Los rincones del API Win32
El caché de WinInet

Y terminamos nuestra serie sobre WinInet, explicando en profundidad qué es el caché de WinInet, y cómo podemos utilizarlo y ampliarlo desde nuestros propios programas.

Introducción

Durante los cuatro últimos artículos hemos ido explicando detalladamente cómo utilizar el API WinInet para el acceso a recursos en Internet. Desde funciones auxiliares para gestión de *cookies*, marcado del módem, etc., hasta el manejo de los protocolos HTTP y FTP. En muchas de esas explicaciones, hemos hablado del modo "offline", o de consultas al caché, aunque muy por encima, ya que no quería perderme por las ramas. Pues ha llegado el momento de hablar sobre una de las mejores virtudes de WinInet: la integración con el caché de Internet Explorer.

Pero... ¿qué es el caché?

Como siempre, vamos ha empezar dando un pequeño repaso a los conceptos teóricos, en nuestro caso, vamos a explicar qué es el caché.

Todos sabemos que en informática algunas operaciones son más lentas que otras. Siempre se nos ha dicho que leer de un disco es más lento que leer de memoria, así como acceder a través de una red es más lento que acceder a nuestra propia máquina. Incluso, utilizar una red como Internet (WAN) es más lenta que una red local (LAN). Esto es inevitable, ciertas operaciones son costosas en tiempo y como mucho podemos arañar algunos milisegundos, pero nunca conseguiremos algo rápido.
En los inicios de la informática, los discos eran de lo más lento que se podía manejar. Pensad en las películas de los años 70-80, en las que aparecían los grandes centros de procesamiento de datos, con esas grandes cintas que se colgaban en vertical de unos armarios refrigerados. Las cintas giraban en uno y otro sentido, buscando los datos desesperadamente. En ocasiones había hasta robots que se encargaban de cambiar la cinta cuando era necesario. En definitiva: el acceso a datos en disco era demasiado lento.

Vamos a imaginarnos una pequeña historia: *"estamos en el centro de procesamiento de datos de algún afamado laboratorio, a principios de los años 70. Nos han encargado implementar un algoritmo (posiblemente en ensamblador) en el que es necesario acceder a un dato concreto que está almacenado en alguna de nuestras cintas. Después de ciertas búsquedas, sabemos que nuestro dato está catalogado en la cinta 327, armario 12, estantería 8. Tenemos que dar la orden al robot, el cual se desplaza hasta la posición indicada, coge la cinta, la sitúa en el lector (que posiblemente está a unos metros de distancia del armario) y da la señal para indicar que ya está lista. Han pasado varias decenas de segundos. Una vez que todo está listo para empezar, nuestro programa abre el fichero, busca la posición adecuada y lee el dato. Como las cintas son un dispositivo de búsqueda secuencial, la búsqueda del dato es muy lenta, y es preciso rebobinar toda la cinta hasta el lugar adecuado. Han pasado otras decenas de segundos. En definitiva, hemos tardado varios minutos para acceder a un único dato"*. Esto es lo que le podía pasar a un programador de la década de los 70,

mientras que hoy en día medimos los tiempos de acceso a disco en milisegundos (5-10 milisegundos es el tiempo más normal en leer un dato de un disco duro).

Continuemos con nuestra historia: *"Lo peor viene después de algunas instrucciones, cuando nos damos cuenta de que volvemos a necesitar el mismo dato. ¡¡Horror!! Tenemos que volver a pasar por el mismo proceso, perdiendo de nuevo algunos minutos. Después de pensarlo se nos ocurre una solución sencilla y efectiva: almacenar en memoria el dato que habíamos obtenido en nuestra primera operación de lectura, y utilizar ese mismo dato para evitar la segunda lectura. Con esto conseguimos dividir a la mitad los tiempos de acceso a ciertos dispositivos lentos."*

A grandes problemas, grandes soluciones, así que esta técnica se fue haciendo muy popular, y cada vez se utilizaba con más frecuencia. Llegó un punto en que se reservó un espacio de memoria fijo para este fin: para almacenar temporalmente ciertos datos de disco, y así evitar su lectura más de una vez. A esta memoria se le llamó "memoria de caché" y se decía que el dato estaba en el caché cuando estaba almacenado en esta memoria, y no era necesario volver a realizar algún proceso para obtenerlo de nuevo.
Y el caché nos acompaña hasta nuestros días. Hoy en día, los discos duros contienen un caché interno que almacena los últimos datos leídos, las CPU tienen una memoria caché para almacenar las últimas instrucciones ejecutadas, los servidores web tienen un caché donde se almacenan las últimas peticiones servidas, y los navegadores tienen un caché donde se almacenan las últimas páginas visitadas.
Como veis, siempre que entra en juego alguna operación lenta (o crítica), el caché puede ser nuestra solución.
Y por si fuera poco, el caché tiene otra utilidad: proveernos del dato si el dispositivo donde estaba almacenado deja de funcionar.

Pero nuestra pequeña historia continúa: *"después de algunas operaciones, y varios minutos de ejecución vemos cómo el robot comienza a comportarse de forma extraña: coge las cintas y las suelta en el aire, o se desplaza de forma aleatoria intentando agarrar algo en el aire. Quizá sean fantasmas. Algo está pasando, y precisamente ahora que necesitamos obtener otro dato de la cinta. Pero algo nos salva: hemos almacenado las últimas 10 lecturas en la memoria caché, así que es posible que el dato que necesitamos ahora ya esté en la caché. Así que lo buscamos, y ¡bingo! Nuestro dato está ahí, así que no es necesario molestar al robot, que parece que se encuentra indispuesto."*
De todas formas, no es oro todo lo que reluce, y el uso del caché introduce nuevos problemas que no teníamos hasta ahora. Volviendo a nuestro centro de procesamiento de datos... *"hemos realizado nuestra primera lectura, y como hacemos uso del caché (muy acertadamente), hemos guardado este dato en nuestra memoria caché. El resto de investigadores continúan trabajando al igual que nosotros, leyendo, y, porqué no, modificando ciertos datos de las cintas. Después de algunos minutos, necesitamos de nuevo nuestro dato, y acudimos a nuestra querida caché para evitar el largo proceso de lectura. Sin embargo, el dato no parece el correcto. Es aquel que hemos leído, pero algo no cuadra. Nadie se ha dado cuenta de que alguien ha utilizado la cinta para modificar ese mismo dato, ya que ha cambiado su cálculo, y nosotros, como lo tenemos almacenado en el caché y confiamos ciegamente en él, vamos a utilizar un dato antiguo y erróneo, así que nuestros resultados podrán ser catastróficos."*

En este caso se dice que el caché "ha perdido la coherencia", y en cierto modo, manda al traste nuestro invento. Este es uno de los principales problemas del caché, ya que, si transcurre mucho tiempo desde que almacenamos el dato en la caché, hasta que hacemos uso de él, puede ser que haya quedado desfasado. De todas formas, poco a poco se han ido creando técnicas para evitar al máximo esta pérdida de coherencia.

Dependiendo del uso que queramos dar al caché, esta situación se puede dar o no. Por ejemplo, las instrucciones de la CPU que se almacenan en el caché, nunca quedan desfasadas, ya que son datos de sólo lectura, y no pueden ser modificados. En el caso que nos ocupa, en el caché de un navegador de internet (como Internet Explorer), sí que se nos produce esta situación, ya que es posible que una página haya sido modificada después de que nosotros la almacenemos en el caché.
Sin embargo, aunque tengamos estos pequeños inconvenientes, el caché de Internet Explorer, como cualquier otro caché, tienen una doble utilidad muy importante: acelerar el acceso a ciertos recursos remotos, y asegurarnos el acceso a algunos recursos aunque no dispongamos de conexión a internet.

El caché en internet

Cuando estamos navegando por internet, la mayoría de los navegadores van almacenando en el disco duro (es decir: en el caché) los datos que vamos recibiendo. Si pulsamos el botón "Atrás", o si volvemos a visitar la misma página, muchos de estos datos se leerán del caché, mientras que otros se volverán a recuperar del servidor original.
El uso del caché en los navegadores de internet, está tan extendido que incluso el protocolo HTTP ha incluído ciertas cabeceras para facilitar su control (como "Last-modified", "Expires", etc.) así como el lenguaje de marcas HTML, que también incluye sus marcas especiales para indicar cuando una página debe almacenarse en el caché, hasta que fecha, etc.
El caché del explorador que estemos utilizando, debe tener en cuenta estos aspectos (las cabeceras HTTP o las marcas especiales en las páginas), para saber si debe almacenar en el caché cierto dato, o si, a partir de cierta fecha, debe dar por caducado algún recurso almacenado en el caché.
Por ejemplo, con la siguiente marca de HTML, se define que la página caduca en cierta fecha y hora, a partir de la cual debe considerarse que, si existe una copia en el caché, está obsoleta:

```
<meta http-equiv="Expires" content="SAT, 19 MAY 2001 22:30:00 GMT" />
```

O esta otra, que indica que la página no debe almacenarse en el caché:

```
<meta http-equiv="Pragma" content="no-cache" />
```

Aunque la mayoría de los navegadores tenga un caché interno, el de Microsoft tienen algo especial: permite navegar por las páginas que tenemos en el caché, pudiendo así simular una conexión a internet, aunque no dispongamos de ella.

Por ejemplo, supongamos que ayer hemos visitado alguna página, y hoy queremos volver a leer algo de ella. Sin embargo, nuestro módem se ha roto, así que nuestra conexión a internet no funciona. Lo único que tenemos que hacer es abrir nuestro Internet Explorer, marcar la opción "Trabajar sin conexión", del menú "Archivo", y teclear la dirección que buscábamos. Si esa página está en el caché, veremos como aparece en pantalla como si estuviéramos conectados a la red. ¡Magia! No, magia no, más bien "caché". Incluso podemos hacer clic en algunos enlaces, concretamente, los que enlazan a páginas que también están en el caché. Si alguna página no está disponible se nos mostrará una ventana indicando que ese recurso no existe, y que es necesario conectar a internet para recuperarlo.

Otro de los puntos fuertes de Internet Explorer es que nos proporciona acceso a su caché a través de funciones especiales dentro del API WinInet. Con estas funciones podemos consultar las entradas del caché, incluso añadir nuestras entradas o modificar las existentes.

Y por si fuera poco, WinInet nos permite simular la navegación tal y como hemos visto, activando la opción INTERNET_FLAG_OFFLINE. Si utilizamos esta bandera, todas las peticiones que hagamos se intentarán resolver sin conexión a la red, utilizando sólo consultas al caché local.

Además, el caché de Internet Explorer almacena otros elementos que no son los de un caché típico: estos son las entradas del historial de navegación y las *cookies* que ha ido almacenando el navegador.

Uso transparente del caché con WinInet

Como acabamos de decir, el caché de Internet Explorer puede ser utilizado de una forma transparente a través de las funciones de WinInet. Simplemente indicando que queremos trabajar sin conexión, podemos hacer cualquier tipo de llamada, pero sólo devolverán datos aquellas que encuentren la información en el caché de Internet Explorer.
Vamos a poner un ejemplo. Recordemos cómo escribir una pequeña rutina (simplificada) que recupere una página web, a través del protocolo HTTP:

```
var
   hInet, hPagina: HINTERNET;
   buff: array[0..255] of char;
   size: DWORD;
begin
    hInet := InternetOpen('Agente WinInet', INTERNET_OPEN_TYPE_PRECONFIG,
                          nil, nil, 0);

    hPagina := InternetOpenUrl(hInet,
                               'http://www.lawebdejm.com/index.html',
                               nil, 0, 0, nil);

    size = 1024;

    InternetReadFile(hPagina, buff, 1024, size);

    MessageBox(GetActiveWindow, buff, 'Contenido', MB_ICONINFORMATION);

    InternetCloseHandle(hPagina);
    InternetCloseHandle(hInet);
end;
```

Como vemos, este pequeño código recupera una página, la almacena en un buffer estático de 1024 bytes y la muestra en un mensaje.
Lógicamente, si la conexión a internet no está disponible, los datos no se podrán recuperar correctamente.
Sin embargo, podemos hacer un pequeño cambio, para acceder al caché en vez de acceder a internet. El cambio es tan sencillo como añadir la bandera INTERNET_FLAG_OFFLINE al último parámetro de InternetOpen:

```
var
   hInet, hPagina: HINTERNET;
   buff: array[0..255] of char;
   size: DWORD;
begin
   hInet := InternetOpen('Agente WinInet', INTERNET_OPEN_TYPE_PRECONFIG,
                         nil, nil, INTERNET_FLAG_OFFLINE);

   hPagina := InternetOpenUrl(hInet,
                              'http://www.lawebdejm.com/index.html',
                              nil, 0, 0, nil);

   if (hPagina = nil) and (GetLastError = ERROR_FILE_NOT_FOUND) then
      MessageBox(GetActiveWindow,
                 'El recurso no se encuentra en el caché de internet.',
                 'Error de caché', MB_ICONINFORMATION)
   else
   begin
      // lectura como en el anterior código
   end;
```

```
end;
```

Simplemente hemos añadido la opción INTERNET_FLAG_OFFLINE, y una comprobación para asegurarnos de que hemos encontrado el recurso en el caché. La mayoría de las funciones de WinInet, establecen el error ERROR_FILE_NOT_FOUND cuando intentan acceder a un recurso que no existe en el caché.
Como vemos, el uso del caché es totalmente transparente, y cambiando una simple bandera, podemos hacer que WinInet se comporte de una u otra forma.
Además, el uso de las funciones de WinInet no sólo consultan los datos del caché, sino que también almacenan los datos recuperados en él. Por ejemplo, cuando se ejecuta el primer código, y se lee el resultado hasta el final, la página quedará almacenada en el caché, a no ser que se haya incluido la bandera INTERNET_FLAG_NO_CACHE_WRITE en la llamada a InternetOpenUrl.
Para configurar este comportamiento, Las funciones de InternetConnect e InternetOpenUrl permiten ciertas banderas especiales, como por ejemplo con INTERNET_FLAG_NO_CACHE_WRITE, INTERNET_FLAG_PRAGMA_NOCACHE, INTERNET_FLAG_RESYNCHRONIZE, etc.

Uso directo del caché con WinInet

El modo más sencillo, y más eficiente, de usar el caché es de forma transparente, como acabamos de ver, ya que tenemos que hacer cambios mínimos en nuestros programas para que se beneficien del uso del caché. Sin embargo, WinInet también ofrece un conjunto de funciones para acceder directamente al caché, permitiendo su consulta y modificación.
Las operaciones más importantes que podemos hacer con el caché son:

- Enumerar las entradas almacenadas
- Consultar las características de una entrada
- Consultar el contenido de una entrada
- Añadir una entrada
- Eliminar una entrada
- Modificar las características una entrada
- Gestionar los grupos de entradas

Enumerar las entradas almacenadas en el caché

De una forma similar a lo que nos ocurría con el protocolo FTP, tenemos que hacer uso de tres funciones, una para iniciar la búsqueda y recuperar la primera entrada, otra para recuperar las siguientes entradas y otra para finalizarla.
Vamos a ver los detalles de cada una de estas funciones:

```
function FindFirstUrlCacheEntry(
                lpszPatrónBúsqueda: PChar;
                var lpInformaciónEntrada: INTERNET_CACHE_ENTRY_INFOA;
                var lpdwLongitudInformaciónEntrada: LongWord
            ): THandle;
```

- **lpszPatrónBúsqueda**: se trata de una cadena que almacena el patrón de búsqueda. Se pueden utilizar los comodines típicos, como "*" y "?", y las cadenas especiales "visited:" o "cookie:", para recuperar las entradas del historial y las *cookies* respectivamente.

- **lpInformaciónEntrada**: es un puntero a una estructura de tipo INTERNET_CACHE_ENTRY_INFO que almacena los datos de la entrada del caché. La definición de esta estructura es bastante compleja, ya que su tamaño no es fijo, sino que

depende de los datos que estén almacenados. Sin embargo, vamos a explicar los atributos más importantes:

```
INTERNET_CACHE_ENTRY_INFOA = record
    dwStructSize:      LongWord;
    lpszSourceUrlName: PChar;
    lpszLocalFileName: PChar;
    CacheEntryType:    LongWord;
    dwUseCount:        LongWord;
    dwHitRate:         LongWord;
    dwSizeLow:         LongWord;
    dwSizeHigh:        LongWord;
    LastModifiedTime:  TFileTime;
    ExpireTime:        TFileTime;
    LastAccessTime:    TFileTime;
    LastSyncTime:      TFileTime;
    lpHeaderInfo:      PByte;
    dwHeaderInfoSize:  LongWord;
    lpszFileExtension: PChar;
    dwReserved:        LongWord;
end;
```

- **dwStructSize**: el valor obtenido de sizeof(INTERNET_CACHE_ENTRY_INFO)
- **lpszSourceUrlName**: una cadena con la URL de la entrada.
- **lpszLocalFileName**: una cadena con la ruta del archivo local que almacena la entrada.
- **CacheEntryType**: se trata de una máscara de bits que indican el tipo de entrada. Actualmente todavía están por definir la mayoría de los tipos aunque hay algunos fijos:
- NORMAL_CACHE_ENTRY: entrada de caché normal.
- COOKIE_CACHE_ENTRY: entrada de caché de *cookie*.
- URLHISTORY_CACHE_ENTRY: entrada de caché de historial.
- **dwUseCount**: número de veces que la entrada del caché ha sido usada.
- **dwHitRate**: número de veces que la entrada del caché ha sido recuperada.
- **dwSizeLow** y **dwSizeHigh**: doble palabra baja y alta del tamaño del archivo.
- **LastModifiedTime**: la fecha y hora de la última modificación de la entrada.
- **LastSyncTime**: la fecha y hora en que la entrada fue sincronizada con el original por última vez.
- **lpHeaderInfo**: Un puntero a una zona de memoria que almacena las cabeceras de información. Esta zona de memoria en realidad está almacenada al final de la estructura, por eso es una estructura de tamaño variable.
- **dwHeaderInfoSize**: longitud del buffer apuntado por lpHeaderInfo.
- **lpszFileExtension**: un puntero a una cadena que almacena la extensión de la entrada. Esta zona de memoria, al igual que lpHeaderInfo, se sitúa al final de la estructura.

- **lpdwLongitudInformaciónEntrada**: el tamaño que hemos reservado para la estructura, pasado por referenia. Si se tratase de una estructura de tamaño fijo, nos valdría con indicar el valor sizeof(INTERNET_CACHE_ENTRY_INFO), sin embargo, al ser una estructura de tamaño variable, debemos pasar un valor, e ir ampliando la estructura hasta que sea suficiente. Al retornar la función, se copia en esta variable el tamaño necesitado.

La función retorna un descriptor de búsqueda que nos servirá para obtener las siguientes entradas. Si se ha producido un error, se retornará **nil**, y debemos comprobar el valor de GetLastError() para saber qué ha ocurrido. Una de las posibles causas de error es que el tamaño reservado para la estructura sea insuficiente. En este caso, la llamada a GetLastError() retornará ERROR_INSUFFICIENT_BUFFER, y en el parámetro "lpdwLongitudInformaciónEntrada" se habrá copiado el tamaño requerido.
En este punto hay que tener en cuenta todo lo que hablamos sobre la fragmentación y la memoria dinámica, en el artículo sobre Los montones (www.lawebdejm.com/?id=22130). Si recordamos, dijimos que como norma general, siempre que hubiese que ampliar un buffer, había que duplicar su

tamaño. Este es uno de los casos, ya que hay que ir ampliando el espacio reservado para la estructura, hasta que la función FindFirstUrlCacheEntry retorne un valor correcto.
En caso de error, GetLastError() puede retornar ERROR_NO_MORE_ITEMS, lo que significa que no hay ninguna entrada que cumpla los criterios de búsqueda.

En el siguiente código podemos ver cómo hacer una llamada típica, redimensionando el buffer hasta su tamaño correcto:

```
var
   info: LPINTERNET_CACHE_ENTRY_INFO;
   size, sizeReq: DWORD;
   busqueda: THandle;
begin
   busqueda = nil;

   // se establece un tamaño inicial arbitrario (potencia de dos)
   size := 128; // ~= sizeof(INTERNET_CACHE_ENTRY_INFO)
   sizeReq := size;
   GetMem(info, size);
   info^.dwStructSize := sizeof(INTERNET_CACHE_ENTRY_INFO);

   // este bucle duplica el tamaño del buffer
   while busqueda = nil do
   begin
      busqueda := FindFirstUrlCacheEntry(nil, info, sizeReq);
      if busqueda = nil then
      begin
         FreeMem(info, size); // se borra el buffer

         case GetLastError() of
         begin
            // era demasido pequeño: lo duplicamos
            ERROR_INSUFFICIENT_BUFFER:
               begin
                  size := size * 2;
                  sizeReq := size;
                  GetMem(info, size);
                  info^.dwStructSize := sizeof(INTERNET_CACHE_ENTRY_INFO);
               end;

            // no se ha encontrado ningún elemento
            ERROR_NO_MORE_ITEMS:
               begin
                  result := 0;
                  exit;
               end;

            // otro error
            else:
               begin
                  result := -1;
                  exit;
               end;
         end;
      end;
   end;

    // aquí ya tenemos en "info" los datos de la primera entrada y
    // podemos ir recuperando el resto de resultados.

    [...]
end;
```

Existe una función extendida llamada FindFirstUrlCacheEntryEx, con la que podemos localizar entradas de un tipo determinado, o pertenecientes a un grupo concreto. No voy a entrar en esta función ya que con la básica nos es suficiente por ahora.
El siguiente paso dentro de la enumeración de entradas es ir recuperando cada uno de los resultados. Esto lo podemos conseguir a través de la función FindNextUrlCacheEntry, que tiene la siguiente sintaxis:

```
function FindNextUrlCacheEntry(
                hBusqueda: THandle;
                var lpInformaciónEntrada: INTERNET_CACHE_ENTRY_INFOA;
                var lpdwLongitudInformaciónEntrada: LongWord
            ): LongBool;
```

- **hBusqueda**: un descriptor obtenido a través de una llamada a FindNextUrlCacheEntry.
- **lpInformaciónEntrada**: un puntero a una estructura de tamaño variable de tipo INTERNET_CACHE_ENTRY_INFO.
- **lpdwLongitudInformaciónEntrada**: un puntero a un valor de 32 bits que contiene el tamaño reservado para la estructura pasada en "lpInformaciónEntrada". Al retornar, la función copia en este parámetro el número de bytes copiados a la estructura.

La función retorna TRUE o FALSE, dependiendo el éxito o fracaso. En esta función hay que tener las mismas precauciones que con FindFirstUrlCacheEntry, a la hora de reservar tamaño para la estructura de tamaño variable. Si la función retorna FALSE, y la llamada a GetLastError() retorna ERROR_INSUFFICIENT_BUFFER, debemos ampliar el tamaño reservado para la estructura INTERNET_CACHE_ENTRY_INFO.
Cuando la función retorne FALSE y la llamada a GetLastError() sea ERROR_NO_MORE_ITEMS, se puede dar por terminada la búsqueda, ya que no hay más resultados.
Y por último, debemos cerrar el descriptor de la búsqueda, utilizando la función FindCloseUrlCache:

```
function FindCloseUrlCache(
                hBusqueda: THandle
            ): LongBool;
```

Símplemente debemos pasar el descriptor de la búsqueda obtenido con FindFirstUrlCacheEntry ó FindFirstUrlCacheEntryEx, la función retornará TRUE si el descriptor se ha cerrado correctamente.

Consultar los datos de una entrada del caché

Si no queremos enumerar todas las entradas del caché, podemos obtener los datos de una única entrada, localizándola a partir de la URL que representa.
Por ejemplo, si queremos saber si el caché almacena la página "http://www.lawebdejm.com/index.html", podemos recuperar sus datos (fichero local, accesos, fecha de caducidad, etc.), y comprobar el valor retornado por la función.

```
function GetUrlCacheEntryInfo(
                lpszUrl: PChar;
                var lpInformaciónEntrada: INTERNET_CACHE_ENTRY_INFOA;
                var lpdwLongitudInformaciónEntrada: LongWord
            ): LongBool;
```

- **lpszUrl**: la URL de la entrada del caché cuya información queremos obtener. Hay que tener cuidado de que esta URL sea directamente una página o recurso, sin parámetros ni anclas. Es decir: si utilizamos la URL www.servidor.com/pagina.php?parametro=valor, no se encontrará nada, porque en el caché se almacenan los recursos sin parámetros. Lo mismo ocurrirá si utilizamos la URL www.servidor.com/pagina.htm#ancla.
- **lpInformaciónEntrada**: un puntero a una estructura de tamaño variable de tipo INTERNET_CACHE_ENTRY_INFO.

- **lpdwLongitudInformaciónEntrada**: un puntero a un valor de 32 bits que contiene el tamaño reservado para la estructura pasada en "lpInformaciónEntrada". Al retornar, la función copia en este parámetro el número de bytes copiados a la estructura.

La función, como ya sabemos, retorna TRUE o FALSE. En caso de error, hay que llamar a GetLastError para averiguar la causa, pudiendo ser ERROR_INSUFFICIENT_BUFFER, si el tamaño para "lpInformaciónEntrada" es insuficiente, o ERROR_FILE_NOT_FOUND si no hay ninguna entrada en el caché para la URL indicada.

Consultar el contenido de una entrada del caché

Una vez que conocemos los datos de una entrada del caché, podemos acceder al fichero donde se almacena dicha entrada, a través del atributo "lpszLocalFileName" de la estructura "INTERNET_CACHE_ENTRY_INFO". Con esta ruta de fichero, podemos manipularlo como queramos, abriéndolo, leyéndolo, etc.
Sin embargo, WinInet no recomienda esta técnica, ya que se podría perder la consistencia del caché. Para ello nos proporcionan una serie de funciones que hacen el trabajo por nosotros, para leer o modificar el contenido de una entrada del caché.
Para la lectura debemos hacer uso de la función "RetrieveUrlCacheEntryStream".

```
function RetrieveUrlCacheEntryStream(
                lpszUrl:           PChar;
                var lpInformaciónEntrada: INTERNET_CACHE_ENTRY_INFOA;
                var lpdwLongitudInformaciónEntrada: LongWord;
                fLecturaAleatoria: LongBool;
                dwReservado:       LongWord
            ): THandle;
```

- **lpszUrl**: la URL de la entrada que queremos leer.
- **lpInformaciónEntrada**: un puntero a una estructura de tamaño variable de tipo INTERNET_CACHE_ENTRY_INFO.
- **lpdwLongitudInformaciónEntrada**: un puntero a un valor de 32 bits que contiene el tamaño reservado para la estructura pasada en "lpInformaciónEntrada". Al retornar, la función copia en este parámetro el número de bytes copiados a la estructura.
- **fLecturaAleatoria**: indica si se va a leer de forma aleatoria. En caso de indicar FALSE, la lectura debe hacerse secuencialmente.
- **dwReservado**: debe ser 0.

La función retorna un descriptor del archivo abierto, que podemos utilizar en la función ReadUrlCacheEntryStream. En caso de error, se retornará **nil**, y debemos llamar a GetLastError(), que podrá informarnos de los errores que ya conocemos: ERROR_FILE_NOT_FOUND y ERROR_INSUFFICIENT_BUFFER.
Una vez que el archivo está abierto, podemos leerlo con ReadUrlCacheEntryStream:

```
function ReadUrlCacheEntryStream(
                hDescriptorArchivo: THandle;
                dwPosición:   LongWord;
                var lpBuffer: Pointer;
                var lpdwLongitudBuffer: LongWord;
                dwReservado:  LongWord
            ): LongBool;
```

- hDescriptorArchivo: un descriptor obtenido con una llamada a RetrieveUrlCacheEntryStream.
- dwPosición: indica la posición en la que queremos leer, pasando 0 si queremos leer desde el principio.

- lpBuffer: un puntero a una zona de memoria donde se almacenarán los datos leídos. Para archivos cuyo contenido sea texto plano, podemos pasar un puntero a cadena de caracteres.
- lpdwLongitudBuffer: un puntero a un valor de 32 bits que contiene la longitud de memoria de "lpBuffer". Al retornar, este valor contiene el número de bytes leído, o los bytes necesarios, si no se han podido copiar todos.
- dwReservado: debe ser 0.

La función retorna TRUE o FALSE. En caso de error, GetLastError() nos informará del tipo de error, pudiendo ser ERROR_INSUFFICIENT_BUFFER si el tamaño pasado en "lpdwLongitudBuffer" es demasiado pequeño.

Una vez que hemos leído el contenido, debemos cerrar el descriptor del archivo a través de la siguiente función:

```
function UnlockUrlCacheEntryStream(
            hDescriptorArchivo: THandle,
            dwReservado: LongWord
        ): LongBool;
```

Tan solo debemos pasar el descriptor obtenido con RetrieveUrlCacheEntryStream, y un cero en el segundo parámetro.

Añadir una entrada al caché

Desde nuestros programas también podemos insertar nuevas entradas en el caché, para que el propio Internet Explorer, o cualquier otro programa, pueda utilizarlas.
Esto se hace en tres pasos: crear la entrada, escribir su contenido y guardar y validar todo.
La función para el primer paso es la siguiente:

```
function CreateUrlCacheEntry(
                lpszUrl:          PChar;
                dwTamañoEsperado: LongWord;
                lpszExtensión:    PChar;
                lpszArchivoLocal: PChar;
                dwReservado:      LongWord
            ): LongBool;
```

- **lpszUrl**: la URL que identificará a la entrada del caché. Posteriormente, podemos localizar la entrada indicando esta URL. No debes incluir parámetros adicionales después del nombre del recurso.
- **dwTamañoEsperado**: el tamaño que tienen el recurso que vamos a almacenar, o un cero si no sabemos realmente su tamaño.
- **lpszExtensión**: una cadena que contiene la extensión del archivo a almacenar.
- **lpszArchivoLocal**: una puntero a una cadena de caracteres donde se almacenará el copiará la ruta y nombre del archivo local donde se va a almacenar el recurso. Este buffer debe ser lo suficientemente grande, de al menos MAX_PATH (255) caracteres.
- **dwReservado**: debe ser 0.

La función retorna TRUE o FALSE, dependiendo de su éxito o fracaso.
Una vez que hemos creado la entrada para el recurso, podemos acceder al fichero local con cualquier de los métodos que tenemos disponibles. Se trata de un archivo normal, así que podemos abrirlo con CreateFile, y escribir en él con WriteFile, y cerrarlo con CloseHandle. También podemos hacer uso de los mecanismos del lenguaje, como ReadLn, WriteLn, etc, o las clases de la VCL, como TFileStream.
Y por último, cuando ya tenemos almacenados los datos del recurso, debemos validar todo, y establecer las propiedades de la entrada en el caché, a través de la siguiente función:

```
function CommitUrlCacheEntry(
               lpszUrl:               PChar;
               lpszArchivoLocal:      PChar;
               FechaCaducidad:        TFileTime;
               FechaModificacion:     TFileTime;
               dwTipo:                LongWord;
               lpInformación:         PByte;
               dwLongitudInformación: LongWord;
               lpszReservado:         PChar;
               lpszUrlOriginal:       LongWord
            ): LongBool;
```

- **lpszUrl**: la URL que hemos indicado en la llamada a CreateUrlCacheEntry.
- **lpszArchivoLocal**: el nombre de archivo que hemos obtenido en la llamada a CrearUrlCacheEntry.
- **FechaCaducidad**: la fecha a partir de la cual se considera que el recurso ya no es válido. Si no queremos que el recurso caduque, podemos pasar un 0.
- **FechaModificacion**: la fecha en que la entrada ha sido introducida en el caché.
- **dwTipo**: Una máscara de bits indicando el tipo de recurso. Puede incluirse cualquier valor de los que hemos indicado en el atributo CacheEntryType de la estructura INTERNET_CACHE_ENTRY_INFO.
- **lpInformación**: un puntero a una cadena donde se puede almacenar información extra, que será retornada en el atributo lpHeaderInfo de la estructura INTERNET_CACHE_ENTRY_INFO. Si no queremos información extra, podemos pasar el valor **nil**.
- **dwLongitudInformacion**: la longitud de la cadena "lpInformación".
- **lpszReservado**: Debe ser **nil**
- **lpszUrlOriginal**: Si el recurso es producto de una redirección, debemos pasar aquí la URL original. En caso contrario, podemos pasar el valor **nil**.

La función retornará o valor booleano indicando su éxito. En caso de error podemos llamar a GetLastError(), que nos retornará ERROR_DISK_FULL, si se ha alcanzado el límite de espacio reservado para el caché, o ERROR_FILE_NOT_FOUND, si el parámetro lpszArchivoLocal contiene un nombre de archivo que no ha sido creado con CreateUrlCacheEntry.

Eliminar una entrada al caché

Otra de las operaciones que podemos realizar con el caché, es eliminar una entrada concreta.
Para ello tenemos que llamar a la siguiente función:

```
function DeleteUrlCacheEntry(
              lpszUrl: PChar
           ): LongBool;
```

Esta vez es fácil ¿no? Simplemente se pasa la URL de la entrada que queremos eliminar, y la función nos retorna un valor indicando si ha sido borrada o no.
En caso de no poder borrar la entrada, podemos llamar a GetLastError() para averiguar la razón, pudiendo ser:

- ERROR_ACCESS_DENIED, si la entrada está bloqueada, porque ha sido abierta con ReadUrlCacheEntryStream o ReadUrlCacheEntryFile. Es este caso, la entrada será marcada para borrarse cuando sea desbloqueada.
- ERROR_FILE_NOT_FOUND: no hay ninguna entrada para la URL indicada.

Modificar los datos de una entrada del caché
Y por último, podemos modificar los datos informativos de una entrada del caché, a través de la siguiente función:

```
funtion SetUrlCacheEntryInfo(
                lpszUrl: PChar;
                var lpInformación: INTERNET_CACHE_ENTRY_INFOA;
                dwCamposModificar: LongWord
            ): LongBool;
```

- **lpszUrl**: una cadena que indica la URL a modificar
- **lpInformación**: un puntero a una estructura que contiene los nuevos datos.
- **dwCamposModificar**: es una máscara de bytes que indica aquellos campos de la estructura "lpInformacion" que se van a establecer. Se puede incluir cualquier combinación de los siguientes valores:
  - CACHE_ENTRY_ACCTIME_FC: establece el valor del campo "LastAccessTime"
  - CACHE_ENTRY_ATTRIBUTE_FC: establece el valor del campo "CacheEntryType"
  - CACHE_ENTRY_EXPTIME_FC: establece el valor del campo "ExpireTime"
  - CACHE_ENTRY_HEADERINFO_FC: establece el valor del campo "lpHeaderInfo"
  - CACHE_ENTRY_HITRATE_FC: establece el valor del campo "dwHitRate"
  - CACHE_ENTRY_MODTIME_FC: establece el valor del campo "LastModifiedTime"
  - CACHE_ENTRY_SYNCTIME_FC: establece el valor del campo "LastSyncTime"

La función, como todas, retorna TRUE o FALSE, y en caso de error, debemos llamar a GetLastError() para averiguar la causa del fallo, pudiendo ser:

- ERROR_INVALID_PARAMETER, alguno de los valores a establecer no es correcto.
- ERROR_FILE_NOT_FOUND: no hay ninguna entrada para la URL indicada.

Gestionar grupos de entradas

Hasta ahora hemos hablado del caché como si fuera una lista lineal de entradas, una detrás de otra. En realidad esto es así, aunque existe un método de jerarquizar la información. A partir de la versión 4 del Internet Explorer, el caché de WinInet permite la creación de grupos, identificados por un número, que permiten agrupar las entradas atendiendo a distintos criterios. Por ejemplo, podemos crear un grupo para todas aquellas entradas que provengan del dominio "www.lawebdejm.com", y posteriormente podemos trabajar con este grupo, enumerando sus entradas, eliminándolas, añadiendo nuevas, etc.
No voy a detallar el uso de las funciones porque es muy parecido a las que ya hemos explicado, simplemente os doy el nombre de las que hay que utilizar en cada caso, y vosotros podéis consultar su sintaxis en el MSDN de Microsoft:

- Crear un nuevo grupo: CreateUrlCacheGroup
- Añadir entrada a un grupo: SetUrlCacheEntryGroup con la bandera INTERNET_CACHE_GROUP_ADD
- Eliminar entrada de un grupo: SetUrlCacheEntryGroup con la bandera INTERNET_CACHE_GROUP_REMOVE
- Enumerar las entradas de un grupo: FindFirstUrlCacheEntryEx y FindNextUrlCacheEntryEx.

Mi asignatura pendiente: mensajes de error

Ya hace unos cuantos meses que estoy escribiendo sobre WinInet, y la verdad es que he aprendido bastante desde aquel primer artículo. Una de las cosas que no expliqué en el artículo introductorio era cómo obtener el mensaje de error a partir de su código. Es decir: siempre hemos dicho la mayoría de las funciones retornan FALSE cuando ocurre un error, y para obtener el código de error hay que llamar a GetLastError. Cada código de error tiene un mensaje descriptivo asociado, aunque hasta ahora no sabía cómo hacer para obtenerlo. No se puede saber todo desde el principio ¿no? Mis averiguaciones se habían atascado en el punto en que sabía que los mensajes estaban almacenados como recursos de cadena dentro del archivo "wininet.dll". Lo que no sabía era cómo localizar el recurso adecuado a partir de un código de error concreto. Después de pasar muchas horas consultando el MSDN, pude encontrar la clave del asunto. La cuestión está en una función que no pertenece al API WinInet, sino al API genérico de Win32: FormatMessage. Esta función trabaja de distintas formas, obteniendo el mensaje (teniendo en cuenta el idioma instalado) de un error genérico de Win32, formateando una cadena con máscaras al estilo "printf", u obteniendo un mensaje a partir de una librería independiente. Ahí está el asunto. Podemos obtener los mensajes que están almacenados como recursos dentro de la librería "wininet.dll", localizándolos a partir del código de error.
Lo primero que necesitamos es un descriptor de la librería, bien abriéndola con LoadLibrary, o intentando obtener un descriptor ya creado por el proceso, a través de la función GetModuleHandle. También podemos hacer una solución mixta, intentando obtener un descriptor ya creado, y si no existe, crearlo nosotros.
El siguiente paso es llamar a la función FormatMessage. La sintaxis es la siguiente, aunque sólo voy a explicarla superficialmente porque esta función tiene muchas formas de trabajar:

```
function FormatMessage(
                dwOpciones:       LongWord;
                lpOrigen:         Pointer;
                dwCódigoError:    LongWord;
                dwIdioma:         LongWord;
                lpBuffer:         PChar;
                dwLongitudBuffer: LongWord;
                argumentos:       Pointer
            ): LongWord;
```

- **dwOpciones**: es una máscara de bits que indican cómo debe comportarse la función. Se puede utilizar una combinación de los siguientes valores:
  - FORMAT_MESSAGE_ALLOCATE_BUFFER: la función reservará memoria en el montón por defecto del proceso (¿alguien se acuerda del artículo sobre “Los montones” (www.lawebdejm.com/?id=22130)?) para almacenar el mensaje. El que llame a FormatMessage con este parámetro es el responsable de liberar el buffer resultante con la función HeapFree(GetProcessHeap(), buffer)
  - FORMAT_MESSAGE_FROM_STRING: hace que la función actúe como un "fprintf", es decir, formateando una cadena que contiene máscaras.
  - FORMAT_MESSAGE_FROM_HMODULE: el mensaje de obtiene de una librería independiente, buscando en los recursos de cadena. Este es el valor que nos interesa para buscar los mensajes en "wininet.dll"
  - FORMAT_MESSAGE_FROM_SYSTEM: se busca en los mensajes de sistema el código indicado. Este parámetro nos sirve para conseguir la mayoría de los mensajes de funciones básicas de Win32, como CreateFile, CloseHandle, etc.

- **lpOrigen**: se trata de un puntero genérico en el que podemos pasar distintos valores. El tipo de dato que debemos pasar depende de las constantes que hayamos pasado en el parámetro dwOpciones, pudiendo ser:
  - FORMAT_MESSAGE_FROM_STRING: un puntero a una cadena que será formateada.
  - FORMAT_MESSAGE_FROM_HMODULE: un descriptor de librería, obtenido con LoadLibrary o GetModuleHandle.

  - cualquier otro: este parámetro debe ser **nil**.

- **dwCódigoError**: el código numérico del error. Normalmente lo obtenemos con GetLastError.

- **dwIdioma**: Se trata de un valor numérico que identifica el idioma en que queremos obtener el mensaje. Este valor lo podemos obtener con las funciones GetSystemDefaultLangID ó GetUserDefaultLangID, las constantes LANG_SYSTEM_DEFAULT o LANG_USER_DEFAULT. Podemos pasar el valor 0 para obtener el mensaje en el idioma por defecto del usuario o sistema.

- **lpBuffer**: en este parámetro se pasa un puntero a una cadena de caracteres en la que se copiará el mensaje obtenido. Si hemos pasado el valor FORMAT_MESSAGE_ALLOCATE_BUFFER en el parámetro dwOpciones, en realidad lo que se debe pasar es la dirección de un puntero a cadena. En esa dirección se copiará a su vez la posición de memoria donde se ha creado el nuevo buffer.

- **dwLongitudBuffer**: este parámetro puede contener el tamaño del buffer pasado en "lpBuffer", o el tamaño mínimo a reservar por la función, esto último sólo si hemos pasado FORMAT_MESSAGE_ALLOCATE_BUFFER en el parámetro dwOpciones.

- **argumentos**: se trata de una lista de argumentos variables que se utilizarán par sustituir las máscaras de la cadena, si hemos utilizado la opción FORMAT_MESSAGE_FROM_STRING. En caso de no necesitar este parámetro, podemos pasar el valor **nil**.

La función retorna el número de caracteres que se ha copiado en el buffer de salida, o 0 en caso de error. Se puede llamar a GetLastError() para averiguar la causa del error.
De todas formas, en el mundo de la programación, un ejemplo vale más que mil palabras, así que aquí tenéis un algoritmo típico para obtener el mensaje de un código de error que retorne GetLastError:

```
var
    buff:    PChar;
    hLib:    HMODULE;
    err:     LongWord;
    liberar: boolean;
begin
   err = GetLastError;
   liberar = false;

   // se obtiene el descriptor de la librería
   hLib := GetModuleHandle('wininet.dll');
   if hLib = nil then
   begin
      hLib := LoadLibrary('wininet.dll');
      liberar := true;
      if hLib = nil then
         exit;
   end;

   FormatMessage(
           FORMAT_MESSAGE_FROM_HMODULE
             or FORMAT_MESSAGE_ALLOCATE_BUFFER, // opciones
           hLib,                 // librería
           err,                  // código de error
           LANG_SYSTEM_DEFAULT, // idioma
           buff, 0,              // buffer y longitud
           nil                   // sin parámetros
        );

   MessageBox(GetActiveWindow, buff, 'Error', MB_ICONERROR);
```

```
   // se libera el buffer que ha reservado FormatMessage
   HeapFree(GetProcessHeap, 0, buff);

   // se libera la librería
   if liberar then
      FreeLibrary(hLib);
end;
```

Conclusión

Bueno, y creo que esto es todo, amigos. Con este último artículo sobre WinInet hemos tratado todos los temas importantes: desde funciones auxiliares, hasta la gestión del caché de WinInet, pasando por dos de los protocolos más importantes de internet: HTTP y FTP.
Espero que os haya resultado útil y entretenido. A mi, al menos, me ha gustado compartir con vosotros estos pequeños apuntes. Nos veremos en la próxima serie de "Los rincones del API Win32".

Los ejemplos

Estoy preparando un pequeño explorador del caché de WinInet, en el que se puede ver cómo utilizar todas estas técnicas que hemos visto. Espero tenerlo listo en unas semanas.

Autor: JM - http://www.lawebdejm.com