On the ways of finding a dispatcher for the current thread in the Windows Runtime

devblogs.microsoft.com/oldnewthing/20201217-00

December 17, 2020

Raymond Chen

If you're working with XAML in the Windows Runtime, there's a good chance that you are going to have to deal with dispatchers. A dispatcher is an object that manages a queue of work items and runs (dispatches) them in some order, usually to a dedicated thread. For UWP XAML apps, the dispatcher is a `CoreDispatcher`. There's also a different dispatcher known as a `DispatcherQueue` which is used by Direct3D and Composition.

If you're writing general-use code, you may want to obtain the dispatcher for the current thread, so you can dispatch work back to thread later.

To obtain the `DispatcherQueue` for the current thread, you can call the `DispatcherQueue.GetForCurrentThread` static method. It returns null if the current thread is not controlled by a `DispatcherQueue`.

Getting the `CoreDispatcher` is trickier, since there is no obvious way to get one. You'll have to get it indirectly.

One way is to ask the misleadingly-named `CoreApplication.GetCurrentView` static method for the `CoreApplicationView` that belongs to the current thread, and then retrieve the `CoreApplicationView.Dispatcher` property.

I say that the `GetCurrentView` method is misleadingly named because the sense of "current" is not "currently on the screen" or "currently has focus", but rather "belonging to the current thread". The word *current* refers to the thread context, not to the view. It's confusing because the word *thread* appears nowhere in the method name!

There's a practical downside of the `GetCurrentView` method: If the current thread does not have an associated `CoreApplicationView`, the method fails, which gets turned by the languages projection into an exception. You have to catch the exception, which is not only annoying, but it's also distracting, because debuggers often break whenever an exception is thrown (even if the exception is ultimately caught). Developers using your library have to disable "break on exception", which may conflict with their preferred settings, especially if the developer is trying to track down the source of an exception that is causing problems.

Even if everybody agrees not to break on exceptions, you still get debug spew about the exception that was thrown and subsequently caught. If a developer is trying to chase down a problem, and they see some debug spew about an exception, they may begin to suspect that your library is somehow the cause of their problem, even though it has nothing to do with their problem.

Fortunately, there's an alternative: You can use the static `CoreWindow.GetForCurrentThread` method, which attempts to find the `CoreWindow` for the current thread, and then retrieve the `CoreWindow.Dispatcher` property. This gives you the same dispatcher that you would have gotten from the `CoreApplicationView`, but the advantage here is that `CoreWindow.GetForCurrentThread` returns `null` if the current thread doesn't have a `CoreWindow`, rather than throwing an exception.

If you want to sniff around for a dispatcher, you can do this:

```
if (auto dispatcherQueue = DispatcherQueue.GetForCurrentThread()) {
  ... do something with the dispatcherQueue ...
} else if (auto window = CoreWindow.GetForCurrentThread()) {
  ... do something with window.Dispatcher() ...
} else {
  ... deal with the fact that there is no dispatcher ...
}
```

Raymond Chen

Follow