**顺序表**

顺序表是基于数组实现的，比如ArrayList，因为顺序，所以要占用一块连续的内存空间，因为不连续就不顺序了，因为要占用连续的内存空间，所以是比较egg pain的。

首先，**它不费内存**，因为只需要存储元素值就行了。但是，它要连续的内存空间，就像住酒店似的，我有很多空房，但是不挨着，你非要一串挨着的，我得跟其他客人说说，让人家腾出来，给你弄几间挨着的，体现在代码中就是: 可能会频繁触发GC，因为没有连续的内存空间，需要将内存重新整理一下，给你腾出来连续的内存空间。所以它的这个特点可以概括为为: **空间占用小，但是要连续**。

第二，它是**支持随机访问的**，因为是连续的内存空间，所以我只要按着序号去找就行了，比如它住了酒店二楼，从1号房开始住到10号房，我要去找第9个客人，直接去敲9号门就行了，一步到位。但是，它的**删除效率极低**，比如，现在5号房间要退房，让5号走就行，但是，它明确了要连续的内存空间，所以5号房间如果空出来，就不连续了，还得让6号去5号，7号去6号...以此类推，有人说:直接让最后一个人(10号)搬到5号不就行了，不行！ 因为这样就破坏了顺序(10号本来在9号后面，现在跑到了前面)，肯定是不行的，所以只能让后面的依次向前挪，同理，如果有个人要来住到5号房间，那又要依次向后挪。它的这两个特点可以概括为: **善于访问，不善于插入删除**

所以，可以总结为: **顺序表是有序的，不耗内存但需要占用连续的内存空间，利于随机访问，不利于插入删除**

顺序表在开发中的用途可太多了，比如ArrayList。

这里其实可以简单的统筹一下ArrayList的使用地方: **凡是不涉及删除插入的都可以用这个**，比如拉服务器数据展示，检索数据库数据展示等。

但是有一点要注意，List存储Integer的时候，调用remove()方法，一定要注意参数是Object还是index:

// 这两个方法很相似，元素是Integer时候，remove(10)默认是remove(index)的

E remove(int index);

boolean remove(Object o);

这是一个坑点，因为大部分人都不看返回值的，想调用remove(Object)的，可以用包装类型转换一下即可。

#### 链表

链表的核心是链，比如最简单的单链表，每个节点除了存储元素值，还有个引用，执向它后面的邻接点，这个引用就是链表的核心，链表是比较费内存的，因为多存了个引用嘛。

// 链表的节点

class Node {

// 存储元素值

Object value;

// 存储它后面的邻接点

Node next;

}

但是，它对内存的要求不高，不需要连续的内存空间，比如，还是住酒店，它们不需要挨着的房间，比如：A说，我住这间，然后它持有B的微信，B说，我住这间，它持有C的微信，以此类推....这样，房间也不用挨着了，自己住自己的就行，而且，如果B走了，C也不用跑到B的房间住，只需要让B把C的联系方式给A，就行了，这样A和C就连接起来了，很方便。但是它有个缺点: 不支持随机访问。比如，我要找它们中的D，我不知道D在哪，我得先去找A，然后通过它的微信找到B，然后再通过B的微信找到C，再通过C的微信找到D..，

所以，它的这些特点可以概括为: **链表是比较耗内存，但是不需要连续的内存空间，利于插入删除，不利于随机访问**

我们假设有链表:A -> B -> C -> D，我们想将B删除，有人说，直接让A->C就行，不行！因为A只能找到B，不能直接找到C，所以我们需要: **让A手里的微信变为B手里的微信**即可。那么就是A.next = B.next即可。

同理，如果我们要在A和B之间插入E呢，也就是要变成: A -> E -> B -> C -> D。我们知道，这样需要 让E有B的微信，A有E的微信 即可，那么，我们可以先让A手里微信变为E，再让E手里的微信变为B。等等，有问题，如果先让A手里的微信变为E，那么A就没有B的微信了，E去哪找B的微信呢，E直接去找B要吗？不行！因为找B首先要去找A要B的微信，但是现在A也没有B的微信了。 所以我们应该: **先让A把B的微信给E，然后再让A手里的微信变为E**，因为E就在这站着呢，A可以直接找到E去要它的微信。也就是:

E.next = A.next; // A先把B的微信给E

A.next = E; // A在把手里的微信变为E

所以我们处理链表有个窍门: **1 先把想要的结果列出来，2 然后将 前面的改变 会引起后面改变 的操作 放到后面**，比如刚刚的例子:

* 先把结果列出来:A.next = E; E.next = A.next;
* 然后排序:

// 顺序1

A.next = E; // A.next改变了

E.next = A.next; // 这里要用A.next，但是前面已经修改了A.next了，引起了这里的改变，不行。那就需要将前面的改变放到后面

那么，我们就需要调整顺序了:

E.next = A.next; // 修改了E.next

A.next = E; // 修改了A.next，这里没有用到E.next，前面改了也没问题

所以，链表类的算法，诀窍就是: **1 列结果 2 调整顺序，将引起后面改变 的操作 放到后面**，当然，某些情况下可能需要引入临时变量，这里不再深入讨论。

那么，链表在哪些开发场景中适用呢，就是: 插入和删除频繁的地方。比如: 一个直播间的房间成员列表，成员频繁进入退出，退出的时候我就要从列表删除这个人。笼统的说: **凡是涉及频繁插入删除元素的地方，都应该用链表**。

综上，我们知道顺序表和链表都是有序的，它们都是按添加元素的顺序有序。其中，**顺序表访问效率高，插入删除效率低，不耗费内存但是需要连续的内存空间；链表访问效率低，插入删除效率高，比较耗内存但是不需要连续的内存空间**。

映射表

映射表的家族比较大，可以细分为很多，但是常用的就几种:

* HashMap: 最基础的映射表，基于链表数组和红黑树实现的，无序的，理想情况下存取效率为O(1)，但是比较费内存。
* TreeMap: 基于红黑树实现的，按照key有序的，因为添加/删除元素涉及到树的旋转操作，所以时间复杂度是log(n)的。
* LinkedHashMap: 跟HashMap的实现原理是一样的，不过内部多了维护元素顺序的链表，是按照元素添加的顺序有序的，而且可以指定是否自动调整最新的元素到头部，可以用来实现LRUCache。
* ConcurrentHashMap: 基于分段锁实现的，支持一定粒度的并发存取，因为内部的实现原理是CAS，所以效率比一般的Synchronized系列高。

**Android中的API**:

 ArrayMap: 基于完全二叉树实现的，跟HashMap的功能基本一样，存取效率比HashMap低，为log(n)，但是很省内存，内部只有两个数组，一个存hash，一个存key和value。

 SparseArray: 跟HashMap类似，但是Key只能是Integer，存取效率为log(n)，很省内存，内部只有两个数组，一个为key，一个为value。

我们可以看到，Android中提供的API，大都是以省内存为目的的，这在内存宝贵的移动端，确实是很明智的。 所以，在Android开发中，如果你的key是Integer，建议优先使用SparseArray；否则建议使用ArrayMap。当然，如果你对性能要求极高，对内存要求不高，则可以使用HashMap，如果你需要按照key有序，则可以使用TreeMap，如果是按照存放顺序有序，则可以使用LinkedHashMap，如果是并发场景，则可以使用ConcurrentHashMap。

比方说，我要维护一个好友列表，并且能够根据id找到好友，那么就需要一个id-好友这样的映射，如果id是Integer类型的，我们可以使用SparseArray，如果id是String类型的，我们可以使用ArrayMap。

再比如，一个音乐App，用户有个"喜欢列表"，能够按照用户收藏的顺序添加进去，同时可以根据歌曲名字搜索，那么这里有两个关键点: 1 按照添加有序 2 能搜索。如果只是按照添加有序，用ArrayList或者LinkedList就行了，但是这样的话，根据歌名搜索就需要遍历，效率太低了，那么既然是根据歌名搜索，肯定有个: 歌名-歌曲映射。所以我们要采用映射表，又因为是按照添加有序，所以我们使用LinkedHashMap。

**LinkedHashMap**在创建的时候，可以指定accessOrder参数，如果为true，则会自动排序，将最新访问的元素放在前面，这样可以保证前面的元素是最近访问的，可以使用这个来实现LRUCache。

举个例子，假如我们要做一个app，需要在本地缓存好友列表，那么我们可以建立一张friends表，保存自己的好友，但是，如果有一天，我在这台设备上登录了另一个账号，那么获取的好友列表就是上一个账号的好友列表，所以，我们要根据不同的用户id建立多张表，让好友表跟用户id对应起来，但是又有问题了，如果我在这台设备挨个登录100个账户呢，难不成手机上存100个friends表？不行，这是手机，不是服务器，太费存储空间了，所以我们可以限制每台设备最多只存3个friends表，那么登录了第4个账号怎么办呢，我们删除第一个登录的吗？不对，我们应该删除最近最少登录的，也就是说: A登录，B登录，C登录，A登录，A虽然是第一个登录的，但是最近他又登录过了，那么B就成了最近最少登录的了，也就是最远登录的了，所以我们应该删除B账户的friends表。但是这样还有问题，假如我的登录顺序是: ABBBBBBBBBBCA，这样，你还删除B吗？不应该，因为B这么频繁登录，我们可以认为他是个常用用户，C和A只是偶尔上来看看，所以我们应该删除C，总的来说，我们不应该仅仅记录登录时间，还要记录登录次数，两个结合起来最少的那个，才应该被删除，这就是最近最少使用(Least Recent Use)，也就是LRU。我们的LRUCache就是这么个原理。

所以，LinkedHashMap适合**要按照元素存放顺序 或者 要实现一个LRU缓存**的场景。

**TreeMap**是按照key有序的，它是基于红黑树实现的，也就是说，每个元素的左孩子都比它小，右孩子都比它大，每次put或者remove元素的时候，都会调整到这种顺序，所以，它是基于二分的思想设计的。当我们需要元素有序的时候，就可以使用TreeMap。

至于**ConcurrentHashMap**，大部分人都很少用的，记住一点就行: 并发的时候，优先用它，因为对于客户端来说，高并发的场景很少见。

综上，**在移动端开发中，当我们的key是Integer时，用SparseArray，否则就用ArrayMap；要按照元素存放有序的 或者 实现LRU缓存，就用LinkedHashMap；当希望可以自定义排序的时候，用TreeMap；遇到并发的时候，使用ConcurrentHashMap即可**。

栈与队列

栈:先进后出，队列:先进先出。换句话，栈是对历史进行回溯，队列是对历史进行还原，回溯是从后往前的，还原是从前往后的。

栈在开发中的使用很少，但是却无处不在，比如我们常见的方法调用:

void funA() {

funB();

}

void funB() {

funC();

}

void funC() {

}

其实在JVM里面，有个方法栈，每个方法调用的是否都会创建一个栈帧并入栈，也就是说: funA()入栈，funB()入栈，funC()入栈，然后执行funC()，然后将funC()出栈，并把返回值给funB()，然后执行funB()，然后funB()出栈，并把结果给funA()，然后funA()出栈。

入栈: funA() -> funB() -> funC(); 出栈: funC() -> funB() -> funA();

所以，**JVM的方法调用是基于栈实现的**。这也体现了**回溯**的思想: 从这条路过去，沿着这条路回去!

队列在开发中很常见，无处不在，比如我们要实现个下载器来排队下载，那么就可以创建一个Queue，每次有任务过来，先添加到Queue里面，然后检测当前是否有任务下载，如果有，就算了，没有就出队并下载，下载完后再检测队列是否还有任务，有就出队并下载，以此类推。

队列在源码中的使用也很多，比如最常见的Handler里面的MessageQueue，再比如OkHttp里面的请求队列就是个双端队列，也是队列的一种。其实LinkedList也是一个双端队列，只不过队列是一种抽象的概念，而LinkedList是队列的一个具体实现。

然后我们来看下优先级队列PriorityQueue。**PriorityQueue是基于堆实现的，其本质是一个完全二叉树，可以分为大顶堆或小顶堆，大顶堆就是根节点最大，小顶堆就是根节点最小，并且，每次添加或者删除元素的时候，它会自动调整，使得剩下元素仍然是大顶堆/小顶堆，而且，遍历大顶堆，每次得到的元素都是最大的那个，小顶堆则是最小的那个**。

那么，优先级队列有什么用途呢，我们先来写个简单的观察者模式: 用户可以从书店订阅书籍变动的消息:

// 定义书本实体

class Book {

String name;

String title;

}

// 观察者接口

interface Observer {

void onBookChanged(Book book);

}

// 定义书店

class BookStore {

// 观察者集合

private List<Observer> observers = new ArrayList<>();

// 添加观察者

public void addObserver(Observer observer) {

if (!observers.contains(observer)) {

observers.add(observer);

}

}

// 移除观察者

public void removeObserver(Observer observer) {

observers.remove(observer);

}

// 通知观察者变更

private void notify(Book book) {

for (Observer observer : observers) {

observer.onBookChanged(book);

}

}

// 修改书籍

void changeBook(Book book) {

book.name = "android";

notify(book);

}

}

我们实现了一个观察者模式，当有书籍修改时，无差别的通知观察者，但是：现在突然有一天，老板说: 如果书籍有改动，先通知我，我怎么才能保证先通知老板呢？我们有两种方案:

* 1 在BookStore里面添加一个单独的观察者，表示老板，那么代码就类似下面这样:

// 定义书店

class BookStore {

private List<Observer> observers = new ArrayList<>();

// 添加了一个老板观察者

private Observer bossObserver;

// 这是注册，也是反注册，传null就是反注册

public void setBossObserver(Observer bossObserver) {

this.bossObserver = bossObserver;

}

public void addObserver(Observer observer) {

if (!observers.contains(observer)) {

observers.add(observer);

}

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

private void notify(Book book) {

// 先通知老板

if (bossObserver!=null) bossObserver.onBookChanged(book);

// 再通知其他人

for (Observer observer : observers) {

observer.onBookChanged(book);

}

}

void changeBook(Book book) {

book.name = "android";

notify(book);

}

}

这样可以吗？可以，但是，这是什么狗屁模式？有注册和反注册，还有setter函数，不伦不类的，而且！最重要的是:如果你有二老板三老板呢，难不成继续添加成员变量吗，肯定是不行的！这样维护起来太费劲了，而且，作为一个书店，还得知道谁是大老板，谁是小老板，其实作为书店，是不应该知道这么细的，这不符合最少知识原则。甚至有一天: 大老板说，你先发给二老板，然后再发给我，....无解！

那么，可以解决吗，可以的，我们使用第二种方法，就是: 让观察者有序，我们书店只管分发就行，只要观察者是有序的，那么最终就会按正确的顺序分发到老板和用户手中，我们这样改:

// 继承Comparable

interface Observer extends Comparable<Observer> {

// 观察者的优先级

int priority();

// 通知的回调

void onBookChanged(Book book);

// 默认实现比较函数，优先级高的先被分发到

@Override

default int compareTo(Observer o) {

return o.priority() - priority();

}

}

我们的BookStore需要稍微修改一下:

// 定义书店

class BookStore {

// 观察者集合

private List<Observer> observers = new ArrayList<>();

// 添加观察者

public void addObserver(Observer observer) {

if (!observers.contains(observer)) {

observers.add(observer);

// 添加了新观察者需要排序

Collections.sort(observers);

}

}

// 移除观察者

public void removeObserver(Observer observer) {

observers.remove(observer);

// 删除了观察者需要排序

Collections.sort(observers);

}

// 通知观察者变更

private void notify(Book book) {

for (Observer observer : observers) {

observer.onBookChanged(book);

}

}

// 修改书籍

void changeBook(Book book) {

book.name = "android";

notify(book);

}

}

改动很少，我们每次添加或删除观察者时，重新排序一下即可。那么，有没有更简单的方法呢，有！用优先级队列，我们前面说过，优先级队列，每次添加/删除元素会自动排序，并且遍历的时候，保证每次得到的都是剩下的最大/最小的元素，我们修改BookStore如下:

// 定义书店

class BookStore {

// 观察者集合，使用PriorityQueue

private PriorityQueue<Observer> observers = new PriorityQueue<>();

// 添加观察者

public void addObserver(Observer observer) {

if (!observers.contains(observer)) {

// 这里调用的是offer

observers.offer(observer);

}

}

// 移除观察者

public void removeObserver(Observer observer) {

observers.remove(observer);

}

// 通知观察者变更

private void notify(Book book) {

for (Observer observer : observers) {

observer.onBookChanged(book);

}

}

// 修改书籍

void changeBook(Book book) {

book.name = "android";

notify(book);

}

}

代码很简单，我们仅仅替换观察者集合为PriorityQueue，同时添加观察者使用offer()。上述代码扩展性很强，想怎么调整观察者顺序都可以，BookStore完全不用变，但是有个风险，就是Observer可以随便返回自己Priority，不是老板也可以返回老板的Priority，所以我们要将权限缩小，可以使用枚举，也可以使用check机制，这里不再废话。

基于上述代码，我们可以自己搭建一个NB的框架，我们可以定义一个事件分发器，一个数据处理器，一个UI处理器，当服务器数据过来时，我们会先在事件分发器接收到，然后分发给数据处理器，数据处理器会解析并保存数据，然后事件分发器接着分发给UI处理器，这时候UI处理器直接去数据处理器里面获取数据即可，代码大致如下:

// 使用枚举定义优先级

enum Priority {

DATA(10),

UI(0);

int priority;

Priority(int priority) {

this.priority = priority;

}

}

interface Observer extends Comparable<Observer> {

Priority priority();

void onBookChanged(Book book);

@Override

default int compareTo(Observer o) {

return o.priority().priority - priority().priority;

}

}

class DataHandler implements Observer {

@Override

public Priority priority() {

// 返回DATA的优先级

return Priority.DATA;

}

@Override

public void onBookChanged(Book book) {

// 存储数据

}

}

class UIHandler implements Observer {

@Override

public Priority priority() {

// 返回UI的优先级

return Priority.UI;

}

@Override

public void onBookChanged(Book book) {

// 获取并展示数据

}

}

原子类型

原子类型指的是AtomicInteger、AtomicBoolean等Atomic一族的类型，它们是基于CAS实现的，是线程安全的。

大家一定遇到过在lambda中使用一个局部变量，但是提示必须为final的问题，关键是要在lambda修改，怎么能声明为final呢，估计这时候有人就去给它提取为成员变量了，这是不对的，这时候就可以用Atomic一族了。

比如，如下代码: 首先用本地url，如果服务器又返回，就用服务器的，

void test() {

// 先取本地url

String url = "http://www.baidu.com";

// 获取服务器url

HttpAPi.getRealUrl("url/data", (bean) -> {

runOnUiThread(() -> {

if (bean != null) {

if (!TextUtils.isEmpty(bean.url)) {

// 又返回，就用服务器url，但是这里报错了，提示在lambda里只能用final，mmp的！

url = bean.url;

}

}

// 打印，又报错

println(url);

});

});

}

上述提示说，在lambda里面只能访问final修饰的变量，但是我想修改它，又不能用final修饰。

void test() {

// 先取本地url。这里创建一个AtomicReference

AtomicReference<String> url = new AtomicReference<>("http://www.baidu.com");

// 获取服务器url

HttpAPi.getRealUrl("url/data", (bean) -> {

runOnUiThread(() -> {

if (bean != null) {

if (!TextUtils.isEmpty(bean.url)) {

// 这里直接set就行

url.set(bean.url);

}

}

// 打印，直接使用get()获取

println(url.get());

});

});

}

可以看到，Atomic系列就像一个仓库，使用put/get存取，很方便。而且它是基于CAS的，是乐观的，不存在效率方面的问题。

 非映射关系，随机访问多，插入删除少的，使用ArrayList；反之则使用LinkedList。

 映射关系，key是Integer的，使用SparseArray，否则使用ArrayMap；如果对性能要求比较高，可以使用HashMap，如果要求按照存放有序，或者实现LRUCache，则使用LinkedHashMap。如果需要自己排序，则可以使用TreeMap，如果有并发场景，则使用ConcurrentHashMap。

 要求先进先出，考虑使用Queue，先进后出则使用Stack，要求元素自动排序，则可以使用PriorityQueue，遇到lambda使用/修改局部变量，则可以使用Atomic系列。