Why does the CLR report a NullReferenceException even if the referenced access is not exactly the null pointer?

devblogs.microsoft.com/oldnewthing/20130809-00

August 9, 2013

Raymond Chen

We saw some time ago that before invoking a method on an object, the CLR will generate a `cmp [ecx], ecx` instruction to force a null reference exception to be raised if you are trying to invoke a method on a null reference.

But why does the CLR raise a `NullReferenceException` if the faulting address is almost but not quite zero?

```
class Program {
 public static unsafe void Main() {
  byte *addr = (byte*)0x42;
  byte val = *addr;
 }
}
```

When run, this program raises a `NullReferenceException` rather than an `AccessViolationException`. On the other hand, if you change the address to `0x80000000`, then you get the expected `AccessViolationException`.

With a little bit of preparation, the CLR optimizes out null pointer checks if it knows that it's going to access the object anyway. For example, if you write

```
class Something {
 int a, b, c;
 static int Test(Something s) { return s.c; }
}
```

then the CLR doesn't need to perform a null pointer test against `s` before trying to read `c`, because the act of reading `c` will raise an exception if `s` is a null reference.

On the other hand, the offset of `c` within `s` is probably not going to be zero, so when the exception is raised by the CPU, the faulting address is not going to be exactly zero but rather some small number.

The CLR therefore assumes that all exceptions at addresses close to the null pointer were the result of trying to access a field relative to a null reference. Once you also ensure that the first 64KB of memory is always invalid, this assumption allows the null pointer check optimization.

Of course, if you start messing with unmanaged code or unsafe code, then you can trigger access violations near the null pointer that are not the result of null references. That's what happens when you operate outside the rules of the managed memory environment.

Mind you, version 1 of the .NET Framework didn't even have an `AccessViolation-Exception`. In purely managed code, all references are either valid or null, so version 1 of the .NET Framework assumed that any access violation was the result of a null reference. There's even a configuration option you can set to force newer versions of the .NET Framework to treat all access violations as null reference exceptions.

Exercise: Respond to the following statement: "Consider a really large class (more than 64KB), and I access a field near the end of the class. In that case, the null pointer optimization won't work because the access will be outside the 64KB range. Aha, I have found a flaw in your design!"

Raymond Chen

Follow