How can I see what files and shares are being accessed remotely, and the general usage pattern for the NetXxx functions

devblogs.microsoft.com/oldnewthing/20130311-00

March 11, 2013

Raymond Chen

Today's Little Program is a command line version of the Shared Folders MMC snap-in. Why? Because it illustrates the usage pattern for the `NetXxx` family of functions. (It's also a clone of the networking portion of the `openfiles` tool.)

The `NetXxx` family of functions generally work like this:

- You pass in some parameters that describe what you want. Server name, that sort of thing.
- You pass a "level" parameter that describes what information you want.
- The function allocates memory to hold the results you requested, and it returns a pointer to that memory through a `bufptr` parameter.
- If the function returns an array, then
  - You can tell the function the maximum number of results you want.
  - The function tells you how much information it returned.
  - If the function did not retrieve all the results (because it exceeded your maximum), it tells you how to get the rest of them.
- When you are finished, you free the memory with `NetApiBufferFree`.

We'll start with the non-array case, since that is much simpler. Suppose you want to get the level 123 information for a Thing.

```
THING_INFO_123 *pinfo123;
if (NetThingGetInfo(pszThing,
                    123, (LPBYTE*)&pinfo123) == NERR_Success)
{
    DoSomethingWith(pinfo123);
    NetApiBufferFree(pinfo123);
}
```

You call the function, passing the desired information level and a pointer to the variable you want to receive the results. You then use the results, and then free them. Let's try it with a simple function to get information about a user.

```
#define UNICODE
#define _UNICODE
#define STRICT
#include <windows.h>
#include <lm.h>
#include <stdio.h>


void PrintProperty(PCWSTR pszProperty, PCWSTR pszValue)
{
 wprintf(L"%ls: %ls\n", pszProperty,
                        pszValue ? pszValue : L"<none>");
}


int __cdecl wmain(int argc, wchar_t **argv)
{
 USER_INFO_10 *pinfo10;
 if (NetUserGetInfo(NULL, L"Administrator", 10,
                    (LPBYTE*)&pinfo10) == NERR_Success) {
  PrintProperty(L"Name", pinfo10->usri10_name);
  PrintProperty(L"Comment", pinfo10->usri10_comment);
  PrintProperty(L"User comment", pinfo10->usri10_usr_comment);
  PrintProperty(L"Full name", pinfo10->usri10_full_name);
  NetApiBufferFree(pinfo10);
 }
 return 0;
}
```

The trickier case is the functions that return arrays of data. In that case, you need to call the functions in a loop, similar to `FindNextFile`, in order to read all the data. But unlike `FindNextFile`, the functions return chunks of data rather than just one entry at a time.

The general pattern goes like this:

```
THING_INFO_123 *pinfo123;
NET_API_STATUS status;
DWORD_PTR resumeHandle = 0;
do {
 DWORD actual, estimatedTotal;
 status = NetThingEnum(pszThing, 123,
                       (LPBYTE*)&pinfo123,
                       MAX_PREFERRED_LENGTH,
                       &actual,
                       &estimatedTotal,
                       &resumeHandle);
 if (status == NERR_Success ||
     status == ERROR_MORE_DATA) {
  for (DWORD i = 0; i < actual; i++) {
   DoSomethingWith(&pinfo123[i]);
  }
  NetApiBufferFree(pinfo123);
 }
} while (status == ERROR_MORE_DATA);
```

The general pattern is to start by calling the data retrieval function. If the function returns with `NERR_Success`, then it means that it was able to get all the information you requested. If the function returns with `ERROR_MORE_DATA`, then it means that it was able to get some of the information you requested. In either of those two cases, it returns the actual number of items retrieved in the `actual` parameter, which you use to read the values out of the results. (It also returns an estimate of the total number of items remaining in the `estimatedTotal` variable, but very few people use that.)

If the return value was `ERROR_MORE_DATA`, then you go back and call the function again to get the next batch of results.

The way the functions can tell whether you're starting a new operation or continuing an old one is via the `resumeHandle` parameter, which must be a pointer to a `DWORD_PTR` variable which the function updates. On the first call, set the `DWORD_PTR` to zero. If the function returns partial results, then it puts an opaque value into the `resumeHandle` so it can remember where it needs to continue. (By comparison, the `FindFirstFile` passes the resume handle as its return value.)

Note that there is no equivalent to `FindClose` when you are finished with the function. If you don't want to retrieve all the results, you just abandon the handle.

```
int __cdecl wmain(int argc, wchar_t **argv)
{
 FILE_INFO_3 *pinfo3;
 NET_API_STATUS status;
 DWORD_PTR resumeHandle = 0;
 do {
  DWORD actual, estimatedTotal;
  status = NetFileEnum(NULL, NULL, NULL, 3,
                       (LPBYTE*)&pinfo3,
                       MAX_PREFERRED_LENGTH,
                       &actual,
                       &estimatedTotal,
                       &resumeHandle);
  if (status == NERR_Success ||
      status == ERROR_MORE_DATA) {
   for (DWORD i = 0; i < actual; i++) {
    PrintProperty(L"Path", pinfo3[i].fi3_pathname);
    PrintProperty(L"User", pinfo3[i].fi3_username);
    if (pinfo3[i].fi3_permissions & PERM_FILE_READ) {
     PrintProperty(L"Access", L"READ");
    }
    if (pinfo3[i].fi3_permissions & PERM_FILE_WRITE) {
     PrintProperty(L"Access", L"WRITE");
    }
    if (pinfo3[i].fi3_permissions & PERM_FILE_CREATE) {
     PrintProperty(L"Access", L"CREATE");
    }
   }
   NetApiBufferFree(pinfo3);
  }
 } while (status == ERROR_MORE_DATA);
 return 0;
}
```

I've been ignoring the parameter known as `prefmaxlen` because you pretty much always pass `MAX_PREFERRED_LENGTH`. The parameter lets you limit how much information is returned at a time, but you nearly always want as much as possible (which is why you nearly always pass `MAX_PREFERRED_LENGTH`). If, for some reason, you want to retrieve only a little bit at a time, you can pass a smaller value as the `prefmaxlen`. Note that `prefmaxlen` is in bytes, not elements, and the size in bytes needs to include the auxiliary data (like the strings), not just the structures. If you pass a custom `prefmaxlen`, then you also have to be prepared to handle the `NERR_BufTooSmall` error code, which means "The value you passed in `prefmaxlen` wasn't big enough to hold even *one* item. You'll have to try again with a bigger buffer size." If you're advanced enough to use a custom buffer size, then you're advanced enough to figure out how to tweak the algorithm to handle it properly.

Note that I have no special knowledge of the `NetXxxx` family of functions. I figured this out by reading the documentation.

Raymond Chen

Follow