实战 7：树形控件——Tree

本小节基于 Vue.js 的递归组件知识，来开发一个常见的树形控件—Tree。

Tree 组件是递归类组件的典型代表，它常用于文件夹、组织架构、生物分类、国家地区等等，世间万物的大多数结构都是树形结构。使用树控件可以完整展现其中的层级关系，并具有展开收起选择等交互功能。

本节要实现的 Tree 组件具有以下功能：

- 节点可以无限延伸（递归）；
- 可以展开 / 收起子节点；

- 节点可以选中，选中父节点，它的所有子节点也全部被选中，同样，反选父节点，其所有子节点也取消选择；
- 同一级所有子节点选中时，它的父级也自动选中，一直递归判断到根节点。

API

Tree 是典型的数据驱动型组件，所以节点的配置就是一个 data，里面描述了所有节点的信息，比如图片中的示例数据为：

```
data: [
  {
    title: 'parent 1',
    expand: true,
    children: [
      {
        title: 'parent 1-1',
        expand: true,
        children: [
          {
            title: 'leaf 1-1-1'
          },
          {
            title: 'leaf 1-1-2'
          }
        ]
      },
      {
        title: 'parent 1-2',
        children: [
          {
            title: 'leaf 1-2-1'
          },
          {
            title: 'leaf 1-2-1'
          }
        ]
      }
    ]
  }
]
```

每个节点的配置（props：data）描述如下：

- **title**：节点标题（本例为纯文本输出，可参考 Table 的 Render 或 slot-scope 将其扩展）；
- **expand**：是否展开直子节点。开启后，其直属子节点将展开；
- **checked**：是否选中该节点。开启后，该节点的 Checkbox 将选中；
- **children**：子节点属性数组。

如果一个节点没有 children 字段，那它就是最后一个节点，这也是递归组件终结的判断依据。

同时再提供一个是否显示多选框的 props：showCheckbox，以及两个 events：

- **on-toggle-expand**：展开和收起子列表时触发；
- **on-check-change**：点击复选框时触发。

因为是数据驱动，组件的 API 都比较简单，这一点跟 Table 组件是一样的，它们复杂的逻辑都在组件本身。

入口 tree.vue

在 src/components 中新建目录 tree，并在 tree 下创建两个组件 tree.vue 和 node.vue。tree.vue 是组件的入口，用于接收和处理数据，并将数据传递给 node.vue；node.vue 就是一个递归组件，它构成了每一个**节点**，即一个可展开 / 关闭的按钮（+或-）、一个多选框（使用第 7 节的 Checkbox 组件）、节点标题以及递归的下一级节点。可能现在听起来比较困惑，不要慌，下面逐一分解。

tree.vue 主要负责两件事：

1. 定义了组件的入口，即组件的 API；
2. 对接收的数据 props：data 初步处理，为了在 tree.vue 中不破坏使用者传递的源数据 data，所以会克隆一份数据

（cloneData）。

因为传递的数据 data 是一个复杂的数组结构，克隆它要使用深拷贝，因为浅拷贝数据仍然是引用关系，会破坏源数据。所以在工具集 src/utils/assist.js 中新加一个深拷贝的工具函数 deepCopy：

```
// assist.js, 部分代码省略
function typeOf(obj) {
  const toString = Object.prototype.toString;
  const map = {
    '[object Boolean]'  : 'boolean',
    '[object Number]'   : 'number',
    '[object String]'   : 'string',
    '[object Function]' : 'function',
    '[object Array]'    : 'array',
    '[object Date]'     : 'date',
    '[object RegExp]'   : 'regExp',
    '[object Undefined]': 'undefined',
    '[object Null]'     : 'null',
    '[object Object]'   : 'object'
  };
  return map[toString.call(obj)];
}
// deepCopy
function deepCopy(data) {
  const t = typeOf(data);
  let o;

  if (t === 'array') {
    o = [];
  } else if ( t === 'object') {
    o = {};
```

```
  } else {
    return data;
  }

  if (t === 'array') {
    for (let i = 0; i < data.length; i++) {
      o.push(deepCopy(data[i]));
    }
  } else if ( t === 'object') {
    for (let i in data) {
      o[i] = deepCopy(data[i]);
    }
  }
  return o;
}

export {deepCopy};
```

deepCopy 函数会递归地对数组或对象进行逐一判断，如果某项是数组或对象，再拆分继续判断，而其它类型就直接赋值了，所以深拷贝的数据不会破坏原有的数据（更多深拷贝与浅拷贝的内容，可阅读扩展阅读 1）。

先来看 tree.vue 的代码：

```
<!-- src/components/tree/tree.vue -->
<template>
  <div>
    <tree-node
      v-for="(item, index) in cloneData"
      :key="index"
      :data="item"
      :show-checkbox="showCheckbox"
```

```
    ></tree-node>
  </div>
</template>
<script>
  import TreeNode from './node.vue';
  import { deepCopy } from
'../../utils/assist.js';

  export default {
    name: 'Tree',
    components: { TreeNode },
    props: {
      data: {
        type: Array,
        default () {
          return [];
        }
      },
      showCheckbox: {
        type: Boolean,
        default: false
      }
    },
    data () {
      return {
        cloneData: []
      }
    },
    created () {
      this.rebuildData();
    },
    watch: {
      data () {
```

```
        this.rebuildData();
      }
    },
    methods: {
      rebuildData () {
        this.cloneData = deepCopy(this.data);
      }
    }
  }
</script>
```

在组件 created 时（以及 watch 监听 data 改变时），调用了 rebuildData 方法克隆源数据，并赋值给了 cloneData。

在 template 中，先是渲染了一个 node.vue 组件（<tree-node>），这一级是 Tree 的根节点，因为 cloneDate 是一个数组，所以这个根节点不一定只有一项，有可能是并列的多项。不过这里使用的 node.vue 还没有用到 Vue.js 的递归组件，它只处理第一级根节点。

<tree-node> 组件（node.vue）接收两个 props：

1. showCheckbox：与 tree.vue 的 showCheckbox 相同，只是进行传递；
2. data：node.vue 接收的 data 是一个 Object 而非 Array，因为它只负责渲染当前的一个节点，并递归渲染下一个子节点（即 children），所以这里对 cloneData 进行循环，将每一项节点数据赋给了 tree-node。

递归组件 node.vue

node.vue 是树组件 Tree 的核心，而一个 tree-node 节点包含 4 个部分：

1. 展开与关闭的按钮（+或-）；
2. 多选框；
3. 节点标题；
4. 递归子节点。

先来看 node.vue 的基本结构：

```
<!-- src/components/tree/node.vue -->
<template>
  <ul class="tree-ul">
    <li class="tree-li">
      <span class="tree-expand"
@click="handleExpand">
        <span v-if="data.children &&
data.children.length && !data.expand">+</span>
        <span v-if="data.children &&
data.children.length && data.expand">-</span>
      </span>
      <i-checkbox
        v-if="showCheckbox"
        :value="data.checked"
        @input="handleCheck"
      ></i-checkbox>
      <span>{{ data.title }}</span>
      <tree-node
        v-if="data.expand"
        v-for="(item, index) in data.children"
        :key="index"
        :data="item"
        :show-checkbox="showCheckbox"
      ></tree-node>
    </li>
  </ul>
```

```
</template>
<script>
  import iCheckbox from
'../checkbox/checkbox.vue';

  export default {
    name: 'TreeNode',
    components: { iCheckbox },
    props: {
      data: {
        type: Object,
        default () {
          return {};
        }
      },
      showCheckbox: {
        type: Boolean,
        default: false
      }
    }
  }
</script>
<style>
  .tree-ul, .tree-li{
    list-style: none;
    padding-left: 10px;
  }
  .tree-expand{
    cursor: pointer;
  }
</style>
```

props：data 包含了当前节点的所有信息，比如是否展开子节点（expand）、是否选中（checked）、子节点数据（children）等。

第一部分 expand，如果当前节点不含有子节点，也就是没有 children 字段或 children 的长度是 0，那就说明当前节点已经是最后一级节点，所以不含有展开 / 收起的按钮。

多选框直接使用了第 7 节的 Checkbox 组件（单用模式），这里将 prop: value 和事件 @input 分开绑定，没有使用 v-model 语法糖。value 绑定的数据 data.checked 表示当前节点是否选中，在点击多选框时，handleCheck 方法会修改 data.checked 数据，下文会分析。这里之所以不使用 v-model 而是分开绑定，是因为 @input 里要额外做一些处理，不是单纯的修改数据。

上一节我们说到，一个 Vue.js 递归组件有两个必要条件：name 特性和终结条件。name 已经指定为 TreeNode，而这个终结递归的条件，就是 v-for="(item, index) in data.children"，当 data.children 不存在或为空数组时，自然就不会继续渲染子节点，递归也就停止了。

注意，这里的 v-if="data.expand" 并不是递归组件的终结条件，虽然它看起来像是一个可以为 false 的判断语句，但它的用处是判断当前节点**的子节点**是否展开（渲染），如果当前节点不展开，那它所有的子节点也就不会展开（渲染）。

上面的代码保留了两个方法 handleExpand 与 handleCheck，先来看前者。

点击 + 号时，会展开直属子节点，点击 - 号关闭，这一步只需在 handleExpand 中修改 data 的 expand 数据即可，同时，我们通过 Tree 的根组件（tree.vue）触发一个自定义事件 @on-toggle-expand（上文已介绍）：

```
// node.vue，部分代码省略
import { findComponentUpward } from
'../../utils/assist.js';

export default {
  data () {
    return {
      tree: findComponentUpward(this, 'Tree')
    }
  },
  methods: {
    handleExpand () {
      this.$set(this.data, 'expand',
!this.data.expand);

      if (this.tree) {
        this.tree.emitEvent('on-toggle-expand',
this.data);
      }
    },
  }
}
```

```
// tree.vue，部分代码省略
export default {
  methods: {
    emitEvent (eventName, data) {
      this.$emit(eventName, data,
this.cloneData);
    }
  }
}
```

在 node.vue 中，通过 findComponentUpward 向上找到了 Tree 的实例，并调用它的 emitEvent 方法来触发自定义事件 @on-toggle-expand。之所以使用 findComponentUpward 寻找组件，而不是用 $parent，是因为当前的 node.vue，它的父级不一定就是 tree.vue，因为它是递归组件，父级有可能还是自己。

这里有一点需要注意，修改 data.expand，我们是通过 Vue 的 $set 方法来修改，并没有像下面这样修改：

```
this.data.expand = !this.data.expand;
```

这样有什么区别呢？如果直接用上面这行代码修改，发现数据虽然被修改了，但是视图并没有更新（原来是 + 号，点击后还是 + 号）。要理解这里，我们先看下，到底修改了什么。这里的 this.data，是一个 props，它是通过上一级传递的，这个上一级有两种可能，一种是递归的 node.vue，一种是根组件 tree.vue，但是递归的 node.vue，最终也是由 tree.vue 传递的，追根溯源，要修改的 this.data 事实上是 tree.vue 的 cloneData。cloneData 里的节点数据，是不一定含有 expand 或 checked 字段的，如果不含有，直接通过 this.data.expand 修改，这个 expand 就不是可响应的数据，Vue.js 是无法追踪到它的变化，视图自然不会更新，而 $set 的用法就是对可响应对象中添加一个属性，并确保这个新属性同样是响应式的，且触发视图更新。总结来说，如果 expand 字段一开始是存在的（不管 true 或 false），不管用哪种方式修改都是可以的，否则必须用 $set 修改，结合起来，干脆直接用 $set 了。同理，后文的 checked 也是一样。

接下来是整个 Tree 组件最复杂的一部分，就是处理节点的响应状态。你可能会问，不就是选中或取消选中吗，跟 expand 一样，修改数据就行了？如果只是考虑一个节点，的确这样就可以了，但树组件是有上下级关系的，它们分为两种逻辑，当选中（或取消选中）一个节点时：

1. 它下面的所有子节点都会被选中；

2. 如果同一级所有子节点选中时，它的父级也自动选中，一直递归判断到根节点。

第 1 个逻辑相对简单，当选中一个节点时，只要递归地遍历它下面所属的所有子节点数据，修改所有的 checked 字段即可：

```
// node.vue，部分代码省略
export default {
  methods: {
    handleCheck (checked) {
      this.updateTreeDown(this.data, checked);

      if (this.tree) {
        this.tree.emitEvent('on-check-change',
this.data);
      }
    },
    updateTreeDown (data, checked) {
      this.$set(data, 'checked', checked);

      if (data.children && data.children.length)
{
        data.children.forEach(item => {
          this.updateTreeDown(item, checked);
        });
      }
    }
  }
}
```

updateTreeDown 只是向下修改了所有的数据，因为当前节点的数据里，是包含其所有子节点数据的，通过递归遍历可以轻松修改，这也是第 1 种逻辑简单的原因。

再来看第 2 个逻辑，它的难点在于，无法通过当前节点数据，修改到它的父节点，因为拿不到。写到这里，正常的思路应该是在 this.updateTreeDown(this.data, checked); 的下面再写一个 updateTreeUp 的方法，向上遍历，问题就是，怎样向上遍历，一种常规的思路是，把 updateTreeUp 方法写在 tree.vue 里，并且在 node.vue 的 handleCheck 方法里，通过 this.tree 调用根组件的 updateTreeUp，并且传递当前节点的数据，在 tree.vue 里，要找到当前节点的位置，那还需要一开始在 cloneData 里预先给每个节点设置一个唯一的 key，后面的逻辑读者应该能想到了，就是通过 key 找到节点位置，并向上递归判断……但是，这个方法想着就麻烦。

正常的思路不太好解决，我们就换个思路。一个节点，除了手动选中（或反选），还有就是第 2 种逻辑的被动选中（或反选），也就是说，如果这个节点的所有直属子节点（就是它的第一级子节点）都选中（或反选）时，这个节点就自动被选中（或反选），递归地，可以一级一级响应上去。有了这个思路，我们就可以通过 watch 来监听当前节点的子节点是否都选中，进而修改当前的 checked 字段：

```
// node.vue，部分代码省略
export default {
  watch: {
    'data.children': {
      handler (data) {
        if (data) {
          const checkedAll = !data.some(item =>
!item.checked);
          this.$set(this.data, 'checked',
checkedAll);
        }
      },
      deep: true
    }
  }
}
```

在 watch 中，监听了 data.children 的改变，并且是深度监听的。这段代码的意思是，当 data.children 中的数据的某个字段发生变化时（这里当然是指 checked 字段）,也就是说它的某个子节点被选中（或反选）了，这时执行绑定的句柄 handler 中的逻辑。const checkedAll = !data.some(item => !item.checked); 也是一个巧妙的缩写，checkedAll 最终返回结果就是当前子节点是否都被选中了。

这里非常巧妙地利用了递归的特性，因为 node.vue 是一个递归组件，那每一个组件里都会有 watch 监听 data.children，要知道，当前的节点有两个”身份“，它既是下属节点的父节点，同时也是上级节点的子节点，它作为下属节点的父节点被修改的同时，也会触发上级节点中的 watch 监听函数。**这就是递归**。

以上就是 Tree 组件的所有内容，完整的代码见：https://github.com/icarusion/vue-component-book/tree/master/src/components/tree (https://github.com/icarusion/vue-component-book/tree/master/src/components/tree)

用例：https://github.com/icarusion/vue-component-book/blob/master/src/views/tree.vue (https://github.com/icarusion/vue-component-book/blob/master/src/views/tree.vue)

结语

递归就像人类繁衍一样，蕴藏了无限可能，充满着神奇与智慧。

扩展阅读

- 浅拷贝与深拷贝 (https://juejin.im/post/5b5dcf8351882519790c9a2e)

注：本节部分代码参考 iView (https://github.com/iview/iview/blob/2.0/src/utils/assist.js#