

## 43 | 拓扑排序：如何确定代码源文件的编译依赖关系？

2019-01-04 王争



朗读人：修阳

时长09:37 大小8.82M



从今天开始，我们就进入了专栏的高级篇。相对基础篇，高级篇涉及的知识点，都比较零散，不是太系统。所以，我会围绕一个实际软件开发的问题，在阐述具体解决方法的过程中，将涉及的知识点给你详细讲解出来。

所以，相较于基础篇“**开篇问题 - 知识讲解 - 回答开篇 - 总结 - 课后思考**”这样的文章结构，高级篇我稍作了些改变，大致分为这样几个部分：“**问题阐述 - 算法解析 - 总结引申 - 课后思考**”。

好了，现在，我们就进入高级篇的第一节，如何确定代码源文件的编译依赖关系？

我们知道，一个完整的项目往往会包含很多代码源文件。编译器在编译整个项目的时候，需要按照依赖关系，依次编译每个源文件。比如，A.cpp 依赖 B.cpp，那在编译的时候，编译器需要先编译 B.cpp，才能编译 A.cpp。

编译器通过分析源文件或者程序员事先写好的编译配置文件（比如 Makefile 文件），来获取这种局部的依赖关系。**那编译器又该如何通过源文件两两之间的局部依赖关系，确定一个全局的编译顺序呢？**

A handwritten diagram on a light yellow background. On the left, three lines of text are grouped by a large right-facing curly brace: 'A.cpp 依赖 B.cpp', 'B.cpp 依赖 C.cpp', and 'D.cpp 依赖 B.cpp'. To the right of the brace is a right-pointing arrow. To the right of the arrow, under the heading '源文件的编译顺序' (Compilation order of source files), are two possible sequences: 'C.cpp → B.cpp → A.cpp → D.cpp' and '或者 C.cpp → B.cpp → D.cpp → A.cpp'.

## 算法解析

这个问题的解决思路与“图”这种数据结构的一个经典算法“拓扑排序算法”有关。那什么是拓扑排序呢？这个概念很好理解，我们先来看一个生活中的拓扑排序的例子。

我们在穿衣服的时候都有一定的顺序，我们可以把这种顺序想成，衣服与衣服之间有一定的依赖关系。比如说，你必须先穿袜子才能穿鞋，先穿内裤才能穿秋裤。假设我们现在有八件衣服要穿，它们之间的两两依赖关系我们已经很清楚了，那如何安排一个穿衣序列，能够满足所有的两两之间的依赖关系？

这就是个拓扑排序问题。从这个例子中，你应该能想到，在很多时候，拓扑排序的序列并不是唯一的。你可以看我画的这幅图，我找到了好几种满足这些局部先后关系的穿衣序列。

两两之间的局部依赖关系：

内裤 → 裤子；内裤 → 鞋子；裤子 → 鞋子，裤子 → 腰带；  
袜子 → 鞋子；衬衣 → 外套；衬衣 → 领带

全局有序序列：

内裤 → 裤子 → 腰带 → 袜子 → 鞋子 → 衬衣 → 领带 → 外套  
衬衣 → 外套 → 领带 → 内裤 → 袜子 → 裤子 → 鞋子 → 腰带


弄懂了这个生活中的例子，开篇的关于编译顺序的问题，你应该也有思路了。开篇问题跟这个问题的模型是一样的，也可以抽象成一个拓扑排序问题。

拓扑排序的原理非常简单，我们的重点应该放到拓扑排序的实现上面。

我前面多次讲过，算法是构建在具体的数据结构之上的。针对这个问题，我们先来看下，如何将问题背景抽象成具体的数据结构？

我们可以把源文件与源文件之间的依赖关系，抽象成一个有向图。每个源文件对应图中的一个顶点，源文件之间的依赖关系就是顶点之间的边。

如果 a 先于 b 执行，也就是说 b 依赖于 a，那么就在顶点 a 和顶点 b 之间，构建一条从 a 指向 b 的边。而且，这个图不仅要是 有向图，还要是一个 有向无环图，也就是不能存在像 a → b → c → a 这样的循环依赖关系。因为图中一旦出现环，拓扑排序就无法工作了。实际上，拓扑排序本身就是基于有向无环图的一个算法。

 复制代码

```
1 public class Graph {  
2     private int v; // 顶点的个数  
3     private LinkedList<Integer> adj[]; // 邻接表  
4  
5     public Graph(int v) {  
6         this.v = v;  
7         adj = new LinkedList[v];
```

```
8     for (int i=0; i<v; ++i) {
9         adj[i] = new LinkedList<>();
10    }
11 }
12
13 public void addEdge(int s, int t) { // s 先于 t, 边 s->t
14     adj[s].add(t);
15 }
16 }
```

数据结构定义好了, 现在, 我们来看, **如何在这个有向无环图上, 实现拓扑排序?**

拓扑排序有两种实现方法, 都不难理解。它们分别是**Kahn 算法**和**DFS 深度优先搜索算法**。我们依次来看下它们都是怎么工作的。


## 1.Kahn 算法

Kahn 算法实际上用的是贪心算法思想, 思路非常简单、好懂。

定义数据结构的时候, 如果  $s$  需要先于  $t$  执行, 那就添加一条  $s$  指向  $t$  的边。所以, 如果某个顶点入度为 0, 也就表示, 没有任何顶点必须先于这个顶点执行, 那么这个顶点就可以执行了。

我们先从图中, 找出一个入度为 0 的顶点, 将其输出到拓扑排序的结果序列中(对应代码中就是把它打印出来), 并且把这个顶点从图中删除(也就是把这个顶点可达的顶点的入度都减 1)。我们循环执行上面的过程, 直到所有的顶点都被输出。最后输出的序列, 就是满足局部依赖关系的拓扑排序。

我把 Kahn 算法用代码实现了一下, 你可以结合着文字描述一块看下。不过, 你应该能发现, 这段代码实现更有技巧一些, 并没有真正删除顶点的操作。代码中有详细的注释, 你自己来看, 我就不多解释了。

 复制代码

```
1 public void topoSortByKahn() {
2     int[] inDegree = new int[v]; // 统计每个顶点的入度
3     for (int i = 0; i < v; ++i) {
4         for (int j = 0; j < adj[i].size(); ++j) {
5             int w = adj[i].get(j); // i->w
6             inDegree[w]++;
7         }
8     }
```

```


8     }
9     LinkedList<Integer> queue = new LinkedList<>();
10    for (int i = 0; i < v; ++i) {
11        if (inDegree[i] == 0) queue.add(i);
12    }
13    while (!queue.isEmpty()) {
14        int i = queue.remove();
15        System.out.print("->" + i);
16        for (int j = 0; j < adj[i].size(); ++j) {
17            int k = adj[i].get(j);
18            inDegree[k]--;
19            if (inDegree[k] == 0) queue.add(k);
20        }
21    }
22 }

```

## 2.DFS 算法

图上的深度优先搜索我们前面已经讲过了，实际上，拓扑排序也可以用深度优先搜索来实现。不过这里的名字要稍微改下，更加确切的说法应该是深度优先遍历，遍历图中的所有顶点，而非只是搜索一个顶点到另一个顶点的路径。

关于这个算法的实现原理，我先把代码贴在下面，下面给你具体解释。

 复制代码

```

1 public void topoSortByDFS() {
2     // 先构建逆邻接表，边 s->t 表示，s 依赖于 t，t 先于 s
3     LinkedList<Integer> inverseAdj[] = new LinkedList[v];
4     for (int i = 0; i < v; ++i) { // 申请空间
5         inverseAdj[i] = new LinkedList<>();
6     }
7     for (int i = 0; i < v; ++i) { // 通过邻接表生成逆邻接表
8         for (int j = 0; j < adj[i].size(); ++j) {
9             int w = adj[i].get(j); // i->w
10            inverseAdj[w].add(i); // w->i
11        }
12    }
13    boolean[] visited = new boolean[v];
14    for (int i = 0; i < v; ++i) { // 深度优先遍历图
15        if (visited[i] == false) {
16            visited[i] = true;
17            dfs(i, inverseAdj, visited);
18        }
19    }
20 }
21

```

```
22 private void dfs(  
23     int vertex, LinkedList<Integer> inverseAdj[], boolean[] visited) {  
24     for (int i = 0; i < inverseAdj[vertex].size(); ++i) {  
25         int w = inverseAdj[vertex].get(i);  
26         if (visited[w] == true) continue;  
27         visited[w] = true;  
28         dfs(w, inverseAdj, visited);  
29     } // 先把 vertex 这个顶点可达的所有顶点都打印出来之后, 再打印它自己  
30     System.out.print("->" + vertex);  
31 }
```

这个算法包含两个关键部分。

第一部分是**通过邻接表构造逆邻接表**。邻接表中, 边  $s \rightarrow t$  表示  $s$  先于  $t$  执行, 也就是  $t$  要依赖  $s$ 。在逆邻接表中, 边  $s \rightarrow t$  表示  $s$  依赖于  $t$ ,  $s$  后于  $t$  执行。为什么这么转化呢? 这个跟我们这个算法的实现思想有关。

第二部分是这个算法的核心, 也就是**递归处理每个顶点**。对于顶点  $vertex$  来说, 我们先输出它可达的所有顶点, 也就是说, 先把它依赖的所有的顶点输出了, 然后再输出自己。

到这里, 用 Kahn 算法和 DFS 算法求拓扑排序的原理和代码实现都讲完了。我们来看下, **这两个算法的时间复杂度分别是多少呢?**

从 Kahn 代码中可以看出, 每个顶点被访问了一次, 每个边也都被访问了一次, 所以, Kahn 算法的时间复杂度就是  $O(V+E)$  ( $V$  表示顶点个数,  $E$  表示边的个数)。

DFS 算法的时间复杂度我们之前分析过。每个顶点被访问两次, 每条边都被访问一次, 所以时间复杂度也是  $O(V+E)$ 。

注意, 这里的图可能不是连通的, 有可能是有好几个不连通的子图构成, 所以,  $E$  并不一定大于  $V$ , 两者的大小关系不确定。所以, 在表示时间复杂度的时候,  $V$ 、 $E$  都要考虑在内。

## 总结引申


在基础篇中, 关于“图”, 我们讲了图的定义和存储、图的广度和深度优先搜索。今天, 我们又讲了一个关于图的算法, 拓扑排序。



拓扑排序应用非常广泛，解决的问题的模型也非常一致。凡是需要通过局部顺序来推导全局顺序的，一般都能用拓扑排序来解决。除此之外，拓扑排序还能检测图中环的存在。对于 Kahn 算法来说，如果最后输出出来的顶点个数，少于图中顶点个数，图中还有入度不是 0 的顶点，那就说明，图中存在环。

关于图中环的检测，我们在[递归](#)那一节讲过一个例子，在查找最终推荐人的时候，可能会因为脏数据，造成存在循环推荐，比如，用户 A 推荐了用户 B，用户 B 推荐了用户 C，用户 C 又推荐了用户 A。如何避免这种脏数据导致的无限递归？这个问题，我当时留给你思考了，现在是时候解答了。

实际上，这就是环的检测问题。因为我们每次都只是查找一个用户的最终推荐人，所以，我们并不需要动用复杂的拓扑排序算法，而只需要记录已经访问过的用户 ID，当用户 ID 第二次被访问的时候，就说明存在环，也就说明存在脏数据。

 复制代码

```
1 HashSet<Integer> hashTable = new HashSet<>(); // 保存已经访问过的 actorId
2 long findRootReferrerId(long actorId) {
3     if (hashTable.contains(actorId)) { // 存在环
4         return;
5     }
6     hashTable.add(actorId);
7     Long referrerId =
8         select referrer_id from [table] where actor_id = actorId;
9     if (referrerId == null) return actorId;
10    return findRootReferrerId(referrerId);
11 }
```

如果把这个问题改一下，我们想要知道，数据库中的所有用户之间的推荐关系了，有没有存在环的情况。这个问题，就需要用到拓扑排序算法了。我们把用户之间的推荐关系，从数据库中加载到内存中，然后构建成今天讲的这种有向图数据结构，再利用拓扑排序，就可以快速检测出是否存在环了。

## 课后思考

1. 在今天的讲解中，我们用图表示依赖关系的时候，如果 a 先于 b 执行，我们就画一条从 a 到 b 的有向边；反过来，如果 a 先于 b，我们画一条从 b 到 a 的有向边，表示 b 依赖 a，那今天讲的 Kahn 算法和 DFS 算法还能否正确工作呢？如果不能，应该如何改造一下呢？

2. 我们今天讲了两种拓扑排序算法的实现思路，Kahn 算法和 DFS 深度优先搜索算法，如果换做 BFS 广度优先搜索算法，还可以实现吗？

欢迎留言和我分享，也欢迎点击“[请朋友读](#)”，把今天的内容分享给你的好友，和他一起讨论、学习。



# 数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争  
前 Google 工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有[现金](#)奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 42 | 动态规划实战：如何实现搜索引擎中的拼写纠错功能？

下一篇 44 | 最短路径：地图软件是如何计算出最优出行路径的？

## 精选留言

 写留言



Jerry银银

2019-01-04

 25

老师，这门专栏快结束了，突然有点新的想法：如果老师在讲解算法的时候，多讲点算法的由来，也就是背景，那就更好了。

...

展开



编辑回复: 这个有意思, 我们想想。



**Jerry银银**

2019-01-04

👍 5

思考题:

1. a先于b执行, 也就说b依赖于a, b指向a, 这样构建有向无环图时, 要找到出度为0的顶点, 然后删除...

展开 ▾



**Aaron**

2019-01-05

👍 4

课后思考里 “BFS 深度优先搜索算法” 是否应该是 “BFS 广度优先搜索算法”? BFS: Breadth-first Search

展开 ▾



**Handongyang**

2019-01-07

👍 3

@Jerry银银

针对你提的算法的由来与背景的问题, 我想我们完全可以通过维基百科查看, 一般都有其背景以及算法应用的场景, 甚至有些算法在维基百科上有相应的文献引用, 这些都可以参考。

作者回复: 银银同学要的显然不是这些

这就好比我在跟大家讲古诗 登黄鹤楼。银银同学想知道的是 怎么才能站在黄鹤楼上 作出登黄鹤楼这么牛逼的诗 诗人的脑回路是咋样的

而并不是想要历史性介绍 这首诗是谁谁谁 在某某年 某某地 历史背景下 做出来的

不知道我理解的对不

关于前者 我在讲解的时候已经尽量还原来龙去脉 但是可能学的并不明显 而且这本身就是很难说清楚的 说不定诗人自己都不知道自己咋写出这么牛逼的诗的

**你有资格吗?**

2019-01-07

👍 1

老师，好像数据结构少了B+树的讲解啊，B+不准备讲吗？

**Edward**

2019-01-05

👍 1

老师你好。我在做一道动态规划题的时候，不借助其他启发性线索时，在纸上演算一遍后，发现自己如果不能直觉地从演算中推演出解答的关键，就会产生强烈的自我怀疑。会有一层对自己智力水平的怀疑，如果没有一定的智商，是不适合做这件事情的。请问老师...

展开 ▾

作者回复: 多练习 多思考 多总结 慢慢就好了 都有这么一个过程的

**纯洁的憎恶**

2019-01-04

👍 1

1.kahn算法找出度为0的节点删除。dfs算法直接用正邻接表即可。

2. BFS也可以。其实与DFS一样，BFS也是从某个节点开始，找到所有与其相连通的节...

展开 ▾

**NeverMore**

2019-01-04

👍 1

1、反过来的话计算的就不是入度了，可以用出度来判断；  
2、BFS的话，则需要记录上一个节点是哪个，可以实现，但是比DFS要麻烦些。  
还请老师指点。...

展开 ▾

作者回复: 专栏结束的时候吧 也算是一个回顾 现在年底忙 没啥时间写呢

**想当架构师**

👍 0

2019-01-26

## 我怎么觉得这个kahn算法其实就是BFS算法

**不系之舟**

2019-01-19

👍 0

老师您好，还看到过另一个深度优先遍历的方法，是通过将节点涂不同的颜色判断是否在遍历的时候遇到了环，这种方法看着应该很明了，但是好像很少看到有人这么写程序，不知道是什么原因呢？

**猫头鹰爱拿铁**

2019-01-18

👍 0

思考题：

1、思路是找出度为0的节点然后打印出来。kahn算法可以和上面的类似通过构建逆邻接表找出入度为0的节点，其余都一样。dfs和讲解中代码一致只是不需要再构建逆邻接表...

展开 ▾

**Alexis何春光**

2019-01-13

👍 0

kahn算法中统计每个顶点的入度，有两层循环，时间复杂度为什么不是 $O(V \cdot E)$ 呢？

作者回复：第一层是 $v$  但第二层不是 $E$ 呢

**Alexis何春光**

2019-01-13

👍 0

这个问题有没有可能通过hashmap来做？用每一个事件之前的一个事件作为key，事件本身作为value，然后遍历一遍

展开 ▾

**DreamYe**

2019-01-09

👍 0

拓扑排序没问题，但是c++编译器需要对cpp文件的编译有顺序要求吗？按照我的理解，每一个cpp文件在预处理之后，都是独立的编译单元，然后编译成.o 文件。然后linker把

所有的.o 文件链接成可执行文件。我并不认为有编译依赖关系。可能其他语言有? 譬如...  
展开 ▾

**spark**

2019-01-08

👍 0

DFS 算法, 里面的递归差点就被绕进去了, 这个递归终止条件太隐蔽了.....不仔细看代码, 还以为没有终止条件会死循环.....好巧妙, 打算我也想不出这样写代码

展开 ▾

**zixuan**

2019-01-07

👍 0

逆邻接表上拓扑排序用BFS实现不了.

**蓝天**

2019-01-07

👍 0

刚解决完工作中类似的问题 老师的文章就来了, 然后才知道那个算法叫kahn

**徐凯**

2019-01-06

👍 0

我感觉bfs并不能找到节点之间的依赖关系, 而且就算找到了入度为0的节点, 可队列前面可能是度非0的节点, 它们不出队后面的节点也无法出队啊

展开 ▾

**往事随风, 顺其自然**

2019-01-05

👍 0

```
public void topoSortByDFS() {  
    // 先构建逆邻接表, 边 s->t 表示, s 依赖于 t, t 先于 s  
    LinkedList<Integer> inverseAdj[] = new LinkedList[v];...
```

展开 ▾

作者回复: i->w表示存在一条顶点i到顶点w的有向边





往事随风，顺其自然

2019-01-05

👍 0

```
public void topoSortByKahn() {  
    for (int i = 0; i < v; ++i) {  
        for (int j = 0; j < adj[i].size(); ++j) {...
```

展开 ▾

作者回复: 遍历每个顶点的链表 每个顶点的链表是独立存储的

