Java 最適化手法

Java アプリのパフォーマンスを最大限引き出す実践ガイド

Maarten De Cock
Application Engineer
ASQdotCOM

Erwin Vervaet
Senior Software Engineer
Ervacon

2002年 6月 11日

Javaプログラムの最適化に役立つ手法は数多くあります。この記事では、特定の1つの手法に焦点を合わせるのではなく最適化プロセスの全体を考察します。著者のErwin VervaetとMaarten De Cockは、パズルを解くプログラムのパフォーマンスのチューニングについて、簡単なテクニカル・ヒントからもっと高度なアルゴリズムの最適化までの手法を織り交ぜながら応用し、読者のみなさんに解説します。最初の実装と完全に最適化されたソリューションとの間に最終結果として目覚ましいパフォーマンスの向上 (100万 倍以上) が見られます。

Javaのパフォーマンスに関連した記事の多くは、プログラマーが自分たちのプログラムの速度を向上させるために使用できる多くの手法に焦点を合わせています。一方の側では、`StringBuffer` クラスの利用法などの比較的単純なプログラミング・イディオムの説明が行われてます。もう一方の側では、オブジェクト・キャッシュの利用法などのもっと高度な手法の検討が行われています。ここでは、そのような手法のリストに加わるのではなく、それらを結合してパズルを解くプログラムの速度を高める実践的な例を紹介します。

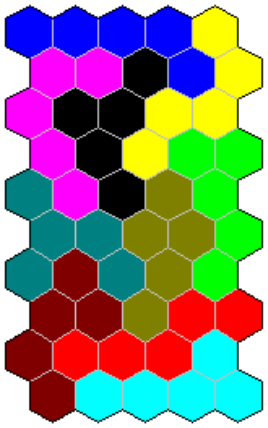
ここで開発して最適化するプログラムは、Meteorパズルのあらゆる可能な解を計算します。Meteorパズルは、1つピースが5つの六角形 (各辺の長さが等しい6辺の多角形) で構成され、色がそれぞれ異なる10個のパズル・ピースから成る問題です。パズル・ボード自体は5 x 10のパターンの50個の六角形のすの目がある長方形のグリッドです。このパズルの解は、使用できる10個のピースを使ってボード全体を埋めることです。このパズルの考えられる解の1つを図1に示します。

Eternityパズル

この記事で取り上げているJavaプログラムは10ピースのMeteorパズルを解いていますが、本当のゴールはEternityパズルと呼ばれるこれよりもずっと大規模な209ピースのパズルを解くことでした。これはChristopher Monckton によって考案され、1999年6月に英国で発表されたものです。Eternityのリリースと同時に、MonckntonはMeteor、Delta、Heartといったそれよりも小規模の多様なパズルをリリースしました。これらのパズルのいずれかを解いたプ

レーヤーは、Monckton の解の中で特定のピースがEternityグリッドのどこに置かれるかを示す様々なヒントの中の1つを取り寄せることができました。Eternityパズルを最初に解いた人への100万ポンド (約150万米ドル) の賞金が提示され、これは最終的にはAlex SelbyとOliver Riordanが2000年5月15日に手にしました。2番目の解は後にGuenter Stertenbrinkによって見つけられました。面白いことには、これらの解のいずれもChristopher Monckton が自身の解 (不明のままです) に合わせて与えた6つの手掛かりとは一致しませんでした。

図1. Meteoパズルの1つの解 (A solution for the Meteor puzzle)



この解を求めるのは簡単なように思えるかもしれませんが、コンピューター・プログラムに実装するのは、それほど簡単な自明のことではありません。そのプログラムを書くことは、他の多くのJavaのパフォーマンス関連の記事にある例と違って新鮮な感じになるでしょう。またそのプログラムによって多数の異なる最適化手法とそれらの結合方法を説明することができます。しかし最適化を始める前に、まず実際に動作するソリューションを開発する必要があります。

まず、実際に動作するソリューション

このセクションではパズルを解くプログラムの最初の実装を議論します。このプログラムには、かなりの量のコードがありますが辛抱してください。まず関係している基本アルゴリズムを説明してから最適化を開始します。この最初の実装のソース・コード、およびこの記事で後述する最適化ソース・コードも、参考文献から入手できます。

パズルを解くアルゴリズム

このパズルを解くプログラムは、Meteorパズルのあらゆる可能な解を計算します。つまりこのことは、考え得るすべてのタイリング方法を余すところなく求めることを意味しています。このタスクを達成するための最初のステップは、1ピースのすべての順列 を決定することです。ここで言う順列とはピースをボードに配置するときの考えられる方法のことです。すべてのピースは左右を逆にして反転でき、その中の1つの六角形の6辺の回りに回転できるので、1つのピースをボードの1つの位置に置く可能な方法は合計12 (2 x 6) 通りであることが分かります。ボードには50の位置があるので、単一のピースをボードに置くときの考えられる方法の合計数は600 (2 x 6 x 50) になります。

もちろん、これらの「可能性」のすべてが実際に有効であるわけではありません。たとえば、ボードの端からピースが突き出るものがあり、これはもちろん解にはなりません。このプロセスをすべてのピースに対して再帰的に繰り返すのが最初のアルゴリズムです。この場合はピースは考え得るすべてのタイリングを試行することにより、可能なすべての解を求めます。リスト1は

このアルゴリズのコードを示したものです。すべてのピースを保持するのに`pieceList` という単純な`ArrayList` オブジェクトを利用しています。`board` オブジェクトはパズル・ボードを表しており、これについては後で簡単に説明します。

リスト1. 最初のパズルを解くアルゴリズム (The initial puzzle-solving algorithm)

```
public void solve() {
  if (!pieceList.isEmpty()) {
    // Take the first available piece
    Piece currentPiece = (Piece)pieceList.remove(0);

    for (int i = 0; i < Piece.NUMBEROFPERMUTATIONS; i++) {
      Piece permutation = currentPiece.nextPermutation();

      for (int j = 0; j < Board.NUMBEROFCELLS; j++) {
        if (board.placePiece(permutation, j)) {

          /* We have now put a piece on the board, so we have to
             continue this process with the next piece by
             recursively calling the solve() method */

          solve();

          /* We're back from the recursion and we have to continue
             searching at this level, so we remove the piece we
             just added from the board */

          board.removePiece(permutation);
        }
        // Else the permutation doesn't fit on the board
      }
    }

    // We're done with this piece
    pieceList.add(0, currentPiece);
  }
  else {

    /* All pieces have been placed on the board so we
       have found a solution! */

    puzzleSolved();
  }
}
```

これで基本アルゴリズムをセットアップしましたが、その他に次の2つの重要なことを調べる必要があります。

- パズルのピースを表す方法。
- パズル・ボードを実装する方法。

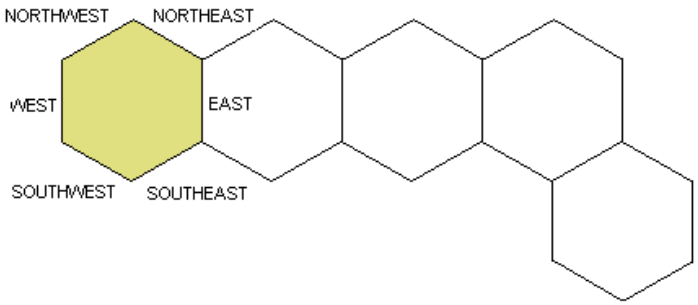
リスト1に示したアルゴリズムでは、`Piece` クラスと`Board` クラスを利用しました。ここでその2つのクラスの実装を考えてみましょう。

Pieceクラス

`Piece` クラスの設計を開始する前に、このクラスが何を表すべきかを検討する必要があります。図2を見ると、Meteorパズルのピースが5つの接続されたセルから成っていることが分かります。各セルは、EAST、SOUTHEAST、SOUTHWEST、WEST、NORTHWEST、およびNORTHEASTの6辺を持つ正六角形です。ピースの中の2つのセルが特定の辺で結合しているとき、これらのセルを近

隣 (neighbours) と呼びます。結局Pieceオブジェクトは、接続された5つのCellオブジェクトの集合以上の何物でもありません。各Cellオブジェクトには6つの辺と、可能性として6つの隣接するセルがあります。Cellクラスの実装はリスト2に示すように簡単です。Cellオブジェクトの中にprocessedフラグを保持していることに注意してください。このフラグはあとで無限ループを回避するのに利用します。

図2. パズル・ピースとそのセル (A puzzle piece and its cells)



リスト2. Cellクラス (The Cell class)

```
public class Cell {
  public static final int NUMBEROFSIDES = 6;

  // The sides of a cell
  public static final int EAST      = 0;
  public static final int SOUTHEAST = 1;
  public static final int SOUTHWEST = 2;
  public static final int WEST      = 3;
  public static final int NORTHWEST = 4;
  public static final int NORTHEAST = 5;

  private Cell[] neighbours = new Cell[NUMBEROFSIDES];

  private boolean processed = false;

  public Cell getNeighbour(int side) {
    return neighbours[side];
  }

  public void setNeighbour(int side, Cell cell) {
    neighbours[side] = cell;
  }

  public boolean isProcessed() {
    return processed;
  }

  public void setProcessed(boolean b) {
    processed = b;
  }
}
```

Pieceの順列を計算するメソッドが必要なため、Pieceクラスはもっと興味深いものです。最初にピースをその中の1つのセルの6辺の回りを1ステップ回転させ、それを左右逆に反転し、次々にその6辺の回りを再度回転させ左右逆転させることにより、すべての順列を求めることができます。前述のようにピースは5つの隣接セルから成ります。ピースを反転したり回転させることは、単にそのセルのすべてを反転したり回転させることです。したがってCellオブジェクトにはflip()およびrotate()メソッドが必要です。反転と回転は隣接する辺を適宜変更することにより容易に

行えます。これらのメソッドは、リスト3に示すCellクラスのPieceCellサブクラスで規定されます。PieceCellオブジェクトはPieceオブジェクトで利用されるセルです。

リスト3. PieceCellサブクラス (The PieceCell subclass)

```
public class PieceCell extends Cell {
  public void flip() {
    Cell buffer = getNeighbour(NORTHEAST);
    setNeighbour(NORTHEAST, getNeighbour(NORTHWEST));
    setNeighbour(NORTHWEST, buffer);
    buffer = getNeighbour(EAST);
    setNeighbour(EAST, getNeighbour(WEST));
    setNeighbour(WEST, buffer);
    buffer = getNeighbour(SOUTHEAST);
    setNeighbour(SOUTHEAST, getNeighbour(SOUTHWEST));
    setNeighbour(SOUTHWEST, buffer);
  }

  public void rotate() {
    // Clockwise rotation
    Cell eastNeighbour = getNeighbour(EAST);
    setNeighbour(EAST, getNeighbour(NORTHEAST));
    setNeighbour(NORTHEAST, getNeighbour(NORTHWEST));
    setNeighbour(NORTHWEST, getNeighbour(WEST));
    setNeighbour(WEST, getNeighbour(SOUTHWEST));
    setNeighbour(SOUTHWEST, getNeighbour(SOUTHEAST));
    setNeighbour(SOUTHEAST, eastNeighbour);
  }
}
```

PieceCellクラスを利用してPieceクラスの実装を完成させます。リスト4にそのソース・コードを示します。

リスト4. Pieceクラス (The Piece class)

```
public class Piece {
  public static final int NUMBEROFCELLS = 5;
  public static final int NUMBEROFPERMUTATIONS = 12;

  private PieceCell[] pieceCells = new PieceCell[NUMBEROFCELLS];
  private int currentPermutation = 0;

  private void rotatePiece() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].rotate();
    }
  }

  private void flipPiece() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].flip();
    }
  }

  public Piece nextPermutation() {
    if (currentPermutation == NUMBEROFPERMUTATIONS)
      currentPermutation = 0;

    switch (currentPermutation%6) {
      case 0:
        // Flip after every 6 rotations
        flipPiece();
        break;
```

```
      default:
        rotatePiece();
        break;
    }

    currentPermutation++;

    return this;
  }

  public void resetProcessed() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].setProcessed(false);
    }
  }

  //Getters and setters have been omitted
}
```

Boardクラス

`Board` クラスを実装する前に、2つの興味深い問題に取り組む必要があります。まず最初にデータ構造を決めなければなりません。Meteorパズル・ボードは、基本的には正六角形の5 x 10のグリッドです。これは50の`Cell` オブジェクトの配列として表すことができます。`Cell` クラスを直接利用するのではなく、リスト5に示す`BoardCell` サブクラスを利用します。これはセルを占有するピースを追跡するものです。

リスト5. BoardCellサブクラス (The BoardCell subclass)

```
public class BoardCell extends Cell {
  private Piece piece = null;

  public Piece getPiece() {
    return piece;
  }

  public void setPiece(Piece piece) {
    this.piece = piece;
  }
}
```

ボードの50のボード・セルをすべて配列でストアする場合は、かなり単調な初期設定コードを書くことになります。この初期設定で、図3に示すようにボードの各セルの隣接ボード・セルを識別します。たとえばセル0にはeastにセル1、southeastにセル5の2つの近隣があります。リスト6 に、この初期設定を行うために`Board` クラスのコンストラクターから呼び出される`initializeBoardCell()` メソッドを示します。

図3. セルの配列として表されるボード (The board represented as an array of cells)

ボードのデータ構造を実装したので、次の問題に移ります。ピースをボードに置く`placePiece()` メソッドの作成です。このメソッドを作成する上で最も難しい部分は、ピースがボードの所定の位置に収まるかどうかを決定することです。ピースが収まるかどうかを判別する1つの方法は、まず最初に、ピースがボードに置かれた場合の、そのピースの各セルで占有される

すべてのボード・セルを求めることです。このボード・セルの集合が分かると、新しいピースが収まるかどうかは容易に判別することができます。つまり対応するボード・セルはすべて空である必要があり、そのピースはボードに完全に収まる必要があります。このプロセスはリスト6に示す`findOccupiedBoardCells()`メソッドと`placePiece()`メソッドで実装されます。`findOccupiedBoardCells()`メソッドでの無限再帰を回避するために`PieceCell`オブジェクトの`processed`フィールドを利用することに注意してください。

リスト6. Boardクラス (The Board class)

```
public class Board {
  public static final int NUMBEROFCELLS = 50;
  public static final int NUMBEROFCELLSINROW = 5;

  private BoardCell[] boardCells = new BoardCell[NUMBEROFCELLS];

  public Board() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      boardCells[i] = new BoardCell();
    }

    for (int i = 0; i < NUMBEROFCELLS; i++) {
      initializeBoardCell(boardCells[i], i);
    }
  }

  /**
   * Initialize the neighbours of the given boardCell at the given
   * index on the board
   */
  private void initializeBoardCell(BoardCell boardCell, int index) {
    int row = index/NUMBEROFCELLSINROW;

    // Check if cell is in last or first column
    boolean isFirst = (index%NUMBEROFCELLSINROW == 0);
    boolean isLast = ((index+1)%NUMBEROFCELLSINROW == 0);

    if (row%2 == 0) { // Even rows
      if (row != 0) {
        // Northern neighbours
        if (!isFirst) {
          boardCell.setNeighbour(Cell.NORTHWEST, boardCells[index-6]);
        }
        boardCell.setNeighbour(Cell.NORTHEAST, boardCells[index-5]);
      }
      if (row != ((NUMBEROFCELLS/NUMBEROFCELLSINROW)-1)) {
        // Southern neighbours
        if (!isFirst) {
          boardCell.setNeighbour(Cell.SOUTHWEST, boardCells[index+4]);
        }
        boardCell.setNeighbour(Cell.SOUTHEAST, boardCells[index+5]);
      }
    }
    else { // Uneven rows
      // Northern neighbours
      if (!isLast) {
        boardCell.setNeighbour(Cell.NORTHEAST, boardCells[index-4]);
      }
      boardCell.setNeighbour(Cell.NORTHWEST, boardCells[index-5]);
      // Southern neighbours
      if (row != ((NUMBEROFCELLS/NUMBEROFCELLSINROW)-1)) {
        if (!isLast) {
          boardCell.setNeighbour(Cell.SOUTHEAST, boardCells[index+6]);
        }
        boardCell.setNeighbour(Cell.SOUTHWEST, boardCells[index+5]);
```

```
    }
  }

  // Set the east and west neighbours
  if (!isFirst) {
    boardCell.setNeighbour(Cell.WEST, boardCells[index-1]);
  }
  if (!isLast) {
    boardCell.setNeighbour(Cell.EAST, boardCells[index+1]);
  }
}

public void findOccupiedBoardCells(
  ArrayList occupiedCells, PieceCell pieceCell, BoardCell boardCell) {
  if (pieceCell != null & boardCell != null & !pieceCell.isProcessed()) {
    occupiedCells.add(boardCell);

    /* Neighbouring cells can form loops, which would lead to an
       infinite recursion. Avoid this by marking the processed
       cells. */

    pieceCell.setProcessed(true);

    // Repeat for each neighbour of the piece cell
    for (int i = 0; i < Cell.NUMBEROFSIDES; i++) {
      findOccupiedBoardCells(occupiedCells,
                             (PieceCell)pieceCell.getNeighbour(i),
                             (BoardCell)boardCell.getNeighbour(i));
    }
  }
}

public boolean placePiece(Piece piece, int boardCellIdx) {
  // We will manipulate the piece using its first cell
  return placePiece(piece, 0, boardCellIdx);
}

public boolean
  placePiece(Piece piece, int pieceCellIdx, int boardCellIdx) {
  // We're going to process the piece
  piece.resetProcessed();

  // Get all the boardCells that this piece would occupy
  ArrayList occupiedBoardCells = new ArrayList();
  findOccupiedBoardCells(occupiedBoardCells,
                         piece.getPieceCell(pieceCellIdx),
                         boardCells[boardCellIdx]);

  if (occupiedBoardCells.size() != Piece.NUMBEROFCELLS) {
    // Some cells of the piece don't fall on the board
    return false;
  }

  for (int i = 0; i < occupiedBoardCells.size(); i++) {
    if (((BoardCell)occupiedBoardCells.get(i)).getPiece() != null)
      // The board cell is already occupied by another piece
      return false;
  }

  // Occupy the board cells with the piece
  for (int i = 0; i < occupiedBoardCells.size(); i++) {
    ((BoardCell)occupiedBoardCells.get(i)).setPiece(piece);
  }

  return true; // The piece fits on the board
}
```

```
  public void removePiece(Piece piece) {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      // Piece objects are unique, so use reference equality
      if (boardCells[i].getPiece() == piece) {
        boardCells[i].setPiece(null);
      }
    }
  }
}
```

これで最初のソリューションの実装が完了しました。これをテストしてみましょう。

プログラムの実行

最初のパズルを解くプログラムは完成したので、これを実行してMeteorパズルのあらゆる可能な解を求めることができます。前のセクションに記載したソース・コードは、ソース・ダウンロードの`meteor.initial`パッケージにあります。このパッケージにはこのプログラムを始動させるための`solve()`メソッドと`main()`メソッドがある`Solver`クラスが含まれています。`Solver`クラスのコンストラクターはすべてのパズル・ピースの初期設定を行い、それらを`pieceList`に追加します。`java meteor.initial.Solver`を利用してプログラムを立ち上げることができます。

プログラムは解をサーチし始めますが、お気付きになるでしょうがプログラムが何かを求めているようには思えません。実際はプログラムは可能なあらゆる解を求めているのですが、非常な忍耐が必要になります。1つの解を求めるだけでも数時間かかります。私たちがテストに使用したコンピューターはAthlon XP 1500+ (512MB RAM) でRedHat Linux 7.2とJava 1.4.0が動作するものですが、最初の解を見つけたのは約8時間後です。すべての解を求めるとすれば、何年ではないにしても、数か月かかるでしょう。

明らかにパフォーマンスの問題があります。最適化の第一候補はパズルを解くアルゴリズムです。可能なすべての解を求めるのに、現在は単純で力ずくの方法を使っています。このアルゴリズムを微調整してみる必要があります。次にできることは、一時データをキャッシュすることです。たとえば、ピースの順列を毎回再計算するのではなく、それらの順列をキャッシュするのです。最後に、不要なメソッド呼び出しを回避するなどの、低水準の最適化手法をいくつか適用してみます。次のセクションでは、これらの最適化手法を検討します。

アルゴリズムの改良

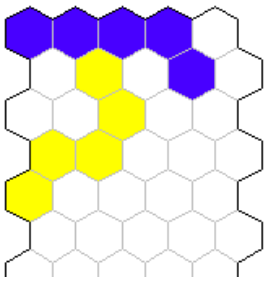
リスト1 を振り返って、最初のパズルを解くアルゴリズムをどのように最適化できるかを考えます。アルゴリズムを最適化するための良い方法は、それをビジュアル化することです。ビジュアル化することにより、実装されているプロセスとそこでの潜在的な弱点をより理解できるようになります。次のセクションでは、認識できる2つの非効率な部分について検討します。パズルを解くプログラムの実際のビジュアル化コードは、関心のある読者にお任せします。

島検出による枝刈り

リスト1 のアルゴリズムは、ピース (もっと正確に言えばピースのピース・セル) をボードのすべての位置にはめ込みます。図4はプロセスの開始時にあり得るボードの状態を示したものです。青色のピースの現在の順列は最初に使用可能なボード位置に置かれており、黄色のピースの現在の順列はそれが2番目に使用可能なボード位置に移動しています。このアルゴリズムでは3番目以降のピースでも同様に続けます。しかし図4を注意深く見ると、青色と黄色のピースがこれらの位置の場合はパズルの可能な解はないことは明らかです。その理由は、それらの2つのピースは3つの

隣接する空セルの島 を形成しているからです。パズル・ピースはすべて5つのセルから成っているため、この島を埋めようがありません。このアルゴリズムが残りの8つのピースをボードに収めようとする試みはすべて無駄なことです。ボード上で埋めることができない島を検出した場合は、このアルゴリズムを打ち切ることが必要です。

図4. ボード上の島 (An island on the board)



再帰的サーチ・アルゴリズムへの割り込みのこのプロセスを、テキスト・ブックでは枝刈り と呼んでいます。ここでの`Solver` クラスに枝刈り機能を追加するのは容易です。`solve()` メソッドの再帰呼び出しのたびに、その前にボード上の島をチェックします。5の倍数ではない数の空セルから成る島を見つけた場合は、再帰呼び出しを行いません。その代わり、アルゴリズムは現行の再帰レベルで継続します。リスト7と8は必要なコードの調整を示したものです。

リスト7. 枝刈りを使用してパズルを解くアルゴリズム (A puzzle-solving algorithm with pruning)

```
public class Solver {
  public void solve() {
    ...
            if (!prune()) solve();
    ...
  }

  private boolean prune() {
    /* We'll use the processed field of board cells to avoid
    infinite loops */
    board.resetProcessed();

    for (int i = 0; i < Board.NUMBEROFCELLS; i++) {
      if (board.getBoardCell(i).getIslandSize()%Piece.NUMBEROFCELLS != 0) {
        // We have found an unsolvable island
        return true;
      }
    }

    return false;
  }
}
```

リスト8. getIslandSize() メソッド (The getIslandSize() method)

```
public class BoardCell {
  public int getIslandSize() {
    if (!isProcessed() & isEmpty()) {
      setProcessed(true); // Avoid infinite recursion
      int numberOfCellsInIsland = 1; // this cell

      for (int i = 0; i < Cell.NUMBEROFSIDES; i++) {
        BoardCell neighbour=(BoardCell)getNeighbour(i);
        if (neighbour != null) {
          numberOfCellsInIsland += neighbour.getIslandSize();
        }
      }
      return numberOfCellsInIsland;
    }
    else {
      return 0;
    }
  }
}
```

充てんアルゴリズム

最初のアルゴリズムの2番目の弱点は、本質的に多数の島を生成することです。これはピースの1順列をとり、次の順列に移る前にボード上を移動させるために起こります。たとえば図5では、青色のピースの現行順列をその3番目に使用可能なボード位置に移動しています。お分かりのように、これによってボードの最上部に島が生成されています。多数の島が生成されるため前のセクションで追加した島検出による枝刈りによってパフォーマンスは大幅に向上しますが、生成される島の数を最初に最小限からするようにアルゴリズムを更新できればさらに良くなります。

図5. 島の生成 (Generating islands)

生成される島の数を減らすには、アルゴリズムで空のボード位置を充てんすることを集中的に行えればそれが最もよい方法でしょう。したがって、ボードのタイリングのあらゆる可能な方法の試行に焦点を合わせるだけでなく、ボードを左から右に、上から下に充てんしてみましょう。新しいパズルを解くアルゴリズムをリスト9に示します。

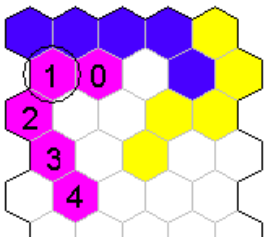
リスト9. 充てんパズルを解くアルゴリズム (The fill-up puzzle-solving algorithm)

```
public void solve() {
  if (!pieceList.isEmpty()) {
    // We'll try to find a piece that fits on this board cell
    int emptyBoardCellIdx = board.getFirstEmptyBoardCellIndex();

    // Try all available pieces
    for (int h = 0; h < pieceList.size(); h++) {
      Piece currentPiece = (Piece)pieceList.remove(h);

      for (int i = 0; i < Piece.NUMBEROFPERMUTATIONS; i++) {
        Piece permutation = currentPiece.nextPermutation();

        /* Instead of always using the first cell to manipulate
```

```
            the piece, we now try to fit any cell of the piece on
            the first empty board cell */

          for (int j = 0; j < Piece.NUMBEROFCELLS; j++) {
            if (board.placePiece(permutation, j, emptyBoardCellIdx)) {
              if (!prune()) solve();
              board.removePiece(permutation);
            }
          }
        }


        /* Put the piece back into the list at the position where
           we took it to maintain the order of the list */

        pieceList.add(h, currentPiece);
      }
    }
    else {
      puzzleSolved();
    }
}
```

新しい方法では、使用可能なピースを最初の空のボード・セルに収めることを試みます。すべての使用可能なピースのすべての可能な順列を試みるだけでは十分ではありません。空のボード・セルをピース内のいずれのピース・セルでも埋めてみる必要があります。最初のアルゴリズムでは、ピースをその1番目のセルを使って操作することを暗黙のうちに想定していました。今度は図6に示すように、ピース内のすべてのセルを試みる必要があります。ピンクのピースの現行の順列は、インデックス0のピース・セルをボード位置5 (図6で円で囲まれた位置) に置こうとしてもボードに収まりませんが、しかし、2番目のピース・セルを使用すると収まります。

図6. ピースのセル (The cells of a piece)



更新したプログラムの実行

最初のプログラムを実行したときは、適度な時間内に解を求めることはうまくいきませんでした。改良したアルゴリズムと島検出による枝刈りを使って再試行してみましょう。このバージョンのプログラムのコードは、パッケージ`meteor.algorithm` にあります。このプログラムを`java meteor.algorithm.Solver` を利用して立ち上げると、ほとんどすぐに解がポップアップ表示されます。私たちがテストに使用したコンピューターでは、157秒で2,098の可能な解がすべて計算されました。このようにパフォーマンスの大幅な改良ができました。解あたり数時間かかっていたものが、1秒の10分の1未満まで改良されました。これで約400,000倍ほど高速になっています。ちなみに島検出による枝刈りと結合した最初のアルゴリズムは、6,363秒で完了しました。したがって枝刈り最適化によって10,000倍高速になり、充てんアルゴリズムによってさらに40倍高速になっています。これがアルゴリズムの検討とその最適化の試みに費やした時間に見合うものであることは明らかです。

中間結果のキャッシング

パズルを解くアルゴリズムの再設計でプログラムの実行速度は劇的に改良されました。さらに最適化するために、技術的なパフォーマンスの手法を調べる必要があります。Javaプログラムを検討するときの重要な問題として、ガーベッジ・コレクションがあります。プログラム実行時のガーベッジ・コレクターの活動は、`-verbose:gc` コマンド・ライン・スイッチを利用して見ることができます。

```
java -verbose:gc meteor.algorithm.Solver
```

このスイッチを使ってこのプログラムを実行すると、ガーベッジ・コレクターからの多数の出力が表示されます。ソース・コードを詳しく調べると、`Board` クラスの`placePiece()` メソッドにある`ArrayList` 一時オブジェクトのインスタンス化が問題であることが分かります (リスト6 を参照)。この`ArrayList` オブジェクトを、ピースの特定の順列が占有するボード・セルを保持するのに利用しています。このリストを毎回再計算する代わりに、あとで参照できるようにその結果をキャッシュするようにすれば改善されると思われます。

`findOccupiedBoardCells()` メソッドは、パズル・ピースの特定のセルが特定のボード位置に置かれたときに、そのピースが占有するパズル・ボードのセルを判別します。メソッドの結果は3つのパラメーターで決まります。まずパズル・ピースの順列。次に、ピースを操作するときに使用するピースのセル。そして最後はピースを置くボードのセルです。これらの結果をキャッシュするには、テーブルとあらゆる可能なピースの順列とを関連付けます。このテーブルは、その順列についての`findOccupiedBoardCells()` メソッドの結果を、指定のピース・セル・インデックスとボード・セル位置を使って保持します。リスト10はそのようなテーブルを保持する、更新したバージョンの`Piece` クラスを示したものです。

リスト10. findOccupiedBoardCells() メソッドの結果のキャッシング (Caching the results of the findOccupiedBoardCells() method)

```
public class Piece {
  private Piece[] permutations = new Piece[NUMBEROFPERMUTATIONS];
  private ArrayList[][] occupiedBoardCells =
    new ArrayList[Piece.NUMBEROFCELLS][Board.NUMBEROFCELLS];

  private void generatePermutations(Board board) {
    Piece prevPermutation=this;
    for (int i = 0; i < NUMBEROFPERMUTATIONS; i++) {
      // The original nextPermutation() has been renamed
      permutations[i]=
        ((Piece)prevPermutation.clone()).nextPermutation_orig();
      prevPermutation=permutations[i];
    }

    // Calculate occupied board cells for every permutation
    for (int i = 0; i < NUMBEROFPERMUTATIONS; i++) {
      permutations[i].generateOccupiedBoardCells(board);
    }
  }

  private void generateOccupiedBoardCells(Board board) {
    for (int i = 0; i < Piece.NUMBEROFCELLS; i++) {
      for (int j = 0; j < Board.NUMBEROFCELLS; j++) {
        occupiedBoardCells[i][j]=new ArrayList();
        resetProcessed(); // We're going to process the piece
        board.findOccupiedBoardCells(occupiedBoardCells[i][j],
```

```
                                    pieceCells[i],
                                    board.getBoardCell(j));
      }
    }
  }

  public Piece nextPermutation() {
    if (currentPermutation == NUMBEROFPERMUTATIONS)
      currentPermutation = 0;

    // The new implementation of nextPermutation()
    // accesses the cache
    return permutations[currentPermutation++];
  }

  public ArrayList
    getOccupiedBoardCells(int pieceCellIdx, int boardCellIdx) {
    // Access requested data in cache
    return occupiedBoardCells[pieceCellIdx][boardCellIdx];
  }
}
```

`generatePermutations()` メソッドは`Piece` オブジェクトが作成されると起動されます。これはピースのあらゆる順列を計算して、それらの順列についての`findOccupiedBoardCells()` メソッドの可能な結果をすべてキャッシュします。占有されているボード・セルを計算する場合は、もちろんパズル・ボードにアクセスする必要があります。ピースの順列はオリジナルの`Piece` オブジェクトのクローンであることも注意してください。`Piece` のクローンの作成では、そのセルのすべてのディープ・コピーが必要です。

残された唯一のやるべきことは、`Board` クラスの`placePiece()` メソッドからキャッシュにアクセスすることです。これをリスト11に示します。

リスト11. 占有ボード・セル・キャッシュへのアクセス (Accessing the occupied-board-cells cache)

```
public class Board {
  public boolean placePiece(Piece piece, int pieceCellIdx, int boardCellIdx) {
// Get all the boardCells that this piece would occupy
    ArrayList occupiedBoardCells =
      piece.getOccupiedBoardCells(pieceCellIdx, boardCellIdx);
    ...
  }
}
```

もう一度プログラムを実行する

更新したこのバージョンのパズルを解くプログラムのソース・コードは、`meteor.caching` パッケージにあります。`java meteor.caching.Solver` を実行すると、パフォーマンスが再びかなり向上したのが分かります。私たちのテスト・マシンでは、すべての解が25秒で求められました。キャッシングによって6倍高速になっています。`-verbose:gc` スイッチを使用すると、ガーベッジ・コレクションは問題ではなくなっていることも分かります。

キャッシュの実装に取り入れた追加のコードは、明らかにプログラムを複雑にしています。これは、中間結果をストアすることによって計算時間を減らそうとするパフォーマンス手法の典型的なマイナス面です。しかしこの場合のパフォーマンスの向上は、追加したコードの複雑さを補ってなお余りあるように思えます。

プログラミングの最適化

このパズルを解くプログラムの考えられる最適化プロセスの最後のステップは、低水準のJavaコード最適化イディオムを使用することです。このアプリケーションでは文字列を一切操作していないため、よく知られている`StringBuffer`イディオムの適用は役に立ちません。getterとsetterを直接メンバー・アクセスに置き換えることによって、それらのgetterとsetterのメソッド呼び出しのオーバーヘッドの回避を試みることもできるでしょう。しかしこれはここでのコードの品質を低下させるのは明らかです。テストでは、これによってはほとんど高速になりませんでした。同じことが`final`メソッドの利用についても言えます。ここでのメソッドを`final`として宣言することにより、動的結合を回避してJava仮想マシンでもっと効果的な静的結合を利用することができます。しかし残念ながら、これも注目に値する高速化は得られませんでした。またJavaコンパイラーの`-O`最適化スイッチを利用しても、実際のパフォーマンスの向上は得られません。

`prune()`メソッドの実装を改良すると、わずかですがまだ実行を高速化できます。リスト7のコードは、ボード・セルがすでに処理済の場合や空ではない場合でも、常に再帰的`getIslandSize()`メソッドの呼び出しを行っています。`getIslandSize()`を呼び出す前にそれらのチェックを積極的に行えば、約10パーセント向上します。

ここでの検討で明らかなように、低水準の最適化によるパフォーマンスの向上は非常に小さなものです。これらの最適化手法のいくつかはコードの品質を低下させるという現実を考えると、低水準の最適化を利用することの魅力はなくなります。

結論

パズルを解くプログラムの実装の改良の試みはすべて確実に成果を挙げました。表1は、作成した異なるバージョンとそれらの実行時間を要約したものです。全体的な結果としては、およそ2,000,000倍という驚くべき高速化です。

表1. 実行時間の比較 (Comparing execution times)

Version	Time (seconds)
`meteor.initial`	~ 60,422,400 (about 2 years)
`meteor.algorithm`	157
`meteor.caching`	25

この最適化がどんなに感動的であったとしても、重要なことはこの実験から何を学べるかということです。ここで使用したさまざまな最適化手法には、それぞれ利点と欠点があります。それらを1つの最適化プロセスに結合すると、それらの使用方法が明確になり、アプリケーションが動作しないということを未然に防ぐことになります。

- ここで使用したアルゴリズムの改良のような高水準の最適化手法には大きな可能性があります。パフォーマンスが重要なコードの最適化を行う必要があるときは、まずそのコードが実装しているプロセスを分析してみてください。プロセスをビジュアル化することは、そのプロセスの理解を深める上で優れた方法です。また、別の角度から問題に取り組んでみてください。もともと考えていたものよりもはるかによいソリューションが見つかる可能性があります。この種の最適化で明らかに難しいのは、一般化するのが困難であるということです。

すべてのアルゴリズムは特定のアプリケーションに固有のものであるため、提供できる汎用ガイドラインはあまりありません。プログラマーの創造性次第です。
- 適切な良いソリューションに至ったと確信できたら、次は、技術的なパフォーマンス改良手法を適用します。基本的な着想は時間がかかること をデータの複雑さ に置き換えることです。オブジェクト・キャッシュはそのような手法の中で最も代表的なものの1つです。Java プログラムでは、オブジェクト・キャッシュを利用すると高価につくオブジェクトの作成とガーベッジ・コレクションのオーバーヘッドを回避できるため、オブジェクト・キャッシュは特に有用です。ただし、この種のシステムでは余分なインフラストラクチャー・コードをプログラムに追加することになるため、その導入が早すぎることがないように気を付けてください。コードが複雑になればなるほど、最適化するのが困難になります。
- 最後に、低水準のプログラミング最適化を適用します。Javaプログラマーの多くはこの種の手法をよく知っています。しかしそれらの利点はたいていは実世界プログラムにおいては限定されます。それらを可能なところに適用するのはよいのですが、最適化の試みの全体の焦点をその種のイディオムに合わせないでください。それらはむしろ、既知のパフォーマンスの罠を避けるために利用するプログラミング・ツール・セットの一部であるべきです。

パズルを解くプログラムで異なる最適化手法を結合して達成したパフォーマンスの目覚ましい向上は、すべてのJavaプログラマーが自分自身のコードを見てそれをどのようにしたら最適化できるかを調べる動機付けになるはずです。

謝辞

著者はこの発表文書のレビューにおけるBieke Meeussenの支援に感謝します。また、Pieter BekaertならびにKris Cardinaelsの貢献に感謝します。Pieterは充てんパズル解きアルゴリズムを考え出し、Krisはパズルのビジュアル化コードの部分を開発しました。

著者について

Maarten De Cock

Maarten De Cock氏は、クリーンで高速のJavaコードを書くことに集中するプログラマーです。氏はベルギーのKatholieke Hogeschool Leuvenを卒業後、1999年にJavaプログラム言語を使い始めました。氏はそれ以来いくつかのe-commerceプロジェクトに参加し、定期的にJavaプログラミング・コースも教えています。氏は現在ASQdotCOM のコンサルタントとして活躍しています。

Erwin Vervaet

Erwin Vervaet氏は、現在のITの概念とツールの応用に強い関心を持つソフトウェア・エンジニアです。氏はJavaプログラム言語を1996年から使用しており、ベルギーのKatholieke Universiteit Leuvenからコンピューター・サイエンスの修士号を授与されています。氏はITの研究、e-commerceプロジェクト、オープン・ソース・イニシアチブ、および産業用ソフトウェア・システムにかかわってきました。Erwinは独立コンサルタントとして、Javaプログラム言語を使用したオブジェクト指向のビジネス情報システムを構築しています。