C++ 算法：并查集

一、 引言

1.1 问题引入：朋友圈的合并与查询

想象一下，我们班里有 N 个同学，刚开学时大家互不相识，每个人都是一个独立的“小团体”。随着时间推移，不断有两人成为好朋友（例如“小明和小红成了好朋友”）。我们规定，**朋友的朋友也是朋友**，这样就会形成一个个越来越大的“朋友圈”。

现在，班主任需要随时进行两种操作：

1. **合并 (Union)**：当两个人成为新朋友时，快速把他们各自的“朋友圈”合并成一个更大的圈子。
2. **查询 (Find)**：随时提问：同学 A 和同学 B 现在属于同一个朋友圈吗？

1.2 常规思路的困境

我们尝试用已学的知识来解决这个问题：

- **思路一：用数组存圈子编号**
  - 定义 `group_id[i]` 表示 i 号同学所在的圈子编号。
  - **查询 (A, B)**：只需比较 `group_id[A] == group_id[B]`。速度极快，时间复杂度 **O(1)**。
  - **合并 (A, B)**：假设A在1号圈，B在2号圈。为了合并，我们必须**遍历所有学生**，把所有组号为 2 的人都改成 1。速度慢，时间复杂度 **O(N)**。
  - 如果进行 M 次合并，总耗时 $O(M \times N)$，数据量稍大就会超时。
- **思路二：用图的邻接表 + DFS/BFS**
  - 把同学看作顶点，朋友关系看作双向边。
  - **合并 (A, B)**：在图里加一条边 `adj[A].push_back(B)`。快，时间复杂度 **O(1)**。
  - **查询 (A, B)**：要判断A和B是否连通，必须从A开始做一次完整的 DFS 或 BFS 去找 B。慢，时间复杂度 **O(N+M)**。

总结：常规方法总是顾此失彼（查询快合并就慢，合并快查询就慢）。我们需要一种全新的数据结构，能让**合并和查询都非常快**。它就是我们今天要学习的——**并查集**。

二、 核心思想

并查集的核心思想，就像现实生活中的组织架构：
每个圈子（集合）都推选出一个唯一的**“老大”（代表元）**，这个圈子里的所有事情，都由老大出面代表。

我们只需要一个非常简单的一维数组：
`int parent[N];`

- `parent[i]` 存储的是 i 号同学的**直接上级**是谁。

2.1 初始化

一开始，所有人都不认识，每个人都是自己圈子的老大。

```
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i; // 初始时，每个人的上级就是他自己（自己是老大）
    }
}
```

2.2 查询操作：`find(x)`

目标：找出 x 所在圈子的**最终老大**是谁。
怎么找？顺藤摸瓜，逐级上报！

1. 问 x 他的直接上级 `parent[x]` 是谁。
2. 如果上级就是他自己（`parent[x] == x`），说明 x 就是最高级别的老大，寻找结束。
3. 如果上级不是自己，就把寻找老大的任务抛给他的上级，去问 `find(parent[x])`。
4. 这样一条链式地问上去，必定能找到最终老大。

如果只做到这一步，并查集还是慢的。试想一条长长的单线联系：1的上级是2，2的上级是3... 99的上级是100（老大）。1找老大要汇报99次！

2.3 优化：路径压缩

小兵 1 心想：既然我最终的老大是司令 100，那我干嘛每次都要通过班长、排长、连长逐级传话呢？我直接越级，**把司令设为我的直接上级**不行吗？

这就是并查集的优化——**路径压缩**。在 `find(x)` 函数**递归返回**的过程中，顺手把经过的所有节点，直接挂到最终老大的名下。

```
int find(int x) {
    // 递归边界：如果我自己就是老大，返回我自己
    if (parent[x] == x) {
        return x;
    }
    // 核心代码：
    // 1. find(parent[x]) 会递归找到最终老大。
    // 2. 将最终老大的编号直接赋值给 parent[x]。
    // 3. 返回这个最终老大。
    return parent[x] = find(parent[x]);
}
```

经过路径压缩后，原本长长的一条链，就会被压缩到扁平。下次再执行 `find` 时，只需一步，时间复杂度接近 **O(1)**！

2.4 合并操作：`unite(u, v)`

有了 `find` 工具，合并操作实现起来就非常轻松。
u 和 v 成为朋友，要求合并他们的朋友圈。怎么做？**让一个圈子的老大，去认另一个圈子的老大当新老大！**

```
void unite(int u, int v) {
    int rootU = find(u); // 找到 u 的老大
    int rootV = find(v); // 找到 v 的老大

    // 如果老大不一样，说明本来不在一个圈子，需要合并
    if (rootU != rootV) {
        parent[rootU] = rootV; // 让 u 的老大 认 v 的老大 当上级
    }
}
```

思考：只改了 `parent[rootU] = rootV` 这一行，整个圈子就合并了吗?
答案是肯定的！ 因为原来以 `rootU` 为老大的圈子里的小弟们，下次调用 `find` 找老大时，会先找到 `rootU`，然后发现 `rootU` 上面又有新老大了（`rootV`），于是大家都会顺藤摸瓜找到 `rootV`，并在路径压缩的帮助下，直接把自己挂到 `rootV` 名下。

三、 实战演练

案例一：亲戚 (P238页 例17-1)

需求：给出 n 个人，m 对亲戚关系（若 x,y 是亲戚，y,z 是亲戚，则 x,z 也是亲戚）。随后有 p 次询问，判断两个人是否是亲戚。

分析：

1. “亲戚关系传递” 就是把人合并到一个圈子。
2. “判断是否是亲戚” 就是判断两人是不是在同一个圈子（老大是不是同一个人）。

解题流程：

1. 读取 n，执行 `init(n)` 初始化并查集。
2. 读取 m 条关系，对每对关系执行 `unite(u, v)`。
3. 读取 p 次询问，对每对查询，判断 `find(u) == find(v)`。是则输出 Yes，否则输出 No。

案例二：村村通 (B0248)

需求：现有 n 个城镇，m 条已有道路。要求整个城镇网络全部连通（任意两城互达）。问：**最少还需要建设多少条道路?**

分析：
城镇是节点，道路是边。这 n 个节点初始时可能由几个互不连通的子图（岛屿）组成。
一个非常重要的图论结论：要把 K 个互不连通的岛屿连接成一个完整的连通图，最少需要 $K-1$ 条边。

现在的核心问题变成了：**如何计算当前图中，有多少个独立连通的“岛屿”（集合）？**

并查集的妙用：

1. **初始状态**：n 个城镇没有任何道路，就是 n 个独立的集合，每个城镇自己是老大。
2. **处理已有道路**：每输入一条道路 `(u, v)`，如果它们本来不连通（老大不同），我们就 `unite(u, v)` 将它们合并。
3. **统计孤岛数量**：所有道路处理完后，我们如何知道还剩多少个集合?
    - 在并查集中，每个独立的集合**有且仅有一个最终老大。**

- 老大的特征是什么？`parent[i] == i`！只有老大的上级是他自己。
- 因此，我们只需遍历 1 到 n，统计有多少个满足 `parent[i] == i` 的节点即可，假设数量为 K。

4. **得出答案**：最少需要的道路数 = $K - 1$。

核心代码片段：

```
int island_count = 0;
for (int i = 1; i <= n; i++) {
    // 如果我的上级还是我自己，说明我是某个 孤岛集合 的老大
    if (parent[i] == i) {
        island_count++;
    }
}
cout << island_count - 1 << endl; // 需要的边数 = 孤岛数 - 1
```

四、 总结

- **并查集 (DSU)** 是一种用于处理**不相交集合的合并及查询**问题的数据结构。
- **核心操作**：
  - `init`：初始化（`parent[i] = i`）。
  - `find`：找老大，并配合**路径压缩**（`parent[x] = find(parent[x])`），将长链压扁，保证效率。
  - `unite`：合并集合，老大认老大（`parent[rootU] = rootV`）。
- **应用场景**：朋友关系传递、连通性判断、图的连通块数量统计（如村村通问题）、以及未来将要学习的 Kruskal 最小生成树算法。
- **复杂度**：在应用了路径压缩后，并查集的查询和合并操作的时间复杂度几乎可以看作是 **O(1)**（准确说复杂度应该是反阿克曼函数）。