Using fibers to simplify enumerators, part 2: When life is easier for the caller

devblogs.microsoft.com/oldnewthing/20041230-00

December 30, 2004

Raymond Chen

Last time, we looked at how a directory tree enumerator function would have been written if the person writing the enumerator (the producer) got to write the spec. Now let's look at what it would look like if the person consuming the enumerator wrote the spec:

```
#include <windows.h>
#include <shlwapi.h>
#include <stdio.h>
#include <strsafe.h>


enum FEFOUND {
 FEF_FILE,          // found a file
 FEF_DIR,           // found a directory
 FEF_LEAVEDIR,      // leaving a directory
 FEF_DONE,          // finished
};


enum FERESULT {
 FER_CONTINUE,      // continue enumerating
                    // (if directory: recurse into it)
 FER_SKIP,          // skip directory (do not recurse)
};


class DirectoryTreeEnumerator {
public:
  DirectoryTreeEnumerator(LPCTSTR pszDir);


  FEFOUND Next();
  void SetResult(FERESULT fer);
  void Skip() { SetResult(FER_SKIP); }


  LPCTSTR GetCurDir();
  LPCTSTR GetCurPath();
  const WIN32_FIND_DATA* GetCurFindData();
private:
    … implementation …
};
```

Under this design, the enumerator spits out files, and the caller tells the enumerator when to move on to the next one, optionally indicating that an enumerated directory should be skipped rather than recursed into.

Notice that there is no `FER_STOP` result code. If the consumer wants to stop enumerating, it will merely stop calling `Next()`.

With this design, our test function that computes the inclusive and exclusive sizes of each directory is quite simple:

```
ULONGLONG FileSize(const WIN32_FIND_DATA *pwfd)
{
  return
    ((ULONGLONG)pwfd->nFileSizeHigh << 32) +
    pwfd->nFileSizeLow;
}


ULONGLONG TestWalk(DirectoryTreeEnumerator* penum)
{
 ULONGLONG ullSizeSelf = 0;
 ULONGLONG ullSizeAll = 0;
 for (;;) {
  FEFOUND fef = penum->Next();
  switch (fef) {
  case FEF_FILE:
   ullSizeSelf += FileSize(penum->GetCurFindData());
   break;


  case FEF_DIR:
   ullSizeAll += TestWalk(penum);
   break;


  case FEF_LEAVEDIR:
   ullSizeAll += ullSizeSelf;
   printf("Size of %s is %I64d (%I64d)\n",
    penum->GetCurDir(), ullSizeSelf, ullSizeAll);
   return ullSizeAll;


  case FEF_DONE:
   return ullSizeAll;
  }
 }
 /* notreached */
}


int __cdecl main(int argc, char **argv)
{
 DirectoryTreeEnumerator e(TEXT("."));
 TestWalk(&e);
 return 0;
}
```

Of course, this design puts all the work on the enumerator. Instead of letting the producer walking the tree and calling the callback as it finds things, the caller calls Next() repeatedly, and each time, the enumerator has to find the next file and return it. Since the enumerator

returns, it can’t store its state in the call stack; instead it has to mimic the call stack manually with a stack data structure.

```
class DirectoryTreeEnumerator {
public:
 DirectoryTreeEnumerator(LPCTSTR pszDir);
 ~DirectoryTreeEnumerator();


 FEFOUND Next();
 void SetResult(FERESULT fer)
  { m_es = fer == FER_SKIP ? ES_SKIP : ES_NORMAL; }
 void Skip() { SetResult(FER_SKIP); }


 LPCTSTR GetCurDir()
    { return m_pseCur->m_szDir; }
 LPCTSTR GetCurPath()
    { return m_szPath; }
 const WIN32_FIND_DATA* GetCurFindData()
    { return &m_pseCur->m_wfd; }


private:
 struct StackEntry {
  StackEntry *m_pseNext;
  HANDLE m_hfind;
  WIN32_FIND_DATA m_wfd;
  TCHAR m_szDir[MAX_PATH];
 };


 StackEntry* Push(LPCTSTR pszDir);
 void StopDir();
 bool Stopped();
 void Pop();


 enum EnumState {
  ES_NORMAL,
  ES_SKIP,
  ES_FIRST,
 };


 StackEntry *m_pseCur;
 EnumState m_es;
 TCHAR m_szPath[MAX_PATH];
};


DirectoryTreeEnumerator::StackEntry*
DirectoryTreeEnumerator::Push(
    LPCTSTR pszDir)
{
 StackEntry* pse = new StackEntry();
```

```
 if (pse &&
     SUCCEEDED(StringCchCopy(pse->m_szDir,
                 MAX_PATH, pszDir)) &&
     PathCombine(m_szPath, pse->m_szDir,
                  TEXT(“*.*”)) &&
     (pse->m_hfind = FindFirstFile(m_szPath,
       &pse->m_wfd)) != INVALID_HANDLE_VALUE) {
  pse->m_pseNext = m_pseCur;
  m_es = ES_FIRST;
  m_pseCur = pse;
 } else {
  delete pse;
  pse = NULL;
 }
 return pse;
}


void DirectoryTreeEnumerator::StopDir()
{
 StackEntry* pse = m_pseCur;
 if (pse->m_hfind != INVALID_HANDLE_VALUE) {
  FindClose(pse->m_hfind);
  pse->m_hfind = INVALID_HANDLE_VALUE;
 }
}


bool DirectoryTreeEnumerator::Stopped()
{
 return m_pseCur->m_hfind == INVALID_HANDLE_VALUE;
}


void DirectoryTreeEnumerator::Pop()
{
 StackEntry* pse = m_pseCur;
 m_pseCur = pse->m_pseNext;
 delete pse;
}


DirectoryTreeEnumerator::~DirectoryTreeEnumerator()
{
 while (m_pseCur) {
  StopDir();
  Pop();
 }
}
```

```
DirectoryTreeEnumerator::
    DirectoryTreeEnumerator(LPCTSTR pszDir)
 : m_pseCur(NULL)
{
 Push(pszDir);
}


FEFOUND DirectoryTreeEnumerator::Next()
{
 for (;;) {
  /* Anything to enumerate? */
  if (!m_pseCur) return FEF_DONE;


  /* If just left a directory, pop */
  if (Stopped()) {
   Pop();
   m_es = ES_NORMAL;
  }


  /* If accepted a directory, recurse */
  else if (m_es == ES_NORMAL &&
      (m_pseCur->m_wfd.dwFileAttributes &
                    FILE_ATTRIBUTE_DIRECTORY)) {
   Push(m_szPath);
  }


  /* Any more files in this directory? */
  if (m_es != ES_FIRST &&
       !FindNextFile(m_pseCur->m_hfind,
            &m_pseCur->m_wfd)) {
   StopDir();
   return FEF_LEAVEDIR;
  }


  /* Don't recurse into . or .. */
  if (lstrcmp(m_pseCur->m_wfd.cFileName,
                  TEXT(".")) == 0 ||
      lstrcmp(m_pseCur->m_wfd.cFileName,
                  TEXT("..")) == 0 ||
      !PathCombine(m_szPath, m_pseCur->m_szDir,
                  m_pseCur->m_wfd.cFileName)) {
   m_es = ES_NORMAL;
   continue;
  }
```

```
  /* Return this found item */
  m_es = ES_NORMAL; /* default state */
  if (m_pseCur->m_wfd.dwFileAttributes &
                      FILE_ATTRIBUTE_DIRECTORY) {
   return FEF_DIR;
  } else {
   return FEF_FILE;
  }
 }
 /* notreached */
}
```

Yuck-o-rama. The simple recursive function has turned into this horrible mess of state management.

Wouldn't it be great if we could have it both ways? The caller would see a simple enumerator that spits out files (or directories). But the enumerator sees a callback that it can throw files into.

We'll build that next time.

Raymond Chen

Follow