Java プロセスのメモリ使用量測定を踏まえた

コンテナ環境における Java オプション設計指針の紹介

2024年09月29日（日） オープンソースカンファレンス2024 広島
NTTコムウェア株式会社　坂本翔平、坂本統

心をつなぐ、未来をつくる

NTTコムウェア

スピーカー紹介

- NTT コムウェア株式会社
  - Java／OpenJDK 専門チーム
- 担当業務
  - **Java システムの技術サポートや新技術調査**
    - JVM プロファイリング＆トラブルシューティング
    - OpenJDK 仕様調査（ソースコード解析）
    - OpenJDK クラッシュ解析
    - etc.

坂本翔平
（テックリード）

坂本統
（テックリード／
OpenJDK Author）

本日のお話

NTTコムウェア

JVM のメモリ消費量を追跡する Native Memory Tracking（NMT） の使用方法と実際に測定した結果に基づくメモリ関連設計指針を紹介

- 目次
  - アプリケーションが利用するリソースを取り巻く変化
  - Java プロセスのメモリ消費内訳
  - NMT とは
  - NMT 実測例
  - Java プロセスのメモリ消費傾向
  - Java プロセスのメモリ消費見積もり方法
  - 見積もり例
  - 特別な見積もり要素

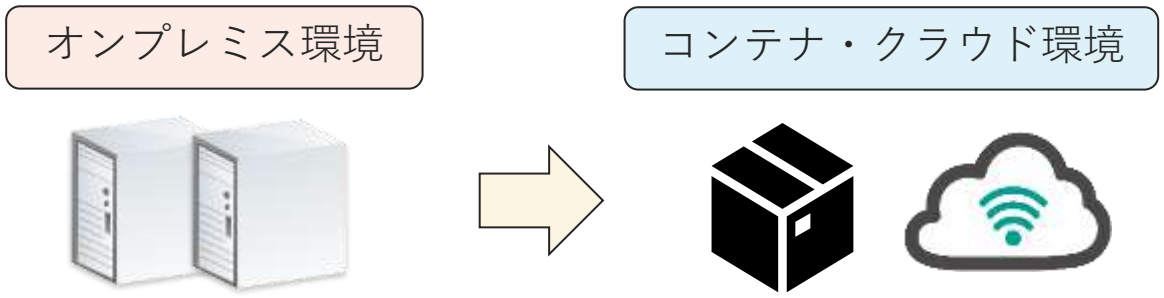
アプリケーションが利用するリソースを取り巻く変化

NTTコムウェア

近年ではコンテナやクラウド環境で Java アプリケーションを動作させるケースが増加

■ 近年のコンテナやクラウドの普及により **Java アプリケーションを動作させる基盤が変化**

- Docker、Podman、Kubernetes
- クラウド事業者が提供するパブリッククラウド
- etc.



コンテナ利用のメリット

- 可搬性の高さ
- 高速起動
- 開発や運用コストの低減

クラウド利用のメリット

- サーバ環境用意が不要
- 保守運用コストの低減
- 拡張性の高さ

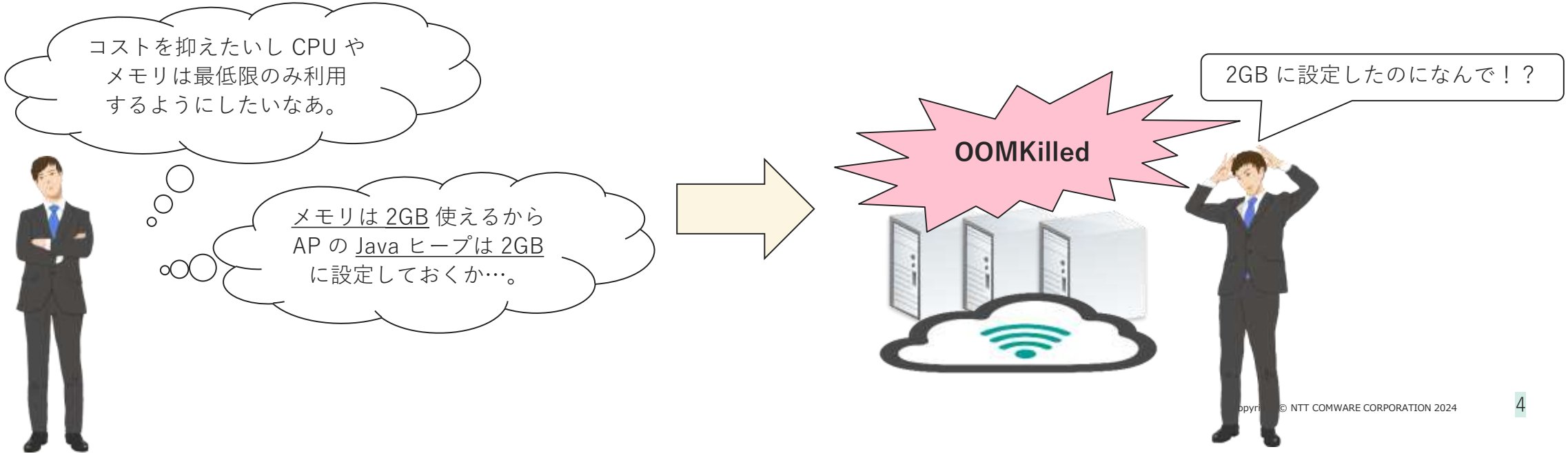
アプリケーションが利用するリソースを取り巻く変化

NTTコムウェア

Java アプリケーションのプロセスのメモリ消費内訳を 正しく把握し見積もることでOOMKilled を防止する必要がある

- コンテナやクラウド環境で動作するアプリケーションのリソース割当てはシビアになる
  - swap がないので厳密なメモリ管理が要求される
  - 従量課金のため必要な分のみリソースを利用する

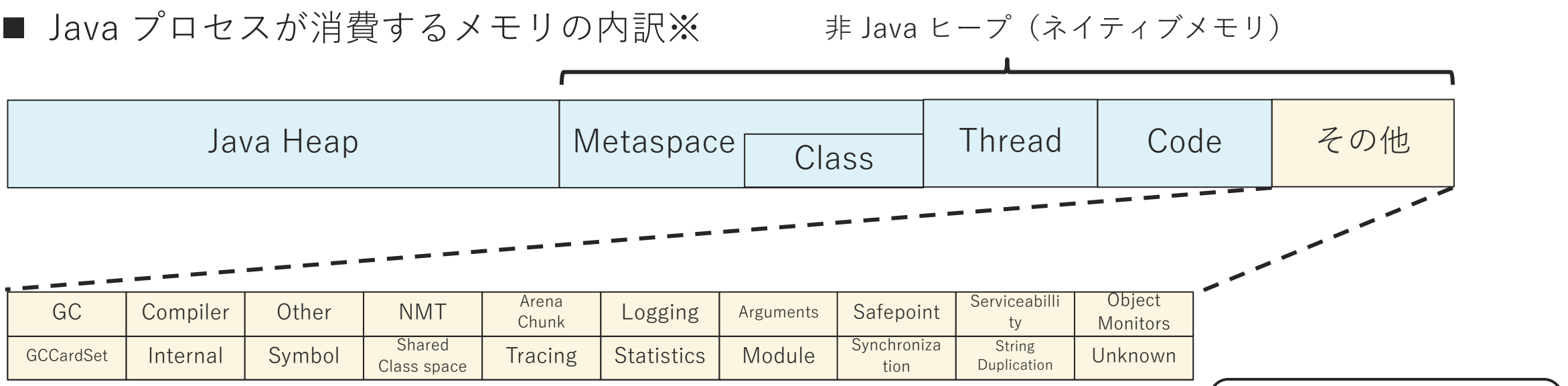
⇒ **OOMKilled を誘発してしまいがち**



Java プロセスのメモリ消費内訳

Java プロセスが消費するメモリ内訳は Java ヒープのみではないことに注意

■ Java プロセスが消費するメモリの内訳※



■ Java オプションでユーザが制御可能な領域

- Java ヒープ、Metaspace（& Compressed Class Space）、スレッドスタック、コードキャッシュ

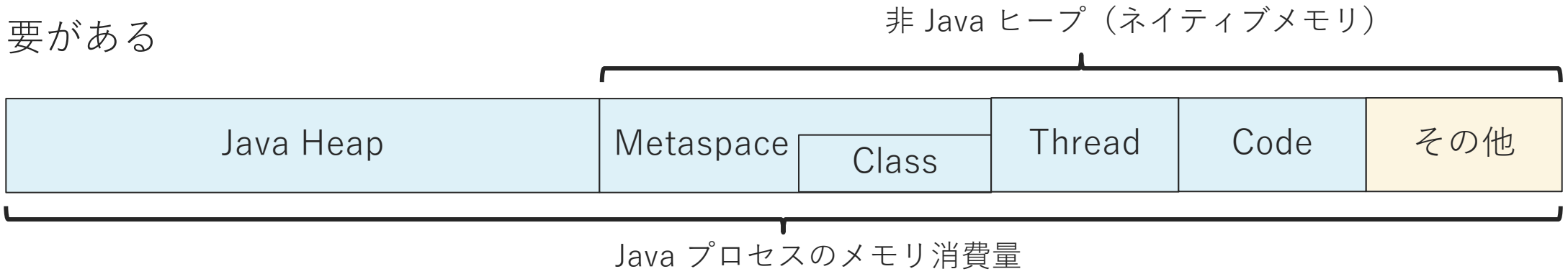
※ AP 実装による Direct Memory Buffer、JNI、FFM API は JVM 管理外の外部メモリを消費するため本日の内容の対象外とする

NMT による Java プロセスのメモリ消費内訳の測定

NTTコムウェア

Java プロセスのメモリ消費内訳の把握には Native Memory Tracking（NMT） を使用する

- Java プロセスのメモリ消費量見積もりには、ネイティブメモリ領域の消費も考慮する必要がある



- **Native Memory Tracking（NMT）**
  - Java プロセスの**ネイティブメモリも含めたメモリ消費状況を追跡**できる機能
    - ネイティブメモリは項目（カテゴリ）毎にメモリ消費量を追跡可能
  - OpenJDK の Java VM（Hotspot） の標準機能
  - デフォルトは無効化／有効化すると 5~10% の性能低下が発生

NMT の使い方（1）

NTTコムウェア

NMT は Java オプションで有効化し、
Java プロセスに対して jcmd コマンドで情報取得する

- NMT の有効化
  - Java オプション **-XX:NativeMemoryTracking** を使用

```
$ java -XX:NativeMemoryTracking=[ off | summary | detail ] <java_app_name>
```

NMT オプション	説明
off	無効化（デフォルト）
summary	サマリ情報を取得
detail	詳細情報を取得

- NMT による情報取得
  - jcmd の診断コマンド **VM.native_memory** で情報取得する

```
$ jcmd <pid> VM.native_memory <option> [ scale= KB | MB | GB ]
```

scale は NMT 情報の出力単位を指定
指定した単位未満のメモリ消費カテゴリは省略
（例：MB 指定ならば 1MB 未満は省略）

NMT オプション	説明
summary	サマリ情報を取得
detail	詳細情報を取得

NMT オプション	説明
baseline	比較元となるスナップショットを取得
summary.diff	ベースラインとの差分サマリ情報を取得
detail.diff	ベースラインとの差分詳細情報を取得

NMT の使い方（2）

NTTコムウェア

■ サマリ情報の出力例

```
Native Memory Tracking:

(Omitting categories weighting less than 1KB)

Total: reserved=18456134KB, committed=1166146KB
       malloc: 24618KB #4042
       mmap:   reserved=18431516KB, committed=1141528KB

-                 Java Heap (reserved=16392192KB, committed=1048576KB)
                            (mmap: reserved=16392192KB, committed=1048576KB)

-                     Class (reserved=1048648KB, committed=200KB)
                            (classes #489)
                            (  instance classes #404, array classes #85)
                            (malloc=72KB #533) (at peak)
                            (mmap: reserved=1048576KB, committed=128KB)
                            (  Metadata:   )
                            (    reserved=65536KB, committed=192KB)
                            (    used=74KB)
                            (    waste=118KB =61.26%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=128KB)
                            (    used=3KB)
                            (    waste=125KB =97.41%)
```

Java プロセス全体のメモリ消費量を出力

項目毎にメモリの reserved および committed を出力

Class ではロードしたクラス数も出力

参考：NMT detail（詳細情報）の出力内容

NTTコムウェア

■ 詳細情報の出力例

```
$ jcmd 192777 VM.native_memory detail
192777:

Native Memory Tracking:

(Omitting categories weighting less than 1KB)

Total: reserved=18456306KB, committed=1166322KB
       malloc: 24790KB #5737
       mmap:   reserved=18431516KB, committed=1141532KB
(省略)
Virtual memory map:

[0x0000000417800000 - 0x0000000800000000] reserved 16392192KB for Java Heap from
    [0x00007fdf41be368c] ReservedSpace::reserve(unsigned long, unsigned long, unsigned long, char*, bool)+0xbc
    [0x00007fdf41be4226] ReservedHeapSpace::try_reserve_range(char*, char*, unsigned long, char*, char*, unsigned long, unsigned
long, unsigned long)+0x146
    [0x00007fdf41be494d] ReservedHeapSpace::initialize_compressed_heap(unsigned long, unsigned long, unsigned long)+0x65d
    [0x00007fdf41be4b59] ReservedHeapSpace::ReservedHeapSpace(unsigned long, unsigned long, unsigned long, char const*)+0x149

        [0x0000000417800000 - 0x0000000456800000] committed 1
            [0x00007fdf4140012e] G1PageBasedVirtualSpace::co
            [0x00007fdf41414c21] G1RegionsLargerThanCommitSi
WorkGang*)+0x1a1
            [0x00007fdf414a93bc] HeapRegionManager::commit_regions(unsigned int, unsigned long, WorkGang*)+0x5c
            [0x00007fdf414ab256] HeapRegionManager::expand(unsigned int, unsigned int, WorkGang*)+0x36
...
```

detail では summary と同様のメモリ消費内訳に加えて、
仮想メモリマッピングの詳細が出力される

メモリの reserved や committed 量とあわせて、
メモリ予約・消費に使われた JVM 関数名も出力される

NMT の使い方（3）

NTTコムウェア

ベースラインを取得し、それと比較する形で情報出力することもできる

■ ベースラインとの比較

VM.native_memory baseline で
比較元となるベースラインを取得

VM.native_memory summary.diff で
サマリ情報と**ベースラインとの差分**も表示

差分は赤枠のように +**表記で表示**される

```
$ jcmd 125492 VM.native_memory baseline
125492:
Baseline succeeded

$ jcmd 125492 VM.native_memory summary.diff
125492:

Native Memory Tracking:

(Omitting categories weighting less than 1KB)

Total: reserved=18456129KB +1KB, committed=1166153KB +17KB
(省略)
-                     Thread (reserved=19554KB, committed=846KB +16KB)
                             (thread #0)
                             (stack: reserved=19504KB, committed=796KB +16KB)
                             (malloc=30KB #118)
                             (arena=20KB #36)
```

NMT の使い方（4）

アプリケーションの停止時に NMT 情報を出力することもできる

- アプリケーション停止時に NMT 情報を出力
  - **-XX:+UnlockDiagnosticVMOptions**
  - **-XX:+PrintNMTStatistics**

-XX:NativeMemoryTracking で NMT を有効化
していないと使用できない点に注意

```
 $ java -XX:NativeMemoryTracking=summary -XX:+UnlockDiagnosticVMOptions ¥
        -XX:+PrintNMTStatistics <java_app_name>
...(AP 停止後)
Native Memory Tracking:

Total: reserved=1661362387, committed=1195094227
       malloc: 26386643 #8545
       mmap:   reserved=1634975744, committed=1168707584

-                 Java Heap (reserved=1073741824, committed=1073741824)
                            (mmap: reserved=1073741824, committed=1073741824)
...
```

出力サイズの scale は指定できず、Byte 表記となる

NMT 実測例（JVM ベンチマークアプリケーションへの適用）

Renaissance Suite を活用して様々なアプリケーション特性のメモリ消費傾向を確認

Renaissance Suite(renaissance.dev) はビッグデータ、機械学習、関数型プログラミングなどの一般的な最新の JVM ワークロードを集約するオープンソースのベンチマークアプリケーション（.jar）

様々な AP 特性を実装するため幅広いメモリ消費傾向を測定できることを期待し活用

【参考】ベンチマークの種類

- (**apache-spark**) als, chi-square, dec-tree, gauss-mix, log-regression, movie-lens, naive-bayes, page-rank
- (**concurrency**) akka-uct, fj-kmeans, reactors
- (**database**) db-shootout, neo4j-analytics
- (**functional**) future-genetic, mnemonics, par-mnemonics, rx-scrabble, scrabble
- (**scala**) dotty, philosophers, scala-doku, scala-kmeans, scala-stm-bench7
- (**web**) finagle-chirper, finagle-http

【参考】使用例（Renaissance Suite が実装する finagle-chirper ベンチマークを繰り返し回数100&強制GCなしのオプションで実行）

```
$ java [JVM Options] -jar renaissance-gpl-0.15.0.jar --repetitions 100 --no-forced-gc finagle-chirper
finagle-http on :33531 spawning 12 client and default number of server workers.
====== finagle-http (web) [default], iteration 0 started ======
GC before operation: completed in 32.755 ms, heap usage 57.985 MB -> 11.894 MB.
====== finagle-http (web) [default], iteration 0 completed (12359.820 ms) ======
…略
```

※finagle-chirper: Twitter Finagle(RPC システム) を使用したマイクロブログサービスをシミュレートするベンチマーク

NMT 実測例（調査観点と検証条件）

カテゴリ単位のメモリ消費傾向の把握と全体消費量の計算方法の確立を目的に調査を実施

■調査観点

カテゴリ単位のメモリ消費傾向
　カテゴリごとのメモリ消費量の確認と、メモリ消費量の見積もりに影響があるカテゴリの特定

Java 起動オプションの影響
　Java ヒープ最大サイズや Metaspace/CCS 最大サイズの指定によるメモリ消費量の増加
　GC 方式の違いによる影響

OOMKilled 発生条件
　JVM チューニングの不備やイレギュラーな条件の特定

■検証条件

Java バージョン
　17/21
CPU&メモリ
　1CPU&メモリ2GB／2CPU&メモリ4GB／4CPU&メモリ8GB　※クラウドサービスを参考にした組み合わせ　※Kubernetes で環境構築（swap なし）
JVM オプション
　Java Heap最大サイズ、Metaspace & Compressed Class Space 最大サイズ、GC 方式（デフォルトのGC方式決定アルゴリズムに準拠）
Renaissance Suite ベンチマーク
　concurrency/database/functional/web グループの合計 12 種類

※本調査結果はこれらの条件のもと得た測定値に基づくものであり、どの環境・条件でも適用できるものではないことに注意
　あくまで設計段階の参考程度に留まるものであり、OOMKilled が発生しないかどうかは試験して試すこと

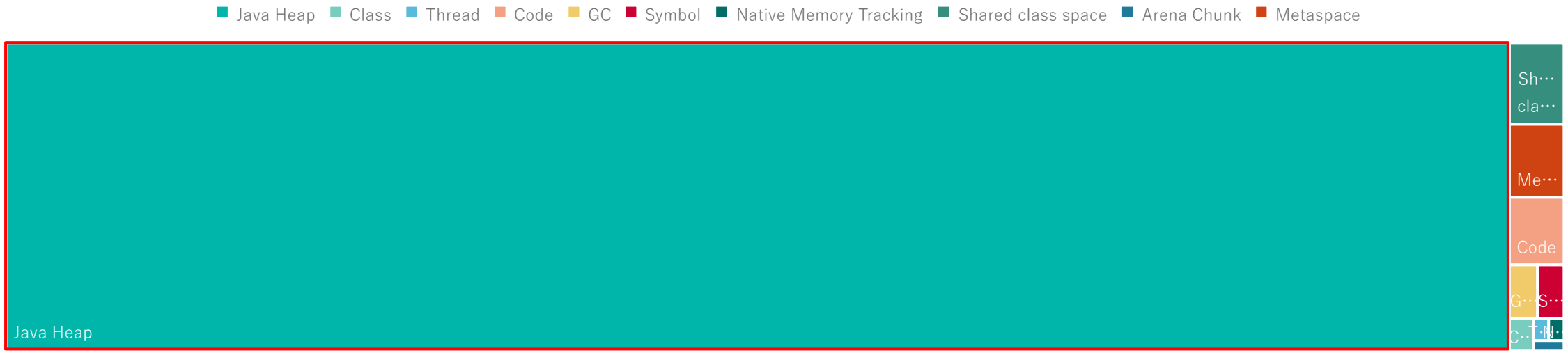
NTTコムウェア

Java Heap がメモリ消費の大部分を占める

起動時に最大サイズだけ reserved され、初期サイズ committed される

長時間稼働を続けると reserved = committed まで到達し、**最大 Java ヒープサイズに等しいメモリ消費**となる

カテゴリごとの相対的なメモリ消費量を示すツリーマップ（CPU 1/Memory 2GB/Serial GC/Java Heap 1.2GB）



※ 本図はカテゴリごとの committed の中央値をツリーマップ化した例で、メモリ消費が 1MB 未満のカテゴリは省略

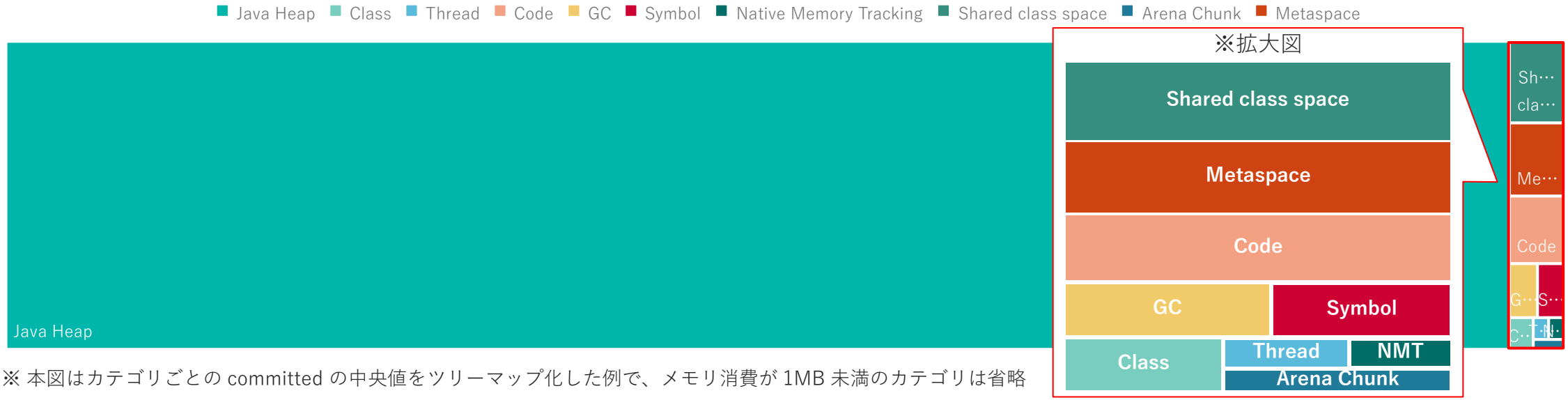
NTTコムウェア

Shared class space/Metaspace/Code/GC/Symbol も比較的メモリを消費

これらは数十〜数百 MB 程度のメモリを消費するため**メモリ消費量の見積もりにおいて考慮すべき**

さらにユーザ制御可能な **Class** とスレッド数やスタックの深さによって消費が増加する **Thread** も考慮すべき

カテゴリごとの相対的なメモリ消費量を示すツリーマップ（CPU 1/Memory 2GB/Serial GC/Java Heap 1.2GB）

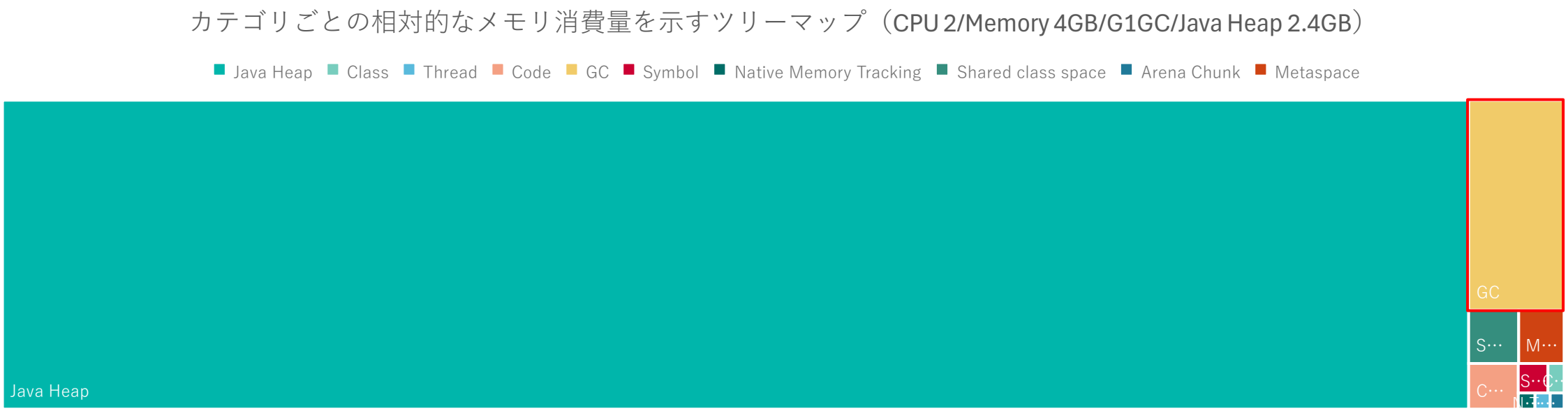


※ 本図はカテゴリごとの committed の中央値をツリーマップ化した例で、メモリ消費が 1MB 未満のカテゴリは省略

G1GC 方式は GC のメモリ消費が大きい

前述の結果は Serial GC を指定しているが、**G1GC を指定すると GC のメモリ消費が増加**

GC のメモリ消費量は **Java Heap 消費量の約 5% 程度**



（補足）JVM は起動時に利用可能な CPU 数を参照してデフォルトの GC 方式を決定する。Java 17/21 では CPU 数 1 以下の時 Serial GC、それ以外場合は G1GC を使用する。

NTTコムウェア

メモリカテゴリごとの最大メモリ消費量を積み上げた合計値を計算

カテゴリ	説明	関連 Java 起動オプション（デフォルト値）	メモリ消費傾向
Java Heap	Java ヒープ領域	最大 Java ヒープサイズ（メモリの 25%） **-Xmx** or **-XX:MaxRAMPercentage**	最大 Java ヒープサイズまで消費
Metaspace	Metaspace 領域	最大 Metaspace サイズ（無制限） **-XX:MaxMetaspaceSize**	測定結果は ~64MB or ~128MB程度
Class	Class Space 領域	（圧縮） Class Space サイズ（1GB） **-XX:CompressedClassSpace**	Metaspace の 25% 程度 ※見積もりでは考慮不要
GC	GC のメモリ	GC 方式の指定（） **-XX:+UseSerialGC** or **-XX:+UserG1GC**	Serial GC は 5MB 程度 G1GC は Java Heap の 5% 程度
Code	コード・キャッシュ	コード・キャッシュサイズ（256MB） -XX:ReservedCodeCache	コード・キャッシュサイズまで消費
Thread	スレッドのメモリ 主にスレッドスタック	１スレッドのスレッドスタックサイズ（1MB） -Xss or –XX:ThreadStackSize	スレッド数 x 1MB
Share class space	共有クラスのメモリ	なし	測定結果は 10MB 程度の消費（安定）
Symbol	シンボルのメモリ	なし	測定結果は ~40MB 程度の消費

Java 起動オプションで調整して見積

デフォルト値から見積

実測値から見積
※合計で50MB~程度

※デフォルト値は Linux 環境を想定

見積もり例

Java 起動オプション設定値と最大アプリケーションスレッド数から算出

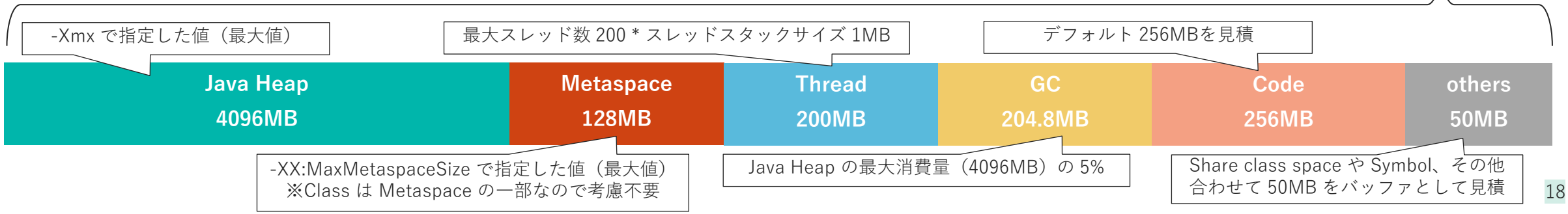
（例）オンプレ環境の Web アプリケーションの設定を変更せずにコンテナ・クラウドリフトする場合のメモリ割当の考え方

- 最大 Java ヒープサイズ **4GB**（-Xmx4g）
- 最大 Metaspace サイズ **128MB**（-XX:MaxMetaspaceSize=128m）
- **G1GC** 方式　※マルチコア環境のデフォルト
- AP サーバのスレッドプール数 **200**

メモリ消費量はカテゴリごとの最大消費量を積み上げて以下の様に見積もることができる

OOMKilled を避けるには **5GB** 以上のメモリを割り当てればよい

JavaHeap 4096MB + Metaspace 128MB + (thread 200 * ThreadStackSize 1MB) + (GC 4096MB * 0.05) + Code 256MB + others 50MB = **4934.8MB**



NTTコムウェア

フライトレコードおよびヒープダンプの出力を考慮する場合は追加でメモリ消費が発生

フライトレコードやヒープダンプの出力時は**ファイルサイズに依存したメモリ消費の急騰**が発生

OOME 時のヒープダンプ出力や AP 終了時のフライトレコード出力時の OOMKilled 発生に注意

これらを設定する場合は前述の見積もり計算に加え、追加の消費を考慮してメモリリソースの空きを用意すること

■ ヒープダンプ

ヒープダンプは Java ヒープ領域に残っているオブジェクト情報とそのシンボル情報を含む
ファイルサイズは最大 Java ヒープサイズ程度まで膨れる可能性があるため **Java Heap のメモリ消費見積もりを 2 倍**にする

■ フライトレコード

ファイルサイズは記録される JFR イベントに依存するため見積不可
代わりに -XX:StartFlightRecording の **maxsize サブオプションで最大ファイルサイズ制限**を推奨（指定値分見積に加算）
また **JFR 起動時には Tracing カテゴリのメモリ消費が発生するため、数百MB~の見積もりを加える**必要があることにも注意

まとめ

- 近年ではコンテナやクラウド環境で Java アプリケーションを動作させるケースが増加
- Java アプリケーションのプロセスのメモリ消費内訳を正しく把握し見積もることで OOMKilled を防止する必要がある
- Java プロセスが消費するメモリの内訳は Java ヒープのみではない

- Java プロセスのメモリ消費内訳の把握には Native Memory Tracking（NMT） を使用する
- NMT は Java オプションで有効化し、Java プロセスに対して jcmd コマンドで情報取得する
- NMT によって任意のタイミングやアプリケーション終了時のメモリ消費内訳を取得できる

- Java プロセスのメモリ消費傾向は Java Heap が消費の大部分を占め、次いで Metaspace や GC、Code、Symbol が消費量が高くなる
- Java プロセスのメモリ消費量を見積もりは以下の計算式を活用するとよい　※G1GC 方式の場合

```
MaxJavaHeapSize + MaxMetaspaceSize + (NumberOfThreads * 1MB) + (MaxJavaHeapSize * 0.05) + 256MB +　50MB
```

- フライトレコードおよびヒープダンプの出力時は瞬間的なメモリ消費の高騰に留意してメモリの空きを考慮する

Q&A