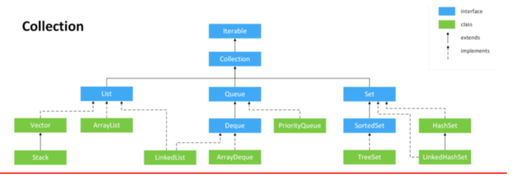
- Java容器
  - 集合概述
    - 容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表。
    - Collection
      - 
    - Map
      - 
    - 设计模式
      - 迭代器模式 — Collection 继承了 Iterable 接口，其中的 iterator() 方法能够产生一个 Iterator 对象，通过这个对象就可以迭代遍历 Collection 中的元素
      - 适配器模式 — java.util.Arrays#asList() 可以把数组类型转换为 List 类型。
    - List、Queue、Set、Map区别
      - List：存储的元素有序且可重复
      - Queue： 按特定的排队规则来确定先后顺序，存储的元素是有序且可重复
      - Set：存储的元素是无序且不可重复
      - Map：使用键值对（key-value）存储，key是无序且不可重复的，value是无序且可重复的
  - Collection
    - List
      - ArrayList
        - 底层：Object[] 数组
        - ArrayList不同步，即不保证线程安全
        - 扩容
          - ArrayList可以动态扩容，默认大小为10，数组在初始化的时候必须指定大小
          - 添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容为原来的1.5倍，oldCapacity + (oldCapacity >> 1)
          - 扩容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建 ArrayList 指定大概的容量大小，减少扩容操作的次数。
        - ArrayList基于数组实现，所以支持随机访问，但插入删除的代价很高，需要移动大量元素
        - ArrayList适用于频繁的查找工作，空间浪费主要体现在list列表结尾会预留一定空间
      - LinkedList
        - 底层：双向链表(jdk1.6之前是循环链表， jdk1.7取消了循环)，Node存储链表节点信息
        - LinkedList不同步，即不保证线程安全
        - LinkedList基于链表实现，所以不支持随机访问，但插入删除只需要改变指针。
        - LinkedList的空间浪费体现在它的每一个元素都需要消耗比ArrayList更多的空间
      - Vector
        - 底层：Object[] 数组
        - Vector的实现与ArrayList类似，但是使用了synchronized进行同步，线程安全
        - 同步会在一个记录上加锁，防止多个程序访问同一条数据导致数据不同步，这样会导致访问速度变慢。
        - Vector扩容请求其大小的2倍，ArrayList是1.5倍
      - ArrayList线程安全方案
        - 可以使用 Collections.synchronizedList()对ArrayList对象进行包装得到一个线程安全的 ArrayList
        - CopyOnWriteArrayList 写时复制
          - 底层通过复制数组的方式实现
          - 读写分离
            - 写操作在复制的数组上进行，读操作还是在原始数组中进行，写操作加锁，防止并发写入时导致写入数据丢失
            - 写操作结束之后需要把原始数组指向新的复制数组
          - 缺陷
            - 内存占用
            - 实时数据不一致：读操作不能读取实时性的数据，部分写操作的数据还未同步到读数组
    - Queue
      - Queue接口
        - Queue单端队列，遵循FIFO，只能一端插入一端删除
        - 插入队尾offer() 删除队头poll() 查询队头peek()
      - Deque接口
        - 双端队列，队列两端均可以插入和删除元素，扩展了Queue接口增加队头和队尾的插入删除方法
        - offerFirst() offerLast() pollFirst() pollLast() peekFirst() peekLast()
        - 可用于模拟栈，push() pop()方法
        - 可用ArrayDeque和LinkedList实现
          - ArrayDeque基于可变长度的数组和双指针实现
          - LinkedList基于双向链表实现
      - ArrayDeque
        - Object[]数组+双指针
        - 在JDK1.6引入，不支持存储NULL数据
        - 在插入时可能存在扩容过程，但平均的插入操作为O(1)
        - LinkedList每次插入数据时需要申请新的堆空间，性能更慢，ArrayDeque的性能更好
      - PriorityQueue
        - Object[]数组实现二叉堆
        - JDK1.5引入，与Queue的区别在于优先级高的元素出队
        - 利用了二叉堆的数据结构来实现的，底层使用可变长的数组来存储数据
        - 通过堆元素的上浮和下沉，实现了在 O(logn) 的时间复杂度内插入元素和删除堆顶元素。
        - 默认是小顶堆，但可以接收一个 Comparator 作为构造参数，从而来自定义元素优先级的先后。
    - Set[元素唯一]
      - HashSet
        - 底层：基于HashMap实现 ，线程不安全
        - HashSet 使用成员对象来计算 hashcode 值，对于两个对象来说 hashcode 可能相同，所以equals()方法用来判断对象的相等性
        - 用于不需要保证元素插入和取出顺序的场景
      - LinkedHashSet
        - 底层：基于LinkedHashMap实现，线程不安全
        - 用于保证元素的插入和取出顺序满足 FIFO 的场景
      - TreeSet
        - 底层：红黑树（自平衡的排序二叉树），线程不安全
        - 元素是有序的，排序的方式有自然排序和定制排序
        - 用于支持对元素自定义排序规则的场景。
  - Map
    - HashMap
      - 底层：数组+链表(jdk1.7)/红黑树(jdk1.8)，线程不安全
      - 线程不安全
        - JDK7
          - JDK1.7中的 HashMap扩容使用头插法将元素迁移到新的数组，多线程的环境下，有可能导致环形链表的出现，形成死循环。
        - JDK7 JDK8
          - 多线程同时执行 put 操作，如果计算出来的索引位置是相同的，那会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。
          - 线程1执行put时，因为元素个数超出threshold而导致rehash，线程2此时执行get，有可能导致get为null
      - 默认设置
        - HashMap 可以存储 null 的 key 和 value，但 null 作为键只能有一个，null 作为值可以有多个
        - 位运算
          - HashMap的大小只能是2的次幂，只有这样才能用位运算替代取模
          - 取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（hash%length==hash&(length-1)，能够提高运算效率
        - 默认HashMap大小为16，负载因子为0.75，最多只能放16*0.75=12个元素，超过则需要动态扩容2倍
      - 插入和冲突解决
        - 使用key计算hash值，再通过hash &（len-1）判断元素存放位置（(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);）
        - 如果冲突，则判断要插入的元素和已存在的元素的hashCode值和key值是否相同，如果相同则覆盖， 如果不同，使用拉链法解决冲突
        - 拉链法：遇到哈希冲突，将冲突的值头插入链表
        - 插入之后判断是否需要扩容还是转换为红黑树（jdk1.8），数组长度小于64数组扩容，否则转为红黑树
        - 扩容操作同样需要把 oldTable 的所有键值对重新插入 newTable 中
        - 数组扩容
          - 在进行扩容时，需要把键值对重新计算下标
      - 红黑树
        - HashMap中的使用
          - JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。
          - 将链表转换成红黑树前会判断，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。
          - 红黑树大小为6时退化为链表
        - 由来
          - 二叉排序树BST在插入有序数据时会退化（链表），查找的时间复杂度就提升，所以提出了平衡二叉树将查找的时间控制在logn
          - 但是平衡二叉树要求节点的左右子树高度差不大于1，导致插入删除之后需要左旋或者右旋来调整
          - 在插入、删除很频繁的场景中，平衡树需要频繁调整，这会使平衡二叉树的性能大打折扣
        - 通过五个特性来实现自平衡
          - 结点是红色或黑色
          - 根结点是黑色。
          - 每个叶子结点都是黑色的空结点（NIL结点）。
          - 每个红色结点的两个子结点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色结点)
          - 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。
    - ConcurrentHashMap
      - ConcurrentHashMap 底层采用 分段的数组+链表 (JDK1.7)，数组+链表/红黑二叉树(JDK1.8)，线程安全
      - 线程安全方式
        - JDK1.7
          - Segment数组+HashEntry数组+链表
          - 分段锁(Segment)：对整个桶数组进行了分割分段，每一把锁只锁容器其中的一部分数据，继承自ReentrantLock
          - HashEntry采用volatile为了保证HashEntry的有序性和可见性
          - 多线程访问容器内不同段的数据，就不存在锁竞争，提高并发访问率
        - JDK1.8
          - Node数组+链表/红黑树，通过在元素的节点上采用synchronized+CAS来实现线程安全的
          - Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode
      - CAS
        - Compare and Swap比较并替换来实现原子操作
        - 三个操作数：内存地址V、旧的预期值A、新值B
        - 当且仅当V=A时，使用新值B更新V的值，否则自旋CAS操作
    - LinkedHashMap
      - 底层：LinkedHashMap继承自HashMap，在数组+链表/红黑树+双向链表，线程不安全
      - 内部维护了一个双向链表，用来维护插入顺序或者 LRU 顺序。
      - accessOrder 决定了顺序，默认为 false，此时维护的是插入顺序。
      - 当一个节点被访问时，如果 accessOrder 为 true，则会将该节点移到链表尾部，此时维护的是LRU顺序。
    - HashTable
      - 底层：数组+链表，线程安全，内部方法使用synchronized修饰
      - Hashtable 不允许有 null 键和 null 值，否则会抛出 NullPointerException
      - 创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。
    - TreeMap
      - 底层：红黑树（自平衡的排序二叉树）
      - 实现 NavigableMap 接口让 TreeMap 有了对集合内元素的搜索的能力。
      - 实现SortedMap接口让 TreeMap 有了对集合中的元素根据键排序的能力。