OpenJDK
Installing 安装
Contributing 贡献
Sponsoring 赞助
Developers' Guide 开发人员指南
Vulnerabilities 漏洞
JDK GA/EA Builds JDK GA/EA 版本
Mailing lists 邮件列表
Wiki · IRC
维基 ·国际广播公司
Mastodon 乳齿象
Bluesky 蓝天
Bylaws · Census
章程 ·人口普查
Legal 法律
**Workshop** 车间
**JEP Process** JEP 流程
**Source code** 源代码
GitHub GitHub的
Mercurial 善变
**Tools** 工具
Git Git 公司
jtreg harness JTREG 线束
**Groups** 组
(overview) （概述）
Adoption 采用
Build 建
Client Libraries 客户端库
Compatibility & Specification Review
兼容性和规格审查
Compiler 编译器
Conformance 一致性
Core Libraries 核心库
Governing Board 理事会
HotSpot 热点
IDE Tooling & Support IDE 工具和支持
Internationalization 国际化

# JEP draft: G1: Improve Application Throughput with a More Efficient Write-Barrier

# JEP 草案：G1：使用更高效的写入屏障提高应用程序吞吐量

| | |
|---|---|
| *Author 作者* | Ivan Walulya & Thomas Schatzl |
| | 伊万·瓦卢利亚和托马斯·沙茨尔 |
| *Owner 所有者* | Ivan Walulya 伊万·瓦卢利亚 |
| *Type 类型* | Feature 特征 |
| *Scope 范围* | Implementation 实现 |
| *Status 地位* | Submitted 提交 |
| *Component 元件* | hotspot / gc 热点 / GC |
| *Discussion 讨论* | hotspot dash gc dash dev at openjdk dot org |
| *Effort 努力* | M |
| *Duration 期间* | M |
| *Reviewed by 校订者* | Vladimir Kozlov 弗拉基米尔·科兹洛夫 |
| *Created 创建* | 2024/09/24 16:13 |
| *Updated 更新* | 2024/12/03 09:11 |
| *Issue 问题* | 8340827 |

## Summary 总结

Increase the application throughput when the G1 garbage collector is used by reducing the impact of its barriers.

使用 G1 垃圾回收器时，通过减少其屏障的影响来提高应用程序吞吐量。

## Goals 目标

- Reduce the overall synchronization overhead necessary to operate the G1 garbage collector,

  减少操作 G1 垃圾回收器所需的整体同步开销，
- Reduce the size of the generated code for the G1 barriers,

  减小为 G1 barrier 生成的代码的大小，

JMX
Members 成员
Networking 联网
Porters 搬运工
Quality 质量
Security 安全
Serviceability 服務性
Vulnerability 脆弱性
Web 蹼

**Projects 项目**
(overview, archive)

（概述、 存档）

Amber 琥珀
Babylon 巴比伦
CRaC
Code Tools 代码工具
Coin 硬币
Common VM Interface 通用 VM 接口
Developers' Guide 开发人员指南
Device I/O 设备 I/O
Duke 公爵
Galahad 加拉哈德
Graal 格拉尔
IcedTea 冰茶
JDK 7 JDK 7 版本
JDK 8 JDK 8 版本
JDK 8 Updates JDK 8 更新
JDK 9 JDK 9 版本
JDK (..., 23, 24, 25)

JDK (..., 23、 24、 25)

JDK Updates JDK 更新
Jigsaw 拼图
Kona 科纳
Kulla 库拉
Lanai 拉奈岛
Leyden 莱 顿
Lilliput 小人 国
Locale Enhancement 区域设置增强
Loom 织布机
Memory Model Update 内存模型更新
Metropolis 都市
Mission Control 任务控制
Multi-Language VM 多语言 VM
Nashorn 纳斯霍恩
New I/O 新 I/O

- Keep the overall behavior of the G1 garbage collector the same, alternating between two phases where G1 only collects the young or both young and old generation memory and reclaims memory incrementally.

  保持 G1 垃圾回收器的整体行为不变，在两个阶段之间交替，其中 G1 仅收集新生代内存或新生代和老代内存，并逐步回收内存。

## Non-Goals 非目标

- Match the throughput of the G1 garbage collector with any other garbage collector.

  将 G1 垃圾回收器的吞吐量与任何其他垃圾回收器相匹配。

## Motivation 赋予动机

Java Virtual Machine (JVM) overhead is always a concern, especially in cloud-based Java deployments where billing is based on actual resource usage. One component that can impact CPU usage significantly is the garbage collector (GC). The default GC in the HotSpot JVM, the Garbage-First (G1) collector is tuned to keep a balance between latency and throughput for Java applications. However, keeping this balance can sometimes lead to considerable processing overhead compared to other throughput-oriented garbage collectors such as Parallel and Serial GC.

Java 虚拟机 （JVM） 开销始终是一个问题，尤其是在基于云的 Java 部署中，其中计费基于实际资源使用情况。垃圾回收器 （GC） 会显著影响 CPU 使用率的一个组件。HotSpot JVM 中的默认 GC 垃圾优先 （G1） 收集器经过调整，可在 Java 应用程序的延迟和吞吐量之间保持平衡。但是，与其他面向吞吐量的垃圾回收器（如 Parallel 和 Serial GC）相比，保持这种平衡有时会导致相当大的处理开销。

The main reason for this difference is that G1 performs some tasks concurrently with the application to meet latency goals, tasks that other garbage collectors might handle during the collection pause. The concurrent task execution requires additional synchronization between the garbage collector and the application, leading to substantial overhead. Specifically, the G1 uses a large amount of code to track every modification to the object graph in what is referred to as a write-barrier. For every field modification in the Java application, the G1 barrier adds about 50 x64 instructions [Protopopovs23], [JEP475], containing very expensive memory barriers, compared to 3 x64 instructions for the Parallel GC write-barrier.

造成这种差异的主要原因是 G1 与应用程序同时执行一些任务以满足延迟目标，这些任务是其他垃圾回收器在回收暂停期间可能会处理的任务。并发任务执行需要在垃圾回收器和应用程序之间进行额外的同步，这会导致大量开销。具体来说，G1 使用大量代码来跟踪对对象图的每次修改，这被称为写入屏障。对于 Java 应用程序中的每次字段修改，G1 屏障都会添加大约 50 个 x64 指令 [Protopopovs23]、[JEP475]，其中包含非常昂贵的内存屏障，而并行 GC 写入屏障则添加 3 个 x64 指令。

Depending on the Java application, this difference can result in up to 10% throughput regression compared to the Parallel collector in the current JDK (as noted in [JDK-8062128], [JDK-8253230], [JDK-8132937], and [deSmet21]).

根据 Java 应用程序的不同，与当前 JDK 中的 Parallel 收集器相比，这种差异可能会导致高达 10% 的吞吐量回归（如 [JDK-8062128]、[JDK-8253230]、[JDK-8132937] 和 [deSmet21] 中所述）。

Reducing this overhead is essential for increasing the adoption of the G1 in the cloud and minimizing the need for users to switch to a different garbage collection algorithm.

减少这种开销对于提高 G1 在云中的采用率和最大限度地减少用户切换到其他垃圾收集算法的需求至关重要。

## Description 描述

G1 garbage collection pause times depend on the number of changes made in the object graph during a Java application's execution. To track these changes, G1 uses a "card table," where each entry corresponds to a small section of memory, or "card," in the Java heap. This information is essential for quickly identifying live objects during a garbage collection pause [Hölzle93].

G1 垃圾回收暂停时间取决于 Java 应用程序执行期间对对象图所做的更改数量。为了跟踪这些变化，G1 使用了一个 “card table”，其中每个条目对应于 Java 堆中的一小部分内存或 “card”。此信息对于在垃圾回收暂停期间快速识别活动对象 [Hölzle93] 至关重要。

During the garbage collection pause, G1 examines the memory contents corresponding to the marked cards. In G1, there are two sources of these marked cards:

在垃圾回收暂停期间，G1 会检查与标记的卡对应的内存内容。在 G1 中，这些标记卡牌有两个来源：

- cards that were recently marked by the application, which G1 has not yet determined whether they contain useful information for the current garbage collection.

  卡，G1 尚未确定它们是否包含对当前垃圾回收有用的信息。

- cards that were already examined concurrently with the Java application, found to be relevant for collecting specific regions of the Java heap, and those regions have been selected for collection.

  已与 Java 应用程序同时检查的卡，发现与收集 Java 堆的特定区域相关，并且已选择这些区域进行收集。

Examining cards on the card table can take up to 60% of the total garbage collection pause time. G1 tries to minimize the number of cards to process during the collection pause to reduce the pause time and maintain its pause time goal.

检查卡表上的卡可能会占用总垃圾回收暂停时间的 60%。G1 会尝试最大限度地减少收集暂停期间要处理的卡的数量，以减少暂停时间并保持其暂停时间目标。

G1 controls the number of cards to process during the pause by examining recently marked cards concurrent to the Java application. Cards of interest are stored in a different data structure and the card is cleared (unmarked) to avoid processing in the pause. This task runs concurrent to the Java application at a frequency the pause time goal demands. The marking, unmarking, and examination of the cards by different threads (application and garbage collection threads) require expensive memory synchronization between these threads for correctness.

G1 通过检查与 Java 应用程序并发的最近标记的卡来控制暂停期间要处理的卡的数量。感兴趣的卡片存储在不同的数据结构中，并且卡片被清除（未标记）以避免在暂停时进行处理。此任务以暂停时间目标所需的频率与 Java 应用程序并发运行。不同线程（应用程序和垃圾回收线程）对卡的标记、取消标记和检查需要这些线程之间昂贵的内存同步才能确保正确性。

This JEP changes how marked cards are managed by G1, switching the current fine-grained memory synchronization in the write-barrier with much less frequent but more intrusive coarse-grained synchronization. This allows G1 to use a much smaller write-barrier; in our implementation for example, the write-barrier is about

20 instructions shorter on x86-64 architecture. Overall this approach results in higher throughput.

这个 JEP 改变了 G1 管理标记卡的方式，将写屏障中当前的细粒度内存同步切换为频率低得多但侵入性更强的粗粒度同步。这允许 G1 使用更小的写入屏障;例如，在我们的实现中，write-barrier 在 x86-64 架构上缩短了大约 20 条指令。总体而言，这种方法可以提高吞吐量。

The main idea is to confine the application and the garbage collection threads to always work on different card tables so that no memory synchronization is needed for examining the marked cards concurrent with the Java application. The application marks in one (regular, existing) card table, and the garbage collection threads process the second, initially empty, refinement (card) table. When G1 determines that enough marks were accumulated in the card table, the VM atomically swaps the card and refinement table. Then the Java application resumes marking the former empty refinement table, and the garbage collection threads process the marks as described before from the former regular card table without any further synchronization. This process can be repeated as necessary.

主要思想是将应用程序和垃圾回收线程限制为始终在不同的卡表上工作，这样就不需要内存同步来检查与 Java 应用程序并发的标记卡。应用程序在一个（常规、现有）card 表中进行标记，垃圾回收线程处理第二个（最初为空的）refinement （card） 表。当 G1 确定 card 表中积累了足够的分数时，VM 会自动交换 card 和 refinement 表。然后，Java 应用程序继续标记以前的空 refinement 表，并且垃圾回收线程如前所述处理来自以前的常规 card 表的标记，而无需任何进一步的同步。此过程可以根据需要重复。

## Performance 性能

With the proposed change, concurrent operations typically take less CPU resources, because the refinement table is very cache-friendly to access compared to the previous data structure that holds random card locations. There is also a small improvement in garbage collection pause times because previously required garbage collection phases can be simplified or completely removed.

通过提议的更改，并发操作通常占用更少的 CPU 资源，因为与之前保存随机卡位置的数据结构相比，优化表对缓存非常友好。垃圾回收暂停时间也有了一个小的改进，因为以前需要的垃圾回收阶段可以简化或完全删除。

Overall the main benefit is increased throughput of the Java application by doing less memory synchronization in the write-barrier, particularly for Java applications that heavily modify the object graph in the Java heap and so execute the write-barrier in full frequently. Depending on the CPU architecture and Java application we measured an increase in throughput by 5-15% or more due to decreased memory synchronization and concurrent examination.

总体而言，主要好处是通过在写入屏障中执行更少的内存同步来提高 Java 应用程序的吞吐量，特别是对于大量修改 Java 堆中的对象图并因此频繁地完全执行写入屏障的 Java 应用程序。根据 CPU 架构和 Java 应用程序，我们测量到由于内存同步和并发检查减少，吞吐量增加了 5-15% 或更多。

Even throughput-oriented applications that do very little or no concurrent examination at all may benefit from the decreased size of the barrier code, showing an increased throughput of around 5%.

即使是很少或根本不执行并发检查的面向吞吐量的应用程序也可能受益于屏障码大小的减小，显示吞吐量增加了约 5%。

Due to G1’s garbage collection CPU-usage based heap sizing, the impact of this change often results in decreased Java heap footprint instead of increased throughput.

由于 G1 的垃圾回收基于 CPU 使用率的堆大小调整，此更改的影响通常会导致 Java 堆占用空间减少，而不是吞吐量增加。

### Native Memory Footprint 本机内存占用

This change allocates an extra card table of the same size as the regular card table. By default, a card table corresponds to around 0.2% of Java heap memory size, which corresponds to 2MB native memory usage per 1GB of Java heap.

此更改分配了一个与常规 card 表大小相同的额外 card table。默认情况下，卡表对应于 Java 堆内存大小的 0.2% 左右，这相当于每 1GB Java 堆使用 2MB 本机内存。

Removing the previous data structures to track concurrent examination of cards approximately offsets half of that native memory unless the application does almost no object graph modifications. The optimization to keep the interesting cards of always collected regions on the card table also saves a significant amount of native memory.

删除以前的数据结构以跟踪卡的并发检查大约会偏移该本机内存的一半，除非应用程序几乎不进行对象图修改。将始终收集的区域的有趣卡片保留在卡片表上的优化还节省了大量的本机内存。

Overall, some applications may even use less native memory than before.

总体而言，某些应用程序使用的本机内存甚至可能比以前少。

## Alternatives 选择

Several alternatives to reduce the throughput difference between G1 and other collectors due to complex write-barriers have been proposed and prototyped before:

之前已经提出了几种替代方案来减少 G1 和其他收集器之间由于复杂的写屏障而导致的吞吐量差异，并进行了原型设计：

- Use operating system-dependent synchronization mechanisms to implement the coarse-grained synchronization between application and garbage collection threads. This made the synchronization operating system and even operating system version dependent. Some platforms do not implement this functionality directly (for example OSX-aarch64, or old versions of linux-riscv) or in a too generic way (older Linux kernels), so workarounds that for example use debugging APIs would need to be used. This would lead to the performance of this functionality being very poor on some platforms, so the current technique using Thread-Local Handshakes [JEP312] would have been needed anyway as a workaround to these deficiencies.

  使用依赖于操作系统的同步机制实现应用程序和垃圾回收线程之间的粗粒度同步。这使得同步操作系统甚至操作系统版本都依赖于同步。某些平台不直接实现此功能（例如 OSX-aarch64 或旧版本的 linux-riscv）或以过于通用的方式实现（较旧的 Linux 内核），因此需要使用使用调试 API 等解决方法。这将导致此功能在某些平台上的性能非常差，因此无论如何都需要使用线程本地握手 [JEP312] 的当前技术来解决这些缺陷。

- Keep existing data structures to store where marked cards on the card table are instead of using the refinement table. This would have the advantage that the change itself would be smaller, however, the possible reduction of the write-barrier size would be much smaller and limited to avoid the memory synchronization, saving only five instructions instead of twenty. Removing the old data structures also reduces work in the garbage collection pause, reducing garbage collection pause times. Management of the refinement table is much less complex. The prototype in this proposal removes around 1000 lines of code in the JVM sources, mostly related to this data structure, also significantly reducing code complexity.

保留现有数据结构以存储卡片表上标记的卡片的位置，而不是使用细化表。这样做的好处是更改本身会更小，但是，写屏障大小的可能减小会小得多，并且会受到限制以避免内存同步，只保存 5 条指令而不是 20 条。删除旧数据结构还可以减少垃圾回收暂停中的工作，从而减少垃圾回收暂停时间。优化表的管理要简单得多。该提案中的原型删除了 JVM 源代码中的大约 1000 行代码，这些代码主要与此数据结构相关，也显著降低了代码复杂性。

- Use the same throughput-optimized write-barrier as Parallel GC, disable all concurrent examination of cards, and have G1 do all work in the garbage collection pauses. The end user determines to use one or the other “garbage collection mode” based on his service level preferences. A master’s thesis [Protopopovs23] showed that the pause time impact can be negligible in throughput-oriented applications, but this proposal would have two distinct “modes” of operation for G1. This adds significant complexity to the code base, increases the test surface significantly, and is inconvenient for the end user as he needs to know in advance which mode to select. Alternatively the end user may as well select Parallel GC instead, which would still be slightly faster than G1 on pure throughput loads as shown in the thesis.

  使用与 Parallel GC 相同的吞吐量优化写屏障，禁用所有并发检查卡，并让 G1 在垃圾回收暂停时执行所有工作。最终用户根据其服务级别首选项决定使用其中一种或另一种 “垃圾收集模式”。硕士论文 [Protopopovs23] 表明，在面向吞吐量的应用程序中，暂停时间的影响可以忽略不计，但该提案将为 G1 提供两种不同的“操作模式”。这大大增加了代码库的复杂性，显著增加了测试表面，并且对最终用户来说很不方便，因为他需要提前知道要选择哪种模式。或者，最终用户也可以选择并行 GC，在纯通量负载上，它仍然比 G1 略快，如论文所示。

- Modify the original write-barrier to batch the memory barrier execution, and remove the per-object graph memory synchronization. The cost of this approach is much more enqueuing work in some applications, decreasing throughput compared to the original G1. There are no observed performance regressions with the approach proposed here.

  修改原始 write-barrier 以批处理内存屏障执行，并删除每个对象的图形内存同步。这种方法的代价是在某些应用程序中需要更多的排队工作，与原始 G1 相比，吞吐量会降低。此处建议的方法没有观察到性能回归。

Overall we think that the trade-off between code complexity, maintenance overhead, and performance characteristics is most favorable for this proposal.

总的来说，我们认为代码复杂性、维护开销和性能特征之间的权衡对这个提案最有利。

## Risks and Assumptions 风险和假设

We assume that the refinement table increases the native memory footprint for G1 slightly. We also assume that a significant portion of this will be compensated by the removal of the data structures for maintaining examining marked cards and optimizations like keeping the interesting cards of always collected regions on the card table.

我们假设 refinement table 略微增加了 G1 的原生内存占用。我们还假设，其中很大一部分将通过删除用于维护检查标记卡片和优化的数据结构来补偿，例如将始终收集的区域的有趣卡片保留在卡片桌上。

This is an intrusive change to critical components of G1 interaction with application threads, there is a non-negligible risk of bugs that may cause failures and introduce performance regressions. To mitigate this risk, we will conduct careful code reviews and perform extensive testing.

这是对 G1 与应用程序线程交互的关键组件的侵入性更改，存在不可忽视的 bug 风险，这些 bug 可能会导致失败并引入性能回归。为了降低这种风险，我们将进行仔细的代码审查并执行广泛的测试。