JVM

请你谈谈对JVM的理解？jdk8虚拟机和之前的变化更新

什么是OOM，什么是栈溢出（stackOverFlow）怎么分析

JVM的常用调优参数

对类加载器的认知

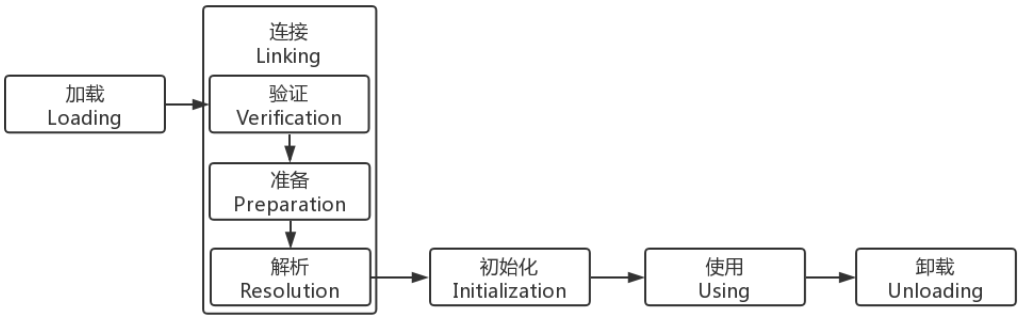
2.JVM体系结构

3.类加载器

- 类加载指将类的class的字节码读取到内存中，并将其放在**运行时数据区的方法区**内，同时在**堆区创建一个Class类型的对象**，这个Class对象就是类加载的终极目的。
- 类加载的时机-遇到new（比如new Student()）、getstatic和putstatic（读取或设置一个类的静态字段，如下代码，读取被final修饰并已在编译器把结果放入常量池的静态字段除外）、invokestatic（调用类的静态方法）这四条指令时，如果对应的类没有初始化，则要对对应的类先进行初始化。

```
Class.forName();
类.class;
对象.getClass()
```

-



加载：通过io读入字节码文件
“加载”过程主要是靠类加载器实现的，包括用户自定义类加载器。类加载过程的一个阶段，ClassLoader通过一个类的完全限定名查找此类字节码文件，并利用字节码文件创建一个class对象。

1、通过一个类的全限定名来获取定义此类的二进制字节流（class文件）。在程序运行过程中，当要访问一个类时，若发现这个类尚未被加载，并满足类初始化的条件时，就根据要被初始化的这个类的全限定名找到该类的二进制字节流，开始加载过程
2、将这个字节流的静态存储结构转化为方法区的运行时数据结构（即class对象）
3、在内存中创建一个该类的java.lang.Class对象，作为方法区该类的各种数据的访问入口

验证：
目的在于确保class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身的安全，主要包括四种验证：文件格式的验证，元数据的验证，字节码验证，符号引用验证。

class文件用16进制打开后，前八位为ca fe ba ba（magic number）后面16为是jdk的版本号

准备：为static分配内存并初始化0值。JDK1.7之前在方法区，1.7之后在堆。
为类变量（static修饰的字段变量）分配内存并且设置该类变量的初始值，（如static int i = 5 这里只是将 i 赋值为0，在初始化的阶段再把 i 赋值为5），这里不包含final修饰的static，因为final在编译的时候就已经分配了。这里不会为实例变量分配初始化，类变量会分配在方法区中，实例变量会随着对象分配到Java堆中。

解析：这里主要的任务是把常量池中的符号引用替换成直接引用

初始化：初始化过程就是调用类初始化方法的过程，完成对static修饰的类变量的手动赋值还有主动调用静态代码块。

这里是类加载的最后阶段，如果该类具有父类就进行对父类进行初始化，执行其静态初始化器（静态代码块）和静态初始化成员变量。（前面已经对static 初始化了默认值，这里我们对它进行赋值，成员变量也将被初始化）

类记载器的任务是根据类的全限定名来读取此类的二进制字节流到 JVM 中，然后转换成一个与目标类对象的java.lang.Class 对象的实例，在java 虚拟机提供三种类加载器：

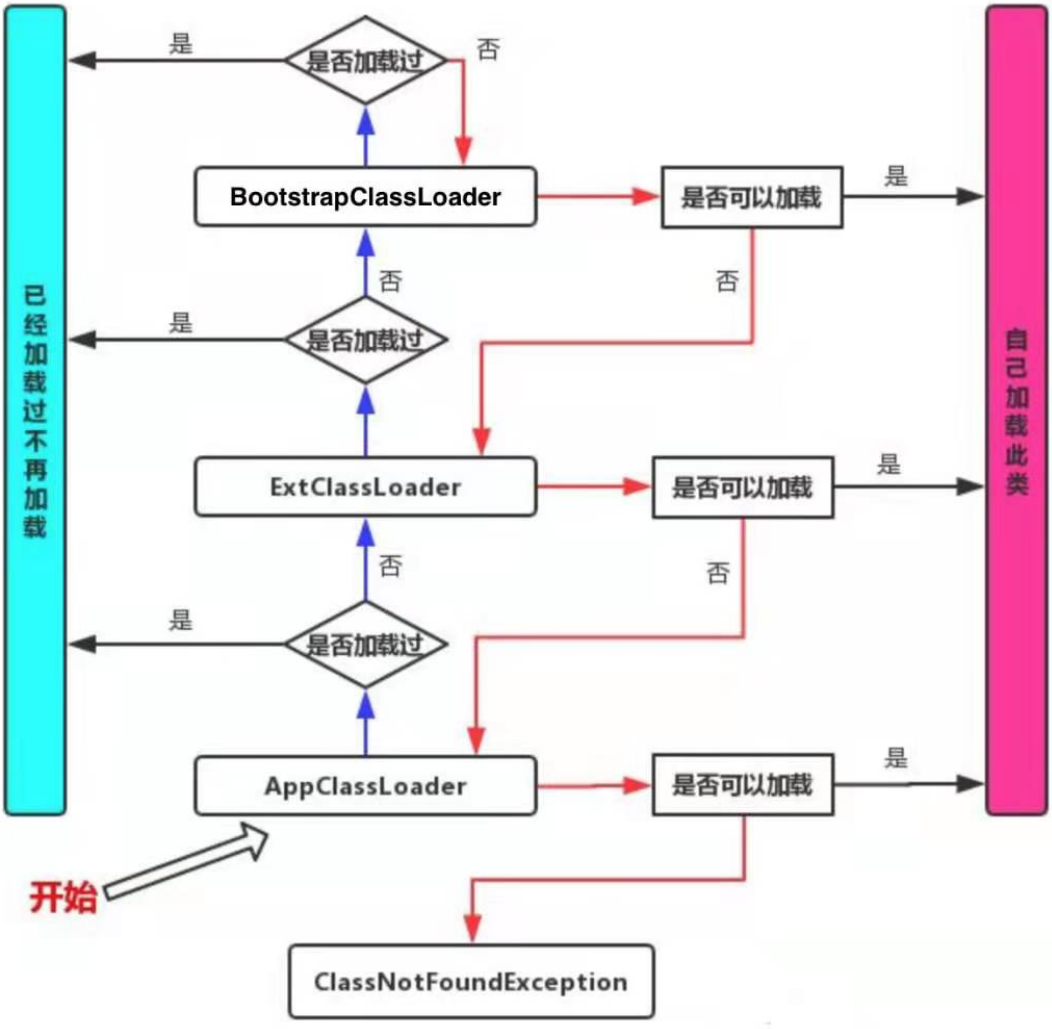
1. 启动类加载器(BootStrapClassLoader)-加载支撑JVM运行的位于JAVA_HOME\lib核心类库，比如rt.jar等
2. 扩展类加载器(ExtClassLoader)-加载JAVA_HOME\lib\ext扩展类库
3. 应用程序类加载器(AppClassLoader)-加载classpath下的类，主要加载你自己的写的类

4. 自定义加载器-加载用户自定义路径下的类(tomcat)

4.双亲委派机制

```
public Class<?> loadClass(String name) throws ClassNotFoundException {
    return loadClass(name, false);
}
//              -----??-----
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
        // 首先，检查是否已经被类加载器加载过
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            try {
                // 存在父加载器，递归的交由父加载器
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    // 直到最上面的Bootstrap类加载器
```

```
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                c = findClass(name);
            }
        }
        return c;
    }
```

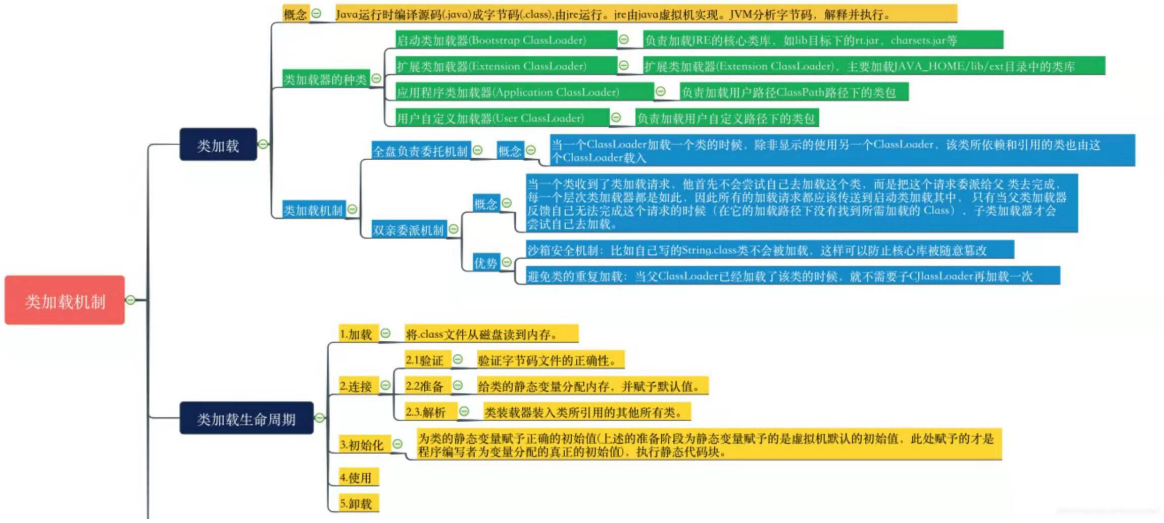


从上图中我们就更容易理解了，当一个Hello.class这样的文件要被加载时。不考虑我们自定义类加载器，首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到父加载器，然后调用父加载器的loadClass方法。父类中同理也会先检查自己是否已经加载过，如果没有再往上。注意这个类似递归的过程，直到到达Bootstrap classLoader之前，都是在检查是否加载过，并不会选择自己去加载。直到BootstrapClassLoader，已经没有父加载器了，这时候开始考虑自己是否能加

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。

5.沙箱安全机制

为什么要设计这种机制
这种设计有个好处是，如果有人想替换系统级别的类：String.java。篡改它的实现，在这种机制下这些系统的类已经被Bootstrap classLoader加载过了（为什么？因为当一个类需要加载的时候，最先去尝试加载的就是BootstrapClassLoader），所以其他类加载器并没有机会再去加载，从一定程度上防止了危险代码的植入



6.native

通过本地方法接口（Java Native Interface JNI），调用跟本地方法库(存放通过C或C++操作CPU\内存等方法)，本地方法库中的方法，可以是C，C++,python等其他语言的方法，方法运行产生的数据会保存在**本地方法栈**中。

7.PC寄存器（程序计数器）

在JVM规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。

任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的Java方法的JVM指令地址；或者，如果是在执行native方法，则是未指定值（undefned）。

它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

8.方法区

属于共享内存区域，存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

运行时常量池：属于方法区一部分，用于存放编译期生成的各种字面量和符号引用。编译器和运行期(String 的 intern())都可以将常量放入池中。内存有限，无法申请时抛出 OutOfMemoryError。

直接内存：非虚拟机运行时数据区的部分

OutOfMemoryError：会受到本机内存限制，如果内存区域总和大于物理内存限制从而导致动态扩展时出现该异常

9.堆栈，新生代，老年代，永久区

虚拟机栈：每一个正在执行的方法，都会在虚拟机栈中被分配一块空间，称之为**栈帧**。每个线程在虚拟机栈都有一个独立的空间。也就是说，虚拟机栈是线程独有（**虚拟机栈，本地方法栈，程序计数器都是线程独有，堆，方法区是线程共享**），使用-Xss进行修改，默认是1m

```
-Xss1m
-Xss1024k
-Xss1048576
```

栈帧：局部变量表，操作数栈，动态链接，返回出口、附加信息

动态链接：每个栈帧内部都包含一个指向**运行时常量池**中该战争所属的方法的引用。

Java源文件被编译到字节码文件时，所有变量和方法都会引用为**符号引用**，保存在class文件的常量池中。

动态链接的作用就是为了将这些符号引用转换为方法的直接引用

```
如果调用一个方法，有静态绑定和动态绑定
静态绑定，是在编译时，明确知道该调用的方法的字节码是谁，在哪（private final static）
动态绑定，在编译时，不知道调用的具体方法体是哪个，在运行时才知道
```

动态链接找到方法区中的运行时常量池中的method ref 找到对应方法的方法表，通过方法表里的引用找到具体方法字节码存在的地址

```
invokeinterface
invokevirtual 调用非private方法（动态绑定方法） vtable(虚拟表)
invokespeciel private、构造方法、final
invokestatic  调用static
```

本地方法栈：区别于 Java 虚拟机栈的是，Java 虚拟机栈为虚拟机执行 Java 方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。也会有 StackOverflowError 和 OutOfMemoryError 异常。

堆：对于绝大多数应用来说，这块区域是 JVM 所管理的内存中最大的一块。线程共享，主要是存放对象实例和数组。内部会划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer, TLAB)。可以位于物理上不连续的空间，但是逻辑上要连续。

元空间

从方法区(PermGen)到元空间(Metaspace)

- **方法区**

1. JDK1.8以前的HotSpot JVM有**方法区**，元空间（meta space）。
2. 方法区用于存放已被虚拟机加载的**类元信息、常量、静态变量，即编译器编译后的代码。**

- **为什么要用Metaspace替代方法区**
  随着动态类加载的情况越来越多，这块内存变得不太可控，如果设置小了，系统运行过程中就容易出现内存溢出，设置大了又浪费内存。

JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）

元空间的本质和永久代类似，**都是对JVM规范中方法区的实现。**不过元空间与永久代之间**最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。**因此，默认情况下，元空间的大小仅受本地内存限制，但可以通过以下参数来指定元空间的大小：

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。
-XX:MaxMetaspaceSize，最大空间，默认是没有限制的。

除了上面两个指定大小的选项以外，还有两个与 GC 相关的属性：
-XX:MinMetaspaceFreeRatio，在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集
-XX:MaxMetaspaceFreeRatio，在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

很多人认为方法区等同与永久代，永久代既然没了，方法区也就没了。但我认为方法区只是一种逻辑上的概念，永久代指物理上的堆内存的一块空间，这块实际的空间完成了方法区存储字节码、静态变量、常量的功能等等。既然如此，现在元空间也可以认为是新的方法区的实现了。

几种常量池：

（1）静态常量池：即*.class文件中的常量池，在Class文件结构中，最头的4个字节存储魔数（0xCAFEBABE），用于确定一个文件是否能被JVM接受，接着4个字节用于存储版本号，前2个为次版本号，后2个主版本号，再接着是用于存放常量的常量池，由于常量的数量是不固定的，所以常量池的入口放置一个U2类型的数据(constant_pool_count)存储常量池容量计数值。

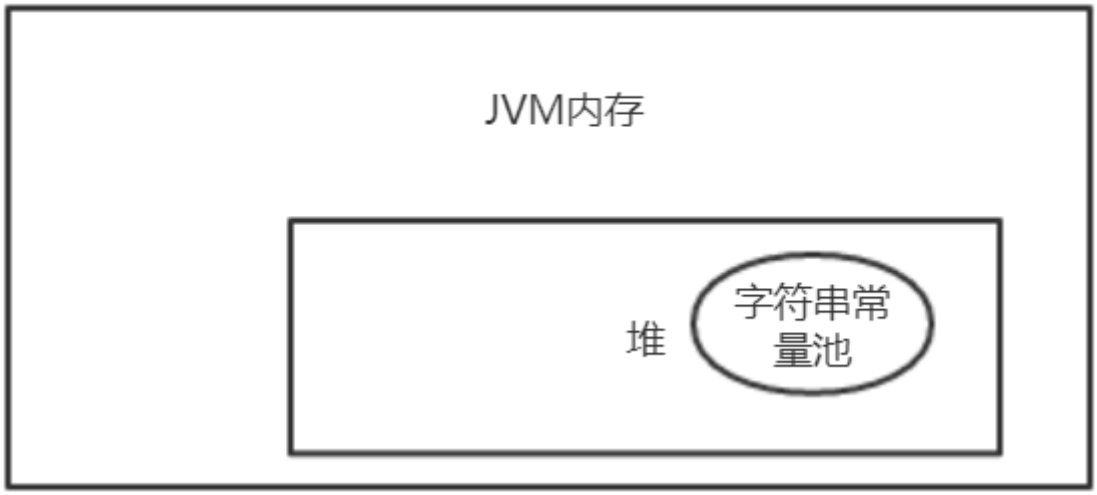
这种常量池占用class文件绝大部分空间，主要用于存放两大类常量：字面量和符号引用量，字面量相当于Java语言层面常量的概念，如文本字符串、基础数据、声明为final的常值等；符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：类和接口的全限定名、字段名称描述符、方法名称描述符。类的加载过程中的链接部分的解析步骤就是把符号引用替换为直接引用，即把那些描述符（名字）替换为能直接定位到字段、方法的引用或句柄（地址）。

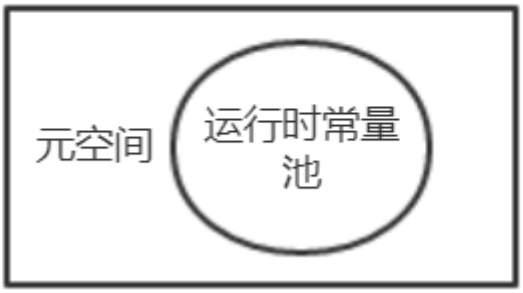
（2）运行时常量池：虚拟机会将各个class文件中的常量池载入到运行时常量池中，即编译期间生成的字面量、符号引用，总之就是装载class文件。为什么它叫运行时常量池呢？因为这个常量池在运行时，里面的常量是可以增加的。如：“+”连接字符生成新字符后调用 intern（）方法、生成基础数据的包装类型等等。

（3）字符串常量池 ： 字符串常量池可以理解为是分担了部分运行时常量池的工作。加载时，对于class文件的静态常量池，如果是字符串就会被装到字符串常量池中。

（4）整型常量池：Integer，类似字符串常量池。管理-128--127的常量。类似的还有Character、Long等常量池（基本数据类型没有，Double、Float也没有常量池）

在永久代移除后，字符串常量池也不再放在永久代了，但是也没有放到新的方法区---元空间里，而是留在了堆里（为了方便回收？）。运行时常量池当然是随着搬家到了元空间里，毕竟它是装类的重要信息的，有它的地方才称得上是方法区





10. GC-常用算法

1.启动Java垃圾回收

作为一个自动的过程，程序员不需要在代码中显示地启动垃圾回收过程。`System.gc()`和`Runtime.gc()`用来请求JVM启动垃圾回收。

虽然这个请求机制提供给程序员一个启动 GC 过程的机会，但是启动由 JVM负责。JVM可以拒绝这个请求，所以并不保证这些调用都将执行垃圾回收。启动时机的选择由JVM决定，并且取决于堆内存中Eden区是否可用。JVM将这个选择留给了Java规范的实现，不同实现具体使用的算法不尽相同。

垃圾回收的过程

Eden 区：对象优先分配在eden区。虚拟机为每个对象分配一个年龄计数器（age）

当Eden区没有足够的空间创建对象时，JVM将会发起一次MinorGC（young GG，指发生在新生代的垃圾回收动作），速度非常快。

Survivor 区（S0 和 S1）：作为年轻代 GC（Minor GC）周期的一部分，当对象经历过一次MinorGC并且存活，存活的对象（仍然被引用的）从 Eden 区被移动到 Survivor 区的 S0 中，age=1。类似的，垃圾回收器会扫描 S0 然后将存活的实例移动到 S1 中。当age=15时，会被存放入老年代

注，如果在eden区经历一次垃圾回收后Survivor区存放不下，有可能会直接移动到老年代

老年代 GC（Old GC）：相对于 Java 垃圾回收过程，老年代是实例生命周期的最后阶段。Old GC 扫描老年代的垃圾回收过程。如果实例不再被引用，那么它们会被标记为回收，否则它们会继续留在老年代中。

年轻代在每次minorGC之前都会计算老年代剩余可用空间，如果空间小于年轻代所有对象大小的和（包括垃圾），会检查-XX:-HandlePromotionFailure 参数（1.8默认开启），如果有，则会检查老年代剩余可用空间是否大于每次MinorGC后进入老年代的对象的平均大小，如果小于，则会触发FULL GC(stop the world 停止全部线程)，如果依然不够，则OOM

大对象：字符串、数组（可以通过 -XX:PretenureSizeThreshold设置大对象的大小），大对象会直接进入到老年代（避免大对象分配内存时的复制操作而降低效率）

```
Partial GC：并不收集整个GC堆的模式
    Young GC：只收集young gen的GC/minior GC
    Old GC：只收集old gen的GC。只有CMS的concurrent collection是这个模式
    Mixed GC：收集整个young gen以及部分old gen的GC。只有G1有这个模式
Full GC：收集整个堆，包括young gen、old gen、metaspace（如果存在的话）等所有部分的模式。
```

内存碎片：一旦实例从堆内存中被删除，其位置就会变空并且可用于未来实例的分配。这些空出的空间将会使整个内存区域碎片化。为了实例的快速分配，需要进行碎片整理。基于垃圾回收器的不同选择，回收的内存区域要么被不停地被整理，要么在一个单独的GC进程中完成。

垃圾回收中实例的终结

在释放一个实例和回收内存空间之前，Java 垃圾回收器会调用实例各自的 `finalize()` 方法，从而该实例有机会释放所持有的资源。虽然可以保证 `finalize()` 会在回收内存空间之前被调用，但是没有指定的顺序和时间。多个实例间的顺序是无法被预知，甚至可能会并行发生。程序不应该预先调整实例之间的顺序并使用 `finalize()` 方法回收资源。

- 任何在 finalize过程中未被捕获的异常会自动被忽略，然后该实例的 finalize 过程被取消。
- JVM 规范中并没有讨论关于弱引用的垃圾回收机制，也没有很明确的要求。具体的实现都由实现方决定。
- 垃圾回收是由一个守护线程完成的。

对象什么时候符合垃圾回收的条件?

- 所有实例都没有活动线程访问。
- 没有被其他任何实例访问的循环引用实例。
- Java 中有不同的引用类型。判断实例是否符合垃圾收集的条件都依赖于它的引用类型

引用类型	垃圾收集
强引用（Strong Reference）	不符合垃圾收集
软引用（Soft Reference）	垃圾收集可能会执行，但会作为最后的选择
弱引用（Weak Reference）	符合垃圾收集
虚引用（Phantom Reference）	符合垃圾收集

Java中默认声明的就是强引用，比如：

```
Object obj = new Object(); //只要obj还指向Object对象，Object对象就不会被回收
obj = null;  //手动置null
```

只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

软引用是用来描述一些非必需但仍有用的对象。**在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。**这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。
在JDK1.2之后，用java.lang.ref.SoftReference类来表示软引用。

下面以一个例子来进一步说明强引用和软引用的区别：
在运行下面的Java代码之前，需要先配置参数 -Xms2M -Xmx3M，将JVM的初始堆内存设为2M，最大可用堆内存为3M。

首先先来测试一下强引用，在限制了JVM内存的前提下，下面的代码运行正常

```
public class TestOOM {

    public static void main(String[] args) {
         testStrongReference();
    }
    private static void testStrongReference() {
        // 当 new byte为 1M 时，程序运行正常
        byte[] buff = new byte[1024 * 1024 * 1];
    }
}
```

但是如果我们将

```
byte[] buff = new byte[1024 * 1024 * 1];
```

替换为创建一个大小为3M的字节数组

```
byte[] buff = new byte[1024 * 1024 * 3];
```

则内存不够使用，程序直接报错，强引用并不会被回收

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

接着来看一下软引用会有什么不一样，在下面的示例中连续创建了10个大小为1M的字节数组，并赋值给了软引用，然后循环遍历将这些对象打印出来。

```
public class TestOOM {
    private static List<Object> list = new ArrayList<>();
    public static void main(String[] args) {
         testSoftReference();
    }
    private static void testSoftReference() {
        for (int i = 0; i < 10; i++) {
            byte[] buff = new byte[1024 * 1024];
            SoftReference<byte[]> sr = new SoftReference<>(buff);
            list.add(sr);
```

```
        }


        for(int i=0; i < list.size(); i++){
            Object obj = ((SoftReference) list.get(i)).get();
            System.out.println(obj);
        }

    }

}
```

打印结果：

```
"C:\Program Files\Java\jdk1.8.0_271\bin\java.exe" ...
null
null
null
null
null
null
null
null
null
[B@74a14482
```

我们发现无论循环创建多少个软引用对象，打印结果总是只有最后一个对象被保留，其他的obj全都被置空回收了。
这里就说明了在内存不足的情况下，软引用将会被自动回收。
值得注意的一点，即使有 byte[] buff 引用指向对象，且 buff 是一个strong reference，但是 SoftReference sr 指向的对象仍然被回收了，这是因为Java的编译器发现了在之后的代码中，buff 已经没有被使用了，所以自动进行了优化。
如果我们将上面示例稍微修改一下：

```
    private static void testSoftReference() {
        byte[] buff = null;

        for (int i = 0; i < 10; i++) {
            buff = new byte[1024 * 1024];
            SoftReference<byte[]> sr = new SoftReference<>(buff);
            list.add(sr);
        }

        System.gc(); //主动通知垃圾回收

        for(int i=0; i < list.size(); i++){
            Object obj = ((SoftReference) list.get(i)).get();
            System.out.println(obj);
        }

        System.out.println("buff: " + buff.toString());
    }
```

则 buff 会因为强引用的存在，而无法被垃圾回收，从而抛出OOM的错误。

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

如果一个对象惟一剩下的引用是软引用，那么该对象是软可及的（softly reachable）。垃圾收集器并不像其收集弱可及的对象一样尽量地收集软可及的对象，相反，它只在真正“需要”内存时才收集软可及的对象。

三，弱引用

弱引用的引用强度比软引用要更弱一些，**无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收。**在 JDK1.2 之后，用 java.lang.ref.WeakReference 来表示弱引用。
我们以与软引用同样的方式来测试一下弱引用：

```
    private static void testWeakReference() {
        for (int i = 0; i < 10; i++) {
            byte[] buff = new byte[1024 * 1024];
            WeakReference<byte[]> sr = new WeakReference<>(buff);
            list.add(sr);
        }
        System.gc(); //强制通知垃圾回收
        for(int i=0; i < list.size(); i++){
            Object obj = ((WeakReference) list.get(i)).get();
            System.out.println(obj);
        }
    }
```

打印结果：

```
"C:\Program Files\Java\jdk1.8.0_271\bin\java.exe" ...
null
null
null
null
null
null
null
null
null
null
```

可以发现所有被弱引用关联的对象都被垃圾回收了。

四，虚引用

虚引用是最弱的一种引用关系，如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收，在 JDK1.2 之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个null，也就是说将永远无法通过虚引用来获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

```
public class PhantomReference<T> extends Reference<T> {
    /**
     * Returns this reference object's referent.  Because the referent of a
     * phantom reference is always inaccessible, this method always returns
     * <code>null</code>.
     *
     * @return  <code>null</code>
     */
    public T get() {
        return null;
    }
    public PhantomReference(T referent, ReferenceQueue<? super T> q) {
        super(referent, q);
    }
}
```

那么传入它的构造方法中的 ReferenceQueue 又是如何使用的呢?

五，引用队列（ReferenceQueue）

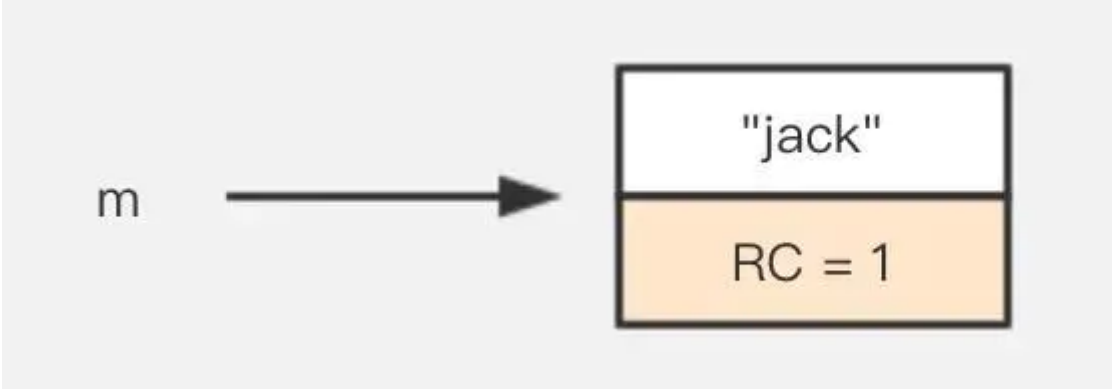
引用队列可以与软引用、弱引用以及虚引用一起配合使用，当垃圾回收器准备回收一个对象时，如果发现它还有引用，那么就会在回收对象之前，把这个引用加入到与之关联的引用队列中去。程序可以通过判断引用队列中是否已经加入了引用，来判断被引用的对象是否将要被垃圾回收，这样就可以在对象被回收之前采取一些必要的措施。

与软引用、弱引用不同，虚引用必须和引用队列一起使用。

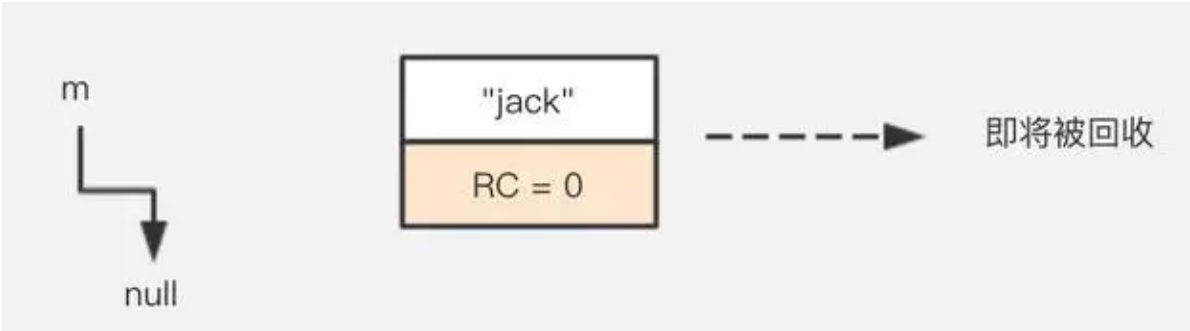
2. 垃圾判断算法

2.1 引用计数法

给每个对象添加一个计数器，当有地方引用该对象时计数器加1，当引用结束时计数器减1。用对象计数器是否为0来判断对象是否可被回收。缺点：**无法解决循环引用的问题**。



先创建一个字符串，`String m = new String("jack");`，这时候 "jack" 有一个引用，就是m。然后将m设置为null，这时候 "jack" 的引用次数就等于 0 了，在引用计数算法中，意味着这块内容就需要被回收了。



引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的`Stop-The-World`的垃圾收集机制。

> 什么是Stop the world
> Java中Stop-The-World机制简称STW，是在执行垃圾收集算法时，Java应用程序的其他所有线程都被挂起（除了垃圾收集帮助器之外）。Java中一种全局暂停现象，全局停顿，所有Java代码停止，native代码可以执行，但不能与JVM交互；这些现象多半是由于gc引起。

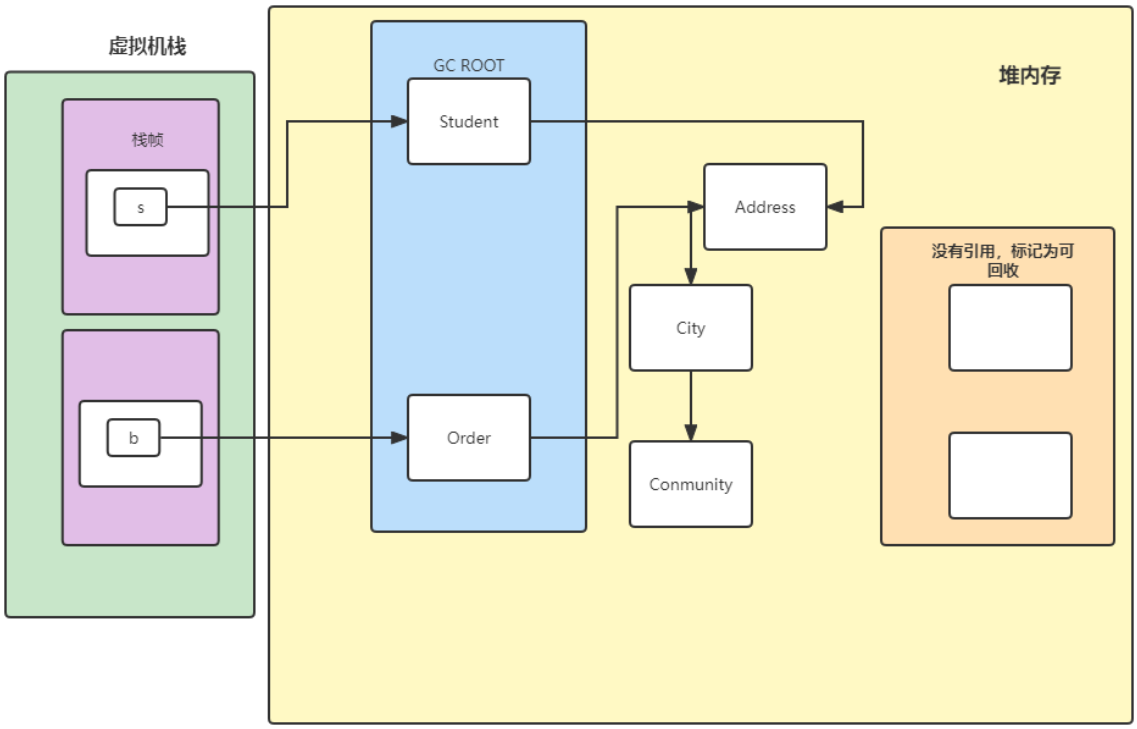
看似很美好，但我们知道JVM的垃圾回收就是`Stop-The-World`的，那是什么原因导致我们最终放弃了引用计数算法呢？看下面的例子。

```
public class ReferenceCountingGC {

  public Object instance;

  public ReferenceCountingGC(String name) {
  }

  public static void testGC(){

    ReferenceCountingGC a = new ReferenceCountingGC("objA");
    ReferenceCountingGC b = new ReferenceCountingGC("objB");

    a.instance = b;
    b.instance = a;

    a = null;
    b = null;
  }
}
```

我们可以看到，最后这2个对象已经不可能再被访问了，但由于他们相互引用着对方，导致它们的引用计数永远都不会为0，通过引用计数算法，也就永远无法通知GC收集器回收它们。

2.2 可达性分析算法

通过一系列称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为“引用链”，当一个对象到 GC Roots 没有任何的引用链相连时(从 GC Roots 到这个对象不可达)时，证明此对象不可用。以下图为例：



在Java语言中，可作为GC Roots的对象包含以下几种：

虚拟机栈(栈帧中的本地变量表)中引用的对象。遍历虚拟机栈，将变量指向的对象作为CG ROOT
方法区中静态属性引用的对象
方法区中常量引用的对象
本地方法栈中(Native方法)引用的对象

- **虚拟机栈中引用的对象**

```
public class StackLocalParameter {

  public StackLocalParameter(String name) {}

  public static void testGC() {
    StackLocalParameter s = new StackLocalParameter("localParameter");
    s = null;
  }
}
```

> 此时的s，即为GC Root，当s置空时，localParameter对象也断掉了与GC Root的引用链，将被回收。

- **方法区中类静态属性引用的对象**

```
public class MethodAreaStaicProperties {

  public static MethodAreaStaicProperties m;

  public MethodAreaStaicProperties(String name) {}

  public static void testGC(){
    MethodAreaStaicProperties s = new MethodAreaStaicProperties("properties");
    s.m = new MethodAreaStaicProperties("parameter");
    s = null;
  }
}
```

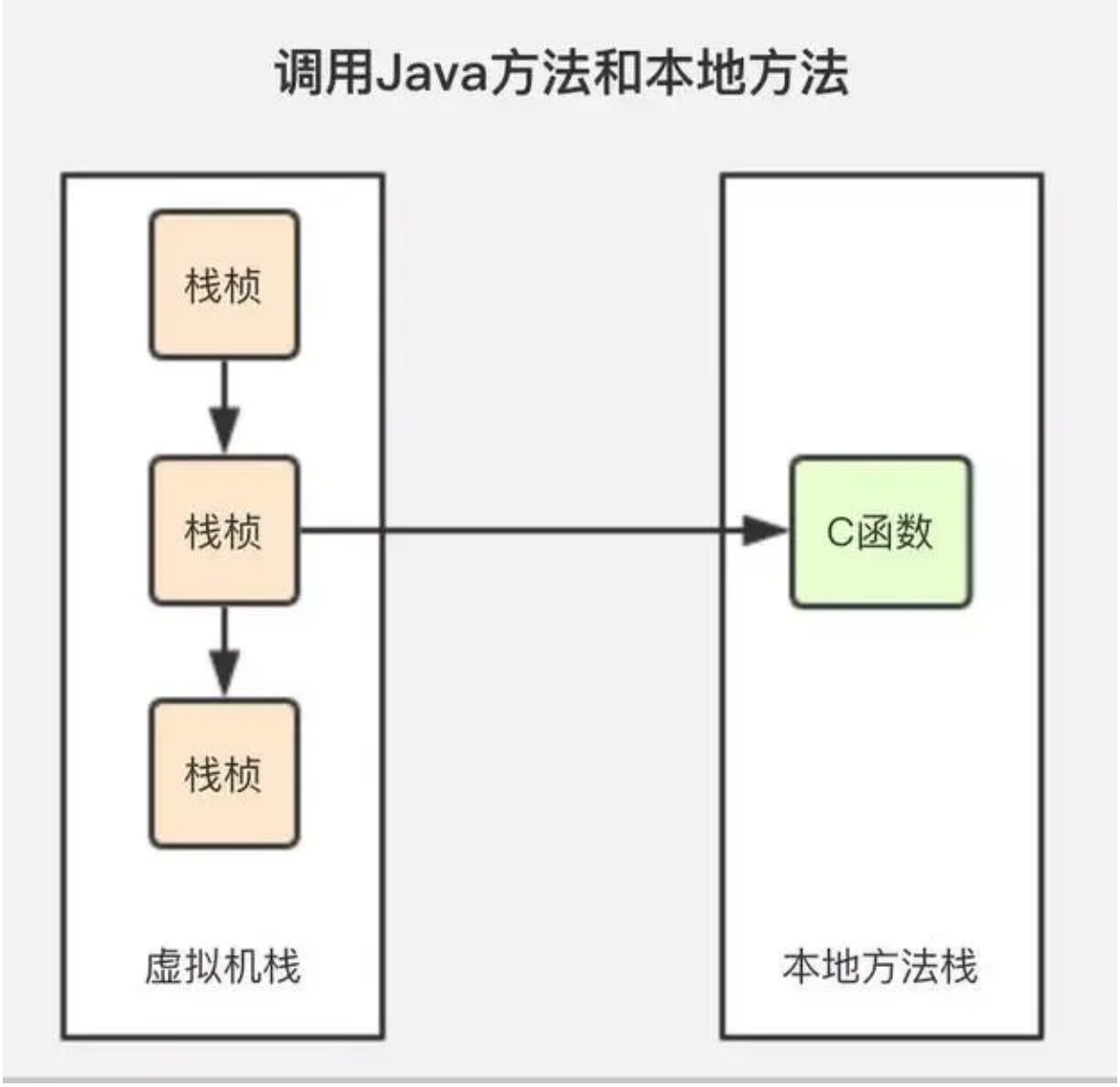
> 此时的s，即为GC Root，s置为null，经过GC后，s所指向的properties对象由于无法与GC Root建立关系被回收。而m作为类的静态属性，也属于GC Root，parameter 对象依然与GC root建立着连接，所以此时parameter对象并不会被回收。

- **方法区中常量引用的对象**

```
public class MethodAreaStaicProperties {

  public static final MethodAreaStaicProperties m =
MethodAreaStaicProperties("final");

  public MethodAreaStaicProperties(String name) {}

  public static void testGC() {
    MethodAreaStaicProperties s = new
MethodAreaStaicProperties("staticProperties");
    s = null;
  }
}
```

> m即为方法区中的常量引用，也为GC Root，s置为null后，final对象也不会因没有与GC Root建立联系而被回收。

- **本地方法栈中引用的对象**

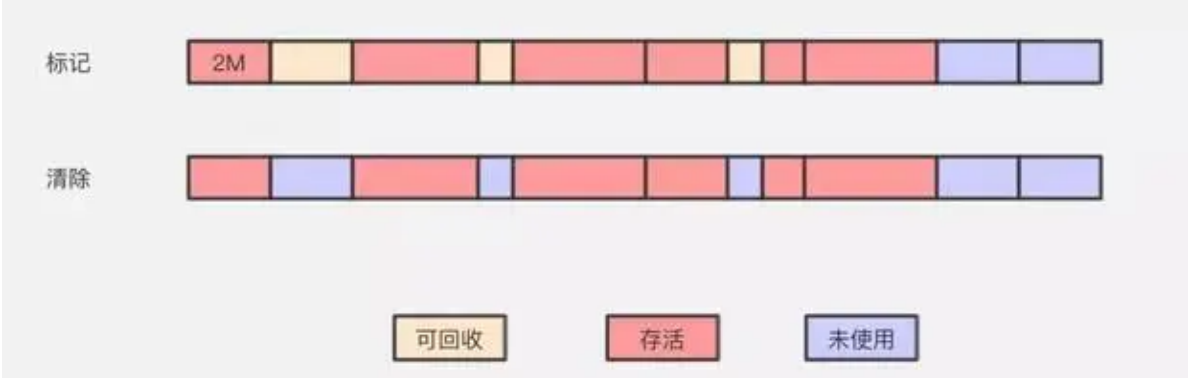


> 任何native接口都会使用某种本地方法栈，实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。

3. 垃圾回收算法

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是开始进行垃圾回收，但是这里面涉及到一个问题是：如何高效地进行垃圾回收。这里我们讨论几种常见的垃圾收集算法的核心思想。

3.1 标记-清除算法



标记清除算法（Mark-Sweep）是最基础的一种垃圾回收算法，它分为2部分，先把内存区域中的这些对象进行标记，哪些属于可回收标记出来，然后把这些垃圾拎出来清理掉。就像上图一样，清理掉的垃圾就变成未使用的内存区域，等待被再次使用。但它存在一个很大的问题，那就是**内存碎片**。

上图中等方块的假设是2M，小一些的是1M，大一些的是4M。等我们回收完，内存就会切成了很多段。我们知道开辟内存空间时，需要的是连续的内存区域，这时候我们需要一个2M的内存区域，其中有2个1M是没法用的。这样就导致，其实我们本身还有这么多的内存的，但却用不了。

3.2 复制算法

复制算法（Copying）是在标记清除算法基础上演化而来，解决标记清除算法的内存碎片问题。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。保证了内存的连续可用，内存分配时也就不用考虑内存碎片等复杂情况。但坏处也是显而易见的，就是直接损失了一半的可用内存。

3.3 标记-压缩（标记-整理）算法

标记-压缩算法标记过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，再清理掉端边界以外的内存区域。

标记压缩算法解决了内存碎片的问题，也规避了复制算法只能利用一半内存区域的弊端。标记压缩算法对内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法要差很多。一般是把Java堆分为**新生代**和**老年代**，这样就可以根据各个年代的特点采用最适当的收集算法。

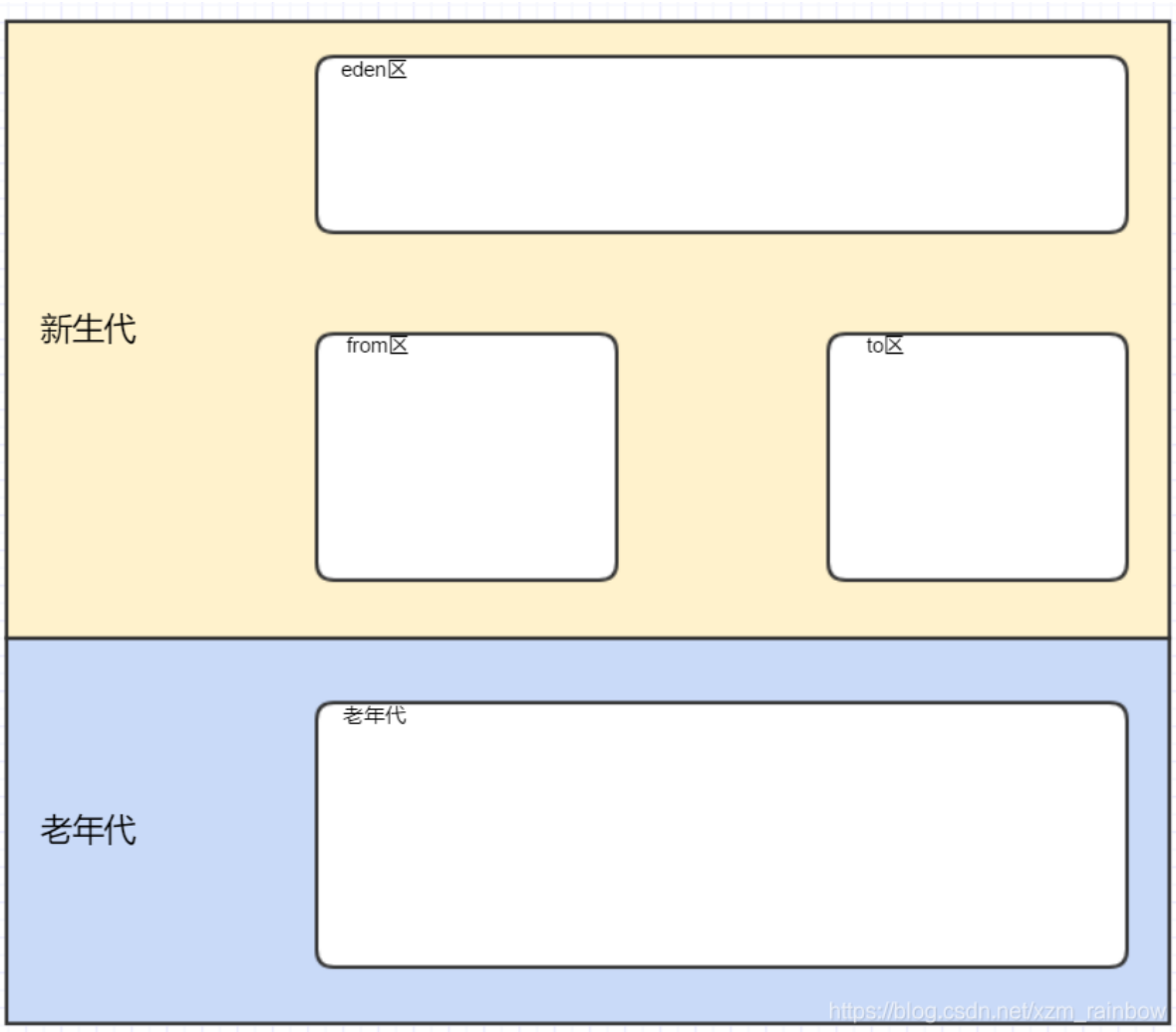
3.4 分代收集算法

分代收集算法分代收集算法严格来说并不是一种思想或理论，而是融合上述3种基础的算法思想，而产生的针对不同情况所采用不同算法的一套组合拳，根据对象存活周期的不同将内存划分为几块。

- 在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用**复制算法**，只需要付出少量存活对象的复制成本就可以完成收集。
- 在老年代中，因为对象存活率高、没有额外空间对它进行分配担保，就必须使用**标记-清理算法**或者**标记-压缩算法**来进行回收。

3.5分代算法执行过程

首先简述一下新生代GC的整个过程：新创建的对象总是在eden区中出生，当eden区满时，会触发Minor GC，此时会将eden区中的存活对象复制到from和to中一个没有被使用的空间中，假设是to区（正在被使用的from区中的存活对象也会被复制到to区中）。



有几种情况，对象会晋升到老年代：

超大对象会直接进入到老年代（受虚拟机参数-XX:PretenureSizeThreshold参数影响，默认值0，即不开启，单位为Byte，例如：3145728=3M，那么超过3M的对象，会直接晋升老年代）

如果to区已满，多出来的对象也会直接晋升老年代

复制15次(15岁)后，依然存活的对象，也会进入老年代

此时eden区和from区都是垃圾对象，可以直接清除。

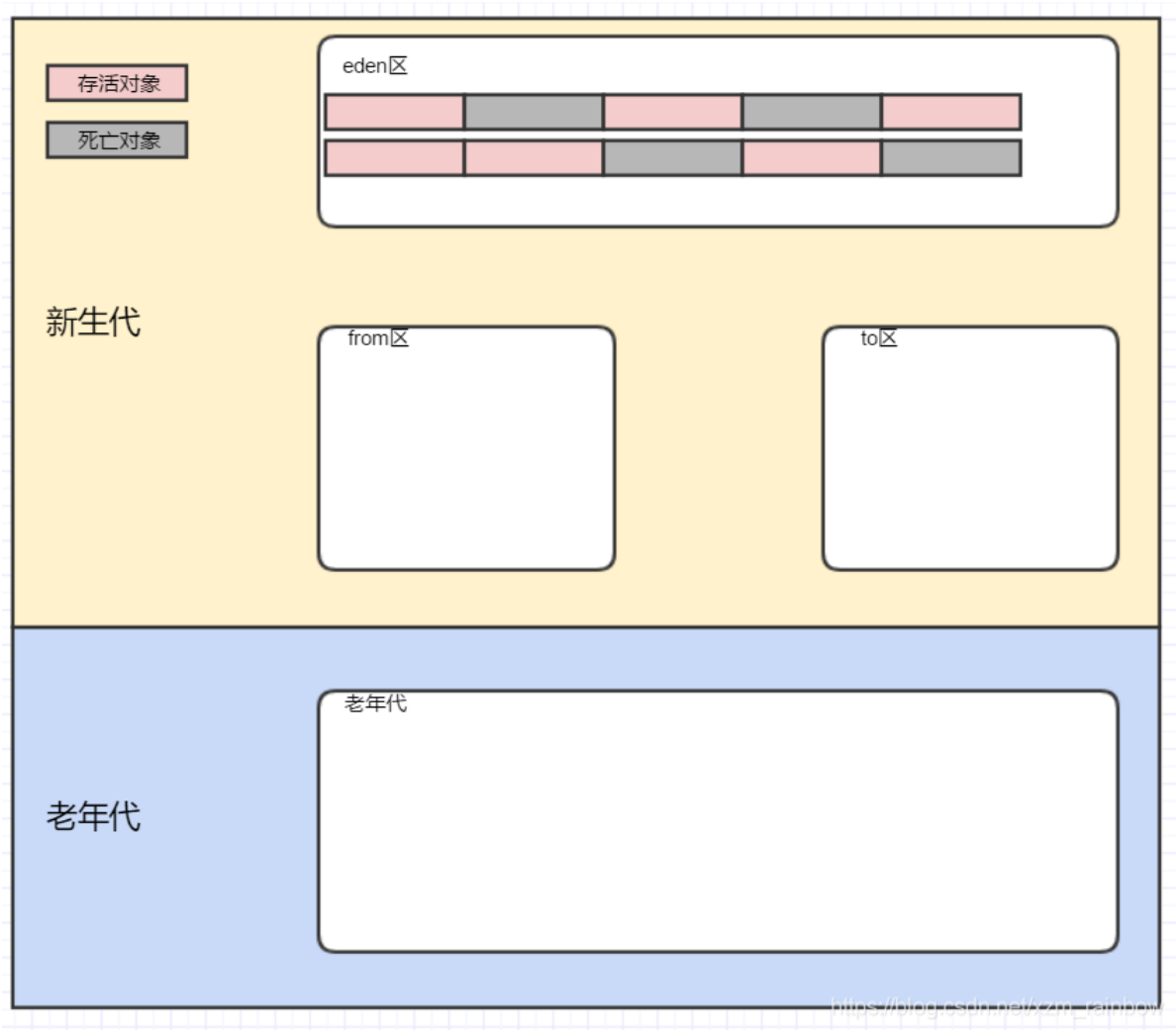
PS：为什么复制15次(15岁)后，被判定为高龄对象，晋升到老年代呢?

因为每个对象的年龄是存在对象头中的，对象头用4bit存储了这个年龄数，而4bit最大可以表示十进制的15，所以是15岁。

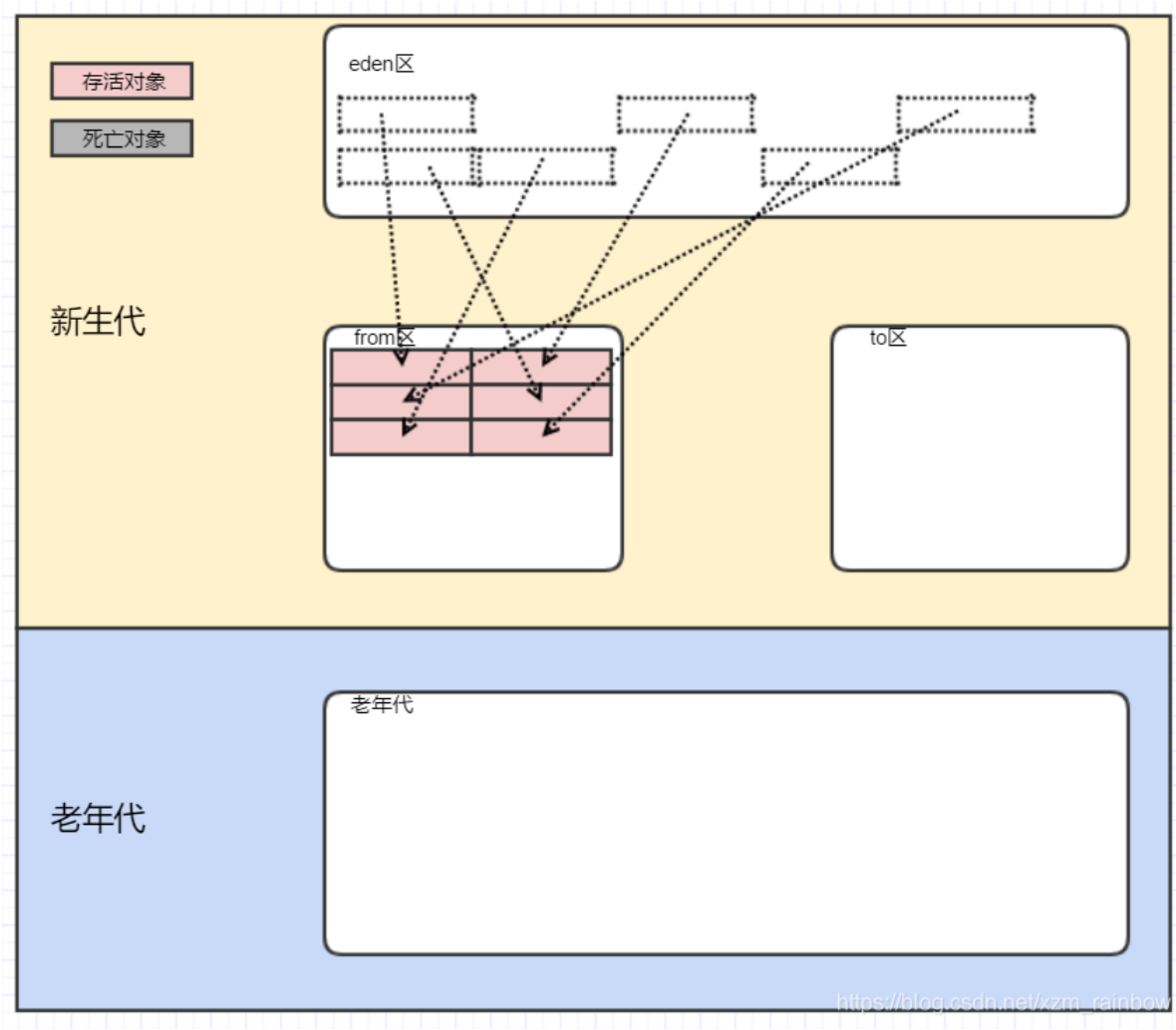
下面从JVM启动开始，描述GC的过程。

JVM刚启动并初始化完成后，几块内存空间分配完毕，此时状态如上图所示。

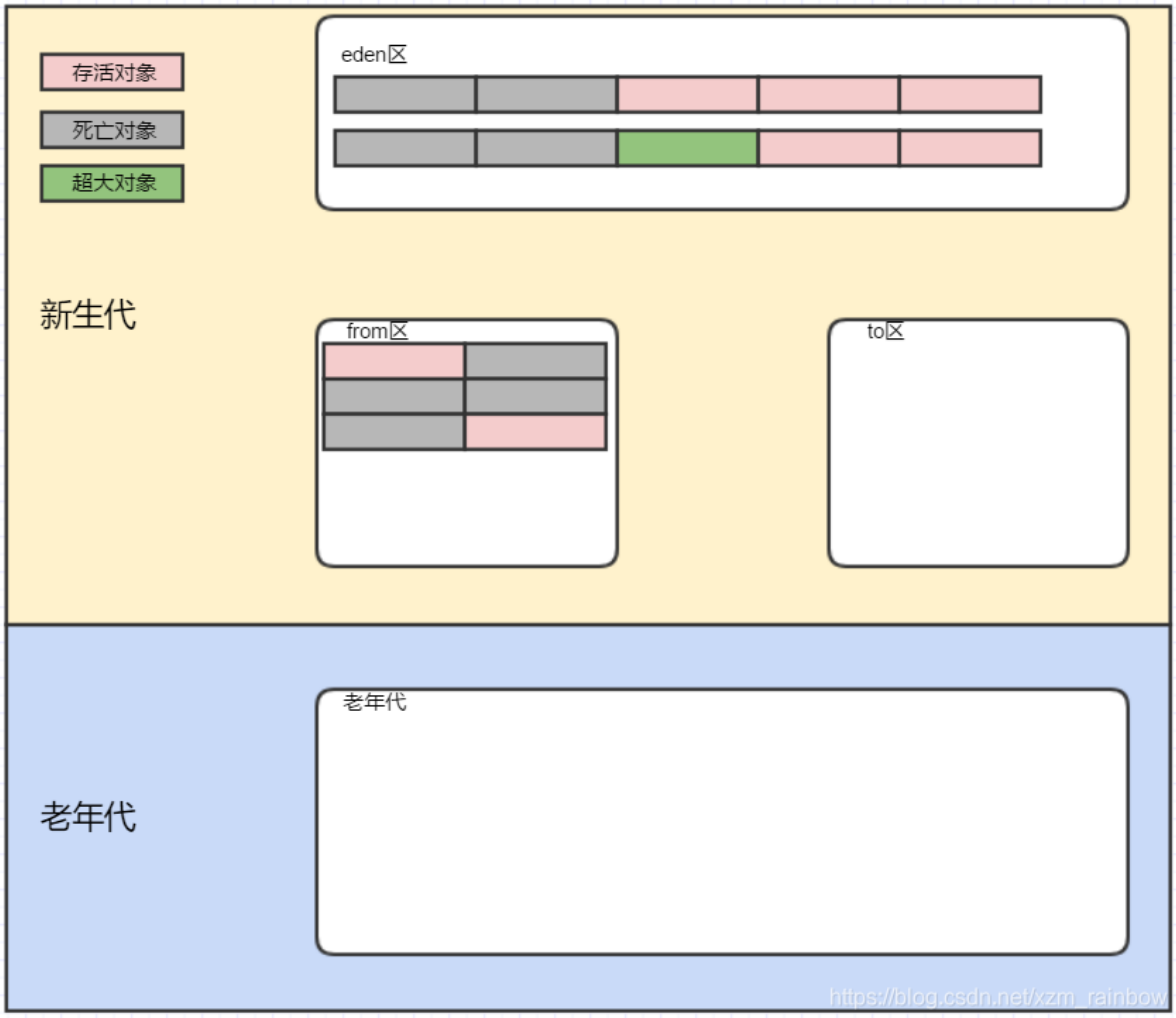
（1）新创建的对象总是会出生在eden区



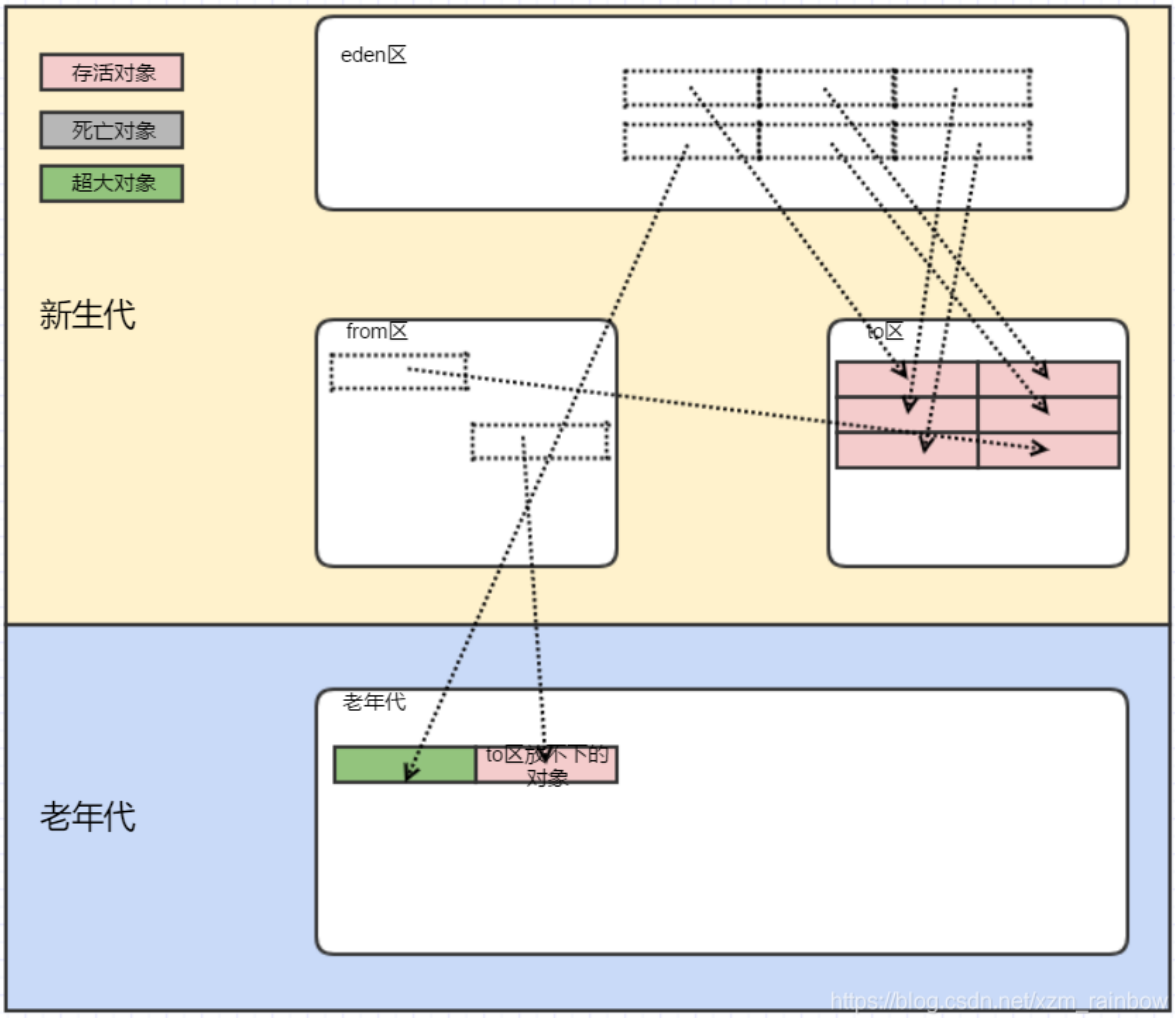
（2）当eden区满的时候，会触发一次Minor GC，此时会从from和to区中找一个没有使用的空间，将eden区中还存活的对象复制过去（第一次from和to都是空的，使用from区），被复制的对象的年龄会+1，并清除eden区中的垃圾对象。



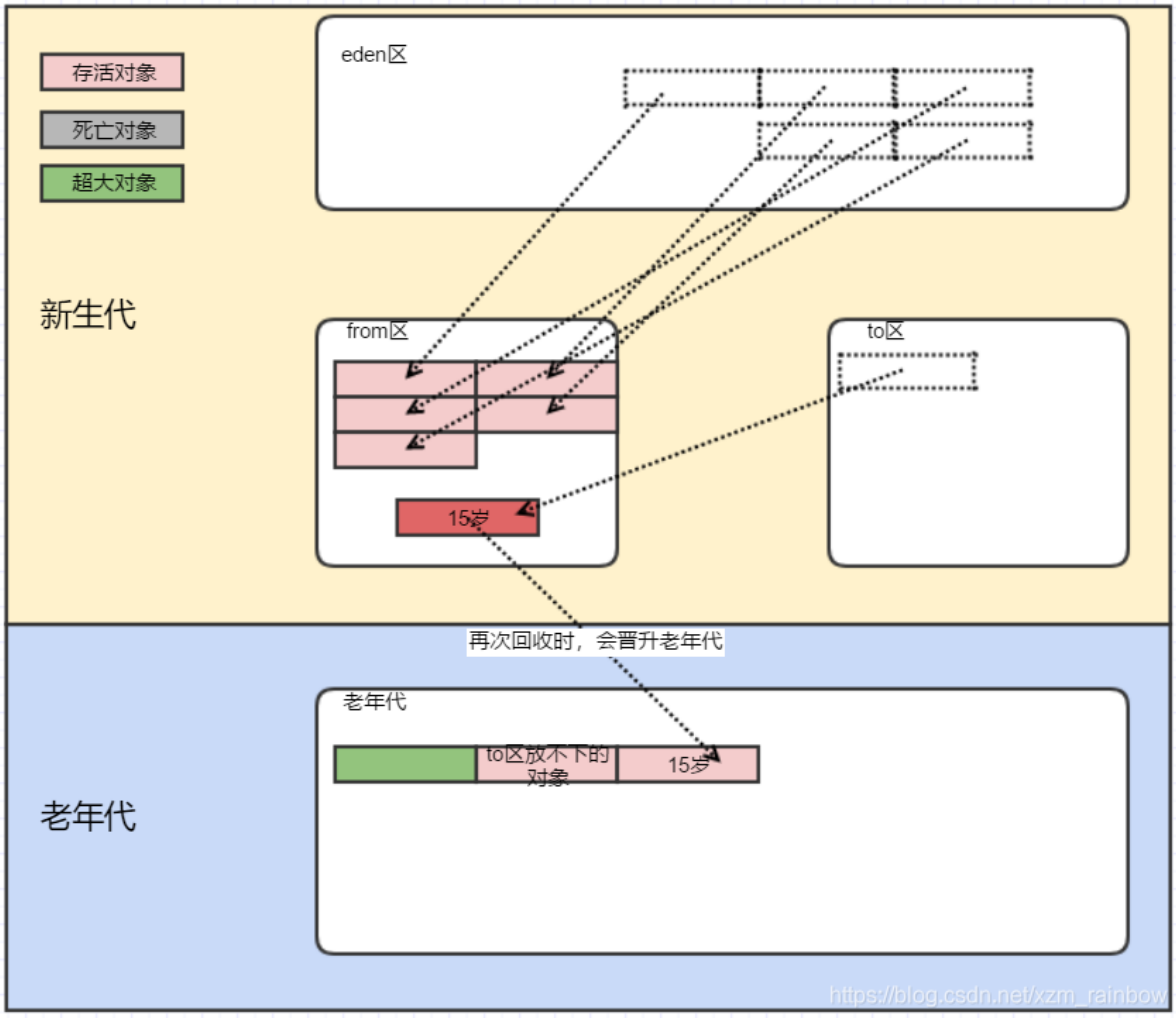
（3）程序继续运行，又在eden区产生了新的对象，并产生了一个超大对象，并产生了一个复制后to区放不下的对象



（4）当eden区再次被填满时，会再一次触发Minor GC，这次GC会将eden区和from区中存活的对象复制到to区，并且对象年龄+1，超大对象会直接晋升到老年代，to区放不下的对象也会直接晋升老年代。

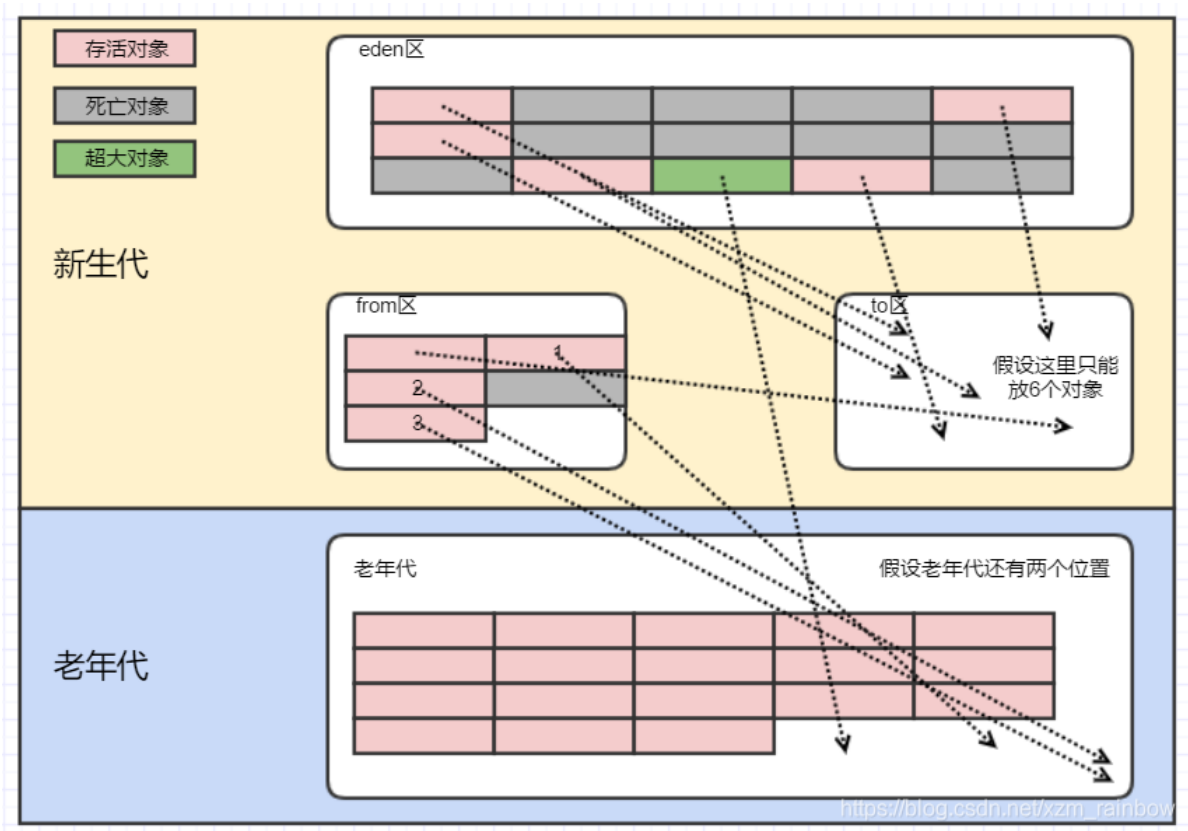


（5）程序继续运行，假设经过15次复制，某一对象依然存活，那么他将直接进入老年代。



（6）老年代 Full GC

在进行Minor GC之前，JVM还有一步操作，他会检查新生代所有对象使用的总内存是否小于老年代最大剩余连续内存，如果上述条件成立，那么这次Minor GC一定是安全的，因为即使所有新生代对象都进入老年代，老年代也不会内存溢出。如果上述条件不成立，JVM会查看参数-XX:HandlePromotionFailure是否开启（JDK1.6以后默认开启），如果没开启，说明Minor GC后可能会存在老年代内存溢出的风险，会进行一次Full GC，如果开启，JVM还会检查历次晋升老年代对象的平均大小是否小于老年代最大连续内存空间，如果成立，会尝试直接进行Minor GC，如果不成立，老年代执行Major GC。

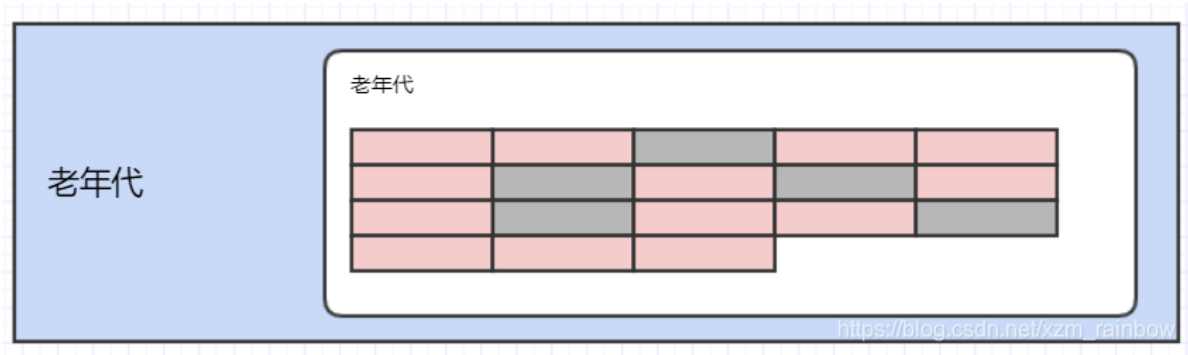


年轻代在每次minorGC之前都会计算老年代剩余可用空间，如果空间小于年轻代所有对象大小的和（包括垃圾），会检查-XX:-HandlePromotionFailure 参数（1.8默认开启），如果有，则会检查老年代剩余可用空间是否大于每次MinorGC后进入老年代的对象的平均大小，如果小于，则会触发FULL GC(stop the world 停止全部线程)，如果依然不够，则OOM

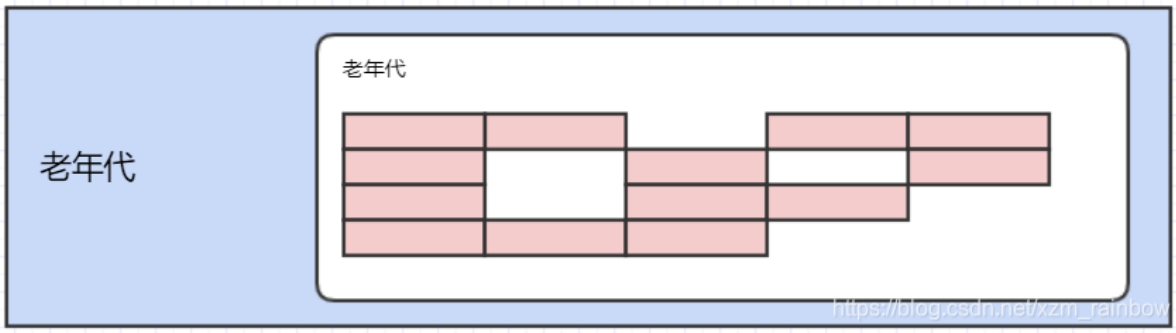
minorGC 是清理整合YouGen的过程， eden 的清理，S0\S1的清理都由于MinorGC
Allocation Failure(YoungGen区内存不足，不够调用) 会触发minorGC

Full GC 是清理整个堆空间—包括年轻代和永久代

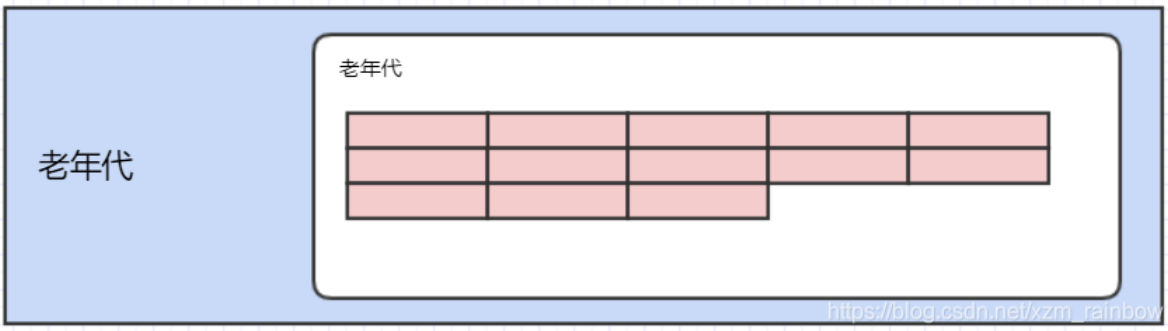
老年代回收-标记：



老年代回收-清除：

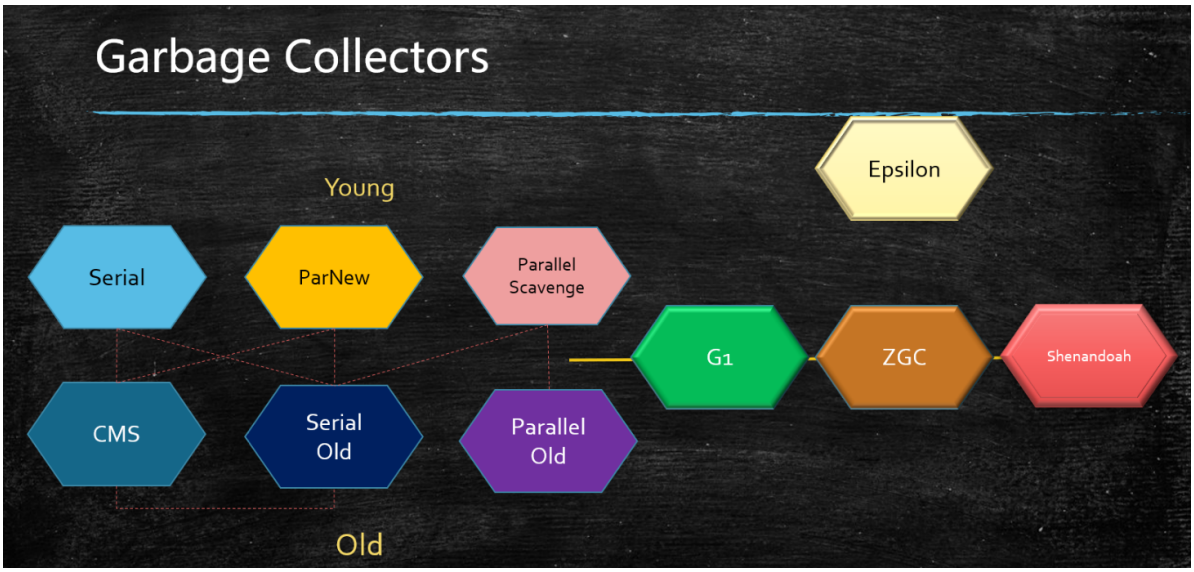


老年代回收-碎片整理：



4.垃圾收集器

根据堆中不同分代的特征，在JVM的历史长河中，诞生了各种各样的收集器，下面我们就以常见的几种做一些基本的认识



在很早以前，计算机内在只有几十M的时候，串行收集器基本上能满足使用，但随着硬件性能不断提高，内存大小和CPU运行速度的提升，在JVM发展的不同时期，诞生了针对当时计算机性能的垃圾回收器
在G1以前，物理和逻辑上都进行了分代，即将堆分为年轻代和老年代，直到G1的出现，这种分代概念就愈发模糊了，因为G1收集器针对的是整堆的收集。

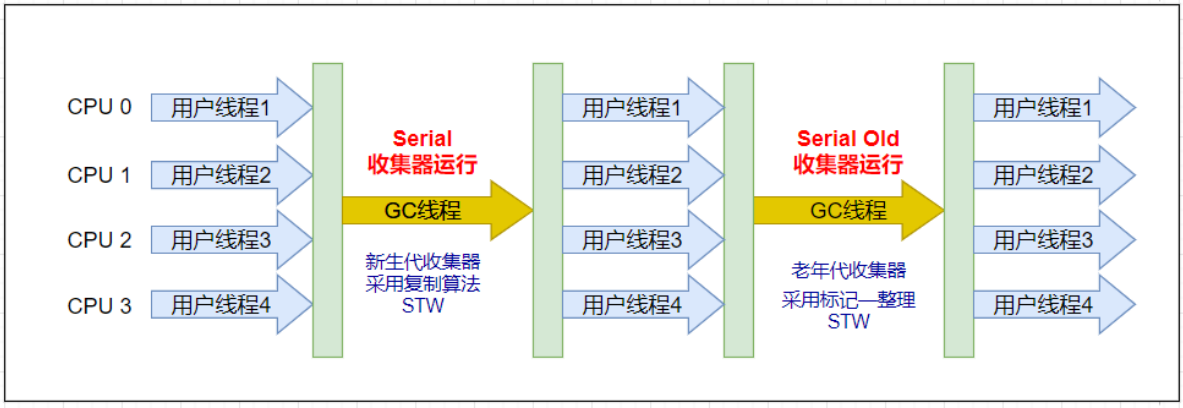
基本概念

STW：Stop The World，当垃圾回收线程工作时，需要暂停当前的用户（业务）线程，这个过程称它为STW;
串行收集：单线程收集器，简单高效，因为是单线程的原因，也就不会产生用户态和内核态切换所带来的开销；(会产生 STW)

并行收集：随着内存地不断增大，单CPU实现了多核的技术，通过多线程的方式收集垃圾可以极大地提升效率；(会产生 STW)
并发收集：指用户线程与垃圾收集线程同时工作（不一定是并行的可能会是交替执行）。用户程序在继续运行，而垃圾收集程序运行在另一个CPU上（多核CPU）(并发标记 和 并发清除 阶段不会产生STW)。缺点就是在垃圾回收的同时还会产生新的垃圾，有可能没有回收完，新的垃圾就把内存占满了，需要重新STW

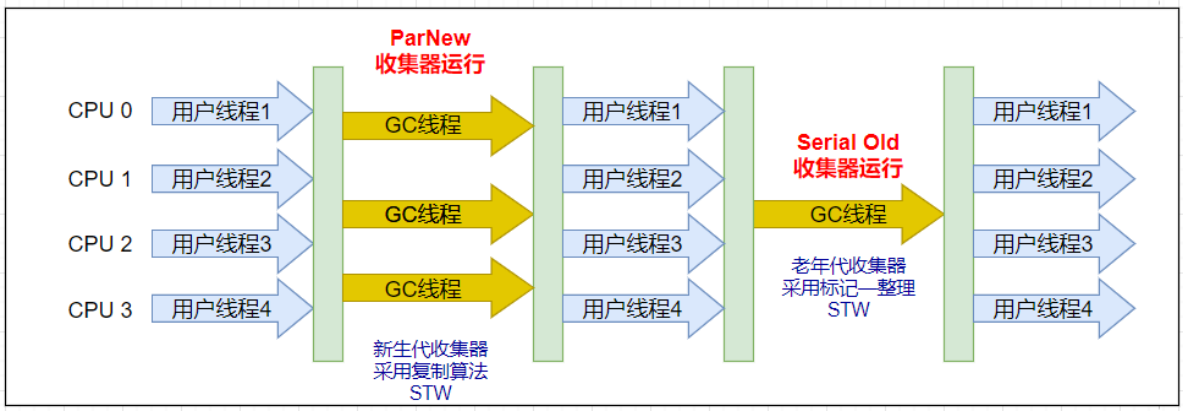
4.1 Serial 收集器

串行收集器，它是最早诞生的垃圾回收器，以单线程的方式进行垃圾收集，在JVM刚出来的情况下，计算机内存与现在相比特别地小，即便是串行回收，它的速度依然很快。
特点：单线程、简单高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。收集器进行垃圾回收时，必须暂停其他所有的工作线程，直到它结束（Stop The World）。
应用场景：小内存、单核CPU情况下的垃圾收集
Serial / Serial Old收集器运行示意图



4.2 ParNew 收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程外其余行为均和Serial收集器一模一样（参数控制、收集算法、Stop The World、对象分配规则、回收策略等）
特点：多线程、ParNew收集器默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境中，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。
和Serial收集器一样存在Stop The World问题
应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为它是除了Serial收集器外，唯一一个能与CMS收集器配合工作的。
ParNew 收集器运行示意图



4.3 Parallel Scavenge 收集器

用于大内存（大于2G）的垃圾收集。收集与吞吐量关系密切，故也称为吞吐量优先收集器。

吞吐量：运行用户代码时间/（运行用户代码时间+垃圾收集时间）

特点：属于新生代收集器也是采用复制算法的收集器，又是并行的多线程收集器（与ParNew收集器类似）。
该收集器的目标是达到一个可控制的吞吐量。还有一个值得关注的点是：GC自适应调节策略（与ParNew收集器最重要的一个区别）

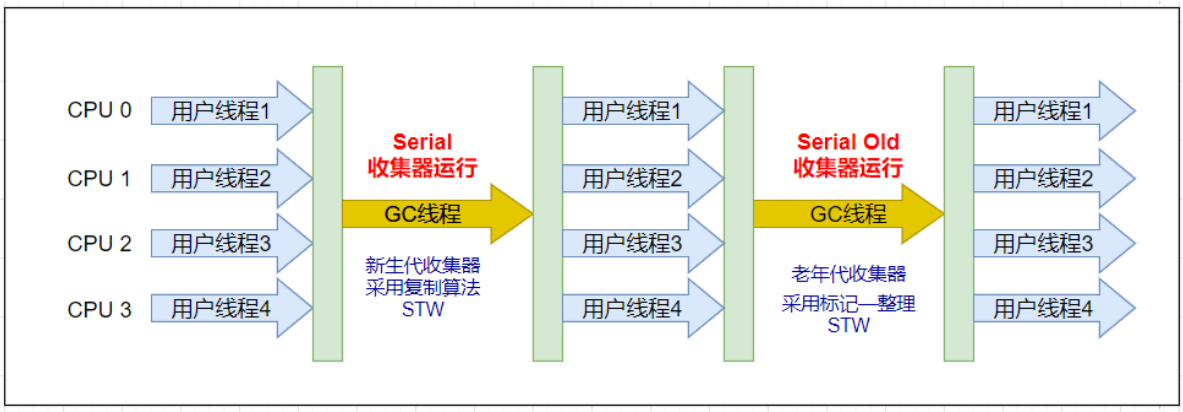
GC自适应调节策略：Parallel Scavenge收集器可设置-XX:+UseAdptiveSizePolicy参数。当开关打开时不需要手动指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThreshold）等，虚拟机会根据系统的运行状况收集性能监控信息，动态设置这些参数以提供最优的停顿时间和最高的吞吐量，这种调节方式称为GC的自适应调节策略。

Parallel Scavenge收集器使用两个参数控制吞吐量：
XX:MaxGCPauseMillis 控制最大的垃圾收集停顿时间
XX:GCRatio 直接设置吞吐量的大小。

4.4 Serial Old 收集器

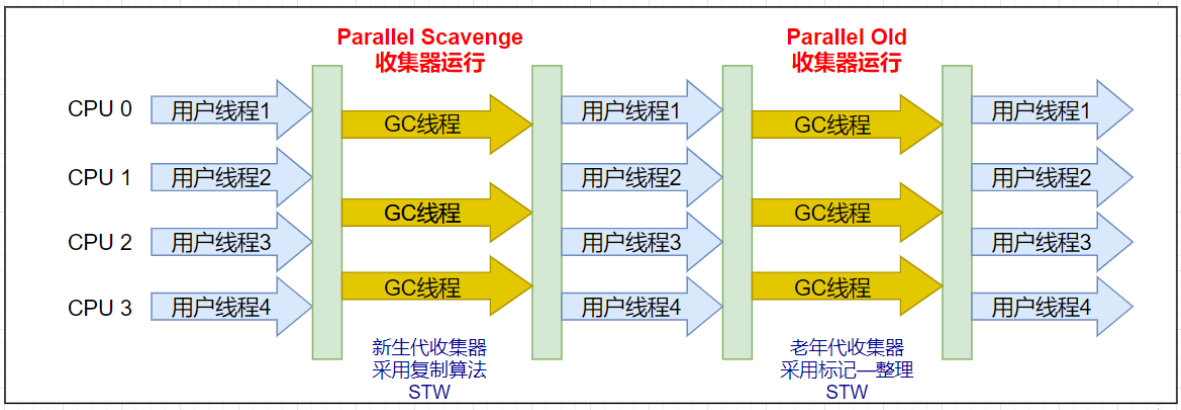
Serial Old是Serial收集器的老年代版本。
特点：同样是单线程收集器，采用**标记-整理**算法。
应用场景：主要也是使用在Client模式下的虚拟机中。也可在Server模式下使用。
它在Server模式下主要的两大用途：

- 在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用。
- 作为CMS收集器的后备方案，在并发收集Concurent Mode Failure时使用。
  Serial / Serial Old收集器工作过程图（Serial收集器图示相同）：



4.5 Parallel Old 收集器

它是Parallel Scavenge收集器的老年代版本。
特点：多线程，采用标记-整理算法。
应用场景：注重**高吞吐量**以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge+Parallel Old（PS + PO，JDK1.8默认） 收集器。
Parallel Scavenge/Parallel Old收集器工作过程图：



4.6 CMS（concurrent mark sweep） 收集器

一种以获取最短回收停顿时间为目标的收集器，**用于老年代的垃圾回收**
特点：基于标记—清除算法实现，与用户线程并发收集、并发清除，低停顿、低延时
应用场景：适用于注重服务的**响应速度**，希望系统**停顿时间最短**，给用户带来更好的体验等场景下。如web程序、b/s服务。
CMS收集器的运行过程可以大致分为以下四个阶段

初始标记
标记老年代中的所有GC Roots对象
标记年轻代中活着的对象引用到老年代的对象（指的是年轻带中还存活的引用类型对象，引用指向老年代中的对象）

并发标记
进行GC Roots Tracing 的过程，找出存活对象且与用户线程可并发执行。
从“初始标记”阶段标记的对象开始找出所有存活的对象
因为是并发执行，在用户线程运行的时候，会发生新生代对象晋升到老年代、或者是更新老年代对象的引用关系等等，对于这些新生成或改变的引用关系，可能会存在漏标，所有就必须要进行下一阶段的“重新标记”，为了提高下了阶段重新标记的效率，该阶段会把上述对象所在的Card标识为Dirty，下一阶段只需扫描这些Dirty Card的对象，避免扫描整个老年代；
并发标记阶段只负责将引用发生改变的Card标记为Dirty状态，不负责处理。
由于这个阶段是和用户线程并发执行的，可能会导致concurrent mode failure

重新标记
为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录。仍然存在Stop The World问题。
由于之前的预处理阶段是与用户线程并发执行的，这时候可能年轻带的对象对老年代的引用已经发生了很多改变，这个时候，remark阶段要花很多时间处理这些改变，会导致很长stop the word，所以通常CMS尽量运行Final Remark阶段在年轻代是足够干净的时候。
另外，还可以开启并行收集：-XX:+CMSParallelRemarkEnabled，提高reMark效率

并发清理
对标记的对象进行清除回收。
通过以上5个阶段的标记，老年代所有存活的对象已经被标记并且现在要通过Garbage Collector采用清扫的方式回收那些不能用的对象了。
这个阶段主要是清除那些没有标记的对象并且回收空间；

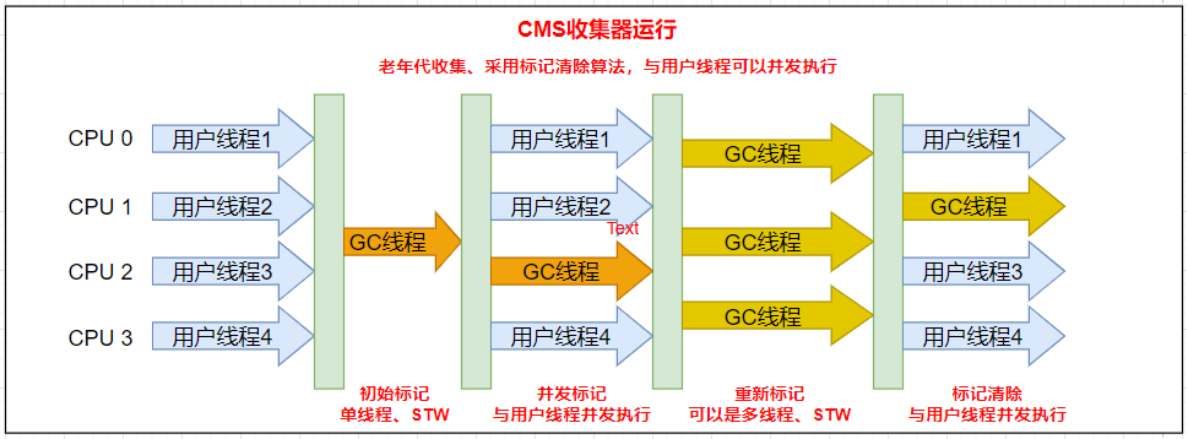
CMS的缺点

由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“**浮动垃圾**”。

由于使用了标记清除算法，所以会产生内存碎片，通过参数-XX:UseCMSCompactAtFullCollection可以让JVM做完标记-清除之后再进行整理

执行过程包含不确定性，会存在上一次垃圾回收没有执行完，然后垃圾回收又被触发的问题，特别是在并发标记 和并发清理阶段，一边回收系统一边运行，可能导致没有执行完成就再次出现FULL GC，产生"concurrent mode failure"，此时会进入SWT，用serial old垃圾收集器来进行回收

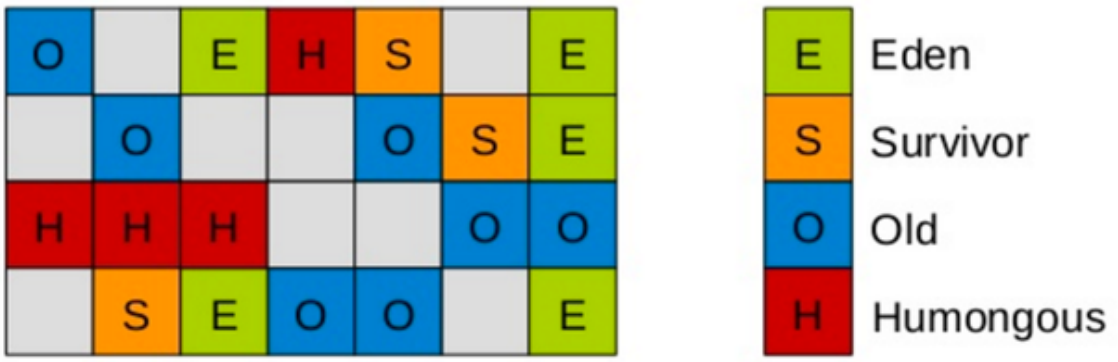
CMS收集器的工作流程图



4.7 G1 （Garbage first）收集器

1.概述

G1是一款面向服务端应用的垃圾收集器
它是在CMS的基础上改进而来，现已被JDK1.9作为默认的垃圾回收器



G1开创的基于Region的堆内存布局。虽然G1也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1不再坚持固定大小以及固定数量的分代区域划分，而是把连续的Java堆划分为多个大小相等的独立区域（Region），每一个Region都可以根据需要，扮演新生代的Eden空间、Survivor空间，或者老年代空间。

收集器能够对扮演不同角色的Region采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。

G1中五种不同类型的Region：

1. Eden regions(年轻代-Eden区)
2. Survivor regions(年轻代-Survivor区)
3. Old regions（老年代）
4. Humongous regions（巨型对象区域，通常也被认为是老年代的一部分，G1认为只要大小超过了一个Region容量一半的对象即可判定为大对象。）
5. Free resgions（未分配区域，也会叫做可用分区）- 上图中空白的区域

每个Region的大小可以通过参数-XX:G1HeapRegionSize设定，取值范围为1MB~32MB，且应为2的N次幂。

而对于那些超过了整个Region容量的超级大对象，将会被存放在N个连续的Humongous Region之中，G1的大多数行为都把Humongous Region作为老年代的一部分来进行看待。

2.特点

并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让Java程序继续运行。
分代收集：G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过15次GC的旧对象以获取更好的收集效果。
空间整合：G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。
***可预测的停顿**：G1除了追求低停顿外，还能建立可预测的停顿时间模型。能让使用者明确**指定**在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。
G1为什么能建立可预测的停顿时间模型?
因为它有计划的避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。这样就保证了在有限的时间内可以获取尽可能高的收集效率。（可以通过-XX:MaxGCPauseMills指定时间，默认200ms）

G1适用场景

- 面向服务端应用，针对具有大内存，多核心处理器的机器，主要对于需要低GC延迟，并且堆具有较大的内存空间的程序提供较好的支持；堆空间大小在6g-8G以上性能优异（G1通过每次只处理部分region的垃圾回收任务，而不是全堆进行垃圾回收，这种增量的垃圾处理方式保证每次GC不会造成长时间停顿）
- 在【超过50%的Java堆被活跃数据占用】、【对象分配频率或者年代提升频率较高】、【GC停顿时间长（0.5-1秒）】的情况下可以考虑G1替换CMS
- Hotspot的垃圾收集器中，除G1之外，其他的垃圾收集器都是使用内置的JVM线程执行GC的多线程操作，而G1可以采用应用线程承担后台运行的GC工作，即当JVM的GC线程处理速度不理想的时候，系统会调用应用程序线程帮助加速垃圾回收工作

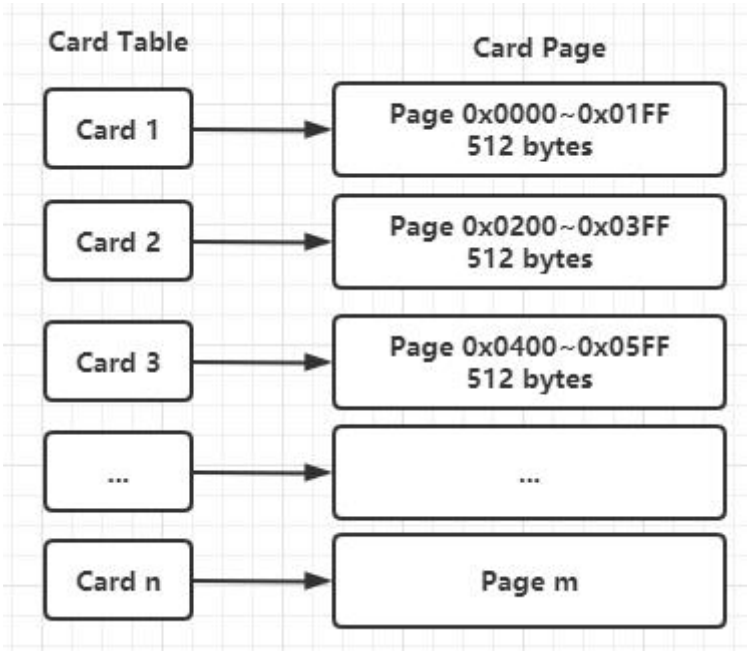
G1与其他收集器的区别：
其他收集器的工作范围是整个新生代或者老年代、G1收集器的工作范围是整个Java堆。在使用G1收集器时，它将整个Java堆划分为多个大小相等的独立区域（Region）。虽然也保留了新生代、老年代的概念，但新生代和老年代不再是相互隔离的，他们都是一部分Region（不需要连续）的集合。尤其大内存的堆空间效果要高于其他。

G1收集器存在的问题：
Region不可能是孤立的，分配在Region中的对象可以与Java堆中的任意对象发生引用关系。在采用可达性分析算法来判断对象是否存活时，得扫描整个Java堆才能保证准确性。其他收集器也存在这种问题（G1更加突出而已）。会导致Minor GC效率下降。

G1收集器是如何解决上述问题的?
采用Remembered Set来避免整堆扫描。G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在对Reference类型进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用对象是否处于多个Region中（即检查老年代中是否引用了新生代中的对象），如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set中。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆进行扫描也不会有遗漏。

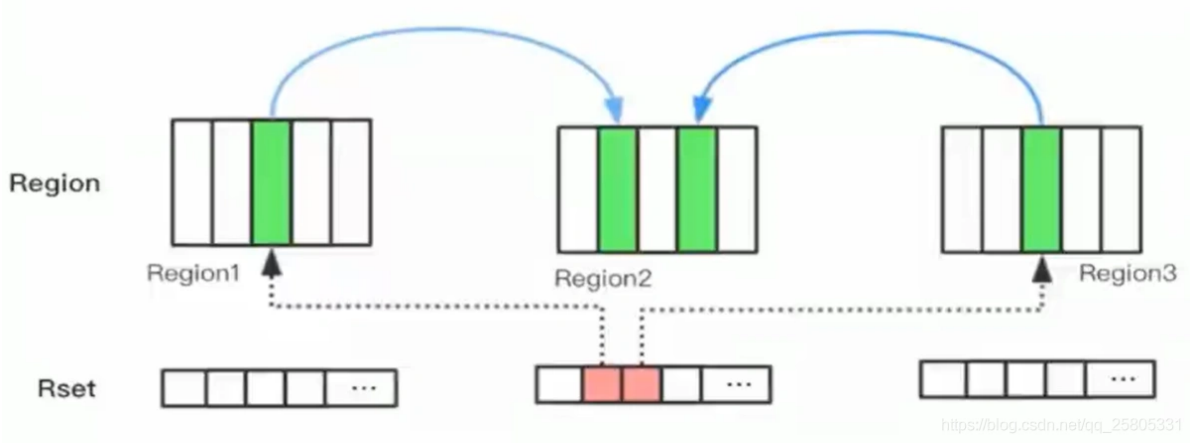
记忆集（Remenbered Set,简称RSet）和卡表（Card Table）

卡表最简单的形式可以只是一个字节数组，数组中的每一个元素都对应着其标识的内存区域中一块特定大小的内存块，整个内存块被称为“卡页”（Card Page）。HotSpot中卡页大小为512字节。如果卡表标识内存区域的起始地址是）0x0000的话，数组的第0、1、2号元素，分别对应了地址范围0x0000 ~ 0x01FF、0x0200 ~ 0x03FF、0x0400 ~ 0x05FF的卡页内存块，如下图所示。



一个卡页的内存中通常包含不止一个对象，只要卡页内有一个（或更多）对象的字段存在着跨代指针，那就对应卡表的数组元素的值标识为1，称为这个元素变脏，没有则标识为0，在垃圾收集发生时，只要筛选出卡表中变脏的元素，就能轻易得出哪些卡页内存块中包含跨代指针，把它们加入GC Roots中一并扫描。

RSet全称是Remembered Set，每个Region中都有一个RSet，记录的是其他Region中的对象引用本Region对象的关系(谁引用了我的对象)。G1里面还有另外一种数据结构就Collection Set(CSet)，CSet记录的是GC要收集的Region的集合，CSet里的Region可以是任意代的。在GC的时候，对于old->young和old->old的跨代对象引用，只要扫描对应的CSet中的RSet即可。



因此，RSet存在的意义就是避免对象跨代引用时对整个堆内存对象的扫描，起到一种类似索引（更像空间索引）的作用。

原始快照（Snapshot At The Beginning，SATB）

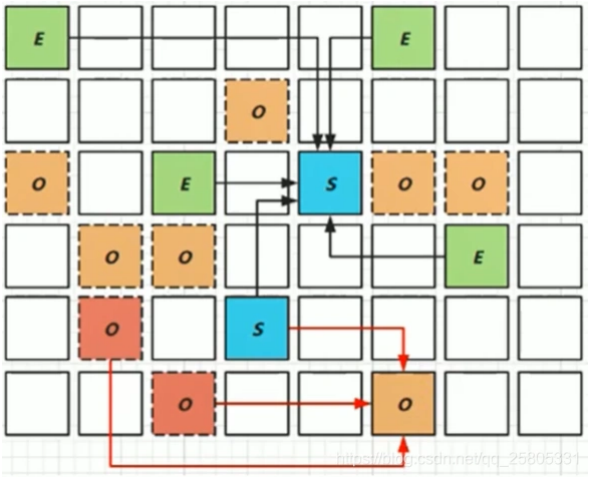
简洁地来说，SATB是维持并发GC的一种手段，因为像CMS、G1等收集器，首先经过初始标记，然后进行并发标记，并发标记过程中，可能对初始标记的结果产生了改动，需要进行修正，分别有增量更新和原始快照两种解决方法。CMS收集器使用增量更新来纠正对象引用关系，而G1收集器使用原始快照的策略。

3.G1垃圾回收的过程

- 新生代垃圾回收
    - **并行的、独占式的（STW发生）的垃圾回收**，可能会发生对象的代晋升，将会把对象放入Survivor或者是老年代。
    - 当Eden区内存空间耗尽的时候，G1会启动一次年轻代垃圾回收，顺带回收Survivor区。
        - G1 Young GC的时候，首先停止程序线程（STW），G1创建回收集（Collection set，需要被回收的内存分段集合，包括Eden区和Survivor区的所有内存分段）
        - 第一阶段：扫描根（GC Roots），GC Roots联通RSet记录的外部引用作为扫描存活对象的入口。
        - 第二阶段：处理**dirty card queue**中的card，更新RSet。此阶段完成后，**RSet可以准确的反应老年代对所在的内存分段中对象的引用。**
        - 第三阶段：识别被老年代对象指向的Eden区中的对象，这些被指向的Eden中对象被认为是存活的对象。
        - 第四阶段：复制对象（**使用的是复制算法**），对象数被遍历，Eden区region中存活的对象会被复制到Survivor区中的region，Survivor region中的存活对象如果年龄未达到阈值，年龄+1，达到阈值会被复制到Old region，如果Survivior的空间不够用，Eden中的部分数据会直接晋升到老年代。
        - 第五阶段：处理Soft、Weak、Phantom、Final、JNI Weak等引用，最终Eden区的数据为空，这些空的region将会等待对象的分配，GC停止工作，目标内存中的对象也都是连续的，没有内存碎片。
- 新生垃圾回收和老年代并发标记
    - 当堆空间的内存占用达到阈值（-XX:InitiatingHeapOccupancyPercent，默认45%）就开始老年代的并发标记过程
        - 初始标记阶段：**标记GC Roots直接可达的对象，也就是直接引用关系对象**，会发生STW（由于是直接可达的对象的标记，所以暂停时间很短），**并且会触发一次Young GC**
        - 根区域的扫描（Root Region Scanning）：G1扫描Survivor区直接可达的老年代区域对象，并标记被引用的对象。这一个过程**必须在Young GC之前完成（因为Young GC会操作Survivor区中的对象）。**
        - 并发标记（Concurrent Marking）：在整个堆中进行并发标记（与程序线程并发执行），**此过程可能会被Young GC打断**，在并发标记阶段中，**若发现某些region中所有对象都是垃圾，那这个region就会被立即回收**，同时并发标记过程中，会**计算每个region的对象活性（该region存活对象的比例，G1垃圾回收的时候并不是所有region都会参与回收的，根据回收的价值高低来优先回收价值较高的region）。**
        - 再次标记（remark）：由于并发标记阶段是收集器的标记线程和程序线程并发执行的，需要进行再次标记，修正上一次的标记结果，可以理解为增量补偿标记。会出现STW（暂停时间较短）。G1采用的是比CMS更快的初始快照算法：snapshot-at-the-beginning（SATB）
        - 独占清理：计算各个region的存活对象和GC回收比例，并进行排序（回收价值高低排序），识别可以混合回收的区域。为下阶段做铺垫，会发生STW。**需要注意的是这个阶段实际上并不会做垃圾的回**收。
        - 并发清理阶段：识别并清理完成空闲的区域

- 混合回收
  - 包括年轻代和老年代的垃圾回收
  - 标记完成后马上开始垃圾的回收。对于一个混合的回收过程，G1从老年代移动存活的对象到空闲区域，这些空闲的区域变成了老年代region。当越来越多的对象晋升到老年代region的时候，为了避免堆内存被耗尽，就会触发以混合垃圾收集Mixed GC，该算法并不是一个Old GC也不是Full GC，除了回收整个Young region之外，还会回收一部分Old region，

    部分的region垃圾回收设计可以对垃圾回收的耗时进行控制

    - 在并发标记结束之后，老年代中能够完全确认为垃圾的region中的内存分段被回收了，部分为垃圾的region中内存分段也被计算出来了，默认情况下，这些老年代的内存分段会被分为8次回收（**可以通过-XX:G1MixedGCCountTarget设置**）。
    - 混合回收的回收集包括1/8的老年代的内存分段，Eden区内存分段，Survivor内存分段，

      混合回收的算法和年轻代回收的算法完全一致

      - 混合回收并不一定要进行8次，有一个**阈值设置：-XX:G1HeapWastePercent，默认值10%，代表允许整个堆内存中有10%的内存可以被浪费，意味着如果发现可以回收的垃圾占对内存的比例低于10%，则不进行混合回收**，因为GC花费的时间相对于较少的垃圾回收来说得不偿失。
    - **由于老年代的内存分段默认分为8次回收，G1会优先回收垃圾多的内存分段，垃圾占内存分段比例越高的会优先被回收。**并且有**一个阈值决定内存分段是否被回收：-XX:G1MixedGCLiveThresholdPercent，默认为65%，代表垃圾占内存分段比例要达到65%来回被回收**，如果垃圾占比太低，意味着存活的对象多，复制算法就会花费更多的时间区复制存活的对象。
- 必要的情况下（对象分配速度远大于回收速度），Full GC仍然会触发（Full GC的成本较高，单线程，性能差，STW时间长）
  - 堆内存太小、对象分配速度远大于回收速度等原因都可以导致G1在复制存活对象的时候没有空闲的内存分段可用，最终造成Full GC的触发。

G1优化建议

- 年轻代的大小
  - 避免使用-Xmn或者-XX:NewRatio等相关参数显式的指定年轻代大小
  - 定死年轻代的大小会导致G1无法自行的动态调优，导致暂停的目标时间（-XX:MaxGCPauseMillis）无法达到。
- 暂停时间设置不要太短
  - G1的吞吐量目标是90%的应用程序时间和10%的垃圾回收时间

- 过于短暂的暂停时间设置，会直接影响到垃圾回收发生的频率，最终转嫁到吞吐量上，造成吞吐量下降

11 堆内存调优

JVM调优

JVM调优一般针对的是吞吐量和暂停时间

- 较高的吞吐量在较长时间段内，会让用户感觉上只有程序线程在执行，就认为程序运行是比较块的。
- 对于交互性较高的应用场景来说，越低的暂停时间对于用户来说是越感觉不到的。

高吞吐量和低暂停时间是一对相矛盾的存在

- 如果调优以吞吐量为主，那么必然会降低内存回收的频率，就会造成每次GC造成的暂停时间变长
- 如果调优以低暂停时间为主，那么每次GC回收的垃圾必然会减少，只能通过频繁的执行GC。

所以目前对于JVM的优化基本是按照应用程序的使用场景来定的，目前G1垃圾收集器的原则是在最大吞吐量优先的情况下，降低停顿时间

G1--JVM heap大小配置建议

2C4G linux64 jdk8

```
-Xmx2688M -Xms2688M -Xmn960M -XX:MaxMetaspaceSize=512M -XX:MetaspaceSize=512M -
XX:+UseG1GC
```

4C8G linux64 jdk8

```
-Xmx5440M -Xms5440M -XX:MaxMetaspaceSize=512M -XX:MetaspaceSize=512M -
XX:+UseG1GC
```

4C16G linux64 jdk8

```
-Xmx10880M -Xms10880M -XX:MaxMetaspaceSize=512M -XX:MetaspaceSize=512M -
XX:+UseG1GC
```

https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html

1、堆设置

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小

2、收集器设置

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

3、垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

4、并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

5、并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。