MIKE DUIGOU

遅延処理によるJavaコレクションの高速化

ArrayListやHashMapに遅延処理を追加し、パフォーマンスの改善とメモリ使用量の削減を図る

Javaのコア・ライブラリ・コミュニティは、Javaのコレクション・フレームワークに遅延処理を加えることによって改善を図ろうと必死になっています。ソフトウェアの世界において遅延とは、結果が必要だと明確にわかるまで結果の生成を遅らせる、アーキテクチャまたはシステムによるアプローチを指します。多くの命令は、複数のサブ命令の集合に分解できます。遅延させるとは、命令の結果が別のサブ命令や命令全体の完了に必要となるまでサブ命令の実行を先送りすることです。

ある命令の完了に遅延を用いないアプローチでは、一連のサブ命令がすべて実行され、その結果を組み合わせて最終結果が生成されます。遅延によるアプローチは、遅延を用いないアプローチよりも効率的です。このアプローチでは、はじめにサブ命令の結果を組み合わせます。サブ命令が実行されていないために必要な結果が存在しないことがわかると、そのサブ命令を実行してサブ結果を求めます。当初、計算済みのサブ結果は存在しません。この状態からスタートし、サブ命令を実行して結果を蓄積してゆくと、その過程でさらにサブ命令の実行が可能となり、最終的に命令全体の結果が導き出されます。遅延処理によってうまく高速化できるのは、命令の結果が出た際に、まだ結果を求めていないサブ命令がある場合です。結果を求めていないのは、その値が最終的な命令の結果を求めるために必要でないからです。

命令の結果が最終結果の一部として必要でないかもしれない場合、その可能性の大小を問わず、結果が実際に必要とわかるまで命令を遅延させるのが合理的です。遅延の例では、式の評価を取り上げるのがもっとも一般的です。次のコードについて考えてみましょう。

```
int x = 5;
int y = 3;
if (x < 2 && y < 7) {
 ...
```

この式を評価するもっとも簡単な方法は、それぞれの項を評価してから組み合わせるというものです。

```
5 < 2 => FALSE
3 < 7 => TRUE
FALSE && TRUE => FALSE
```

ここで注意すべき点は、最初の項がfalseと評価される場合、この式全体は必ずfalseになることです。そのため、最初の項がtrueと評価される場合以外は、わざわざ2番目の項を評価する必要はありません。

Javaプログラムでは、式の項は左から右に評価され、結果に必要でない項はまったく評価されないことが『Java Language Specification』で規定されています。多くの場合、この仕様によって計算を省略できます。この仕組みは、次のような場合にも便利です。

```
if (foo != null && foo.bar() == 3) {
```

この例では、bar()メソッドを呼び出す場合、fooがnullでない必要があります。fooがnullの場合にこの式の2番目の項を評価しようとすると、NullPointerExceptionが発生します。Javaの遅延評価ルールによって、2番目の項は最初の項がtrueの場合にのみ評価されることが保証されています。論理積（&&）でつながれた項は、ネストと同じ意味と考える

ことができます。そのため、

```
if (foo && bar && baz) {
```

は、次と同じ意味になります。

```
if (foo)
 if (bar)
   if (baz)
```

このようにネストした条件式に書き換えると、不要な項が評価されない理由もわかりやすくなります。遅延評価は、次のような論理和（||）の式にも適用されます。

```
if (true || something) {
```

この条件式のsomethingの項は評価されません。評価前に式の値を決定できるからです。

以上のような、式の評価における遅延処理の例は、効率的かつ簡潔（効率と同じくらい重要な要素と思われます）なプログラム・ロジックを書く上で役立ちます。他の形態の遅延処理も同じく有用なものですが、プログラムの流れで行われる判断と一般的に得られるメリットの間の関係は直接的なものでも、即座にわかるものでもありません。

計算の結果が使われることなく捨てられる場合、その頻度によらず、結果の生成に必要なリソースの使用を、結果が必要になるまで回避しようと考えるのは合理的なことです。遅延処理によって節約できるリソースのうち、もっともわかりやすいのはCPUサイクルです。しかし、割当てを回避することによるメモリの節約や、不要なファイル、ソケット、スレッド、データベース接続などを作らないことによるシステム・リソースの節約も可能です。状況によっては、大幅なリソースの節約につながる可能性もあります。

遅延処理の実装がシステム・パフォーマンスを向上させる決定的な最適化戦略になることもあります。遅延処理によるパフォーマンスの向上は、作業の効率を上げることではなく、不要な作業を避けることによって実現されます。遅延処理は、アプリケーションが行うデータベースの問合せの数を30%減らすことに似ています。同じアプリケーションでデータベースの問合せのパフォーマンスを3%向上させるのとは正反対のアプローチです。実現可能な場合、前者にエネルギーを注ぐ方がはるかに効果的です。

ごく一部ですが、遅延処理によってメモリ使用量が最大20%削減され、メモリの書き換え回数も同じくらい少なくなったアプリケーションもありました。

コレクションに遅延処理を導入する際の課題

Javaコレクション・フレームワークは、すでにかなりの最適化が行われています。そこに遅延処理を実装するというアイデアは、アプリケーションの動作を分析した結果生まれました。OracleのPerformance Scalability and Reliability（PSR）チームは、いくつかのOracleフレームワークやそのフレームワーク上で動作するアプリケーションのパフォーマンスを評価しました。PSRチームは、ArrayListやHashMapのインスタンスが割り当てられても、そのインスタンスが使われることがないまま、インスタンスが格納されたオブジェクトの生存期間が終了する場合が非常に多いことを発見しました。この結果は、アプリケーションとミドルウェアの両方に当てはまるものでした。割り当てられたArrayListやHashMapのインスタンスの約2%は、何の要素も受け取ることがありませんでした。さらに分析を進めたところ、コレクションが使われている場合もありましたが、常に必要というわけではないこともわかりました。そこでPSRチームは、アプリケーションをリファクタリングして同じベース・クラスを持つ個別のクラスを作り、1つのサブクラスで使われない場合があるコレクションを定義することによって、コレクションが必要な場合と必要でない場合を扱うことができるかどうかを確認しました。しかし、このアプローチは有効ではありませんでした。次の代表的な例に見られるようなケースが多かったからです。

```
public abstract class RestRequest {
   protected final Map<String,String> httpHeaders =
       new HashMap<>();
   protected final
       Map<String,List<String>> httpParams =
           new HashMap<>();
   protected final Set<Cookie> httpCookies =
```

```
new HashSet<>();
```

RestRequestは、アプリケーションがHTTP RESTの問合せを行う際に使用する架空のサンプル・クラスです。RESTは、HTTP通信を使用したAPIを作成する際に一般的に用いられます。この例において、それぞれのRestRequestオブジェクトは個別にフォーマットされたHTTPリクエストであり、REST API（ホスト・アプリケーションが提供）の呼出しを表しています。各RestRequestインスタンスは、リクエストを受信するアプリケーションにHTTPメッセージの重要な特性を提示できる必要があります。重要な特性には、HTTPヘッダー（httpHeaders）、問合せ文字列またはフォーム・データのいずれかのHTTPパラメータ（httpParams）、HTTP Cookie（httpCookies）などがあります。HTTPのこのような特性のそれぞれは、どのHTTPメッセージにも存在する可能性があります。しかし、正確な用途は、REST APIを提供する個々のアプリケーションやそのREST APIを使用するクライアント・アプリケーションによって決まります。

リクエストでどのようなHTTP特性が利用されるかは事前にはわかりません。そのため、使われる可能性があるそれぞれの特性のためのデータ構造をRestRequestに含めてしまうことは問題です。HTTPヘッダー、パラメータ、Cookieは一般的で、HTTPプロトコルには必須です。しかし、アプリケーションでこれらすべての特性を使う必要があるわけではなく、1つも使わないことすらあります。REST APIの中には、HTTPパラメータを使うものもあれば、Cookieやヘッダーを使うものもあります。このようなオプションの特性に対応するアプローチの1つは、使用される可能性のあるHTTP特性のすべての組み合わせを表すさまざまな種類のRestRequestクラスを用意することです。しかし、この方法は煩わしく使いにくいものです。もしある種類のリクエストが、あるHTTP特性を使うことがわかったとしても、一部のリクエストでしかその特性を使わないということもよくあります。たとえば、同じリクエストに対して、認証されたユーザーはCookieを使っても、認証されていないユーザーは使わないという場合を考えます。大多数のリクエストは認証されていないユーザーのものでしょう。

RestRequestクラスは1つだけにし、httpParamsフィールドは使うか使わないかにかかわらず常に初期化していつでも使えるようにしておく方が、プログラム・ロジックははるかにシンプルになります（後ほどこの点に戻ります）。

フレームワークやアプリケーションをリファクタリングして、必要不可欠ではあるがほとんど使われないhttpParamsフィールドやhttpCookiesフィールドを削除することはできないことがわかりました。そのため、代案が必要になります。

最終的な目的は、httpParamsのようなフィールドが実際には使われない場合に、クラスにそのフィールドを作成することによって発生するコストを回避し、アプリケーションのパフォーマンスを改善することです。解決策の1つは、RestRequestクラスのhttpParamsフィールドの初期化を遅延させることです。つまり、HTTPパラメータが存在するとわかった場合にのみHashMapを作成するということです。この方法では、httpParamsを使うすべての場所に、次のコードによるガード・チェックの追加が必要になります。

```
if( httpParams != null )
```

しかし、RestRequestクラスは拡張できるように設計されているため、RestRequestを拡張したすべてのサブクラスで、httpParamsを使うたびに同じチェックをする必要があります。このチェックを行っていない既存のコードもたくさんあったため、RestRequestが突然一律にフィールドの初期化を止めるというのは現実的ではありませんでした。

httpParamsにnullを許可すると、それぞれに対してガード・チェックの追加が必要となるため、メソッドのロジックが複雑化することにもなります。RestRequestにあるさまざまなフィールドで同じアプローチを使う場合、RestRequestやそのサブクラスのメソッドのロジックでは、RestRequest内のどの特性が使われているのかを判定するために繰り返されるチェックに過度に集中することになります。やがて、必然的にエラーが紛れ込むか、RestRequestの別の部分を変更した際に対応漏れが発生することになるでしょう。nullを含むフィールドのガード・チェックを1つ忘れただけで、丸1日が犠牲になることすらあるかもしれません。

また、多くの場合に有効ですが、今回は使うことができない解決策もあります。この方法では、次に示すhttpParamsの宣言部分は変更せず、そのままにしておきます。

```
protected final Map<String,List<String>> httpParams;
```

その上で、初期化処理をコンストラクタに移し、HTTPパラメータが

リクエストに存在しないことがわかった場合、リクエストごとに専用のHashMapインスタンスを作るのではなく、httpParamsフィールドをCollections.emptyMap()で初期化します

Java 8では、重要な遅延処理の実装も新しく追加されています。Streams APIです。**このライブラリでは、根幹をなす原理として遅延処理を使用しています。**

(Collections.emptyMap()、emptyList()、emptySet()を使用します。この3つを含むCollectionsユーティリティ・クラスの類似のユーティリティ・メソッドを使うと、効率よく空のコレクションを提供できます。このような空のコレクションは、空のアイテムを格納する専用のインスタンスを作成する代わりによく使われます。多くの場合、この方法はnullを返すよりも効率的です。領域をさらに消費することはない上、nullを返すこともないため、必要となる結果のnullチェックも省けるためです)。

ここまでは手始めに、HashMapおよびArrayListのインスタンスの作成を回避することで、フレームワークやアプリケーション内に遅延処理を追加してきました。しかし、もっとも効果的なのは、Javaコレクション・フレームワーク自体の内部に遅延処理を実装するアプローチであることが明らかになりました。

Javaコレクションの更新

ArrayListやHashMapなどの基本的なJavaクラスを変更するのは大変な作業です。数え切れないほどのプログラマーや、それ以上に多くのコードがこういったクラスを使っています。プログラマーもプログラムも、Javaのバージョンが変わっても信頼性のある一貫した動作とパフォーマンスが提供されることを期待しています。Javaコレクション・フレームワークは開発者との約束事であり、指定された動作を提供するためのプログラムです。Javaのアップデートや、ひいてはメジャー・リリースであっても、JDKのクラスの機能を再定義することは、約束事を変えることになり、本質的に不可能です。APIの約束事をわずかに見直すことは可能ですが、Javaコレクション・フレームワークに対して行われてきた改善のほとんどは内部的な変更でした。さらに、内部的な変更であっても、望まない副次的作用や意図しない動作変更につながらないように、十分慎重に検討する必要があります。

先ほど、一部のフィールドがnullになる可能性がある場合、RestRequestの例のようなクラスを使うのは難しいと述べました。publicフィールドやprotectedフィールドがnullになる可能性がある場合、これらのフィールドにアクセスする際には追加の作業が必要となります。フィールドの参照先を見るたびに、そのフィールドのnullチェックをするというガードが必要になるということです。nullチェックが漏れるというのは、ベース・クラスにnullになる可能性があるフィールドが存在するプログラムで起こりがちなエラーです。protectedやpublicのフィールドにはnullを認めないというのが、一般的に推奨される事項です。フィールドがprivateの場合、nullになる可能性があるフィールドはわずかに扱いやすくなります。フィールドへの参照がすべて1つのファイルの中にあるため、オブジェクトのすべての状態においてフィールドがどのような値を取りうるかについて考えることがはるかに簡単であるからです。

ArrayListとHashMapは、要素やマップ・エントリを格納する基幹データ構造にpackage privateな配列フィールドを使用しています。ArrayListやHashMapオブジェクト自体を除けば、この配列はArrayListや空または空に近いHashMapsによって割り当てられる唯一のメモリであり、もっとも多くのメモリが割り当てられます。この2つのクラスに遅延処理を導入するというアップグレードを考える場合、その根幹となるのは、この配列フィールドがnullになることに対するガード・チェックの追加です。

ArrayListやHashMapに遅延処理を追加する主なメリットは、最初の要素が配列やマップに格納されるまでこの配列の割当てを遅らせることができる、または割当てを行わずに済む可能性があることです。アプリケーションによっては、この内部配列のメモリの割当てにかかるコストは大きなものになります。高コストに該当しないアプリケーションでも、未使用のコレクションが非常に短命で、割当てメソッドが完了してから割り当てられるメモリはないことがわかっている場合は特に、配列を初期化する計算を節約できるというメリットがあります。

関連するフィールド参照の対象が配列であるため、JVMは、インデックスを使って配列を参照する前や、配列の長さを求める前に、配列がnull

でないのをチェックすることをすでに要求されていました。追加のガード・チェックは、すでに行われていた暗黙的なnullチェックを明示的に行うことにすぎませんでした。つまり、ガード・チェックを追加しても、パフォーマンス面で不利になることはありませんでした。

もう1つの懸念事項は、さまざまなメソッドでのガード・チェックや割当てロジックの追加によって、HotSpotがメソッドをインライン化するかどうかの選択に影響するのではないかということでした。メソッドのインライン化が行われない場合、パフォーマンスを損なう可能性があるため、この選択は重要です。インライン化は、HotSpotが小さなメソッドに対して行う最適化です。多くの場合、HotSpotが短いかまたはシンプルなメソッドを呼び出すコードをコンパイルする際に、メソッドの呼出しを、呼び出されるメソッドの実際のコードで置換します。HotSpotがインライン化するメソッドのサイズには制限があります。調査の結果、遅延処理の追加によって変更したすべての重要なメソッドで、インライン化の制限境界には近づいていないことがわかりました。また、いくつかの場所で既存の内部メソッドを再利用することによって、HotSpotによるインライン化の改善もできました。あまり重要でないメソッドの中には、遅延に関連する変更の結果、わずかに遅くなったものがいくつかありました。しかし、汎用的なベンチマークやパフォーマンス・テストでは、変更によって最終的には高速化されていることが判明しました。筆者の知る限り、この変更によって深刻な副次的作用が発生したという事例は現在まで1つもありません。

まとめ

JVMアプリケーションのメモリ使用量を評価する場合、最大メモリ使用量の他にも考慮すべき点があります。JVMはメモリ管理にガベージ・コレクションを使用しているため、メモリ割当て速度やガベージ・コレクションの負荷も考慮する必要があります。メモリ割当て速度とは、アプリケーションが新しいオブジェクトを割り当てる時間に対する、割当てサイズの比率を指します。アプリケーションによって割当て速度は大きく異なり、多くの場合、この割当て速度がスループットにおける重要な要素になります。割当て速度に関係するのは、未使用オブジェクトのガベージ・コレクションに費やす必要がある作業量です。アプリケーションの実行に必要な空きメモリを常に十分確保するために、ガベージ・コレクションが行われます。ガベージ・コレクションの負荷とは、ガベージ・コレクションによってどのくらいのスループットが犠牲になっているかを示します。一般論としては、割当て速度が低下すれば、ガベージ・コレクションが必要な量も少なくなります。

一般的なフレームワーク・アプリケーションでは、ArrayListやHashMapの初期化の遅延という変更によって改善されるのは、ほんの1〜2%のメモリ使用量や割当て速度で、パフォーマンスの向上もかろうじて測定できる程度です。ただ、重要なのは、メモリ使用量が増えたり、パフォーマンスが落ちたりするアプリケーションはなかったということです。ごく一部ですが、メモリ使用量が最大20%削減され、メモリの書き換え回数も同じくらい少なくなったアプリケーションもありました。一部のアプリケーションにこのような劇的なメリットがある一方で、ほとんどのアプリケーションがわずかなメリットを得ることができ、認識されている悪影響がないことから、初期化の遅延という変更は重要な改善となりました。

ここで再びRestRequestの例を考えてみましょう。ArrayListやHashMapに遅延処理が追加されたことによって、このクラスの動作やパフォーマンスにはどのような影響があるでしょうか。RestRequestフィールドがfinalなArrayListやHashMapのインスタンスで無条件に初期化される点は変わりません。つまり、RestRequestの使い方は変わらないということです。ユーザーのプログラムに、フィールドがnullになるかどうかのチェックを追加する必要はありません。しかし、現実的には、作成されるコレクションのインスタンスの多くが最終的に空のままである可能性が高いでしょう。要素の配列の作成が先延ばしされるため、メモリとCPUサイクルの両方の節約になります。

Javaのように20年以上にわたって改善が続けられてきたソフトウェア・システムでは、あらゆるケースで副次的作用なしにメリットを提供できる実装の変更を見つけるのは難しいものです。この変更について検討したとき、ArrayListやHashMapの一般的な使用に悪影響がないことを確認しました。HashMap.get()のような重要なメソッドのパフォーマンスを分析する際には、一般的にすべてのJavaバイトコードとすべてのCPUサイクルのコストを調査します。こういったメソッドは、何百万ものJVMで毎年何兆回も実行されるものだからです。わずかなパフォーマンス・コストの変化や、ある実行パスから別の遅いパスへの変更のコストは許容できるかもしれません。しかし、パフォーマンスの悪化はごくわずかである必要があります。たいていの場合は別の場所で行うパフォーマンスの改善が悪化を大きく上回る必要があります。

問題の分析は、未使用のコレクションのコストを削減するためにでき

ることを探してアプリケーションやフレームワークの動作を調査するところから始まりました。この種のトップダウンによるパフォーマンス分析は、アプリケーションのパフォーマンスを改善する際に取ることができる圧倒的に優れたアプローチです。

Javaコレクション・フレームワークに別の形で遅延処理を導入することも検討されています。もっとも要望が強い変更は、HashMapの構築とサイズ変更の方法です。HashMapの典型的な使用パターンから、小さなマップ（と最初にget()操作を行う前の大きなマップ）に別のデータ構造を使用すると、実装にメリットがある可能性がわかっています。

Javaクラス・ライブラリには、他にも遅延処理が使われている場所があります。もっともよく挙げられるのは、hashCode()メソッド内でハッシュ・コードの計算結果をキャッシュする部分でしょう。この処理によって、Stringなどのクラスのパフォーマンスに大きなメリットが生まれています。キャッシュに関連する同じような事例も追加されています。こういったキャッシュには、繰り返し作業を避けることによってパフォーマンスを向上させるものがあります。また、同じデータ構造を複数の操作で再利用してメモリを節約するキャッシュもあります。メリットがあることがわかれば、他にもキャッシュの事例や、遅延処理による最適化が将来のJavaアップデートに追加される可能性があります。キャッシュが無駄だったり、キャッシュを維持するために実際に必要となる作業が多すぎたりしたため、実装からキャッシュが削除された場合もありました。多くの場合、Javaライブラリに遅延処理の実装を追加しても、APIを変更する必要はありません。そのため、メリットが大きく影響が少ない変更と言えます。

Java 8では、重要な遅延処理の実装も新しく追加されています。Streams APIです。このライブラリでは、根幹をなす原理として遅延処理を使用しており、多くの場合、シンプルな宣言的アプローチよりもはるかに高いパフォーマンスが実現します。

遅延処理は、Javaライブラリに大きなメリットをもたらした重要な最適化です。他のユーザーが使用するライブラリのパフォーマンスを改善する必要がある場合や、アルゴリズムの改良など主要な最適化の多くがすでに実装されている場合は、遅延処理について真剣に検討してみるとよいでしょう。**</article>**

Mike Duigou（@mjduigou）：Liquid Robotics所属。海洋で動作するJavaベースのロボットに携わる。元OracleのJavaコア・ライブラリ・チームの開発者で、コア・コレクションやJava 8のラムダ式ライブラリに貢献。自動運転車、踊るロボット、産業用リアルタイム・アプリケーションにも従事。

learn more

遅延処理がArrayListの割当てサイズに与える影響