What happens if I mutate a string in a p/invoke?

devblogs.microsoft.com/oldnewthing/20181128-00

November 28, 2018

Raymond Chen

When it comes time to p/invoke to a Win32 function that writes to a string buffer, everybody uses a `StringBuilder` class to receive the string.

But could we just use a `string`? I mean, we can still allocate a buffer for the `string` and then ask Win32 to fill the buffer.

```
// Code in italics is wrong
[DllImport("user32.dll", CharSet = CharSet.Unicode)]
extern public static int GetKeyboardLayoutName(string buffer);

var buffer = new String('\0', 9);
GetKeyboardLayoutName(buffer);
```

I mean, sure C# strings are immutable, but that just means that you can't mutate them from within the C# language. The runtime will allocate some memory for the string, and that memory will be writable in practice, so the `GetKeyboardLayoutName` function will be able to write to it, and bingo, the results are in the string! What could possibly go wrong?

What could possibly go wrong is that you're violating the rules of the language, namely that strings are immutable.

Passing a `string` via platform invoke means that the runtime will pass a null terminated C-style string that it expects to be read from. If the native function ends up writing to it, then what happens next is unpredictable.

For example, the platform invoke code is not required to pass a pointer to the internal string buffer. It might copy the string contents to a temporary buffer and pass a pointer to that temporary buffer. If the native function modifies that buffer, the runtime won't try to copy the results back to the original string buffer, because the runtime doesn't expect the native function to modify the buffer at all.

In fact, there is a case where this temporary buffer is guaranteed to exist: when the function being called takes an ANSI string. Because the raw internal string buffer is in the wrong format, namely Unicode (UTF-16LE), so the CLR needs to create a temporary ANSI version of the string.

Even if you manage to cajole the runtime into passing a pointer to the raw string buffer, the runtime doesn't expect the string to change, and if the native function doesn't fill the entire buffer, the runtime won't notice. You'll have a string with extra junk in it.

And the fact that you're mutating what is supposed to be immutable is going to cause its own problems:

```
using System;
using System.Runtime.InteropServices;
using System.Collections.Generic;

// Code in italics is wrong
class Program
{
  [DllImport("user32.dll", CharSet = CharSet.Unicode)]
  extern public static int GetKeyboardLayoutName(string buffer);

  public static void Main()
  {
    var hash = new Dictionary<string, int>();
    string buffer = new string('\0', 10);
    hash[buffer] = 2;
    GetKeyboardLayoutName(buffer);

    string buffer2 = new string('\0', 10);
    Console.WriteLine(hash[buffer2]);
  }
}
```

Strings are immutable, and therefore they can safely be used as keys in dictionaries. But in the above example, we are mutating the string that is being used as a key, which messes up the dictionary. Not only did the item's key change, but nobody can find the new key because its hash code is different, so it's in the wrong bucket in the dictionary.

Basically, you created a dictionary that violates the dictionary invariants.

Another case where mutating a string violates the rules of C# can be found in the reference source for the `String.CompareOrdinalHelper` method. The method compares two characters at a time, and once it finds a difference, it looks to see which character of the pair is the one that caused the strings to be different. This assumes that strings are immutable.

But if you mutate the internal buffer from another thread, it's possible that the first loop finds a pair of characters which don't match, but when it goes to see which of the pair it is, the contents of the buffer changed, and now the characters match after all. Assertion failure. Function returns incorrect result.

If you are passing a string buffer that native code will write to, use a `StringBuilder`. That's what it's for.

Raymond Chen

Follow