1.介绍

gSOAP 编译工具提供了一个 SOAP/XML 关于 C/C++ 语言的实现，从而让 C/C++语言开发 web 服务或客户端程序的工作变得轻松了很多。绝大多数的 C++web 服务工具包提供一组 API 函数类库来处理特定的 SOAP 数据结构，这样就使得用户必须改变程序结构来适应相关的类库。与之相反，gSOAP 利用编译器技术提供了一组透明化的 SOAP API，并将与开发无关的 SOAP 实现细节相关的内容对用户隐藏起来。gSOAP 的编译器能够自动的将用户定义的本地化的 C 或 C++数据类型转变为符合 XML 语法的数据结构，反之亦然。这样，只用一组简单的 API 就将用户从 SOAP 细节实现工作中解脱了出来，可以专注与应用程序逻辑的实现工作了。gSOAP 编译器可以集成 C/C++和 Fortran 代码（通过一个 Fortran 到 C 的接口），嵌入式系统，其他 SOAP 程序提供的实时软件的资源和信息；可以跨越多个操作系统，语言环境以及在防火墙后的不同组织。

gSOAP 使编写 web 服务的工作最小化了。gSOAP 编译器生成 SOAP 的代码来序列化或反序列化 C/C++的数据结构。gSOAP 包含一个 WSDL 生成器，用它来为你的 web 服务生成 web 服务的解释。gSOAP 的解释器及导入器可以使用户不需要分析 web 服务的细节就可以

实现一个客户端或服务端程序。

下面是 gSOAP 的一些特点：

×gSOAP 编译器可以根据用户定义的 C 和 C++数据结构自动生成符合 SOAP 的实例化代码。

×gSOAP 支持 WSDL 1.1, SOAP 1.1, SOAP 1.2, SOAP RPC 编码方式以及 literal/document 方式.

×gSOAP 是少数完全支持 SOAP1.1 RPC 编码功能的工具包，包括多维数组及动态类型。比如，一个包含一个基类参数的远程方法可以接收客户端传来的子类实例。子类实例通过动态绑定技术来保持一致性。

×gSOAP 支持 MIME (SwA) 和 DIME 附件包。

×gSOAP 是唯一支持 DIME 附件传输的工具包。它允许你在保证 XML 可用性的同时能够以最快的方式（流方式）传递近乎无大小限制的二进制数据。

×gSOAP 支持 SOAP-over-UDP。

×gSOAP 支持 IPv4 and IPv6.

×gSOAP 支持 Zlib deflate and gzip compression (for HTTP, TCP/IP, and XML file storage)。

×gSOAP 支持 SSL (HTTPS)。

×gSOAP 支持 HTTP/1.0, HTTP/1.1 保持连接, 分块传输及基本验证。

×gSOAP 支持 SOAP 单向消息。

×gSOAP 包含一个 WSDL 生成器，便于 web 服务的发布。

×gSOAP 包含一个 WSDL 解析器 (将 WSDL 转换为 gSOAP 头文件)，可以自动化用户客户端及服务端的开发。

×生成可以单独运行的 web 服务及客户端程序。

×因为只需要很少内存空间，所以可以运行在类似 Palm OS, Symbian, Pocket PC 的小型设备中。

×适用于以 C 或 C++开发的 web 服务中。

×跨平台：Windows, Unix, Linux, Mac OS X, Pocket PC, Palm OS, Symbian 等。

×支持序列化程序中的本地化 C/C++数据结构。

×可以使用输入和输出缓冲区来提高效率，但是不用完全消息缓冲来确定 HTTP 消息的长度。取而代之的是一个三相序列化方法。这样，像 64 位编码的图像就可以在小内存设备（如 PDA）中以 DIME 附件或其他方式传输。

×支持 C++单继承，动态绑定，重载，指针结构（列表、树、图、循环图，定长数组，动态数组，枚举，64 位 2 进制编码及 16 进制编码）。

×不需要重写现有的 C/C++应用。但是，不能用 unions，指针和空指针来作为远程方法调用参数的数据结构中元素。

×三相编组：1）分析指针，引用，循环数据结构；2）确定 HTTP 消息长度；3）将数据序列化位 SOAP1.1 编码方式或用户定义的数据编码方式。

×双相编组：1）SOAP 解释及编码；2）分解“forward”指针（例如：分解 SOAP 中的 href 属性）。

×完整可定制的 SOAP 错误处理机制。

×可定制的 SOAP 消息头处理机制，可以用来保持状态信息。

2、符号规定

3.gSoap2.5 版与 gSOAP 2.4 版（或以前版本）的不同

按照 WS-I Basic Profile 1.0a 的要求，gSOAP2.5 及以上版本默认使用 SOAP RPC 文字。这不需要去关心，因为 WSDL 解析器 wsdl2h 在你提供一个 WSDL 文档时，会自动注意这些不同点。增加了一个 soapcpp2 编译器的编译选项 -e ，用来保持与 gSOAP2.4 及以前版本的兼容性。

4.gSoap2.2 版与 gSOAP 2.1 版（或以前版本）的不同

如果你是从 2.1 版升级到 2.2 或以后版本，请注意这些变化。
为了能够分离传输、内容编码、映射中的接收/发送设置，改变了运行时选项及标志。这些标志分布再四个类中：传输（IO），内容编码（ENC），XML 编组（XML）及 C/C++数据映射。不再提倡使用旧标志 soap_disable_X 及 soap_enable_X（其中，X 表示选项名）。具体内容请参见 9.12 节。

5. gSoap2.x 版与 gSOAP 1.x 版的不同

如果你是从 1.x 版升级到 2.x 版，请注意下面的内容。
gSOAP2.0 及之后的版本是在 1.x 版基础上重写的。gSOAP2.0 之后的版本是线程安全的，但之前版本不是。gSOAP2.x 版本中的主要文件已经重新命名，以便与 1.x 版区分。

gSOAP 1.X	gSOAP 2.X
soapcpp	soapcpp2
soapcpp.exe	soapcpp2.exe
stdsoap.h	stdsoap2.h
stdsoap.c	stdsoap2.c
stdsoap.cpp	stdsoap2.cpp

从 1.x 版升级到 2.x 版并不需要进行大量的代码重写工作。所有 2.x 版相关的函数都定义在 stdsoap2.c[pp]文件中，这个文件是由 gSOAP 编译器自动生成的。所以，用 1.x 版开发的服务端或客户端代码需要进行修改以适应 2.x 版中函数的变化：在 2.x 版中，所有的 gSOAP 函数都增加了一个参数用来保存一个 gSOAP 运行环境实例。这个参数包括了文件描述，表，缓冲，标志位等，它在所有 gSOAP 函数中都是第一个参数。
gSOAP 运行环境实例是一个 struct soap 类型的变量。当客户端程序访问远程方法前或当服务端程序能够接收一个请求前，必须先将这个运行环境变量初始化。在 2.x 版中新增了 3 个函数来负责这些事情：

函数 解释
soap_init(struct soap *soap) 初始化环境变量（只需执行一次）
struct soap *soap_new() 定义并初始化环境变量并返回一个该变量的指针
struct soap *soap_copy(struct soap *soap) 定义一个环境变量并从已有的环境变量中拷贝环境信息

环境变量定义好后就可以重复使用而不必再次初始化了。只有当线程独占访问时，我们才需要一个新的环境变量。例如，下面的代码分配了一个用于多个远程方法的环境变量：

```
int main()
{
  struct soap soap;
  ...
  soap_init(&soap); // 初始化环境变量
  ...
  soap_call_ns__method1(&soap, ...); // 调用一个远程方法
  ...
  soap_call_ns__method2(&soap, ...); // 调用另一个远程方法
  ...
  soap_end(&soap); // 清除环境变量
  ...
}
```

我们也可以像下面这样定义环境变量：

```
int main()
{
  struct soap *soap;
  ...
  soap = soap_new(); // 定义并初始化环境变量
  if (!soap) // 如果不能定义，退出
  ...
  soap_call_ns__method1(soap, ...); // 调用远程函数
  ...
  soap_call_ns__method2(soap, ...); // 调用另一个远程函数
  ...
  soap_end(soap); // 清除环境变量
  ...
  free(soap); // 释放环境变量空间
}
```

服务端代码在调用 soap_serve 函数前，需要定义相关环境变量:

```
    int main()
{
```

```
  struct soap soap;
  soap_init(&soap);
  soap_serve(&soap);
}
```

或者像下面这样:

```
int main()
{
  soap_serve(soap_new());
}
```

soap_serve 函数用来处理一个或多个（当允许 HTTP keep-alive 时，参见 18.11 节中的 SOAP_IO_KEEPALIVE 标志）请求。

一个 web 服务可以用多线程技术来处理请求：

```
int main()
{
  struct soap soap1, soap2;
  pthread_t tid;
  ...
  soap_init(&soap1);
  if (soap_bind(&soap1, host, port, backlog) < 0) exit(1);
  if (soap_accept(&soap1) < 0) exit(1);
  pthread_create(&tid, NULL, (void*(*)(void*))soap_serve, (void*)&soap1);

  ...
  soap_init(&soap2);
  soap_call_ns__method(&soap2, ...); // 调用远程方法
  ...
  soap_end(&soap2);
  ...
  pthread_join(tid, NULL); // 等待线程结束
  soap_end(&soap1); // 释放环境变量
}
```

在上面的例子中，需要两个环境变量信息。而在 1.x 版本中，由于静态分配环境变量，多线程技术是不被允许的（只有一个线程可以用这个环境变量调用远程方法或处理请求信息）。

8.2.4 节将给出一个具体的多线程服务实例，它为每个 SOAP 请求分配一个独立线程进行处理。

6、

gSOAP 使用下面的软件包验证了其可用性: Apache 2.2 Apache Axis ASP.NET Cape Connect Delphi easySOAP++ eSOAP Frontier GLUE Iona XMLBus kSOAP MS SOAP Phalanx SIM SOAP::Lite SOAP4R Spray SQLData Wasp Adv. Wasp C++ White Mesa xSOAP ZSI 4S4C

准备工作

要开始用 gSOAP 创建一个 web 服务应用，你需要:

一个 C/C++编译器.

拥有根据操作系统平台创建的可执行的 gSOAP 的 stdsoap2（windows 下为 stdsoap2.exe）编译器。

拥有根据操作系统平台创建的可执行的 gSOAP 的 wsdl2h（windows 下为 wsdl2h.exe）WSDL 解析器。

需要'stdsoap2.c'或'stdsoap2.cpp'及'stdsoap2.h'文件来实现你的 SOAP 功能。你可以创建一个 dll 或动态库以便简化连接。

如果你要支持 SSL（HTTPS）及压缩的话，可以安装 OpenSSL 及 Zlib 库。

gSOAP 是独立开发包，不需要任何第三方的软件支持（除非你要用到 OpenSSL 及 Zlib）。

与平台无关的 gSOAP 版本需要你下面的工具编译'soapcpp2'及'wsdl2h'文件：

一个 C++编译器（用来编译'wsdl2h'WSDL 解析器）。

Bison 或 Yacc

Flex 或 Lex

推荐使用 Bison 及 Flex。

在软件包 samples 目录下有大量的开发实例。可以用'make'来编译这些例子。这些例子包含了 gSOAP 中的各个方面。其中，最简单的例子是 one-liners（samples/oneliners）。

快速指南

本指南旨在让你快速开始你的 gSOAP 开发之旅。阅读本节的内容，需要你对 SOAP 1.1 协议及 C/C++语法有大体的了解。虽然使用 gSOAP 编译器可以直接用 C/C++开始编写 web 服务及客户端程序而不需要了解 SOAP 协议的细节，但是由于我们在本节中使用了大量的实例来说明 gSOAP 与其他 SOAP 实现的连接及通讯，所以了解一些 SOAP 及 WSDL 协议也是必需的。

8.1 如何使用 gSOAP 编译环境来编译 SOAP 客户端程序

通常，一个 SOAP 客户端应用的实现需要为每个客户端需要调用的远程方法提供一个存根例程（stub routine）。存根例程主要负责编码参数信息；将包含参数信息的调用请求发

送给制定的 SOAP 服务；等待返回结果；将结果中的参数信息编码。客户端程序调用访问远程方法的存根例程就像调用本地方法一样。用 C/C++手工别写一个存根例程是个十分痛苦的差使，尤其当远程方法的参数中包含特定的数据结构（如：记录、数组、图等）时。幸运的是，gSOAP 包中的'wsdl2h'WSDL 解析器和'soapcpp2'存根及架构编译器能够将 web 服务客户端及服务端的开发工作自动化。

'soapcpp2'存根及架构编译器是可以生成构建 C++ SOAP 客户端所需的 C++源码的预编译器。该预编译器的输入参数是一个标准的 C/C++头文件。这个头文件可以由 WSDL 解析器根据相关的 WSDL 文档自动生成。

参见下面的命令：

```
$ wsdl2h -o quote.h http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

上面的命令根据制定 URL 提供的 WSDL 文档生成一个 C++语法结构的头文件。

如果需要生成一个纯 C 的头文件，需要一下命令：

```
$ wsdl2h -c -o quote.h http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

更多关于 WSDL 解析器及其选项的细节信息，请参见 8.2.10 节。

执行上述命令后，quote.h 文件就生成了。其中包含开发客户端或服务端程序的存根例程定义。SOAP 服务远程方法以函数声明的方式在这个头文件中被定义。C/C++源代码的存根例程将通过预编译器自动实现。同时，每个远程方法的程序框架也被自动生成了，它可以用来建立 SOAP 服务端程序应用。

SOAP 服务的输入输出参数可以是简单的数据类型或复杂的数据结构，可以由 WSDL 解析器自动生成或手工定义。预编译器将自动生成序列化/反序列化这些数据的代码，以便存根例程可以将这些数据以 XML 的方式编码或解码。

8.1.1 例子

XMethods Delayed Stock Quote 服务提供一个 getQuote 方法（由'wsdl2h'WSDL 解析器生成的 quote.h 定义）。这个方法根据提供的股票名称返回相应的股票价格。下面是这个方法的 WSDL 文档信息：

```
    Endpoint URL: http://services.xmethods.net:80/soap
SOAP action: "" (2 quotes)
Remote method namespace: urn:xmethods-delayed-quotes
Remote method name: getQuote
  Input parameter: symbol of type xsd:string
  Output parameter: Result of type xsd:float
```

下面是由 WSDL 解析器生成的 getQuote.h 头文件（实际的文件内容与'wsdl2h'版本及生成选项有关）：

```
//gsoap ns1 service name: net_DOTxmethods_DOTservices_DOTstockquote_DOTStockQuoteBinding
//gsoap ns1 service type: net_DOTxmethods_DOTservices_DOTstockquote_DOTStockQuotePortType
```

```
//gsoap ns1 service port: http://66.28.98.121:9090/soap
//gsoap ns1 service namespace: urn:xmethods-delayed-quotes
//gsoap ns1 service documentation: Definitions generated by the gSOAP WSDL parser 1.0
// Service net.xmethods.services.stockquote.StockQuoteService : net.xmethods.services.stockquote.StockQuote web service

    //gsoap ns1 service method-style: getQuote rpc
//gsoap ns1 service method-encoding: getQuote http://schemas.xmlsoap.org/soap/encoding/
//gsoap ns1 service method-action: getQuote urn:xmethods-delayed-quotes#getQuote
int ns1__getQuote(char *symbol, float &Result);
```

这个头文件用 C/C++代码为 gSOAP 预编译器指定了 web 服务的细节。远程方法被定义为函数 ns1__getQuote，同时指定了客户端调用 web 服务所需的所有细节信息。

getQuote 远程方法需要一个名为 symbol 的字符串作为输入参数，同时需要一个名为 Result 的浮点数作为输出参数。预编译器生成的远程方法调用函数中，最后一个参数必须是输出参数，该参数以引用方式传递或定义为指针类型。除此之外的所有参数都是输入参数，这些参数必须以传值方式传递。函数返回一个整型值，其值说明 web 服务调用成功或出现的错误。具体的错误代码信息参见 10.2 节。

函数名中命名空间前缀 ns1__的细节信息将在 8.1.2 节中讨论。简单的说，命名空间前缀与函数名之间用两个下划线分割。比如，ns1__getQuote 中，ns1 为命名空间前缀，getQuote 是函数名称。（函数名中单个下划线将在 XML 中解释为破折号-，因为在 XML 中破折号比下划线更常用，细节请参见 10.3 节）

用下面命令执行预编译器：

```
soapcpp2 getQuote.h
```

预编译器根据 getQuote.h 中定义的信息来生成存根例程的代码框架。这个存根例程可以在客户端程序中随处调用。存根例程被声明为下面的样子：

```
int soap_call_ns1__getQuote(struct soap *soap, char *URL, char *action, char *symbol, float &Result);
```

存根例程保存在 soapClient.cpp 文件中；soapC.cpp 文件包含了序列化、反序列化数据的函数。你可以用 -c 编译选项来生成纯 C 的代码。

注意，soap_call_ns1__getQuote 在 ns1__getQuote 的参数基础上又增加了三个参数：soap 必须是指向一个 gSOAP 运行环境的合法指针；URL 是 web 服务的 URL；action 指明了 web 服务需要的 SOAP action。XMethods Delayed Stock Quote 服务的 URL 是 http://66.28.98.121:9090/soap，action 是"" (2 quotes)。你可以动态的改变 URL 及 action。如果上述两个变量定义为 NULL，则会使用头文件中定义的信息。

下面的例子调用远程方法获取 IBM 的股票信息：

```
#include "soapH.h" // 包含生成的存根例程定义
#include "net_DOT_xmethods_DOT_services_DOT_stockquote_DOT_StockQuoteBinding.nsmap" // 包含命名空间表
int main()
{
  struct soap soap; // gSOAP 运行环境
  float quote;
  soap_init(&soap); // 初始化运行环境（只执行一次）
  if (soap_call_ns1__getQuote(&soap, NULL, NULL, "IBM", &quote) == SOAP_OK)
    std::cout << "Current IBM Stock Quote = " << quote << std::endl;
  else // an error occurred
    soap_print_fault(&soap, stderr); // 在 stderr 中显示错误信息
  soap_destroy(&soap); // 删除类实例（仅用于 C++中）
  soap_end(&soap); // 清楚运行环境变量
  soap_done(&soap); // 卸载运行环境变量
  return 0;
}
```

调用成功后，存根例程返回 SOAP_OK 同时 quote 变量保存着股票信息；如果调用失败则产生一个错误，同时通过 soap_print_fault 函数显示错误信息。
gSOAP 编译器同时为 C++客户端程序生成了一个代理类。

```
#include "soapnet_DOT_xmethods_DOT_services_DOT_stockquote_DOT_StockQuoteBindingProxy.h" // 获得代理
#include "net_DOT_xmethods_DOT_services_DOT_stockquote_DOT_StockQuoteBinding.nsmap" // 包含命名空间表
int main()
{
  net q; // "net" 是这个服务代理类的短名称
  float r;
  if (q.ns1__getQuote("IBM", r) == SOAP_OK)
    std::cout << r << std::endl;
  else
    soap_print_fault(q.soap, stderr);
  return 0;
}
```

代理类的构造函数定义并初始化了一个 gSOAP 环境变量实例。所有的 HTTP 及 SOAP/XML 处理被隐藏在后台处理。你可以改变 WSDL 解析器生成的头文件中 web 服务的名称。web 服务的名字是由 WSDL 内容中萃取的，并不总是短名称格式的。你可以随意更改这个条目

//gsoap ns1 service name: net_DOT_xmethods_DOT_services_DOT_stockquote_DOT_StockQuoteBinding

来使用一个更合适的名字。这个名字将决定代理类文件及 XML 命名空间表文件的名字。

下面的函数可以用来建立一个 gSOAP 运行环境（struct soap）：

soap_init(struct soap *soap) 初始化运行环境变量（只需要执行一次）

soap_init1(struct soap *soap, soap_mode iomode) 初始化运行环境变量同时设置 in/out 模式

soap_init2(struct soap *soap, soap_mode imode, soap_mode omode) 初始化运行环境变量同时分别设置 in/out 模式

struct soap *soap_new() 定义、初始化运行环境并返回一个执行运行环境的指针

struct soap *soap_new1(soap_mode iomode) 定义、初始化运行环境并返回一个执行运行环境的指针并设置 in/out 模式

struct soap *soap_new2(soap_mode imode, soap_mode omode) 定义、初始化运行环境并返回一个执行运行环境的指针并分别设置 in/out 模式

struct soap *soap_copy(struct soap *soap) 定义一个新的环境变量并将现有环境信息赋值给新的变量

soap_done(struct soap *soap) 重置、关闭连接，清除环境变量

环境变量可以在程序中任意次数的使用。每个新的线程就需要一个新的环境变量实例。

当例子中的客户端程序执行时，SOAP 请求通过 soap_call_ns1__getQuote 函数来调用，它生成下面的 SOAP RPC 请求信息：

```
POST /soap HTTP/1.1
Host: services.xmethods.net
Content-Type: text/xml
Content-Length: 529
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="urn:xmethods-delayed-quotes"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Body>
<ns1:getQuote>
<symbol>IBM</symbol>
</ns1:getQuote>
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

XMethods Delayed Stock Quote 服务返回如下的信息：

```
HTTP/1.1 200 OK
Date: Sat, 25 Aug 2001 19:28:59 GMT
Content-Type: text/xml
Server: Electric/1.0
Connection: Keep-Alive
Content-Length: 491
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/?
<soap:Body>
<n:getQuoteResponse xmlns:n="urn:xmethods-delayed-quotes?
<Result xsi:type="xsd:float?41.81</Result>
</n:getQuoteResponse>
</soap:Body>
</soap:Envelope>
```

服务返回的信息通过存根例程来解析，并保存在 soap_call_ns1__getQuote 函数的 quote 参数中。

客户端程序可以在任意时间多次调用远程方法。请看下面的例子：

```
...
struct soap soap;
float quotes[3]; char *myportfolio[] = {"IBM", "MSDN"};
soap_init(&soap); // need to initialize only once
for (int i = 0; i < 3; i++)
  if (soap_call_ns1__getQuote(&soap, "http://services.xmethods.net:80/soa
p", "", myportfolio[i], &quotes[i]) != SOAP_OK)
    break;
if (soap.error) // an error occurred
  soap_print_fault(&soap, stderr);
soap_end(&soap); // clean up all deserialized data
...
```

这个客户端程序通过调用 ns1__getQuote 存根例程来为数组中的每个股票代码获得信息。

上面的例子给我们示范了使用 gSOAP 创建一个 SOAP 客户端时多么容易的事情啊。

8.1.2 关于命名空间

函数 ns1__getQuote（上节提到的）中，使用了 ns1__作为远程方法的命名空间。使用命名空间是为了防止远程方法名冲突，比方多个服务中使用同一个远程方法名的情况。

命名空间前缀及命名空间名称同时也被用来验证 SOAP 信息的内容有效性。存根例程通过命名空间表中的信息来验证服务返回信息。命名空间表在运行时被取出用于解析命名空间绑定，反序列化数据结构，解码并验证服务返回信息。命名空间表不应该包含在 gSOAP 预编译器所需输入的头文件中。在 18.2 节中将会对命名空间表做详细介绍。

Delayed Stock Quote 服务客户端的命名空间表如下：

```
    struct Namespace namespaces[] =
{   // {"命名前缀", "空间名称"}
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // 必须是第一行
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // 必须是第二行
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance"}, // 必须是第三行
  {"xsd", "http://www.w3.org/2001/XMLSchema"}, // 2001 XML 大纲
  {"ns1", "urn:xmethods-delayed-quotes"}, // 通过服务描述获取
  {NULL, NULL} // 结束
};
```

第一行命名空间是 SOAP1.1 协议默认命名空间。事实上，命名空间表就是用来让程序员可以规定 SOAP 编码方式，能够用包含命名空间的命名空间前缀来满足指定 SOAP 服务的命名空间需求的。举例来说，使用前面命名空间表中定义的命名空间前缀 ns1，存根例程就可以对 getQuote 方法的请求进行编码。这个过程由 gSOAP 预编译器通过在 getQuote.h 文件中定义的包含前缀 ns1 的 ns1__getQuote 函数自动完成。通常，如果要在远程方法名，结构名，类名，字段名等结构或类中使用命名空间前缀，就必须在命名空间表中进行定义。

命名空间表将会被存根例程封装，并按下面的形式输出：

```
    ...
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="urn:xmethods-delayed-quotes"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

...

这个命名空间绑定将被 SOAP 服务用来验证 SOAP 请求。

8.1.3 例子

使用命名空间前缀可以解决在不同的服务中使用相同名称的远程方法的问题，看下面的例子：

```
// Contents of file "getQuote.h":
int ns1__getQuote(char *symbol, float &Result);
int ns2__getQuote(char *ticker, char *&quote);
```

这个例子允许客户端程序使用不同的命名空间以便连接到不同的服务程序执行其中的远程方法。

命名空间前缀也可以用在类声明中使用，在 XML 大纲中区分同名但不同命名空间的 SOAP 值。例如：

```
class e__Address // an electronic address
{
  char *email;
  char *url;
};
class s__Address // a street address
{
  char *street;
  int number;
  char *city;
};
```

在生成的序列化函数中，使用 e__Address 的一个实例来表示 e 命名空间前缀的一个地址元素类型。

```
<e:Address xsi:type="e:Address">
<email xsi:type="string">me@home</email>
<url xsi:type="string">www.me.com</url>
</e:Address>
```

用 s__Address 的一个实例来表示 s 命名空间前缀的一个地址元素类型。

```
<s:Address xsi:type="s:Address">
<street xsi:type="string">Technology Drive</street>
<number xsi:type="int">5</number>
```

```
<city xsi:type="string">Softcity</city>
</s:Address>
```

客户端程序的命名空间表必须有 e 和 s 的数据类型定义：

```
struct Namespace namespaces[] =
{ ...
  {"e", "http://www.me.com/schemas/electronic-address"},
  {"s", "http://www.me.com/schemas/street-address"},
...
```

命名空间表必须作为客户端程序的一部分，使客户端程序在运行时可以对数据进行序列化及反序列化。

8.1.4 如何建立客户端程序代理类

用于 C++客户端程序的代理类信息是由 gSOAP 预编译器自动创建的。为了说明代理类的生成过程，我们在 getQuote.h 头文件中加入一些信息，以便 gSOAP 预编译器可以生成代理类。这些信息就类似于 WSDL 解析器自动生成的头文件中就已经包含的信息。

```
//"getQuote.h"的内容:
//gsoap ns1 service name: Quote
//gsoap ns1 service location: http://services.xmethods.net/soap
//gsoap ns1 service namespace: urn:xmethods-delayed-quotes
//gsoap ns1 service style: rpc
//gsoap ns1 service encoding: encoded
//gsoap ns1 service method-action: getQuote ""
int ns1__getQuote(char *symbol, float &Result);
```

前三行指令用于定义代理类的名称，服务地址，命名空间。第四行、第五行指令定义了使用 SOAP RPC 编码方式。最后一行定义了可选的 SOAPAction。当需要 SOAPAction 时，这行信息将提供给每个远程方法。使用 soapcpp2 对该头文件进行编译后，将会产生 soapQuoteProxy.h 文件。它包含下面的内容：

```
#include "soapH.h"
class Quote
{ public:
  struct soap *soap;
  const char *endpoint;
  Quote() { soap = soap_new(); endpoint = "http://services.xmethods.net
/soap"; };
  ~Quote() { if (soap) { soap_destroy(soap); soap_end(soap); soap_done
(soap); free((void*)soap); }};
  int getQuote(char *symbol, float &Result) { return soap ? soap_call_ns1
```

```
__getQuote(soap, endpoint, "", symbol, Result) : SOAP_EOM; };
};
```

为了能够在运行时刻对 gSOAP 环境变量及命名空间进行定制，上述两个变量被定义程全局变量。

生成的代理类可以同命名空间表一起包含在客户端程序中，请看下面的例子：

```
    #include "soapQuoteProxy.h" // 获得代理类
#include "Quote.nsmap" // 获得命名空间绑定
int main()
{
  Quote q;
  float r;
  if (q.ns1__getQuote("IBM", r) == SOAP_OK)
    std::cout << r << std::endl;
  else
    soap_print_fault(q.soap, stderr);
  return 0;
}
```

Quote 构造函数定义并初始化了一个 gSOAP 运行环境实例。所有的 HTTP 及 SOAP /XML 进程都被隐藏在后台自动执行。

如果你需要多个命名空间表或要联合多个客户端，你可以在执行 soapcpp2 时添加参数-n 及-p 来生成命名空间表以防止冲突。详细信息请看 9.1 及 18.33 节。同时，你可以使用 C++代码命名空间来创建一个命名空间限制的代理类，详细信息请看 18.32 节。

8.1.5 XSD 类型编码

许多 SOAP 服务需要在 SOAP 负载中使用 XML 编码。在 gSOAP 预编译器中使用的默认编码为 SOAP RPC 编码。然而，使用 XSD 类型编码可以改善互操作性。XSD 类型在头文件中用 typedef 定义。举个例子，下面的定义将 C/C++类型转换为 XSD 类型：

```
    // Contents of header file:
...
typedef char *xsd__string; // encode xsd__string value as the xsd:string s
chema type
typedef char *xsd__anyURI; // encode xsd__anyURI value as the xsd:any
URI schema type
typedef float xsd__float; // encode xsd__float value as the xsd:float schem
a type
typedef long xsd__int; // encode xsd__int value as the xsd:int schema typ
e
typedef bool xsd__boolean; // encode xsd__boolean value as the xsd:bool
ean schema type
```

```
typedef unsigned long long xsd__positiveInteger; // encode xsd__positiveIn
teger value as the xsd:positiveInteger schema type
...
```

这些简单的声明告诉 gSOAP 预编译器当远程方法参数中使用上述定义的类型时，就把相关的 C++类型转当作内建的 XSD 类型进行编码、解码。同时，使用 typedef 不需要使用内建 C++类型的客户端或服务端程序更改现有代码（但只是当参数为简单的 C++类型时，请参看 11.2.2 节来使用 XSD 类型表示组合的数据类型）。

8.1.6 例子

重新考虑一席 getQuote 的例子。现在用 XSD 类型来重写代码：

```
// Contents of file "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, xsd__float &Result);
```

使用预编译器编译这个头文件，将会生成与原来相同的存根例程代码：

```
int soap_call_ns1__getQuote(struct soap *soap, char *URL, char *actio
n, char *symbol, float &Result);
```

客户端程序不需要进行任何改动，即可使用 XSD 类型类编码、解码数据类型信息了。举个例子，当客户端程序调用代理时，代理方法用 xsd:string 类型产生一个 SOAP 请求：

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">IBM</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

服务端的返回码为:

```
...
<soap:Body>
<n:getQuoteResponse xmlns:n="urn:xmethods-delayed-quotes">
<Result xsi:type="xsd:float">41.81</Result>
</n:getQuoteResponse>
</soap:Body>
...
```

8.1.7 如何改变回传元素的名称

SOAP 返回消息重的元素命名没有固定的方式，但是推荐使用方法名加 Response 结尾。

例如，getQuote 方法的返回参数为 getQuoteResponse。
返回参数的名称可以在头文件中以类或结构体的方式声明。这个类或结构体的名字就是服务返回参数的名字。因此，远程方法的输出参数必须声明为类或结构体的一个（多个）字段。gSOAP 预编译器可以自动生成用于存根例程使用的结构体或类。

8.1.8 例子

我们将 getQuote 远程方法的返回参数做如下的改变：

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
struct ns1__getQuoteResponse {xsd__float Result;};
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse &
r);
```

输入参数还是和原来一样的:

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">IBM</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

不同的是输出参数必须符合 getQuoteResponse 的名称并有同样的命名空间:

```
...
<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</soap:Body>
...
```

类或结构体的定义也可以在函数内部进行，如:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse {xsd
__float Result;} &r);
```

8.1.9 如何指定多个输出参数

gSOAP 预编译器将远程方法的最后一个参数作为输出参数，其余的参数都作为输入参数。如果要使用多个输出参数，就必须将输出参数定义为结构或类的形式。

8.1.10 例子

下面的例子中，getNames 函数有个 SSN 作为输入参数，还有两个输出参数 first 及 last。这个函数可以如下定义：

```
// Contents of file "getNames.h":
int ns3__getNames(char *SSN, struct ns3__getNamesResponse {char *first;
 char *last;} &r);
```

gSOAP 将按照上述信息生成函数 soap_call_ns3__getNames. 其请求信息为:

```
...
<SOAP-ENV:Envelope ... xmlns:ns3="urn:names" ...>
...
<ns3:getNames>
<SSN>999 99 9999</SSN>
</ns3:getNames>
...
```

输出信息为:

```
...
<m:getNamesResponse xmlns:m="urn:names">
<first>John</first>
<last>Doe</last>
</m:getNamesResponse>
...
```

其中 frist 及 last 是远程方法的两个输出参数。

如果使用同一个变量作为输入、输出参数，也可以用结构体来定义。看下面的例子：

```
// Content of file "copy.h":
int X_rox__copy_name(char *name, struct X_rox__copy_nameResponse {c
har *name;} &r);
```

用类或结构体来定义为返回值，就可以定义上述输入、输出参数相同的远程方法。gSOAP 生成的请求信息如下：

```
...
<SOAP-ENV:Envelope ... xmlns:X-rox="urn:copy" ...>
...
```

```
<X-rox:copy-name>
<name>SOAP</name>
</X-rox:copy-name>
...
```

返回信息如下:

```
...
<m:copy-nameResponse xmlns:m="urn:copy">
<name>SOAP</name>
</m:copy-nameResponse>
...
```

8.1.11 如何将输出参数定义为类和结构体类型

如果远程方法唯一的返回参数使用类或结构体等复杂的数据类型，就必须在调用远程方法时使用类或结构作为返回元素。

8.1.12 例子

下面举个例子来说明一下。Flighttracker 服务提供实时的航班信息。该服务需要一个航班代码及飞行号码作为输入参数。远程方法名字为 getFlightInfo，它包含两个字符串参数：航班代码，飞行号码。这两个参数都必须用 xsd:string 编码作为参数类型。该远程方法返回一个 getFlightResponse 元素，其中包含一个复杂数据类型 FlightInfo。FlightInfo 类型在头文件中以类方式定义，他的字段信息与 FlightInfo 类型中的字段相同。

```
// Contents of file "flight.h":
typedef char *xsd__string;
class ns2__FlightInfo
{
  public:
  xsd__string airline;
  xsd__string flightNumber;
  xsd__string altitude;
  xsd__string currentLocation;
  xsd__string equipment;
  xsd__string speed;
};
struct ns1__getFlightInfoResponse {ns2__FlightInfo return_;};
int ns1__getFlightInfo(xsd__string param1, xsd__string param2, struct ns1
__getFlightInfoResponse &r);
```

返回元素 ns1__getFlightInfoResponse 中包含一个 ns2__FlightInfo 类型的字段 return_。该字段以_结尾，以便和 return 关键字分别。详细信息请参看 10.3 节：将 C++标识符转换为 XML 元素名。

gSOAP 预编译器生成 soap_call_ns1__getFlightInfo 代理类。下面是一个客户端使用这个代理类的例子：

```
    struct soap soap;
...
soap_init(&soap);
...
soap_call_ns1__getFlightInfo(&soap, "testvger.objectspace.com/soap/servlet
/rpcrouter",
  "urn:galdemo:flighttracker", "UAL", "184", r);
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema"},
  {"ns1", "urn:galdemo:flighttracker"},
  {"ns2", "http://galdemo.flighttracker.com"},
  {NULL, NULL}
};
```

客户端执行时，代理类生成如下请求信息：

```
    POST /soap/servlet/rpcrouter HTTP/1.1
Host: testvger.objectspace.com
Content-Type: text/xml
Content-Length: 634
SOAPAction: "urn:galdemo:flighttracker"

    <?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="urn:galdemo:flighttracker"
  xmlns:ns2="http://galdemo.flighttracker.com"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<SOAP-ENV:Body>
<ns1:getFlightInfo xsi:type="ns1:getFlightInfo">
<param1 xsi:type="xsd:string">UAL</param1>
<param2 xsi:type="xsd:string">184</param2>
</ns1:getFlightInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Flighttracker 服务返回:

```
HTTP/1.1 200 ok
Date: Thu, 30 Aug 2001 00:34:17 GMT
Server: IBM_HTTP_Server/1.3.12.3 Apache/1.3.12 (Win32)
Set-Cookie: sesessionid=2GFVTOGC30D0LGRGU2L4HFA;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Content-Length: 861
Content-Type: text/xml; charset=utf-8
Content-Language: en

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getFlightInfoResponse xmlns:ns1="urn:galdemo:flighttracker"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<return xmlns:ns2="http://galdemo.flighttracker.com" xsi:type="ns2:FlightI
nfo">
<equipment xsi:type="xsd:string">A320</equipment>
<airline xsi:type="xsd:string">UAL</airline>
<currentLocation xsi:type="xsd:string">188 mi W of Lincoln, NE</currentL
ocation>
<altitude xsi:type="xsd:string">37000</altitude>
<speed xsi:type="xsd:string">497</speed>
<flightNumber xsi:type="xsd:string">184</flightNumber>
</return>
</ns1:getFlightInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

代理类使用 ns1__getFlightInfoResponse 结构变量 r 来返回信息：

```
cout << r.return_.equipment << " flight " << r.return_.airline << r.re
turn_.flightNumber
  << " traveling " << r.return_.speed << " mph " << " at " << r.return
_.altitude
  << " ft, is located " << r.return_.currentLocation << endl;
```

上述代码将显示如下信息:

```
A320 flight UAL184 traveling 497 mph at 37000 ft, is located 188 mi
W of Lincoln, NE
```

8.1.13 如何指定匿名参数名

SOAP1.1 协议允许使用匿名参数。也就是说，远程方法的输出参数名不必与客户端的参数名严格保持一致。同样，远程方法的输入参数名也不必与客户端的参数名严格保持一致。虽然这种转换在 SOAP1.2 版本中不被提倡，但是 gSOAP 预编译器可以生成支持匿名参数的存根例程。只要在远程方法调用函数中的参数名称省略就可以使用匿名参数了。看下面的例子：

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string, xsd__float&);
```

要让参数名称在接收端匿名，在头文件定义函数时，参数名应该以下划线(_)开始。举个例子:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, xsd__float &_return);
```

或者用一个结构体返回信息:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
struct ns1__getQuoteResponse {xsd__float _return;};
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse &
r);
```

在这个例子中，_return 是个匿名输出参数。当服务回应该请求时，就可以使用任意的输出参数名称。输入参数名也可以是匿名的。这将影响 web 服务在 gSOAP 中的实现以

及服务中参数名称的对应关系。
如果使用匿名参数，则函数定义中的参数顺序必须与所调用服务的参数顺序保持一致。

8.1.14 如何定义没有输入参数的方法

要定义一个没有输入参数的方法，只需要在函数定义中只保留一个参数作为输出参数就可以了。但是，某些 C/C++编译器不能够编译空结构体（该结构体是 gSOAP 生成存储 SOAP 请求信息用）。这样，我们指定一个输入参数，使其类型为 void*（gSOAP 不能序列化 void* 数据）。举个例子：

```
    struct ns3__SOAPService
{
  public:
  int ID;
  char *name;
  char *owner;
  char *description;
  char *homepageURL;
  char *endpoint;
  char *SOAPAction;
  char *methodNamespaceURI;
  char *serviceStatus;
  char *methodName;
  char *dateCreated;
  char *downloadURL;
  char *wsdlURL;
  char *instructions;
  char *contactEmail;
  char *serverImplementation;
};
struct ArrayOfSOAPService {struct ns3__SOAPService *__ptr; int __size;};

int ns__getAllSOAPServices(void *_, struct ArrayOfSOAPService &_retur
n);
```

ns__getAllSOAPServices 方法包含一个 void*类型的输入参数，它不会被 gSOAP 做序列化处理。多数的 C/C++编译器允许空的结构体，所以 void*类型的参数就不必使用了。

8.1.15 如何定义没有输出参数的方法

要指定一个没有输出参数的方法，需要将输出参数类型定义为一个空的结构体指针。

```
    enum ns__event { off, on, stand_by };
int ns__signal(enum ns__event in, struct ns__signalResponse { } *out);
```

因为输出结构体为空，所以就没有输出参数了。如果编译器不支持空结构体，就指定一个包含 void*类型变量的结构体。

有些 SOAP 资源用空的输出参数作为单向 SOAP 消息使用。但是，我们使用异步调用方式来支持单向消息。我们将在 8.3 节中详细讨论。