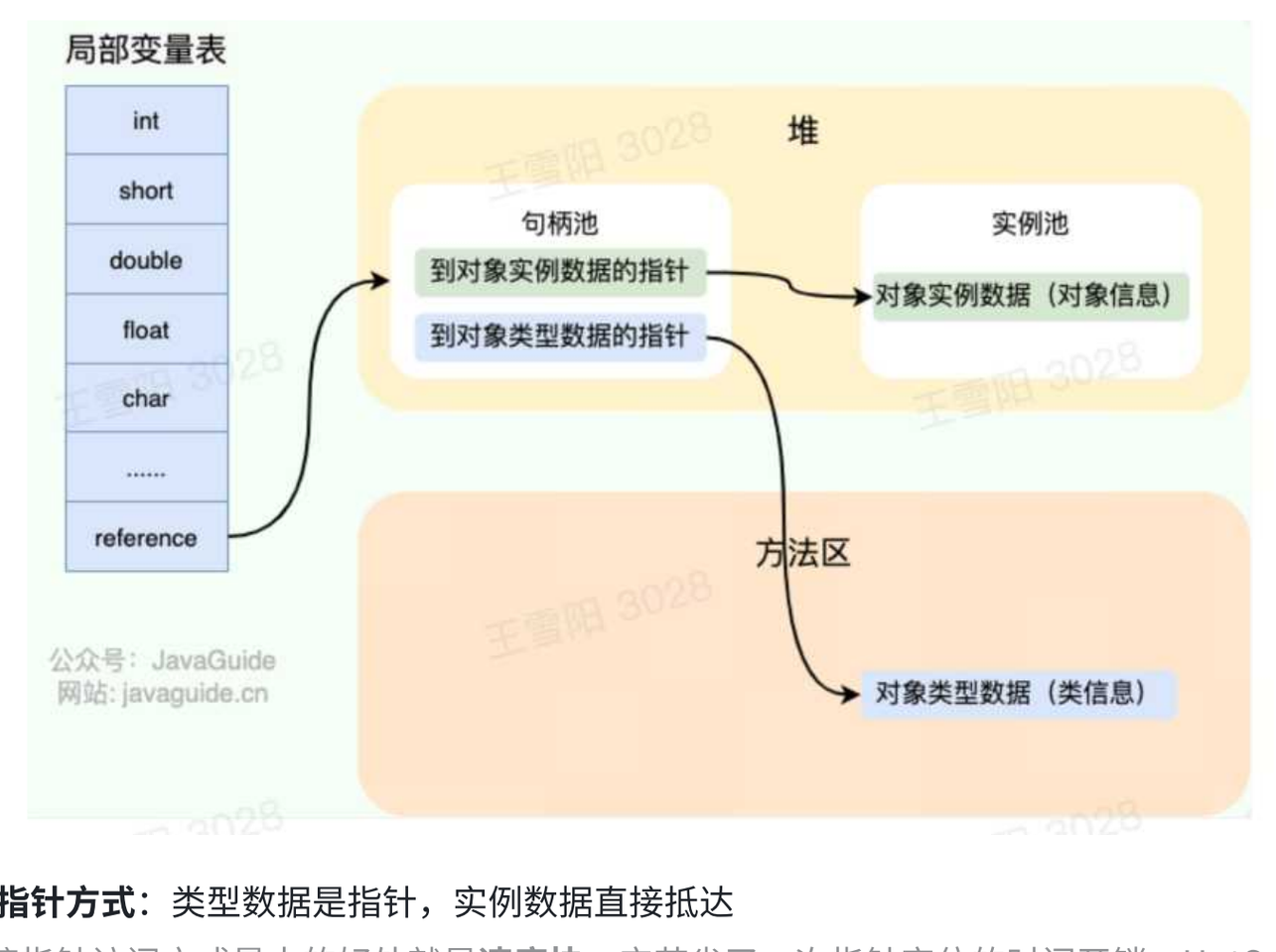
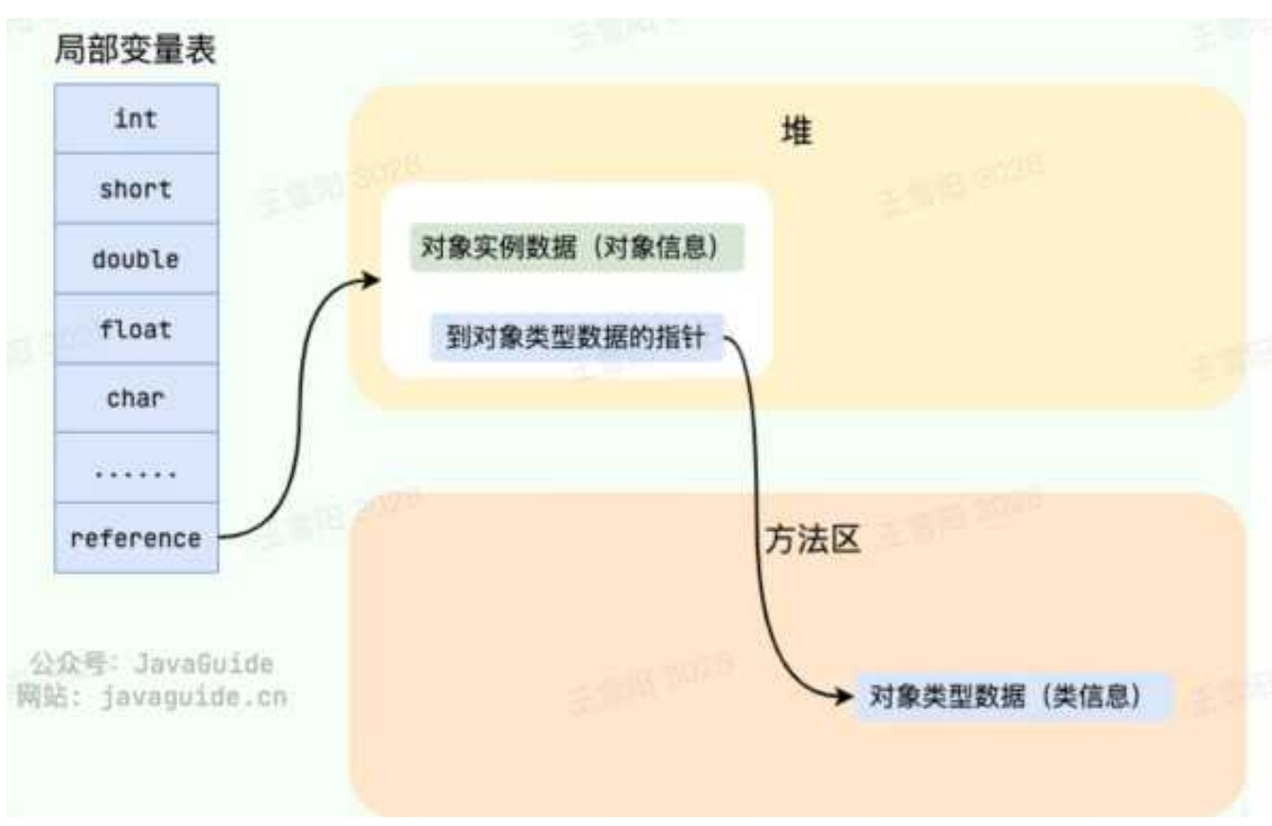
JVM

- 内存区域
  - 线程共享
    - 堆
      存放实例对象
      - 字符串常量池
        原本在永久代，为了更高效回收字符串内容。1.7后移到堆中
      - 静态变量
      - 区域划分
        - 新生代
          - Eden
            第一次GC后仍然存活
          - s0
          - s1
        - 老年代
    - 输入文本
  - 线程私有
    - 虚拟机栈
      为执行java方法服务
      - 栈帧
        - 局部变量表
          存放编译期可知的数据类型例如：基本数据类型、对象引用
        - 操作数栈
          中转站存储中间结果
        - 动态链接
          符号引用转化为直接引用
        - 方法返回地址
    - 本地方法栈
      类似虚拟机栈，为Native方法服务
    - 程序计数器
  - 本地内存
    - 元空间
      永久被 Metaspace(元空间) 取代，元空间使用的是本地内存
    - 直接内存
      不受Java堆的限制
  - 方法区（抽象）
    - 运行时常量池
    - JIT代码缓存
    - 方法区的实现
      - 永久代
        HotSpot在JDK8以后已被 Metaspace(元空间) 取代，元空间使用的是本地内存
        - 被取代原因
          - 整个永久代有一个 JVM 本身设置的固定大小上限无法进行调整，更容易溢出
          - 元空间存放类的元数据，由系统的实际可用空间来控制，这样能加载的类就更多
          - 给GC带来不必要的复杂度，回收效率偏低
        - 用如下参数调节
          -XX:PermSize=N //方法区 (永久代) 初始大小
          -XX:MaxPermSize=N //方法区 (永久代) 最大大小,超过这个值将会抛出 OutOfMemoryError 异常:java.lang.OutOfMemoryError: PermGen
- HotSpot对象
  HotSpot虚拟机要求8的整数倍对齐
  - 创建过程
    - 类加载检查
      在常量池中定位到类的**符号引用**，并检查这个符号引用代表的类是否已被加载过、解析和初始化过
    - 分配内存
      - 指针碰撞
        内存规整的情况下，没有内存碎片
        Serial, ParNew收集器使用
      - 空闲列表
        内存不规整的情况下使用，CMS收集器
      - 内存并发分配
        - CAS
        - TLAB
          每一个线程预先在 Eden 区分配一块儿内存，不够时再采用CAS方式
    - 初始化零值
    - 设置对象头
      这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息
    - 执行init方法
  - 内存布局
    - 对象头
      - 标记字段
        如哈希码（HashCode）、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等
      - 指针类型
        通过这个指针来确定这个对象是哪个类的实例
    - 实例数据
    - 对齐填充
  - 对象的访问
    - 句柄方式：实例数据和类型数据都是指针
      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

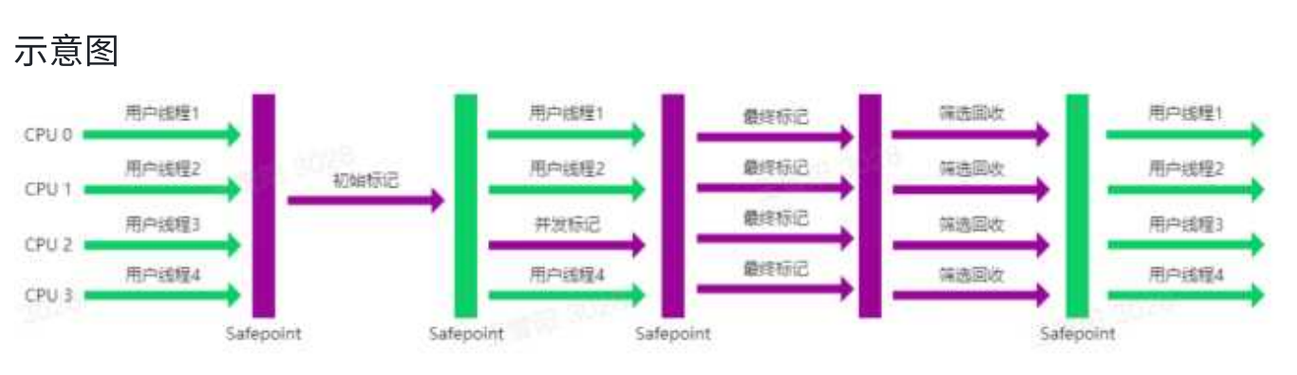


    - **直接指针方式**：类型数据是指针，实例数据直接抵达
      直接指针访问方式最大的好处就是**速度快**，它节省了一次指针定位的时间开销。HotSpot采用此方式



- 垃圾回收机制
  - 分配机制
    - 对象优先分配在Eden区
    - 大对象（比如数组，字符串）直接进入老年代
      避免进入新生代占用大量空间导致频繁GC
    - 长时间存活对象进入老年代
      熬过一次GC年龄+1，15岁（CMS是6）进入老年代
  - GC
    - 部分收集
      - 新生代收集MinorGC
        Eden区域不足
      - 老年代收集MajorGC
      - 混合收集
    - 整堆收集FullGC
    - 空间分配担保
      老年代能否容纳所有新生代的对象
      - 能容纳所有新生代
      - 不能容纳，是否允许「担保失败和冒险」
        - 允许：minorGC
        - 不允许：FullGC
  - 死亡对象判定
    - 引用计数法
      难以解决对象之间的循环引用问题
    - 可达性分析法
      - GC-Roots
        - **虚拟机栈（局部变量）**引用的对象
        - **本地方法栈**（Native方法）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
        - 被同步锁持有的对象
        - JNI引用的对象
      - 引用类型
        - **强引用**
          宁愿抛错也不回收
        - **软引用**
          GC时空间足够则不回收，可用于内存敏感的高速缓存
        - 弱引用
          比软引用生命周期更短，每次GC必然回收
        - 虚引用
          任何时候都可以回收，跟没引用一样
    - 无用类的判定
      同时满足才「可以」被回收
      - 所有实例被回收
      - 类加载器被回收
      - java.lang.Class对象没被引用（任何地方无法通过反射访问）
  - GC算法
    - 标记-清除
    - 标记-复制
      内存一分为二
    - 标记-整理
      内存整理到一侧
    - 分代收集：这也是分成新生代和老年代的原因
      根据生命周期采用不同策略
      新生代：复制，因为存活的少
      老年代：清除/整理
  - 垃圾收集器
    - Serial串行收集器
      复制&整理；新生代老年代都单线程
    - ParNew多线程串行收集器
      复制&整理；新生代多线程，老年代单线程
    - Parallel Scavenge收集器
      复制；多线程；关注吞吐量；新生老年代都多线程
    - CMS收集器
      最短回收停顿为目标；真正的并发（但标记仍然不是）
      - 初始标记
        串行；标记与GCroots相连的对象
      - 并发标记
        从GCroots标记可达对象
      - 重新标记
        修正并发标记期间的变化
      - 并发清除
      - 示意图：

    - G1收集器
      高吞吐且极高概率满足停顿需求
      - 初始标记
      - 并发标记
      - 最终标记
      - 筛选回收
      - 示意图



      - 特点
        - 并行与并发
        - 分代收集
        - 空间整合
        - 可预测停顿
      - G1 收集器在后台维护了一个**优先列表**，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)
    - ZGC
      复制；暂停时间短且不受堆大小影响
- 类
  - 类结构

```
u4          magic; //Class 文件的标志
u2          minor_version;//Class 的小版本号
u2          major_version;//Class 的大版本号
u2          constant_pool_count;//常量池的数量
cp_info     constant_pool[constant_pool_count-1];//常量池
u2          access_flags;//Class 的访问标记
u2          this_class;//当前类
u2          super_class;//父类
u2          interfaces_count;//接口数量
u2          interfaces[interfaces_count];//一个类可以实现多个接口
u2          fields_count;//字段数量
field_info  fields[fields_count];//一个类可以有多个字段
u2          methods_count;//方法数量
method_info methods[methods_count];//一个类可以有个多个方法
u2          attributes_count;//此类的属性表中的属性数
attribute_info attributes[attributes_count];//属性表集合
```

  - 类加载过程
    - 加载
      类加载器：主要加载字节码到JVM中
      - 加载规则
        动态加载
        - JVM判断是不是同一个类不仅要看类名，也要看是不是同一个类加载器
      - 内置的类加载器
        - **BootstrapClassLoader(启动类加载器)**
          最顶层的加载类（C++实现），加载JDK内部核心库
          - 每个 ClassLoader 可以通过getParent()获取其父 ClassLoader，如果获取到 ClassLoader 为null的话，那么该类是通过 BootstrapClassLoader 加载的
          - 除了它由JVM实现，其他都继承自ClassLoader。
        - **ExtensionClassLoader(扩展类加载器)**
        - **AppClassLoader(应用程序类加载器)**
      - 自定义类加载器
        同样继承自ClasLoader，两个重要方法
        1. loadClass()
        2. findClass()
        - 我们不想打破双亲委派模型，就重写 ClassLoader 类中的 findClass() 方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。
        - 如果想打破双亲委派模型则需要重写 loadClass() 方法
      - 双亲委派模型
        加载一个类的时候，具体是哪个类加载器加载
        可以避免类的重复加载，保证Java核心API不被篡改
        - 每当一个类加载器接收到加载请求时，它会先将**请求转发给父类加载器**。在**父类加载器没有找到**所请求的类的情况下，该子类加载器才会尝试去加载
        - 详细流程
          - 在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载（每个父类加载器都会走一遍这个流程）。
          - 类加载器在进行类加载的时候，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成（调用父加载器 loadClass()方法来加载类）。这样的话，所有的请求最终都会传送到顶层的启动类加载器 BootstrapClassLoader 中
          - 只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载（调用自己的 findClass() 方法来加载类）
          - 如果子类加载器也无法加载这个类，那么它会抛出一个 ClassNotFoundException 异常
    - 链接
      - 验证
        符合《Java 虚拟机规范》的全部约束
        - 文件格式验证
          是否符合class文件规范
        - 元数据验证
          字节码语意检查
        - 字节码验证
          程序语意检查
        - 符号引用验证
          类正确性检查
      - 准备
        分配内存并设置变量初始值
      - 解析
        将常量池的符号引用替换为直接引用
    - 初始化：clinit方法，类加载最后一步
      生命周期再加上：使用和卸载