nginx 函数众多，指针复杂，加上大量函数指针的干扰，要仅靠阅读源代码来理清 nginx 程序的执行流程难度较大，这一节，我将提供一个获得 nginx 程序执行流程的快速方法，方便大家阅读源码。

gcc 提供了一个名为“-finstrument-functions”编译选项，也许不少人都用过，这个选项有什么作用，我这里就省略暂且不讲，不过它将是这一节的主角，阅读了下面的内容，大家会逐渐了解，如果需要马上知道的，可以 google 或查阅 man 手册。

第一：

首先，我们要把这个“-finstrument-functions”选项添加到 nginx 源码的编译过程里，这可以通过修改 Makefile 做到。大家肯定知道*nix 下编译源代码的三步骤：

./configure

make

make install

对nginx进行configure之后在nginx的源码下将生成一个objs文件夹，其中要修改的Makefile就在里面[1][2]。

修改它的CFLAGS设定，原本如下[3]：

CFLAGS = -O -pipe -O -W -Wall -Wpointer-arith -Wno-unused-parameter -Wno-unused-function -Wunused-variable -Wunused-value -Werror -g

修改成这样：

CFLAGS = -pipe -W -Wall -Wpointer-arith -Wno-unused-parameter -Wno-unused-function -Wunused-variable -Wunused-value -Werror -g -finstrument-functions

即是：

1，在末尾加上了“-finstrument-functions”选项

2，优化选项-O去掉了，原本就有两个，两个都去掉。在gcc 4.0 以后的版本[4]，这种优化选项的存在话会导致之后编译的nginx执行出现段错，如果你那也是如此请照做即可。另外，如果还有其他的优化选项比如-O1、-O2 等等都应去掉。

3，注意：应保证有-g 选项，如果没有则是因为 configure 时没带--with-debug 选项，这里直接填上-g 就可以了。

注意：

1，如果你在执行过 make 之后修改了 Makefile 的 CFLAGS 选项，要想立即把这个选项的改动更新应用到所有源文件上去，需要 make clean，再执行 configure，然后在执行 make，如果漏掉了中间的 configure 则将获得如下错误信息：

make: *** No targets specified and no makefile found. Stop.

因为 nginx 的 Makefile 文件在 make clean 的时候也一同被删除了，而执行 configure 又创建了新的 Makefile 文件，因此应注意修改 Makefile 文件的时机。

不过，这里提供一个更快速的方法，在 nginx 的 src 目录执行如下命令：

lenky@lenky-desktop:~/nginx/nginx-0.6.36/src$ find . -name "*" | xargs touch

刷新各源文件的时间戳即可。

2，直接在 make 命令里加上“-finstrument-functions”选项：

lenky@lenky-desktop:~/nginx/nginx-0.6.36$ sudo make CFLAGS+=-finstrument-functions

make -f objs/Makefile

make[1]: Entering directory `/home/lenky/nginx/nginx-0.6.36'

gcc -c -finstrument-functions -I src/core -I src/event -I src/event/modules -I src/os/unix -I objs \

-o objs/src/core/ngx_resolver.o \

src/core/ngx_resolver.c

这种方法看上去会丢失其它编译选项，其实我们只要把需要的选项同样加上也就是可以了：

lenky@lenky-desktop:~/nginx/nginx-0.6.36$ sudo make CFLAGS+="-finstrument-functions -g"

make -f objs/Makefile

make[1]: Entering directory `/home/lenky/nginx/nginx-0.6.36'

gcc -c -finstrument-functions -g -I src/core -I src/event -I src/event/modules -I src/os/unix -I objs \

-o objs/src/core/ngx_garbage_collector.o \

src/core/ngx_garbage_collector.c

总之，要看到滚动的编译输出屏幕上的每一条 gcc 命令后面都带有“-finstrument-functions”选项就表示成功完成了第一步。

3，另外，记住这个文件，等下后面还要修改它。

第二：

我这里提供两个文件，一个头文件，一个源文件，请照搬下来：

文件名：my_debug.h

文件内容：

```
#ifndef MY_DEBUG_LENKY_H
#define MY_DEBUG_LENKY_H

#include <stdio.h>

void enable_my_debug( void ) __attribute__
  ((no_instrument_function));

void disable_my_debug( void ) __attribute__
  ((no_instrument_function));

int get_my_debug_flag( void ) __attribute__
  ((no_instrument_function));
```

```
void set_my_debug_flag( int ) __attribute__
    ((no_instrument_function));

void main_constructor( void ) __attribute__
    ((no_instrument_function, constructor));

void main_destructor( void ) __attribute__
    ((no_instrument_function, destructor));

void __cyg_profile_func_enter( void *, void *) __attribute__
    ((no_instrument_function));

void __cyg_profile_func_exit( void *, void *) __attribute__
    ((no_instrument_function));

#ifndef MY_DEBUG_MAIN
    extern FILE *my_debug_fd;
#else
    FILE *my_debug_fd;
#endif

#endif
```

该头文件申明了一些函数，定义/申明了一个 FILE 类型变量。

介绍 no_instrument_function 时，将 nginx 的 log 日志函数的申明加上该属性，去掉对其调用的记录，因为这些函数的调用对理解 nginx 流程无帮助。但是好多啊？或许可以直接在进入日志前关闭记录，出来后打开日志。
关闭 log 日志功能也可以，但是这样的话就无法获得日志信息了。

利用另外两个选项：
-finstrument-functions-exclude-file-list=file,file,...
-finstrument-functions-exclude-function-list=sym,sym,...

文件名：my_debug.c

文件内容：

```
#include "my_debug.h"

#define MY_DEBUG_FILE_PATH "/usr/local/nginx/sbin/mydebug.log"

int _flag = 0;

#define open_my_debug_file()    \
    (my_debug_fd = fopen(MY_DEBUG_FILE_PATH, "a"))
```

```
#define close_my_debug_file()    \
    do{    \
        if (NULL != my_debug_fd)    \
        {    \
            fclose(my_debug_fd);    \
        }    \
    }while(0)

#define my_debug_print(args, fmt...) \
    do{    \
        if (0 == _flag) \
        {    \
            break;    \
        }    \
        if (NULL == my_debug_fd && NULL == open_my_debug_file())    \
        {    \
            printf("Err: Can not open output file.\n");    \
            break;    \
        }    \
        fprintf(my_debug_fd, args, ##fmt);    \
        fflush(my_debug_fd);    \
    }while(0)

void enable_my_debug( void )
{
    _flag = 1;
}

void disable_my_debug( void )
{
    _flag = 0;
}

int get_my_debug_flag( void )
{
    return _flag;
}

void set_my_debug_flag( int flag )
{
    _flag = flag;
}
```

```
void main_constructor( void )
{
   //Do Nothing
}

void main_destructor( void )
{
   close_my_debug_file();
}

void __cyg_profile_func_enter( void *this, void *call )
{
   my_debug_print("Enter\n%p\n%p\n", call, this);
}

void __cyg_profile_func_exit( void *this, void *call )
{
   my_debug_print("Exit\n%p\n%p\n", call, this);
}
```

该源文件实现了头文件 my_debug.h 定义的函数，它们的实现都很简单，省略待详写……

文件建立好后放在 nginx 的 core 源文件目录：

```
lenky@lenky-desktop:~/nginx/nginx-0.6.36/src/core$ ls my_debug.*
my_debug.c    my_debug.h
```

第三：

将这两个文件引入 nginx，怎么引入呢？还是修改前面提到的 Makefile 文件，修改地方比较多（其实都很简单，一看就知道的），下面一一列出：

1，核心依赖：

```
CORE_DEPS = src/core/nginx.h \
   src/core/my_debug.h    \
   …
```

2，HTTP 依赖：

```
HTTP_DEPS = src/http/ngx_http.h \
   src/core/my_debug.h    \
   …
```

3，nginx 目标依赖：

```
objs/nginx:     objs/src/core/nginx.o \
   objs/src/core/my_debug.o    \
   …
```

4，连接目标：

```
$(LINK) -o objs/nginx \
        objs/src/core/my_debug.o    \
        …
```

5，my_debug.o 规则：

```
objs/src/core/my_debug.o:    $(CORE_DEPS) \
        src/core/my_debug.c
        $(CC) -c $(CFLAGS) $(CORE_INCS) \
            -o objs/src/core/my_debug.o \
            src/core/my_debug.c
```

还要在 nginx 所有源文件都包含有头文件 my_debug.h，当然，没必要每个源文件都去添加对这个头文件的引入，我们只需要在头文件 ngx_core.h 内加入对 my_debug.h 文件的引入即可，这样其它 nginx 的源文件就间接的引入了这个文件：

```
lenky@lenky-desktop:~/nginx/nginx-0.6.36/src/core$ grep -A 3 -B 3 my_debug.h ngx_core.h
#define    NGX_ABORT          -6


#include "my_debug.h"

#include <ngx_errno.h>
#include <ngx_atomic.h>
/home/lenky/nginx/nginx-0.6.36/src/core
```

在源文件 nginx.c 的最前面加上对宏 MY_DEBUG_MAIN 的定义，以使得我们的 nginx 程序有（且仅有）一个 my_debug_fd 变量的定义：

```
lenky@lenky-desktop:~/nginx/nginx-0.6.36/src/core$ grep -A 3 -B 3 MY_DEBUG_MAIN nginx.c
 * Copyright (C) Igor Sysoev
 */

#define MY_DEBUG_MAIN 1

#include <ngx_config.h>
#include <ngx_core.h>
lenky@lenky-desktop:~/nginx/nginx-0.6.36/src/core$
```

另外，加上我们的 debug 启动代码，即在合适的位置调用函数：

```
enable_my_debug();
```

然后可以再在合适的位置调用函数：

```
disable_my_debug();
```

以便只获取我们最为关注的流程信息。这里我就在 nginx 的 main 函数入口处调用开关函数 enable_my_debug();：

lenky@lenky-desktop:~/nginx/nginx-0.6.36/src/core$ grep -A 3 -B 3 enable_my_debug nginx.c

```
        ngx_cycle_t            *cycle, init_cycle;
        ngx_core_conf_t    *ccf;

        enable_my_debug();

    #if (NGX_FREEBSD)
        ngx_debug_init();
```

好了，到这里，工作完成一大半了，make 一下 nginx 并运行它，我建议以单进程模式运行并且将 log 日志功能的记录级别设置低一点，只记录少量的日志，否则将有大量的 log 日志函数调用堆栈信息，经过这样的设置后，我们才能获得更清晰的 nginx 执行流程。

这只需在 nginx 的启动配置文件内设置：

master_process off;

error_log logs/error.log emerg;

即可。

正常运行后的 nginx 将产生一个记录程序执行流程的文件（这个文件会随着 nginx 的持续运行迅速增大，所以大家在获得想要的信息后应记得及时关闭 nginx）：

lenky@lenky-desktop:/usr/local/nginx/logs$ ls mydebug.log

mydebug.log

这个文件记录的是类似如下信息：

```
Enter
0x804a505
0x8058244
Enter
0x805829e
0x80582b3
Enter
0x80582f7
0x80586e8
Exit
0x80582f7
0x80586e8
Enter
0x80583fa
0x8058907
Exit
0x80583fa
0x8058907
```

…
…

这就是函数的调用关系，不过这里的函数还只是显示为其对应的地址而已，利用我们之前用过的 addr2line 命令，将地址转为对应的函数名不是相当容易的么？不过，一个个转也太麻烦了，好，下面再看看我提供的一个 shell 脚本：

文件名：addr2line.sh

文件内容：

```
#!/bin/sh

if [ $# != 3 ]; then
   echo 'Usage: addr2line.sh executefile addressfile functionfile'
   exit
fi;

cat $2 | while read line
   do
      if [ "$line" = 'Enter' ]; then
         read line1
         read line2
#          echo $line >> $3
         addr2line -e $1 -f $line1 -s >> $3
         echo "----->" >> $3
         addr2line -e $1 -f $line2 -s | sed 's/^/        /' >> $3
         echo >> $3
      elif [ "$line" = 'Exit' ]; then
         read line1
         read line2
         addr2line -e $1 -f $line2 -s | sed 's/^/        /' >> $3
         echo "<-----" >> $3
         addr2line -e $1 -f $line1 -s >> $3
#          echo $line >> $3
         echo >> $3
      fi;
   done
```

很明显，该 shell 脚本做的工作就是逐行读取我们的记录地址信息的文件，判断该行是否为“Enter”或者是“Exit”，如果是则紧跟着的两行表示为函数调用或退出的相关地址地址，利用 addr2line 对它进行转换。

示例：

```
lenky@lenky-desktop:/usr/local/nginx/sbin$ ls
addr2line.sh  mydebug.log  nginx  nginx.old
lenky@lenky-desktop:/usr/local/nginx/sbin$ ./addr2line.sh nginx mydebug.log myfun
lenky@lenky-desktop:/usr/local/nginx/sbin$ ls
```

addr2line.sh mydebug.log myfun nginx nginx.old

如果 mydebug.log 比较庞大，那么执行 addr2line.sh 脚本将耗费不少时间，因此也是我前面提到的应及时关闭 nginx，只获取我们需要的信息。不过就算时间比较长，这点时间还是可以等待的，因为它将大大缩减我们后面具体分析源码的时间。

你看到的 myfun 文件内容也许是这样：

```
main
nginx.c:212
----->
    ngx_time_init
    ngx_times.c:46

ngx_time_init
ngx_times.c:57
----->
    ngx_time_update
    ngx_times.c:63

ngx_time_update
ngx_times.c:69
----->
    ngx_atomic_cmp_set
    ngx_gcc_atomic_x86.h:39

    ngx_atomic_cmp_set
    ngx_gcc_atomic_x86.h:39
<-----
ngx_time_update
ngx_times.c:69

ngx_time_update
ngx_times.c:104
----->
    ngx_gmtime
    ngx_times.c:205

    ngx_gmtime
    ngx_times.c:205
<-----
ngx_time_update
ngx_times.c:104
…
…
```

不容我细说了，很容易看懂这个我们自己生成的 nginx 执行流程，比如首先 main 函数在调用（位置根据 nginx.c:212 可看到在源文件 nginx.c 的第 212 前面一句，注意这里记录的行是被调函数执行返回后执行语句所在的行，所以实际的调用函数语句是前一句）了函数 ngx_time_init（该函数定义在源文件 ngx_times.c 内，从第 46 行开始，也就是函数的入口代码行），ngx_time_init 又调用 ngx_time_update，ngx_time_update 接着调用 ngx_atomic_cmp_set，ngx_atomic_cmp_set 函数执行完后返回到 ngx_time_update，等等……

这就是我们的最终结果，应该还算不错吧？

“-finstrument-functions”选项实现的缺点：

根据我们获得的结果可以看到，当一个函数内有多处调用同一个函数时，根据可以调用函数返回执行代码所在的行号还是容易定位到是在哪儿调入的。

但是，当一个函数有多处地方可以返回时，无法根据记录的这些信息简单的看出被调函数是从哪儿返回的，因为当函数退出时记录的也是函数的入口代码行号，从上面的红/蓝一摸一样的代码可以看到这个缺点。

如何改进以在函数退出时获得函数实际执行的返回代码所在的行呢？也许需要修改GCC的这个编译选项[5]。

lenky0401@2009-12-20、草稿文档。

lenky0401@163.com

---

[1] 至于怎么知道是这个文件，这些太细节的东西就不讲了，不是我们关注的重点，以下同。

[2] 如果没权限则需chmod a+w Makefile，额，感觉太啰嗦了，下面这种类似小问题也不再叙述。

[3] 你看到的或许和我的不一样，因为configure时设定的参数不一样，不影响。

[4] 我也不肯定，但是我在gcc3 版本下未出现这种问题，据说是gcc4 的优化做了较大改动，也许与此有关，不过具体原因我也未做深入研究。关于gcc4 的优化改进，感兴趣的读者可以查阅gcc官方介绍：http://gcc.gnu.org/gcc-4.0/changes.html

[5] 修改gcc太过于复杂，不是本文的重点，在这里暂且省略，感兴趣的读者可以以关键字“gcc hack”或“gcc rtl”来搜索google，也欢迎和lenky邮件交流。