



CODE > JAVASCRIPT

# JavaScript数据结构：树

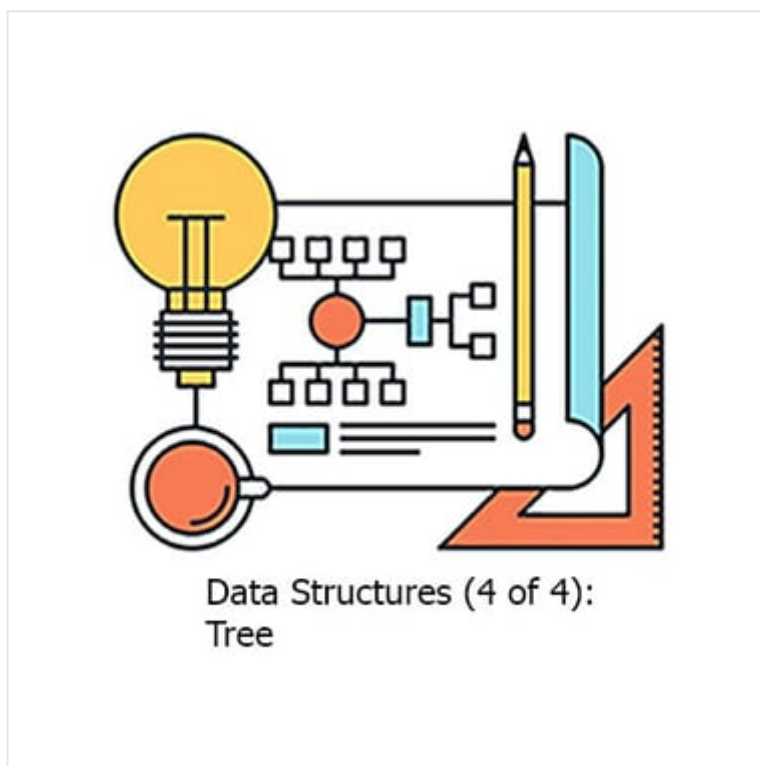
by [Cho S. Kim](#) Sep 24, 2015

Read Time: 14 mins Languages: 中文 (简体)

This post is part of a series called [Data Structures in JavaScript](#).

[Data Structures With JavaScript: Singly-Linked List and Doubly-Linked List](#)

Chinese (Simplified) (中文 (简体)) translation by [Alixwang](#) (you can also [view the original English article](#))



What You'll Be Creating

树是在 web 开发中最常用的数据结构之一。 这种说法对开发者和用户都是正

确的。 每个 HTML 的开发者 在网站中 添加这些就创建一个树。 这个树通常被

用的。每个与HTML的开发者，把网页载入浏览器就创建了一棵树，这棵树通常叫做文档对象模型（DOM）。相应地，每个网络上的消费信息的人，接受信息也以DOM树的形式。每个编写HTML并将其加载到Web浏览器的Web开发人员都创建了一个树，称为文档对象模型（DOM）。互联网上的每一个用户，在互联网上获取信息的时候，都是以树的形式收到的-DOM。

现在，高潮来了：你正在读的本文在浏览器中就是以树的形式进行渲染的。文字由<p>元素进行表示；<p>元素又嵌套在<body>元素中；<body>元素又嵌套在<html>元素中。您正在阅读的段落表示为<p>元素中的文本；<p>元素嵌套在<body>元素中；<body>元素嵌套在<html>元素中。

这些嵌套数据和家族数类似。<html>是父元素，<body>是子元素，<p>又是<body>的子元素 如果这个比喻对你有点用的话，你将会发现在我们介绍树的时候会用到更多的类比。

在本文中，我们将会通过两个不同的树遍历方式来创建一个树：深度优先(DFS)和广度优先(BFS)。（如果你对遍历这个词感到比较陌生，不妨将他想象成访问树中的每一个节点。）这两种类型的遍历强调了与树交互的不同方式，DFS和BFS分别用栈和队列来访问节点。听起来很酷！！

## 树（深度搜索和广度搜索）

在计算机科学中，树是一种用节点来模拟分层数据的数据结构。每个树节点都包含他本身的数据及指向其他节点的指针。

节点和指针这些术语可能对一些读者来说比较陌生，所以让我们用类比来进一步描述他们。让我们将树与组织图结构图进行比较。这个结构图有一个顶级位置（根节点），比如CEO。在这个节点下面还有一些其他的节点，比如副总裁（VP）。

为了表示这种关系，我们用箭头从CEO指向VP。一个位置，比如CEO，是一个节点；我们创建的CEO到VP的关系是一个指针。在我们的组织结构图中去创建更多的关系，我们只要重复这些步骤即可---我们让一个节点指向另一个节点。

在概念层次上，我希望节点和指针有意义。在实际中，我们能从更科学的实例中获取收益。让我们来思考DOM。DOM有<html>元素作为其顶级位置（根节

点)。这个节点指向 `<head>` 元素和 `<body>` 元素。这些步骤在DOM的所有节点中重复。

这种设计的一个优点是能够嵌套节点：例如：一个 `<ul>` 元素能够包含很多个 `<li>` 元素；此外，每个 `<li>` 元素能拥有兄弟 `<li>` 元素。着很怪异，但是确实真实有趣！

## 操作树

由于每个树都包含节点，其可以是来自树的单独构造器，我们将概述两个构造函数的操作：`Node` 和 `Tree`

## 节点

- `data` 存储值。
- `parent` 指向节点的父节点。
- `children`指向列表中的下一个节点。

## 树

- `_root`指向一个树的根节点。
- `traverseDF(callback)` 对树进行DFS遍历。
- `traverseBF(callback)` 对树进行BFS遍历。
- `contains(data, traversal)` 搜索树中的节点。
- `add(data, toData, traverse)` 向树中添加节点。
- `remove(child, parent)` 移除树中的节点。

Advertisement

## 实现树

现在让我们写下树的代码！

## 节点的属性

在我们的实现中，我们首先定义一个叫做 `Node` 的函数，然后构造一个 `Tree`。

```
1 function Node(data) {  
2     this.data = data;  
3     this.parent = null;  
4     this.children = [];  
5 }
```

---

每一个 `Node` 实例都包含三个属性：`data`，`parent`，和 `children`。第一个属性保存与节点相关联的数据。第二个属性指向一个节点。第三个属性指向许多子节点。

## 树的属性

现在让我们来定义 `Tree` 的构造函数，其中包括 `Node` 构造函数的定义：

```
1 function Tree(data) {  
2     var node = new Node(data);  
3     this._root = node;  
4 }
```

---

`Tree` 包含两行代码。第一行创建了一个 `Node` 的新实例；第二行让 `node` 等于树的根节点。

`Tree` 和 `Node` 的定义只需要几行代码。但是，通过这几行足以帮助我们模拟分层数据。为了证明这一点，让我们用一些示例数据去创建 `Tree` 的示例（和间接的 `Node`）。

```
1 var tree = new Tree('CEO');  
2  
3 // {data: 'CEO', parent: null, children: []}  
4 tree._root;
```

幸亏有 `parent` 和 `children` 的存在，我们可以为 `_root` 添加子节点和让这些子节点的父节点等于 `_root`。换一种说法，我们可以模拟分层数据的创建。

## Tree的方法

接下来我们将要创建以下五种方法。

### 树

1. `traverseDF(callback)`
2. `traverseBF(callback)`
3. `contains(data, traversal)`
4. `add(child, parent)`
5. `remove(node, parent)`

因为每种方法都需要我们去遍历一个树，所以我们首先要实现一个方法去定义不同的树遍历。（遍历树是一种正式的方式来访问树的每个节点。）

#### 1 of 5: `traverseDF(callback)`

这种方法以DFS这种方式遍历树。

```
01 | Tree.prototype.traverseDF = function(callback) {
02 |
03 |     // this is a recurse and immediately-invoking function
04 |     (function recurse(currentNode) {
05 |         // step 2
06 |         for (var i = 0, length = currentNode.children.length; i < length; i++) {
07 |             // step 3
08 |             recurse(currentNode.children[i]);
09 |         }
10 |
11 |         // step 4
12 |         callback(currentNode);
13 |
14 |         // step 1
15 |     })(this._root);
16 |
17 | };
```

`traverseDF(callback)` 有一个参数 `callback`

如果对这个名字不明白

`callback` 被假定

`traverseDF(callback)` 是一个函数，将在后面被 `traverseDF(callback)` 调用。

`traverseDF(callback)` 的函数体含有另一个叫做 `recurse` 的函数。这个函数是一个递归函数！换句话说，它是自我调用和自我终止。使用 `recurse` 的注释中提到的步骤，我将描述递归用来 `recurse` 整个树的一般过程。

这里是步骤：

1. 立即使用树的根节点作为其参数调用 `recurse`。此时，`currentNode` 指向当前节点。
2. 进入 `for` 循环并且从第一个子节点开始，每一个子节点都迭代一次 `currentNode` 函数。
3. 在 `for` 循环体内，使用 `currentNode` 的子元素调用递归。确切的子节点取决于当前 `for` 循环的当前迭代。
4. 当 `currentNode` 不存在子节点时，我们退出 `for` 循环并 `callback` 我们在调用 `traverseDF(callback)` 期间传递的回调。

步骤2（自终止），3（自调用）和4（回调）重复，直到我们遍历树的每个节点。

递归是一个非常困难的话题，需要一个完整的文章来充分解释它。由于递归的解释不是本文的重点 - 重点是实现一棵树 - 我建议任何读者没有很好地掌握递归做以下两件事。

首先，实验我们当前的 `traverseDF(callback)` 实现，并尝试一定程度上理解它是如何工作的。第二，如果你想要我写一篇关于递归的文章，那么请在本文的评论中请求它。

以下示例演示如何使用 `traverseDF(callback)` 遍历树。要遍历树，我将在下面的示例中创建一个。我现在使用的方法不是理想的，但它能很好的工作。一个更好的方法是使用 `add(value)`，我们将在第4步和第5步中实现。

```
01 | var tree = new Tree('one');
02 |
03 | tree._root.children.push(new Node('two'));
04 | tree._root.children[0].parent = tree;
05 |
```

```

06 tree._root.children.push(new Node('three'));
07 tree._root.children[1].parent = tree;
08
09
10 tree._root.children.push(new Node('four'));
11 tree._root.children[2].parent = tree;
12
13 tree._root.children[0].children.push(new Node('five'));
14 tree._root.children[0].children[0].parent = tree._root.children[0];
15
16 tree._root.children[0].children.push(new Node('six'));
17 tree._root.children[0].children[1].parent = tree._root.children[0];
18
19 tree._root.children[2].children.push(new Node('seven'));
20 tree._root.children[2].children[0].parent = tree._root.children[2];
21
22 /*
23
24 creates this tree
25
26   one
27   |
28   |--- two --- five
29   |           |
30   |           |--- six
31   |--- three
32   |--- four --- seven
33
34 */

```

现在，让我们调用 `traverseDF(callback)`

```

01 tree.traverseDF(function(node) {
02     console.log(node.data)
03 });
04
05 /*
06
07 logs the following strings to the console
08
09 'five'
10 'six'
11 'two'
12 'three'
13 'seven'
14 'four'
15 'one'
16
17 */

```

## 2 of 5: `traverseBF(callback)`

这个方法使用深度优先搜索去遍历树

深度优先搜索和广度优先搜索之间的差别涉及树的节点访问的序列。 为了说明

这一点，让我们使用我们从 `traverseDF (callback)` 创建的树。

```

01
02  /*
03
04     tree
05
06     one (depth: 0)
07     |
08     |--- two (depth: 1)
09     |       |
10     |       |--- five (depth: 2)
11     |       |--- six (depth: 2)
12     |--- three (depth: 1)
13     |--- four (depth: 1)
           |
           |--- seven (depth: 2)
  
```

现在，让我们传递 `traverseBF (callback)` 和我们用于 `traverseDF (callback)` 的回调。

```

01  tree.traverseBF(function (node) {
02      console.log(node.data)
03  });
04
05  /*
06
07  logs the following strings to the console
08
09  'one'
10  'two'
11  'three'
12  'four'
13  'five'
14  'six'
15  'seven'
16
17  */
  
```

来自控制台的日志和我们的树的图显示了关于广度优先搜索的模式。从根节点开始;然后行进一个深度并访问该深度从左到右的每个节点。重复此过程，直到没有更多的深度要移动。

由于我们有一个广度优先搜索的概念模型，现在让我们实现使我们的示例工作的代码。

```

01  Tree.prototype.traverseBF = function (callback) {
02      var queue = new Queue();
03
04      queue.enqueue(this._root);
05
06      currentTree = queue.dequeue();
07
08      while (currentTree) {
09
  
```



```

09         for (var i = 0, length = currentTree.children.length; i < length; i++) {
10             queue.enqueue(currentTree.children[i]);
11         }
12
13
14         callback(currentTree);
15         currentTree = queue.dequeue();
16     }
};

```

我们对 `traverseBF(callback)` 的定义包含了很多逻辑。因此，我将以可管理的步骤解释逻辑：

1. 创建 `Queue` 的实例。
2. 将调用 `traverseBF(callback)` 的节点添加到 `Queue` 的实例。
3. 定义一个变量 `currentNode` 并且将他的值初始化为刚才添加到队列里的 `node`
4. 当 `currentNode` 指向一个节点时，执行 `while` 循环里面的代码。
5. 用 `for` 循环去迭代 `currentNode` 的子节点。
6. 在 `for` 循环体内，将每个子元素加入队列。
7. 获取 `currentNode` 并将其作为 `callback` 的参数传递。
8. 将 `currentNode` 重新分配给正从队列中删除的节点。
9. 直到 `currentNode` 不在指向任何节点-也就是说树中的每个节点都访问过了-重复4-8步。

## contains(callback, traversal)

让我们定义一个方法，让我们在树中搜索一个特定的值。去使用我们创建的任一种树遍历方法，我们已经定义了 `contains(callback, traversal)` 接收两个参数：搜索的数据和遍历的类型。

```

1  Tree.prototype.contains = function(callback, traversal) {
2      traversal.call(this, callback);
3  };

```

在 `contains(callback, traversal)` 函数体内，我们用 `call` 方法去传递 `this` 和 `callback`。第一个参数将 `traversal` 绑定到被调用的树 `contains(callback, traversal)`；

想象一下，我们要将包含奇数数据的任何节点记录到控制台，并使用BFS遍历树中的每个节点。代码我们可以这么写：

```

1  // tree is an example of a root node
2  tree.contains(function(node) {
3      if (node.data === 'two') {
4          console.log(node);
5      }
6  }, tree.traverseBF);

```

## add(data, toData, traversal)

现在我们有了一个可以搜索树中特定节点的方法。现在让我们定义一个允许我们向指定节点添加节点的方法。

```

01  Tree.prototype.add = function(data, toData, traversal) {
02      var child = new Node(data),
03          parent = null,
04          callback = function(node) {
05              if (node.data === toData) {
06                  parent = node;
07              }
08          };
09
10      this.contains(callback, traversal);
11
12      if (parent) {
13          parent.children.push(child);
14          child.parent = parent;
15      } else {
16          throw new Error('Cannot add node to a non-existent parent. ');
17      }
18  };

```

`add(data, toData, traversal)` 定义了三个参数。第一个参数，`data`，用来创建一个 `Node` 的新实例。第二个参数，`toData`，用来比较树中的每个节点。第三个参数，`traversal`，是这个方法中用来遍历树的类型。

在 `add(data, toData, traversal)` 函数体内，我们声明了三个变量。第一个变量，`child`，代表初始化的 `Node` 实例。第二个变量，`parent`，初始化为 `null`；但是将来会指向匹配 `toData` 值的树中的任意节点。`parent` 的重新分配发生在我们声明的第三个变量，这就是 `callback`。

`callback` 是一个将 `toData` 和每一个节点的 `data` 属性做比较的函数。如果 `if` 语句的值是 `true`，那么 `parent` 将被赋值给 `if` 语句中匹配比较的节点。

每个节点的 `toData` 比较发生在 `contains(callback, traversal)`。遍历类型和 `callback` 必须

作为 `contains(callback, traversal)` 的参数进行传递。

最后，如果 `parent` 不存在于树中，我们将 `child` 推入 `parent.children`；同时也要将 `parent` 赋值给 `child` 的父级。否则，将抛出错误。

让我们用 `add(data, toData, traversal)` 做个例子：

```
01 | var tree = new Tree('CEO');
02 |
03 | tree.add('VP of Happiness', 'CEO', tree.traverseBF);
04 |
05 | /*
06 |
07 | our tree
08 |
09 | 'CEO'
10 |   └── 'VP of Happiness'
11 |
12 | */
```

---

这里是 `add(addData, toData, traversal)` 的更加复杂的例子：

```
01 | var tree = new Tree('CEO');
02 |
03 | tree.add('VP of Happiness', 'CEO', tree.traverseBF);
04 | tree.add('VP of Finance', 'CEO', tree.traverseBF);
05 | tree.add('VP of Sadness', 'CEO', tree.traverseBF);
06 |
07 | tree.add('Director of Puppies', 'VP of Finance', tree.traverseBF);
08 | tree.add('Manager of Puppies', 'Director of Puppies', tree.traverseBF);
09 |
10 | /*
11 |
12 | tree
13 |
14 | 'CEO'
15 |   ├── 'VP of Happiness'
16 |   ├── 'VP of Finance'
17 |       ├── 'Director of Puppies'
18 |       ├── 'Manager of Puppies'
19 |   └── 'VP of Sadness'
20 |
21 | */
```

---

## remove(data, fromData, traversal)

为了完成 `Tree` 的实现，我们将添加一个叫做 `remove(data, fromData, traversal)` 的方法。跟从DOM里面移除节点类似，这个方法将移除一个节点和他的所有子级。

```
01 | Tree.prototype.remove = function(data, fromData, traversal) {
02 |     var tree = this,
03 |         parent = null,
04 |         childToRemove = null,
05 |         index;
06 |
07 |     var callback = function(node) {
08 |         if (node.data === fromData) {
09 |             parent = node;
10 |         }
11 |     };
12 |
13 |     this.contains(callback, traversal);
14 |
15 |     if (parent) {
16 |         index = findIndex(parent.children, data);
17 |
18 |         if (index === undefined) {
19 |             throw new Error('Node to remove does not exist.');
```

与 `add(data, toData, traversal)` 类似，移除将遍历树以查找包含第二个参数的节点，现在为 `fromData`。如果这个节点被发现了，那么 `parent` 将指向它。

在这时候，我们到达了第一个 `if` 语句。如果 `parent` 不存在，将抛出错误。如果 `parent` 不存在，我们使用 `parent.children` 调用 `findIndex()` 和我们要从 `parent` 节点的孩子节点中删除的数据（`findIndex()` 是一个帮助方法，我将在下面定义。）

```
01 | function findIndex(arr, data) {
02 |     var index;
03 |
04 |     for (var i = 0; i < arr.length; i++) {
05 |         if (arr[i].data === data) {
06 |             index = i;
07 |         }
08 |     }
09 |
10 |     return index;
11 | }
```

在 `findIndex()` 里面，以下逻辑将发生。如果 `parent.children` 中的任意一个节点包含匹配 `data` 值的数据，那么变量 `index` 赋值为一个整数。如果没有子级的数值属性匹配 `data`，那么 `index` 保留他的默认值 `undefined`。在最后一行的 `findIndex()` 方法，我

们返回一个 `index`。

我们现在去 `remove(data, fromData, traversal)` 如果 `index` 的值是 `undefined`，将会抛出错误。如果 `index` 的值存在，我们用它来拼接我们想从 `parent` 的子节点中删除的节点。同样我们给删除的子级赋值为 `childToRemove`。

最后，我们返回 `childToRemove`。

## 至此我们完成了Tree的完整实现。

我们的 `Tree` 实现已经完成。回过头看看-我们完成了很多工作：

```

001 function Node(data) {
002     this.data = data;
003     this.parent = null;
004     this.children = [];
005 }
006
007 function Tree(data) {
008     var node = new Node(data);
009     this._root = node;
010 }
011
012 Tree.prototype.traverseDF = function(callback) {
013
014     // this is a recurse and immediately-invoking function
015     (function recurse(currentNode) {
016         // step 2
017         for (var i = 0, length = currentNode.children.length; i < length; i++) {
018             // step 3
019             recurse(currentNode.children[i]);
020         }
021
022         // step 4
023         callback(currentNode);
024
025         // step 1
026     })(this._root);
027
028 };
029
030 Tree.prototype.traverseBF = function(callback) {
031     var queue = new Queue();
032
033     queue.enqueue(this._root);
034
035     currentTree = queue.dequeue();
036
037     while (currentTree) {
038         for (var i = 0, length = currentTree.children.length; i < length; i++) {
039             queue.enqueue(currentTree.children[i]);
040         }
041
042         callback(currentTree);

```

```
043         // ... ,
044         currentTree = queue.dequeue();
045     };
046
047     Tree.prototype.contains = function(callback, traversal) {
048         traversal.call(this, callback);
049     };
050
051     Tree.prototype.add = function(data, toData, traversal) {
052         var child = new Node(data),
053             parent = null,
054             callback = function(node) {
055                 if (node.data === toData) {
056                     parent = node;
057                 }
058             };
059
060         this.contains(callback, traversal);
061
062         if (parent) {
063             parent.children.push(child);
064             child.parent = parent;
065         } else {
066             throw new Error('Cannot add node to a non-existent parent.');
```

```
108 |     return index;  
109 | }
```

## 总结

树的模拟分层数据。 我们的周围有许多类似于这种类型的层次结构，如网页和我们的家庭。 任何时候你发现自己需要结构化数据与层次结构，考虑使用树。



关注我们的公众号



Advertisement

JavaScript Programming Fundamentals





Cho S. Kim

San Francisco, CA

Cho is a full-stack web-application developer. He dislikes mean people but likes the MEAN stack (MongoDB, ExpressJS, AngularJS, Node.js). During a typical week, he'll be coding in JavaScript, writing about JavaScript, or watching movies NOT about JavaScript.

 [choskim](#)

 FEED  LIKE  FOLLOW

## Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

[Update me weekly](#)

[View on GitHub](#)



Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTORIALS




JOIN OUR COMMUNITY



HELP





30,415

Tutorials

1,316

Courses

49,884

Translations

tuts+

30,415 Tutorials

Certified B Corporation

30,415 Tutorials

Envato Envato Elements Envato Market Placeit by Envato Milkshake All products Careers Sitemap

© 2021 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

