Java* によるインテル® Optane™ DC パーシステント・メモリーのサポート

この記事は、インテル® デベロッパー・ゾーンに公開されている「Java* Support for Intel® Optane™ DC Persistent Memory」の日本語参考訳です。

はじめに

この記事では、Java* アプリケーションが DRAM アクセスに加え、インテル® Optane™ DC パーシステント・メモリーを揮発性メモリーとして使用するさまざまなメカニズムについて説明します。Java* のメモリー管理とインテル® Optane™ DC パーシステント・メモリー・モジュールの OS サポートの簡単な紹介から始めて、Java* における利用メカニズムを説明します。

Java* のメモリー管理

Java* 仮想マシン (JVM*) は、ターゲット・プラットフォーム上で Java* プログラムを実行するマネージドランタイム環境を提供します。Java* プログラムは、プラットフォームに依存しないバイトコードにコンパイルされ、jar ファイルまたはモジュールに照合されて JVM* によりターゲット・プラットフォームで実行されます。

メモリー管理は JVM* で提供される重要な機能です。JVM* は、アプリケーションに代わって Java* ヒープと呼ばれるメモリーチャンクを管理します。アプリケーションによって生成されるすべての Java* オブジェクトは、新しいキーワードを使用して JVM* によって Java* ヒープ上に割り当てられます。Java* アプリケーションはまた、割り当てられたメモリーを解放する必要がありません。JVM* は、ガベージコレクションと呼ばれる処理によって、未使用のオブジェクトを自動的に判別して解放しメモリーを再利用します。

Java* ヒープのデフォルトサイズは JVM* の実装によって異なりますが、通常システムで利用可能なメモリー容量に応じて決定されます。また、JVM* のコマンドラインで Java* ヒープサイズを指定することもできます。

OpenJDK* は、Java* Platform Standard Edition (SE) ランタイムにおけるオープンソースの主要実装であり、業界で広く採用されています。Java* SE のリファレンス実装も OpenJDK* を基にしています。

インテル® Optane™ DC パーシステント・メモリー

インテル® Optane™ DC パーシステント・メモリーは、データセンターでの利用に特化して設計された新しい種類のメモリーおよびストレージ・テクノロジーです。この製品は、大容量、低価格、および永続性というこれまでにない組み合わせをもたらします。手ごろな価格でシステムメモリーの容量を拡張できるため、利用者はこの新しいメモリーを搭載するシステムを使用して、プロセッサーに近いメモリー階層に大量のデータを移動または保持し、システムストレージからデータを取り込むレイテンシーを最小限に抑えることで、ワークロードを最適化できます。

インテル® Optane™ DC パーシステント・メモリー・モジュールを搭載したインテル® プラットフォームでは、アプリケーションの要件に応じて異なるモードを構成できます。Memory モードでは、パーシステント・メモリー

はアドレス指定可能なシステムメモリー全体として見え、DRAM はキャッシュレイヤーとして機能します。データ配置をより細かく制御するには、アプリケーションが DRAM とパーシステント・メモリー・モジュールの両方をアドレス指定可能な App Direct モードでインテル® プラットフォームを構成します。インテル® パーシステント・メモリーの構成の詳細については、「次世代メモリーへの準備」の記事をご覧ください。

このドキュメントで示されるすべての利用例では、App Direct モードのインテル® Optane™ DC パーシステント・メモリー・モジュールを使用して、大容量/低コストの揮発性メモリーを提供します。

オペレーティング・システムによるサポート

Linux* や Windows* などのオペレーティング・システム (OS) は、ダイレクトアクセス (DAX) モードを使用する特殊なファイルシステムを介してパーシステント・メモリーを扱います。DAX モードでは、ファイルシステム (NTFS、EXT4、XFS など) はファイルシステムとしてパーシステント・メモリーをマウントできます。OS のページキャッシュをバイパスして、バイトアドレス指定可能なパーシステント・メモリー・モジュールを直接読み書きするアプリケーションに、インテル® Optane™ DC パーシステント・メモリー・モジュールへのダイレクトアクセスを可能にします。DAX マウントされたファイルシステムのマッピング (Linux* の mmap を使用) は、インテル® Optane™ DC パーシステント・メモリー・モジュールのページをユーザー空間に直接マップします。

DAX ファイルシステムとしてパーシステント・メモリー・モジュールをマウントする手順については、Linux* 向けの NDCTL Getting Started (英語) と pmem.io の Creating a Windows Development Environment (英語) で詳しく説明されています。

Java* における利用シナリオ

Java* アプリケーションでは、Memory または App Direct モードでインテル® Optane™ DC パーシステント・メモリーを使用できます。Memory モードを使用する場合、インテル® Optane™ DC パーシステント・メモリー・モジュールを DRAM がキャッシュとして機能するシステムのアドレス指定可能なメモリー全体として OS を設定するだけです。Memory モードを使用するため、Java* アプリケーションや JVM* の構成を変更する必要はありません。この構成では、すべての JVM*、JIT コンパイルされた Java* アプリケーション・コード、メタデータ、Java* スタック、および Java* ヒープはパーシステント・メモリーに配置されます。

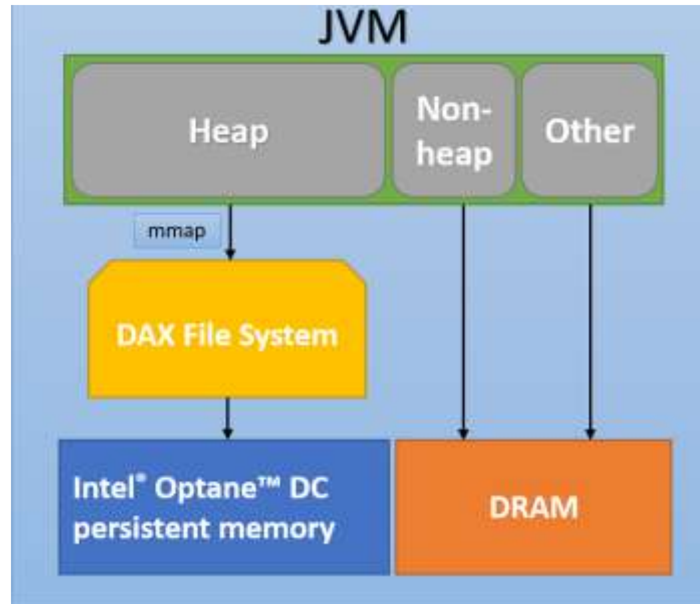
App Direct モードでは、DRAM とインテル® Optane™ DC パーシステント・メモリー・モジュールの両方がアドレス指定可能なメモリーとしてアプリケーションに見えるため、さらに細かな制御が可能です。このモードでは、JVM*、JIT コンパイルされた Java* アプリケーション・コード、スタック、およびメタデータを DRAM に保持し、Java* ヒープ全体、Java* ヒープの一部、または Java* オブジェクトを細かな粒度でパーシステント・メモリーに配置できます。

インテル® Optane™ DC パーシステント・メモリーのレイテンシーと帯域幅特性は DRAM とは異なり、最適な構成は Java* アプリケーションの特性とニーズに依存します。次のセクションでは、Java* アプリケーションを App Direct モードのさまざまな粒度レベルでパーシステント・メモリーを利用する方法を説明します。

ヒープ全体をパーシステント・メモリーに配置

JVM* は、新しいキーワードを使用してアプリケーションが作成したすべての Java* オブジェクトを Java* ヒープに割り当てます。インテル® Optane™ DC パーシステント・メモリーに Java* ヒープを配置するには、OpenJDK* に追加された新しい機能を使用する必要があります。この機能は、現在、JDK 11 (英語) の

OpenJDK* バイナリーの一部で利用できます。この機能を有効にするには、コマンドライン・フラグ **-XX:AllocateHeapAt** を使用します。このフラグは、パーシステント・メモリーにアプリケーションの Java* ヒープを割り当てるよう JVM* に指示します。フラグが指定されると、JVM* はパーシステント・メモリーがマウントされているパスを取得して、Java* ヒープのバッキングファイルとして使用する一時ファイルを作成します。次の図で具体的に示します。



Linux* での例を詳しく見てみましょう。

1. インテル® Optane™ DC パーシステント・メモリー・モジュールは、/mnt/pmem1 にマウントされています。

```
1. # sudo mount -v | grep /pmem
2. /dev/pmem1 on /mnt/pmem1 type ext4 (rw,relatime,dax)
3. 次に示すように、インテル® Optane™ DC パーシステント・メモリーに割り当てられた 128GB のヒープで MyApp を実行します。
4.
5. # java -version
6. openjdk version "11.0.2" 2019-01-15
7. OpenJDK Runtime Environment 18.9 (build 11.0.2+9)
8. OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode)
9.
10.    # java -Xmx128g -XX:AllocateHeapAt=/mnt/pmem1 MyApp &
11.    [1] 13068
```

2. Java* プロセスのプロセス・マップ・ファイルには、7ef394000000-7f1394000000 の 128GB のメモリー範囲がインテル® Optane™ DC パーシステント・メモリー・モジュールに割り当てられていることが示されています。

```
1. # cat /proc/13068/maps | grep heap
2. ...
3. 7ef394000000-7f1394000000 rw-s 00000000 103:01 11 /mnt/pmem1/jvmheap
4. ...
```

JIT コンパイルされた Java* アプリケーション・コード、スタック、メタデータなど、JVM* によって管理されるメモリーブロックは従来通り DRAM に割り当てられることに注意してください。このフラグは、JVM* のそれ以外

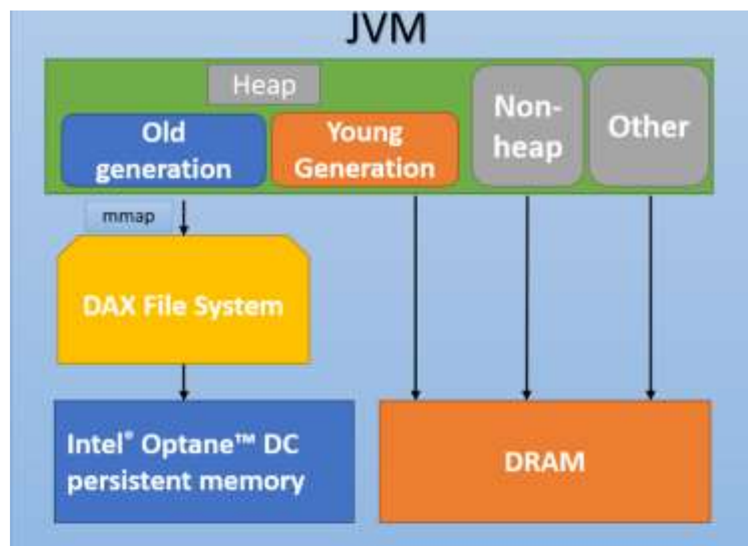
の動作に影響しません。これは、既存のヒープとガベージコレクション関連の **-Xmx**、**-Xms** などのフラグのセマンティクスと同じです。

ヒープの一部をパーシステント・メモリーに配置

Java* メモリー管理は、ヒープを 2 つに分割します (若い世代と古い世代)。新しい割り当ては若い世代で行われ、オブジェクトは数回のガベージコレクション (GC) サイクルの間若い世代に留まります。オブジェクトは数回の GC サイクル存続した後、古い世代に移動します。古い世代に移動すると、オブジェクトは到達不可になるまでそこに残り、GC によって収集されます。

アプリケーションごとに動作は異なりますが、通常、メモリーアクセスは古い世代に比べて若い世代で多くなります。また、若い世代は通常、合計ヒープのごく一部です。これは、アプリケーションがインテル® Optane™ DC パーシステント・メモリーに大きな古い世代を割り当て、DRAM に小さな若い世代を配置できることを意味します。

パーシステント・メモリーに古い世代の Java* ヒープのみを保持するには、OpenJDK* で提供される新しい試験的機能を使用する必要があります。この機能は、現在、JDK 12 (英語) 12 の OpenJDK* バイナリーの一部で利用できます。この機能を有効にするには、コマンドライン・フラグ **-XX:AllocateOldGenAt** を使用します。このフラグは、パーシステント・メモリーに古い世代の Java* ヒープを割り当てるよう JVM に指示します。若い世代は継続して DRAM に割り当てられます。次の図はマッピングの様子を示します。



Linux* での例を詳しく見てみましょう。

1. 前回同様、インテル® Optane™ DC パーシステント・メモリーは、/mnt/pmem1 にマウントされています。

```
1. # sudo mount -v | grep /pmem
2. /dev/pmem1 on /mnt/pmem1 type ext4 (rw,relatime,dax)
```

2. パーシステント・メモリーに配置された古い世代で MyApp クラスを実行します。

```
1. # java -version
2. openjdk version "12" 2019-03-19
3. OpenJDK Runtime Environment (build 12+33)
```

```
4. OpenJDK 64-Bit Server VM (build 12+33, mixed mode, sharing)
5.
6. # java -XX:+UseG1GC -Xmx128g -Xmn16g -XX:+UnlockExperimentalVMOptions
   -XX:AllocateOldGenAt=/mnt/pmem1 MyApp
```

このオプションは、G1 および ParallelOld ガベージコレクションのアルゴリズムでサポートされています。この機能は、GC の機能を維持しつつヒープ世代の動的なサイズ変更と連動します。機能の詳細は、リリースノート (英語) をご覧ください。

パーシステント・メモリー上の DirectByteBuffer

Java* アプリケーションのヒープ全体や部分的なヒープよりも細かな粒度でパーシステント・メモリーを使用する場合、DirectByteBuffer を使用してオブジェクトをインテル® Optane™ DC パーシステント・メモリーに割り当てできます。ByteBuffers (英語) は JDK 1.4 以降で利用できます。これは、次に示す 2 つのメカニズムによって実現されます。

FileChannel を使用した DirectByteBuffer

DirectByteBuffer は、ファイル領域をメモリーに直接マッピングすることで作成できます。**FileChannel** クラスで提供される機能を使用して、**DirectByteBuffer** をインテル® Optane™ DC パーシステント・メモリーにマップできます。

次のコードはこのアプローチを示します。

```
import java.nio.channels.FileChannel;
import java.nio.channels.FileChannel.MapMode;
import java.nio.ByteBuffer;
import java.nio.ByteBuffer;
import java.io.RandomAccessFile;

Class OffHeap {
  public static void main(String[] args) {
    // インテル(R) DCPMM は DAX ファイルシステムとして /mnt/pmem にマウントされます。
    // read/write 用にインテル(R) DCPMM 上のファイルを開きます。
    RandomAccessFile pmFile = new RandomAccessFile("/mnt/pmem/myfile", "rw");

    // 対応するファイルチャネルを取得します。
    FileChannel pmChannel = pmFile.getChannel();

    // DAX ファイル内の 1GB をメモリーにマップします。
    ByteBuffer pmembb = pmChannel.map(MapMode.READ_WRITE, 0, (1 << 30) - 1);


    // 書き込みを行います。
    for (int i = 0; i < (1<< 31)-1; i++) {
       pmembb.put(i, (byte)(i&0xff));

    // 読み取りと確認を行います。
    for (int i = 0; i < (1<< 31)-1; i++) {
       if (pmembb.get(i) != (byte)(i&0xff)) {
          System.out.println("Erorr at index:" + i);
       }
    }

    // ファイルサイズを 0 に切り捨てます。
    pmchannel.truncate(0);

    // ファイルチャネルを閉じます。
```

```
      pmchannel.close();

      // ファイルを閉じます。
      pmfile.close();
    }
}
```

上記の例では、**ByteBuffer** への **put()** と **get()** 呼び出しは、パーシステント・メモリーへの直接書き込みおよび読み取りを行います。モード **READ_WRITE** または **READ** を使用する必要があることを忘れないでください。**MapMode.PRIVATE** は、パーシステント・メモリーへの直接マップを提供しません。

Java* ネイティブ・インターフェイス (JNI) を使用する DirectByteBuffer

OpenJDK* は、JNI を介したネイティブコードからの **DirectByteBuffer** 作成もサポートします。パーシステント・メモリーに **DirectByteBuffer** を割り当てるのに JNI を使用できます。JNI コードは、Memkind[3] などのライブラリーを利用できます。Memkind は、インテル® Optane™ DC パーシステント・メモリーなどさまざまなメモリーを管理し、**memkind_alloc()** などの呼び出しにより異なるサイズのメモリーブロックを割り当てます。「Memkind を使用して大きな揮発性メモリー容量を管理する」(英語) 記事では、インテル® Optane™ DC パーシステント・メモリーで Memkind を使用する方法を説明しています。JNI には、**NewDirectByteBuffer**[4] と呼ばれるメカニズムがあり、Java* コードからアクセスするため、ネイティブに割り当てられたパーシステント・メモリーを **DirectByteBuffer** にラップできます。

まとめ

この記事では、Java* アプリケーションで App Direct モードでインテル® Optane™ DC パーシステント・メモリーを使用するいくつかの方法を検討しました。これらのメカニズムを使用して、さまざまなレベルの粒度でパーシステント・メモリー上のデータ配置を制御することができます。最適な構成は、アプリケーションの特性とニーズによって異なります。

著者紹介

Kishor Kharbas は、インテルに勤務するソフトウェア・エンジニアです。2011 年に入社し、インテル® サーバー・プラットフォーム向けの Java* 仮想マシンの最適化に取り組んできました。

Sandhya Viswanathan は、コンパイラーとソフトウェア開発ツールで 25 年以上の経験を持つインテルのソフトウェア・エンジニアです。2008 年に入社し、インテル® サーバー・プラットフォーム向けの Java* 仮想マシンの最適化に取り組んできました。彼女は、Java* JIT コンパイラー、ランタイム、GC 最適化に注目する Java* 開発チームのリーダーです。

参考資料

1. パーシステント・メモリーの Wiki (英語)
2. Windows Server* 2016 でバイトアドレス可能なストレージとして不揮発性メモリー (NVDIMM-N) を使用する (英語)
3. Memkind
4. NewDirectByteBuffer (英語)
5. Persistent Memory Documentation (英語)
6. ダイナミック・ランダムアクセス・メモリー (DRAM) を使用してパーシステント・メモリーをエミュレートする (英語)

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。