Java 標準ソート API における プリミティブ配列とオブジェクト配列の実行特性比較

深川 和葵 [1]　池田 孝利 [1]

概要：Java の標準ソート API では，int[]と Integer[]の間で実行時間の差が観測される．ただし，両者では内部のソートアルゴリズムも異なるため，この差をデータ表現だけに帰属することはできない．本研究では JDK 17 の Arrays.sort(int[])と Arrays.sort(Integer[])を対象に，JMH による定常状態および single-shot 計測，配列再設定のみを測る reset-only 対照，GC プロファイラ，ハードウェア性能カウンタを用いて実行特性を調べた．GC プロファイラ付きの計測では，100 万要素のランダム入力に対する配列再設定を含む測定区間の平均時間が int[]で約 72 ms，参照配列を複製する Integer[]条件で約 297 ms だった．性能カウンタ等の指標は予備的な取得段階にある．本発表では，標準 API 比較で観測できる範囲と，アルゴリズムとデータ表現の要因分離に向けて残る課題を整理する．

キーワード：Java，ソート，性能評価，JMH，ハードウェア性能カウンタ

Execution Characteristics of Primitive and Object Arrays in Java's Standard Sorting API

KAZUKI FUKAGAWA[1]　TAKATOSHI IKEDA[1]

1. はじめに

現代のソフトウェアシステムにおいて，データ処理の効率化は重要であり，その中でもソート処理は頻用される基本アルゴリズムの一つである．Java 等のオブジェクト指向言語では，プリミティブ型配列（int[]）とオブジェクト型配列（Integer[]）の双方が利用可能であるが，データ表現の違いが実行性能に与える影響の評価は多岐にわたる．著者らは先行研究[1]において，Python，C，Java の代表的なソート手段を同一条件で比較するベンチマークを構築し，Java の Arrays.sort(int[])と Arrays.sort(Integer[])の実行時間の比がデータサイズによって変化することを観測した．一方，プログラムの性能評価では，計測方法や実験設定の違いによって結論が変わりうることが指摘されている[2][3][4]．そこで本研究では，Java 標準ソート API における両者の挙動の違いに着目し，その実行特性を評価・分析する．

2. API と比較対象

JDK 17 の API 文書の実装注記によれば，Arrays.sort(int[])は Dual-Pivot Quicksort を，Integer[]を渡す Arrays.sort(Object[])は TimSort に由来する安定な適応型マージソートを用いる[5]．関連研究には，Java 7 版 Dual-Pivot Quicksort の平均計算量解析[6]と TimSort の最悪計算量解析[8]があり，TimSort の設計は CPython の解説文書[7]にも記載されている．これらはアルゴリズムの背景資料とし，本稿の対象実装の説明には JDK 17 の文書[5]を用いる．両 API の比較にはデータ表現の違いとアルゴリズムの違いが同時に含まれる．本研究では，プリミティブ型配列とオブジェクト型配列について，配列の準備，リセット処理（reset），ソート API の呼び出しを区別して測定条件を整理する．各条件の定義を表 1 に示す．

表 1　比較条件

条件	reset で行う操作	sort API
int[]	乱数配列を再コピー	sort(int[])
Integer[] rebox	int 値から Integer を再生成	sort(Object[])
Integer[] clone	箱詰め済みの配列を clone()	sort(Object[])

sort API はいずれも java.util.Arrays のメソッド．各条件について，reset＋sort と reset のみ（reset-only）を計測した．

3. 実験環境と測定方法

本研究では，測定ノイズの抑制を図ったベアメタル環境（ホスト名: fukagawa-sortperf-2026，CPU: Intel(R) Core(TM) i3-9300 CPU @ 3.70GHz，OS: Ubuntu 24.04.5 LTS）において計測を実施する．使用したランタイム環境は OpenJDK 17.0.20.1（OpenJDK 64-Bit Server VM）および JMH 1.37 [9]であり，JVM 設定は -Xms2g -Xmx2g -XX:+UseG1GC を適用した． 先行研究では JVM のウォームアップ挙動[10]と，VM 実行間・反復間の変動を考慮した計測設計[11]が議論されている．本研究ではウォームアップ後の計測と single-shot 計測を分け，複数の fork（JVM プロセス）で反復した．文献[10][11]は計測設計の背景として参照し，今回採用した反復回数の十分性を保証するものではない．JMH の測定モードおよび実行パラメータは実験内容に応じて設定しており，定常状態（steady）計測では fork 数 5，warmup 5 回×1 秒，measurement 10 回×1 秒を採用している．また，single-shot 計測（cold）では fork 数 20，warmup 0，measurement 1 回とし，GC プロファイリング計測では fork 数 5，warmup 5

1 大和大学 情報学部
Yamato University, Faculty of Informatics

ⓒ2026 Information Processing Society of Japan

回×1 秒，measurement 10 回×1 秒を設定して評価を行っている．各実行における入力データは固定の乱数 seed に基づいて生成され，データ manifest（18 ファイル）の SHA-256 検証により入力ファイルの同一性が確認されている（ただし，これは入力データの同一性を確認するものであり，実行時間そのものの完全な再現性を保証するものではない）．

表 2　実験環境と測定条件

項目	設定
CPU	Intel Core i3-9300 @ 3.70 GHz
OS	Ubuntu 24.04.5 LTS（ベアメタル）
JDK / JMH	OpenJDK 17.0.20.1 / JMH 1.37
JVM オプション	-Xms2g -Xmx2g -XX:+UseG1GC
steady	fork 5，warmup 5×1 s，measurement 10×1 s
cold (ss)	fork 20，warmup 0，measurement 1 回
GC profiling	fork 5，warmup 5×1 s，measurement 10×1 s

代表 run ID：
steady　20261020T191422_steady_21f9b1a-dirty_comsys_20261020
cold　20261020T185601_cold_21f9b1a-dirty_comsys_20261020
GC　20261020T185703_gc_21f9b1a-dirty_comsys_20261020

なお，本稿の概要に記載した実行時間（約 72 ms，約 297 ms）は 10/14 に実施した GC プロファイラ付き計測（fork 2, measurement 5）の値であり，第 4 章で示す値（73.75 ms/op, 295.3 ms/op）は 10/20 に実施した定常状態計測（fork 5, measurement 10）の値である．両者は計測の実施日および実験条件が異なる別の測定セッションに基づく実測値である．

4. 実験結果

4.1 実行時間評価（Steady State）

定常状態（steady state）における，要素数 n=1,000,000 のランダム入力パターンに対する各実装の実行時間を測定した．本計測における測定区間は配列のリセット処理とソート処理の合算（reset + sort）であり，得られた結果を図 1 に示す．

実験の結果，int[] によるプリミティブ配列のソート（int[] sort）の実行時間は 73.75 ms/op であった．これに対し，オブジェクト型配列である Integer[] を用いた場合，ボクシングを伴う再ボクシング実装（Integer[] sort (rebox)）では 329.7 ms/op，クローンを伴う実装（Integer[] sort (clone)）では 295.3 ms/op となり，プリミティブ型配列と比較して大幅な実行時間の増加が観測された．

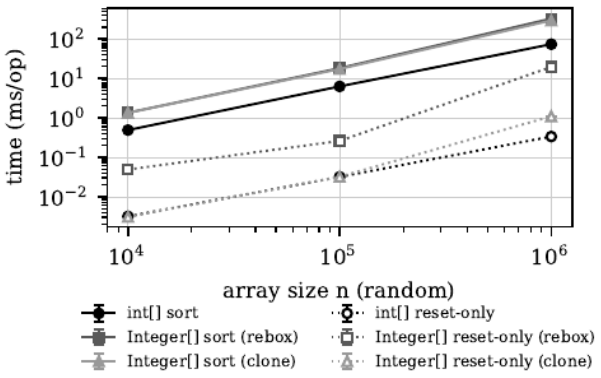

図 1　配列長別の実行時間

ランダム入力における配列長別の実行時間．実線は reset＋sort，点線は reset-only．縦軸は対数目盛（ms/op）．JMH によるウォームアップ後の計測．

採用 run ID：20261020T191422_steady_21f9b1a-dirty_comsys_20261020.

4.2 メモリ割当量評価（GC Profiling）

JMH のプロファイラ（-prof gc）を用いて，要素数 n=1,000,000 のランダム入力における 1 オペレーションあたりのメモリ割当量を評価し，その結果を図 2 に示す．なお，計測されたアロケーション量には配列のリセット処理分が含まれている点に留意する必要がある．

測定の結果，各条件におけるメモリ割当量は以下の通りとなった：

- int[] sort: 4.00 MB/op （実測値: 4,000,416 B/op）
- Integer[] sort (rebox): 24.10 MB/op （実測値: 24,096,298 B/op）
- Integer[] sort (clone): 8.10 MB/op （実測値: 8,098,276 B/op）

また，比較対照として測定したリセット処理単体（reset-only）の割当量は，それぞれ 4.00 MB/op（int[]），20.00 MB/op（Integer[] rebox），4.00 MB/op（Integer[] clone）であった．ただし，reset-only の割当量と reset＋sort の実行時間は測定区間が異なり，両者を用いて GC の寄与を判断することはできない．本稿では GC の発生回数・実行時間との対応を評価していないため，割当量や GC が実行時間差にどの程度寄与したかは特定できない．

ⓒ2026 Information Processing Society of Japan

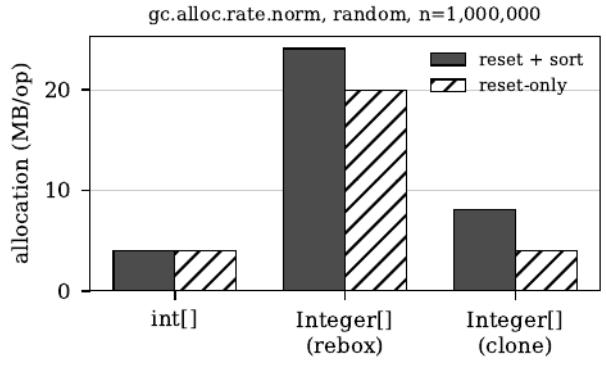

図 2　1 操作あたりのメモリ割当量
ランダム入力（n=1,000,000）の 1 操作あたりメモリ割当量（gc.alloc.rate.norm，MB/op）．reset＋sort と reset-only を比較する．reset＋sort の値には再設定の割当量を含む．
採用 run ID：20261020T185703_gc_21f9b1a-dirty_comsys_20261020.

4.3 その他の計測に関する補足（cold および perfnorm）

本検討では，定常状態（steady state）の評価に加え，ウォームアップを行わない single-shot 計測（cold）および，ハードウェア性能カウンタ等を用いた予備的な計測（perfnorm）の取得状況を確認している．まず，fork ごとに新規 JVM を起動してウォームアップなしで測定する cold 条件（run_cold.sh による SingleShotTime モード）は，定常状態の計測値とは異なる初期実行時の挙動を捉えるものである．また，10 月 20 日に取得した cold run（random，要素数 n=1,000,000）の実測値（20 fork の平均）では，int[] が約 130.3 ms/op，Integer[] (rebox) が約 461.0 ms/op，Integer[] (clone) が約 443.9 ms/op であり，ウォームアップなしの初期実行時における傾向を確認している．

これらは steady 条件の数値とは性質が異なるため，同一の傾向として混同せず別個に扱う．一方，perfnorm に関する計測データについては，現段階では予備的な取得・確認段階に留まっており，厳密な要因分離や性能差の直接的な原因説明の根拠としては使用しない．これらは今後の詳細なプロファイリングに向けた位置づけとする．

5. 考察と限界

5.1 観測結果の解釈

本稿で示したランダム入力（n=1,000,000）の結果では，Integer[] を用いたソート処理は，プリミティブ型配列である int[] と比較して実行時間およびメモリ割当量がともに増大することが確認された．特に，Integer[] sort (rebox) では約 4.47 倍（329.7 ms/op），Integer[] sort (clone) では約 4.00 倍（295.3 ms/op）の実行時間差が観測された．これに伴い，メモリ割当量についても Integer[] sort (rebox) では約 24.10 MB/op と，int[] sort（4.00 MB/op）に比べて大幅な増加が見られた．

本計測の時間と割当量には配列の再設定（reset）が含まれる．また，int[]と Integer[]では内部ソートアルゴリズムや要素へのアクセス方法も異なる．本実験ではこれらの条件を個別に統制していないため，観測された実行時間差をデータ表現，割当量，GC などの単一の要因に帰属することはできない．したがって，本結果は Java 標準 API をそのまま利用した場合の複合要因の比較（API-level comparison）として扱う．

5.2 本研究における測定の限界と今後の課題

本研究の評価においては，以下の限界が存在する．第一に，計測区間が配列のリセット処理とソート処理の合算（reset + sort）となっているため，純粋なソートアルゴリズム自体の実行時間を完全に分離して評価するには至っていない．リセット処理単体（reset-only）との差分を算出することは参考になるものの，これを厳密なソート本体の時間と断定することは避けるべきである．第二に，評価対象とした入力パターンやデータサイズ（n=1,000,000 を中心としたランダム入力等の傾向）は限定的であり，「すべてのサイズ帯や全パターンにおいて int[] が常に最速である」と一般化することはできない．今後は，アルゴリズムの差異を排除した統制比較（controlled comparison）の導入や，より詳細なプロファイリングを通じた要因の切り分けが不可欠である．

6. まとめ

本稿では，JDK 17 環境および JMH を用いたベンチマークにより，Java 標準ソート API におけるプリミティブ型配列（int[]）とオブジェクト型配列（Integer[]）の実行特性を評価した．定常状態（steady state，要素数 n=1,000,000，ランダム入力）における実測値として，int[] sort が 73.75 ms/op であるのに対し，Integer[] sort (rebox) は 329.7 ms/op，Integer[] sort (clone) は 295.3 ms/op となり，大きな性能差が観測された．しかし，本測定の区間には配列の初期化およびリセット処理が含まれており，アルゴリズムの差異（Dual-Pivot Quicksort vs TimSort）やメモリ割当量の影響を含め，純粋な要因分離には至っていない．今後は，統制比較（controlled comparison）の導入を通じて詳細な要因分析を進めていく予定である．

参考文献

[1] 深川和葵，池田孝利：データ表現とデータ特性に着目した多言語ソート性能比較ベンチマークの構築，ソフトウェアエンジニアリングシンポジウム 2026（SES2026），pp.278–279（2026）．

[2] Georges, A., Buytaert, D. and Eeckhout, L.: Statistically Rigorous Java Performance Evaluation, Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07), pp. 57–76, ACM (2007). DOI: 10.1145/1297027.1297033.

ⓒ2026 Information Processing Society of Japan

[3] Mytkowicz, T., Diwan, A., Hauswirth, M. and Sweeney, P. F.: Producing Wrong Data Without Doing Anything Obviously Wrong!, Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09), pp. 265–276, ACM (2009). DOI: 10.1145/1508244.1508275.

[4] Costa, D., Bezemer, C.-P., Leitner, P. and Andrzejak, A.: What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks, IEEE Transactions on Software Engineering, Vol. 47, No. 7, pp. 1452–1467 (2021). DOI: 10.1109/TSE.2019.2925345.

[5] Oracle：Arrays (Java SE 17 & JDK 17)（オンライン），入手先〈https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html〉（参照 2026-10-25）.

[6] Wild, S. and Nebel, M. E.: Average Case Analysis of Java 7's Dual Pivot Quicksort, Proc. 20th Annual European Symposium on Algorithms (ESA 2012), LNCS, Vol. 7501, pp. 825–836, Springer (2012). DOI: 10.1007/978-3-642-33090-2_71.

[7] Peters, T.: listsort.txt, CPython v3.10.0 ソースコード（オンライン），入手先〈https://github.com/python/cpython/blob/v3.10.0/Objects/listsort.txt〉（参照 2026-10-25）.

[8] Auger, N., Jugé, V., Nicaud, C. and Pivoteau, C.: On the Worst-Case Complexity of TimSort, Proc. 26th Annual European Symposium on Algorithms (ESA 2018), LIPIcs, Vol. 112, pp. 4:1–4:13 (2018). DOI: 10.4230/LIPIcs.ESA.2018.4.

[9] OpenJDK：Code Tools: jmh（オンライン），入手先〈https://openjdk.org/projects/code-tools/jmh/〉（参照 2026-10-25）.

[10] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S. and Tratt, L.: Virtual Machine Warmup Blows Hot and Cold, Proc. ACM on Programming Languages, Vol. 1, No. OOPSLA, Article 52 (2017). DOI: 10.1145/3133876.

[11] Kalibera, T. and Jones, R.: Rigorous Benchmarking in Reasonable Time, Proc. 2013 International Symposium on Memory Management (ISMM '13), pp. 63–74, ACM (2013). DOI: 10.1145/2464157.2464160.

ⓒ2026 Information Processing Society of Japan