经典动态规划：完全背包问题

原创 labuladong labuladong 2020年04月12日 07:30

点击上方蓝字设为星标✨

东哥带你手把手撕力扣~

作者：labuladong

公众号：labuladong

若已授权白名单也必须保留以上来源信息

零钱兑换 2 是另一种典型背包问题的变体，我们前文已经讲了 经典动态规划：0-1 背包问题 和 背包问题变体：相等子集分割。

希望你已经看过前两篇文章，看过了动态规划和背包问题的套路，这篇继续按照背包问题的套路，列举一个背包问题的变形。

本文聊的是 LeetCode 第 518 题 Coin Change 2，题目如下：

518. 零钱兑换 II

难度 中等 122 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

```
int change(int amount, int[] coins);
```

PS：至于 Coin Change 1，在我们前文 动态规划套路详解 写过。

我们可以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**」，没啥高大上的，无非就是状态转移方程有一点变化而已。

下面就以背包问题的描述形式，继续按照流程来分析。

解题思路

第一步要明确两点，「状态」和「选择」。

这部分都是背包问题的老套路了，我还是啰嗦一下吧：

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」。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
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][...] = 计算(选择1，选择2...)
```

第二步要明确 `dp` 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 `dp` 数组。

`dp[i][j]` 的定义如下：

若只使用前 `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。

换句话说，翻译回我们题目的意思就是：

若只使用 `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。

经过以上的定义，可以得到：

base case 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“无为而治”就是唯一的一种凑法。

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

大致的伪码思路如下：

```
int dp[N+1][amount+1]
dp[0][..] = 0
dp[..][0] = 1
```

```
for i in [1..N]:
    for j in [1..amount]:
        把物品 i 装进背包,
        不把物品 i 装进背包
return dp[N][amount]
```

第三步，根据「选择」，思考状态转移的逻辑。

注意，我们这个问题的特殊点在于物品的数量是无限的，所以这里和之前写的背包问题文章有所不同。

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]]`。

首先由于 `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]]` 也不难理解，如果你决定使用这个面值的硬币，那么就应该关注如何凑出金额 `j - coins[i-1]`。

比如说，你想用面值为 2 的硬币凑出金额 5，那么如果你知道了凑出金额 3 的方法，再加上一枚面额为 2 的硬币，不就可以凑出 5 了嘛。

综上就是两种选择，而我们想求的 `dp[i][j]` 是「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：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= amount; j++) {
        if (j - coins[i-1] >= 0)
            dp[i][j] = dp[i - 1][j]
                     + dp[i][j-coins[i-1]];
return dp[N][W]
```

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了一些边界问题：

```
int change(int amount, int[] coins) {
    int n = coins.length;
    int[][] dp = amount int[n + 1][amount + 1];
    // base case
    for (int i = 0; i <= n; i++)
        dp[i][0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= amount; j++)
            if (j - coins[i-1] >= 0)
                dp[i][j] = dp[i - 1][j]
                         + dp[i][j - coins[i-1]];
            else
                dp[i][j] = dp[i - 1][j];
    }
```

```
    return dp[n][amount];
}
```

而且，我们通过观察可以发现，dp 数组的转移只和 dp[i][..] 和 dp[i-1][..] 有关，所以可以压缩状态，进一步降低算法的空间复杂度：

```
int change(int amount, int[] coins) {
    int n = coins.length;
    int[] dp = new int[amount + 1];
    dp[0] = 1; // base case
    for (int i = 0; i < n; i++)
        for (int j = 1; j <= amount; j++)
            if (j - coins[i] >= 0)
                dp[j] = dp[j] + dp[j-coins[i]];

    return dp[amount];
}
```

这个解法和之前的思路完全相同，将二维 dp 数组压缩为一维，时间复杂度 O(N*amount)，空间复杂度 O(amount)。

至此，这道零钱兑换问题也通过背包问题的框架解决了。

往期推荐 🔗

数据结构和算法学习指南

为了学会二分搜索，我写了首诗

回溯算法解题框架

动态规划解题框架

经典动态规划：高楼扔鸡蛋

经动态规划：编辑距离

动态规划之博弈问题

动态规划之KMP算法详解

公众号：labuladong

B站：labuladong

知乎：labuladong

作者在 GitHub 上的 fucking-algorithm 仓库上个月获得 23k star，又开始了本月的 GitHub Trending 霸榜…… 扫码关注公众号，东哥带你手撕 LeetCode，感受支配算法的乐趣~

后台回复『pdf』限时免费下载《labuladong的算法小抄》，回复『加群』可加入 LeetCode 刷题群，大家一起刷题、内推：

手把手刷动态规划 31 二维动态规划 16 背包问题 3

手把手刷动态规划 · 目录

上一篇
经典动态规划：0-1背包问题的变体

下一篇
经典动态规划：戳气球问题