2048 游戏功能实现

一、 设计游戏界面

1. 构建网页游戏布局

编写 index.html 文件，在文件中完成 2048 游戏的页面布局，具体代码如下。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>网页版 2048 游戏</title>
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    <div id="game">
      分数：<span id="game_score">0</span>
      <div id="game_container"></div>
    </div>
    <script src="jquery-3.6.0.min.js"></script>
  </body>
</html>
```

在上述代码中，第 10 行 id 为 game_score 的元素用于显示分数；第 11 行 id 为 game_container 的 元素用于显示数字方块，这些方块将会由 JavaScript 自动生成。

接着在第 13 行代码的下面添加以下代码，利用 JavaScript 封装一个 Game2048 函数。

```
<script src="Game2048.js"></script>
<script>
  Game2048({prefix: 'game', len: 4, size: 100, margin: 20});
</script>
```

上述代码在调用 Game2048()函数时，传递了对象形式的参数。其中，prefix 表示网页中的 id 前缀，用来限制函数内部的代码只对指定 id 前缀的元素进行操作；len 表示棋盘格的单边单元格数量，由于棋盘格是正方形，因此设为 4 就表示 4×4 的单元格布局；size 表示每个单元格的单边长度（像素），设为 100 则单元格大小为 100px×100px；margin 表示单元格间距（像素），设为 20 则每个单元格之间的距离为 20px。

2. 初始化游戏界面

编写 Game2048.js 文件，用于保存与游戏相关的代码，具体代码如下。

```
(function(window, document, $) {
  function Game2048(opt) {
    var prefix = opt.prefix, len = opt.len, size = opt.size, margin = opt.margin;
    var view = new View(prefix, len, size, margin);
  }
  window['Game2048'] = Game2048;
})(window, document, jQuery);
```

上述代码是一个自调用函数，第 7 行在调用函数时传入了 window、document 和 jQuery 参数，表示该函数依赖这些全局变量；第 3 行代码从 opt 对象参数中取出成员，并保存为对应名称的变量；第 4 行代码创建了 view 对象，该对象将用于处理游戏的页面效果。

接下来编写 View 构造函数，在构造函数中设置棋盘背景的宽度和高度，具体代码如下。

```
1  function View(prefix, len, size, margin) {
2    this.prefix = prefix;      // id 或 class 前缀
3    this.len = len;            // 棋盘单边单元格数量（总数量为 len * len）
4    this.size = size;          // 单元格边长，单元格大小为 size×size
5    this.margin = margin;      // 单元格间距
6    this.container = $('#' + prefix + '_container');
7    var containerSize = len * size + margin * (len + 1);
8    this.container.css({width: containerSize , height: containerSize});
9    this.nums = {};            // 保存所有数字单元格对象
10 }
```

在上述代码中，第 6 行用于获取页面中 id 为 game_container 的 元素，然后通过第 7~8 行代码设置该元素的宽和高。其中，第 7 行代码用来计算边长，即通过“len * size”得到单元格总边长，再用“margin * (len + 1)”得到间距的总边长，两者加起来就是棋盘的边长。

3. 自动生成空棋盘格

完成棋盘设置后，下面在棋盘中绘制空单元格形成棋盘格子。首先通过 JavaScript 在 game_container 容器中自动生成如下形式的 game-cell 元素，来表示每个单元格。

```
<div class="game-cell" style="width: 100px; height: 100px; top: 20px;
left: 20px"></div>
```

在生成结果中，width 和 height 表示单元格的宽和高，top 和 left 用于定位单元格的位置。需要注意的是，为了使定位生效，需要将容器 game_container 的 position 样式设为 relative，并将单元格的 position 样式设为 absolute，让单元格相对于容器来定位。

设置定位后，单元格的 top 值就表示距离容器顶部多少像素，left 值表示距离容器左边多少像素。其计算公式为“margin + n * (size + margin)”，n 表示当前单元格前共有多少个单元格，margin 表示间距，size 表示单元格边长。例如，横向第 2 个单元格的 left 值为 20 + 1* (100 + 20) = 140px。

接下来继续编写 View 对象，实现自动生成空棋盘格，具体代码如下。

```
1  View.prototype = {
2    getPos: function(n) {
3      return this.margin + n * (this.size + this.margin);
4    },
5    init: function() {
6      for (var x = 0, len = this.len; x < len; ++x) {
7        for (var y = 0; y < len; ++y) {
8          var $cell = $('<div class="' + this.prefix + '-cell"></div>');
9          $cell.css({
10           width: this.size + 'px', height: this.size + 'px',
11           top: this.getPos(x), left: this.getPos(y)
12         }).appendTo(this.container);
13       }
14     }
15   }
```

```
16 };
```

从上述代码可以看出，init()方法用于根据 len（棋盘单边单元格数量）自动生成空单元格，生成后将会添加到 game_container 容器中。第 10~11 行代码用于指定空单元格的样式，其 top 与 left 通过 getPos()方法进行计算。

为了测试程序是否能够正常运行，在 Game2028 函数中调用 init()方法，如下所示。

```
view.init();
```

通过浏览器访问，运行结果如图 12-1 所示。

图12-1　初始化游戏棋盘

二、 控制游戏数值

1. 创建棋盘数组

创建二维数组，用于保存棋盘中的数值，外层数组表示行，内层数组表示列。在保存时，如果单元格为空，则数组元素对应的值为 0。

为了使代码更好维护，下面将通过 Board 构造函数专门处理单元格中的数值，具体代码如下。

```
1  function Board(len) {
2    this.len = len;
3    this.arr = [];
4  }
5  Board.prototype = {
6    init: function() {
7      for (var arr = [], len = this.len, x = 0; x < len; ++x) {
8        arr[x] = [];
9        for (var y = 0; y < len; ++y) {
10         arr[x][y] = 0;
```

```
11      }
12    }
13    this.arr = arr;
14  }
15 };
```

在上述代码中，init()方法用于根据指定 len 创建二维数组，在初始情况下所有的单元格都是空的，因此第 8 行代码为每个单元格赋值为 0。

接下来在 Game2048 函数中测试程序，具体代码如下。

```
1  var board = new Board(len);
2  board.init();
3  console.log(board.arr);
```

在控制台中查看自动生成的二维数组，如图 12-2 所示。

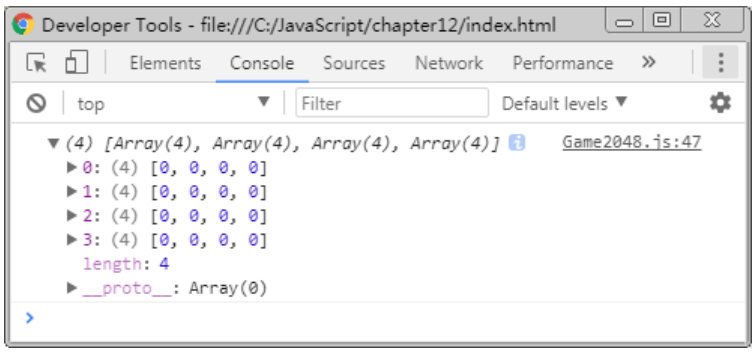

图12-2　创建二维数组

2.　为棋盘生成随机数字单元格

在 2048 游戏开始时，会在棋盘格中的随机位置生成两个随机数字（2 或 4）。下面编写代码，为 Board 原型对象增加方法，用于生成随机数字单元格，具体代码如下。

```
1  // 随机生成数字 2 或 4，保存到数组的随机位置
2  generate: function() {
3    var empty = [];
4    // 查找数组中所有值为 0 的元素索引
5    for (var x = 0, arr = this.arr, len = arr.length; x < len; ++x) {
6      for (var y = 0; y < len; ++y) {
7        if (arr[x][y] === 0) {
8          empty.push({x: x, y: y});
9        }
10     }
11   }
12   if (empty.length < 1) {
13     return false;
14   }
15   var pos = empty[Math.floor((Math.random() * empty.length))];
16   this.arr[pos.x][pos.y] = Math.random() < 0.5 ? 2 : 4;
17   this.onGenerate({x: pos.x, y: pos.y, num: this.arr[pos.x][pos.y]});
18 },
19 // 每当 generate()方法被调用时，执行此方法
```

```
20 onGenerate: function() {},
```

在上述代码中，第 5~11 行用于获取 this.arr 数组中所有空单元格的下标并保存到 empty 数组中，第 15 行代码随机选取 empty 中的一个空单元格，第 16 行代码随机生成一个 2 或 4，第 17 行将数字填入到单元格中。

由于 Board 只用于处理数据，而 View 用于处理页面，为了让两个对象联动，第 17 行通过调用事件方法 this.onGenerate()触发事件，将新创建的单元格在二维数组中的位置和数字内容传递过去。

接下来，在 Game2048 函数中测试程序，生成 2 个随机数。具体代码如下。

```
1  board.onGenerate = function(e) {
2    console.log(e);
3  };
4  board.generate();
5  board.generate();
```

在控制台中输出的结果如图 12-3 所示。

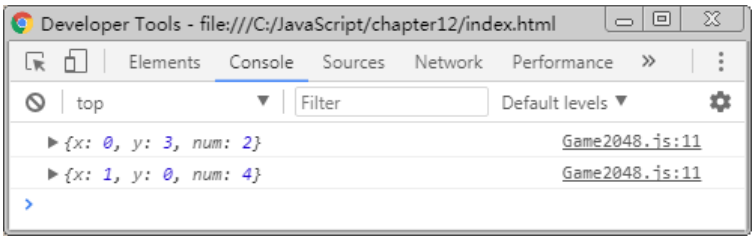

图12-3　生成随机单元格

从图 12-3 中可以看出，新生成的两个单元格的数字分别是 2 和 4，单元格 2 在数组中的位置为[0][3]，单元格 4 在数组中的位置为[1][0]。

3.　在页面中显示数字单元格

为 View 原型对象增加 addNum()方法，根据 x、y 和 num 显示数字单元格，具体代码如下。

```
1  addNum: function(x, y, num) {
2    var $num = $('<div class="' + this.prefix + '-num ' +
3                  this.prefix + '-num-' + num + ' ">');
4    $num.text(num).css({
5      top: this.getPos(x) + parseInt(this.size / 2),      // 用于从中心位置展开
6      left: this.getPos(y) + parseInt(this.size / 2)      // 用于从中心位置展开
7    }).appendTo(this.container).animate({
8      width: this.size + 'px', height: this.size + 'px',
9      lineHeight: this.size + 'px',
10     top: this.getPos(x), left: this.getPos(y)
11   }, 100);
12   this.nums[x + '-' + y] = $num;
13 },
```

在上述代码中，第 2~3 行创建的 元素表示数字单元格，其生成结果示例如下。

```
<div class="game-num game-num-2"></div>
```

在生成结果中，class 为 game-num-2 表示这个单元格按照数字 2 的样式显示。在游戏操作区中，为了明显区分某个数值的单元格，将根据不同的数值设置不同的背景色和文字颜色。CSS 样式示例如下。

```
1  .game-num {width:0px;height:0px;color:#fff;font-size:40px;position:absolute;}
2  .game-num-2 {background:#eee4da;color:#776e65;}
3  .game-num-4 {background:#ede0c8;color:#776e65;}
```

```
4  .game-num-8 {background:#f2b179;}
5  .game-num-16 {background:#f59563;}
6  .game-num-32 {background:#f67c5f;}
7  .game-num-64 {background:#f65e3b;}
8  .game-num-128 {background:#edcf72;font-size:35px;}
9  .game-num-256 {background:#edcc61;font-size:35px;}
10 .game-num-512 {background:#9c0;font-size:35px;}
11 .game-num-1024 {background:#33b5e5;font-size:30px;}
12 .game-num-2048 {background:#09c;font-size:30px;}
```

在上述代码中，第 1 行将 game-num 的宽和高设为 0，用于在数字单元格显示时以“展开”的动画效果出现。为了实现这个效果，通过 addNum()方法的第 5~6 行代码，将单元格的 top 和 left 设置为一个单元格的中心位置，然后在第 8~10 行代码中以动画形式过渡为最终样式。其动画效果如图 12-4 所示。

图12-4　“展开”动画效果

addNum()方法的第 12 行代码用于将新生成的数字单元格保存到 this.nums 中，保存的属性名为单元格在 board.arr 数组中的下标位置。保存后，在进行单元格移动操作时会用到这些对象。

接下来在 Game2048 函数中测试 view.addNum()，具体代码如下。

```
1  board.onGenerate = function(e) {
2    view.addNum(e.x, e.y, e.num);     // 替换原来的“console.log(e);”
3  };
```

通过浏览器访问测试，运行结果如图 12-5 所示。

图12-5　在页面中显示数字单元格

三、 实现单元格移动

1. 单元格左移

2028 游戏支持使用键盘对单元格进行上移、下移、左移、右移操作。下面以左移操作为例，选取棋盘格中的某一行，分析游戏的移动规则，如表 12-1 所示。

表12-1 左移示例

左移前	左移后
0200、0020、0002	2000
2200、0220、0022、2020、0202、2002	4000
0222、2022、2202、2220	4200
2222	4400
0420、0042、0402	4200

从表 12-1 中可以看出，对单元格进行左移后，一行中所有的数字将移动到左边。如果相邻的两个数字（包括中间有空单元格的情况）相等，则进行合并。对于 2222 这种情况，其合并过程为 2222→4022→4202→4400，已经合并过的单元格不会再次合并，因此结果不是 8000。对于 2000、4200 等情况，由于数字已经在左边，且相邻数字无法累加，因此将无法左移。

在分析了左移的规则后，接下来在 Board 原型对象中新增 moveLeft()方法，对二维数组中的数字进行左移操作，具体代码如下。

```
1  moveLeft: function() {
2    var moved = false;    // 是否有单元格被移动
3    // 外层循环从上到下遍历“行”，内层循环从左到右遍历“列”
4    for (var x = 0, len = this.arr.length; x < len; ++x) {
5      for (var y = 0, arr = this.arr[x]; y < len; ++y) {
6        // 从 y + 1 位置开始，向右查找
7        for (var next = y + 1; next < len; ++next) {
8          // 如果 next 单元格是 0，找下一个不是 0 的单元格
9          if (arr[next] === 0) {
10           continue;
11         }
12         // 如果 y 单元格数字是 0，则将 next 移动到 y 位置，然后将 y 减 1 重新查找
13         if (arr[y] === 0) {
14           arr[y] = arr[next];
15           this.onMove({from: {x: x, y: next, num: arr[next]},
16                        to: {x: x, y: y, num: arr[y]}});
17           arr[next] = 0;
18           moved = true;
19           --y;
20         // 如果 y 与 next 单元格数字相等，则将 next 移动并合并给 y
21         } else if (arr[y] === arr[next]) {
22           arr[y] *= 2;
23           this.onMove({from: {x: x, y: next, num: arr[next]},
24                        to: {x: x, y: y, num: arr[y]}});
```

```
25          arr[next] = 0;
26          moved = true;
27        }
28        break;
29      }
30    }
31  }
32  this.onMoveComplete({moved: moved});
33 },
34 onMove: function() {},
35 onMoveComplete: function() {},
```

为了使读者更好的理解上述代码，下面对其实现原理进行分析，具体如下。

① 遍历数组，外层循环从上到下遍历数组行，内层循环从左到右遍历数组列。

② 在遍历到第 1 行第 1 列时，向右依次查找 1 个非 0 单元格，如果找不到，则跳转到第⑤步。

③ 判断第 1 列是否为 0，如果是，将找到的非 0 单元格移动到第 1 列，然后重复第②步。

④ 判断第 1 列与找到的非 0 单元格数字是否相等，如果相等，则将第 1 列数字乘以 2，然后将找到的非 0 单元格数字置为 0，实现左移合并的效果。

⑤ 第 1 列的操作结束，进入第 2 列的操作，类似于第②步。

在代码中，第 15 和 23 行调用了 onMove()方法，该方法表示每次单元格移动时触发的事件，其参数是一个对象，form 保存被移动的单元格的 x、y 位置和数字，to 保存目标单元格的 x、y 位置和数字。

第 32 行调用了 onMoveComplete()方法，该方法表示在整个左移操作完成后触发的事件。其参数是一个对象，moved 表示本次操作是否发生过单元格移动。在 2048 游戏中，如果发生过单元格移动，则会在棋盘中自动增加一个新的随机数字单元格，为了实现这个效果，就需要用到这里的 onMoveComplete()方法和变量 moved 保存的结果。

接下来在 Game2048 函数中测试程序，具体代码如下。

```
1  // 为了测试程序，临时更改 board.arr 的值
2  board.arr = [
3    [0, 0, 0, 2], [0, 2, 0, 2], [2, 2, 2, 2], [0, 2, 4, 0],
4  ];
5  board.moveLeft();
6  console.log(board.arr);
```

在控制台中输出的结果如图 12-6 所示。

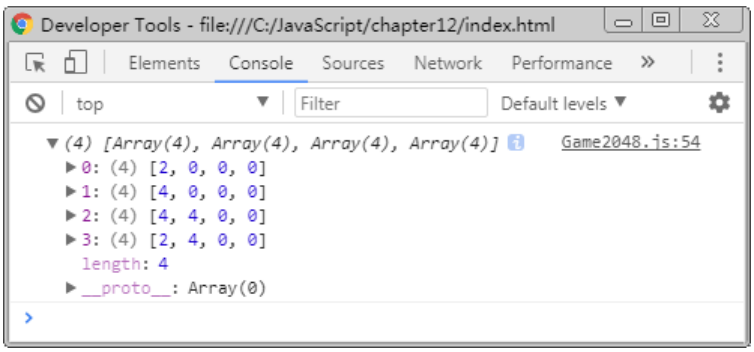

图12-6　在页面中显示数字单元格

从运行结果可以看出，左移操作已经正确移动完成。需要注意的是，测试成功后，应及时删除测试代码，以避免影响后面的开发工作。

由于 2048 游戏的右移、上移、下移操作与左移的实现原理是一样的，这里不再进行代码演示，读者可参考 moveLeft()方法完成 moveRight()、moveUp()、moveDown()方法的编写。最终代码可参考本书的配套源代码。

2. 以动画效果移动单元格

在 board 对象中已经提供了 onMove()和 onMoveComplete()事件方法与页面进行联动，下面在 Game2048 函数中编写以下代码，在单元格移动后执行一些相关操作。

```
1  board.onMove = function(e) {
2    // 每当 border.arr 中的单元格移动时，调用此方法控制页面中的单元格移动
3    view.move(e.from, e.to);
4  };
5  board.onMoveComplete = function(e) {
6    if (e.moved) {
7      // 一次移动操作全部结束后，如果移动成功，则在棋盘中增加一个新单元格
8      setTimeout(function(){ board.generate(); }, 200);
9    }
10 };
```

为 View 原型对象新增 move()方法，对 this.nums 对象中保存的单元格进行处理。具体代码如下。

```
1  move: function(from, to) {
2    var fromIndex = from.x + '-' + from.y, toIndex = to.x + '-' + to.y;
3    var clean = this.nums[toIndex];
4    this.nums[toIndex] = this.nums[fromIndex];
5    delete this.nums[fromIndex];
6    var prefix = this.prefix + '-num-';
7    var pos = {top: this.getPos(to.x), left: this.getPos(to.y)};
8    this.nums[toIndex].finish().animate(pos, 200, function() {
9      if (to.num > from.num) {     // 判断数字是否合并（合并后 to.num 大于 from.num）
10       clean.remove();
11       $(this).text(to.num).removeClass(prefix + from.num).addClass(prefix + to.num);
12     }
13   });
14 },
```

在上述代码中，第 2 行根据参数 from 和 to 对象中保存的 x、y 值，拼接成“x-y”形式的字符串，用于从 this.nums 中获取 fromIndex（被移动对象下标）和 toIndex（目标对象下标）元素。

从 this.nums 中获取到单元格对象后，第 3~13 代码执行了如下操作，实现移动效果。

① 将 this.nums 中的 toIndex 对象替换成 fromIndex 对象。在替换前，先用变量 clean 保存 toIndex 对象。替换后，clean 是目标对象，toIndex 和 FormIndex 是被移动对象。

② 删除 this.nums 中的 FormIndex 属性，此时只有 toIndex 是被移动对象。

③ 为 toIndex 对象设置动画，以 200 毫秒的过渡时间移动到目标对象的位置。

④ 动画执行结束后，判断当前是否为数字单元格合并操作，如果是，将 clean 单元格从页面中删除，并将 toIndex 单元格中的文本更改为新的数字，将 class 更新为新数字对应的样式。

值得一提的是，第 8 行代码在调用 animate()前先调用了 finish()方法结束前一个动画，通过这个操作可以避免用户在使用键盘快速移动时，出现动画效果重叠的问题。

3. 通过键盘移动单元格

在 Game2048 函数中编写如下代码，为 document 添加键盘按下事件。具体代码如下。

```
1  $(document).keydown(function(e) {
2    switch (e.which) {
3      case 37: board.moveLeft();  break;      // 左移
4      case 38: board.moveUp();    break;      // 上移
5      case 39: board.moveRight(); break;      // 右移
6      case 40: board.moveDown();  break;      // 下移
7    }
8  });
```

完成键盘事件后，通过浏览器测试程序，操作键盘的方向键，按上（↑）、下（↓）、左（←）、右（→）控制数字单元格的移动，观察程序运行结果。

四、 设置游戏分数

在游戏开始后，每次两个相同数字进行合并时，要同时更新棋盘区域上方的分数，分数计算方式为合并后的值进行累加。由于 border 对象的 onMove()方法会在单元格移动后自动调用，在 Game2048 函数中，可以自定义 onMove()方法，通过判断其参数来确定是否发生了合并操作，具体代码如下。

```
1  var score = 0;                  // 通过变量保存分数
2  board.onMove = function(e) {
3    if (e.to.num > e.from.num) {
4      score += e.to.num;          // 累加分数
5      view.updateScore(score);    // 更新页面中显示的分数
6    }
7    ……（原有代码）
8  };
```

在上述代码中，第 3 行用于判断 e.to.num（合并后数字）和 e.from.num（被移动数字）的大小，如果 e.to.num 大于 e.from.num，说明发生了合并操作，将合并后的数字累加到分数中即可。

在 View 构造函数中编写如下代码，获取显示分数的元素对象，并将其保存到 this.score 中。

```
1  this.score = $('#' + prefix + '_score');
```

然后在 View 原型对象中增加 updateSocre()方法，更新页面显示的分数。

```
1  updateScore: function(score) {
2    this.score.text(score);
3  },
```

通过浏览器访问测试，当发生合并时，观察分数的变化，如图 12-7 所示。

图12-7　设置游戏分数

五、 判断胜利和失败

1. 判断游戏是否获胜

2048 游戏的胜利条件为，玩家合并出了数字为 2048 的单元格。因此，下面在 Game2048 函数中编写代码，判断合并后单元格的数值是否达到了 2048。如果达到了，说明游戏已经结束，玩家获得了胜利，此时可以弹出一个“您获胜了”的提示信息。具体代码如下。

```
1  var winNum = 2048;              // 胜利条件
2  var isGameOver = false;         // 游戏是否已经结束
3  board.onMove = function(e) {
4    if (e.to.num >= winNum) {
5      isGameOver = true;
6      setTimeout(function() { alert('您获胜了'); }, 300);
7    }
8    ……（原有代码）
9  };
```

在上述代码中，第 6 行用于延迟 300 毫秒后再弹出提示信息，这样可以等待移动单元格的动画结束后再出现提示。具体延迟的时间长短根据实际体验而定即可。

游戏结束后，为了避免键盘还能进行单元格移动，在处理键盘事件前，判断 isGameOver 变量，如果游戏结束，则不再执行移动操作。修改 keydown()事件方法，具体代码如下。

```
1  $(document).keydown(function(e) {
2    if (isGameOver) {
3      return false;
4    }
```

```
5    ……（原有代码）
6  };
```

由于合并出 2048 数字并不简单，为了方便测试，可以先将胜利条件修改为比较小的数字，如 128，观察程序是否能够判断玩家已经获得胜利。

2. 判断游戏是否失败

游戏失败的条件为，数字填满了所有的单元格，并且相邻单元格也无法合并。为了判断这种情况，下面在 border 的原型对象中增加 canMove()方法，表示当前是否还可以继续移动，具体代码如下。

```
1  canMove: function() {
2    for (var x = 0, arr = this.arr, len = arr.length; x < len; ++x) {
3      for (var y = 0; y < len; ++y) {
4        if (arr[x][y] === 0) {
5          return true;
6        }
7        var curr = arr[x][y], right = arr[x][y + 1];
8        var down = arr[x + 1] ? arr[x + 1][y] : null;
9        if (right === curr || down === curr) {
10         return true;
11       }
12     }
13   }
14   return false;
15 },
```

上述代码用于以左上角的单元格为基点，开始遍历。如果当前单元格为 0，表示可以移动；如果相邻的右、下单元格与当前单元格数字相等，表示可以合并；如果遍历完成后仍然没有符合条件的单元格，则说明当前已经无法移动了。

接下来在 Game2048 函数中调用 board.canMove()方法，判断是否已经失败，具体代码如下。

```
1  board.onMoveComplete = function(e) {
2    // 判断是否失败
3    if (!board.canMove()) {
4      isGameOver = true;
5      setTimeout(function() { alert('本次得分：' + score); }, 300);
6    }
7    ……（原有代码）
8  };
```

通过浏览器进行测试，观察程序是否能够判断游戏失败的情况。

3. 完善游戏结束页面

在游戏胜利或失败时，直接弹出警告框的交互体验并不友好，接下来将优化此功能。在 index.html 中，找到 game_container 容器，在容器内部新增如下代码，显示游戏结束画面。

```
1  <div id="game_over" class="game-hide">
2    <div id="game_over_info"></div>
3    <span id="game_restart">重新开始</span>
4  </div>
```

在上述代码中，第 1 行 class 为 game-hide 的元素表示隐藏元素，其样式代码如下。

```
<style>
  .game-hide { display: none; }
</style>
```

在页面中，id 为 game_over_info 的元素用于显示游戏结束时的提示信息，考虑到游戏获胜和失败时显示的信息不同，该元素的内容留空，通过 JavaScript 来更改内容。

在 View 原型对象中增加 win()和 over()方法，分别用于显示获胜和失败的信息，具体代码如下。

```
win: function() {
  $('#' + this.prefix + '_over_info').html('<p>您获胜了</p>');
  $('#' + this.prefix + '_over').removeClass(this.prefix + '-hide');
},
over: function(score) {
  $('#' + this.prefix + '_over_info').html('<p>本次得分</p><p>' + score + '</p>');
  $('#' + this.prefix + '_over').removeClass(this.prefix + '-hide');
},
```

在上述代码中，第 2 行和第 6 行用于在 id 为 game_over_info 元素中添加提示信息，第 3 行和第 7 行用于移除 id 为 game_over 元素的 class 样式 game-hide，移除后提示信息就会显示出来。

在增加了 win()和 over()方法后，在 Game2048 函数中按照如下步骤进行修改。

```
// ① 在 board.onMove 中找到如下代码：
setTimeout(function() { alert('您获胜了'); }, 300);
// 修改为：
setTimeout(function() { view.win(); }, 300);
// ② 在 board.onMoveComplete 中找到如下代码：
setTimeout(function() { alert('本次得分：' + score); }, 300);
// 修改为：
setTimeout(function() { view.over(score); }, 300);
```

通过浏览器访问测试，当游戏结束后，就会看到如图 10-2 所示的效果。

六、 重新开始游戏

当游戏结束后，在页面中提供“重新开始”按钮，单击此按钮可以重新开始游戏。为了实现这个功能，首先在 View 原型对象中增加 cleanNum()方法，用于清空页面中所有的数字单元格，具体代码如下。

```
cleanNum: function() {
  this.nums = {};
  $('#' + this.prefix + '_over').addClass(this.prefix + '-hide');
  $('.' + this.prefix + '-num').remove();
},
```

上述第 2 行代码用于清空 this.nums 中保存的所有数字单元格对象，第 3 行代码用于隐藏游戏结束时的提示信息，第 4 行代码用于移除页面中所有的数字单元格。

然后修改 Game2048 函数中的代码，将开始游戏相关的功能整理到 start()函数中。具体代码如下。

```
function Game2048(opt) {
  var score = 0;            // 初始分数
  var winNum = 2048;        // 获胜条件
  var isGameOver = true;    // 在调用 start()前游戏处于停止状态
  ……
```

```
6    function start() {          // 开始游戏
7      score = 0;                // 将保存的分数重置为 0
8      view.updateScore(0);      // 将页面中的分数重置为 0
9      view.cleanNum();          // 清空页面中多余的数字单元格
10     board.init();             // 初始化单元格数组
11     board.generate();         // 生成第 1 个数字
12     board.generate();         // 生成第 2 个数字
13     isGameOver = false;       // 将游戏状态设为开始
14   }
15   $('#' + prefix + '_restart').click(start);    // 为“重新开始”按钮添加单击事件
16   start();                                      // 开始游戏
17 }
```

在上述代码中，第 6~14 行将初始化游戏相关的代码整理到 start()函数中， 第 15 行代码用于获取 id 为 game_restart 的元素（即“重新开始”按钮），为其添加单击事件，单击时执行 start()函数开始游戏。第 16 行代码调用了 start()函数，用于在页面打开后自动开始游戏。

至此，一个简单的网页版 2048 游戏已经开发完成。