Les Chroniques du Campagnol
Season 2 Episode 7
Mercator - Determinism

April 30, 2022

This time we'll try to make error reports a lot more useful.

1 User Needs

As developers, what we need is a way to obtain clean error logs to help debug the application. We want as much information as possible, and especially technical information about the source code where the error occurred.

As end users, what we need is a straightforward way of sending clean error logs to the developers. We don't want to go to the Windows Event Log (which does not even exist on Linux), we just want simple and clean explanations such as "go to that directory and mail me all the files less than 3 days old."

2 Design

We will store each crash report in its own file. Each file shall contain enough information to investigate the crash. Crash reports can be searched with `grep`, archived with `tar`, attached to an e-mail, etc.

3 CrashReportException

The application shall create crash reports into a specific directory. We can then simply direct the user at sending the corresponding file. Let's do that in the most straightforward way imaginable...

```
public class CrashReportException : Exception
{
    public CrashReportException (Exception Inner, FileInfo CrashFile)
        : base ($"Crash {Inner.Message} reported in {CrashFile}", Inner)
    {
        this.CrashFile = CrashFile;
    }

    public CrashReportException (Exception Inner, DirectoryInfo Directory)
        : this (Inner, Directory.UniqueFile (FileNamePattern).WriteContents (Describe
           (Inner)))
    {
    }

    public override string ToString () => this.Message;
}
```

Yup, that's it. Well no, it's not completely it, but almost. I defined a `CrashReportException` class that wraps the actual exception we want to report. The new exception will create a crash dump (the `Describe` function that I cleverly failed to include...) then make sure that its own stack is not shown too easily when code tries to log the exception.

Of course, that only works a the very top level. Once we wrap the true exception in a `CrashReportException`, we forfeit the chance of catching specific exceptions in the code. But that's OK as this contraption is only used at the highest level, when we have stopped trying to recover from the error.

4 Avoiding reporting the description of a crash dump

We want to avoid wrapping a `CrashReportException` inside a `CrashReportException` so instead of systematically create a new exception, you call the `From` factory function:

```
public class CrashReportException : Exception
{
    public static CrashReportException From (Exception Inner, DirectoryInfo Directory)
    {
        if (Inner is CrashReportException e)
        {
            return e;
        }
        else
        {
            return new CrashReportException (Inner, Directory);
        }
    }
}
```

5 Wrapping an async task in a Crash Dump catcher

To do so once, it's easy:

```
try
{
    await MyFunction ();
}
catch (Exception e)
{
    throw CrashReportException.From (e, MyDirectory);
}
```

Whoops, I once again forgot to handle `OperationCanceledException`.

5.1 A little functional programming

But we want it to be a little more generic so that we can take any async function and wrap it in a handler:

```
public static class TaskExtensions
{
    public static async Task ReportFailure (this Task t, ILogger Logger)
    {
        try
        {
            await t;
        }
        catch (OperationCanceledException)
        {
            throw;
        }
        catch (Exception e)
        {
            Logger.LogError ("Background task ended in error: {0}", e);
        }
    }

    public static async Task CrashDump (this Task t, DirectoryInfo Destination)
    {
        try
        {
            await t;
        }
        catch (OperationCanceledException)
        {
            throw;
        }
        catch (Exception e)
        {
            throw await CrashReportException.FromAsync (e, Destination);
        }
    }
}
```

Notice that we call `FromAsync` and not `From`. The logic is the same but the crash dump file is generated in an async function...

5.2 Wrapping background tasks automatically

Remember that the background worker maintains a list of tasks in a variable. There is a function `RegisterTask` so that we can add more tasks, and initially it did something quite simple:

```
public void RegisterTask (Task t)
{
    _Tasks.Add (t);
}
```

Let's make it decorate background tasks automatically using the extension methods:

```
protected virtual Task DecorateTask (Task t)
{
    return t
        .CrashDump (LogDirectory)
        .ReportFailure (Logger);
}

public void RegisterTask (Task t)
{
    _Tasks.Add (DecorateTask (t));
}

public void RegisterTaskRange (IEnumerable<Task> t)
{
    _Tasks.AddRange (t.Select (DecorateTask));
}
```

5.3 Not caring about that WhenAny issue anymore

The nice thing is that doing so solves two of the three problems I mentioned last time: if the task exits with an exception, the `ReportFailure` wrapper will immediately report that in the error log; but before that, the `CrashDump` wrapper will replace the exception with a reference to its crash dump file. This will happen as soon as the task exits, not when it is removed from the queue.

So we can care less about the dangling tasks (not that much, because they are still hanging around, consuming memory and paving the way to a memory leak...) so that we can re-bury that issue under the carpet. For now.

6 Creating the crash dump

I suspensefully left the `Describe` function undescribed. Well, it's simply a function that returns a description of the exception that we want to report. But instead of returning a mere string, it returns each line individually in an `IEnumerable`.

And that is done on purpose so I can use `yield` when writing the function:

```
public static IEnumerable<string> Describe (Exception Inner)
{
    yield return "Crash report";
    yield return "============";
    yield return "Error information";
    yield return "------------------";
    yield return $"Current Date/Time (local): {DateTime.Now}";
    yield return $"Current Date/Time (UTC):   {DateTime.UtcNow}";
    // ...
    yield return "Build information";
    yield return "------------------";
    yield return $"Build Directory:           {BuildInformation.BuildDirectory}";
    yield return $"Local Build:               {BuildInformation.IsLocalBuild}";
    yield return $"GIT commit:                {BuildInformation.GitCommit}";
    foreach (GitChange c in BuildInformation.GitChanges)
    {
        yield return $"GIT changed file:          {c.FileName}";
    }

    yield return "";
    yield return "Stack trace";
    yield return "-----------";
    yield return Inner.ToString ().Replace (
        ") in " + BuildInformation.BuildDirectory.IfEmpty
            ("3585f16a-b44e-441b-bef7-88c594fba6c6") + Path.DirectorySeparatorChar,
        ") in ");
}
```

What??? Git changed file???

Yes, I managed to include the exact state of the source code inside resources, so that when receiving a crash dump from a user, I'm theoretically able to rebuild the exact source code corresponding to its build, even if the executable was built while I had some files not commited into git. That is why this episode is named DETERMINISM.

Yeah! Great idea, Einstein! Give the source code to the user!

I don't. I only give them the diff. And it is encrypted. But that is another story. And anyway there are tools to retrieve an abundant quantity of information about the source code.

And what about that GUID in the stack trace?

I do not put a GUID in the stack trace, it's just a dirty hack to ensure that the value `BuildDirectory` is not empty and never appears anywhere in a stack trace. The trick is that the stack trace contains absolute file paths, and I want relative file paths. And as I said it's a dirty hack, we should do something better. But that's for another episode.

7 What's next

Error reporting is good enough for now, let's build more tools before we revisit this topic.