Using fibers to simplify enumerators, part 5: Composition

devblogs.microsoft.com/oldnewthing/20050104-00

January 4, 2005

Raymond Chen

Another type of higher-order enumeration is composition, where one enumerator actually combines the results of multiple enumerators. (Everybody knows about derivation, but composition is another powerful concept in object-oriented programming. We've seen it before when building context menus.)

In a producer-driven enumerator, you would implement composition by calling the two enumeration functions one after the other. In a consumer-driven enumerator, you would implement composition by wrapping the two enumerators inside a large enumerator which then chooses between the two based on which enumerator was currently active.

A fiber-based enumerator behaves more like a consumer-driven enumerator, again, with easier state management.

Let's write a composite enumerator that enumerates everything in the root of your C: drive (no subdirectories), plus everything in the current directory (including subdirectories).

```
class CompositeEnumerator : public FiberEnumerator {
public:
 CompositeEnumerator()
   : m_eFiltered(TEXT("C:\\"))
   , m_eCd(TEXT(".")) { }
 LPCTSTR GetCurDir()
    { return m_peCur->GetCurDir(); }
 LPCTSTR GetCurPath()
    { return m_peCur->GetCurPath(); }
 const WIN32_FIND_DATA* GetCurFindData()
    { return m_peCur->GetCurFindData(); }
private:
 void FiberProc();
private:
 FiberEnumerator* m_peCur;
 FilteredEnumerator m_eFiltered;
 DirectoryTreeEnumerator m_eCd;
};
void CompositeEnumerator::FiberProc()
{
 FEFOUND fef;
 m_peCur = &m_eFiltered;
 while ((fef = m_peCur->Next()) != FEF_DONE &&
        fef != FEF_LEAVEDIR) {
  m_peCur->SetResult(Produce(fef));
 }
 m_peCur = &m_eCd;
 while ((fef = m_peCur->Next()) != FEF_DONE) {
  m_peCur->SetResult(Produce(fef));
 }
}
```

Sidebar: Our composite enumeration is complicated by the fact that our `FilteredEnumerator` spits out a `FEF_LEAVEDIR` at the end, but which we want to suppress, so we have to check for it and eat it.

In the more common case where the enumerator is generating a flat list, it would be a simple matter of just forwarding the two enumerators one after the other. Something like this:

```
void CompositeEnumerator2::FiberProc()
{
 Enum(&m_eFiltered);
 Enum(&m_eCd);
}
void CompositeEnumerator2::Enum(FiberEnumerator *pe)
{
 m_peCur = pe;
 FEFOUND fef;
 while ((fef = m_peCur->Next()) != FEF_DONE) {
  m_peCur->SetResult(Produce(fef));
 }
}
```

End sidebar.

You can try out this `CompositeEnumerator` with the program you've been playing with for the past few days. Just change the line in `main` that creates the enumerator to the following:

```
 CompositeEnumerator e;
```

Exercise: Gosh, why is the total so unusually large?

Exercise: How many fibers are there in the program?

Exercise: Draw a diagram showing how control flows among the various fibers in this program.

Before you get all excited about fibers, consider the following:

- Converting a thread to a fiber needs to be coordinated among all the components in the process so that it is converted only once and stays converted until everybody is finished. This means that if you are writing a plug-in that will go into some other process, you probably should avoid fibers, since you don't know what the other components in the process are going to do with fibers.
- Fibers do not completely solve the one-thread-per-connection problem. They do reduce the context switching, but the memory footprint will still limit you to 2000 fibers per process (assuming a 2GB user-mode address space) since each fiber has a stack, which defaults to 1MB.

I think that's enough about fibers for now.

Raymond Chen

Follow