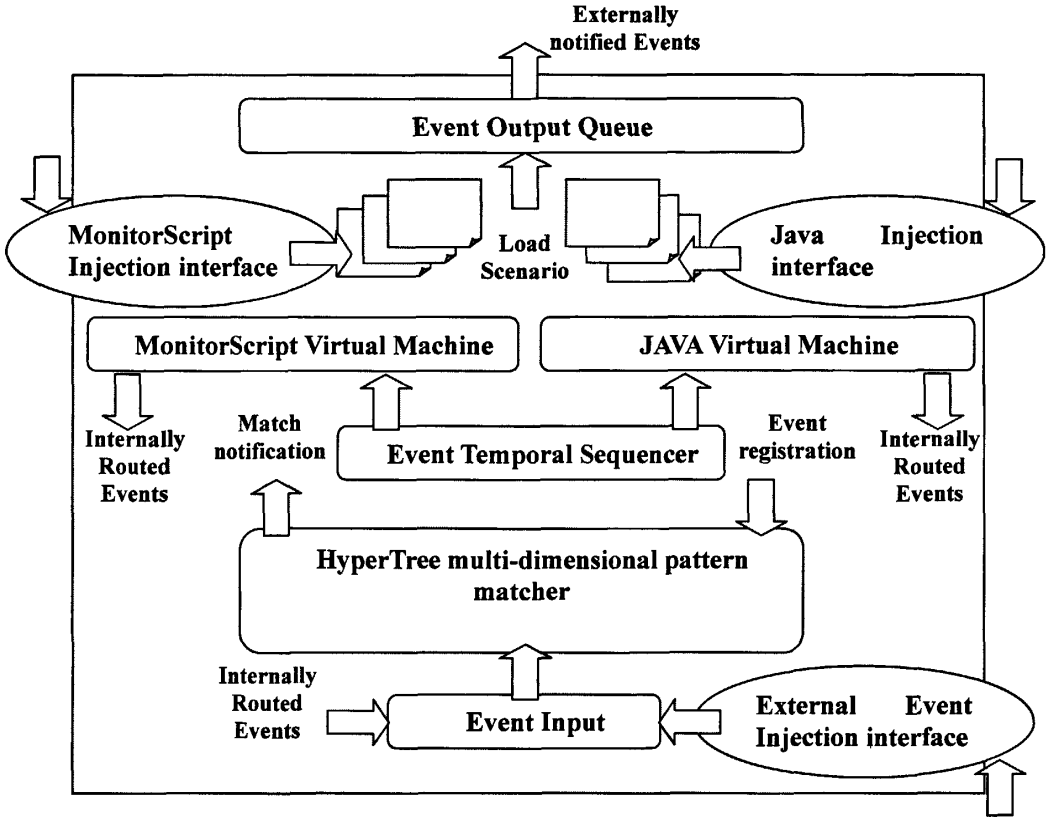

图 2.3 Correlator 结构

下面简要介绍 Correlator 的几个主要组件：

- 事件输入 / 输出队列（Event Input / Output Queue）

  事件在被 Correlator 接收之后，先存放在输入队列中，同样的，事件在 Correlator 中被处理之后，也是先存放在输出队列中，再发往其他的地方，这里要说明的是，不管是输出队列还是输入队列，其中保存的事件的发送方和接收方都可能为 Correlator 内部的 Scenario，另外，当某个 Scenario 发送的事件还会被 Correlator 所接收时，该事件被插入到输入队列的头部，该过程如图 2.4 所示：

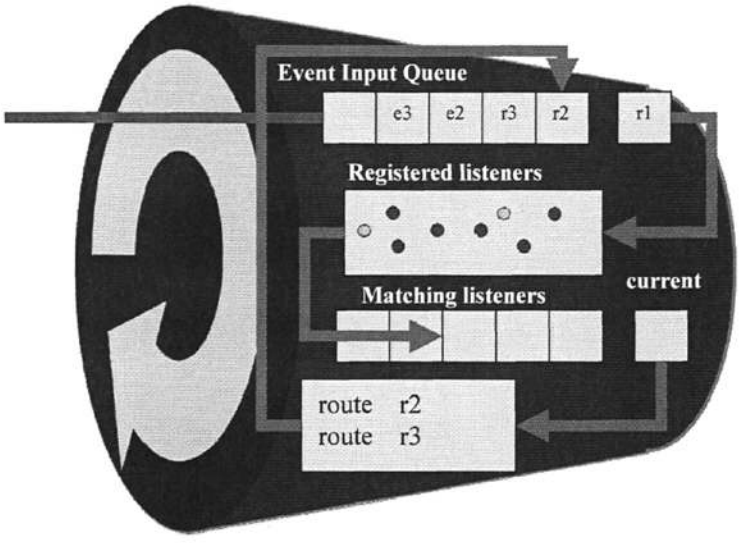

图 2.4 Correlator 内部事件输入 / 输出队列

- **超树（HyperTree）**

  超树包含了用来进行高效的多维事件过滤的数据结构和算法，超树预先载入了类似事件多维查询的事件模板（Event Template），事件模板是进行事件监测的最小单元。超树具有良好的性能，即使同时有上百万的事件模板需要进行多维查询，也不会有很大的延迟；

- **临时序列（Temporal Sequence）**

  超树提供对单一事件进行匹配的模式，而临时序列则将多个单一的事件关联起来；

- **MonitorScript 虚拟机和 JAVA 虚拟机（MVM、JVM）**

  这两个在 Correlator 中内置的虚拟机可以执行相应的用 MonitorScript 或 JAVA 实现的 Scenario。

2.4.4 MonitorScript

Apama 平台中对于事件的处理逻辑都是通过注入 Correlator 的 Scenario 来实现的，而为了对 Scenario 进行定义，开发人员可以使用一种特别为 Apama 平台所设计的事件编程语言（EPL，Event Programming Language）：MonitorScript。MonitorScript 类似脚本语言解释执行，无需进行编译，其所实现的代码以.mon 格式的文件形式存在（类似 Java 语言对应的.Java 文件）。当*.mon 文件被注入到 Correlator 之后，Correlator 可以对其进行解析。这一节，我们将简要的介绍这种类似脚本语言的事件编程语言。

2.4.4.1 事件类型定义

Correlator 要识别输入事件中的事件类型，首先需要对输入事件中值得关注的事件进行抽象，即预先定义所需要关注事件的类型[22]，这是 MonitorScript 所提供的最基本的功能：事件类型定义。Correlator 可以同时处理大量的不同的事件类型，图 2.5 是 MonitorScript 定义的一个股票交易所的股票报价事件类型：

```
Event StockPrice{
  string symbol;
  float price;
}
```

图 2.5 股票事件类型定义

2.4.4.2 Monitor

如果说事件类型定义是 MonitorScript 的一个基本构造结构，那么另一个基本构造结构无疑就是 Monitor 了。顾名思义，这个部件就是用来对事件类型定义中的事件进行模式匹配。除此之外，更重要的作用是还包含了对匹配的事件进行相应处理的逻辑[23, 24]，进一步说，Monitor 由以下两部分组成：

- **Listener**

  Listener 定义了 Correlator 监控的事件所需要满足的条件模式，当有事件

注入到 Correlator 时，首先判断该事件是否匹配某个已存在的 Listener 所定义的条件模式，匹配则进行 Action 里的操作，反之，抛弃该事件；

- **Action**

  Action 类似于 C 或者 C++等高级语言中的函数，它定义了对于匹配的事件所进行处理的逻辑，值得注意的是，在 Action 中还可以含有 Listener，当有新的事件匹配了 Listener 的模式时，可以进一步触发不同的 Action。

下面图2.6(图中绿色字体表示Listener，红色表示Action)举例用MonitorScript代码实现了当监测到IBM的股票价格高于100时，打出log记录该事件（股票价格的事件类型定义如图2.5所示）：

```
monitor PriceAlert{
    StockPrice stockPrice;
    action onload{
        on all StockPrice("IBM",>100):stockPrice alertThePrice;
    }
    action alertThePrice{
        log "IBM has hit" + stockPrice.price.toString();
    }
}
```

图 2.6 monitor 的股票价格例子

2.4.4.3 组合事件

上面我们提到的例子都是对单一事件而言的，而在实际应用环境中，各种输入事件间往往存在着各种各样的联系，如时间关系、因果关系等[25, 26]，此时在MonitorScript 中同样可以用代码来抽象表示这些事件间的联系，下图 2.7 的例子用 MonitorScript 代码实现了当某只股票价格在 1 分钟之内提高了 1%时，打出 log 记录该价格提高的发生。

```
monitor PriceRiseAlert{
    StockPrice stockPrice;
    action onload{
        on all StockPrice(*,*):stockPrice —>
        StockPrice(stockPrice.symbol, >=stockPrice.price*1.01)
        within(60.0) alertThePriceRise
    }
    action alertThePriceRise{
        log "The price rise of stock:" + stockPrice.symbol + "has
        occurred";
    }
}
```

图 2.7 monitor 组合事件联系的例子

2.5 本章小结

本章主要对本文所使用的事件驱动架构做一个全面的介绍，包括这种架构的产生、特征以及主要的组成构件。后面是对本文要实现的交易清算系统 T 所使用 Apama 平台和 MonitorScript 语言做一个简单的介绍。包括 Apama 平台的结构以及其核心组件 Correlator 的作用和组成，还有 MonitorScript 简单用法。

第3章 交易清算系统T的实现

金融市场瞬息万变，为了赢得更好的商业优势，能够处理大量的实时输入事件数据并且在一定的时间范围内对这些事件数据进行模式匹配、过滤、交互处理成为许多金融系统所必须要满足的首要需求。根据上文对事件驱动架构的深入分析，可以得出该架构非常适合使用在金融系统的实现中，所以，本章将尝试基于这种架构，并利用特定的事件处理平台 Apama 和事件编程语言 MonitorScript 来完整的设计和实现一个交易清算系统T。

3.1 项目背景

3.1.1 金融交易中的清算

清算是金融交易过程中不可缺少的一个重要步骤，是指证券买卖双方在证券交易所进行的证券买卖成交之后，通过证券交易所将证券商之间证券买卖的数量和金额进行计算的过程。通过这个过程，各方将买卖股票的数量和金额分别予以抵消，然后通过证券交易所交割净差额股票或价款的一种程序。

清算的意义，在于同时减少通过证券交易所实际交割的股票与价款，节省大量的人力、物力和财力。证券交易所如果没有清算，那么每位证券商都必须向对方逐笔交割股票与价款，手续相当繁琐，占用大量的人力、物力、财力和时间。

3.1.2 自清算（self clearing），系统B和系统T

清算是银行及证券交易系统内必须的步骤，但它通常是通过中央结算系统进行。但一些大型电子证券交易商会通过自行开发软件，以便自结算来节约交易成本，这就称为自清算。在美国金融市场，经纪人（broker）只有通过了证券交易委员会（Securities and Exchange Commission, SEC）和全国证券交易商协会（National Association of Securities Dealers, NASD）的授权后，才具有持有（carry）客户账户的能力，才能进行自清算。这样的经纪人被称为自清算经纪人（self-clearing broker）。

自清算给了 broker 更大的权力，为了更好的保证客户的利益，证券交易委员会有多条规章制度（multiple rules and regulations）来保护客户的资产。证券交易委员会规则 15c3-3CR（Customer Reserve）要求了在客户有价证券（Customer Securities）和客户基金（Customer Funds）等多方面的规则。其中要求自清算经纪人必须将交易信息向美国三大金融清算监管机构汇报，它们是 NSCC（National Securities Clearance Corporation），DTCC（Depository Trust Company）和 OCC（Options Clearing Corporation）。

自清算经纪人作为一个清算的身份角色，除了自己开发软件还可以选择一个第三方的系统来支持该汇报业务，该系统即为清算代理（clearing agent）。作为新进的自清算代理人，银行 S 急需快速建立自己的自清算系统，采用成熟第三方系统无论在时间上还是风险控制上都是更好的选择。公司 A 的系统 B 作为美国金融市场一个成熟的清算代理机构成功取得了代理权，而连接银行 S 的内部系统和系统 B 的需求促使交易清算系统 T 的产生。至此，系统 T 的需求也就浮出水面。

3.2 系统 T 在业务逻辑上的功能需求

具体到系统而言，功能需求主要包括以下几个方面：

1. **系统 F 的实时交易清算（Trade Clearing）**

   系统 F 是银行 S 主要的交易系统，当交易员（trader）在系统 F 的前台做出订单（order）并通过各种方式（如内部撮合 cross 或 broker fill）完成这笔订单后，相应的交易信息就产生了。该系统中的内存数据库 R 即时的反映着所有的交易信息的变化，而外界系统可以通过 TCL 脚本实现的 stream 实时监听 R 并获取其感兴趣的某部分信息。系统 T 被要求从 F 中获取所有美国市场的交易信息，转换成系统 B 所要求的格式并发送过去；

2. **系统 L 的批量交易清算**

   系统 L 是银行 S 较早期的交易系统，在美国市场上仍有少部分领域需要使用。在每个交易日结束时（End Of Day, EOD），系统 L 将当天的交易输出到一个 EIL 文件中。系统 T 被要求在每天 EOD 时从 L 中获取 EIL 文件

并提取出交易信息，转换成系统B所要求的格式并发送过去；

3. **清算结果确认（Clearing Result Confirmation）**

   金融服务公司必须把部分重要的交易信息发送给交易用户进行确认，系统F也要求实时显示交易的清算状态，对于清算出现错误的交易信息，操作员也必须尽快得到警告并及时进行修改。

4. **清算记录保存（Clearing Record Keeping）**

   金融服务公司必须有序的保存其业务和内部组织的信息，其中包括该公司提供的所有服务以及它所执行的交易。同时，这些信息必须足够充分以使得SEC和NASD规定的相关主管机构可以对这些信息进行必要的审核，来判断该公司的行为是否遵循SEC和NASD所规定的要求以及确认该公司是否履行对客户的所有职责要求。保存后的信息可以进行更新和其他修改。另外，所保存的信息必须保证真实性可用性和可靠性。

5. **清算核对（Recon job）**

   除了通过系统T进行清算，另外还有一个独立的监控系统在每天EOD时核查当天的清算结果，比对账户是否正确。为了保证这部分工作的独立性，它的实现不属于系统T的范围。

3.3 T系统在非业务方面的功能需求

1. **性能（performance）和流控制（flow control）**

   交易系统F作为银行S的主要交易系统，交易日的每天都会有数百万的交易信息产生，其中给系统T的stream上的信息数量也在百万级别。系统T需要保证在如此大数据量下系统能正常工作。另外由于两端系统的吞吐速度（throughput）相差较大，系统F的stream能以峰值约为1000条/秒的速度输出交易信息，而系统B最多只能以200条/秒的速度接受信息，处于两者之间的系统T也就担当了流控制的责任。

2. **可扩展性（scalability）**

   系统T被设计出来主要是为了实现清算业务，所以不能把它局限在某一

个特定的清算流程，即这里的从交易系统L和F到清算系统B的清算流程。系统T在各交易系统中被要求起到一个Hub的作用，当有新的交易系统和清算系统加入，或者任何其他的新清算流程时，系统T必须能够方便的扩展，同时还不能影响到现存的流程。

3. **异常恢复（recovery）和可靠性（reliability）**

清算是交易中不可缺少的一个环节，清算的缺失会直接导致交易无效这样严重的后果，所以系统T必须保证每一条流入的交易信息都能送出到系统B，并对系统B的返回进行相应的处理，即使在系统出现崩溃等重大错误时，重启后依然能顺利恢复。

3.4 系统T的构成

由上两节的项目背景和需求分析，我们可知系统的事件流程主要包括前端到后端以及反方向的这两条线路。又由于系统T需要访问多种异构的系统或应用程序，由软件工程的模块化思想，系统T内部将分模块设计。又由于Apama平台的Correlator-Adapter设计思想，系统的核心业务逻辑将在中心的Correlator节点实现，而与外部其他系统的链接交换则由数个不同的Adapter完成，系统T的最终设计拓扑结构将是以Correlator为中心的星状结构。

得益于事件驱动架构，由于各个Adapter与Correlator是通过事件交换，从而实现了松耦合的设计，又由于Apdater的加入和移除可以在系统运行时进行，当有新的外部系统加入和移除时，系统T也可以快速的进行相应的调整，从而也就完成了可扩展性的要求。而这些都是传统的系统架构所无法完成的。

在设计和实现系统T的架构前，我们可以结合上两节的项目背景和需求分析，来了解一下交易清算系统T与周边系统或应用程序之间的关系。

对于流过系统T的每一条交易信息及其清算结果，系统T通过JDBCAdapter存入数据库。这就满足了清算记录保存的要求。清算核对由另外的系统GM完成，不在本文的讨论范围。

我们可以用图3.2大体表示清算流程在各模块间的关系。

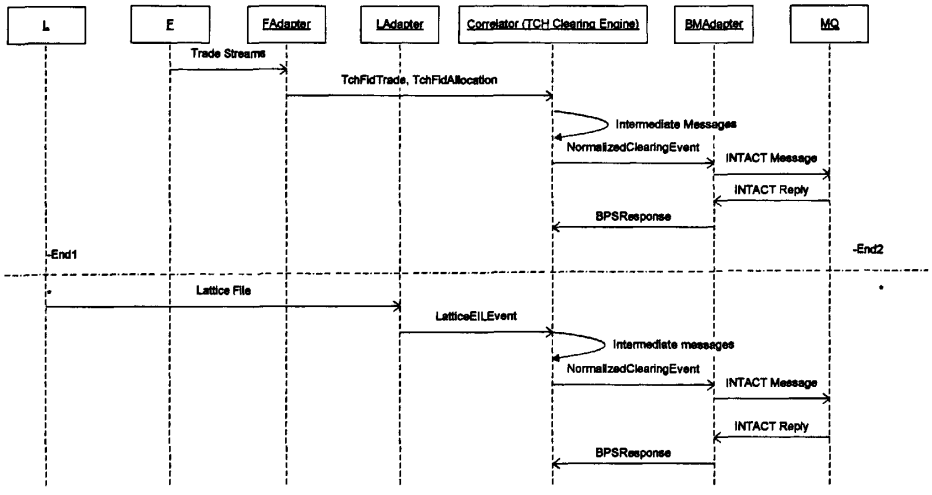

图 3.1 当前的清算流程

对于非业务方面的三项需求，可扩展性在架构设计上的体现最明显。由于需要支持未来可能的其他交易系统和其他清算机构。我们可以将清算流程抽象成两层，即交易系统接口层（Trading system interface layer）和清算系统接口层（Clearing systems interface layer），而系统 T 只是起到了一个接口的作用。可以用图 3.3 表达这种抽象的思想。

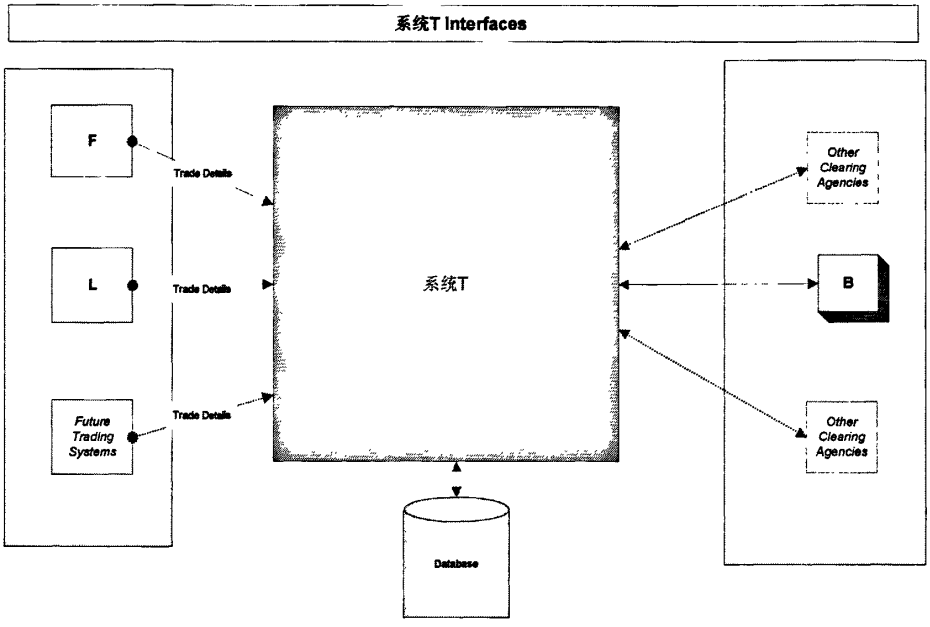

图 3.2 系统 T 和抽象后的清算流程结构

接下来在具体的实现中，需要将代码模块分为两个部分，一类是所有清算流程所公用的，一类是为某一个清算流程所特定设计的。对于公用的代码模块，应该尽量做到类似面对对象程序设计中的泛型化（generic），当有新的清算要求时，这部分模块不需要改动，只要新增加对应的特有模块即可，这样就做到了模块间尽量小的依赖关系和系统良好的可扩展性。

综上所述，考虑到可扩展性和当前的具体需求，系统 T 的最终设计如图 3.4。其中的实线部分是当前需求里的清算流程，虚线部分是未来可能的其他的流程。

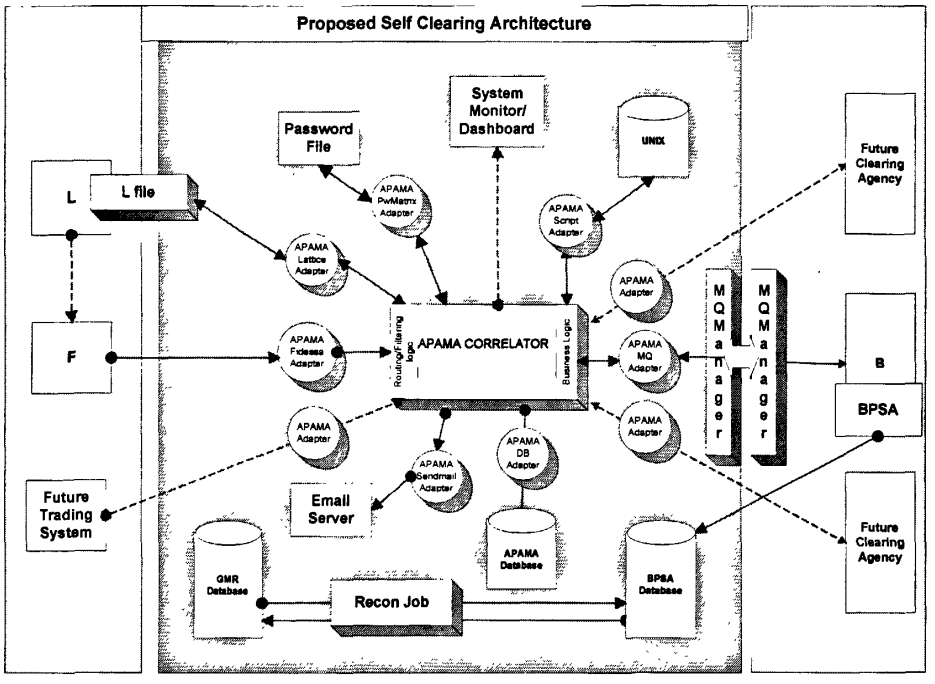

图 3.3 系统T含未知可扩展的结构设计图

由图3.4可见，Adapter机制大大的提高了系统的可扩展性。

去掉未知扩展节点，就得到了系统T的最终架构图，见图3.5。系统T大致可以用阴影部分表示，而阴影部分外表示系统外部的各个周边系统。阴影正中的correlator是系统T的核心，所有业务处理逻辑都将在correlator里完成；correlator周边的Adapter起到了连接correlator和各个外部系统的桥梁作用，correlator通过FAdapter和LAdapter从交易系统F和L获得交易信息，并回写结果（仅系统F）；通过SendMailAdapter发送电子邮件给相关用户；通过ScriptAdapter调用操作系统上的shell命令和脚本来完成特定工作；通过BMAdapter来将交易清算信息发送到系统B并获得清算返回结果；通过PwMatrixAdapter来获取系统运行时需要的各种密码（如数据库密码）；通过JDBCAdapter来存取访问数据库。下面将分别介绍各个模块的设计。

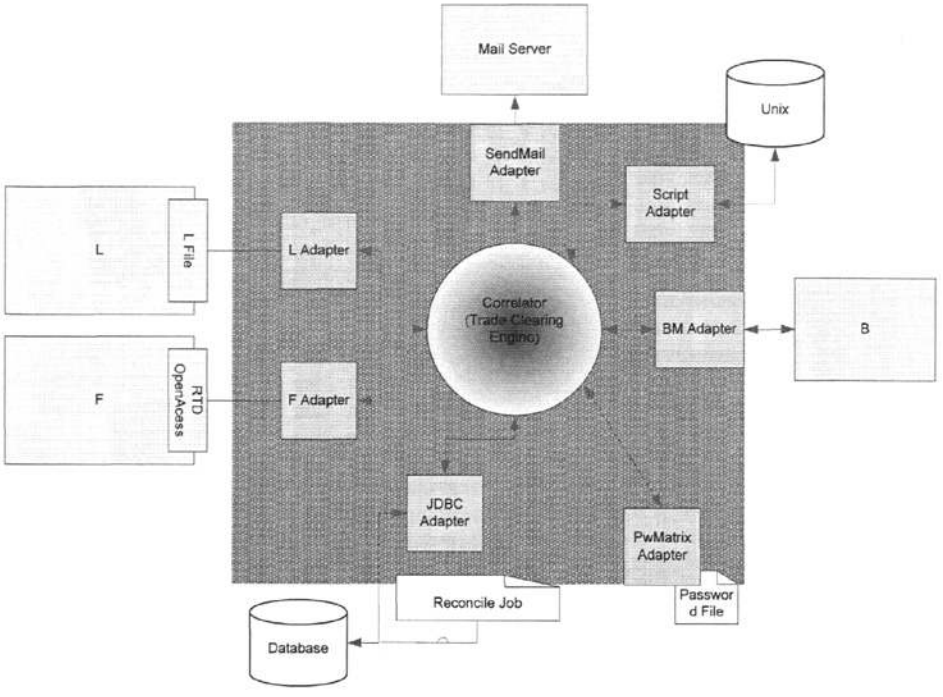

图 3.4 系统T最终的结构设计图

3.5 Correlator的业务处理逻辑的实现

根据软件工程模块化的思想，correlator中的业务处理逻辑的实现也分模块设计。每个模块负责逻辑上相对独立的任务，由于Apama平台有基于事件处理的固有特点，模块间可以通过事件交互的方式轻易实现松耦合。又由SOA的思想，该系统中每个模块的产生都可以看作是为了提供某一种服务，所以所有的模块都可以以Service的概念创建。

Correlator中的模块主要包括11个：LService，FService，BCService，BMService，SystemAlertService，IAFStatusManager，DBService，LoggingManager，PasswordService，SendMailService和AppManager。各模块的功能如下：

LService：负责与LAdapter交互，将交易系统L中的交易信息发送到BCService，同时还要负责管理和监控LAdapter的状态；

FService：负责与 FAdapter 交互，将交易系统 F 中的交易信息发送到 BCService,并将 BCService 返回的清算结果返回到系统 F 中，同时还要负责管理和监控 FAdapter 的状态；

SystemAlertService：监听系统的异常事件和各种警告，并将其转发到 SendMailService；

DBService：负责与 JDBCAdapter 交互，让系统中其他模块可以存取数据库，同时还要负责管理和监控 JDBCAdapter 的状态；

SendMailService：监听系统中的邮件发送请求，并将其转发到 SendMailAdapter，同时还要负责管理和监控 SendMailAdapter 的状态；

IAFSatusManager：向各个 Adapter 的 Service 提供管理和监控 Adapter 的接口；

LoggingManager：向各个模块提供统一的打日志（log）的接口；

PasswordService：负责与 PwMatrixAdapter 交互，提供接口为其他模块获取密码，同时还要负责管理和监控 PwMatrixAdapter 的状态；

BMService：负责与 BMAdapter 交互，将 BCService 发过来的清算事件转发到 BMAdapter 中，并将 Adapter 返回的结果返回给 BCService，同时还要负责管理和监控 BMAdapter 的状态；

BCService：负责接受 LService 和 FService 发送过来的原始交易信息，然后根据它们生成出系统 B 所要求的信息格式并转发到 BMService 中。同时监听 BMService 的返回信息，将系统 F 的结果返回至 FAdpater，将有错误的结果发送给 SendMailService；

AppManager：统一管理所有的模块，包括启动、关闭和监听状态等，并向外部提供调用接口，使得外部可通过调用特定脚本来控制整个系统。

图 3.6 说明了 Correlator 内部各模块和事件处理的大致流程。

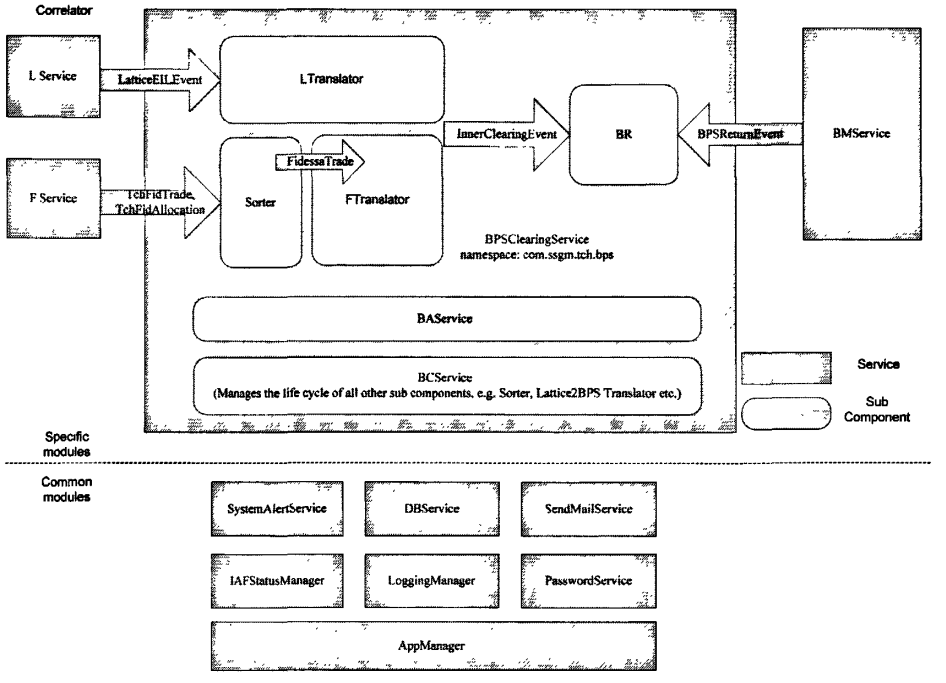

图 3.5 Correlator 内部各模块及事件处理流程

由于系统的可扩展性要求，我们将模块分为公用模块（common module）和非公用模块（specific module）两类。公用模块设计时有可复用的要求，当新的清算流程加入时，就可以以这些公用的模块为基础进行快速的开发。图 3.6 也表达了这种思想，SendMailService，DBService，SystemAlertService，IAFStatusManager，LoggingManager，PasswordService 和 AppManager 属于公用模块，为上层模块提供支持；而 LService，FService，BMService 和 BCService 属于为系统 B 清算流程所特定的模块，在公用模块的基础上完成这个特定的清算流程。当新的清算流程加入时，就可以基于公用模块搭建工作流程，而不会影响之前的流程。在具体实现中，这些模块由 Apama 所特有的 MonitorScript 语言编写实现，下面将分别介绍这些模块。

3.5.1 通过 AppManager 管理整个系统 T

AppManager 管理着系统所有的模块，主要实现三项功能：

1. 控制系统各模块的启动和关闭

Apama平台提供了事件注入API（engine_send），使得系统外部可以通过调用相应的命令向系统注入事件。AppManager根据系统外部注入的控制事件来启动和关闭系统各个模块。由于模块间存在启动和关闭的依赖关系，各模块的启动和关闭需要按一定的次序执行，AppManager负责按次序串行依次启动和关闭各模块，当任何中间模块出错，AppManager停止后续动作并发出警告，（如果SendMailService已经启动或还未关闭，警告将通过邮件发出，否则只记录到日志里）。图3.7和图3.8分别为系统各模块启动和关闭的流程。

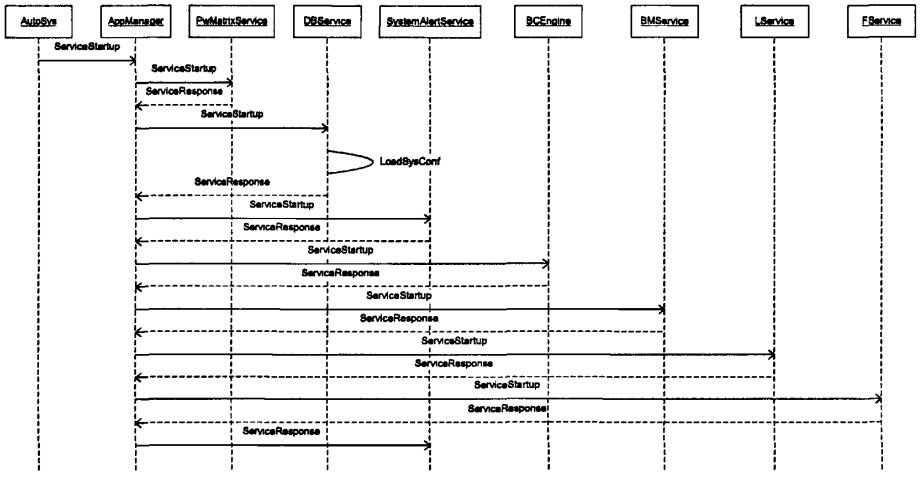

图 3.6 系统各模块启动流程

在启动过程中，外部系统Autosys通过调用脚本向Correlaotr注入启动事件，AppManager收到后首先完成自己的启动和初始化，然后发出ServiceStartup开始启动第一个模块PwMatrixService。各模块根据自身的启动情况返回相应的ServiceResponse事件。AppManager根据ServiceResponse来得知当前模块是否启动成功，来决定是否继续后续动作。由于大部分模块需要从数据库中获取相应的配置信息，所以DBService被排在启动序列里靠前的位置，而数据库连接需要密码，所以

PwMatrixService 紧接着 AppManager 启动。在启动完 PwMatrixService 和 DBService 后，由于需要尽量将启动时可能出现的错误通过邮件通知相关用户，SystemAlertService 被排在第三位。然后启动负责转换、报告、确认交易清算的核心模块 BCService。它的启动完成意味着整个系统的所有内部模块都已经准备就绪，接下来就可以开始启动外部服务。首先是 BMService，它的启动成功意味着与系统 B 的连接畅通，于是可以放心的启动前端连接，即分别启动 LService 和 FService，它们启动好后交易系统 L 和 F 的交易信息就开始流入系统，系统的清算工作开始运转，整个系统的启动完成。

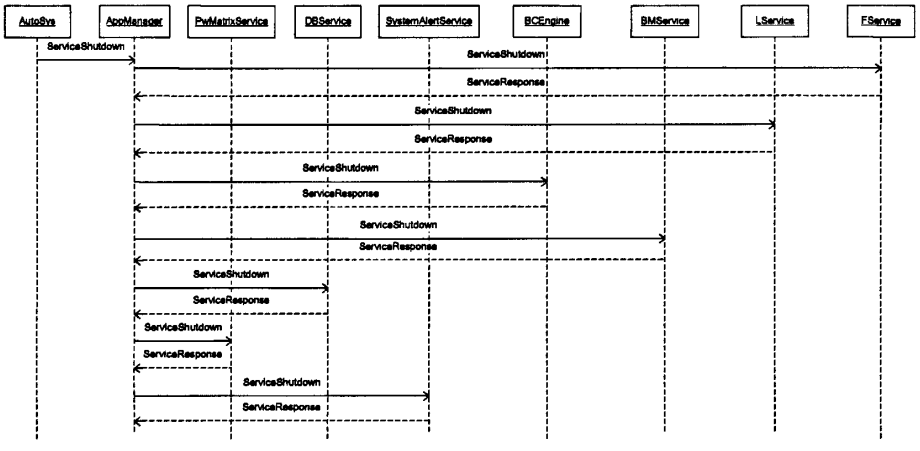

图 3.7 系统各模块关闭流程

在关闭过程中，外部系统 Autosys 通过调用脚本向 Correlaotr 注入关闭事件，AppManager 开始发出 ServiceShutdown 事件按序关闭各个模块。关闭的顺序基本上是启动顺序的逆序。首先关闭两个前台系统模块，通过关闭 FService 和 LService，使交易系统 F 和 L 的交易信息不再流入。然后关闭 BMService，使系统 B 的返回信息不再流入。此时系统与外部的连接都已经终止，可以开始安全的关闭内部模块。首先关闭的是核心模块 BCService，然后是 DBService 和 PwMatrixService。SystemAlertService

被留到最后以报告系统关闭情况和其他任何异常。最后，AppManager 完成自己的关闭，整个系统的关闭完成。

2. 启动配置信息传达

很多模块启动时需要指定一些特定的配置，如 DBService 需要数据库密码，SendMailService 需要邮件接收者名单，BCService 需要一些转换配置，BMService，FService 和 LService 需要连接信息等。在启动过程中，AppManager 就起到了配置信息传递的作用。ServiceStartup 事件上携带了当前模块所需的启动信息，而从该模块得到的供后续模块使用的信息通过 ServiceResponse 返回，AppManager 将它再放入给下一个模块的 ServiceStartup 中。如 PwMatrixService 启动后得到了数据库的密码，把它放入 ServiceResponse 中返回，接下来 AppManager 向 DBService 发送的 ServiceStartup 中就携带了该密码，使 DBService 能成功启动，同时从数据库中得到的更多配置也将由 DBService 的 ServiceResponse 返回供后续模块使用。

3. 各模块健康状态监控

在系统启动和关闭过程中，AppManager 通过监听 ServiceResponse 来得知各个模块的状态；在系统运行过程中，AppManager 通过监听 ServiceStatus 来得知各个模块运行时的状态。有些关键模块崩溃时，必须关闭整个系统并发出警告；而其他一些非关键模块崩溃时，系统可以继续运行而不需要关闭。系统内部通过维护两个向量 StartMode 和 RunMode 来表示各个模块在启动时和运行时是否属于关键模块。向量中的每个元素有两类值：乐观模式（Optimistic）表示该模块是非关键模块，它的崩溃对系统的影响是可以保持乐观的，系统可以继续运行；悲观模式（Pessimistic）表示该模块是关键模块，它的崩溃对系统的影响是让人悲观的，系统无法继续运行。AppManager 就根据这两个向量和各个模块的状态来进行异常处理。

3.5.2 BCService

BCService 是系统清算报告的核心，负责交易消息的筛选（sorting），处理（processing），报告（reporting），回复确认（acknowledgement），可持久化（persistence）和异常恢复（recovery）。由于功能较为复杂，BCService 内部又划分为多个子模块（sub component），由图 3.6 里的蓝色块表示，包括 LTranslator, Sorter, FTranslator, BR, BAService 和 BCService。

消息流程也在图 3.6 中用箭头表示。由于系统 F 中过来的交易信息有多种且不是所有种类需要进行清算，Sorter 子模块接受从 FService 过来的原始交易信息，进行筛选过虑和合并，然后才转发到 FTranslator 中去。FTranslator 和 LTranslator 对交易系统 F 和 L 的交易信息进行转换，生成系统 B 所接受的格式然后转发到 BR 中。BR 完成对清算消息的报告和回复确认，同时保证消息报告的可靠性。BAService 负责接受各子模块的异常信息并发送到 SendMailService 中去来发送警告邮件。BCService 统一管理所有子模块。

3.5.2.1 BCService

BCService 本身起到衔接内外的左右，对外与其他模块一样，接受 AppManager 的统一调度；对内起到管理作用，它与各个子模块的关系类似于 AppManager 与各个模块的关系，负责完成子模块的启动和关闭，启动配置消息传达和健康状态监控的作用。图 3.9 表示了各子模块启动关闭流程。

BCService 接受到 AppManager 的 ServiceStartup 事件后，首先启动自身并完成初始化，然后发送模块内部的 ServiceStartup（在具体实现上通过事件定义的包名的不同来与外部的 ServiceStartup 区分）依次启动子模块。子模块通过发送模块内部的 ServiceResponse 来返回启动结果。启动顺序安排的大体思想与外部模块的相似，即先启动负责监控的子模块 BAService，使可能的异常信息能及早的报告。然后启动内部子模块 Sorter，再是负责向后连接的外部子模块 BR，最后是负责向前连接的外部子模块 LTranslator 和 FTranslator。所有子模块启动完毕，BCService 返回 ServiceResponse 给 AppManager。子模块的关闭流程与

启动流程基本相反，不再累述。

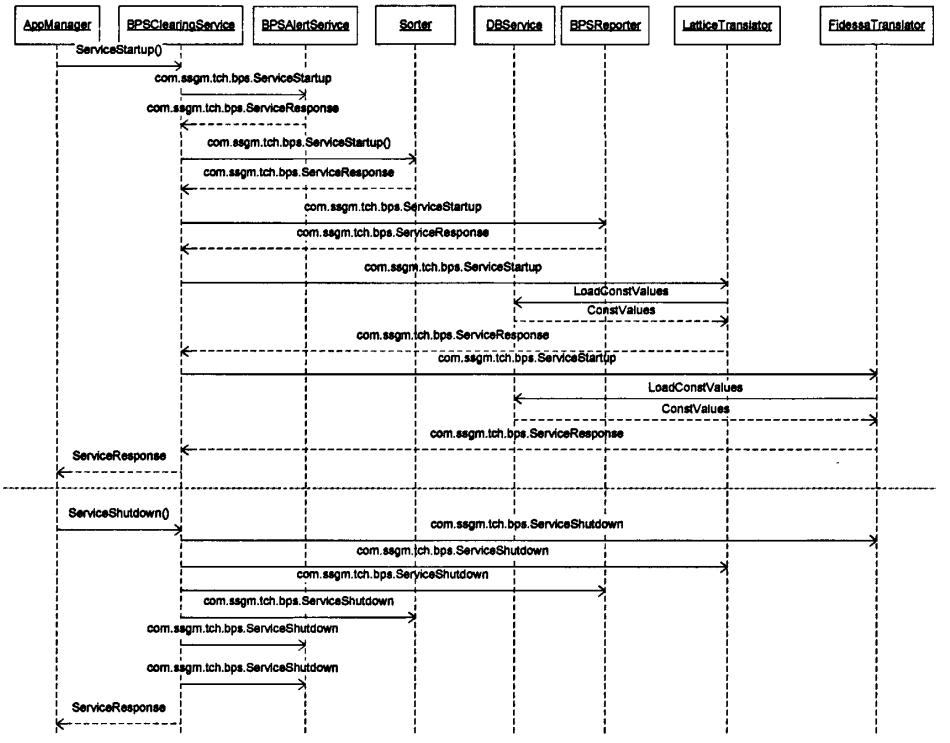

图 3.8 BCService 内部各子模块启动关闭流程

3.5.2.2 其他子模块

LTranslator，Sorter，FTranslator 和 BAService 这四个模块具有一定的共性，都是接受-转换-发送的流程，由于都与具体业务逻辑关系紧密且技术上比较简单，故不作过多介绍。

3.5.3 通过 SendMailService 保证系统可靠性

系统中各个模块里负责 Alert 的 Service(如 BCServcie 里的 BAService，整个系统级别的 SystemAlertService) 或模块本身（如 LTransactionService，负责保证载入 L 文件并报告整个事务的完整性）监听各自范围内的运行状态，当遇到异常时发送出相应的 AlertContent 事件。而 SendMailService 是工作在

SendMailAdapter之上的MonitorScript，它监听所有的系统警告AlertContent事件并把它转换为最终的Mail事件发送给SendMailAdapter。除此之外，它还负责将较短时间间隔内的多个AlertContent合并到同一封邮件中去，保证用户不会收到太多邮件。

通过这样的设计使得Mail的层次结构非常清晰，方便新Alert的加入和实现，同时不会对现有模块造成影响。图3.11表示了这3层结构。

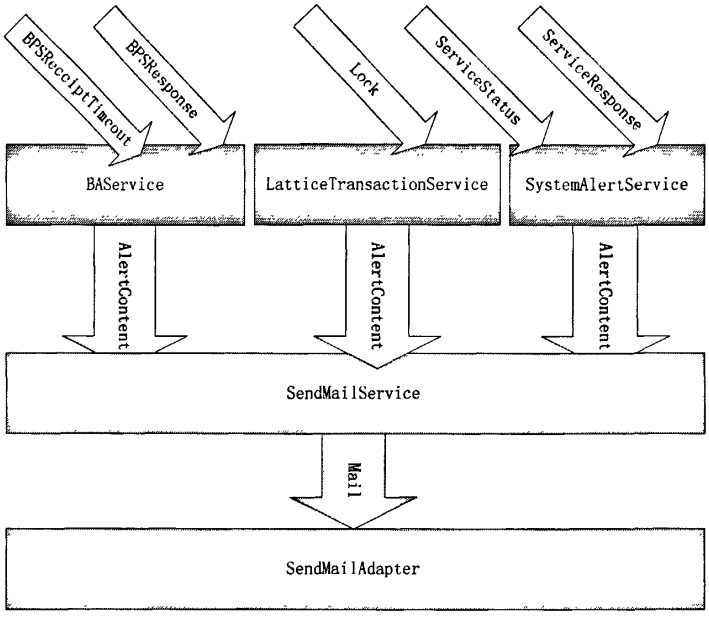

图 3.9 SendMailService与Mail的层次结构

3.5.4 通过SystemAlertService保证系统可靠性

SystemAlertService监听系统中的关键事件，若有异常就将其包装成AlertContent事件发出。含有AppHealthMonitor和LTransactionMonitor两个子模块。

3.5.4.1 AppHealthMonitor

AppHealthMonitor通过监听ServiceStatus和ServiceResponse来获得各个模块的状态信息，与AppManager一起监控和维护系统的正常运行。

3.5.4.2 LTransactionMonitor

由于从交易系统L传递过来的不是实时数据而是一个文件，系统T必须批量的载入、报告和确认回复。从一个文件载入到其中所有数据接受到回复或回复超时，就是完成了一次系统 L 的清算，我们可以将整个过程看作一个事务。LTransactionService 负责处理该事物，而 LTransactionMonitor 负责对事务过程的完成进行监控。图 3.12 表示了该事务的过程。

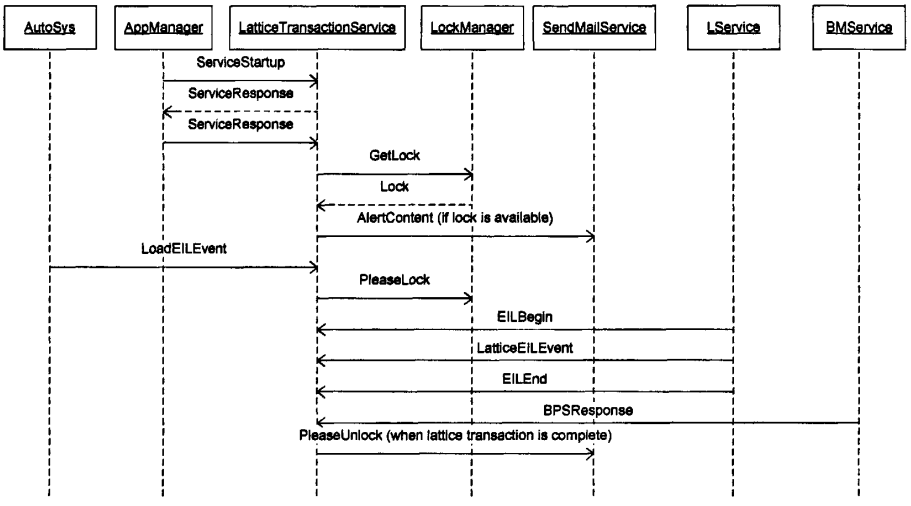

图 3.10 载入系统L文件的事务过程

LockManager 是 Apama 平台提供的一个原子锁，LTransactionService 利用它进行事务操作。当收到 Autosys 要求载入系统 L 的文件时， 它将锁锁上，表示正在有事务在进行。之后 LTransactionService 开始载入数据，LService 会依次发送这个批次的头，文件中的交易数据，批次的尾给 BCService，当全部交易报告完毕并得到回复或超时之后，此次事务过程完成，LTransactionService 再将锁清除。在中间任何一个过程系统奔溃，都不会使锁清除，所以在系统恢复重启后只需检查锁的状态就可以知道事务的完成情况。

LTransactionMonitor 的监控检查批次头是否缺失，批次内交易事件总数量

是否和头中的值一致，批次尾是否缺失，批次内的交易报告是否有遗漏等各种异常。

3.5.5 通过DBService实现可持久化

DBService和JDBCAdapter是Apama平台默认提供的对数据库的访问途径。在系统T中出现了两种存储机制，一种是通过JDBCAdapter对真正的外部数据库存取，另一种是通过称为StateStore的机制对Correlator中类似内存数据库的Store进行存取，Store里的数据实时的记录在Correlator维护的文件中。前者由于可使用SQL直接存取关系数据库，功能较强大，适合数量大和复杂程度高的数据，但使用较为复杂和不便，并需要在MonitorScript中嵌入SQL语句；后者较为轻量，适合对数量不大的简单数据存取，同时API也比较方便。

DBService对外为这两种方式提供了统一的API。图3.13表示了它的结构。DBService本身对整个模块进行管理，负责启动和关闭各个子模块，同时负责从PwMatrixService获取数据库密码。无论是在数据库中还是在StateStore中的数据，我们为每一张表都建立一个子模块，该子模块封装了对StateStore或数据库的访问细节，并提供统一的API方便调用，于是外部用户模块将不需要在自己代码中嵌入SQL语句。

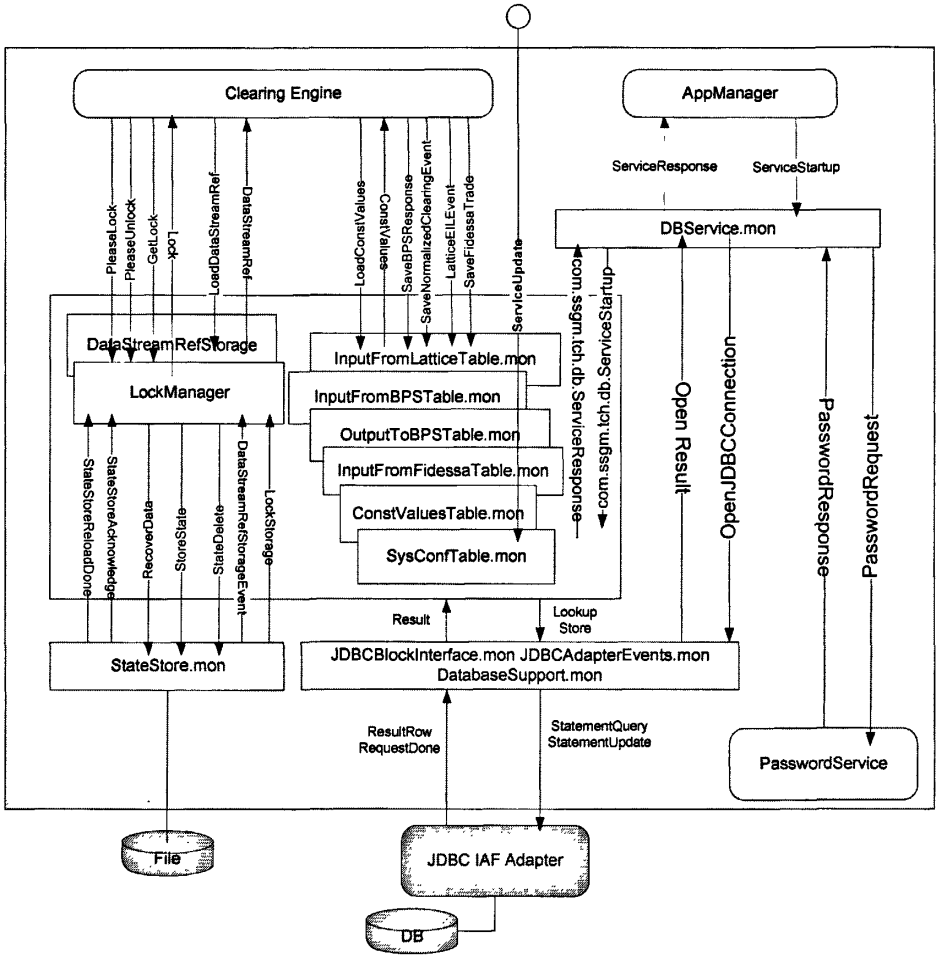

图 3.11 DBService 的结构

3.5.6 通过 IAFStatusManager 保证系统的可靠性

系统 T 遵循 Apama 平台的 IAFStatusManager 及其称为 StatusSupport 的接口来使 MonitorScript 管理 Adapter 的连接信息。每个 Adapter 对应的 Service 模块只需要增加少量的代码即可完成监控。

图 3.14 表示了 IAFStatusManager 管理 Adpater 状态的流程。各个 Service 只需要向 StatusManager 发送 SubscirbeStatus 事件订阅状态信息，就可以每隔固定时间得到最新的 Adapter 状态。IAFStatusManager 则包装了对 Adapter 的状

态查询和回复。

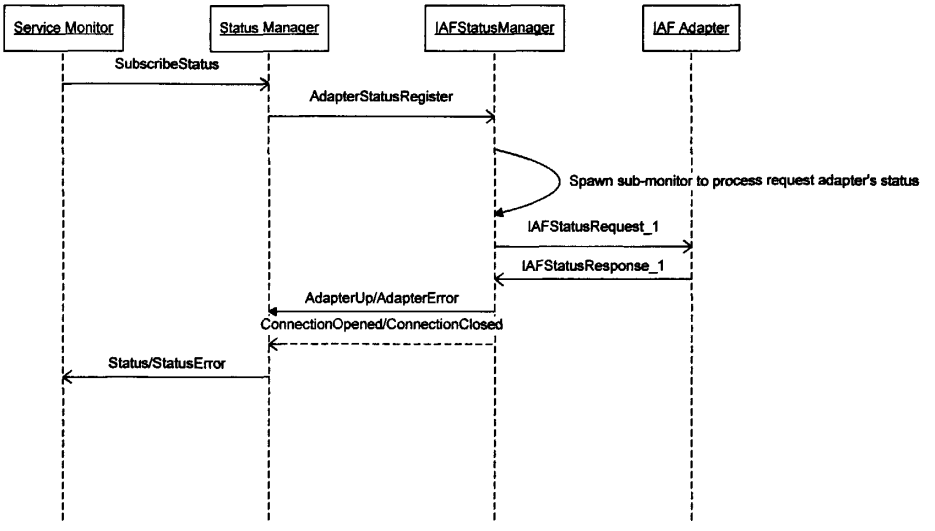

图 3.12 IAFStatusManager 管理 Adpater 状态的流程

3.5.7 LService, FService, BMService, PasswordService, 和 LoggingManager

LService, FService, BMService 和 PasswordService 与相对应的 Adapter 联系紧密，将在 3.6 节介绍 Adapter 时一起说明。LoggingManager 因功能简单不再累述。

3.6 通过 Correlator-Adapter 机制与外部交换

3.6.1 FAdapter 与系统 F 的交互

系统 F 为对外连接接口提供一套称之为 OpenAccess 的通信协议，其中包括多种消息协议，如通过 stream 方式实时获得内存数据库 RTD（real time database）变化的流协议，通过事务调用（transaction service）回写 RTD 的回写协议，通过查询调用（query service）来做简单信息查询的查询协议。而 FAdapter 的功能就是对外用 Java 实现这些协议来与系统 F 进行交互，对内用 Apama 特有的

MonitorScript 与 correlator 进行交互，最终完成系统 F 与 correlator 的交互。

同时，出于系统异常恢复和可靠性的考虑，由于 FAdapter 进程可能因为某个异常而崩溃，所以在 correlator 内部用 DataStreamRef 这一属性来保存了 FAdapter 和 F 系统之间的 trade stream 上的传送过的最后一条交易信息的时间戳，如果 FAdapter 经历异常，那么在重启之后，FAdapter 可以根据 correlator 中的 DataStreamRef 来取得下一条需要传送的交易信息，避免了交易信息的任何遗漏和重复。我们可以描述 F 系统、FAdapter 和 FService.mon 之间的事件流如图 3.15 所示：

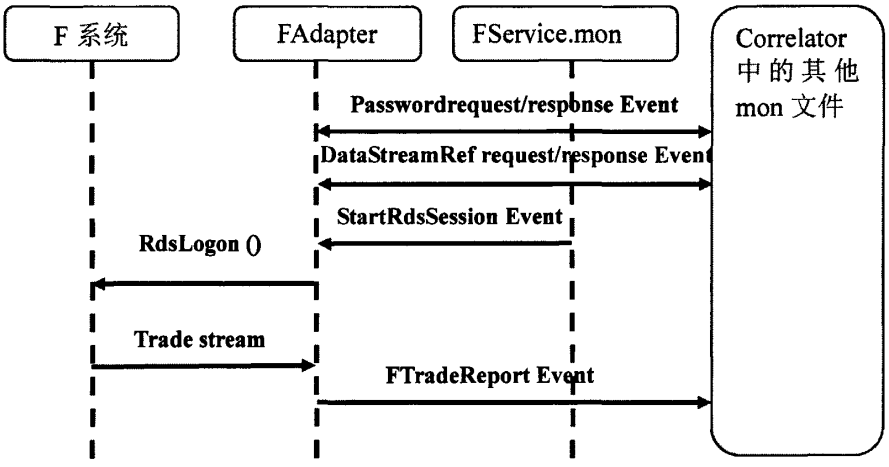

图 3.13 FAdpater 工作流程

3.6.2 LAdapter 与系统 L 的交互

LAdapter 负责读取交易系统 L 产生的文件，然后进行解析并传送给 correlator。其主要功能包括受 correlator 触发后载入文件，保证载入事务的完整性，和报告载入异常。LAdapter 和 LService，LTransactionService 一起保证系统 L 交易报告事务的完整性，3.8 节将做详细介绍。

3.6.3 BMAdapter 与系统 B 的交互

清算系统 B 对外提供 IBM Websphere MQ 的连接方式，所以系统 T 需要将清

算信息发送到MQ服务器上，而BMAdapter就实现了这个功能。其具体功能包括：

1. **信息组装/拆分和双向传递**

   BMAdapter需要将Correlator过来的信息发送至MQ Server，同时需要将MQ Server上系统B的返回信息发送至Correlator。由于系统B需要特定格式的MQ消息，在送出和返回的过程中，BMAdapter还要分别负责消息的组装和拆分的任务。当组装和拆分出现错误时，通知Correlator。

2. **按需建立连接**

   系统T和系统B间存在着两个级别的连接：MQ级别的连接，即BMAdapter到银行S的内部MQ Server再到系统B的MQ Server的连接；业务逻辑上的连接，即系统B的清算流程是否在运作。BMAdapter在系统启动时，首先建立MQ级别的连接，然后通过发送SOD（start of day）消息打开业务逻辑上的连接，当得到系统B对SOD的正确回复后，连接建立完毕。

3. **监控连接并报告状态**

   系统T和系统B间的协议包含有HRBT（heart beat）消息机制，即系统B会对非交易消息HRBT给予即时回复。通过发送HRBT和监听HRBT回复，就可以完成对两个级别连接的监控。当HRBT回复丢失或超时时，BMAdapter将向correlator发送警告。

4. **填写MQ Header**

   系统B接受到MQ消息后，根据MQ Header决定返回地址，另MQ Header还包括消息优先级，消息持久化等配置。BMAdapter负责为每一条消息生成合适的MQ Header。

5. **保证双向高吞吐量**

   由于BMAdapter是整个系统T里消息流量最大的节点，所以必须保证双向的高吞吐量。在实现上通过多线程、缓存消息然后批量传送的方式保证了150消息/秒的双向高吞吐量。

6. **流控制**

   由于系统B的处理速度不稳定，且系统T与系统B之间的连接经过多重

网络结构，BMAdapter参与了流控制的工作，3.7节将详细介绍。

图3.16简单的表示了BMAdapter工作流程。

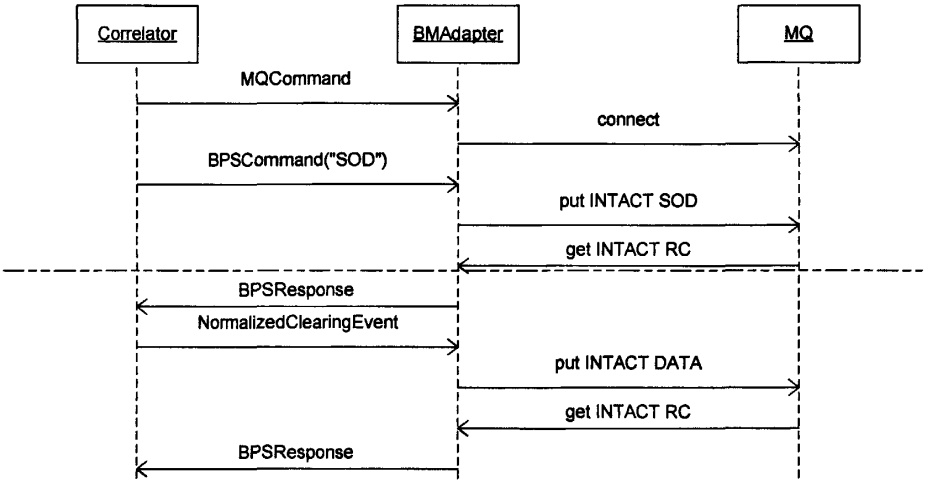

图 3.14 BMAdapter工作流程

3.6.4 EmailAdapter，JDBCAdapter，ScriptAdapter和PwMatrixAdapter

EmailAdapter，JDBCAdapter和PwMatrixAdapter是Apama平台捆绑自带的Adapter，ScriptAdapter结构简单，均不再详细介绍。

3.7 性能与流控制

交易系统F作为银行S的主要交易系统，交易日的每天都会有数百万的交易信息产生，其中给系统T的stream上的信息数量也在百万级别。系统T需要保证在如此大数据量下系统能正常工作。另外由于两端系统的吞吐速度（throughput）相差较大，系统F的stream能以峰值约为1000条/秒的速度输出交易信息，而系统B最多只能以200条/秒的速度接受信息，处于两者之间的系统T也就担当了流控制的责任。

为了完成这项要求，系统中任何有大量数据流经的节点（如BMAdapter和

FAdapter），都对外提供 PleaseSlowdown 和 PleaseSpeedup 事件接口，并在 AppManager 中注册为流量敏感节点。然后可将这些流量敏感节点按其对事件数据是产生还是消费分为生产节点和消费节点。需要注意的是由于消费和生产是相对系统T自己而言的，所以同一个节点可以即是生产节点也是消费节点，即在处理不同流向的消息时节点角色也不同，如 BMAdapter 在发送清算消息时是消费节点，而在接受返回消息时是生产节点。为了降低模块间的依赖，各消费节点和生产节点相互透明，由 AppManager 统一管理。

当系统运行时，消费节点负责定期检查外界系统（消费者）的消费情况，当发现消费速度太慢导致消息事件堆积到一定阀值时，消费节点发出 PleaseSlowdown 事件，请求降低系统生产速度。由于消费节点并不知道谁是生产节点，所以由 AppManager 负责监听这个消息并向生产节点发出 PleaseSlowdown 事件，生产节点收到该消息后减缓或停止从外部系统接受事件消息，于是，慢速消费者可以有足够时间消费事件消息，在这个过程中新的事件消息不会进入系统造成堵塞。当消费节点发现堆积的事件已经全部消耗完或低到某一个阀值时，发出 PleaseSpeedup 事件，AppManager 收到后将其转发给相应的生产节点，生产节点加快或恢复从外部系统接受事件消息，系统重新开始高速运转。

以 BMAdapter 和 FAdapter 为例，在发送清算消息时它是消费节点。每隔 10 秒它通过 IBM Webshphere MQ 的 API 检查 MQ Server 上消息队列的长度，当其当前长度与最大长度的比值高于设定的警戒阀值（80%）时，说明系统B的消费过慢，BMAdapter 向 AppManager 发出 PleaseSlowdown 请求，AppManager 转发至 FAdapter，后者停止从系统F中接受消息。于是系统中已有的消息逐渐减少，当消息队列当前的长度低于最大长度的 20%时，BMAdapter 发出 PleaseSpeedup 事件，经由 AppManager 传送到 FAdapter。FAdapter 重新开始接受系统 F 的消息，系统T的处理恢复正常。

3.8 异常恢复与可靠性

由于系统T包含交易系统L和F的清算报告，所以在异常恢复和清算可靠性

上也要分两部分考虑。

3.8.1 系统L交易事务完整性保证

当系统异常终止，可能有些事务还没有完成（如系统 L 的文件正在载入），这样会导致有些交易报告丢失，系统T必须探测到这种情况并通知用户。所以在系统重启时，LTransactionService 必须在 StateStore 检查锁是否锁上，如果锁上则说明上次事务没有完成系统就关闭了。

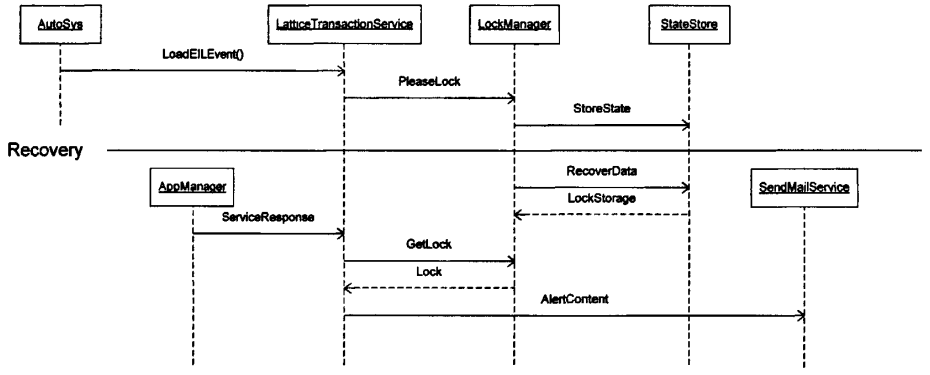

图 3.15 L交易事务处理流程

在每次收到要求开始载入系统 L 文件时，LTransactionService 通过 LockStorage 在 StateStore 中写入锁；如果文件中每一条交易报告都收到回复，LTransactionService 再把锁清除。所以任何时间点上的系统崩溃都不会影响到事务完成状态的记录。另外系统也提供了外部接口，让用户可以手动清除锁。图 3.17 表示了L交易事务处理流程。

3.8.2 系统F交易报告可靠性

由于系统B可以检测出重复的报告并拒绝，系统F交易报告的可靠性可基于交易报告的时间戳来保证。BR 保存了被系统 B 接受的最后一条的时间戳，通过 StateStore 写入文件保证持久化。如果系统异常崩溃，则重启后系统根据保存的时间戳从系统F中继续请求交易信息，这样就保证了交易信息不会丢失。图 3.18

表示了系统F交易报告可靠性保证的流程。

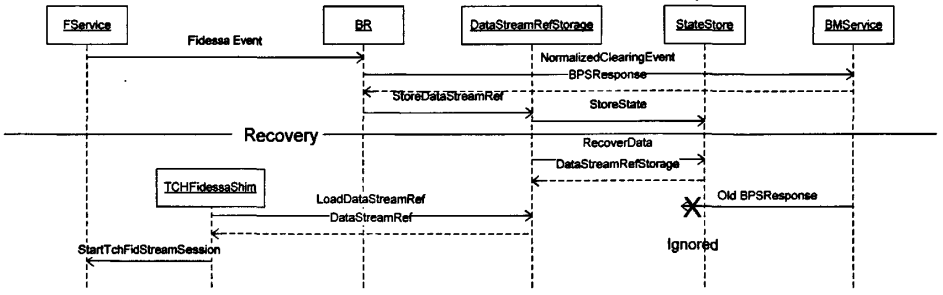

图 3.16 系统F交易报告可靠性保证

3.9 本章小结

在本章中，首先介绍了项目背景，分析了系统T的功能需求，以及外部周边与之进行交互的其他系统。然后，基于事件驱动架构设计了系统T的整体结构，主要包含各个Adapter和事件处理引擎Correlator之间的联系和事件流的解析。然后详细介绍了Correlator内部几个重要模块的设计，和几个主要Adapter的设计。最后说明了系统如何满足性能和流控制要求，以及在异常恢复中保证清算报告的可靠性。

第4章 可持久化事件处理

在金融系统中，信息的可靠性往往是一个重要的要求，在实现系统T的过程中，我们发现现存的复杂事件处理系统没有完善的机制保证信息的可靠性，本章研究了通用的可持久化事件处理方案，从而可以实现信息的可靠性。

4.1 现阶段事件驱动架构的不足

现代企业信息系统已经被广泛使用于各行各业，每刻都产生着大量的数据。在这海量的数据背后蕴藏着丰富的有价值的信息，需要有相应的机制和系统来发现和提取。在传统的请求/处理机制中，用户所需信息需要靠其主动查询，故不能提供及时的反应性[27]，也不能提供松耦合的通信模型。而信息系统即使获得了这些数据，也缺少有效的机制来提取数据之间蕴含的丰富信息[28]。

复杂事件处理机制（Complex Event Processing, CEP）[29, 30]则是一种有效的办法。它从大量的数据中抽象出原始事件，并通过一定的事件操作符将不同的事件关联起来形成复合事件，表示新的意义，揭示数据之间隐含的信息。这些事件，特别是复合事件快速地传送到关注它的用户或者系统那里，以便他们进行决策，提高企业的响应能力，这就是事件驱动的思想[28]。事件驱动体系结构上是对反应式系统的抽象，通过“推”模式的事件通信提供并发的反应式处理，特别适合松耦合通信和支持感知的应用，是解决SOA松耦合通信与协同处理的理想解决方案[31]。

事件驱动体系结构也存在缺点，例如系统设计复杂、可理解性差。另外系统设计时面临高并发通信和处理的性能挑战[31]。另一个重要的缺点就是缺少平台级的可持久化能力。由于事件处理在工程应用中还是个新生事物，没有许多现存的请求/处理平台所具有的较完善的异常恢复、数据可持久化的机制。在企业应用中，基于CEP平台（如Progress公司的Apama平台等）的开发人员往往需要自己来实现系统的异常可恢复性和事件数据可持久化，大大的增加了开发代价。本章则提供了基于现有CEP平台的一种较通用的增强方案，可实现可持久化的事件

处理平台。

本章下文将首先分析了两类事件处理平台的模型，抽象提取出可广泛适用的处理过程描述。然后重点针对较复杂的非实时输出情况提出了两类可持久化的解决方法，并结合实例给出了通用的处理过程描述。

4.2 两类系统的分析

依据输入输出事件是否为实时，可以将此处理平台分为4大类，分别为实时输入实时输出、实时输入非实时输出、非实时输入实时输出和非实时输入非实时输出。从系统的可恢复性角度来看，因为非实时的某时刻批量输入可被看作该时刻瞬间大量的实时输入，所以实时输入和非实时输入可以归为同一类，不需要分情况讨论。故下文只以输出是否为实时为标准分两类来分析系统的模型。由于非实时输出的平台往往要求系统自身维护事件状态信息，导致复杂度和可维护难度大大高于实时输出的相应系统，所以将重点表述了非实时输出的情况。

4.2.1 非实时输出系统

非实时输出的处理平台的最重要特点是需要将输入事件的相关状态保存在自身的缓存中。逻辑相关的事件将拥有同一个状态，当新的相关事件到来时，状态将被新建、更新或删除。直到特定的触发发生（例如是一个定时器或一个状态条件的满足，或者由外部程序调用输出命令），才将缓存里保存的事件以当前状态输出。

我们可以抽象的描述非实时输出平台事件处理过程如下：

系统输入为离散的事件流 E1, E2...En；

对于相关联的事件 Ei, Ej...Em($1 \leqslant i<j<m \leqslant n$)，系统维护相应的状态信息于 Sp 中，所有状态信息可构成状态集合{Sq}($1 \leqslant q \leqslant maxQ$)，一般来说 $maxQ \leqslant n$；

对于每个新输入的 Ei，系统将作出下列三种动作之一：

(1) 若{Sq}中没有一个 Ei 对应的状态，那么一个新状态 SmaxQ+1 将产生，并且根据 Ei 将其初始化，然后增加到{Sq}中；

(2) 若{Sq}中已有 Ei 对应的状态 Sp 且 Ei 是要求状态更新的事件，则 Sp 由

Ei 中的信息更新；

(3) 若{Sq}中已有 Ei 对应的状态 Sp 且 Ei 是要求状态删除的事件，则 Sp 将从集合{Sq}中删除；

当接受到输出指令时，系统将根据当前时刻{Sq}集合中的信息，产生 O1, O2...Oq 的输出。

4.2.2 一个实例

以下即为一个典型的实时输入非实时输出的事件处理平台。此平台为一家美国托管银行基于复杂事件处理系统 Apama 开发的交易清算系统。平台通过 OpenAccess 协议，从交易系统 F 中获得实时的交易事件，并维护于系统的内存中。当接收到输出指令时，根据系统中的所有交易事件的最新状态产生一个清算请求文件并发送至下游清算代理机构的服务器上，从而完成一次输入输出过程。

该系统的输入交易事件有 3 类，分别为新建、修改和删除，即有 Ei∈{Enew, Eamendment, Ecancel}。系统维护的交易状态也是新建、修改和删除，即 Sq∈{Snew, Samendment, Scancel}。如图 4.1 所示，交易状态的转移关系可用类似于有穷自动机的状态转移图来表示。

4.2.3 实时输出系统

实时输出系统自身不需要保持事件状态信息，只需要将输入按一定的规则转换成输出即可，相当于一个没有状态集合{Sq}的非实时输出系统，故不做累述。

4.3 系统恢复的实现

系统的恢复实际上包含两类恢复。第一类是系统崩溃，重新启动后如何恢复到与之前最后的正常情况一致的状态；第二种是系统的输出由于某种原因丢失，如何重新产生它。对于上述的两类不同系统，在系统恢复的角度也有着不同的考虑。由于非实时输出系统的恢复实现更加复杂，下文将重点讨论这种情况。

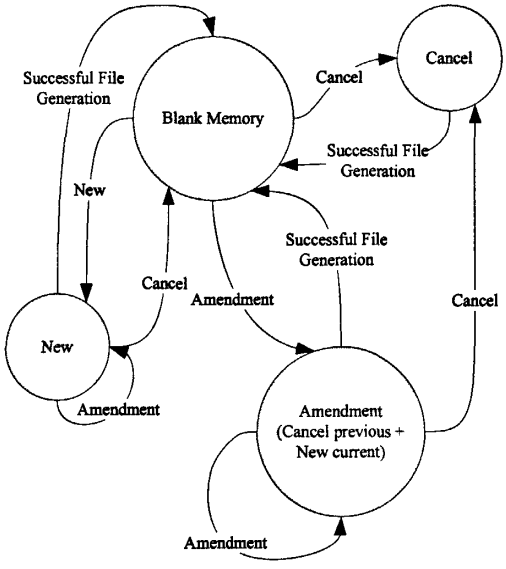

图 4.1 状态转移图

4.3.1 非实时输出系统的恢复

4.3.1.1 问题分析

图 4.2 为一个实时输入非实时输出系统，每一次触发系统输出（Trigger i），系统都会产生一个输出（Output i），这个输出可能是一个文件的产生或是一个数据库中表的更新。对于第一类恢复，即当系统意外崩溃后，我们希望重启系统能使其恢复到图中 Present 时刻的状态，其中恢复的关键是重新得到 Present 时刻的状态集合{Sq}p。

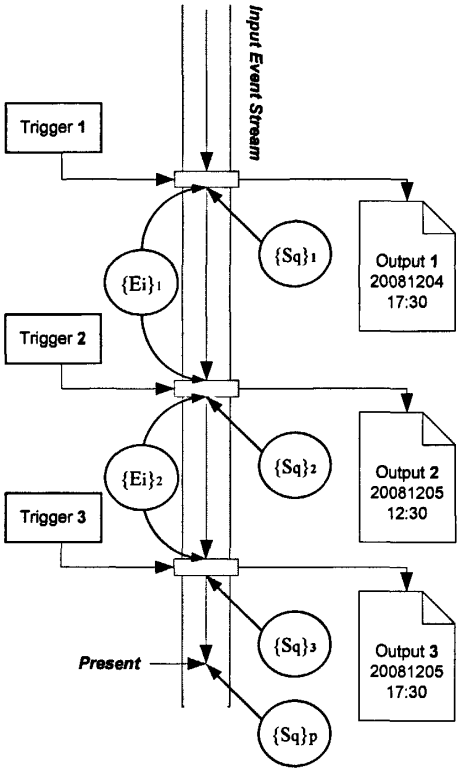

图 4.2 非实时输出系统的处理流程

第二类恢复则是在异常情况下，如何重新产生丢失的输出。不妨假设丢失的输出是 Output C，显然 C 的产生只由两个因素决定：Trigger 2 后系统保存的状态集合{Sq}2，以及 Trigger2 与 Trigger3 之间的输入事件。所以要恢复 Output C，只需要恢复 Trigger 2 后的状态集合{Sq}2 和 Trigger 2 至 Trigger 3 之间的所有输入事件{Ei}2 即可。

下面将提供两种解决方法并提供比较。

4.3.1.2 第一类解决方法

若系统的输入源是可持久化的其他系统（比如 JMS 的 Topic），并且它产生的每个事件都有唯一的时间戳，则可以利用其来实现本系统的持久化。可广泛适用的表述实现过程如下：

(1) 系统正常启动，运行；

(2) 记录下每次输出触发时输入事件的时间戳；

(3) 系统异常崩溃，一个或多个输出丢失，设最后一个丢失的输出为Output n，由Trigger n触发；

(4) 以恢复模式重启系统，系统以一个足够早的时间戳去请求事件源重新发送事件；

(5) 输入事件重新从输入源流入系统，系统检查每个流入事件的时间戳，一旦与原先记录的时间戳相吻合，则自动触发输出。

此方法的实质就是在短时间内重新进行一次系统处理：从重新接受输入，重新新建、更新和删除事件状态，到重新触发输出、产生输出，于是原有的输出将被依次恢复。系统也就能恢复到最新的Present状态。

上述解决方法的优点是实现简单，把保存事件的工作交由外部系统（输入源）来做，平台自身仅仅需要维护每次输出触发时的时间戳即可。但此方法也存在如下问题：

(1) 对外部系统有依赖。要求输入源为可持久化的系统，如果输入源发生不可恢复的系统崩溃，则本系统亦无法进行恢复；

(2) 在恢复过程的第四步中，“足够早”的时间戳实际中不好确定。如果不够早的话，可能导致某些状态信息的丢失以至于输出不正确；如果太早的话，大量的输入会影响整个流程中各个系统的正常处理；

(3) 由于恢复时要求输入事件短时间内重新流入，若输入事件的数量较大且输入源同时被多个系统共享，则此系统的恢复将对输入源和其他共用输入源的系统的即时吞吐量造成影响，在实时性要求较高的情况下，该解决方法将无法满足要求；

(4) 若系统的输入来自多个输入源，并且不同类输入事件都有同一类输出，则所有事件都要共用同一状态信息集合。此时若依然采用此方法进行恢复，则复杂度和对周边相关系统实时性的影响将大大增加。并且任何一个输入源的恢复失败，将导致整个系统的恢复失败。

(5) 恢复任意一个或几个丢失的输出，需要恢复最近所有输出，所以此方法在性能/系统开销（包括空间和时间）比上来说不是一个很好的办法。

4.3.1.3 第二类解决方法

经分析，第一类解决方法产生各种局限性的根本原因是对外部系统存在依赖。由前文问题分析可知，我们恢复某一个输出，只需要知道前一个输出触发时系统的状态{Sq}和前一个触发与要恢复的输出所对应的触发之间的所有输入事件即可。若我们能保存这两个关键信息，则无需外部输入源的重新输入，同样可以恢复需要的输出。由此我们提出第二类恢复实现方法。可广泛适用的描述实现过程如下：

(1) 系统正常启动，运行；

(2) 记录下每个输入事件，同时记录下每次输出触发的时间戳和当时系统的状态集合{Sq}m；

(3) 系统异常崩溃，一个或多个输出丢失，设最后一个丢失的输出为Output n，由Trigger n触发；

(4) 以恢复模式重启系统，从恢复记录中载入 Trigger i 时的状态集合{Sq}i，触发输出即可。

(5) 清空状态集合，从恢复记录中载入最后一次 Trigger n 时的状态集合{Sq}n，再从恢复记录中把Trigger n之后的输入事件按其时间戳依次导入系统，全部导入完后，系统也就重新恢复到Present状态。

这种恢复实现虽比第一种方法较为复杂，但完全解决了第一种方法的各种缺点，不再对输入系统存在依赖，恢复时也不会对其他系统的造成影响，重新恢复一个或部分输出的代价也很小，故其比较适合于系统实时吞吐量要求高、多输入同时存在等较复杂的应用情况。

4.3.2 实时输出系统的恢复

由于没有状态集合{Sq}，实时系统的恢复相对简单很多，若输入源为可持久化系统，则只需要记录最新的事件的时间戳即可；否则则可采用类似上述第二类

方法的方式，保存每条输入事件，即可实现恢复。

4.4 本章小结

通过对现行复杂事件处理平台处理流程的分析和抽象，本章提出了两种较通用的增强方案，实现了可恢复、可持久化的事件处理平台。方法一简单易行，但存在对上游输入系统的依赖和恢复时对周边系统的影响，改进之后的方法二独立可靠，可广泛适用于现行大部分复杂事件处理系统。该方法已成功应用于本文所实现的实时事件处理系统中，取得了良好的效果。

第5章 总结

在设计和实现具有异构、分布和松耦合特点的系统时，传统的系统架构由于自身设计的缺点使其难以满足此系统的需要。而事件驱动架构作为一种新型的软件系统架构，非常适合应用在具有松耦合的服务和软件组件的系统中。如果能把这类系统中不同组件间的交互抽象为系统可以识别的事件，则应用事件驱动架构的系统会有较好的对抽象后事件的产生、监听和处理能力，有别于传统的以数据库为中心的数据处理方式的事件处理方式将在这种架构中得到应用。

具体到项目而言，交易清算系统 T 由于要连接 7 个不同的异构系统，同时又要保证松耦合和可扩展性，传统的系统架构无法满足它的需求。于是，本文以系统 T 作为研究的对象，尝试着用基于事件驱动的架构来具体的实现该系统。

为了更加良好全面的设计出需求的系统 T，本文深入分析了事件驱动架构的各种特性，以及这种架构的构成。另外，由于 Apama 和 MonitorScript 是用于实现系统 T 的事件处理平台和事件编程语言，所以本文也分别对这两者就其与系统 T 具体实现较为相关的方面进行了简单的介绍。

根据前面的理论背景和实际的项目背景，结合外部周边与之进行交互的其他系统的关系，文章基于事件驱动架构设计了系统 T 的整体结构，主要包含各个 Adapter 和事件处理引擎 Correlator 之间的联系和事件流的解析。然后详细介绍了 Correlator 内部几个重要模块的设计，和几个主要 Adapter 的设计。

交易清算系统 T 作为金融系统有其特殊的要求，所以在设计时除了要实现业务功能方面的需求，还必须要保证一些非功能方面的需求：如大数据量下系统的性能和稳定性，如何进行流控制，在系统异常崩溃后如何保证报告的可靠性。所以在说明了系统 T 在业务功能的模块后，介绍了系统如何做到高性能和流控制，以及异常恢复和保证可靠性。

最后，针对现有复杂事件处理平台缺乏可持久化能力的状况，通过对现行复杂事件处理平台处理流程的分析和抽象，文章提出了两种较通用的增强方案，实

现了可恢复、可持久化的事件处理平台。

总而言之，事件处理的企业级应用还处于发展初期，而作为一种新型的系统架构，事件驱动架构受到了很多的关注，它代表着分散松耦合系统的架构的未来发展趋势。将这种架构运用在交易清算系统的实现中具有不小的实际价值。不过目前事件驱动架构还没有成熟的标准，这使得其在灵活使用的同时，也可能存在无法控制的一面。所以在实际应用中，应尽可能全面的衡量采用它时的各方面利弊，一旦决定使用，则应尽量提高其规范性和可控性，在使用之前进行良好充分的系统整体设计规划，为未来的扩展和改进留下足够合适的空间。

参考文献

[1] Brenda M.Michelson. Event-Driven Architecture Overview. Sr.VP and Sr. Consultant, Patricia Seybold Group,2006,2

[2] Bertrand Meyer. The Power of Abstraction, Reuse, and Simplicity: An Object-Oriented Library for Event-Driven Design. ISBN 978-3-540-21366-6

[3] Carol Sliwa. Event-Driven Architecture poised for wide adoption. Computerworld, 2003,5

[4] John Bates. Apama Technology White Paper. http://www.progress.com

[5] Stephen Ferg. Event-Driven Programming: Introduction, Tutorial, History. Version 0.2,2006,2

[6] Chung-Sheng Li. Real-time event driven architecture for activity monitoring and early warning. Emerging Information Technology Conference, 2005

[7] Brad Bailey. Confusion about Event Processing (CEP): Overview of CEP in the Capital Markets. Aite Group,2007,2

[8] Opher Etzion. Towards an Event-Driven Architecture: An Infrastructure for Event Processing Position Paper. ISBN 978-3-540-29922-6

[9] David C.Luckham. The Power of Events, Introduction to Complex Event Processing. AddisonWesley,2002

[10] Opher Etzion. A Subjective Tour of Event Processing. First Event Processing Symposium,2006,3

[11] Bobby Woolf. Event-Driven Architecture and Service-Oriented Architecture. IBM Corporation, 2006

[12] 王园,吉国力,苏秦. 事件驱动体系结构在 GIS 中的设计实现.厦门大学学报, 2005,7(4): 34-36

[13] Gregor Hohpe. Programming Without a Call Stack Event-Driven Architectures, http://www.eaipatterns.com

[14] George S.Fishman. Discrete-Event Simulation-Modeling, Programming, and Analysis, Springer, ISBN 0-387-95160-1

[15] David C.Luckham, Brian Frasca. Complex Event Processing in Distributed Systems Stanford University Technical Report CSL-TR-98-754,1998,3

[16] Orlando FL. Real Time Agility through Event Processing and Business Activity Monitoring. Gartner Event Processing Summit,2007,9:19-21

[17] N.F.Maxemehuk, D.H.Shur.Multicast. System for the Stork Market. ACM Transactions on Computer Systems (TOCS),2001,8(19-3)

[18] Eugene Wu, Yanlei Diao, Shariq Rizvi. High-Performance Complex Event Processing over Streams. SIGMOD,2006,6:27-29

[19] Faison Ted. Event-Based Programming: Taking Events to the Limit. Apress. ISBN 1-59059-643-9,2006

[20] Progress Corporation. Progress Apama Event Manager MonitorScript. Release 3.0.3, 2008,1

[21] 高波涌,宋宇宁. 基于SOA的事件驱动型金融业EAI技术研究.计算机工程与设计,2006(7): 83-85

[22] Mark Palmer. Event Stream Processing, a New Computing Physics of Software. DM Direct,2005,7

[23] David C.Luckham, James Vera. An event-based architecture definition language. IEEE Transactions on Software Engineering,1995,9

[24] Global Research Partners. Event-Driven Architecture: The next big thing. Gartner Application Integration and Web Services Summit, 2004

[25] Marek Mickiewicz. Send-On-Delta Concept: An Event-Based Data Reporting Strategy. Sensors,2006, 6:49-63

[26] Jean-Pierre Casey, Karel Lannoo. The MiFID Revolution. ECMI Policy Brief,2006,11(3)

[27] Tombros D, Geppert A, Dittrich K. Semantics of reactive components in event-driven workflow execution[C]. Proceedings of the International Conference on Advanced Information Systems Engineering. Barcelona, Spain, 1997: 409-422

[28] 臧传真, 范玉顺. 基于智能物件的制造企业复杂事件处理研究[J]. 计算机集成制造系统, 2007, 13(11): 2243-2253

[29] David Luckham. The power of events: an introduction to complex event processing in distributed enterprise systems[M]. Boston, Mass., USA: Addison Wesley, 2002

[30] WANG F, LIU S, LIU P, et al. Bridging physical and virtual worlds: complex event processing for RFID data streams[C]. Proceedings of the 10th International Conference on Extending Database Technology (EDBT). Berlin, Germany, 2006：588-607

[31] 刘家红，吴泉源. 一个基于事件驱动的面向服务计算平台[J]. 计算机学报, 2008, 31(4): 588-599

攻读硕士学位期间主要的研究成果

[1] 李新玉,黄忠东. 基于 CEP 的可持久化事件处理方案. 计算机应用与软件,(录用还未发表)

[2] 彭成斌,李新玉,安利峰. 一种倒计时交通灯的控制算法. 计算机工程,2009,(35)

致谢

春华秋实、时光飞逝，两年半的求是园的研究生生活行将结束。在这段平凡而充实的日子里，有很多事值得我常常追忆，又有很多人值得我心存感激。

首先，我要向我的导师黄忠东老师致以最诚挚的谢意。感谢黄老师为我创造了良好的学习、科研和实习环境，而这一切是我今天能够满怀收获喜悦的源泉。同时，在学习、生活中黄老师也给了我教诲和指导，令我受益匪浅。黄老师严谨的治学态度，平易近人的处世风范，周密务实的工作作风给了我深深的印象，导师将是我今后事业和生活中永远的楷模。

感谢同门师兄辛晶艺，是你在相关领域的开拓探索指引了我了学习研究的方向，感谢你的无私帮助。

感谢同项目组的同事，在我进入项目组的时候，你们所给予的无私帮助，使我能很好地融入这个集体。和你们一起工作的一年时间里我收获了很多：编程技能、项目经验、团队精神、还有充实快乐的感觉。我能完成今天的论文，很多的研究都是基于你们之前所做的一切。

还有我的室友：姚磊、李昕和林建智。在 25 舍 227 一起走过的日子永远值得我去回忆。

最后，是我的父母，感谢你们在我做出每一个选择的时候，总有你们在我的身边。

李新玉

2010. 1. 20 于求是园