Java共有クラス

Javaアプリケーションをより高速に、小容量のメモリで起動する

Lakshmi Shankar
Java Technology Center Development Team
IBM Hursley Labs

2004年 5月 08日

Simon Burns (simon_burns@uk.ibm.com)
Java Technology Center Development Team
IBM Hursley Labs

Roshan Nichani (nichanir@uk.ibm.com)
Java Technology Center Development Team
IBM Hursley Labs

Javaアプリケーションは現在、問題に直面しています。Javaアプリケーションを収容できるのはJava仮想マシン（JVM）プロセスのみです。複数のJVMがある場合には、それぞれアプリケーションを分離させておく必要がありますが、これには２つの大きなマイナス面があるのです。一つは各JVMの呼び出しに関わる起動時間であり、もう一つは各JVMが要求するメモリの容量(memory footprint)です。こうしたコストに加え、JVM内でアプリケーションを分離できないことを考えると、問題解決のためには何か根本的なことが必要なのは明らかです。その答えが、共有クラスなのです。この記事では、IBMのJava技術センター開発チーム（IBM Java Technology Center Development Team）のメンバーであるLakshmi ShankarとSimon BurnsそれにRoshan Nichaniが、JVMにおける共有クラスの背景にある概念、また共有クラスがどのように動作するのか、ユーザーはどのようにこの技術を利用できるのかについて説明します。さらにこの技術の実装の現状と、将来どのような利用が行われるかについても説明を行います。

それぞれのJavaアプリケーションを本当に分離するには、本質的に複数のJVMが必要になります。ただし、初期コストやメモリ消費の面から、この手法はとても理想的なものとは言えません。ところが共有クラスは、この両方の問題を解決しようとしているのです。複数JVM環境下では、共有クラスを使うことにより、（システムクラスを共有メモリにロードして）システムクラスのコア・セットを共有できるようになります。こうした共有クラスは、すべてのJVMに対して一定のまま保持される共有のメモリ領域に置かれます。その結果、共有クラスは最初に使う時にのみメモリにロードすればよく、その後に行われるJVM呼び出しの度にロードする必要が無くなり、また各JVMでのメモリ容量のコストも削減できるのです。

共有クラス技術はIBMがz/OSプラットフォーム上で実装しています。Apple Computer Inc.ではJava Shared Archive (JSA)として知られる、共有クラスと似た雰囲気を持つ技術をMac OS Xで実装しています。またSunがJ2SE 1.5リリースで導入したClass Data Sharing (CDS)は、JSA技術に基づいています。では、これらそれぞれの実装がどのように動作するのかを見て行きましょう。

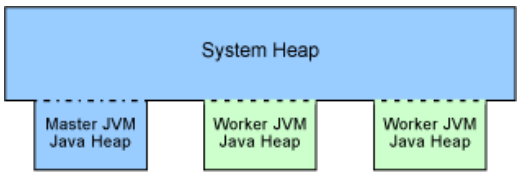
IBMの実装

IBMでの共有クラスの実装は、z/OSプラットフォーム上でのJ2SE 1.3.1から使用できるようになっています。この実装はプライマリ（またはマスター）JVMに対して、コア・システム・クラスを共有メモリにロードさせることで動作します。これは具体的にはどういう事なのでしょう。

分割ヒープ

メモリは共有ヒープとJavaヒープに分割されます。マスターJVMはシステム・ヒープ（つまり共有ヒープ）を共有メモリに割り当てますが、ここはシステム・クラスが置かれる所です。システム・ヒープはマスターJVMの存続期間存在し、ガーベジ・コレクション（GC）の対象にはなりません。それに続く（あるいはワーカーの）JVMは、それぞれこのシステム・ヒープに付加され、自分自身のJavaヒープ用に非共有メモリを割り当てます（図1）。このJavaヒープは通常のGCの対象になります。またこのJavaヒープには、インスタンス化される全オブジェクトと共に、各ワーカーJVMが実行するアプリケーション固有の非共有クラスがあります。

図1. 共有クラスの分割ヒープ



共有可能クラスローダー

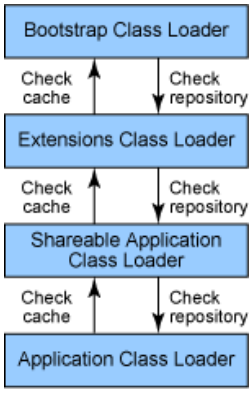
各ワーカーJVMは、ロードするクラスを共有可能クラスローダーのクラスパスに置くことで、共有ヒープにクラスをロードすることができます。共有クラスは通常のクラスと同じ、つまりparent-delegationモデルを使ってロードされます。

階層構造中にある各クラスローダーは、クラスが既にロードされているかどうかをチェックするために自分のキャッシュを調べます。まだロードされていなければ、そのクラスローダーは親のクラスローダーに対して、同じチェックするように要求を渡します。これをずっと上にまで上げて行き、階層構造の最上部にあるprimordialクラスやブートストラップ・クラスローダーにまで上げます。どのキャッシュにもクラスが見つからない場合には、各クラスローダーは自分のリポジトリからクラスをロードしようと試み、成功すればクラスを返します。ロードできない場合には、階層構造の下に向かって要求を戻します。このモデルを使うことで、最も信頼できるリポジトリが確実に最初にチェックされるようになります。また起こりうる間違い、つまり同じ名前をコアAPIの一部だと見なしてしまい、信頼できるコアAPIクラスを信頼度の劣るソースからのコードで置き換えてしまう、という間違いを防ぐことができます。

もしクラスがprimordialクラスの場合、または定義しているクラスローダーが共有可能クラスローダーの場合には、共有ヒープにクラス・オブジェクトが生成され、そのクラスは共有クラスとしてマーク付けされます。図2は、ブートストラップ・クラスローダーがクラスローダー階層構造の最上部に位置し、コアAPIからクラスをロードする責任がある様子を示しています。これらのクラ

スが最も信頼できるクラスです。拡張クラスローダーは拡張ディレクトリにある標準拡張JARファイルからクラスをロードします。共有可能アプリケーションのクラスローダーは、ユーザー・クラスやアプリケーション・クラスの共有に使用することができます。

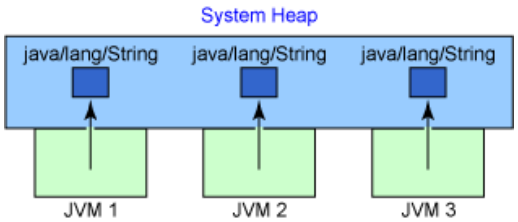
図2. クラスローダーの階層構造



ではこの実装では実際どのようにして、複数のJVMがクラスを共有するのでしょう？

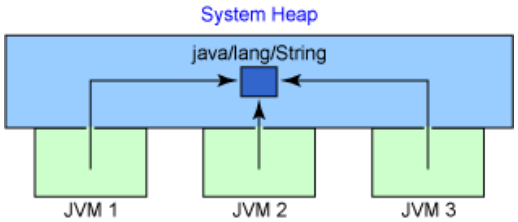
JVM 1がjava/lang/Stringをロードした場合を考えてください。java/lang/Stringはブートストラップ・クラスローダーがロードするシステムクラスです。もしJVM 2がjava/lang/Stringをロードしようとしても、JVM 1のブートストラップ・クラスローダーのキャッシュにはアクセスできないので、JVM 2は自分のブートストラップ・クラスローダーを使って再度初めからクラスをロードしなければなりません。この例では図3に示すように、（複数の）JVMはなにもクラスを共有してはいません。

図3. JVM間で共有されていないクラス



ですから図4に示すように、JVM同士で同じクラスを共有することが望ましいわけです。

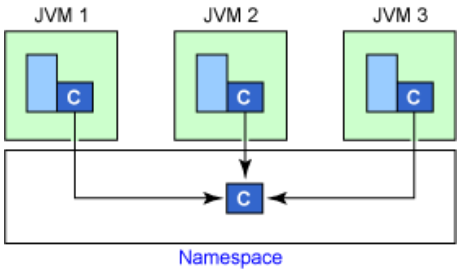
図4. JVM間で共有されているクラス



この問題を解決するには、名前空間と呼ばれるグローバル・クラス・キャッシュを作ることで、クラス・キャッシュの概念を拡張します。各JVMのクラスローダーは名前空間に登録する必要があります。共有クラスローダーがクラスをロードすると、そのクラスはローカルクラス・キャッシュと名前空間の両方に置かれます（図5）。こうすることで他のJVMにある（名前空間に登録さ

れた）クラスローダーは、そのクラスをロードすることなくそのクラスにアクセスすることができます。

図5. JVM間で共有される名前空間



保護ドメイン

クラスローダーには１つ以上のコードソース・オブジェクト（クラスのロード元のJARファイルまたはディレクトリ）があります。これらのオブジェクトは保護ドメイン(protection domains)を作るために使用するもので、`defineClass()`メソッド・コールに渡されます。共有クラスを使用している他のJVMがこの情報を必要としますが、保護ドメインはローカル情報を含んでいるので、共有することはできません。この問題を解決するために、コードソースをシステム・ヒープに置きます。またパッケージ情報も共有する必要があります。

レース条件

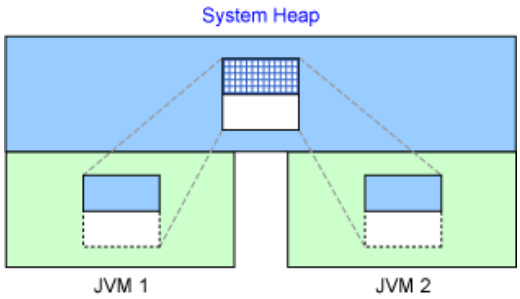
複数のJVMが共有データを読み書きするため、レース条件に対応する必要があります。最も単純な手法としてはグローバル・ロックを使うことですが、パフォーマンスやスケーラビリティを考えると、ごく稀にしか使うべきではありません。

すべてのJVMをロックしないようにする回避手段の一つは、楽観的アトミック更新（optimistic atomic updates）を使うことです。例えばクラスをロードする時に、クラスローダーは（自分のローカルクラス・キャッシュをチェックした後で）自分の名前空間をチェックします。そのクラスが見つからない時にはそのクラスをロードします。ロードが完了するとクラスローダーは、他のJVMがそのクラスをロードしていないことをアトミックにチェックしてから、名前空間を更新します。

グローバル・データ対ローカル・データ

クラス中の情報（例えば名前）の一部はすべてのJVMに渡って同じですが、そのクラスをロードしたクラスローダーなど他の情報はローカルである必要があります。各JVMはクラスの一部分に関するローカル・コピーを必要としますが、これはJVMがクラスをロードする時に生成します（図6）。この図で、JVMの中にあるクラスで影のついた部分はローカル・コピーです。クラスで影のついていない部分はグローバルな部分です。

図6. グローバル・データとローカル・データの共有



クラスを共有する時に時々発生する問題として、1つのJVMが（例えば静的フィールドの変更によって）あるクラスを更新すると、この変更が他のすべてのJVMに見えてしまうことがあります。これは望ましいものではなく、予期しない振る舞いを引き起こしかねません。分離を確実にするために各JVMは、各共有クラスの静的フィールドすべてのコピーを持つようにします。

JITでコンパイルしたコード

共有クラスに属するコードをjust-in-time (JIT)コンパイラーでコンパイルすると、そのコードは自動的に共有されます。これはつまり、どのJVMがコードをコンパイルしても、（JITのコストを負担するのは1つのJVMだけで）すべてのJVMがパフォーマンスの恩恵を受けられるということになります。

ランチャー・プログラム

IBMの実装では、ランチャーがJVMの生成を制御するように要求しています。ランチャーは、ユーザーがネイティブ・コードで書く必要があります。サンプルのランチャーに対する擬似コードをリスト1に示します。

リスト1. ランチャーの擬似コード

```
{ create Master JVM (and store returned token*);
    while(work to do) {
        create a Worker JVM passing in token from Master JVM;
        do work on Worker JVM;
        terminate Worker JVM;
    }
    terminate Master JVM;
}
* token returned from the Master JVM is the address of the shared heap.
```

Appleの実装

共有クラス技術のAppleでの実装が、Mac OS XのJava共有アーカイブ（Java shared archive: JSA）です。

基本的にJSAは共有メモリにメモリ・マップしたファイルであり、これを使うことによって、複数のプロセス（つまり複数のJVM）がそのファイルにアクセスできるようになります。Javaランタイム環境（Java Runtime Environment: JRE）がインストールされた後に、システムJARファイルにあるクラスからJSAが作られます。こうしたコア・クラスの内部表現はこのアーカイブに保存されます。このデータは静的なもので、従って変更されません。これは分離の問題無しにクラスを共有できるということになります。またこうしたクラスはガーベジ・コレクションを受ける必要もな

いことになります。Apple JVMは世代別GC（generational GC）を使っているので、こうしたクラスはガーベジ・コレクションから保護しておく必要があります。この保護はimmortalオブジェクトという概念を導入することで実現できます。またJSAにあるクラスは全てimmortalだと指定されています。

この技術はデフォルトで使用可能であり、IBMの実装で必要とされた、ランチャーのような特別なプログラムを書く必要もありません。もう一つの利点としては、システム・リセットでもこの機能が有効なことで、ある特定のJVMの存続期間に限定されていないことです。ただし欠点としてクラスのコア・セットに限定されており、アプリケーション・クラスやJITコンパイルされたコードを共有する機能はありません。

Sunの実装

Sunによる共有クラス技術の実装はClass Data Sharing (CDS)として知られており、J2SE 1.5の新しい機能の一つです。CDSはAppleのJSA実装に基づいています。

JSAと同様、CDSも読み取り専用の、メモリーマップしたアーカイブ・ファイルを使います。このファイルにはコア・システムクラスの内部表現を含んでおり、起動時に各JVMのJavaヒープにマップされます。CDSファイルはSun JREインストーラーが生成するか、SunのCDS資料に説明されている方法に従って手作業で作ります。

ここでもまた主な利点は、起動時のコストが節約できることと、メモリ要求が少ないということです。起動時間が短くて済むのは、コア・クラスのロードに伝統的なクラス・ローディング機構を使わないためです（つまりJVMは各クラスを個々にロードせず、すべてのコア・クラスを一度にマップするのです）。複数のJVMが読み取り専用のクラス・データを（それぞれ自分用のコピーを持つことなく）共有するので、使用するメモリの量は少なくなります。またクラスは伝統的な機構を使ってロードされるわけではないので、使われていないメソッドをJVMが処理することはありません。

この記事の執筆時点でSunの実装はコア・システムクラスの共有に限定されており、アプリケーション・レベルのクラスやJITコンパイルされたコードの共有はできません。そのアプリケーションが使うコア・クラスの数に比べてアプリケーションが小さければ小さいほど、起動期間の節約も大きくなります。

SunのCDS資料によると、より大きなアプリケーションの起動コストを改善するため、将来的には共有クラスの機能もアプリケーション・レベルのクラスに拡張される予定です。

共有クラス使用の現状

共有クラス技術はJavaプラットフォームのユーザーの間で既に利用されています。このセクションでは現在の展開状況の一端を説明します。

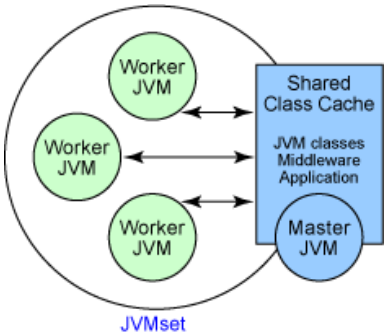
トランザクション環境

システムクラスの場合であれアプリケーション・クラスの場合であれ、CICSやDB2などのようなトランザクション環境（各トランザクションやアプリケーションが個々のJVMにラップされている）では、クラスの共有がJVMの起動とメモリ容量に大きな影響を与えます。

Customer Information Control System Transaction Server (CICS TS) for z/OS 2.3は、Javaトランザクションやアプリケーションを実行する主要な商業製品の一つで、現在はIBM JVMの共有クラス技術を利用しています。CICS TS 2.3は共有クラス・キャッシュ機能を導入しています。これは制御対象のJVMのプール（JVMsetとして知られる）にまで、この共有クラス機能を拡張するものです。

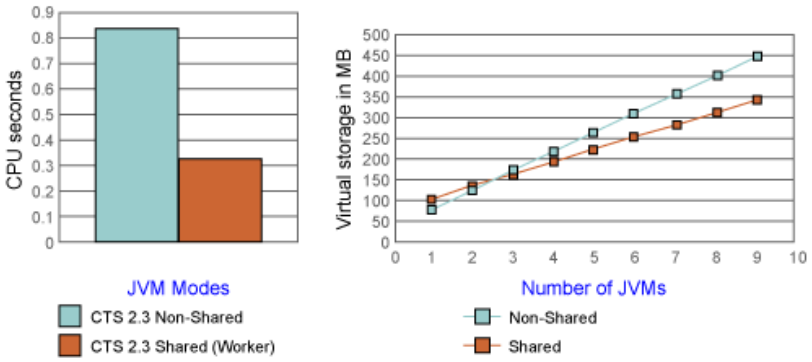
CICSはカスタムのランチャー・プログラムを使って、（Javaコンポーネントの実行要求のサービスに必要な）マスターJVMとワーカーの起動を制御します。CICSではシステム・ヒープでクラスを共有するのとは別に、図7に示すようにワーカーJVM間で特別な「ミドルウェア・クラス」や一部の「アプリケーション・クラス」を共有することもできます。この共有クラス・キャッシュ機能はCICSの顧客にとって多くの利点があります。例えばJavaクラスは、JVM毎に一度ではなくCICS領域毎に一度、CICSが起動した時にマスターJVMがロードし、（JVMsetの全ワーカーのためにクラスをローディングするという）オーバーヘッドを削減しています。共有クラス・キャッシュ機能はまた、各クラスのコピーを、各ワーカーJVMのヒープ毎に持つ代わりにキャッシュ領域に一つ持つことで、JVMsetに対する全体的なストレージ要求を削減しています。

図7. CICSでの共有クラス



大量のトランザクションに対応する時には、高速な起動と効率的なメモリ操作による恩恵が非常に大きくなります。図8の最初のグラフは、共有CICS環境と非共有CICS環境での、簡単なトランザクションの起動時間の比較です。２番目のグラフは同じく共有CICS環境と非共有CICS環境での、全体的なストレージ・コストの比較です。JVMの数が増えるに従って、JVM毎に必要なメモリ容量が（非共有の場合に比べて）減少していることが明らかに分かります。

図8.共有クラスによるCICSパフォーマンスの効果



デスクトップ・システム

Apple Mac OS Xプラットフォーム上で実行しているJavaアプリケーションであれば、自動的にこの技術の恩恵を受けることができます。またSun J2SE 1.5のユーザーも同様です。

将来的な共有クラスの使用可能性

共有クラス技術は、Javaプラットフォームに基づく他の技術の利用者にも、大きな恩恵をもたらす可能性があります。このセクションでは、今後可能性のあるアプリケーションの一部を紹介します。

パーベイシブ環境

Javaアプリケーションの実行に関してメモリ容量が少なくて済むということは、PDAや携帯電話のようなパーベイシブ環境に大きな影響を与えます。これまでは特別なVMやJITコンパイラー、J2MEにあるクラスなどで対応してきましたが、パーベイシブなデバイスが本当にどこにでもあるようになると、複数のJVMが必要となる可能性も高くなります。（コア・システムクラスを含む）メモリ・マップしたファイルを使うことで大幅なメモリの節約になり、これはパーベイシブ・デバイスにとって格段に有利になります。

グリッド・コンピューティング

グリッド・コンピューティング環境では、発生するコストは利用するCPU時間やメモリ容量のようなリソースによって決まりますが、多くの場合最適化が効果を発揮します。

共有クラスを使うことによって、グリッドのサプライヤーに対してより大きな容量をJavaプログラム実行用に与えられるので、顧客のジョブを同時に、より多く処理できるようになります。これはまた顧客にとって使用毎のコストが下がることにもなり、顧客がより高い価値を持つこと、サプライヤーの競争力が高まることを意味します。

Javaアプリケーション

IBM WebSphereのような複雑なJ2EEアプリケーションでは、潜在的には何千というクラスがロードされる可能性があります。コア・システムクラスやWebSphereクラスが（各JVMが使用前にそれぞれロードする代わりに）共有できれば、アプリケーションの起動時間は大幅に短縮できます。メモリ容量が少なくて済むことと併せて、WebSphereがホストする様々なアプリケーションにもこの利点が及ぶことになります。

Javaユーザーにとって、メモリを少なく保ったままアプリケーションを早く起動できるのは確かに有利です。もし（Eclipseのような）Javaアプリケーションを起動した時にコア・クラスが既にロードされていれば、こうしたアプリケーションはコア・クラスを共有でき、起動時に別々にコア・クラスをロードする必要が無くなります。SwingやAWTを使ったアプリケーションでは、起動が遅いとかメモリを使いすぎているといった不満が既に上がってきているので、この手法は特に有利になります。

まとめ

この記事ではJavaの共有クラス技術の全体的な概要を説明しました。またいくつか異なった種類の共有クラス技術を紹介し、それらがもたらす共通の利点、つまりJavaアプリケーションに対して起動時間が高速になる点や、メモリ容量が小さくてすむ点についても説明しました。さらに、共有クラスを利用した現在の技術と将来の技術についても調べてみました。

起動時間やメモリ容量が懸念材料となっているようなJavaアプリケーションは、どんなものでも共有クラス技術の恩恵を受けることができます。現状の実装では、どこでも使えるわけではないこと、あるいはアプリケーション・クラスは共有できない（またはその両方）ことなどから制限があります。こうした問題が解決されれば、より多くのユーザーがこの技術の恩恵を受けることができ、Javaアプリケーションがより魅力的なものとなるでしょう。

著者について

Lakshmi Shankar

Lakshmi Shankarは、イギリスのIBM Hursley Labsのソフトウェア技術者です。IBMで3年以上働いており、Javaパフォーマンスやテスト、開発など、Hursley Labsで幅広い経験を積んでいます。最近まで、IBM Java技術のクラス・ローディング・コンポーネントの所有者でした。現在は、情報管理チームの開発者の一員です。

Simon Burns

Simon Burnsは、IBM Hursley LabsのJava Technology Centreにおける、Persistant Reusable JVM のコンポーネント所有者であり、チームリーダーでした。JVM開発に3年以上従事しており、Persistant Reusable JVM技術とz/OSプラットフォームを専門にしています。またCICSとも緊密に作業しながら、この技術の活用のために協力してきました。オープンソースのEclipse Equinoxプロジェクト（現在はEclipse 3.1に統合されています）の一員として、OSGiフレームワークに従事した経験もあります。現在は、コンポーネント化に取り組んでいます。

Roshan Nichani

Roshan NichaniはIBM Hursley Labs でのJava Reflectionコンポーネントの所有者です。IBM Global ServicesやJava Technology Centerで働いてきており、幅広い経験を積んいます。彼もCICSでShiraz技術を利用するための作業に従事してきました。IBMでは４年以上働いています。