ロープ: 理論と実践

ストリングの処理に Ropes for Java を使用する理由、そして使用に適した条件

Amin Ahmad
Independent Consultant
Ahmadsoft, LLC

2008年 2月 12日

Java™ 言語のデフォルトの `String` クラスと `StringBuilder` クラスは、大量のストリング・データを処理するシステムとなると十分に対応しきれません。この場合、これらのデフォルト・クラスより優れた手段となるのがロープ (rope) というデータ構造です。この記事では Java プラットフォームのロープ実装、Ropes for Java について紹介し、パフォーマンス上の問題を探るとともに、このライブラリーの効果的な使い方をアドバイスします。

ロープ・データ構造は、不変 (immutable) の文字シーケンスを表します。これは Java の `String` とよく似てはいるものの、ロープのミューテーション (状態を変化させること) は `String` やその仲間の可変 (mutable) `StringBuffer` および `StringBuilder` とは違って極めて効率的なため、特にマルチスレッド環境をはじめ、大量のストリングの処理をするアプリケーションに最適です。

この記事ではロープ・データ構造の概要について簡単に要約した後、Java プラットフォームへのロープ実装である、Ropes for Java について紹介します。続いて Java プラットフォームで `String` と `StringBuffer`、そして Ropes for Java の`Rope` クラスのベンチマークを通して特殊なパフォーマンス上の問題を探り、最後にアプリケーションでロープを使うべきとき (そして使うべきでないとき) について説明して記事を締めくくります。

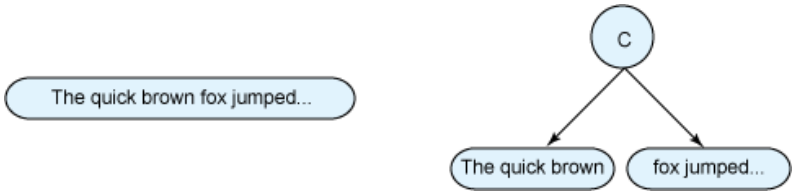
ロープ: 概要

ロープは不変の文字シーケンスを表し、その長さは単純にシーケンスに含まれる文字数となります。ほとんどのストリング表現はすべての文字が隣接してメモリーに保存されますが、ロープの決定的特徴は、この制約が取り払われるという点です。ロープでは、ストリングのフラグメントを隣接させずに常駐させることができ、連結ノードを使ってそのフラグメントを結合します。このような設計により、Java の `String` と比べて連結に要する時間は短くなり、その差は連結操作が増えるほど広がります。つまり、`String` による連結では、結合させるストリングを新しいロケーションにコピーしなければならないため、連結に要する時間は O(n) となりますが、ロープによる連結では、単純に新しい連結ノードが作成されるだけなので、連結に要する時間は O(1) となるの

です (ビッグオー記法 (O( )) に馴染みがない場合は、「参考文献」に記載した説明資料へのリンクにアクセスしてください)。

図 1 にストリングを表す場合の 2 種類の方法を示します。左側のストリングでは、文字は隣接したメモリー・ロケーションに配置されます。これが、Java でストリングを表す場合の方法です。一方、右側のストリングでは、バラバラになっているストリングを連結ノードによって結合して表します。

図 1. ストリングを表す 2 種類の方法



ロープ実装では、サブストリング・ノードを導入して大規模なサブストリングの処理を委ねることもよくあります。サブストリング・ノードを使用すると、長さ n のサブストリングを抽出する時間が O(n) から O(log n) に、さらには O(1) にまで短縮されることもよくあります。Java の `String` は不変であることから、このクラスにも一定時間で処理することが可能なサブストリングがありますが、`StringBuilder` にはそのようなサブストリングがないことがほとんどであるということも、注意しておくべき重要なことです。

フラット・ロープ (連結ノードもサブストリング・ノードもないロープのこと) の深さは 1 です。連結およびサブストリング・ロープの深さは、そのロープが囲む最も深いノードの深さに 1 を足した数となります。

ロープには、メモリー・ロケーションに隣接して配置された文字ストリングにはないオーバーヘッドが 2 つあります。まず 1 つは、サブストリング・ノードおよび連結ノードの上部構造を巡回しないと指定した文字に到達できないという点です。さらに、巡回時間を最小限にするためには、この上部ツリー構造をできるだけバランスの取れた状態に維持しなければなりません。つまり、読み取りパフォーマンスを良好に保つために時々ロープのバランスを取り直す必要があるということです。ロープのバランスが取れているとしても、特定の位置にある文字を取得するには O(log n) 時間かかります。ここで、n はロープを構成するサブストリング・ノードと連結ノードの合計を表します (便宜上、n をロープの長さに置き換えることもできます。ロープの長さはロープに含まれるサブストリングおよび連結ノードの合計よりも必ず大きくなるからです)。

幸い、ロープのバランスを取り直すのに時間はかかりません。また、バランスを取り直すタイミングは、例えばロープの長さと深さの比較によって自動的に決定することができます。大抵のデータ処理ルーチンでは、ロープの文字への順次アクセスが必要になりますが、この場合ロープ・イテレーターがほぼ均一的な O(1) アクセス速度を実現できます。

表 1 に、ロープと Java String の一般的なストリング処理の予測実行時間を比較します。

表 1. 一般的なストリング処理の予測実行時間

処理	ロープ	Java String

連結	O(1)	O(n)
サブストリング	O(1)	O(1)
文字の取得	O(log n)	O(1)
すべての文字の順次取得 (文字単位のコスト)	O(1)	O(1)

Ropes for Java の紹介

メモリー上の問題

Java コードにおける一定時間で処理できるサブストリングの実装は、メモリー上の問題を引き起こすことになります。それは、サブストリング参照によって元のストリングに対するガーベッジ・コレクションができなくなるためです。この問題は `Rope` と `String` の両方が抱えています。

Java プラットフォームを対象とした高品質なロープ・データ構造の実装である Ropes for Java (「参考文献」を参照) は、卓越したパフォーマンスとメモリー使用量を全面的に実現できるように、広範にわたる最適化を行います。このセクションではロープを Java アプリケーションに統合する方法を説明した後、ロープのパフォーマンスを `String` および `StringBuffer` のパフォーマンスと比較します。

Ropes for Java 実装がクライアントに公開するのは、`Rope` というクラスだけです。`Rope` インスタンスは、`Rope.BUILDER` ファクトリー・ビルダーを使って任意の `CharSequence` から作成します。

リスト 1 に、ロープの作成方法を示します。

リスト 1. ロープの作成方法

```
Rope r = Rope.BUILDER.build("Hello World");
```

`Rope` は標準的な処理を行う一連のメソッドを公開します。これらのメソッドには以下の実行内容が含まれます。

- 別の文字シーケンスを付加する
- サブシーケンスを削除する
- 別の文字シーケンスをロープに挿入する

注意する点として、`String` の場合と同様、これらのミューテーションのそれぞれが新しいロープを返すため、元のロープは変更されずに維持されます。リスト 2 に、上記の処理を行った例をいくつか記載します。

リスト 2. ロープのミューテーション

```
Rope r = Rope.BUILDER.build("Hello World");
r = r.append("!");  // r is now "Hello World!"
r = r.delete(0,6);  // r is now "World!"
```

効率的なロープの繰り返しの処理

ロープを繰り返し処理するには注意が必要です。リスト 3 の 2 つのコード・ブロックを検討して、どちらのパフォーマンスが優れているか判断してみてください。

リスト 3. ループの繰り返しの処理の 2 つの手法

```
//Technique 1
final Rope r=some initialization code;
for (int j=0; j<r.length(); ++j)
    result+=r.charAt(j);

//Technique 2
final Rope r=some initialization code;
for (final char c: r)
    result+=c;
```

ループ内の任意の位置にある単一の文字を返す処理にかかる時間は、O(log n) であることを思い出してください。しかし、リスト 3 の最初のコード・ブロックでは文字ごとに `charAt` を行っているため、n 回分の O(log n) 検索時間がかかってしまいます。 2 番目のブロックでは代わりに `Iterator` を使用しているので、最初のブロックより実行時間が短くなるはずです。表 2 に、それぞれの手法を使って長さ 10,690,488 のループを繰り返し処理した場合のパフォーマンスを要約します。比較のため、表 2 には `String` と `StringBuffer` の時間を記載しています

表 2. 複雑なループの繰り返しの処理のパフォーマンス

手法	時間 (ns)
`String`	69,809,420
`StringBuffer`	251,652,393
`Rope.charAt`	79,441,772,895
`Rope.iterator`	1,910,836,551

表 2 に記載した結果 (予測時間と一致) を得るために使用したループは、最初のストリングに対して複雑な一連のミューテーションを実行して作成したものです。一方、ループを文字シーケンスから直接作成し、それから後はミューテーションを一切使用しないとすると、パフォーマンスの数値は驚くほど変わってきます。表 3 に、今度は Project Gutenberg 出版のチャールズ・ディケンス作『クリスマス・キャロル』(英語版) が含まれる文字配列からループを直接初期化し、そのループに含まれる 182,029 文字すべてを 2 つの手法それぞれで繰り返し処理した場合のパフォーマンスを比較します。

表 3. ループの繰り返しの処理のパフォーマンス

手法	時間 (ns)
`String`	602,162
`StringBuffer`	2,259,917
`Rope.charAt`	462,904
`Rope.iterator`	3,466,047

このパフォーマンスの逆転を前述の理論的検討に照らし合わせて説明すると、その大きな要因となっているのはループの構造です。基礎となる `CharacterSequence` または文字配列から直接組み立てられたループは、たった 1 つのフラット・ロープからなる単純な構造となります。このループには連結ノードもサブストリング・ノードも含まれないため、文字検索は基礎となるシーケンスの `charAt` メソッドに直接委任して行われる (`CharacterSequence` の場合) か、あるいは配列を直接検索する (配列の場合) ことになります。フラット・ロープに対する `Rope.charAt` のパフォー

マンスは、基礎となる文字表現のパフォーマンスと同じです (大抵は O(1))。したがって、このパフォーマンスの違いが生まれます。

洞察力のある読者なら、`charAt` とイテレーターのアクセス時間がどちらも O(1) だとすると、どうして前者の速度が後者の 7 倍以上にもなるのか不思議に思うことでしょう。この違いの原因は、Java 言語では `Iterator` が `Object` を返さなければならないからです。`charAt` 実装は `char` プリミティブをそのまま返しますが、イテレーター実装では各 `char` を `Character` オブジェクトにボックス化する必要があります。Autoboxing によってプリミティブをボックス化するという構文上の苦労は軽減されるとしても、それによるパフォーマンスへの影響をなくすことはできません。

最後に、`Rope.charAt` のパフォーマンスが `String.charAt` のパフォーマンスに勝っていることも注目に値します。その理由は、`Rope` は専用クラスを使用して遅延サブストリングを表すため、`charAt` の実装を通常のループのために単純に維持できるからです。それとは対照的に、`String` の Java SE 実装は同じクラスを使用して通常のストリングと遅延サブストリングを表すことから、`charAt` のロジックがある程度複雑になります。そのため通常のストリングを繰り返し処理する際には、パフォーマンスに多少影響が出てくるというわけです。

ループの繰り返しの処理については、ループでの正規条件検索のパフォーマンスを最適化する方法について説明するときに再び話題にのぼります。

Rope.write による出力の最適化

ある時点で、大抵のロープ・インスタンスはどこかしらの場所に書き込まなければなりません。通常、これはストリームに書き込みますが、ストリームに任意のオブジェクトを書き込むと、そのオブジェクトの `toString` メソッドが呼び出されることになります。このシリアライズ手法では、オブジェクト全体のストリング表現をメモリーに作成してからでないと単一の文字を書き込むことができません。大規模なオブジェクトの場合、これはパフォーマンスにとって大きな不利となります。大規模なストリング・オブジェクトを考慮して設計された Ropes for Java には、それよりも優れた手法が用意されています。

パフォーマンスを改善するため、`Rope` ではライター (`Writer`) と範囲指定を受け入れる write メソッドを導入し、ロープの指定範囲をそのライターに書き込みます。これによってロープから `String` を組み立てる際に節約される時間とメモリー・コストは、大規模なロープとなると計り知れません。リスト 4 に、ロープ出力の標準手法と拡張手法の両方を示します。

リスト 4. Rope 出力の 2 つの手法

```
out.write(rope);
rope.write(out);
```

表 4 に、長さ 10,690,488、深さ 65 のロープをメモリー内バッファーでバックアップされたストリームに書き込んだベンチマーク結果を記載します。この表からわかるのは時間が節約されることだけですが、一時的に割り当てられるヒープ・スペースの節約はこれよりも遥かに大きいことに注意してください。

表 4. ロープの出力パフォーマンス

手法	時間 (ns)

`out.write`	75,136,884
`rope.write`	13,591,023

mutator のパフォーマンスのベンチマーク

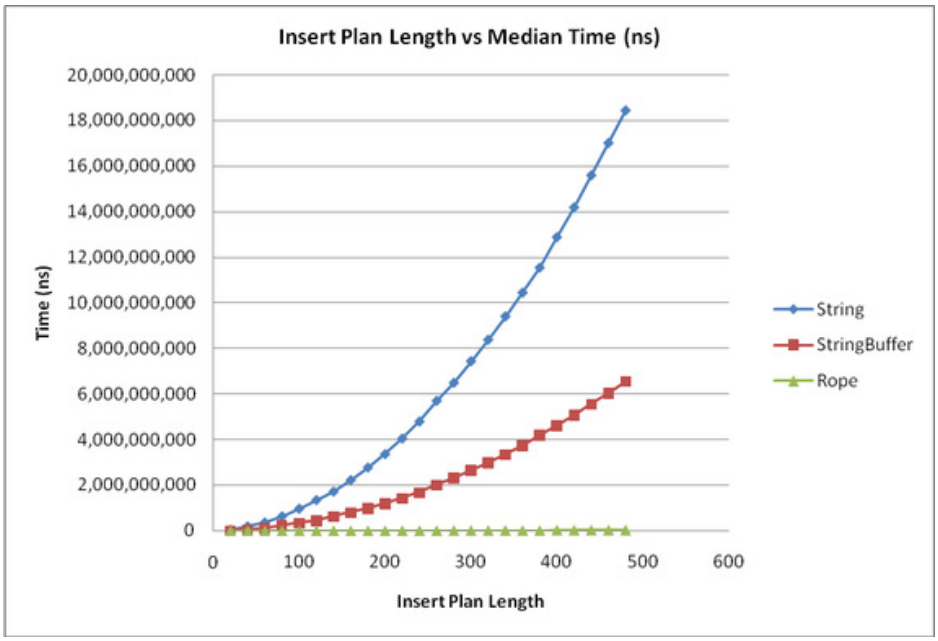
理論からすると、隣接した文字ストリング表現のミューテーションに比べ、ロープのミューテーションは遥かに短時間で行われますが、その反面、前述したように結果としてロープの繰り返し処理に時間がかかることになります。このセクションでは、いくつかのベンチマークによって Ropes for Java と `String` および `StringBuffer` のミューテーションのパフォーマンスを比較します。

すべてのテストは Project Gutenberg の 電子書籍『クリスマス・キャロル』(182,029 バイト) を使って初期化し、連続した一連のミューテーションを適用します。ほとんどの場合、データ構造にどの程度のスケーラビリティーがあるかがわかるように、ミューテーションの数は 20 から 480 の範囲で変えています (図 2、3、4 では、これを Plan Length (予定長) と呼んでいます)。また、各テストは 7 回ずつ実行し、その平均の結果を使用しました。

挿入のベンチマーク

挿入のベンチマークでは、前の繰り返しによる出力から無作為にサブストリングを選択し、ストリング内の任意の場所に挿入しました。このベンチマークの結果は、図 2 のとおりです。

図 2. 挿入のベンチマークの結果



`String` と `StringBuffer` ではいずれも、予定長が増えるに従って、ベンチマークを完了するまでの所要時間が指数関数的に増大しています。それとは対照的に、ロープのパフォーマンスは非常に優れています。

付加のベンチマーク

付加のベンチマークでは、入力ストリング自体に任意の範囲の入力ストリングを付加しています。このテストは、短時間で付加を実行するために `StringBuffer` が最適化されていることから興味深い内容となっています。このベンチマークの結果は、図 3 のとおりです。

図 3. 付加のベンチマークの結果

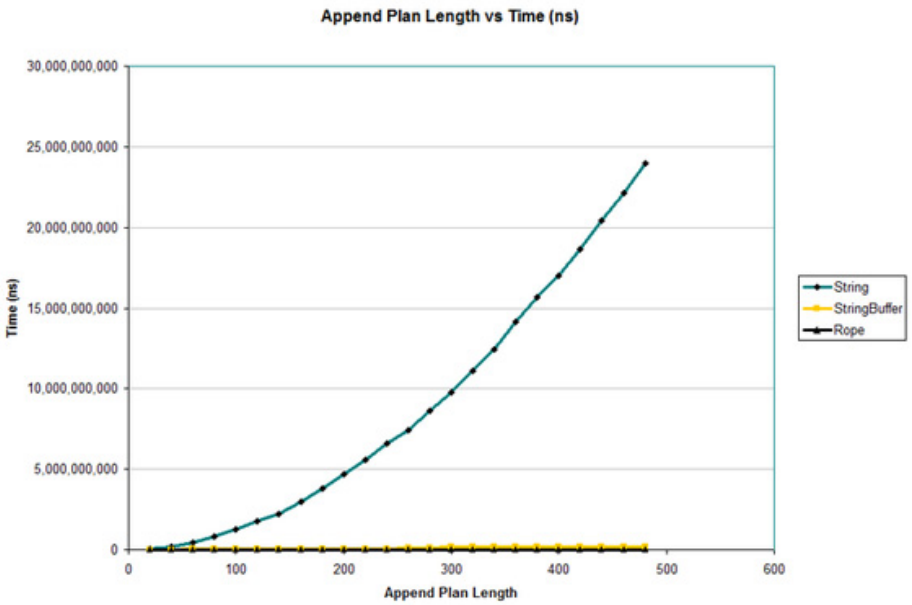
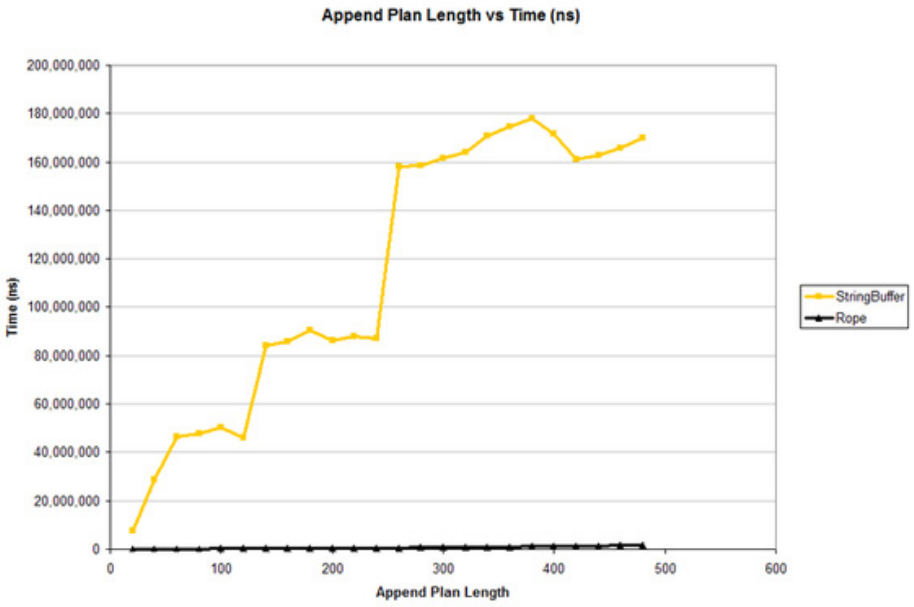


図 3 の表示では、残念ながら `String` の不良なパフォーマンスのために目立っていませんが、`StringBuffer` のパフォーマンスは期待通り優れたものになっています。

図 4 では `String` を除外し、`Rope` と `StringBuffer` のパフォーマンスをより明らかに比較できるようにグラフを作成し直しています。

図 4. 付加のベンチマークの結果 (String は除外)



ロープの不変性: 注意点

Rope クラスは不変になるように設計されています。これはつまり、mutator 関数は決して元の Rope インスタンスを変更することはなく、代わりに変更を加えたコピーを返すということです。このような不変性はさまざまな利点をもたらします。なかでも筆頭に挙げられるのは、Rope インスタンスをマルチスレッド環境で共有する際に、共有する上での事前処理が必要ないため、プログラム・ロジックが大幅に簡潔になるという点です。

Ropes for Java は基礎となる文字シーケンスからロープを組み立てます。CharSequence はインターフェースであることから、ロープの構成にはかなりの柔軟性がもたらされ、ロープをディスク上のファイル、メモリー内バッファー、リモート・サーバーに保存された文書などを始めとする異種のソースから組み立てることができます。ただし、不変であることが保証された String とは異なり、CharSequence の実装にはそのような制約が課せられません。そのためアプリケーション・プログラマーは、ロープを作成する際の基礎として使われる文字シーケンスがロープ・インスタンスの存続期間中、事実上不変であることを確実にする必要があります。

図 4 では、図 3 では明らかでなかった `Rope` と `StringBuffer` の劇的な違いがわかります。`Rope` は x 軸のわずかに上を行く値を示しています。しかし真に興味深いのは `StringBuffer` のグラフです。このグラフは滑らかに上昇しているのではなく、急激な上昇と停滞が繰り返されています (演習として、次のパラグラフを読む前にその理由を考えてみてください)。

その理由は、`StringBuffer` がメモリーを割り当てる方法に関係しています。このクラスは内部にある配列の最後に追加のスペースを割り当てることを思い出してください。これは効率的に付加を行えるようにするための措置ですが、そのスペースを使い尽くした後には、まったく新しい配列を割り当て、すべてのデータをそこにコピーしなければなりません。通常、新しい配列のサイズは現行の長さを基準に決定されます。したがって、予定長が増えるにつれ、結果として作成される `StringBuffer` の長さも同様に増えていきます。そしてサイズ変更しきい値に達すると、サイズ変更とコピーが発生することから所要時間が急激に増え、その後何回か予定長が増える間はパフォーマンスの停滞 (つまり、所要時間の緩やかな上昇) が起こるわけです。予定長項目ごとの合

計ストリング長さの増分値はほとんど同じなので、急激な所要時間の増大に続くパフォーマンスの停滞の比率が基礎となる配列のサイズ変更係数の指標となります。この特定の `StringBuffer` 実装の場合、以上の結果に基づいた正確な係数推定値は 1.5 前後です。

結果のまとめ

これまでは、`Rope`、`String`、そして `StringBuffer` でのパフォーマンスの違いをグラフを用いて説明してきましたが、今度は表 5 に、すべてのミューテーション・ベンチマークの時間計測結果を記載します。ここで使用しているのは、480 項目の予定長です。

表 5. 削除結果を追加した、480 項目の予定長による平均ミューテーション時間

ミューテーション/データ構造	時間 (ns)
挿入	
`String`	18,447,562,195
`StringBuffer`	6,540,357,890
`Rope`	31,571,989
前に付加	
`String`	22,730,410,698
`StringBuffer`	6,251,045,63
`Rope`	57,748,641
後ろに付加	
`String`	23,984,100,264
`StringBuffer`	169,927,944
`Rope`	1,532,799
削除 (最初のテキストから 230 の任意の範囲を削除)	
`String`	162,563,010
`StringBuffer`	10,422,938
`Rope`	865,154

ベンチマーク・プログラムへのリンクは、「参考文献」を参照してください。このプログラムをご使用のプラットフォームで実行して、ここに記載した結果を検証することをお勧めします。

正規表現パフォーマンスの最適化

JDK のバージョン 1.4 で導入された正規表現は多くのアプリケーションで広く使用されている機能です。そのため、正規表現がロープで優れたパフォーマンスを示すことが重要となります。Java 言語では、正規表現は `Pattern` として表されます。`Pattern` を任意の `CharSequence` の領域に突き合わせるには、パターン・インスタンスを使って `Matcher` オブジェクトを構成し、`CharSequence` をパラメーターとして渡します。

`CharSequence` を操作できるということは、Java の正規表現ライブラリーに極めて高い柔軟性がもたらされますが、これには深刻な欠点も伴います。`CharSequence` が提供するメソッドは、その文字にアクセスするためのメソッド、`charAt(x)`. だけです。概要のセクションで説明したように、多数の内部ノードを持つロープで任意の文字にアクセスするのに要する時間はおおよそ O(log n)

で表されるので、巡回に要する時間は O(n log n) となります。これによって生じる問題を説明するため、長さ 10,690,488 のストリングで「`Crachit*`」パターンのすべてのインスタンスを検索するために必要な時間のベンチマーク・テストを行いました。使用したロープは、挿入のベンチマークと同じ一連のミューテーションによって構成したもので、その深さは 65 です。表 6 に、結果を記載します。

表 6. 正規表現の検索時間

手法	時間 (ns)
`String`	75,286,078
`StringBuffer`	86,083,830
`Rope`	12,507,367,218
バランスを取り直した後の Rope	2,628,889,679

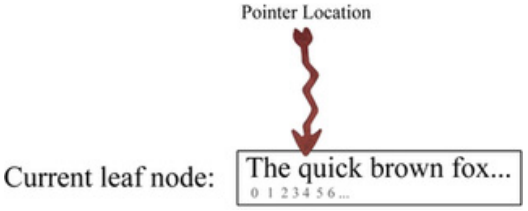
明らかに、`Rope` でマッチングを行うときのパフォーマンスは劣っています (誤解を避けるために言っておくと、これが当てはまるのは多くの内部ノードを持つ `Rope` です。フラット・ロープに関しては、パフォーマンスは基礎となる文字表現の場合とほとんど同じになります)。`Rope` のバランスを取り直すことによってマッチングは 3.5 倍速くなっていますが、それでも `Rope` は `String` と `StringBuffer` のいずれにも及びません。

この状況を改善するには、大幅に高速な O(1) アクセス時間を実現する `Rope` イテレーターを `Matcher` オブジェクトから利用するようにする必要があります。この仕組みを把握するためにはまず、`Matcher` オブジェクトがその `CharSequence` にアクセスするパターンを理解しなければなりません。

正規表現のマッチングでもっとも一般的なシナリオは、まず文字シーケンスのいずれかのポイントから開始し、すべての一致が検出されてシーケンスの終わりに達するまで順方向に進むというものです。このシナリオではマッチャーは主に、大抵は一度に複数の文字分、順方向に進みます。しかし時には、マッチャーが逆方向に進まざるを得ない場合もあります。

一度に複数の文字分、順方向にスキップさせるように `Rope` イテレーターを変更するのは簡単ですが、逆方向の移動となるとそう簡単には行きません。内部では、イテレーターが `Rope` の先行順深さ優先探索による巡回を実行してリーフ・ノードのそれぞれにアクセスするからです。巡回スタックには前のリーフに移るだけの十分な情報がありませんが、イテレーターが現行のリーフを離れない範囲であれば、逆方向の移動にも対応することはできます。説明のため、図 5 にこの仮説に基づいたイテレーターの状態を示します。このイテレーターは逆方向に 1 から 4 文字分の範囲を移動しますが、それ以上は移動しません。なぜなら、それ以上移動すると前のリーフにアクセスしなければならなくなるからです。

図 5. サンプル・イテレーターの状態



この前のリーフにアクセスする新しい機能を有効にするには、`Rope` の `charAt` メソッドを変更して、最初の呼び出しで指定の位置にイテレーターが構成されるようにします。それ以降の呼び出しで、イテレーターを必要な距離だけ逆方向または順方向に移動させます。イテレーターが必要な距離だけ逆方向に移動できない場合には、(めったに起こって欲しくないシナリオですが) 文字の値を取得するためにデフォルト `charAt` ルーチンが実行されます。

この最適化は通常は適用可能でないこと、そして新しいメンバー変数の導入が必要になることから、`Rope` クラスに直接追加することは避けなければなりません。代わりに取れる手段は、要求に応じてロープをこの最適化で修飾することです。この手段を可能にするため、Ropes for Java は指定のパターンを対象に最適化されたマッチャーを作成するメソッドを `Rope` クラスに組み込みます。リスト 5 に、その手法を示します。

リスト 5. 最適化された正規表現のマッチング

```
Pattern p = ...
Matcher m = rope.matcher(p);
```

リスト 5 の 2 行目の呼び出しが、ロープを修飾して正規表現のマッチングを最適化します。

表 7 に、この手法を使った場合のベンチマークの結果を記載します。この表には、明確にするために前の結果 (表 6) も併せて記載しています。

表 7. rope.matcher() による正規表現の検索時間

手法	時間 (ns)
`String`	75,286,078
`StringBuffer`	86,083,830
`Rope`	12,507,367,218
バランスを取り直した後の Rope	2,628,889,679
`Rope.matcher`	246,633,828

この最適化はバランスを取り直した後のロープの 10.6 倍もの飛躍的改善をもたらし、ロープと String のパフォーマンスの違いを 3.2 倍以下に抑えています。

ロープの適用

ロープを使うべきでない場合

エンタープライズ Java アプリケーションには、以下のようなコードが含まれていることがよくあります。

```
String x = "<input type='text' name='name' value='"
+ escapePcData(bean.getName()) + "'>";
```

x は HTML ページの一部として、後でクライアントのブラウザーに送信されます。この x の値を計算するには、コンパイラーがデフォルトで生成する `StringBuilder` ではなく、ロープを使ったほうが理にかなっているでしょうか。

答えはノーです。これにはさまざまな理由があります。まず、連結されているデータの量はおそらく少ないため、ロープを使用することによって堅牢性とスケーラビリティーは改善されることはあっても、パフォーマンスの改善は見込めないからです (`getName` が意外にも

50 MB のストリングを返したとしたら、両方のソリューションがどのように振る舞うかを考えてみてください)。

一方、議論のネタとして、多数のデータ・チャンクが連結されている場合を想像してみてください。付加を実行するときのパフォーマンスは、一般に `StringBuffer` よりも `Rope` のほうが優れていることから、この場合はロープを使うのが妥当でしょうか。その答えは同じくノーです。入力されたデータを結合してフォーマット化した出力を作成する場合は常に、StringTemplate や FreeMarker などのテンプレート・エンジンを使用するのが最も簡潔かつ効率的な手法となります。この手法は表現マークアップを明確にコードから分離するだけでなく、いったん (大抵は JVM バイトコードに) コンパイルされたテンプレートは再利用できるため、優れたパフォーマンス特性がもたらされます。

テンプレートを使用する上でのもう 1 つの利点は、ロープを使用して作成したものをはじめ、上記のコードに示されたような出力作成ルーチンによくある基本的な欠陥を明らかにします。その利点とは、テンプレートはインクリメンタルに評価できるため、作成された出力をまず不必要にメモリー内に蓄積するのではなく、作成されると同時に出力をライターに書き込めるということです。Java EE アプリケーションの場合、ライターは実際にはクライアントのブラウザーへのバッファー接続なので、その他のソリューションのメモリー使用量は O(n) である一方、この出力レンダリング手法が使用するのは一定のメモリー、つまり O(1) です。小規模な入力やアプリケーションの負荷が小さい場合には明白ではありませんが、これはアプリケーションのスケーラビリティーと堅牢性に大きな改善をもたらします (「参考文献」にリンクを記載した、ストリーミング・アーキテクチャーに関する 2 つの記事を読んでください。これらの記事ではこの手法の詳細を説明し、定量化しています)。

ロープのパフォーマンスについて十分理解できたところで、今度はロープの従来の使用方法、ならびに Java EE アプリケーションで陥りがちな不適切な使用方法について検討します。

ロープは、連続したメモリーを使用するストリングの代わりとして多目的に使用することができますが、顕著なパフォーマンスの改善が期待できるのは、大規模なストリングを大々的に変更するような場合のみです。当然のことかもしれませんが、初期の頃のロープは、テキスト・エディターで文書を表すことに適用されました。極めて大規模な文書の場合でも、ロープを使うとほぼ一定の時間でテキストを挿入および削除できるだけでなく、ロープの不変性が取り消しスタックの実装をありきたりなものにします。つまり、変更を行うごとに前のロープへの参照を保存すればよいだけの話となります。

さらに巧妙にロープを適用できるのが、仮想マシンでの状態を表す処理です。例えば、ICFP 2007 Programming Contest では、サイクルごとにその状態を変更し、何百万回にもわたって入力サイクルを実行する仮想マシンの実装が絡んでいました (「参考文献」を参照)。ある Java 実装では特殊化した `StringBuffer` の代わりに `Rope` を使って状態を表現するようにしたことにより、仮想マシンの速度が 3 桁の規模 (毎秒最大 50 サイクルから毎秒 50,000 サイクル以上) で改善されました。

今後の検討方針

Ropes for Java は新しいライブラリーとはいえ、土台となっている概念は以前からのものなので、このライブラリーはロープのパフォーマンスを約束すると思われます。その一方、このプロジェクトでは今後のリリースで以下のようなライブラリーの改善を予定しています。

- 他にもよく使用されるストリングの処理のハイパフォーマンス実装を提供すること。
- ロープを Scala (「参考文献」を参照) やその他の Java プラットフォーム用拡張言語にシームレスに統合するアダプターを作成すること。
- さらに自動化を進めたテストによって品質を向上させること。Ropes for Java のテストは現在、手動で作成した自動 JUnit テストと、JUnit Factory によって生成された自動テストの両

方を使って行われます。ESC/Java2 でチェックされた JML (Java Modeling Language) アノテーション (「参考文献」を参照) を組み込むことで、さらなる品質の向上が期待できます。

著者について

Amin Ahmad

Amin Ahmad は無所属のコンサルタントとして活躍する傍ら、Ahmadsoft, LLC の代表取締役を務めています。Java コンサルタントとしての経験は 7 年以上を数え、ハイパフォーマンスのエンタープライズ Java システムをクライアントにもたらすことを専門としています。