第 5 章

调试器的基本命令

通过前面的章节，可以了解到目前支持. NET Core 调试的调试器主要是 LLDB 和 Windbg。LLDB 适用于 Linux 和 macOS 操作系统，而 Windbg 则适用于 Windows 操作系统。本章介绍这两种调试器的一些基本调试指令。

从本章以后，所有涉及的相关调试项目工程，请通过以下地址进行下载：

https：//github. com/micli/netcoredebugging

5.1 使用 LLDB 进行调试

由于 LLDB 的高可扩展性和组件化设计，使得 LLDB 成了. NET Core 在 Linux 和 macOS 平台上进行调试的唯一选择。. NET Core 为 LLDB 创建了调试扩展，在 LLDB 启动之后，通过挂载. NET Core 的调试扩展来兼容. NET Core 的托管代码调试。下面介绍 LLDB 调试器的基本使用方法。

5.1.1 LLDB 调试器简介

LLDB 是下一代高性能调试器。它在一种基于“BSD 风格”的开源许可，即 LLVM 开源许可下开放源代码。LLDB 功能强大，是 XCode IDE 指定的默认调试器。在 Linux 上调试. NET Core 应用程序就必须使用 LLDB 调试器。

LLDB 调试器是一款高度组件化的调试器。它是 Clang 编译器发展的必然产物。因为基于 LLVM 许可协议开放源代码，LLDB 具有以下特点：

(1) 支持插件化的软件架构，具有高可扩展性。

(2) 支持 Python 对调试器的完全控制，可以用 Python 控制调试器完成自动化动作。

(3) 支持多框架多平台的远程调试，包括 x86、x64 和 macOS，是 XCode 默认调试器。

(4) 完全开放的 API 和开发库。

正是基于 LLDB 调试器的高度可扩展性，开发人员通过扩展插件支持. NET Core 的代码调试。而传统的 GDB 是没有这样的支持插件的能力的，也无法用来调试. NET Core 代码，LLDB 调试器是目前 Linux 和 macOS 平台上的唯一选择。

5.1.2　命令行参数

LLDB 工具位于/usr/bin/目录下，单独启动 LLDB 非常简单，直接输入 lldb 即可。对于 Debian/Ubuntu，安装的 LLDB 3.9 可执行文件都带有版本号码。对此，可以通过“ln”命令为 lldb-3.9 文件设置软连接的方式，将版本号码掩盖。如图 5.1 是直接使用 lldb-3.9 的方式。

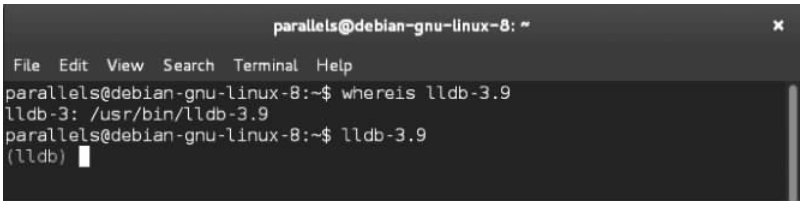

图 5.1　查找 LLDB 3.9 的位置

设置软连接如命令 5.1 所示。

```
# 查看 lldb-3.9 可执行文件的路径
debian-gnu-linux-8:~$ whereis lldb-3.9
lldb-3: /usr/bin/lldb-3.9 # 返回 lldb-3.9 的路径
# 设置软连接
debian-gnu-linux-8:~$ sudo ln -s /usr/bin/lldb-3.9 /usr/bin/lldb
# 重定向 lldb-server 3.9
debian-gnu-linux-8:~$ sudo update-alternatives --install /usr/bin/lldb-server
lldb-server /usr/bin/lldb-server-3.9 100
```

命令 5.1　设置软连接

设置之后，Debian 或者 Ubuntu 可以使用 lldb 命令来直接启动 LLDB 3.9 版本的调试器，如图 5.2 所示。

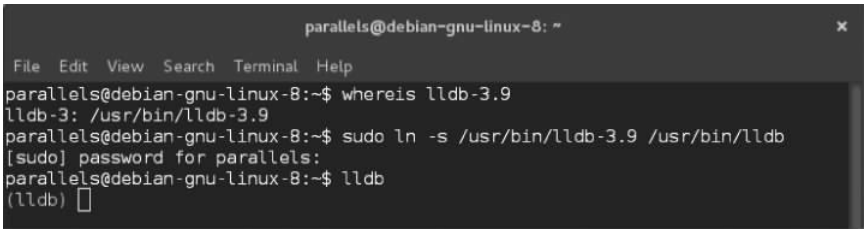

图 5.2　用 lldb 命令启动 LLDB 3.9

(lldb)是 LLDB 的命令行提示符，表示当前终端正运行在 LLDB 调试器之下。在当前提示符之下输入 help 命令，可以查看 LLDB 内置的最基础的调试命令。但是，LLDB 的使用方式是在启动时同时挂载应用程序进程或者应用程序的内存转储文件，以便对程序进行

调试，因此 LLDB 通常是带参数启动。

LLDB 的启动参数格式如命令 5.2 所示。

```
lldb [[ -<参数缩写>]] [[ -- ] <参数全名 - 1 > [<参数全名 - 2 > . . . ]]
```

命令 5.2　LLDB 启动参数格式

在 Linux/macOS 世界，通常情况下，“--”后面会跟随参数的完整名字，而“-”后面跟随参数的缩写。在 Windows 世界，通常只使用参数的缩写来标明参数，即使使用参数全名称也不会通过破折号的数量来标明是参数的全名还是缩写。

LLDB 的启动参数如表 5.1 所示。

表 5.1　LLDB 3.9 启动参数

参 数 名 称	含　义
-a < arch > --arch < arch >	向 LLDB 调试器声明要调试的应用程序的架构。是 64 位的还是 32 位的或者是 ARM 的
-f < filename > --file < filename >	向调试器指定要调试的应用程序对应在磁盘上的文件，LLDB 调试器可以帮助使用者从磁盘上运行一个新的应用程序实例
-c < filename > --core < filename >	在进行事后调试时，向调试器指定内存转储文件的位置，也就是 core 文件的位置
-p < pid > --attach-pid < pid >	要调试一个正在运行的应用程序时，通过该参数告诉 LLDB 调试器要附加的目标进程 ID
-n < process-name > --attach-name < process-name >	要调试一个正在运行的应用程序时，通过该参数告诉 LLDB 调试器要附加的目标进程的进程名字
-w --wait-for	告诉 LLDB 调试器，在附加某个指定的进程之前，等待某个给定的进程 ID(pid)或者进程名字的进程启动
-s < filename > --source < filename >	告诉调试器，在命令行上 LLDB 启动之后，将指定的文件读入 LLDB 调试器中
-o --one-line	告诉 LLDB 调试器，在某个指定的文件加载之前运行一行指定的 LLDB 调试器命令
-S < filename > --source-before-file < filename >	让 LLDB 调试器在加载指定的文件之前，先读取该参数指定的 LLDB 命令批处理文件，并执行
-O --one-line-before-file	告诉 LLDB 调试器，在加载指定的文件之前，先执行该参数指定的一行 LLDB 调试指令
-k --one-line-on-crash	告诉调试器，一旦被调试的应用程序发生崩溃的情况，先执行一行指定的 LLDB 调试命令
-K < filename > --source-on-crash < filename >	告诉 LLDB 调试器，一旦被调试的应用程序发生崩溃的情况，先加载指定的文件，并读取文件的内容执行
-Q --source-quietly	告诉 LLDB 调试器，在加载任何指定的文件之前，先预先执行一行 LLDB 调试命令

续表

参 数 名 称	含　义
-b --batch	告诉调试器从-s,-S,-o 和-O 运行参数指定的命令,然后退出。但是,如果任何运行命令由于信号而停止或崩溃,调试器将返回到交互式提示符,并停止在崩溃的地方
-l < script-language > --script-language < script-language >	告诉调试器使用指定的脚本语言替换 LLDB 调试器默认指定的脚本语言。可以指定的语言包括 Python,Perl,Ruby 和 Tcl。目前只有 Python 扩展已经实现
-d --debug	告诉 LLDB 调试器,输出更多与自身有关的调试信息

5.1.3　一段用于演示的代码

为了演示方便,下面的操作会使用一段 C++的快速排序代码。源代码如代码 5.1 所示。

```
#include<iostream>
#include "stdio.h"
using namespace std;

void quickSort(int a[],int,int);

int main()
{
    int array[] = {14, 61, 32, 53, 37, 25, 87, 21, 13, 77},k;
    int len = sizeof(array) / sizeof(int);
    cout << "The orginal arrayare:" << endl;
    for(k = 0; k < len; k++)
        cout << array[k] << ",";
    cout << endl;
    quickSort(array, 0, len - 1);
    cout << "The sorted arrayare:" << endl;
    for(k = 0; k < len; k++)
        cout << array[k] << ",";
    cout << endl;
    ::getchar();
    return 0;
}

void quickSort(int s[], int l, int r)
{
    if (l < r)
    {
        int i = l, j = r, x = s[l];
        while (i < j)
```

```
        {
            while(i < j && s[j] >= x)
                j--;
            if(i < j)
                s[i++] = s[j];
            while(i < j && s[i]< x)
                i++;
            if(i < j)
                s[j--] = s[i];
        }
        s[i] = x;
        quickSort(s, l, i - 1);
        quickSort(s, i + 1, r);
    }
}
```

代码 5.1　快速排序代码

这段代码用来将一个给定的无序数组用快速排序算法进行排序，并将排序的结果进行输出。以上代码编译后的结果是一个名为 qsort 的可执行文件。

5.1.4　LLDB 的启动和退出

(1) 通过 LLDB 来启动应用程序并进行调试。

在这种情况下，只需要传递要启动的应用程序给 LLDB 即可，如命令 5.3 所示。

```
# 假设 qsort 就在当前路径下
lldb ./qsort
```

命令 5.3　LLDB 加载 qsort 程序

(2) 通过 LLDB 调试器附加到已经启动的应用程序进程上。

在这种情况下，首先要获取应用程序的进程 ID 或进程名字，然后在启动 LLDB 调试器时，传入应用程序进程的 ID 或名字。有些情况下，一个应用程序会在内存中有多个副本，因此推荐使用应用程序进程 ID 的方式。

在附加进程时，还需要 LLDB 调试器运行在管理员权限下，否则无法成功地附加到一个指定进程上。一个进程上只能同时附加一个调试器。

运行进程的 ID 可以通过 ps aux 命令进行查询，如命令 5.4 所示。

```
# 假设 qsort 的进程 ID 是 2254
sudo lldb -p 2254
```

命令 5.4　LLDB 附加到 qsort 进程

(3) 使用 LLDB 对内存转储文件进行事后调试。

在这种情况下,需要使用-c 或--core 参数向调试器指定要加载的内存转储文件的位置,如命令 5.5 所示。

```
# 加载转储文件
sudo lldb -c ./qsort.core
```

命令 5.5　LLDB 打开 qsort 内存转储文件

5.1.5　设置断点

LLDB 支持在程序的源代码中预先设置断点,当代码运行到预设的断点时,程序会临时终止运行并切换到调试模式下。此时调试者可以使用调试命令查看堆栈、内存以及相关变量等。因此,设置断点是最基础的操作,是用 LLDB 调试应用程序的前提。

LLDB 设置断点的调试命令是 breakpoint,缩写是 br。LLDB 支持通过函数名字或者指定源代码文件和行数的方式设置断点,如命令 5.6 所示。

```
(lldb) breakpoint set -- method main
(lldb) breakpoint set -- file main.cpp -- line 22
```

命令 5.6　LLDB 设置断点

在实际操作中,在 qsort 示例的源代码文件第 31 行设置一个断点,并运行程序。当程序运行到断点位置时,LLDB 将会自动切换回调试模式,如图 5.3 所示。

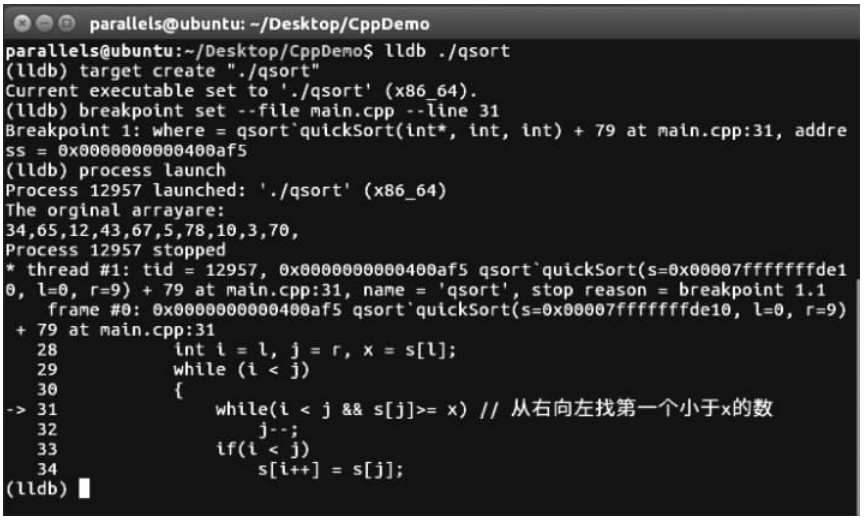

图 5.3　LLDB 断点触发

断点触发后，LLDB 会先打印出断点的信息，并将断点一定的段代码片段进行输出。调试者此时就可以对程序进行进一步的调试。当调试动作结束后，调试者可以输入 continue 命令，让应用程序继续运行。LLDB 会切换到应用程序运行模式，直到下一次断点被触发。如例子中，代码的第 31 行是 while 循环的一部分，所以断点会不断地被触发，如图 5.4 所示。

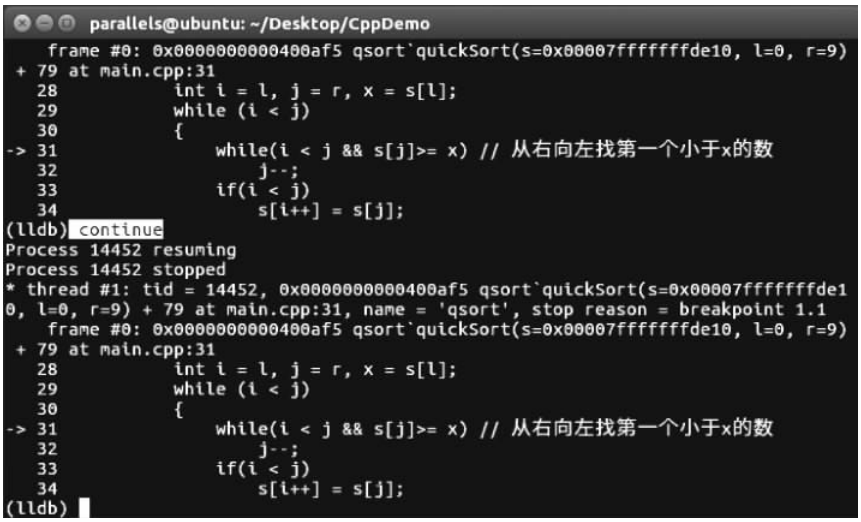

图 5.4　continue 命令

如果需要重新运行应用程序，而不是继续运行，那么就需要输入 run 命令。该命令会强制结束当前应用程序进程，让 LLDB 重新启动一个新的应用程序进程，并保留之前设置的断点信息。

如果调试者需要在应用程序运行的任意时刻暂停应用程序来进行调试，而不依赖断点触发，那么可以按下“Ctrl + C”键，将 LLDB 转为调试模式对应用程序进行调试。这在没有预设调试断点，临时进行调试的场景中非常重要。

当进入调试模式之后，调试者为了进一步地分析问题，经常需要单步执行代码，一步一步地查看代码执行过程和相关变量值。在这种情况下就需要使用单步调试指令。

5.1.6　单步调试指令

所谓单步调试指令，通常是指执行下一行代码、执行下一个函数、从当前函数跳出等操作。调试者可以利用单步调试指令对应用程序进行单行，以及单行代码调用的函数进行调试。这是调试中经常使用到的操作。

LLDB 提供的单步调试操作分为执行下一个函数操作 next，执行下一行汇编指令操作 ni 两种。next 指令可以让 LLDB 控制调试器的执行过程直接跳转到下一个函数调用中，而 ni 指令是让 LLDB 执行处理器 rip 存储器中的下一行汇编指令。要知道，即使是源代码中

一行简单的代码也会被编译器编译成多行汇编指令，因此 next 和 ni 是有很大区别的。

LLDB 也同时提供了从被调用函数或者子过程中返回的调试指令 finish。这个指令用来从函数或者子过程中返回到上一级调用。

回到 qsort 例子，在源代码 main.cpp 文件的第 15 行，调用了 quickSort()函数。可以在这里设置断点、运行应用程序并等待断点触发。

触发断点之后，通过 step 命令，可以从 main 函数的第 15 行，跳入到 quickSort 函数中去(源代码第 26 行)，如图 5.5 所示。

```
* thread #1: tid = 11003, 0x0000000000400a0d qsort`main + 199 at main.cpp:15, name
 = 'qsort', stop reason = breakpoint 1.1
    frame #0: 0x0000000000400a0d qsort`main + 199 at main.cpp:15
   12       for(k=0;k<len;k++)
   13           cout<<array[k]<<",";
   14       cout<<endl;
-> 15       quickSort(array,0,len-1);
   16       cout<<"The sorted arrayare:"<<endl;
   17       for(k=0;k<len;k++)
   18           cout<<array[k]<<",";
(lldb) step
Process 11003 stopped
* thread #1: tid = 11003, 0x0000000000400ab8 qsort`quickSort(s=0x00007fffffffde10,
 l=0, r=9) + 18 at main.cpp:26, name = 'qsort', stop reason = step in
    frame #0: 0x0000000000400ab8 qsort`quickSort(s=0x00007fffffffde10, l=0, r=9) +
 18 at main.cpp:26
   23
   24   void quickSort(int s[], int l, int r)
   25   {
-> 26       if (l< r)
   27       {
   28           int i = l, j = r, x = s[l];
   29           while (i < j)
(lldb)
```

图 5.5　step 命令

若需要从 quickSort 函数返回到调用它的那一行代码，执行 finish 命令。这样 LLDB 调试器就会控制应用程序执行完 quickSort()函数的全部代码，并将代码调试停止在 main 函数源代码的第 16 行，如图 5.6 所示。

```
(lldb) finish
Process 11003 stopped
* thread #1: tid = 11003, 0x0000000000400a24 qsort`main + 222 at main.cpp:16, name
 = 'qsort', stop reason = step out
    frame #0: 0x0000000000400a24 qsort`main + 222 at main.cpp:16
   13           cout<<array[k]<<",";
   14       cout<<endl;
   15       quickSort(array,0,len-1);
-> 16       cout<<"The sorted arrayare:"<<endl;
   17       for(k=0;k<len;k++)
   18           cout<<array[k]<<",";
   19       cout<<endl;
(lldb)
```

图 5.6　finish 命令

5.1.7　查看调用堆栈

当应用程序运行过程中触发了断点之后，就可以对应用程序进行进一步的调试。往往调试者首先需要看的就是应用程序调用堆栈。调用堆栈会显示当前线程中的代码调用回溯

信息。通过回溯信息可以了解到函数的层层调用过程。

查看调用堆栈信息的调试命令是 backtrace，简写为 bt。在示例中，待断点触发之后，输入 bt，如图 5.7 所示。

```
(lldb) bt
* thread #1: tid = 15686, 0x0000000000400af5 qsort`quickSort(s=0x00007fffffffde1
0, l=0, r=9) + 79 at main.cpp:31, name = 'qsort', stop reason = breakpoint 1.1
  * frame #0: 0x0000000000400af5 qsort`quickSort(s=0x00007fffffffde10, l=0, r=9)
 + 79 at main.cpp:31
    frame #1: 0x0000000000400a24 qsort`main + 222 at main.cpp:15
    frame #2: 0x00007ffff76ab830 libc.so.6`__libc_start_main(main=(qsort`main at
 main.cpp:8), argc=1, argv=0x00007fffffffdf28, init=<unavailable>, fini=<unavail
able>, rtld_fini=<unavailable>, stack_end=0x00007fffffffdf18) + 240 at libc-star
t.c:291
    frame #3: 0x0000000000400879 qsort`_start + 41
(lldb)
```

图 5.7　backtrace 命令

在图 5.7 中，一共显示了三帧(frame)数据，以倒序排列。位于顶端的 0 号帧(frame ＃0)，是断点触发时正在执行的函数 quickSort；1 号帧(frame ＃1)是 main 函数；2 号帧是库 libc 中的函数_libc_start_main。

当需要聚焦到调用堆栈的某一帧时，需要使用 frame select 或者 f（frame select 的缩写）调试命令，对帧进行选择，如图 5.8 所示。

```
parallels@ubuntu: ~/Desktop/CppDemo
(lldb) frame select 0
frame #0: 0x0000000000400af5 qsort`quickSort(s=0x00007fffffffde10, l=0, r=9) + 7
9 at main.cpp:31
   28           int i = l, j = r, x = s[l];
   29           while (i < j)
   30           {
-> 31               while(i < j && s[j]>= x) // 从右向左找第一个小于x的数
   32                   j--;
   33               if(i < j)
   34                   s[i++] = s[j];
(lldb) frame select 1
frame #1: 0x0000000000400a24 qsort`main + 222 at main.cpp:15
   12       for(k=0;k<len;k++)
   13           cout<<array[k]<<",";
   14       cout<<endl;
-> 15       quickSort(array,0,len-1);
   16       cout<<"The sorted arrayare:"<<endl;
   17       for(k=0;k<len;k++)
   18           cout<<array[k]<<",";
(lldb) frame select 2
frame #2: 0x00007ffff76ab830 libc.so.6`__libc_start_main(main=(qsort`main at mai
n.cpp:8), argc=1, argv=0x00007fffffffdf28, init=<unavailable>, fini=<unavailable
>, rtld_fini=<unavailable>, stack_end=0x00007fffffffdf18) + 240 at libc-start.c:
291
(lldb)
```

图 5.8　frame select 命令

跳转到某个指定的帧时，会自动输出该帧对应的源代码(如果有的话)，并且在帧对应的源代码的指定行数上，会显示一个箭头标明对应关系。

frame 命令还可以用来查看当前帧的相关临时变量的值，如图 5.9 所示。

通过观察临时变量的值，也可以帮助调试者定位应用程序中的问题。如果调试者只想查看某一个变量，可以用 frame variable <变量名>的方式来显示指定变量的值。这是非常有用的，毕竟一段代码中使用到的变量可能会有很多个。

```
(lldb) frame variable
(int *) s = 0x00007fffffffde10
(int) l = 0
(int) r = 3
(int) i = 0
(int) j = 3
(int) x = 3
(lldb)
```

图 5.9　frame variable 命令

5.1.8　线程切换

通常情况下，一个应用程序启动后会创建一个或多个线程。运行 main 函数的那个线程通常又称为主线程。对于相对复杂的逻辑，应用程序的主线程还会启动其他的线程来辅助进行计算工作。应用程序任何一个线程的代码都有可能出现问题，需要调试，因此调试器就必须具备在各个线程之间切换的能力。

LLDB 调试器的线程查看、调试的命令是 thread，由于线程操作功能众多，因此操作线程的命令往往是 thread <子命令>的形式。

thread list 用来枚举应用程序中的全部线程，并显示线程的简要信息。通过这个命令可以查看到当前被调试的应用程序有几个线程正在运行。

thread select <线程 ID >命令用于线程上下文切换。之前介绍的堆栈查看命令 backtrace(bt)用来显示当前线程的调用堆栈信息。那么如果想看看别的线程的调用堆栈信息，就需要使用 thread select 命令先切换线程上下文，将当前线程上下文切换到要查看的线程上，再执行 backtrace 命令进行查看。因此，thread select 命令非常重要，并且使用频率很高。为了简化操作，thread select 被简写为 t。假设要跳转到 #1 线程，那么既可以输入调试命令 thread select 1，也可以输入命令 t 1，这两者是等价的，如图 5.10 所示。

```
(lldb) thread select 1
(lldb) * thread #1: tid = 24298, 0x00007ffff7782230 libc.so.6`__GI___read + 16 a
t syscall-template.S:84, name = 'qsort', stop reason = signal SIGSTOP
    frame #0: 0x00007ffff7782230 libc.so.6`__GI___read + 16 at syscall-template.
S:84
(lldb) t 1
(lldb) * thread #1: tid = 24298, 0x00007ffff7782230 libc.so.6`__GI___read + 16 a
t syscall-template.S:84, name = 'qsort', stop reason = signal SIGSTOP
    frame #0: 0x00007ffff7782230 libc.so.6`__GI___read + 16 at syscall-template.
S:84
(lldb)
```

图 5.10　线程跳转命令

在线程上下文切换完成之后，LLDB 会自动显示当前线程的基本信息。如果有必要，也可以通过 thread info 来让 LLDB 调试器显示当前线程的信息。

5.1.9　寄存器调试指令

在具体进行调试时，往往还需要查看处理器的寄存器数据。处理器的寄存器不仅保存当前汇编指令的操作数，还保存当前执行的指令序列地址等其他关键信息。一些数据的首

地址，如字符串或者数组的首地址也保存在处理器的寄存器中。因此，在调试应用程序时，查看寄存器数据也是非常重要的。

LLDB 的寄存器调试指令是 register，查看所有寄存器的值可以使用 register read 指令。如果在 read 之后跟上寄存器的名字，就只显示指定寄存器中的数值。寄存器的数据与堆栈的帧密切相关。由于每个堆栈帧都在执行特定的函数或者汇编指令，因此寄存器中的数据也是与堆栈帧一一对应的。register 指令只会显示当前线程中调试者指定的堆栈帧上的寄存器数据，如图 5.11 所示。

```
(lldb) register read
General Purpose Registers:
       rax = 0xffffffffffffe00
       rbx = 0x00007ffff7a4f8e0  _IO_2_1_stdin_
       rcx = 0x00007ffff7782230  libc.so.6`__GI___read + 16
       rdx = 0x0000000000000400
       rdi = 0x0000000000000000
       rsi = 0x0000000000615030
       rbp = 0x00007ffff7a50620  _IO_2_1_stdout_
       rsp = 0x00007fffffffddb8
        r8 = 0x00007ffff7a51780  libc.so.6..bss + 4192
        r9 = 0x00007ffff7fd8740
       r10 = 0x000000000000035c
       r11 = 0x0000000000000246
       r12 = 0x0000000000400850  qsort`_start
       r13 = 0x00007fffffffdf20
       r14 = 0x0000000000000000
       r15 = 0x0000000000000000
       rip = 0x00007ffff7782230  libc.so.6`__GI___read + 16
    rflags = 0x0000000000000246
        cs = 0x0000000000000033
        fs = 0x0000000000000000
        gs = 0x0000000000000000
        ss = 0x000000000000002b
        ds = 0x0000000000000000
```

图 5.11　查看寄存器数据

在图 5.11 可以看到 rip 寄存器的地址实际上是指向 libc.so 中的 read()函数。rip 寄存器用来保存当前线程要执行的下一行代码的地址。处理器通过 rip 寄存器的地址确定到底要执行哪一行代码。

处理器寄存器的数据也可以通过 register write 指令进行修改。但是由于修改了寄存器数据之后会影响应用程序的执行结果，甚至不恰当的寄存器数据修改导致应用程序崩溃，因此，应谨慎使用 register write 对寄存器数据进行修改。

5.1.10　查看内存数据

在调试中，除了要查看应用程序代码的执行(backtrace 命令)，还需要查看应用程序中的数据。也就是说，要对应用程序占用的内存部分进行查看，以便检查变量的值以及引用的地址是否正确等。LLDB 的 x 调试命令(memory read)用来查看应用程序内存。x 调试命令使用相对复杂，参数众多。总结起来，x 命令通过命令 5.7 所示的格式来使用。

```
x <命令参数> <地址表达式> [<地址表达式>]
```

命令 5.7　x 命令参数格式

下面通过调试之前的应用程序来学习 x 命令的用法。首先可以在应用程序的第 32 行设置断点，然后查看 quicksort 函数的第一个参数的地址，并显示这个内存中指定的内容，如图 5.12 所示。

```
(lldb) breakpoint set -f main.cpp -l 32
Breakpoint 1: where = qsort`quickSort(int*, int, int) + 114 at main.cpp:32, addr
ess = 0x0000000000400b18
(lldb) run
Process 10742 launched: './qsort' (x86_64)
The orginal arrayare:
34,65,12,43,67,5,78,10,3,70,
Process 10742 stopped
* thread #1: tid = 10742, 0x0000000000400b18 qsort`quickSort(s=0x00007fffffffde1
0, l=0, r=9) + 114 at main.cpp:32, name = 'qsort', stop reason = breakpoint 1.1
    frame #0: 0x0000000000400b18 qsort`quickSort(s=0x00007fffffffde10, l=0, r=9)
 + 114 at main.cpp:32
   29          while (i < j)
   30          {
   31              while(i < j && s[j]>= x) // 从右向左找第一个小于x的数
-> 32                  j--;
   33              if(i < j)
   34                  s[i++] = s[j];
   35              while(i < j && s[i]< x) // 从左向右找第一个大于等于x的数
(lldb) x -count 8 0x00007fffffffde10
0x7fffffffde10: 22 00 00 00 41 00 00 00                          "...A...
(lldb)
```

图 5.12　x 命令

在堆栈的输出内容上，可以看到 quicksort 函数的第一个参数(int *)指向的地址是 0x00007fffffffde10。通过 x 命令查看这段内存的数据，并通过-count 参数执行查看 8 个字节。显示为 22 00 00 00 41 00 00 00 的内容，数据是以十六进制显示的，因此 0x21 实际上是 34，0x41 实际上是 65，也就是待排序的无序数组的前两个值。

至于为什么是 22000000，而不是 00000022？这其实是字节序的问题。从数据上可以看出，采用的是小字节序即低位数据占据低端内存，高位数据占据高端内存。这是英特尔系列处理器的典型架构，而 SPARC 或者 Power PC 等芯片的架构是大字节序，即高位数据占据低位内存，低位数据占据高位内存。

5.2　Windbg 调试器和基本指令

5.1 节主要介绍了 Linux 和 macOS 上的调试器 LLDB 的基本用法。在 Windows 平台上，更推荐使用 Windbg 作为应用程序的调试器。Windbg 具有图形界面，使用简单，支持调试命令丰富等特点，是 Windows 平台的首选调试器。

5.2.1　Windbg 简介

Windbg 是 Debugging Tools for Windows 的简称，是微软公司随 Windows SDK 推出的，用于调试 Windows 操作系统上运行的应用程序的调试器。

实际上 Windbg 是一组工具，包括抓取 dump、转储线程对象、符号表管理、堆检查工具等，而 Windbg 作为调试器，是这些工具中最常被使用的。Debugging Tools for Windows

的下载地址为 https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/。

在安装之后，即可使用 Windbg 调试 Windows 平台上的原生代码应用和.NET Core 应用。Windbg 只是一个壳程序，它封装了 cdb 的调试功能，以图形界面的方式向用户提供更好的调试交互功能(cdb 是 Microsoft Console Debugger 的缩写。它与 LLDB 一样，是一款运行在命令行下的调试器)。

5.2.2 Windbg 的启动和退出

Windbg 可以根据实际的使用场景以不同的方式启动。

第一种启动方式是作为 Windows 默认调试器启动。Windows 注册表上可以配置默认调试器，以便在应用程序运行崩溃时及时自动启动。通常情况下，Windows 的默认调试器是华生医生(Dr. Waston)，当应用程序崩溃时，Windows 会根据注册表的配置及时启动华生医生，然后华生医生会为应用程序创建一个迷你内存转储文件。这一切操作都是后台执行的，不易被服务器管理员发现。如果在系统盘上搜索时发现有.dmp 文件，那多半都是由华生医生背后生成的。

用 Windbg 替换默认的调试器，在测试环境中调试应用程序时非常有用。Windbg 的 I 参数会自行向注册表发起修改操作，将默认调试器的位置指向 Windbg 调试器自身。当应用程序运行过程中出现崩溃时，Windows 会自动加载 Windbg 并将 Windbg 调试器附加在崩溃的进程上，调试者直接就可以操作调试指令进行调试。

设置 Windbg 为默认调试器时需要以管理员权限运行，具体如命令 5.8 所示。

```
> Windbg -I
```

命令 5.8 设置 Windbg 为默认调试器

这种 Windbg 启动方式会将崩溃的进程挂起，因此若该功能用在声场环境中，会导致进程监控工具如看门狗等错认为应用程序在正常运行，不会及时干预并恢复服务。因此强烈建议不在生产环境中使用。

第二种启动方式是直接双击 Windbg 图标启动调试器。通过这种方式启动 Windbg 既可以可视化地附加到某个进程上，又可以可视化地加载某个内存转储文件(.dmp 文件)。不过，对于单一的 Windbg 工作区，这两件事一次只能做一件。如果要附加进程调试就不能加载内存转储文件，反之亦然。

用 Windbg 可视化地附加到一个应用程序进程上，可以通过菜单完成操作。单击File→Attach to Process，Windbg 就会显示一个本机进程列表，以供调试者选择，如图 5.13 所示。

当调试者选择某个进程，并单击“确定”按钮之后，Windbg 就会附加到这个进程上，并且中断该进程的运行切换进入调试模式。如果此时并不想立即调试应用程序，可输入 gn 命令让应用程序继续运行。

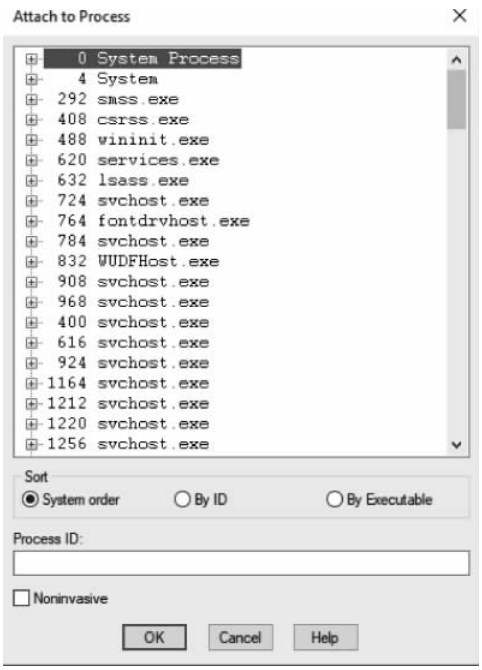

图 5.13　Windbg 选择进程

如果加载内存转储文件，进行事后调试，那么就需要通过点击 File→Open Crash Dump 菜单来实现加载。

第三种启动 Windbg 的方式是通过命令行的方式。以上功能转化为命令行如命令 5.9 所示。

```
> rem 通过命令行附加到指定名字的进程
> windbg -pn <要调试的进程名称>
> rem 通过命令行附加到指定名字的进程 ID
> windbg -p <要调试的进程 ID>
> rem 通过命令行附加到指定服务名称
> windbg -psn <要调试的服务名称>
> rem 通过命令行加载内存转储文件
> windbg -z 要加载的内存转储文件路径
```

命令 5.9　Windbg 启动命令

附加进程启动时，要求命令行窗体一定要具备管理员权限，否则会因为权限不足而附加进程失败。

在 Windbg 启动后，请确认已经正确地配置了符号表服务器，否则会因为部分符号表缺失而给调试增加难度。

5.2.3 Windbg 设置断点

C#设置断点需要使用.NET 调试扩展的!bpmd 命令。

BPMD 命令调试格式如命令 5.10 所示。

```
!BPMD [ - nofuturemodule] < module name > < method name > [< il offset >]
!BPMD < source file name >:< line number >
!BPMD - md < MethodDesc >
!BPMD - list
!BPMD - clear < pending breakpoint number >
!BPMD - clearall
```

命令 5.10 !BPMD 命令

BPMD 提供托管代码的断点调试支持。如果一个方法可以被解析到一个已经加载的加载模块中的函数,BPMD 命令将用 Windbg 命令“bp”创建一个调试断点。如果没有找到,那么意味着包含该方法的模块还没有被加载,或者模块被加载但是功能尚未被激活。在这些情况下,BPMD 要求 Windows 调试器接收 CLR 通知,并等待接收模块负载和 JIT 的消息,此时它将尝试用断点数据解析加载的函数。

以下是 BPMD 的一些使用示例,如命令 5.11 所示。

```
public interface I1                                    //接口声明
{
   void M1();
}
public class ExplicitItfImpl : I1                      //接口实现
{
    …
   void I1.M1()                                        // 'I1.M1'是接口实现方法名
   { …}
}

!bpmd myapp.exe ExplicitItfImpl.I1.M1

public interface IT1 < T >                             //泛型接口
{
   void M1(T t);
}

public class ExplicitItfImpl < U > : IT1 < U >         //泛型接口实现类
{
    …
```

```
        void IT1<U>.M1(U u)                          // 'IT1<U>.M1' 泛型接口函数名
        {… }
}
!bpmd bpmd.exe ExplictItfImpl'1.ITi<U>.M1
```

命令 5.11　BPMD 调试命令

5.2.4　Windbg 查看堆栈调用

Windbg 查看调用堆栈调用的命令是以 k 开头的系列命令。k 命令配合着不同的字母组合可以达到不同的堆栈查看效果。具体如命令 5.12 所示。

```
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr FrameCount
[~Thread] kd [WordCount]
```

命令 5.12　堆栈调试命令

以下是 k 系列命令的主要参数介绍：

b：显示传递给堆栈跟踪中每个函数的前三个参数。

c：显示一个干净的堆栈跟踪。每条显示行只包含模块名称和功能名称。

p：显示堆栈跟踪中调用的每个函数的所有参数。参数列表包括每个参数的数据类型、名称和值。p 选项区分大小写。该参数需要完整的符号信息。

P：显示堆栈跟踪中调用的每个函数的所有参数，如 p 参数。但是，对于 P，功能参数将显示在显示屏的第二行上，而不是与其余数据在同一行上。

v：显示帧指针省略（FPO）信息。在基于 x86 的处理器上，显示屏还包含调用约定信息。

n：显示堆栈调用帧号码。

f：显示相邻帧之间的距离。这个距离是分隔实际堆栈上的帧的字节数。

L：隐藏显示中的源代码行。L 区分大小写。

M：使用调试器标记语言显示输出。输出的每个堆栈调用帧号都是一个链接，可以单击该链接来设置本地上下文并显示本地变量。

以上参数可以一个或者多个进行配合使用，比如，以调用帧的形式显示调用堆栈并带上每个函数的参数数据，就可以使用 kpM 这样的命令来实现，如图 5.14 所示。

5.2.5　Windbg 线程相关指令

Windbg 的线程跳转命令是通过波浪线符号实现的。“～”＋线程号码＋“s”构成线程跳转的方法，s 代表 Set current thread。“～”还与一些特殊符号一起构成常用的线程表达

图 5.14　kpM 命令

式。例如：“～.”表示当前线程；“～＃”表示跳转到当前出现异常的线程；“～＊”表示所有线程。

Windbg 的扩展命令!runaway 提供了线程运行的基本信息。它可以显示几号线程在 CPU 上运行的时长，从而帮助调试者判断线程的繁忙情况，以及占用 CPU 资源最多的线程是哪一个，如命令 5.13 所示。

```
!runaway [标志]
```

命令 5.13　!runaway 命令格式

其中 runaway 的标志位包含以下一个或多个情况：

Bit 0（0x1）：用来指示该命令显示每个线程在 CPU 上的执行时间。

Bit 1（0x2）：用来指示该命令显示每个线程在内核上用于调度的执行时间。

Bit 2（0x4）：用来指示该命令显示线程自创建起所经过的时间。

执行效果如命令 5.14 所示。

```
0:001> !runaway 7

User Mode Time
Thread       Time
0:55c        0:00:00.0093
1:1a4        0:00:00.0000

Kernel Mode Time
Thread       Time
0:55c        0:00:00.0140
1:1a4        0:00:00.0000

Elapsed Time
```

```
Thread      Time
0:55c       0:00:43.0533
1:1a4       0:00:25.0876
```

命令 5.14　!runaway 输出

Windbg 的!thread 命令用来显示线程的摘要信息，包括 ETHREAD 块。语法格式如命令 5.15 所示。

```
!thread [-p] [-t] [地址 [标志]]
```

命令 5.15　!thread 命令

其中-p 参数表示用来显示有关拥有线程的进程的摘要信息。当命令使用-t 参数时，地址是线程 ID，而不是线程地址。

5.2.6　Windbg 寄存器相关指令

Windbg 上查看寄存器的命令非常简单，就是一个字母“r”，代表 register 的意思。命令格式如命令 5.16 所示。

```
[~Thread] r[M 掩码|F|X|?] [ Register[:[Num]Type] [ = [Value]] ]
```

命令 5.16　r 命令

掩码用于指定调试器显示寄存器的数据和类型。“M”必须是大写字母。掩码是指示寄存器显示的一些位的总和。这些位的含义取决于处理器和模式，如果省略 M，则使用默认掩码。

掩码 F 代表显示浮点寄存器，“F”必须是大写字母，该选项相当于 M 0x4；掩码 X 代表显示 SSE XMM 寄存器，该选项相当于 M 0x40；掩码 Y 代表显示 AVX YMM 寄存器，该选项相当于 M 0x200。掩码 YI 表示显示 AVX YMM 整数寄存器，该选项相当于 M 0x400。

一个显示寄存器数据的例子如命令 5.17 所示。

```
0:000> rF
fpcw=027F    fpsw=0000    fptw=0000
st0=        0.00000000000000000000e+0000            st1=
0.00000000000000000000e+0000
st2=        0.00000000000000000000e+0000            st3=
0.00000000000000000000e+0000
st4=        0.00000000000000000000e+0000            st5=
```

```
0.00000000000000000000e+0000
st6=        0.00000000000000000000e+0000              st7=
0.00000000000000000000e+0000
ntdll!NtWaitForMultipleObjects+0x14:
00007ffe59250994 c3              ret
```

命令 5.17　r 命令输出

5.2.7　Windbg 查看内存数据

Windbg 提供一系列的以字母 d 命令开头的内存查看命令，分别用于查看直接内存段、字符串、结构体等内容。具体命令格式如命令 5.18 所示。

```
d{a|b|c|d|D|f|p|q|u|w|W} [Options] [Range]
dy{b|d} [Options] [Range]
d [Options] [Range]
```

命令 5.18　显示内存命令

下面简单介绍 d 系列的几个常用命令：

da：按照给定内存地址按照 ASCII 码格式字符串显示。

dd：按照双字(DWORD)格式显示指定内存地址的内容。

du：按照给定内存地址按照 Unicode 码格式字符串显示。

dW：每条显示行中第一个字的地址和最多八个十六进制字符值。