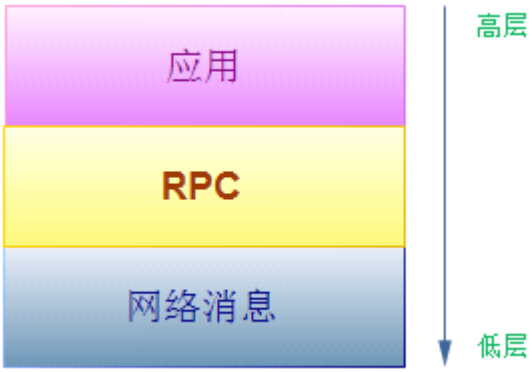
RPC 的实现

易剑 2012/4/21

目录

1. 前言

RPC 全称为 Remote Procedure Call，即远过程调用。如果没有 RPC，那么跨机器间的进程通讯通常得采用消息，这会降低开发效率，也会增加网络层和上层的耦合度， RPC 可以帮助我们解决这些问题。



从上图可以看出，RPC 是基于消息实现的，只不过它处于更上层，做了一层抽象和封装。 实现上有很多现存的 RPC 实现，如 Facebook 出品 Thrift、微软的 COM/DCOM/COM+、跨平台的 Corba、以及 ACE 提供的 Tao 等。本文将力图用比较简单的语言阐述一个 RPC 是如何实现的。

2. 基本概念

在正式讲解之前，先介绍一下与 RPC 有关的基本概念：

2.1.IDL

IDL 的全称是 Interface Definition Language，即接口定义语言（有时也叫作接口描述语言）。因为 RPC 通常是跨进程、跨机器、跨系统和跨语言的，IDL 是用来解决这个问题的，它与语言无关，借助编译器将它翻译成不同的编程语言。

Google 开源的 ProtoBuf 中的“.proto”文件就是一种 IDL 文件：

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

在.proto 文件中 message 类似于 C 语言中的 struct 的，转换成 C++语言后，它对应于 C++中的一个类。有关 ProtoBuf 的更多信息，可参考：http://code.google.com/p/protobuf/。

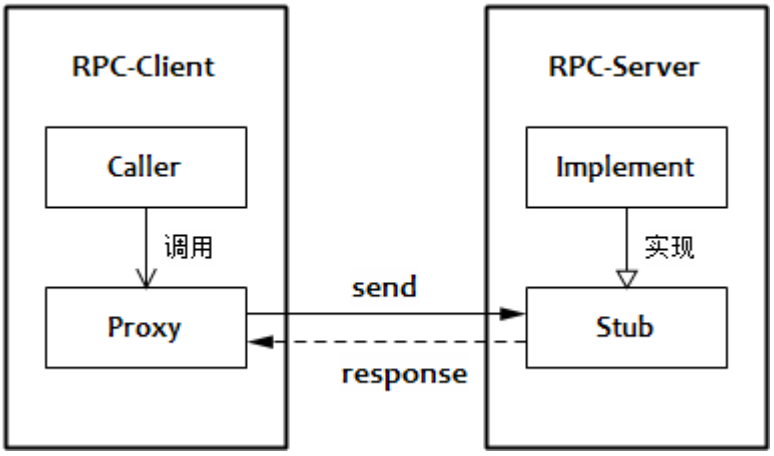
请注意，IDL 中的数据类型（如 ProtoBuf 中的 int32）是独立于任何语言的，但它通常会和目标语言中的数据类型有着映射关系，否则将无法把 IDL 文件编译成目标语言文件。

2.2.代理（Proxy）

代理（Proxy）是 RPC 的客户端实现，客户端代码总是通过代理来与 RPC 服务端通讯。Proxy 的代码完全由 IDL 编译器生成。

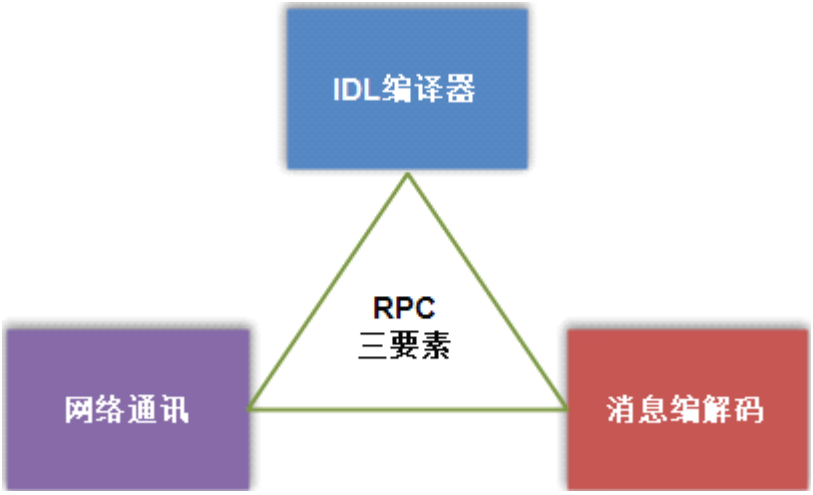
2.3.存根（Stub）

存根（Stub）是 RPC 的服务端实现。在服务端，需要实现 IDL 文件中定义的接口；而在客户端直接使用。代理和存根的关系如下图所示：



3. 三要素

要实现一个完整的 RPC，需要完成以下三件事，在这里我们把这三件事称作三要素：



3.1.网络通讯

负责将客户端的请求发送到服务端，和将服务端的响应回送给客户端。这是大家都熟悉的一块，主要就是高性能网络程序的实现。

3.2.消息编解码

IDL 中定义接口、函数和数据等，需要在发送前编码成字节流，在收到后进行解码。比如将函数名、参数类型和参数值等编码成字节流，然后发送给对端，然后对端进行解码，还原成函数调用。ProtoBuf 就是一个非常好的编解码工具。

请注意 IDL 支持的所有数据类型要求是可编解码的，IDL 编译器需要知道如何将它编码到字节流，和从字节流里解码还原出来。

3.3.IDL 编译器

对大多数人来说，这块的工作是陌生的，因为日常开发接触不多。也因为如此，一般人都会觉得这块很难高深。其实只要克服心理障碍，学习它比想象中的要容易许多。总而言之，上手并不难，特别是在阅读了本文之后。但是如果需要实现一个类似于 Hive 或 GCC 东东，那是有相当大的难度的，其中对优化语法树就是一个非常大的挑战。

为了能够使用 RPC，需要将 IDL 文件编译成指定的语言代码。ProtoBuf 实际上已经实现了这个功能。如果基于 ProtoBuf 实现一个 RPC，则这 IDL 编译这一步可以跳过，将只需要实现网络通讯，以及实现 google::protobuf::RpcController 和 google::protobuf::RpcChannel 两个接口即可。

本文是为了介绍 RPC 的实现，目标是让读者能够自己实现一套 RPC，而对于三要素中的网络通讯和消息编解码，一般人都容易理解和上手，但对于 IDL 编译这块相对会陌生许多。为此，本文余下部分将着重介绍 IDL 编译的实现，所有的实现都将基于 Flex 和 Bison 两个开源的工具，当然也可使用 JavaCC、SableCC 和 Antlr（ANother Tool for Language Recognition）等。

4. flex 和 bison

经典的 lex 和 yacc 由贝尔实验室在 1970 年代开发，flex 和 bison 是它们的现代版本。lex 由 Mike Lesk 和 Eric Schidt（埃里克-施密特，Google 前 CEO）设计，yacc 则由 Stephen C.Johnson 开发，它们的主页为：

http://flex.sourceforge.net

http://www.gnu.org/software/bison

如果想深入学习 Flex 和 Bison，推荐阅读《flex 与 bison》一书，这是一本非常精彩的书，是经典 O'Reilly 系列书籍《lex & yacc》的续篇。

4.1.准备概念

在阅读后续章节时，最好对以下几个概念有些了解，当然不了解也成，但可能会有些懵懂。本节是全文最晦涩的部分，**可以尝试跳过本节往后看，或者有需要时再回头看看**，但

最终还是需要了解的。《flex 与 bison》一书对编译原理的概念讲得不多，但如果多懂一点，将更有利于学习 flex 与 bison，因此辅以阅读《编译原理》是非常有帮助的，下面介绍的有些概念就摘自《编译原理 第 2 版》一书。

4.1.1. 正则表达式（regex/regexp）

正则表达式（regular expression），这个太重要了，一定要先懂一点，不然后面没法玩。如有需要，可参考百度百科：http://baike.baidu.com/view/94238.htm。

4.1.2. 符号∈

“∈”是数学中的一种符号，读作“属于”。

通常用大写拉丁字母 A，B，C，…表示集合，用小写拉丁字母 a，b，c，…表示集合中的元素。如果 a 是集合 A 的元素，就说 a 属于（belong to）集合 A，记作 a ∈ A。例如，我们用 A 表示“1~20 以内的所有素数”组成的集合，则有 3 ∈ A。

百度百科：http://baike.baidu.com/view/53231.htm。

4.1.3. 终结符/非终结符/产生式

维基百科：http://zh.wikipedia.org/wiki/终结符。

在 C/C++/Java 等语言中的 if-else 语句，通常具有如下的形式：

if (expression) statement **else** statement

即一个 if-else 语句由关键字 **if**、左括号、表达式、右括号、语句、关键字 **else** 和另一个语句连接而成。如果用变量 expr 表示表达式，用变量 stmt 表示语句，那么这个构造规则可表示成：

stmt → **if** (*expr*) *stmt* **else** *stmt*

其中箭头（→）可以读作“可以具有如下形式”，在 **BNF** 中使用双冒号（**::**）来替代箭头，而在 **bison** 规则部分，则使用单冒号（**:**）替代箭头。这样的规则称为**产生式**（production）。在一个产生式中，像关键字 **if**、**else** 和括号这样的词法元素称为**终结符**（terminal），像 *expr* 和 *stmt* 这样的变量称为**非终结符**（nonterminal）。

4.1.4. 记号（Token）

终结符和非终结符，都是 Token。在 flex 和 bison 中，记号由两部分组成：记号编号和记号值，其中不同的记号值可以有不同的类型，具体由 bison 中的“%union”控制。记号的值要存储在全局变量 yyval 中。

记号的编号在 bison 编译时自动按顺连续分配，最小值从 258 开始。之所以从 258 开始，

是因为 258 之前的数值是 ACSII 字符的值。

4.1.5. 形式文法

一个形式文法 **G** 是下述元素构成的一个元组（N, **Σ**, P, S）：

1) 非终结符号集合 N
2) 终结符号集合 Σ（Σ 与 N 无交）
3) 起始符号 S（S ∈ N）
4) 取如下形式的一组产生式规则 P：

(Σ ∪ N)*中的字串 -> (Σ ∪ N)* 中的字串，并且产生式左侧的字串中必须至少包括一个非终结符号。

百度百科：http://baike.baidu.com/view/135643.htm。

- **举例**

考虑如下的文法 G，其中 N = {S, B}， Σ = {a, b, c}，P 包含下述规则：

1) S -> aBSc
2) S -> abc
3) Ba -> aB
4) Bb -> bb

非终结符号 **S** 作为初始符号。下面给出字串推导的例子：（推导使用的产生规则用括号标出，替换的字串用黑体标出）：

S -> (2) abc

S -> (1) aBSc -> (2) aBabcc -> (3) aaBbcc -> (4) aabbcc

S -> (1) aBSc -> (1) aBaBScc -> (2) aBaBabccc -> (3) aaBBabccc -> (3) aaBaBbccc -> (3) aaaBBbccc -> (4) aaaBbbccc -> (4) aaabbbccc

从这可以看出，这个文法定义了语言 { anbncn | n > 0 }，这里 an 表示含有 n 个 a 的字串。

4.1.6. 上下文无关文法（CFG）

一个形式文法 G =（N，Σ，P，S），如果它的产生式规则都取如下的形式：V -> w ，这里 V∈N ，w∈(N∪Σ)*，上下文无关文法取名为“**上下文无关**”的原因就是因为字符 V 总可以被字串 w 自由替换，而无需考虑字符 V 出现的上下文。

一个上下文无关文法（**C**ontext-**F**ree **G**rammar）由四个部分组成：

1) 终结符集合

终结符是文法所定义语言的基本符号的集合。

2) 非终结符集合

每个非终结符表示一个终结符的集合，非终结符给出了语言的层次结构，而这种层次结构是语法分析和翻译的关键，因此规则部分是 **bison** 语法文件的核心部分。

3) 产生式集合

每个产生式包含一个称为产生式头或左侧的非终结符，一个箭头，和一个称为产生式右侧的由终结符、非终结符组成的序列。**上下文无关文法要求产生式左侧只能包含一个非终结符号**。

4) 开始符号

开始符号总是一个非终结符。

- **举例**

可以产生变量 x，y，z 的算术表达式：

S -> T + S | T - S | T

T -> T * T | T / T | (S) | x | y | z

字串“(x + y) * x - z * y / (x + x)”就可以用这个文法来产生。

4.1.7. BNF

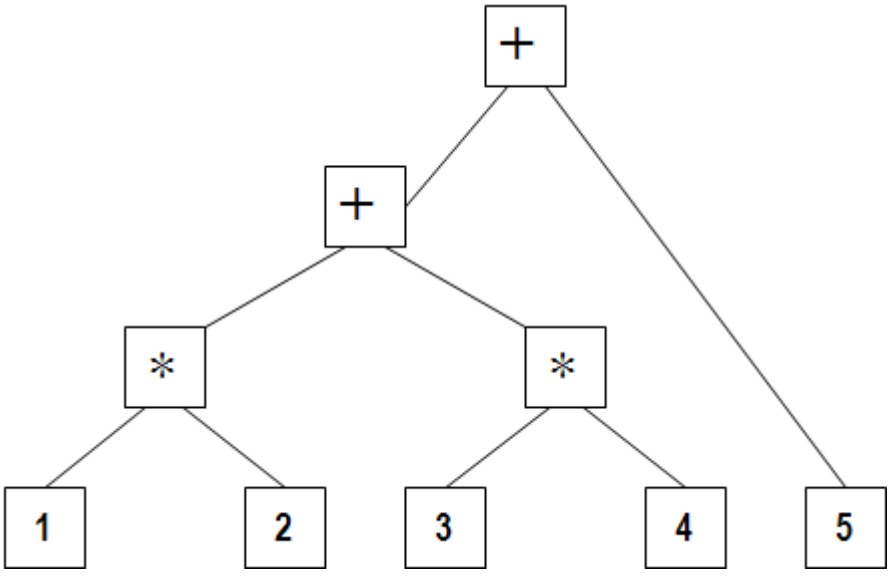
Backus-**N**aur **F**orm（巴科斯范式）的缩写，是由 John Backus 和 Peter Naur 首先引入的用来描述计算机语言语法的符号集，**经常用来表达上下文无关文法**。

百度百科：http://baike.baidu.com/view/1137652.htm。

4.1.8. 推导

4.1.9. 语法树

语法分析器的工作是推导出 Token 之间的关系，语法树经常被用来表达这种关系。算术表达式“1 * 2 + 3 * 4 + 5”的语法树如下图所示：



由于乘法比加法具有更高的优先级，所以前两个表达式为“1 * 2”和“3 * 4”。这颗树的每个分支都显示了 Token 之间或 Token 与下面子树的关系。

4.1.10. LL(k)

一种自顶向下分析的分析方法，LL(1)是 LL(k)的特例，其中的 k 则表示向前看 k 个符号，
百度百科：http://baike.baidu.com/view/1352042.htm。

4.1.11. LR(k)

一种从左至右扫描和自底向上的语法分析方法。
百度百科：http://baike.baidu.com/view/6921179.htm。

4.1.12. LALR(k)

自左向右向前查看一个记号。bison 默认使用 LALR（1）分析方法。
百度文库：http://wenku.baidu.com/view/e8144723bcd126fff7050b99.html。

4.1.13. GLR

通用的自左向右。当 bison 使用 LALR（1）不能用任的时候，可以选择使用 GLR 作为分析方法。

4.1.14. 移进/归约

移进（shift），归约（reduction）

4.2. flex 和 bison 文件格式

不管是 flex 的词法文件，还是 bison 的语法文件，格式均为：

```
。。。定义部分。。。
%%
。。。规则部分。。。
%%
。。。用户子例程部分。。。
```

4.2.1. 定义部分

flex 和 bison 的定义部分都可以包含由“**%{**”和“**%}**”括起来的块，这块的内部按照 C/C++规则来写，通常会有一些“#include”和一些“extern”声明等。除此之外，还可以包含一些在 flex 和 bison 章节介绍到的信息。

4.2.2. 规则部分

在规则部分：对于 flex，主要是定义“模式”和“模式对应的动作”；对于 bison，主要是定义推导规则。在 flex 和 bison 再分开讲解。

不管是 flex 还是 bison，在规则部分都可以添加注释，但两者方式有不同之处：

1) flex

注释不能顶格写，“/*”前至少要有一个空格或 Tab，“*/”可以顶格，还可以与“/*”不在同一行，这段会被照搬到 lex.yy.c 文件中。

在 flex 的规则部分也可以使用双斜杠“//”注释，但只能使用到规则的开始部分，也就是所有的模式之前才可以使用，否则只能使用“/*。。。*/”注释，并且都不能顶格写。

2) bison

注释可以顶格写，而且可以使用双斜杠“//”，这段不会被搬到.tab.c 文件中，而是被忽略掉了。

4.2.3. 用户子例程部分

这部分是按 C/C++规则编写的代码或注释等，经 flex 和 bison 编译后，会被原样搬到相应的**.c** 文件中。

4.3. flex 基础

flex 是一个快速的词法分析（lexical analysis，或简称 scanning）工具。flex 通过分析输入流，得到一个个 Token，如：“flex and bison”被解析成三个 Token：flex、and 和 bison。可以定义不同类型的 Toekn，由 bison 中的“%union”控制。

flex 词法文件名一般习惯以“**.l**”或“**.ll**”结尾，使用 flex 编译“.l”或“.ll”文件后，会生成名称为 **lex.yy.c** 文件，这是默认时候生成的文件名。

Token 实际上为 flex 规则部分定义的“单词”，只是这个“单词”可能是普通意义上的单词，也可能不是，它可能为普通意义上的短语等。

简单的说，flex 的工作就是将输入分解成一个个的 Token，并且在分析出一个 Token 时，可以执行指定的动作，动作以 C/C++代码方式表示，也可以没有任何动作。

4.3.1. flex 文件格式

定义部分可包含：
1) 选项（总是以%option 打头）
2) C/C++代码块，要求使用“%{”和“%}”包含起来
3) 名字定义（类似于 C 中的宏定义）
4) 开始条件
5) 转换
6) 以空格和 Tab 开始的行（这些将被原样的搬到 lex.yy.c 文件中）

%%

词法规则部分包含：
1) 模式行
2) C/C++代码和注释

%%

。。。**用户子例程**部分。。。

4.3.2. 选项

选项要求顶格写，前面不能有任何空格或 Tab，用它来指示 flex 编译行为的，和 GCC 的编译选项有些类似，经常用的选项有：

1) %option **yylineno**
   表示定义一个 int 类型的全局变量 yylineno，其值代表当前的行号。
2) %option **noyywrap**
   表示不需要提供 int yywrap()函数，否则必须显示的实现 yywrap()函数，不然链接时会报找不到 yywrap 符号错误。
3) %option **yywrap**
   这个是默认的，隐式存在的 option。

4.3.3. 名字定义

名字定义类似于 C 语言中的宏定义，目的是方便在词法规则部分引用，格式为：

NAME definition

名字 NAME 可以包含字母、数字、连字符“-”和下划线，且不能以数字打头。definition 为正则表达式。

在词法部分需要引用它时，需要使用**花括号“{}”**括起来，如：{NAME}，NAME 会在词法规则部分被展开成由一对圆括号括住的该名字的定义，即{NAME}展开成(definition)。

假如有两个名字定义：DIG [0-9]和 CHR [a-zA-Z]，则**{**DIG**}{**CHR**}**等价于**(**[0-9]**)(**[a-zA-Z]**)**。

4.3.4. 词法规则

1) 模式行

模式行包含一个模式、一些空白字符、以入模式匹配时执行的 C/C++代码，如果 C/C++代码超过一条语句或跨越多行，则必须用“{ }”或“%{ %}”包含起来。

2) C/C++代码和注释

这里又包含两种情况：

a. 以空格或 Tab 打头的行

b. 处于“%{”和“%}”之间的内容

它们都会被原样的搬到 yylex()函数中。而位于模式行之前的，则会被搬到 yylex()函数的开头。

4.3.5. 匹配规则

当 flex 词法分析器运行时，它根据词法规则部分定义的模式进行匹配，每发现一个匹配（匹配的输入称为记号 Token）时，就执行这个模式所关联的 C/C++代码。

如果模式后面紧跟一个竖线“|”，而不是 C/C++代码，则该模式使用下一个模式相同的 C/C++代码。

如果输入字符或字符串无法匹配任何模式，则认为它匹配了代码为 ECHO 的模式，该记号会被输出。

如果模式后什么也没有，则相当于“{ }”，也就是空动作。

格式	示例	备注
Pattern C-statement	[,] return ','; 	一条语句时，可以不用“{ }”
Pattern { C-statement }	{int16} { yylval.sval = strdup(yytext); return int16; } 	超过一条语句必须使用“{ }”

Pattern {
C-statement
}
	{int16} {
	yylval.sval = strdup(yytext);
	return int16;
	}
	
		“{”必须和模式处于同一行，其它则无约束

Pattern | Pattern C-statement	[,] | [;] { printf("%s", yytext); } 	相当于： [,] { printf("%s", yytext); } [;] { printf("%s", yytext); }
Pattern	[,] {} 	动作部分为空
Pattern	[,] 	相当于： [,] {}

除了上面列出的外，还可以使用“%{ %}”替代“{ }”。

4.3.6. %option

flex 提供了几百个选项，用以控制编译词法分析器的行为。大多数选项可写成“**%option** name”的形式，如果需要关闭一个选项，只需要将 name 换成 noname 即可。flex 自带文档中的“Index of Scanner Options”一节列出了所有的选项，也可以访问网址：

http://flex.sourceforge.net/manual/Index-of-Scanner-Options.html

下表为部分经常使用到的选项：

选项	作用
%option **yylineno**	启用内置的全局变量 yylineno，该变量记录了行号
%option **noyywrap**	不使用 yywrap()函数

4.3.7. 全局变量 yytext

flex 使用 yytext 来存放当前记号，它总是一个字符串类型。

4.3.8. 全局变量 yyval

用来保存 Token 的值，通常为一个 union，以支持不同的 Token 类型，它和 yytext 紧密联系，但两者是有区别的，通过后面的实例即可看出。

4.3.9. 全局变量 yyleng

flex 使用 yyleng 存放 yytext 的长度，因为 yyleng 的值和 strlen(yytext)相等。

4.3.10. 全局函数 yylex

yylex()词法分析的入口函数，通常无参数，它借助全局变量和函数与其它部分交互。bison 的 yyparse()函数调用 yylex()来做词法分析，如果不使用 flex，则可自定义一个 yylex()函数。

4.3.11. 全局函数 yywrap

yywrap()是一个回调函数，由选项来控制是否需要它。当 flex 词法分析器到达文件尾时，可选择调用 yywrap()来决定下一步操作。

如果 yywrap()返回 0，将继续分析；如果返回 1，则返回一个 0 记号来表示文件结束。

如果不使用 yywrap()，选项“%option noyywrap”可以用来屏蔽对 yywrap()的调用。

当需要处理多个文件时，这个函数就可以派上用场了。

4.4.bison 基础

bison 是一个语法分析（syntax analysis，或简称为 parsing）工具。

bison 词法文件名一般习惯以“**.y**”或“**.yy**”结尾，使用 bison 编译“.y”或“.yy”文件后，会生成带后缀“.tab.c”文件。如果带参数“**-d**”，则还会生成后缀为“**.tab.h**”的头文件，一般这个头文件供 flex 词法文件使用，因为其中定义了 yyval 和 Token 的编号等。假设 bison 语法文件名为“x.y”，则使用“bison x.y”编译后，会生成文件 x.tab.c；如果使用“bison -d x.y”编译，则会生成 x.tab.c 和 x.tab.h 两个文件。

简单的说，bison 的工作就是推导出 Token 之间的关系，每推导出一个关系后，都可执行指定的动作，动作以 C/C++代码方式表示，也可以没有任何动作。。

4.4.1. bison 文件格式

定义部分可包含：

1) C/C++代码块，要求使用“%{”和“%}”包含起来
2) 各种声明，包括：
   a) %union
   b) %token
   c) %type
   d) %left
   e) %right
   f) %nonassoc

%%

。。。**语法规则**部分。。。

%%

。。。**用户子例程**部分。。。

4.4.2. %union

如果有不同类型的 Token，只需要使用“**%union**”，假如有整数和字符串两种类型的 Token，则：

```
%union
{
    int text;
    char* ival;
};
```

经过 bison 编译后，“**%union**”会被翻译成 **C 语言的 union**，如：

```
typedef union YYSTYPTE {
    int ival;
    char* sval;
} YYSTYPE;
```

在编译后生成的“**.tab.c**”文件中，即可看到该 **union** 的定义。

4.4.3. %token

定义记号。只有定义后，“bison -d”才会在“.tab.h”文件中定义记号的编号，flex 词法文件会用到记号的编号，如果漏定义，则会报找不到编译错误。bison 用这些记号的编号做状态跳转。

4.4.4. 全局函数 yyerror()

yyerror()是一个回调函数，原型为：

```
void yyerror(const char* s)
```

当 bison 语法分析器检测到语法错误时，通过回调 yyerror()来报告错误，参数为描述错误的字符串。

4.4.5. 全局函数 yyparse()

bison 生成的请求分析器的入口函数就是 yyparse()。当程序调用 yyparse()时，语法分析器将试图分析输入流或指定的 Buffer，如果分析成功则返回 0，否则返回非 0 值。和 yylex()一样，该函数通常不带任何参数。

4.5. 例 1：单词计数

4.5.1. 目的

单词计数的实现只需要用到 flex，不需要 bison，实现非常简单，适合第一次接触。编程语言喜欢以“Hello World”作为入门，而编译领域喜欢以“单词计数”和“算术表达式”作为入门学习。

4.5.2. flex 词法文件 wc.l

```
 // 使用内置的行号全局变量 yylineno（请注意注释不能顶格写，需以空格或 Tab 打头）
%option yylineno
 // 不使用 yywrap()函数
%option noyywrap
%{
// 这个区域是纯粹的 C/C++代码（也因此，注释顶格写是可以的）

#include <stdio.h>   // 需要用到 printf 函数

static int num_chars = 0;     // 记录字符个数
static int num_words = 0;   // 记录单词个数

%}

%%
 // 这里是规则定义区间，注释不有顶格写
[a-zA-Z]      { ++ num_chars;   /* 花括号间是 C/C++代码域，是模式对应的动作代码 */ }
[a-zA-Z]+     { ++ num_words ; }

%%

int main()
{
      printf("chars: %d, words=%d, lines=%d\n", num_chars, num_words, yylineno);
      return 0;
}
```

4.5.3. Makefile

```
wc: wc.l wc.y
          flex wc.l
          g++ -g -o wc lex.yy.c

clean:
          rm -f wc
```

使用 flex 编译 wc.l 后，会生成 lex.yy.c 文件，**wc** 即是单词计数程序，可以这样使用：**./wc < 被统计的文件，如：./wc < wc.l**。

4.6. 例 2：表达式

4.6.1. 目的

介绍一个可以内嵌到程序中使用的表达式实现。

4.6.2. flex 词法 exp.l

```
 // 从这里开始，是词法文件的定义部分，请注意这里的注释不能顶格写
%option noyywrap
%{
#include "exp.tab.h"
#include <stdio.h>
%}

 // 名字定义，类型 C 语言中的宏定义，number 相当于宏名，“[0-9]+”相当于宏的内容
number [0-9]+

%%

{number} { yylval = atoi(yytext); return NUMBER; }
"+"          { return ADD; }
"-"          { return SUB; }
"*"          { return MUL; }
"/"          { return DIV; }
[\n]         { return EOL; }
[ \t]        { }
.            { printf("invalid: %s\n", yytext); }

%%
```

4.6.3. bison 语法 exp.y

```
%{

#include <stdio.h>

extern char* yytext;
```

```
extern int yylex(void);
extern void yyerror(const char* s);

%}

%token NUMBER
%left ADD SUB MUL DIV
%token EOL

%%

explist:
        | explist exp EOL
        {
                // $2 代表 exp 的值
                printf("= %d\n", $2);
        }
        ;

// 这里需要考虑优先级问题，加减法的优先级低于乘除法，所以分成两步推导方式
exp: factor
        | exp ADD factor
        {
                // $$代表“exp: factor”中的 exp
                // $1 代表“exp ADD factor”中的 exp 的值
                // $3 代表“exp ADD factor”中的 factor 的值
                // $2 代表 ADD
                $$ = $1 + $3;
        }
        | exp SUB factor
        {
                $$ = $1 - $3;
        }
        ;

factor: NUMBER
        | factor MUL NUMBER { $$ = $1 * $3; }
        | factor DIV NUMBER { $$ = $1 / $3; }
        ;

%%

void yyerror(const char* s)
{
```

```
        printf("Error: %s: %s\n", s, yytext);
}

int main(int argc, char* argv[])
{
        // yyparse 是语法分析函数，它内部会调用 yylex()进行词法分析
        yyparse();
        return 0;
}
```

4.6.4. Makefile

```
exp: exp.l exp.y
        flex exp.l
        bison -d exp.y
        g++ -g -o exp lex.yy.c exp.tab.c -I.

clean:
        rm -f exp
```

Make 成功后，会在当前目录下生成一个可执行文件 **exp**，然后运行 **echo "1+1*2" | ./exp** 即可查看到效果，也可以交互式方式运行**./exp**。

4.6.5. 代码集成

上述的实现，是从标准输入读入需要计算的表达式，但要嵌入到程序中使用，则需要支持从指定的字符串中读入需要计算的表达式，flex 对这个提供了很好的支持，在 lex.yy.c 中有三个函数可以使用：

YY_BUFFER_STATE **yy_scan_buffer** (char *base,yy_size_t size);
YY_BUFFER_STATE **yy_scan_string** (yyconst char *yy_str);
YY_BUFFER_STATE **yy_scan_bytes** (yyconst char *bytes,int len);

其中 YY_BUFFER_STATE 实际为：typedef struct yy_buffer_state *YY_BUFFER_STATE;。这个时候，只需要将 main 函数改成如下实现：

```
// 请注意，需要加上的声明 yy_scan_string，否则会报该函数未声明，
// 除了将声明放在 main()函数前外，还可以将声明放在 exp.y 文件头部。
extern struct yy_buffer_state* yy_scan_string(const char* s);
int main(int argc, char* argv[])
{
        // 如果没有带参数，则仍从标准输入 stdin 读入表达式
        if (argc > 1)
        {
                std::string exp = argv[1];
```

```
                // c.y 中定义的语法要求以换行符结尾，
                // 如果不想有 append 这一句，则需要去掉“explist exp EOL”中的“EOL”，
                // 并且不能使用 flex 选项 “%option noyywrap”，这时可显示定义
                // yywrap()函数，并让它返回 1
                exp.append("\n");
                yy_scan_string(exp.c_str());   // 使用 exp 替代 stdin 作为输入
        }

        yyparse();
        return 0;
}
```

通过这个改进，就不难知道，如何实现一个可集成到程序中的表达式解析器了。

4.7.例 3：函数

4.7.1. 目的

介绍如何编译 IDL 函数，但在这里不会真正去实现一个 RPC 函数，因为那会让问题变得复杂起来。但有了语法树后，也就有了函数的描述信息，在此基础上实现 RPC 函数就有眉目了。

4.7.2. func.h

```
// 这里定义了语法树，
// 这里语法树只有三个结点：
// 1) 根结点 g_func_set
// 2) 第一级子结点 Function
// 3) 第二级子结点 Parameter，第二级子结点可能为空，因为函数不一定要有参数
#ifndef _FUNC_H
#define _FUNC_H

#include <stdio.h>
#include <map>
#include <string>
#include <vector>
using std::string;
using std::vector;

struct Parameter   // 函数参数定义
```

```
{
        string name;    // 函数参数的名称
        string type;    // 函数参数的类型
        void clear()    // 重置，以便用于下一个参数
        {
                name.clear();
                type.clear();
        }
};

struct Function   // 函数定义，根据它就可以将 IDL 中定义的函数翻译成不同的语言版本
{
        string name;                    // 函数名
        string type;                    // 函数的返回类型
        vector<Parameter> params;       // 函数的参数列表
        void clear()                    // 重置，以便用于下一个函数
        {
                name.clear();
                type.clear();
                params.clear();
        }
};

extern vector<Function> g_func_set;         // 根结点，保存所有的函数

#endif // _FUNC_H
```

4.7.3. func.c

```
#include "func.h"
#include <iostream>
using std::cout;
using std::endl;

vector<Function> g_func_set;

extern int yyparse();       // 实现在 x.tab.c 文件中
int main(int argc, char* argv[])
{
        yyparse();      // 调用语法分析函数，它内部会调用词法分析函数 yylex()

        // 遍历所有的函数，
        // g_func_set.size()的值为函数的个数
```

```
        for (vector<Function>::size_type i=0; i<g_func_set.size(); ++i)
        {
                const Function& func = g_func_set[i];
                cout << "function name: " << func.name << endl;        // 输出函数名
                cout << "return type: " << func.type << endl;          // 输出函数返回类型

                cout << "parameters:" << endl;     // 提示输出参数信息
                if (func.params.empty())
                {
                        cout << "\tempty" << endl;    // 如果函数没有参数，则输入 empty
                }
                else
                {
                        // 遍历函数的所有参数
                        for (vector<Parameter>::size_type j=0; j<func.params.size(); ++j)
                        {
                                const Parameter& param = func.params[j];
                                cout << "\tname: " << param.name << ", type: " <<
param.type << endl;
                        }
                }
                cout << endl;
        }

        return 0;
}
```

4.7.4. IDL 代码 func.idl

```
int32 foo();
int16 zoo(int16 zz);
int16 zoo(int16 zz, int32 yy);
```

请注意：为了简化词法和语法，不支持注释等，仅支持不带引用和指针类型参数、返回值的函数，否则会增加词法和语法的复杂度，不利于初次学习。当你掌握之后，可尝试加入注释等。当然仍可以在 idl 写入其它的内容，但会报编译错误。

4.7.5. flex 词法 func.l

```
 // 从这里开始，是词法文件的定义部分，请注意这里的注释不能顶格写
%option yylineno
```

```
%{
// 这区间完全是 C/C++代码，所以按照 C/C++风格来写即可，
// 经 flex 编译后，会被搬到 lex.yy.c 文件中

#include "func.h"              // 要用到 func.h 中定义的 g_lineno
#include "func.tab.h"          // 要用到 func.tab.h 中定义的 token 和 yylval

%}

 /* 名字定义，有点类似于 C 语言中的宏，但在使用的时候需要使用花括号“{}”括起来，
    如：chars 类似于宏名，[a-zA-Z]类似于宏的值
 */
chars               [a-zA-Z]
number              [0-9]
int16               (int16)
int32               (int32)
xname               ({chars})+(\_|{chars}|{number})*

 // 从这里开始，是词法的文件的规则部分
%%
 // 模式和该模式的动作，动作为 C/C++代码
 // 在这里使用了前面定义的名字，注意要用花括号“{}”括起来
 // 返回的 Token 值，要求在 func.y 文件中定义，或自己显示的定义出来。
 // 动作部分为花括号{}括起来的 C/C++代码，因此可以使用 C/C++规则。
{int16}                 { yylval.sval = strdup(yytext); return int16; }
{int32}                 { yylval.sval = strdup(yytext); return int32; }
{xname}                 { yylval.sval = strdup(yytext); return xname; }

[\(]        { return '('; }
[\)]        { return ')'; }
[,]         { return ','; }
[;]         { return ';'; }
[\t]        { /* 什么也不做，花括号都可以省掉，如下就省掉了花括号 */ }
[ ]
[\n]
.           { return ERROR;   /* 非法字符 */ }

 /* 从这里开始，是词法的文件的用户子例程部分（这里不能用双斜杠注释） */
%%
// 从这开始，又是全 C/C++代码区间了，将完全被搬到 func.tab.c 文件的最尾部，
// 包括这个注释

int yywrap()   // 这个函数非必须的，可以使用%option noyywrap 表示不用
{
```

```
        return 1;
}
```

4.7.6. bison 语法 func.y

```
%{
// 这区间完全是 C/C++代码，所以按照 C/C++风格来写即可，
// 这个区间内的代码，会原原本本搬到 func.tab.c 文件的顶部（但不是最顶部）

#include "func.h"
#include <stdio.h>
#include <iostream>

extern int yylineno;      // flex 编译 func.l 后，定义在 lex.yy.c 中
extern char* yytext;      // 在编译 func.l 后生成的 lex.yy.c 中定义
extern void yyerror(const char *s);      // 下面的代码将使用到，所以需要事先声明一下
extern int yylex(void);   // 在编译 func.l 后生成的 lex.yy.c 中定义，用来做词法分析

static string g_type;                    // 用来存储当前正解析到的函数返回类型或参数类型
static struct Parameter s_param;    // 用来存储当前正解析到的参数信息
static struct Function s_func;      // 用来存储当前正解析到的函数信息

%}

// 当 Token 存在不同类型时，%union 就可派上用场了
// 使用 bison 编译后，会变成：
// typedef union YYSTYPE {
//      int ival;
//      char* sval;
// } YYSTYPE;
// 此外，还会定义全局变量 yylval：
// extern YYSTYPE yylval;
// 这样，在 flex 文件（本例为 func.l）即可使用 yylval 了。
%union
{
        int ival;       // 存储整数类型的 Token
        char* sval;     // 存储字符串类型的 Token，本示例只有它，无整数类型的 Token
};

// 所有的 Token，都必须在这里声明
// 并且在 bison 编译后，会变成：
// #ifndef YYTOKENTYPE
// # define YYTOKENTYPE
```

```
// enum yytokentype {
//         int16 = 258,
//         int32 = 259,
//         xname = 260,
//         ERROR = 261
// };
// #endif
// Token 的值总是从 258 开始，以避免与 a、b、c 等字符值产生冲突。
// 同时，还会定义等值的宏：
// #define int16 258
// #define int32 259
// #define xname 260
// #define ERROR 261
%token <sval> int16
%token <sval> int32
%token <sval> xname        // 函数名、参数名等，如果规则不一样，可取不同 name
%token <sval> ERROR        // 其它的非法输入

%%
// 函数集，用以支持多个函数，但是也可以只有一个函数，所以有两个规则
function_set: function_prototype
            | function_set function_prototype
            ;

// 函数原型，函数可以带参数，也可以无参数，
// 花括号{}内的是 C/C++代码，因此适用 C/C++规则。
function_prototype:    return_type xname '(' ')' ';'
            {
                    // 无参数
                    s_func.name = $2;                    // 保存函数名
                    g_func_set.push_back(s_func);        // 当前函数完整了
                    s_func.clear();                      // 以便 s_func.用于下一个函数
            }
            | return_type xname '(' parameter_list ')' ';'
            {
                    // 有参数
                    s_func.name = $2;
                    g_func_set.push_back(s_func);
                    s_func.clear();
            }
            ;

// 可以只有一个参数，也可以是多个参数
parameter_list: parameter_define
```

```
        | parameter_list ',' parameter_define
        ;

// 参数的定义
parameter_define: parameter_type xname
        {
                s_param.name = $2;
                s_func.params.push_back(s_param);
                s_param.clear();
        }
        ;

// 参数类型
parameter_type: data_type
        {
                // 记下参数类型
                s_param.type = g_type;
        }
        ;

// 函数返回值类型
return_type: data_type
        {
                // 记下函数返回类型
                s_func.type = g_type;
        }
        ;

// 支持的基本数据类型，可自行扩展
// 需要分成 parameter_type 和 return_type 两个是因为需要给 s_param 和 s_func 区开赋值
data_type: int16 { g_type = $1; }
        | int32
        {
                // 类型的名称用 g_type 存储起来
                g_type = $1;
        }
        ;

%%
// 从这开始，又是全 C/C++代码区间了，将完全被搬到 func.tab.c 文件的最尾部，
// 包括这个注释

// 遇到错误就在屏幕上打印出来
void yyerror(const char* s)
```

```
{
        printf("Error(%d): %s: %s\n", yylineno, s, yytext);
}
```

4.7.7. Makefile

```
func: func.l func.y func.h func.c
        flex func.l          # 编译词法文件
        bison -d func.y      # 编译语法文件，这里将会产生 func.tab.h 和 func.tab.c 两个文件
        g++ -g -o func lex.yy.c func.tab.c func.c -I.      # 生成 IDL 编译器 func

clean:
        rm -f func
```

Make 成功后，会在当前目录下生成一个可执行文件 **func**，然后运行**./func < func.idl** 即可查看到效果。

5. 进阶

掌握以上基础后，就具备了实现 RPC 的能力。在上一节中的“函数”实现过去简单，还不能直观的理解 RPC 函数是如何调用和被调用的，这一节就要解决这个问题。为了降低复杂度，这里采用伪代码方式。

从前面的讲解可以知道，RPC 函数的实现是分客户端和服务端的，这里的就分别讲解它们是如何实现的。

5.1.客户端函数实现

客户端函数完全由 IDL 编译器实现，用伪代码表示，实现如下：

```
void Proxy::foo(Callback* cb_func, int m, int n)
{
    string    bytes_stream;

    encode(&bytes_stream, "foo");    // 将函数名编码到字符流
    encode(&bytes_stream, m);        // 将参数 m 编码到字符流
    encode(&bytes_stream, n);        // 将参数 n 编码到字符流

    // 当 RPC 因网络超时或网络故障失败时，异步回调 cb_func;
    // 或者 RPC 请求收到响应时，异步回调 cb_func,
    // 实际中也可以同步调用，但性能低
    // RpcSession 应当是基于 Socket 的，它能容忍网络连接断开，执行自动重连重试，
```

```
    // 它需要维护一个 RPC 调用的状态和上下文关系。
    RpcSession session = get_rcp_session(cb_func);
    // 通过 send 将编码后的字节流发送到服务端
    session->send(bytes_stream.data(), bytes_stream.size());
}
```

从上面的实现可以看出，要求 RPC 函数的参数必须是可编码类型的，否则无法将它序列化到字节流中，返回类型同样要求是可编码的。如果不想自己实现一套编解码函数，可以直接使用 ProtoBuf 来做这件事。

5.2.服务端函数实现

服务端函数的实现需要分成两部分：一是 Stub 部分的实现，另一部分是用户代码自己的实现。

5.2.1. Stub 部分实现

NetThread 表示网络线程，用于接收网络数据。

```
void NetThread::run()
{
    while (true)
    {
        string    bytes_stream;
        socket->receive(bytes_stream, nums_bytes);

        // 下面这部分代码都是 IDL 编译器依据 IDL 文件生成的
        string func_name;
        decode(&bytes_stream, &func_name);   // 解码出函数名

        // 根据不同函数名做不同调用
        switch (func_name)
        {
            case foo:
            {
                typedef void (*FOO)(int, int);
                FOO    foo = (FOO)map[func_name];

                int m = 0;
                int n = 0;
                decode(&bytes_stream, &m);   // 解码出参数 m
                decode(&bytes_stream, &n);   // 解码出参数 n
                (*foo)(m, n);   // 调用服务端的 foo()实现
```

```
                    break;
                }
                case woo:
                {
                    。。。。
                }
            }
        }
}
```

从实现的效率出发，一般不直接对函数名编码，而是对函数编号，这样一个下标即可确定是对哪个函数的调用，性能会高出很多。

5.2.2. 用户部分实现

```
void Server::foo(int m, int n)
{
    // Do something or response
}
```

这部分代码是由用户实现的，而非 IDL 编译器生成，如果只是 C 代码，则可采用函数指针方式，如果是 C++代码，则可以使用接口。

6. 参考资料

《百度百科及其它网上资料》
《flex 与 bison》/（美）John Levine 著；陆军译；东南大学出版社出版
《编译原理第 2 版》/（美）Alfred V.Aho Monica S.Lam Ravi Sethi Jeffrey D.Ullman 著；
赵建华 郑滔 戴新宇 译
机械工业出版社出版