深入浅出 Linux 平台代码覆盖率测试

——原理、工具、分析

（第一版）

作者：余祖波

Mail：livelylittlefish@gmail.com

Blog：http://blog.csdn.net/livelylittlefish

http://www.abo321.org

Contents

Linux 平台代码覆盖率测试工具 GCOV 简介

2011 年 4 月 11 日
17:53

1. gcov 是什么？

- Gcov is GCC Coverage
- 是一个测试代码覆盖率的工具
- 是一个命令行方式的控制台程序

- 伴随 GCC 发布，配合 GCC 共同实现对 C/C++文件的语句覆盖和分支覆盖测试；
- 与程序概要分析工具(profiling tool，例如 **gprof**)一起工作，可以估计程序中哪一段代码最耗时；

注：程序概要分析工具是分析代码性能的工具。

2. gcov 能做什么？

gcov 可以统计
- 每一行代码的执行频率
- 实际上哪些代码确实被执行了
- 每一段代码(section code)的耗时(执行时间)

因此，gcov 可以帮你优化代码，当然这个优化动作还是应该有开发者完成。

3. 如何使用 gcov？

笔者也以 gcov 的 manual 页自带的例子为例，代码(没有做任何改动)如下。

filename: test.c

```
01: #include <stdio.h>
02:
03: int main (void)
04: {
05:   int i, total;
06:
07:   total = 0;
08:
09:   for (i = 0; i < 10; i++)
10:     total += i;
11:
12:   if (total ! = 45)
13:     printf ("Failure\n");
14:   else
15:     printf ("Success\n");
16:   return 0;
17: }
18:
```

3.1 使用 gcov 的 3 个阶段

(1) 编译

```
# gcc -fprofile-arcs -ftest-coverage -o test test.c
# ls
test  test.c  test.gcno
```

-fprofile-arcs -ftest-coverage 告诉编译器生成 gcov 需要的额外信息，并在目标文件中插入 gcov 需要的 extra profiling information。因此，该命令在生成可执行文件 test 的同时生成 test.gcno 文件(gcov note 文件)。

(2) 收集信息

```
# ./test
Success
# ls
test  test.c  test.gcda  test.gcno
```

执行该程序，生成 test.gcda 文件(gcov data 文件)。

(3) 报告

```
# gcov test.c
File 'test.c'
Lines executed:87.50% of 8
test.c:creating 'test.c.gcov'

# ls
test  test.c  test.c.gcov  test.gcda  test.gcno
```

生成 test.c.gcov 文件，该文件记录了每行代码被执行的次数。

test.c.gcov 文件内容如下，蓝色表示笔者添加的注释。

```
        -:    0:Source:test.c
        -:    0:Graph:test.gcno
        -:    0:Data:test.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>          //前面的数字表明该 clause 被执行的次数，下同
        -:    2:
        -:    3:int main (void)
        1:    4:{
        -:    5:    int i, total;
        -:    6:
        1:    7:    total = 0;
        -:    8:
       11:    9:    for (i = 0; i < 10; i++)//前面的数字 11 表明该 clause 被执行 11 次
       10:   10:        total += i;
        -:   11:
        1:   12:    if (total != 45)
    #####:   13:        printf ("Failure\n");
        -:   14:    else
        1:   15:        printf ("Success\n");
        1:   16:    return 0;
        -:   17:}
        -:   18:
```

3.2 gcov 的选项

gcov 的选项不多，也好理解，此处选 3 个典型的选项并结合例子加以说明。

(1) -a, --all-blocks

在.gcov 文件中输出每个基本快(basic block)的执行次数。如果没有-a 选项，则输出'main'函数这个 block 的执行次数，如上所示。使用该选项可以

```
    Write individual execution counts for every basic block.  Normally gcov
outputs execution counts only for the main blocks of a line.  With this option
you can determine if blocks within a single line are not being executed.
```

```
# gcov -a test.c
File 'test.c'
Lines executed:87.50% of 8
test.c:creating 'test.c.gcov'
```

Test.c.gcov 文件内容。

```
        -:    0:Source:test.c
        -:    0:Graph:test.gcno
        -:    0:Data:test.gcda
        -:    0:Runs:1
```

```
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
        1:    4:{
        -:    5:    int i, total;
        -:    6:
        1:    7:    total = 0;
        -:    8:
       11:    9:    for (i = 0; i < 10; i++)
        1:    9-block  0
       10:    9-block  1
       11:    9-block  2
       10:   10:        total += i;
        -:   11:
        1:   12:    if (total != 45)
        1:   12-block  0
    #####:   13:        printf ("Failure\n");
    $$$$$:   13-block  0
        -:   14:    else
        1:   15:        printf ("Success\n");
        1:   15-block  0
        1:   16:    return 0;
        1:   16-block  0
        -:   17:}
        -:   18:
```

(2) -b, --branch-probabilities

在.gcov 文件中输出每个分支的执行频率，并有分支统计信息。

```
# gcov -b test.c
File 'test.c'
Lines executed:87.50% of 8
Branches executed:100.00% of 4
Taken at least once:75.00% of 4
Calls executed:50.00% of 2
test.c:creating 'test.c.gcov'
```

```
        -:    0:Source:test.c
        -:    0:Graph:test.gcno
        -:    0:Data:test.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 100% blocks executed 86%
        1:    4:{
        -:    5:    int i, total;
        -:    6:
        1:    7:    total = 0;
        -:    8:
       11:    9:    for (i = 0; i < 10; i++)
branch  0 taken 91%
branch  1 taken 9% (fallthrough)
       10:   10:        total += i;
        -:   11:
        1:   12:    if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   13:        printf ("Failure\n");
call    0 never executed
        -:   14:    else
        1:   15:        printf ("Success\n");
call    0 returned 100%
        1:   16:    return 0;
        -:   17:}
        -:   18:
```

(3) -c, --branch-counts

在.gcov 文件中输出每个分支的执行次数。

```
# gcov -c test.c
File 'test.c'
Lines executed:87.50% of 8
test.c:creating 'test.c.gcov'
```

-c 是默认选项，其结果与"gcov test.c"执行结果相同。

其他选项，请读者参考相关文档。

4. 小结

本文简单介绍了 Linux 平台 GCC 自带的代码覆盖率测试工具 GCOV 的基本情况是使用方法。详细研究需要参考官方文档或者一些研究者的论文。

Reference

Gcov 的 manual 页
http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
http://dev.firnow.com/course/6_system/linux/Linuxjs/20071129/88999.html

Linux 平台代码覆盖率测试工具 GCOV 的前端工具 LCOV 简介

2011 年 4 月 12 日
8:47

1. Lcov 是什么？

- 是 GCOV 图形化的前端工具
- 是 Linux Test Project 维护的开放源代码工具，最初被设计用来支持 Linux 内核覆盖率的度量
- 基于 Html 输出，并生成一棵完整的 HTML 树
- 输出包括概述、覆盖率百分比、图表，能快速浏览覆盖率数据
- 支持大项目，提供三个级别的视图：目录视图、文件视图、源码视图

```
Use lcov to collect coverage data and genhtml to create HTML pages. Coverage
data can either be collected from the currently running Linux kernel or from a
user space application. To do this, you  have  to  complete the following
preparation steps:

For Linux kernel coverage:
  Follow  the setup instructions for the gcov-kernel infrastructure:
http://ltp.sourceforge.net/coverage/gcov.php

For user space application coverage:
  Compile the application with GCC using the options "-fprofile-arcs" and "-
ftest-coverage".
```

2. 如何在 Linux 平台安装 Lcov？

```
# wget http://downloads.sourceforge.net/ltp/lcov-1.9.tar.gz
# tar -zxvf lcov-1.9.tar.gz
# cd lcov-1.9
# ls
bin      contrib  descriptions.tests  lcovrc    man     rpm
CHANGES  COPYING  example             Makefile  README
# make install
```

不需要编译，直接安装即可，lcov, gendesc, genhtml, geninfo, genpng 将被安装到/usr/bin 目录。

3. 如何使用 Lcov？

以 Linux 平台代码覆盖率测试工具 GCOV 简介一文的例子为例。

(1) 使用 lcov 收集覆盖率数据并写入文件

```
# lcov --capture --directory . --output-file test.info --test-name test
Capturing coverage data from .
Found gcov version: 4.1.2
Scanning . for .gcda files ...
Found 1 data files in .
Processing test.gcda
Finished .info-file creation
```

.表示当前目录，收集 coverage data，即.gcda 文件中的信息，并写入 test.info 文件，且取名为 test。其他选项请参考 lcov 的 manual 页。

test.info 文件内容如下。

```
TN:test
SF:/home/zubo/gcc/2011-04-10.sample/test.c
FN:4,main
FNDA:1,main
FNF:1
FNH:1
BRDA:9,2,0,10
```

```
BRDA:9,2,1,1
BRDA:12,0,0,0
BRDA:12,0,1,1
BRF:4
BRH:3
DA:4,1
DA:7,1
DA:9,11
DA:10,10
DA:12,1
DA:13,0
DA:15,1
DA:16,1
LF:8
LH:7
end_of_record
```

(2) 使用 genhtml 生成基于 HTML 的输出

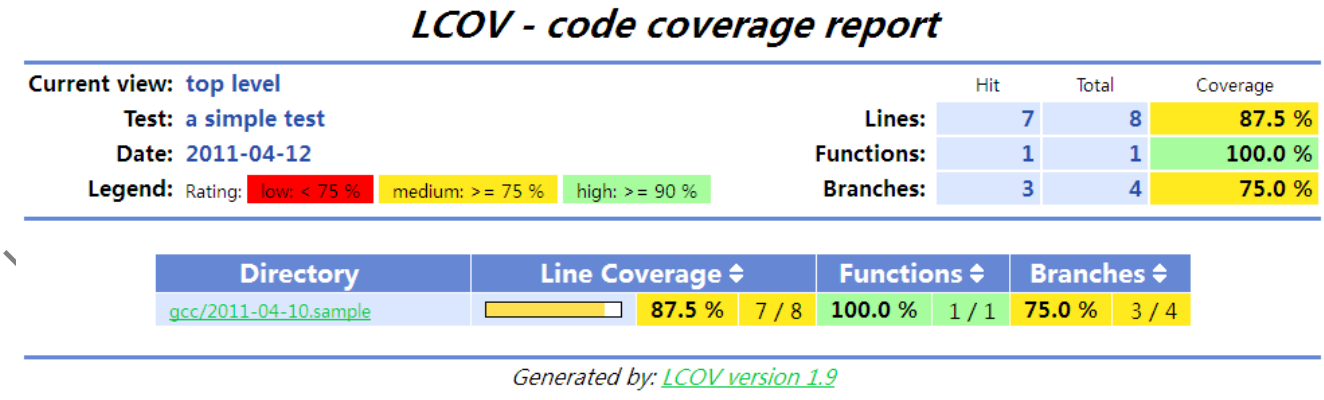
```
# genhtml test.info --output-directory output --title "a simple test" --show-details --legend
Reading data file test.info
Found 1 entries.
Found common filename prefix "/home/zubo"
Writing .css and .png files.
Generating output.
Processing file gcc/2011-04-10.sample/test.c
Writing directory view page.
Overall coverage rate:
  lines......: 87.5% (7 of 8 lines)
  functions..: 100.0% (1 of 1 function)
  branches...: 75.0% (3 of 4 branches)
```

选项解释请参考 genhtml 的 manual 页。cd 到 output 目录，可以看到，生成了很多相关文件，如下。

```
# cd output
# ls
amber.png    gcov.css    index-sort-b.html  ruby.png
emerald.png  glass.png   index-sort-f.html  snow.png
gcc          index.html  index-sort-l.html  updown.png
```

(3) 该例子的图形显示

(3.1) top level 的视图



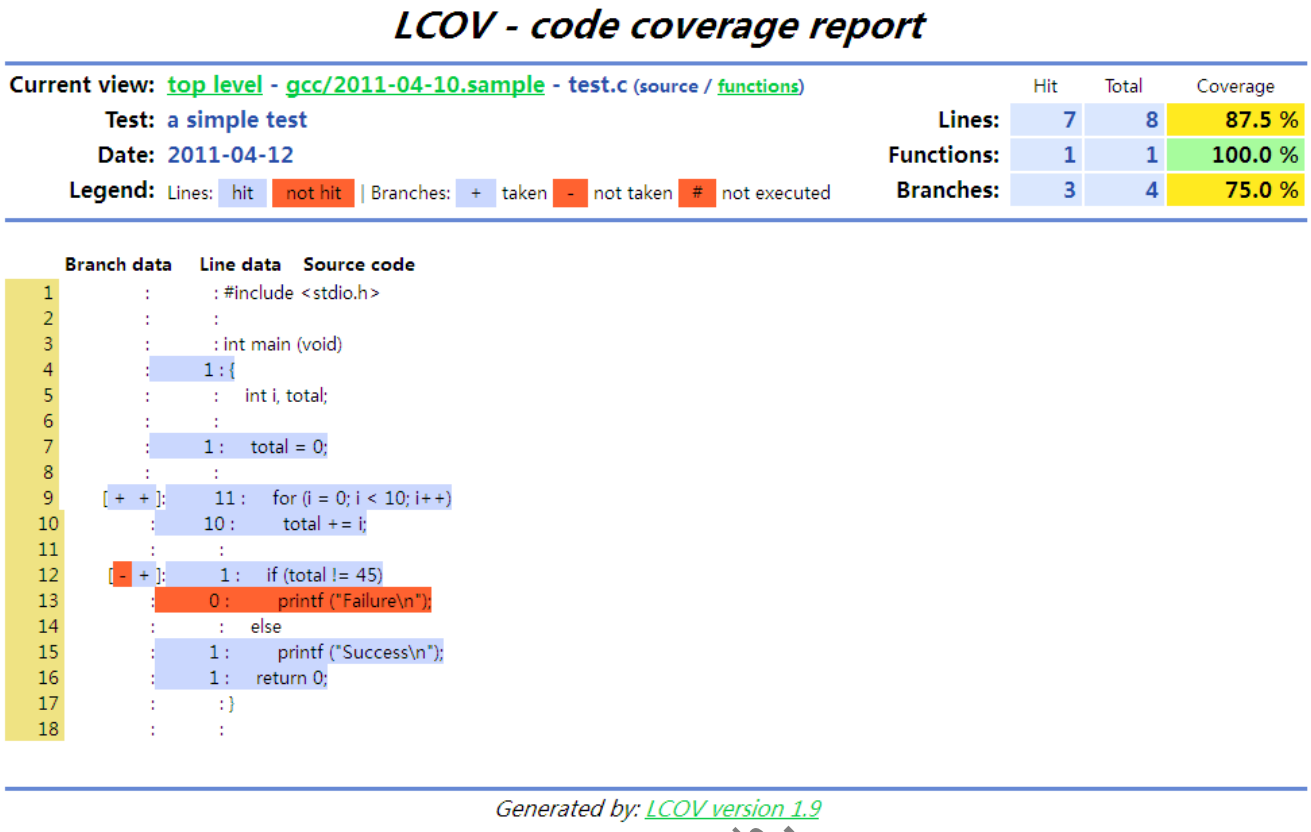
LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level		Lines:	7	8	87.5 %
Test:	a simple test		Functions:	1	1	100.0 %
Date:	2011-04-12		Branches:	3	4	75.0 %
Legend:	Rating:	low: < 75 % / medium: >= 75 % / high: >= 90 %				

Directory	Line Coverage		Functions		Branches	
gcc/2011-04-10.sample		87.5 %	7 / 8	100.0 %	1 / 1	75.0 %	3 / 4

Generated by: LCOV version 1.9

(3.2) 文件或函数的视图



LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level - gcc/2011-04-10.sample - test.c (source / functions)		Lines:	7	8	87.5 %
Test:	a simple test		Functions:	1	1	100.0 %
Date:	2011-04-12		Branches:	3	4	75.0 %
Legend:	Lines: hit not hit	Branches: + taken - not taken # not executed				

	Branch data	Line data	Source code
1	:	:	#include <stdio.h>
2	:	:	
3	:	:	int main (void)
4	:	1 :	{
5	:	:	int i, total;
6	:	:	
7	:	1 :	total = 0;
8	:	:	
9	[+ +]:	11 :	for (i = 0; i < 10; i++)
10	:	10 :	total += i;
11	:	:	
12	[- +]:	1 :	if (total != 45)
13	:	0 :	printf ("Failure\n");
14	:	:	else
15	:	1 :	printf ("Success\n");
16	:	1 :	return 0;
17	:	:	}
18	:	:	

Generated by: LCOV version 1.9

4. 编译 lcov 自带例子

```
# cd /usr/src/lcov-1.9/example
# make
```

编译、运行自带例子并查看结果是快速学习某个工具最好的方法。从 example 的 makefile 文件和编译输出，都可以学习相关概念和命令的使用方法。Html 输出可以由/usr/src/lcov-1.9/example/output/index.html 查看。读者可自行实验。

5. 其他相关工具

(1) gcov-dump

或许，我们还可以使用 gcov-dump 命令输出 gcov 的相关数据，但 gcc 默认不编译 gcov-dump，因此，要使用它，可能需要重新编译 gcc。

(2) ggcov

Ggcov is a Graphical tool for displaying gcov test coverage data. 详细信息可参考 http://ggcov.sourceforge.net。

Reference

lcov 的 manual 页
genhtml 的 manual 页
geninfo 的 manual 页
lcov 的 readme 文件，本文/usr/src/lcov-1.9/README
lcov 的 makefile 文件，本文为/usr/src/lcov-1.9/Makefile

Linux 平台代码覆盖率测试工具 GCOV 相关文件分析

2011 年 4 月 13 日
15:05

本文仍以 Linux 平台代码覆盖率测试工具 GCOV 简介一文的例子为例，分析 gcda/gcno 的文件格式和读取/写入方法。

1. 使用 od 命令 dump 文件内容

```
# od -t x4 -w16 test.gcda
0000000 67636461 34303170 4e8eb3f0 01000000  //magic version stamp tag
0000020 00000002 00000003 eb65a768 01a10000  //length func_ident checksum tag
0000040 0000000a 0000000a 00000000 00000000  //length
0000060 00000000 00000001 00000000 00000000
0000100 00000000 00000001 00000000 a1000000  //0xa1000000 is
GCOV_TAG_OBJECT_SUMMARY
0000120 00000009 00000000 00000005 00000001  //length
0000140 0000000c 00000000 0000000a 00000000
0000160 0000000a 00000000 a3000000 00000009  //0xa3000000 is
GCOV_TAG_PROGRAM_SUMMARY
0000200 51924f98 00000005 00000001 0000000c  //checksum number runs
0000220 00000000 0000000a 00000000 0000000a
0000240 00000000 00000000
0000250
```

od 命令的使用方法可参考其 manual 页。

2. 文件内容解析

(1) file magic

0x67636461 is file magic, that is, "gcda".

0x67636461 是怎么来的呢？细心的读者一定会发现，实际上就是'g','c','d','a'字符的 ASCII 码组成的，即 0x67,0x63,0x64,0x61。即采用字符的 ASCII 码作为文件 magic。可参考附录的解释。

defined as the following.

```
/* File suffixes.  */
#define GCOV_DATA_SUFFIX ".gcda"
#define GCOV_NOTE_SUFFIX ".gcno"

/* File magic. Must not be palindromes.  */
#define GCOV_DATA_MAGIC ((gcov_unsigned_t)0x67636461) /* "gcda" */
#define GCOV_NOTE_MAGIC ((gcov_unsigned_t)0x67636e6f) /* "gcno" */
```

(2) version

0x34303170 is the GCOV_VERSION, that is, 401p, 即 4.1.2p
该版本常量在 gcov_iov.h 文件中定义，如下。

```
/* Generated automatically by the program `./gcov-iov'
   from `4.1.2 (4 1) and p (p)'.  */

#define GCOV_VERSION ((gcov_unsigned_t)0x34303170)  /* 401p */
```

然而，这个文件是在编译 GCC 时自动产生的；文件的内容，是有 gcov_iov 程序产生，该程序由 gcov_iov.c 编译得来，我们可以直接在 gcc 源代码下的 gcc 目录编译该文件，例如。

```
# cd /home/abo/gcc-4.1.2/gcc
# gcc -g -o gcov-iov gcov-iov.c
# ./gcov-iov 4.1.2 p
/* Generated automatically by the program `./gcov-iov'
   from `4.1.2 (4 1) and p (p)'.  */
```

```
#define GCOV_VERSION ((gcov_unsigned_t)0x34303170)  /* 401p */

# ./gcov-iov 4.1 p
/* Generated automatically by the program `./gcov-iov'
   from `4.1 (4 1) and p (p)'.  */

#define GCOV_VERSION ((gcov_unsigned_t)0x34303170)  /* 401p */
```

同样的道理，0x34303170 即为字符'4','0','1','p'的 ASCII 码组成。'p'代表 prerelease，请参考附录。

(3) time stamp

0x4e8eb3f0=1317975024 is the time stamp from GreenWich, it will be read and discarded.

可以使用 date 名验证这个时间，如下。不过，数值上好像有些差异，至于原因，本文不再研究。

```
# date -d @1317975024 +"%F %T %z"
2011-10-07 16:10:24 +0800
# date --date='2011-04-13 11:13:07' +%s
1302664387
```

(4) FUNCTION tag

0x01000000 is a **FUNCTION** tag, defined as follows.

```
/* The record tags.  Values [1..3f] are for tags which may be in either
   file.  Values [41..9f] for those in the note file and [a1..ff] for
   the data file.  The tag value zero is used as an explicit end of
   file marker -- it is not required to be present.  */
#define GCOV_TAG_FUNCTION               ((gcov_unsigned_t)0x01000000)
#define GCOV_TAG_FUNCTION_LENGTH        (2)
#define GCOV_TAG_BLOCKS                 ((gcov_unsigned_t)0x01410000)
#define GCOV_TAG_BLOCKS_LENGTH(NUM)     (NUM)
#define GCOV_TAG_BLOCKS_NUM(LENGTH)     (LENGTH)
#define GCOV_TAG_ARCS                   ((gcov_unsigned_t)0x01430000)
#define GCOV_TAG_ARCS_LENGTH(NUM)       (1 + (NUM) * 2)
#define GCOV_TAG_ARCS_NUM(LENGTH)       (((LENGTH) - 1) / 2)
#define GCOV_TAG_LINES                  ((gcov_unsigned_t)0x01450000)
#define GCOV_TAG_COUNTER_BASE           ((gcov_unsigned_t)0x01a10000)
#define GCOV_TAG_COUNTER_LENGTH(NUM)    ((NUM) * 2)
#define GCOV_TAG_COUNTER_NUM(LENGTH)    ((LENGTH) / 2)
#define GCOV_TAG_OBJECT_SUMMARY         ((gcov_unsigned_t)0xa1000000)
#define GCOV_TAG_PROGRAM_SUMMARY        ((gcov_unsigned_t)0xa3000000)
#define GCOV_TAG_SUMMARY_LENGTH         (1 + GCOV_COUNTERS_SUMMABLE * (2 + 3 * 2))
```

then, 0x00000002 is its length; and 0x00000003 is the function identifier. Next, 0xeb65a768 is the checksum.

注 1：只有是 FUNCTION tag 时，才会有后续的 length, function identifier 和 checksum。

FUNCTION 数据结构如下。

```
/* Information about a single function.  This uses the trailing array
   idiom. The number of counters is determined from the counter_mask
   in gcov_info.  We hold an array of function info, so have to
   explicitly calculate the correct array stride.  */
struct gcov_fn_info
{
  gcov_unsigned_t ident;        /* unique ident of function */
  gcov_unsigned_t checksum;        /* function checksum */
  unsigned n_ctrs[0];                /* instrumented counters */
};
```

(5) COUNTER tag

0x01a10000 is a **COUNTER** tag, defined as above. then, 0x0000000a is its length. 因此，counter number 由宏 GCOV_TAG_COUNTER_NUM 计算得来，为 5。

接下来就是 5 个 counters，每个 counter 为 2 个 words，每个 word 为 4Byte，即每个 counter 为 64bits 的整数，共 40Bytes。由 gcov_read_counter()函数完成读取，从对该函数的调用可以看出，每次均读取 2 个 words(8Bytes)。

counter 定义如下。

```
/* Type of function used to merge counters.  */
typedef void (*gcov_merge_fn) (gcov_type *, gcov_unsigned_t);

/* Information about counters.  */
struct gcov_ctr_info
{
  gcov_unsigned_t num;                  /* number of counters.  */
  gcov_type *values;                   /* their values.  */
  gcov_merge_fn merge;              /* The function used to merge them.  */
};
```

由该结构也可看出，num 后面即是他们的 vlaues，类型是 gcov_type 的指针，然后是 merge function 指针。

(6) OBJECT SUMMARY tag

0xa1000000 is **GCOV_TAG_OBJECT_SUMMARY**, and the following 0x00000009 is the length. Next, 9 words following it.

Object 结构如下。

```
/* Cumulative counter data.  */
struct gcov_ctr_summary
{
  gcov_unsigned_t num;                  /* number of counters.  */
  gcov_unsigned_t runs;                 /* number of program runs */
  gcov_type sum_all;                    /* sum of all counters accumulated.  */
  gcov_type run_max;                    /* maximum value on a single run.  */
  gcov_type sum_max;                /* sum of individual run max values.  */
};

/* Object & program summary record.  */
struct gcov_summary
{
  gcov_unsigned_t checksum;         /* checksum of program */
  struct gcov_ctr_summary ctrs[GCOV_COUNTERS_SUMMABLE];
};
```

(7) PROGRAM SUMMARY tag

0xa3000000 is **GCOV_TAG_PROGRAM_SUMMARY**, and the following 0x00000009 is the length, then, 0x51924f98 is the checksum.其结构定义与 Object 相同，如上所示。

then, 3 counters, that is, sum of all counters accumulated, maximum value on a single run, and sum of individual run max values, 即 sum_all, run_max, sum_max。每个 program summary 是 32Bytes。

(8) file end

最后一个 unsigned 数是 0x00000000，读到后即退出循环，并关闭文件。由 tag 的定义解释可以看出该设计，如(4)。

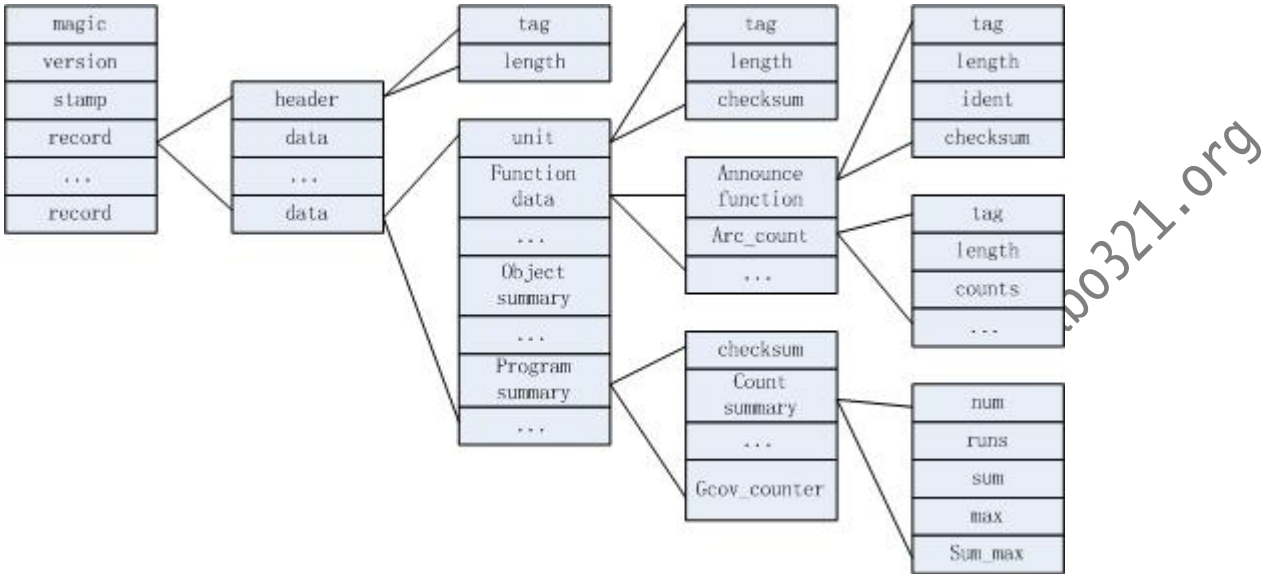
至此，文件分析完毕。以上所有定义基本上都在 gcov_io.h 文件中。

注 2：本文的分析，只看到了值为 0x**01**000000, 0x**01**a10000, 0x**a1**000000tag 信息，另外的 0x**0143**0000, 0x**0145**0000tag 信息可在.gcno 文件中看到，本文不再叙述。附录的解释很清楚，如下。*Level values [1..3f] are used for common tags, values [41..9f] for the notes file and [a1..ff] for the data file.*

注 3：附录中的解释是从 gcov_io.h 文件中摘录而来，该官方文档对文件格式的解释非常详细，供参考。

(9) .gcda/.gcno 文件格式小结

其文件格式如下图所示。



其中，
magic 是注释文件和数据文件的区别标记；
verson 记录 GCC 的版本信息；
stamp 是时间戳，主要用于区别编译/运行/再编译的阶段周期；
record 记录信息，由 header 和 data 两部分组成，不能嵌套使用，通过 header 中的 tag 组织成层次结构。tag 在文件中是唯一的。header 部分的 length 表示 data 项的数量。

summary 给出了整个目标文件和程序的相关信息。有 Object summary 和 program summary 两种，其数据结构相同。

record 记录中的数据主要放在 data 部分。在 data 项中，unit 用来区别同一 record 记录下不同的数据项。
function_data 和 summary 项则真正记录了笔者关心的剖视信息。
function_data 包含 2 个部分：announce_function 和 arc_counts；

announce_function 各个域如下。

tag(32 位)	length(32 位)	ident(32 位)	checksum(32 位)
标记 arc_counts	数据项数	函数的唯一标识	校验码

arc_counts 中包含的 tag 域给出标明了该 arc_counts 项记录的信息类型，目前 GCC 所能支持的值信息类型主要有 7 种，可在 gcov_io.h 文件中看到。

3. 文件读取函数及其调用过程

3.1 读取/写入相关调用

上述读取该文件的函数均在 gcov_io.c 文件中实现。该过程的函数调用顺序如下。

```
 main
->toplev_main
 ->do_compile
  ->compile_file
   ->coverage_init
```

```
    ->read_counts_file  //读取 gcda 文件便在该函数中完成
```

read_counts_file()函数将调用 gcov_read_words(), gcov_read_unsigned(), gcov_read_counter(), gcov_read_string(), gcov_read_summary()完成读取。
相反地，gcov_write_words(), gcov_write_unsigned(), gcov_write_counter()等完成写入。且大部分写入操作均在 gcov_exit()中完成。

gcov_exit()将调用 3 个文件操作 gcov_open, gcov_close, gcov_write_block，当然也会调用中间层次的函数如 gcov_write_tag_length, gcov_write_counter, gcov_write_summary, gcov_write_unsigned, gcov_write_words 等。

3.2 程序退出点

程序是在 atexit()中调用 gcov_exit()退出的，在 gcov_exit()，将调用写入操作，如上分析。
atexit 是 glibc 的函数。

那么注册 gcov_exit()函数是谁完成的呢？
__gcov_init()函数调用 atexit()完成 gcov_exit()的注册。因此，当程序退出时将在 atexit()中调用 gcov_exit()完成文件的写入。__gcov_init 函数在 libgcov.c 文件中。

Reference

man date
info date
man od
date 源代码
gcov_iov.c
gcov_io.h
gcov_io.c
Libgcov.c
Coverage.c
Coverage.h

Appendix: gcov 文件格式定义

```
//此段文字描述 coverage information 的文件，有.gcno(note 文件)和.gcda(data 文件)。
Coverage information is held in two files.  A notes file, which is generated by
the compiler, and a data file, which is generated by the program under test.
Both files use a similar structure.  We do not attempt to make these files
backwards compatible with previous  versions, as you only need coverage
information when developing a program.  We do hold version information, so that
mismatches can be detected, and we use a format that allows tools to skip
information they do not understand or are not interested in.

Numbers are recorded in the 32 bit unsigned binary form of the endianness of
the machine generating the file. 64 bit numbers are stored as two 32 bit
numbers, the low part first. Strings are padded with 1 to 4 NUL bytes, to bring
the length up to a multiple of 4. The number of 4 bytes is stored, followed by
the padded string. Zero length and NULL strings are simply stored as a length
of zero (they have no trailing NUL or padding).

     int32:  byte3 byte2 byte1 byte0 | byte0 byte1 byte2 byte3  //32Bits 的数据
构成
     int64:  int32:low int32:high  //64Bits 的数据构成
     string: int32:0 | int32:length char* char:0 padding
       padding: | char:0 | char:0 char:0 | char:0 char:0 char:0
     item: int32 | int64 | string  //item 由 1 个 32Bits，1 个 64Bits 和 1 个 string 构
成

//文件格式如下
The basic format of the files is

//此处印证本文的分析，文件前面 12 字节即为 magic,version,stamp(各 4Bytes)
```

```
     file : int32:magic int32:version int32:stamp record*
```

//此段文字描述文件头各个字段的作用。
The magic ident is different for the notes and the data files. The magic ident is used to determine the endianness of the file, when reading. The version is the same for both files and is derived from gcc's version number. The stamp value is used to synchronize note and data files and to synchronize merging within a data file. It need not be an **absolute time stamp**, merely a ticker that increments fast enough and cycles slow enough to distinguish different compile/run/compile cycles.

//此段文字表述文件头各个字段怎么来的，尤其详细介绍了 version 的构成。
Although the ident and version are formally 32 bit numbers, they are derived from **4 character ASCII strings**. The version number consists of the single character major version number, a two character minor version number (leading zero for versions less than 10), and a single character indicating the status of the release.
That will be 'e' experimental, 'p' prerelease and 'r' for release. Because, by good fortune, these are in alphabetical order, string collating can be used to compare version strings. Be aware that the 'e' designation will (naturally) be unstable and might be incompatible with itself. For gcc 3.4 experimental, it would be '304e' (0x33303465). When the major version reaches 10, the letters A-Z will be used. Assuming minor increments releases every 6 months, we have to make a major increment every 50 years. Assuming major increments releases every 5 years, we're ok for the next 155 years -- good enough for me.

A record has a tag, length and variable amount of data.

```
     record: header data              //record 由 header 和 data 组成
     header: int32:tag int32:length   //header 由一个 32Bits 的 tag 和一个 32bits 的 length 组成
     data: item*                      //随后就是一些数据
```

//此段文字描述 tag 的组成规则，由 4 个 level 的数字组成，每个 level 是 1 个字节，反映的是 record 层次。
Records are not nested, but there is a record hierarchy. Tag numbers reflect this hierarchy. Tags are unique across note and data files. Some record types have a varying amount of data. The LENGTH is the number of 4bytes that follow and is usually used to determine how much data. The tag value is split into 4 8-bit fields, one for each of four possible levels. The most significant is allocated first. Unused levels are zero. Active levels are odd-valued, so that the LSB of the level is one. A sub-level incorporates the values of its superlevels. This formatting allows you to determine the tag hierarchy, without understanding the tags themselves, and is similar to the standard section numbering used in technical documents. Level values [1..3f] are used for common tags, values [41..9f] for the notes file and [a1..ff] for the data file.

```
The basic block graph file contains the following records  //注意缩进，缩进代表构成层次
note: unit function-graph*  //note 文件由 unit 和 function-graph 数据组成(解释方法下同)
unit: header int32:checksum string:source  //unit 由 header 和 32Bits 的 checksum 和 source 字符串组成
  string:name string:source int32:lineno
function-graph: announce_function basic_blocks {arcs | lines}*  //*表示 0 个或多个
  announce_function: header int32:ident int32:checksum
  basic_block: header int32:flags*  //基本块由 header 和 0 个或多个 32bits 的 flag 构成
  arcs: header int32:block_no arc*  //arcs 即为跳转表，由 header,32bits 的块号和 0 个或多个 arc 构成
    arc:  int32:dest_block int32:flags  //跳转由 32bits 的目标块和 32bits 的 flag 构成
  lines: header int32:block_no line*
         int32:0 string:NULL
    line:  int32:line_no | int32:0 string:filename
```

//此段文字描述基本块(basic block)的组成
The **BASIC_BLOCK record** holds per-bb flags. The number of blocks can be inferred from its data length. There is one **ARCS record** per basic block. The number of

arcs from a bb is implicit from the data length. It enumerates the destination bb and per-arc flags. There is one **LINES record** per basic block, it enumerates the source lines which belong to that basic block. **Source file names** are introduced by a line number of 0, following lines are from the new source file. The initial source file for the function is NULL, but the current source file should be remembered from one LINES record to the next. The **end of a block** is indicated by an empty filename this does not reset the current source file. Note there is no ordering of the ARCS and LINES records: they may be in any order, interleaved in any manner. The current filename follows the order the LINES records are stored in the file, *not* the ordering of the blocks they are for.

(1) BB 记录含有每个 BB 的 flags。
(2) 每个 BB 有 1 个 ARCS 记录，每个 ARCS 记录中 arcs 个数由该 ARCS 记录的数据长度计算出来，由宏 GCOV_TAG_ARCS_NUM 完成，如下。每个 ARCS 记录列出目的 BB 的 blockno，及其 flags。

```
#define GCOV_TAG_ARCS_NUM(LENGTH)  (((LENGTH) - 1) / 2)
```

(3) 每个 BB 有 1 个 LINES 记录，列出属于该 BB 的所有代码行号。
(4) 源文件名有 lineno=0 开始，且其后的所有 lineno 均为该源文件中的行号。

```
//data 文件的构成
The data file contains the following records.
data: {unit function-data* summary:object summary:program*}*  //定义方式同上
  unit: header int32:checksum
  function-data:        announce_function arc_counts  //function-data 构成
    announce_function: header int32:ident int32:checksum
    arc_counts: header int64:count*
  summary: int32:checksum {count-summary}GCOV_COUNTERS
    count-summary: int32:num int32:runs int64:sum int64:max int64:sum_max
//32Bytes
```

//此处的 count-summary 描述正对应 gcov_ctr_summary 结构，如下。

```
/* Cumulative counter data.  */
struct gcov_ctr_summary
{
  gcov_unsigned_t num;            /* number of counters.  */
  gcov_unsigned_t runs;            /* number of program runs */
  gcov_type sum_all;              /* sum of all counters accumulated.  */
  gcov_type run_max;              /* maximum value on a single run.  */
  gcov_type sum_max;          /* sum of individual run max values.  */
};
```

```
//此段文字描述每个字段的作用
The ANNOUNCE_FUNCTION record is the same as that in the note file, but without the source location. The ARC_COUNTS gives the counter values for those arcs that are instrumented. The SUMMARY records give information about the whole object file and about the whole program. The checksum is used for whole program summaries, and disambiguates different programs which include the same instrumented object file. There may be several program summaries, each with a unique checksum. The object summary's checksum is zero. Note that the data file might contain information from several runs concatenated, or the data might be merged.
```

//此段文字描述该文件会被 gcc 源代码、gcov 工具和运行库包含，且通过宏 IN_LIBGCOV 和 IN_GCOV 来区分。

```
This file is included by both the compiler, gcov tools and the runtime support library libgcov. IN_LIBGCOV and IN_GCOV are used to distinguish which case is which. If IN_LIBGCOV is nonzero, libgcov is being built. If IN_GCOV is nonzero, the gcov tools are being built. Otherwise the compiler is being built. IN_GCOV may be positive or negative. If positive, we are compiling a tool that requires additional functions (see the code for knowledge of what those functions are).
```

```
//宏总结如下
Build libgcov  : IN_LIBGCOV=1
Build gcov tool: IN_GCOV=1 (为正值时需要额外的函数)
Build gcc      : otherwise
```

注：了解了文件格式后，再写读取和写入的程序就容易多了，因为采用的是二进制方式读取/写入，用的最多的操作就是 fread 和 fwrite。

Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析

2011 年 4 月 25 日
18:26

0. 序

某些版本的 Gcc 在默认情况下编译，可能不会产生 gcov-dump 程序，或者不会安装到/usr/bin。但 gcov-dump 程序在做覆盖率测试时 dump 相关文件(.gcda/.gcno)内容时非常必要和好用。

Gcc 的编译耗时又繁琐，如果某些配置不正确，会导致编译过程中各种莫名其妙的错误。因此，本文主要讲述在不重新编译整个 Gcc 项目的情况下，如何获得 gcov-dump 程序。

本文在 Linux 平台上实验，以 gcc-4.1.2 为例，且 gcc 源代码在/usr/src/gcc-4.1.2 目录。以下若不做特别说明，.表示 gcc 源代码目录，即/usr/src/gcc-4.1.2。

1. 编译 gcov/gcov-dump

Gcov-dump.c 位于./gcc 目录下，因此，可以通过./gcc 的 makefile 文件编译生成 gcov-dump。gcc 目录下 configure 程序即可生成该 makefile。

makefile 文件中的 gcov-dump 如下，由./gcc/build 下的 Makefile 文件中抽取出来。

```
exeext =

CPPLIB = ../libcpp/libcpp.a
LIBIBERTY = ../libiberty/libiberty.a

# Internationalization library.
LIBINTL =
LIBINTL_DEP =

# Character encoding conversion library.
LIBICONV =
LIBICONV_DEP =

LIBS =  $(CPPLIB) $(LIBINTL) $(LIBICONV) $(LIBIBERTY)

gcov.o: gcov.c gcov-io.c $(GCOV_IO_H) intl.h $(SYSTEM_H) coretypes.h $(TM_H) \
   $(CONFIG_H) version.h
gcov-dump.o: gcov-dump.c gcov-io.c $(GCOV_IO_H) $(SYSTEM_H) coretypes.h \
   $(TM_H) $(CONFIG_H)

GCOV_OBJS = gcov.o intl.o version.o errors.o
gcov$(exeext): $(GCOV_OBJS) $(LIBDEPS)
     $(CC) $(ALL_CFLAGS) $(LDFLAGS) $(GCOV_OBJS) $(LIBS) -o $@

GCOV_DUMP_OBJS = gcov-dump.o version.o errors.o
gcov-dump$(exeext): $(GCOV_DUMP_OBJS) $(LIBDEPS)
     $(CC) $(ALL_CFLAGS) $(LDFLAGS) $(GCOV_DUMP_OBJS) $(LIBS) -o $@
```

至于其他的定义，非本文重点，不予解释。

```
# cd /usr/src/gcc-4.1.2/gcc/build
# make clean
# make gcov-dump
...
make: *** No rule to make target `../build-i686-pc-linux-
gnu/libiberty/libiberty.a', needed by `build/genmodes'.  Stop.

# make gcov
...
make: *** No rule to make target `../build-i686-pc-linux-
gnu/libiberty/libiberty.a', needed by `build/genmodes'.  Stop.
```

程序提示在../build-i686-pc-linux-gnu/libiberty 目录下没有找到 genmodes 所需的静态库 libiberty.a。

从 makefile 文件中也了解到，gcov/gcov-dump 实际所需的静态库是 libcpp.a 和 libiberty.a。实际上，只需在 gcc 目录下的 makefile 文件中指定好这两个静态库的路径(绝对路径和相对路径均可)即可解决问题。例如：/usr/bin/libiberty.a。

2. 额外的话

不得不指出的一点：事实上，对 libcpp.a 的依赖几乎为 0，且对 libiberty.a 的依赖也仅限于以下函数。

```
FILE *fopen_unlocked (const char *, const char *);
void unlock_std_streams (void);
```

这两个函数的声明在./include/libiberty.h 中。

因此，了解这些之后，就可以将 gcov/gcov-dump 相关的文件抽取出来，单独成为一个独立的项目，来编译出 gcov/gcov-dump，以方便对 gcov/gcov-dump 源代码和原理的学习、调试。
——将另文讨论。

3. gcov-dump 程序的一个 bug

3.1 bug 描述

用上述生成的 gcov-dump 程序 dump 出某个.gcda 文件的内容，如下。

```
# cd /home/zubo/gcc/test
# /usr/src/gcc-4.1.2/gcc/build/gcov-dump test.gcda
test.gcda:data:magic `gcda':version `401p'
test.gcda:stamp 1427241144
test.gcda: 01000000:   2:FUNCTION ident=3, checksum=0xeb65a768
test.gcda:  01a10000:  10:COUNTERS arcs 5 counts
test.gcda:              0 10 0 1 0 1
test.gcda: a1000000:   9:OBJECT_SUMMARY checksum=0x00000000
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda:              counts=1, runs=1, sum_all=577750259318514008, run_max=-
6845471430389142944, sum_max=577764773093965833
test.gcda:              counts=3214008580, runs=134522083,
sum_all=577765013746656483, run_max=-1208884056, sum_max=29051165790196
test.gcda:              counts=134527488, runs=1882271796, sum_all=-
6845471431969184921, run_max=577765507560701952, sum_max=4429488512
test.gcda:              counts=1734567009, runs=875573616, sum_all=4429489327,
run_max=91621554360, sum_max=-6845471433603153901
test.gcda: a3000000:   9:PROGRAM_SUMMARY checksum=0x51924f98
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda:              counts=1, runs=1, sum_all=577750259318514008, run_max=-
6701356242313287072, sum_max=577764837518475273
test.gcda:              counts=3214008580, runs=134522083,
sum_all=577765013746656483, run_max=-1208884056, sum_max=29051165790196
test.gcda:              counts=134527488, runs=1882271796, sum_all=-
6701356243893329049, run_max=577765507560701952, sum_max=4429488512
test.gcda:              counts=1734567009, runs=875573616, sum_all=4429489327,
run_max=138866194616, sum_max=-6701356245527298018
```

dump 出的数据怎么会这么大？会不会有什么问题？使用 od 命令 dump 该文件的内容，看看该文件里到底有没有这些庞大的数据。如下。数据分析可以参考"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"一文。

```
# od -t x4 -w16 test.gcda
0000000 67636461 34303170 5511f8b8 01000000  //'gcda', '401p', timestamp,
tag=0x01000000
0000020 00000002 00000003 eb65a768 01a10000  //FUNCTION, 0x01a10000
0000040 0000000a 0000000a 00000000 00000000  //length=0xa=10, counter content:
```

```
0xa, 0, 1, 0, 1
0000060 00000000 00000001 00000000 00000000  //8 Bytes for each counter
0000100 00000000 00000001 00000000 a1000000
0000120 00000009 00000000 00000005 00000001  //length=9, checksum=0, counts=5,
runs=1
0000140 0000000c 00000000 0000000a 00000000  //sum_all=0xc=12(8 Bytes),
run_max=0xa=10(8 Bytes)
0000160 0000000a 00000000 a3000000 00000009  //sum_max=0xa=10(8 Bytes),
tag=a3000000, length=9
0000200 51924f98 00000005 00000001 0000000c  //same as above
0000220 00000000 0000000a 00000000 0000000a
0000240 00000000 00000000
0000250
```

很显然，该文件里并没有这些庞大的数据，也就证实了我们的猜测。基本可以确定，gcov-dump 有 bug。

3.2 bug 分析与修复

如何分析，自然想到了 gdb。有问题的数据貌似在 dump Object Summary 和 Program Summary 时出现的。那么在 tag_summary()函数中设置断点是很自然的事。经一番调试后，发现问题就在 tag_summary()函数里。如下。

```
static void
tag_summary (const char *filename ATTRIBUTE_UNUSED,
         unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
  struct gcov_summary summary;
  unsigned ix;

  unsigned count = gcov_read_summary (&summary); /* 还应该修改该函数的声明和定义 */
  printf (" checksum=0x%08x", summary.checksum);

  /* for (ix = 0; ix != GCOV_COUNTERS; ix++) */  /* 原来的代码 */
  for (ix = 0; ix < count; ix++)                 /* 应该如此修改 */
    {
      printf ("\n");
      print_prefix (filename, 0, 0);
      printf ("\t\tcounts=%u, runs=%u",
           summary.ctrs[ix].num, summary.ctrs[ix].runs);

      printf (", sum_all=" HOST_WIDEST_INT_PRINT_DEC,
           (HOST_WIDEST_INT)summary.ctrs[ix].sum_all);
      printf (", run_max=" HOST_WIDEST_INT_PRINT_DEC,
           (HOST_WIDEST_INT)summary.ctrs[ix].run_max);
      printf (", sum_max=" HOST_WIDEST_INT_PRINT_DEC,
           (HOST_WIDEST_INT)summary.ctrs[ix].sum_max);
    }
}
```

其中，GCOV_COUNTERS 的定义如下，其值为 5，故每次打印均打印出 5 个 summary 的内容。但实际应该是按照 summary(Object 或者 program)的个数来打印信息。

```
/* Counters that are collected. */
#define GCOV_COUNTER_ARCS 0 /* Arc transitions. */
#define GCOV_COUNTERS_SUMMABLE 1 /* Counters which can be summaried. */
#define GCOV_FIRST_VALUE_COUNTER 1/* The first of counters used for value
    profiling. They must form a consecutive
    interval and their order must match the order of HIST_TYPEs in
    value- prof.h. */
#define GCOV_COUNTER_V_INTERVAL 1 /* Histogram of value inside an interval. */
#define GCOV_COUNTER_V_POW2 2 /* Histogram of exact power2 logarithm of a value. */
#define GCOV_COUNTER_V_SINGLE 3 /* The most common value of expression. */
#define GCOV_COUNTER_V_DELTA 4 /* The most common difference between consecutive values
of expression. */
#define GCOV_LAST_VALUE_COUNTER 4 /* The last of counters used for value profiling. */
#define GCOV_COUNTERS 5
```

输出的信息，如，
counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
即为 gcov_ctr_summary 结构，其定义如下。在./gcc/gcov_io.h 文件中。

```
/* Cumulative counter data. */
struct gcov_ctr_summary
{
    gcov_unsigned_t num;  /* number of ounters. */
    gcov_unsigned_t runs; /* number of program runs */
    gcov_type sum_all;    /* sum of all counters accumulated. */
    gcov_type run_max;    /* maximum value on a single run. */
    gcov_type sum_max;    /* sum of individual run max values. */
};

/* Object & program summary record. */
struct gcov_summary
{
    gcov_unsigned_t checksum; / * checksum of program */
    struct gcov_ctr_summary ctrs[GCOV_COUNTERS_SUMMABLE];
};
```

GCOV_COUNTERS_SUMMABLE=1，因此，gcov_summary 结构中的 ctrs 数组，实际上是一个指针而已，当 summary 有多个时(不超过 5 个)，应该为其分配空间。

3.3 正确的输出

修复该 bug 后，在看结果，就正常了，如下。对照上面的分析，一目了然。

```
# /usr/src/gcc-4.1.2/gcc/build/gcov-dump test.gcda
test.gcda:data:magic `gcda':version `401p'
test.gcda:stamp 1427241144
test.gcda: 01000000:   2:FUNCTION ident=3, checksum=0xeb65a768
test.gcda:  01a10000:  10:COUNTERS arcs 5 counts
test.gcda: a1000000:   9:OBJECT_SUMMARY checksum=0x00000000
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda: a3000000:   9:PROGRAM_SUMMARY checksum=0x51924f98
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
```

可以看到，输出的 Object Summary 和 Program Summary 均只有一个。

dump 相应的.gcno 文件，如下。

```
# /usr/src/gcc-4.1.2/gcc/build/gcov-dump test.gcno
test.gcno:note:magic `gcno':version `401p'
test.gcno:stamp 1487149731
test.gcno: 01000000:   9:FUNCTION ident=3, checksum=0xeb65a768, `main' test.c:4
test.gcno:  01410000:   9:BLOCKS 9 blocks
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:  01450000:  10:LINES
test.gcno:  01450000:   9:LINES
test.gcno:  01450000:   8:LINES
test.gcno:  01450000:   8:LINES
test.gcno:  01450000:   8:LINES
test.gcno:  01450000:   8:LINES
```

3.4 gcov-dump 的打印开关

Gcov-dump 程序中有两个打印开关，如下。

```
static int flag_dump_contents = 1;  //打开该开关，将其置为 1 即可，将打印 coutner 的内容
static int flag_dump_positions = 0;
```

将 flag_dump_contents 开关置为 1，重新编译 gcov-dump，便可打印 counter 的内容。如下。

```
# /usr/src/gcc-4.1.2/gcc/build/gcov-dump test.gcda
test.gcda:data:magic `gcda':version `401p'
test.gcda:stamp 1427241144
test.gcda: 01000000:   2:FUNCTION ident=3, checksum=0xeb65a768
test.gcda:  01a10000:  10:COUNTERS arcs 5 counts
test.gcda:              0 10 0 1 0 1  //此处便是 5 个 counter，共 40 字节
test.gcda: a1000000:   9:OBJECT_SUMMARY checksum=0x00000000
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda: a3000000:   9:PROGRAM_SUMMARY checksum=0x51924f98
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
```

将 flag_dump_positions 开关置为 1，重新编译 gcov-dump，便可打印读取的每一个 tag 的 position 及 counter 的 position。如下。

```
# /usr/src/gcc-4.1.2/gcc/build/gcov-dump test.gcda
test.gcda:data:magic `gcda':version `401p'
test.gcda:stamp 1427241144
test.gcda:3: 01000000:   2:FUNCTION ident=3, checksum=0xeb65a768
test.gcda:7:  01a10000:  10:COUNTERS arcs 5 counts
test.gcda:9:            0 10 0 1 0 1
test.gcda:19: a1000000:   9:OBJECT_SUMMARY checksum=0x00000000
test.gcda:0:            counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda:30: a3000000:   9:PROGRAM_SUMMARY checksum=0x51924f98
test.gcda:0:            counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
```

3.5 一个问题

flag_dump_contents 开关置 1 时，上面红色的 0 是什么？谁打印出来的？

只有第一个 counter 需要显示(打印)文件名，因此 tag_counters()函数通过位与运算判断是否是第一个 counter，如果是第一个 counter，则打印文件名并显示该序号 ix。如下。

```
static void
tag_counters (const char *filename ATTRIBUTE_UNUSED,
              unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
  static const char *const counter_names[] = GCOV_COUNTER_NAMES;
  unsigned n_counts = GCOV_TAG_COUNTER_NUM (length);

  printf (" %s %u counts",
          counter_names[GCOV_COUNTER_FOR_TAG (tag)], n_counts);
  if (flag_dump_contents)
    {
      unsigned ix;

      for (ix = 0; ix != n_counts; ix++)
        {
          gcov_type count;

          if (!(ix & 7))  //只有 ix 为 0 时该条件为真
            {
              printf ("\n");
              print_prefix (filename, 0, gcov_position ());
              printf ("\t\t%u", ix);  //此处打印这个 ix=0
            }

          count = gcov_read_counter ();
          printf (" ");
          printf (HOST_WIDEST_INT_PRINT_DEC, count);
        }
```

```
    }
}
```

实际上应该是 gcc 的设计者的一个技巧，此处的 0，永远是 0，即文件名之后打印一个 0，表示后续 counter 内容的开始，亦可做测试时使用，如果此处不为 0，一定是代码有 bug。

4．总结

本文主要描述了 Linux 平台代码覆盖率测试工具 GCOV 相关程序，主要是 gcov-dump 程序的编译生成及其 bug 分析、修复。后续的文章开始讨论 gcov/gcov-dump 设计及其原理。

Reference

./gcc/gcov-io.h
./gcc/gcov-io.c
./gcc/gcov-dump.c

Linux 平台代码覆盖率测试-从 GCC 源码中抽取 gcov/gcov-dump 程序

2011 年 4 月 28 日
11:33

0. 序

若想研究 gcov/gcov-dump 原理或者代码，深入函数内部跟踪调试是最好的理解方式，但 gcc 的源代码毕竟比较庞大，欲从中抽丝剥茧，往往会被 gcc 的庞大源代码吓住。那么，有没有一种方式，允许我们从 gcc 的源代码中抽取想要研究的程序或代码？

有！

本文以 gcov 程序为例，说明如何从 GCC 源代码中抽取 gcov/gcov-dump 程序并编译生成可执行的程序。有了这个独立的 gcov/gcov-dump，研究、调试很方便。想搞清楚 gcc 的内部机理，并非一朝一夕之功，本文只是一种探索，希望对一些想研究 gcc coverage test 的朋友有些帮助。余愿足矣。

本文 gcc 源代码版本为 gcc-4.1.2，其位置在/usr/src/gcc-4.1.2 目录，.表示/usr/src/gcc-4.1.2。

1. gcov

gcov 程序的输入是一个.c 文件，前提是已经编译生成了.gcno 文件并运行可执行程序生成.gcda 文件；gcov 根据.c 文件相应的.gcda 文件和.gcno 文件生成相应的.c.gcov 并报告覆盖率测试结果。

1.1 gcov 必须的文件

(1) 实现文件

根据"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文的讨论，gcov 所需的.c 文件有 gcov.c, gcov-io.c, intl.c, error.c, version.c。

注：gcov-io.c 在编译 gcov 时并没有显示被编译(成.o 文件)，实际上，gcov-io.c 被包含进了 gcov.c 文件中，请参考 gcov.c 代码。

因此，我们需要将这些.c 文件及其.h 文件抽取出来。

(2) 版本文件

gcov-iov.h: 该文件的内容由./gcc/gcov-iov 程序生成。请参考"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"一文。内容如下。

gcov-iov.h

(3) 配置文件

auto-host.h
config.h

其中，
auto-host.h 文件可以使用./gcc/configure 程序自动生成，当然，这里的 auto-host.h 文件只需要包含在 gcov 程序中需要的常量，且有些常量需要修改，内容如下。

auto-host.h

config.h 文件也可以参考./gcc/build/config.h(该文件是在编译 gcc 时自动生成的)，也可自己手写，内容如下。

config.h

(4) 系统文件

以下 4 个文件均是 gcc 的源代码文件，可以直接从 gcc 源代码中拷贝出来。
system.h
safe-ctype.h
hwint.h
filenames.h

但需要对 system.h 做少量的修改：

- 加入如下函数的声明，以通过编译或者消除一些 warning

```
FILE *fopen_unlocked (const char *, const char *);
void unlock_std_streams (void);
void *xcalloc (size_t nelem, size_t elsize);
void *xmalloc (size_t size);
char *xstrdup (const char * s);
```

实际上，从"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文中，可以看出，gcov 对静态库 libiberty.a 的依赖，但 gcov 对该库的依赖仅限于以上几个文件操作或内存操作函数，因 libiberty.a 是被安装到系统的静态库(/usr/lib/libiberty.a)，不如直接用它。因此，在 system.h 文件中，只需加入声明，链接的时候要将 libiberty.a 一起链接。

这一点从 1.2 节的 makefile 文件也能看出。

- 删除一些不必要的包含头文件

删除如下头文件。
/* #include <safe-ctype.h> */
/* #include "libiberty.h" */

修改后的 system.h 文件请参考 http://download.csdn.net/source/3235106。

1.2 如何编译生成 gcov

本文从 gcc 源代码中抽取 gcov 程序，重点是如何编写 makefile 文件。可以参考./gcc/build/makefile 文件中 gcov 程序，或者参考"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文。

笔者编写的 makefile 文件如下。

```
CC = gcc
CXXFLAGS += -g -Wall -Wextra

TARGET = gcov

CLEANUP = rm -f $(TARGET) *.o

all : $(TARGET)

clean :
	$(CLEANUP)
LIBIBERTY = /usr/lib/libiberty.a

gcov.o: gcov.c
	$(CC) $(CXXFLAGS) -c $^
intl.o: intl.c
	$(CC) $(CXXFLAGS) -c $^
version.o: version.c
	$(CC) $(CXXFLAGS) -c $^
errors.o : errors.c
	$(CC) $(CXXFLAGS) -c $^

$(TARGET): version.o errors.o intl.o gcov.o
```

```
    $(CC) $(CXXFLAGS) $^ $(LIBIBERTY) -o $@
```

至此，gcov 程序即从庞大的 GCC 源代码中抽取出来。如果只是研究 gcov 本身，这就足够了。

2. gcov-dump

gcov-dump 是一个 dump 程序，输入是一个 gcov 的文件，或者.gcda，即 gcov 的 data 文件；或者.gcno，即 gcov 的 note 文件。它需要的文件与 gcov 程序唯一不同的是 gcov-dump.c，而 gcov 的实现是 gcov.c。

笔者为 gcov-dump 编写的 makefile 文件如下。

```
CC = gcc
CXXFLAGS += -g -Wall -Wextra

TARGET = gcov-dump

CLEANUP = rm -f $(TARGET) *.o

all : $(TARGET)

clean :
    $(CLEANUP)

LIBIBERTY = /usr/lib/libiberty.a

gcov-dump.o: gcov-dump.c
    $(CC) $(CXXFLAGS) -c $^
version.o: version.c
    $(CC) $(CXXFLAGS) -c $^
errors.o : errors.c
    $(CC) $(CXXFLAGS) -c $^

$(TARGET): version.o errors.o gcov-dump.o
    $(CC) $(CXXFLAGS) $^ $(LIBIBERTY) -o $@
```

3. gcov-tools

抽取并生成 gcov 和 gcov-dump 程序后，笔者发现可将二者结合到一起，这就是 gcov-tools，因为他们需要很多共同的文件。其 makefile 文件如下。

```
CC = gcc
CXXFLAGS += -g -Wall -Wextra

TARGET = gcov gcov-dump

CLEANUP = rm -f $(TARGET) *.o

all : $(TARGET)

clean :
    $(CLEANUP)

LIBIBERTY = /usr/lib/libiberty.a

gcov-dump.o: gcov-dump.c
    $(CC) $(CXXFLAGS) -c $^
gcov.o: gcov.c
    $(CC) $(CXXFLAGS) -c $^
intl.o: intl.c
    $(CC) $(CXXFLAGS) -c $^
version.o: version.c
    $(CC) $(CXXFLAGS) -c $^
errors.o : errors.c
    $(CC) $(CXXFLAGS) -c $^
```

```
all:
gcov: version.o errors.o intl.o gcov.o
     $(CC) $(CXXFLAGS) $^ $(LIBIBERTY) -o $@
gcov-dump: version.o errors.o gcov-dump.o
     $(CC) $(CXXFLAGS) $^ $(LIBIBERTY) -o $@
```

4. 小结

gcov-dump 是一个 dump 程序，输入是一个 gcov 的文件，或者.gcda，即 gcov 的 data 文件；或者.gcno，即 gcov 的 note 文件。

gcov 的输入是一个.c 文件，前提是已经编译生成了.gcno 文件并运行可执行程序生成.gcda 文件；gcov 根据.c 文件相应的.gcda 文件和.gcno 文件生成相应的.c.gcov 并报告覆盖率测试结果。

从 GCC 源代码中抽取 gcov/gcov-dump 相关代码并生成，重点是编写 makefile 文件。

Reference

./gcc/build/makefile
http://blog.csdn.net/livelylittlefish/archive/2011/05/01/6382489.aspx

附：本文代码下载地址

根据本文抽取出的 gcov/gcov-dump 程序代码已经打包并上传到 CSDN 的资源，欢迎下载、指正。
gcov-dump-1.0.tar.gz : http://download.csdn.net/source/3235106
gcov-1.0.tar.gz : http://download.csdn.net/source/3235119
gcov-tools-1.0.tar.gz: http://download.csdn.net/source/3235127

Linux 平台代码覆盖率测试-gcov-dump 原理分析

2011 年 4 月 27 日
15:18

1. 序

gcov 的相关文件.gcda(data 文件)/.gcno(note 文件)文件是以二进制方式写入的(fwrite)，普通编辑文件打开看到的只是乱码，用 ultraedit 打开也只是看到十六进制的数据。如果你了解.gcda/.gcno 的文件格式(可以参考"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析")，看起来会好些；否则，看起来便不知所云，除非有一种工具或程序能将其内容按照有意义的(文件)格式 dump 出来，如果再加上一些提示，就更好了。

——这就是 gcov-dump 程序。

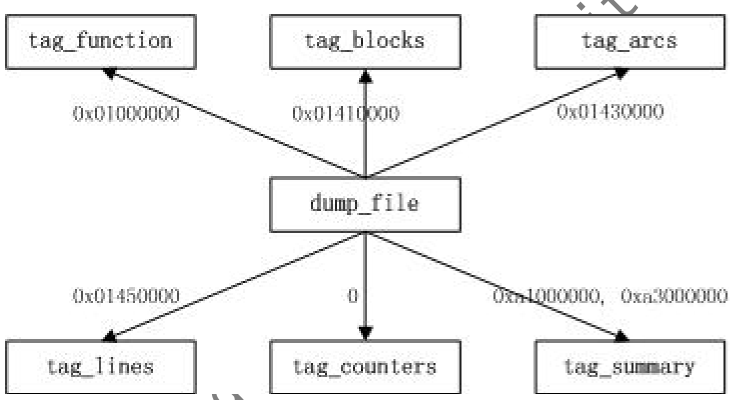
gcov-dump 是一个 dump 程序，输入是一个 gcov 的文件，或者.gcda，即 gcov 的 data 文件；或者.gcno，即 gcov 的 note 文件。

有了"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"和"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"这两篇文章做基础，gcov-dump 的原理就很好理解了。本文不予详细叙述，只做一些代码注释和简单记录，便于用到的时候查询。*好头脑赶不上烂笔头嘛。*

本文例子所用的 gcov-dump 程序来自"Linux 平台代码覆盖率测试-从 GCC 源码中抽取 gcov/gcov-dump 程序"一文。

2. gcov-dump 原理分析

2.1 gcov-dump 程序结构



图中实线表示调用，实线旁边的数字表示 tag 值。tag 的值请参考 gcov_io.h 文件，或者"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"。

2.2 dump_file 函数分析

gcov-dump 程序的主函数 main，是靠调用 dump_file()函数来完成文件内容的输出。该函数定义如下。其中的注释为笔者所加。

```
static void
dump_file (const char *filename)
{
    unsigned tags[4];
    unsigned depth = 0;

    if (! gcov_open (filename, 1))  /* it will open .gcda/.gcno file, and save
information into gcov_var */
    {
```

```
        fprintf (stderr, "%s:cannot open\n", filename);
        return;
    }

    /* magic */
    {
        unsigned magic = gcov_read_unsigned ();
        unsigned version;
        const char *type = NULL;
        int endianness = 0;
        char m[4], v[4];

        /***** compare magic read just now with "gcda" or "gcno" to confirm file type
*/
        if ((endianness = gcov_magic (magic, GCOV_DATA_MAGIC)))
            type = "data";
        else if ((endianness = gcov_magic (magic, GCOV_NOTE_MAGIC)))
            type = "note";
        else
        {
            printf ("%s:not a gcov file\n", filename);
            gcov_close ();
            return;
        }
        /***** read version, an unsigned word */
        version = gcov_read_unsigned ();

        /***** Convert a magic or version number to a 4 character string with ASCII */
        GCOV_UNSIGNED2STRING (v, version);
        GCOV_UNSIGNED2STRING (m, magic);
        printf ("%s:%s:magic `%.4s':version `%.4s'%s\n", filename, type,
                m, v, endianness < 0 ? " (swapped endianness)" : "");
        if (version != GCOV_VERSION)
        {
            char e[4];
            GCOV_UNSIGNED2STRING (e, GCOV_VERSION);
            printf ("%s:warning:current version is `%.4s'\n", filename, e);
        }
    }

    /* stamp */
    {
        unsigned stamp = gcov_read_unsigned ();
        printf ("%s:stamp %lu\n", filename, (unsigned long)stamp);
    }

    while (1)
    {
        gcov_position_t base, position = gcov_position ();
        unsigned tag, length; tag_format_t const *format; unsigned tag_depth;
        int error;
        unsigned mask;

        /***** read a tag, for example, 0x01000000, 0x01a10000, 0xa1000000, etc */
        tag = gcov_read_unsigned ();
        if (! tag) /***** tag=0x00000000, then, to the end of file, break *****/
            break;

        /***** read its length tag */
        length = gcov_read_unsigned ();
        base = gcov_position ();

        /***** for example, tag=0x01000000, then, tag- 1=0xFFFFFF,
          * then, GCOV_TAG_MASK (tag)=0x1FFFFFF, then, mask = 0x1FFFFFF/ 2 = 0xFFFFFF
        */
        mask = GCOV_TAG_MASK (tag) >> 1;

        /****** validate the tag */
        for (tag_depth = 4; mask; mask >>= 8)
        {
            if ((mask & 0xff) != 0xff)
            {
                printf ("%s:tag `%08x' is invalid\n", filename, tag);
                break;
            }
            tag_depth-- ;
        }
```

```
        /***** find the tag in tag_table, if found, then call its procedure */
        for (format = tag_table; format- >name; format++)
            if (format- >tag == tag)
                goto found;
        format = &tag_table[GCOV_TAG_IS_COUNTER (tag) ? 2: 1];
found:
        ;
        if (tag)
        {
            if (depth && depth < tag_depth)
            {
                if (! GCOV_TAG_IS_SUBTAG (tags[depth - 1], tag))
                printf ("%s:tag `%08x' is incorrectly nested\n",
                filename, tag);
            }
            depth = tag_depth;
            tags[depth - 1] = tag;
        }

        /***** print some spaces to represent the depth level */
        print_prefix (filename, tag_depth, position);
        printf ("%08x:%4u:%s", tag, length, format- >name);
        /***** call the procedure of this tag stored in tag_table */
        if (format- >proc)
            (*format- >proc) (filename, tag, length);  //此处调用相应的tag处理函数

        printf ("\n");
        if (flag_dump_contents && format- >proc)
        {
            unsigned long actual_length = gcov_position () - base;
            if (actual_length > length)
                printf ("%s:record size mismatch %lu bytes overread\n",
                    filename, actual_length - length);
            else if (length > actual_length)
                printf ("%s:record size mismatch %lu bytes unread\n",
                    filename, length - actual_length);
        }

        /***** base stands for the base position of a tag, then, synchronize the
pointer */
        gcov_sync (base, length);
        if ((error = gcov_is_error ()))
        {
            printf (error < 0 ? "%s:counter overflow at %lu\n" :
                "%s:read error at %lu\n", filename,
                (long unsigned) gcov_position ());
            break;
        }
    }
    gcov_close ();
}
```

dump_file 函数首先通过 gcov_open 打开.gcda/.gcno 文件，将文件信息保存到全局变量 gcov_var(稍后介绍该变量)，接着读取文件头信息，包括 magic，version，stamp，然后循环读取每个 tag，length，并通过函数指针处理该 tag，直到文件结束(0x00000000)。下面介绍各种 tag 的 callback。

2.3 处理各种 tag 的 callback 定义

处理 tag 的 callback 函数定义如下。

```
static const tag_format_t tag_table[] =
{
    {0,"NOP", NULL},
    {0,"UNKNOWN", NULL},
    {0,"COUNTERS", tag_counters},
    {GCOV_TAG_FUNCTION, "FUNCTION", tag_function},
    {GCOV_TAG_BLOCKS,   "BLOCKS",   tag_blocks},
    {GCOV_TAG_ARCS,     "ARCS",     tag_arcs},
    {GCOV_TAG_LINES,    "LINES",    tag_lines},
    {GCOV_TAG_OBJECT_SUMMARY,  "OBJECT_SUMMARY",  tag_summary},
    {GCOV_TAG_PROGRAM_SUMMARY, "PROGRAM_SUMMARY", tag_summary},
    {0, NULL, NULL}
};
```

其类型 tag_format_t 为一个结构，分别由 tag 本身，tag name 和处理该 tag 的函数指针组成，定义如下。

```
typedef struct tag_format
{
    unsigned    tag;
    char const *name;
    void (*proc) (const char *, unsigned, unsigned);
} tag_format_t;
```

2.4 基本读取函数 gcov_read_words

对.gcda/.gcno 文件的读取/写入，均以 4 字节(1 个 words)为单位进行。下面分析从.gcda/.gcno 文件中读取 words 的基本读取函数 gcov_read_words。代码如下。其中的注释为笔者所加。

```
/ * Return a pointer to read BYTES bytes from the gcov file. Returns
    NULL on failure (read past EOF). */
static const gcov_unsigned_t *
gcov_read_words (unsigned words)
{
    const gcov_unsigned_t *result;

    /**excess is the number of words which can be excessed*/
    unsigned excess = gcov_var.length - gcov_var.offset;

    gcc_assert (gcov_var.mode > 0);
    if (excess < words)
    {
        gcov_var.start += gcov_var.offset;
#if IN_LIBGCOV
        if (excess)
        {
            gcc_assert (excess == 1);
            memcpy (gcov_var.buffer, gcov_var.buffer + gcov_var.offset, 4);
        }
#else
        //在 gcov-dump 程序中，执行 memmove
        memmove (gcov_var.buffer, gcov_var.buffer + gcov_var.offset, excess * 4);
#endif
        gcov_var.offset = 0;
        gcov_var.length = excess;
#if IN_LIBGCOV
        gcc_assert (! gcov_var.length || gcov_var.length == 1);
        excess = GCOV_BLOCK_SIZE;
#else
        //在 gcov-dump 程序中，执行 gcov_allocate
        if (gcov_var.length + words > gcov_var.alloc)
            /** allocate space, the space pointer is saved in gcov_var.buffer */
            gcov_allocate (gcov_var.length + words);
        excess = gcov_var.alloc - gcov_var.length; /** if program can run here, then,
excess = 2050 */
#endif

        /*****
         * >>2, that is, divided by 4, it is for 4 Bytes as a unit.
         * for example, a file with 168B, then, will read 168B, but excess is 168/
4=42.
         * gcov_var.buffer will save the file content.
         */
        excess = fread (gcov_var.buffer + gcov_var.length, 1, excess << 2,
gcov_var.file) >> 2;
        gcov_var.length += excess;
        if (gcov_var.length < words)
        {
            gcov_var.overread += words - gcov_var.length;
            gcov_var.length = 0;
            return 0;
        }
    }

    /***** then, return an unsigned word */
    result = &gcov_var.buffer[gcov_var.offset]; gcov_var.offset += words;
    return result;
}
```

第一次调用该函数时，gcov_var.alloc=0，然后一定会调用 gcov_allocate，调用 gcov_allocate 后，gcov_var.alloc=2050。跟踪执行发现，第一次调用 fread 之前，excess = gcov_var.alloc - gcov_var.length = 2050，调用 fread 后，仍以 test.c 产生的 test.gcda 为例(可参考前面的文章)，excess=168/4=42。因为 test.gcda 较小，只有 168 字节，故调用 fread 后，gcov_var.buffer 中就存放了整个文件的内容(168 字节)，如下所示，虽然为 gcov_var.buffer 分配了 8200 自己的空间。

```
(gdb) p gcov_var
$1 = {
  file = 0x810c008,        //文件指针
  start = 0,
  offset = 0,
  length = 0,
  overread = 4294967295,   //4294967295=0xffffffff=-1
  error = 0,
  mode = 1,
  endian = 0,
  alloc = 2050,
  buffer = 0x810c170
}
(gdb) x /42w 0x810c170     //查看 buffer 中的前 42 个字，共 168 字节，就是 test.gcda 文件的内容
0x810c170:      0x67636461      0x34303170      0xc5ecae39      0x01000000
0x810c180:      0x00000002      0x00000003      0xeb65a768      0x01a10000
0x810c190:      0x0000000a      0x0000000a      0x00000000      0x00000000
0x810c1a0:      0x00000000      0x00000001      0x00000000      0x00000000
0x810c1b0:      0x00000000      0x00000001      0x00000000      0xa1000000
0x810c1c0:      0x00000009      0x00000000      0x00000005      0x00000001
0x810c1d0:      0x0000000c      0x00000000      0x0000000a      0x00000000
0x810c1e0:      0x0000000a      0x00000000      0xa3000000      0x00000009
0x810c1f0:      0x51924f98      0x00000005      0x00000001      0x0000000c
0x810c200:      0x00000000      0x0000000a      0x00000000      0x0000000a
0x810c210:      0x00000000      0x00000000
```

其中，前 3 个字(4 字节/字)即为 magic, version, stamp；蓝色部分即为 tag，可以参考"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文的 3.3 和 3.4 节，也可以参考本文第 3 节。

为什么为 gcov_var.buffer 分配了 8200 字节的空间？
——这就是 gcov_allocate 完成的。

2.5 分配空间函数 gcov_allocate

代码如下。其中的注释为笔者加入。

```
#if ! IN_LIBGCOV
static void
gcov_allocate (unsigned length)
{
    size_t new_size = gcov_var.alloc;

    if (! new_size)
        new_size = GCOV_BLOCK_SIZE; /***** if new_size==0, then, new_size=1024(GCOV_BLOCK_SIZE=1024) */
    new_size += length;  /***** if length==1, then, new_size=1025 */
    new_size *= 2;       /***** then, new_size=1025*2=2050 */

    gcov_var.alloc = new_size;
    gcov_var.buffer = xrealloc (gcov_var.buffer, new_size << 2); /***** size=1025*4=8200 */
}
#endif
```

实际上 gcov_var.alloc 是一个内存 block，以 4 字节为一个单位。由代码及其注释可以看出，当 length=1 时，gcov_var.alloc=2050，调用 gcov_allocate 后，实际上分配了 2050*4=8200 个字节的空间给 gcov_var.buffer。

此处，不得不介绍一下 gcov_var。

2.6 重要数据结构 gcov_var

gcov_var 是个全局变量，其作用就是在 gcov/gcov-dump 程序运行期间保存操作的文件信息，例如，文件指针、某个 block 的 start/offset/length、文件内容 buffer 等信息，定义如下。

```
/* Optimum number of gcov_unsigned_t's read from or written to disk. */
#define GCOV_BLOCK_SIZE (1<< 10)

GCOV_LINKAGE struct gcov_var
{
    FILE *file;
    gcov_position_t start;    /* Position of first byte of block */
    unsigned offset;          /* Read/ write position within the block. */
    unsigned length;          /* Read limit in the block. */
    unsigned overread;        /* Number of words overread. */
    int error;                /* < 0 overflow, > 0 disk error. */
    int mode;                 /* < 0 writing, > 0 reading */
#if IN_LIBGCOV
    /* Holds one block plus 4 bytes, thus all coverage reads & writes
      fit within this buffer and we always can transfer GCOV_BLOCK_SIZE
      to and from the disk. libgcov never backtracks and only writes 4 or 8 byte
     objects. */
    gcov_unsigned_t buffer[GCOV_BLOCK_SIZE + 1];
#else
    int endian;               /* Swap endianness. */
    /* Holds a variable length block, as the compiler can write strings and needs to
backtrack. */
    size_t alloc;
    gcov_unsigned_t *buffer;
#endif
}gcov_var ATTRIBUTE_HIDDEN;
```

在 gcov-dump 程序中，sizeof(gcov_type)=sizeof(gcov_unsigned_t)=4，sizeof(gcov_var)=40。gcov_var 的值一个例子可以参考 2.4 节，此处不再赘述。

3. 处理 tag 的 callback 分析

3.1 FUNCTION tag: tag_function()函数

```
static void
tag_function (const char *filename ATTRIBUTE_UNUSED,
      unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    unsigned long pos = gcov_position ();

    /***** for function, it will print ident and checksum */
    printf (" ident=%u", gcov_read_unsigned ());
    printf (", checksum=0x%08x", gcov_read_unsigned ());

    if (gcov_position () - pos < length)  //一般对于.gcno 文件该条件才满足，才能执行该 clause
    {
        const char *name;
        name = gcov_read_string ();  //该函数读取 length(4 字节)和 length 个 words(4*length 字节)，读取函数名字
        printf (", `%s'", name ? name : "NULL");
        name = gcov_read_string ();  //读取源文件名字
        printf (" %s", name ? name : "NULL");
        printf (":%u", gcov_read_unsigned ());  //读取该函数在该源文件中的行号
    }
}
```

输出格式：
.gcda 文件输出：ident=3, checksum=0xeb65a768
.gcno 文件输出：ident=3, checksum=0xeb65a768, `main' test.c:4
其中，划线部分分别为：函数名 源文件名:行号

3.2 BLOCKS tag: tag_blocks()函数

```
static void
tag_blocks (const char *filename ATTRIBUTE_UNUSED,
     unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    unsigned n_blocks = GCOV_TAG_BLOCKS_NUM (length);
    printf (" %u blocks", n_blocks);

    if (flag_dump_contents)
    {
        unsigned ix;
        for (ix = 0; ix != n_blocks; ix++)
        {
            if (! (ix & 7))  //如果 blocks 较多，则每 8 个为 1 行输出，且按 0，8，16，...输出序号
            {
                printf ("\n");
                print_prefix (filename, 0, gcov_position ());
                printf ("\t\t%u", ix);  //输出序号
            }
            printf (" %04x", gcov_read_unsigned ());
        }
    }
}
```

其中，*flag_dump_contents* 是打印开关，*flag_dump_contents* 非 0 时将打印 COUNTER 的内容。
输出格式：

```
    n blocks
    0 block0 block1 ... block7   //每 8 个 1 行输出，前面的 0 表示序号
    8 block8 block9 ... block15  //前面的 8 表示序号
    ...
```

当然，需要注意前导符或者输出位置。

3.3 ARCS tag: tag_arcs()函数

```
static void
tag_arcs (const char *filename ATTRIBUTE_UNUSED,
     unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    unsigned n_arcs = GCOV_TAG_ARCS_NUM (length);
    printf (" %u arcs", n_arcs);  //输出提示信息，几个 arcs

    if (flag_dump_contents)
    {
        unsigned ix;
        unsigned blockno = gcov_read_unsigned ();

        for (ix = 0; ix != n_arcs; ix++)
        {
            unsigned dst, flags;
            if (! (ix & 3))  //如果 arcs 较多，则每 4 个为 1 行输出，且每行前面输出 blockno
            {
                printf ("\n");
                print_prefix (filename, 0, gcov_position ());
                printf ("\tblock %u:", blockno);
            }

            dst = gcov_read_unsigned ();   //读取目的 blockno
            flags = gcov_read_unsigned ();
            printf (" %u:%04x", dst, flags);
        }
    }
}
```

输出格式：

```
    n arcs   //n 个 arcs，每 4 个为 1 行输出
    blockno: dst0:flags0 dst1:flags1 dst2:flags2 dst3:flags3
    blockno: dst4:flags4 dst5:flags5 dst6:flags6 dst7:flags7
```

```
    ...
```

同上，需要注意前导符或者输出位置。dst0, ..., 表示目的 block 的 blockno。

3.4 LINES tag: tag_lines()函数

```
static void
tag_lines (const char *filename ATTRIBUTE_UNUSED,
     unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    if (flag_dump_contents)
    {
        unsigned blockno = gcov_read_unsigned ();
        char const *sep = NULL;

        while (1)
        {
            gcov_position_t position = gcov_position ();
            const char *source = NULL;
            unsigned lineno = gcov_read_unsigned ();

            if (! lineno)  //lineno=0 时才会执行该 clause，因此 lineno=0 即为以后的新的文件的标志
            {
                source = gcov_read_string ();  //该函数读取 length(4 字节)和 length 个
words(4*length 字节)
                if (! source)                  //source 即为文件名，没有源文件了，就退出
                    break;
                sep = NULL;
            }

            if (! sep)  //sep=NULL 才会执行该 clause，那么什么时候会为 NULL 呢？——就是新的文件开
始，实际上就是 lineno=0
            {
                printf ("\n");
                print_prefix (filename, 0, position);
                printf ("\tblock %u:", blockno);
                sep = "";
            }

            if (lineno)
            {
                printf ("%s%u", sep, lineno);
                sep = ", ";
            }
            else
            {
                printf ("%s`%s'", sep, source);  //lineno=0 时，输出该文件名，之后 sep=":"
                sep = ":";
            }
        }
    }
}
```

输出格式：

```
    block blockno:'filename':lineno1, lineno2, ...
```

例如：block 1:'test.c':4, 7, 9
其中，前面的 block 为提示信息。同上，需要注意前导符或者输出位置。

3.5 COUNTER tag: tag_counters()函数

```
static void
tag_counters (const char *filename ATTRIBUTE_UNUSED,
     unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    static const char *const counter_names[] = GCOV_COUNTER_NAMES;
    unsigned n_counts = GCOV_TAG_COUNTER_NUM (length);

    printf (" %s %u counts",
        counter_names[GCOV_COUNTER_FOR_TAG (tag)], n_counts);
    if (flag_dump_contents)
    {
```

```
        unsigned ix;

        for (ix = 0; ix != n_counts; ix++)
        {
            gcov_type count;
            if (! (ix & 7))  //如果 counter 较多，则每 8 个 1 行输出，且按 0，8，16，...输出序号
            {
                printf ("\n");
                print_prefix (filename, 0, gcov_position ());
                printf ("\t\t%u", ix);  //输出序号
            }

            count = gcov_read_counter ();  //读取该 counter，读取 8 字节，但返回 4 字节
            printf (" ");
            printf (HOST_WIDEST_INT_PRINT_DEC, count);
        }
    }
}
```

关于 **GCOV_TAG_COUNTER_NUM** 和 **GCOV_COUNTER_FOR_TAG**，请参考源代码。

counter 的名字定义如下。

```
/* A list of human readable names of the counters */
#define GCOV_COUNTER_NAMES        {"arcs", "interval", "pow2", "single", "delta"}
```

输出格式：

```
    arcs n counts  //arcs 即为 counter 的名字，如上
    0 counter0 counter1 ... counter7   //每 8 个 1 行输出，前面的 0 表示序号
    8 counter8 counter9 ... counter15  //前面的 8 表示序号
    ...
```

同上，需要注意前导符或者输出位置。

3.6 OBJECT/PROGRAM SUMMARY tag: tag_summary()函数

```
static void
tag_summary (const char *filename ATTRIBUTE_UNUSED,
     unsigned tag ATTRIBUTE_UNUSED, unsigned length ATTRIBUTE_UNUSED)
{
    struct gcov_summary summary;
    unsigned ix;

    /***** initialize all members with 0 *****/
    memset(&summary, 0, sizeof(summary));

    unsigned count = gcov_read_summary (&summary);  //读取该 summary
    printf (" checksum=0x%08x", summary.checksum);

    /* for (ix = 0; ix ! = GCOV_COUNTERS; ix++) */  /* 原来的代码 */
    for (ix = 0; ix < count; ix++)                  /* 应该如此修改*/
    {
        printf ("\n");
        print_prefix (filename, 0, 0);
        printf ("\t\tcounts=%u, runs=%u", summary.ctrs[ix].num, summary.ctrs[ix].runs);
        printf (", sum_all=" HOST_WIDEST_INT_PRINT_DEC,
(HOST_WIDEST_INT)summary.ctrs[ix].sum_all);
        printf (", run_max=" HOST_WIDEST_INT_PRINT_DEC,
(HOST_WIDEST_INT)summary.ctrs[ix].run_max);
        printf (", sum_max=" HOST_WIDEST_INT_PRINT_DEC,
(HOST_WIDEST_INT)summary.ctrs[ix].sum_max);
    }
}
```

输出格式：

```
    checksum=0x51924f98
    counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
```

同上，也需要注意前导符或者输出位置。

其中，gcov_read_summary 函数是修改后的函数，在"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文没有列出该修改后的函数，其实这篇文章中的 bug 与该函数有关。此处列出其代码。

```
GCOV_LINKAGE unsigned
gcov_read_summary (struct gcov_summary *summary)
{
    unsigned ix;
    struct gcov_ctr_summary *csum;

    summary- >checksum = gcov_read_unsigned (); /***** checksum is a words (4Bytes)
*****/

    /***** that is, a summry is 32Bytes (sizeof(gcov_type)=4) or 20Bytes
(sizeof(gcov_type)=8) *****/
    for (csum = summary- >ctrs, ix = GCOV_COUNTERS_SUMMABLE; ix-- ; csum++)
    {
        csum- >num = gcov_read_unsigned (); /***** 4Bytes *****/
        csum- >runs = gcov_read_unsigned (); /***** 4Bytes *****/
        csum- >sum_all = gcov_read_counter (); /***** 8Bytes, but return 4Bytes *****/
        csum- >run_max = gcov_read_counter (); /***** 8Bytes, but return 4Bytes *****/
        csum- >sum_max = gcov_read_counter (); /***** 8Bytes, but return 4Bytes *****/
    }

    return GCOV_COUNTERS_SUMMABLE; /* zubo modified to return the hubmer */
}
```

gcov_summary 及 gcov_ctr_summary 结构的定义可参考源代码或者"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"。

4. 小结

本文详细叙述了 gcov-dump 程序的结构和实现原理，也从中学习了其处理各种 tag 用到的 callback 方法，如果你想深入跟踪或学习 gcc 源码，请注意 callback 的使用，因为 gcc 源码中大量地使用了 callback。

Reference

http://blog.csdn.net/livelylittlefish/archive/2011/04/13/6321909.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/01/6382489.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/26/6448799.aspx

Linux 平台代码覆盖率测试-.gcda/.gcno 文件及其格式分析

2011 年 5 月 25 日
19:33

0. 序

在"Linux 平台代码覆盖率测试-gcov-dump 原理分析"一文中，我们详细分析了 gcov-dump 程序的实现原理及每种 tag 的输出格式，本文，仍然以前面几篇文章的 test.c 为例，说明 gcov-dump 程序的输出结果，并总结.gcda/.gcno 文件格式。

1. .gcda 文件分析

1.1 gcov-dump 程序输出结果

以下 dump 结果请参考"Linux 平台代码覆盖率测试-GCC 如何编译生成 gcov/gcov-dump 程序及其 bug 分析"一文的 3.3 和 3.4 节。

```
# /home/zubo/gcc/2011-04-11.gcov-dump/gcov-dump test.gcda
test.gcda:data:magic `gcda':version `401p'
test.gcda:stamp 3320622649  //对应下面的 0xc5ecae39
test.gcda: 01000000:   2:FUNCTION ident=3, checksum=0xeb65a768 //tag, length=2, ident,
checksum
test.gcda:  01a10000:  10:COUNTERS arcs 5 counts  //tag, length=10, 5 个 COUNTERS
test.gcda:              0 10 0 1 0 1  //此处便是 5 个 counter，共 40 字节
test.gcda: a1000000:   9:OBJECT_SUMMARY checksum=0x00000000
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
test.gcda: a3000000:   9:PROGRAM_SUMMARY checksum=0x51924f98
test.gcda:              counts=5, runs=1, sum_all=12, run_max=10, sum_max=10
```

输出格式可以参考源代码，或者"Linux 平台代码覆盖率测试-gcov-dump 原理分析"。

1.2 文件实际内容

文件的二进制内容如下，对应以上输出，更清楚。

```
# od -t x4 -w16 test.gcda
0000000 67636461 34303170 c5ecae39 01000000  //'gcda', '401p', timestamp,
tag=0x01000000
0000020 00000002 00000003 eb65a768 01a10000  //length=2, ident=3, checksum, 0x01a10000
0000040 0000000a 0000000a 00000000 00000000  //length=0xa=10, counter content: 0xa, 0,
1, 0, 1
0000060 00000000 00000001 00000000 00000000  //8 Bytes for each counter
0000100 00000000 00000001 00000000 a1000000  //                          ,
tag=0xa1000000
0000120 00000009 00000000 00000005 00000001  //length=9, checksum=0, counts=5, runs=1
0000140 0000000c 00000000 0000000a 00000000  //sum_all=0xc=12(8 Bytes),
run_max=0xa=10(8 Bytes)
0000160 0000000a 00000000 a3000000 00000009  //sum_max=0xa=10(8 Bytes), tag=a3000000,
length=9
0000200 51924f98 00000005 00000001 0000000c  //same as above
0000220 00000000 0000000a 00000000 0000000a
0000240 00000000 00000000
0000250
```

格式信息可以参考源代码，也可以参考"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"。

1.3 文件格式总结

在写入/读取文件时均以 4 字节为单位，下面的分析如不特别注明，每个数据均为 4 字节。

(0) file header 格式

magic='gcda', version, stamp

(1) FUNCTION 格式

tag=0x01000000, length, ident, checksum

(2) COUNTERS 格式

tag=0x01a10000, length, **counter1, counter2, ..., countern**

其中，划线部分均为 8 字节，其他为 4 字节。另外，n=length/2。

(3) OBJECT/PROGRAM SUMMARY 格式

tag=0xa1000000/0xa3000000, length, checksum=0, counts, runs, **sum_all, run_max, sum_max**

其中，划线部分均为 8 字节，其他为 4 字节。

2. .gcno 文件分析

2.1 gcov-dump 程序输出结果

其中的空行和//注释为笔者所加。

```
# /home/zubo/gcc/2011-04-11.gcov-dump/gcov-dump test.gcno

//magic:version，和 stamp，对应下面的 0xc5ecae39，与 test.gcda 一一对应
test.gcno:note:magic `gcno':version `401p'
test.gcno:stamp 3320622649
test.gcno: 01000000:   9:FUNCTION ident=3, checksum=0xeb65a768, `main' test.c:4
       //: tag=0x01000000,length=9,tagname=FUNCTION,function 的信息(ident,checksum,函数名,
文件名,行号)

//以下为 9 个 BLOCKS 记录
test.gcno:  01410000:   9:BLOCKS 9 blocks
test.gcno:              0 0000 0000 0000 0000 0000 0000 0000 0000  //0 为序号，每 8 个
blocks 为一行
test.gcno:              8 0000                                     //8 为序号，一共 9 个

//以下为 8 个 ARCS 记录，小写的 arcs 和 block 为提示信息，大写的 ARCS 为 tag 名字
test.gcno:  01430000:   3:ARCS 1 arcs  //tag=0x01430000:length=3:tagname=ARCS n_arcs=1,
格式下同
test.gcno:      block 0: 1:0005        //blockno=0:dst=1:flags=0005
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:      block 1: 3:0005
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:      block 2: 3:0005
test.gcno:  01430000:   5:ARCS 2 arcs  //2 个 arcs
test.gcno:      block 3: 2:0000 4:0005 //有两个目的地，格式：blockno=3: dst1=2:flags1
dst2=4:flags2
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:      block 4: 5:0004 6:0000
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:      block 5: 7:0004 8:0003
test.gcno:  01430000:   5:ARCS 2 arcs
test.gcno:      block 6: 7:0004 8:0003
test.gcno:  01430000:   3:ARCS 1 arcs
test.gcno:      block 7: 8:0001

//以下为 6 个 LINES 记录，小写的 block 为提示信息，大写的 LINES 为 tag 名字
test.gcno:  01450000:  10:LINES          //tag=0x01450000:length=10:tagname=LINES
test.gcno:      block 1:`test.c':4, 7, 9  //blockno=1:'文件名
':lineno1=4,lineno2=7,lineno3=9
test.gcno:  01450000:   9:LINES
test.gcno:      block 2:`test.c':10, 9
test.gcno:  01450000:   8:LINES
test.gcno:      block 4:`test.c':12
test.gcno:  01450000:   8:LINES
test.gcno:      block 5:`test.c':13
test.gcno:  01450000:   8:LINES
test.gcno:      block 6:`test.c':15
```

```
test.gcno:  01450000:   8:LINES
test.gcno:      block 7:`test.c':16
```

输出格式可以参考源代码，或者"Linux 平台代码覆盖率测试-gcov-dump 原理分析"。

2.2 文件实际内容

test.gcno 文件的实际内容，使用 od 命令输出，如下。

```
# od -t x4 -w16 test.gcno
0000000 67636e6f 34303170 c5ecae39 01000000  //magic="gcno", version="401p", stamp,
tag=0x01000000
0000020 00000009 00000003 eb65a768 00000002  //length, ident, checksum, length=2
0000040 6e69616d 00000000 00000002 74736574  //functionname="niam"(8Bytes), length=2,
filename=
0000060 0000632e 00000004 01410000 00000009  //"test.c"(8Bytes), lineno=4,
tag=0x01410000, length=9
0000100 00000000 00000000 00000000 00000000  //9 blocks' content, all is 0
*                                            //* represents all 0 repeated (reference
'man od')
0000140 00000000 01430000 00000003 00000000  //tag=0x01430000, length=3, src=0, dest=1,
flags=5
0000160 00000001 00000005 01430000 00000003  //tag=0x01430000, length=3, src=1, dest=3,
flags=5
0000200 00000001 00000003 00000005 01430000  //tag=0x01430000, length=3, src=2, dest=3,
flags=5
0000220 00000003 00000002 00000003 00000005
0000240 01430000 00000005 00000003 00000002  //tag=0x01430000, length=5, src=3,
dest1=2, flags1=0
0000260 00000000 00000004 00000005 01430000  //
dest2=4, flags2=5
0000300 00000005 00000004 00000005 00000004  //tag=0x01430000, length=5, src=4,
dest1=5, flags1=4
0000320 00000006 00000000 01430000 00000005  //
dest2=6, flags2=0
0000340 00000005 00000007 00000004 00000008  //tag=0x01430000, length=5, src=5,
dest1=7, flags1=4
0000360 00000003 01430000 00000005 00000006  //
dest2=8, flags2=3
0000400 00000007 00000004 00000008 00000003  //tag=0x01430000, length=5, src=6,
dest1=7, flags1=4
                                             //
dest2=8, flags2=3
0000420 01430000 00000003 00000007 00000008  //tag=0x01430000, length=3, src=7, dest=8,
flags=1
0000440 00000001 01450000 0000000a 00000001  //tag=0x01450000, length=10, blockno=1
0000460 00000000 00000002 74736574 0000632e  //lineno=0, length=2, filename="test.c"
0000500 00000004 00000007 00000009 00000000  //lineno=4, lineno=7, lineno=9, lineno=0
0000520 00000000 01450000 00000009 00000002  //lineno=0, tag=0x01450000, length=9,
blockno=2
0000540 00000000 00000002 74736574 0000632e  //lineno=0, length=2, filename="test.c"
0000560 0000000a 00000009 00000000 00000000  //lineno=10, lineno=9, lineno=0, lineno=0
0000600 01450000 00000008 00000004 00000000  //tag=0x01450000, length=8, blockno=4,
lineno=0
0000620 00000002 74736574 0000632e 0000000c  //length=2, filename="test.c", lineno=12
0000640 00000000 00000000 01450000 00000008  //lineno=0, lineno=0, tag=0x01450000,
length=8
0000660 00000005 00000000 00000002 74736574  //blockno=5, lineno=0, length=2,
filename="test.c"
0000700 0000632e 0000000d 00000000 00000000  //           lineno=13, lineno=0, lineno=0
0000720 01450000 00000008 00000006 00000000  //tag=0x01450000, length=8, blockno=6,
lineno=0
0000740 00000002 74736574 0000632e 0000000f  //length=2, filename="test.c", lineno=15
0000760 00000000 00000000 01450000 00000008  //lineno=0, lineno=0, tag=0x01450000,
length=8
0001000 00000007 00000000 00000002 74736574  //blockno=7, lineno=0, length=2,
filename="test.c"
0001020 0000632e 00000010 00000000 00000000  //         , lineno=16, lineno=0, lineno=0
0001040
```

格式信息可以参考源代码，也可以参考"Linux 平台代码覆盖率测试工具 GCOV 相关文件分析"。

注：
(1) 粗体蓝色为 tag。
(2) 00000000 6e69616d: 'n', 'i', 'a', 'm' => 函数名 main

(3) 0000632e 74736574: 'c', '.', 't', 's', 'e', 't' => 文件名 test.c
(4) *表示该行与上一行完全相同，即 line suppression，行隐藏/行压缩(可参考 od 命令的 manual 页)。使用如下命令可以消除，即按文件真实内容显示。
od -t x4 -w16 -v test.gcno
(5) Function name 和 filename 由 gcov_read_string()函数完成，在该函数中，先读取 length，然后在读取 length 个字(4 字节/字)。

2.3 文件格式总结

在写入/读取文件时均以 4 字节为单位，下面的分析如不特别注明，每个数据均为 4 字节。

(0) file header 格式

magic='gcda', version, stamp

(1) FUNCTION 格式

tag=0x01000000, length, ident, checksum, length1, function name, length2, filename, lineno

其中，length1 为其后的 function name 的长度，function name 的大小为 4*length1 字节。length2 为其后的 filename 的长度，filename 的大小为 4*length2 字节。filename 即为该 function 所在的源文件的名字。

(2) BLOCKS 格式

tag=0x01410000, length, block1, block2, ..., blockn

(3) ARCS 格式

tag=0x01430000, length, src, dst1, flags1, dst2, flags2, ..., dstn, flagsn

一个 arc 的 src block 对应多个 dst block，且每个 dst block 均有对应的 flags。

(4) LINES 格式

tag=0x01450000, length, blockno, linenoa=0, lengtha>0, filename, lineno1, lineno2, ..., linenon, linenob=0, lengthb=0

其中，
linenoa=0 表明源文件信息的开始，lengtha(一定大于 0)为其后的 filename 的长度，filename 的大小为 4*lengtha。
linenob=0 表明该源文件信息的结束，且结束信息为 8 个字节的 0，其中前 4 个字节是 linenob=0，后 4 个字节是 lengthb=0。
中间是 lineno 信息，有 1 个或多个 lineno，且每个 lineno 不为 0。

3. 小结

本文在"Linux 平台代码覆盖率测试-gcov-dump 原理分析"一文基础上，说明 gcov-dump 程序的输出结果，并总结.gcda/.gcno 文件格式。至此，对 Linux 平台代码覆盖率测试工具 GCOV 相关文件的构成，有较深的认识。

Reference

http://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/Gcov-Data-Files.html
http://blog.csdn.net/livelylittlefish/archive/2011/04/13/6321909.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/01/6382489.aspx

Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析

2011 年 5 月 6 日
14:19

0. 序

在"Linux 平台代码覆盖率测试-GCC 插桩基本概念和原理分析"一文中，我们已经知道，GCC 插桩乃汇编级的插桩，那么，本文仍然以 test.c 为例，来分析加入覆盖率测试选项"-fprofile-arcs -ftest-coverage"前后，即插桩前后汇编代码的变化。本文所用 gcc 版本为 gcc-4.1.2。test.c 代码如下。

```
/**
 * filename: test.c
 */
#include <stdio.h>

int main (void)
{
    int i, total;

    total = 0;

    for (i = 0; i < 10; i++)
        total += i;

    if (total != 45)
        printf ("Failure\n");
    else
        printf ("Success\n");
    return 0;
}
```


1. 如何编译

1.1 未加入覆盖率测试选项

```
# cpp test.c -o test.i   //预处理：生成 test.i 文件，或者"cpp test.c > test.i"
或者
# gcc -E test.c -o test.i
# gcc -S test.i          //编译：生成 test.s 文件(未加入覆盖率测试选项)
# as -o test.o test.s    //汇编：生成 test.o 文件，或者"gcc -c test.s -o test.o"
# gcc -o test test.o     //链接：生成可执行文件 test
```

以上过程可参考
http://blog.csdn.net/livelylittlefish/archive/2009/12/30/5109300.aspx。

查看 test.o 文件中的符号

```
# nm test.o
00000000 T main
         U puts
```

1.2 加入覆盖率测试选项

```
# cpp test.c -o test.i                                  //预处理：生成 test.i 文件
# gcc -fprofile-arcs -ftest-coverage -S test.i  //编译：生成 test.s 文件(加入覆盖率
测试选项)
# as -o test.o test.s                                   //汇编：生成 test.o 文件
# gcc -o test test.o                                    //链接：生成可执行文件 test
```

查看 test.o 文件中的符号

```
# nm test.o
000000eb t _GLOBAL__I_0_main
         U __gcov_init
         U __gcov_merge_add
00000000 T main
         U puts
```

1.3 分析

从上面 nm 命令的结果可以看出，加入覆盖率测试选项后的 test.o 文件，多了 3 个符号，如上。其中，_GLOBAL__I_0_main 就是插入的部分桩代码。section2 和 section3 将对比分析插桩前后汇编代码的变化，section3 重点分析插入的桩代码。

2．未加入覆盖率测试选项的汇编代码分析

采用"# gcc -S test.i"命令得到的 test.s 汇编代码如下。#后面的注释为笔者所加。

```
    .file    "test.c"
    .section    .rodata
.LC0:
    .string    "Failure"
.LC1:
    .string    "Success"
    .text
.globl main
    .type    main, @function
main:
    leal    4(%esp), %ecx     #这几句就是保护现场
    andl    $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ecx
    subl    $20, %esp

    movl    $0, -8(%ebp)      #初始化 total=0,total 的值在-8(%ebp)中
    movl    $0, -12(%ebp)     #初始化循环变量 i=0,i 的值在-12(%ebp)中
    jmp    .L2
.L3:
    movl    -12(%ebp), %eax   #将 i 的值移到%eax 中，即%eax=i
    addl    %eax, -8(%ebp)    #将%eax 的值加到-8(%ebp)，total=total+i
    addl    $1, -12(%ebp)     #循环变量加 1，即 i++
.L2:
    cmpl    $9, -12(%ebp)     #比较循环变量 i 与 9 的大小
    jle    .L3                #如果 i<=9，跳到.L3，继续累加
    cmpl    $45, -8(%ebp)     #否则，比较 total 的值与 45 的大小
    je     .L5                #若 total=45，跳到.L5
    movl    $.LC0, (%esp)     #否 total 的值不为 45，则将$.LC0 放入%esp
    call    puts              #输出 Failure
    jmp    .L7                #跳到.L7
.L5:
    movl    $.LC1, (%esp)     #将$.LC1 放入%esp
    call    puts              #输出 Success
.L7:
    movl    $0, %eax          #返回值 0 放入%eax

    addl    $20, %esp         #这几句恢复现场
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret

    .size    main, .-main
    .ident    "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
    .section    .note.GNU-stack,"",@progbits
```


注：$9 表示常量 9，即立即数(Immediate Operand)。-8(%ebp)即为 total，-12(%ebp)即是循环变量 i。

3. 加入覆盖率测试选项的汇编代码分析

采用"# gcc -fprofile-arcs -ftest-coverage **-S** test.i"命令得到的 test.s 汇编代码如下。前面的蓝色部分及后面的.LC2, .LC3, .LPBX0, _GLOBAL__I_0_main 等均为插入的桩代码。#后面的注释为笔者所加。

```
    .file       "test.c"
    .section    .rodata
.LC0:
    .string     "Failure"
.LC1:
    .string     "Success"
    .text
.globl main
    .type    main, @function
main:
    leal    4(%esp), %ecx     #这几句就是保护现场
    andl    $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ecx
    subl    $20, %esp

    movl    $0, -8(%ebp)      #初始化 total=0,total 的值在-8(%ebp)中
    movl    $0, -12(%ebp)     #初始化循环变量 i=0,i 的值在-12(%ebp)中
    jmp    .L2

.L3:                          #以下这几句就是插入的桩代码
    movl    .LPBX1, %eax      #将.LPBX1 移到%eax，即%eax=.LPBX1
    movl    .LPBX1+4, %edx    #edx=.LPBX1+4
    addl    $1, %eax          #eax=%eax+1
    adcl    $0, %edx          #edx=%edx+0
    movl    %eax, .LPBX1      #将%eax 移回.LPBX1
    movl    %edx, .LPBX1+4    #将%edx 移回.LPBX1+4

    movl    -12(%ebp), %eax   #将 i 的值移到%eax 中，即%eax=i
    addl    %eax, -8(%ebp)    #将%eax 的值加到-8(%ebp)，total=total+i
    addl    $1, -12(%ebp)     #循环变量加 1，即 i++

.L2:
    cmpl    $9, -12(%ebp)     #比较循环变量 i 与 9 的大小
    jle    .L3                #如果 i<=9，跳到.L3，继续累加
    cmpl    $45, -8(%ebp)     #否则，比较 total 的值与 45 的大小
    je     .L5                #若 total=45，跳到.L5

    #以下也为桩代码
    movl    .LPBX1+8, %eax    #eax=.LPBX1+8
    movl    .LPBX1+12, %edx   #edx=.LPBX1+12
    addl    $1, %eax          #eax=%eax+1
    adcl    $0, %edx          #edx=%edx+0
    movl    %eax, .LPBX1+8    #将%eax 移回.LPBX1+8
    movl    %edx, .LPBX1+12   #将%eax 移回.LPBX1+12

    movl    $.LC0, (%esp)     #否 total 的值不为 45，则将$.LC0 放入%esp
    call    puts              #输出 Failure

    #以下也为桩代码，功能同上，不再解释
    movl    .LPBX1+24, %eax
    movl    .LPBX1+28, %edx
    addl    $1, %eax
    adcl    $0, %edx
    movl    %eax, .LPBX1+24
    movl    %edx, .LPBX1+28
```

```
    jmp     .L7                 #跳到.L7
.L5:
    #以下也为桩代码，功能同上，不再解释
    movl    .LPBX1+16, %eax
    movl    .LPBX1+20, %edx
    addl    $1, %eax
    adcl    $0, %edx
    movl    %eax, .LPBX1+16
    movl    %edx, .LPBX1+20

    movl    $.LC1, (%esp)       #将$.LC1 放入%esp
    call    puts                #输出 Success

    #以下也为桩代码，功能同上，不再解释
    movl    .LPBX1+32, %eax
    movl    .LPBX1+36, %edx
    addl    $1, %eax
    adcl    $0, %edx
    movl    %eax, .LPBX1+32
    movl    %edx, .LPBX1+36

.L7:
    movl    $0, %eax            #返回值 0 放入%eax
    addl    $20, %esp           #这几句回复现场
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret

    .size    main, .-main

    #以下部分均是加入 coverage 选项后编译器加入的桩代码

    .local   .LPBX1
    .comm    .LPBX1,40,32

    .section .rodata        #只读 section
    .align   4
.LC2:                       #文件名常量，只读
    .string  "/home/zubo/gcc/test/test.gcda"

    .data                   #data 数据段
    .align   4
.LC3:
    .long    3              #ident=3
    .long    -1345659544    #即 checksum=0xeb65a768
    .long    5              #counters

    .align   32
    .type    .LPBX0, @object #.LPBX0 是一个对象
    .size    .LPBX0, 52     #.LPBX0 大小为 52 字节
.LPBX0:                     #结构的起始地址，即结构名，该结构即为 gcov_info 结构
    .long    875573616      #即 version=0x34303170，即版本为 4.1p
    .long    0              #即 next 指针，为 0
    .long    -979544300     #即 stamp=0xc59d5714
    .long    .LC2           #filename，值为.LC2 的常量
    .long    1              #n_functions=1
    .long    .LC3           #functions 指针，指向.LC3
    .long    1              #ctr_mask=1
    .long    5              #以下 3 个字段构成 gcov_ctr_info 结构，该字段 num=5，即
counter 的个数
    .long    .LPBX1         #values 指针，指向.LPBX1，即 5 个 counter 的内容在.LPBX1 结构
```

```
中
    .long    __gcov_merge_add #merge 指针，指向__gcov_merge_add 函数
    .zero    12            #应该是 12 个 0

    .text                                    #text 代码段
    .type    _GLOBAL__I_0_main, @function    #类型是 function
_GLOBAL__I_0_main:                           #以下是函数体
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $.LPBX0, (%esp)    #将$.LPBX0，即.LPBX0 的地址，存入%esp 所指单元
                                #实际上是为下面调用__gcov_init 准备参数,即 gcov_info
结构指针
    call     __gcov_init        #调用__gcov_init
    leave
    ret

    .size    _GLOBAL__I_0_main, .-_GLOBAL__I_0_main
    .section    .ctors,"aw",@progbits      #该函数位于 ctors 段
    .align 4
    .long    _GLOBAL__I_0_main
    .align 4
    .long    _GLOBAL__I_0_main

   .ident    "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
   .section    .note.GNU-stack,"",@progbits
```

3.1 计数桩代码分析

共插入了 6 段桩代码，前 5 段桩代码很容易理解。实际上就是一个计数器，只要每次执行到相关代码，即会让该计数器加 1。我们以第一处桩代码为例，如下

```
    movl    .LPBX1, %eax      #将.LPBX1 移到%eax，即%eax=.LPBX1
    movl    .LPBX1+4, %edx    #edx=.LPBX1+4
    addl    $1, %eax          #eax=%eax+1
    adcl    $0, %edx          #edx=%edx+0
    movl    %eax, .LPBX1      #将%eax 移回.LPBX1
    movl    %edx, .LPBX1+4    #将%edx 移回.LPBX1+4
```

从该段汇编代码可以看出，这段代码要完成的功能实际上就是让这个计数器加 1，但该计数器是谁？——就是.LPBX1 和.LPBX1+4 组成的 8 个字节的长长整数。而前 5 处桩代码，实际上就是对一个有 5 个长长整数元素的静态数组的

为什么是静态数组？

```
    .local   .LPBX1
    .comm    .LPBX1,40,32
    .section .rodata       #只读 section
    .align   4
```

从.LPBX1 的 section 属性可以看出该数组应该是 rodata，即只读，其中的 40 应该就是其长度，即 40 字节。如下便是 LPBX1 数组，大小共 40 字节，以 4 字节方式对齐。

+0	+4	+8	+12	+16	+20	+24	+28	+32	+36
10	0	0	0	1	0	0	0	1	0

代码运行后，该数组的值就记录了桩代码被执行的次数，也即其后的代码块被执行的次数，如上所示。

3.2 构造函数桩代码分析

插入的第 6 段桩代码，先不管他的功能，先分析一下以下代码。

```
    .text                                       #text 代码段
    .type    _GLOBAL__I_0_main, @function       #类型是 function
_GLOBAL__I_0_main:                              #以下是函数体
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    movl     $.LPBX0, (%esp)    #将$.LPBX0，即.LPBX0 的地址，存入%esp 所指单元
                                #实际上是为下面调用__gcov_init 准备参数,即 gcov_info
结构指针
    call     __gcov_init        #调用__gcov_init
    leave
    ret
```

可以看出，这是一个函数，函数名为_GLOBAL__I_0_main，该函数的主要目的是调用__gcov_init 函数，调用参数就是.LPBX0 结构。

将可执行文件 test 通过 objdump 命令 dump 出来，查看该符号，也一目了然。

```
0804891b <_GLOBAL__I_0_main>:
 804891b:       55                          push   %ebp
 804891c:       89 e5                       mov    %esp,%ebp
 804891e:       83 ec 08                    sub    $0x8,%esp
                                            //将$.LPBX0，即.LPBX0 的地址，存入%esp 所指
单元
                                            //实际上是为下面调用__gcov_init 准备参数,即
gcov_info 结构指针
                                            //此处 gcov_info 的地址即为 0x804b7a0，当然
这是一个虚拟地址
 8048921:       c7 04 24 a0 b7 04 08        movl   $0x804b7a0,(%esp)
 8048928:       e8 93 01 00 00              call   8048ac0 <__gcov_init>    //调用
__gcov_init
 804892d:       c9                          leave
 804892e:       c3                          ret
 804892f:       90                          nop
```

接下来，看看__gcov_init 函数，定义如下。

```
void __gcov_init (struct gcov_info *info)
{
    if (! info- >version)
        return;

    if (gcov_version (info, info->version, 0))
    {
        const char *ptr = info- >filename;
        gcov_unsigned_t crc32 = gcov_crc32;
        size_t filename_length = strlen(info- >filename);

        /* Refresh the longest file name information */
        if (filename_length > gcov_max_filename)
            gcov_max_filename = filename_length;

        do
        {
            unsigned ix;
            gcov_unsigned_t value = *ptr << 24;

            for (ix = 8; ix-- ; value <<= 1)
            {
                gcov_unsigned_t feedback;
                feedback = (value ^ crc32) & 0x80000000 ? 0x04c11db7 : 0;
                crc32 <<= 1;
                crc32 ^= feedback;
            }
```

```
        }while (*ptr++);

        gcov_crc32 = crc32;

        if (! gcov_list)
            atexit (gcov_exit);

        info- >next = gcov_list; /* Insert info into list gcov_list */
        gcov_list = info;        /* gcov_list is the list head */
    }
    info->version = 0;
}
```

由此，我们得到两个结论：
(1) .LPBX0 结构就是 gcov_info 结构，二者相同。
(2) __gcov_init 的功能：将.LPBX0 结构，即 gcov_info 结构，串成一个链表，该链表指针就是 gcov_list。

我们先看看这些数据结构。

3.3 数据结构分析

.LPBX0 结构即为 gcov_info 结构，定义如下。

```
/* Type of function used to merge counters. */
typedef void (*gcov_merge_fn) (gcov_type *, gcov_unsigned_t);

/* Information about counters. */
struct gcov_ctr_info
{
    gcov_unsigned_t num;    /* number of counters. */
    gcov_type *values;      /* their values. */
    gcov_merge_fn merge;    /* The function used to merge them. */
};

/* Information about a single object file. */
struct gcov_info
{
    gcov_unsigned_t version; /* expected version number */
    struct gcov_info *next;  /* link to next, used by libgcov */

    gcov_unsigned_t stamp;   /* uniquifying time stamp */
    const char *filename;    /* output file name */

    unsigned n_functions;                /* number of functions */
    const struct gcov_fn_info *functions; /* table of functions */

    unsigned ctr_mask;                   /* mask of counters instrumented. */
    struct gcov_ctr_info counts[0];      /* count data. The number of bits
                                            set in the ctr_mask field
                                            determines how big this array is. */
};
```

对应于上述代码中的解释，便一目了然。此处再重复一下对该结构的解释。

```
    .align   32
    .type    .LPBX0, @object  #.LPBX0 是一个对象
    .size    .LPBX0, 52       #.LPBX0 大小为 52 字节
.LPBX0:                       #结构的起始地址，即结构名，该结构即为 gcov_info 结构
    .long    875573616        #即 version=0x34303170，即版本为 4.1p
    .long    0                #即 next 指针，为 0，next 为空
    .long    -979544300       #即 stamp=0xc59d5714
    .long    .LC2             #filename，值为.LC2 的常量
    .long    1                #n_functions=1，1 个函数
    .long    .LC3             #functions 指针，指向.LC3
    .long    1                #ctr_mask=1
```

```
    .long    5                  #以下 3 个字段构成 gcov_ctr_info 结构，该字段 num=5，即
counter 的个数
    .long    .LPBX1             #values 指针，指向.LPBX1，即 5 个 counter 的内容
在.LPBX1 结构中
    .long    __gcov_merge_add #merge 指针，指向__gcov_merge_add 函数
    .zero    12                 #应该是 12 个 0
```

上述的.LC2 即为文件名，如下。

```
    .section .rodata            #只读 section
    .align   4
.LC2:                           #文件名常量，只读
    .string  "/home/zubo/gcc/test/test.gcda"
```

然后就是 functions 结构，1 个函数，函数结构就是.LC3 的内容。

```
.LC3:
    .long    3              #ident=3
    .long    -345659544     #即 checksum=0xeb65a768
    .long    5              #counters
```

其对应的结构为 gcov_fn_info，定义如下。

```
/ * Information about a single function. This uses the trailing array idiom. The number
of
    counters is determined from the counter_mask in gcov_info. We hold an array of
function
    info, so have to explicitly calculate the correct array stride. */
struct gcov_fn_info
{
    gcov_unsigned_t ident;    /* unique ident of function */
    gcov_unsigned_t checksum; /* function checksum */
    unsigned n_ctrs[0];       /* instrumented counters */
};
```

3.4 构造函数桩代码小结

gcov_init 函数中的 gcov_list 是一个全局指针，指向 gcov_info 结构的链表，定义如下。

```
/ * Chain of per- object gcov structures. */
static struct gcov_info *gcov_list;
```

因此，被测文件在进入 main 之前，所有文件的.LPBX0 结构就被组织成一个链表，链表头就是 gcov_list。被测程序运行完之后，在__gcov_init()中通过 atexit()注册的函数 gcov_exit()就被调用。该函数将从 gcov_list 的第一个.LPBX0 结构开始，为每个被测文件生成一个.gcda 文件。.gcda 文件的主要内容就是.LPBX0 结构的内容。

至此，我们可以做这样的总结：将.LPBX0 结构串成链表的目的是在被测程序运行结束时统一写入计数信息到.gcda 文件。

因此，为了将 LPBX0 结构链成一条链，GCC 要为每个被测试源文件中插入一个构造函数 _GLOBAL__I_0_main 的桩代码，该函数名根据当前被测文件中的第一个全局函数的名字生成，其中 main 即为 test.c 中的第一个全局函数名，防止重名。

而之所以称为构造函数，是因为该函数类似 C++的构造函数，在调用 main 函数之前就会被调用。

4. 说明

本文参考文献中实际分析的 gcc 代码应该是 gcc-2.95 版本，而本文分析的 gcc 代码是 gcc-4.1.2 版本。可以发现这两个版本间变化非常非常大。

gcc-2.95 版本中有__bb_init_func()函数和__bb_exit_func()函数，并且其中的结构为 bb 结构。但在 gcc-4.1.2 版本中，就变为__gcov_init()函数和 gcov_exit()函数，对应的结构为 gcov_info 结构。

5. 小结

本文详细叙述了 Linux 平台代码覆盖率测试插桩前后汇编代码的变化及分析，对于分析 gcc 插桩、gcov 原理有很大的帮助。

Reference

费训,罗蕾. 利用 GNU 工具实现汇编程序覆盖测试, 计算机应用, 24 卷, 2004.
吴康. 面向多语言混合编程的嵌入式测试软件设计与实现(硕士论文). 电子科技大学, 2007.
http://gcc.parentingamerica.com/releases/gcc-2.95.3
http://gcc.parentingamerica.com/releases/gcc-4.1.2

Linux 平台代码覆盖率测试-编译过程自动化及对链接的解释

2011 年 5 月 13 日
10:52

0. 序

"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文中的编译过程能否自动化？能否使用 makefile 的方式进行？其链接过程使用 ld 或者 collect2 如何？

——这将是本文讨论的重点。本文 gcc 版本为 gcc-4.1.2。

1. 生成各个文件的步骤

1.1 未加入覆盖率测试选项

1.1.1 编译步骤

(1) 预处理：生成 test.i 文件

```
# cpp test.c -o test.i     //或者
# cpp test.c > test.i      //或者
# gcc -E test.c -o test.i
```

(2) 编译：生成 test.s 文件

```
# gcc -S test.i
```

(3) 汇编：生成 test.o 文件

```
# as -o test.o test.s      //或者
# gcc -c test.s -o test.o
```

(4) 链接：生成可执行文件 test

```
# gcc -o test test.o
```

或者，

```
# /usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2 --eh-frame-hdr --build-id -
m elf_i386 --hash-style=gnu -dynamic-linker /lib/ld-linux.so.2
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o /usr/lib/gcc/i386-redhat-
linux/4.1.2/../../../crti.o /usr/lib/gcc/i386-redhat-linux/4.1.2/crtbegin.o -
L/usr/lib/gcc/i386-redhat-linux/4.1.2 -L/usr/lib/gcc/i386-redhat-linux/4.1.2 -
L/usr/lib/gcc/i386-redhat-linux/4.1.2/../../.. -lgcc --as-needed -lgcc_s --no-
as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i386-
redhat-linux/4.1.2/crtend.o /usr/lib/gcc/i386-redhat-
linux/4.1.2/../../../crtn.o test.o -o test
```

因/usr/lib/gcc/i386-redhat-linux/4.1.2/../../..就是/usr/lib，因此，也可以简写为。

```
/usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2 --eh-frame-hdr --build-id -m
elf_i386 --hash-style=gnu -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/gcc/i386-redhat-linux/4.1.2/crtbegin.o -
L/usr/lib/gcc/i386-redhat-linux/4.1.2 -L/usr/lib/gcc/i386-redhat-linux/4.1.2 -
L/usr/lib -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -
lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.2/crtend.o
/usr/lib/crtn.o test.o -o test
```

1.1.2 目标文件的符号表

可通过如下命令查看 test.o 中的符号表。//后是笔者加入的注释。

```
# objdump -t test.o
```

```
test.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 test.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     F .text  0000005f main
00000000         *UND*  00000000 puts    //就是 undefined，故需要连接其他的目标文件

# nm test.o
00000000 T main    //T 即为 text symbol，大写表示 global
         U puts    //就是 undefined，故需要连接其他的目标文件
```

1.2 加入覆盖率测试选项

1.2.1 编译步骤

步骤同上，只是在编译生成 test.s 文件时加上"-fprofile-arcs -ftest-coverage"覆盖率测试选项即可。

另外，使用 collect2 的链接步骤稍有不同，需要链接 gcov 静态库(libgcov.a)。如下。

```
# /usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2 --eh-frame-hdr --build-id -
m elf_i386 --hash-style=gnu -dynamic-linker /lib/ld-linux.so.2
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o /usr/lib/gcc/i386-redhat-
linux/4.1.2/../../../crti.o /usr/lib/gcc/i386-redhat-linux/4.1.2/crtbegin.o
test.o -L/usr/lib/gcc/i386-redhat-linux/4.1.2 -L/usr/lib/gcc/i386-redhat-
linux/4.1.2 -L/usr/lib/gcc/i386-redhat-linux/4.1.2/../../.. -lgcov -lgcc --as-
needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/i386-redhat-linux/4.1.2/crtend.o /usr/lib/gcc/i386-redhat-
linux/4.1.2/../../../crtn.o -o test
```

或者简写为。

```
/usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2 --eh-frame-hdr --build-id -m
elf_i386 --hash-style=gnu -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/gcc/i386-redhat-linux/4.1.2/crtbegin.o test.o -
L/usr/lib/gcc/i386-redhat-linux/4.1.2 -L/usr/lib/gcc/i386-redhat-linux/4.1.2 -
L/usr/lib -lgcov -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
-lgcc_s --no-as-needed /usr/lib/gcc/i386-redhat-linux/4.1.2/crtend.o
/usr/lib/crtn.o -o test
```

当然，命令亦会完成链接并生成可执行文件。

```
# gcc -o test test.o -lgcov
```

libgcov.a 位于/usr/lib/gcc/i386-redhat-linux/4.1.2 目录。

1.2.2 目标文件的符号表

看看加入覆盖率测试选项后的目标文件 test.o 中的符号表。

```
# objdump -t test.o

test.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 test.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
```

```
00000000 l    d  .rodata        00000000 .rodata
000000eb l     F .text  00000014 _GLOBAL__I_0_main
00000000 l    d  .ctors 00000000 .ctors
00000000 l    d  .note.GNU-stack        00000000 .note.GNU-stack
00000000 l    d  .comment       00000000 .comment
00000000 g     F .text  000000eb main
00000000         *UND*  00000000 puts              //就是 undefined，故需要连接其他的目标文件，下同
00000000         *UND*  00000000 __gcov_merge_add
00000000         *UND*  00000000 __gcov_init

# nm test.o
000000eb t _GLOBAL__I_0_main    //t 即为 text symbol，小写表示 local
         U __gcov_init          //就是 undefined，故需要连接其他的目标文件，下同
         U __gcov_merge_add
00000000 T main
         U puts
```

1.3 gcc verbose 选项

这是如何知道的呢？——gcc 命令-v 选项即可。

```
# gcc -v test.c [-o test]
# gcc -fprofile-arcs -ftest-coverage -v test.c [-o test]
```

[]表示出现 0 次或 1 次。

-v，即 verbose 选项，以下是<An Introduction to GCC>中的解释，原文更贴切，此处不翻译。

> The '-v' option can also be used to *display detailed information about the exact sequence of commands used to compile and link a program*.

2. 编译自动化

2.1 使用 collect2 的 makefile

编写 makefile 文件，使上述过程自动化。针对该例子，笔者编写的 makefile 如下。#后面是笔者加入的注释。

```
PP = cpp      #预处理程序
CC = gcc      #编译程序
AS = as       #汇编程序

CXX_FLAGS += -g -Wall -Wextra
ASM_FLAGS = -S
COV_FLAGS = -fprofile-arcs -ftest-coverage     #如果不加入覆盖率测试选项，只需将其值置为空即可
LINK_GCOV = -lgcov                             #此处链接 gcov 静态库，如不需要，将其值置为空即可

TARGET_FILE = test     #test 即是 TARGET_FILE 全局变量的值，如要用到别的程序，只需修改此值即可使用

SRC_FILE = $(TARGET_FILE).c
CPP_FILE = $(TARGET_FILE).i
ASM_FILE = $(TARGET_FILE).s
OBJ_FILE = $(TARGET_FILE).o
EXE_FILE = $(TARGET_FILE)

TARGET = $(CPP_FILE) $(ASM_FILE) $(OBJ_FILE) $(EXE_FILE)

CLEANUP = rm -f $(TARGET)

all : $(TARGET)

clean :
```

```
    $(CLEANUP)

#如果是其他的平台或 GCC，此 3 路径或许需要修改才能使用
COLLECT2_DIR=/usr/libexec/gcc/i386-redhat-linux/4.1.2
CRTLIB_DIR=/usr/lib/gcc/i386-redhat-linux/4.1.2
LIB_DIR=/usr/lib

COLLECT2=$(COLLECT2_DIR)/collect2
LDSO=/lib/ld-linux.so.2

CRT1=$(LIB_DIR)/crt1.o
CRTI=$(LIB_DIR)/crti.o
CRTBEGIN=$(CRTLIB_DIR)/crtbegin.o
CRTEND=$(CRTLIB_DIR)/crtend.o
CRTN=$(LIB_DIR)/crtn.o

LINK_FLAGS = --eh-frame-hdr --build-id -m elf_i386 --hash-style=gnu -dynamic-
linker
LINK_LIBS  = -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -
lgcc_s --no-as-needed
SEARCH_DIR = -L$(CRTLIB_DIR) -L$(CRTLIB_DIR) -L$(LIB_DIR)

$(TARGET):
$(CPP_FILE): $(SRC_FILE)
    $(PP) $^ -o $@

$(ASM_FILE): $(CPP_FILE)
    $(CC) $(ASM_FLAGS) $(COV_FLAGS) $^ -o $@

$(OBJ_FILE): $(ASM_FILE)
    $(AS) $^ -o $@

$(EXE_FILE): $(OBJ_FILE)
    $(COLLECT2) $(LINK_FLAGS) $(LDSO) $(OBJ_FILE) $(CRT1) $(CRTI) $(CRTBEGIN)
    $(SEARCH_DIR) $(LINK_GCOV) $(LINK_LIBS) $(CRTEND) $(CRTN) -o $@
```

这是一个通用的 makefile，如果要用到别的程序，只需修改 TARGET_FILE 的值即可。

2.2 不使用 collect2 的 makefile

如果不使用 collect2 进行链接，直接使用 gcc 命令，makefile 就很简单了，如下。

```
PP = cpp
CC = gcc
AS = as

CXX_FLAGS += -g -Wall -Wextra
ASM_FLAGS = -S
COV_FLAGS = -fprofile-arcs -ftest-coverage     #如果不加入覆盖率测试选项，只需将其值置
为空即可
LINK_GCOV = -lgcov                             #此处链接 gcov 静态库，如不需要，将其值
置为空即可

TARGET_FILE = test     #test 即是 TARGET_FILE 全局变量的值，如要用到别的程序，只需修改此
值即可使用

SRC_FILE = $(TARGET_FILE).c
CPP_FILE = $(TARGET_FILE).i
ASM_FILE = $(TARGET_FILE).s
OBJ_FILE = $(TARGET_FILE).o
EXE_FILE = $(TARGET_FILE)

TARGET = $(CPP_FILE) $(ASM_FILE) $(OBJ_FILE) $(EXE_FILE)

CLEANUP = rm -f $(TARGET)
```

```
all : $(TARGET)

clean :
     $(CLEANUP)

$(TARGET):
$(CPP_FILE): $(SRC_FILE)
     $(PP) $^ -o $@

$(ASM_FILE): $(CPP_FILE)
     $(CC) $(ASM_FLAGS) $(COV_FLAGS) $^ -o $@

$(OBJ_FILE): $(ASM_FILE)
     $(AS) $^ -o $@

$(EXE_FILE): $(OBJ_FILE)
     $(CC) $^ $(LINK_GCOV) -o $@     //链接 gcov 静态库必须在 test.o 之后
```

3. 关于链接的讨论

3.1 链接顺序

test.o 必须在 gcov 静态库(libgcov.a)之前被链接。因为 test.o 中引用的符号__gcov_init 和 __gcov_merge_add 在 gcov 静态库中。否则，会出现如下错误。

```
# gcc -lgcov test.o -o test
test.o: In function `global constructors keyed to 0_main':
test.c:(.text+0xf9): undefined reference to `__gcov_init'
test.o:(.data+0x44): undefined reference to `__gcov_merge_add'
collect2: ld returned 1 exit status
```

为什么会出现错误？
——因为，gcc 链接规定，链接时，若 A 和 B 同时需要链接，不论 A/B 是目标文件还是库文件，若 A 中引用了 B 的符号，例如函数或者全局变量，则在链接时，必须将 A 写在 B 前面；因为，链接时从左向右搜索外部符号。

以下是<An Introduction to GCC>中的解释，原文更贴切，此处不翻译。

链接目标文件。

> On Unix-like systems, the traditional behavior of compilers and linkers is to *search for external functions from left to right in the object files specified on the command line*. This means that the *object file which contains the definition of a function should appear after any files which call that function*.

链接库文件。

> they are searched *from left to right* — *a library containing the definition of a function should appear after any source files or object files which use it*.

当然，现在有的编译器在链接时会搜索所有的目标文件或库文件，不考虑顺序。但并不是所有的编译器都这样做，因此，最好还是遵循从左向右的习惯写目标文件或库文件。

关于链接顺序，请参考"GCC 的链接顺序"。

3.2 错误链接顺序的例子

当然，也可以使用 collect2 进行链接。调整其中链接的目标文件和库文件的顺序，若顺序不正确，也会出现类似的错误。有兴趣的读者可自行实验。此处只给出两个错误的例子。

例 1。

```
$(COLLECT2) $(LINK_FLAGS) $(LDSO) $(CRT1) $(CRTI) $(CRTBEGIN) $(SEARCH_DIR)
$(LINK_LIBS) $(OBJ_FILE)  $(LINK_GCOV) $(CRTEND) $(CRTN) -o $@
/usr/lib/gcc/i386-redhat-linux/4.1.2/libgcov.a(_gcov.o): In function
`gcov_version':
(.text+0xa6): undefined reference to `__stack_chk_fail_local'
/usr/lib/gcc/i386-redhat-linux/4.1.2/libgcov.a(_gcov.o): In function
`__gcov_init':
(.text+0x17b): undefined reference to `atexit'
/usr/lib/gcc/i386-redhat-linux/4.1.2/libgcov.a(_gcov.o): In function
`gcov_exit':
(.text+0x1471): undefined reference to `__stack_chk_fail_local'
collect2: ld returned 1 exit status
make: *** [test] Error 1
```

例 2。

```
$(COLLECT2) $(LINK_FLAGS) $(OBJ_FILE) $(LDSO) $(CRT1) $(CRTI) $(CRTBEGIN)
$(SEARCH_DIR) $(LINK_GCOV) $(LINK_LIBS) $(CRTEND) $(CRTN) -o $@
/usr/lib/crt1.o: In function `_start':
(.text+0x18): undefined reference to `main'
collect2: ld returned 1 exit status
make: *** [test] Error 1
```

为什么出现这样的错误，根据 3.1 的讨论，原因即知。当然，这里涉及到程序启动_init 和结束_fini 这两个函数，关于程序的启动和结束分析，将另文讨论。

4．额外的话

从链接命令来看，要链接生成一个可执行程序，需要 crt1.o, crti.o, crtn.o, crtbegin.o, crtend.o 文件一起链接。

```
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o，即/usr/lib/crt1.o
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crti.o，即/usr/lib/crti.o
/usr/lib/gcc/i386-redhat-linux/4.1.2/crtbegin.o
/usr/lib/gcc/i386-redhat-linux/4.1.2/crtend.o
/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crtn.o，即/usr/lib/crtn.o
```

如果是其他版本的 gcc，该目录可能会有变化，例如在笔者的虚拟机上，gcc 版本为 gcc-4.4.1，collect2/crtbegin.o/crtend.o 的目录变为/usr/lib/gcc/i486-linux-gnu/4.4.1，如下。

```
/usr/lib/gcc/i486-linux-gnu/4.4.1/collect2
/usr/lib/gcc/i486-linux-gnu/4.4.1/../../../../lib/crt1.o
/usr/lib/gcc/i486-linux-gnu/4.4.1/crtbegin.o
```

这些目标文件的作用就是为 main 函数的运行建立运行环境，并提供在运行结束后释放资源。关于其详细解释和代码分析，将另文讨论。

5．小结

本文针对"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文中的编译过程，重点讨论该编译过程的自动化，并讨论了链接顺序及其原因。编译自动化，关键还是 makefile 文件的编写。

Reference

<An Introduction to GCC for the GNU Compilers gcc and g++>, by Brian Gough, Richard M. Stallman.
<Using the GNU Compiler Collection>, by Richard M. Stallman and the GCC Developer Community.

Linux 平台代码覆盖率测试-GCC 插桩基本概念及原理分析

2011 年 5 月 11 日
11:13

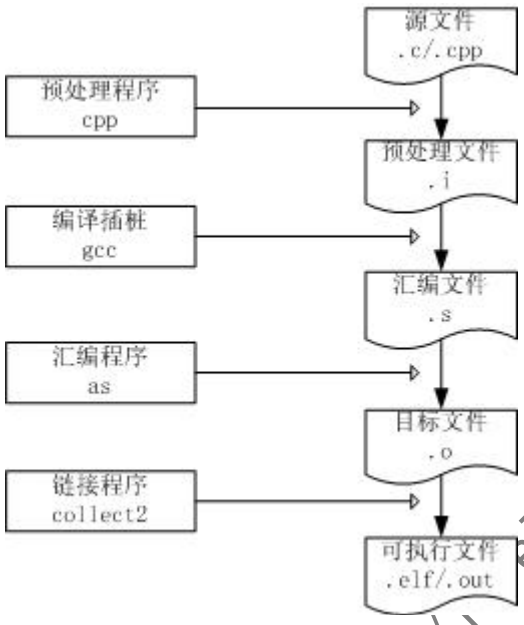
1. 序

在"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文，我们已经分析了 GCC 插桩前后汇编代码的变化，对 GCC 插入的桩代码有比较全面的了解。本文将简单叙述 GCC 插桩的基本概念和原理。

2. GCC 插桩原理

2.1 GCC 编译插桩的过程

GCC 的插桩和编译过程如下图。



过程描述：
(1) 编译预处理程序对源文件进行预处理，生成预处理文件(.i 文件)
(2) 编译插桩程序对.i 文件进行编译，生成汇编文件(.s 文件)，同时完成插桩
(3) 汇编程序对.s 文件进行汇编，生成目标文件(.o 文件)
(4) 链接程序对.o 文件进行连接，生成可执行文件(.out/.elf 文件)

因此，插桩是汇编一级的插桩，每个桩点插入 6 条汇编语句，直接插入.s 文件中。在程序执行的过程中，这些桩代码负责收集程序的执行信息。桩代码及其分析请参考"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文。

2.2 GCC 在何处插桩

GCC 对部分 arc 插桩，为什么？

- 从 arc 的执行次数可以很容易的计算出每个 BB 的执行次数，而从 BB 的执行次数推算出每个 arc 的执行次数的困难的；
- 也没有必要对每个 arc 都插桩，只需要知道图中部分关键 arc 的执行次数就可以计算出所有 BB 和 arc 的执行次数。桩的数量只要保证能够计算出所有 BB 和 arc 的执行次数的最低限度即可。

2.3 GCC 如何才能在编译的同时插桩

BB 和 arc 都是汇编一级的概念，那么如何才能让 GCC 在编译的同时插入桩代码呢？

在编译时加入"-ftest-coverage -fprofile-arcs"选项即可。

-ftest-coverage 选项会让 GCC 为每个源文件生成同名的.gcno 文件，在 gcov 程序中，将读取.gcno 文件，重组每一个可执行程序的程序流图。

-fprofile-arcs 选项会让 GCC 为每个源文件生成同名的.gcda 文件，该文件包含每一个指令分支的执行次数信息。请参考"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"及前面的文章。

3. 小结

本文简单地叙述 GCC 插桩的基本概念和原理，包括编译插桩过程，插桩点及编译(插桩)选项。

Reference

费训,罗蕾. 利用 GNU 工具实现汇编程序覆盖测试, 计算机应用, 24 卷, 2004.
吴康. 面向多语言混合编程的嵌入式测试软件设计与实现(硕士论文). 电子科技大学, 2007.
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448906.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/26/6448799.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/01/6382489.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/04/13/6321909.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448885.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448867.aspx

Linux 平台代码覆盖率测试-基本块图、插桩位置及桩代码执行分析

2011 年 4 月 14 日
16:25

0. 序

由前面几篇文章，例如，
Linux 平台代码覆盖率测试-gcov-dump 原理分析
Linux 平台代码覆盖率测试-.gcda/.gcno 文件及其格式分析
Linux 平台代码覆盖率测试-GCC 插桩基本概念及原理分析
Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析

我们已经知道了代码覆盖率测试的一些相关概念，例如 Basic Block，Arcs 等。本文重点叙述这些基本概念，并仍以前面的 test.c 为例说明基本块图(Basic Block Graph)及插桩点。

本文所用 gcc 版本为 gcc-4.1.2。test.c 代码如下。

```
00001: #include <stdio.h>
00002:
00003: int main (void)
00004: {
00005:    int i, total;
00006:
00007:    total = 0;
00008:
00009:    for (i = 0; i < 10; i++)
00010:    total += i;
00011:
00012:    if (total != 45)
00013:        printf ("Failure\n");
00014:    else
00015:        printf ("Success\n");
00016:    return 0;
00017: }
00018:
```

1. 基本块概念

通过基本块的划分，可以把程序分为一组代码段，每一段代码中的第一条语句被执行，则整段代码都被执行一次。

那么，什么是基本块(Basic Block)？

基本块
- 程序中一个顺序执行的语句序列
- 只有一个出口语句和一个入口语句
- 执行时只能从入口语句入，从出口语句退出
- 基本块中的所有语句的执行次数一定是相同的
- 一般由多个顺序执行语句后跟一个跳转语句组成

因此，基本块的最后一条语句一定是一个跳转语句，且跳转的目的地一定是另一个基本块的第一条语句。如果跳转语句是有条件的，就产生了一个分支(arc)，该基本块就有两个基本块作为目的地。如果把每个基本块当作一个节点，那么一个函数中的所有基本块就构成了一个有向图，称之为基本块图(Basic Block Graph)。且只要知道图中部分 BB 或 arc 的执行次数就可以推算出所有的 BB 和所有的 arc 的执行次数。

本文附录介绍了 GCC 源代码中对 Basic Block 的解释。下面以 test.c 为例，说明这些问题。

2. 基本块图及插桩点分析

2.1 基本块图

—

根据"Linux 平台代码覆盖率测试-.gcda/.gcno 文件及其格式分析"一文 1.1 节和 2.1 节的输出信息，我们可以画出如下的基本块图(Basic Block Graph)。

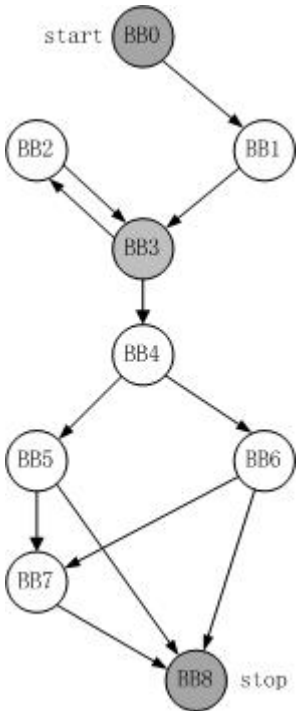


图 1 基本块图

其中，BB0 和 BB8 分别对应着开始和结束的虚拟基本块，表明程序从 BB0 开始，从 BB8 退出。因此，BB0 没有入边，只有出边，而 BB8 只有入边，没有出边。8 个 ARCS 记录对应图中的 8 个有出度的节点，12 条有向边实际上就是所有的 arcs 总数。

2.2 有效基本块图

从输出结果可以看出，BB0,BB8,BB3 没有对应的代码行，可以当作虚拟的基本块。有效的基本块只有 BB1, BB2, BB4, BB5, BB6, BB7。

如果在图 1 中，我们删除虚拟基本块 BB0 和 BB8，并加入"Linux 平台代码覆盖率测试-.gcda/.gcno 文件及其格式分析"一文 2.1 节的输出的 LINES 信息(BB 的行号信息)，那么程序的逻辑会更清楚。如图 2 所示。

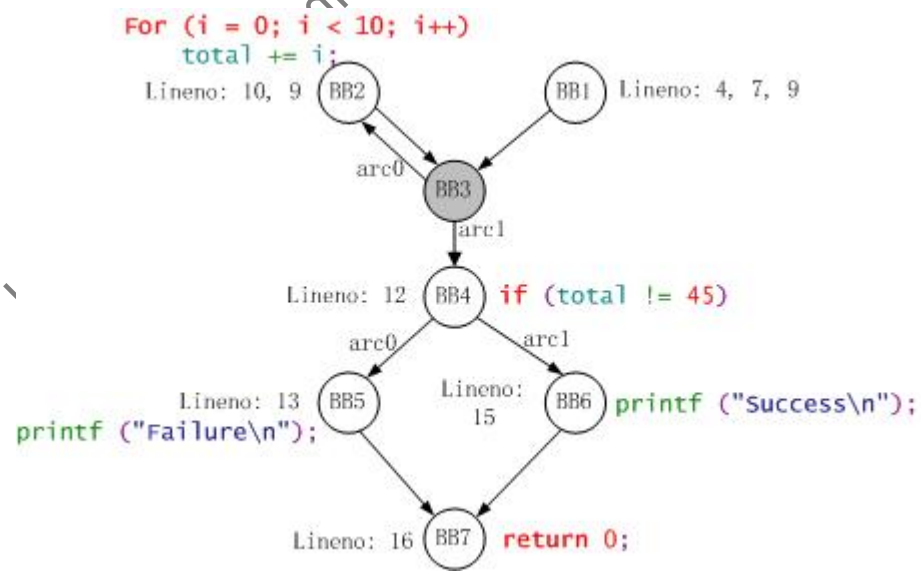


图 2 有效基本块(带行号信息)图

其中，每个节点旁边的 lineno(行号)表示构成该 BB 的代码行，这些代码也显示在旁边。

这里需要说明的问题：**为什么 BB1 和 BB2 都有 lineno=9 的代码？**

lineno=9 的代码即为 for 语句本身，但实际上，for 循环的执行顺序是：
(0) 初始化(某些场合初始化可以在 for 外面)
(1) 判断条件，如果条件成立，则执行 for 循环体，再对循环变量进行增/减操作；否则，直接进入 for 循环后的块；

因此，这里 BB2 中的 lineno=9 实际上只是 for 里面的循环变量自增语句(i++)，这也是为什么 BB2 的代码行信息 lineno=10 写在 lineno=9 前面的原因；而 BB1 中的 lineno=9 实际上只用到判断条件(i < 10)。

2.3 含桩点信息的有效基本块图

结合"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文对插入的桩代码的分析，可以很容易地在图 2 的基础上画出带有桩点信息的有效基本块图，如图 3 所示。

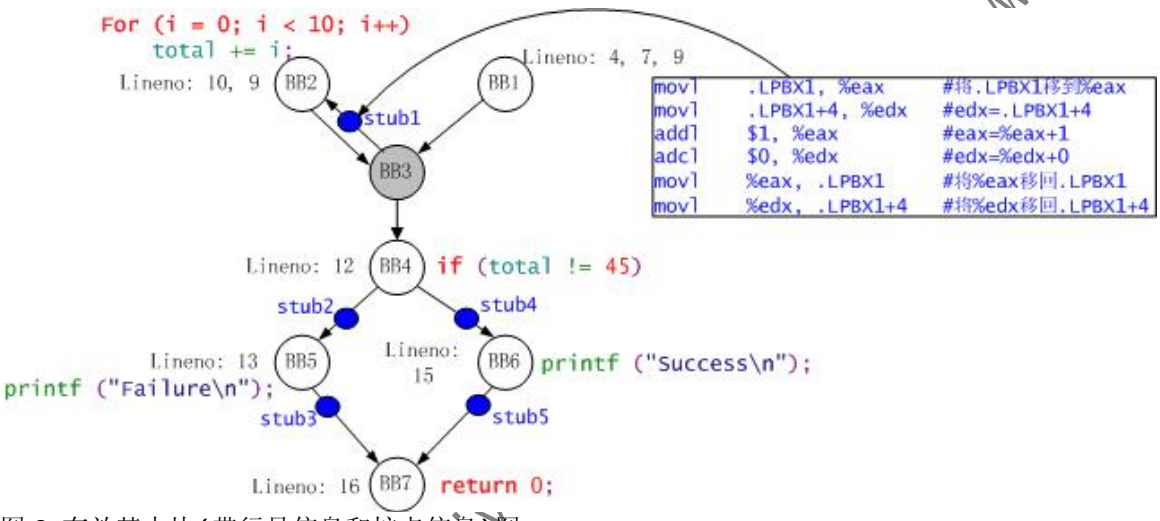


图 3 有效基本块(带行号信息和桩点信息)图

若将"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文中前 5 段桩代码按其在文件中的先后顺序依次记为 stub1, stub2, ..., stub5，则图中的 stub1～stub5 分别与该文的 stub1～stub5 一一对应。此处为读者方便理解，列出图中的第 1 个桩点 stub1 的代码，如下。关于桩代码的分析已在该文中详细叙述，不再赘述。

```
    movl    .LPBX1, %eax      #将.LPBX1 移到%eax，即%eax=.LPBX1
    movl    .LPBX1+4, %edx    #edx=.LPBX1+4
    addl    $1, %eax          #eax=%eax+1
    adcl    $0, %edx          #edx=%edx+0
    movl    %eax, .LPBX1      #将%eax 移回.LPBX1
    movl    %edx, .LPBX1+4    #将%edx 移回.LPBX1+4
```

关于该段桩代码的解释，可参考"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文的 3.1 节。

2.4 插桩位置及桩代码执行情况分析

根据 2.3 节的叙述及"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文的解释，总结一下该例子的插桩位置。

stub1: 被插在第 10 行代码之前，被执行 10 次，存于.LPBX1+0 位置。
stub2：被插在第 13 行代码之前，被执行 0 次，存于.LPBX1+8 位置。

stub3：被插在第 13 行代码之后，被执行 0 次，存于.LPBX1+24 位置。
stub4：被插在第 15 行代码之前，被执行 1 次，存于.LPBX1+16 位置。
stub5：被插在第 15 行代码之后，被执行 1 次，存于.LPBX1+32 位置。

上述执行次数就是"Linux 平台代码覆盖率测试-GCC 插桩前后汇编代码对比分析"一文 3.1 节中的静态数组.LPBX1 元素的值(注意元素的存放位置)。

+0	+4	+8	+12	+16	+20	+24	+28	+32	+36
10	0	0	0	1	0	0	0	1	0

而这些值就被作为 COUNTERS 写入了 test.gcda 文件，tag=0x01a10000，每个计数值(counter)8 个字节。请参考"Linux 平台代码覆盖率测试-gcov-dump 原理分析"和"Linux 平台代码覆盖率测试-.gcda/.gcno 文件及其格式分析"。

3. 小结

本文简单介绍了基本块(Basic Block)和分支的基本概念，并通过例子重点叙述了基本块图(Basic Block Graph)、有效基本块图、含桩点信息的有效基本块图，同时分析了插桩位置和桩代码执行情况。

Reference

./gcc/basic_block.h(本文对 Basic Block 的英文解释即在该文件中)
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448867.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448885.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6448906.aspx
http://blog.csdn.net/livelylittlefish/archive/2011/05/27/6449256.aspx

Appendix: 源代码中对 Basic Block 的解释

A basic block is a sequence of **instructions** with **only entry and only one exit**. If any one of the instructions are executed, they will all be executed, and in sequence from first to last.

There may be **COND_EXEC instructions** in the basic block. The COND_EXEC *instructions* will be executed -- but if the condition is false the conditionally executed *expressions* will of course not be executed. We don't consider the conditionally executed expression (which might have side-effects) to be in a separate basic block because **the program counter** will ***always be*** at the same location after the COND_EXEC instruction, regardless of whether the condition is true or not.

Basic blocks need not start with a label nor end with a **jump insn**. For example, a previous basic block may just "conditionally fall" into the succeeding basic block, and the last basic block need not end with a jump insn. Block 0 is a descendant of the entry block.

A basic block beginning with two labels cannot have notes between the labels.

Data for jump tables are stored in **jump_insns** that occur in no basic block even though these insns can follow or precede insns in basic blocks.

Basic block information indexed by **block number**.

未完，待续。。。

敬请关注。