Eclipse OpenJ9 内のクラス共有

クラス共有機能を使用して、メモリー・フットプリントを削減し、起動時間を短縮する

Ben Corrie
Hang Shao

2018年 12月 27日

Java 仮想マシン (JVM) の重要なパフォーマンス・メトリックとしてはメモリー・フットプリントと起動時間の 2 つが挙げられますが、クラウド環境では特に、メモリー・フットプリントが非常に重要となります。クラウド環境内では、アプリケーションが使用した分のメモリーに対して料金が発生するためです。このチュートリアルで、Eclipse OpenJ9 内のクラス共有機能を使用して、メモリー・フットプリントを削減するとともに、JVM の起動時間を短縮する方法を学んでください。

Java 仮想マシン (JVM) の重要なパフォーマンス・メトリックとしてはメモリー・フットプリントと起動時間の 2 つが挙げられますが、クラウド環境では特に、メモリー・フットプリントが非常に重要となります。クラウド環境内では、アプリケーションが使用した分のメモリーに対して料金が発生するためです。そこで、このチュートリアルでは Eclipse OpenJ9 内のクラス共有機能を使用して、メモリー・フットプリントを削減するとともに、JVM の起動時間を短縮する方法を説明します。

Eclipse OpenJ9 とクラス共有

2017 年、IBM は J9 JVM をオープンソース化して Eclipse Foundation に寄贈しました。これによって誕生したのが、Eclipse OpenJ9 プロジェクトです。J9 JVM では Java 5 以降、10 年間にわたり、システム・クラスからアプリケーション・クラスまでのクラス共有をサポートしてきました。

OpenJ9 実装では、システム・クラス、アプリケーション・クラス、そして Ahead-of-Time (AOT) でコンパイルされたコードのすべてを、共有メモリー内の動的クラス・キャッシュに保管できます。このクラス共有機能は、OpenJ9 をサポートしているすべてのプラットフォーム上で実装されています。クラス共有機能では、ランタイム・バイトコード変更を統合することもできます (これについては、記事の後半で説明します)。

クラス共有はいったん有効にすれば後は考慮する必要のない機能でありながらも、メモリー・フットプリントを削減し、JVM 起動時間を短縮するための大きな余裕をもたらします。このことから、複数の JVM が同様のコードを実行する環境や、JVM が定期的に再起動される環境には、クラス共有が最適な機能となります。

クラス共有機能を使用すると、JVM とそのクラス・ローダー内でランタイム・クラスを共有できます。さらに、クラス共有のサポートをカスタム・クラス・ローダーに統合できるよう、パブリック Helper API も提供されています (詳細については、後で説明します)。

このチュートリアルの例に従って実際にコードを操作するとしたら、JDK と OpenJ9 を Adopt OpenJDK プロジェクトからダウンロードするか、またはこのリンク先の Docker イメージからプルしてください。

仕組み

まずは、クラス共有機能がどのように動作するかについて技術的な詳細を探りましょう。

クラス共有を有効化する

クラス共有を有効化するには、既存の Java コマンド・ラインに `-Xshareclasses[:name=<######>]` を追加します。JVM は起動時に、指定された名前の共有キャッシュを検索し (名前が指定されていない場合は、現在のユーザー名が使用されます)、既存の共有キャッシュに接続するか、必要に応じて新しい共有キャッシュを作成します。

共有キャッシュのサイズは、パラメーター `-Xscmx<###>[k|m|g` を使用して指定できます。このパラメーターは、新しい共有キャッシュを作成する場合にのみ適用されます。このオプションを省略した場合に使用されるデフォルト値は、プラットフォームによって異なります。オペレーティング・システムには、割り当て可能な共有メモリーの量を制限する設定があることに注意してください。例えば、Linux 上の `SHMMAX` は通常、約 32MB に設定されます。これらの設定の詳細については、このリンク先のユーザー・ガイドで、オペレーティング・システムごとの共有クラスに関する説明を参照してください。

共有クラス・キャッシュ

共有クラス・キャッシュは、固定サイズの共有メモリーの一部を占める領域です。非永続共有キャッシュを使用しない限り、JVM の存続期間が過ぎても、あるいはシステムがリブートされても、共有クラス・キャッシュは存続します。システム上に存在できる共有キャッシュの数に制限はありませんが、どの共有キャッシュにもオペレーティング・システムの設定と制約が適用されます。

共有キャッシュが JVM によって所有されることはありません。また、マスター/スレーブの JVM の概念も適用されないため、任意の数の JVM が共有キャッシュに対して同時に読み取り、書き込みを行うことができます。

共有キャッシュがサイズの点で増大することはありません。キャッシュがフルになると、JVM は引き続きキャッシュからクラスをロードすることはできても、データを保管することはできなくなります。そこで、サイズの大きい共有クラス・キャッシュを前もって作成し、使用可能な共有キャッシュの最大スペースに対してソフト・リミットを設定するという方法を使用できます。ソフト・リミットであれば、共有キャッシュに接続されている JVM をシャットダウンすることなく、必要に応じて値を大きくして共有キャッシュに保管するデータを増やすことができます。最大許容スペースのソフト・リミットについて詳しくは、このリンク先の OpenJ9 の資料を参照してください。

さらに、アクティブなキャッシュを管理するための JVM ユーティリティーもあります。これらのユーティリティーについては、「共有クラス・ユーティリティー」のセクションで説明します。

共有キャッシュは、JVM コマンド・ラインを使用して明示的に破棄した時点で削除されます。

クラスはどのようにキャッシュされるのか

JVM がクラスをロードするときには、最初にクラス・ローダー・キャッシュ内で、必要とするクラスがすでに存在しているかどうかを調べます。該当するクラスが見つかると、JVM はクラス・ローダー・キャッシュからそのクラスを返します。見つからない場合は、ファイル・システムから該当するクラスをロードし、`defineClass()` 呼び出しの一環として、そのクラスをキャッシュに書き込みます。したがって、非共有 JVM でのクラス・ローダー・ルックアップは次の順序で行われることになります。

1. クラス・ローダー・キャッシュ
2. 親
3. ファイル・システム

一方、クラス共有機能が有効化された JVM は、次の順序に従います。

1. クラス・ローダー・キャッシュ
2. 親
3. 共有クラス・キャッシュ
4. ファイル・システム

共有クラス・キャッシュに対するクラスの読み取りと書き込みは、パブリック Helper API を使用して行います。Helper API は `java.net.URLClassLoader` (および Java 9 以降では `jdk.internal.loader.BuiltinClassLoader`) に統合されています。したがって、`java.net.URLClassLoader` を継承するすべてのクラス・ローダーでは無償でクラス共有がサポートされます。カスタム・クラス・ローダーにクラス共有を実装する場合も、OpenJ9 には、そのための HelperAPI が用意されています。

何がキャッシュされるのか

共有クラス・キャッシュには、ブートストラップ・クラスおよびアプリケーション・クラス、クラスを記述するメタデータ、そして Ahead-of-Time (AOT) によってコンパイルされたコードを格納することができます。

OpenJ9 実装内で、Java クラスは以下の 2 つに分割されます。

- `ROMClass`。これは読み取り専用の部分であり、クラスの不変データのすべてが含まれます。
- `RAMClass`。静的クラス変数などの可変データが含まれる部分です。

`RAMClass` は `ROMClass` 内のデータを参照しますが、この 2 つは完全に分離されています。したがって、JVM 間でも、同じ JVM 内の `RAMClass` 間でも、`ROMClass` を安全に共有できます。

クラス共有機能が有効化されていない場合、JVM はクラスをロードするときに、`ROMClass` と `RAMClass` を別々に作成して、その両方をローカル・プロセス・メモリー内に保管します。クラス共有機能が有効化されていれば、JVM が共有クラス・キャッシュ内で `ROMClass` を検出した場合、

ローカル・メモリー内に `RAMClass` のみを作成するだけで済みます。作成された `RAMClass` は、共有されている `ROMClass` を参照します。

クラス・データの大部分は ROMClass 内に格納されることから、メモリーの節約対象となるのは、この部分です (詳細については、「メモリー・フットプリント」のセクションで説明します)。キャッシュにクラスが取り込まれていれば、JVM の起動時間も大幅に短縮されます。各共有クラスを定義するための処理の一部はすでに完了していることから、ファイル・システムからではなく、メモリーからクラスをロードできるためです。各クラスをそれが定義されている共有キャッシュに再配置するだけでよければ、起動時に新しい共有キャッシュを取り込むオーバーヘッドはそれほど大きくなりません。

AOT によってコンパイルされたコードも共有キャッシュ内に保管されます。共有クラス・キャッシュが有効化されていると、自動的に AOT コンパイラーがアクティブになります。AOT コンパイルを使用すると、Java クラスを、同じプログラムの後続実行用のネイティブ・コードにコンパイルすることができます。AOT コンパイラーはアプリケーションの実行中に動的にネイティブ・コードを生成し、生成した AOT コードのすべてを共有クラス・キャッシュに入れます。通常、AOT によってコンパイルされたコードの実行時間は、バイトコードを解釈して実行する場合よりも短いものの、JIT によってコンパイルされたコードほど高速ではありません。ただし、後続の JVM がメソッドを実行するときには、共有キャッシュから AOT コードをロードして使用できるため、JIT によってコンパイルしたコードを生成する際に伴うパフォーマンスの劣化は生じないため、結果的には起動時間が短縮されることになります。新しい共有キャッシュを作成する際は、`-Xscminaot<x>` および `-Xscmaxaot<x>` オプションを使用して、共有キャッシュ内の AOT スペースのサイズを設定できます。`-Xscminaot` も `-Xscmaxaot` も使用しなければ、共有キャッシュ内に利用できるスペースがある限り、AOT コードは共有キャッシュ内に保管されます。

ファイル・システム上でクラスが変更された場合はどうなるのか

共有クラス・キャッシュは永久に存続できますが、ファイル・システムの更新によって共有キャッシュ内のクラスと AOT コードが無効になる場合があります。クラス・ローダーが共有クラスを要求した後に返されるクラスは、ファイル・システムからロードされた時点でのクラスと常に同じであるとは限りません。このような事態はクラスをロードするときに透過的に発生します。そのため、常に正しいクラスがロードされるよう、共有クラス・キャッシュの存続期間中には、ユーザーが必要に応じていくつでもクラスを変更して更新できるようになっています。

クラスの変更に伴う落とし穴: 2 つの例

例えば、JVM によって クラス C1 が共有キャッシュに保管された後、JVM がシャットダウンし、C1 が変更されて再コンパイルされたとします。この場合、JVM は再起動する時点で、キャッシュされたバージョンの C1 をロードすべきではありません。

同様に、`/mystuff:/mystuff/myClasses.jar` というクラス・パスを使用して実行中の JVM が、`myClasses.jar` に含まれている C2 を共有キャッシュにロードするとします。その後、別の C2.class が `/myStuff` に追加され、別の JVM が起動して同じアプリケーションを実行します。この場合、その別の JVM が、キャッシュされたバージョンの `C2` をロードするのは誤りです。

JVM はファイル・システムの更新を検出するためにタイムスタンプ値を共有キャッシュに保管し、クラスのロード時には常に、キャッシュされている値と実際の値を比較します。この比較に

よって、JAR ファイルが更新されていることを検出しても、JVM にはどのクラスが変更されたかどうかはわかりません。したがって、キャッシュ内のその JAR に含まれているクラスと AOT コードのすべてが古いものとしてマークされ、キャッシュからロードできないようにされます。ただし、当該 JAR に含まれているクラスがファイル・システムからロードされてキャッシュに再度追加されるときには、変更されているクラスだけがそっくりそのまま追加されます。つまり、変更されていないクラスは実際上、古いものとして扱われません。

クラスを共有クラス・キャッシュからパージすることはできませんが、JVM は持っているスペースを最大限有効に利用しようとします。例えば、多数の異なる場所からクラスがロードされるとしても、同じクラスが 2 回追加されることはありません。したがって、3 つの異なる JVM が同じクラス C3 をそれぞれ /A.jar、/B.jar、/C.jar からロードする場合、クラス・データは一度しか追加されませんが、3 つのメタデータがクラスのロード元を記述します。

共有クラス・ユーティリティー

共有クラス・キャッシュを管理するために使用できる、さまざまなユーティリティーがあります。これらはすべて、`-Xshareclasses` のサブオプションです (`-Xshareclasses:help` を使用すると、すべてのサブオプションのリストを表示できます)。

これらのオプションを使用する方法について、例を用いて見ていきましょう。

まず、2 つの共有キャッシュを作成するために、異なるキャッシュ名を指定して `Hello` クラスを実行します (リスト 1 を参照)。

リスト 1. 2 つの共有キャッシュを作成する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -cp . -Xshareclasses:name=Cache1 Hello
Hello
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -cp . -Xshareclasses:name=Cache2 Hello
Hello
```

`listAllCaches` サブオプションを実行すると、システム上のすべてのキャッシュが一覧表示されて、それぞれのキャッシュが使用中であるかどうかが示されます (リスト 2 を参照)

リスト 2. すべての共有キャッシュを一覧表示する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:listAllCaches
Listing all caches in cacheDir C:\Users\Hang Shao\AppData\Local\javasharedresources\
Cache name              level         cache-type      feature         last detach time
Compatible shared caches
Cache1                  Java8 64-bit  persistent      cr              Mon Apr 23 15:48:12 2018
Cache2                  Java8 64-bit  persistent      cr              Mon Apr 23 15:49:46 2018
```

`printStats` オプションを実行すると、名前付きキャッシュのサマリー統計が出力されます (リスト 3 を参照)。printStats オプションの詳細については、このリンク先のユーザー・ガイドを参照してください。

リスト 3. 共有キャッシュのサマリー統計

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,printStats

Current statistics for cache "Cache1":

Cache created with:
```

```
        -Xnolinenumbers                     = false
        BCI Enabled                         = true
        Restrict Classpaths                 = false
        Feature                             = cr

Cache contains only classes with line numbers

base address                        = 0x000000001214C000
end address                         = 0x0000000013130000
allocation pointer                  = 0x0000000012297DB8

cache size                          = 16776608
softmx bytes                        = 16776608
free bytes                          = 13049592
ROMClass bytes                      = 1359288
AOT bytes                           = 72
Reserved space for AOT bytes        = -1
Maximum space for AOT bytes         = -1
JIT data bytes                      = 1056
Reserved space for JIT data bytes   = -1
Maximum space for JIT data bytes    = -1
Zip cache bytes                     = 902472
Data bytes                          = 114080
Metadata bytes                      = 18848
Metadata % used                     = 0%
Class debug area size               = 1331200
Class debug area used bytes         = 132152
Class debug area % used             = 9%

# ROMClasses                        = 461
# AOT Methods                       = 0
# Classpaths                        = 2
# URLs                              = 0
# Tokens                            = 0
# Zip caches                        = 5
# Stale classes                     = 0
% Stale classes                     = 0%

Cache is 22% full

Cache is accessible to current user = true
```

共有キャッシュ内の特定のデータを出力するために使用できる printStats サブオプションもあります。該当するサブオプションは、printStats=help を使用して確認できます。例えば、クラス・パス・データを確認するには、`printStats=classpath` を使用します。

リスト 4. 共有キャッシュのクラス・パスの内容を一覧表示する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,printStats=classpath

Current statistics for cache "Cache1":

1: 0x000000001360E3FC CLASSPATH
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\bin\compressedrefs\jclSC180\vm.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\se-service.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\rt.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\resources.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\jsse.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\charsets.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\jce.jar
        C:\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\tools.jar
1: 0x000000001360A144 CLASSPATH
        C:\OpenJ9
…
…
```

共有キャッシュを破棄するには、`destroy` オプションを使用します (リスト 5 を参照)。同様に、`destroyAll` を使用すると、使用中ではないキャッシュと、ユーザーが破棄する権限を持っているキャッシュのすべてが破棄されます。

リスト 5. キャッシュを破棄する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,destroy
JVMSHRC806I Compressed references persistent shared cache "Cache1" has been destroyed. Use option -
Xnocompressedrefs if you want to destroy a non-compressed references cache.
```

`expire` オプション (リスト 6 を参照) はハウスキーピング・オプションです。このオプションをコマンド・ラインに追加して期間を指定すると、その期間、何にも接続されていないキャッシュを自動的に破棄することができます。リスト 6 では、1 週間 (10,080 分) 使用されていないキャッシュを探し、該当するキャッシュを破棄してから JVM を起動します。

`reset` オプションは常に新しい共有キャッシュを作成します。同じ名前を持つキャッシュがすでに存在する場合、そのキャッシュは破棄されて、新しいキャッシュが作成されます。

リスト 6. 1 週間使用されていないキャッシュを破棄する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,expire=10080 Hello
Hello
```

verbose オプション

各種の verbose オプションを使用して、クラス共有が行っていることに関する有用なフィードバックを入手できます。これらのオプションはすべて、`-Xshareclasses` のサブオプションです。このセクションでは、さまざまな verbose オプションの使用例を記載します。

`verbose` オプション (リスト 7 を参照) は、JVM の起動とシャットダウンに関する簡潔なステータス情報を表示します。

リスト 7. JVM ステータス情報を取得する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,verbose Hello
[-Xshareclasses persistent cache enabled]
[-Xshareclasses verbose output enabled]
JVMSHRC236I Created shared classes persistent cache Cache1
JVMSHRC246I Attached shared classes persistent cache Cache1
JVMSHRC765I Memory page protection on runtime data, string read-write data and partially filled pages is
 successfully enabled
Hello
JVMSHRC168I Total shared class bytes read=11088. Total bytes stored=2416962
JVMSHRC818I Total unstored bytes due to the setting of shared cache soft max is 0. Unstored AOT bytes due to
 the setting of -Xscmaxaot is 0. Unstored JIT bytes due to the setting of -Xscmaxjitdata is 0.
```

`verboseIO` オプションは、共有キャッシュに対するクラス・ロード要求ごとに 1 行のステータス情報を出力します。`verboseIO` の出力を理解するには、クラス・ローダー階層を理解する必要があります。ブートストラップ・クラス・ローダー以外のクラス・ローダーによってロードされるクラスには、明らかにこの階層が表れるためです。出力内では、各クラス・ローダーに固有 ID が割り当てられますが、ブートストラップ・ローダーの ID は常に 0 となります。

verboseIO を使用する場合、すでにキャッシュされているクラスであっても、ディスクからロードされてキャッシュに保管されたとして示されることがありますが、これは正常な動作です。例えば、アプリケーションのクラス・パス上の各 JAR からロードされる最初のクラスは、それがキャッシュ内にあるかどうかにかかわらず、常にディスクからロードされて保管されます。なぜこのようにするのかというと、クラス・パスに含まれている JAR がファイル・システム上に存在することを確認するためです。

リスト 8 では、最初のセクションにキャッシュへのクラスの取り込み、2 番目のセクションに、キャッシュに入れられたクラスの読み取りが示されています。

リスト 8. verboseIO を使用する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,verboseIO Hello
[-Xshareclasses verbose I/O output enabled]
Failed to find class java/lang/Object in shared cache for class-loader id 0.
Stored class java/lang/Object in shared cache for class-loader id 0 with URL C:\OpenJ9\wa6480_openj9\j2sdk-
image\jre\lib\rt.jar (index 2).
Failed to find class java/lang/J9VMInternals in shared cache for class-loader id 0.
Stored class java/lang/J9VMInternals in shared cache for class-loader id 0 with URL C:
\OpenJ9\wa6480_openj9\j2sdk-image\jre\lib\rt.jar (index 2).
Failed to find class com/ibm/oti/vm/VM in shared cache for class-loader id 0.
Stored class com/ibm/oti/vm/VM in shared cache for class-loader id 0 with URL C:\OpenJ9\wa6480_openj9\j2sdk-
image\jre\lib\rt.jar (index 2).
Failed to find class java/lang/J9VMInternals$ClassInitializationLock in shared cache for class-loader id 0.
…
…
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,verboseIO Hello
[-Xshareclasses verbose I/O output enabled]
Found class java/lang/Object in shared cache for class-loader id 0.
Found class java/lang/J9VMInternals in shared cache for class-loader id 0.
Found class com/ibm/oti/vm/VM in shared cache for class-loader id 0.
Found class java/lang/J9VMInternals$ClassInitializationLock in shared cache for class-loader id 0.
…
…
```

verboseHelper サブオプション (リスト 9 を参照) は、Helper API からのステータス出力を表示する拡張オプションです。Helper API を使用する開発者は、verboseHelper サブオプションを利用して、この API の動作状況を理解することができます。この出力についての詳細は、このリンク先の JVM 診断ガイドに記載されています。

リスト 9. Helper API からのステータス出力

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=Cache1,verboseHelper Hello
[-Xshareclasses Helper API verbose output enabled]
Info for SharedClassURLClasspathHelper id 1: Verbose output enabled for SharedClassURLClasspathHelper id 1
Info for SharedClassURLClasspathHelper id 1: Created SharedClassURLClasspathHelper with id 1
Info for SharedClassURLClasspathHelper id 2: Verbose output enabled for SharedClassURLClasspathHelper id 2
Info for SharedClassURLClasspathHelper id 2: Created SharedClassURLClasspathHelper with id 2
Info for SharedClassURLClasspathHelper id 1: There are no confirmed elements in the classpath. Returning
 null.
Info for SharedClassURLClasspathHelper id 2: There are no confirmed elements in the classpath. Returning
 null.
Info for SharedClassURLClasspathHelper id 2: setClasspath() updated classpath. No invalid URLs found
Info for SharedClassURLClasspathHelper id 2: Number of confirmed entries is now 1
Hello
```

verboseAOT および -Xjit:verbose サブオプション (リスト 10 を参照) を使用すると、共有ファイルに対する AOT ロードおよび保管に関する情報を取得できます。

リスト 10. AOT ロードおよび保管に関する詳細情報

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:name=demo,verboseAOT -Xjit:verbose -cp
 shcdemo.jar ClassLoadStress
…
+ (AOT cold) java/nio/Bits.makeChar(BB)C @ 0x00000000540049E0-0x0000000054004ABF OrdinaryMethod - Q_SZ=2
 Q_SZI=2 QW=6 j9m=0000000004A4B690 bcsz=12 GCR compThread=1 CpuLoad=298%(37%avg) JvmCpu=175%
Stored AOT code for ROMMethod 0x00000000123C2168 in shared cache.
…
+ (AOT load) java/lang/String.substring(II)Ljava/lang/String; @ 0x0000000054017728-0x00000000540179DD Q_SZ=0
 Q_SZI=0 QW=1 j9m=00000000049D9DF0 bcsz=100 compThread=0
Found AOT code for ROMMethod 0x0000000012375700 in shared cache.
…
```

ランタイム・バイトコード変更

ランタイム・バイトコード変更は、動作を Java クラスにインスツルメンテーションするためによく使われている方法です。これを実行するには、JVM Tools Interface (JVMTI) フックを使用します (詳細については、ここをクリックしてください)。あるいは別の方法として、クラスが定義される前に、クラス・バイトをクラス・ローダーで置き換えることもできます。ただしこの場合、クラス共有に余分な課題が生じることになります。それは、ある JVM によってキャッシュされたインスツルメンテーション済みのバイトコードを、その JVM とキャッシュを共有する別の JVM がロードしないようにする必要がある場合があるためです。

そうは言っても、OpenJ9 の共有クラス実装は動的であることから、それぞれに異なるタイプの変更を使用する複数の JVM でも、安全に同じキャッシュを共有することができます。実のところ、変換を行う必要があるのは一度だけなので、バイトコード変更にコストがかかるとしても、変更されたクラスをキャッシュすることにはさらに大きなメリットがあります。唯一の条件は、バイトコード変更に決定性と予測可能性を持たせることです。クラスが変更されてキャッシュされた後は、クラスをさらに変更することは不可能になります。

変更されたバイトコードを共有するには、`-Xshareclasses` の `modified=<context>` サブオプションを使用します。ここで、context はユーザー定義の名前です。この名前を付けて作成される、共有キャッシュ内の論理パーティションに、この JVM によってロードされるすべてのクラスが保管されます。その特定の変更を使用するすべての JVM は、同じ変更コンテキスト名を使用して、同じ共有キャッシュ・パーティションからクラスをロードする必要があります。`modified` サブオプションを使用せずに同じ共有キャッシュを使用する JVM は、通常のクラスを見つけて保管することになります。

考えられる落とし穴

クラス・バイトを変更するために登録された JVMTI エージェントを使用して JVM を実行している一方、`modified` サブオプションが使用されていないとしても、他の通常の JVM、または他のエージェントを使用する JVM とのクラス共有は安全に管理されます。けれどもこの場合、追加のチェックが必要になることから、パフォーマンスがわずかに犠牲になります。したがって、効率化のためには、常に modified サブオプションを使用する必要があります。

modified サブオプションが使用されていなくてもクラス共有を安全に管理できる理由は、JVM が JVMTI API の使用を手掛かりにバイトコード変更がいつ行われるのかを把握しているからに他なりません。再定義されて再変換されたクラスはキャッシュ内に保管されません。JVM が通常のクラ

ス・バイト・データを共有キャッシュ内に保管することによって、キャッシュからロードされるすべてのクラスに対して JVMTI `ClassFileLoadHook` イベントをトリガーすることが可能になります。したがって、カスタム・クラス・ローダーがクラスを定義する前に、JVMTI も `modified` サブオプションも使用せずにクラス・バイトを変更した場合、定義されるクラスは通常のクラスと見なされて、他の JVM によって誤ってロードされる可能性があります。

変更されたバイトコードの共有について詳しくは、このリンク先を参照してください。

Helper API の使用

OpenJ9 で Shared Classes Helper API を提供するようになったことから、開発者はクラス共有のサポートをカスタム・クラス・ローダーに統合できます。クラス共有のサポートを統合する必要があるのは、`java.net.URLClassLoader` を拡張していないクラス・ローダーのみです。java.net.URLClassLoader の拡張であるクラス・ローダーは、自動的にクラス共有のサポートを継承します。

Helper API に関する包括的なチュートリアルはこの記事の範囲外ですが、全体的な概要を説明します。詳細について知りたい場合は、GitHub 上にある Helper API 実装を調べてください。

Helper API: 要約

Helper API のクラスはすべて、`com.ibm.oti.shared` パッケージ内に集められています。クラスを共有しようとするクラス・ローダーのそれぞれが、`SharedClassHelperFactory` から `SharedClassHelper` オブジェクトを取得する必要があります。作成された `SharedClassHelper` は、それを要求したクラス・ローダーに属し、そのクラス・ローダーが定義するクラスだけを保管します。`SharedClassHelper` はクラス・ローダーに、共有キャッシュ内のクラスを検出して保管するために使用できる、単純な API を提供します。クラス・ローダーにガーベッジ・コレクションが適用される場合、そのクラス・ローダーの `SharedClassHelper` にもガーベッジ・コレクションが適用されます。

SharedClassHelperFactory の使用方法

`SharedClassHelperFactory` はシングルトンであり、静的メソッド `com.ibm.oti.shared.Shared.getSharedClassHelperFactory()` を使用して取得します。このメソッドは、JVM 内でクラス共有が有効化されている場合はファクトリーを返し、そうでなければ `null` を返します。

SharedClassHelpers の使用方法

ファクトリーから取得できる `SharedClassHelper` には、3 つのタイプがあり、それぞれ異なるタイプのクラス・ローダーで使用するように意図されています。

- `SharedClassURLClasspathHelper`: このヘルパーは、URL クラス・パスの概念を使用するクラス・ローダーで使用するように意図されています。この場合、URL クラス・パス配列を使用して、共有キャッシュ内のクラスを保管および検出します。クラス・パスに含まれる URL リソースがファイル・システム上でアクセス可能になっていなければ、クラスをキャッシュできません。このヘルパーを使用する場合、その存続期間中にクラス・パスを変更する方法に関しても制約が課せられます。

- `SharedClassURLHelper`: このヘルパーは、任意の URL からクラスをロードできるクラス・ローダーで使用するように意図されています。指定された URL リソースがファイル・システム上でアクセス可能になっていなければ、クラスをキャッシュできません。
- `SharedClassTokenHelper`: このヘルパーは共有クラス・キャッシュを効率的に単純なハッシュ・テーブルに変換します。共有キャッシュには意味のない文字列キー・トークンに対して、クラスが保管されます。このヘルパーにだけは、動的更新機能が備わっていません。それは、保管されるクラスには、ファイル・システムのコンテキストが関連付けられていないからです。

`SharedClassHelper` ごとに以下の 2 つの基本メソッドがあります。メソッドのパラメーターは、ヘルパーのタイプによって少々異なります。

- `byte[] findSharedClass(String classname...)`。クラス・ローダーがクラスの親 (存在する場合) を要求した後に呼び出すメソッドです。`findSharedClass()` から `null` が返されなかった場合、クラス・ローダーは返されたバイト配列に対して `defineClass()` を呼び出します。この関数は実際のクラス・バイトではなく、`defineClass()` 用の特殊な cookie を返すため、バイトをインスツルメンテーションすることはできません。
- `boolean storeSharedClass(Class clazz...)`。クラスが定義された直後に呼び出すメソッドです。このメソッドは、クラスが正常に保管された場合は true、そうでない場合は `false` を返します。

その他の考慮事項

アプリケーションとともにクラス共有をデプロイする際は、セキュリティーやキャッシュの調整などの要素を考慮する必要があります。これらの考慮事項について、以下に簡単にまとめます。

セキュリティー

デフォルトでは、共有キャッシュはユーザー・レベルのセキュリティーを使用して作成されます。つまり、共有キャッシュにアクセスできるのは、その共有キャッシュを作成したユーザーに限られます。このことから、デフォルトのキャッシュ名はユーザーごとに異なるため、キャッシュ名の競合が発生することはありません。UNIX 上では、`groupAccess` を指定するサブオプションを使用して、キャッシュを作成したユーザーの 1 次グループに属するすべてのユーザーにキャッシュへのアクセスを可能にすることができます。

これに加え、`SecurityManager` がインストールされている場合、クラス・ローダーが共有できるクラスは、適切な権限が明示的に付与されているクラスに限られます。これらの権限を設定する方法について詳しくは、このリンク先のユーザー・ガイドを参照してください。

ガーベッジ・コレクションと Just-in-Time コンパイル

クラス共有を有効化して実行しても、クラスのガーベッジ・コレクション (GC) にはまったく影響がありません。クラス共有を有効化していない場合と同じく、クラスとクラス・ローダーはガーベッジ・コレクションの対象になります。また、クラス共有を使用する場合に適用される、GC モードや構成に対する制約もありません。

クラス・キャッシュ内に Just-in-Time (JIT) でコンパイルされたコードをキャッシュすることはできません。共有キャッシュ内の AOT コードにも JIT コンパイルが適用されます。このことは、メソッドに JIT が適用される方法とタイミングに影響を与えます。さらに、共有キャッシュには、JIT

のヒントとプロファイル・データを保管できます。こうした JIT データを保管する共有キャッシュ・スペースのサイズは、`-Xscmaxjitdata<x>` および `-Xscminjitdata<x>` オプションを使用して設定できます。

キャッシュ・サイズの制限

現在、理論上の最大キャッシュ・サイズは 2GB です。キャッシュ・サイズは、利用可能なシステム・メモリー、利用可能な仮想アドレス空間、利用可能なディスク・スペースなどの要素によって制限されます。詳細については、ここをクリックしてください。

例

実際的な例でクラス共有のメリットを明らかにするために、このセクションでは単純なグラフのデモを紹介します。ソースとバイナリーは GitHub 上に用意されています。

Java 8 上で稼働するこのデモ・アプリは、jre\lib ディレクトリーを検索して各 JAR を開き、検出したクラスごとに `Class.forName()` を呼び出します。これにより、約 16,000 個のクラスが JVM にロードされることになります。そして JVM がこれらのクラスをすべてロードするまでにかかった時間を、デモ・アプリがレポートします。やや不自然な例ですが、クラス共有のメリットを明らかにするには効果的なものです。アプリケーションを実行して、結果を確認しましょう。

クラスのロード・パフォーマンス

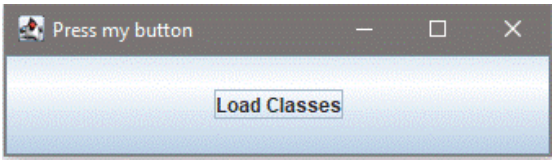
1. JDK と OpenJ9 を Adopt OpenJDK プロジェクトからダウンロードするか、Docker イメージからプルします。
2. GitHub から shcdemo.jar をダウンロードします。
3. システムのディスク・キャッシュをウォームアップするために、リスト 11 のコマンドを使用して、クラス共有を有効化せずにテストを 2、3 回実行します。

   リスト 11. ディスク・キャッシュをウォームアップする

   ```
   C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -Xshareclasses:none -cp shcdemo.jar ClassLoadStress
   ```

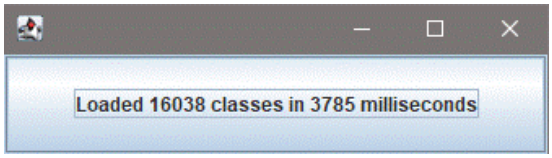
   図 1 に示すウィンドウが表示されたら、ボタンをクリックします。これにより、アプリがクラスをロードします。

   図 1. ボタンをクリックする

   クラスのロードが完了すると、アプリケーションがロードしたクラスの数とその所要時間をレポートします (図 2 を参照)。

図 2. 結果が表示されました！



アプリケーションを実行するたびに、ロードの所要時間がわずかに短縮されていくことに気付くかもしれません。この現象は、オペレーティング・システムの最適化によるものです。

4. 今度は、クラス共有を有効化してデモを実行します。リスト 12 に示されているように、新しい共有キャッシュが作成されます。すべてのクラスを保管するのに十分なスペースを確保するために、キャッシュ・サイズを約 50MB に指定できます。リスト 12 には、コマンド・ラインと出力例が示されています。

リスト 12. クラス共有を有効化してデモを実行する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -cp shcdemo.jar -Xshareclasses:name=demo,verbose -Xscmx50m
 ClassLoadStress
[-Xshareclasses persistent cache enabled]
[-Xshareclasses verbose output enabled]
JVMSHRC236I Created shared classes persistent cache demo
JVMSHRC246I Attached shared classes persistent cache demo
JVMSHRC765I Memory page protection on runtime data, string read-write data and partially filled pages
 is successfully enabled
JVMSHRC168I Total shared class bytes read=1111375. Total bytes stored=40947096
JVMSHRC818I Total unstored bytes due to the setting of shared cache soft max is 0. Unstored AOT bytes
 due to the setting of -Xscmaxaot is 0. Unstored JIT bytes due to the setting of -Xscmaxjitdata is 0.
```

キャッシュ統計に printStats が使用されていることを確認することもできます (リスト 13 を参照)。

リスト 13. キャッシュされたクラスの数を確認する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -cp shcdemo.jar -Xshareclasses:name=demo,printStats

Current statistics for cache "demo":

Cache created with:
        -Xnolinenumbers                     = false
        BCI Enabled                         = true
        Restrict Classpaths                 = false
        Feature                             = cr

Cache contains only classes with line numbers

base address                        = 0x0000000011F96000
end address                         = 0x0000000015140000
allocation pointer                  = 0x000000001403FF50

cache size                          = 52428192
softmx bytes                        = 52428192
free bytes                          = 10874992
ROMClass bytes                      = 34250576
AOT bytes                           = 1193452
Reserved space for AOT bytes        = -1
Maximum space for AOT bytes         = -1
JIT data bytes                      = 28208
Reserved space for JIT data bytes   = -1
Maximum space for JIT data bytes    = -1
```

```
Zip cache bytes                         = 902472
Data bytes                              = 351648
Metadata bytes                          = 661212
Metadata % used                         = 1%
Class debug area size                   = 4165632
Class debug area used bytes             = 3911176
Class debug area % used                 = 93%

# ROMClasses                            = 17062
# AOT Methods                           = 559
# Classpaths                            = 3
# URLs                                  = 0
# Tokens                                = 0
# Zip caches                            = 5
# Stale classes                         = 0
% Stale classes                         = 0%

Cache is 79% full

Cache is accessible to current user = true
```

5. 同じ Java コマンド・ラインを使用して、デモをもう一度開始します。以下の出力を見るとわかるように、今回は、共有クラス・キャッシュからクラスが読み取られています。

リスト 14. ウォームアップされた共有キャッシュを使用してアプリケーションを実行する

```
C:\OpenJ9>wa6480_openj9\j2sdk-image\bin\java -cp shcdemo.jar -Xshareclasses:name=demo,verbose -Xscmx50m
 ClassLoadStress
[-Xshareclasses persistent cache enabled]
[-Xshareclasses verbose output enabled]
JVMSHRC237I Opened shared classes persistent cache demo
JVMSHRC246I Attached shared classes persistent cache demo
JVMSHRC765I Memory page protection on runtime data, string read-write data and partially filled pages
 is successfully enabled
JVMSHRC168I Total shared class bytes read=36841382. Total bytes stored=50652
JVMSHRC818I Total unstored bytes due to the setting of shared cache soft max is 0. Unstored AOT bytes
 due to the setting of -Xscmaxaot is 0. Unstored JIT bytes due to the setting of -Xscmaxjitdata is 0.
```

明らかに、クラスのロード時間が大幅に (約 40%) 短縮されています。また、この場合もデモを実行するたびに、オペレーティング・システムの最適化によってパフォーマンスが少しずつ改善されていきます。

図 3. ウォームアップされたキャッシュでの結果

実験として、いくつかのバリエーションを加えることができます。例えば、javaw コマンドを使用して複数のデモを開始し、すべてのデモに同時にクラスをロードさせて、並列処理のパフォーマンスを確認するなどです。

実際のシナリオでは、クラス共有を使用することによって得られる全体的な JVM 起動時間のメリットは、アプリケーションがロードするクラスの数に依存します。例えば、HelloWorld プログラムではそれほど大きなメリットはありませんが、大規模な Web サーバーでは顕著なメリットが

表れるはずです。そうは言っても、この例から、クラス共有を実験してそのメリットをテストするのは簡単であることが明らかになったと思います。

メモリー・フットプリント

複数の JVM 内でサンプル・プログラムを実行すると、メモリーがどれだけ節約されるかも簡単に確認することができます。

以下に記載する 4 つの VMMap のスナップショットは、前の例と同じマシンを使用して取得したものです。図 4 には、クラス共有を有効化せずに、デモの 2 つのインスタンスを実行して完了するまでのメモリー使用量が示されています。図 5 には、前と同じコマンド・ラインを使用してクラス共有を有効化した状態で、2 つのインスタンスを実行して完了するまでのメモリー使用量が示されています。

図 4. クラス共有を有効化していない状態での 2 つのデモ・インスタンス

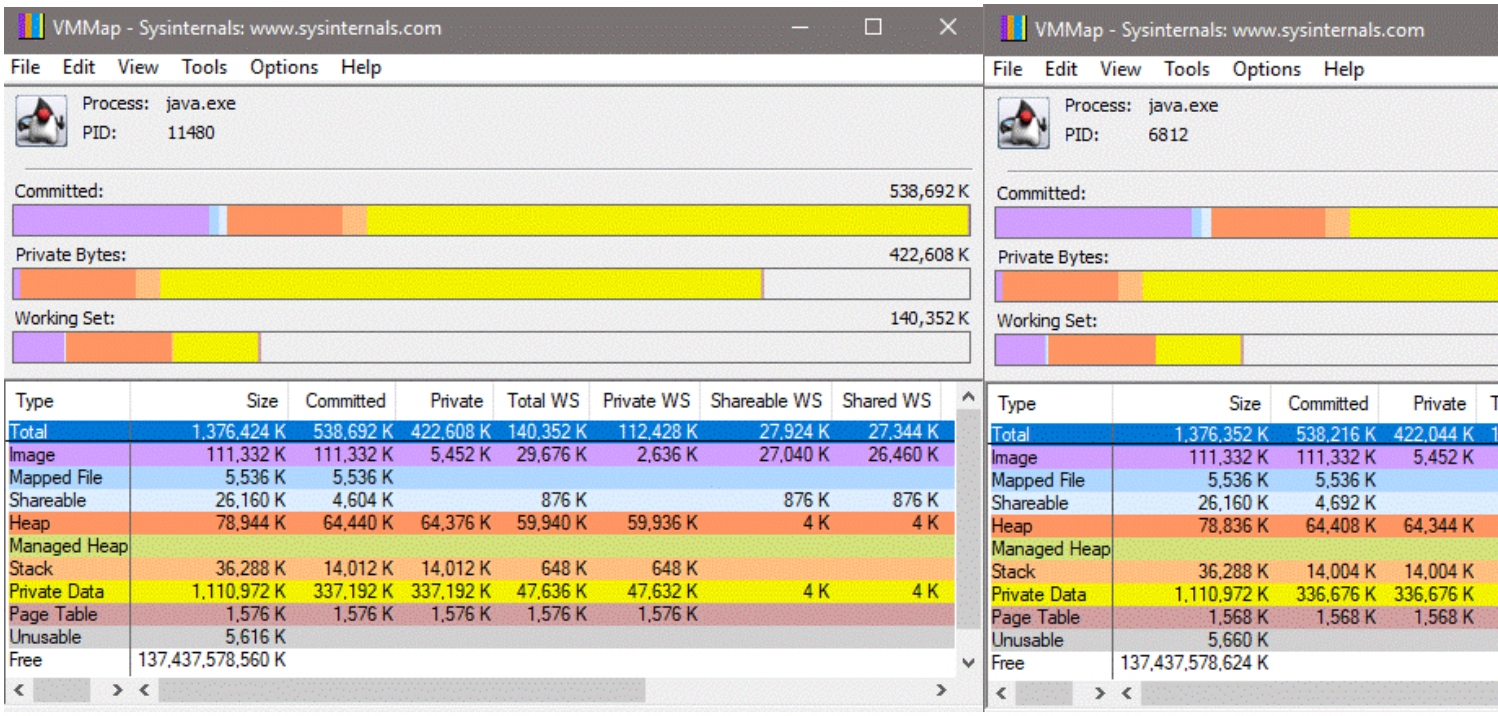
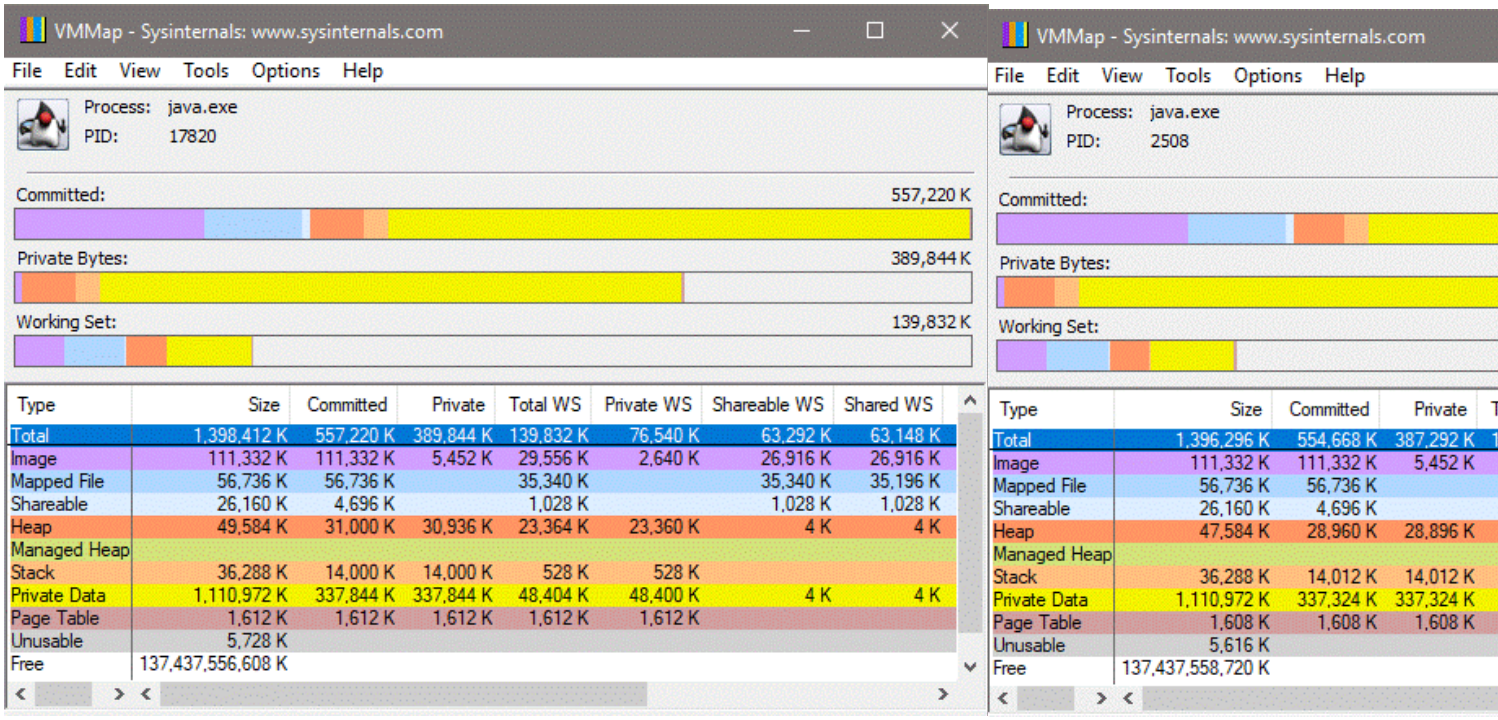


図 5. クラス共有を有効化した状態での 2 つのデモ・インスタンス



この実験では共有キャッシュ・サイズを 50MB に設定しているため、図 5 に示されている各インスタンスの「Mapped Files (マッピングされたファイル)」のサイズは、図 4 に比べて 50MB (56736KB – 5536KB) 大きくなっています。

クラス共有を有効化すると、メモリー使用量 (「Private WS (専用 WS)」) が大幅に少なくなることは明白です。2 つの JVM インスタンスに関して、約 70MB の専用 WS が節約されています。クラス共有を有効化した状態で、デモのインスタンス数を増やして起動すれば、さらにメモリー使用量が節約されることを観測できるでしょう。上記の結果は、32GB の RAM と Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz を搭載した Windows 10 ラップトップ上で取得したものです。

Linux x64 マシン上でも、同じメモリー・フットプリントの実験を行いました。リスト 15 はクラス共有を有効化していない場合の 2 つの JVM インスタンスの結果、リスト 16 はクラス共有を有効化した場合の 2 つの JVM インスタンスの結果を示しています。

これらの結果を見ると、クラス共有を有効化しても RSS に大幅な改善は表れていません。なぜなら、共有キャッシュ全体が RSS 内に含まれているからです。一方、各 JVM に対して共有キャッシュのサイズが半分だけ (2 つの JVM によって共有されているため) の PSSを見ると、約 34MB の節約になっています。

リスト 15. クラス共有が無効化された Linux 上のフットプリント

```
pmap -X 9612
9612:   xa6480_openj9/j2sdk-image/jre/bin/java -cp shcdemo.jar ClassLoadStress
Address Perm  …   Size    Rss     Pss Referenced Anonymous Swap Locked Mapping
…
                ======= ======= ===== ========   ========= ==== ====
                2676500 118280 106192 118280     95860     0    0 KB
pmap -X 9850
9850:   xa6480_openj9/j2sdk-image/jre/bin/java -cp shcdemo.jar ClassLoadStress
Address Perm  …   Size    Rss     Pss Referenced Anonymous Swap Locked Mapping
…
                ======= ======= ===== ========   ========= ==== ====
                2676500 124852 112792 124852     102448    0    0 KB
```

リスト 16. クラス共有が有効化された Linux 上のフットプリント

```
pmap -X 4501
4501:   xa6480_openj9/j2sdk-image/jre/bin/java -Xshareclasses:name=demo -Xscmx50m -cp shcdemo.jar
 ClassLoadStress
Address Perm  …   Size    Rss     Pss Referenced Anonymous Swap Locked Mapping
…
7fe7d0e00000 rw-s 4       4       2       4        0    0      0 C290M4F1A64P_demo_G35
7fe7d0e01000 r--s 33356   33356   16678   33356    0    0      0 C290M4F1A64P_demo_G35
7fe7d2e94000 rw-s 11096   48      24      48       0    0      0 C290M4F1A64P_demo_G35
7fe7d396a000 r--s 5376    1640    832     1640     0    0      0 C290M4F1A64P_demo_G35
7fe7d3eaa000 rw-s 296     0       0       0        0    0      0 C290M4F1A64P_demo_G35
7fe7d3ef4000 r--s 1072    0       0       0        0    0      0 C290M4F1A64P_demo_G35
…
                ======= ======= ===== ======== ====== ====== ====
                2732852 120656  90817 97988    62572  0      0 KB
pmap -X 4574
4574:   xa6480_openj9/j2sdk-image/jre/bin/java -Xshareclasses:name=demo -Xscmx50m -cp shcdemo.jar
 ClassLoadStress
Address Perm  …   Size    Rss     Pss Referenced Anonymous Swap Locked Mapping
…
7f308ce00000 rw-s 4       4       2       4        0    0      0 C290M4F1A64P_demo_G35
7f308ce01000 r--s 33356   33356   16678   33356    0    0      0 C290M4F1A64P_demo_G35
7f308ee94000 rw-s 11080   48      24      48       0    0      0 C290M4F1A64P_demo_G35
7f308f966000 r--s 5392    1632    824     1632     0    0      0 C290M4F1A64P_demo_G35
7f308feaa000 rw-s 296     0       0       0        0    0      0 C290M4F1A64P_demo_G35
7f308fef4000 r--s 1072    0       0       0        0    0      0 C290M4F1A64P_demo_G35
…
                ======= ======= ===== ======== ====== ====== ====
                2730800 122832  92911 102584   64812  0      0 KB
```

まとめ

メモリー・フットプリントを削減すると同時に JVM 起動時間を短縮する上で、OpenJ9 実装内のクラス共有機能は単純かつ柔軟な手段です。この記事では、この機能を有効化する方法、キャッシュ・ユーティリティーを使用する方法、そしてメリットを測定する定量化可能な方法を説明しました。