调试工具gdb

在linux环境下编写C/C++程序当需要进行调试时，gdb是非常常用且强大的利器。在此之前， 因为自己写的程序普遍比较短，所以基本很少需要用到具体的调试工具，用的最多的还是万能调试语句printf来打印出具体的变量值来确定是否存在问题。

不过大型一些的工程用上面的方式来调试就不是很妥了，gdb将可以为我们带来调试上的便捷，当然，从本质上来说gdb的功能其实和printf非常接近，printf本质上也是在程序中某个中间位置打印出变量的值。

当我们在命令行下输入gdb之后，若能进入gdb待输入模式下则说明电脑中已经安装了gdb。在使用gdb前我们必须先直到个gdb的输入可执行文件，实际上也就是源文件经过预处理、编译、汇编、链接之后的可执行二进制文件。

由于我们我们这里进行的是对可执行文件的调试，所以在用gcc进行编译时需要添加选项-g用于在我们生成的可执行文件中加入调试信息，这样在调试时候就不会看到全是地址，而是用函数名、变量名代替。另外尤其需要注意的是，在使用gdb时不能使用gcc的优化选项，即-O选项，当使用优化选项之后，部分变量存储的位置就和执行顺序就会发生变化，不易于我们调试。

在介绍gdb的基本用法时候，我们将会通过以下代码来讲解。

// 文件保存为debug.c

#include <stdio.h>

int sum = 0;

int add(int n)

{

int i;

for(i = 1; i <= 100; ++i)

sum = sum + i;

return sum;

}

int main()

{

int i = 100;

int sum;

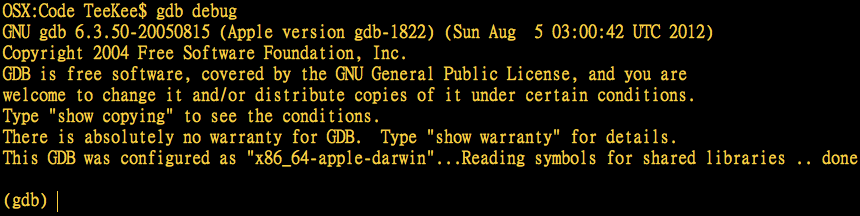
sum = add(i);

printf("result = %d\n", sum);

}

上述代码保存之后，我们编译方式为：gcc debug.c -g -o debug

之后通过以下方式导入可执行文件，当然也可以直接先输入gdb按下会回车再在(gdb)提示符之后输入将要导入的可执行文件名。



（图1：gdb导入可执行文件）

在完成执行文件的导入之后，当需要显示导入二进制文件的源代码时，可以通过gdb内置的list命令来显示。至于list的参数有很多，在我目前使用当中最常用的包括以下几个：

1. (gdb)list [linenum] // 显示第line行附近的源代码

2. (gdb)list [function] // 显示function的源代码

3. (gdb)list // 显示当前行后面的源代码

4. (gdb)list - // 显示当前行前面的源代码

比如我们要显示debug.c文件中add函数的源代码时候只需要输入list add即可。

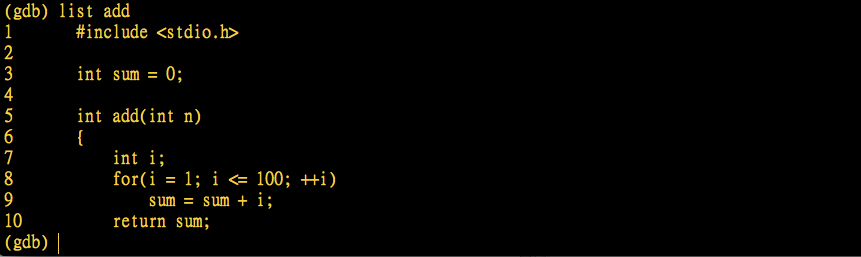


图2：list列出源代码

这里可以看到列出源代码时并不是把我们作为参数的函数放在首行的，而是置于所列部分的中间位置。在默认情况下gdb会输出10行代码，不过我们可以同过set listsize [size]来调整输出的行数，调整后显示的行数明显增加了很多。

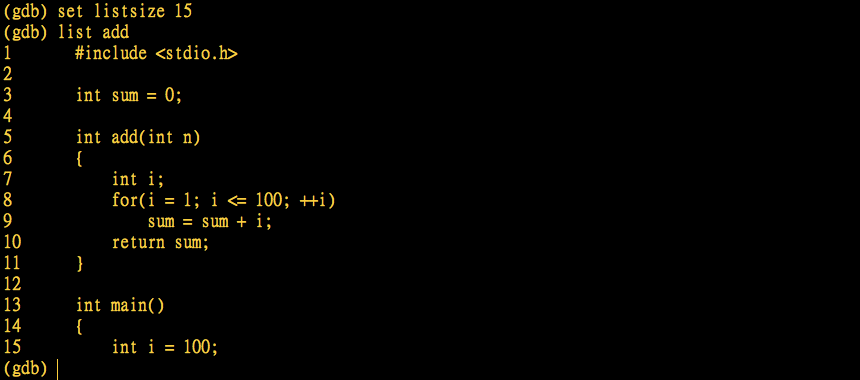


图3：修改list显示行数

不过，直到现在，对于本质工作的调试任务，我们依旧没有做出任何实际的事情。

对于gdb而言，最基本的操作就是设置断点，让程序在执行过程中具体停在某行或者某个函数的入口处，这样我们可以从断点处逐条语句执行，然后在这执行过程中既可以输出我们所需变量的值，又可以查看代码实际执行的顺序（回忆用了switch而没有加break的情况），帮助我们判断执行顺序或结果和我们的预期是否一致。

设置断点的在gdb中的命令是break（也可以简写成b），常见的设置断点位置的的方式有以下几个：

1. (gdb)break [linenum] // 在line行设置断点

2. (gdb)break [function] // 在function入口设置断点

3. (gdb)break +offset // 断点设在当前停止位置+offset行处

4. (gdb)break –offset // 断点设在当前停止位置-offset行处

比如我们需要在add函数入口和第17行代码处设置断点，则只需要分别输入break 17和break add即可。

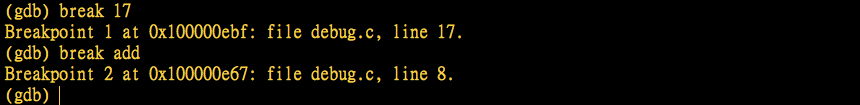


图4：设置断点

断点被设置后，会在接下来一行立即得到反馈是否已经设置成功，如果设置成功的话会得到上图所示的提示，并且也可以发现原本在add函数入口设置的断点被转换成了代码第8行的位置，但实际上二者是通用的。另外可以看到，每个断点都按照输入的顺序被赋予了一个唯一的断点号，暂时我们只需要知道有这么个东西就行，具体使用到它的时候会着重提一下。当我们设置大量断点之后，我们可能需要查看我们已经设置了哪些断点以及每个断点的信息，这就需要通过info命令来提供帮助，键入info breakpoints就可以得到关于已设断点的内容。

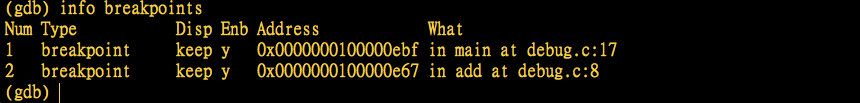


图5：列出断点信息

当然，有些时候，断点使用太多时，调试会很慢，这时候我们可以删除某些不需要的断点，这里就涉及到clear、delete、disable命令了。他们的常见用法如下：

(gdb)clear [linenum] // 清除第line行断点

(gdb)clear [function] // 清除function处断点

(gdb)delete [breakpoints] [range] // 删除指定停止点

(gdb)disable [breakpoints] [range] // 使指定停止点暂时无效

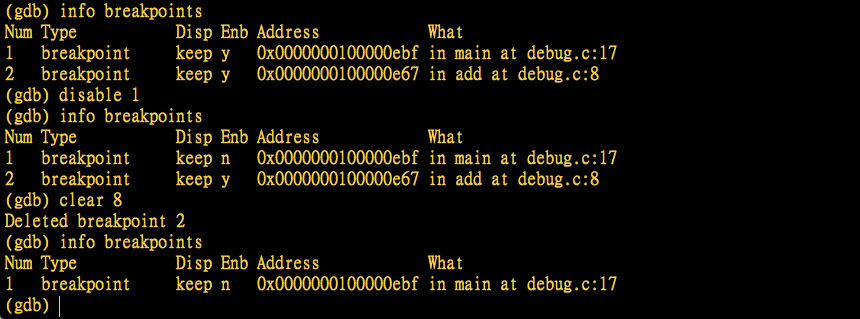


图6：去除断点

上图最开始展示了当前断点情况，可见两个断点号分别分1、2的两个断点，并且这两个断点的enable位都显示二者是y（可用）。当输入了disable 1之后再打印断点信息可见断点号为1的断点的enable位被置为了n。接下来使用了delete命令删除了断点号为2的断点，再去查看断点信息时发现该断点确实已经被删除了。

至于enable位被置为n的断点并没有被删除，只需要通过命令enable就可以将其恢复。

当我们通过断点将程序暂停在某条语句的执行上时，我们可以在此断点位置打印出即刻的变量值，但是我们通常不这么做。在我看来，设置断点的主要目的是为了判断程序是否按照既定的准确的执行顺序执行，对于数据值的变化更倾向于使用watchpoints（观察点）。当然了，观察点也是一种断点。

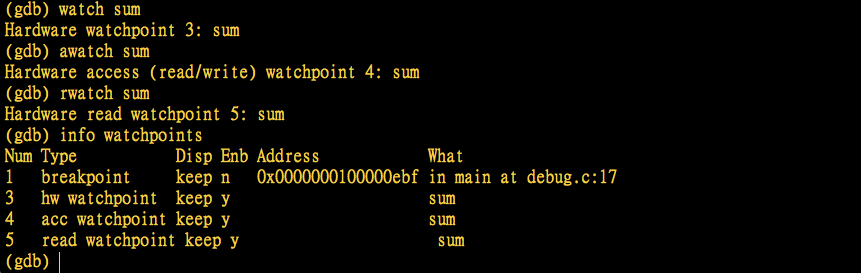


图7：设置观察点

上图展示了设置观察点最常见的三种用法，三条命令观察点都是sum变量，程序执行过程中当触发这三种命令的条件时就会暂停在该行。不过实际运用中是不会将三个观察点设置在一处的，因为三种命令本身功能上有重合的地方。

先来解释一下三者的功能：

(gdb)watch [expr] // 表达式或变量值一旦变化立刻暂停

(gdb)awatch [expr] // 表达式或变量一旦被读或改写立刻暂停

(gdb)rwatch [expr] // 表达式或变量一旦被读立刻暂停

看了上面的功能描述想必就不用再去解释截图中各条语句的意思了，特别需要注意的是当通过info命令列出所有观察点时也出现了之前设置的断点，确实二者都是可以暂停程序的执行。另外，断点（观察点）号是自增的，当我们上一步删除了断点2之后，断点的序号将会从3开始分配。

接下来我们需要在gdb环境下执行debug程序，方式就是输入命令run，输入后程序将会执行，按照预期，程序将会在读取和改变add函数中sum变量时被暂停。

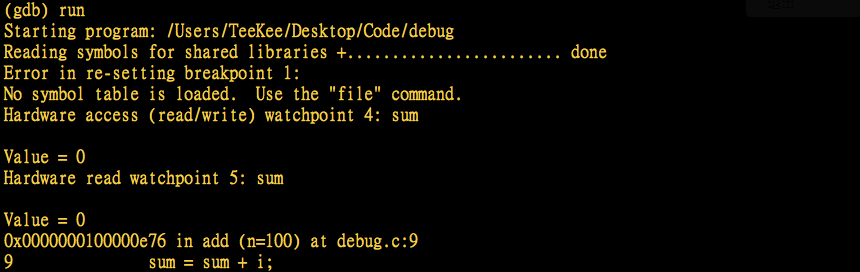


图8：gdb下执行程序及触发观察点

通过输出可以发现观察点号为4（awatch）、5（rwatch）的条件分别被触发而暂停了程序，使程序停在了代码的第九行语句上，通过图2可知该语句是sum = sum + i。至于为什么没有出发watch命令的条件在此断点中断从三种命令的描述中即可得出：awatch和rwatch的触发条件中只需要程序读取sum变量就会中断程序（此时只完成了等号右侧sum取初值工作），而watch则是当sum值被改变时才会被触发。而当sum变量刚被读取就因触发了awatch和rwatch的条件而被中断，所以此时并没有来得及改变sum值，也就没看到观察点3的信息。

我们可以通过命令next（也可以只输入n）来从断点重新启动程序，不过需要说明的是，通过next一次只能向下执行一行代码。

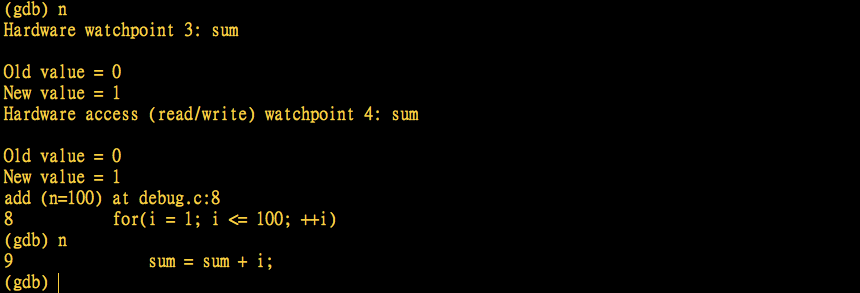


图9：gdb单步执行及数值变化

当我们通过n（next）命令向下多执行一条指令的时候，此时完成了sum = sum + i的执行，sum的值由初值0变成了1。因为只有watch和awatch（相当于watch和rwatch功能之和）会关注变量数值被改写的问题，所以此时观察点3号和4号被触发，暂停在该语句，之后打印出sum变量在执行该语句之前和之后值的变化。

虽然next命令可以加指令条数的参数，但通常还是用于单行代码执行较多，与next具有差不过功能的还有step功能，这里就不再一一说明了。上面提到的next命令是执行断点位置的下一行代码，其实next还有更精细化操作的版本nexti，我们知道，一条代码编译后会形成多条汇编指令，每一条汇编指令对应至少一条机器指令，而nexti则可以保证在断点处之后每次只执行一条机器指令，通常只有很罕见的问题才需要通过机器指令级调试来法线问题（比如多线程并发）。

由于上面有100次循环，所以我们使用next命令时候，程序会不停地在第八第九行代码之间来回切换。为了解决这种问题，我们需要介绍下一个命令：continue（简写c）。

当我们把断点设置在某处得到想要的信息之后，如果想要恢复程序执行则需要通过continue命令来完成，continue命令可以回复程序执行，直到程序结束或是下一个断点到来，具有功能的命令还有fg。

大部分情况下continue都可以帮助我们跳出循环语句，但是在本例中却不能达到效果。原因是contiune在不加参数情况下当遇到下一个断点/观察点时依旧会暂停程序，因为本例中sum的值每次循环都会变化，所以每次循环回来都会触发一次暂停，所以continue基本就退化成了next命令。

我们可以通过给continue增加参数来完成跳过循环的任务，在continue后可以增加参数[ignore-count]来跳过表示忽略其后断点次数。本例情况比较复杂，为了能直接结束程序，这里使用的ignore-count参数值给了300，结果成功忽略了所有观察点的暂停，执行完了程序，返回结果5050。

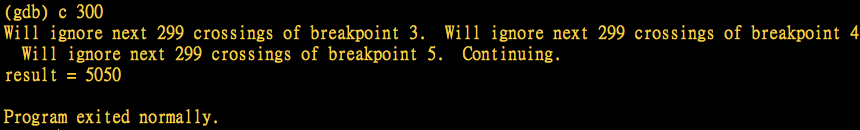


图10：忽略指定数量断点

上面已经涉及了大部分gdb的基本操作，但还有很多高级的玩法没有描述。比如我们需要在某个断点处设定运行命令。为了简化描述过程，我重新打开了一次gdb并载入刚刚的debug可执行文件。

假设我有个需求，在add函数中把sum值小与10前每次的变化都显示出来。这时候首先需要在第九行设置断点，然后判断此刻sum的当前值，如果sum值小于10则输出。但这里毕竟不是在写程序，想要完成这个步骤必须先就将我们所需执行的命令保存好并与某个断点号绑定，这样我们就可以在断点处执行预先保存好的命令了。能够完成这一任务的就是commands命令，下面先看一下它的用法截图。

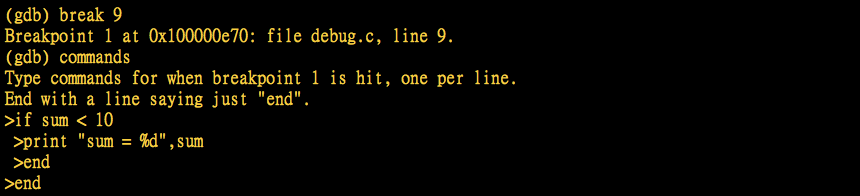


图11：commands运行命令

在设置断点同时键入commands可以进入编辑模式，之后输入条件判断语句和输出语句，输入完成之后通过在最后添加end来结束本层输入，当结束最外层输入的end键入后会从command编辑模式回到gdb模式下。此时运行程序时会首先在第九行的断点处暂停，之后每次输入c（continue）之后都会先判断sum值然后输出，不过根据条件，只会输出sum小于10的几个结果。

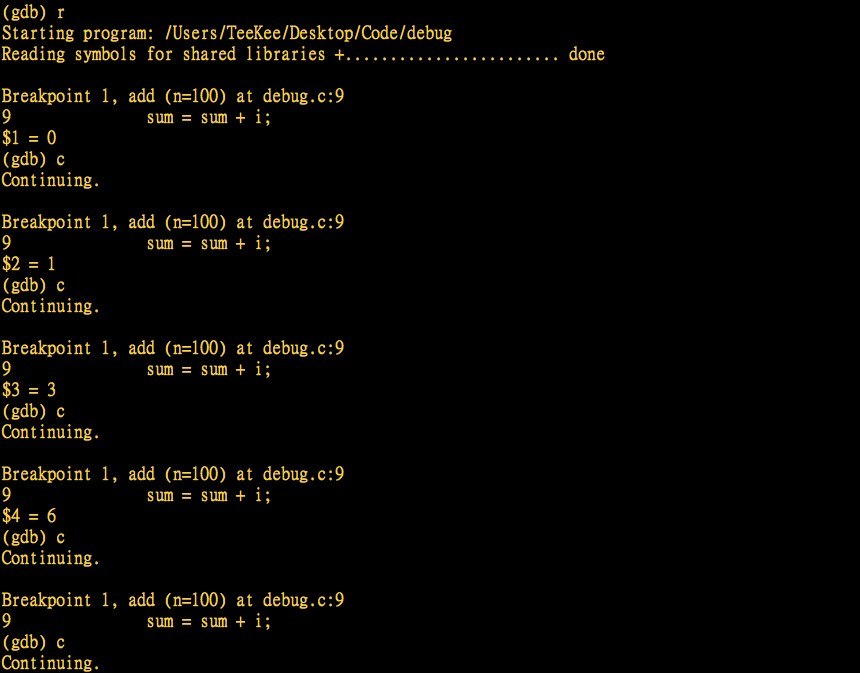


图12：commands测试结果

如我们commends中设定的条件，通过执行程序，打印出了sum小于10的所有值。

除了以上提到的内容外，gdb还可以做很多高级的操作，下次有机会用到再说。