代码自动插装技术的研究与实现

晏 华* 袁海东 尹立孟

(电子科技大学计算机科学与工程学院 成都 610054) (北京科银京成技术有限公司成都研发中心 成都 610051)

【摘要】介绍了代码插装技术的应用背景，比较了手工和自动两种代码插装方式，重点分析和比较了代码自动插装在编译过程中各阶段实现的可行性和思路，提出在编译预处理和编译阶段之间增加一个CPU无关的编译预处理文件，且具有代码自动插装功能的语法词法分析阶段的一种最佳实现方案，给出了一个简单原型的实现。

关 键 词 覆盖测试; 代码插装; 编译; 词法分析; 语法分析

中图分类号 TP314

软件测试是保证软件可靠性的重要手段之一，其方法和技术多种多样，但最基本、最直观、最有效的测试方法是覆盖测试[1, 2]。

覆盖测试也称白盒测试或路径测试，是一种动态的软件测试方法，即使程序在控制下运行，也能通过运行结果观测程序运行行为，从而发现错误。通常覆盖测试的过程分为三步：

1) 对被测程序的源代码进行插装，得到语义/功能上等效的源代码；

2) 对插装后的代码编译，并与包含覆盖信息采集功能函数的运行库链接，获得可执行文件；

3) 运行程序，获得覆盖信息，通过覆盖测试软件分析输出用户所需的测试和统计结果。

可见，代码插装是实现覆盖测试的关键步骤，本文研究在程序的编译阶段识别出插装点，从而完成代码的自动插装。

1 代码插装方式比较

覆盖测试软件对被测程序的源代码进行插装的方式有手工方式和自动方式。手工方式是覆盖测试软件提供一组完成覆盖信息采集功能的函数，用户手工在源程序需要获取程序运行信息的地方写入信息采集代码段(代码段可以是覆盖信息采集函数调用，也可以是简单的赋值语句)完成代码插装工作，然后编译链接程序时需将包含覆盖信息采集功能函数的库链接进来。在自动方式下，用户只需使用覆盖测试软件提供的接口进行一个简单的设置，覆盖测试软件即可在编译链接时自动插入信息采集代码段，并自动链接相关库函数，生成可执行文件。采用手工方式的优点在于用户可只采集自己关心的一些信息，而无需在一大堆信息中提取所需信息，比较灵活，而且无需对编译器或编译的过程进行某些修改，即可做到编译器无关；采用手工方式的缺点在于破坏源程序，在不进行测试时，需要删除插入的代码或使用宏定义方式屏蔽插入的代码，比较繁琐和低效。

采用自动方式的优点在于代码插入可针对编译链接的中间文件进行，不会破坏源程序。此外用户能通过软件提供的接口(如设定信息采集级别等选项)来控制插装代码的类型、位置和多少，插入过程、编译链接过程均无需用户介入，测试过程比较简单；采用自动方式的缺点是代码插装实现与编译器或编译过程相关，而且实现难度较大。可见，代码自动插装技术是实现覆盖测试软件简单易用目标的关键。

2001年9月3日收稿

* 女 31岁 硕士 讲师

2 代码自动插装技术的研究与实现

2.1 代码插装切入点分析与比较

通过对GCC源码的分析以及编译过程的调试跟踪，从C语言源程序到目标代码经历的阶段如图1所示。

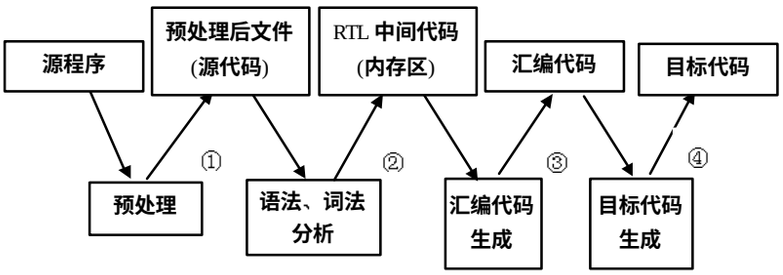

图 1　GCC 编译过程

1) 在第①阶段进行代码插装。预处理阶段通常做宏替换处理。如果源文件中包含宏定义，特别是一些具有分支路径的宏定义，例如下面的程序段：

```
#define abc(i) if (i <=10) {return 10;}    else {return 100;}
void main()
{
    int j;
    j=abc(99);
}
```

通常的情形是将代码行abc(99)简单地替换成if(99<=10){return 10;}else {return 100;}而无法识别出其中的分支路径，因此此阶段的插装只适于不含有分支路径宏定义的源代码插装。

2) 以第②阶段作为代码插装的切入点。此阶段的输入为宏替换已经完成的预处理后的文件，输出为RTL(Register Transfer Language)语言形式描述的中间代码，语法词法分析器能够识别程序的所有特征，因此以这个阶段作为代码插装的切入点是完全可行的。事实上GCC在实现代码插装功能时插装点的识别就是此阶段进行的，GCC将生成的中间代码表达式以一种双向链表的形式组织起来，在双向链表中有一些特殊的结点，这些结点中记录了程序结构信息，例如有记录函数的起始位置和结束位置的结点，有记录循环起始位置和结束位置的结点，有记录程序跳转位置的结点，还有记录基本程序块的起始位置和结束位置的结点等。GCC在利用中间代码生成汇编代码时，如果扫描到这些特殊结点就会根据用户的需求适当地插入一些完成信息采集功能的汇编代码行，从而实现代码插装。因此GCC的代码插装工作实际上是第②阶段和第③阶段共同完成。这种实现代码插装方式的优点是充分利用了编译器在语法词法分析过程中已经获得的程序结构信息，这种方式的缺点也很明显，即代码插装和编译器结合很紧密，并且在第③阶段的汇编代码的生成过程中需要针对不同的CPU生成不同的汇编代码，CPU关联性强，不便于移植。

下面进一步考虑将代码插装这一部分独立出来，使插装后的代码可作为不同编译器(不同的编译器支持同一种语言规范)的输入。插装点识别过程中的语法词法分析只需识别一些有限的程序结构特征即可，而中间代码代码生成过程中所做的语法词法分析最终是为生成汇编代码做准备，需要识别出程序的所有结构特征，因此用于插装点识别的语法词法分析器可以是一个比中间代码生成的语法词法分析器简单的语法词法分析程序，可以重新编写一个满足代码插装要求的语法词法分析程序，该程序的输入为预处理后的源代码文件，输出是插装了信息采集代码段的源代码文件，其处理过程插入到图1的第①和第②阶段之间，代码插装技术实现思路的编译过程如图2所示：

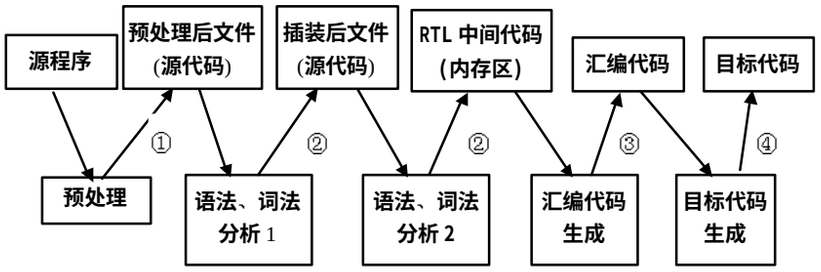

图 2 增加插装阶段的编译过程

采用这种方式最明显的一个优点是代码插装过程与具体编译器代码编译过程无关，插装后的代码可作为多种编译器的输入，缺点是需要重新编写一个满足插装要求的语法词法分析程序。事实上已有商业软件采用这种方法。例如，美国AMC**公司的高性能测试工具**CodeTest**的专利插装技术即是针对预处理后的文件进行赋值语句插装，插装完成后再编译链接生成可运行代码。**

在第④阶段进行代码插装，理论上也是可行的，但此阶段的工作与CPU**类型关联性同样很强，进行代码插装需要针对**CPU**类型进行，不具有通用性。**

综上所述，从代码插装的通用性考虑，图2**所示的代码插装技术实现思路是一种最佳方案。**

2.2 代码自动插装的一种实现方法

实现上述代码插装方案的重点是根据插装需求编写一个语法词法分析器，该分析器需要完成对源程序的扫描，输出插装代码后的源程序。分析器对源程序的分析扫描过程包含词法分析和语法分析。词法分析负责将源程序中的若干字符划分为若干记号，语法分析负责从若干记号中完成程序结构分析，识别出函数体、语句、表达式、关键字、程序分支等[3, 4]**。如果仅学习编译原理自行编写这两个程序，则需要耗费很大的精力，可以利用**UNIX**系统中的两个实用工具**Lex & Yacc**来生成语法和词法分析程序**[5]**。**

2.2.1 Lex & Yacc

Lex**是词法分析程序的生成程序，它接收一个面向问题的用于字符匹配检查的高级说明文件，并产生出用通用语言书写的能识别正则表达式的程序。**Yacc**是编译程序的编译程序，它的主要用途是生成语法分析程序，**Lex & Yacc**的工作原理如图**3**所示。从图**3**可见，语法词法分析程序的编写已转化为词法和语法规则说明文件的编写。**

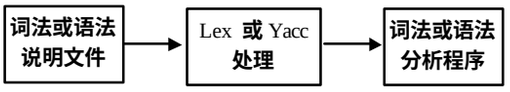

图 3 Lex & Yacc 工作原理图

在词法说明文件中定义输入文件中需要识别的符号并规定符号识别出来后所需做的处理。词法说明文件的核心部分是规则段，由多条规则组成，以一张表的形式存在，表的左列是正则表达式，右列是相应的动作，即识别出左边的正则表达式后要执行的程序片段。

在语法说明文件中定义所支持的语法结构规则以及识别出相应的语法结构后所需要做的处理。语法说明文件的核心是规则段，由一条或多条文法规则构成。规则段定义输入流应满足的语法规则以及相应的动作程序。

2.2.2 一个简单原型的实现

设计目标：在IF ELSE**分支语句中插入代码**enter_ifbranch(),enter_elsebranch()。

相关的词法说明规则如下：

```
"if"        {   return(IF); }
```

```
"else"      {    return(ELSE); }
```

上述两条规则表明当词法扫描程序识别到关键字"if"**和**"else"**时，返回语法扫描程序能够识别的符号。**

相关的语法说明规则如下：

```
if_prefix
        : IF '(' expression ')' {ifbranch_mark=1;}
else_prefix
        : ELSE {elsebranch_mark=1;}
selection_statement
                : if_prefix statement
                | if_prefix statement else_prefix statement
                | SWITCH '(' expression ')' statement
                ;
statement
                : labeled_statement
                | compound_statement
                | expression_statement
                | selection_statement
                | iteration_statement
                | jump_statement
compound_statement
                : '{'
                {
if(ifbranch_mark==1) {fputs("enter_ifbranch();\n",yyout);ifbranch_mark=0;}
if(elsebranch_mark==1) {fputs("enter_elsebranch();\n",yyout);elsebranch_mark=0;}
    }
                    '}'
                | '{'
                {
if(ifbranch_mark==1) {fputs("enter_ifbranch();\n",yyout);ifbranch_mark=0;}
if(elsebranch_mark==1) {fputs("enter_elsebranch();\n",yyout);elsebranch_mark=0;}
                    }
                      statement_list '}'
                | '{' declaration_list
                {
if(ifbranch_mark==1) {fputs("enter_ifbranch();\n",yyout);ifbranch_mark=0;}
if(elsebranch_mark==1) {fputs("enter_elsebranch();\n",yyout);elsebranch_mark=0;}
    }
    '}'
                | '{' declaration_list
                {
if(ifbranch_mark==1) {fputs("enter_ifbranch();\n",yyout);ifbranch_mark=0;}
```

```
if(elsebranch_mark==1) {fputs("enter_elsebranch();\n",yyout);elsebranch_mark=0;}
    }
    statement_list '}'
                ;
```

第一条和第二条语法规则定义了if语句和else语句的表示形式，并且规定当识别到满足规则的语句时置标志符；第三条规则定义了分支语句的三种形式；第四条规则定义了C语言所规定的语句形式；第五条规则定义了复合语句的四种形式。在每种形式的适当位置可添加相应的动作即可实现代码的插装。

3 结 束 语

在实际应用中覆盖测试分析采用测量方法的多样性决定了代码插装需要识别程序结构特征的复杂性。本文对简单原型的实现仅仅是对编译预处理文件进行代码自动插装方案的可行性的验证，对代码插装程序的实现进行了有益的探索，还需要对覆盖测试分析理论以及编程语言语法进一步的学习和研究，才能实现满足实际需要的代码插装程序。

参 考 文 献

1 雷 航，熊光泽，刘锦德. 实时多任务系统的超时故障分析. 电子科技大学学报, 1997, 26(3): 273-278

2 罗 蕾，熊光泽. 实时多任务应用最坏情况设计的研究. 电子科技大学学报, 1997, 26(1): 74-77

3 Kenneth C.Louden著. 编译原理及实践. 冯博琴，冯 岚译. 北京：机械工业出版社，2000: 1-427

4 龚天富，侯文永. 程序设计语言与编译. 北京：电子工业出版社，1997: 1-242

5 施渝萍. 软件开发环境. 成都：电子科技大学出版社，1994: 1-328

The Research & Implementation of Code Automatic Instrumentation Technology

Yan Hua Yuan Haidong

(College of Computer Science and Engineering, UEST of China Chengdu 610054)

Yin Limeng

(Coretek Systems Inc. Chengdu 610051)

Abstract This paper introduces the background of code instrumentation technology and gives a brief comparison between manual and automatic instrumentation at first. Then the paper puts the focus on the analysis and comparison of the feasibility and solution of implementing code instrumentation in different compiling phases. One CPU-independent syntax and lexical analysis phase with code instrumentation function based on the preprocessed file inserted between preprocessing and compiling phase is proposed and the implementation of one simple prototype is provided at last.

Key words coverage test; code instrumentation; compile; lexical analysis; syntax analysis