파일 변환 Tool 개발

프로젝트 보고서

2023. 12. 22

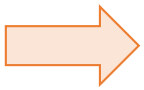
안세현

목차

1. 프로그램 구조

초기 구조

```
class Form {
  1 fileWatcher
  1 fileQueue
  1 monitoring thread
  3 worker thread
}
```



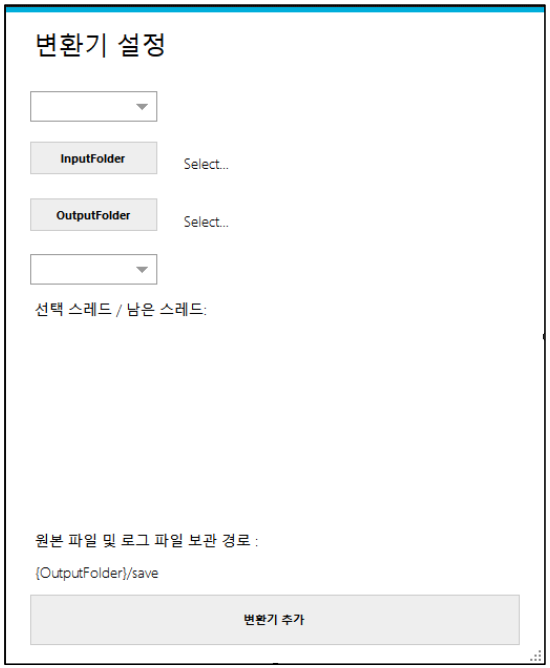
현재 구조

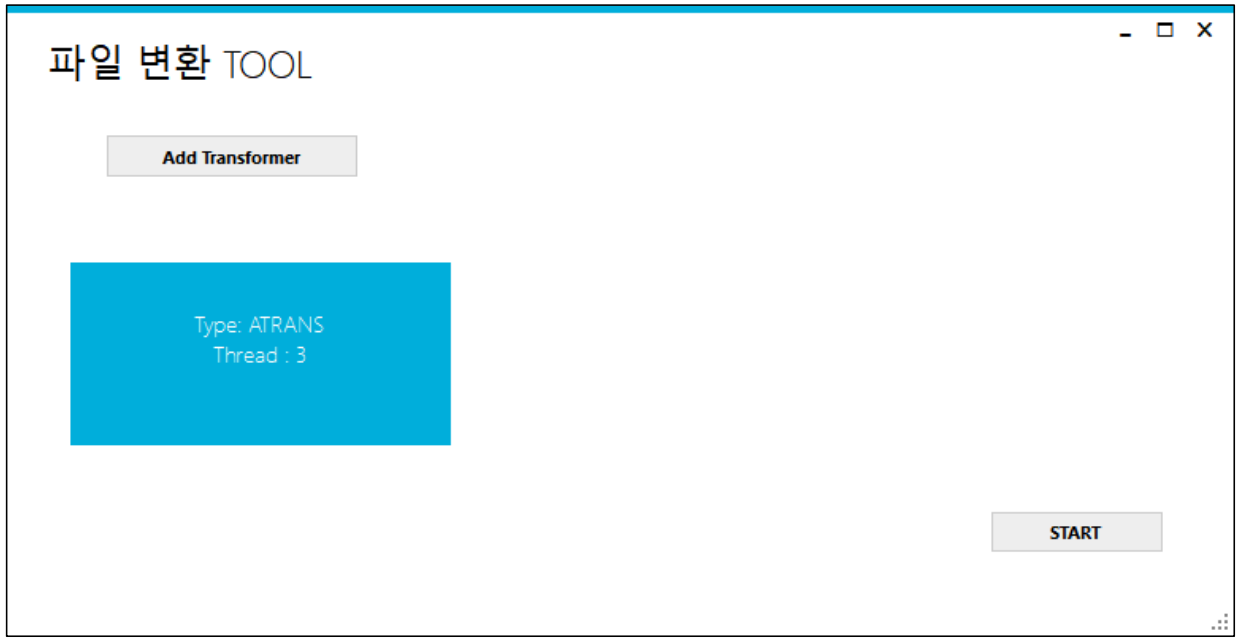
```
publicclass Form
{
    private List<Transformer> transformers = new List<Transformer>();
    private List<string> transformTypeList = new List<string>(); // { ATRANS, BTRANS ...}
    private int leftThread = 3

    private class Transformer
    {
      private string inputPath;
      private string outputPath;
      private int threadCount; // 모든 transformers list 의 threadCount의 합은 3 이하
      private int transformerIndex;
      private string transformType; // ATRANS, BTRANS, ...
      private Task[] workers; // worker의 수는 threadCount 만큼
      private BlockingCollection<string> fileQueue = new BlockingCollection<string>();
      FileSystemWatcher watcher;
    }
}
```

이 방식은 스레드 별 파일 변환 타입을 다르게 하거나,
스레드 별 경로를 다르게 할 때 문제가 발생함.

프로그램 클래스 내부에 Transformer 클래스를 둔 후,
각 Transformer가 **각각의 설정된 경로와 작업 스레드의 개수를 가지도록 설정**

1. 프로그램 구조





Transform Type은 지정된 만큼(현재는 ATRANS), Thread의 총 개수는 3개 이내로 Transformer를 추가할 수 있다.

2. 주요 기능 - 인코딩

■지원 인코딩

입력 파일

- **UTF-8(No BOM)**
- **UTF-8**
- **UTF-16 LE**
- **UTF-16BE**

출력 파일

- **UTF8 사용**

```
private static void SetEncoding(string filePath, ref FileData fileData)
{
    // 파일의 첫 부분에서 인코딩을 확인
    byte[] buffer = new byte[4];
    using (FileStream fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
    {
        fileStream.Read(buffer, 0, 4);
    }

    if (buffer[0] == 0xFF && buffer[1] == 0xFE)
    {
        fileData.encoding = Encoding.Unicode; // UTF-16 little-endian
    }
    else if (buffer[0] == 0xFE && buffer[1] == 0xFF)
    {
        fileData.encoding =  Encoding.BigEndianUnicode; // UTF-16 big-endian
    }
    else
    {
        fileData.encoding =  Encoding.UTF8; // 기본 인코딩 UTF8 사용
    }
}
```

BOM을 이용하여 알 수 있는 인코딩 종류:

- **UTF-8**
- **UTF-16LE**
- **UTF-16BE**
- UTF-32LE
- UTF-32BE

UTF-32는 현재 자주 사용하지 않으므로 처리하지 않았다.
UTF-8 의 경우, LE와 BE의 구분 없이 하나의 BOM으로 구분되는데
따라서 보통의 경우에는 UTF-8 인코딩 방식의 파일에는 BOM이

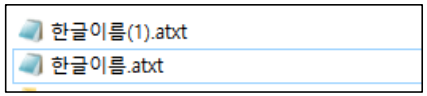
UTF-8 인코딩 방식은 자주 사용되지만,
사용 프로그램 등에 따라 BOM이 있을 수도, 없을 수도 있기 때문에
UTF-16 방식이 아닐 때엔
모두 UTF-8 을 기본 인코딩 방식으로 적용하여
BOM이 있는 파일과 없는 파일 모두 적용되도록 설정하였다.

2. 주요 기능 – 중복된 파일 이름 입력

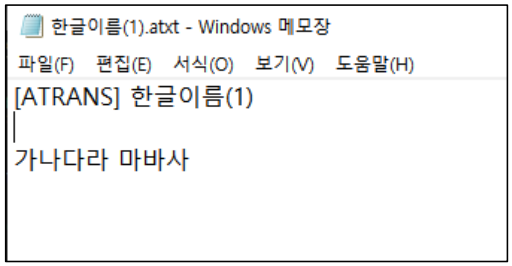
■같은 이름의 파일을 두 번 입력했을 때의 출력 파일

파일을 저장할 경로에 같은 이름의 파일이 존재할 경우,
해당 파일 이름 뒤에 (1), (2)... 등의 숫자를 붙여 새로운 파일 이름 생성

```
private string GetNewFileSavePath(string fileName, string extention, string savePath)
{
    string newFilePath = Path.Combine(savePath, fileName + extention);
    string newFileName = fileName;

    if (File.Exists(newFilePath))
    {
        // 파일 이름에 "(1)", "(2)" 등의 숫자를 붙여서 새로운 파일 이름 생성
        int counter = 1;
        string fileNameWithoutExtension = newFileName;
        do
        {
            newFileName = $"{fileNameWithoutExtension}({counter}){extention}";
            newFilePath = Path.Combine(savePath, newFileName);
            counter++;
        } while (File.Exists(newFilePath));
    }

    return newFilePath;
}
```

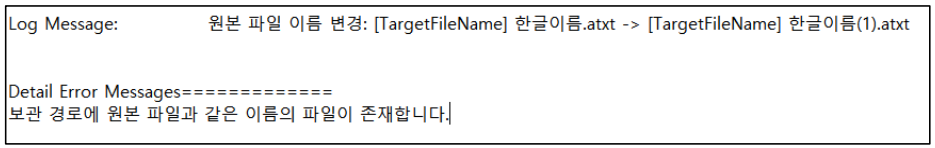


이미 '한글이름.atxt' 라는 파일이 출력 경로에 이미 존재하면,
출력 파일의 이름은 뒤에 (1) 이 붙은 '한글이름(1).atxt' 가 된다.



또한 출력되는 파일의 헤더에는, 출력 파일의 이름과 동일하게
이름에 (1)이 추가되어 **파일명과 헤더의 통일성을 유지**시킨다.

단, 원본 파일을 보관경로에 이동할 때,
보관 경로에 동일한 이름의 파일이 존재할 경우,
원본을 보존하기 위해 헤더는 변경하지 않고 파일명만 변경한 후,
로그파일에 해당 사실을 기록한다.



2. 주요 기능 – 대기 큐의 파일 변경

■ 입력 파일에 넣은 후, 파일을 변경할 시

- 이미 작업 중인 파일은 열려 있으므로 수정할 수 없음

➔ 작업 대기 중인 파일을 변경할 시

```
private Dictionary<string, string> renamedFilePath = new Dictionary<string, string>();
참조 2개
private void WatcherFileRenamed(object sender, RenamedEventArgs e)
{
    renamedFilePath.Add(e.OldFullPath, e.FullPath);
}
```

fileWatcher의 Renamed 이벤트를 받아온 후,
Dictionary 타입의 renamedFilePath에 이전 이름과 새로운 이름을 추가

```
if (File.Exists(currentTransfortFilePath) == false && renamedFilePath.ContainsKey(currentTransfortFilePath))
{
    currentTransfortFilePath = renamedFilePath[currentTransfortFilePath];
    renamedFilePath.Remove(currentTransfortFilePath);
}
```

fileQueue에서 변환할 파일을 꺼냈을 때, 해당 파일이 존재하지 않다면,
renamedFilePath에 이전 경로와 같은 값이 있는지 확인 후,
변경된 이름이 존재한다면 해당 파일을 변환.

이전 파일 이름과 새로운 파일 이름을 동시에 저장해야 했으며,
해당 자료 구조에 key 값이 존재하는지 확인해야 하기 때문에
Dictionary 구조를 사용하였다.

2. 주요 기능 – 변환 도중 프로그램 종료

■변환 도중 프로그램 종료 처리

- CancellationToken을 사용하여 스레드 작업을 취소

- 초기 구현
  -파일이 대기 큐에 있는 경우
  로그파일에 Thread Stop 으로 로그를 생성한 후,
  대기 큐에 있던 파일을 모두 보관경로로 이동
  - 파일을 열고 읽고 있는 경우
  로그 파일 없이,
  보관 경로에 이동하지 않고 그대로 프로그램 종료

-> **스레드 작업이 멈췄을 때의 결과에 통일성이 없음**

- 변경 후
  - 파일이 대기 큐에 있는 경우
    - 로그파일에 Thread Stop 으로 로그를 생성한 후, 보관 경로로 이동하지 않고 그대로 프로그램 종료
  - 파일을 열고 읽고 있는 경우
    - 로그 파일에 Thread Stop During Working 으로 로그를 생성한 후, 보관 경로로 이동하지 않고 그대로 프로그램 종료

중간에 프로그램이 종료한 경우, 오류 파일들이 '**완료되지 않은**' 상태이기 때문에
보관경로에 이동하지 않는 것으로 판단하였다.
하지만 사용자가 이전 작업에 대한 로그를 확인하고 싶어 할 수 있기 때문에,
로그 파일만 생성하여 보관 경로에 저장하였다.

2. 주요 기능 – 오류 메세지 실시간 업데이트

■ 새로운 창에 띄워지는 오류 메시지 리스트

-> 새로운 오류가 생성되면 실시간으로 바로 리스트에 오류 메시지 추가

```
private void thread1ErrorCnt_Click(object sender, EventArgs e)
{
    if (errorList.ContainsKey(0))
    {
        ObservableCollection<ListViewItem> items = errorList[0];

        if (errorWindow_1 == null || errorWindow_1.IsDisposed)
        {
            errorWindow_1 = new ErrorListForm(items, 0);
            errorWindow_1.Closed += (s, events) =>
            {
                errorWindow_1 = null;
            };
            items.CollectionChanged += (s, events) =>
            {
                if (errorWindow_1 != null )
                    errorWindow_1.UpdateErrorList(items);
            };
            errorWindow_1.Show();
        }
        else
        {
            errorWindow_1.Activate();
        }
    }
}
```

```
public void UpdateErrorList(ObservableCollection<ListViewItem> newList)
{
    if (errorListView.InvokeRequired)
    {
        errorListView.Invoke(new Action(() => UpdateErrorList(newList)));
    }
    else
    {
        items = newList;
        errorListView.Items.Clear();
        errorListView.Items.AddRange(items.ToArray());
    }
}
```

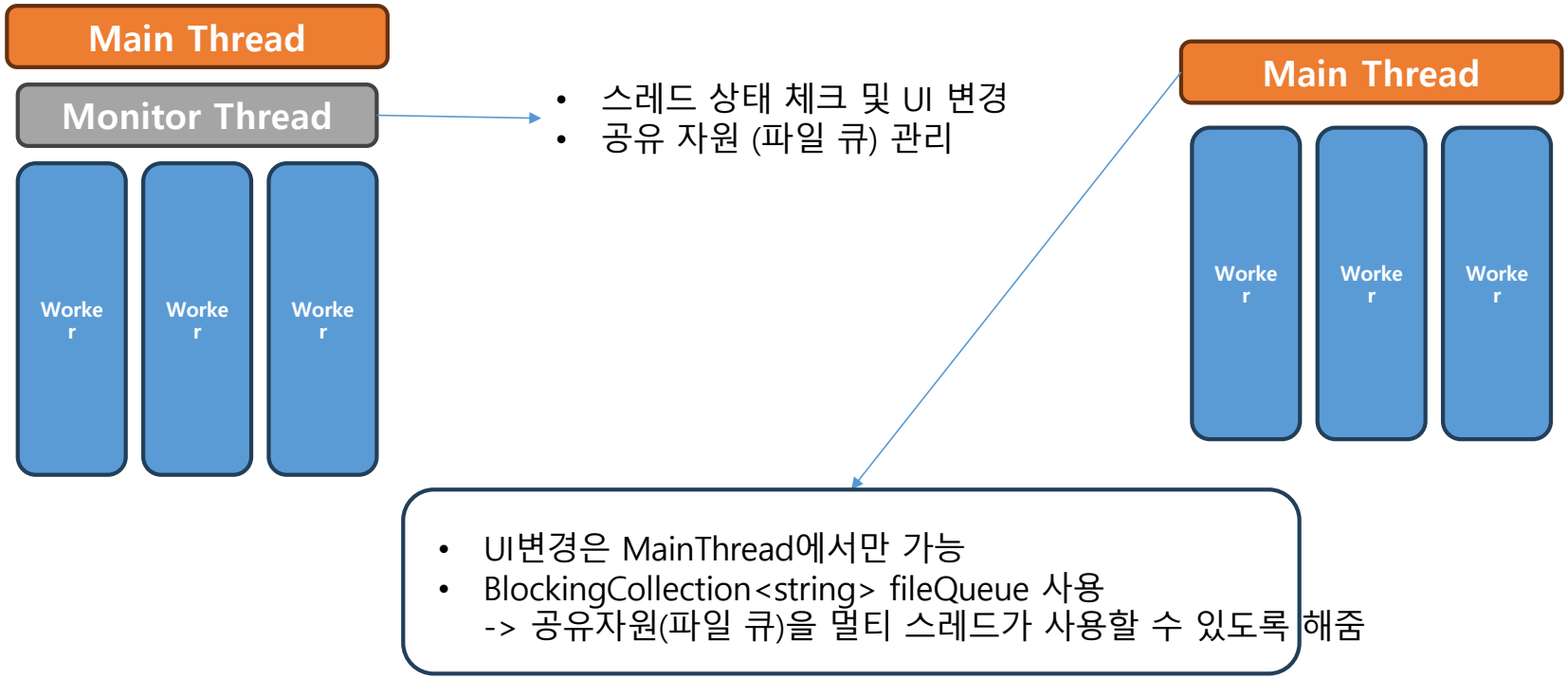
ObservavleCollection을 사용하여, errorList 에 새로운 값이 추가되면
바로 UpdateErrorList 함수가 동작하여 메시지를 추가하도록 구현

3. 고민했던 부분 – (C#) BackgroundWorker, Thread, ThreadPool, Task

1. BackgroundWorker
   - 일반적으로 UI를 주로 다룰 경우 BackgroundWorker를 사용하는게 적합
   - 하지만 Atrans 프로그램의 경우, UI업데이트보다는 변환이 주요 동작이기 때문에 변경
2. Thread
   - 기본적으로 foreground로 동작하며, 새로 스레드를 생성하고, 작업이 끝나면 스레드를 종료하여 없앤다.
   - 스레드를 생성하고 종료하는 작업이 많이 일어날 수록 성능이 떨어진다.
   - 3개 이상의 스레드를 생성하게 바뀐다면?
   - 혹은 스레드의 동작 방식이 바뀌어 무한 루프가 아니라 한 번의 작업 이후 스레드가 죽는다면?
3. ThreadPool
   - ThreadPool은 thread를 미리 생성해두고 pooling 하여 사용하는 방식
4. Task
   - ThreadPool 방식을 사용하며, 비동기 작업에 더 적합하고 편리하게 만들어주는 클래스
   - MS에서도 병렬 작업을 위해 더욱 권장하는 방법 (링크)

3. 고민했던 부분 – 모니터링 스레드

■모니터링 스레드 유무



감사합니다.