Queue

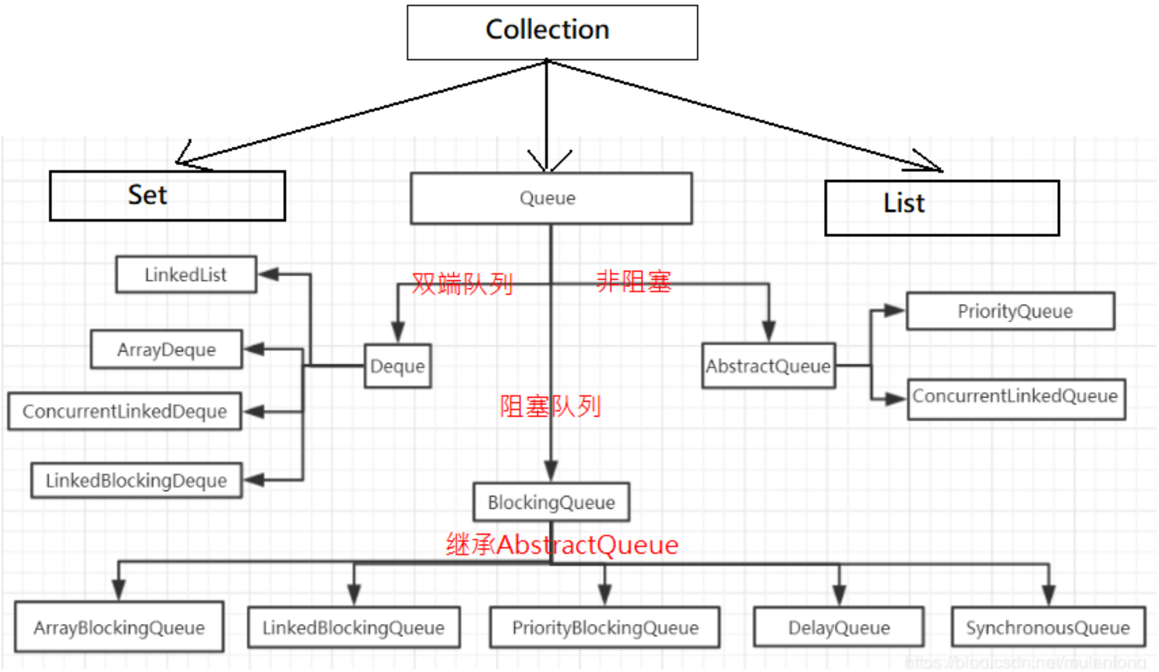
Queue： 基本上，一个队列就是一个先入先出（FIFO）的数据结构

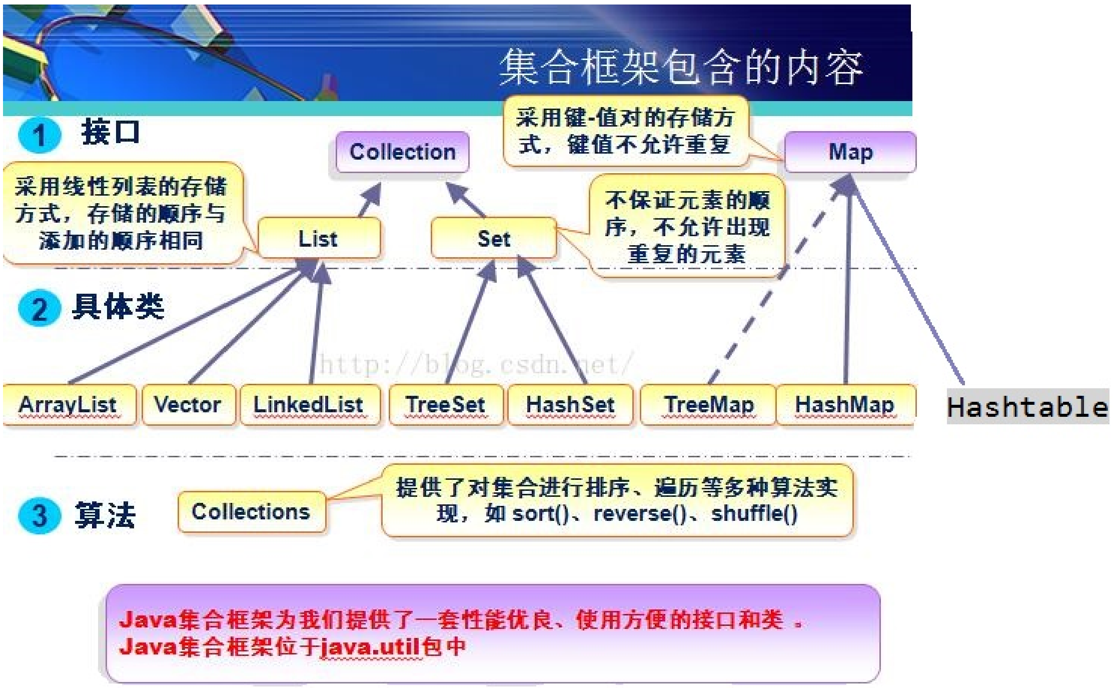
Queue接口与List、Set同一级别，都是继承了Collection接口。LinkedList实现了Deque接 口。

下表显示了jdk1.5中的阻塞队列的操作：

add	增加一个元素	如果队列已满，则抛出一个IIlegalSlabEepeplian异常
remove	移除并返回队列头部的元素	如果队列为空，则抛出一个NoSuchElementException异常
element	返回队列头部的元素	如果队列为空，则抛出一个NoSuchElementException异常
offer	添加一个元素并返回true	如果队列已满，则返回false
poll	移除并返问队列头部的元素	如果队列为空，则返回null
peek	返回队列头部的元素	如果队列为空，则返回null
put	添加一个元素	如果队列满，则阻塞
take	移除并返回队列头部的元素	如果队列为空，则阻塞

remove、element、offer 、poll、peek 其实是属于Queue接口。





集合框架包含的内容

1 接口

采用线性列表的存储方式，存储的顺序与添加的顺序相同

Collection

采用键-值对的存储方式，键值不允许重复

Map

List

Set

不保证元素的顺序，不允许出现重复的元素

2 具体类

ArrayList Vector LinkedList TreeSet HashSet TreeMap HashMap Hashtable

http://blog.csdn.net/

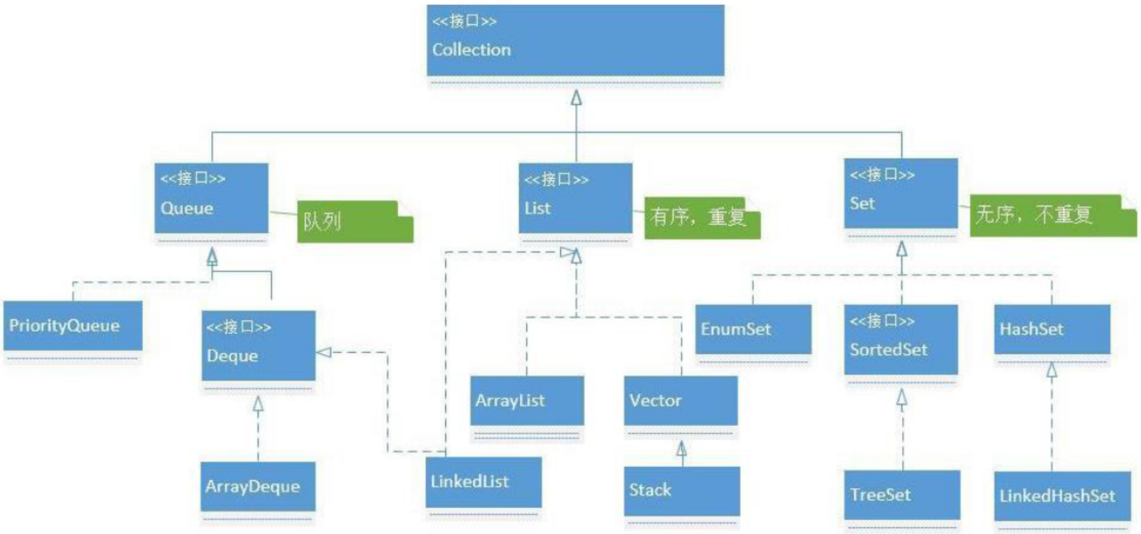
3 算法

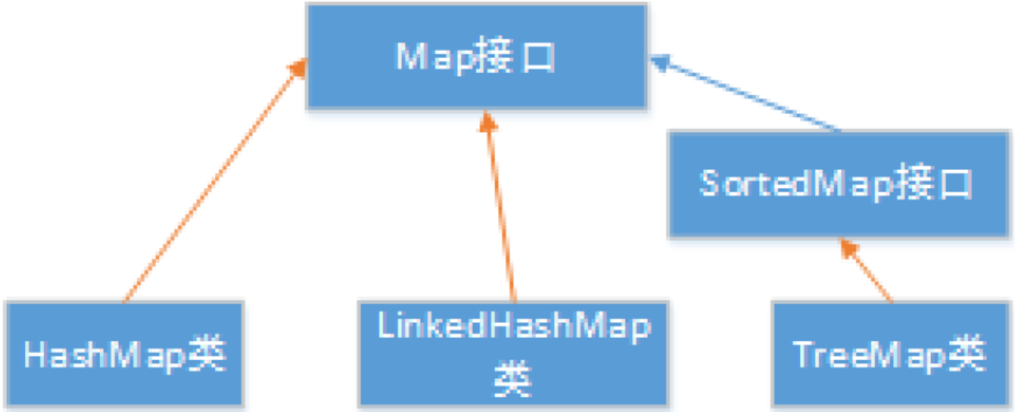
Collections

提供了对集合进行排序、遍历等多种算法实现，如 sort()、reverse()、shuffle()

Java集合框架为我们提供了一套性能优良、使用方便的接口和类 。
Java集合框架位于java.util包中

Collection:





二叉排序树(BST)、二叉搜索树

相关术语

①结点：包含一个数据元素及若干指向子树分支的信息 [5] 。

②结点的度：一个结点拥有子树的数目称为结点的度 [5] 。

③叶子结点：也称为终端结点，没有子树的结点或者度为零的结点 [5] 。

④分支结点：也称为非终端结点，度不为零的结点称为非终端结点 [5] 。

⑤树的度：树中所有结点的度的最大值 [5] 。

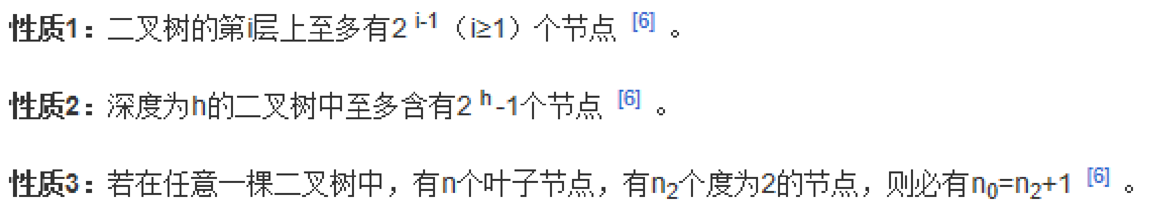
⑥结点的层次：从根结点开始，假设根结点为第1层，根结点的子节点为第2层，依此类推，如果某一个结点位于第L层，则其子节点位于第L+1层 [5] 。

⑦树的深度：也称为树的高度，树中所有结点的层次最大值称为树的深度 [5] 。

⑧有序树：如果树中各棵子树的次序是有先后次序，则称该树为有序树 [5] 。

⑨无序树：如果树中各棵子树的次序没有先后次序，则称该树为无序树 [5] 。

⑩森林：由m（m≥0）棵互不相交的树构成一片森林。如果把一棵非空的树的根结点删除，则该树就变成了一片森林，森林中的树由原来根结点的各棵子树构成 [5] 。



性质1：二叉树的第i层上至多有2^{i-1}（i≥1）个节点 [6] 。

性质2：深度为h的二叉树中至多含有2^{h}-1个节点 [6] 。

性质3：若在任意一棵二叉树中，有n个叶子节点，有n_2个度为2的节点，则必有$n_0=n_2+1$ [6] 。

```
public class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    public TreeNode(int value) {
        this.value = value;
    }
}
```

树:

```
public class BinaryTree {
    TreeNode root;

    /**
     * 添加数据
     *
     * @param data
     */
    public void add(int data) {
        TreeNode node = new TreeNode(data);
        if (this.root == null) {
            this.root = node;
            return;
        }
        TreeNode temp = this.root;
        while (true) {
            if (data < temp.value) {
                TreeNode parent = temp;
                temp = temp.left;
                if (temp == null) {
                    parent.left = node;
                    return;
                }
            } else {
                TreeNode parent = temp;
                temp = temp.right;
                if (temp == null) {
                    parent.right = node;
                    return;
                }
            }
        }
    }

    /**
     * 先序遍历
     */
    public void frontShow() {
        if (root != null) {
            System.out.println("先序遍历:");
            this.frotOrder(root);
            System.out.println();
            this.stackFrotOrder(root);
            System.out.println();
        }
    }

    /**
     * 中序遍历
     */
    public void middleShow() {
        if (root != null) {
            System.out.println("中序遍历:");
            this.middleOrder(root);
            System.out.println();
            this.stackMiddleOrder(root);
            System.out.println();
        }
    }

    /**
     * 后续遍历
     */
    public void behindShow() {
        if (root != null) {
            System.out.println("后序遍历:");
            this.behindOrder(root);
            System.out.println();
            this.stackBehindOrder(root);
            System.out.println();
        }
    }

    //先序遍历(递归法)
    public void frotOrder(TreeNode node) {
        if (node != null) { //前序遍历,"中左右"
            System.out.print(node.getValue() + " ");
            frotOrder(node.left);
            frotOrder(node.right);
        }
    }

    //先序遍历(用栈)
    public void stackFrotOrder(TreeNode root) {
        Stack stack = new Stack();  //用栈进行前序遍历
        if (root != null) {
            stack.push(root);
        }
        while (!stack.isEmpty()) {
            TreeNode node = (TreeNode) stack.pop();
            System.out.print(node.getValue() + " ");
            if (node.right != null) stack.push(node.right);   //先右后左
            if (node.left != null) stack.push(node.left);
        }
    }


    //中序遍历(递归法)
    public void middleOrder(TreeNode node) {
        if (node != null) {//中序遍历，"左中右"(从第1个没有左孩子的节点开始遍历)
            middleOrder(node.left);
            System.out.print(node.getValue() + " ");
            middleOrder(node.right);
        }
    }

    //中序遍历(用栈)
    public void stackMiddleOrder(TreeNode root) {
        Stack stack = new Stack();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            if (!stack.isEmpty()) {
                TreeNode treeNode = (TreeNode) stack.pop();
                System.out.print(treeNode.getValue() + " ");
                root = treeNode.right;
            }
        }
    }

    //后序遍历(递归法)
    public void behindOrder(TreeNode node) {
        if (node != null) {//后续遍历，"左右中"
            behindOrder(node.left);
            behindOrder(node.right);
            System.out.print(node.getValue() + " ");
        }
    }

    //后序遍历(用栈)
    public void stackBehindOrder(TreeNode root) {
        Stack input = new Stack();
        Stack output = new Stack(); // 中间栈存储逆后序遍历结果
        while (root!=null || !input.isEmpty()){
            if (root != null) {
                output.push(root);
                input.push(root);
                root = root.right;
            } else {
                root = (TreeNode) input.pop();
                root = root.left;
            }
        }
        while (!output.isEmpty()) {
            TreeNode treeNode = (TreeNode) output.pop();
            System.out.print(treeNode.getValue() + " ");
        }
    }

    /**
     * 查找节点
     * @return
     */
    public TreeNode find(int value) {
        TreeNode current = root;
        while (current.getValue() != value) {
            if (value < current.getValue()) {
                current = current.left;
            } else {
                current = current.right;
            }
            if (current == null) return null;
        }
        return current;
    }

    /**
     * 二叉树的深度
     */
    public void showTreeDeapth() {
        System.out.println(this.treeDeapth(root));
    }

    public int treeDeapth(TreeNode node) {
        if (node == null) return 0;
        int left = treeDeapth(node.left);
        int right = treeDeapth(node.right);
        return left > right? left + 1 : right + 1;
    }
}
```

test:

```
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        int[] arr = new int[]{7, 3, 10, 12, 5, 1, 9};
        for (int i : arr) {
            tree.add(i);
        }
//        tree.frontShow();
//        tree.middleShow();
//        tree.behindShow();
        TreeNode node = tree.find(3);
        System.out.println(node.getValue());
    }
}

/*
7, 3, 10, 12, 5, 1, 9
先序遍历:
7 3 1 5 10 9 12
中序遍历:
1 3 5 7 9 10 12
后序遍历:
1 5 3 9 12 10 7
*/
```

ArrayLIst

int newCapacity = oldCapacity + (oldCapacity >> 1),所以 ArrayList 每次扩容之后容量都会变为原来的 1.5 倍左右（oldCapacity为偶数就是1.5倍，否则是1.5倍左右）！ 奇偶不同，比如 ：10+10/2 = 15, 33+33/2=49。如果是奇数的话会丢掉小数.

```
public class MyArrayList {
    private static final int DEFAULT_CAPACITY = 5;//默认容量
    private int capacity;//容量
    private int size = 0;//线性表的元素总数
    private int[] data;

    public MyArrayList(int capacity) {
        this.capacity = capacity;
        data = new int[capacity];
    }

    public MyArrayList() {
        data = new int[DEFAULT_CAPACITY];
    }

    public void add(int temp) {
        //判断是否需要扩容
        isDilatation();
        data[size++] = temp;
    }

    public void show() {
        Arrays.stream(data).forEach(s -> System.out.print(s + " "));
    }

    private void isDilatation() {
        if (size + 1 > capacity) {
            capacity *= 2;
            //将原数组复制到新数组
            //原数组没有新数据长，新数组后面的值默认填充为0
            int[] tempData = Arrays.copyOf(data, capacity);
            data = tempData;
        }
    }
}
```

SingleLinkedList

```
public class SingleLinkedList {
    private Node head;
    private Node tail;
    private int size = 0;
    class Node{
        int data;
        Node next;

        public Node(int data) {
            this.data = data;
        }
    }

    public SingleLinkedList() {

    }

    /**
     * 添加元素
     * @param data
     */
    public void add(int data) {
        Node node = new Node(data);
        if (size == 0) {
            head = node;
            tail = head;
            size++;
        } else {
            tail.next = node;
            tail = tail.next;
            size++;
        }
    }


    /**
     * 将单链表反向
     */
    public void reverse() {
        Node reverseHead = head;
        Node node = head; //当前节点
        Node prev = null; //假装为它的上个节点
        while (node != null) {
            Node next = node.next;
            if (next == null) {
                reverseHead = node;
            }
            node.next = prev;  //让当前节点指向上一个节点
            prev = node; //将上一个节点的引用切换到当前节点
            node = next; //遍历节点
        }
        head = reverseHead;
    }


    public void show() {
        Node temp = head;
        while (temp != null) {
            System.out.print(" " + temp.data);
            temp = temp.next;
        }
    }
}
```

DoubleLinkedList

```
public class MyArrayList {
    transient int size = 0;
    transient Node first;
    transient Node last;

    private static class Node{ //构造节点
        int item;
        Node prev;
        Node next;
        public Node(int item) {
            this.item = item;
        }
    }

    //判断是否为空
    public boolean isEmpty() {
        if (first == null) return true;
        return false;
    }

    //添加元素
    public void add(int data) {
        Node node = new Node(data);
        if (isEmpty()) {
            first = node;
            first.prev = null;
            last = node;
        } else {
            node.prev = last;
            last.next = node;
            last = last.next;
        }
        size++;
    }
    //移除最后一个元素并返回
    public int removeLast() {
        if (isEmpty()) {
            throw new RuntimeException("链表为空");
        }
        Node temp = last;
        last = last.prev;
        size--;
        return temp.item;
    }

    //根据索引查找元素
    public int findByIndex(int index) {
        if (index > size) {
            throw new RuntimeException("索引越界");
        }
        Node temp = first;
        for (int i = 0; i < index - 1; i++) {
            temp = temp.next;
        }
        return temp.item;
    }

    //输出链表
    public void show() {
        Node temp = first;
        int index = 0;
        while (index < size) {
            System.out.print(temp.item + " ");
            temp = temp.next;
            index++;
        }
    }

}
```

Stack

```
public class MyStack {
    transient Node head; //栈顶
    transient int size = 0;

    private static class Node {
        int data;
        Node next;

        public Node(int data) {
            this.data = data;
        }
    }

    //判断是否为空
    public boolean isEmpty() {
        return size == 0;
    }

    //入栈
    public void push(int data) {
        Node node = new Node(data);
        if (isEmpty()) {
            head = node;
            head.next = null;
            size++;
        } else {
            node.next = head;
            head = node;
            size++;
        }
    }

    //出栈
    public int pop() {
        if (!isEmpty()) {
            Node temp = head;
            head = head.next;
            size--;
            return temp.data;
        }
        throw new RuntimeException("栈已空不能再出栈");
    }
}
```