并查集

定义

在一个网络拓扑结构中，节点A与节点B连通，节点D与节点C连通，节点C与节点A连通，如何快速判断节点A和节点D是否连通？

起由

一个解决办法就是将整个网络拓扑结构抽象成一个无向图，然后使用相关算法来解决。但是实际上有着更高效的数据结构来判断节点间是否具有连通性，那就是并查集

接口

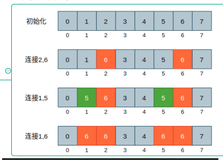
```
public class UnionFind {
    UnionFind(int N)                // 初始化节点
    void union(int p, int q)        // 连接p，q
    int find(int p)                 // 查找p的所在位
    boolean connected(int p, int q)  // 判断p，q是否连接
}
```

并查集这一数据结构由数组构建而成，使用数组下标来表示具体的节点，使用数组保存的值来表示连通性，听起来非常的抽象，所以还是直接看代码实现

Quick-Find

设计思路

Quick-Find的基本思想是，当连接两个节点时，将数组内相应的值更新为相等



在连接P、Q两个节点时，首先查找P、Q两个节点的标识位，将所有与P连接的节点的标识位改为Q的标识位

实现

```
public int find(int p) {
    // 查找p的标识符
    return data[p];
}

public void union(int p, int q) {
    int pID = find(p);
    int qID = find(q);

    if (pID == qID) return;

    // 遍历所有与节点p标识位相同的节点，并将其标识位改为q的标识位
    for (int i = 0; i < data.length; i++) {
        if (data[i] == pID)
            data[i] = qID;
    }
}
```

在判断两个节点是否连通时，只需要比较它们在数组中的标志位是否相同即可，若相同，则表示连通，若不同，则表示不连通

数组取下标元素内容的时间复杂度为O(1)，所以判断连通性的时间复杂度也为O(1)，所以该实现叫做Quick-Find

不过union的时间复杂度为O(n)，在每次连接节点时，都需要遍历整个数组进行查找

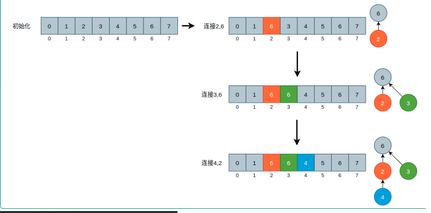
Quick-Union

当整个集群内拥有大量的节点，并且节点的连接操作执行频率很高时，Quick-Find由于其O(n)的连接复杂度，并不能满足系统的需求，因此有了另一种实现方式，Quick-Union

设计思路

Quick-Union算法的思路是将每一个元素看成是一个节点，将数组整理成为一个树结构，并由孩子节点指向父亲节点

在Quick-Find实现中，我们说数组的索引代表了节点本身，而数组的索引值代表了节点的标识位。而现在，数组的索引值不再代表节点的标识位了，而是代表其父节点

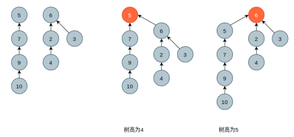


实现

```
public int find(int p) {
    // 循环找出父亲节点
    while (p != data[p])
        p = data[p];
    return p;
}

public void union(int p, int q) {
    // 将一个root节点的父亲是另一个节点的根节点
    int pRoot = find(p);
    int qRoot = find(q);

    if (pRoot == qRoot) return;

    data[pRoot] = qRoot;  // 使p的父节点指向q的父节点
}
```

那么此时判断两个节点是否连通就不能简单的判断数组元素内容了，而是需要在数组构建的"树"中找到节点对应的根节点，判断根节点是否为同一个

但是，使用此类方法实现时，find和union的平均时间复杂度均为O(n)，极端情况下会多次遍历数组



Quick-Union的优化

基于树高的优化

现象

如上图所示，当连通左右两个节点树时，将会有着不同的指针指向方式，两种方式最后的树高也不相同

所以，为了使得联合后的树高更小，应该让树高小的去指向树高更大的树的根节点

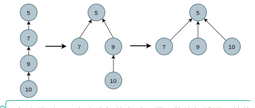
实现

在原有属性上，为每个数组内所构成的数添加树高

```
public class UnionFind{
    private int[] data;
    private int[] rank;

    UnionFind(int N) {
        data = new int[N];
        rank = new int[N]
        for(int i = 0; i < N; i++) {
            data[i] = i;
            rank[i] = 1;  // 初始化时每棵树的高度均为1
        }
    }
```

添加树高后，在union方法中，进行树高的判断，使rank[n]更小的节点指向rank[m]更大的节点

```
public void union(int p, int q) {
    int pRoot = find(p);
    int qRoot = find(q);

    if (pRoot == qRoot) return;

    if (rank[pRoot] < rank[qRoot])
        data[pRoot] = qRoot;
    else if (rank[pRoot] > rank[qRoot])
        data[qRoot] = pRoot;
    else {
        data[pRoot] = qRoot;
        rank[qRoot] ++;
    }
}
```



路径压缩

现象

如上图所示，在查询根节点时，遍历的路径越短，其效率也就越高，本质上和优化树高是一个意思，只不过优化的时机不同

对于路径压缩，通常会在find操作中进行优化，只需要让某一个节点直接指向节点父亲节点的父亲节点即可

实现

```
public int find(int p) {
    while (p != data[p]) {
        data[p] = data[data[p]];
        p = data[p];
    }
    return p;
}
```