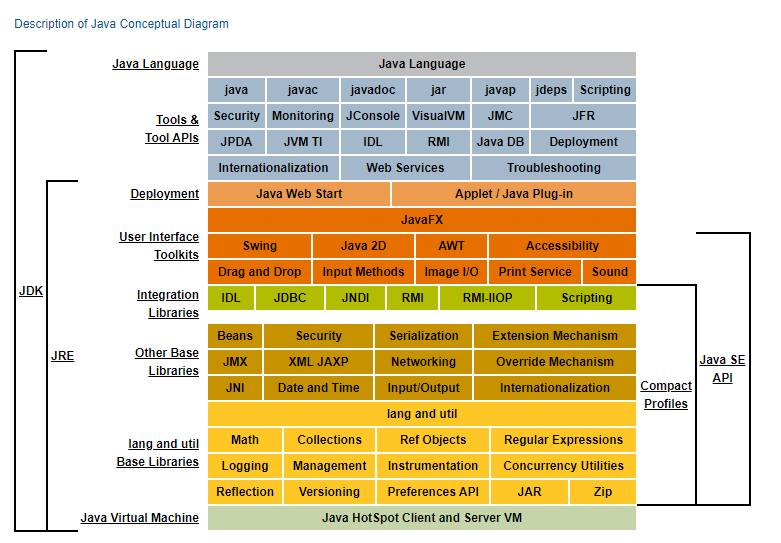
# JVM

## JDK，JRE，JVM之间的关系

JVM1.8的官网https://docs.oracle.com/javase/8/

JDK 8是JRE 8的超集，包含JRE 8中的所有内容，以及开发小程序和应用程序所需的工具，例如编译器和调试器。 JRE 8提供了库，Java虚拟机（JVM）和其他组件，以运行用Java编程语言编写的小程序和应用程序。 请注意，JRE包含Java SE规范不需要的组件，包括标准和非标准Java组件。

以下概念图说明了Oracle Java SE产品的组件：



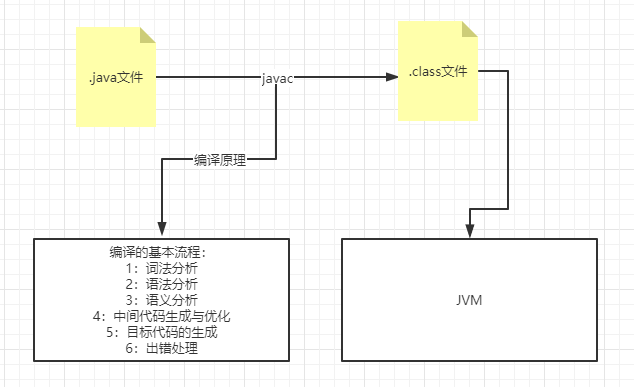
比如：java运行程序，javac可以编译，javadoc可以生成技术文档；从这张图上就能看出，JDK其实就是比JRE多了一层Tools&Tool APIs，而这一层中就包含了开发小程序和应用程序所需的工具，比如编译器，调试器等；

JDK多包含的那一层，可以这么去理解，JRE是我们的运行环境，如果我们的程序需要运行，则需要运行在拥有JRE的环境中，如果既要开发还要运行则需要JDK；

## 为什么需要学习JVM？

Java程序之所以能够一次编译到处运行，就是得益于JVM实现了与各大平台之间的连接，而我们只需要将编译好的class文件交给JVM即可，所以我们需要知道JVM到底是如何来加载和管理class文件的，这样有助于我们了解写的每一行Java代码背后的代码逻辑，从而提高我们对程序的把控性。

## 编译java文件宏观原理（大致了解即可）



编译可以分为五个基本步骤:词法分析、语法分析、语义分析 及中间代码的生成、优化、目标代码的生成。这是每个编译器都必须的基本步骤和流程, 从源头输入高级语言源程序输出目标语言代码。

### 1 词法分析

词法分析器是通过词法分析程序对构成源程序的字符串从左到右的扫描, 逐个字符地读, 识别出每个单词符号, 识别出的符号一般以二元式形式输出, 即包含符号种类的编码和该符号的值。词法分析器一般以函数的形式存在, 供语法分析器调用。当然也可以一个独立的词法分析器程序存在。完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。

### 2 语法分析

语法分析是编译过程的第二个阶段。这阶段的任务是在词法分析的基础上将识别出的单词符号序列组合成各类语法短语, 如“语句”, “表达式”等.语法分析程序的主要步骤是判断源程序语句是否符合定义的语法规则, 在语法结构上是否正确。而一个语法规则又称为文法, 乔姆斯基将文法根据施加不同的限制分为0型、1型、2型、3型文法, 0型文法又称短语文法, 1型称为上下文有关文法, 2型称为上下文无关文法, 3型文法称为正规文法, 限制条件依次递增。

### 3 语义分析

词法分析注重的是每个单词是否合法, 以及这个单词属于语言中的哪些部分。语法分析的上下文无关文法注重的是输入语句是否可以依据文法匹配产生式。那么, 语义分析就是要了解各个语法单位之间的关系是否合法。实际应用中就是对结构上正确的源程序进行上下文有关性质的审查, 进行类型审查等。

### 4 中间代码生成与优化

在进行了语法分析和语义分析阶段的工作之后, 有的编译程序将源程序变成一种内部表示形式, 这种内部表示形式叫做中间语言或中间表示或中间代码。所谓“中间代码”是一种结构简单、含义明确的记号系统, 这种记号系统复杂性介于源程序语言和机器语言之间, 容易将它翻译成目标代码。另外, 还可以在中间代码一级进行与机器无关的优化。

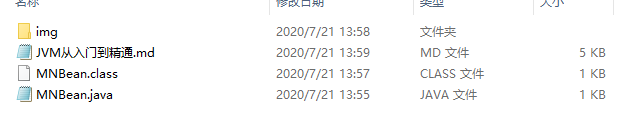
### 5 目标代码的生成

根据优化后的中间代码, 可生成有效的目标代码。而通常编译器将其翻译为汇编代码, 此时还需要将汇编代码经汇编器汇编为目标机器的机器语言。

### 6 出错处理

编译的各个阶段都有可能发现源码中的错误, 尤其是语法分析阶段可能会发现大量的错误, 因此编译器需要做出错处理, 报告错误类型及错误位置等信息。

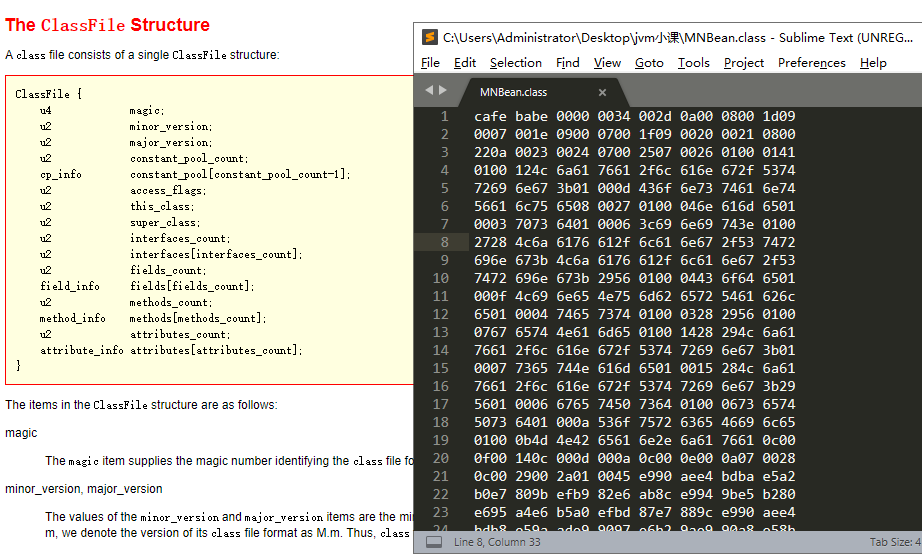
## 通过javac编译java文件

image-20200721135824277

如上两张图所示，通过javac命令将MNBean.java文件编译成MNBean.class文件。

## class文件解读

官网class file解读地址：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.htmljvms-4.1



上图为官方定义的ClassFile文件结构以及通过sublime打开MNBean.class文件（sublime支持16进制格式查看）

### 文件解读

1：u4代表8个16进制位。u4就相当于cafe babe；

​ u4后跟的magic代表：The `magic` item supplies the magic number identifying the `class` file format; it has the value `0xCAFEBABE`.（官网定义）

​ 它代表的是class文件的一个开头的格式；只要是class文件，都是以cafe babe开头。

2：u2代表 0000

3：第3个u2代表 0034

依此类推；（具体的表述可以参考下方代码块内容，大概了解即可，不需要熟记）

```java

struct ClassFile

{

u4 magic; //Class文件格式起始标识，具体值为0xCAFEBABE

u2 minor\_version; // 副版本号

u2 major\_version; // 主版本号

u2 constant\_pool\_count; // 常量池个数

cp\_info \*\*constant\_pool; // 常量池

u2 access\_flags; //Class的声明中使用的修饰符掩码

u2 this\_class; //常数表索引，索引内保存类名或接口名

u2 super\_class; //常数表索引，索引内保存父类名

u2 interfaces\_count; //超接口数

u2 \*interfaces; //常数表索引，各超接口名称

u2 fields\_count; //类的域数

field\_info \*\*fields; //域数据，包括属性名称索引

u2 methods\_count; //方法数

method\_info \*\*methods; //方法数据，包括方法名称索引，方法修饰符等

u2 attributes\_count; //类附加属性数

attribute\_info \*\*attributes; //类附加属性数据，包括源文件名等

};

```

## 类加载机制

官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html

### 装载

概念：Java虚拟机动态加载，链接和初始化类和接口。加载是查找具有特定名称的类或接口类型的二进制表示形式并从该二进制表示形式创建类或接口的过程。

装载就是以一个class文件派生出一个类的过程。

1）查找class文件：ClassLoader.loadClass()，根据类的全称进行加载，得到二进制流数据

2) 将二进制流数据进行读取：类的描述信息：时间，作者，版本等（保存在方法区）

3）存储class对象，比如：MNBean对象（存储在堆区）

\*\*注意：在加载过程中也是需要对class进行验证的，比如是否为class file结构，不是则引发ClassFormatError；还需要检查class file是否支持主要或者次要版本，不然则报UnsupportedClassVersionError错误；还包括一些其他验证，比如NoClassDefFoundError等；\*\*

### 链接

概念：链接是获取类或接口并将其组合到Java虚拟机的运行时状态以便可以执行的过程。主要过程包括：验证被加载类的正确性；准备工作；在链接之前，要确保类都已经完全加载；在初始化之前，也必须验证和做一些初始化的准备工作；由于链接涉及到分配新的数据结构，因此可能会失败，并引发OutOfMemoryError错误。

1）验证：确保类或者接口的二进制结构是正确的，以确保字节流包含的信息对虚拟机来说是安全的。

2）准备：准备工作包括为类或接口创建静态字段，并将这些字段初始化为其默认值。这不需要执行任何Java虚拟机代码。静态字段的显式初始化程序是初始化的一部分，而不是准备工作。

比如：

```java

public static int FLAG = 1; // 在准备阶段，在方法去将这个FLAG值初始化为0，而不是1；

```

**3）解析：将类的符号引用转换为直接引用**

符号引用：对于class文件来说，里面的数据都是符号，这些符号jvm虽然认识，但符号从某种意义上来说是无意义的，只是某种约定而已；（这是class文件中的）

直接引用：class被加载后，如果需要运行，则需要去操作具体的比如某个对象的内存地址，这时候就需要将符号引用替换为指向某个对象内存地址的指针，这个指针就是直接引用；（这是jvm内存空间的）

### 初始化

概念：类或接口的初始化包括执行类或接口的初始化方法。

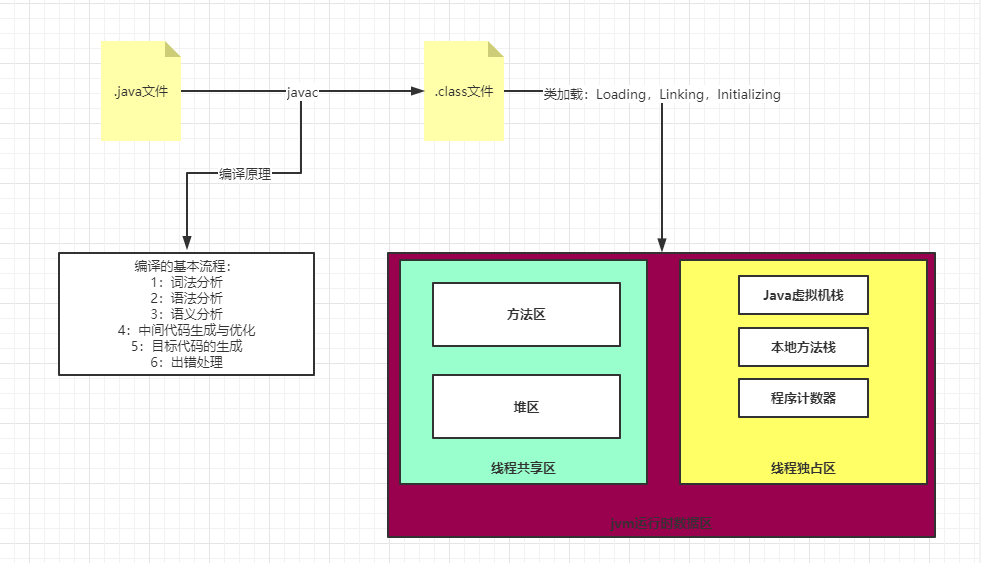
1）对类的静态变量，代码块进行初始化操作；

比如：

```java

public static int FLAG = 1; // 在准备阶段，在方法去将这个FLAG值初始化为0；在初始化阶段就是将FLAG设置为1；

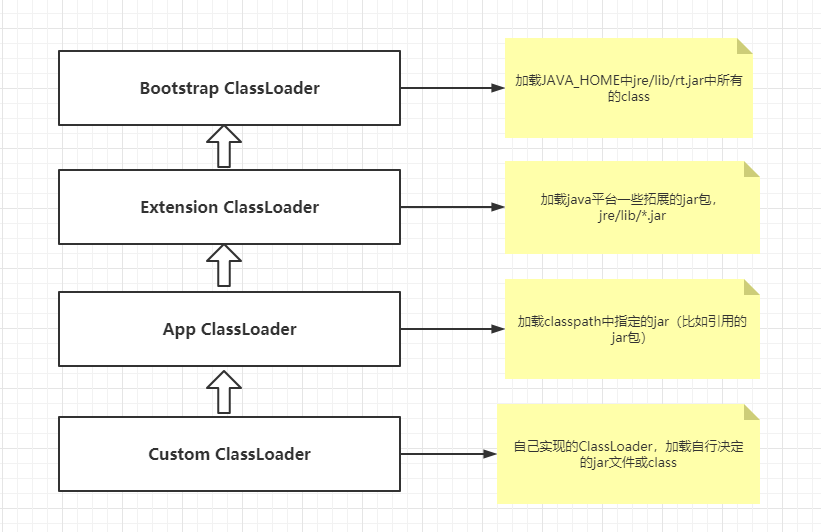
```



如上图，java文件编译成class文件，jvm通过加载类（Loading，Linking，Initializing）将类等信息加载进jvm的不同区域，而整体关于这几个区域可以统称为jvm运行时数据区。

## ClassLoader的双亲委派模型

Java虚拟机提供的引导类加载器和用户定义的类加载器。每个用户定义的类加载器都是抽象类ClassLoader的子类的实例。 应用程序使用用户定义的类加载器，以扩展Java虚拟机动态加载创建类的方式。 用户定义的类加载器可用于创建源自用户定义的源的类。 例如，可以通过网络下载类，动态生成类或从加密文件中提取类。

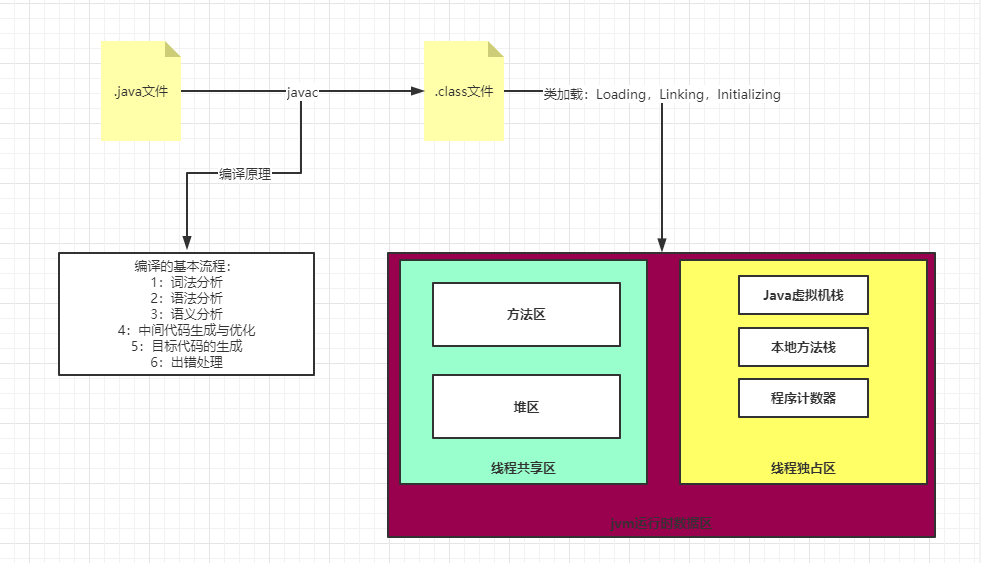


所谓的委派方式，可以非常简单的理解为，为了让jvm在加载class的时候不出现重复加载，当要加载一个类的时候，会先交给其“父”ClassLoader进行加载，如果其“父”ClassLoader可以加载，则自己不加载，如果加载不了再交给下一级加载。

## JVM运行时数据区

官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.htmljvms-2.5

当class文件被jvm加载之后，则进入运行阶段，下图为jvm运行时数据区



注意：在官网中，运行时数据区分为6个部分；比上图中多了一个Run-Time Constant Pool区域，但这个区域也是属于方法区的一部分；所以目前大部分都主要分为5个部分。

### 方法区

每一个jvm虚拟机只有一个方法区，生命周期与jvm一致，能够被所有线程共享；存储类信息，静态变量(static)，常量(final)；

#### 运行时常量池（Run-Time Constant Pool）

运行时常量池是类文件中constant\_pool表的按类或按接口的运行时表示。它包含常量，从编译时已知的数字文字到必须在运行时解析的方法和字段引用。每个运行时常量池都是从Java虚拟机的方法区分配的。当Java虚拟机创建类或接口时，将为类或接口构造运行时常量池。

### 堆区

每一个jvm虚拟机只有一个堆区，生命周期与jvm一致，能够被所有线程共享；存储对象和数组；

### Java虚拟机栈

每个Java虚拟机线程都有一个私有的Java虚拟机栈，在线程创建时创建；Java虚拟机栈存储栈帧

### 本地方法栈

每个Java虚拟机线程都有可能存在一个私有的本地方法栈，如果线程没有调用本地方法（native,c++之类的）则可能不存在本地方法栈；

### 程序计数器

线程在运行时都会抢CPU的资源进行运行，如果线程在运行时来回切换，为了保证切换回来后知道执行在哪个位置，需要程序计数器来记住当前执行的位置；

## Java虚拟机栈（栈帧）详解

官网地址：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.htmljvms-2.6

**一个线程就会创建一个虚拟机栈，而线程中每运行的一个方法就对应一个栈帧。栈帧用于存储数据和部分结果，以及执行动态链接，方法的返回值和调度异常。**

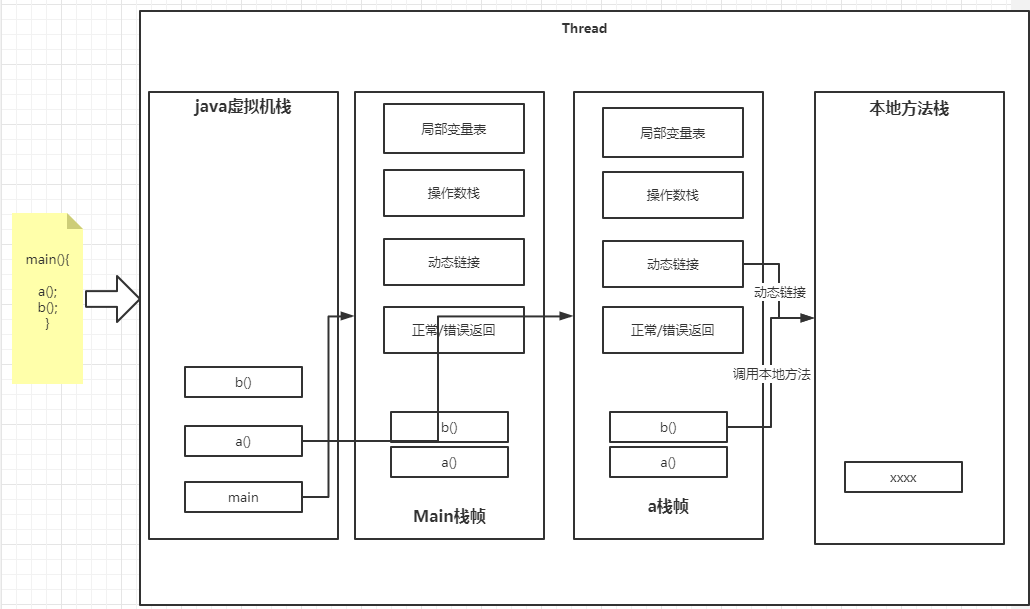
每次调用方法时都会创建一个新栈帧。当栈帧的方法调用完成时，无论结果是正常的还是错误（比如引发未捕获的异常），栈帧都会被销毁。栈帧是从创建栈帧的线程的Java虚拟机堆栈中分配的。每个帧都有其自己的局部变量数组，自己的操作数堆栈，以及对当前方法类的运行时常量池的引用 。

局部变量数组和操作数堆栈的大小在编译时确定。因此，帧数据结构的大小仅取决于Java虚拟机的实现，并且可以在方法调用时分配用于这些结构的内存。

在给定的控制线程中，只有一个栈帧（用于执行方法的帧）处于活动状态。该帧称为当前帧，其方法称为当前方法。定义当前方法的类是当前类。局部变量和操作数堆栈上的操作通常参考当前帧。

如果栈帧的方法调用另一个方法或该栈帧的方法完成，则该栈帧将不再是当前栈帧。调用方法时，将创建新栈帧，并在控制权转移到新方法时变为新栈帧。在方法返回时，当前帧将其方法调用的结果（如果有的话）传回上一帧。然后，当前一帧变为当前帧时，将丢弃当前帧。

请注意，由线程创建的框架在该线程本地，并且不能被任何其他线程引用。



上图为线程/Java虚拟机栈/栈帧/本地方法栈之间的关系（上图画的正常/错误返回，也有人称之为：返回地址）

## 栈帧(看图理解)

一个方法就是一个栈帧，看看如何执行的；

局部变量表：就是存储局部变量；

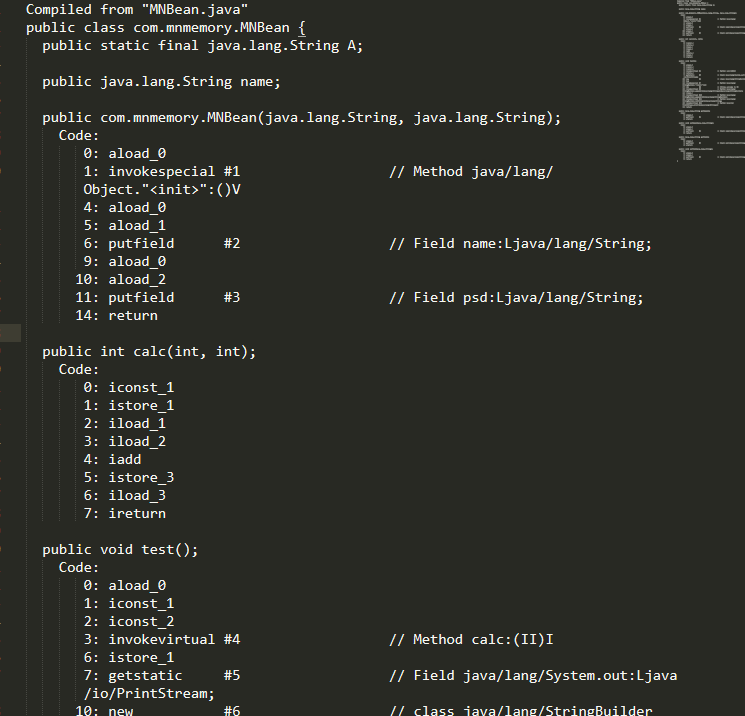
操作数栈：栈的结构【先进后出】，用来存储操作数；

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，根据需要加载类以解析尚未定义的符号，并将变量访问转换为与这些变量的运行时位置关联的存储结构中的适当偏移量。

方法返回地址：记录一个方法执行完成之后返回的地方；

先看一下class文件被装载到jvm中后的执行流程（字节码指令：jvm认识的方式）；

先通过命令行：javap -c .\MNBean.class > MNBean.txt 得到字节码文件；



开始解读：

```java

public int calc(int, int); // 通过calc方法进行解读

Code:

0: iconst\_1 // 1（int）值入栈

1: istore\_1 // 将栈顶int类型值保存到局部变量1中

2: iload\_1 // 从局部变量1中装载int类型值入栈

3: iload\_2 // 从局部变量2中装载int类型值入栈

4: iadd // 将栈顶两int类型数相加，结果入栈

5: istore\_3 // 将栈顶int类型值保存到局部变量3中

6: iload\_3 // 从局部变量3中装载int类型值入栈

7: ireturn // 返回int类型值

```

指令集合解读博客：https://www.cnblogs.com/longjee/p/8675771.html

从上述字节码解读，我们可以得出一个结论，平常我们写的代码比如a=1，看上去这是一行代码，但成为字节码之后会变成两行，而线程在执行的时候是会进行交替处理的，也就代表着，就这么简单的一行代码都有可能出问题。而你得知道为什么会出现这样的问题；

栈指向堆是什么意思？

```java

public static final String A = "MNBEAN";

public String name;

private String psd;

private Object object = new Object();

public MNBean(String name, String psd) {

this.name = name;

this.psd = psd;

}

public int calc(int a,int b){

// 局部变量引用全局变量

Object o = this.object;

a = 1;

int c = a+b;

return c;

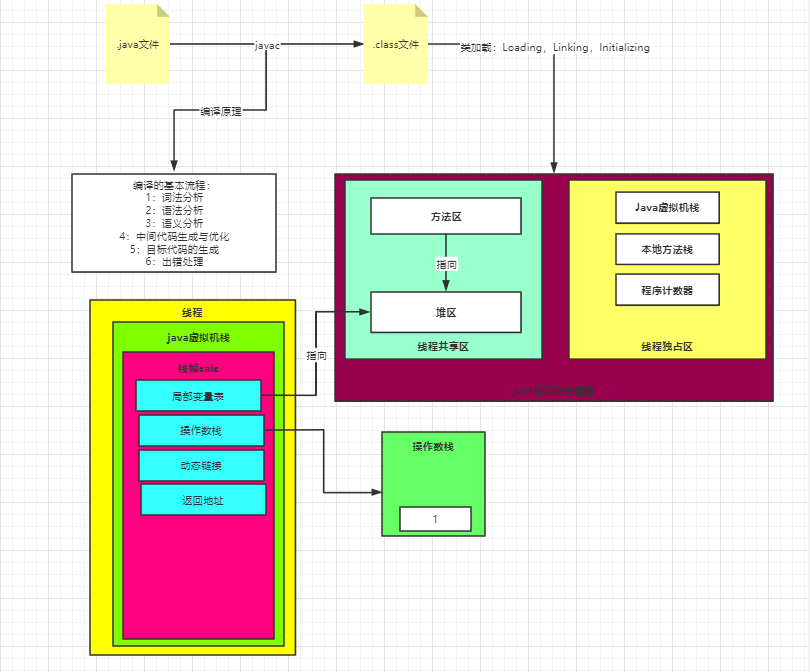
}

```

当局部变量引用了全局变量在内存中会有什么样的体验？

这代表着栈帧中局部变量表是可能指向堆区的，这就是为什么有人说变量是保存在栈中，而实际的实例是保存在堆中；

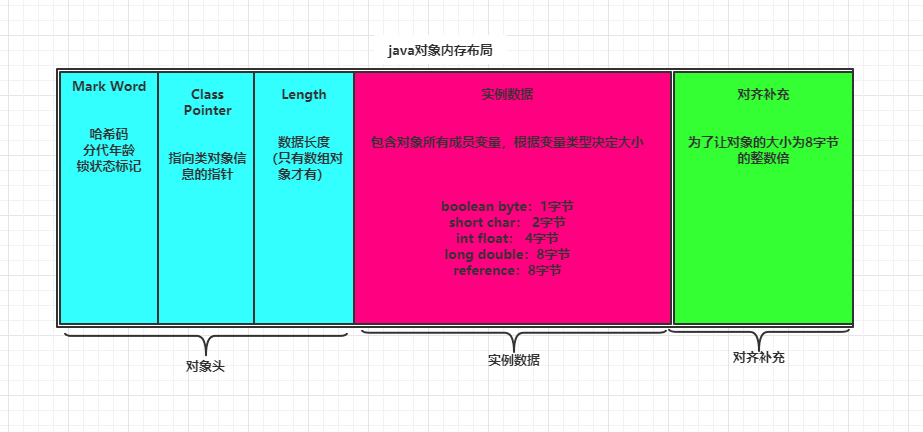
同样，由于方法区中保存了static和常量，那么他们也有可能指向堆区；如下图；



那么现在问题来了，如果说static和final的那些变量的实例都是存放在堆中，那么非static和final呢？又是存在哪里？答案：还是存在堆中，只是看这个类的实例什么时候初始化到堆中；

## 对象在内存中如何存在？

堆也可以指向方法区；对象在内存中如何存在；new MNBean(),这个MNBean对象有多大？由哪个类创建的？



上图是在64位系统中

class pointer: 这个对象到底属于方法区中的哪个类信息？就是在这里记住的；也是在这里指向的；

public class A{

private int a =1; // 4个字节

private String b = "aaa"; // 8个字节,String是引用类型不要搞错了,”aaa”是在堆中

private A aa = new A(); // 8个字节

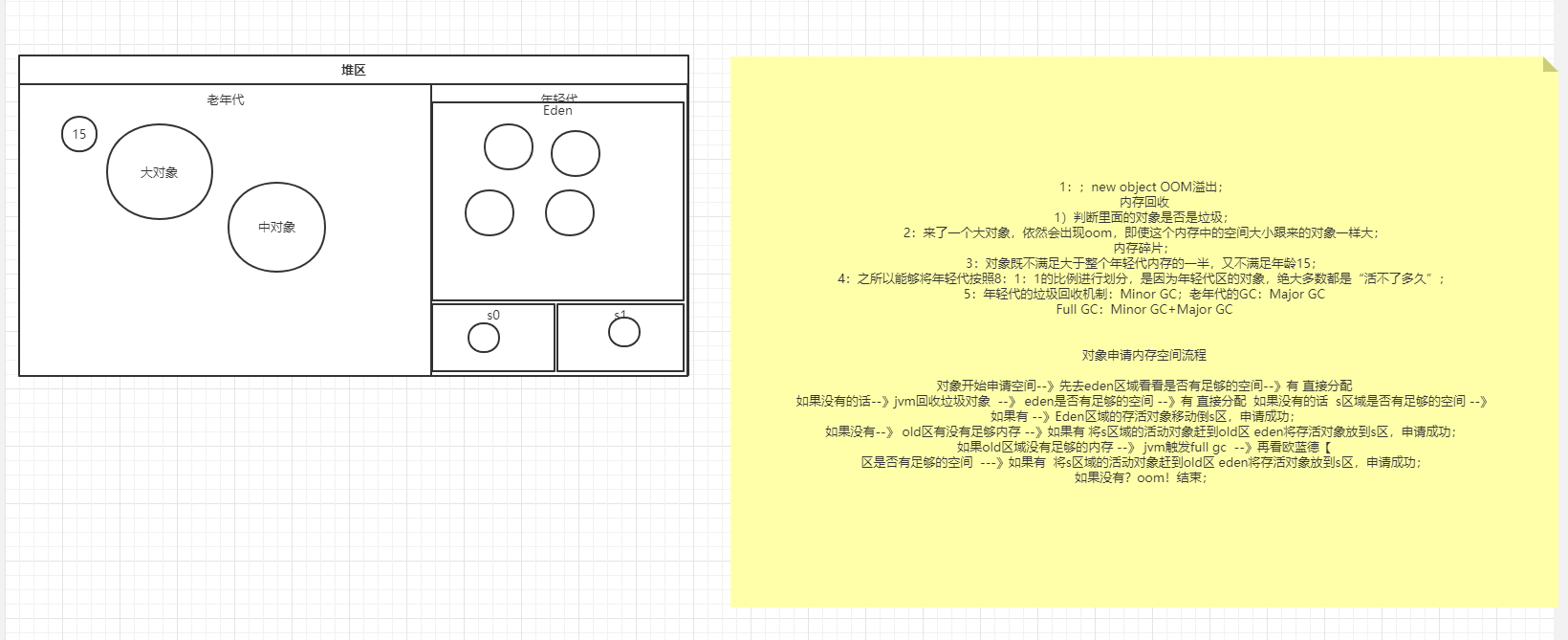
//对齐补齐

// 对象头 8字节

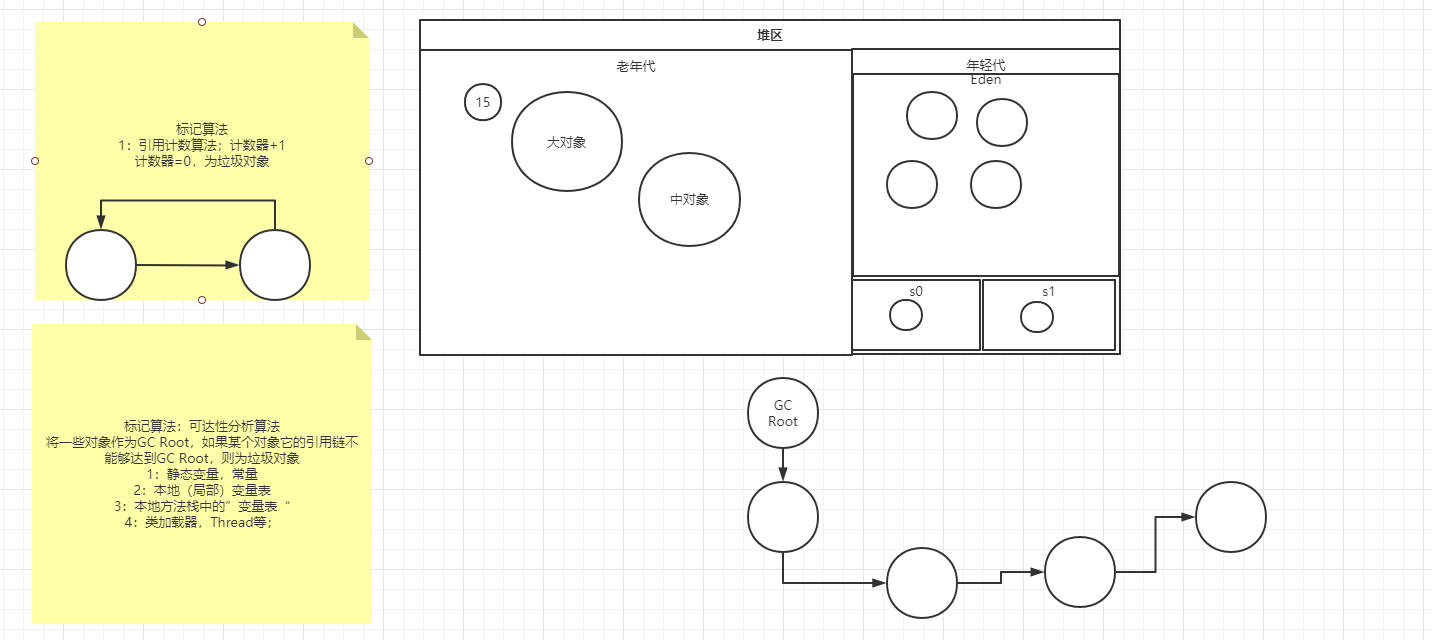
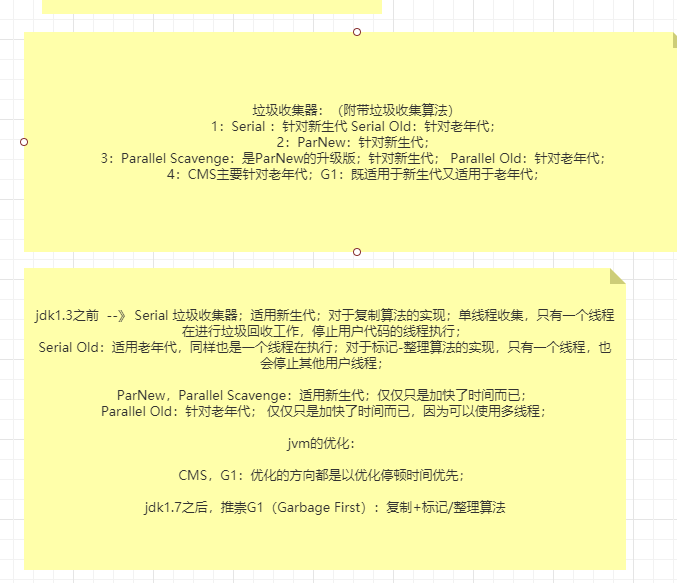
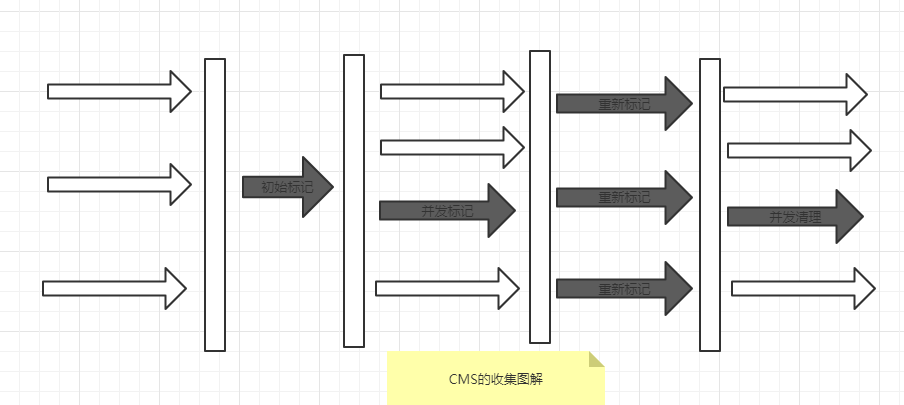
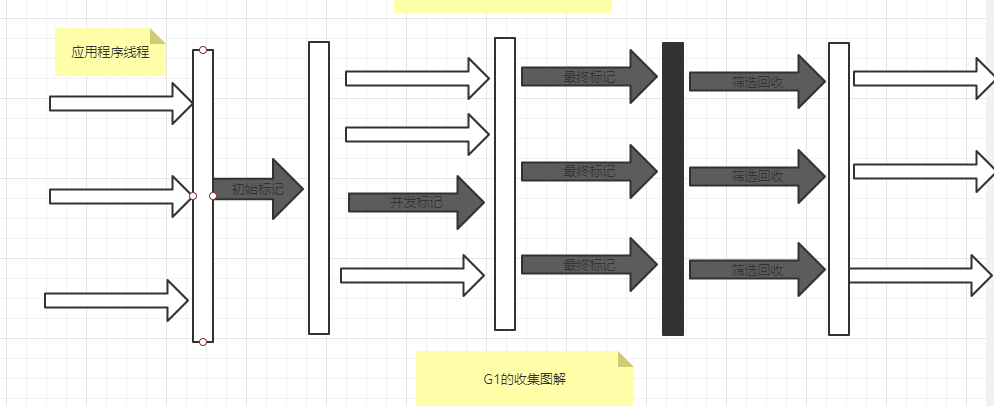
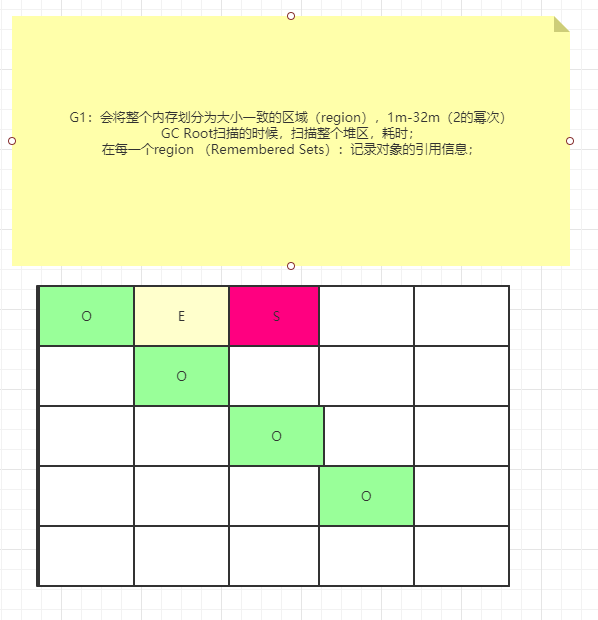
// 内存条 一个一个电子器件组成；一个电子器件就代表1位；

// private byte[] bytes = new byte[1024\*1024]; //这个造的数组太大了

}



上图为 java对象申请内存空间流程



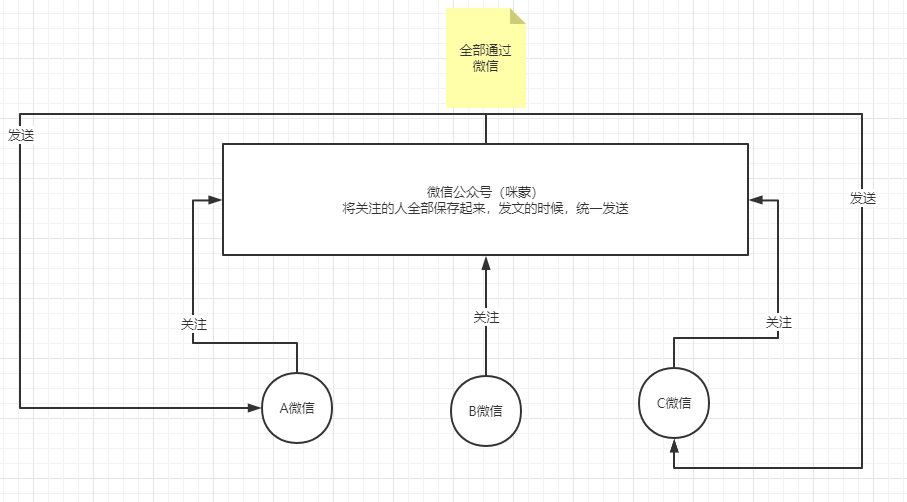
垃圾收集器分类：

串行收集器：只有一个垃圾回收线程，暂停用户代码；Serial，Serial Old

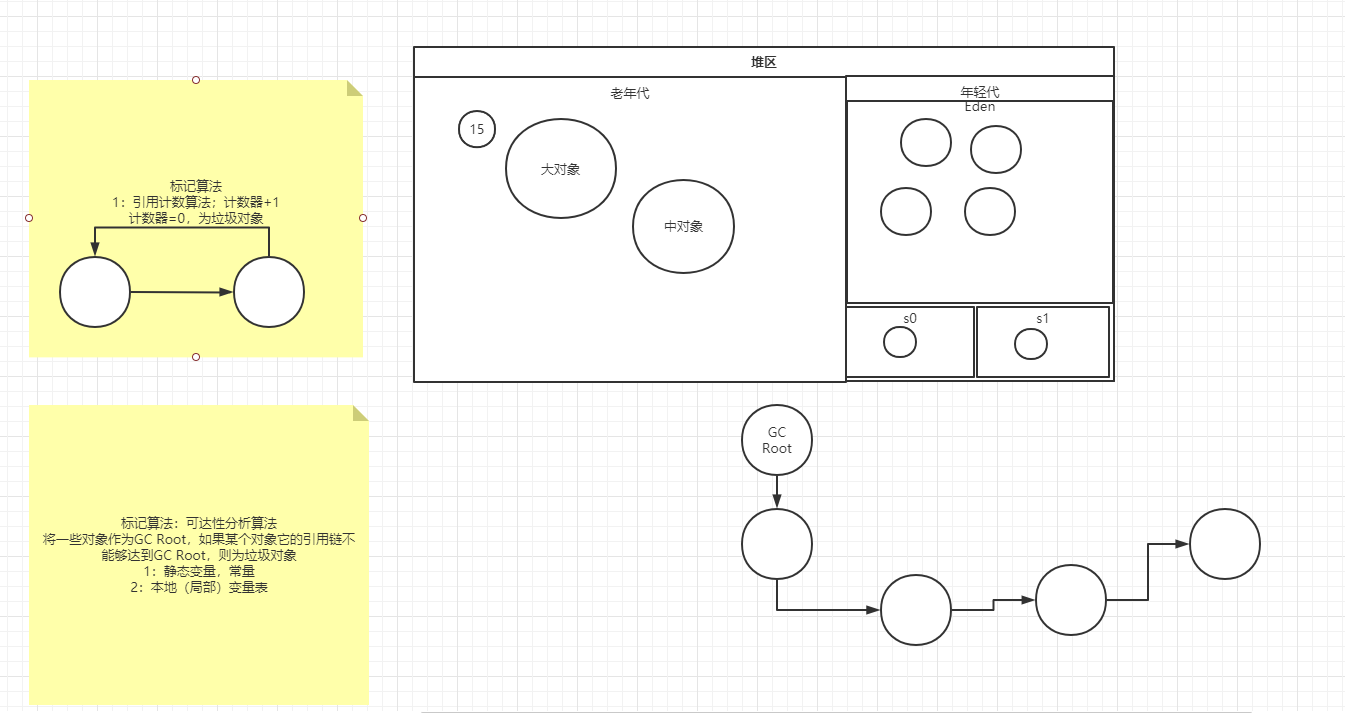
并行收集器：[它的关注点在于吞吐量]，多个垃圾收集器线程同时工作；暂停用户代码；Parallel Scanvenge，Parallel Old；

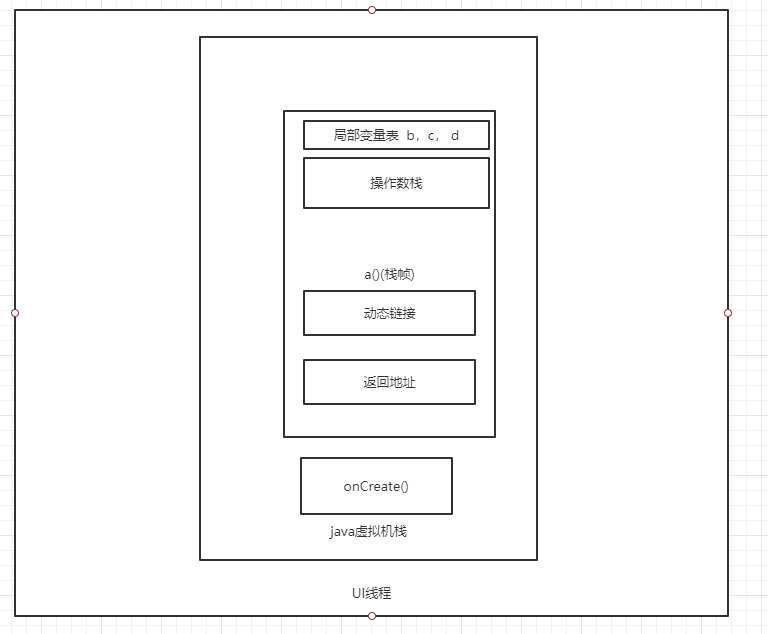
并发收集器：[关注点：时间优先]CMS，G1，完全停止用户代码线程；（停顿时间非常短，如果处于标记阶段，也会有知识暂停线程的可能）

# 2020-9-14 vip上提到的图片如下：



上图是解释了观察者模式





public class MainActivityTemp extends AppCompatActivity {

public static int Flag = 1;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

a(); //UI线程

}

public void a() {

//局部变量 //操作数栈1

int b = 1;

int c = 2;

int d = b + c;

//动态链接,其实就是一个类的加载过程(中间有一个动态连接的过程)

//a是存在栈帧的局部变量中,而这个对象是存在共享的堆中的

A a = new A();

//返回地址

int k = c();

}

private int c() {

return 0;

}

public class A {

private int a = 1;

private String b = "aaaaaaaaaa";

private void a() {

}

}

}

# 思考:内存需要关注哪些问题

关注内存3个主要点  
 1：代码在jvm中到底是怎么存在的（栈帧到底什么关系）  
 2：某个对象在内存中到底占用多少内存  
 3：某个对象的生命周期；线程，application