My program for faking debugging demos

devblogs.microsoft.com/oldnewthing/20160113-00

January 13, 2016

Raymond Chen

When I present a debugging session at TechReady, there are a lot of things that can go wrong. A network hiccup might cause my debugger session to hang trying to load symbols over the network. An errant typo may cause the debugger to run off into the weeds. The machine I'm given to present on may be running an old version of Windows that the debugger doesn't support.

For my last TechReady presentation, I decided to play it safe and use a canned demo. The session still acted like a debugger, but I taught it canned responses to the commands I intended to execute. Today's Little Program is the program I wrote to do that. I intentionally limited myself to features available in C# 2.0, just to play it safe(r).

```
using System;
using System.IO;
using System.Threading;
using System.Collections.Generic;
using System.Text.RegularExpressions;

static class Extensions
{
 public static TValue ItemOrDefault<TKey, TValue>(
   this Dictionary<TKey, TValue> dict,
   TKey key,
   TValue def = default(TValue))
 {
  TValue value;
  return dict.TryGetValue(key, out value) ? value : def;
 }
}
```

The `ItemOrDefault` extension method looks up a value in a dictionary and returns the associated value if it exists, or a default value if it doesn't.

The rest of the work happens in the debugger emulator.

When you type a command, the emulator looks up the command in the master table and spits back the canned response. Inside the canned response you can embed commands. For example, the `@sleep nnn` command pauses playback for the specified number of milliseconds. There is also an `@if` command that lets me change the response based on the state of the virtual debugger.

Let's dive in.

```
class Emulator
{
 class CommandDescription
 {
  [Flags]
  public enum CommandFlags
  {
   None = 0,
   Meta = 1,
  }
```

Most of the time, everything inside a false conditional is skipped. The exceptions are the flow control primitives like `@else` and `@endif`, because if they got skipped over, you wouldn't be able to regain control! I called those commands *Meta* commands since they are commands which control other commands.

```
  public CommandDescription(Action<string> action,
                            CommandFlags flags = CommandFlags.None)
  {
   this.action = action;
   this.flags = flags;
  }

  public Action<string> action;
  public CommandFlags flags;
 }
 private Dictionary<string, CommandDescription> _commands;
```

The `_commands` dictionary maps `@`-commands to the corresponding handlers. The handler itself is passed its arguments as a string (for example, the `@sleep` handler receives `nnn` as its argument parameter). The flags specify whether the command should be skipped when encountered in a false conditional.

```
 void cmdSleep(string arg)
 {
  int milliseconds = int.Parse(arg);
  Thread.Sleep(milliseconds);
 }
```

The `@sleep` command simply sleeps the emulator for a little while. I use this to simulate the debugger pausing to do some thinking. No big deal.

```
void cmdLet(string arg)
{
 var args = arg.Split(new char[] { '=' }, 2);
 if (args.Length == 2)
 {
  _vars[args[0]] = args[1];
 }
 else
 {
  Echo("Internal debugger error 1");
 }
}
private Dictionary<string, string> _vars =
         new Dictionary<string, string>();
```

The `@let x=y` command creates a variable named `x` and sets its value to `y`, overwriting any previous one if it existed. The value of the variable is just a string. There is no math or anything. We implement it by splitting on the equals-sign, treating the left-hand side as the variable name and the right-hand side as the value.

```
private bool _taken = true;

void cmdIf(string arg)
{
 var args = arg.Split(new char[] { '=' }, 2);
 if (args.Length == 2)
 {
  _taken = _vars.ItemOrDefault(args[0], string.Empty) == args[1];
 }
 else
 {
  Echo("Internal debugger error 2");
 }
}

void cmdElse(string arg)
{
 _taken = !_taken;
}

void cmdEndif(string arg)
{
 _taken = true;
}
```

The `@if x=y` command checks whether the variable `x` has the value `y`. A variable that has not been defined is treated as if its value is the empty string.

We set the `_taken` member variable based on whether the conditional was satisfied or not. Note that `@if` does not nest. I could add supporting for nesting, but I've never needed it. If I ever need it, I'll add it. (There's also no error checking.)

```
 Emulator()
 {
  _commands = new Dictionary<string, CommandDescription> {
   { "sleep", new CommandDescription(cmdSleep) },
   { "let", new CommandDescription(cmdLet) },
   { "if", new CommandDescription(cmdIf, CommandDescription.CommandFlags.Meta) },
   { "else", new CommandDescription(cmdElse, CommandDescription.CommandFlags.Meta) },
   { "endif", new CommandDescription(cmdEndif, CommandDescription.CommandFlags.Meta)
},
  };
 }
```

And those are the commands we support in our debugger simulator. Not much, really.

```
 private bool _interrupted = false;

 void Echo(string s)
 {
  if (!_interrupted)
  {
   Thread.Sleep(1);
   Console.WriteLine(s);
  }
 }
```

Generated output is produced by the `Echo` method. If the user hasn't hit `Ctrl`+`C` to sto the output, the requested output is printed to the screen, with a brief delay to make the output speed look more realistic.

```
void ProcessResponseFile(string file)
{
 using (var streamCurrent = new StreamReader(file))
 {
  string s;
  while ((s = streamCurrent.ReadLine()) != null)
  {
   if (s.Length > 0 && s[0] == '@')
   {
    var args = s.Substring(1).Split(new char[] { ' ' }, 2,
                        StringSplitOptions.RemoveEmptyEntries);
    var desc = _commands.ItemOrDefault(args[0]);
    if (desc != null)
    {
     if ((desc.flags & CommandDescription.CommandFlags.Meta) != 0 ||
      _taken)
     {
      desc.action(args.Length > 1 ? args[1] : string.Empty);
     }
    }
    else
    {
     Echo("Internal debugger error 3");
    }
   }
   else
   {
    if (_taken)
    {
     Echo(s);
    }
   }
  }
 }
}
```

The `ProcessResponseFile` method reads the canned response and executes it. If the line begins with an `@`, we take the first word, look it up in the `_commands`, and execute it if we are in a taken branch, or if it is a meta-command. If it is not command, then we simply echo it if we are in a taken branch.

Finally, the program entry point loads up the rules and drives execution based on user commands.

```
Dictionary<string, string> _rules =
       new Dictionary<string, string>();
List<Tuple<string, string>> _rewrite =
       new List<Tuple<string, string>>();
```

The `_rules` dictionary maps strings typed by the user into response files. We'll talk about the `_rewrite` dictionary a little later.

```
void Run(string[] args)
{
 Console.CancelKeyPress += delegate(object sender,
                           ConsoleCancelEventArgs e)
 {
  e.Cancel = true;
  this._interrupted = true;
 };
```

We hook up a `Ctrl`+`C` handler that sets the `_interrupted` flag. This causes output to be suppressed, so that we can get back to the command loop faster.

```
 using (var rulesFile = new StreamReader(args[0]))
 {
  string s;
  while ((s = rulesFile.ReadLine()) != null)
  {
   if (s.Length > 0 && s[0] != ';')
   {
    var index = s.LastIndexOf('=');
    if (index >= 0)
    {
     var left = s.Substring(0, index);
     var right = s.Substring(index + 1);
     if (left.StartsWith("@rewrite "))
     {
      _rewrite.Add(new Tuple<string, string>
                   (left.Substring(9), right));
     }
     else
     {
      _rules.Add(left, right);
     }
    }
   }
  }
 }
```

The command line parameter is the name of the rules file. The rules file consists of lines of the form `x=y` where `x` is a command string and `y` is the name of the canned response file to use when the user types `x`. For example,

```
!locks=locks.txt
!locks -v=locksv.txt
```

says that if I type `!locks`, it will print the contents of the `locks.txt` file (executing any possible `@`-commands), whereas if I type `!locks -v`, it will use `locksv.txt`.

If the rules file has a line of the form

```
@rewrite x=y
```

then any command that starts with `x` will be treated as if it started with `y`. In my case, I had a rewrite rule

```
@rewrite !ntsdexts.locks=!locks
```

so that all the `!locks` commands also work with `!ntsdexts.locks`. From a minimalist point of view, the `@rewrite` rule wasn't necessary, because I could have just repeated every single `!locks` command with a matching `!ntsdexts.locks` command, but it's much less error-prone.

Blank lines and lines beginning with semicolons are ignored.

```
var startup = _rules.ItemOrDefault("@startup");
if (startup != null)
{
 ProcessResponseFile(startup);
}
```

If there is a rule of the form `@startup=xyz`, then we will run the `xyz` response file at startup. This lets me print the debugger banner and set variables to their initial values.

```
var prompt = _rules.ItemOrDefault("@prompt", string.Empty);
```

Oh, and the rule `@prompt=xyz` sets the prompt to `xyz`. I don't have any commands to change the prompt dynamically since I never needed it, but if I did, I would probably switch over to making `prompt` a special variable settable via `@let`.

```
StreamReader input = null;
if (args.Length > 1)
{
 input = new StreamReader(args[1]);
}
```

For testing purposes, I can pass a second file name on the command line which acts as canned input for the emulator. This lets me feed it canned input and watch it generate canned output. It's like letting two chatbots have a conversion.

```
while (true)
{
 _interrupted = false;
 Console.Write(prompt);
 string s;
 if (input == null)
 {
  s = Console.ReadLine();
 }
 else
 {
  s = input.ReadLine();
  if (s != null)
  {
   Console.Write(s);
   if (s.StartsWith("$$"))
   {
    Console.WriteLine();
   }
   else
   {
    // let user see the next step
    if (Console.ReadLine() == "q")
    {
     break;
    }
   }
  }
 }
 if (s == null)
 {
  break;
 }
```

Here is the main loop. We reset the interrupt flag because we successfully returned to the main loop. (The interrupt, if requested, did its job.) We display the prompt and collect a line (either from the canned input the actual console). If the line came from the canned input, we print it to the screen, and if it doesn't begin with two dollar signs (the comment command to the debugger), we also pause for input to let the human being running the program abandon the program by typing `q`.

```
 // Apply rewrite rules
 foreach (var tuple in _rewrite)
 {
  if (s.StartsWith(tuple.Item1))
  {
   s = tuple.Item2 + s.Substring(tuple.Item1.Length);
  }
 }
```

After getting the command, we apply the rewrite rules.

```
   var file = _rules.ItemOrDefault(s);
   if (file != null)
   {
    ProcessResponseFile(file);
   }
  }
```

We look up the command in our rules list to get the response file and execute it. If we didn't find the command in our list, we just ignore it. (If I tell a joke, people might not notice the mistake.)

```
  if (input != null)
  {
   input.Close();
  }
 }
```

And we finish with some cleanup.

```
 public static void Main(string[] args)
 {
  var p = new Emulator();
  p.Run(args);
 }
}
```

The main function just instantiates the emulator and runs it.

Here's a sample rules file:

```
bp contoso!MyFunction=bpmyfunction.txt
r=r.txt
g=g.txt

; Note! There is a space at the end of the next line
@prompt=0:001>

@startup=startup.txt
```

This is a simple debugger session where I break in at one location, set a breakpoint, resume execution, and watch the breakpoint get hit.

startup.txt

```
@let state=initialbreakpoint
(6820.6dfc): Break instruction exception - code 80000003 (first chance)
eax=7ece8000 ebx=00000000 ecx=00000000 edx=77b9dbeb esi=00000000 edi=00000000
eip=77b2f9fc esp=0120fb2c ebp=0120fb58 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!DbgBreakPoint:
77b2f9fc cc              int     3
```

r.txt

```
@if state=initialbreakpoint
eax=7ece8000 ebx=00000000 ecx=00000000 edx=77b9dbeb esi=00000000 edi=00000000
eip=77b2f9fc esp=0120f864 ebp=0120f890 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!DbgBreakPoint:
77b2f9fc cc              int     3
@else
eax=00f4fc84 ebx=00000000 ecx=00000000 edx=00000000 esi=af9208a3 edi=75cbb8ad
eip=00d310f3 esp=00f4fc6c ebp=00f4fca0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
contoso!MyFunction:
00d310f3 55              push    ebp
@endif
```

bpmyfunction.txt

```
@let state=myfunctionbreakpoint
```

g.txt

```
Breakpoint 0 hit
eax=00f4fc84 ebx=00000000 ecx=00000000 edx=00000000 esi=af9208a3 edi=75cbb8ad
eip=00d310f3 esp=00f4fc6c ebp=00f4fca0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
contoso!MyFunction:
00d310f3 55              push    ebp
```

The debug session goes like this:

```
(6820.6dfc): Break instruction exception - code 80000003 (first chance)
eax=7ece8000 ebx=00000000 ecx=00000000 edx=77b9dbeb esi=00000000 edi=00000000
eip=77b2f9fc esp=0120fb2c ebp=0120fb58 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!DbgBreakPoint:
77b2f9fc cc              int     3
0:001> bp contoso!MyFunction
0:001> r
eax=7ece8000 ebx=00000000 ecx=00000000 edx=77b9dbeb esi=00000000 edi=00000000
eip=77b2f9fc esp=0120f864 ebp=0120f890 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!DbgBreakPoint:
77b2f9fc cc              int     3
0:001> g
Breakpoint 0 hit
eax=00f4fc84 ebx=00000000 ecx=00000000 edx=00000000 esi=af9208a3 edi=75cbb8ad
eip=00d310f3 esp=00f4fc6c ebp=00f4fca0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
contoso!MyFunction:
0:001> r
00d310f3 55              push    ebp
eax=00f4fc84 ebx=00000000 ecx=00000000 edx=00000000 esi=af9208a3 edi=75cbb8ad
eip=00d310f3 esp=00f4fc6c ebp=00f4fca0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
contoso!MyFunction:
00d310f3 55              push    ebp
0:001> q
```

Raymond Chen

Follow