主要内容：

- Java的一些基础知识
- 项目问题
- 手撕代码

# Java基础知识

### 1. 哪些存储容器是线程安全的？ （不熟）

同步容器类：使用了synchronized

- Vector
- HashTable

并发容器类：

- ConcurrentHashMap：分段
- CopyOnWriteArrayList：写时复制
- CopyOnWriteArraySet：写时复制

Queue

- ConcurrentLinkedQueue：使用非阻塞式的方式实现的基于链接节点的无界的线程安全队列，性能非常好。（java.util.concurrent.BlockingQueue 接口代表了线程安全的队列）
- ArrayBlockingQueue：基于数组的有界阻塞队列
- LinkedBlockingQueue：基于链表的有界阻塞队列
- PriorityBolckingQueue：表示优先级的无界阻塞队列，即该阻塞队列中的元素可自动排序。默认情况下，元素采用自然升序
- DelayQueue：一种延时获取元素的无界阻塞队列
- SynchronousQueue：不存储元素的阻塞队列。每个put操作必须等待一个take操作，否则不能继续添加元素；内部起始没有任何一个元素，容量是0

Deque接口定义了双向队列，双向队列允许在队列头和尾部继续入队出队操作

- ArrayDeque：基于数组的双向非阻塞队列
- LinkedBlockingDeque：基于链表的双向阻塞队列

Sorted容器

- ConcurrentSkipListMap：是TreeMap的线程安全版本
- ConcurrentSkipListSet：是TreeSet的线程安全版本

### 2. StringBuffer 和 StringBuilder 有线程安全的吗（没听清，以为是多线程的东西，能回答上来的）

- `String` 中的对象是不可变的，也就可以理解为常量，线程安全。
- `StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。
- `StringBuilder` 并没有对方法进行加同步锁，所以是非线程安全的。

### 3. 项目中使用多线程的场景（目前写过的项目没有使用过多线程！！！）

- 多线程是一种允许多个任务同时执行的技术。它通过将任务分解为更小的部分并在不同的线程上运行这些部分来实现。
- 这可以提高程序的性能，尤其是当任务是CPU密集型或涉及大量IO时
- 在工作中，有多种场景可以使用多线程，一下是一些最常见的例子：
  - 后台任务：后台任务是通常在用户不知不觉中运行的任务，例如：文件上传或数据处理，使用多线程可以将这些任务与主应用程序分开，这样他们就不会阻塞主应用程序并导致其无响应
  - 网络请求：当用户从网站或应用程序请求数据时，会发出网络请求，使用多线程可以同时处理多个网络请求，从而提高响应速度

### 4. 多线程使用过程中，需要注意的点是什么？（不熟）

- 死锁
- 线程安全
- 并发

### 5. 多线程中的各项资源这么处理呢？（不熟）

- 同步关键字：synchronized关键字是 Java 中最常用的同步机制，它通过 一次只有一个线程执行代码块或方法来保护共享资源
- 锁：java 提供了显示锁机制，例如：ReentrantLock 和 ReadWriteLock，这些锁比 synchronized 关键字更灵活，但也更复杂
- 原子操作：Java 提供了原子操作，例如 AtomicInteger 和 AtomicLong，原子操作是不可分割的操作，可以确保共享资源的更新不会出现数据竞争
- 线程安全类：Java 提供了许多线程安全类，例如 Vector 和 HashMap，这些类内部使用了同步机制来保护共享资源，因此无需显示同步

### 6. Synchronized中 类锁 和 对象锁的区别？（以前学习过，忘记了）

- 类锁和对象锁都是 Java 中的同步机制，用于控制对共享资源的访问，但是，他们之间还存在一定的区别
- 获取方式
  - 类锁是通过 synchronized 关键字 修饰静态方法或代码块来获取的
  - 对象锁是通过 synchronized 关键字修饰实力方法或代码块来获取的，也可以通过显示锁（例如 ReentrantLock）来获取
- 作用范围
  - 类锁是整个类，意味着同一时刻只能有一个线程持有类锁
  - 对象锁的作用范围是单个对象，意味着同一时刻可以有多线线程持有不同的对象锁
- 同步效果
  - 类锁用于同步对静态方法和静态变量的访问。静态方法和静态变量是和类关联的，而不是与特定对象关联的。因此，类锁可以确保同一时刻只能有一个线程执行静态方法或访问静态变量
  - 对象锁用于同步对实例方法和实例变量的访问。实例方法和实例变量是与特定对象关联的，因此，对象锁可以确保同一时刻只能有一个线程执行同一个对象的实例方法或访问同一个对象的实例变量

### 7. maven工程和java工程的区别？（用自己的语言说出来了）

- 项目结构
  - maven
    - 具有标准化的项目结构，所有项目文件都位于特定的目录中，这使得maven工程易于理解和维护
    - 使用基于 XML 的配置文件（pom.xml）来定义项目的构建过程。可以自动下载依赖项，编译代码，打包应用程序等
    - 具有强大的依赖管理功能，可以自动下载、安装和管理项目所需的依赖项
  - java
    - 可以任意定义，这可能会导致难以理解和维护
    - 通常需要使用其他的构建根据，例如 Ant 或 Gradle 来定义构建过程，这些根据可能比 maven 更复杂
    - 手动管理依赖，可能会导致依赖项冲突和其他问题
- 可移植性
  - maven工程可移植，可以在任何具有 maven安装的环境下运行
  - java工程的可移植性可能取决于所使用的构建工具和其他依赖项

### 8. 类的加载机制清楚吗？（背过，说不清楚！！！）

- 类从加载到虚拟机中开始，直到卸载为止，它的整个生命周期包括了：加载、验证、准备、解析、初始化、使用和卸载这7个阶段。其中，验证、准备和解析这三个部分统称为连接（linking）
  - 1.==加载==：查找和导入class文件
  - 2.==验证==：保证加载类的准确性
  - 3.==准备==：为类变量分配内存并设置类变量初始值
  - 4.==解析==：把类中的符号引用转换为直接引用
  - 5.==初始化==：对类的静态变量，静态代码块执行初始化操作
  - 6.==使用==：JVM 开始从入口（main）方法开始执行用户的程序代码
  - 7.==卸载==：当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存

### 9.什么是双亲委派模型？及其好处（背过，说不清楚！！！）

- 如果一个类加载器收到了类加载的请求，它首先==不会自己尝试加载这个类==，而是把这请求委派给父类加载器去完成，每一个层次的类加载器都是如此
- 因此所有的加载请求最终都应该委派到顶层的启动类加载器中，只有当父类加载器返回自己无法完成这个加载请求（它的搜索返回中没有找到所需的类）时，子类加载器才会尝试自己去加载

好处：

- 第一、通过双亲委派机制可以==避免==某一个==类被重复加载==，当父类已经加载后则无需重复加载，保证唯一性。
- 第二、为了==安全==，保证==类库API不会被修改==

### 10.什么是反射，其优缺点？应用场景（看过，说不清楚！！！）

- 通过反射你可以获取任意一个类的所有属性和方法，你还可以调用这些方法和属性。

1. 优点：反射可以让我们的代码更加灵活、为各种框架提供开箱即用的功能提供了便利;
2. 缺点：
   - 性能开销：反射操作比直接访问类、字段和方法的性能要低。这是因为反射需要在运行时进行额外的解析和检查
   - 安全隐患：反射允许代码执行一些在正常情况下不允许的操作。例如访问私有成员、修改类定义等。这可能会导致安全漏洞，例如未经授权地访问或修改数据
   - 代码可读性：会使代码更加复杂和难以理解，因为它增加了额外的抽象层
   - 编译时检查缺失：由于反射是在运行时进行的，编译器无法检查许多反射相关的错误，例如类名拼写错误、方法名错误等，只能在运行时发现这些错误，有可能会导致程序崩溃
3. 像 Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制

4. 且这些框架使用了大量的AOP（动态代理），AOP的实现也依赖反射
5. 提高代码的重用性，允许程序以通用的方式处理不同的类、字段和方法。例如，假设一个程序需要验证多个对象的属性值是否为空，使用反射，程序可以编写一个通用方法来验证任意对象的任意属性

### 11. 开发中用到了那些设计模式？（不熟）

- 设计模式（design pattern）是解决软件设计中常见问题的通用解决方案，它为开发人员提供了一套结果验证的方案，用于解决常见的软件设计问题，提高代码的可复用性、可维护性和可拓展性

根据功能和用途、设计模式可以分为以下三大类：

- 创建型模式（creational Patterns）：用于常见对象的模式
- 结构性模式（Structural Patterns）：用于描述如何将类或对象组合成更大的结构的模式
- 行为型模式（Behavioral Patterns）：用于描述对象之间如何通信和交互的模式

开发中常用的设计模式有

#### 创建型模式

- 单例模式（Singleton Pattern）：确保一个类只有一个实例，并提供一个全局访问点
- 工厂方法模式（Factory Method Pattern）：定义一个创建对象的接口，让子类决定实例化哪个类
- 抽象工厂模式（Abstract Factory Pattern）：提供一个创建多个相关或依赖对象的接口
- 建造者模式（BUilder Pattern）：将一个复杂对象的创建分为多个步骤，并用不同的对象封装这些步骤

#### 结构型模式

- 适配器模式（Adapter Pattern）：将一个类的接口转换为另一个类所需的接口
- 桥接模式（Bridge Pattern）：将一个对象的接口和实现解耦，使得二者可以独立变化
- 组合模式（Composite Pattern）：将多个对象组合成树形结构，并向客户提供统一的接口
- 装饰者模式（Decorator Pattern）：为一个对象添加新的功能，保持原有功能不变。

### 12.JVM的垃圾回收算法知道吗？

- 标签-清除算法：最早使用的垃圾回收方法之一，该方法首先标记所有可达的对象，然后回收所有未标记的对象，效率比较低，需要扫描整个堆空间，内存碎片比较大
- 复制算法：将所有可达的对象复制到一个新的内存区域，然后回收旧的内存区域。效率比较高，但需要大量的内存空间
- 标签-整理算法：结合标签-清除算法和压缩法的有点，先标记所有可达的对象，然后将这些对象移动到堆空间的一端，并回收剩余的内存空间。目前最常用的垃圾回收方法之一。

- 分代收集算法：将堆空间划分为年轻代和老年代，年轻代包含新创建的对象，老年代包含存活时间较长的大型对象。年轻代的回收频率比极高（java8使用复制算法），老年代的垃圾回收频率比较低，可以提高垃圾回收的效率

### 13. 一些算法的原理

- 动态规划算法（Dynamic Programming Algorithm）：一种通过将问题分解为子问题，并重复利用子问题的解决来解决问题的算法。动态规划算法常用于解决具有重叠子问题的问题。
- 贪心算法（Greedy Algorithm）：一种通过在每一步做出局部最优选择的算法，通常用于解决 NP 难问题
- 并查集：一种用于处理集合的数据结构，并查集支持两种操作，查找和合并，通常用于解决连通性问题
- 快排基本原理：通过分治法将一个数组划分成两个数组，然后递归地对子数组进行排序

### 14. 优化mysql的方法

- 使用分布式数据库
- 使用负载均衡
- 使用监控工具

### 15. Redis缓存击穿、穿透、雪崩（击穿和穿透区分了好久）

缓存穿透是指 缓存和数据库中都没有的数据 ，而用户不断发起请求。

1. 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对 写为 key-null，缓存有效时间可以设置短点，如30s（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击；
3. 布隆过滤器。bloomfilter就类似于一个HashSet，用于快速判断某个元素是否存在于集合中，其典型的应用场景就是快速判断一个key是否存在于某容器，不存在就直接返回。布隆过滤器的关键就在于hash算法和容器大小

缓存击穿是指 缓存中没有但数据库中有的数据 （一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没有读到数据，又同时去数据库读取数据，引起数据库压力瞬间增大，造成过大压力。

1. 热点数据支持续期，持续访问的数据可以不断续期，避免因为过期失效而被击穿
2. （互斥锁）发现缓存失效，重建缓存加互斥锁，当线程查询缓存发现缓存不存在就会尝试加锁，线程争抢锁，拿到锁的线程就会查询数据库，然后重建缓存，争抢锁失败的线程，可以加一个睡眠然后循环重试。
3. （逻辑过期）把过期时间设置在value中，查询缓存中的数据时，如果发现逻辑时间已过期，尝试获取锁，开启一个异步的方法进行缓存的更新操作，先返回过期的数据。

缓存雪崩，是指大量的应用请求因为异常无法在Redis缓存中进行处理，像雪崩一样，直接打到数据库。
缓存中 数据大批量过期，而查询数据量巨大，引起数据库压力过大甚至宕机

1. 缓存数据的过期时间设置随机，分支同一时间大量数据过期现象发生
2. 重建缓存加互斥锁，当线程拿到缓存发现缓存不存在就会尝试加锁，线程争抢锁，拿到锁的线程就会进行查询数据库，然后重建缓存，争抢锁失败的线程，可以加一个睡眠然后循环重试

**16. Vue常见的钩子函数，以及作用？** `(就说了onMounted和onUnmounted)``

- onMounted()　　注册一个回调函数，在组件挂载完成后执行。
- onUpdated()　　注册一个回调函数，在组件因为响应式状态变更而更新其 DOM 树之后调用。
- onUnmounted()　　注册一个回调函数，在组件实例被卸载之后调用。
- onBeforeMount()　　注册一个钩子，在组件被挂载之前被调用。
- onBeforeUpdate()　　注册一个钩子，在组件即将因为响应式状态变更而更新其 DOM 树之前调用。
- onBeforeUnmount() 注册一个钩子，在组件实例被卸载之前调用。

## 项目拷打（伙伴匹配和API项目）

**伙伴匹配项目的规模有多大，有几个人开发，都分别是什么角色**

- 按照开发的流程，说自己之前的公司比较小，就只有 前端、后端、运维这些人，一共十几个人

**这两个项目你做了那些优化呢？遇到的难题有吗？**

- 这里乱答的，背了一些相关的面试题
- 难题因为没做这两个项目，就没怎么回答上来
- 优化就说了这些
  - 伙伴匹配
    - 使用knife4j+swagger，优化接口调试
    - 使用Stream API + Lambda，简化集合处理
    - 自主编写Dockerfile，实现自动化镜像构建及容器部署
  - API项目
    - 使用Spring Cloud Gateway 作为API网关
    - 使用Spring Boot Starter 开发客户端SDK
    - balabala

## 手撕算法

因为说了自己只对数据结构相关的算法比较熟悉，就出了一个二叉树的题目

104. 二叉树的最大深度 - 力扣（LeetCode）

**搞清楚什么是深度和高度**

**深度**：任意一个节点到root节点的距离
**高度**：任意一个节点到叶子节点的距离

**后序遍历（左右中）**：适用于求树的高度，因为通过左右节点，可以将结果返回给当前的父节点，实现了从底部往上的一个计数过程

**前序遍历（中左右）**：适用于求树的深度，往下遍历一次，就深度+1，从而不断遍历，向下探索

- 明面上来说是求最大深度，==本质上是求根节点的高度==，所以我们使用==后序遍历==
- 递归三部曲：
  - 返回值int，入参节点
  - 终止条件，入参的节点为null
  - 遍历顺序，左右中，左右为递归，==中为取左右中的最大高度+1==

```
class Solution {
    public int maxDepth(TreeNode root) {
        if(root==null) return 0;
        int leftHeight = maxDepth(root.left);
        int rightHeight = maxDepth(root.right);
        return Math.max(leftHeight,rightHeight) + 1;
    }
}
```