Previous in the Series 该系列的上一页	Current Tutorial 当前教程	That's the end of the series! 系列到此结束！
Overview of ZGC ZGC 概述	Deep-dive of ZGC's Architecture 深入探讨 ZGC 的架构	

Deep-dive of ZGC's Architecture
深入探讨 ZGC 的架构

Introduction 介绍

In the previous article, we did a high-level overview of ZGC and how to configure it. This article will dive deep into the key implementation details and architectural decisions that guide ZGC.

在上一篇文章中，我们对 ZGC 及其配置方法进行了高级概述。本文将深入探讨指导 ZGC 的关键实现细节和架构决策。

Concurrency and GC Cycle
并发和 GC 循环

One of ZGC's major advantages is its extremely low pause times of under 1 ms. This is accomplished by ZGC being an almost entirely concurrent garbage collector. Below is a chart of some of the high-level processes GCs go through each GC cycle and if the process is performed concurrently(✅):

ZGC 的主要优势之一是其极低的暂停时间，不到 1 毫秒。这是通过 ZGC 几乎完全是一个并发垃圾回收器来实现的。下面是一个图表，列出了 GC 在每个 GC 周期中经历的一些高级流程，以及如果该流程同时执行（✅）：

	Serial 串行	Parallel 平行	G1	ZGC
Marking 标记	❌	❌	✅*	✅
Relocation/Compaction 重新定位/压缩	❌	❌	❌	✅
Reference Processing 参考处理	❌	❌	❌**	✅
Relocation Set Selection 重定位集选择	❌	❌	❌	✅
JNI WeakRef Cleaning JNI WeakRef 清理	❌	❌	❌	✅
JNI GlobalRefs Scanning JNI GlobalRefs 扫描	❌	❌	❌	✅
Class Unloading 类卸载	❌	❌	❌	✅
Thread Stack Scanning 线程堆栈扫描	❌	❌	❌	✅

Note: Chart based on JDK 19

注意： 基于 JDK 19 的图表

* Old Gen only

* 仅限老一代

** Partially concurrent ** 部分并发

ZGC being able to handle almost every GC process concurrently essentially turns the pause phases into short synchronization points that don't increase with the live set's size and provide consistent performance regardless of scale.

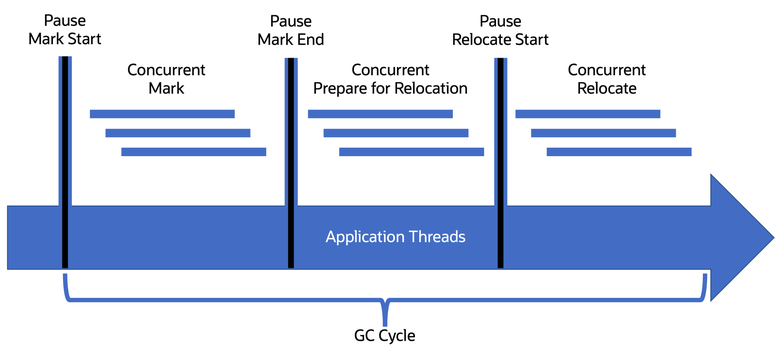
ZGC 能够同时处理几乎所有 GC 流程，基本上将暂停阶段转变为较短的同步点，这些同步点不会随着实时集的大小而增加，并且无论规模如何，都能提供一致的性能。

ZGC GC Cycle ZGC GC 循环

The GC cycle consists of three pauses and three concurrent phases, each with distinct responsibilities.

Below is a diagram showing a simplified view of the ZGC's GC cycle:

GC 周期由 3 个 paping 和 3 个并发阶段组成，每个阶段都有不同的职责。下图显示了 ZGC GC 周期的简化视图：



Pause Mark Start 暂停标记开始

Synchronization point to signal the start of the mark phase.

同步点，用于表示标记阶段的开始。

During this and all pause phases, only minor actions are being taken, like setting boolean flags and what the "good" current global colors are; see Color Pointers.

在此和所有暂停阶段，仅执行一些小作，例如设置布尔标志以及当前全局颜色的“良好”颜色是什么;请参阅颜色指针。

Concurrent Mark 并发标记

During this concurrent phase, ZGC will walk the entire object graph and mark all objects.

在这个并发阶段，ZGC 将遍历整个对象图并标记所有对象。

Pause Mark End 暂停标记结束

Synchronization point to signal the end of marking.

同步点，用于表示标记结束。

Concurrent Prepare for Relocation
并发准备重新定位

During this concurrent phase, ZGC will remove objects

在此并发阶段，ZGC 将删除对象

Pause Relocate Start 暂停 重新定位起点

Synchronization point signals to threads that objects will be moved around in the heap.

同步点向线程发出信号，表明对象将在堆中移动。

Concurrent Relocate 并发重定位

During this concurrent phase, ZGC will move objects and compact regions in the heap to free up space. For more on this phase, check the section on compaction.

在此并发阶段，ZGC 将移动堆中的对象和压缩区域以释放空间。有关此阶段的更多信息，请查看 压缩.

Colored Pointers 彩色指针

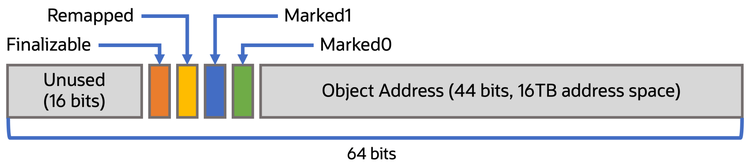
A central part of GC work is to move objects around on the heap while avoiding the application using an outdated reference to a moved object. A straightforward way to achieve that is to pause the application during that work, but for ZGC to achieve its goal of low pause times, it must perform nearly all its work concurrently. You can see the potential issue: Even though ZGC performs its work while the application is running, it must ensure the application always gets the correct reference. It accomplishes this through two key architectural decisions, colored pointers and load barriers. Let’s take a look at colored pointers.

GC 工作的核心部分是在堆上移动对象，同时避免应用程序使用对移动对象的过时引用。实现这一目标的一种直接方法是在该工作期间暂停应用程序，但要使 ZGC 实现其低暂停时间的目标，它必须同时执行几乎所有工作。您可以看到潜在的问题：即使 ZGC 在应用程序运行时执行其工作，它也必须确保应用程序始终获得正确的引用。它通过两个关键的架构决策来实现这一点，即彩色指针和负载屏障。让我们看一下彩色指针。

In this tutorial 在本教程中

ZGC uses a 64-bit pointer with 22 bits reserved for metadata about the pointer. The 22 metadata bits provide "color" to the pointer that can provide information about the current state of the pointer. Colored pointers are similar to tag and version pointers used in other GC implementations. As of JDK 19, colored pointers in ZGC look like this diagram:

ZGC 使用 64 位指针，其中 22 位保留用于有关指针的元数据。22 个元数据位为指针提供“颜色”，该指针可以提供有关指针当前状态的信息。彩色指针类似于其他 GC 实现中使用的 tag 和 version 指针。从 JDK 19 开始，ZGC 中的彩色指针如下图所示：



Currently, 4 bits are in use, while the other 18 remain in reserve for future use. The purpose for each bit is as follows:

目前，4 位正在使用中，而其他 18 位仍保留以备将来使用。每个位的用途如下：

- **Finalizable:** This bit indicates if the object is only reachable through a finalizer. Note that finalization was designated as deprecated for removal in JDK 18 with JEP 421: Deprecate Finalization for Removal.

  可定型： 此位指示是否只能通过 Finalizer 访问对象。请注意，在 JDK 18 中，使用 JEP 421 将终结指定为已弃用，以便删除。

- **Remapped:** This bit indicates if the pointer is known *not* to point into the relocation set.

  重新映射： 此位指示是否已知指针*未*指向重定位集。

- **Marked0 & Marked1:** These bits indicate if the object is known to be marked by the GC. ZGC alternates between these two bits as to which is "good" for each GC cycle.

  标记 0 & 标记 1： 这些位指示是否已知对象由 GC 标记。ZGC 在这两个位之间交替，以确定哪个位对于每个 GC 周期来说是 “好的”。

Each bit has a "good" and "bad" color; however, what is a "good" or "bad" color would be context specific to when the object is accessed. The application itself would not be aware of the colored pointers; the reading of colored pointers is handled by load barriers when an object is loaded from heap memory.

每个位都有 “good” 和 “bad” 颜色;但是，什么是“好”或“坏”颜色将特定于访问对象的上下文。应用程序本身不会知道彩色指针;当从堆内存加载对象时，彩色指针的读取由 Load Barriers 处理。

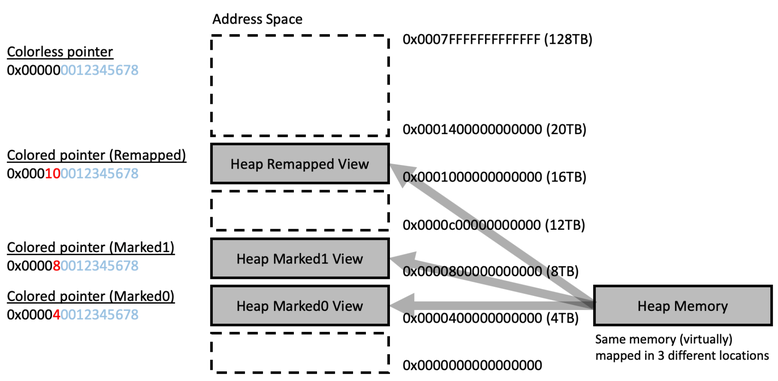
Heap Multi-Mapping 堆多重映射

Because ZGC can move the physical location of an object in heap memory while the application is running, multiple paths need to be provided to the current physical location where the object resides. In ZGC, this is accomplished through heap multi-mapping. With multi-mapping, the physical location of an object is mapped to three views in virtual memory, corresponding to each of the pointer's potential "colors". This allows a load barrier to locate an object if it has been moved since the last synchronization point.

由于 ZGC 可以在应用程序运行时移动对象在堆内存中的物理位置，因此需要提供多个路径到对象所在的当前物理位置。在 ZGC 中，这是通过堆多重映射实现的。使用多重映射，对象的物理位置将映射到虚拟内存中的三个视图，对应于指针的每个潜在“颜色”。这允许 load barrier 定位自上一个同步点以来已移动的对象。

A consequence of this design decision is that a system can report ZGC's memory usage as higher than its real usage. This is a consequence of the triplicate addressing of an object in *virtual* memory; however, the real memory usage is only from where the actual object is located. This can most easily be understood when a system reports memory usage higher than the physical memory installed on the system. The below chart demonstrates what multi-mapping looks like in practice:

此设计决策的结果是，系统可以将 ZGC 的内存使用率报告为高于其实际使用率。这是*虚拟内存中对象*一式三份寻址的结果;但是，实际内存使用情况仅来自实际对象所在的位置。当系统报告的内存使用率高于系统上安装的物理内存时，这很容易理解。下图演示了实际中的多映射：



Load Barriers 负载障碍

In the previous section, we covered how colored pointers were one of ZGC's major architectural decisions to allow concurrent processing; this section covers the other key architectural decision, load barriers.

在上一节中，我们介绍了彩色指针如何成为 ZGC 允许并发处理的主要架构决策之一;本节介绍另一个关键的架构决策，即 Load Barriers。

Load barriers are code segments injected by the C2 compiler, part of the JIT, into class files when the JVM parses them. Load barriers are added into class files where an object would be retrieved from the heap. The below Java code example shows where a load barrier would be added:

Load barrier 是 C2 编译器（JIT 的一部分）在 JVM 解析时注入类文件的代码段。Load barrier 被添加到类文件中，其中将从堆中检索对象。下面的 Java 代码示例显示了将添加负载屏障的位置：

```
Object o = obj.fieldA();

<load barrier added here by C2>

Object p = o; //No barrier, not a load from the heap

o.doSomething(); //No barrier, not a load from the heap

int i = obj.fieldB(); //No barrier, not and object reference
```

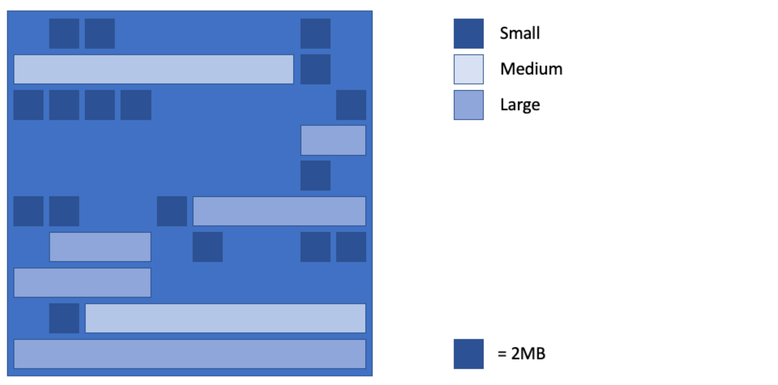
The load barrier adds behavior that will check the "colors" of an object's pointer when loaded from the heap. Load barriers are optimized for the "good" color case, which is the common case, to allow quicker pass-through. Suppose a load barrier encounters a "bad" color. In that case, it will attempt to heal the color, which might mean updating the pointer to put the object's new location on the heap or even relocating the object itself before returning the reference to the system. This healing ensures that subsequent loads of the object from the heap take the fast path.

load barrier 添加了从堆加载时检查对象指针 “colors” 的行为。负载屏障针对 “良好” 颜色情况（常见情况）进行了优化，以允许更快的直通。假设 Load Barrier 遇到 “bad” 颜色。在这种情况下，它将尝试修复颜色，这可能意味着更新指针以将对象的新位置放在堆上，甚至在将引用返回给系统之前重新定位对象本身。这种修复可确保从堆中加载对象的后续作采用快速路径。

Regions 地区

ZGC does not treat the heap as a single bucket to toss objects into but dynamically divides the heap into separate memory regions, like in this (simplified) diagram below:

ZGC 不会将堆视为一个用于扔出对象的存储桶，而是动态地将堆划分为单独的内存区域，如下图（简化）所示：



This follows a similar pattern to G1 GC, which also uses memory regions. However, ZGC regions, internally defined as ZPages, are more dynamic, with small, medium, and large sizes; the number of active regions can increase and decrease depending on the needs of the live set.

这遵循与 G1 GC 类似的模式，后者也使用内存区域。但是，ZGC 区域（内部定义为 ZPages）更具动态性，具有小型、中型和大型大小;活动区域的数量可以根据 Live Set 的需要增加和减少。

Dividing the heap into regions can provide several benefits to GC performance, including; the cost of allocating and deallocating a region of a set size would be constant, the GC can deallocate an entire region

when all objects within in it are no longer reachable, related objects can be grouped into a region.

将堆划分为多个区域可以为 GC 性能带来多种好处，包括：分配和取消分配一定大小的区域的成本是恒定的，当其中的所有对象都不再可访问时，GC 可以取消分配整个区域，相关对象可以分组到一个区域中。

Region Sizes 区域大小

As per the above chart, ZGC has three different sizes for regions; small, medium, and large. It also stands out that, paradoxically, a large region can be smaller than a medium region. The below covers the different region sizes and their purposes.

根据上图，ZGC 有三种不同的区域大小;小、中、大。此外，自相矛盾的是，大区域可能小于中等区域。下面介绍了不同的区域大小及其用途。

Small Regions 小区域

Small regions are 2 MB in size. Objects less than 1/8th (12.5%) the size of a small region, so less than or equal to 256 KB, are stored in a small region.

Medium Regions

The size of a medium region can vary depending on what the max heap size (`-Xmx`) is set to. 1 GB or greater and medium regions are set to 32 MB in size; below 128 MB, medium regions are disabled. Like with small regions, objects 1/8th (12.5%) or less in size than the set size of a medium region will be stored there. Below is the chart for the ranges of medium region size:

Max Heap Size	Medium Region Size
>= 1024 MB	32 MB
>= 512 MB	16 MB
>= 256 MB	8 MB
>= 128 MB	4 MB
< 128 MB	off

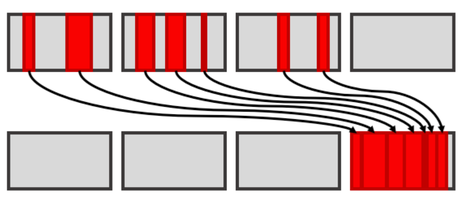
Large Regions

Large regions are reserved for humongous objects and are tightly fitted in 2 MB increments to the object's size. So a 13 MB object would be stored in a 14 MB large region. Any object too large to fit in a medium region will be placed in its own large region.

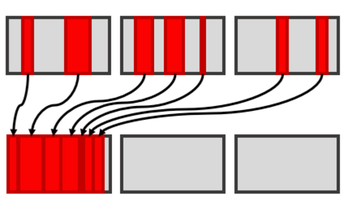
Compaction and Relocation

Regions are designed to take advantage of the fact that *most* objects created at the same time will leave scope at the same time. However, as implied with the qualifier *most*, that isn't always the case. Through internal GC heuristics, the GC might eventually copy objects out of a region populated mainly by inaccessible objects into a new region to allow the old region to be deallocated and memory to be freed up. This is called compaction and relocation. ZGC, since JDK 16, accomplishes compaction through two methods of relocation in-place and not-in-place.

Not-in-place relocation is performed when empty regions are available and is ZGC's preferred relocation method. Below is an example of what not-in-place relocation looks like:



If no empty regions are available, ZGC will use in-place relocation. In this scenario, ZGC will move objects into a sparsely populated region. Below is an example of in-place relocation:



With in-place relocation, ZGC must first compact the objects within the region designated for objects to be relocated into. This can negatively impact performance as only a single thread can perform this work. Increasing heap size can help ZGC avoid having to use in-place relocation.

Additional Reading

Here are a few links that might be worth checking out to learn more about ZGC.

ZGC Team Wiki: https://wiki.openjdk.org/display/zgc/Main

Per Liden's (original developer of ZGC) blog: https://malloc.se/

Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK: https://dl.acm.org/doi/full/10.1145/3538532

More Learning

Java's Highly Scalable Low-Latency Garbage Col...

Z Garbage Collector: The Next Generation

Last update: *March 6, 2022*

Previous in the Series		Current Tutorial		
Overview of ZGC	←	**Deep-dive of ZGC's Architecture**	→	That's the end of the series!

Home > Tutorials > Garbage Collection in Java Overview > Deep-dive of ZGC's Architecture