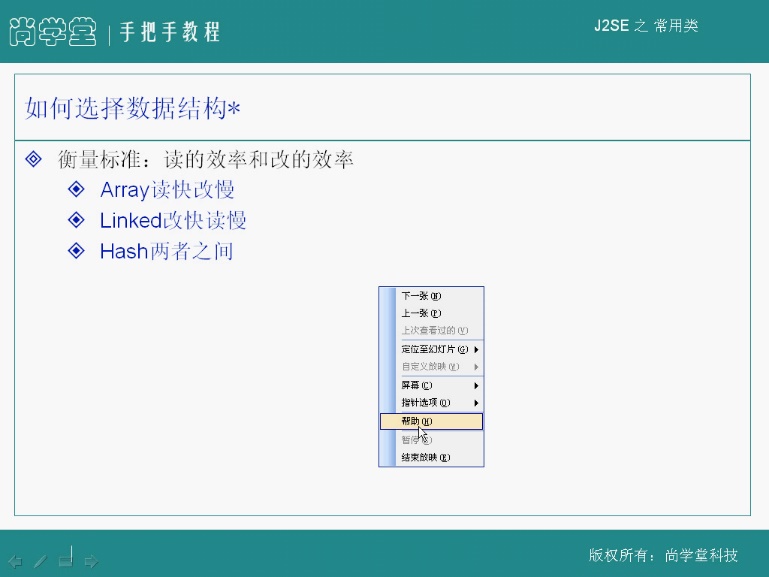
**09\_Map\_1\_笔记**

**一、如何选择数据结构**

**1. 初学者的选择建议 ﻿**

* 灵活替换原则: 使用父类引用指向子类对象（如Collection c = new ArrayList()），后期可轻松替换实现类（如换成LinkedList）
* 初期选择建议: 初学者可任意选择，重点掌握标准写法，实际应用中再根据需求调整
* 

**2. 选择数据结构的衡量标准**

* 核心指标: 读操作效率（检索速度）和改操作效率（增删速度）的平衡
* 类比说明: 如同选择容器装馒头，需根据实际使用场景（装什么/怎么访问）决定用篮子、桶或塑料袋
* 决策要点: 没有放之四海皆准的标准，需结合实际应用需求（类似杨贵妃与赵飞燕的美丑标准取决于个人偏好）

**3. ArrayList与LinkedList的比较**

* ArrayList特性:
  + 读快: 基于数组实现，通过偏移量直接定位（如访问第5个元素：5×4字节=20字节偏移）
  + 改慢: 删除中间元素需重建数组，涉及大量数据拷贝
* LinkedList特性:
  + 改快: 基于链表实现，增删只需调整指针指向（如删除元素只需断开前后链接）
  + 读慢: 访问第N个元素需从头遍历N次（访问第5万个元素需循环5万次）
* 选择策略: 读多选ArrayList，改多选LinkedList

**4. HashTable与Vector的简介及不推荐原因**

* 遗留问题:
  + 效率低下: 内部采用同步锁机制，读写操作都受限制
  + 替代方案: 现代开发推荐使用Set/List/Map系列集合类
* Hash折中方案:
  + 特点: 读写效率介于ArrayList和LinkedList之间
  + 代表类: HashSet/HashMap等哈希实现类

**5. 总结：数据结构选择依据**

* 速记口诀:
  + Array开头：读快改慢
  + Link开头：改快读慢
  + Hash开头：折中方案
* 终极原则: 根据实际应用场景需求（读/写频率）做出选择

**二、Map接口**

**1. Map接口的定义**

* 核心概念: 存储键值对（key-value pairs）的容器类
* 实现类对比:
  + HashMap: 基于哈希表实现，适合快速查找
  + TreeMap: 基于红黑树（平衡二叉树）实现，保持元素有序
* 键的特性:
  + 唯一性: 键不能重复（通过equals判断，实际比较hashCode提高效率）
  + 重写规则: 重写equals()必须同时重写hashCode()（因Map依赖hashCode比较键）



**2. Map接口的方法 ﻿**

* 核心方法:
  + put(Object key, Object value): 添加键值对，返回被替换的旧value
  + get(Object key): 通过键获取对应value
  + remove(Object key): 删除指定键的映射
  + containsKey/Value(): 检查包含特定键/值
  + size()/isEmpty(): 获取元素对数/判空
  + putAll(Map t): 合并另一个Map的所有映射
  + clear(): 清空所有映射

**3. 应用案例**

**1）例题:Map接口方法使用**



* 关键操作解析:
  + 类型转换: 获取value后需强制转换（如(Integer)ml.get("two")）
  + 值提取: 包装类转基本类型需调用intValue()等方法
  + 合并映射: putAll()会将两个Map的键值对合并（重复键会被覆盖）
  + 包含检查: containsValue()通过equals比较值对象

**三、知识小结**

|  |  |  |  |
| --- | --- | --- | --- |
| 知识点 | 核心内容 | 考试重点/易混淆点 | 难度系数 |
| 数据结构选择原则 | 初学者可灵活选择数据结构，通过父类引用指向子类对象实现灵活替换（如List接口下ArrayList与LinkedList的切换） | 接口与实现类的关系：Collection c = new ArrayList()的扩展性 | ⭐⭐ |
| ArrayList vs LinkedList | ArrayList：读快改慢（数组实现，通过偏移量直接定位）；LinkedList：改快读慢（链表实现，需遍历节点） | 性能对比：高频读取选ArrayList，高频修改选LinkedList | ⭐⭐⭐ |
| Map接口基础 | 存储键值对（Key-Value），键不可重复（基于equals和hashCode比较） | 重写equals必须重写hashCode：Map通过hashCode快速比较键的唯一性 | ⭐⭐⭐⭐ |
| HashMap vs TreeMap | HashMap：哈希表实现，无序；TreeMap：红黑树（二叉树）实现，有序 | 索引效率：HashMap适合哈希索引，TreeMap适合范围查询 | ⭐⭐⭐ |
| Map操作方法 | put(key,value)：替换重复键的旧值并返回；get(key)：通过键检索值；remove(key)：删除键值对 | 返回值注意：put返回被替换的旧值 | ⭐⭐⭐ |
| 实际应用示例 | 员工信息存储需建立索引（类似字典），避免全表扫描 | 索引设计：键的选取直接影响查询效率（如员工ID作为Key） | ⭐⭐⭐⭐ |

该部分主要讲述了Java中Map数据结构的工作原理和键值对的存储方式。Map使用hash code快速定位键值对，当键存在时，会替换掉原来的值。通过get方法可以获取键对应的值，而containsKey方法则用于检查Map中是否包含某个键。此外，Map还有一个size方法，可以返回存储的键值对数量。还提到了String类型键的特殊性，以及如何使用equals方法比较两个对象是否相等。

分段总结

数据结构选择

1.初学者可以选择任意一种数据结构开始学习，如ArrayList或LinkedList。

2.选择数据结构时，应考虑实际应用程序的需求，选择最适合的数据结构。

3.数据结构的选择没有绝对的标准，需根据具体应用场景进行评估。

衡量标准

1.衡量数据结构优劣的标准主要包括读取效率和修改效率。

2.读取效率高适合读操作频繁的应用，修改效率高适合写操作频繁的应用。

3.例如，ArrayList适合读操作频繁的应用，LinkedList适合写操作频繁的应用。

ArrayList与LinkedList的比较

1.ArrayList基于数组实现，读取效率高但修改效率低。

2.LinkedList基于链表实现，读取效率低但修改效率高。

3.选择数据结构时应根据实际需求平衡读取和修改的效率。

Hashtable和Vector

1.Hashtable和Vector是较古老的数据结构实现，内部锁定导致效率较低。

2.现在更推荐使用Set、List或Map等数据结构。

3.选择数据结构时应考虑其读取和修改的效率。

Map接口

1.Map接口用于存储键值对，键唯一且不能重复。

2.Map接口的实现类包括HashMap和TreeMap，分别使用哈希表和二叉树进行索引。

3.键值对的存储和检索通过键来进行，键不能重复。

Map接口的方法

1.Map接口定义了put(key, value)、get(key)、remove(key)、containsKey(key)、containsValue(value)、size()、isEmpty()、clear()等方法。

2.put方法用于向Map中添加或更新键值对，返回旧值。

3.get方法通过键获取对应的值。

4.remove方法通过键删除键值对。

5.containsKey和containsValue方法用于检查Map中是否包含特定的键或值。

6.size方法返回Map中键值对的数量。

7.isEmpty方法检查Map是否为空。

8.clear方法清除Map中的所有键值对。

Map接口的示例代码

1.示例代码展示了如何创建HashMap和TreeMap对象，并向其中添加、获取、删除键值对。

2.示例代码还演示了如何检查Map中是否包含特定的键或值，以及获取Map的大小和清除所有键值对。

重点

Arraylist与Linkedlist的区别



数据结构的选择原则

A screenshot of a computer screen

AI-generated content may be incorrect.

Map接口的作用

A screenshot of a computer screen

AI-generated content may be incorrect.