数据结构与算法（一）

1. 课程目标

1：理解什么是线性表

2：掌握数组数据结构，理解 ArrayList 的源码

3：掌握链表数据结构，理解 LinkedList 的源码

4：掌握栈这种数据结构，理解 Stack 的部分源码

5：掌握队列这种数据结构

2. 入门概述

2.1：什么是数据结构和算法

很多教材或者教程在开篇的时候都会来介绍这两个概念，但是概念毕竟是抽象的，所以我们不需要死扣定义，毕竟我们不是为了考试而学的，但这并不是说我们不需要理解其概念，我们只是说不要陷入概念的怪圈。下面我们来介绍一下相关概念：

1. 数据结构包括数据对象集以及它们在计算机中的组织方式，即它们的逻辑结构和物理存储结构，一般我们可以认为数据结构指的是一组数据的存储结构。

2. 算法就是操作数据的方法，即如何操作数据效率更高，更节省资源。

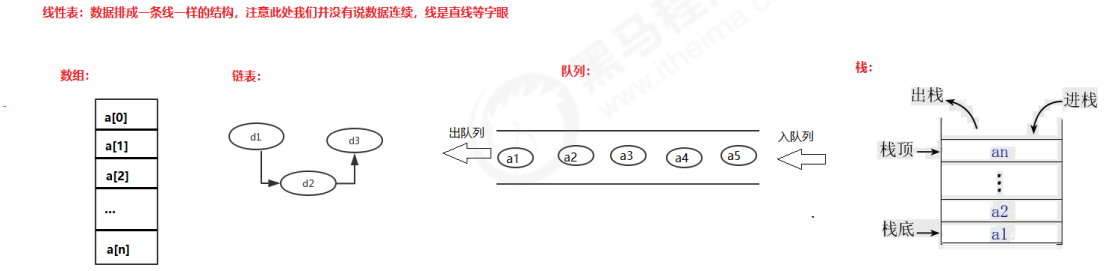
这只是抽象的定义，我们来举一个例子，你有一批货物需要运走，你是找小轿车来运还是找卡车来运？这就是数据结构的范畴，选取什么样的结构来存储；至于你货物装车的时候是把货物堆放在一起还是分开放这就是算法放到范畴了，如何放置货物更有效率更节省空间。

数据结构和算法看起来是两个东西，但是我们为什么要放在一起来说呢？那是因为数据结构和算法是相辅相成的，数据结构是为算法服务的，而算法要作用在特定的数据结构之上，因此，我们无法孤立数据结构来讲算法，也无法孤立算法来讲数据结构。

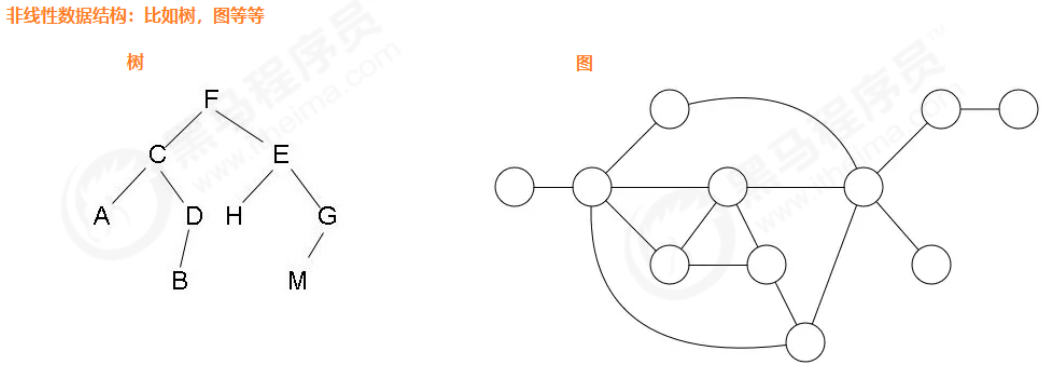
在学习算法之前我们先来学习几个最基本的数据结构，这是后续学习其他数据结构和算法的基础，也就是我们今天要来学习的：数组，链表，栈，队列。这个四个数据结构又可称之为线性表数据结构。

2.2：线性表概念

所谓的线性表，就是将数据排成像一条长线一样的结构，数组，链表，栈，队列都是线性表结构，线性表上的每个数据最多只有前后两个方向，下面以一幅图的形式来展现一下线性表结构



与这种线性结构对应的就是非线性结构，比如后续要学习的数，堆，图等，在这些非线性数据结构中，数据之间并不是简单的前后关系，如下图：



好，理解完线性表的概念之后，接下来正式学习今天的四个数据结构

3. 数组

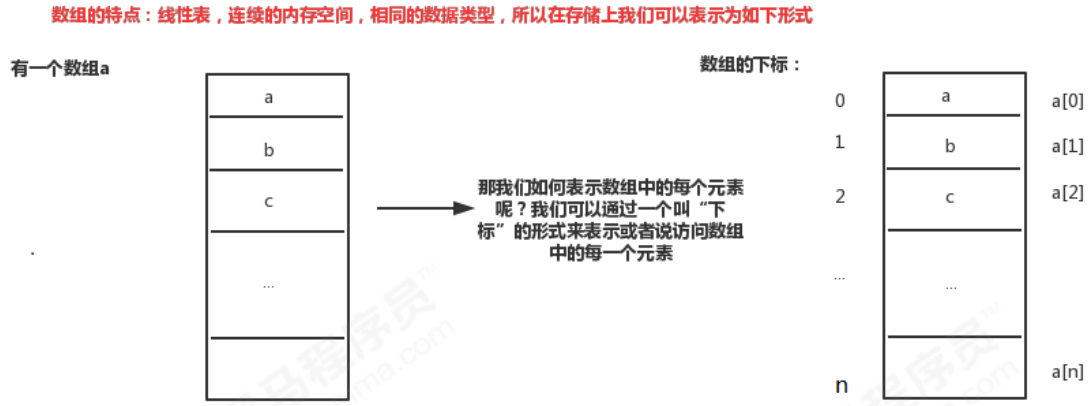
3.1：概念

数组是一种**线性表数据结构**，它用**一组连续的内存空间**，来存储一组具有**相同类型的数据**。这里我们要抽取出三个跟数组相关的关键词：线性表，连续内存空间，相同数据类型；数组具有连续的内存空间，存储相同类型的数据，正是该特性使得数组具有一个特性：随机访问。但是有利有弊，这个特性虽然使得访问数组边得非

常容易但是也使得数组插入和删除操作会变得很低效，插入和删除数据后为了保证连续性，要做很多数据搬迁工作。

3.2：逻辑结构和物理结构

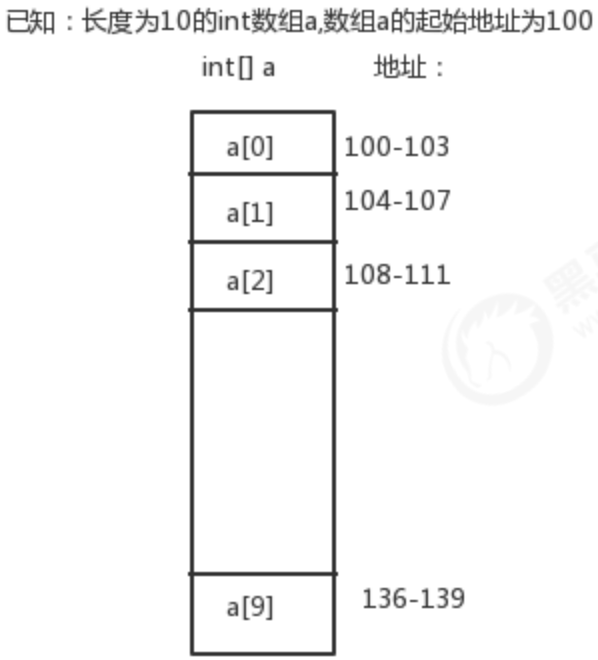
所谓的数组的逻辑结构指的是我们可以用什么的表示方式来描述数组元素，比如有一个数组 a,数组中有 n 个元素，我们可以用(a1,a2,a3,.....an)来描述数组中的每个元素，当然后面我们会将具体如何访问数组中的每个元素。数组的物理结构指的是数组元素实际的存储形式，当然了从概念我们可以看出数组的物理存储空间是一块连续的内存单已，为了说明白这个事情，这里给大家准备了一副图



从图中我们可以看出访问数组中的元素是通过下标来访问的，那是如何做到的呢？

3.2.1：数组元素的访问

我们拿一个长度为 10 的数组来举例，int [] a= new int[10]，在下面的图中，计算机给数组分配了一块连续的空间，100-139，其中内存的起始地址为 baseAddress=100



我们知道，计算机给每个内存单元都分配了一个地址，通过地址来访问其数据，因此要访问数组中的某个元素时，首先要经过一个寻址公式计算要访问的元素在内存中的地址：

```
a[i] = baseAddress + i * dataTypeSize
```

其中 dataTypeSize 代表数组中元素类型的大小，在这个例子中，存储的是 int 型的数据，因此 dataTypeSize=4 个字节

3.2.2：数组下标为什么从 0 开始

数组的下标为什么要从 0 开始而不是从 1 开始呢？

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）”。前面也讲到，如果用 array 来表示数组的首地址，array[0] 就是偏移为 0 的位置，也就是首地址，array[k] 就表示偏移 k 个 type_size 的位置，所以计算 array[k] 的内存地址只需要用这个公式:

```
array[k]_address = base_address + k * type_size
```

但是如果下标从 1 开始，那么计算 array[k]的内存地址会变成：

```
array[k]_address = base_address + (k-1)*type_size
```

对比两个公式，不难发现从数组下标从 1 开始如果根据下标去访问数组元素，对于 CPU 来说，就多了一次减法指令。

当然另一方面也是由于历史原因，c 语言设计者使用 0 开始作为数组的下标，后来的高级语言沿用了这一设计。

3.3：数组的特点

3.3.1：高效的随机访问

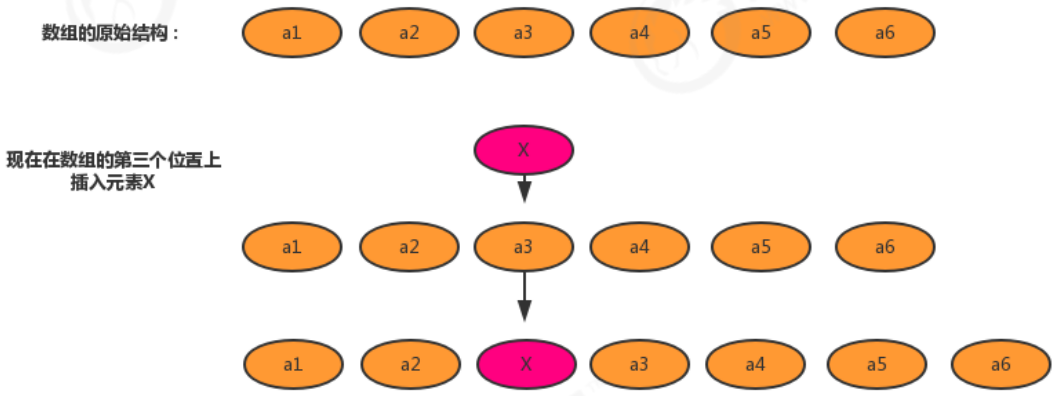
通过前面的学习我们已经知道，数组元素的访问是通过下标来访问的，计算机通过数组的首地址和寻址公式能够很快速的找到想要访问的元素

3.3.2：低效插入和删除

前面我们已经讲到数组是一段连续的内存空间，因此为了保证数组的连续性会使得数组的插入和删除的效率变的很低，下面我们来分析一下，

- 插入

假设数组的长度为 n，现在如果我们需要将一个数据插入到数组中的第 k 个位置。为了把第 k 个位置腾出来给新来的数据，我们需要将第 k～n 这部分的元素都顺序地往后挪一位。如下图所示：



那数组插入有没有相对优化的方案呢？

如果数组中的数据是有序的，我们在某个位置插入一个新的元素时，就必须按照刚才的方法搬移 k 之后的数据。但是，如果数组中存储的数据并没有任何规律，数组只是被当作一个存储数据的集合。在这种情况下，如果要将某个数组插入到第 k

个位置，为了避免大规模的数据搬移，我们还有一个简单的办法就是，直接将第 k 位的数据搬移到数组元素的最后，把新的元素直接放入第 k 个位置。这种处理思想会在快排中用到

- **删除**

如果我们要删除第 k 个位置的数据，为了内存的连续性，也需要搬移数据，不然中间就会出现空洞，内存就不连续了。

实际上，在某些特殊场景下，我们并不一定非得追求**数组中数据的连续性**。如果我们将多次删除操作集中在一起执行，删除的效率是不是会提高很多呢？举个例子

数组 a[6] 中存储了 6 个元素：a1,a2,a3,a4,a5,a6。现在，我们要依次删除 a1,a2 这两个元素。

为了避免 a3,a4,a5,a6 这几个数据会被搬移两次，我们可以先记录下已经删除的数据。每次的删除操作并不是真正地搬移数据，只是记录数据已经被删除。当数组没有更多空间存储数据时，我们再触发执行一次真正的删除操作，这样就大大减少了删除操作导致的数据搬移。

如果你了解 JVM，你会发现，这不就是 JVM 标记清除垃圾回收算法的核心思想吗？没错，数据结构和算法的魅力就在于此，**很多时候我们并不是要去死记硬背某个数据结构或者算法，而是要学习它背后的思想和处理技巧，这些东西才是最有价值的**。如果你细心留意，不管是在软件开发还是架构设计中，总能找到某些算法和数据结构的影子

3.4：数组的应用

针对数组类型，很多语言都提供了容器类，比如 Java 中的 ArrayList、C++ STL 中的 vector。在项目开发中，什么时候适合用数组，什么时候适合用容器呢？

这里我拿 Java 语言来举例。如果你是 Java 工程师，几乎天天都在用 ArrayList，对它应该非常熟悉。那它与数组相比，到底有哪些优势呢？

我个人觉得，ArrayList 最大的优势就是可以将很多数组操作的细节封装起来。比如前面提到的数组插入、删除数据时需要搬移其他数据等。另外，它还有一个优势，

就是 **支持动态扩容** 。数组本身在定义的时候需要预先指定大小，因为需要分配连续的内存空间。如果我们申请了大小为 10 的数组，当第 11 个数据需要存储到数组中时我们就需要重新分配一块更大的空间，将原来的数据复制过去，然后再将新的数据插入。如果使用 ArrayList，我们就完全不需要关心底层的扩容逻辑，ArrayList 已经帮我们实现好了。每次存储空间不够的时候，它都会将空间自动扩容为 1.5 倍大小。不过，这里需要注意一点，因为扩容操作涉及内存申请和数据搬移，是比较耗时的。所以，**如果事先能确定需要存储的数据大小，最好在创建 ArrayList 的时候事先指定数据大小**。

如果你对以上关于 ArrayList 的讲解不太明白，完全不用担心，因为接下来我们就针对 java 中 ArrayList 底层操作逻辑进行分析，理解数组的应用及 ArrayList 底层的扩容逻辑。

3.4.1：ArrayList 源码分析

首先我们要明确，ArrayList 底层是采用数组来进行数据的存储，其次 ArrayList 提供了相关的方法来对底层数组中的元素进行操作，包括数据的获取，数据的插入，数据的删除等，此次课程中不会完全讲解所有的操作，笔者将会从两个个方面来进行 ArrayList 底层源码发剖析。

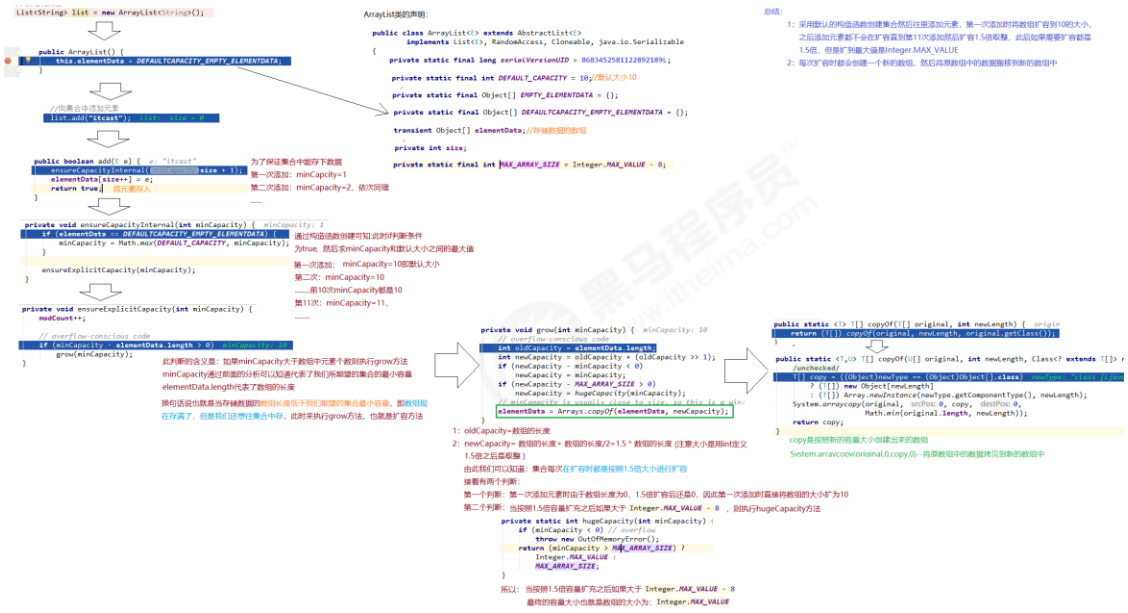
3.4.2：容器构建及添加元素

这里有一段非常简短的代码

```
public static void main(String[] args) {
        //初始化集合
        List<String> list = new ArrayList<String>();

        //向集合中添加元素
        list.add("itcast");
    }
```

下面我们以断点调试的方式查看一下，当我们创建集合对象的时候发生了什么事



通过分析可以知道：

1：创建 ArrayList 时采用默认的构造函数创建集合然后往里添加元素，第一次添加时将数组扩容到 10 的大小，之后添加元素都不会在扩容，直到第 11 次添加然后扩容 1.5 倍取整，此后如果需要扩容都是 1.5 倍取整，但是扩容到的最大值是 Integer.MAX_VALUE

2：每次扩容时都会创建一个新的数组，然后将原数组中的数据搬移到新的数组中

所以回到开始我们所说的：**如果事先能确定需要存储的数据大小，最好在创建 ArrayList 的时候事先指定数据大小**

我们可以查看 ArrayList 的带参构造函数：

```
public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            //直接使用传递的容量大小创建数组，这样的话只会在不够存储的时候才会需要扩容
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
```

比如我们要从数据库中取出 10000 条数据放入 ArrayList。我们看下面这几行代码，你会发现，相比之下，事先指定数据大小可以省掉很多次内存申请和数据搬移操作。

```
ArrayList<User> users = new ArrayList(10000);
for (int i = 0; i < 10000; ++i) {
  users.add(xxx);
}
```

3.4.3：获取元素

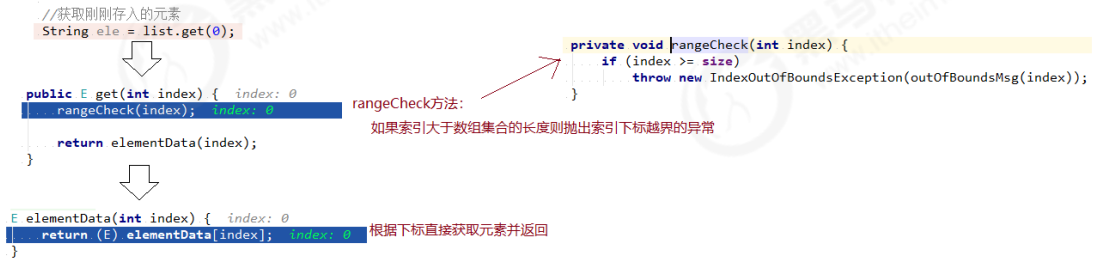
接下来我们写一段代码来分析一下从集合中获取元素

```
public static void main(String[] args) {
        //初始化集合
        List<String> list = new ArrayList<String>(1);

        //向集合中添加元素
        list.add("itcast");

        //获取刚刚存入的元素
        String ele = list.get(0);
    }
```

分析查看集合的 get 方法



到这里我们关于 ArrayList 的底层源码解析就结束了，当然了我们只分析了 ArrayList 中很小一部分源码，它还提供了很多的方法，大家不妨自己去分析分析看！同时通过我们的分析，你能不能自己手动实现一个基于数组并且支持动态扩容的集合类呢？写写看吧！

作为高级语言编程者，是不是数组就无用武之地了呢？当然不是，有些时候，用数组会更合适些，我总结了几点自己的经验。

1.Java ArrayList 无法存储基本类型，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。

2.如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。

我总结一下，对于业务开发，直接使用容器就足够了，省时省力。毕竟损耗一丢丢性能，完全不会影响到系统整体的性能。但如果你是做一些非常底层的开发，比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。

至此，数组部分的学习就完成了，接下来我们进入链表的学习。

4. 链表

4.1：概念

链表（Linked list）是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

读完这段对链表的解释之后我们可能还是不太明白链表到底是怎么一回事，那接下来我们从底层的存储结构开始解开链表的面纱

4.2：存储结构

相比数组，链表是一种稍微复杂一点的数据结构。对于初学者来说，掌握起来也要比数组稍难一些。这两个非常基础、非常常用的数据结构，我们常常将会放到一块儿来比较。所以我们先来看，这两者有什么区别。

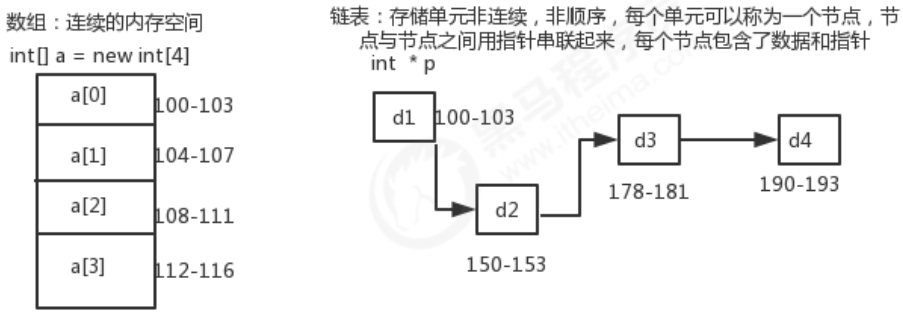
首先我们从底层存储结构来看：

数组：需要一块连续的存储空间，对内存的要求比较高，比如我们要申请一个 1000M 的数组，如果内存中没有连续的足够大的存储空间则会申请失败，即便内存的剩余可用空间大于 1000M，仍然会申请失败。

链表：与数组相反，它并不需要一块连续的内存空间，它通过指针将一组**零散的内存块**串联起来使用，所以如果我们申请一个 1000M 大小的链表，只要内存剩余的可用空间大于 1000M，便不会出现问题。

下方图片对比了链表和数组的存储结构

数组和链表的内存存储结构对比



此时我们在来回顾我们刚刚提到的链表的概念：物理存储单元上非连续、非顺序，元素的逻辑顺序是通过链表中的指针链接次序实现的，链表由一系列结点组成，每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域

当然了这只是我们提到的链表的最基本的存储形式，其实链表有很多种不同的类型，分别对应了不同的结构，接下来我们就来分析不同的链表类型

4.3：链表类型

4.3.1：单链表

所谓的单链表就是我们刚刚讲到的链表的最基本的结构，链表通过指针将一组零散的内存块串联在一起。其中，我们把内存块称为链表的“**结点**”。为了将所有的结点串起来，每个链表的结点除了存储数据之外，还需要记录链上的下一个结点的地址。如图所示，我们把这个记录下个结点地址的指针叫作**后继指针 next**，如果链表中的某个节点为 p，p 的下一个节点为 q，我们可以表示为：p->next=q

下面的图更加详细的描述了单链表的存储结构

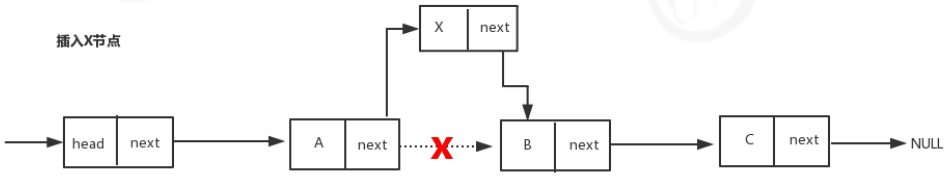
单链表结构：

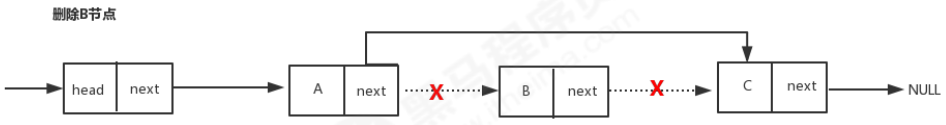
从我画的单链表图中，你应该可以发现，其中有两个结点是比较特殊的，它们分别是第一个结点和最后一个结点。我们习惯性地把**第一个结点叫作头结点**，把**最后一个结点叫作尾结点**。其中，**头结点用来记录链表的基地址**,有了它，我们就可以遍历得到整条链表。而**尾结点特殊的地方是：指针不是指向下一个结点，而是指向一个空地址 NULL**，表示这是链表上最后一个结点。

与数组一样，链表也支持数据的查找、插入和删除操作。

我们知道，在进行数组的插入、删除操作时，为了保持内存数据的连续性，需要做大量的数据搬移。而在链表中插入或者删除一个数据，我们并不需要为了保持内存的连续性而搬移结点，因为链表的存储空间本身就不是连续的。所以，在链表中插入和删除一个数据是非常快速的。

为了方便你理解，我画了一张图，从图中我们可以看出，针对链表的插入和删除操作，我们只需要
考虑相邻结点的指针改变

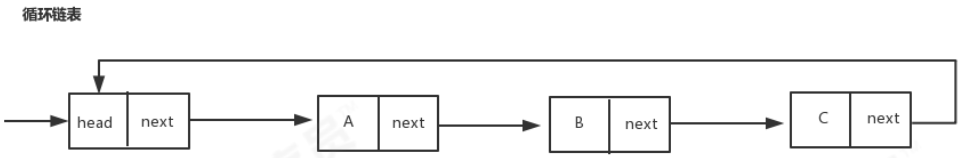




但是，有利就有弊。链表要想随机访问第 k 个元素，就没有数组那么高效了。因为链表中的数据并非连续存储的，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据针一个结点一个结点地依次遍历，直到找到相应的结点，所以，链表随机访问的性能没有数组好。

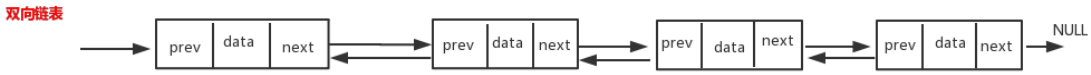
4.3.2：循环链表

循环链表是一种特殊的单链表。实际上，循环链表也很简单。它跟单链表唯一的区别就在尾结点。我们知道，单链表的尾结点指针指向空地址，表示这就是最后的结点了。而循环链表的尾结点指针是指向链表的头结点。从我画的循环链表图中，你应该可以看出来，它像一个环一样首尾相连，所以叫作“循环”链表，和单链表相比，循环链表的优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用循环链表，循环链表的结构如图所示



4.3.3：双向链表

单向链表只有一个方向，结点只有一个后继指针 next 指向后面的结点。而双向链表，顾名思义，它支持两个方向，每个结点不止有一个后继指针 next 指向后面的结点，还有一个前驱指针 prev 指向前面的结点，如图所示



从图中可以看出来，双向链表需要额外的两个空间来存储后继结点和前驱结点的地址。所以，如果存储同样多的数据，双向链表要比单链表占用更多的内存空间。虽然两个指针比较浪费存储空间，但可以支持双向遍历，这样也带来了双向链表操作的灵活性。

4.3.4：双向循环链表

了解了循环链表和双向链表，如果把这两种链表整合在一起就是一个**双向循环链表**

4.4：链表和数组性能比较

通过前面内容的学习，你应该已经知道，数组和链表是两种截然不同的内存组织方式。正是因为内存存储的区别，它们插入、删除、随机访问操作的性能正好相反

数组简单易用，在实现上使用的是连续的内存空间,缺点是大小固定，一经声明就要占用整块连续内存空间。如果声明的数组过大，系统可能没有足够的连续内存空间分配给它，导致“内存不足（out of memory）”。如果声明的数组过小，则可能出现不够用的情况。这时只能再申请一个更大的内存空间，把原数组拷贝进去，非常费时。

链表本身没有大小的限制，天然地支持动态扩容，我觉得这也是它与数组最大的区别。你可能会说，我们 Java 中的 ArrayList 容器，也可以支持动态扩容啊？我们上一节课讲过，当我们往支持动态扩容的数组中插入一个数据时，如果数组中没有空闲空间了，就会申请一个更大的空间，将数据拷贝过去，而数据拷贝的操作是非常耗时的。

所以：数组的优势是查询速度非常快，但是增删改慢；链表的优势是增删改快，但是查询慢。

4.5：链表的应用

学习数组的时候我们分析了基于数组而实现的 ArrayList 集合，那 java 中也有基于链表实现的集合 LinkedList，接下来我们就来分析一下 LinkedList 的底层源码

4.5.1：LinkedList 源码分析

首先我们写一段最简单的代码如下

```
public static void main(String[] args) {
        //构建容器
        List<String> list = new LinkedList<String>();
        //向容器中添加数据
        list.add("itheima");
        list.add("itcast");

    }
```

然后我们以断点调试的方式进入底层源码进行查看，我们主要关注容器的构建和元素的添加

4.5.2：容器构建及添加元素

1：点开 LinkedList 源码查看 LinkedList 类的声明及属性

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;
    transient Node<E> first;//存储整个链表的头节点
    transient Node<E> last;//存储整个链表的尾节点
    //LinkedList 构造函数
     public LinkedList() {
    }

    //静态内部类Node，也就是我们链表中的概念：节点
    private static class Node<E> {
        E item;//  节点中存储的数据
        Node<E> next;//节点中存储的后继指针next
        Node<E> prev;//节点中存储的前驱指针prev

            //节点的构造函数
        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    //添加数据
    public boolean add(E e) {
        linkLast(e);
        return true;
    }
```

```
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
}
```

调试过程如图所示：

第一次添加：itheima

1：此时因为容器刚创建last尾节点为null,通过Node带参构造函数创建了一个新的节点newNode
在创建newNode节点时将新节点的prev指针指向last尾节点，由于此时是第一次添加last=null,所以 newNode.prev=null
newNode.next=null, newNode.item=itheima

2：将newNode作为last尾节点

3：由于是第一次添加所以 头节点first=null，因此将新创建的newNode作为头节点first

4：容器长度size加1

如右图

第二次添加：itcast

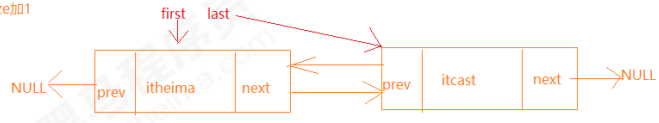
1：l=last也就是刚刚的节点itheima，然后创建了一个新节点newNode，通过节点构造函数可知，newNode.prev=itheima节点，newNode.next=null,newNode.item=itcast

2：last=newNode，说明现在尾节点变成了我们刚刚创建的节点itcast

3：变量l现在为itheima节点，然后l.next=newNode即，itheima节点.next=newNode，也就是itheima节点.next=itcast节点

4：容器长度size加1

如右图



依次同理

通过分析我们可以知道，LinkedList 内部采用双向链表实现数据的存储，那接下来我们来分析一下，如果从 LinkedList 容器中获取数据

4.5.3：获取元素

从 LinkedList 中获取元素的代码如下：

```
public static void main(String[] args) {
        //构建容器
        List<String> list = new LinkedList<String>();
        //向容器中添加数据
        list.add("itheima");
        list.add("itcast");
        //获取元素
        String s = list.get(1);
        System.out.println(s);
    }
```

查看 LinkedList 中的 get 方法，源码如下:

```
public E get(int index) {
    //判断索引是否越界
        checkElementIndex(index);
        return node(index).item;
    }
private void checkElementIndex(int index) {
        if (!isElementIndex(index))
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
    }
private boolean isElementIndex(int index) {
    //如果索引不在[0，size),范围内就抛出索引越界异常，其中size 是集合的长度
        return index >= 0 && index < size;
    }
//根据索引从链表中获取节点
Node<E> node(int index) {
        // assert isElementIndex(index);
    //如果索引小于集合长度的一半则从头节点开始遍历，否则从尾节点开始遍历,因为底层用的是双向链表支持
    //前后查询遍历
        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
```

```
                x = x.prev;
            return x;
        }
    }
```

代码跟踪效果如下：

至此，关于 LinkedList 的源码解析就告一段落了，本课程中只分析了构建集合，向集合中添加元素以及根据索引从集合中获取元素的相关方法，其他方法的分析大家可以尝试着自己分析一下。

4.5.4：面试题：

(1)、LinkedList 和 ArrayList 的比较

前面我们分析过 ArrayList 的源码，现在又分析了 LinkedList 源码，接下来将二者进行比较：

1：ArrayList 的实现基于数组，LinkedList 的实现基于双向链表

2：对于随机访问，ArrayList 优于 LinkedList，ArrayList 可以根据下标对元素进行随机访问。而 LinkedList 的每一个元素都依靠地址指针和它后一个元素连接在一起，在这种情况下，查找某个元素只能从链表头开始查询直到找到为止

3：对于插入和删除操作，LinkedList 优于 ArrayList，因为当元素被添加到 LinkedList 任意位置的时候，不需要像 ArrayList 那样重新计算大小或者是更新索引。

4：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

(2)、反转单链表

上一小节我们分析了 LinkedList 的底层源码，那对于数据结构和算法来说我们不仅仅要掌握理论知识点，还要能够根据我们掌握的理论知识点去解题，达到对我们思维上的训练，接下来我们就以一道跟链表相关的面试来训练一下我们的思维及代码编写能力

反转单链表：

英文版：https://leetcode.com/problems/reverse-linked-list/

中文版：https://leetcode-cn.com/problems/reverse-linked-list/

leetcode 编码参考答案：

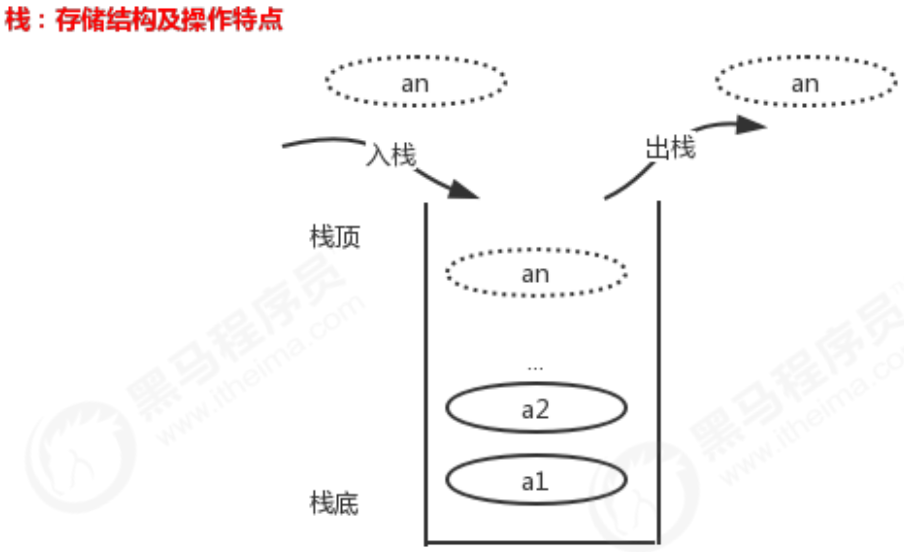
```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null; //前指针节点
        ListNode curr = head; //当前指针节点
        //每次循环，都将当前节点指向它前面的节点，然后当前节点和前节点后移
        while (curr != null) {
            ListNode nextTemp = curr.next; //临时节点，暂存当前节点的下一节点，用于后移
            curr.next = prev; //将当前节点指向它前面的节点
            prev = curr; //前指针后移
            curr = nextTemp; //当前指针后移
        }
        return prev;
    }
}
```

至此，链表部分的知识就学习完成了，接下来我们学习“栈”这种数据结构。

5. 栈

5.1：概念

关于“栈”这种数据结构，它有一个典型的特点：**先进后出，后进先出**；只要满足这种特点的数据结构我们就可以说这是典型的“栈”数据结构，我们一般将这个特点归纳为一个：后进先出,英文表示为：Last In First Out 即 **LIFO**。为了更好的理解栈这种数据结构，我们以一幅图的形式来表示，如下：



我们从栈的操作特点上来看，似乎受到了限制，的确，**栈就是一种操作受限的线性表，只允许在栈的一端进行数据的插入和删除，这两种操作分别叫做入栈和出栈。当某个数据集合如果只涉及到在其一端进行数据的插入和删除操作，并且满足先进后出，后进先出的特性时，我们应该首选栈这种数据结构来进行数据的存储。**

5.2：栈的实现

从对栈的定义中我们发现栈主要包含两个操作，入栈和出栈，也就是在栈顶插入一个元素和在栈底删除一个元素，那理解了这个定义之后，我们来思考一个问题，如何来手动实现一个栈呢？

实际上，栈既可以用数组来实现，也可以用链表来实现。用数组实现的栈叫**顺序栈**，用链表实现的叫**链式栈**。接下来我们就这两种形式分别去实现一下。

5.2.1：基于数组顺序栈的实现

以下是基于数组的顺序栈的实现代码

```
public class ArrayStack {
    // 栈大小
    private int size;
    // 默认栈容量
    private int DEFAULT_CAPACITY=10;
    // 栈数据
    private Object[] elements;

    private int MAX_ARRAY_SIZE  = Integer.MAX_VALUE-8;

    /**
     * 默认构造创建大小为 10 的栈
     */
    public ArrayStack(){
        elements = new Object[DEFAULT_CAPACITY];
    }

    /**
     * 通过指定大小创建栈
     * @param capacity
     */
    public ArrayStack(int capacity){
        elements = new Object[capacity];
    }

    /**
     * 入栈
     * @param element
     * @return
     */
    public boolean push(Object element){
        try {
            checkCapacity(size+1);
```

```
            elements[size++]=element;
            return true;
        }catch (RuntimeException e){
            return false;
        }
    }

    /**
     * 检查栈容量是否还够
     */
    private void checkCapacity(int minCapacity) {
        if(elements.length - minCapacity < 0 ){
             throw  new RuntimeException("栈容量不够!");
        }
    }

    /**
     * 出栈
     * @return
     */
    public Object pop(){
        if(size<=0){
            return null;//栈为空则直接返回null
        }
        Object obj = elements[size-1];
        elements[--size] = null;
        return obj;
    }

    /**
     * 获取栈的大小
     * @return
     */
    public int size(){
        return size;
    }

}
```

编写好用数组实现的栈之后，我们来编写一个测试类如下：

```
public class ArrayStackTest {

    public static void main(String[] args) {
        ArrayStack stack = new ArrayStack();
        for (int i=0; i<13; i++) {
            boolean push = stack.push(i);
```

```
            System.out.println("第"+(i+1)+"次存储数据为:"+i+",存储结果是:
"+push);
        }
        // stack.push(1);
        for (int i=0; i<11; i++) {
            Object pop = stack.pop();
            System.out.println(pop);
        }
    }
}
```

5.2.2: 支持动态扩容的顺序栈

刚才那个基于数组实现的栈，是一个固定大小的栈，也就是说，在初始化栈时需要事先指定栈的大小。当栈满之后，就无法再往栈里添加数据了。那我们如何基于数组实现一个可以支持动态扩容的栈呢？

你还记得，我们在数组那一节，是如何来实现一个支持动态扩容的数组的吗？当数组空间不够时，我们就重新申请一块更大的内存，将原来数组中数据统统拷贝过去。这样就实现了一个支持动态扩容的数组。

所以，如果要实现一个支持动态扩容的栈，我们只需要底层依赖一个支持动态扩容的数组就可以了。当栈满了之后，我们就申请一个更大的数组，将原来的数据搬移到新数组中。

下面是在前面顺序栈的基础之上添加了支持动态扩容后的代码：

```
public class ArrayStack {
    // 栈大小
    private int size;
    // 默认栈容量
    private int DEFAULT_CAPACITY=10;
    // 栈数据
    private Object[] elements;

    private int MAX_ARRAY_SIZE  = Integer.MAX_VALUE-8;

    /**
     * 默认构造创建大小为 10 的栈
     */
    public ArrayStack(){
        elements = new Object[DEFAULT_CAPACITY];
    }

    /**
     * 通过指定大小创建栈
```

```
     * @param capacity
     */
    public ArrayStack(int capacity){
        elements = new Object[capacity];
    }

    /**
     * 入栈
     * @param element
     * @return
     */
    public boolean push(Object element){
        try {
            checkCapacity(size+1);
            elements[size++]=element;
            return true;
        }catch (RuntimeException e){
            return false;
        }
    }

    /**
     * 检查栈容量是否还够
     */
    private void checkCapacity(int minCapacity) {
        if(elements.length - minCapacity < 0 ){
            //throw  new RuntimeException("栈容量不够!");
            grow(elements.length);
        }
    }

    /**
     * 扩容
     * @param oldCapacity 原始容量
     */
    private void grow(int oldCapacity) {
        int newCapacity = oldCapacity+(oldCapacity>>1);
        if(newCapacity-oldCapacity<0){
            newCapacity = DEFAULT_CAPACITY;
        }
        if(newCapacity-MAX_ARRAY_SIZE>0){
            newCapacity = hugeCapacity(newCapacity);
        }
        elements = Arrays.copyOf(elements,newCapacity);
    }
    private int hugeCapacity(int newCapacity) {
```

```
        return (newCapacity>MAX_ARRAY_SIZE)? Integer.MAX_VALUE:newCapac
ity;
    }

    /**
     * 出栈
     * @return
     */
    public Object pop(){
        if(size<=0){
            return null;//栈为空则直接返回null
        }
        Object obj = elements[size-1];
        elements[--size] = null;
        return obj;
    }

    /**
     * 获取栈的大小
     * @return
     */
    public int size(){
        return size;
    }

}
```

添加测试代码如下：

```
public class ArrayStackTest {

    public static void main(String[] args) {
        ArrayStack stack = new ArrayStack(5);
        for (int i=0; i<40; i++) {
            boolean push = stack.push(i);
            System.out.println("第"+(i+1)+"次存储数据为:"+i+",存储结果是:
"+push);
        }
        // stack.push(1);
        for (int i=0; i<11; i++) {
            Object pop = stack.pop();
            System.out.println(pop);
        }
    }
}
```

5.2.3: 基于链表的链式栈的实现

上一小节我们手动实现了基于数组的顺序栈，这一小节中我们来实现一下基于链表的链式栈，基于链表的栈跟基于数组的顺序栈有一个很大的区别就是链式栈天生就具备动态扩容的特点。

我们知道链表中的每一个元素都可以称之为节点，因此我们先创建一个节点对象如下：

```
public class Node {
    //前驱节点
    public Node prev;
    //节点数据
    private Object data;
    //后继节点
    public Node next;

    public Node(Node prev,Object data,Node next){
        this.prev = prev;
        this.data =data;
        this.next = next;
    }

    public Object getData() {
        return data;
    }
}
```

接下来实现一个链式栈，提供出栈和入栈的操作：

```
/**
 * 基于双向链表的链式栈实现
 */
public class LinkedListStack {
    // 栈大小
    private int size;
    //存储链表尾节点
    private Node tail;

    public LinkedListStack(){
        this.tail = null;
    }

    /**
     * 入栈
     * @param data
```

```
     * @return
     */
    public boolean push(Object data){
        Node newNode = new Node(tail,data,null);
        if(size>0){
            tail.next = newNode;
        }
        tail = newNode;
        size++;
        return true;
    }

    /**
     * 出栈
     * @return
     */
    public Object pop(){
        if((size-1) < 0){
            //栈为空
            return  null;
        }
        Object data = tail.getData();
        tail = tail.prev;
        if(tail!=null){
            tail.next = null;
        }
        size--;
        return data;
    }
}
```

从以上的实现我们可以看出，我们采用的是双向链表来实现的链式栈，入栈和出栈操作虽然都很快速，但是由于双向链表需要额外的存储空间来存储前驱指针，因此我们这段程序在空间资源的节省上显得力度不够，所以我们想能不能不用双向链表只用单向链表来解决这个问题呢？答案是肯定的，接下来我们就针对刚刚的代码做出改造

首先改造我们的节点对象，每个节点对象中只存储数据和 next 指针，并且上面的实现我们是单独创建的节点对象，那现在我们将节点对象声明为栈的内部类

```
/**
 * 基于单链表实现的栈
 */
public class StackBasedOnLinkedList {
    // 存储链表头节点
    private Node head;
```

```
public StackBasedOnLinkedList(){
    this.head = null;
}

/**
 * 入栈
 * @param data
 * @return
 */
public boolean push(Object data){
    Node newNode = new Node(data,head);
    head = newNode;
    return true;
}

/**
 * 出栈
 * @return
 */
public Object pop(){
    if(head==null){
        return null;
    }
    Node topNode = head;
    head = topNode.next;
    topNode.next=null;
    return topNode.data;
}


/**
 * 节点对象
 */
private static  class Node {
    //节点数据
    private Object data;
    // next 指针
    private Node next;

    public Node(Object data,Node next){
        this.data =  data;
        this.next = next;
    }

    public Object getData() {
```

```
            return data;
        }
    }
}
```

然后我们编写测试类如下：

```
public class TestStackBasedOnLinkedList {
    public static void main(String[] args) {
        StackBasedOnLinkedList stack = new StackBasedOnLinkedList();
        for(int i=0;i<6;i++){
            stack.push(i+"");
            System.out.println("第"+(i+1)+"次入栈,入栈的值为:"+i);
        }

        for(int i=0;i<8;i++){
            Object pop = stack.pop();
            System.out.println("取出的结果:"+pop);
        }
    }
}
```

通过测试我们发现我们基于单链表实现的链式栈跟我们用双向链表实现的链式栈在功能上都是一样的，但是基于单链表实现的链式栈很明显更节约内存空间。

以上内容就是我们自己手动编程来实现的栈，这块内容希望大家自己能够手动编码实现，这对大家思维能力的训练和代码能力的提升都会有很大的帮助。

5.3：栈的应用

5.3.1：Stack 源码分析

首先写一小段测试代码

```
public class JdkStack {
    public static void main(String[] args) {
        List list;
        //创建栈对象
        Stack stack = new Stack();
        //数据入栈
        stack.push("itcast");
        stack.push("itheima");
        //数据出栈
        Object item = stack.pop();
        System.out.println(item);
    }
}
```

(1)、栈的创建和元素入栈

我们先查看栈的构造函数如下：

```
public class Stack<E> extends Vector<E> {
    /**
     * Creates an empty Stack.
     */
    public Stack() {
    }
}
```

是空实现，并没有其他操作，接下来断点调试元素入栈的 push 方法，push 方法的实现如下：

```
// 元素入栈
public E push(E item) {
        addElement(item);//调用父类的addElement 方法
        return item;
    }
```

查看 Stack 的父类 Vectory，将相关的属性和方法罗列如下：

```
public class Vector<E> extends AbstractList<E> implements List<E>, Rand
omAccess, Cloneable, java.io.Serializable
{
      protected Object[] elementData;
      protected int elementCount;
      protected int capacityIncrement;
      //添加元素到数组
       public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);//确保容量
        elementData[elementCount++] = obj;
    }
    private void ensureCapacityHelper(int minCapacity) {
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)//数组长度不够
            grow(minCapacity);//扩容
    }
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacit
```

```
y);
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        elementData = Arrays.copyOf(elementData, newCapacity);
    }

    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }
}
```

断点调试结果如图：

从调试结果可知，Stack 底层的实现思路和我们自己手动实现的基于数组的顺序栈的思路是一致的。

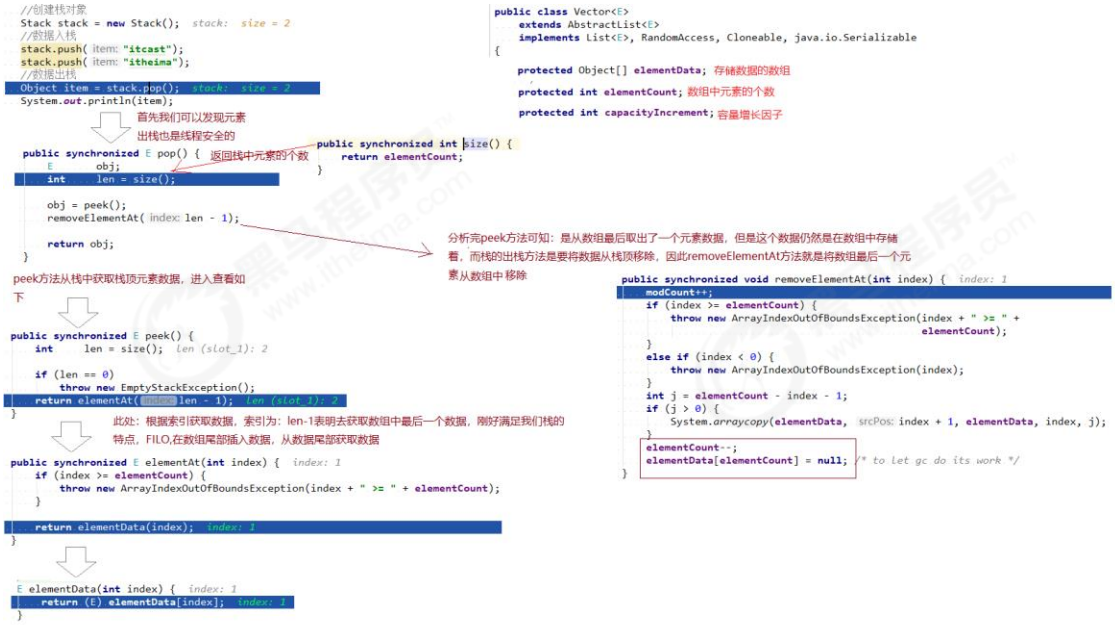
(2)、元素出栈

分析完元素的入栈操作之后我们接着来分析元素的出栈操作，首先我们罗列出元素出栈的相关方法如下：

```
public synchronized E pop() {
        E       obj;
        int     len = size();//调用父类Vector 的方法
        obj = peek();
        removeElementAt(len - 1);//调用的是父类的方法
        return obj;
}
public synchronized E peek() {
        int     len = size();

        if (len == 0)
            throw new EmptyStackException();
        return elementAt(len - 1);//调用的是父类的方法
    }
```

断点调试结果如下图：



从调试结果可知，元素出栈就是将数组的最后元素取出并将数组最后一个位置置为null，相当于将元素从数组中移除并返回。

5.3.2： 思考

学习完“栈”这个数据结构之后我们要思考一个问题，我们能用栈来做什么呢？在实际的编程过程中哪些地方能用得到栈这个数据结构呢？

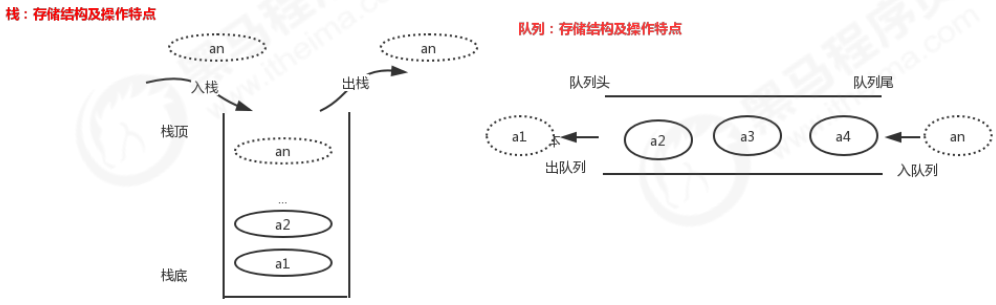
众所周知，浏览器有一个前进和后退的功能，当我们依次访问完一连串页面 A-B-C 之后，点击浏览器的后退按钮，就可以查看之前浏览过的 B 页面，在点击后退可以浏览之前的 A 页面，然后点击前进按钮又可以查看 B 页面，再次点击可以查看 C 页面，但是如果你回退到页面 B 后又点击了新的页面 D，那就再无法通过回退和前进功能查看页面 C 了，假设你是 Chrome 浏览器的研发工程师，你如何来实现这个功能？

至此，栈这部分的知识我们就学习完了，接下来我们进入队列的学习。

6. 队列

6.1：概念

队列的概念非常容易理解，我们拿日常生活中的一个场景来举例说明，我们去车站的窗口买票，那就要排队，那先来的人就先买，后到的人就后买，先来的人排到队头，后来的人排在队尾，不允许插队，**先进先出，这就是典型的队列。** 队列先进先出的特点英文表示为：First In First Out 即 **FIFO**,为了更好的理解队列这种数据结构，我们以一幅图的形式来表示，并且我们将队列的特点和栈进行比较，如下：



从图中我们可以发现，**队列和栈一样都属于一种操作受限的线性表**，栈只允许在一端进行操作，分别是入栈 push()和出栈 pop()，而**队列跟栈很相似，支持的操作也有限，最基本的两个操作一个叫入队列 enqueue()，将数据插入到队列尾部，另一个叫出队列 dequeue()，从队列头部取出一个数据。**

队列的概念很好理解，基本操作也很容易掌握。作为一种非常基础的数据结构，队列的应用也非常广泛，特别是一些具有某些额外特性的队列，比如循环队列、阻塞队列、并发队列。它们在很多偏底层系统、框架、中间件的开发中，起着关键性的作用。比如高性能队列 Disruptor【dɪs'rʌptər】、Linux 环形缓存都用到了循环并发队列，在 java 中，concurrent 并发包利用 ArrayBlockingQueue 来实现公平锁等。

6.2：常见队列及实现

在这节中我们要介绍几种常见的队列：顺序队列，链式队列，循环队列，阻塞队列，并发队列等，并且会针对其中某些队列作手动实现

6.2.1：顺序队列的实现

跟栈一样，队列可以用数组来实现，也可以用链表来实现。用数组实现的栈叫作顺序栈，用链表实
现的栈叫作链式栈。同样，**用数组实现的队列叫作顺序队列，用链表实现的队列叫作链式队列**。

我们先来看基于数组实现的顺序队列：

```
public class ArrayQueue {
    // 存储数据的数组
    private Object[] elements;
    //队列大小
    private int size;
    // 默认队列容量
    private int DEFAULT_CAPACITY = 10;
    // 队列头指针
    private int head;
    // 队列尾指针
    private int tail;

    private int MAX_ARRAY_SIZE  = Integer.MAX_VALUE-8;

    /**
     * 默认构造函数 初始化大小为 10 的队列
     */
    public ArrayQueue(){
        elements = new Object[DEFAULT_CAPACITY];
        initPointer(0,0);
    }

    /**
     * 通过传入的容量大小创建队列
     * @param capacity
     */
    public ArrayQueue(int capacity){
        elements = new Object[capacity];
        initPointer(0,0);
    }

    /**
```

```
     * 初始化队列头尾指针
     * @param head
     * @param tail
     */
    private void initPointer(int head,int tail){
        this.head = head;
        this.tail = tail;
    }

    /**
     * 元素入队列
     * @param element
     * @return
     */
    public boolean enqueue(Object element){
        ensureCapacityHelper();
        elements[tail++] = element;//在尾指针处存入元素且尾指针后移
        size++;//队列元素个数加1
        return true;
    }

    private void ensureCapacityHelper() {
        if(tail==elements.length){//尾指针已越过数组尾端
            //判断队列是否已满 即判断数组中是否还有可用存储空间
            //
            if(size<elements.length){
            if(head==0){
                //扩容
                grow(elements.length);
            }else{
                //进行数据搬移操作 将数组中的数据依次向前挪动直至顶部
                for(int i= head;i<tail;i++){
                    elements[i-head]=elements[i];
                }
                //数据搬移完后重新初始化头尾指针
                initPointer(0,tail-head);
            }
        }
    }
    /**
     * 扩容
     * @param oldCapacity 原始容量
     */
    private void grow(int oldCapacity) {
        int newCapacity = oldCapacity+(oldCapacity>>1);
        if(newCapacity-oldCapacity<0){
```

```
            newCapacity = DEFAULT_CAPACITY;
        }
        if(newCapacity-MAX_ARRAY_SIZE>0){
            newCapacity = hugeCapacity(newCapacity);
        }
        elements = Arrays.copyOf(elements,newCapacity);
    }
    private int hugeCapacity(int newCapacity) {
        return (newCapacity>MAX_ARRAY_SIZE)? Integer.MAX_VALUE:newCapac
ity;
    }

    /**
     * 出队列
     * @return
     */
    public Object dequeue(){
        if(head==tail){
            return null;//队列中没有数据
        }
        Object obj=elements[head++];//取出队列头的元素且头指针后移
        size--;//队列中元素个数减1
        return obj;
    }

    /**
     * 获取队列元素个数
     * @return
     */
    public int getSize() {
        return size;
    }
}
```

我们先来编写一个简单的测试类测试我们写好的代码：

```
public class TestArrayQueue {

    public static void main(String[] args) {
        ArrayQueue queue = new ArrayQueue(4);
        //入队列
        queue.enqueue("itcast1");
        queue.enqueue("itcast2");
        queue.enqueue("itcast3");
        queue.enqueue("itcast4");
        //此时入队列应该走扩容的逻辑
        queue.enqueue("itcast5");
```

```
            queue.enqueue("itcast6");
            //出队列
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            //此时入队列应该走数据搬移逻辑
            queue.enqueue("itcast7");
            //出队列
            System.out.println(queue.dequeue());
            //入队列
            queue.enqueue("itcast8");
            //出队列
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            //入队列
            queue.enqueue("itcat9");
            queue.enqueue("itcat10");
            queue.enqueue("itcat11");
            queue.enqueue("itcat12");
            //出队列
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
    }
}
```

先来查看控制台输出的运行结果：

```
itcast1
itcast2
itcast3
itcast4
itcast5
itcast6
itcast7
itcast8
null
itcat9
itcat10
```

通过控制台的输出结果我们可以发现基于数组的顺序队列确实满足先进先出的特点，接下来我们就一起来分析一下实现原理：

对于栈来说，我们只需要一个栈顶指针就可以了。但是队列需要两个指针(数组下标)：一个是 head 指针，指向队头；一个是 tail 指针，指向队尾。每一次元素入队

列操作，我们的 tail 指针就需要向后移动一位；每一次出队列操作，我们的 head 指针就要向后移动一位。

但是当 tail 指针移动到数组末尾之后，由于入队列操作都是从数组尾部存入数据，那此时数组尾部已不能在插入数据了，我们就要根据情况采取不同的措施，第一种情况是数组中已经没有位置可以存储数据了那就要对数组进行扩容，第二种情况是数组中还有位置可以存储数据，只不过是数组尾端已不能再插入数据了，那此时就要将数组中的数据依次向前挪动，将数组尾部的位置空出来供入队列操作，详细的操作流程见下图：

我们主要看元素的入队列操作

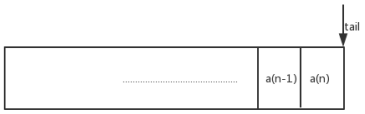
```
public boolean enqueue(Object element){
    ensureCapacityHelper();
    elements[tail++] = element;//在尾指针处存入元素且尾指针后移
    size++;//队列元素个数加1
    return true;
}
```

将元素存储在尾指针处，然后尾指针后移，队列中元素个数加1

我们主要看ensureCapacityHelper方法

```
private void ensureCapacityHelper() {
    if(tail==elements.length){//尾指针已越过数组尾端
        //判断队列是否已满 即判断数组中是否还有可用存储空间
        //if(size<elements.length){
        if(head==0){
            //扩容
            grow(elements.length);
        }else{
            //进行数据搬移操作 将数组中的数据依次向前挪动直至顶部
            for(int i= head;i<tail;i++){
                elements[i-head]=elements[i];
            }
            //数据搬移完后重新初始化头尾指针
            initPointer( head: 0, tail: tail-head);
        }
    }
}
```

判断条件：tail==elements.length）即尾指针已到达数组最右端，如图所示：



此时：我们无法再向数组尾部插入元素了，在这样一个情况下又分为两种情况：

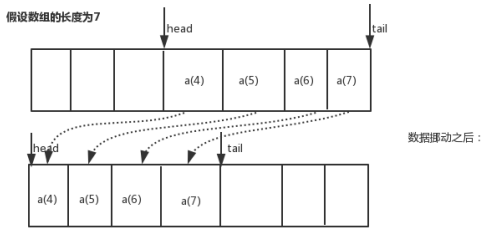
1：数组中还有位置可以存储元素

2：数组中已经没有位置可以存储元素了

所以接下来有一个判断：if(head==0){

这属于第二种情况：数组中已经没有位置可以存储元素了，那接下来就要对队列进行扩容，执行 grow(elements.length); 底层的扩容逻辑和之前的扩容逻辑和ArrayList的扩容逻辑一样这里就不再赘述了

如果判断条件不满足：则属于第一种情况，数组中还有位置可以存储，只不过数组末尾不能在存储数据了而已，那此时我们需要做的事情就是将数组中的数据依次向前挪动，将数组尾部空出来可以继续插入元素，如下图所示：




```
public class ArrayQueue {
    // 存储数据的数组
    private Object[] elements;
    //队列大小
    private int size;
    // 默认队列容量
    private int DEFAULT_CAPACITY = 10;
    // 队列头指针
    private int head;
    // 队列尾指针
    private int tail;

    private int MAX_ARRAY_SIZE  = Integer.MAX_VALUE-8;

    /**
     * 默认构造函数 初始化大小为10的队列
     */
    public ArrayQueue(){
        elements = new Object[DEFAULT_CAPACITY];
        initPointer( head: 0, tail: 0);
    }
    /**
     * 通过传入的容量大小创建队列
     * @param capacity
     */
    public ArrayQueue(int capacity){
        elements = new Object[capacity];
        initPointer( head: 0, tail: 0);
    }

    /**
     * 初始化队列头尾指针
     * @param head
     * @param tail
     */
    private void initPointer(int head,int tail){
        this.head = head;
        this.tail = tail;
    }
```

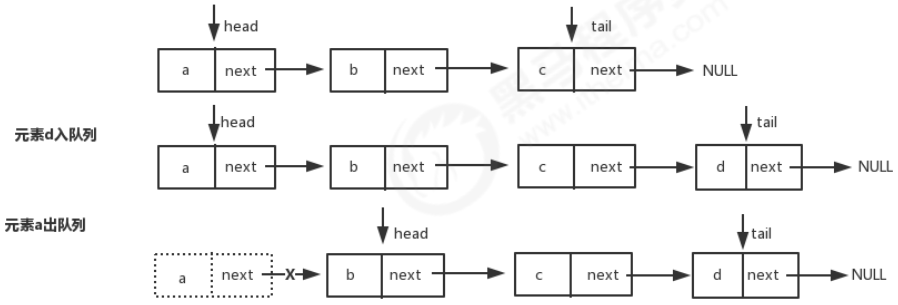
队列的类结构视图：通过默认的构造函数创建队列会初始化一个大小为10的数组存储元素，当然也可以在创建队列的时候指定数组容量大小

那对于元素出队列来说，只要将 head 指针处的元素从数组中取出来，然后 head 指针后移即可，当然了如果 head 指针和 tail 指针如果重合了则表明队列中已经没有元素数据了此时直接返回 null 即可。

6.2.2：链式队列的实现

在上一小节中我们使用数组实现了一个队列，那在这一小节中我们使用链表来实现一个队列，通过之前的学习我们已经知道用链表实现的队列叫链式队列。

基于链表的实现，我们同样需要两个指针：head 指针和 tail 指针。它们分别指向链表的第一个结点和最后一个结点。如图所示，入队时，tail->next= new_node, tail = tail->next；出队时，head= head->nex，如下图所示：



下面进行代码实现：

```
public class LinkedListQueue {
    //队列元素个数
    private int size;
    //头节点
    private Node head;
    //尾节点
    private Node tail;

    public LinkedListQueue(){
        this.head = null;
        this.tail = null;
    }

    /**
     * 入队列
     * @param data
     * @return
     */
    public boolean enqueue(Object data){
        Node newNode = new Node(data,null);
        if(tail == null){
            tail = newNode;
            head = newNode;
        }else {
            tail.next = newNode;
            tail = newNode;
```

```
            }
            size++;
            return true;
        }

        /**
         * 出队列
         * @return
         */
        public Object dequeue(){
            if(head==null){
                return null;
            }
            Object data = head.data;
            head = head.next;
            if(head==null){
                tail = null;
            }
            size--;
            return data;
        }


        private static class Node{
            //节点数据
            private Object data;
            //后继节点
            private Node next;

            public Node(Object data,Node next){
                this.data = data;
                this.next = next;
            }
        }

        public int getSize() {
            return size;
        }
    }
```

编写一个测试类进行测试：

```
public class TestLinkedListQueue {
    public static void main(String[] args) {
        LinkedListQueue queue = new LinkedListQueue();
        System.out.println(queue.dequeue());
        queue.enqueue("itcast1");
```

```
            queue.enqueue("itcast2");
            queue.enqueue("itcast3");
            queue.enqueue("itcast4");
            queue.enqueue("itcast5");
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            queue.enqueue("itcast6");
            queue.enqueue("itcast7");
            queue.enqueue("itcast8");
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
            System.out.println(queue.dequeue());
        }
}
```

运行结果如下：

```
null
itcast1
itcast2
itcast3
itcast4
itcast5
null
itcast6
itcast7
itcast8
```

通过控制台的输出结果我们可以发现基于链表的链式队列也满足先进先出的特点。

6.2.3：独立思考

在这个实现中我们发现，循环队列满的时候数组中其实还是有一个位置可以进行数据的存储的，也就是说当下的循环队列的实现方式会浪费一个存储空间，那有没有什么更好的解决方案呢？这个问题留做课后思考题大家请先自行分析并实现一下。

6.3：队列的应用

上一章节中我们手动实现了顺序队列，链式队列及循环队列，那在 java 中是否有已经实现好的队列呢？在回答这个问题之前，我们先来聊一聊在实际的软件研发中哪些地方能够用到队列这种数据结构？

6.3.1：队列的应用场景

首先我们得回顾本章开始所讲到的队列的特点：**先进先出(FIFO)**，一般情况下，如果是对一些及时消息的处理，并且处理时间很短的情况下是不需要队列的，直接阻塞式的方法调用就可以了。但是如果在消息处理的时候特别费时间，这个时候如果有新消息来了，就只能处于阻塞状态，造成用户等待。这个时候便需要引入队列了。当接收到消息后，先把消息放入队列中，然后再用新的线程进行处理，这个时候就不会有消息阻塞了。所以队列用来存放等待处理元素的集合，这种场景一般用于缓冲、并发访问，及时消息通信，分布式消息队列等。

6.3.2：课后思考

我们知道，CPU 资源是有限的，任务的处理速度与线程个数并不是线性正相关。相反，过多的线程反而会导致 CPU 频繁切换，处理性能下降。所以，线程池的大小一般都是综合考虑要处理任务的特点和硬件环境，来事先设置的。当我们向固定大小的线程池中请求一个线程时，如果线程池中没有空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢？

7. 今日总结及作业安排

7.1：总结

在今天的课程中我们主要学习了如下的几个内容：

- 数组：

  数组是一种线性表数据结构，它用一组连续的内存空间，来存储一组具有相同类型的数据。数组通过下标来访问元素，数组访问时可以通过寻址公式快速定位元素位置，但是插入和删除操作为了保证数组的连续性要进行很多数据的搬移操作

- 链表

  链表（Linked list）是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。链表在进行元素的插入和删除操作时的效率高，但是链表在进行元素获取时的效率低，而链表又分为单链表，双向链表，循环链表，双向循环链表等

- 栈

栈满足先进后出，后进先出即 LIFO，属于一种操作受限的线性表，只允许在一端进行元素的插入和删除，分别是入栈和出栈，时间复杂度都是 O(1)，栈可以基于数组实现也可以基于链表实现，分别是顺序栈和链式栈。

- 队列

  队列满足先进先出的特点即 FIFO，队列跟栈一样都属于操作受限的线性表，只不过队列是只允许在队列头获取元素，在队列尾插入元素，分别叫入队列和出队列，当然队列也可以基于数组和链表来实现，分别叫顺序队列和链式队列，在队列中还有一种循环队列可以解决顺序队列需要频繁搬移数据的问题。

7.2：作业

1：手动分析一遍 ArrayList 的源码，比如 set 方法等等

2：手动分析一遍 LinkedList 的源码

3：自主完成单链表反转的面试题

4：手动分析一遍 Stack 的源码

5：自主实现基于数组且支持动态扩容的顺序栈

6：自主实现基于链表的链式栈

7：自主实现基于数组的顺序队列

8：自主实现基于链表的链式队列

9：自主实现一个基于数组的循环队列

10：完成课程中留下的课后思考题