词组“循环访问目录树”的意思是在指定的根文件夹下，访问每个嵌套子目录中任意深度的所有文件。您不必打开每一个文件。可以只检索 **string** 形式的文件名或子目录名，或者可以检索 [System.IO.FileInfo](https://msdn.microsoft.com/zh-cn/library/system.io.fileinfo.aspx) 或 [System.IO.DirectoryInfo](https://msdn.microsoft.com/zh-cn/library/system.io.directoryinfo.aspx) 对象形式的其他信息。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 在 Windows 中，可交换使用术语“目录”和“文件夹”。大多数文档和用户界面文本使用术语“文件夹”，但 .NET Framework 类库使用术语“目录”。 |

最简单的例子，如果您确信您拥有指定根目录下所有目录的访问权限，则您可以使用 System.IO.SearchOption.AllDirectories 标志。该标志返回与指定模式相匹配的所有嵌套子目录。下面的示例演示如何使用该标志。

root.GetDirectories("\*.\*", System.IO.SearchOption.AllDirectories);

此方法的缺点是，如果指定根目录下任何一个子目录引发了 [DirectoryNotFoundException](https://msdn.microsoft.com/zh-cn/library/system.io.directorynotfoundexception.aspx) 或 [UnauthorizedAccessException](https://msdn.microsoft.com/zh-cn/library/system.unauthorizedaccessexception.aspx)，则整个方法将会失败并且不返回任何目录。使用 [GetFiles](https://msdn.microsoft.com/zh-cn/library/4cyf24ss.aspx) 方法时也是如此。如果一定要处理特定子文件夹中的这些异常，则必须手动遍历该目录树，如下面的示例所示。

手动遍历目录树时，可以先处理子目录（前序遍历），或者可以先处理文件（后序遍历）。如果执行前序遍历，则在循环访问直接位于当前文件夹本身中的文件之前，请先遍历当前文件夹下的整个树。本文档后面的示例执行的是后序遍历，但您可以轻松地将它们修改为执行前序遍历。

另外还可以选择是使用递归遍历，还是使用基于堆栈的遍历。本文档后面的示例对这两种方法进行了演示。

如果必须对文件和文件夹上执行各种操作，则可以通过下面的方法将这些示例模块化：将操作重构到单独的函数，这样您就可以通过单个委托来调用这些函数。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| NTFS 文件系统可以包含交接点、符号链接和硬链接形式的重新分析点。.NET Framework 方法（如 [GetFiles](https://msdn.microsoft.com/zh-cn/library/4cyf24ss.aspx) 和 [GetDirectories](https://msdn.microsoft.com/zh-cn/library/s7xk2b58.aspx)）将不返回重新分析点下的任何子目录。此行为可防止两个重新分析点相互引用时陷入无限循环。通常，为了确保您不会无意修改或删除文件，在使用重新分析点时应特别小心。如果需要精确控制重新分析点，请使用平台调用或本机代码直接调用相应的 Win32 文件系统方法。 |

下面的示例演示如何使用递归遍历目录树。递归方法很简洁，但如果目录树很大且嵌套很深，则有可能会引起堆栈溢出异常。

对于所处理的特定异常以及在每个文件和文件夹上执行的特定操作，都只是作为示例提供。您应该修改此代码来满足自己特定的需要。有关更多信息，请参见代码中的注释。

public class RecursiveFileSearch

{

static System.Collections.Specialized.StringCollection log = new System.Collections.Specialized.StringCollection();

static void Main()

{

// Start with drives if you have to search the entire computer.

string[] drives = System.Environment.GetLogicalDrives();

foreach (string dr in drives)

{

System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

// Here we skip the drive if it is not ready to be read. This

// is not necessarily the appropriate action in all scenarios.

if (!di.IsReady)

{

Console.WriteLine("The drive {0} could not be read", di.Name);

continue;

}

System.IO.DirectoryInfo rootDir = di.RootDirectory;

WalkDirectoryTree(rootDir);

}

// Write out all the files that could not be processed.

Console.WriteLine("Files with restricted access:");

foreach (string s in log)

{

Console.WriteLine(s);

}

// Keep the console window open in debug mode.

Console.WriteLine("Press any key");

Console.ReadKey();

}

static void WalkDirectoryTree(System.IO.DirectoryInfo root)

{

System.IO.FileInfo[] files = null;

System.IO.DirectoryInfo[] subDirs = null;

// First, process all the files directly under this folder

try

{

files = root.GetFiles("\*.\*");

}

// This is thrown if even one of the files requires permissions greater

// than the application provides.

catch (UnauthorizedAccessException e)

{

// This code just writes out the message and continues to recurse.

// You may decide to do something different here. For example, you

// can try to elevate your privileges and access the file again.

log.Add(e.Message);

}

catch (System.IO.DirectoryNotFoundException e)

{

Console.WriteLine(e.Message);

}

if (files != null)

{

foreach (System.IO.FileInfo fi in files)

{

// In this example, we only access the existing FileInfo object. If we

// want to open, delete or modify the file, then

// a try-catch block is required here to handle the case

// where the file has been deleted since the call to TraverseTree().

Console.WriteLine(fi.FullName);

}

// Now find all the subdirectories under this directory.

subDirs = root.GetDirectories();

foreach (System.IO.DirectoryInfo dirInfo in subDirs)

{

// Resursive call for each subdirectory.

WalkDirectoryTree(dirInfo);

}

}

}

}

下面的示例演示在不使用递归的情况下如何循环访问目录树中的文件和文件夹。该技术使用泛型 [Stack<T>](https://msdn.microsoft.com/zh-cn/library/3278tedw.aspx) 集合类型，该类型是一个后进先出 (LIFO) 堆栈。

对于所处理的特定异常以及在每个文件和文件夹上执行的特定操作，都只是作为示例提供。您应该修改此代码来满足自己特定的需要。有关更多信息，请参见代码中的注释。

public class StackBasedIteration

{

static void Main(string[] args)

{

// Specify the starting folder on the command line, or in

// Visual Studio in the Project > Properties > Debug pane.

TraverseTree(args[0]);

Console.WriteLine("Press any key");

Console.ReadKey();

}

public static void TraverseTree(string root)

{

// Data structure to hold names of subfolders to be

// examined for files.

Stack<string> dirs = new Stack<string>(20);

if (!System.IO.Directory.Exists(root))

{

throw new ArgumentException();

}

dirs.Push(root);

while (dirs.Count > 0)

{

string currentDir = dirs.Pop();

string[] subDirs;

try

{

subDirs = System.IO.Directory.GetDirectories(currentDir);

}

// An UnauthorizedAccessException exception will be thrown if we do not have

// discovery permission on a folder or file. It may or may not be acceptable

// to ignore the exception and continue enumerating the remaining files and

// folders. It is also possible (but unlikely) that a DirectoryNotFound exception

// will be raised. This will happen if currentDir has been deleted by

// another application or thread after our call to Directory.Exists. The

// choice of which exceptions to catch depends entirely on the specific task

// you are intending to perform and also on how much you know with certainty

// about the systems on which this code will run.

catch (UnauthorizedAccessException e)

{

Console.WriteLine(e.Message);

continue;

}

catch (System.IO.DirectoryNotFoundException e)

{

Console.WriteLine(e.Message);

continue;

}

string[] files = null;

try

{

files = System.IO.Directory.GetFiles(currentDir);

}

catch (UnauthorizedAccessException e)

{

Console.WriteLine(e.Message);

continue;

}

catch (System.IO.DirectoryNotFoundException e)

{

Console.WriteLine(e.Message);

continue;

}

// Perform the required action on each file here.

// Modify this block to perform your required task.

foreach (string file in files)

{

try

{

// Perform whatever action is required in your scenario.

System.IO.FileInfo fi = new System.IO.FileInfo(file);

Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);

}

catch (System.IO.FileNotFoundException e)

{

// If file was deleted by a separate application

// or thread since the call to TraverseTree()

// then just continue.

Console.WriteLine(e.Message);

continue;

}

}

// Push the subdirectories onto the stack for traversal.

// This could also be done before handing the files.

foreach (string str in subDirs)

dirs.Push(str);

}

}

}

通常，测试每个文件夹以确定您的应用程序是否有权限打开它是非常耗时的。因此，代码示例只是将该部分操作放到了一个 **try/catch** 块内。您可以修改该 **catch** 块，以便在访问某个文件夹遭受拒绝时，您可以提升自己的权限，然后再次访问它。通常，只捕捉那些无需使应用程序停留在一个未知状态就可以处理的异常。

如果必须将目录树的内容存储到内存或磁盘中，则最好只存储每个文件的 [FullName](https://msdn.microsoft.com/zh-cn/library/system.io.filesysteminfo.fullname.aspx) 属性（**string** 类型）。然后，可以根据需要使用该字符串创建一个新的 [FileInfo](https://msdn.microsoft.com/zh-cn/library/system.io.fileinfo.aspx) 或 [DirectoryInfo](https://msdn.microsoft.com/zh-cn/library/system.io.directoryinfo.aspx) 对象，或者打开任何需要进行其他处理的文件。

**可靠编程**

可靠文件迭代代码必须考虑文件系统的诸多复杂性。有关更多信息，请参见 [NTFS Technical Reference](http://go.microsoft.com/fwlink/?LinkId=79488)（NTFS 技术参考）。

**请参阅**

[System.IO](https://msdn.microsoft.com/zh-cn/library/system.io.aspx)

[LINQ and File Directories](https://msdn.microsoft.com/zh-cn/library/bb397911.aspx)

[文件系统和注册表（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/2kzb96fk.aspx)