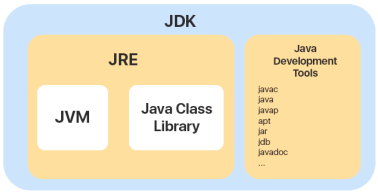
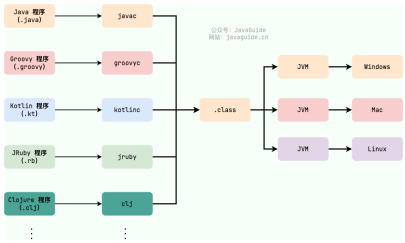
Java

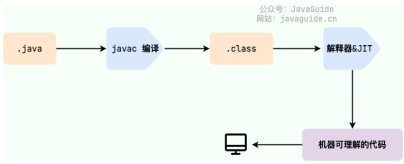
- **基础**
  - 常见面试题上
    - 基本概念
      - JVM&JDK&JRE



        - JVM,运行java字节码的虚拟机



        - JDK
          - Java Development Kit，功能齐全sdk，包含JRE
        - JRE
          - 运行环境，包括jvm，基础类库
      - 字节码，.class文件
        - jvm理解，只面向jvm
        - 编译和解释并存
          -



          - JIT（Just in Time Compilation），运行时编译并保存热点代码的机器码
          - 先编译后解释
      - AOT(Ahead of Time Compilation)

- 程序执行前编译，静态编译，避免JIT预热，提高性能降低开销，减少内存占用，增强安全性，适合云原生常见
- AOT无法支持动态特性：反射、动态代理，动态加载，JNI等，spring用到这些框架

- Java vs C++
  - Java：不提供指针、类单继承、接口多继承、自动内存管理垃圾回收机制、只支持方法重载
  - C++：指针、多继承、手动释放内存、同时支持方法重载和操作符重载

- 基本数据类型
  - 包装类型：可用于泛型，存储在堆中，默认值为空，比较方法：equals
    - 包装类型的缓存机制
      - Byte,Short,Integer,Long默认 [-128，127] 的相应类型的缓存数据；Character 创建了数值在 [0,127] 范围的缓存数据，Boolean 直接返回 True or False
  - 基本类型：不可用于泛型，存储在jvm的堆，占用空间更小，有默认值且非空，比较==
  - 自动装箱和拆箱
    - 装箱：将基本类型用它们对应的引用类型包装起来Integer i = 10 等价于 Integer i = Integer.valueOf(10)
      int n = i 等价于 int n = i.intValue();
    - 拆箱：将包装类型转换为基本数据类型Integer i = 10; //装箱
      int n = i; //拆箱
  - 精度问题
    - 二进制，会截断，近似值
    - BigDecimal 可以实现对浮点数的运算，不会造成精度丢失。
    - BigInteger > long > int

- 变量
  - 成员（类），可被static，public等修饰。static修饰则存属于类，反之，属于对象实例，存在堆中。随对象创建存在。自动赋默认值
  - 局部（方法，代码块），无法被static修饰。属于对象，对象在堆内存。随防范生成消亡，不会自动赋值
  - why成员变量默认值：读取随机值出现意外、成员变量运行时赋值，无法判断

- 方法

- 静态方法无法调用非静态成员
  - 静态方法属于类，类加载则分配内存，通过类名直接访问，非静态成员属于实例对象，实例化后才存在
  - 静态方法只能访问静态成员变量和静态方法，因为实例成员在对象 实例化后才存在，静态方法在类加载时存在
- 重载
  - 同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理
    - 修改方法的外部参数、返回值等
- 重写
  - 发生在运行期，子类对父类的允许访问的方法的实现过程进行重新编写（除private/final/static/构造方法），形参不可变，方法名不可变，返回值类型需更小（void可不修改），异常更小，方法访问权限更大
    - 修改方法的内部逻辑
- 可变长参数(String... args, String arg)
  - 方法重载优先匹配固定长参数

- 常见面试题中
  - 面向对象基础
    - 对象相等和引用相等
      - 对象：内容
      - 引用：内存地址
    - 面向对象三大特征
      - 封装、继承、多态
        - 继承：子类拥有父类所有属性和方法，拥有但不访问私有属性方法；子类可以拥有自己属性和方法，可用自己的方式实现父类方法
        - 多态：一个对象有多种状态。父类的引用指向子类的实例
          - 对象类型和引用类型之间具有继承（类）/实现（接口）的关系
          - 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定
          - 多态不能调用“只在子类存在但在父类不存在”的方法
          - 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法
    - 接口/抽象类

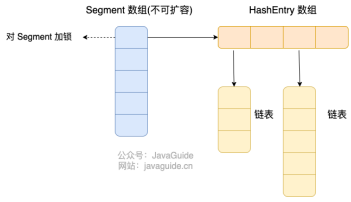
- 共同点：不能实例化，都可以包含抽象方法，都可以有默认实现的方法（Java 8 default来定义默认方法）
- 区别
  - 接口主要用于对类的行为进行约束，你实现了某个接口就具有了对应的行为。抽象类主要用于代码复用，强调的是所属关系。
  - 一个类只能继承一个类，但是可以实现多个接口
  - 接口中的成员变量只能是 public static final 类型的，不能被修改且必须有初始值，而抽象类的成员变量默认 default，可在子类中被重新定义，也可被重新赋值。
- 深拷贝、浅拷贝
  - 深：完全复制整个对象，包括这个对象所包含的内部对象
  - 浅：创建一个新的对象，若对象内部属性时引用，则赋值该内存地址
  - 引用拷贝：两个不同的引用指向同一个对象
- Object（所有类的父类）
  - 常见方法：getClass，hashCode，equals（比较内存地址，Strings重写比较内容）,clone,toString(),notify(),wait等
  - 为何要有hashCode()并同时有equals方法
    - 以hashSet为例，不可加入重复对象，因此先计算hashCode，不同则加入，相同则再用equals判断是否为同一对象
  - 重写equals必须重写hashCode()
    - 对象相等,hashCode一定相等
    - equals判断相等,hashCode不一定相等
- String
  - String、StringBuffer、StringBuilder
    - String不可变(String 类中使用 final 关键字修饰字符数组来保存字符串，因此会新建对象)，常量；StringBuffer线程安全（同步锁）；性能：String修改会生成新的对象，StringBuffer操作对象本身
    - 操作少量数据用String；单线程StringBuilder；多线程StringBuffer
    - String不可变：final修饰，未提供修改方法；final修饰不可被继承，避免子类破坏
  - 字符串拼接
    - +， += 为例重载过的运算符，实际上是通过 StringBuilder 调用 append() 方法实现的，拼接完成之后调用 toString() 得到一个 String 对象

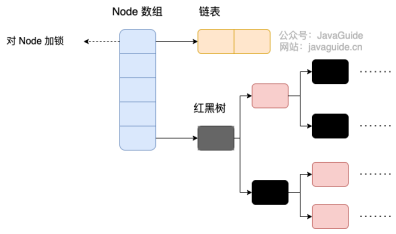
        - 循环会创建多个StringBuilder对象；直接使用 StringBuilder 对象进行字符串拼接即可
    - 字符串常量池，主要目的是为了避免字符串的重复创建，创建时先检查常量池
- 常见面试题下
    - Exception & Error，继承Throwable
    - 泛型，新特性，通过泛型参数指明传入对象；使用场景：自定义接口返回类，Excel工具类
        - 泛型类，Generic
            - Generic<Integer> genericInteger = new Generic<Integer>(123456);
        - 泛型接口
        - 泛型方法
    - 反射，分析类以及执行类中方法的能力
        - Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制（动态代理：运行时不改源码从而修改方法）
        - 注解，比如@Component-->Spring bean，@Value读取配置，基于反射分析类获取注解后进一步处理
    - 注解Annotation，用于修饰类、方法或者变量，提供某些信息供程序在编译或者运行时使用
        - 解析方法
            - 编译期直接扫描，@Override
            - 运行期通过反射处理, @Value, @Component
    - 序列化（会话层）
        - 序列化
            - 将数据结构或对象转换成二进制字节流的过程
        - 反序列化
            - 将在序列化过程中所生成的二进制字节流转换成数据结构或者对象的过程
        - 场景
            - 网络传输
            - 存储到文件
            - 存储到数据库
            - 存储到内存

- transient修饰可不序列化
- IO
  - 四个抽象类
    - InputStream/Reader，字节/字符（字符流：不知道编码，字节容易乱码）
    - OutputStream/Writer，字节/字符
  - 设计模式
    - 装饰器
      - 不改变原有对象扩展其功能
    - 适配器
      - 用于接口互不兼容的类的协调工作
  - BIO/NIO/AIO
    - BIO同步阻塞 IO 模型
      - 程序发起 read 调用后，会一直阻塞，直到内核把数据拷贝到用户空间，连接数量低
      - 同步非阻塞 IO：一直发read，轮询，耗cpu
    - Non-blocking IO I/O多路复用
      - 高负载、高并发的（网络）应用
      - 先询问（selector选择器，多路复用器）数据是否准备就绪，再发read，read'过程中阻塞
    - AIO（NIO2）异步 IO 模型，不会阻塞

集合

- Collection单一
  - Set,非线程安全
    - HashSet，哈希表
      - LinkedHashSet，链表+哈希表
    - TreeSet有序,红黑树(自平衡的排序二叉树)
    - Comparable & Comparator，均用于排序
    - 无序性：哈希值决定，不按数组索引顺序添加
      不可重复性：equals判断不等，重写equals & hashCode
  - Queue
    - PriorityQueue实现小顶堆

- Queue：单边插入删除
  Deque：双边
- List
  - ArrayList
  - vector
  - LinkedList
    - why不能实现RandomAccess，因为该接口是根据索引随机访问，链表不支持根据索引访问
- Map键值对key-value，key不可重复
  - HashMap
    - 数组+链表（拉链法）
    - 链表长度>=8，变为红黑树
    - 如果当前数组的长度小于 64，那么会选择先进行数组扩容，而不是转换为红黑树
  - LinkedHashMap，在HashMap基础上增加双向链表
  - HashTable，线程安全（对整个hashtable上锁）
  - TreeMap红黑树
  - ConcurrentHashMap（线程安全）
    - 分段数组+链表+synchronized锁
      - 

    - JDK1.8 Node数组+链表+红黑树（Node + CAS + synchronized）
      synchronized 只锁定当前链表或红黑二叉树的首节点，只要 hash 不冲突，就不会产生并发
      - 

- key不可为null：无法区分这个键是否存在于 ConcurrentHashMap 中，还是根本没有这个键；value不可Null：无法区分这个值是否是真正存储在 ConcurrentHashMap 中的，还是因为找不到对应的键而返回的

并发

线程&进程

- 从JVM角度考虑
- 为什么要使用多线程呢?
  - 从计算机角度来说主要是为了充分利用多核CPU 的能力，从项目角度来说主要是为了提升系统的性能
- 线程的生命周期和状态
  - 6 种状态（NEW 、 RUNNABLE 、 BLOCKED 、 WAITING 、 TIME_WAITING 、 TERMINATE
    D ）
- 什么是线程死锁?如何避免死锁?如何预防和避免线程死锁?
  - 双方争夺并等待对方释放资源
  - 互斥、占有等待、不可剥夺、循环等待
  - 破坏占有和等待条件（一开始就分配所需资源）；破坏不可抢占条件(拿不到其他资源必须释放已经拿到的资源)；破坏循环等待条件（设置线程的请求顺序）

synchronized 关键字

- 修饰方法和代码块
  - 静态方法：占用当前类的锁、非静态方法：占用对象的锁，所以二者不互斥
- JDK1.6 之后的 synchronized 底层做了哪些优化？锁升级原理了解吗?
  - synchronized 引入了大量的优化如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销，这些优化让 synchronized 锁的效率提升了很多
- 底层实现：早期互斥锁、后来自旋锁、轻量级锁、偏向锁
- 锁升级：锁的状态从低到高的切换
  - 无锁：对象刚创建
  - 偏向锁：只有一个线程访问同步块
  - 轻量级锁：另一个线程尝试获取已被某个线程占有的偏向锁
  - 重量级锁：锁的竞争达到一定程度，自旋无法互获得锁的线程变多
- 自旋锁

- 线程尝试获得锁时不阻塞或者挂起，而是先循环检查是否锁被释放（线程自旋），减少线程上下文切换的开销

- synchronized 和 volatile 有什么区别?
  - volatile 是线程同步的轻量级实现，性能比synchronized要好；volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块 。volatile 能保证数据的可见性，但不能保证数据的原子性。synchronized两者都能保证。volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。
- ReentrantLock 是什么?
  - ReentrantLock 实现了Lock 接口，基于 AQS接口，可重入且独占式的锁。更灵活、更强大，增加了轮询、超时、中断、公平锁和非公平锁等高级功能
  - synchronized 和 ReentrantLock 有什么区别?
    - 都是可重入锁：线程可以再次获取自己的内部锁
    - synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API
    - ReentrantLock 比 synchronized 增加了一些高级功能
- happens-before
  - 判断数据是否存在竞争、线程是否安全
  - 指定两个操作之间的执行顺序，以确保内存的可见性。如果一个操作happens-before另一个操作，那么第一个操作的结果对后一个操作是可见的。
  - 遵守了happens-before规则，JMM就能保证一个线程对共享变量的写入操作对其他线程来说是可见的。
- ThreadLocal
  - 让每个线程绑定自己的值，可以将ThreadLocal类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。
  - ThreadLocal变量，访问这个变量的每个线程都会有这个变量的本地副本。他们可以使用 get() 和 set() 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。
- 线程池
  - 通过ThreadPoolExecutor构造函数来创建；通过 Executor 框架的工具类 Executors 来创建
  - 核心参数
    - corePoolSize、WorkQueue、maximumPoolSize、KeepAliveTime(线程池的线程数量大于corePoolSize，不会立即销毁，等待一段时间后销毁)
  - 饱和策略

- Abort策略：抛出 RejectedExecutionException来拒绝新任务的处理

- 如何设置线程池大小
  - 根据任务：IO密集（2*cpu核心数）/CPU密集（cpu核心数+1）

- 乐观锁和悲观锁的区别
  - 悲观锁：共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。synchronized和ReentrantLock等独占锁就是悲观锁思想的实现
  - 乐观锁：乐观锁总是假设最好的情况，认为共享资源每次被访问的时候不会出现问题，只是在提交修改的时候去验证对应的资源（也就是数据）是否被其它线程修改了。
    - 多版本控制，每次给数据加上version
    - CAS：CAS 的全称是 Compare And Swap（比较与交换） 。CAS 的思想很简单，就是用一个预期值和要更新的变量值进行比较，两值相等才会进行更新。CAS 是一个原子操作，底层依赖于一条 CPU 的原子指令。
    - ABA问题：如变量 V 初次读取的时候是 A ，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。
      - ABA 问题的解决思路是在变量前面追加上版本号或者时间戳

IO

- 同步阻塞BIO：发起read后，等os将data拷贝到用户空间
- 同步非阻塞IO：通过轮询os是否准备好数据，避免一直阻塞，耗cpu
- NIO(IO多路复用)：对应java.nio包，先调用select询问内核数据是否准备好，等准备好后用户线程再发起read调用（这个过程还是阻塞的），减少无效的系统调用，减少cpu消耗
- AIO（NIO2）：异步的，基于事件和回调机制实现，发起read后不会阻塞，性能没什么提升

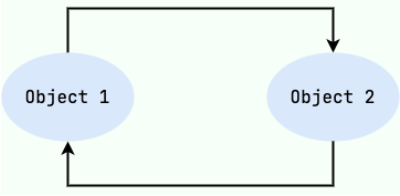
JVM

- 内存区域
  - 堆：存放对象，垃圾回收
    - 新生代
    - 老生代
    - Metaspace(元空间)
  - 线程：程序计数器、本地方法栈、虚拟机栈

- 方法区，线程共享
- 运行时常量池
- 字符串常量池
- 直接内存
- HotSpot创建对象
  - 创建
    - 类加载
    - 分配内存
    - 初始化0值
    - 设置对象头（元数据，哈希值，属于类等）
    - 执行init方法，按照程序员意愿初始化
  - 对象的内存布局
    - 对象头
    - 示例数据
    - 对齐填充
  - 对象的访问定位
    - 句柄
      - reference存放对象实例数据以及数据的地址
    - 直接指针
      - reference存放地址

- 垃圾回收
  - 内存分配和回收原则
    - 优先在新生代创建
    - 大对象（需要大量连续内存空间的对象（比如：字符串、数组））直接进入老年代
    - 长期存活的对象进入老年代
    - 主要进行gc的区域
      - 新生代收集、老年代收集
      - 整堆收集（Java堆和方法区）
  - 死亡对象判断

- 引用计数（为0，很难解决循环引用问题）



- 可达性分析方法。跟GC Roots有无路径相连
- 废弃常量：无引用
- 废弃类：1、实例都被回收；2、ClassLoader被回收；3、该类的java.lang.Class没有被引用

- 垃圾收集算法
  - 标记清除
    - 碎片，效率低
  - 标记复制算法
    - 复制到一边，效率低，内存利用率低
  - 标记整理
    - 整理到一端，适合老年代
  - 分代收集
    - 新生代：标记复制
    - 老年代：标记清除/标记整理
- 垃圾收集器gc
  - serial：串行、单线程
  - ParNew：serial的多线程版
  - Parallel Scavenage(jdk 1.8默认)
    - 重点关注吞吐量，高效利用cpu
  - serial old
  - parallel old
  - cms
    - 以获取最短回收停顿时间为目标
    - HotSpot 虚拟机第一款真正意义上的并发收集器 ,垃圾收集线程与用户线程（基本上）同时工作

        - 初始标记、并发标记（gc和用户线程同时运行）、重新标记（并发时用户线程会更新引用，重新标记产生变动的内存）、并发清除
    - g1
        - 面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器.以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征
    - zgc
- 类加载
    - 每个Java类都有一个引用指向加载他的ClassLoader
    - 加载Java类的字节码.class到JVM中
    - BootstrapClassLoader(启动类加载器)、ExtensionClassLoader(扩展类加载器)、AppClassLoader(应用程序类加载器)
        - 三个默认类加载器、遵循双亲委派模型
    - 双亲委派模型：类加载器先找父类的加载类来加载，加载不到再去自己加载，这个过程会一直向上递归，直到达到最顶层的类加载器。如何绕过双亲委派模型：自定义类加载器，继承ClassLoader类重写loadclass方法；使用线程上下文类加载器。由于加载工作最终由顶层开始，因此所有的类都会被加载一次，避免了在JVM中的多份同样的类。
- 参数调优
    - -Xms512m表示设置JVM启动时的堆内存为512MB。
      -Xmx2048m表示设置JVM最大可用堆内存为2048MB。
      -Xmn：设置年轻代（Young Generation）的大小。年轻代的大小直接影响到垃圾收集的性能，适当调整可以优化GC性能。
      -XX:NewRatio=3表示老年代与年轻代的比值为3，即老年代是年轻代的3倍大小。
      -XX:MaxMetaspaceSize（Java 8及之后）：设置元空间的最大大小，元空间用于存放类的元数据信息。
      -XX:+PrintGCDetails：打印垃圾回收的详细信息。
      -Xss256k会设置每个线程的栈大小为256KB
    - 实战：full gc问题，频繁的大对象的建立导致老年代被填满，触发full gc。增加堆内存大小、调整老年代和新生代比例。python oj前端一次1000个题目，大对象，触发full gc。日志：[Full GC (Allocation Failure)
    - GC 性能指标通常关注吞吐量、停顿时间和垃圾回收频率。GC 优化的目标就是降低 Full GC 的频率以及减少 Full GC 的执行时间

- **新特性**