< 2017年10月 >

日	一	二	三	四	五	六
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

公告

友情链接

HotApp小程序统计

昵称：Likwo
园龄：8年3个月
粉丝：417
关注：30
+加关注

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

iphone(155)
android(80)
linux(39)
java(31)
php(25)
webkit(24)
yii(24)

C++ Profiler工具之初体验

转 http://www.cnblogs.com/lenolix/archive/2010/12/13/1904868.html

概要：本文同期调研了google profile工具以及其他常用profile的工具，如GNU gprof、oprofile等(都是开源项目)，并对其实现原理做了简单分析和比较。希望对之后的推广使用或二期开发有所帮助。

一、 GUN Gropf

Gprof是GNU profiler工具。可以显示程序运行的“flatprofile”，包括每个函数的调用次数，每个函数消耗的处理器时间。也可以显示“调用图”，包括 函数的调用关系，每个函数调用花费了多少时间。还可以显示“注释的源代码”，是程序源代码的一个复本，标记有程序中每行代码的执行次数。关于Gprof的 使用以及实现原理网上已有多篇文章提及，本文就不再详述，只是对其进行梳理和总结，方便阅读。（Gprof的官方网 址：http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html，http: //sourceware.org/binutils/docs/gprof/index.html 绝对权威的参考资料。）

1.1 安装

Glibc自带，无需另外安装

1.2 用法

参考http://hi.baidu.com/juventus/blog/item/312dd42a0faf169b033bf6ff.html/cmtid/3c34349bb5a8ceb8c9eaf4c5

图形化输出请参考大师blog：http://www.51testing.com/?uid-13997-action-viewspace-itemid-79952

1.3 实现原理

微信小程序(21)
c++(16)
stl(11)
更多

随笔档案

2017年10月 (2)
2017年8月 (6)
2017年3月 (3)
2017年2月 (1)
2017年1月 (2)
2016年12月 (14)
2016年11月 (15)
2016年10月 (8)
2016年9月 (8)
2016年8月 (9)
2016年7月 (4)
2016年6月 (3)
2016年4月 (2)
2016年3月 (3)
2016年2月 (5)
2016年1月 (3)
2015年11月 (1)
2015年10月 (3)
2015年9月 (4)
2015年8月 (2)
2015年7月 (5)
2015年6月 (4)
2015年5月 (4)
2015年3月 (4)
2015年1月 (2)
2014年12月 (4)
2014年11月 (1)
2014年10月 (3)
2014年9月 (2)
2014年8月 (7)
2014年7月 (9)
2014年6月 (1)
2014年5月 (6)
2014年4月 (3)
2014年3月 (5)
2014年2月 (2)
2014年1月 (12)
2013年12月 (11)
2013年11月 (3)

引用官网说明：

> Profiling works by changing how every function in your program iscompiled so that when it is called, it will stash away some informationabout where it was called from. From this, the profiler can figure outwhat function called it, and can count how many times it was called.This change is made by the compiler when your program is compiled withthe `-pg' option.
>
> Profiling also involves watching your program as it runs, andkeeping a histogram of where the program counter happens to be everynow and then. Typically the program counter is looked at around 100times per second of run time, but the exact frequency may vary fromsystem to system.
>
> A special startup routine allocates memory for the histogram andsets up a clock signal handler to make entries in it. Use of thisspecial startup routine is one of the effects of using `gcc ... -pg' tolink. The startup file also includes an `exit' function which isresponsible for writing the file `gmon.out'.
>
> Number-of-calls information for library routines is collected byusing a special version of the C library. The programs in it are thesame as in the usual C library, but they were compiled with `-pg'. Ifyou link your program with `gcc ... -pg', it automatically uses theprofiling version of the library.
>
> The output from gprof gives no indication of parts of your programthat are limited by I/O or swapping bandwidth. This is because samplesof the program counter are taken at fixed intervals of run time.Therefore, the time measurements in gprof output say nothing about timethat your program was not running. For example, a part of the programthat creates so much data that it cannot all fit in physical memory atonce may run very slowly due to thrashing, but gprof will say it useslittle time. On the other hand, sampling by run time has the advantagethat the amount of load due to other users won't directly affect theoutput you get.

当我们使用"-pg" 选项编译程序后，gcc会做三个工作：

1. 程序的入口处(main 函数之前)插入monstartup函数的调用代码，完成profile的初始化工作，包括分配保存信息的内存以及设置一个clock信号处理函数；

2. 在每个函数的入口处插入_mcount函数的调用代码，用于统计函数的调用信息：包括调用时间、调用次数以及调用栈信息；

3. 在程序退出时(在 atexit () 里)插入_mcleanup()函数的调用代码，负责将profile信息输出到gmon.out中。

这些过程可以通过objdump反汇编显示出来：

```
objdump -S a.out

0000000000400aba<main>:

400aba:      55                      push   %rbp
```

```
400abb:    48 89e5          mov    %rsp,%rbp
400abe:    48 83 ec20       sub    $0x20,%rsp
400ac2:    e8 69 fd ffff    callq  400830<mcount@plt>
.....
```

可以看出，在main函数的入口插入了一行汇编代码：callq 400830 <mcount@plt> ，这样main函数的第一行执行代码就是调用_mcount函数。

我们接下来再看看glibc的这三个函数具体都做了什么：

a) __monstartup 此函数的定义在glibc的gmon/gmon.c中

> A special startup routine allocates memory for the histogram andeither calls profil() or sets up a clock signal handler. This routine(monstartup) can be invoked in several ways. On Linux systems, aspecial profiling startup file gcrt0.o, which invokes monstartup beforemain, is used instead of the default crt0.o. Use of this specialstartup file is one of the effects of using `gcc ... -pg' to link. OnSPARC systems, no special startup files are used. Rather, the mcountroutine, when it is invoked for the first time (typically when main iscalled), calls monstartup.

linux系统中，__monstartup是在__gmon_start__ 中调用的。在程序链接过程中，gcc用gcrt0.o替代了默认的crt0.o，从而修改了main函数执行前的初始化工作：

> crt0.o是应用程序编译链接时需要的起动文件，在程序链接阶段被链接。主要工作是初试化应用程序栈，初试化程序的运行环境和在程序退出时清除和释放资源。

__gmon_start__的定义在csu/gmon-start.c中

```
void
__gmon_start__ (void)
{
#ifdef HAVE_INITFINI
  /* Protect from being called more than once.  Since crti.o is linked
     into every shared library, each of their init functions will call us.  */
```

2010年2月 (5)
2010年1月 (3)
2009年12月 (10)
2009年11月 (1)
2009年10月 (5)
2009年9月 (12)
2009年8月 (2)

link
tag

积分与排名
积分 - 635302
排名 - 165

最新评论
1. Re:Android 通过Socket 和服务器通讯
推荐一个socket编程的异步socket类库，，对一些业务场景做了支持断线重连一对一请求通知、粘性通知串行请求合并包分片处理缓存拦截器支持rxjava，提供类似于retrofit的支持...
--tong__
2. Re:微信小程序之后台https域名绑定以及免费的https证书申请
博主有心了，我已申请一个
--ycookiee
3. Re:微信小程序需要https后台的创业机会思考
应该说，还是有一些机会的
--ycookiee
4. Re:Android ListView的item背景色设置
讲解的很清晰
--LiuHDme
5. Re:nginx配置url重写
非常详细！
--weiyinfu

阅读排行榜
1. php 解析xml 的四种方法（转）(121901)

```
 static int called;


 if (called)
   return;


 called = 1;
#endif


 /* Start keeping profiling records.  */
 __monstartup ((u_long) TEXT_START, (u_long) &etext);


 /* Call _mcleanup before exiting; it will write out gmon.out from the
    collected data.  */
 atexit (&_mcleanup);
```

__gmon_start__ 不仅调用了__monstartup函数，还注册了一个清理函数_mcleanup，此函数将在程序结束时被调用。_mcleanup的功能会在后续说明，接下来让我们看看__monstartup函数都做了什么。

```
 void
__monstartup (lowpc, highpc)
    u_long lowpc;
    u_long highpc;
{
 register int o;
 char *cp;
 struct gmonparam *p = &_gmonparam;
```

2. Eclipse 代码提示功能设置。(114589)
3. Velocity 语法（转）(106025)
4. Android通过tcpdump抓包(59577)
5. Intellij IDEA常用快捷键和一些配置——Mac版(55413)

评论排行榜

1. 聊聊我的业余创业经历（只有干货）(66)
2. Android 通过Socket 和服务器通讯(28)
3. Eclipse 代码提示功能设置。(14)
4. IOS多线程读写Sqlite问题解决(12)
5. 微信小程序之后台https域名绑定以及免费的https证书申请(11)

推荐排行榜

1. 聊聊我的业余创业经历（只有干货）(46)
2. UIView你知道多少(13)
3. Velocity 语法（转）(10)
4. Eclipse 代码提示功能设置。(9)
5. IOS 多线程的一些总结(6)

```
/*
 * round lowpc and highpc to multiples of the density we're using
 * so the rest of the scaling (here and in gprof) stays in ints.
 */
p->lowpc = ROUNDDOWN(lowpc, HISTFRACTION * sizeof(HISTCOUNTER));
p->highpc = ROUNDUP(highpc, HISTFRACTION * sizeof(HISTCOUNTER));
p->textsize = p->highpc - p->lowpc;
p->kcountsize = ROUNDUP(p->textsize / HISTFRACTION, sizeof(*p->froms));
p->hashfraction = HASHFRACTION;
p->log_hashfraction = -1;
/* The following test must be kept in sync with the corresponding
   test in mcount.c.  */
if ((HASHFRACTION & (HASHFRACTION - 1)) == 0) {
    /* if HASHFRACTION is a power of two, mcount can use shifting
   instead of integer division.  Precompute shift amount. */
    p->log_hashfraction = ffs(p->hashfraction * sizeof(*p->froms)) - 1;
}
p->fromssize = p->textsize / HASHFRACTION;
p->tolimit = p->textsize * ARCDENSITY / 100;
if (p->tolimit < MINARCS)
  p->tolimit = MINARCS;
else if (p->tolimit > MAXARCS)
  p->tolimit = MAXARCS;
p->tossize = p->tolimit * sizeof(struct tostruct);
```

```
  cp = calloc (p->kcountsize + p->fromssize + p->tossize, 1);
  if (! cp)
    {
      ERR("monstartup: out of memory\n");
      p->tos = NULL;
      p->state = GMON_PROF_ERROR;
      return;
    }
  p->tos = (struct tostruct *)cp;
  cp += p->tossize;
  p->kcount = (HISTCOUNTER *)cp;
  cp += p->kcountsize;
  p->froms = (ARCINDEX *)cp;

  p->tos[0].link = 0;

  o = p->highpc - p->lowpc;
  if (p->kcountsize < (u_long) o)
    {
#ifndef hp300
      s_scale = ((float)p->kcountsize / o ) * SCALE_1_TO_1;
#else
      /* avoid floating point operations */
      int quot = o / p->kcountsize;
```

```
    if (quot >= 0x10000)
  s_scale = 1;
    else if (quot >= 0x100)
  s_scale = 0x10000 / quot;
    else if (o >= 0x800000)
  s_scale = 0x1000000 / (o / (p->kcountsize >> 8));
    else
  s_scale = 0x1000000 / ((o << 8) / p->kcountsize);
#endif
  } else
    s_scale = SCALE_1_TO_1;

 __moncontrol(1);
}
```

可以看书，函数中的大部分代码都是在做初始化工作，为profile信息分配存储空间，它的两个参数lowpc，highpc（**通过调试可以得知lowpc起始是程序代码段的起始地址，而highpc是程序代码段的结束地址，&etext**），分别代表了需要记录profile信息的地址范围，超过这个范围的地址，gprof是不会记录profile信息的。**这也解释了为何gprof不能支持对动态库的解析，以为动态库的装载是在程序代码段之外的。**我们通过一个实例可以证明这一点。

以一个简单的测试程序为例：

```
#include <stdio.h>
int or_f(int a,int b)
{
  return a^b;
}
```

```
int main(int argc,char** argv)
{
    printf("%d\n",or_f(1,2));
    sleep(30);
    return 1;
}
```

编译生成./test可执行程序。我们用readelf工具获取test文件的段信息，

readelf -S test

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
......
[12] .text             PROGBITS        0000000000400540  00000540
       0000000000000278  0000000000000000  AX       0     0     16
.......
```

从输出可以看出，test可执行程序的text代码地址为0x400540 ~ 0x400540 + 0x278。

接下来运行./test ，通过对glibc代码的修改，我们打印出__monstartup函数的两个实参值，结果如下：

lowpc: 400540, highpc: 4007c6，正好对应着test程序的代码段范围。

同时我们也dump出test程度在内存中的装载地址：

cat /proc/$self/maps:

```
00400000-00401000 r-xp 00000000 08:03 70746688                           /tmp/test
00600000-00601000 rw-p 00000000 08:03 70746688                           /tmp/test
10ca4000-10cc5000 rw-p 10ca4000 00:00 0                                  [heap]
```

```
3536600000-353661c000 r-xp 00000000 08:03 93028660                      /lib64/ld-2.5.so
353681b000-353681c000 r--p 0001b000 08:03 93028660                      /lib64/ld-2.5.so
353681c000-353681d000 rw-p 0001c000 08:03 93028660                      /lib64/ld-2.5.so
2b4f1af23000-2b4f1af25000 rw-p 2b4f1af23000 00:00 0
2b4f1af25000-2b4f1b063000 r-xp 00000000 08:03 32931849       /root/glibc-2.5-42-build/lib/libc-2.5.so
2b4f1b063000-2b4f1b263000 ---p 0013e000 08:03 32931849       /root/glibc-2.5-42-build/lib/libc-2.5.so
2b4f1b263000-2b4f1b267000 r--p 0013e000 08:03 32931849       /root/glibc-2.5-42-build/lib/libc-2.5.so
2b4f1b267000-2b4f1b268000 rw-p 00142000 08:03 32931849       /root/glibc-2.5-42-build/lib/libc-2.5.so
2b4f1b268000-2b4f1b26f000 rw-p 2b4f1b268000 00:00 0
7fffa306b000-7fffa3080000 rw-p 7ffffffea000 00:00 0                      [stack]
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0                  [vdso]
```

test装载到内存的地址范围为00400000-00401000，为libc.so装载到内存的地址范围为2b4f1af25000-2b4f1b063000，现在不在lowpc和highpc范围之内，所以libc中的函数是不会被gprof解析的。

__monstartup函数的最后会调用__moncontrol函数来设置一个clock信号处理函数用于设置提取sample。

__moncontrol的定义在glibc的gmon/gmon.c中

```
 void
__moncontrol (mode)
     int mode;
{
 struct gmonparam *p = &_gmonparam;


 /* Don't change the state if we ran into an error.  */
 if (p->state == GMON_PROF_ERROR)
   return;
```

```
  if (mode)
    {
      /* start */
      __profil((void *) p->kcount, p->kcountsize, p->lowpc, s_scale);
      p->state = GMON_PROF_ON;
    }
  else
    {
      /* stop */
      __profil(NULL, 0, 0, 0);
      p->state = GMON_PROF_OFF;
    }
}
```

其中__profil的定义在sysdeps/posix/profil.c中

```
int
__profil (u_short *sample_buffer, size_t size, size_t offset, u_int scale)
{
  struct sigaction act;
  struct itimerval timer;
#ifndef IS_IN_rtld
  static struct sigaction oact;
  static struct itimerval otimer;
```

```
# define oact_ptr &oact
# define otimer_ptr &otimer


  if (sample_buffer == NULL)
    {
      /* Disable profiling.  */
      if (samples == NULL)
    /* Wasn't turned on.  */
    return 0;


      if (__setitimer (ITIMER_PROF, &otimer, NULL) < 0)
    return -1;
      samples = NULL;
      return __sigaction (SIGPROF, &oact, NULL);
    }

 if (samples)
    {
      /* Was already turned on.  Restore old timer and signal handler
     first.  */
      if (__setitimer (ITIMER_PROF, &otimer, NULL) < 0
      || __sigaction (SIGPROF, &oact, NULL) < 0)
    return -1;
    }
#else
```

```
 /* In ld.so profiling should never be disabled once it runs.  */
 //assert (sample_buffer != NULL);
# define oact_ptr NULL
# define otimer_ptr NULL
#endif

  samples = sample_buffer;
  nsamples = size / sizeof *samples;
  pc_offset = offset;
  pc_scale = scale;

  act.sa_handler = (sighandler_t) &profil_counter;
  act.sa_flags = SA_RESTART;
  __sigfillset (&act.sa_mask);
  if (__sigaction (SIGPROF, &act, oact_ptr) < 0)
    return -1;

  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 1000000 / __profile_frequency ();
  timer.it_interval = timer.it_value;
  return __setitimer (ITIMER_PROF, &timer, otimer_ptr);
}
```

这个函数的主要作用就是定义了一个SIGPROF信号处理函数，并通过__setitimer函数设置SIGPROF的发送频率。这个信号处理函数的功能很关键，后续仍会说明。

b) _mcount 此函数的定义在sysdeps/generic/machine-gmon.h中

```
#define MCOUNT \
void _mcount (void)                             \
{                                       \
  mcount_internal ((u_long) RETURN_ADDRESS (1), (u_long) RETURN_ADDRESS (0)); \
}
```

其中((u_long) RETURN_ADDRESS (nr)调用了__builtin_return_address(nr)函数，__builtin_return_address(nr)会返回当前调 用栈中第nr帧的pc地址。所以(u_long)RETURN_ADDRESS (0)返回的是当前函数地址topc；而(u_long) RETURN_ADDRESS(1)返回的是当前函数的返回地址frompc。

```
__builtin_return_address(LEVEL)

---This function returns the return address of the currentfunction,or of one of its callers. The LEVEL argument is number offrames to scan up the call stack. A value of '0' yields the returnaddress of the current function,a value of '1' yields the returnaddress of the caller of the current function,and so forth.
```

mcount_internal的定义在gmon/mcont.c中

```
_MCOUNT_DECL(frompc, selfpc)    /* _mcount; may be static, inline, etc */
{
  register ARCINDEX *frompcindex;
  register struct tostruct *top, *prevtop;
  register struct gmonparam *p;
  register ARCINDEX toindex;
  int i;
```

```
p = &_gmonparam;
/*
 * check that we are profiling
 * and that we aren't recursively invoked.
 */
if (catomic_compare_and_exchange_bool_acq (&p->state, GMON_PROF_BUSY,
                    GMON_PROF_ON))
  return;

/*
 * check that frompcindex is a reasonable pc value.
 * for example:    signal catchers get called from the stack,
 *        not from text space.  too bad.
 */
frompc -= p->lowpc;
if (frompc > p->textsize)
    goto done;

/* The following test used to be
    if (p->log_hashfraction >= 0)
   But we can simplify this if we assume the profiling data
   is always initialized by the functions in gmon.c.  But
   then it is possible to avoid a runtime check and use the
   smae `if' as in gmon.c.  So keep these tests in sync.  */
```

```
if ((HASHFRACTION & (HASHFRACTION - 1)) == 0) {
  /* avoid integer divide if possible: */
    i = frompc >> p->log_hashfraction;
} else {
    i = frompc / (p->hashfraction * sizeof(*p->froms));
}
frompcindex = &p->froms[i];
toindex = *frompcindex;
if (toindex == 0) {
    /*
     *    first time traversing this arc
     */
    toindex = ++p->tos[0].link;
    if (toindex >= p->tolimit)
        /* halt further profiling */
        goto overflow;

    *frompcindex = toindex;
    top = &p->tos[toindex];
    top->selfpc = selfpc;
    top->count = 1;
    top->link = 0;
    goto done;
}
top = &p->tos[toindex];
```

```
if (top->selfpc == selfpc) {
    /*
     * arc at front of chain; usual case.
     */
    top->count++;
    goto done;
}
/*
 * have to go looking down chain for it.
 * top points to what we are looking at,
 * prevtop points to previous top.
 * we know it is not at the head of the chain.
 */
for (; /* goto done */; ) {
    if (top->link == 0) {
        /*
         * top is end of the chain and none of the chain
         * had top->selfpc == selfpc.
         * so we allocate a new tostruct
         * and link it to the head of the chain.
         */
        toindex = ++p->tos[0].link;
        if (toindex >= p->tolimit)
            goto overflow;
```

```
        top = &p->tos[toindex];
        top->selfpc = selfpc;
        top->count = 1;
        top->link = *frompcindex;
        *frompcindex = toindex;
        goto done;
    }
    /*
     * otherwise, check the next arc on the chain.
     */
    prevtop = top;
    top = &p->tos[top->link];
    if (top->selfpc == selfpc) {
        /*
         * there it is.
         * increment its count
         * move it to the head of the chain.
         */
        top->count++;
        toindex = prevtop->link;
        prevtop->link = top->link;
        top->link = *frompcindex;
        *frompcindex = toindex;
        goto done;
    }
```

```
    }
done:
    p->state = GMON_PROF_ON;
    return;
overflow:
    p->state = GMON_PROF_ERROR;
    return;
}
```

此函数的主要功能就是记录每个函数的调用次数，以及函数之间的调用关系表。并将这些信息保存在全局变量_gmonparam中。**由于此函数是通过hack的方式来调用的（插入入口代码），因此其获取的信息都是精确的。强调z这一点的目的是为了下面将要介绍的另一个主要函数: profil_counter 。**回 溯到gcc的一个步骤，monstartup函数在初始化的最后阶段，通过sigaction调用注册了一个SIGPROF信号处理函数，这个函数 profil_counter。这个函数会以__profile_frequency()的频率被调用，并完成profile的主要工作：收集 sample信息，以此来计算每个函数的消耗时间。

profil_counter函数的定义依赖于具体的系统平台，X86_64平台下的定义是在sysdeps/unix/sysv/linux/x86_64/profil-counter.h中

```
 static void
profil_counter (int signo, SIGCONTEXT scp)
{
 profil_count ((void *) GET_PC (scp));

 /* This is a hack to prevent the compiler from implementing the
    above function call as a sibcall.  The sibcall would overwrite
    the signal context.  */
 asm volatile ("");
}
```

其最终调用的profil_count定义在sysdeps/posix/profil.c中

```
 static inline void
profil_count (void *pc)
{
 size_t i = (pc - pc_offset - (void *) 0) / 2;

 if (sizeof (unsigned long long int) > sizeof (size_t))
   i = (unsigned long long int) i * pc_scale / 65536;
 else
   i = i / 65536 * pc_scale + i % 65536 * pc_scale / 65536;

 if (i < nsamples)
   ++samples[i];
}
```

这段代码的逻辑有点晦涩，需要联系之前的处理逻辑来理解。**pc_offset、pc_scale以及samples这些 全局变量的赋值是在__profil函数中处理的。回溯__profil的逻辑代码，就可以看出samples=_gmonparam-> kcount, 用于保存sample信息，pc_offset =p->lowpc，是程序代码段的起始地址，pc_scale是一个比例因子，用于控制sample的提取粒度。综合上下文，gprof在这里的 处理逻辑是将lowpc~lowpc+65536（linux下默认一个段的大小为64K）范围内的代码映射到一个内存数组，而pc_scale其实就是 决定了映射粒度。对于任何一个处于[lowpc,lowpc+65536]范围内的pc，其对应的数组下标是: pc - lowpc / (65536/ pc_scale) = (pc - lowpc) * pc_scale /65536; 这样一个数组项（一个sample）对应了一段pc_scale长度的程序地址，而每当这段地址内的代码被执行时，相应的sample计数 就会加1。**

c）最后当程序结束时，会调用_mcleanup，其定义在gmon/gmon.c中。

```
 void
_mcleanup (void)
```

```
{
  __moncontrol (0);

  if (_gmonparam.state != GMON_PROF_ERROR)
    write_gmon ();

  /* free the memory. */
  free (_gmonparam.tos);
}
```

首先其通过__moncontrol（0）结束profil工作，其次通过write_gmon ()函数将profile信息输出到gmon.out文件中。

write_gmo函数的定义在gmon/gmon.c中

```
static void
write_gmon (void)
{
    struct gmon_hdr ghdr __attribute__ ((aligned (__alignof__ (int))));
    int fd = -1;
    char *env;

#ifndef O_NOFOLLOW
# define O_NOFOLLOW    0
#endif

    env = getenv ("GMON_OUT_PREFIX");
    if (env != NULL && !__libc_enable_secure)
```

```
  {
size_t len = strlen (env);
char buf[len + 20];
__snprintf (buf, sizeof (buf), "%s.%u", env, __getpid ());
fd = open_not_cancel (buf, O_CREAT|O_TRUNC|O_WRONLY|O_NOFOLLOW, 0666);
  }

if (fd == -1)
  {
fd = open_not_cancel ("gmon.out", O_CREAT|O_TRUNC|O_WRONLY|O_NOFOLLOW,
              0666);
if (fd < 0)
  {
    char buf[300];
    int errnum = errno;
    __fxprintf (NULL, "_mcleanup: gmon.out: %s\n",
        __strerror_r (errnum, buf, sizeof buf));
    return;
  }
  }

/* write gmon.out header: */
memset (&ghdr, '\0', sizeof (struct gmon_hdr));
memcpy (&ghdr.cookie[0], GMON_MAGIC, sizeof (ghdr.cookie));
*(int32_t *) ghdr.version = GMON_VERSION;
```

```
    write_not_cancel (fd, &ghdr, sizeof (struct gmon_hdr));

    /* write PC histogram: */
    write_hist (fd);

    /* write call-graph: */
    write_call_graph (fd);

    /* write basic-block execution counts: */
    write_bb_counts (fd);

    close_not_cancel_no_status (fd);
}
```

通过write_hist、write_call_graph、write_bb_counts这三个子函数，其分别将pc histogram、call-graph以及basic-block execution counts信息输出到gmon.out中。

1.4 gprof的输出分析

在gmon.out文件产生之后，可以通过GNU binutils中提供的工具gprof来分析数据，转换成容易阅读、理解的格式（文字、图片等）。

gprof的主要代码在gprof/gprof.c中

在gmon_out_read函数中，其分别通过hist_read_rec、cg_read_rec、bb_read_rec来读取 gmon.out中对应的pc histogram、call-graph以及basic-block executioncounts信息。在将pchistogram映射到具体函数时间的处理上，gprof采用了一种近似算法：

```
sym_high_pc

sym_low_pc
```

其中，bin_low_pc待用sample数组中的任意一项所对应的PC地址：而bin_high_pc代表bin_low_pc下一个sample对应的PC地址：

bin_low_pc = lowpc + (bfd_vma)(hist_scale * i);

bin_high_pc = lowpc +(bfd_vma) (hist_scale * (i + 1));

sym_low_pc待用可执行程序中某个符号（函数名、段名等）所对应的PC地址，sym_high_pc为下一个符号项所对应的PC地址：

sym_low_pc =symtab.base[j].hist.scaled_addr;

sym_high_pc = symtab.base[j +1].hist.scaled_addr;

gprof只将[bin_low_pc, bin_high_pc]和[sym_low_pc ,sym_high_pc]重合区域（以箭头标识）的sample次数算为sym_low_pc符号的消耗时间。

overlap = MIN (bin_high_pc,sym_high_pc) - MAX (bin_low_pc, sym_low_pc);

credit = overlap * time /hist_scale; // time = sample[i], hist_scale = pc_scale.

1.5 小结

Gprof是GUN 工具链中自带的profiler，无需安装成本，与gcc的结合让其使用方便，能够快速上手。但是gprof也有其一定的缺陷，

1、它的测试结果并不能保证完全准确： 它无法统计程序耗在IO以及swap上的时间：

> The output from gprof gives no indication of parts of your programthat are limited by I/O or swapping bandwidth. This is because samplesof the program counter are taken at fixed intervals of the program'srun time. Therefore, the time measurements in gprof output say nothingabout time that your program was not running. For example, a part ofthe program that creates so much data that

it cannot all fit inphysical memory at once may run very slowly due to thrashing, but gprofwill say it uses little time. On the other hand, sampling by run timehas the advantage that the amount of load due to other users won't directly affect the output you get.

而且，由于其通过采集sample来计算profile的方式，本身就存在一定的失真：

The run-time figures that gprof gives you are based on a samplingprocess, so they are subject to statistical inaccuracy. If a functionruns only a small amount of time, so that on the average the samplingprocess ought to catch that function in the act only once, there is apretty good chance it will actually find that function zero times, ortwice.

By contrast, the number-of-calls figures are derived by counting,not sampling. They are completely accurate and will not vary from runto run if your program is deterministic.

The *sampling period* that is printed at the beginning of theflat profile says how often samples are taken. The rule of thumb isthat a run-time figure is accurate if it is considerably bigger thanthe sampling period.

The actual amount of error is usually more than one sampling period. In fact, if a value is *n* times the sampling period, the *expected* error in it is the square-root of *n*sampling periods. If the sampling period is 0.01 seconds and foo'srun-time is 1 second, the expected error in foo's run-time is 0.1seconds. It is likely to vary this much *on the average* from one profiling run to the next. (*Sometimes* it will vary more.)

This does not mean that a small run-time figure is devoid of information. If the program's *total*run-time is large, a small run-time for one function does tell you thatthat function used an insignificant fraction of the whole program'stime. Usually this means it is not worth optimizing.

2. gprof不能支持动态库的解析。原因在本文中已经分析。

3. gprof不易维护和扩展，因为gprof的代码是封装在GNU工具链的glibc以及binutils中，修改libc的风险较大，而且版本也不易维护（不同系统中使用的libc版本不一致，如果单独更新glibc，会出现程序crash）。

二、 GooglePerformance Tools

Goolgleperformance tools是google公司开发的一套用于C++Profile的工具集。其中包括：

一个优化的内存管理算法—tcmalloc性能优于malloc。

一个用于CPU profile的工具，用于检测程序的性能热点，这个功能和gprof类似。

一个用于堆检查工具，用于检测程序在是够有内存泄露，这个功能和valgrind类似。

一个用于Heap profile的工具，用于监控程序在执行过程的内存使用情况。

官方文档：

http://code.google.com/p/google-perftools/wiki/GooglePerformanceTools

它的使用方式比较简单：首先在编译程序的时候加上相应的链接库，然后在运行程序时

通过设置相应的环境变量来激活工具。

1.使用其提供的内存管理函数---TC Malloc:

```
gcc [...] -ltcmalloc
```

2.使用其堆内存检查工具:

```
gcc [...] -o myprogram -ltcmalloc
HEAPCHECK=normal ./myprogram
```

3.使用Heap Profiler:

```
gcc [...] -o myprogram -ltcmalloc
HEAPPROFILE=/tmp/netheap ./myprogram
```

4.使用Cpu Profiler:

```
gcc [...] -o myprogram -lprofiler
CPUPROFILE=/tmp/profile ./myprogram
```

它的输出也很清晰,下图是一个CpuProfiler的结果图，其中每个方块代码一个函数，方块间的箭头描述了函数之间的调用关系，每个方块里面有 两个数字：X ofY，其中Y表示在程序执行过程中函数所消耗的总体时间，X表示函数自身所消耗的时间，所以Y-X及时函数所调用的子函数消耗时间。如果函数没有子函 数，则只显示总体时间。（X，Y的单位得sample，每个sample所代表的时间可以设置，默认为10ms）

2.1 安装

a) 安装libunwind

libunwind是一个用于解析程序调用栈的C++库，由于glibc内建的栈回滚功能在64位系统上有bug，因此googleperformance tools建议使用libunwind

下载libunwind-0.99-beta.tar.gz

cd $HOME

tarxzvf libunwind-0.99-beta.tar.gz

mkdir libunwind-0.99-beta-build

cd libunwind-0.99-beta

./configure –prefix=$HOME/libunwind-0.99-beta-build

b) 安装Google PerformanceTools

注意：如果在系统目录中找不到libunwind，google performance tools将默认使用glibc的内建功能，因此我们需要手动设置libunwind的安装目录。

下载google-perftools-1.6.tar.gz

cd $HOME

tar xzvf google-perftools-1.6.tar.gz

mkdir google-perftools-1.6-build

cd google-perftools-1.6

./configure –prefix=$HOME/ google-perftools-1.6-build

CPPFLAGS=-I$HOME/libunwind-0.99-beta-build/include

LDFLAGS=-L$HOME/libunwind-0.99-beta-build/lib

make && make install

2.2 用法

参考官方文档。

这里有两点想突出介绍下，一个是对动态库的支持，一个对动态profiler功能的支持。

2.2.1 动态库的支持

在第一章节里面我们已经证明和分析GUNProfiler不提供对动态库的支持，虽然可以通过修改glibc的代码来扩展此功能，但是 维护成本较大。而Goolgle performancetools本身就已经提供了对动态库的支持功能。当然动态库的使用也分两种情况：一种是在运行时动态链接库，一种是在运行时动态加 载库。

运行时链接可以动态地将程序和共享库链接并让 Linux 在执行时加载库（如果它已经在内存中了，则无需再加载）。以一个具体例子来说明：

```
//libtestprofiler.h
extern "C"{
int loopop();
}
```

libtestprofiler.cpp只定义了一个耗时计算函数，便于分析。

```
// libtestprofiler.cpp
#include "libtestprofiler.h"
extern "C"{
int loopop()
{
    int n = 0;
    for(int i = 0; i < 1000000; i++)
        for(int j = 0; j < 10000; j++)
```

```
        {
            n |= i%100 + j/100;
        }
    return n;
}
```

将libtestprofiler.cpp编译为动态库：

g++--shared -fPIC -g -O0 -o libtestprofiler.so libtestprofiler.cpp

在主程序中调用动态库：

```
#include <iostream>
#include "libtestprofiler.h"
using namespace std;

int main(int argc,char** argv)
{
    cout << "loopop: " << loopop() << endl;
    return 1;
}
```

编译主程序，并动态链接libtestprofiler.so：

a) 首先采用GUN Profile的方式编译主程序

g++ -g -O0 -omain main.cpp -ltestprofiler -L. –pg

./main

gprof –b ./main结果如下：

```
Each sample counts as 0.01 seconds.
```

```
no time accumulated


 %   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.00     0.00        1     0.00     0.00  global constructors keyed to main
 0.00      0.00     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
    0.00      0.00     0.00        1     0.00     0.00  data_start
```

和预想一样，GNU Profile 不能解析动态库的性能热点。

b) 再以google CPU Profile的方式编译主程序：

g++ -g -O0 -omain main.cpp -ltestprofiler -L. -lprofiler-L/home/wul/google-perftools-1.6-build/lib

CPUPROFILE=perf.out./main

pprof --text./main ./perf.out，结果如下：

```
Using local file ./main.
Using local file ./perf.out.
Removing killpg from all stack traces.
Total: 5923 samples
    5923 100.0% 100.0%     5923 100.0% loopop
       0   0.0% 100.0%     5923 100.0% __libc_start_main
       0   0.0% 100.0%     5923 100.0% _start
          0   0.0% 100.0%     5923 100.0% main
```

由此证明，Google CPU Profiler支持对动态链接库的性能分析。

运行时加载允许程序可以有选择地调用库中的函数。使用动态加载过程，程序可以先加载一个特定的库（已加载则不必），然后调用该库中的某一特定函数，这是构建支持插件的应用程序的一个普遍的方法。

还是以上述程序为例，对主程序代码进修改：

```
#include <stdio.h>
#include <dlfcn.h>

char LIBPATH[] = "./libtestprofiler.so";
typedef int (*op_t) ();

int main(int argc,char** argv)
{
    void* dl_handle;
    op_t loopop;
    char* error;

    /* Open the shared object */
    dl_handle = dlopen( LIBPATH, RTLD_LAZY );
    if (!dl_handle) {
        printf( "dlopen failed! %s\n", dlerror() );
        return 1;
    }

    /* Resolve the symbol (loopop) from the object */
    loopop = (op_t)dlsym( dl_handle, "loopop");
```

```
    error = dlerror();
    if (error != NULL) {
        printf( "dlsym failed! %s\n", error );
        return 1;
}

    /* Call the resolved loopop and print the result */
    printf("result:  %d\n", (loopop)() );

    /* Close the object */
    dlclose( dl_handle );

    return 0;
}
```

编译：

g++ -g -O0 -o main_dl main_dl.cpp -lprofiler -L/home/wul/google-perftools-1.6-build/lib-ldl

CPUPROFILE=perf_dl.out./main_dl

pprof--text ./main_dl ./perf_dl.out，结果如下：

```
Using local file ./main_dl.
Using local file ./perf_dl.out.
Removing killpg from all stack traces.
Total: 5949 samples
     843  14.2%  14.2%      843  14.2% 0x00002b2f203d25d6
     ……
```

```
    0   0.0% 100.0%       1   0.0% 0x00002b2f203d25ed
    0   0.0% 100.0%    5949 100.0% __libc_start_main
    0   0.0% 100.0%    5949 100.0% _start
    0   0.0% 100.0%    5949 100.0% main
```

很奇怪，这个结果显示libtestprofiler.so库中的符号没有正确解析，perf_dl.out文件也没有包含 libtestprofiler.so的内存映射信息，但是我们确实在主程序已经通过dlopen将动态库装载到内存并执行成功了，为何在主程序的内存映 射表中找不到动态库的信息呢？经过一番分析和调查，终于找到原因，因为perf_dl.out文件的输出工作是在主程序执行结束之后、系统回收资源的时候 调用的（具体见实现原理一节），而在此时主程序执行了dlclose()函数卸载了libtestprofiler.so，所以随后dump出的内存映射 表当然就不会包含libtestprofiler.so的信息了。我们测试下将**dlclose(dl_handle)**注释后的运行结果：

```
Using local file ./main_dl.
Using local file ./perf_dl.out.
Removing killpg from all stack traces.
Total: 5923 samples
   5923 100.0% 100.0%    5923 100.0% loopop
      0   0.0% 100.0%    5923 100.0% __libc_start_main
      0   0.0% 100.0%    5923 100.0% _start
         0   0.0% 100.0%    5923 100.0% main
```

哈哈，动态库中的符号又能正常解析了。

2.2.2 动态profiler功能

这里首先需要解释下何谓动态profiler功能：传统的profiler工具，以GUNProfiler为例，只能编译阶段控制profiler的开关（-fprofile-arcs-ftest-coverage），但是我们有时候需要在程序的运行阶段，或者说运行的中间阶段控制profiler的开 关。Googleperformance tools可以通过CPUPROFILE环境变量在程序运行初阶段控制cpuprofiler的开关，而且根据文档/usr/doc/google- perftools-1.5/pprof_remote_servers.html的提示，可以通过功能扩展可以实现在运行中间阶段或通过http协议远 程控制profiler信息的功能。gperftools-httpd项目就已经初步完成了这个功能，我们可以体验一下。

1．从http://code.google.com/p/gperftools-httpd/下载gperftools-httpd安装。

2．修改下测试程序 main.cpp, 正常运行时间，方便测试

```
#include <iostream>
#include "gperftools-httpd.h"
#include "libtestprofiler.h"
using namespace std;
int main(int argc,char** argv)
{
    ghttpd();
    while(1)
        cout << "loopop: " << loopop() << endl;
    return 1;
}
```

这个程序主要做了两点修改，调用ghttpd()启动一个轻量级web servive，已完成pprof的远程请求服务；通过while循环加长了程序的执行时间，已方便验证动态profiler功能。

3．编译，需要连接libghttpd.so、libprofiler.so

g++-g -O0 -o main main.cpp-I/home/wul/gperftools-httpd-0.2-ltestprofiler -L.-L/home/wul/gperftools-httpd-0.2/ -lghttpd -lprofiler -L/home/wul/google-perftools-1.6-build/lib-dl -lpthread

4. 启动测试程序

./main **注意我们这时并没有设置CPUPROFILE环境变量，即表示此时CPU PROFILE功能还没有打开。**

5．通过pprof工具远程打开测试程序的CPU profile功能：

pprof ./main http://localhost:9999/pprof/profile，结果如下：

```
Using local file ./main.
Gathering CPU profile from http://localhost:9999/pprof/profile?seconds=30 for 30 seconds to
  /home/wul/pprof/main.1292168091.localhost
```

```
Be patient...
Wrote profile to /home/wul/pprof/main.1292168091.localhost
Removing _L_mutex_unlock_15 from all stack traces.
Welcome to pprof!  For help, type 'help'.
(pprof) text
Total: 2728 samples
    2728 100.0% 100.0%     2728 100.0% loopop
       0   0.0% 100.0%     2728 100.0% __libc_start_main
       0   0.0% 100.0%     2728 100.0% _start
       0   0.0% 100.0%     2728 100.0% main
```

从结果中可以看出，当pprof向本地web服务http://localhost:9999/发 送Getpprof/profile请求时，测试程序就会自动开启profile功能，默认的监控时间段是now~now+30s（时间长短可以通过 seconds参数设置），等待30s之后，测试程序停止profile，将结果返回给pprof并保存在/home/wul/pprof /main.1292168091.localhost中，此时再通过text命令就可以看到解析后的输出了。pprof工具还支持其它的query参 数，譬如采样频率控制、触发采样事件等，具体可以参考gperftools-httpd以及google performancetools的官方文档。

2.3 实现原理

Google performance tools包含四大功能，但是本章主要集中介绍CPU profiler功能，以便和GNU profiler做横向对比。

2.3.1 CPU Profile

googleCPU profile的实现方式不同于gprof，但是两个的实现原理有点相似。CPUprofiler是通过设置SIGPROF信号处理函数来采集 sample的，这点和gprof一样，但是CPUprofiler没有在函数入口插入代码，而是通过保存调用栈信息来记录函数的调用图和调用次数。 CPUprofiler的主要实现代码在src/profiler.cc中。这个文件中定义了一个CpuProfiler类，并声明一个该类的静态实例。 这样在main函数之前，此静态实例就会被初始化。

```
// Initialize profiling: activated if getenv("CPUPROFILE") exists.
```

```
CpuProfiler::CpuProfiler()
    : prof_handler_token_(NULL) {
  // TODO(cgd) Move this code *out* of the CpuProfile constructor into a
  // separate object responsible for initialization. With ProfileHandler there
  // is no need to limit the number of profilers.  charfname[PATH_MAX];  if
(!GetUniquePathFromEnv("CPUPROFILE", fname)) {    return;
  }
  // We don't enable profiling if setuid -- it's a security risk
#ifdef HAVE_GETEUID
  if (getuid() != geteuid())
    return;
#endif
  if (!Start(fname, NULL)) {
    RAW_LOG(FATAL, "Can't turn on cpu profiling for '%s': %s\n",
            fname, strerror(errno));
  }
}
```

该构造函数首先会判断系统变量CPUPROFILE是否被设置，如果设置了，则启动CPU profiler进程，否则，直接返回。我们在看看Start函数做了什么：

```
bool CpuProfiler::Start(const char* fname, const ProfilerOptions* options) {
  SpinLockHolder cl(&lock_);
```

```
  if (collector_.enabled()) {
    return false;
  }

  ProfileHandlerState prof_handler_state;
  ProfileHandlerGetState(&prof_handler_state);

  ProfileData::Options collector_options;
  collector_options.set_frequency(prof_handler_state.frequency);
  if (!collector_.Start(fname, collector_options)) {
    return false;
  }

  filter_ = NULL;
  if (options != NULL && options->filter_in_thread != NULL) {
    filter_ = options->filter_in_thread;
    filter_arg_ = options->filter_in_thread_arg;
  }

  // Setup handler for SIGPROF interrupts
  EnableHandler();

  return true;
}
```

此函数首先会调用ProfileHandlerGetState来获取其它的控制参数，包括CPUPROFILE_REALTIME和CPUPROFILE_FREQUENCY。

CPUPROFILE_FREQUENCY=x	default: 100	How many interrupts/second the cpu-profiler samples.
CPUPROFILE_REALTIME=1	default: [not set]	If set to any value (including 0 or the empty string), useITIMER_REAL instead of ITIMER_PROF to gather profiles. In general,ITIMER_REAL is not as accurate as ITIMER_PROF, and also interacts badlywith use of alarm(), so prefer ITIMER_PROF unless you have a reasonprefer ITIMER_REAL.

其次，函数调用ProfileData::Start为记录profiler信息分配内存并初始化，其定义在profiledata.cc中。

```
bool ProfileData::Start(const char* fname,
                        const ProfileData::Options& options) {
 if (enabled()) {
   return false;
 }

 // Open output file and initialize various data structures
 int fd = open(fname, O_CREAT | O_WRONLY | O_TRUNC, 0666);
 if (fd < 0) {
   // Can't open outfile for write
```

```
  return false;
}

start_time_ = time(NULL);
fname_ = strdup(fname);

// Reset counters
num_evicted_ = 0;
count_       = 0;
evictions_   = 0;
total_bytes_ = 0;

hash_ = new Bucket[kBuckets];
evict_ = new Slot[kBufferLength];
memset(hash_, 0, sizeof(hash_[0]) * kBuckets);
// Record special entries
evict_[num_evicted_++] = 0;                  // count for header
evict_[num_evicted_++] = 3;                  // depth for header
evict_[num_evicted_++] = 0;                  // Version number
CHECK_NE(0, options.frequency());
int period = 1000000 / options.frequency();
evict_[num_evicted_++] = period;               // Period (microseconds)
evict_[num_evicted_++] = 0;                  // Padding

out_ = fd;
```

```
  return true;
}
```

其中slot数组evict_就是profiler输出文件中的保存内容，具体可参考profiler输出文件的格式说明。Bucket数组hash_是用于临时保存程序调用栈信息的hash表，num_evicted记录evict_数组中的有效长度。这些变量在后续将会经常出现。回到profiler.cc中的CpuProfiler::Start函数，其最后一步调用的是EnableHandler()， 用于设置SIGPROF的信号处理函数。

```
void CpuProfiler::EnableHandler() {
RAW_CHECK(prof_handler_token_ == NULL, "SIGPROF handler already registered");
prof_handler_token_ = ProfileHandlerRegisterCallback(prof_handler, this);
RAW_CHECK(prof_handler_token_ != NULL, "Failed to set up SIGPROF handler");
}
```

函数通过ProfileHandlerRegisterCallback注册了一个回调函数prof_handler：

```
ProfileHandlerToken* ProfileHandler::RegisterCallback(
ProfileHandlerCallback callback, void* callback_arg) {
ProfileHandlerToken* token = new ProfileHandlerToken(callback, callback_arg);

SpinLockHolder cl(&control_lock_);
DisableHandler();
{
```

```
SpinLockHolder sl(&signal_lock_);
callbacks_.push_back(token);
}
// Start the timer if timer is shared and this is a first callback.
if ((callback_count_ == 0) && (timer_sharing_ == TIMERS_SHARED)) {
StartTimer();
}
++callback_count_;
EnableHandler();
return token;
}
```

紧接着通过ProfileHandler::EnableHandler注册SIGPROF信号处理函数SignalHandler。

```
void ProfileHandler::EnableHandler() {
struct sigaction sa;
sa.sa_sigaction = SignalHandler;
sa.sa_flags = SA_RESTART | SA_SIGINFO;
sigemptyset(&sa.sa_mask);
const int signal_number = (timer_type_ == ITIMER_PROF ? SIGPROF : SIGALRM);
RAW_CHECK(sigaction(signal_number, &sa, NULL) == 0, "sigprof (enable)");
}
```

到此，CPU profile的初始化工作基本上都完成了，总结一下主要是完成了两个工作：一个是内存的分配以及初始化，一个是注册SIGPROF信号处理函数，以便采集sample信息。所以接下来的重点将是分析CPU profile是如何采集sample的。首先看看SignalHandler函数的定义：

```
void ProfileHandler::SignalHandler(int sig, siginfo_t* sinfo, void* ucontext) {
  int saved_errno = errno;
  RAW_CHECK(instance_ != NULL, "ProfileHandler is not initialized");
  {
    SpinLockHolder sl(&instance_->signal_lock_);
    ++instance_->interrupts_;
    for (CallbackIterator it = instance_->callbacks_.begin();
         it != instance_->callbacks_.end();
         ++it) {
      (*it)->callback(sig, sinfo, ucontext, (*it)->callback_arg);
    }
  }
  errno = saved_errno;
}
```

从代码中可以看出，SignalHandler除了记录中断次数之外，遍历调用了callbacks_链中的所有回调函数，回溯CPU Profile前面的初始化工作，这里就会调用prof_handler函数：

```
// Signal handler that records the pc in the profile-data structure. We do no
// synchronization here.  profile-handler.cc guarantees that at most one
// instance of prof_handler() will run at a time. All other routines that
```

```
// access the data touched by prof_handler() disable this signal handler before
// accessing the data and therefore cannot execute concurrently with
// prof_handler().
void CpuProfiler::prof_handler(int sig, siginfo_t*, void* signal_ucontext,
                               void* cpu_profiler) {
  CpuProfiler* instance = static_cast<CpuProfiler*>(cpu_profiler);

  if (instance->filter_ == NULL ||
      (*instance->filter_)(instance->filter_arg_)) {
    void* stack[ProfileData::kMaxStackDepth];
    // The top-most active routine doesn't show up as a normal
    // frame, but as the "pc" value in the signal handler context.
    stack[0] = GetPC(*reinterpret_cast<ucontext_t*>(signal_ucontext));

    // We skip the top two stack trace entries (this function and one
    // signal handler frame) since they are artifacts of profiling and
    // should not be measured.  Other profiling related frames may be
    // removed by "pprof" at analysis time.  Instead of skipping the top
    // frames, we could skip nothing, but that would increase the
    // profile size unnecessarily.
    int depth = GetStackTraceWithContext(stack + 1, arraysize(stack) - 1,
                                         2, signal_ucontext);
    depth++;  // To account for pc value in stack[0];

    instance->collector_.Add(depth, stack);
```

```
  }
}
```

从代码的注解片段中可以理解此函数的主要工作就是记录将当前程序的调用栈信息。顾名思义，GetPC函数用于获取当前pc指针，它是利用linux系统的信号处理机制来获取当前pc的（具体可参考《unix环境高级编程》），其主要实现代码在getpc.h中：

```
inline void* GetPC(const ucontext_t& signal_ucontext) {
//    fprintf(stderr,"In GetPC3");
    return (void*)signal_ucontext.PC_FROM_UCONTEXT;   // defined in config.h
}
```

GetStackTraceWithContext函数完成了cpu profiler过程中最重要的一步，它最终调用了libunwind库，dump出了当前的函数调用栈信息，其主要实现代码在stacktrace_libunwind-inl.h中：

```
int GET_STACK_TRACE_OR_FRAMES {
  fprintf(stderr,"in libunwind\n");
  void *ip;
  int n = 0;
  unw_cursor_t cursor;
  unw_context_t uc;
#if IS_STACK_FRAMES
  unw_word_t sp = 0, next_sp = 0;
#endif
```

```
  if (recursive) {
    return 0;
  }
  ++recursive;
  unw_getcontext(&uc);
  int ret = unw_init_local(&cursor, &uc);
  assert(ret >= 0);
  skip_count++;         // Do not include current frame

  while (skip_count--) {
      if (unw_step(&cursor) <= 0) {
          goto out;
      }
#if IS_STACK_FRAMES
      if (unw_get_reg(&cursor, UNW_REG_SP, &next_sp)) {
          goto out;
      }
#endif
  }

  while (n < max_depth) {
      if (unw_get_reg(&cursor, UNW_REG_IP, (unw_word_t *) &ip) < 0) {
          break;
      }
```

```
#if IS_STACK_FRAMES
      sizes[n] = 0;
#endif
    result[n++] = ip;
    if (unw_step(&cursor) <= 0) {
      break;
    }
#if IS_STACK_FRAMES
    sp = next_sp;
    if (unw_get_reg(&cursor, UNW_REG_SP, &next_sp) , 0) {
      break;
    }
    sizes[n - 1] = next_sp - sp;
#endif
  }
out:
  --recursive;
  return n;
```

这个函数的过程有点复杂，它的主要功能是回滚当前调用栈，并将栈指针都保存在stack数组中，根据这些信息就可以记录程序指令的执行次数，以及描述函数之间的调用关系图。（具体实现原理请参考libunwind官网说明）。再对到prof_handler函数中，程序的最后一步就是将当前获取的调用栈信息保存到预先分配的内存中，其具体实现在profiledata.cc文件中：

```
void ProfileData::Add(int depth, const void* const* stack) {
```

```
if (!enabled()) {
  return;
}

if (depth > kMaxStackDepth) depth = kMaxStackDepth;
RAW_CHECK(depth > 0, "ProfileData::Add depth <= 0");

// Make hash-value
Slot h = 0;
for (int i = 0; i < depth; i++) {
  Slot slot = reinterpret_cast<Slot>(stack[i]);
  h = (h << 8) | (h >> (8*(sizeof(h)-1)));
  h += (slot * 31) + (slot * 7) + (slot * 3);
}

count_++;

// See if table already has an entry for this trace
bool done = false;
Bucket* bucket = &hash_[h % kBuckets];
for (int a = 0; a < kAssociativity; a++) {
  Entry* e = &bucket->entry[a];
  if (e->depth == depth) {
    bool match = true;
    for (int i = 0; i < depth; i++) {
```

```
      if (e->stack[i] != reinterpret_cast<Slot>(stack[i])) {
        match = false;
        break;
      }
    }
    if (match) {
      e->count++;
      done = true;
      break;
    }
  }
}

if (!done) {
  // Evict entry with smallest count
  Entry* e = &bucket->entry[0];
  for (int a = 1; a < kAssociativity; a++) {
    if (bucket->entry[a].count < e->count) {
      e = &bucket->entry[a];
    }
  }
  if (e->count > 0) {
    evictions_++;
    Evict(*e);
  }
```

```
    // Use the newly evicted entry
    e->depth = depth;
    e->count = 1;
    for (int i = 0; i < depth; i++) {
      e->stack[i] = reinterpret_cast<Slot>(stack[i]);
    }
  }
}
```

此函数的处理流程如下:

1．对stack数组的所有项做hash，得到一个hash值；

2．根据hash值在hash_表中查找此调用栈，如果找到匹配项则增加该项的执行次数；

3．如果没有找到则将从相应的hash槽中pop出执行次数最少的一个调用栈，将此调用栈中的所有栈指针值按顺序保存到evict_数组中，并将新调用栈push到hash槽中。

到此，CPU profile的主要流程都走完了，总结一下其一直在循环执行一个动作：定期保存程序的当前调用栈信息。在被测程序执行结束之后，CPU profile所做的最后一步工作就是将evict_数组中保存的数据输出到%CPUPROFILER环境变量制定的文件中（profiledata.cc）：

```
void ProfileData::Stop() {
  if (!enabled()) {
    return;
  }

  // Move data from hash table to eviction buffer
```

```
for (int b = 0; b < kBuckets; b++) {
  Bucket* bucket = &hash_[b];
  for (int a = 0; a < kAssociativity; a++) {
    if (bucket->entry[a].count > 0) {
      Evict(bucket->entry[a]);
    }
  }
}

if (num_evicted_ + 3 > kBufferLength) {
  // Ensure there is enough room for end of data marker
  FlushEvicted();
}

// Write end of data marker
evict_[num_evicted_++] = 0;          // count
evict_[num_evicted_++] = 1;          // depth
evict_[num_evicted_++] = 0;          // end of data marker
FlushEvicted();

// Dump "/proc/self/maps" so we get list of mapped shared libraries
DumpProcSelfMaps(out_);

Reset();
fprintf(stderr, "PROFILE: interrupts/evictions/bytes = %d/%d/%" PRIuS "\n",
```

```
        count_, evictions_, total_bytes_);
}
```

在dump出evict_数组数据之后，函数还通过DumpProcSelfMaps将/prof/self/map中的信息追加到输出文件中，这 些信息记录了应用程序的内存映射情况，是pprof工具解析指令符号的重要依据。（关于/prof/self/map中的信息说明可以参考《程序员的自我 修养》）

虽然监控程序已经停止，但是CPUprofiler的工作还没完全结束，因为之前保存在$CPUPROFILER文件中的数据都是二进制格式的，不具备可读性，需要借助pprof工具的解析功能才能揭露它的真实信息。

pprof是用perl语言编写的解析工具，它的主要功能就是将CPU profile的输出数据转换成容易阅读理解的可视格式，如text、pdf、gif等，接下来本文将讲解pprof的主要工作原理，具体细节可以参考pprof代码。

$CPUPROFILER文件中保存了两部分信息：前部分是定期dump的调用栈信息，每个调用栈信息中都包含了执行次数、栈深度以及 栈指针值（即指令地址）；后半部分记录应用程序的内存映射图。所以第一步，pprof根据内存映射图和程序符号表将调用栈中的指令地址翻译成容易理解的程 序代码；第二步，pprof根据第一部分保存的栈信息描述出程序中的函数调用图；最后一步，pprof根据栈执行次数计算出每段代码的执行次数，再根据定 时器的执行频率估算出程序段的执行时间，进而找出程序的性能热点。

2.4 小结

Google performance tools采用了和GUNProfiler近似的原理、不同的方式来达到profiler的效果。由于其通过记录调用栈信息来反推程序段的执行次数，不可 避免地会出现遗漏和误算情况，而且和GUNProfiler一样，它也是通过sample的采样频率来估算程序段的运行时间，因此最终计算结果并不是十分 精确的，具有一定的误差。但是，Google performance tools较之其他Profiler工具而言，有其自身的特点和优势，Googleperformance tools是一个用户态程序，不需要内核提供支持（对比oprofiler）；它对被监控程序的入侵程序度较小（对比GUNProfiler），无需修改 程序代码，以attach的方式跟踪程序执行状态；而且它也是google的开源项目之一，工程量较小，方面后期扩展和二次开发。

三、 C++ Profiler工具特性对比

总结前两章的调研结果，对目前常用的C++ profiler工具做了一个简单的对比，对比的焦点主要集中在日常使用中大家所发现或比较关注的问题。不过由于时间关系，所选工具和对比项都十分有限，希望能在后期的进一步工作中完善补充。

C++Profiler工具	精确度	对动态库的支持	对动态控制的支持	二次开发和维护成本
GUN profile	较高，对函数执行次数的统计是100%正确的，但是对函数执行时间的统计是通过采样平率估算的，存在一定的偏差。	No	编译时决定，灵活性较差	代码集成在glibc中，二次开发和修改的影响面较大，而且发布不易。
Google performance tools	一般，对函数次数和执行时间的统计都是通过采样频率估算的，存在一定的偏差和遗漏。	Yes	运行时控制，更方面操作	独立的第三方库，开源项目，二次开发和维护成本较低。
Oprofile	待调查	待调查	待调查	待调查

未完待续……

标签: c++, c++profile

Likwo
关注 - 30
粉丝 - 417
+加关注

« 上一篇：c++ source code ide
» 下一篇：使用Google的开源TCMalloc库，提高MySQL在高并发情况下的性能

posted on 2012-12-20 20:17 Likwo 阅读(8997) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

最新知识库文章:

· 写给初学前端工程师的一封信
· 实用VPC虚拟私有云设计原则
· 如何阅读计算机科学类的书
· Google 及其云智慧
· 做到这一点，你也可以成为优秀的程序员
» 更多知识库文章...