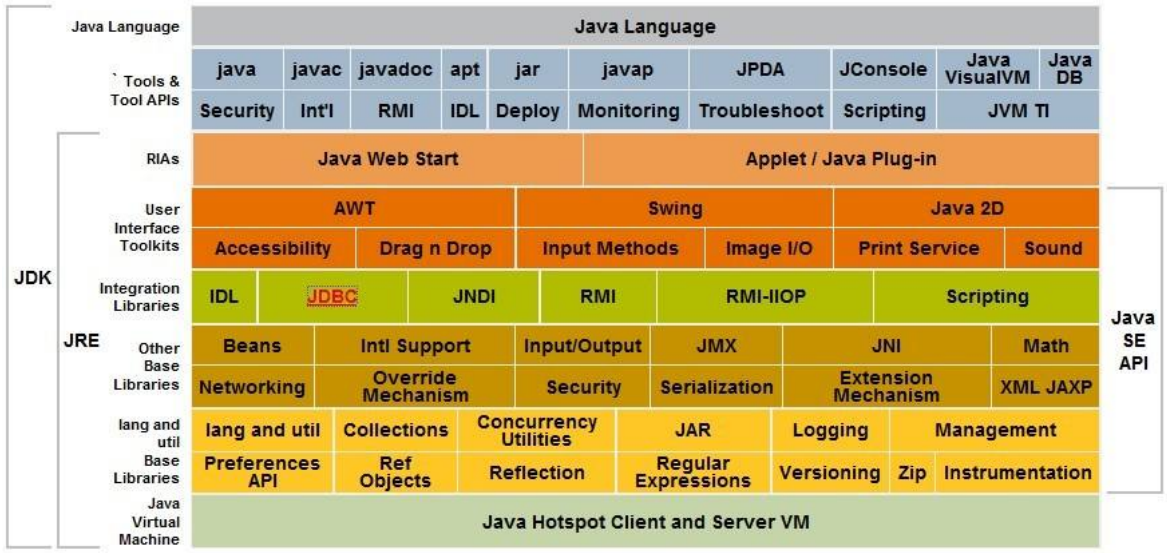
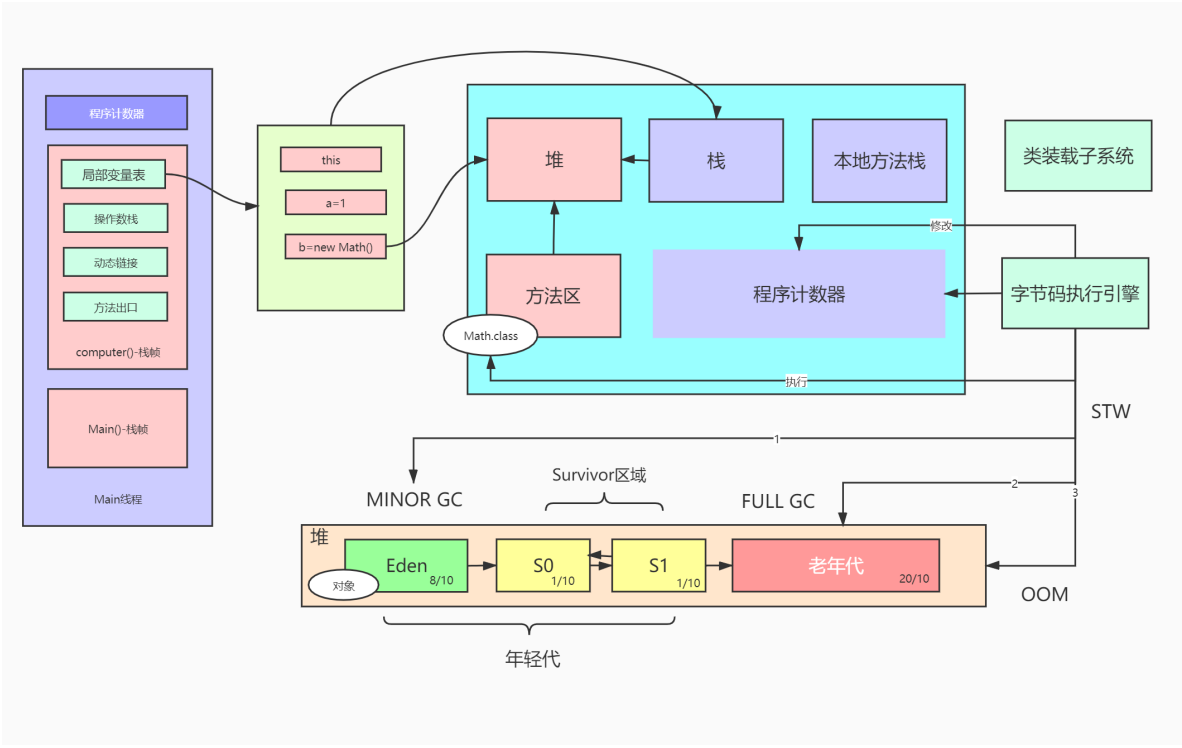
JAVA体系结构图





JAVA内存模型解析

- 栈帧：一个方法开始运行，就会分配一块独立的内存空间，该空间称为栈帧，方法一执行完，该空间将会被释放掉
- 程序计数器：用来记录程序当前运行的代码位置，CPU再次调度该线程时，知道从哪继续执行
- 动态链接：方法代码在内存空间的地址，比如调用computer，computer()就是指向它的方法区地址
- 方法出口：方法执行完回到调用该方法的位置，如执行完computer()后回到main方法的指定行
- 堆：堆中存着复杂对象的数据
- 栈：栈中存着基本类型的局部变量和复杂对象的数据
- 方法区：存放着常量，静态变量，类信息

```
1.8之前，叫做元空间
1.8之后，使用的是直接内存，也就是底层内存，而不是虚拟机内存
```

- 本地方法栈：java前期对于C,C++做出的兼容操作，调用底层的dll库，现在用的很少了

```
# Thread.java
private native void start0();
```

Minor GC

- 触发时机：**当Eden区的对象存放满的时候**
- GCRoot：一个对象引用的最终的对象，引用链条的最顶端，当GCRoot不再被使用时，该链条的所有对象都将成为垃圾对象
- 过程：每一次GC，都将对Eden区和Survivor的对象进行GCRoot引用判断，如果无引用将会释放，如果有引用，将Eden区的对象移动到Survivor区域，或者将Survivor区域的分代年龄加1，**当对象的分代年龄达到15的时候，或者本次minor gc的对象总大小超过survivor大小的50%将进入老年区**，如IOC容器中的bean，静态引用对象，数据库连接池对象等一般都会进入老年区，而局部变量可能还没进入老年区就被释放了
    1. 分代年龄达到15
    2. 本次minor gc的对象总大小超过survivor大小的50%
    3. 老年区担保机制
- 验证：jvisualvm
- 当老年代放满的时候，会进行full gc

FULL GC

- 触发时机：**当老年区的对象存放满的时候，java线程觉得自己还能再抢救一下**
- 触发堆内所有对象的GC操作

OOM

- 触发时机：**老年区满了，但是还再往里面塞对象**
- 挂掉

STW

- 触发时机：每一次GC开始的时候
- 停掉用户线程
- 问题：为什么在gc的时候要停掉用户线程?

  如果gc过程中，方法执行完了，栈空间中的所有对象引用都会被释放，这样的话之前gc标记非垃圾的对象将会缺少引用，这样这样又要重新做一遍gc，反而影响性能

对象头

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量(重量级锁)的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	分代年龄	1	01

字节码文件解析

```
public class Basic {
    public static final int initData = 666;
    private int computer(){
        int i = 1;
        int j =2;
        return i+j;
    }
    @Test
    public void test(){
        int computer = computer();
        System.out.println(computer);
    }
}
```

```
# basic.class  字节码文件
cafe babe 0000 0034 002c 0a00 0900 2509
0003 0026 0700 270a 0003 0025 0a00 0300
....
```

```
# javap -c Basic.class jvm指令码
Compiled from "Basic.java"
public class Basic {
  public static final int initData;

  public Basic();
    Code:
       0: aload_0       # 从局部变量0中装载引用类型值入栈。
       1: invokespecial # 编译时方法绑定调用方法。                =
       4: return        # void函数返回。

  public void test();
    Code:
       0: aload_0       # 从局部变量0中装载引用类型值入栈。
       1: invokespecial # 编译时方法绑定调用方法。
       4: istore_1      # 将栈顶int类型值保存到局部变量1中。
       5: getstatic     # 获取静态字段的值。
       8: iload_1       # 从局部变量1中装载int类型值入栈。
       9: invokevirtual # 运行时方法绑定调用方法。
      12: return        # void函数返回。
}
```

JVM参数

```
# 内存分配相关
-Xmx 堆的大小上限
-Xms 堆区内存初始内存分配的大小
-XX:MaxPermSize 永久代上限
-XX:SurvivorRatio Eden与Survivor区的比例
-XX:+/-UseTLAB 是否使用TLAB来创建对象
-XX:PretenureSizeThreshold 晋升老年代对象大小
-XX:NewRatio 新生代（Eden+2S）和老年代的比值，4表示1：4
```

```
# gc相关
-XX:+UseConcMarkSwerrpGC 指定老年代的GC收集器为CMS
-XX:+UseParNewGC 指定新生代GC收集器为ParNew
-XX:ParallelGCThreads 限制GC收集器的线程数
-XX:MaxGCPauseMillis 设定GC最大停顿时间（以牺牲吞吐量为代价）
-XX:GCTimeRatio 设定GC吞吐量，计算方式为吞吐量=1/(1+n)，n为设定的值
-XX:CMSInitiatingOccupancyFraction 用于设置触发GC的百分比，在jdk 1.6中，这个值时92%
```

```
# 异常相关
-XX:+HeapDumpOnOutOfMemoryError 在出现OOM异常之后自动生成heapdump文件
-XX:+HeapDumpOnCtrlBreak 使用Ctrl+Break键可以让虚拟机生成heapdump文件bash
```

JVM调优实战

- 高并发情况下的频繁full gc问题，就是因为一次生成的对象过多，直接提到了老年代，这样就只能full gc才能解决

  > 在进行性能调优的情况下，可以先根据用户量估计一下产生对象的量和所占用的内存空间，并且考虑转换的垃圾对象的时间

  问题：为什么会转换成垃圾对象还会出现堆内存打满的情况

  minor gc只会影响年轻代的对象，一旦进入老年代，只有full gc才能干掉它，所以如果一次批量minor gc对象的量很大(高并发情况下)，这样会导致对象直接进入老年代，这样就会导致老年代内存打满，频繁触发full gc，这样性能就完蛋了

  ```
  # 亿级别流量的对象生成，根据系统内存的情况设置堆，方法区内存大小
  # 默认情况下，老年区将会占用2/3的内存，余下的内存Eden占用8/10,剩下的两个survivor区域各占1/10
  java -Xms3072M -Xmx3072 -Xss1M -XX:MetaspaceSize=512M -XX:MaxMetaspaceSize=512M -jar microservice-eureka-server.jar
  ```

JMM内存模型

- JSR133规范