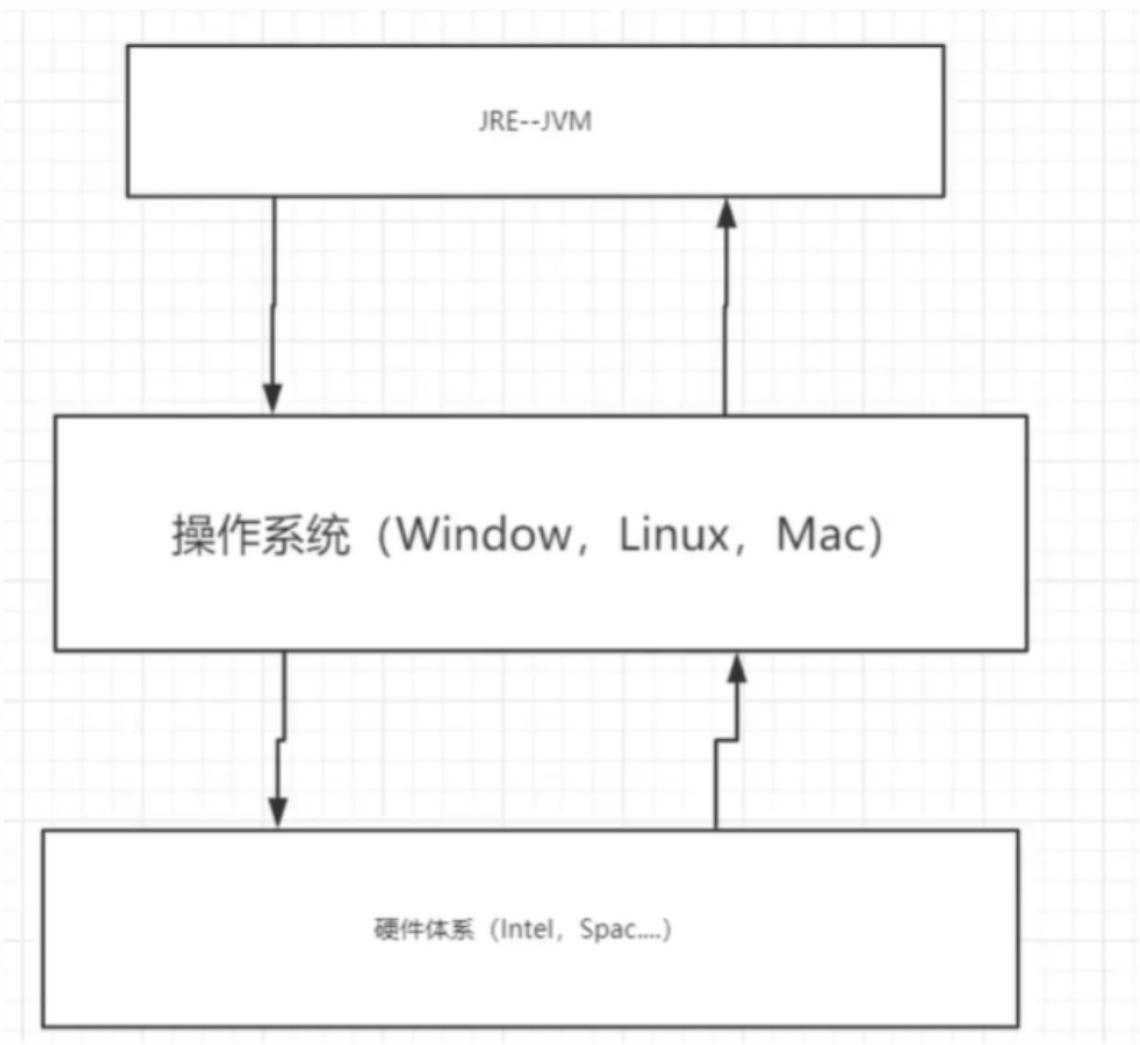
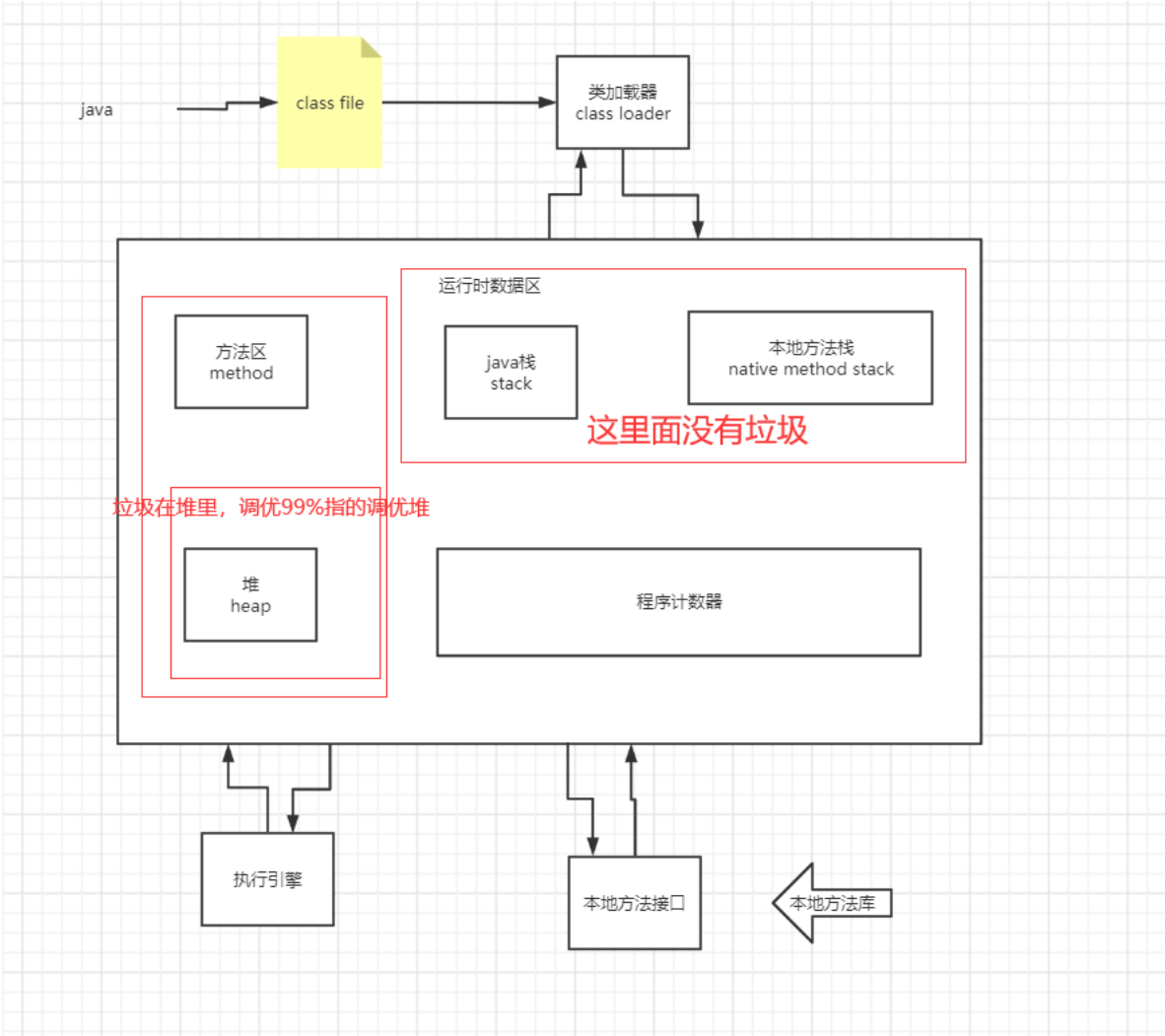
JVM

1.jvm的位置



2.jvm的体系结构



3、类加载器

作用:加载Class文件

- 虚拟机自带的加载器
- 启动类（根)加载器　boot　ext的父加载器，无法获取到，因为是c、c++编写的
- 扩展类加载器　ext　　// ExtClassLoader　\jre\lib\ext
- 应用程序（系统）类加载器 app　//AppClassLoader

双亲委派机制：

先从app->ext->boot 查找，找到了之后还要继续往后，最后执行高一等级的类

1. 类加载器收到类加载的请求，将这个请求向上委托给父类加载器去完成，一直向上委托，直到启动类加载器，
2. 启动加载器检查是否能够加载当前这个类，能加载就结束，使用当前的加载器，否则抛出异常，通过子加载器加载

> 自己的理解

就是我们加载一个类，首先先去查找当前这个类的加载器有没有加载这个类，如果没有的话就委托当前类加载的器的父加载器去find一下有没有加载这个类，然后一直委托到根加载器(BootstrapClassLoader)。中间如果有任何一个加载器已经加载过这个类了，就直接返回。

如果都没加载过这个类，那么就从根加载器开始看能不能加载这个类，如果根加载器加载不了，就让下一级的加载器加载，持续这个过程一直到应用程序类加载器加载。如果都加载不了，就是CLass not find了。

参考博客:**https://www.cnblogs.com/ITPower/p/13205903.html**

> 为什么要有双亲委派机制

```
#两个原因:
#1. 沙箱安全机制，自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意修改
#2. 避免类重复加载. 比如之前说的，在AppClassLoader里面有java/jre/lib包下的类，他会加载么？不会，他会让上面的类加载器加载，当上面的类加载器加载以后，就直接返回了，避免了重复加载.
```

4、native关键字

凡是带了native关键字的，说明java的作用范围达不到了，会去调用底层C语言的库，会进入本地方法栈调用本地方法接口 JNI：扩展java的使用，融合不同的编程语言为java所用

private native void start0();

> 沙箱安全机制

查一下

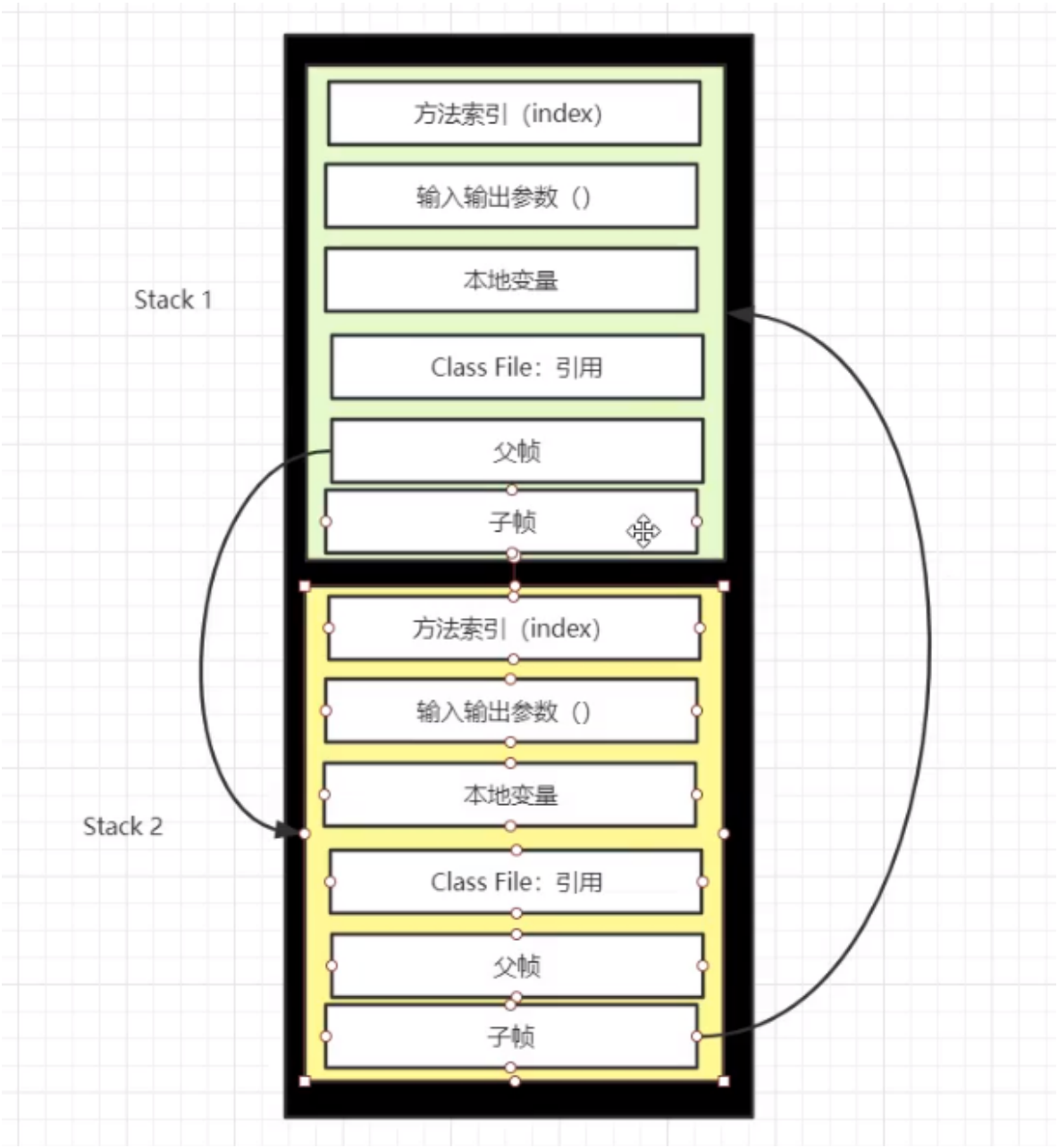
> 方法区存了哪些东西

5、栈

线程结束，栈内存也就释放了，对于栈来说，不存在垃圾回收问题

> 栈里面存的东西

8大基本类型，对象引用，实例的方法



6、堆

> 三种jvm

- sun公司的　-- hotsopt（我们用的）
- Oracle -- JRockit
- IBM　-- j9VM

我们学习的都是HotSpot

heap，一个JVM只有一个堆内存，堆内存的大小是可以调解的

存放了 类，方法，常量，变量和所有引用类型的真实对象

堆内存细分为三个区域：

- 新生区(伊甸园区)
  - 伊甸园
  - 幸存0区
  - 幸存1区

- 养老区
- 永久区（元空间，方法区）

GC垃圾回收，主要在伊甸园和养老区

OOM错误，堆内存满了

jdk8以后，永久区改了名字-元空间

> 新生区

- 类诞生和成长的地方，甚至死亡。
- 伊甸园：所有的对象都是在这new出来的 (对象太大的话会在老年区那出来)
- 幸存区（0，1）：

> 永久区

- jdk1.6之前：永久代，常量池在方法区
- jdk1.7: 永久代，但是慢慢退化了，去永久代，常量池在堆中
- jdk1.8之后：无永久代，常量池在元空间

>

默认情况下，分配的总内存是电脑内存的1/4，而初始化的内存是：1/64

```
-Xms1024m -Xmx1024m -XX:+PrintGCDetails

public class test {
    public static void main(String[] args) {
        long max = Runtime.getRuntime().maxMemory();
        long total = Runtime.getRuntime().totalMemory();

        System.out.println("max="+max+"字节"+(max/(double)1024/1024)+"M");
        System.out.println("total="+max+"字节"+(total/(double)1024/1024)+"M");
    }
}
```

```
max=1029177344字节981.5M
total=1029177344字节981.5M
Heap
 PSYoungGen      total 305664K, used 20971K [0x00000000eab00000, 0x0000000100000000, 0x0000000100000000)
  eden space 262144K, 8% used [0x00000000eab00000,0x00000000ebf7afb8,0x00000000fab00000)
  from space 43520K, 0% used [0x00000000fd580000,0x00000000fd580000,0x0000000100000000)
  to   space 43520K, 0% used [0x00000000fab00000,0x00000000fab00000,0x00000000fd580000)
 ParOldGen       total 699392K, used 0K [0x00000000c0000000, 0x00000000eab00000, 0x00000000eab00000)
  object space 699392K, 0% used [0x00000000c0000000,0x00000000c0000000,0x00000000eab00000)
 Metaspace       used 3502K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
```

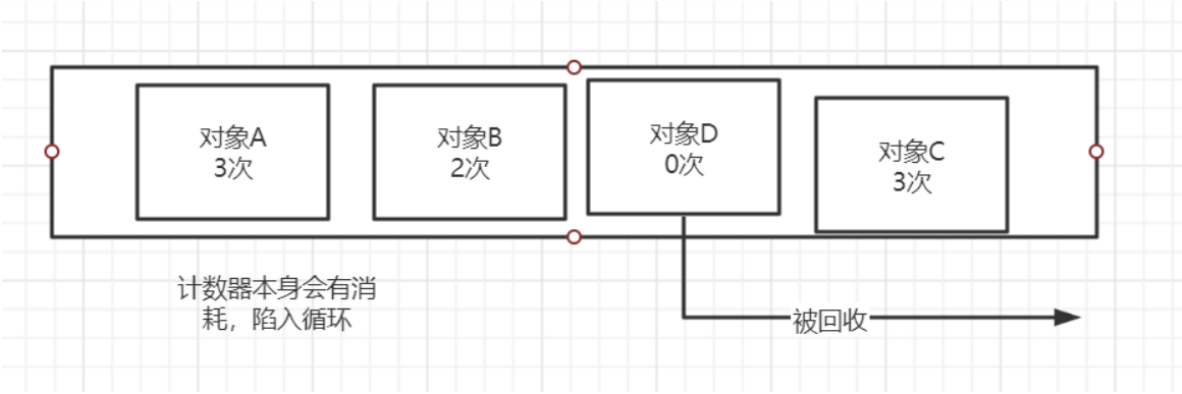
这里这两个相加得到的大小就是981.5M，元空间并不在这里

7、GC垃圾回收方法

新生区，幸存区from，幸存区to，老年区

- 轻GC（普通的GC）　新生区，偶尔在幸存区
- 重GC（全局GC）　　老年区

> 引用计数法



成本：

- 计数器
- 次数为0的就被清理了

复制算法

from 和to 区，**谁空谁是to**，然后伊甸园区和from区的垃圾就去往to区。当一个对象经历了15次(默认值)GC都没有死，就去了养老区

- 好处：没有内存的碎片
- 坏处：浪费了内存空间(幸存区有一个为空)

复制算法最佳使用场景：对象存活度较低的时候：新生区

标记清除法

两次扫描：第一次标记存活的对象，第二次清楚没有标记的对象

可达性分析算法

缺点：

- 两次扫描严重浪费空间
- 会产生内存碎片

优点:

- 不需要额外的空间

标记压缩

在标记清楚方法上进行优化：

- 压缩
  - 防止内存碎片的产生
  - 再次扫描，然后把对象整理一下，有了移动的成本

全程应该为标记清除压缩算法

小优化：先标记清除几次，之后再压缩一次

总结

内存效率：复制算法>标记清除算法>标记压缩(时间复杂度)

内存整齐度：复制算法=标记压缩>标记清除算法

内存利用率：标记压缩 = 标记清除算法>复制算法

GC ->分代收集算法

年轻代：

- 存活率低
- 复制算法

老年代：

- 存活率高，区域大
- 标记清除+标记压缩 混合实现

JMM

java memory model

它是干嘛的？ 官方，博客