RPC技术在移动互联网应用中的实现

RPC technology in the mobile Internet applications

摘 要

互联网高速发展的今天，社交、搜索、博客、微博等互联网应用层出不穷，这些应用面临一些共同的挑战：一是用户数和访问量巨大，百万、千万用户级别的应用比比皆是，其中有些系统的用户数甚至已经上亿；二是请求和访问量往往成倍增加，而不是线性增加。

为支撑如此大规模的用户和用户增长率，互联网应用的后端都是规模庞大的分布式集群，而RPC系统是其中最为核心的组件，各类应用甚至是其他分布式系统大都基于RPC系统构建。在移动互联网的应用场景中，RPC面临着诸多新的技术问题和挑战，特别是应用终端与平台后端之间的服务交互。

本文通过分析移动互联网应用开发时面临的问题，对Rpc技术原理进行学习和研究，结合移动互联网应用开发的关注点，实现一套完整的、轻量级的RPC系统，并使其方案具有实际的使用价值和参考意义。

关键词： 移动互联网　远程过程调用(RPC)　接口定义语言 (IDL)

Abstract

Today, with rapid development of Internet, social networking, search, blogs, microblogging and other Internet application, the application of some common challenges: one is the number of users and huge traffic, millions and millions user-level applications abound, some users have even hundreds of millions of systems; second it is the request and traffic tend to multiply, rather than linear increase.

To support such a large user and user growth, Internet application back-end are large distributed cluster, the RPC system is one of the most core components, all kinds of application and even other distributed systems are mostly based on RPC system building. In the mobile Internet application scenario, the RPC is faced with many new technical problems and challenges, especially the application back-end service interaction between terminals and platform.

Based on the analysis of mobile Internet application development when facing problems, to study and research of Rpc technology principle, the combination of mobile Internet application development focus, a set of complete, lightweight Rpc system, and its solution has practical use value and reference significance.

Keywords : the mobile internet, remote procedure call(RPC), Interactive Data Language(IDL)

目　录

1 绪论

1.1 研究工作背景

互联网高速发展的今天，社交、搜索、博客、微博等互联网应用层出不穷，这些应用面临一些共同的挑战：一是用户数和访问量巨大，百万、千万用户级别的应用比比皆是，其中有些系统的用户数甚至已经上亿；二是请求和访问量往往成倍增加，而不是线性增加。

为支撑如此大规模的用户和用户增长率，互联网应用的后端都是规模庞大的分布式集群，而RPC系统是其中最为核心的组件，各类应用甚至是其他分布式系统大都基于RPC系统构建。在移动互联网的应用场景中，RPC面临着诸多新的技术问题和挑战，特别是应用终端与平台后端之间的服务交互。所以通过学习、掌握RPC的理论原理，结合移动互联网的技术特点，设计开发一套完整的RPC技术方案具有比较现实的使用价值和研究意义。

RPC是SUN公司在70年代提出的通信数据交换的技术，其是面对过程的操作。随后出现的CORBA是面向对象语言的一种抽象，允许开发者进行跨机器、跨语言的通信，通过接口定义语言（IDL）指定远程对象的接口，用于生成远程系统中的对象接口在本机中的桩代码，并且在实际的语言实现与抽象接口之间生成映射关系。

CORBA现在基本已经不再使用，但是其基于多语言编程基础的接口模式一直是后来者借鉴的。平台级的RPC有J2EE的RMI技术、阿里的DUBBO、Zeroc ICE。前两个基本上用的很少了，ICE是目前广泛使用的工业级的高性能RPC框架。Thrift 是支持多语言开发的高性能RPC框架，但没有达到平台级别，其不具备服务注册表、负载均衡、以及服务部署、管理监控等一个平台必要的组件和功能。

现有的RPC技术框架基本都是通用型的，且基本都是面对平台后端服务系统的解决方案。可以说，以上这些RPC方案可以在局域网内工作的很好，特别是ICE框架，其覆盖了从简单的RPC调用请求调用到网格计算、集群部署、监控管理等各方面，虽然它也推出了移动互联网的客户端版本(Android、iOS、websocket），但由于其一体化的解决方案带来的问题就是过于的庞大和复杂，不能灵活的针对不同的应用需求做出快速适配和调整。例

如，其无法对新增程序语言或嵌入式平台的提供支持，此时仅仅掌握如何使用ICE的用户便显得无从下手。

1.2 研究工作意义

近几年的项目中，服务化和微服务化渐渐成为中大型分布式系统架构的主流方式，而 RPC 在其中扮演着关键的作用。 在平时的日常开发中我们都在隐式或显式的使用 RPC，一些刚入行的程序员会感觉 RPC 比较神秘，而一些有多年使用 RPC 经验的程序员虽然使用经验丰富，但有些对其原理也不甚了了。 缺乏对原理层面的理解，往往也会造成开发中的一些误用。

目前国内很少有针对移动互联网应用的RPC框架的轻量级实现，通过本课题的研究，分析清楚移动互联网应用开发中的分布式服务相关的关键问题，对如何实现轻量级RPC提出关键思路和技术方法，并通过开发实现一套完整的RPC框架，为RPC的研究、学习、应用提供很好的参考价值。

1.3. 移动互联网软件开发的技术特点和关注点

随着智能移动通信终端设备的普及，越来越多的互联网的应用和用户交互行为逐步迁移到了移动互联网，人们通过手机、PAD等移动终端设备可以不受时间、地点、环境的限制进行电子商务、社交、多媒体等活动。

移动互联网应用系统的构建主要包括2部分： 移动前端软件和后端平台服务系统。

面对互联网就是要应对高速的需求变化，随着业务需求的变化，前后端交互方式和技术变得越来越复杂。技术公司为了实现各自不同的需求目标，选择或者开发适合自己的技术或服务系统。移动前端软件的开发要面对相当多的问题，包括： 技术方案选型、NAT穿透、网络传输控制、身份与安全等等。

目前移动前端开发的方案基本有几种：Native、Html5、Hybrid。

- Native方式是开发者利用移动端系统自身提供的本地SDK组件、功能接口完成移动App所有功能的开发。其优点在于本地SDK可以提供最丰富的接口功能给开发者，利用这些功能，开发者可以实现所有的应用功能，获得最好的用户交互体验。Native是在特定平台上的特定的软件开发方式，其不能实现一份程序跨多个平台的可能，需要支持多个移动平台，必须开发多个平台的软件版本，这对于软件交付的一致性和快速更新迭代带来了很大的麻烦，且对开发人员的技能要求也比较高。

- Html5方式是提供一种低成本的移动App开发实现方法。移动App的业务功能由Html5的页面完成，通过在移动App软件中内嵌Webkit来呈现业务功能页面，这种开发模式快速流行起来 ，其降低了移动开发技术门槛，令一大批从事Web前端开发的技术人员利用Javascript也能快速高效地完成移动软件的开发，结合平台服务开发的NodeJs技术，可以实现javascript程序员的全栈式开发。 这种应用技术最早出现是PhoneGap开源项目，其提供了一个App软件访问本地系统功能的代理设施，业务代码透过这个代理可访问到本地系统的功能，例如： 照相机、文件系统、传感器、GPS等等。 但是这种开发方式完全是基于页面来实现的，Webkit完成数据的加载、呈现，其性能就无法与Native方式相比，往往在用户交互体验上被用户诟病，且由于这种方式不能直接访问系统功能和资源（必须通过开发代理才能扩展具备这种能力），不能很好发挥出系统的性能，所以Html5开发一般使用在轻量级的应用场景中，例如：微信公众号、淘宝商铺等等。

- Hybrid是目前最为流行的前端App开发技术，最具影响力的是Facebook的React-Native技术，FaceBook的社交系统开发过程也是经历了从Native到Html5，最终自行开发和使用React-Native。Hybrid的原理是通过ReactNative内嵌javascript解释器来执行用户交互和业务代码，这种模式可以最大化利用系统资源，并降低开发难度，并且保证一份程序在多平台上的运行一致性。

前端软件与平台后端服务的传输通常采用标准的HTTP协议，内容编码使用Form ，

XML, JSON格式。不同的系统平台均提供了Http数据处理的工具包，使得开发移动前端软件与平台交互的过程变得很简单。 基于Http协议提供的是 “请求-响应”式的交互方式，这种方式限制了必须是由客户端请求发起到服务器处理返回，由于这种模式够清晰、简单，所以一般的应用方案都使用这种技术方式。

互联网的应用需求的快速变化，使得单独的HTTP交互方式已经无法完全支撑。例如在社交系统中，即时通信软件必须要实现客户机与服务器之间实时消息流的传递，由于NAT和HTTP的制约，所以必须借助Tcp建立与服务器的Socket通信长连接来提供服务器发送给客户机的通道，这无疑提高了软件开发的技术难度。

在移动互联网场景中，对于数据传输的实时性和安全性也提出了高要求。 移动终端设备的网络带宽往往是受限制的，所以在设计传输协议时必须考虑流量控制，尽可能以最少的字节描述业务行为。采用基于Tcp的二进制编码方式在部分场合可以替代HTTP的文本编码。在有些对安全要求高的场景必须充分考虑对数据安全、交互安全和传输安全的处理。 在数据层面，应该对数据在前后端系统进行落地加密，在网络传输时，应该采用ssl/https的加密方案，对某些高级别要求的场合应该配置双向认证对终端用户身份进行验证，并配合指纹、声波等传感设备识别合法用户。 在应用层面，要求系统结构设计合理，模块、接口划分清晰，用户角色、权限进行控制，系统日志记录齐全。

Rpc 实现可以基于不同的通信协议、网络环境、操作系统， 开发移动互联网的Rpc框架作为移动互联网应用的基础服务，提供低成本的、可靠的、网络交互实现，避免重复造轮子，这是相当必要和应用价值的。

2. RPC 的理论原理

2.1 Rpc 的定义

RPC 的全称是 Remote Procedure Call 是一种进程间通信方式。 它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。即程序员无论是调用本地的还是远程的函数，本质上编写的调用代

码基本相同。

RPC 的主要目标是让构建分布式计算（应用）更容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。 为实现该目标，RPC 框架需提供一种透明调用机制让使用者不必显式的区分本地调用和远程调用。

2.2 起源

RPC 这个概念术语在上世纪 80 年代由 Bruce Jay Nelson（参考[1]）提出。 这里我们追溯下当初开发 RPC 的原动机是什么？

在 Nelson 的论文 *Implementing Remote Procedure Calls*（参考[2]） 中他提到了几点：

- 简单：RPC 概念的语义十分清晰和简单，这样建立分布式计算就更容易。
- 高效：过程调用看起来十分简单而且高效。
- 通用：在单机计算中「过程」往往是不同算法部分间最重要的通信机制。

通俗一点说，就是一般程序员对于本地的过程调用很熟悉，那么我们把 RPC 做成和本地调用完全类似，那么就更容易被接受，使用起来毫无障碍。 Nelson 的论文发表于 30 年前，其观点今天看来确实高瞻远瞩，今天我们使用的 RPC 框架基本就是按这个目标来实现的。

2.3 调用方式

RPC 调用分以下两种：

- 同步调用：

  客户端等待调用执行完成并获取到执行结果。

- 异步调用：

  客户端调用后不用等待执行结果返回，但依然可以通过回调通知等方式获取返回结

果。若客户端不关心调用返回结果，则变成单向异步调用，单向调用不用返回结果。

异步和同步的区分在于是否等待服务端执行完成并返回结果。异步调用由于不阻塞特性，所以可以在短时间内发起大量的请求，但往往请求返回的处理并不在调用请求的同一个执行上下文中，所以，编写异步处理代码相对于同步代码增加了不少难度。

2.4 模型

RPC 的程序包括 5 个理论模型部分：User 、User-stub 、RPCRuntime 、 Server-stub 、 Server。下图展示了一次Rpc调用的完整处理过程。

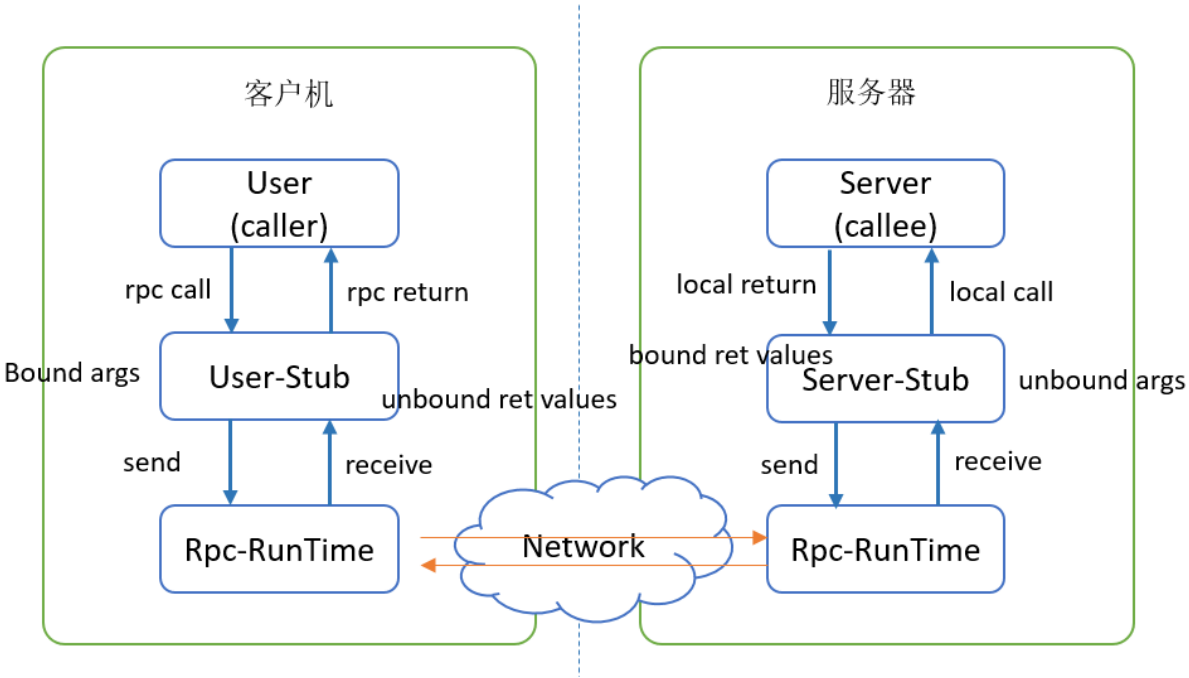


图 2 -1 Rpc 调用模式

在Rpc应用场景中，不管客户端或服务器，其中均包含三部分，分别是： 应用代码、桩代码（Stub）、Rpc运行库。

应用代码提供Rpc服务时，应遵循Stub生成的框架，实现Rpc接口功能的细节，当作为Rpc请求者时，应使用Stub代码中的Rpc访问设施进行远程过程调用；

Stub代码由IDL编译程序根据应用接口描述翻译而成，其内部封装了数据序列化、调用管理、消息处理等代码；

Rpc运行库提供统一的Rpc运行环境，提供Rpc服务的基础功能的实现和管理，其主要

提供接受上层应用请求消息，将其编码发送，接收返回消息进行解码并路由分派到上层用户应用代码的过程，其封装了实现的细节，应用代码一般不直接访问调用 到Rpc运行库的功能接口。

Rpc 调用过程：

1. 客户端应用程序调用UserStub中的接口函数，UserStub将接口参数进行编码并交给Rpc运行库
2. Rpc运行库将请求处理为Rpc消息通过网络传送给远程服务器
3. 服务器的Rpc运行库接收到客户机的消息，进行消息解析，并传递给Server Stub接口程序
4. Server Stub接口程序根据请求的接口和调用函数的标示，将请求路由传递给服务器应用层代码的服务函数中 (local call)
5. 服务器应用服务代码处理完毕，生成返回值交给Server Stub代码处理
6. Server Stub代码将返回值编码交给Rpc运行库，后者将返回值数据转换为Rpc消息，通过网络传送给客户机
7. 客户机的Rpc运行库接收到调用返回消息，进行解码，并将返回值传递给User Stub代码，后者再解析出对应的值，返回给应用代码。

2.5 面向接口定义 IDL

为了解决异构平台的 RPC，CORBA首先提出了 IDL（Interface Definition Language）来定义远程接口，并将其映射到特定的平台语言中。微软早年在Windows平台推出的DCOM技术也是基于IDL来定义远程服务的。

大部分的跨语言平台 RPC 基本都采用了此类方式，比如我们熟悉的 Web Service（SOAP），近年开源的ICE，Thrift 等都通过 IDL 定义，并提供工具来映射生成不同语言平台的 User-stub 和 Server-stub，并通过框架库来提供 RPCRuntime 的支持。 基本上每个不同的 RPC 框架都定义了各自不同的 IDL 格式。

IDL 是为了跨平台语言实现 RPC 不得已的选择，要解决更广泛的问题自然导致了更复杂的方案。 而对于同一平台内的 RPC 而言显然没必要搞个中间语言出来，例如 Java 原生的 RMI，这样对于 Java 程序员而言显得更直接简单，降低使用的学习成本。

客户端代码为了能够发起调用必须要获得远程接口的方法或过程定义。大部分跨语言平台 RPC 框架采用根据 IDL 定义通过 code generator 去生成 User-stub 代码，这种方式下实际导入的过程就是通过代码生成器在编译期完成的。我所使用过的一些跨语言平台 RPC 框架如 CORBAR、WebService（SOAP）、ICE、Thrift 均是此类方式。代码生成的方式对跨语言平台 RPC 框架而言是必然的选择。

RPC远程服务接口描述内容，主要包含：“远程接口定义 ”，“交换的数据类型定义 ”。

2.6 协议

协议指 RPC 调用在网络传输中约定的数据封装方式，包括三个部分：Rpc调用编码、 Rpc内容编码和传输控制编码。

不同的RPC框架出于不同目的，在编码实现时采用了不同的编码方案，粗略分为： 二进制编码和文本编码 两类。

2.6.1 调用编码

客户端代理在发起调用前需要对调用信息进行编码，这就要考虑需要编码些什么信息并以什么格式传输到服务端才能让服务端完成调用。 出于效率考虑，编码的信息越少越好（传输数据少），编码的规则越简单越好（执行效率高）。

Rpc的消息包主要分为请求调用和调用返回两种

调用编码

A. 接口方法： 包括接口名、方法名

B. 方法参数： 包括参数类型、参数值

C. 调用属性： 包括调用属性信息，例如调用附加的隐式参数、调用超时时间、异步同步控制等

返回编码

A. 返回结果： 接口方法中定义的返回值(包括void类型)

B. 异常： 调用异常信息

2.6.2 内容编码

编码是指在一个Rpc的消息包体内，对一次接口请求或返回处理中携带的数据的编码。 这些数据在IDL定义中以 不同的形式所表示，可以是： 数值型、布尔型、浮点型、字符型还有表示集合的数组和字典，更复杂的类型描述采用 复合结构类型描述。 这些数据类型在Rpc调用过程中以接口参数的形式传递给远程服务或从远程服务返回。

2.6.3 传输控制编码

传输时，在Rpc消息包体外围存在一层传输控制协议结构，其描述一个独立的传输消息单元，其内部包含诸如： 消息标识（Magic） ，压缩类型，加密，版本 等消息传输的控制信息，同时在面向流的传输(TCP)时，对数据进行粘包处理。

2.7 传输

协议编码之后，自然就是需要将编码后的 RPC 请求消息传输到服务端，服务方执行后返回结果消息或确认消息给客户端。RPC 的应用场景实质是一种可靠的请求应答消息流，这点和 HTTP 类似。因此选择长连接方式的 TCP 协议会更高效，与 HTTP 不同的是在协议层面我们定义了每个消息的唯一 id，因此可以更容易的复用连接。

既然使用长连接，那么第一个问题是到底客户端和服务端之间需要多少根连接？实际上单连接和多连接在使用上没有区别，对于数据传输量较小的应用类型，单连接基本足够。单连接和多连接最大的区别在于，每根连接都有自己私有的发送和接收缓冲区，因此大数据量传输时分散在不同的连接缓冲区会得到更好的吞吐效率。

所以，如果你的数据传输量不足以让单连接的缓冲区一直处于饱和状态的话，那么使用多连接并不会产生任何明显的提升，反而会增加连接管理的开销。

连接是由客户端发起建立并维持的，如果客户端和服务端之间是直连的，那么连接一般不会中断（当然物理链路故障除外）。如果客户端和服务端连接经过一些负载中转设备，有可能连接一段时间不活跃时会被这些中间设备中断。为了保持连接有必要定时为每个连接发送心跳数据以维持连接不中断。心跳消息是 RPC 框架库使用的内部消息，在前文协议头结构中也有一个专门的心跳位，就是用来标记心跳消息的，它对业务应用透明。

2.8 异常

无论 RPC 怎样努力把远程调用伪装的像本地调用，但它们依然有很大的不同点，而且有一些异常情况是在本地调用时绝对不会碰到的。在说异常处理之前，我们先比较下本地调用和 RPC 调用的一些差异：

- 本地调用一定会执行，而远程调用则不一定，调用消息可能因为网络原因并未发送到服务方。
- 本地调用只会抛出接口声明的异常，而远程调用还会跑出 RPC 框架运行时的其他异常。

本地调用和远程调用的性能可能差距很大，这取决于 RPC 固有消耗所占的比重。正是这些区别决定了使用 RPC 时需要更多考量。 当调用远程接口抛出异常时，异常可能是一个业务异常，也可能是 RPC 框架抛出的运行时异常（如：网络中断等）。业务异常表明服务方已经执行了调用，可能因为某些原因导致未能正常执行，而 RPC 运行时异常则有可能服务方根本没有执行，对调用方而言的异常处理策略自然需要区分。

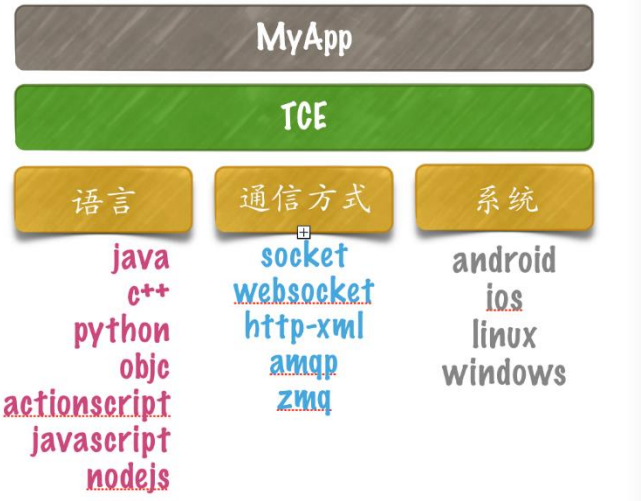
2.9 IDL 编译

RPC核心是IDL定义，使用IDL描述业务功能，RPC服务的提供者(Server)和使用者(User) 依赖IDL映射成自己使用的代码Stub。这些Stub的生成需要IDL编译程序将业务接口翻译成不同程序语言的代码。

3. 实现 RPC 框架

3.1. 简介

TCE(Tiny Communication Engine)是本课题研究开发的RPC框架名。 作为一个轻量级的Rpc实现，实现了Rpc理论模型的主体功能，结合移动互联网特点增加了多款实际应用价值的功能，例如：反向推送、长连接、心跳、外带数据等。



TCE的目标是提供一种便宜的Rpc解决方案，实现跨程序语言、通信方式、操作系统，作为分布式基础设施，为上层软件提供便利。RPC框架工具包包含以下内容：

- 接口定义语言
- 接口编译程序

- RPC 运行库

3.2. 接口定义语言设计

3.2.1 文档结构

IDL文档描述的内容主要包含 “数据类型定义” 和 “服务接口定义“

- “interface “ 远程服务接口，其内部由若干接口函数组成；
- “module” “interface“的集合。
- “type-def” 数据类型定义

可以使用任何文本编辑创建IDL文件。 在IDL文件中可以定义多个 “module”，“module” 包含多个“interface” 和“ type-def” 。

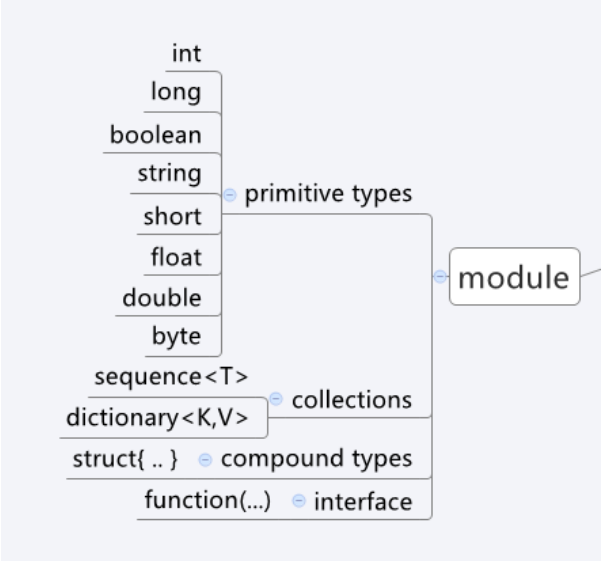


图 F-3-1 接口 IDL 构成元素

样例：test.idl

```
import base.idl
module Test{
        sequence<string>  NameList;
        struct ServiceNode{
            string address;
            int    index;
```

```
        }
        interface Server{
            string echo(string greeting);
        };
}
```

3.2.1 IDL 语法

关键字：

ID	Name	Description
1	import	导入其他 idl 模块
2	module	表示一个模块定义，可理解为定义 namespace
3	interface	服务接口定义
4	extends	接口继承
5	struct	复合数据集合
6	sequence	数组类型
7	dictionary	字典类型
8	<primitive types>	byte,short,int,long,float,double,string,bool 基础数据类型

表 3-1 接口定义关键字类型

3.2.2 数据类型

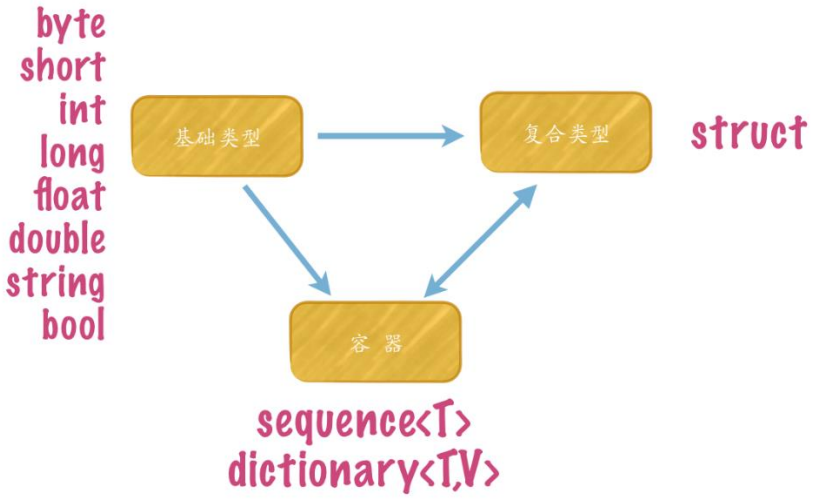


图 3-2 接口数据类型关系

3.2.2.1 基础类型（ primitive types ）

ID	name	size	C++	java	desciption
1	byte	1	Uint8	Byte	单字节（无符）
2	short	2	Int16	Short	双字节有符
3	int	4	Int32	Int	4 字节有符
4	long	8	Long	Long	8 字节有符
5	float	4	Float	Float	单精度
6	double	8	Double	Double	双精度
7	string	N+4	Std::string	String	字符串前导 4 字节长度域（ utf-8)
8	bool	1	bool	boolean	

表 3-2 基础类型长度规格

3.2.2.2 集合类型（ collections)

集合容器有两种，分别是 “sequence” 代表数组，“dictionary” 代表 字典。集合分别对应不同程序语言的collection类型。

idl	c++	python	java	as	object-c	csharp
sequence	std::vector	list	Vector	Array	NSArray	List
dictionary	std::map	dict	HashMap	HashMap	NSDictionary	Dictionary

表 3-3 容器类型与程序语言的映射

a.)数组 sequence<T>

Sequence提供数组对象的随机访问。sequence<T> 支持对任意数据类型的嵌套，也就是说， T可以是简单数据类型、容器类型、结构类型。

```
sequence<string> string_list;
sequence<int>   int_list;
sequence<string_list> tables;
struct Pet{ ... };
sequence<Pet> PetList;
dictionary<string,Pet> PetsNamed;
sequence<PetsNames> PetsNamedList;
```

b.)字典 dictionay<K,V>

dictionary 是字典结构（hash），κ 必须是简单数据类型，不能是复合对象;v可以是简单数据类型、容器类型、结构类型。

3.2.2.3 复合类型 struct

struct 提供了属性集对象的功能。从oo的角度来看，任何对象都可以是一个结构类型的定义 。

```
struct Cat {
    string  skin;
    int     age;
    int     categary;
    ...
 };
```

idl	c++	python	java	as	object-c	csharp
struct	struct	class	class		interface	class

表 3-3　复合类型与程序语言的映射

3.3 IDL 编译程序设计

3.3.1　简介

RPC在实际的应用开发中，RPC的接口定义IDL被翻译成不同程序语言的代码Stub才能在应用软件中使用。

Tmake是TCE项目的IDL编译程序，其功能就是　将IDL内容提取解释为不同的程序Stub。 Tmake位置在 $TCE/bin/tmake/ 目录中，为不同的程序语言提供不同的编译实现，例如Java 和 Python　则分别是　tce2java.py ， tce2py.py 。

Tmake采用Python语言开发，其对IDL的解析采用PLY 实现。

3.3.2 PLY 介绍：

PLY是 lex & yacc 的python版本。PLY 包含两个独立的模块：lex.py 和 yacc.py。lex.py 模块用来将输入字符通过一系列的正则表达式分解成标记序列，yacc.py 通过一些上下文无关的文法来识别编程语言语法。yacc.py 使用 LR 解析法，并使用 LALR(1)算法（默认）或者 SLR 算法生成分析表。(参考［1］PLY)

1. 词法分析

PLY词法单元的类型定义为tokens元组，元组的元素就是所有的词法单元类型。tokens元组定义了词法分析器可以产生的所有词法单元的类型，并且在语法分析时这个元组同样会用到，来识别终结符.

```
Tokens =
(  ‘ NUMBER’,’ PLUS’,’ MINUS’,’ TIMES’,’ DIVIDE’,’ LPAREN’,’ RPAREN’  , .. )
```

上述tokens元组就包括了所有的词法单元，必须采用tokens作为元组的变量名，这是PLY强制规定的。PLY各个词法单元的模式描述，可以采用正则表达式字符串或者函数来定义，但是必须采用t_TOKENNAME的模式命名，比如对于NUMBER类型的词法单元，变量名或是函数名必须是t_NUMBER。简单的正则表达式，当输入的字符序列符合这个正则表达式时，该序列就会被识别为该类型的词法单元

t_PLUS = r'\+'

当识别出词法单元时，还需要执行一些动作时，可以使用函数定义

```
def  t_NUMBER(t):
    r’ \d+’      #  描述模式的正则表达式
```

```
        t.value = int(t.value)
        return t    #  最后必须返回 t，如果不返回，这个 token 就会被丢弃掉
```

描述的正则表达式作为函数的doc，参数t是LexToken类型，表示识别出的词法单元，具有属性：value：默认就是识别出的字符串序列；type：词法单元的类型，就是在tokens元组中的定义的；line：词法单元在源代码中的行号 ； lexpos：词法单元在该行的列号。

2. 语法分析

文法规则的描述：

每个文法规则（grammar rule）被描述为一个函数，这个函数的文档字符串（doc string）描述了对应的上下文无关文法的规则。函数体用来实现规则的语义动作。每个函数都会接受一个参数p，这个参数是一个序列（sequence），包含了组成这个规则的所有的语法符号，p[i]是规则中的第i个语法符号。比如：对于序列中的词法单元，p[i]的值就是该词法单元的值，也就是在词法分析器中赋值的p.value。而对于非终结符的值则取决于该规则解析时p[0]中存放的值，这个值可以使任意类型，比如tuple、dict、类实例等。

```
def p_expression_plus(p):
    'expression : expression PLUS expression'
    #    |            |      |        |
    #  p[0]         p[1]   p[2]     p[3]

    p[0] = p[1] + p[3]
```

PLY实现简单的四则运算的文法定义例子：

```
statement : NAME "=" expression
          | expression
expression : expression '+' term
          | expression '-' term
term : term '*' factor
```

```
            | term '/' factor
factor : NUMBER
            | NAME
            | '-' expression
            | '(' expression ')'
```

3.3.3 执行编译

IDL的编译，需要解析IDL的内容，进而针对目的程序语言，将IDL内部的数据和接口定义翻译成目的程序语言的本地实现，其过程主要包括：词法定义、语法分析、程序翻译三个过程。

1.词法定义

```
tokens=(      'IDENTIFIER','STRUCT','NUMBER','INTERFACE',
              'SEQUENCE','DICTIONARY','EXCEPTION','COMMENTLINE',
              'IMPORT','MODULE','NAMESPACEIDENTIFIER','EXTENDS',
              'FILENAME','ANNOTATION_ATTRS','EQUALS','SCONST'
       )
```

名称	说明	词法描述
'IDENTIFIER'	变量名称定义	'[A-Za-z_][A-Za-z0-9_]*'
'STRUCT'	结构类型定义	'struct'
'NUMBER'	数值类型定义	r'\d+([uU]|[lL]|[uU][lL]|[lL][uU])?'
'INTERFACE'	接口类型定义	'interface'
'SEQUENCE'	数组序列定义	'sequence'
'DICTIONARY'	字典类型定义	'dictionary'
'SCONST'	字符串常量类型定义	'\"([^\\\n]|(\\.))*?\"'
'MODULE'	模块关键字	'module'
'EXTENDS'	接口继承关键字	'extends'
'COMMENTLINE'	行注释	'//.*\n'
'NAMESPACEIDENTIFIER'	模块空间定义(支持 A::B 格式)	'[A-Za-z_][A-Za-z0-9_]*\:\:[A-Za-z_][A-Za-z0-9_]*'

表 3-4　IDL 词分 Token

根据IDL的语法设计的这些词分，经过PLY的处理，依次提取出对应的Token，送入下

一步的文法处理。(见 $TCE/bin/tmake/mylex.py)

2. 语法定义

IDL的文法对接口数据类型、接口定义等进行了规定，例举简单的idl样例：

```
import base.idl
module Test{
        sequence<string>  NameList;
        struct ServiceNode{
            string address;
            int    index;
           NameList props;
        };
         dictionay<string,ServiceNode> ServiceNodeDict;
        interface Server extends BaseServer{
            string echo(string greeting);
             NameList getAlias();
        };

}
```

以上idl内容定义了 sequence、struct、interface, 其各自描述方式均有不同，tmake的文法处理使用PLY来描述这些规则。

2.1 module

```
'''module_def : MODULE IDENTIFIER '{' module_elements '}'   '''
```

描述一个module的定义区间。MODULE 是 关键字’module’ , IDENTIFIER 则是 应用模块名称 ‘Test’。 module_elements 表示module内部的元素，主要包括: sequence , dictionary, struct, interafce 四种子元素。

```
module_elements : module_element_def
module_elements :  module_elements module_element_def
```

```
module_element_def :  struct_def ';'
      | interface_def ';'
      | sequence_def ';'
      | dictionary_def ';'
      |
```

2.2 sequence

```
sequence_def : SEQUENCE '<' type '>' IDENTIFIER
```

Sequence描述一个带有 type 参数的数组类型，这有点类似程序语言中的“泛型参数”。样例中的： sequence<string> NameList; 声明了一种 NameList的字符串数组类型。 “Type 类型是允许类型嵌套的”

2.3 dictionay

```
dictionary_def : DICTIONARY '<' type ',' type '>' IDENTIFIER
```

Dictionary描述字典类型，其Key和Value都可以是任意数据类型 type 。 样例中:

dictionay<string,ServiceNode> ServiceNodeDict;

以上定义了一种称为ServiceNodeDict类型的字典结构，其K是字符型，V则是一个Struct复合数据类型。

2.4 struct

```
  struct_def : STRUCT IDENTIFIER '{' datamembers '}'
```

Struct描述一种复合对象类型，其类似与c/c++中的结构定义struct。 其datamembers可以是基础数据类型或者集合类型(sequence,dictionay),也可以包含struct的嵌套。

```
struct ServiceNode{
     string address;
     int    index;
    NameList props;
 };
```

以上接口描述中，struct定义了一个服务对象ServiceNode，其成员包含由string，int，seqence<string>组成的若干属性。

继续定义其内部成员datamembers的语法 ：

```
datamembers :  datamember
datamembers : datamembers datamember  #struct 的成员可以是一个或多个
datamember : type_id ';'              #成员以 ; 分割
type_id : type IDENTIFIER             #成员定义格式 : 类型名 + 标识符
type : IDENTIFIER
   | IDENTIFIER ':' ':' IDENTIFIER    # 支持命名空间方式定义
```

2.5 interface

接口是描述RPC服务具体为客户提供了哪些功能。 接口的语法定义包括接口(interface_def)和函数(operatemembers)两部分。

```
interface_def :
   | annotation_def INTERFACE IDENTIFIER '{' operatememberss '}'
   | INTERFACE IDENTIFIER '{' operatememberss '}'
   |  INTERFACE IDENTIFIER EXTENDS type '{' operatememberss '}'
   | annotation_def INTERFACE IDENTIFIER EXTENDS type '{' operatememberss
'}'
```

语法规定了接口内部由若干函数(operatemembers)构成，接口支持继承(EXTENDS)。

```
operatemembers : operatemember
operatemembers :  operatemembers operatemember
operatemember :  callreturn IDENTIFIER '(' operateparams ')' ';'
   | annotation_def callreturn IDENTIFIER '(' operateparams ')' ';'
operateparams : type_id
   |
operateparams : type_id ',' operateparams
type_id : type IDENTIFIER
type : IDENTIFIER
   | IDENTIFIER ':' ':' IDENTIFIER
```

Operatemember表示一个接口函数，函数的声明内容包括：　返回值(callreturn)、函数名称(IDENTIFIER)、参数列表(　operateparams) 。

进而又对函数参数做了描述：参数可以是 0 个或者多个，其格式为：”类型名　变量名”（ type_id : type IDENTIFIER)

```
interface Server extends BaseServer{
      string echo(string greeting);
};
```

以上样例代码中描述了一个服务对象Server,其继承自BaseServer,并提供了一个echo()的功能方法，参数和返回值均是string 类型。

3． 数据结构

在完成词法、语法设计之后，PLY可以帮我们提取出符合IDL描述规格的数据对象了，她们包括：数据类型、接口类型等等。这些对象在PLY处理过程中并没有被有效的组织，PLY也不知如何管理这些对象，我们将这些对象组装到一颗倒置的语法树，以便下一步在程序翻译阶段对数据进行遍历、处理。

IDL对象封装采用了容器与物件的思想，可以理解为每个module就是一个容器，那sequece,dictionry,struct,interface便是module的物件，其之间是父子嵌套关系。

在TCE的IDL设计中，module是最顶层容器，所以module是那颗倒置语法的树根，在处理PLY提取出的token对象时，需要将这些对象按照层次关系分别组装到这颗树上去。

这棵树每个节点都有其对应的数据描述对象，以下就分别介绍这些数据结构(参见: $TCE/bin/tmake/lexparser.py) 。

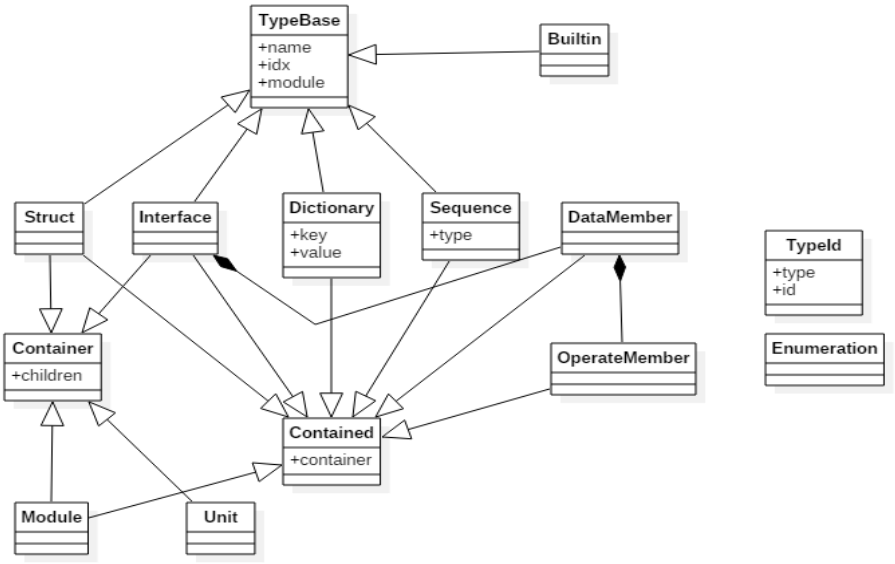


图 3-3 IDL 编译程序功能类图

4. 执行编译

4.1 关键字处理

在翻译IDL到程序语言时，必须注意IDL中IDENTIFIER命名与程序语言冲突问题，也就是说IDL定义的类型名和变量名不能与程序语言的关键字相同。

例如 Java关键字包含：

```
['for', 'import', 'float', 'new', 'class', 'interface',
'extends', 'while', 'do', 'package']
```

在定义IDL时如果也出现以上关键字，则编译程序应通过添加后缀字符之类的方式自动转换标识符命名，例如: for 转换为 for_

4.2 基础类型转换

TCE的基础类型分别是: 'byte' , 'bool', 'short', 'int', 'long', 'float', 'double', 'string',

翻译成Java依次为: Byte,Boolean,Short,Integer,Float,Long,Double,String。

样例:

IDL	Jave
string echo(string greeting);	String echo(String greeting);

数据序列化:

(java)生成的序列化代码中，使用工具类 DataOutputStream 完成数据的序列化和反序列化操作。

```
ds.writeByte, ds.writeShort,ds.writeInt,ds.Long,ds.Float,ds.Double
```

字符串序列化时，Tce将字符串内容以utf-8编码，并在字符串数据前添加4字节长度来描述后续字符串内容的大小字节。

```
bytes = s.getBytes()
ds.writeInt( bytes.length)
ds.write( bytes,0 , bytes.length)
```

网络字节序问题:

Java默认使用BigEndian,当翻译成其他程序语言时，须注意手动转换数值从主机序到网络序，在接收端进行反向处理。

4.3 数组类型

sequence<T> 进行数组类型的定义，T 可以是任意的已定义的数据类型，包括: 简单

类型、数组、字典和结构）。

sequence被映射到java的类型 Vector,他们具有类似的语义。以下示例定义了字符串数组类型 NameList，并在接口函数中使用。

```
sequence<string> NameList;
interface Server{
    NameList getAlias();
}
```

```
//Java
Class Server {
public Vector<String> getAlias(RpcContext ctx){
     return new Vector<String>();
   }
}
```

NameList被翻译成Vector<String>类型，并在接口Server::getAlias() 中被使用。

字节数组的特殊处理：

Sequence<byte> 描述字节数组，在很多应用场景中，很多二进制数据类型都可以用sequence<byte>来表述，比如：图片、声音、流媒体等等。对数组的编译默认是将其映射到 Vector<T>(java)类型，但这不适合字节类型，这会造成内存资源浪费和访问性能下降。实际的应用是需要提供对字节数组的进行随机、连续的访问，所以TCE将sequence<byte>映射为字节数组 byte[]。

4.4 字典类型

Dictionary<K,V> 描述字典数据类型，其可以对应java的 HashMap<K,V> 类型。 V可以是idl定义范围内的任何有效定义的数据类型。考虑到K键值hash生成的复杂性，TCE限定K必须是简单数据类型，不能是sequence,dictionary,struct。

4.5 结构类型

Struct 是复合对象类型，其内部可以由若干其他数据类型构成，并且允许Struct嵌套定义。 通常在描述赋值数据类型对象时被使用。

4.6 接口类型

interface 描述Rpc远程服务，其内部若干功能函数组成。interface 通常被翻译为程序语言的接口类型或者抽象类。Rpc服务程序编写时，需要从这些接口或抽象类派生，通过复写功能函数来提供业务功能。

编译程序通过扫描PLY生成的语法树，找到interface的定义，为interface生成 指定程序语言的抽象类定义和函数接口，除此之外编译程序还得完成两部分工作：生成 调用代理对象 和 服务端对象委托。

- 调用代理（Proxy）：是应用软件访问Rpc服务的客户端基础设施，实现与远程服务通信交互的细节，并提供多种调用模式(同步、异步、超时等等) 给上层应用软件使用。
- 对象委托(Delegator) : 委托对象是Rpc服务器侧进行数据反序列化、消息分派路由到用户Servant对象的基础设施。 编译程序根据接口定义的服务类，自动生成delegator类对象。

Rpc的请求调用是基于接口和接口函数进行的，在处理接口时，编译程序自动分配接口和接口函数的序号，默认从 0 开始。

4.7 标注 Annotation

为了方便控制编写idl和生成语言框架代码，在最近的tmake中支持对接口和函数进行

元属性定义。

元属性，这个概念类似C#,java中的标注Annotation,我借用了C#的语法，在 接口和函数定义之前可添加标注属性。属性定义的格式 ： [key=value,key=value,...]

目前支持的key类型:

- index - 自定义接口序列化编号
- comment - 描述
- skeleton_xxx - 规定接口实现是否生成服务端代码

样例:

```
module test{
    [index=11,comment="base server"]
    interface BaseServer{
        [index=10]
        string datetime();
    };


[skeleton_js=false,skeleton_objc=false,skeleton_as=false,skeleton_cpp=false,skeleton_
csharp=false,skeleton_java=false,comment=""]
    interface Server extends BaseServer{
        string echo(string text);
        [index=5,comment="test"]
        void  timeout(int secs);
        [index=10]
        void heartbeat(string hello);
        [index=11]
        void bidirection();
    };
}
```

在以上idl定义中，将 BaseServer的序列化编号定义为11（当然这个序列化编号对用户是透明的，无需关心，由tce自动维护，但是在软件版本变更的场景时是必须的， 因为接口增加、删除会打乱原始的序列化，所以提供一种方法令用户固定唯一的接口编号。

Server接口定义了诸多 'skeleton_xxx'，每一项对应不同程序语言的输出控制，如果是false,则 Server接口不输出服务侧代码。

3.4 运行库结构设计

TCE运行库包含几个重要部件：

- Communicator 通信管理器
- Adapter 通信适配器
- Servant 接口服务对象
- Proxy 请求代理
- Connection Rpc网络连接
- Endpoint Rpc服务资源的描述

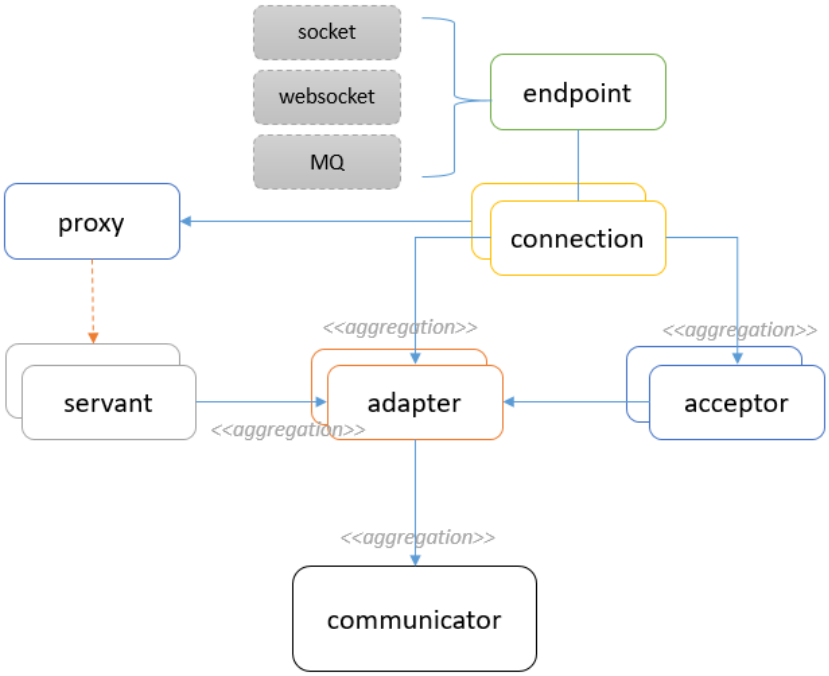


图 3-4 RPC 运行库主体模块结构

3.4.1 Communicator 通信器

Communicator是Rpc框架运行的全局管理对象，负责Rpc服务的初始化，启动、终止和管理工作，其提供Rpc服务各个设施运行的环境,管理和协调Rpc内部运行时的各类资源和

部件，基本功能包括：

- 初始和配置
- 本地服务对象管理
- 通信管理
- 消息分派

3.4.2 Servant 服务实体

Servant是指一个Rpc服务接口的具体实现。IDL接口类被翻译成Server-Stub框架代码，这些框架代码的功能被用户程序实现（往往通过继承重载的方式完成）。Servant必须加入Adapter之后，方可被远程用户访问。

3.4.3 EndPoint 端点

Endpoint是对网络通信服务端口的描述，其描述格式：

```
[scheme:][//host:port][path][?query]
```

- Scheme - 表示具体的应用协议类型，例如 http , tcp , ws
- Host:Port - 标识主机地址和端口。
- Path - 访问路径，表示具体的功能模块分割。
- Query - 具体的通信参数设置

Endpoint在Rpc客户端请求远程服务时，描述的是远端服务器的通信端口被代理Proxy使用；在服务器侧用于描述一个本地通信服务端口，被Acceptor所使用。

3.4.4 Connection 连接对象

Connection描述一个Rpc客户端与服务端建立的网络通信连接。 连接可以是临时的(短连接)或者是持久的(长连接)。移动互联网应用中，TCP协议和HTTP协议是前端与平台服务器之间交互通信的标准协议。

Connection封装了通信实现的细节，用户无需关心其内部实现。Proxy直接操作Connection，通过Connection进行消息的发送和接收。在服务器侧，Acceptor负责创建和管理Connection，一个Connection表示一个前端设备的连接进入。在移动互联网应用中，有很多使用长连接的情况，其主要是为了解决NAT后的设备接收服务平台侧发送实时通知消息问题。

3.4.5 Acceptor 服务接受器

Acceptor提供对外侦听网络服务的功能，Acceptor可以由不同通信方式和协议实现，例如：socket-acceptor ， http-acceptor 或mq-acceptor。

Adapter创建和管理这些Acceptor。客户端建立与acceptor连接，acceptor创建工作线程完成数据的接收处理，并将数据翻译成Rpc接口调用消息，再送入Adapter。

3.4.6 Adapter 通信适配器

Adapter称为通信服务适配器，其主要功能是：

- 通信服务功能：adapter可以 绑定多种通信协议进行服务侦听(acceptor)，客户代理(Proxy)发起连接到达Adapter并将Rpc调用请求传递给Adapter内的Servant对象。
- 服务对象Servant的容器功能：在TCE中，Servant作为Rpc服务接口的实现，要让Servant工作起来，必须将Servant添加进Adapter的管理容器，Adapter接收远程Rpc请求之后，根据请求的接口标识，在管理容器中找到对应的Servant对象，并执行Servant的接口函数。

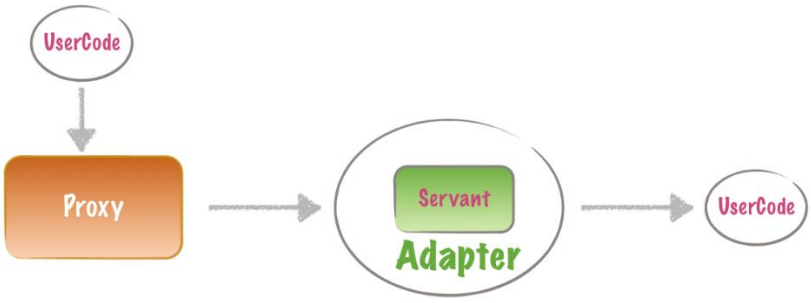


图 3-5 Adapter 功能图

3.4.7 Proxy 代理服务对象

Proxy是Rpc框架中访问Rpc服务的客户端设施，Proxy包装了远程过程调用的处理细节，提供本地函数接口来访问远程过程。

Proxy代码由IDL编译程序根据idl定义自动生成，其内部自动生成了多种调用模式代码，包括：阻塞、异步、单向、超时。Proxy内部建立、管理Connection对象来完成网络通信建立、数据编码、数据传输控制的过程。

```
def Proxy():
   ep = tce.RpcEndPoint(host='localhost',port=16005)
   return  ServerPrx.create(ep)
prx = Proxy()
prx.echo("hello")
```

以上代码使用 ServerPrx的静态方法create() 创建了一个远程代理对象prx ， 并发起了一次 echo方法的调用。

3.4.8 运行控制

服务器Rpc初始化：

```
tce.RpcCommunicator.instance().init('server')
ep = tce.RpcEndPoint(host='',port=16005,type_='socket')
adapter = tce.RpcCommunicator.instance().createAdapter('first_server',ep)
servant = ServerImpl()
adapter.addServant(servant)
tce.RpcCommunicator.instance().waitForShutdown()
```

以上代码演示了Rpc初始化过程，定义了一个本地服务侦听端点Endpoint，创建服务实现对象servant，并将其加入到first_server的通信适配器中，打开通信适配器开始提供Rpc远程服务，然后等待程序运行终止。

3.5 消息编码

TCE的消息封包包括两类：

- Rpc接口调用的消息封包
- TCP传输控制的数据封包

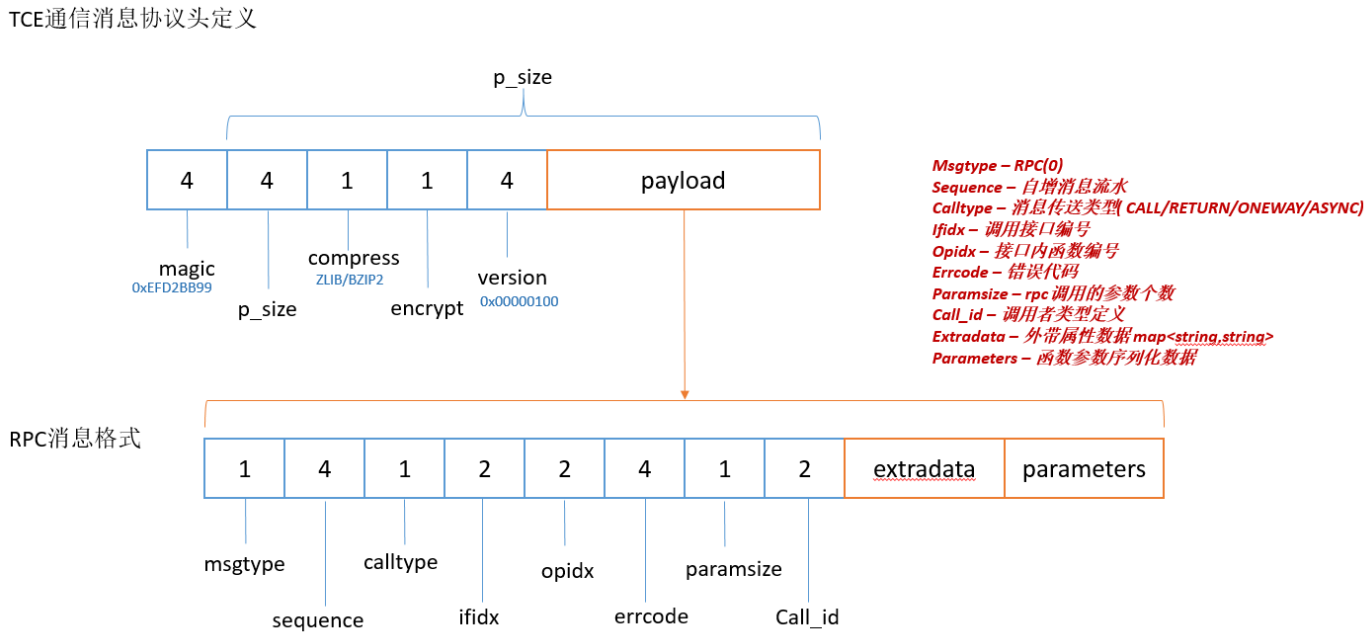


图 3-6 Rpc 通信传输协议定义

3.5.1 TCP 封包:

对于Tcp传输来说，Rpc调用消息是Tcp传输封包的内容(payload)。每个Tcp封包都具有14字节的头信息 和不定长的payload数据。头信息的作用是描述和控制payload具体传输、处理的方法。

- Magic是个固定的幻数值(0xeeffaacc) 用于识别此封包是TCE专用。
- P_size 描述了整个Tcp封包的长度，但不包括magic大小

- Compress 表示压缩算法 (0 - NONE ; 1 - ZLIB ; 2 - BZIP2) 低于100字节不压缩。
- Encrypt 加密算法 (0 - NONE ; 1 - DES ; 2 - AES)
- Vertion 版本

3.5.2 RPC 消息封包:

Rpc消息格式主要包括三部分: 消息控制头、外带数据区Extradata、参数内容Parameters。

消息控制头

- Msgtype 固定为1 ，表示此为RPC消息
- Sequence 表示一次Rpc消息的流水编号，编号由Rpc请求发送者生成，并在RETURN消息中夹带回来，用于Request/Response的调用返回的匹配。
- Calltype 表示此Rpc消息类型，主要分为： CALL 和RETURN
- Ifidx 远程服务接口编号
- Opidx 服务接口内函数编号
- Errcode 远程Rpc过程返回的异常编号
- Paramsize 描述Rpc接口函数的参数个数
- Call_id 调用者编号（保留 ）

Extradata - 表示外带数据，其类型是 dictionary<string,string>，TCE除了传递IDL定义的接口函数中的参数，还提供了一种不改变接口原型的情况下传递额外数据的方法。

Parameters - 表示一个idl接口函数的参数序列化数据内容，同时也表示RETURN的返回值序列化内容。

3.6 异常处理

在分布式的系统中，Rpc服务的实现过程往往并不在调用者的相同进程，或者在远端的某一台服务器上，这种调用是不可靠的，为了捕获这种不可靠就要使用异常处理。

异常在TCE中分为两类：本地异常和远程调用异常。

- 本地异常是指程序在使用和运行本地Rpc运行设施的过程中产生的异常，可以是操作异常、数据异常和网络异常。
- 远程调用异常是指一次调用传递到服务器之后，在服务器的执行过程中产生的异常，由服务器侧的Rpc运行环境捕获的异常，并通过网络传递返回到Rpc调用者。

这两种异常都需要通过程序语言的捕获机制进行处理(try .. catch) 。

异常类定义:

```
class RpcException(Exception):
    def __init__(self,errcode,errmsg='',data=None):
        self.errcode = errcode
        self.errmsg = errmsg
```

在Rpc消息结构定义（3.5.2）中 errcode字段描述消息的错误码，非0表示Rpc调用异常返回消息。

Name	Value	Description
RPCERROR_SUCC	0	成功
RPCERROR_SENDFAILED	1	网络发送失败
RPCERROR_DATADIRTY	2	数据非法
RPCERROR_TIMEOUT	3	调用超时
RPCERROR_INTERFACE_NOTFOUND	4	Rpc 远端接口不存在
RPCERROR_UNSERIALIZE_FAILED	5	数据序列化错误
RPCERROR_REMOTEMETHOD_EXCEPTION	6	远端 Rpc 过程执行异常
RPCERROR_DATA_INSUFFICIENT	7	消息报文格式错误
RPCERROR_REMOTE_EXCEPTION	8	远端执行故障
RPCERROR_CONNECT_UNREACHABLE	9	连接不可达
RPCERROR_CONNECT_FAILED	10	连接失败
RPCERROR_CONNECT_REJECT	11	连接拒绝

RPCERROR_CONNECTION_LOST	12	连接丢失
RPCERROR_INTERNAL_EXCEPTION	13	本地 Rpc 执行故障

表 3-5 异常类型定义

3.7 网络传输控制

TCE默认采用Tcp协议来控制Rpc消息报文的传输。Rpc请求根据不同的调用模式创建不同的消息封包在一个网络连接上双向传递。

TCE不保证消息报文能正确被发送和接收， 其原因是在传输的过程中造成失败的原因有很多，可能是网络故障问题或者系统某个环节出了问题，如果让Rpc框架来一一处理这些问题，保证可靠的传输控制，那必须并将这些处理细节的控制与上层用户代码进行紧密的耦合，这就背离了TCE简单易用的设计目标。

懒惰特性：TCE中的网络连接对象是具有Lazy特性的，只有当Proxy发起第一个远程调用时，Connection才会发起真正的网络连接。当连接断开时，不进行网络重连工作，上层应用软件并不能感知到网络已断开(无需关心)，只有当Proxy发起新的远程调用时，Connection才会重新建立网络连接。

3.8 调用模式

在TCE中，Rpc消息交换的类型有5种，见RpcMessage(java)定义：

1	CALL	0x01	请求消息
2	RETURN	0x02	返回消息
3	TWOWAY	0x10	请求调用有返回值
4	ONEWAY	0x20	请求调用无返回值
5	ASYNC	0x40	异步调用标识

表 3-6 Rpc 消息包类型

3.8.1 two-way

two-way描述的是一种有返回消息的Rpc同步调用模式，调用发起并等待返回。 这种调用方式常见与c/s交互场景或实时性效率不高的场景。阻塞调用对使用用户来说是最易于理解和使用的模式，发起调用之后，执行线程将一直保持阻塞，直到到服务处理返回或者异常产生。

这种同步调用方式虽然简单且容易理解，但其会阻塞执行者线程，效率就非常低下，不能在平台服务器侧使用，同样也不能使用在移动终端软件的开发。

Rpc Message Type: CALL | TWOWAY	
Idl	Java
String echo(string hello)	String echo(string hello,RpcExtract ctx)

3.8.2 one-way

oneway模式是指对接口中无返回消息的函数的调用。 因为没有返回，所以调用者无需等待，执行线程并不会被阻塞。 oneway的接口函数在idl定义时必须是 void 类型。

TCE 为 void 接口函数自动生成 以 _oneway后缀的函数名，例如：

Rpc Message Type: CALL | ONEWAY	
Idl	Java
Void heartbeat(string msg)	void heartbeat_oneway(string hello)

3.8.3 async-call

async调用在用户发起请求之后，通过回调函数来接收返回值。 TCE自动生成后缀"_async"的函数名。

Rpc Message Type: CALL | ASYNC	
Idl	Java
string echo(string msg)	void echo_async(string hello,asyc_callback cb)

```
    def hello_callback_async(result,proxy,cookie):  #回调接口
        print 'async call result:',result

    prx.echo_async('pingpang',hello_callback_async,cookie='cookie',extra={})
```

3.8.4 timeout-call

timeout-call提供阻塞调用等待超时的功能，同步调用方式是最容易理解和操作的方式，用户可以指定期待处理的等待时间，在调用发起后等待返回，直到超时发生。TCE自动生成 后缀"_timeout"的函数名。

```
    try:
        print prx.timeout(3,6,extra={})
    except tce.RpcException, e:
        print e.what()
```

3.8.5 bidirection

基于Rpc的调用往往都是单向的，其使用规则也相当简单，便是：A调用B的远程接口，那A主动建立B的网络连接，反之亦然。

在互联网应用环境中，客户机往往都是安置在NAT之后，服务器与客户机通信必须由客户机主动发起对服务器的连接进而获取服务器资源，那问题来了，如何让躲在NAT之后的客户机的Rpc接口服务提供给服务器调用呢？ 因为服务器无法主动连接到NAT之后的客户机，所以，解决的办法只有通过复用客户机建立到服务器的连接来传递服务器发送的Rpc消息到客户机， 这种实现方式就是bidirection。要注意的是由客户端发起的连接必须保持。

在3.10.1章节的程序中演示了bidirection过程，其中定义了客户端接口 Iterminal 并创建类TerminalImpl实现其功能。

```
   interface ITerminal{
     void onMessage(string  message);
   };

#python
   class TerminalImpl(ITerminal):
     def onMessage(self,message,ctx):
```

```
        print 'onMessage:',message
```

为了接收Rpc请求调用，在客户端创建适配器Adapter，实例化TermimalImpl这个servant加入到Adapter，然后创建服务器的访问代理对象，并将此访问代理关联到Adapter。

```
adapter = tce.RpcCommAdapter('adapter')
impl = TerminalImpl()
adapter.addServant(impl)
tce.RpcCommunicator.instance().addAdapter(adapter)

ep = tce.RpcEndPoint(host='localhost',port=12002)
prxServer = ServerPrx.create(ep)
adapter.addConnection(prxServer.conn)
```

服务器要调用客户端的接口，必须要求客户端发起对服务器的一次远程调用，在服务器的代码中获取本地调用的通信连接对象，并由此连接构建出远程客户端接口ITerminalPrx的访问代理。只要客户端的连接没有断开，此访问代理对象一直有效。

```
# server.py
def bidirection(self,ctx):
        self.clientprx = ITerminalPrx(ctx.conn)
        self.clientprx.onMessage_oneway('server push message!')
        #调用 client 端接口方法，采用单向
```

3.9 异步编程

在移动互联网应用开发过程中，服务节点之间、客户机与服务器之间的网络通信采用异步通信方式可以获得最大的性能。但是相对于同步编程来说，编写异步程序的复杂度却是高了很多。

同步与异步的编程差异主要是在对于调用状态和错误的控制不同。同步模式降低了编程的难度，同步模式下所有操作的请求和返回都是串行的处理，连续的操作只有等待前者调用完成返回才可进行后者的请求，这种方式严重造成了系统IO资源的浪费，在主流的移动端App开发环境中为了保证系统运行的流畅和用户体验，在 IO处理时是不允许同步调用的，特别是在服务平台系统的开发时必须保证所有的IO、网络请求是异步实现的。

在同步编程时，对状态和错误的控制很简单，通过函数返回值或者捕获异常就可以进一步决策程序的分派。但在异步编程时，客户程序不会等待服务器程序的处理返回而直接运行完毕所有的异步请求代码，并通过设置相应的接收函数来获得服务器处理返回的结果，期间为了使调用请求和数据返回能匹配关联上，采用令牌Token方式标识一次请求调用，并在客户机与服务器之间传递。

RPC 的实现包括服务端和客户端，分别作为 Rpc 服务的提供者和使用者，以下分别讨论两种角色的实现细节。

3.9.1 服务端处理

Rpc服务端在启动之后侦听网络端口(Endpoint),当请求到达时（Connection被建立 ）解析二进制数据包为一个Rpc消息，并根据Rpc消息规格的定义，将消息分派到具体编号的接口对象实现者(Servant)的函数上，这个过程通过Rpc运行库完成。 用户可以通过设置回调方式(callback)或者重载(override)来获得函数的执行，并通过return将Rpc返回结果传递给 Rpc运行库，由后者再传递Rpc客户端。

服务端的异步处理体现在消息分派和接口函数执行的异步控制。这两部分需要设置各自独立的工作线程池。Adapter的功能是管理接口对象（Servant）,并提供访问此对象的网络服务能力，在每个Adapter中可以设置通信数据处理线程池，其作用是接收和发送Rpc请求和返回消息，并执行Rpc消息的编码和解码工作。 当Rpc消息被调度到Servant对象函数时，必须保证有足够的工作线程去执行这些Servant对象的函数，这些工作线程的数量决定服务端对Servant对象提供服务的响应能力。

3.9.2 客户端处理

在3.8章节中描述内容了Rpc客户端的调用模式。在移动互联网应用中，异步模式AsyncCall是首选的。

Rpc代理Proxy是Rpc服务访问的基础客户端部件，不同的业务接口都有对应自身接口的代理部件，这是通过IDL编译程序自动完成的。（编译程序根据IDL定义翻译生成Rpc骨

架Stub代码和对应的调用代理对象Proxy代码）

Rpc客户端主要处理两个内容： ***异步接收*** 和 ***异常处理***

异步接收通常采用回调方式实现，例如以下python示例：

```
def hello_callback_async(result,proxy,cookie,error):
    print 'async call result:',result

prx.echo_async('pingpang',hello_callback_async,cookie='cookie',extra={})
```

以上代码中，调用服务器的echo( )方法时，采用了异步调用 echo_async模式 。在echo_async( )原型中，增加了第二个参数用于设置异步数据接收函数。

Hello_callback_async() 函数作为 echo 调用的接收，其参数分别表示:

- Result - 返回值
- Proxy - 调用时的本地代理对象
- Cookie - 应用于上下文的usercode
- Error - Rpc请求调用的异常

异常处理

客户端的异常包括：本地调用异常和远程服务端处理异常。 本地异常内容可以是数据处理异常，比如Rpc数据输入容错异常、Rpc运行库异常、网络通信错误等等。 远程异常内容可以是数据序列化异常、远程Rpc运行库异常、Rpc消息分派异常（匹配接口和函数失败）和远程服务函数执行异常等等。

异常均以异常编号加上异常信息来描述，并组装成本地异常对象通过异步调用的回调函数中的error字段抛给用户处理。

异步处理过程：

用户通过代理Proxy发起对Rpc接口的异步调用(_async) ， Proxy代理将输入参数编码成Rpc消息m1，创建调用令牌Token，并将Token与Rpc消息和回调接口对象cb一并置入Rpc运行库的发送队列qs中；

Rpc运行库的发送线程ts从发送队列中提取出Rpc消息，创建或匹配到与目标服务器网络连接对象Connection，并通过Connection将Rpc消息发送出去，同时将Token、Rpc消息置入到Rpc等待队列 qw。

当Rpc处理返回时，Rpc运行库的接收线程tr从Rpc消息m2中获取Token，并从qw队列中根据Token找到m1消息对象，进而获得异步回调接口对象cb，最后调用cb，将Rpc结果传递给用户。

3.9.3 Promise 使用

Promise作为客户端异步编程的代表模式而出现，Promise充斥在js各个角落，各种知名第三方软件项目都存在类似Promise的组件。那Promise是个什么样的东西，且为何要使用Promise呢？

Promise表示一个未发生的执行事件的定义，在Java的Netty项目中对应的组件叫做`Future`.现在越来越的应用系统开发采用异步IO来提高系统处理的能力，且事实证明其的确是高效。

Javascript ES6已经内置了Promise组件, js所有的io操作均是异步操作。 Android通过handler处理异步操作，防止主线程被阻塞。 Tornado , NodeJs, epoll, Netty, libaio, libev 都是相关的异步处理技术。但是异步处理也带来的开发的复杂性，使得交互处理行为无法被串行处理，不同的操作步骤被分割到不同的异步回调函数中钩挂，使得代码可读性、可维护性变得很差。

Promise的作用是将 异步调用的嵌套 转为同一平面的调用，提供类似同步调用的方式。

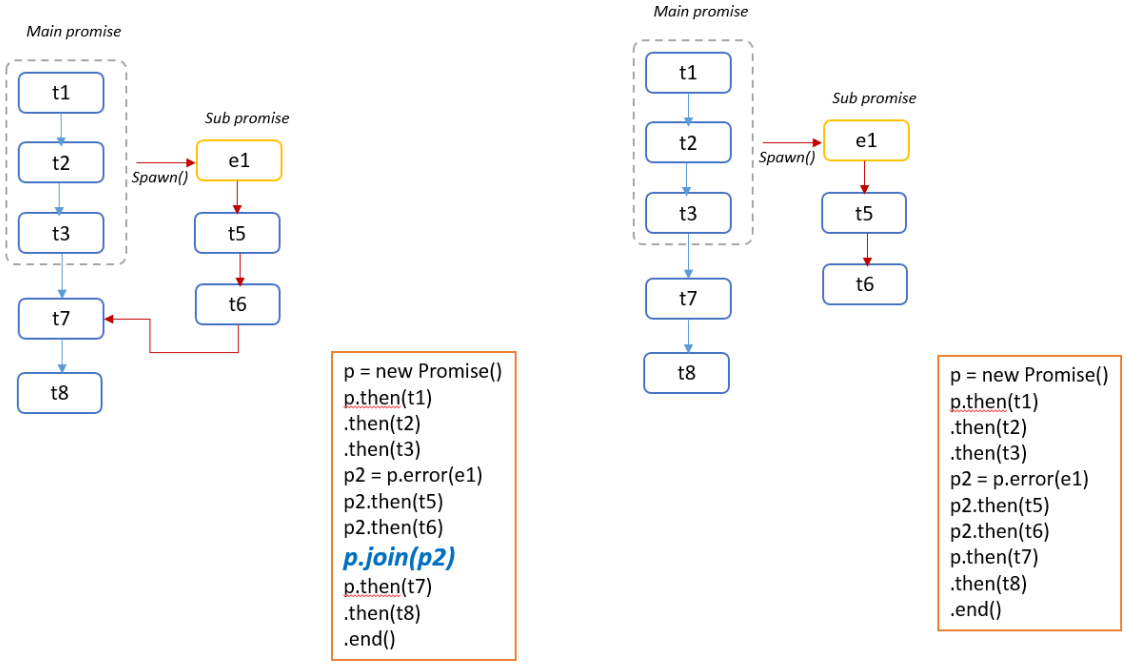
如何实现Promise

TCE的client调用模式中的异步调用在使用时需提供返回接口callback，callback实现方式可以是函数地址，委托，接口方式。

异步带来的问题在于调用嵌套，例如：a,b,c是一组连续调用，且后者必须在前者调

用成功之后 被调用，这就导致tree嵌套调用，使得代码可读性，可维护性变得很糟糕。针对这种异步调用嵌套的解决方案就是Promise，类似链表管理节点一样链接不同的调用，提供一种同步调用的方式。异步调用不再是返回void了，而是RpcPromise对象，然后通过promise.then(result,error)来驱动串行调用。通过promise传递两次调用的输出和输入值。需要在异步接口函数加入promise对象，用于保存本次异步调用需向后一个调用传递的值。promise.then()的回调函数也需增加promise输入来获得上一处理结果的输出。

Promise的异步调用



3.10 TCE 使用示例(python)

这里将用代码演示整个Rpc服务调用的过程，首先编写idl服务接口，编译生成stub代码，然后编写 server程序以类继承方式实现功能接口，编写client程序，请求server的远程接口。

运行步骤：

1.编写test.idl

2.编译idl

python $tce/tce2py.py -i test.idl ./ 生成python的stub文件 test.py

3.执行

1. 运行server ， python server.py
2. 运行client ， python client.py

3.10.1. 接口定义

```
module test{

    interface BaseServer{
        string datetime();
    };

    interface Server extends BaseServer{
        string echo(string text);
        void  timeout(int secs);
        void heartbeat(string hello);
        void bidirection();
    };

    interface ITerminal{
        void onMessage(string  message);
    };

}
```

client - 实现ITerminal接口，提供server的反向调用;

server - 实现 Server 功能接口

以上接口定义中描述了三个接口，分别是客户端接口ITerminal、服务端接口Server和基础服务接口BaseServer。在客户端和服务端必须分别实现各自服务接口的内容。 接口ITerminal::onMessage() 方法用于接收服务器反向推送消息。

3.10.2. server 代码

```
class ServerImpl(Server):
```

```
    def __init__(self):
        Server.__init__(self)
        self.clientprx = None

    def echo(self, text, ctx):
        print 'extra oob data:',ctx.msg.extra.props
        return 'Yah! '+text

    def timeout(self,secs,ctx):
        print 'enter timeout:',secs
        time.sleep( secs)

    def heartbeat(self,hello,ctx):
        print hello

    def bidirection(self,ctx):
        self.clientprx = ITerminalPrx(ctx.conn)
        self.clientprx.onMessage_oneway('server push message!')

def main():
    tce.RpcCommunicator.instance().init('server',’settings.yaml’)
    adapter  = tce.RpcAdapterSocket.create('adapter',’server')
    servant = ServerImpl()
    adapter.addServant(servant).start()
    tce.RpcCommunicator.instance().waitForShutdown()
```

3.10.3.client 代码

```
class TerminalImpl(ITerminal):
    def __init__(self):
        ITerminal.__init__(self)

    def onMessage(self,message,ctx):
        #这里接收 server 推送的消息
        print 'onMessage:',message

def call_twoway():
    print prxServer.echo("hello")

def call_timeout():
    try:
        print prxServer.timeout(3,6)
    except tce.RpcException, e:
        print e.what()

def call_async():
    # python 方式的异步回调接口
    def hello_callback_async(result,proxy,cookie):
        print 'async call result:',result
```

```
        print 'cookie:',cookie

    prxServer.echo_async('pingpang',hello_callback_async,'cookie')


def call_extras():
    print prxServer.echo("hello",extra={'name':'scott.bo'})

def call_oneway():
    prxServer.heartbeat_oneway('hello world!')

def call_bidirection():
    adapter = tce.RpcCommAdapter('adapter')
    impl = TerminalImpl()
    adapter.addConnection(prxServer.conn)
    adapter.addServant(impl)
    tce.RpcCommunicator.instance().addAdapter(adapter)
    #触发 server 进行反向调用
    prxServer.bidirection_oneway()

def Proxy():
    """
    获得代理
    """
    ep = tce.RpcEndPoint(host='localhost',port=12002)
    prx_server = ServerPrx.create(ep)

    return prx_server


tce.RpcCommunicator.instance().init()  #初始化 tce 库
prxServer = Proxy()              #获取访问代理

call_twoway()   #执行调用
tce.sleep()
```

3.11 多语言支持

为了增加新的程序语言支持，需要对新增语言的了解，包括语法学习和程序语言的运行库的应用（系统功能调用、网络通信接口）。

接着开发对新增语言的编译程序(tmake/tce2xxx.py)，实现IDL的数据类型和接口定义的程序代码翻译，生成Stub代码，包括：客户端的Proxy和服务端的Delegate。

最后实现Rpc运行时设施代码，包括：Communicator,Adapter,Servant,Connection等必要的功能类。

3.12 多平台支持

当对RPC有了充分的了解和掌握之后，TCE作为轻量级的RPC实现，利用其思想和技术手段可以很轻松的移植Rpc到不同的程序语言和系统平台。

Stub代码通过IDL编译程序生成，其主要关注在程序语言的编译实现，而真正平台相关的是Rpc运行库(RunTime)的实现差异。这种差异与具体的平台系统运行机制和提供的系统功能密切相关，这涉及：内存分配和回收、网络通信、异步与同步控制、IPC通信、文件系统接口、安全接口、ＵＩ处理等相关内容。例如：android的Handler与Ios的GCD的block机制，两者同样提供异步请求处理的方式，但在两个平台上实现差异很大。

TCE为构建平台Rpc提供了框架，利用Communicator,Adapter等相关概念设施可以为新平台系统Rpc实现提供参考。

结 束 语

通过本课题的学习和研究，掌握了RPC运行原理和结构。在实际的工作中，涉及到系统与系统之间的交互时，往往都是重复编写通信代码，这种方式效率低而且容易犯错，所以避免重复造轮子的过程最好的办法就是提取工作中共性的东西，将软件开发进行模块化分析和设计。

设计开发一款自主的Rpc框架是我个人一直想完成的一个心愿，虽然目前仅仅是个实验性质的软件产品，但我想这会是个很好的开始，以后我会结合当下软件开发技术趋势和特点逐步完善Rpc服务框架。

本文在一、二章节主要讨论了Rpc的理论、构成内容和移动互联网的应用场景，从第三章节开始描述具体实现一套Rpc框架的方法和技术要点，包括接口语言的设计、接口编译、Rpc运行时框架结构设计，并详细阐述了网络通信细节、Rpc请求调用模式等。

TCE是此次毕业设计的软件项目，其已经具备作为一个基础Rpc框架的能力，已实现多程序语言在移动平台上的Rpc应用，这些语言包括 android平台的Java，ios平台的ObjC和服务平台的Python。 本文阐述的技术实现手法均取自与TCE项目工程。

TCE代码托管地址：https://github.com/adoggie/TCE