Reinterpreting the bits of a 64-bit integer as if they were a double-precision floating point number (and vice versa)

devblogs.microsoft.com/oldnewthing/20150622-00

June 22, 2015

Raymond Chen

Today's Little Program takes a 64-bit integer and reinterprets its physical representation as a double-precision floating point number.

```
using System;

class Program
{
 static double ReinterpretAsDouble(long longValue)
 {
  return BitConverter.ToDouble(BitConverter.GetBytes(longValue), 0);
 }

 static long ReinterpretAsLong(double doubleValue)
 {
  return BitConverter.ToInt64(BitConverter.GetBytes(doubleValue), 0);
 }

 static void Main()
 {
  Console.WriteLine(ReinterpretAsDouble(0x4000000000000000));
  Console.WriteLine("{0:X}", ReinterpretAsLong(2.0));
 }
}
```

Our first attempt uses the `BitConverter` class to convert the 64-bit integer to an array of bytes, and then parses a double-precision floating point number from that byte array.

Maybe you're not happy that this creates a short-lived `byte[]` array that will need to be GC'd. So here's another version that is a little sneakier.

```
using System;
using System.Runtime.InteropServices;

class Program
{
 [StructLayout(LayoutKind.Explicit)]
 struct LongAndDouble
 {
  [FieldOffset(0)] public long longValue;
  [FieldOffset(0)] public double doubleValue;
 }

 static double ReinterpretAsDouble(long longValue)
 {
  LongAndDouble both;
  both.doubleValue = 0.0;
  both.longValue = longValue;
  return both.doubleValue;
 }

 static long ReinterpretAsLong(double doubleValue)
 {
  LongAndDouble both;
  both.longValue = 0;
  both.doubleValue = doubleValue;
  return both.longValue;
 }
 ...
}
```

This version creates a structure with an unusual layout: The two members occupy the same physical storage. The conversion is done by storing the 64-bit integer into that storage location, then reading the double-precision floating point value out.

There's a third method that involves writing the 64-bit integer to a memory stream via `BinaryWriter` then reading it back with `BinaryReader`, but this is clearly inferior to the `BitConverter` so I didn't bother writing it up.

Update: Damien points out that this functionality already exists in the BCL: `BitConverter.DoubleToInt64Bits` and `BitConverter.Int64BitsToDouble`. But there doesn't appear to be a `BitConverter.FloatToInt32Bits` method, so the techniques discussed above are not completely useless.

Exercise: Why did I have to initialize the `doubleValue` before writing to `longValue`, and vice versa? What are the implications of the answer to the above question? (Yes, I could have written `LongAndDouble both = new LongAndDouble();`, which automatically zero-initializes everything, but then I wouldn't have had an interesting exercise!)

Raymond Chen

Follow