软件开发中常用调试技巧的原理和实践

调试是软件开发过程中必不可少的环节，对于嵌入式开发者来讲很多工作都是体现在调试上，有句话讲“程序不是写出来的，好程序是调出来的”一点不夸张。纵观来看调试可以分为硬件断点调试和软件断点调试。硬件调试需要CPU的支持，CPU内部提供了两组寄存器用来存储设置的断点，但是这种场景决定于内部硬件设计与调试器和其他调试工具无关。本chat主要讲软件层面的调试，也是开发者工作中最常用到的调试方法。
谈到软件，无疑其决定因素是操作系统，这里我们以linux操作系统为例对内核态，用户态以及其调试工具和调试方法进行剖析。

内核态

先介绍下单步调试的方式，相信大部分工程师常用单步调试方式都会选择GDB，GDB是最常用的调试工具，信号是GDB实现断点功能的基础，用GDB向某个地址打断点的时候，实际上就是向改地址写入断点指令。当程序运行到这条指令后就会触发SIGTRAP信号，这时候GDB就会收到这个信号，根据程序停止在的位置查询断点表，要是发现在断点表里有此地址，则判定找到断点。下面我们先讲解下如何用GDB单步调试内核。

内核调试的配置

用GDB调试内核的话首先需要内核支持KGDB，配置如下：

$make menuconfig

```
Kernel hacking--->
        -*- Magic SysRq key
        [*] Kernel debugging
        [*] Compile the kernel with debug info
        [*] KGDB: kernel debugging with remote gdb --->
                <*> KGDB: use kgdb over the serial console
```

配置成功后编译内核，然后修改uboot启动参数以支持KGDB调试，

setenv bootargs ‘console=ttyS0,115200n8 kgdboc=ttyS0,115200 kgdbwait ……
nfsroot=……’

由uboot启动，等到KGDB的位置时停下等待GDB的连接，如图：

```
Serial: 8250/16550 driver, 4 ports, IRQ sharing enabled
serial8250.0: ttyS0 at MMIO 0x4806a000 (irq = 72) is a ST16654
console [ttyS0] enabled  #by guoingbo
serial8250.1: ttyS1 at MMIO 0x4806c000 (irq = 73) is a ST16654
serial8250.2: ttyS2 at MMIO 0x49020000 (irq = 74) is a ST16654
kgdb: Registered I/O driver kgdboc.
kgdb: Waiting for connection from remote gdb...
```

然后用GDB调试vmlinux（注意此时是交叉编译器的GDB）

$arm-none-linux-gnueabi-gdb vmlinux

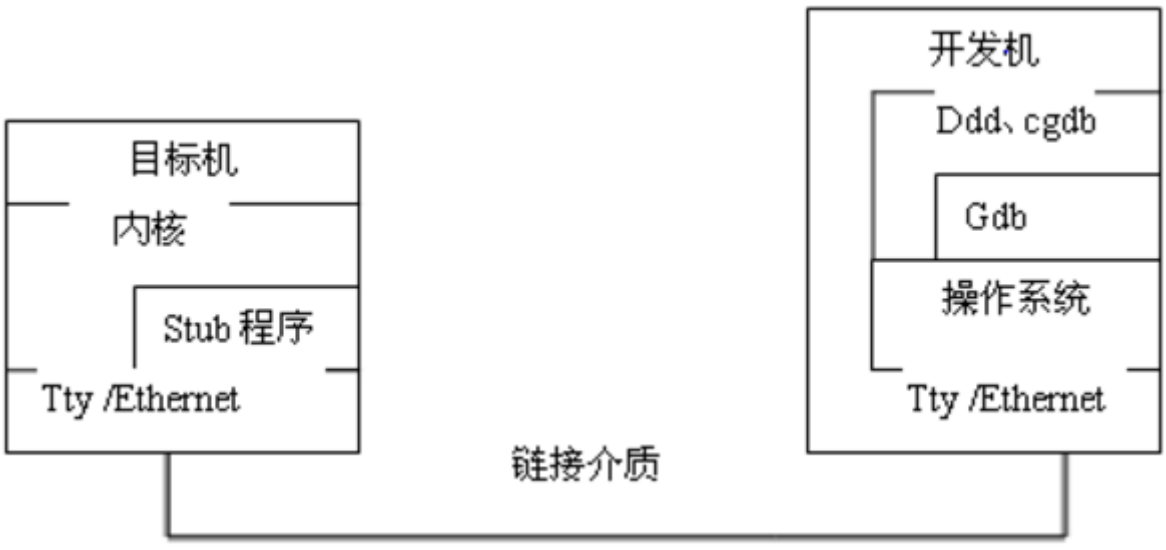
顺利的话显示如下：

```
During symbol reading, DW_AT_name missing from DW_TAG_base_type.
During symbol reading, unsupported tag: 'DW_TAG_const_type'.
During symbol reading, Child DIE 0x408b33 and its abstract origin 0x4085ee have different parents
During symbol reading, DW_AT_type missing from DW_TAG_subrange_type.
kgdb_breakpoint () at kernel/kgdb.c:1744
1744            arch_kgdb_breakpoint();
(gdb)
```

接下来就和调试应用程序一样调试内核了。

KGDB的调试原理

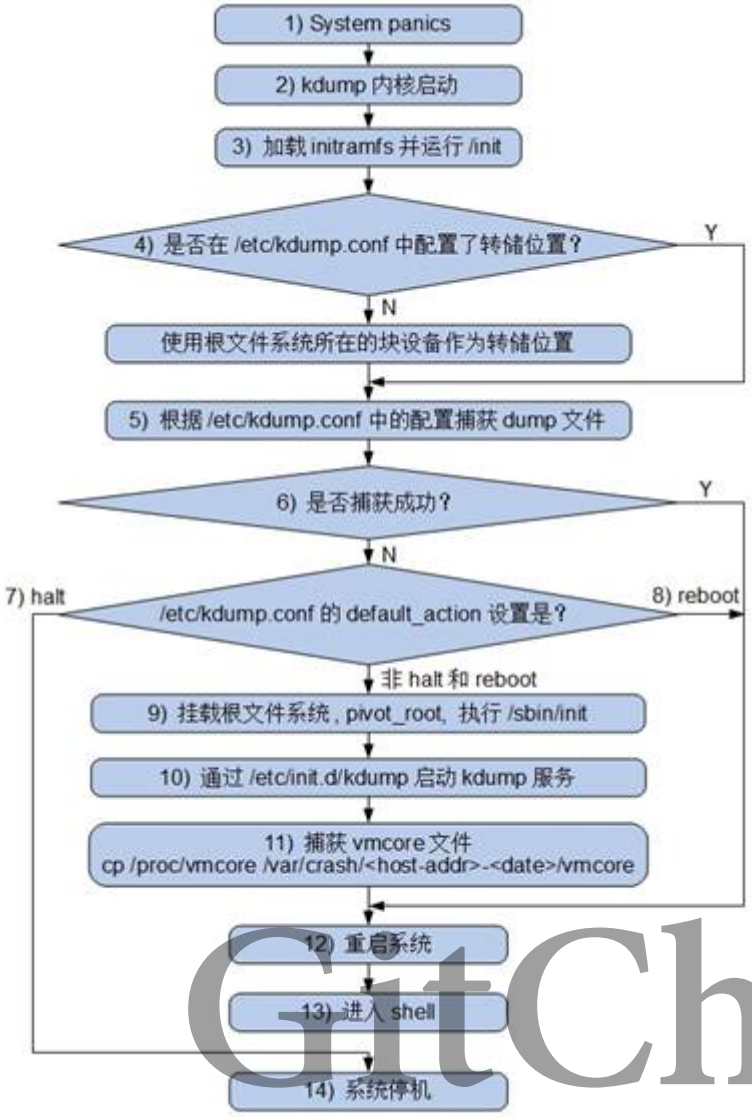
kgdb补丁的主要作用是在Linux内核中添加了一个调试Stub。调试Stub是Linux内核中的一小段代码，提供了运行gdb的开发机和所调试内核之间的一个媒介。gdb和调试stub之间通过gdb串行协议进行通讯。gdb串行协议是一种基于消息的ASCII码协议，包含了各种调试命令。当设置断点时，kgdb负责在设置断点的指令前增加一条trap指令，当执行到断点时控制权就转移到调试stub中去。此时，调试stub的任务就是使用远程串行通信协议将当前环境传送给gdb，然后从gdb处接受命令。gdb命令告诉stub下一步该做什么，当stub收到继续执行的命令时，将恢复程序的运行环境，把对CPU的控制权重新交还给内核。



使用kgdb作为内核调试环境最大的不足在于对kgdb硬件环境的要求较高，必须使用两台计算机分别作为target和development机。尽管使用虚拟机的方法可以只用一台PC即能搭建调试环境，但是对系统其他方面的性能也提出了一定的要求，同时也增加了搭建调试环境时复杂程度。另外，kgdb内核的编译、配置也比较复杂，需要一定的技巧。当调试过程结束后时，还需要重新制作所要发布的内核。使用kgdb并不能进行全程调试，也就是说kgdb并不能用于调试系统一开始的初始化引导过程。

Kdump的调试方式

除了单步调试外还有一种场景是我们经常碰到的，在系统运行中linux内核发生崩溃的时候。这时候可以通过kdump方式收集内核崩溃前的内存，生成vmcore文件，通过vmcore文件诊断出内核崩溃的原因。其中kexec是实现 kdump 机制的关键，它包括 2 个组成部分：一是内核空间的系统调用kexec_load，负责在生产内核（ production kernel 或 first kernel ）启动时将捕获内核（ capture kernel 或 sencond kernel ）加载到指定地址。二是用户空间的工具kexec-tools，他将捕获内核的地址传递给生产内核，从而在系统崩溃的时候能够找到捕获内核的地址并运行。没有kexec 就没有 kdump。先有 kexec实现了在一个内核中可以启动另一个内核，才让 kdump有了用武之地。下面附一张kdump的实现原理，感兴趣的小伙伴可以进一步研究，这部分不是本chat重点，下面让我们具体讲下如何使用kdump。



经常用到分析vmcore的工具就是crash，掌握crash的技巧对于定位问题有着很重要的意义。配置kdump完成后当系统崩溃的时候在/var/crash/当天日期/生成一个vmcore文件，下面就可以对vmcore文件进行分析。

> crash vmlinuz-4.2.0-27-generic vmcore
> 进入crash
> crash>
> 进入crash环境后就可以运用crash命令进行调试，比如bt,dis,struct等。

用户态

GDB的使用

按照上面讲解内核的顺序我们先讲解如何在用户态用GDB工具调试。其实在用户态有两种方式可以进入GDB，一种是直接在命令上输入GDB，然后再在GDB中用file命令加载要调试的程序，另外一种是直接在命令行上使用GDB程序名。

```
liupp@ubuntu:~$ gdb a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...(no debugging symbols found)...done.
(gdb)
```

GDB的常用命令

在讲gdb命令之前我们先来了解下gdb的基础知识：

- 调试器指示的是将要执行的代码行。
- 只有在编译时拥有调试符号（gcc -g）的程序才能在调试时看到源代码。
- 同一行上有多个断点时，GDB仅中断在断点号最小的那个断点上。
- 断点可以设置在同一程序的不同文件中。
- 在任何给定时间，GDB只有一个焦点，即当前执行的文件。
- 源文件改变后，断点发生移动，但是断点属性的行号不变。

GDB常用命令如下：

- 进入GDB #gdb test

```
liupp@ubuntu:~$ gdb test
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...(no debugging symbols found)...done.
(gdb)
```

- 设置断点

```
(gdb) b 2
Breakpoint 1 at 0x80483f6: file test.c, line 2.
```

- 查看断点处情况 (gdb) info b

```
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483f6 in main at test.c:2
```

- 运行代码 (gdb) r

```
(gdb) r
Starting program: /home/liupp/test

Breakpoint 1, main (argc=1, argv=0xbffff184) at test.c:7
7           int a=0, b=0, c=0;
```

- 单步运行 (gdb) n

```
(gdb) n
9           for (i=0; i<100; i++)
```

- 程序继续运行 (gdb) c

  使程序继续往下运行，直到再次遇到断点或程序结束。

- (gdb) bt

  用来打印栈帧指针，也可以在该命令后加上要打印的栈帧指针的个数，查看程序执行到此时经过哪些函数呼叫的程序，程序调用堆栈是当前函数之前的所有已调用函数的列表。每个函数及其变量都被分配了一个帧，最近调用的函数在0号帧中。

- (gdb) frame 1 用于打印指定栈帧。

- (gdb) info reg 查看寄存器使用情况。

- (gdb) info stack 查看堆栈使用情况。

- 退出GDB (gdb) q。

```
(gdb) q
liupp@ubuntu:~$
```

接下来讲下当程序运行过程中崩溃时发生的情况。当程序异常或者崩溃时会发送信号使其生成core dump文件，如程序调用abort()函数，访存错误，非法指令等。举个例子：

```c
#include <stdio.h>

int main()
{
    int *ptr = NULL;
    *ptr = 10;

    return 0;
}
```

编译后运行会发生段错误，最后会在当前目录下生成core文件，如果没有生成，运行ulimit -c unlimited命令开启core dump功能。

最后就可以安心地使用GDB调试core文件了：

```
liupp@ubuntu:~$ gdb a.out core
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.

warning: exec file is newer than core file.
[New LWP 3194]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x080483fd in main () at test.c:6
6           *ptr = 10;
(gdb)
```


怎么样，是不是很清楚的显示第六行出现了段错误。

讲到这里小伙伴们是不是以后遇到系统突然崩溃的场景有自信去分析解决了，哈哈。不过有有心的读者会发现我们之前所有的分析是建立在core文件的基础上啊，万一有时候配置不给力，linux系统会直接把进程dump掉但没有了core文件，还有种情况，刚好磁盘空间不足了，core文件也就保存不下来了，这时候我们该怎么分析呢？

GDB的多线程调试

在实际的开发项目中基本都是多线程的信息交互开发，同理在开发中遇到的bug也大都是多线程引起的，比如由于多线程之前对临界区的竞争导致死锁等。像这种问题在实际项目运行中“偶尔”出现的问题是很难调试的，这时候GDB在多线程中的使用就显的尤为重要。先写个多线程的例子：

```cpp
#include<iostream>
#include<pthread.h>
#include<string>
#include <unistd.h>
using namespace std;
void print(string words)
{
        std::cout << words << std::endl;
}
void* threadPrintHello(void* arg)
{
    while(1)
    {
        sleep(5);
        print(string("Hello"));
    }
}
void* threadPrintWorld(void* arg)
{
    while(1)
    {
        sleep(5);
        print(string("World"));
    }
}
int main( int argc , char* argv[])
{
    pthread_t pid_hello , pid_world;
    int ret = 0;
    ret = pthread_create(&pid_hello , NULL , threadPrintHello , NULL);
    if( ret != 0 )
    {
        std::cout << "Create threadHello error" << std::endl;
        return -1;
    }
    ret = pthread_create(&pid_world , NULL , threadPrintWorld , NULL);
    if( ret != 0 )
    {
        std::cout << "Create threadWorld error" << std::endl;
        return -1;
    }
    while(1)
    {
        sleep(10);
        std::cout << "In main thread"  << std::endl;
    }
    pthread_join(pid_hello , NULL);
    pthread_join(pid_world , NULL);
    return 0;
}
```


运行thread进程后找到该进程的进程号。

```
liupp@ubuntu:~$ ./thread
```

```
liupp@ubuntu:~$ ps aux | grep thread
root         2  0.0  0.0      0     0 ?        S    01:13   0:00 [kthreadd]
liupp     3358  0.0  0.1  22056  1144 pts/20   Sl+  04:48   0:00 ./thread
liupp     3366  0.0  0.1   4692  2036 pts/0    S+   04:49   0:00 grep --color=auto thread
```

Attach该进程。

```
liupp@ubuntu:~$ sudo gdb thread 3358
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from thread...done.
Attaching to program: /home/liupp/thread, process 3358
Reading symbols from /lib/i386-linux-gnu/libpthread.so.0...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/libpt
hread-2.19.so...done.
done.
[New LWP 3360]
[New LWP 3359]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Loaded symbols for /lib/i386-linux-gnu/libpthread.so.0
Reading symbols from /usr/lib/i386-linux-gnu/libstdc++.so.6...(no debugging symbols found)...done.
Loaded symbols for /usr/lib/i386-linux-gnu/libstdc++.so.6
Reading symbols from /lib/i386-linux-gnu/libgcc_s.so.1...(no debugging symbols found)...done.
Loaded symbols for /lib/i386-linux-gnu/libgcc_s.so.1
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/libc-2.19.s
o...done.
done.
Loaded symbols for /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/ld-linux.so.2...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/ld-2.19.so...done.
done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/i386-linux-gnu/libm.so.6...Reading symbols from /usr/lib/debug//lib/i386-linux-gnu/libm-2.19.s
o...done.
done.
Loaded symbols for /lib/i386-linux-gnu/libm.so.6
0xb7773cb0 in ?? ()
(gdb)
```

显示线程信息。

```
(gdb) info threads
  Id   Target Id         Frame
  3    Thread 0xb7441b40 (LWP 3359) "thread" 0xb7773cb0 in ?? ()
  2    Thread 0xb6c40b40 (LWP 3360) "thread" 0xb7773cb0 in ?? ()
* 1    Thread 0xb7443700 (LWP 3358) "thread" 0xb7773cb0 in ?? ()
(gdb)
```


切换线程。

```
(gdb) thread 2
[Switching to thread 2 (Thread 0xb6c40b40 (LWP 3360))]
#0  0xb7773cb0 in ?? ()
```

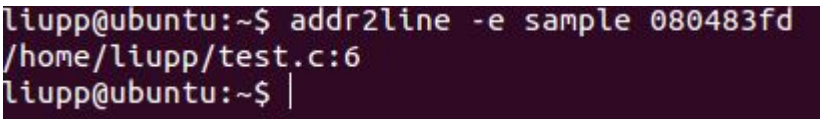
addr2line的使用

addr2line登场，addr2line是可以将指令的地址和可执行映像转换成文件名，函数名和源代码行数的工具。

我们以上面的程序为例，编译gcc -g -o sample sample.c（注意加-g是为了增加调试信息），运行执行文件sample后会发现段错误，然后通过dmesg命令发现具体错误信息如下：

```
[13719.323899] sample[2856]: segfault at 0 ip 080483fd sp bf91afa8 error 6 in sample[8048000+1000]
```

ip字段后面的数字就是sample出错时执行的位置，用addr2line就可以将位置080483fd转换成出错程序的位置：



可以清楚的看出是第六行*ptr = 10出错了。怎么样，是不是很轻松的分析出引起段错误的位置了。

GDB与addr2line的区别

GDB与addr2line都可以进行crash的分析，根据不同的场景采取不同的方法。建议如果有core文件使用GDB调试比较方便(也能实现地址转化为函数名的功能)。如果没有core文件，只有一份crash的log，那就只能使用addr2line进行分析。

总结

本chat以linux系统为例介绍了编译器gcc，调试器gdb以及堆栈的调试方法和一些常用工具和基本命令，包括run,info,bt,list等。在面对工作中常碰到的调试技术掌握这些基本可以应付了，如果你想获取更多的调试技巧请参考官方提供的GDB调试手册。

参考资料

https://www.ibm.com/developerworks/library/l-gdb/index.html?S_TACT=105AGX52&S_CMP=cn-a-l

http://www.yolinux.com/TUTORIALS/GDB-Commands.html

https://www.gnu.org/manual/gdb-4.17/gdb.html