# GDB

所谓的代码调试就是进行代码级别的问题跟踪定位。

可以分为以下几种：

（1）执行期待的行为并结束；

（2）不执行期待的行为并结束；

（3）不结束。

GDB是一个由GNU开源组织发布的、UNIX/LINUX操作系统下的、基于命令行的、功能强大的程序调试工具，对于快速发现定位代码的问题有着非常大的作用。

# 前提条件

若想执行gdb调试，在Makefile文件中需要增加编译调试选项-g，例如：

gdb dup\_file.c –o dum\_file\_elf –g –lpthread

-g选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行。但不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证gdb能找到源文件。

# 应用场景

在实际应用中，一般有两种调试方法：在线调试和离线调试。

离线调试适用于开发测试环境，可以自由启停进程，设置断点；在线调试一般用于现场问题分析，不能随便启停进程，否则会带来很多问题。

值得一提的是，上述可执行文件（即Linux的ELF文件）需要在编译时加上-g参数才可以进行调试，否则gdb是没有输出信息的。

# 使用方法

大致的调试流程：gdb –p pid 🡪 b set\_break 🡪 c/r 🡪 达到断点 🡪 set进入函数内部 🡪 n 🡪 p

## 启动进程

开发测试和现场应用环境中进行代码调试稍有不同，一般有如下几种方式：

1） 在开发中可以将源码和可执行文件拷贝到某一目录下，使用gdb启动进程进行调试；2）可以不拷贝源码和可执行文件，使用NFS挂载到编译环境执行调试；

3）在现场环境中使用ps获取进程的pid，然后gdb –p pid执行在线调试。

**具体指令：**

### 前提条件

#### 编译

对于C程序来说，需要在编译时加上-g参数，保留调试信息，否则不能使用GDB进行调试。

#### readelf查看段信息

例如：

1$ readelf -S helloWorld|grep debug

2 [28] .debug\_aranges PROGBITS 0000000000000000 0000106d

3 [29] .debug\_info PROGBITS 0000000000000000 0000109d

4 [30] .debug\_abbrev PROGBITS 0000000000000000 0000115b

5 [31] .debug\_line PROGBITS 0000000000000000 000011b9

6 [32] .debug\_str PROGBITS 0000000000000000 000011fc

helloWorld为文件名，如果没有任何debug信息，则不能被调试。

#### file查看strip状况

下面的情况也是不可调试的：

1 file helloWorld

2 helloWorld: (省略前面内容) stripped

如果最后是stripped，则说明该文件的符号表信息和调试信息已被去除，不能使用gdb调试。但是not stripped的情况并不能说明能够被调试。

### 离线调试

离线调试（无参数）：gdb 进程名

gdb –tui 进程名

说明：

使用gdb 进程名的方式，在启动gdb之后直接载入了app的可执行程序，需要注意的是，载入的app程序必须在编译的时候有gdb调试信息，例如gcc –g app app.c，如果修改了程序的源代码但是没有编译，那么在gdb中显示的会是改动后的源代码，但是运行的是改动前的程序，这样会导致跟踪错乱。

使用-tui参数可以将调试窗口分为两部分：上面是源码，下面是调试信息，使用Ctrl+n/Ctrl+p或者方向键进行翻页。

离线调试（带参数）：gdb 可执行文件🡪run 参数

gdb 可执行文件🡪set args 参数🡪run

### 在线调试

#### gdb -p

在线调试：ps –A | grep 进程名

gdb <program> <PID>

gdb –p pid/gdb attach pid

说明：<program>是可执行文件名，<PID>是要调试程序的PID，如果你的程序是一个服务程序，那么可以指定这个服务程序运行时的进程ID。Gdb会自动attach上去，并调试。Program应该在PATH环境变量中搜索的到。但是此时不能查看源代码，需要使用file命令查看源代码的位置。

#### file

(gdb)file app

说明：载入指定的程序，这样在gdb中载入想要调试的可执行程序app。如果刚开始运行gdb而不是用gdb app启动的话就可以这样载入app程序，当然编译的时候需要加上-g调试选项。

#### attach

要调试守护进程等已经启动的进程，或者调试陷入死循环而无法返回控制台的进程时，可以使用attach命令。

gdb🡪attach pid

说明：针对已经运行的进程，gdb🡪sttach需要调试进程的pid，与gdb –p pid类似。

程序等待的原因有很多，可能有调用输入输出或者系统调用时发生**阻塞**，或**等待获得锁**等。死循环，就是持续邓艾某条件变为真，其原因从自旋锁（spin lock）到单纯的逻辑错误（指定了绝不可能变成真的条件而导致的bug）有很多种。

确定了行为之后，需要在gdb和进程分离时使用detach命令，这样调试中的进程就被从gdb的控制下释放出来，进程被detach后会继续运行。

### 带参调试

1、启动的时候带上参数

gdb --args xxx 参数

2、启动之后 run 带上参数

# gdb xxx

(gdb)run 参数

3、启动之后 set args 设置参数

# gdb xxx

(gdb) set args 参数

### core文件调试

当程序core dump时，可能会产生core文件，它能够很大程序帮助我们定位问题。但前提是系统没有限制core文件的产生。可以使用命令limit -c查看：

1$ ulimit -c

如果结果是0，即便程序core dump了也不会有core文件留下。我们需要让core文件能够产生：

1 ulimit -c unlimied #表示不限制core文件大小

2 ulimit -c 10 #设置最大大小，单位为块，一块默认为512字节

上面两种方式可选其一。第一种无限制，第二种指定最大产生的大小。

针对生成core文件进行调试，可以采用在线加载和离线加载的方式，如下：

gdb 可执行文件 core文件

gdb –c core文件—>执行可执行文件

## 断点操作

### 设置断点

打断点还是比较有技巧的，虽然有很多打断点的方法，但是实际调试中一般就使用以下几种：

函数打断点：b/break 函数名

b/break 源文件：函数名

某一行打断点：b/break 源文件：行号（这种情况下源码可能与ELF对应的行号有一点差别）

说明：对于有些操作打断点后运行是无法执行到断点的，需要界面出发某些操作才可以。

### 条件断点

有一种断点仅在特定条件下中断。其格式为：

break 断点 if 条件

continue 断点编号

continue 断点编号 条件

该命令可以给指定的断点添加或删除触发条件，第1个格式删除执行编号断点的触发条件，第2个格式给断点添加触发条件。

### 查看断点

**查看断点：**info breakpoint/info break/info b

说明：这里会显示所有的断点、断点号、断点位置。

### 删除断点

**删除断点：**delete 断点号

delete（删除所有的断点）

clean N（清除行N上的所有断点）

### 禁用/开启断点

禁用/开启断点：disable/enable breakpoint

## 执行程序

执行程序的方法有两种：一种是从main函数开始执行逐步分析，一种是执行到断点处。

具体指令为：

### 重新运行

重新运行：r/run

说明：要想运行准备调试的程序，可使用run命令，在它后面可以跟随发给该程序的任何参数，包括标准输入和标准输出说明符（< 和 >）和shell通配符（\*、？、[、]）。

### 继续运行

继续执行：c/continue

### 单步

单步执行：n/next/next N（执行N次下一步）

单步进入：step（遇到函数进入函数内部，退出函数时使用finish）

说明：这样也会执行一行代码，但是与next不同的是如果遇到函数，则会进入函数内部，再逐行执行，可以使用finish快速执行函数到尾部。

### 结束

结束函数：finish

说明：这样运行程序，知道当前函数运行完毕返回再停止。例如进入的单步执行如果已经进入某函数，而向退出该函数返回到它的调用函数中，可使用命令finish。

### 返回

强制返回：return（忽略当前未执行的部分，强制返回）

说明：如果调试时使用r/run🡪n/next，此时进程会重新执行，gdb –p pid的PID就不是原来的PID了，会生成一个新的PID，应该使用c/continue🡪n/next。

这样将会忽略当前函数还没有执行完毕的语句，强行返回。return后面可以接一个表达式，表达式的返回值就是函数的返回值。

### until

（gdb）until/u

说明：指定程序直到退出当前循环体。

这里发现需要把光标停止在循环的头部，然后输入n这样就自动执行全部的循环了。

### 跳转

（gbd）jump N

说明：这里表示跳转到第N行执行完毕之后，如果后面没有断点则继续执行，而并不是停在那里。

另外，跳转不会改变当前的堆栈内容，所以跳到别的函数中就会有奇怪的现象。因此最好跳转在一个函数内部进行，跳转的参数也可以是程序代码行的地址、函数名等。

## 显示堆栈

（gbd）backstrace/bt

有时候跳转的次数太多，不知道具体调用的层级关系了，可以使用bt查看堆栈，该命令会产生一张列表，包含着运行过程和相关的参数。

该命令会产生一张列表，包含着从最近的过程开始的所有有效过程和调用这个过程的参数。当然，这里也会显示当前运行到了哪里（文件/行）。

## 显示设置

有的时候在进行代码调试的时候，会出现变量内容省略的情况，如果需要完整打印较长字符串的信息，可以采用如下设置：

set print element 0

如果需要更改打印的格式，使用便于人阅读的格式，可以做如下设置：

set print pretty on

对于存在多种继承关系的变量，如果想要显示各个类变量的实际子类，可以：

set p pre

set p obj

## 变量操作

### 设置变量

在调试的时候，有时候需要设置一些假数据查看对应输出，比如根据布尔值查看流程执行情况，此时就需要在执行到指定位置时手动设置一下数据的取值。

具体指令：

复制变量：set 变量=表达式

### 监控变量

watch 变量🡪c（改变时暂停运行）

awatch <表达式> 被访问或者改变时暂停运行

rwatch <表达式> 被访问时暂停运行

说明：有些时候变量所在的函数直接执行不到哪里，外部消息也无法触发执行，这个时候就可以使用变量追踪。尤其适用于全局变量的追踪，可以发现其初始化、赋值等异常。

### 打印变量/表达式

#### print

在调试中需要不断打印变量或者函数执行结果，已查看是在哪一步执行错误。

具体指令：

打印变量：p 变量

打印字符/表达式：print “%s”,字符/表达式

print /x var

用16进制显示var值，print可以指定显示的格式，这里用/x表示16进制的合适。支持的变量显示格式有：

x 16进制格式显示变量

d 10进制格式

u 16进制显示无符号整型

o 8进制显示变量

t 2进制显示变量

a 16进制格式显示变量

c 字符格式显示变量

f 浮点数格式显示变量

有时候需要打印结构体的成员变量，直接p是打印不出来的，需要格式化输出打印的字符串或者数字。比如：

print “%s”,array[i].name

格式化输出：p/格式控制符 打印内容（格式控制符包括：d十进制打印，c字符打印等）

#### x

也可以使用x（Examination）来打印需要显示的字符信息，格式如下：

x/格式 地址

格式一般是NFU，N表示重复次数，F表示格式，U表示单位（b：字节，h：半字[2字节]，w：字[4字节，默认]，g：双字[8字节]）。

注：print打印字符的内容，x打印的是内存地址的内容。

#### display

display e 程序断住显示某个变量的值

### 变量类型

ptype var 变量类型

whatis var 显示一个变量var的类型

## 查看代码

列出指定区域（n1到n2之间的）代码：

（gdb） list n1 n2 / list file:n

其中list可以简写为l，如果使用-tui启动gdb的话，将会在相应的位置显示。如果没有n1和n2参数，那么就会默认显示当前行和行后的10行diamante，再执行又往下滚动10行。另外，list还可以接函数名。

一般来说，在list后面可以跟下列参数：

<linenum> 行号

<+offset> 当前行号的正偏移量

<filename:linenum> 哪个文件的哪一行

<function> 函数名

<filename:function> 哪个文件中的哪个函数

<\*address> 程序运行时的语句在内存中的地址

有时候对于比较长的结构体或者类成员显示不完整或格式混乱，可以采用如下办法：

set print pretty on //设置符合阅读的代码显示

set print element 0 //完整打印字符信息

### 修改源码路径

如果将源码（比如main.c）文件移动到/home/tmp目录下，再次执行l命令：

(gdb) l

main.c: No such file or directory.

会提示找不到源码文件了，我们可以使用dir命名指定源码路径，例如：

(gdb) dir ./temp

Source directories searched: /home/hyb/workspaces/gdb/sourceCode/./temp:$cdir:$cwd

这个时候它就能找到源码路径了。我这里使用的是相对路径，保险起见，你也可以使用绝对路径。

### 编辑源码

可以直接在gdb模式下编辑源码，它默认使用的编辑器是/bin/ex，如果机器上未安装该编辑器或者想使用别的编辑器，可以通过下面的方式进行设置：

$ EDITOR=/usr/bin/vim

$ export EDITOR

/usr/bin/vim替换为你使用的编辑器的路径，如果你不知道你的编辑器在什么位置，可借助whereis命令或者witch命令查看：

$ whereis vim

vim: /usr/bin/vim /usr/bin/vim.tiny /usr/bin/vim.basic /usr/bin/vim.gnome /etc/vim

/usr/share/vim /usr/share/man/man1/vim.1.gz

$ which vim

/usr/bin/vim

设置之后，就可以在gdb调试模式下进行编辑源码了，使用命令edit location，例如：

(gdb) edit 3 #编辑第三行

(gdb) edit printNum #编辑printNum函数

(gdb) edit test.c:5 #编辑test.c第五行

编辑完保存后，别忘了重新编译程序：

(gdb)shell gcc -g -o main main.c test.c

这里要注意，为了在gdb调试模式下执行shell命令，需要在命令之前加上shell，表明这是一条shell命令。这样就能在不用退出GDB调试模式的情况下编译程序了。

## 调试函数

### disassemble

可以使用反汇编的指令去探究究竟在函数中发生了哪些操作，具体如下：

1、disassemble

2、disassemble 程序计数器

3、disassemble 开始地址 结束地址

格式1表示反汇编当前整个函数，格式2表示反汇编计数器所在函数的整个函数，格式3表示反汇编从开始地址到结束地址的部分。

## 退出调试

在执行到断点后，采用q/quit指令退出。

说明：比如使用gdb调试的进程A，然后不小心使用kill -9将进程A杀死了，这就产生了僵尸进程，即进程A依附的父进程gdb还未杀死，此时使用ps查看gdb进程号，然后kill -9杀死即可。

# 多线程调试

## 查看线程

查看线程：info threads

注：输出信息前面有“\*”表示调试的当前线程。

## 切换线程

切换线程：thread N

注：通过打印counter，可以看到多个线程都是在运行的，如果想要让其他线程处于停止状态，只有当前调试的线程执行，可以采用set scheduler-locking on。

## 阻塞线程

阻塞其他线程，仅调试当前线程工作：set scheduler-locking [on|off|step]

多线程环境下，会存在多个线程运行，这会影响调试某个线程的结果，这个命令可以设置调试的时候多个线程的运行情况，on 表示只有当前调试的线程会继续执行，off 表示不屏蔽任何线程，所有线程都可以执行，step 表示在单步执行时，只有当前线程会执行。

注：此时打印counter计数器，则仅当前调试的线程计数器增加，其余线程的计数器不改变。

# 多进程调试

## 配置

### detach-on-fork

该属性决定了gdb是同时调试父子进程，还是在fork了子进程之后，将子进程分离出去。

on：子进程（或者父进程，取决于gdb在初始时，要调试的进程，也就是follow-fork-mode的值）

off：同时调试父子进程，一个进程处于被调试的状态，而另一个则被gdb挂起

设置：set detach-on-fork on/off

### follow-fork-mode

该属性决定了gdb在进程调用fork之后的行为。

set follow-fork-mode parent：默认情况下，在调用fork之后，gdb选择跟随（也就是调试）父进程，而子进程则在处于运行的状态（此时父进程处于阻塞的状态）。

set follow-fork-mode child：fork之后gdb选择调试子进程，而父进程处于运行的状态。

## 查看进程

查看当前调试的进程：info inferiors

# core文件调试

core是程序非法执行后core dump产生的文件，可以通过core文件直接定位到程序崩溃的位置。如果开启了core文件，那么在系统异常退出并生成core文件的时候，调试就显得非常简单了。

具体指令：

ulimit –c unlimited （产生core文件）

gdb <program> core

# 内核调试

## 字节序

低位数据排在内存低地址。叫做Little Endian，这是Intel架构采用的方法。相反，将高位数据放在低位地址的方式叫做Big Endian，为MIPS、SPARC架构采用。

以0x12345678为例子，小段模式显示如下：

0003 0002 0001 0000

0x12 0x34 0x56 0x78