[用GDB 调试Java程序](http://blog.csdn.net/haoel/article/details/2289305)

标签： [java](http://www.csdn.net/tag/java)[thread](http://www.csdn.net/tag/thread)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[debugging](http://www.csdn.net/tag/debugging)[optimization](http://www.csdn.net/tag/optimization)[compilation](http://www.csdn.net/tag/compilation)

2008-04-13 21:39 22280人阅读 [评论](http://blog.csdn.net/haoel/article/details/2289305#comments)(10) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/haoel/article/details/2289305#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

编程工具（18） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/haoel/article/details/2289305)

**用GDB 调试**[**Java**](http://lib.csdn.net/base/javase)**程序**

**陈皓**

<http://blog.csdn.net/haoel>

**背景**

想要使用GDB调试程序，就需要用GNU的编译器编译程序。如：用GCC编译的C/C++的程序，才能用GDB调试。对于Java程序也是一样的，如果想要用GDB调试，那么就需要用GNU的Java编译器——GCJ来编译Java程序。

目前，很多[**Linux**](http://lib.csdn.net/base/linux)都不会预装Sun的JVM，取而代之是使用GNU的开源编译器来编译和运行Java程序。比如RedHat和Ubuntu，其默认安装都是使用GNU的Java编译器（gcj）和解释器（gij）。当然，它们都被脚本javac和java包装了起来，你一不小心还以为是使用了Sun的JVM。

为什么GNU要搞出一个Java的编译和解释器来呢？其大致有以下几点：

a)      传统的JVM太慢了，因为它解释的是class文件中的bytecode。这种方法实在是太慢了。

b)      为了优化性能，引入了JIT（Just-In-Time），JIT会分析代码，找出那些被反复调用到一定次数的方法和函数，然后直接把这个方法直接处理成汇编machine code，以后就直接运行机器码了。

c)      当然，JIT也有问题，一个是startup overhead，就是说启动的时候有点过分了，表现为时间慢，并且，每次编译后，都需要JIT重新做来过。另一个问题是JIT比较耗费空间。

d)      传统的java还有一个比较扯的问题，就是布署起来太麻烦了，需要有N个jar文件，而不是一个可执行文件。并且，Java需要一个很肥大的运行环境。另外，在java和c/c++之间的调用慢得令人受不了。

**GNU的Java编译器GCJ**

上述的东西是催生出现gcj的原因，GNU用了Ahead-of-Time Compilation来形容GCJ。GNU对GCJ的出现在理由做了下面的说明：

a)      GCC本来可以编译多种程序语言，所以，把java整进来也是一件make sense(合乎逻辑)的事情。

b)      Java的编译是一件非常简单的事情，因为没有C++的模板和预编译器，而且system type, object model 和 exception handling 也很简单。所以，这对于擅长编译技术的GNU来说，从编译方面优化Java的性能是一些很简单的事。

c)       gcj会对java程序做N多的优化工作，比如：common sub-expression elimination, strength reduction, loop optimization 和 register allocation。在优化方面，是GCJ牛还是JIT牛，存在一些较大的争论。对于JIT来说，它可以裁剪和做适时优化，因为是在运行时。 Sun的HotSpot技术是其中比较牛的技术，但gcj的技术也不一定就比JIT差。

d)      对于使用gcj的人来说，最大的一个好处就是startup speed和内存空间使用率。启动JVM或JIT会肖耗很大的内存，例如：NetBean启动就需要74M的内存（什么事也没有干）， JEmacs使用Swing，一启动就是26M，而XEmacs只有8M（这些数据是比较老的了，大约在2003年的数据）。

e)       当GCJ刚出道时，有人比较了Kawa Test Suite在GCJ和JDK1.3.1下的运行比较。结果是，GCJ速度比Sun的JIT快两倍，因为GCJ比Sun的JDK少了一半以上的内存访问未命中的事情，也就是说少了一半的内存换页。并且，实际运行过程中，也少了25%的内存使用。

f)        最后，GCJ用的是一个so的库来做编译，他可以把.java的程序直接编译成.o文件和可执行文件。并且用gdb调试。

本文主要讲述如果使用GDB调试Java程序。关于GDB的使用，请参看我的另一篇文章《[用GDB调试程序](http://blog.csdn.net/haoel/archive/2003/07/02/2879.aspx)》。

**用GCJ编译Java程序**

用GCJ编译Java程序很简单，关于编译成.o和执成文件，如下所示：

gcj -c -g -O MyJavaProg.java  
gcj -g --main=MyJavaProg -o MyJavaProg MyJavaProg.o

很明显，基本上就是gcc的语法。当然，你也可以一步编译出可执行文件：

            gcj -g --main=MyJavaProg -o MyJavaProg MyJavaProg.java

其中，使用-g参数表示加入调试信息，这对于调试时相当重要。不然，无法看到实际的源码和函数。而关于--main参数，意思是指定main函数所在的Java类。

如果你需要使用makefile，想使用类似于CFLAGS这样的变量，我们可以使用GCJFLAGS这个变量名。

**使用GDB调试Java程序**

如同我的《[用GDB调试程序](http://blog.csdn.net/haoel/archive/2003/07/02/2879.aspx)》一文，我使用如下的Java程序作为演示程序。

**1 public class sum{**

**2    public static long Sum(int n){**

**3        long result=0, i;**

**4        for(i=0; i<n; i++){**

**5            result += i;**

**6        }**

**7        return result;**

**8    }**

**9**

**10**

**11     public static final void main( String argc[] ) {**

**12         int i;**

**13         int result=0;**

**14         for (i=1; i<=100; i++){**

**15             result += i;**

**16         }**

**17         System.out.println("result = "+result);**

**18         System.out.println("result = "+Sum(1000));**

**19     }**

**20 }**

下面是程序编译：（注意-g选项）

hchen@ubuntu:~/java$ **gcj --main=sum -g -o sum sum.java**

进入GDB环境：

hchen@ubuntu:~/java$ **gdb ./sum**

GNU gdb 6.6-debian

Copyright (C) 2006 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i486-linux-gnu"...

Using host libthread\_db library "/lib/tls/i686/cmov/libthread\_db.so.1".

**(gdb)**

你可能在直接使用函数名会有以下问题：

(gdb) **break sum.main()**

Function "sum.main()" not defined.

Make breakpoint pending on future shared library load? (y or [n]) n

目前我不知道是否是GDB的bug，不过Workaround的解决方案如下：

1）先List类的构造函数，这样可以找到源文件。

2）使用源文件的行号进行break。

(gdb**) l sum::sum()**

1

2       public class sum{

3          public static long Sum(int n){

4              long result=0, i;

5              for(i=0; i<n; i++){

6                  result += i;

7              }

8              return result;

9          }

10

(gdb) **l**

11

12          public static final void main( String argc[] ) {

13              int i;

14              int result=0;

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

16                  result += i;

17              }

18              System.out.println("result = "+result);

19              System.out.println("result = "+Sum(1000));

20          }

(gdb) **break 13**

Breakpoint 1 at 0x8048d38: file sum.java, line 13.

(gdb) **break 16 if i==50**

Breakpoint 2 at 0x8048d61: file sum.java, line 16.

运行并调式程序：

对于下面出现在GDB命令我不在作过多解释，请参看我的《[用GDB调试程序](http://blog.csdn.net/haoel/archive/2003/07/02/2879.aspx)》

**(gdb) r**

Starting program: /home/hchen/java/sum

[Thread debugging using libthread\_db enabled]

[New Thread -1243736400 (LWP 18131)]

[New Thread -1245406320 (LWP 18134)]

[Switching to Thread -1243736400 (LWP 18131)]

Breakpoint 1, sum.main(java.lang.String[])void (argc=@2bfa8) at sum.java:14

14              int result=0;

Current language:  auto; currently java

**(gdb) break sum.Sum               <-----  设置函数断点**

Breakpoint 3 at 0x8048b68: file sum.java, line 4.

**(gdb) c**

Continuing.

Breakpoint 2, sum.main(java.lang.String[])void (argc=@2bfa8) at sum.java:16  **<---条件断点**

16                  result += i;

**(gdb) p result**

$2 = 1225

(gdb) **n**

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

**(gdb) c**

Continuing.

result = 5050

Breakpoint 3, sum.Sum(int)long (n=1000) at sum.java:4  **<-----  函数断点**

4              long result=0, i;

**(gdb) bt                       <-----  打出函数栈**

#0  sum.Sum(int)long (n=1000) at sum.java:4

#1  0x08048edf in sum.main(java.lang.String[])void (argc=@2bfa8) at sum.java:19

#2  0xb6b17611 in gnu::java::lang::MainThread::call\_main () from /usr/lib/libgcj.so.81

#3  0xb6b86797 in gnu::java::lang::MainThread::run () from /usr/lib/libgcj.so.81

#4  0xb6b29cf3 in \_Jv\_ThreadRun () from /usr/lib/libgcj.so.81

#5  0xb6ad77dd in \_Jv\_RunMain () from /usr/lib/libgcj.so.81

#6  0xb6ad7994 in \_Jv\_RunMain () from /usr/lib/libgcj.so.81

#7  0xb6ad7a1b in JvRunMain () from /usr/lib/libgcj.so.81

#8  0x08048b38 in main (argc=Cannot access memory at address 0x0) at /tmp/ccKMKFB0.i:11

**(gdb) n**

5              for(i=0; i<n; i++){

**(gdb) n**

6                  result += i;

**(gdb) n**

5              for(i=0; i<n; i++){

**(gdb) finish                   <----- 退出函数**

Run till exit from #0  sum.Sum(int)long (n=1000) at sum.java:5

0x08048edf in sum.main(java.lang.String[])void (argc=@2bfa8) at sum.java:19

19              System.out.println("result = "+Sum(1000));

Value returned is $1 = 499500

**(gdb) n**

result = 499500

0xb6b17611 in gnu::java::lang::MainThread::call\_main () from /usr/lib/libgcj.so.81

**(gdb) info thread                   <-----  查看线程**

  2 Thread -1245553776 (LWP 18143)  0xffffe410 in \_\_kernel\_vsyscall ()

\* 1 Thread -1243883856 (LWP 18142)  0xb6b17611 in gnu::java::lang::MainThread::call\_main () from /usr/lib/libgcj.so.81

(gdb)

**其它注意事项**

当你使用GDB调试被GCJ编译的程序时，你需要让GDB忽略SIGPWR和SIGCPU这两个信号。这两个信号被垃圾回收器使用，为了让调试工作进行的更顺利，我们需要使用GDB的命令来忽略这两个信号：

**(gdb) handle SIGPWR nostop noprint**

Signal        Stop      Print   Pass to program Description

SIGPWR        No        No      Yes             Power fail/restart

**(gdb) handle SIGXCPU nostop noprint**

Signal        Stop      Print   Pass to program Description

SIGXCPU       No        No      Yes             CPU time limit exceeded

当然，你并不用每次都需要设置这两个命令，你可以设置$HOME目录下的.gdbinit文件来把这两个命令作为GDB的初始化选项。