Coredump 简介及使用

DOCUMENT HISTORY

Version	Date	Author	Type of Modification
1.0	04/27/15	Vincent Song	Creation

目录

目录

一、什么是 Coredump

有些 C/C++程序或者通过 JNI 调用了 C/C++的 APK 程序可以通过编译，但在运行时会出现错误，比如常见的 signal 11 (SIGSEGV)，这样的程序都是可以通过编译的，而且这样的错误一般情况下不会像编译错误一样告诉你具体在代码的那一行有问题，所以调试起来比较困难和麻烦。

因为上面的这种 debug 困难的情况，所以就产生了 coredump 这种机制（**很多操作系统都支持这种机制，并不是 Linux 独有，也不是专门为 Android 而生**），系统（或者标准库）在发现程序错误的异常退出时，就会把程序当时整个进程的状态 dump 出来，生成一个 coredump 文件，**通常情况下 coredump 包含了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息等，可以理解为把程序工作的当前状态存储成一个文件，很幸运，我们现在使用 Android 系统是基于 Linux 内核，Linux 内核原生就支持这种机制。**

二、Coredump 产生的原因

造成程序 coredump 的原因很多，这里根据以往的经验总结一下：

1、 内存访问越界

a) 由于使用错误的下标，导致数组访问越界

b) 搜索字符串时，依靠字符串结束符来判断字符串是否结束，但是字符串没有正常的使用结束符

c) 使用 strcpy, strcat, sprintf, strcmp, strcasecmp 等字符串操作函数，将目标字符串读/写爆。应该使用 strncpy, strlcpy, strncat, strlcat, snprintf, strncmp, strncasecmp 等函数防止读写越界。

2、 多线程程序使用了线程不安全的函数

第 1 类：不保护共享变量的函数

第 2 类：保持跨越多个调用的状态函数

第 3 类：返回指向静态变量指针的函数

第 4 类：调用线程不安全函数的函数

3、 多线程读写的数据未加锁保护

一个线程做完条件判断准备使用时发生了调度，另外一个线程将其释放并置空或者修改了内容，就会导致程序执行流错乱，要么指针异常，要么状态错乱发生另外的异常

4、 非法指针

a) 使用空指针

b) 随意使用指针转换。一个指向一段内存的指针，除非确定这段内存原先就分配为某种结构或类型，或者这种结构或类型的数组，否则不要将它转换为这种结构或类型 的指针，而应该将这段内存拷贝到一个这种结构或类型中，再访问这个结构或类型。这是因为如果这段内存的开始地址不是按照这种结构或类型对齐的，那么访问它 时就很容易因为 bus error 而 core dump

c) 使用未初始化或者已经释放或者状态不明的野指针

5、 堆栈溢出

不要使用大的局部变量（因为局部变量都分配在栈上），这样容易造成堆栈溢出，

破坏系统的栈和堆结构，导致出现莫名其妙的错误

三、如何控制产生 Coredump

直接修改 init.rc（system/core/rootdir）文件，在 setrlimit 13 40 40 后面添加：

```
setrlimit 4 -1 -1
mkdir /data/coredump 0777 system system
write /proc/sys/kernel/core_pattern /data/coredump/core.%e.%p
write /proc/sys/fs/suid_dumpable 1
```

这些命令的目的是将 RLIMIT_CORE 的大小修改为 unlimited，具体参见：keywords.h 中 do_setrlimit 函数。其中 RLIMIT_CORE 的定义在 Resource.h 中。在 data 目录下创建 coredump 文件夹，并给系统赋予可读可写权限。改写 coredump 文件的存储路径。使调用了 seteuid()/setegid()的程序能够生成 coredump，默认情况下系统不会为这些进程生成 Coredump，需要设置 suid_dumpable 为 1。

尽管是做了上面这些动作，但是仍然只能为 native 程序产生 coredump，对于由 zygote fork 的 apk 进程则不会生成 coredump，因为 zygote 在做初始化的时候会忽略 Linux 的配置，使用默认不生成 coredump 的配置，system server 同样是由 zygote fork 的，所以也不会产生 coredump，所以如果出现因为 system server 的 thread crash 引起的重启则会非常头疼，需要为 system server 配置生成 coredump，如果其他 apk 进程需要产生 coredump 同样需要在开机初始化的时候为 zygote 做特殊的设置。

所谓的特殊设置就是将 zygote 的启动参数中应用 rlimit 的配置，system server 也是同样。

四、使用 Coredump 的准备

通过设置在出问题时产生了 coredump，接下来我们就要使用这个 coredump 为我们分析具体的问题。工欲善其事必先利其器，我们首先要准备好解析 coredump 的环境和工具。

1、GDB，由于我们是分析嵌入式 Linux 内核下的产生的 coredump，所以我们需要专门的交叉编译工具链中的 arm-linux-androideabi-gdb，大概位置：

prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb

2、有了 GDB 还不够，我们需要更加人性化、简单易操作的图形化工具—Eclipse+CDT 来查看 coredump

如何获得：

a、Eclipse C/C++ IDE for Luna SR2.click to download bundle package.

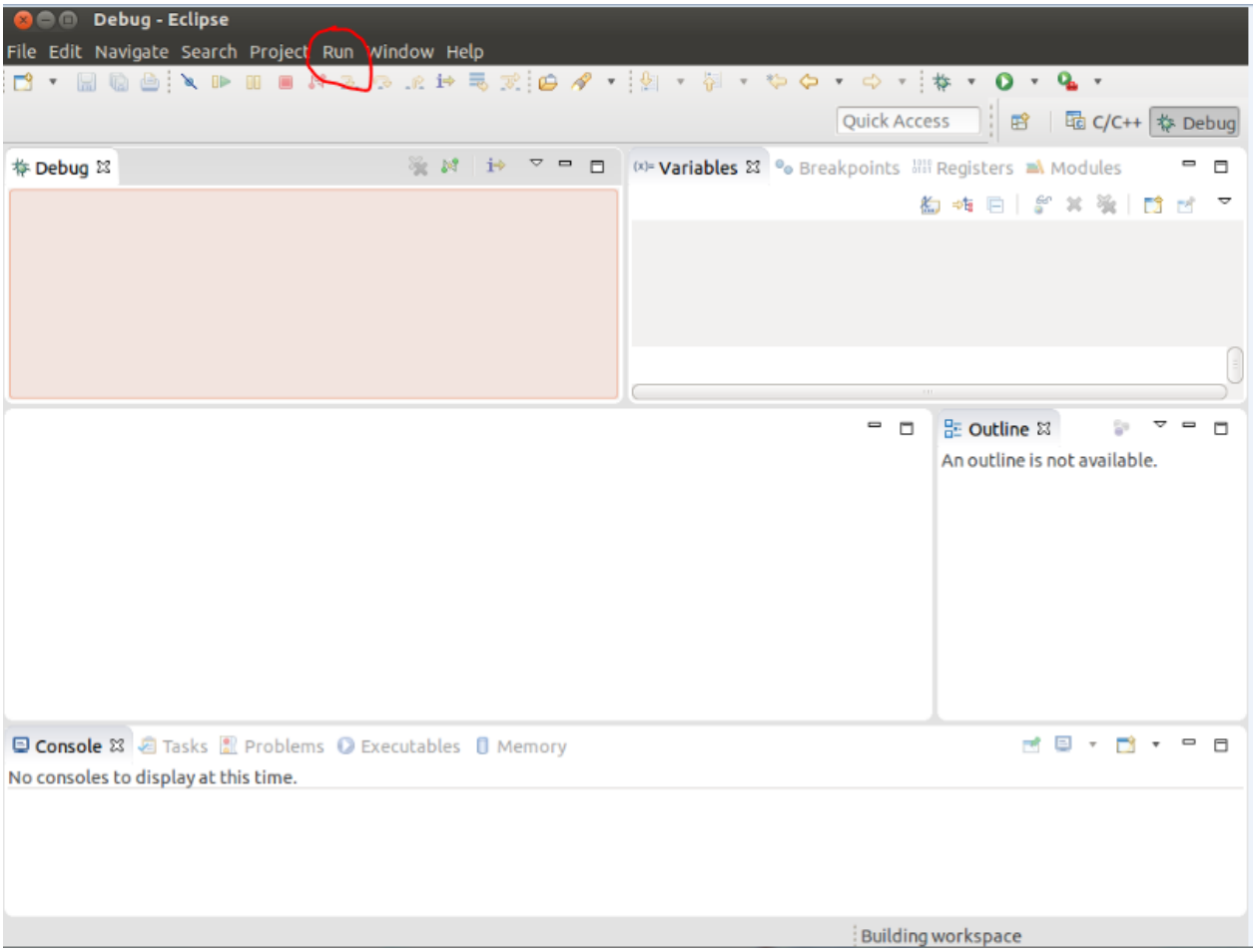
b、在 ADT 中更新你的 CDT 插件，help-》install new software-》type url：http://download.eclipse.org/tools/cdt/releases/8.6

3、准备与 coredump 版本对应的源代码环境，最好与编译版本时的路径保持一致，以便定位具体的代码

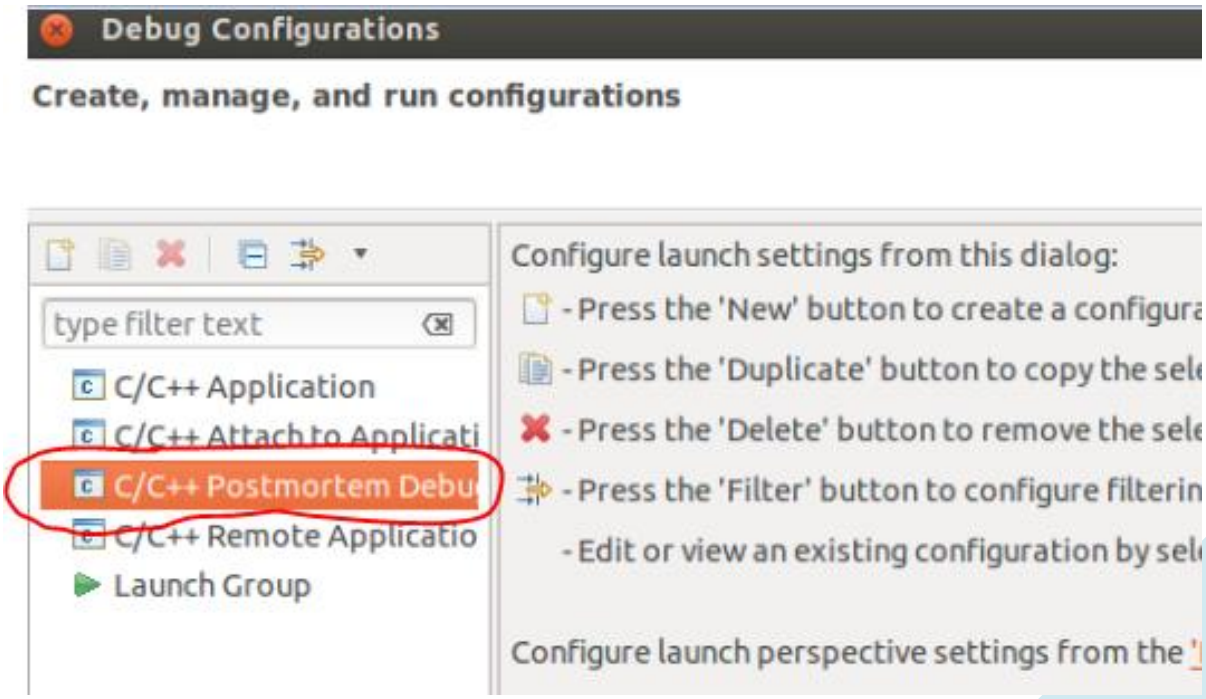
4、准备与 coredump 版本对应的 symbols

五、开始使用 Coredump

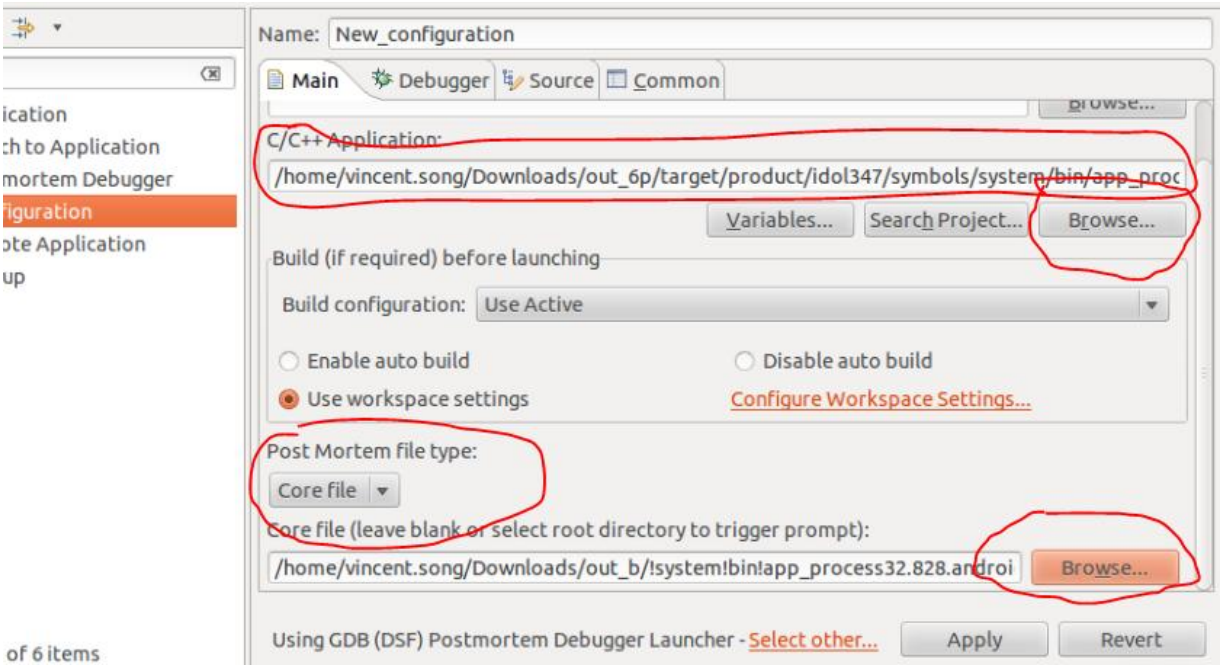
打开 Eclipse+CDT 或者 ADT，这里以 Eclipse+CDT 为例，打开后的主界面如下：



然后点击 Run-》Debug Configurations，出现以下界面：



双击 C/C++ Postmortem（验尸）Debug 出现下面的界面：



这里我们以 32 位的 5.0.2L 的 system server crash 为例，由于 system server 是由 zygote fork 出来的，而 zygote 的主入口程序是 app_process32，所以首先我们需要在 C/C++ Application 一栏中点击 Browse 找到 app_process32，我们在上面的准备工作中说了要下载好与出问题版本对应的 symbols 文件，这里就要用到，带 symbols 的 app_prcoess32 的相对路径在 out/target/product/idol347/symbols/system/bin/app_process32，选中它。

然后在 Post Mortem file type 一栏中点击 Browse，选中你出问题时生成的 coredump 文件。

上面的步骤完成之后，接下来点击 TAB 项中的 Debugger 出现以下界面：

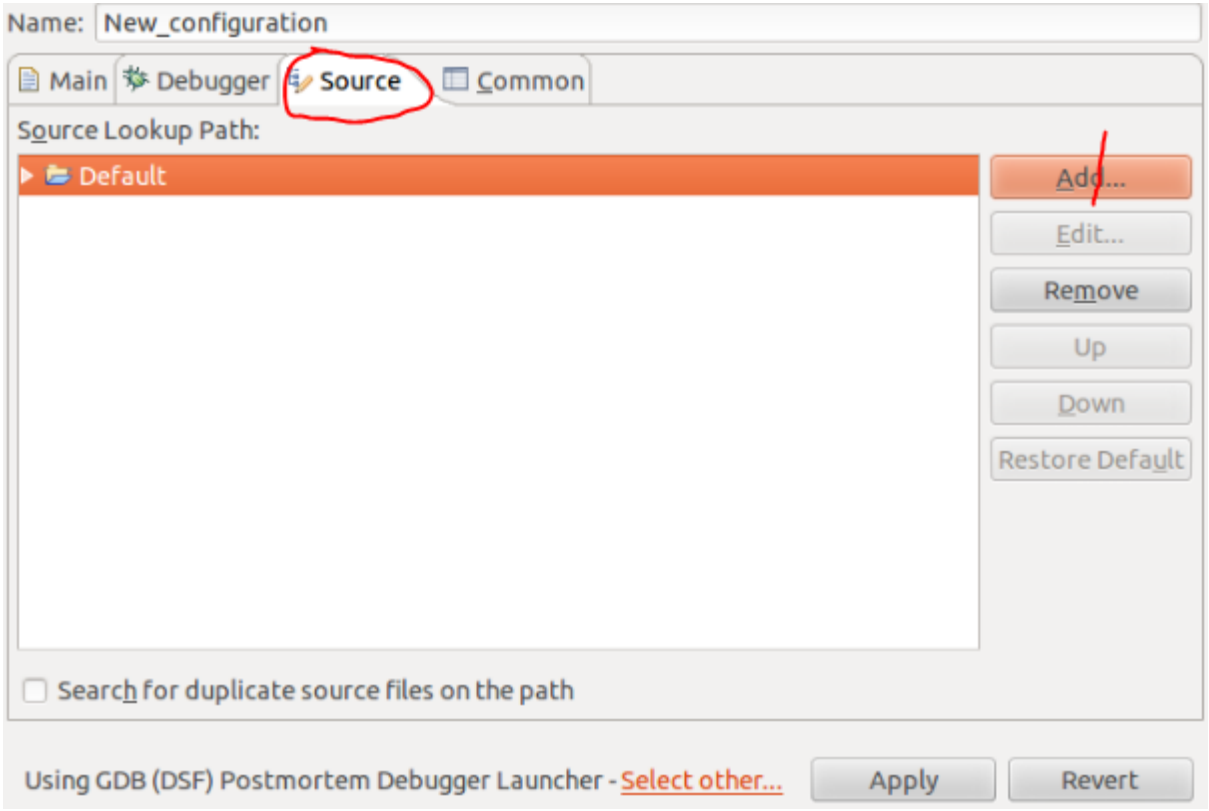
在 GDB debugger 一栏点击 Browse 找到 gdb，在源码目录中的相对路径是：prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb

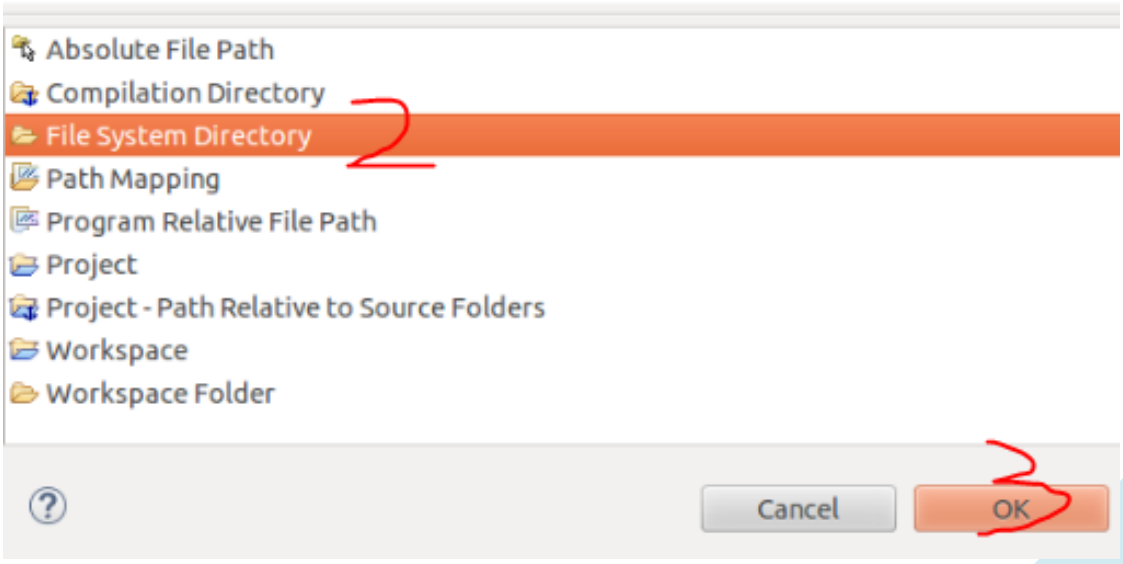
在 GDB command file 一栏点击 Browse 找到你的 gdbinit 文件，这里说明一下这个文件时在 gdb 启动时默认读取命令的文件，所以你可以在这个文件中写上你需要执行 gdb 命令，我这里写了两条供参考：

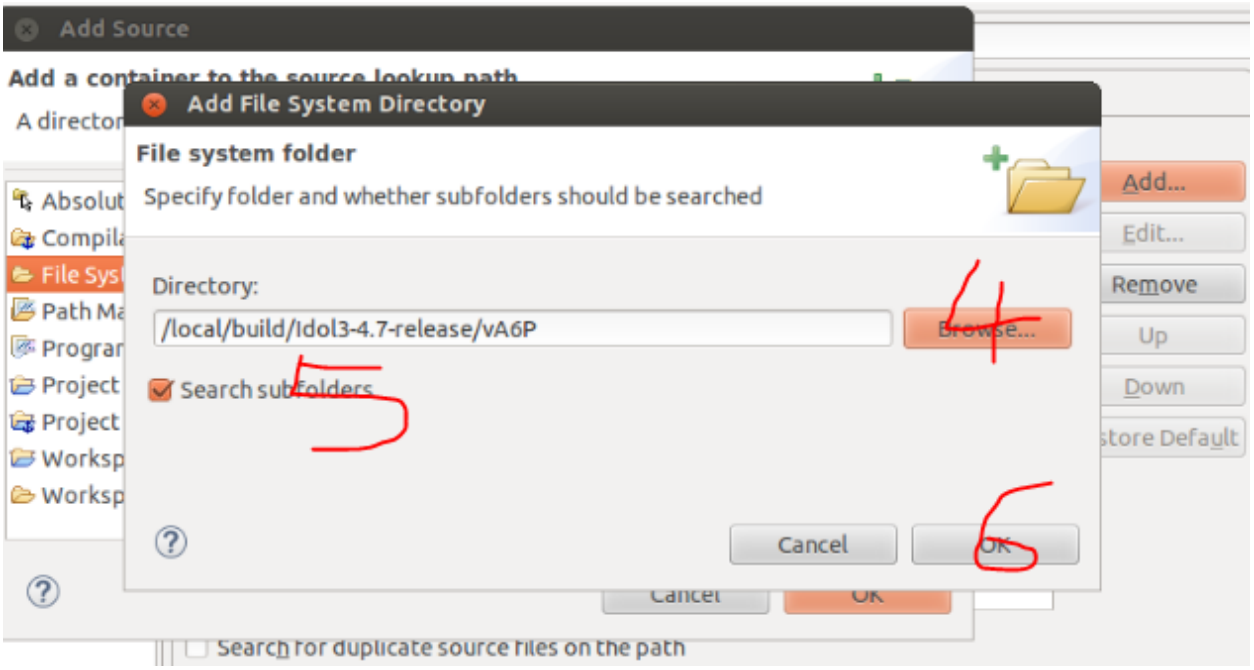
```
1  set solib-absolute-prefix /home/vincent.song/Downloads/out_6p/target/product/idol347/symbols
2  set solib-search-path /home/vincent.song/Downloads/out_6p/target/product/idol347/symbols/system/lib
```

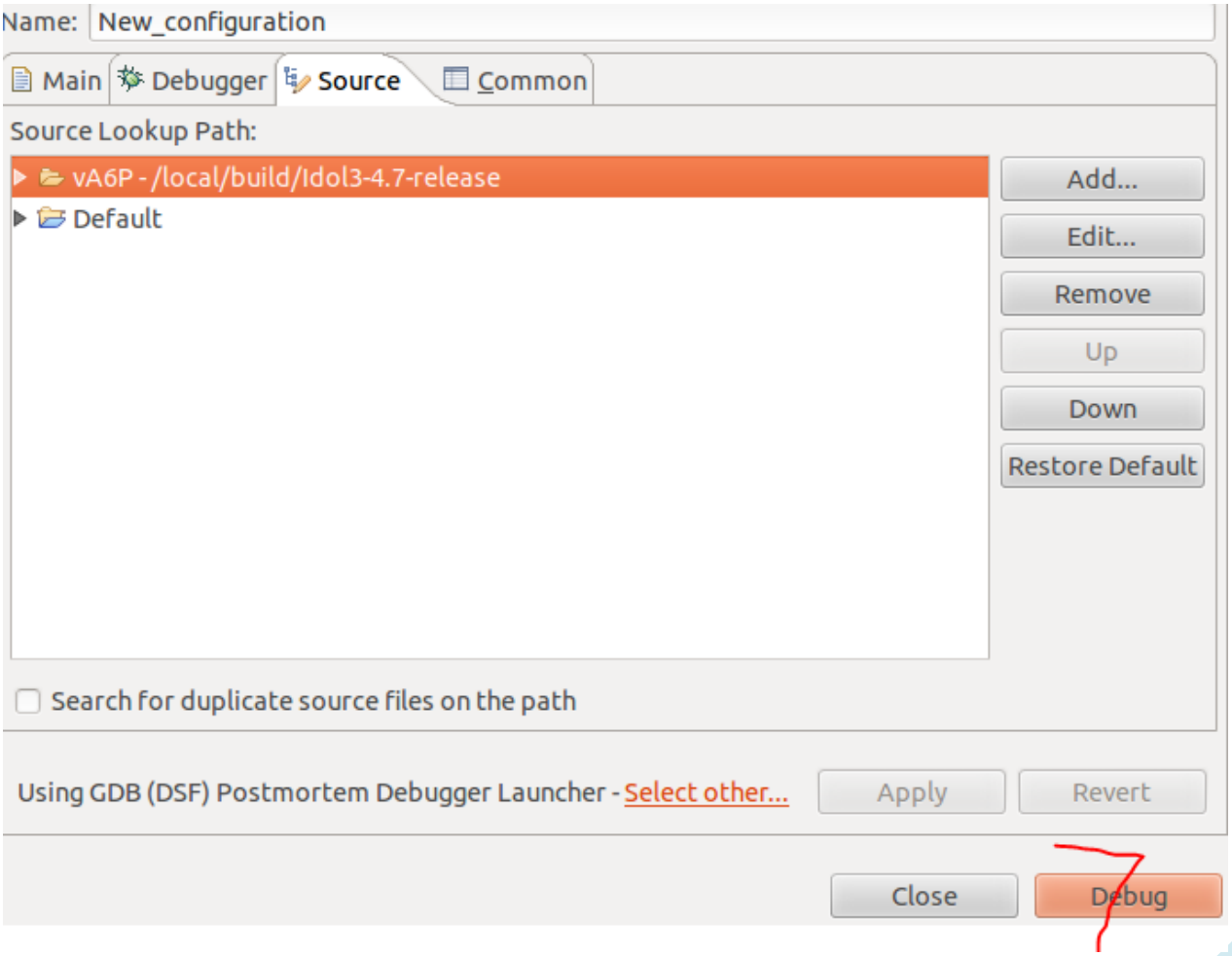
以上两条是帮助我们指定 symbols 中 so lib 的相对路径以及搜索路径，你还可以设置其他命令，具体请参考 gdb debuge 手册。

最后再点击 TAB 项中的 Source 选择源码的路径，点击后出现以下界面：

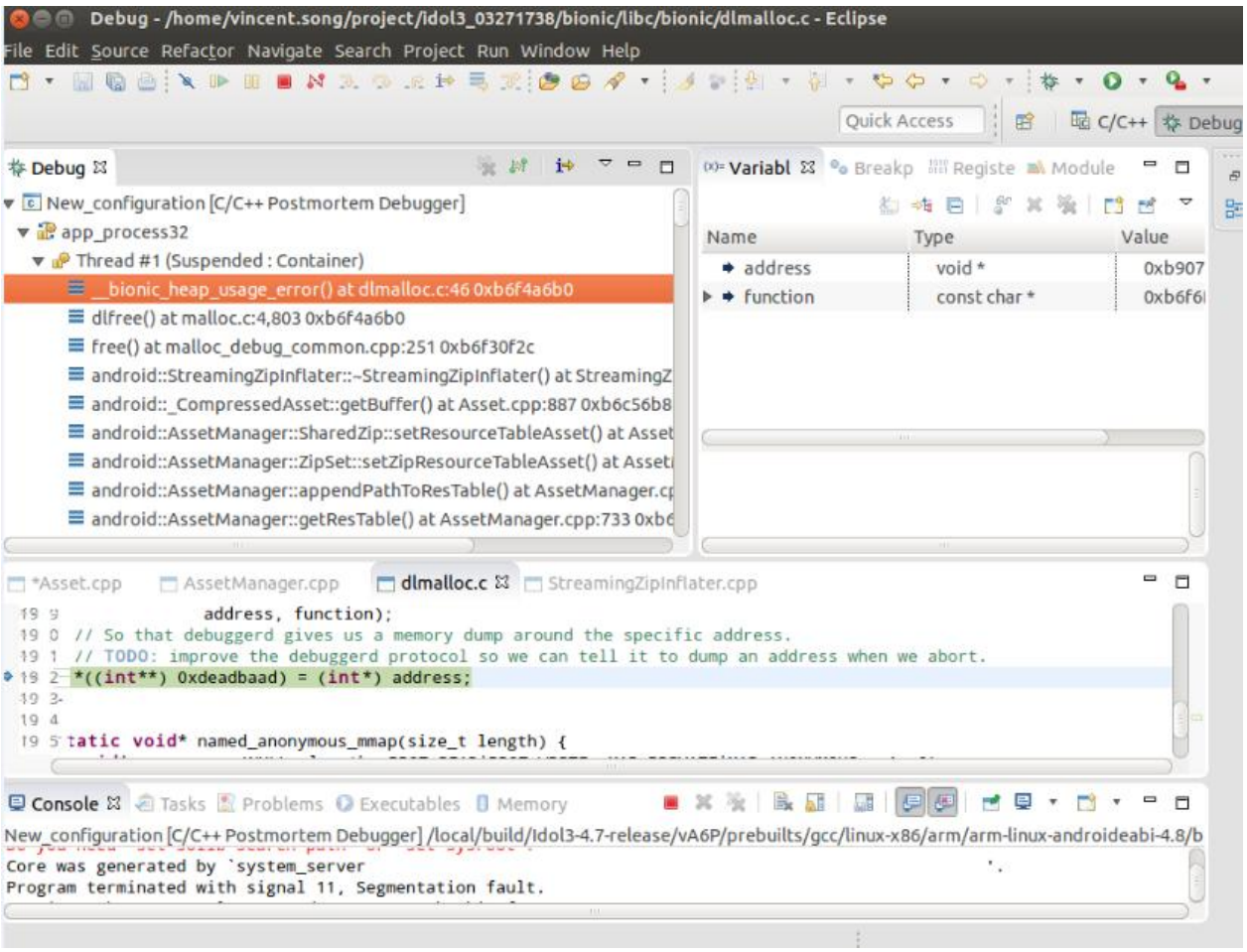








按照上面的步骤设置好之后，并在第 7 步点击 Debug 按钮之后，就会开始加载，最后进入调试的主界面：



我们可以看到主界面的布局，包括 Debug 的 callstack，variable，source cdoe，console 等，其中 Debug 界面中显示的 callstack 就是我们这个 coredump 对应的程序最后 crash 的 callstack，我们可以逐层往上追溯，点击对应的调用项就可以显示对应的源码，同时将鼠标放在对应的对象或者变量上就会显示出当时程序运行状态下所对应的值，这样我们就可以重新还原程序出问题时的场景和上下文，方便我们找到问题所在。

以上是一个基本使用的介绍，其他高级功能大家可以在以后的使用中不断探索，如果发现也请共享给大家，谢谢！

END OF DOCUMENT