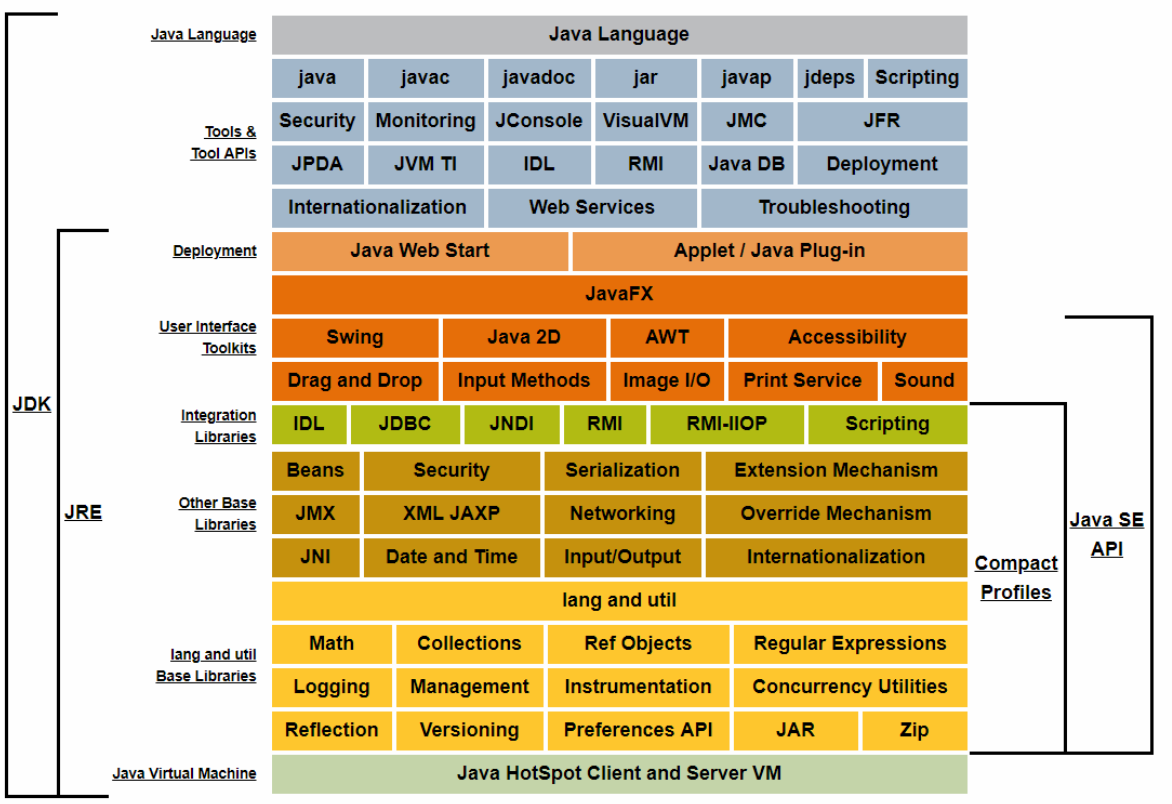
JVM面试突击班

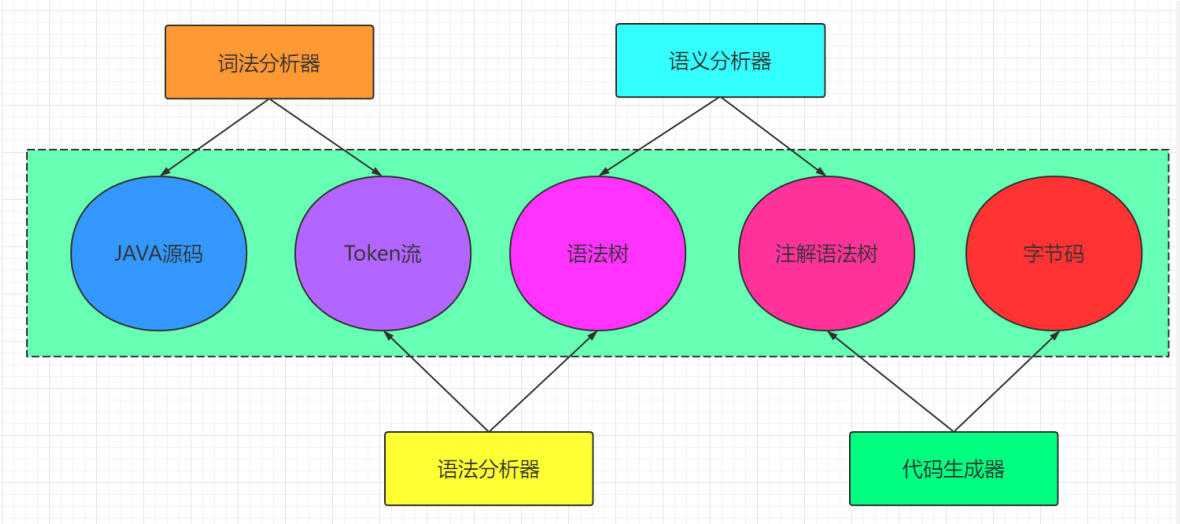
核心原理　2节课　　调参　实操　解决错误　　1节课

JDK，JRE以及JVM的关系

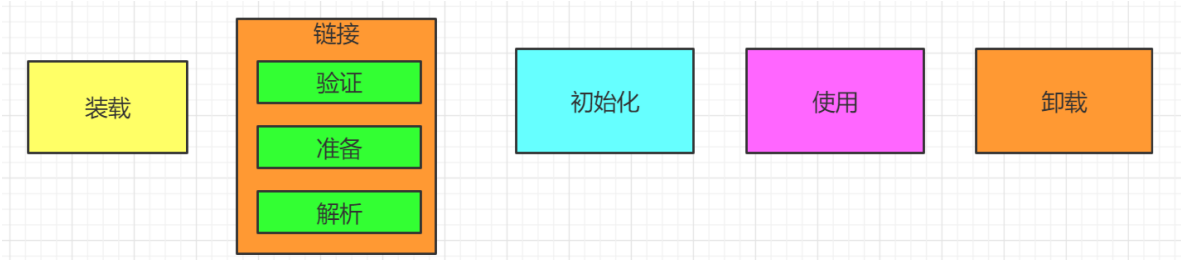


我们的编译器到底干了什么事?

仅仅是将我们的 .java 文件转换成了 .class 文件，实际上就是文件格式的转换，对等信息转换。



类加载机制是什么?



> **所谓类加载机制就是**

```
虚拟机把Class文件加载到内存
并对数据进行校验，转换解析和初始化
形成可以虚拟机直接使用的Java类型，即java.lang.Class
```

装载(Load)

ClassFile--- 字节流 ---- 类加载器

> 查找和导入class文件

（1）通过一个类的全限定名获取定义此类的二进制字节流
（2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
（3）在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口

链接(Link)

验证(Verify)

> 保证被加载类的正确性

- 文件格式验证
- 元数据验证
- 字节码验证
- 符号引用验证

准备(Prepare)

> 为类的静态变量分配内存，并将其初始化为默认值

```
public class Demo1 {
private static int i;

public static void main(String[] args) {
    // 正常打印出0，因为静态变量i在准备阶段会有默认值0
    System.out.println(i);
}
}
```

```
public class Demo2 {
    public static void main(String[] args) {
        // 编译通不过，因为局部变量没有赋值不能被使用
        int i;
        System.out.println(i);
    }
}
```

解析(Resolve)

> 把类中的符号引用转换为直接引用
>
> > 符号引用就是一组符号来描述目标，可以是任何字面量。
> > 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。

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。
解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符7类符号引用进行。

初始化(Initialize)

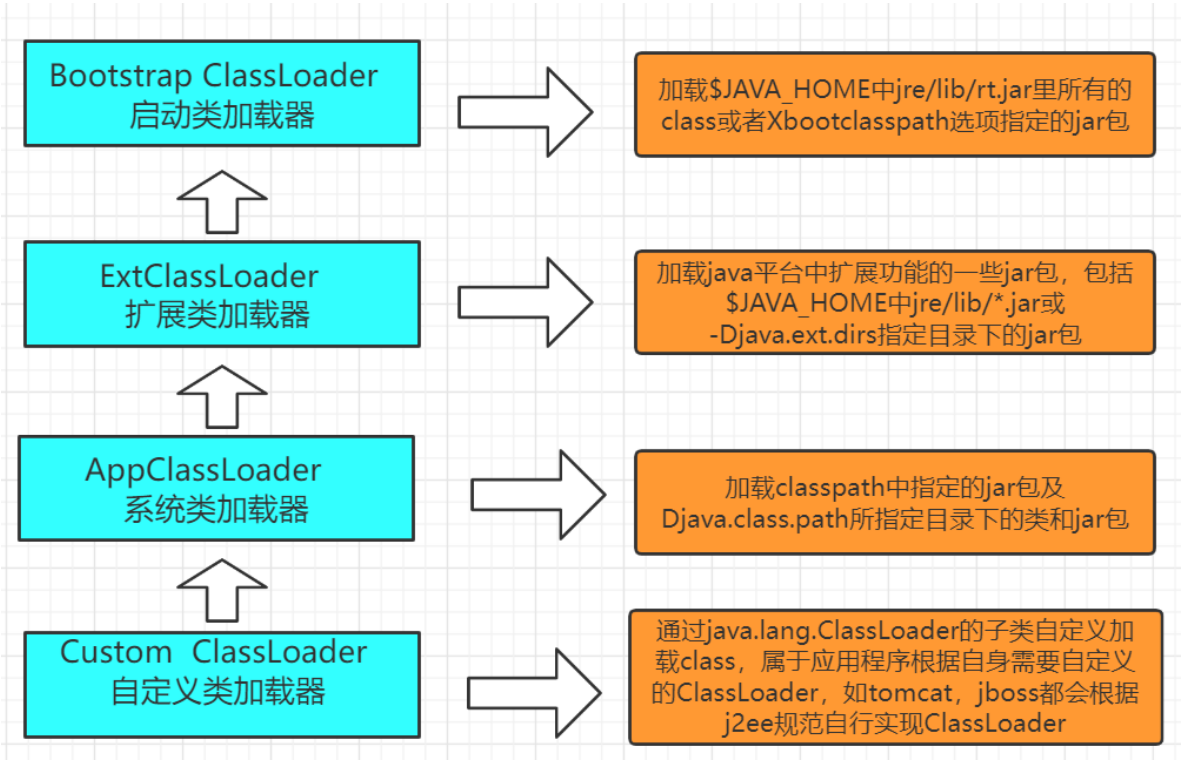
> 对类的静态变量，静态代码块执行初始化操作 执行了Clinit方法

类加载器有哪些?

类加载器ClassLoader

> 在装载(Load)阶段，其中第(1)步:通过类的全限定名获取其定义的二进制字节流，需要借助类装载器完成，顾名思义，就是用来装载Class文件的。

图解:



1）Bootstrap ClassLoader 负责加载 JAVA_HOME中 jre/lib/rt.jar 里所有的class或Xbootclassoath选项指定的jar包。由C++实现，不是ClassLoader子类。

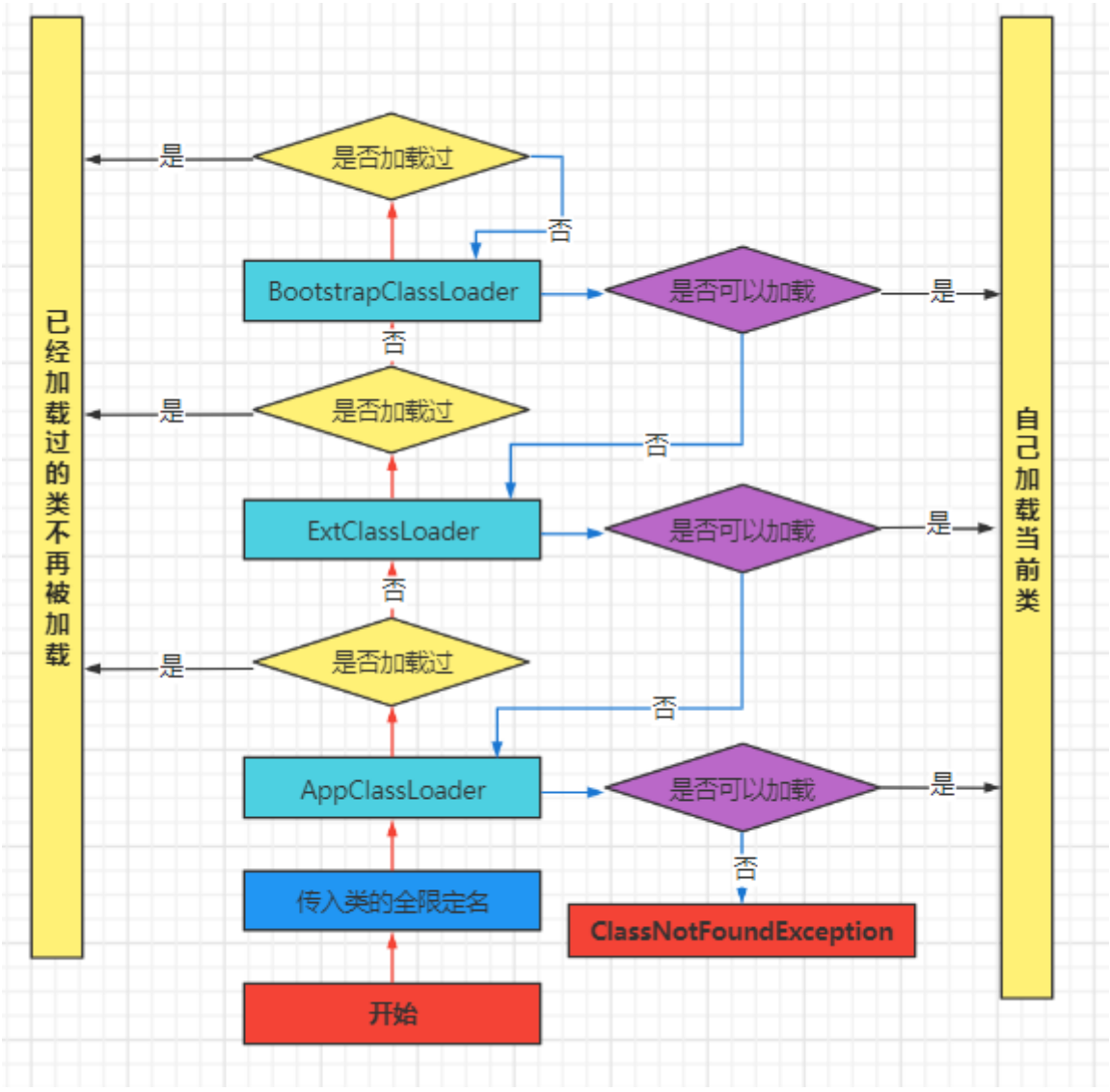
2）Extension ClassLoader 负责加载java平台中扩展功能的一些jar包，包括`$$JAVA_HOME中 jre/lib/*.jar 或 -Djava.ext.dirs指定目录下的jar包。

3）App ClassLoader 负责加载classpath中指定的jar包及 Djava.class.path 所指定目录下的类和jar包。

4）Custom ClassLoader 通过java.lang.ClassLoader的子类自定义加载class，属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader。

双亲委派以及打破双亲委派 父类委托机制

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        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.
                    long t1 = System.nanoTime();
                    c = findClass(name);

                    // this is the defining class loader; record the stats
                    sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 -
t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
```



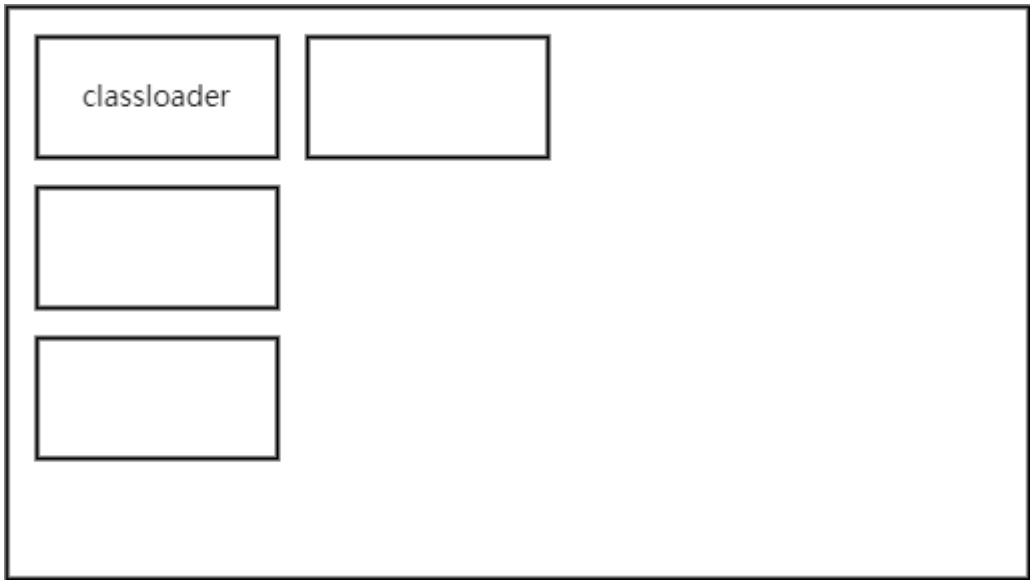
向上检查　向下委派

打破双亲委派：

3种方式

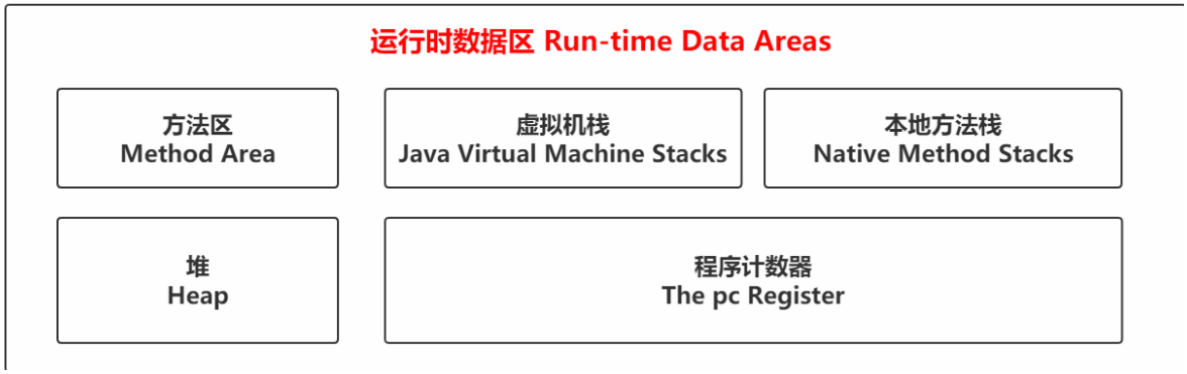
复写

SPI　Service　Provider Interface　　接口



OSGI　热更新　热部署　　外包中的外包

运行时数据区



（1）方法区是各个线程共享的内存区域，在虚拟机启动时创建

```
The Java Virtual Machine has a method area that is shared among all Java Virtual
Machine threads.
The method area is created on virtual machine start-up.
```

（2）虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来

```
Although the method area is logically part of the heap,......
```

（3）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等数据

```
It stores per-class structures such as the run-time constant pool, field and
method data, and the code for methods and constructors, including the special
methods (§2.9) used in class and instance initialization and interface
initialization.
```

（4）当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常

```
If memory in the method area cannot be made available to satisfy an allocation
request, the Java Virtual Machine throws an OutOfMemoryError.
```

> 注意：JVM运行时数据区是一种规范，真正的实现在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space

Heap(堆)

（1）Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。

（2）Java对象实例以及数组都在堆上分配。

Java Virtual Machine Stacks(虚拟机栈)

假如目前的阶段是初始化完成了，后续做啥呢？肯定是Use使用咯，不用的话这样折腾来折腾去有什么意义？那怎样才能被使用到？换句话说里面内容怎样才能被执行？比如通过主函数main调用其他方法，这种方式实际上是main线程执行之后调用的方法，即要想使用里面的各种内容，得要以线程为单位，执行相应的方法才行。**那一个线程执行的状态如何维护？一个线程可以执行多少个方法？这样的关系怎么维护呢？

（1）虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。

```
Each Java Virtual Machine thread has a private Java Virtual Machine stack,
created at the same time as the thread.
```

（2）每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。

调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。

```
 A Java Virtual Machine stack stores frames (§2.6).
```

```
A new frame is created each time a method is invoked. A frame is destroyed when
its method invocation completes.
```

栈帧：每个栈帧对应一个被调用的方法，可以理解为一个方法的运行空间。

每个栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向运行时常量池的引用(A reference to the run-time constant pool)、方法返回地址(Return Address)和附加信息。

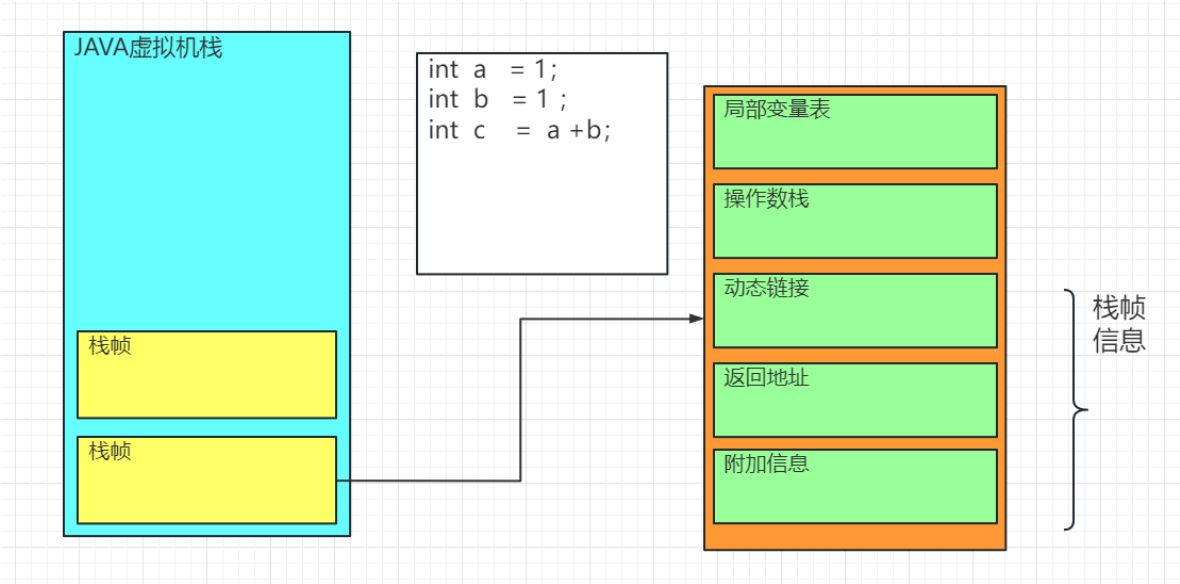
局部变量表:方法中定义的局部变量以及方法的参数存放在这张表中
局部变量表中的变量不可直接使用，如需要使用的话，必须通过相关指令将其加载至操作数栈中作为操作数使用。

操作数栈:以压栈和出栈的方式存储操作数的

动态链接:每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)。

方法返回地址:当一个方法开始执行后,只有两种方式可以退出，一种是遇到方法返回的字节码指令；一种是遇见异常，并且这个异常没有在方法体内得到处理。

栈帧的结构



局部变量表：方法中的局部变量以及方法的参数会存放在这

操作数栈：也是一个栈，他是以压栈以及出栈的方式来存储操作数的

```
int   a  =  1;

int   b   = 1 ;

int  c   = a  + b;
```

方法的返回地址：

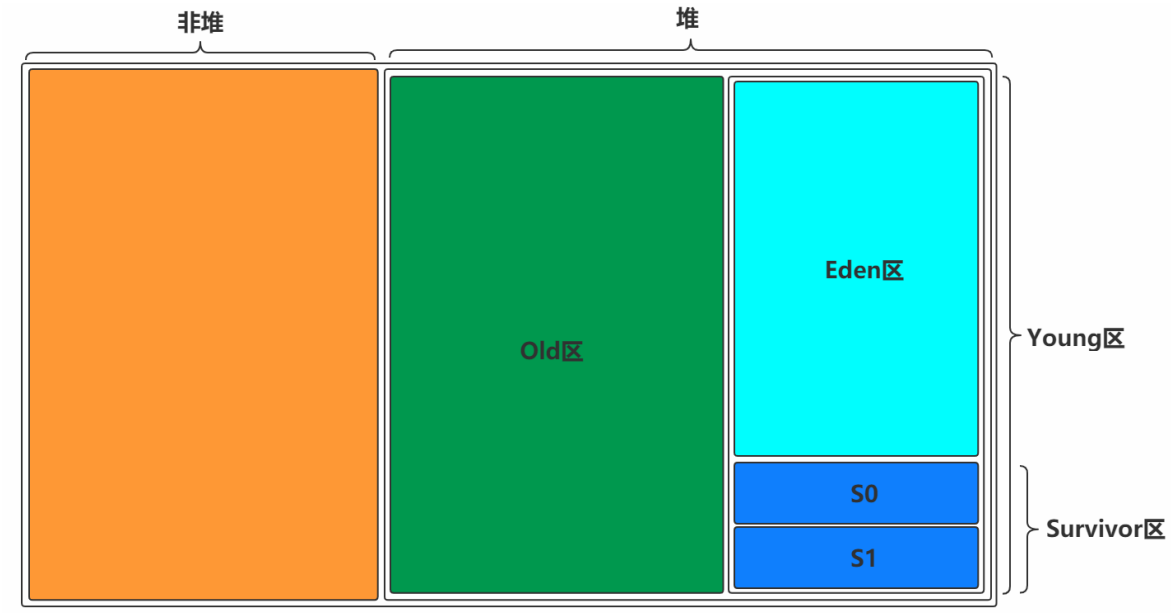
一个方法执行之后，只有两种情况可以退出，遇到返回的字节码指令　　异常返回

动态链接：动态链接将这些符号方法引用转换为具体的方法引用

符号引用转化成直接引用

```
void   a(){

b();

}

void   b(){

c();

}

void   c(){
xxxxx
}
```

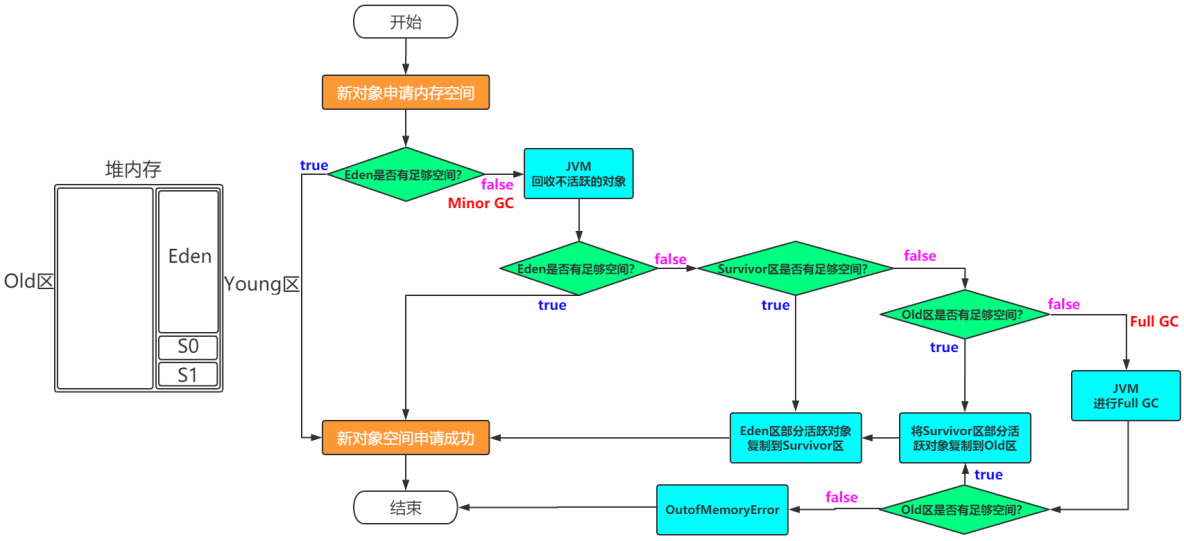
堆为什么进行分代设计



老年代的担保机制

为什么Eden：S0：S1 是8：1：1

对象的创建以及分配过程



方法区与元数据区以及持久代到底是什么关系?

Full GC = young GC + Old GC + Meta Space GC

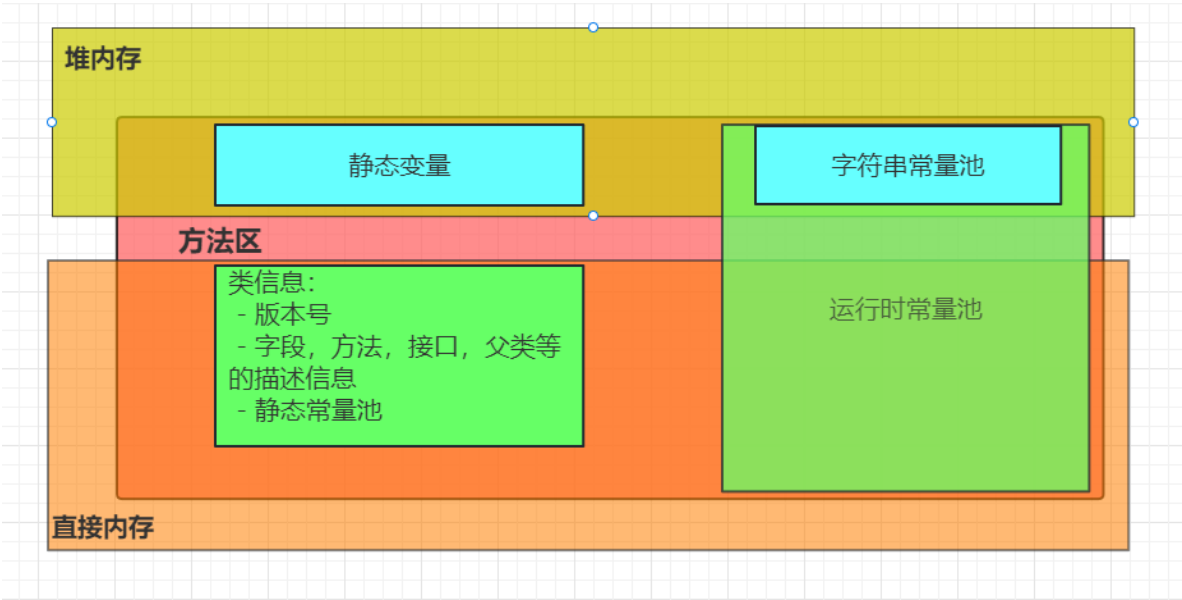
规范：方法区

实现：

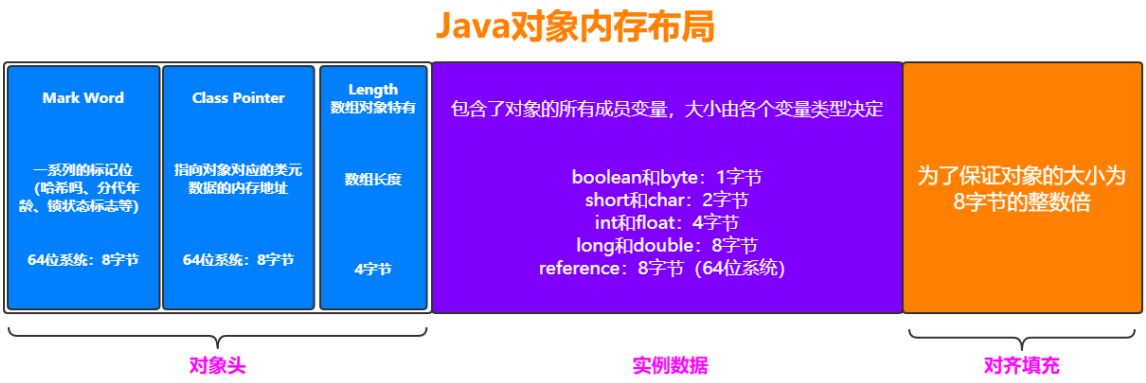
JDK1.7之前　永久代 持久代　Perm Space　类的总数　常量池大小　方法的数量

JDK1.8以及其之后　元空间 元数据区　MetaSpace

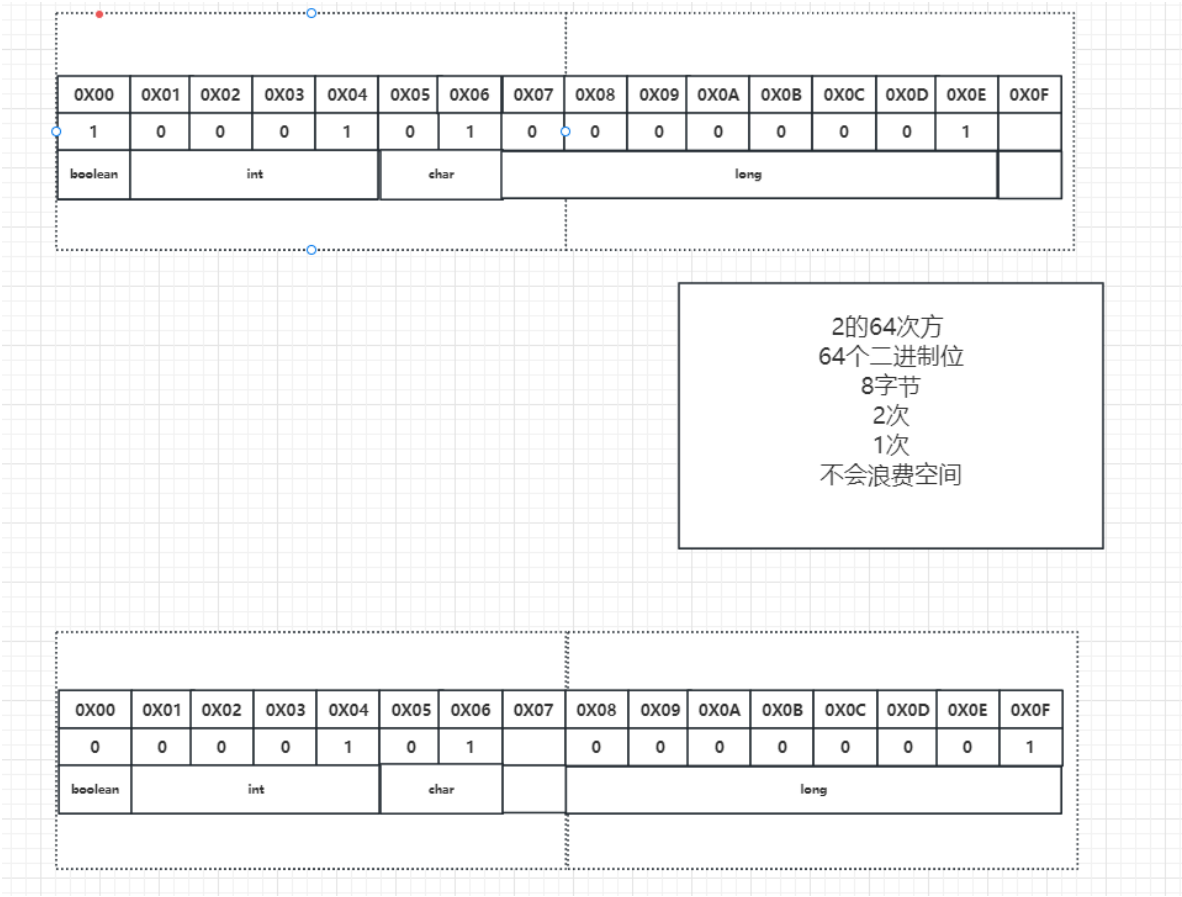
JVMTI　开后门



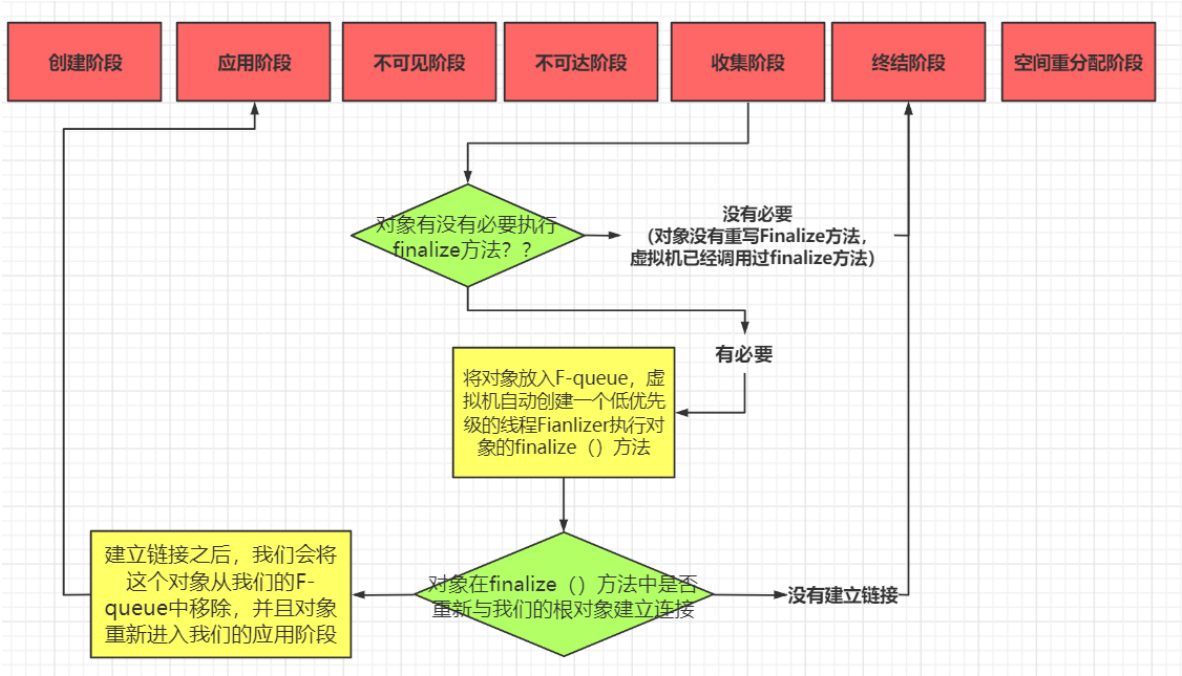
对象的内存布局



为了加快CPU的读取效率 哪怕是引用类型 也只是读取一次



对象被判定为不可达对象之后就“死”了吗



垃圾收集算法

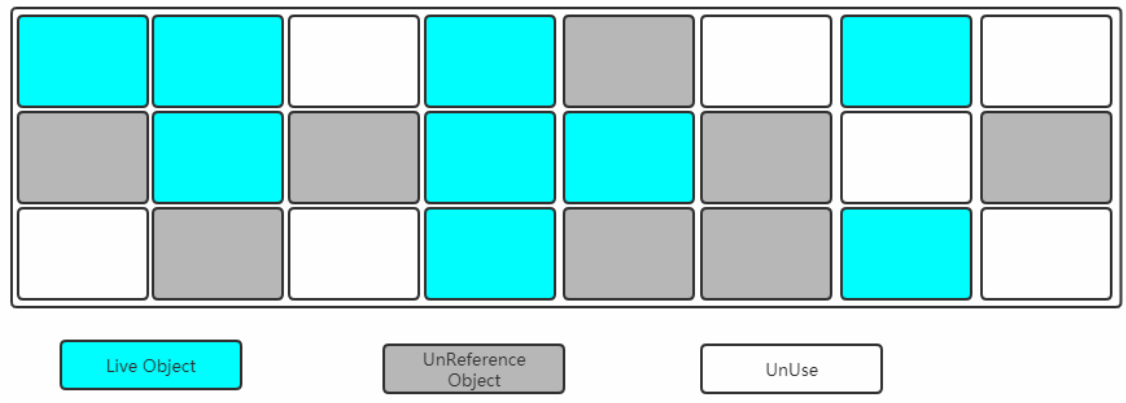
> 已经能够确定一个对象为垃圾之后，接下来要考虑的就是回收，怎么回收呢？得要有对应的算法，下面介绍常见的垃圾回收算法。高效　健壮

标记-清除(Mark-Sweep)

- 标记

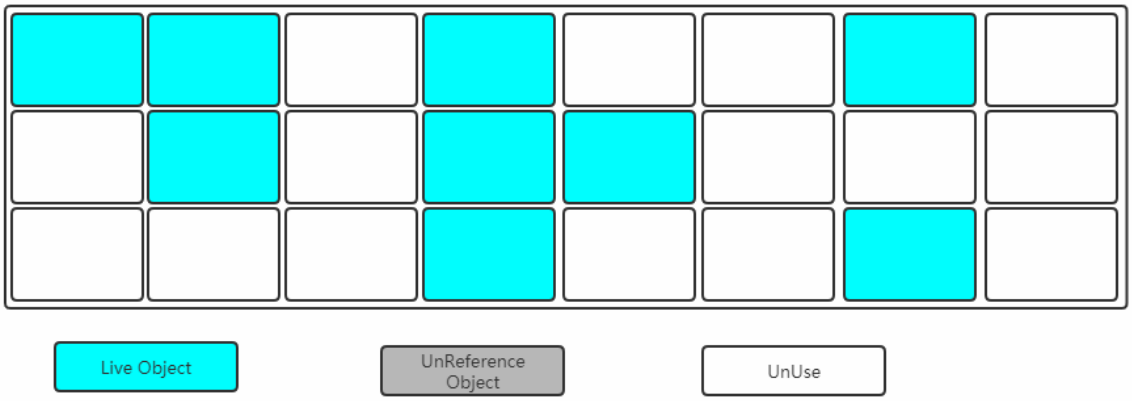
找出内存中需要回收的对象，并且把它们标记出来

> 此时堆中所有的对象都会被扫描一遍，从而才能确定需要回收的对象，比较耗时



- 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

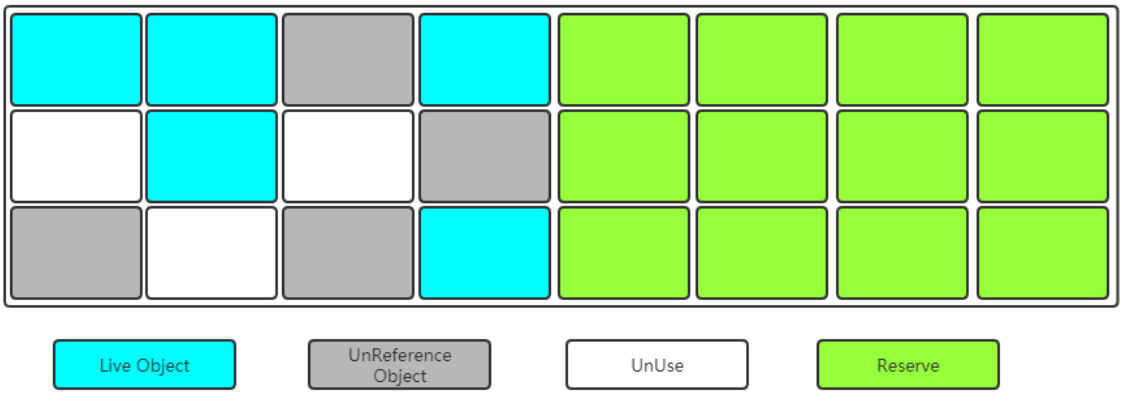


缺点

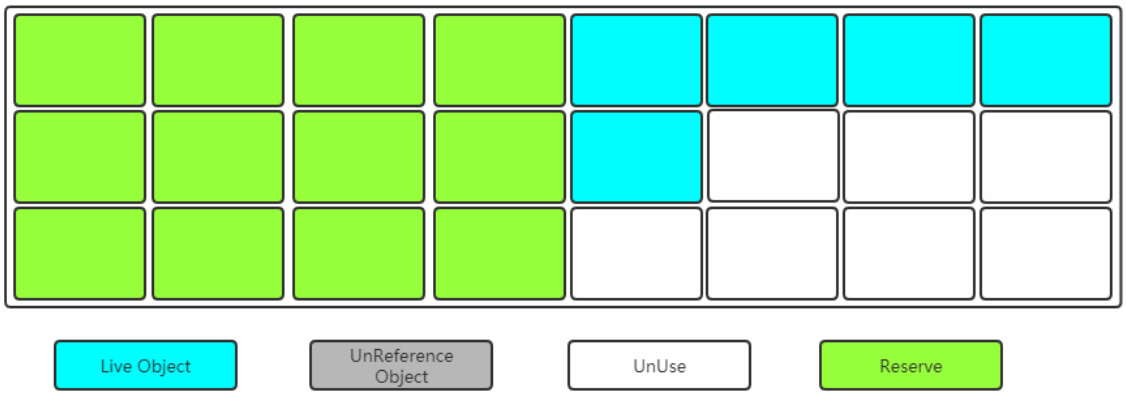
标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程
序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。
(1)标记和清除两个过程都比较耗时，效率不高
(2)会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

标记-复制(Mark-Copying)

将内存划分为两块相等的区域，每次只使用其中一块，如下图所示：



当其中一块内存使用完了，就将还存活的对象复制到另外一块上面，然后把已经使用过的内存空间一次清除掉。



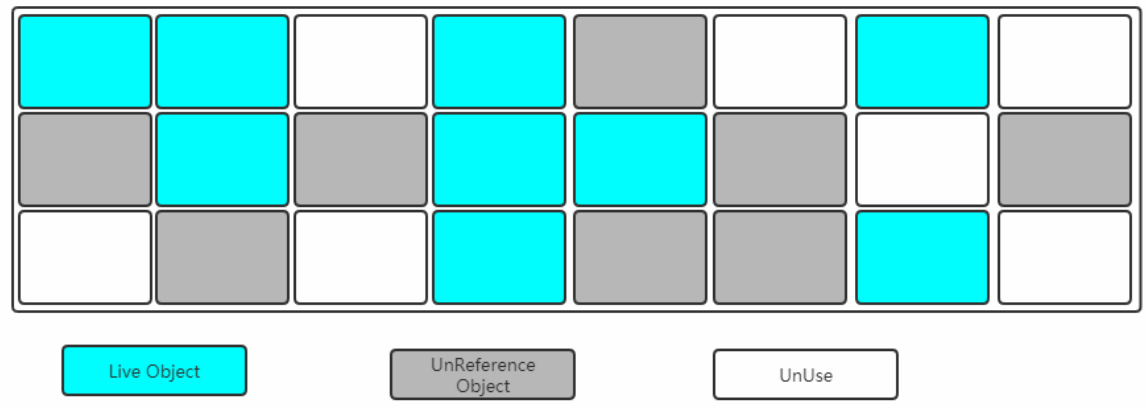
缺点：空间利用率降低。

标记-整理(Mark-Compact)

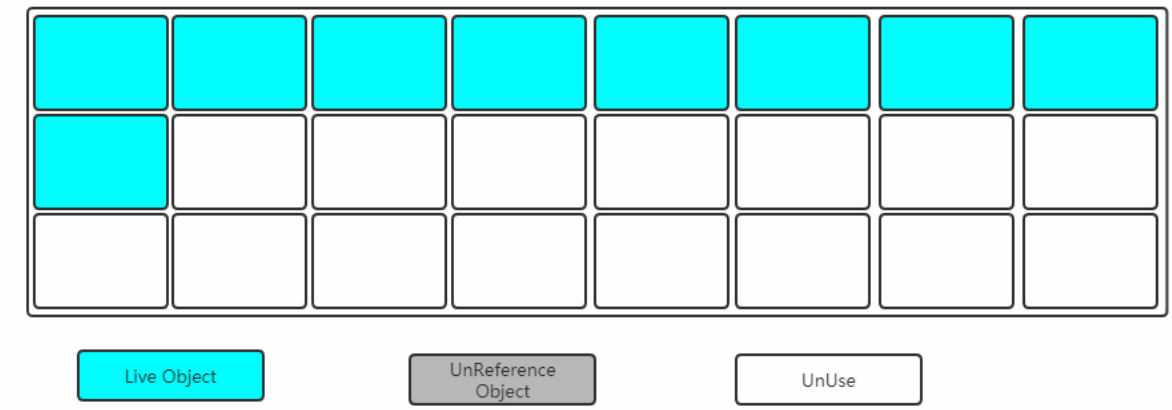
> 复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都有100%存活的极端情况，所以老年代一般不能直接选用这种算法。

标记过程仍然与"标记-清除"算法一样，但是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，然后直接清理掉端边界以外的内存。

> 其实上述过程相对"复制算法"来讲，少了一个"保留区"



让所有存活的对象都向一端移动，清理掉边界意外的内存。



分代收集算法

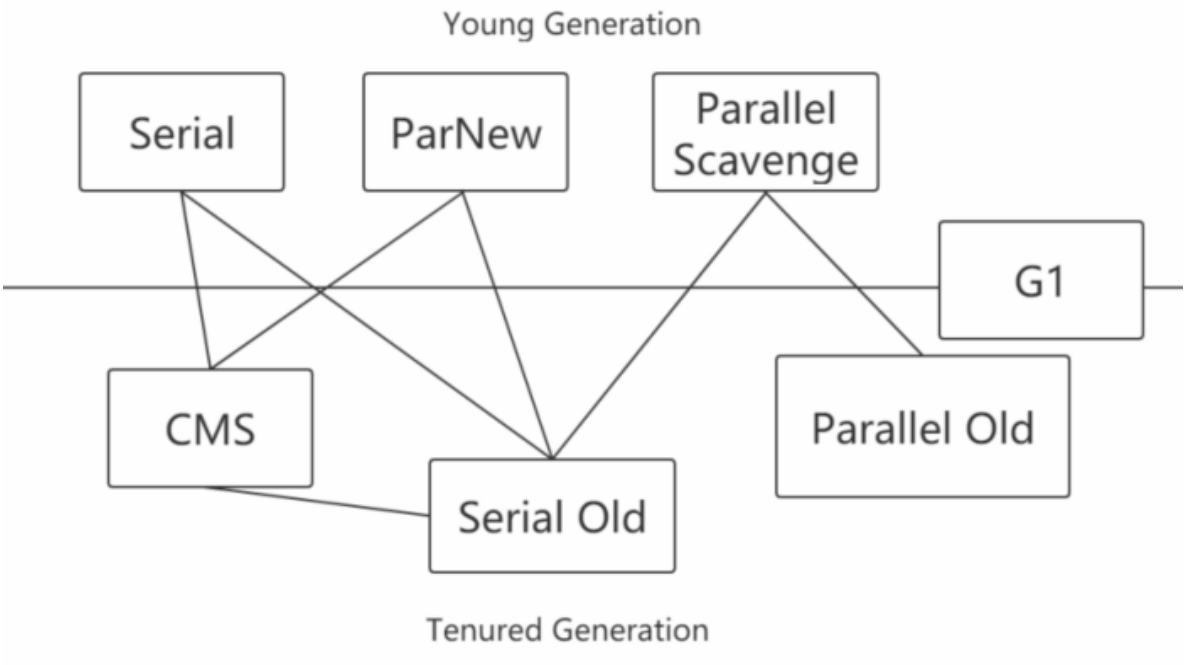
> 既然上面介绍了3中垃圾收集算法，那么在堆内存中到底用哪一个呢?

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)

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。

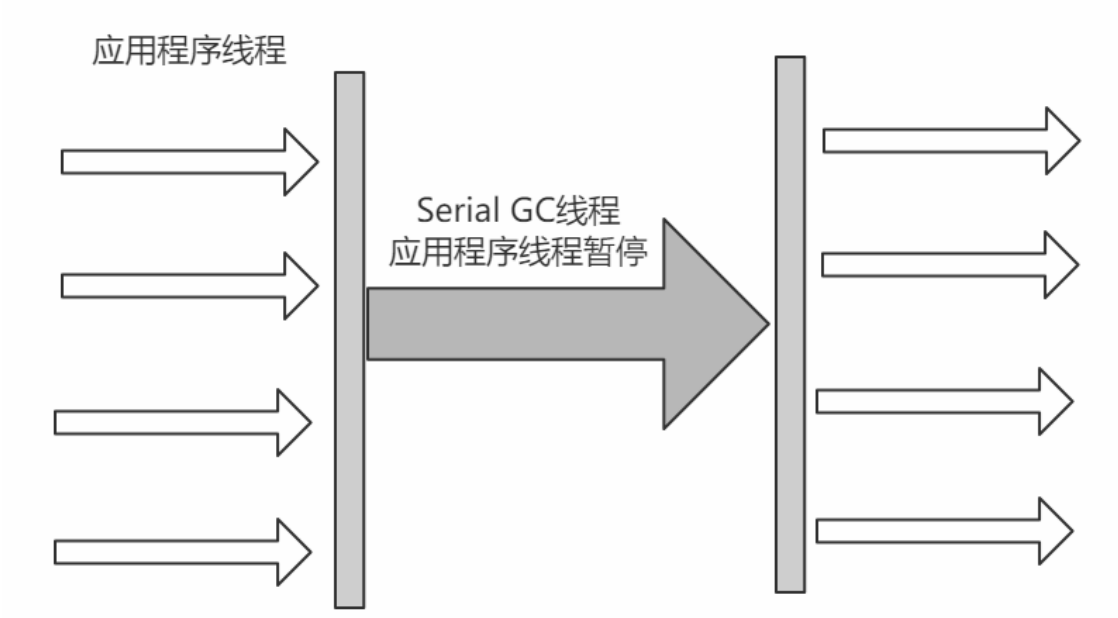


- Serial

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK1.3.1之前）是虚拟机新生代收集的唯一选择。

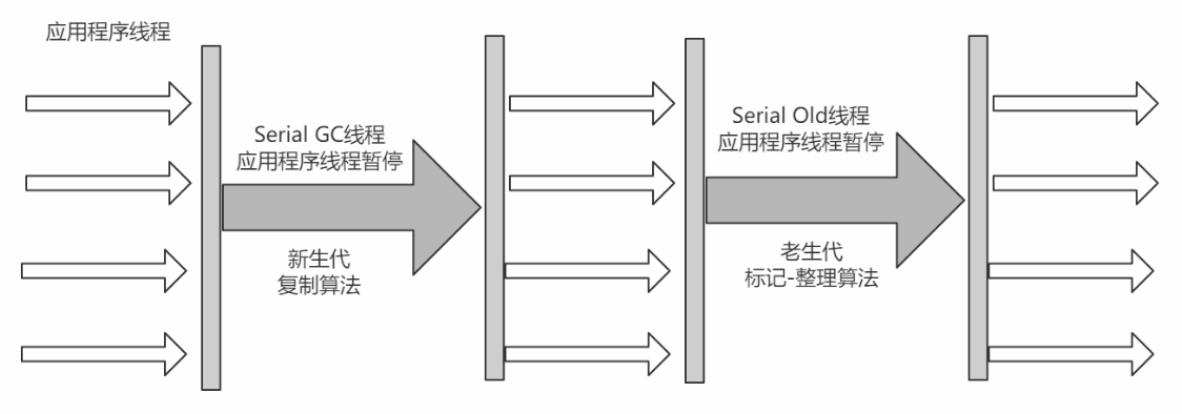
它是一种单线程收集器，不仅仅意味着它只会使用一个CPU或者一条收集线程去完成垃圾收集工作，更重要的是其在进行垃圾收集的时候需要暂停其他线程。

```
优点：简单高效，拥有很高的单线程收集效率
缺点：收集过程需要暂停所有线程
算法：复制算法
适用范围：新生代
应用：Client模式下的默认新生代收集器
```



- Serial Old

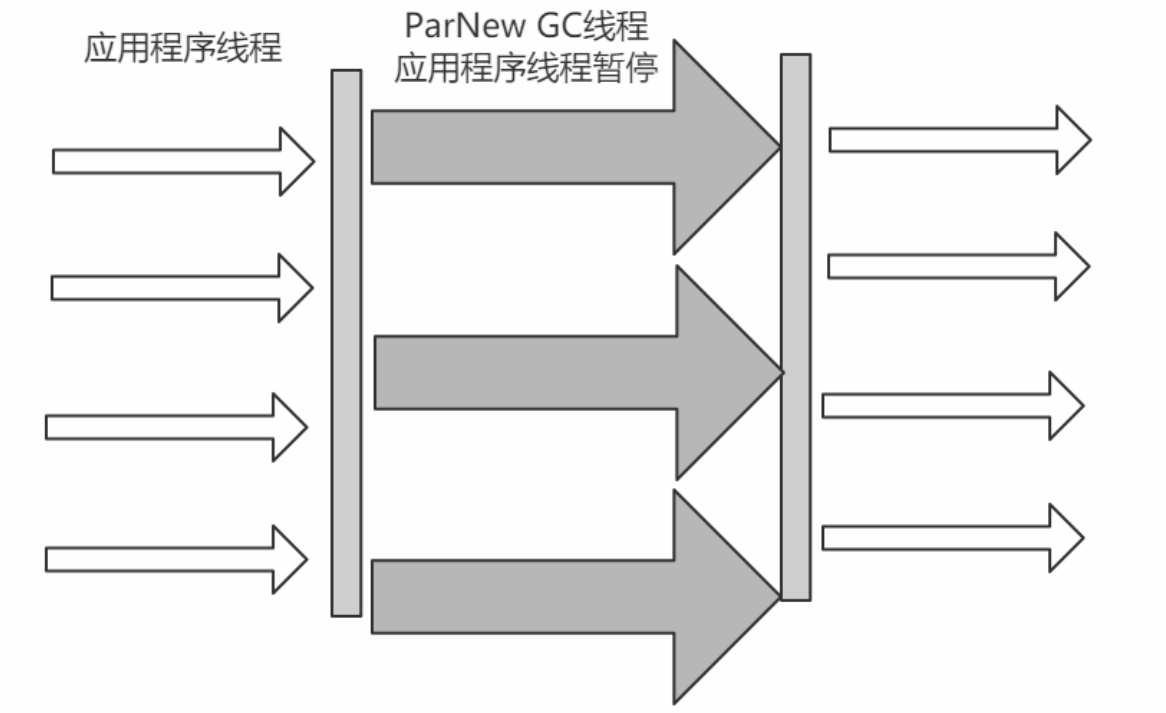
Serial Old收集器是Serial收集器的老年代版本，也是一个单线程收集器，不同的是采用"**标记-整理算法**"，运行过程和Serial收集器一样。



- ParNew

可以把这个收集器理解为Serial收集器的多线程版本。

```
优点：在多CPU时，比Serial效率高。
缺点：收集过程暂停所有应用程序线程，单CPU时比Serial效率差。
算法：复制算法
适用范围：新生代
应用：运行在Server模式下的虚拟机中首选的新生代收集器
```



- Parallel Scavenge

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去和ParNew一样，但是Parallel Scanvenge更关注系统的吞吐量。

> **吞吐量=运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)**
>
> **比如虚拟机总共运行了100分钟，垃圾收集时间用了1分钟，吞吐量=(100-1)/100=99%。**
>
> **若吞吐量越大，意味着垃圾收集的时间越短，则用户代码可以充分利用CPU资源，尽快完成程序的运算任务。**

```
-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。
```

- Parallel Old

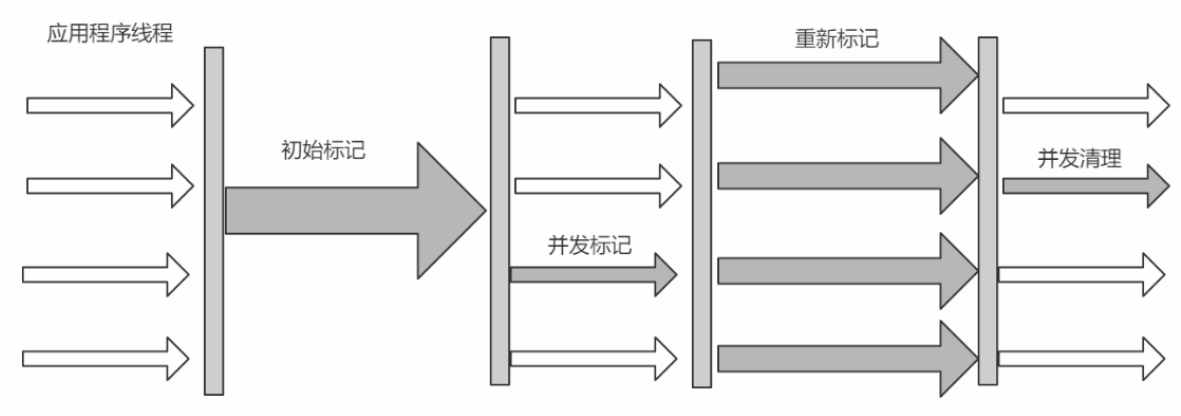
Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法进行垃圾回收，也是更加关注系统的**吞吐量。**

- CMS

> 官网：
> https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#concurrent_mark_sweep_cms_collector
>
> **CMS(Concurrent Mark Sweep)收集器是一种以获取** 最短回收停顿时间 **为目标的收集器。**
>
> **采用的是"标记-清除算法",整个过程分为4步**

```
(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
```

> **由于整个过程中，并发标记和并发清除，收集器线程可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的。**

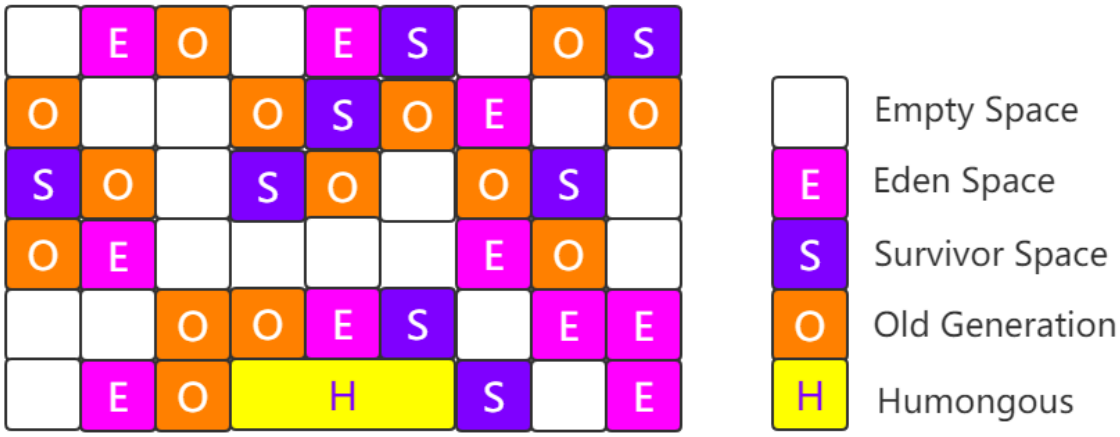



```
优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量
```

- G1(Garbage-First)

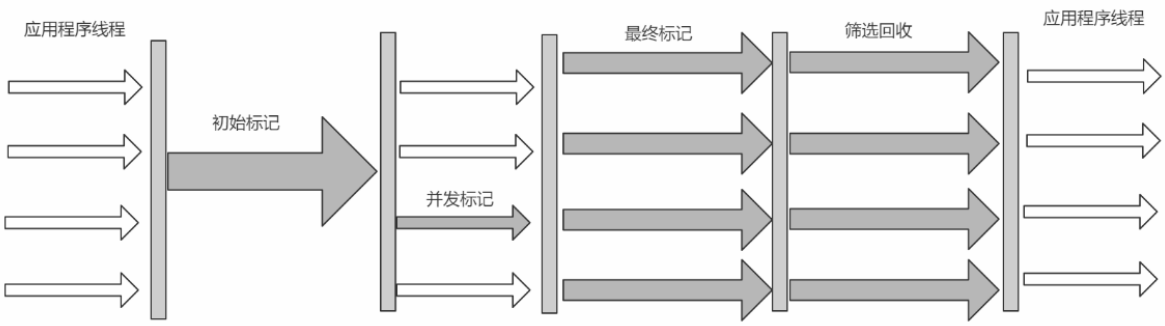
> 官网：
> https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html#garbage_first_garbage_collection
>
> **使用G1收集器时，Java堆的内存布局与就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。**
>
> **每个Region大小都是一样的，可以是1M到32M之间的数值，但是必须保证是2的n次幂**
>
> **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**

> （1）分代收集（仍然保留了分代的概念）
> （2）空间整合（整体上属于“标记-整理”算法，不会导致空间碎片）
> （3）可预测的停顿（比CMS更先进的地方在于能让使用者明确指定一个长度为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）



工作过程可以分为如下几步

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
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
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
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



- ZGC

> 官网： https://docs.oracle.com/en/java/javase/11/gctuning/z-garbage-collector1.html#GUID-A5A42691-095E-47BA-B6DC-FB4E5FAA43D0
>
> **JDK11新引入的ZGC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**

（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