Parsing ETL traces yourself, part 2: The EventLogReader

devblogs.microsoft.com/oldnewthing/20201210-00

December 10, 2020

Raymond Chen

The CLR `System.Diagnostics.Eventing.Reader` namespace contains classes for parsing classic ETL events. You create an `EventLogReader` object, giving it the path to the ETL file, and then call `ReadEvent` to extract the next event from the event log.

Here's a simple program that parses an event log looking for start and stop events from a specific provider and event ID, and calculating the net time:

```
using System;
using System.Diagnostics.Eventing.Reader;

class Program
{
  static readonly Guid MyProviderId = new Guid("...");
  const int beginId = 3141;
  const int endId = 3142;

  static public void Main(string[] args)
  {
    var filename = args[0];
    using (var log = new EventLogReader(filename, PathType.FilePath)) {
      string currentScenario = string.Empty;
      DateTime beginTime = DateTime.MinValue;
      TimeSpan totalDuration = TimeSpan.Zero;

      EventLogRecord data;
      while ((data = (EventLogRecord)log.ReadEvent()) != null) {
        if (data.ProviderId == MyProviderId) {
          if (data.Id == beginId) {
            if (beginTime != DateTime.MinValue) {
              throw new Exception("Overlapping events not supported");
            }
            currentScenario = (string)data.Properties[0].Value;
            beginTime = data.TimeCreated.Value;
          }
          else if (data.Id == endId) {
            if (beginTime != DateTime.MinValue) {
              if (currentScenario != (string)data.Properties[0].Value) {
                throw new Exception("Unsupported event sequence");
              }
              var duration = data.TimeCreated.Value - beginTime;
              System.Console.WriteLine($"{currentScenario},{duration}");
              totalDuration += duration;
              beginTime = DateTime.MinValue;
            }
          }
        }
      }
      System.Console.WriteLine($"Total,{totalDuration}");
    }
  }
}
```

We start by opening an `EventLogReader` on our event log file. The `EventLogReader` can connect to an active event log, but we're using it here for offline analysis.

The `currentScenario` variable remembers the name of the current scenario being tracked.

The `beginTime` variable remembers when the current begin/end pair began. It is `MinValue` if nothing is being tracked.

The `totalDuration` variable accumulates the total time consumed by our begin/end pairs so far.

We pull events from the event log by calling `ReadEvent`. We check if the event was generated by our provider. We could have checked the `ProviderName` property, but I'm going for `ProviderId` because it avoid case-sensitivity questions, ensures uniqueness in the face of somebody else coincidentally choosing the same name, and appeals to me because I like to think comparing two `Guid` objects is faster than comparing two strings.

Once we verify that the event belongs to our provider, we check the event IDs. If it's our *begin* event, then we remember the event timestamp and the scenario name, which is a string provided as the first (and only) event payload.

There are methods and properties on the `EventLogRecord` that let you learn about the type of payload attached to the event, but I'm just hard-coding specific knowledge of the way my provider generates the events. This is common when doing event processing. After all, you generated the event. You should know what's in it.

I add a little defensive check that there are not two *begin* events in a row. This is a hole in my processing: If there are two *begin* events in a row, then they should both be tracked, and the corresponding *end* events should be matched up against the *begin* events. This is a quick-and-dirty program, so I just complain if overlapping events are discovered, so I know that I need to go back and add support for that.[1]

If we get an *end* event, we check if there is active sequence, and if so, verify that the *end* event matches the *begin*.[1] Assuming everything passes, we print the elapsed time for that sequence, and accumulate it for final reporting.

Note that we do not complain if we see an *end* event without a *begin*. This can happen if the trace began while a sequence was already in progress.

When all the events are done, we print the cumulative time.

The `EventLogReader` is a wrapper around the native functions in the `winevt.h` header file, like `EvtOpenSession` and `EvtNext`. It's handy, but only if you are interested in class for reading classic manifested events. It doesn't understand the ad-hoc events generated by TraceLogging. The native header file for decoding TraceLogging events is `tdh.h`, but the `EventLogReader` doesn't use it.

Oh no, what are we going to do?

Relief arrives next time.

[1] It's important to check for these cases, even if you don't support them, because you may encounter a trace that includes overlapping events, and if you didn't check for that case, the program would generate incorrect results and cause you to draw the wrong conclusions. Bad data is worse than no data.

Raymond Chen

Follow