File Tracker–Based Incremental Build

Incremental Build

Incremental build of a project means reusing the results of the preceding build while processing only the changes made to the project since the preceding build. An important corollary to this is that if nothing changed since the last build, then no processing occurs (other than some checks). This saves time over a complete build since we don't have to process items that haven't changed. MSBuild supports an accurate incremental build.

Recast in MSBuild terms, incremental build reduces to running only those tasks whose inputs have changed since the last build. Such a scheme can be implemented only when (1) we know the complete set of input files a task consumes; and (2) we know whether an input file has been modified since the last build. The second problem is easy to solve by comparing the current timestamp values of the input files with the values for the preceding build. The first problem, on the other hand, is more difficult. To understand why, consider the CL task that compiles .cpp files in a project to .obj binaries. The CL task needs to run whenever any of the .cpp files is changed directly or if any of the .h files included (or even the .h files that these in turn include, and so on) is touched. The .h files referred by the .cpp files may not be part of the project, but they are still consumed by the CL task, so there is no one place where the list of all such files is statically stored. So how does MSBuild solve the problem of finding the complete set of input files to a task? The answer is File Tracker.

File Tracker

File Tracker, a new feature in MSBuild 4.0, is a "file up-to-date check" infrastructure that powers MSBuild's incremental build feature. File Tracker, as the name implies, tracks read and write accesses to files made by a process. Most of the Visual C++ tools, such as cl.exe and link.exe, are run in their own separate processes by the corresponding tasks. The tool tasks don't create the tool process directly. They create the File Tracker executable—the Tracker.exe process—and hand over the responsibility of executing the tool to it. File Tracker wraps these tool processes and keeps track of all the files read by and written to by the process (and any other process it spawns) in a lightweight manner. File Tracker achieves this by detouring several Win32 file access application programming interface (API) calls such as CreateFile and CopyFile, recording the access, and then passing on the call to the operating system normally. This eavesdropping means that MSBuild doesn't have to depend on the user supplying the list of inputs and outputs that a task consumes (as was the case in MSBuild 3.5). File Tracker then writes the list of these file paths in tracking log files (with the extension .tlog) in the project's intermediate directory. Tlog files are in a human-readable format although in the normal course of events, the only reason that you should ever need to look at the log files themselves is to satisfy your curiosity.

When a task is scheduled to run during an incremental build, MSBuild compares the timestamp-on-disk for the input files to the timestamp-on-disk for the output files. A set of input and output files is judged to be up to date if the oldest output is newer than the newest input. If the set of files is not up to date, the task is run; otherwise, it is not. Notice that we only used timestamps on disk for checking the up-to-date status. Because of this, no timing information is ever written to the tlog files.

One way to get a task to participate in an incremental build is to make it File Tracker–enabled. Architecturally, this means that the class that implements the task behavior needs to use the File Tracker API to submit its tool for tracking. Note that the File Tracker API has been published by Microsoft for use by anyone. See the class *Microsoft.Build.Utilities.FileTracker* in the Microsoft.Build.Utilities.v4.0.dll assembly. Instead of interacting with the File Tracker API, you can simply derive from the *Microsoft.Build.CPPTasks.TrackedVCToolTask* class (available in the Microsoft.Build.CPPTasks.Common.dll assembly). By abstracting out most of the File Tracker interaction, this class makes it easier to create your own File Tracker–enabled task that invokes an external tool. Be aware of the fact that although this class is public, the details of inheriting from this class are not straightforward. Most likely, it will become easier in later editions of MSBuild. In Visual C++ 2010, all tool tasks, like CL and Link (the Visual C++ task that links object libraries), are File Tracker–enabled. Hence, they are capable of participating in an incremental build. Other tasks are not. In fact, if all tasks were File Tracker–enabled by default, then we could never write simple logging tasks such as one that would log some message during a build. Such tasks have no inputs and hence will always be up to date and never run a second time.

When building using MSBuild.exe from the command line, an up-to-date check is done on the level of the individual tools—CL will look only at its tracking logs, RC will look only at its tracking logs, and so on. If CL skips building, that has no impact on whether or not RC skips (except in the case of tools that depend on each other's outputs—for example, if CL does not skip, Link will also be forced to rebuild because the obj files will now be out of date).

In the IDE, there is also a "fast up-to-date check," which does a project-level check of the up-to-date status of the solution, and only spawns a project build if the up-to-date check fails. This up-to-date check also (among other checks) uses the File Tracker API to do a very simple all reads vs. all writes comparison; if the project is determined to be at all out of date, it triggers a build and the decision of whether to skip falls back to the more fine-grained dependency checking provided by the individual tasks.

We also need to mention that it is up to the individual tool tasks to determine how they use the File Tracker API to implement an incremental build. It is possible to do a very fine-grained incremental build, as is demonstrated by CL. The cl.exe tool (invoked by the CL task) produces one .obj file for each .cpp file and its imports. The linker combines the .obj files into the final dll/exe binary. Now, if a build is invoked on a project and only one .cpp file was changed, the CL task passes *only* this changed .cpp file to the cl.exe compiler to produce the corresponding .obj file. Thus, CL task runs optimized even in the case when some of its inputs have changed. The Link task on the other hand always processes all its inputs.

Trust Visual C++ Incremental Build

Visual C++ supported incremental builds even in earlier editions. It used the dependency checker infrastructure that stored relationships between the source files—for example, the cpp files, class definitions, and header files—in a database file (.idb) on the first compile and used it to make decisions on future incremental builds. However, it has never been very reliable. This was further aggravated by the fact that VCBuild did not have good logging capabilities and it was very hard to tell what caused a rebuild when the user didn't expect it to happen. And if the user did figure it out somehow, it was not clear at all how to fix it as the user had very little influence on how VCbuild worked. However, because of the way that File Tracker captures input and output file information by automatically working behind the scenes, incremental builds in Visual C++ 2010 are extremely reliable. Incremental builds in Visual C++ 2010 are also way faster than the incremental builds in Visual C++ 2008, in part because MSBuild checks timestamps in parallel.

We cannot overemphasize the fact that in Visual C++ 2010, incremental builds can be *trusted*. That means that you can use them for real builds, not just ad-hoc builds, which can improve your productivity considerably in some cases. If you're doing clean builds (a "rebuild"), question why. There's rarely a good reason to do so.

Troubleshooting

Say that an incremental build (in the IDE or command line) is taking longer than expected. From the build log file, you see that a tool task that you did not expect to run has actually run. This means that one of the build input files for the tool had changed since the last build. How do you go about knowing the exact files that changed? The answer is by looking in the same build log file. When the verbosity level is cranked up to "detailed" or "diagnostic," detailed information about the task (and target) execution will be written to the log files. You will find that not only can you get to know about the files that were found to be modified since the last build, you can also know about the .tlog files that were read by the tool task to determine the up-to-date status.

Property Sheets

Property sheets are a powerful way of sharing settings among multiple projects, which relieves you from defining those settings in each individual project. This is analogous to how header files are used to share type declarations among multiple class files. A Visual C++ property sheet is an MSBuild file that has a .props extension by convention (the extension