Algorithm

437. Path Sum III

```
class Solution {
private:
    int count = 0;
public:
    int pathSum(TreeNode* root, int sum) {
        std::vector<int> arr;
        dfs(root, arr, sum);
        return count;
    }
private:
    void dfs(TreeNode* root, std::vector<int>& arr, int sum) {
        if(root == NULL) return;
        arr.push_back(0);
        for(auto& i : arr){
            i += root->val;
            if(i == sum) count++;
        }

        dfs(root->left, arr, sum);
        dfs(root->right, arr, sum);

        for(auto &i : arr){
            i -= root->val;
        }
        arr.pop_back();
    }
};
```

这道题是考察的如何在一个二叉树中寻找连续节点，其和是等于特定值，这里需要判断有多少种可能性。可以利用一个中间结构，每添加一个新元素就在这个维护的列表上累加一次，其列表中的值就表示每次加入一个节点后历史上所有可能节点的和，然后查看哪些是满足条件的。

这里也可以做优化，用一个multiset来保存和的中间结果，当有新节点到来时，判断当前multiset中`target - root->val`存在数量，表明当前存在的可能性，然后把当前节点累加值插入到multiset中。

这里每次结束都要把当前节点退出来，并消除累加值，退回上一个节点，从另一个孩子节点开始递归。

Review

Kubernetes 101: Pods, Nodes, Containers, and Clusters

kubernetes作为云计算时代的软件部署及管理的新一代标准，这篇文章讲述了kubernetes软件中的几个重要的组成部分，是一篇很好的入门文章。

- 硬件部分
  - 节点：节点是kubernetes计算硬件中的最小节点。代表集群中最小的一个计算单元。它可能是计算中心一台计算机，也可能是云服务器上一台虚拟主机。把物理机器抽象成一组由cpu，内存组成资源，可以使得我们能在其上抽象层，更好的统一管理。在kubernetes中，一个节点是能够用另一个节点替换的。
  - 集群：在kubernetes中，物理机是通过集群来发挥作用的，不用担心某个节点的状态。当我们把程序部署在集群之上，那么kubernetes会自动把任务在各个节点上分发。如果某个节点移除或者添加，kubernetes会把任务在集群内转移。
  - 持久卷：由于集群(cluster)是一个动态变化的状态，程序与节点之间没有固定的对应关系。因此，运行数据不能随意的存放在任意位置。比如程序将数据存储在特定位置，当某个机器因为故障下线时，这时就会发生数据丢失。因此，可以把存储持久卷当成一中文件系统，挂载在某个集群之上，这样就实现了对某个特定节点的解耦。
- 软件部分
  - 容器：在kubernetes上运行的程序被打包成linux容器。容器是一种被广泛接受的标准，也有很多编译好的容器可供直接使用。容器允许将程序及其运行环境打包成一个单独的文件而共享。同时，容器可以提供一致的开发运行环境，使得不用维护多套环境。
  - Pods：pods是kubernetes中最小的运行单元，pod可以将若干个容器打包成一个更高层次的资源。同一个pod内的容器将共享网络和资源。pod最为kubernetes中管理的最小单元，根据设定可以有多个副本同时提供服务，kubernetes可以根据设定在集群内运行多个副本。
  - Depolyments: 虽然pod是kubernetes中基础单元，但是kubernetes并不直接运行pod，而是在其之上通过deployment来管理pod。deployments会根据需要动态创建或者关掉对应的副本，利用deployment，不需要手动去关注pod的状态，只需要指定需要的状态，deployment会自动完成这些。

Tech

动态规划(1) 入门

0-1背包问题

```
//回溯算法实现
private int maxW = Integer.MIN_VALUE;
private int[] weight = {2, 2, 4, 6, 3};
private int n = 5; //最大个数
private int w = 9; //背包能装最大重量
public void f(int n, int cw){ //调用f(0,0)
    if(cw > maxW) maxW = cw;
    return;
}
f(i+1, cw); //选择不装第i个物品
if(cw + weight[i] <= w){
    f(i+1, cw+weight[i]); //选择第i个物品
}
```

递归树如下

```
                    f(0,0)
                    /    \
                f(1,0)    f(1,2)
                /    \      /    \
            f(2,0)  f(2,2) f(2,2)  f(2,4)
            。。。。。。。。。。。。。。。。
```

递归树可以看到会有重复计算节点，因此可以优化：

```
//回溯算法实现
private int maxW = Integer.MIN_VALUE;
private int[] weight = {2, 2, 4, 6, 3};
private int n = 5; //最大个数
private int w = 9; //背包能装最大重量
private boolean[][] mem = new boolean[5][10]
public void f(int n, int cw){ //调用f(0,0)
    if(cw > maxW) maxW = cw;
    return;
}
if(mem[i][cw]) return; //重复状态
mem[i][cw] = true; //记录(i, cw)状态
f(i+1, cw); //选择不装第i个物品
if(cw + weight[i] <= w){
    f(i+1, cw+weight[i]); //选择第i个物品
}
```

动态规划思路： 把问题分为多个阶段，每个阶段对应一个决策。记录每一个阶段可达的状态集合，然后通过当前阶段的状态集合，来推导下一个阶段的状态集合，动态地推进。

对于背包问题，可以设定一个二维数组 `bool state[i][cw]` 其中i表示第几个物品，cw表示重量。考虑第一个物品，有两种可能放置或不放置，形成两个状态，考虑第二个元素，前两个状态的基础之上形成第二行的状态，依次进行到最后一个元素，然后考察state状态表中最大的元素即为能容纳的最大容量。代码如下：

```
public int knapsack(int[] weight, int n, int w){ //n,物品数量，w:可承载重量， weight：物品重量
    boolean[][] states = new boolean[n][w+1];
    states[0][0] = true;
    if(weight[0] < w){
        states[0][weight[0] = true;
    }

    for(int i = 1; i < n; ++i){
        for(int j = 0; j <= w; ++j){ //不把第i个放入
            if(states[i-1][j]) states[i][j] = true;
        }

        for(int j = 0; j <= w - weight[i]; ++j){ //把第i个物品放入背包
            if(staes[i-1][j]) states[i][j+weight[i]] = true;
        }
    }
    for(int i = w; i >= 0; --i){
        if(state[n-1][i]) return i;
    }
    return 0;
}
```

Tech

读写文件的过程

读文件

1. 进程调用库函数向内核发起读文件请求
2. 内核通过检查进程的文件描述符定位到虚拟文件系统(VFS)的已打开文件列表表项
3. 调用该文件可用的系统调用函数read(),read()函数通过文件下表详连接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的inode。
4. 在inode中，通过文件内容偏移量计算要读取的页。
5. 通过inode找到文件对应的address_space
6. 在address_space中访问该文件的页缓存树，查找对应页缓存节点。
    - 命中缓存，直接返回文件内容
    - 缺失，产生一个缺页中断，同时通过inode找到该文件页的磁盘地址。读取相应的页填充该缓存页；从新在上一步查找缓存
7. 读取成功

写文件

前5步和读文件一致，在address_space中查询对应页的页缓存是否存在。

6. 如果页缓存命中，直接把文件内容修改更新在页缓存的页中，这时候文件修改位于页缓存，并没有写回磁盘文件。
7. 缺失，则产生缺页中断，创建一个页缓存页，返回第6步操作。
8. 被修改则被标记成脏页，通过两种方式写回：
    - 手动调用sync()或者fsync()系统调用把脏页写回
    - pdflush进程会定时把脏页写回到磁盘。