

为Java开疆拓土的ZGC深度剖析

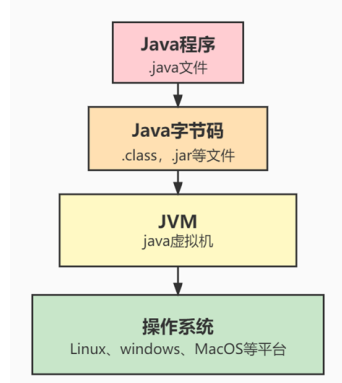
引言

有道云链接: <http://note.youdao.com/noteshare?id=dc72118bc273acecfe83ff045e0278b2&sub=B9AB3211720149288CFEC501FEE4B94C>

对于Java 程序员来说, JVM 帮助我们做了很多事情。

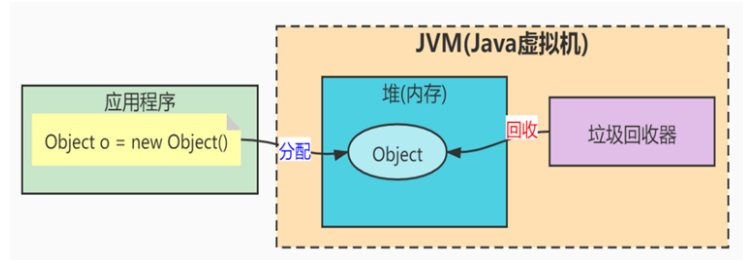
JVM是虚拟机, 能够识别字节码, 就是class文件或者你打包的jar文件, 运行在操作系统上。

JVM帮我们实现了跨平台, 你只需要编译一次, 就可以在不同的操作系统上运行, 并且效果是一致的。



在Java中你使用对象, 使用内存, 不用担心回收, 只管new对象就行了, 不用管垃圾的回收。

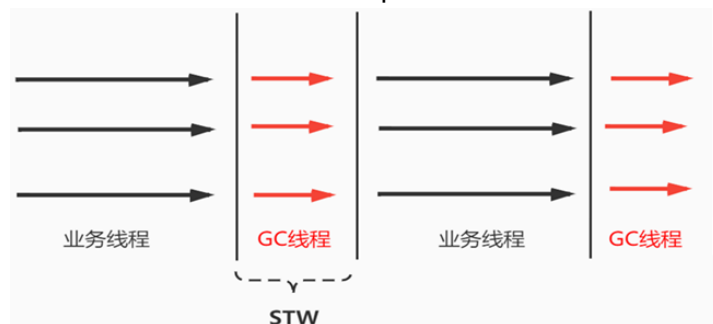
因为Java当中是自动化的垃圾回收机制。JVM有专门的垃圾回收器, 把垃圾回收这件事给干了。



ZGC出现背景

但是对于Java的项目来说, JVM进行垃圾回收会有一个很大的问题, 就是STW。

什么是STW, STW的全称是StopTheWorld。



Java项目中, 如果JVM要进行垃圾回收, 会暂停所有的业务线程, 也就是项目中的线程, 这样会导致业务系统暂停。

STW带来的问题

手机系统(Android) 显示卡顿

Google 主导的 Android 系统需要解决的一大问题就是显示卡顿问题, 通过对 GC 算法的不断演进, 停顿时间控制在几个ms 级别。

所以这也是Android与苹果IOS系统竞争的一大利器。

证券交易系统实时性要求

证券交易系统主要就是买入、卖出，现在都是使用系统完成自动下单，如果用Java系统来做，遇到了STW，假如STW的时间是3秒。

刚收到市场行情是比较低的买入的，但是因为STW卡顿了3秒，3秒后的市场行情可能完全不同。所以如果使用Java来做证券系统，一定是要求STW时间越短越好！

大数据平台(Hadoop集群性能)

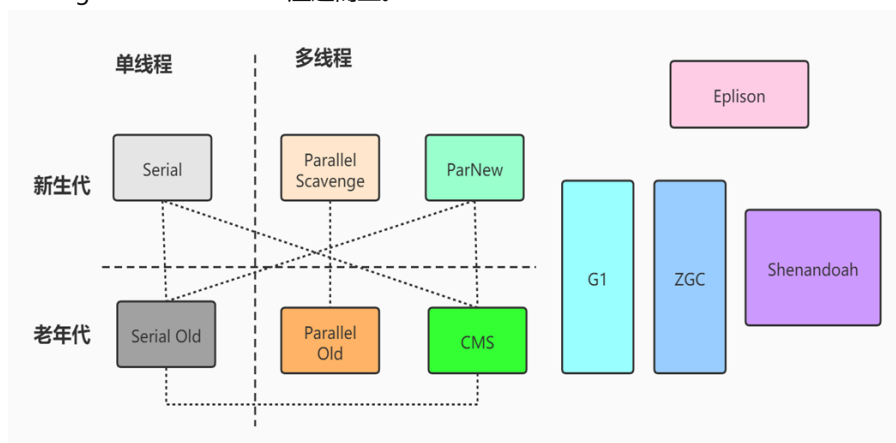
同样，像King老师的前东家，58同城的大数据系统，单集群5000+的Hadoop集群，日万亿级实时数据分发。如果遇到STW也是不行的。

垃圾回收器的发展

为了满足不同的业务需求，Java 的 GC 算法也在不停迭代，对于特定的应用，选择其最适合的 GC 算法，才能更高效的帮助业务实现其业务目标。对于这些延迟敏感的应用来说，GC 停顿已经成为阻碍 Java 广泛应用的一大顽疾，需要更适合的 GC 算法以满足这些业务的需求。

近些年来，服务器的性能越来越强劲，各种应用可使用的堆内存也越来越大，常见的堆大小从 10G 到百 G 级别，部分机型甚至可以到达 TB 级别，在这类大堆应用上，传统的 GC，如 CMS、G1 的停顿时间也跟随着堆大小的增长而同步增加，即堆大小指数级增长时，停顿时间也会指数级增长。特别是当触发 Full GC 时，停顿可达分钟级别(百GB级别的堆)。当业务应用需要提供高服务级别协议（Service Level Agreement, SLA），例如 99.99% 的响应时间不能超过 100ms，此时 CMS、G1 等就无法满足业务的需求。

为满足当前应用对于超低停顿、并应对大堆和超大堆带来的挑战，伴随着 2018 年发布的 JDK 11，A Scalable Low-Latency Garbage Collector - ZGC 应运而生。



ZGC介绍

ZGC (The Z Garbage Collector) 是[JDK 11](#)中推出的一款追求极致低延迟的垃圾收集器，它曾经设计目标包括：

- 停顿时间不超过10ms (JDK16已经达到不超过1ms)；
- 停顿时间不会随着堆的大小，或者活跃对象的大小而增加；
- 支持8MB~4TB级别的堆，JDK15后已经可以支持16TB。

这么去想，如果使用ZGC来做Java项目，像对STW敏感的证券系统，游戏的系统都可以去用Java来做（以前都是C或者C++的市场），所以ZGC的出现就是为了抢占其他语言的市场（卷！）。

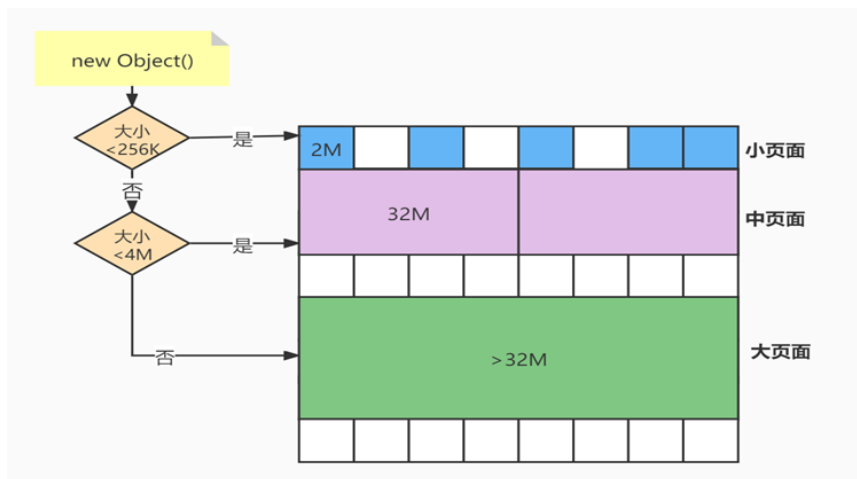
ZGC中的内存布局

为了细粒度地控制内存的分配，和G1一样，ZGC将内存划分成小的分区，在ZGC中称为页面（page）。

ZGC中没有分代的概念（新生代、老年代）

ZGC支持3种页面，分别为小页面、中页面和大页面。

其中小页面指的是2MB的页面空间，中页面指32MB的页面空间，大页面指受操作系统控制的大页。



当对象大小小于等于256KB时，对象分配在小页面。

当对象大小在256KB和4M之间，对象分配在中页面。

当对象大于4M，对象分配在大页面。

ZGC对于不同页面回收的策略也不同。**简单地说，小页面优先回收；中页面和大页面则尽量不回收。**

为什么这么设计？

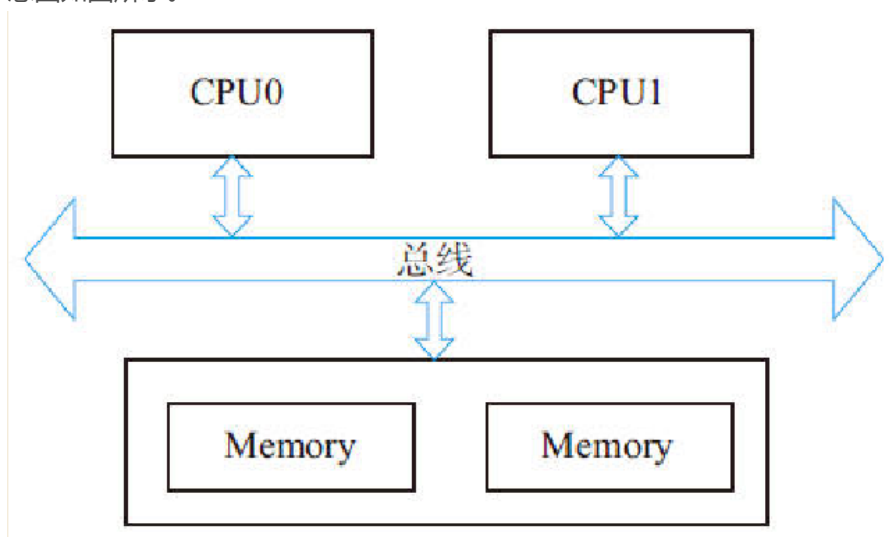
标准大页（huge page）是Linux Kernel 2.6引入的，目的是通过使用大页内存来取代传统的4KB内存页面，以适应越来越大的系统内存，让操作系统可以支持现代硬件架构的大页面容量功能。

Huge pages 有两种格式大小：2MB 和 1GB，2MB 页块大小适合用于 GB 大小的内存，1GB 页块大小适合用于 TB 级别的内存；2MB 是默认的页大小。

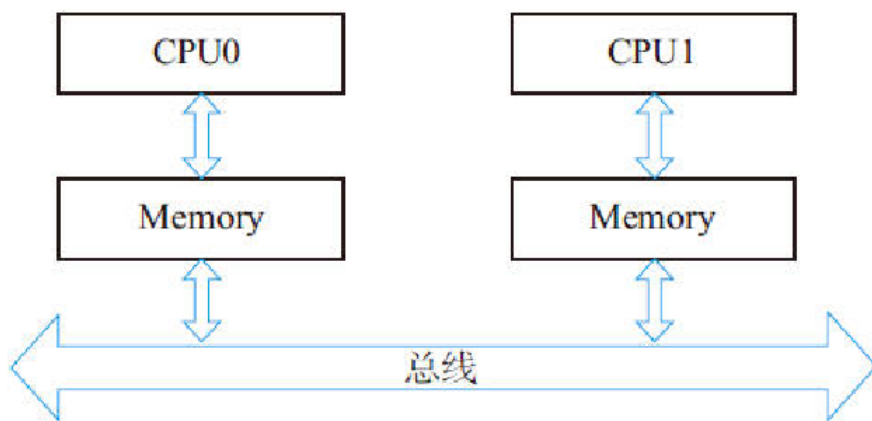
所以ZGC这么设置也是为了适应现代硬件架构的发展，提升性能。

ZGC支持NUMA（了解即可）

在过去，对于X86架构的计算机，内存控制器还没有整合进CPU，所有对内存的访问都需要通过北桥芯片来完成。X86系统中的所有内存都可以通过CPU进行同等访问。任何CPU访问任何内存的速度是一致的，不必考虑不同内存地址之间的差异，这称为“统一内存访问”（Uniform Memory Access, UMA）。UMA系统的架构示意图如图所示。



在UMA中，各处理器与内存单元通过互联总线进行连接，各个CPU之间没有主从关系。之后的X86平台经历了一场从“拼频率”到“拼核心数”的转变，越来越多的核心被尽可能地塞进了同一块芯片上，各个核心对于内存带宽的争抢访问成为瓶颈，所以人们希望能够把CPU和内存集成在一个单元上（称Socket），这就是非统一内存访问（Non-Uniform Memory Access, NUMA）。很明显，在NUMA下，CPU访问本地存储器的速度比访问非本地存储器快一些。下图所示是支持NUMA处理器架构示意图。



ZGC是支持NUMA的，在进行小页面分配时会优先从本地内存分配，当不能分配时才会从远端的内存分配。对于中页面和大页面的分配，ZGC并没有要求从本地内存分配，而是直接交给操作系统，由操作系统找到一块能满足ZGC页面的空间。ZGC这样设计的目的在于，对于小页面，存放的都是小对象，从本地内存分配速度很快，且不会造成内存使用的不平衡，而中页面和大页面因为需要的空间大，如果也优先从本地内存分配，极易造成内存使用不均衡，反而影响性能。

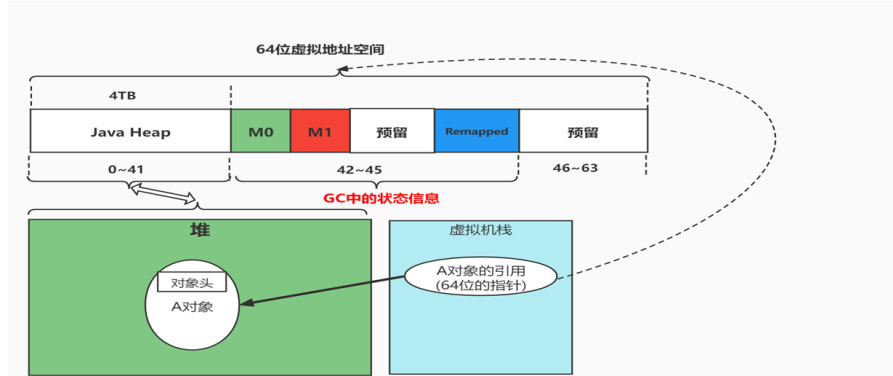
ZGC的核心概念

指针着色技术 (Color Pointers)

颜色指针可以说是ZGC的核心概念。因为他在指针中借了几个位出来做事情，所以它必须要求在64位的机器上才可以工作。并且因为要求64位的指针，也就不能支持压缩指针。

ZGC中低42位表示使用中的堆空间

ZGC借几位高位来做GC相关的事情(快速实现垃圾回收中的并发标记、转移和重定位等)



我们通过一个例子演示Linux多视图映射。Linux中主要通过系统函数mmap完成视图映射。多个视图映射就是多次调用mmap函数，多次调用的返回结果就是不同的虚拟地址。示例代码如下

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>

int main()
{
    //创建一个共享内存的文件描述符
    int fd = shm_open("/example", O_RDWR | O_CREAT | O_EXCL, 0600);
    if (fd == -1) return 0;
```

```

//防止资源泄露,需要删除。执行之后共享对象仍然存活,但是不能通过名字访问
shm_unlink("/example");

//将共享内存对象的大小设置为4字节
size_t size = sizeof(uint32_t);
ftruncate(fd, size);

//3次调用mmap,把一个共享内存对象映射到3个虚拟地址上
int prot = PROT_READ | PROT_WRITE;
uint32_t *remapped = mmap(NULL, size, prot, MAP_SHARED, fd, 0);
uint32_t *m0 = mmap(NULL, size, prot, MAP_SHARED, fd, 0);
uint32_t *m1 = mmap(NULL, size, prot, MAP_SHARED, fd, 0);

//关闭文件描述符
close(fd);

//测试,通过一个虚拟地址设置数据,3个虚拟地址得到相同的数据
*remapped = 0xdeafbeef;
printf("48bit of remapped is: %p, value of 32bit is: 0x%x\n", remapped, *remapped);
printf("48bit of m0 is: %p, value of 32bit is: 0x%x\n", m0, *m0);
printf("48bit of m1 is: %p, value of 32bit is: 0x%x\n", m1, *m1);

return 0;
}

```

在Linux上通过gcc编译后运行文件,得到的执行文件:

```
gcc -lrt -o mapping mapping.c
```

```

[root@centosvm zgc]# gcc -lrt -o mapping mapping.c
[root@centosvm zgc]# ls
mapping mapping.c

```

然后执行下,我们来看下执行结果

```

[root@centosvm zgc]# ./mapping
48bit of remapped is: 0x7f93aef8e000, value of 32bit is: 0xdeafbeef
48bit of m0 is: 0x7f93aef8d000, value of 32bit is: 0xdeafbeef
48bit of m1 is: 0x7f93aef8c000, value of 32bit is: 0xdeafbeef

```

从结果我们可以发现,3个变量对应3个不同的虚拟地址。

实地址: (32位指针) 是: 0xdeafbeef <一位16进制代表4位二进制>

虚地址: (48位指针):

0x7f93aef8e000 <虚地址remapped>

0x7f93aef8d000 <虚地址m0>

0x7f93aef8c000 <虚地址m1>

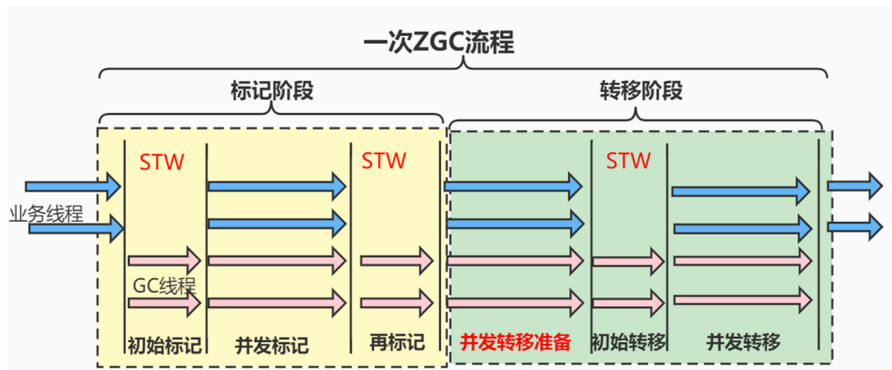
但是因为它们都是通过mmap映射同一个内存共享对象,所以它们的物理地址是一样的,并且它们的值都是0xdeafbeef。

ZGC流程

一次ZGC流程

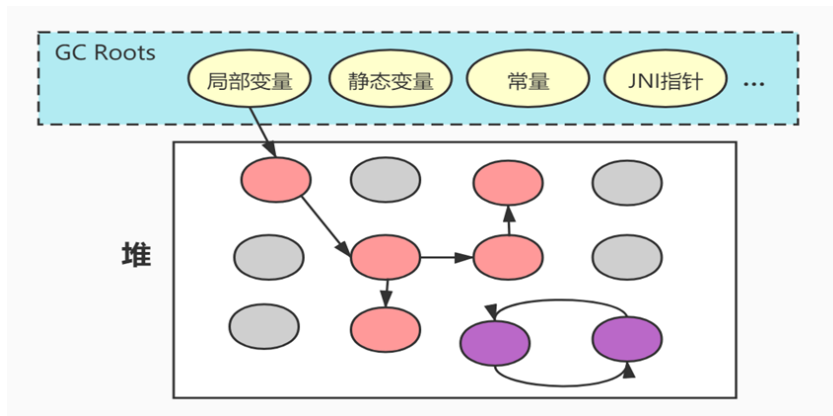
标记阶段(标识垃圾)

转移阶段(对象复制或移动)



根可达算法

来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。



作为GC Roots的对象主要包括下面4种

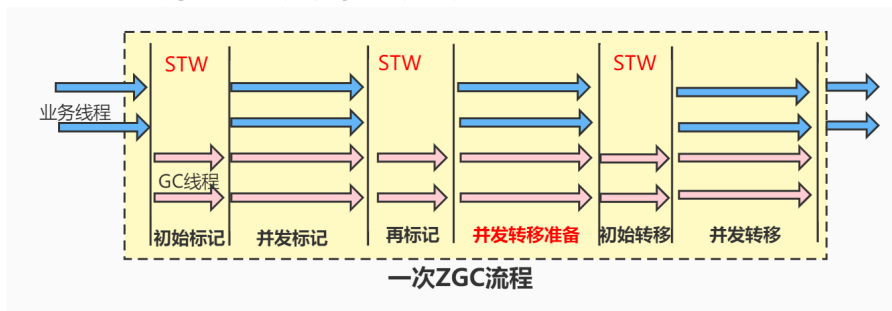
- 虚拟机栈（栈帧中的本地变量表）：各个线程调用方法堆栈中使用到的参数、局部变量、临时变量等。
- 方法区中类静态变量：java类的引用类型静态变量。
- 方法区中常量：比如：字符串常量池里的引用。
- 本地方法栈中JNI指针：（即一般说的Native方法）。

ZGC中初始标记和并发标记

初始标记：从根集合(GC Roots)出发，找出根集合直接引用的活跃对象(根对象)

并发标记：根据初始标记找到的根对象，使用深度优先遍历对象的成员变量进行标记

ZGC基于指针着色的并发标记算法



0. 初始阶段

在ZGC初始化之后，此时地址视图为Remapped，程序正常运行，在内存中分配对象，满足一定条件后垃圾回收启动。

1、初始标记

这个阶段需要暂停（STW），初始标记只需要扫描所有GC Roots，其处理时间和GC Roots的数量成正比，停顿时间不会随着堆的大小或者活跃对象的大小而增加。

2、并发标记

这个阶段不需要暂停（没有STW），扫描剩余的所有对象，这个处理时间比较长，所以走并发，业务线程与GC线程同时运行。但是这个阶段会产生漏标问题。

3、再标记

这个阶段需要暂停（没有STW），主要处理漏标对象，通过SATB算法解决（G1中的解决漏标的方案）。

ZGC基于指针着色的并发转移算法

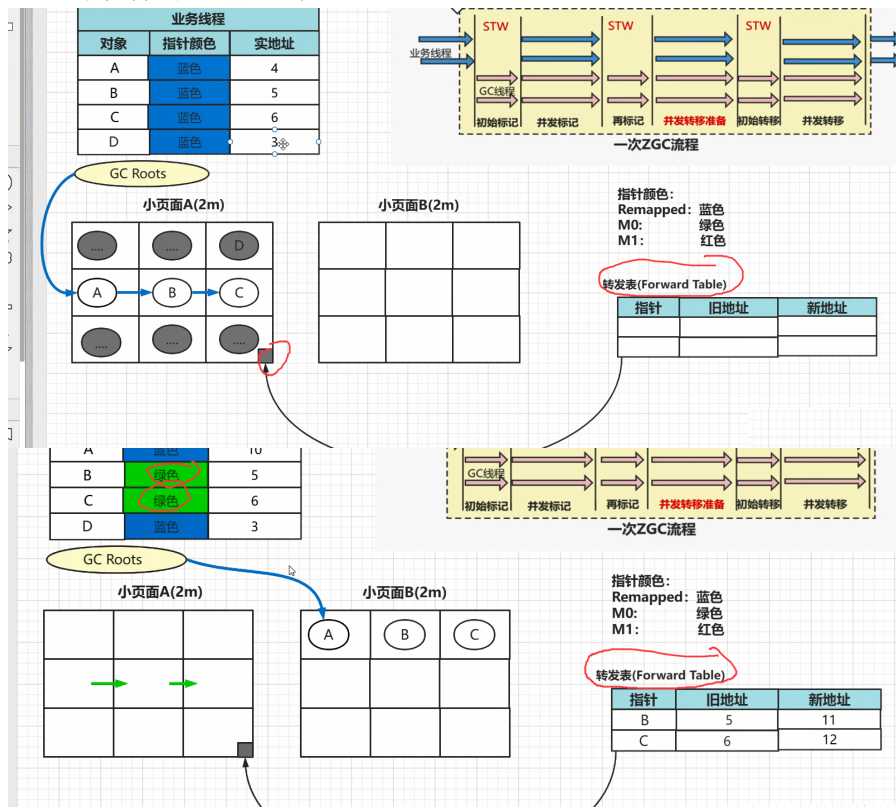
ZGC的转移阶段

- 并发转移准备(分析最有价值GC分页<无STW >)
- 初始转移（转移初始标记的存活对象同时做对象重定位<有STW> ）
- 并发转移（对转移并发标记的存活对象做转移<无STW>）

如何做到并发转移？

转发表(类似于HashMap)

对象转移和插转发表做原子操作

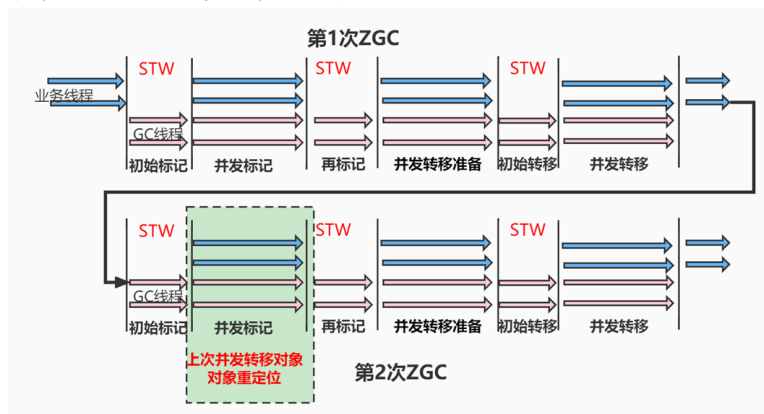


ZGC基于指针着色的重定位算法

并发标记对象的重定位

下次GC中的并发标记（同时做上次并发标记对象的重定位）

技术上：指针着色中M0和M1区分



ZGC中读屏障

ZGC中的读屏障

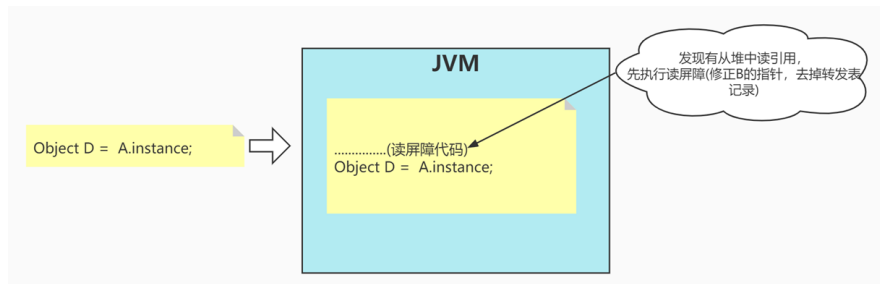
涉及对象：并发转移但还没做对象重定位的对象（着色指针使用M0和M1可以区分）

触发时机：在两次GC之间业务线程访问这样的对象

触发操作：对象重定位+删除转发表记录（两个一起做原子操作）

读屏障是JVM向应用代码插入一小段代码的技术。当应用线程从堆中读取对象引用时，就会执行这段代码。

需要注意的是，仅“从堆中读取对象引用”才会触发这段代码。



ZGC中GC触发机制（JAVA16）

预热规则：服务刚启动时出现，一般不需要关注。日志中关键字是“Warmup”。

JVM启动预热，如果从来没有发生过GC，则在堆内存使用超过10%、20%、30%时，分别触发一次GC，以收集GC数据。

```
0) Garbage Collection (Warmup) 252M(12%)->638M(31%)
1) Garbage Collection (Warmup)
1) Pause Mark Start 0.017ms
1) Concurrent Mark 509.124ms
1) Pause Mark End 0.010ms
1) Concurrent Process Non-Strong References 0.582ms
1) Concurrent Reset Relocation Set 0.001ms
1) Concurrent Select Relocation Set 8.341ms
1) Pause Relocate Start 0.006ms
1) Concurrent Relocate 142.085ms
```

基于分配速率的自适应算法：最主要的GC触发方式（默认方式），其算法原理可简单描述为“ZGC根据近期的对象分配速率以及GC时间，计算出当内存占用达到什么阈值时触发下一次GC”。通过ZAllocationSpikeTolerance参数控制阈值大小，该参数默认2，数值越大，越早的触发GC。日志中关键字是“Allocation Rate”。

```
] [gc          ] GC(127) Garbage Collection (Allocation Rate) 1254M(61%)->744M(36%)
] [gc, start    ] GC(128) Garbage Collection (Allocation Rate)
] [gc, phases   ] GC(128) Pause Mark Start 0.005ms

] [gc, phases   ] GC(128) Concurrent Mark 124.350ms
] [gc, phases   ] GC(128) Pause Mark End 0.269ms
] [gc, phases   ] GC(128) Concurrent Process Non-Strong References 0.821ms
] [gc, phases   ] GC(128) Concurrent Reset Relocation Set 0.076ms
] [gc, phases   ] GC(128) Concurrent Select Relocation Set 6.188ms
] [gc, phases   ] GC(128) Pause Relocate Start 0.006ms
] [gc, phases   ] GC(128) Concurrent Relocate 90.977ms
```

基于固定时间间隔：通过ZCollectionInterval控制，适合应对突增流量场景。流量平稳变化时，自适应算法可能在堆使用率达到95%以上才触发GC。流量突增时，自适应算法触发的时机可能会过晚，导致部分线程阻塞。我们通过调整此参数解决流量突增场景的问题，比如定时活动、秒杀等场景。

主动触发规则：类似于固定间隔规则，但时间间隔不固定，是ZGC自行算出来的时机，我们的服务因为已经加了基于固定时间间隔的触发机制，所以通过-ZProactive参数将该功能关闭，以免GC频繁，影响服务可用性。

阻塞内存分配请求触发：当垃圾来不及回收，垃圾将堆占满时，会导致部分线程阻塞。我们应当避免出现这种触发方式。日志中关键字是“Allocation Stall”。

外部触发：代码中显式调用System.gc()触发。日志中关键字是“System.gc()”。

元数据分配触发：元数据区不足时导致，一般不需要关注。日志中关键字是“Metadata GC Threshold”。

ZGC参数设置

ZGC 优势不仅在于其超低的 STW 停顿，也在于其参数的简单，绝大部分生产场景都可以自适应。当然，极端情况下，还是有可能需要对 ZGC 个别参数做个调整，大致可以分为三类：

- **堆大小**：Xmx。当分配速率过高，超过回收速率，造成堆内存不够时，会触发 Allocation Stall，这类 Stall 会减缓当前的用户线程。因此，当我们在 GC 日志中看到 Allocation Stall，通常可以认为堆空间偏小或者 concurrent gc threads 数偏小。
- **GC 触发时机**：ZAllocationSpikeTolerance, ZCollectionInterval。ZAllocationSpikeTolerance 用来估算当前的堆内存分配速率，在当前剩余的堆内存下，ZAllocationSpikeTolerance 越大，估算的达到 OOM 的时间越快，ZGC 就会更早地进行触发 GC。ZCollectionInterval 用来指定 GC 发生的间隔，以秒为单位触发 GC。
- **GC 线程**：ParallelGCThreads, ConcGCThreads。ParallelGCThreads 是设置 STW 任务的 GC 线程数目，默认为 CPU 个数的 60%；ConcGCThreads 是并发阶段 GC 线程的数目，默认为 CPU 个数的 12.5%。增加 GC 线程数目，可以加快 GC 完成任务，减少各个阶段的时间，但也会增加 CPU 的抢占开销，可根据生产情况调整。

由上可以看出 ZGC 需要调整的参数十分简单，通常设置 Xmx 即可满足业务的需求，大大减轻 Java 开发者的负担。

ZGC典型应用场景

对于性能来说，不同的配置对性能的影响是不同的，如充足的内存下即大堆场景，ZGC 在各类 Benchmark 中能够超过 G1 大约 5% 到 20%，而在小堆情况下，则要低于 G1 大约 10%；不同的配置对于应用的影响不尽相同，开发者需要根据使用场景来合理判断。

当前 ZGC 不支持压缩指针和分代 GC，其内存占用相对于 G1 来说要稍大，在小堆情况下较为明显，而在大堆情况下，这些多占用的内存则显得不那么突出。**因此，以下两类应用强烈建议使用 ZGC 来提升业务体验：**

- **超大堆应用**。超大堆（百 G 以上）下，CMS 或者 G1 如果发生 Full GC，停顿会在分钟级别，可能会造成业务的终端，强烈推荐使用 ZGC。
- **当业务应用需要提供高服务级别协议（Service Level Agreement, SLA）**，例如 99.99% 的响应时间不能超过 100ms，此类应用无论堆大小，均推荐采用低停顿的 ZGC。

ZGC生产注意事项

RSS 内存异常现象

由前面 ZGC 原理可知，ZGC 采用多映射 multi-mapping 的方法实现了三份虚拟内存指向同一份物理内存。而 Linux 统计进程 RSS 内存占用的算法是比较脆弱的，这种多映射的方式并没有考虑完整，因此根据当前 Linux 采用大页和小页时，其统计的开启 ZGC 的 Java 进程的内存表现是不同的。在内核使用小页的 Linux 版本上，这种三映射的同一块物理内存会被 linux 的 RSS 占用算法统计 3 次，因此通常可以看到使用 ZGC 的 Java 进程的 RSS 内存膨胀了三倍左右，但是实际占用只有统计数据的三分之一，会对运维或者其他业务造成一定的困扰。而在内核使用大页的 Linux 版本上，这部分三映射的物理内存则会统计到 hugetlbfs inode 上，而不是当前 Java 进程上。

共享内存调整

ZGC 需要在 share memory 中建立一个内存文件来作为实际物理内存占用，因此当要使用的 Java 的堆大小大于 /dev/shm 的大小时，需要对 /dev/shm 的大小进行调整。通常来说，命令如下（下面是将 /dev/shm 调整为 64G）：

```
vi/etc/fstabtmpfs /dev/shm tmpfs defaults,size= 65536M00
```

首先修改 fstab 中 shm 配置的大小，size 的值根据需求进行修改，然后再进行 shm 的 mount 和 umount。
umount/dev/shmmount /dev/shm

mmap 节点上限调整

ZGC 的堆申请和传统的 GC 有所不同，需要占用的 memory mapping 数目更多，即每个 ZPage 需要 mmap 映射三次，这样系统中仅 Java Heap 所占用的 mmap 个数为 $(Xmx / zpage_size) * 3$ ，默认情况下 zpage_size 的大小为 2M。

为了给 JNI 等 native 模块中的 mmap 映射数目留出空间，内存映射的数目应该调整为 $(Xmx / zpage_size) * 3 * 1.2$ 。

默认的系统 memory mapping 数目由文件 `/proc/sys/vm/max_map_count` 指定，通常数目为 65536，当给 JVM 配置一个很大的堆时，需要调整该文件的配置，使得其大于 $(Xmx / zpage_size) * 3 * 1.2$ 。

ZGC存在的问题及持续改进

目前ZGC历代版本中存在的一些问题（阿里、腾讯、美团、华为等大厂在支持业务切换 ZGC 的出现的），基本上都已经将遇到的相关问题和修复积极向社区报告和回馈，很多问题在JDK16和JDK17已经修复完善。另外的话，问题相对来说不是非常严重，如果遇到类似的问题可以查看下JVM团队的历代修复日志，同时King老师的建议就是尽量使用比较新的版本来上线，以免重复掉坑里面。