link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=136 sentences=244, words=1701

1.位运算

1.1 比特

- 比特(bit)是表示信息的最小单位
- 比特(bit)是二进制单位(binary unit)的缩写
- 比特(bit)只有两种状态：0和1
- 一般来说n比特的信息量可以表示出2的n次方种选择


```
0b1000=2*2*2=Math.pow(2,3)=8=0~7
```

1.2 数值数据

1.2.1 无符号数据的表示

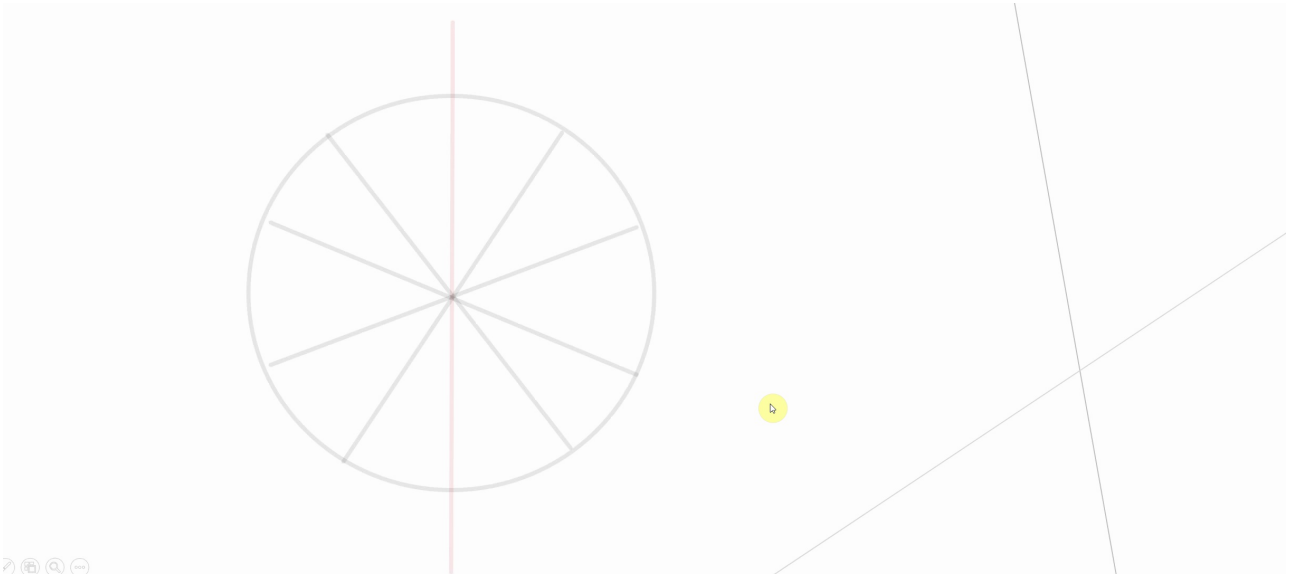
- 原码: 3个bit能表示8个数 0 1 2 3 4 5 6 7

1.2.2 有符号数据的表示

1.2.2.1 原码

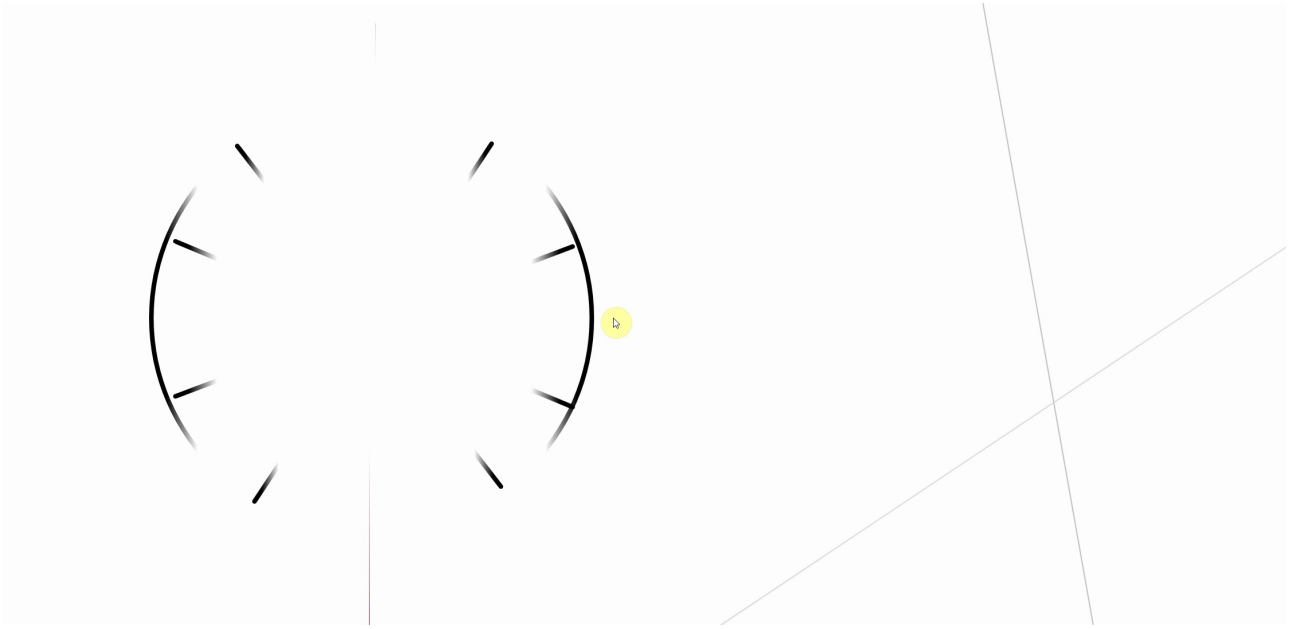
- 符号: 用0、1表示正负号,放在数值的最高位,0表示正数，1表示负数

- 原码: 3个bit能表示8个数 +0 +1 +2 +3 -0 -1 -2 -3



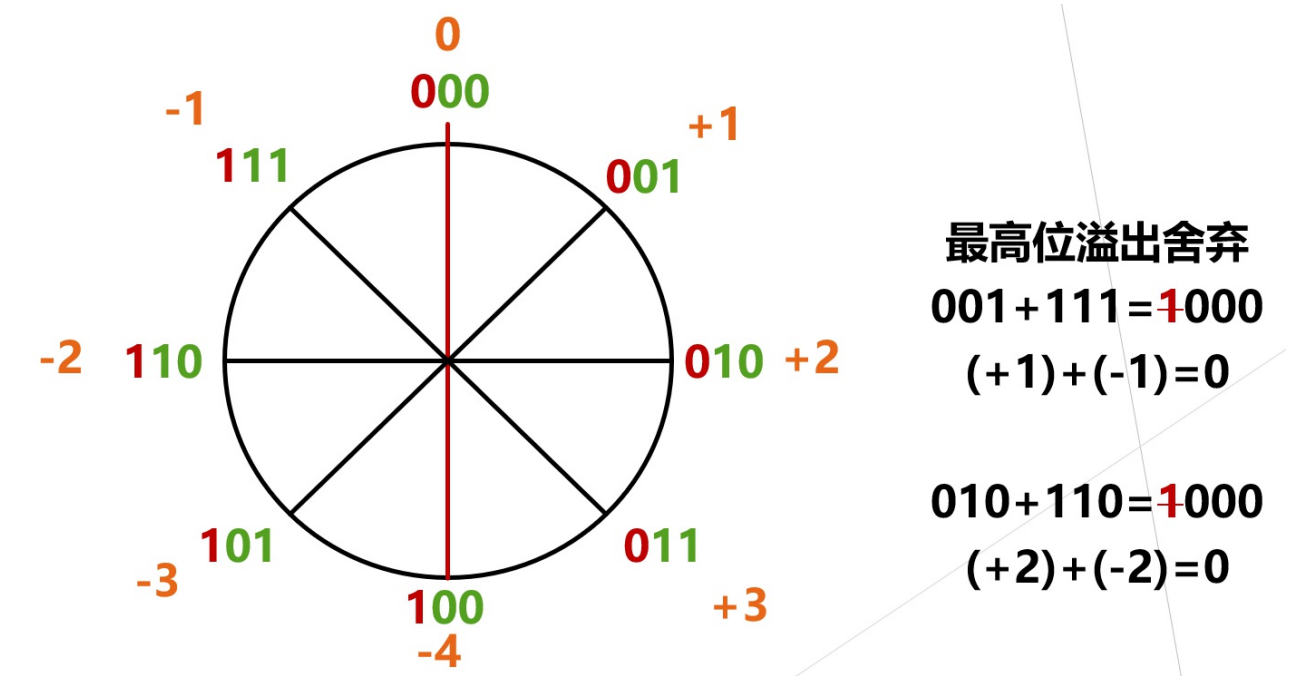
1.2.2.2 反码

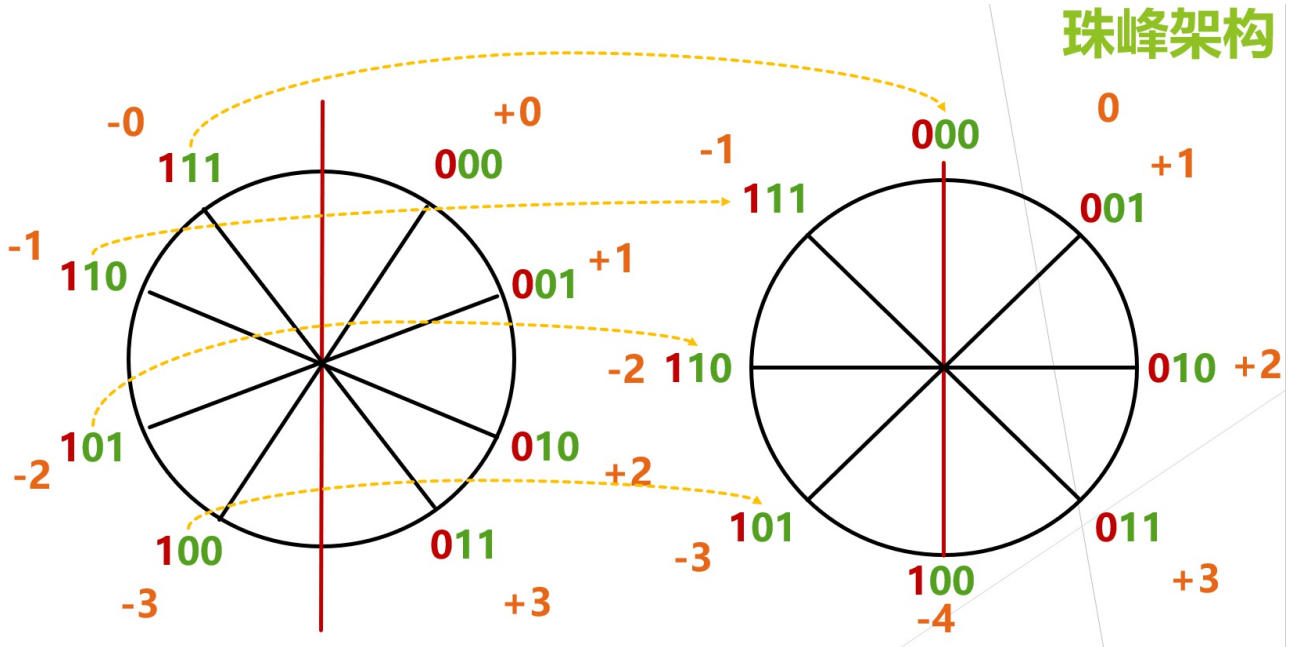
- 反码:正数不变,负数的除符号位外取反



1.2.2.3 补码

- 补码:正数不变,负数在反码的基础上加1
- 补码解决了,可以带符号位进行运算,不需要单独标识
- 补码解决了自然码正负0的表示方法
- 补码实现了减法变加法





1.3 位运算

- Binary Bitwise Operators (https://262.ecma-international.org/5.1/#sec-11.10)
- 按位与 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_AND)
- 按位或 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_OR)
- 按位异或 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR)
- 按位非 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Bitwise_NOT)
- 左移 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Left_shift)
- 有符号右移 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Right_shift)
- 无符号右移 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift)

运算 使用 说明 按位与(&) x & y 每一个比特位都为1时，结果为1，否则为0 按位或(

|

) x

|

y 每一个比特位都为0时，结果为0，否则为1 按位异或(^) x ^ y 每一个比特位相同结果为0，否则为1 按位非(~) ~ x 对每一个比特位取反，0变为1，1变为0 左移(<

```
let a = 0b100;
let b = 0b011;
console.log((a & b).toString(2));
console.log((a | b).toString(2));
console.log((a ^ b).toString(2));

console.log((~a));
console.log((~a).toString(2));
```

```
let a = 0b001;

console.log((a << 2).toString(2));

let b = 0b100;
console.log((b >>> 1).toString(2));

let c = 0b101;
console.log((-3 >> 1));
```

1.4 使用

```

const OP_INSERT = 1 << 0;
const OP_REMOVE = 1 << 1;

let OP = 0b000;

OP |= OP_INSERT;
OP |= OP_REMOVE;
console.log(OP.toString(2));

OP = OP & ~OP_INSERT;
console.log(OP.toString(2));

console.log((OP & OP_INSERT) === OP_INSERT);
console.log((OP & OP_REMOVE) === OP_REMOVE);

console.log((OP & OP_INSERT) === 0);
console.log((OP & OP_REMOVE) === 0);
```

1.5 ES5规范

- Binary Bitwise Operators (https://262.ecma-international.org/5.1/#sec-11.10)
- ToInt32: (Signed 32 Bit Integer) (https://262.ecma-international.org/5.1/#sec-9.5)
- 位运算只支持整数运算
- 位运算中的左右操作数都会转换为有符号32位整型, 且返回结果也是有符号32位整型
- 操作数的大小超过Int32范围(-2^31 ~ 2^31-1). 超过范围的二进制位会被截断, 取低位32bit

1.6 Hydration

- 水合反应（ hydrated reaction），也叫作水化
- 是指物质溶解在水里时，与水发生的化学作用,水合分子的过程
- 组件在服务器端拉取数据(水)，并在服务器端首次渲染
- 脱水: 对组件进行脱水，变成HTML字符串，脱去动态数据，成为风干标本快照
- 注水：发送到客户端后，重新注入数据(水)，重新变成可交互组件


这个是晒干之后的银耳
呈自然的淡黄色

1.7 React中的应用场景

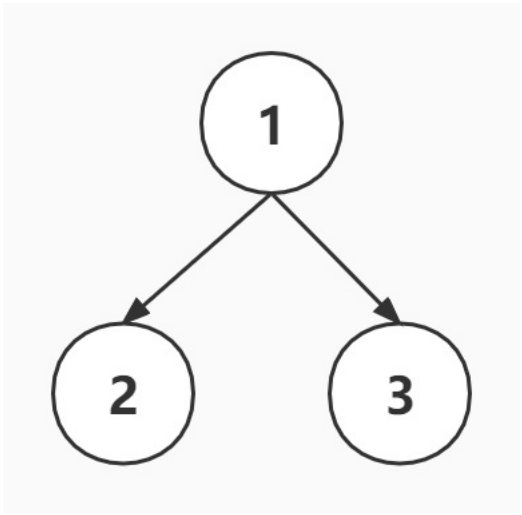
- React中用lane(车道)模型来表示任务优先级
- 一共有31条优先级，数字越小优先级越高，某些车道的优先级相同
- clz32 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Math/clz32())函数返回开头的 0 的个数

```
const SyncLanePriority = 15;
const SyncBatchedLanePriority = 14;

const InputDiscreteHydrationLanePriority = 13;
const InputDiscreteLanePriority = 12;

const InputContinuousHydrationLanePriority = 11;
const InputContinuousLanePriority = 10;

const DefaultHydrationLanePriority = 9;
const DefaultLanePriority = 8;

const TransitionHydrationPriority = 7;
const TransitionPriority = 6;
const RetryLanePriority = 5;
const SelectiveHydrationLanePriority = 4;

const IdleHydrationLanePriority = 3;
const IdleLanePriority = 2;

const OffscreenLanePriority = 1;

const NoLanePriority = 0;

const TotalLanes = 31;

const NoLanes = 0b0000000000000000000000000000000;
const NoLane = 0b0000000000000000000000000000000;

const SyncLane = 0b0000000000000000000000000000001;
const SyncBatchedLane = 0b0000000000000000000000000000010;

const InputDiscreteHydrationLane = 0b0000000000000000000000000000100;
const InputDiscreteLanes = 0b0000000000000000000000000011000;

const InputContinuousHydrationLane = 0b0000000000000000000000000100000;
const InputContinuousLanes = 0b0000000000000000000000011000000;

const DefaultHydrationLane = 0b0000000000000000000000100000000;
const DefaultLanes = 0b0000000000000000000111000000000;

const TransitionHydrationLane = 0b0000000000000000001000000000000;
const TransitionLanes = 0b0000000001111111110000000000000;

const RetryLanes = 0b0000011110000000000000000000000;
const SomeRetryLane = 0b0000010000000000000000000000000;

const SelectiveHydrationLane = 0b0000100000000000000000000000000;

const NonIdleLanes = 0b0000111111111111111111111111111;
const IdleHydrationLane = 0b0001000000000000000000000000000;

const IdleLanes = 0b0110000000000000000000000000000;

const OffscreenLane = 0b1000000000000000000000000000000;

function getHighestPriorityLane(lanes) {
    return lanes & -lanes;
}

function getLowestPriorityLane(lanes) {
    const index = 31 - Math.clz32(lanes);
    return index < 0 ? NoLanes : 1 << index;
}
console.log(getLowestPriorityLane(InputDiscreteLanes));
console.log('00000000000000000000000011000');
console.log(Math.clz32(InputDiscreteLanes));
console.log(31 - Math.clz32(InputDiscreteLanes));
```

2. 最小堆

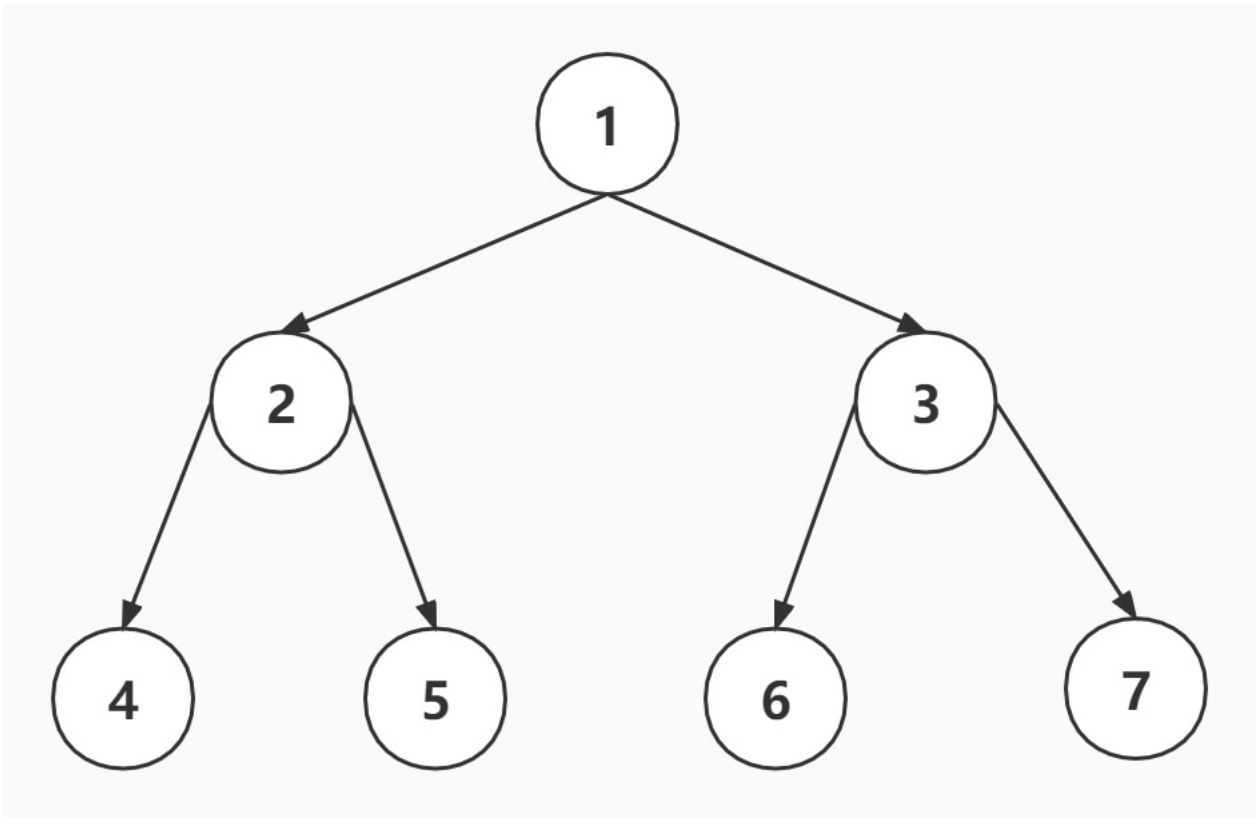
2.1 二叉树 #

- 每个节点最多有两个子节点



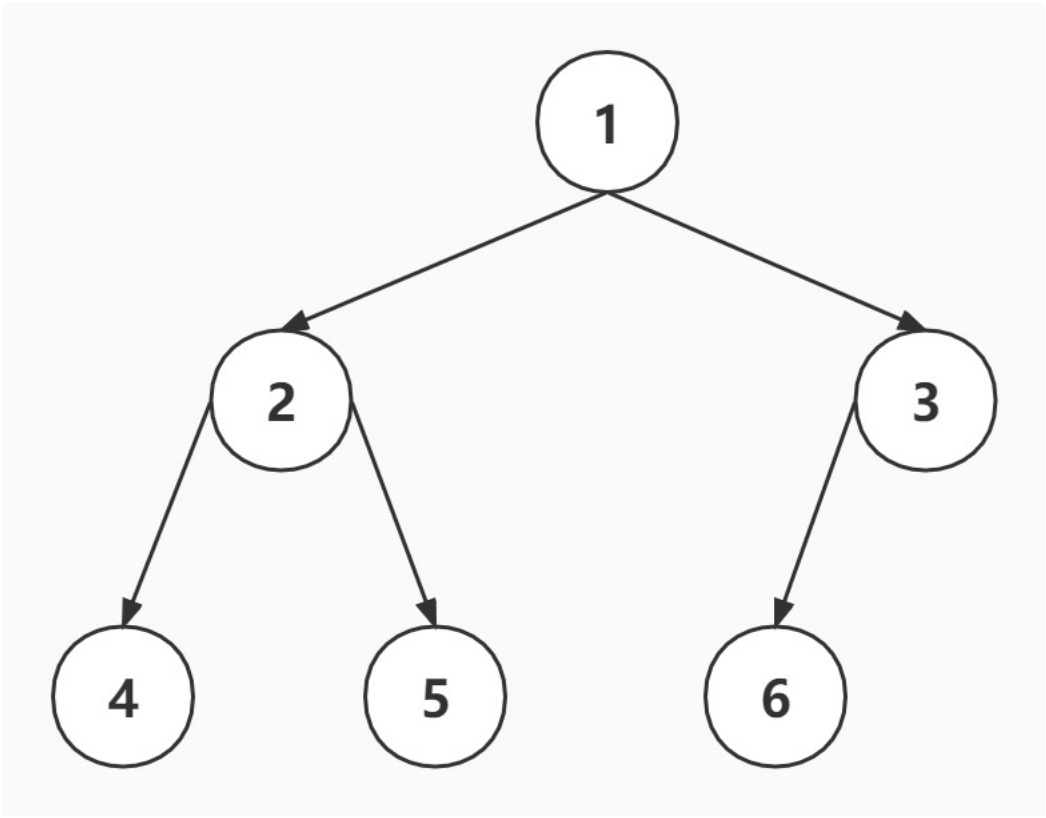
2.2 满二叉树 #

- 除最后一层无任何子节点外，每一层上的所有结点都有两个子结点的二叉树



2.3 完全二叉树 #

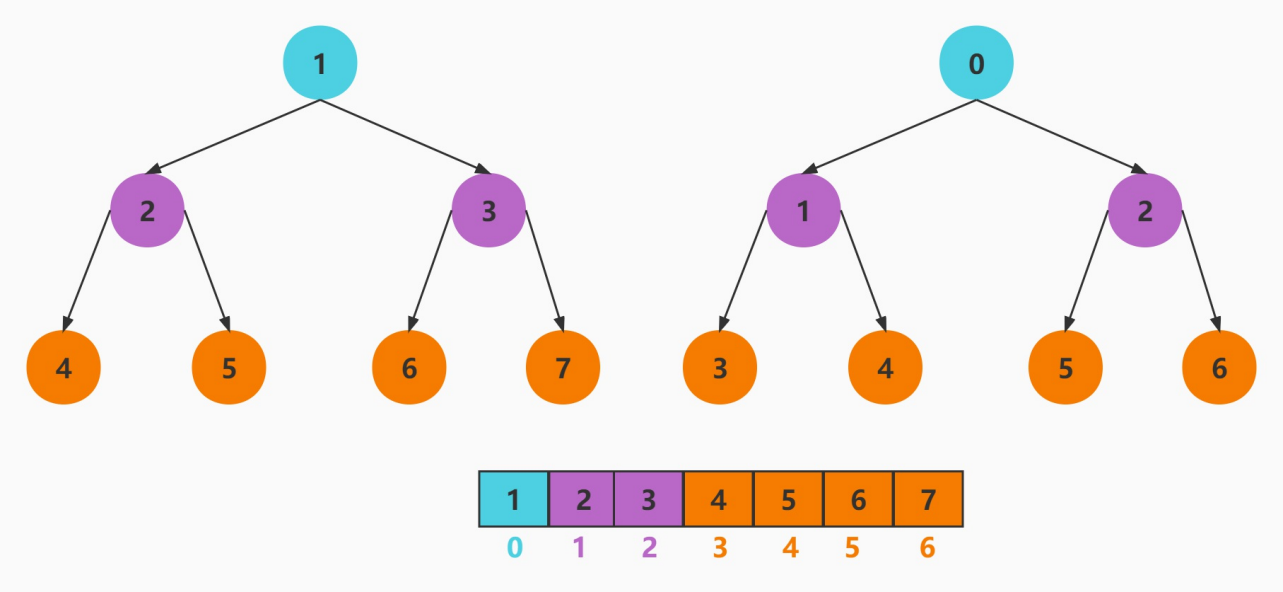
- 叶子结点只能出现在最下层和次下层
- 且最下层的叶子结点集中在树的左部



2.4 最小堆 #

- processon (https://www.processon.com/diagraming/61f26156e0b34d06c3b5bf48)
- 最小堆是一种经过排序的完全二叉树
- 其中任一非终端节点的数据值均不大于其左子节点和右子节点的值
- 根结点值是所有堆结点值中最小者
- 编号关系
    - 左子节点编号=父节点编号 _2 1_2=2
    - 右子节点编号=左子节点编号+1

    - 父节点编号=子节点编号/2 2/2=1
- 索引关系
    - 左子节点索引=(父节点索引+1) _2-1 (0+1)_2-1=1
    - 右子节点索引=左子节点索引+1
    - 父节点索引=(子节点索引-1)/2 (1-1)/2=0
- Unsigned_right_shift (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Unsigned_right_shift)



2.5 SchedulerMinHeap.js

- peek() 查看堆的顶点
- pop() 弹出堆的定点后需要调用 `siftDown`函数向下调整堆
- push() 添加新节点后需要调用 `siftUp`函数向上调整堆
- siftDown() 向下调整堆结构, 保证最小堆
- siftUp() 需要向上调整堆结构, 保证最小堆

react\packages\scheduler\src\SchedulerMinHeap.js

```
export function push(heap, node) {
  const index = heap.length;
  heap.push(node);
  siftUp(heap, node, index);
}
export function peek(heap) {
  const first = heap[0];
  return first === undefined ? null : first;
}
export function pop(heap) {
  const first = heap[0];
  if (first !== undefined) {
    const last = heap.pop();
    if (last !== first) {
      heap[0] = last;
      siftDown(heap, last, 0);
    }
    return first;
  } else {
    return null;
  }
}

function siftUp(heap, node, i) {
  let index = i;
  while (true) {
    const parentIndex = index - 1 >>> 1;
    const parent = heap[parentIndex];
    if (parent !== undefined && compare(parent, node) > 0) {
      heap[parentIndex] = node;
      heap[index] = parent;
      index = parentIndex;
    } else {

      return;
    }
  }
}

function siftDown(heap, node, i) {
  let index = i;
  const length = heap.length;
  while (index < length) {
    const leftIndex = (index + 1) * 2 - 1;
    const left = heap[leftIndex];
    const rightIndex = leftIndex + 1;
    const right = heap[rightIndex];
    if (left !== undefined && compare(left, node) < 0) {
      if (right !== undefined && compare(right, left) < 0) {
        heap[index] = right;
        heap[rightIndex] = node;
        index = rightIndex;
      } else {
        heap[index] = left;
        heap[leftIndex] = node;
        index = leftIndex;
      }
    } else if (right !== undefined && compare(right, node) < 0) {
      heap[index] = right;
      heap[rightIndex] = node;
      index = rightIndex;
    } else {
      return;
    }
  }
}

function compare(a, b) {
  const diff = a.sortIndex - b.sortIndex;
  return diff !== 0 ? diff : a.id - b.id;
}
```

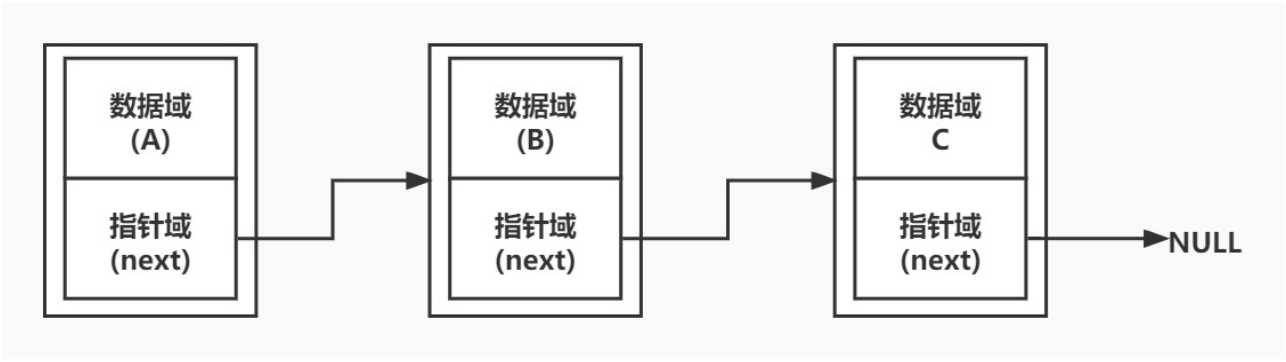
```
const { push, pop, peek } = require('./SchedulerMinHeap');
let heap = [];
push(heap, { sortIndex: 1 });
push(heap, { sortIndex: 2 });
push(heap, { sortIndex: 3 });
console.log(peek(heap));
push(heap, { sortIndex: 4 });
push(heap, { sortIndex: 5 });
push(heap, { sortIndex: 6 });
push(heap, { sortIndex: 7 });
console.log(peek(heap));
pop(heap);
console.log(peek(heap));
```

3. 链表

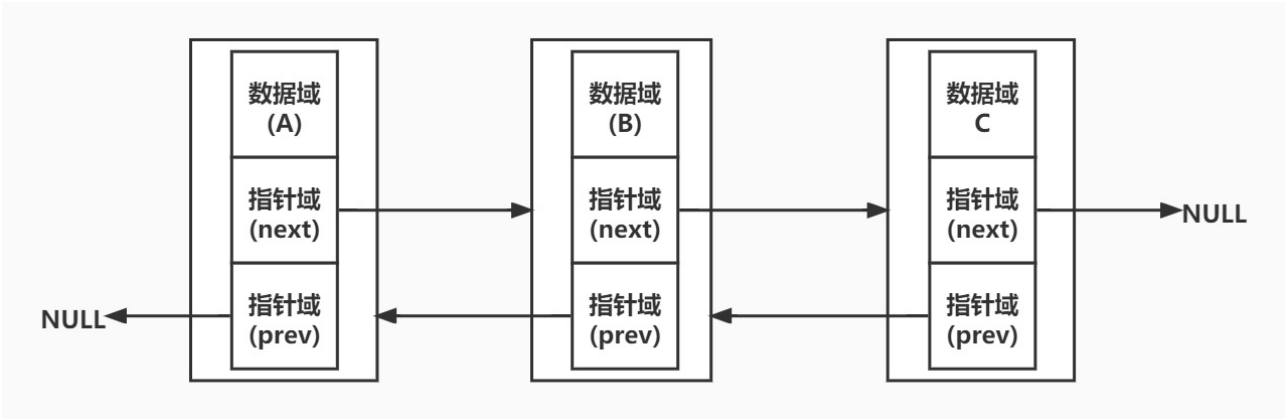
- 链表是一种物理存储单元上非连续、非顺序的存储结构
- 数据元素的逻辑顺序是通过链表中的指针链接次序实现的
- 链表由一系列结点组成
- 每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域

3.1 链表分类

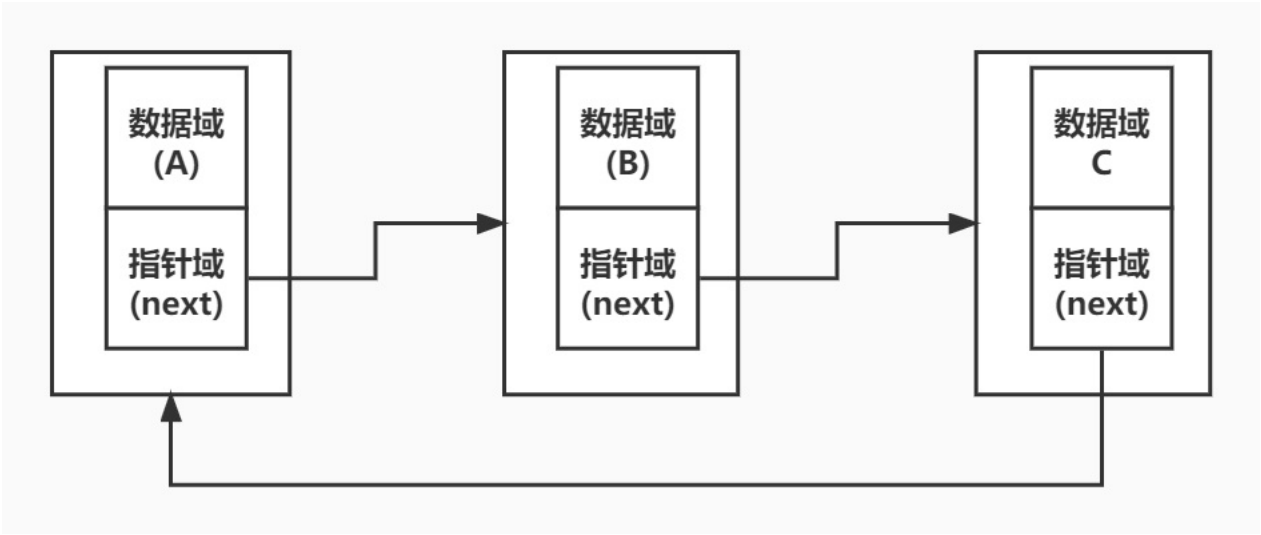
3.1.1 单向链表

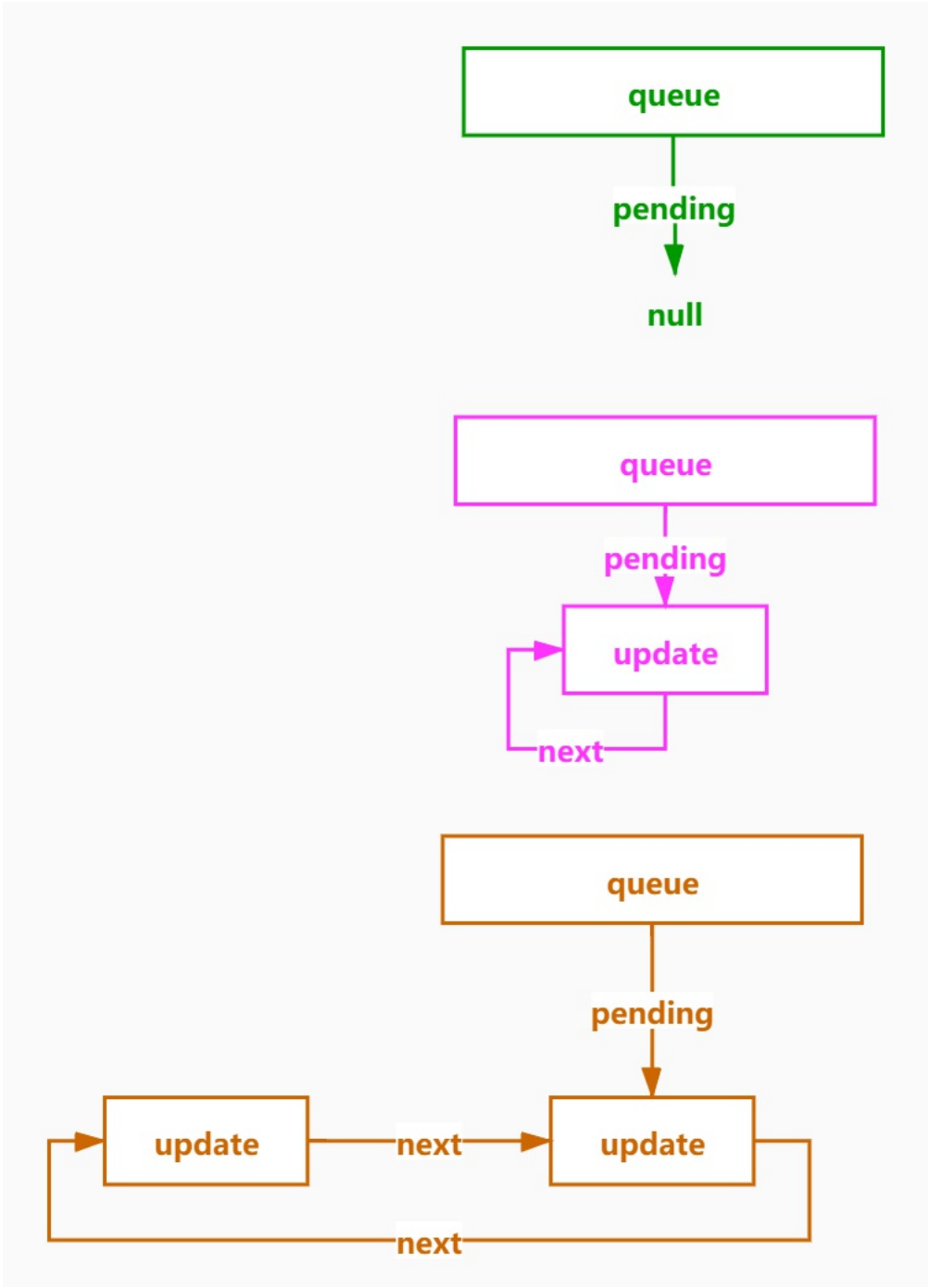


3.1.2 双向链表 #



3.1.3 循环链表 #





3.1.4 示例

3.1.4.1 ReactFiberLane.js

ReactFiberLane.js

```
const NoLanes = 0b00;
const NoLane = 0b00;
const SyncLane = 0b01;
const SyncBatchedLane = 0b10;
/**
 * 判断subset是不是set的子集
 * @param {*} set
 * @param {*} subset
 * @returns
 */
function isSubsetOfLanes(set, subset) {
    return (set & subset) === subset;
}
/**
 * 合并两个车道
 * @param {*} a
 * @param {*} b
 * @returns
 */
function mergeLanes(a, b) {
    return a | b;
}
module.exports = {
    NoLane,
    NoLanes,
    SyncLane,
    SyncBatchedLane,
    isSubsetOfLanes,
    mergeLanes
}
```

3.1.4.2 ReactUpdateQueue.js

ReactUpdateQueue.js

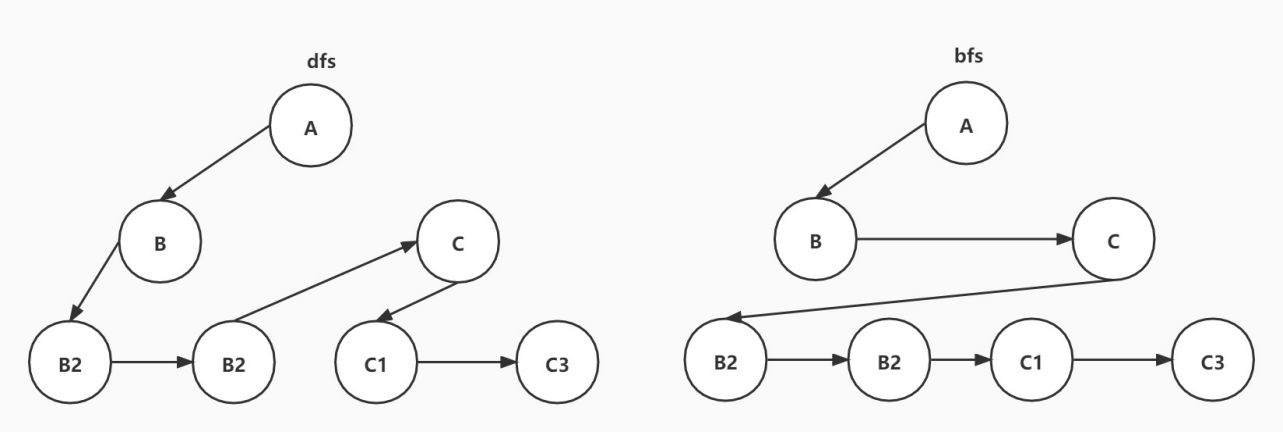
```
const { NoLane, NoLanes, isSubsetOfLanes, mergeLanes } = require('./ReactFiberLane') ;
function initializeUpdateQueue(fiber) {
    const queue = {
        baseState: fiber.memoizedState,
        firstBaseUpdate: null,
        lastBaseUpdate: null,
        shared: {

            pending: null
        }
    }
    fiber.updateQueue = queue;
}
function enqueueUpdate(fiber, update) {
    const updateQueue = fiber.updateQueue;
    if (updateQueue === null) {
        return;
    }
    const sharedQueue = updateQueue.shared;
    const pending = sharedQueue.pending;
    if (pending === null) {
        update.next = update;
    } else {
        update.next = pending.next;
        pending.next = update;
    }
    sharedQueue.pending = update;
}

function processUpdateQueue(fiber, renderLanes) {

    const queue = fiber.updateQueue;

    let firstBaseUpdate = queue.firstBaseUpdate;
    let lastBaseUpdate = queue.lastBaseUpdate;

    let pendingQueue = queue.shared.pending;
    if (pendingQueue !== null) {

        queue.shared.pending = null;

        const lastPendingUpdate = pendingQueue;

        const firstPendingUpdate = lastPendingUpdate.next;

        lastPendingUpdate.next = null;

        if (lastBaseUpdate === null) {
            firstBaseUpdate = firstPendingUpdate;
        } else {
            lastBaseUpdate.next = firstPendingUpdate;
        }

        lastBaseUpdate = lastPendingUpdate;
    }

    if (firstBaseUpdate !== null) {

        let newState = queue.baseState;
        let newLanes = NoLanes;
        let newBaseState = null;
        let newFirstBaseUpdate = null;
        let newLastBaseUpdate = null;
        let update = firstBaseUpdate;
        do {

            const updateLane = update.lane;

            if (!isSubsetOfLanes(renderLanes, updateLane)) {
```

```
                const clone = {
                    lane: updateLane,
                    payload: update.payload
                };
                if (newLastBaseUpdate === null) {
                    newFirstBaseUpdate = newLastBaseUpdate = clone;
                    newBaseState = newState;
                } else {
                    newLastBaseUpdate = newLastBaseUpdate.next = clone;
                }

                newLanes = mergeLanes(newLanes, updateLane);
            } else {

                if (newLastBaseUpdate !== null) {
                    const clone = {

                        lane: NoLane,
                        payload: update.payload
                    };
                    newLastBaseUpdate = newLastBaseUpdate.next = clone;
                }
                newState = getStateFromUpdate(update, newState);
            }
            update = update.next;
            if (!update) {
                break;
            }
        } while (true);

        if (!newLastBaseUpdate) {
            newBaseState = newState;
        }
        queue.baseState = newBaseState;
        queue.firstBaseUpdate = newFirstBaseUpdate;
        queue.lastBaseUpdate = newLastBaseUpdate;
        fiber.lanes = newLanes;
        fiber.memoizedState = newState;
    }
}

function getStateFromUpdate(update, prevState) {
    const payload = update.payload;
    let partialState = payload(prevState);
    return Object.assign({}, prevState, partialState);
}
module.exports = {
    initializeUpdateQueue,
    enqueueUpdate,
    processUpdateQueue
}
```

3.1.4.3 use.js

use.js

4. 树的遍历



4.1 深度优先(DFS)

- 深度优先搜索英文缩写为DFS即 Depth First Search
- 其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次
- 应用场景
    - React虚拟DOM的构建
    - React的fiber树构建

```
function dfs(node) {
    console.log(node.name);
    node.children&&node.children.forEach(child => {
        dfs(child);
    });
}
let root = {
    name: 'A',
    children: [
        {
            name: 'B',
            children: [
                { name: 'B1' },
                { name: 'B2' }
            ]
        },
        {
            name: 'C',
            children: [
                { name: 'C1' },
                { name: 'C2' }
            ]
        }
    ]
}
dfs(root);
```

4.2 广度优先(BFS)

- 宽度优先搜索算法（又称广度优先搜索），其英文全称是Breadth First Search
- 算法首先搜索距离为 k的所有顶点，然后再去搜索距离为 k+1的其他顶点

```
function bfs(node) {
    const stack = [];
    stack.push(node);
    let current;
    while (current = stack.shift()) {
        console.log(current.name);
        current.children && current.children.forEach(child => {
            stack.push(child);
        });
    }
}
let root = {
    name: 'A',
    children: [
        {
            name: 'B',
            children: [
                { name: 'B1' },
                { name: 'B2' }
            ]
        },
        {
            name: 'C',
            children: [
                { name: 'C1' },
                { name: 'C2' }
            ]
        }
    ]
}
bfs(root);
```

5. 栈

- 栈（stack）又名堆栈，它是一种运算受限的线性表
- 限定仅在表尾进行插入和删除操作的线性表，这一端被称为栈顶，相对地，把另一端称为栈底
- 向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素
- 从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素

5.1 栈代码

```
class Stack {
    constructor() {
        this.data = [];
        this.top = 0;
    }
    push(node) {
        this.data[this.top++] = node;
    }
    pop() {
        return this.data[--this.top];
    }
    peek() {
        return this.data[this.top - 1];
    }
    size() {
        return this.top;
    }
    clear() {
        this.top = 0;
    }
}

const stack = new Stack();
stack.push('1');
stack.push('2');
stack.push('3');
console.log('stack.size()', stack.size());
console.log('stack.peek', stack.peek());
console.log('stack.pop()', stack.pop());
console.log('stack.peek()', stack.peek());
stack.push('4');
console.log('stack.peek', stack.peek());
stack.clear();
console.log('stack.size', stack.size());
stack.push('5');
console.log('stack.peek', stack.peek());
```

5.2 应用场景

5.2.1 App.js

```
import React from "react";
const NumberContext = React.createContext(0);
export default function App() {
    return (
        <NumberContext.Provider value={'A'}>
            <NumberContext.Consumer>
                {(v1) => (
                    <NumberContext.Provider value={'B'}>
                        <NumberContext.Consumer>
                            {(v2) => (
                                <NumberContext.Provider value={'C'}>
                                    <NumberContext.Consumer>
                                        {(v3) => (
                                            <div>
                                                {v1} {v2} {v3}
                                            div>
                                        )}
                                    NumberContext.Consumer>
                                NumberContext.Provider>
                            )}
                        NumberContext.Consumer>
                    NumberContext.Provider>
                )}
            NumberContext.Consumer>
        NumberContext.Provider>
    );
}
```

5.2.2 ReactFiberStack.js

src\react\packages\react-reconciler\src\ReactFiberStack.js

```
const valueStack = [];
let index = -1;
function createCursor(defaultValue) {
    return {current: defaultValue};
}

function isEmpty() {
    return index === -1;
}

function pop(cursor) {
    if (index < 0) {
        return;
    }
    cursor.current = valueStack[index];
    valueStack[index] = null;
    index--;
}

function push(cursor, value) {
    index++;
    valueStack[index] = cursor.current;
    cursor.current = value;
}

module.exports =  {
    createCursor, isEmpty, pop, push
};
```

5.2.3 ReactFiberNewContext.js

src\react\packages\react-reconciler\src\ReactFiberNewContext.js

```
function pushProvider(providerFiber, nextValue) {
    const context = providerFiber.type._context;
    push(valueCursor, context._currentValue, providerFiber);
    context._currentValue = nextValue;
}
function popProvider(providerFiber) {
    const currentValue = valueCursor.current;
    pop(valueCursor, providerFiber);
    const context = providerFiber.type._context;
    context._currentValue = currentValue;
}
module.exports =  {
    pushProvider, popProvider
};
```

5.2.4 use.js

```
const { createCursor, pop, push } = require('./ReactFiberStack.js');
const { pushProvider, popProvider } = require('./ReactFiberNewContext.js');
let valueCursor = createCursor();
let providerFiber = { type: { _context: {} } };
pushProvider(providerFiber, 'A');
console.log(providerFiber.type._context._currentValue);
pushProvider(providerFiber, 'B');
console.log(providerFiber.type._context._currentValue);
pushProvider(providerFiber, 'C');
console.log(providerFiber.type._context._currentValue);
popProvider(providerFiber);
console.log(providerFiber.type._context._currentValue);
popProvider(providerFiber);
console.log(providerFiber.type._context._currentValue);
popProvider(providerFiber);
```

6.二进制

- 计算机用二进制来存储数字
- 为了简化运算，二进制数都是用一个字节(8个二进制位)来简化说明
- 在线工具 (unit.html)

6.1 真值

- 8位二进制数能表示的真值范围是[-2^8, +2^8]

```
+ 00000001 # +1
- 00000001 # -1
```

6.2 原码

- 由于计算机只能存储0和1，不能存储正负
- 所以用8个二进制位的最高位来表示符号，0表示正，1表示负，用后七位来表示真值的绝对值
- 这种表示方法称为原码表示法，简称原码
- 由于 10000000的意思是-0，这个没有意义，所有这个数字被用来表示-128
- 由于最高位被用来表示符号了，现在能表示的范围是[-2^7, +2^7-1]，即[-128, +127]

```
0 0000001 # +1
1 0000001 # -1
```

6.3 反码

- 反码是另一种表示数字的方法
- 其规则是整数的反码何其原码一样
- 负数的反码将其原码的符号位不变，其余各位按位取反
- 反码的表示范围是[-2^7, +2^7-1]，即[-128, +127]

```
0 0000001 # +1
1 1111110 # -1
```

6.4 补码

- 补码是为了简化运算，将减法变为加法而发明的数字表示法
- 其规则是整数的补码和原码一样，负数的补码是其反码末尾加1
- 8位补码表示的范围是[-2^7, +2^7-1]，即[-128, +127]
- 快速计算负数补码的规则就是，由其原码低位向高位找到第一个1，1和其低位不变，1前面的高位按位取反即可

```
0 0000001 # +1
1 1111111 # -1
```

6.5 二进制数整数

- 只要对js中的任何数字做位运算操作系统内部都会将其转换成整形
- js中的这种整形是区分正负数的
- js中的整数的表示范围是[-2^31, +2^31-1]，即[-2147483648, +2147483647]

6.6 ~非

- ~操作符会将操作数的每一位取反，如果是1则变为0，如果是0则边为1

```
0b00000011
3
~0b00000011 =>  0b11111100
-4
(~0b00000011).toString();
'-4'
(~0b00000011).toString(2);
'-100'

求补码的真值
1 表示负号
剩下的 1111100 开始转换
1111100 减1
1111011 取反
0000100 4
```

6.7 getHighestPriorityLane

- 可以找到最右边的1
- 最右边的1右边的全是0，全是0取反就全是1，再加上就会全部进位到1取反的位置
- 最右边的1和右边的数跟原来的值是完全一样的，左边的全是反的

```
function getHighestPriorityLane(lanes) {
    return lanes & -lanes;
}

lanes=0b00001100=12
-lanes=-12
1
0001100
1110011
1110100
11110100
00001100
```

6.8 左移

- 左移的规则就是每一位都向左移动一位，末尾补0，其效果相当于×2
- 计算机就是用移位操作来计算乘法的

```
(0b00000010<<1).toString(2)
```

6.9 >> 有符号右移

- 有符号右移也就是移位的时候高位补的是其符号位，整数则补0，负数则补1

```
(-0b111>>1).toString(2) "-100"
-0b111 -7
100000111 原码
111111000 反码
111111001 补码
111111100

1
111111100
111111011
000000100
1000000100
-100
-4
```

6.10 >>>无符号右移

- 右移的时候高位始终补0
- 整数和有符号右移没有区别
- 负数右移后会变为正数

```
(0b111>>>1).toString(2)
>>> "11"
```

```
(-0b111>>>1).toString(2)
>>> "11111111111111111111111111111100"

00000000000000000000000000000111
11111111111111111111111111111000
11111111111111111111111111111001
01111111111111111111111111111100
2147483644
```