jvm

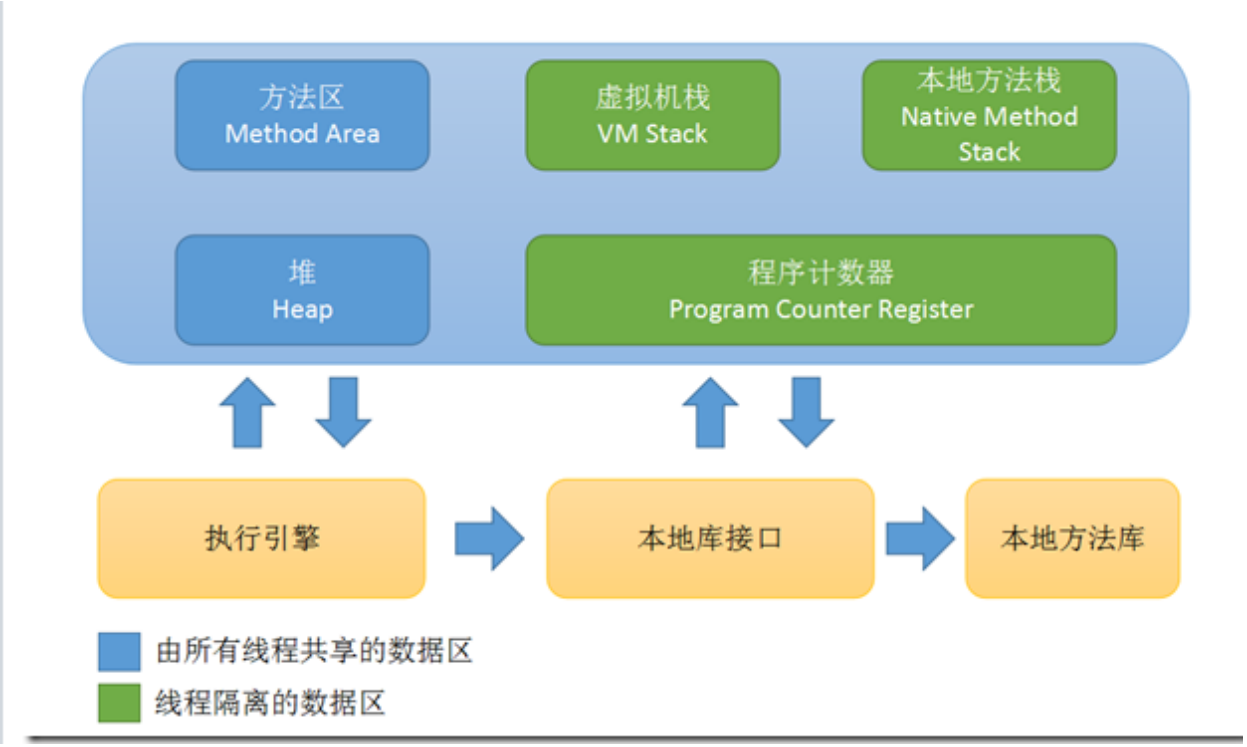
- jvm
  - 1 内存区域与溢出异常
    - 1.1 运行时数据区域
      - 程序计数器
      - java虚拟机栈
      - 本地方法栈
      - java堆
      - 方法区
      - 运行时常量池
      - 直接内存
    - 1.2 HotSpot虚拟机对象探秘
      - 对象的创建
      - 对象的内存布局
      - 对象的访问定位
    - 1.3 异常分析
      - java堆异常
      - 虚拟机栈和本地方法栈溢出
      - 方法区和运行时常量池溢出
      - 本机直接内存溢出
  - 垃圾收集器与内存分配策略
    - 对象状态判断
      - 引用计数法
      - 可达性算法分析
      - 引用的分类
      - 对象的死亡过程
      - 回收方法区
    - 垃圾收集算法
      - 标记-清除算法
      - 复制算法
      - 标记-整理算法
      - 分代收集算法
    - HotSpot的算法实现
      - 枚举根节点
      - 安全点
      - 安全区域
    - 垃圾收集器
      - Serial收集器

        - ParNew收集器
        - Parallel Scavenge收集器
        - Serial Old收集器
        - Parallel Old收集器
        - CMS收集器
        - G1收集器
    - 内存分配与回收策略
- 虚拟机性能监控与故障处理工具
    - jdk的命令行工具
    - jdk的可视化工具
        - jConsole：java监视与管理控制台
        - VisualVM:多合一故障处理工具
- 类文件结构
    - class类文件结构
        - 魔数
        - 常量池
        - 访问标志
        - 类索引、父类索引和接口索引集合
        - 字段表集合
        - 方法表集合
        - 属性表集合
    - 字节码指令简介
        - 字节码与数据类型
- 虚拟机类加载机制
    - 类加载的时机
    - 类加载的过程
        - 加载
        - 验证
        - 准备
        - 解析
        - 初始化
    - 类加载器
        - 双亲委派模型
        - 破坏双亲委派模型
- 虚拟机字节码执行引擎
    - 运行时帧栈结构
        - 局部变量表
        - 操作数栈
        - 动态连接
        - 方法返回地址
        - 附加信息

        - 方法调用
            - 解析调用
            - 分派调用
            - 动态类型语言
        - 基于栈的字节码解释执行引擎
            - 解释执行
            - 基于栈的指令集与基于寄存器的指令集
    - 类加载即执行子系统的案例(以tomcat为例)
    - java内存模型与线程
        - 概述
        - java内存模型
            - 主内存与工作内存
            - 内存间交互操作
            - volatile型变量的特殊规则
        - java与线程
            - 线程的实现
            - 线程调度
            - 线程的状态
    - 线程安全与锁优化
        - 线程安全
            - 线程安全的实现方法
        - 锁优化
            - 自旋锁与自适应自旋
            - 锁消除
            - 锁粗化
            - 轻量级锁
            - 偏向锁

1 内存区域与溢出异常

1.1 运行时数据区域



程序计数器

程序计数器是一块较小的内存空间，可以看做当前线程所执行的字节码的行号指示器。在虚拟机概念中，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都是依赖这个计数器来实现。

由于java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程间计数器互不影响，独立存储。即“线程私有”内存。

如果线程此时执行的是一个java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址。如果执行的是native方法，则计数器为空。

该内存区域是唯一一个在java虚拟规范中没有规定任何OutOfMemoryError的情况。

java虚拟机栈

它也是线程私有的，即生命周期与线程相同。描述的是java方法在执行时的内存模型，每个方法执行时都会创建一个帧帧（stack frame）用于存储局部变量表，操作数帧，动态链接，操作出口信息等。方法的调用到执行，就对应一个栈帧在虚拟机栈中的入栈和出栈。

通常我们所说的栈内存，即是指虚拟机栈中的局部变量部分：各种基本数据类型，对象引用等。

比如64位的long和double类型的数据会占用2个局部变量空间（slot），其余数据类型只占一个。局部变量所需空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间时确定的，运行期间不会改变。

这个区域两种异常：线程请求的栈深度大于虚拟机栈所允许的深度，抛出StackOverflowError异常。如果虚拟机栈深度可以动态扩展，但是申请内存失败，抛出OutOfMemoryError。

本地方法栈

本地方法栈与虚拟机栈锁发挥的作用非常相似，只不过虚拟机栈为虚拟机执行java方法（也就是字节码），而本地方法栈则为虚拟机使用到的native方法服务。

抛出异常也是StackOverflowError的异常和OutOfMemory的异常。

Native Method就是一个java调用非java代码的接口。

java堆

java虚拟机中所占内存最大的一块。

线程共享的，在虚拟机启动时就被创建。

此内存区域的目的是存放对象实例。所有对象实例和数组都要在堆上分配。

Java对是垃圾收集管理的主要区域，也称GC堆，从内存回收的角度来看（根据分代收集算法）分为新生代和老生代。细分还可以分为Eden空间，From Survivor空间，To Survivor空间等。

从内存分配上看：线程共享的java堆可以分出多个线程私有的分配缓冲区（但存的也都是实例）。

java堆在物理上是不连续的内存空间，逻辑上连续。实现时，可以固定大小，也可以是可扩展的，如果堆中没有内存分配实例了，且堆无法扩展，抛出OutOfMemoryError。

方法区

线程共享，用于存储已被java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。

之前是将GC分带扩展至方法区，就是使用永久代来实现方法区，这样HotSpot的垃圾收集器可以像管理java堆一样管理这部分内存。但现在已经逐渐放弃这种方法，改为使用Native Memory来实现方法去规划。

java虚拟机规范对方法区的限制很宽松，类似java堆，不需要使用连续的物理内存和可扩展，还可以选择不识闲垃圾收集。垃圾收集行为在这个区比较少，同时也比较难。当方法去无法满足内存分配需求时，将抛出OutOfMemory的错误。

运行时常量池

属于方法区的一部分，class文件中除了有类的版本、字段、方法、接口等信息以外。还有一项信息是常量池，用于存放编译器生成的各种字面量和符号引用，这部分内容在类加载后进入方法区的运行时常量区中。

运行时常量池相对于Class文件常量池的特点是动态性，即并非预置入Class文件常量池中的内容才能进入方法去运行时常量池，运行期间也可以将新的常量放入池中。比如String类的intern()方法

常量池无法申请到内存时，报OutOfMemoryError的错误

直接内存

并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。

JDK1.4加入了NIO，引入一种基于通道与缓冲区的I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。因为避免了在Java堆和Native堆中来回复制数据，提高了性能。

当各个内存区域总和大于物理内存限制，抛出OutOfMemoryError异常。

1.2 HotSpot虚拟机对象探秘

对象的创建

- 虚拟机遇到new指令时，先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就先执行类加载过程。
- 类加载检查通过后，为新生对象进行内存分配。所需内存大小在类加载完后便可以确定，即从java堆中划出一块确定大小的内存用来保存新new的对象。
    - 在分配内存的时候如果java堆中的内存是规整的（即已使用的内存全部在一边，未使用的内存全部在另一边），则采取“指针碰撞”分配内存————指针移动对象大小的内存空间；如果java堆中的内存不是规整的，则采取“空闲列表”分配内存————根据表中空闲区域记录的大小进行分配。而java堆中的是否规整由所采用的垃圾收集器有关。
    - 内存分配时的并发问题，第一个是虚拟机采用CAS配上失败重试的方法保证更新操作的原子性。或者将内存分配的动作按照线程划分在不同的空间中国能进行，即每个线程在java堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB）。哪个线程需要分配内存，就在哪个线程上的TLAB上分配，只有TLAB用完并重新分配TLAB时才同步锁定（TLAB的使用时可选的）。
- 内存分配完成后，虚拟机需要将分配到的内存空间都初始化为0值，这一步操作保证了对象的实例字段在java代码中可以不赋初始这就直接使用，程序能直接访问到这些字段的数据类型对应的零值。（如果使用TLAB，这一工作在TLAB分配时进行）
- 对对象进行必要的设置，如该对象是哪个类的实例等。这些信息保存在对象的对象头中。
- 虚拟机新建完成后，调用init方法进行对象创建，将对象引入栈。

对象的内存布局

对象在内存中的存储布局分为3块区域：对象头，实例数据和对齐填充。

对象头：

1. 第一部分主要存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志等等。
    - 由于对象头的数据过多，但与对象自身定义无关，所以通常被设计成一个非固定的数据结构以便在极小的空间内存内以存储更多的信息。
2. 第二部分是指针，即对象指向他的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪一个类的实例。
    - 但这个指针不是必需的，也就是说查找对象的元数据信息可以不通过对象本身。

实例数据：实例数据是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容，无论是从父类继承的还是子类定义的，都需要记录起来。

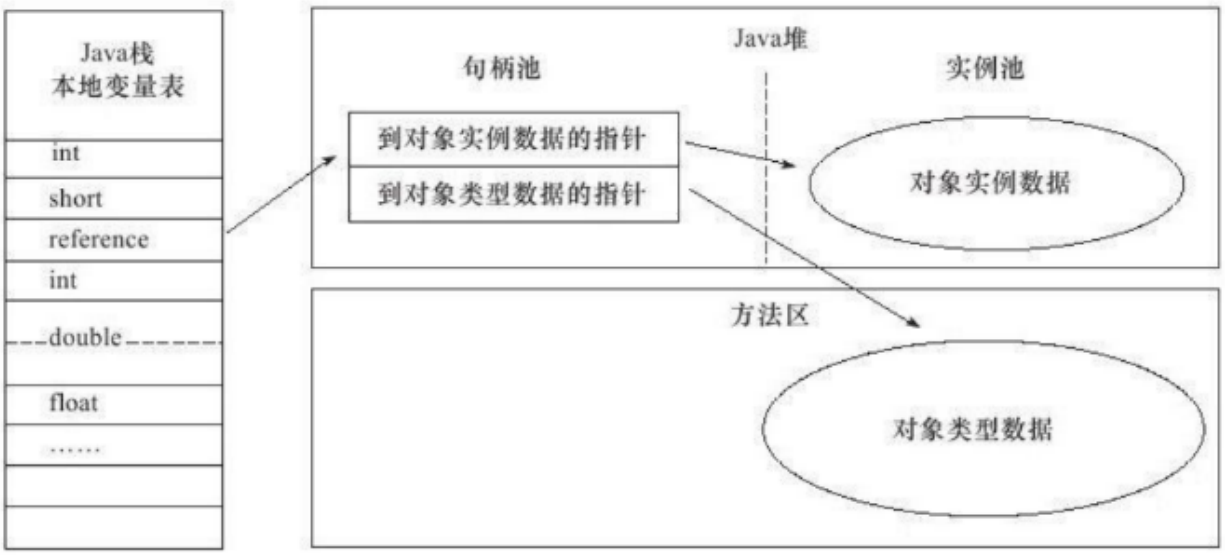
存储顺序收到虚拟机分配策略参数和字段在java源码中定义顺序的影响。（父类在子类前，相同宽度的字段在一起）

对齐填充： 不是必然存在的，仅仅是其占位符的作用，保证对象大小必须是8字节的整数倍。

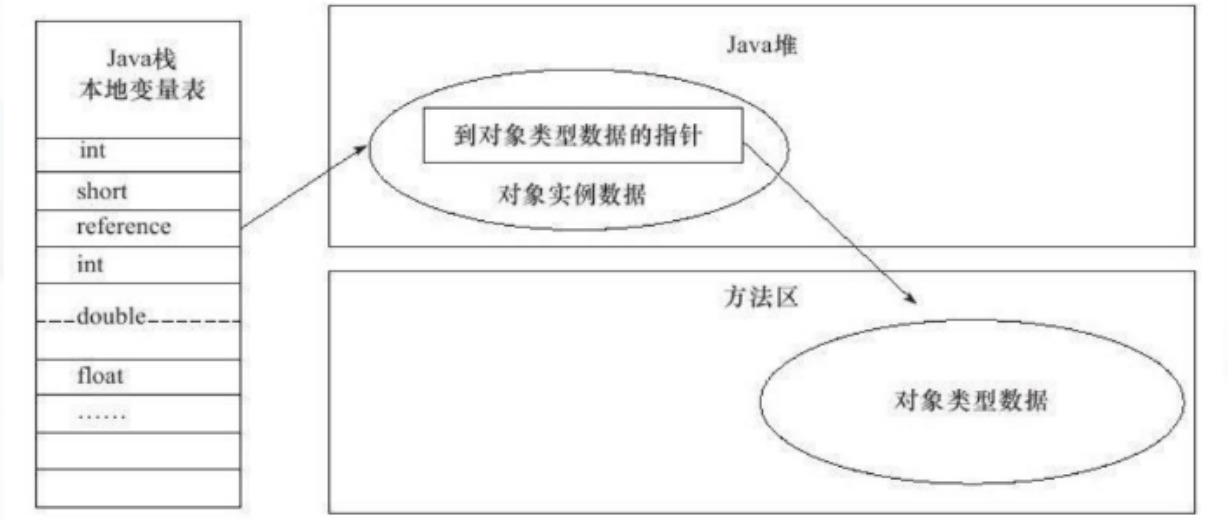
对象的访问定位

我们java程序通过栈上的reference数据来操作堆上的具体对象。由于reference类型在java虚拟机中只规定了一个指向对象的引用，那如何通过这个引用去定位对象的具体位置。 有两种主流方法，使用句柄和直接指针。

使用句柄 java堆中划分出一块内存作为句柄池，那reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。



直接指针 如果使用直接指针访问，那么java堆对象的布局中就必须考虑如何防止访问类型数据的相关信息，而reference中存储的直接就是对象地址。



比较分析 句柄访问的最大好处就是reference中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改

直接指针访问的最大好处就是速度更快，节省了一次指针定位的时间开销，由于对象的访问在java中非常频繁，因此这类开销也较大。

主要虚拟机HotSpot使用的是直接指针访问方式。

1.3 异常分析

java堆异常

java堆用于存储对象实例，只要不断地创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么对象数量到达最大堆容量的限制后会产生内存溢出异常。

虚拟机栈和本地方法栈溢出

- 如果线程请求的栈深度大于虚拟机所允许的栈深度，抛出StackOverflowError异常
- 虚拟机在扩展栈时申请不到足够空间，抛出OutOfMemoryError异常。

这两个异常本质是有重复的。

```
public class javaVMStackSOF{
    private int stackLen = 1;

    public void stackLeak(){
        stackLen++;
        stackLeak();
    }

    public static void main(String args[]){
        javaVMStackSOF test = new javaVMStackSOF();
        try{
            test.stackLeak();
        }catch(Throwable e){
            System.out.println(test.stackLen);
            throw e;
        }
    }
}
```

多线程的线程过多会产生SOF的异常。

方法区和运行时常量池溢出

用intern方法来测试运行时常量池区溢出(intern的作用是判断常量池中是否包含这个string对象，没有就新建并返回引用，有就返回这个对象的字符串)

```
/*
 * VM Args: -XX:PermSize=10m -XX:MaxPermSize=10m
 */
public class RuntimeConstantPoolOOM {
    public static void main(String[] args) {
```

```
        // 使用list保持着常量池引用，避免Full GC回收常量池行为
        List<String> list = new ArrayList<String>();

        int i = 0;
        while (true) {
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

PermGen space报错说明运行时常量池属于方法区。

这里还有jdk1.6和1.7的一点关于intern方法的区别，1.6是首次遇到的字符串实例赋值到永久代中，返回的是永久代实例的引用。而1.7不是赋值实例，二是记录常量池首次出现的实例引用，如果已经出现过则返回之前的引用。

本机直接内存溢出

DirectMemory 直接内存容量可以进行指定（或默认与java堆最大值一样）。是一个堆外内存，通过在Java堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。

这块代码越过了DirectByteBuffer类，通过反射获取实例进行内存的分配。

有directMemory导致的内存溢出，在Heap Dump中看不到明显异常。

```
//测试代码如下，运行时添加参数 -Xmx20M -XX:MaxDirectMemorySize=10M 设置降低直接内存的空间来加快异常的抛出
public class DirectrMemoryOOM {
    private static final int _1M = 1024 * 1024;

    public static void main(String[] args) throws Exception {
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        @SuppressWarnings("restriction")
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while(true) {
            unsafe.allocateMemory(_1M);
        }
    }

}
```

不断申请1M的空间，最终会耗尽内存抛出 OutOfMemoryError 异常。

垃圾收集器与内存分配策略

概述：程序计数器，虚拟机栈和本地方法栈的生命周期都是随线程一起，所以这几个区域的内存分配和会后都具备确定性。但java堆和方法区的对象创建，内存分配以及回收都是动态的。我们指的内存分配和垃圾收集也主要是这一块的。

对象状态判断

引用计数法

给对象添加一个引用计数器，每当一个地方引用它时，计数器值就加1；当引用失效时，计数器值减1。当计数器为0的对象就不可能再被使用。

缺点：无法解决对象之间相互循环引用的问题。

```
public class Main {

    private Object instance;

    public Main() {
        byte [] m = new byte[20*1024*1024];
    }
    public static void main(String[] args) {
        Main m1 = new Main();
        Main m2 = new Main();

        m1.instance = m2;
        m2.instance = m1;

        m1 = null;
        m2 = null;

        System.gc();
    }
```

在这里，m1,m2是存在于堆中的两个对象实例，然后m1.instance引用了m2，m2.instance引用了m1.当令他们=null的时候，其实他们的引用计数器就应该为0（栈中的应用并没有指向堆中的相应对象了），但被互相的实例引用，所以导致对象的引用计数器不为0，按该算法的时候不会进行垃圾回收

可以利用弱引用的方法来解决这个问题

可达性算法分析

即通过一系列称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain),当一个对象到GC Roots没有任何引用链相连，则证明此对象不可用，判定为可回收对象。

作为GC Roots的对象：

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI（即Native方法）引用的对象

引用的分类

引用的初始定义：如果reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。

- 强引用：类似"Object obj=new Object()"这类的引用，只要强引用还存在，垃圾收集器用于不会回收被引用的对象。
  - Object obj将反映到Java栈的本地变量表，这是一个本地变量的定义。是一个引用类型。
  - new Object()将会反映在Java堆中。存储了Object类型的所有实例数据值（次内存是不固定大小的，因为谁也无法确定这是对象的大小）。
  - 程序运行，类型信息已经加载到内存里，这些数据就在Java方法区中，包括：类型的父类型，实现的接口、包含的方法等类型信息。new Object()，根据这些信息建立对象。可以看到这些信息是线程共享的。
- 软引用：用来描述一些还有用但并非必需的对象。软引用关联的对象，系统会在即将发生内存溢出之前才会对这些对象进行二次会后，若是回收后还是内存不够，就会抛出内存溢出异常。SoftReference类来实现软引用。
- 弱引用： 也是用来描述非必需对象，强度比软引用弱。关联的对象只能生存岛下一次垃圾回收之前，无论内存是否够用，弱引用关联对象都会被回收。WeakReference类
- 虚引用：最弱的引用关系，唯一目的就是让这个对象被收集器回收时收到系统通知。PhantomReference类。

对象的死亡过程

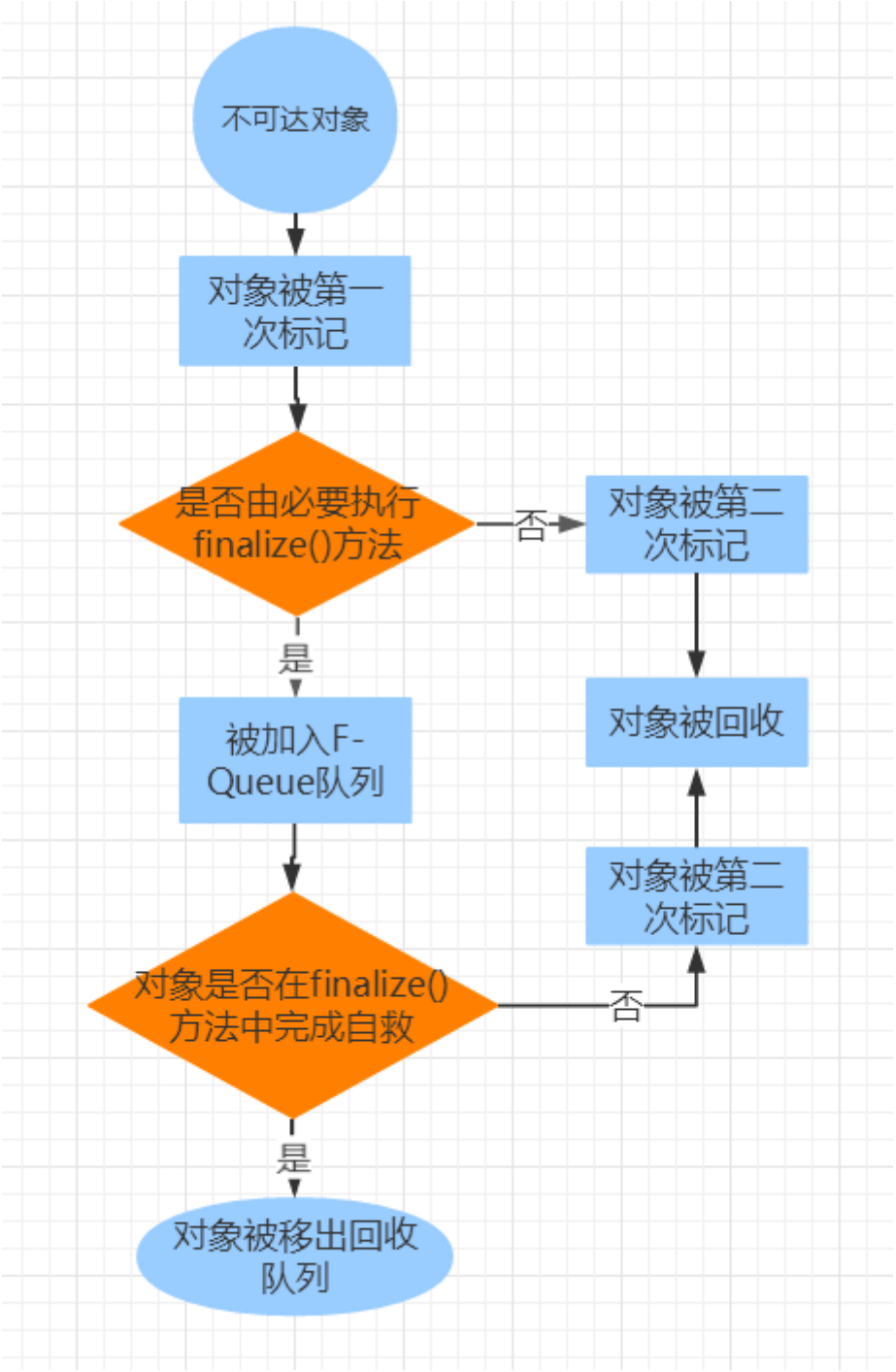
一个对象的真正死亡，至少要经历两次标记，第一次是经过可达性分析，发现GC Roots无法通过引用链到达该对象。

第二次标记的条件是次对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，则视为“没有必要执行”，进行第二次标记，然后死亡。

在F-Queue中会“执行”finalize方法，但只是触发，为了避免finalize方法执行缓慢或者死循环等情况，不一定会执行完。但同理可以在finalize进行“自救”----重新与引用链上的任何一个对象建立关联，比如把自己(this关键字)赋值给某个类变量或者对象的成员变量。那么第二次标记时会被移出。

```
public class FinalizeEscape {

    public static FinalizeEscape FC = null;

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize method invoke");
        FC = this;
    }

    public static void main(String[] args) throws Exception {
        FC = new FinalizeEscape();

        FC = null;
        System.gc();
```

```
        Thread.sleep(500);
        if(FC != null)
            System.out.println("i am alive");
        else
            System.out.println("i am dead");

        FC = null;
        System.gc();
        Thread.sleep(500);
        if(FC != null)
            System.out.println("i am alive");
        else
            System.out.println("i am dead");
    }
}
---------------------
finalize method invoke
i am alive
i am dead --- finalize()的方法只会被虚拟机调用一次
```



回收方法区

方法区（HotSpot虚拟机中的永久代）中垃圾回收的主要对象是：废弃常量和无用的类。

回收废弃常量：当没有任何对象引用方法区中的某个常量，且也没有其他地方引用了这个字面量，这个常量就可以被回收，其他接口，字段符号也类似。

无用的类：

- 该类所有的实例都已被回收，也就是java堆中不存在该类的任何实例
- 加载该类的classloader已经被回收

- 该类对应的java.lang.Class对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法

在可回收的前提下，HotSpot还会查看类加载和卸载信息等来判断是否进行回收。

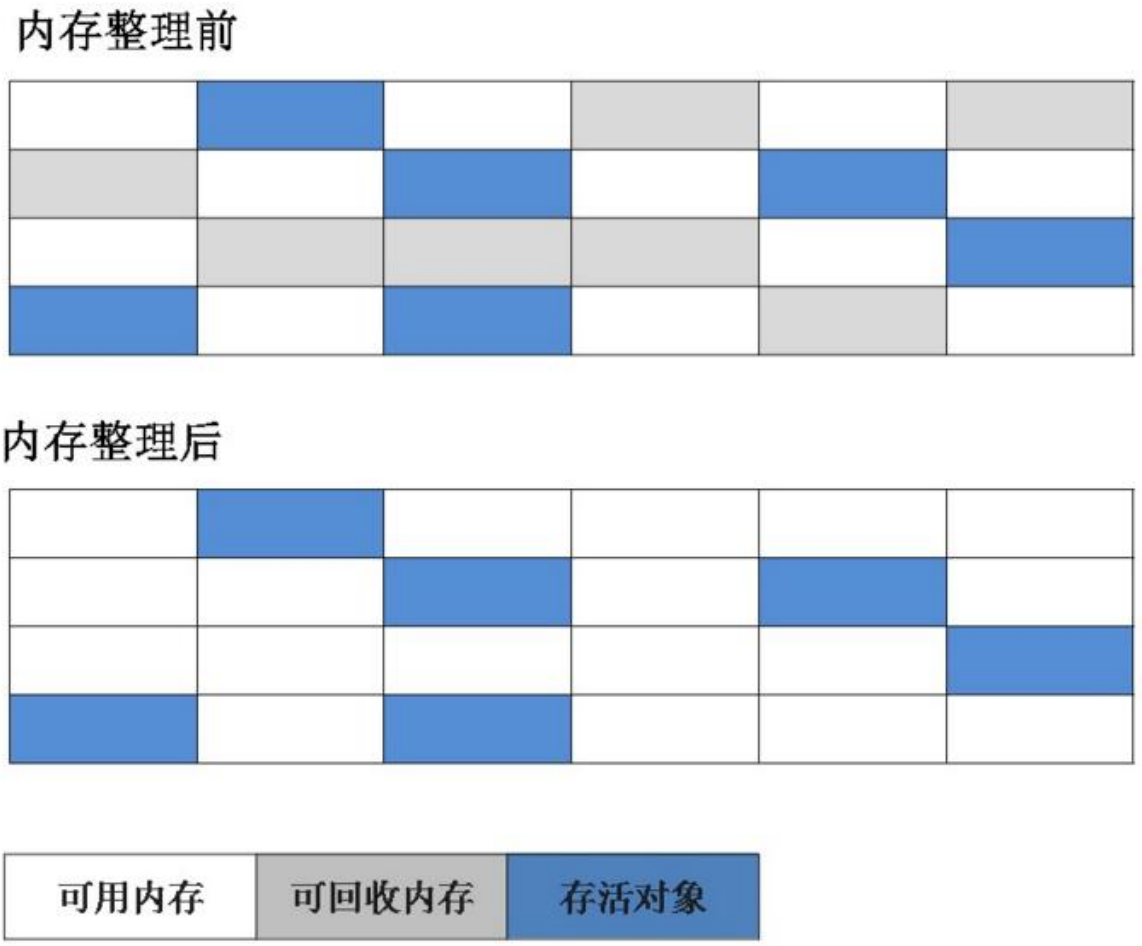
垃圾收集算法

标记-清除算法

1.先对可回收的对象进行标记， 2.在标记完成后进行清楚

缺点：

- 效率问题，标记和清除的效率不高
- 空间问题：标记清除后会产生大量的不连续内存碎片，空间碎片太多可能会导致以后在程序运行时需要分配较大对象时，因为找不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。

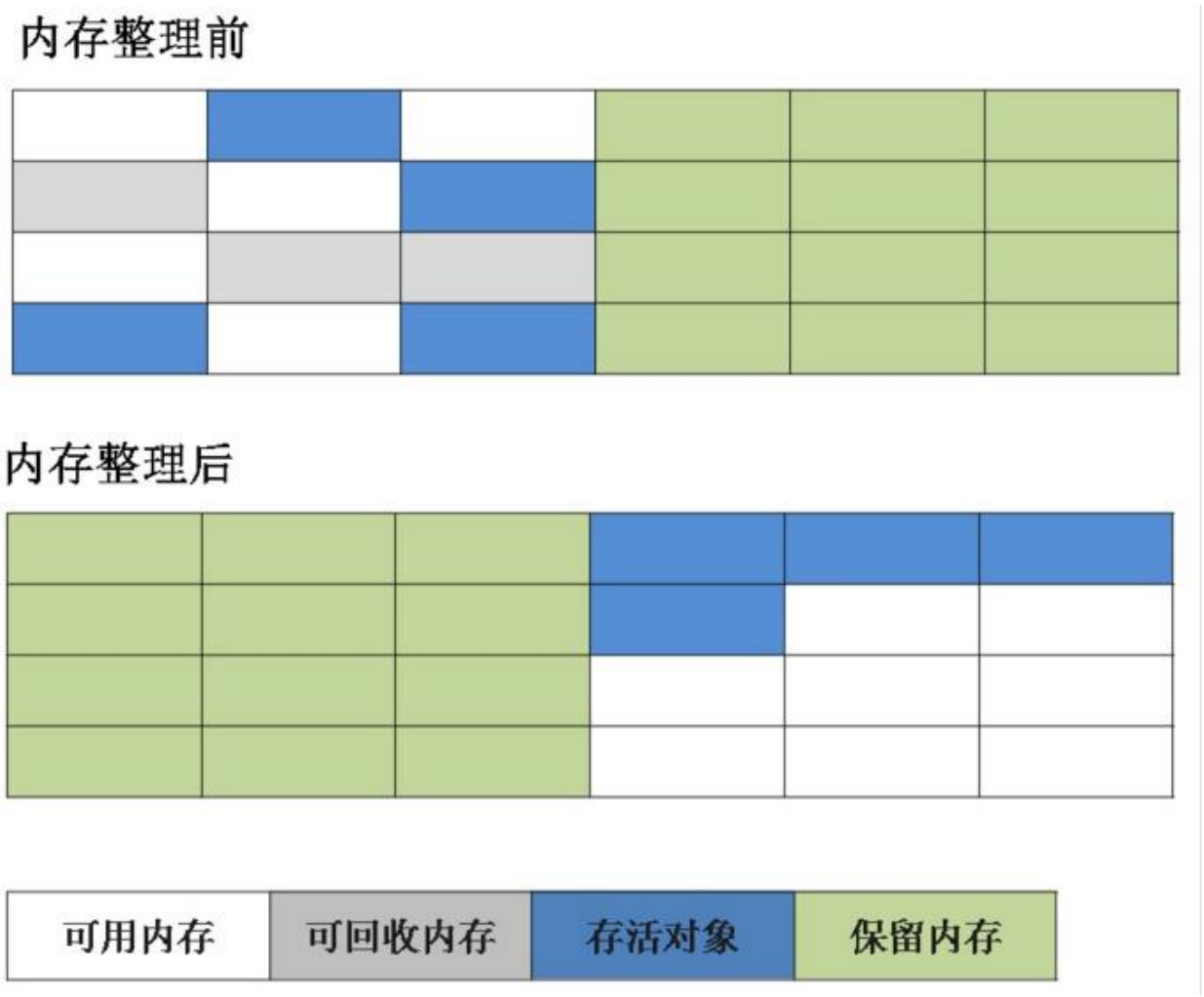


复制算法

将容量分为大小相等的两块，每次只使用其中的一块，当这一块用完了，将还存活的对象复制到另一块上，然后把已使用的内存空间一次清理掉。

优点是内存分配不用考虑内存碎片等复杂情况，只需要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

缺点是算法的代价将内存缩小为原来的一半。代价过大



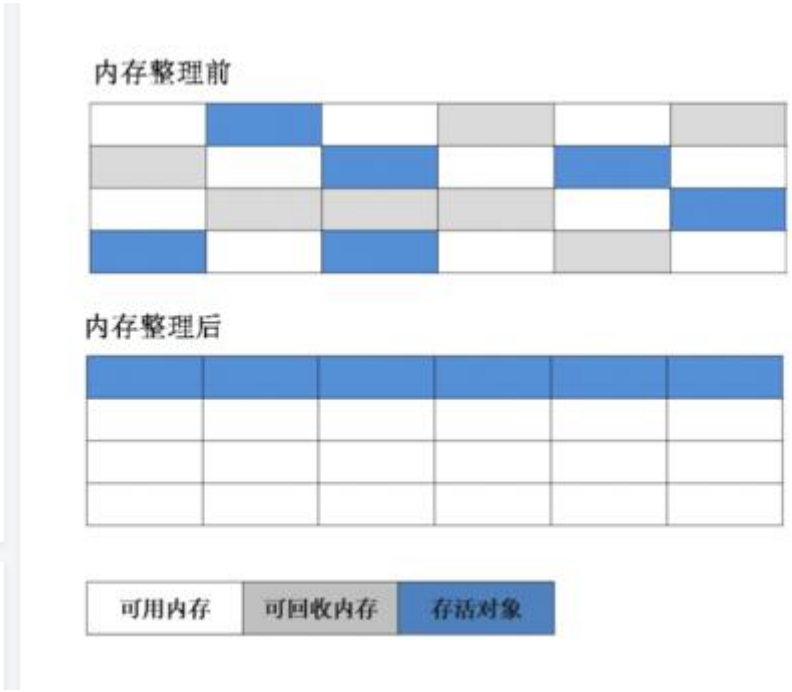
而现在发现新生代中98%的对象都是“朝生夕死”的。于是我们不按1:1来划分。将内存分为一块较大的Eden区和两块较小的Survivor区。每次使用Eden和一块Survivo区。当回收时，将其还活着的对象一次性复制到另一个空白的Survivor区。

默认Eden:Survivor=8:1。也就是新生代可用内存为整个新生代容量的90%，只有10%的内存会被浪费。但我们无法保证每次回收都只有不多余10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（老年代）进行分配担保，也就是多于10%的对象，会进入老年代。

年老代里存放的都是存活时间较久的，大小较大的对象，因此年老代使用标记整理算法。当年老代容量满的时候，会触发一次Major GC（full GC），回收年老代和年轻代中不再被使用的对象资源。

标记-整理算法

- 第一步进行标记
- 第二步不是直接对可回收对象进行清除，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。



分代收集算法

堆大小=新生代+老生代

新生代：每次垃圾收集时都有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成垃圾收集。

老生代：对象存活率高，没有额外的空间对它进行分配担保，就必须使用“标记-清理”或者“标记-整理”算法进行收集。

HotSpot的算法实现

这里主要探讨GC是如何实现的。枚举根节点说的是如何找到与GC Roots相连的节点。 安全点讲的是进行GC的条件和时间。 安全区域指的是没获得CPU时，是如何进行GC的。

枚举根节点

可达性分析中，可以作为GC Roots的节点主要在全局性的引用（例如常量或者静态属性）与执行上下文中（例如帧栈中的本地变量表）中。但逐个检查这里面的引用消耗时间过大。

同时，可达性分析工作必须在一个能确保一致性的“快照”中进行，不可以出现分析过程中，对象的引用关系还发生变化的情况。这点也是导致GC进行时必须停顿所有Java执行线程的一个重要原因。

然后在主流的java虚拟机使用的都是准确式GC，当执行系统停下来时，虚拟机是通过一个叫OopMAap的的数据结构来直接得知哪些地方存放着对象引用，而不是去全内存的去找每一个可能的节点。

安全点

在OopMap的协助下，HotSpot可以快速且准确地完成GC Roots枚举，但为每个指令都生成一个oopmap的话，需要空间过大。

实际上，并不是为每条指令都生成了oopmap，而是在特定的位置记录这些信息，这些位置称为“安全点”

即程序只有在安全点暂停才可以开始GC。

安全点的选择，既不能太少以至于让GC太久才进行一次，也不能太多以至于过分增大运行时负荷。安全点的选择是以程序“是否具有让程序长时间执行的特征”为标准进行选定。而长时间执行的最明显特征就是指令序列复用，如方法调用，循环跳转，异常跳转等。

最后是如何在GC发生时，所有线程都跑到安全点上

- 抢先式中断：当GC发生时，先把所有线程中断，如果发现线程中断的地方不在安全点，就恢复，让它跑到安全点。---现在不再使用
- 主动式中断：当GC需要中断线程时，不直接对线程操作，而是仅标志位置，各个线程执行时主动去轮询这个标志，发现中断标志为真就自己中断挂起。

安全区域

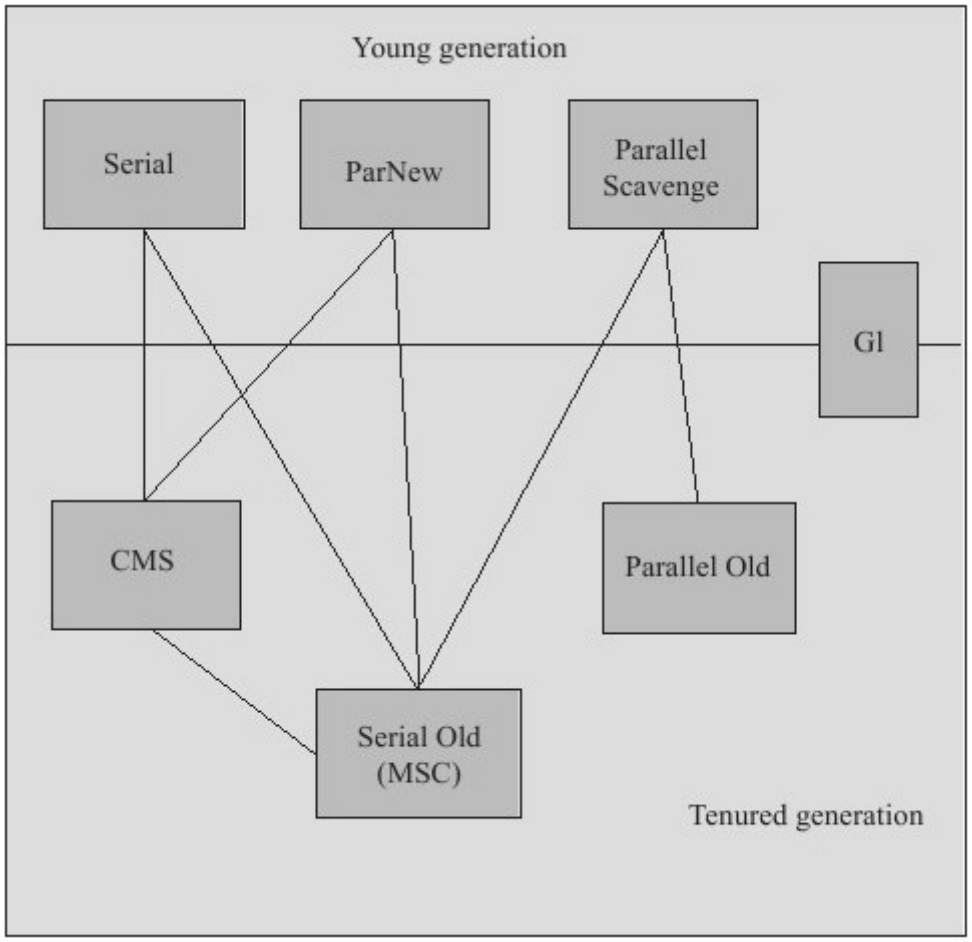
解决的是当没有分配CPU时间，无法响应JVM的中断请求，如何进行GC呢？

安全区域是指在一段代码片段中，引用关系不会发生变化。在这个区域内任意地方开始GC都是安全的，我们也可以把安全区域理解为扩展的安全点。

若是线程要离开安全区域了，但系统在安全区域内没有执行完根节点枚举，那线程就要继续等待，直到收到可以安全离开安全区域的信号为止。

垃圾收集器

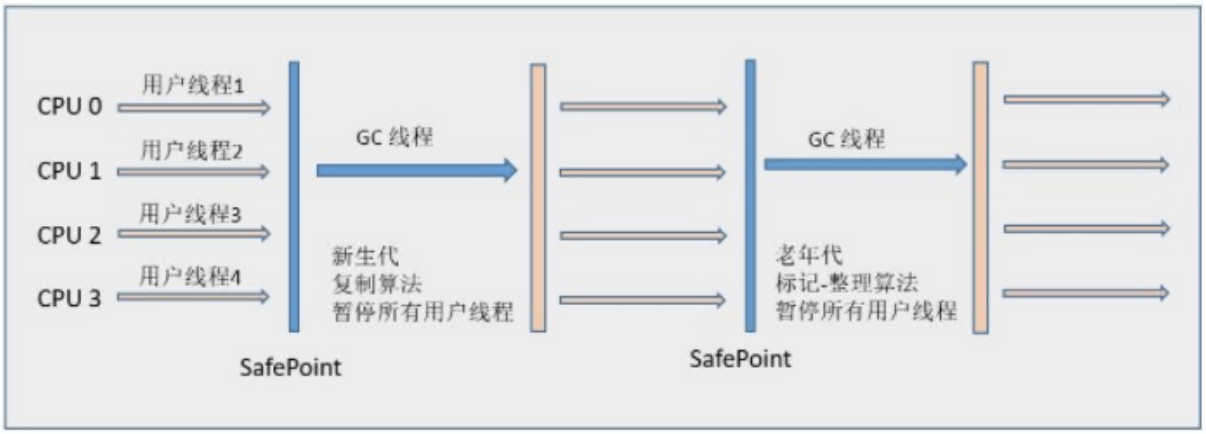
收集算法是内存回收的方法论，垃圾收集器是内存回收的具体实现



Serial收集器

Serial收集器是一个单线程的收集器，，是指它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

优点:简单而高效，对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集可以获得最高的单线程收集效率。

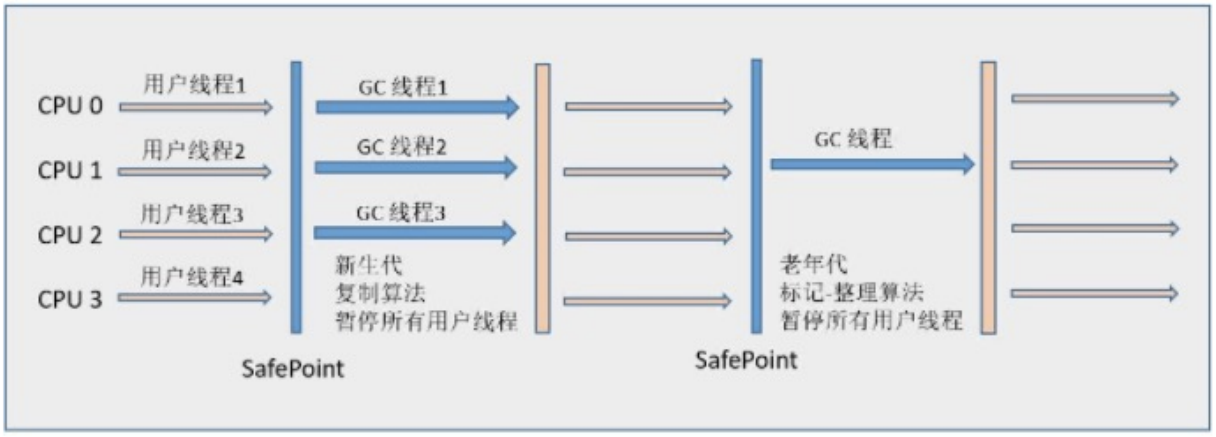


ParNew收集器

其实就是Serial收集器的多线程版本。在收集算法、对象分配规则等各个方面都与Serial收集器完全一样

特点只有他能在新生代与CMS收集器进行配合。

ParNew收集器在单CPU环境中，由于线程交互的开销，表现没有Serial收集器好，但随着CPU数量的增多，对于GC时系统资源的有效利用还是有好处的。



Parallel Scavenge收集器

并行的多线程收集器，使用的是复制算法。

特点是这个收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，达到一个可控制的吞吐量。吞吐量是CPU用于运行用户代码的时间与CPU总消耗时间的比值。吞吐量=运行用户代码时间/(运行用户代码时间+垃圾回收时间)

Paralle Scavenge收集器提供两个参数用于精确控制吞吐量：

- MaxGCPauseMills最大垃圾收集停顿时间 ：参数允许的是一个大于0的毫秒值，收集器将尽可能保证内存回收所花的时间不超过限定值。但GC停顿时间缩短是以牺牲吞吐量和新生代空间换来的。
- GCTimeRatio吞吐量大小：参数的值是大于0且小于100的整数

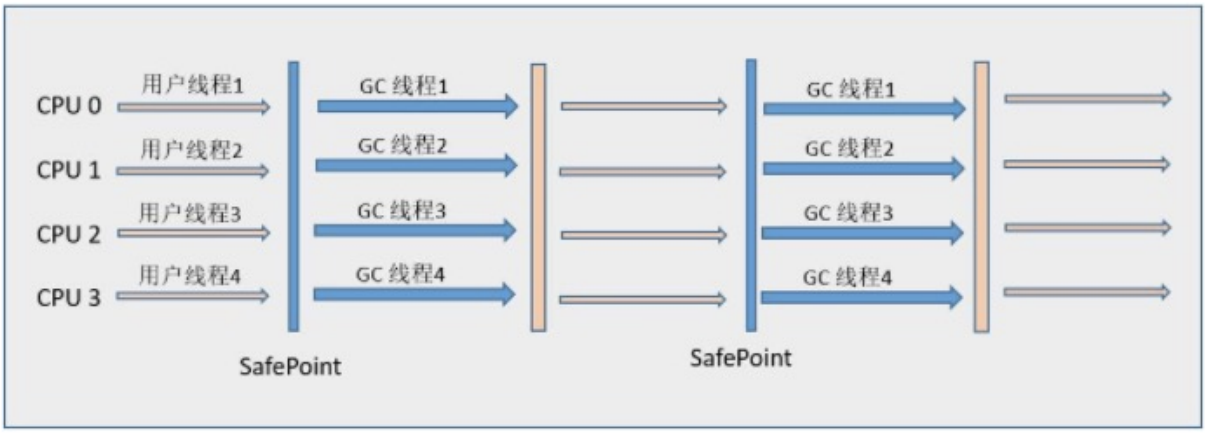
还有一个参数UseAdaptiveSizePolicy:这是一个开关参数，可以让虚拟机自己设置内部详细参数，只需要给优化目标（就是前两个参数）。

Serial Old收集器

Serial Old就是Serial收集器的老年代版本，运行示意图和Serial收集器类似。

Parallel Old收集器

是Paralle Scavenge收集器的老年代版本，使用多线层和“标记-整理”算法



CMS收集器

Current Mark Sweep收集器是一各以获取最短回收停顿时间为目标的收集器。适用于当前B/S系统的服务端，注重响应速度，给用户带来较好的体验。算法是“标记-清除”

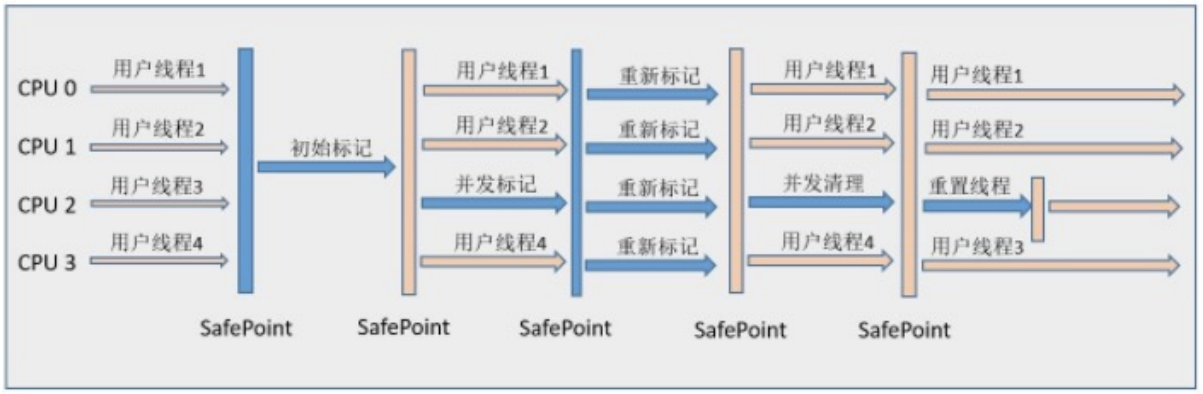
步骤： 初始标记 ----并发标记----重新标记----并发清除

初始标记和重新标记仍然需要“stop the world”。初始标记是标记GC Roots能直接关联到的对象（时间较短），并发标记则是GC Roots tracing的过程，重新标记是为了修正并发标记期间因用户程序继续运作而导致的标记产生变动的那一部分对象的标记记录，这一部分停顿时间较长。

优点：并发收集、低停顿。

缺点:

- CMS对CPU资源非常敏感。在并发阶段，虽然不会导致用户停顿，但会占用CPU资源导致应用程序变慢，吞吐量变低。当CPU低于2个时，对程序运行的影响就很大。
- 无法处理浮动垃圾：因为并发清理时，用户线程也在产生垃圾，且由于还没有被标记，只能下次GC时进行清理，我们称为“浮动垃圾”。同样，由于在垃圾收集阶段用于线程还在运行，所以需要预留内存给用户线程使用，而不能像其他收集器一样，等到老年代满了以后再进行手机。（通常是0.68）
- 是基于“标记-清除”算法，会产生大量的空间碎片，给分配大对象造成麻烦，导致提前触发GC。（存在参数是进行内存碎片整合操作）



G1收集器

特点：

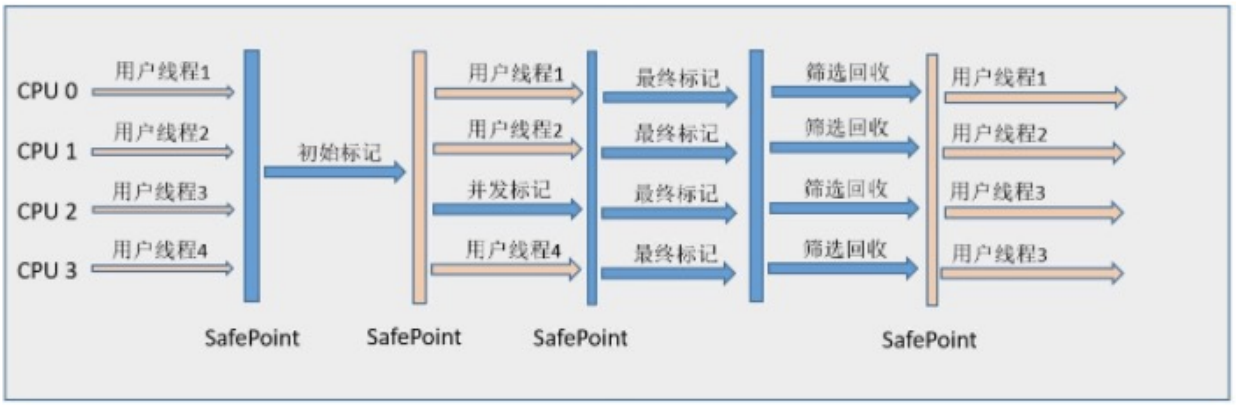
- 并行与并发：利用多CPU降低stop-the-world的时间。同时在执行GC动作时，其他java线程可以继续执行。
- 分代收集：分代概念得以保存，但G1可以独立管理，且对于新创建对象和老不死对象会采取不同的方式去处理获得更好的回收效果
- 空间整合：整体上是基于“标记整理”，局部基于“复制”，都不会产生内存空间碎片。
- 可预测的停顿：除开降低停顿时间,还可以建立预测的停顿时间模型，能让使用者指定在一个明确长度为M毫秒的时间内，垃圾收集所耗时间不超过N秒。

优势的原因 它将整个Java堆划分为多个大小相等的独立区域，虽然还保留有新生代和老年代的概念，但他们并不再是物理隔离的了，都是一部分Region的集合。

之后G1通过一个叫Remebered Set的东西来避免全堆扫描，能够以region为单位进行垃圾回收。每一个region都有一个这个set，当对这个reference类数据进行写操作时，就会判断引用对象是否处于不同的region中，如果是，则将信息引入这个set中，这样进行GC根节点枚举时，不进行全堆的扫描也不会遗漏。

步骤

- 初始标记：标记GC Roots能直接关联到的对象
- 并发标记：开始对堆中对象进行可达性分析，找出存活对象，耗时长，但可以与用于线程并行。
- 最终标记：修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分，并将其对象变化记录在Remebered Set中
- 筛选回收：对各个region的回收价值和成本进行排序，根据用户期望的GC停顿事假你来指定回收计划



内存分配与回收策略

1.对象有限在Eden去分配，当Edne区没有足够的空间进行分配时，虚拟机发起一次Minor GC 2.大对象直接进入老年代 3.长期存活对象将进入老年代。（虚拟机给每个对象都定义一个对象年龄计数器，在Eden出生并经历第一次Minor GC后仍然存活并进入Survivor的话，对象年龄为1，在Survivor区每熬过一次Minor GC ，年龄+1，当到15时，会进入老年代） 4.动态判断对象年龄：并不是用于要求对象年龄达到MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象

大小的总和大于survivor空间的一半，年龄大于或等于该年龄的对象直接进入老年代 5.空间分配担保：在发生MinorGC 前，虚拟机会先检查老年代最大可用空间的连续空间是否大于新生代空间的所有对象总和，如果条件成立，那么可以确保Minor GC是可靠的。如果允许，会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果达到，会尝试进行Minor GC，尽管是有风险的。如果小于或者说不允许冒风险，则进行full GC。因为老年代要替第二个Survivor区进行担保

虚拟机性能监控与故障处理工具

这章讲的主要是利用jdk/bin下面的各种java自带命令行工具进行运行日志、异常堆栈、GC日志等信息的查询，从而帮助我们更快地分析数据、定位解决问题。

jdk的命令行工具

- jps:虚拟机进程状况工具，主要功能是列出正在运行的虚拟机进程，并显示虚拟机执行主类名以及这些进程的本地虚拟机唯一IP（Local Virtual Machine Identifier），同时还能区分多虚拟机情况。
- jstat：虚拟机统计信息监视工具，用于监视虚拟机各种运行状态信息的命令行工具
- jinfo：java配置信息工具，可以实时查看和调整虚拟机各项参数
- jmap：java内存映像工具，用于生成堆转存储快照
- jhat：虚拟机堆转存储快照分析工具，用来配合jmap，分析jmap生成的堆转存储快照，比如分析dump文件
- jstack：java堆栈跟踪工具。用于生成虚拟机当前时刻的线程快照，就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，主要目的是定位线程出现长时间停顿的原因，比如死锁等。
- HSDIS:JIT生成代码汇编，作用是将动态生成的本地代码还原为汇编代码输出，同时生成大量有价值的注释，方便从汇编代码中分析问题。

jdk的可视化工具

jConsole：java监视与管理控制台

启动：通过jdk/bin/jconsole.exe可以直接启动，它是以PID作为连接标识的，可以连接本地的也可以连接远程的。

- 内存监控：相当于可视化的jstat，主要监视受收集器管理的虚拟机内存（永久代和java堆）的变化趋势
- 线程能监控：相当于可视化的jstack命令，遇到线程停顿时可以利用这个进行分析。可以查看各个线程的状态。

VisualVM:多合一故障处理工具

是需要网上下安装包然后操作的一个插件，除开与jconsole一些相同的功能外，介绍一下它独特的功能。

- 生成，浏览堆转储快照

- 分析程序性能：在CPU和内存上了解所占空间等情况
- BTrace动态日志跟踪：在不停止目标程序运行的前提下，动态加入原本不存在的调试代码

类文件结构

这里强调实现语言无关性的基础仍然是虚拟机和字节码存储格式。java虚拟机不与包括java在内的任何语言绑定，它只与“class文件”这种特定的二进制文件格式所关联，而且不关系Class的来源是何种语言。

class类文件结构

首先了解Class文件是一组以8位字节为基础单位的二进制流，各个数据项目按照顺序紧凑地排列在Class文件中，中间没有任何分隔符。

Class文件格式采用一种类似C语言结构体的伪结构来存储数据，这种伪结构只有两种数据类型：无符号数和表。

- 无符号数：属于基本的数据类型，用u1，u2等分别来代表1个zijie，2个字节的无符号数。无符号数可以用来描述数字、索引引用、数量制或者按照UTF-8编码构成字符串值。
- 表：由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯以_info结尾。整个class文件本质就是一张表。

魔数

每个Class文件的头4个字节称为魔数，它的唯一作用就是确定这个文件是否是一个能被虚拟机接受的Class文件。魔数是0xCAFEBABE.

紧接着魔数的4个字节存储的是Llass文件的版本号，5和6字节是次版本号，7和8字节是主版本号。（1.7是51）

常量池

在版本号后面是常量池入口，它是Class文件结构中与其他项目关联最多的数据类型，也是占用class文件空间最大的数据之一，也是第一个出现的表类型数据项目。

1.常量池入口是u2类型的数据u，代表常量池容量计数器(从1开始计数) 2.字面量：接近于java语言层面的常量概念，如文本字符串、声明为final的常量值等 3.符号引用：类和接口的全限名称，字段的名称和描述符，方法的名称和描述符。

通常有14种常量值，而每个常量都是一个表。在每个u2字符描述是哪种类型的常量值之后，后面紧接着的就是描述这个常量值具体结构和地址指向的参数。然后依次查表遍历。

访问标志

在常量池之后，这个标志用来识别一些类或者接口层次的访问信息：包括这个class是类还是接口，是否定义为public类型，是否定义为abstract类型，如果是类的话，是否被声明为final。

类索引、父类索引和接口索引集合

类索引确定这个类的全限定名，父索引确定这个类的父类的全限定名（因为java不允许多继承，所以父类索引只有一个），接口索引用来描述这个类实现了哪些接口。

类索引和，父类索引，接口索引按顺序排列在访问标志之后

类索引和父类索引引用u2类型的索引值，指向类描述符常量，通过其中的索引值，找到指定的名字字符串

接口索引是一组u2类型的数据的集合，第一个是接口计数器，表示索引容量，之后是接口索引所指的接口名。

字段表集合

用于描述接口或者类中声明的变量。字段包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。

字段表集合

类型	名称	数量	类型	名称	数量
u2	access_flags	1	u2	attributes_count	1
u2	name_index	1	attribute_info	attributes	attributes_count
u2	descriptor_index	1			

- access_flags：是字段修饰符，表示字段的作用域(public,private,protected)，是实例变量还是类变量(static),可变性(final)等等
- name_index：对常量池的引用，代表字段的简单名称
- descriptor_index：对常量池的引用，代表方法的描述符(double,char,byte等)。
- 之后的是属性表集合用于存储一些额外的信息，在之后介绍

方法表集合

Class文件存储格式对方法的描述和对字段的描述几乎采用了完全一致的方式，。

类型	名称	数量	类型	名称	数量
u2	access_flags	1	u2	attributes_count	1
u2	name_index	1	attribute_info	attributes	attributes_count
u2	descriptor_index	1			

- access_flags:相比于字段表的修饰，少了volatile和transient不能用类修饰方法，多了synchronized，native，abstract等关键字用来修饰方法
- name_index:对常量池引用，代表方法的名称
- descriptor_index：代表方法的返回值(方法的具体实现代码，存放在属性表集合的code属性里)

ps:在class文件中，对于重载的条件包括，其他相同，但返回值不同。

属性表集合

属性表里主要定义了21种属性

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

- code属性：java程序方法体中的代码经过javac编译器处理后，最终变为字节码指令存储在Code属性内。如果把一个java程序中的信息分为代码(code)和元数据(包括类、字段、方法及其他信息)。那么整个class文件中，code用于描述代码，所有其他数据用于描述元数据。具体包括属性名，属性长度，最大帧栈深度，代码长度，代码等信息。
- Exception属性：其作用是列举出方法中可能抛出的受查异常，也就是方法描述时在throws关键字后面列举的异常
- LineNumberTable属性：用于描述java源码行号和字节码行号之间的对应关系，不是运行必须的。
- LocalVariableTable属性：用于描述帧栈中局部变量表中的变量与java源码中定义的变量之间的关系。（比如java中定义的abc，在class文件中使用哪个arg0代表，方便调试）
- SourceFile属性：用于记录生成这个class文件的源码文件名称。
- ConstatnFile属性：通知虚拟机自动为静态标量赋值，只有被static关键字修饰的变量才可以使用这个属性。(没有被static修饰的，赋值是在实例构造器的init方法里赋值，而类变量是通过生成ConstantValue属性来进行赋值)
- InnerClasses属性：用于记录内部类和宿主类之间的关系。如果一个类中定义了内部类，那编译器会为它以及它所包含的内部类生成InnerClassed属性。描述内部类常量索引，访问标志等。
- Deprecated及Synthetic属性：这两个是标志类型的布尔属性，前一个用来表示这各不再被推荐使用，后一个表示这个字段或者方法不是java源码直接产生。
- StackMapTable属性：这个属性会在虚拟机类加载的字节码验证阶段被新类型检查器验证，目的在于代替以前比较消耗性能的基于数据流分析的类型推导验证器。就是验证字节码的正确性，比以前通过数据流去验证更加简单
- Signature属性：可选的定长属性，jdk1.5之后，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量或者参数化类型，则signature属性会为它记录泛型签名信息。
- BootstrapMethods属性：用于保存invokedynamic指令引用的引导方法限定符。

字节码指令简介

字节码 = 操作码 + 操作数；

java虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字(即操作码Opcode)以及跟随其后的零至多个代表此操作所需参数(即操作数Operands)而组成。而由于java虚拟机采用的是面向操作数栈而不是寄存器的架构，所以大多数指令都不含操作数，只有一个操作码。

优势是数据量小，高传输速率。缺点是总操作码数不能超过256位，若超过了会在执行字节码时损失一些性能。

字节码与数据类型

从表中我们发现大部分的指令都没有支持整数类型byte，char和short，甚至没有任何指令支持boolean的。编译器会在编译期间将byte和short类型的数据带符号扩展为int型，将boolean和char类型数据零位扩展为相应的int类数据，从而使用int类型对应的字节码指令来处理。

OPCODE	BYTE	SHORT	INT	LONG	FLOAT	DOUBLE	CHAR	REFERENCE
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

- 加载和存储指令：用于将数据在帧栈中的局部变量表和操作数栈之间来回传输
  - 将一个局部变量加载到操作栈：Tload;
  - 将数值从操作数栈存储到局部变量表：Tstore；
  - 将常量加载到操作数栈：Tipush，Tconst；
  - 扩充局部变量表的访问索引：wide；
- 运算指令：用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。大体分为对整形数据进行运算和对浮点型数据进行运算。
- 类型转换指令：将两种不同的数值类型进行相互转换，一般用于实现用户代码中的显示类型转换操作。
  - java虚拟机直接支持宽化类型转换(即小范围向大范围转化)：int->long->float->double。
  - 窄化类型转换必须显式地使用转换指令来完成
- 对象创建与访问指令：对类实例和数组的创建与操作使用的是不同的字节码指令。具体包括创建，访问，加载，取数组长度等指令。
- 操作数栈管理指令：和普通的栈操作一样，虚拟机提供了用于直接操作操作数栈的指令。
- 控制转移指令：用于让java虚拟机有条件或无条件的从指令的位置指令而不是控制转移指令的下一条指令继续执行程序，即修改pc寄存器的值(条件分支，复合条件分支和无条件分支)

- 方法调用和返回指令：主要是调用一些方法的指令，比如调用对象的实例方法，接口方法等。
- 异常处理指令：在java程序中显示抛出异常的操作(throw语句)都由athrow指令实现，catch语句不是用字节码语句。
- 同步指令：java虚拟机可以支持方法及的同步和方法内部一段指令序列的同步，都是使用monitor来支持

虚拟机类加载机制

类加载的时机

类加载的整个生命周期：加载->验证->准备->解析->初始化->使用->卸载。

其中加载，验证，准备，初始化和卸载的阶段顺序是确定的。但解析不一定，它可以在初始化阶段之后再开始，这是为了支持java语言的运行时绑定。

初始化的5个时机(主动引用)：

- 遇到new，getstatic,putstatic和invokestatic这四条字节码指令时，如果类没有进行初始化，则需要先触发其初始化。包括使用new关键字实例化对象，读取或者是一个类的静态字段，调用一个类的静态方法的时候
- 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有初始化，要先触发其初始化
- 当初始化一个类的时候，如果其父类还没有经过初始化，则需要先触发其父类的初始化
- 当虚拟机启动，用户需要指定一个要执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类。
- jdk1.7动态语言支持时，java.lang.invoke.MethodHandle(通过反射调用方法的类),当该实例解析结果有相关方法句柄，且对应的类没有经过初始化时，就会对其进行初始化。

被动引用

- 通过子类引用父类的静态变量，不会导致子类初始化
- 通过数组定义来引用类，不会触发此类的初始化
- 常量在编译阶段会存入调用类的常量池，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

类加载的过程

加载

加载阶段的三部曲：

- 通过一个类的全限定名来获取定义此类的二进制字节流
- 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
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

非数组类的加载过程是开发人员可控性最前搞得，因果加载阶段的类加载器可以使用系统默认，或者用户自定义的类加载器。但对于数组类，它是由java虚拟机直接创建的，但数组类的元素类型还是需要类加载器去完成。所以一个数组类创建需要遵循的规则如下：

- 如果数组的组件类型(维度-1后的类型)是引用类型，那就递归采用本节中定义的加载过程去加载，shuzu将在加载该组件类型的类加载器的类名称空间上被标识。
- 如果数组的组件类型不是引用类型（int[]），java虚拟机会把数组标记为与引导类加载器关联。
- 数组类的可见性与它的组件类型可见性一致，如果组件类型不是引用类型，数组可见性默认public。

验证

这个阶段是确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并不会危害虚拟机自身的安全。

- 文件格式验证：验证字节流是否符合Class文件格式的规范，并且能够被当前虚拟机处理。只有这阶段通过后，字节流才会进入内存的方法区存储，后面的验证也都是基于方法区的存储结构进行的。
  - 是否以魔数开头
  - 主次版本号是否在当前虚拟机处理范围内
  - 常量池常量中是否有不被支持的常量类型
  - .....
- 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范的要求。
  - 这个类是否有父类(除开java.lang.Object都有父类
  - 这个类是否继承了不允许被继承的类(final修饰的类)
  - 这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法
  - .....
- 字节码验证:主要通过数据流和控制流分析，确定程序语义是合法的，主要对类的方法体进行校验分析
  - 保证跳转指令不会跳转到方法体以外的字节码指令上
  - 保证操作数栈的数据类型与指令代码序列能配合工作(比如用fadd操作int型数据就是错误的)
  - .....
- 符号引用验证:对类自身以外的(常量池中的各种符号引用)的信息进行匹配性校验
  - 符号引用中通过字符啊穿描述的全限定名能否找到指定的类
  - 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段
  - 是否可访问(根据访问修饰符)

准备

准备阶段是正式为类变量分配内存并设置初始值的阶段，这些变量所使用的的内存都将在方法区中进行分配。

- 内存分配的仅包括类变量(被static修饰的变量)不包括实例变量，是咧变量会在对象实例化时随着对象一起分配在java堆中。

- 这里的初始值：是指数据类型的零值。

解析

是虚拟机将常量池内的符号引用替换为直接引用的过程。其次，由于存在动态解析的情况(这里的动态指的是等到程序运行到这条指令时，解析动作才进行)，索引解析的发生阶段不确定。解析的主要对象：类或接口、字段、类方法、接口方法、方法类型、方法句柄与调用点限定符

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可，且与虚拟机实现的内存布局没有关系，引用的目标不一定已经加载到内存中。

直接引用：是直接指向目标的指针、相对偏移量或者一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，但如果有了直接引用，那引用的目标必然在虚拟机中存在。

- 类或接口的解析：当前代码所处类为D，要将一个从未解析过的符号N解析为一个类或者接口C
  - C不是数组类型，虚拟机将N所代表全限定名传递给D的类加载去去加载这个类C，在加载过程中，需要元数据验证，字节码验证的需要，可能需要出发其他类的加载动作(比如父类的加载或接口实现)，一旦加载过程出现异常，则解析失败。
  - C是数组类型，并且数组的元素类型为对象。N的描述符会是类似[Ljava/lang/Iteger形式，会先按前一个负责加载数组元素类型，比如加载java.lang.Integer元素类型，接下里虚拟机生成一个代表此数组维度和元素的对象。
  - 如果以上都没有任何异常，C在虚拟机就是一个有效的类或者接口，之后对C进行符号引用验证，比如是否具备访问权限等。之后就好了。
- 字段的解析：先对字段表内class_index项中索引的符号，也就是字段所属的类或者接口的符号引用解析。之后用C表示这个字段所属的类或接口。
  - 如果C本身包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束
  - 否则，若在C中实现了接口，则按继承关系从上往下递归搜索各个接口和它的父接口，如果包含了与其匹配的字段，则返回这个字段的直接引用，查找结束
  - 否则，C不是java.lang.Object类的话，则按继承关系依次递归搜索它的父类，如果包含了简单名称和字段描述符都相匹配的字段，则返回这个字段的直接引用，查找结束。
  - 否则，查找失败，抛出java.lang.NoSuchFieldError异常
  - 最后，进行权限验证。
- 类方法的解析：同样先解析出类方法表中索引的方法所属的类或接口的符号引用，解析成功进行之后步骤，用C表示这个类
  - 类方法和接口方法符号引用的常量类型定义是分开，如果类方法表中发现C是个接口，直接抛出异常
  - 通过第一步，然后在类C中查找是否有简单名称和描述符都与目标相匹配的方法，有则返回方法的直接引用，结束
  - 没有，在C的父类中递归查找，有则返回结束
  - 没有, 在C的父接口和实现的接口列表中递归查找，有说明C是抽象类，查找技术，抛出异常
  - 没有，宣告方法查找失败，抛出异常
  - 若查找成功，对方法进行权限验证。

- 接口方法的解析：先对符号引用进行解析，用C表示这个接口
  - 如果发现C是个类不是接口，直接抛出异常
  - 否则，在C中查找是否有简单名称和描述符都与目标相匹配的方法，有则返回，查找结束
  - 否则，在接口C的父接口中递归查找，直到java.lang.Object类，看是否有匹配，有则返回
  - 否则，宣告方法查找失败，抛出异常
  - 因为接口方法默认public，不需要进行权限验证。

初始化

初始化阶段就是执行类构造器<clinit>()方法的过程。

- <clinit>()方法是由编译器自动收集类中的所有类变量的复制动作和静态语句中的语句合并产生的，收集的顺序由语句在源文件中出现的顺序所决定，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但不能访问
- <clinit>()方法与类的构造函数不同，它不需要显示地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行前，父类的该方法已经执行完成，所以虚拟机第一个被执行<clinit>()方法的类一定是java.lang.Object。
- 由于父类的<clinit>()方法先执行，所以父类总定义的静态语句块要优先于子类的赋值变量。
- <clinit>()方法对于接口或类来说不是必需的，如果一个类没有静态语句块，也没有对变量的赋值操作，那么编译器就可以不生成<clinit>()方法
- 接口中不能使用今天语句亏啊，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口不同的是，执行接口<clinit>()方法不需要先执行父类的其方法，除服父接口中定义的变量使用时，父接口才会初始化。实现类同理。
- 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁，同步，若果多个线程同时去初始化一个类，只有一个线程去执行该类的<clinit>()方法方法，其他线程都会阻塞等待。

类加载器

类加载阶段的“通过一个类的全限定名来获取描述此类的二进制字节流”，这个动作放到java虚拟机之外去实现，而实现这个动作的代码模块就是“类加载器”。

对于任意一个类，都需要由它的加载器和这个类本身一同个群里其在java虚拟机中的唯一性。也就是说比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则必不相等。

双亲委派模型

从java虚拟机的角度来说，只存在两种不同的类加载器，一个是启动类加载器(Bootstrap ClassLoader)，这个类用C++实现，是虚拟机自身的一部分；另一个就是所有其他的类加载器，这些都用java语言实现，独立于虚拟机外部，并且全部继承自抽象类java.lang.ClassLoader.

加载器的分类

- 启动类加载器(Bootstrap ClassLoader)：这个类负责将存放在<JAVA_HOME>\lib目录中，或者被-Xbootclasspath参数所指定的路径中，并且是被虚拟机识别的类库加载到虚拟机内存中。

启动类加载器无法被java程序直接引用，用户在编写自定义加载器时，如果需要把加载请求委派给引导类加载器，直接使用null代替即可。

- 扩展类加载器(Extension ClassLoader)：这个负责加载<JAVA_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。
- 应用程序类加载器(Application ClassLoader)：这个类加载器时ClassLoader中的getSystemClassLoader()方法的返回值，议程系统类加载器，它负责用户类路径(ClassPath)上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序没有顶一个，就回使用程序默认的加载器。

所有的加载器的父子关系不是以继承的关系实现，二是都使用组合来复用父加载器的代码。

双亲委派的步骤 一个类加载器收到了类加载的请求，自己不会先尝试加载，而是将请求委派给父类加载器完成，每一个层次的加载器都是如此，只有当父类加载器反馈无法加载时，子加载器才会尝试自己去加载。

双亲委派模型的好处 java类随着它的加载器一起具备了一种带有优先级的层次关系。 代码实现在loadClas()中，就是先检查是否被加载过，没有则调用父类的loadClass()，若父加载器默认为空则使用启动类加载器，父类加载失败抛出异常，子类自己加载。

破坏双亲委派模型

虚拟机字节码执行引擎

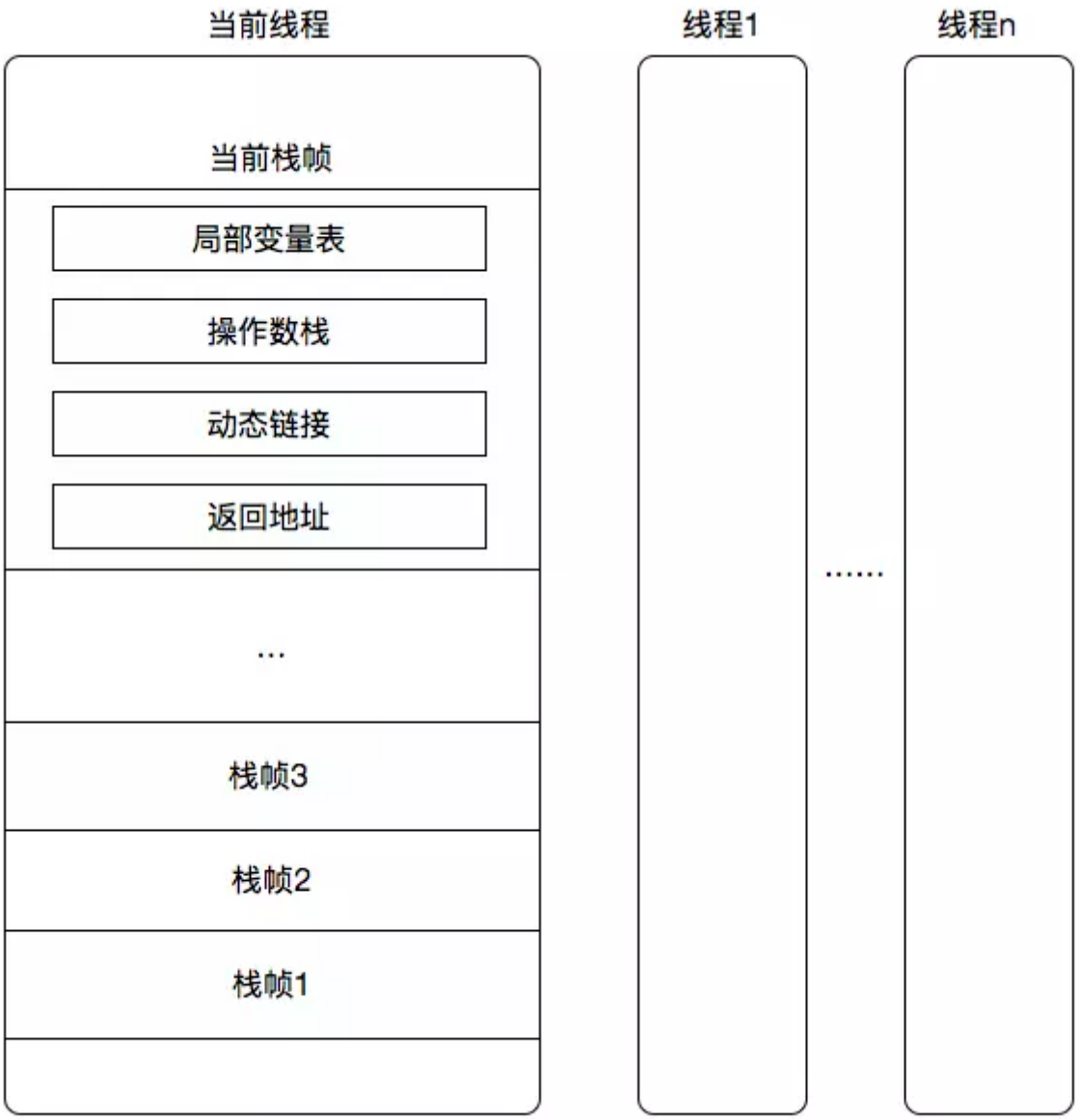
总的来说，java虚拟机的执行引擎的效果是，输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。

运行时帧栈结构

帧栈适用于支持虚拟机进行方法调用和方法执行的数据结构，它是雪泥机运行时数据区中的虚拟机栈的栈元素。帧栈存储了额方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始至执行完成的过程，都对应一个帧栈在虚拟机栈里面从入栈到出栈的过程。

每一个帧栈包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。且帧栈的深度是在编译期间就确定，并写入方发表的code属性中。

对于执行引擎来说，只有位于栈顶的帧栈才是有效的，称为当前帧栈，与这个帧栈相关联的方法称为当前方法。



局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。

在编译时，就在方法的Code属性的max_locals数据项中确认了该方法所需要分配的局部变量表的最大容量。

局部变量表的容量是以变量槽(Slot)为最小单位，虚拟机并没有明确指明一个Slot占用的内存大小，只是说明每个Slot都能存储一个boolean,byte,char,short,int,float,reference,returnAddress类型的数据，这个8种数据类型，都可以使用32位或者更小的物理内存来存放，但不是slot=32位长度。它是随着处理机和操作系统或者虚拟机的不同而调整。

对于64位的数据类型，虚拟机会以高位对齐的方式为其分配两个连续的Slot空间，所以对于long和double的数据类型读写会分成两次32位的读写，但因为局部变量表示建立在线程的堆栈上，是线程私有的，所有就算不是原子操作，都不会引起安全问题。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围从0开始到局部变量表的最大Slot数量。如果访问的是32位数据类型的变量，索引n就代表了使用第n个Slot，如果是64位数据类型变量，则同时使用n和n+1两个Slot进行访问。且对于相邻的两个共同存放同一个64位数据的slot，不允许被单独访问。

方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的。比如执行的是实例方法，那么局部变量表的第0个索引slot默认用来传递方法所属对象实例的引用，其余参数则按照参数表顺序排列，依次占用从1开始的Slot，参数表分配完成后，再根据方法体内部定义的变量顺序和作用域来分配其余slot。

PS:局部变量没有“准备”阶段，需要程序员手动的位局部变量赋予初始值。

```
public static void main(String[] args){
    {
        byte[] abc = new byte[64*1024*1024]
    }
    int a = 0;
    System.gc();
}
```

这段代码中，只有经过手动设置为null这个操作，把局部变量表清空，才可以对其进行垃圾回收，或者说方便后面耗时长的代码运行。

操作数栈

也称操作栈，是后入先出栈。其最大深度也是在编译时就写入到了code属性中。操作数栈的每一个元素可以是任意的java数据类型，包括double和long。32位数据类型占栈容量1，64位占2.。

在一个方法刚开始执行的时候，操作数栈是空的，随着方法的执行，会有各种字节码往操作数栈中写入和提取内容，也就是出栈/入栈操作。

操作数中元素的数据类型必须与字节码指令的序列严格匹配。

动态连接

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

在 Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另外一部分将在每一次的运行期期间转化为直接引用，这部分称为动态连接。

方法返回地址

在一个方法被执行后，有两种方式退出这个方法：正常完成出口和异常完成出口

- 正常完成出口：当执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者（调用当前方法的方法称为调用者），是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定

- 异常完成出口：在方法执行的过程中如果遇到了异常，并且这个异常没有再方法体内得到处理，无论是 Java 虚拟机内部产生的异常，还是在代码中使用 athrow 字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出

无论何种方式退出，都需要返回到方法被调用的位置，程序才能继续进行。所以方法返回时需要在帧栈中保存一些信息，用来帮助恢复它的上层方法的执行状态。

- 正常退出时：调用者的PC计数器的值作为返回地址，帧栈中保存这个计数器值
- 异常退出时：返回地址通过异常处理器表来确定，帧栈中通常不保存。

方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令等

附加信息

允许具体的虚拟机实现增加一些规范之外的信息到帧栈之中。

方法调用

方法调用的唯一任务就是确定被调用方法的版本，不涉及方法内部的具体运行过程。

但因为Class文件的编译过程不包含传统编译的连接步骤，也就是生活一切方法调用在Class文件中存储的知识符号引用，而不是方法在实际运行时内存布局的入口地址。这个给java带来了更强大的动态扩展能力，但也使得调用方法更加复杂，需要在类加载期间，或运行阶段才能确定目标方法的直接引用。

解析调用

在类加载的解析阶段，将一部分符号引用转换成直接引用，前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期间是不可改变的。这类方法的调用，我们称为解析。

在java字节码中，调用方法的字节码一共有

- invokestatic：调用静态方法
- invokespecial：调用实例构造器方法<init>、私有方法、父类方法
- invokevirtual：调用所有的虚方法
- invokeinterface：调用接口方法，会在运行时确认一个实现此接口的对象
- invokedynamic：先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。

但只有第一二种指令调用方法符合解析调用，即静态方法、私有方法、实例构造器和父类方法，它们在类加载的时候会把符号引用直接解析为该方法的直接应用，这些称为非虚方法。同时还有final方法，因为final修饰的方法无法被覆盖，也没有其他版本，所以也是非虚方法。

分派调用

解析调用是一个静态的过程，在加载阶段就可以确认目标方法的直接引用。分派调用有可能是静态的，也有可能是动态的，根据分派的宗量数又可以分为单分派和多分派，这两类两两组合，所以分派共可以细分为：静态单分派、静态多分派、动态单分派、动态多分派。

静态分派 将静态方法与重载进行结合思考

```
public class StaticDispatch {

    static abstract class Human {

    }

    static class Man extends Human {

    }

    static class Woman extends Human {

    }

    public void sayHello(Human guy) {
        System.out.println("hello, guy");
    }

    public void sayHello(Main guy) {
        System.out.println("hello, man");
    }

    public void sayHello(Woman guy) {
        System.out.println("hello, woman");
    }

    public static void main(String[] args){
        Human man = new Man();
        Human woman = new Woman();
        StaticDispatch dispatch = new StaticDispatch();
        dispatch.sayHello(man);
        dispatch.sayHello(woman);
    }
}

------
hello,guy
hello,guy
```

现在我们对新建操作进行分析

```
Human man = new Man();
```

在这段代码中，我们将Human称为变量的静态类型，后面的Man()称为实例类型。变量的静态类型本身不会发生变化，只有在使用时才会发生变化，并且最终的静态类型在编译期间可知。而实际类型变化的结果在运行期才可以确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

```
// 实际类型变化
Human man = new Man();
man = new Woman();

// 静态类型变化
dispatch.sayHello((Man) man);
dispatch.sayHello((Woman) man);
```

对于虚拟机(编译器)在实现重载时是通过参数的静态类型而不是实际类型做成判断的，并且静态类型是编译器可知的。因此在编译阶段，编译器会根据变量的静态类型决定使用哪个重载方法。

所有依赖静态类型定位目标方法的分派动作称为静态分派，静态分派典型的应用就是方法的重载。静态分派发生在编译阶段，所以方法的静态分派动作是由编译器执行的。

另外，虽然编译器能确定出方法的重载版本，但很多情况下重载版本不是唯一的，往往是选择一个更适合的版本，也就是说重载静态类型的选择是有优先级的，比如char->int->long->float->double，或者自我装箱，甚至还可以转型成实现的接口或父类。

动态分派 动态分配，我们把它理解为它是重写的体现。

```
public class DynamicDispatch {

    static abstract class Human {
        abstract void sayHello();
    }

    static class Man extends Human {
        void sayHello() {
            System.out.println("hello, man");
        }
    }

    static class Woman extends Human {
        void sayHello() {
            System.out.println("hello, woman");
        }
    }

    public static void main(String[] args){
        Human man = new Man();
        Human woman = new Woman();
        man.sayHello();
        woman.sayHello();
        man = new Woman();
        man.sayHello();
    }
}
------
hello, man
hello, woman
hello, man
```

导致输出不同的明显原因就是，两个变量的实际类型不一样，原因可以从字节码得出。

```
Human.sayHello();
```

他们调用的都是Human.sayHello()的符号引用，但最终两指令执行的目标方法不相同，原因要从invokevirtual(调用所有的虚方法)指令的多态查找过程开始。

- 找到操作数栈顶的第一个元素所指向的对象的实际类型，记做 C
- 在类型 C 中查找与常量中的描述符和简单名称相同的方法，如果找到则进行访问权限的判断，如果通过则返回这个方法的直接引用，查找结束；如果权限不通过，则返回 java.lang.IllegalAccessError 的异常
- 如果在 C 中没有找到描述符和简单名称都符合的方法，则按照继承关系从下往上依次在 C 的父类中进行查找和验证过程
- 如果最终还是没有找到该方法，则抛出 java.lang.AbstractMethodError 的异常

由于该指令在第一步就是在运行期确定接受者的实际类型，所以两次调用invokevitural指令，把常量池中的类方法符号引用解析到了不同的直接引用上，这就是重写的本质。

单分派和多分派 方法的接受者与方法的参数统称为方法的宗量，根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。单分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择。

```
public class Main {

    static class Tencent {
    }

    static class Alibaba {
    }

    static class Father {

        public void findJob(Tencent tencent) {

            System.out.println("father work in tencent");
        }

        public void findJob(Alibaba ali) {

            System.out.println("father work in alibaba");
        }
    }

    static class Son extends Father {

        @Override
        public void findJob(Tencent tencent) {
            System.out.println("son work in tencent");
        }

        @Override
```

```
        public void findJob(Alibaba ali) {
            System.out.println("son work in alibaba");
        }

    }


    public static void main(String[] args) {
        Father father = new Father();
        Father son = new Son();

        father.findJob(new Tencent());
        son.findJob(new Alibaba());

    }

}
-------
father work in tencent
son work in alibaba
```

在上述代码中，编译时确定了两个宗量，一个是接受者（静态类型）father还是son的判断，第二个时方法参数是alibaba还是tencent，这次的选择最终产物是两条invokevitural指令，两条指令的参数分别为常量池中指向Father.findJob(tencene)和Father.finJob(Alibaba)方法的符号引用，所以java语言的静态分派属于多分派类型。

而在运行阶段虚拟机的选择，也就是动态分派的过程，以第一个方法为例，它已经确定思findJob(tencent)，重点是判断接受者的实例类型是father还是son，多以java语言的动态分派是单分派类型。

到目前为止，java语言还是一个“静态多分派，动态单分派”的语言，也就是说在执行静态分派时是根据多个宗量判断调用哪个方法的，因为在静态分派时要根据不同的静态类型和不同的方法描述符选择目标方法，在动态分派的时候，是根据单宗量选择目标方法的，因为在运行期，方法的描述符已经确定好，invokevirtual 字节码指令根据变量的实际类型选择目标方法。

虚拟机动态分派的实现 而由于动态分派是非常频繁的动作，且动态分派的方法版本选择过程需要在运行时在类的方法元数据中搜索合适的目标方法，因此在实际实现时，都不会基于此进行频繁搜索。而是为类在方法区中建立一个需方发表，使用需方法表索引来代替元数据以提高性能。

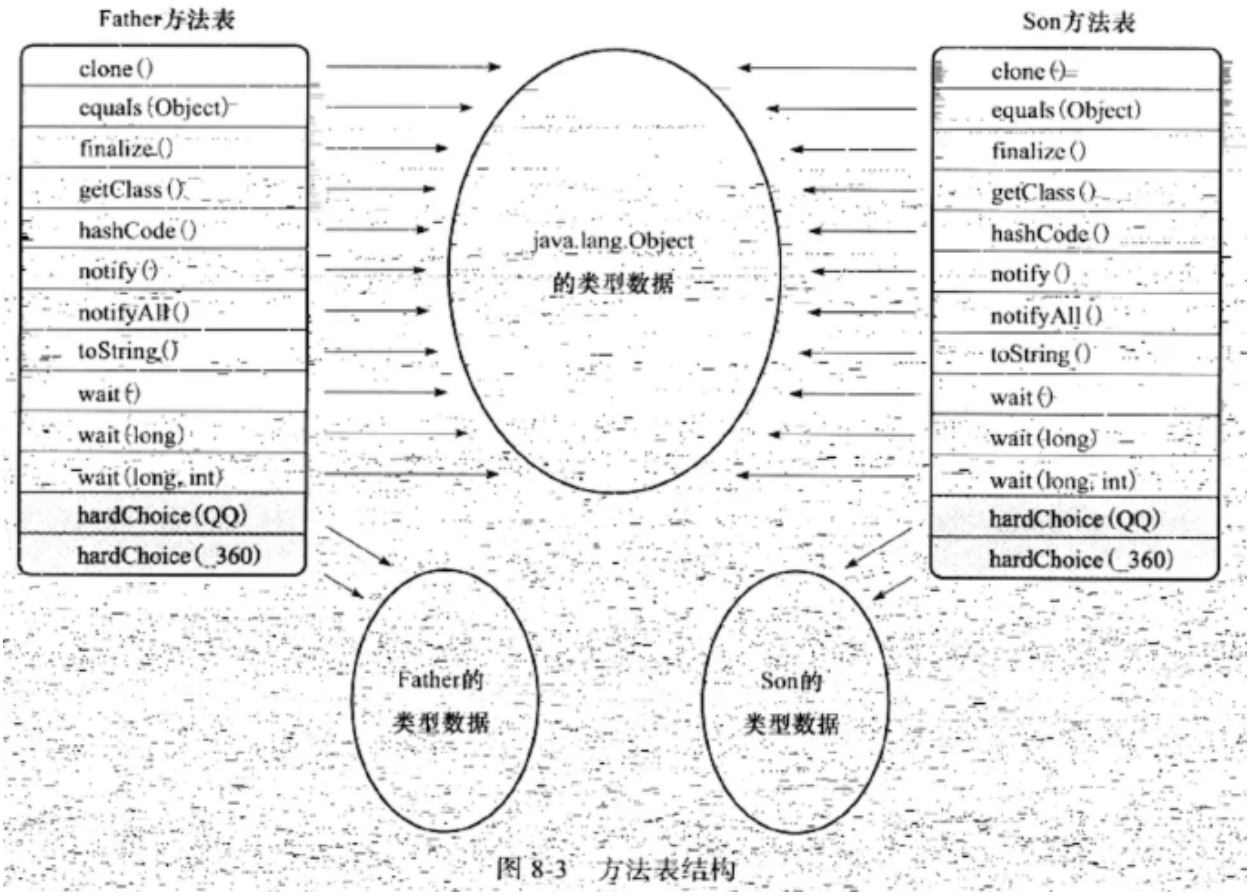


图 8-3　方法表结构

上图就是一个虚方法表，Father、Son、Object 三个类在方法区中都有一个自己的虚方法表，如果子类中实现了父类的方法，那么在子类的虚方法表中该方法就指向子类实现的该方法的入口地址，如果子类中没有重写父类中的方法，那么在子类的虚方法表中，该方法的索引就指向父类的虚方法表中的方法的入口地址。有两点需要注意：

- 为了程序实现上的方便，一个具有相同签名的方法，在子类的方法表和父类的方法表中应该具有相同的索引，这样在类型变化的时候，只需要改变查找方法的虚方法表即可。
- 虚方法表是在类加载的连接阶段实现的，类的变量初始化完成之后，虚拟机就会初始化该类的虚方法表

动态类型语言

- 动态类型语言：它的类型检查的主体过程是在运行期间而不是编译期间 ，为开发人员提供更大的灵活性
- 静态类型语言：在编译器确定类型，好处就是编译器可以提供严谨的类型检查。

java虚拟机要逐渐发展成为对动态类型语言支持的虚拟机。 这里重点是MethodHandle方法和invokedynamic指令

基于栈的字节码解释执行引擎

解释执行

- 编译执行：由解释器根据输入的数据当场执行而不生成任何目标程序

- 解释执行：是高级语言翻译程序的一种，它将源语言书写的源程序作为输入，解释一句后就提交计算机执行依据，并不形成目标程序。

基于栈的指令集与基于寄存器的指令集

以“1+1”为例

```
iconst_1
iconst_1
iadd
istore_0
```

基于栈的指令集:两条iconst_1指令连续把两个常量1压入栈后，iadd指令把栈顶的两个值出栈相加，然后把结果放回栈顶，最后istore_0把栈顶的值放到局部变量表的第0个slot中。

```
mov eax, 1
add eax, 1
```

基于寄存器，mov指令把EAX寄存器的值设为1，然后add指令再把这个值加1，结果就保存在EAX寄存器里。

基于栈的指令集的优点是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。同时还有，比如代码更加紧凑(字节码中每个字节就对应一条指令，而多地址指令集中还要存放参数)，编译器实现更加简单（不需要考虑分配的问题，所需空间都在栈上完成）

缺点是执行速度相对来说更慢，而且指令数量更多(多了很多出栈和入栈操作)，以及频繁的栈访问也就是呢次云访问，相对于处理器来说，内存始终是执行速度额瓶颈。

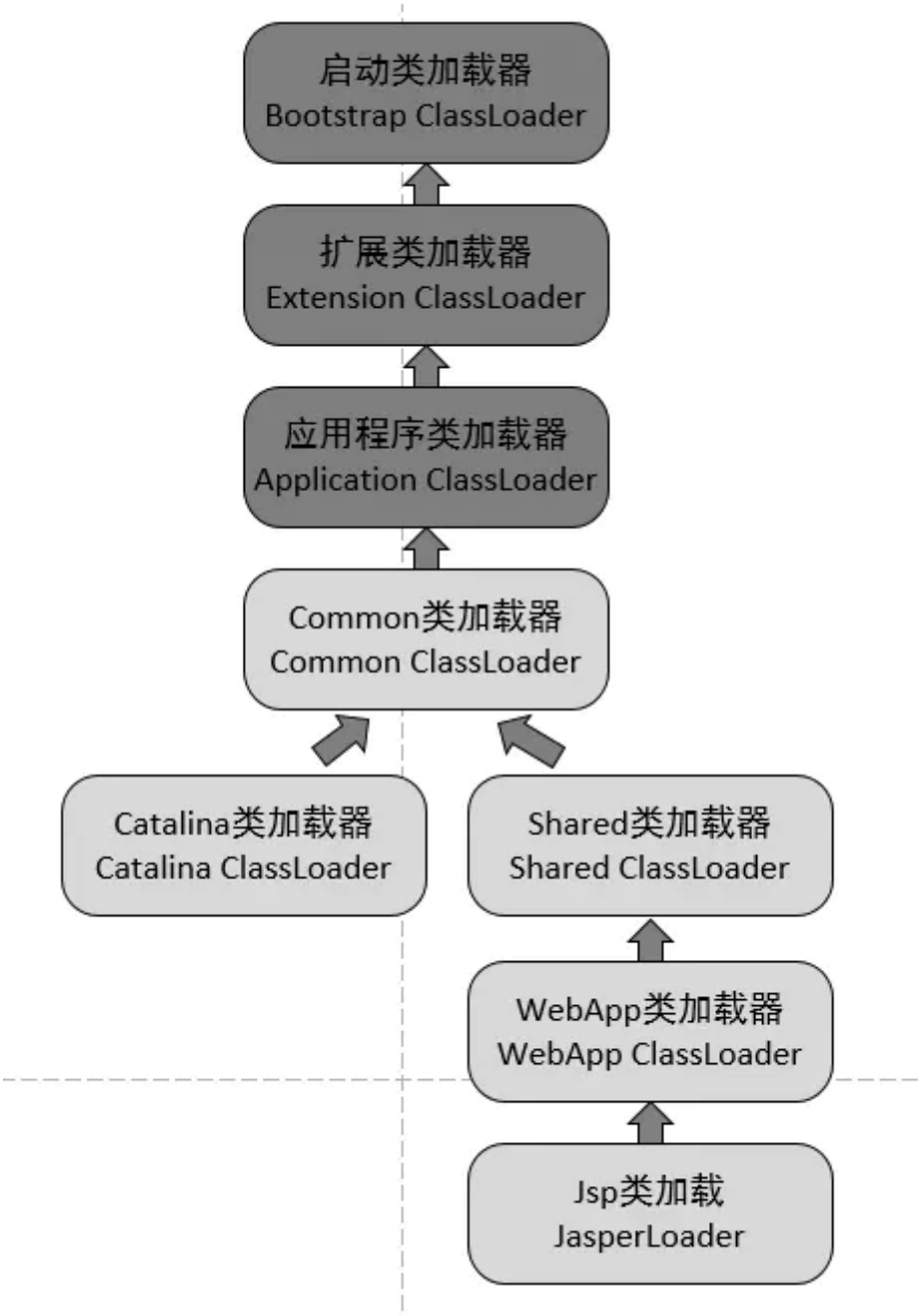
类加载即执行子系统的案例(以tomcat为例)

需要解决的问题

- 部署在同一个服务器上的两个web应用程序所使用的java类库可以实现相互隔离。
- 部署在同一个服务器上的两个web应用程序所使用的java类库可以互相共享。
- 服务器需要尽可能保证自身的安全不受部署的web应用程序影响
- 支持JSP应用的web服务器，需要支持HotSwap功能，也就是热部署。

tomcat加载器加载步骤
tomcat的加载器前三个和默认提供的加载器相同，我们看到，前面3个类加载和默认的一致，CommonClassLoader、CatalinaClassLoader、SharedClassLoader和WebappClassLoader则是Tomcat自己定义的类加载器，它们分别加载/common/、/server/、/shared/（在tomcat 6之后已经合并到根目录下的lib目录下）和/WebApp/WEB-INF/*中的Java类库。其中WebApp类加载器和Jsp类加载器通常会存在多个实例，每一个Web应用程序对应一个WebApp类加载器，每一个JSP文件对应一个Jsp类加载器。

- commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
- catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
- sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
- WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见；



java内存模型与线程

概述

1.衡量一个服务性能的高地好坏，每秒事务处理数(transactions per second TPS)是最重要的指标之一，它代表着一秒内服务端平均能响应的请求总数。

为了解决处理器和内存的速度矛盾，现代计算机都加入一层读写速度尽可能接近处理器运算速度的告诉缓存(cache)作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理就无需等待缓慢的内存读写。

同时为了解决缓存一致性(多个处理器的运算任务都涉及同一块内存区域时，可能导致各自的缓存数据不一样)，需要各个处理器访问缓存时都遵循一些协议，在读写时根据协议来进行操作。

除开增加告诉缓存，为了使处理器内部的运算单元能够被充分利用，处理器可能会对输入代码进行乱序优化，处理器会在计算之后将乱序执行的结果充足，保证该结果与顺序执行的结果是一直的，但不保证程序中各个语句计算的先后顺序与输入代码的顺序一致。

java内存模型

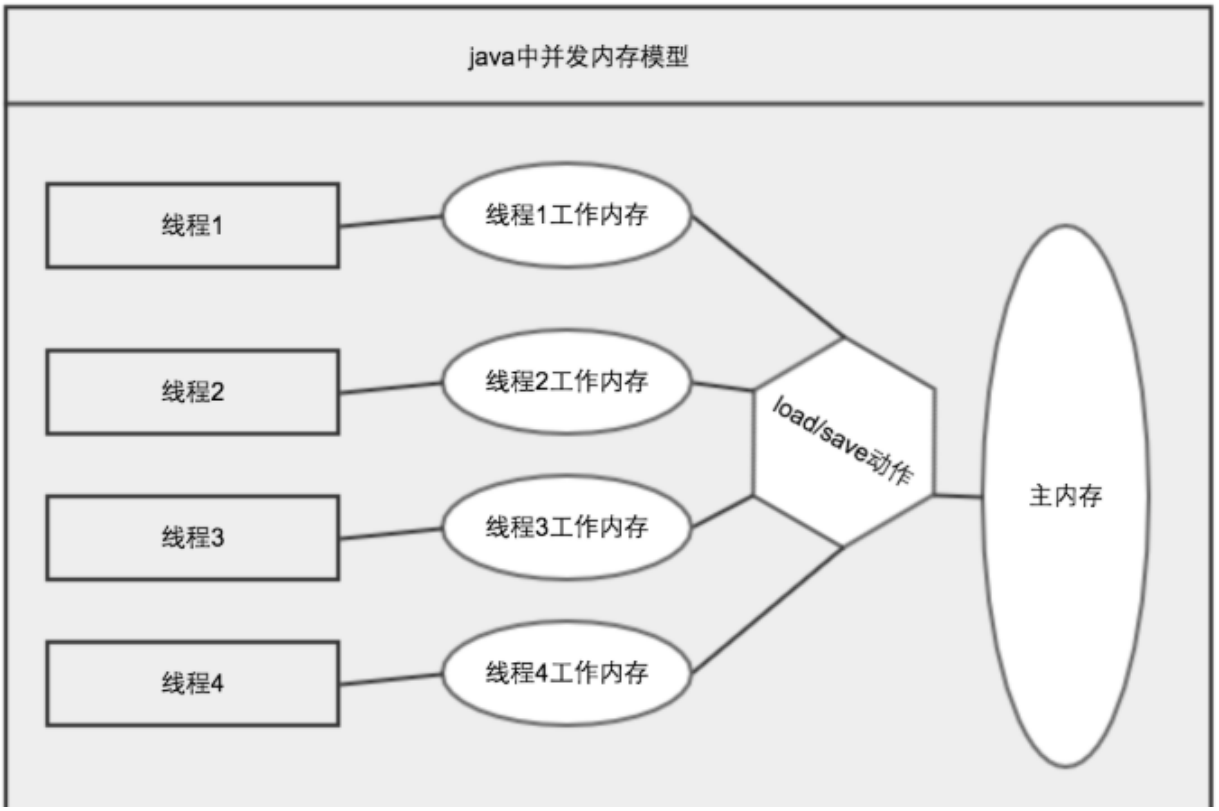
java虚拟机规范中试图定义一种java内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现java程序在各种平台下都能达到一致的内存访问效果。

主内存与工作内存

java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素，但不包括局部变量和方法参数，因为后者是线程私有的，不存在竞争问题。

java内存模型规定了所有的变量都存储在主内存中(这个主内存是虚拟机内存的一部分，并不是实际的硬件主内存)

每条线程有自己的工作内存，线程的工作内存保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。



java内存模型(jmm)和jvm的内存区域是两个不同概念，要分清。

内存间交互操作

关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、又如何从工作内存同步回主内存的细节，java内存模型定义了8种操作来完成，虚拟机中一下操作通常是源自和不可再分的，例外再讨论

- lock：作用于主内存的变量，它把一个变量标识为一条线程独占的状态
- unlock：作用于主内存变量，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
- read：作用于主内存的变量，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使用
- load：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中
- use：作用与工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作
- assign：作用于工作内存的变量，它把一个从执行引擎接受到的值赋给工作内存中的变变量，每当虚拟机遇到一个给变量值赋值的字节码指令时执行这个操作。
- store：作用于工作内存的变量，它把工作内存中的一个变量的值传送到主内存中，以便随后的write操作使用
- write：作用于主内存的变量，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

把一个变量从主内存复制到工作内存，就需要顺序地执行read和load操作，而把变量从工作内存同步回主内存，就需要顺序地执行store和write操作，但注意是顺序执行，并不是连续执行。除此外，java内存模型还规定了在执行上述8种基本操作时必须满足的规则。

- 不允许load和read，store和write操作之一单独出现，即不允许一个变量从主内存读出后但工作内存不接受，或者从工作内存发起了回写但是主内存不接受的情况
- 不允许一个线程丢弃它最近的assign操作，即变量在工作内存中被赋值后必须把变化同步到主内存中去
- 不允许一个线程无原因地（即没有发生过assign操作）把数据从线程的工作内存同步到主内存中区
- 一个变量只允许在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化(load或者assign)的变量，也就是说对一个变量进行use或者store操作前，必须是经过load和assign的
- 一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被一条线程重复多次，多次执行lock后，只有执行相应次数的unlock，变量才会被解锁
- 如果对一个变量进行lock操作，那么会清空工作内存中该变量的值，在执行引擎使用该变量前，需要重新执行load或assign操作初始化变量的值
- 如果一个变量事先没有被lock操作，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量
- 对一个变量进行unlock操作前，必须把此变量同步回主内存中（执行store和write操作）

volatile型变量的特殊规则

volatile的可见性
这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说都是可以立即得知的。也就是volatile变量在各个线程中是一致的。但是，并不能得出，基于volatile变量的运算在并发条件下是安全的这个结论
----是因为java里面的运算操作并非原子操作，导致volatile变量的运算在并发下一样是不安全的。

比如多线程的对volatile变量进行add自增操作。在字节码层面我们就很容易得出：getstatic指令把变量值取到操作栈顶，此时volatile关键字保证了变量值是正确的，但在执行iconst_1,iadd这些指令操作时，其他线程可能已经对变量值进行了自增操作，这时操作栈顶的数据就是过期的数据，putstatic执行指令后将较小的变量值同步回内存中，产生错误。

由于volatile变量只能保证可见性，在不符合一下两条规则的运算场景中，我们仍然需要通过加锁(使用synchronized或java.util.concurrent中的原子类)来保证原子性：

- 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值
  - 运算结果不依赖变量当前值可以理解为，a=a+1就是依赖当前值的操作，但a=4就不是一个依赖当前值的操作
- 变量不需要与其他的状态变量共同参与不受约束

volatile-禁止指令重排序优化
在有volatile修饰的变量，在进行赋值操作等之后会多执行一个lock (把寄存器的值加0的空操作，不是空指令操作)操作，这个操作相当于以个内存屏障(指重排序时不能把后面的指令排序到内存屏障之前的位置)，只有一个cpu访问内存时，并不需要内存屏障；但如果有多个cpu访问同一块内存，且其中一个在观察另一个，就需要内存屏障来保证一致性。
lock的作用是使得本cpu的cache写入了内存，该写入动作也会引起别的cpu或者别的内核无效化其cache，这种操作相当于对cache中的变量进行了一次store和write操作，让volatile变量的修改对其他cpu立即可见。

那从禁止指令重排序上来说，因为带有lock addle$0x0,(%esp)指令把修改同步到内存时，意味着之前的操作都已经完成并且进入了内存，这样便形成了“指令重排序无法越过内存屏障”的效果。

volatile结合java内存模型操作要求
假定T表示一个线程，V和W分别表示两个volatile型变量，在进行read,load,use等操作时需要满足的规则：

- 只有当线程T对变量V执行的前一个动作时load时，线程T才能对变量V执行use操作；并且只有当T对变量V执行的后一个动作是use时，线程T才可以对变量V执行load操作。也就是说线程T对V的use操作可以认为是和线程T对V的load和read操作是相关联的（这条规则要求在工作内存中，每次使用V前先从主内存中刷新出最新的值，用于保证看见其他线程对V所做操作后修改的值）
- 同上，不过是assign和store，write操作必须相关联(也就是要求在工作内存中，每次对V进行修改后，都必须同步到主内存中，保证其他线程看到对V所做的修改)
- 线程T：F是use和assign，A是load和store。线程T：G是use或assign，B是load和store。两组操作的是不同volatile变量，若F比G早，那A也要比G早(也就是volatiel修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同。）

long和double的特殊规则 java内存模型的8个操作都有原子性，但对于64位的数据类型，如果没有被volatile修饰，允许对其的读写操作划分成两次32位的操作来实现。

原子性，可见性与有序性

- 由java内存模型来直接保证的原子性操作包括read,load,assign,use,store,write。我们大致可以认为基本数据类型的访问读写是具备原子性的。如果应用场景需要更大范围的原子性保证,就是synchronized。
- 可见性：可见性是指当一个线程修改了共享变量的值，其他线程能够立即获得这个通知。除了volatile外，还有synchronized和final两个关键字可以使用。final的可见性是指被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把this的引用传递出去，那么在其他线程中就能看到final字段的值。
- 有序性：java程序中天然的有效性是指“线程内表现为串行的语义，但存在指令重排序和工作内存与主内存同步延迟”。java提供volatile和synchronized两个关键字来保证线程之间操作的有序性。volatile本身就包括了禁止指令重排序的语义，而synchronized是由“一个比啊你浪在同一个时刻只允许一条线程对其进行lock操作”这条规则获得的，这条规则决定了持有同一个锁的两个同步块只能串行地进入。

先行发生原则
java语言中的先行发生原则，是判断数据是否存在竞争，线程是否安全的主要依据。

- 程序次序规则：在一个线程内，按照程序代码顺序，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操作先行发生于后面对于同一个锁的lock操作
- volatile变量规则：对于一个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：thread对象的start方法先于此线程的每一个动作
- 线程终止规则：线程中的所有操作都先行发生于堆此线程的终止检测
- 对象中断规则：对线程interrupt()方法的调用先行发生与被中断线程的代码检测到中断事件的发生。
- 对象终结规则：一个兑现的初始化完成先行于他的finalize()方法的开始

- 传递性：操作a先于b，操作b先于c，那a先于c

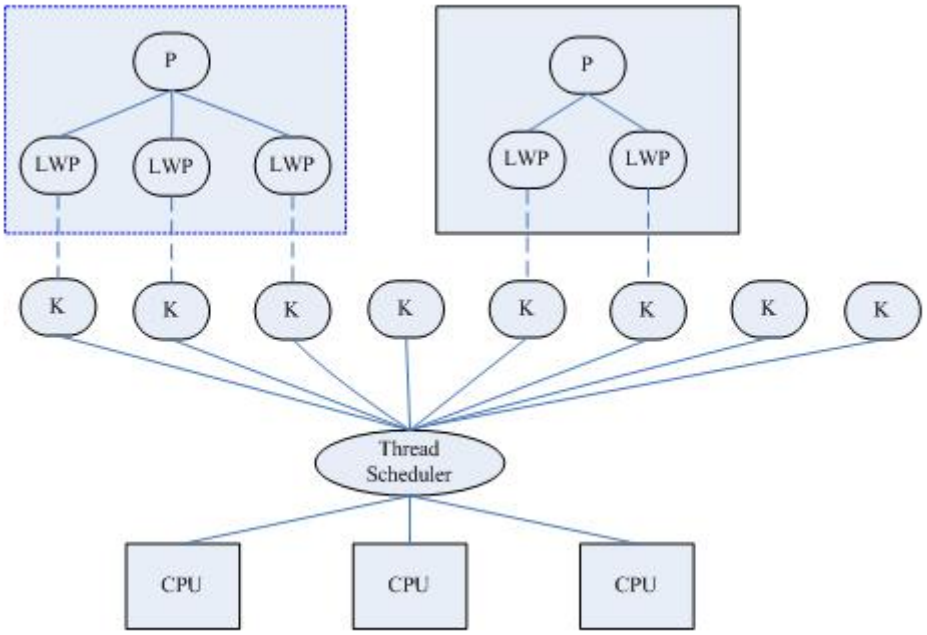
java与线程

线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源(内存地址、文件I/O等)，又可以独立调度（线程是CPU调度的基本单位）

线程的实现

使用线程的实现主要有三种方式，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。

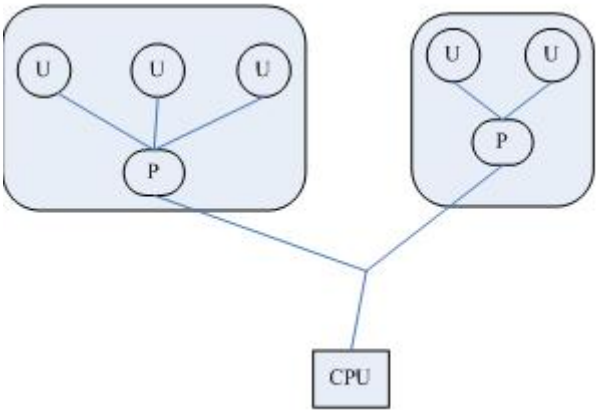
使用内核线程实现：
内核线程(Kernel-Level Thread, KLT)就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫做多线程内核。
程序是通过使用内核线程的高级接口--轻量级进程(light weight process,LWP)，也就是我们常说的线程。由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核线程之间1:1的关系称为一对一的线程模型。



它的优势是：由于每个LWP都与一个特定的内核线程关联，因此每个LWP都是一个独立的线程调度单元。即使有一个LWP在系统调用中阻塞，也不会影响整个进程的执行。
局限是：首先，大多数LWP的操作，如建立、析构以及同步，都需要进行系统调用。系统调用的代价相对较高：需要在user mode和kernel mode中切换。其次，每个LWP都需要有一个内核线程支持，因此LWP要消耗内核资源（内核线程的栈空间）。因此一个系统不能支持大量的LWP。

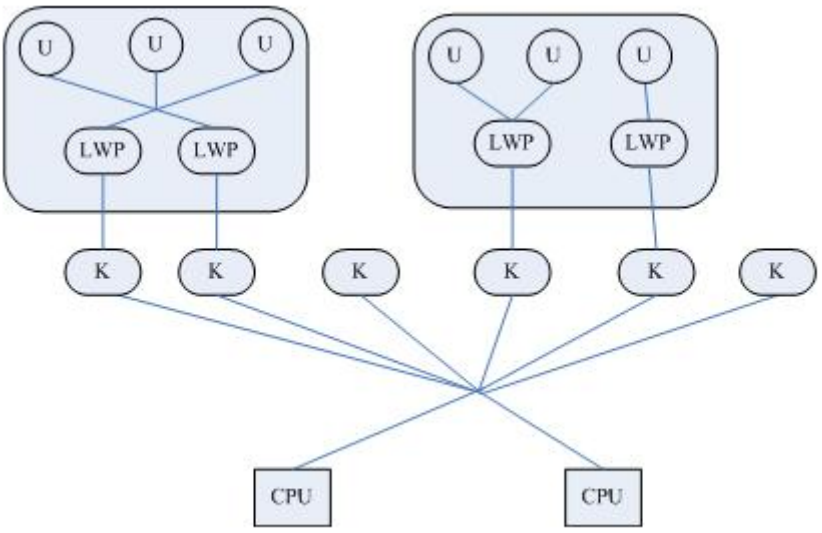
使用用户线程实现 LWP虽然本质上属于用户线程，但LWP线程库是建立在内核之上的，LWP的许多操作都要进行系统调用，因此效率不高。而这里的用户线程指的是完全建立在用户空间的线程库，用户线程的建立，同步，销毁，调度完全在用户空间完成，不需要内核的帮助。因此这种线程的操作是

极其快速的且低消耗的。也可以支持更大规模的线程数量。



优势是不需要内核支援，劣势也是在这里，所有的线程操作需要用户自己考虑，包括线程的创建，切换调度，以及阻塞如何处理等。

使用用户线程加轻量级进程混合实现 用户线程库还是完全建立在用户空间中，因此用户线程的操作还是很廉价，因此可以建立任意多需要的用户线程。操作系统提供了 LWP 作为用户线程和内核线程之间的桥梁。 LWP 还是和前面提到的一样，具有内核线程支持，是内核的调度单元，并且用户线程的系统调用要通过 LWP ，因此进程中某个用户线程的阻塞不会影响整个进程的执行。用户线程库将建立的用户线程关联到 LWP 上， LWP 与用户线程的数量不一定一致。当内核调度到某个 LWP 上时，此时与该 LWP 关联的用户线程就被执行。



线程调度

线程调度是指操作系统为线程分配处理器使用权的过程，主要调度方式有两种，分别是协同式线程调度和抢占式线程调度。

协同式线程调度：线程的执行时间由线程本身控制，线程把自己的工作执行完成后，要主动通知系统切换到另外一个线程上。好处是由于线程把自己的事干完才会进行切换，切换操作已知，不存在线程同步的问题。但会出现严重的阻塞问题。

抢占式调度：每个线程由系统来分配执行时间，线程的切换不由线程本身来决定。优点是线程的执行时间系统可控，也不会有一个线程导致整个进程阻塞的情况。

线程的状态

- 新建(new):创建后尚未启动的线程处于这个状态
- 运行(runnable)：runable包括了操作系统中线程状态的running和ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着CPU为他分配执行时间
- 无限期等待(waiting)：处于这种状态的线程不会被分配cpu执行时间，它要等待被其他线程显式地唤醒，一下方法会让线程陷入无限期的等待状态
  - 没有设置timeout的Object.wait()方法
  - 没有设置timeout参数的Thread.join()方法
  - LockSupport.park()方法
- 限期等待(timed waiting)：不会被分配CPU时间，在一定时间后被系统自动唤醒
  - Thread.sleep()
  - 设置了timeout参数的Object.wait()或者Thread.join()
  - LockSupport.parkNanos()方法或者LockSupport.parkUntil()方法
- 阻塞(blocked)：阻塞是在等待着获取到一个排它锁，这个时间将在另一个线程放弃这个锁的时候发生。
- 结束(terminated)：已经终止线程的线程状态。

线程安全与锁优化

线程安全的定义：当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下调度和交替进行，也不需要进行额外的同步，或者在调用方法进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的。

线程安全

我们将java语言中各种操作共享的数据分为：不可变，绝对线程安全，相对线程安全，线程兼容和线程对立

不可变
在java语言中，不可变的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障措施。以finla为例，只有一个不可变的对象被正确地构建出来(没有发生this引用逃逸的情况)，那其外部的可见状态用于也不会改变，永远也不会看到它在多个线程之中处于不一致的状态。
java语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用final关键字修饰就可以保证它不变。如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行，其中最简单的就是把对象中带有状态的变量都声明为final，这样在构造函数后，它就是不可变的。

绝对线程安全
通常我们所说的绝对线程安全，是指一个类要达到不管运行时环境如何，调用者都不需要任何额外的同步措施。这个是需要极大的代价的。即使是java api中标记的线程安全的类，

```
public class vector {
    private static Vector<Integer> vector=new Vector<>();
```

```
    public static void main(String[] args){
        while(true){
            for (int i=0;i<10;i++){
                vector.add(i);
            }
            Thread removedThread=new Thread(new Runnable() {
                @Override
                public void run() {
                    for(int i=0;i<vector.size();i++){
                        vector.remove(i);
                    }
                }
            });

            Thread printThread=new Thread(new Runnable() {
                @Override
                public void run() {
                    for(int i=0;i<vector.size();i++){
                        System.out.println((vector.get(i)));
                    }
                }
            });
            removedThread.start();
            printThread.start();
            //不产生过多的线程，否则会导致操作系统假死
            while(Thread.activeCount()>30);
        }
    }
```

这里vecotr使用的get(),remove(),size()方法都是同步的，但是在多线程环境中，如果不在方法调用端做额外的同步的话，仍然是不安全的，如果一个线程恰好在错误的时间里删除了一个元素，导致i不可用，就会抛出arrayindexoutofboundsException的错误。

我们需要对两个线程方法加上

```
hread removedThread=new Thread(new Runnable() {
                @Override
                public void run() {
                    /**
                     * //使用synchronized锁机制将操作对象进行锁定，
                     * 当执行这个remove操作的时候，其他vector不可以访问当时的这个资源i
                     */
                    synchronized (vector) {
                        for (int i = 0; i < vector.size(); i++) {
                            vector.remove(i);
                        }
                    }
                }
            });

            Thread printThread=new Thread(new Runnable() {
                @Override
                public void run() {
```

```
                    for(int i=0;i<vector.size();i++){
                        System.out.println((vector.get(i)));
                    }
                }
            });
```

相对线程安全 它需要保证这个对象单独的操作是线程安全的，我们在调用的时候不需要额外的保障措施，但对于特定顺序的连续调用，需要在调用端使用额外的同步手段来保证调用的正确性。如上面所示

线程兼容 对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全的使用。比如arraylist或者hashmap

线程对立 是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。典型的例子就是thread类的suspend()和resume()，如果同时持有一个线程对象，一个尝试中断，一个尝试恢复，如果并发进行，无论调用时是否进行了同步，都存在死锁风险。

线程安全的实现方法

互斥同步 互斥同步是一种常见的并发正确性保障手段。同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个线程使用，而互斥是实现同步的一种手段，临界区，互斥量和信号量都是主要的互斥实现方式。因此在这4个字里，互斥是方法，同步是目的。

而在java中最基本的互斥同步手段就是synchronized。synchronized关键字经过编译后，会在同步块钱分别形成monitorenter和monitorexit，这两个字节码都需要一个reference类型的参数来指明要锁定和解锁的对象。其次synchronized锁是可重入的，不会把自己锁死，但因为java线程是直接映射到操作系统的原生线程上，所以阻塞和唤醒一个线程，都需要操作系统来完成，从用户态转为核心态，消耗很多cpu时间。所以synchronized是一个重量级的锁，不轻易使用，但现在也通过通知线程阻塞前加入自选等待的过程，来避免进入核心态。

还有一个关键字是java.util.concurrent中的重入锁reetranlock。但reetranlock是api层面的互斥锁，synchronized是原生语法层次的互斥。reentranlock额外的特点

- 等待可中断，指持有锁的线程长期不释放锁的时候，正在等待的线层可以选择放弃等待，改为处理其他事情
- 可以实现公平锁，公平锁是指必须按照申请锁的时间顺序来一次获得锁，但非公平锁不保证这一点，当锁被释放时，任何一个等待锁的线程都有机会获得锁。sychronized是非公平锁，reentranlock既可以是公平锁，也可以是非公平锁
- 锁绑定多个条件，reentranlock对象可以同时绑定多个condition对象，而synchronized锁对象的wait(),notify()或notifyAll()可以实现一个隐含的条件，若要和多一个的条件关联，就不得不额外地添加一个锁。reentranlock只需要多次调用newCondition()即可。

非阻塞同步 互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步。从处理问题的方式上说，互斥同步属于一种悲观的并发策略。

基于冲突检测的乐观并发策略：就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取其他的不久措施(最常见的是不断重试，直到成功).这种乐观的额并发策略许多实现都不需要把线程挂起，称这种阻塞是非同步阻塞。

但需要硬件指令集的发展，保证操作和冲突检测两个步骤具有原子性。硬件要保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就完成，如

- 测试并设置(test and set)
- 获取并增加(fetch and increment)
- 交换(swap)
- 比较并交换(compare and swap)
- 加载链接/条件存储(load linked/stroe conditional)

CAS指令需要3个操作数，分别是内存位置(V)，旧的预期值(A)和新值(B)，每次cas执行指令时，仅当V符合旧预期值A时，处理值才会用新值B更新V，否则就不进行更新，但无论是否更新了V，都会返回V的旧值。这一系列步骤是一个原子操作。

但会存在ABA的覆盖问题无法解决。

无同步方案 同步是保证数据共享争用时的正确性的手段，如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性，因此会有一些代码天生就是线程安全的：

- 可重入代码：也称为纯代码，可以在代码执行的任何阶段中断它，转而执行另外一段代码，而在控制权返回后，原来的程序不会出现任何错误。如何判断代码是否具有可重入性：返回结果是可以预测的，只要输入了相同的数据，就能返回相同的结果，他就是满足可重入性要求，是线程安全的。
- 线程本地存储：如果一段代码中所需的数据必须与其他代码共享，那就判断这些共享数据的代码是否能保证在同一个线程里，如果可以，那我们就可以把共享数据的可见范围限制在同一个线程内，这样无需同步也能保证线程之间出现数据争用的问题。比如threadlocal类对象。

锁优化

自旋锁与自适应自旋

自旋锁是计算机科学用于多线程同步的一种锁，线程反复检查锁变量是否可用。由于线程在这一过程中保持执行，因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁。

但由于一直在尝试自旋，虽然少了线程切换的开销，会一直占用cpu。因此自选等待的时间必须要有一定限度，如果自旋超过一定次数没有成功获得锁，就应该按传统方式挂起锁。

自适应自旋是指，自旋时间不再固定，而是由前一次在同一个锁上的自旋时间以及锁的拥有者状态来决定。

锁消除

锁消除是Java虚拟机在JIT编译是，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过锁消除，可以节省毫无意义的请求锁时间。

出现原因：主要判断依据就是来源于逃逸分析的数据支持，如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上的数据对待，认为它们是线程私有，同步锁无须进行。

```
public String concatString(String s1, String s2, String s3){
    return s1+s2+s3;
}


public String concatString(String s1, String s2, String s3){
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}
```

本身对于String的连接操作JVM会转化成StringBudiler进行连接，既然转化成了sb，那么其实sb是一个局部变量，sb的所有引用不会套溢出concatString的局部方法。对于append操作加锁并没有任何效果，因为其他线程在外部是访问不到它的，就可以进行锁的消除。

锁粗化

原则上对于锁的使用都是需要细化到尽量小的范围，大部分情况下，都是正确的，但是一些列的加锁和解锁操作很是麻烦。造成性能损坏。

所以有了锁的粗化的概念。类似上图中的append方法，如果虚拟机探测到有一串零碎操作都是对同一对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部，也就是在第一个和最后一个append操作之后。

轻量级锁

通过同步对象的对象头信息进行是否有锁的判定。加锁和解锁的过程都是CAS的原子操作。

偏向锁

目的是消除数据在无竞争情况下的同步原语，也就是说偏向锁就是在无竞争的情况下把整个同步都消掉，连CAS操作都不做了。