# 动态模板驱动的高性能 SOAP 处理

花 磊 魏 峻 牛春雷 郑浩然

(中国科学院软件研究所软件工程技术中心 北京 100080)

**摘 要** 通过实验分析了 Java 平台上 SOAP 处理的性能,发现 XML 数据与 Java 数据间的数据模型映射是影响 SOAP 处理、决定 Web 服务性能的关键因素.对此,提出了一种新的数据模型映射机制——动态提前绑定,通过在运行时动态产生驱动数据模型映射的模板来避免耗时的 Java 反射操作,并实现 XML 数据对象和 Java 数据对象间的快速映射.动态产生的数据映射模板由上下文无关文法定义,并通过带输出的下推自动机实现.将此技术应用于自主开发的高性能 SOAP 引擎——SOAPExpress,实验表明 SOAPExpress 的平均性能比 Apache Axis 1.2 提高 100%以上.

**关键词** 面向服务体系结构;Web 服务;性能;SOAP;数据模型映射;数据映射模板
**中图法分类号** TP311

# High Performance SOAP Processing Based on Dynamic Template-Driven Mechanism

HUA Lei WEI Jun NIU Chun-Lei ZHENG Hao-Ran

(*Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing* 100080)

**Abstract** In this paper, the authors first analyze the performance of SOAP processing on Java platform, and identify that data model mapping between XML data and Java data is the main impact factor on performance. Therefore, the authors propose a new paradigm of data model mapping—"Dynamic Early Binding" which enables to improve SOAP processing by avoiding Java reflection operations and proactively generating processing codes. This dynamic early binding is realized by Data Mapping Template (DMT), which is specified by extended context free grammar and implemented by pushdown automaton with output. The DMTs are generated and compiled at runtime and drive the data mapping process with XML Pull Parsing. The authors illustrate the effectiveness by applying it into a high performance SOAP engine, SOAPExpress, and yielding over 100% speedups compared to Apache Axis 1.2 in Sun Microsystems' benchmark—WS Test 1.0.

**Keywords** SOA; Web services; performance; SOAP; data model mapping; DMT

## 1 引 言

近年来,面向服务的体系结构 SOA (Service-Oriented Architecture)受到学术界和工业界的广泛关注.和面向对象的体系架构不同,SOA 将应用程序的不同功能单元抽象为服务,服务之间通过定义良好的接口和契约联系起来,从而有效地实现服务之间的松耦合.作为一种先进的理念和体系架构,SOA 最早由 Gartner 公司于 1996 年提出,早期人们曾用 CORBA 来构建 SOA 体系架构,但 SOA 的广泛采用却是由于近年来 Web 服务的兴起和普及.

收稿日期:2006-02-17;修改稿收到日期:2006-04-19.本课题得到国家自然科学基金(60573126)和国家"九七三"重点基础研究发展规划项目基金(2002CB312005)资助.花 磊,男,1981 年生,硕士,主要研究方向为 Web 服务和分布式计算.E-mail:hualei@otcaix.iscas.ac.cn.魏 峻,男,1970 年生,博士,研究员,CCF 高级会员,主要研究领域为面向服务计算、中间件、移动计算和软件工程.牛春雷,男,1981 年生,硕士,主要研究方向为网络分布计算和软件工程.郑浩然,男,1981 年生,硕士,主要研究方向为 SOA 和 Web 服务.

Web 服务提供了一套基于 XML① 的服务传输、描述、查找机制，包括简单对象传输协议 SOAP②、Web 服务描述语言 WSDL③、Web 服务的发现和集成 UDDI④ 等，具有平台无关、互操作性强的特点，很好地解决了异构平台环境的互操作问题. Web 服务的特点符合 SOA 体系架构的理念，目前作为 SOA 体系架构的主要实现方式被广泛采用.

目前，随着 SOA 体系架构在商业计算领域中的广泛应用，人们对其性能包括响应时间、传输速度等提出了更高的要求. 但作为 SOA 的主流实现方式，Web 服务与其它分布式计算模型如 EJB、CORBA 和 DCOM 等相比，在性能上有相当的差距，这影响了它在高性能计算环境中的应用. 因此，高性能 Web 服务正日益成为面向服务体系架构研究中的一个重要问题.

有研究[1]指出，分布式系统的性能和消息传输的格式密切相关. 传统的高性能通信系统采用的 MPI 和客户机-服务器通信系统采用的 RPC 提供了高性能的通信机制，但要求通信双方遵循特定的通信协议，这导致了很高的系统耦合度；分布式组件模型的消息格式(如 Java RMI 采用的序列化对象)降低了系统的耦合度，但带来了序列化和反序列化对象的开销；Web 服务采用 XML 作为消息传输的格式，实现了异构平台之间的松耦合，但基于 ASCII 文本的 XML 消息解析、XML 数据对象和平台相关数据对象之间的映射都降低了 Web 服务的性能.

简单对象传输协议 SOAP 协议是 Web 服务的核心协议之一. 它定义了一种独立于特定编程语言和平台的可扩展消息处理框架，是 Web 服务松耦合、高互操作性的基础. SOAP 引擎是 Web 服务的核心功能组件，主要功能是进行 SOAP 消息的处理，包括基于 XML 的 SOAP 消息解析、XML 数据和平台相关对象数据之间的数据模型映射等. SOAP 消息处理的性能直接决定了 Web 服务的性能.

本文通过目前广泛使用的开源 SOAP 引擎 Apache Axis 1.2⑤，分析 Java 平台上 SOAP 消息处理的流程并确定其性能瓶颈，实验证实 XML 数据和 Java 数据间的数据模型映射是影响 SOAP 消息性能的最主要因素.

基于实验结果，我们提出了一种称为“动态提前绑定”(dynamic early binding)的新型数据模型映射机制. 这种数据映射方式在运行时动态生成的模板中保存数据模型的映射信息和映射动作，并由动态模板驱动实现数据模型间的快速映射. 这种数据映射模板由上下文无关文法(CFG)刻画，并通过带输出的下推自动机(pushdown automaton)实现. 我们将动态模板驱动的数据绑定技术应用于自主开发的高性能 SOAP 引擎——SOAPExpress 中，其 SOAP 消息处理性能和 Apache Axis 1.2 相比有大幅提高.

本文第 2 节介绍 SOAP 协议性能的相关研究工作；第 3 节通过实验分析 SOAP 消息处理的性能瓶颈，介绍包括 XML 解析技术在内的 SOAP 处理相关技术，并提出动态提前绑定技术；第 4 节将详细介绍动态提前绑定的实现机制——数据映射模板 Data Mapping Template(DMT)；第 5 节将介绍 DMT 在 SOAPExpress 中的实现，并通过实验验证基于 DMT 的动态提前绑定机制将大幅提高 SOAP 消息处理性能；第 6 节给出结论并介绍未来的工作.

## 2 相关工作

国内外有很多针对 SOAP 协议性能的研究工作[2~6]，这些研究都认为基于 XML 的 SOAP 协议与基于二进制的协议相比在性能上有较大差距.

Davis 通过实验评估并分析了不同 SOAP 引擎的性能，并且和其它协议(如 Java RMI 和 CORBA/IIOP)进行了对比分析[2]. 分析结果表明，导致 SOAP 协议低效的因素有两个：(1)XML 消息解析和格式化时间；(2)Web 服务的网络通信协议(如 HTTP 带来的性能消耗). 文献[3]在与 CORBA 的比较中也给出相似的结论. Chiu 等指出，在科学计算中，SOAP 协议的最大性能瓶颈是 ASCII 字符与浮点数之间的映射[4]；而 Kohlhoff 指出，优化 SOAP 编码和解码将提高 Web 服务在商业应用中的性能[5]. 文献[2~5]都认为除了 XML 解析，XML 数据和实际应用数据间的映射是影响 SOAP 性能的关键因素，Ng 等通过对商用 SOAP 引擎进行基准测试验证了这个结论[6].

XML 数据和应用数据间的数据映射也被称为序列化和反序列化. 它们将极大地影响 SOAP 处理的性能. 最近的研究中，Abu-Ghazaleh 和 Suzumura

① Extensible Markup Language 1.0, 2004. http://www.w3.org/TR/2004/REC-xml-20040204
② Simple Object Access Protocol 1.1, 2003. http://www.w3.org/TR/SOAP/
③ Web Services Description Language, 2001. http://www.w3.org/TR/wsdl
④ Universal Description, Discovery and Integration, 2002. http://uddi.org/
⑤ The Apache Software Foundation, Apache Axis 1.2. http://ws.apache.org/axis/

分别通过优化序列化[7]和反序列化[8]的性能来提高 Java 平台上 SOAP 消息处理的总体性能. 文献[7]将一个序列化的 XML 消息拷贝保存在 SOAP 消息的发送端，将此作为下一个相同类型 SOAP 消息的模板，这将节省从头开始序列化 SOAP 消息的开销. 文献[8]采用的方法是重用已经反序列化的应用数据对象，只对 SOAP 消息中每次新增加或改变的区域进行反序列化操作. 但是，对于大型 SOAP 消息，特别是应用数据频繁变化的 SOAP 消息，文献[8]的性能提升并不明显. 同时，文献[8]使用 Java 反射技术对应用数据对象进行赋值操作，这对大型应用数据对象尤其是深嵌套的对象，将增加反序列化过程的性能损耗.

## 3　Web 服务性能分析和相关技术

本节将分析 Web 服务的服务器端 SOAP 消息的处理流程，并找出 SOAP 消息处理的性能瓶颈. 针对性能瓶颈，我们将介绍与 SOAP 消息处理相关的若干基本技术，并提出动态提前绑定技术.

### 3.1　Web 服务性能分析

我们采用目前广泛使用的 SOAP 引擎 Apache Axis 1.2 作为分析对象. Axis 作为一个 Web 应用置于 Web 容器中，因此 Web 容器承担了通过 HTTP 协议接收和发送 SOAP 消息的工作. 尽管 HTTP 协议本身可能是 Web 服务的性能瓶颈之一[2]，本文将不考虑 HTTP 协议对 Web 服务性能的影响.

一个完整的请求-应答式的 Web 服务调用可分解为如图 1 所示 5 个阶段.

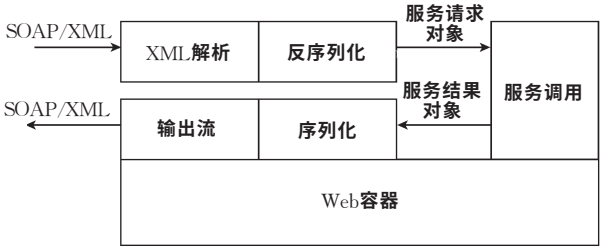



图 1　SOAP 消息处理流程图

(1) XML 解析阶段. SOAP 引擎将解析基于 XML 的 SOAP 消息. Axis 1.2 采用 SAX 解析器解析 XML 消息，XML 消息被解析为一系列 SAX 事件.

(2)反序列化阶段. 被解析的 XML 数据对象将被反序列化为 Java 数据对象. 在此阶段，Axis 1.2 将回放被记录的 SAX 事件，并通知 SAX 事件对应的反序列化器进行反序列化工作.

(3)服务调用阶段. 在得到服务调用数据后，Web 服务的具体实现将被调用. 调用所需时间和服务具体实现的复杂程度相关.

(4)序列化阶段. 以 Java 数据对象形式返回的服务结果将在此阶段被序列化为 XML 数据对象. Axis 1.2 将 XML 数据对象写入系统缓存中.

(5)输出流阶段. 缓存的 XML 数据对象将被写入输出流. Axis 1.2 将 XML 数据对象写入 HTTP 输出流，通过 HTTP 连接返回服务调用端.

我们采用 Sun 公司的 WS Test 1.0① 测试 SOAP 消息处理的各阶段所消耗的时间. WS Test 包括多个测试方法. 测试 SOAP 引擎处理不同类型 SOAP 消息时的性能. 这些方法包括：

(1)echoVoid. 接收并发送 SOAP 消息体为空的 SOAP 消息.

(2)echoStruct. 接收并发送长度为 20 的一个结构体数组. 每个结构体由一个整型数、浮点数和字符串组成.

(3)echoList. 接收并发送一个长度为 20 的结构体链表. 每个链表元素是由一个整型数、浮点数和字符串组成的结构体.

下面给出实验环境的设置.

CPU：Pentinm-4.1 2.80GHz；内存：512 MB；操作系统：Windows XP Professional SP2；JVM：J2SK 1.4.2；SOAP 引擎：Apache Axis 1.2；Web 容器：Tomcat 5.0；XML 解析器：Apache Xerces-J 2.6.2.

SOAP 引擎客户端对每个 Web 服务发送 10000 请求，每秒将发送 5 个请求消息.

图 2(a)所示为 XML 载荷②为 4KB 时 WS Test 的 3 个测试用例在 SOAP 消息处理各阶段的平均时间. 从实验数据中可以发现，在 SOAP 消息处理过程中，XML 解析、反序列化和序列化是 3 个最耗时的处理阶段.

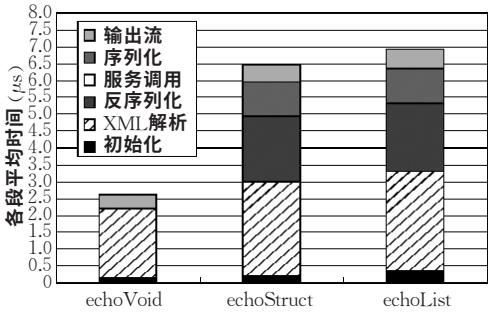
表 1 所示为 XML 解析、反序列化和序列化在 SOAP 消息处理过程中所占比例，3 个阶段所用时间占 SOAP 消息处理总时间的 90%左右，序列化和反序列化所占时间在 50%以上.

表　1

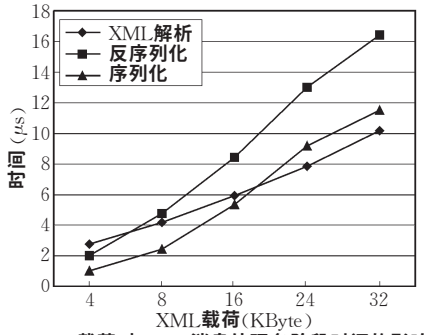
| 测试用例 | 时间比例(%) | | |
|---|---|---|---|
| | XML 解析 | 反序列化 | 序列化 |
| echoVoid | 80 | 0 | 0 |
| echoStruct | 39 | 33 | 19 |
| echoList | 38 | 32 | 18 |

① WS Test 1.0, Sun Microsystems, http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf

② 将被映射为平台相关数据对象的 XML 数据大小.



(a) SOAP 消息处理各阶段平均时间



(b) XML载荷对SOAP消息处理各阶段时间的影响

图 2

我们通过 echoList 方法测试 XML 载荷增加对 SOAP 消息处理的性能影响. 图 2(b)所示为 XML 解析、反序列化和序列化时间随 XML 载荷的变化. 可以看到,当 XML 载荷增加后,3 个阶段的处理时间都相应增大,但反序列化和序列化阶段的时间增长幅度明显大于 XML 解析阶段. 由图中数据可知,当 XML 载荷大于 8KB 时,反序列化阶段时间占 SOAP 处理总时间的比例在 3 个阶段中最大,并随着 XML 载荷的增加进一步增长;当 XML 载荷大于 16KB,序列化阶段所占时间比例也超过 XML 解析阶段.

总之,XML 解析、反序列化和序列化是 SOAP 处理的性能瓶颈,当处理的 XML 载荷较小时,XML 解析是影响 SOAP 消息处理性能的主要因素;当 XML 载荷中等或较大的,序列化和反序列化是影响性能的关键.

Apache Axis 1.2 在反序列化和序列化阶段采用 Java 反射技术实现 XML 数据对象和 Java 数据对象的映射. Java 反射技术可以在运行时得到 Java 类的属性、方法等信息,改变 Java 对象的属性和执行 Java 对象的方法,这给数据对象间的转换带来很大的灵活性,但这种技术会导致较大的性能开销. 我们通过实验发现,对于复杂 Java 对象特别是深嵌套 Java 对象的序列化和反序列化过程,多数时间被消耗在 Java 反射操作上. 因此,在数据模型的映射中减少或避免 Java 反射的使用将有效地提高 SOAP 消息处理性能.

### 3.2 基于拉模式的 XML 解析

XML 消息解析是 SOAP 消息处理的重要组成部分,从 3.1 节可以了解到 XML 消息解析是耗时的处理阶段之一,因此高性能的 XML 消息解析将提高 SOAP 消息处理性能.

目前,最常用的 XML 处理接口是 DOM① 和 SAX[9]. DOM 接口在处理 XML 文件时,将整个 XML 文件读入内存,并在内存中建立 XML 文件的对象模型,这种处理方式的缺点是在处理大型 XML 文件时将占用大量内存. SAX 接口解析 XML 文件时产生一系列的 SAX 事件,应用程序通过编写 SAX 事件的回调函数来处理 XML 文件. 和 DOM 接口相比,SAX 接口不需要将整个 XML 文件读入内存,这将减少内存的占用;但通过编写回调函数进行 XML 文件处理的方式增加了实现的复杂度.

而如图 3 所示,拉模式的 XML 解析是一种应用程序驱动的 XML 解析机制,由应用程序驱动拉模式的 XML 解析器(XML Pull Parser)进行 XML 文件解析,解析将直接获得 XML 元素对应的各种事件. 基于拉模式的 XML 解析不需要将像 DOM 将 XML 文件一次读入内存,这将降低 XML 文件解析的内存开销;同时应用程序不必像 SAX 编写 XML事件对应的回调函数. 这种应用程序驱动的 XML 解析方式给 SOAP 消息处理带来了很大的灵活性,也是动态模板驱动的数据模型映射技术的基础.

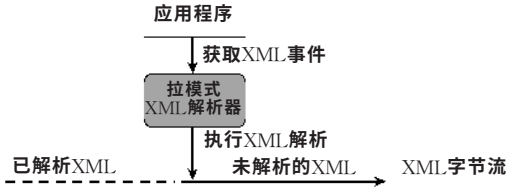


图 3 拉模式的 XML 解析原理图

### 3.3 基于动态代码生成技术的提前绑定

Web 服务调用过程中,SOAP 消息处理的核心操作是 XML 数据和平台相关数据间的映射,SOAP 消息处理流程中的数据模型映射包括 XML 数据解析、序列化和反序列化. 一般来说,Web 服务中数据模型映射的元素包括以 XML Schema 定义的 XML 类型、特定平台的数据类型以及两者的映射规则. 本

① Document Object Model, http://www.w3.org/DOM/

文关注 Java 平台上 XML 数据和 Java 对象间的映射，所需的元素是 Java 语言定义的平台相关数据类型或以 XML Schema 定义的 XML 类型、Java 类型与 XML 类型之间的映射规则.

3.1 节指出 Java 数据模型和 XML 数据模型间的映射是 Java 平台上 SOAP 引擎的性能瓶颈，我们将首先介绍 Java 平台上数据模型映射的常用技术——提前绑定和延迟绑定，并给出一种新的数据模型映射技术——“动态提前绑定”. 动态提前绑定技术集中了提前绑定技术和延迟绑定技术的优点. 本节中我们将使用数据绑定和数据模型映射这两个等价概念.

SOAP 消息是由 SOAP 信封包含的 SOAP 体和 SOAP 头组成，其中，SOAP 头主要描述 Web 服务的安全性、可靠性等 QoS 信息；而 SOAP 体描述了 Web 服务的具体业务逻辑，包括 Web 服务对应具体实现的方法名、方法调用参数或返回结果. 图 4 所示为一个 Web 服务的 Java 实现和对应的 SOAP 请求消息，SOAP 体中包含了 Web 服务具体实现的方法名 echoStruct 和方法调用参数 foo，阴影部分表示 Java 数据模型和 XML 数据模型之间的映射.

```
〈SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"〉

 〈SOAP-ENV:Body〉
   〈echoStruct xmlns="http://www.sun.com/wstest/wsdl/"〉
     〈foo〉
       〈varInt〉1〈/varInt〉
       〈varFloat〉1.0〈/varFloat〉
       〈varString〉ss〈/varString〉
     〈/foo〉
   〈/echoStruct〉
 〈/SOAP-ENV:Body〉

〈/SOAP-ENV:Envelope〉
```

```
public class echoServerImpl{

   public Struct echoStruct(Struct foo)
     {    return foo;
     }
   }

public class Struct {
   public int varInt;
   public float varFloat;
   public java.lang.String varString;

   public Struct(){
   }

}
```

图 4　SOAP 消息和 Java 类型定义关系图

首先我们解释两对概念，分别是延迟绑定和提前绑定，动态绑定和静态绑定.

(1)延迟绑定和提前绑定

图 5 所示为提前绑定和延迟绑定两种机制的对比图. 两种绑定机制的区别是获得绑定信息和执行绑定的时机，这里的绑定信息指 XML 数据和 Java 数据的映射关系. 提前绑定在绑定进行前获得全部绑定信息，而后进行数据绑定；延迟绑定在绑定进行的同时获得绑定信息，绑定消息的获得和绑定的执行是交叉进行的.

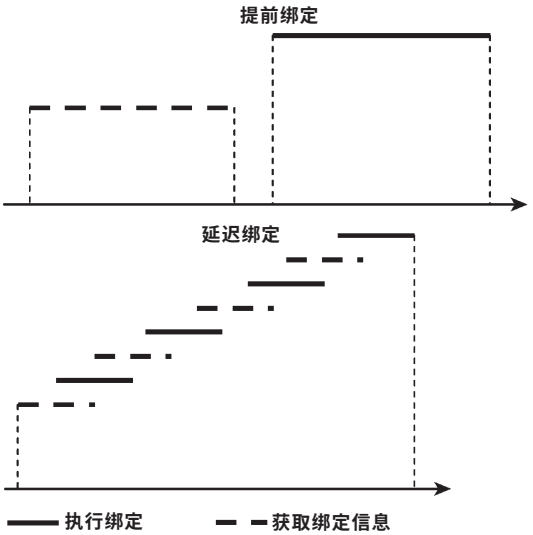



图 5　提前绑定和延迟绑定

(2)动态绑定和静态绑定

动态绑定是指在运行时动态添加新的 XML-Java 映射类型对的绑定机制，而静态绑定是指在编译时静态添加新的映射类型对的绑定机制.

根据我们的定义，Java 平台上现有的数据绑定实现可以分成两类：动态延迟绑定和静态提前绑定.

(1)动态延迟绑定在绑定进行时通过 Java 反射操作获得绑定信息，然后使用绑定信息进行 XML 数据和 Java 数据间的映射，其代表实现有 Apache Axis 1.2；同时，动态绑定技术可以在运行时实现数据模型映射并动态添加新的 XML-Java 映射对，这将增加绑定实现的灵活性，但 Java 反射的频繁使用将带来性能的损耗.

(2)静态提前绑定在绑定进行前通过代码生成技术生成记录绑定信息的模板，并借助模板进行数据模型映射. 静态提前绑定技术通过编译时获得绑定信息来避免 Java 反射的使用，这提高了数据模型映射的性能，但由于静态提前绑定只能在运行前生成模板，无法在运行时添加新的映射对，其灵活性和

动态延迟绑定相比较差.

图 6 所示是动态提前绑定(dynamic early binding)的原理图. 动态提前绑定在编译时确立 XML Schema 和 Java 类之间的映射规则,在运行时根据 XML Schema、Java 类和两者的映射规则,通过动态代码生成技术生成记录映射信息和映射操作的模板 DMT,并由数据映射模板驱动数据模型映射. 如果 XML Schema 不存在,将首先通过 Java 类型和映射规则生成 XML Schema,然后再通过动态代码生成技术生成数据映射模板.

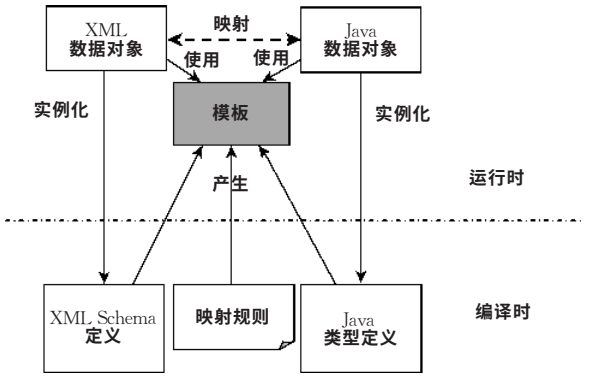



图 6 动态提前绑定原理图

表 2 所示为三种数据绑定技术的对比. 动态提前绑定技术通过动态记录的映射信息驱动数据模型映射,避免在运行时依赖 Java 反射技术获得绑定信息,这将显著提高数据模型转换的性能;同时,数据映射模板在运行时动态产生, 这使动态提前绑定技术具有像延迟绑定一样的灵活性. 动态提前绑定技术综合了延迟绑定和提前绑定的优点,是实现高性能 SOAP 处理的关键技术.

表 2 绑定技术对比表

| 特性 | 绑定机制 | | |
|---|---|---|---|
| | 动态延迟绑定 | 静态提前绑定 | 动态提前绑定 |
| 关键技术 | Java 反射 | 代码生成 | Java 反射+动态代码生成 |
| 添加新类型 | 运行时 | 运行前 | 运行时 |
| 获取绑定信息 | 绑定执行时 | 绑定执行前 | 绑定执行前 |
| 灵活性 | 强 | 弱 | 强 |
| 性能 | 低 | 高 | 高 |
| 代表实现 | Apache Axis 1.2 | XMLBeans | 数据映射模板 DMT |

## 4 动态模板驱动的数据模型映射

上文分析了动态提前绑定技术的特点,指出这是提高 SOAP 引擎性能的一种关键技术. 动态提前绑定技术的核心是数据映射模板 DMT,本节将给出动态提前绑定技术的具体实现——数据映射模板 DMT 的定义、概念模型和实现模型.

### 4.1 基于上下文无关文法的数据映射模板刻画

图 7 描述了 XML Schema 定义的 XML 数据类型和 Java 语言定义的 Java 数据类型. 这里我们采用扩展的上下无关文法来刻画 XML 数据模型和 Java 数据模型,并通过文法产生式对应的映射方案(mapping scheme)描述两种数据模型的映射关系.

```
〈xsd:schema
xsd:xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="person"xmlns:typens="person"〉
  〈xsd:complexType name="Person"〉
    〈xsd:sequence〉
      〈xsd:element name="name" type="xsd:string"/〉
      〈xsd:element name="age" type="xsd:int"/〉
      〈xsd:element name="addr" type="typens:Address"/〉
    〈/xsd:sequence〉
  〈/xsd:complexType〉
〈/xsd:schema〉
```

```
public class Person{
   public String name;
   public int age;
   public String addr;
}
```

图 7 XML Schema 定义和 Java 定义对比图

**定义 1.** 数据映射模板(Data Mapping Template,DMT). 对于任意数据类型 $T$,其数据映射模板 $DMT=(G^X,G^J)$,$G^X$ 和 $G^J$ 分别是对应 XML 数据模型和 Java 数据模型的两个上下文无关文法.

**定义 2.** 模板的上下文无关文法 $G=(V,T,P,S,M)$,其中:

$V$ 是非终结符的非空有限集,$\forall A\in V$,$A$ 叫做非终结符. 对于 $G^X$,$V$ 是数据类型 $T$ 的 XML Schema 定义中数据类型的集合,包括简单类型、复杂类型和数组类型等;对于 $G^J$,$V$ 是数据类型 $T$ 的 Java 定义中数据类型的集合,包括 Java 语言中原始类型、用户自定义 Java 类和数组类型等;

$T$ 是终结符的非空有限集,$\forall\alpha\in T$,$\alpha$ 叫做终结符. 对于 $G^X$,$T$ 是 XML Schema 中标签名称的集合;对 $G^J$,$T$ 是 Java 类中属性名称的集合;

$P$ 是产生式的非空有限集合,每个产生式 $p$ 具

有形式 $A\rightarrow\alpha, A\in V, |A|\leq|\alpha|$；

$S$ 是非终结符，称为开始符号，它定义的串集就是该文法定义的数据模型；

$M$ 是映射方案(mapping scheme)集合. $\forall m\in M, m$ 为某个产生式 $p$ 对应的映射方案，它定义了产生式在规约过程中对应的系列动作，$m$ 是由一组原子操作组成. $m$ 和 $p$ 是一一对应关系，$\forall m\in M, \exists p\in P, p\leftrightarrow m$，反之也成立.

文法 $G$ 的定义中提到，每个文法产生式对应一个或一组原子操作，表 3 列出了 XML 数据模型和 Java 数据模型的原子操作集合. XML 数据通常组织为树状结构，树状结构的 XML 数据的节点称为 XML 元素，包括元素名称和元素值，由 XML 解析器解析得到. XML 数据模型的原子操作包括 XML 元素的创建、XML 子元素的取值和赋值、元素值的取值和赋值等.

Java 是一种面向对象的语言，其数据类型包括原始类型、数组类型和用户自定义类. 原始类型包括 Java 语言支持的原始数据类型(如 int)和原始数据类型的包装类(如 Integer). Java 语言中的类用来定义用户自定义类型，其公有属性值可以直接得到，私有属性值由 Java 类中定义的访问方法(如 get 和 set)得到；数组类型可以通过指定的下标进行赋值和取值. Java 数据模型的原子操作包括不同 Java 类型的创建和取值操作、用户定义 Java 类的属性赋值和取值操作、Java 数组类型的赋值和取值操作等.

**表 3　XML 数据模型和 Java 数据模型原子操作集合**

| XML 数据模型的原子操作集合 | Java 数据模型的原子操作集合 |
|---|---|
| creatElement(eleName) | creatJavaType(typeName) |
| getElementValue(eleName) | setSimpleValue() |
| setElementValue(eleValue) | setFieldValue(fieldName, fieldValue) |
| addChildElement(ele) | getFieldName() |
| getNextChildElement() | getFieldValue(fieldName) |
| returnElement() | setIndexValue(index, indexValue) |
| | getIndexValue(index) |
| | returnTypeValue() |

对于动态延迟绑定，Java 数据模型的原子操作在运行时通过 Java 反射机制实现. 本文介绍的动态提前绑定，每个 Java 数据类型均有特定的数据映射模板 DMT，因此，Java 数据模型的原子操作可以通过在动态生成的 DMT 中调用 Java 数据类型自身的方法实现，从而避免了 Java 反射的使用，并有效地提高了 SOAP 消息处理的性能.

表 4 分别包括 XML 数据模型和 Java 数据模型的文法 $G$ 和映射方案 $M$.

**表 4　$G^X$，$G^J$ 的映射方案**

| $G^X$ 的映射方案 $M^X$ | $G^J$ 的映射方案 $M^J$ |
|---|---|
| $S\rightarrow tag\{T^X.ele=S.ele\}T^X tag'\ \{S.value=T^X.value\}$ | $S\rightarrow\{S.creatElement(), T^J.value=S.value, T^J.ele=S.ele\}T^J$ |
| $T^X\rightarrow\{T_0^{XC}.value=createJavaType(), T_0^X.ele=T^{XC}.ele\}T_0^{XC}\{T^X.value=T_0^{XC}.value\}$ | $T^J\rightarrow\{T_0^{JC}.ele=T^J.ele, T_0^{JC}.value=T^J.value\}T_0^{JC}$ |
| $T_i^{XC}\rightarrow tag\{T^{X'}.ele=T_i^{XC}.getNextChildElement()\}T^{X'}\{T_i^{XC}.setFieldValue(T^{X'}.value)\}$ $tag'\{T_{i+1}^{XC}.value=T_i^{XC}.value, T_{i+1}^{XC}.ele=T_i^{XC}.ele\}T_{i+1}^{XC}$ | $T_i^{JC}\rightarrow field\{T^{J'}.value=T_i^{JC}.getFieldValue(field)\}, T^{J'}.creatElement()\}T^{J'}\{T_i^{JC}.addChildElement(T^{J'}.ele), T_{i+1}^{JC}.value=T_i^{JC}.value, T_{i+1}^{JC}.ele=T_i^{JC}.ele\}T_{i+1}^{JC}$ |
| $T_{i+1}^{XC}\rightarrow\varepsilon, i=0,1,2,\cdots$ | $T_{i+1}^{JC}\rightarrow\varepsilon, i=0,1,2,\cdots$ |
| $T^X\rightarrow\{T^{XS}.value=createJavaType(), T^{XS}.ele=T^X.ele\}T^{XS}\{T^X.value=T^{XS}.value\}$ | $T^J\rightarrow\{T^{JS}.ele=T^J.ele, T^{JS}.value=T^J.value\}T^{JS}$ |
| $T^X\rightarrow\{T^{XA}.value=createJavaType(), T^{XA}.ele=T^X.ele\}T^{XA}\{T^X.value=T^{XA}.value\}$ | $T^{JS}\rightarrow\{T^{JS}.setElementValue(T^{JS}.value)\}\varepsilon$ |
| $T^{XS}\rightarrow\varepsilon\{T^{XS}.setSimpleValue(T^{XS}.getElementValue()\}$ | $T^J\rightarrow\{T^{JA}.ele=T^J.ele, T^{JA}.value=T^J.value\}T^{JA}$ |
| $T^{XA}\rightarrow tag\{T^X.ele=T^{XA}.getNextChildElement()\}T^X\{T^{XA}.setIndexValue(T^X.value)\}tag'T^{XA}$ | $T^{JA}\rightarrow\{T^J.creatElement(), T^J.value=T^{JA}.getIndexValue()\}T^J\{T^{JA}.addChildElement(T^J.ele)\}T^{JA}$ |
| $T^{XA}\rightarrow\varepsilon$ | $T^{JA}\rightarrow\varepsilon$ |

文法 $G^X$ 描述以 XML Schema 定义的 XML 数据模型. 文法产生式 $P^X$ 中，$S$ 是开始符号，$tag$ 标示 XML 元素的开始标签名，$tag'$ 标示 XML 元素的结束标签名；$T^X$ 标示 XML 数据类型，分为 XML Schema 内置的简单类型 $T^{XS}$、数组类型 $T^{XA}$ 和复杂类型 $T^{XC}$. $G^X$ 将 XML 数据模型定义为由 XML 标签名称组成的树状结构. 其中，$T^X$ 包括两个属性：$T^X.ele$ 和 $T^X.value$. $T^X.ele$ 代表 XML 数据，$T^X.value$ 代表将被映射成的 Java 数据.

文法 $G^J$ 描述以 Java 语言定义的 Java 数据模型. 文法产生式 $P^J$ 中，$T^J$ 标示 Java 数据类型，分为原始类型 $T^{JS}$、数组类型 $T^{JA}$ 和 Java 类 $T^{JC}$；$field$ 是 Java 类中的属性名称. $T^J$ 包括两个属性：$T^J.value$ 和 $T^J.ele$. $T^J.value$ 代表 Java 数据值，$T^J.ele$ 代表将被映射成的 XML 数据. 从 $G^J$ 的角度来看，Java 数据类型可以看作结构化的树状结构，以和 XML 数据类似的方式描述.

数据模型映射的元素包括 XML 数据模型、Java数据模型和两者的映射规则. 其中，XML 数据模型和 Java 数据模型分别由 $G^X$ 和 $G^J$ 描述，而两者

的映射规则由映射方案 $M$ 描述，表 3 中以{}表示的映射方案是映射规则的实现. JAX-RPC① 定义了 XML Schema 和 Java 数据类型间的一种映射规则.

数据映射模板 DMT 定义了特定类型 $T$ 的数据模型映射方案，包括 Java 数据到 XML 数据的映射和 XML 数据到 Java 数据的映射. 生成数据映射模板时，XML 数据类型或 Java 数据类型、两者的映射规则是不可缺少的，本文将从 Java 数据类型和映射规则生成数据映射模板.

图 8 所示为 DMT 的生成算法框图. 算法的输入是某种 Java 数据类型，输出为 $G^X$、$G^J$ 和对应的映射方案 $M$ 组成的 DMT 实例. 算法的主要分析过程是对 Java 数据类型进行深度优先遍历. 为简单起见，图 8 描述的算法省略了用户自定义 Java 类和数组类型的递归分析逻辑.

对图 7 所示的 XML Schema 和 Java 类型，根据上述算法生成的 DMT 如表 5 所示.

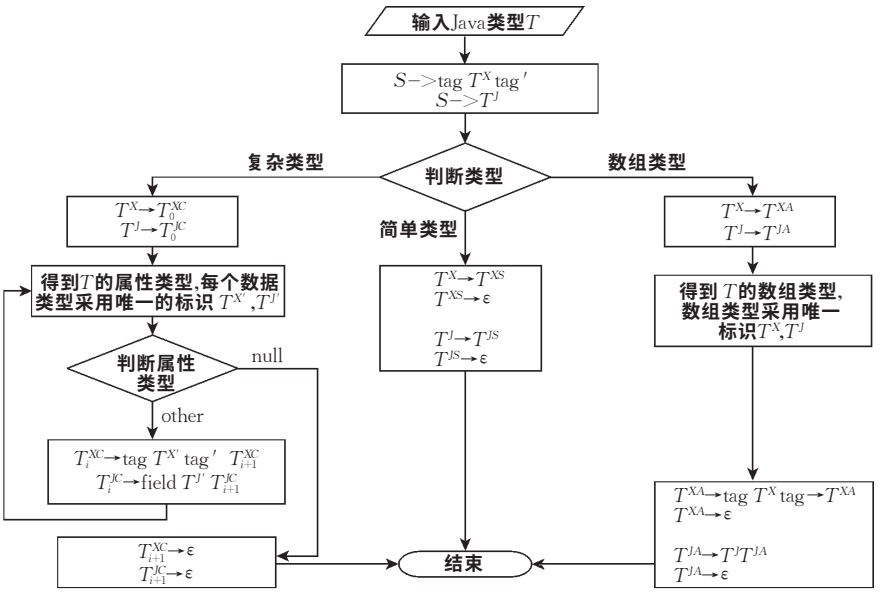



图 8 DMT 的生成算法框图

表 5

| $G^X$ |
|---|
| $S \rightarrow p\{T^x.ele=S.ele\}T^x p'\{S.value=T^x.value\}$ |
| $T^x \rightarrow \{T_0^{xc}.value=new\ Person(), T_0^{xc}.ele=T^x.ele\}T_0^{xc}\{T^x.value=T_0^{xc}.value\}$ |
| $T_0^{xc} \rightarrow name\{T^{xn}.ele=T_0^{xc}.getNextChildElement()\}T^{xn}\{T_0^{xc}.value.name=T^{xn}.value\}name'\{T_1^{xc}.value=T_0^{xc}.value, T_1^{xc}.ele=T_0^{xc}.ele\}T_1^{xc}$ |
| $T^{xn} \rightarrow \{N.ele=T^{xn}.ele, N.value=new\ String()\}N\{T^{xn}.value=N.value\}$ |
| $N \rightarrow \varepsilon\{N.value=N.getElementValue()\}$ |
| $T_1^{xc} \rightarrow age\{T^{xa}.ele=T_1^{xc}.getNextChildElement()\}T^{xa}\{T_1^{xc}.value.age=T^{xa}.value\}age'\{T_2^{xc}.value=T_1^{xc}.value, T_2^{xc}.ele=T_1^{xc}.ele\}T_2^{xc}$ |
| $T^{xa} \rightarrow \{A.ele=T^{xa}.ele, A.value=(int)0\}A\{T^{xa}.value=A.value\}$ |
| $A \rightarrow \varepsilon\{A.value=Integer.parseInt(A.getElementValue())\}$ |
| $T_2^{xc} \rightarrow addr\{T^{xad}.ele=T_2^{xc}.getNextChildElement()\}T^{xad}\{T_2^{xc}.value.addr=T^{xad}.value\}addr'\{T_3^{xc}.value=T_2^{xc}.value, T_3^{xc}.ele=T_2^{xc}.ele\}T_3^{xc}$ |
| $T_3^{xc} \rightarrow \varepsilon$ |
| $T^{xad} \rightarrow \{ADDR.ele=T^{xad}.ele, ADDR.value=new\ String()\}ADDR\ \{T^{xad}.value=ADDR.value\}$ |
| $ADDR \rightarrow \varepsilon\{ADDR.value=ADDR.getElementValue()\}$ |
| $S \rightarrow \{S.creatElement(\text{“}p\text{”}), T^j.value=S.value, T^j.ele=S.ele\}T^j$ |
| $T^j \rightarrow \{T_0^{jc}.ele=T^j.ele, T_0^{jc}.value=T^j.value\}T_0^{jc}$ |
| $T_0^{jc} \rightarrow name\ \{T^{jn}.value=T_0^{jc}.value.name, T^{jn}.creatElement(\text{“}name\text{”})\}T^{jn}\{T_0^{jc}.addChildElement(T^{jn}.ele), T_1^{jc}.value=T_0^{jc}.value, T_1^{jc}.ele=T_0^{jc}.ele\}T_1^{jc}$ |
| $T^{jn} \rightarrow \{N.ele=T^{jn}.ele, N.value=T^{jn}.value\}N$ |
| $N \rightarrow \{N.setElementValue(N.value)\}\varepsilon$ |
| $T_1^{jc} \rightarrow age\{T^{ja}.value=T_1^{jc}.value.name, T^{ja}.creatElement(\text{“}age\text{”})\}T^{ja}\{T_1^{jc}.addChildElement(T^{ja}.ele), T_2^{jc}.value=T_1^{jc}.value, T_2^{jc}.ele=T_1^{jc}.ele\}T_2^{jc}$ |
| $T^{ja} \rightarrow \{A.ele=T^{ja}.ele, A.value=T^{ja}.value\}A$ |

① Java API for XML-Based RPC. http://java.sun.com/webservices/jaxrpc/docs.htm

（续　表）

| $G^X$ |
|---|
| $A\rightarrow\{A.setElementValue(A.value)\}\varepsilon$ |
| $T_2^{jc}\rightarrow address\{T^{jad}.value=T_2^{jc}.value.addr, T^{jad}.creatElement(\text{“}address\text{”})\}T^{jad}\{T_2^{jc}.addChildElement(T^{jad}.ele), T_3^{jc}.value=T_2^{jc}.value, T_3^{jc}.ele=T_2^{jc}.ele\}T_3^{jc}$ |
| $T_3^{jc}\rightarrow\varepsilon$ |
| $T^{jad}\rightarrow\{ADDR.ele=T^{jad}.ele, ADDR.value=T^{jad}.value\}ADDR$ |
| $ADDR\rightarrow\{ADDR.setElementValue(ADDR.value)\}\varepsilon$ |

### 4.2　基于下推自动机实现数据映射模板

上文介绍了基于上下文无关文法刻画的数据映射模板 DMT 的概念模型，本节给出数据映射模板的实现模型. 对于任意数据类型 $T$，数据映射模板 DMT 的实现模型是用于识别 $G^X$ 和 $G^J$ 文法定义的数据模型的一对带输出的下推自动机，我们把这种带输出的下推自动机称为数据映射自动机 DMA (Data Mapping Automaton).

**定义 3.**　数据映射自动机 $DMA=(Q,\Sigma,\Gamma,Z_0,q_0,F,O,\delta)$，其中：

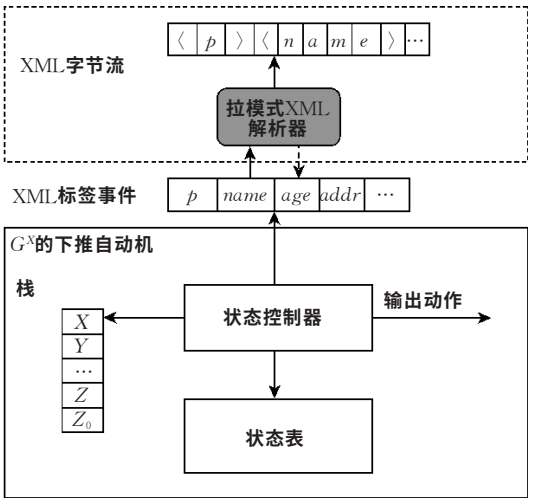
$Q$ 表示状态有限集合；$\Sigma$ 是输入符合的集合；$\Gamma$ 是栈符号表；$Z_0\in\Gamma$ 叫做开始符合，是 $M$ 启动时栈内唯一的一个符号；$q_0\in Q$ 是 $M$ 的开始状态；$F$ 是 $M$ 终止状态的集合；$O$ 是输出动作集合；$\delta$ 是状态转换函数，$\delta(q,a,Z)=\{(p_1,\gamma_1,O_1),(p_2,\gamma_2,O_2),\cdots,(p_m,\gamma_m,O_m)\}$，表示 $M$ 在状态 $q$，栈顶元素为 $Z$ 时，读入字符 $a$，对于 $i=1,2,3,\cdots,m$，可以选择性地将状态变成 $p_i$，将栈顶符号 $Z$ 弹出，将 $\gamma_i$ 中的符号从右到左一次压入栈，同时执行输出动作 $O_i$.

数据映射自动机 $DMA=(Q,\Sigma,\Gamma,Z_0,q_0,F,O,\delta)$ 可以由上下文无关文法 $G=(V,T,P,S,M)$ 产生. 下推自动机和上下文无关文法之间的转换和等价证明参见文献[10]，这里不再赘述.

图 9 所示分别是文法 $G^X$ 和 $G^J$ 对应的数据映射自动机 $DMA^X$ 和 $DMA^J$，它们都由实际输入对象、输入转换器、存放文法符号的栈、状态控制器 SC 和状态表 ST 组成. 其中，输入转换器将实际输入转换为自动机可以识别的输入对象；状态表是二维数组 $P[A,a]$，$A$ 是非终结符，$a$ 是终结符，数组元素 $P[A,a]$ 记录状态转移和输出动作；状态控制器控制状态转移并执行输出动作.

$DMA^X$ 的输入是 XML 字节流，其输入转换器是 XML Pull Parser. $DMA^X$ 运行时，状态控制器驱动 XML Pull Parser 读取 XML 字节流并得到 XML 事件；状态控制器根据获得的 XML 事件、栈顶符号，查找状态表并执行状态转移操作，同时执行构造 Java 对象、Java 对象赋值等映射操作，当自动机 $DMA^X$ 运行结束时，返回构造完成的 Java 对象.

$DMA^J$ 的输入是 Java 对象实例，其输入转换器是 Java Type Reader. Java 数据类型可以被视为结构化的树状结构. 对特定 Java 数据类型，一个虚拟的树状层次结构将被产生并保存在内存中. Java Type Reader 深度遍历特定数据类型的虚拟树状结构，得到 Java 类型事件(Java Type Event)，并将其返回给 $PDA^J$ 的状态控制器. 状态控制器根据获得的 Java 类型事件、栈顶符号，查找状态表并执行状



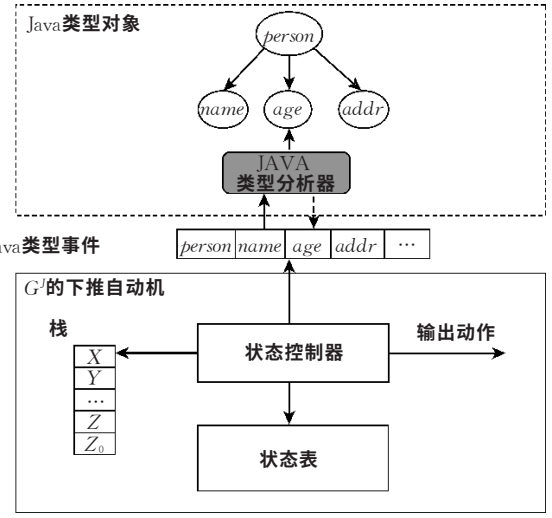




图 9　文法 $G^X$ 和 $G^J$ 对应的数据映射自动机 $DMA^X$ 和 $DMA^J$

态转移操作,同时执行构造 XML 输出流、将 Java 对象的属性值以结构化的形式写入 XML 输出流等映射操作. 自动机 $DMA^J$ 运行结束时,返回构造完成的 XML 输出流.

数据映射模板 DMT 充分利用了 XML Pull Parser 的特点,避免对 XML 数据进行多次遍历,通过 $DMA^X$ 遍历 XML 输入流,同时通过自动机的输出动作完成 Java 对象的构建,在一次遍历中即完成 XML 对象到 Java 对象的数据模型映射;同样,DMT 通过结构化的方式遍历 Java 对象的属性,同时通过自动机的输出动作完成 XML 输出流的构建,通过一次遍历 Java 结构信息完成 Java 对象到 XML 对象的数据模型映射.

这种数据映射模板驱动的映射机制,通过模板记录的映射信息直接进行映射操作,避免了 Java 反射的使用,提高了数据模型映射的性能;同时充分利用了带输出的有限自动机的特点,在遍历数据模型的同时进行映射操作,避免对数据模型进行多次遍历.

## 5 DMT 在 SOAPExpress 中的应用及评价

SOAPExpress 是采用基于拉模式的 XML 解析和动态提前绑定技术的高性能 SOAP 引擎. 动态提前绑定技术的使用,提高了 SOAPExpress 处理 SOAP 消息尤其是数据模型映射的性能;同时,动态提前绑定可以在运行时动态添加新的映射类型,保证了 SOAPExpress 的灵活性. SOAPExpress 中采用的动态提前绑定技术是通过第 4 节介绍的数据映射模板 DMT 来实现的.

### 5.1 DMT 在 SOAPExpress 中的应用

SOAPExpress 是作为 Web 应用置于 Web 容器中,如图 10 所示. 对于 RPC-encoded 和 literal-wrapped 类型的 Web 服务,SOAP 消息体中包含了 Web 服务的操作名. 当 SOAPExpress 接收到 SOAP 请求消息后,服务接受者(Service Receiver)将通过 XML Pull Parser 得到 SOAP 消息对应的操作名,并从 DMT 管理器中得到该操作的调用参数所对应的数据映射模板 DMT,数据映射模板将执行 XML 数据对象到 Java 数据对象的映射操作. 此后,服务调用器(Service Invoker)将执行服务调用,并得到 Java 对象形式的调用结果;服务发送者(Service Sender)得到返回结果后,从 DMT Manager 中得到返回参数对应的数据映射模板,数据映射模板将执行 Java 对象到 XML 数据的映射过程.

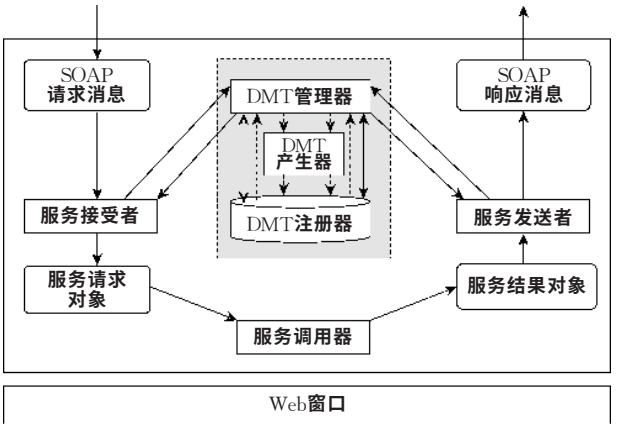



图 10 SOAPExpress 中 SOAP 消息处理流程图

SOAPExpress 利用 DMT 管理器、DMT 生成器和 DMT 注册表等组件来支持 DMT 驱动的数据模型映射. 其中,DMT 管理器提供 DMT 操作接口,包括 DMT 实例的查找、生成、缓存等操作;DMT 生成器负责根据 Java 类型定义生成 DMT 实例;DMT 注册表负责缓存 DMT 实例,并通过 DMT 注册表实现 DMT 实例的快速查找. DMT 管理器查找 DMT 实例时,首先在注册表中直接查找,如果没有找到,则调用 DMT 生成器生成 DMT 实例,并置入 DMT 注册表中.

图 11 所示为 DMT 注册表的结构图,DMT 注册表记录并维护 DMT 实例. DMT 注册表的键值是服务名称,DMT 注册表的值是调用参数或者返回参数对应的 DMT 实例. DMT 注册表的主要功能是维护服务名称到 DMT 模板的映射关系.

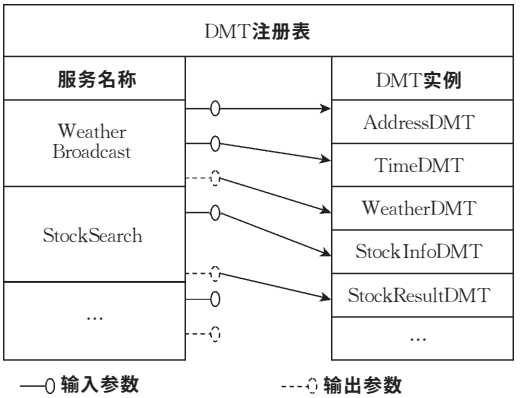



图 11 DMT 注册表结构图

图 12 所示为 DMT 产生器的结构图,描述了通过 Java 类型定义生成 DMT 实例的流程. DMT 产生器主要包括类型分析器、模型创建器和 Java 字节码生成器组成. 类型分析器将使用 Java 反射技术分

析 Java 类型定义的层次结构，并根据图 8 描述的 DMT 生成算法生成 DMT 实例；此后，字节码生成器将使用 Javassist① 生成 DMT 实例对应的 Java 字节码. 对于一个特定的 Java 类型，它的 DMT 实例只在第一次生成，并当此 Java 类型不发生改变时被重用.

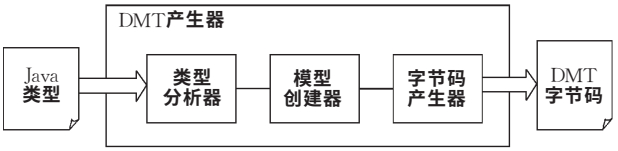



图 12　DMT 产生器结构图

SOAPExpress 利用 DMT 驱动数据模型映射，由此带来的性能开销包括 DMT 的产生和初始化、DMT 的查询等. 对于特定类型的 Java 类，其 DMT 实例只生成并初始化一次；同时，SOAPExpress 通过哈希表保存 DMT 实例，由实验可知，对已经存于哈希表中的 DMT 实例，其平均查询时间小于 0.01μs，因此其对 SOAP 消息处理性能的影响可以忽略.

### 5.2　实验及结果分析

本节将给出 SOAPExpress 处理 SOAP 消息的性能指标测试结果，并将其和 Apache Axis 1.2 作性能比较，本节中 SOAPExpress 的测试环境与 3.1 节中 Apache Axis 1.2 的测试环境相同.

图 13 显示了 Apache Axis 1.2 和 SOAPExpress 针对 WS Test 1.0 中测试用例的性能对比，对 echoStruct 和 echoList 方法，XML 载荷均为 4KB. 实验所测时间是从 Web 容器接受包含 SOAP 消息的 HTTP 请求消息开始，到 Web 容器向客户端发送完成包含 SOAP 消息的 HTTP 响应消息结束. 实验数据表明，因为 echoVoid 方法本身并没有业务逻辑，只是返回 SOAP 体为空的 SOAP 消息，所以两种 SOAP 引擎在 echoVoid 方法的平均处理时间差别不大；对于方法 echoList，SOAPExpress 所用平均时间是 Axis 的 46%，对于方法 echoStruct，SOAPExpress 所用平均时间是 Axis 的 44%. echoList 和 echoStruct 的参数类型分别是深嵌套数据类型和复杂数组类型. 实验结果表明 SOAPExpress 处理 SOAP 消息的性能明显优于 Apache Axis 1.2.

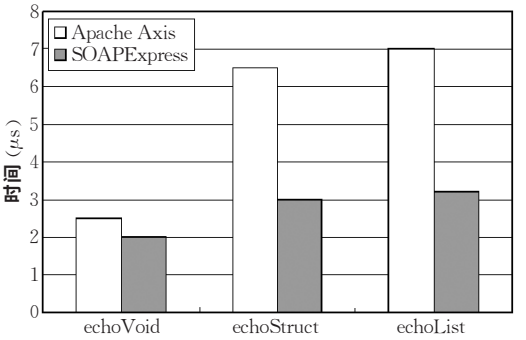



图 13　Apache Axis 1.2 和 SOAPExpress 性能对比

图 14 所示为 XML 负载增加时，SOAPExpress 和 Apache Axis 1.2 的性能比较. 如图所示，当 XML 载荷增大时，SOAPExpress 处理时间的增长速度远小于 Apache Axis 1.2，对于大载荷的 XML 数据尤为明显.

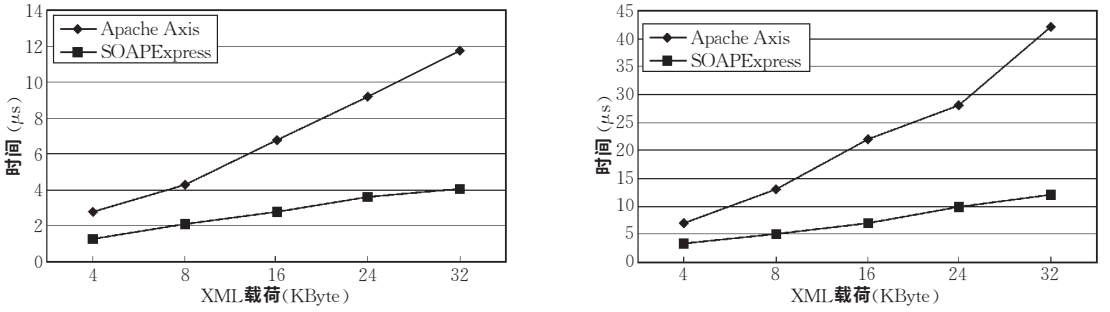



图 14　Apache Axis 1.2 和 SOAPExpress 对不同 XML 负载的对比图

本文 3.1 节分析指出 Apache Axis 1.2 在数据模型映射时频繁使用 Java 反射技术，这使 Axis 处理大负载 XML 数据时性能低下；而 SOAPExpress 采用的基于模板驱动的数据模型映射技术，避免了 Java 反射的应用，因此其处理 SOAP 消息的性能明显优于 Apache Axis 1.2，这进一步证明了基于动态模板驱动的数据模型映射技术的有效性.

## 6　结论和未来工作

本文介绍了一种新的数据模型映射机制“动态提前绑定”，并给出动态提前绑定技术的具体实

① The JBoss Community, Javassist. http://www.jboss.com/products/javassist

现——数据映射模板 DMT. DMT 从概念上说是描述 XML 数据和 Java 数据的上下文无关文法;从实现上说,DMT 是用 Java 语言实现的上下文无关文法对应的带输出的自动机,DMT 驱动的数据模型映射技术实现了 XML 数据和 Java 数据之间的快速映射. SOAPExpress 是基于动态模板 DMT 技术的高性能 SOAP 引擎,实验表明,SOAPExpress 处理 SOAP 消息的性能优于 Apache Axis,对于大型或者中等大小的 SOAP 消息,SOAPExpress 的性能优势尤为明显.

本文所提出的动态提前绑定技术,并没有对 XML Schema 提供完全支持,例如目前的实现中并未对 XML 的命名空间、属性等提供支持,SOAPExpress 将在下一步对 XML Schema 提供完全支持,同时在模板上下文无关文法中加入异常处理功能.

## 参 考 文 献


1 Bustamante F. E., Eisenhauer G., Schwan K., Widener P.. Efficient wire formats for high performance computing. In: Proceedings of the ACM/IEEE SC 2000 Conference (SC'00), Dallas, USA, 2000, 39

2 Davis D., Parashar M.. Latency performance of SOAP implementations. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, Berlin, 2002, 407

3 Elfwing R., Paulsson U., Lundberg L.. Performance of SOAP in Web service environment compared to CORBA. In: Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC'02), Queensland, Australia, 2002, 84

4 Chiu K., Govindaraju M., Bramley R.. Investigating the limits of SOAP performance for scientific computing. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, 2002, 246

5 Kohlhoff C., Steele R.. Evaluating SOAP for high performance business applications: Real-time trading systems. In: Proceedings of the WWW2003, Budapest, Hungary, 2003, 255

6 Ng A., Chen Shiping, Greenfield P.. An evaluation of contemporary commercial SOAP implementations. In: Proceedings of the 5th Australasian Workshop on Software and System Architecture, Adelaide, Australia, 2004, 65

7 Abu-Ghazaleh N., Lewis M. J., Madhusudhan Govindaraju. Differential serialization for optimized SOAP performance. In: Proceedings of the 13th IEEE International Symposium on High Performance. Distributed Computing (HPDC-13), Honolulu, Hawaii, 2004, 55

8 Suzumura T., Takase T., Tatsubori M.. Optimizing Web services performance by differential deserialization. In: Proceedings of the IEEE/ACM International Conference on Web Services, Orlando, USA, 2005, 185

9 Brownell D.. SAX2. Sebastopolo, USA: Reilly & Associates, Inc, 2002

10 Linz P. An Introduction to Formal Languages and Automata. Sudbury, USA: Jones & Bartlett Publishers, 2000



**HUA Lei**, born in 1981, M. S. candidate. His research interests include Web services and distributed computing.

**WEI Jun**, born in 1970, Ph. D., professor. His research interests include service oriented computing, middleware, mobile computing, software engineering.

**NIU Chun-Lei**, born in 1981, M. S. candidate. His research interest includes distributed computing and software engineering.

**ZHENG Hao-Ran**, born in 1981, M. S. candidate. His research interests include SOA and Web services.


**Background**

This research is supported by the National Natural Science Foundation of China under grant No. 60573126 and the National Basic Research Program of China (973 Program) under grant No. 2002CB312005. Both the two projects put emphasis on the service-oriented architecture and Web services research, especially the latter focus on the high performance infrastructure of service-oriented architecture.

Web services, as the mainstream implementation in SOA, suffers performance penalty because its XML-based protocol stack, and SOAP processing is the core performance bottleneck of Web services. The authors try to improve the SOAP processing performance from many perspectives, such as Web services caching, high performance XML parser for SOAP, high performance data model mapping, and optimization on transport protocols. And this paper proposes a new paradigm of data model mapping, "Dynamic Early Binding", which enables to improve SOAP processing by avoiding Java reflection operations, and presents its implementation, Data Mapping Template. This work would be solid progress to improve the Web service performance of the SOA platform.