Java源代码（.java）—>编译器（javac.exe）—>字节码文件（.class）—>JVM解释器（java.exe）—>机器可执行

的代码

javap 反编译

# 类加载过程

类加载就是虚拟机把Class文件加载到内存，并对数据进行校验，解析和初始化，形成可以虚拟机直接使用的Java类

型，即java.lang.Class

**加载：**通过类的全限定名获取类的二进制字节流，然后将类的静态存储结构转化为方法区的运行时数据结构，在堆中

生成代表该类的java.lang.Class对象，作为方法区数据的入口

**验证：**保证Class文件中的字节流符合虚拟机的要求，并且不会危害虚拟机的安全

**准备：**在方法区内为类变量分配内存并设置默认值

**解析：**将符号引用转化为直接引用（在编译时每个java类都会被编译成一个class文件，但是编译的时候虚拟机并不

知道所引用类的地址，所以就用符号引用代替，而在这个解析阶段就是把这个符号引用转化为了真正的地址）

**初始化：**执行类的静态构造方法和静态代码块，真正初始化

**什么时候发生类的初始化？**

1. 创建类的实例
2. 访问类的静态变量（final常量除外）
3. 调用类的静态方法
4. 反射调用：使用java.lang.reflect包的方法对类进行反射调用
5. 初始化类的子类：当初始化一个类，如果其父类没有被初始化，则先初始化他的父类
6. 虚拟机启动时指定的类（包含main方法的类）

# 如果获得Class对象

1. Class.forName(“类的全限定名”)
2. 实例对象.getClass()
3. 类名.class

# 简述字面量、符号引用和直接引用

（1）字面量：给基本类型变量赋值的方式就叫字面量。

1）文本字符串

2）基本数据类型的值

3）被声明为final的常量

（2）符号引用：就是一组符号来描述目标，可以是任何字面量。

1）类和接口的全限定名

2）字段的名称和描述符

3）方法的名称和描述符

（3）直接引用：就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

1）指向目标的指针：指向对象的指针

2）相对偏移量：指向实例变量、实例方法的指针

3）间接定位到目标的句柄

（4）引用：如果一个变量的类型是类类型，而非基本类型，那么该变量又叫做引用

# 类的生命周期

类的生命周期包括加载、连接、初始化、使用和卸载，其中前三步是类的加载的过程。

使用：new出对象程序中使用

卸载：执行垃圾回收

# 简述Java创建对象的过程

1. 当碰到new关键字时，JVM先去运行时常量池查找该引用指向的类有没有被JVM加载过，如果没有则执行类加

载过程。

（2）检查通过后虚拟机将为新生对象分配内存（指针碰撞和空闲列表），内存大小在类加载时就已经确定下来了。

（3）为成员变量进行零值初始化，引用数据类型默认初始化为null，基本数据类型为0，布尔为false。（这也解释了

为什么类的属性字段无需显示初始化就可以被使用，而方法的局部变量却需要显示初始化才可以访问）

（4）设置对象头，包括该对象哈希码、GC分代年龄、锁信息、类型指针等。

（5）执行类的初始化方法，初始化成员变量，并把堆内对象的首地址赋值给引用。

# 简述JVM给对象分配内存的策略（内存分配方式）

1. 指针碰撞： 假设堆内存规整，在内存中放一个指针作为分界指示器，将使用过的内存放在一边，空闲的放在另

一边，通过挪动指针完成分配。

1. 空闲列表： 如果堆内存不规整，虚拟机必须维护一个列表记录哪些内存可用，在分配时从列表中找到一块足够

大的空间划分给对象并更新列表记录。

选择哪种方式由堆是否规整决定，堆是否规整由垃圾回收器是否有空间压缩能力决定。使用Serial、ParNew等收集器

时，系统采用指针碰撞。使用CMS这种基于清除算法的垃圾收集器时，采用空闲列表。

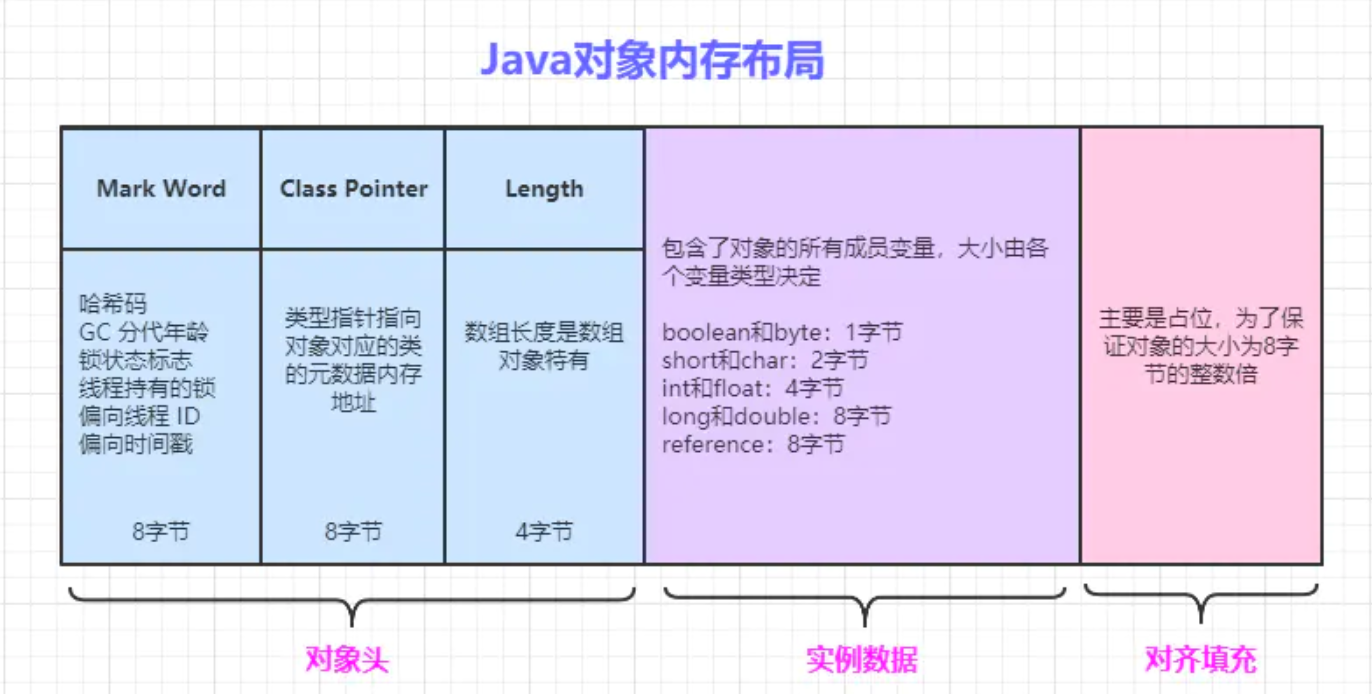
# java对象内存分配是如何保证线程安全的

（1）CAS + 失败重试：该方式效率低。

（2）TLAB：每个线程在Eden区预先分配一小块内存，JVM在给对象分配内存的时候，首先在自己这块"私有"内存

中分配。一般采用这种策略。

# 简述对象的内存布局



对象在堆内存的存储布局可分为对象头（12B）、实例数据和对齐填充。

1. 对象头主要包含两部分数据： Mark Word（8B）、类型指针（4B）。

Mark Word 用于哈希吗、GC分代年龄、锁状态标志位、线程持有的锁、偏向线程ID等信息。

类型指针即对象指向他的类元数据指针，JVM通过该指针来确定对象是哪个类的实例。

如果对象是一个 Java 数组，会有一个Length字段用于记录数组长度的数据。

（2）实例数据存储代码中所定义的各种类型的字段信息。

（3）对齐填充起占位作用。HotSpot虚拟机要求对象的起始地址必须是8的整数倍，因此需要对齐填充。

# 对象的访问定位有哪两种方式

Java中的对象在堆内存中分配内存空间，引用保存在栈内存中，通过引用定位对象通常有两种方式：

1. 句柄访问：此方式在堆空间维护一个句柄池，对象引用中保存的是对象的句柄位置，句柄中包含各对象的实例

数据和类型数据的地址信息。

1. 直接指针访问：即对象引用中保存就是对象的地址。

句柄访问的好处是对象引用中保存的是稳定的对象句柄的地址，因为GC时移动对象是非常普遍的行为，只需要改变

句柄中对象实例地址即可。缺点就是访问效率受影响。

直接指针访问的优点是少了一次指针定位的开销，访问速度快。缺点是当对象地址发生变化时引用中保存的数据也

需要变化。

# 简述JVM中的类加载器

BootstrapClassLoader启动类加载器：加载JAVA\_HOME/jre/lib下的jar包和类。C++编写。

ExtensionClassLoader扩展类加载器：加载JAVA\_HOME/jre/lib/ext目录下的jar包和类。java编写。

AppClassLoader应用类加载器，加载当前应用的ClassPath下的jar包和类。java编写。

# 如何构建自定义类加载器

（1）新建自定义类继承自java.lang.ClassLoader

（2）重写findClass、defineClass、loadClass方法

**findClass()、defineClass()、loadClass()的区别？**

findClass() 根据名称或位置加载.class字节码

defineClass() 把字节码转化为Class

loadClass() 就是主要进行类加载的方法，默认的双亲委派机制就实现在这个方法中。

**自定义类使用双亲委派模型？**

如果你想定义一个自己的类加载器，并且要遵守双亲委派模型，那么可以继承ClassLoader，并且在findClass中实现

你自己的加载逻辑即可。

# 简述双亲委派机制

一个类加载器收到类加载请求之后，首先判断当前类是否被加载过。已经被加载的类会直接返回，如果没有被加载，

首先将类加载请求转发给父类加载器，一直转发到启动类加载器，只有当父类加载器无法完成时才尝试自己加载。

加载类顺序：BootstrapClassLoader->ExtensionClassLoader->AppClassLoader->CustomClassLoader

检查是否加载顺序：CustomClassLoader->AppClassLoader->ExtensionClassLoader->BootstrapClassLoader

**双亲委派机制的优点？**

（1）避免类的重复加载。相同的类被不同的类加载器加载会产生不同的类，双亲委派保证了java程序的稳定运行。

（2）保证核心API不被修改

**如何破坏双亲委派机制？**

重载loadClass()方法

# 简述运行时数据区

JVM内存结构分为三部分：类加载器子系统、运行时数据区、执行引擎

# 简述运行时数据区

线程私有的运行时数据区: 程序计数器、虚拟机栈、本地方法栈。（生命周期和线程一致）

线程共享的运行时数据区:堆、方法区。（随着虚拟机启动而创建，随着虚拟机退出而销毁）

# 简述程序计数器

程序计数器表示当前线程所执行的字节码的行号指示器。

如果线程正在执行Java方法，计数器记录正在执行的虚拟机字节码指令地址。

如果是本地方法，计数器值为Undefined。

占用内存空间小，运行速度最快，无OutOfMemoryError、StackOverflowError。

**为什么用程序计数器记录当前线程的执行地址？**

因为JVM的线程是通过线程轮流切换并分配cpu执行时间的方式来实现的，当线程切换回来的时候，就得知道接着从

哪开始继续执行。

**程序计数器为什么是线程私有的？**

因为JVM的线程是通过线程轮流切换并分配cpu执行时间的方式来实现的，为了准确记录每个线程正在执行的字节码

指令地址，最好的办法就是为每个线程分配一个程序计数器，这样各个线程的程序计数器互不影响，独立存储。

# 简述虚拟机栈

虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会创建一个栈帧用于存储局部变量表、操作

栈、动态链接、方法返回等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈

到出栈的过程。

虚拟机栈会产生两类异常：

StackOverflowError：线程请求的栈深度大于虚拟机允许的深度抛出。

OutOfMemoryError：如果 JVM 栈容量可以动态扩展，虚拟机栈占用内存超出抛出。

1. 局部变量表：用于存放方法参数和方法内定义的局部变量（栈帧的大小主要受局部变量表影响，而局部变量表

的大小在编译期就确定了）

（2）操作数栈：操作数栈主要用于保存计算过程的中间结果。

1. 动态链接：每个栈帧中都包含一个指向运行时常量池中该栈帧所属方法的引用。目的是: 当前方法中如果需要

调用其他方法的时候，能够从运行时常量池中找到对应的符号引用，然后将符号引用转换为直接引用。

1. 方法返回地址：

**虚拟机栈和本地方法栈为什么是线程私有的？**

因为JVM的线程是通过线程轮流切换并分配cpu执行时间的方式来实现的，为了保证线程中的局部变量不被其他线

程访问到，所以虚拟机栈和本地方法栈是线程私有的。

# 简述本地方法栈

本地方法栈与虚拟机栈作用相似，不同的是虚拟机栈为虚拟机执行 Java 方法服务，本地方法栈为本地方法服务。

本地方法栈在栈深度异常和栈扩展失败时分别抛出 StackOverflowError 和 OutOfMemoryError。

# 简述JVM中的堆

堆是虚拟机所管理的内存中最大的一块，被所有线程共享，在虚拟机启动时创建。堆主要作用是存放对象实例，Java

里几乎所有对象实例和数组都在堆分配内存，Java的垃圾回收主要就是针对堆这一区域进行。可通过 -Xms 和 -Xmx

设置堆的最小和最大容量。

如果堆没有内存分配也无法扩展会抛出 OutOfMemoryError异常。

**堆都是线程共享的吗？**

不一定，TLAB。

# 简述方法区

方法区是各个线程共享的内存区域，在虚拟机启动时创建，用于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缓存等数据。方法区在JDK1.7以前采用永久代实现，在JDK1.8中采用元空间实现。

方法区会抛出 OutOfMemoryError异常。

**为什么使用元空间替换永久代？**

1. 避免OOM，因为元空间存在于本地内存，只要本地内存足够，就不会像永久代出现OOM
2. HotSpot和JRockit的合并，JRockit从来没有所谓的永久代

# class常量池、运行时常量池、字符串常量池

1. class常量池：当java文件被编译成class文件之后，会在class文件中生成class常量池，class文件中除了包含

类的版本、字段、方法、接口等描述信息外，还有一个常量池，用于存放编译器生成的各种字面量和符号引用

1. 运行时常量池：当类加载到内存中，jvm就会将class常量池中内容存放到方法区的运行时常量池，由此可知，

运行时常量池也是每个类都有一个。经类加载的解析之后，把符号引用替换为直接引用。

1. 字符串常量池：字符串常量池存的是引用值而不是具体的实例对象，具体的实例对象是在堆中开辟的一块空间

存放的。字符串常量池在JVM中只有一个，被所有类共享。JDK1.6及以前，字符串常量池在运行时常量池。JDK1.7

及以后，字符串常量池被放到了堆内存。

# 堆、栈、方法区存放什么

（1）栈：存放基本数据类型和对象的引用

（2）堆：存放所有new出来的对象和数组

（3）方法区：存放类信息、常量、静态变量

# 堆、栈、方法区溢出的原因

**堆溢出的原因？**

不断创建对象并保证 GC Roots 到对象有可达路径避免垃圾回收，当对象数量多了就会发生OOM。例如在while死循

环中一直new创建实例。

通过堆转储快照分清到底是内存泄露还是内存溢出，如果是内存泄露就找发生内存泄露代码的具体位置，如果是内存

溢出并且内存中对象都必须存活的就调大内存或者查看设计是否合理等。

**栈溢出的原因？**

线程请求的栈深度大于虚拟机所允许的深度，就会发生StackOverflowError，例如一个方法不断递归调用自己

**方法区溢出的原因？**

方法区主要存储类信息、常量、静态变量等，只要不断在运行时产生大量类，方法区就会溢出。例如使用JDK反射或

者CGLib

**运行时常量池溢出原因？**

在JDK1.6及以前，在while死循环中调用intern方法

# 简述堆的分区

堆区分为两大块，一个是Young区，一个是Old区。

Young区分为两大块，一个是Eden区，一个是Survivor区（S0+S1）。

Survivor区分为两大块，一个是S0区，一个是S1区。S0和S1一样大，也可以叫From和To。

Eden:S0:S1=8:1:1

**为什么是8:1:1？**

因为Eden区中的对象都是朝生夕死的，统计学测算出超过98%以上的对象是一次就会被minor gc时回收的。GC时9

层丢掉，剩下的放在S区，所以给了1层，因为S0和S1要相互对调，这样比例就是8:1:1了。

**为什么需要Survivor区?只有Eden不行吗？**

Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生。Survivor的预筛选保证，只有经历

16Minor GC还能在新生代中存活的对象,才会被送到老年代。

**为什么需要两个Survivor区？**

最大的好处就是解决了碎片化。因为Eden满了的时候，此时进行Minor GC，Eden和Survivor各有一些存活对象，如

果此时把Eden区的存活对象硬放到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎

片化。

**为什么分代回收？**

（1）绝大多数对象都是朝生夕死的

（2）熬过越多次垃圾收集过程的对象就越难以消亡

# Minor GC 、Major GC 和 Full GC

Minor GC：新生代

Major GC：老年代（基本上发生了一次Major GC 就会发生一次 Minor GC。并且Major GC 的速度往往会比 Minor

GC 慢 10 倍）

Full GC：新生代 + 老年代

**Minor GC 与 Full GC分别什么时候发生？**

Minor GC：JVM无法为一个新对象分配空间，比如Eden 空间占满了的时候（Survivor满不会引发GC）

Full GC：

1. 老年代空间不足，大对象分配或者持续存活的对象转移到老年代引发的空间不足
2. 方法区空间不足
3. 显示调用System.gc( )，系统建议执行Fu11GC，但是不必然执行
4. 由Eden区、From区向To区复制时，对象大小大于To区可用内存，则把该对象转存到老年代，且老年代的可用

内存该对象大小，同时未开启空间分配担保（空间分配担保机制）

（5）在Minor GC之前判断历次晋升老年代的平均大小大于老年代的可用内存（空间分配担保机制）

# JVM对象的内存分配策略

首先会进行逃逸分析，如果一个对象没有逃逸出方法的话，可能会优化成栈上分配。否则就在Eden区分配，如果开

启了本地线程分配缓冲，则优先在TLAB上分配，线程的TLAB区分配不下才会往公共的Eden区分配。当Eden没有足

够空间时将发起一次MinorGC。如果是大对象（很长的字符串和数组）直接分配在老年代。长期存活的对象（默认年

龄15）直接进入老年代。

# 逃逸分析

在JIT（即时编译：将热点代码翻译成机器代码） 阶段，会判断对象是否发生逃逸。如果未发生逃逸，则会做出栈上

分配、同步消除、标量替换等优化。

1. 栈上分配：将堆分配转化为栈分配。
2. 同步消除：如果一个对象被发现只有一个线程被访问到，那么对于这个对象的操作可以不考虑同步。
3. 标量替换：对象拆解成若干个成员变量来代替，这个过程就是标量替换。

# 什么是TLAB

TLAB（Thread Local Allocation Buffer），即线程分配缓冲区，在堆中划分出一块区域，为每个线程所独占

# JVM发生GC时对象在堆内存移动的判断机制（如何判断发生GC时对象在堆内存移动呢）

1. 动态年龄判断机制：如果在Survivor空间中相同年龄所有对象大小的综合大于Survivor空间的一半，年龄大于

或者等于该年龄的对象就可以直接进入老年代，无需等到参数设置的年龄。

1. 空间分配担保机制：虚拟机在进行minorGC之前会判断老年代最大的可用连续空间是否大于新生代的所有对象

总空间

1. 如果大于的话，直接执行minorGC
2. 如果小于，是否开启HandlerPromotionFailure，没有开启直接FullGC
3. 如果开启了HanlerPromotionFailure, JVM会判断历次晋升的大小是否大于老年代的最大连续内存空间，如果大

于直接执行FullGC

1. 如果小于的话，执行minorGC

# 如何判断对象是否是垃圾

1. 引用计数法：对象上添加一个引用计数器，每当有一个对象引用它时，计数器加1，引用失效时，计数器减1，

计数器为0表示该对象不被使用。缺点是无法解决对象之间相互引用的问题。

（2）可达性分析：通过 GC Roots 的根对象作为起始节点，从这些节点开始，根据引用关系向下搜索，如果某个对

象没有被搜到，则会被标记为垃圾。

可作为GC Roots的对象：

（1）虚拟机栈(栈帧中的本地变量表)引用的对象

（2）方法区中静态变量引用的对象

（3）方法区中常量引用的对象

（4）Native方法引用的对象

# 垃圾收集器的原理是什么

通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。当GC确定一些对象为"不可达"时，GC就有责任回收

这些内存空间。

**垃圾回收器可以马上回收内存吗？**

垃圾收集器不可以被强制执行，但程序员可以通过调研System.gc方法来建议执行垃圾收集。

**有什么办法主动通知虚拟机进行垃圾回收？**

程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

# 不可达的对象是否非死不可

使用可达性分析法判断一个对象是否可被回收需要经过两次标记：

（1）当发现对象到GC Roots没有引用链，进行第一次标记。

（2）如果对象重写了finalize方法，且还未执行过，那么对象会被插入到一个F-Queue队列中，再由一个虚拟机自动

创建的，低优先级的Finalizer线程触发其finalize方法。finalize方法中如果对象与引用链中的任意对象建立联系，那

么在进行第二次标记时ObjectA会被移出即将回收的集合（注：finalize方法只会被JVM调用一次）

# 详解方法区中的垃圾回收

方法区的垃圾回收主要回收两部分内容：1. 废弃常量。2. 无用的类。

**如何判断一个常量是废弃常量？**

以字面量回收为例，如果一个字符串“abc”已经进入常量池，但是当前系统没有任何一个String对象引用了叫做

“abc”的字面量，那么，如果发生垃圾回收并且有必要时，“abc”就会被系统移出常量池。

**如何判断一个类是无用的类？**

（1）该类的所有实例都已经被回收，即Java堆中不存在该类的任何实例。

（2）加载该类的ClassLoader已经被回收。

（3）该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

# 如何减少GC的次数

（1）对象不用时最好显式置为 Null

（2）尽量少用 System.gc()

（3）尽量少用静态变量：静态变量不会被 GC 回收,它们会一直占用内存

（4）尽量使用 StringBuffer,而不用 String 来累加字符串

（5）分散对象创建或删除的时间

（6）尽量少用 finalize 函数

（7）使用软引用类型

# 如简述Java的引用类型

（1）强引用： 被强引用关联的对象不会被回收。一般采用 new 方法创建强引用。

（2）软引用：被软引用关联的对象只有在内存不够的情况下才会被回收。一般采用 SoftReference 类来创建

软引用。

（3）弱引用：垃圾收集器碰到即回收，也就是说它只能存活到下一次垃圾回收发生之前。一般采用

WeakReference 类来创建弱引用。

（4）虚引用： 无法通过该引用获取对象。唯一目的就是为了能在对象被回收时收到一个系统通知。虚引用必

须与引用队列联合使用。

# 详解垃圾回收算法

1. 标记清除（老年代）：分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所

有被标记的对象。这种算法一是效率不高，二是会产生大量不连续的碎片，不利于大对象的分配。

1. 复制（新生代）：复制算法是为了解决效率问题而出现的，它将可用的内存分为两块，每次只用其中一块，当

这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉，同时

解决了内存碎片化问题。

（3）标记整理（老年代）：复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%

存活，那么需要有额外的空间进行分配担保。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用

复制算法。标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象

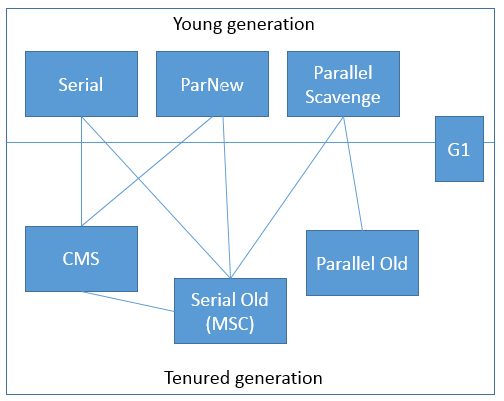
都向一端移动，然后直接清理掉边界以外的内存。

（4）分代收集：现代商用虚拟机基本都采用分代收集算法来进行垃圾回收。根据对象的生命周期的不同将内存划分

为几块，然后根据各块的特点采用最适当的收集算法。大批对象死去、少量对象存活的（新生代），使用复制算法，

复制成本低；对象存活率高、没有额外空间进行分配担保的（老年代），采用标记-清理算法或者标记-整理算法。

# 详解垃圾回收器



垃圾收集器就是垃圾回收算法的具体实现了，HotSpot包含的回收器如上图

1. **Serial收集器：**是一个采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程

去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止。

特点：

1. 需要STW（Stop The World），停顿时间长。
2. 简单高效，对于单个CPU环境而言，Serial收集器由于没有线程交互开销，可以获取最高的单线程收集效率。
3. **ParNew收集器：**ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集外，其

余行为和Serial收集器完全一样，包括使用的也是复制算法。在多CPU环境下，比Serial收集器资源利用率高。

1. **Parallel Scavenge收集器：**Parallel Scavenge收集器也是一个新生代收集器，也是用复制算法的收集器，也是

并行的多线程收集器，主要关注吞吐量。

1. **Serial Old收集器：**Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理算法”，这个收集器

的主要意义也是在于给Client模式下的虚拟机使用。

1. **Parallel Old收集器：**Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。吞吐量优先。
2. **CMS收集器：**CMS收集器以最小的停顿时间为目标的收集器。使用标记 - 清除算法。收集过程分为如下四步：

1. 初始标记：标记GCRoots能直接关联到的对象，Stop The World，时间很短。

2. 并发标记：进行GCRoots Tracing（可达性分析）过程，时间很长。

3. 重新标记：修改并发标记因用户程序变动的内容，Stop The World。

4. 并发清除：回收内存空间，时间很长。

因为耗时最长的并发标记和并发清除两个阶段可以和用户线程一起工作，所以总体可以看成CMS的内存回收过程是与

用户线程一起并发执行的。

特点：

1. 对CPU资源非常敏感，可能会导致应用程序变慢，吞吐量下降。
2. 无法处理浮动垃圾，因为并发清理阶段用户线程还在运行，可能会产生新的垃圾，只能等下次GC清除。
3. 由于采用的标记 - 清除算法，会产生大量的内存碎片，不利于大对象的分配，可能会提前触发一次Full GC。

**（7）G1收集器：**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。收集过程如下：

1. 初始标记：标记GCRoots能直接关联到的对象，Stop The World，时间很短。
2. 并发标记：进行GCRoots Tracing（可达性分析）过程。
3. 最终标记：修改并发标记因用户程序变动的内容，Stop The World。
4. 筛选回收：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。

与前几个收集器相比，G1有以下特点：

1. 并行和并发：使用多个CPU来缩短Stop The World停顿时间，与用户线程并发执行。

2. 分代收集：独立管理整个堆。

3. 空间整合：基于标记 - 整理算法，无内存碎片产生。

4. 可预测的停顿：能指定一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

# 吞吐量和停顿时间

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）

停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；

高吞吐量则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。

# 内存泄漏与内存溢出的区别

内存泄漏是指不再使用的对象无法得到及时的回收，持续占用内存空间，从而造成内存空间的浪费。

内存溢出是指程序申请内存时，没有足够的内存供申请者使用。

内存泄漏很容易导致内存溢出，但内存溢出不一定是内存泄漏导致的。

# JDK1.8默认垃圾回收器

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9 默认垃圾收集器G1

# JDK1.8新特性

（1）Lambda表达式

（2）函数式接口

（3）默认方法：default和static

（4）方法引用

（5）Stream API

（6）Optional类

（7）新的日期API

# JVM性能调优常用命令

1. jps：查看java进程
2. jinfo：实时查看和调整JVM配置参数
3. jstat：查看虚拟机性能统计信息
4. jstack：查看线程堆栈信息
5. jmap：生成堆的dump文件

# JVM性能调优常用工具

1. JConsole：提供内存监控，线程监控
2. JVisual VM：
3. Arthas：阿里的 阿尔萨斯
4. MAT（Memory Analyzer tool）：

# JVM性能调优日志文件分析工具

1. gceasy：内存泄露检测、GC暂停原因分析、JVM配置建议优化等
2. GCViewer