WriteAsync .NET

Testing, coding, in that order.

Using PLA.dll to collect ETW traces

As demonstrated previously, PLA.dll allows you to collect perf counter logs. It can also be used to collect ETW traces.

To collect traces, you need to add one or more trace data providers to your collector. Trace data providers are identified by a GUID known as the ETW provider ID. Windows comes with many built-in providers. In addition, third-party services and applications will often register their own providers on installation. To get a list of provider names and IDs currently registered on your system, open an elevated command prompt and run `logman.exe query providers`.

To represent a provider, I added a `ProviderInfo` class to the sample. (Remember, all this code is available on the GitHub PlaSample project.)

```
public class ProviderInfo
{
    public ProviderInfo(Guid id)
    {
        this.Id = id;
    }

    public Guid Id { get; private set; }

    public uint? Level { get; set; }

    public ulong? KeywordsAny { get; set; }

    public ulong? KeywordsAll { get; set; }
}
```

The level indicates the highest trace level to collect; ETW typically uses levels 1-5 (critical, error, warning, informational, verbose). The keywords are 64-bit masks which can be used as custom filters (e.g. the CLR defines many keywords in its trace provider), where "any" means "match events with any of these bits set" and "all" means "only match events with all these bits."

For the trace collector, I have exposed many options to control the log file size, whether to use a circular buffer (i.e. keep overwriting the log file with newer events once it reaches a max size), how big the event buffer should be, and so on. (It should be noted that many of these work with perf counter collectors as well, but were omitted for simplicity.)

```
public class TraceCollectorInfo
{
    public TraceCollectorInfo(string name)
    {
        this.Name = name;
        this.Providers = new List<ProviderInfo>();
    }

    public string Name { get; private set; }

    public string OutputPath { get; set; }

    public uint? BufferSizeInKB { get; set; }

    public bool? Circular { get; set; }

    public TimeSpan? FlushTimer { get; set; }

    public TimeSpan? MaxDuration { get; set; }

    public uint? MaxSizeInMB { get; set; }

    public uint? MaximumBuffers { get; set; }

    public uint? MinimumBuffers { get; set; }

    public bool? Segmented { get; set; }

    public IList<ProviderInfo> Providers { get; private se
}
```

The PLA code to create a trace collector is fairly similar to the perf counter collector code. The differences are mostly in the number of properties to set and the way the provider list is built up (note the helper methods to simplify the optional value processing):

```
public ICollectorSet Create()
{
    // Data collector set is the core abstraction for coll
    DataCollectorSet dcs = new DataCollectorSet();

    // Set base folder to place output files.
    dcs.RootPath = this.OutputPath;

    // Create a data collector for traces.
    ITraceDataCollector dc = (ITraceDataCollector)dcs.Data
    dc.name = this.Name + "_DC";
    dcs.DataCollectors.Add(dc);

    // Set output file name to use a pattern, as described
    // http://msdn.microsoft.com/en-us/library/windows/des
    dc.FileName = this.Name;
    dc.FileNameFormat = AutoPathFormat.plaPattern;
    dc.FileNameFormatPattern = @"\-yyyyMMdd\-HHmmss";

    // Set various values (if present)
    SetValue(dc, this.BufferSizeInKB, (d, v) => d.BufferSi
    SetValue(dc, this.Circular, (d, v) => d.LogCircular =
    SetValue(dc, this.FlushTimer, (d, v) => d.FlushTimer =
    SetValue(dc, this.MaximumBuffers, (d, v) => d.MaximumB
    SetValue(dc, this.MinimumBuffers, (d, v) => d.MinimumB
    SetValue(dc, this.MinimumBuffers, (d, v) => d.MinimumB

    SetValue(dcs, this.MaxDuration, (d, v) => d.SegmentMax
    SetValue(dcs, this.MaxSizeInMB, (d, v) => d.SegmentMax
    SetValue(dcs, this.Segmented, (d, v) => d.Segment = v)

    // Build up the list of providers.
    foreach (ProviderInfo providerInfo in this.Providers)
    {
        TraceDataProvider provider = dc.TraceDataProviders
        dc.TraceDataProviders.Add(provider);

        provider.Guid = providerInfo.Id;
        AddValue(provider.KeywordsAll, providerInfo.Keywor
        AddValue(provider.KeywordsAny, providerInfo.Keywor
        AddValue(provider.Level, providerInfo.Level);
    }

    // Now actually create (or modify existing) the set.
    dcs.Commit(this.Name, null, CommitMode.plaCreateOrModi

    // Return an opaque wrapper with which the user can co
    return new CollectorSetWrapper(dcs);
}

private static void SetValue<TClass, TValue>(TClass c, TVa
{
    if (v.HasValue)
    {
        setValue(c, v.Value);
    }
}

private static void AddValue<TValue>(IValueMap map, TValue
{
    if (v.HasValue)
    {
        map.Add(v.Value);
    }
}
```

Now some sample code to show how to use the trace collector. This example collects kernel process traces for about five seconds, creating new files after every one second. Remember to run this elevated:

```
private static void CreateTraceCollector()
{
    TraceCollectorInfo info = new TraceCollectorInfo("MyTr

    info.BufferSizeInKB = 64;
    info.Segmented = true;
    info.MaxDuration = TimeSpan.FromSeconds(1.0d);
    info.OutputPath = Environment.CurrentDirectory;

    // Microsoft-Windows-Kernel-Process
    Guid providerId = new Guid("{22FB2CD6-0E7B-422B-A0C7-2

    info.Providers.Add(new ProviderInfo(providerId) { Leve

    ICollectorSet collector = info.Create();
    collector.Start();

    Thread.Sleep(5000);

    collector.Stop();

    collector.Delete();
}
```

After the app finishes, you should see files like the following:

`MyTraces-20140101-125602.etl`
`MyTraces-20140101-125603.etl`
`...`

These files can be decoded by Windows Event Viewer or with tools like tracerpt.exe.

Using `tracerpt.exe [file.etl] -o [file.xml]`, you can dump the traces to a human-readable XML file. The file will contain a sequence of `<Event>` elements such as the following:

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/eve
  <System>
    <Provider Name="Microsoft-Windows-Kernel-Process" Guid
    <EventID>8</EventID>
    <!-- ... -->
  </System>
  <EventData>
    <Data Name="ProcessID">    1788</Data>
    <Data Name="ThreadID">  411224</Data>
    <Data Name="OldPriority">16</Data>
    <Data Name="NewPriority">10</Data>
  </EventData>
  <RenderingInfo Culture="en-US">
    <Level>Information </Level>
    <Opcode>Info </Opcode>
    <Keywords>
      <Keyword>WINEVENT_KEYWORD_CPU_PRIORITY</Keyword>
    </Keywords>
    <Task>CpuPriorityChange</Task>
    <Message>CPU priority of thread 411224 in process 1788
    <Channel>Microsoft-Windows-Kernel-Process/Analytic</Ch
    <Provider>Microsoft-Windows-Kernel-Process </Provider>
  </RenderingInfo>
</Event>
```

By Brian Rogers | 1 January, 2014 | diagnostics | No Comments |

← Threads don't scale   Real-time trace sessions with TraceEvent →

Leave a Reply