Java 言語による Unicode サロゲート・プログラミング

コーディングの選択肢とパフォーマンスに関する考慮事項

Masahiko Maedera (maedera@jp.ibm.com)
Software engineer
I.B.M.

2010年 8月 24日

Java™ 言語ではバージョン 1.5 以降、16 ビットの `char` データ型単独では表現できない Unicode 補助文字に対応するための API を提供しています。この記事ではこれらの API の機能を取り上げ、正しい使用方法を説明するとともに、それぞれの API を使用して処理を行った場合のパフォーマンスを評価します。
※この記事は2010年3月に公開した記事「Java による Unicode サロゲートプログラミング」を、英語版として再編集したものの翻訳版です。

Java の初期のバージョンでは Unicode 文字を 16 ビットの `char` データ型を使用して表現していました。当時、Unicode 文字の数は 65,535 個 (0xFFFF) に満たず、すべての文字を 16 ビットで表現できたことから、この設計は妥当なものでした。けれどもその後、Unicode 文字の数は 1,114,111 個 (0x10FFFF) にまで増え、Unicode バージョン 3.1 のすべての Unicode 文字を表現するには 16 ビットの値では足りなくなってきました。この問題に対処するために UTF-32 エンコード方式に導入されたのが、コード・ポイントと呼ばれる 32 ビットの値です。

ところが効率的にメモリーを使用するには 32 ビットの値よりも 16 ビットの値のほうが望ましいことから、16 ビットの値を引き続き使用できるようにする新しい設計が Unicode に導入されました。UTF-16 エンコード方式に採り入れられたこの設計では、16 ビットの上位サロゲートに 1,024 個の値を割り当て、16 ビットの下位サロゲートに別の 1,024 個の値を割り当てます。そして上位サロゲートの後に下位サロゲートを組み合わせたサロゲート・ペアを使用して、65,536 個 (0x10000) から 1,114,111 個 (0x10FFFF) までの 1,048,576 (1,024×1,024) 個 (0x100000) の値を表現します。

Java 1.5 では、(既存のプログラムとの互換性のため) `char` 型で UTF-16 の値を表現するという振る舞いを維持し、UTF-32 の値を表現するためにコード・ポイントの概念を実装しました。この拡張 (JSR 204: Unicode Supplementary Character Support に準拠して実装) により、Unicode コード・ポイントの正確な値や変換アルゴリズムを覚える必要はなくなりましたが、サロゲート API の適切な使用方法を理解することは重要です。

東アジアの国々と地域では、近年、ユーザーのニーズに応えて文字セット標準に含まれる文字の数を増やしてきました。例えば、中国の国家標準化管理委員会による GB 18030 や、日本工業規格の JIS X 0213 もそのような標準の例です。その結果、これらの標準への準拠を目指すプログラムでは Unicode サロゲート・ペアへの対応がますます必要になってきています。この記事では、`char` 型で表現される文字のみに対応したソフトウェアを、サロゲート・ペア対応の新しいバージョンに設計しなおそうと計画している読者の皆さんのために、関連する Java API とコーディングの選択肢を説明します。また、サロゲート・ペアに対応することによる処理時間への影響についても評価します。

順次アクセス

順次アクセスは、Java 言語で文字列を処理する際の基本操作の 1 つです。順次アクセスでは、入力文字列の先頭の文字から末尾の文字まで、場合によっては末尾の文字から先頭の文字まで、各文字に順番にアクセスしていきます。このセクションでは、順次アクセスを使用して文字列から 32 ビットのコード・ポイントの配列を作成する例として 7 通りの方法を説明し、それぞれの処理時間を評価します。

例 1-1: ベンチマーク (サロゲート・ペア非対応)

リスト 1 のコードでは、サロゲート・ペアについては考慮せずに、16 ビットの `char` 型の値を 32 ビットのコード・ポイントの値に直接割り当てています。

リスト 1. サロゲート・ペアに対応しない場合

```
int[] toCodePointArray(String str) { // Example 1-1
    int len = str.length();          // the length of str
    int[] acp = new int[len];        // an array of code points

    for (int i = 0, j = 0; i < len; i++) {
        acp[j++] = str.charAt(i);
    }
    return acp;
}
```

これはサロゲート・ペアに対応しない例ですが、以降の順次アクセスの例と処理時間を比較するためのベンチマークとなります。

例 1-2: `isSurrogatePair()` を使用する場合

リスト 2 のコードは、`isSurrogatePair()` を使用してサロゲート・ペアの合計数をカウントし、カウントした値を保存するのに十分なメモリーをコード・ポイントの配列に割り当てます。次に、順次アクセスのループに入り、`isHighSurrogate()` と `isLowSurrogate()` を使用してサロゲート文字のそれぞれが上位サロゲートであるか、下位サロゲートであるかを判別します。上位サロゲートの後に下位サロゲートが続いていることを検出すると、`toCodePoint()` を使用してサロゲート・ペアをコード・ポイント値に変換し、現在のインデックスに 2 を加算します。サロゲート・ペアでない場合は、`char` 型の値をコード・ポイントの値に直接割り当て、現在のインデックスに 1 を加算します。この例の処理時間は、例 1-1 の 1.38 倍です。

リスト 2. 問題点の多い方法でのサロゲート・ペアのサポート

```
int[] toCodePointArray(String str) { // Example 1-2
    int len = str.length();          // the length of str
    int[] acp;                       // an array of code points
    int surrogatePairCount = 0;      // the count of surrogate pairs
```

```
    for (int i = 1; i < len; i++) {
        if (Character.isSurrogatePair(str.charAt(i - 1), str.charAt(i))) {
            surrogatePairCount++;
            i++;
        }
    }
    acp = new int[len - surrogatePairCount];
    for (int i = 0, j = 0; i < len; i++) {
        char ch0 = str.charAt(i);         // the current char
        if (Character.isHighSurrogate(ch0) && i + 1 < len) {
            char ch1 = str.charAt(i + 1); // the next char
            if (Character.isLowSurrogate(ch1)) {
                acp[j++] = Character.toCodePoint(ch0, ch1);
                i++;
                continue;
            }
        }
        acp[j++] = ch0;
    }
    return acp;
}
```

リスト 2 のネイティブなやり方でリスト 1 のソフトウェアを更新する方法には問題があります。この方法は面倒な上に、広範囲にわたる変更が必要なことから、作成されたソフトウェアは不安定なものになりがちで、後から変更するのも困難です。具体的に言うと、以下の点が問題になります。

- 十分なメモリーを割り当てるためにコード・ポイントの数をカウントしなければならないこと
- 文字列内の指定されたインデックスに対応する正しいコード・ポイントの値を取得するのが難しいこと
- 現在のインデックスを次の処理ステップのために正しく移動するのが難しいこと

次の例は、以上の問題点を改善したアルゴリズムです。

例 1-3: 基本サポート

例 1.2 に伴う 3 つの問題に対処するために Java 1.5 で提供するようになったのが、`codePointCount()`、`codePointAt()`、`offsetByCodePoints()` という 3 つのメソッドです。リスト 3 では、これらのメソッドを使用してアルゴリズムを理解しやすいように改善しています。

リスト 3. 基本サポート

```
int[] toCodePointArray(String str) { // Example 1-3
    int len = str.length();          // the length of str
    int[] acp = new int[str.codePointCount(0, len)];

    for (int i = 0, j = 0; i < len; i = str.offsetByCodePoints(i, 1)) {
        acp[j++] = str.codePointAt(i);
    }
    return acp;
}
```

けれどもリスト 3 の処理時間は、リスト 1 の 2.80 倍となります。

例 1-4: `codePointBefore()` を使用する場合

`offsetByCodePoints()` は、2 番目の引数として負の数を受け取った場合、文字列の先頭方向への絶対オフセットを計算することができます。さらに `codePointBefore()` は、指定されたインデックスの直前にコード・ポイントの値を返すことができます。リスト 4 は、この 2 つのメソッドを使用して文字列を末尾から先頭に向かってトラバースする例です。

リスト 4. `codePointBefore()` による基本サポート

```
int[] toCodePointArray(String str) { // Example 1-4
    int len = str.length();          // the length of str
    int[] acp = new int[str.codePointCount(0, len)];
    int j = acp.length;              // an index for acp

    for (int i = len; i > 0; i = str.offsetByCodePoints(i, -1)) {
        acp[--j] = str.codePointBefore(i);
    }
    return acp;
}
```

この例の処理時間は、例 1-1 の 2.72 倍で、例 1-3 よりもわずかに短縮されています。一般に JVM のコード・サイズは、値の比較を行う際にゼロ以外の値に対してではなく、ゼロに対して行うと小さくなります。このことから、パフォーマンスが向上することもありますが、わずかなパフォーマンスの向上のために可読性を犠牲にする価値はないかもしれません。

例 1-5: charCount() を使用する場合

サロゲート・ペアの基本サポートを提供する例 1-3 と例 1-4 は、いずれも一時変数が必要とならないことから、堅牢なコーディング手法となります。処理時間を短縮するには、`offsetByCodePoints()` の代わりに `charCount()` を使用すると効果的ですが、その場合、コード・ポイントの値を保持するための一時変数が必要になってきます (リスト 5 を参照)。

リスト 5. `charCount()` を使用して最適化したサポート

```
int[] toCodePointArray(String str) { // Example 1-5
    int len = str.length();          // the length of str
    int[] acp = new int[str.codePointCount(0, len)];
    int j = 0;                       // an index for acp

    for (int i = 0, cp; i < len; i += Character.charCount(cp)) {
        cp = str.codePointAt(i);
        acp[j++] = cp;
    }
    return acp;
}
```

リスト 5 の場合の処理時間は、例 1-1 の 1.68 倍にまで減ります。

例 1-6: `char` 配列にアクセスする場合

リスト 6 は、例 1.5 に示した最適化を使用する一方で、`char` 型の配列に直接アクセスします。

リスト 6. char 配列を使用して最適化したサポート

```
int[] toCodePointArray(String str) { // Example 1-6
    char[] ach = str.toCharArray();  // a char array copied from str
    int len = ach.length;            // the length of ach
    int[] acp = new int[Character.codePointCount(ach, 0, len)];
    int j = 0;                       // an index for acp

    for (int i = 0, cp; i < len; i += Character.charCount(cp)) {
        cp = Character.codePointAt(ach, i);
        acp[j++] = cp;
    }
    return acp;
}
```

char 配列は toCharArray() を使用して文字列からコピーされます。メソッドを介した間接アクセスよりも、配列に直接アクセスしたほうが速いことから、パフォーマンスは改善されます。処理時間は例 1-1 の 1.51 倍です。その一方で、toCharArray() を呼び出すと、新しい配列を作成してデータをそこにコピーするためのオーバーヘッドが生じます。さらに、String クラスが提供する便利なメソッドも使用することができません。それでもやはり、大量のデータを処理する場合にはこのアルゴリズムが役立ちます。

例 1-7: オブジェクト指向のアルゴリズム

この例に示すオブジェクト指向のアルゴリズムは CharBuffer クラスを使用します (リスト 7 を参照)。

リスト 7. CharSequence を使用したオブジェクト指向の手法

```
int[] toCodePointArray(String str) {        // Example 1-7
    CharBuffer cBuf = CharBuffer.wrap(str); // Buffer to wrap str
    IntBuffer iBuf = IntBuffer.allocate(    // Buffer to store code points
            Character.codePointCount(cBuf, 0, cBuf.capacity()));

    while (cBuf.remaining() > 0) {
        int cp = Character.codePointAt(cBuf, 0); // the current code point
        iBuf.put(cp);
        cBuf.position(cBuf.position() + Character.charCount(cp));
    }
    return iBuf.array();
}
```

これまでの例とは異なり、リスト 7 では順次アクセスを行うために、インデックスを使用して現在の位置を保持する必要がありません。インデックスの代わりに、CharBuffer が現在の位置を内部で追跡するためです。Character クラスが提供する静的メソッド codePointCount() および codePointAt() は、CharSequence インターフェースを介して CharBuffer を扱うことができます。CharBuffer は常に現在の位置を CharSequence の先頭に設定するため、codePointAt() が呼び出されるときの 2 番目の引数は常に 0 に設定されます。この例での処理時間は、例 1-1 の 2.15 倍です。

処理時間の比較

順次アクセスの例の処理時間を測定するためのテストで使用したサンプル文字列には、10,000 のサロゲート・ペアと 10,000 の非サロゲートが含まれます。この文字列から、コード・ポイントの配列を 10,000 回作成しました。テスト環境は、以下の構成になっています。

- OS: Microsoft Windows® XP Professional SP2
- Java: IBM Java 1.5 SR7
- CPU: Intel® Core 2 Duo CPU T8300 @ 2.40GHz
- メモリー: 2.97GB RAM

表 1 に、例 1-1 から例 1-7 までの絶対処理時間と相対処理時間、および関連 API を記載します。

表 1. 順次アクセスの例での処理時間および API

例	説明	処理 時間 (ms)	例 1-1 との比	API
1-1	サロゲート・ペア非対応	2031	1.00	
1-2	問題点の多い方法でのサロゲート・ペアのサポート	2797	1.38	`Character` クラス: • `static boolean isHighSurrogate(char ch)` • `static boolean isLowSurrogate(char ch)` • `static boolean isSurrogatePair(char high, char low)` • `static int toCodePoint(char high, char low)`
1-3	基本サポート	5687	2.80	`String` クラス: • `int codePointAt(int index)` • `int codePointCount(int begin, int end)` • `int offsetByCodePoints(int index, int cpOffset)`
1-4	`codePointBefore()`による基本サポート	5516	2.72	`String` クラス: • `int codePointBefore(int index)`
1-5	`charCount()`を使用して最適化したサポート	3406	1.68	`Character` クラス: • `static int charCount(int cp)`
1-6	`char` 配列を使用して最適化したサポート	3062	1.51	`Character` クラス: • `static int codePointAt(char[] ach, int index)` • `static int codePointCount(char[] ach, int offset, int count)`
1-7	`CharSequence`によるオブジェクト指向の手法	4360	2.15	`Character` クラス:

				• static int codePointAt(CharSequenc seq, int index) • static int codePointCount(CharSequ seq, int begin, int end)

ランダム・アクセス

ランダム・アクセスとは、文字列の任意の位置に直接アクセスすることです。文字列にアクセスするときのインデックスは、16 ビット `char` 型を単位とした値となりますが、32 ビットのコード・ポイントを使用している文字列には、32 ビットのコード・ポイントを単位としたインデックスを使ってアクセスすることはできません。その場合には `offsetByCodePoints()` を使用して、コード・ポイントのインデックスを `char` 型のインデックスに変換する必要があります。アルゴリズムが上手く設計されていないと、この変換がパフォーマンス劣化の原因になってしまいます。なぜなら、`offsetByCodePoints()` では、1 番目の引数で指定されたインデックスから 2 番目の引数で指定されたコード・ポイント分だけ離れた位置のインデックスが、文字列内に収まっているかどうかを必ず計算するからです。このセクションでは、ある小さな単位を使って長い文字列を短い文字列に分割する 3 つの例を比較します。

例 2-1: ベンチマーク (サロゲート・ペア非対応)

リスト 8 に、ある大きさの単位 (width) で文字列を分割する方法を示します。サロゲート・ペアに対応しないこの例は、以降の例を比較するためのベンチマークです。

リスト 8. サロゲート・ペアに対応しない場合

```
String[] sliceString(String str, int width) { // Example 2-1
    // It must be that "str != null && width > 0".
    List<String> slices = new ArrayList<String>();
    int len = str.length();       // (1) the length of str
    int sliceLimit = len - width; // (2) Do not slice beyond here.
    int pos = 0;                  // the current position per char type

    while (pos < sliceLimit) {
        int begin = pos;                      // (3)
        int end   = pos + width;              // (4)
        slices.add(str.substring(begin, end));
        pos += width;                         // (5)
    }
    slices.add(str.substring(pos));           // (6)
    return slices.toArray(new String[slices.size()]); }
```

`sliceLimit` 変数は、分割位置の上限を保持します。これは、文字列の残りの部分の長さが現在の単位 width で分割するには足りない場合に `IndexOutOfBoundsException` のインスタンスがスローされないようにするためです。このアルゴリズムは、現在の位置が `sliceLimit` を超えた時点で `while` ループから抜け、最後に残った短い文字列を処理します。

例 2-2: コード・ポイントのインデックスを使用する場合

リスト 9 に、コード・ポイントのインデックスを使用して文字列にランダムにアクセスする方法を示します。

リスト 9. パフォーマンスの低い処理

```
String[] sliceString(String str, int width) { // Example 2-2
    // It must be that "str != null && width > 0".
    List<String> slices = new ArrayList<String>();
    int len = str.codePointCount(0, str.length()); // (1) code point count [Modified]
    int sliceLimit = len - width; // (2) Do not slice beyond here.
    int pos = 0;                  // the current position per code point

    while (pos < sliceLimit) {
        int begin = str.offsetByCodePoints(0, pos);            // (3) [Modified]
        int end   = str.offsetByCodePoints(0, pos + width);    // (4) [Modified]
        slices.add(str.substring(begin, end));
        pos += width;                                          // (5)
    }
    slices.add(str.substring(str.offsetByCodePoints(0, pos))); // (6) [Modified]
    return slices.toArray(new String[slices.size()]); }
```

リスト 9 では、リスト 8 の行がいくつか変更されています。まず、行 (1) では `length()` を `codePointCount()` に置き換えています。さらに行 (3)、(4)、(6) のそれぞれで、`offsetByCodePoints()` を使用して `char` 型のインデックスをコード・ポイントのインデックスに置き換えています。

このアルゴリズムの基本的な流れは、例 2-1 とほとんど変わらないように見えますが、処理時間は例 2-1 と比べ、文字列の長さに比例して長くなっています。その理由は、この例 2-2 での `offsetByCodePoints()` は、文字列の先頭から 2 番目の引数で指定されたコード・ポイント分だけ離れた位置のインデックスが、文字列内に収まっているかどうかを必ず計算するからです。

例 2-3: 処理時間の短縮

例 2-2/ のパフォーマンス問題は、リスト 10 に示す手法によって回避することができます。

リスト 10. 改善されたパフォーマンス

```
String[] sliceString(String str, int width) { // Example 2-3
    // It must be that "str != null && width > 0".
    List<String> slices = new ArrayList<String>();
    int len = str.length(); // (1) the length of str
    int sliceLimit          // (2) Do not slice beyond here. [Modified]
            = (len >= width * 2 || str.codePointCount(0, len) > width)
            ? str.offsetByCodePoints(len, -width) : 0;
    int pos = 0;            // the current position per char type

    while (pos < sliceLimit) {
        int begin = pos;                                  // (3)
        int end   = str.offsetByCodePoints(pos, width); // (4) [Modified]
        slices.add(str.substring(begin, end));
        pos = end;                                        // (5) [Modified]
    }
    slices.add(str.substring(pos));                       // (6)
    return slices.toArray(new String[slices.size()]); }
```

まず行 (2) では、`len-width` という式 (リスト 9) が `offsetByCodePoints(len,-width)` に置き換えられています。けれどもこの場合、`width` の値がコード・ポイントの数より大きいと、`IndexOutOfBoundsException` のインスタンスがスローされることになります。`try/catch` 例外ハンドラーを使用した節を使うという手もありますが、例外が発生しないようにするには、境界条件を考慮しなければなりません。`len>width*2` という式が true であれ

ば、`offsetByCodePoints()` を安全に呼び出せます。これは、すべてのコード・ポイントがサロゲート・ペアに変換されたとしても、コード・ポイントの数が `width` の値を上回るためです。あるいは、`codePointCount(0,len)>width` という式が true の場合でも、`offsetByCodePoints()` を安全に呼び出せます。以上の場合に当てはまらなければ、`sliceLimit` は `0` に設定されます。

リスト 9 の `pos + width` という式は、`while` ループ内の行 (4) で `offsetByCodePoints(pos,width)` に置き換える必要があります。これにより、最初の引数で現在の位置を指定し、2 番目の引数で `width` の値を指定しているため、行わなければならない計算の量は `width` の値の範囲に収まるからです。次に、行 (5) で `pos+=width` という式を `pos=end` という式に置き換えます。こうすることにより、同じインデックスを計算するために `offsetByCodePoints()` が 2 回呼び出されることがなくなります。このソース・コードは、処理時間をさらに短縮するための修正を加えることができます。

処理時間の比較

図 1 と図 2 に、例 2-1、2-2、2-3 の処理時間を示します。サンプル文字列には、サロゲート・ペアと非サロゲートがそれぞれ同じ数だけ含まれています。このサンプル文字列を、文字列の長さと `width` の値を変更しながら 10,000 回分割しました。

図 1. 分割した短い文字列の単位 (width) が一定になるようにした場合

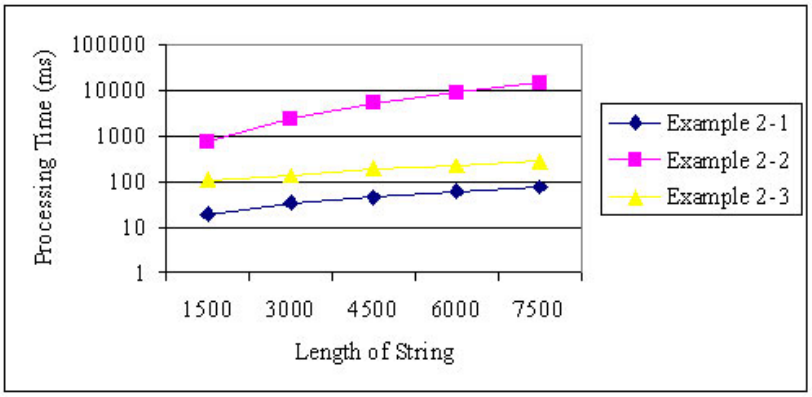
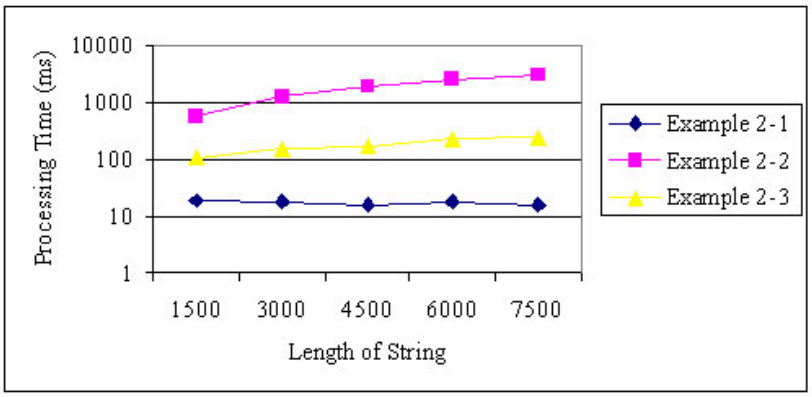


図 2. 分割した短い文字列の個数が一定になるようにした場合



例 2-1 と例 2-3 では、長さに比例して処理時間が長くなる一方、例 2-2 では長さの二乗に比例して処理時間が長くなります。短い文字列の個数は変えずに文字列の長さと `width` の値を増やしてい

くと、例 2-1 の処理時間に変化はありませんが、例 2-2 と例 2-3 では `width` の値に比例して処理時間が長くなります。

情報取得 API

サロゲートを処理するための情報取得 API の大半には、同じ名前を持つ 2 つのタイプのメソッドがあります。一方のタイプのメソッドは引数を 16 ビットの`char`型として受け取り、もう一方は 32 ビットのコード・ポイントとして受け取ります。表 2 に、各 API の戻り値を記載します。この表の 3 列目は U+53F1 の場合の戻り値、4 列目は U+20B9F の場合の戻り値です。そして最後の列に、U+20B9F が U+D842 の後に U+DF9F を組み合わせたサロゲート・ペアに変換された場合の U+D842 の値 (上位サロゲート) を記載します。プログラムがサロゲート・ペアの処理に対応していない場合、値が U+20B9F ではなく U+D842 であると、予期せぬ結果になります (該当する結果は、表 2 に太字のイタリック体で示されています)。

表 2. 情報を取得するサロゲート API

クラス	メソッド/コンストラクター	値 (U+53F1)	値 (U+20B9F)	値 (U+D842)
		叱	叱	
`Character`	`static byte getDirectionality(int cp)`	`0`	`0`	`0`
	`static int getNumericValue(int cp)`	`-1`	`-1`	`-1`
	`static int getType(int cp)`	`5`	`5`	***`19`***
	`static boolean isDefined(int cp)`	`true`	`true`	`true`
	`static boolean isDigit(int cp)`	`false`	`false`	`false`
	`static boolean isISOControl(int cp)`	`false`	`false`	`false`
	`static boolean isIdentifierIgnorable(int cp)`	`false`	`false`	`false`
	`static boolean isJavaIdentifierPart(int cp)`	`true`	`true`	***`false`***
	`static boolean isJavaIdentifierStart(int cp)`	`true`	`true`	***`false`***
	`static boolean isLetter(int cp)`	`true`	`true`	***`false`***
	`static boolean isLetterOrDigit(int cp)`	`true`	`true`	***`false`***
	`static boolean isLowerCase(int cp)`	`false`	`false`	`false`

	`static boolean isMirrored(int cp)`	`false`	`false`	`false`
	`static boolean isSpaceChar(int cp)`	`false`	`false`	`false`
	`static boolean isSupplementaryCodePoint(int cp)`	`false`	***`true`***	`false`
	`static boolean isTitleCase(int cp)`	`false`	`false`	`false`
	`static boolean isUnicodeIdentifierPart(int cp)`	`true`	`true`	***`false`***
	`static boolean isUnicodeIdentifierStart(int cp)`	`true`	`true`	***`false`***
	`static boolean isUpperCase(int cp)`	`false`	`false`	`false`
	`static boolean isValidCodePoint(int cp)`	`true`	`true`	`true`
	`static boolean isWhitespace(int cp)`	`false`	`false`	`false`
	`static int toLowerCase(int cp)`	(不変)		
	`static int toTitleCase(int cp)`	(不変)		
	`static int toUpperCase(int cp)`	(不変)		
`Character.UnicodeBlock`	`Character.UnicodeBlock of(int cp)`	`CJK_UNIFIED_IDEOGRAPHS`	`CJK_UNIFIED_IDEOGRAPHS`	`HIGH_SURROGATES`
`Font`	`boolean canDisplay(int cp)`	(`Font` インスタンスに依存)		
`FontMetrics`	`int charWidth(int cp)`	(`FontMetrics` インスタンスに依存)		
`String`	`int indexOf(int cp)`	(`String` インスタンスに依存)		
	`int lastIndexOf(int cp)`	(`String` インスタンスに依存)		

その他の API

このセクションでは、これまでのセクションで取り上げていないサロゲート・ペア関連の API を紹介します。表 3 に、これらの残りの API をすべて記載します。サロゲート・ペア API はすべて、表 1、2、または 3 のいずれかに記載されています。

表 3. その他のサロゲート API

クラス	メソッド/コンストラクター
`Character`	`static int codePointAt(char[] ach, int index, int limit)`
	`static int codePointBefore(char[] ach, int index)`

	`static int codePointBefore(char[] ach, int index, int start)`
	`static int codePointBefore(CharSequence seq, int index)`
	`static int digit(int cp, int radix)`
	`static int offsetByCodePoints(char[] ach, int start, int count, int index, int cpOffset)`
	`static int offsetByCodePoints(CharSequence seq, int index, int cpOffset)`
	`static char[] toChars(int cp)`
	`static int toChars(int cp, char[] dst, int dstIndex)`
`String`	`String(int[] acp, int offset, int count)`
	`int indexOf(int cp, int fromIndex)`
	`int lastIndexOf(int cp, int fromIndex)`
`StringBuffer`	`StringBuffer appendCodePoint(int cp)`
	`int codePointAt(int index)`
	`int codePointBefore(int index)`
	`int codePointCount(int beginIndex, int endIndex)`
	`int offsetByCodePoints(int index, int cpOffset)`
`StringBuilder`	`StringBuilder appendCodePoint(int cp)`
	`int codePointAt(int index)`
	`int codePointBefore(int index)`
	`int codePointCount(int beginIndex, int endIndex)`
	`int offsetByCodePoints(int index, int cpOffset)`
`IllegalFormat CodePointException`	`IllegalFormatCodePointException(int cp)`
	`int getCodePoint()`

リスト 11 に、コード・ポイントから文字列を作成する 5 つの方法を示します。テストに使用したコード・ポイントは U+53F1 および U+20B9F で、ある文字列で百億回繰り返しました。リスト 11 に処理時間をコメントとして記載します。

リスト 11. コード・ポイントから文字列を作成する 5 つの方法

```
int cp = 0x20b9f; // CJK Ideograph Extension B
String str1 = new String(new int[]{cp}, 0, 1);    // processing time: 206ms
String str2 = new String(Character.toChars(cp));                  //  187ms
String str3 = String.valueOf(Character.toChars(cp));              //  195ms
String str4 = new StringBuilder().appendCodePoint(cp).toString(); //  269ms
String str5 = String.format("%c", cp);                            // 3781ms
```

`str1`、`str2`、`str3`、および `str4` を作成する場合の処理時間に顕著な違いはありません。それとは対照的に、`str5` を作成するには遥かに長い時間がかかります。その理由は、この文字列で使用する `String.format()` は、ロケールおよびフォーマットの情報に基づく柔軟な出力をサポートするためです。したがって `str5` の方法は、プログラムが最後にテキストを出力する段階になるまで使用しないでください。

まとめ

Unicode の新しいバージョンが登場するごとに、サロゲート・ペアで表現された文字が新しく定義されています。このように新しい文字が定義されているのは、東アジアの文字セット標準だけではありません。例えば、携帯電話での絵文字 (顔文字) にしても、さまざまな古代文字にしても、対応が求められています。この記事で学んだ手法とパフォーマンス分析を参考に、Java アプリケーションでこれらの文字に効率的に対応してください。

著者について

Masahiko Maedera

前寺 正彦は、日本アイ・ビー・エム 大和ソフトウェア研究所 (YSL) の Globalization Center of Competency に所属しています。Lotus Notes/Domino の文字列ライブラリーの設計、実装を担当しています。