Oracle® Developer Studio 12.5: dbxtool 教程

2016 年 6 月

文件号码 E71968

- “简介” [2]
- “程序示例” [2]
- “配置 dbxtool” [3]
- “诊断信息转储” [9]
- “使用断点和步进” [13]
- “使用高级断点技巧” [21]
- “使用断点脚本修补代码” [38]

简介

本教程使用一个“有错误”的示例程序来演示如何有效地使用 dbxtool。dbxtool 是 dbx 调试器的独立图形用户界面 (graphical user interface, GUI)。首先从基础功能开始，然后是较为高级的功能。

程序示例

本教程模拟了一个简化的、有些虚拟的 dbx 调试器。可从 Oracle Developer Studio 12.5 下载网页上的示例应用程序 zip 文件中下载此 C++ 程序的源代码，网址为：http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html。

接受许可并下载后，可以将 zip 文件提取到所选择的目录中。

1. 如果您尚未执行此操作，请下载样例应用程序 zip 文件，然后将该文件解压缩到您选择的某个位置。debug_tutorial 应用程序位于 OracleDeveloperStudio12.5-Samples 目录的 Debugger 子目录中。
2. 生成程序。

```
$ make
CC -g   -c  main.cc
CC -g   -c  interp.cc
CC -g   -c  cmd.cc
CC -g   -c  debugger.cc
CC -g   -c  cmds.cc
CC -g   main.o interp.o cmd.o debugger.o cmds.o -o a.out
```

程序由下列模块组成：

cmd.h	cmd.cc	类 Cmd，用于实现调试器命令的基类
interp.h	interp.cc	类 Interp，一个简单的命令解释程序
debugger.h	debugger.cc	类 Debugger，用于模拟调试器的主要语义
cmds.h	cmds.cc	各种调试命令的实现
main.h	main.cc	main（） 函数和出错处理。设置 Interp，创建各种命令并将其分配给 Interp。运行 Interp。

运行程序并尝试几个 dbx 命令。

```
$ a.out
> display var
will display 'var'
```

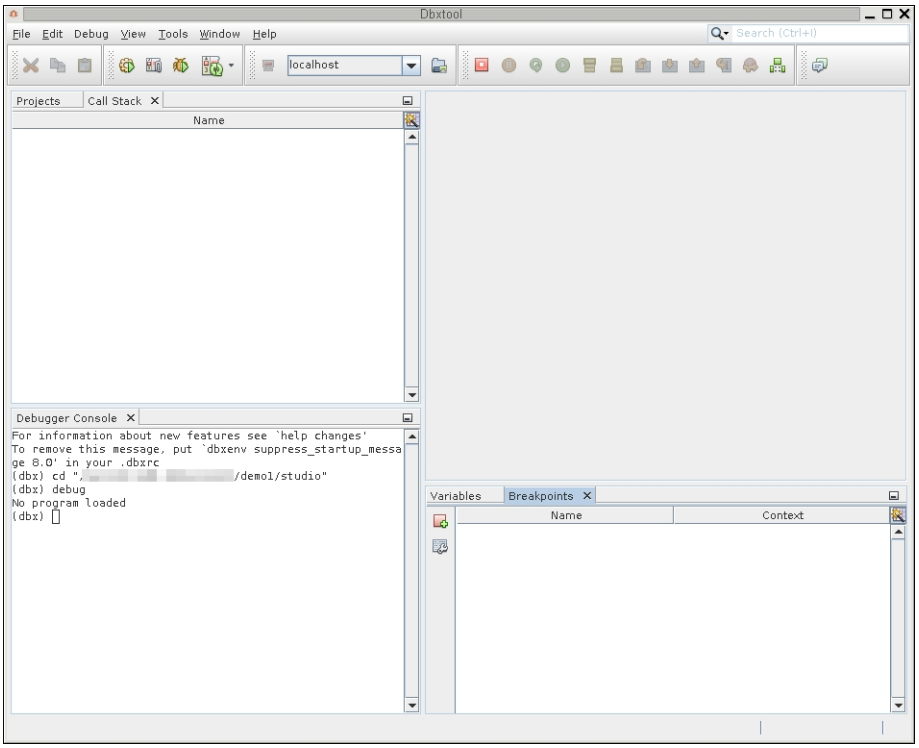
```
> stop in X
> run running ...
stopped in X
var = {
        a = '100'
        b = '101'
        c = '<error>'
        d = '102'
        e = '103'
        f = '104'
}
> quit
Goodby
$
```

配置 dbxtool

通过键入以下内容启动 dbxtool：

install-dir/bin/dbxtool

首次启动 dbxtool 时，窗口的外观可能如下所示：



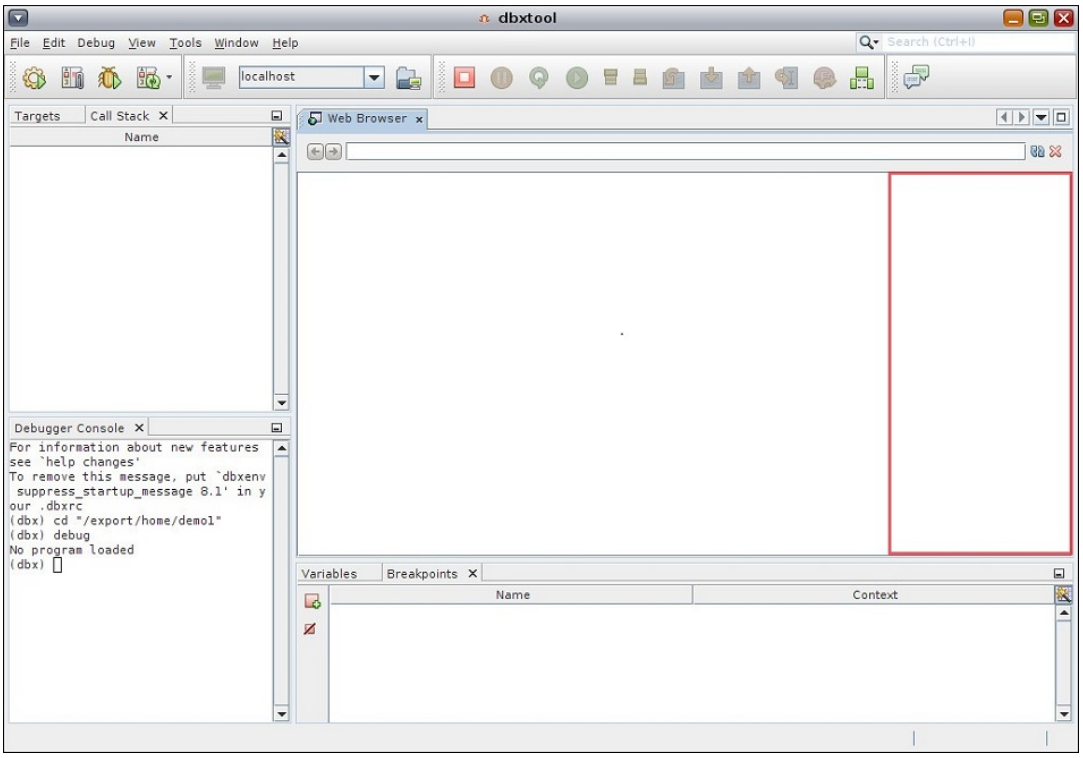
注 - 本教程中的图可能与您使用 dbxtool 时看到的不同。

如果您需要更多空间来容纳其他应用程序（如 Web 浏览器），您可能需要定制 dbxtool 以占据较少的空间。

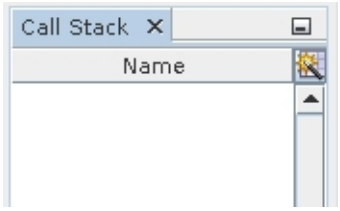
下面是定制 dbxtool 的各种方法的示例。

- 使工具栏图标变小。
  - 右键单击工具栏中的任意位置并选择 "Small Toolbar Icons"（小工具栏图标）。
- 将 "Call Stack"（调用堆栈）窗口移开。

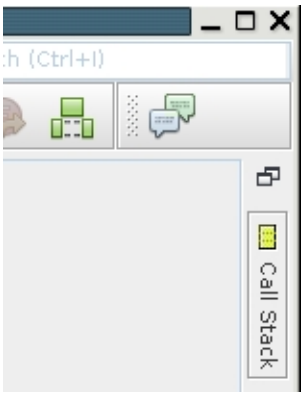
1. 单击 "Call Stack"（调用堆栈）窗口的标题，并向下向右拖动该窗口。当红色轮廓位于下图所示的位置时释放：



2. 单击 "Call Stack"（调用堆栈）窗口标题中的最小化按钮。

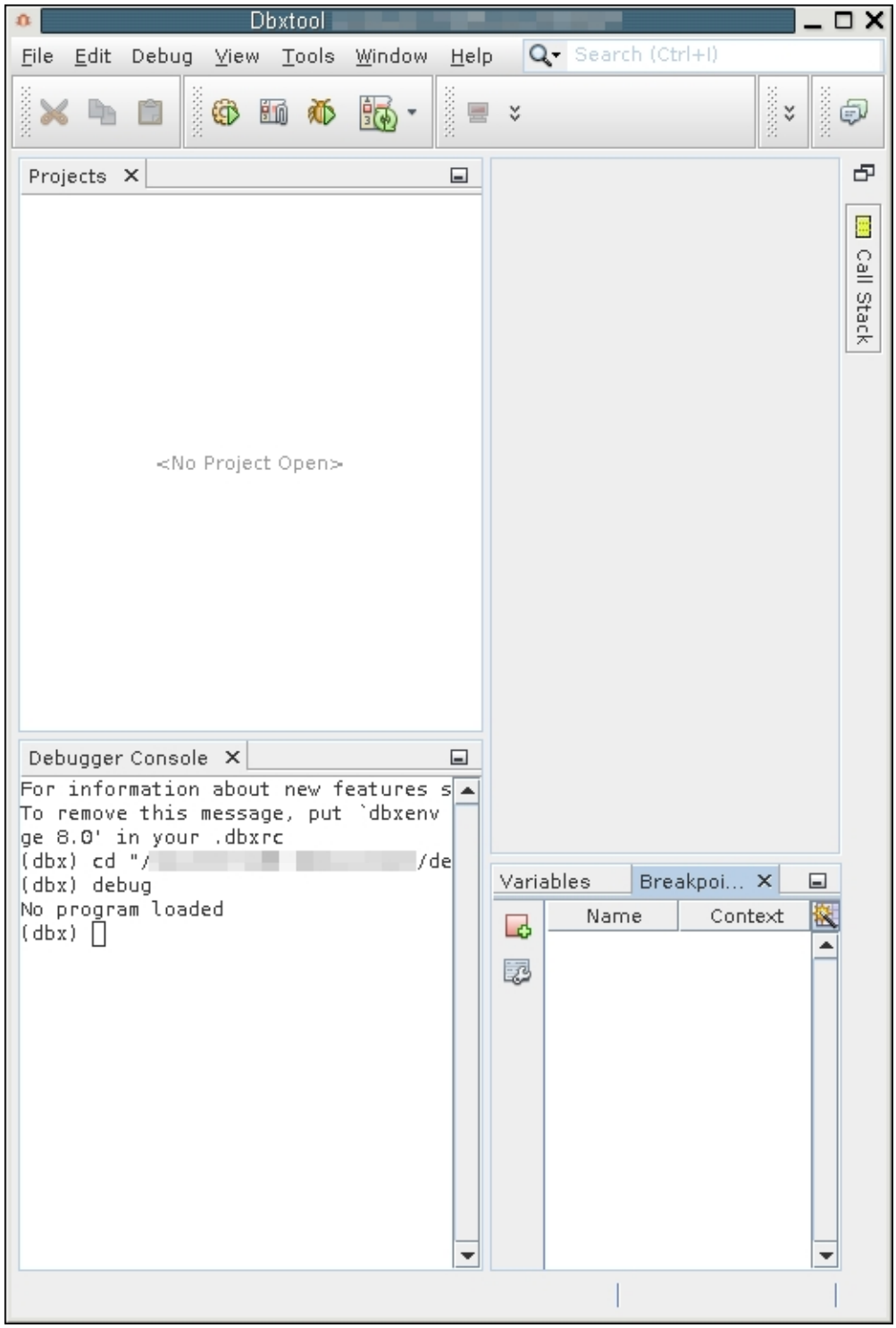


"Call Stack"（调用堆栈）窗口会在右边界中最小化。



如果将光标悬停在最小化的 "Call Stack"（调用堆栈）图标上方，"Call Stack"（调用堆栈）窗口会最大化，直到将焦点转移到另一个窗口。如果单击最小化的 "Call Stack"（调用堆栈）图标，"Call Stack"（调用堆栈）窗口会最大化，直到再次单击该图标。

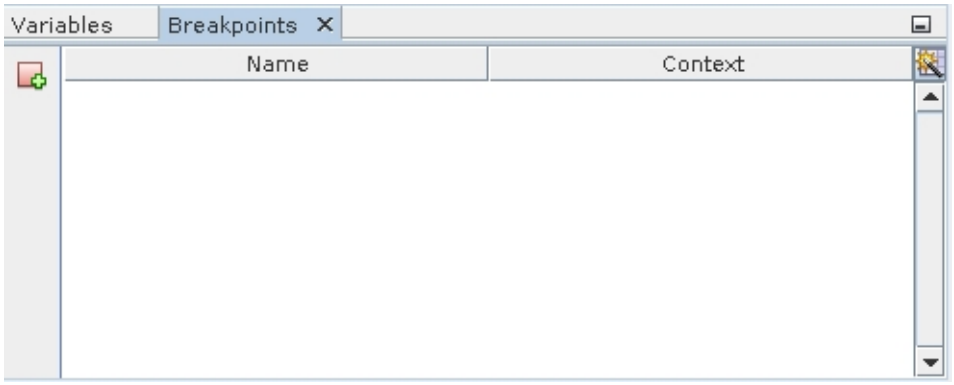
3. 将主窗口的宽度缩小至半个屏幕：



- **最小化窗口分组：**

  dbxtool 可以将多个窗口归为一组。除了单个窗口外，您还可以对窗口组执行操作。对于每个窗口所属的组，您可以将其最小化/还原、拖动到新位置、浮动在单独的窗口中或者停靠回 IDE 窗口。

例如，如果单击 "Breakpoints"（断点）选项卡，然后单击最小化窗口按钮，整个窗口组都会最小化。



- **取消停靠 "Output"（输出）窗口，以便可以轻松与您正在调试的程序的输入和输出进行交互，同时可以轻松访问 dbxtool 窗口中的其他选项卡。**
  - 如果您未看到 "Output"（输出）窗口，请单击 "Window"（窗口）> "Output"（输出）或按 Ctrl - 4 组合键。
  - 单击并按住 "Output"（输出）窗口的标题，将该窗口拖到 dbxtool 窗口之外，然后放到桌面上。

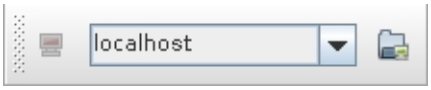
    要在 dbxtool 窗口中重新停靠 "Output"（输出）窗口，请在 "Output"（输出）窗口中右键单击，然后选择 "Dock Group"（停靠组）。
- **在编辑器中设置字体大小。**在编辑器窗口中显示一些源代码后，执行以下操作以设置字体大小：

  1. 选择 "Tools"（工具）> "Options"（选项）。
  2. 在 "Options"（选项）窗口中，选择 "Fonts & Colors"（字体和颜色）类别。
  3. 在 "Syntax"（语法）选项卡上，确保从 "Languages"（语言）下拉式列表中选择 "All Languages"（所有语言）。
  4. 单击 "Font"（字体）文本框旁边的 "Browse"（浏览）(...) 按钮。
  5. 在 "Font Chooser"（字体选择器）对话框中，设置字体、样式和大小，然后单击 "OK"（确定）。
  6. 在 "Options"（选项）窗口中单击 "OK"（确定）。
- **在终端窗口中设置字体大小。**"Debugger Console"（调试器控制台）和 "Output"（输出）窗口是 ANSI 终端仿真器。

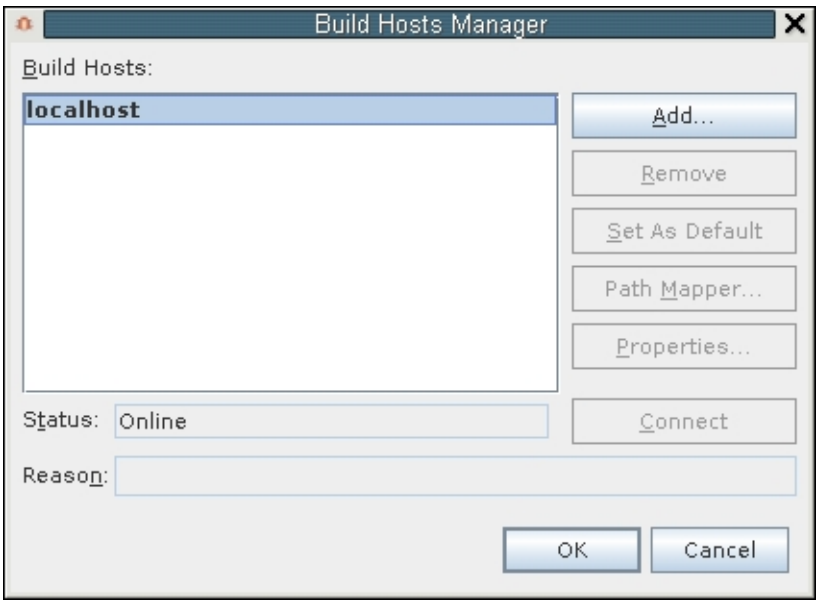
  1. 选择 "Tools"（工具）> "Options"（选项）。
  2. 在 "Options"（选项）窗口中，选择 "Miscellaneous"（其他）类别。
  3. 单击 "Terminal"（终端）选项卡。
  4. 选择 "Font Size"（字体大小）和 "Click To Type"（单击以键入）等设置。
  5. 单击 "OK"（确定）。
- **添加要运行调试器的远程主机。**dbxtool 支持访问要运行 dbxtool 的远程服务器以及访问远程文件。

  向 dbxtool 添加远程主机：

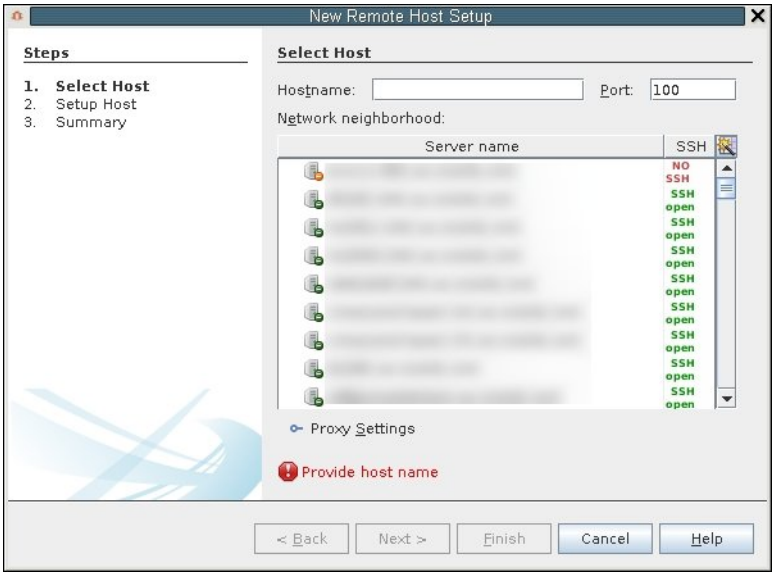
  1. 在 "Remote"（远程）工具栏中，单击主机下拉式列表的向下箭头并选择 "Manage Hosts"（管理主机）。



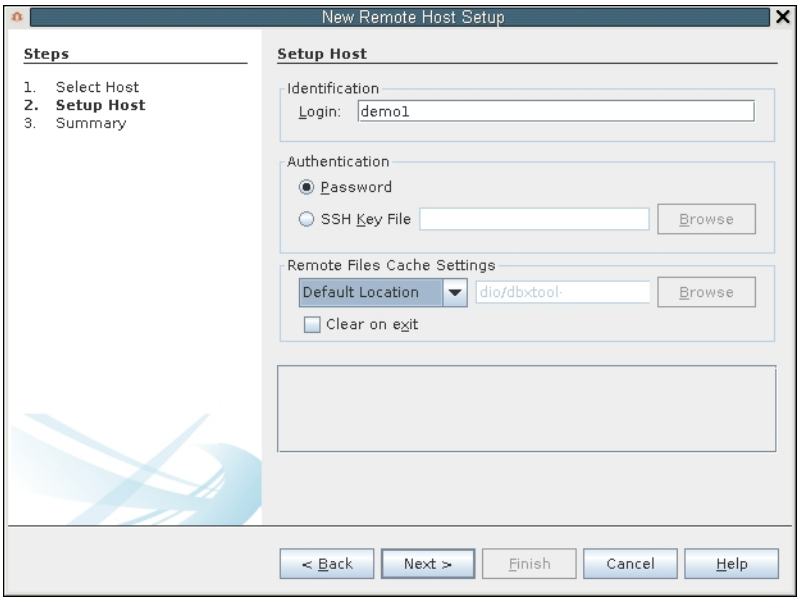
2. "Build Hosts Manager"（生成主机管理器）将打开。单击 "Add"（添加）以添加新服务器。



3. 在 "New Remote Host Setup"（新建远程主机设置）向导中，从 "Network neighborhood"（网上邻居）列表中选择一个可用的服务器并单击 "Next"（下一步）。



4. 输入登录信息，选择验证方法，然后单击 "Next"（下一步）。如果选择了 "Password"（口令），请在提示时输入口令。



5. 连接主机后，摘要页面中会显示连接状态。可以在工作时从 "Remote"（远程）工具栏选择该远程主机。

   有关远程主机的更多信息，请参见 dbxtool 中的联机帮助（在 "Remote Debugging"（远程调试）主题下）。

完成定制后，退出 dbxtool。下次运行 dbxtool 时，dbxtool 会记住您的首选项。

诊断信息转储

要查找错误，请再次运行示例程序，然后在不输入命令的情况下按回车键。

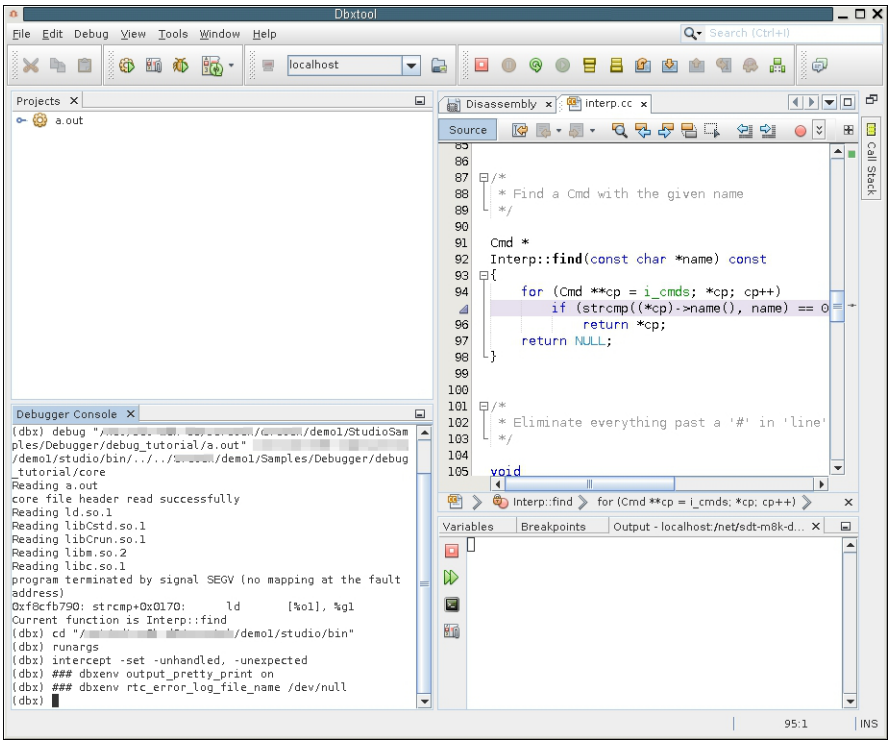
```
$ a.out
> display var
will display 'var'
>
Segmentation Fault (core dumped)
$
```

启动包含可执行文件和信息转储文件的 `dbxtool`。

```
$ dbxtool a.out core
```

请注意，`dbxtool` 命令接受的参数与 `dbx` 命令相同。

`dbxtool` 显示与以下示例类似的输出。

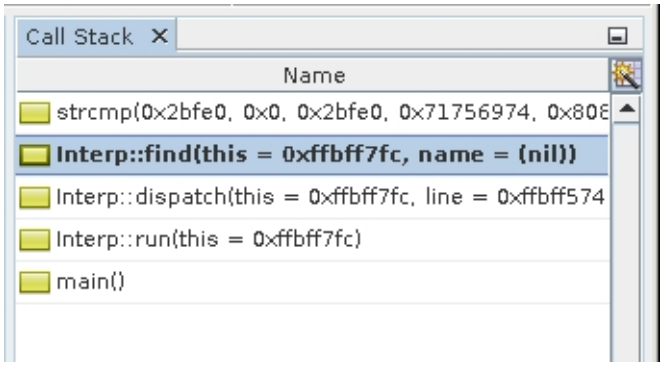


请注意以下几点：

- 在 "Debugger Console"（调试器控制台）窗口中，您将看到一条类似于以下示例的消息：

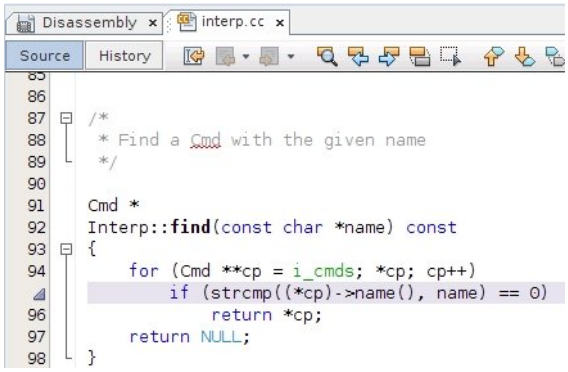
```
program terminated by signal SEGV (no mapping at fault address)
0xf8cfb790: strcmp+0x0170:      ld       [%o1], %gl
Current function is Interp::find
```

- 即使 `strcmp（）` 函数中发生了 `SEGV`，`dbx` 也会自动在第一个帧中显示包含调试信息的函数。请注意，"Call Stack"（调用堆栈）窗口中的堆栈跟踪在当前帧的图标四周有一个边框。



请注意，"Call Stack"（调用堆栈）窗口显示参数名称和值。在该示例中，传递给 `strcmp（）` 的第二个参数为 0x0，`name` 的值为 NULL。

- 在编辑器窗口中，第 95 行上的淡紫色长条和一个三角形（而不是绿色条和箭头）表示调用 `strcmp（）` 的位置，而不是错误的实际位置：

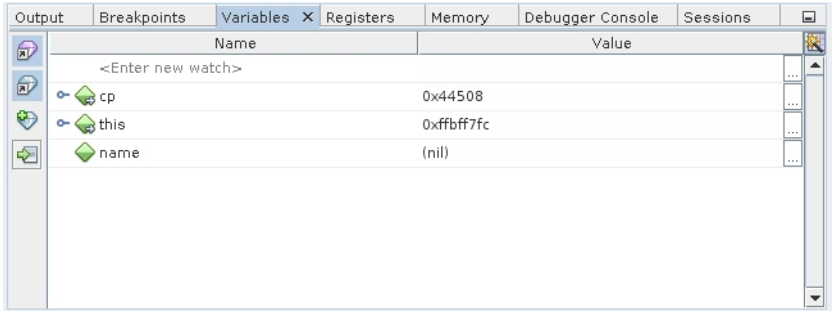


如果您没有看到参数值，请检查 `.dbxrc` 文件中的 `dbxenv` 变量 `stack_verbose` 是否设置为 `on`。您还可以在 "Call Stack"（调用堆栈）窗口中设置详细模式，方法是在该窗口中右键单击并选择 "Verbose"（详细）选项。有关 `dbxenv` 变量和 `.dbxrc` 的更多信息，请参见《Oracle Developer Studio 12.5：使用 dbx 调试程序》 中的 第 3 章, “定制 dbx”。

向函数传递错误值作为参数时，函数通常会失败。检查传递给 `strcmp（）` 的值：

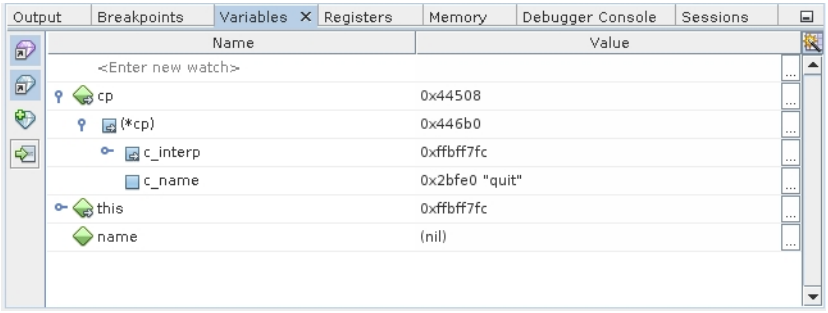
- "Variables"（变量）窗口自动显示所有本地变量。检查 "Variables"（变量）窗口中参数的值。

  1. 单击 "Variables"（变量）选项卡。



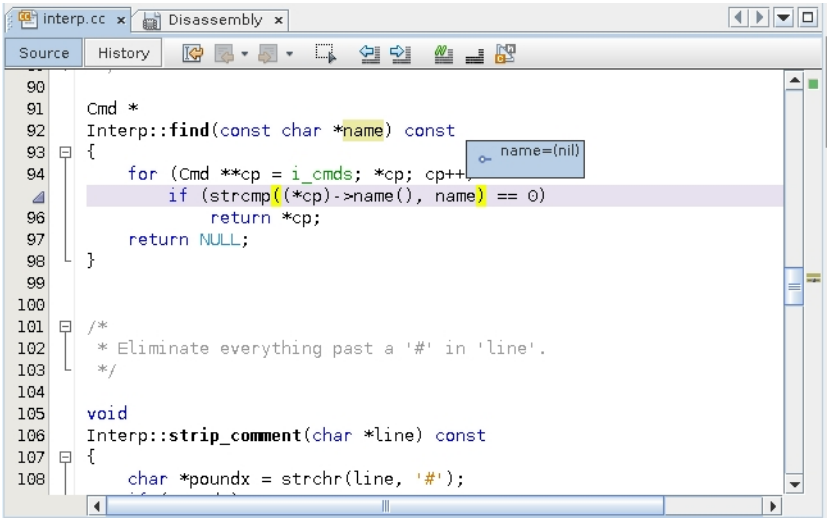
请注意，`name` 的值为 NULL。该值很可能是导致 SEGV 的原因，但我们还是来检查一下另一个参数 `(*cp)->name()` 的值。

  2. 在 "Variables"（变量）窗口中，展开 `cp` 节点，然后展开 `(cp*)` 节点。此处所涉及的 `name` 为 "quit"，该名称是有效的：



如果展开 *cp 节点后未显示其他变量，请检查 .dbxrc 文件中的 dbx 环境变量 output_inherited_members 是否设置为 on。通过在窗口中右键单击并选中 "Inherited Members"（继承的成员）复选框来添加复选标记，您还可以显示继承的成员。

- 使用气球表达式求值确认参数的值。单击进入编辑器窗口，然后将光标悬停在传递至 strcmp() 的 name 变量上方。将显示一个提示，其中，name 的值显示为 NULL。

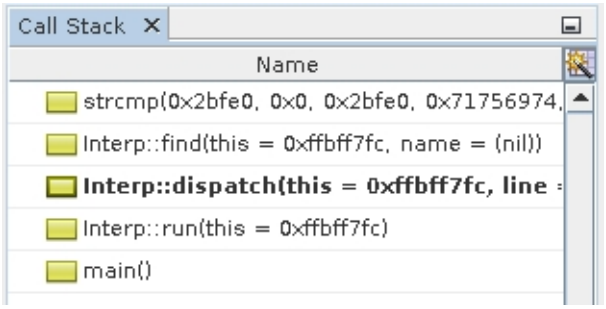


使用气球表达式求值时，您还可以将光标放置到一个表达式的上方，例如 (*cp)->name()。但是，对于包含函数调用的表达式，会禁用气球表达式求值，因为：

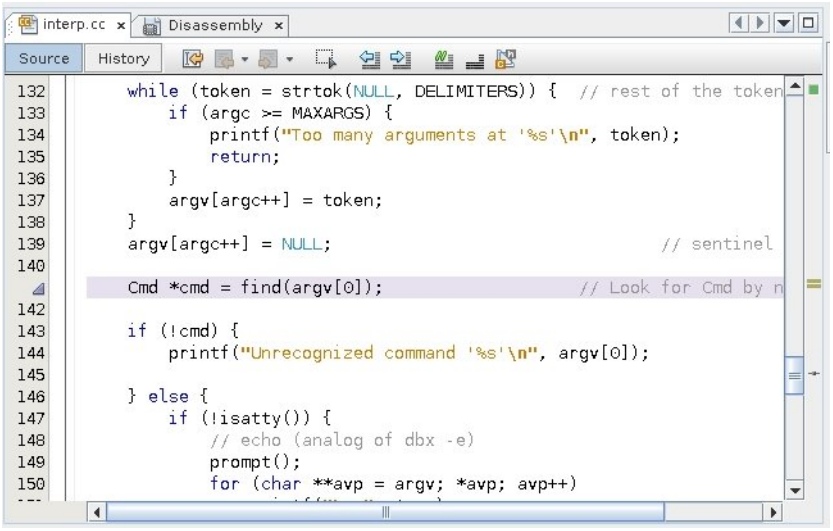
- 您正在调试一个信息转储文件。
- 因为在编辑器窗口中随意悬停光标，函数调用可能具有副作用。

由于 name 的值不应为 NULL，因此您需要找出是哪部分代码将该错误的值传递至 Interp::find()。要找出：

1. 通过选择 "Debug"（调试）> "Stack"（堆栈）> "Make Caller Current"（使调用方成为当前调用方）或单击工具栏上的 "Make Caller Current"（使调用方成为当前调用方）按钮（Alt - Page Down 组合键）  ，上移调用堆栈。



2. 在 "Call Stack"（调用堆栈）窗口中，双击与 `Interp::dispatch()` 相对应的帧。
   现在编辑器窗口中突出显示了相应的代码：



   该代码是未知代码，除 `argv[0]` 的值为 NULL 外不提供任何线索。

通过动态使用断点和单步执行，调试该问题可能会更加容易。

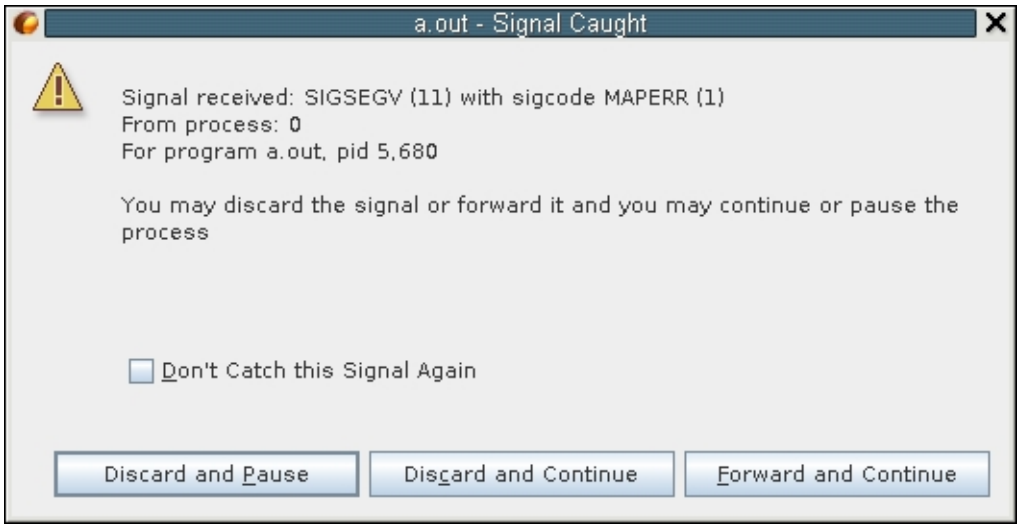
使用断点和步进

使用断点可以将程序停止在错误位置之前，并单步执行代码以查找何处出错。

如果您尚未取消停靠 "Output"（输出）窗口，请执行该操作。

您早先是从命令行运行的程序。通过在 `dbxtool` 中运行程序来重现错误。

1. 在工具栏中单击 "Restart"（重新启动）按钮  或在 "Debugger Console"（调试器控制台）窗口中键入 `run`。
2. 在 "Debugger Console"（调试器控制台）窗口中按回车键。
   警报框会提供有关 SEGV 的信息。



3. 在警报框中，单击 "Discard and Pause"（放弃并暂停）。

   "Editor"（编辑器）窗口会再次在 `Interp::find（）` 中突出显示对 `strcmp（）` 的调用。

4. 在工具栏中单击 "Make Caller Current"（使调用方成为当前调用方）按钮  以转至您先前在 `Interp::dispatch（）` 中看到的未知代码。

5. 在下一节，您将在调用 `find（）` 之前一个位的位置设置断点，这样您就可以单步执行代码以了解为何出错。

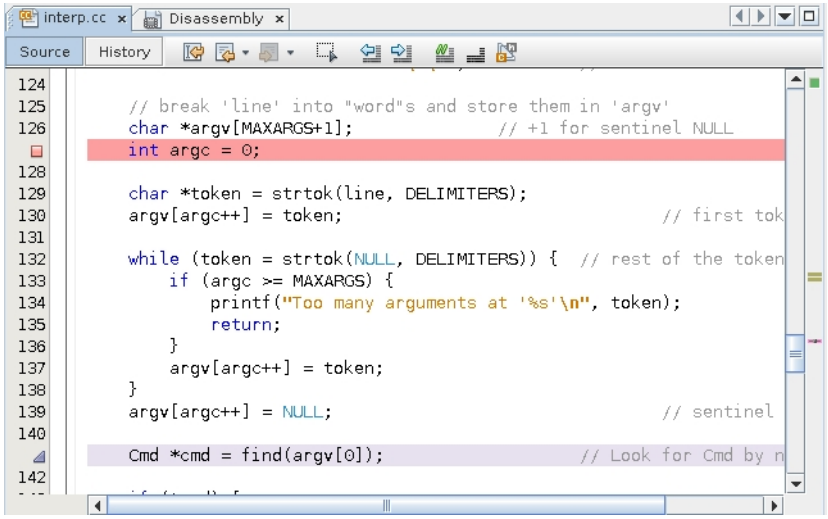
设置断点

可以通过多种方式设置断点，如行断点或函数断点。以下列表介绍了创建断点的多种方式。

注 - 如果未显示行号，请通过在左边界中右键单击并选择 "Show Line Numbers"（显示行号）选项，在编辑器中启用行号。

- **设置行断点**

  通过在第 127 行旁边的左边界中单击，开启/关闭行断点。

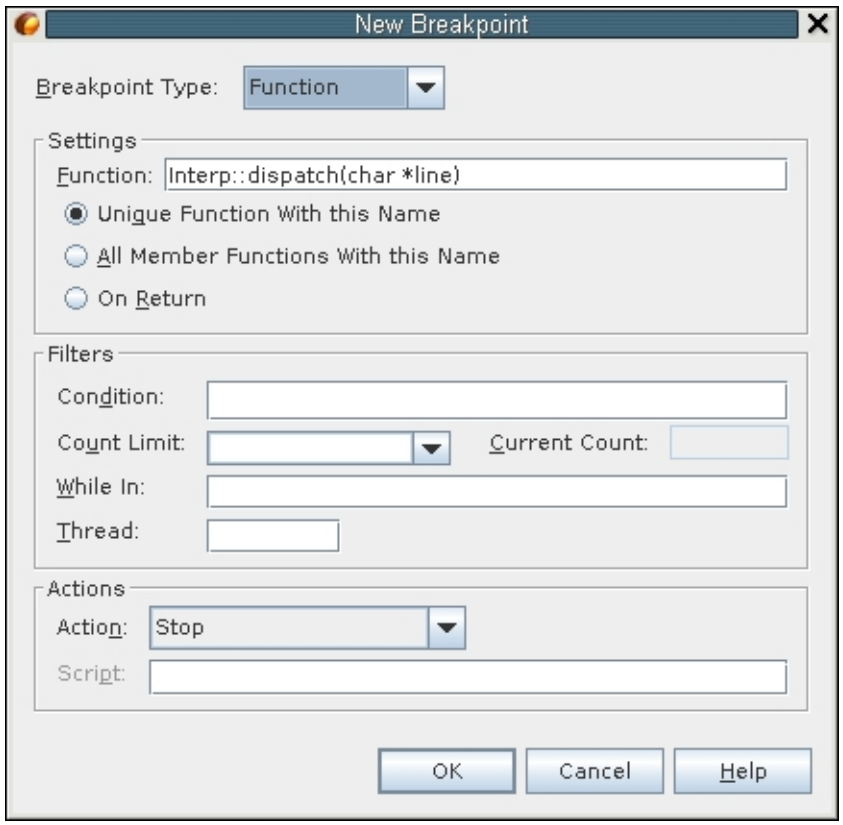


- **设置函数断点**

  设置函数断点

  1. 在编辑器窗口中选择 `Interp::dispatch`。
  2. 选择 "Debug"（调试）> "New Breakpoint"（新建断点）或右键单击并选择 "New Breakpoint"（新建断点）。

     将显示 "New Breakpoint"（新建断点）对话框。



请注意，"Function"（函数）字段随选定的函数名称进行填充。

3. 单击 "OK"（确定）。

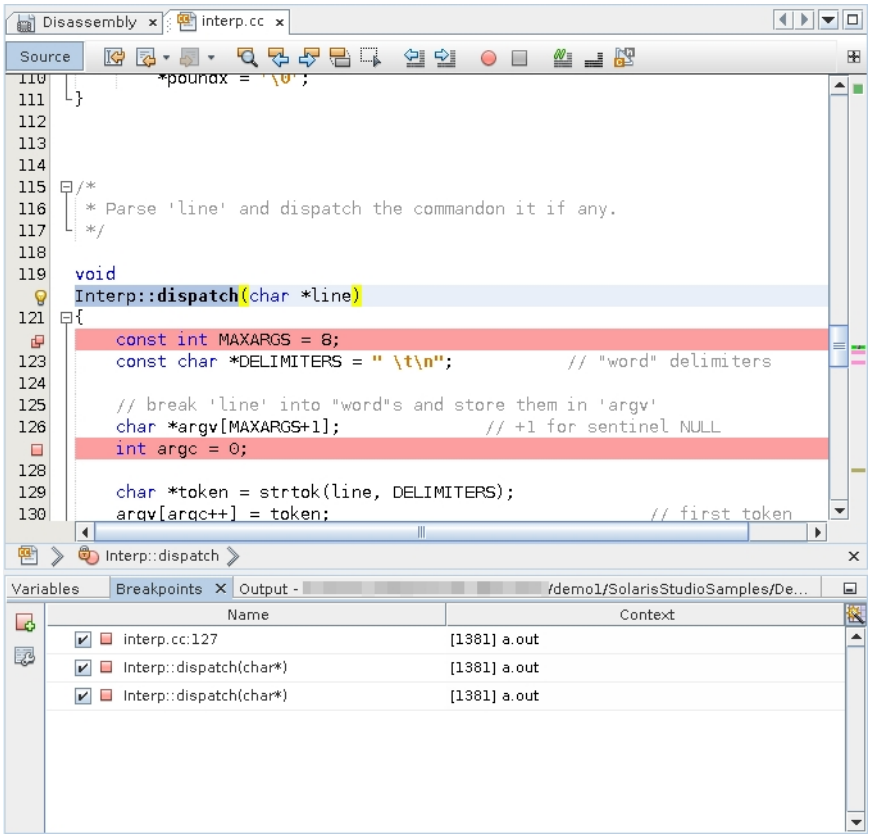
- **从命令行设置断点**

  设置函数断点最容易的方法是从 `dbx` 命令行进行设置。在 "Debugger Console"（调试器控制台）窗口中键入 `stop in` 命令：

```
(dbx) stop in dispatch
(4) stop in Interp::dispatch(char*)
(dbx)
```

  请注意，不必键入 `Interp::dispatch`。只需键入函数名称即可。

断点窗口和编辑器的外观可能如下所示：



要避免编辑器中变得杂乱，请使用 "Breakpoints"（断点）窗口。

1. 单击 "Breakpoints"（断点）选项卡（如果先前最小化了该选项卡，请先将其最大化）。
2. 选择行断点和其中一个函数断点，右键单击，然后选择 "Delete"（删除）。

有关断点的更多信息，请参见《Oracle Developer Studio 12.5：使用 dbx 调试程序》 中的 第 6 章, “设置断点和跟踪”。

函数断点的优点

通过在编辑器中切换来设置行断点可能比较直观。但是，许多 dbx 用户喜欢使用函数断点是由于以下原因：

- 在 "Debugger Console"（调试器控制台）窗口中键入 `si dispatch` 意味着您不必在编辑器中打开文件并滚动到某行设置断点。
- 由于您可以通过在编辑器中选择任何文本来创建函数断点，因此可以从函数的调用点在此函数上设置断点，而不必打开文件。

提示 - `si` 是 `stop in` 的别名。大多数 dbx 用户都会定义许多别名，并将这些别名放置在 dbx 配置文件 `~/.dbxrc` 中。一些常见的示例有：

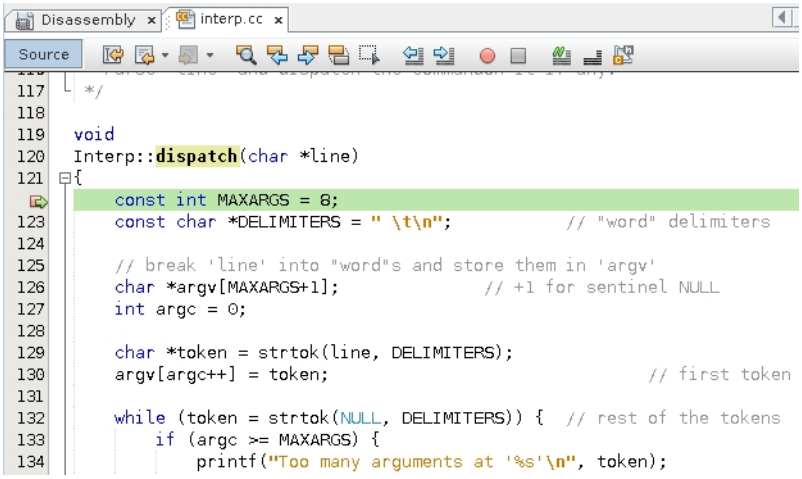
```
alias si stop in
alias sa stop at
alias s step
alias n next
alias r run
```

有关定制 `.dbxrc` 文件和 dbxenv 变量的更多信息，请参见《Oracle Developer Studio 12.5：使用 dbx 调试程序》 中的 “设置 dbxenv 变量”。

- 在 "Breakpoints"（断点）窗口中，函数断点的名称是描述性的。行断点的名称不是描述性的，但可以通过在 "Breakpoints"（断点）窗口中右键单击行断点并选择 "Go To Source"（转至源）或双击断点来了解第 127 行中的内容。
- 函数断点更为稳定。由于 `dbxtool` 会保留断点，因此，如果您编辑了代码或执行了源代码控制合并，行断点可能容易发生偏移。而函数名称则不太容易受到编辑操作的影响。

使用监视和步进

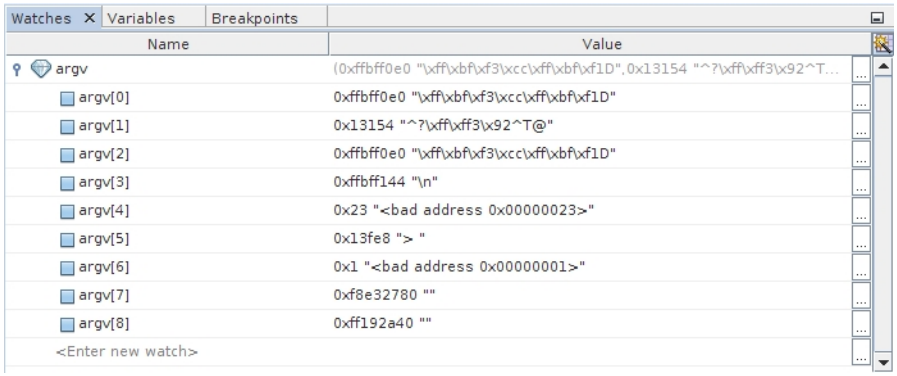
既然您在 `Interp::dispatch()` 处设置了单个断点，那么如果您再次单击 "Restart"（重新启动）  并在 "Debugger Console"（调试器控制台）窗口中按回车键，程序会在 `dispatch()` 函数包含可执行代码的第一行停止。



由于您已经确定了传递给 `find()` 的 `argv[0]` 问题，因此可以对 `argv` 使用监视：

1. 在编辑器窗口中选择 `argv` 的一个实例。
2. 右键单击并选择 "New Watch"（新建监视）。"New Watch"（新建监视）对话框将打开，并随选定的文本进行填充：

3. 单击 "OK"（确定）。
4. 要打开 "Watches"（监视）窗口，请选择 "Window"（窗口）> "Watches"（监视）（Alt + Shift + 2 组合键）。
5. 在 "Watches"（监视）窗口中，展开 `argv`。



请注意，`argv` 没有初始化，而且由于 `argv` 是局部变量，因此可能会从先前的调用中“继承”堆栈上留下的随机值。这会不会是问题的原因？

注 - 可在 "Variables"（变量）窗口以及 "Watches"（监视）窗口中查看监视变量。

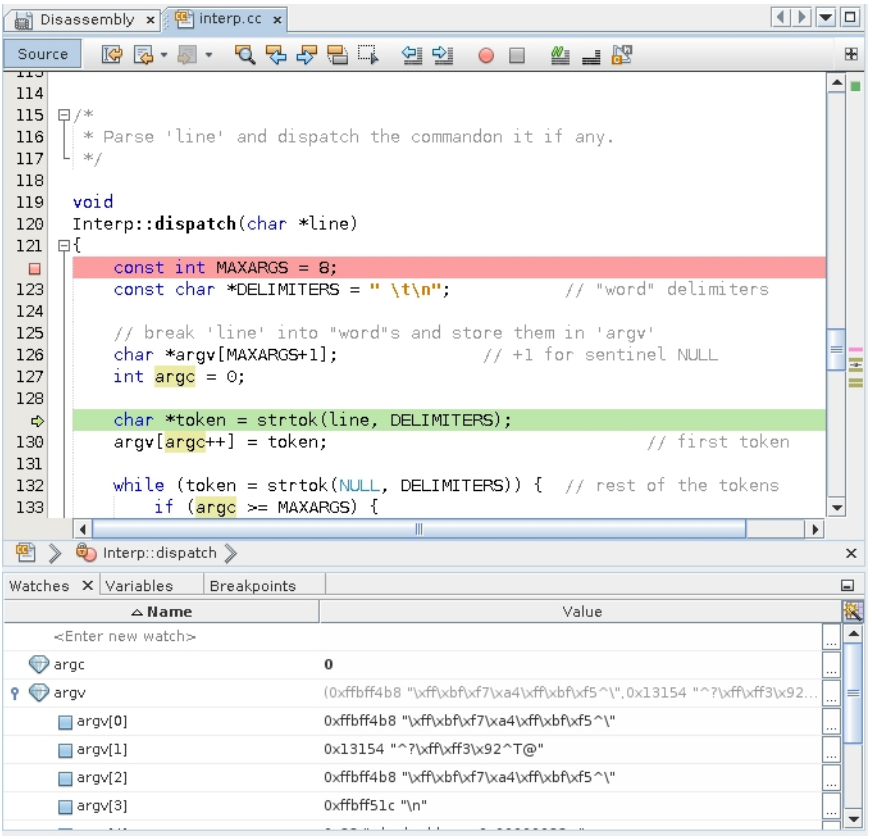
---

6. 单击 "Step Over"（步过）(F8)  两次，直到绿色的 PC 箭头指向 `int argc = 0;`。
7. 由于 `argc` 将成为 `argv` 的索引，因此也为 `argc` 创建监视。请注意，`argc` 当前也未初始化，并且可能包含不需要的值。

   由于您在 `argv` 的监视之后为 `argc` 创建了监视，因此它会在 "Watches"（监视）窗口中显示在第二的位置。
8. 要按照字母顺序对监视名称进行排序，请单击 "Name"（名称）列标题对该列进行排序。请注意下图中的排序三角形：

9. 单击 "Step Over"（步过）(F8)  。

   `argc` 现在显示其初始化值 0 并以粗体显示，以表示该值刚刚更改过。



   应用程序将调用 `strtok（）`。

10. 单击 "Step Over"（步过）步过该函数，并观察 `token` 是否为 NULL（例如，通过使用气球表达式求值）。

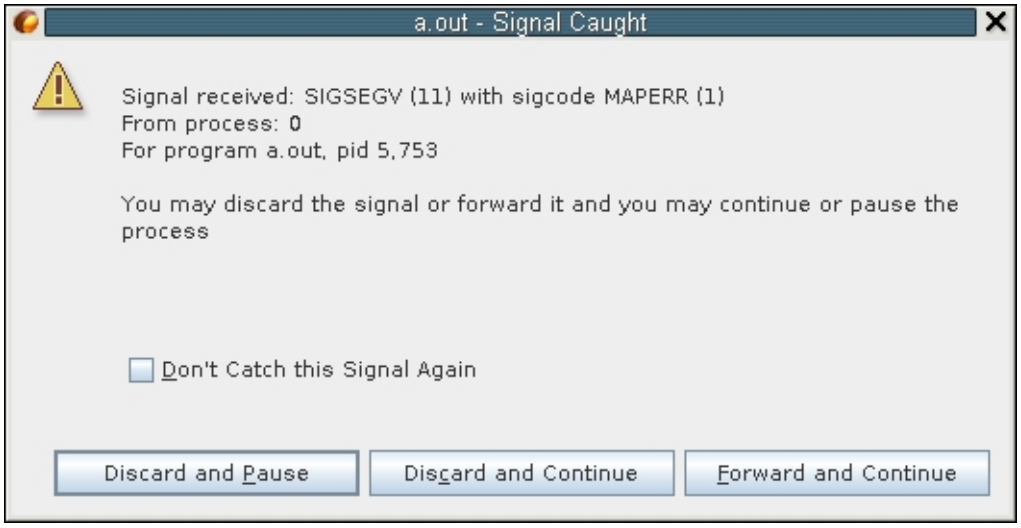
    `strtok（）` 函数有助于拆分字符串，例如拆分为由 `DELIMITERS` 之一分隔的标记。有关更多信息，请参见 `strtok`(3) 手册页。

11. 再次单击 "Step Over"（步过）将标记分配给 `argv`。循环中将存在对 `strtok（）` 的调用。

    在步过时，您不会进入循环（不再有标记），系统会分配一个 NULL。

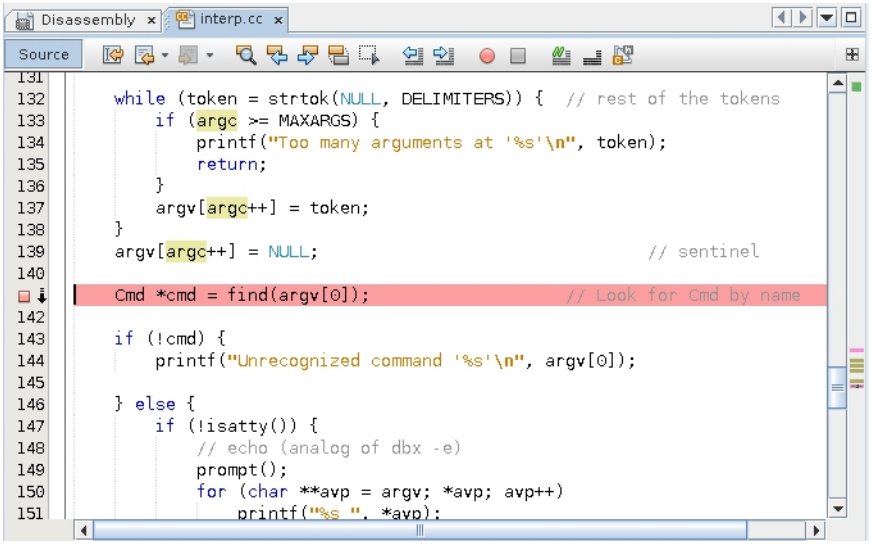
12. 也步过该分配，以达到调用的阈值，从而查明样例程序发生崩溃的位置。

13. 为了仔细检查程序是否在此处崩溃，步过对 `find（）` 的调用。
    "Signal Caught"（捕获的信号）警报框再次显示。



14. 像以前一样单击 "Discard and Pause"（放弃并暂停）。
    在 `Interp::dispatch（）` 中停止之后第一次对 `find（）` 的调用实际上就是出错的位置。您可以快速返回到初始调用 `find（）` 的位置。

    a. 单击 "Make Caller Current"（使调用方成为当前调用方）  。
    b. 在 `find（）` 的调用点处开启/关闭行断点。
    c. 打开 "Breakpoints"（断点）窗口并禁用 `Interp::dispatch（）` 函数断点。
    `dbxtool` 的外观应如下所示：



    d. 向下的箭头表示在第 141 行上设置了两个断点，其中一个断点已禁用。

15. 单击 "Restart"（重新启动）  并在 "Debugger Console"（调试器控制台）窗口中按回车键。

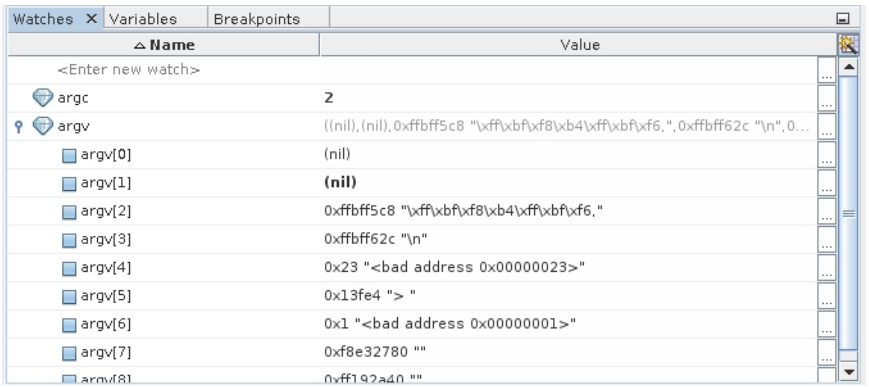
    程序在对 `find（）` 的调用前返回。请注意，按 "Restart"（重新启动）按钮会导致重新启动。进行调试时，重新启动的频率要比刚开始时大很多。

    提示 - 如果您重新生成程序（例如，在搜索和修复错误之后），不需要退出 `dbxtool` 并重新启动。单击 "Restart"（重新启动）按钮时，`dbx` 会检测到程序（或其任何成员）已重新编译，因此会将其重新加载。

    因此，请考虑将 `dbxtool` 放在桌面上（可以最小化），以便随时用于调试问题。

    ---

16. 错误在何处？请再次查看监视：



    请注意，`argv[0]` 为 NULL，因为对 `strtok（）` 的第一次调用返回 NULL（由于该行为空，没有标记）。

    如果您愿意，可以先修复此错误，然后再继续本教程的剩余部分。

    如果您要在调试器下运行该程序，则可以在调试器中修补代码（如“使用断点脚本修补代码” [38]中所述）。

该示例代码的开发者很可能已测试此条件并绕过 `Interp::dispatch（）` 的剩余部分。

讨论

该示例说明了最常见的调试模式，在该模式下，用户会在出错之前的某个点停止行为异常的程序，然后单步执行代码，将代码的本意与代码的实际行为相比较。

下一节介绍一些使用断点的高级技巧以避免在本示例中使用的某些单步执行和监视。

使用高级断点技巧

本节演示了使用断点的一些高级技巧：

- 使用断点计数
- 使用受限制的断点

- 拾取有用的断点计数
- 监视点
- 使用断点条件
- 使用弹出进行微重播
- 使用修复并继续

本节和示例程序受到了在 dbx 中发现的一个实际错误的启发，该错误的发现顺序与本节中所述的顺序相同。

注 - 要获取本节中显示的正确输出，示例程序必须仍“包含错误”。如果修复了错误，请从“程序示例” [2]重新下载 OracleDeveloperStudio12.5-Samples 目录。

源代码包含一个名为 in 的样例输入文件，该文件会在示例程序中触发一个错误。in 包含以下代码：

```
display nonexistent_var # should yield an error
display var
stop in X # will cause one "stopped" message and display
stop in Y # will cause second "stopped" message and display
run
cont
cont
run
cont
cont
```

使用该输入文件运行程序时，输出如下所示：

```
$ a.out < in
> display nonexistent_var
error: Don't know about 'nonexistent_var'
> display var
will display 'var'
> stop in X
> stop in Y
> run
running ...
stopped in X
var = {
      a = '100'
      b = '101
      c = '<error>'
      d = '102
      e = '103'
      f = '104'
      }
> cont
stopped in Y
var = {
      a = '105'
      b = '106'
      c = '<error>'
      d = '107'
      e = '108'
      f = '109'
      }
> cont
exited
> run
running ...
stopped in X
var = {
```

```
      a = '110'
      b = '111'

      c = '<error>'
      d = '112'
      e = '113'
      f = '114'
      }
> cont
stopped in Y
var = {
      a = '115'
      b = '116'

      c = '<error>'
      d = '117'
      e = '118'
      f = '119'
      }
> cont
exited
> quit
Goodby
```

此输出可能看起来很长，但此示例的重点是展示长时间运行的复杂程序所使用的技术，在这些程序中，单步执行代码或仅仅进行跟踪是不切实际的。

请注意，显示字段 `c` 的值时，您将获取 `<error>` 的值。如果该字段包含错误地址，可能会发生这样的情形。

问题

请注意，当您第二次运行程序时，您收到了在第一次运行时没有收到的其他错误消息：

```
error: cannot get value of 'var.c'
```

`error（）` 函数使用变量 `err_silent` 使错误消息在某些情况下不再出现。例如，对于 display 命令，不会显示错误消息，而会将问题显示为 `c = '<error>'`。

步骤 1：反复性

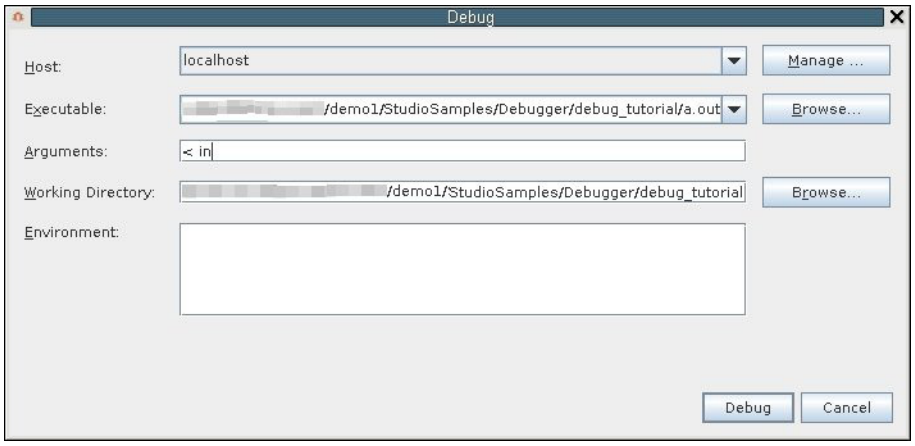
第一步是设置一个调试目标并配置该目标，以便可通过单击 "Restart"（重新启动）  轻松地重复错误。

按如下方式开始调试程序：

1. 如果您尚未编译示例程序，请按照 “程序示例” [2] 中的说明进行操作。
2. 选择 "Debug"（调试）> "Debug Executable"（调试可执行文件）。
3. 在 "Debug Executable"（调试可执行文件）对话框中，浏览可执行文件或键入可执行文件的路径。
4. 在 "Arguments"（参数）字段中，键入：

   ```
   < in
   ```

   可执行文件路径的目录部分显示在 "Working Directory"（工作目录）字段中。
5. 单击 "Debug"（调试）。



在真实情形中，您可能还希望填充 "Environment"（环境）字段。

调试程序时，`dbxtool` 会创建一个调试目标。您可以通过选择 "Debug"（调试）> "Debug Recent"（调试近来的）并选择所需的可执行文件来使用同一调试配置。

您可以从 `dbx` 命令行设置其中的多个属性。这些属性将存储在调试目标配置中。

以下技巧有助于轻松保持可重复性。添加断点时，可以通过单击 "Restart"（重新启动）快速转至某个感兴趣的位置，而不必在出现各种中间断点时单击 "Continue"（继续）。

步骤 2：第一个断点

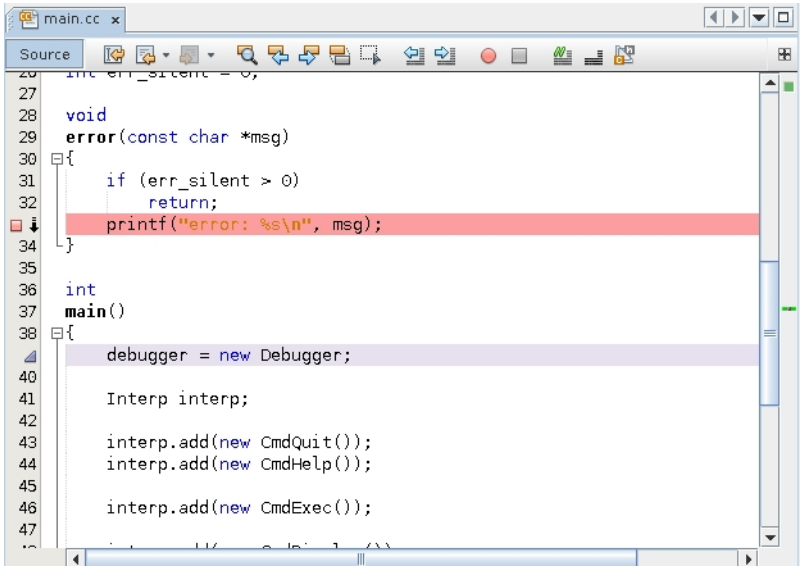
在 `error（）` 函数输出一条错误消息时，在该函数内放置第一个断点。此断点将成为第 33 行上的一个行断点。

在较大的程序中，您可以通过键入以下内容（例如，在 "Debugger Console"（调试器控制台）窗口中）来轻松更改编辑器窗口中的当前函数：

```
(dbx) func error
```

淡紫色条表示 `func` 命令所找到的匹配项。

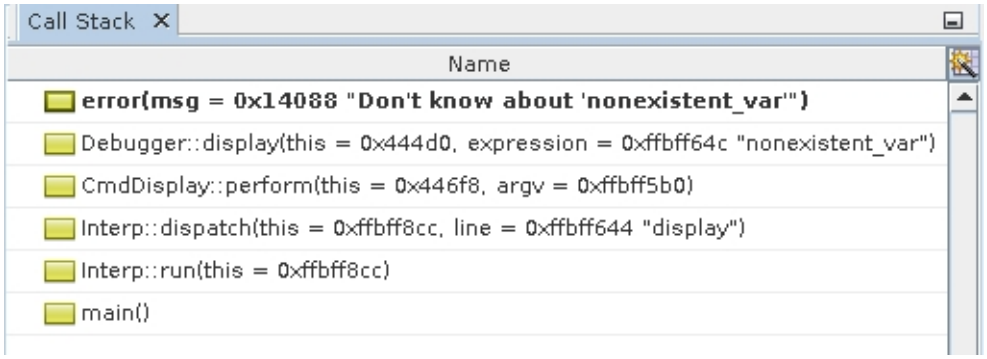
1. 通过在编辑器窗口左边界中数字 33 的上面单击，创建行断点。



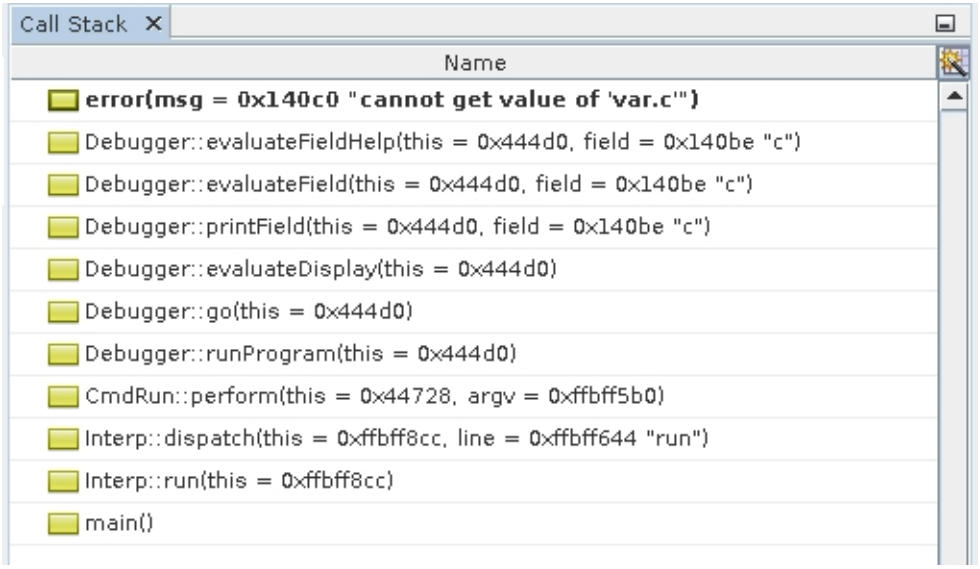
2. 单击 "Restart"（重新启动）  以运行该程序，命中断点时，堆栈跟踪会显示由于 `in` 文件中的模拟命令而生成的错误消息：

```
> display var # should yield an error
```

调用 `error()` 是预期行为。



3. 单击 "Continue"（继续）  继续执行该进程并再次命中该断点。
此时将显示一条意外的错误消息。



步骤 3：断点计数

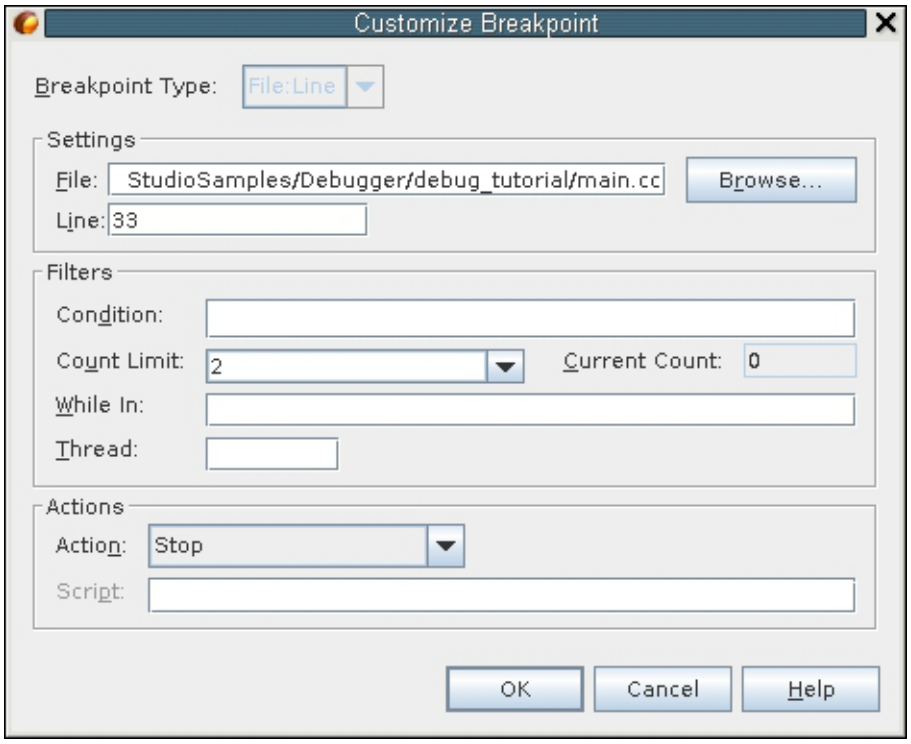
最好是在每次运行时都能反复到达此位置，而不必在第一次命中断点之后由于以下命令而单击 "Continue"（继续）：

```
> display var # should yield an error
```

您可以编辑程序或输入脚本，然后消除第一个麻烦的 display 命令。但是，您正在处理的特定输入顺序可能是重现此错误的关键，因此您不需要更改输入。

由于您关注的是第二次到达此断点，因此，可将其计数设置为 2。

1. 在 "Breakpoints"（断点）窗口中，右键单击该断点，然后选择 "Customize"（定制）。
2. 在 "Customize Breakpoint"（定制断点）对话框中，在 "Count Limit"（计数限制）字段中键入 2。
3. 单击 "OK"（确定）。



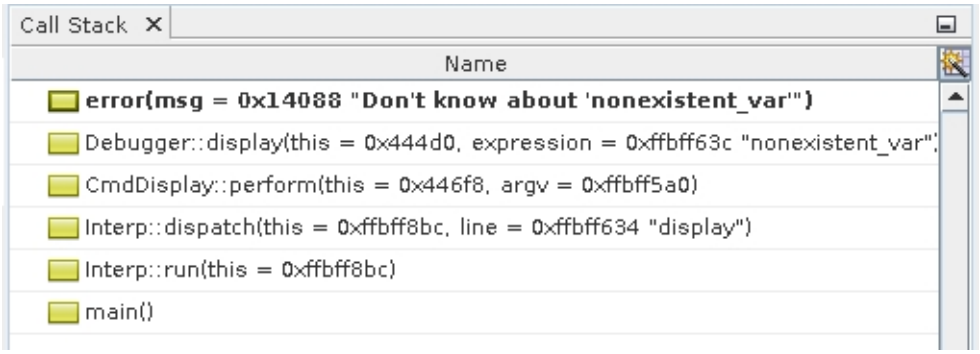
现在，您就可以反复到达您关注的位置了。

在本例中，是否选择计数 2 无关紧要。但是，有时会对某个感兴趣的位置进行多次调用。请参阅“步骤 7：确定计数值” [32]以轻松选择合适的计数值。但现在，您可以先了解一下如何通过另一种方式仅在您关注的调用中在 `error（）` 中停止。

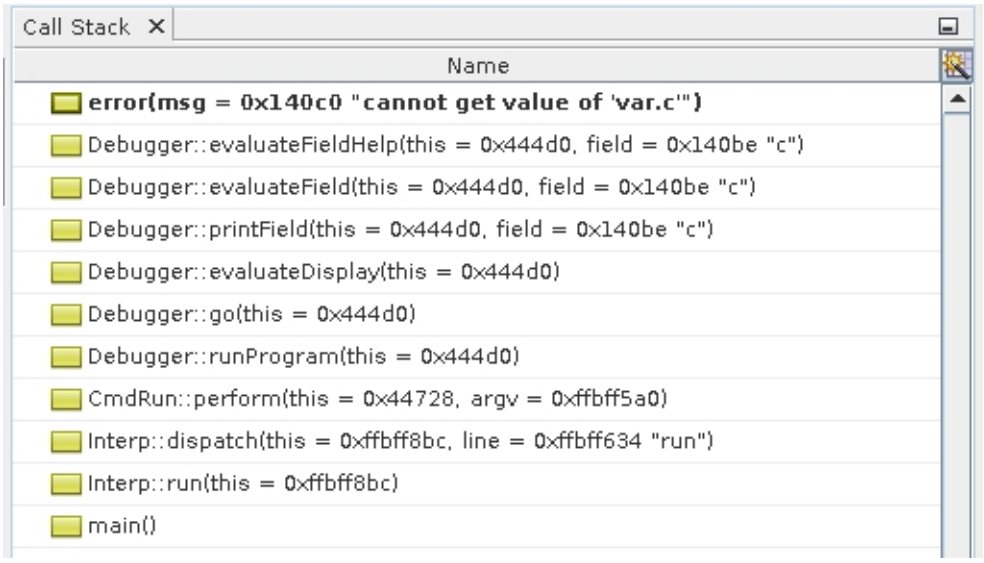
步骤 4：受限制的断点

1. 对于 `error（）` 内部的断点，打开 "Customize Breakpoint"（定制断点）对话框，然后通过从 "Count Limit"（计数限制）的下拉式列表中选择 "Always stop"（总是停止）禁用断点计数。
2. 重新运行程序。

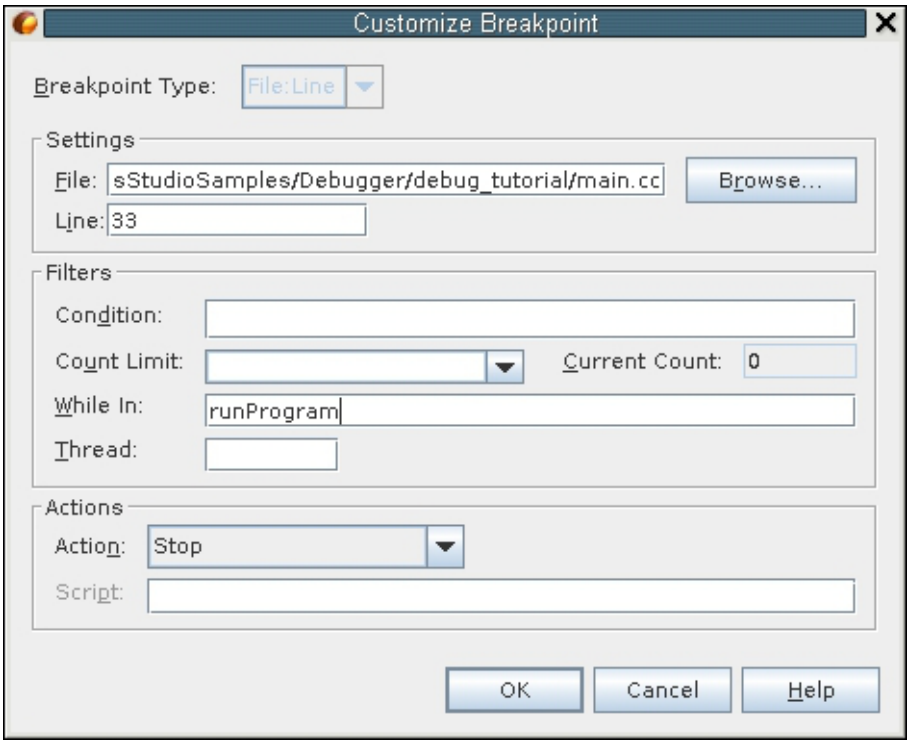
   注意两次在 `error（）` 中停止的堆栈跟踪。第一次，在 `error（）` 中的停止与以下屏幕相似：



第二次，在 `error（）` 中的停止与以下屏幕相似：



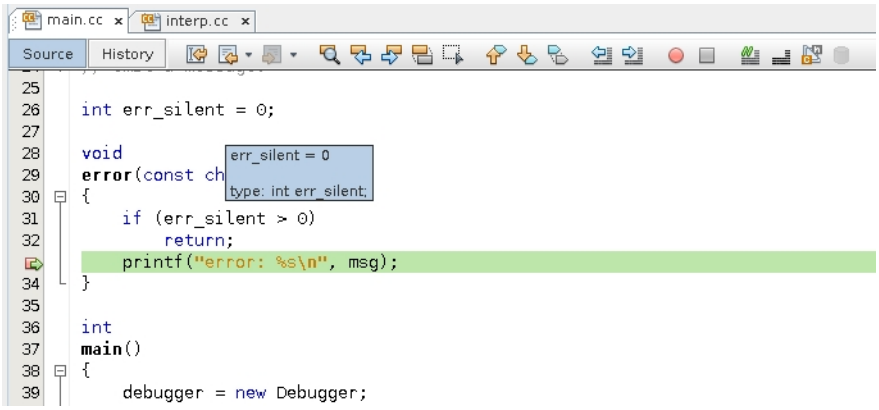
要安排为在从 `runProgram`（第 [7] 帧）调用时在此断点处停止，请再次打开 "Customize Breakpoint"（定制断点）对话框并将 "While In"（满足条件）字段设置为 runProgram。



步骤 5：查找原因

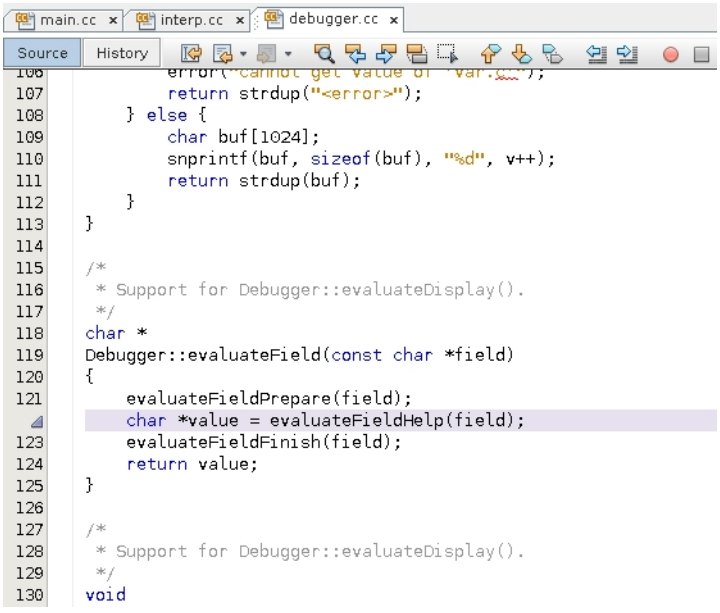
由于 err_silent 不 > 0，因此发出了不需要的错误消息。通过气球表达式求值看一下 err_silent 的值。

1. 将光标放在第 31 行中 err_silent 的上方，然后等待其值显示出来。

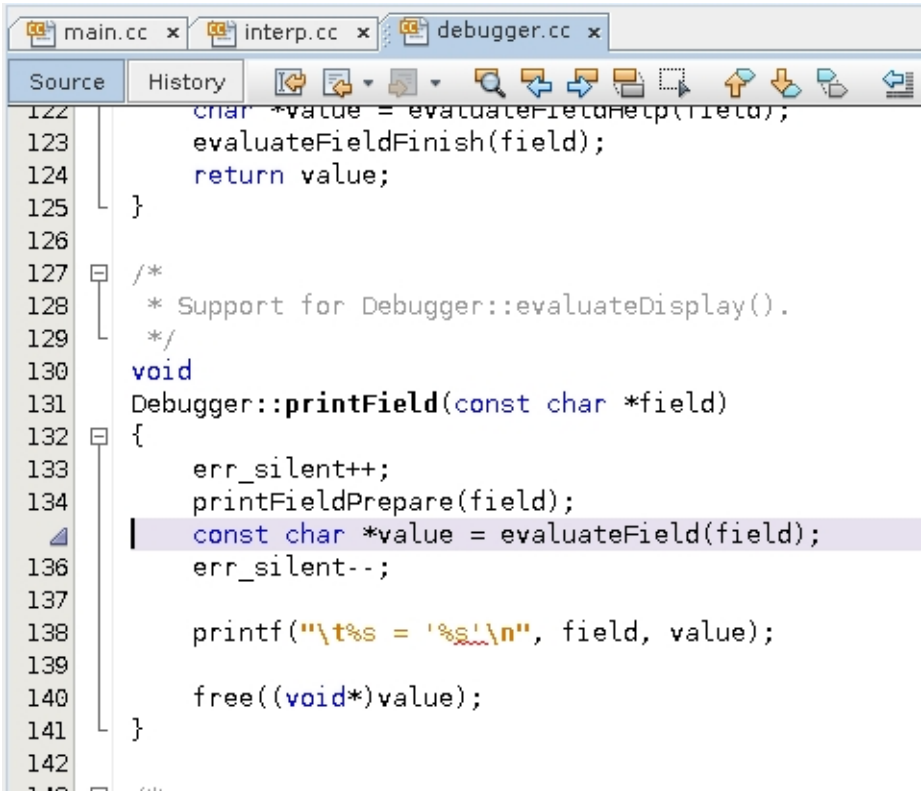


跟随堆栈看一下设置 err_silent 的位置。

2. 单击 "Make Caller Current"（使调用方成为当前调用方）  两次以到达 evaluateField ，该函数已调用 evaluateFieldPrepare（），模拟一个可能正在处理 err_silent 的复杂函数(。)



3. 再次单击 "Make Caller Current"（使调用方成为当前调用方）以到达 printField（），此处 err_silent 正在递增。printField（） 也已调用 printFieldPrepare（），printFieldPrepare 也模拟一个可能正在处理 err_silent 的复杂函数。



请注意 `err_silent++` 和 `err_silent--` 如何将某些代码包围起来。

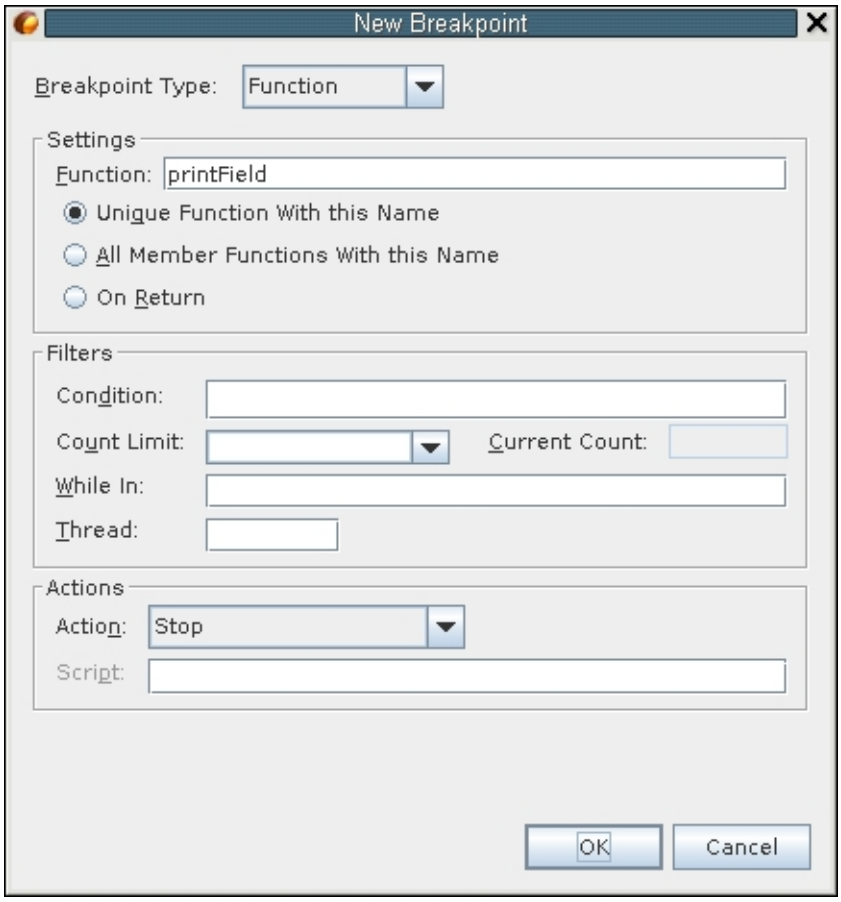
`err_silent` 可能在 `printFieldPrepare（）` 或 `evaluateFieldPrepare（）` 中出错，或者当控制到达 `printField（）` 时 err_silent 已经出错。

步骤 6：更多断点计数

要查明 `err_silent` 是在对 `printField（）` 的调用之前还是之后出错，请在 `printField（）` 中放置一个断点。

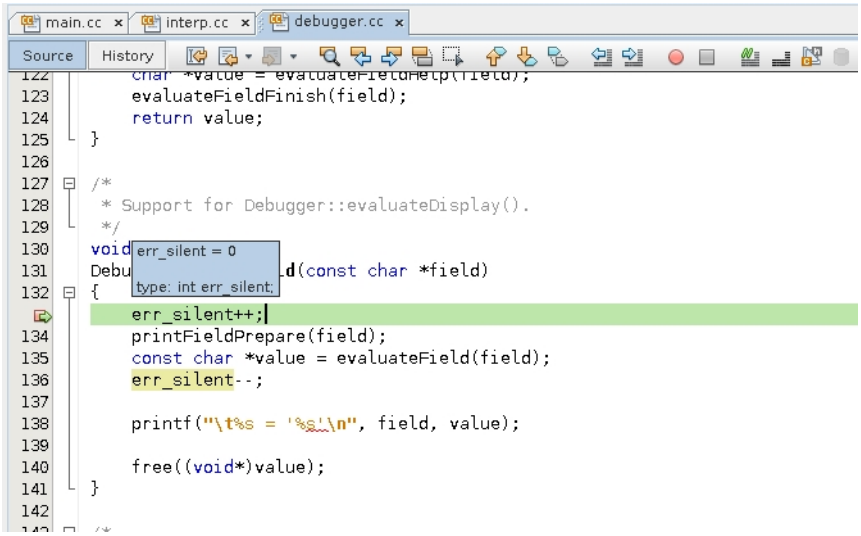
1. 选择 `printField（）`，右键单击，然后选择 "New Breakpoint"（新建断点）。

   新的断点类型已预先选择，并且 "Function"（函数）字段已使用 `printfield` 预先填充。
2. 单击 "OK"（确定）。



3. 单击 "Restart"（重新启动）  。

   第一次命中断点的时间是第一次运行期间第一次停止在第一个字段 `var.a` 上时。`err_silent` 为 0 是可以接受的。



4. 单击 "Continue"（继续）。

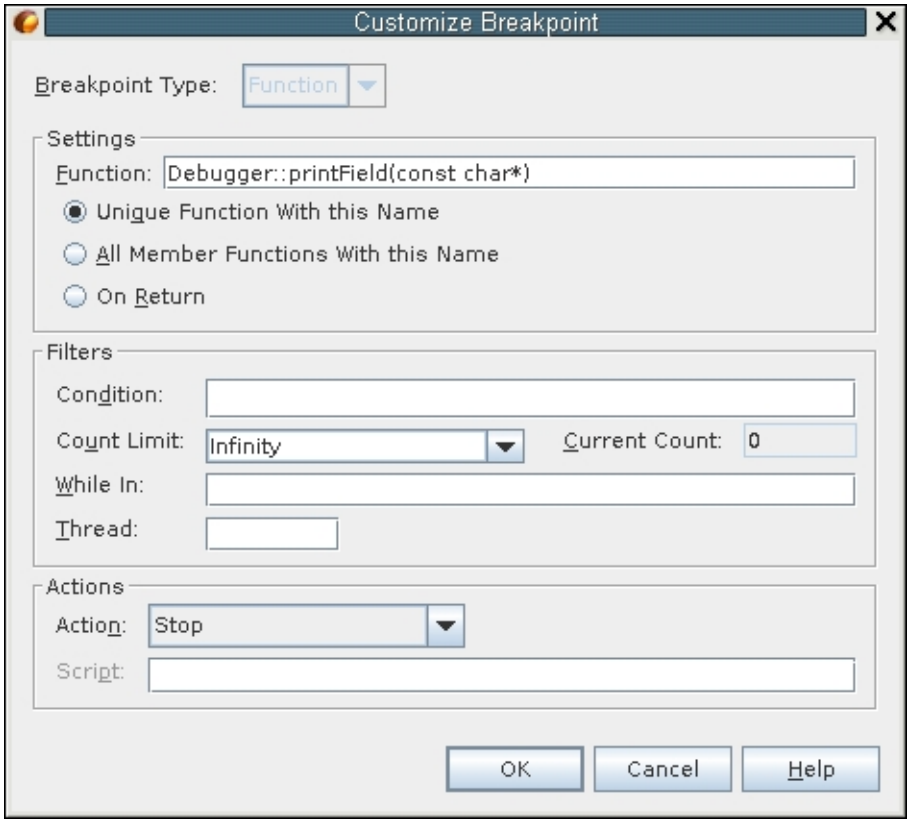
   `err_silent` 仍可以接受。
5. 再次单击 "Continue"（继续）。

   `err_silent` 仍可以接受。

到达对 `printField()` 的特定调用（该调用导致了不需要的错误消息）可能需要一段时间。您需要在 `printField` 断点上使用断点计数。可是应该将计数设置成什么呢？在该简单示例中，您可以尝试对运行和停止以及显示的字段进行计数，但实际上该过程可能会更加困难。有一种方法可以半自动地确定该计数。

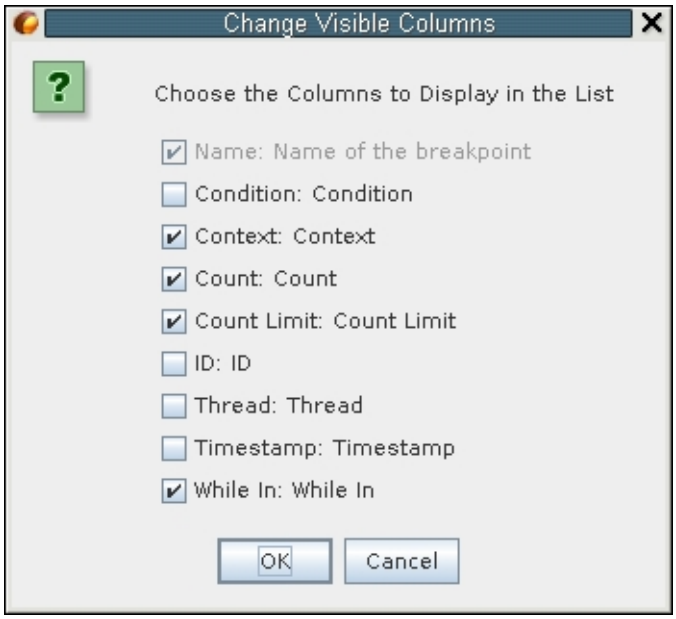
步骤 7：确定计数值

1. 打开 "Customize Breakpoint"（定制断点）对话框，找到 `printField()` 上的断点，然后将 "Count Limit"（计数限制）字段设置为无限大。

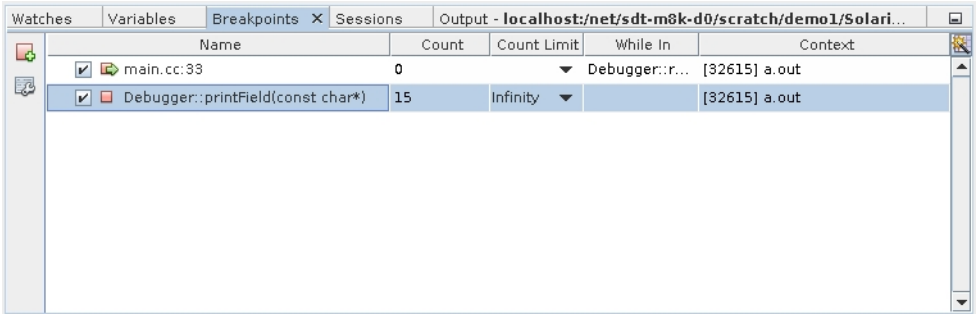


   此设置意味着您将永远不会在此断点处停止。但是，仍将进行计数。
2. 将 "Breakpoints"（断点）窗口设置为显示更多属性（如计数）。

   a. 单击 "Breakpoints"（断点）窗口右上角的 "Change Visible Columns"（更改可视列）按钮  。
   b. 选择 "Count Limit"（计数限制）、"Count"（计数）和 "While In"（满足条件）。
   c. 单击 "OK"（确定）。



3. 再次运行程序。您将命中 `error（）` 内部的断点，也就是受 `runProgram（）` 限制的断点。
4. 现在来看一下 `printField（）` 上断点的计数。



计数为 15。

5. 再次在 "Customize Breakpoint"（定制断点）窗口中单击 "Count Limit"（计数限制）列中的下拉式列表，选择 "Use current Count value"（使用当前计数值）将当前计数传送给计数限制，然后单击 "OK"（确定）。

现在如果运行程序，您将在最后一次调用 `printField（）` 时在该函数中停止，然后显示意外的错误消息。

步骤 8：确定具体原因

再次使用气球表达式求值检查 `err_silent`。现在的值为 -1。最有可能的原因是，在您到达 `printField（）` 之前，一个 `err_silent--` 执行得太多，或者一个 `err_silent++` 执行得太少。

您可以通过仔细检查代码，在与该示例类似的小程序中找到这个不匹配的 `err_silent` 对。但是，大型程序可能包含大量的以下配对：

```
err_silent++;
err_silent--;
```

更为快捷地找到不匹配的 err_silent 对的方法是使用监视点。

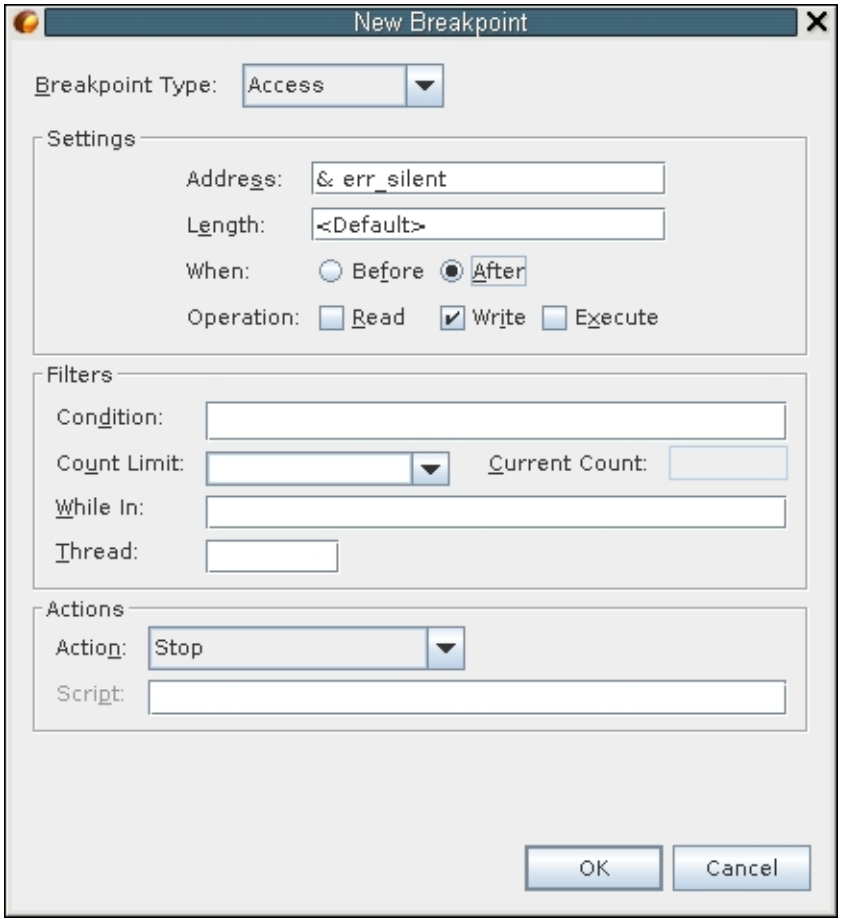
错误的原因可能根本不是不匹配的 `err_silent++;` 和 `err_silent--;` 对，而是一个覆盖了 `err_silent` 内容的异常指针。在捕获此类问题时，监视点会比较有效。

步骤 9：使用监视点

在 `err_silent` 上创建监视点：

1. 选择 `err_silent` 变量，右键单击，然后选择 "New Breakpoint"（新建断点）。
2. 将 "Breakpoint Type"（断点类型）设置为 "Access"（访问）。

   请注意 "Settings"（设置）部分如何变化以及 "Address"（地址）字段是如何成为 `& err_silent` 的。
3. 在 "When"（时间）字段中选择 "After"（之后）。
4. 在 "Operation"（操作）字段中选择 "Write"（写入）。
5. 单击 "OK"（确定）。



6. 运行程序。

   您在 `init（）` 处停止。`err_silent` 递增到了 1，之后就停止了执行。

7. 单击 "Continue"（继续）。

   您再次在 `init`（） 中停止。
8. 再次单击 "Continue"（继续）。

   您再次在 `init`（） 中停止。
9. 再次单击 "Continue"（继续）。

   您再次在 `init`（） 中停止。
10. 再次单击 "Continue"（继续）。

    现在您将在 `stopIn`（） 中停止。此时看起来也是一切正常，没有出现 -1。

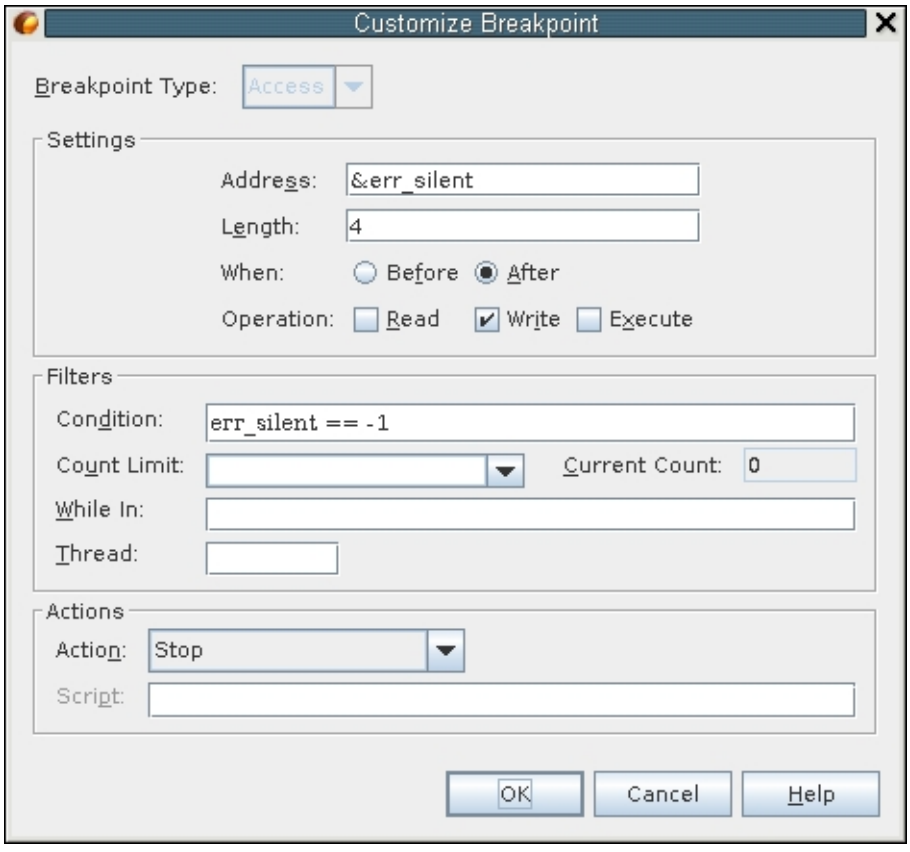
可以设置断点条件，而不是反复地单击 "Continue"（继续），直到将 `err_silent` 设置为 -1。

步骤 10：断点条件

为您的监视点添加一个条件：

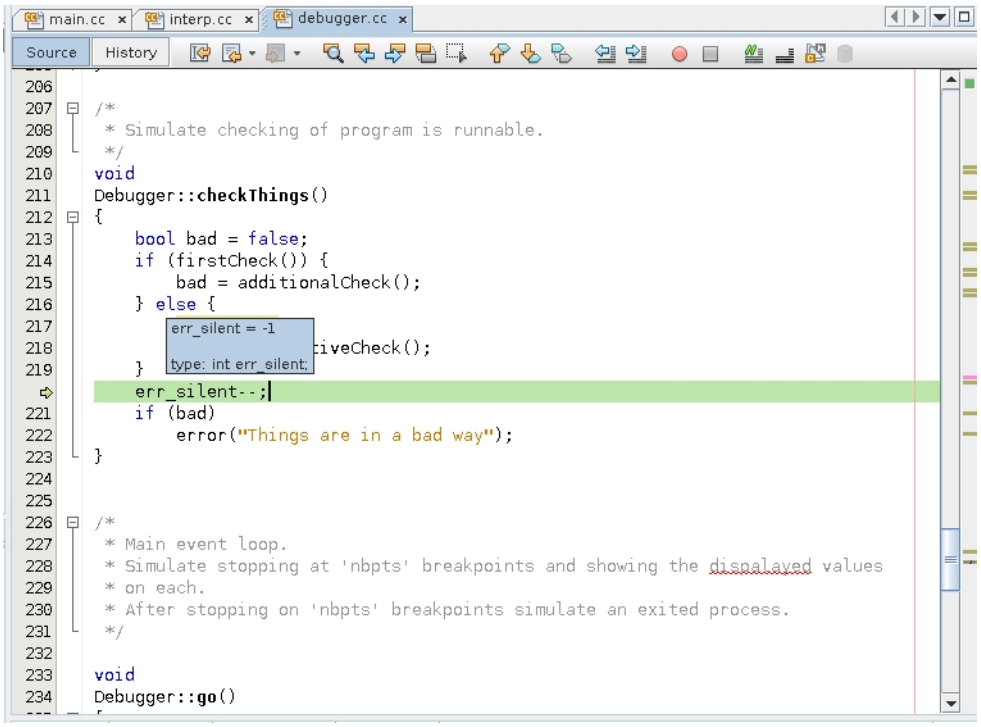
1. 在 "Breakpoints"（断点）窗口中，右键单击 "After"（之后）写入断点，然后选择 "Customize"（定制）。
2. 验证是否在 "When"（时间）字段中选择了 "After"（之后）。

   通过选择 "After"（之后），您可以查看更改后的 `err_silent` 的值。
3. 将 "Condition"（条件）字段设置为 `err_silent == -1`。
4. 单击 "OK"（确定）。



5. 再次运行程序。

   您在 checkThings（） 处停止，这是第一次将 err_silent 设置为 -1。在您查找匹配的 err_silent++ 时，您会看清错误：err_silent 仅在该函数的 else 部分中递增。



   这是您所要寻找的错误吗？

步骤 11：通过弹出堆栈来验证诊断

有一种方法可以核实您是否确实检查完函数的 else 块，那就是在 checkThings（） 上设置一个断点并运行程序。但 checkThings（） 可能会被多次调用。您可以使用断点计数或受限制的断点来实现对 checkThings（） 的正确调用，但重播最近所执行内容的更快方法是弹出堆栈。

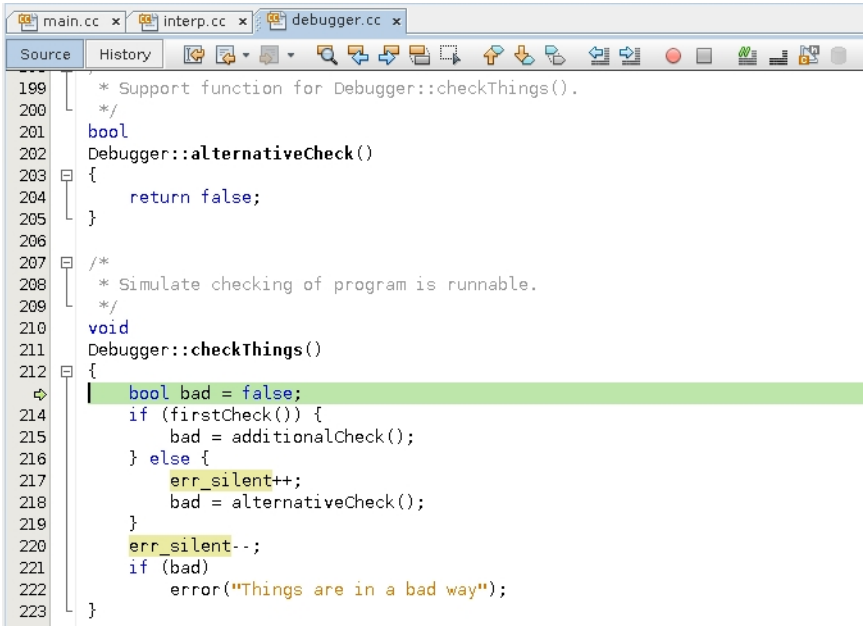
1. 选择 "Debug"（调试）> "Stack"（堆栈）> "Pop Topmost Call"（弹出最顶层调用）。

   请注意 "Pop Topmost Call"（弹出最顶层调用）不会撤消任何内容。尤其是，err_silent 的值已出错，因为您正从数据调试切换到控制流调试。

   进程状态恢复到包含对 checkThings（） 的调用的行的开始处。

2. 单击 "Step Into"（步入）  并在再次调用 checkThings 时进行观察。

   在单步执行 checkThings（） 时，您可以验证该进程是否执行了 if 块（此处 err_silent 没有递增，并且接着会递减至 -1）。

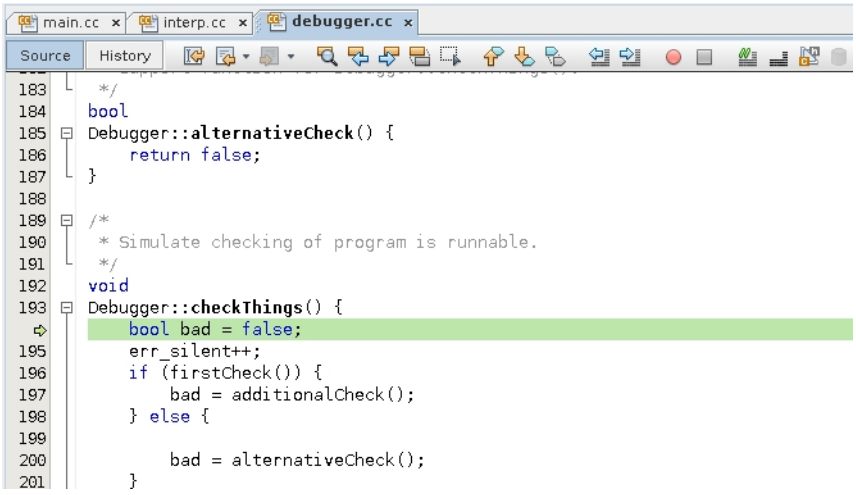


尽管您似乎已找到编程错误，但您可能需要反复对其进行检查。

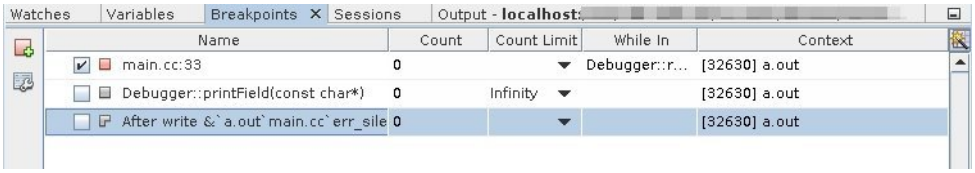
步骤 12：使用修复进一步验证诊断

请修复代码并验证错误确实已经消除。

1. 通过将 `err_silent++` 置于 `if` 语句的上方来修复代码。

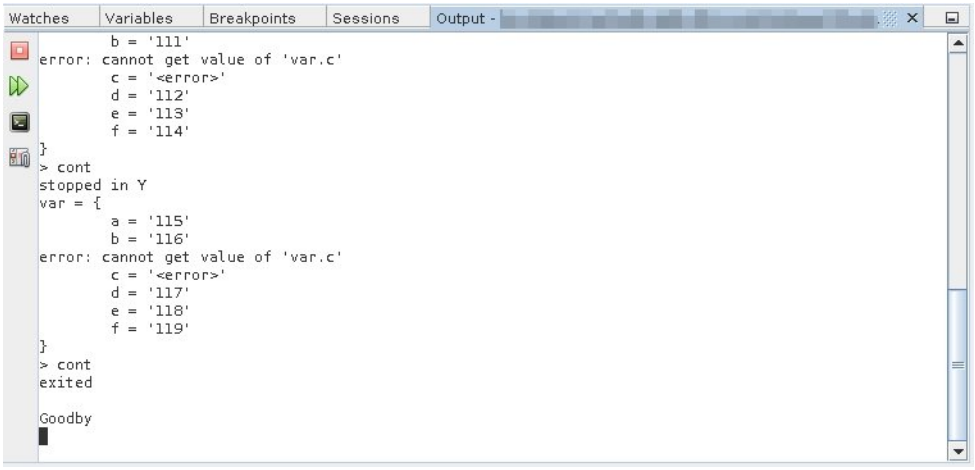


2. 选择 "Debug"（调试）> "Apply Code Changes"（应用代码更改），或者按 "Apply Code Changes"（应用代码更改）按钮  。
3. 禁用 `printField` 断点和监视点，但保留 `error()` 中断点的启用状态。



4. 再次运行程序。

请注意，程序已完成但没有命中 `error（）` 中的断点，其输出符合预期。



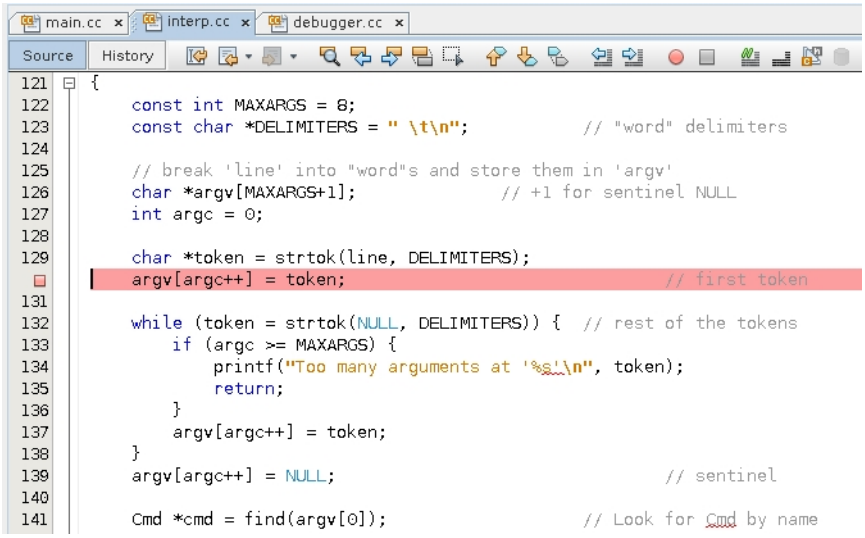
讨论

该示例说明了与“使用断点和步进” [13]结尾处所讨论的模式相同的模式，即，用户在出错之前的某个点停止行为异常的程序，然后单步执行代码，将代码的本意与代码实际的行为相比较。主要差异在于，查找出错之前的点的过程要复杂一些。

使用断点脚本修补代码

在“使用断点和步进” [13]中，您发现了一个错误，一个空行产生了一个 NULL 首标记并导致一个 SEGV。您可以使用某种解决方法来避免该错误。

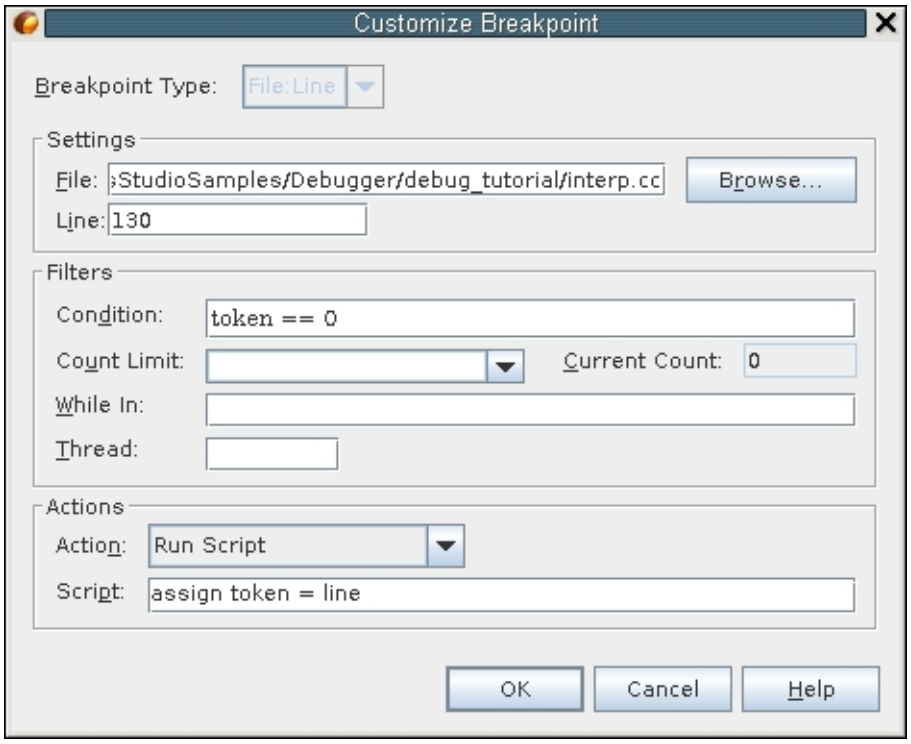
1. 删除您先前创建的所有断点。您可以通过在 "Breakpoints"（断点）窗口中右键单击并选择 "Delete All"（全部删除）快速执行此操作。
2. 在 "Debug Executable"（调试可执行文件）对话框中删除 `<in` 参数。
3. 在 `interp.cc` 中第 130 行处开启/关闭一个行断点。



4. 在 "Breakpoints"（断点）窗口中，右键单击刚刚创建的断点，然后选择 "Customize"（定制）。
5. 在 "Customize Breakpoint"（定制断点）对话框中，在 "Condition"（条件）字段中键入 `token == 0`。
6. 从 "Action"（操作）下拉式列表中选择 "Run Script"（运行脚本）。
7. 在 "Script"（脚本）字段中，键入 `assign token = line`。

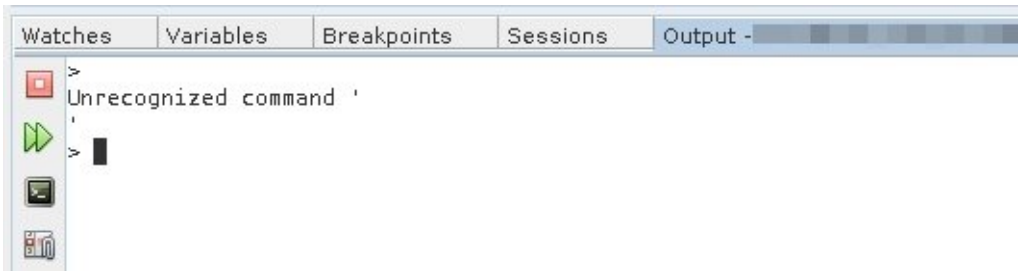
注 - 您无法执行 `assign token = “dummy”`，因为 `dbx` 无法在调试的进程中分配 `dummy` 字符串。另一方面，已知 `line` 等于 `""`。

该对话框看起来应如以下屏幕所示：



8. 单击 "OK"（确定）。

现在，如果您运行程序并输入一个空行，该程序会向您发出警告，而不是崩溃，如以下屏幕中所示：



如果您查看 `dbxtool` 发送给 `dbx` 的命令，可以更加清楚地了解该解决方法。

```
when at "interp.cc":130 -if token == 0 { assign token = line; }
```

结论

通过 Oracle Developer Studio `dbxtool`，您可以使用方便的 GUI 格式精确定位导致您的程序崩溃的问题区域。借助 `dbxtool`，通过在您的代码中创建断点和单步执行，即可轻松调试您的代码。通过 `dbxtool`，还可以使用一些高级断点技巧以及监视点、断点条件和弹出堆栈等功能，帮助您识别出代码中的错误并修复问题。

Oracle Developer Studio 12.5: dbxtool 教程

文件号码 E71968

版权所有 © 2010, 2016, Oracle 和/或其附属公司。保留所有权利。

本软件和相关文档是根据许可证协议提供的，该许可证协议中规定了关于使用和公开本软件和相关文档的各种限制，并受知识产权法的保护。除非在许可证协议中明确许可或适用法律明确授权，否则不得以任何形式、任何方式使用、拷贝、复制、翻译、广播、修改、授权、传播、分发、展示、执行、发布或显示本软件和相关文档的任何部分。除非法律要求实现互操作，否则严禁对本软件进行逆向工程设计、反汇编或反编译。

此文档所含信息可能随时被修改，恕不另行通知，我们不保证该信息没有错误。如果贵方发现任何问题，请书面通知我们。

如果将本软件或相关文档交付给美国政府，或者交付给以美国政府名义获得许可证的任何机构，则适用以下注意事项：

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

本软件或硬件是为了在各种信息管理应用领域内的一般使用而开发的。它不应被应用于任何存在危险或潜在危险的应用领域，也不是为此而开发的，其中包括可能会产生人身伤害的应用领域。如果在危险应用领域内使用本软件或硬件，贵方应负责采取所有适当的防范措施，包括备份、冗余和其它确保安全使用本软件或硬件的措施。对于因在危险应用领域内使用本软件或硬件所造成的一切损失或损害，Oracle Corporation 及其附属公司概不负责。

Oracle 和 Java 是 Oracle 和/或其附属公司的注册商标。其他名称可能是各自所有者的商标。

Intel 和 Intel Xeon 是 Intel Corporation 的商标或注册商标。所有 SPARC 商标均是 SPARC International, Inc 的商标或注册商标，并应按照许可证的规定使用。AMD、Opteron、AMD 徽标以及 AMD Opteron 徽标是 Advanced Micro Devices 的商标或注册商标。UNIX 是 The Open Group 的注册商标。

本软件或硬件以及文档可能提供了访问第三方内容、产品和服务的方式或有关这些内容、产品和服务的信息。除非您与 Oracle 签订的相应协议另行规定，否则对于第三方内容、产品和服务，Oracle Corporation 及其附属公司明确表示不承担任何种类的保证，亦不对其承担任何责任。除非您和 Oracle 签订的相应协议另行规定，否则对于因访问或使用第三方内容、产品或服务所造成的任何损失、成本或损害，Oracle Corporation 及其附属公司概不负责。

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

Part No: E71968

Copyright © 2010, 2016, Oracle and/or its affiliates. All rights reserved.