面试真题

面向简历

熟悉Java集合

Java 中的集合类主要分为两大类：

Collection 接口：存储对象的集合类

Collection 接口下又分为 List、Set 和 Queue 接口。

Map 接口：存储的是键值对（key-value）。

Collection 接口

**List 接口：**

* ArrayList：基于动态数组，查询速度快，插入、删除慢。
* LinkedList：基于双向链表，插入、删除快，查询速度慢。
* Vector：线程安全的动态数组，类似于 ArrayList，但开销较大。

**Set 接口：**

* HashSet：基于哈希表，元素无序，不允许重复。
* LinkedHashSet：基于链表和哈希表，维护插入顺序，不允许重复。
* TreeSet：基于红黑树，元素有序，不允许重复。

所以网上有些说 Set 是无序集合非常不准确，因为需要看具体的实现类。

**Queue 接口：**

* PriorityQueue：基于优先级堆，元素按照自然顺序或指定比较器排序。
* LinkedList：可以作为队列使用，支持 FIFO（先进先出）操作。

Map 接口

* HashMap：基于哈希表，键值对无序，不允许键重复。
* LinkedHashMap：基于链表和哈希表，维护插入顺序，不允许键重复。
* TreeMap：基于红黑树，键值对有序，不允许键重复。
* Hashtable：线程安全的哈希表，不允许键或值为 null。
* ConcurrentHashMap：线程安全的哈希表，适合高并发环境，不允许键或值为 null。

I/O流

字节流和字符流的区别

**字节流**：

* 继承自InputStream和OutputStream，以字节为单位进行数据的读写。
* 适用于所有类型的数据，包括文本、图片、音频等。
* FileInputStream、FileOutputStream、BufferedInputStream、BufferedOutputStream。

**字符流**：

* 继承自Reader和Writer，以字符为单位进行数据读写。
* 自动处理字符编解码，适用于纯文本文件。
* FileReader、FileWriter、BufferedReader、BufferedWriter。

缓冲流（Buffered Streams）：

BufferedXXX

通过内存缓冲区来提升I/O效率，提高读写性能，减少系统调用次数。

对象流（Object Streams）：

ObjectXXX

允许Java对象在网络中传输或保存在文件中（序列化）。

**注意**：对象流只能处理实现了Serializable接口的对象。

I/O异常处理

使用try-catch-finally块进行异常处理。为了简化资源管理，Java 7引入了**try-with-resources**语句，自动关闭流，避免资源泄漏。在I/O操作完成后，需要显式关闭流，推荐使用try-with-resources来自动关闭。

NIO（New I/O）与传统I/O的区别

**NIO**提供了更高效的I/O操作，主要通过缓冲区和通道进行数据传输。支持异步I/O和非阻塞I/O，能够更好地处理大规模并发请求，适用于高性能应用。

反射机制

Java反射机制定义

* 反射允许在运行时查询和操作类的信息（如类名、方法、字段等）。
* 通过反射可以动态加载类、调用方法、访问字段，甚至修改类的属性或方法。

主要作用

* **动态加载类**：在运行时根据类名动态加载类。
* **访问类的成员**：通过反射获取类的字段、方法、构造器等信息。
* **动态代理**：反射可以与Java的动态代理结合使用，生成运行时代理对象。
* **框架和库**：许多框架（如Spring、Hibernate）通过反射机制进行依赖注入、对象映射等操作。

反射机制的核心类

* **Class类：提供了访问类的基本信息（如类名、父类、接口、方法、构造器等）。**
  + **通过Class.forName("className")来动态加载类。**
  + **通过getDeclaredMethods()、getDeclaredFields()等方法获取类的成员。**
* **Method类：代表类的方法，可以通过反射动态调用方法。**
* **Field类：代表类的字段，可以通过反射访问或修改字段的值。**
* **Constructor类：代表类的构造函数，可以用来动态创建对象实例。**

反射常用操作：

* **加载类：**

**java**

**复制代码**

**Class<?> clazz = Class.forName("com.example.MyClass");**

* + **Class.forName("className")可以根据类名动态加载类。**
* **创建对象：**

**java**

**复制代码**

**Constructor<?> constructor = clazz.getConstructor(String.class, int.class);**

**Object obj = constructor.newInstance("John", 25);**

* + **通过getConstructor获取构造器，使用newInstance创建对象。**
* **访问字段：**

**java**

**复制代码**

**Field field = clazz.getDeclaredField("name");**

**field.setAccessible(true);**

**String value = (String) field.get(obj);**

* + **通过getDeclaredField获取字段，setAccessible(true)可以访问私有字段。**
* **调用方法：**

**java**

**复制代码**

**Method method = clazz.getDeclaredMethod("setName", String.class);**

**method.invoke(obj, "NewName");**

* + **通过getDeclaredMethod获取方法，使用invoke动态调用方法。**

反射的优缺点：

* **优点**：
  + 提供了灵活性，能够在运行时动态调用类的成员。
  + 常用于框架、库的实现（如依赖注入、对象映射等）。
* **缺点**：
  + 性能开销较大：反射涉及动态类型检查，比静态方法调用慢。
  + 安全问题：可以访问和修改私有字段和方法，可能带来安全隐患。
  + 代码可读性差：过度使用反射可能导致代码难以理解和维护。

应用场景：

* **依赖注入（DI）**：通过反射动态注入对象，Spring框架使用反射机制来管理Bean的生命周期和依赖关系。
* **动态代理**：Java的动态代理机制使用反射来生成代理类，并在运行时为接口方法提供代理逻辑。
* **ORM框架**：如Hibernate，使用反射来自动映射数据库字段和Java对象属性。

HashMap源码及其扩容机制

**负载因子：**HashMap 中的扩容是基于负载因子来决定的，默认为 0.75，这意味着当 HashMap 的已存储元素数量超过当前容量的 75% 时，就会触发扩容操作。

例如，初始容量为 16，负载因子为 0.75，则扩容阈值为 16 × 0.75 = 12。当存入第 13 个元素时，HashMap 就会触发扩容。

**Rehashing：**当触发扩容时，HashMap 的容量会扩大为当前容量的**两倍**。例如，容量从 16 增加到 32。扩容时，HashMap 需要重新计算所有元素的哈希值，并将它们重新分配到新的哈希桶中，这个过程称为**rehashing**。每个元素的存储位置会根据新容量的大小重新计算哈希值，并移动到新的数组中。

**Rehashing 细节：**

**数组长度和扩容机制：**按照我们的思维，每一个元素应该是重新 hash 一个一个搬迁过去。在 1.7 的时候就是这样实现的，然而 1.8 在这里做了优化，Java 8 之后的扩容不需要每个节点重新 hash 算下标，因为元素的新位置只与高位有关，通过和老数组长度的 & 计算是否为 0 就能判断新下标的位置，因此链表中的元素可能只需要部分移动。这一优化减少了扩容时的计算开销。

关键点就在于数组的长度是 2 的次方，且扩容为 2 倍。

**利用hash计算位置：**因为数组的长度是 2 的 n 次方，所以假设以前的数组长度（16）二进制表示是 010000，那么新数组的长度（32）二进制表示是 100000，，而我们通过 key 的 hash 值定位其在数组位置所采用的方法是 (数组长度-1) & hash。我们还是拿 16 和 32 长度来举例：

16-1=15，二进制为 001111

32-1=31，二进制为 011111

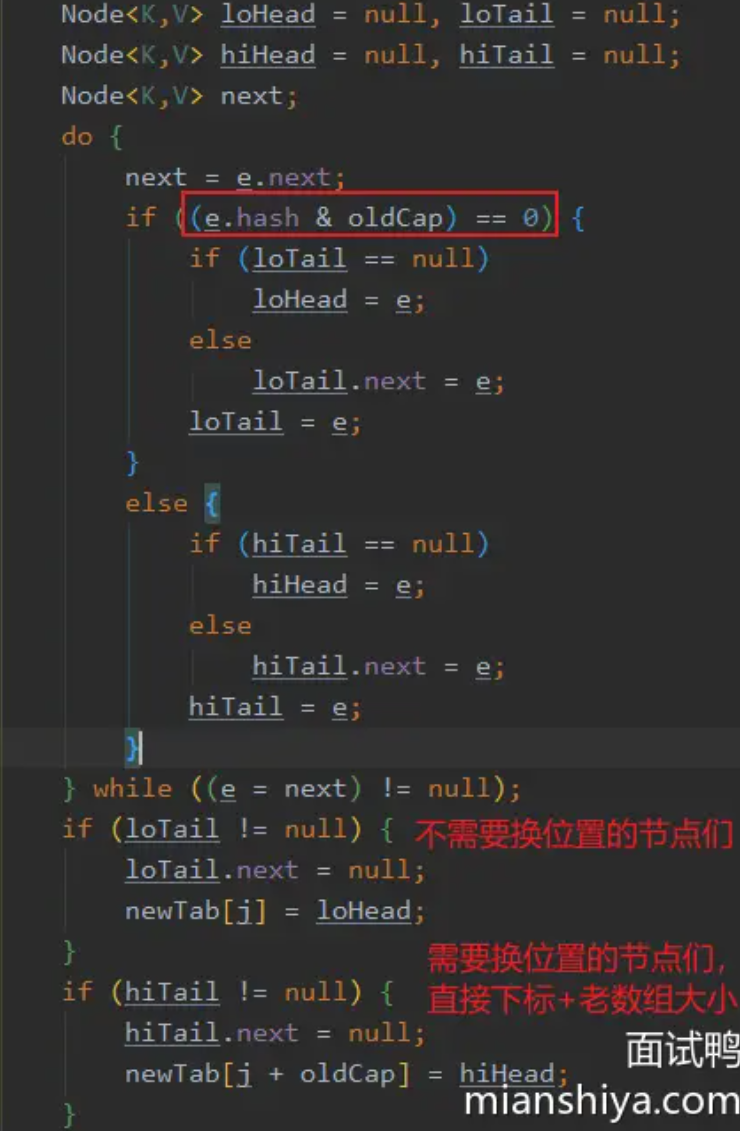
所以重点就在 key 的 hash 值的从右往左数第五位是否是 1：

* 如果是 1 说明需要搬迁到新位置，且新位置的下标就是原下标+16（原数组大小），
* 如果是 0 说明吃不到新数组长度的高位，那就还是在原位置，不需要迁移。

所以，我们刚好拿老数组的长度（010000）来判断高位是否是 1，这里只有两种情况，要么是 1 要么是 0 。

通俗理解：

1. 每次扩容，新数组是老数组的2倍，所以新数组可以看成是老空间+一块一样大小的新空间
2. 扩容时，下标与老数组长度进行&运算，本质上是判断当前下标是在老数组的高位还是低位。
3. 所谓高位、低位，是指该下标在数组的左半边还是右半边
4. 如果下标在老数组的低位，那么直接平移到**新数组中老空间**的那半边
5. 如果下标在老数组的高位，那么直接平移到**新数组中新空间**的那半边，具体位置就是原位置 + 老数组的长度



知识框架

Java集合