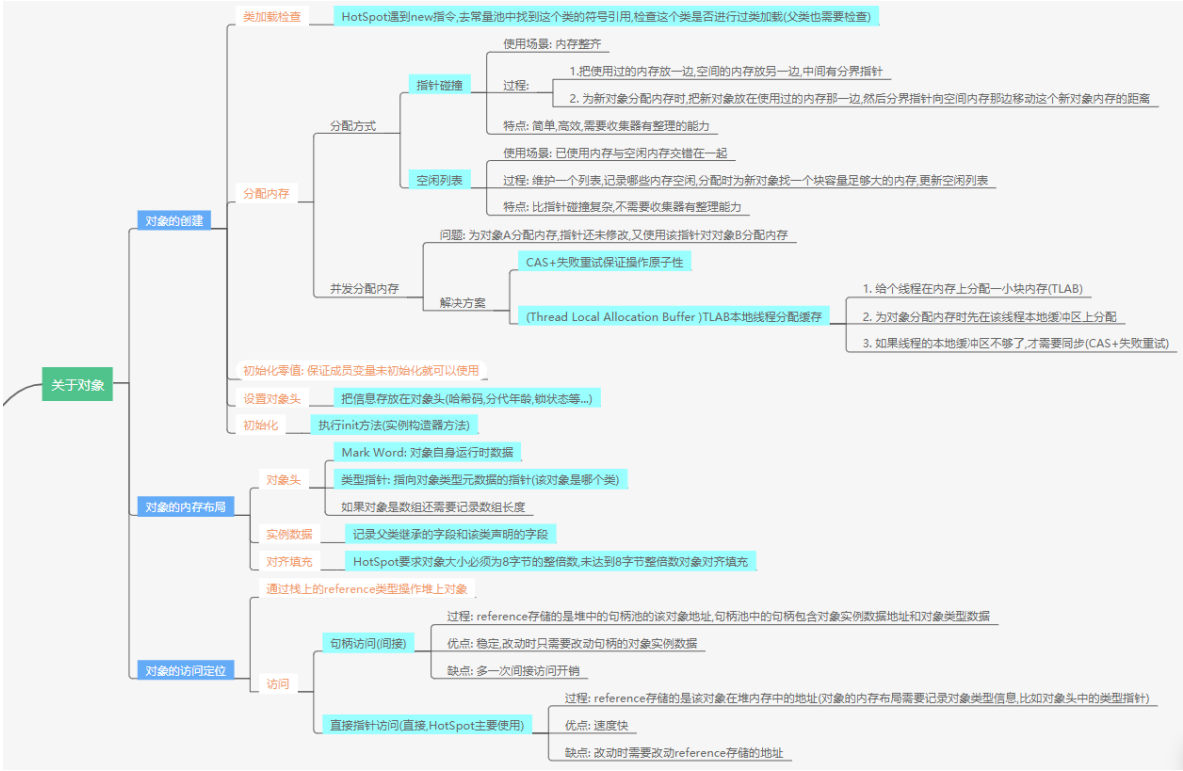
- HotSpot虚拟机对象探秘
  - 对象的创建
    - 类加载检查
    - 分配内存
      - 分配内存方式
      - 分配内存流程
    - 初始化零值
    - 设置对象头
    - 执行init方法
  - 对象的内存布局
    - 对象内存信息
    - 分析对象占用字节
    - 结构图
  - 对象的访问定位
- Java内存模型与线程
  - Java内存模型
    - 交互操作以及注意事项
    - volatile特性
      - 可见性
      - 不保证原子性
      - 禁止指令重排序
    - double,long的非原子性协定
    - 原子性,可见性与有序性
    - 先行发生原则
  - 线程
    - 线程的实现
      - 内核线程实现
      - 用户线程实现
      - 混合实现
    - Java线程的实现
    - Java线程的调度
    - 线程状态转换
- Java内存区域与内存溢出异常
  - 程序计数器
  - Java虚拟机栈
    - 简介
    - 运行时栈帧结构
      - 局部变量表
      - 操作数栈
      - 动态连接
      - 方法返回地址
      - 附加信息
    - 模拟栈溢出
  - 本地方法栈
  - Java 堆
    - 简介
    - 堆的内存结构
    - 内存调优
      - 堆内存常用参数
      - 查看堆内存

HotSpot虚拟机对象探秘



对象的创建

对象的创建可以分为五个步骤:**检查类加载,分配内存,初始化零值,设置对象头,执行实例构造器**

类加载检查

- **HotSpot虚拟机遇到一条new指令,会先检查能否在常量池中定位到这个类的符号引用,检查这个类是否类加载过**
  - 没有类加载过就去类加载
  - 类加载过就进行下一步分配内存

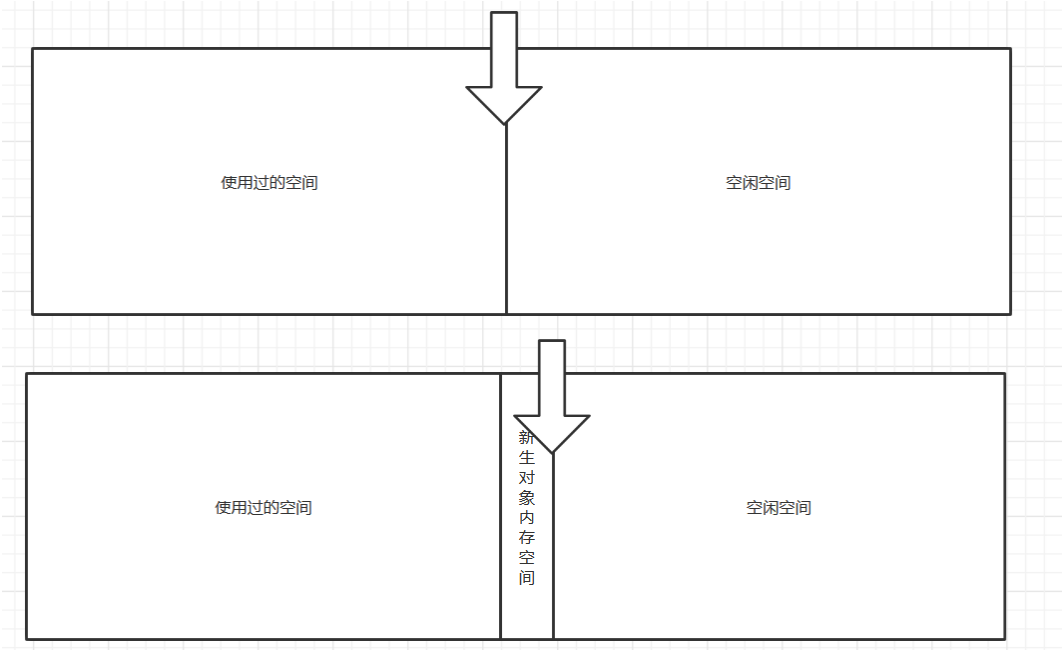
分配内存

对象所需的内存在类加载完成后就可以完全确定

分配内存方式

虚拟机在堆上为新对象分配内存,有两种内存分配的方式:**指针碰撞,空闲列表**

- 指针碰撞
  - 使用场景: 堆内存规整整齐
  - 过程: 使用过的空间放在一边,空闲的空间放在另一边,中间有一个指针作为分界点指示器,把新生对象放在使用过空间的那一边,中间指针向空闲空间那边挪动一个新生对象的内存大小的距离即可



  - 特点: `简单,高效` ,因为要堆内存规整整齐,所以垃圾收集器应该要有 `压缩整理` 的能力
- 空闲列表
  - 使用场景: 已使用空间和空闲空间交错在一起
  - 过程: 虚拟机维护一个列表,列表中记录了哪些内存空间可用,分配时找一块足够大的内存空间划分给新生对象,然后更新列表
  - 特点: 比指针碰撞复杂, 但是对垃圾收集器可以不用压缩整理的能力

分配内存流程

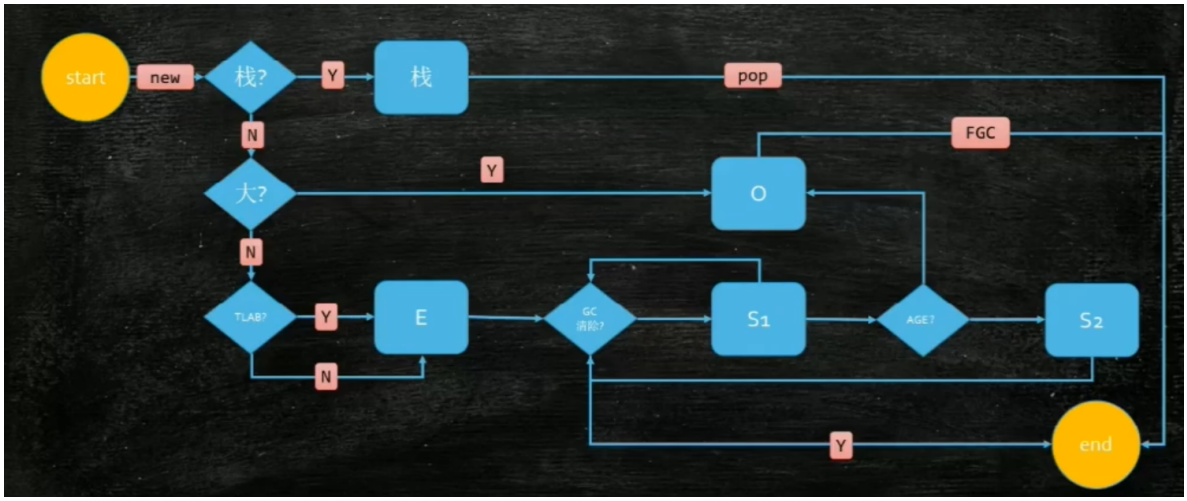
> 分配内存流程(栈--老年代--TLAB--Eden)

因为在堆上为对象分配内存,内存不足会引起GC,引起GC可能会有STW(Stop The World)影响响应

为了优化减少GC,当**对象不会发生逃逸(作用域只在方法中,不会被外界调用)且栈内存足够时,直接在栈上为对象分配内存**,当线程结束后,栈空间被回收,(局部变量也被回收)就不用进行垃圾回收了

开启逃逸分析`-XX:+DoEscapeAnalysis`满足条件的对象就在栈上分配内存

(当对象满足不会逃逸条件除了能够优化在栈上分配内存还会带来锁消除,标量替换等优化...)



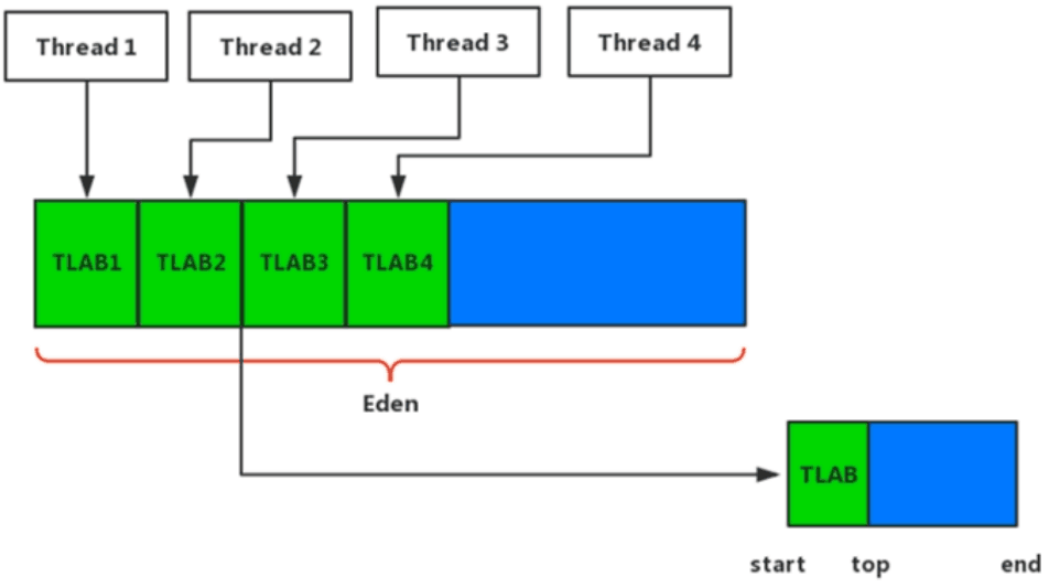
1. 尝试该对象能不能在栈上分配内存
2. 如果不符合1,且该对象特别的大,比如内存超过了JVM设置的大对象的值就直接在老年代上为它分配内存
3. 如果这个对象不大,为了解决并发分配内存,采用 `TLAB 本地线程分配缓冲`

> TLAB 本地线程分配缓存

堆内存是线程共享的,并发情况下从堆中划分线程内存不安全,如果直接加锁会影响并发性能

为每个线程在Eden区分配小小一块属于线程的内存,类似缓冲区

哪个线程要分配内存就在那个线程的缓冲区上分配,只有缓冲区满了,不够了才使用乐观的同步策略(CAS+失败重试)保证分配内存的原子性



在并发情况下分配内存是不安全的(正在给A对象分配内存,指针还未修改,使用原来的指针为对象B分配内存),虚拟机**采用TLAB(Thread Local Allocation Buffer本地线程分配缓冲)和CAS+失败重试**来保证线程安全

- TLAB：为每一个线程预先在伊甸园区（Eden）分配一块内存，JVM给线程中的对象分配内存时先在TLAB分配，直到对象大于TLAB中剩余的内存或TLAB内存已用尽时才需要同步锁定(也就是CAS+失败重试)
- CAS+失败重试：采用CAS配上失败重试的方式保证更新操作的原子性

初始化零值

分配内存完成后,虚拟机将分配的内存空间初始化为零值(不包括对象头) (零值: Integer对应0等)

保证了对象的成员字段(成员变量)在Java代码中不赋初始值就可以使用

设置对象头

把一些信息(这个对象属于哪个类? 对象哈希码,对象GC分代年龄)存放在对象头中 (后面详细说明对象头)

执行init方法

init方法 = 实例变量赋值 + 实例代码块 + 实例构造器

按照我们自己的意愿进行初始化

对象的内存布局

对象内存信息

对象在堆中的内存布局可以分为三个部分: `对象头,实例数据,对齐填充`

- 对象头包括两类信息(8Byte + 4Byte)
  1. `Mark Word` :用于存储该对象自身运行时数据(该对象的**哈希码信息**,**GC信息**:分代年龄,**锁信息**:状态标志等)
  2. `类型指针(对象指向它类型元数据的指针)` :HotSpot通过类型指针确定该对象是哪个类的实例 (**如果该对象是数组,对象头中还必须记录数组的长度**)

     类型指针默认是压缩指针,内存超过32G时为了寻址就不能采用压缩指针了
- 实例数据是对象真正存储的有效信息
  1. **记录从父类中继承的字段和该类中定义的字段**
  2. 父类的字段会出现在子类字段之前,默认子类较小的字段可以插入父类字段间的空隙以此来节约空间(`+XX:CompactFields`)
- 对齐填充

  HotSpot要求对象起始地址必须是8字节整倍数

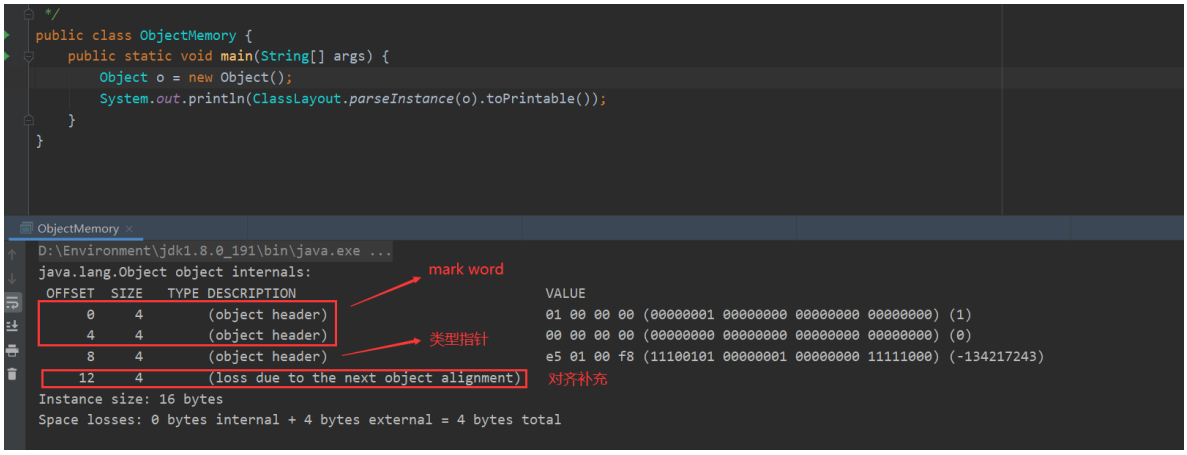
  所以 `任何对象的大小都必须是8字节的整倍` ,如果对象实例数据部分未到达8字节就会通过对齐填充进行补全

分析对象占用字节

> Object obj = new Object(); 占多少字节?

导入JOL依赖

```
<!-- https://mvnrepository.com/artifact/org.openjdk.jol/jol-core -->
        <dependency>
            <groupId>org.openjdk.jol</groupId>
            <artifactId>jol-core</artifactId>
            <version>0.12</version>
        </dependency>
```

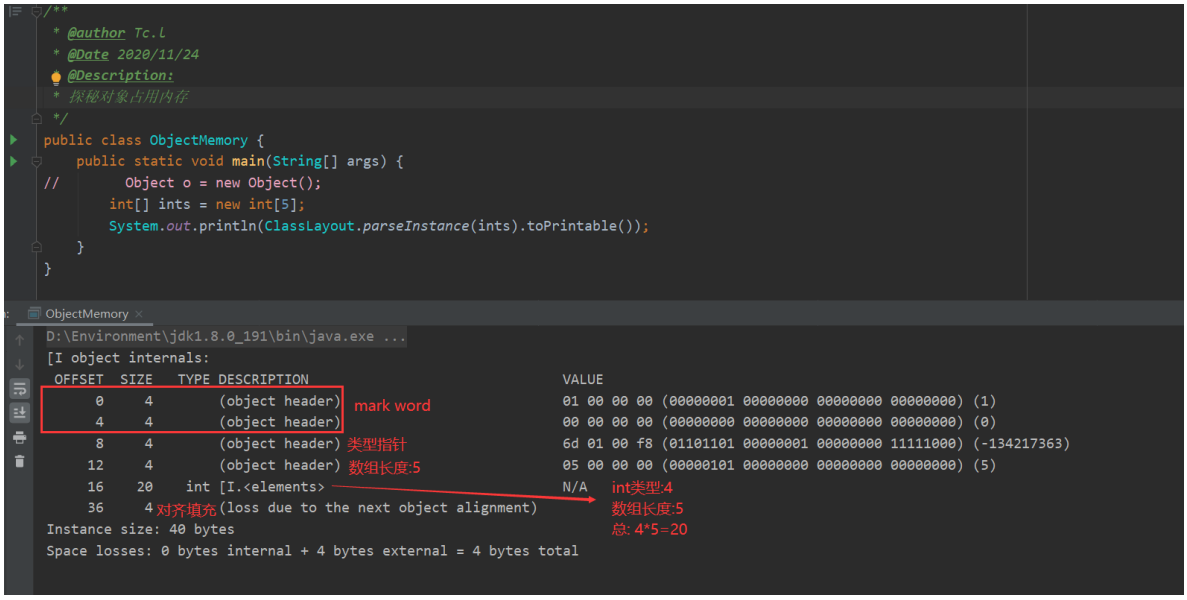


mark word : 8 byte

类型指针: 4 byte

对齐填充 12->16 byte

> int[] ints = new int[5]; //占多少内存?



mark word:8 byte

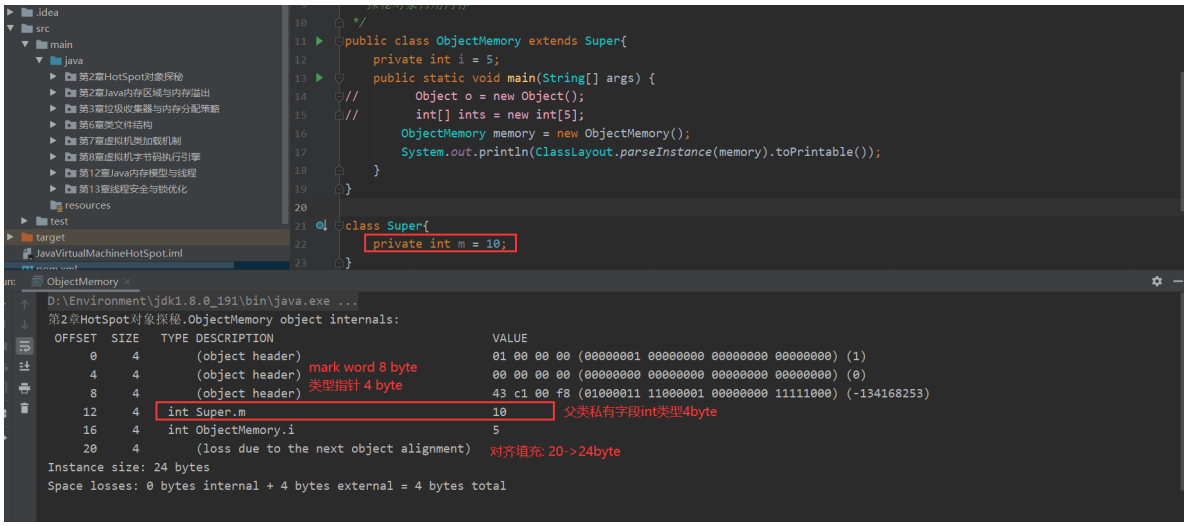
类型指针: 4 byte

数组长度: 4 byte

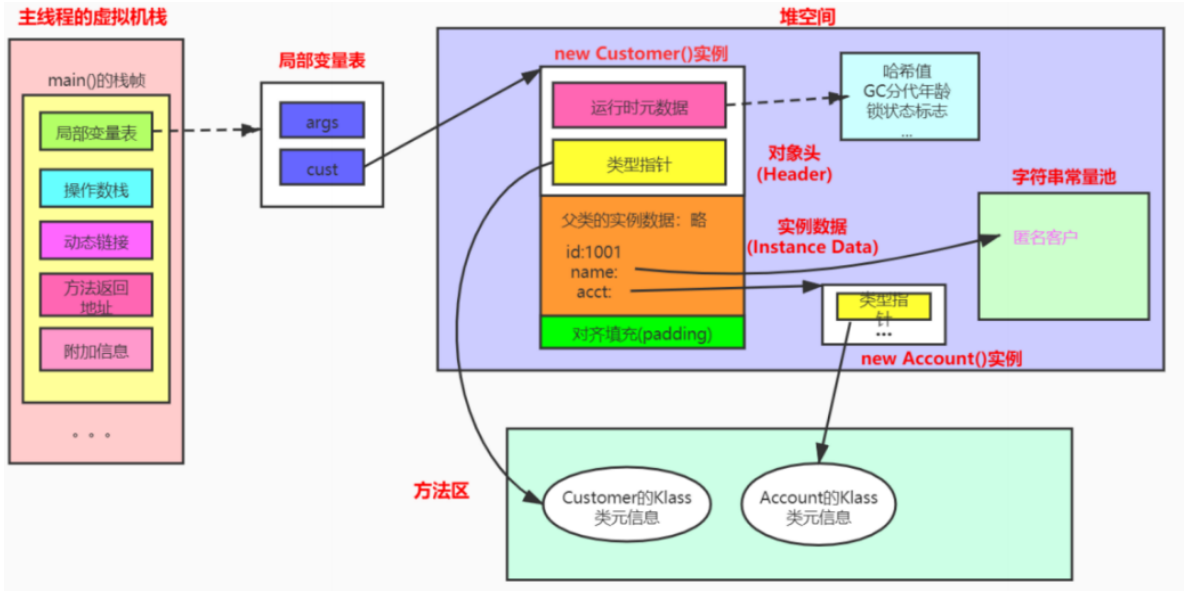
数组内容初始化: 4*5=20byte

对齐填充: 36 -> 40 byte

> 父类私有字段到底能不能被子类继承?



结构图



对象的访问定位

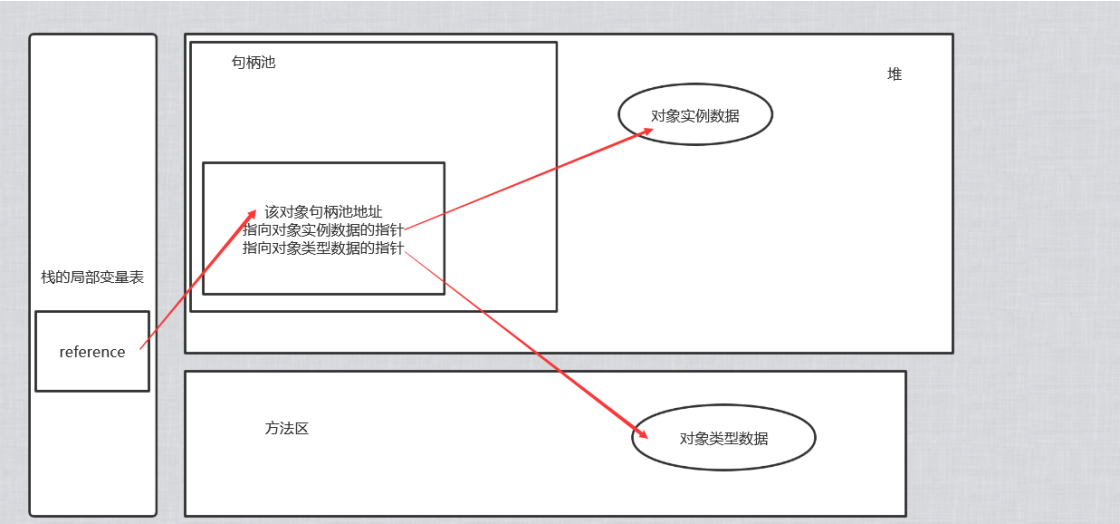
Java程序通过栈上的reference类型数据来操作堆上的对象

> 访问方式

对象实例数据: 对象的有效信息字段等(就是上面说的数据)

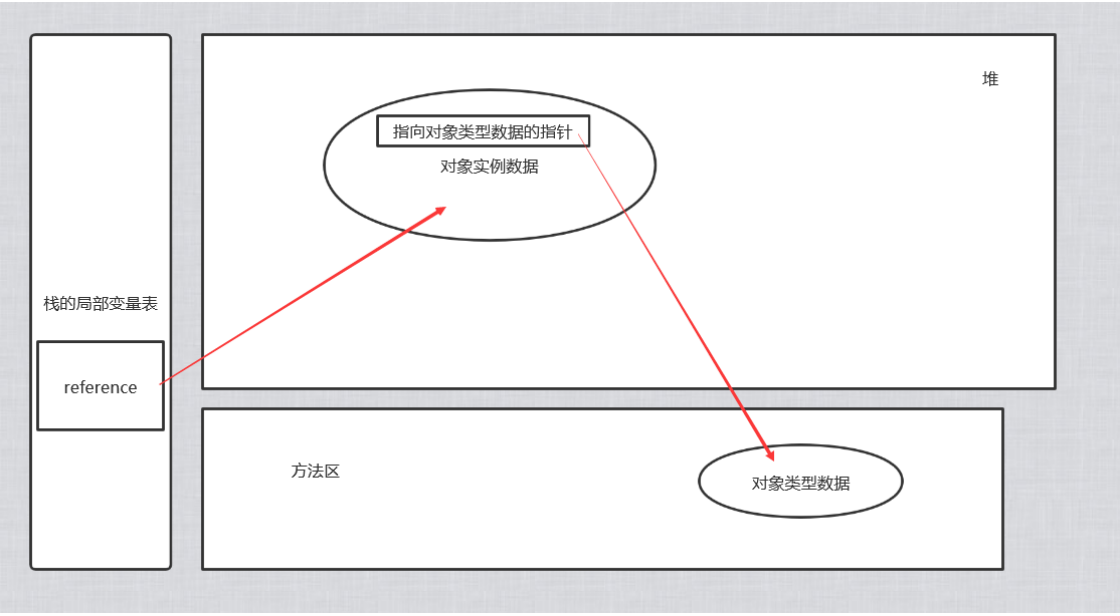
对象类型数据: 该对象所属类的类信息(存于方法区中)

- 句柄访问



- 在堆中开辟一块内存作为句柄池,栈中的reference**数据存储的是该对象句柄池的地址,句柄中包含了对象实例数据和对象类型数据**
- **优点: 稳定,对象被移动时(压缩或复制算法),只需要改动该句柄的对象实例数据指针**
- **缺点: 多一次间接访问的开销**

- 直接指针访问



- 栈中的reference数据存储堆中该对象的地址(reference指向该对象),但是**对象的内存布局需要保存对象类型数据**
- **优点: 访问速度快**
- **缺点: 不稳定,对象被移动时(压缩或复制算法),需要改动指针**

访问方式是虚拟机来规定的,**Hotspot主要使用直接指针访问**

HotSpot虚拟机对象探秘
对象的创建
类加载检查
分配内存
分配内存方式
分配内存流程

Java内存模型与线程

Java内存模型

目的: 为了定义程序中各种共享变量访问规则

Java内存模型规定:

1. **所有的共享变量都存储在主内存中**(物理上是虚拟机的一部分)
2. **每条线程有自己的工作内存**
3. **线程的工作内存保存了被该线程使用变量的主内存副本**
4. **线程对内存的所有操作(读写等)都要在工作内存进行,不能直接操作主内存**
5. **不同线程间无法访问对方工作内存的变量,线程间变量值传递需要通过主内存来完成**

注意: `主内存与工作内存` 可以类比为 `内存与高速缓冲存储器(cache)`

从主内存中读取数据到工作内存,线程操作工作内存修改数据,最终从工作内存写到主内存上

从内存中读取数据到cache,CPU操作cache上的数据,最终从cache再写回到主内存上

交互操作以及注意事项

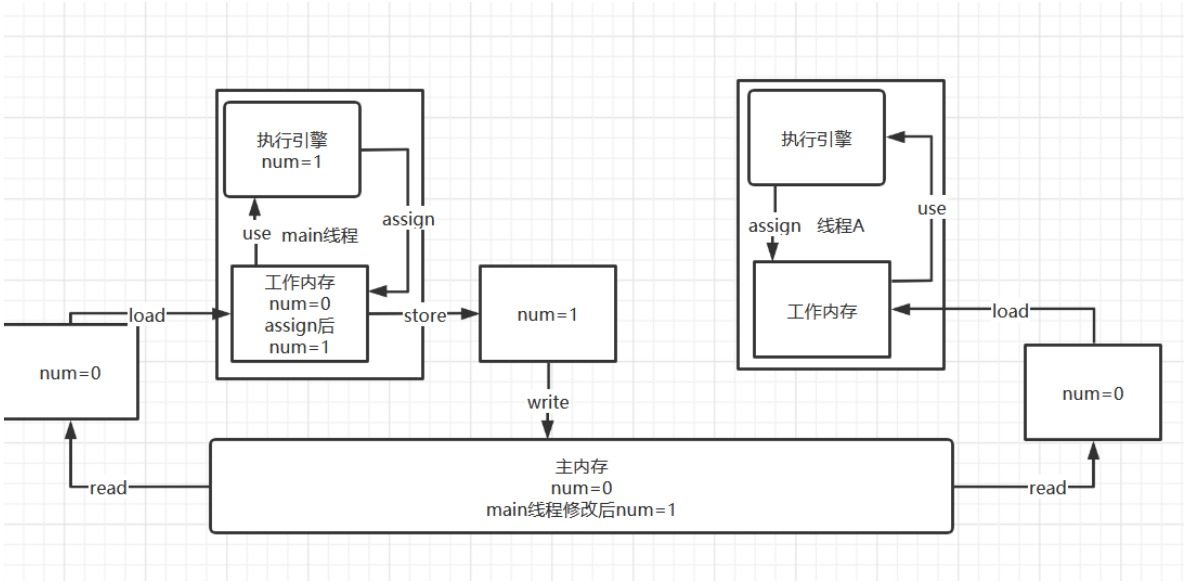
```java
/**
 * @author Tc.l
 * @Date 2020/11/3
 * @Description:
 * 线程A读到num为0,让num为0时,线程A就循环
 * 然后通过主线程修改num的值
 * 但是程序不能停下来 一直处于运行状态(线程A依旧在循环)
 */
public class JavaMemoryModel {
    static int num = 0;
    public static void main(String[] args) {
        new Thread(()->{
            while (num==0){

            }
        },"线程A").start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        num=1;
    }
}
```

上面的代码用Java内存模型图可以这样表示:



8种内存交互操作

主内存与工作内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

1. lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
2. unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
3. read （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用
4. load （载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
5. use （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
6. assign （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
7. store （存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
8. write （写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中

使用规则与注意事项

JMM对这八种指令的使用，制定了如下规则：

1. 不允许read和load、store和write操作之一单独出现。**即使用了read必须load，使用了store必须write**
2. 不允许线程丢弃他最近的assign操作，即**工作变量的数据改变了之后，必须告知主存**
3. 不允许一个线程将没有assign的数据从工作内存同步回主内存,即**未改变数据,又把数据从工作内存写回主内存**
4. 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。**就是对变量实施use、store操作之前，必须经过load,assign操作**

5. 一个变量同一时间只有一个线程能对其进行lock。**多次lock后，必须执行相同次数的unlock才能解锁**
6. 如果对一个变量进行lock操作，**加锁会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值**
7. 如果一个变量**没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量**
8. 对一个变量进行unlock**操作之前，必须把此变量同步回主内存**

volatile特性

volatile是Java虚拟机提供最轻量级的同步机制

可见性

> 验证可见性

保证此变量对所有线程的可见性(例子中: 主线程修改num后,线程A识别num不为0退出循环)

```
public class JavaMemoryModel {
    static volatile int num = 0;
    public static void main(String[] args) {
        new Thread(()->{
            while (num==0){

            }
        },"线程A").start();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        num=1;
    }
}
```

保证新值能立即同步到主内存,以及每次使用前立即从主内存刷新

不保证原子性

> 验证不保证原子性

原子性: 要么同时成功,要么同时失败

```
public class AtomicTest {
    static int num = 0;

    public static void add(){
        num++;
    }
    public static void main(String[] args) {
```

```
8           //多线程 执行 num自增 十万次
9           for (int i = 0; i < 10; i++) {
10              new Thread(()->{
11                  for (int j = 0; j <10000 ; j++) {
12                      add();
13                  }
14              }).start();
15          }
16          //保证线程都执行完
17          while (Thread.activeCount()>2){
18              Thread.yield();
19          }
20          //38806 ,  26357
21          System.out.println(num);
22      }
23  }
```

理论上num应该为十万,但是每次都少很多

使用 `javap -c` 进行反编译 查看字节码

实际上num++时需要拿到这个静态变量然后操作,操作完再记录回去,在多线程中可能有的线程已经自加了但是还未记录回去,让别的线程读到错误的数量而导致不安全

```
D:\代码\JUC\src\main\java\com\tcl\volatileTest>javap -c AtomicTest.class
Compiled from "AtomicTest.java"
public class com.tcl.volatileTest.AtomicTest {
  static int num;

  public com.tcl.volatileTest.AtomicTest();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void add();
    Code:
       0: getstatic     #2                  // Field num:I
       3: iconst_1
       4: iadd
       5: putstatic     #2                  // Field num:I
       8: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        10
       5: if_icmpge     29
       8: new           #3                  // class java/lang/Thread
```

禁止指令重排序

> 指令重排

Java虚拟机的即时编译器有对指令重排序的优化

指令重排序: 不影响最终正确结果的情况下,指令执行顺序可能会与程序代码中执行顺序不同

我们写的程序到机器可以执行的指令,之间这个过程可能会改变指令执行的顺序

源代码->编译器优化重排->指令并行重排->内存重排->机器执行

进行指令重排时,会考虑数据间的依赖

```
int x = 0;//1
int y = 4;//2
y = x - 1;//3
x = x * x;//4
```

我们写的顺序是1234,但是执行的时候可能是2134或1423这都是不影响结果的

但是在多线程中(默认一开始b,c,x,y都是0)

线程A	线程B
x = c	y = b
b = 1	c = 2

结果: x = 0, y = 0

重排指令后

线程A	线程B
b = 1	c = 2
x = c	y = b

结果: x = 2 , y = 1

在多线程中是不安全的(逻辑上存在的)

使用volatile可以禁止指令重排,以防这种情况发生

> volatile避免指令重排

CPU指令的作用,使用内存屏障 指令重排不能把内存屏障后的指令重排到内存屏障前

1. 保证特定操作执行顺序(比如禁止指令顺序交换)
2. 保证某些变量内存可见性(利用这个volatile实现可见性)

内存屏障是一个lock前缀的空操作,把前面锁住,前面没执行完就不能执行后面

lock前缀空操作的作用: 将本处理器的缓存写入内存中,该写入动作也会引起别的处理器或别的内核无效化其缓存,相当于把缓存中的变量store,write写入主内存中,别的处理器发现缓存无效了立马去主内存中读,就实现了可见性通过这个空操作,volatile实现可见性

lock前缀空操作指令修改同步到内存时,意味着之前操作执行完成,所以指令重排序无法越过内存屏障

> volatile变量与普通变量消耗性能的区别

- 读操作: volatile变量与普通变量读操作时消耗性能差不多
- 写操作: 因为volatile变量要使用内存屏障防止指令重排所以消耗会大些

double,long的非原子性协定

允许虚拟机自行实现是否保证64位数据类型的load,stroe,read,write四个原子性操作

主流平台下64位Java虚拟机不会出现非原子性访问行为,而32位存在此风险

原子性,可见性与有序性

- 原子性

  lock,unlock操作未直接开放给用户,但是提供了字节码指令 `monitorenter,monitorexit` 来隐式使用lock,unlock(在Java代码中就是synchronized关键字)

- 可见性
  - volatile : **保证新值能立即同步到主内存,以及每次使用前立即从主内存刷新**
  - synchronized: **对变量执行unlock前,要先执行store,write操作写入主内存**
  - final : **被final修饰不能改变所以无须同步可以被其他线程正确访问(引用未逃逸的情况下)**
- 有序性
  - volatile : **禁止指令重排**
  - synchronized : **一个变量在同一时刻只允许一条线程对其lock操作** 决定持有同一个锁的多个同步块只能串行进入

先行发生原则

- 程序次序规则: **同个线程内,书写在前面的操作先行发生于书写在后面的操作**
- 管程锁定规则: **一个unlock操作先行发生在后面对同一个锁的lock操作**
- volatile变量规则: **对一个volatile变量的写操作先行发生于后面对这个变量的读操作**
- 线程启动规则: **线程start()先行发生于此线程每个动作**
- 线程终止规则: **线程所有操作先行发生于此线程终止检测**
- 线程中断规则: **线程interrupt()方法调用先行发生于被中断线程的代码检测到中端事件发生**
- 对象终结规则: **一个对象初始化完成先行发生于他的finalize()方法**
- 线程join规则: **如果在线程A中执行 ThreadB.join()成功的话,线程B的操作先行发生于线程A的ThreadB.join()返回**
- 传递性: **操作A先行发生于操作B,操作B先行发生于操作C,可得操作A先行发生于操作C**

时间先后顺序于先行发生原则没有因果关系,衡量并发问题不要受时间顺序干扰,一切必须以先行发生原则为准

线程

线程是比进程更轻量级的调度执行单位

线程的实现

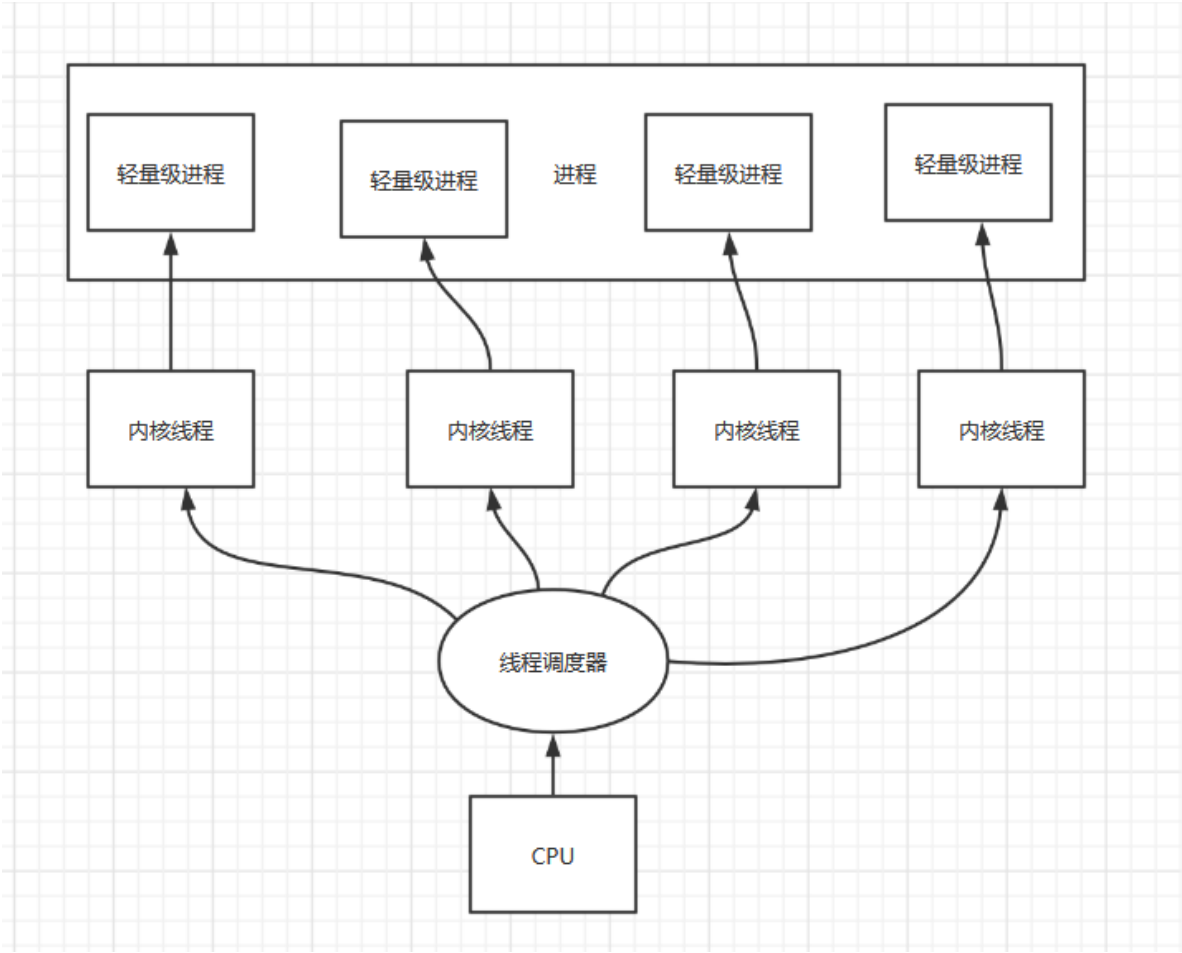
线程的实现主要有三种方法: **使用内核线程实现,使用用户线程实现,使用用户线程加轻量级进程混合实现**

内核线程实现

内核线程就是直接由操作系统内核支持的线程,该线程由内核来完成线程的切换

内核通过线程调度器对线程进行调度,并负责将线程任务映射到各个处理器

一般使用内核线程的高级接口 `轻量级进程(线程)` ,轻量级进程与内核线程1:1对应

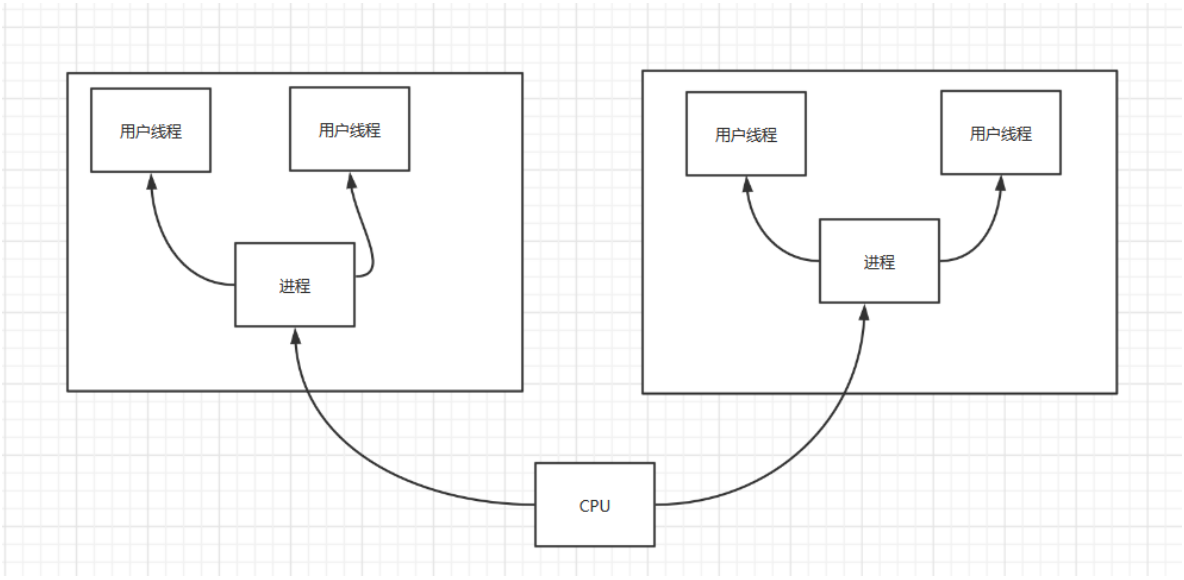


系统调用代价高,需要在用户态和内核态切换,会消耗一定内核资源,所以一个系统支持轻量级进程有限

用户线程实现

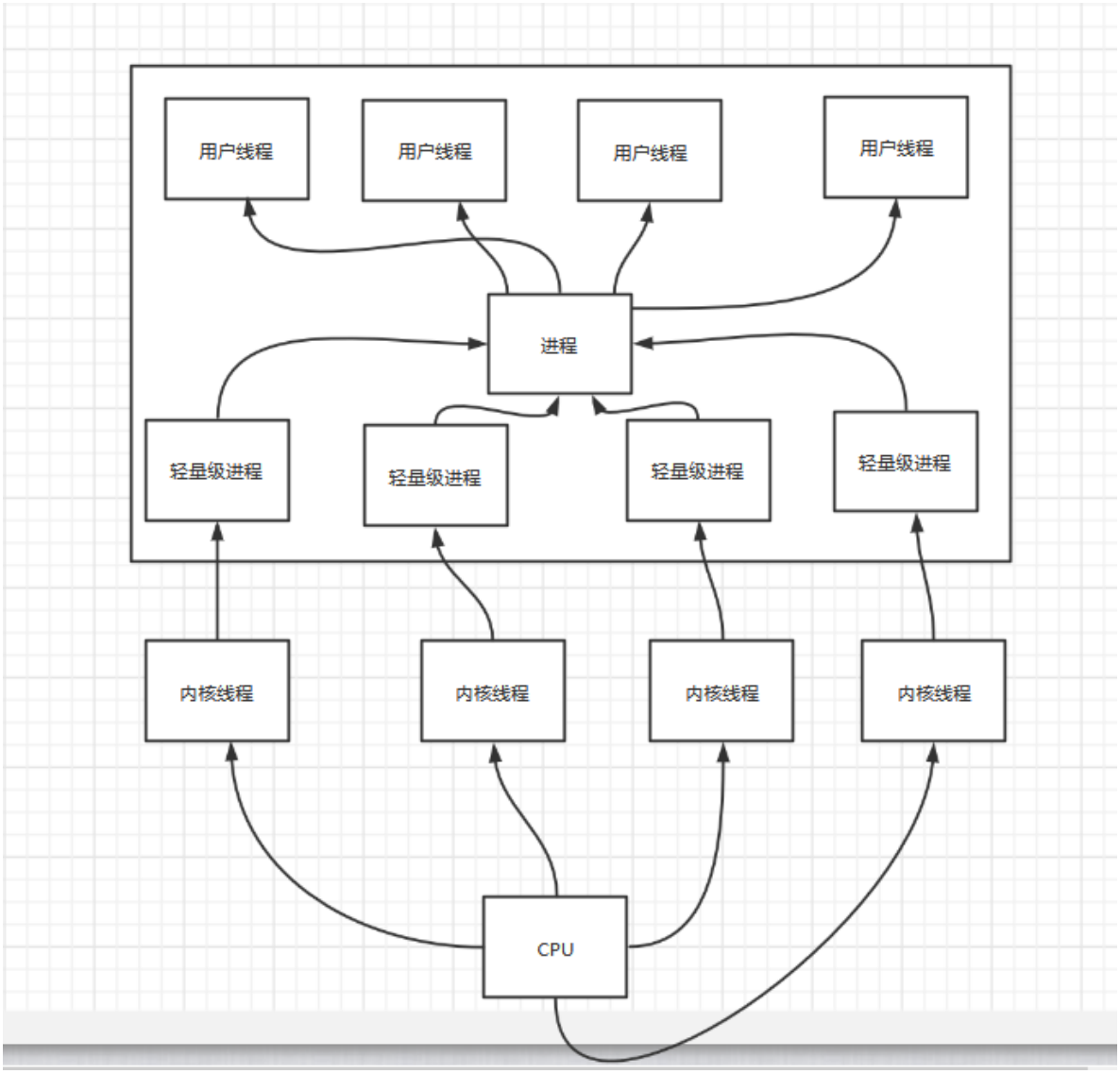
广义: 只要不是内核线程就是用户线程

侠义: 完全建立在用户态上的线程(系统内核不知道线程如何实现)



不需要内核帮助,执行速度快,低消耗,支持大规模线程数量,但是线程的创建,消耗,切换都必须要用户去考虑

混合实现



支持大规模用户线程并发,轻量级进程作为用户线程和内核线程的桥梁,内核提供线程调度功能

Java线程的实现

hotspot: 每个Java线程直接映射到操作系统原生线程来实现,中间没有额外间接结构,全权交给操作系统

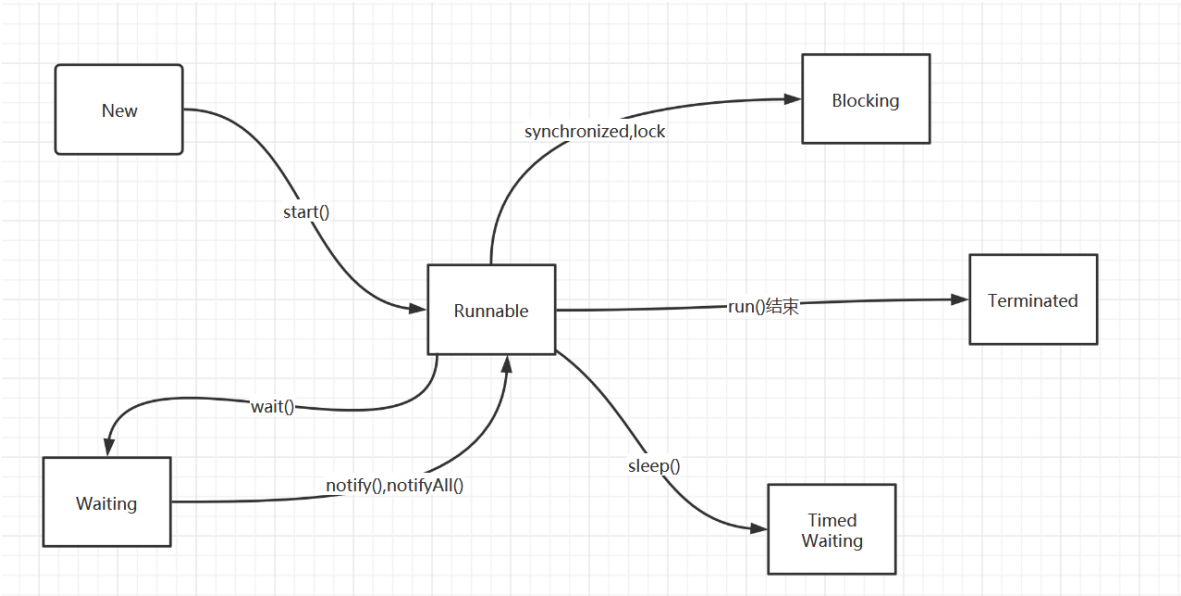
Java线程的调度

调度分为协同式线程调度和抢占式线程调度

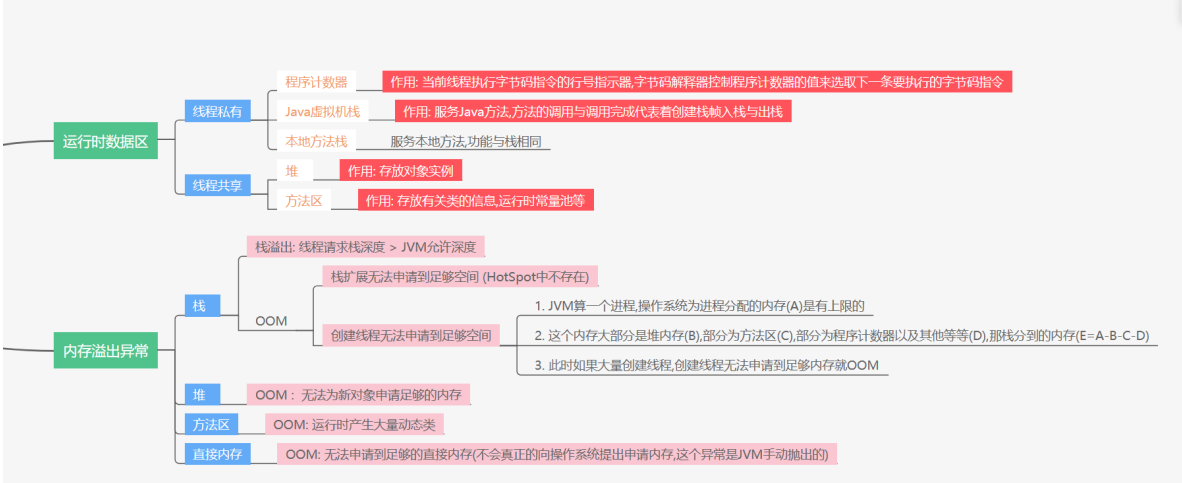
- 协同式线程调度
  - 优点: 实现简单
  - 缺点: 线程执行时间不可控制,不告诉系统切换就不切换,如果某个线程阻塞就一直阻塞(一直等)
- 抢占式线程调度(Java线程的调度方式)
  - 优点: 系统来进行分配执行时间,线程切换不由线程本身决定(可以通过设置线程优先级来优先执行,但这是不稳定的,Java中有10个优先级,Windows有7个,效果不会很理想)
  - 缺点: 线程无法主动获取执行时间(可以被动让出 `yield()`)

线程状态转换

- `NEW` 尚未启动的线程处于此状态
- `RUNNABLE` 在Java虚拟机中执行的线程处于此状态(**RUNNABLE状态可能处于执行状态也可能处于就绪状态**)
- `BLOCKED` 被阻塞等待监视器锁定的线程处于此状态**被阻塞**
- `WAITING` 正在等待另一个线程执行特定动作的线程处于此状态 **等待**
- `TIMED_WAITING` 正在等待另一个线程执行动作达到指定等待时间的线程处于此状态 **定时等待**
- `TERMINATED` 已退出的线程处于此状态 **死亡**



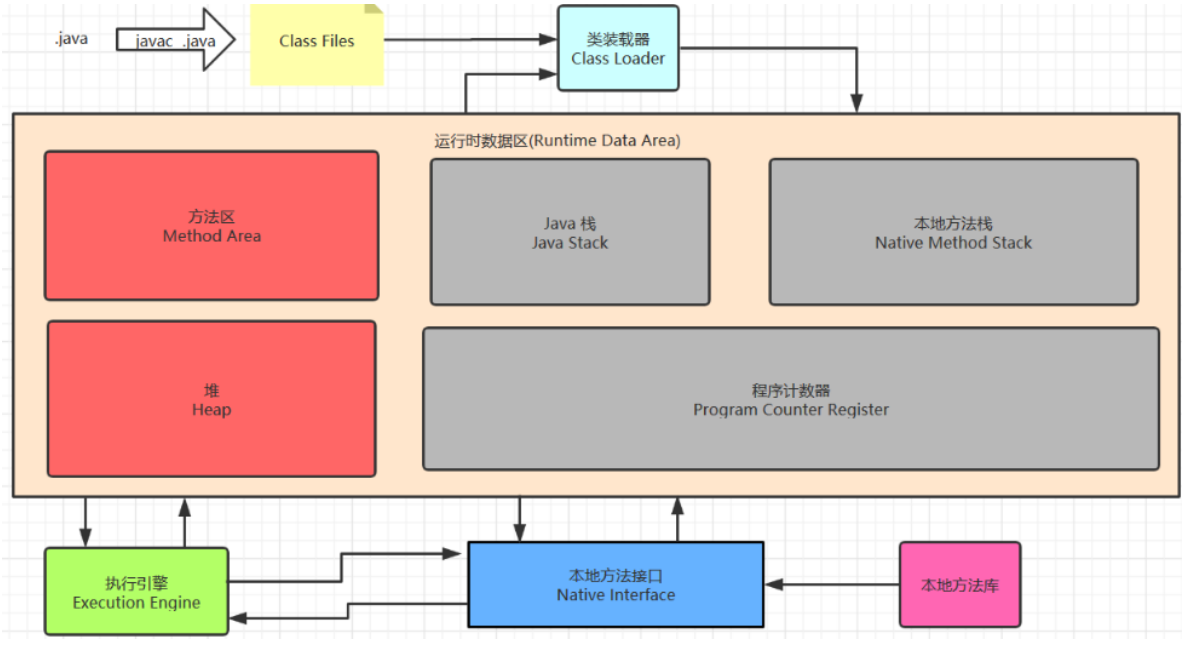
- HotSpot虚拟机对象探秘
  - 对象的创建
    - 类加载检查
    - 分配内存
      - 分配内存方式
      - 分配内存流程
    - 初始化零值
    - 设置对象头
    - 执行init方法
  - 对象的内存布局
    - 对象内存信息
    - 分析对象占用字节
    - 结构图
  - 对象的访问定位
- Java内存模型与线程
  - Java内存模型
    - 交互操作以及注意事项
    - volatile特性
      - 可见性
      - 不保证原子性
      - 禁止指令重排序
    - double,long的非原子性协定
    - 原子性,可见性与有序性
    - 先行发生原则
  - 线程
    - 线程的实现
      - 内核线程实现
      - 用户线程实现
      - 混合实现
    - Java线程的实现
    - Java线程的调度
    - 线程状态转换
- Java内存区域与内存溢出异常
  - 程序计数器
  - Java虚拟机栈
    - 简介
    - 运行时栈帧结构
      - 局部变量表
      - 操作数栈
      - 动态连接
      - 方法返回地址
      - 附加信息
    - 模拟栈溢出
  - 本地方法栈
  - Java 堆
    - 简介
    - 堆的内存结构
    - 内存调优
      - 堆内存常用参数
      - 查看堆内存
      - 修改堆内存
    - 模拟堆OOM异常



Java内存区域与内存溢出异常

Java虚拟机在运行Java程序时,把所管理的内存分为多个区域, 这些区域就是运行时数据区

运行时数据区可以分为:程序计数器,Java虚拟机栈,本地方法栈,堆和方法区



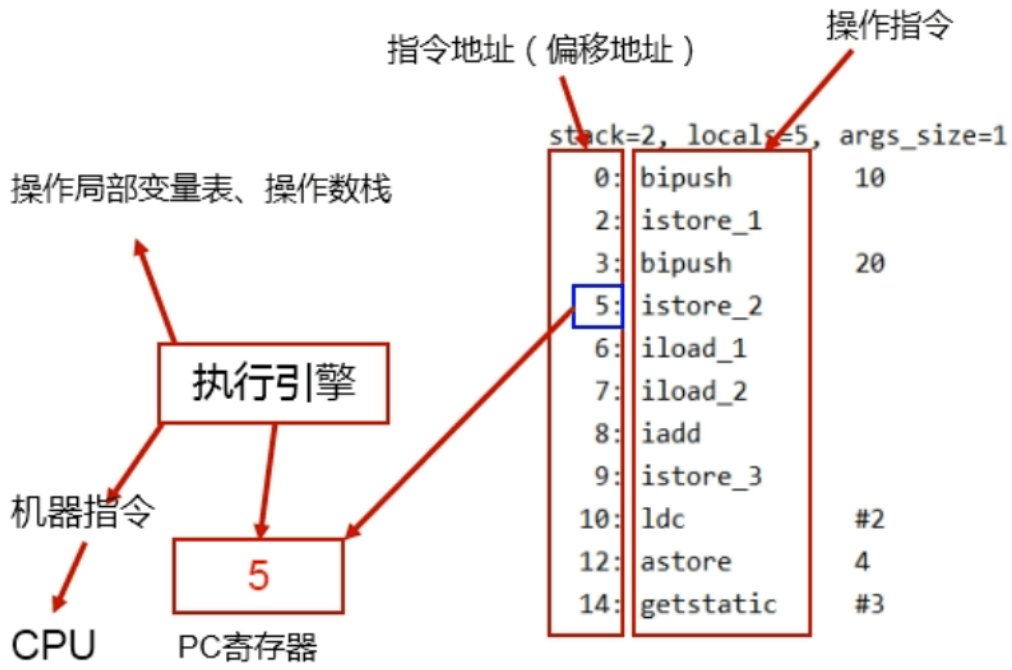
程序计数器

Program Counter Register 程序记数寄存器

- 什么是程序计数器?
- 程序计数器是一块**很小的内存**,它可以当作**当前线程执行字节码的行号指示器**
- 程序计数器的作用是什么?
    1. **字节码解释器通过改变程序计数器中存储的下一条字节码指令地址以此来达到流程控制**
    2. Java多线程的线程会切换,为了保存线程切换前的正确执行位置,每个线程都应该有程序计数器,因此**程序计数器是线程私有的**

    线程**执行Java方法时,程序计数器记录的是正在执行的虚拟机字节码指令地址**

    线程**执行本地方法时,程序计数器记录的是空**



pc寄存器保存下一条要执行的字节码指令地址

执行引擎根据pc寄存器找到对应字节码指令来使用当前线程中的局部变量表(取某个值)或操作数栈(入栈,出栈..)又或是将字节码指令翻译成机器指令,然后由CPU进行运算

- 生命周期
    - 因为程序计数器是线程私有的,所以**生命周期是随着线程的创建而创建,随着线程的消亡而消亡**
- 内存溢出异常
    - **程序计数器是唯一一个没有OOM(OutOfMemoryError)异常的数据区**

Java虚拟机栈

简介

Java Virtual Mechine Stack

- Java虚拟机栈描述 **线程执行Java方法时的内存模型**
- Java虚拟机栈的作用

    - 方法被执行时,JVM会创建一个**栈帧(Stack Fr1ame):用来存储局部变量,动态链接,操作数栈,方法出口等信息**
    - **方法被调用到结束对应着栈帧在JVM栈中的入栈到出栈操作**
- 生命周期

  因为是**线程私有的,所以随着线程的创建而创建,随着线程的消亡而消亡**

"栈"通常情况指的就是JVM栈,更多情况下**"栈"指的是JVM栈中的局部变量表**

- 局部变量表内容
    1. 八大基本数据类型
    2. 对象引用
        - 可以是指向对象起始地址的指针
        - 也可以是指向对象的句柄
    3. returnAddress类型(指向字节码指令的地址)

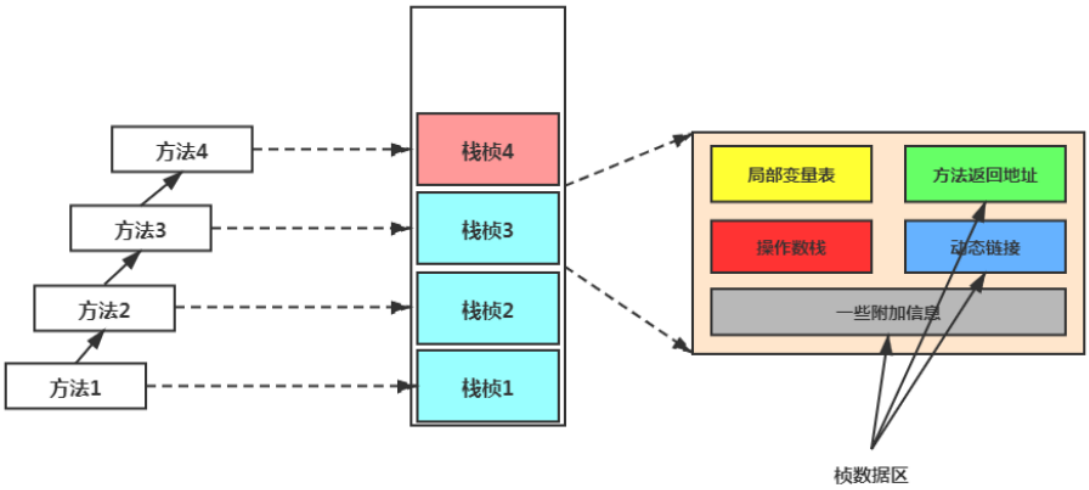
局部变量表中的存储空间以 局部变量槽(Slot) 来表示, double和long 64位的用2个槽来表示,其他数据类型都是1个

内存空间是在编译期间就已经确定的,运行时不能更改

这里的局部变量槽真正的大小由JVM来决定

运行时栈帧结构

> 结构图



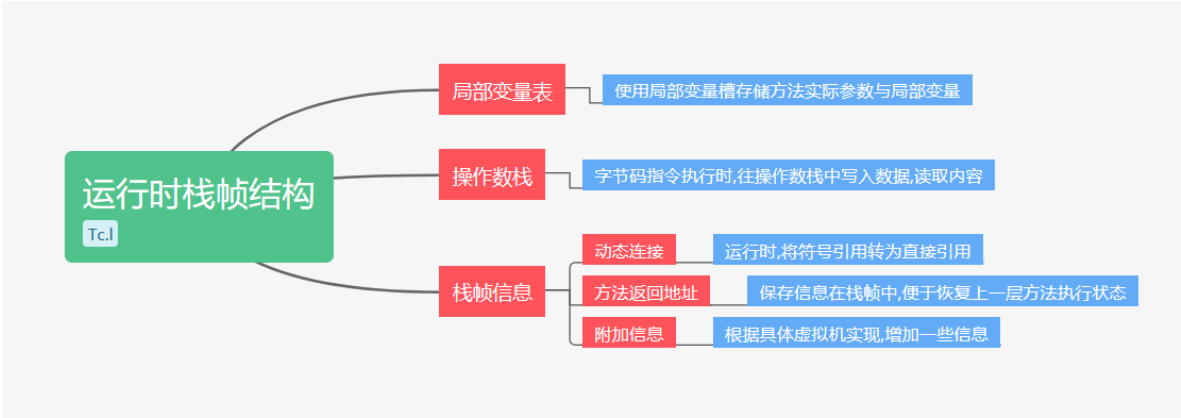
栈帧是Java虚拟机栈中的数据结构

Java虚拟机栈又是属于线程私有的

调用方法和方法结束 可以看作是 栈帧入栈,出栈操作

Java虚拟机以方法作为最基本的执行单位

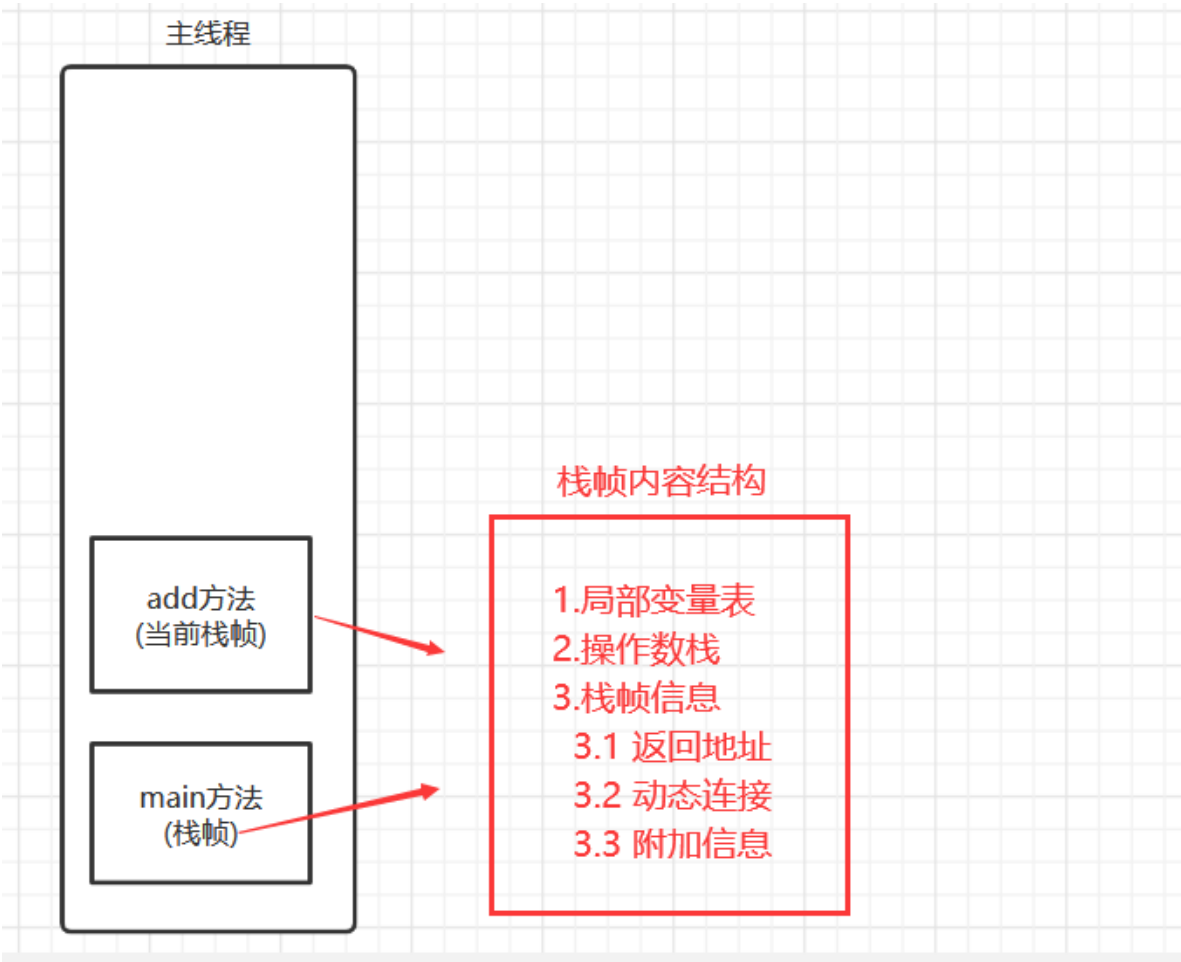
每个栈帧中包括: 局部变量表,操作数栈,栈帧信息(返回地址,动态连接,附加信息)



从Java程序来看:在调用堆栈的所有方法都同时处于执行状态(比如:main方法中调用其他方法)

从执行引擎来看:**当前线程只有处于栈顶的栈帧才是当前栈帧,此栈帧对应的方法为当前方法,执行引擎所运行的字节码指令只针对当前栈帧** 也就是执行引擎执行的字节码指令只针对栈顶栈帧(方法)

```
1       public void add(int a){
2           a=a+2;
3       }
4       public static void main(String[] args) {
5           new Test().add(10);
6       }
```



局部变量表

局部变量表用于存放方法中的 实际参数 **和** 方法内部定义的变量 **(存储)**

以局部变量槽为单位(编译期间就确定了)

每个局部变量槽都可以存放 `byte,short,int,float,boolean,reference,returnAddress`

byte,short,char,boolean在存储前转为int (boolean:0为false 非0为true)

而 `double,long` 由 两个局部变量槽存放

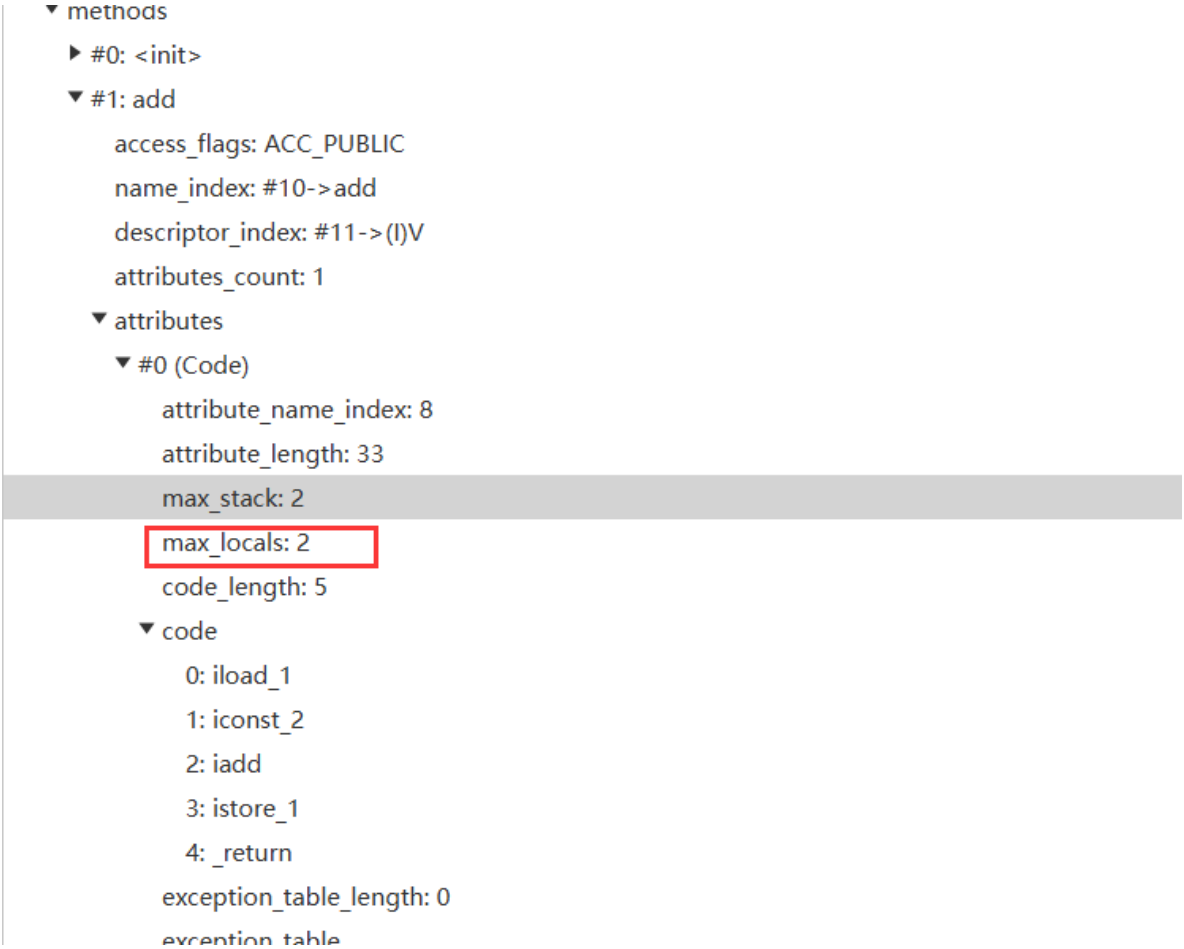
每个局部变量槽的真正大小应该是由JVM来决定的

> reference 和 returnAddress 类型是什么

- **reference : 直接或间接的查找到对象实例数据(堆中)和对象类型数据(方法区)** 也就是通常说的引用
- returnAddress: 曾经用来实现异常处理跳转,现在不用了,使用异常表代替

Java虚拟机通过 `定位索引` 的方式来使用局部变量表

局部变量表的范围: `0~max_locals-1`

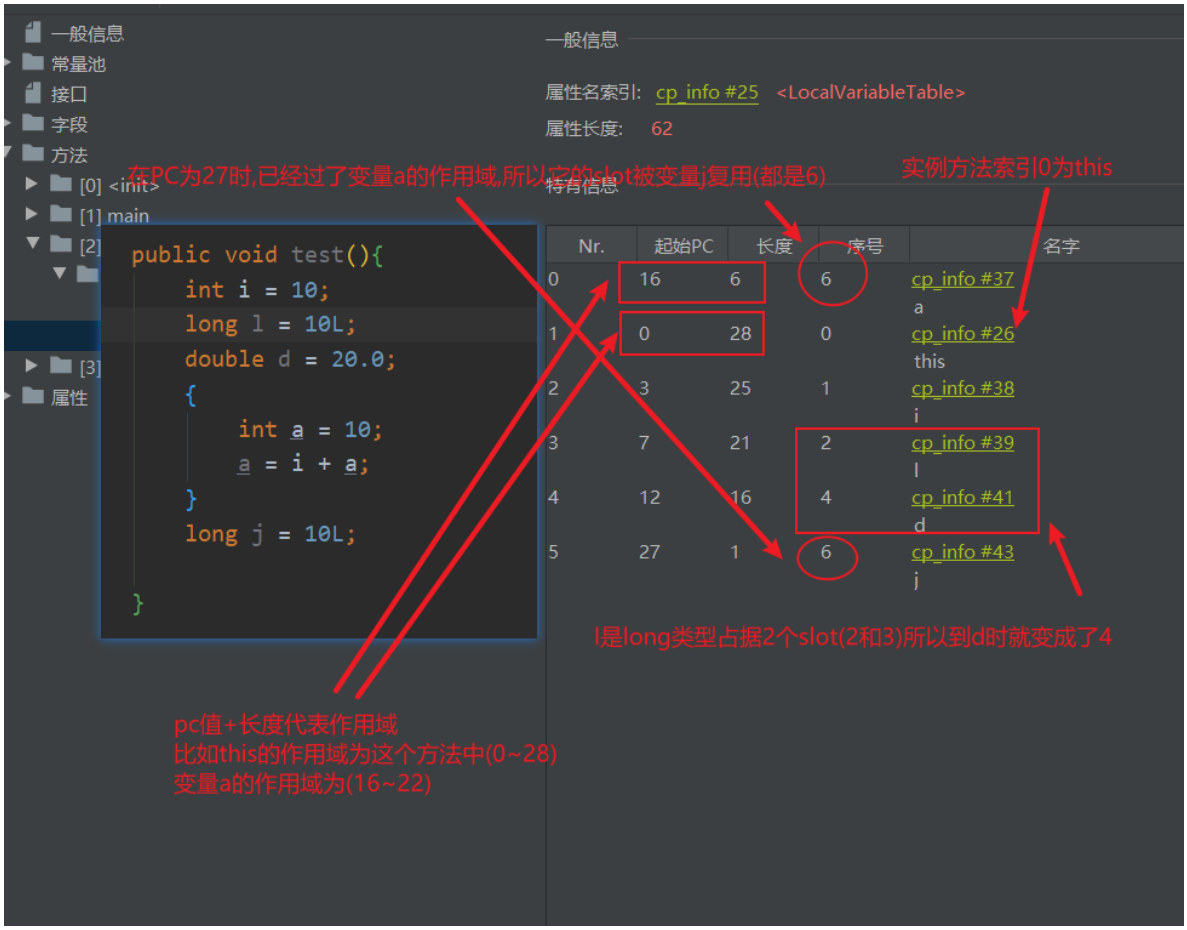


比如: 我们上面代码中add()方法只有一个int参数,也没有局部变量,为什么最大变量槽数量为2呢?

实际上: **默认局部变量槽中索引0的是方法调用者的引用(通过"this"可以访问这个对象)**

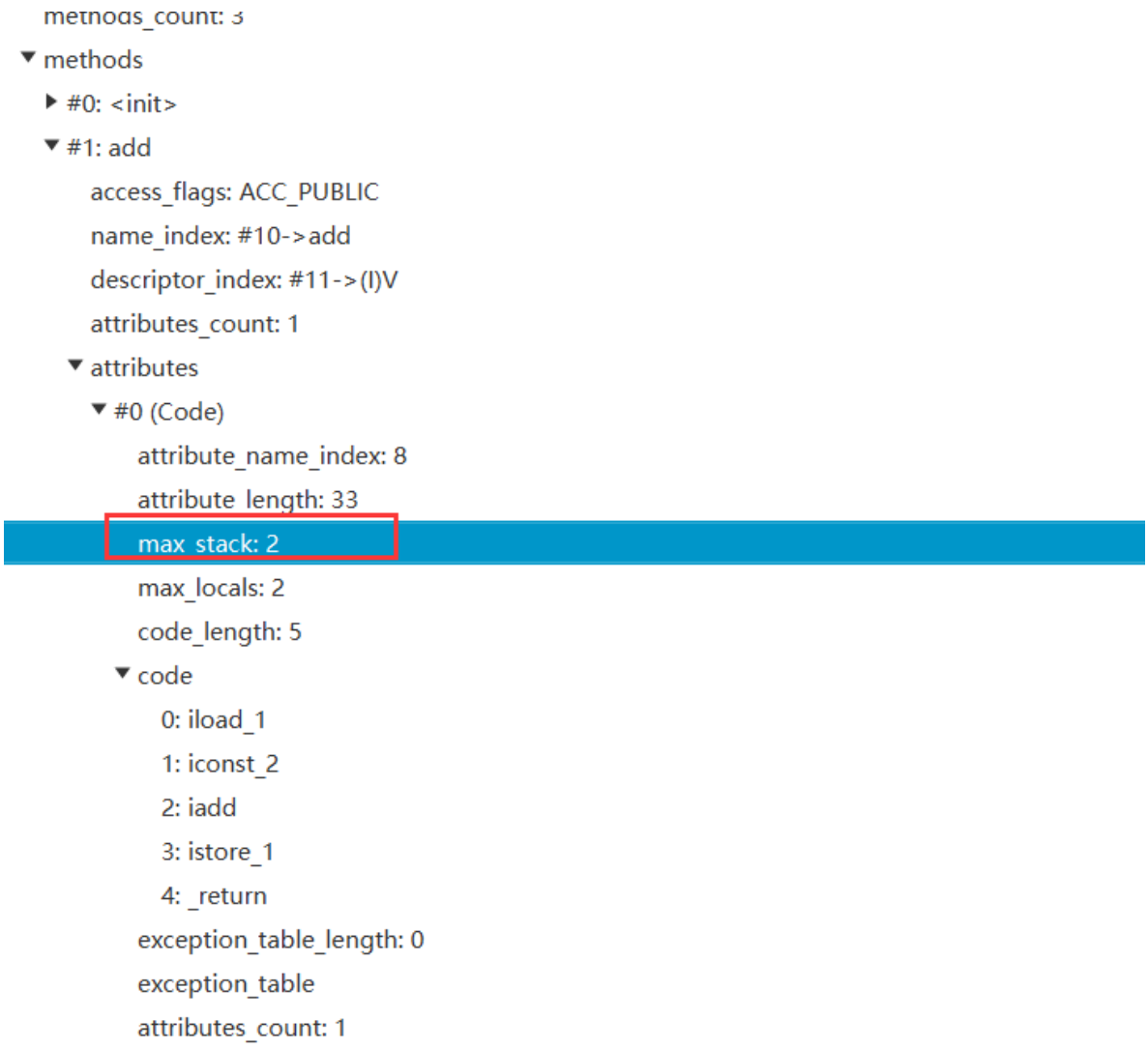
其余参数则**按照申明顺序**在局部变量槽的索引中

槽的复用:如果PC指令申明局部变量(j)已经超过了某个局部变量(a)的作用域,那么j就会复用a的slot



操作数栈

`max_stack` 操作数栈的最大深度也是编译时就确定下来了的



在方法执行的时候(字节码指令执行),会往操作数栈中写入和提取内容(比如add方法中 `a=a+2` ,a入栈,常数2入栈,执行相加的字节码指令,它们都出栈,然后把和再入栈)

操作数栈中的数据类型必须与字节码指令匹配(比如 `a=a+2` 都是Int类型的,字节码指令应该是 `iadd` 操作int类型相加,而不能出现不匹配的情况)

这是在类加载时验证阶段的字节码验证过程需要保证的

动态连接

动态连接:栈帧中指向运行时常量池所属方法的引用

> 静态解析与动态连接

符号引用转换为直接引用有两种方式

- **静态解析:在类加载时解析阶段将符号引用解析为直接引用**
- **动态连接:每次运行期间把符号引用解析为直接引用**(因为只有在运行时才知道到底指向哪个方法)

方法返回地址

执行方法后,有两种方式可以退出

> 正常调用完成与异常调用完成

- 正常调用完成: 遇到方法返回的字节码指令
  - 方法退出有时需要在栈帧中保存一些信息以恢复上一层方法的执行状态(程序计数器的值)
- 异常调用完成: 遇到异常未捕获(未搜索到匹配的异常处理器)
  - 以异常调用完成方式退出方法,不会在栈帧中保存信息,通过异常处理器来确定

附加信息

增加一些《Java虚拟机规范》中没有描述的信息在栈帧中(取决于具体虚拟机实现)

模拟栈溢出

- 内存溢出异常

1. 线程请求栈深度大于JVM允许深度,抛出StackOverflowError异常
2. 栈扩展无法申请到足够内存,抛出OOM异常
3. 创建线程无法申请到足够内存,抛出OOM异常

关于栈的两种异常

1. 线程请求**栈深度大于JVM允许深度**,抛出StackOverflowError异常
2. **栈扩展无法申请到足够内存** 或 **创建线程无法申请到足够的内存时**,抛出OOM异常

> 测试StackeOverflowError

另外在 hotSpot虚拟机中不区分虚拟机栈和本地方法栈 ,所以 `-Xoss` 无效,只有 `-Xss` 设置单个线程栈的大小

```
/**
 * @author Tc.l
 * @Date 2020/10/27
 * @Description: 测试栈溢出StackOverflowError
 * -Xss:128k 设置每个线程的栈内存为128k
 */
public class StackSOF {
    private int depth=1;

    public void recursion(){
```

```
        depth++;
        recursion();
    }

    public static void main(String[] args) throws Throwable {
        StackSOF sof = new StackSOF();
        try {
            sof.recursion();
        } catch (Throwable e) {
            System.out.println("depth:"+sof.depth);
            throw e;
        }
    }
}
/*
depth:1001
Exception in thread "main" java.lang.StackOverflowError
    at 第2章Java内存区域与内存溢出.StackSOF.recursion(StackSOF.java:12)
    at 第2章Java内存区域与内存溢出.StackSOF.recursion(StackSOF.java:13)
    ...
    at 第2章Java内存区域与内存溢出.StackSOF.recursion(StackSOF.java:13)
    at 第2章Java内存区域与内存溢出.StackSOF.main(StackSOF.java:19)
*/
```

减小了栈内存的空间,又递归调用频繁的创建栈帧,很快就会超过栈内存,从而导致StackOverflowError

> 测试OOM

在我们经常使用的hotSpot虚拟机中是不支持栈扩展的

所以线程运行时不会因为扩展栈而导致OOM,只有可能是**创建线程无法申请到足够内存而导致OOM**

```
/**
 * @author Tc.l
 * @Date 2020/10/27
 * @Description: 测试栈内存溢出OOM
 * -Xss2m 设置每个线程的栈内存为2m
 */
public class StackOOM {
    public void testStackOOM(){
        //无限创建线程
        while (true){
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    //让线程活着
                    while (true) {

                    }
                }
            });
            thread.start();
        }
    }

    public static void main(String[] args) {
        StackOOM stackOOM = new StackOOM();
        stackOOM.testStackOOM();
```

```
27          }
28      }
29
30      /*
31      Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
        native thread
32          at java.lang.Thread.start0(Native Method)
33          at java.lang.Thread.start(Thread.java:717)
34          at 第2章Java内存区域与内存溢出.StackOOM.testStackOOM(StackOOM.java:19)
35          at 第2章Java内存区域与内存溢出.StackOOM.main(StackOOM.java:25)
36      */
```

操作系统为(JVM)进程分配的内存大小是有效的,这个内存再减去堆内存,方法区内存,程序计数器内存,直接内存,虚拟机消耗内存等,剩下的就是虚拟机栈内存和本地方法栈内存

此时增加了线程分配到的栈内存大小,又在无限建立线程,就很容易把剩下的内存耗尽,最终抛出OOM

如果是因为这个原因出现的OOM,创建线程又是必要的,解决办法可以是 减小堆内存和减小线程占用栈内存大小

本地方法栈

Native Method Stacks

与JVM栈作用类似

JVM栈为Java方法服务

本地方法栈为本地方法服务

内存溢出异常也与JVM栈相同

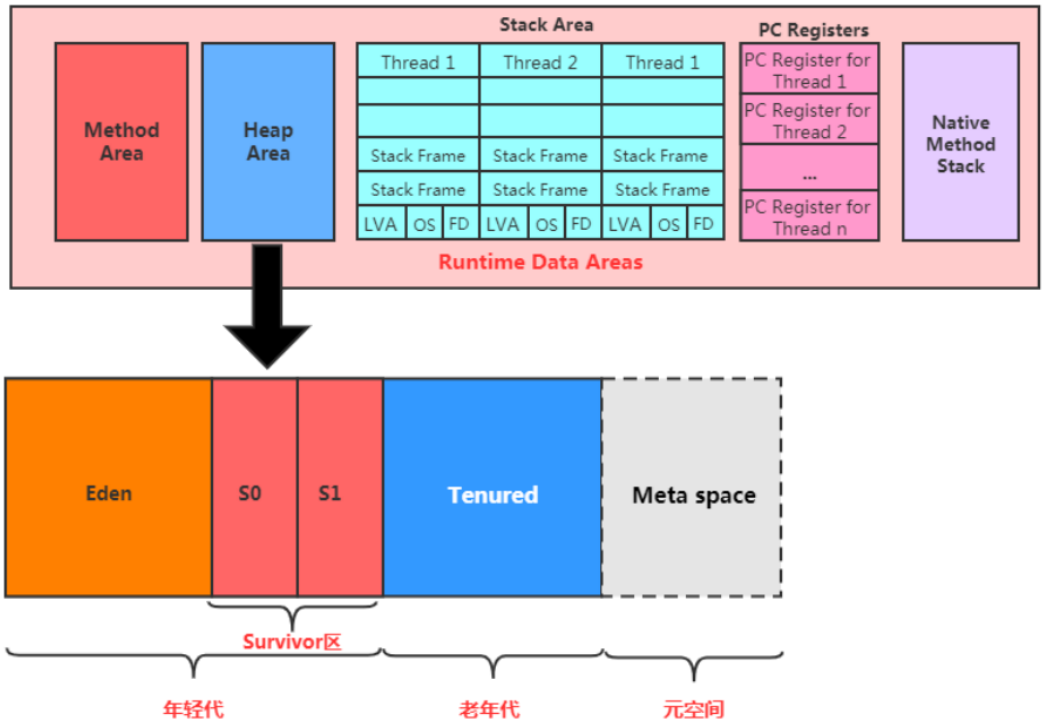
hotspot将本地方法栈和Java虚拟机栈合并

Java 堆

简介

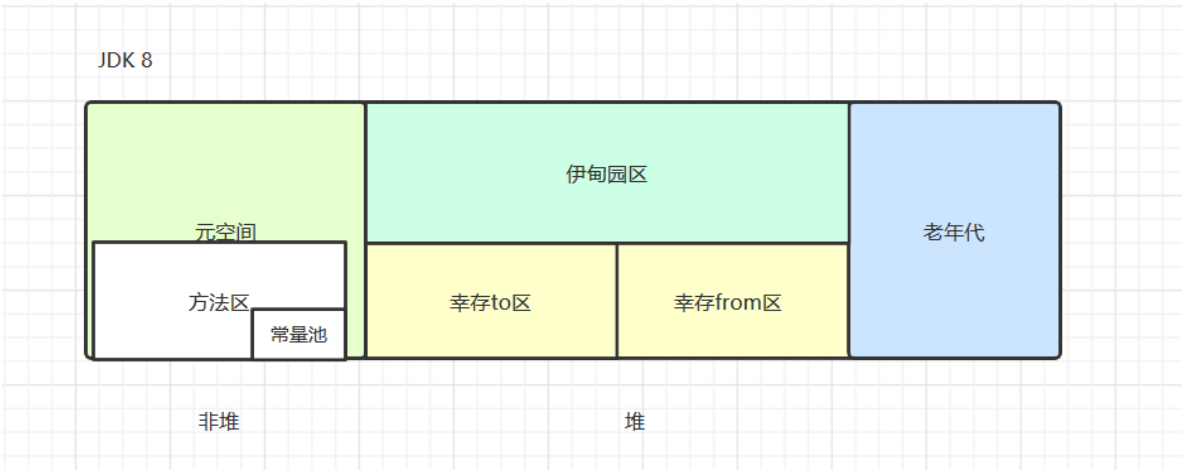
- 什么是堆?
  - 堆是JVM内存管理中最大的一块区域
- 堆的作用是什么?
- 堆的目的就是**为了存放对象实例数据**
- 生命周期
  - 因为大部分对象实例都是存放在堆中,所以JVM启动时,堆就创建了 (注意这里的大部分,不是所有对象都存储在堆中)
  - 又因为线程都要去用对象,因此**堆是线程共享的**
- 堆内存
  - **堆的内存在物理上是可以不连续的,在逻辑上是连续的**
  - 堆内存可以是固定的,也是扩展(-Xmx , -Xms)

堆的内存结构



- 年轻代
  - 伊甸园区(eden)
    - 大部分对象都是伊甸园区被new出来的
  - 幸存to区(Survive to)
  - 幸存from区(Survive from)
- 老年代
- 永久代(JDK8后变为元空间)
  - 常驻内存，用来存放JDK自身携带的Class对象，存储的是Java运行时的一些环境
  - JDK 6之前：永久代，静态常量池在方法区
  - JDK 7： 永久代，慢慢退化，去永久代，将静态常量池移到堆中（字符串常量池也是）
  - JDK 8后： **无永久代，方法，静态常量池在元空间，元空间仍与堆不相连，但与堆共享物理内存，逻辑上可认为在堆中**
  - 不存在垃圾回收，关闭JVM就会释放这个区域的内存
  - 什么情况下永久代会崩：
    - 一个启动类加载大量第三方jar包
    - tomcat部署太多应用
    - 大量动态生成反射类，不断被加载直到内存满，就出现OOM

    因为这些原因容易OOM所以将永久代换成元空间，使用本地内存



元空间：逻辑上存在堆，物理上不存在堆（使用本地内存）

GC垃圾回收主要在**伊甸园区，老年区**

内存调优

堆内存常用参数

指令	作用
-Xms	设置初始化内存大小 默认1/64
-Xmx	设置最大分配内存 默认1/4
-XX:+PrintGCDetails	输出详细的GC处理日志
-XX:NewRatio = 2	设置老年代占堆内存比例 **默认新生代:老年代=1:2**(新生代永远为1,设置的值是多少老年代就占多少)
-XX:SurvivorRatio = 8	设置eden与survivor内存比例 **文档上默认8:1:1实际上6:1:1**(设置的值是多少eden区就占多少)
-Xmn	设置新生代内存大小
-XX:MaxTenuringThreshold	设置新生代去老年代的阈值
-XX:+PrintFlagsInitial	查看所有参数默认值
-XX:+PrintFlagsFinal	查看所有参数最终值

查看堆内存

```
public class HeapTotal {
    public static void main(String[] args) {
        //JVM试图使用最大内存
        long maxMemory = Runtime.getRuntime().maxMemory();
        //JVM初始化总内存
        long totalMemory = Runtime.getRuntime().totalMemory();

        System.out.println("JVM试图使用最大内存-->"+maxMemory+"KB 或"+
(maxMemory/1024/1024)+"MB");
```

```
9           System.out.println("JVM初始化总内存-->"+totalMemory+"KB 或"+
    (totalMemory/1024/1024)+"MB");
10          /*
11          JVM试图使用最大内存-->2820669440KB 或2690MB
12          JVM初始化总内存-->191365120KB 或182MB
13          */
14      }
15  }
```

默认情况下 JVM试图使用最大内存是电脑内存的1/4 JVM初始化总内存是电脑内存的1/64 （电脑内存：12 G）

修改堆内存

> 使用-Xms1024m -Xmx1024m -XX:+PrintGCDetails 执行HeapTotal

```
1   JVM试图使用最大内存-->1029177344B 或981MB
2   JVM初始化总内存-->1029177344B 或981MB
3   Heap
4    PSYoungGen      total 305664K, used 15729K [0x00000000eab00000,
    0x0000000100000000, 0x0000000100000000)
5     eden space 262144K, 6% used
    [0x00000000eab00000,0x00000000eba5c420,0x00000000fab00000)
6     from space 43520K, 0% used
    [0x00000000fd580000,0x00000000fd580000,0x0000000100000000)
7     to   space 43520K, 0% used
    [0x00000000fab00000,0x00000000fab00000,0x00000000fd580000)
8    ParOldGen       total 699392K, used 0K [0x00000000c0000000, 0x00000000eab00000,
    0x00000000eab00000)
9     object space 699392K, 0% used
    [0x00000000c0000000,0x00000000c0000000,0x00000000eab00000)
10   Metaspace       used 3180K, capacity 4496K, committed 4864K, reserved 1056768K
11    class space    used 343K, capacity 388K, committed 512K, reserved 1048576K
```

最好-Xms初始化分配内存与-Xmx最大分配内存一致,因为扩容需要开销

为什么明明设置的是1024m 它显示使用的是981m?

因为幸存from,to区采用复制算法,总有一个幸存区的内存会被浪费

年轻代内存大小 = eden + 1个幸存区 (305664 = 262144 + 43520)

堆内存大小 = 年轻代内存大小 + 老年代内存大小 (305664 + 699392 = 1005056KB/1024 = 981MB)

所以说: **元空间逻辑上存在堆内存，但是物理上不存在堆内存**

模拟堆OOM异常

因为堆是存放对象实例的地方,所以只需要不断的创建对象

并且让 `GC Roots` 到各个对象间有可达路径来避免清除这些对象(因为用可达性分析算法来确定垃圾)

最终就可以导致堆内存没有内存再为新创建的对象分配内存,从而导致OOM

```java
/**
 * @author Tc.l
 * @Date 2020/10/27
 * @Description: 测试堆内存溢出
 */
public class HeapOOM {
    /**
     * -Xms20m 初始化堆内存
     * -Xmx20m 最大堆内存
     * -XX:+HeapDumpOnOutOfMemoryError Dump出OOM的内存快照
     */
    public static void main(String[] args) {
        ArrayList<HeapOOM> list = new ArrayList<>();
        while (true){
            list.add(new HeapOOM());
        }
    }
}

/*
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid17060.hprof ...
Heap dump file created [28270137 bytes in 0.121 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3210)
    at java.util.Arrays.copyOf(Arrays.java:3181)
    at java.util.ArrayList.grow(ArrayList.java:265)
    at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:239)
    at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:231)
    at java.util.ArrayList.add(ArrayList.java:462)
    at 第2章Java内存区域与内存溢出.HeapOOM.main(HeapOOM.java:20)
*/
```

解决这个内存区域的异常的常用思路:

- 确定内存中导致出现OOM的对象是否必要(确定是内存泄漏还是内存溢出)
    - **内存泄漏: 使用内存快照工具找到泄漏对象到GC Roots的引用类,找出泄漏原因**
    - **内存溢出: 根据物理内存试试能不能再把堆内存调大些,减少生命周期过长等设计不合理的对象,降低内存消耗**

方法区

简介

> Method Area

- 什么是方法区?
  - 方法区在逻辑上是堆的一个部分,但在物理上不是,又名"非堆"(Non Heap)就是为了区分堆
- 方法区的作用是什么?
  - 方法区用来存储**类型信息,常量,静态变量,即时编译器编译后的代码缓存**等数据

  也和堆一样可以固定内存也可以扩展
- 生命周期
  - 因为存储了类型信息,常量,静态变量等信息,很多信息线程都会使用到,因此**方法区也是一个线程共享的区域**
- 历史
  - JDK 6 前 HotSpot设计团队使用"永久代"来实现方法区
  - Oracle收购BEA后,想把JRockit的优秀功能移植到HotSpot,但是发现JRockit与HotSpot内部实现不同,没有永久代(并且发现永久代更容易遇到内存溢出问题)
  - JDK 6 计划放弃永久代,逐步改为采用 `本地内存` (Native Memory)来实现方法区
  - JDK 7 把永久代中的 `字符串常量池,静态变量等` **移出到堆中**
    - 为什么要把字符串常量池放到堆中?
      - **字符串常量池在永久代只有FULL GC才可以被回收,开发中会有大量字符串被创建,方法区回收频率低,放在堆中回收频率高**
  - JDK 8 完全废弃永久代,改用与JRockit , J9 一样的方式**采用本地内存中实现的元空间来代替**,把原本永久代中剩下的信息(类型信息)全放在元空间中
- 内存溢出异常
  - 方法区无法满足新的内存分配时,抛出OOM异常

模拟方法区OOM异常

因为方法区的主要责任是用于存放相关类信息,只需要运行时产生大量的类让方法区存放,直到方法区内存不够抛出OOM

使用CGlib操作字节码运行时生成大量动态类

导入CGlib依赖

```
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib-nodep</artifactId>
            <version>3.3.0</version>
        </dependency>
```

```
/*
 * -XX:MaxMetaspaceSize=20m 设置元空间最大内存20m
 * -XX:MetaspaceSize=20m    设置元空间初始内存20m
 */
public class JavaMethodOOM {
    public static void main(String[] args) {
        while (true){
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(JavaMethodOOM.class);
            enhancer.setUseCache(false);
```

```
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }
}

/*
Caused by: java.lang.OutOfMemoryError: Metaspace
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
    ... 11 more
*/
```

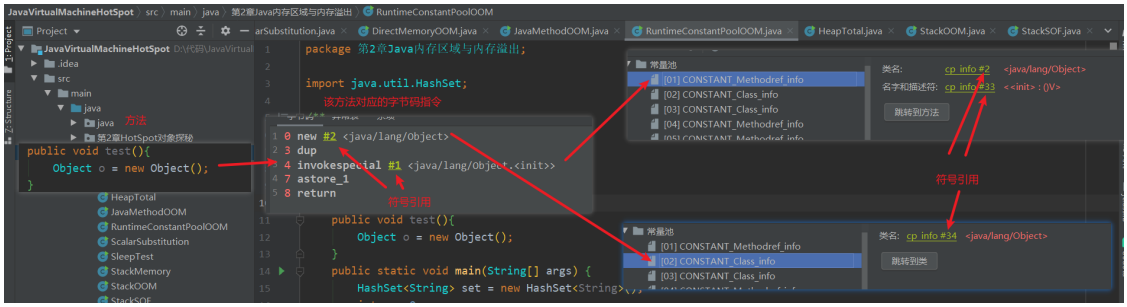
很多主流框架(Spring)对类增强时都会用到这类字节码技术

所以增强的类越多,存放在方法区就越容易溢出

运行时常量池

> Runtime Constant Pool

- 什么是运行时常量池?
  - 运行时常量池是方法区中的一部分
- 运行时常量池的作用是什么?
  - 类加载后,将Class文件中常量池表(Constant Pool Table)中的**字面量和符号引用**保存到运行时常量池中



符号引用:#xx 会指向常量池中的一个直接引用(比如类引用Object)

  - 并且会把**符号引用翻译成直接引用保存在运行时常量池**中
  - **运行时也可以将常量放在运行时常量池(String的intern方法)**

运行时常量池中,**绝大部分是随着JVM运行,从常量池中转化过来的,还有部分可能是通过动态放进来的(String的intern)**

- 生命周期和内存溢出异常
  - 因为是方法区的一部分所以与方法区相同

直接内存

简介

> Direct Memory

直接内存不是运行时数据区的一部分,因为这部分内存被频繁使用,有可能导致抛出OOM

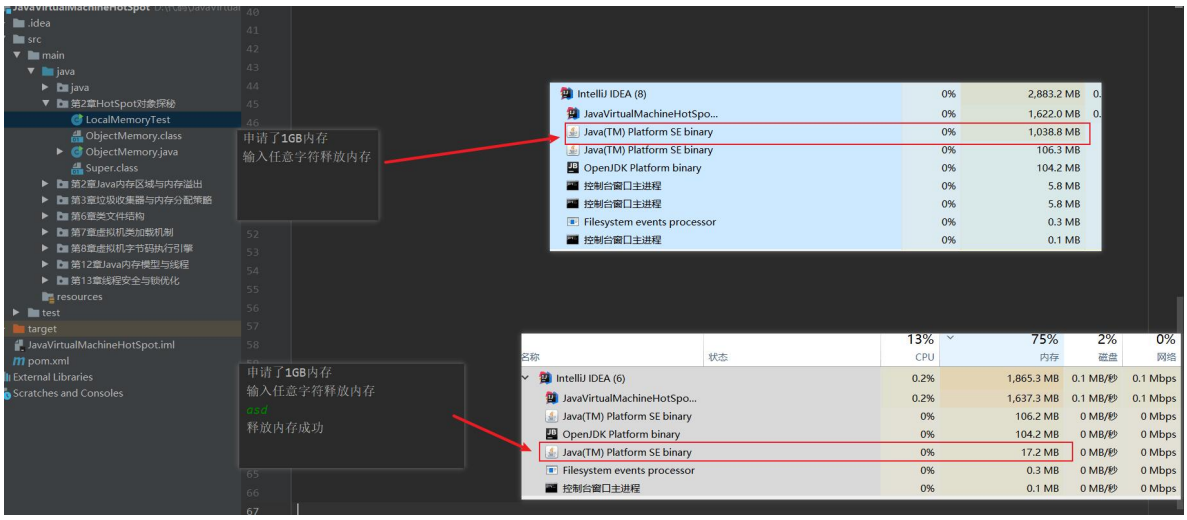
Java1.4加入了NIO类,引入了以 通道传输,缓冲区存储 的IO方式

它可以让本地方法库直接分配物理内存,通过一个在Java堆中 `DirectByteBuffer` 的对象作为这块物理内存的引用进行IO操作 **避免在Java堆中和本地物理内存堆中来回copy数据**

直接内存分配不受Java堆大小的影响,如果忽略掉直接内存,使得各个内存区域大小总和大于物理内存限制,扩展时就会抛出OOM

测试分配直接内存

```
1   public class LocalMemoryTest {
2       private static final int BUFFER = 1024 * 1024 * 1024 ;//1GB
3
4       public static void main(String[] args) {
5           ByteBuffer buffer = ByteBuffer.allocateDirect(BUFFER);
6           System.out.println("申请了1GB内存");
7
8           System.out.println("输入任意字符释放内存");
9
10          Scanner scanner = new Scanner(System.in);
11          scanner.next();
12          System.out.println("释放内存成功");
13          buffer=null;
14          System.gc();
15          while (!scanner.next().equalsIgnoreCase("exit")){
16
17          }
18          System.out.println("退出程序");
19      }
20  }
```



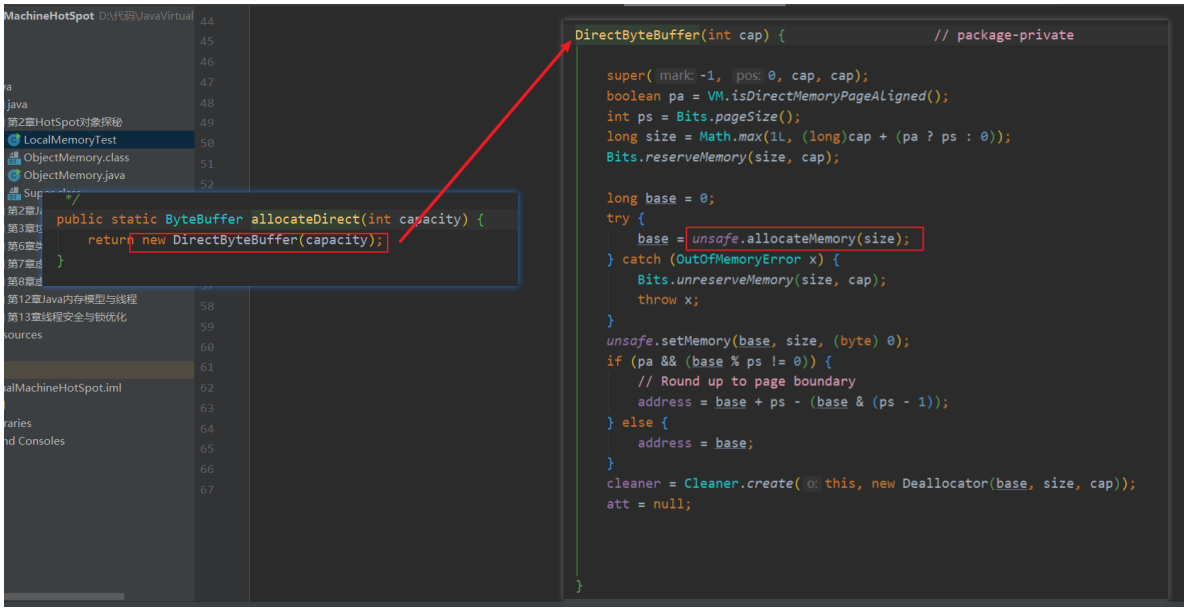
模拟直接内存溢出

默认直接内存与最大堆内存一致

`-XX:MaxDirectMemorySize` 可以修改直接内存

使用NIO中的 `DirectByteBuffer` 分配直接内存也会抛出内存溢出异常,但是它抛出异常并没有真正向操作系统申请空间,只是通过计算内存不足,自己手动抛出的异常

真正申请分配直接内存的方法是Unsafe::allocateMemory()




```java
/* 测试直接内存OOM
 * -XX:MaxDirectMemorySize=10m
 * -Xmx20m
 */
public class DirectMemoryOOM {
    static final int _1MB = 1024*1024;
    public static void main(String[] args) throws IllegalAccessException {
        Field declaredField = Unsafe.class.getDeclaredFields()[0];
        declaredField.setAccessible(true);
        Unsafe unsafe  =(Unsafe) declaredField.get(null);
        while (true){
            unsafe.allocateMemory(_1MB);
        }
    }
```

```
15    }
```

由直接内存出现的OOM的明显特征就是:Dump 堆快照中,没有什么明显的异常

如果是这种情况,且使用了NIO的直接内存可以考虑这方面的原因

本地方法接口与本地方法库

本地方法: 关键字 `native` 修饰的方法,Java调用非Java代码的接口

注意: `native` 不能和 `abstract` 一起修饰方法

> 为什么需要本地方法

1. Java需要调用其他语言 (C,C++等)
2. Java要与操作系统交互 (JVM部分也是由C实现)

本地方法很少了,部分都是与硬件有关(比如启动线程 `start0()`)

只是部分虚拟机支持本地方法

> 本地方法接口

本地方法通过本地方法接口来访问虚拟机中的运行时数据区

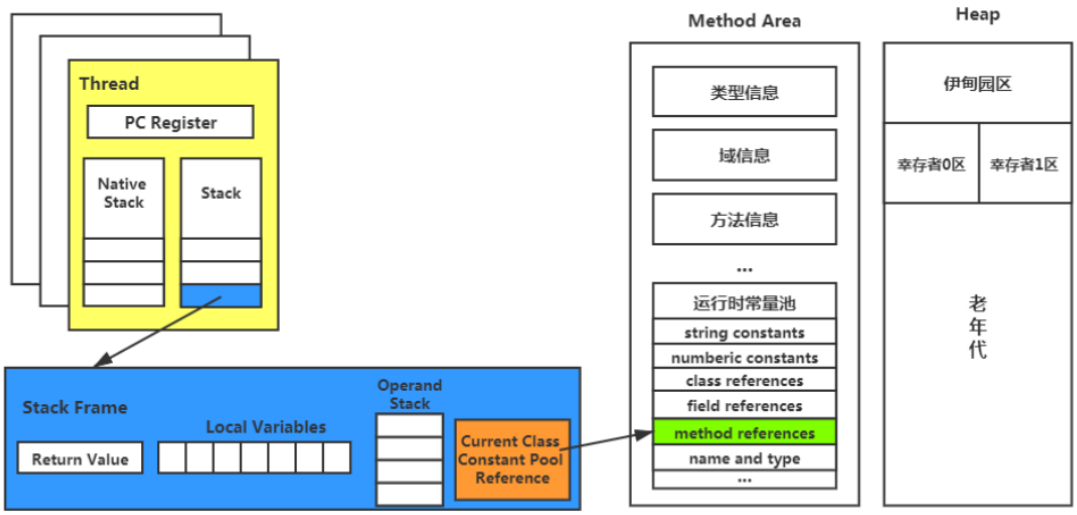
某线程调用本地方法时,它就不受虚拟机的限制,在OS眼里它和JVM有同样权限

可以直接使用本地处理器中的寄存器,直接从本地内存分配任意内存

> 本地方法库

本地方法栈中登记 `native` 修饰的方法,由执行引擎来加载本地方法库

总结


Thread
PC Register
Native Stack
Stack
Stack Frame
Return Value
Local Variables
Operand Stack
Current Class Constant Pool Reference
Method Area
类型信息
域信息
方法信息
...
运行时常量池
string constants
numberic constants
class references
field references
method references
name and type
...
Heap
伊甸园区
幸存者0区
幸存者1区
老年代