参与本项目，**贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！**

动态规划：01背包理论基础

《代码随想录》算法视频公开课：带你学透0-1背包问题！**，相信结合视频再看本篇题解，更有助于大家对本题的理解。**

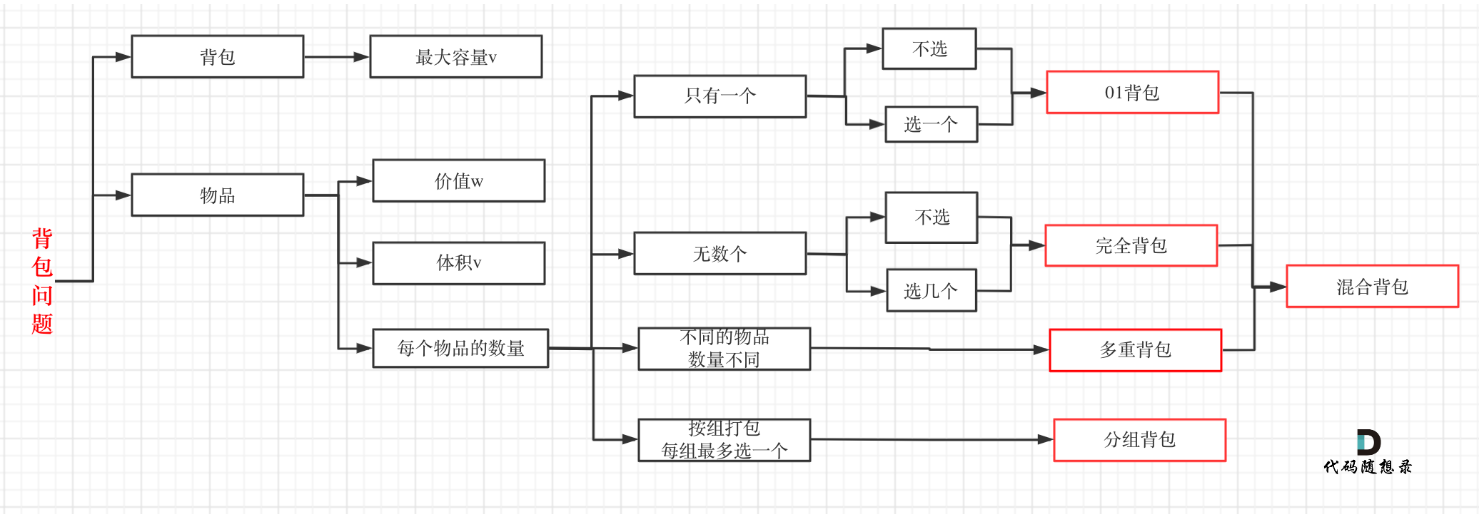
这周我们正式开始讲解背包问题！

背包问题的经典资料当然是：背包九讲。在公众号「代码随想录」后台回复：背包九讲，就可以获得背包九讲的pdf。

但说实话，背包九讲对于小白来说确实不太友好，看起来还是有点费劲的，而且都是伪代码理解起来也吃力。

对于面试的话，其实掌握01背包，和完全背包，就够用了，最多可以再来一个多重背包。

如果这几种背包，分不清，我这里画了一个图，如下：



至于背包九讲其其他背包，面试几乎不会问，都是竞赛级别的了，leetcode上连多重背包的题目都没有，所以题库也告诉我们，01背包和完全背包就够用了。

而完全背包又是也是01背包稍作变化而来，即：完全背包的物品数量是无限的。

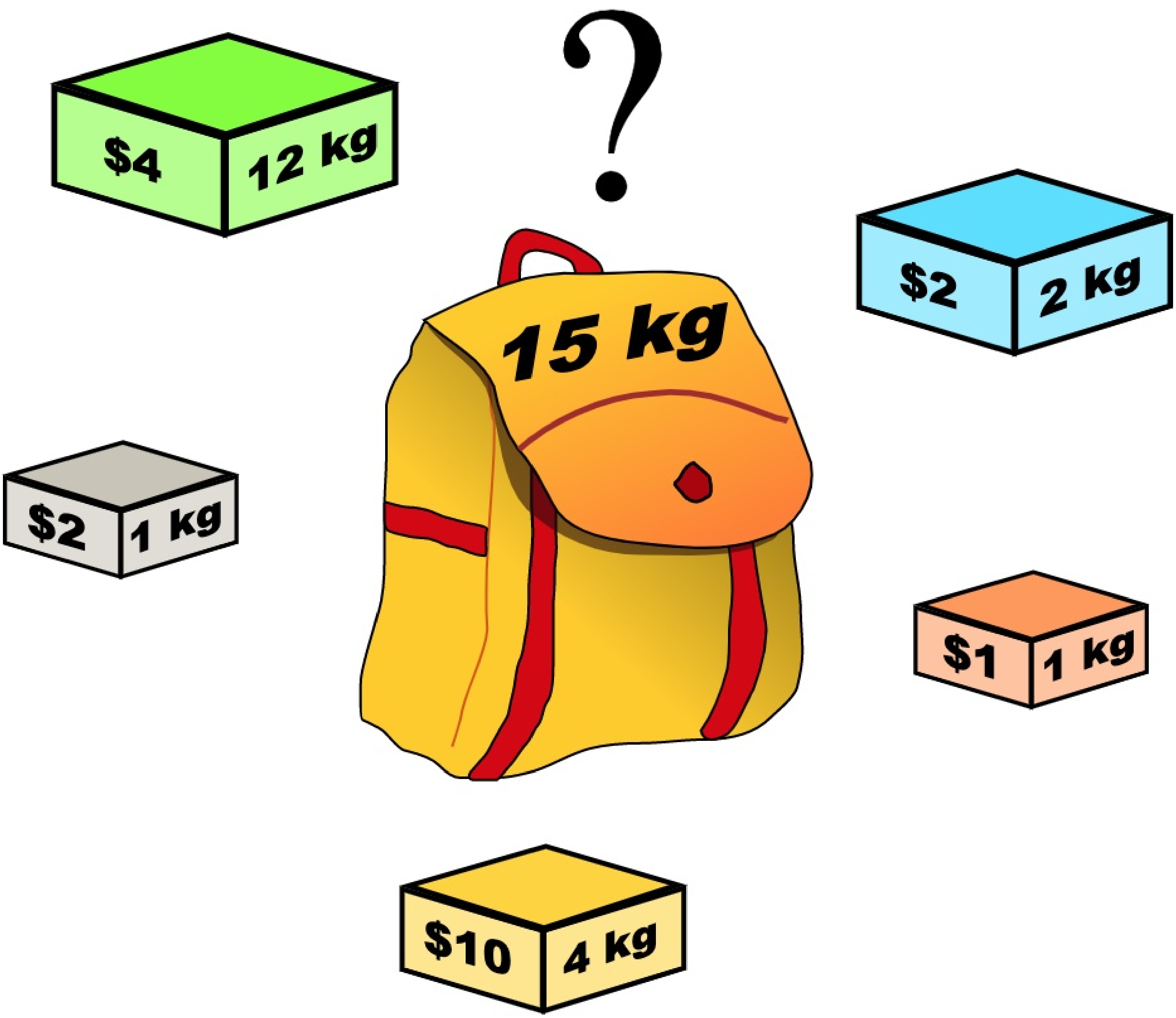
所以背包问题的理论基础重中之重是01背包，一定要理解透！

所以我先通过纯01背包问题，把01背包原理讲清楚，后续再讲解leetcode题目的时候，重点就是讲解如何转化为01背包问题了。

之前可能有些录友已经可以熟练写出背包了，但只要把这个文章仔细看完，相信你会意外收获！

01 背包

有n件物品和一个最多能背重量为w 的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品只能用一次**，求解将哪些物品装入背包里物品价值总和最大。



这是标准的背包问题，以至于很多同学看了这个自然就会想到背包，甚至都不知道暴力的解法应该怎么解了。

这样其实是没有从底向上去思考，而是习惯性想到了背包，那么暴力的解法应该是怎么样的呢？

所以暴力的解法是指数级别的时间复杂度。进而才需要动态规划的解法来进行优化！

在下面的讲解中，我举一个例子：

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

问背包能背的物品最大价值是多少？

以下讲解和图示中出现的数字都是以这个例子为例。

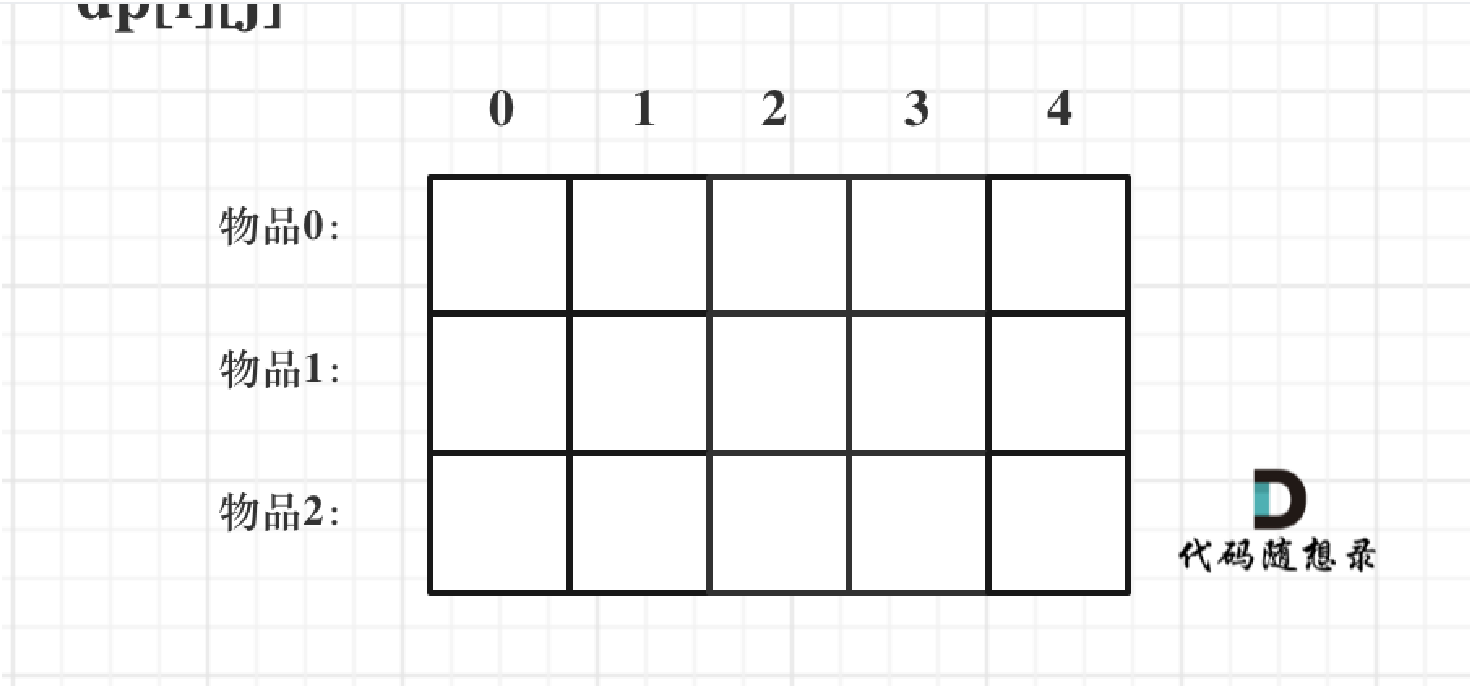
二维dp数组01背包

依然动规五部曲分析一波。

1. 确定dp数组以及下标的含义

对于背包问题，有一种写法， 是使用二维数组，即dp[i][j] **表示从下标为[0-i]的物品里任意取，放进容量为j的背包，价值总和最大是多少。**

只看这个二维数组的定义，大家一定会有点懵，看下面这个图：



要时刻记着这个dp数组的含义，下面的一些步骤都围绕这dp数组的含义进行的，如果哪里看懵了，就来回顾一下i代表什么，j又代表什么。

2. 确定递推公式

再回顾一下dp[i][j]的含义：从下标为[0-i]的物品里任意取，放进容量为j的背包，价值总和最大是多少。

那么可以有两个方向推出来dp[i][j]，

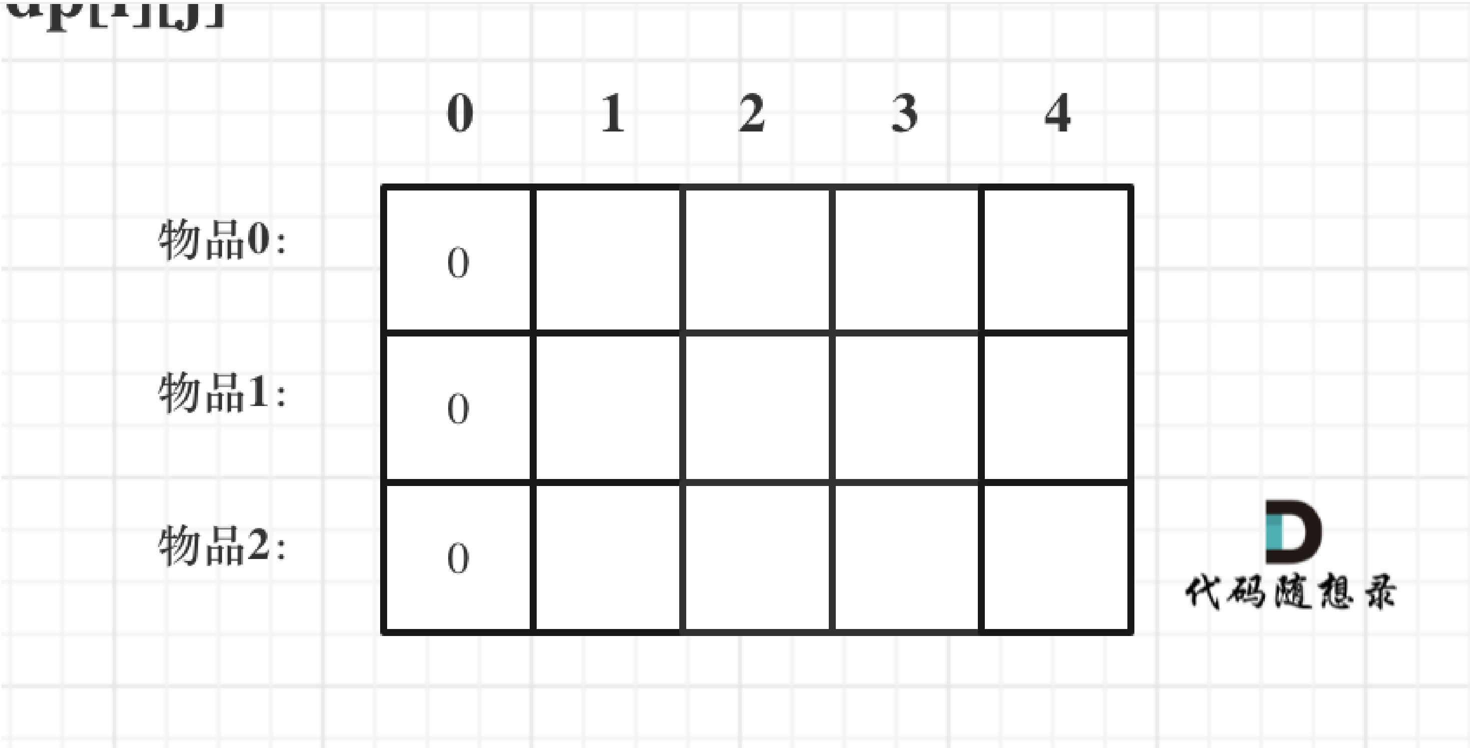
- **不放物品i**：由dp[i - 1][j]推出，即背包容量为j，里面不放物品i的最大价值，此时dp[i][j]就是dp[i - 1][j]。(其实就是当物品i的重量大于背包j的重量时，物品i无法放进背包中，所以被背包内的价值依然和前面相同。)
- **放物品i**：由dp[i - 1][j - weight[i]]推出，dp[i - 1][j - weight[i]] 为背包容量为j - weight[i]的时候不放物品i的最大价值，那么dp[i - 1][j - weight[i]] + value[i] （物品i的价值），就是背包放物品i得到的最大价值

所以递归公式： dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);

3. dp数组如何初始化

关于初始化，一定要和dp数组的定义吻合，否则到递推公式的时候就会越来越乱。

首先从dp[i][j]的定义出发，如果背包容量j为0的话，即dp[i][0]，无论是选取哪些物品，背包价值总和一定为0。如图：



在看其他情况。

状态转移方程 dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]); 可以看出i 是由 i-1 推导出来，那么i为0的时候就一定要初始化。

dp[0][j]，即：i为0，存放编号0的物品的时候，各个容量的背包所能存放的最大价值。

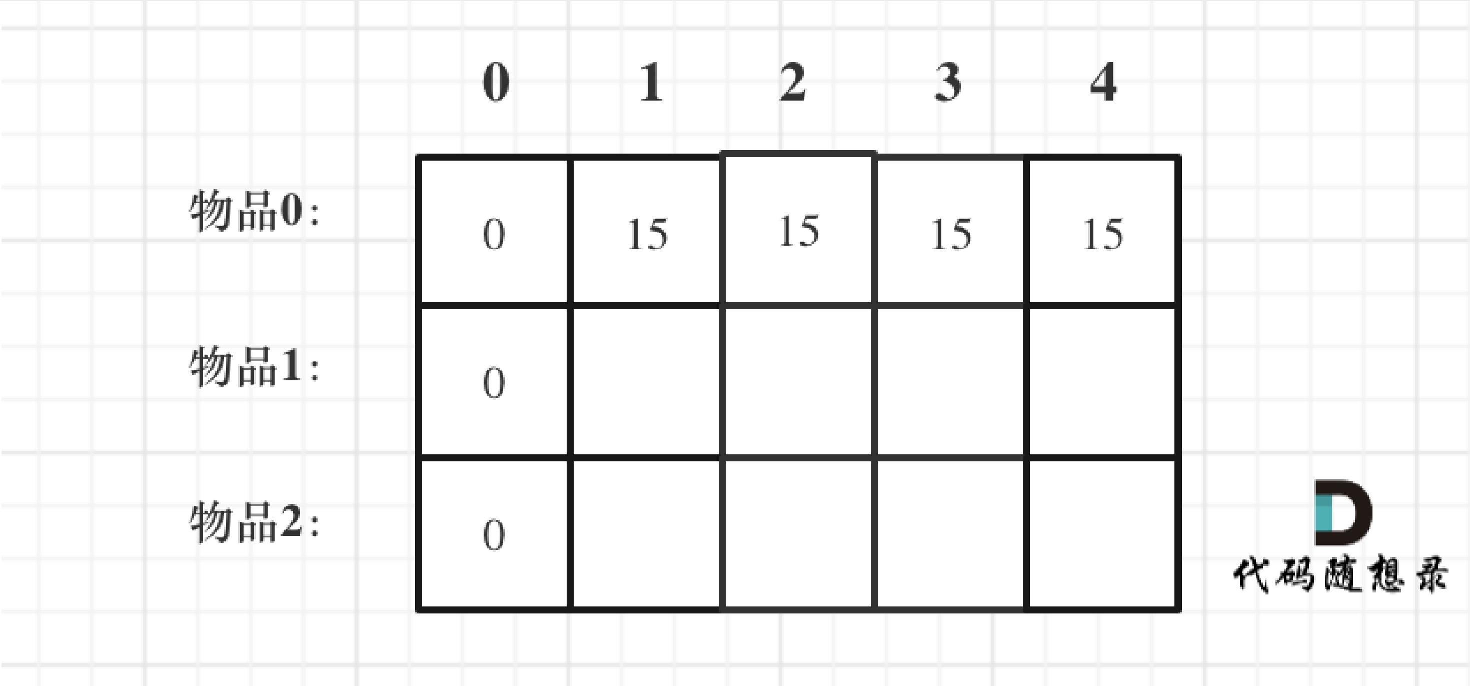
那么很明显当 j < weight[0]的时候，dp[0][j] 应该是 0，因为背包容量比编号0的物品重量还小。

当j >= weight[0]时，dp[0][j] 应该是value[0]，因为背包容量放足够放编号0物品。

代码初始化如下：

```
for (int j = 0 ; j < weight[0]; j++) {  // 当然这一步，如果把dp数组预先初始化为0了，这一步就可
    dp[0][j] = 0;
}
// 正序遍历
for (int j = weight[0]; j <= bagweight; j++) {
    dp[0][j] = value[0];
}
```

此时dp数组初始化情况如图所示：



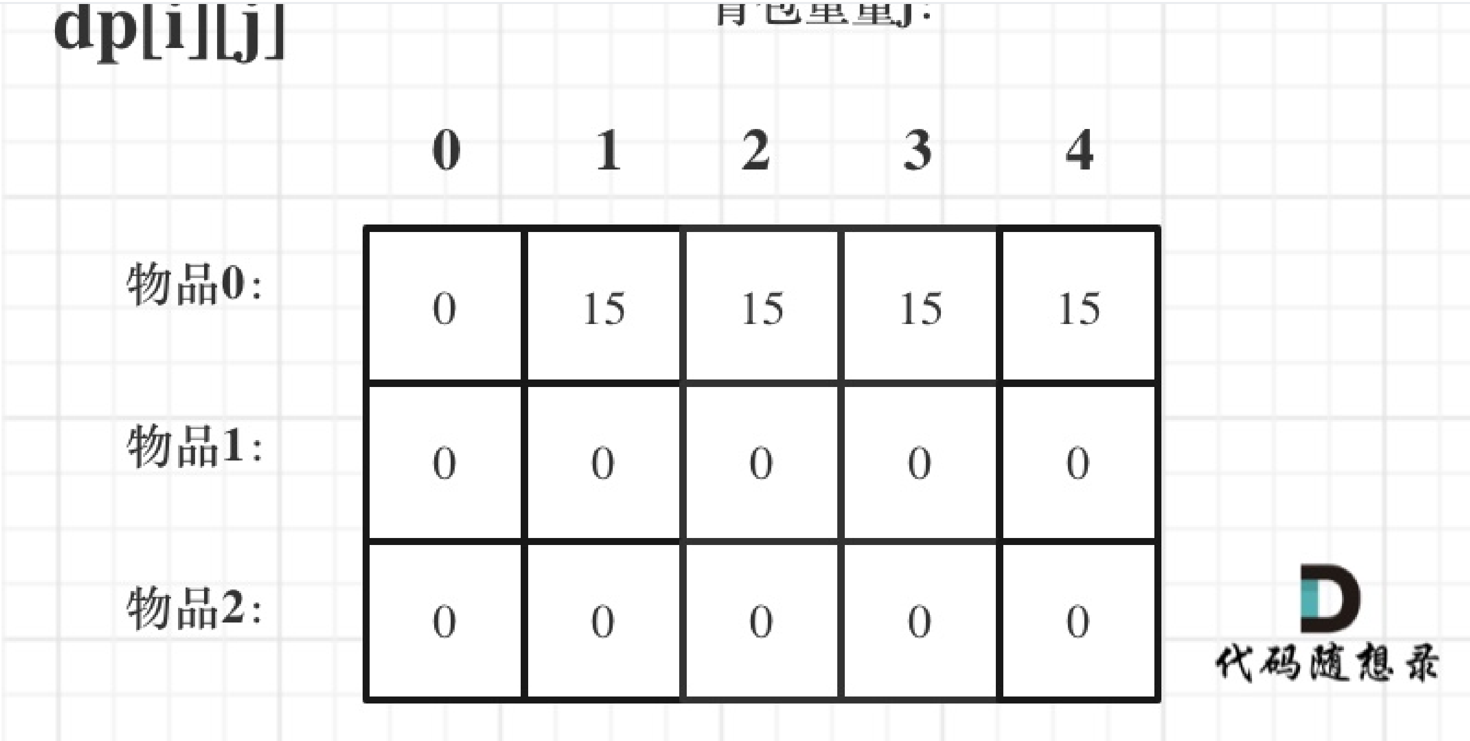
dp[0][j] 和 dp[i][0] 都已经初始化了，那么其他下标应该初始化多少呢？

其实从递归公式： dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]); 可以看出dp[i][j] 是由左上方数值推导出来了，那么 其他下标初始为什么数值都可以，因为都会被覆盖。

初始-1，初始-2，初始100，都可以！

但只不过一开始就统一把dp数组统一初始为0，更方便一些。

如图：


dp[i][j]

	0	1	2	3	4
物品0:	0	15	15	15	15
物品1:	0	0	0	0	0
物品2:	0	0	0	0	0

最后初始化代码如下：

```
// 初始化 dp
vector<vector<int>> dp(weight.size(), vector<int>(bagweight + 1, 0));
for (int j = weight[0]; j <= bagweight; j++) {
    dp[0][j] = value[0];
}
```

费了这么大的功夫，才把如何初始化讲清楚，相信不少同学平时初始化dp数组是凭感觉来的，但有时候感觉是不靠谱的。

4. 确定遍历顺序

在如下图中，可以看出，有两个遍历的维度：物品与背包重量

dp[i][j]

	0	1	2	3	4
物品0:	0	15	15	15	15
物品1:	0	0	0	0	0
物品2:	0	0	0	0	0

那么问题来了，**先遍历 物品还是先遍历背包重量呢?**

其实都可以！！ 但是先遍历物品更好理解。

那么我先给出先遍历物品，然后遍历背包重量的代码。

```
// weight数组的大小 就是物品个数
for(int i = 1; i < weight.size(); i++) { // 遍历物品
    for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);

    }
}
```

先遍历背包，再遍历物品，也是可以的！（注意我这里使用的二维dp数组）

例如这样:

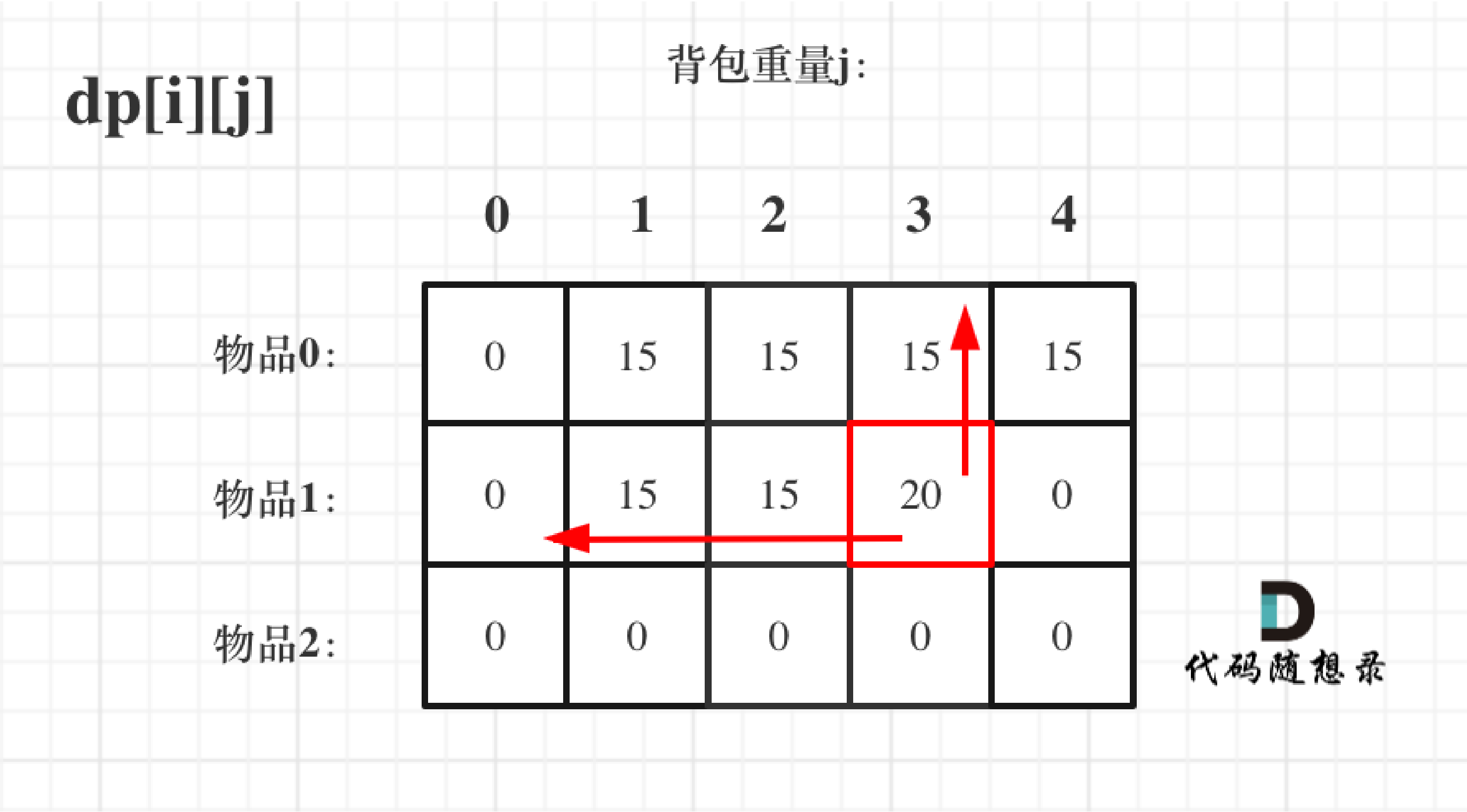
```


// weight数组的大小 就是物品个数
for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
    for(int i = 1; i < weight.size(); i++) { // 遍历物品
        if (j < weight[i]) dp[i][j] = dp[i - 1][j];
        else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
```

为什么也是可以的呢？

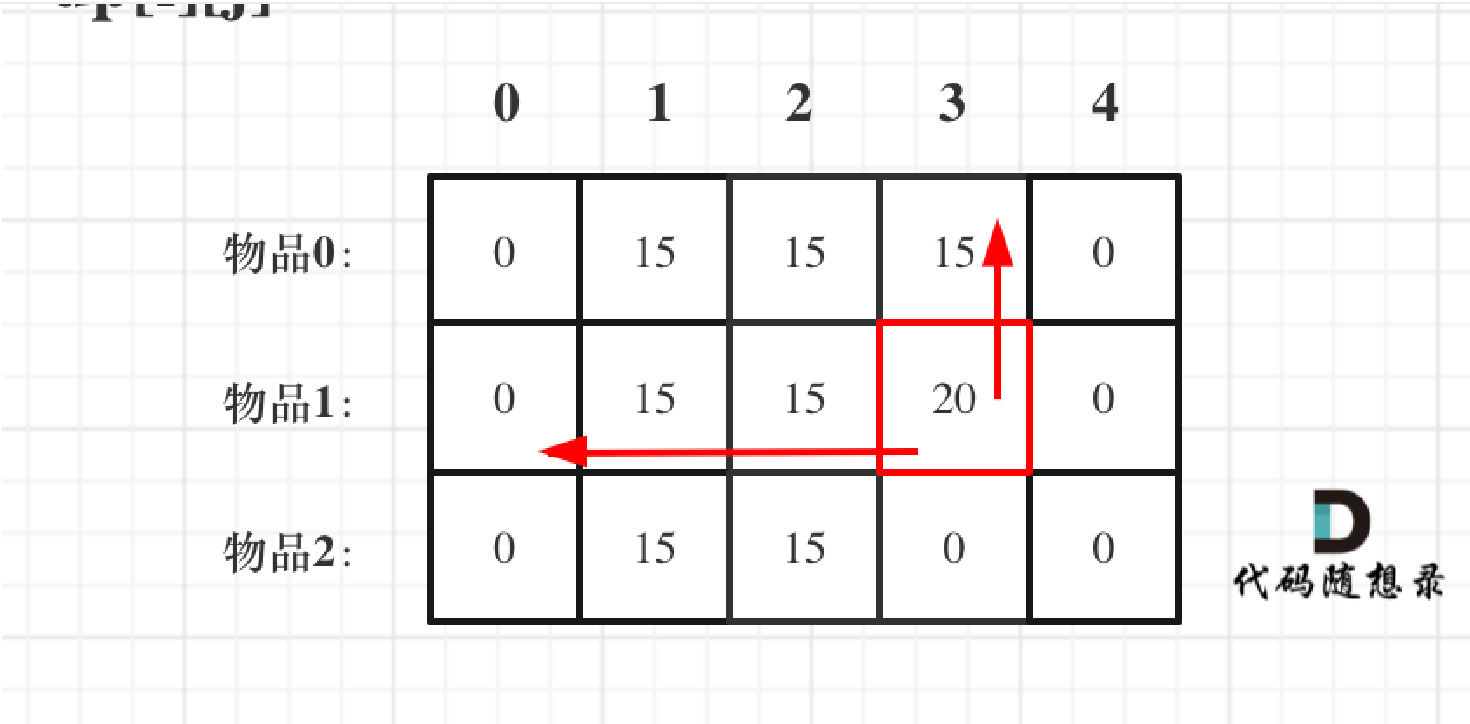
要理解递归的本质和递推的方向。

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]); 递归公式中可以看出dp[i][j]是靠dp[i-1][j]和dp[i - 1][j - weight[i]]推导出来的。

dp[i-1][j]和dp[i - 1][j - weight[i]] 都在dp[i][j]的左上角方向（包括正上方向），那么先遍历物品，再遍历背包的过程如图所示：



再来看看先遍历背包，再遍历物品呢，如图：



大家可以看出，虽然两个for循环遍历的次序不同，但是dp[i][j]所需要的数据就是左上角，根本不影响dp[i][j]公式的推导！

但先遍历物品再遍历背包这个顺序更好理解。

其实背包问题里，两个for循环的先后循序是非常有讲究的，理解遍历顺序其实比理解推导公式难多了。

5. 举例推导dp数组

来看一下对应的dp数组的数值，如图：

	0	1	2	3	4
物品0：	0	15	15	15	15
物品1：	0	15	15	20	35
物品2：	0	15	15	20	35

最终结果就是dp[2][4]。

建议大家此时自己在纸上推导一遍，看看dp数组里每一个数值是不是这样的。

做动态规划的题目，最好的过程就是自己在纸上举一个例子把对应的dp数组的数值推导一下，然后在动手写代码！

很多同学做dp题目，遇到各种问题，然后凭感觉东改改西改改，怎么改都不对，或者稀里糊涂就改过了。

主要就是自己没有动手推导一下dp数组的演变过程，如果推导明白了，代码写出来就算有问题，只要把dp数组打印出来，对比一下和自己推导的有什么差异，很快就可以发现问题了。

完整c++测试代码

```

void test_2_wei_bag_problem1() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;

    // 二维数组
    vector<vector<int>> dp(weight.size(), vector<int>(bagweight + 1, 0));

    // 初始化
    for (int j = weight[0]; j <= bagweight; j++) {
        dp[0][j] = value[0];
    }

```

```
    for(int i = 1; i < weight.size(); i++) { // 遍历物品
        for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
            if (j < weight[i]) dp[i][j] = dp[i - 1][j];
            else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);

        }
    }

    cout << dp[weight.size() - 1][bagweight] << endl;
}

int main() {
    test_2_wei_bag_problem1();
}
```

总结

讲了这么多才刚刚把二维dp的01背包讲完，**这里大家其实可以发现最简单的是推导公式了，推导公式估计看一遍就记下来了，但难就难在如何初始化和遍历顺序上。**

可能有的同学并没有注意到初始化 和 遍历顺序的重要性，我们后面做力扣上背包面试题目的时候，大家就会感受出来了。

下一篇 还是理论基础，我们再来讲一维dp数组实现的01背包（滚动数组），分析一下和二维有什么区别，在初始化和遍历顺序上又有什么差异，敬请期待！

其他语言版本

java

```java
public class BagProblem {
    public static void main(String[] args) {
        int[] weight = {1,3,4};
        int[] value = {15,20,30};
        int bagSize = 4;
        testWeightBagProblem(weight,value,bagSize);
    }

```

```java
 * @param weight  物品的重量
 * @param value   物品的价值
 * @param bagSize 背包的容量
 */
public static void testWeightBagProblem(int[] weight, int[] value, int bagSize){

    // 创建dp数组
    int goods = weight.length;  // 获取物品的数量
    int[][] dp = new int[goods][bagSize + 1];

    // 初始化dp数组
    // 创建数组后，其中默认的值就是0
    for (int j = weight[0]; j <= bagSize; j++) {
        dp[0][j] = value[0];
    }

    // 填充dp数组
    for (int i = 1; i < weight.length; i++) {
        for (int j = 1; j <= bagSize; j++) {
            if (j < weight[i]) {
                /**
                 * 当前背包的容量都没有当前物品i大的时候，是不放物品i的
                 * 那么前i-1个物品能放下的最大价值就是当前情况的最大价值
                 */
                dp[i][j] = dp[i-1][j];
            } else {
                /**
                 * 当前背包的容量可以放下物品i
                 * 那么此时分两种情况：
                 *    1、不放物品i
                 *    2、放物品i
                 * 比较这两种情况下，哪种背包中物品的最大价值最大
                 */
                dp[i][j] = Math.max(dp[i-1][j] , dp[i-1][j-weight[i]] + value[i]);
            }
        }
    }

    // 打印dp数组
    for (int i = 0; i < goods; i++) {
        for (int j = 0; j <= bagSize; j++) {
            System.out.print(dp[i][j] + "\t");
        }
        System.out.println("\n");
    }
```

n

```py
def test_2_wei_bag_problem1(bag_size, weight, value) -> int:
    rows, cols = len(weight), bag_size + 1
    dp = [[0 for _ in range(cols)] for _ in range(rows)]

    # 初始化dp数组.
    for i in range(rows):
        dp[i][0] = 0
    first_item_weight, first_item_value = weight[0], value[0]
    for j in range(1, cols):
        if first_item_weight <= j:
            dp[0][j] = first_item_value

    # 更新dp数组: 先遍历物品, 再遍历背包.
    for i in range(1, len(weight)):
        cur_weight, cur_val = weight[i], value[i]
        for j in range(1, cols):
            if cur_weight > j: # 说明背包装不下当前物品.
                dp[i][j] = dp[i - 1][j] # 所以不装当前物品.
            else:
                # 定义dp数组: dp[i][j] 前i个物品里，放进容量为j的背包，价值总和最大是多少。
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cur_weight]+ cur_val)

    print(dp)


if __name__ == "__main__":
    bag_size = 4
    weight = [1, 3, 4]
    value = [15, 20, 30]
    test_2_wei_bag_problem1(bag_size, weight, value)
```

go

```go
func test_2_wei_bag_problem1(weight, value []int, bagweight int) int {
    // 定义dp数组
```

```go
        dp[i] = make([]int, bagweight+1)
    }
    // 初始化
    for j := bagweight; j >= weight[0]; j-- {
        dp[0][j] = dp[0][j-weight[0]] + value[0]
    }
    // 递推公式
    for i := 1; i < len(weight); i++ {
        //正序,也可以倒序
        for j := 0; j <= bagweight; j++ {
            if j < weight[i] {
                dp[i][j] = dp[i-1][j]
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]]+value[i])
            }
        }
    }
    return dp[len(weight)-1][bagweight]
}

func max(a,b int) int {
    if a > b {
        return a
    }
    return b
}

func main() {
    weight := []int{1,3,4}
    value := []int{15,20,30}
    test_2_wei_bag_problem1(weight,value,4)
}
```

javascript

```js
function testWeightBagProblem (weight, value, size) {
    // 定义 dp 数组
    const len = weight.length,
          dp = Array(len).fill().map(() => Array(size + 1).fill(0));

    // 初始化
```

```javascript
    }

    // weight 数组的长度len 就是物品个数
    for(let i = 1; i < len; i++) { // 遍历物品
        for(let j = 0; j <= size; j++) { // 遍历背包容量
            if(j < weight[i]) dp[i][j] = dp[i - 1][j];
            else dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
        }
    }

    console.table(dp)

    return dp[len - 1][size];
}

function test () {
    console.log(testWeightBagProblem([1, 3, 4, 5], [15, 20, 30, 55], 6));
}

test();
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX(a, b) (((a) > (b)) ? (a) : (b))
#define ARR_SIZE(a) (sizeof((a)) / sizeof((a)[0]))
#define BAG_WEIGHT 4

void backPack(int* weights, int weightSize, int* costs, int costSize, int bagWeight) {
    // 开辟dp数组
    int dp[weightSize][bagWeight + 1];
    memset(dp, 0, sizeof(int) * weightSize * (bagWeight + 1));

    int i, j;
    // 当背包容量大于物品0的重量时，将物品0放入到背包中
    for(j = weights[0]; j <= bagWeight; ++j) {
        dp[0][j] = costs[0];
    }
```

```c
    for(j = 1; j <= bagWeight; ++j) {
        for(i = 1; i < weightSize; ++i) {
            // 如果当前背包容量小于物品重量
            if(j < weights[i])
                // 背包物品的价值等于背包不放置当前物品时的价值
                dp[i][j] = dp[i-1][j];
            // 若背包当前重量可以放置物品
            else
                // 背包的价值等于放置该物品或不放置该物品的最大值
                dp[i][j] = MAX(dp[i - 1][j],  dp[i - 1][j - weights[i]] + costs[i]);
        }
    }

    printf("%d\n", dp[weightSize - 1][bagWeight]);
}

int main(int argc, char* argv[]) {
    int weights[] = {1, 3, 4};
    int costs[] = {15, 20, 30};
    backPack(weights, ARR_SIZE(weights), costs, ARR_SIZE(costs), BAG_WEIGHT);
    return 0;
}
```

TypeScript

```ts
function testWeightBagProblem(
  weight: number[],
  value: number[],
  size: number
): number {
  /**
   * dp[i][j]: 前i个物品，背包容量为j，能获得的最大价值
   * dp[0][*]: u=weight[0],u之前为0，u之后（含u）为value[0]
   * dp[*][0]: 0
   * ...
   * dp[i][j]: max(dp[i-1][j], dp[i-1][j-weight[i]]+value[i]);
   */
  const goodsNum: number = weight.length;
  const dp: number[][] = new Array(goodsNum)
    .fill(0)
    .map((_) => new Array(size + 1).fill(0));
```

```
  for (let i = 1; i < goodsNum; i++) {
    for (let j = 1; j <= size; j++) {
      if (j < weight[i]) {
        dp[i][j] = dp[i - 1][j];
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);
      }
    }
  }
  return dp[goodsNum - 1][size];
}
// test
const weight = [1, 3, 4];
const value = [15, 20, 30];
const size = 4;
console.log(testWeightBagProblem(weight, value, size));
```

```

object Solution {
  // 01背包
  def test_2_wei_bag_problem1(): Unit = {
    var weight = Array[Int](1, 3, 4)
    var value = Array[Int](15, 20, 30)
    var baseweight = 4

    // 二维数组
    var dp = Array.ofDim[Int](weight.length, baseweight + 1)

    // 初始化
    for (j <- weight(0) to baseweight) {
      dp(0)(j) = value(0)
    }

    // 遍历
    for (i <- 1 until weight.length; j <- 1 to baseweight) {
      if (j - weight(i) >= 0) dp(i)(j) = dp(i - 1)(j - weight(i)) + value(i)
      dp(i)(j) = math.max(dp(i)(j), dp(i - 1)(j))
    }

```

```
    dp(weight.length - 1)(baseweight) // 最终返回
  }

  def main(args: Array[String]): Unit = {
    test_2_wei_bag_problem1()
  }
}
```

上次更新：: 1/6/2023, 12:38:01 PM