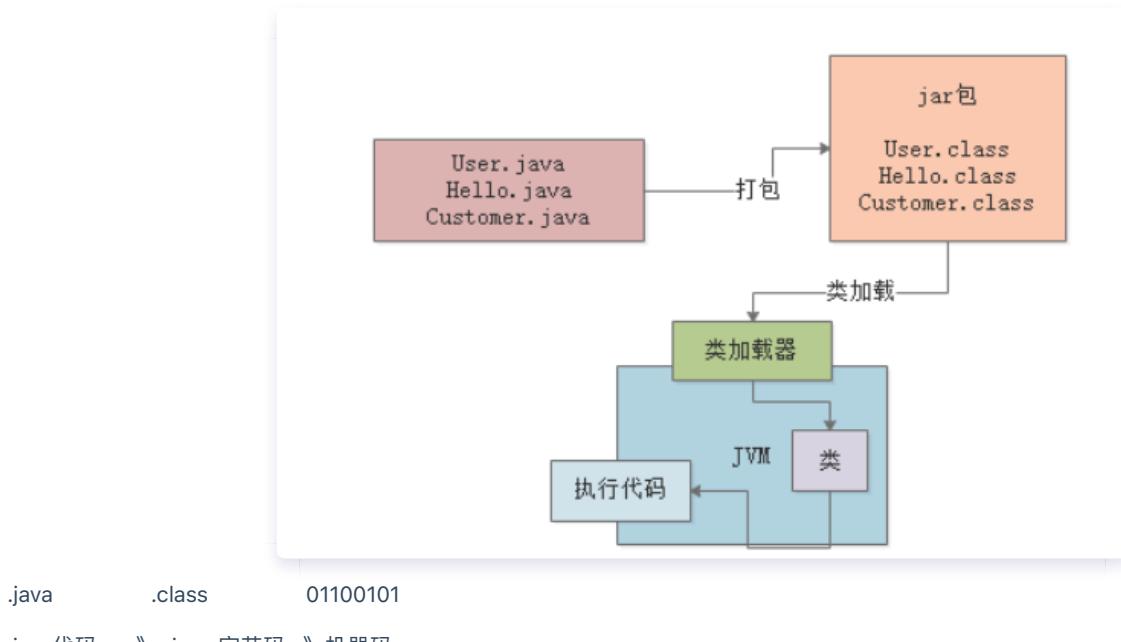


JVM

笔记来自于：《从 0 开始带你成为 JVM 实战高手》 -- 儒猿技术窝

1. 前言

1.1 java 代码如何运行？



1.2 类加载到使用过程 (.class 字节码文件加载到 JVM 内存中)

加载 验证 准备 解析 初始化 使用 卸载

实用角度简单解析

- 验证 .class 文件内容是否符合 JVM 规范
- 准备 类变量(static 修饰的变量)分配内存空间 赋默认的初始值
- 解析 (很复杂, 符号引用替换为实际引用)
- 初始化 (核心阶段) 执行类的初始化代码块 (static 代码块)

什么时候初始化一个类？

1. new 实例化对象时，触发类的加载到初始化的全过程
2. 包含 main 方法的主类，必须立马初始化好
3. 初始化类时，他的父类没有初始化，就必须先初始化他的父类

1.3 什么情况会加载一个类?

1. 类有main方法作为程序主入口
2. 代码执行用到某个类的时候

1.4 类加载器和双亲委派机制

1.启动类加载器

BootStrap ClassLoader 加载java目录下的核心类 (java安装目录下的"lib"目录) java最核心的类库, 支撑java系统运行
一旦jvm启动, 首先就会依托启动类加载器。

2.扩展类加载器

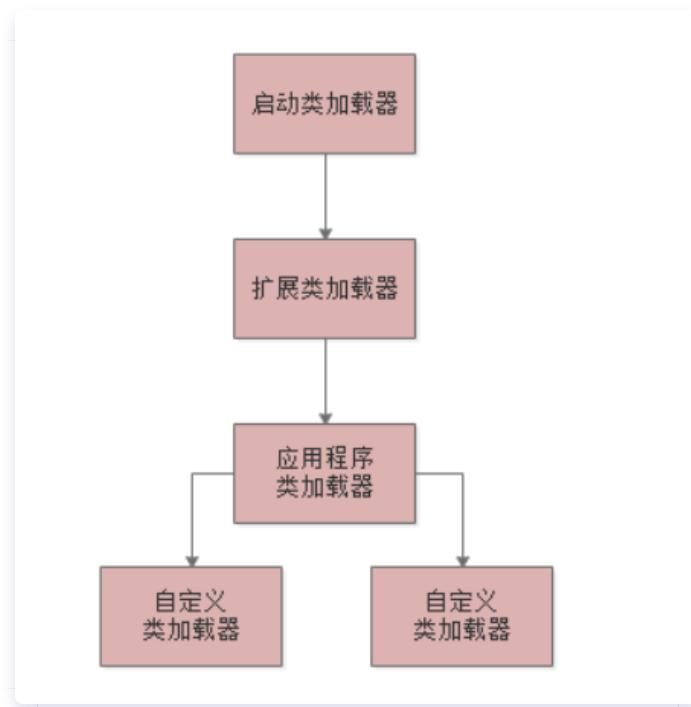
Extension ClassLoader java安装目录下有 "lib/ext"目录, 就是用它来加载, 支撑系统运行。

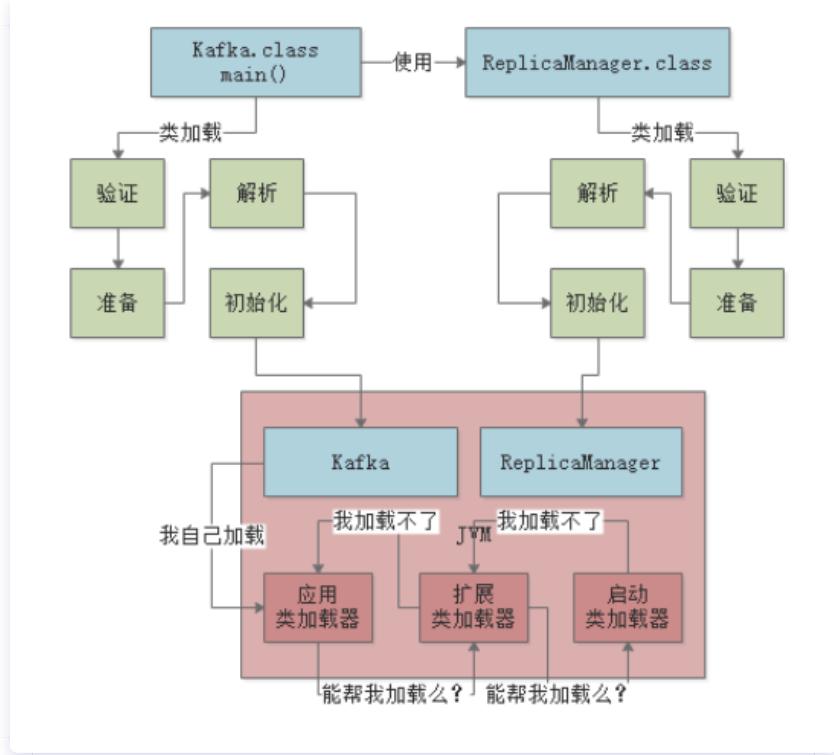
3.应用程序类加载器

Application ClassLoader 复制加载 "ClassPath" 环境变量所指定的路径的类 (大致理解为加载自己写好的Java代码)

4.自定义类加载器

根据自己的需求加载





双亲委派流程:

当一个Hello.class这样的文件要被加载时。不考虑我们自定义类加载器，首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到父加载器，然后调用父加载器的loadClass方法。父类中同理也会先检查自己是否已经加载过，如果没有再往上。注意这个类似递归的过程，直到到达Bootstrap classLoader之前，都是在检查是否加载过，并不会选择自己去加载。直到BootstrapClassLoader，已经没有父加载器了，这时候开始考虑自己是否能加载了，如果自己无法加载，会下沉到子加载器去加载，一直到最底层，如果没有任何加载器能加载，就会抛出ClassNotFoundException。

双亲委派模式优势

避免重复加载 + 避免核心类篡改

采用双亲委派模式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关系可以避免类的重复加载，当父亲已经加载了该类时，就没有必要由ClassLoader再加载一次。其次是考虑到安全因素，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心JavaAPI发现这个名字的类，发现该类已被加载，并不会重新加载网络传递过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。

为什么必须要一级一级类加载器的往上找，直接从顶层类加载器开始找不就行了吗？

其实关于这个问题，不用过于纠结，每一层类加载器对某个类的加载，上推给父类加载器，到顶层类加载器，如果发现自己加载不到，再下推回子类加载器来加载，这样可以保证绝对不会重复加载某个类。至于为什么不直接从顶层类加载器开始找，那是因为类加载器本身就是做的父子关系模型

你想一下Java代码实现，他最底下的子类加载器，只能通过自己引用的父类加载器去找。如果直接找顶层类加载器，不合适的，那么顶层类加载器不就必须硬编码规定了吗？

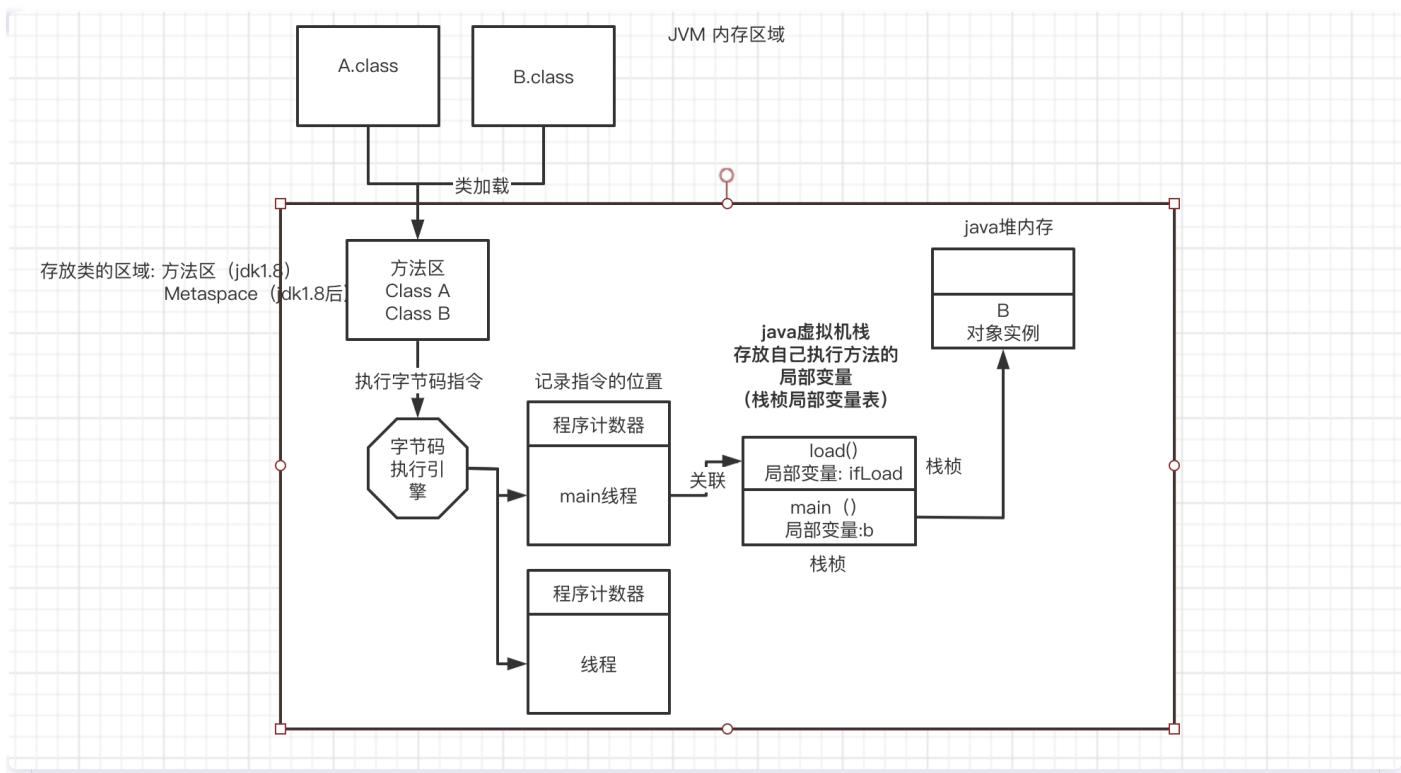
这就是一个代码设计思想，保证代码的可扩展性。

ps: 为了防止源代码泄露, 可以用工具对字节码加密。在类加载时, 用自定义的类加载器来解密文件。

1.5 JVM 内存区域

```
public class A {  
    public static void main(String[] args) {  
        B b = new B();  
        b.load();  
    }  
}
```

```
public class B {  
  
    public void load(){  
        boolean ifLoad = false;  
    }  
}
```



内存区域流程:

结合上图, JVM启动, 加载 A类到内存中, 然后有一个main线程执行main方法, main线程关联一个**程序计数器**, 记录执行到哪一行。其次, 在main线程关联的**java虚拟机栈**里压入main方法的**栈帧**。main方法中需要创建B类的实例对象, 这时把B类加载到内存中, 创建B的对象实例分配在**java 堆内存**中, 并且在main()方法的栈帧中的局部变量表引入b变量, 引用到B对象在堆内存中的地址。接着执行b.load 方法, 创建栈帧, 创建局部变量。执行完之后, 把对应的方法出栈。

12.JVM中有哪些内存区域 答：

a.方法区：在JDK1.8以后，这块区域的名字改了，叫做"Metaspace"。主要存放我们自己写的各种类相关的信息。

b.程序计数器：字节码指令通过字节码执行引擎被一条一条执行，才能实现我们写好的代码执行的效果。

程序计数器就是用来记录当前执行的字节码指令的位置，也就是记录目前执行到了哪一条字节码指令。

JVM是支持多个线程的，所以就会有多个线程来并发执行不同的代码指令，因此，每个线程都会有自己的一个程序计数器，专门记录当前这个线程目前执行到了哪一条字节码指令。

c.Java虚拟机栈：保存每个方法内的局部变量等数据。每个线程都会有自己的Java虚拟机栈。

如果线程执行了一个方法，就会对这个方法调用创建对应的一个栈帧（栈帧里就有这个方法的局部变量表、操作数栈、动态链接、方法出口等），然后压入线程的Java虚拟机栈。方法执行完毕之后就从Java虚拟机栈出栈。

因此，每个线程在执行代码时，除了程序计数器以外，还搭配了一个Java虚拟机内存区域来存放每个方法中的局部变量。

d. Java堆内存：存放我们在代码中创建的各种对象。对象实例里面会包含一些数据。而Java虚拟机栈的栈帧局部变量表里面的对象，其实是一个引用类型的局部变量，存放了对应Java堆内存对象的地址。可以理解为局部变量表里的对象指向了Java

1.6 垃圾回收

我们在Java堆内存里创建的对象，都是占用内存资源的，而且内存资源有限。

如上面 b.load() 方法执行完之后，Java堆内存的 B 对象实例没有变量引用它了，这时这可能就是一种浪费。

解决方法：Jvm垃圾回收机制

本身是一个后台自动运行的线程，只要启动一个JVM进程，就会自带这么一个垃圾回收的后台线程。

这个线程会不断检查JVM堆内存中的各个实例对象。

如果某个实例的对象没有任何一个方法的局部变量、类的静态变量，包括常量等地方在指向他。那么垃圾回收线程就会把这个没人指向的 "B" 实例对象回收，从内存里清除。这些不被指向的对象实例就是Jvm的"垃圾"。

思考问题：我们创建的对象，到底在Java堆内存中占用多少内存？

一个对象对内存的占用分两块

- 对象自己本身的一些信息
- 对象的实例变量作为数据占用的空间

在64位的linux操作系统上，对象头会占用 16 字节，如果实例对象内部有个int类型的实例变量，会占用4个字节，如果是long类型的实例变量会占用8个字节。如果是map, list就更多了。

jvm有许多优化的地方，如 补齐机制，指针压缩机制。较复杂，暂时不学。

既然堆内存的对象会被回收，那么方法区（永久代）的类会被垃圾回收吗？什么时候回收？为什么？

会，但是要满足下面的三个条件

- 首先，该类的所有实例对象都已经从 java 堆内存里被回收
- 其次，加载这个类的 ClassLoader 已经被收回
- 最后，对该类的 Class 对象没有任何引用（Class 对象无法通过任何途径访问（包括反射）

1.7 tomcat 相关问题

tomcat本身是java程序，那么tomcat的实现程序的class是由应用类加载器加载的，用户自己的java程序war包，放入tomcat的程序的classpath中这样用户的程序和tomcat的程序都是由应用类加载器加载了，也就是处于一个jvm中了

前言结束下面的学习会更细节。

1.8 内存分代模型

根据对象生命相对的长短，JVM对对象进行了分代，年轻代、老年代、永久代。

JVM将 Java堆内存 划分为了两个区域，一个是年轻代，一个是老年代。

- 年轻代，创建和使用完之后立马就要回收的对象存放的区域
- 老年代，创建之后需要一直长期存在的对象存放的区域。
- 永久代其实就是之前说的方法区，保存类相关信息

永久代类的回收满足下面三个条件就可以回收该类了

- 首先该类的所有实例对象都已经从Java堆内存里被回收；
- 其次加载这个类的ClassLoader已经被回收；
- 最后对该类的Class对象没有任何引用。

结合代码和图例进行理解

```
public class A {  
    private static C c = new C();  
    public static void main(String[] args) throws InterruptedException {  
        loadB();  
    }  
}
```

```

while(true){
    loadC();
    Thread.sleep(1000);
}

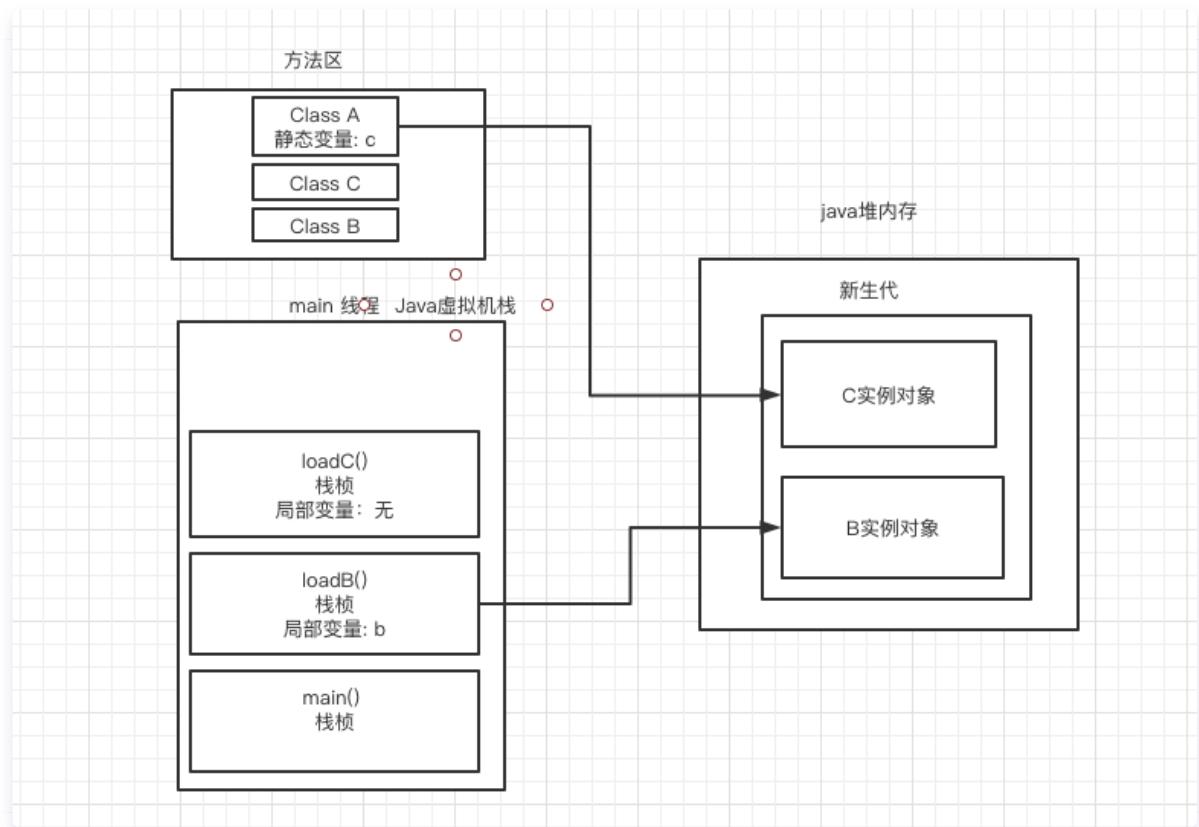
public static void loadB(){
    B b =new B();
    b.loadFromDisk();

}

public static void loadC(){
    c.loadFromRemote();
}

}

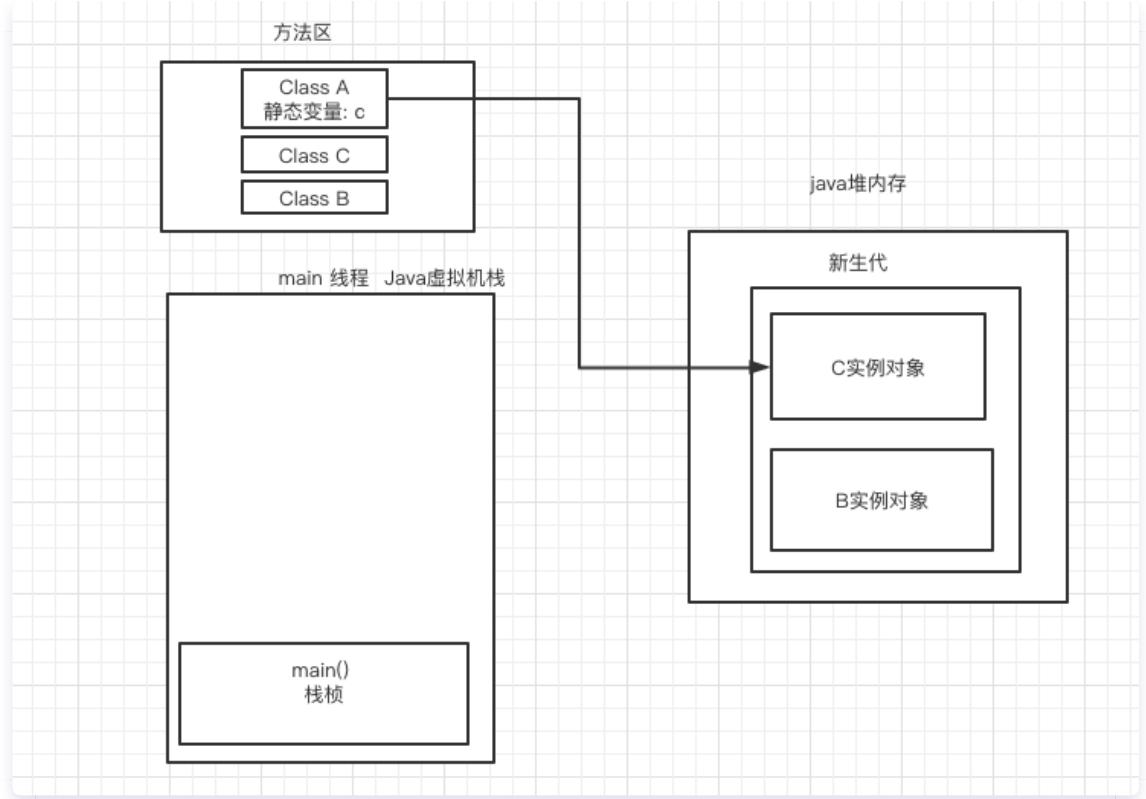
```



大部分正常对象都是优先在新生代分配内存的，类静态变量c 引用的C对象实例是会长期存活在内存里的，但是一开始也是分配在新生代里。

那么什么情况下会触发新生代的回收？

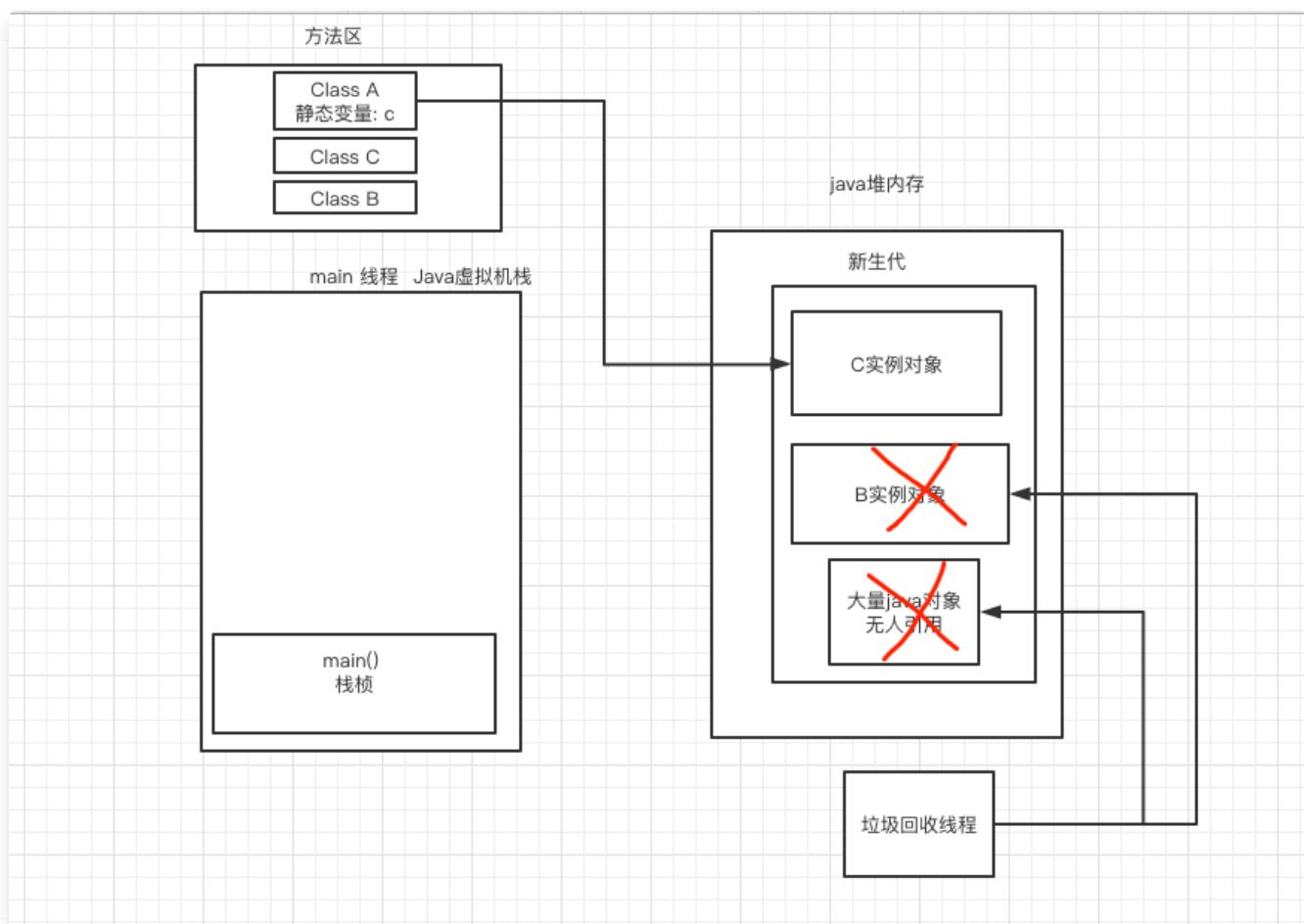
假设 loadB () 方法执行完毕，这个方法的栈帧出栈，导致没有变量引用B实例对象。



会立即触发垃圾回收吗？

不会，垃圾回收需要触发条件的，假设现在创建了许多的对象。导致java堆内囤积了大量的对象，这个时候新生代预先分配的内存空间满了，他又需要在新生代分配一个新对象。这时候就触发了一次新生代的垃圾回收 "Minor GC" 也叫"Young GC"

这时候就会把大量没人引用的对象包括B回收掉。



但是“C对象”，会一直存活在新生代里，因为它一直被 A类的静态变量引用，不会被回收。

这时JVM规定，对象每垃圾回收一次，年龄就+1，所以当上图的 C对象在新生代成功躲过10多次垃圾回收，成为“老年人”就会被认为是会长期存活在内存里的对象，被转移到 java堆内存的老年代。

(后续) 对象分配还有许多复杂机制:

- 新生代垃圾回收后，存活对象太多导致大量对象进入老年代
- 特别大的超大对象直接不经过新生代就进入老年代
- 动态对象年龄判断机制
- 空间担保机制

1.9 Jvm核心参数

一般来说，对于线上部署系统启动的时候，有多种方式设置Jvm参数。

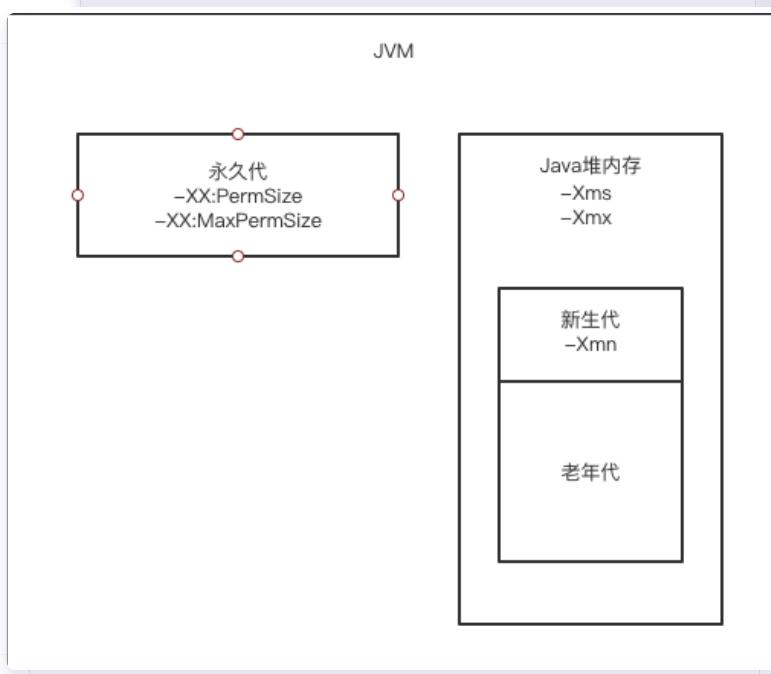
Idea中设置 Debug JVM Arguments

java -jar 命令启动时直接在后面跟上jvm参数

部署到Tomcat时可以在Tomcat的catalina.sh中设置Tomcat的Jvm参数，使用Springboot也可以在启动时指定Jvm参数。

1. -Xms: java堆内存大小
2. -Xmx: java堆内存最大大小
3. -Xmn: java堆内存中新生代大小，扣除新生代剩下的就是老年代内存大小
4. -XX:PermSize 永久代大小 (jdk1.8后) -xx:MetaspaceSize
5. -XX:MaxPermSize 永久代最大大小 (jdk1.8后) -xx:MetaspaceSize
6. -Xss: 每个线程的栈内存大小
7. -XX:NewSize 年轻代大小
8. -XX:NewRatio=4: 设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1: 4，年轻代占整个堆栈的 1/5
-XX:SurvivorRatio=4: 设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6
9. -XX: InitiaHeapSize 堆的大小
10. -XX: MaxTenuringThreshold (最大任期阈值) 最大年龄 15
11. -XX: PretenureSizeThreshold (新生代大小阈值) 超过直接进入老年代

- -XX:+CMSParallelInitialMarkEnabled表示在初始标记的多线程执行，减少STW；
- -XX:+CMSScavengeBeforeRemark：在重新标记之前执行minorGC减少重新标记时间；
- -XX:+CMSParallelRemarkEnabled：在重新标记的时候多线程执行，降低STW；
- -XX: CMSInitiatingOccupancyFraction=92和-XX:+UseCMSInitiatingOccupancyOnly配套使用，如果不设置后者，jvm第一次会采用92%但是后续jvm会根据运行时采集的数据来进行GC周期，如果设置后者则jvm每次都会在92%的时候进行gc；
- -XX:+PrintHeapAtGC：在每次GC前都要GC堆的概况输出



1.10 元空间

Java8 取消了PermGen。取而代之的是MetaSpace，方法区在java8以后移至MetaSpace。Jdk8开始把类的元数据放到本地内存（native heap），称之为MetaSpace

理论上本地内存剩余多少，MetaSpace就有多大，当然我们也不可能无限制的增大MetaSpace，需要用`-XX:MaxMetaSpaceSize`来指定MetaSpace区域大小。

关于used capacity committed 和reserved，在stackoverflow找到个比较靠谱的答案，我尝试翻译一下：MetaSpace由一个或多个Virtual Space（虚拟空间）组成。虚拟空间是操作系统的连续存储空间，虚拟空间是按需分配的。当被分配时，虚拟空间会向操作系统预留(reserve) 空间，但还没有被提交(committed)。

MetaSpace的预留空间(reserved) 是全部虚拟空间的大小。虚拟空间的最小分配单元是MetaChunk（也可以说是Chunk）。

当新的Chunk被分配至虚拟空间时，与Chunk相关的内存空间被提交了(committed)。MetaSpace的committed指的是所有Chunk占有的空间。

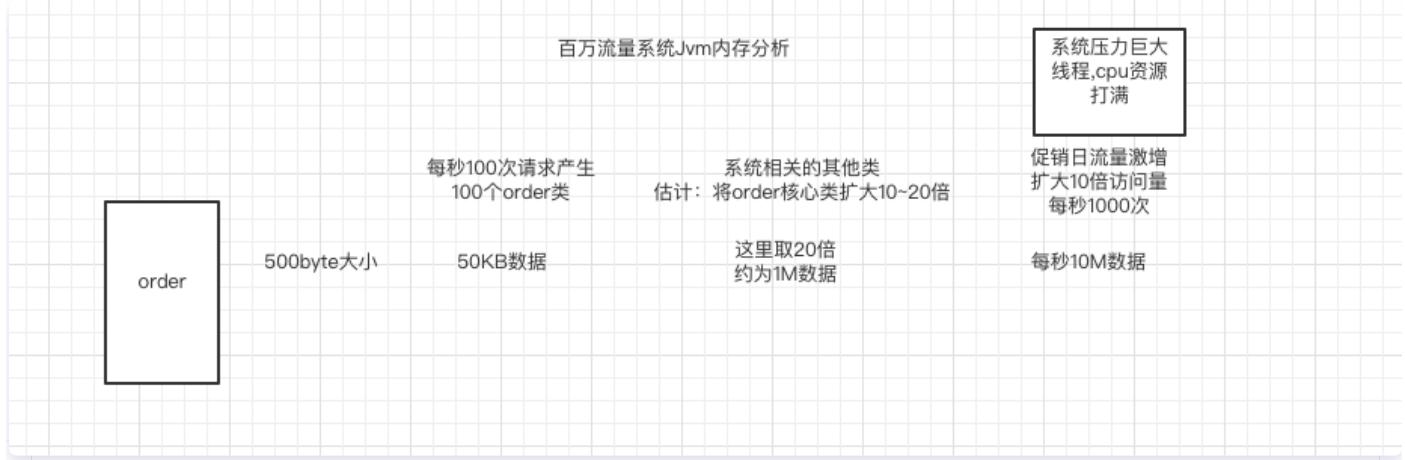
每个Chunk占据空间不同，当一个类加载器(Class Loader) 被gc时，所有与之关联的Chunk被释放(freed)。这些被释放的Chunk被维护在一个全局的释放数组里。

MetaSpace的capacity指的是所有未被释放的Chunk占据的空间。这么看gc日志发现自己committed是4864K, capacity4486K。有一部分的Chunk已经被释放了，代表有类加载器被回收了

附上原文链接：

<https://stackoverflow.com/questions/40891433/understanding-metaspace-line-in-jvm-heap-printout>

2. 百万流量订单系统预估



现在一秒可能处理不了1000个订单了，因为压力骤增，系统性能下降，可能偶尔会出现某个请求处理完毕需要几秒甚至几十秒的时间，这时频繁触发gc回收机制，而一些请求处理的特别慢，就会进入老年期，老年期也越来越多，也会频繁触发老年期的垃圾回收，老年期的垃圾回收非常慢，极大影响系统性能。

永久代大小: 一般 几百MB够用。

栈内存大小: 一般默认 512KB~1Mb

3. 垃圾回收

被哪些变量引用的对象是不能回收的？

可达性分析算法: 每个对象都分析一下有谁在引用它，然后一层一层往上判断，看是否有一个 GC Roots

方法的局部变量，类的静态变量都可以看做是一种 GC roots。

总结： 对象被 **方法的局部变量, 类的静态变量** 给引用，就不会回收。

java中对象不同的引用类型

参考链接:<http://www.cnblogs.com/dolphin0520/p/3784171.html>

- 强引用

如果内存不足，JVM会抛出OOM错误也不会回收object指向的对象。

(B b = new B()) b 对 B 就是强引用

- 软引用 (内存不足回收)

软引用是用来描述一些有用但并不是必需的对象，只有在内存不足的时候JVM才会回收该对象。因此，这一点可以很好地用来解决OOM的问题，并且这个特性很适合用来实现缓存：比如网页缓存、图片缓存等

```
SoftReference sr = new SoftReference( new String( "hello" ));
```

sr 对 String对象的引用就是软引用

- 弱引用 (都会被回收)

弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。

```
WeakReference sr = new WeakReference(newString("hello"));
```

- 虚引用 (PhantomReference)

基本不用。

它并不影响对象的生命周期，如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。

要注意的是，虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

如何利用软引用和弱引用解决OOM问题？

假如有一个应用需要读取大量的本地图片，如果每次读取图片都从硬盘读取，则会严重影响性能，但是如果全部加载到内存当中，又有可能造成内存溢出，此时使用软引用可以解决这个问题。

设计思路是：用一个HashMap来保存图片的路径 和 相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占用的空间，从而有效地避免了OOM的问题。在Android开发中对于大量图片下载会经常用到。

finalize()方法的作用

没有Gc roots 引用的对象可以回收，有 GC Roots 的对象不能回收，如果有 Gc roots 引用，但是如果是软引用或者弱引用也有可能被回收掉。

假设没有Gc roots引用的对象是一定立马被回收吗？

finalize () 方法可以拯救。

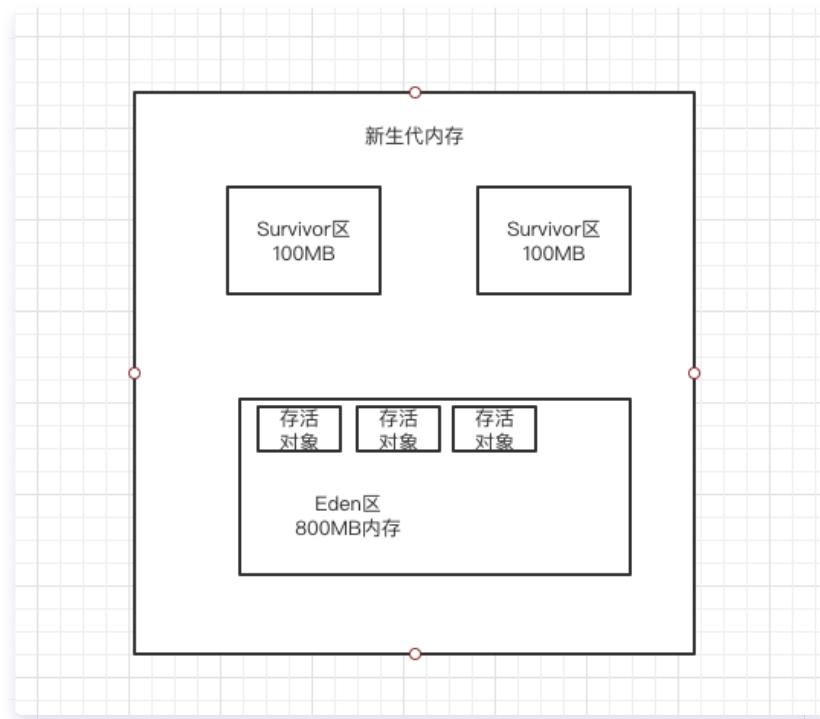
加入某个对象要被垃圾回收了，但是这个对象重写了Object类中的 finalize()方法并且这个方法中把自己给某个Gc roots变量。

```
public class ReplicaManager {  
    public static ReplicaManager instance;  
    @Override  
    protected void finalize() throws Throwable {  
        ReplicaManager.instance = this;  
    }  
}
```

重新让instance这个 gc roots 变量引用了自己。就不用被垃圾回收了。

3.1 新生代的垃圾回收算法

3.1.1 复制算法：Eden区和Survivor区



一个 Eden区，两个survivor区，eden区占80%内存空间，每一块survivor区占 10%。

平时使用一块eden和一块survivor区所以内存使用率为 90%。

刚开始对象都分配在eden区，如果eden区快满了就触发垃圾回收，把eden区中的存活对象转移到一块空着的survivor区，eden区清空，然后再次分配新对象到eden区，再触发垃圾回收，就把eden区存活的和survivor区存活的转移到另一块空着的survivor。

这么设计的原因：每次垃圾回收可能存活下来的对象就1%，如果eden+一块survivor满了900MB，一次垃圾回收下来有10MB存活，就把10MB转移到另一块survivor区。始终保持一块survivor区是空着的。这样可以控制内存碎片，而且内存的使用率都很高。

遗留问题：

- 万一存活下来的对象超过10%内存空间，在另外一块survivor区放不下怎么办？
- 万一突然分配超级大的对象，大到新生代找不到连续的空间来存放，怎么办？
- 到底一个存活对象要在新生代来回倒腾多少次才会去老年代？

解答

比如 static B b = new B();

静态变量b会一直引用B对象，这类对象不会被回收掉，每在新生代里躲过一次gc被转移到一块Survivor区时，年龄就长一岁。

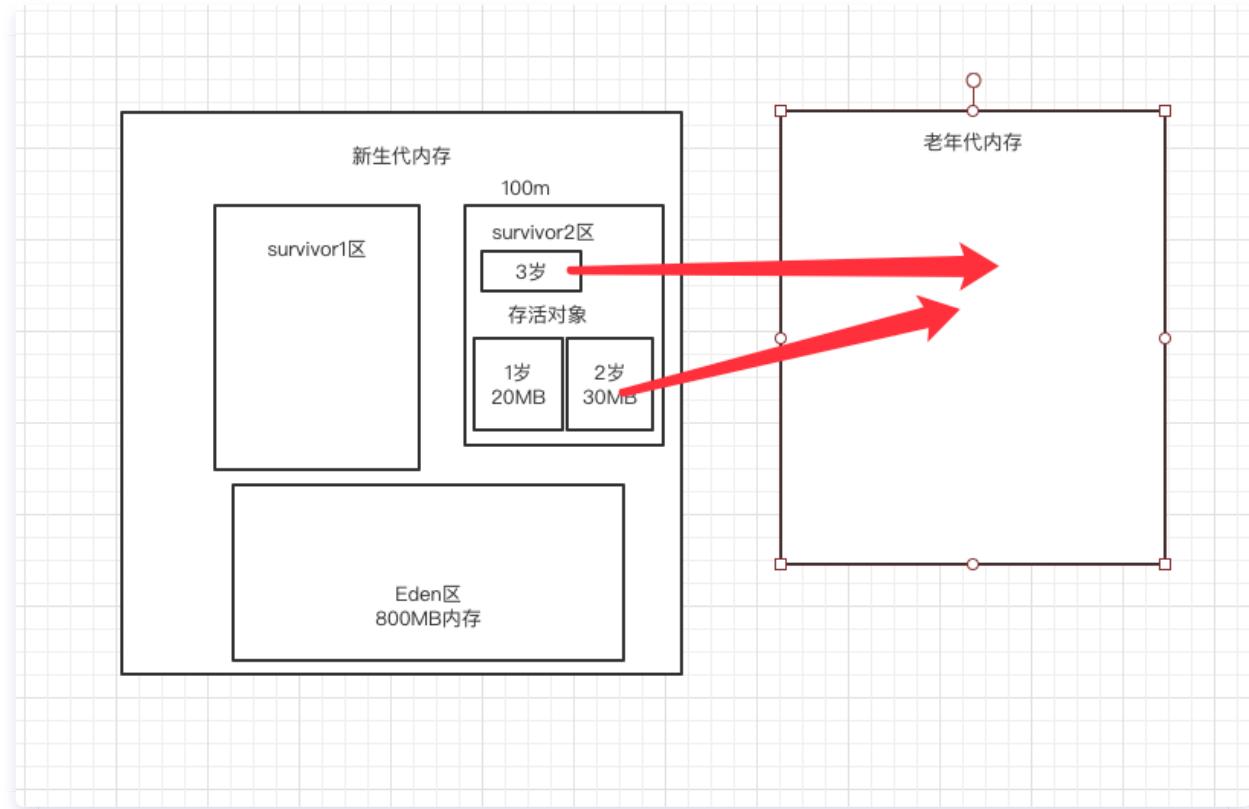
默认的设置是达到15岁时转移到老年代。

也可以通过jvm参数（年龄阈值）"-XX:MaxTenuringThreshold"设置

3.1.2 动态对象年龄判断

有另一个规则可以让对象早点进入老年代：[动态对象年龄判断](#)

触发时机：发生Minor Gc后，将存活的对象移动到空闲的 Survivor区时触发



图里一岁和二岁的对象加起来 \geq survivor区的一半50MB，那么survivor2区里年龄大于等于2岁的对象要提前进入老年代。

3.1.3 大对象直接进入老年代

有一个jvm参数 "-XX: PretenureSizeThreshold" 可以把他的值设为字节数。比如 1048576字节，就是1MB

如果你创建了一个大于这个大小的对象，比如一个超级大的数组，就直接把这个大对象放到老年代里。不会经过新生代。

之所以这么做，就是要避免新生代出现大对象，然后屡次躲过GC,还要把他在两个Survivor区域里来回复制多次之后才进入老年代。

3.1.4 空间分配担保机制

survivor区内存比较小，所以尽可能保证在一次Minor gc后，如果survivor区放不下，老年代要放得下，所以在Minor gc前要先计算老年代的可用空间够不够，能不能兜底。

抛几个问题：

1.什么是空间分配担保？

在发生 [Minor GC](#) 之前，虚拟机会检查 [老年代最大可用的连续空间](#) 是否大于 [新生代所有对象的总空间](#)，

如果大于，则此次 [Minor GC](#)是安全的

如果小于，则虚拟机会查看 [HandlePromotionFailure](#) 设置值是否允许担保失败。如果HandlePromotionFailure=true，那么会继续检查老年代最大可用连续空间是否大于 [历次晋升到老年代的对象的平均大小](#)，如果大于，则尝试进行一次Minor GC，但这次Minor GC依然是有风险的；如果小于或者HandlePromotionFailure=false，则改为进行一次Full GC。

2.为什么要进行空间担保？

是因为新生代采用 **复制收集算法**，假如大量对象在Minor GC后仍然存活（最极端情况为内存回收后新生代中所有对象均存活），而 Survivor空间是比较小的，这时就需要老年代进行分配担保，把Survivor无法容纳的对象放到老年代。**老年代要进行空间分配担保，前提是老年代得有足够的空间来容纳这些对象**，但一共有多少对象在内存回收后存活下来是不可预知的，因此只好取之前每次垃圾回收后晋升到老年代的对象大小的平均值作为参考。使用这个平均值与老年代剩余空间进行比较，来决定是否进行Full GC来让老年代腾出更多空间。

问题：Minor Gc后的对象太多无法放入Survivor区怎么办？

假如在发生gc的时候，eden区里有150MB对象存活，而Survivor区只有100MB，无法全部放入，这时就必须把这些对象全部直接转移到老年代里。

问题：接着上面的问题，如果这时老年代的可用内存小于新生代全部对象大小，万一Minor gc后新生代的对象都存活下来，然后需要全部转移到老年代，但是老年代空间不够，怎么办？

理论上有这个可能。

这时如果设置了 "-XX:-HandlePromotionFailure" 的参数，就会尝试判断，看老年代内存大小是否大于之前每一次Minor gc后进入老年代的对象的平均大小。

比如说，之前Minor gc 平均10M左右的对象进入老年代，此时老年代可用内存大于10MB,那么大概率老年代空间是足够的。

如果上面那个判断失败，或者是根本没设置这个参数，那就直接触发"Full GC"，对老年代进行垃圾回收，腾出些空间，再Minor gc。

如果判断成功了，那么大概率老年代内存是够的，就冒风险尝试Minor gc。这时有以下几种可能。

- Minor Gc 后，剩余的存活对象大小，小于Survivor区，那就直接进入Survivor区。
- Minor Gc 后，剩余的存活对象大小，大于Survivor区，小于老年代可用内存，那就直接去老年代。
- Minor Gc后，大于Survivor，老年代，很不幸，就会发生"Handle Promotion Failure"的情况，触发"Full GC"。

Full gc 就是对老年代进行垃圾回收，同时也一般会对新生代进行垃圾回收。

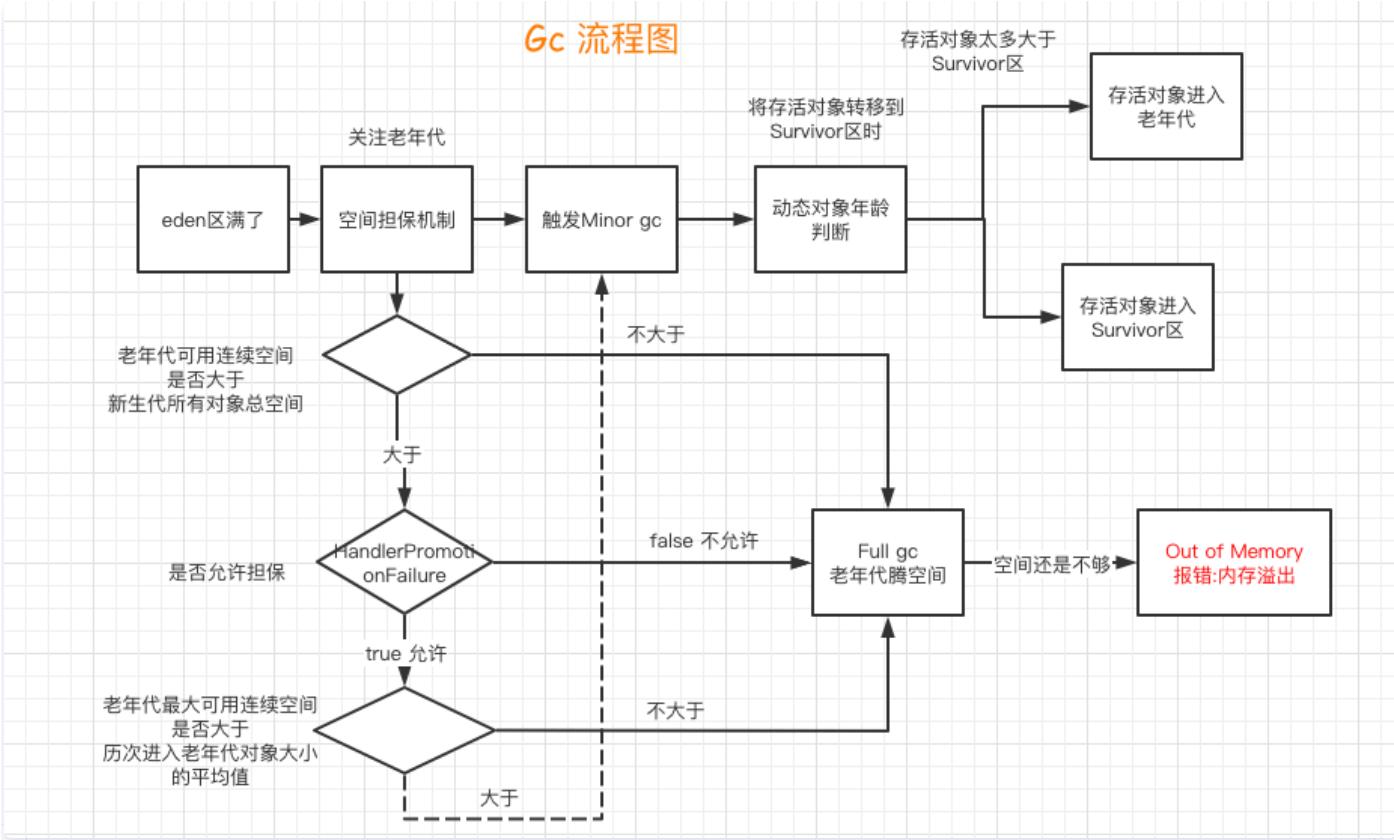
如果 Full gc后老年代还是没有足够的空间存放剩余的存活对象，那么就会导致 "**OOM**" out of memory 内存溢出。

所以Minor gc 触发要先对老年代空间做检查看看老年代空间够不够。检查失败的时候触发"Full Gc"给老年代腾空间，或者Minor gc 后剩余对象太多放入老年代内存都不敢，也要触发"Full Gc"。

总结：触发老年代垃圾回收 (Full gc) 的时机

- Minor gc 前，检查一下发现之前的 Avg (进入老年代对象的大小) > 现在老年代可用空间，提前触发Full gc。
- Minor gc 后，发现剩余对象 > 老年代可用空间，触发Full gc。

一张图总结一下



根据上图回答问题

- 什么时候会触发Minor gc?
- 触发Minor gc 之前会如何检查老年代的大小，涉及哪几个步骤和条件?
- 什么时候在Minor Gc 之前就会提前触发一次 Full gc?
 - 年轻代对象大小 > 老年代可用内存 && 没开通内存分配担保情况
 - 年轻代对象大小 > 老年代可用内存 && 开通内存分配担保情况下 历次年轻代GC进入老年代大小平均值 > 老年代可用内存大小
 -
 - 老年代已用内存空间超过 "-XX: CMSInitiatingOccupancyFraction" 参数指定的比例，自动触发Full gc
- Minor gc 过后可能对应那几种情况?

Minor gc 存活对象总大小 = M.memory Survivor区可用空间 = S.memory 老年代可用连续空间 = O.memory

- M.memory < S.memory M.memory 移迁至 Survivor区后触发（大龄对象判定，动态年龄判定）视情况移入老年代，如果老年代可用连续空间不够，先 Full gc 再 Minor Gc 还不够，报OOM。
- S.memory < M.memory < O.memory 存活对象直接迁移至老年代中
- O.memory < M.memory 先Full gc 再Minor gc，老年代空间还是不够，报OOM。

- 哪些情况下Minor gc 的对象会进入老年代?

Minor gc 后，存活对象太多，Survivor区放不下。存活对象直接进入老年代。

Stop the World 问题

JVM最让人无奈的痛点: 在垃圾回收时，jvm发送“stop the world”让java后台程序停止运行以免产生新对象，专心垃圾回收。

3.2 年轻代垃圾回收器

ParNew (多线程)

-XX:+UseParNewGC 指定使用parnew

默认情况下 ParNew线程数量和cpu的核数一样

可以通过 -XX:ParallelGCThreads 设置线程数量

Serial (单线程)

什么时候用parnew ? 什么时候用 Serial ?

启动系统的时候可以区分客户端模式和服务器端模式。

系统如果部署在服务器上，就应该用服务器模式，如果你的系统是运行在比如Windows上的客户端程序，就应该是客户端模式。

部署在服务器上的话，应该使用Parnew，因为多线程并行垃圾回收，充分利用多核CPU资源。提升性能

是客户端程序的话，很多都是单核cpu，此时如果还要用 ParNew来回收的话，就会导致一个cpu运行多个线程，反而加重了性能开销，因为单cpu运行多线程会导致频繁的上下文切换。

3.3 老年代垃圾回收的算法

对象进入老年代的条件

- 对象在年轻代躲过15次垃圾回收，年龄太大，进入老年代
- (动态年龄判定规则) Survivor 区域中的对象占用了超过50%的内存，此时判断如果 年龄1+年龄2+年龄n 的对象总和超过了survivor区域的 50%，此时年龄n及以上的对象都进入老年代
- 对象太大，超过一定阈值，直接进入老年代
- 一次 young gc 后存活对象太多了，导致survivor区域放不下，这批对象会进入老年代

一旦老年代对象过多，Juin可能会触发Full gc ,Full gc 必然会带着 old gc ，而且一般会跟着一次 young gc ，也会触发永久代的Gc。

触发老年代垃圾回收的条件

- -XX: CMSInitiatingOccupancyFraction 设置阈值，当老年代内存使用达到这个阈值时触发
- 在执行 young gc 之前判断，老年代可用空间<历次进入老年代的平均对象大小。那么就会在这次young gc前进行 Full gc ,先回收掉老年代一批对象，然后再执行Young Gc
- young gc 过后存活对象太多，survivor区放不下，就要进入老年代，老年代放不下就会 Full gc ,回收掉一批对象，再放入老年代。

标记清理算法: 先把老年代中存活的对象打标记，然后把标记的存活对象挪到一边，然后再清空非存活对象的那一边。

先通过追踪 GC Roots 的方法，看看各个对象是否被Gc Roots给引用了，如果是的话，就是存活对象，否则就是垃圾对象。将垃圾对象标记出来，然后一次性回收掉。

3.4 老年代垃圾回收器

CMS 采用标记清理算法

CMS 采取 垃圾回收线程和系统工作线程尽量同时执行的模式来处理

如何实现？

- **初始标记**

这个阶段，系统的工作线程全部停止，进入"stop the world",标记出所有的Gc roots对象。虽然要 stw,但是影响不大，因为速度非常快。

- **并发标记 (Gc roots 深度追踪)**

系统线程可以随意创建各种新对象，继续运行，这个阶段可能会创建新的存活对象，也可能部分存活对象失去引用。在这个过程中，垃圾回收线程会尽可能的对已有的对象进行 GC Roots追踪。

简而言之，会对老年代所有对象进行GC Roots 追踪，是最耗时的。被方法区静态变量引用的类其中的实例变量引用的类这一阶段会被标记存活。

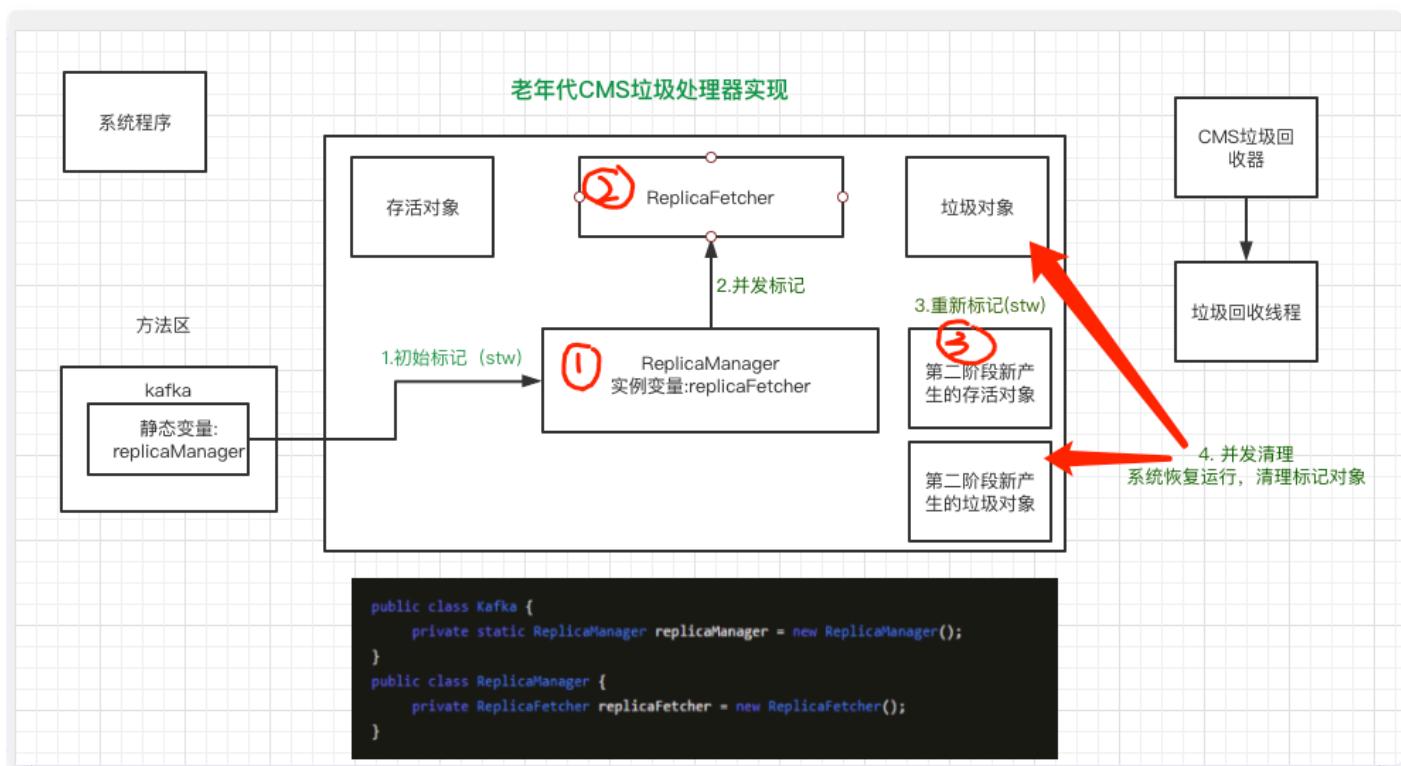
- **并发预清理阶段**：查找前一阶段执行过程中,从新生代晋升或新分配或被更新的对象。通过并发地重新扫描这些对象，预清理阶段可以减少下一个stop-the-world 重新标记阶段的工作量。
- **并发可中止的预清理阶段**：这个阶段其实跟上一个阶段做的东西一样，也是为了减少下一个STW重新标记阶段的工作量。增加这一阶段是为了让我们可以控制这个阶段的结束时机，比如扫描多长时间（默认5秒）或者Eden区使用占比达到期望比例（默认50%）就结束本阶段。

- **重新标记**

因为第二阶段结束，会多出许多之前没标记的存活对象和垃圾对象，所以再次进入"stop the world",然后重新标记在第二阶段新创建的一些对象和失去引用变成垃圾的对象。这个阶段速度很快。需要注意的是，虽然CMS只回收老年代的垃圾对象，但是这个阶段依然需要扫描新生代，因为很多GC Root都在新生代，而这些GC Root指向的对象又在老年代，这称为“跨代引用”。

- **并发清理**

系统恢复运行，然后清理之前标记的垃圾对象。很耗时，但是是和程序并发运行，所以不影响系统的运行。



简单来说，为了避免长时间的"Stop the World" ,CMS采用了4个阶段来垃圾回收，其中初始标记和重新标记，耗时短，虽然会导致stw，但是影响不大，然后并发标记和并发清理，两个耗时最长，但是可以跟系统的工作线程并发运行，所以对系统影响不大。

这就是CMS的基本工作原理。

CMS 的弊端：

- **消耗cpu资源**
- **Concurrent Mode Failure (并发模式失败)**

并发清理的时候，系统一直在运行，可能会随着系统的运行让一些对象进入老年代，变成垃圾对象。这种垃圾对象----浮动垃圾。需要等到下一次 Full Gc清理他们。所以为了保证在CMS垃圾回收期间还有一定的空间让一些对象进入老年代，一般会预留一些空间。

"-XX: CMSInitiatingOccupancyFraction" 参数设置老年代占用多少比例时触发CMS垃圾回收。

jdk1.6 默认是 92%

如果在这期间，进入老年代的对象大于可用内存空间，那么会触发 Concurrent Mode Failure，并发垃圾回收失败。这时会自动用"Serial Old" 垃圾回收器替代CMS。就是直接把系统"Stop the World" 重新进行长时间的Gc Roots追踪，标记出全部垃圾对象，不允许新的对象。然后一次性回收，再恢复系统。

- 内存碎片

老年代CMS 采用标记清理算法，标记垃圾，回收，会产生大量的内存碎片。太多的内存碎片会导致频繁的Gc。

"-XX: +UseCMSCompactAtFullCollection" 参数默认打开。

表示在 Full Gc后要再次进行"Stop the world" 停止工作线程，再进行碎片整理，把存活的对象挪到一边，空出大片连续的内存空间。

"-XX: CMSFullGCsBeforeCompaction"

执行多少次 Full Gc 之后再执行内存碎片整理工作。默认是 0，每次都需要。

为什么老年代的 Full GC 比新生代的 Minor GC 慢很多倍？一般在10倍以上？

ParNew的Minor GC

新生代一般存活对象少，采用复制算法，从Gc root出发标记存活对象，直接把存活对象复制到另一块内存，其余直接清除。

新生代执行速度快，因为直接从 Gc Roots 出发，追踪哪些对象是存活的就行，新生代存活的对象是很少的。然后直接放入Survivor区，就一次性回收eden区和之前的Survivor区。

CMS的Full GC

老年代，对象存活量大，每次遍历堆分别去标记存活对象和垃圾对象，再遍历把垃圾对象清除，最后还要一定存活对象，防止太多内存碎片。

- 在并发标记阶段，需要去深度追踪所有的存活对象，老年代存活对象很多。
- 并发清理阶段，也不是一次性回收一大片内存，而是找到零零散散在各个地方的垃圾对象。
- 最后还要执行内存碎片整理，把存活对象移一起，空出连续空间，这个过程还得 "Stop the world"
- 并发清理期间，剩余空间不足存放新进的对象时，还会触发"Concureent Mode Failure"，更加麻烦，还要使用"Serial Old"单线程的垃圾回收器，"Stop the world"后再重新来一遍回收过程。更加耗时。

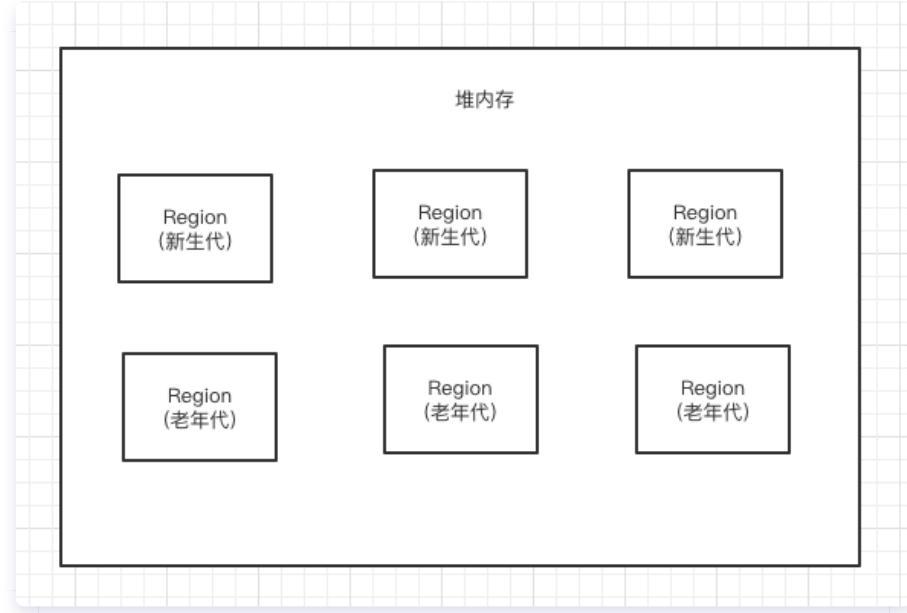
3.5 最新的 G1 垃圾回收器

Parnew + CMS 的痛点：无论是新生代还是老年代的垃圾回收，都会或多或少产生"stop the world"，对系统运行有一定影响。而且不是可控的，只能优化。

3.5.1 G1垃圾回收器概述

G1 垃圾回收器可以同时回收新生代和老年代的对象，一个人负责所有。

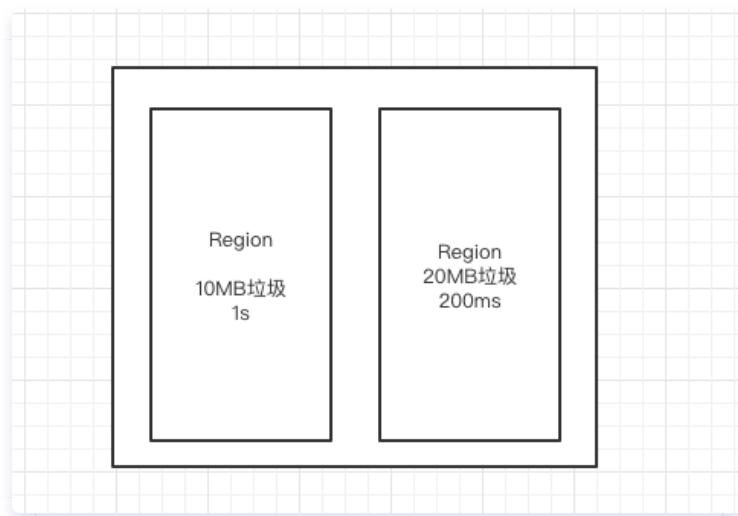
内存结构：把java堆内存拆分为多个大小相等的Region,新生代可能包含了某些块，老年代可能包含某些块。



特点: 可以设置垃圾回收的预期停顿时间

比如我们可以指定G1 在垃圾回收时候可以保证，在一小时内垃圾回收导致的"stop the world"时间不超过一分钟。

实现原理: 追踪每个 Region里的 **回收价值** (耗费时间, 垃圾对象大小)



假设 G1通过追踪发现，1个Region中的垃圾对象由10m，回收需要1s，另一个垃圾对象20m，回收需要200ms。下一次执行垃圾回收时，回收掉上图中只需要200ms就能回收200mb垃圾的Region。

核心设计思路: 简单来说，G1可以做到让你来设定垃圾回收对系统的影响，通过把内存拆分为大量小Region,追踪每个Region可以回收对象的大小和预估时间，最后在垃圾回收的时候，尽量把垃圾回收对系统造成的影响控制在指定的时间范围内。有限时间内回收尽可能多的对象。

3.5.2 设定内存大小

用 "-Xm" 和 "-Xmx" 设置堆内存的大小。

"-XX: +UseG1GC" 指定使用G1 垃圾回收器。

Jvm自动用堆大小除以2048

Jvm最多可以有 2048个Region ,Region的大小必须是2的倍数。比如1MB、2MB、4MB

比如堆大小是 4G=4096MB 除以 2048 那么每个Region大小是2MB。

一般保持这种默认的计算方式。如果通过手动方式指定: "-XX:G1HeapRegionSize"

刚开始的时候， 默认新生代对堆内存的占比是5%,也就是200MB左右的内存， 大概100个Region。

可以通过 "-XX: G1NewSizePercent" 设置新生代初始占比。

在系统运行中， Jvm会不停地给新生代增加更多的Region,但是最多新生代的占比不会超过 60%。

可以通过 "-XX: G1MaxNewSizePercent"改变。

新生代还是有 Eden 和 Survivor 划分的。

"-XX: SurvivorRatio=8", 区分eden和 survivor比例。

比如新生代一开始有100个 Region, 那么80个Region是eden,两个Survivor各占10个。

3.5.3 新生代垃圾回收

既然新生代也有 eden 和 survivor , 那么触发垃圾回收的机制都是类似的。

Jvm不断的给新生代加入更多的Region,直到新生代占据堆大小的最大比例 60%。

新生代大概占据1000个region了， 还占满了对象， 每个Survivor是100个大小。

这时候触发 新生代的Gc, G1就用之前的复制算法进行垃圾回收， 进入"stop the world"。

eden中的存活对象放入 S1 的region中， 接着回收掉eden中的对象。但是这个过程是有区别的， 因为G1可以设定目标停顿时间。最多系统停顿时间。可以通过 "-XX: MaxGCPauseMills" 设定。默认值是 200ms。是每次" stop the world " 的停顿时间。

G1 就会通过之前说的， 追踪每个Region回收时间， 垃圾对象大小， 保证Gc 停顿时间控制在指定范围内， 尽可能多的回收一些对象。

3.5.4 老年代垃圾回收

老年代最多占据 40%的 Region,大概就是800个左右的Region。

对象进入老年代的条件

和之前几乎一模一样。

- 躲过多次垃圾回收， 达到一定年龄。

" -XX: MaxTenuringThreshold " 设置年龄， 进入老年代。

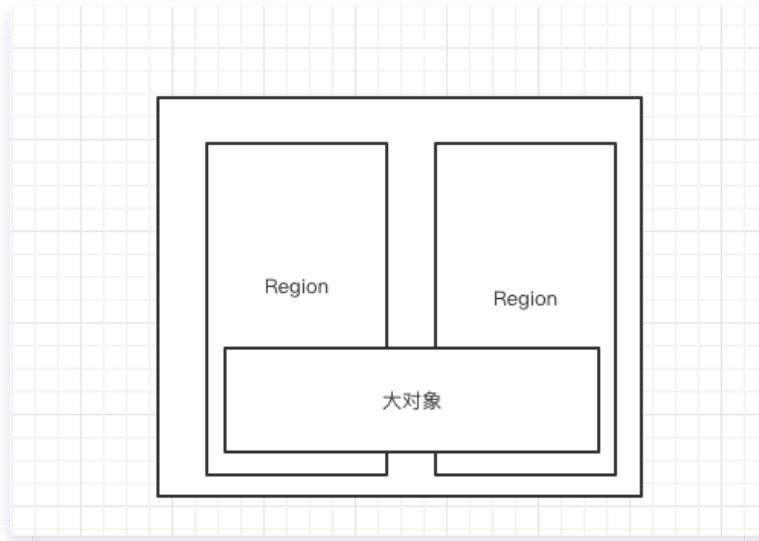
- 动态年龄判断规则,某次Gc后， 存活对象超过Survivor区的50%。

此时会判断， 比如1岁、2岁、3岁、4岁以上的对象大小综合超过了Survivor区的50%， 此时4岁以上的对象会全部进入老年代， 这就是动态年龄判断规则。

3.5.5 大对象回收分配策略

大对象不属于新生代和老年代，在G1里，新生代和老年代是动态的，不断变化的。比如一次垃圾回收后，eden里面的1000个region都空了，就可以让一些region存放大对象。

新、老年代在回收的时候，会顺带带着大对象Region一起回收。



3.5.6 混合垃圾回收 (Mixed-GC)

条件:

G1有一个参数（IHOP） "-XX: InitiatingHeapoccupancyPercent" 默认值 45%

如果 老年代占据了堆内存的 45%的Region,此时就会触发 新生代+老年代+大对象 一起回收的混合回收。

过程:

- **初始标记**: 进入"stop the world" ,就只标记Gc roots 直接引用的对象(线程栈内存中的局部变量, 方法区中的类静态变量)。速度很快。
- **并发标记**: 允许系统程序的运行, 同时进行"GC Roots"追踪, 追踪所有存活对象(间接引用的对象)。很耗时, 要追踪全部的存活对象。但是是并发运行, 对系统影响不大。Jvm还会对这个阶段, 一些对象的修改记录起来, 比如对象新建, 对象失去引用。
- **最终标记**: 进入"stop the world" , 根据并发标记记录的对象修改, 最终标记存活对象, 垃圾对象。
- **混合回收**: 计算老年代中每个Region的存活对象数量, 存活对象占比, 执行垃圾回收的预期性能和效率。然后停止系统运行, 全力以赴进行垃圾回收, 为了控制垃圾回收时间在指定的范围, 会对部分的region进行回收。
- Mixed GC过程中会将多个老年代Region中仍存活的小对象集中到一个Region中, 也就是说 Mixed GC会对老年代空间进行内存整理

ps: 在最后一个阶段"混合回收"的时候, 会停止程序运行, 但是为了不让程序过长时间停止, 可以多次执行"混合回收"。

比如说, 要回收100个Region先停止工作, 执行一次混合回收, 回收掉30个Region, 接着恢复系统运行, 然后再停止系统运行, 再执行一次混合回收, 回收掉30个。以此往复。有一个参数可以控制这个次数。

"-XX: G1MixedGCCountTarget" 默认值为8。

在一次混合回收中, 最后一个阶段执行几次。

"-XX: G1HeapWastePercent" 默认值5%。

也就是在全局标记结束后能够统计出所有Cset内可被回收的垃圾占整对的比例值, 如果超过5%, 那么就会触发之后的多轮Mixed GC, 如果不超过, 那么会在之后的某次Young GC中重新执行全局并发标记。可以尝试适当的调高此阈值, 能够适当的降低Mixed GC的频率。

空闲的Region数量达到老年代堆内存的5%就会停止回收, 比如正常是8次回收, 但是到第4次, 空闲Region达到5%了, 就不进行后续的混合回收了。

" -XX: G1MixedGCLiveThresholdPrcnt " 默认值 85% 。

存活对象低于85%的 Region才可以进行回收。

如果一个对象存活对象多余85%，要把这些对象拷贝到别的Region，成本很高。

Mixed-Gc回收失败：

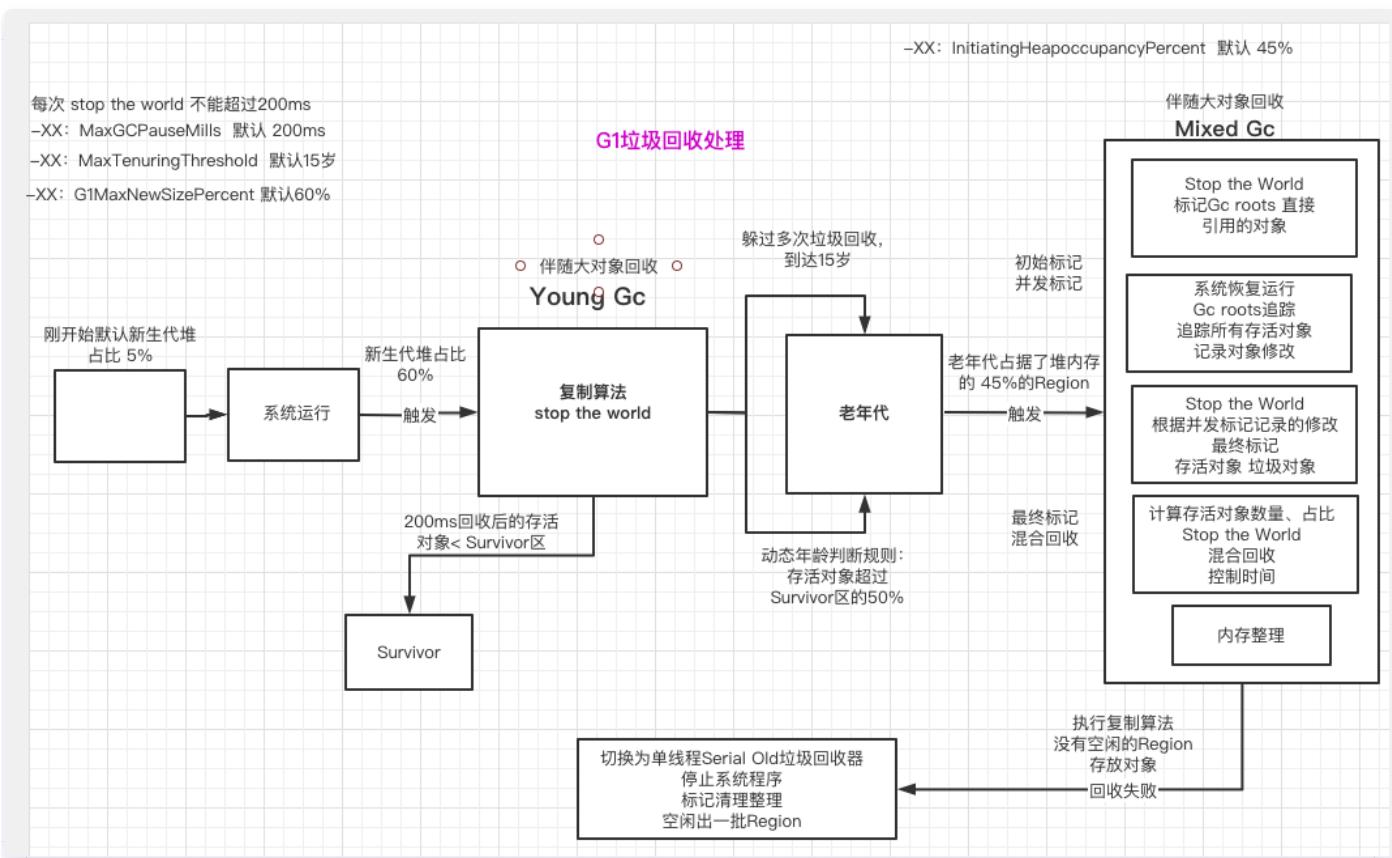
因为回收时，无论老年代还是年轻代执行复制算法，都要把存活对象拷贝到别的Region中，万一没有空闲的Region可以放存活对象了。就会失败。

一旦失败，立马切换为单线程垃圾回收，停止系统程序，进行标记、清理、整理。空闲出来一批Region。过程很慢。

3.5.7 总结

当新生代Region大于 60 %进行新生代垃圾回收时，如果200ms回收后的Region是小于 survivor区的大小的话就不需要进入老年代了。

当老年代 Region大于 45%的时候才会去触发 mixed gc。



为什么要选择 G1 垃圾处理器？

- G1在停顿时间上添加了预测机制，用户可以设置期望停顿时间，Stop The World时间相对可控。
- G1的年轻代和老年代空间并不是固定的，当现有年轻代分区占满时，JVM会分配新的空闲Region加入到年轻代空间，老年代也是如此。还记得在CMS中出现的年轻代空间有大量浪费的情况吗？那将不再是问题。另一方面，由于无法保证每台机器的吞吐完全一致，不同机器的GC压力也是不一样的，该特性使得系统进程可以根据情况自行调整年轻代和老年代的空间大小，以应对不同的GC压力
- G1 GC的回收过程中有内存整理，理论上不会产生内存碎片！
- G1很适合大内存的机器，因为如果是 ParNew+CMS，每次Gc都是内存快满了，此时一下子要回收对象太多，导致gc停顿时间太长。针对大内存机器，G1很适合。

如何尽量减少Mixed Gc 频率?

Mixed gc 触发条件是老年代到达 IHOP 设置的值。

- 让垃圾对象尽可能在新生代就被回收掉，让短命对象不进入老年代，这就要求根据具体应用系统来合理设置新生代Eden大小和Survivor的大小。
- 将新生代分配更大空间，老年代设置为较小值。但是如果有关较多需要长期存活的对象的话，容易导致Full Gc 和 OOM。
- 提高 IHOP 的值。虽然降低了Mixed Gc 频率，单、但导致老年代存在过多对象，增加了每次老年代回收时"并发标记"阶段的计算负担和"mixed gc"阶段计算和预估的负担。不适合CPU负载较高的计算型业务系统。

4.日处理上亿数据的系统(分配内存区域大小)

优化思路：

1.上线前，根据预期的并发量、平均每个任务的内存需求大小等，然后评估需要几台机器，需要怎样的配置

2.根据系统的任务处理速度，评估内存使用情况，然后合理分配Eden、Survivor,老年代的内存大小。

一句话总结：

尽量让每次 Young Gc 后的存活对象小于 survivor区的50%，都留存在年轻代中，尽量别让对象进入老年代。尽量减少 Full Gc的频率，避免频繁Full Gc对Jvm性能的影响。

总体原则：

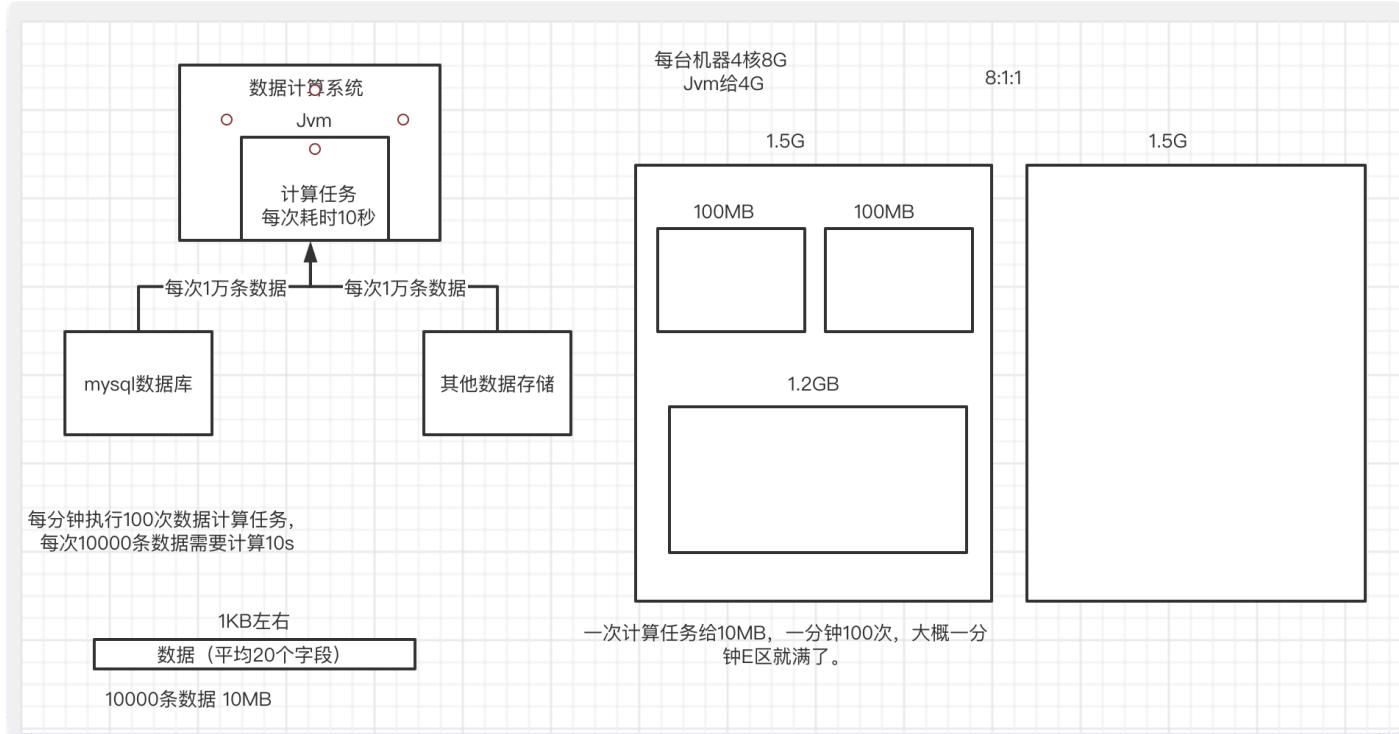
短命对象在young gc就被回收，不要进入老年代。长期存活的对象，尽早进入老年代，不要在新生代复制来复制去。对系统响应时间敏感且内存需求大的，建议采用G1回收器。

在给定的内存大小合理的前提下，合理设置 eden、survivor、老年代的内存比例，尽可能让存活的对象放入到survivor区（下次Minor Gc就回收，不进入老年代），延长两次 Full gc之间的时间。

评估指标：

- 根据内存增速来评估多久进行 young Gc
- 根据每次Young Gc的存活，评估Survivor区大小是否设置合理
- 评估多久进行一次"Stop the World"，是否可以接受。

要结合自己系统的运行，根据系统的内存占用情况，GC后的对象存活情况，合理分配 Eden、Survivor、老年代的内存大小，合理设置一些参数。



背景1:

一分钟之后，新生区快满了，新生代对象总共大概1.2G,而老年代可用内存为1.5G，即便一次Minor gc后，全部对象都存活，老年代也能放得下，直接执行 Minor Gc。

假设100个计算任务都结束，还有20个计算任务总共200M数据还在计算中，200MB的对象是存活的，1Gb的对象是回收掉的。

但是因为 Survivor区只有100Mb的大小，这些数据无法放入Survivor区，所以只能进入老年代中了，这样子大概7分钟过后，1.4G对象进入老年代，老年代剩余空间不到100MB了。

第8分钟运行结束，新生代满了，老年代也不够，直接触发 Full Gc。

平均7、8分钟一次Full Gc。性能很差。

性能优化1:

2GB给新生代 1GB给老年代 Survivor 大概200MB 刚好能放下Minor gc 后的对象了。

背景2:

这个中型电商系统在大促期间的瞬时高峰下单场景，假设大促高峰期间，每秒每台机器会有300个下单请求。每秒使用60MB内存，有一台4核8G的机器，应该如何合理给JVM各个区域分配内存。

20多秒 eden区满了 Minor Gc 剩下大约 100MB的存活对象，进入 Survivor区。

性能优化2:

-XX:MaxTenuringThreshold =5

这个参数让一两分钟内连续躲过5次Minor gc的对象迅速进入老年代。这种对象一般是一些@Service、@Controller之类的注解标注的系统业务逻辑组件。

发生 Full gc 的触发条件。

- 没有打开 -XX:HandlePromotionFailure 结果老年代内存可用内存也就1G,新生代对象总大小最大可以有1.8G,那么Minor gc 前，一检查，“老年代可用内存” < “新生代总对象”，导致每次Minor gc 前都触发Full gc。

jdk1.6以后废弃了 -XX:HandlePromotionFailure

只要满足 每次Minor gc前 老年代可用内存空间 < 历代Minor gc后升入老年代的平均对象大小

就可以进入老年代了。

- 老年代可用内存空间 > 历代Minor gc后升入老年代的平均对象大小
- 可能某次Minor gc 后要升入老年代的对象有几百MB,但是老年代可用空间不足。
- 设置了"-XX: CMSInitiatingOccupancyFraction" 参数, 老年代使用空间超过了设定的92%,自行触发 Full gc。

可能会因为上面的条件任何一个满足, 触发 Full gc。但是一般需要老年代几乎占满的时候, 才会触发。

思考: 这时老年代Full gc 会发生"Concurrent Mode Failure" ?

CMS在并发清理的时候, 系统程序是可以并发运行的, 假设老年代大概有900MB的对象, 剩余空间仅仅只有100MB了, 此时系统程序还在不停地创建对象, 万一这个时候系统运行触发了某个条件, 比如说有200MB对象要进入老年代。此时就会触发 Concurrent Mode Failure , 因为此时老年代没有内存来放下这200MB对象。这时就会立马"stop the world", 然后切换CMS 为 Serial Old。直接禁止程序运行, 单线程进行老年代垃圾回收, 回收掉 900MB后, 再让系统运行。

但是发生的概率是比较小的, 因为必须是 CMS 触发 Full gc的时候, 系统运行区间还让200MB对象进入老年代。

参数设置:

所以就保持默认的设置, 每次Full GC之后都执行一次内存碎片整理就可以, 目前JVM参数如下:

```
"-Xms3072M -Xmx3072M -Xmn2048M -Xss1M -  
XX:PermSize=256M -XX:MaxPermSize=256M -  
XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=5 -  
XX:PretenureSizeThreshold=1M -XX:+UseParNewGC -  
XX:+UseConcMarkSweepGC -  
XX:CMSInitiatingOccupancyFraction=92 -  
XX:+UseCMSCompactAtFullCollection -  
XX:CMSFullGCsBeforeCompaction=0"
```

今日分享: 线上机器CPU负载过高如何去排查? 如何优化JVM参数?

(1) 第一可以通过 top -Hp PID , jstack PID等查看占用CPU资源过高的线程是哪些? 可能会看到是GC线程。

或者是其他线程, 如果是GC线程, 那么jstat查看JVM垃圾回收器工作的情况。如果是其他业务线程, 可能需要跟踪栈信息, 追踪到代码中进行分析。(可能是一直循环处理业务, 数据量大, 处理耗时。)

-- 业务线程导致CPU高的场景, 没有关注过, 不知道我的理解对不对。

(2) 查看JVM垃圾回收情况相关的信息

YoungGC频率, YoungGC耗时, 每次GC过后Eden+S0区的垃圾回收情况; 进入老年代的大小

FullGC频率, YoungGC耗时, 每次GC过后老年代的垃圾回收情况;

我的总结:

(1) 如果是YoungGC，FullGC频繁，但是每次FullGC回收之后，垃圾回收率很高，可能就是高并发引起的；要关注下是不是年轻代内存分配不合理，是否需要加大JVM堆内存。升级机器等。根据YoungGC每次进入老年代的大小，重新预估一下，新生代需要分配多大的内存比较合适，来解决这个高并发引起的FullGC问题。

(2) 如果FullGC之后，回收的内存时多时少，或者说很少，甚至可能出现连续的1-3次，回收之后，老年代剩余空间是在增长的，这个时候就要排除是否有发生内存泄漏的嫌疑。或者说高并发引起的问题导致对象无法被回收。这时候一般需要dump下内存快照，使用MAT工具分析内存快照，可以短时间连续dump两次，对比两次内存快照，查看哪些对象不断在增长。这些对象是不是大对象，因为并发问题无法被及时回收，JVM处在OOM边缘。然后根据派排查结果，优化代码问题。

优化JVM参数：

(1) 优化新生代和老年代的占比。尽量保证每次YoungGC之后。可以打印对内存信息，计算一下一般存活对象是多大。假设是100M。那么可以将S0设置成 $S0 * 50\% = 100M$ 。S0大概需要200M。这样就可以把Eden和老年代大小预估出来了，一般业务系统。老年代分配1至2G就可以了。

(2) 需要开启CMS垃圾回收器的对内存碎片进行整理。一般每次FullGC都整理一遍最好。

(3) -XX:CMSInitiatingOccupancyFraction=92。老年代占用92%以上发生FullGC。一般这个值我觉得偏大了，如果是高并发系统，会有较大可能出现Concurrent Mode Failure。所以我对CMS设置的时候，会把老年代预留空间大小预估到S0区大小这样。一般设置80%。对于这点不知道是否合理？

(4) 还会设置一些额外的参数：-XX:-OmitStackTraceInFastThrow

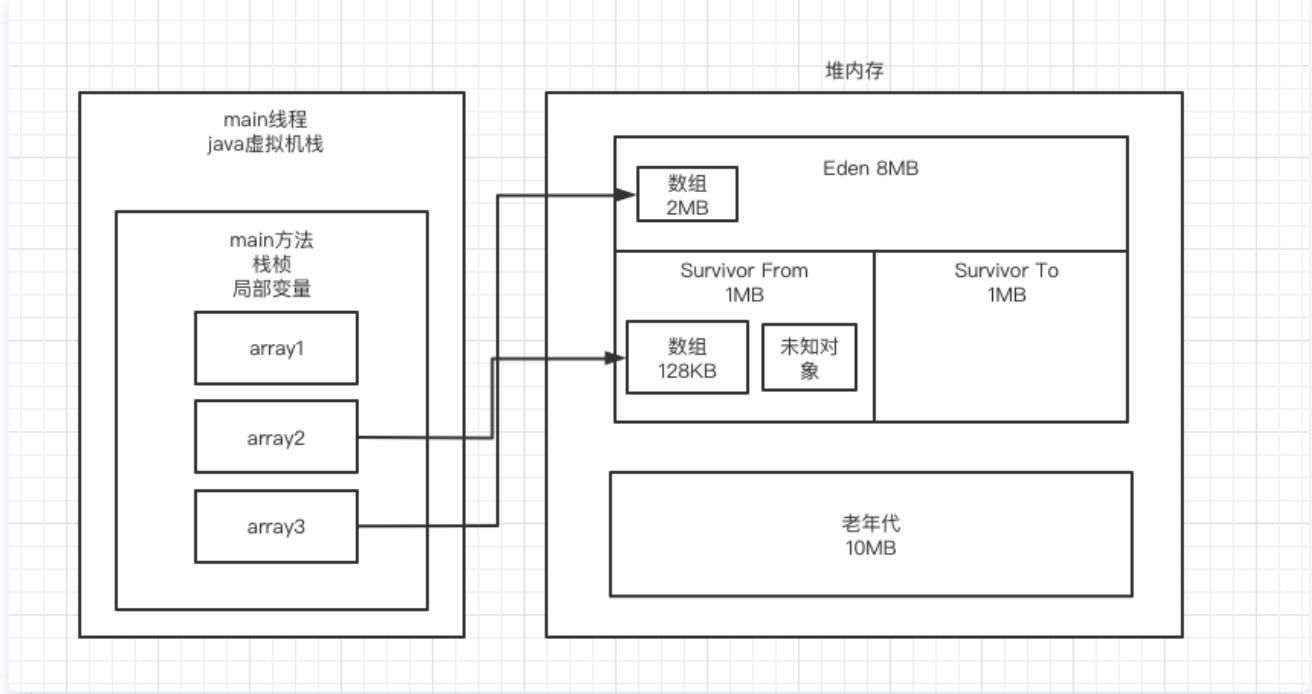
还有打印GC日志，发生OOM的时候dump内存快照参数。

(5) 还有一点，如果没有发生什么特殊的问题，不会对其他的参数进行优化。能简单设置就采用简单的设置。

下次分享：你说你精通jvm调优，能够说下G1工作原理以及整体流程吗？

5. 日志分析

```
public class Demo1 {  
    public static void main(String[] args) {  
        byte[] array1 = new byte[2*1024*1024];  
        array1 = new byte[2*1024*1024];  
        array1 = new byte[2*1024*1024];  
        array1 = null;  
  
        byte[] array2 = new byte[128*1024];  
        byte[] array3 = new byte[2*1024*1024];  
  
    }  
}
```



回收前的全体对象

```

CommandLine flags: -XX:InitialHeapSize=20971520 -XX:MaxHeapSize=20971520 -XX:MaxNewSize=10485760 -XX:MaxTenuringThreshold=15 -XX:NewSize=10485760 -XX:OldPLABSize=16 -XX:PretenureSize=16
0.131: [GC (Allocation Failure) 0.131: [ParNew: 7649K->502K(9216K), 0.0028100 secs] 7649K->502K(19456K), 0.0029567 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap   存活对象 +Array3的2MB对象   存活对象   eden+S_from   eden+S_from+老年代
par new generation total 9216K, used 2633K [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)
eden space 8192K, 26% used [0x00000007bec00000, 0x00000007bee14930, 0x00000007bf600000)
from space 1024K, 49% used [0x00000007bf500000, 0x00000007bf57d000, 0x00000007bf600000)
to   space 1024K, 0% used [0x00000007bf400000, 0x00000007bf400000, 0x00000007bf500000)
concurrent mark-sweep generation total 10240K, used 0K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)
Metaspace      used 2927K, capacity 4496K, committed 4864K, reserved 1056768K
class space     used 319K, capacity 388K, committed 512K, reserved 1048576K

```

5.1 年轻代gc日志分析

jvm参数:

```

-XX:NewSize=5242880 -XX:MaxNewSize=5242880 -XX:InitialHeapSize=10485760 -XX:MaxHeapSize=10485760 -
XX:SurvivorRatio=8 -XX:PretenureSizeThreshold=10485760 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:gc.log

```

情况一:

老年代垃圾回收，发生young gc。进入 Survivor区700kb存活对象，下一次young gc前执行动态年龄判断，这700kb对象进入老年代。

情况二:

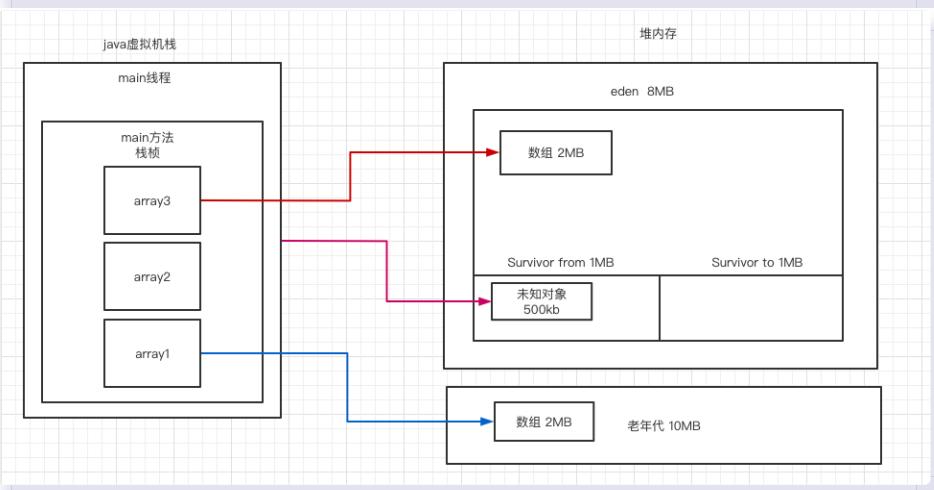
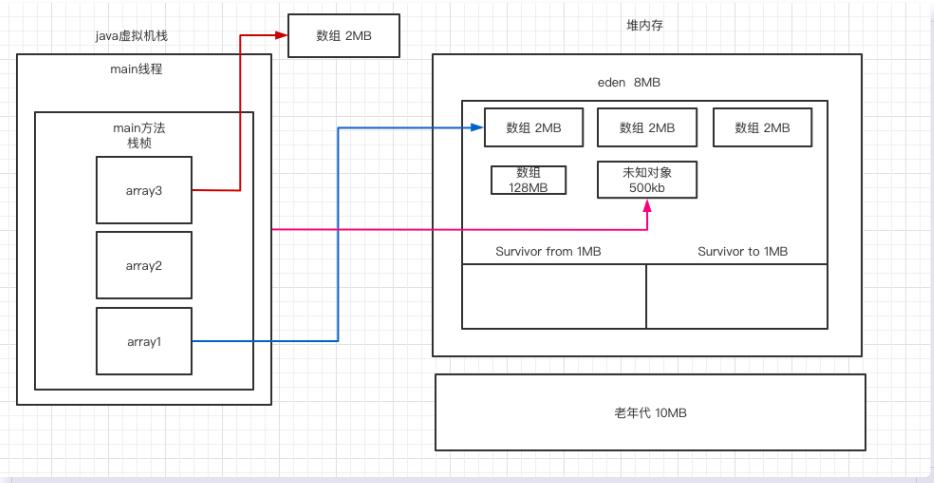
老年代垃圾回收，发生young gc。有2MB存活对象和 500kb的未知存活对象。survivor区只有1MB。这时，500kb未知存活对象进入Survivor区，2MB存活对象进入老年代。

```

public class Demo1 {
    public static void main(String[] args) {
        byte[] array1 = new byte[2*1024*1024];
        array1 = new byte[2*1024*1024];
        array1 = new byte[2*1024*1024];

        byte[] array2 = new byte[128*1024];
        array2 = null;
        byte[] array3 = new byte[2*1024*1024];
    }
}

```



如下是日志

```

CommandLine flags: -XX:InitialHeapSize=20971520 -XX:MaxHeapSize=20971520 -XX:MaxNewSize=10485760 -XX:MaxTenuringThreshold=15 -XX:NewSize=10485760 -XX:OldPLA
0.126: [GC (Allocation Failure) 0.126: [ParNew: 7649K->397K(9216K), 0.0034386 secs] 7649K->2447K(19456K), 0.0035683 secs] [Times: user=0.01 sys=0.00, real=0
Heap
par new generation total 9216K, used 2609K [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)
eden space 8192K, 27% used [0x00000007bec00000, 0x00000007bee29140, 0x00000007bf400000)
from space 1024K, 38% used [0x00000007bf500000, 0x00000007bf5635b0, 0x00000007bf600000)
to space 1024K, 0% used [0x00000007bf400000, 0x00000007bf400000, 0x00000007bf500000)
concurrent mark-sweep generation total 10240K, used 2050K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)
Metaspace      used 2953K, capacity 4496K, committed 4864K, reserved 1056768K
class space    used 327K, capacity 388K, committed 512K, reserved 1048576K

```

5.2 老年代gc日志分析

参数

```
-XX:NewSize=5242880 -XX:MaxNewSize=5242880 -XX:InitialHeapSize=10485760 -XX:MaxHeapSize=10485760 -  
XX:SurvivorRatio=8 -XX:PretenureSizeThreshold=3145728 -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails  
-XX:+PrintGCTimeStamps -Xloggc:gc.log
```

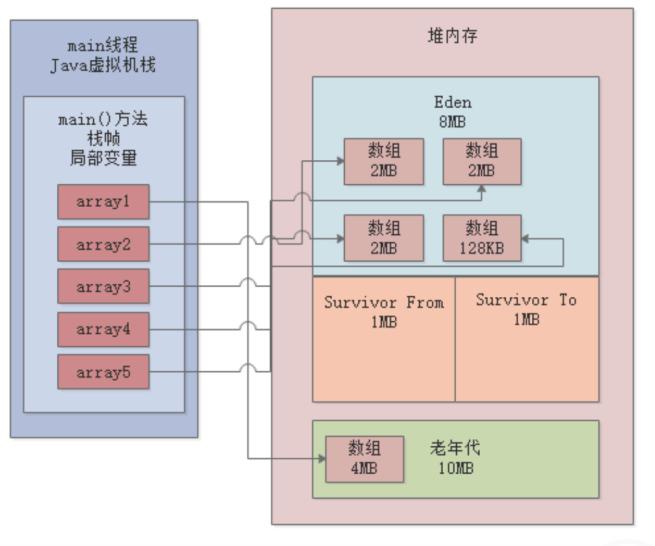
就更改了 -XX:PretenureSizeThreshold = 3145728 大于3MB的对象直接进入老年代。

情况一：

年轻代存活对象太多，老年代放不下。触发CMS的Full gc。

```
public static void main(String[] args) {  
    byte[] array1 = new byte[4*1024*1024];  
    array1 = null;  
    byte[] array2 = new byte[2 * 1024 * 1024];  
    byte[] array3 = new byte[2 * 1024 * 1024];  
    byte[] array4 = new byte[2 * 1024 * 1024];  
    byte[] array5 = new byte[128 * 1024];  
  
    byte[] array6 = new byte[2 * 1024 * 1024];  
}
```

内存模型如下



日志如下

```
CommandLine flags: -XX:InitialHeapSize=20971520 -XX:MaxHeapSize=20971520 -XX:MaxNewSize=10485760 -XX:MaxTenuringThreshold=15 -XX:NewSize=10485760  
0.142: [GC (Allocation Failure) 0.142: [ParNew (promotion failed): 7844K->8386K(9216K), 0.0048484 secs] 0.147: [CMS: 8194K->6651K(10240K), 0.004  
Heap  
par new generation total 9216K, used 2214K [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)  
eden space 8192K, 27% used [0x00000007bec00000, 0x00000007bee29808, 0x00000007bf400000)  
from space 1024K, 0% used [0x00000007bf500000, 0x00000007bf500000, 0x00000007bf600000)  
to space 1024K, 0% used [0x00000007bf400000, 0x00000007bf400000, 0x00000007bf500000)  
concurrent mark-sweep generation total 10240K, used 6651K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)  
Metaspace used 3052K, capacity 4496K, committed 4864K, reserved 1056768K  
class space used 334K, capacity 388K, committed 512K, reserved 1048576K
```

执行到 array6时无法分配，新生代没有足够空间。此时触发 young gc

```
[ParNew (promotion failed): 7844K->8386K(9216K), 0.0048484 secs]
```

eden区原本有7000多kb对象，但是回收后发现一个都回收不掉，因为都被变量引用了。

这时要把这些对象放入老年年代了，但是老年年代里还有一个4MB的垃圾对象，老年年代空间也不够。

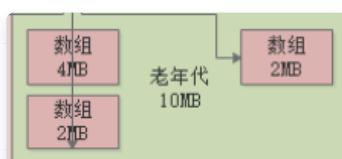
此时触发 Full Gc。

```
[CMS: 8194K->6651K(10240K), 0.0043053 secs] 11941K->6651K(19456K), [Metaspace: 3031K->3031K(1056768K)], 0.0094032 secs]
```

老年年代的Old Gc。

CMS: 8194K->6651K(10240K) 老年代的 8MB对象占用变成了6MB。为什么是8MB?

是因为在young gc的时候先把 2个 2MB数组对象放入老年年代。



再接着放剩下的2MB数组对象和一个128KB的数组对象时，放不下了，这时才触发的 Full Gc。

回收掉那 4MB的垃圾对象。接着再放入。



最后年轻代已经空了，所以把 array6变量引用的 2MB数组对象放入 eden。

```
eden space 8192K, 27% used
```

流程总结:

1. 4M的大对象直接进入老年年代。
2. 年轻代eden区域分配完3个2M对象和一个128Kb对象后，再分配第四个2M对象时，此时eden空间不足，分配失败。触发 young gc。此时 新生代存活对象 $6M + 128KB >$ 老年代可用连续空间 但是 老年代连续可用空间 $>$ 历次进入老年年代平均新生代平均大小，所以分配担保成功，不需要触发 Full gc。执行 young Gc。
3. 执行 young gc，但是 3个2MB和1个128KB都有引用，都存活，但是Survivor_from只有1MB空间，所以存活对象进入老年年代。
4. 先将2个存活的2MB对象和一个128KB对象放入老年年代，当分配最后一个2MB对象时，老年年代可用空间不够，触发Full Gc。回收原先的 4M垃圾对象。回收完后，将新生代活着的2MB对象放入老年年代。
5. 新生代放入最开始想要分配的2MB对象。

情况二：

老年年代可用空间 $<$ 历次young gc 进入老年年代对象的平均大小

情况三：老年年代使用率达到了 92%的阈值。

6.JVM监控工具

6.1 jstat

随时执行 jstat看到的就是系统启动以来的数据。

```
chenqipeng@chenqipengdeMacBook-Pro ~ % pgrep -l java
1703 java
1780 java
1781 java
chenqipeng@chenqipengdeMacBook-Pro ~ % jstat -gc 1781
    S0C   S1C   S0U   S1U     EC      EU      OC      OU      MC
MU   CCSC   CCSU   YGC     YGCT    FGCT    FGCT      GCT
14336.0 12288.0  0.0   12279.1 145920.0 32943.1  163840.0 15601.9  44672.
0 41763.7 6272.0 5623.4      7  0.052   2      0.074   0.126
chenqipeng@chenqipengdeMacBook-Pro ~ %
```

```
pgrep -l java
# 查看java进程
jps
jstat -gc Pid

# 每3分钟执行一次统计，连续执行10次
jstat -gc Pid 180000 10

#每隔一秒钟进行一次统计，连续执行1000次
jstat -gc Pid 1000 1000
```

S0C: 这是From Survivor区的大小
S1C: 这是To Survivor区的大小
S0U: 这是From Survivor区当前使用的内存大小
S1U: 这是To Survivor区当前使用的内存大小
EC: 这是Eden区的大小
EU: 这是Eden区当前使用的内存大小
OC: 这是老年代的大小
OU: 这是老年代当前使用的内存大小
MC: 这是方法区（永久代、元数据区）的大小
MU: 这是方法区（永久代、元数据区）的当前使用的内存大小
YGC: 这是系统运行迄今为止的Young GC次数
YGCT: 这是Young GC的耗时
FGC: 这是系统运行迄今为止的Full GC次数
FGCT: 这是Full GC的耗时
GCT: 这是所有GC的总耗时

6.2 jmap 和 jhat

了解Jvm运行的运行状况，然后进行优化。

按照各种对象内存空间的大小降序排列，占用内存最多的对象放最上面。

The screenshot shows a terminal window titled '-zsh' with the command 'jmap -histo' followed by a process ID. The output lists memory usage for various Java classes, including sun.util.calendar.Gregorian, sun.util.locale.provider.AuxLocaleProviderAdapter\$NullProvider, sun.util.locale.provider.CalendarDataUtility\$CalendarWeekParameterGetter, sun.util.locale.provider.SPILocaleProviderAdapter, sun.util.locale.provider.TimeZoneNameUtility\$TimeZoneNameGetter, sun.util.resources.LocaleData, and sun.util.resources.LocaleData\$LocaleDataResourceBundleControl. It also shows a total summary with 'Total 1229086 140257776'. The terminal prompt is 'chenqipeng@chenqipengdeMacBook-Pro ~ %'.

```
3969:          1      16 sun.security.util.ByteArrayLexOrder
3970:          1      16 sun.security.util.ByteArrayTagOrder
3971:          1      16 sun.text.normalizer.NormalizerBase$Mode
3972:          1      16 sun.text.normalizer.NormalizerBase$NFCMod
e
3973:          1      16 sun.text.normalizer.NormalizerBase$NFDMod
e
3974:          1      16 sun.text.normalizer.NormalizerBase$NFKCMo
de
3975:          1      16 sun.text.normalizer.NormalizerBase$NFKDMo
de
3976:          1      16 sun.util.calendar.Gregorian
3977:          1      16 sun.util.locale.provider.AuxLocaleProvide
rAdapter$NullProvider
3978:          1      16 sun.util.locale.provider.CalendarDataUtil
ity$CalendarWeekParameterGetter
3979:          1      16 sun.util.locale.provider.SPILocaleProvide
rAdapter
3980:          1      16 sun.util.locale.provider.TimeZoneNameUtil
ity$TimeZoneNameGetter
3981:          1      16 sun.util.resources.LocaleData
3982:          1      16 sun.util.resources.LocaleData$LocaleDataR
esourceBundleControl
Total        1229086    140257776
chenqipeng@chenqipengdeMacBook-Pro ~ %
```

```
jmap -histo Pid
```

简单了解当前jvm中的对象对内存的占用，直接用上面的方法即可。

使用jmap生成堆内存存储快照

```
# 在当前目录下生成一个 dump.hprof文件，这是二进制的格式，无法直接打开，把jvm内存的所有对象快照存放在文件中，供后续研究
jmap -dump:live,format=b,file=dump.hprof Pid
```

使用jhat分析快照

jhat 内置了web服务器，支持通过浏览器以图形化的方式分析堆存储快照。

使用如下命令即可启动 jhat 服务器，还可以指定自己想要的http端口号，默认是7000端口号。

```
jhat dump.hprof -port 7000
```

接着，在当前电脑上访问当前这台机器的7000端口号，就可以通过图形化的方式去分析堆内存里的对象分布情况。

6.3监控系统

大型公司会部署专门的监控系统，比较常见的有 Zabbix、OpenFalcon、Ganglia。

```
jps
```

```
jstat -gc
```

7. 测试 --- 上线 分析JVM运行状况及优化

7.1 预估性优化

完成开发后，测试，上线。

自行估算系统：

- 每秒大概多少请求
- 每个请求创建多少对象
- 占用多少内存
- 机器选什么配置
- 年轻代给多少内存
- young gc 触发的频率
- 对象进入老年代的速率
- 老年代给多少内存
- Full gc 触发的频率

这些都可以根据自己写的代码，大致合理预估。

预估之后，设置初始性的JVM参数。

优化思路：尽量让每次 young Gc后的存活对象<50%，都留存在年轻代里，尽量别让对象进入老年代，尽量减少Full gc 的频率。

7.2 系统压力测试

开源工具： 压力测试。

jstat分析。

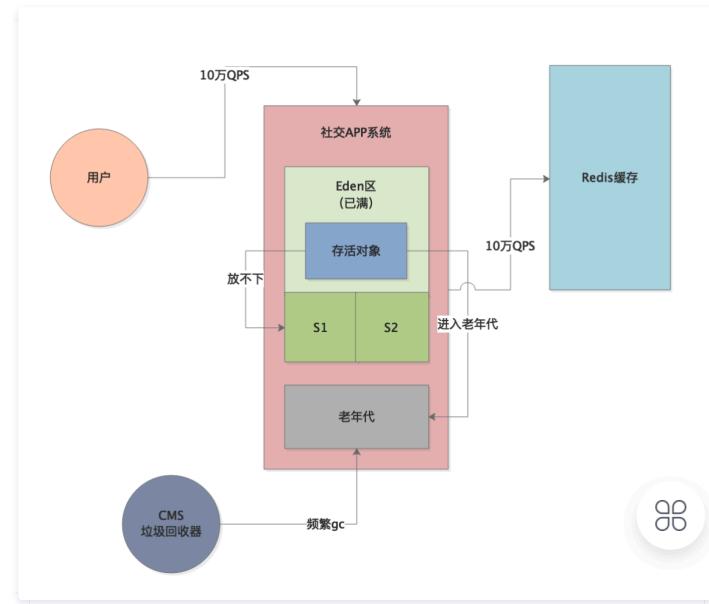
7.3 线上分析

大型公司会部署专门的监控系统，比较常见的有 Zabbix、OpenFalcon、Ganglia。

jps

jstat -gc

8. 每秒10万 qps的社交app优化gc性能



社交app流量最大的个人主页模块高峰期最多每秒会有10万+的QPS

高并发查询导致对象快速进入老年代，每次young gc的时候，实际上还有很多请求没处理完。

导致Eden区每次触发 young gc时，都有很多对象需要存活下来。在高峰期的时候，经常会出现Young Gc 过后存活对象较多，在Survivor区中放不下的问题。

老年代必然会频繁触发 Gc。

优化参数

1. 最核心 加机器
2. 给年轻代 Survivor 区域更大的内存空间
3. 老年代优化

-xx: +UseCMSFullGCsBeforeCompaction

-xx: CMSFullGCsBeforeCompaction=5

5次Full gc 后会触发一次 Compaction操作

整理内存碎片 压缩

把存活对象放到紧邻在一起，避免大量的内存碎片。

1. 参数优化：增加年轻代和S区的大小

2. 参数优化：

- -xx: +UseCMSFullGCsBeforeCompaction
- -xx: CMSFullGCsBeforeCompaction=0

每次full gc后都整理一下内存碎片

如果不整理内存碎片会导致 Full gc 越来越频繁

两个参数可以帮助优化Full GC的性能 把每次Full GC的时间进一步降低一些。

3.参数优化

- -XX:+CMSParallelInitialMarkEnabled

这个参数会在CMS垃圾回收器的“初始标记”阶段开启多线程并发执行。初始标记阶段，是会进行Stop the World的，会导致系统停顿，所以这个阶段开启多线程并发之后，可以尽可能优化这个阶段的性能，减少Stop the World的时间。

- -XX:+CMSScavengeBeforeRemark

这个参数会在CMS的重新标记阶段之前，先尽量执行一次Young GC。CMS的重新标记也是会Stop the World的。如果在重新标记之前，先执行一次Young GC，就会回收掉一些年轻代里没有人引用的对象。那么在CMS的重新标记阶段就可以少扫描一些对象，此时就可以提升CMS的重新标记阶段的性能，减少他的耗时。

老年代扫描的时候要确认老年代里哪些对象是存活的，这个时候必然会扫描到年轻代，因为有些年轻代的对象可能引用了老年代的对象，所以提前做young gc可以把年轻代里一些对象回收掉，减少了扫描的时间，可以提升性能

9.案例实战

通用优化步骤：

- 1.分析机器情况（机器配置，堆内存大小，运行时长，FullGC次数、时间，YoungGC次数、时间）
- 2.查看具体的jvm参数配置
- 3.然后根据JVM参数配置梳理出JVM模型，每个区间的大小是多少，画出来JVM模型（考虑每个设置在申请情况下会执行GC）
- 4.结合jstat查看的GC情况，在结合JVM模型进行二次分析
- 5.jmap dump内存快照，通过jhat或者Visual VM之类的工具查看具体的对象分类情况
- 6.根据分析的情况再具体到问题（Bug、或者参数设置等问题）
- 7.修复代码Bug，优化JVM参数

9.1 反射导致 Full gc (设置错参数)

背景：

系统在运行过程中，不停的有新的类产生被加载到 Metaspace （方法区）。不停的把MetaSpace占满，接着触发一次Full gc回收掉 MetaSpace区域中的部分类。这个过程不断的循环，造成反复的Full gc。

问题1：什么类不停被加载到MetaSpace？

JVM启动参数中加入：-XX:TraceClassLoading -XX:TraceClassUnloading

会通过日志打印出jvm加载了哪些类，卸载了哪些类。我们可以在 Tomcat 的 catalina.out 日志文件中看到。

```
【Loaded sun.reflect.GeneratedSerializationConstructorAccessor from  
__JVM_Defined_Class】
```

明显看到，JVM在运行期间不停的加载了大量的 "GeneratedSerializationConstructorAccessor" 类到了 MetaSpace。

问题2: 为什么会频繁加载奇怪的类?

Google 或者 百度一下。

那个类是在使用 java反射时加载的。

在执行这种反射代码时, jvm会在反射调用一定次数后就动态生成一些类, 就是类似上面的奇怪的类。下次再执行反射时, 直接调用这些类的方法, 这是jvm一个底层优化的机制。

问题3:为什么会有不停的创建这个奇怪的类放入MetaSpace?

因为jvm自己创建的这些类, class对象都是 SoftReference (软引用)

软引用在正常情况下不会回收, 但是如果内存紧张的时候就会回收这些对象。

软引用在gc时, 要不要回收通过下面的公式判断

```
clock - timestamp <= freespace * SoftRefLRUPolicyMSPerMB
```

clock - timestamp 表示一个软引用对象多久没有被访问了

freespace JVM中空闲内存空间

SoftRefLRUPolicyMSPerMB 每一MB空闲内存空间可以允许 软引用对象存活多久

举个例子:

JVM空闲内存空间 3000MB SoftRefLRUPolicyMSPerMB 默认值: 1000ms

$3000 * 1000 = 3000s$ 大概50分钟

一般来说发生gc时, jvm内部或多或少总有一些空间内存, 所以基本上如果不是快要发生OOM内存溢出, 一般软引用也不会被回收。

jvm随着反射代码的执行, 动态创建一些奇怪的类, 但是class对象都是软引用, 正常情况下不会被回收, 也不应该快速增长。

原因:

有人直接把 SoftRefLRUPolicyMSPerMB jvm启动参数设置为0, 他想的是任何软引用对象可以尽快释放, 不留存, 提高内存利用效率。

但是这个想法是错误的。

直接导致 软引用对象刚创建出来就被一次 young gc 带着立马回收了。

接着 jvm在反射代码执行下, 继续创建, 在jvm机制下, 导致这种奇怪的类越来越多。(涉及到jdk底层源码)

解决方法:

在有大量反射代码的场景下

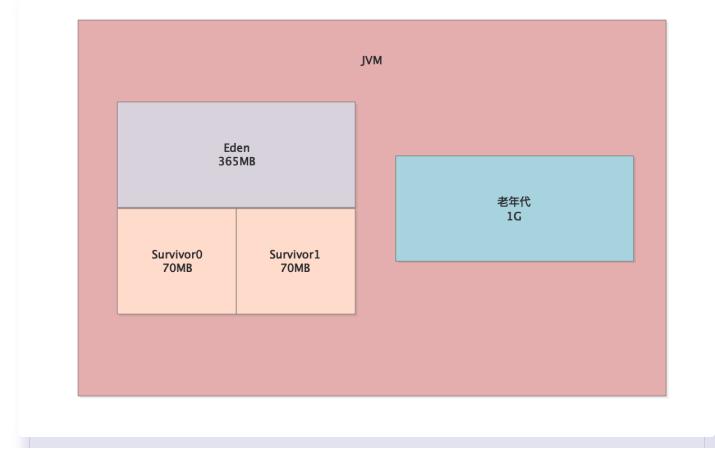
-XX: SoftRefLRUPolicyMSPerMB = xxxx 设置的大一些 1000 2000 5000

9.2 大对象直接进入老年代

- 机器配置: 2核4G
- JVM堆内存大小: 2G
- 系统运行时间: 6天
- 系统运行6天内发生的Full GC次数和耗时: 250次, 70多秒
- 系统运行6天内发生的Young GC次数和耗时: 2.6万次, 1400秒

下面是未优化前的线上JVM参数，大致如下：

```
-Xms1536M -Xmx1536M -Xmn512M -Xss256K -XX:SurvivorRatio=5 -XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=68 -XX:+CMSParallelRemarkEnabled  
-XX:+UseCMSInitiatingOccupancyOnly -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC
```



-XX:CMSInitiatingOccupancyFraction"参数设置为了68

所以一旦老年代内存占用达到68%，也就是大概有680MB左右的对象时，就会触发一次Full GC。

所以基本上我们就能根据推导出的运行内存模型得出一个结论：

每隔20秒会让300多MB的Eden区满触发一次Young GC，一次Young GC耗时50毫秒左右。

每隔30分钟会让老年代里600多MB空间占满，进而触发一次CMS的GC，一次Full GC耗时300毫秒左右。

猜测：

1. 是不是因为Survivor区域太小了，导致Young GC后的存活对象太多放不下，就一直有对象流入老年代，进而导致30分钟后触发Full GC？
2. 有可能是有很多长时间存活的对象太多了，都积累在老年代里，始终回收不掉，进而导致老年代很容易就达到68%的占比触发GC。

用jstat在高峰期观察一下JVM实际运行的情况。

通过jstat的观察，我们当时可以明确看到，每次Young GC过后升入老年代里的对象很少

一般来说，每次Young GC过后大概就存活几十MB而已，那么Survivor区域因为就70MB，所以经常会触发动态年龄判断规则，导致偶尔一次Young GC过后有几十MB对象进入老年代。

所以正常来说，应该不至于30分钟就导致老年代占用空间达到68%。

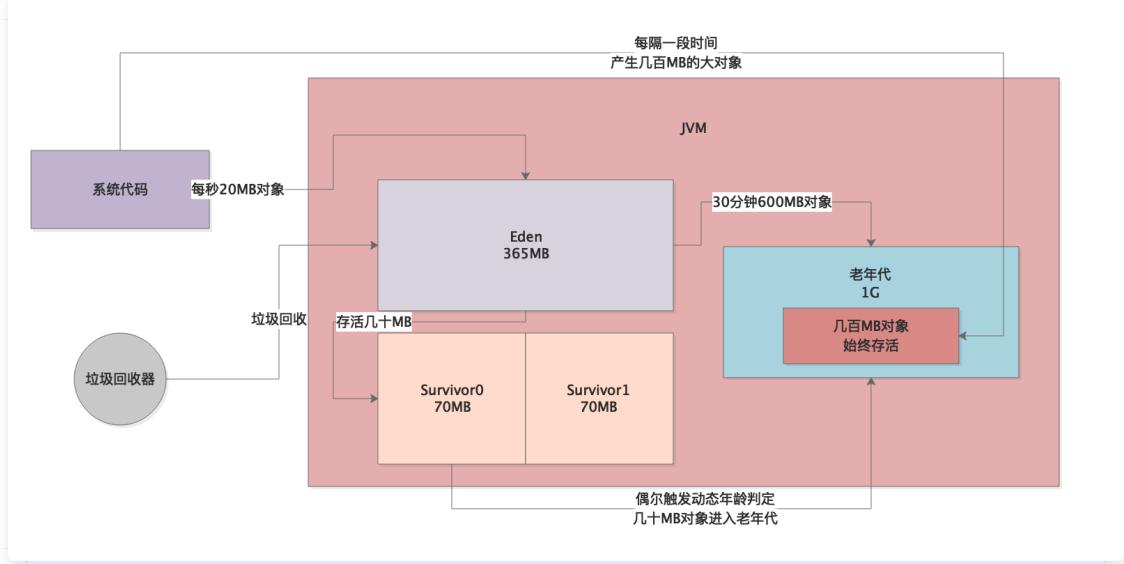
那么老年代里到底为什么有那么多对象呢？

这个时候我们通过jstat运行的时候就观察到一个现象，就是老年代里的内存占用在系统运行的时候，不知道为什么系统运行着运行着，就会突然有几百MB的对象占据在里面，大概有五六百MB的对象，一直占据在老年代中

原因：

一定是系统运行的时候，每隔一段时间就会突然产生几百MB的**大对象**，直接进入老年代，不会走年轻代的Eden区域。

然后再配合上年轻代还偶尔会有Young GC后几十MB对象进入老年代，所以才会30分钟触发一次Full GC！



分析到这里，就很简单了，我们只需要采用之前给大家介绍的jmap工具，通过后台jstat工具观察系统，什么时候发现老年代里突然进入了几百MB的大对象，就立马用jmap工具导出一份dump内存快照。

接着可以采用之前说过的Jhat，或者是Visual VM之类的可视化工具来分析dump内存快照

通过内存快照的分析，直接定位出来那个几百MB的大对象，就是几个Map之类的数据结构。

```
select * from tbl.
```

没有where条件，就代表这个SQL可能会把表中几十万条数据直接全部查出来！

优化思路：

1. 第一步，让开发解决代码中的bug，避免一些极端情况下SQL语句里不拼接where条件，务必要拼接上where条件，不允许查询表中全部数据。彻底解决那个时不时有几百MB对象进入老年代的问题。
2. 年轻代明显过小，Survivor区域空间不够，因为每次Young GC后存活对象在几十MB左右，如果Survivor就70MB很容易触发动态年龄判定，让对象进入老年代中。所以直接调整JVM参数如下：
直接把年轻代空间调整为700MB左右，每个Survivor是150MB左右，此时Young GC过后就几十MB存活对象，一般不会进入老年代。反之老年代就留500MB左右就足够了，因为一般不会有对象进入老年代。而且调整了参数“-XX:CMSInitiatingOccupancyFraction=92”，避免老年代仅仅占用到68%就触发GC，现在必须要占用到92%才会触发GC。
3. 最后，就是主动设置了永久代大小为256MB，因为如果不主动设置会导致默认永久代就在几十MB的样子，很容易导致万一系统运行时候采用了反射之类的机制，可能一旦动态加载的类过多，就会频繁触发Full GC。

9.3 cpu飙高-内存泄露（mat使用）

背景：CPU负载过高

常见场景：

1. 自己系统创建了大量线程，同时并发运行，且工作负载都很重
2. JVM频繁Full Gc，Full gc 非常耗费cpu资源

通过监控平台发现jvm的Full gc 突然变得频繁。

可能性有3个：

1. 内存分配不合理，导致对象频繁进入老年代
2. 内存泄露问题，内存里驻留了大量的对象塞满了老年代，导致稍微有一些对象进入就会触发 Full gc。
3. 永久代里的类太多，触发 Full gc。

也有可能 工程师错误的执行 System.gc()导致。高并发场景下会导致机器卡死。但是这个可以在jvm参数中禁止这种显示触发的gc。

用 Jstat 分析，并不存在内存不合理分配，永久代也正常。

原因：

老年代驻留了大量的对象，内存泄露。

解决方法：

可以用 jmap+jhat 分析内存里的大对象。

也可以用另一个强大的工具，MAT。

首先用jmap对线上系统导出一份 **内存快照**。

```
1190 Jps
1176 Launcher
1177 Demo1
1151
```

接着执行下面的jmap命令可以导出dump内存快照：

```
jmap -dump:live,format=b,file=dump.hprof 1177
```

```
jmap -dump:format=b,file=文件名[服务进程id]
```

其实就是一份文件，接着可以用 **jhat** 和 **MAT** 之类的工具进行分析。

MAT下载地址：<https://www.eclipse.org/mat/downloads.php>

tips：如果报 failer create java virtual machine

解决方法：

1. vi /Applications/mat.app/Contents/Eclipse/MemoryAnalyzer.ini
2. 增加

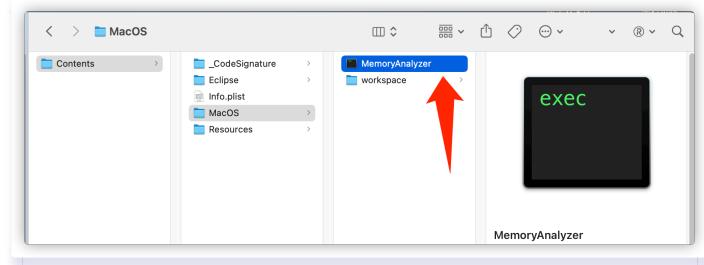
```
-VM
/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin
```

```

#!/bin/sh
# startup
./Eclipse/plugins/org.eclipse.equinox.launcher_1.5.700.v20200207-2156.jar
--launcher.library
./Eclipse/plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.1100.v20190907-0426
-vm
/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin
-Xmargs
-Xmx1024m
-Dorg.eclipse.swt.internal.carbon.smallFonts
-XstartOnFirstThread

```

3. 执行 MemoryAnalyzer 脚本



4. 再启动mat就可以了

下载后，安装目录里有一个文件名字: MemoryAnalyzer.ini

这个文件里的内容类似如下所示：

```

-startup
./Eclipse/plugins/org.eclipse.equinox.launcher_1.5.0.v20180512-
1130.jar
--launcher.library
./Eclipse/plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1
.1.700.v20180518-1200
-Xmargs
-Xmx1024m
-Dorg.eclipse.swt.internal.carbon.smallFonts
-XstartOnFirstThread

```

务必记得：如果 dump快照文件很大，比如几个g，务必在启动MAT之前，给MAT本身设置一下堆内存大小。默认这里是 -Xmx 1024m。一个G。

接着直接启动MAT，启动之后界面有一个选项: Open a Heap Dump。打开内存快照。

基于MAT进行内存分析：

工具栏上有一个按钮： Leak Suspects。 这就是内存泄露的分析

原因：

系统做了一个本地缓存，把很多数据都加载到内存里缓存起来了，然后提供查询服务直接从本地内存走。但是没有限制本地缓存的大小，并且没有使用LRU算法定期淘汰一些缓存里的数据，导致缓存在内存里的对象越来越多，造成 **内存泄露**（系统创建大量对象，很多都不需要使用，而且还无法回收）。

解决方法:

类似 EHCache之类的缓存框架就可以，他会固定最多缓存多少个对象，定期淘汰删除掉一些不怎么访问的缓存，以便于新数据进入缓存。

总结:

dump.hprof文件来分析， dump文件一般超大，用MAT之前设置下MAT自身的堆内存大小。

到这基本就能分析出来是哪段代码，导致频繁的full gc，而频繁的full gc又导致CPU飙高了。总结一下排查思路

1. CPU飙高的两个猜测： a. 线程并发过高 b. 并发的full gc过于频繁
2. 使用jstat分析gc情况大概猜测是内存泄露导致频繁full gc，导致CPU飙高
3. 使用top命令查看占CPU最高的进程PID
4. 通过ps -mp pid -o THREAD,tid,time找到这个进程中占用CPU最高的线程id： TID
5. 通过printf "%x\n" tid TID转成16进制找到jstack中的NID
6. 通过jstack pid |grep nid -A 30定位到具体的jvm中的线程从而确定是full gc线程导致的问题
7. 使用jmap -dump:live,format=b,file=dump.hprof PID 转储堆快照
8. 使用idea + MAT分析堆快照找到内存泄露的真凶！

线上String.split()方法导致频繁full gc

9.4 String.split() 造成内存泄露

jdk1.6 String 底层是基于数组存放字符的 切割出来的字符串不会对应新的数组，而是直接映射到原来那个字符串的数组，用偏移量表明自己是对应原始数组的那些元素。

jdk1.7 则给每一个切分出来的字符串都创建一个新的数组。

当线上系统加载大量数据时，数据主要是字符串时，对这些字符串进行切割，每个字符串都会切割为 N 个字符串，这会导致系统频繁产生大量对象。

优化逻辑: 开启多线程并发处理大量的数据，尽量提升数据处理完毕的速度。避免触发 young gc时，大量对象存活下来。

9.5 JVM参数模板

这是我针对 8C 16G 机器 结合线上真实服务的gc情况
一套jvm参数配置，对于一般的非报表类那种一次性
大数据出来的 业务处理类服务基本够用

```
-Xms7500m -Xmx7500m -Xmn3000m -Xss512k -  
XX:MaxMetaspaceSize=512m -XX:MetaspaceSize=5  
-Dfile.encoding=UTF-8  
-XX:+UseConcMarkSweepGC  
-XX:+UseParNewGC  
-XX:+UseCMSCompactAtFullCollection  
-XX:CMSFullGCsBeforeCompaction=0  
-XX:CMSInitiatingOccupancyFraction=65  
-XX:SurvivorRatio=8  
-XX:+UseCMSInitiatingOccupancyOnly  
-XX:+CMSScavengeBeforeRemark ## 在cms gc之  
行一次 minor gc 减少 remark的开销  
-XX:+ParallelRefProcEnabled #并行处理 reference 加  
remak  
-XX:+CMSClassUnloadingEnabled  
-XX:+DisableExplicitGC ## 防止某些“大神”在后台里  
行 System.gc()  
-XX:+CMSParallelInitialMarkEnabled ## 初始标记 多线  
行 一般没必要，初始标记都很快  
-XX:+CMSClassUnloadingEnabled  
-XX:ParallelGCThreads=$CPU_Count  
-Xloggc:${PROJECT_DIR}/logs/gc.log  
-XX:+PrintGCDateStamps  
-XX:+PrintGCDetails  
-XX:+HeapDumpOnOutOfMemoryError  
-  
XX:HeapDumpPath=${PROJECT_DIR}/logs/heapdump
```

收起

2020-8-9

5 ❤️

9.6 结合自己的项目优化jvm

2、现在的你应该如何在面试中回答JVM生产优化问题？

现在的你已经学习了这么多的内容，**应该如何在面试中回答JVM生产优化问题？**

一种比较常见的做法，就是把之前学习过的知识，归纳总结出来一套通用的方法论，然后面试的时候就聊这套通用方法论即可。

这个方法没有问题，很多面试官其实听到这套回答已经眼前一亮了，因为国内很少有人能把JVM生产优化的方法论总结的如此之系统的。

但是还不够，因为面试官想听的，实际上是你自己负责的系统是如何进行JVM优化的！

3、如果你的系统访问量和数据量暴增10倍或者100倍

所以你在这里应该思考的一个问题，就是你负责的系统，假设数据量和访问量暴增10倍，或者100倍，此时会不会出现频繁Full GC的问题？

利用学习过的知识去倒推一下，其实很可能的，在有限的机器资源下，一旦压力增长，很可能因为内存分配不合理，导致频繁Full GC的！

上面我们说过好几种频繁Full GC的触发条件，你是不是都可以放在自己的系统里去思考一下，自己的系统有没有可能会发生上述几种场景下的频繁Full GC？

如果会的话，那么一旦发生了，如何定位、分析和解决？

你应该把频繁Full GC问题和你自己的业务系统结合起来，自己深度思考，自己整理出来几个自己系统可能产生的JVM性能问题，然后整理出一套解决方案出来。

未来在面试的时候，应该结合自己的系统去跟面试官聊，说自己的系统可能在哪些情况下发生频繁Full GC，在压测的时候就发现了这些问题，然后你是如何进行JVM性能优化的！

这样面试官一定会认可你对JVM这块技术的掌握和实践经验的！

10. 内存溢出 (OOM)

详情: 72课

可能发生内存溢出的三个地方:

- MetaSpace
- java虚拟机栈
- 堆内存

10.1 MetaSpace 内存溢出

metaSpace里的类回收的条件相当的苛刻: 比如这个类的类加载器先要被回收, 比如这个类的所有对象实例都要被回收, 等等。

发生MetaSpace 内存溢出的条件:

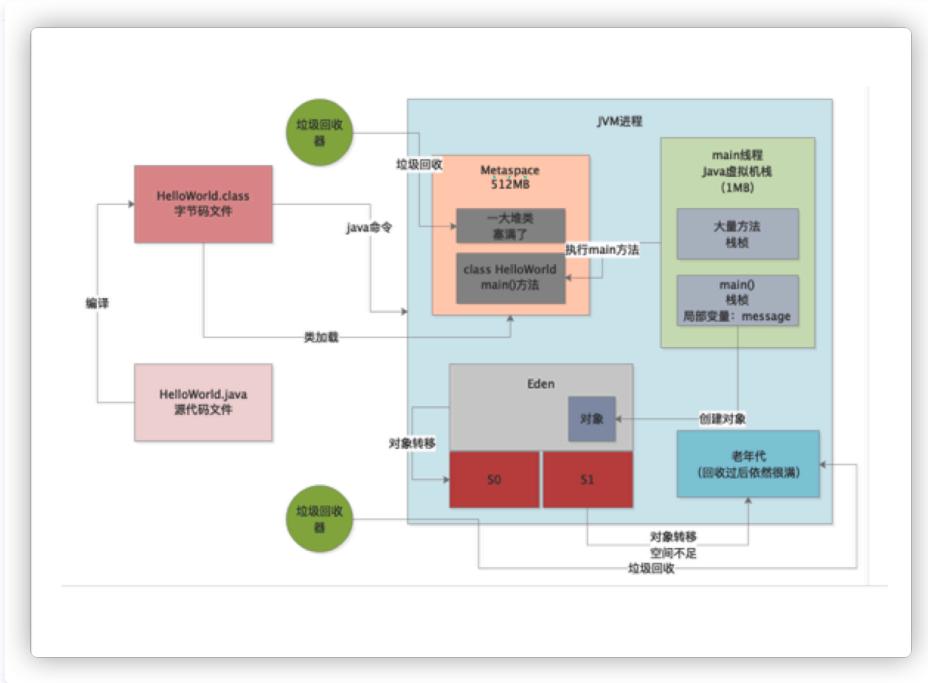
- 上线系统的时候对 Metaspace 使用默认的参数, 根本不设置大小
- 很多人写代码的时候会用cglib之类的技术动态生成一些类, 一旦代码中没有控制好, 导致你生成的类过多, 就容易把Metaspace塞满。

解决方法:

- -XX: MetaspaceSize = 512m
- -XX: MaxMetaspaceSize = 512m

设置一下大小, 默认只有几十MB

模拟MetaSpace内存溢出



设置jvm参数

-XX:MetaspaceSize=10m
-XX:MaxMetaspaceSize=10m

```
public class Demo1 {  
  
    public static void main(String[] args) {  
        int counter = 0;  
        while(true){  
            Enhancer enhancer = new Enhancer();  
            enhancer.setSuperclass(Car.class);  
            enhancer.setUseCache(false);  
            enhancer.setCallback(new MethodInterceptor() {  
                @Override  
                public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws  
Throwable {  
  
                    if(method.getName().equals("run")){  
                        System.out.println("启动汽车之前，先进行自动的安全检查.....");  
                        return methodProxy.invokeSuper(o,objects);  
                    }else{  
                        return methodProxy.invokeSuper(o,objects);  
                    }  
                }  
            });  
  
            Car car = (Car) enhancer.create();  
            car.run();  
            counter++;  
            System.out.println("目前创建了"+counter+"个car类的子类了");  
        }  
    }  
  
    static class Car{  
        public void run(){  
            System.out.println("汽车启动，开始行驶.....");  
        }  
    }  
}
```

```

后启动汽车之前，先进行自动的安全检查.....  

汽车启动，开始行驶.....  

目前创建了265个car类的子类  

启动汽车之前，先进行自动的安全检查.....  

汽车启动，开始行驶.....  

目前创建了266个car类的子类  

启动汽车之前，先进行自动的安全检查.....  

汽车启动，开始行驶.....  

目前创建了267个car类的子类  

启动汽车之前，先进行自动的安全检查.....  

Exception in thread "main" net.sf.cglib.core.CodeGenerationException Create breakpoint : java.lang.reflect.InvocationTargetException-->null  

    at net.sf.cglib.core.AbstractClassGenerator.generate(AbstractClassGenerator.java:348)  

    at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:96)  

    at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:94)  

    at net.sf.cglib.core.internal.LoadingCache$2.call(LoadingCache.java:54) <1 internal call>  

    at net.sf.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:61)  

    at net.sf.cglib.core.internal.LoadingCache.get(LoadingCache.java:34)  

    at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData.get(AbstractClassGenerator.java:119)  

    at net.sf.cglib.core.AbstractClassGenerator.create(AbstractClassGenerator.java:294)  

    at net.sf.cglib.reflect.FastClass$Generator.create(FastClass.java:65)  

    at net.sf.cglib.proxy.MethodProxy.helper(MethodProxy.java:121)  

    at net.sf.cglib.proxy.MethodProxy.init(MethodProxy.java:75)  

    at net.sf.cglib.proxy.MethodProxy.invokeSuper(MethodProxy.java:226)  

    at Demo01$1.intercept(Demo01.java:28)  

    at Demo01$Car$$EnhancerByCGLIB$$d2213dee_268.run(<generated>)  

    at Demo01.main(Demo01.java:36)  

Caused by: java.lang.reflect.InvocationTargetException <3 internal calls>  

    at net.sf.cglib.core.ReflectUtils.defineClass(ReflectUtils.java:459)  

    at net.sf.cglib.core.AbstractClassGenerator.generate(AbstractClassGenerator.java:339)  

... 15 more  

Caused by: java.lang.OutOfMemoryError Create breakpoint : Metaspace
    at java.lang.ClassLoader.defineClass(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
... 20 more

```



用mat分析内存情况:

jvm参数

```

-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:MetaspaceSize=10m
-XX:MaxMetaspaceSize=10m
-XX:+PrintGCDetails
-Xloggc:gc.log
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=./

```

gc.log

```

Java HotSpot(TM) 64-Bit Server VM (25.191-b12) for bsd-amd64 JRE (1.8.0_191-b12), built on Oct 6 2018 08:37:07 by
"java_re" with gcc 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)
Memory: 4k page, physical 16777216k(909532k free)

```

/proc/meminfo:

```

CommandLine flags: -XX:CompressedClassSpaceSize=2097152 -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=./ -
XX:InitialHeapSize=268435456 -XX:MaxHeapSize=4294967296 -XX:MaxMetaspaceSize=10485760 -XX:MaxNewSize=697933824 -
XX:MaxTenuringThreshold=6 -XX:MetaspaceSize=10485760 -XX:OldPLABSize=16 -XX:+PrintGC -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseConcMarkSweepGC -
XX:+UseParNewGC
1.233: [GC (Allocation Failure) 1.234: [ParNew: 69952K->2029K(78656K), 0.0075673 secs] 69952K->2029K(253440K),
0.0077994 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
1.752: [Full GC (Metadata GC Threshold) 1.752: [CMS: 0K->2483K(174784K), 0.0597141 secs] 67239K->2483K(253440K),
[Metaspace: 9632K->9632K(1058816K)], 0.0598852 secs] [Times: user=0.11 sys=0.02, real=0.06 secs]
1.812: [Full GC (Last ditch collection) 1.812: [CMS: 2483K->1491K(174784K), 0.0158886 secs] 2483K->1491K(253504K),
[Metaspace: 9632K->9632K(1058816K)], 0.0160045 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

```

```

1.866: [Full GC (Metadata GC Threshold) 1.866: [CMS: 1491K->1489K(174784K), 0.0190498 secs] 2836K->1489K(253504K),
[Metaspace: 9653K->9653K(1058816K)], 0.0193842 secs] [Times: user=0.04 sys=0.00, real=0.02 secs]
1.885: [Full GC (Last ditch collection) 1.885: [CMS: 1489K->1489K(174784K), 0.0177313 secs] 1489K->1489K(253504K),
[Metaspace: 9653K->9653K(1058816K)], 0.0179918 secs] [Times: user=0.04 sys=0.00, real=0.02 secs]
1.903: [GC (CMS Initial Mark) [1 CMS-initial-mark: 1489K(174784K)] 1489K(253504K), 0.0021433 secs] [Times:
user=0.00 sys=0.01, real=0.00 secs]
1.906: [CMS-concurrent-mark-start]
1.909: [CMS-concurrent-mark: 0.003/0.003 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
Heap
1.909: [CMS-concurrent-preclean-start]
par new generation total 78720K, used 3446K [0x00000006ffe00000, 0x0000000705360000, 0x0000000729790000)
eden space 70016K, 4% used [0x00000006ffe00000, 0x000000070015d858, 0x0000000704260000)
from space 8704K, 0% used [0x0000000704260000, 0x0000000704260000, 0x0000000704ae0000)
to space 8704K, 0% used [0x0000000704ae0000, 0x0000000704ae0000, 0x0000000705360000)
concurrent mark-sweep generation total 174784K, used 1489K [0x0000000729790000, 0x0000000734240000,
0x00000007ffe00000)
Metaspace used 9660K, capacity 10186K, committed 10240K, reserved 1058816K
 class space used 855K, capacity 881K, committed 896K, reserved 1048576K
1.910: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.910: [CMS-concurrent-abortable-preclean-start]

```

分析log:

```

1.233: [GC (Allocation Failure) 1.234: [ParNew: 69952K->2029K(78656K), 0.0075673 secs] 69952K->2029K(253440K),
0.0077994 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

```

第一次 young gc: 第一次 gc Allocation Failure 说明: 在eden区分配对象时, 发现eden区内存不足, 于是触发young gc。

```
Enhancer enhancer = new Enhancer();
```

因为 Enhancer本身是一个对象。上述代码不止会动态生成类, 还会生成很多对象。

```

1.752: [Full GC (Metadata GC Threshold) 1.752: [CMS: 0K->2483K(174784K), 0.0597141 secs] 67239K->2483K(253440K),
[Metaspace: 9632K->9632K(1058816K)], 0.0598852 secs] [Times: user=0.11 sys=0.02, real=0.06 secs]

```

第二次 Full gc: Metadata GC Threshold 说明 Metaspace区域满了, 所以触发了Full GC. 老年代增加了 2483K , 总的堆内存从 67239k 减少到 2483k metaSpace 9632k保持不变。说明这次Full gc 把 eden区的大部分垃圾回收了, 剩下的 2483k很有可能是 jvm运行需要的对象, 放不下s区, 直接放入老年代, 然而Metaspace区域的类内部是被 cglib引用的, 所以全部无法回收。 tip: 参数里我们设置了metaspace 为10m, 这里的 1058816k 不用理会。

```

1.812: [Full GC (Last ditch collection) 1.812: [CMS: 2483K->1491K(174784K), 0.0158886 secs] 2483K->1491K(253504K),
[Metaspace: 9632K->9632K(1058816K)], 0.0160045 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

```

第三次 Full gc: Last ditch collection 说明这是最后拯救的机会, metaSpace 还是无法回收。将近占满了 10M内存。接着jvm直接退出了, 年轻代和老年代几乎没什么占用。

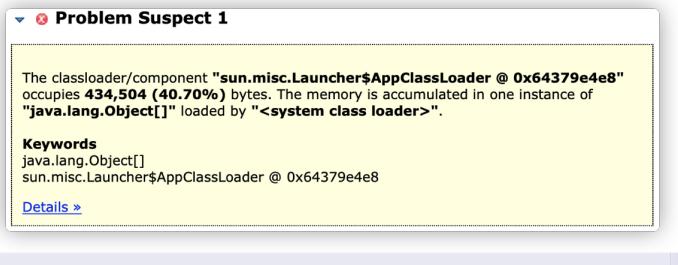
```

Caused by: java.lang.OutOfMemoryError Create breakpoint : Metaspace
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
... 11 more

```

控制台明确抛出oom 异常

用 mat 进行内存快照分析



说明实例最多的是 AppClassLoader。因为CGLIB 之类的东西在动态生成类的时候搞出来的。点击 [Details](#) 查看。

有一大堆 Demo1 动态生成出来的 Car EnhancerBYCGLIB 的类。

真相大白，是Demo1里面搞出来的一大堆的动态生成类，填满了Metaspace。

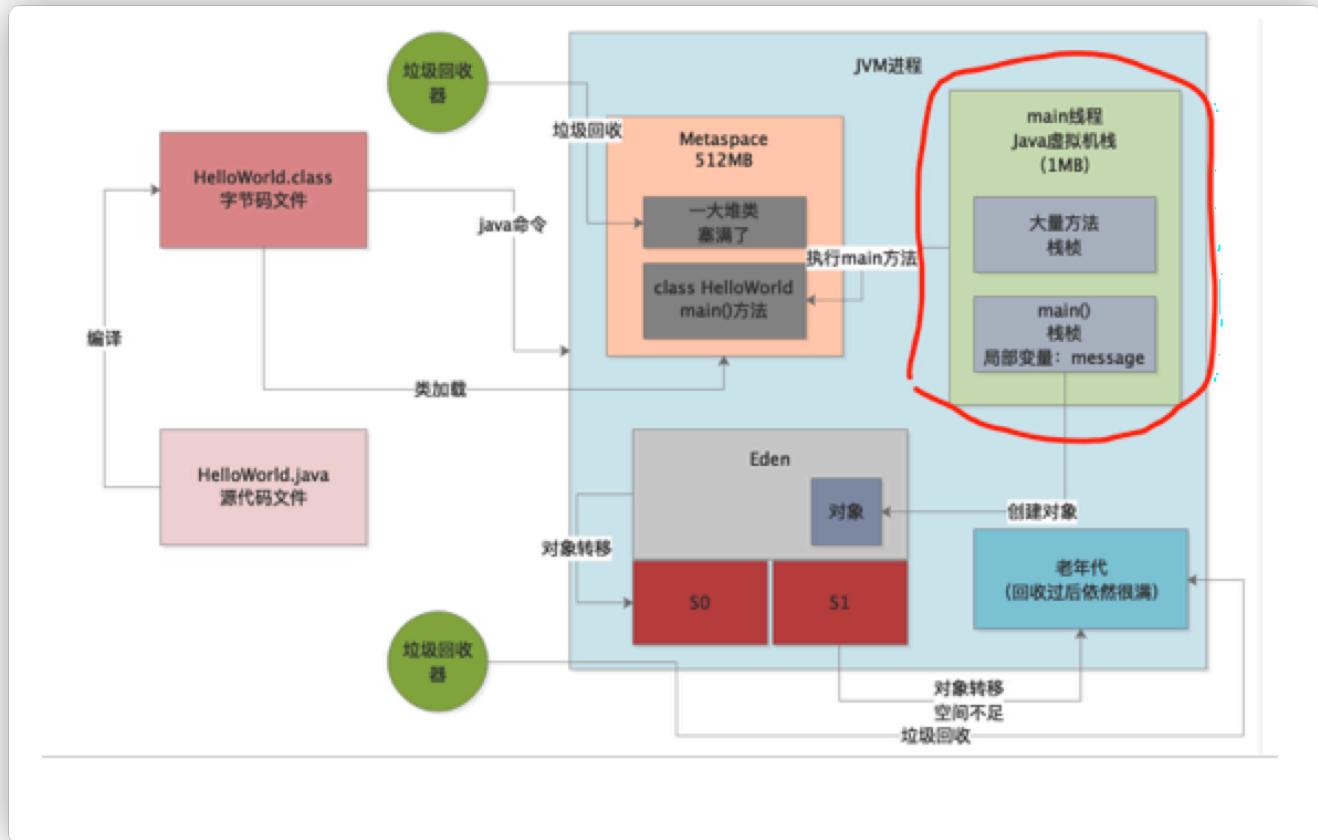
解决方法：对Enhancer 缓存。

10.2 虚拟机栈内存溢出

```
public class HelloWorld {
    public static void main(String[] args) {
        String message = "HelloWorld";
        System.out.println(message);
        sayHello("救火队队长");
    }
    public static void sayHello(String name) {
        System.out.println("你好，救火队队长");
    }
}
```

jvm启动后，HelloWorld 类被加载到内存里，通过 main线程执行main()方法，在main线程的虚拟机栈内，压入main () 方法对应的栈桢，里面放入main () 方法的局部变量。

main中调用 sayHello()方法，sayHello也一样，它的栈桢也压入到main线程的虚拟机栈中。



`sayHello` 方法如果运行完，就不需要这个方法在内存中保存他的局部变量了，此时他的栈桢将会出栈。

每次方法调用的栈桢都是要占内存。

一个线程的虚拟机栈内存大小是有限的，比如1MB。如果不调用方法，最终就会发生栈内存溢出。

```
public static void sayHello(String name) {
    sayHello(name);
}
```

递归不停调用同一个方法，很容易栈内存溢出。

一般来说栈内存溢出，都是代码的bug，正常情况下发生的比较少。

模拟虚拟机栈内存溢出

```
public class Demo2 {  
    public static long counter = 0;  
    public static void main(String[] args) {  
        work();  
    }  
    public static void work() {  
        System.out.println("目前是第" + (++counter) + "次调用方法");  
        work();  
    }  
}
```

上面的代码非常简单，就是work()方法调用自己，进入一个无限制的递归调用，陷入死循环，也就是说在main线程的栈中，会不停的压入work()方法调用的栈桢，直到1MB的内存空间耗尽。

另外大家需要设置这个程序的JVM参数如下： -XX:ThreadStackSize=1m，通过这个参数设置JVM的栈内存为1MB。

接着大家运行这段代码，会看到如下所示的打印输出：

```
前提是第5675次调用方法  
java.lang.StackOverflowError
```

也就是说，当这个线程调用了5675次方法之后，他的栈里压入了5675个栈桢，最终把1MB的栈内存给塞满了，引发了栈内存的溢出。大家看到StackOverflowError，就知道是线程栈内存溢出了。

栈内存溢出分析 gc日志和内存快照是没用的。

看本地日志文件即可。

10.3 堆内存溢出

发生堆内存溢出的原因其实总结下来，就一句话：

有限的内存中放了过多的对象，而且大多数都是存活的，此时即使GC过后还是大部分都存活，所以要继续放入更多对象已经不可能了，此时只能引发内存溢出问题。

所以一般来说发生内存溢出有两种主要的场景：

- ① 系统承载高并发请求，因为请求量过大，导致大量对象都是存活的，所以要继续放入新的对象实在是不行了，此时就会引发OOM系统崩溃
- ② 系统有内存泄漏的问题，就是莫名其妙弄了很多的对象，结果对象都是存活的，没有及时取消对他们的引用，导致触发GC还是无法回收，此时只能引发内存溢出，因为内存实在放不下更多对象了

因此总结起来，一般引发OOM，要不然是系统负载过高，要不然就是有内存泄漏的问题

[模拟堆内存溢出](#)

```
public class Demo3 {  
    public static void main(String[] args) {  
        long counter = 0;  
        List<Object> list = new ArrayList<Object>();  
        while(true) {  
            list.add(new Object());  
            System.out.println("当前创建了第" + (++counter) + "个对象");  
        }  
    }  
}
```

代码很简单，就在一个while循环里不停的创建对象，而且对象全部都是放在List里面被引用的，也就是不能被回收。

大家试想一下，如果你不停的创建对象，Eden区满了，他们全部存活会全部转移到老年带，反复几次之后老年带满了。

然后Eden区再次满了，ygc后存活对象再次进入老年带，此时老年带先full gc，但是回收不了任何对象，因此ygc后的存活对象就一定是无法进入老年带的。

所以我们用下面的JVM参数来运行一下代码：-Xms10m -Xmx10m，我们限制了堆内存大小总共就只有10m，这样可以尽快触发堆内存的溢出。

我们在控制台打印的信息中可以看到如下的信息：

```
当前创建了第360145个对象  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap  
space
```

所以从这里就可以看到，在10M的堆内存中，用最简单的Object对象搞到老年带被塞满大概需要36万个对象。然后堆内存实在放不下任何其他对象，此时就会OutOfMemory了，而且告诉了你是Java heap space，也就是堆空间发生了内存溢出的。

11.JVM内存溢出的时候自动dump内存快照

```
public class Demo3 {
    public static void main(String[] args) {
        long counter = 0;
        List<Object> list = new ArrayList<Object>();
        while(true) {
            list.add(new Object());
            System.out.println("当前创建了第" + (++counter) + "个对象");
        }
    }
}
```

代码很简单，就在一个while循环里不停的创建对象，而且对象全部都是放在List里面被引用的，也就是不能被回收。

大家试想一下，如果你不停的创建对象，Eden区满了，他们全部存活会全部转移到老年带，反复几次之后老年带满了。

然后Eden区再次满了，ygc后存活对象再次进入老年带，此时老年带先full gc，但是回收不了任何对象，因此ygc后的存活对象就一定是无法进入老年带的。

所以我们用下面的JVM参数来运行一下代码：-Xms10m -Xmx10m，我们限制了堆内存大小总共就只有10m，这样可以尽快触发堆内存的溢出。

我们在控制台打印的信息中可以看到如下的信息：

```
当前创建了第360145个对象
Exception in thread "main" java.lang.OutOfMemoryError: Java heap
space
```

所以从这里就可以看到，在10M的堆内存中，用最简单的Object对象搞到老年带被塞满大概需要36万个对象。然后堆内存实在放不下任何其他对象，此时就会OutOfMemory了，而且告诉了你是Java heap space，也就是堆空间发生了内存溢出的。

```
“-Xms4096M -Xmx4096M -Xmn3072M -Xss1M -  
XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -  
XX:+UseParNewGC -XX:+UseConcMarkSweepGC -  
XX:CMSInitiatingOccupancyFraction=92 -  
XX:+UseCMSCompactAtFullCollection -  
XX:CMSFullGCsBeforeCompaction=0 -  
XX:+CMSParallelInitialMarkEnabled -XX:+CMSScavengeBeforeRemark  
-XX:+DisableExplicitGC -XX:+PrintGCDetails -Xloggc:gc.log -  
XX:+HeapDumpOnOutOfMemoryError -  
XX:HeapDumpPath=/usr/local/app/oom”
```

这份JVM参数模板基本上涵盖了所有你需要的一些参数

首先是各个内存区域的大小分配，这个是需要你精心调优的

其次是两种垃圾回收器的指定，接着是一些常规性的CMS垃圾回收的参数，可以帮助优化偶尔发生的Full GC性能。

最重要的，就是平时要打印出来GC日志，GC日志可以配合你用jstat工具分析GC频率和性能的时候用，jstat可以分析出来GC的频率，但是对每次具体的GC情况，可以结合GC日志来看。

还有就是在OOM的时候需要自动dump内存快照，这样即使突然发生OOM，你只要得知了这个事，立马就可以去分析内存快照了。

分析堆内存溢出

```
java.lang.OutOfMemoryError: Java heap space  
Dumping heap to ./java_pid1023.hprof ...  
Heap dump file created [13409210 bytes in 0.033 secs]  
  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

这个就很明显告诉我们，是Java堆内存溢出了，而且他还给我们导出了一份内存快照。

所以此时我们GC日志都不用分析了，因为堆内存溢出往往对应着大量的GC日志，所以你分析起来是很麻烦的。

此时直接将线上自动导出的内存快照拷贝回本地笔记本电脑，然后用MAT分析即可。

12.案例实战

12.1 微服务远程调用失败导致oom

背景：系统突然崩溃， oom问题。

1. 登录系统线上机器查看对应日志。

看两点：

- 看看是堆溢出、还是栈内存溢出或是Metaspace溢出
- 看看是哪个线程(tomcat线程 我们自己写的代码的线程)

当时在机器的日志文件中看到类似下面的一句话：

```
Exception in thread "http-nio-8080-exec-1089" java.lang.OutOfMemoryError: Java heap space
```

这其实是tomcat的工作线程：堆内存溢出

线上系统记得设置：

```
-XX:+HeapDumpOnOutOfMemoryError
```

这个参数会在系统内存溢出的时候导出来一份内存快照到我们指定的位置。

2. Mat 对内存快照进行分析

发现占据内存最大的是大量的“byte[]”数组一大堆的byte[]数组就占据了大约8G左右的内存空间。而我们当时线上机器给Tomcat的JVM堆内存分配的也就是8G左右的内存而已。

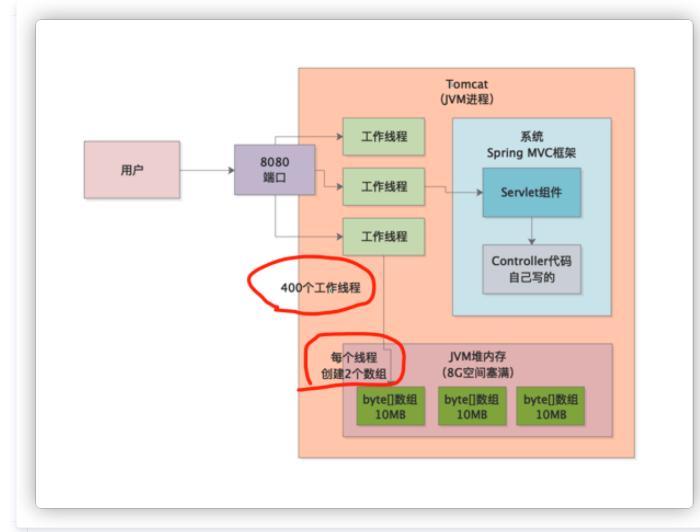
结论：Tomcat工作线程在处理请求的时候会创建大量的byte[]数组，大概有8G左右，直接把JVM堆内存占满了。

在MAT上可以继续查看一下这个数组是被谁引用的，大致可以发现是Tomcat的类引用的，具体来说是类似下面的这个类：
org.apache.tomcat.util.threads.TaskThread

这个类一看就是Tomcat自己的线程类，因此可以认为是Tomcat的线程创建了大量的byte[]数组，占据了8G的内存空间。

我们发现Tomcat的工作线程大致有400个左右，也就是说每个Tomcat的工作线程会创建2个byte[]数组，每个byte[]数组是10MB左右

最终就是400个Tomcat工作线程同时在处理请求，结果创建出来了8G内存的byte[]数组，进而导致了内存溢出。



3. 分析系统

一秒钟之内瞬间来了400个请求，导致Tomcat的400个工作线程全部上阵处理请求，每个工作线程在处理一个请求的时候，会创建2个数组，每个数组是10MB，结果导致瞬间就让8G的内存空间被占满了。

检查了一下系统的监控，发现每秒请求并不是400，而是100！

也就是说明每秒才100个请求，[怎么可能Tomcat的400个线程都处于工作状态？](#)

如果每秒来100个请求，但是每个请求处理完毕需要4秒的时间，那么在4秒内瞬间会导致有400个请求同时在处理，也就会导致Tomcat的400个工作线程都在工作！接着就会导致上述的情况。

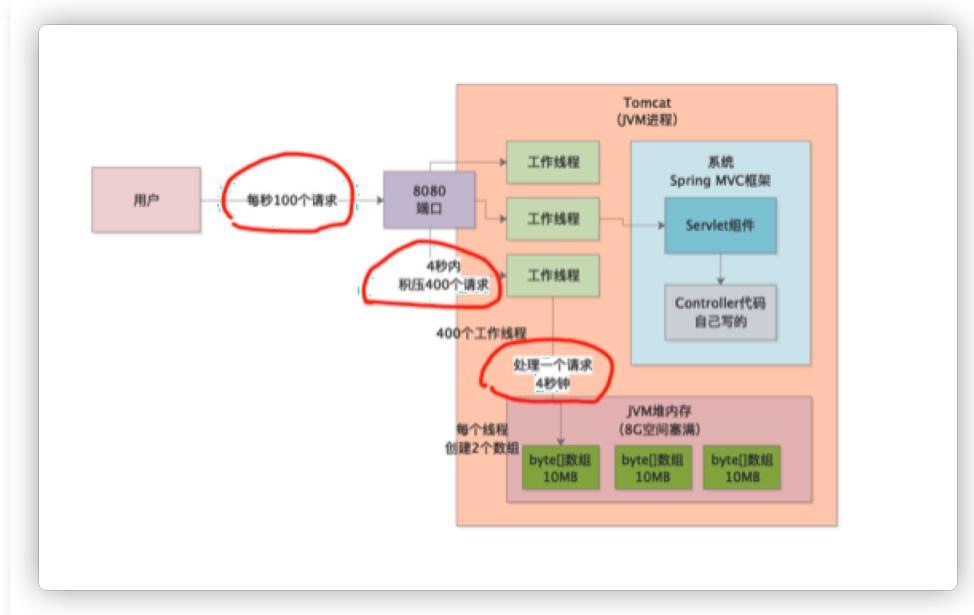
为什么Tomcat工作线程在处理一个请求的时候会创建2个10MB的数组？

到Tomcat的配置文件里搜索一下，发现了如下的一个配置：max-http-header-size: 10000000

导致Tomcat工作线程在处理请求的时候会创建2个数组，每个数组的大小如上面配置就是10MB。

梳理一遍系统运行时候的场景：

每秒100个请求，每个请求处理需要4秒，导致4秒内有400个请求同时被400个线程在处理，每个线程会根据配置创建2个数组，每个数组是10MB，占满了8G的内存。



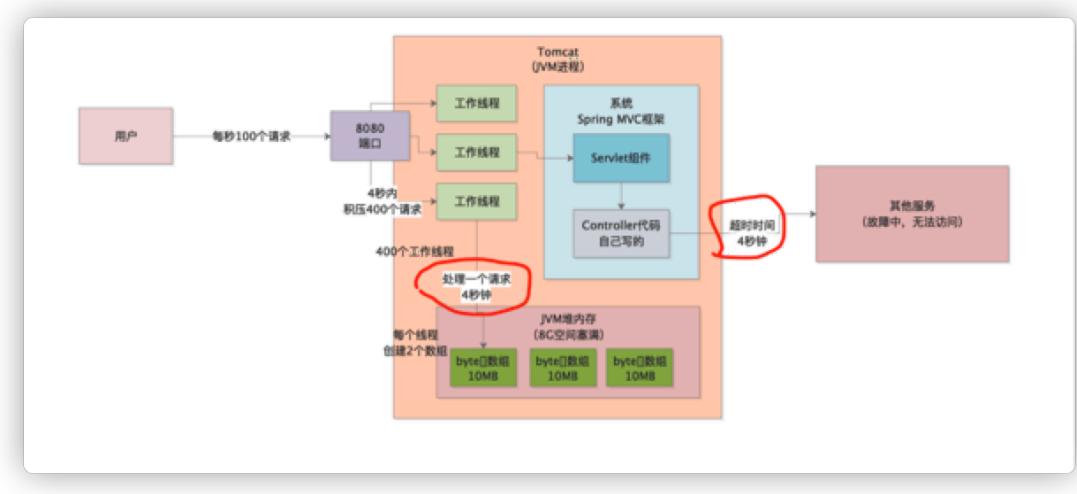
• 为什么处理一个请求需要4秒钟？

这个问题绝对是偶发性的，不是平时每次处理请求都如此，因为负责这个系统的工程师说了，平时处理一个请求也就几百毫秒的时间而已。

好，那么既然如此，唯一的办法只能是在日志里去找问题了，继续翻看事故发生时的日志，发现日志中除了OOM以外，其实有大量的服务请求调用超时的异常，类似下面那样子：

Timeout Exception....

也就是说，我们的这个系统在通过RPC调用其他系统的时候突然出现了大量的请求超时，立马翻看一下系统的RPC调用超时的配置，惊讶的发现，负责这个系统的工程师居然将服务RPC调用超时时间设置为了刚好是4秒！



之所以每个请求需要处理4秒钟，是因为下游服务故障了，网络请求都是失败的，此时会按照设置好的4秒超时时间一直卡住4秒钟之后才会抛出Timeout异常，然后请求处理结束。

解决方法：

最核心的问题就是那个超时时间设置的实在太长了，因此立马将超时时间改为1秒即可。

另外一个，对Tomcat的那个参数，max-http-header-size，可以适当调节的小一些就可以了，这样Tomcat工作线程自身为请求创建的数组，不会占据太大的内存空间的。

超时时间调为1秒个人不太认可,基于RPC的调用时不时会有处理超过1秒的情况，如果超时设置成一秒，失败请求率会太高。tomcat的工作线程数量400个已经挺多了，其实没有必要弄这么多。

我的建议：

连接的超时时间 ribbon.ConnectTimeout=2000

获取数据超时时间 ribbon.ReadTimeout=3000

工作线程数 tomcat.max-threads: 200

tomcat.accept-count: 1000

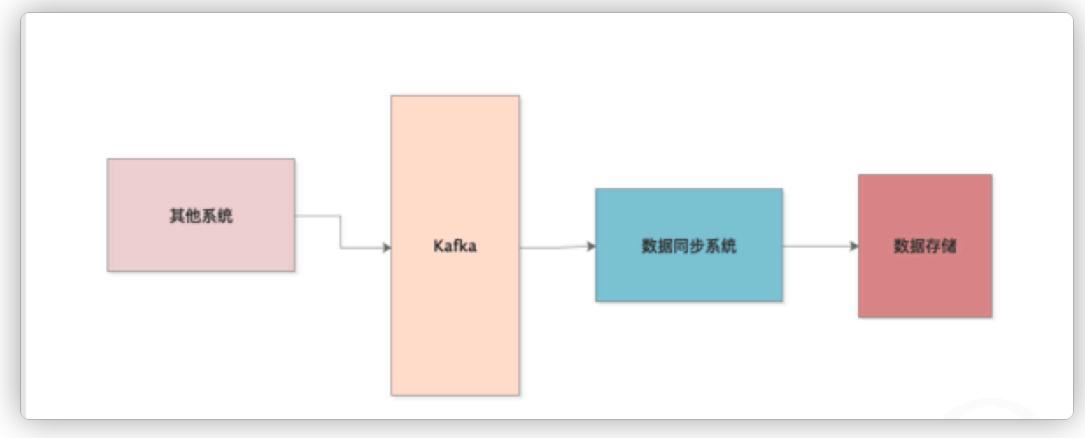
如果队列中已经有上千个请求等待处理基本可以断定处理过程出现问题,这个时候没有必要再向队列中放请求了,放进去一时半会也处理不了,不如直接快速失败,返回错误,保护自身应用

单台服务器接受20000个tcp连接已经很多了 tomcat.max-connections: 20000

12.2 数据同步系统频繁 Full gc

背景：

线上有一个数据同步系统，专门负责从另外一个系统同步数据，从kafka里消费数据，接着保存到自己的数据库。



这个系统时不时报一个内存溢出的错误，重启系统，过了一段时间会再次内存溢出。

分析：既然是每次重启过后都会在一段时间出现内存溢出，说明肯定是每次重启过后，内存都会不断上涨。

一般JVM出现内存溢出，通常就两种情况：

- 并发太高，瞬间创建大量对象
- 内存泄露，很多对象赖在内存里，无法回收

排查：

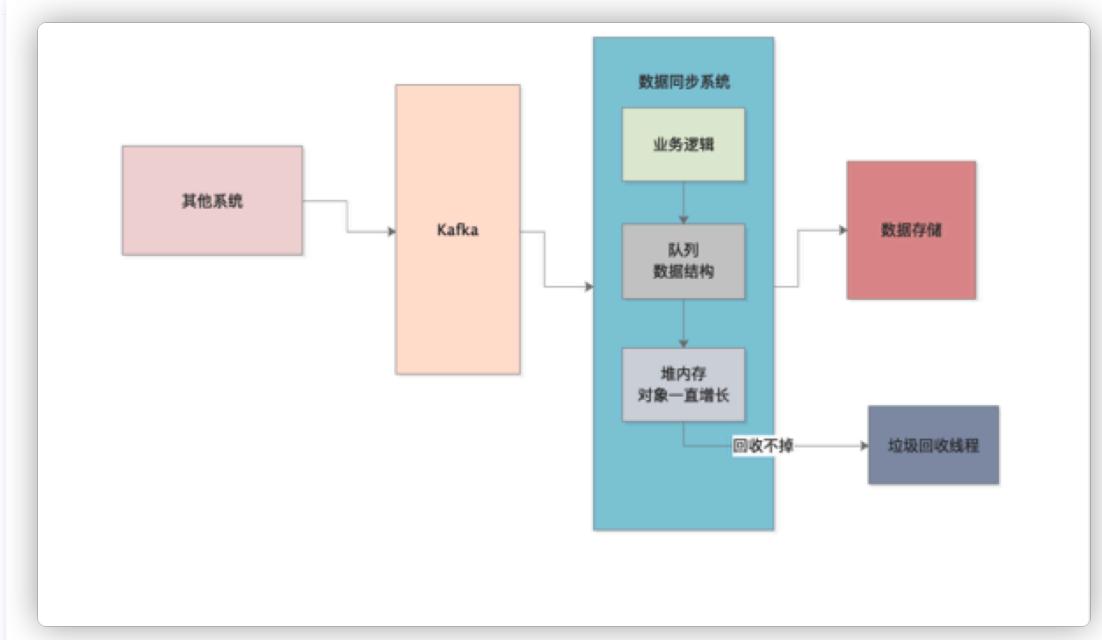
1. Jstat观察jvm情况

发现老年代一直在增长。Full gc 回收不掉老年代对象

2. Mat 分析内存快照

有一个队列数据结构，直接引用大量的数据，就是这个队列数据结构占满了内存。

那么这个队列是干什么用的？简单来说，从 Kafka消费出来的数据会先写入这个队列，接着从这个队列再慢慢写入数据库中，这个主要是要额外做一些中间的数据处理和转换，所以自己在中间又加了一个队列。



那么这个队列是怎么用的?问题就出在这里了!

大家都知道,从 Kafka 消费数据,是可以一下子消费一批出来的,比如消费几百条数据粗来。因此当时这个写代码的同学,直接就是每次消费几百条数据出来给做成一个List,然后把这个List放入到队列里去!

最后就搞成了,一个队列比如有1000个元素,每个元素都是一个List,每个List里都有几百条数据!这种做法怎么行?会导致内存中的队列里积压几十万条,甚至百万条数据!最终一定会导致内存溢出!

而且只要你数据还停留在队列中,就是没有办法被回收的

典型的生产和消费的速率没控制好。从 kafka 里消费出数据放入队列速度很快,但是从队列里消费数据进行处理写入存储的速度较慢。导致内存队列快速积压数据,导致内存溢出。

解决方法:

把队列改成定长的阻塞队列,比如最多 1024 个元素,每次从 kafka 消费出来的数据,一条一条写入队列,而不是做成list 放入队列作为一个元素。

一旦内存队列满了,此时 kafka 消费线程会停止工作,因为被队列给阻塞了。不会让内存队列中的数据过多。

