応用Javaプログラミング
第9回

1. マルチスレッド（応用編）
2. 生産者／消費者デザインパターン

「生産者／消費者」パターン

デッドロックが生じやすい例の１つ.

下記のようなシナリオ（「生産者／消費者」パターン）を考えよう:

- 各生産スレッド（producer, 生産者）は,
  データ（製品）を次々に作ってキューに格納する;
- 各消費スレッド（consumer, 消費者）は,
  キューに格納されたデータを次々に処理（消費）する.

注意1：生産スレッドが先行した場合には，キューが一杯になってしまい，
消費スレッドがデータを消費して キューに空き領域を作るまで，
生産スレッドは待たされる.

注意2：消費スレッドが先行した場合には，キューが空になってしまい，
生産スレッドがデータを生産して キューに詰めるまで，
消費スレッドは待たされる.

java.util.concurrent.BlockingQueue インターフェース

ブロッキングキュー：キューに要素を入れようとしたときや，キューから要素を取り出そうとしたときに，待たされる（ブロックする）ことがあるキューのこと．

主なメソッド

public abstract void put(E e) 

指定された要素をこのキューに挿入する．必要に応じて，空きが生じるまで待機．

public abstract E take( ) 

このキューの先頭を取得して削除する．必要に応じて，要素が利用可能になるまで待機．

public abstract boolean offer(E e)

指定された要素を，このキューに容量制限に違反することなしにすぐに挿入できる場合には，そうしてからtrueを返す．そうでない場合，falseを返す．

public abstract E poll(long timeout, TimeUnit unit)

このキューの先頭を取得して削除する．必要に応じて，指定された待機時間まで要素が利用可能になるのを待機する．

public abstract E peek( )

このキューの先頭を取得するが，削除しない．キューが空のとき，nullを返す．

java.util.concurrent.BlockingQueue インターフェースを実装した主なクラス

LinkedBlockingQueue: 連結リストを使って**BlockingQueue**インターフェースを実装したクラス．キューの保持できる要素の個数の上限を指定する必要がない．

LinkedBlockingDeque: 双方向連結リストを使って実装している点を除いて，**LinkedBlockingQueue**と同じ．

ArrayBlockingQueue: 配列を使って**BlockingQueue**インターフェースを実装したクラス．キューの保持できる要素の個数の上限を指定する必要がある．

PriorityBlockingQueue: 優先度ヒープを使って**BlockingQueue**インターフェースを実装したクラス．キューの保持できる要素の個数 の上限を指定する必要がない．

「生産者／消費者」パターンの具体例

コマンドラインから開始ディレクトリ および 検索キーワードを入力してから，開始ディレクトリ および その下にあるサブディレクトリにあるすべてのファイルを列挙して，その各ファイルで検索キーワードを検索して検索キーワードを含む行をすべて画面に表示したい．

●開始ディレクトリ および その下にあるサブディレクトリにあるすべてのファイルを列挙してブロッキングキューに登録していくのは速くできるので，1本の生産スレッドで行う．

●生産スレッドが列挙したファイルの中で検索キーワードを検索するのは大変な作業なので，複数の消費スレッドで行う．

Fileクラス

- ファイルおよびディレクトリのパス名の抽象表現（パス名や，セパレータ文字，最終更新日といったファイル関連の情報）を取得するために使われるクラス．

コンストラクタ（一部）
File(String pathName)
指定されたpathNameのファイルまたはディレクトリの抽象表現を構築．

メソッド（一部）
public String getPath( )
この抽象パス名をパス名文字列に変換して返す．
public boolean isFile( )
この抽象パス名が普通のファイルを示すなら trueを，さもなければ falseを返す．
public Files[ ] listFiles( )
この抽象パス名がディレクトリを示す場合，その下にあるファイル（サブディレクトリを含む）を示す抽象パス名の配列を返す；さもなければ，nullを返す．

列挙スレッドの仕事を表すクラス

```
import  java.io.* ; // Fileクラスを使うので必要
import  java.util.concurrent.* ; // BlockingQueueインターフェースを使うので必要

class  EnumerationTask implements Runnable {
    final static File DUMMY_FILE = new File("") ; //列挙の終了を示すのに使われる.
    private BlockingQueue<File> queue ;  // ブロッキングキューを指す変数.
    private File startDirectory ;  // 検索の開始ディレクトリを指す変数.

   // コンストラクタ(列挙の仕事を1つ作る)
    public EnumerationTask(BlockingQueue<File> queue, File startDirectory) {
        this.queue = queue; //ブロッキングキューへの参照を呼び出し側のプログラムからもらう.
        this.startDirectory = startDirectory ; // 検索の開始ディレクトリへの参照を呼び出し
                                               // 側のプログラムからもらう.
    }

    // 検索スレッドの行う仕事(指定されたディレクトリとそのサブディレクトリの下にあるファイルの列挙)
    public void run( ) {
        try {
            enumerate(startDirectory); //自作メソッド(次頁)でファイルを列挙してキューに追加.
            queue.put(DUMMY_FILE); // キューにダミーを置く(i.e., 列挙の完了の合図を送る).
        } catch ( InterruptedException e) {Thread.currentThread().interrupt( ) ; }
    }
    // enumerateメソッド(次頁)をここに書く.
} // EnumerationTask.java の終わり
```

EnumerationTask.java

指定されたディレクトリ および その下のサブディレクトリにあるファイルを列挙するには？

```
// 引数directoryの示すディレクトリの下にあるファイル（サブディレクトリを含む）を
// すべて列挙してブロッキングキューに置くためのメソッド.
public void enumerate( File directory ) throws InterruptedException {
    File[ ] files = directory.listFiles( ) ; // 引数directory の示すディレクトリのすぐ下
                                             // にあるすべてのファイル（ディレクトリを含む）を求める.
    if (files == null)   // 引数directory は正しいパス名ではなければ,
        return ;         // 何もしないで列挙を終える.
    for (int i = 0; i < files.length ; i++) {
        if ( files[i].isDirectory( ) ) // ディレクトリであれば, その下およびそのサブ
            enumerate( files[i] ) ; // ディレクトリの下にあるファイルを再帰的に列挙.
        else                  // ディレクトリではなければ（i.e., 普通のファイルであれば）,
            queue.put( files[i] ) ;  // それをブロッキングキューに追加する.
    }
} // enumerateメソッドの終わり
```

検索スレッドの仕事を表すクラス

```
import  java.io.* ; // Fileクラスを使うので必要
import  java.util.concurrent.* ; // BlockingQueueインターフェースを使うので必要
import  java.util.* ; // Scannerクラスを使うので必要

class  SearchTask implements Runnable {
   private BlockingQueue<File> queue ;  // ブロッキングキューを指す変数.
   private String keyword ;  // 検索キーワードを指す変数.

  // コンストラクタ（検索の仕事を1つ作る）
   public SearchTask(BlockingQueue<File> queue, String keyword) {
      this.queue = queue; //ブロッキングキューへの参照を呼び出し側のプログラムからもらう.
      this.keyword = keyword; // 検索キーワードへの参照を呼び出し側の プログラムからもらう.
   }

   public void run( ) { // 検索スレッドの行う仕事（ブロッキングキューからファイルを1つずつ
                        // 取ってきて，その中で検索キーワードを含む行をすべて見つける）.
      try {
         while ( true ) { // 下記をずっと繰り返し：
            File  file = queue.take( ) ; // キューからファイルを1つ取ってくる.
```

次頁に続く...

SearchTask.java

検索スレッドの仕事を表すクラス

```
            if ( file == EnumerationTask.DUMMY_FILE ) { // 取ってきたのがダミーなら,
                queue.put( file ) ; // それが終わりの合図なので, キューに戻す.
                return ; // 検索が終了した.
            }
            if ( ! file.canRead( ) ) continue ; //読み出せないならば検索をスキップ
            Scanner in = null ;
            try {
                in = new Scanner(file); // ファイルを開く.
                int lineNumber = 0 ; // ファイルの行の番号を覚えるための変数.
                while ( in.hasNextLine( ) ) {  // まだ検索していない行がある間,
                    lineNumber++ ;  // 行の番号を1増やし,
                    String line = in.nextLine( ) ; // 検索していない次の行を読み込み,
                    if (line.contains(keyword)) //その行が検索キーワードを含む場合結果表示.
                        System.out.println(file.getPath( ) + ":" + lineNumber + ":" + line);
                }                                  // containsはStringクラスのインスタンス・メソッドの1つ
            } finally {
                if (in != null) in.close( ) ; // 開いたファイルを閉じる.
            }
        }  // while-文の終わり
    } catch ( IOException e) {  e.printStackTrace() ;
    } catch ( InterruptedException e) {Thread.currentThread().interrupt( ) ; }
  } // runメソッドの終わり
} // SearchTask.java の終わり
```

SearchTask.java

KeywordSearchクラス

```
import  java.io.* ; // Fileクラスを使うので必要
import  java.util.concurrent.* ; // BlockingQueueインターフェースを使うので必要
import  java.util.* ; // Scannerクラスを使うので必要

public  class  KeywordSearch {
    private static final int  SIZE = 10;  // ブロッキングキューに置けるファイルの最大個数.
    private static final int  SEARCHER = 2 ;  // 検索を担当するスレッドの本数.
          // 2でなく, Runtime.getRuntime().availableProcessors() – 2 にしたほうがよい.

    public static void main(String[ ] args ) {
       // 指定されたサイズのブロッキングキューを作成する
       BlockingQueue<File> queue = new ArrayBlockingQueue<File>( SIZE ) ;

       Scanner in = new Scanner(System.in); // キーボードからの入力を行うスキャナーを生成.

       System.out.print( "開始ディレクトリ:" );  //利用者に開始ディレクトリの入力を催促.
       String directory = in.nextLine( ) ; // キーボードから開始ディレクトリを入力する.

       System.out.print( "キーワード:" );  //利用者にキーワードの入力を催促する.
       String keyword = in.nextLine( ) ; // キーボードからキーワードを入力する.

       File  startDirectory = new File( directory ) ; // 開始ディレクトリの抽象表現を生成.

       // 列挙スレッドの仕事を生成する.
       EnumerationTask et = new EnumerationTask(queue, startDirectory) ;
```

次頁に続く...

KeywordSearch.java

KeywordSearchクラス

```
        // ファイルの列挙を担当するスレッドを1本作成する.
        Thread enumerator = new Thread( et ) ;

        // 検索スレッドをたくさん生成する.
        Thread[ ] searchers = new Thread[SEARCHER]; //検索スレッドを覚える配列を生成.
        for (int i = 0 ; i < SEARCHER ; i++)
            // i番目の検索スレッドを作成する.
            searchers[i] = new Thread(new SearchTask(queue, keyword)) ;

        // ファイルの列挙を担当するスレッドを起動する.
        enumerator.start( ) ;

        // 検索スレッドを起動する.
        for (int i = 0 ; i < SEARCHER ; i++)
            searchers[i].start( ) ; // i番目の検索スレッドを起動する.
    } // mainメソッドの終わり
} // KeywordSearch.javaの終わり
```

KeywordSearch.java

「生産者／消費者」パターンの具体例2

素因数分解を例にする.

- 生産スレッド(producer)が，次々にランダムに整数(10000以上)を生成して，それが合成数であればキューに置く.
  キューが満杯になったら，待たされる.
- 消費スレッド(consumer)が，キューから合成数を取得してその素因数分解を求めて画面に表示する.
  キューが空になったら，待たされる.
- 合計5000個の合成数の素因数分解を求めたらプログラムを終了.

課題：「生産者／消費者」パターンの具体例2のプログラムを書け.

ただし，
- 生産者数：1人
- 消費者数：Runtime.getRuntime( ).availableProcessors( ) – 2
- キューの容量：100

とする.

「生産者／消費者」パターンの具体例2

前頁の課題のヒント:

① 1つの合成数の素因数分解を1つの文字列にまとめてから画面に表示したほうがよいだろう.

② 念のために，下記のメソッド（System.out.printlnのスレッドセーフ版）を使ってみてよい.

```
public void safePrintln(String s) {
    synchronized (System.out) {
        System.out.println(s);
    }
}
```