**Google CPU Profiler使用指南及小工具**

[搜索技术博客－淘宝](http://ju.outofmemory.cn/feed/75/) 2012-12-28 **570** 阅读

[性能优化](http://ju.outofmemory.cn/tag/%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96/)

**Google CPU Profiler使用指南及小工具**  
–五竹,2012/12/27

**(一) gperftools-2.0 编译**

wget https://gperftools.googlecode.com/files/gperftools-2.0.tar.gz  
tar –xzf gperftools-2.0.tar.gz  
cd gperftools-2.0

./configure –prefix=/home/wuzhu/tools/gperftools –enable-frame-pointers

make && make install

注：  
编译时打开了 –enable-frame-pointers ，这要求被测试的程序在编译时要加上gcc编译选项，否则某些多线程程序可能会 core:  
CCFLAGS=-fno-omit-frame-pointer

**(二) google cpu profiler 基本使用**  
**嵌入 google cpu profiler 代码与编译**  
1) 在我们要测试的程序源码中先 include 头文件，然后在要 profile 的代码前后加上ProfilerStart() 和 ProfilerStop()  
其接口声明如下：

#include <google/profiler.h>

int ProfilerStart(const char\* fname);

void ProfilerFlush();

void ProfilerStop();

2) 在编译时加上 -fno-omit-frame-pointer 和 链接库 -ltcmalloc\_and\_profiler  
CCFLAGS=-fno-omit-frame-pointer

-ltcmalloc\_and\_profiler

3) 执行程序，生成 profile 数据文件  
4) 分析生成的数据文件

% pprof /bin/ls ls.prof  
Enters “interactive” mode  
% pprof –text /bin/ls ls.prof  
Outputs one line per procedure  
% pprof –gv /bin/ls ls.prof  
Displays annotated call-graph via ‘gv’  
% pprof –gv –focus=Mutex /bin/ls ls.prof  
Restricts to code paths including a .\*Mutex.\* entry  
% pprof –gv –focus=Mutex –ignore=string /bin/ls ls.prof  
Code paths including Mutex but not string  
% pprof –list=getdir /bin/ls ls.prof  
(Per-line) annotated source listing for getdir()  
% pprof –disasm=getdir /bin/ls ls.prof  
(Per-PC) annotated disassembly for getdir()  
% pprof –text localhost:1234  
Outputs one line per procedure for localhost:1234  
% pprof –callgrind /bin/ls ls.prof  
Outputs the call information in callgrind format

**示例一：**  
1) cpu\_profiler\_example.cpp

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

#include <gperftools/profiler.h>

using namespace std;

int loopop()

{

char buffer1 [1024];

char buffer2 [1024];

int n = 0;

for(int i = 0; i < 10000000; i++)

{

for(int j = 0; j < 1000; j++)

{

n |= i%100 + j/100;

}

memset(buffer1,0xa,1024);

memcpy(buffer2,buffer1,1024);

}

return n;

}

int main(int argc,char\*\* argv)

{

char program[1024]={0};

snprintf(program,1023,"%s\_%d.prof",argv[0],getpid());

ProfilerStart(program);

printf("result: %d\n", (loopop)() );

ProfilerStop();

return 0;

}

2) Makefile

GPerfTools=/home/wuzhu/tools/gperftools

CCFLAGS=-fno-omit-frame-pointer -g

ALL\_BINS=cpu\_profiler\_example  
all:$(ALL\_BINS)

cpu\_profiler\_example :cpu\_profiler\_example.o  
g++ $(CCFLAGS) -o $@ $^ -L./ -L$(GPerfTools)/lib -Wl,-Bdynamic -ltcmalloc\_and\_profiler

.cpp.o:  
g++ $(CCFLAGS) -c -I./ -I$(GPerfTools)/include -fPIC -o $@ $<  
clean:  
rm -f $(ALL\_BINS) \*.o

3) 执行

[wuzhu@search041142.sqa.cm4 cpu\_profiler]$ ./cpu\_profiler\_example  
loopop: 255  
PROFILE: interrupts/evictions/bytes = 5848/3861/185584

会产生性能数据文件: cpu\_profiler\_example\_29502.prof

4). 分析性能数据

pprof –text cpu\_profiler\_example cpu\_profiler\_example\_29502.prof

Using local file cpu\_profiler\_example.  
Using local file cpu\_profiler\_example\_29502.prof.  
Removing killpg from all stack traces.  
Removing main from all stack traces.  
Removing \_\_libc\_start\_main from all stack traces.  
Total: 5962 samples  
5894 98.9% 98.9% 5894 98.9% loopop  
38 0.6% 99.5% 38 0.6% memcpy  
28 0.5% 100.0% 28 0.5% memset  
2 0.0% 100.0% 2 0.0% \_init  
注：  
更详细的使用方法请见 google cpu profiler wiki:

http://gperftools.googlecode.com/svn/trunk/doc/cpuprofile.html

**(三) Google CPU Profiler支持对动态链接库进行性能分析**  
下面，通过示例2来说明 Google Cpu Profiler 是支持对动态链接库进行性能分析的。

1) 源代码

//TestProfiler.h

extern "C"

{

int loopop();

}

//TestProfiler.cpp只定义了一个耗时计算函数，便于分析。

#include <string.h>

#include "TestProfiler.h"

extern "C"

{

int loopop()

{

char buffer1 [1024];

char buffer2 [1024];

int n = 0;

for(int i = 0; i < 10000000; i++)

{

for(int j = 0; j < 1000; j++)

{

n |= i%100 + j/100;

}

memset(buffer1,0xa,1024);

memcpy(buffer2,buffer1,1024);

}

return n;

}

}

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

#include <gperftools/profiler.h>

#include "TestProfiler.h"

using namespace std;

int main(int argc,char\*\* argv)

{

char program[1024]={0};

snprintf(program,1023,"%s\_%d.prof",argv[0],getpid());

ProfilerStart(program);

printf("result: %d\n", (loopop)() );

ProfilerStop();

return 0;

}

2) Makefile

GPerfTools=/home/wuzhu/tools/gperftools

CCFLAGS=-fno-omit-frame-pointer -g

ALL\_BINS=libTestProfiler.so main\_dynamic\_link  
all:$(ALL\_BINS)

main\_dynamic\_link :main\_dynamic\_link.o  
g++ $(CCFLAGS) -o $@ $^ -L./ -L$(GPerfTools)/lib -Wl,-Bdynamic -lTestProfiler -ltcmalloc\_and\_profiler

libTestProfiler.so:TestProfiler.o  
g++ -shared $(CCFLAGS) -o $@ $^

.cpp.o:  
g++ $(CCFLAGS) -c -I./ -I$(GPerfTools)/include -fPIC -o $@ $<  
clean:  
rm -f $(ALL\_BINS) \*.o \*.prof

3) 执行

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:`pwd`  
./main\_dynamic\_link

result: 127  
PROFILE: interrupts/evictions/bytes = 5991/3915/188368

产生性能数据文件: main\_dynamic\_link\_24327.prof

4) 分析性能数据

pprof –text main\_dynamic\_link main\_dynamic\_link\_24327.prof

Using local file main\_dynamic\_link.  
Using local file main\_dynamic\_link\_24327.prof.  
Removing killpg from all stack traces.  
Removing main from all stack traces.  
Removing \_\_libc\_start\_main from all stack traces.  
Total: 5991 samples  
5899 98.5% 98.5% 5899 98.5% loopop  
53 0.9% 99.3% 53 0.9% memcpy  
38 0.6% 100.0% 38 0.6% memset  
1 0.0% 100.0% 1 0.0% \_init

由此证明，Google CPU Profiler支持对动态链接库的性能分析

**(四) Google CPU Profiler 对用 dlopen 方式打开动态库的程序支持**  
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。  
示例 3 主要演示 Google CPU Profiler 是否支持对用 dlopen 方式打开动态库的程序进行性能分析。

还是以上述程序为例，对主程序代码进修改。

1) 源代码 (libTestProfiler.so 的源码同示例 2)

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

#include <gperftools/profiler.h>

typedef int (\*op\_t) ();

int main(int argc,char\*\* argv)

{

int not\_close=0;

if(argc >= 2)

not\_close=atoi(argv[1]);

void\* dl\_handle=NULL;

op\_t loopop;

char\* error=NULL;

char program[1024]={0};

snprintf(program,1023,"%s\_%d.prof",argv[0],getpid());

ProfilerStart(program);

dl\_handle = dlopen("./libTestProfiler.so", RTLD\_LAZY );

if (!dl\_handle)

{

printf( "dlopen failed! %s\n", dlerror() );

return -1;

}

loopop = (op\_t)dlsym( dl\_handle, "loopop");

error = dlerror();

if (error != NULL)

{

printf( "dlsym failed! %s\n", error );

return -1;

}

printf("result: %d\n", (loopop)() );

if(not\_close == 0)

{

printf("do dlclose()\n");

dlclose( dl\_handle );

}

else

{

printf("not dlclose()\n");

}

ProfilerStop();

return 0;

}

2) Makefile

GPerfTools=/home/wuzhu/tools/gperftools

CCFLAGS=-fno-omit-frame-pointer -g

ALL\_BINS=libTestProfiler.so main\_dlopen\_link  
all:$(ALL\_BINS)  
main\_dlopen\_link :main\_dlopen\_link.o  
g++ $(CCFLAGS) -o $@ $^ -L./ -L$(GPerfTools)/lib -Wl,-Bdynamic -ltcmalloc\_and\_profiler -ldl

libTestProfiler.so:TestProfiler.o  
g++ -shared $(CCFLAGS) -o $@ $^

.cpp.o:  
g++ $(CCFLAGS) -c -I./ -I$(GPerfTools)/include -fPIC -o $@ $<  
clean:  
rm -f $(ALL\_BINS) \*.o \*.prof

3) 运行

[wuzhu@search041142.sqa.cm4 cpu\_profiler]$ ./main\_dlopen\_link  
result: 127  
do dlclose()  
PROFILE: interrupts/evictions/bytes = 5984/3559/171280

产生性能分析数据：main\_dlopen\_link\_1256.prof

4) 分析性能数据  
pprof –text main\_dlopen\_link main\_dlopen\_link\_13598.prof

Using local file main\_dlopen\_link.  
Using local file main\_dlopen\_link\_13598.prof.  
Removing killpg from all stack traces.  
Removing main from all stack traces.  
Removing \_\_libc\_start\_main from all stack traces.  
Total: 5984 samples  
801 13.4% 13.4% 801 13.4% 0x00002b124c0b4688  
736 12.3% 25.7% 736 12.3% 0x00002b124c0b4652  
735 12.3% 38.0% 735 12.3% 0x00002b124c0b4685  
425 7.1% 45.1% 425 7.1% 0x00002b124c0b4671  
414 6.9% 52.0% 414 6.9% 0x00002b124c0b464b  
412 6.9% 58.9% 412 6.9% 0x00002b124c0b4676  
401 6.7% 65.6% 401 6.7% 0x00002b124c0b4656  
401 6.7% 72.3% 401 6.7% 0x00002b124c0b465c  
392 6.6% 78.8% 392 6.6% 0x00002b124c0b4624  
387 6.5% 85.3% 387 6.5% 0x00002b124c0b463e  
382 6.4% 91.7% 382 6.4% 0x00002b124c0b467f  
368 6.1% 97.8% 368 6.1% 0x00002b124c0b4639  
45 0.8% 98.6% 45 0.8% memcpy  
42 0.7% 99.3% 42 0.7% memset  
21 0.4% 99.6% 21 0.4% 0x00002b124c0b468c  
9 0.2% 99.8% 9 0.2% 0x00002b124c0b4645  
3 0.1% 99.8% 3 0.1% 0x00002b124c0b467d  
3 0.1% 99.9% 3 0.1% 0x00002b124c0b469c  
2 0.0% 99.9% 2 0.0% 0x00002b124c0b46a6  
1 0.0% 99.9% 1 0.0% 0x00002b124c0b4518  
1 0.0% 99.9% 1 0.0% 0x00002b124c0b461b  
1 0.0% 100.0% 1 0.0% 0x00002b124c0b4643  
1 0.0% 100.0% 1 0.0% 0x00002b124c0b4695  
1 0.0% 100.0% 1 0.0% 0x00002b124c0b46d4

很奇怪，这个结果显示 libTestProfiler.so 库中的符号没有正确解析，main\_dlopen\_link\_1256.prof 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main\_dlopen\_link\_1256.prof 文件的输出工作是在dlclose()函数执行之后调用的，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。

下面，我们通过传入参数，禁止dlclose()

[wuzhu@search041142.sqa.cm4 cpu\_profiler]$ ./main\_dlopen\_link 1  
result: 127  
not dlclose()  
PROFILE: interrupts/evictions/bytes = 5968/3520/169408

pprof –text main\_dlopen\_link main\_dlopen\_link\_18394.prof

Using local file main\_dlopen\_link.  
Using local file main\_dlopen\_link\_18394.prof.  
Removing killpg from all stack traces.  
Removing main from all stack traces.  
Removing \_\_libc\_start\_main from all stack traces.  
Total: 5968 samples  
5893 98.7% 98.7% 5893 98.7% loopop  
46 0.8% 99.5% 46 0.8% memcpy  
27 0.5% 100.0% 27 0.5% memset  
2 0.0% 100.0% 2 0.0% \_init

此时，动态库中的符号又能正常解析了。

**(五) GoogleProfiler.h 小工具**

#ifndef \_\_GOOGLE\_PROFILER\_\_

#define \_\_GOOGLE\_PROFILER\_\_

#include <stdio.h>

#include <stdint.h>

#include <string>

#ifdef CPU\_PROFILER

#include <google/profiler.h>

#endif

class GoogleProfiler

{

public:

enum

{

MAXBUFSIZE=1024

};

public:

GoogleProfiler():\_ready(false)

{

}

~GoogleProfiler()

{

ProfilerStop();

}

void ProfilerStart()

{

#ifdef CPU\_PROFILER

char buf[MAXBUFSIZE]={0};

int count = readlink("/proc/self/exe",buf, MAXBUFSIZE );

if(count < 0 || count > MAXBUFSIZE)

return;

\_programBin.assign(buf);

printf("ProgramBin=%s\n",\_programBin.c\_str());

\_pid=getpid();

size\_t pos=\_programBin.find\_last\_of('/');

if(pos != std::string::npos)

{

\_programName=\_programBin.substr(pos+1);

}

else

{

\_programName=\_programBin;

}

printf("ProfilerStart(%s)\n",buf);

snprintf(buf,MAXBUFSIZE,"./%s\_%d.prof",\_programName.c\_str(),\_pid);

::ProfilerStart(buf);

\_ready=true;

#endif

}

void ProfilerStop()

{

#ifdef CPU\_PROFILER

if(\_ready)

{

::ProfilerStop();

char cmd[2048]={0};

snprintf(cmd,2047,"pprof --callgrind %s %s\_%d.prof >%s\_%d.callgrind"

,\_programBin.c\_str(),\_programName.c\_str(),\_pid,\_programName.c\_str(),\_pid);

printf("cmd=%s\n",cmd);

system(cmd);

\_ready=false;

}

#endif

}

void ProfilerFlush()

{

::ProfilerFlush();

}

private:

bool \_ready;

int \_pid;

std::string \_programBin;

std::string \_programName;

};

#endif

该小工具主要是提供如下功能：  
″ 自动生成性能数据文件的名称：程序名\_进程号.prof  
″ 将性能数据文件自动转换成 .callgrind 格式，方便 kcachegrind 上展示  
″ 提供宏定义，可通过是否传 -DCPU\_PROFILER 宏来打开或关闭google cpu profiler 性能采集.

下面是使用示例:

ν Main 程序

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

#include <gperftools/profiler.h>

#include "GoogleProfiler.h"

typedef int (\*op\_t) ();

int main(int argc,char\*\* argv)

{

int not\_close=0;

if(argc >= 2)

not\_close=atoi(argv[1]);

void\* dl\_handle=NULL;

op\_t loopop;

char\* error=NULL;

GoogleProfiler profiler;

profiler.ProfilerStart();

dl\_handle = dlopen("./libTestProfiler.so", RTLD\_LAZY );

if (!dl\_handle)

{

printf( "dlopen failed! %s\n", dlerror() );

return -1;

}

loopop = (op\_t)dlsym( dl\_handle, "loopop");

error = dlerror();

if (error != NULL)

{

printf( "dlsym failed! %s\n", error );

return -1;

}

printf("result: %d\n", (loopop)() );

if(not\_close == 0)

{

printf("do dlclose()\n");

dlclose( dl\_handle );

}

else

{

printf("not dlclose()\n");

}

profiler.ProfilerStop();

return 0;

}

ν Makefile

GPerfTools=/home/wuzhu/tools/gperftools

CCFLAGS=-fno-omit-frame-pointer -g -DCPU\_PROFILER

ALL\_BINS=libTestProfiler.so \  
main\_dlopen\_link2

all:$(ALL\_BINS)

main\_dlopen\_link2 :main\_dlopen\_link2.o  
g++ $(CCFLAGS) -o $@ $^ -L./ -L$(GPerfTools)/lib -Wl,-Bdynamic -ltcmalloc\_and\_profiler -ldl

libTestProfiler.so:TestProfiler.o  
g++ -shared $(CCFLAGS) -o $@ $^

.cpp.o:  
g++ $(CCFLAGS) -c -I./ -I$(GPerfTools)/include -fPIC -o $@ $<  
clean:  
rm -f $(ALL\_BINS) \*.o \*.prof

ν 执行  
[wuzhu@search041142.sqa.cm4 cpu\_profiler]$ ./main\_dlopen\_link2 1  
ProgramBin=/home/wuzhu/develop/mytest/cpu\_profiler/main\_dlopen\_link2  
ProfilerStart(/home/wuzhu/develop/mytest/cpu\_profiler/main\_dlopen\_link2)  
result: 127  
not dlclose()  
PROFILE: interrupts/evictions/bytes = 5962/3555/171088  
cmd=pprof –callgrind /home/wuzhu/develop/mytest/cpu\_profiler/main\_dlopen\_link2 main\_dlopen\_link2\_24515.prof >main\_dlopen\_link2\_24515.callgrind  
Using local file /home/wuzhu/develop/mytest/cpu\_profiler/main\_dlopen\_link2.  
Using local file main\_dlopen\_link2\_24515.prof.  
Removing killpg from all stack traces.  
Removing main from all stack traces.  
Removing \_\_libc\_start\_main from all stack traces.  
[wuzhu@search041142.sqa.cm4 cpu\_profiler]$

生成 main\_dlopen\_link2\_24515.prof 和 main\_dlopen\_link2\_24515.callgrind

ν 用 kcachegrind 打开查看

**(六) 参考**  
ν http://www.cnblogs.com/lenolix/archive/2010/12/13/1904868.html

ν http://gperftools.googlecode.com/svn/trunk/doc/cpuprofile.html