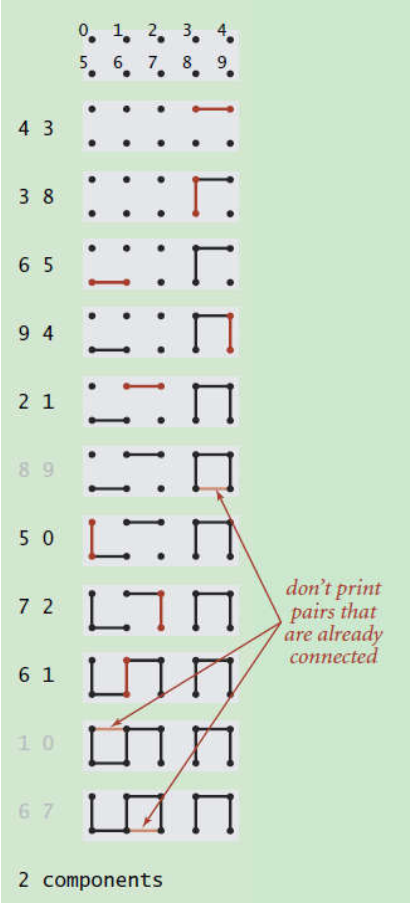
并查集(Union-Find)算法介绍

原创 2012年06月12日 13:57:16 标签：algorithm / Algorithm / 并查集 / 数据结构 / 算法

本文主要介绍解决动态连通性一类问题的一种算法，使用到了一种叫做并查集的数据结构，称为Union-Find。
更多的信息可以参考Algorithms 一书的Section 1.5，实际上本文也就是基于它的一篇读后感吧。
原文中更多的是给出一些结论，我尝试给出一些思路上的过程，即为什么要使用这个方法，而不是别的什么方法。我觉得这个可能更加有意义一些，相比于记下一些结论。

关于动态连通性

我们看一张图来了解一下什么是动态连通性：



假设我们输入了一组整数对，即上图中的(4, 3) (3, 8)等等，每对整数代表这两个points/sites是连通的。那么随着数据的不断输入，整个图的连通性也会发生变化，从上图中可以很清晰的发现这一点。同时，对于已经处于连通状态的points/sites，直接忽略，比如上图中的(8,
9)。

动态连通性的应用场景：

- 网络连接判断：

如果每个pair中的两个整数分别代表一个网络节点，那么该pair就是用来表示这两个节点是需要连通的。那么为所有的pairs建立了动态连通图后，就能够尽可能少的减少布线的需要，因为已经连通的两个节点会被直接忽略掉。

- 变量名等同性(类似于指针的概念)：
在程序中，可以声明多个引用来指向同一对象，这个时候就可以通过为程序中声明的引用和实际对象建立动态连通图来判断哪些引用实际上是指向同一对象。

对问题建模：

在对问题进行建模的时候，我们应该尽量想清楚需要解决的问题是什么。因为模型中选择的数据结构和算法显然会根据问题的不同而不同，就动态连通性这个场景而言，我们需要解决的问题可能是：

- 给出两个节点，判断它们是否连通，如果连通，不需要给出具体的路径
- 给出两个节点，判断它们是否连通，如果连通，需要给出具体的路径

就上面两种问题而言，虽然只有是否能够给出具体路径的区别，但是这个区别导致了选择算法的不同，本文主要介绍的是第一种情况，即不需要给出具体路径的Union-Find算法，而第二种情况可以使用基于DFS的算法。

建模思路：

最简单而直观的假设是，对于连通的所有节点，我们可以认为它们属于一个组，因此不连通的节点必然就属于不同的组。随着Pair的输入，我们需要首先判断输入的两个节点是否连通。如何判断呢？按照上面的假设，我们可以通过判断它们属于的组，然后看看这两个组是否相同，如果相同，那么这两个节点连通，反之不连通。为简单起见，我们将所有的节点以整数表示，即对N个节点使用0到N-1的整数表示。而在处理输入的Pair之前，每个节点必然都是孤立的，即他们分属于不同的组，可以使用数组来表示这一层关系，数组的index是节点的整数表示，而相应的值就是该节点的组号了。该数组可以初始化为：

```[java]
1. for(int i = 0; i < size; i++)
2.     id[i] = i;
```

即对于节点i，它的组号也是i。

初始化完毕之后，对该动态连通图有几种可能的操作：

- 查询节点属于的组

数组对应位置的值即为组号

- 判断两个节点是否属于同一个组

分别得到两个节点的组号，然后判断组号是否相等

- 连接两个节点，使之属于同一个组

分别得到两个节点的组号，组号相同时操作结束，不同时，将其中的一个节点的组号换成另一个节点的组号

- 获取组的数目

初始化为节点的数目，然后每次成功连接两个节点之后，递减1

API

我们可以设计相应的API：

public class UF

	UF(int N)	*initialize N sites with integer names (0 to N-1)*
void	union(int p, int q)	*add connection between p and q*
int	find(int p)	*component identifier for p (0 to N-1)*
boolean	connected(int p, int q)	*return true if p and q are in the same component*
int	count()	*number of components*

Union-find API

注意其中使用整数来表示节点，如果需要使用其他的数据类型表示节点，比如使用字符串，那么可以用哈希表来进行映射，即将String映射成这里需要的Integer类型。

分析以上的API，方法connected和union都依赖于find，connected对两个参数调用两次find方法，而union在真正执行union之前也需要判断是否连通，这又是两次调用find方法。因此我们需要把find方法的实现设计的尽可能的高效。所以就有了下面的Quick-Find实现。

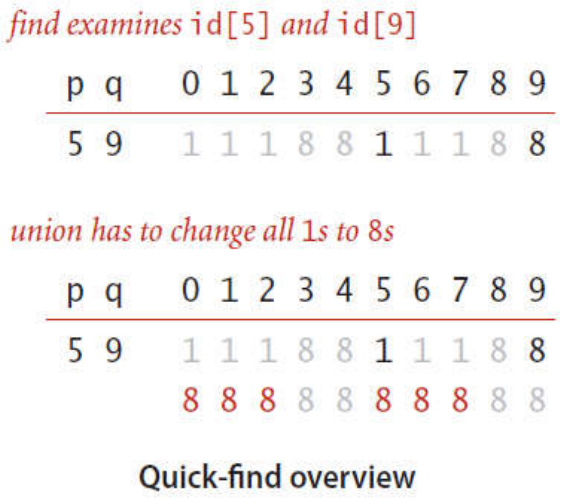
Quick-Find 算法：

[java]

```java
public class UF
{
    private int[] id; // access to component id (site indexed)
    private int count; // number of components
    public UF(int N)
    {
        // Initialize component id array.
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }
    public int count()
    { return count; }
    public boolean connected(int p, int q)
    { return find(p) == find(q); }
    public int find(int p)
    { return id[p]; }
    public void union(int p, int q)
    {
```

```
21.             // 获得p和q的组号
22.             int pID = find(p);
23.             int qID = find(q);
24.             // 如果两个组号相等，直接返回
25.             if (pID == qID) return;
26.             // 遍历一次，改变组号使他们属于一个组
27.             for (int i = 0; i < id.length; i++)
28.                 if (id[i] == pID) id[i] = qID;
29.             count--;
30.         }
31.     }
```

举个例子，比如输入的Pair是(5， 9)，那么首先通过find方法发现它们的组号并不相同，然后在union的时候通过一次遍历，将组号1都改成8。当然，由8改成1也是可以的，保证操作时都使用一种规则就行。

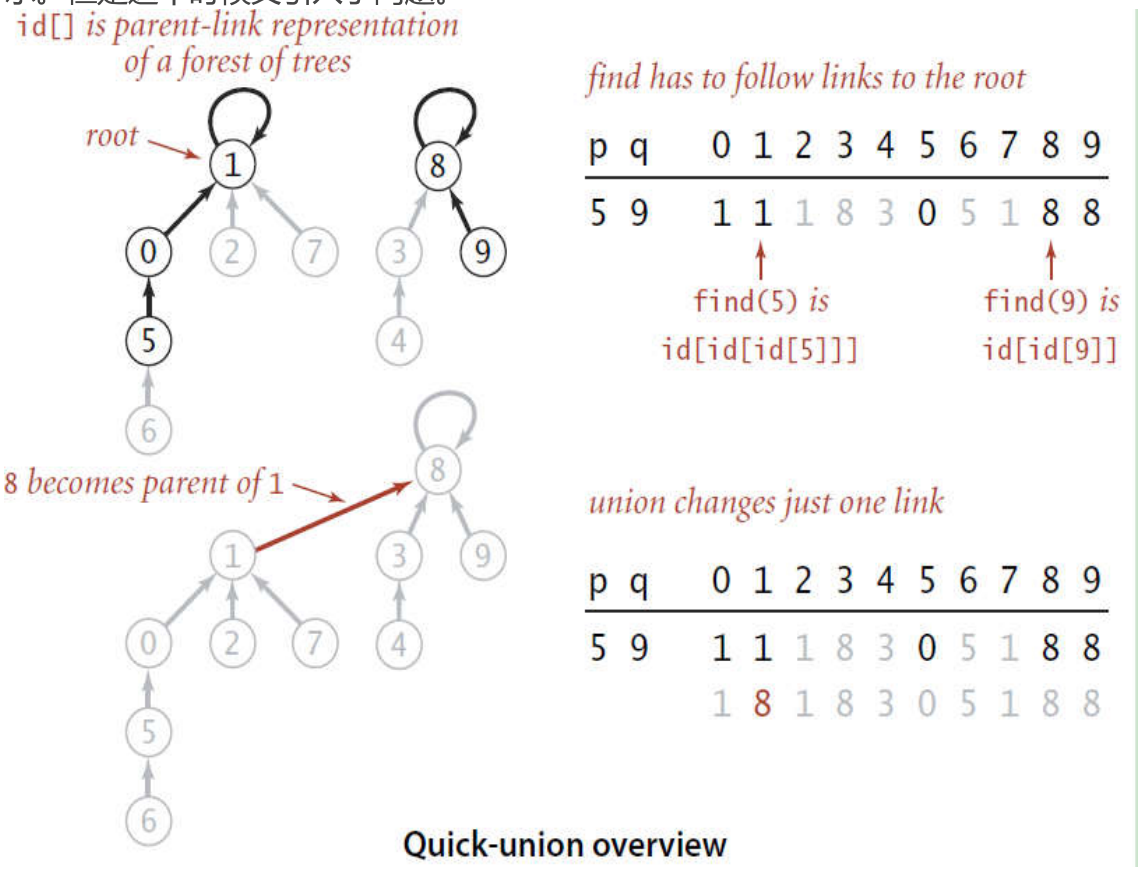


上述代码的find方法十分高效，因为仅仅需要一次数组读取操作就能够找到该节点的组号，但是问题随之而来，对于需要添加新路径的情况，就涉及到对于组号的修改，因为并不能确定哪些节点的组号需要被修改，因此就必须对整个数组进行遍历，找到需要修改的节点，逐一修改，这一下每次添加新路径带来的复杂度就是线性关系了，如果要添加的新路径的数量是M，节点数量是N，那么最后的时间复杂度就是MN，显然是一个平方阶的复杂度，对于大规模的数据而言，平方阶的算法是存在问题的，这种情况下，每次添加新路径就是“牵一发而动全身”，想要解决这个问题，关键就是要提高union方法的效率，让它不再需要遍历整个数组。

Quick-Union 算法：

考虑一下，为什么以上的解法会造成“牵一发而动全身”？因为每个节点所属的组号都是单独记录，各自为政的，没有将它们以更好的方式组织起来，当涉及到修改的时候，除了逐一通知、修改，别无他法。所以现在的问题就变成了，如何将节点以更好的方式组织起来，组织的方式有很多种，但是最直观的还是将组号相同的节点组织在一起，想想所学的数据结构，什么样子的数据结构能够将一些节点给组织起来？常见的就是链表，图，树，什么的了。但是哪种结构对于查找和修改的效率最高？毫无疑问是树，因此考虑如何将节点和组的关系以树的形式表现出来。

如果不改变底层数据结构，即不改变使用数组的表示方法的话。可以采用parent-link的方式将节点组织起来，举例而言，id[p]的值就是p节点的父节点的序号，如果p是树根的话，id[p]的值就是p，因此最后经过若干次查找，一个节点总是能够找到它的根节点，即满足id[root] = root的节点也就是组的根节点了，然后就可以使用根节点的序号来表示组号。所以在处理一个pair的时候，将首先找到pair中每一个节点的组号(即它们所在树的根节点的序号)，如果属于不同的组的话，就将其中一个根节点的父节点设置为另外一个根节点，相当于将一颗独立的树编程另一颗独立的树的子树。直观的过程如下图所示。但是这个时候又引入了问题。



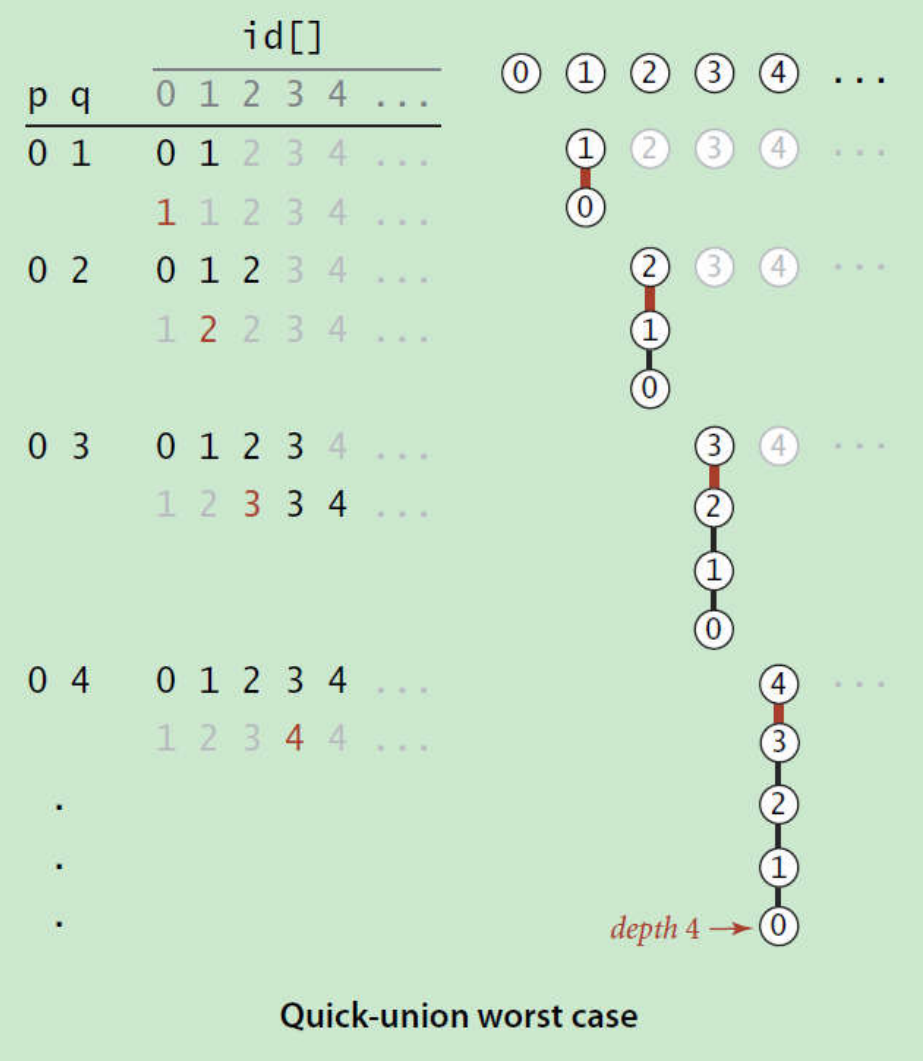
Quick-union overview

在实现上，和之前的Quick-Find只有find和union两个方法有所不同：

```java
private int find(int p)
{
    // 寻找p节点所在组的根节点，根节点具有性质id[root] = root
    while (p != id[p]) p = id[p];
    return p;
}
public void union(int p, int q)
{
    // Give p and q the same root.
    int pRoot = find(p);
    int qRoot = find(q);
```

```
12.     if (pRoot == qRoot)
13.         return;
14.     id[pRoot] = qRoot;     // 将一颗树(即一个组)变成另外一课树(即一个组)的子树
15.     count--;
16. }
```

树这种数据结构容易出现极端情况，因为在建树的过程中，树的最终形态严重依赖于输入数据本身的性质，比如数据是否排序，是否随机分布等等。比如在输入数据是有序的情况下，构造的BST会退化成一个链表。在我们这个问题中，也是会出现的极端情况的，如下图所示。



Quick-union worst case

为了克服这个问题，BST可以演变成为红黑树或者AVL树等等。

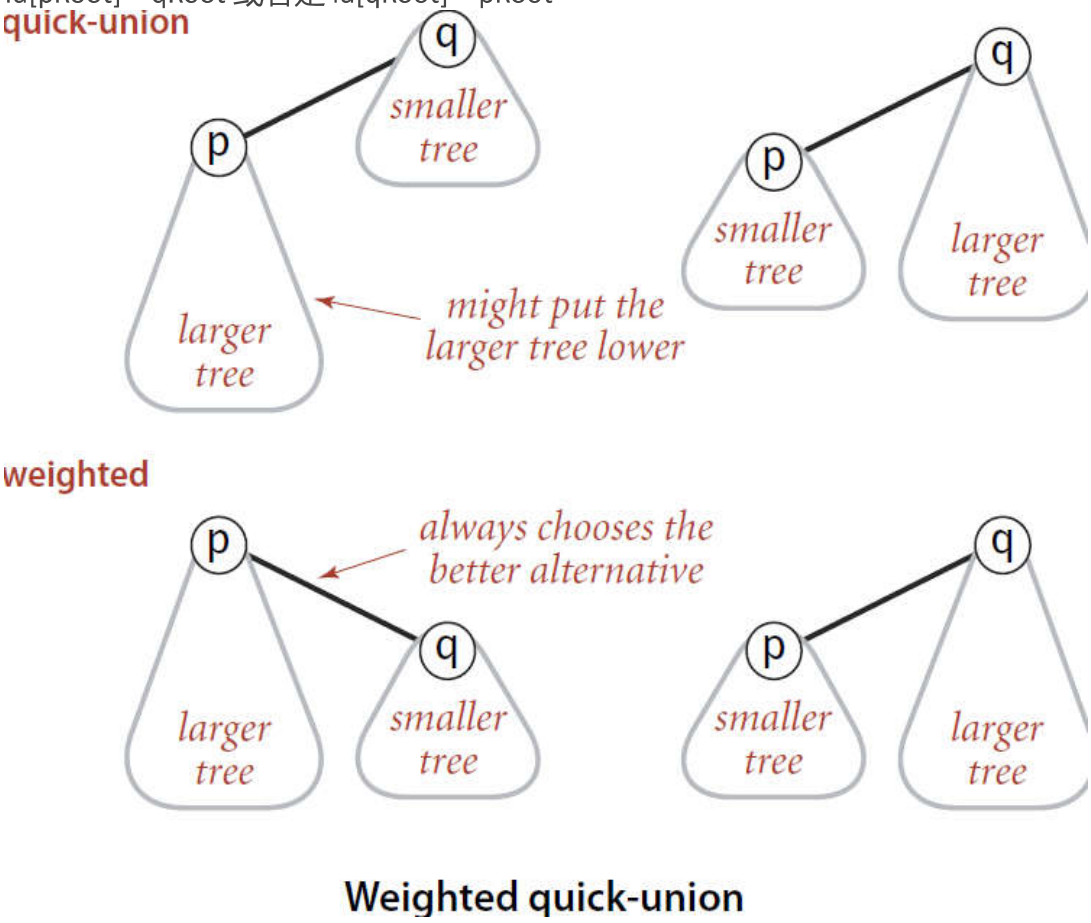
然而，在我们考虑的这个应用场景中，每对节点之间是不具备可比性的。因此需要想其它的办法。在没有什么思路的时候，多看看相应的代码可能会有一些启发，考虑一下Quick-Union算法中的union方法实现：

```
[java]
1. public void union(int p, int q)
2. {
3.     // Give p and q the same root.
```

```
4.      int pRoot = find(p);
5.      int qRoot = find(q);
6.      if (pRoot == qRoot)
7.          return;
8.      id[pRoot] = qRoot;  // 将一颗树(即一个组)变成另外一课树(即一个组)的子树
9.      count--;
10. }
```

上面 id[pRoot] = qRoot 这行代码看上去似乎不太对劲。因为这也属于一种“硬编码”，这样实现是基于一个约定，即p所在的树总是会被作为q所在树的子树，从而实现两颗独立的树的融合。那么这样的约定是不是总是合理的呢？显然不是，比如p所在的树的规模比q所在的树的规模大的多时，p和q结合之后形成的树就是十分不和谐的一头轻一头重的”畸形树“了。

所以我们应该考虑树的大小，然后再来决定到底是调用：
id[pRoot] = qRoot 或者是 id[qRoot] = pRoot



Weighted quick-union

即总是size小的树作为子树和size大的树进行合并。这样就能够尽量的保持整棵树的平衡。

所以现在的问题就变成了：树的大小该如何确定？
我们回到最初的情形，即每个节点最一开始都是属于一个独立的组，通过下面的代码进行初始化：

```java
for (int i = 0; i < N; i++)
    id[i] = i;    // 每个节点的组号就是该节点的序号
```

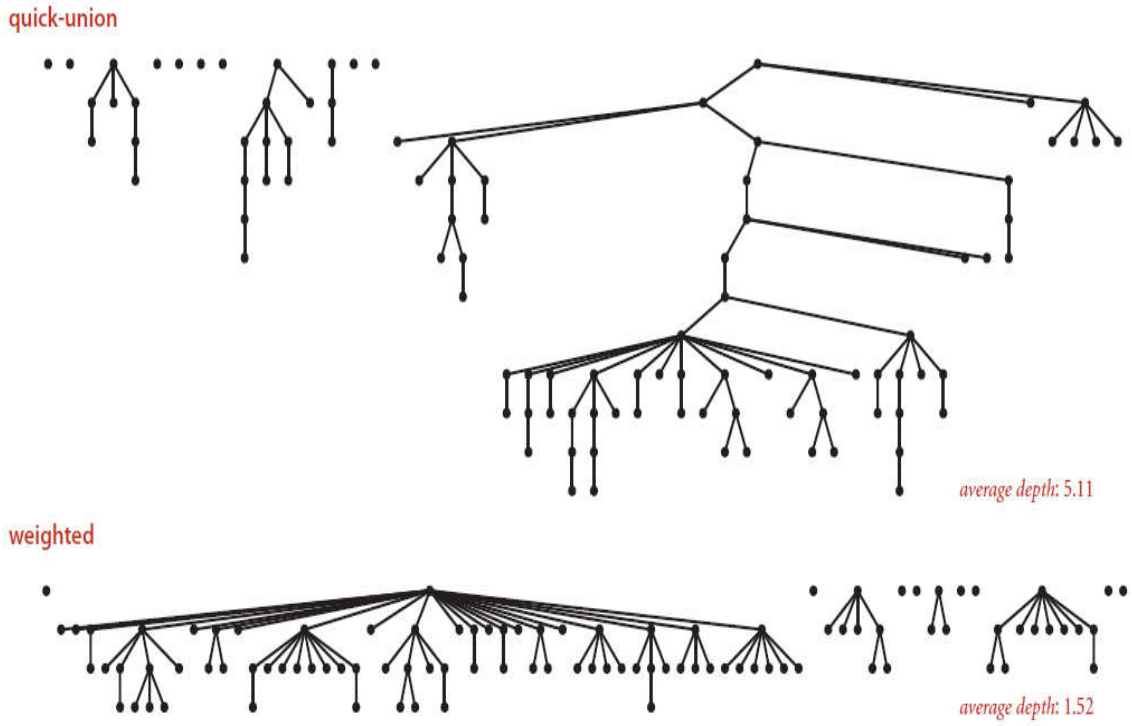
以此类推，在初始情况下，每个组的大小都是1，因为只含有一个节点，所以我们可以使用额外的一个数组来维护每个组的大小，对该数组的初始化也很直观：

```java
for (int i = 0; i < N; i++)
    sz[i] = 1;    // 初始情况下，每个组的大小都是1
```

而在进行合并的时候，会首先判断待合并的两棵树的大小，然后按照上面图中的思想进行合并，实现代码：

```java
public void union(int p, int q)
{
    int i = find(p);
    int j = find(q);
    if (i == j) return;
    // 将小树作为大树的子树
    if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
    count--;
}
```

Quick-Union 和 Weighted Quick-Union 的比较：



Quick-union and weighted quick-union (100 sites, 88 union() operations)

可以发现，通过sz数组决定如何对两棵树进行合并之后，最后得到的树的高度大幅度减小了。这是十分有意义的，因为在Quick-Union算法中的任何操作，都不可避免的需要调用find方法，而该方法的执行效率依赖于树的高度。树的高度减小了，find方法的效率就增加了，从而也就增加了整个Quick-Union算法的效率。

上图其实还可以给我们一些启示，即对于Quick-Union算法而言，节点组织的理想情况应该是一颗十分扁平的树，所有的孩子节点应该都在height为1的地方，即所有的孩子都直接连接到根节点。这样的组织结构能够保证find操作的最高效率。

那么如何构造这种理想结构呢？
在find方法的执行过程中，不是需要进行一个while循环找到根节点嘛？如果保存所有路过的中间节点到一个数组中，然后在while循环结束之后，将这些中间节点的父节点指向根节点，不就行了么？但是这个方法也有问题，因为find操作的频繁性，会造成频繁生成中间节点数组，相应的分配销毁的时间自然就上升了。那么有没有更好的方法呢？还是有的，即将节点的父节点指向该节点的爷爷节点，这一点很巧妙，十分方便且有效，相当于在寻找根节点的同时，对路径进行了压缩，使整个树结构扁平化。相应的实现如下，实际上只需要添加一行代码：

[java]

```java
private int find(int p)
{
    while (p != id[p])
    {
        // 将p节点的父节点设置为它的爷爷节点
        id[p] = id[id[p]];
        p = id[p];
    }
```

```
9.      return p;
10.  }
```

至此，动态连通性相关的Union-Find算法基本上就介绍完了，从容易想到的Quick-Find到相对复杂但是更加高效的Quick-Union，然后到对Quick-Union的几项改进，让我们的算法的效率不断的提高。
这几种算法的时间复杂度如下所示：

Algorithm	Constructor	Union	Find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted Quick-Union	N	lgN	lgN
Weighted Quick-Union With Path Compression	N	Very near to 1 (amortized)	Very near to 1 (amortized)

对大规模数据进行处理，使用平方阶的算法是不合适的，比如简单直观的Quick-Find算法，通过发现问题的更多特点，找到合适的数据结构，然后有针对性的进行改进，得到了Quick-Union算法及其多种改进算法，最终使得算法的复杂度降低到了近乎线性复杂度。

如果需要的功能不仅仅是检测两个节点是否连通，还需要在连通时得到具体的路径，那么就需要用到别的算法了，比如DFS或者BFS。

并查集的应用，可以参考另外一篇文章
并查集应用举例