数据结构、算法、设计模式

一、算法性能分析

概论

- 时间复杂度
- 空间复杂度

时间复杂度

- 规模: 不同量级有不同的速度，比如水 vs 水杯: 水
- 测试环境: 在不同测试环境，速度也不同，比如手机 vs 电脑: 电脑

大 O 表示法

```
public class Tmp {
    public void tmp(int n) {
        int add = 0;
        for (int i = 0; i < n; i++) {
            add += 1;
        }
    }
}
```

运行时间：T(n) = (2n + 1) * unit

T(n) = O(f(n)) , O表示 T(n) 与 f(n) 成正比

O 表示渐近时间复杂度

表示代码执行时间随数据规模增长的变化趋势

> 当 n 很大时,低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略．就可以记为：T(n) = O(n)； T(n) = O(n2)。

只关注循环次数多的代码

```
public class Tmp {
    public void tmp(int n) {
        int add = 0;
        for (int i = 0; i < n; i++) {
            add += 1;
        }
    }
}
```

O(n)

选大量级

```
public class Tmp {
    public void tmp(int n) {
        int add = 0;
        for (int i = 0; i < 999; i++) {
            System.out.println(123);
        }

        for (int i = 0; i < n; i++) {
            System.out.println(1);
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.println(2);
            }
        }
    }
}
```

O(n2)

嵌套循环要乘积

```
public class Tmp {
    public void tmp(int n) {
        int add = 0;

        for (int i = 0; i < n; i++) {
            this.a(i);
        }
    }

    public void a(int n) {
        for (int i = 0; i < n; i++) {
            System.out.println(3);
        }
    }
}
```

O(n2)

常见复杂度分析

- 非多项式量级（过于低效）：O(2n) 和 O(n!)。
- 多项式量级:O(1), O(logn), O(n), O(nlogn), O(nk)

O(1)

```
public class Tmp {
    public void tmp(int n) {
        int a = 2;
        int b = 3;
        int c = 4;
    }
}
```

O(logn)

```
public class Tmp {
    public void tmp(int n) {
        int i = 1;

        while (i < n) {
            i = i * 2;
        }
    }
}
```

i = 20,21, 22, 23...2x

退出循环的条件是 : 2x = n ,即 x = log2n，时间复杂度为 O(log2n)

```
public class Tmp {
    public void tmp(int n) {
        int i = 1;

        while (i < n) {
            i = i * 3;
        }
    }
}
```

log3n 就等于 log32 * log3n，所以 O(log3n) = O(C * log2n)，其中 C=log32 是一个常量。基于我们前面的一个理论：在采用大 O 标记复杂度的时候，可以忽略系数，即 O(Cf(n)) = O(f(n))。所以，O(log2n) 就等于 O(log3n)。因此，在对数阶时间复杂度的表示方法里，我们忽略对数的“底”，统一表示为 O(logn)

O(m+n)

```java
public class Tmp {
    public void tmp(int n, int m) {
        for (int i = 0; i < n; i++) {
            System.out.println(1);
        }

        for (int i = 0; i < m; i++) {
            System.out.println(2);
        }
    }
}
```

空间复杂度

- 渐进时间复杂度：表示算法的执行时间与数据规模之间的增长关系。
- 渐进空间复杂度：表示算法的存储空间与数据规模之间的增长关系。

```java
public class Tmp {
    public void tmp(int n) {
        int[] a = new int[n];

        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

空间复杂度是： O(n)

二、数组与列表

什么是数组?

数组是**线性表**数据结构。用**连续的内存空间**存储**相同类型**的数据。

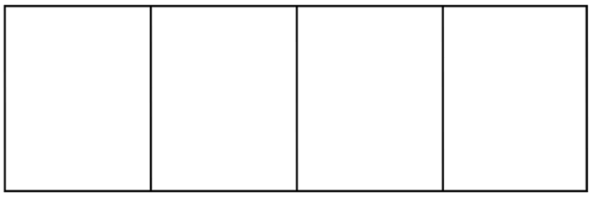
- 线性表：线性表是数据排成一条线一样的结构。每个线性表上的数据最多只有前和后两个方向。包括：数组，链表、队列、栈。
- 非线性表：数据之间并不是简单的前后关系。 包括：二叉树、堆、图等。
- 连续的内存空间和相同类型的数据：使数组支持“随机访问”。但在数组中删除、插入数据时，需要做大量的数据搬移工作。

数组如何实现随机访问?

C 语言代码： `int[] tmp = new int[4]`，这个数组在内存中连续放置：

插入操作

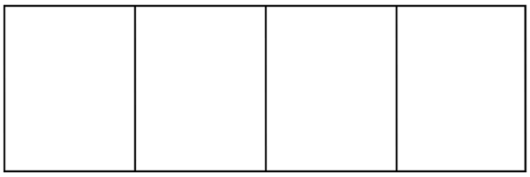
- 在数组末尾插入元素，不需要移动数据，时间复杂度为 O(1)。
- 在数组开头插入元素，那所有的数据都需要依次往后移动一位，所以最坏时间复杂度是 O(n)。
- 假设每个位置插入元素概率相同，平均情况时间复杂度为 (1+2+...n)/n=O(n)。



> 如果元素无序，可直接换位：

删除操作

- 删除数组末尾数据：则最好情况时间复杂度为 O(1)；
- 删除开头的数据：则最坏情况时间复杂度为 O(n)；
- 平均情况时间复杂度也为 O(n)。



> 预删除思想：JVM 标记清除垃圾回收算法

改进数组

编程语言封装了数组，比如 Java 的 ArrayList， Python 的 List ，可实现自动扩容，多种数据类型组合。

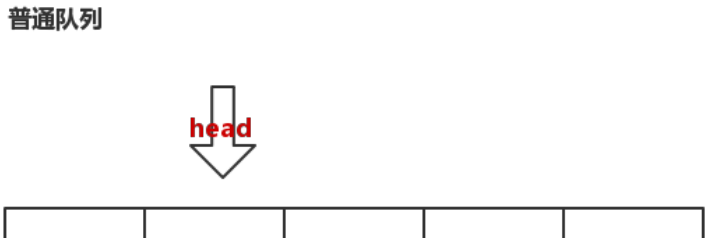
> 如果你是底层工程师，需要极致的比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。

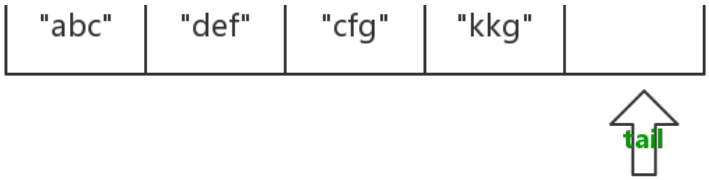
实现方法

```
import java.util.Arrays;

import org.graalvm.compiler.core.common.alloc.Trace;

public class Array {
    public int[] data;
    private int n;
    private int count;

    public Array(int capacity) {
        data = new int[capacity];
        n = capacity;
        count = 0;

    }

    public boolean insert(int location, int value) {
        if (n == count) {
            return false;
        }

        if (location < 0 || location > count) {
            return false;
        }

        for (int i = count; i > location; i--) {
            data[i] = data[i-1];
        }

        data[location] = value;
        count++;
        return true;
    }

    public int find(int location) {
        if (location < 0 || location >= count) {
            return -1;
        }

        return data[location];
    }

    public boolean delete(int location) {
        if (location < 0 || location >= count) {
            return false;
        }
```

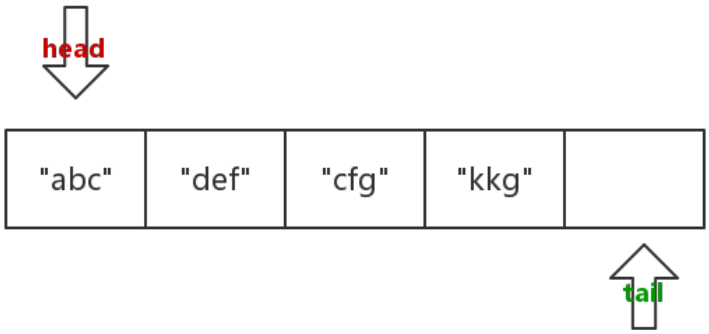
```
        for (int i = location + 1; i < count; i++) {
            data[i-1] = data[i];
        }

        count--;
        return true;


    }

    public static void main(String[] args) {
        Array array = new Array(5);
        array.insert(0, 1);
        array.insert(0, 2);
        array.insert(1, 3);
        array.insert(2, 4);
        array.insert(4, 5);
        // 判断插入不成功
        assert !array.insert(0, 100);
        assert array.find(0) == 2;
        assert array.find(2) == 4;
        assert array.find(4) == 5;
        assert array.find(10) == -1;
        assert array.count == 5;
        boolean removed = array.delete(4);
        assert removed;
        assert array.find(4) == -1;
        removed = array.delete(10);
        assert !removed;
        // 2 3 4 1 5
        int[] tmp = new int[] { 2, 3, 4, 1, 5 };
        assert Arrays.equals(array.data, tmp);
    }

}
```

三、队列

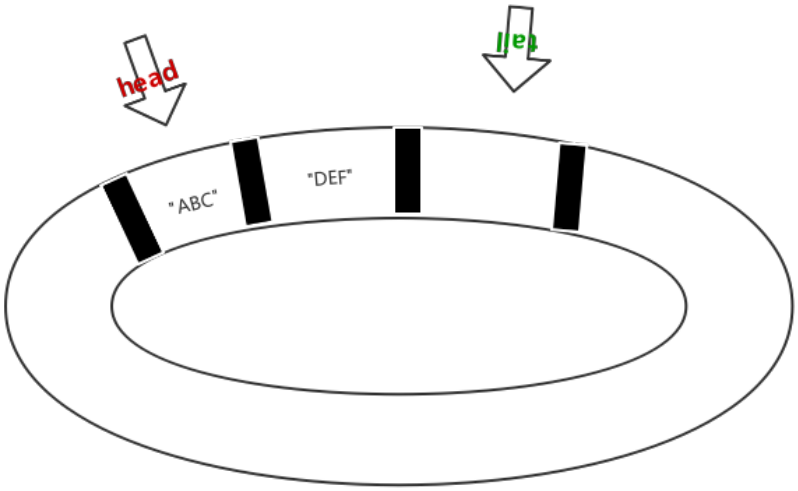




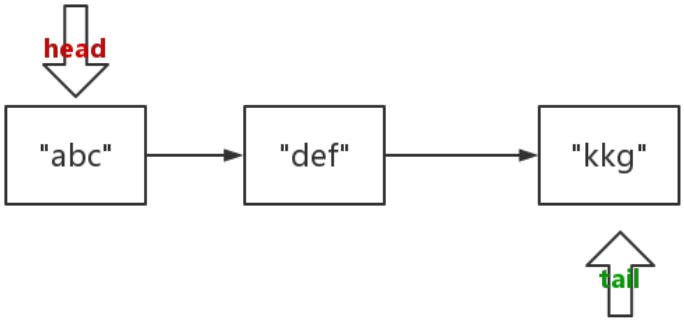
优化队列



循环队列



链式队列



利用数组实现

普通队列 优化队列 循环队列

利于优化

利用链表实现 普通链表队列

不易优化

队列二要素：入队，出队

头结点：

- 删除指定位置元素
- 判断队列是否为空

尾结点：

- 指定新元素的插入位置
- 通过尾结点判断队列是否满了，如果满了，就禁止插入
- 判断队列是否为空

顺序队列

基础队列

```
public class Queue {
    int n = 0;
    String[] items;
    int head;
    int tail;

    public Queue(int capacity) {
        n = capacity;
        items = new String[capacity];
    }

    public boolean enqueue(String item) {
        //如果队列满
        if (tail == n) return false;
        items[tail++] = item;
        return true;
    }

    public String dequeue() {
        if (head == tail) return null;
        return items[head++];

    }

    public static void main(String[] args) {
        Queue a = new Queue(10);
        a.enqueue("10");
        a.enqueue("20");
```

```
        String dequeItem = a.dequeue();
        System.out.println(dequeItem.equals("10"));
        a.enqueue("30");
        System.out.println(a.items[a.head].equals("20"));
        System.out.println(a.items[a.tail - 1].equals("30"));

    }
}
```

加入数据迁移的队列

上一种队列会越用越小，就需要对数据进行迁移，实现队列的自动整理。

```
public class Queue {
    private String[] items;
    private int n = 0;
    private int head = 0;
    private int tail = 0;

    public tmp(int capacity) {
        n = capacity;
        items = new String[n];
    }

    public boolean enqueue(String item) {
        if (tail == n) {
            if (head == 0) return false;
            for (int i = head; i < tail; ++i) {
                items[i-head] = items[i];
            }
            tail -= head;
            head = 0;
        }
        items[tail] = item;
        ++tail;
        return true;
    }

    public String dequeue() {
        if (head == tail)
            return null;
        return items[head++];
    }

    public static void main(String[] args) {
        Queue a = new Queue(3);
        a.enqueue("10");
        a.enqueue("20");
        a.enqueue("30");
```

```
        boolean result = a.enqueue("40");
        System.out.println(!result);
        String dequeItem = a.dequeue();
        System.out.println(dequeItem.equals("10"));
        a.enqueue("30");
        System.out.println(a.items[0].equals("20"));
        System.out.println(a.items[2].equals("30"));

    }
}
```

链式队列

```
public class Queue {
    Node head = null;
    Node tail = null;
    public void enqueue(String item) {
        if (tail == null) {
            Node newNode = new Node(item);
            head = newNode;
            tail = newNode;
        }else {
            tail.next = new Node(item);
            tail = tail.next;
        }
    }

    public String dequeue() {
        if (head == null) return null;
        String value = head.data;
        head = head.next;
        return value;
    }

    public static void main(String[] args) {
        Queue a = new Queue();
        a.enqueue("10");
        a.enqueue("20");
        a.enqueue("30");
        String dequeItem = a.dequeue();
        System.out.println(dequeItem == "10");
        System.out.println(a.head.data == "20");
        System.out.println(a.head.next.data == "30");

    }

    public static class Node {
        String data;
```

```
        Node next = null;
        public Node(String data) {
            this.data = data;
        }

    }

}
```

循环队列

```
public class Queue {
    int n = 0;
    String[] items;
    int head = 0;
    int tail = 0;

    public Queue(int capacity) {
        n = capacity;
        items = new String[capacity];

    }

    public boolean enqueue(String item) {
        if ((tail + 1) % n == head)
            return false;
        items[tail] = item;
        tail = (tail + 1) % n;
        return true;

    }

    public String dequeue() {
        if (head == tail)
            return null;
        String value = items[head];
        head = (head + 1) % n;
        return value;
    }

    public static void main(String[] args) {
        Queue a = new Queue(3);
        a.enqueue("10");
        a.enqueue("20");
        boolean result = a.enqueue("30");
        // 循环队列需要空出一格
        System.out.println(!result);
        String dequeItem = a.dequeue();
```

```
        a.enqueue("30");
        System.out.println(a.items[2] == "30");
        result = a.enqueue("10");
        System.out.println(result == false);

    }

}
```

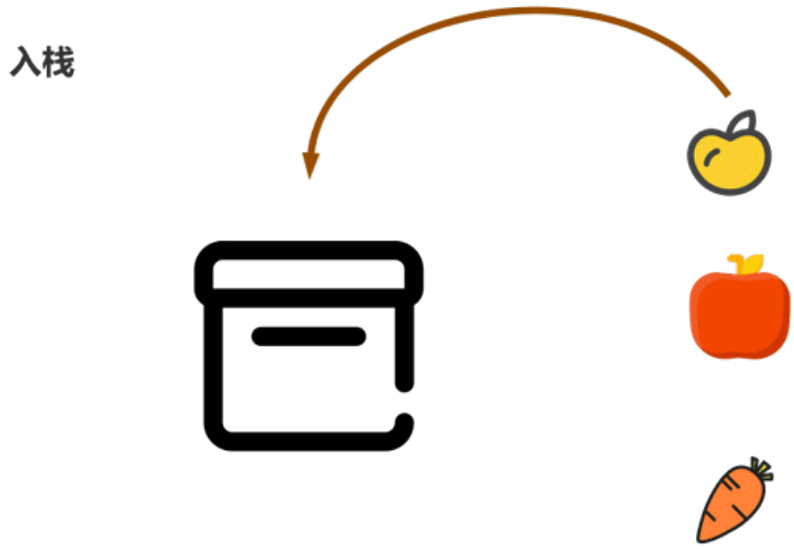
阻塞队列

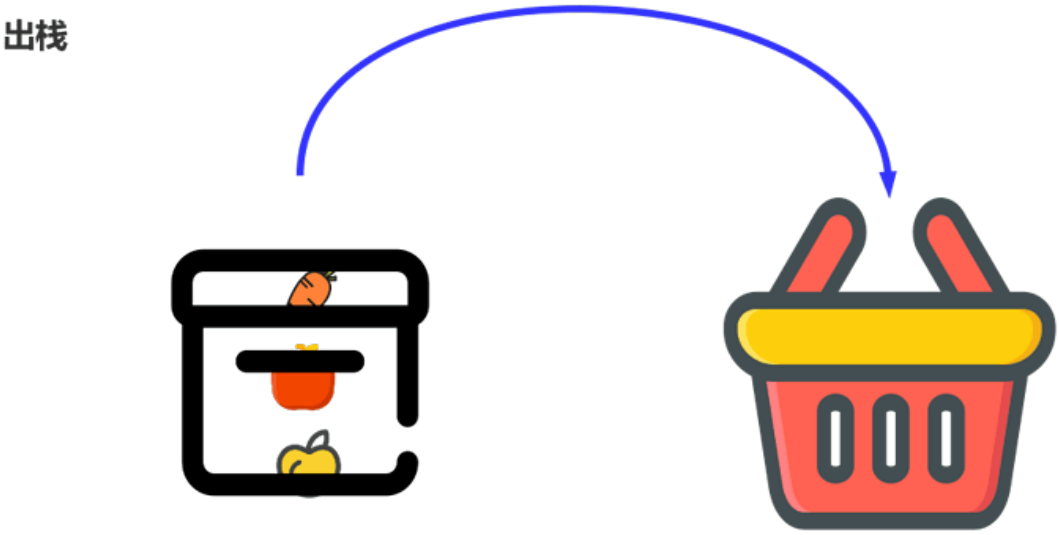
入队和出队可以等待

并发队列

支持多线程的阻塞队列

四、堆栈





- 函数调用
- 编译原理：语法分析
- 括号匹配

问题：栈是**操作受限**的线性表，为什么不直接用数组或者链表？

数组和链表暴露了太多接口，操作虽然灵活，但不可控，容易出错。比如数组支持任意位置插入数组，如果插入位置写错将改变所有数组的内容。

而栈只能在**一端插入和删除数据，并且后进先出**。

顺序栈

使用数组实现栈

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

// 基于数组实现的顺序栈
public class ArrayStack {
    private String[] data;
    int n;
    int count;

    public ArrayStack(int n) {
        this.n = n;
        count = 0;
        data = new String[n];

    }

    public boolean push(String value) {
        if (n == count)
            return false;

        data[count++] = value;
        return true;

    }

    public String pop() {
        if (count == 0)
            return null;

        return data[--count];
    }

    public static void main(String[] args) {
        ArrayStack arrayStack = new ArrayStack(5);
        List<String> data = new ArrayList<String>(Arrays.asList("a", "b", "c",
"d", "e"));

        for (String i : data) {
            arrayStack.push(i);
        }
        boolean result = arrayStack.push("a");
        System.out.println(!result);
        Collections.reverse(data);
        for (String i : data) {
            System.out.println(i.equals(arrayStack.pop()));
```

```
        }
        System.out.println(arrayStack.pop() == null);

    }
}
```

入栈时间复杂度：O(1) 出栈时间复杂度：O(1)

链式栈

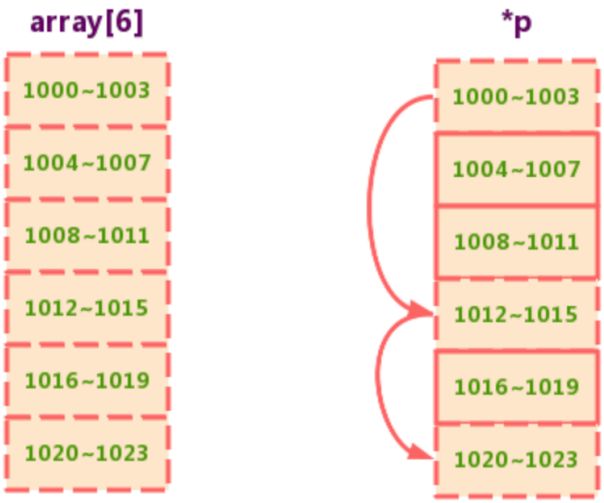
使用链表实现栈

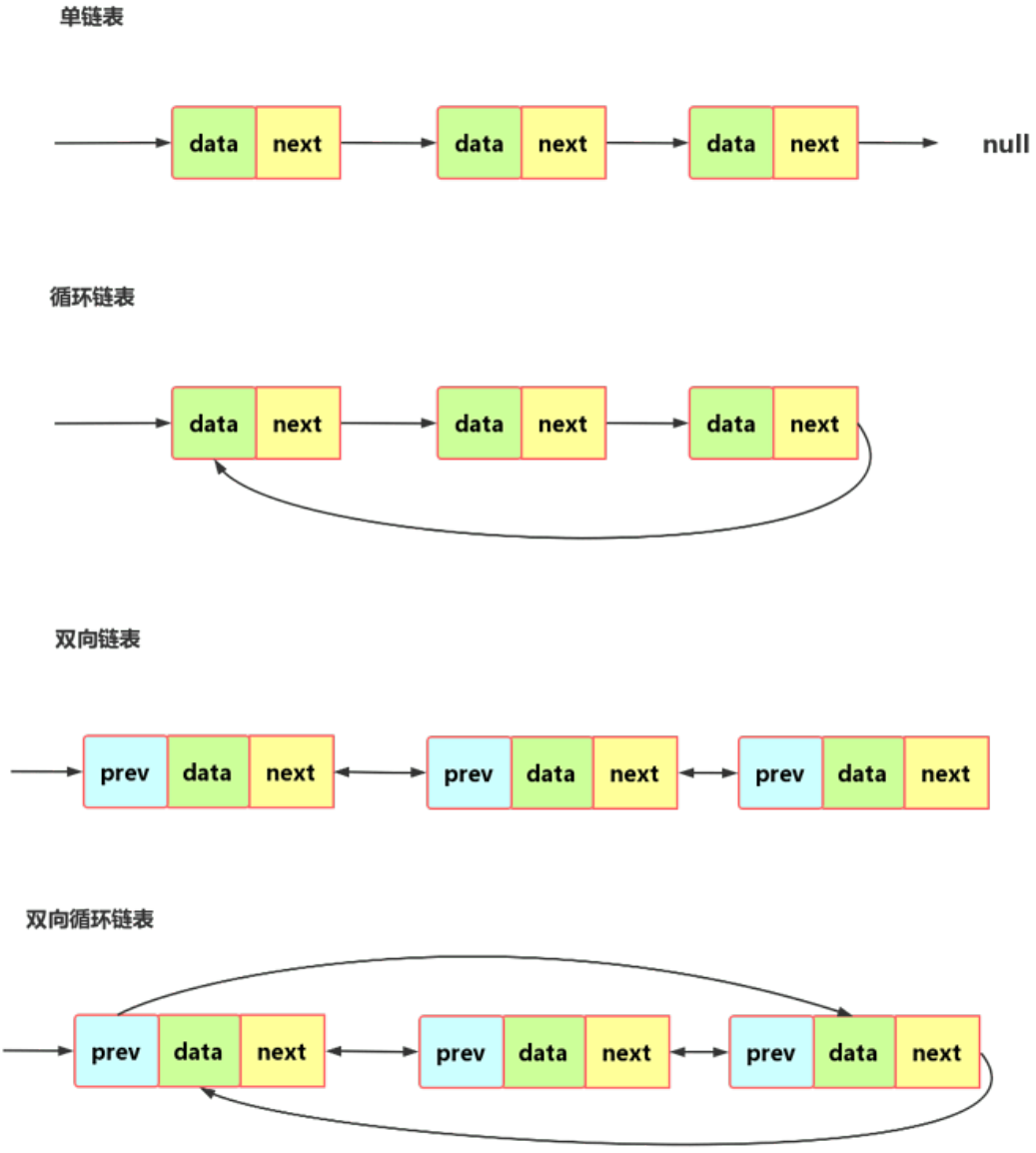
```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class StackBasedOnLinkedList {
    private Node top;

    public void push(int value) {
        Node newNode = new Node(value);
        if (top == null)
            top = newNode;
        else {
            newNode.next = top;
            top = newNode;
        }

    }

    public int pop() {
        if (top == null)
            return -1;

        int result = top.data;
        top = top.next;
        return result;
    }


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

    }
```

```
    public static void main(String[] args) {
        StackBasedOnLinkedList stack = new StackBasedOnLinkedList();
        List<Integer> data = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        for (int i : data) {
            stack.push(i);
        }
        Collections.reverse(data);
        for (int i : data) {
            System.out.println(i == stack.pop());
        }
        System.out.println(stack.pop() == -1);

    }
}
```

五、链表

链表与数组的区别





单链表与循环链表

注意链表中的头结点和尾结点。

循环链表从尾可以方便的到头，适合环型结构数据，比如约瑟夫问题。

双向链表

优势：

O(1) 时间复杂度找到前驱结点

删除，插入更高效。考虑以下两种情况：

- 删除结点中“值等于某个给定值”的结点

- 删除给定指针指向的结点

查询更高效：记录上次查找的位置 p，每次查询时，根据要查找的值与 p 的大小关系，决定是往前还是往后查找，所以平均只需要查找一半的数据。

再次比较数组和链表

从复杂度分析：

时间复杂度	数组	链表
插入删除	O(n)	O(1)
随机访问	O(1)	O(n)

其它角度：

- 内存连续，利用 CPU 的缓存机制，预读数组中的数据，所以访问效率更高。
- 而链表在内存中并不是连续存储，所以对 CPU 缓存不友好，没办法有效预读。
- 数组的大小固定，即使动态申请，也需要拷贝数据，费时费力。
- 链表支持动态扩容.

```
public class SinglyLinkedList {
    Node head;

    public void insertTail(int value) {
        if (head == null) {
            insertToHead(value);
            return;
        }
        Node q = head;
        while (q.next != null) {
            q = q.next;
        }
        Node newNode = new Node(value);
        q.next = newNode;

    }

    public void insertToHead(int value) {
        Node newNode = new Node(value);
        newNode.next = head;
        head = newNode;

    }

    public boolean deleteByValue(int value) {
        if (head == null)
            return false;
```

```
        Node p = head;
        Node q = null;
        while (p != null && p.data != value) {
            q = p;
            p = p.next;
        }
        // 链表中，没有 value
        if (p == null)
            return false;
        // 数据在 head 上
        if (q == null) {
            head = head.next;
        } else {
            q.next = p.next;
        }
        return true;

    }

    public Node findByValue(int value) {
        if (head == null)
            return null;
        Node q = head;
        while (q != null && q.data != value) {
            q = q.next;
        }
        if (q == null)
            return null;

        return q;
    }

    public void insertAfter(Node node, int value) {
        if (node == null)
            return;

        Node newNode = new Node(value);
        newNode.next = node.next;
        node.next = newNode;

    }

    public void insertBefore(Node node, int value) {
        if (node == null)
            return;
        if (head == null) {
            insertToHead(value);
        } else {
            Node q = head;
```

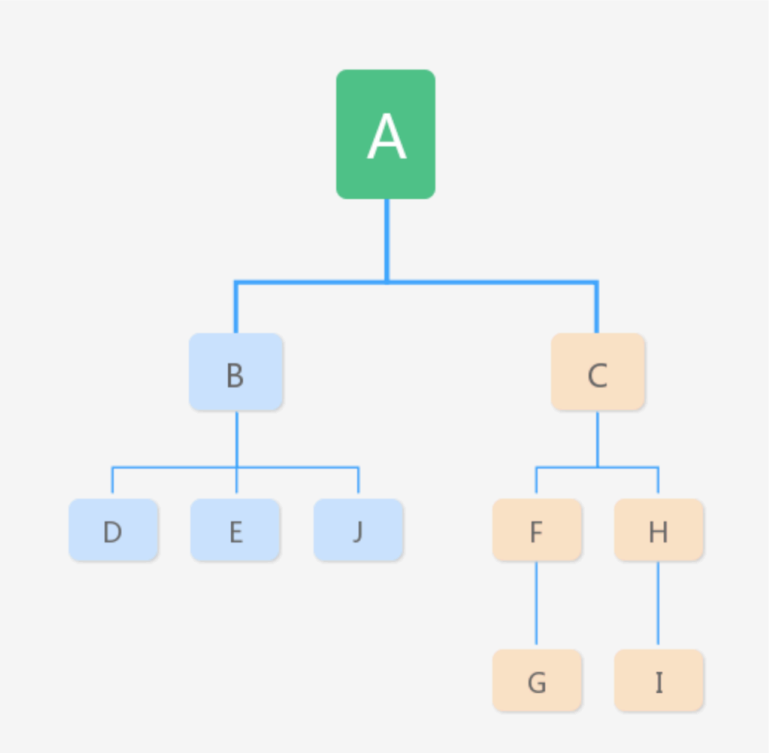
```
            while (q != null && q.next != node) {
                q = q.next;
            }
            if (q == null)
                return;
            Node newNode = new Node(value);
            newNode.next = q.next;
            q.next = newNode;
        }

    }

    public void printAll() {
        if (head == null)
            return;
        Node q = head;
        while (q != null) {
            System.out.print(q.data + "  ");
            q = q.next;
        }
        System.out.println();
    }

    public static class Node {
        int data;
        Node next;

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

    public static void main(String[] args) {
        SinglyLinkedList link = new SinglyLinkedList();
        int data[] = { 1, 2, 5, 3, 1 };
        for (int i = 0; i < data.length; i++) {
            link.insertTail(data[i]);
        }
        link.insertToHead(99);
        // 打印内容为 99 1 2 5 3 1
        link.printAll();
        link.deleteByValue(2);
        System.out.println(!link.deleteByValue(999));
        System.out.println(link.deleteByValue(99));
        // 打印内容为 1 5 3 1
        link.printAll();
        System.out.println(link.findByValue(2) == null);
        Node newNode = link.findByValue(3);
        link.insertAfter(newNode, 10);
```

```
        System.out.println(link.findByValue(3).next.data == 10);
        link.insertBefore(newNode, 30);
        System.out.println(link.findByValue(5).next.data == 30);

    }

}
```

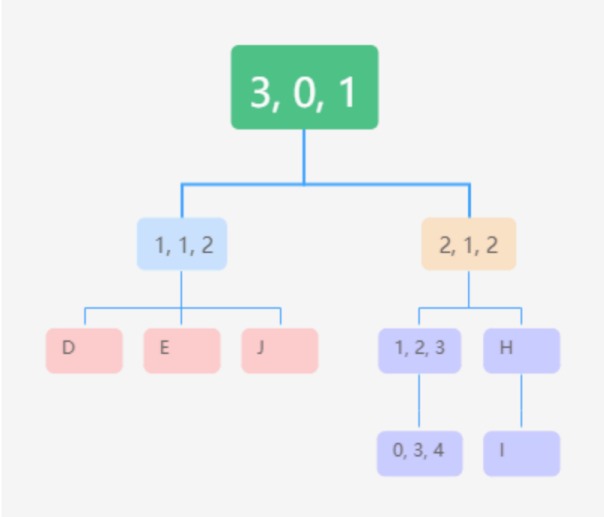
六、二叉树

树的基本术语

父节点，子节点： A 和B 兄弟节点：D E J 根节点：A 叶节点：G I



- 节点高度：节点到叶节点的最长路径（边数）
- 节点深度：根节点到这节点所经历的边的个数
- 节点的层数：节点的深度 + 1
- 树高度：根节点的高度

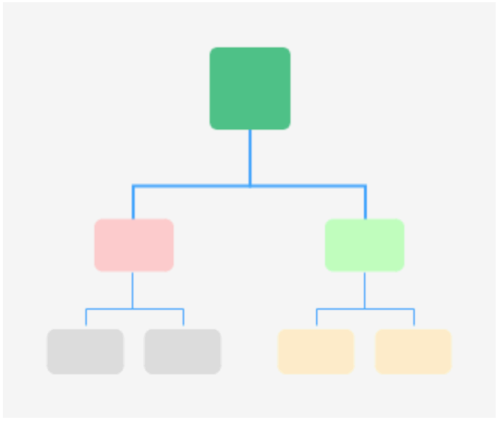


- 节点的高度：一个人量身体各部位长度，从脚下拉尺到各部位（腿长）
- 节点深度：量水中物品的深度，从水面拉尺到水中的宝藏
- 节点的层数：尺子从 1 开始计数
- 树高度：头到脚的高度

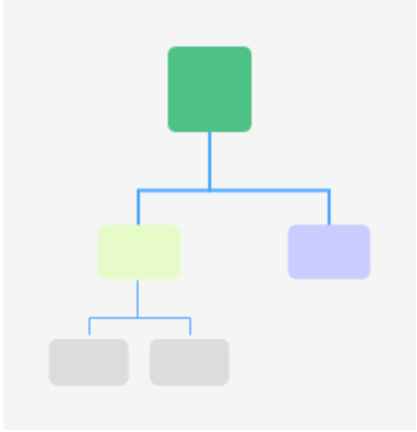
二叉树

最多有两个叉的树。

满二叉树：叶子节点全在最底层，除了叶子节点之外，每个节点都有左右两个子节点。

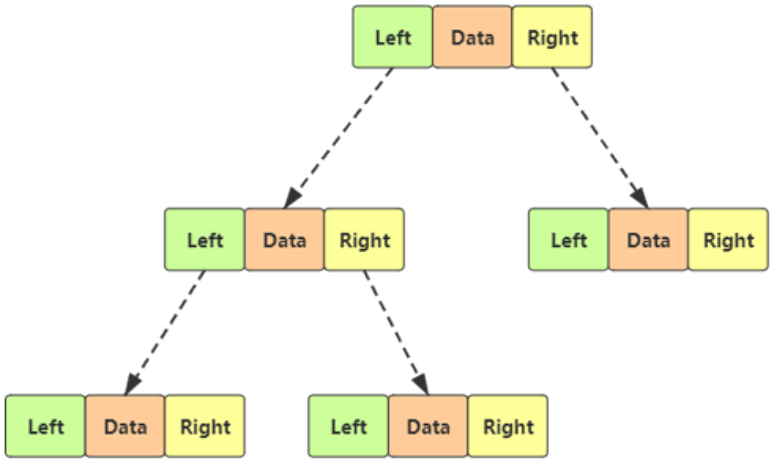


完全二叉树：叶子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大。

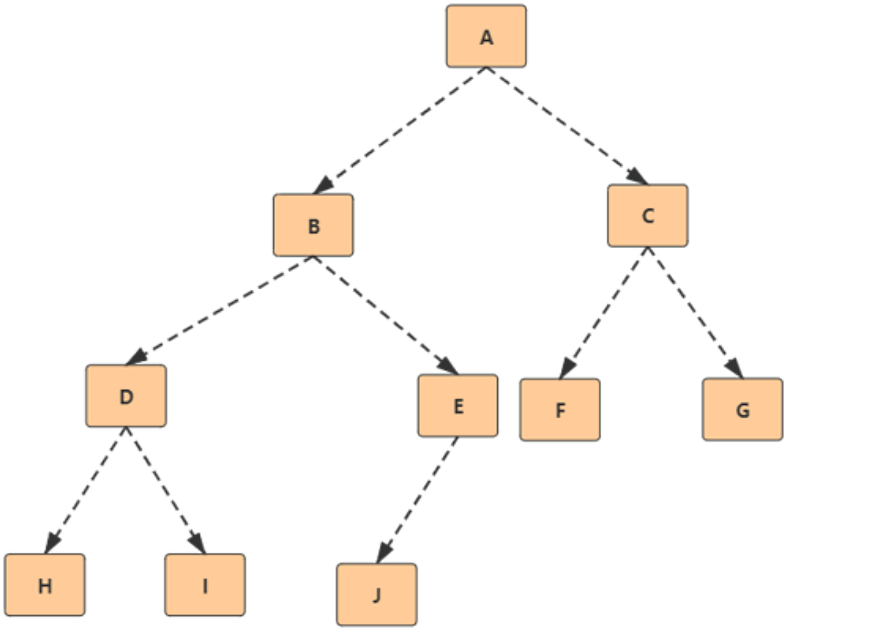


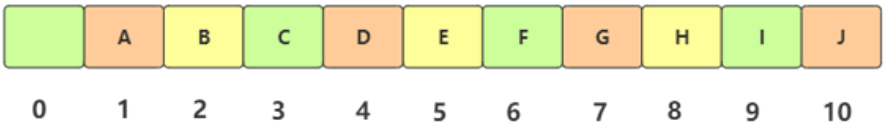
满二叉树和完全二叉树的意义是什么?

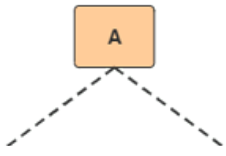
链式二叉树

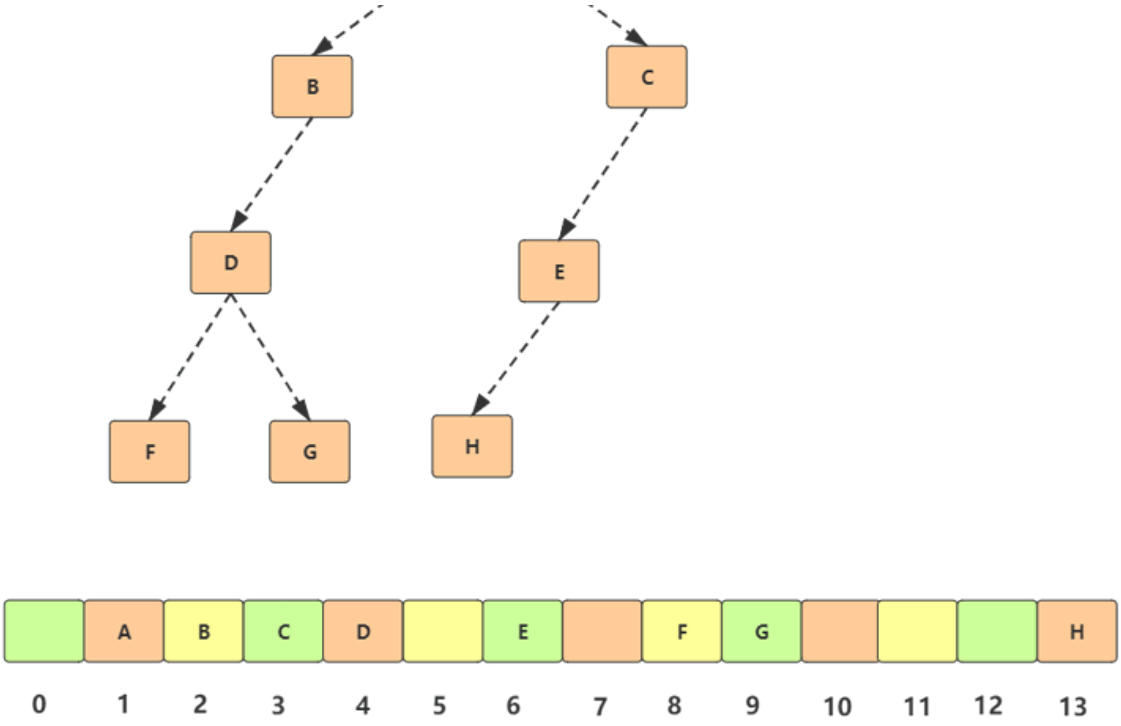


顺序二叉树



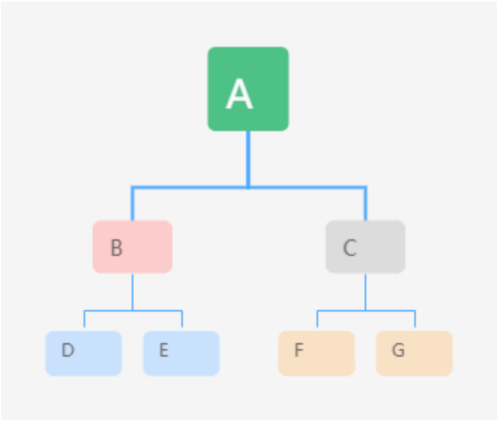






- 节点 X 在数组下标 i
- 左子节点：2 * i
- 右子节点：2 * i + 1
- 父节点：i/2

二叉树的遍历



根据节点打印的**顺序**分**前**，**中**，**后**。比如：

- 前序遍历：节点 → 左子树 → 右子树。A->B->D->E->C->F->G
- 中序遍历：左子树 → 节点 → 右子树。D->B->E->A->F->C->G
- 后序遍历：左子树 → 右子树 → 节点。D->E->B->F->G->C->A

递推关系式：

```
前序遍历的递推公式：
preOrder(r) = print r->preOrder(r->left)->preOrder(r->right)

中序遍历的递推公式：
inOrder(r) = inOrder(r->left)->print r->inOrder(r->right)

后序遍历的递推公式：
postOrder(r) = postOrder(r->left)->postOrder(r->right)->print r
```

代码：

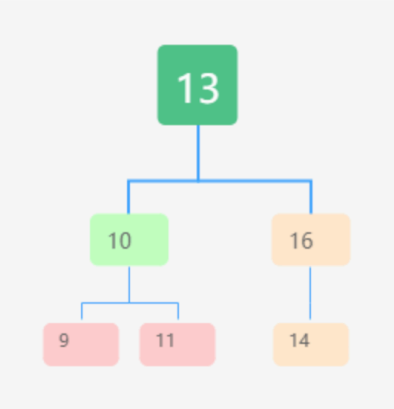
```
void preOrder(Node* root) {
  if (root == null) return;
  print root // 此处为伪代码，表示打印root节点
  preOrder(root->left);
  preOrder(root->right);
}

void inOrder(Node* root) {
  if (root == null) return;
  inOrder(root->left);
  print root // 此处为伪代码，表示打印root节点
  inOrder(root->right);
}

void postOrder(Node* root) {
  if (root == null) return;
  postOrder(root->left);
  postOrder(root->right);
  print root // 此处为伪代码，表示打印root节点
}
```

复杂度：

O(n)：遍历操作的时间复杂度，跟节点的个数 n 成正比

二叉查找树

左子树每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值。

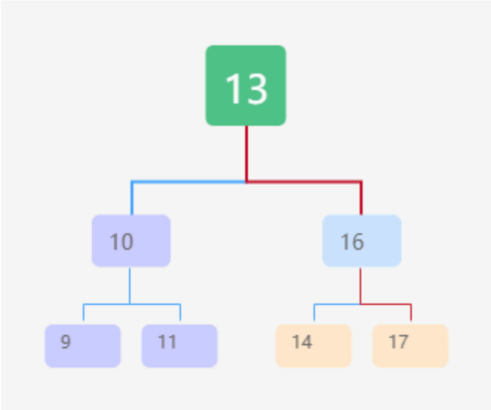


查找

根节点 -比之小> 左子树中递归查找。 -比之大> 右子树中递归查找。

插入

- 从根节点开始比较，如果比之大，并且节点右子树为空，就将新数据插到右子节点；
- 如果不为空，就再递归遍历右子树，查找插入位置。
- 如果要插入的数据比节点数值小，并且节点的左子树为空，就将新数据插入到左子节点的位置。
- 如果不为空，就再递归遍历左子树，查找插入位置。



删除

情况一：要删除的节点没有子节点

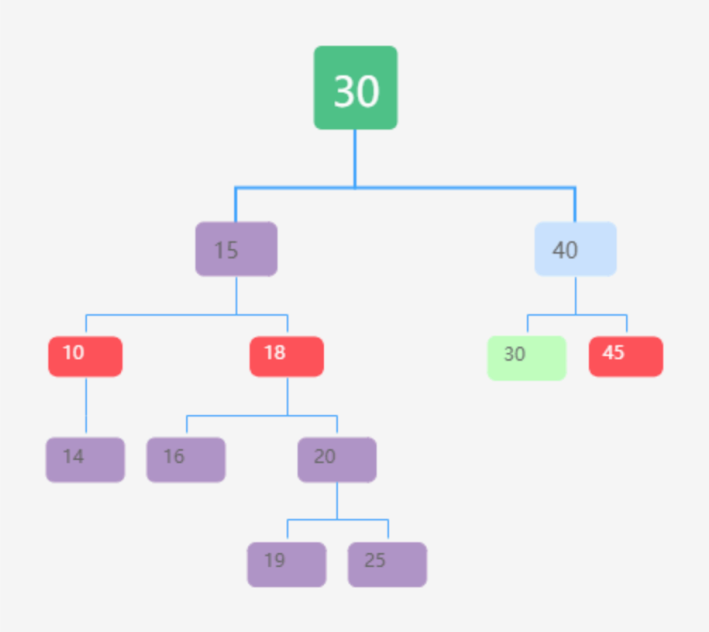
只需要直接将父节点中，指向要删除节点的指针置为 null。比如图中的删除节点 55。

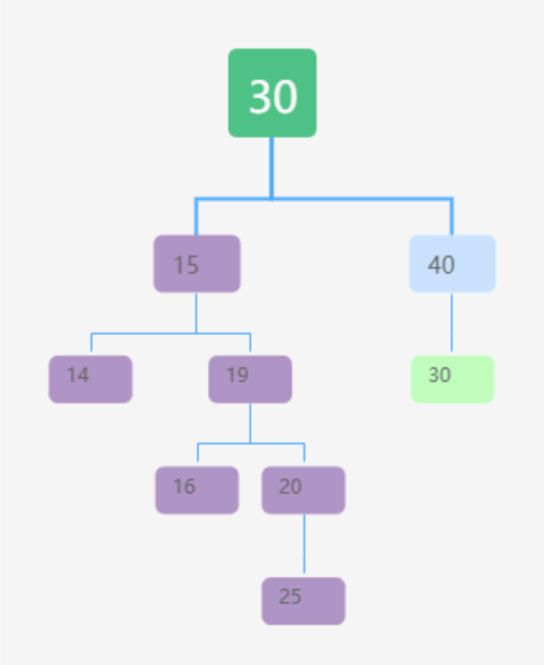
情况二：如果要删除的节点只有一个子节点（只有左子节点或者右子节点），

只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。比如图中的删除节点 13。

情况三：如果要删除的节点有两个子节点

- 找到这个节点的右子树中的最小节点，把它替换到要删除的节点上。
- 然后再删除掉这个最小节点，因为最小节点肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。比如图中的删除节点 18。





> 取巧方法：将要删除的节点标记为“已删除”。

重要特性

中序遍历二叉查找树，可以得到有序的数据序列，时间复杂度是 O(n)。因此，二叉查找树又称二叉排序树。

```
import org.graalvm.compiler.nodes.calc.RightShiftNode;

public class BinarySearchTree {
    Node tree;

    public void insert(int value) {
        if (tree == null) {
            tree = new Node(value);
```

```
            return;
        }
        Node p = tree;
        while (p != null) {
            if (value > p.data) {
                if (p.right == null) {
                    p.right = new Node(value);
                    return;
                }
                p = p.right;
            } else if (value < p.data) {
                if (p.left == null) {
                    p.left = new Node(value);
                    return;
                }
                p = p.left;
            }
        }
    }

    public Node find(int value) {
        Node p = tree;
        while (p != null) {
            if (value > p.data)
                p = p.right;
            else if (value < p.data)
                p = p.left;
            else
                return p;
        }

        return null;

    }

    public void delete(int value) {
        // 要删除的结点
        Node p = tree;
        // 要删除结点的父结点
        Node pp = null;

        while (p != null && p.data != value) {
            if (value > p.data) {
                pp = p;
                p = p.right;
            }
            else if (value < p.data) {
                pp = p;
                p = p.left;
```

```
            }
        }
        // 没找到
        if (p == null)
            return;

        if (p.left != null && p.right != null) {
            Node tmpP = p.right;
            Node tmpPP = p;
            while (tmpP.left != null) {
                tmpPP = tmpP;
                tmpP = tmpP.left;
            }
            p.data = tmpP.data;
            p = tmpP;
            pp = tmpPP;
        }

        // 如果 p 结点下面有一个或没有孩子时
        Node child;
        if (p.left != null)
            child = p.left;
        else if (p.right != null)
            child = p.right;
        else
            child = null;

        // 把孩子接到 p 的父结点上 pp

        if (pp == null)
            tree = child;
        else if (pp.right == p)
            pp.right = child;
        else if (pp.left == p)
            pp.left = child;

    }

    public void preOrder(Node node) {
        if (node == null) return;
        System.out.println(node.data);
        preOrder(node.left);
        preOrder(node.right);
    }

    public void inOrder(Node node) {
        if (node == null) return;
        inOrder(node.left);
        System.out.println(node.data);
```

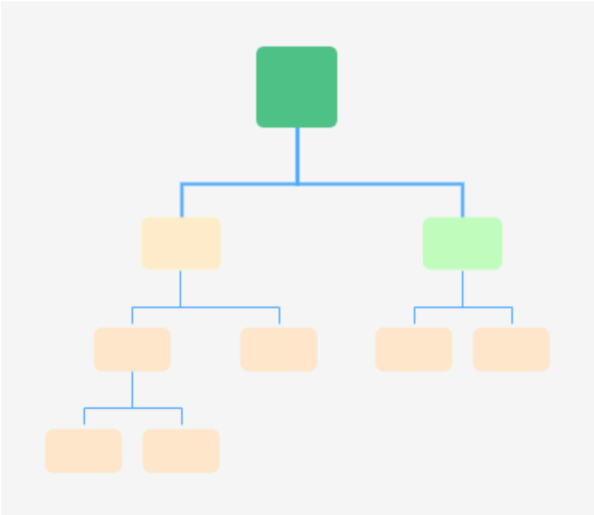
```
        inOrder(node.right);
    }

    public void postOrder(Node node) {
        if (node == null) return;
        postOrder(node.left);
        postOrder(node.right);
        System.out.println(node.data);
    }

    public static class Node {
        int data;
        Node left;
        Node right;

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

        }
    }

    public static void main(String[] args) {
        BinarySearchTree binarySearchTree = new BinarySearchTree();
        int[] data = new int[] { 1, 10, 20, 40, 13 };
        for (int i : data) {
            binarySearchTree.insert(i);
        }
        System.out.println(20 == binarySearchTree.find(20).data);
        binarySearchTree.delete(20);
        System.out.println(null == binarySearchTree.find(20));
        // 1 10 40 13
        binarySearchTree.preOrder(binarySearchTree.tree);
        System.out.println("-----------------------");
        // 1 10 13 40
        binarySearchTree.inOrder(binarySearchTree.tree);
        System.out.println("-----------------------");
        // 13 40 10 1
        binarySearchTree.postOrder(binarySearchTree.tree);

    }
}
```

七、多叉树与红黑树

引言

时间复杂度分析

树的插入，删除，查找的**时间复杂度与树的高度成正比**： O(高度) == O(最大层数 - 1) ，最符合需求的就是完全二叉树，来看下完全二叉树的最大高度：



完全二叉树除最后一层，每层结点数：1, 2, 4,..., 2k-1 最后一层：1~2(L-1) 完全二叉树的结点数 n：

n >= 1+2+4+8+...+2L-2 + 1

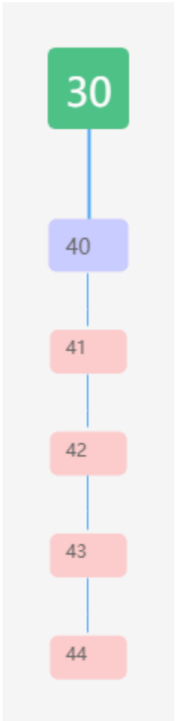
n <= 1+2+4+8+...+2L-2 + 2L-1

利用等比数列求和公式：

$$S_n = a_1 \cdot \frac{1-q^n}{1-q} = \frac{a_1 - a_n \cdot q}{1-q} \quad (q \neq 1)$$

L 的范围是 [log2(n+1), (log2n) +1]。完全二叉树的层数 <= (log2n) +1 ，其高度小于等于 log2n

二叉查找树会出现树的高度过高，从而导致各个操作的效率下降，比如下图只有右孩子的树（退化成了链表）：



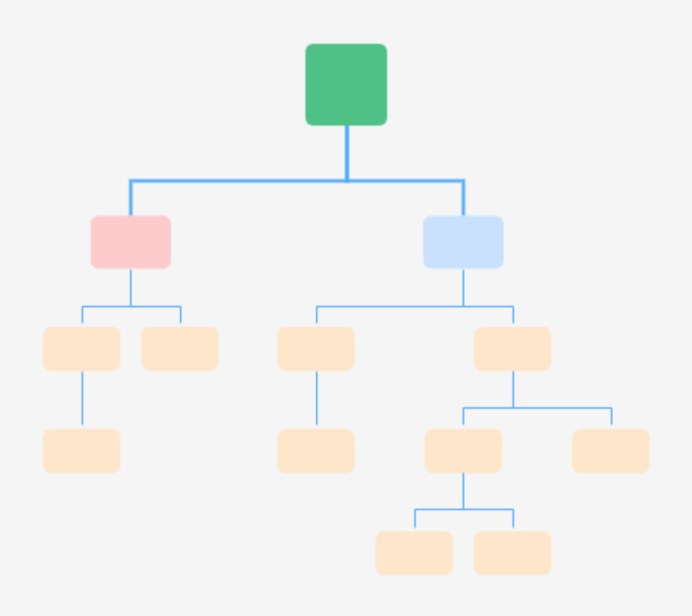
为避免这种情况，一般选用平衡二叉查找树：红黑树。

什么是平衡二叉查找树?

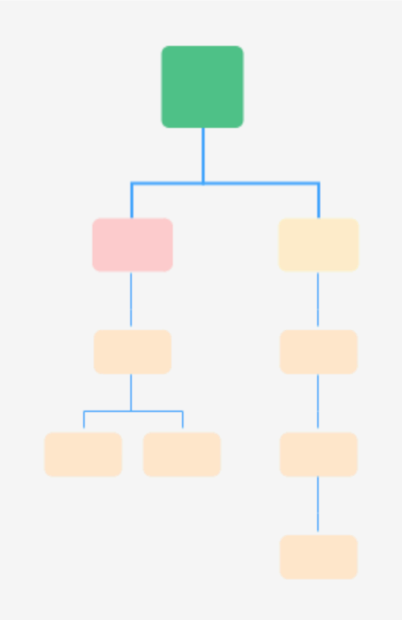
平衡二叉树：任意节点左右子树的高度差 <= 1：

- 完全二叉树
- 满二叉树
- 一些非完全二叉树

平衡二叉树：



非平衡二叉树



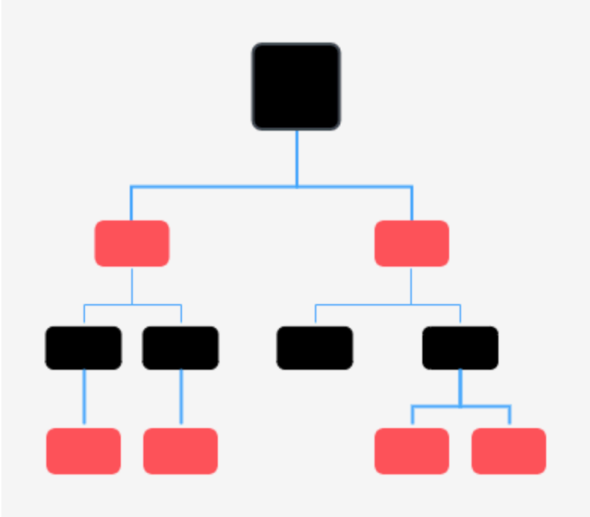
早期的 AVL 树是**平衡二叉树** + **查找** = **平衡二叉查找树**。

但为了实际应用，后续的树都没遵从平衡树的定义（比如红黑树），只要树的高度不比 log2n 大很多，都符合定义。

红黑树

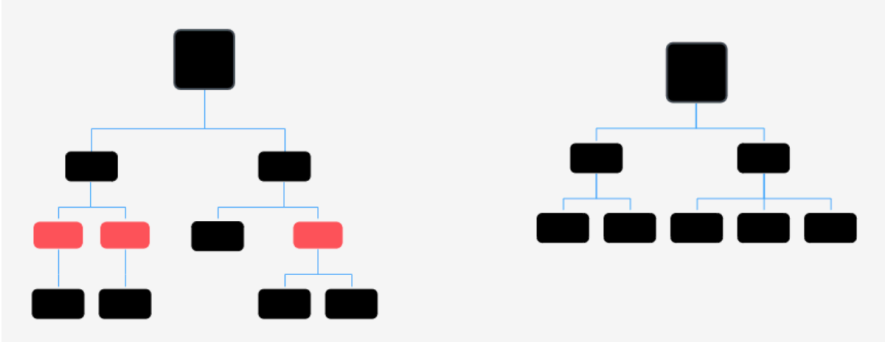
红黑树的英文是 **Red-Black Tree**，简称 R-B Tree。

- 根节点是黑色。
- 叶子节点是黑色的空节点（为了简化代码，本章节可以不考虑）
- 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）
- 每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点



为什么红黑树的高度可以稳定地趋近 log2n？

将红色节点删除，变成下图（四叉树），由于每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点，于是四叉树可以转换成完全二叉树，于是**仅包含黑色节点的四叉树的高度，比包含相同节点个数的完全二叉树的高度还要小**



把红色节点加回去，由于红色节点不能相邻（红色数量 ~= 黑色数量）。红黑树的高度近似 2log2n。

平衡树比较

- Treap（树堆）、Splay Tree（伸展树）无法避免时间复杂度的退化
- AVL 树是一高度平衡的二叉树，查找效率非常高，但是为了维持高度平衡，每次插入、删除都要做调整
- 红黑树做到了近似平衡，维护平衡的成本比 AVL 树低

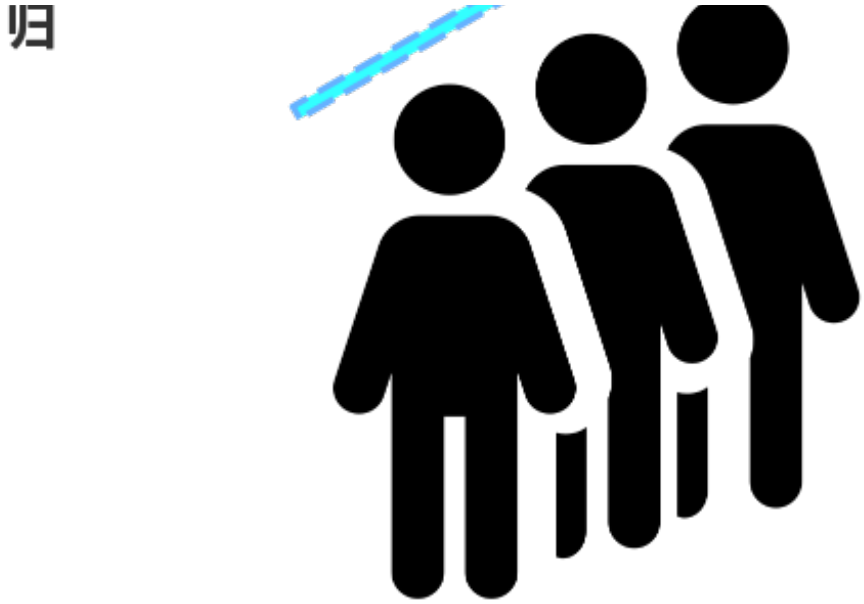
八、递归计算

什么是递归?

递归应用非常广泛，比如 DFS 深度优先搜索、前中后序二叉树遍历等等。

假设一群人在排队，你不清楚自己前面有多少人，可以问前面的人：

但你前面也不清楚，整体过程就是递归：



如果用递推公式描述：

```
f(n)=f(n-1)+1 其中，f(1)=1

int f(int n) {
  if (n == 1) return 1;
  return f(n-1) + 1;
}
```

什么情况可以使用递归？

1. 一个问题可以拆分成子问题.
2. 这些问题求解思路相同（数据规模不同）
3. 拥有终止条件

来看一个问题：

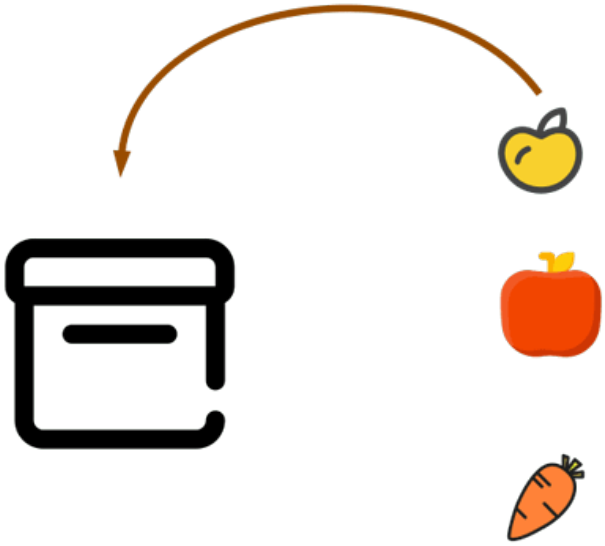
假如 n 个台阶（n >= 3），每次可以走 1 或 2 个台阶，走完 n 个台阶有多少种走法？ 比如有 3 个台阶，可以 1，2 或者 2，1：

```
f(n) = f(n-1)+f(n-2)  其中，f(1)=1，f(2)=2

int f(int n) {
  if (n == 1) return 1;
  if (n == 2) return 2;
  return f(n-1) + f(n-2);
}
```

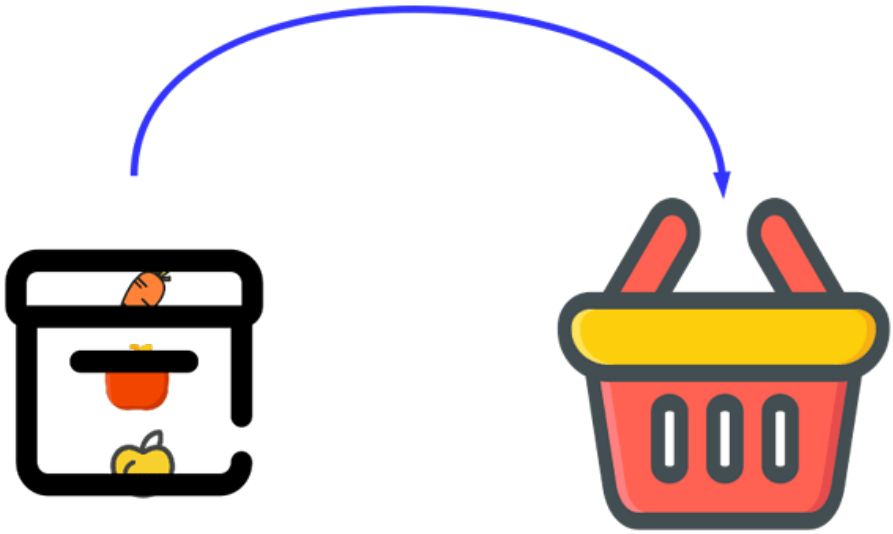
常见问题

深度问题

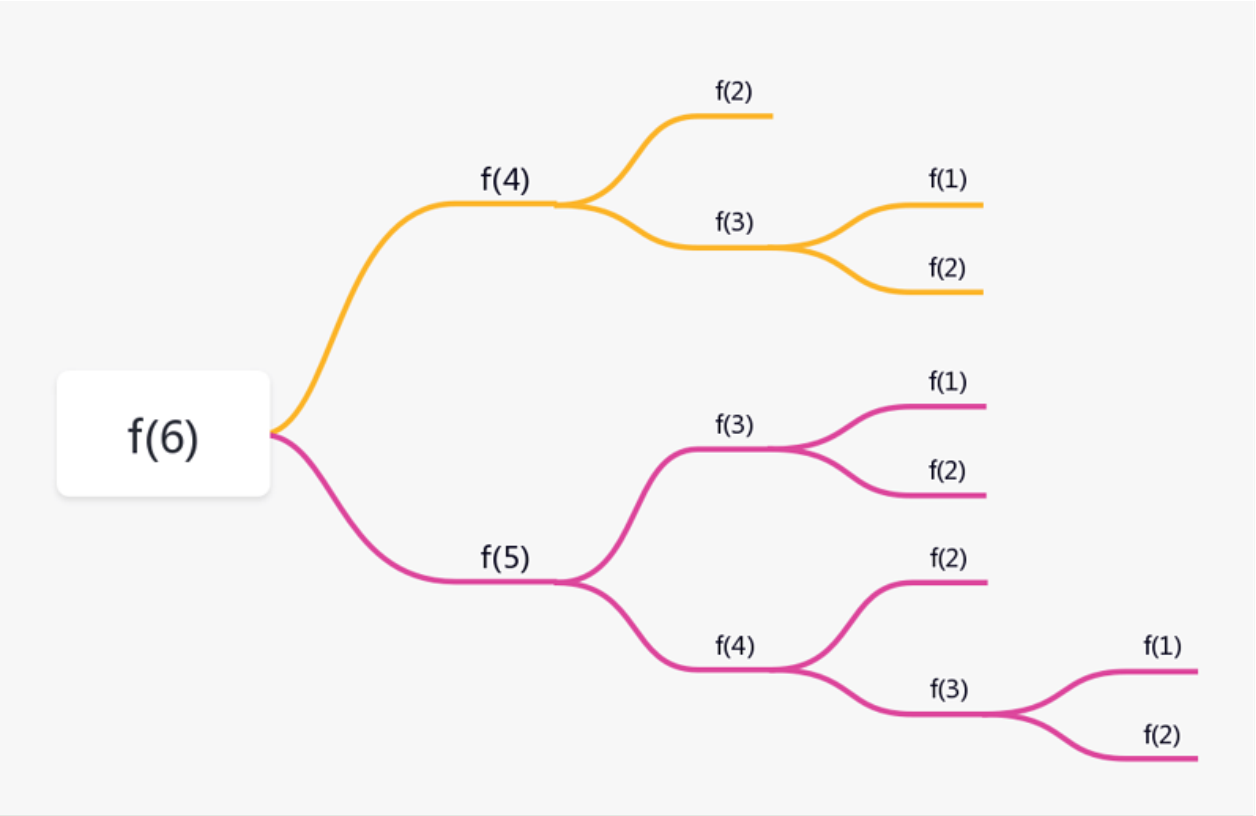
入栈



出栈



重复计算问题




```
public int f(int n) {
  if (n == 1) return 1;
  if (n == 2) return 2;

  // hasSolvedList可以理解成一个Map，key是n，value是f(n)
  if (hasSolvedList.containsKey(n)) {
    return hasSolvedList.get(n);
  }

  int ret = f(n-1) + f(n-2);
  hasSolvedList.put(n, ret);
  return ret;
}
```

内存开销问题

```
int f(int n) {
  if (n == 1) return 1;
  return f(n-1) + 1;
}
```

空间复杂度：O(n)

将递归改写为非递归

```
int f(int n) {
  int ret = 1;
  for (int i = 2; i <= n; ++i) {
    ret = ret + 1;
  }
  return ret;
}
```

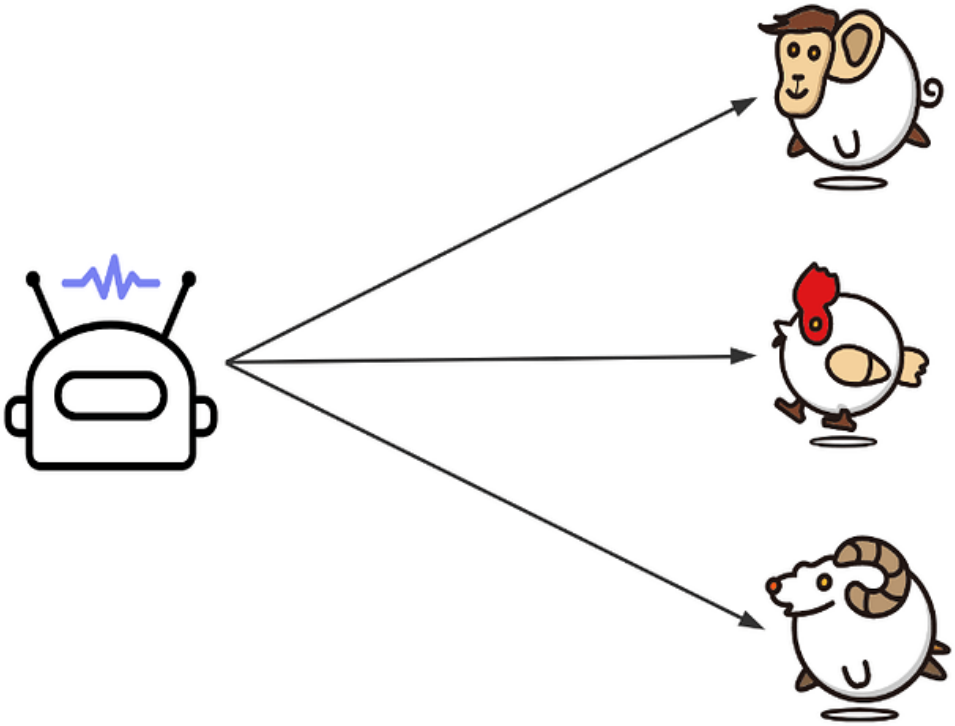
```
int f(int n) {
  if (n == 1) return 1;
  if (n == 2) return 2;

  int ret = 0;
  int pre = 2;
  int prepre = 1;
  for (int i = 3; i <= n; ++i) {
    ret = pre + prepre;
    prepre = pre;
    pre = ret;
  }
  return ret;
}
```

原理：函数调用底层原理是栈，可以利用 for 循环模拟入栈和出栈操作

九、单例模式

为什么要使用单例

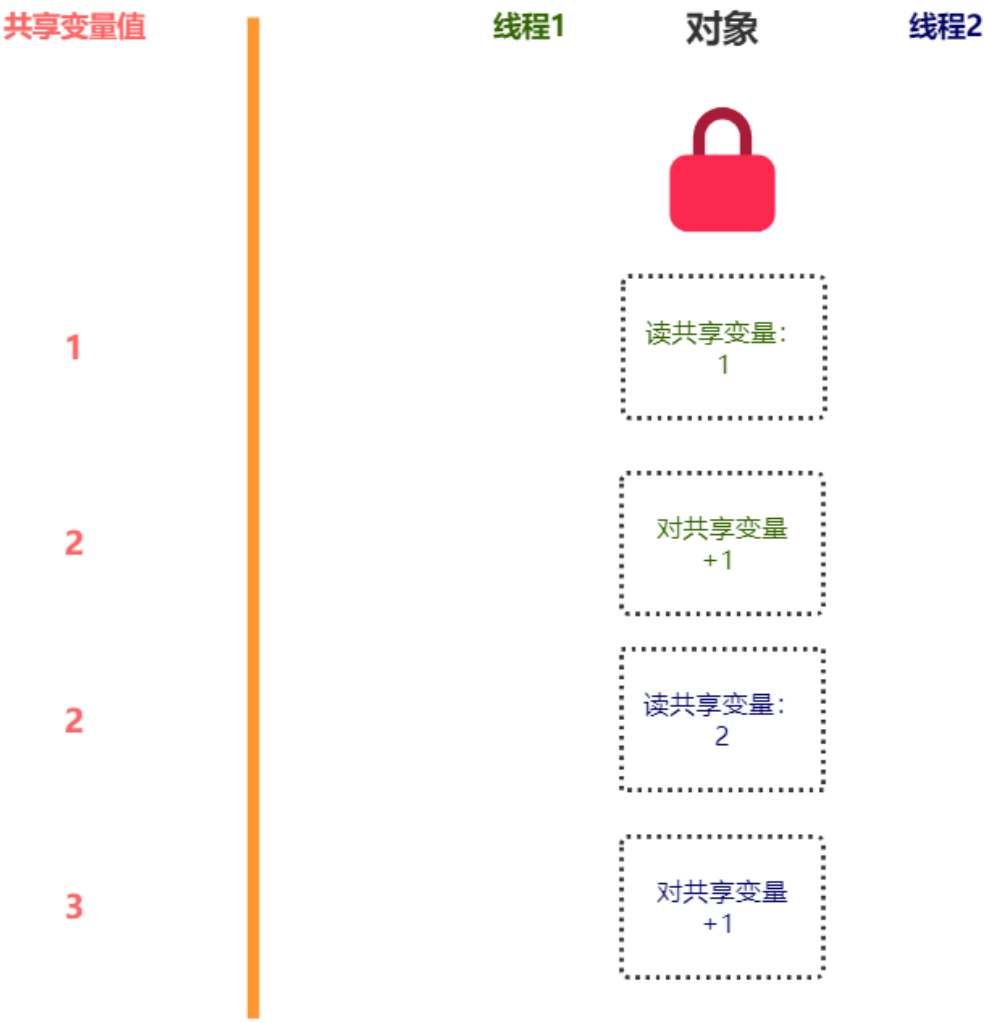
普通的类就像生产机器，可以生产多个实例（动物糕点），就像下图：



单例设计模式（Singleton Design Pattern）对实例做限制，一个类只允许创建一个对象。

那么，对类进行限制有什么好处？假设对文件进行写入，**同一个对的多个线程同时写入会造成资源竞争问题**，比如两个线程同时给一个共享变量加 1 ，共享变量最后的结果可能只加了 1 。

如果是同一个对象，加入**对象级别互斥锁**就能解解决问题。



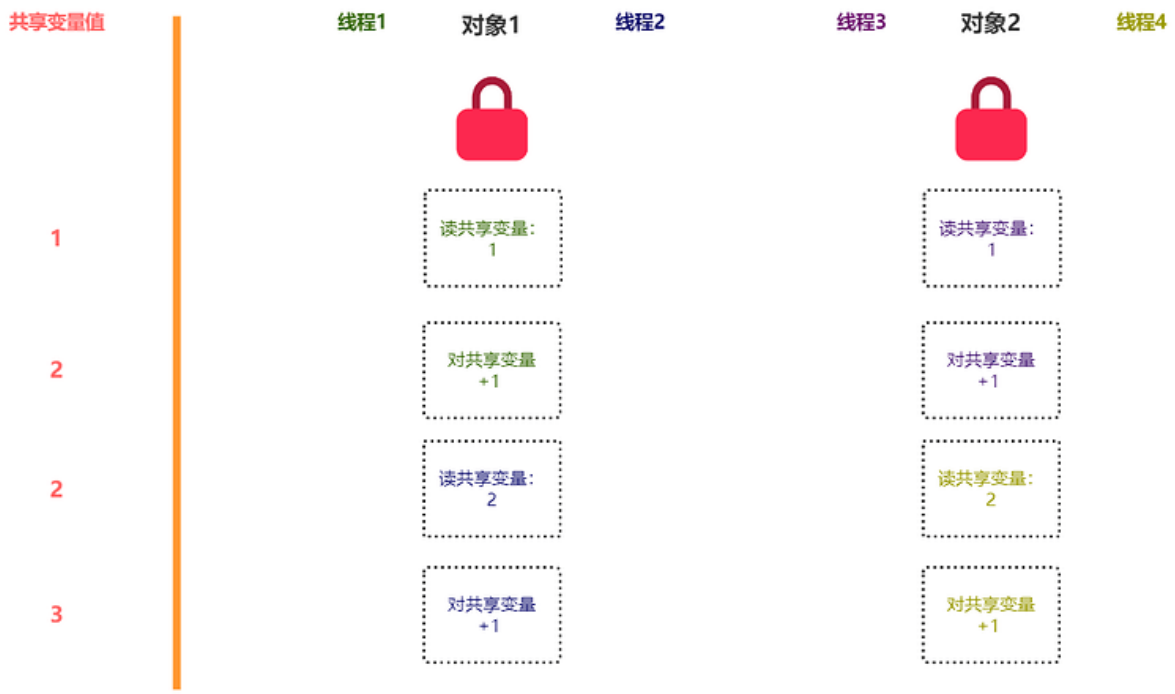
如果是 Java 程序，可以用 synchronized 关键字给写入操作加入互斥锁：

```
public class Logger {
```

```
  private FileWriter writer;

  public Logger() {

    // 创建文件

    File file = new File("/Users/wangzheng/log.txt");

   // 进行追加写入

    writer = new FileWriter(file, true);

  }

  public void log(String message) {

    // 为 write 加入互斥锁

    synchronized(this) {

      writer.write(mesasge);

    }

  }

}
```

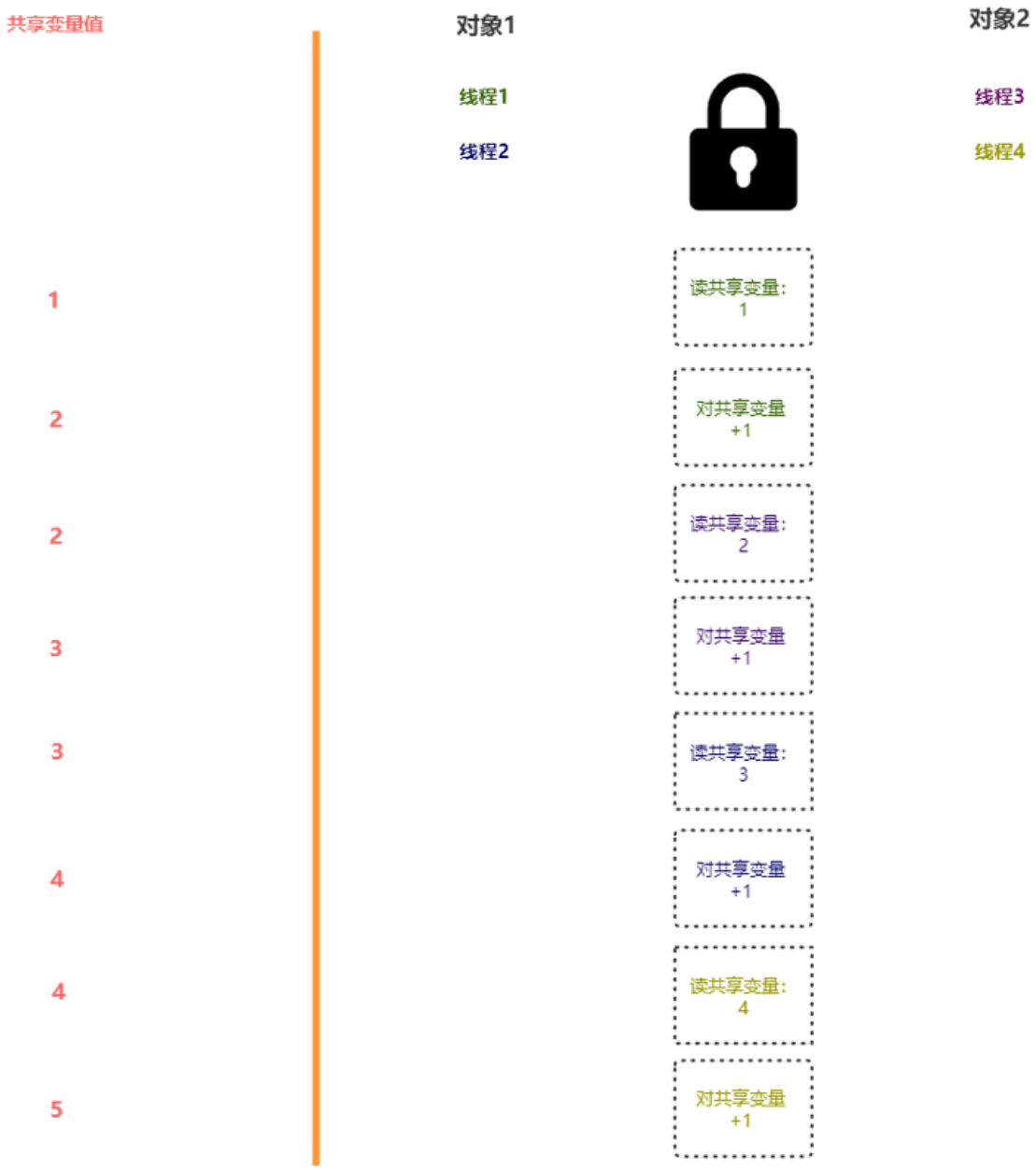
但是，这段代码存在两个问题：

1. FileWriter 本身就是对象安全的，代码多此一举，。
2. 这是对象锁，不同线程使用同一对象才有效，若使用不同对象，没有效（如果不理解，见下图）。



其实问题不难解决，把**对象级别锁提升到类级别**即可：

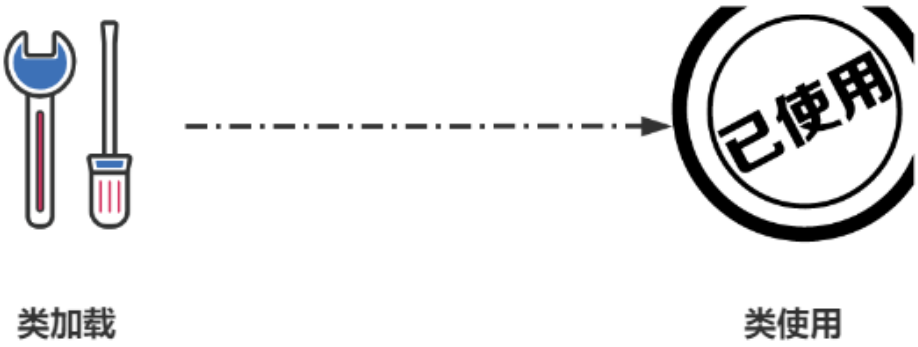
```
public class Logger {

  private FileWriter writer;

  public Logger() {

    File file = new File("/Users/wangzheng/log.txt");

    writer = new FileWriter(file, true);

  }

  public void log(String message) {

    // 使用类级别的锁

    synchronized(Logger.class) {

      writer.write(mesasge);

    }

  }

}
```



其实还有很多方案，分布式锁，并发队列等等，但**这些方案都比较复杂**，而**单例模式的思路简单清晰**。下面来实现多个单例。

饿汉式单例

顾名思义，饥饿难忍，在**类加载时就创建好实例**。



很多语言都支持在类加载前就初始化变量的操作，比如 Java 的静态变量和 Python 的类变量，利用这个特性就能实现饿汉式单例。

java 实现

```
public class IdMaker {

    // Java 静态属性只会加载一次，所以 instance 只会被初始化一次

    // 利用 Java 静态属性，可以避免多线程资源竞争问题

    // 饿汉式：静态属性，在类加载阶段，完成初始化

    private static IdMaker instance = new IdMaker();

    // ID 计数器，默认值是 -1

    private int id = -1;

    // 把 IdMaker 的构造函数，设为私有（用于阻止调用者实例化 IdMaker）

    private IdMaker() {

    }

    // 获取实例

    public static IdMaker getInstance() {

        return instance;

    }

    // 通过 ++ 操作，获取不一样的 ID 值
```

```
    public int getId() {
        id++;
        return id;
    }
}
class TestIdMaker {
    public static void main(String[] args) {
        // 获取唯一的 ID
        int id1 = IdMaker.getInstance().getId();
        int id2 = IdMaker.getInstance().getId();
        int id3 = IdMaker.getInstance().getId();
        System.out.println(id1);
        System.out.println(id2);
        System.out.println(id3);
        // 0 1 2
    }
}
```

python实现

```
class IdMaker:
    # python 的类变量会被多个类，实例共享
    __instance = None
    # __id 也是类变量，多个实例或类共享
    __id = -1
    # python 在类加载阶段，通过父类的 __new__ 创建实例，如果我们重写 __new__
    # 就不会调用父类的 __new__ ，就会调用我们写的 __new__ 创建实例
```

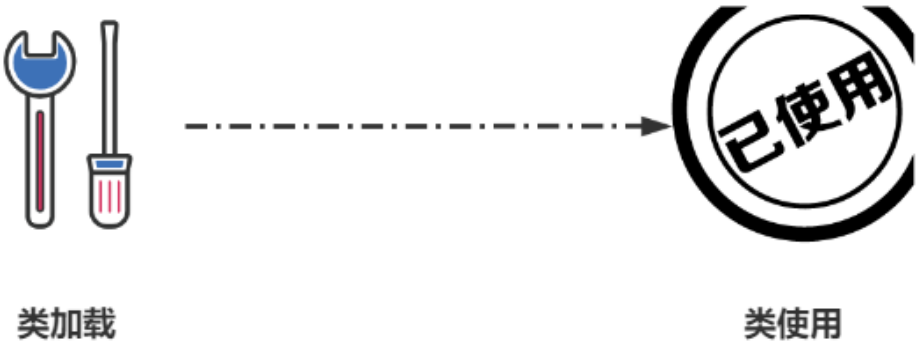
```
    # __new__ 需要返回一个实例，如果不返回，就不会实例化
    def __new__(cls):
        if cls.__instance is None:
            # 父类的 __new__ ，参数接收一个类名，会返回类的实例
            cls.__instance = super().__new__(cls)
        return cls.__instance
    # 计数器，在获取前，进行 + 1
    def get_id(self):
        self.__id += 1
        return self.__id
def test_id_maker():
    # IdMaker 是单例类，只允许有一个实例
    id1 = IdMaker().get_id()
    id2 = IdMaker().get_id()
    id3 = IdMaker().get_id()
    print(id1, id2, id3)
if __name__ == "__main__":
    test_id_maker()
    # 0 1 2
```

因为过于饥饿，该实现不支持延迟加载（用到的时候再初始化），如果实例占用资源多会造成初始化的慢，卡。其实，将占用资源多的方法提前（饿汉），有利有弊：

- 利：避免运行中卡顿，在初始化阶段就能发现错误
- 弊：提前初始化浪费资源

懒汉式单例

懒汉非常懒惰，在**类使用阶段才会创建实例**。



懒汉式可以弥补饿汉式缺点，由于在使用实例时才创建，避免初始化阶段卡慢。

Java 实现

```
public class IdMaker {

    // Java 引用类型的默认属性是 null

    // 在类加载阶段，不进行初始化

    private static IdMaker instance;

    // ID 计数器，默认值是 -1

    private int id = -1;

    // 把 IdMaker 的构造函数，设为私有（用于阻止调用者实例化 IdMaker）

    private IdMaker() {

    }

    // 懒汉式：在获取实例的阶段，进行初始化

    // synchronized 是互斥锁，为了保证在多线程时，只实例化一次

    public static synchronized IdMaker getInstance() {

        // 如果发现 instance 没有被初始化，就完成初始化

        if (instance == null)

            instance = new IdMaker();

        return instance;
```

```
    }
    // 通过 ++ 操作，获取不一样的 ID 值
    public int getId() {
        id++;
        return id;
    }
}
class TestIdMaker {
    public static void main(String[] args) {
        // 获取唯一的 ID
        int id1 = IdMaker.getInstance().getId();
        int id2 = IdMaker.getInstance().getId();
        int id3 = IdMaker.getInstance().getId();
        System.out.println(id1);
        System.out.println(id2);
        System.out.println(id3);
        // 0 1 2
    }
}
```

python 实现

```
from threading import Lock
class IdMaker:
    # 申请一个线程锁
    __instance_lock = Lock()
```

```
    # python 的类变量会被多个类，实例共享
    __instance = None
    # __id 也是类变量，多个实例或类共享
    __id = -1
    # 如果 __new__ 抛出异常，就不允许调用者进行实例化
    def __new__(cls):
        raise ImportError("Instantition not allowed")
    # 类方法不用实例化也能调用，因为我们不允许进行实例化，所以要使用类方法
    @classmethod
    def get_instance(cls):
        # with 会帮我们自动的上锁和释放，不用我们操心
        with cls.__instance_lock:
            if cls.__instance is None:
                # 因为我们的 __new__ 代码不允许进行实例化，所以可以借用父类的 __new__ 进行实例化
                cls.__instance = super().__new__(cls)
        return cls.__instance
    # 计数器，在获取前，进行 + 1
    def get_id(self):
        self.__id += 1
        return self.__id
def test_id_maker():
    # IdMaker 是单例类，只允许有一个实例
    id1 = IdMaker.get_instance().get_id()
    id2 = IdMaker.get_instance().get_id()
```

```
    id3 = IdMaker.get_instance().get_id()

    print(id1, id2, id3)

if __name__ == "__main__":

    test_id_maker()

    # 0 1 2
```

十、工厂设计模式

简介

当创建对象的代码多而杂时，可以用工厂模式将对象的创建和使用分离，让代码更加清晰。

比如下面代码根据不同的 rule 创建不同的对象。

java demo

```
public class Demo {

    // 解析文件，返回解析内容

    public void load(String rule) {

        IParse parse = null;

        // 根据不同的文件类型，执行不同的内容

        // 分析：如果代码中创建对象的过程很复杂，就需要把这段代码移出去，单独封装成类，封装的类就是工厂类

        if ("xml".equals(rule))

            parse = new XmlParse();

        else if ("json".equals(rule))

            parse = new JsonParse();

        else if ("excel".equals(rule))

            parse = new ExcelParse();

        else if ("csv".equals(rule))

            parse = new CsvParse();
```

```
        else
            parse = new OtherParse();
        // parse 是解析类之一，具体是哪一个，要根据 rule 来决定
        parse.parse();
        // 省略其它有关 load 的大量操作
    }
    public static void main(String[] args) {
        Demo demo = new Demo();
        // 传入 json
        demo.load("json");
    }
}
// 所有解析类的接口，规定解析类的方法
interface IParse {
    // 需要解析类去实现
    public void parse();
}
class XmlParse implements IParse {
    // 解析结果
    public void parse() {
        System.out.println("XmlParse");
    }
}
class JsonParse implements IParse {
    public void parse() {
```

```
        System.out.println("JsonParse");
    }
}
class ExcelParse implements IParse {
    public void parse() {
        System.out.println("ExcelParse");
    }
}
class CsvParse implements IParse {
    public void parse() {
        System.out.println("CsvParse");
    }
}
class OtherParse implements IParse {
    public void parse() {
        System.out.println("OtherParse");
    }
}
```

python demo

```
# Demo 用于加载不同的文件，对不同的文件作不同的处理
class Demo:
    def load(self, rule):
        parse = None
        # 根据不同的 rule ，创建不同的对象
```

```
        if "xml" == rule:
            parse = XmlParse()
        elif "json" == rule:
            parse = JsonParse()
        elif "excel" == rule:
            parse = ExcelParse()
        elif "csv" == rule:
            parse = CsvParse()
        else:
            parse = OtherParse()
        # 调用对象的方法进行操作
        parse.parse()
# 相当于接口，用于规范各个解析类
# 每个解析类都要实现 parse 方法，否则在调用的时候就会报错
class IParse:
    def parse(self):
        raise ValueError()
class XmlParse(IParse):
    def parse(self):
        print("XmlParse")
class JsonParse(IParse):
    def parse(self):
        print("JsonParse")
class ExcelParse(IParse):
    def parse(self):
```

```
        print("ExcelParse")
class CsvParse(IParse):
    def parse(self):
        print("CsvParse")
class OtherParse(IParse):
    def parse(self):
        print("OtherParse")
if __name__ == "__main__":
    Demo().load("json")
```

简单工厂

把创建大量实例的代码放到工厂类中。

java 实现

```
public class Demo {
    // 解析文件，返回解析内容
    public void load(String rule) {
        IParse parse = ParseFactory.createParse(rule);
        // parse 是解析类之一，具体是哪一个，要根据 rule 来决定
        parse.parse();
        // 省略其它有关 load 的大量操作
    }
    public static void main(String[] args) {
        Demo demo = new Demo();
        // 传入 json
        demo.load("json");
```

```
    }
}
// 简单工厂：把创建对象的代码移动到这个类中的一个方法，这个类就叫做简单工厂
// 简单工厂的类名字通常以 Factory 结尾
// 简单工厂的方法，通常叫 createParse
class ParseFactory {
    public static IParse createParse(String rule) {
        IParse parse = null;
        // 根据不同的文件类型，执行不同的内容
        if ("xml".equals(rule))
            parse = new XmlParse();
        else if ("json".equals(rule))
            parse = new JsonParse();
        else if ("excel".equals(rule))
            parse = new ExcelParse();
        else if ("csv".equals(rule))
            parse = new CsvParse();
        else
            parse = new OtherParse();
        return parse;
    }
}
    // 所有解析类的接口，规定解析类的方法
    interface IParse {
        // 需要解析类去实现
```

```
        public void parse();
    }
    class XmlParse implements IParse {
        // 解析结果
        public void parse() {
            System.out.println("XmlParse");
        }
    }
    class JsonParse implements IParse {
        public void parse() {
            System.out.println("JsonParse");
        }
    }
    class ExcelParse implements IParse {
        public void parse() {
            System.out.println("ExcelParse");
        }
    }
    class CsvParse implements IParse {
        public void parse() {
            System.out.println("CsvParse");
        }
    }
class OtherParse implements IParse {
```

```
    public void parse() {

        System.out.println("OtherParse");

    }

}
```

python 实现

```
# Demo 用于加载不同的文件，对不同的文件作不同的处理

# 问题：如果创建对象的代码比如多，可能还会创建 text ，md，yml 等等

# 简单工厂解决：把对象的创建移动到其它类中， load 方法就会很简洁

class Demo:

    def load(self, rule):

        parse = ParseRuleFactory().create_parse(rule)

        # 调用对象的方法进行操作

        parse.parse()

# 简单工厂类：用于实例的创建，根据 rule 创建不同的实例。本质就是把 Demo 中原来创建实例的代码，给迁移过来

class ParseRuleFactory:

    def create_parse(self, rule):

        parse = None

        # 根据不同的 rule ，创建不同的对象

        if "xml" == rule:

            parse = XmlParse()

        elif "json" == rule:

            parse = JsonParse()

        elif "excel" == rule:

            parse = ExcelParse()
```

```
        elif "csv" == rule:
            parse = CsvParse()
        else:
            parse = OtherParse()
        return parse


# 相当于接口，用于规范各个解析类
# 每个解析类都要实现 parse 方法，否则在调用的时候就会报错
class IParse:
    def parse(self):
        raise ValueError()
class XmlParse(IParse):
    def parse(self):
        print("XmlParse")
class JsonParse(IParse):
    def parse(self):
        print("JsonParse")
class ExcelParse(IParse):
    def parse(self):
        print("ExcelParse")
class CsvParse(IParse):
    def parse(self):
        print("CsvParse")
class OtherParse(IParse):
    def parse(self):
```

```
        print("OtherParse")

if __name__ == "__main__":

    Demo().load("json")
```

工厂方法

如果创建实例的代码非常复杂，就可以把创建实例的代码单独放入一个类。

比如下面的例子，创建实例代码复杂：

java 实现

```
public class Demo {

    // 如果创建对象时，代码很复杂，简单工厂就不能解决问题，因为即使使用简单工厂，创建实例依旧很复杂

    // 此时就需要工厂方法来解决、

    // 工厂方法解决方案：将每一个创建过程都封装到工厂类中

    // 比如下面的 JsonParseRuleFactory ，将复杂的代码都放到了 JsonParseRuleFactory 类中

    public void load(String rule) {

        IParse parse = null;

        if ("xml".equals(rule))

            // 省略了 1000 行代码

            parse = new XmlParse();

        else if ("json".equals(rule))

            parse = new JsonParseRuleFactory().createParse();

        else if ("excel".equals(rule))

            // 省略了 1000 行代码

            parse = new ExcelParse();

        else if ("csv".equals(rule))
```

```
            // 省略了 1000 行代码
            parse = new CsvParse();
        else
            // 省略了 1000 行代码
            parse = new OtherParse();
        // parse 是解析类之一，具体是哪一个，要根据 rule 来决定
        parse.parse();
        // 省略其它有关 load 的大量操作
    }
    public static void main(String[] args) {
        Demo demo = new Demo();
        // 传入 json
        demo.load("json");
    }
}
// 工厂方法的接口：要根据不同的调用，实现不同的操作
interface IParseRuleFactory {
    // createParse 接口要创建对象以及复杂操作
    public IParse createParse();
}
// 把原来 json 处理的所有代码，都迁移过来
class JsonParseRuleFactory implements IParseRuleFactory {
    public IParse createParse() {
        // 省略了 1000 行代码
        return new JsonParse();
```

```
    }
}
// 所有解析类的接口，规定解析类的方法
interface IParse {
    // 需要解析类去实现
    public void parse();
}
class XmlParse implements IParse {
    // 解析结果
    public void parse() {
        System.out.println("XmlParse");
    }
}
class JsonParse implements IParse {
    public void parse() {
        System.out.println("JsonParse");
    }
}
class ExcelParse implements IParse {
    public void parse() {
        System.out.println("ExcelParse");
    }
}
class CsvParse implements IParse {
```

```
    public void parse() {
        System.out.println("CsvParse");
    }
}
class OtherParse implements IParse {
    public void parse() {
        System.out.println("OtherParse");
    }
}
```

python 实现

```
# 问题：简单工厂不能解决创建实例的代码可能很复杂，即使迁移到了简单工厂中，复杂的创建过程依旧存在

# 解决：使用工厂方法，把创建过程封装到工厂类

class Demo:
    def load(self, rule):
        parse = None
        if "xml" == rule:
            # 省略 1000 行代码
            parse = XmlParse()
        elif "json" == rule:
            parse = JsonParseRuleFactory().create_parse()
        elif "excel" == rule:
            # 省略 1000 行代码
            parse = ExcelParse()
        elif "csv" == rule:
```

```
            # 省略 1000 行代码
            parse = CsvParse()
        else:
            # 省略 1000 行代码
            parse = OtherParse()
        # 调用对象的方法进行操作
        parse.parse()
# 相当于接口，用于规范各个工厂类
class IParseRuleFactory:
    def create_parse(self):
        raise ValueError()
# 工厂：把 Json 的解析放到此工厂下面
class JsonParseRuleFactory (IParseRuleFactory):
    def create_parse(self):
        # 省略 1000 行代码
        return JsonParse()
# 相当于接口，用于规范各个解析类
# 每个解析类都要实现 parse 方法，否则在调用的时候就会报错
class IParse:
    def parse(self):
        raise ValueError()
class XmlParse(IParse):
    def parse(self):
        print("XmlParse")
class JsonParse(IParse):
```

```
    def parse(self):
        print("JsonParse")
class ExcelParse(IParse):
    def parse(self):
        print("ExcelParse")
class CsvParse(IParse):
    def parse(self):
        print("CsvParse")
class OtherParse(IParse):
    def parse(self):
        print("OtherParse")
if __name__ == "__main__":
    Demo().load("json")
```

抽象工厂

假设有 A 公司， B 公司，C 公司... 都要有自己的解析方法，比如：A 公司有 AXmlParseFactory ， B 公司有 BXmlParseFactory ， C 公司有 CXmlParseFactory 。如果此时使用简单工厂和工厂方法，工作量非常大，因为要给每一个公司都创建所有的解析工厂。

此时可以用抽象工厂解决问题。解析工厂返回多个实例，比如，XmlParseFactory 可以返回 A ， B ， C 的实例。

java 实现

```
// 抽象工厂：使工厂类具有返回多个实例的功能
interface IParseRuleFactory {
    public IParse AcreateParse();
    public IParse BcreateParse();
    public IParse CcreateParse();
}
```

```
class XmlParseRuleFactory implements IParseRuleFactory{

    public IParse AcreateParse() {}

    public IParse BcreateParse(){}

    public IParse CcreateParse(){}

}
```

python 实现

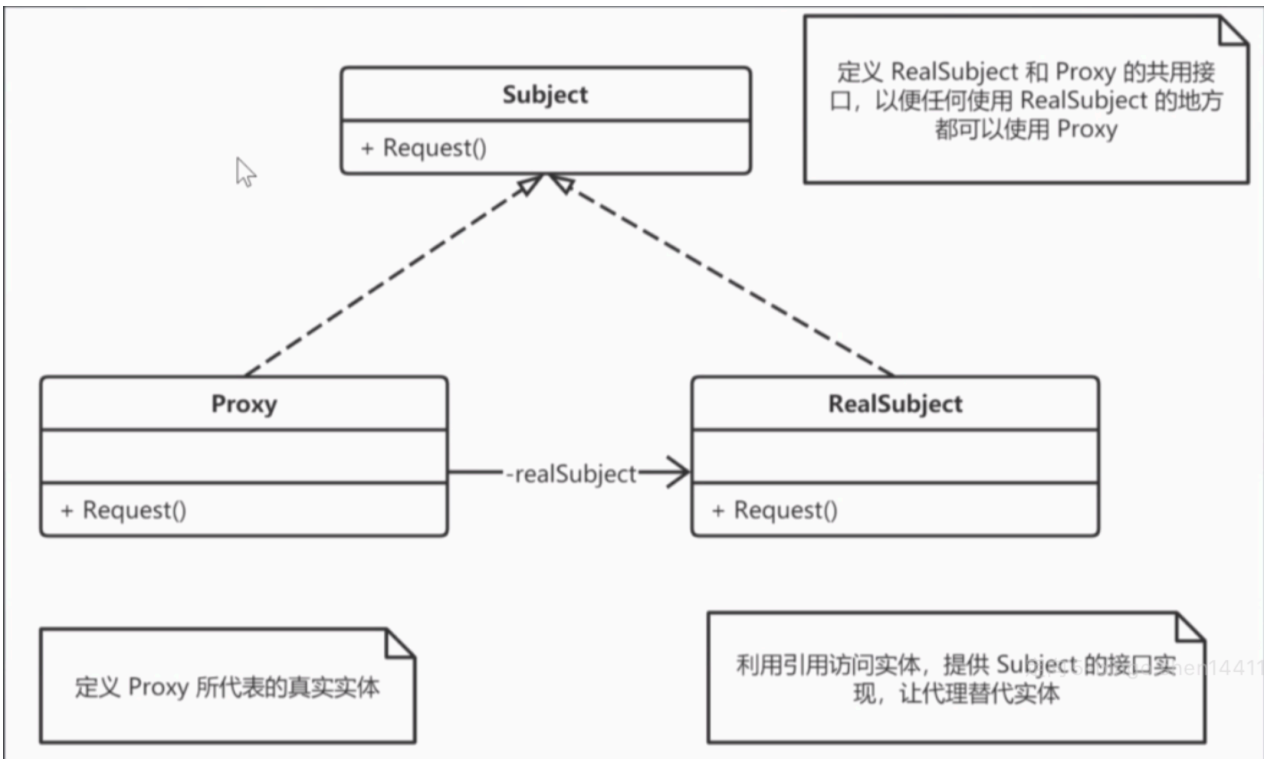
```
# 问题：如果多个公司都要封装工厂，比如 A, B, C ...公司都要封装自己的工厂，就要封装 n 个工厂类

# 解决：可以使用抽象工厂解决问题，每个工厂类可以创建多个实例，比如 JsonParseRuleFactory，可以创建 A, B, C 公司的实例

# 一个工厂类，可以生成多个公司的解析方法

class IParseRuleFactory:

    def a_create_parse(self):

        raise ValueError()

    def b_create_parse(self):

        raise ValueError()

    def c_create_parse(self):

        raise ValueError()



# 实现时候，一个工厂类就可以生成多个公司的实例

class JsonParseRuleFactory(IParseRuleFactory):

    def a_create_parse(self):

        """

        """

    def b_create_parse(self):
```

```
        """

        """


    def c_create_parse(self):

        """

        """
```

十一、代理设计模式（了解）

定义：为其它对象提供一种代理，以控制对这个对象的访问

代理模式UML图



代理模式应用

1.远程代理：在代码中加入Web引用

2.虚拟代理：存放实体化需要很久的对象，比如网页先看到文字，后看到图片

3.安全代理：控制实体对象的访问权限

4.智能指引：调用真实对象时，处理另外一些事，比如计算真实对象的引用次数

十二、设计模式基本原则

SOLID

英文名	中文	解释
Single Responsibility Principle	单一职责	一个类或者模块只负责完成一个职责（或者功能）
Open Closed Principle	开闭原则	对扩展开放、对修改关闭
Liskov Substitution Principle	里式替换	子类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏
Interface Segregation Principle	接口隔离原则	客户端不应该被强迫依赖它不需要的接口
Dependency Inversion Principle	依赖反转原则	高层模块不要依赖低层模块。高层模块和低层模块应该通过抽象来互相依赖。除此之外，抽象不要依赖具体实现细节，具体实现细节依赖抽象。

其它原则

简称	英文名	中文	解释
KISS	Keep It Simple and Stupid	尽量保持简单	保持简单
YAGNI	You Ain’t Gonna Need It	你不会需要它，不要做过度设计	
DRY	Don’t Repeat Yourself	不重复	不要编写重复的代码
LOD	Law of Demeter	迪米特法则	每个模块只应该了解那些与它关系密切的模块的有限知识