Parlay 网关 TCAP 类协议映射子系统的设计与实现

摘 要

下一代网络的出现是增值业务发展的必然趋势，下一代网络是基于分组交互的网络，它采用了分层的结构体系，分为业务层、控制层、媒体传输层和接入层。在下一代网络体系中，应用服务器位于业务层，它为各种增值业务和智能业务的执行和管理提供环境，也对第三方业务提供商提供开放的业务接口。目前广泛使用的一类开放业务接口是 Parlay API(Application Programming Interface)。在 Parlay 体系结构中，Parlay 网关屏蔽了底层网络的异构性，向应用服务器和第三方应用以标准的 Parlay API 形式提供了对底层网络的控制能力。

Parlay 网关由框架子系统、业务能力服务器子系统、协议映射子系统、网管子系统和操作维护子系统组成。协议映射子系统完成底层网络设备协议的适配功能。TCAP(Transaction Capabilities Application Part)类协议映射子系统是协议映射子系统的一个组成部分，当底层网络是传统的固定或移动智能网时，通过它实现智能网的 TCAP 协议和 TC(Transaction Capabilities)用户协议的适配。TCAP 协议可以分为成分子层和事务子层，在 TCAP 协议上层是 TC 用户应用协议。TCAP 类协议映射子系统在设计中实现了 TCAP 成分子层和 TC 用户协议层的适配。整个子系统可以分为三部分：协议栈适配模块、TCAP 协议处理模块和 TC 用户处理模块。在考虑同步和异步这两种模块间的消息传递机制时，我们采用了后者。并根据 CORBA(Common Object Request Broker Architecture)和非 CORBA 的环境提出两种异步消息传递机制的实现方式：CORBA 异步方法调用和消息队列方式。协议栈适配模块负责和七号信令协议栈交互，实现 TCAP 协议数据单元的收发，同上层模块间是 CORBA 接口。TCAP 协议处理模块分为面向 TC PDU(Protocol Data Unit)的接口、TC 用户的接口和辅助接口，定义了 TCAP 对话原语到 IDL(Interface Definition Language)接口的交互翻译。在 TCAP 消息处理上，根据请求原语和指示原语设计了一对接口用于实现对话的交互规则。TC 用户是用 ASN.1(Abstract Syntax Notation one)定义的，我们参考规范，实现了从 ASN.1 定义的 TC 用户协议到 IDL 定义的接口翻译算法，论文中对于这部分主要讨论了从应用上下文宏到 IDL 接口的翻译方法，并举例说明了 CAP(CAMEL

Application Part)协议处理的实现。然后，本论文分析了系统在底层网络发起对话和应用发起对话两种情形下的处理流程。

我们测试的结果表明 TCAP 类协议映射子系统能够满足设计目标。论文最后提出了系统的不足和下一步的改进工作，展望了下一代网络的发展情况。

关键词：下一代网络，Parlay 网关，TCAP 协议，TC 用户，协议映射，CORBA，ASN.1

THE DESIGN AND IMPLEMENTATION OF TCAP PROTOCOL MAPPING SUBSYSTEM IN PARLEY GATEWAY

ABSTRACT

The development of value-added services leads to the emergence of NGN(Next Generation Networks), which is based on packet switching. NGN adapts the layered structure consisting service layer, control layer, media transport layer and access layer. The Application Server locates on NGN's service layer. It provides environment of executing and managing various value-added services and intelligent services. Besides, it provides the third party service providers with open service interfaces. One of the open service interfaces that applies widely is Parlay API(Application Programming Interface). In the architecture of Parlay, the Parlay gateway screens the heterogeneity of underlying networks and provides the Application Server and third party service providers with the capability of controlling underlying networks in the standardized form of Parlay API.

Parlay gateway is composed of framework subsystem, protocol mapping subsystem, networks management subsystem and operation and maintaining subsystem. Protocol mapping subsystem performs adaption of underlying network devices. TCAP(Transaction Capabilities Application Part) protocol mapping subsystem is one component of the whole protocol mapping subsystem, which provides adaption of TCAP and TC(Transaction Capabilities) user protocols when underlying network is traditional fixed or mobile Intelligent Network. TCAP is divided into two sublayers: component sublayer and transaction sublayer. Above the TCAP, there are TC user application protocols. TCAP protocol mapping subsystem performs the adaption of TCAP component sublayer and TC user protocols. The subsystem is composed of three modules: protocol stack adapter, TCAP protocol processor and TC user processor. We adopt asynchronous mode instead of synchronous one as message

transmission mechanism between any above two modules. And then two asynchronous mechanisms are proposed based on two different situations: CORBA(Common Object Request Broker Architecture) environment and non-CORBA environment. The protocol stack adapter interacts with SS7(Signalling System No.7) stack, sending and receiving TCAP PDU(Protocol Data Unit). The interface with its upper module is based on CORBA. TCAP protocol processor consists of three parts: interface to TC PDU, interface to TC user and an assistant interface. In this module rules of translating TCAP dialogue primitive into IDL(Interface Definition Language) interface are defined. Besides, a couple of interfaces are designed to handle TCAP dialogue's interaction rules. TC user protocol is defined in ASN.1(Abstract Syntax Notation one). We have implemented the algorithm to translate its ASN.1 definition into IDL interface. The article mainly discusses the rules of translating Application Context Macro into IDL interface and takes CAP processing as an example. After that, we analyze the system's processing flows in two situations: dialogues initiated by underlying networks and dialogues initiated by upper applications.

Testing has proved that TCAP protocol mapping subsystem can meet the designing requirement. In the end of this thesis we point out the deficiency of existing system, solutions to these problems, and the prospect of NGN(Next Generation Network) in China.

KEY WORDS: NGN, Parlay Gateway, TCAP, TC User, Protocol Mapping, CORBA, ASN.1

第一章 绪论

1.1. 下一代网络简介

当今电信行业迅猛发展，增值业务的提供能力已经成为众运营商竞争和发展的重点。传统的在交换机上实现业务的方式周期长，成本高，可靠性差，阻碍了电信新业务的进一步推广。智能网(Intelligent Networks, IN)体系结构的提出，将传统交换机的交换功能和业务控制功能相分离，使新业务的生成变得快速、方便、经济、灵活有效，在一定时间内极大程度上解决了运营商和用户的业务需求。但随着技术的发展和用户需求的提高，智能网的业务提供能力越来越不能够满足电信运营商的要求了。主要表现在：

1) 业务生成仍不够灵活，业务开发人员必须花费大量的时间和精力了解网络结构和协议的底层细节来部署智能网业务。
2) 标准协议过于复杂，网络结构不够开放，缺少业务生成、业务管理、业务部署的标准接口，这些都增加了第三方业务提供商进入智能网市场的难度。
3) 传统智能网主要应用于电路交换网络，它提供数据业务的能力非常有限，不适合开展多媒体业务。

从当前电信发展的大趋势看，IP(Internet Protocol)业务将成为未来业务的主体，IP向传统电信业务的渗透和传统电信业务与IP的融合步伐将大大加快，在这样的背景下，下一代网络(Next Generation Networks, NGN)的概念被提出。

下一代网络是一个分组网络，它提供包括话音、数据和多媒体业务在内的多种业务，能够利用多种带宽和具有QoS(Quality of Service)能力的传送技术，实现业务功能与底层传送技术的分离；它提供用户对不同业务提供商网络的自由地接入，并支持通用移动性，实现用户对业务使用的一致性和统一性。它有如下的三个主要特征：

1) 采用开放的网络架构体系。将传统交换机的功能模块分离成为独立的网络部件，各个部件可以按相应的功能划分，独立发展；部件间的协议接口基于相应的标准，接口的标准化可以实现各种异构网络的互通。
2) 下一代网络是业务驱动的网络。业务与呼叫控制分离，呼叫与承载分离；使业务真正独立于承载网络，灵活有效地实现业务的提供
3) 下一代网络是基于统一协议的分组网络。近几年IP的发展，使人们认识

到电信网、广电网和计算机网最终会统一为 IP 分组网络，互联网迅猛发展，开始提供全方位的，开放的，可支持视频和音频的各种业务，而在这方面真是电信网络的固有的缺陷，互联网的发展给我们带来了机遇。无论从技术和业务商，语音网络与数据网络的融合将成为网络发展的必然趋势。

下一代网络的功能模型可以分为四个层次，如图 1-1 所示[1]：

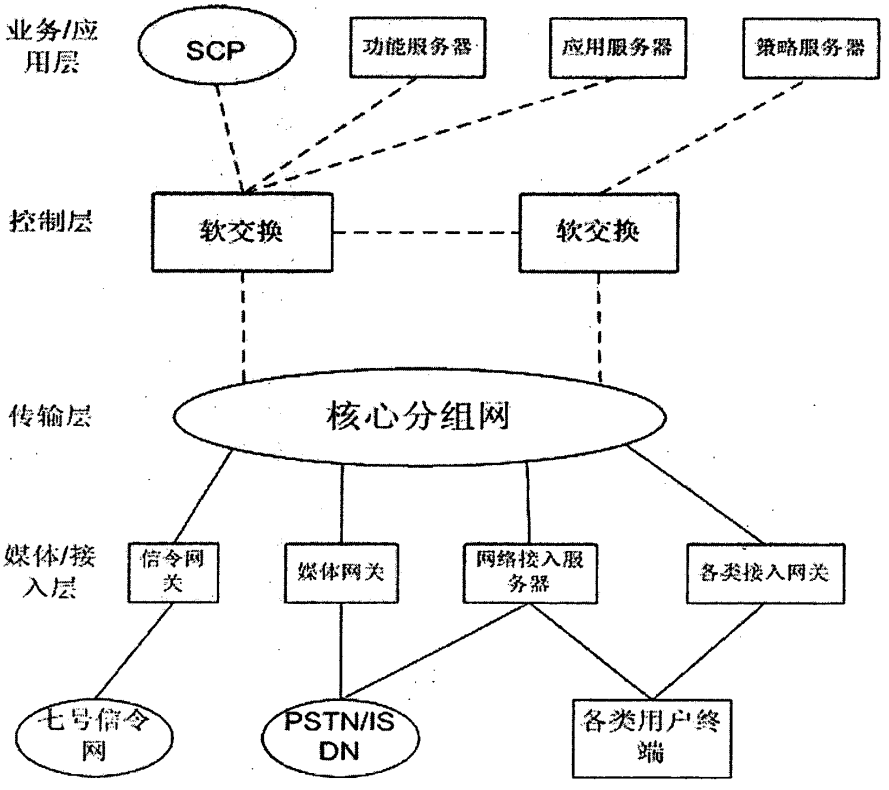

图　1-1 下一代网络的功能模型

1) 接入和传输层：将用户连接到网络，包括集中用户业务并将它们传输到目的地，包括各种接入手段，将原有七号信令网、PSTN(Public Switched Telephone Network)网络接入到下一代网络的功能。
2) 媒体层：将信息格式转换成为能够在网络上传递的格式。例如将话音信号分割成 IP 包或 ATM(Asynchronous Transfer Mode)信元。此外，媒体层可以将信息选路到目的地。
3) 控制层：包括呼叫智能。此层决定用户收到的服务，并能控制底层网络元素对业务流的处理，其核心是软交换系统。
4) 业务/应用层：在呼叫建立的基础上提供额外的服务，由一系列的业

务应用服务器组成，提供各种各样的业务控制逻辑，完成增值业务处理。同时提供开放的第三方应用编程接口 API，易于引入独立于网络的新型业务。

1.2. 软交换简介

软交换是一种功能实体，为下一代网络提供具有实时性要求的业务的呼叫控制和连接功能，是下一代网络呼叫和控制的核心设备。

在传统程控交换机中，"呼叫控制"功能是和业务结合在一起的，不同的业务所需要的呼叫控制功能不同。软交换的基本含义就是把呼叫控制功能从媒体网关(传输层)中分离出来，通过服务器或网元上的软件实现基本呼叫控制功能，包含呼叫选路、管理控制、连接控制(建立会话、拆除会话)、信令互通。其结果就是把呼叫传输与呼叫控制分离开，为控制、交换和软件可编程功能建立分离的平面，使业务提供者可以自由地将传输业务与控制协议结合起来，实现业务转移。

软交换设备的框架结构如图 1—2 所示。软交换具有如下的主要功能：

1) 呼叫控制和处理功能：软交换设备可以为基本呼叫的建立、维护和释放提供控制功能，包括呼叫处理、连接控制、智能呼叫触发和资源控制等。
2) 协议功能：软交换是一个开放的、多协议的实体，因此必须采用标准协议与各种媒体网关、终端和网络进行通信,这些协议包括：H.248、SCTP(Strum Control Transport Protocol) 、ISUP(ISDN User Part)、SNMP(Simple Network Management Protocol)、SIP(Session Initial Protocol)、MGCP(Media Gateway Control Protocol)协议等。
3) 业务提供功能：下一代网络是业务驱动的网络，软交换可以提供基本的交换控制业务，软交换的提供的基本业务通过开放的接口提供给第三方合作,不仅增加了服务的种类，而且加快了应用服务的速度。
4) 业务交换功能：业务交换功能与呼叫控制功能相结提供了呼叫控制功能和业务控制功能(Service Controlling Function, SCF) 之间进行通信所要求的一组功能。
5) 操作维护功能：操作维护系统是软交换设备中负责系统的管理和操作维护的部分,是用户使用、配置、管理、监视软交换设备的工具集合。
6) 计费功能：具有采集详细话单及复式计次功能，并能够按照运营商的需求将话单传送到相应的计费中心。当使用记账卡等业务时，软交换应具备实时断线的功能。

7) 软交换与其他网络的互通：软交换是下一代网络的核心设备，各个运营商在组建以软交换为核心的下一代网络时，其网络体系结构可能有所不同，但必须考虑与其他各种网络的互通，如与现有七号信令网的互通，与现有智能网的互通，与采用 H.323 协议的 IP 电话网的互通等等

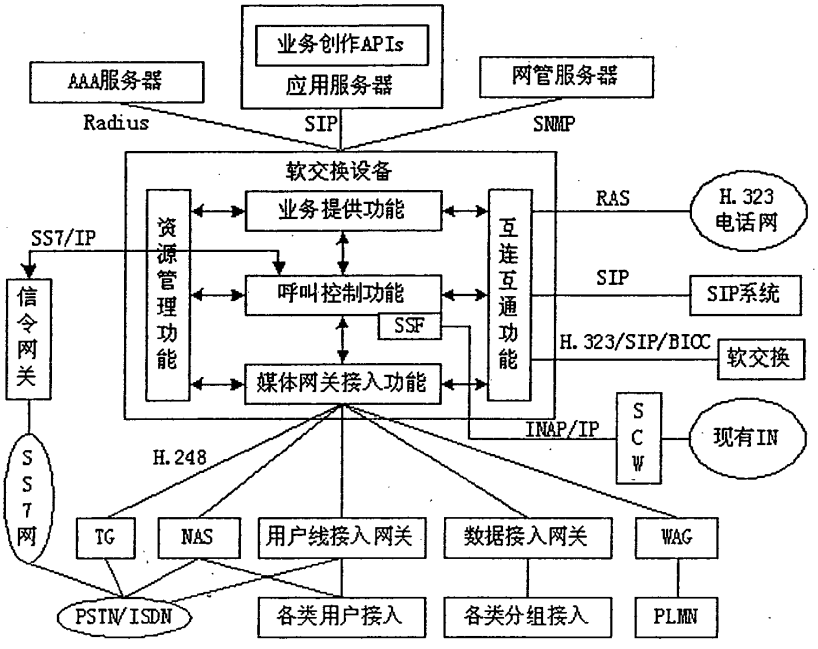

图 1-2 软交换设备的框架结构

1.3. 应用服务器简介

强大的业务提供能力是以软交换为核心的下一代网络的一个亮点，而应用服务器就是专为增值业务而引入的。应用服务器是下一代网络中引入的新的组件，它为各种增值业务和智能业务的执行和管理提供环境，业务可以驻留在应用服务器上，也可以通过应用服务器所提供的 Parlay API 接口接入，为第三方提供业务执行平台。应用服务器与控制层的软交换无关，从而实现了业务与呼叫控制的分离，有利于新业务的引入。

应用服务器能够处理来自软交换网络的 SIP 呼叫，可选地处理智能呼叫，包括来自软交换的智能呼叫、通过信令网关来自 PSTN 网络的 IN 呼叫、来自 GSM(Global System For Mobile Communications) 网络的 CAMEL(Customized Application for Mobile, Network Enhanced Logic)呼叫和

来自 CDMA(Code Division Multiple Access)网络的 WIN(Wireless Intelligent Network)呼叫等。当应用服务器和软交换采用 INAP(Intelligent Network Application Protocol)协议互通提供智能业务时，由软交换控制媒体服务器，提供业务所需的媒体资源。当应用服务器通过信令网关与 PSTN/GSM/CDMA 网络的智能网互通智能业务时，由 PSTN/GSM/CDMA 网的 SSP (Service Switching Point)或 IP(Intelligent Peripheral)提供媒体资源。

根据业务执行的需要，应用服务器还可以使用 SIP 协议、H.248 协议(可选)、MGCP 协议(可选)直接向媒体服务器发出资源调用请求，或者通过软交换控制媒体服务器，为增值业务和智能业务(可选)提供媒体资源。

应用服务器在下一代网络中主要提供如下功能：呼叫控制功能、媒体控制功能、业务数据功能、协议适配功能、计费功能、API 接口功能、Parlay 网关功能、应用执行环境功能、操作维护管理功能。

应用服务器的功能模块如图 1-3 所示[2]：

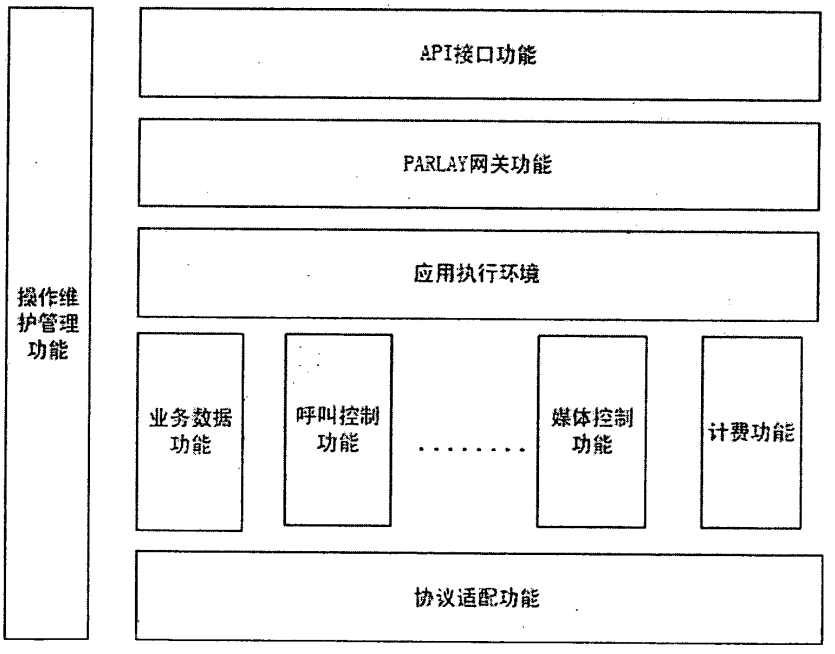

图 1-3 应用服务器的功能模块图

本文的论述重点是 Parlay 网关功能中的 TCAP(Transaction Capabilities Application Part)类协议映射子系统，作者将在下一章中对 Parlay 网关作进一步介绍。

第二章 Parlay 网关

2.1. Parlay API 的体系结构

应用服务器上提供了各种业务应用的编程接口，使得不必对软交换功能进行升级就可以实现增值业务的开发、管理和应用。而目前应用的较为广泛的开放编程接口是 Parlay API。

Parlay 协议是 Parlay 工作组制定、由欧洲电信标准委员会(ETSI, European Telecommunications Standards Institute)发布的开放业务接入的应用编程接口标准，是 NGN 重要的业务接口应用协议。该协议针对高层应用协议接口，采用面向对象的方法，使用标准建模语言(UML, Unified Modeling Language)，分别从类(class)、方法(method)、参数(parameter)和状态模型(state model)等方面进行描述，用接口定义语言 IDL(Interface Definition Language)描述其所有的操作和消息。

Parlay API 是一组开放的、独立于技术的、可扩展的 API。可适用于不同的通信网络，通过它能够完成应用服务器和软交换间的通信，同时应用服务器提供开放的应用编程接口。业务应用开发者通过此开放的标准接口，能实现安全和公开的接入现有网络，利用网络能力为各个网络的用户提供服务。

Parlay API 技术规范共定义了以下 6 种接口，如图 2-1 所示[3]。

1) 客户应用和框架间的接口(接口 1)
2) 客户应用和业务能力特征之间的接口(接口 2)
3) 框架和业务能力特征之间的接口(接口 3)
4) 框架和企业经营者之间的接口(接口 4)
5) 框架和第三方业务提供商之间的接口(接口 5)
6) 业务能力特征和企业经营者之间的接口(接口 6)

Parlay API 体系结构由两个接口部分组成，如图 2-2 所示。

1) 业务接口(Service Interface)：这类接口可以访问 Parlay 网关所提供的一系列基本业务功能，譬如建立或释放路由、与用户交互、发送用户消息、设定 QoS 级别等。业务供应商可以按照不同的业务逻辑对它们进行调用以实现不同的业务。
2) 框架接口(Framework Interface)：这类接口提供业务接口必需的安全认证和管理的能力，以确保业务接口的开放性、安全性、可管理性和弹性。

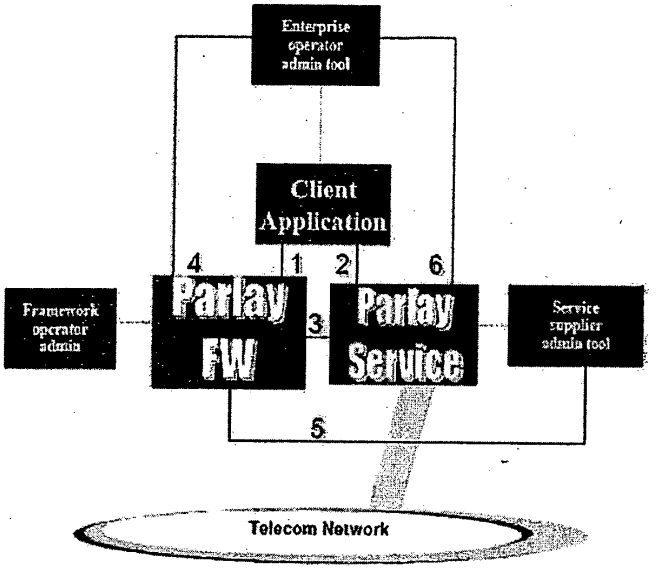

图 2-1 Parlay 接口

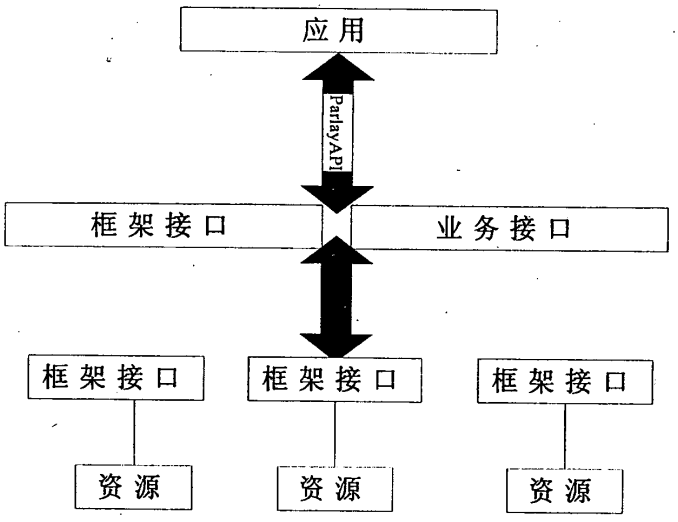

图 2-2 Parlay 体系结构图

2.2. Parlay 提供的业务

Parlay API 是一个标准的接口，从而能够使得第三方通过此接口利用运营商的基础网络提供丰富多彩的业务。例如统一消息业务、基于位置的业务、

呼叫中心业务等，这写业务的业务逻辑都位于应用服务器中。

通过Parlay API提供的第三方业务重要可以分为以下几类：

1) 通信类业务：如点击拨号、VOIP(Voice Over IP)、点击传真、可视电话、会议电话等，以及与位置相关的紧急呼叫业务等。
2) 消息类业务：如统一消息、短消息、语音信箱、E-Mail、多媒体消息、聊天等。
3) 信息类业务：如新闻、体育、金融、天气、票务等各种信息的查询、定制、通知等，以及基于位置的人员跟踪、找朋友等。
4) 支付类业务：如电子商务、移动银行、网上支付、即时售订票、收费浏览等。
5) 娱乐类业务：如游戏、博彩、教育、广告等。

各类业务可以相对独立，也可以有机结合，例如可以在查询信息时根据相应的信息进行支付类业务，再如各种娱乐可以通过不同的消息方式来表现(短消息、Email)，将娱乐与消息业务相结合。

2.3. Parlay网关介绍

2.3.1. Parlay网关的功能

Parlay API位于底层网络设备之上。在Parlay的体系结构中，应用服务器是Parlay的客户端，在应用服务器和底层网络设备之间是Parlay服务器端，也就是Parlay网关。Parlay网关中实现了以下功能：

1) 业务控制功能

对于来自软交换或信令网关的呼叫，Parlay网关能够根据收到的呼叫相关信息确定是需要调用第三方应用还是由应用服务器自身处理。对于需要调用第三方应用的呼叫，Parlay网关可通过API接口，向第三方应用发送调用请求，在第三方应用的控制下，完成呼叫处理。对于需要应用服务器处理的呼叫，Parlay网关可通过API接口，向应用服务器发送调用请求，在应用服务器的控制下，通过与呼叫控制实体的交互完成对呼叫的控制功能，软交换和智能网的SSP都可以作为呼叫控制实体。

2) 协议处理功能

对于来自不同网络实体的不同呼叫，Parlay网关能进行正确的协议处理。对于软交换设备的SIP呼叫，Parlay网关能提供SIP协议并正确处理。对于软交换设备的INAP呼叫，Parlay网关可支持并正确处理INAP协议。对于GSM网络，Parlay网关支持并正确处理CAP(CAMEL Application Part)协议。

3) API接口功能

Parlay网关能够向第三方应用提供Parlay/OSA(Open Service Access) API接口，具有Parlay/OSA API所定义的框架以及SCF，从而调用第三方所提供的各种应用。

4) Parlay网关功能

能对第三方应用提供基于OSA/Parlay 的安全体系认证与授权。提供SIP、H.248和MGCP协议到Parlay/OSA API的映射。提供INAP、CAP、MAP(Mobile Application Part)、WIN MAP到Parlay/OSA API的映射。

5) 针对第三方应用的业务数据功能

Parlay 网关能够根据需要为第三方应用提供某些业务数据功能。Parlay网关允许第三方应用通过API方式对业务数据进行存储、访问和管理等。

6) 针对第三方应用的计费功能

Parlay网关能够提供对第三方应用的计费功能。Parlay网关具有各种所需要的计费信息、完成计费数据的产生、存储和传送。

对于软交换的SIP呼叫，Parlay网关能根据软交换通过SIP 协议送来的相关信息(例如主叫号码、被叫号码、IP地址、会话时长等)和具体业务的要求(例如业务类型、业务折扣等)对相应的第三方应用进行计费。Parlay 网关能够按照会话时长、流量、组合等原则进行计费，并可根据业务需要提供多种优惠折扣。

按照具体增值业务的需求，Parlay 网关能够把费用记到主叫号码、被叫号码或某个特定号码上(例如卡号、银行帐号等)。

7) 操作维护管理功能

Parlay 网关提供操作维护管理功能，完成对第三方应用业务能力、Parlay/OSA 框架的操作、维护和管理。Parlay网关能为第三方应用提供包括权限、日志、SCF能力、消息跟踪能力、测量、告警等功能。

Parlay 网关同时提供本地操作维护和远程操作维护能力，具有良好的人机界面。实现的管理维护功能包括业务管理，配置管理，统计功能和出错处理。

业务管理提供允许运营者管理第三方的应用和业务，包括加载业务，终止业务，更新业务，显示业务信息等。配置管理提供运行及维护Parlay网关的能力，能够修改系统参数。Parlay网关能收集系统运行信息，进行统计整理。统计信息包括Parlay网关中当前的会话数量，当前活跃的会话数量，处理的会话记录，非正常中断传输的统计数据等。出错处理完成的功能包括告警收集，告警管理，告警报告，得到告警的日志及当前告警的列表等。

2.3.2. Parlay 网关的系统组成

图 2-3 描述了 Parlay 网关的总体结构。整个 Parlay 网关由五个子系统构成：框架 (FW, Framework Server) 子系统、业务能力服务器(SCS, Service Capability Server)子系统、协议映射子系统、操作维护子系统和网管子系统。

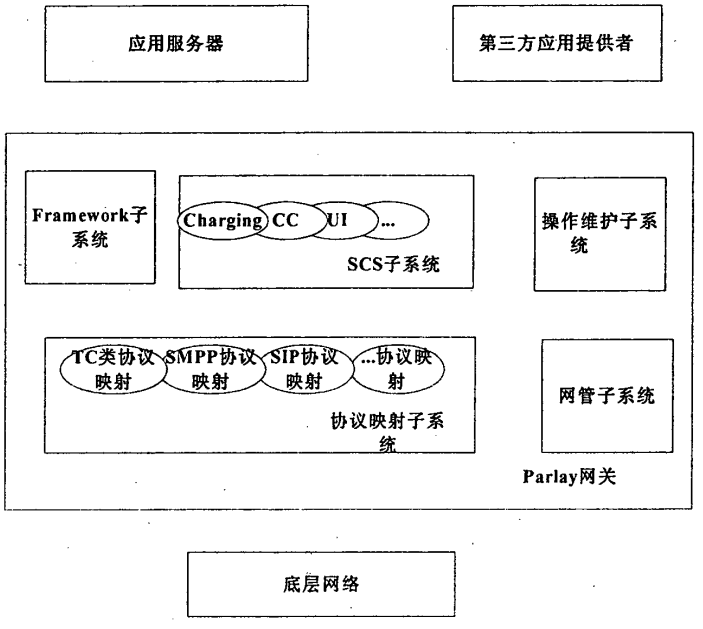

图 2-3 Parlay 网关的总体结构

FW 子系统实现了 Parlay API 中的框架接口，同时提供了 Framework 接口之外的组件注册和注销、组件寻址、负荷分担、流量控制、等支撑和保障功能。

SCS 子系统实现了 Parlay API 的业务接口。整个 SCS 子系统由多个 SCS 组件组成。SCS 组件包括目前 Parlay 定义的 12 个业务能力特征 SCF。此业务能力特征是对网络提供的功能的抽象，负责为高层应用提供访问网络资源和信息的能力。

操作维护子系统提供系统本地的设备和网络管理功能，包括性能管理、配置管理、告警管理和拓扑管理等。

网管子系统根据标准的网管规范提供远端的网络管理功能，包括拓扑管理、配置管理、故障管理、性能管理、用户管理和系统管理。

协议映射子系统完成 Parlay API 与底层通信协议的映射。这部分内容在 Parlay 规范中没有定义，但 3GPP(The 3rd Generation Partnership Project)的

29.998 系列文件对 Parlay API 与底层协议的映射规则作了指导性建议。协议映射子系统由多个组件构成，不同组件分别实现对不同通信协议的映射。常用的通信协议从技术上可分为两大类：TCAP 类协议和非 TCAP 类协议。前者是指基于 TCAP 的智能网协议，后者则是基于 TCP/IP(Transport Control Protocol/Internet Protocol)的协议。这两类协议映射在技术实现上有本质的不同，本文主要对前者作详细论述。

第三章　TCAP 协议介绍

3.1. TCAP 协议简述

TCAP 即事务处理能力应用部分，也称作 TC(Transaction Capabilities)，它为 TCAP 协议上层的各种应用和网络层业务之间提供一系列通信能力。使用 TCAP 提供的服务的上层应用被称为 TC 用户。它为大量分散在电信网中的交换机和专用中心(业务控制点，网管中心等)等应用提供功能和规程，在七号信令网中，它可用于：

1) 交换机之间
2) 交换机和网络服务中心(如：数据库，专用功能单元，操作和维护中心)之间；
3) 网络服务中心之间。

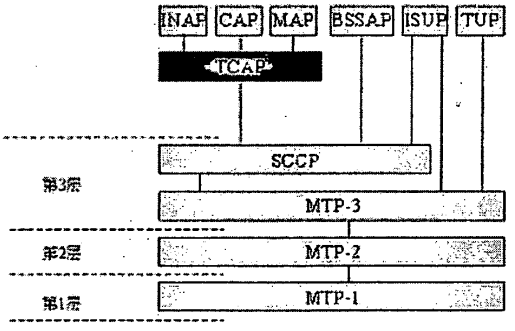

图 3-1 七号信令系统协议族

目前，只有七号信令的消息传递部分(MTP, Message Transferring Part)加上信令连接控制部分(SCCP, Signaling Connection Control Part)是 TC 的网络层业务的提供者，TC 位于 OSI(Open Systems Interconnection)模型的网络层之上。如图 3-1 描述了 TCAP 协议在七号信令系统协议族中的位置。

3.2. TCAP 协议的基本结构

TCAP 由两个子层组成，如图 3-2。

1) 成分子层(CSL, Component Sublayer)：处理成分，即传送远端操作及

响应的应用协议数据单元(Application Protocol Data Unit, APDU)和作为任选的对话部分

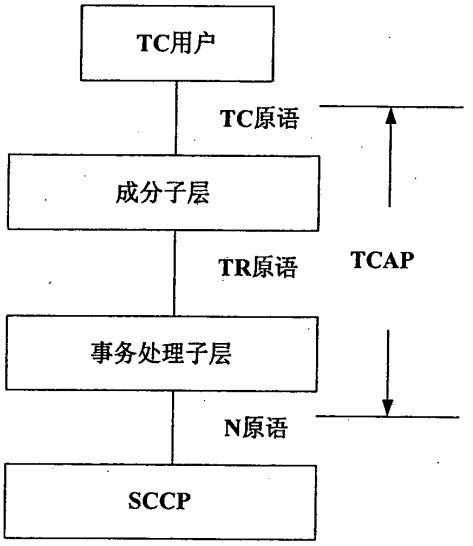

图 3-2 TCAP 的分层

2) 事务处理子层(TSL, Transaction Sublayer)：处理 TC 用户之间包含成分及对话部分(任选)的消息的交换。

在本文论述的 TCAP 类协议映射子系统中，主要涉及成分子层以及其上的 TC 用户协议的映射，不涉及事务处理子层的映射。因此，以下主要介绍成分子层以及 TC 原语，对事务处理子层不作详细介绍。

3.3. 成分子层

3.3.1. 成分类型

成分是用来传送执行一个操作的请求或应答的基本单元。一个成分从属于一个操作，它可以是关于某个操作执行的请求，也可以是某个操作执行的结果。操作是由远端要执行的一个动作，它可以带相关的参数。操作的调用由调用识别号(Invoke ID)识别，这就允许同一或不同操作的几个调用同时存在[4]。

成分在 TC 用户和成分子层之间分别通过。起源于 TC 用户向成分子层发送成分。当几个成分组成一个单消息时，才把它们传至远端。在远端的成分子层收到单消息中的成分后，就把它们分别传给目的地 TC 用户。消息中的成分传到远端 TC 用户后，仍保持在起源接口提供的顺序。

虽然成分的内容与具体的应用有关，但是无论是什么应用系统，从操作过程来看，都可以归为下面五类型：

1) 操作调用成分(INV: Invoke)：INV 成分的作用是要求远端用户执行某一动作。
2) 回送结果－非最后结果成分(RR-NL: Return Result Not Last)：发送此成分说明该操作已被远端成功执行，但由于回送的信息较长，超过网络层允许的最大消息长度，成功的结果需分段传送，此成分不是最后的分段。
3) 回送结果－最后结果成分(RR-L: Return Result Last)：操作已被成功执行，远端 TC 用户用 RR-L 将结果的最后分段传送给始发端，当结果只需用一条消息传送时，也使用该成分传送。
4) 回送差错成分(RE: Return Error)：操作失败，远端用 RE 成分表示失败，并说明失败原因。
5) 拒绝成分(RJ: Reject)：当 TC 用户或 TC 的成分子层发现成分信息出错或者无法理解时，拒绝执行该操作，用 RJ 成分拒绝该成分，并用问题码说明拒绝理由。

3.3.2. 操作类别

根据对操作执行结果应答的不同要求，将操作分为四种类别：

类别 1：成功和失败都报告。

类别 2：仅报告失败。

类别 3：仅报告成功。

类别 4：成功和失败都不报告。

操作类别是操作定义的一部分，每个操作都有一个特定的类别。这表明操作的目的端要么是回送一个成功的输出(结果)或是一个失败的输出(差错)，要么是两者都有或两者皆无。

3.3.3. 对话

为了执行一个应用，两个 TC 用户之间连续的成分交换就构成了一个对话。成分子层提供对话功能，并允许几个对话在两给定 TC 用户之间同时进行。

对话处理也允许 TC 用户之间作为任选传送和协商应用上下文名称以及透明传送用户信息即：非成分数据)。

对话分为两种：非结构化对话和结构对话。

1） 非结构化对话

TC 用户发送不期待回答的成分，并且对话没有开始、继续和结束，这种情形称为非结构化对话。

当一个 TC 用户向它的同层发送单向(Unidirectional)消息时，这表明使用了非结构化对话功能。当一个 TC 用户收到一个单向消息，若要报告协议差错，它也在单向消息中返回。

2） 结构化对话

TC 用户指明对话的开始(或对话形成)，对话的继续和对话结束，这种情形称为结构化对话。结构化对话允许两个 TC 用户之间同时进行几个对话，每个对话由一个特定的对话 ID(Dialog ID)识别。

当使用结构化对话时，TC 用户在向它的同层实体发送成分前必须指明如下四种可能性之一：

- 对话开始；
- 对话证实：第一个后向继续表明对话建立并可以继续。
- 对话继续：TC 用户继续一个已建立的对话且成分可全双工交换。
- 对话结束：发送端不再发送成分也不再接收远端送来的成分。

对话的结束有如下三种情形：

预先安排结束：TC 用户由预先安排决定何时结束对话。在 TC－END 请求原语发送后，对话不发送也不接受成分。

基本结束：TC－END 原语使得未决成分传送且指示这个对话在任一方向都不再交换成分。

对话由 TC 用户中止：TC 用户可以不考虑任何未决操作调用而请求立即结束对话。TC 用户的中止请求使得对话的所有未决操作终结。TC 用户提供端到端信息来指示中止原因和诊断信息。

作为任选，在对话开始阶段和对话证实阶段中，应用上下文信息和用户信息可进行交换。在这种情况下，用户信息也可在对话继续阶段和对话结束期间阶段发送。

3.3.4. 原语

TC 用户与成分子层的接口是 TC 原语。TC 原语可以分为成分原语和对话原语。原语类型分为请求和指示两类。请求(req)类可表示将成分从 TC 用户传送到成分子层，指示(ind)类表示将成分传给 TC 用户。

1） 成分原语

成分处理原语用来处理操作和应答。共有以下几类

TC－INVOKE(请求，指示)：调用一个操作，这个操作也可链接至另一个操作调用。

TC－RESULT－L(请求，指示)：仅为成功执行的操作的结果或分段结果的最终段。

TC－RESULT－NL(请求，指示)：成功执行的操作的分段结果的非最终部分。

TC－U－ERROR(请求，指示)：当 TC 用户收到虽“明白(understand)”但不能执行的操作(1 类或 2 类)，它就用 TC－U－ERROR 请求原语来指明失败理由(差错参数)。调用这个操作的 TC 用户是由 TC－U－ERROR 指示原语来通知的。

TC－L－CANCEL(指示)：成分子层用撤销功能通知 TC 用户与操作类别 1，2，3 有关的操作的时限到。4 类操作的报告是与实施有关的。对于 1 类操作，“时限到”是一个非正常情况。而对 2，3，4 类操作，“时限到”是一个“正常”情况。

TC－U－CANCEL(请求)：TC 用户用 TC－U－CANCEL 请求原语把撤销决定通知本地成分子层。

TC－L－REJECT(指示)(本地拒绝)：成分子层发现收到的成分无效时，则用这个原语通知本地 TC 用户。原语中包括拒绝的原语(问题码参数)。

TC－R－REJECT(指示)(远端拒绝)：成分子层通知本地 TC 用户成分被远端成分子层拒绝。

TC－U－REJECT(请求，指示)：TC 用户可以拒绝任何由其同层实体产生的它认为不正确的成分(拒绝成分除外)。拒绝的原因在问题码参数中指明。

成分原语的参数定义

与成分处理原语有关的参数定义如下：

类别(Class)：操作类别

对话 ID：把成分与一个特定的对话相联系。

调用 ID(Invoke)：识别一个操作调用和它的结果。

链接 ID(Linked ID)：把一个操作调用链接至一个由远端 TC 用户调用的一个先前的操作。

差错(Error)：包含 TC 用户提供的当操作返回失败时的信息。

最终成分(Last Component)：仅用于“指示”类原语，它构成消息的最终成分。

操作(Operation)：识别在另一 TC 用户得请求下由 TC 用户执行的动作。

参数(Parameters)：包含伴随一个操作或为应答一个操作而提供的参数。

问题码(Problem code)：识别拒绝一个成分的原因。

时限(Timeout)：指明操作调用的最长有效时间。

2) 对话原语

对话处理原语用来请求(request)或指示(indicate)与消息传送或对话处理有关的低(子)层功能。

成分子层用于对话处理的原语如下：

TC－UNI(请求，指示)：请求/指明一个非结构化对话。

TC－BEGIN(请求，指示)：开始一个对话。

TC－CONTINUE(请求，指示)：继续一个对话。

TC－END(请求，指示)：结束一个对话。

TC－U－ABORT(指示)：允许 TC 用户突然地终结对话而不传送未决成分。

TC－P－ABORT(指示)：为响应事务处理子层的事务处理中止而通知 TC 用户，对话由业务提供者(即：TC 事务处理子层)而终结。未决成分不传送。

TC－NOTICE(指示)：通知 TC 用户网络业务提供者已不能提供所请求的业务。

用于对话处理原语的参数定义如下：

中止理由(Abort Reason)：指明对话是由于收到的应用上下文名称不支持并且无可选择(中止理由＝应用上下文不支持)或由于其他问题(中止理由＝用户(定义)专用)而中止。

地址参数：起源地址(Originating Address)和目的地地址(Destination Address)是用来识别起源 TC－用户和目的 TC－用户。

应用上下文名称(Application Context Name)：应用上下文是对话启动者或对话响应者建议的应用上下文识别。它用来识别应用业务单元(ASE,Application Service Element)并与对话中应用实体(AE)的互通的必要信息有关。

成分存在(Component Present)：指明成分是否存在。

对话 ID(Dialogue ID)：这个参数也在成分处理原语中出现用于把成分与对话联系起来，同一对话中必须使用同一对话 ID。对于非结构化对话，同一对话 ID 的成分放在有同一目的地地址地单向消息中。对于结构化对话，用于 ID 用于识别从对话开始至结束的属于同一对话的所有成分。

P－ABORT：包含的信息指明 TCAP 决定中止一个对话的原因。

业务质量(Quality of Service)：TC－用户指示可接受的业务质量。目前，

无连结SCCP网络业务的“业务质量”参数由如下组成：

返回选择(Return Option)：规定SCCP“返回消息差错”是否被请求。

顺序控制(Sequence Control)：指明在请求SCCP的协议类别1的业务，使得一系列消息按顺序传送。

终结(Termination)：指明TC－用户选择了何种对话结束(基本的或预先安排的)

用户信息：(User Information)：独立于远端操作业务的TCAP用户之间可交换的信息。

报告原因(Report Cause)：指明SCCP返回消息时的原因，这些原因的规定在SCCP规范中。这个参数用于TC－NOTICE指示原语。

3.4. 成分子层处理过程

与成分子层的两类原语对应，成分子层提供两类过程：成分处理过程和对话处理过程。

3.4.1. 成分处理过程

1） TC成分处理业务原语与成分类型的对应

成分处理即TC－用户调用远端过程并接收响应的能力。成分处理过程将成分处理原语与各个成分相对应。当收到TC用户发出的成分请求原语时，成分处理过程对原语进行处理，产生相应的成分子层APDU。当收到远端发来的成分时，成分处理过程对其进行处理，并产生相应的TC成分指示原语通知TC用户。成分处理原语与成分子层的协议数据单元对应关系见表3-1所示。

表3-1 TC 成分原语与成分类型的对应

业务原语	缩写	成分类型
TC-INVOKE	INV	调用
TC-RESULT-L	RR-L	返回结果(最终)
TC-U-ERROR	RE	返回错误
TC-U-REJECT	RJ	拒绝
TC-R-REJECT	RJ	拒绝
TC-L-REJECT		
TC-RESULT-NL	RR-NL	返回结果(非最终)
TC-L-CANCEL		

TC-U-CANCEL		

2） 调用 ID 的管理

调用 ID 由调用端在操作调用时分配。TC 用户在调用操作之前不需要等候另一个操作的完成。在任何时刻，TC 用户可以有任意个操作在远端进行(虽然远端可因缺乏资源而拒绝一个调用成分)。

每个调用 ID 值与一个操作调用及其相应的调用状态机(ISM)相联系。对该调用 ID 状态机的管理仅仅在调用操作的一端发生。另一端在其对操作调用的回答中反映这一调用 ID，并不管理该调用 ID 的状态机。注意，两端都可以以全双工方式调用操作；即每一端都可管理它调用的操作的状态机，以及可以自由地分配各自独立的调用 ID。

当相应的状态机回复到空闲时，成分 ID 值可以重新分配。然而，当发生某些不正常情况时，立即重新分配可能困难。因此释放的 ID 值(当状态机回复空闲时)不应立即进行重新分配，这种方式是与实际实现有关的。

在章节 3.2.2.中已经提到，根据对操作执行结果应答的不同要求，操作可分为四个类别，每个操作类别，规定了不同类型的状态机。如图 3-3 所示。

3.4.2. 对话处理过程

1） 对话处理原语与对话控制协议数据单元的关系。

TC－UNI，TC－BEGIN，TC－CONTINUE 和 TC－END 四个请求原语被 TC－用户用来控制成分的传送

当 TC 用户发出 TC-UNI 请求原语中包含应用上下文参数时，成分子层将产生单向对话协议数据单元。

当 TC 用户发出 TC-BEGIN 请求原语中包含应用上下文参数时，成分子层将产生对话请求协议数据单元。

当 TC 用户为相应包含应用上下文参数的 TC-BEGIN 指示原语而发出的第一个 TC-CONTINUE 请求原语，成分子层将产生对话响应协议数据单元。

同样，TC 用户为相应 TC-BEGIN 指示原语而发出的第一个 TC-END 请求原语，成分子层也将产生对话响应协议数据单元。

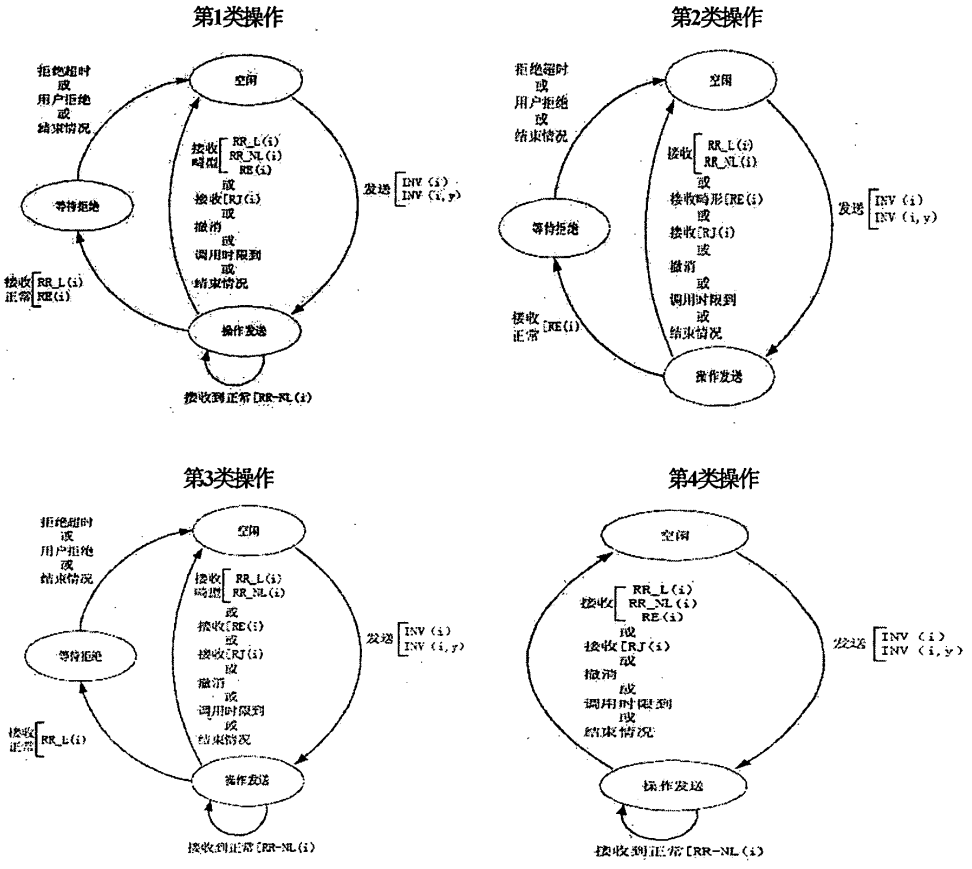

图 3-3 四类操作的状态转换图

TC 对话处理原语和对话控制 APDU 的关系见表 3-2。

表 3-2 TC 对话处理原语与对话控制(APDU)的对应

TC 原语 (请求)	对话控制APDU
TC-UNI	对话 UNI
TC-BEGIN	对话请求
TC-CONTINUE	对话响应
TC-END	对话响应
TC-U-ABORT	对话终止 对话响应

2) 对话过程

TC 用户应用对话控制请求原语来触发所有先前通过的具有相同对话 ID 成分的传送。成分子层收到一上 TC 对话控制原语后通过事务子层的 TR 原语向事务子层触发相应业务请求。

对话过程可以分为对话开始、对话证实、对话继续和对话结束 4 个阶段。

➢ 对话开始

TC-BEGIN 请求原语开始事务子层的一个事务处理并发送任意个(0 或多个)具有相同对话 ID 的成分到事务子层。如果 TC-BEGIN 请求原语中已包括应用上下文名称，对话请求 APDU 也与成分部分连在一起发送。

在目的地侧，成分子层收到事务子层传来的指示原语，将 TC-BEGIN 指示原语和对话信息，以及与后续接收的每一个成分相联系的成分处理原语传送到 TC 用户，开始对话。

➢ 对话证实

如果 TC 用户在 TC-BEGIN 指示原语中收到应用上下文名称参数，且这个应用上下文是可接收的。TC 用户应把同样的值放在第一个后向 TC-CONTINUE 请求原语中。这样对话响应 APDU 与 Continue 消息中的成分一起发送。

如果提供的应用上下文名称不可接受，TC 用户仍可继续对话但要在第一个后向 TC-CONTINUE 请求原语中提供不同的应用上下文名称。这样对话响应 APDU 与 Continue 消息中的成分一起发送。

➢ 对话继续

对话建立以后，两端的 TC-用户就可以使用 TC-CONTINUE 请求原语，通过接口发送具有相同对话 ID 的任何成分。如果对话控制协议数据单元在对话建立期间已经交换，则对话响应中发送的应用上下文名称则被认为是整个对话中 TC 用户之间的应用上下文。在这个阶段带有用户定义的抽象语法的对话部分，作为任选也可能出现，并与成分数据一起被传送。

➢ 对话结束

TC 用户可是用 TC-END 请求原语来请求结束一个对话。在对话基本结束的情况下，任何具有相同对话为对话 ID 的每个成分，及由成分子层行程的此对话的任何拒绝成分，都传送到事务子层，对话即告结束。

在目的端，当每个成分都由跟随 TC-END 指示的相应成分处理原语传到 TC 用户时，对话即告结束。

第四章　TCAP 类协议映射子系统的设计与实现

4.1. TCAP 类协议映射子系统的功能

TCAP 类协议映射子系统是 Parlay 网关协议映射子系统的一个模块，它在 Parlay 网关中的位置如图 2-3 所示。它位于 Parlay 网关 SCS 子系统之下，底层网络设备之上。

SCS 子系统需要向上层应用服务器和第三方应用提供 Parlay API，提供的方式是基于 CORBA(Common Object Request Broker Architecture)服务的、以 IDL 语言定义的分布式调用接口。而底层网络实体是各种智能网应用实体，包括固定智能网、GSM 移动智能网的各种设备，如业务交换点、业务控制点(Service Controlling Point, SCP)等。这些智能网应用实体使用的 TC 协议是基于传统电路交换域的七号信令系统提供的服务。它们和 TCAP 类协议映射子系统之间通过七号信令网进行通信。如图 4-1。

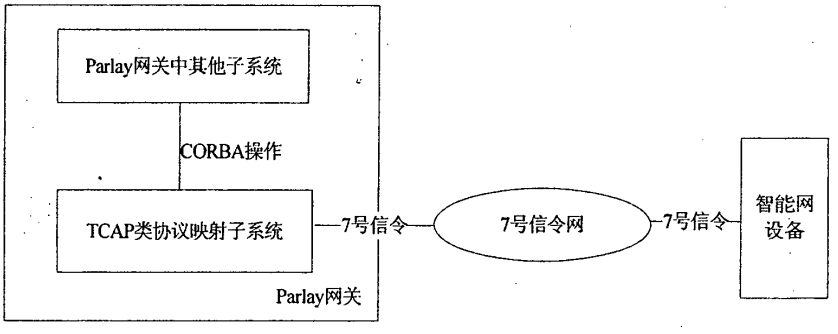

图 4-1 Parlay 网关和七号信令网互操作模型

TCAP 类协议映射子系统的功能就是完成从基于七号信令 SS7(Signaling System No.7)的 TCAP 协议到 CORBA IDL 接口的转换。

由图 4-1 可以看出，把 TCAP 类协议映射到 Parlay API 的步骤可以分为两步。首先是下层的 TCAP 协议的映射，其次是 TCAP 层上 TC 用户协议的映射。根据这两个步骤，可以更进一步细分功能如下：

1) TCAP 协议的映射功能。

这部分功能把来自底层 SS7 协议栈的 TCAP 协议数据单元封装转化为 CORBA 的 IDL 接口中的参数，再交给上层的 TC 用户协议映射功能

处理；或把来自上层协议映射功能的、包含了 TCAP 协议数据的 CORBA 的 IDL 接口参数转化为协议栈 API 的 TCAP 协议数据单元，完成第一步映射。

2) TC 用户协议的映射功能。

这部分功能从包含TCAP协议数据单元的IDL接口参数中剥离出TC用户协议数据单元，然后解码，再将TC用户协议的参数传给上层的SCS；或者反之，将SCS下发的TC用户协议数据单元编码后调用TCAP协议处理模块的方法。目前设计的TC用户协议包括CAP、MAP和INAP。

4.2. 涉及的技术和第三方软件

系统的设计和实现与涉及的技术以及第三方软件都有密切的联系，在设计之前，需要对这些技术和软件进行全方位的考察论证，它们的技术特点，效率，可靠性，成本等对使用它们的系统有很大的影响。因此我们简要介绍一下 TCAP 类协议映射子系统中使用的技术和第三方软件：CORBA 中间件及 SS7 协议栈。

4.2.1. CORBA 技术和 CORBA 中间件

Parlay规范中以IDL语言的形式定义了Parlay API接口，CORBA是Parlay规范建议使用的一种分布式技术，因此 Parlay 网关的架构也是基于 CORBA 技术设计的。Parlay 网关是个分布式系统，其中的各个子系统逻辑上表现为一个个独立的、可插拔组件，在物理上各个子系统进程可以分布在不同地理位置的主机上，各个子系统之间通过 CORBA 中间件进行通信。

CORBA 由对象管理组 (Object Management Group, OMG)提出。它在分布式环境下实现应用的集成，使基于对象的软件成员，在分布的、异构的环境下可重用、可移植、可互操作。CORBA 提供一个框架，如果符合这一框架，就可以在主要的硬件平台和操作系统上建立一个异质的分布式应用。它使用描述性语言 IDL 来定义接口，让客户/服务器对象在特定的对象请求代理(Object Request Broker, ORB)中进行通信。

ORB 是一个在对象间建立客户/服务器联系的中件。使用 ORB，客户可以调用服务器的对象或对象中的应用，被调用的对象不要求在同一台机器上。由 ORB 负责进行通信，同时 ORB 也负责寻找适于完成这一工作的对象，并在服务器对象完成后返回结果。客户对象完全可以不关心服务器对象的位置，实现它所采用的具体技术和工作的硬件平台，甚至不必关心服务器对象

的与服务无关的接口信息。

CORBA体系结构中，应用开发者自己开发的对象实体被称为应用对象；在通用领域和专用领域内已定义的对象被称为公共设施；为公共设施和各种应用对象提供的基本服务则称为对象服务。

CORBA 体系中的应用对象可以分为服务器和客户这两类组件。服务器组件被调用，为其他组件提供某种服务。客户组件调用其他组件，使用服务器提供的服务。服务器和客户是相对的，任何一个组件只要提供了服务就是一个服务器，如果它同时使用其他组件的服务，它也可以是一个客户。TCAP类协议映射子系统中各个模块之间也是互为服务器和客户的关系对象。

在设计实现TCAP类协议映射子系统中，我们选用的CORBA中间件是IONA公司的Orbix。这是一个较为成熟且功能强大的CORBA中间件产品。支持CORBA规范中一些主要的对象服务，如命名服务；支持主流的编程语言C,C++和Java；同时还提供了开发维护工具。

4.2.2. SS7协议栈

SS7协议栈位于TCAP类协议映射子系统之下，主要为TCAP类协议映射子系统提供七号信令协议族各层协议的编解码，信令的传输等功能。TCAP类协议映射子系统通过它和底层的网络实体以七号信令进行通信。

我们选用HP公司的Opencall SS7协议栈作为底层协议栈。它能完成TCAP层及其以下各层的编解码、传输等工作。向上，协议栈提供了一系列API供应用调用，包括消息收发等基本功能。在这一层次上，上层应用可以看到TCAP的成分子层的内容，这些消息和参数包含在以C语言的类型定义的数据结构中。这样，在TCAP类协议映射子系统中只需关心TCAP成分子层以上的设计，而对下面的层次则可以不考虑。

4.3. TCAP类协议映射子系统总体设计

和其他的协议映射子系统相比，TCAP类协议映射子系统相对复杂，它要处理的协议有两层：TCAP层和TC用户层协议。为了降低复杂性，我们采用分层的体系结构，每一层都构建在下一层提供的服务之上，并在层与层之间定义互操作的接口。这样设计的优点是，各层之间相对独立，职责分明，通过统一定义的接口通信，互相屏蔽了内部的实现，模块间的藕合程度降低，有利于实现模块复用；模块内部的改动对外透明，模块间只需修改接口定义即可，不至于牵一发而动全身，便于整个系统开发，升级和移植。系统分层

结构如图4-2所示。

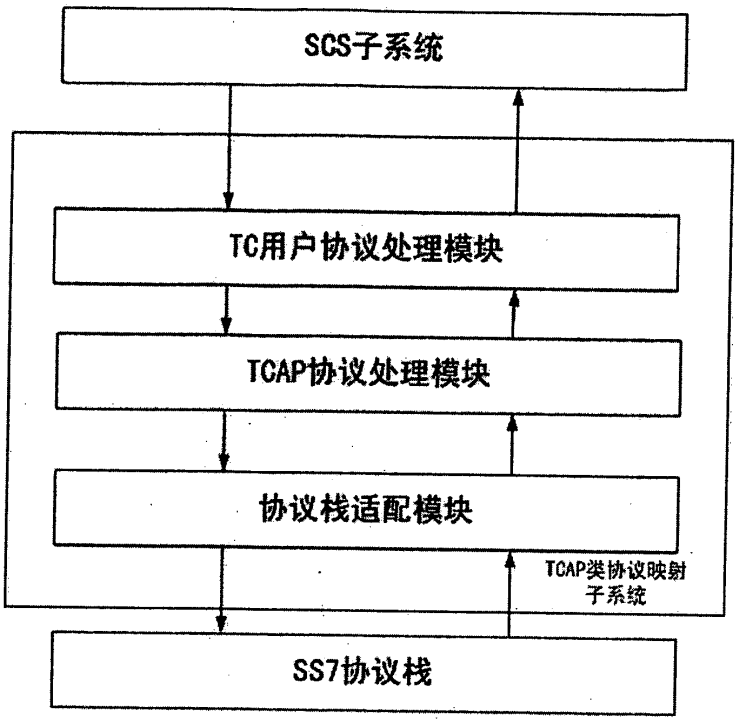

图 4-2 TCAP类协议映射子系统的分层结构

TCAP类协议映射子系统分为三个模块：协议栈适配模块，TCAP协议处理模块和TC用户协议处理模块。各模块的功能如下：

1) 协议栈适配模块

在这个模块中，TCAP协议消息的传递完成了非CORBA域与CORBA域的过渡。自此向上的模块间通信都是通过CORBA ORB总线进行。该模块在整个子系统最底层，和SS7协议栈交互，封装SS7协议栈API，以CORBA的接口的形式为上层提供接收和发送TCAP消息的服务。

2) TCAP协议处理模块

提供TCAP协议翻译的功能。生成基于CORBA服务的一系列接口，将TC对话控制功能映射为模块内对象的调用或被调用行为；提供TCAP会话信息维护、会话控制功能；实现TC用户协议处理对象的创建，定位和删除；接收上层发来的TC用户协议数据单元，或将底层传上来的TC消息中的TC用户协议数据单元解码后发往上层处理。

3) TC用户协议处理模块

该模块根据一定规则将TC用户协议定义的操作映射为CORBA的IDL定义的接口方法，实现TC用户协议参数与Parlay API参数的相互转化。

从逻辑上来看，整个 TCAP 类协议映射子系统由至少一个协议栈适配模块，至少一个 TCAP 协议处理模块和多个 TC 用户协议处理模块组成。在实际运行中，每个模块都可以同时启动多个相同的进程，其中的对象通过 CORBA 命名服务注册到 ORB 总线上，为上层提供服务；而上层模块可以根据一定规则，同样通过命名服务选择任意一个底层模块进程中的对象为其服务。图 4-3 是系统的逻辑视图。

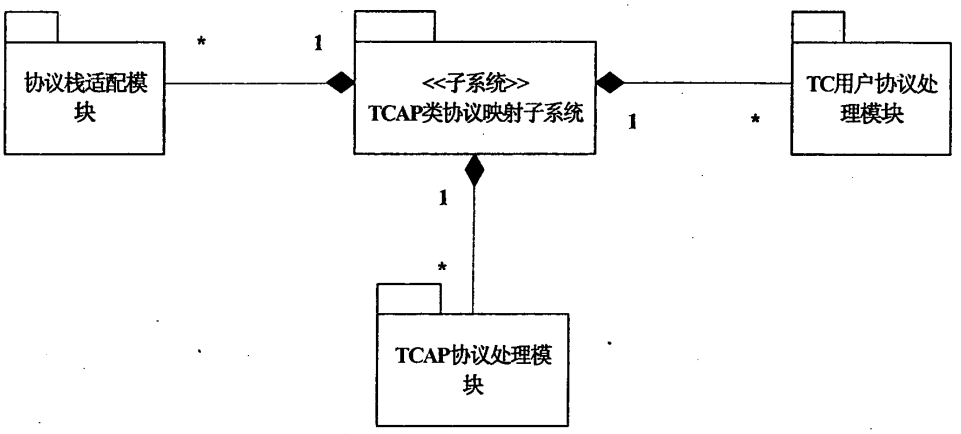

图 4-3 TCAP 类协议映射子系统逻辑视图

接下来我们考虑模块内的主要类以及模块间的接口，此处模块间消息传递机制的设计实现是关键所在。因为协议的交互是双向的，对于每个模块，既要处理向上的消息，也要处理向下的消息，因此，在 TCAP 协议处理模块和 TC 用户模块内应各有两个负责处理不同方向消息流的类。而两个相邻模块的接口也是双向的，既应该有向上传递消息的接口，也应该有向下传递消息的接口。

常见的消息传递机制有两类，同步方式和异步方式。假设消息由模块 A 向模块 B 传递。第一种，同步方式，即当 A 有消息需要传递给 B 时，A 需要一直等待，并且不能作其他的工作，直到 B 接收后 A 才能返回；第二种，异步方式，即当 A 有消息需要传递给 B 时，A 无需等待 B 处理完即可返回。

从我们的系统来看，它是一个协议处理系统，协议交互的双方实体之间通过网络连接，并且在实际应用中可能要处理大量数据。因此在应用中可能出现各种情况，包括网络故障、传输时延，数据处理时延，突发的大量数据到来等。如果使用同步的消息传递方式，有可能某个模块因为突发因素引起处理效率下降而导致其他模块效率下降，从而最终造成整个系统的处理效率下降；而且一个模块的异常也可能引起相邻模块的崩溃。使用异步消息传递方式，克服了上述的缺点，提高了效率，有利于整个系统的稳定，因此这是设计实现本系统的必然选择。

在CORBA的环境下实现模块间消息传递是通过方法调用实现的。入消息的模块是客户端，出消息的模块是服务器，客户端公布调用的接口方法供服务器调用，并在接口方法中完成对收到消息的处理。在Orbix中，CORBA的方法调用也有两种，通常我们使用的是同步的远程方法调用，客户在调用服务器的方法时处于阻塞状态，不能进行其他工作；此外Orbix也实现了对异步远程方法调用机制的支持，这种机制是由底层ORB来实现的，对于上层的服务器和客户端进程来说实现细节是透明的，客户端只需发起一个调用后即可返回，ORB能在服务器端返回结果后自动把结果返回给客户端。

另外一种消息传递方式是消息队列的方式，这也是一种异步的方式，也是常用的消息传递方式，不论是否在CORBA的环境下都可以使用这种方式。实现的方式是设置一个一定容量的缓冲区作为消息队列，当有A有消息要传给B时，就将消息放入缓冲区中，然后立即返回；B在空闲时从缓冲区中取出消息处理。

上述两种异步方式各有利弊，CORBA的异步调用使用简单，内部机制由ORB负责处理，对使用者透明，但只能在CORBA环境下使用。而消息队列方式虽然稍微复杂，需要设计者自己实现细节，但等实现方式灵活，设计者可以依照实际需要设计实现，并且在任何环境下均可以实现。

首先我们针对内部接口，即TCAP协议处理模块和TC用户协议处理模块间的接口来考虑采用何种异步方式。这两个模块均位于CORBA域中，两个方向的接口都是基于CORBA的远程方法调用接口。在消息传递时，两个模块互为服务器和客户端。具体地，当消息由TCAP协议处理模块向TC用户处理模块传递时，TCAP模块向TC用户模块发起调用，TCAP模块是客户，TC用户模块是服务器；当消息传递方向相反时，两者角色也随之转换。经分析，它们之间的双向接口使用CORBA的异步方法调用较为合理。

其次是另一个内部接口：TCAP协议处理模块和协议栈适配模块的接口。协议栈适配模块是完成由非CORBA域与CORBA域之间转化的模块，它位于非CORBA域；而TCAP协议处理模块位于CORBA域。因此协议栈适配模块和TCAP协议处理模块之间的接口特点与前一段所述的情况不同。我们分两种情况考虑这里的接口。

当消息由协议栈适配模块向TCAP协议处理模块传递时，完成由非CORBA域到CORBA的转化，并且TCAP协议处理模块作为CORBA调用的服务器端，协议栈处理模块作为客户端，协议栈处理模块可以通过ORB总线直接将消息传递给上层模块，这种情况下，使用CORBA异步方法调用可以直接使用中间件提供的服务，实现方式简单。

当消息由TCAP协议处理模块向协议栈处理模块传递时，完成由CORBA域

到非 CORBA 的转化。因下层模块没有可供调用的 CORBA 方法接口，上层模块无法直接使用 CORBA 异步调用向下传递消息，这样，消息队列方式成为必然选择。在消息队列实现时，我们设计了一个 CORBA 域内的消息队列对象，当有消息到来时，上层模块通过该对象的 CORBA 方法将消息存入队列；下层模块同样可以调用该对象的 CORBA 方法将消息取出队列。这就实现了下行方向的异步消息传递。

最后我们讨论系统的外部接口。TC 用户协议处理模块和上层 SCS 子系统的接口同 TCAP 协议处理模块与 TC 用户协议处理模块间的接口类似，均为 CORBA 域内的接口，因此接口设计也同样采用 CORBA 异步方法调用实现。协议栈处理模块和 SS7 协议栈的接口已经由 SS7 协议栈提供的 API 实现，不作考虑。

根据章节 3.3.3 中所述，TCAP 的结构化对话包含各种不同的状态以及和对话相关的参数信息，对话控制功能需要通过对对话相关信息进行维护来实现，并且这些信息是由参与 TCAP 对话的各个模块都共同维护使用的。据此，在 TCAP 类协议映射子系统内应实现 TCAP 对话信息维护的功能，负责存储所有和特定对话相关的对话信息，并向各工作模块提供方法调用，供这些模块获取、修改和删除对话信息。经分析，这个功能可由一个 CORBA 接口类来实现，通过 ORB 以命名服务方式公布它的方法，这样各个模块均可以获取并使用它，实现了功能的共享。具体实现时，我们在 TCAP 协议处理模块内实现这个类。

根据以上分析，可以得出 TCAP 类协议映射子系统的基本结构。如图 4-4 所示。

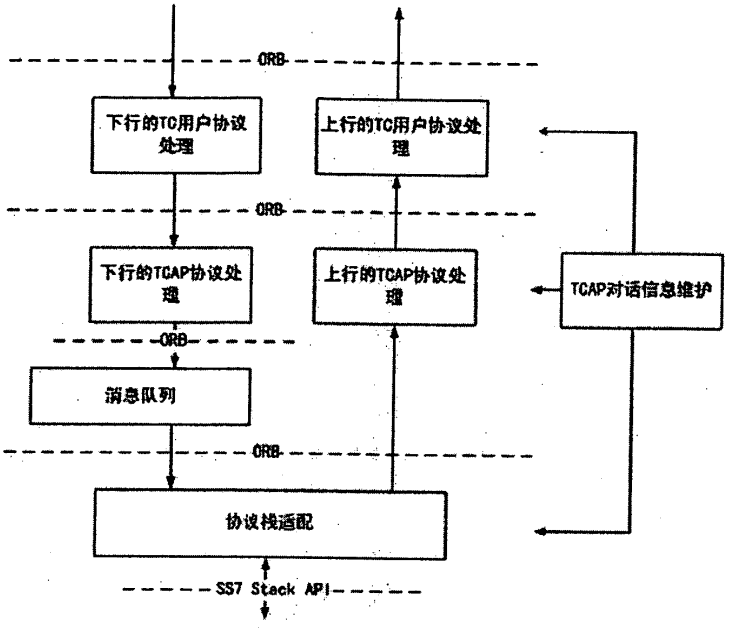

图 4-4 TCAP 类协议映射子系统的结构及接口

从进程角度考虑，协议栈适配模块功能直接和协议栈交互，因此设计为一个进程。TCAP 协议处理和 TC 用户协议处理虽然功能相对独立，但设计思路参考文献[6]，设计时应整体考虑，二者的关系紧密，更适宜部署在同一进程里。

4.4. 协议栈适配模块的设计与实现

我们首先分析一下的协议栈适配模块特点：作为一个底层的模块，主要完成消息的分发工作，因此它的结构应该清晰简单，功能单一，处理效率高，不能设计地过于复杂，形成整个系统的瓶颈；该模块的在消息发送和接收是循环不停地进行的。根据上述分析，我们设计的协议栈适配模块由一个主控程序 tccorba 和一个类 ss7_tc 组成。主控程序启动后完成模块初始化、获取和本模块交互的 TCAP 协议处理模块内 CORBA 对象的对象引用、创建 ss7_tc 类的实例的工作。ss7_tc 类完成本模块的主要功能，与其上层的 CORBA 对象进行 TCAP 消息数据的交换，维护同协议栈的连接，收发 TCAP 消息。

本模块的主要算法如下：

```
连接参数初始化;
while (1){
    if (未与协议栈建立连接)
```

```
    {
        建立与协议栈的连接;
    }
    if (已建立与协议栈的连接)
    {
        \\处理来自底层网络的消息
        尝试从协议栈接收一条消息;
        if (接收到消息)
        {
            处理消息;
            向上层模块传递消息;
        }
        \\处理来自上层应用的消息
        从消息队列中取出下行的消息，发送到协议栈;
    }
}
```

上述主要流程由 ss7_tc 的 mainLoop ()方法实现。该方法包含一个消息处理的主循环，ss7_tc 类在这个方法内循环进行维护协议栈连接，接收、解析上行消息和封装、发送下行消息。

从上述流程中可以看到，在建立了与协议栈的连接后，剩下的工作可以简单地概括为两个，即处理来自底层网络的消息和处理来自上层应用的消息。在一次循环内，两个动作各被执行一次。下面重点描述一下这两步的实现。

处理来自底层网络消息的算法如下：

```
接收一条 TCAP 对话原语;
对话原语解码;
if (对话的第一条原语)
{
    在对话信息表中新增一个条目;
}
修改对话信息表中的对应项;
do
{
    if (对话原语中的下一个成分存在)
```

```
            对话原语中的成分解码;
        else
            跳出循环;
    }
    根据对话原语解码确定调用的上层模块方法;
    将解码的成分作为参数发起调用;
```

对话原语解码主要是得到TCAP对话原语的类型，并根据此对话原语类型能够确定对话当的状态：开始，证实，继续，结束或是其他状态；修改对话信息表的对应项则是把对话信息保存，供其他模块以及后续会话进行使用。特别地，当对话原语是TC_BEGIN时，需要在对话信息表中新建一个表项，并填入初始信息；而当对话原因是TC_END时，则应删除对应的表项。

对话原语中的成分解码，只是把协议栈的数据结构中的成分参数搬移到要被调用的CORBA接口方法的对应参。在这里仅仅简单的拷贝，对成分内的数据并不作任何运算操作。例如：TC_INVOKE成分，对应到TcSignaling.idl接口文件中定义的Invoke结构类型中，ivk_id，lnk_id等成分内参数也和Invoke结构内的参数一一对应。这里需要注意的是，每个对话原语中的成分个数可以是0个，1个或者多个，因此，应循环处理对话原语中的成分，直到所有成分都被处理。

完成了上述的步骤后，就可以向TCAP协议处理模块发起调用了。每一个对话原语在TCAP协议处理模块内都有一个IDL定义的方法供调用，根据对话原语解码得到的原语类型，调用对应的方法，即完成了上行消息的传递工作。

因为这里用的是CORBA异步方法调用机制，因此在调用后可以立即返回处理下面的消息，处理效率较高。

处理来自上层应用消息的算法如下：

```
从消息队列中取出第一条消息;
if( 消息不为空 )
{
    填写对话原语参数;
    构建成分;
    修改对话信息表中对应项的信息;
    发送消息到协议栈;
}
```

最后把ss7_tc类的所有方法列出如下：

➢ 方法：mainLoop ()

说明：本方法是 ss7_tc 类的主循环，ss7_tc 类在这个方法内循环进行维护协议栈连接，接收、解析上行消息和封装、发送下行消息。

参数及类型：argv：char*

返回值类型：int

➢ 方法：connection_handler()

说明：本方法建立一个 SS7 协议栈的连接，如果失败则程序退出。

参数及类型：

ossn：int，指定本地 SSN；

sccp_service：tcx_sccp_service，指定 sccp 服务类型；

ai：int，指定 applicationID，亦即 TC 用户的子系统号；

ii：int，指定 instanceID。

返回：void

➢ 方法：receive_message()

说明：从 SS7 协议栈中接收 TCAP 消息。

参数及类型：cnxId：int，connection_handler 方法中获得的连接句柄；

返回值类型：TC_BOOL，返回连接的状态关闭或是未关闭，true 为关闭，false 未关闭。

➢ 方法：decode_component()

说明：接收成分,并处理成分。

参数及类型：

uid：int，指定 user_dialogue_id；

pid：int，指定 provider_dialogue_id；

inv_id：int*，传出参数及类型，返回 invoke_id；

返回值类型：TC_BOOL，返回是否还有未处理的成分,true：有，false：无。

➢ 方法：error_handler()

说明：接收原语出错时的处理函数。

参数及类型：无

返回值类型：TC_BOOL，返回连接的状态，true 为关闭，false 未关闭。

➢ 方法：call_TcPduUser_func()

说明：本方法把收到的原语和成分中的字段填入 TcPduUser 的方法参数及类型，然后调用 TcPduUser 方法把 TCAP 消息中的内容传给 TcPduUser。

参数及类型：

comp：tcx_component，原语中携带的成分，如果没有携带，则填NULL。

funcName：TcPduUserFunc，调用的TcPduUser方法名称。

返回值类型：void

➢ 方法：send_to_ssp()

说明：本方法从消息队列中取出来自TcPduProvider的TCAP消息，然后调用构造成分和TCAP消息的方法，最后发送消息。

参数及类型：无

返回值类型：void

➢ 方法：build_component()

说明：本方法构造原语中的成分，然后交给协议栈等待发送。

参数及类型：

type：tc_component_type，成分类型。

inv_id：int，调用id。

pid：int，provider_dialogue_id;

data：char*,指向TcUser数据的指针

dataLength：int，

返回值类型：void

➢ 方法：send_message()

说明：本方法构造TCAP原语，然后调用协议栈API发送TCAP消息

参数及类型：

sendData：tcx_primitive*，指向发送的原语的指针。

p_type：tc_primitive_type，原语类型。

pid：int，provider_dialogue_id;

uid：int，user_dialogue_id;

返回值类型：void

4.5. TCAP 协议处理模块的设计

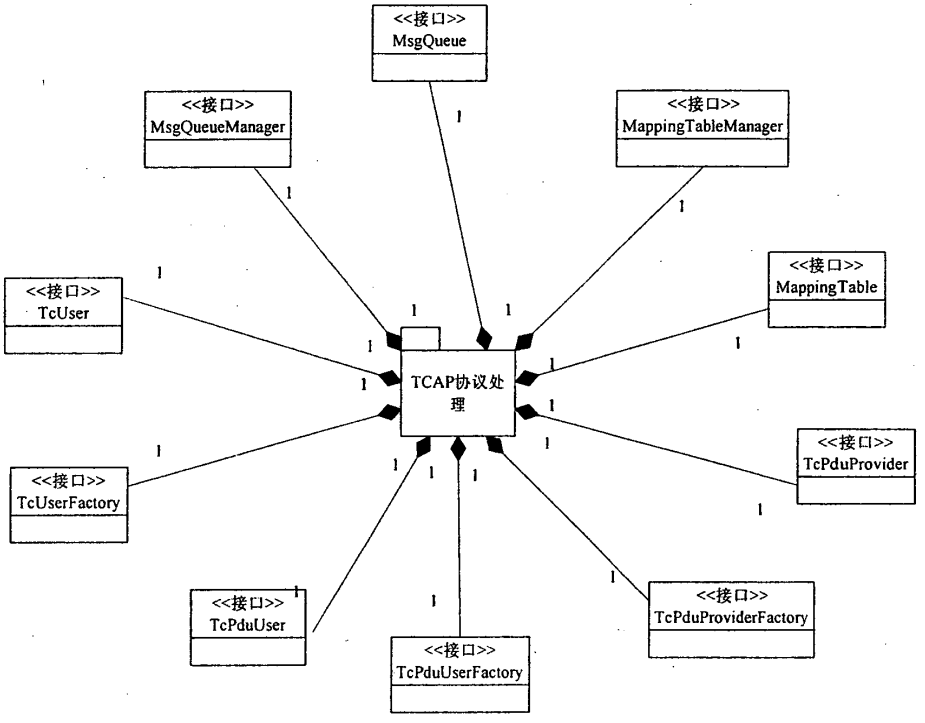

图 4-5 TCAP 协议处理模块的类

TCAP 协议处理模块根据一定规则，规定如何了 TCAP 协议翻译到 IDL 定义，我们称之为交互翻译。对应于该模块的接口文件是 TcSignaling.idl，本模块的主要接口也是基于 TcSignaling.idl 生成的。

TcSignaling.idl 中的接口可以分为：面向 TC PDU 的接口，面向 TC User 的接口和辅助接口。类关系如图 4-5。

4.5.1.面向 TC PDU 的接口

这类接口即 TCAP 协议数据单元的处理接口，由 TcPduUser 和 TcPduProvider 两个接口以及这两个接口的工厂对象组成。根据前面章节所述，协议消息的流向有两个方向，因此，该接口分为上行处理和下行处理。TcPduUser 处理上行的消息，TcPduProvider 处理下行的数据。这两个接口的命名根据是：对于 TC 用户来说，TcPduProvider 相当于 TC PDU 的提供者——TCAP 成分子层;而对于下层的协议适配模块来说，TcPduUser 又相当于 TC PDU 的使用者——TC 用户。

TC PDU 接口和上下层以交互方式进行工作，情形和 TCAP 对话原语的交互相似。我们可以根据 TCAP 对话原语设计其中的方法。TCAP 消息中的原语有两类，即指示原语和请求原语。请求原语由上层的 TC 用户发往 TCAP 层，指示原语由 TCAP 层发往上层。TcPduUser 中包含的方法应与指示原语一一对应；TcPduProvider 中包含的方法应与请求原语一一对应。

1） TcPduUser 接口

TcPduUser 中与指示原语相关的方法包括 uni_ind, begin_ind, continue_ind, end_ind, u_abort_ind, notice_ind, l_cancel_ind, p_abort_ind。以下给出了 TcPduUser 的 IDL 定义。

```
interface TcPduUser{
void   uni_ind ( in TcPduProvider sender,
in DialogQos qos,
in TcAddress dest,
in TcAddress orig,
                    in DialogId d_id,
in DialogPortion d_p,
in ComponentList c_list)
          raises (NoMoreDialogs);

     void begin_ind( in TcPduProvider sender,
in DialogQos qos,
in TcAddress dest,
in TcAddress orig,
in DialogId d_id,
in DialogPortion d_p,
in ComponentList c_list)
          raises (NoMoreDialogs);

     void continue_ind( in DialogId d_id,
                    in DialogPortion d_p,
                    in ComponentList c_list)
          raises (InvalidDialogId);

     void   end_ind( in DialogId d_id,
```

```
    in DialogPortion d_p,
    in ComponentList c_list)
        raises (InvalidDialogId);

    void
    u_abort_ind( in DialogId d_id,
                 in DialogPortion d_p)
        raises (InvalidDialogId);

    void
    notice_ind( in DialogId d_id,
                in short report_cause)
        raises (InvalidDialogId);

    void
    l_cancel_ind( in DialogId d_id,
                  in InvokeId ivk_id)
        raises(InvalidDialogId, InvalidParameter);

    void
    p_abort_ind( in DialogId d_id,
                 in PAbortReason reason)
        raises (InvalidDialogId);

    void setMappingTable( in MappingTable theMappingTable);
};
```

可以看到每一条指示原语映射到一个接口方法上，而原语中的参数也映射到方法的传入参数中。每个调用都有可能发生失败的情况，因此每个方法也设计了对应的异常抛出。下面解释一下主要参数。

TcPduProvider类型参数sender中保存了本次对话中与TcPduUser对应的TcPduProvider对象引用，便于在TcPduUser中引用TcPduProvider。

对话ID和调用ID分别映射为IDL中的unsigned long。

地址类型的参数映射为TcAddress类型，它是一个字符串的序列。

对话QoS映射为DialogQos类型，它是一个整型的常数，用于表示不同

的Qos。定义如下

```
typedef short DialogQos;
const DialogQos SCCP_CLASS_0_NO_ERROR = 0;
const DialogQos SCCP_CLASS_0_WITH_ERROR = 1;
const DialogQos SCCP_CLASS_1_NO_ERROR = 2;
const DialogQos SCCP_CLASS_1_WITH_ERROR = 3;
const DialogQos QOS_NOT_SPECIFIED = 3;
```

原语中携带的成分映射到ComponentList类型，相关的IDL定义如下：

```
typedef sequence<Component> ComponentList;
union Component switch(ComponentType) {
    case TC_INVOKE : Invoke i;
    case TC_RESULT_L : ResultL r_l;
    case TC_RESULT_NL : ResultNl r_nl;
    case TC_U_ERROR : UError u_e;
    case TC_U_REJECT : UReject u_r;
    case TC_R_REJECT : RReject r_r;
};
```

ComponentList定义为Component的不定长序列，Component可能是不同的成分原语，因此定义为联合类型；以TC_INVOKE成分原语为例，我们看看对应类型中的内容。

```
struct Invoke{
    OperationClass op_class;
    InvokeId ivk_id;
    InvokeId lnk_id;
    OperationId op_id;
    Asn1Data oper;
    Timeout op_timer;
};
```

TC_INVOKE成分原语映射为结构类型Invoke，原语的参数也一一映射到结构中的成员变量。其他的成分原语的映射也与TC_INVOKE类似，不再赘述。

参数对话部分映射为DialogPortion类型，IDL定义如下。

```
union DialogPortion switch(boolean) {
    case TRUE: ApplicationContext a_c;
```

```
        case FALSE: Asn1Data dialog_info;
    };
```

DialogPortion 中可能携带的应用上下文或者对话信息，因此被映射为联合类型；联合中的成员变量 a_c 和 dialog_info 分别以八位组的序列存放应用上下文或对话信息。

除了与 TCAP 原语对应的方法外，TcPduUser 中还定义了 setMappingTable 方法。此方法在 TcPduUser 对象中设置了对话信息表的对象引用，使得 TcPduUser 对象可以操作对话信息表，维护其中的对话信息。

我们以 begin_ind()方法为例，说明由指示原语映射的方法的实现。

```
if (应用上下文不支持)
    发送 U_ABORT 原语，返回;
根据应用上下文，生成上层模块的一对 TcUser 对象;
根据对话号检索对话信息表，将 TcUser 对象应用加入对话的条目中;
修改对话状态为 BEGIN;
从 ComponentList 参数中取出 TC 用户协议数据，解码;
把解码后的参数作为 TCUser 对象接口方法内的参数向上层发起调用;
```

作为 begin_ind()原语的处理，在建立对话之前首先要判断收到的原语中的应用上下文是否能是上层应用支持的，应用上下文对于每个 TC 用户是特定的，如果不能支持，则立即通过 TcPduProvider 对象发出一条 U_ABORT 请求原语，对话不建立；

在应用上下文支持的情况下，根据应用上下文的指示，创建一对 TC 用户对象，这是和本层 TcPdu 对象交互的上层应用对象，根据不同的应用上下文建立的 TC 用户对象也不同。因为这一对 TC 用户对象也是和本次对话相关的，因此必须把它们的对象引用添加到对应的对话信息表中，供对话的继续进行使用。接着对话信息表中的对话状态被置为 BEGIN 状态，为对话服务的对象已生成，表示对话已经建立，后续的消息交互可以继续了。

此后从包含成分的参数中取出成分进行处理，不同的成分处理如下。

对于 TC_INVOKE 成分，取出其中的 TC 用户协议数据，解码后通过 TcUser 对象的方法发往上层。

对于 TC_RESULT_L，TC_RESULT_NL 和 TC_U_ERROR 成分，取出其中的 TC 用户协议数据，解码后将结果参数或错误参数存入对话信息表的相应对话条目中，上层 TC 用户可以从对话信息表中获取这些参数。

对于 TC_U_REJECT 和 TC_R_REJECT 成分，则无需返回结果。

TcPduUser 中的其他原语映射的方法实现过程与 begin_ind()的实现大同

小异，都包含对话状态的维护和TC用户参数的解码发送。

2） TcPduProvider接口

TcPduProvider 接口中与请求原语相关的方法有：uni_req, begin_req, continue_confirm_req, continue_req, end_req, u_abort_req, u_cancel_req。下面给出TcPduProvider接口的IDL定义。

```
interface TcPduProvider {
    void uni_req( in DialogQos qos,
                  in TcAddress dest,
                  in TcAddress orig,
                  in DialogId d_id,
                  in DialogPortion d_p,
                  in ComponentList c_list)
            raises ( NoMoreDialogs, LReject);

    void begin_req( in DialogQos qos,
                  in TcAddress dest,
                  in TcAddress orig,
                  in DialogId d_id,
                  in DialogPortion d_p,
                  in ComponentList c_list)
            raises ( NoMoreDialogs, LReject);

    void continue_confirm_req( in TcAddress orig,
                               in DialogId d_id,
                               in DialogPortion d_p,
                               in ComponentList c_list)
            raises ( InvalidDialogId, LReject);

    void   continue_req( in DialogId d_id,
                  in DialogPortion d_p,
                  in ComponentList c_list)
            raises ( InvalidDialogId, LReject);

    void end_req( in DialogId d_id,
```

```
                in DialogPortion d_p,
                in TerminationType term,
                in ComponentList c_list)
            raises ( InvalidDialogId, LReject);

        void u_abort_req( in DialogId d_id,
                in DialogPortion d_p )
            raises ( InvalidDialogId );

        void u_cancel_req( in DialogId d_id,
                in InvokeId ivk_id)
            raises( InvalidDialogId, InvalidParameter);

        boolean  setMsgQueue( in MsgQueue theMsgQueue);
    };
```

TcPduProvider中的参数和TcPduUser的参数类似，都是由TCAP原语中的参数映射得到，在TcSignaling.idl中定义。除了由原语映射的方法外，TcPduProvider中还有一个setMsgQueue()方法，这个方法用于设置和TcPduProvider相关联的消息队列，这样TcPduProvider通过它下发消息。

以continue_req方法为例，我们介绍TcPduProvider中请求原语映射的方法实现。它的流程如下。

```
请求分配一个消息队列单元;
将TCAP消息的参数填入消息队列单元中，消息类型设为TC_CONTINUE;
修改对话信息表中对应项的对话状态;
将消息发送给消息队列;
```

这部分的实现较简单。TC用户的参数和其他对话参数由上层传入TcPduProvider。成分列表中已经包含了编码过的TC用户的操作，因此TcPduProvider中无需再对参数作太多的运算转化，直接发往消息队列即可。需要注意的仅仅是要根据被调用的方法填入正确的原语类型。

3) TcPduUserFactory接口

在协议映射子系统工作的过程中，每当一个TCAP对话建立，都要有一个新的TcPduUser对象来处理对话。因此TcPduUser是根据实际需要动态创建的。TcPduUserFactory接口是TcPduUser的工厂，它封装了创建的TcPduUser

方法，以便在需要时动态生成 TcPduUser 对象。下面是 TcPduUserFactory 的 IDL 定义。

```
interface TcPduUserFactory
{
    TcPduUser   create_tc_pdu_user(
            in ApplicationContext application_context )
        raises( NoMoreDialogs );
};
```

TcPduUserFactory 中有一个方法 create_tc_pdu_user，传入参数是应用上下文信息，方法返回一个创建好的 TcPduUser 对象引用。

4） TcPduProviderFactory 接口

这个接口和 TcPduUserFactory 类似，它是 TcPduProvider 的工厂，提供了动态创建 TcPduProvider 的方法。它的 IDL 定义如下。

```
interface TcPduProviderFactory{
    TcPduProvider
    create_tc_pdu_provider(in TcPduUser user,
                           in MsgQueue msg_queue,
                           out DialogId d_id
                          )
        raises(NoMoreDialogs);
};
```

该方法传入类型为 TcPduUser 和 MsgQueue 的参数把与 TcPduProvider 关联的 TcPduUser 和消息队列传给了工厂对象，这样 TcPduProvider 在被创建时，即可和它们关联起来。对话建立时标示对话的唯一对话号由 TcPduProviderFactory 分配，同时由传出参数类型 DialogId 返回。

4.5.2. 面向 TC User 的接口

面向 TC User 的接口包括 TcUser 和 TcUserFactory。

1） TcUser 接口

该接口继承自 CORBA 生命周期服务中的 LifeCycleObject 接口。它是 TC 用户协议处理模块中所有 TC 用户对象的基接口，定义了所有 TC 用户对象支持的公共操作。在本模块中对 TcUser 接口仅定义其 IDL，不实现。而在 TC 用户协议处理模块中派生出它的子接口中实现对了不同 TC 用户协议的处理。TcUser 的 IDL 定义如下

```
interface TcUser:CosLifeCycle::LifeCycleObject {
    void    bind(
            in AssociationId a_id,
            in TcPduProvider tc_pdu_provider,
            in TcPduUser tc_pdu_user)
raises (UnknownAssociation, InvalidParameter);
};
```

TcUser 中只有一个 bind 方法，用于把特定的 TcPduProvider 和 TcPduUser 与 TcUser 关联，这样上层的 TcUser 可以找到消息下发的下层模块对象引用。

2) TcUserFactory 接口

该接口是 TcUser 的工厂，提供了 TcUser 的动态创建的方法。和 TcUser 一样，这个接口在本模块中也仅定义了 IDL，不实现。它的定义如下。

```
interface TcUserFactory{
    TcUser
    create_tc_user_responder(in ScopedName responder,
                             in TcUser initiator,
                             in AssociationId a_id,
                             in TcContextSetting tc_context_setting)
        raises(CosLifeCycle::NoFactory,
               NoMoreAssociations, UnsupportedTcContext);

    TcUser
    create_tc_user_initiator(in ScopedName initiator)
        raises(CosLifeCycle::NoFactory);
};
```

4.5.3 辅助接口

辅助接口 MappingTable 接口、MappingTableManager 接口、MsgQueue 接口和 MsgQueueManager 接口。这些接口不直接处理 TCAP 协议映射，但是为 TC PDU 接口提供了 TCAP 协议处理的辅助功能。在 CORBA 中，所有的供远程调用的功能都是以接口方法的形式提供的，因此在 TCAP 类协议映射中，对话信息表和消息队列的功能也是通过相应的接口来封装。

1) MappingTable 接口

该接口提供了对话信息的保存功能，并通过 ORB 发布了自己的对象引

用，这样 TCAP 类协议映射子系统内的各个模块都可以获得它的引用。与它相关的 IDL 定义如下。

```
interface MappingTable{
        boolean addNode( in unsigned long theProviderID,
                         in Asn1Data ac,
                         out unsigned long theDialogID);

        boolean getProviderID( in unsigned long    theDialogID,
                               out unsigned long    theProviderID);

        boolean deleteNode( in unsigned long theDialogID);

        boolean searchNode( in unsigned long theDialogID,
                            out unsigned long theIndexOfNode);
        TcPduUser   getTcPduUser( in unsigned long theDialogID);

        TcPduProvider   getTcPduProvider( in unsigned long theDialogID);

        boolean getAppContext( in unsigned long theDialogID,
                               out Asn1Data ac);
        boolean getAppContextAvailable( in unsigned long theDialogID);

        boolean setAppContextAvailable( in unsigned long theDialogID,
                                        in boolean value);

        unsigned long getDialogStatus( in unsigned long theDialogID);

        boolean setDialogStatus( in unsigned long theDialogID,
                                 in unsigned long newStatus);

        boolean setMsgQueue( in MsgQueue theMsgQueue);

        boolean setTcUser( in unsigned long theDialogID,
                           in TcUserId theTcUser);
```

```
TcUserId getTcUser( in unsigned long theDialogID,
                    in boolean isInit);

boolean setSessionId( in unsigned long theSessionID,
                      in unsigned long theDialogID);

unsigned long getSessionIdWithDid( in unsigned long theDialogID);

boolean setCorrelatedId( in unsigned long theCorrelatedId,
                         in unsigned long theDialogID);

unsigned long getCorrelatedId( in unsigned long theDialogID);
unsigned long getSessionIdWithCid(
                      in unsigned long theCorrelatedID);

unsigned long getDialogIdWithSid(
                      in unsigned long theSessionID,
                      in TcUserType theTypeOfUser);

short getOperRet( in unsigned long theDialogID,
                  in unsigned long theInvokeID,
                  out Asn1Data retData,
                  out unsigned long retCode);

void notifyTimeout( in unsigned long theDialogID);

boolean checkInvokeId( in unsigned long theInvokeID,
                       in unsigned long theDialogID);

void registOper( in unsigned long theDialogID,
                 in unsigned long operClass,
                 in unsigned long invokeId,
                 in unsigned long opId,
```

```
                    in unsigned long startTime,
                    in unsigned long timer);

    void deregistOper( in unsigned long theDialogID,
                       in unsigned long invokeId);

    void putResult( in AsnIData retData,
                    in unsigned long theInvokeID,
                    in unsigned long theDialogID,
                    in unsigned long theOperID,
                    in unsigned long retType
                  );
};
```

MappingTable 中保存对话信息的结构定义如下。

```
typedef struct ID_Reference_Struct
{
    time_t                      lastActivity;
    unsigned long               dialogID;
    unsigned long               providerID;
    bool                        appContextAvailable;
    char                        applicationText[7];
    TcSignaling::TcPduUser_ptr  theTcPduUser;
    TcSignaling::TcPduProvider_ptr theTcPduProvider;
    DIALOG_STATUS               dialogStatus;
    unsigned long               sessionID;
    unsigned long               correlatedID;
    TcSignaling::TcUser_ptr     pTcUserResp;
    char                        typeOfResp;
    TcSignaling::TcUser_ptr     pTcUserInit;
    char                        typeOfInit;
    OPER_INFO                   operation;
}ID_Reference;
```

MappingTable 中用于保存对话信息的结构 ID_Reference_Struct 保存了所有和对话有关的信息。主要包括：上个操作到来的时刻，对话 ID，在协议栈

中使用的providerID，应用上下文，应用上下文是否可用，TcPdu对象引用，对话状态，TcUser的引用和它们的类型，以及上个操作的相关信息。MappingTable接口中的addNode方法用于增加一个对话信息表项，deleteNode用于删除一个对话信息表项，其他的方法则提供了对对话信息参数的存储和修改功能。

2) MappingTableManager接口

此接口是MappingTable的管理者对象，提供了一系列方法，包括对MappingTable的动态创建、删除MappingTable对象和获取已创建的MappingTable对象。它的IDL定义如下。

```
struct MappingTableId
{
        MappingTable theTable;
        long entryId;
};
interface MappingTableManager
{
        MappingTableId createMappingTable(
                in long applicationId
                );

        MappingTableId getMappingTable(in long applicationId);
        boolean deleteMappingTable(in long theEntryId);
};
```

结构MappingTableId定义了MappingTableManager中存储MappingTable的单元，包括MappingTable的对象引用和索引号。

3) MsgQueue

这个接口提供了消息队列相关的功能。它的IDL定义如下。

```
struct SendMessageArg
    {
        ComponentList component_list;
        unsigned long          d_id;
        TCDialogArgType        p_type;
        TCDialogArg tc_dialog_arg;
    };
```

```
interface MsgQueue
{
    boolean setMsg(in SendMessageArg theSMArg);
    SendMessageArg getMsg();
};
```

SendMessageArg 结构里定义了消息队列里存储消息的每个单元。结构里的参数有：对话原语的类型，成分列表，对话号，对话原语的其他参数。消息队列里定义了消息的出入队列的方法。

4) MsgQueueManager

此接口是 MsgQueue 的管理者对象，提供了一系列方法，包括对 MsgQueue 的动态创建、删除 MsgQueue 对象。此接口的设计和 MappingTableManager 相似，它的 IDL 定义如下。

```
struct MsgQueueId
{
        MsgQueue theQueue;
        long entryId;
};

interface MsgQueueManager{
        MsgQueueId createMsgQueue();
        boolean deleteMsgQueue(in long theEntryId);
};
```

4.6. TC 用户协议处理模块的设计

在这一层中，TC 用户协议被映射到 IDL 定义上，然后继续通过 ORB 总线发往上层或下层模块。这个模块的设计的重点是如何设计协议的翻译算法。

4.6.1. TC 用户协议的规范翻译算法

TC 用户协议是一类远程操作服务要素(Remote Operation Service Element)规程，它使用 TCAP 中的远端操作服务(ROS, Remote Operation Service)。ROS 提供了类似于 RPC 的机制来支持分布式的应用之间的互操作，它的所有定

义都使用是抽象语法标记 ASN.1(Abstract Syntax Notation one)。TC 用户协议使用 ASN.1 定义并采用 ROS 信息对象类 OPERATION，ERROR 和 EXTENTION 来规定远端操作。因此，规范翻译就是将这种以 ASN.1 描述的 TC 用户协议映射为对应的 CORBA 的 IDL 构造单元。这一翻译过程称作我们称之为规范翻译。规范翻译的设计参考了文献[6]。

1998 年，国际组织 NMF 和 The OpenGroup 联合制定了 JIDM(Joint Inter-Domain Management)，JIDM 中规范从 ASN.1 基本类型到 IDL 的标准映射方法。文献[6]在对 ASN.1 基本类型的翻译上参考了 JIDM 的算法，但 JIDM 并没有关于 TC 用户协议中接口的映射规则。文献[6]把 JIDM 的方法加以扩充，定义了 ASN.1 构造单元到 IDL 接口的映射。在 TC 用户协议中，使用若干个 ASN.1 构造单元(CONTRACT 或 APPLICATION-CONTEXT)或者采用 ASN.1 信息对象类(Information Object Class)来定义协议，因此 TC 用户协议中 IDL 接口的映射规则是根据 ASN.1 的构造单元或信息对象类来定义的。

文献[6]中的规范翻译列举了三种定义 TC 用户应用接口的 ASN.1 形式和它们的映射规则。这三种形式是：

1) 使用自然语言和 ASN.1 描述；形式如下。

```
MODULE ::= <module_name>
BEGIN
IMPORTS <import_list>
EXPORTS <export_list>
OPERATION MACRO_1
...
OPERATION MACRO_n
ERROR MACRO_1
...
ERROR MACRO_m
END
```

2) 较早的只使用 ASN.1 描述；形式如下。

```
MODULE ::= <module_name>
BEGIN
IMPORTS <import_list>
EXPORTS <export_list>
OPERATION MACRO_1
...
```

```
OPERATION MACRO_n
ERROR MACRO_1
...
ERROR MACRO_m
APPLICATION-SERVICE-ELEMENT MACRO_1
...
APPLICATION-SERVICE-ELEMENT MACRO_x
APPLICATION-CONTEXT MACRO_1
...
APPLICATION-CONTEXT MACRO_y
END
```

3) 新的ASN.1记法，使用信息对象类；形式如下。

```
MODULE ::= <module_name>
BEGIN
IMPORTS <import_list>
EXPORTS <export_list>
OPERATION information object_1
...
OPERATION information object_n
ERROR information object_1
...
ERROR information object_m
OPERATION-PACKAGE information object_1
...
OPERATION-PACKAGE information object_x
CONTRACT information object_1
...
CONTRACT information object_y
END
```

在本文中我们只介绍第二种映射规则。

首先，我们澄清几个概念。在TC用户协议中，一个操作(OPERATION)实例返回零个或一个结果(RESULT)，并涉及零个、一个或多个错误宏的实例。一个应用服务单元(APPLICATION-SERVICE-ELEMENT)实例指的是一个或多个操作。一个应用上下文(APPLICATION-CONTEXT)实例指的是一个或多

个应用服务单元.

把TC/ROS用户规范静态映射到CORBA IDL规范的基本映射规则如下。

根据JIDM ASN.1模块到IDL模块映射规则，将包含在ASN.1模块中的TC用户应用定义映射到相同名称的IDL模块中。所有原模块中的接口，类型和常量必须满足：在相应的IDL模块作用域内；把ASN.1模块中的引入类型以typedef声明为IDL中的引入类型。把每个ASN.1类型使用JIDM中的映射规则到对应的IDL类型。一个复杂的ASN.1数据类型(用于描述TC用户应用数据单元)可能生成多于一个IDL数据定义。

操作宏在TC用户协议里用于定义一个操作。它的映射规则是：一个ASN.1操作映射到一个IDL接口的方法，方法名称和操作名称一致。关键字“ARGUMENT”(它是个单一的ASN.1类型)映射到IDL方法中的一个单一的参数类型。创建一个TcContext类型的inout参数ctext。这个参数用于携带TC对话处理信息，调用id，关联id和用于链接操作的链接id。ROS的“RESLUT”参数(这也是个单一的ASN.1类型)被映射到IDL方法中的返回结果类型；如果在“RESULT”关键字后没有ASN.1数据类型或者“RESULT”关键字不存在，则IDL结果类型为“void”。如果一个操作中定义了差错，则在对应的方法中定义一个抛出异常，异常名和差错名相同。

TcContext类型定义如下。变量ctr用于指示TCAP协议处理模块流量控制，用什么对话原语发送该操作；变量ivk_id是操作标识符，用于区分不同的操作；变量lnk_id用于链接操作标识；a_id是一次TC对话的关联标识，属于同一个对话的不同操作可以用此标识关联，我们可以把这个标识理解为等同于TCAP对话原语中的对话id。

```
struct TcContext {
    DialogFlowCtr ctr;
    InvokeId ivk_id;
    InvokeId lnk_id;
    AssociationId a_id;
};
const DialogFlowCtr BEGIN = 0;
const DialogFlowCtr CONTINUE = 1;
const DialogFlowCtr END = 2;
const DialogFlowCtr QUEUE_COMPONENT = 3;
const DialogFlowCtr UNIDIRECTIONAL = 4;
const DialogFlowCtr NOT_SPECIFIED = 5;
```

下面的例子描述了如何将一个操作映射到IDL方法。操作的ASN.1定义如下。

```
InitialDP ::= OPERATION
ARGUMENT InitialDPArg
ERRORS { MissingCustomerRecord, MissingParameter,
SystemFailure, TaskRefused, UnexpectedComponentSequence,
UnexpectedDataValue, UnexpectedParameter }
```

对应的IDL定义如下。

```
void InitialDP (in InitialDPArgType InitialDPArg,
        inout TcContext ctext)
raises (MissingCustomerRecord,
        MissingParameter,
        SystemFailure,
        TaskRefused,
        UnexpectComponentSequence,
        UnexpectedDataValue,
        UnexpectedParameter);
```

差错宏映射为映射到IDL用户定义异常。映射算法是：差错的名称映射为IDL的异常名称；如果该差错中有参数定义，则根据JIDM翻译规则把该参数映射为IDL异常中的一个单一参数；如果差错中的参数以未命名的结构定义，则把它映射为一个IDL类型，并以“<差错名>参数”来命名；在IDL差错类型中创建一个名为”ctr”的附加参数，参数类型是DialogFlowCtr，用于流量控制。

差错宏的映射举例如下。ASN.1定义为：

```
SystemFailure ::= ERROR
PARAMETER UnavailableNetworkResource
```

IDL定义为

```
exception SystemFailure (
    UnavailableNetworkResourceType unavailableNetworkResource;
    DialogFlowCtr ctr);
```

应用上下文用于定义TC用户协议的约定规则，它的映射规则如下。

创建一个名为“<AC_name>Responder”的接口，根据ASN.1定义中的关键字对应用上下文中每个操作用前述的操作映射规则定义方法。这些操作包括：INITIATOR CONSUMER的的每个ASE中的CONSUMER INVOKES

OF 类操作；RESPONDER CONSUMER 的每个 ASE 中的 SUPPLIER INVOKES OF 类操作；SUPPLIER INVOKES/CONSUMER INVOKE 的每个 ASE 中的 OPERATIONS OF 类操作；OPERATIONS 的每个 ASE 中的 SUPPLIER INVOKES/CONSUMER INVOKE/OPERATIONS 的操作。

创建一个名为“<AC_name>Initiator”的接口，根据 ASN.1 定义中的关键字对应用上下文中每个操作用前述的操作映射规则定义方法。这些操作包括：RESPONDER CONSUMER 的每个 ASE 中的 CONSUMER INVOKES OF 类操作；INITIATOR CONSUMER 的每个 ASE 中的 SUPPLIER INVOKES OF 类操作；SUPPLIER INVOKES/CONSUMER INVOKE 的每个 ASE 中的 OPERATIONS OF 类操作；OPERATIONS 的每个 ASE 中的 SUPPLIER INVOKES/CONSUMER INVOKE/OPERATIONS 的操作。

定义一个名为“<AC_name>InitiatorFactory”的接口作为，该接口继承自 TCAP 协议处理模块的 TcUserFactory 接口，并在此接口内定义名为“create_<interface_name>_initiator”的方法。此接口是<AC_name>Initiator 接口的工厂接口。

定义一个名为“<AC_name> Responder Factory”的接口作为，该接口继承自 TCAP 协议处理模块的 TcUserFactory 接口，并在此接口内定义名为“create_<interface_name>_Responder”的方法。此接口是<AC_name> Responder 接口的工厂接口。

4.6.2. TC 用户协议处理实现举例——CAP 协议处理的实现

跟据上述的规范翻译算法，我们可以设计出对 CAP，INAP 和 MAP 协议的处理模块映射后得到的 IDL 接口以及类型定义。在这里我们以 CAP 协议为例，说明 TC 用户协议处理模块的实现。

文献[8]是 CAMEL2 的规范定义，其中定义了 17 个应用业务单元和 3 个应用上下文。我们由它们可以得到 CAP 协议处理模块中的 IDL 接口定义。

17 个应用业务单元是

```
GSM-SCF-activation-ASE::= APPLICATION-SERVICE-ELEMENT
-- 用户是 gsmSSF
CONSUMER INVOKES {
initialDP
}

GSM-SCF-GSM-SRF-activation-of-assist-ASE ::=
```

```
                    APPLICATION-SERVICE-ELEMENT
    -- 用户是 gsmSSF/gsmSRF
    CONSUMER INVOKES {
    assistRequestInstructions
    }

    Assist-connection-establishment-ASE ::=
                APPLICATION-SERVICE-ELEMENT
    -- 提供者是 gsmSCF
    SUPPLIER INVOKES {
    establishTemporaryConnection
        }

    Generic-disconnect-resource-ASE::=
                APPLICATION-SERVICE-ELEMENT
    -- 提供者是 gsmSCF
    SUPPLIER INVOKES {
    disconnectForwardConnection
        }
    Non-assisted-connection-establishment-ASE::=
                    APPLICATION-SERVICE-ELEMENT
    -- 提供者是 gsmSCF
    SUPPLIER INVOKES {
    connectToResource
        }

    Connect-ASE::= APPLICATION-SERVICE-ELEMENT
    -- 提供者是 gsmSCF
    SUPPLIER INVOKES {
    connect
        }

    Call-handling-ASE::= APPLICATION-SERVICE-ELEMENT
    -- 提供者是 gsmSCF
```

```
SUPPLIER INVOKES {
releaseCall
    }

BCSM-event-handling-ASE::= APPLICATION-SERVICE-ELEMENT
-- 用户是 gsmSSF
CONSUMER INVOKES {
eventReportBCSM
    }
-- 提供者是 gsmSCF
SUPPLIER INVOKES {
requestReportBCSMEvent
    }

GSM-SSF-call-processing-ASE::=
    APPLICATION-SERVICE-ELEMENT
-- 提供者是 gsmSCF
SUPPLIER INVOKES {
continue}
Timer-ASE::= APPLICATION-SERVICE-ELEMENT
--提供者是 gsmSCF
SUPPLIER INVOKES {
resetTimer
    }

Billing-ASE::= APPLICATION-SERVICE-ELEMENT
-- 提供者是 gsmSCF
SUPPLIER INVOKES {
furnishChargingInformation
    }

Charging-ASE::= APPLICATION-SERVICE-ELEMENT
-- 用户是 gsmSSF
CONSUMER INVOKES {
```

```
    applyChargingReport
        }
    -- 提供者是 gsmSCF
    SUPPLIER INVOKES {
    applyCharging
        }

    Call-report-ASE::= APPLICATION-SERVICE-ELEMENT
    -- 用户是 SSF
    CONSUMER INVOKES {
    callInformationReport
        }
    -- 提供者是 SCF
    SUPPLIER INVOKES {
    callInformationRequest
        }

    Signalling-control-ASE::= APPLICATION-SERVICE-ELEMENT
    -- 提供者是 SCF
    SUPPLIER INVOKES {
    sendChargingInformation
        }

    Specialized-resource-control-ASE::=
            APPLICATION-SERVICE-ELEMENT
    -- 用户是 SSF/gsmSRF
    CONSUMER INVOKES {
    specializedResourceReport
        }
    -- 提供者是 SCF
    SUPPLIER INVOKES {
        playAnnouncement,
        promptAndCollectUserInformation
        }
```

```
Cancel-ASE::= APPLICATION-SERVICE-ELEMENT
-- 提供者是 SCF
SUPPLIER INVOKES {
cancel
    }

Activity-test-ASE::= APPLICATION-SERVICE-ELEMENT
--提供者是 gsmSCF
SUPPLIER INVOKES {
activityTest
    }
```

1) 应用上下文 CAP-v2-gsmSSF-to-gsmSCF-AC 包含了由 gsmSSF 向 gsmSCF 发起对话的应用业务单元。它在规范中定义如下：

```
CAP-v2-gsmSSF-to-gsmSCF-AC APPLICATION-CONTEXT
-- gsmSSF 采用 InitialDP 启动的对话
INITIATOR CONSUMER OF {
GSM-SCF-activation-ASE,
Assist-connection-establishment-ASE,
Non-assisted-connection-establishment-ASE,
Generic-disconnect-resource-ASE,
Connect-ASE,
Call-handling-ASE,
BCSM-event-handling-ASE,
Charging-ASE,
GSM-SSF-call-processing-ASE,
Timer-ASE,
Billing-ASE,
Call-report-ASE,
Signalling-control-ASE,
Specialized-resource-control-ASE,
Cancel-ASE,
Activity-test-ASE
}
```

由此应用上下文生成接口生成一对Initiator和Responder接口以及它们的工厂对象，这些接口用在由gsmSSF以InitialDP向gsmSCF发起的对话中。IDL定义如下：

```
interface CAP_v2_gsmSSF_to_gsmSCF_ACInitiator: TcSignaling::TcUser{
    void EstablishTemporaryConnection (
        in EstablishTemporaryConnectionArgType
            EstablishTemporaryConnectionArg,
        inout TcContext ctext)
    raises (ETSI_300_374INAP_1::ETCFailed,
        ETSI_300_374INAP_1::MissingParameter,
        ETSI_300_374INAP_1::SystemFailure,
        ETSI_300_374INAP_1::TaskRefused,
        ETSI_300_374INAP_1::UnexpectedComponentSequence,
        ETSI_300_374INAP_1::UnexpectedDataValue,
        ETSI_300_374INAP_1::UnexpectedParameter);

    void DisconnectForwardConnection (inout TcContext ctext)
    raises (ETSI_300_374INAP_1::SystemFailure,
        ETSI_300_374INAP_1::TaskRefused,
        ETSI_300_374INAP_1::UnexpectedComponentSequence);

    void ConnectToResource (
        in ConnectToResourceArgType ConnectToResourceArg,
        inout TcContext ctext)
    raises (ETSI_300_374INAP_1::MissingParameter,
        ETSI_300_374INAP_1::SystemFailure,
        ETSI_300_374INAP_1::TaskRefused,
        ETSI_300_374INAP_1::UnexpectedComponentSequence,
        ETSI_300_374INAP_1::UnexpectedDataValue,
        ETSI_300_374INAP_1::UnexpectedParameter);

    void Connect (in ConnectArgType   ConnectArg,
            inout TcContext ctext)
    raises (ETSI_300_374INAP_1::MissingParameter,
```

```
            ETSI_300_374INAP_1::SystemFailure,
            ETSI_300_374INAP_1::TaskRefused,
            ETSI_300_374INAP_1::UnexpectedComponentSequence,
            ETSI_300_374INAP_1::UnexpectedDataValue,
            ETSI_300_374INAP_1::UnexpectedParameter);

    void ReleaseCall (in ReleaseCallArgType ReleaseCallArg,
                      inout TcContext ctext);

    void EventReportBCSM (
            in EventReportBCSMArgType EventReportBCSMArg,
            inout TcContext ctext);

    void ApplyChargingReport (
        in ApplyChargingReportArgType ApplyChargingReportArg,
        inout TcContext ctext)
    raises (ETSI_300_374INAP_1::MissingParameter,
            ETSI_300_374INAP_1::UnexpectedComponentSequence,
            ETSI_300_374INAP_1::UnexpectedParameter,
            ETSI_300_374INAP_1::UnexpectedDataValue,
            ETSI_300_374INAP_1::ParameterOutOfRange,
            ETSI_300_374INAP_1::SystemFailure,
            ETSI_300_374INAP_1::TaskRefused);

    void Continue (inout TcContext ctext);

    void ResetTimer (in ResetTimerArgType ResetTimerArg,
                     inout TcContext ctext)
    raises (ETSI_300_374INAP_1::MissingParameter,
            ETSI_300_374INAP_1::TaskRefused,
            ETSI_300_374INAP_1::UnexpectedComponentSequence,
            ETSI_300_374INAP_1::UnexpectedDataValue,
            ETSI_300_374INAP_1::UnexpectedParameter);
```

```
        void FurnishChargingInformation (
                in FurnishChargingInformationArgType
                        FurnishChargingInformationArg,
                inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::TaskRefused,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::UnexpectedParameter);

        void CallInformationReport (
                in CallInformationReportArgType CallInformationReportArg,
                inout TcContext ctext);

        void SendChargingInformation (
                        in SendChargingInformationArgType
                                        SendChargingInformationArg,
                        inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedParameter,
                ETSI_300_374INAP_1::ParameterOutOfRange,
                ETSI_300_374INAP_1::SystemFailure,
                ETSI_300_374INAP_1::TaskRefused,
                ETSI_300_374INAP_1::UnknownLegID);

        void Cancel (in CancelArgType CancelArg, inout TcContext ctext)
                        raises (ETSI_300_374INAP_1::CancelFailed);

        void ActiveTest (inout TcContext ctext);
    };

interface CAP_v2_gsmSSF_to_gsmSCF_ACResponder:TcSignaling::TcUser{
```

```
        void InitialDP (in InitialDPArgType InitialDPArg,
                        inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingCustomerRecord,
                ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::SystemFailure,
                ETSI_300_374INAP_1::TaskRefused,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::UnexpectedParameter);

        void EventReportBCSM (
                in EventReportBCSMArgType EventReportBCSMArg,
                inout TcContext ctext);

        void ApplyChargingReport (
            in ApplyChargingReportArgType ApplyChargingReportArg,
            inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedParameter,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::ParameterOutOfRange,
                ETSI_300_374INAP_1::SystemFailure,
                ETSI_300_374INAP_1::TaskRefused);

        void CallInformationReport (
                in CallInformationReportArgType CallInformationReportArg,
                inout TcContext ctext);

        void SpecializedResourceReport(
                in SpecializedResourceReportArgType
                        SpecializedResourceReportArg,
                inout TcContext ctext);
        };
```

```
interface CAP_v2_gsmSSF_to_gsmSCF_ACInitiatorFactory:TcSignaling::
        TcUserInitiatorFactory
        {
            CAP_v2_gsmSSF_to_gsmSCF_ACInitiator
            create_CAP_v2_gsmSSF_to_gsmSCF_AC_initiator()
                    raises(TcSignaling::NoMoreAssociations,
                                TcSignaling::UnsupportedTcContext);
        };

interface CAP_v2_gsmSSF_to_gsmSCF_ACResponderFactory:TcSignaling::
        TcUserResponderFactory
        {
            CAP_v2_gsmSSF_to_gsmSCF_ACResponder
            create_CAP_v2_gsmSSF_to_gsmSCF_AC_responder(
                        in CAP_v2_gsmSSF_to_gsmSCF_ACInitiator initiator,
                        in TcSignaling::AssociationId a_id,
                        in TcSignaling::TcContextSetting tc_context_setting)
                    raises(TcSignaling::NoMoreAssociations,
                    TcSignaling::UnsupportedTcContext);
        };
```

2) 应用上下文 CAP-v2-assist-gsmSSF-to-gsmSCF-AC 包含了由辅助 gsmSSF 向 gsmSCF 发起对话的应用业务单元，它的规范定义如下：

```
CAP-v2-assist-gsmSSF-to-gsmSCF-AC    APPLICATION-CONTEXT
-- gsmSSF 采用 AssistRequestInstructions 启动的对话
INITIATOR CONSUMER OF {
GSM-SCF-GSM-SRF-activation-of-assist-ASE,
Generic-disconnect-resource-ASE,
Non-assisted-connection-establishment-ASE,
Timer-ASE,
Specialized-resource-control-ASE,
Cancel-ASE,
Activity-test-ASE
}
```

```
        ::=  {ccitt(0)  identified-organization(4)  etsi(0)  mobileDomain(0)
gsm-Network(1) ac(0)cap-assist-handoff-gsmssf-to-gsmscf(51) version2(1)};
```

由此得到的接口和它们的工厂接口用于处理辅助 gsmSSF 以 AssistReqestInstructions 操作向 gsmSCF 发起对话的情形。它们的 IDL 定义如下：

```
interface CAP_v2assist_gsmSSF_to_gsmSCF_ACInitiator:TcSignaling::TcUser{
        void DisconnectForwardConnection (inout TcContext ctext)
                raises (ETSI_300_374INAP_1::SystemFailure,
                        ETSI_300_374INAP_1::TaskRefused,
                        ETSI_300_374INAP_1::UnexpectedComponentSequence);

        void ConnectToResource (
                in ConnectToResourceArgType ConnectToResourceArg,
                inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::SystemFailure,
                ETSI_300_374INAP_1::TaskRefused,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::UnexpectedParameter);

        void ResetTimer (in ResetTimerArgType ResetTimerArg,
                        inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::TaskRefused,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::UnexpectedParameter);

        void Cancel (in CancelArgType CancelArg, inout TcContext ctext)
                        raises (ETSI_300_374INAP_1::CancelFailed);

        void ActiveTest(inout TcContext ctext);
    };
```

```
interface
CAP_v2assist_gsmSSF_to_gsmSCF_ACResponder:TcSignaling::TcUser{
        void AssistRequestInstructions (
            in AssistRequestInstructionsArgType
                AssistRequestInstructionsArg,
            inout TcContext ctext)
        raises (ETSI_300_374INAP_1::MissingCustomerRecord,
            ETSI_300_374INAP_1::MissingParameter,
            ETSI_300_374INAP_1::TaskRefused,
            ETSI_300_374INAP_1::UnexpectedComponentSequence,
            ETSI_300_374INAP_1::UnexpectedDataValue,
            ETSI_300_374INAP_1::UnexpectedParameter);

        void SpecializedResourceReport(
            in SpecializedResourceReportArgType
                SpecializedResourceReportArg,
            inout TcContext ctext);
    };
interface CAP_v2assist_gsmSSF_to_gsmSCF_ACInitiatorFactory
        :TcSignaling::TcUserInitiatorFactory
{
        CAP_v2assist_gsmSSF_to_gsmSCF_ACInitiator
        create_CAP_v2assist_gsmSSF_to_gsmSCF_AC_initiator()
            raises(TcSignaling::NoMoreAssociations,
                TcSignaling::UnsupportedTcContext);
};

interface CAP_v2assist_gsmSSF_to_gsmSCF_ACResponderFactory
        :TcSignaling::TcUserResponderFactory
{
    CAP_v2assist_gsmSSF_to_gsmSCF_ACResponder
        create_CAP_v2assist_gsmSSF_to_gsmSCF_AC_responder(
            in CAP_v2assist_gsmSSF_to_gsmSCF_ACInitiator initiator,
```

```
                in TcSignaling::AssociationId a_id,
                in TcSignaling::TcContextSetting tc_context_setting)
        raises(TcSignaling::NoMoreAssociations,
                TcSignaling::UnsupportedTcContext);
};
```

3) 应用上下文 CAP-v2-gsmSRF-to-gsmSCF-AC 包含了由 gsmSRF 向 SCF 发起对话的应用业务单元，它的规范定义如下：

```
CAP-v2-gsmSRF-to-gsmSCF-AC APPLICATION-CONTEXT
-- gsmSRF 采用 AssistRequestInstructions 启动的对话
INITIATOR CONSUMER OF {
GSM-SCF-GSM-SRF-activation-of-assist-ASE,
Specialized-resource-control-ASE,
Cancel-ASE,
Activity-test-ASE
}
::= {ccitt(0) identified-organization(4) etsi(0) mobileDomain(0)
gsm-Network(1) ac(0)cap-gsmSRF-to-gsmscf(52) version2(1)};
```

由此得到的接口和它们的工厂接口用于处理 gsmSRF 以 AssistReqestInstructions 操作向 gsmSCF 发起对话的情形。它们的 IDL 定义如下：

```
interface CAP_v2_gsmSRF_to_gsmSCF_ACInitiator: TcSignaling::TcUser{
    void PlayAnnouncement(
            in PlayAnnouncementArgType PlayAnnouncementArg,
            inout TcContext ctext)
        raises (ETSI_300_374INAP_1::Cancelled,
                ETSI_300_374INAP_1::MissingParameter,
                ETSI_300_374INAP_1::SystemFailure,
                ETSI_300_374INAP_1::UnexpectedComponentSequence,
                ETSI_300_374INAP_1::UnexpectedDataValue,
                ETSI_300_374INAP_1::UnexpectedParameter,
                ETSI_300_374INAP_1::UnavailableResource);

    ReceivedInformationArgType
    PromptAndCollectUserInformation(
```

```
            in PromptAndCollectUserInformationArgType
                PromptAndCollectUserInformationArg,
            inout TcContext ctext)
        raises(ETSI_300_374INAP_1::Cancelled,
            ETSI_300_374INAP_1::ImproperCallerResponse,
            ETSI_300_374INAP_1::MissingParameter,
            ETSI_300_374INAP_1::SystemFailure,
            ETSI_300_374INAP_1::TaskRefused,
            ETSI_300_374INAP_1::UnexpectedComponentSequence,
            ETSI_300_374INAP_1::UnavailableResource,
            ETSI_300_374INAP_1::UnexpectedDataValue,
            ETSI_300_374INAP_1::UnexpectedParameter);

    void Cancel (in CancelArgType CancelArg, inout TcContext ctext)
                raises (ETSI_300_374INAP_1::CancelFailed);

    void ActiveTest (inout TcContext ctext);
};

interface CAP_v2_gsmSRF_to_gsmSCF_ACResponder: TcSignaling::TcUser{

    void AssistRequestInstructions (
        in AssistRequestInstructionsArgType AssistRequestInstructionsArg,
        inout TcContext ctext)
    raises (ETSI_300_374INAP_1::MissingCustomerRecord,
        ETSI_300_374INAP_1::MissingParameter,
        ETSI_300_374INAP_1::TaskRefused,
        ETSI_300_374INAP_1::UnexpectedComponentSequence,
        ETSI_300_374INAP_1::UnexpectedDataValue,
        ETSI_300_374INAP_1::UnexpectedParameter);

    void SpecializedResourceReport(
        in SpecializedResourceReportArgType
            SpecializedResourceReportArg,
```

```
            inout TcContext ctext);
};

interface CAP_v2_gsmSRF_to_gsmSCF_ACInitiatorFactory:
            TcSignaling::TcUserInitiatorFactory
{
    CAP_v2_gsmSRF_to_gsmSCF_ACInitiator
    create_CAP_v2_gsmSRF_to_gsmSCF_AC_initiator()
            raises(TcSignaling::NoMoreAssociations,
                    TcSignaling::UnsupportedTcContext);

};

interface CAP_v2_gsmSRF_to_gsmSCF_ACResponderFactory:
            TcSignaling::TcUserResponderFactory
{
   CAP_v2_gsmSRF_to_gsmSCF_ACResponder
    create_CAP_v2_gsmSRF_to_gsmSCF_AC_responder(
            in CAP_v2_gsmSRF_to_gsmSCF_ACInitiator initiator,
            in TcSignaling::AssociationId a_id,
            in TcSignaling::TcContextSetting tc_context_setting)
    raises(TcSignaling::NoMoreAssociations,
            TcSignaling::UnsupportedTcContext);
};
```

上述的接口中，Responder 类接口中包含方法的都是由 gsmSRF/gsmSSF 发往 gsmSCF 的操作映射得到，因此，Responder 类接口处理的是发往上层应用的消息。Initiator 类接口中包含方法的都是由 gsmSCF 发往 gsmSRF/gsmSSF 发往的操作映射得到，因此，Responder 类接口处理的是上层应用发出的消息。当然，根据具体的 TC 用户协议的不同定义，这两类接口起的作用有可能相反。

4.7. TCAP 类协议映射子系统的工作流程

最后我们根据上面设计的系统描述一下 TCAP 类协议映射子系统的实际工作流程。图 4-6 是 TCAP 类协议映射子系统的工作流程示意图。

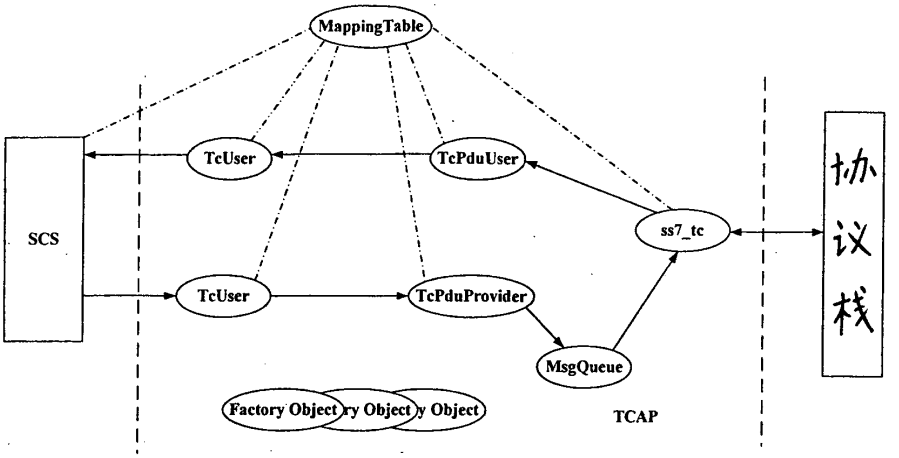

图 4-6 TCAP 类协议映射子系统的工作流程示意

实线的表示的是消息传递的方法调用，箭头指示了消息的传递方向。可以看到参与消息传递的对象有 ss7_tc, MsgQueue, TcPduUser, TcPduProvider 和 TcUser。MappingTable 保存了对话信息，点划线表示它为这些对象提供对话信息。工厂对象动态创建上述的对象。

以下分为两类情况论述系统的工作流程。

4.7.1. 由底层网络发起呼叫的调用流程

1) ss7_tc 通过命名服务,得到 MappingTableManager,的对象引用。创建 MappingTable。
2) ss7_tc 收到协议栈传来的 TC_BEGIN 原语,调用 MappingTable 的 addNode 方法增加一个表项保存对话信息。addNode 首先调用 TcPduUserFactory 的 create_tc_pdu_user()和 TcPduProviderFactory 的 create_tc_pdu_provider()创建一对 TcPduUser 和 TcPduProvider。create_tc_pdu_provider 方法同时分配一个 dialog_id,创建一个 MsgQueue 对象与 TcPduProvider 相关联。MsgQueue 对象也接着对话相关的 dialog_id,provider_id,ApplicationContext，MsgQueue 对象引用，TcPduUser 对象引用和 TcPduProvider 对象引用被保存到对话映射表中。对话状态被设置为 BEGIN。
3) ss7_tc 调用 TcPduUser 的 begin_ind 方法，把 TCAP 消息、dialog_id 和 TcPduProvider 作为参数传给 TcPduUser; TcPduUser 分析参数中应用上下文，判断上层的应用，创建一对的 TcUserInitiator 和 TcUserResponder，再把 TcUserInitiator 和 TcUserResponder 的对象引用保存到对话映射表中。TcPduUser 同时解码 TC 用户协议数据单元，

发给处理上行消息的 TcUser 对象。

4) 处理上行消息的 TcUser 对象把 TC 用户协议的参数映射为对应的 Parlay API 参数上，并将 associated_id(由 dialog_id 得到)作为关联对话的参数，然后把消息发往上层。

5) 上层应用处理后，向 scs 发起调用，scs 将 ParlayAPI 参数映射为 TC 用户协议数据单元，通过 associated_id 从 MappingTable 中找到处理下行消息的 TcUserInitiator 对象引用，发起调用，并通过当前的对话状态指示流控信息 TcContext。

6) TcUserInitiator 编码 TcUser 用户数据单元，根据 MappingTable 找到 TcPduProvider 对象引用，填入相应参数后，调用 TcPduProvider 的方法

7) 每个 TcPduProvider 根据 MappingTable 找到 MsgQueue 将 TCAP 消息送入队列

8) ss7_tc 进程轮询自带的对话映射表中每个条目，把每个 TcPduProvider 的 MsgQueue 中消息取出发送。并修改 MappingTable 中的对话状态，若对话结束则删除对应的条目。

4.7.2. 由上层发起呼叫的调用流程

1) 上层应用向 scs 发起调用，scs 获得 MappingTableManager 的对象引用，并创建一个 MappingTable 对象。

2) scs 调用 MappingTable 的 addNode 方法，addNode 首先调用 TcPduUserFactory 的 create_tc_pdu_user()和 TcPduProviderFactory 的 create_tc_pdu_provider()创建一对 TcPduUser 和 TcPduProvider。create_tc_pdu_provider 方法同时分配一个 dialog_id,创建一个 MsgQueue 对象与 TcPduProvider 相关联。MsgQueue 对象也接着对话相关的 dialog_id,provider_id,ApplicationContext，MsgQueue 对象引用，TcPduUser 对象引用和 TcPduProvider 对象引用被保存到对话映射表中。对话状态被设置为 BEGIN。

3) 调用 TcUser 的工厂接口的方法创建 TcUserInitiator 和 TcUserResponder，并存入 MappingTable。

4) scs 把 Parlay API 参数映射为 TcUser 消息；把 TcUser 消息的内容作为参数向处理下行消息的 TcUser 发起调用。

5) 处理下行消息的 TcUser 将 TcUser 协议数据单元编码后，向 TcPduProvider 发起调用

6) TcPduProvider把待发送的消息存放在和它关联消息队列中。

7) ss7_tc 进程根据轮循 MappingTable 中的条目，对每个对话，从MsgQueue中消息取出发送。

8) 当后续的消息被 ss7 进程收到，它就根据 dialog_id 查找MappingTable，找出关联的TcPduUser将消息上发。

第五章 改进及展望

5.1. 下一步的工作

根据上述的设计框架，作者实现了TCAP类协议映射子系统的一个子集，包括协议栈适配模块，TCAP协议处理模块和CAP协议处理模块。在Parlay网关，应用服务器和底层网络设备的联合测试中，该子集能够完成TCAP协议、CAP协议的映射，配合应用服务器提供CAMEL业务，达到预期的效果。但这个系统还有一些方面需要进一步完善。

首先，设计方案提出了实现TCAP类协议映射的通用框架，完善的TCAP类子系统的TC用户协议处理模块应该包括CAP、MAP和INAP协议处理，现阶段仅实现了CAP协议的处理，下一步应实现MAP和INAP协议的处理模块。使之成为一个完整的TCAP类协议映射系统。

其次，底层的协议栈处理模块还需要作进一步改进才能达到实用要求。主要问题是：目前该模块仅支持建立到一条协议栈的连接；应用服务器和Parlay网关在提供CAMEL业务时，在CAMEL网络的三种物理配置方式中，也只支持通过SSP中继到IP的方式。改进的方向应使该模块能以多线程的方式处理多条SS7链路，并且可以更方便灵活地配置使之满足不同物理配置情形的需要。

5.2. 下一代网络在中国的发展

NGN在中国只有3年的发展时间。运营商在厂商的推动和树立网络技术优势的双重驱动下，纷纷加紧网络铺设和试用，NGN成为中国通信市场的一大热点，NGN产业链呈现出共赢与竞争博弈并存的局面。2005年为中国NGN商用元年，主流运营商开始了NGN的规模化建设，而且将其用于建设基础网络。例如中国电信和中国网通都开始启动建设NGN长途骨干网，电信在南方大规模推向以NGN为主要方式的网络智能化，网通则在南方全面商用NGN，在北方部分省份也开始NGN试商用。中国NGN网络建设从试验网开始进入商用阶段，NGN网络建设规模上突破1000万端口，相比2004年的500万端口，有了显著提高。这一系列情况表明，中国已经处在NGN时代的前夜。

NGN的大规模商用，使互联网市场面临着重新洗牌，也必将打造一条新的

产业链，给运营商、设备制造商、内容提供商等带来更大的发展空间。虽然目前基于NGN的业务应用还不多，为运营商和用户创造的价值还不显著，但随着越来越多的增值应用深入到企业和个人用户中，现有的网络基础将不能满足不断增长的用户个性化需求和服务的增值服务，那么NGN的开放性，可管理性等网络优势才得以发挥，因此，能够推动NGN发展的关键因素还在于应用。不同的接入方式更是需要一个智能化的网络来管理调配资源，网络带宽和管理等问题变得越来越重要，现有的宽带发展和日渐丰富的增值应用为下一代网络的发展奠定了良好的基础。

用户的发展是决定中国NGN市场的先决条件，中国电信运营商的NGN建网思路是边发展边建设，中国网通、中国电信、中国移动、中国联通都是积极的建设者。预计2006-2010年是中国NGN用户快速增长的阶段，2010年将达到5000万的规模。

今后NGN的发展将呈现两大趋势。一种趋势是融合，包括网络融合和业务融合。另一种发展趋势是开放，包括业务的开放和网络的开放。但网络融合、业务融合是个漫长的过程。运营商们从上世纪70年代采用基于电路交换技术的ISDN，到上世纪80年代采用基于ATM技术的B-ISDN，再到上世纪90年代Internet爆炸式增长时试图通过路由器技术提供所有电信业务，种种做法的目的都是要实现融合。但这些努力都未达到预期目标，而NGN无疑是为实现上述目标进行的一次新的努力。这种努力，最需要的就是时间。

伴随中国通信产业的快速发展，NGN作为下一代网络的发展方向，成为通信产业的关注焦点，虽然目前下一代网络的发展方向还不完全明朗，包括总体架构、业务开放模式、服务质量、安全性等问题还没有得到完全解决。但在NGN领域，中国的研究与国外基本处于同一个水平，NGN对中国的电信产业是一个很好的历史机遇。信息产业部曾明确表示，中国政府一直鼓励和支持中国的产业界对NGN的技术、业务、商业模式等问题开展有效的研究，从而积极有效地引导中国信息产业的发展，保障国家通信网的安全，增强中国通信产业的核心竞争力，增强国内企业的国际竞争力和占领国际市场的能力。中国政府从政策上将对NGN的发展给予有力的支持。

结束语

NGN具有丰富的应用前景，满足人们多样化和个性化的业务需求。在NGN统一网络上，可融合通信、信息、电子商务、娱乐等业务，实现互动梦想，新型语音、数据、图像融合业务将层出不穷，改善人们沟通和生活的方式。Parlay网关是下一代网络业务平台的重要组成部分。对于在NGN的分层体系结构中快速、方便开发新业务，Parlay API体现出一定的优越性，它是一种重要的业务开放技术。

作者在研究生期间参与了实验室内部项目“Parlay/Parlay网关与应用服务器”及信息产业部电子发展基金项目“下一代网络核心业务平台”，负责TCAP类协议映射子系统的设计与实现。本论文是对作者开发工作的部分技术成果及软件实践的总结。

作者在研究生期间主要从事了以下工作：

1. Parlay网关SCS子系统Charging SCF模块的实现
2. Parlay网关CMPP协议映射子系统的设计与实现
3. Parlay网关TCAP协议映射子系统设计与实现
4. Parlay网关SCS子系统UI SCF和GCC SCF模块的部分设计和实现
5. 作为项目组的配制管理员参与CMMI2级认证

限于本人的学识和精力有限，论文中肯定有一些不足之处，敬请批评指正。

参考文献

[1] 赵慧玲，叶华，以软交换为核心的下一代网络技术，人民邮电出版社，2002年
[2] 中华人民共和国信息产业部，基于软交换的应用服务器设备技术要求(征求意见稿)，2005年
[3] 赵慧玲，基于软交换的下一代网络技术，电信建设，2002年第四期，4－10，2002年4月
[4] 桂海源，骆亚国 No.7信令系统，北京邮电大学出版社，1999年
[5] ITU-T Q.771~775 Specifications of Signalling System No. 7 – Transaction capabilities application part，1997年
[6] CORBA/TC Interworking and SCCP Inter-ORB Protocol Specification，2001-01
[7]中华人民共和国信息产业部，Parlay 应用程序接口(API)技术规范(送审稿)，2002年
[8] 3GPP TS 09.78. CAMEL application part(CAP) specification.1997
[9] 廖建新，移动智能网，北京邮电大学出版社，2000年
[10] 马旭涛，朱晓民，杨孟辉，Parlay 网关中的 TCAP 协议映射研究，高技术通讯，2004年，第14卷，第12期
[11] 马旭涛，朱晓民，杨孟辉，TC-CORBA 关口中关键技术研究，计算机工程与应用，2004年，第40卷，第35期
[12] 朱其亮，郑斌，CORBA 原理及应用，北京：北京邮电大学出版社,2001年
[13] Michi Henning, Steve Vinoski, Advanced CORBA Programming with C++，清华大学出版社，2000年7月

已发表论文

[1] 林荣、廖建新、曹予飞、朱晓民，TC-CORBA 网关实现方案的改进，电信工程技术与标准化，2005 年 5 月

致谢

感谢廖建新教授对我的教导和培养；
感谢朱晓民、程莉老师对论文的悉心审阅和大量诚恳的建议；
感谢实验室各位老师在平时给予我的鼓励和指导；
感谢项目组的同事同学们在各方面给予我的大力支持；
最后，感谢各位评委老师在百忙中审阅本文。