

Verzeichnisstrukturen ohne Zeitdruck einlesen

Ordner ohne Ende

So schnell moderne Controller und Festplatten auch sind, das Einlesen von Verzeichnisstrukturen kann sehr lange dauern. dotnetpro stellt eine einfache und effiziente Lösung vor, um dies im Hintergrund zu erledigen, ohne die Geduld des Anwenders zu strapazieren.

Damit ein Programm auch dann noch sauber läuft, wenn gerade Hunderte von Unterverzeichnissen eingelesen werden, sind schon ein paar Gedanken notwendig. Der Windows Explorer beispielsweise liest Unterverzeichnisse oft gar nicht ein und zeigt vor jedem Verzeichnis ein Plus-Symbol an. Lässt sich der betreffende Zweig nicht expandieren, verschwindet es einfach.

Der hier vorgestellte Weg benutzt einen Threadpool, um Verzeichnisse im Hintergrund einzulesen. Hierbei werden die Plus-Symbole nachgetragen, wenn die Pool-Threads die Daten gelesen haben. Der Test der Lösung erfolgte mit bis zu 1500 Unterverzeichnissen und bis zu 10 000 Dateien, die auch im Netzwerk lagen.

Im Beispielprogramm wurde die Funktionalität über Vererbung in den *TreeView* eingebaut. Dies hat den Vorteil, dass sich die Klasse separat kompilieren lässt. Die

Auf einen Blick

Autor

Michael Hartmann ist Diplom-Kaufmann und arbeitet seit 1991 als selbstständiger Softwareentwickler in München. Schwerpunkte sind die Win32-API-Programmierung, SQL-Datenbanken und seit zwei Jahren .NET. Sie erreichen ihn unter m.hartmann@site99.de.



dotnetpro.code
A0512Asynchron



Sprachen C#

Technik .NET 1.1 .NET 2.0

Voraussetzungen Visual Studio .NET 2003 oder 2005, .NET SDK 1.1 oder 2.0

Listing 1

Ein einzelnes Verzeichnis einlesen.

```
private void ReadDir_ThreadFunction(object o) {
    TreeInfoArgs e = o as TreeInfoArgs;
    e.DirList = Directory.GetDirectories(e.Dir);
    base.BeginInvoke(new ReadCompleted(e.Callback), new object[] { e });
}

private void ReadDir_Callback(TreeInfoArgs e) {
    if (e.Node.TreeView != null) {
        int cnt = e.DirList.Length;
        TreeNode[] cache = new TreeNode[cnt];
        for (int i = 0; i < cnt; i++) {
            cache[i] = new TreeNode(Path.GetFileName(e.DirList[i]));
        }
        e.Node.Nodes.AddRange(cache);
    }
}

private void ReadDir(TreeNode Node) {
    TreeInfoArgs Args = new TreeInfoArgs(Node.FullPath, Node);
    Args.Callback = new ReadCompleted(ReadDir_Callback);
    ThreadPool.QueueUserWorkItem(new WaitCallback(ReadDir_ThreadFunction), Args);
}
```

Heft-CD enthält außer den Projektdateien auch eine umfangreiche Lösung, wie der Kasten *Einsatz in der Praxis* erläutert.

Einlesen im Hintergrund

Zunächst ist über eine Thread-Funktion ein Verzeichnis im Hintergrund einzulesen, wie dies in Listing 1 geschieht. Dies ist denkbar einfach. Zuerst wird der Parameter vom Typ *object* konvertiert, dann das Verzeichnis gelesen und in einem String-Array gespeichert. Zuletzt wird über die Message-Queue die Callback-Funktion aufgerufen.

Dieser Aufruf ist notwendig, weil der Threadpool-Thread nicht direkt auf das *TreeView*-Steuerelement zugreifen darf und mithin die Verzeichnisse nicht selbst in den Baum einfügen kann. Der Aufruf kann durch *BeginInvoke()* erfolgen, denn

der Pool-Thread hat seine Arbeit bereits vollständig erledigt. Es gibt dann keinen Grund, mit *Invoke()* zu warten, bis der Haupt-Thread den Baum aktualisiert hat.

Wichtig ist die Festlegung der Argumente in der Klasse *TreeInfoArgs* in Listing 2. Werden alle relevanten Daten auf dem Stack übergeben, erübrigt sich eine explizite Synchronisierung der beteiligten Threads. Die Rückgabe des Ergebnisses in einem String-Array vermeidet unnötige Aufrufe von *BeginInvoke()* und damit von Kontextwechseln. Zudem kann die Callback-Funktion die Prozedur *AddRange()* verwenden, was deutlich schneller ist als das Einfügen von einzelnen Einträgen.

Der Parameter *Node* wird hier nicht verwendet, sondern nur an die Rückrufmethode weitergegeben.

Die Callback-Funktion aus Listing 1 prüft zunächst, ob der jeweilige Knoten

Einsatz in der Praxis

Die vorgestellte Methode wurde für das File Manager Kit entwickelt. Sein vollständiger Quellcode befindet sich auf der Heft-CD und enthält unter anderem einen kompletten Dateimanager und ein Steuerelement zum Darstellen von Verzeichnisstrukturen in einem kombinierter Tree-/ListView. Die Implementierung des Verfahrens ist etwas umfangreicher als im Beispielprogramm und deshalb in zwei Klassen aufgeteilt. Die Thread-Funktion ist in der Klasse *RelicIO*, der Rest in der Klasse *RelicTree* zu finden.

noch im Baum enthalten ist. Falls nicht, hat der Haupt-Thread in der Zwischenzeit einen neuen Baum eingelesen und die Daten werden nicht mehr benötigt. Der Knoten selbst muss nicht auf *null* geprüft werden.

Die Garbage-Collection entfernt den Knoten nicht aus dem Speicher, solange er noch über den Parameter referenziert wird. Die Thread-Synchronisierung erfolgt implizit über den Einsatz von *BeginInvoke()* und die vollständige Parametrisierung.

Die Einträge für die neuen Verzeichnisse werden zunächst in einem Array ge-

Listing 2

Die Parameter-Klasse TreeInfoArgs.

```
public delegate void ReadCompleted(TreeInfoArgs e);

public class TreeInfoArgs {
    public string Dir;
    public string[] DirList;

    public ReadCompleted Callback;
    public TreeNode Node;

    public TreeInfoArgs(string pDir, TreeNode pNode) {
        Dir = pDir;
        Node = pNode;
    }
}
```

Listing 3

Einen Baum einlesen.

```
private void ReadTree() {
    base.Nodes.Clear();
    RootNode = base.Nodes.Add(mDir.Trim(Path.DirectorySeparatorChar));
    TreeInfoArgs Args = new TreeInfoArgs(mDir, RootNode);
    Args.Callback = new ReadCompleted(ReadTree_Callback);
    ThreadPool.QueueUserWorkItem(new WaitCallback(ReadDir_ThreadFunction), Args);
}

private void ReadTree_Callback(TreeInfoArgs e) {
    if (e.Node.TreeView != null) {
        ReadDir_Callback(e);
        RootNode.Expand();
    }
}
```

speichert und dann mit *AddRange()* in den Baum eingefügt.

Um ein einzelnes Verzeichnis einzulesen, ist lediglich ein Objekt vom Typ *TreeInfoArgs* anzulegen und zu initialisieren. Danach wird der Vorgang mit *QueueUserWorkItem* gestartet, siehe Listing 1.

Mehrere Verzeichnisse parallel einlesen

Um ein Verzeichnis im Baum zu expandieren, genügt es, die folgende Schleife in den Event-Handler *OnBeforeExpand* einzutragen:

```
foreach (TreeNode n in e.Node.Nodes) {
    ReadDir(n);
}
```

Auf diese Weise wird für jedes Unterverzeichnis das Einlesen im Hintergrund gestartet.

Die Threadpool-Verwaltung startet jedoch keineswegs für jedes einzelne Verzeichnis einen eigenen Thread. Wie viele Threads gestartet werden, hängt stark von der Systemumgebung und der Konfiguration ab. Durch Ausgabe von Verzeichnis und Thread-ID mit *Debug.WriteLine* lässt sich leicht kontrollieren, welche Threads aktiv sind.

Bei Tests des Verfahrens mit 500 Unterverzeichnissen auf drei verschiedenen Rechnern wurden zwei bis neun Threads verwendet. Die Anzahl der Threads kann sogar auf demselben System bei wiederholtem Zugriff auf die gleichen Daten variieren.

Die Thread-Funktion *ReadDir_ThreadFunction()* wird in Listing 3 auch benutzt,

um zunächst einmal den Baum einzulesen. Die aufrufende Funktion *ReadTree()* löscht den alten Baum und legt den ersten Knoten an. Die Callback-Funktion trägt wie gehabt die Verzeichnisse im Baum ein. Zusätzlich ist der oberste Knoten zu expandieren, damit über den Event-Handler *OnBeforeExpand* das Einlesen der nächsten Verzeichnisebene gestartet wird.

Fehlerprüfungen

Für den praktischen Einsatz sind Fehlerprüfungen notwendig. *UnauthorizedAccess*-Ausnahmen sollten Sie auf jeden Fall abfangen. *DirectoryNotFound*- und *IO*-Ausnahmen zu behandeln, ist ebenfalls empfehlenswert.

Die Fehler müssen in der Thread-Funktion abgefangen werden. Um im Baum beispielsweise ein entsprechendes Symbol anzuzeigen, ist am besten eine zweite Callback-Funktion zu verwenden.

Sind mehrere Formulare im Spiel, sollten Sie vor *BeginInvoke()* mit *IsHandleCreated* prüfen, ob das Fenster mit dem *TreeView*-Steuerelement noch geöffnet ist. Alternativ wäre die *InvalidOperationException*-Ausnahme abzufangen.

Fazit

Verzeichnisse über Threadpool-Threads einzulesen ist effizient und der Programmieraufwand vergleichsweise gering. Die Methode lässt sich auch in anderen Fällen anwenden, beispielsweise wenn Sie Verzeichnisstrukturen durchsuchen und die Ergebnisse in einem *ListView* darstellen wollen. |||||