摘　要

Web 服务是一个崭新的分布式计算模型，它的出现解决了企业之间及企业内部异构系统之间的互操作和互通信的问题。事务则是一种保证应用一致可靠的有效机制。Web 服务松偶合的特点使得 Web 服务事务尤为重要。Web 服务事务，尤其是组合服务事务对服务质量（QoS）和服务计算的一致性和可靠性起着重要作用。

本文针对组合服务事务的长事务特点，提出了一个基于域的组合服务事务嵌套模型（SBET），通过对组合服务事务进行进一步的事务划分，将组合服务平坦的事务模型扩展为嵌套的事务模型，使得事务撤销时，可以选择性的回退到某一个一致点，以保证事务的语义一致性。为了保证服务质量，在 SBET 的基础上提出一个支持 QoS 约束的组合服务事务恢复算法。该算法能将补偿的范围控制在底层的域内，减少补偿代价，保证服务质量在用户可接受的范围内接近最优，同时保证事务的语义原子性。为了提高组合服务的并发度并保证并发的正确性，提出一个基于冲突概率的组合服务事务并发控制算法。该算法充分考虑了组合服务事务中的各种并发情况，引入语义单元、冲突类锁和临界区的概念来保证事务调度的语义可串行性，同时通过调度的优先级规则来避免死锁。最后，针对当前组合服务执行引擎大都缺乏对组合服务事务的有力支持，设计了一个支持组合服务事务的原型系统（TCWS）。该系统基于 WS-Transaction 规范，在协调各成员服务的事务行为的同时，还提供了错误恢复和并发控制的功能。

本文主要在组合服务事务模型、错误恢复、并发控制及支持事务的组合服务原型系统等方面进行研究，为组合服务事务的研究提供了新的思路和方法。

关键词　组合服务，组合服务事务，错误恢复，并发控制，QoS

ABSTRACT

Web service is a new distributed computing model, which solves problems of inter-collaboration and inter-communication between heterogeneous systems from different enterprises or one enterprise. Transaction is an efficient mechanism to insure consistency and reliability of the application. Web service transaction is especially importance for the characteristic of relaxed coupling. Web service transaction, especially composite service transaction plays an important role in quality of service (QoS) and consistency and reliability for service computing.

Composite service transaction is long transaction, so this paper presents a scope based embedded transaction model of composite service (SBET), which is extended from flat model of composite service transaction through further transaction partition. When transaction is cancelled, users can rollback to a certain consistent point selectively to insure semantic consistency. In order to guarantee QoS, a transaction recover algorithm supporting QoS restriction is presented, based on SBET of composite service. The algorithm can keep the area of compensation in a bottom scope, reduce the cost of compensation, insure QoS approaching to the best when transaction recovering, and can also insure semantic atomic of transaction. In order to enhance concurrency and insure correctness, a concurrency control algorithm of composite service transaction based on probability of conflict is presented. With fully considering of each concurrency control circumstance, the algorithm introduces the concept of semantic unit, lock on conflict class and critical section to guarantee semantic serializability of transaction scheduling, and avoids dead locking by priority rule of scheduling. At last, a prototype of composite service system supporting transaction named TCWS is designed, which solves the problem that mostly existing products or prototypes for composite service executing lack transaction supporting. The prototype is based on WS-Transaction specification. Besides coordinating transactional behavior of member services, it also provides functions of error recovery and concurrency control.

Our research on transaction model of composite service, error recovery, concurrency control and composite service prototype system supporting transaction provides a new idea and a new approach for research on composite service transaction.

KEY WORDS composite service, composite service transaction, error recovery, concurrency control, quality of service

原创性声明

本人声明，所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了论文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得中南大学或其他单位的学位或证书而使用过的材料。与我共同工作的同志对本研究所作的贡献均已在论文中作了明确的说明。

作者签名：曾慧琼　　日期：2008 年 5 月 27 日

学位论文版权使用授权书

本人了解中南大学有关保留、使用学位论文的规定，即：学校有权保留学位论文并根据国家或湖南省有关部门规定送交学位论文，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以采用复印、缩印或其它手段保存学位论文。同时授权中国科学技术信息研究所将本学位论文收录到《中国学位论文全文数据库》，并通过网络向社会公众提供信息服务。

作者签名：曾慧琼　　导师签名：李建华　　日期：2008 年 5 月 27 日

第一章 绪论

1.1 引言

1.1.1 研究背景

上世纪末到本世纪初，Web 服务迅速发展起来。Web 服务是自包涵自描述性的、独立的、模块化的应用程序，具有分布、异构、自治、松耦合、以及运行时间长和不可预见性等特点。它能有效地集成 Internet 上的业务过程和应用程序。Web 服务涉及到的最基本的技术规范包括：Web 服务描述语言 WSDL[1]、统一描述、发现和集成 UDDI[2]以及简单对象访问协议 SOAP[3]。由于 Web 服务建立于 SOAP 协议之上，基于 XML 的 SOAP 消息在 HTTP 协议上传输，能很好地穿越防火墙，从而使电子事务的过程可以跨越各个系统。同时使用 XML 进行消息传递，也使信息和数据能在异构系统间传递，并使不同系统间的协同工作成为可能[4]。Web 服务技术应用的日益广泛使 Web 服务成为 Internet 网络环境中资源封装的标准[5]。

随着电子商务的不断发展以及跨组织应用的日益复杂，单个 Web 服务已无法满足复杂业务的需求，同时，运行在不同平台之上的 Web 服务可能是以不同的方式创建、用不同程序语言实现、由不同供应商提供的，因此需要根据特定的应用背景和需求将多个具有不同业务功能的 Web 服务按照一定的业务流程逻辑组合起来，构建复杂的服务满足业务需求[5]，这就是组合服务。组合服务是通过将现有的 Web 服务聚合起来按照一定的次序工作而提供的一个新的增值服务[6]，其中现有的 Web 服务称为它的成员服务。组合服务为复杂的 Web 应用提供了有效的解决方案，它提高了 Web 服务组件的可重用性和利用率，使得企业应用集成变得越来越容易。总的来说，Web 服务可以划分为两种，基本服务和组合服务。基本服务即原子服务，不能再分离出其他服务，其功能单一，往往不能满足终端应用的需求；组合服务则是由基本服务组合而成，通过组合可以得到新的复杂的能满足终端应用需求的 Web 服务。

为了满足建立在资源动态变化之上业务日益复杂的商务应用的需求，Web 服务有效地利用现有的 Web 数据集成、数据过滤、服务组合等技术，以进行 Web 上分散资源的集成，并在其中应用访问控制、事务机制，以保证服务响应的可靠性以及服务组件的协同工作[7]，这些技术构成了 Web 服务的核心支撑技术，也是 Web 服务领域研究的热点。在 Web 服务的组合和协调工作中，首先需要保证服务执行的质量和可靠性，这是 Web 服务的质量控制问题；其次需要保证服务执行中的安全性，这是 Web 服务的访问控制问题[7]。这两个问题都是 Web 服务，尤其是组合服务所面临的重要课题。

1.1.2 问题的提出

Web服务提供的是相互独立的网络平台上应用的连接和信息的集成，在这样的分布式环境中，Web应用需要相互协同工作并保持一致，以得到可靠的结果和输出[7]，我们可以借用事务机制来实现。事务机制是Web服务能否投入商业应用的关键技术之一。作为容错系统故障，提高系统可靠性的有效手段，事务机制被应用到Web服务中，为Web服务执行结果的正确性、一致性提供保障。应用于Web服务中的事务称为Web服务事务。

相对于基本服务，组合服务出错的可能性更大，可能的情况有：① 成员服务失败；② 网络环境故障；③ 组合服务运行平台出错[8]。同时组合服务的业务过程由来自于不同业务领域的业务任务组成，而这些业务域是松散耦合的，它们位置分散，且各自有运行平台[9]。因此组合服务需要事务机制来进行错误恢复、协调管理不同业务域的交互合作以及对多个成员服务的输出状态或结果进行处理，以保证组合服务的正确性与可靠性，这种事务称为组合服务事务。组合服务事务并不严格遵循 ACID 属性。由于组合服务所处的松散耦合环境和长时间运行的特点，传统的锁技术不再适用，组合服务事务需要放松事务的 ACID 属性，并通过补偿事务来保证这种放松的属性。本文将重点对组合服务的事务性进行研究探讨，文中的 Web 服务事务主要指的是组合服务事务。

1.1.3 研究的目的与意义

由于单个 Web 服务功能单一，Web 服务只有组合起来才能实现功能强大的增值服务，但是组合服务的真正价值会受到其可靠性的影响。不同的 Web 服务组合，由于涉及不同领域、不同组织、不同平台，而且各服务均是自治独立的，加上网络等因素，错误发生的几率很大，所以事务必然成为服务必不可少的一部分，组合服务需要事务处理来保证结果的一致可靠性。

随着人们对组合服务质量要求的提高，组合服务事务变得越来越重要。组合服务事务机制的使用不仅能保证服务质量，还能在一定程度上提高服务质量。例如组合服务事务的并发控制能提高服务效率和系统吞吐量，缩短服务请求的响应时间，并保证服务在并发过程中的正确性。而错误恢复则能为服务事务的一致性提供保证，使得服务事务在错误发生时，也能正确恢复。因此，本文希望通过组合服务事务的研究，提供一些新的思路来保证并提高服务质量。

1.2 Web 服务的事务性研究现状

Web服务中的事务处理通常是对高级事务处理技术的扩展。工业界注重的是 Web 服务事务规范和协议的标准化，并且提出多种 Web 服务事务相关规范；学术界的相

关研究工作则主要来源于事务性过程、Web 服务组合的事务性等研究领域。

1.2.1 Web 服务事务处理相关工业规范

常见的 Web 服务事务规范包括 BTP（Business Transaction Protocol）[10]、WS-T/WS-C（Web Services Transaction/Web Services Coordination）[11-13]、WS-CAF（Web Services Composite Application Framework）[14-16]等。

BTP 规范是由 OASIS 组织在 2002 年针对 Web 应用提出的一个基于 XML 的事务支持协议，用于协调多个有自主行为的参与者。它支持两种扩展事务，即原子事务和内聚事务。WS-T/WS-C 是 IBM、Microsoft、BEA 等在 2002 年 8 月发布的基于 XML、SOAP、WSDL 等 Web 服务标准的规范。它定义有两种事务类型，原子事务和业务活动，同时它定义了一个可扩展的协调框架。该框架提供三种协调服务：激活服务、注册服务、协议服务，以支持原子事务和业务活动。WS-CAF 则是 Arjuna、Fujitsu、IONA、Oracle 等公司于 2003 年 7 月发布的。此规范定义了三种事务类型，ACID 事务、长时间活动和业务流程。这三种规范都是工业界提出的，在第二章将做详细介绍。

这些规范由不同的厂商提出，具有相似性，同时又存在着竞争，不利于 Web 服务的发展。2007 年 5 月 8 日，国际标准组织 OASIS 宣布 WS-Transaction（Web Service Transaction）1.1 版为最高等级的 OASIS 标准。这对 Web 服务和 SOA 未来的发展至关重要，它为开发人员提供了构造可靠分布式程序所需的框架。

1.2.2 Web 服务事务处理的学术研究现状

学术界主要从事务模型、并发控制以及错误恢复等方面对 Web 服务的事务性进行研究。

1、Web 服务事务模型

Web 服务事务是松散耦合的，是跨企业的，具有不可预知性和运行时间长的特点。而传统的事务模型都是 ACID 事务，不能满足 Web 服务事务的需求。因此必须针对 Web 服务的环境特点，在高级事务模型的基础上做扩展。

文献[17]提出一种基于尝试保持和补偿机制的组合服务事务管理模型，同时提出了一种组合服务的多维协商模型。尝试保持能使得用户获取最新数据，能减少 cancel 的情况。在多个客户对同一个资源尝试保持时，一旦某个客户真正得到这个资源，其他客户就会被通知保持失效。文献[18]提出一种基于代理技术的事务处理模型，这种模型能够同时协调 Web 服务环境下的短事务和长事务，具有自动可靠的故障恢复机制。模型的核心部件是代理，代理主要负责创建事务（子事务）和协调器（参与者），生成补偿事务，同时负责超时检测。该模型能协调两种事务，原子事务和聚合事务。补偿事务能有效地撤销事务产生的影响。

为了能更有效可靠的组合服务，部分文献使用嵌套的事务模型框架[19,20]。其中文献[19]提出的事务框架聚集了不同的事务行为和事务语义，具有一定的通用性，同时该文提出一系列事务有效性和正确性的保证规则，并使用 ATS（Accepted Termination States）作为保证规则中的关键元素。

组合服务在执行前需要静态建模，部分研究者则考虑在组合服务建模时加入 Web 服务的事务属性，即在应用开发的早期就融入事务属性，如文献[21]和[22]。文献[21]中将组合服务的建模分为四层，结构模型（Structural Model）、工作流模型（Workflow Model）、安全模型（Security Model）以及事务模型（Transaction Model）。其中事务模型位于顶层。在事务模型中采用 UML 建模，并采用了 WS-Transaction 规范中的原子事务和业务活动以区分 ACID 事务和长事务，在支持事务补偿的同时，还可以指定服务质量。

2、Web服务事务的错误恢复

在 Web 服务环境中，存在很多可预知的或者不可预知的因素，会导致 Web 服务的失败。在这种情况下，Web 服务事务需要一种有效的错误恢复机制来处理，保证 Web 服务事务的一致性结果。研究者们纷纷提出各种错误恢复算法。总得来说错误恢复算法可以划分为两种[23]，向前恢复[6,8,24]和向后恢复[25-28]。前者往往是根据应用需求，依赖于异常处理机制，将事务状态迁移到一种可以接受的状态；后者则可以有多种策略去实现，如补偿、重试、替换等等。

向前恢复的算法基本上都采用 ATS 来描述事务的一致性，即事务所有可能的终止状态都被事先定义好，事务执行的终止状态只有满足 ATS 才能保证事务的一致性。文献[6]提出一种构建可靠组合服务的方法，以保证组合服务事务放宽的原子性，该文通过 Web 服务间的事务依赖关系来得到组合服务的事务流，事务流执行的终止状态只有满足 ATS 才能保证事务的一致性。文献[8]提出一种基于事务模型的向前恢复的方法，该文将 CA Action（Coordinated Atomic Action）这一概念应用到 Web 服务事务，从而提出 WSCA（Web Service Composition Action）。它将成员服务事务与组合服务事务分离，即对于组合服务来讲，其成员服务只是外部资源，同时它为每一个参与者指定了异常时应采取的行为。

分布式事务主要使用向后恢复的方法来保证事务完全或者部分的满足 ACID 属性[29]。Web 服务事务也不例外。虽然一些 Web 服务协议在提供事务管理功能时，采用补偿机制来进行错误恢复，然而用于这些模型和协议中的补偿机制是固定的，不能满足不同应用的需求。所以文献[25]针对业务流程提出一种多补偿机制以保证错误恢复，即一个业务活动可以定制多个补偿操作，当发生异常时，可以根据异常的情况选择合适的补偿操作，从而提高了补偿的灵活性，更能满足业务的需求。文献[26]则根据终端用户需求以及业务规则建立补偿依赖关系，该文中定义了三种补偿依赖（requirement、exclusive、hint），协调器则根据这些依赖关系，以及由参与者传回来

的条件值做补偿的决定。

文献[30]则将向前恢复和向后恢复结合起来，充分发挥了这两种算法的优点。该文中向前恢复部分主要采用了 ATS 来放宽事务的原子性。而在向后恢复部分则采用域的形式来编制 Web 服务，域中的异常是可以捕获处理的，其异常处理主要基于四种策略：skip，retry，alternate，replace。

3、Web 服务事务的并发控制

并发控制是事务的关键技术之一，它可以保证多个事务正确有效的并发执行，以提高系统的吞吐量、效率和响应时间。人们通常采取可串行化作为事务并发执行正确的判断准则，即事务的并发执行当且仅当其执行结果与这些事务按某一次序串行执行时的结果相同时，才是正确的。

短事务的并发控制常采用两段提交协议，然而这对长事务并不适用，因此一些高级事务模型中提出一种 check out/check in 的并发控制机制[31]。文[32]则提出一种名为 Jenova 的并发控制机制，该机制建立于嵌套事务和允许控制的基础上。其核心是允许控制（Admission Control），即一个服务在调度执行前，先检查资源是否足够，只有在资源足够的情况下，才能调度执行，同时更新资源。对于运行时间长的服务，该机制采取资源锁的策略。文[33]则结合了当前的 Web 服务事务协议和规范，基于服务提供者提供的依赖关系图，利用事务依赖管理器，对 Web 服务事务进行并发控制。文中扩展了 WS-Transaction，使其具备并发控制的功能。文[34]则在现有两段提交协议的基础上，提出一种基于优先级调度机制的优先提交协议（Priority Commit Protocol），该协议考虑了消息的延迟问题，延迟消息的优先级别会降低，从而保证实时事务能得到及时的处理。

4、Web 服务事务的调度和优化

除了并发控制和错误恢复，事务流程的调度和优化也是 Web 服务事务监控执行中的关键技术。Web 服务事务的调度包括对 Web 服务事务的执行顺序的检测、验证和优化。其中检测和验证是为了分析 Web 服务执行顺序流中是否存在不可达节点以及是否存在死锁和活锁，而优化是对 Web 服务执行和调度顺序进行改进，使优化后的执行效果等同于优化前，而性能得到提高[35]。

Web 服务组合的优化更着重于流程，通过分析并改进业务流程，以最大化流程内部的并行度。例如，文[36]提出了一种分析算法，基于程序依赖图（Program Dependence Graph）将组合的 Web 服务进行代码分离，并重新生成符合原来语义的程序图和新代码，将 Web 服务组合并行化。文[37]提出了一种 Web 服务组合的行为分析（Behavioral Analysis）算法，通过检查 Web 服务组合的各个状态，分析各服务调用以及执行步骤的静态依赖关系，将服务调用并行化，减少网络通信开销，从而提高 Web 服务组合的性能。

对于 Web 服务，当有多个服务请求时会存在一个服务请求队列。Web 服务事务

流的处理多采用尽力而为(Best Effort)的服务模型，即单队列、先到先服务(First Come First Service，FCFS）的服务模型，当等待队列满时，采用尾部丢掉的方法[38]。但是这种方法在 Web 服务的事务处理中显得粗糙，它对服务类型不加区分、对服务请求不分轻重缓急统一处理，造成处理效率不高、服务质量不好。为了区分服务请求，最常用的是优先调度机制[39,40]，在这种机制下，不同类型的服务请求被赋予不同的优先级，优先级决定了服务请求的调度顺序。有研究者提出基于服务质量（Quality of Service，QoS）的优先级调度，例如文献[41]和[42]，其中文[42]通过服务请求者的 QoS 要求，如服务价格、响应时间等来对服务请求划分优先级。该文提出的 QoS 管理调度既支持静态调度，也支持动态调度。静态调度时优先级值从三个层次级别考虑，即应用层优先级别、设备层优先级别、客户层优先级别；动态调度则通过配置来实现，动态调度组件对于系统管理员是透明的，一旦配置好，则自动执行。

1.3 研究内容

Web 服务环境需要事务处理机制所提供的协调行为，以保证一致可靠的结果。本课题研究的是组合服务的事务性，具体研究内容有：

1、组合服务事务模型。组合服务所处的松散耦合环境要求其事务进一步放松 ACID 属性，对高级事务模型进行扩展，以保证语义上的 ACID 属性。同时为了提高组合服务事务处理的灵活性和可靠性，必须对组合服务事务进行进一步的事务划分，将平坦的组合服务事务模型扩展为嵌套的事务模型。

2、组合服务事务的并发控制研究。组合服务事务的并发包括来自同一种组合服务的事务实例并发和来自不同种组合服务的事务实例并发。组合服务事务的并发控制对于保证服务执行的正确性、提高服务效率以及服务质量都是很重要的。因此需要对组合服务事务的并发控制进行研究，以提出一个有效的组合服务事务并发控制算法。

3、组合服务事务的错误恢复研究。由于 Web 服务自身的特点，传统的回滚机制不再适用，补偿机制变得至关重要。同时错误的发生，会影响服务质量，因此 QoS 也是恢复中要考虑的一个重要因素。因此需要从这两点出发，对组合服务的事务恢复进行研究。

4、原型系统的设计。目前虽然出现了多种的组合服务执行引擎，如 IBM 的 BPWS4J[43]等等。但是大都缺乏对组合服务事务的强有力支持。因此支持组合服务事务的原型系统也是本文的研究内容之一。

1.4 本文组织结构

本文共分六个章节，各章节内容如下：

第一章介绍 Web 服务事务研究的背景，分析了 Web 服务事务的当前研究现状，

指出了研究的意义和目的，并在此基础上提出了本文的研究内容。

第二章介绍了事务处理技术的相关概念，并对 Web 服务事务的相关规范做了分析和比较。同时还介绍了服务组合语言 BPEL，并对 BPEL 语言进行了事务支持扩展。最后提出了一个基于域的组合服务事务模型 SBET。

第三章结合组合服务事务模型 SBET，在组合服务事务恢复的过程中引入 QoS，提出了一个支持 QoS 约束的组合服务事务恢复算法，并对其进行正确性分析以及模拟实验验证。

第四章提出了一个冲突概率模型，并在此基础上结合乐观并发控制与悲观并发控制提出一个基于冲突概率的混合并发控制算法。同时对该算法进行了可串行性分析和死锁分析。

第五章主要在前面的基础上，基于开源项目 ActiveBPEL，设计了一个遵循 WS-BPEL 规范的支持组合服务事务的原型系统 TCWS，重点在组合服务事务管理系统 TMSS 以及其各关键部件的设计。

第六章对本文的工作进行总结，并指出本文工作的不足及以后研究的重点。

第二章 Web服务事务及其模型

2.1 事务处理技术

事务处理概念诞生于20世纪70年代初，最早源于数据库管理系统。事务处理技术对数据库管理系统（Database Management System，DBMS）的真正成熟和实用化，以及顺利进入市场起到了关键作用[29]。事务最早是在商务运作的应用程序中用于保护集中式数据库中的数据。后来，事务的概念逐渐扩展到分布式计算这一更广泛的领域中。今天，事务已经成为一个重要的编程范例，是构建可靠的分布式应用程序的关键。

2.1.1 传统事务及其属性

事务是构成一个逻辑工作单元的操作集合[29]，它是保证共享数据的并发访问和失效恢复的关键。并发控制和错误恢复是事务的核心技术[44]。为确保数据库中数据的一致性，由离散的数据操作组成的事务具有逻辑完整性，只有这些操作全部完成时，数据的一致性才能得以保持；任何一个操作失败，都视整个事务失败，事务应该回滚到初始状态。

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的原子单位。它主要有以下四个属性，原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），简称为ACID属性，其具体含义如下：

1、原子性：事务中所有的操作是不可分割的，要么全做要么全不做，即如果事务成功，则所有操作都发生；如果不成功，则所有操作都不发生。

2、一致性：事务执行的结果必须是使数据库从一个一致性状态变迁到另一个一致性状态，没有其它中间状态的存在。这意味着只有在应用程序完成的时候才进行有效状态的转变，从而使所有的数据都保持一致性。

3、隔离性：一个事务的执行不能被其他事务干扰，即在事务成功完成之前，其内部的操作及使用的数据对其它并发事务是隔离的，各项操作的结果都不能被外界共享。隔离性是事务可并发执行的保证。

4、持久性：一个事务一旦提交，它对数据库中数据的改变就是永久性的，工作的结果将会持久化保存起来，即使以后系统发生故障，也能保持或者恢复。

在数据库系统中，事务的正确性由DBMS予以保证。DBMS一般通过恢复协议保证事务的原子性和一致性，通过并发控制协议保证事务的隔离性，通过提交协议保证事务的持久性。

2.1.2 高级事务模型

学者们已经提出多种事务模型，其中扁平事务模型是所有事务模型中最简单的一种，其事务性应用控制只有一层。扁平事务模型能保证事务的ACID属性，适用于短事务和较为简单的应用。但在面对长事务时，扁平事务模型不能满足需求。长事务不能通过简单的回滚来保证一致性，因为这会导致时间和资源的浪费。于是人们对扁平事务模型进行扩展，适当调整并放松事务的ACID要求，提出多种高级事务模型，主要有嵌套事务模型、Sagas模型、分支/汇合事务模型、Flexible事务模型等等。

1、嵌套事务模型[45]

嵌套事务模型允许子事务的包含关系，父子事务之间形成树形结构。子事务可以是嵌套的，也可以是扁平的，但是处在叶子节点的子事务一定是扁平的。子事务只能在父事务开始后才能开始，而且只有父事务提交时，子事务的提交才能生效。父事务回滚则其子事务全部回滚。分层结构提高了事务模块化程度，允许更细粒度的恢复和更高的并发度。

2、Sagas模型[46]

Sagas模型的关键概念是补偿事务，它容许将长事务划分为多个子事务。每一个子事务都对应一个补偿事务，用于取消失败时子事务产生的影响。Sagas是由预先定义好执行顺序的子事务集合T和对应的补偿子事务集合CT组成。一旦某个子事务t失败，系统将通过逆序执行补偿子事务以撤销t和t之前提交的所有子事务产生的影响。

3、分支/汇合事务模型[47]

分支/汇合事务模型采用动态重构技术，对活动的各个并发事务进行动态的修改。在执行中，一个事务可以分支为两个独立或者相互依赖的事务，然后可以将这两个事务合并成一个事务。通过动态重构执行中的事务，可解决多个用户的合作问题，提高事务的并发度和吞吐率。

4、Flexible事务模型[48]

Flexible事务模型是一种适用于多数据库系统的事务模型。一个Flexible事务由多个子事务组成。根据子事务的恢复特性，可将其分为可补偿的、可重试的和Pivot三种。该模型通过指明各子事务间的依赖关系来支持对事务执行的柔性控制，其关键是提供备用执行路径。如果主子事务被撤销，那么提交备用子事务并希望它成功执行以完成指定的任务。

2.1.3 Web服务的事务性

Web服务不能只使用独立于其他服务或者组件的事务，这种事务不能离开其所依赖的Web服务进入另一个Web服务。因此，Web服务需要一种能在服务间流动的事

务来保证应用的一致性结果。这就需要在传统事务模型的基础上，吸取高级事务模型的做法，放松 ACID 属性，以适应 Web 服务环境。例如，Web 服务事务多采用高级事务模型中的补偿技术，而且允许部分参与者工作失败而整个事务继续运行。在 Web 服务集成和交互过程中，Web 服务事务能协调各服务，并对其输出结果进行处理。

相对于传统事务，Web 服务事务主要有以下特点：

1、Web 服务涉及到商务处理，加上网络延迟和与用户的交互，往往运行时间长，使得传统的锁定资源的策略不再适用，取而代之 Web 服务事务采用补偿技术来保证一致性。

2、Web 服务事务的参与者分布于 Internet 上，一般来自于不同的组织和部门，而且各参与者有着自治的特点，协调起来比较困难。

3、Web 服务事务比传统事务更松散更灵活，放松了事务 ACID 属性中的原子性，Web 服务的商业逻辑往往允许部分参与者工作的失败不取消整个事务的运行。

4、Web 服务事务处于完全开放的环境下，可能会遇到更多的故障问题，如网络通讯中断、网络阻塞延迟、系统节点崩溃等等。Web 服务事务需要有能力对各种故障做出及时的处理和恢复。

5、Web 服务事务对于服务质量和计算的可靠性起着重要的作用[7]。

2.2 Web 服务事务相关规范的分析与比较

由于开始没有一个统一标准的 Web 服务事务规范，业界各组织企业纷纷提出自己的 Web 服务事务规范。目前已经出现的 Web 服务事务规范包括 BTP[10]、WS-T/WS-C[11-13]、WS-CAF[14-16]等。

2.2.1 Web 服务事务规范 WS-Transaction OASIS 标准

WS-T/WS-C 最早是由 IBM、Microsoft、BEA 等在 2002 年 8 月发布的事务规范，它基于 XML、SOAP、WSDL 等 Web 服务标准，跟 Web 服务紧密绑定，是针对 Web 服务提出的。之后，经过了多次修改，于 2005 年 8 月形成 WS-Transaction 1.0 版本。国际标准化组织 OASIS 则在 1.0 版的基础上，于 2007 年 4 月发布 WS-Transaction 1.1 版本。WS-Transaction 最终成为 Web 服务事务统一的标准规范。

WS-Transaction OASIS 标准由 3 个子规范组成：WS-Coordination[13]、WS-AtomicTransaction[11]及 WS-BusinessActivity[12]。该规范能使现存的业务流程、工作流以及其他应用系统隐藏其私有协议，并在异构环境内运行。

规范中定义了两种事务类型：原子事务和业务活动。其中原子事务是为了兼容遗留的需要 ACID 属性的应用系统而设计的。原子事务假设事务是短事务，它具有严格的 ACID 属性，其操作要么全部发生，要么一个也不发生。而业务活动则是针对长事

务设计的，Web服务上的操作表现为松散的工作单元，也即任务。任务的结果在整个活动完成之前就可以被其他事务共享。同时它采用补偿机制来实现错误恢复，补偿的语义是每个事务参与者会撤消它在对话期间已经执行完的操作。原子事务并不是业务活动的一个特例。

针对这两种事务类型，该规范大致提供了两种协调协议：① 原子事务协议。主要用于处理短期存在的活动，要求事务作用域内所有工作全部完成，也就是说，活动如果成功了，则所有任务都已执行，如果不成功，则未执行任何任务。只有当成功完成时，活动的结果才对其他事务可见。② 业务活动协议。主要用于处理运行时间长的业务活动。由于业务活动可能需要长时间才能完成，为了使其他用户能在最短的延迟内访问该活动所占用的资源，必须在整个活动完成之前将任务的结果释放。鉴于此，规范引入了故障处理机制和补偿处理机制，以保证在出错时能将业务活动恢复到语义一致性状态。

WS-Coordination 子规范提供了一个可扩展的协调框架，使应用程序能够创建协调上下文。同时它支持以下服务：激活服务、注册服务和协调服务，其中激活服务主要用于开始一个新的事务并指定该事务可以使用的协调协议；注册服务提供注册操作以保证 Web 服务通过注册以参与协调；协调服务则通过所选择的协调协议控制已注册 Web 服务的事务行为以完成活动的处理。

2.2.2 其他规范

为了解决 Web 应用中来自不同组织的参与方之间在商务上的协调合作问题，OASIS 组织在 2002 年提出了 BTP 规范，它是松耦合领域中业务到业务的事务规范，是第一次跨行业的尝试，目的是为了制订一个用于 B2B 事务的 XML 标准，允许多个 xml 消息以&的方式混合传递。BTP 规范既适用于 Web 服务，还适用于其它任何长时间运行的事务。它定义了两种类型的事务：① 原子型（Atom）事务，具有原子性，即事务中的操作要么全部执行，要么全部不执行。② 内聚型（Cohesion）事务，放宽了原子性的要求。一个中心协调者负责检查每个事务参与者的状态，即使某些参与者提交失败，该协调者仍然可以决定让其他参与者提交。同时该规范扩展了传统的两阶段协议，在两个阶段协议中插入业务逻辑决策。这就意味着需要用户显示的驱动这两个阶段，而且“准备”成为了服务业务逻辑的一部分。但是该规范有一个缺陷，它基于一个假定：两阶段协议是适合于所有用例的。

WS-CAF 规范是由 Arjuna、Fujitsu、IONA、Oracle 等公司于 2003 年 7 月共同定制发布的。它是一种开放的、灵活的和轻型的规范，该规范由三个子规范组成：Web 服务上下文 WS-CTX（Web Service Context）[15]，Web 服务协调框架 WS-CF（Web Service Coordination Framework）[16]和 Web 服务事务管理 WS-TXM（Web Service Transaction Management）[14]。其中 WS-CTX 负责上下文 Context 的管理，与任务相

关的所有 Web 服务可以就共同的结果共享 Context 和交换信息。该子规范已于 2007 年 4 月成为 OASIS 标准，它为 Web 服务的边界划定及相互协作提供了标准的、可互操作的方法，能保证多个 Web 服务在复杂的执行环境中运行如同在单一连续的环境中运行。WS-CF 定义了一个名为 coordinator 的软件级代理，提供对事件机制的支持，同时定义了协调服务参与者的角色和职责、Web 服务环境中的消息映射。WS-TXM 则定义三种类型的事务：① ACID 事务，即传统的 ACID 事务。② 长时间活动 LRA（Long Running Action），不具有 ACID 属性，但仍然具有原子性。③ 业务流程 BP（Business Process），复杂的业务处理。同时定义了三个与事务类型对应的事务协调协议，这些协议能够插入协调框架以实现事务管理器、长期运行补偿、和异步业务流程之间的互操作。同时 WS-TXM 支持多种业务模式。

2.2.3 Web 服务事务规范的比较

BTP 支持参与者的自治，这成为 BTP 可用于 Web 服务的有利证据。然而 BTP 不是专门用于 Web 服务的事务处理协议，其目的是也能用于其他的环境中，因此 BTP 定义了事务性的 XML 协议，而且必须指定所有的服务依赖[49]。但是 BTP 的一个协议就可以适用于所有用例的观念不能得到普遍接受。

WS-CAF 的 LRA 和 WS-BA 相似，两者都采用补偿机制。但 LRA 的参与者注册是在其工作完成时才进行，而且事务结果需要所有参与者参与协调。WS-BA 则允许单个参与者补偿，而不涉及整个事务结果的协调。ACID 事务和 WS-AT 事务都必须是短事务，具有 ACID 属性，处于紧密耦合环境中。但是 WS-CAF 的失败恢复语义跟协调器所支持的协议紧密结合，不由规范统管，这会导致错误恢复比较困难。

WS-Transaction 的优点在于协调框架和事务类型的分开，具有良好的可扩展性。另外，它定义了两种事务类型，一方面，WS-AT 可以处理传统的 ACID 事务，这使得 Web 服务和传统系统之间的互操作不会因为事务类型不一致而受到影响。另一方面，WS-BA 的使用给了 Web 服务提供者很大的发挥空间，它们可以根据 Web 服务自身的特点提供个性化的补偿机制。

2.3 Web 服务组合语言 BPEL 的事务扩展

2.3.1 BPEL 概述

组合服务可以用业务流程来描述。业务流程由一系列逻辑相关的任务组成，通过这些任务的执行能得到预先定义好的业务输出[20]。BPEL（Business Process Execution Language）[50]作为业务过程描述语言，它基于 XML 和 Web 服务技术，融合了早期 IBM 的 WSFL（Web Services Flow Language）及微软的 XLang 规范的很多特点。2007 年 4 月，WS-BPEL 2.0 版于被宣布成为 OASIS 标准。至此，BPEL 已经成为被广泛支

持的最成熟的技术。WS-BPEL 基于 Web 服务标准，可以用于指定一组 Web 服务操作的可能执行顺序及其相互依赖关系，以保证业务流程能完成特定功能。它将业务流程中涉及到的公众的方面和内部私有的方面进行分离，既可适用于描述业务交互的可执行流程，也可用于描述公众可见的抽象流程。抽象流程主要进行描述性工作并且允许多种使用案例。

WS-BPEL 通过标准化以及广泛协作形成了 SOA 的基础标准。通过它可以组合、编制和协调 Web 服务，可将多个 Web 服务组合到一个业务流程中，从而得到更加强大的 Web 服务——组合服务。同时 BPEL 提供错误处理和补偿机制，补偿成为业务逻辑的一部分。然而它并没有包含与主流商业事务规范 WS-Transaction 相兼容的编程结构，即 BPEL 并没有对 WS-Transaction 提供支持[51]。

2.3.2 BPEL 的事务支持扩展

BPEL 所描述的业务流程支持事务有两种情况：流程所调用的 Web 服务作为事务参与者；业务流程本身封装成 Web 服务，作为其他组合服务的事务参与者。BPEL 主要通过作用域（scope）来实现错误处理和补偿，作用域可以嵌套，它具有事务的某些功能特性，但是作用域不等同于事务。虽然 BPEL 可以通过调用参与者提供的 execute 操作或者 cancel 等操作来达到最终的结果（confirm 或者 cancel），但是这些对事务的支持还不够。为了保证组合服务一致可靠的运行结果，本文参考文献[52]，对 BPEL 进行扩展，使之支持 WS-Transaction 事务协议。

在业务流程中，活动是能执行一定功能的元素，它可能只是简单的发送或者接收信息，也可能执行复杂的业务功能[20]。在 BPEL 中，即使控制结构也用活动表示。因此，为了满足事务需要，根据文献[20]以及 WS-BusinessActivity，本文在 BPEL 中引入任务这一新元素。每一个任务包含且只包含一个 invoke 活动，同时可以包含其他的例如 assign、copy 之类的活动。

1、任务的定义

任务是能完成一定业务功能的基本元素，应该具有的属性有：任务名，功能。示例如下：

```
<task name="bookHotel" funtion="bookHotel" />
……
<invoke partnerLink="partnerLink1"…… />
……
</task>
```

2、候选服务绑定

在 BPEL 中，Web 服务被称作伙伴，而服务操作则通过伙伴连接来绑定。为了引进候选服务，需要对 partnerLink 元素进行扩展，使之与任务绑定。用 task 属性表示

该伙伴连接对应的任务。示例如下：

```
<partnerLink name="bookHotel1" partnerRole="partnerRole"
partnerLinkType="ns:partnerBookHotel1" task="bookHotel" />
<partnerLink name="bookHotel2" partnerRole="partnerRole"
partnerLinkType="ns:partnerBookHotel2" task="bookHotel" />
```

3、协调上下文变量的定义

在 WS-Transaction 中，协调上下文是事务参与者之间共享的信息。在 BPEL 中，用变量来定义在业务流程中需要处理的相关数据，以说明与合作伙伴之间有状态交互。因此在 BPEL 中可以通过变量申明来定义协调上下文。示例如下：

```
<variables>
<variable name="travelTransactionCtx" type="wscoor:coordinationContext" />
.....
</variables>
```

3、事务分界：事务开始

在业务流程中显示的划分事务界限，可以使组合服务的事务处理更加灵活。本文引进标签 businessTransaction 来标识事务，并用 action 属性来表示事务行为，当 action 值为 new 时，表示事务开始。为了表示事务的嵌套关系，可以在事务初始化时加上父事务 parent 属性。同时为了区分组合服务事务和任务事务以及活动事务，还必须加一个属性 transactionType，其值可以为 composite、scope、task。事务特性则用 recoverProperty 表示，其值可以为：compensatable、retriable、pivot、ignorable、replaceable、decided。示例如下：

```
<businessTransaction name="bookHotelTransaction" action="new"
recoverProperty="compensatable" parent="travelTransaction"
context="travelTransactionCtx" transactionType="task"/>
```

4、与外界交互的活动的扩展

在 BPEL 中存在三种与外界交互的活动，分别为 receive 活动、invoke 活动和 reply 活动。为了使其能够识别业务事务的上下文，需要分别进行扩展。

(1) receive 活动主要用来接受外部的调用，以启动业务流程。在接受外部调用时，receive 活动会获得外部传来的协调上下文，为此增加一个 inputContext 属性来标识。示例如下：

```
<receive name="bookingReceive" parterLink="customer"
portType="ins:travelBookPT" operation="travelBook" createInstance="yes"
variable="bookingRequest" inputContext="receivedContext" />
```

(2) invoke 活动主要用来跟成员服务交互，以完成业务功能。当业务流程调用一个成员服务时，会启动该服务的事务，即外部事物。为了保证外部事务参与协调，必

须对该活动进行扩展，加入属性 inputContext 和 outputContext，分别代表外部事务协调上下文和当前事务协调上下文，使其彼此能识别。示例如下：

<invoke name="bookHotel" parterLink="bookHotel" operation="bookHotel"
portType="bookHotel:bookHotelPT" inputVariable="hotelNumber" .....
inputContext="hotelTransactionCtx" outputContext="travelTransactionCtx" />

(3) reply 活动主要在同步交互过程中用来响应外部的调用。需要把当前事务上下文返回给其合作伙伴，为此增加一个属性 outputContext。示例如下：

<reply name="bookReply" parterLink="customer"
portType="ins:travelBookPT" operation="travelBook"
variable="replyMsg" outputContext="travelTransactionCtx" />

5、待询问的参与者的指定

根据 WS-Transaction 规范，在协调的过程中，业务活动需要检查哪些任务已经完成，以决定事务是提交还是取消。同时为了增加灵活性，进一步放开事务的原子性，允许只询问部分事务参与者是否已经完成，为此加入 participants 属性。示例如下：

<businessTransaction name="travelTransaction" action="complete"
participants="bookTrain,bookHotel" context="travelTransactionCtx"/>

6、事务分界：事务结束。

根据各个任务的完成情况，事务要么确认提交结束，要么失败回退中止。因此驱动事务结束的动作有两种：confirm、rollback。示例如下：

<businessTransaction name="travelTransaction" action="confirm"
context="travelTransactionCtx"/>

2.4 基于域的组合服务嵌套事务模型 SBET

2.4.1 嵌套事务模型 SBET

从组合服务提供者的角度来看，因为看不到成员服务的具体事务行为，所以组合服务事务只是一个平面事务，而其成员服务则只是具有事务行为的事务参与者，能参与组合服务的协调。为了提高组合服务事务处理的灵活性和可靠性，本文从服务提供者的角度考虑，结合补偿的特点，通过对组合服务事务进行进一步的事务划分，将平坦的组合服务事务模型扩展为嵌套的事务模型。

定义 2-1（域）：域是多个连续任务或子域的有意义的封装，可以看作一个特殊的任务。用 S 表示一个域，则 $S=[t_1,\ldots,t_n]$，其中 t_i 表示一个任务或者子域。

例如组合服务中存在两个连续任务，订火车票 t1，订酒店 t2，若 Internet 上存在能同时完成这两个功能的服务，就可以将它们封装为一个域，S=[t1,t2]。

定义 2-2（内部事务）：内部事务是指在组合服务流程内部激活的事务。

内部事务主要包括三类事务：处于最顶层的组合服务事务CST，处于中间层的域事务ST，处于最底层的任务事务TT。其中ST和TT都是CST的子事务。TT是组合服务嵌套事务树的叶子节点，它对应于业务流程中的任务，在组合服务的执行过程中随着任务的激活而激活。在未做域的划分时，TT直接作为CST的子事务存在。ST能提高CST的效率和并发度，不同的ST可以并发执行，同时ST能提高CST的错误处理能力，它跟CST错误恢复的粒度直接相关。

定义2-3（外部事务）：外部事务是指由组合服务以外其他实体（例如成员服务）激活的事务。

外部事务由成员服务提供，可能只是一个基本服务事务，也有可能是一个新的组合服务事务。对于组合服务来讲，它只是其事务参与者，只需参与其事务的协调即可。外部事务具体的事务行对组合服务事务是不可见的。

图2-1是基于域的组合服务嵌套事务模型及嵌套事务树。图中的组合服务事务为三层的嵌套事务，每个任务都对应一个成员服务，成员服务产生的事务是外部事务。

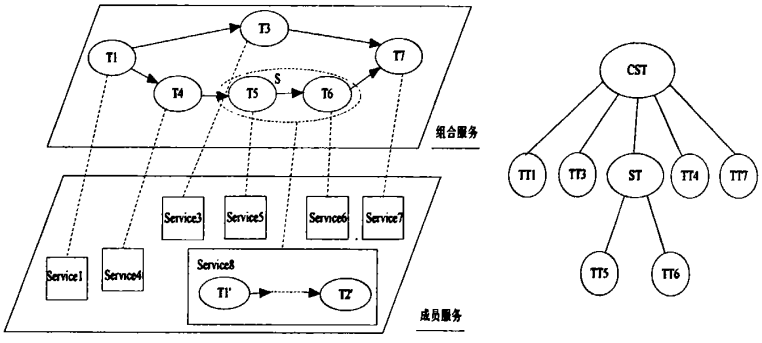

图2-1 组合服务嵌套事务模型及事务树

2.4.2 SBET中的事务属性

组合服务事务CST对传统的ACID属性放宽了要求，采用松弛的原子性、一致性和隔离性，即语义原子性、语义一致性、语义隔离性。

定义2-4（语义原子性）：一个事务，① 如果允许其操作序列中部分操作失败，而事务继续前进，直至执行完毕；② 如果某操作失败后能回退至一个用户可接受状态，并对已执行完的操作进行补偿，撤销其产生的影响，则称这个事务具有语义原子性。

换而言之，组合服务事务满足语义原子性，当且仅当它满足以下两条件之一：① 执行完毕，成功完成；② 有效撤消，成功中止。语义原子性是一种弱化的原子性。

定义2-5（语义一致性）：一个事务，如果失败后能通过补偿回退到一个用户可接受的一致状态，即使执行前的系统状态不能完全恢复，这个事务仍具有一致性。这种弱化的一致性称为语义一致性。

换而言之，事务的最终状态只要是用户可接受的状态，即使是失败，都认为该事务是语义一致的。

定义 2-6（语义隔离性）：在组合服务事务 CST 内部，由于子事务的提交可能需要在整个 CST 全部完成后才能执行，因此子事务可以读取其他子事务的未提交数据，各子事务间没有严格的隔离；在 CST 之间，因为 CST 运行时间可能会很长，因此彼此之间也不能完全的隔离，部分执行结果对彼此都可见，这种弱化的隔离性则称为语义隔离性。本文通过临界区（见 4.1.3 节）来实现组合服务事务间的语义隔离。

以上定义中的用户可接受状态是指事务可以从这个状态按照用户指示继续执行或进入一个用户认为是合理的终止状态，这个状态是满足一致性要求的。因此，我们把达到用户可接受状态的点叫做一致点。事务处于起始点要么是因为事务刚激活，尚未执行，要么是事务执行失败，通过补偿回退到起始点，因此事务起始点必定是事务的一致点。

2.4.3 SBET 中的事务特性

事务特性是用于事务错误恢复的属性特征。Web 服务处于完全开放的松散耦合中，出错的可能性较大。当发生错误时，组合服务事务必须能采取一定的措施来保证错误的恢复，这种措施也叫错误恢复策略。错误恢复策略随着错误的具体情况而定，而事务特性则在恢复策略的决策过程中起指示作用。事务特性对应于错误恢复策略，对每一个事务特性都有相应的错误恢复策略。组合服务提供者可以在编制组合服务的过程中，通过 recoverProperty 属性（见 2.3.2 节）静态地设置事务特性。在组合服务事务中，基本事务特性有：可补偿，关键的，可重试，可替换，可忽略，决定性的。具体描述如下：

1、可补偿 cp：指的是对于已经胜利完成的任务，可以通过执行补偿事务来从语义上撤销其已经产生的影响。

2、关键的 p：指的是一旦任务胜利完成，其效果是永久性的，即使在现实语义中也无法补偿撤销。现实世界中没有固定的方式和规则来处理这类事务的错误。

3、可重试 r：指的是在满足用户要求的服务质量的情况下，而且重试次数没有达到最大值时，允许重新激活任务，尽量完成组合服务事务。

4、可替换 rp：指若当前任务对应的成员服务不能成功执行完毕，在满足用户要求的服务质量的情况下，可以被另外一个具有相同功能的候选服务替换。

5、可忽略 i：指任务的功能对其父域的功能影响不大，没能成功执行完毕也能满足用户的需求时可以跳过当前任务，激活下一个任务。

6、决定性的 d：指的是当前任务的成功与否决定了整个组合服务事务是否成功。即如果当前任务失败，则整个组合服务事务必定失败。对于处于嵌套底层的任务，一旦失败，则直接决定了整个组合服务事务失败撤销，而无需向父域报告异常失败。

其中{cp，p}用于处理已经执行完毕的任务，其他的事务特性{r，rp，i，d}则用于当前任务执行失败时的处理。假设基本事务特性集 RPS＝{r，cp，p，rp，i，d}，其子集 RPDS={cp，p}，用于表示已经执行完毕任务的事务特性；子集 RPES={r，rp，i，d}，用于表示当前执行任务的基本事务特性。

规则 1 组合规则：将 RPS 中的基本事务特性合理组合起来，形成新的事务特性。

为了提高错误恢复的能力，可以利用组合规则，将基本事务特性组合起来，例如，(r，rp)表示任务即是可重试的，又是可替换的。显然 RPDS 和 RPES 之间的事务属性可以任意组合。

规则 2 相斥规则：RPS 中的两个基本事务特性是相斥的，当且仅当它们在逻辑上是相互矛盾。

RPDS 和 RPES 各自内部的基本事务特性不能任意组合，如(cp，p)是不行的，一个任务不可能同时即是可补偿的，又是不可补偿的，这是相互矛盾的。满足相斥规则的基本事务特性对有(cp，p)，(rp，d)，(i，d)。

规则 3 继承规则：

1、若父事务可忽略，则子事务一定可忽略，但子事务可忽略，父事务不一定可忽略。

2、若父事务可重试，则子事务一定可重试，但子事务可重试，父事务不一定可重试。

规则 4 优先规则：错误处理判断的优先顺序，i>r>rp>d。

错误处理策略以事务特性为依据，而事务特性可以组合起来，此时需要按照一定的判断顺序来做出错误处理策略，例如对于事务特性(rp，r)，根据优先规则，所做的处理应该是先查看是否可以重试，在重试也不成功的情况下，采取替代的策略。在优先规则中，可忽略的优先级最高，是首先应该予以判断的。当前任务执行失败时，应该根据优先规则采取相应的错误处理策略，尽量恢复组合服务事务。

2.5 本章小节

本章首先介绍了事务处理技术，同时对工业界提出的各 Web 服务事务相关规范进行分析，尤其是对已经成为 OASIS 标准的 WS-Transaction 规范进行了分析，并对各规范进行了较全面的比较。然后简要介绍了 Web 服务组合语言 BPEL，并在前人的工作基础上，针对 BPEL 对 WS-Transaction 支持的不足对 BPEL 进行事务支持扩展。最后本章提出了一个基于域的组合服务嵌套事务模型 SBET，该模型进一步放松了事务的 ACID 属性，允许服务提供者在业务流程的基础上进行域的划分，能更好的满足于用户的需求。

第三章 支持 QoS 约束的组合服务事务恢复

错误恢复是事务处理的核心技术之一。它能在组合服务事务出错时将其恢复至语义一致性状态，保证组合服务要么继续向前执行，要么撤销中止。错误恢复在松散耦合的组合服务环境中显得尤为重要。组合服务的运行失败，不仅会影响到服务提供商的当前商业利益，还会影响到其信誉度。怎样提高组合服务的成功率，保证组合服务在出现错误时能得到恢复，同时保证服务质量，这已经成为组合服务事务亟需解决的问题。

3.1 相关工作

尽管高级事务模型(ATM:Advanced Transaction Model)，如Sagas[46]等的研究已经取得了一定成果，但并不完全适合组合服务事务，原因在于ATM通常面向数据集中的应用，事务结构模式固定，而组合服务事务的失效恢复要比ATM复杂。

文献[28]着重讨论了大部分现有的事务模型所未考虑到的两个重要方面，补偿代价和终端用户需求。并在补偿代价方面提出了一个多层补偿的方法，但该方法在现有的规范和平台上是不可行的，因为没法得到组合服务在不同层次的全局视图。

文献[62]在Web服务事务中融入了QoS管理，提出了一个基于QoS的主动两段提交协议，但是该文并没有讨论Web服务事务恢复中的QoS管理。

本文在组合服务事务恢复的过程中引入了 QoS，讨论了组合服务在执行过程中动态 QoS 的计算，建立了一个基于域的组合服务嵌套事务模型，并在此基础上提出一个组合服务事务恢复算法。组合服务的事务恢复受极限 QoS 约束，使得 QoS 接近最优，并且在极限 QoS 约束不了时，能保证整个组合服务事务得到撤销，成功中止。

3.2 组合服务事务恢复中的 QoS 指标

Web 服务中的服务质量问题越来越受人们的关注。有保证的 QoS 是 Web 服务在商业应用中获得成功的关键因素。QoS 作为一个可以衡量服务质量的概念，包含了费用、运行时间、可用性、信誉度、可靠性等非功能属性[53]，能体现事务的好坏。采用支持QoS约束的组合服务事务恢复是在事务出错时有效的保证QoS的一个重要手段。

3.2.1 QoS 指标分析

Web 服务质量模型考虑的 QoS 指标主要有执行费用、响应时间、成功率、信誉度、可用性[54]。由于信誉度和可用性都涉及到组合服务的多次执行，所以组合服务事

务在恢复过程只需考虑执行费用 P、响应时间 T、成功率 S 三个指标。其中执行费用 P 指组合服务在执行过程中所消耗的总的费用；响应时间 T 指组合服务从收到服务请求到执行完毕所需的时间；成功率 S 指组合服务执行成功的概率。

表 3-1 事务恢复处理中的各种 QoS

QoS 名称	QoS 指标	描述
QoS_{pvd}	$[P_{pvd}, T_{pvd}, S_{pvd}]$	服务提供商发布的服务质量
QoS_{ulimit}	$[P_{ulimit}, T_{ulimit}, S_{ulimit}]$	服务用户能接受的极限服务质量
QoS_{plimit}	$[P_{plimit}, T_{plimit}, S_{plimit}]$	服务提供商能接受的极限服务质量
QoS_{act}	$[P_{act}, T_{act}, S_{act}]$	组合服务执行过程中实际可能的服务质量
QoS_{future}	$[P_{future}, T_{future}, S_{future}]$	组合服务执行到具体某个任务时，将来可能消耗的服务质量
QoS_{past}	$[P_{past}, T_{past}, S_{past}]$	组合服务执行过程中已经消耗的服务质量

上表中前三种 QoS 在组合服务执行前就能确定，在执行过程中固定不变。QoS_{pvd} 服务提供者静态发布的。极限 QoS 用于描述在组合服务没法达到预期的 QoS_{pvd} 时用户或者服务提供商所能忍受的最差的服务质量，在组合服务的事务恢复过程中必须予以考虑。QoS_{act} 和 QoS_{future} 则是动态的，随着业务流程的推进而动态变化，其中 QoS_{act} 受极限 QoS 约束。组合服务事务在执行过程中，若满足不了该约束，则只能中止。

设 $P_{limit} = mix(P_{ulimit}, P_{plimit})$，$T_{limit} = mix(T_{ulimit}, T_{plimit})$，$S_{limit} = max(S_{ulimit}, S_{plimit})$，则有约束关系如下：

$$P_{act} < P_{limit},\ T_{act} < T_{limit},\ S_{act} > S_{limit} \qquad \text{公式（3-1）}$$

为了直观的比较 QoS 的大小，必须对 QoS 进行量化，量化公式如下：

$$W(QoS) = w_P \times \Delta P + w_T \times \Delta T + w_S \times \Delta S \qquad \text{公式（3-2）}$$

其中 w_P、w_T、w_S 为权值，分别表示 P、T、S 对组合服务用户的重要性，且有 $w_P + w_T + w_S = 1$。ΔP、ΔT、ΔS 为各 QoS 指标相对于极限 QoS 的离差。离差的引入，可以保证在执行费用值越小、执行时间值越小、成功率值越大时，服务质量的量化值越大，具体的计算公式如下：

$$\Delta P = \frac{P_{limit} - P}{P_{limit}},\quad \Delta T = \frac{T_{limit} - T}{T_{limit}},\quad \Delta S = \frac{S - S_{limit}}{S_{limit}} \qquad \text{公式（3-3）}$$

3.2.2 QoS_{act} 及 QoS_{future} 的计算

参考文献[54]提出的思想，基于组合服务的执行路线计算 QoS 各指标值，有：

$$P = \sum_{i=1}^{n} P^{ii},\quad T = \sum_{i=1}^{n} T^{ii},\quad S = \prod_{i=1}^{n} S^{ii} \qquad \text{公式（3-4）}$$

基于执行路线简化了 QoS 的计算，但是组合服务的执行路线只有执行完毕后才能确定。若要估计未执行完的复杂的组合服务的 QoS，这种方法不可行。但是若组合服务业务流程中没有特殊的控制结构，例如与结构、循环结构，各个任务是顺序执行的，则可以直接使用公式（3-4）。下面我们讨论 QoS_{act} 和 QoS_{future} 的计算方法。

1、QoS_{act} 的计算

若组合服务事务在执行 t_i 时发生错误，根据已经消耗掉的服务质量 QoS_{past}、撤销事务所需要的服务质量 QoS_{cancel} 和将要消耗的服务质量 QoS_{future}，可以得到 QoS_{act}：

$P_{act}=P_{past}+P_{future}+P_{cancel}$，$T_{act}=T_{past}+T_{future}+T_{cancel}$，$S_{act}=S_{past}\times S_{future}$　　公式（3-5）

其中 S_{past}、P_{past} 和 T_{past} 为 QoS_{past} 的指标值，S_{past} 表示已经成功的执行完 t_i 之前的任务，P_{past}、T_{past} 则表示执行中已经消耗掉的费用和时间；S_{cancel}、P_{cancel} 和 T_{cancel} 为 QoS_{cancel} 的指标值，S_{cancel} 表示撤销操作的成功率，P_{cancel}、T_{cancel} 为撤销操作所需消耗的费用和时间；S_{future}、P_{future} 和 T_{future} 为 QoS_{future} 的指标值，S_{future} 为执行所有尚未执行的成员服务可能的成功率，P_{future}、T_{future} 则为将来可能消耗的费用和时间。

对于 QoS_{past}，由于组合服务已经成功的执行完 t_i 之前的任务，所以可以确定 QoS_{past} 各指标 S_{past}、P_{past} 和 T_{past} 的值，且有 $S_{past}=1$。

对于 QoS_{cancel}，本文以补偿事务的正确执行为前提，所以必有 $S_{cancel}=1$。当异常的 t_i 为任务时，在异常点进行错误恢复，无需额外的撤销操作，则有 $P_{cancel}=0$、$T_{cancel}=0$；当异常的 t_i 为域时，因为补偿事务是将撤销操作按照任务执行轨迹的逆序串起来的，不会有特殊的控制结构存在，因此可以根据公式（3-4）计算得到。

2、QoS_{future} 的计算

由于公式（3-4）是基于执行路线的，不能直接用于计算 P_{future}、T_{future}、S_{future}，因此在套用其来计算 QoS_{future} 之前首先必须对业务流程图进行预处理。下面给出相对于异常任务的 QoS_{future} 的计算步骤：

(1) 确定从当前异常任务节点开始的业务流程图 G_{future}。根据控制结构的类型，可以分以下四种情况讨论：

1) 异常任务节点出现在顺序结构中的情况。此时，G_{future} 直接从异常任务节点开始直至整个业务流程结束，如图 3-1 中 a 所示。

2) 异常任务节点出现在或分支结构中的情况，如图 3-1 中 b 所示。因为组合服务在执行过程中已经选择了异常任务所在的分支，所以在 G_{future} 中必须放弃其他分支。

3) 异常任务节点出现在循环结构中的情况，如图 3-1 中 c 所示。设循环次数为 n，在异常出现时，循环刚好执行了 k 次，则原循环结构变为以异常任务为起始节点的 n-k 次循环结构。

4) 异常任务节点出现在与分支结构中的情况。此时，考虑到其他分支也在执行，必须将异常任务节点和其他分支中未完成的任务节点组成新的与分支结构。如图 3-1 中 d 所示，图中加入一个空任务节点 start 以保证业务流程单入口的特点，该任务不

执行任何动作，其 P=0，T=0，S=1。

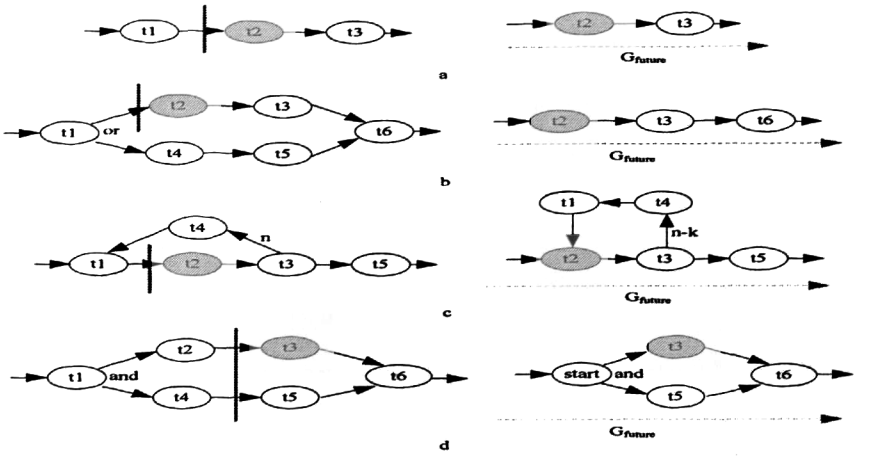

图3-1 G_{future}的四种可能情况

(2) 简化 G_{future}，替换 G_{future} 中特殊的控制结构。将图中的或分支结构、循环结构以及与分支结构封装为特殊的任务 as，并对这个任务计算其可能消耗的 QoS 指标值。

1) 对于或分支结构，如图 3-2 中 a 所示，有：

$$P_{future}^{as}=\sum_{i=1}^{n}(pr_i\times P_{future}^{ti})，\ T_{future}^{as}=\sum_{i=1}^{n}(pr_i\times T_{future}^{ti})，\ S_{future}^{as}=\sum_{i=1}^{n}(pr_i\times S_{future}^{ti})，其中$$

pr_i 为各分支的概率。

2) 对于循环结构，如下图 3-2 中 b 所示，有：

$$P_{future}^{as}=\sum_{i=1}^{n}P_{future}^{t1}，\ T_{future}^{as}=\sum_{i=1}^{n}T_{future}^{t1}，\ S_{future}^{as}=\prod_{i=1}^{n}S_{future}^{t1}$$

3) 对于与分支结构，如下图 3-2 中 c 所示，有：

$$P_{future}^{as}=\sum_{i=1}^{n}P_{future}^{ti}，\ T_{future}^{as}=\max(T_{future}^{t1},\ldots,T_{future}^{tn})，\ S_{future}^{as}=\prod_{i=1}^{n}S_{future}^{ti}$$

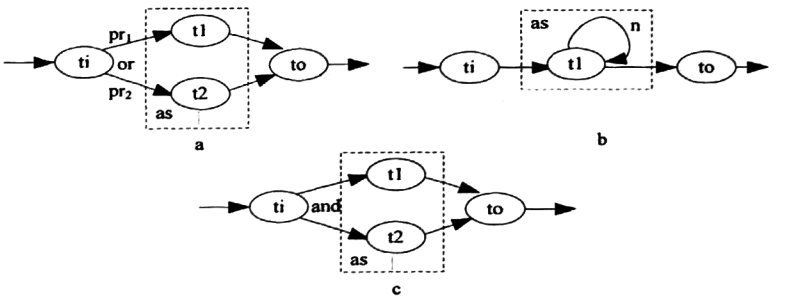

图3-2 三种特殊的控制结构

(3) 计算 QoS_{future}。经过上述处理后，G_{future} 中不再有特殊结构，用 t1 到 tn 标注，此时可以利用公式（3-4）计算，给出 QoS_{future} 各指标的计算公式如下：

$$P_{future}=\sum_{i=1}^{n}P_{future}^{ti}，\ T_{future}=\sum_{i=1}^{n}T_{future}^{ti}，\ S_{future}=\prod_{i=1}^{n}S_{future}^{ti}$$

3.3 组合服务事务恢复中的相关问题

3.3.1 SBET 中事务域的划分

根据第 2.4 节中关于基于域的嵌套事务模型 SBET 的描述可知，域是嵌套模型的关键，域的划分不仅关系到嵌套的层次，而且关系到事务进行错误恢复的粒度。因此，下面对域的划分进行介绍。

组合服务的业务流程图为有向图，用 $G_B(V_B,E_B)$表示。给定一个任务集 X，可以得到 X 在 G_B 的一个有向子图 $G_X(V_X,E_X)$，其中 $V_X=X\subseteq V_B$，E_X 为 V_X 在 G_B 中所有的有向边的集合，$E_X\subseteq E_B$。同理，给定一个域 S，根据域中的任务集，可以得到 S 在 G_B 中的子图 $G_S(V_S,E_S)$。

定义 3-1（连续的）：给定一个任务集 X，称 X 中的任务是连续的，当且仅当 G_X 是连通的。

定义 3-2（始任务、终任务）：在 G_B 中给定子图 G_S，$G_B\neq G_S$，存在 $t_{ss}\in V_S$，$v_B\in V_B$，$v_B\notin V_S$，且$(t_{ss},v_B)\in E_B$，则称 t_{ss} 为 G_S 的始任务。存在 $t_{se}\in V_S$，$v_B\in V_B$，$v_B\notin V_S$，且$(v_B, t_{se}) \in E_B$，则称 t_{se} 为 G_S 的终任务。当 $G_B=G_S$ 时，G_S 的始任务为 G_B 的起始任务，G_S 的终任务为 G_B 的终止任务。

域的划分规则如下：

规则 1 域之间不能交叉重叠，但可以嵌套。

规则 2 域必须是单入口的，即 G_S 只有一个始任务，但可以有多个终任务。

规则 3 域必须具有一定的意义，即组合服务能对这个域的异常进行有效处理，如忽略、替换等。

规则 4 域中的任务必须是连续的，只有连续的任务或域才能划分为新的域。

规则 5 整个业务流程是一个域 S_B，处于最顶层。

给定一个业务流程，可以根据此规则进行域的划分。以下面的业务流程图为例：

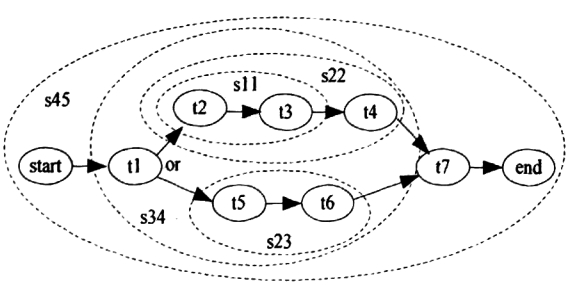

图 3-3 域的划分示例图

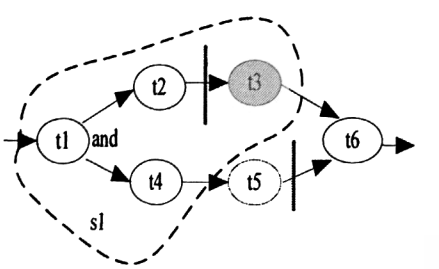

图 3-4 越界示例图

图 3-3 中有 s11、s22、s23、s34、s45 五个域，s11=[t2,t3]，s22=[s11,t4]，s23=[t5,t6]，s34=[t1,s22,s23]，s45=[start,s34,t7,end]，域之间的嵌套关系很明显。

根据上述规则，以及域之间的嵌套关系，可以构造一棵事务树。树的节点为业务流程中的任务事务和域事务。其中任务事务以事务树叶子节点的形式出现，而 S_B 构

成的事务即组合服务事务则是事务树的根。

定义 3-3（越界）：若存在已完成的任务，在业务流程中它处在异常任务的父域的后面，这种情况称为越界。如图 3-4 所示。

3.3.2 事务恢复策略

错误恢复是事务常见的一个问题。事务出现错误时，应能得到及时合理的处理，保证事务能正确执行，或者撤销中止。传统的数据库事务通过日志来回滚，高级事务模型则通过采用补偿技术来撤销事务产生的影响。显然这些措施在组合服务的环境中是不够的。在组合服务事务中，当错误发生时，可采取的基本措施有：重试，替换，忽略，中止，这是事务的四个基本错误恢复策略，RS={rs，rps，is，ds}。基本错误恢复策略和 RPES（见第 2.4.3 节）中的基本事务特性一一对应，也即每一个基本错误恢复策略对应于 RPES 中的一个基本事务特性，其对应关系表如下：

表3-2 基本事务特性与基本错误恢复策略对应关系

基本事务特性	基本错误恢复策略
可重试r	重试rs，重新激活事务直至最大次数
可替换rp	替换rps，用具有相同功能的候选服务替代当前的成员服务
可忽略i	忽略is，跳过当前执行失败的子事务，激活下一个子事务
决定性的d	中止ds，中止整个组合事务，撤销所有能撤销的事务效果

恢复策略是任务或域在发生错误时的恢复方法，目的在于让事务尽量的恢复，从而保证事务的语义一致性。跟事务特性对应，它也可能是多种基本错误恢复策略的组合。在事务发生错误时，事务错误处理模块根据表 3-2 以及事务特性的优先规则，做出正确合理的决策，以确保事务能得到正确恢复。对于具有关键的这一基本事务特性的任务，由于现实世界中没有固定的方式和规则来处理这类任务的错误，因此我们暂时不考虑这种情况。本文假设任何任务都能通过补偿来撤销其影响，也即假设任务必定是可补偿的。

组合服务事务的恢复既要保证事务的语义原子性，又要保证事务语义一致性，同时还得保证服务质量，保证满足用户的需求，因此本文在错误恢复的过程中引入了服务质量 QoS。

3.3.3 补偿事务

定义 3-4（补偿事务）：组合服务的补偿事务是指用来从语义上撤销已执行完毕的子事务所产生影响的事务。

在组合服务事务中，每一个任务都关联一个补偿事务，该补偿事务由当前任务对

应的成员服务的撤消操作组成。如果组合服务业务流程中某个任务执行失败，则必须根据需要调用已经执行完毕的任务的补偿事务，以从语义上撤销其产生的影响，使组合服务事务回退到一致点。

域是一个特殊的任务，它是多个任务的聚合。如果域执行完毕，则它必须有一个能完全的从语义上撤销其产生的影响的补偿事务，称之为完全补偿事务；如果域发生异常，则它必须有一个能从语义上撤销域中已经执行完毕的任务产生的影响的补偿事务，称之为部分补偿事务。

1、域的完全补偿事务

对于已经完成的域，必须是可以从语义上撤销的。当某个域 S 完成时，按照 S 中任务（子域）的执行轨迹的逆序将任务（子域）的完全补偿事务串起来构成 S 的完全补偿事务，如图 3-5 中 a 所示。图中域 s2 是已经执行完毕的，其执行轨迹为(t1, s1, t2)，则它的完全补偿事务为($t2^{-1}$, $s1^{-1}$, $t1^{-1}$)。一旦一个域完成，则其完全补偿事务产生并生效。完全补偿事务由父域启动执行。

2、域的部分补偿事务

对于尚未完成的域，若要进行重试或替换等处理时，必须对域中已经完成的任务（子域）进行撤销。一旦某个域发生异常，则其部分补偿事务产生并生效。当前域的部分补偿事务是根据其已经执行完的任务（子域）的执行顺序的逆序将任务（子域）的完全补偿事务串联起来构成的，如图 3-5 中 b 所示。图中域 s2 在任务 t2 处发生异常，其已经执行的轨迹为(t1, s1)，则它的部分补偿事务为($s1^{-1}$, $t1^{-1}$)。

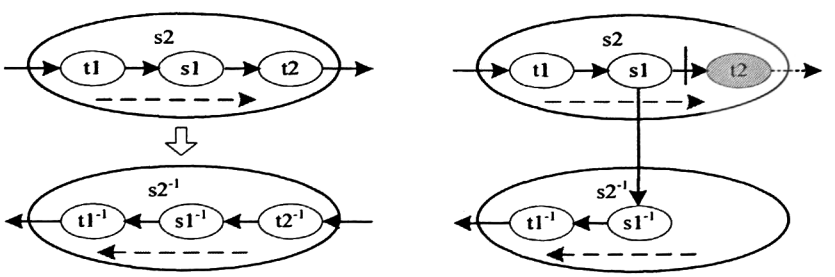

图3-5 域的补偿事务

3.4 组合服务事务恢复算法

3.4.1 算法描述

1、相关子算法

(1) 域的补偿事务的生成：域的补偿事务是根据域中已完成任务（子域）的执行轨迹的逆序以及任务（子域）的完全补偿事务生成的。域的完全补偿事务随着域的完成而生成，而域的部分补偿事务则随着域的错误的发生而生成。

(2) 计算 QoS_{act} 的子算法 calQoS：QoS_{act} 是动态变化的，在未执行的任务节点上

选择不同的成员服务，值不同。当某个任务节点失效时，可以按照第 3.1.2 节中给出的计算步骤和公式对其每一个候选服务计算 QoS_{act}。

(3) 判断是否满足 QoS 约束的子算法 isSat：在选择候选服务时，此算法用于判断所选候选服务能否保证整个组合服务的 QoS_{act} 满足极限 QoS 约束。首先调用 calQoS 计算 QoS_{act} 的各指标值，然后再根据公式（3-1）判断该候选服务是否满足约束条件。

(4) 对 QoS_{act} 进行量化的子算法 estQoS：主要根据公式（3-2）对 QoS_{act} 进行量化，以便替换成员服务时能选取最佳的候选服务。

2、组合服务事务恢复算法

组合服务事务恢复算法基于组合服务嵌套事务模型（见第 2.4 节），有如下前提条件：① 组合服务所选取的初始成员服务能保证组合服务达到 QoS 最优。② 补偿事务能正确执行。③ 不会出现越界的情况。④ 针对成员服务失败的情况。算法的具体描述如下。

算法 3-1：支持 QoS 约束的组合服务事务错误恢复算法 CWERA(ID, QoSLimit)

输入：异常任务 ID，QoS_{limit}

输出：错误恢复决策

(1) curFaultTaskID:=ID

(2) if 当前异常任务为域 then

　　产生部分补偿事务并启动其执行

　else 直接放弃发生错误的成员服务

(3) if 异常任务可以忽略 then

　　错误恢复决策为跳过当前异常任务，算法结束

(4) if 异常任务可以重试 then

　　if isSat(重试) then

　　错误恢复决策为重试，算法结束。

(5) if 异常任务是可替换的 then

(5.1) for 异常任务的每一个候选服务 do

　　if isSat(候选服务) then

　　estQoS(QoS_{act})

(5.2) if 父域存在 and 可以补偿 then

　　for 父域的每个候选服务 do

　　if isSat(候选服务) then

　　estQoS(QoS_{act})

(5.3) if (5.1)中量化的结果不为空 or (5.2)中量化的结果不为空 then

　　if 异常任务某个候选服务的 W(QoS_{act})最大 then

　　错误恢复决策为替换当前异常任务，算法结束

else if 父域某个候选服务的 $W(QoS_{act})$最大　then {

父域产生部分补偿事务并启动其执行

错误恢复决策为替换当前异常任务的父域，算法结束

}

(6) if　curFaultTaskID:=顶层域 ID　then

错误恢复决策为中止组合服务事务，算法结束

(7) if　异常任务为决定性的　then

curFaultTaskID:=顶层域 ID，转向(2)

(8) curFaultTaskID:=父域 ID，转向(2)

本算法能成功的将失效的范围尽可能限制在底层的域内，因为越往上走，要撤销的任务越多，消耗的撤销费用和时间也越多。

3.4.2 正确性分析

本节主要从语义原子性、语义一致性以及算法正确性三方面对支持 QoS 约束的组合服务事务恢复算法进行分析。

1、语义原子性分析

组合服务事务具有语义原子性。根据语义原子性的定义，在组合服务事务的子事务操作失败时，组合服务事务要么继续执行，直至结束，要么失败中止。

因为本算法存在两个出口：

(1) 在事务恢复策略的作用下，满足约束条件时，忽略、重试或者替换任务（域），使组合服务得以恢复，继续向前推进。

(2) 在极限 QoS 约束不了或无处理策略时，异常被抛给父域，直至顶层域 S_B 中止结束，或者直接抛给顶层域 S_B 中止结束，使得组合服务得以成功撤销中止。

这满足语义原子性的定义，所以在组合服务事务中采用此算法，能保证组合服务要么成功执行完毕，要么被有效撤销并中止，从而保证了事务的语义原子性。

2、语义一致性分析

组合服务事务具有语义一致性。根据语义一致性的定义，在组合服务事务发生错误时，能通过补偿回退到一个用户可接受的一致状态，即回退至一个一致点。

因为在本算法中，当任务（域）出错时，组合服务事务可能的回退有两种：

(1) 当父域存在 $W(QoS_{act})$满足极限 QoS 约束且相对最优的候选服务时，回退至父域的起始点；

(2) 若该任务是决定性的，或者在极限 QoS 约束不了或无处理策略时，异常被抛给父域，直至顶层域 S_B，此时，组合服务事务将回退至顶层域 S_B 的起始点，也即组合服务事务的起始点。

显然第一种情况是满足用户 QoS 要求的，是用户可以接受的。对于第二种情况，

因为组合服务事务起始点是事务一致点，所以这也是用户可以接受的。因此本算法能保证组合服务事务的语义一致性。

3、正确性分析

要证明本算法是正确的，必须证明在任务（域）出错的情况下，能得到正确合理的处理。本算法是根据优先规则来进行判断并处理的，其正确性分析如下：

在本算法中，对于任意出错的任务 t，有：

(1) 若 t 为可忽略的，则 t 的执行成功与否不会对整个事务产生影响，事务将跳过该任务，进而执行下一任务。

(2) 若 t 为可重试的，则重试尚未达到最大重试次数，且重试能满足极限 QoS 约束的情况下，重新激活 t 并执行。

(3) 若 t 为可替换的，则在极限 QoS 约束下，若父域不可替换，则选 t 最优的候选服务替换执行；若父域为可替换的，则选 t 和父域中 $W(QoS_{act})$最优的替换执行。

(4) 若 t 为决定性的，则整个组合服务事务回退至事务起始点，执行顶层域的部分补偿事务，以撤销事务产生的效果。

(5) 若不能用以上四种方法进行处理，则 t 的异常抛给父域，然后对父域进行错误处理，直至顶层域 S_B 撤销中止。

因此，本算法能保证 t 在发生错误时得到正确合理的处理，保证组合服务事务要么继续执行，直至执行完毕，要么回退至事务起始点，事务中止。

3.4.3 模拟实验

为了更直接的说明本算法能保证组合服务在出错时，能得到正确合理的处理，本文采用模拟实验的方法进行分析说明。鉴于目前没有相关的标准平台和标准的测试数据集，我们建立了一个 Web 服务模拟环境，利用模拟 Web 服务来模拟实现组合服务的事务恢复算法。

实验环境：Intel P4 CPU 2.4GHz、1G 内存、XP SP2 操作系统、JDK 1.5、Eclipse3.1。

首先设计一个如图 3-3 所示的模拟组合服务，并按照图 3-3 划分域，同时为组合服务中每一个任务准备一个模拟成员服务，为每一个任务以及域节点准备一些模拟的服务候选者。成员服务的 QoS 预先给定，候选服务的 QoS 利用随机数生成器根据当前成员服务的 QoS 随机生成。

实验中，w_P=0.5，w_T=0.2，w_S=0.3，QoS_{limit}={1500，21，0.90}，QoS_{pvd}={1262，17，0.9526}，W(QoS)=0.134978。各任务的事务特性设置，及初始 QoS（成员服务的 QoS_{pvd}）值见表 3-3。初始的成员服务能保证组合服务达到 QoS 最优。

表 3-3 任务的恢复策略设置以及初始 QoS 值

任务名	t1	t2	t3	t4	t5	t6	t7	s11	s22	s23	s34	s45
事务特性	r，rp cp	rp，cp	cp，i	rp，cp	cp，d	r，rp cp	rp，cp	cp，rp	cp，rp	cp，rp	cp，rp	cp，rp
候选服务	9 个	12 个	10 个	8 个	8 个	10 个	11 个	4 个	3 个	3 个	1 个	1 个
初始 QoS	{50,1,0.99}	{190,3,0.93}	{110,2,0.87}	{310,4,0.91}	{250,1,0.99}	{280,5,0.83}	{72,1,0.94}					

对 t1 到 t7 出错的 7 种情况分别进行实验，表 3-4 为利用本文事务恢复算法处理的结果。在任务 t4 处出错，由于不能保证服务质量而被撤销中止。在任务 t5 处出错时，由于该任务的事务特性为可补偿的和决定性的，所以组合服务也被撤销中止。从表中可以看出，本算法能保证在成员服务出错时，得到正确合理的处理，即组合服务要么在保证 QoS 情况下，继续执行直至结束，要么因为不能保证 QoS 或者因为事务特性而成功撤销中止。

表 3-4 事务恢复算法处理的结果

任务名	t1	t2	t3	t4	t5	t6	t7
$W(QoS_{act})$	0.12123	0.13094	0.13486	-0.04623	-	0.13254	0.13394
错误处理策略	重试	替换父域	忽略	中止	中止	重试	替换任务

3.5 小结

本章针对组合服务事务错误恢复过程中保证 QoS 的问题，讨论了组合服务在执行过程中动态 QoS 的计算以及 SBET 中域的划分，并在此基础上提出一个组合服务事务恢复算法。该算法能保证组合服务在错误恢复时，能受极限 QoS 约束，将组合服务事务的失效范围尽量限制在底层的域内，有效减少撤销所带来的开销，保证 QoS 接近最优，并在极限 QoS 约束不了时，最终整个组合服务事务得到撤销，成功中止。

第四章 基于冲突概率的组合服务事务并发控制

在组合服务运行环境中，往往不止一个服务在运行。由于组合服务事务的长事务特点，在多个组合服务事务实例并发执行的环境中，保证各实例之间不受彼此的影响，正确高效的执行是非常重要的。本章将讨论组合服务事务的并发控制机制，以确保在复杂的应用环境中，多个组合服务实例并发执行的正确性和可靠性。

4.1 相关工作

传统的短事务并发控制常采用两段提交协议，然而这对长事务并不适用。一些高级事务模型中提出一种 check out/check in 的并发控制机制[31]，然而这种机制需要所有的事务都具有相同的事务行为，显然这种机制对于跨组织的组合服务并不适用。

文献[32]虽然针对组合服务提出了一种名为 Jenova 并发控制机制，该机制要求一个服务在调度执行前，必须先检查资源是否足够，只有在资源足够的情况下，才能调度执行，但是该文是针对具体资源的，如订房间的服务，检查的是空房的间数，而对于抽象的数据，以及一些相互依赖、冲突的关系，该文并没有给出一个很好的描述，从而不能很好的保证并发的正确性。

文献[33]虽然扩展了 WS-Transaction 规范，提出基于服务提供者提供的依赖关系进行并发控制，但是在依赖关系比较复杂的情况，尤其是碰到运行时间长的事务时，阻塞的事务可能无限期的等待，而且通过协调器来传递并发信息，反而会增加并发控制的复杂性。

文献[32]和文献[33]都是针对具体场合的，要么用于冲突概率较大的场合，要么用于冲突概率较小的场合，不具灵活性。因此本文提出了一种基于冲突概率的混合并发控制算法。该算法能很好的应用于各种复杂的服务环境，服务环境中可以只有同种组合服务的事务实例，也可以同时存在不同种组合服务的事务实例。同时它具有一定的灵活性，既可用于冲突概率较小的场合，也可用于冲突概率较大的场合。该算法充分的发挥了悲观并发控制和乐观并发控制的优点，在保证组合服务事务并发执行的正确性的同时，提高了其并发度。

4.2 事务并发控制相关概念

事务并发是指两个或多个事务在同一个时间内执行，而并发控制则是保证事务并发执行时的正确性和可靠性的一种机制。在组合服务的运行环境中，多个组合服务实例可以并发执行，表现为来自不同组合服务实例的任务实例的交叉执行。

组合服务的一次具体执行叫做组合服务的一个（事务）实例。本文用 CW 表示一个组合服务，用 ct 表示该组合服务一个运行的（事务）实例。

4.2.1 语义单元的定义

组合服务事务具有语义一致性（见第 2.4.2 节）。为了保证事务的语义一致性，事务必须满足语义可串行（Semantic Serializability）[56]。语义可串行性的基础是利用对象的语义信息，也即语义单元（Semantic Unit），符合组合服务事务语义一致性要求。因此本节先引入语义单元的概念。

定义 4-1（语义单元 SU）：设 RS 为所有组合服务的资源集合，SU 为 RS 上的语义单元，是 RS 的子集，有

(1) $RS=\bigcup_{i=1}^{n} SU$；

(2) $1\le i,j\le n, SU_i \cap SU_j = \varnothing$；

(3) $(\forall x \in SU_i, \forall y \in SU_j, i \neq j) \Rightarrow (x \notin \text{dependsOnSet}(y) \wedge y \notin \text{dependsOnSet}(x))$。

其中，dependsOnSet(x)表示资源 x 所依赖的资源的集合，x 依赖于 y 是指对 x 的访问以及修改的结果是资源 y 的值的函数。(3)表明在一个语义单元中的任意资源上的修改只依赖于同一语义单元中资源的值。

4.2.2 冲突及冲突类

定义 4-2（冲突）：对于任务（事务）实例 t_i、t_j，若序列<...t_i...t_j...>的执行效果与<...t_j...t_i...>的执行效果不相同，则称 t_i 和 t_j 是相互冲突的，记做 t_i *conf* t_j。

定义 4-3（语义单元冲突）：任务实例 t_i、t_j，若因为访问同一个语义单元 SU 上的资源而发生冲突，则称 t_i 和 t_j 关于语义单元 SU 相互冲突，记做 t_i $conf_{SU}$ t_j。

若序列<...t_i...t_j...>的执行效果与<...t_j...t_i...>的执行效果相同，则称 t_i、t_j 不相互冲突，记为 t_i *unconf* t_j。

多个组合服务实例并发执行，必然存在冲突的问题。根据冲突的定义，冲突可以归纳为两类：① 因为访问同一语义单元上的资源而引起的冲突，即语义单元冲突；② 因为违反了业务流程中定义的执行顺序而引起的冲突。其中②可以通过业务流程的定义来控制，在并发控制的过程中无需考虑。因此本文只考虑语义单元冲突。

定义 4-3（运行上下文）：运行上下文 $ctx=(CW^*_{ctx}, ct^*_{ctx}, \sigma, tt^*_{ctx}, conf)$，其中 CW^*_{ctx} 为上下文中存在运行实例的组合服务；ct^*_{ctx} 为当前并发的所有组合服务事务实例；σ 为 ct^*_{ctx} 到 CW^*_{ctx} 的一个映射，可以表示为 $\sigma : ct^*_{ctx} \rightarrow CT^*_{ctx}$；$tt^*_{ctx}$ 表示 ct^*_{ctx} 中所有事务实例的所有任务实例集合；*conf* 表示 tt^*_{ctx} 中任务实例间的所有冲突关系，$conf \subseteq tt^*_{ctx} \times tt^*_{ctx}$。可以用一个图来描述上下文，如图 4-3。

设 *pconf* 为来自不同的组合服务实例的任务实例间所有可能发生的冲突关系，则

$conf \subseteq pconf$。

定义 4-4（组合服务事务实例）：组合服务事务实例可以用一个三元组来表示，ct=(ctx,tt_{ct},$\prec$)，ctx 表示当前的运行上下文，tt_{ct} 表示组成 ct 的任务实例集，$\prec$是 tt_{ct} 中任务实例的偏序关系，$\prec \subseteq tt_{ct} \times tt_{ct}$。

定义 4-5（冲突类 confC）：冲突关系具有可传递性，如果 t_i $conf_{SU}$ t_j，t_j $conf_{SU}$ t_k，则 t_i $conf_{SU}$ t_k。利用冲突关系的可传递性，将所有关于语义单元 SU 相互冲突的 t 集合在一起形成一个关于 SU 的冲突类，例如$\{t_i,t_j,t_k\}$就可以构成一个冲突类，冲突类中任何两个元素关于语义单元 SU 相互冲突。

对于任务实例 t，可能存在于多个冲突类中，这些冲突类构成的集合用ω表示。对于同一个任务产生的实例 t_i 和 t_j，有$\omega_i=\omega_j$。若冲突类上有某任务实例正在执行，则该冲突类必需被标记，该标记称为冲突类锁 cl。ct 上的任务实例 t 在调度执行前，必需先检查 ct 是否能获取到ω_t上的所有冲突类锁。只有得到所有锁时，t 才能得到调度执行。

4.2.3　临界区及优先调度规则

定义 4-6（临界区）：任务实例集 CRTSet=$\{t_1,t_2,...t_n\}$构成一个临界区，当且仅当

(1) $t_1,t_2,...t_n \in ct.tt_{ct}$，即 $t_1,t_2,...t_n$ 来自于同一个组合服务实例；

(2) CRTSet 是连续的（见定义 3-1）；

(3) $\exists t_i \in CRTSet(\forall t_k \in CRTSet(i \neq k \wedge t_i \prec t_k)) \wedge \exists t_j \in CRTSet(\forall t_k \in CRTSet(j \neq k \wedge t_k \prec t_j)) \wedge \exists confC\ (t_i \in confC \wedge t_j \in confC)$。

临界区是一个不可分的单元，其任务实例是连续的不间断的执行的。我们用一个三元组来表示临界区，有 CRTSect=(ct, CRTSet, confC, $tSet_{confC}$)，其中 $tSet_{confC}$=CRTSet $\cap$ confC。一个组合服务实例只有获得了其在冲突类 confC 上的锁，才有机会进入临界区执行。一旦进入了一个临界区，则临界区的任务实例的执行结果相对外界是不可见的。图 4-1 展现了 ct1 的临界区，CRTSet={t12,t13,t14,t15}，$tSet_{confC}$={t12, t15}。

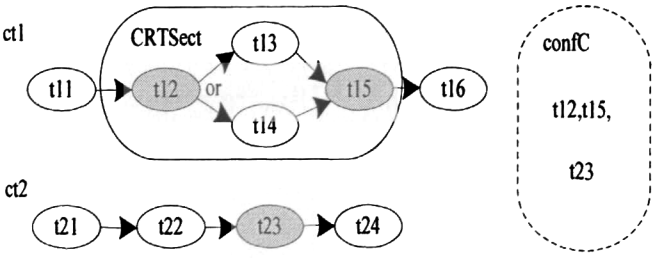

图 4-1 临界区示例图

对于临界区 CRTSect，若$\exists t_i \in CRTSect.confC$，$\exists t_i \notin CRTSect.CRTSet$，且 t_i 与临界区上的任务并发执行，则称之为临界区交叉。以图 4-1 为例，执行序列<t12,t13, t23,t15>会产生临界区交叉，这是不允许的，因为 t15 在 confC 上直接依赖于 t12，而

t23 会改变这种依赖，从而导致执行结果的错误。组合服务实例一旦进入临界区 CRTSect，对于悲观并发控制，会对 CRTSect.confC 上锁，所以不会出现临界区交叉的情况；对于乐观并发控制，在调度执行 CRTSect 中每一个来自 CRTSect.confC 的任务实例前，检测是否有可能出现临界区交叉的情况，从而避免临界区交叉（见 4.3.1）。

为了表示任务实例的调度顺序，设任务实例存在优先级 PRI，PRI 高的任务实例具有优先调度权。当存在多个可以调度执行的任务实例时，PRI 高的任务实例能得到优先调度；若 PRI 值相同则任选一个调度执行。

规则 1 优先调度规则：组合服务实例 ct 若进入临界区 CRTSect，则有 $\forall t_i \in CRTSet$, $\forall t_j \notin CRTSet$ $(PRI(t_i)>PRI(t_j))$，即临界区的任务实例比其他的任务实例的调度优先级高。

优先调度是在 ct 的偏序关系 $\prec$ 的作用下进行的。对于 $t_i \prec t_j$，即使 $PRI(t_i)=PRI(t_j)$，也必定会有 t_i 在 t_j 之前调度执行，因为在 $\prec$ 的作用下，调度模块不会面临 t_i 和 t_j 哪个先执行的选择，所以也不存在 t_i 和 t_j 优先级的比较。

4.3 组合服务事务冲突模型

冲突问题是并发控制的关键所在。为了解决并发控制中的冲突问题，我们建立了一个冲突模型，用于描述各服务间的冲突关系，并在此基础上进行冲突概率的计算。

4.3.1 组合服务冲突模型

组合服务冲突模型用于描述整个服务环境所提供的组合服务之间的冲突关系，是一个抽象的关系模型，文中用 WCM 表示。由此模型结合具体的实例运行场景，可以推导出实例运行上下文，从而得出所有任务实例之间的冲突关系。

定义 4-7（组合服务冲突模型）：WCM 为一个三元组，$WCM=(CW^*,TT^*,\theta)$，其中 CW^* 为服务环境所提供的所有组合服务，TT^* 表示 CW^* 中所有组合服务的所有任务，$\theta \subseteq TT^* \times TT^*$，表示任务事务之间的冲突关系。

下面以两个组合服务 CW1, CW2 为例，用一个图形来描述 WCM，如图 4-2。其中自反箭头表示同种任务的实例会相互冲突；单箭头表示来自不同种组合服务的服务实例的任务实例会相互冲突；双箭头表示来自同种组合服务的不同服务实例的任务实例会相互冲突。从图中可以看出 TT^*={T11,T12,T21,T22,T23}，且 T11,T12 来自于 CW1，T21,T22,T23 来自于 CW2，而任务事务之间的冲突关系 θ ={<T11,T11>, <T12,T12>,<T21,T21>,<T22,T22>,<T23,T23>,<T11,T22>,<T12,T23> ,<T21,T23>, <T23, T21>}。从此关系中可以看出来自同一种组合服务的任务实例间的冲突概率比较大，因为同一个任务的不同实例一定是相互冲突的。

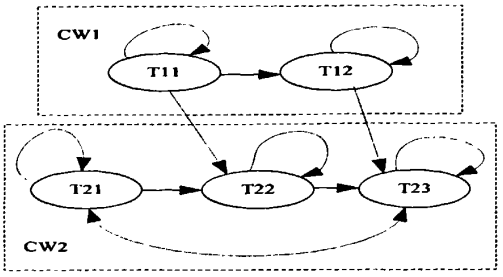

图 4-2 冲突模型例图

WCM 是事务实例运行上下文 ctx 的基础。其相互关系可以归纳为以下三点：

(1) $CW^*_{ctx} \subseteq CW^*$，$tt^*_{ctx}$ 中的任务实例一定是为 TT^* 中的任务产生的，θ表示任务 T 间的冲突关系，*conf* 则表示 T 的实例 t 间的冲突关系。

(2) ctx 是相对于具体的运行环境而言的，是动态的，随着时间的推进而变化；而 WCM 则是基本服务环境，是静态的。

(3) 根据组合服务冲突模型 WCM 以及运行上下文中的 CW^*_{ctx}、ct^*_{ctx}、σ、tt^*_{ctx}，可以推导出当前运行环境的冲突关系 *conf*，具体推导过程见第 4.2.2 节。

定义 4-8（内在冲突）：用于描述一种组合服务的内部冲突关系 *inner*，$inner \subseteq \theta$。在冲突模型中，当 CW^* 中只有一种组合服务时，$\theta = inner$。在图 4-2 中，CW2 的内在冲突关系：*inner*={<T21,T21>,<T22,T22>,<T23,T23>,<T21,T23>,<T23,T21>}，*n*(*inner*)=5。

定义 4-9（交叉冲突）：用于描述任意两种组合服务 CW_1、CW_2 之间的冲突关系 *cross*，$cross \subseteq \theta$。在图 2 中，CW1, CW2 之间的交叉冲突关系 *cross*={<T11, T22>, <T12,T23>}, *n*(*cross*)=2

4.3.2 冲突概率的计算

设运行上下文中所有任务实例间的实际冲突对数用 c 表示，则 c 为 *conf* 中的元素个数，$c=n(conf)$。另假设在该上下文中所有可能的任务实例冲突对数用 C 表示，则 C 为 *pconf* 的元素个数，$C=n(pconf)$，则有任务实例之间的冲突概率为：

$$p = \lambda \cdot \frac{c}{C} \qquad \text{公式（4-1）}$$

其中 λ 为冲突系数，$\lambda = 1 - \frac{1}{n}$，它与该时刻组合服务事务实例的个数 n 有关，n 越大，则 λ 越大，且 $0 \le \lambda < 1$，当 $n=1$ 时，有 $\lambda = 0$。λ 是一个动态因素，其值与实例个数直接相关。在只有一种组合服务的特殊的运行环境里，不管组合服务实例数为多少，$\frac{c}{C}$ 的比值是固定不变的，因此我们引入冲突系数 λ，以保证实例数越多，发生冲突的概率越大。

设当前有 n 个组合服务实例在运行，每个实例所具有的任务数分别为

$m_1, m_2 \cdots m_n$，可以推导出可能发生的冲突对数：

$$C = \frac{m_1\left(\sum_{i=1}^{n} m_i - m_1\right) + m_2\left(\sum_{i=1}^{n} m_i - m_2\right) + \cdots + m_n\left(\sum_{i=1}^{n} m_i - m_n\right)}{2}$$

$$\Rightarrow C = \frac{\left(\sum_{i=1}^{n} m_i\right)^2 - \sum_{i=1}^{n} m_i^2}{2}$$

公式（4-2）

实际冲突对数c的值和冲突模型密切相关。$c = c_{inner} + c_{cross}$，其中$c_{inner}$为所有组合服务实例产生的内在冲突对数，$c_{cross}$为所有组合服务实例产生的交叉冲突对数。设当前运行环境中存在$a = n\left(CW^*_{ctx}\right)$种组合服务，每一种组合服务产生的事务实例数分别为$b_1,b_2,\ldots,b_a$，$b_1+b_2+\ldots+b_a=n$，每种组合服务的内在冲突对数分别为$c_1,c_2,\ldots,c_a$，每两种组合服务间的交叉冲突对数分别为$c_{12},c_{13},\ldots,c_{1a},c_{23},c_{24},\ldots,c_{2a},c_{34},\ldots,c_{(a-1)a}$，则这些组合服务实例产生的所有内在冲突对数：

$$c_{inner} = \frac{b_1(b_1-1)\times c_1 + b_2(b_2-1)\times c_2 + \cdots + b_a(b_a-1)\times c_a}{2}$$

$$\Rightarrow c_{inner} = \frac{1}{2}\sum_{i=1}^{a} b_i(b_i-1)c_i$$

公式（4-3）

这些组合服务实例产生的所有交叉冲突对数：

$$c_{cross} = c_{12}b_1b_2 + c_{13}b_1b_3 + \cdots + c_{1a}b_1b_a + c_{23}b_2b_3 + \cdots + c_{2a}b_2b_a + c_{34}b_3b_4 + \cdots + c_{(a-1)a}b_{(a-1)}b_a$$

$$\Rightarrow c_{cross} = \sum_{j=1}^{a-1} b_j\left(\sum_{i=j+1}^{a} c_{ji}b_i\right)$$

公式（4-4）

下面以图 4-2 中的冲突模型为例来计算C、c的值，假设当前运行上下文中$CW^*_{ctx} = CW^* = \{CW1, CW2\}$，$ct^*_{ctx} = \{ct1, ct2, ct3\}$，其中 ct1 产生于 CW1，ct2、ct3 产生于 CW2，$tt^*_{ctx} = \{t11, t12, t21, t22, t23, t31, t32, t33\}$，则我们可以画出当前上下文的冲突关系图如下：

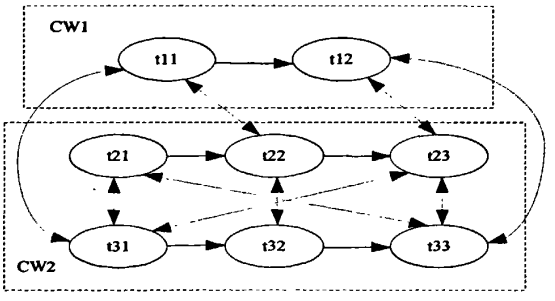

图 4-3 组合服务事务实例冲突关系图

从图 4-3 中可以数出c_{inner}=5，c_{cross}=4，c=9，C=21。利用公式，因为 a=2，

b_1=1，b_2=2，c_1=2，c_2=5，c_{12}=2，可以算出 $c_{inner}=\frac{1\times(1-1)\times 2+2\times(2-1)\times 5}{2}=5$，

$c_{cross}=2\times 1\times 2=4$，c=4+5=9，$C=\frac{(2+3+3)^2-(2^2+3^2+3^2)}{2}=21$，和图中数出的结果一致。此时，由 $p=\lambda\cdot\frac{c_{inner}+c_{cross}}{C}$ 可以得出冲突概率。

本文利用概率论中的古典概型，事件 ρ 的概率是子集 A（样本空间）的元素（基本事件）个数 $n(A)$ 与全集 I（事件域）的元素个数 $n(I)$ 的比值，即 $p(\omega)=\frac{n(A)}{n(I)}$。本文中 *pconf* 为所有任务事务实例间可能发生的冲突，构成了冲突事件 ρ 的全集，*conf* 则是实际发生的冲突集，$conf \subseteq pconf$，因此有 $p(\omega)=\frac{n(conf)}{n(pconf)}=\frac{c}{C}$。同时考虑到组合服务实例数对冲突概率的影响，引入一个冲突系数 λ，得 $p=\lambda\cdot\frac{c}{C}$。

4.4 基于冲突概率的服务事务并发控制算法

4.4.1 组合服务事务的混合并发控制的算法

一组组合服务事务的所有任务序列，在遵循各自组合服务偏序关系的情况下，任意归并成的一个单一序列称为一个调度。若每个组合服务事务的输出都已经知道，则称这样的调度为历史。调度历史表示任务实例的执行顺序，可以用一个列表或一个数组来描述。组合服务事务并发控制的目标就是产生一个正确的调度历史。

1、乐观并发控制算法

乐观并发控制应用于冲突概率很小，甚至不存在冲突的情况。它主要使事务并发执行而很少发生阻塞[57]。传统的乐观并发控制的基本思想是在事务执行完毕时进行有效性确认，如果不存在冲突则提交，否则取消事务。由于提交之前，数据并没有真正的写入数据库中，所以取消事务的操作很容易实现。但对于组合服务，如果执行完毕后才检测到冲突，撤销补偿没法实现，因此我们在临界区进行有效性确认，在确认不会发生临界区交叉后组合服务实例才能继续执行，否则回退至临界区的起始点。

组合服务实例进入临界区 CRTSect 后，对于 CRTSect 中每一个来自 CRTSect.confC 的任务实例 t，在调度执行前，都要检测是否有可能出现在 t 执行完后发生临界区交叉的情况，如果有可能则必须撤销补偿并回退到 CRTSect 的起始点重新开始执行，否则调度 t 执行。乐观并发控制算法描述如算法 4-1 所示。显然，在冲突发生概

率很小或者根本不会发生冲突时，这种方法能大幅度提高服务的并发度，保证系统的效率。但是一旦系统检测可能发生临界区交叉，则要回退并撤销补偿临界区，这是需要一定代价的，尤其是商业性质的服务。

算法 4-1：组合服务事务乐观并发控制算法 OCCA(WCM, ctx)

输入：冲突模型 WCM，当前的运行时上下文 ctx

输出：调度历史 H

(1) 按照业务流程依次调度执行组合服务 ct 中任务实例

(2) if ct 进入一个临界区 CRTSect then{

(2.1) 对于 CRTSect 中来自其冲突类的任务实例 t，在调度运行前，

找出与 ct 并发执行的组合服务实例集 δ

(2.2) for δ 中的每一个组合服务实例 ct_i do

if ct 与 ct_i 有可能相对于 CRTSect 发生临界区交叉 then {

撤销补偿 CRTSect

回退到 CRTSect 的起始点重新执行

}

}

(3) if ct 执行完毕 then

根据任务实例的执行顺序生成调度历史 H，算法结束

else 转向(1)继续执行

2、悲观的并发控制算法

在冲突概率较大的情况下，乐观并发控制会造成大量组合服务实例临界区回退，从而引起对已完成任务实例的大量补偿，代价不可估量，此时乐观并发控制不可行。为了避免过高的补偿代价，必须采取保守的悲观并发控制。它在识别到冲突时，使组合服务实例阻塞，当冲突不再存在时，再调度执行。这就无需补偿，但却牺牲了系统的效率。在极端情况下，可能只有一个组合服务实例在运行，其他所有实例均被阻塞。

本算法采用冲突类锁进行调度控制。冲突类中的任何任务实例，必须获取到其冲突类锁 cl，才有机会执行。对于任务实例同时属于多个冲突类的情况，则只有该实例获取到所有的冲突类锁时，才能调度执行。为了防止死锁，任何任务实例都不能在阻塞的状态持有冲突类锁，即它要么持有所需要的全部冲突类锁，要么一个都不持有。具体算法描述如下。

算法 4-2：基于冲突类锁的悲观并发控制算法 PCCA(WCM, ctx)

输入：冲突模型 WCM，当前的运行时上下文 ctx

输出：调度历史 H

(1) 根据优先调度规则初始化组合服务实例 ct 中任务实例的优先级

(2) 对于 ct 中任务实例 t，在调度运行前，找出其所属的所有冲突类的集合 ω

(3) if 存在优先于 t 的其他任务，then

t 阻塞

(4) if 不存在优先于 t 的其他任务 and ω 为空 then

调度 t 执行

(5) if 不存在优先于 t 的其他任务 and ω 不为空 then{

(5.1) for ω 中每一个冲突类 confC do

if ct 未获得 confC 的冲突锁 cl and confC 没有被其他组合服务实例上锁 then

对该冲突类加锁 cl(confC)

(5.2) if t 没有得到 ω 上所有的冲突类锁 then{

释放已经获得的 ω 上的冲突类锁，并通知其他阻塞中的任务实例

t 阻塞，等待其他组合服务实例释放锁的通知

if 收到释放锁的通知 then

跳转到(3)

}else if t 获取到了 ω 中所有冲突类的锁 then{

更新 t 所在的临界区中所有任务实例的优先级

调度执行至完毕

if 临界区执行完毕 then

释放 ω 上的冲突类锁，并通知其他阻塞中的任务实例

}

}

(6) 当所有任务事务实例执行完时，组合服务事务实例提交完成

根据任务实例的执行顺序生成调度历史 H，算法结束

3、基于冲突概率的并发调度算法

本算法有两个前提条件：① 单个组合服务实例在没有并发控制的情况下一定能正确执行。② 实际冲突概率 p 不会在基准概率 P 的上下频繁波动。

运行上下文 ctx 是由系统维护的，每当有组合服务事务实例开始或者提交结束时，都要更新 ctx。给定 WCM 和运行上下文 ctx，由公式 $p = \lambda \cdot \frac{c}{C}$ 可以计算出当前服务事务的实例化可能导致的实际冲突概率 p。然后比较 p 和 P 的大小，决定采用乐观并发控制还是悲观并发控制。基于冲突概率的并发调度算法详细描述如下。

算法 4-3：基于冲突概率的并发调度算法 MCCA(P, WCM, ctx)

输入：基准概率 P，冲突模型 WCM，当前的运行时上下文 ctx

输出：调度历史 H

(1) 实例化组合服务得到其实例 ct

(2) 更新当前的运行时上下文 ctx

(3) 计算 p 值

(4) if p<P then

调用算法 OCCA(WCM, ctx)

(5) if p≥P then

调用算法 PCCA(WCM, ctx)

关于并发控制方法转换的问题，悲观并发控制和乐观并发控制的相互转换可能会导致因冲突没检测到而出错，尤其是在组合服务运行时间相当长时。为了防止错误，利用乐观并发控制提交时才进行冲突检测的特点，给出下面的约定：① 从乐观到悲观的并发控制转换时，对于已在使用乐观并发控制的组合服务实例继续使用乐观控制方法执行完。② 从悲观到乐观的并发控制转换时，对于已在使用悲观并发控制的事务实例转换为使用乐观并发控制的方法来调度。在前提条件②下，不会出现这两种方法频繁转换的情况。

4、基准概率 P 的选取

从算法 4-3 中可以看出，基准概率 P 直接关系到乐观并发控制算法和悲观并发控制算法的选择。如果基准概率选取不当，可能导致算法 4-3 起不到真正的作用，组合服务实例一直运行在乐观并发控制或者悲观并发控制下。在算法 4-3 中，基准概率是直接作为一个输入参数给出的。下面我们给出一种自适应的基准概率选取方法。

假设在组合服务运行环境中因为乐观并发控制而导致的补偿代价可以衡量，则在一个时间段内，因为检测到冲突而放弃中止的所有补偿代价（简称为单位补偿代价）可以确定。对于一个具体的组合服务运行环境，其所能承受的单位补偿代价是有限度的，我们称这个限度为单位补偿代价的阀值。显然，单位补偿代价超过这个阀值的情况是不允许的。因此，当单位补偿代价超过其阀值时，应该将基准概率调整为当前的实际冲突概率 p。自适应基准概率选择的具体步骤如下：

(1) 置初始基准概率 P=1。

(2) 监控检测单位补偿代价 cc。

(3) 若单位补偿代价 cc 超过其阀值 vv，则置 P=p。

(4) 若 cc-vv<τ（τ 为一个极小的正数），则中止退出；否则转向(2)。

通过这种方法能找到一个最合适的基准概率。当基准概率为 1 时，系统运行在乐观并发控制下。

4.4.2 正确性分析

基于冲突概率的并发调度算法（算法 4-3）是乐观并发控制算法（算法 4-1）和基于冲突类锁的悲观并发控制算法（算法 4-2）的结合，因此对其正确性证明可以分别进行。对乐观并发控制算法主要进行可串行化分析，对悲观并发控制算法则除了可串行化分析外，还得进行死锁分析。

1、可串行化分析

为了判断事务调度是否正确，数据库中普遍采用调度历史的可串行化准则。文献[56]指出不满足可串行性，但满足语义可串行性的历史能够保证事务的一致性，因此在组合服务事务中，可以采用语义可串行性准则来判断调度历史的正确性，即如果组合服务事务的调度历史是语义可串行的，则并发调度是正确的。文献[56]中提出了一种判断调度历史是否具有语义可串行性的方法，下面先简要介绍此方法。

定义 4-10（语义串行图）[56]：对于在事务集 $T=\{T_1,T_2,T_3...T_n\}$上的历史 H，它的语义串行化图记作 SeSG(H)，SeSG(H)=(N,E)，N=T，E 是带标记边的集合，N 中任意两个事务 T_i和 $T_j(i\neq j)$，分别属于 T_i和 T_j的两个任务 p 和 q 对语义单元 SU 中资源的操作发生冲突，并且 p 在 q 之前，则存在一条带标记的边 $T_i \xrightarrow{SU} T_j$。

定理 4-1[56] 历史 H 具有语义可串行性，当且仅当 SeSG(H)中不包含由具有相同标记的有向边构成的环。

文献[56]利用语义串行化图提出并证明定理 4-1 可以判定一个调度历史是否具有语义可串行性。本节将采用该方法来判断组合服务事务实例的调度历史是否具有语义可串行性。下面利用反证法对算法 4-1 和算法 4-2 分别加以证明。

设事务 CST_i在语义单元 SU_k上的任务为 $T^{SU_K}(CST_i)$，CST_i在 SU_k上所有任务为 $AT^{SU_K}(CST_i)$，必有 $T^{SU_K}(CST_i)\in AT^{SU_K}(CST_i)$。

(1) 算法 4-1 产生的调度历史具有语义可串行性。

证明：假设由本算法产生的历史 H 的语义串行化图 SeSG (H) 中包含由具有相同标记 SU_K的有向边构成的环，设为 $CST_1 \xrightarrow{SU_K} CST_2 \xrightarrow{SU_K} CST_3 \xrightarrow{SU_K} \dots \xrightarrow{SU_K} CST_n \xrightarrow{SU_K} CST_1$。

根据本算法，在调度执行临界区 CRTSect 中每一个来自 CRTSect.confC 的任务实例 t 之前，先检测是否有可能出现临界区交叉的情况，如果可能出现临界区交叉则必须撤销补偿并回退到 CRTSect 的起始点重新开始执行，否则调度 t 执行。因此本算法能保证临界区 CRTSect 的连续执行。设临界区 CRTSect 对应的语义单元为 SU_K，则本算法能保证 $AT^{SU_K}(CST_i)$在 $T^{SU_K}(CST_j)$之前或者之后执行。因此对于 $CST_i \xrightarrow{SU_K} CST_j$，根据语义串行图的定义，必有 $AT^{SU_K}(CST_i)$先于 $AT^{SU_K}(CST_j)$执行完，记为 $TME(CST_i)<TME(CST_j)$。

因为 $CST_1 \xrightarrow{SU_K} CST_2 \xrightarrow{SU_K} CST_3 \xrightarrow{SU_K} \dots \xrightarrow{SU_K} CST_n \xrightarrow{SU_K} CST_1$，所以必有 $TME(CST_1)<TME(CST_2)<\dots<TME(CST_n)<TME(CST_1)$，矛盾。

因此本算法产生的调度历史 H 的语义串行化图 SeSG(H)不存在环。因为定理 4-1，所以本算法产生的调度历史具有语义可串行性。

(2) 算法 4-2 产生的调度历史具有语义可串行性。

证明：假设由本算法产生的历史 H 的语义串行化图 SeSG (H) 中包含由具有相同标记 SU_K 的有向边构成的环，设为 $CST_1 \xrightarrow{SU_K} CST_2 \xrightarrow{SU_K} CST_3 \xrightarrow{SU_K} \ldots \xrightarrow{SU_K} CST_n \xrightarrow{SU_K} CST_1$。

对于 $CST_i \xrightarrow{SU_K} CST_j$，根据语义串行图的定义，组合服务事务 CST_i、CST_j 必有任务相对于 SU_k 上的资源冲突，设分别为 T_i、T_j，则 T_i 必在 T_j 之前执行，且 T_i 已经释放了冲突类锁。因为在本算法中，冲突类锁并不是任务执行完就释放，而是在当前任务所属的语义单元上的所有任务都执行完后才释放冲突类锁。所以 CST_j 在语义单元 SU_k 上的任务 T_j 的执行，即 $T_j^{SU_K}(CST_j)$ 的执行，必在 $AT^{SU_K}(CST_i)$ 都执行完毕后才开始，因此必有 $AT^{SU_K}(CST_i)$ 先于 $AT^{SU_K}(CST_j)$ 执行完，记为 $TME(CST_i)<TME(CST_j)$。

因为 $CST_1 \xrightarrow{SU_K} CST_2 \xrightarrow{SU_K} CST_3 \xrightarrow{SU_K} \ldots \xrightarrow{SU_K} CST_n \xrightarrow{SU_K} CST_1$，所以必有 $TME(CST_1)<TME(CST_2)<\cdots<TME(CST_n)<TME(CST_1)$，矛盾。

因此本算法产生的调度历史 H 的语义串行化图 SeSG(H)不存在环。因为定理 4-1，所以本算法产生的调度历史具有语义可串行性。

2、悲观并发控制算法的死锁分析

对于基于锁的并发控制，最关键的问题是死锁。在数据库事务中，若每个事务都在持有一些锁的同时请求另外的锁，可能最终导致相互阻塞。关于死锁的处理，有很多种方法，例如死锁的检测与解除，以及死锁的预防。本算法采取死锁的预防与避免，能有效的防止死锁，下面就本算法中的死锁问题加以阐述。

本文使用的是冲突类锁，主要从以下两个方面来避免死锁：① 采用优先调度规则。② 破坏锁的持有与保持条件。本算法能有效的避免死锁，分析如下：

假设采用本算法出现了死锁问题，必定会有如下情况：存在组合服务实例 ct_1 和 ct_2，ct_1 持有冲突类 $conf_1$ 的锁 cl_1，同时在申请冲突类 $conf_2$ 的锁 cl_2，而 ct_2 持有冲突类锁 cl_2，同时在申请冲突类锁 cl_1，如图 4-4 所示。

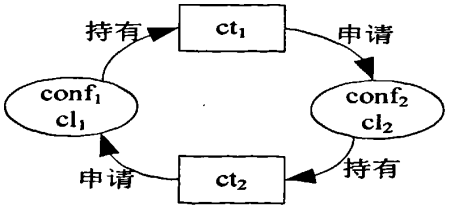

图 4-4 死锁

设 ct_1 开始获得 cl_1 时是任务实例 t_{11} 要调度执行，ct_2 开始获得 cl_2 时是任务实例 t_{21} 要调度执行。ct_1 申请 cl_2 是因为 t_{12} 要调度执行，ct_2 申请 cl_1 是因为 t_{22} 要调度执行。

因为 ct_1 持有 cl_1 并没有释放，说明 ct_1 上必然还有一个尚未执行的任务实例 t_{1i}，$t_{1i} \in conf_1$，且 t_{11} 和 t_{1i} 来自同一个临界区，设为 $CRTSect_1$。同理 ct_2 上必然还有一个尚未执行的任务实例 t_{2i}，$t_{2i} \in conf_2$，且 t_{21} 和 t_{2i} 来自同一个临界区，设为 $CRTSect_2$。

下面根据 ct_1 获得锁 cl_1 和 ct_2 获得锁 cl_2 的先后顺序分两种情况讨论：

(1) 若 ct_1 先持有 cl_1，则根据优先调度规则，必然有 $PRI(t_{1i})>PRI(t_{21})$，也即 t_{1i} 肯定在 t_{21} 之前执行，这与实际情况正好相反。

(2) 若 ct_2 先持有 cl_2，则根据优先调度规则，必然有 $PRI(t_{2i})>PRI(t_{11})$，也即 t_{2i} 肯定在 t_{11} 之前执行，这与实际情况正好相反。

因此不管哪一种情况都是矛盾的，假设不成立，本算法能有效的避免死锁。

4.4.3 实例分析

我们已经证明本文算法的正确性，下面我们通过一个简单的例子来对算法 4-3 进行分析。假设当前环境中有四种组合服务：CW1、CW2、CW3、CW4。图 4-5 展示了组合服务并发控制的一个场景，图中用灰色标注的任务实例关于同一个语义单元冲突，服务请求的顺序通过时间轴显示。从图中可以看出在时间点 Time5 有服务 CW4 的两个服务请求到来，产生实例 ct5 和 ct6。针对这个场景采用算法 4-3 进行并发控制，初始基准概率 P=1，具体并发控制过程如下：

1、在时间点 Time1 处产生服务实例 ct1，由于当前运行环境中只有一个服务实例，p=0<P，此时采用的是乐观并发控制算法。

2、在时间点 Time2 处产生服务实例 ct2，根据公式（4-1）得 p=0.0500<P，继续采用乐观并发控制算法。此时假若出现了 t14、t21、t15 的执行顺序，则在 t23 调度执行之前会被检测出来，因而必须撤销补偿 ct2 的临界区 CRTSect，并回退到其起始点开始执行，ct1 结束。

3、在时间点 Time3 处产生服务实例 ct3，根据公式（4-1）得 p=0.1250<P，继续采用乐观并发控制算法。假若调度执行 t33 之前，检测到执行顺序 t31、t21、t22、t23、t32，发生临界区交叉，此时必须撤销补偿 ct3 的临界区 CRTSect，并回退到 CRTSect 的起始点开始执行，ct2 结束。假设此时单位补偿代价超过其阀值，则置 P=0.1250。

4、在时间点 Time4 处产生服务实例 ct4，根据公式（4-1）得 p=0.0625<P，继续采用乐观并发控制算法。ct3 在 Time5 之前执行完毕，且不存在与 ct4 的临界区交叉。

5、在时间点 Time5 处产生服务实例 ct5、ct6，根据公式（4-1）得 p=0.1333>P，此时转为采用悲观并发控制的算法，每调度一个任务实例执行都必须先获得其所需要的所有冲突类锁突检测。ct4 则继续采用乐观并发控制算法直至执行完毕。

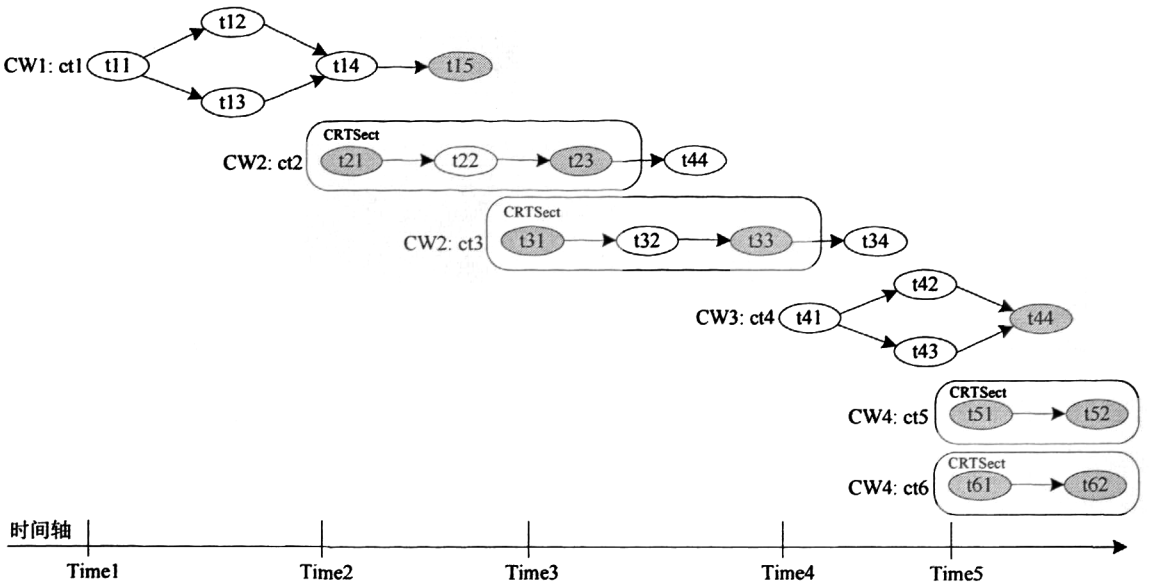

图 4-5 并发控制示例图

4.5 小结

本章提出了一个冲突概率模型，同时提出了一个组合服务事务乐观并发控制算法和一个基于冲突类锁的悲观并发控制算法，并在此基础上提出一个基于冲突概率的混合并发控制算法。此算法充分结合了乐观并发控制与悲观并发控制的优点，当冲突概率较小时，以提高事务的并发度、提高服务的效率以及系统的吞吐率为主，采取乐观的并发控制机制；在冲突概率较大时，以保证组合服务事务调度的正确性为主，采取悲观的并发控制机制。

第五章 支持组合服务事务的原型系统的设计

在第二章中已经提到，WS-BPEL 作为组合服务业务流程描述的规范语言已经成为 OASIS 标准，并为越来越多的人所接受。目前基于 WS-BPEL 的组合服务执行引擎有多种，如 IBM 的 BPWS4J[43]，微软的 BizTalk[58]，以及一些支持 WS-BPEL 的开源项目，如 ActiveBPEL[59]，Twister[60]。虽然它们能提供一些基本的故障处理功能，但是没有明确的事务处理行为，缺乏对组合服务事务的强有力支持，因而限制了其实用性。本章在前面研究的基础上，基于开源的组合服务执行引擎 ActiveBPEL，设计了一个支持组合服务事务的原型系统 TCWS。通过对 ActiveBPEL 进行扩展，添加一个事务管理子系统 TMSS，使之支持组合服务事务。考虑到本文的研究课题是组合服务的事务处理，因此本章重点在 TMSS 的设计。

5.1 TCWS 体系结构

1、ActiveBPEL

组合服务执行引擎 ActivBPEL 是基于 BPEL 标准的的 Java 开源实现，为业务流程提供了一个健壮的运行环境。它能解析 BPEL 流程定义以及相关的一些 WSDL 文件，创建相应的 BPEL 流程实例，同时它还有协议规范处理器，能把数据转化为一种特殊的协议消息，反之亦然。一旦有输入消息到来，引擎就会创建一个新的流程实例，并对其进行管理，同时负责持久化、队列、报警，以及其他一些执行上的细节。ActiveBPEL 引擎运行于标准的 Servlet 容器（如 Tomcat）上，再在 Servlet 容器上部署 AXIS 引擎，作为 SOAP 服务器。同时 ActivBPEL 还有相应配套的 BPEL 流程编辑器 ActiveBPEL Designer，可以利用它来生成符合 WS-BPEL 标准的能被 ActivBPEL 解析的组合服务业务流程。在第二章中，我们对 BPEL 进行了事务支持扩展，因此必须对 ActiveBPEL Designer 进行改进，使它能编辑出支持事务的组合服务业务流程；同时还得改进 ActiveBPEL，使之能识别事务相关的标签，正确解析支持事务的 BPEL 流程。对 ActiveBPEL Designer 和 ActiveBPEL 的具体改进过程本文不做阐述。

2、TCWS

TCWS 是以开源项目 ActivBPEL 的研究为基础的，因此其体系结构的主体部分主要参考了 activeBPEL 引擎的架构，如图 5-1 所示。本文所做的工作是对 activeBPEL 进行扩展，往 BPEL 引擎架构中嵌入一个事务管理子系统 TMSS，图中灰色部分所示。TCWS 主要由五部分组成：Web 应用服务器、SOAP 引擎、BPEL 执行引擎、事务管理子系统 TMSS 和数据库，其中 BPEL 执行引擎是核心，由 ActiveBPEL 提供。

BPEL 执行引擎，根据 ActiveBPEL 的官方文档，又可细化为三个部分：执行引

擎 Engine，流程 Processes 以及活动 Activities。Engine 协调一个或多个 BPEL 流程的执行，而流程则由多个活动构成。另外，它还有辅助模块：处理模块 Handlers 和管理器 Managers，为 BPEL 流程的执行提供相关的一些辅助处理功能，例如处理模块提供计时服务、伙伴绑定等，而管理器则提供流程部署、流程状态等的管理。

AXIS（Apache Extensible Inteaction System）实质上是 SOAP 服务器，一般部署在 Web 应用服务器（如 Tomcat）上运行。它提供创建服务器端、客户端和网关 SOAP 操作的基本框架，能对 SOAP 消息进行解析与封装，同时可以通过基于时间的 SAX 对 XML 文档进行处理，从而可以获得较好的速度和效率。

TMSS 是扩展的模块，提供事务处理功能，主要包含事务的管理、协调、监控、并发控制以及错误恢复等功能，下文将做详细阐述。

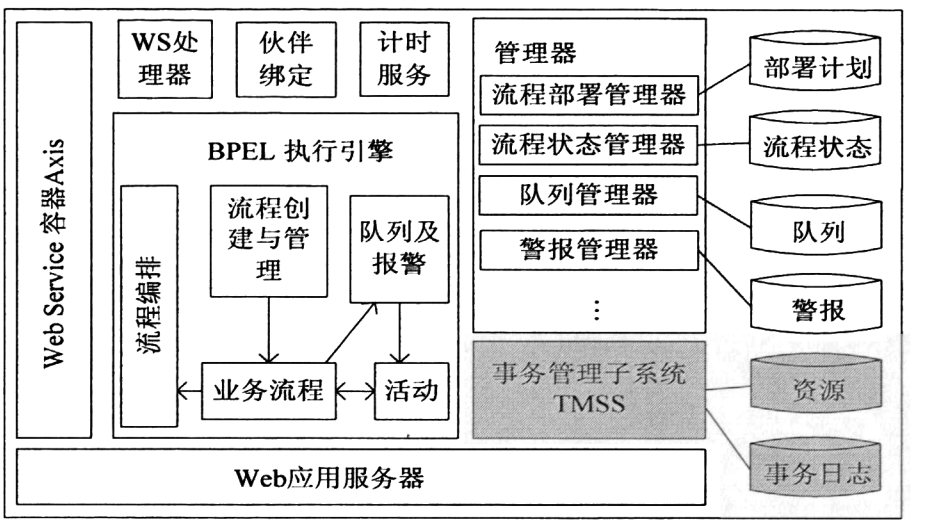

图 5-1 TCWS 系统结构

5.2 TMSS 总体设计

5.2.1 总体结构

TMSS 的总体结构如图 5-2 所示。该子系统主要由事务管理器、协调器、并发控制模块、错误恢复模块，以及事务监控模块组成。其中事务管理器是核心部件，它总揽事务处理全局，决定事务的启动和结束。其次是协调器，负责各事务参与者的协调。协调器是 WS-Transaction 规范下协调模型的实现，可支持规范中的两种事务类型 AT 和 BA。并发控制模块和错误恢复模块则分别负责事务处理的关键问题并发控制和错误恢复，并发控制模块保证多个组合服务事务实例并发执行的正确性，错误恢复模块则保证在出错时，能及时恢复组合服务事务，保证其语义一致性。事务监控模块主要用于组合服务事务状态的浏览和查询，同时为工作人员介入事务提供接口。工作人员既可以通过此模块来查询事务当前所处的状态、事务参与者相关信息，还能通过此模块直接回退或者取消退出事务。

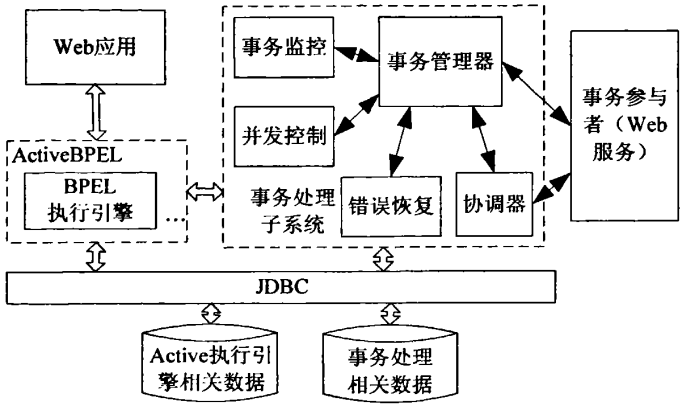

图 5-2 事务管理子系统 TMSS 总体结构

当 Web 应用服务器启动时，启动 ActiveBPEL 执行引擎，同时启动事务管理器。图 5-3 是基于业务一致协调器完成协议（见 5.2.3 节）的事务处理时序图，不考虑错误和并发的情况。在组合服务 A 的请求下，事务管理器 A 创建事务，并向协调器 A 发送创建协调上下文的消息。协调器 A 收到此消息后生成相应的协调上下文对象。在事务参与者 B 向协调器 A 注册之后，事务管理器 A 向协调器 A 发送协调的事务命令，此时协调器 A 使用业务一致协调器完成协议协调多个事务参与者完成事务操作。最后，事务管理器 A 将事务处理结果返回给组合服务 A。

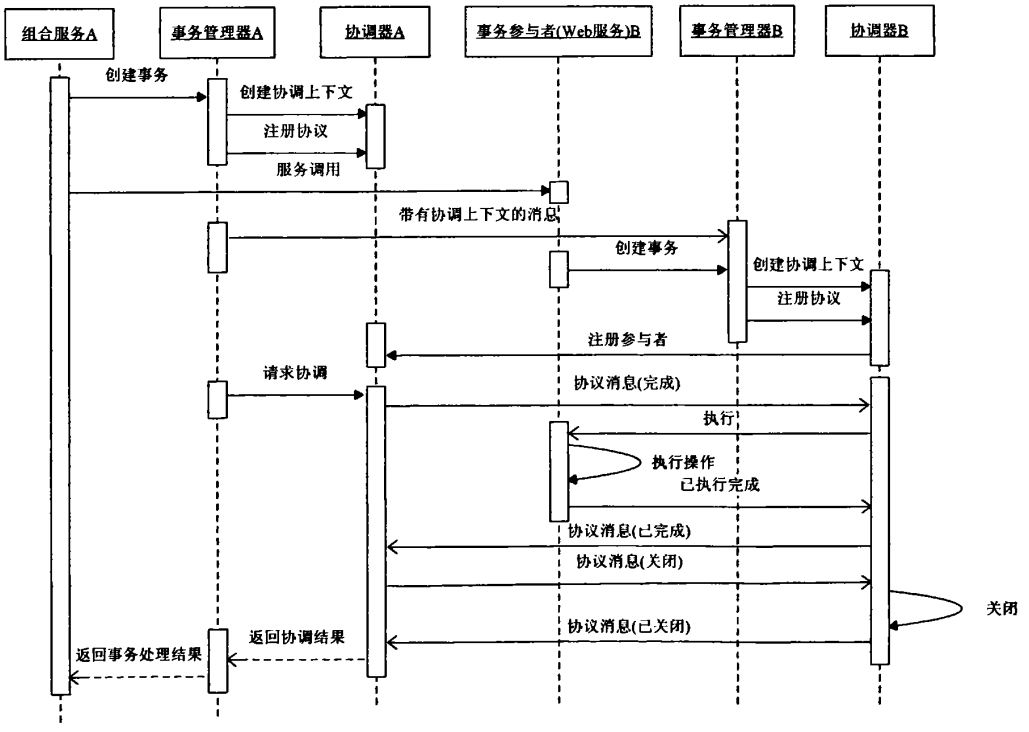

图 5-3 组合服务事务处理时序图

5.2.2 数据结构设计

组合服务事务相关的主要数据有：事务上下文、协调上下文、协议消息、参与者。对这些数据的数据结构设计如下：

1、事务上下文

(1) 事务 ID：事务的唯一标识。

(2) 父事务 ID：存储父事务 ID。

(3) 事务状态：记录事务当前的状态，事务的状态包括初始状态、激活状态、中止状态、已完成状态、失败状态、取消状态、已补偿状态、结束状态等等，还包括完成中状态、补偿中状态等等中间状态。

(4) 开始时间和结束时间：记录事务的起止时间。

(5) 协调上下文：记录用于协调的相关信息。

(6) 参与者列表：记录事务的所有参与者。

(7) 协调协议：用于协调的协调协议，事务在创建时需指明其使用的协调协议。

2、协调上下文

(1) 协调上下文 ID：协调上下文的唯一标识。

(2) 超时值：协调消息的超时值。

(3) 协调类型：协调时使用的协调类型，协调类型有：原子事务、业务活动。

(4) 协议服务地址：协调时使用的协议服务程序的端口地址。协调器中有一组协议服务程序，创建协调上下文时根据协调协议选择对应的协议服务程序。

(5) 注册服务地址：参与者通过注册服务地址向注册服务注册。

3、协议消息

(1) 协调上下文 ID：协议消息对应的协调上下文的 ID。

(2) 消息内容：组合服务的协议服务程序与成员服务的协议服务程序之间交互的具体消息内容，例如 complete、completed 等等。

(3) 协调状态：记录参与者或者协调者在协调过程中的状态，包括注册状态、开始状态、投票状态、提交状态、回退状态和结束状态等等。

4、参与者

(1) 参与者 ID：参与者在事务中的标识。

(2) 事务 ID：对应的事务的 ID。

(3) 事务角色：事务角色分两种，事务参与者，事务协调者。参与者的事务角色为事务参与者。

(4) 协调状态：记录参与者在协调过程中的状态，包括开始状态、投票状态、提交状态、回退状态和结束状态等等。

5.2.3 事务协调协议

本文在第二章中对 WS-Transaction 规范做了简要的分析，提到 WS-Transaction 规范中定义了两种事务类型：原子事务和业务活动，每个事务类型都有自己的协调协议。TMSS 的协调器是 WS-Transaction 协调框架的实现，因此下面先对这些协调协议做详细介绍。

1、原子事务协议

原子事务协议主要用于处理短期存在的活动。WS-AtomicTransaction 规范为 Web 服务原子事务定义了两种协议：完成协议、两段提交协议。

完成协议：应用程序使用这个协议来提交或者终止一个原子事务，事务结束后，状态码返回给应用程序。图 5-4 中 a 为在完成协议控制下的事务状态转换图。图中的节点代表事务的状态。其中 aborting 和 completing 为中间状态。在此协议下，事务的提交与回退由事务管理器决定，事务管理器向协调器发出提交或回退的消息后，协调器才开始提交或中止。

两段提交协议：此协议用来协调参与者以便协调器做出提交或回退的决定。跟完成协议不同，完成协议的提交或回退是由事务管理器直接做出。两段提交协议包括可变两段提交协议和持久两段提交协议，其中可变两段提交协议针对缓存之类的可变资源，持久两段提交协议则针对数据库之类的可持久化资源。同时注册可变两段提交协议的参与者不一定能收到最终结果的通知，而注册持久两段提交协议的参与者一定能收到最终结果的通知。两段提交协议的状态图如图 5-4 中 b 所示，图中每一个节点代表事务的一个状态。

2、业务活动协议

WS-BusinessActivity 规范为 Web 服务业务活动定义了两种协议：业务一致参与者完成协议和业务一致协调器完成协议。鉴于业务活动运行时间较长的特点，这两种协议都引入了补偿机制和故障处理机制。其区别在于参与者的自主性，前者参与者知道主动执行（当有服务请求的情况下），而后者则依赖于协调器告诉它开始执行操作。

业务一致参与者完成协议：参与者在 active 状态时就已经处于执行状态，执行完再通知协调器，由协调器决定事务提交还是补偿。不管怎么样，事务都会进入最终的终止状态。图 5-4 中 c 为业务一致参与者完成协议的事务状态转换图。

业务一致协调器完成协议：参与者在 active 状态时并未执行，而是在等待协调器的通知。一旦收到完成的通知，参与者就开始执行，执行完再通知协调器，由协调器决定事务提交还是补偿。不管怎么样，事务都会进入最终的终止状态。图 5-4 中 d 为业务一致协调器完成协议的事务状态转换图，从图中可看出后面的部分和业务一致参与者完成协议的协调过程是一样的。

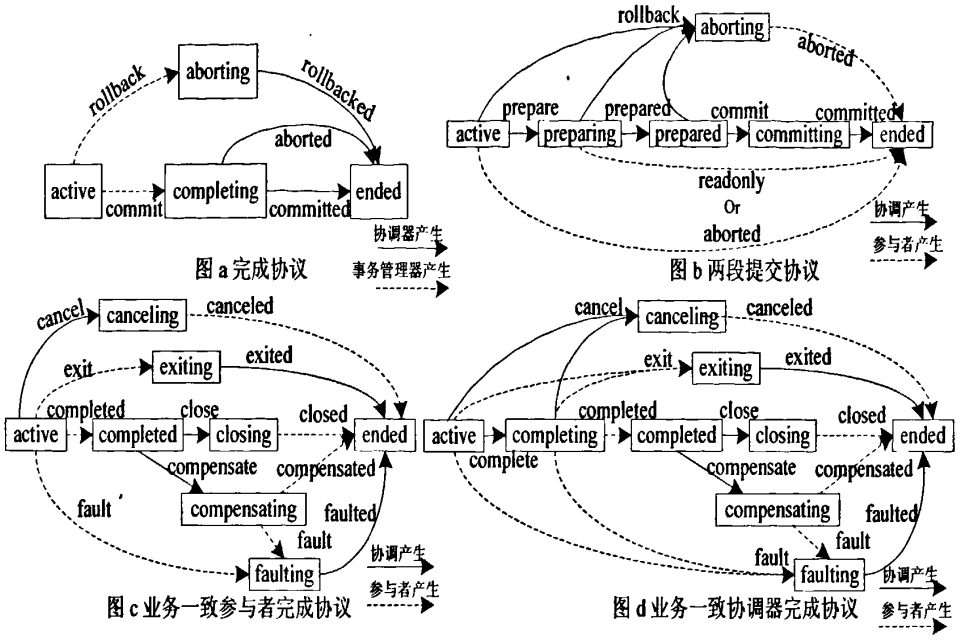

图5-4 WS-Transaction中各协议状态转换图

5.3 TMSS 关键部件设计

5.3.1 事务协调器设计

WS-Coordination 规范中提出了一个可扩展的协调框架，此框架主要提供三种服务：激活服务，注册服务，协议服务。

1、激活服务（Activation Service）：定义创建协调上下文的操作，用于开始一个新的事务并指定该事务可以使用的协调协议以及注册服务的地址。协调上下文可以在组合服务和成员服务之间传递。

2、注册服务（Register Service）：定义注册操作以保证 Web 服务通过注册以参与协调。协调器只能对注册了的参与者按照注册的协议进行协调。注册的过程主要是交换组合服务的协议服务地址和成员服务的协议服务地址的过程。

3、协议服务（Protocol Service）：用于协调已注册的事务参与者完成事务处理。前面已经介绍详细了 WS-Transaction 规范的所有协调协议，每个协调协议对应一个协议服务，提供完成相应的协调行为的操作。

根据协调框架，协调器必须包括一个激活器、一个注册器和一组协议服务程序，分别提供激活服务、注册服务和各种协议服务。每一个协议服务程序都对应于一种特定的协调协议，提供相应的协议服务，以规范事务参与者和协调器的行为。组合服务通过它自己的协议服务与其参与者的协议服务通信，从而实现协调过程。图 5-5 展示了协调器的功能及其处理流程。

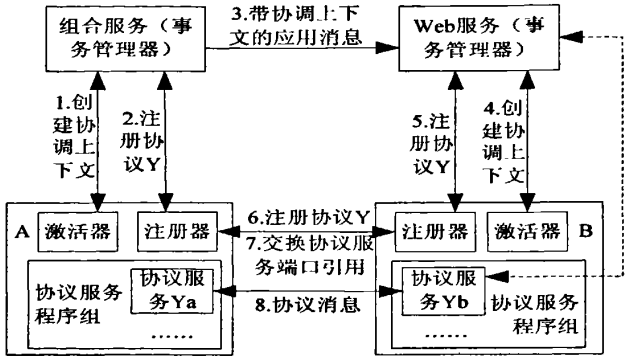

图5-5 协调器的事务处理流程图

图5-6为协调器的类图，展示了协调器以及其内部各组件的关系。其中Coordinator主要负责事务处理的相关操作。ProtocolService是相关协议的实现，按照指定协议提供协调行为，并能传送相关的协议消息。Coordinator的事务处理接口主要是为事务管理器提供的，使得事务管理器具有对事务的整体决策权，如提交、回退、取消等。而ProtocolService则提供事务具体的协调行为，同时为协调器提供一个总的协调入口。

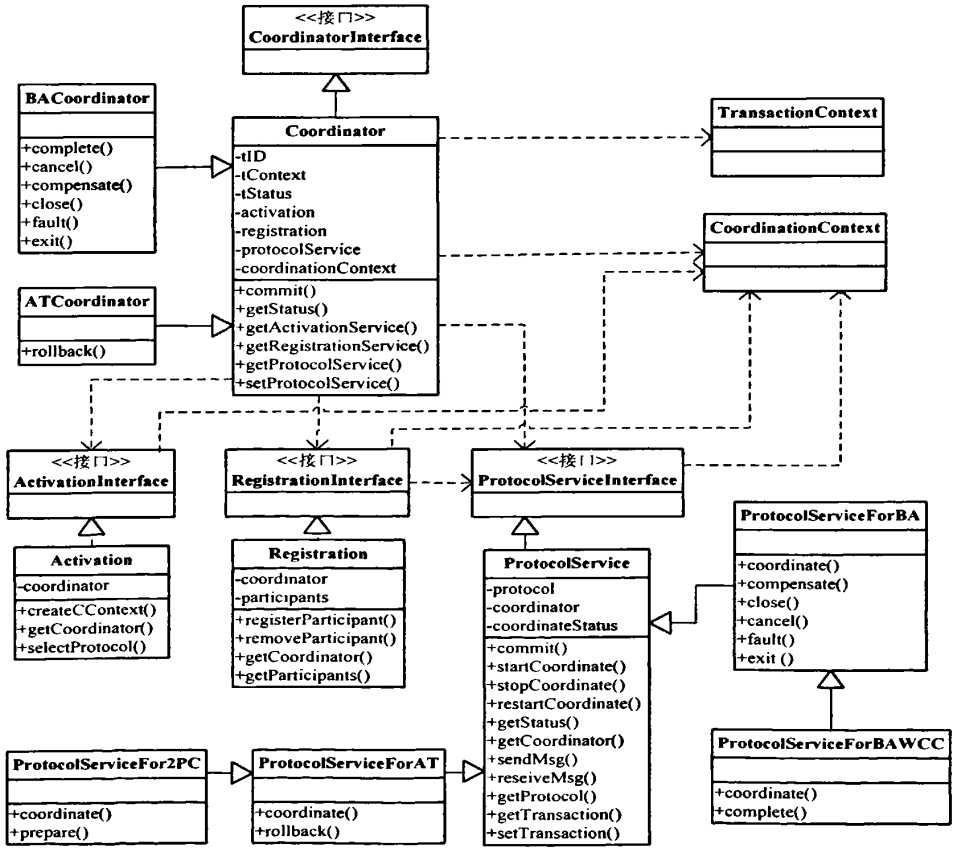

图5-6 协调器类图

5.3.2 事务管理器设计

事务管理器总揽全局，联系着应用程序、事务协调器、并发控制模块、错误处理模块，以及事务参与者，主要负责事务的创建以及事务的管理，并且能控制全局事务边界，决定事务的提交或者回退，为应用程序提供事务接口。它能向事务协调器下达事务处理的命令。至于具体的事务协调行为，由协调器负责，而事务的并发控制以及错误恢复则分别由并发控制模块和错误处理模块负责。应用程序通过事务管理器开始以及结束一个事务。事务管理器维护着一个正在运行的事务列表，存储着每一个事务的当前状态，并且随着时间的推进不断更新。

事务管理器是 TMSS 的核心，在为组合服务应用程序和事务监控模块提供事务处理接口的同时，还依赖于协调器、并发控制模块以及错误处理模块来完成事务行为。图 5-7 展示了事务管理器相关的一些重要类。图中 createTransaction 方法能创建事务，若传入父事务上下文则还能创建子事务。beginTransaction 方法能开始事务，事务管理器向协调器请求创建协调上下文。endTransaction 方法表示事务结束，事务管理器向协调器请求协调，协调器负责协调处理，并返回事务处理结果。block 和 unblock 为并发控制模块提供阻塞和解阻塞的方法。同时事务管理器还为应用程序、错误处理模块提供控制事务的方法，如 rollback、commit 等等。

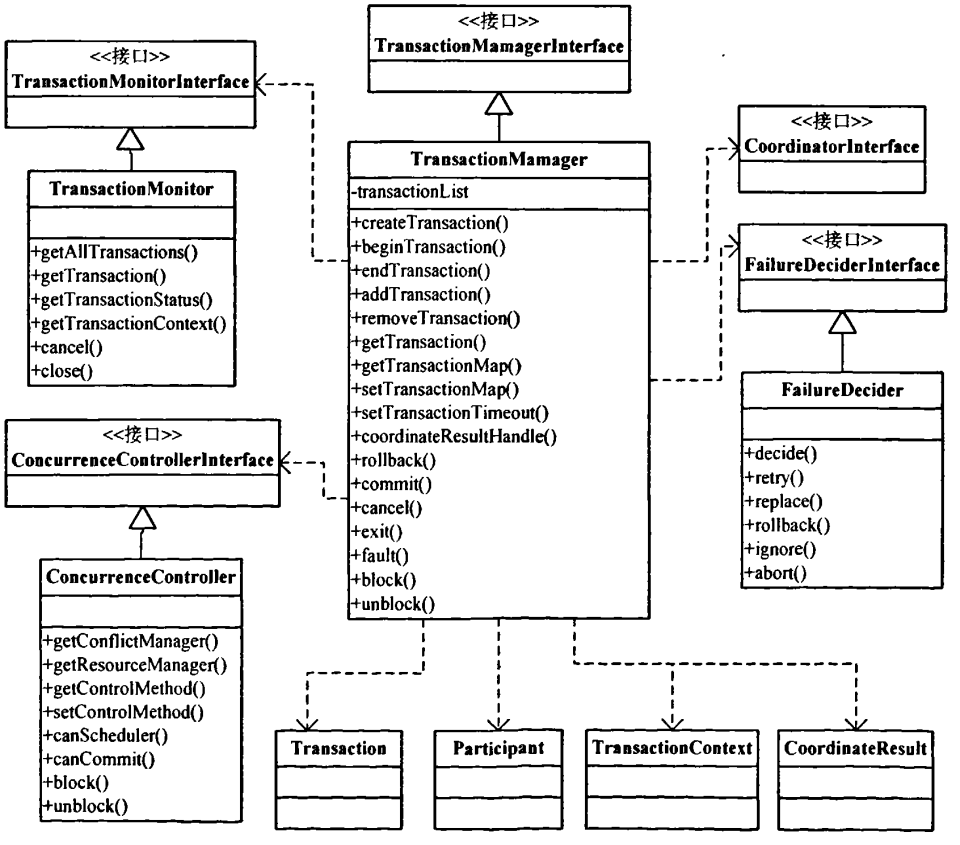

图5-7 管理器类图

5.3.3　错误处理模块

第 1.1.2 节中提出了组合服务可能的三种错误，其中组合服务运行平台错误是由组合服务执行引擎引起的，可以通过不断完善 activeBPEL 来减少其发生。网络环境故障主要指网络设备连接错误，它和成员服务失败都可以通过错误处理模块来处理。当网络设备连接发生错误或成员服务执行失败时，错误处理模块根据错误类型，做出事务处理决策。本文在第三章中针对成员服务失败，提出了一种支持 QoS 约束的组合服务事务恢复算法。错误处理模块在此算法的基础上，兼顾网络设备连接错误。

总的来说，错误处理模块主要由五部分组成：QoS 的计算程序、事务决策器、补偿事务管理、定制服务、设备连接检测程序。其中 QoS 的计算程序主要负责 QoS_{act}、QoS_{future} 的计算；错误决策器负责分析错误信息，做出事务处理决策，在保证 QoS 的前提下，尽量让事务继续向前执行；补偿事务管理模块负责补偿事务的生成和执行；定制服务提供候选服务和补偿任务的定制接口，用户可以通过该接口定制候选服务和补偿任务，这增加了错误恢复的灵活性，以保证业务流程能尽量向前执行，直至结束；设备连接检测程序通过响应时间来检测网络设备是否正常，若在规定时间内无响应消息返回，则通知决策器做出阻塞或取消的决策。

错误处理决策不同，处理流程也不尽相同，下面以替换父域的处理决策为例阐述，时序图如图 5-8 所示。成员服务执行失败，错误信息通过组合服务的事务管理 A 传送给错误决策器 A，错误决策器 A 做出替换父域的决策后，要求补偿事务管理 B 生成部分补偿事务并启动执行，同时把错误决策信息返回给事务管理器 A，此时事务管理器 A 会调用组合服务的相关接口以回退到父域的起始点。

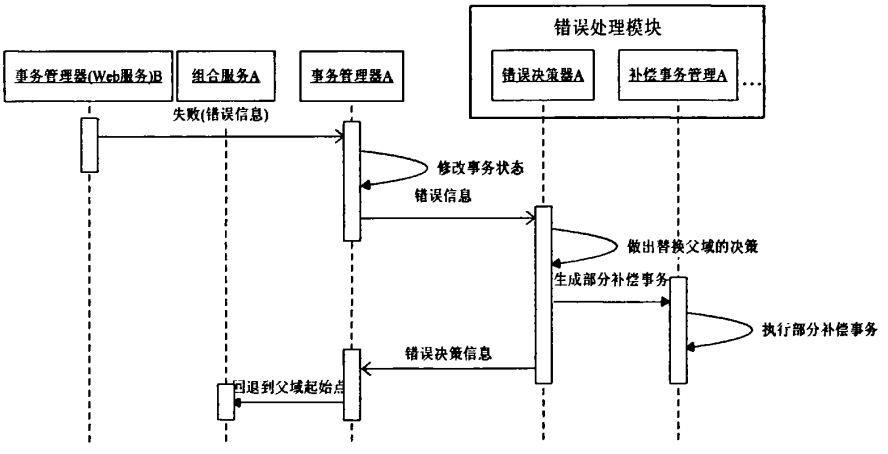

图 5-8 错误处理时序图

5.3.4　并发控制模块

当组合服务事务实例个数多余一个时，实例之间可能会发生冲突，从而导致事务

运行结果的不一致。并发控制模块是基于冲突概率的并发控制调度算法（见第四章）的实现，主要包括以下四个子模块：

1、运行上下文管理，管理运行上下文的生成以及更新。随着旧事务的结束、及新事务的产生，当前并发的组合服务事务实例会发生变化，从而使得冲突关系也发生变化，因此运行上下文在事务的运行过程中是动态变化的，需要维护与管理。

2、概率监控，监控运行环境中冲突概率的变化。当有组合服务实例生成或者结束时，根据运行上下文和组合服务冲突模型计算冲突概率，将其和基准概率比较，以确定要采用乐观并发控制还是悲观并发控制。

3、冲突管理，提供冲突类、冲突类锁、冲突模型的管理维护，同时提供语义单元以及临界区的管理维护。随着事务的推进，冲突类、冲突类锁、语义单元、临界区都是动态变化的，冲突管理对其进行管理维护，从而为调度模块提供调度的依据。

4、调度模块，控制组合服务事务的调度，能开始、提交、中止、阻塞及解阻塞组合服务事务。若当前采用的是乐观并发控制，则在组合服务执行完毕后，根据临界区交叉情况判断是提交结束还是中止；若当前采用的是悲观并发控制，则在调度执行任务实例之前，根据优先调度规则及冲突类锁的持有情况判断是调度执行还是阻塞。

图 5-9 展示了并发控制各子模块及它们之间的关系。从图中可以看出，调度模块是并发控制的核心，其它的子模块最终都是为调度模块服务的。冲突管理和概率监控都依赖于运行上下管理为其提供相关数据。调度模块则依赖于冲突管理和概率监控。同时它提供和事务管理器通信的接口，事务管理器要开始一个事务或者提交一个事务都要通过该接口征得调度模块的“同意”。

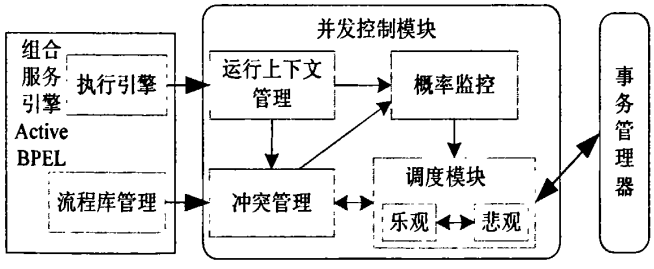

图 5-9 并发控制各子模块关系图

5.4 小结

本章基于开源项目 ActiveBPEL，设计了一个遵循 WS-BPEL 规范的支持组合服务事务处理的原型系统 TCWS。通过对组合服务执行引擎 ActiveBPEL 的扩展，增加一个事务管理子系统 TMSS。TMSS 是一个遵循 WS-Transaction 规范的组合服务事务管理子系统，本章对 TMSS 各关键部件的设计进行了具体的阐述。其中协调器是 WS-Coordination 规范的一个实现，而错误处理模块和并发控制模块则是在第三章和第四章的基础上设计的。

第六章 总结与展望

6.1 本文总结

事务机制是保证服务可靠性的有效手段，它能保证多个 Web 服务交互获得正确的执行和一致性的结果。本文在组合服务事务模型、并发控制以及错误恢复等方面进行了探讨，为组合服务的事务性研究提供了新的思路和方法。本文的主要工作可以归纳为以下几点：

1、提出一个基于域的组合服务嵌套事务模型 SBET。该模型进一步放松了事务的 ACID 属性，通过业务流程域的划分，使得子事务在不能处理错误时，将错误抛给父事务处理，从而能有效的保证事务的语义一致性。

2、提出一个支持 QoS 约束的组合服务事务恢复算法。本文在错误恢复时引入了 QoS，使得组合服务在错误恢复的过程中，能保证服务质量接近最优，这也是用户所需求的。当发生错误时，本算法能根据事务特性，做出正确的决策，保证事务能从错误中恢复过来。

3、提出一个组合服务事务并发控制算法。本算法是基于冲突概率，结合乐观并发控制和悲观并发控制的混合并发控制算法，它能充分发挥乐观并发控制与悲观并发控制的优点。同时本文就该算法进行了可串行化分析，证明了其正确性。

4、事务管理子系统 TMSS 的设计。TMSS 是在 activeBPEL 的基础上设计的，当前大部分组合服务执行引擎都不能提供完全的组合服务事务行为。本文选择开源引擎 activeBPEL，在其基础上设计事务管理子系统，从而对组合服务提供事务处理支持。

6.2 进一步研究方向

组合服务的事务性研究是当前 Web 服务研究的一大热点，我们在研究组合服务事务模型、并发控制以及错误恢复的过程中，又发现了一些新的问题。这些问题有待进一步研究：

1、组合服务事务模型的进一步深入研究。本文中的 SBET 是在组合服务的层次上提出的，在做事务决策时，只考虑了成员服务的执行结果，而对成员服务的具体执行情况没有考虑。然而成员服务也可能是组合服务，所以若从全局的角度看，组合服务事务应该是一个分层的嵌套事务。采用分层的嵌套事务事务模型，结合终端用户的需求可以进一步减少事务恢复时的补偿代价，这在本文中并没有研究。

2、组合服务事务调度的优化。并发控制能提高系统的并发度和吞吐率，本文提出了一种混合的并发控制算法，它能并发的调度事务，并能保证并发事务的正确执行，

但是在事务调度的优化方面并没有深入考虑。怎样进一步优化调度，以提高系统效率、缩短响应时间将是进一步研究的重点。

3、组合服务事务的形式化理论研究。本文在形式化理论方面并没有过多涉及。怎样对组合服务事务的相关技术，例如错误恢复、并发控制，进行形式化描述和性质验证有待进一步研究。

4、服务的动态发现问题。本文中的成员服务以及候选服务都是预先绑定的，或者是手工定制的，这会限制组合服务的服务质量。通过动态服务发现，可以实时的发现 QoS 最优的候选服务，从而保证组合服务的服务质量。所以 Web 服务动态发现问题也是下一步研究的内容。

参考文献

[1] Web Services Description Working Group. Web Services Description Language (WSDL). Version 1.2, 2003-03, http://www.w3.org/TR/wsdl12/2003

[2] UDDI Spec. Technical Committee Specification. UDDI Version 3.0, 2002-07, http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm

[3] Simple Object Access Protocol(SOAP). http://www.w3.0 TR/SOAP/

[4] 欧毓毅, 郭荷清, 许伯桐. Web 服务动态组合的研究.计算机应用研, 2006, 4:22~27

[5] 刘方方. Web 服务合成与可用性的若干关键技术研究: [博士学位论文]. 上海:复旦大学, 2007

[6] Sami Bhiri, Olivier Perrin, Claude Godart. Ensuring Required Failure Atomicity of Composite Web Services. ACM International World Wide Web Conference Committee, 2005.138~147

[7] 岳昆, 王晓玲, 周傲英. Web 服务核心支撑技术:研究综述.软件学报, 2004, 15(3): 428~442

[8] F.Tartanoglu, V.Issarny, A.Romanovsky, et al. Coordinated Forward Error Recovery for Composite Web Services. In Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems, 2003.1~10

[9] Interposition, Web Services and the Babel[1] Fish. http://www.hpts.ws/papers/2003/29.pdf

[10] OASIS. Business Transaction Protocol. Version 1.0, 2002-07

[11] BEA, IBM, Microsoft. Web Services Atomic Transactions (WS-AtomicTransaction). Version 1.0, 2005-08

[12] BEA, IBM, Microsoft. Web Services Business Activity Framework (WS-BusinessActivity). Version 1.0, 2005-08

[13] BEA, IBM, Microsoft. Web Services Coordination (WS-Coordination). Version 1.0, 2005-08

[14] Arjuna, Fujitsu, IONA, et al. Web Services Composite Application Framework (WS-CAF). Version 1.0, 2003-07

[15] Arjuna, Fujitsu, IONA, et al. Web Services Context (WS-Context). Version 1.0, 2003-07

[16] Arjuna, Fujitsu, IONA, et al. Web Services Coordination Framework (WS-CF). Version 1.0, 2003-07

[17] Benchaphon Limthanmaphon, Yanchun Zhang. Web Service Composition Transaction

Management. In the 5th Australasian Database Conference, 2004, 27:171~179

[18]唐飞龙, 李明禄, 曹 健. 一个 Web 服务事务处理模型:结构、算法和事务补偿. 电子学报, 2003, 31(12):2074~2078

[19]Sami Bhiri, Claude Godart, Olivier Perrin. A Transaction-oriented Framework for Composing Transactional Web Services, IEEE International Conference on (SCC'04), 2004. 654~663

[20]Michael P. Papazoglou. Web Services and Business Transactions. In: World Wide Web: Internet and Web Information Systems, 2003, 6:49~91

[21]Benjamin A.Schmit, Schahram Dustdar. Model-driven Development of Web Service Transactions, 2005. http://www.infosys.tuwien.ac.at/staff/sd/papers/btw05.pdf

[22]Benjamin A.Schmit, Schahram Dustdar. Towards Transactional Web Services. In: Proceedings of the 7th IEEE International Conference on E-Commerce Technology Workshops (CECW'05), 2005

[23]P.A.Lee, T.Anderson. Fault Tolerance Principles and Practice, vol. 3 of Dependable Computing and Fault Tolerant Systems. Springer-Verlag. 2nd edition, 1990

[24]陈树根, 姜新文, 宋狄. Web 服务事务模型的形式化建模. 计算机应用, 2006, 26(12): 239~241

[25]Zaihan Yang, Chengfei Liu. On the Development of a Multiple-Compensation Mechanism for Business Transactions. WAIM, 2006. 581~592.

[26]Lili Lin, Fangfang Liu. Compensation with Dependency in Web Services Composition. In: Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP'05), 2005

[27]Zaihan Yang, Chengfei Liu. Implementing a Flexible Compensation Mechanism for Business Processes in Web Service Environment. IEEE International Conference on Web Services (ICWS'06), 2006

[28]Debmalya Biswas. Compensation in the World of Web Services Composition. SWSWPC, 2004. 69~80

[29]J.Gray, A.Reuter. Transaction Processing: Concepts and Techniques. San Mateo. CA: Morgan Kaufmann, 1993

[30]An Liu, Liusheng Huang, et al. Fault-Tolerant Orchestration of Transactional Web Services. WISE, 2006. 90~101

[31]Kim W, Lorie R, et al. A Transaction Mechanism for Engineering Design Databases. In: Proceedings of the 10th International Conference on Very Large Databases. San Francisco: Morgan Kaufmann Publishers, 1984. 355~362

[32]Heqing Guan, et al. Jenova: New Approach on Concurrency Control in Web Service

Transaction Management. In: Proceedings of the 2nd IEEE International Symposium on Service-Oriented System Engineering (SOSE'06), 2006

[33]Mohammad Alrifai, Peter Dolog. Transactions Concurrency Control in Web Service Environment. In: Proceedings of the European Conference on Web Services (ECOWS'06), 2006

[34]M.Younas, I.Awan. Efficient Commit Processing of Web Transactions using Priority Scheduling Mechanism. In: Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE'03), 2003

[35]官荷卿. Web 服务事务的研究综述. 计算机科学, 2005, 32(5):13-16

[36]Nanda MG, Karnik N. Synchronization Analysis for Decentralizing Composite Web Services. In: Proceedings of the ACM Symp. on Applied Computing (SAC), 2003. 407~414

[37]许峰, 徐碧云, 黄皓, 等. Web 服务事务中的补偿机制研究与实现. 计算机科学, 2006, 33(7):242~244

[38]李振东, 谢立. Web 服务器群的 QoS 确保及其接纳控制研究. 计算机研究与发展, 2005, 42(4):662~668

[39]D.Hong, T.Suda. Congestion Control and Prevention in ATM Networks. IEEE Network Magazine, 1991. 10~16

[40]I.Awan, D.Kouvatsos. Approximate Analysis of Arbitrary QNMs with HoL Priorities, CBS Buffer Management Scheme and RS-RD Blocking. In: Proceedings of the 18th UKPEW. Glasgow. UK, 2002-07. 15~26

[41]Mauro Andreolini, Emiliano Casalicchio, et al. QoS-Aware Switching Policies for a Locally Distributed Web System, In: Proceedings of the 11th International World Wide Web Conference. Hawai. USA, 2002

[42]Amit Sharma, Hemant Adarkar, et al. Managing QoS through Prioritization in Web Services. In: Proceedings of the 4th International Conference on Web Information Systems Engineering Workshops (WISEW'03), 2003

[43]IBM. BPWS4J. http://www.alphaworks.ibm.com/tech/bpws4j

[44]G.Weikum, G.vosen. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. San Francisco: Morgan Kaufmann, 2001

[45]J.E.B.Moss. Nested Transactions: an Introduction. In: B Bhargava ed. Concurrency Control and Reliability in Distributed Systems. New York: Van Nostrand Reinhold, 1987. 395~425

[46]H.Garcia, Molina, K.Salem. Sagas. ACM SIGMOD Record, 1987, 16(3):249~259

[47]Pu C, Kaiser GE, Hutchinson N. Split-Transactions for Open-Ended Activities. In:

Proceedings of the 14th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann, 1988. 26~37

[48]A. Zhang, et al. Ensuring Relaxed Atomicity for Flexible Ttransactions in Multi-Database Systems. ACM SIGMOD Record, 1994, 23(2):67~78

[49]A Comparison of Web Services Transaction Protocols. http://www.ibm.com/developerworks/library/ws-comproto/index.html?S_TACT=105AGX52&S_CMP=cn-a-ws

[50]OASIS. Business Process Execution Language for Web Services (WS-BPEL). Version 2.0, 2007-05

[51]王勇, 张煜, 尹瑞. Web 服务组合中商业事务处理的研究. 小型微型计算机系统, 2006, 27(1):121~125

[52]BPEL and Business Transaction Management: Choreology Submission to OASIS WS-BPEL Technical Committee. http://www.choreology.com/downloads/2003-08-26.BPEL.and.Business.Transaction.Management.submission.pdf

[53]Bryson J., Martin D., et al. Toward Behavioral Intelligence in the Semantic Web. IEEE Computer, 2002, 25(11):48~54

[54]Liangzhao Zeng, Boualem Benatallah, et al. QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineening, 2004, 30(5):311~327

[55]温嘉佳. Web 服务组合及其相关技术的研究:[博士学位论文]. 北京:北京邮电大学, 2007

[56]Brayner A., Harder T., et al. Semantic Serializability: A Correctness Criterion for Processing Transactions in Advanced Database Applications. Data&Knowledge Engineering, 1999, 31:1~24

[57]Gerhard Weikum, Gottfried Vossen. 事务信息系统并发控制与恢复的理论、算法与实践. 陈立军等译. 北京:机械工业出版社, 2006.71~98

[58]Microsoft. Biztalk. http://www.microsoft.com/biztalk/default.mspx

[59]Active endpoints. ActiveBPEL. http://www.activevos.com/community-open-source.php

[60]Sourceforge. Twister. http://sourceforge.net/projects/wf-twister/

[61]喻坚, 韩燕波. 面向服务的计算—原理和应用, 北京:清华大学出版社, 2006. 115~191

[62]李杰, 唐卫清, 王行刚. Web 服务事务 QoS 管理. 计算机工程, 2006, 32(7):94~96

致谢

首先要感谢我的导师李建华教授。感谢李老师在我的学习、工作、生活上所给予的无私关怀和悉心指导。本文从选题、研究到撰写都是在李老师的悉心指导下完成的，在论文完成之际首先向李老师表示衷心的感谢。李老师严谨的治学态度、渊博的学识、正直向上的人格魅力和高尚的品德更是深深地影响了我的价值观、人生观，让我受用终生，再次向他致以我最诚挚的谢意。

感谢实验室的各位师兄师姐及师弟师妹，感谢你们在这三年中给予我的关心照顾，特别是马华师兄、许甸师兄、郝丽波师姐、夏援师姐，感谢你们对我的论文提出的各种意见及建议，有了你们的帮助，才使得我的工作更加顺利。还要特别感谢实验室的各位同学，和你们一起度过的这三年时光是我今生最难忘的时光，希望我们的友谊长存！

还要感谢我的父母及恋人，谢谢你们给予我的鼓励、关心和支持；感谢其他亲人对我的鼓励和关心。你们的关爱与支持是我的力量之源。

最后要感谢参加论文评审和答辩委员会的各位老师。

再次向所有帮助过我的人们致以诚挚的谢意！

曾慧琼

二零零八年五月于中南大学

攻读学位期间主要的研究成果

已发表学术论文：

[1] 曾慧琼, 李建华, 许甸, 马华. 基于冲突概率的组合服务事务混合并发控制算法. 计算机应用, 2007, 27(10):2433～2436

[2] 李建华, 曾慧琼, 陈松乔. 支持QoS约束的组合服务事务恢复算法. 计算机工程. 拟刊在2008年第14期

[3] 曾慧琼, 李建华. 组合服务事务的异常处理机制研究. 计算机工程与设计, 2008, 29(8):1886～1889

参加的科研项目：

[1] 参与了长沙科创信息技术有限公司的内容管理系统的设计和开发

[2] 参与了长沙科创信息技术有限公司的开源工作流建模工具 JAWE 及引擎 SHARK 的研究和改进

[3] 参与了长沙科创信息技术有限公司的中南大学信息港门户网站的设计和开发

[4] 参与了长沙科创信息技术有限公司的中南大学湘雅遗传学实验室项目的改进与维护