动态规划怎么又来了?

今日目标:

1:完成背包系列题目

2:完成打家劫舍系列题目

背包系列

要论经典的动态规划问题，首先想到的就是背包问题，其实背包又分很多种，大多数人首先遇到的其实是背包中的0-1背包。

1、0-1背包

1.1、问题描述

给你一个可放总重量为 W 的背包和 N 个物品，对每个物品，有重量 w 和价值 v 两个属性，那么第 i 个物品的重量为 w[i]，价值为 v[i]。现在让你用这个背包装物品，每种物品可以选0个或1个，问最多能装的价值是多少?

```
示例:

输入: W = 5, N = 3
     w = [3, 2, 1], v = [5, 2, 3]
输出: 8
解释: 选择 i=0 和 i=2 这两件物品装进背包。它们的总重量 4 小于 W，同时可以获得最大价值 8。
```

> 创建: `com.itheima.dp.BackPack`，便于实现

1.2、算法分析和实现

1、首先想能否对w和v排序?

答案是不能，因为一旦重新排序后，w[i] 和 v[i] 对应的不一定是同一个物品了。

2、一看到最多的价值，判断出这是一个最优化问题，首先想到贪心。那么贪心算法的局部最优能解决我们的问题吗?

事实上不太能，因为如果按照贪心算法来解的话，每次都找价值最大的，且重量在合理范围内。因为可能涉及到要对价值数组v进行排序，而本题如果排序是不可行的。另外贪心的局部最优不见得能保证全局最优。

3、要获得整体最优解，我们貌似只能进行穷举，既然涉及到穷举我们就要优先想是否能用动态规划来解决这个问题。因此首先判断该题是否满足动态规划的特征呢?

> 1:是否具有重叠子问题，对于 0-1 背包问题来说，即便我们不画出求解树，也能很容易看出在穷举的过程中存在重复计算的问题。这是因为各种排列组合间肯定存在重叠子问题的情况
>
> 2:无后效性:当我们选定了一个物品后，它的重量与价值就随即确定了，后续选择的物品不会对当前这个选择产生副作用。因此，该问题无后效性;

3：最优子结构：当我们选定了一个物品后，继续做决策时，我们是可以使用之前计算的重量和价值的，也就是说后续的计算可以通过前面的状态推导出来。因此，该问题存在最优子结构。

4、既然可以用动态规划解决解决，下面按照动态规划是思路来求解

1：寻找状态参数，

在原问题和子问题之间发生变化的变量，

```
当我们将某个物品 i 放入 背包中后，
1、背包内物品的数量n在增加，可选择的物品在减少，它是一个变量；n增加到题设条件N时可选择物品为0，

2、背包的重量在增加，即背包剩余还能装下的重量w在减少，它也是一个变量；w减到0意味着不能在装物品了
```

因此，当前背包内的物品数量 N 和背包还能装下的重量 W 就是这个动态规划问题的状态参数。

2、如何决策/选择，决策无非就是该不该把当前这个物品放入背包中：放怎么样，不放怎么样

有了状态和选择，拿以前的框架先套一下

```
for 状态1 in 状态1的所有取值：
 for 状态2 in 状态2的所有取值：
     for ...
         dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

3、**明确dp数组的定义**

dp数组是什么？其实就是用于存储状态信息的，存储的值就是我们想要的结果（一般就是题目要求返回的数据）。

现在的状态有两个，也就是说我们需要一个二维 `dp` 数组，一维表示可选择的物品，一维表示背包的容量。

`dp[i][w]` **的定义如下：对于前 i 个物品，当前背包的容量为 w，这种情况下可以装的最大价值是** `dp[i][w]`。

```
比如说，如果 dp[3][5] = 8，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 8。

为什么要这么定义？便于状态转移，或者说这就是套路，记下来就行了。
```

W=5, N=3

w={3, 2, 1}　v={5, 2, 3}

背包容量w

		0	1	2	3	4	5
物品数量i	0	0	0	0	0	0	0
	1	0	0	0	5	5	5
	2	0	0	2	5	5	7
	3	0	3	3	5	8	8

根据这个定义，我们想求的最终答案就是`dp[N][W]`。

此时可以细化我们的动归框架：

```
int dp[N+1][W+1]

for i in [1..N]:
 for w in [1..W]:
     dp[i][w] = max( 把物品 i 装进背包 , 不把物品 i 装进背包 )
return dp[N][W]
```

4、寻找初始化状态，

任何穷举算法（包括递归在内）都需要一个终止条件，这个所谓的终止条件，就是我们在动态规划解法当中的最初子问题，因此我们将其称作初始化状态。

```
在 0-1 背包中，这个终止条件是什么呢？
1、当背包的剩余容量为0 或者 物品的数量为0 时要终止执行。

即：不选物品（物品数量为0），或者 背包容量为0

初始化状态就是d[0][..] = dp[..][0] = 0，因为没有物品或者背包没有空间的时候，能装的最大价值就是0。
```

```
//初始条件：
dp[0][..] = 0
dp[..][0] = 0
```

5、**思考状态转移逻辑**

简单说就是，上面伪码中「把物品`i`装进背包」和「不把物品`i`装进背包」怎么用代码体现出来

刚刚分析到：`dp[i][w]`表示：对于前`i`个物品，当前背包的容量为`w`时，这种情况下可以装下的最大价值是`dp[i][w]`

1、**如果你没有把这第 i 个物品装入背包**，那么很显然，最大价值`dp[i][w]`应该等于`dp[i-1][w]`。你不装嘛，那就继承之前的结果，即前（i-1）个物品在当前容量w下的最大价值。

```
不把该物品i放入背包的原因是：
1、背包剩余的容量 < 该物品i的重量，物品i想放放不进去，只能不装
2、背包剩余的容量 > 该物品i的重量，物品i可以放进去，但可以选择不装进去
```

2、**如果你把这第i个物品装入了背包**，那么最大价值`dp[i][w]`应该等于`dp[i-1][ w - w[i-1]] + v[i-1]`。

```
其中不太好理解的就是：`dp[i-1][ w-w[i-1] ]  和 v[i-1]`

现在我们想装第i个物品，并计算这时候的最大价值，我们去找它的子问题
显然，我们应该寻求前`i-1`个物品在剩余重量`w-w[i-1]` 限制下能装的最大价值，加上第i个物品的价值`v[i-1]`，这就是装第i个物品后背包可以装的最大价值。

w数组和v数组的下标都取`i-1`的原因是`i`是从1开始的。
```

综上就是两种选择，我们都已经分析完毕，也就是写出来了状态转移方程，可以进一步细化代码：

```
for i in [1..N]:
 for w in [1..W]:
     dp[i][w] = max( dp[i-1][w] , dp[i-1][w - w[i-1]] + v[i-1] )
return dp[N][W]
```

6、把状态转移方程翻译成代码，并处理细节

```java
package com.itheima.dp;

/**
 * Created by 传智教育*黑马程序员.
 */
public class BackPack {


    public int dp(int[] wt,int[] v,int N,int W) {
        //1.创建dp数组
        int[][] dp = new int[N+1][W+1];
        //2.定义初始化状态
        //2.1.dp[0][..] = 0 物品个数0,背包有再大的容量最大价值也是0
        for (int i=0;i<W+1;i++) {
             dp[0][i] = 0;
        }
        //2.2.dp[..][0]=0 背包容量为0,有再多的物品最大价值也是0
        for (int i=0;i<N+1;i++) {
            dp[i][0] = 0;
        }

        //3.套模板,每种状态的每个取值都取值一遍 从子问题开始
        for (int i=1;i<N+1;i++) { //物品个数从[1,N]
            for (int w=1;w<W+1;w++) { //背包容量从[1,W]
                //4.根据状态转移方程找最优解
```

```java

                if (wt[i-1] > w) {
                    //4.1 物品i的重量>背包容量，物品i放不进去只能选择不装进去
                    dp[i][w] = dp[i-1][w];
                }else {
                    //4.2 装进去和不装进去两种择优。 放入物品i后的最大价值= 前(i-1)个物品在(w - wt[i-1])限制下的最大价值 + 当前物品i的价值v[i-1]
                    dp[i][w] = Math.max(dp[i-1][w],dp[i-1][w-wt[i-1]] + v[i-1]);
                }

            }
        }

        return dp[N][W];
    }

    public static void main(String[] args) {
        int N = 3, W = 5; // 物品的总数，背包能容纳的总重量
        int[] w = {3, 2, 1}; // 物品的重量
        int[] v = {5, 2, 3}; // 物品的价值
        System.out.println(new BackPack().dp(w,v,N,W));
    }

}

```

为了加深对代码的理解，可参考下方图解

背包 容量　　w = {3, 2, 1} v = {5, 2, 3}

物品数量	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	5	5	5
2	0	0	2	5	5	7
3	0	3	3	5	8	8

扩展题目：[1049. 最后一块石头的重量 II]

2、416. 分割等和子集

前面学习了0-1背包问题，来看看背包问题的思想能够如何运用到其他算法题目。

[字节，腾讯，百度面试题，416. 分割等和子集]

对于这个问题，看起来和0-1背包没有任何关系，但是我们可以将其转换成0-1背包问题，怎么转换？

> 首先回忆一下0-1背包问题是怎么描述的？

> 给你一个可装载重量为`W`的背包和`N`个物品，每个物品有重量和价值两个属性。其中第`i`个物品的重量为`w[i]`，价值为`v[i]`，现在让你用这个背包装物品，能装的最大价值是多少？

转换成0-1背包的描述如下：

我们可以先对集合求和，得出`sum`，**给一个可装载重量为sum/2的背包和N个物品，每个物品的重量为nums[i]。现在让你装物品，是否存在一种装法，能够恰好将背包装满**？

2.1、解法分析

1、确定状态参数：

状态就是 背包的容量 W 和 可选择的物品N

2、决策/选择：

将物品`i`装入背包，不将物品`i`装入背包。

3、明确dp数组的含义：

`dp[i][j]=true`代表，对于前`i`个物品，当前背包的容量为`j`时恰好将背包装满，`dp[i][j]=false`则说明不能恰好将背包装满。

> 比如说，如果`dp[4][9] = true`，其含义为：对于容量为 9 的背包，若只是用前 4 个物品，可以有一种方法把背包恰好装满。
>
> 而对于本题来说它的具体含义是指：对于给定的集合中，若只对前 4 个数字进行选择，存在一个子集的和可以恰好凑出 9。

		背包容量w											
		0	1	2	3	4	5	6	7	8	9	10	11
物品个数i	0	T	F	F	F	F	F	F	F	F	F	F	F
	1	T	T	F	F	F	F	F	F	F	F	F	F
	2	T	T	F	F	F	T	T	F	F	F	F	F
	3	T	T	F	F	F	T	T	F	F	F	F	T
	4	T	T	F	F	F	T	T	F	F	F	F	T

根据这个定义，我们想求的最终答案就是`dp[N][sum/2]`。

4、明确初始化状态：

背包容量W=0，肯定存在一种装法，那就是不装；可选择物品`N`为0则肯定不存在一种装法

初始条件就是`dp[..][0] = true`和`dp[0][..] = false`，因为背包没有空间的时候，就相当于装满了，而当没有物品可选择的时候，肯定没办法装满背包。

5、状态转移的逻辑：

针对`dp`数组含义，可以根据我们的选择对`dp[i][j]`得到以下状态转移：

- 如果不把`nums[i]`算入子集，**或者说你不把这第i个物品装入背包**，那么是否能够恰好装满背包，取决于上一个状态`dp[i-1][j]`，继承之前的结果。
- 如果把`nums[i]`算入子集，**或者说你把这第i个物品装入了背包**，那么是否能够恰好装满背包，取决于状态`dp[i - 1][j-nums[i-1]]`。

> 1、由于`i`是从 1 开始的，而数组索引是从 0 开始的，所以第`i`个物品的重量应该是`nums[i-1]`

> 2、`dp[i - 1][j-nums[i-1]]`也很好理解：你如果装了第`i`个物品，就要看背包的剩余重量`j - nums[i-1]`限制下是否能够被恰好装满。
>
> 换句话说，如果`j - nums[i-1]`的重量可以被恰好装满，那么只要把第`i`个物品装进去，也可恰好装满`j`的重量；否则的话，重量`j`肯定是装不满的。

6、代码实现

```java
class Solution {
    //DP解法
    public boolean canPartition(int[] nums) {
        //将问题转换成0-1背包问题，
        /*
            对nums求和为sum,该问题转换为,给你N个物品和sum/2容量大小的背包,是否存在一种能将背包恰好装满的方法。
            1.确定状态：容量，物品个数
            2.选择/决策：将物品i装入背包,将物品i不装入背包.
            3.初始状态：容量为0,存在一种放法，为true。物品个数为0肯定不存在满足条件的放法为false。
            4.明确dp数组的含义，dp[i][j],前i个物品在背包容量为j下是否存在一种装法刚好装满
                初始条件:dp[0][...] =false; dp[...][0]=true;
            5.状态转移逻辑：
                不将物品i放入背包,返回值由dp[i-1][j]决定
                将物品i放入背包，返回值由dp[i-1][j-nums[i-1]]决定
        */
        //求和
        int sum = 0 ;
        int max = 0;
        for (int num:nums) {
            sum+=num;
            max = Math.max(max,num);
        }
        //剪枝1,如果所有数的和为奇数,则肯定不存在有i个物品重量和为 sum/2;
        if (sum % 2 !=0) {
            return false;
        }
        //剪枝2 如果集合中的最大数大于sum/2则肯定不存在这种组合
        if (max > sum/2) {
            return false;
        }
        int N = nums.length;
        int M = sum/2;

        //定义dp数组
        boolean[][] dp = new boolean[N+1][M+1];
        //初始化
        for (int i=0;i<=M;i++) {
            dp[0][i] = false;
        }
        for (int i=0;i<=N;i++) {
            dp[i][0] = true;
        }

        //状态转移
        for (int i=1;i<=N;i++) {
```

```java
                for (int j=1;j<=M;j++) {
                    if (j < nums[i-1]) {
                        //物品i背包放不下那只能选择不放进去,因此是否有一种满足条件的放法取决 (i-1)物品在容量j限制下是否满足
                        dp[i][j] = dp[i-1][j];
                    }else {
                        //背包能放下物品i,但是也可以选择放和不放,最终是否满足题意取决放或者不放的结果
                        //当物品i放入背包时,最终的结果取决与(i-1)物品在( j - num[i-1] )限制下的结果。
                        dp[i][j] =  dp[i-1][j] || dp[i-1][j - nums[i-1]];
                    }
                }
            }
            return dp[N][M];
        }
}
```

> 时间复杂度：O(N * M)
>
> 空间复杂度：O(N * M)

2.2、状态压缩

对于刚刚的代码是否还能优化?

通过分析代码，我们发现一个问题，`dp[i][j]` 都是通过上一行 `dp[i-1][..]` 转移过来的，之前的数据都不会再使用了。

所以，我们可以进行**状态压缩**，将二维 `dp` 数组压缩为一维，**降低空间复杂度**

```java
class Solution {
    //DP解法
    public boolean canPartition(int[] nums) {
        /*
          每一行的dp值都只与上一行的dp值有关，因此只需要一个一维数组
        */
        //求和
        int sum = 0 ;
        int max = 0;
        for (int num:nums) {
            sum+=num;
            max = Math.max(max,num);
        }
        //剪枝1,如果所有数的和为奇数,则肯定不存在有i个物品重量和为 sum/2;
        if (sum % 2 !=0) {
            return false;
        }
        //剪枝2 如果集合中的最大数大于sum/2则肯定不存在这种组合
        if (max > sum/2) {
            return false;
        }
        int N = nums.length;
        int M = sum/2;

        //定义dp数组
        boolean[] dp = new boolean[M+1];
        //初始化
```

```java
        dp[0] = true;

        //状态转移
        for (int i=1;i<=N;i++) {
            //注意这里需要从大到小
            for (int j=M;j>=0;j--) {
                if ( j>=nums[i-1]) {
                    dp[j] =  dp[j] || dp[j - nums[i-1]];
                }
            }
        }
        return dp[M];
    }
}
```

> 压缩到一维dp之后，内层循环我们需要从大到小计算，为什么？
>
> 1、`dp[i][j]`的取值都是从上一行`dp[i-1][...]`转移过来的，并且是从上一行的该列或该列的前面几列转移过来的
>
> 2、压缩后dp中存储的是上一行中每一列的取值，每个`i`的取值在内层循环结束后dp中存储的就是该行每列的取值了，然后进入到下一行。
>
> 3、如果列从小到大更新，内层循环每计算一次会将dp中的值修改为当前行该列的值，当我们在计算后面列的时候，dp中存储的就不再是上一行对应列的取值了。
>
> 4、但是如果是从大到小更新就不存在这个问题了。

> 优化后代码的时间复杂度：O（N * M），空间复杂度：O(M)
>
> 另外该题还有非动态规划解法，利用剪枝+回溯！

注意：压缩后的状态只跟背包的容量有关了！

进阶： 0-1 背包问题：

[474. 一和零]

[494. 目标和]

[879. 盈利计划]

3、518. 零钱兑换 II

[华为，字节，腾讯面试题，518. 零钱兑换 II]

把这个问题转化为背包问题的描述形式：

有一个背包，最大容量为`amount`，有一系列物品`coins`，每个物品的重量为`coins[i]`，**每个物品的数量无限。**请问有多少种方法，能够把背包恰好装满？

> 这个问题和我们前面讲过的两个背包问题，有一个最大的区别就是，每个物品的数量是无限的，这也就是传说中的**完全背包问题**

算法解析：

1、明确状态参数和决策

状态有两个，就是`背包的容量`和`可选择的物品`，选择就是`装进背包`或者`不装进背包`。

2、定义dp数组

`dp[i][j]`的定义如下：**使用前i个物品，当背包容量为j时，有`dp[i][j]`种方法可以装满背包。不同于以前的是物品可以使用多次**

翻译回我们题目的意思就是：**使用coins中的前i个硬币的面值，若想凑出金额j，有`dp[i][j]`种凑法。**

3、初始化状态

初始化状态为：`dp[0][..] = 0`，`dp[..][0] = 1`。因为如果不使用任何硬币面值，就无法凑出任何金额；如果凑出的目标金额为 0，那么“不凑”就是唯一的一种凑法。

		总金额j					
		0	1	2	3	4	5
	0	1	0	0	0	0	0
硬币个数i	1	1	1	1	1	1	1
	2	1	1	2	2	3	3
	3	1	1	2	2	3	4

最终想得到的答案就是`dp[N][amount]`，其中`N`为`coins`数组的大小。

4、状态转移逻辑

本题的不同点在于物品的数量是无限的，因此状态转移逻辑如下：

- **如果你不把这第i个物品装入背包**，也就是说你不使用`coins[i]`这个面值的硬币，那么凑出面额`j`的方法数`dp[i][j]`应该等于`dp[i-1][j]`，继承之前的结果。
- **如果你把这第i个物品装入了背包**，也就是说你使用`coins[i]`这个面值的硬币，那么`dp[i][j]`应该等于`dp[i][j-coins[i-1]]`。

> 由于`i`是从 1 开始的，所以`coins`的索引是`i-1`时表示第`i`个硬币的面值。

`dp[i][j-coins[i-1]]`代表如果用这个面值的硬币，接着只需关注如何凑出金额`j - coins[i-1]`

综上两种选择，而我们想求的`dp[i][j]`又代表了共有多少种凑法，所以`dp[i][j]`的值应该是以上两种选择的结果之和

5、代码实现

```
class Solution {
    public int change(int amount, int[] coins) {
```

```java

        //定义dp
        int n = coins.length;
        int[][] dp = new int[n+1][amount+1];
        //初始化条件 dp[0][....] = 0;dp[...][0] = 1
        for (int i=0;i<=amount;i++) {
            dp[0][i] = 0;
        }
        for (int i=0;i<=n;i++) {
            dp[i][0] = 1;
        }
        //状态转移
        for (int i=1;i<=n;i++) {
            for (int j=1;j<=amount;j++) {
                if (j < coins[i-1]) {
                    dp[i][j] = dp[i-1][j];
                }else {
                    dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]];
                }
            }
        }
        return dp[n][amount];
    }
}
```

> 时间复杂度：O(N * amount)，空间复杂度：O （N * amount)

6、进一步优化，状态压缩

```java
class Solution {
    public int change(int amount, int[] coins) {

        //定义dp
        int n = coins.length;
        //状态压缩
        int[] dp = new int[amount+1];
        //初始化条件 dp[0] = 1
        dp[0] =1;
        //状态转移
        for (int i=1;i<=n;i++) {
            for (int j=1;j<=amount;j++) {
                if( j>= coins[i-1]) {
                    dp[j] = dp[j] + dp[j-coins[i-1]];
                }
            }
        }
        return dp[amount];
    }
}
//更为直观明了的写法如下
class Solution {
    public int change(int amount, int[] coins) {
        //定义dp,状态压缩
        int[] dp = new int[amount+1];
        //初始化条件 dp[0] = 1
        dp[0] =1;
```

```
        //状态转移
        for (int coin:coins) {
            for (int j=1;j<=amount;j++) {
                if( j>= coin) {
                    dp[j] = dp[j] + dp[j-coin];
                }
            }
        }
        return dp[amount];
    }
}
```

> 注意此处：
>
> 1、一维dp中保存的是上一行的dp值，
>
> 2、内存循环从小到大原因是：
>
> 当前行的状态值要么直接等于上一行该列的状态值，
>
> 要么等于上一行当前列的值`dp[j]`+当前行前面列的值`dp[j-coins[i-1]]`，因此要从小到大来计算。
>
> 如果从大到小计算上面的`dp[j-coins[i-1]]`还是上一行前面列的值，并不是当前行的，因为当前行的还没被修改。

注意：压缩后的状态只跟amount有关了！

打家劫舍系列

198. 打家劫舍

[思科，字节，中国电信，阿里最近面试题，198. 打家劫舍]

算法分析：

1、确定状态参数

要注意的是：由题意可得，原问题和子问题之间并不是所给的房间个数变少了。而是我可以选择从某个位置开始偷，一直偷到最后面。

> 原问题：从第一间房子开始偷，一直偷到最后
>
> ```
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
> ```
>
> 子问题：从第三间房子开始偷，一直偷到最后
>
> ```
> 输入：[1,2,3,1]
> 输出：3
> 解释：偷窃 3 号房屋 (金额 = 3)。
> ```

由此可见，给定的房间数组`int[] nums`的下标`i`就是我们的状态参数。

2、确定选择

选择也很简单，强盗从左到右走过这一排房子，在每间房子前都有两种选择：**抢**或者**不抢。**

3、定义dp数组的含义

`dp[i]=x`的含义是：从下标为`i`的房子开始抢，一直到最后所能抢到的最高金额为`x`。

> 配图：
>
> ```
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
> ```
>
> i从n-1往前遍历
>
> | i | 0 | 1 | 2 | 3 | 4 | 5 |
> |---|---|---|---|---|---|---|
> | dp[i] | 4 | 3 | 3 | 1 | 0 | 0 |
>
> `dp[1]=3`代表的是强盗从第二间房屋开始抢到最后所能抢到的最大金额是3.
>
> 而本题我们最终需要返回的是：`dp[0]`,从第一间房屋就开始做选择。

4、确定初始状态

强盗从左到右走过这一排房子，在每间房子前可以选择抢或者不抢，当**它走过了最后一间房子后**，就没得抢了，**能抢到的钱显然是 0**

因此初始条件为：`dp[nums.length]=0`

5、状态转移逻辑

按照我们的选择，强盗从左到右走过这一排房子，在每间房子前都有两种选择：**抢**或者**不抢。**

- 如果你抢了`i`这间房子，那么你肯定不能抢相邻的下一间房子了，只能从下下间房子`i+2`开始做选择。此时`dp[i]=dp[i+2]+nums[i]`，其中`nums[i]`就是我在房间`i`抢到的money。
- 如果你不抢`i`这间房子，那么你可以走到下一间房子`i+1`前，继续做选择，此时`dp[i]=dp[i+1]`

在两个选择中，每次都选更大的结果，最后得到的就是最多能抢到的 money

```
dp[i] = Math.max(dp[i+1],dp[i+2] + nums[i]);
```

6、代码实现

```java
class Solution {
    //dp解法
    public int rob(int[] nums) {
        int n = nums.length;
        //定义dp数组
        int[] dp = new int[n+2];
        //初始化状态
        dp[n] = dp[n+1] = 0;

        /*
            状态转移,先从子问题开始求解
            i=n-1:就是最后一间房,它依赖(i+1)和(i+2), 故dp数组的长度为:n+2
            初始条件则为:dp[n] = dp[n+1] = 0;
        */
        for (int i=n-1;i>=0;i--) {
            dp[i] = Math.max(dp[i+1],nums[i]+ dp[i+2]);
        }
        return dp[0];
    }
}
```

7、思考状态是否能压缩

我们发现状态转移只和 `dp[i]` 最近的两个状态 `dp[i+1]`， `dp[i+2]` 有关，所以可以进一步优化，将空间复杂度降低到 O(1)。

```java
class Solution {
    //dp解法
    public int rob(int[] nums) {
        int n = nums.length;
        //发现状态转移只和dp[i]最近的两个状态dp[i+1], dp[i+2]有关,进一步优化状态空间
        int dp_i = 0;
        int dp_i_1 = 0;
        int dp_i_2 = 0;
        for (int i=n-1;i>=0;i--) {
            dp_i = Math.max(dp_i_1,nums[i]+ dp_i_2);
            //以前的dp_i_1转移后成为了dp_i_2,dp_i转移后成为了dp_i_1
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i;
    }
}
```

进阶：

[213. 打家劫舍 II]

[337. 打家劫舍 III]