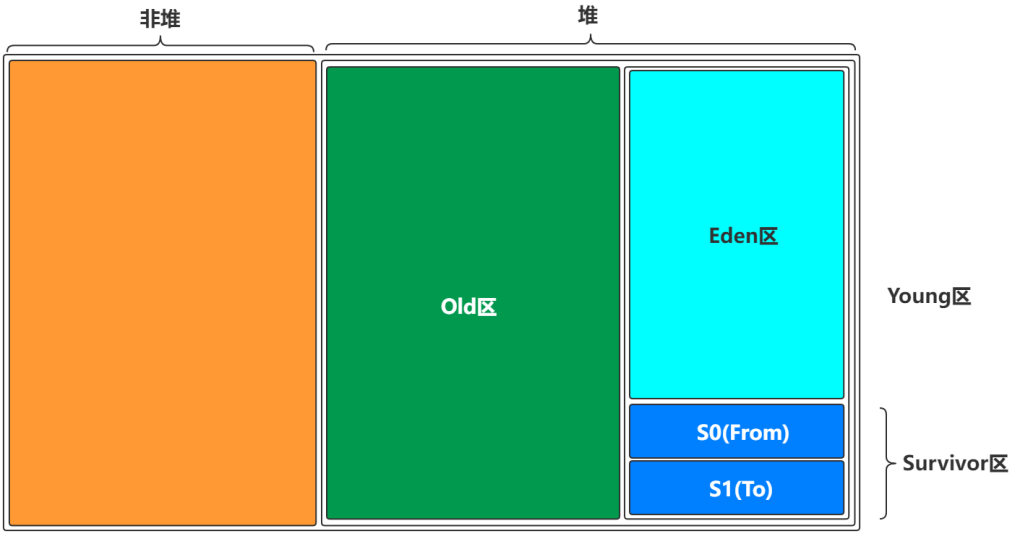
01 JVM内存模型

1.1 整体描述

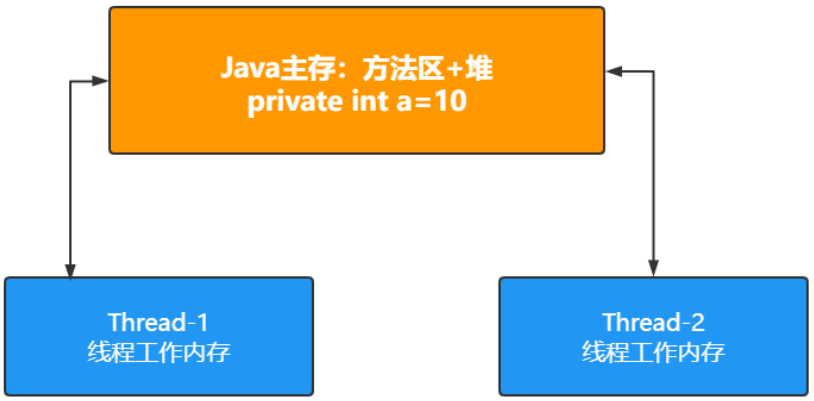
上面对运行时数据区描述了很多，其实重点存储数据的是堆和方法区(非堆)，所以内存的设计也着重从这两方面展开(注意这两块区域都是线程共享的)。对于虚拟机栈，本地方法栈，程序计数器都是线程私有的。

```
（1）一块是非堆区，一块是堆区
（2）堆区分为两大块：一个是Old区，一个是Young区
（3）Young区分为两大块：一个是Survivor区（S0+S1），一块是Eden区
（4）S0和S1一样大，也可以叫From和To
```

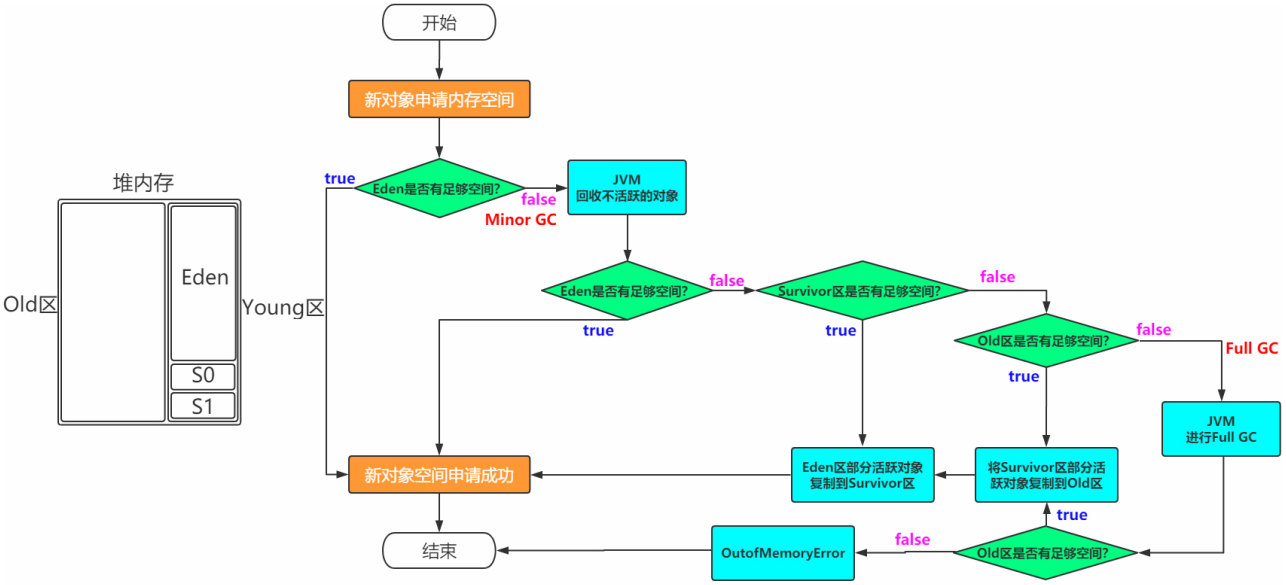


1.2 Java Memory Model

官网：https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4



1.3 对象在内存中的分配回收



1.4 代码体验

1.4.1 堆的OOM

> -Xmx50M -Xms50M

```
@RestController
public class HeapController {
 List<Person> list=new ArrayList<User>();
 @GetMapping("/heap")
 public String heap(){
 while(true){
 list.add(new User());
 }
 }
}
```

1.4.2 方法区OOM

> -XX:MetaspaceSize=50M -XX:MaxMetaspaceSize=50M

（1）asm依赖

```
<dependency>
 <groupId>asm</groupId>
 <artifactId>asm</artifactId>
 <version>3.3.1</version>
</dependency>
```

（2）工具类

```
public class MyMetaspace extends ClassLoader {
 public static List<Class<?>> createClasses() {
 List<Class<?>> classes = new ArrayList<Class<?>>();
 for (int i = 0; i < 10000000; ++i) {
 ClassWriter cw = new ClassWriter(0);
 cw.visit(Opcodes.V1_1, Opcodes.ACC_PUBLIC, "Class" + i, null,
 "java/lang/Object", null);
 MethodVisitor mw = cw.visitMethod(Opcodes.ACC_PUBLIC, "<init>",
 "()V", null, null);
 mw.visitVarInsn(Opcodes.ALOAD, 0);
 mw.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object",
 "<init>", "()V");
 mw.visitInsn(Opcodes.RETURN);
 mw.visitMaxs(1, 1);
 mw.visitEnd();
 Metaspace test = new Metaspace();
 byte[] code = cw.toByteArray();
 Class<?> exampleClass = test.defineClass("Class" + i, code, 0,
```

```
19  code.length);
20  classes.add(exampleClass);
21  }
22  return classes;
23  }
24 }
```

（3）代码

```
1  @RestController
2  public class NonHeapController {
3   List<Class<?>> list=new ArrayList<Class<?>>();
4   @GetMapping("/nonheap")
5   public String nonheap(){
6   while(true){
7   list.addAll(MyMetaspace.createClasses());
8   }
9   }
10 }
```

1.4.3 栈溢出

（1）代码

```
1  public class StackDemo {
2   public static long count=0;
3   public static void method(){
4   System.out.println(count++);
5   method();
6   }
7   public static void main(String[] args) {
8   method();
9   }
10 }
```

（2）运行结果

```
7252
7253
7254
7255
Exception in thread "main" java.lang.StackOverflowError
    at sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:691)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
```

02 垃圾收集

2.1 确定垃圾对象

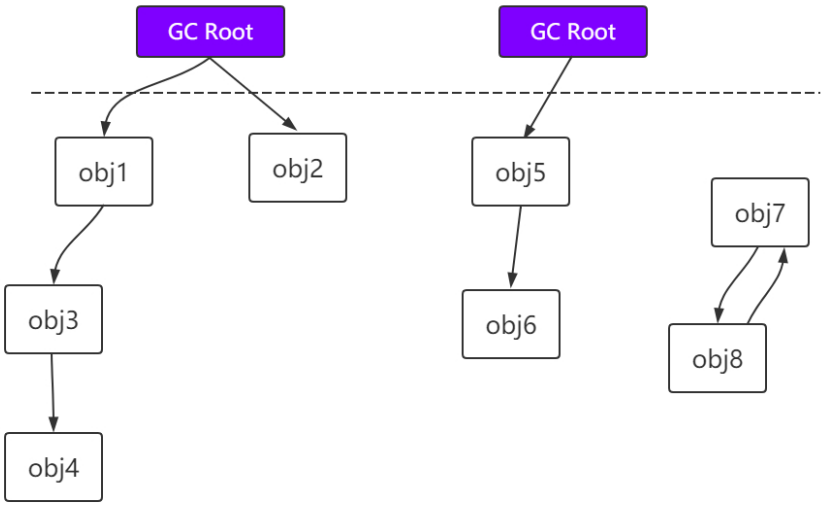
（1）引用计数法

对于某个对象而言，只要应用程序中持有该对象的引用，就说明该对象不是垃圾，如果一个对象没有任何指针对其引用，它就是垃圾。

（2）可达性分析

通过GC Root的对象，开始向下寻找，看某个对象是否可达。

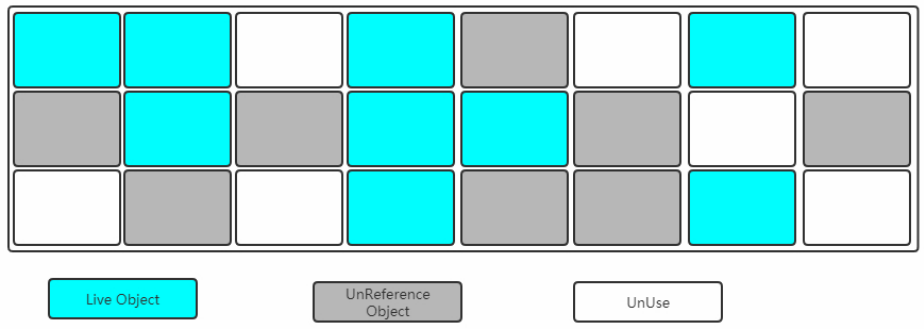
> 能作为GC Root: 类加载器、Thread、虚拟机栈的本地变量表、static成员、常量引用、本地方法栈的变量等。



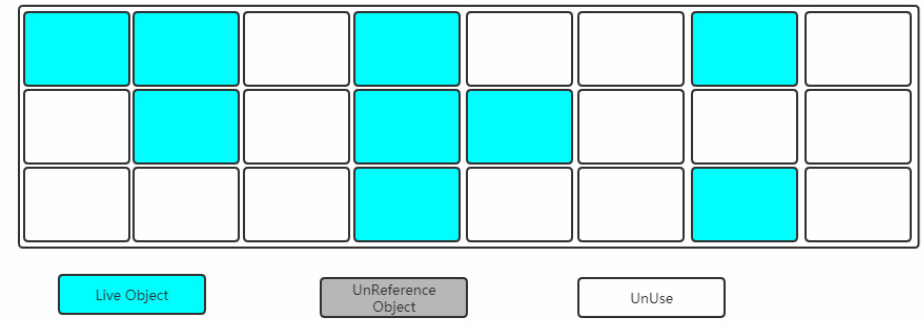
2.2 垃圾收集算法

2.2.1 标记-清除

- 标记：找出内存中需要回收的对象，并且把它们标记出来



- 清除：清除掉被标记需要回收的对象，释放出对应的内存空间



缺点

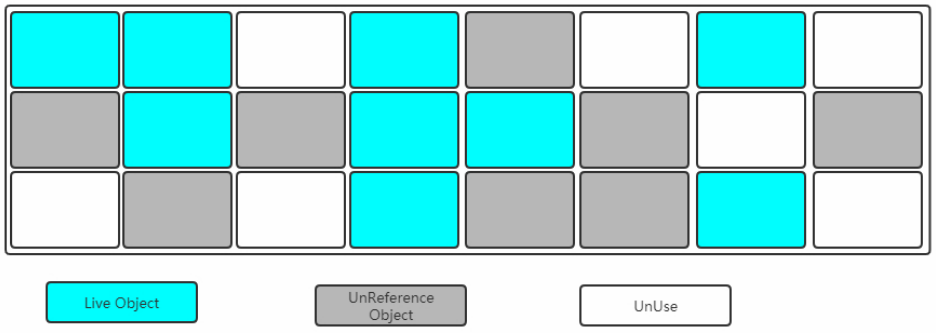
1 标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

2 （1)标记和清除两个过程都比较耗时，效率不高

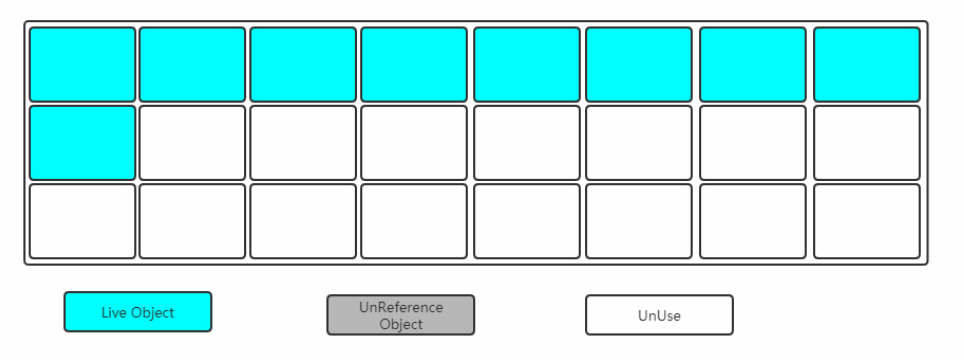
3 （2)会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作

2.2.2 标记-整理

- 标记

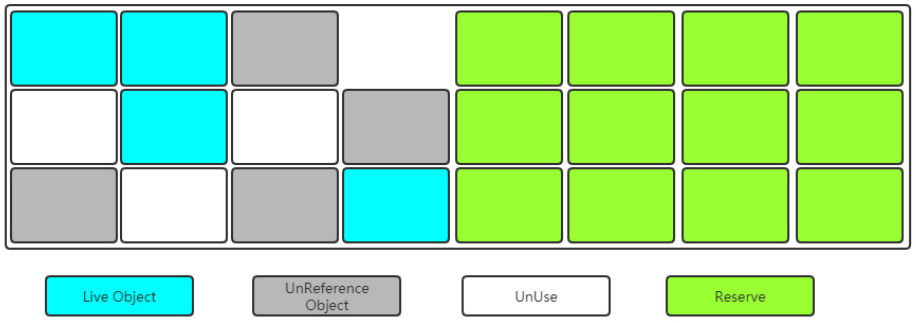


- 整理

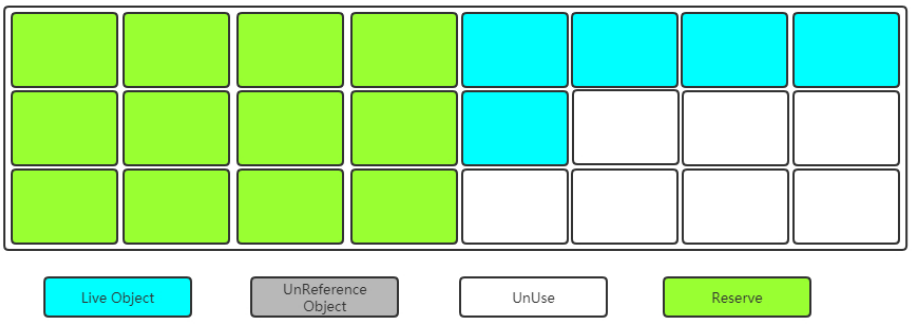


2.2.3 标记-复制

- 标记



- 复制



> 缺点

```
(1)存在大量的复制操作，效率会降低
(2)空间利用率降低
```

2.3 垃圾收集算法选择

Young区：复制算法(对象在被分配之后，可能生命周期比较短，Young区复制效率比较高)

Old区：标记清除或标记整理(Old区对象存活时间比较长，复制来复制去没必要，不如做个标记再清理)

2.4 垃圾收集器

> 如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

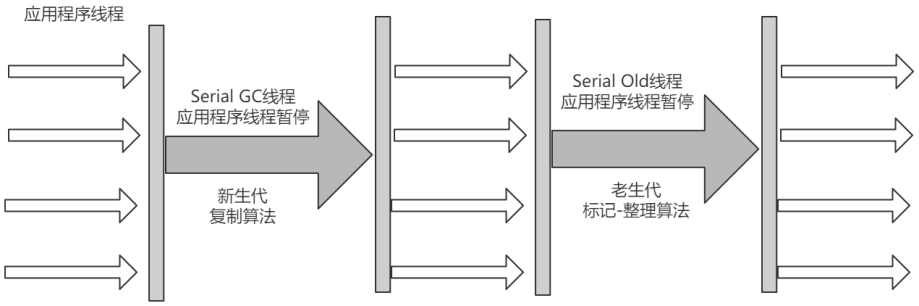
Java8 官网：
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref27

Java17 官网：https://docs.oracle.com/en/java/javase/17/gctuning/available-collectors.html#GUID-F215A508-9E58-40B4-90A5-74E29BF3BD3C

2.4.1 Serial

> 可以用于新老年代
> 新生代: 复制算法
> 老年代: 标记-整理算法

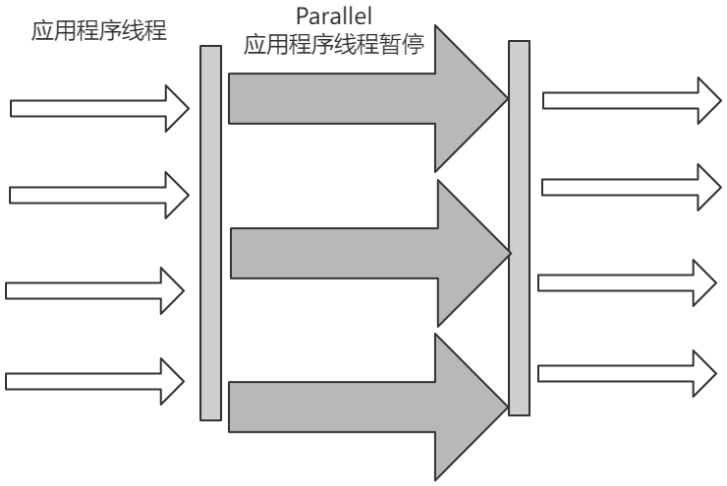
The serial collector uses a single thread to perform all garbage collection work



2.4.2 Parallel

> 可以用于新老年代
> 新生代：复制算法
> 老年代：标记整理算法

The parallel collector is also known as throughput collector, it's a generational collector similar to the serial collector. The primary difference between the serial and parallel collectors is that the parallel collector has
multiple threads that are used to speed up garbage collection.



2.4.3 CMS(ConcMarkSweepGC)

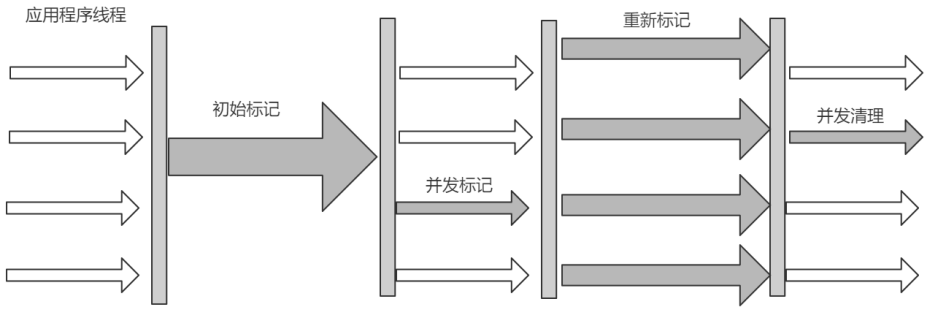
官网：
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#concurrent_mark_

> 可以用于老年代
> 采用标记-清除算法
> 回收过程：https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

The Concurrent Mark Sweep (CMS) collector is designed for applications that prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage

collector while the application is
running.

```
（1）初始标记 CMS initial mark 标记GC Roots直接关联对象，不用Tracing，速度很快
（2）并发标记 CMS concurrent mark 进行GC Roots Tracing
（3）重新标记 CMS remark 修改并发标记因用户程序变动的内容
（4）并发清除 CMS concurrent sweep 清除不可达对象回收空间，同时有新垃圾产生，留着下次清理称为浮动垃圾
```



> 优缺点

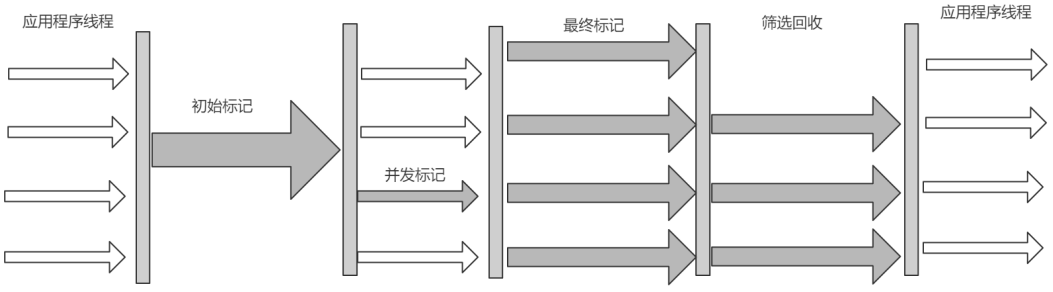
```
优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量
```

2.4.4 G1(Garbage First)

> 可以用于新老年代
> 整体上采用标记-整理算法
> 回收过程：https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

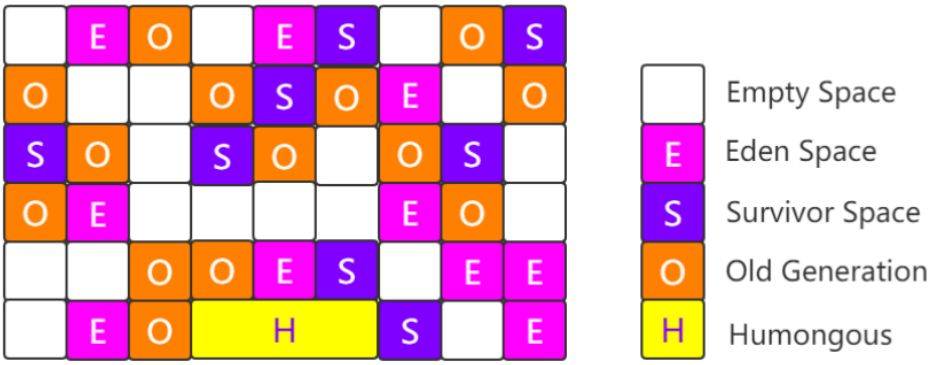
G1 is a mostly concurrent collector. Mostly concurrent collectors perform some expensive work concurrently to the application. This collector is designed to scale from small machines to large multiprocessor machines
with a large amount of memory. It provides the capability to meet a pause-time goal with high probability, while achieving high throughput.

```
（1）初始标记（Initial Marking） 标记以下GC Roots能够关联的对象，并且修改TAMS的值，需要暂停用户线程
（2）并发标记（Concurrent Marking） 从GC Roots进行可达性分析，找出存活的对象，与用户线程并发执行
（3）最终标记（Final Marking） 修正在并发标记阶段因为用户程序的并发执行导致变动的数据，需暂停用户线程
（4）筛选回收（Live Data Counting and Evacuation） 对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间制定回收计划
```



> 理解

```
使用G1收集器时，Java堆的内存布局与就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合
每个Region大小都是一样的，可以是1M到32M之间的数值，但是必须保证是2的n次幂
如果对象太大，一个Region放不下[超过Region大小的50%]，那么就会直接放到H中
设置Region大小：-XX:G1HeapRegionSize=M
所谓Garbage-Frist，其实就是优先回收垃圾最多的Region区域
```



2.4.5 ZGC

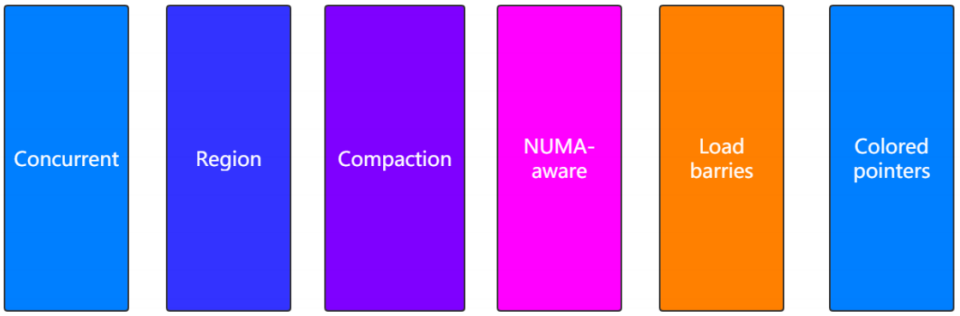
> Java11引入的垃圾收集器
> 不管是物理上还是逻辑上，ZGC中已经不存在新老年代的概念了
> 会分为一个个page，当进行GC操作时会对page进行压缩，因此没有碎片问题
> 只能在64位的linux上使用，目前用得还比较少

The Z Garbage Collector (ZGC) is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads.

（1）可以达到10ms以内的停顿时间要求

（2）支持TB级别的内存

（3）堆内存变大后停顿时间还是在10ms以内

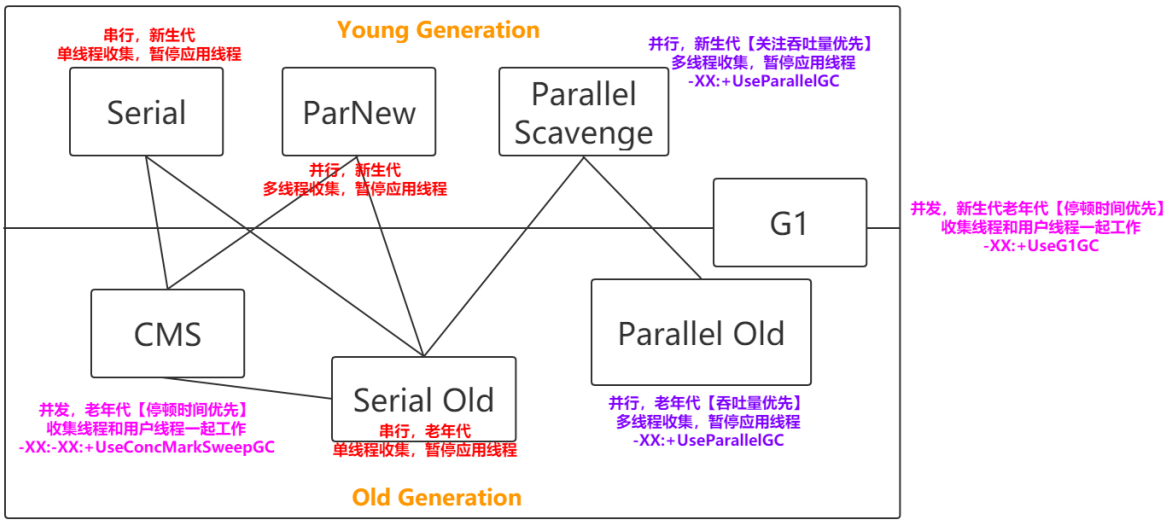


2.5 垃圾收集器分类

（1）串行：Serial　　适合内存比较小的嵌入式设备

（2）并行：Parallel　更加关注吞吐量：适合科学计算、后台处理等若交互场景

（3）并发：CMS、G1　　更加关注停顿时间：适合web交互场景



2.6 Reponsiveness and throughput

停顿时间->垃圾收集器 进行 垃圾回收终端应用执行响应的时间

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)

Responsiveness

Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:

- How quickly a desktop UI responds to an event
- How fast a website returns a page
- How fast a database query is returned

For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

Throughput

Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:

- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.

2.7 什么时候会发生垃圾收集

> GC是由JVM自动完成的，根据JVM系统环境而定，所以时机是不确定的。
> 当然，我们可以手动进行垃圾回收，比如调用System.gc()方法通知JVM进行一次垃圾回收，但是具体什么时刻运行也无法控制。
> 也就是说System.gc()只是通知要回收，什么时候回收由JVM决定。但是不建议手动调用该方法，因为GC消耗的资源比较大。

（1）当Eden区或者S区不够用了： Young GC 或 Minor GC

（2）老年代空间不够用了： Old GC 或 Major GC

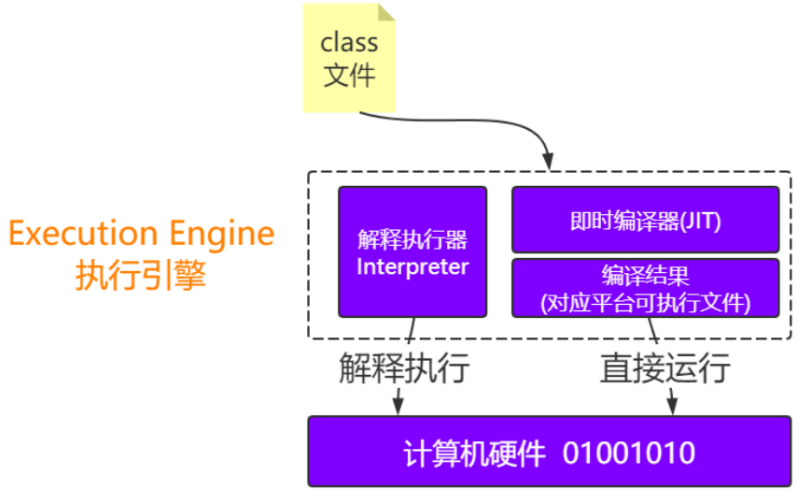
（3）方法区空间不够用了： Metaspace GC

（4）System.gc()

Full GC=Young GC+Old GC+Metaspace GC

03 Execution Engine

> User.java源码文件是Java这门高级开发语言，对程序员友好，方便我们开发。
> javac编译器将User.java源码文件编译成class文件[我们把这里的编译称为前期编译]，交给JVM运行，因为JVM只能认识class字节码文件。同时在不同的操作系统上安装对应版本的JDK，里面包含了各自屏蔽操作系统底层细节的JVM，这样同一份class文件就能运行在不同的操作系统平台之上，得益于JVM。这也是Write Once，Run Anywhere的原因所在。
> 最终JVM需要把字节码指令转换为机器码，可以理解为是0101这样的机器语言，这样才能运行在不同的机器上，那么由字节码转变为机器码是谁来做的呢？说白了就是谁来执行这些字节码指令的呢？这就是执行引擎里面的解释执行器和编译器所要做的事情。



3.1 Interpreter

Interpreter，解释器逐条把字节码翻译成机器码并执行，跨平台的保证。

刚开始执行引擎只采用了解释执行的，但是后来发现某些方法或者代码块被调用执行的特别频繁时，就会把这些代码认定为“热点代码”。

3.2 即时编译器

Just-In-Time compilation(JIT)，即时编译器先将字节码编译成对应平台的可执行文件，运行速度快。即时编译器会把这些热点代码编译成与本地平台关联的机器码，并且进行各层次的优化，保存到内存中。

3.2.1 C1和C2

HotSpot虚拟机里面内置了两个JIT：C1和C2

```
C1也称为Client Compiler，适用于执行时间短或者对启动性能有要求的程序
C2也称为Server Compiler，适用于执行时间长或者对峰值性能有要求的程序
```

Java7开始，HotSpot会使用分层编译的方式

```
也就是会结合C1的启动性能优势和C2的峰值性能优势，热点方法会先被C1编译，然后热点方法中的热点会被C2再次编译
```

3.2.2 AOT

在Java9中，引入了AOT(Ahead of Time)编译器

即时编译器是在程序运行过程中，将字节码翻译成机器码。而AOT是在程序运行之前，将字节码转换为机器码

```
优势 ：这样不需要在运行过程中消耗计算机资源来进行即时编译
劣势 ：AOT 编译无法得知程序运行时的信息，因此也无法进行基于类层次分析的完全虚方法内联，或者基于程序profifile的投机性优化（并非硬性限制，我们可以通过限制运行范围，或者利用上一次运行的程序profifile来绕开这两个限制）
```

3.2.3 Graal VM

官网：https://www.graalvm.org/

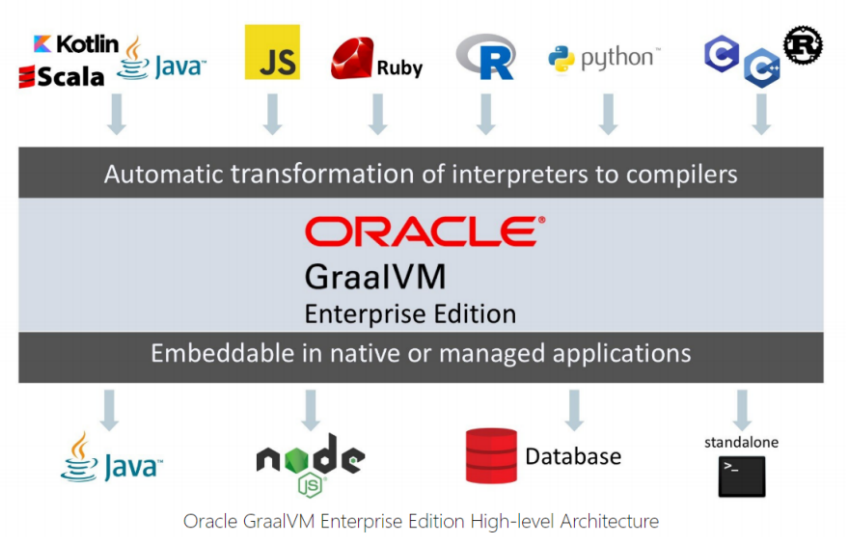
在Java10中，新的JIT编译器Graal被引入。它是一个以Java为主要编程语言，面向字节码的编译器。跟C++实现的C1和C2相比，模块化更加明显，也更加容易维护。

Graal既可以作为动态编译器，在运行时编译热点方法；也可以作为静态编译器，实现AOT编译。

除此之外，它还移除了编程语言之间的边界，并且支持通过即时编译技术，将混杂了不同的编程语言的代码编译到同一段二进制码之中，从而实现不同语言之间的无缝切换。

> **GraalVM core features include:**
>
> - GraalVM Native Image, available as an early access feature — allows scripted applications to be compiled ahead of time into a native machine-code binary
> - GraalVM Compiler — generates compiled code to run applications on a JVM, standalone, or embedded in another system

- Polyglot Capabilities — supports Java, Scala, Kotlin, JavaScript, and Node.js Language Implementation Framework — enables implementing any language for the GraalVM environment
- LLVM Runtime— permits native code to run in a managed environment in GraalVM Enterprise



Oracle GraalVM Enterprise Edition High-level Architecture

> Spring Native 和 Dubbo Native

Spring Native：https://spring.io/blog/2021/03/11/announcing-spring-native-beta

Dubbo Native：https://dubbo.apache.org/zh/docs/references/graalvm/support-graalvm/