第 1 章

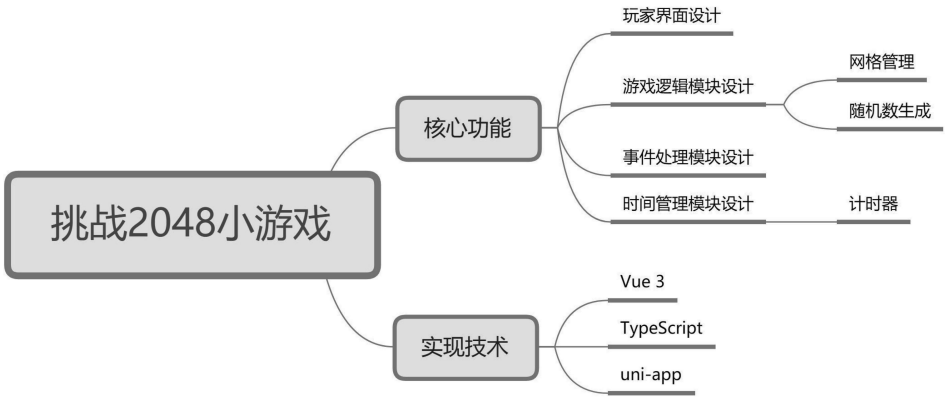
挑战 2048 小游戏

——Vue 3 + TypeScript + uni-app

在移动互联网高速发展的今天，游戏已成为人们日常生活中重要的休闲娱乐方式。2048 小游戏作为一款经典的数字益智游戏，凭借其简单的规则、清晰的逻辑和富有挑战性的玩法，深受广大玩家喜爱。本章将采用 Vue 3、TypeScript 和 uni-app 等关键技术，开发一个响应迅速、界面友好、逻辑清晰，既能提升用户体验，又能为后续功能扩展奠定基础的 2048 小游戏。

项目微视频

本项目的核心功能及实现技术如下。



1.1 开 发 背 景

2048 小游戏是一款经典的数字益智游戏，玩家通过滑动屏幕或者单击方向键，控制棋盘上的数字方块移动。当两个相同数字的方块碰撞时，它们会合并为一个更大的数字（如 2+2=4）。2048 小游戏的目标是通过不断合并数字，最终得到一个数值为“2048”的方块，并挑战更高的分数。2048 小游戏虽然规则简单，但极具策略性，既考验玩家的逻辑思维，又能带来轻松愉悦的游戏体验。

2048 小游戏主要实现以下功能。

- ☑ 简洁美观的界面：采用清晰的棋盘布局和色彩区分，确保玩家能够直观地查看数字和操作反馈。
- ☑ 流畅的游戏逻辑：实现数字方块的移动、合并、得分计算等核心机制，确保游戏运行稳定且响应迅速。
- ☑ 状态记录功能：实时显示游戏用时和当前分数，帮助玩家追踪游戏进度。
- ☑ 游戏结束判定：当棋盘填满且无法继续合并时，自动判定游戏结束，并提供重新开始选项。

1.2 系 统 设 计

1.2.1 开发环境

本项目的开发及运行环境如下。

- ☑ 操作系统：推荐 Windows 10 或 Windows 11 及以上版本。
- ☑ 开发工具：HBuilderX。
- ☑ 开发框架：Vue 3。
- ☑ 开发环境：uni-app。

1.2.2 业务流程

启动游戏后，游戏会在一个空的 4 × 4 网格中随机生成 6 个数字（4 个 2，1 个 4 或 8，1 个 4 或 8），这些数字分布在不同的网格中。玩家可以通过单击屏幕下方的上、下、左、右 4 个方向的按钮对网格中的数字执行移动操作。每执行一次移动操作，所有数字都会按照指定方向移动至所在行或列的末端。在移动的过程中，如果行或列有两个相同数字，那么它们将合并为一个数字。例如，两个 2 合并为一个 4，两个 4 合并为一个 8，以此类推。每次成功合并后，玩家的总分都会增加（得分为合并后数字的一半），例如两个 8 合并为一个 16，则玩家的总分会增加 8。每次执行完移动操作后，都会在两个随机的空白网格中插入随机数字（2、4 或 8）。当空白位置少于两个时，游戏结束。“挑战 2048 小游戏”的业务流程如图 1.1 所示。

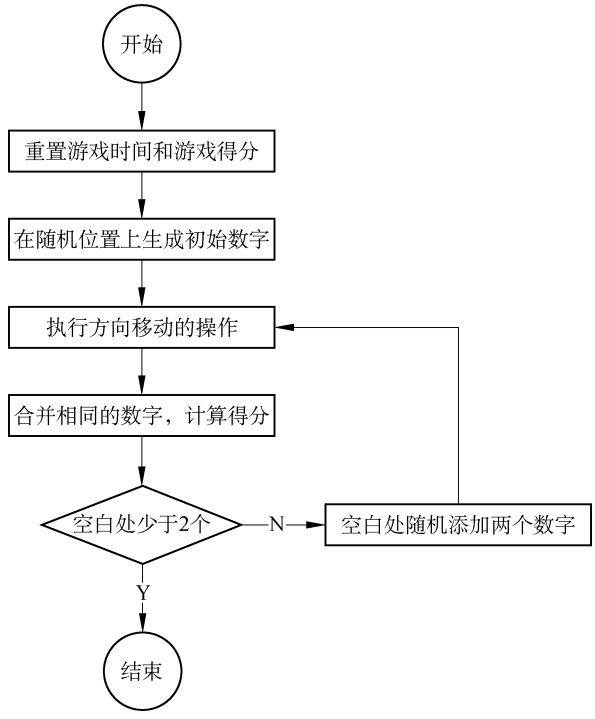


图 1.1 “挑战 2048 小游戏”的业务流程图

1.2.3 功能结构

“挑战 2048 小游戏”通过合理的数据结构和算法设计，实现了一个富有挑战性的数字拼图游戏。其中的各功能模块不仅保证了可维护性，还具备了良好的扩展空间。“挑战 2048 小游戏”的功能模块如下。

- ☑ 玩家界面：包括游戏网格，上、下、左、右 4 个方向的控制按钮，得分显示和计时器等元素的设计。
- ☑ 网格管理：负责生成初始网格、处理网格中数字的移动和合并。
- ☑ 随机数生成：在每次移动操作后，生成两个新的数字（2、4 或 8）填充空白的网格。
- ☑ 计时器：记录玩家的游戏时间，增加游戏的紧张感。
- ☑ 事件处理：跟踪游戏的当前状态，判断游戏是否结束，更新得分和时间，以及处理玩家的移动操作。

1.3 技 术 准 备

1.3.1 技术概览

1. Vue.js

Vue.js 是一个渐进式 JavaScript 框架，采用基于 HTML 的模板语法和响应式数据绑定，通过选项式 API（options API）提供组件化开发能力，其核心包括虚拟 DOM、指令系统和生命周期钩子，适合构建中小型应用。

Vue 3 作为 Vue.js 的升级版本，在保留核心设计理念的同时，引入了基于 Proxy 的响应式系统、Composition API 组合式逻辑复用、Teleport 跨层级渲染等新特性，并优化了性能（如 Tree-Shaking 支持和更高效的虚拟 DOM Diff 算法），更适合大型应用开发。

虽然 Vue 3 是 Vue.js 的升级版本，但学习 Vue.js 仍是必要的，具体原因如下。

（1）基础概念一致：Vue 3 的模板语法、组件通信等核心机制继承自 Vue.js，掌握后者能快速理解前者。

（2）渐进式迁移需求：现有项目多基于 Vue.js，维护或升级需熟悉其逻辑。

（3）设计思想延续：Vue.js 的响应式原理和单向数据流是理解 Vue 3 优化的基础。

二者是继承与发展的关系，而非替代关系。

2. JavaScript

JavaScript 是一种动态类型的脚本语言，作为 Web 开发的基石，直接在浏览器或 Node.js 环境中运行。其核心优势在于灵活性，但动态类型特性可能导致运行时出现难以追踪的类型错误。

TypeScript 是 JavaScript 的超集，通过静态类型系统（如类型注解、接口、泛型）扩展了 JavaScript，在编译阶段进行类型检查，可显著提升代码健壮性和可维护性，同时完全兼容 JavaScript 语法和生态。

虽然 TypeScript 是 JavaScript 的超集，但二者本质上是工具链与语言的关系。

（1）底层依赖：TypeScript 最终编译为 JavaScript，理解 JavaScript 的运行时行为（如原型链、事件循环）是调试和优化的基础。

（2）渐进式采用：现有项目或库可能仍使用纯 JavaScript，直接维护需掌握原生语法。

（3）设计思想：JavaScript 的动态特性（如弱类型、高阶函数）是理解 TypeScript 类型系统的前提。

二者并非替代关系，而是互补协作关系——TypeScript 为 JavaScript 提供开发阶段的“安全护栏”，而 JavaScript 是实际开发中使用的最终语言。

有关 Vue.js 和 JavaScript 的知识，在《Vue.js 从入门到精通》和《JavaScript 从入门到精通（第 5 版）》中有详细的讲解，对这两块知识不太熟悉的读者可以参考书中对应的内容。下面将对 Vue 3、TypeScript 和 uni-app 进行必要介绍，以确保读者可以顺利完成本项目。

1.3.2 Vue 3

虽然 Vue 3 和 Vue 的核心概念相同，但 Vue 3 新增了以下特性。

- ☑ Composition API：通过 setup()函数和响应式函数（ref、reactive）组织逻辑。
- ☑ Teleport：使用<teleport to="body"> 实现跨 DOM 层级渲染。
- ☑ Suspense：统一管理异步组件的加载状态。

☑ 多个 v-model：支持在同一组件上绑定多个 v-model。

在本项目中，Composition API 的代码如下。

```
import { ref } from 'vue'
// 定义游戏状态，初始为未开始
const gameStatus = ref<boolean>(false);
// 定义游戏板，初始为 4 × 4 的矩阵，所有元素为 0
let gameBoard = ref<number[][]>(Array.from({ length: 4 }, () => Array(4).fill(0)));
```

上述代码使用 ref 创建响应式变量，其优势在于本项目中的相关逻辑（如游戏初始化）可以按功能组合，提升代码的组织性。

此外，本项目在处理“方向移动”的逻辑时，还使用了 Vue 3 的“响应式数据处理”，代码如下：

```
const moveAndMerge = (dir) => {
    // 根据方向移动和合并数字
    if (dir == 'shang') {
        // 向上移动并合并
        gameBoard.value = addNum(gameBoard.value);
    }
    // 其他方向处理
}
```

在上述代码中，当修改 gameBoard.value 时，Vue 3 会自动检测变化并更新玩家界面。

此外，Vue 3 在本项目中的核心作用如下。

☑ 逻辑组织：通过 Composition API 集中管理游戏状态（分数、网格、计时器）。

☑ 响应式驱动：数据变化自动更新视图，无须手动操作 DOM。

☑ 性能优势：Proxy 响应式、优化的虚拟 DOM 和 Tree-Shaking 提升运行效率。

说明

DOM 是文档对象模型（Document Object Model）的缩写，它是浏览器将 HTML/XML 文档解析后生成的一个树形结构，允许程序（如 JavaScript）动态访问和操作网页的内容、结构和样式。

1.3.3 TypeScript

TypeScript 是 JavaScript 的超集（JavaScript + 静态类型 + 编译时检查），完全保留了 JavaScript 的语法规则。需要特别说明的是，TypeScript 添加了静态类型系统。TypeScript 的核心特点如下。

☑ 静态类型检查：在编译时捕获类型错误（如变量类型不匹配）。

☑ 类型注解：显式声明变量、函数参数和返回值的类型。

☑ 面向对象增强：支持类、接口、泛型等高级特性。

☑ 工具链支持：提供更好的代码提示和重构能力。

下面列举 TypeScript 在本项目中的一些应用。

1. 类型注解（type annotations）

（1）变量类型定义。

```
// 定义游戏状态，初始为未开始
const gameStatus = ref<boolean>(false);
// 定义游戏板，初始为 4 × 4 的矩阵，所有元素为 0
let gameBoard = ref<number[][]>(Array.from({ length: 4 }, () => Array(4).fill(0)));
```

上述代码确保 gameStatus 的值只能是 true 或 false，gameBoard 只能是数字矩阵。这段代码与 1.3.2 节中的“Composition API 的代码体现”完全相同，这是因为在实际开发中 Vue 3 与 TypeScript 是深度集成的。Vue 3 与 TypeScript 深度集成的优势在于，既能通过编译时类型检查避免运行时错误（如误传非数组参数），又能借助编辑器的智能提示提升开发效率。

（2）函数参数与返回值。

```
const rotate90Clockwise = (matrix: number[][]): number[][] => { ... };
```

上述代码的作用是限制输入和输出的类型必须为 number[][]，避免传递错误的数据类型。

2. 泛型（generics）

TypeScript 中泛型的语法格式如下。

```
const ref<T>(value: T): Ref<T>;          // Vue 的 ref 泛型
```

它在本项目中的应用如下。

```
const total = ref<number>();             // 泛型指定为 number
```

上述代码的作用是明确 total.value 只能是数字类型（如 0、100），不能是字符串或其他类型。

3. 类型推断（type inference）

TypeScript 在本项目中的应用如下：

```
const randomNumbers = [2, 4, 8];         // 推断为 number[]
```

在上述代码中，TypeScript 会根据数组初始值[2, 4, 8]自动推断 randomNumbers 的类型为 number[]。这一类型推断会限制后续操作：例如 randomNumbers[0]只能调用数字类型的方法（如 toFixed()），若尝试调用字符串方法（如 split()）则会在编译时报错。

综上所述，TypeScript 提升了本项目源码的可靠性和可维护性，尤其在复杂状态管理（如向网格添加数字、分数计算）中避免了潜在的类型错误。

1.3.4 uni-app

uni-app 是一个基于 Vue.js 的跨平台开发框架，允许开发者使用一套代码，同时发布到 iOS、Android、Web 及各类小程序（如微信、支付宝）等多个平台。其核心特点如下。

☑ 跨平台能力：通过条件编译和统一 API，适配不同平台的特性差异。

示例：<view>组件在 Web 端编译为 ，在小程序端编译为原生视图组件。

☑ Vue.js 语法支持：完全兼容 Vue 的模板语法、组件化开发和状态管理（如 Vuex）。

☑ 原生性能：通过 Weex 渲染引擎或小程序原生渲染，接近原生应用的性能。

☑ 丰富的插件生态：支持通过插件扩展功能（如地图、支付等）。

下面列举 uni-app 在本项目中的一些应用。

1. 跨平台组件

在本项目中，“跨平台组件”的代码体现如下。

```
<view class="page"> <!-- 替代 HTML 的 div -->
<button @click="gameStart()">游戏开始</button> <!-- 跨平台按钮 -->
</view>
```

在上述代码中，<view>和<button>是 uni-app 提供的跨平台组件，在不同环境中会被编译为对应的原生

组件（如微信小程序的<view>和<button>）。

2. 样式适配

```
.page {
  width: 100vw; /* 视口宽度单位 */
  height: 100vh; /* 视口高度单位 */
}
```

在上述代码中，vw 和 vh 是 uni-app 推荐的响应式单位，可根据不同平台的屏幕尺寸自动适配布局。

此外，在本项目中 uni-app 的核心作用如下。

- ☑ 跨端兼容性：本项目的代码只需编写一次，即可通过 uni-app 编译为微信小程序、H5 等多端应用，无须重写逻辑。
- ☑ 组件标准化：使用<view>、<button>等统一组件，避免直接操作平台特定的 DOM API。
- ☑ 样式统一处理：SCSS 预处理器和 scoped 样式由 uni-app 编译为各平台支持的 CSS 格式。

1.3.5 Vue 3、TypeScript 和 uni-app 的关系

在研发阶段，Vue 3 提供响应式数据处理能力和组件化开发体系。TypeScript 确保代码的类型安全，以减少运行时错误。uni-app 提供跨平台组件和构建流程。在运行阶段，用户操作触发 Vue 3 响应式更新→TypeScript 校验数据→uni-app 渲染平台特定 UI 界面。Vue 3、TypeScript 和 uni-app 的关系详见表 1.1。

表 1.1 Vue 3、TypeScript 和 uni-app 的关系

技　术	核心功能	代码示例	优　势
Vue 3	响应式数据、Composition API	const score = ref<number>(0);	逻辑复用性强，性能优化
TypeScript	静态类型检查、接口泛型	function add(a: number, b: number): number	减少运行时错误，提升可维护性
uni-app	跨平台组件、多端编译	<view>、<button>	一次开发，多端发布

1.4 玩家界面设计

玩家界面分为顶部（top）、中部（center）和底部（bottom）三个部分。其中，顶部显示了游戏的用时和得分；中部展示了 4 × 4 的游戏板，每个网格根据数值显示不同的背景色和数字；底部提供了“游戏开始”按钮、“结束”按钮和上、下、左、右 4 个方向的控制按钮，如图 1.2 和图 1.3 所示。

1.4.1 顶部状态栏设计

玩家界面的顶部用于显示游戏的用时和得分，效果如图 1.4 所示。

在实际开发中，使用<view>表示“挑战 2048 小游戏”的玩家界面的顶部。在<view>中需包含两个子<view>：一个子<view>的 class 名为 time，显示的内容是“用时:”；另一个子<view>的 class 名为 score，显示的内容是“得分:”，代码如下。

```
<!-- 顶部信息显示区域 -->
<view class="top">
    <!-- 显示游戏用时 -->
    <view class="time">
```

```
            用时:{{allTime}}s
        </view>
        <!-- 显示游戏得分 -->
        <view class="score">
            得分:{{total}}
        </view>
</view>
```

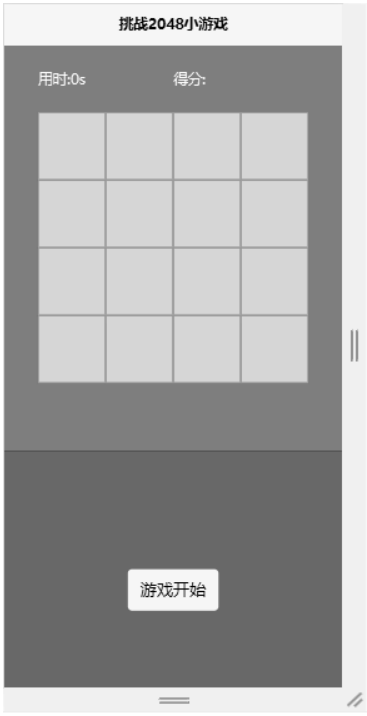


图 1.2　游戏未开始的效果图

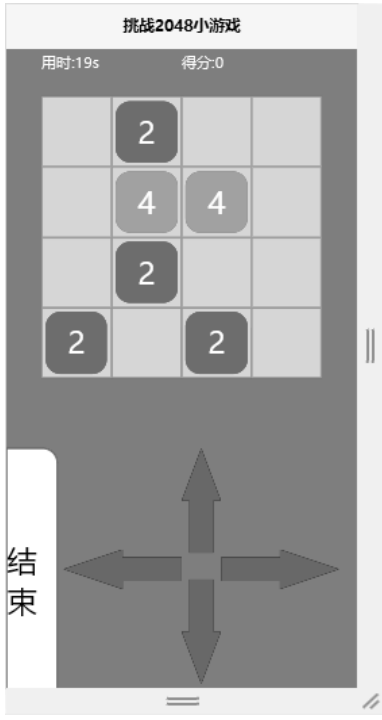


图 1.3　游戏进行中的效果图

图 1.4　顶部状态栏

在名为 time 的子<view>的后面，绑定了一个动态数据，即{{allTime}}，表示游戏的“用时”，其后的 s 表示“秒”。在名为 score 的子<view>的后面，也绑定了一个动态数据，即{{total}}，它表示游戏的“得分”。

1.4.2　中部游戏板设计

“挑战 2048 小游戏”的玩家界面的中部被称作游戏板（即游戏面板），其实现效果如图 1.5 所示。游戏板是“挑战 2048 小游戏”的核心区域，其中包含动态生成的游戏网格，代码如下。

```
<!-- 中心游戏板区域 -->
<view class="center">
    <!-- 主游戏板容器 -->
    <view class="mainBox">
        <!-- 遍历游戏板的每一行 -->
        <view class="row" v-for="(row, rowIndex) in gameBoard" :key="rowIndex">
            <!-- 遍历每一行中的每一个单元格 -->
            <view class="cell" v-for="(cell, cellIndex) in row" :key="cellIndex">
                <!-- 根据条件动态设置单元格的样式 -->
                <view
                    :class="cellIndex==newArr[0][1]&&rowIndex==newArr[0][0]||cellIndex==newArr[1][1]
                    &&rowIndex==newArr[1][0]?'newBox':cell!==0?'cellBox':''">
                    <!-- 根据单元格的值设置背景颜色和文本内容 -->
                    <view class="colorBox"
```

```
                                :style="{backgroundColor:cell==2?'#ff3a3a':cell==4?'#ff9b29':cell==8?'#ebff31':
                                cell==16?'#34ff31':cell==32?'#369083':cell==64?'#2e3cff':cell==128?'#c12fff':
                                cell==256?'#ff77ed':cell==512?'#ffe9fe':cell==1024?'#fffcd4':cell==2048?'#04010b':''}">
                                <text v-show=" cell!==0&&cell!==1">{{ cell }}</text>
                            </view>
                        </view>
                    </view>
                </view>
            </view>
        </view>
```

<view class="center">指的是游戏板的所在区域，<view class="mainBox">包含了动态生成的游戏网格。在动态生成游戏网格的过程中，使用 v-for 动态生成游戏网格的行和列，游戏网格中的数字不同则游戏网格的背景色也不同。

1.4.3 底部控制区设计

“挑战 2048 小游戏”玩家界面的底部控制区会根据游戏状态显示不同的操作信息。如果游戏未开始，那么操作界面只显示一个“游戏开始”按钮，如图 1.6 所示。如果游戏正在进行中，那么操作界面会显示“结束”按钮和上、下、左、右 4 个方向的控制按钮，如图 1.7 所示。

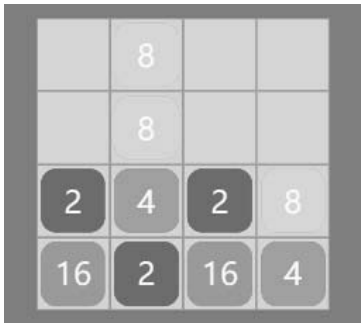


图 1.5 中部游戏板

图 1.6 游戏未开始时的底部控制区

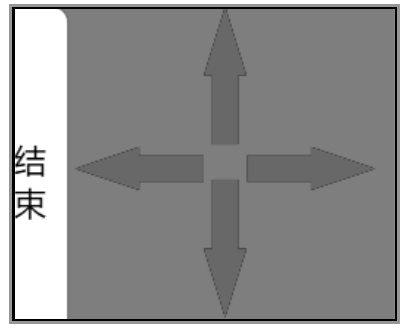


图 1.7 游戏进行中的底部控制区

底部控制区的代码如下。

```
<!-- 底部控制按钮区域 -->
<view class="bottom">
    <!-- 游戏未开始时的开始按钮 -->
    <view class="kaishi" v-show="gameStatus==false">
        <view class="flexBox"><button @click="gameStart()">游戏开始</button></view>
    </view>
    <!-- 游戏进行中的控制按钮 -->
    <view class="jinxing" v-show="gameStatus==true">
        <view class="flexBox">
            <!-- 结束游戏按钮 -->
            <view class="gameOver">
                <view class="gameOverButton" @click="gameOver()">
                    结束
                </view>
            </view>
            <!-- 方向控制按钮 -->
            <view class="contorl">
                <view class="shang" @click="shang()">
                </view>
                <view class="xia" @click="xia()">
                </view>
```

```
                        <view class="zuo" @click="zuo()">
                        </view>
                        <view class="you" @click="you()">
                        </view>
                    </view>
                </view>
            </view>
        </view>
```

说明

“挑战 2048 小游戏”通过上、下、左、右 4 个方向的控制按钮实现游戏逻辑，因此每个按钮都需要通过@click 绑定对应方向的操作方法。

1.5 玩家界面样式设计

在“挑战 2048 小游戏”中，除了 1.4 节已经介绍的 Vue 组件，还需要定义 Vue 组件中的样式部分，其中包括顶部区域样式、中部区域样式和底部区域样式。下面将依次介绍上述各个样式的实现逻辑。

1.5.1 顶部区域样式

定义一个名为.top 的容器及其内部元素（.score 和.time）的样式，主要用于展示分数和时间信息。容器.top 的样式说明如表 1.2 所示。

表 1.2 容器.top 的样式说明

属　性	值	作　用
width	80%	宽度为父容器（或视口）的 80%
height	20vw	高度为视口宽度的 20%（响应式高度）
display	flex	启用 Flexbox 布局
align-items	center	子元素垂直居中对齐
margin-left	10%	左外边距为父容器宽度的 10%（水平居中效果）
font-size	1rem	基础字体大小为根元素（<html>）的字体大小

通过定义容器.top 的样式，将呈现出如下的布局效果。

☑ 容器宽度为 80%，左侧偏移 10%，实际居中显示（10% + 80% + 10% = 100%）。

☑ 高度基于视口宽度（vw），保持响应式。

☑ 内部子元素（.score 和.time）通过 Flexbox 布局并排显示，且垂直居中。

子元素.score 和.time 的样式说明如表 1.3 所示。

表 1.3 子元素.score 和.time 的样式说明

属　性	值	作　用
flex	1	子元素均分剩余空间（Flexbox 弹性布局）
color	white	文字颜色为白色

在表 1.3 中有两个关键点，具体如下。

- ☑ flex 属性：该属性值为 1 时，表示子元素会等比扩展，以填充.top 容器的剩余宽度。如果只有一个子元素设置了 flex: 1，则该元素会占据全部剩余空间；这里两个子元素均设置了此属性，因此它们各占 50%。
- ☑ color 属性：文字为白色，通常需要确保背景（如.page 的绿色背景）色与白色文字之间有足够的对比度。

顶部区域样式代码如下。

```
.top {
    width: 80%;                          /* 设置宽度为视口宽度的 80% */
    height: 20%vw;                       /* 设置高度为视口高度的 20% */
    display: flex;                       /* 设置为弹性布局 */
    align-items: center;                 /* 垂直居中对齐 */
    margin-left: 10%;                    /* 设置左边距为视口的 10% */
    font-size: 1rem;                     /* 设置字体大小为 1rem */

    /* 分数区域样式 */
    .score {
        flex: 1;                         /* 设置分数区域的 flex 属性为 1，使其占据剩余空间 */
        color:white;                     /* 设置分数区域的文字颜色为白色 */
    }

    /* 时间区域样式 */
    .time {
        flex: 1;                         /* 设置时间区域的 flex 属性为 1，使其占据剩余空间 */
        color:white;                     /* 设置时间区域的文字颜色为白色 */
    }
}
```

1.5.2 中部区域样式

定义一个名为.center 的区域，主要用于构建一个网格布局系统（类似游戏棋盘或卡片容器），包含弹性布局、单元格样式和动画效果。.center 区域的布局结构如下。

```
.center (100vw × 100vw)
  └── .mainBox (80%宽度，居中)
      └── .row (Flex 垂直布局)
          └── .cell (Flex 弹性单元格)
              ├── .newBox (带动画的内容)
              ├── .cellBox (静态内容)
              └── .colorBox (填充容器)
```

.center 区域通过多层 display: flex 实现灵活的网格系统，使用百分比和 vw 单位适配不同屏幕，其中的内容区域（.newBox 或.cellBox）通过缩进和圆角增强立体感，.newBox 则通过缩放动画实现从 0 到 90% 的“弹出”效果。下面具体说明.center 区域布局结构中的各个内容。

容器.center 的样式代码如下。

```
.center {
    width: 100vw;                        /* 设置宽度为视口宽度 */
    height: 100vw;                       /* 设置高度与视口宽度相同 */
    /* 省略其他内容样式的代码 */
```

主容器.mainBox 的样式代码如下。

```
.mainBox {
    width: 80%;                          /* 设置宽度为视口宽度的 80% */
    margin: 0% 10%;                      /* 设置左右边距为视口的 10%，实现水平居中*/
```

```
    height: 80%;                    /* 设置高度为视口高度的 80% */
    border-radius: 15px;            /* 设置边框圆角的半径为 15px */
    display: flex;                  /* 设置为弹性布局 */
    /* 省略其他内容样式的代码  */
```

通过定义主容器.mainBox 的样式，将呈现出“一个居中显示的正方形区域，占视口宽度的 80%，高度自适应”的布局效果。

行.row 的样式代码如下。

```
.row {
    flex: 1;                        /* 占据剩余空间 */
    display: flex;                  /* 使用弹性布局 */
    flex-direction: column;         /* 设置为垂直布局垂直排列子元素 */
}
```

通过定义行.row 的样式，构建垂直方向的弹性布局，通常用于多行网格。

单元格.cell 的样式代码如下。

```
.cell {
    flex: 1;                        /* 设置单元格的弹性布局（占据剩余空间） */
    border: 1px solid #ff80c2;      /* 设置单元格的边框样式 */
    background-color: #b5f2ff;      /* 设置单元格的背景颜色 */
    display: flex;                  /* 使用弹性布局 */
    justify-content: center;        /* 设置单元格的内容水平居中 */
    align-items: center;            /* 设置单元格的内容垂直居中 */
    color: #ffffff;                 /* 设置单元格内容的字体颜色 */
    font-size: 2rem;                /* 设置单元格内容的字体大小 */
    /* 省略其他内容样式的代码 */
```

通过定义单元格.cell 的样式，构建网格中的单个单元格，用于显示内容（如数字）。

子元素.newBox（带动画的内容）的样式代码如下。

```
.newBox {
    width: 90%;                     /* 设置宽度为父元素的 90% */
    height: 90%;                    /* 设置高度为父元素的 90% */
    background-color: #9d6fff;      /* 设置背景颜色 */
    border-radius: 15px;            /* 设置边框圆角的半径 */
    display: flex;                  /* 使用弹性布局 */
    justify-content: center;        /* 水平居中 */
    align-items: center;            /* 垂直居中 */
    animation: newBox 0.5s;         /* 设置动画为 newBox，持续时间为 0.5s */
}
```

通过定义子元素.newBox 的样式，可构建新增元素时的动画效果（如游戏中随机数字的出现）。

子元素.cellBox（静态内容）的样式代码如下。

```
.cellBox {
    width: 90%;                     /* 设置宽度为父元素的 90% */
    height: 90%;                    /* 设置高度为父元素的 90% */
    background-color: #9d6fff;      /* 设置背景颜色 */
    border-radius: 15px;            /* 设置边框圆角的半径 */
}
```

通过定义子元素.cellBox 的样式，可构建普通单元格内容的样式（无动画效果）。

子元素.colorBox（颜色填充容器）的样式代码如下。

```
.colorBox {
    width: 100%;                    /* 设置颜色盒子的宽度为父元素的宽度 */
    border-radius: 15px;            /* 设置颜色盒子边框的圆角半径为 15px */
    height: 100%;                   /* 设置颜色盒子的高度为父元素的高度 */
    display: flex;                  /* 使用弹性布局 */
```

```
    justify-content: center;                /* 设置颜色盒子水平居中 */
    align-items: center;                    /* 设置颜色盒子垂直居中 */
}
```

1.5.3 底部区域样式

定义一个名为.bottom 的区域，其中包含了两种状态：一种是在游戏未开始时显示开始按钮（即.kaishi）；另一种是在游戏进行中显示方向按钮和结束按钮（即.jinxing）。.bottom 区域的布局结构如下。

```
.bottom（Flex 容器）
├── .kaishi（全屏开始按钮）
│   └── .flexBox（居中内容）
└── .jinxing（控制面板，默认隐藏）
    └── .flexBox（水平布局）
        ├── .contorl（方向按钮容器）
        │   ├── .shang（上）
        │   ├── .xia（下）
        │   ├── .zuo（左）
        │   └── .you（右）
        └── .gameOver（结束按钮）
```

在.bottom 区域中，.kaishi 和.jinxing 这两种状态均通过 position: absolute 切换显示。在游戏进行中的状态（即.jinxing）下，.bottom 区域通过所有按钮悬停显示蓝色边框，从而增强可操作性；为了节省空间，方向按钮通过 clip-path 以避免使用图片。下面具体说明一下.bottom 区域的布局结构中的各项内容。

容器.bottom 的样式代码如下。

```
.bottom {
    flex: 1;                                /* 设置 flex 属性，使其占据剩余空间 */
    position: relative;                     /* 设置定位属性，使其相对于父元素定位 */
    /* 省略其他内容样式的代码 */
```

.kaishi 的样式的代码如下。

```
.kaishi {
    width: 100%;                            /* 设置宽度为父元素的宽度 */
    height: 100%;                           /* 设置高度为父元素的高度 */
    background-color: #ff0000;              /* 设置背景颜色为红色 */
    position: absolute;                     /* 设置定位为绝对定位 */
    /* 省略其他内容样式的代码 */
```

在.kaishi 中，有一个子元素.flexBox，.flexBox 的样式代码如下。

```
.flexBox {
    width: inherit;                         /* 设置宽度继承父元素的宽度 */
    height: inherit;                        /* 设置高度继承父元素的高度 */
    display: flex;                          /* 使用弹性布局 */
    justify-content: center;                /* 设置水平居中 */
    align-items: center;                    /* 设置垂直居中 */
}
```

.jinxing 的样式代码如下。

```
.jinxing {
    width: 100%;                            /* 设置宽度为父元素的宽度 */
    height: 100%;                           /* 设置高度为父元素的高度 */
    position: absolute;                     /* 设置定位为绝对定位 */
    /* 省略其他内容样式的代码 */
```

在.jinxing 中，也有一个子元素.flexBox，.flexBox 的样式代码如下。

```
.flexBox {
    width: inherit;                    /* 设置宽度继承父元素的宽度 */
    height: inherit;                   /* 设置高度继承父元素的高度 */
    display: flex;                     /* 使用弹性布局 */
    flex-direction: row;               /* 设置弹性布局方向为水平排列 */
    /* 省略其他内容样式的代码 */
```

说明

上述代码中的“flex-direction: row;”表示水平排列控制按钮。

在定义方向按钮.contorl 时，通过绝对定位和 clip-path 绘制四个方向箭头，代码如下。

```
.contorl {
    flex: 1;                           /* 设置 flex 属性，使其在父容器中占据剩余空间 */

    .shang {
        width: 40px;                   /* 设置宽度为 40px */
        height: 40%;                   /* 设置高度为父元素高度的 40% */
        position: absolute;            /* 设置绝对定位 */
        left: 50%;                     /* 设置左边距为 50% */
        background-color: #ff0777;     /* 设置背景颜色为#ff0777 */
        /* 设置裁剪路径为多边形 */
        clip-path: polygon(0% 50%, 50% 0%, 100% 50%, 80% 50%, 80% 100%, 20% 100%, 20% 50%);
    }

    .shang:hover {
        border: 1px solid #3d37ff;     /* 当鼠标悬停在.shang 元素上时，添加 1 像素的实线边框，颜色为#3d37ff */
    }

    .xia {
        width: 40px;                   /* 设置宽度为 40px */
        height: 40%;                   /* 设置高度为父元素高度的 40% */
        position: absolute;            /* 设置绝对定位 */
        top: 50%;                      /* 设置顶部距离为 50% */
        left: 50%;                     /* 设置左侧距离为 50% */
        background-color: #ff0777;     /* 设置背景颜色为#ff0777 */
        /* 设置裁剪路径为多边形 */
        clip-path: polygon(20% 0%, 80% 0%, 80% 50%, 100% 50%, 50% 100%, 0% 50%, 20% 50%);
    }

    .xia:hover {
        border: 1px solid #3d37ff;     /* 当鼠标悬停在.xia 元素上时，添加 1 像素的实线边框，颜色为#3d37ff */
    }

    /* 左侧按钮样式 */
    .zuo {
        width: 120px;                  /* 设置宽度为 120px */
        height: 40px;                  /* 设置高度为 40px */
        position: absolute;            /* 设置绝对定位 */
        top: calc(50% - 30px);         /* 设置顶部位置，计算 50%减去 30px */
        left: calc(50% - 120px);       /* 设置左侧位置，计算 50%减去 120px */
        background-color: #ff0777;     /* 设置背景颜色为#ff0777 */
        /* 设置裁剪路径为一个多边形 */
        clip-path: polygon(0% 50%, 50% 0%, 50% 20%, 100% 20%, 100% 80%, 50% 80%, 50% 100%);
    }

    .zuo:hover {
        border: 1px solid #3d37ff;     /* 当鼠标悬停在.zuo 类上时，设置边框为 1 像素的蓝色实线 */
    }
    /* 右箭头按钮样式 */
```

```
    .you {
        width: 120px;                    /* 设置宽度为 120px */
        height: 40px;                    /* 设置高度为 40px */
        position: absolute;              /* 设置绝对定位 */
        top: calc(50% - 30px);           /* 设置顶部位置为 50%减去 30px */
        left: calc(50% + 40px);          /* 设置左侧位置为 50%加上 40px */
        background-color: #ff0777;       /* 设置背景颜色为#ff0777 */
        /* 设置裁剪路径为多边形 */
        clip-path: polygon(0% 20%, 50% 20%, 50% 0%, 100% 50%, 50% 100%, 50% 80%, 0% 80%);
    }

    .you:hover {
        border: 1px solid #3d37ff;       /* 当鼠标悬停在.you 元素上时，添加 1 像素的实线边框，颜色为#3d37ff */
    }
}
```

通过定义方向按钮.contorl 的样式，使得上、下、左、右这 4 个按钮都把背景色设置为#ff0777（玫瑰红色），并采用绝对定位（通过 left/top 和 calc 计算居中位置）。这些按钮都通过 clip-path 使用多边形裁剪出箭头形状，并在悬停时显示蓝色边框。

左侧的结束按钮.gameOver 的样式代码如下。

```
.gameOver {
    /* 游戏结束按钮样式 */
    .gameOverButton {
        width: 50px;                     /* 设置宽度为 50px */
        height: 100%;                    /* 设置高度为父元素高度 */
        font-size: 2rem;                 /* 设置字体大小 */
        display: flex;                   /* 使用弹性布局 */
        justify-content: center;         /* 设置水平居中 */
        align-items: center;             /* 设置垂直居中 */
        background-color: #fff;          /* 设置背景颜色 */
        border-radius: 0 15px 0 0;       /* 设置边框的圆角半径 */
        border: 1px solid #a860ff;       /* 设置边框样式 */
    }
}
```

1.6 游戏逻辑模块设计

在“挑战 2048 小游戏”的游戏逻辑模块设计中，包含了两个功能模块，即网格管理和随机数生成。为了完成游戏逻辑模块设计，需要执行以下 5 个步骤：初始化矩阵、旋转矩阵、向上合并数字、添加随机数字和方向移动。下面将依次介绍这 5 个步骤的实现过程。

图 1.8 初始化矩阵

1.6.1 初始化矩阵

设计游戏逻辑模块的第 1 步是定义一个 numInit()函数，该函数用于初始化一个 4×4 的数字矩阵。在 numInit()函数中，首先创建一个 4×4 的二维数组（矩阵），数组中所有元素的初始值都为 0；然后随机选择 6 个不同的位置，前 4 个位置填充数字 2，后两个位置随机填充 4 或 8，如图 1.8 所示；最后该函数将返回初始化的矩阵。

初始化矩阵的实现代码如下。

```
const numInit = () => {
    // 创建一个 4 × 4 的矩阵，所有元素为 0
    const array = Array.from({ length: 4 }, () => Array(4).fill(0));
    // 存储所有可能的位置坐标
    const positions = [];
    for (let i = 0; i < 4; i++) {
        for (let j = 0; j < 4; j++) {
            positions.push({ x: i, y: j });
        }
    }
    // 存储被选中的位置坐标
    const selectedPositions = [];
    for (let i = 0; i < 6; i++) {
        // 随机选择一个位置坐标并从 positions 中移除
        const randomIndex = getRandomInt(0, positions.length - 1);
        selectedPositions.push(positions[randomIndex]);
        positions.splice(randomIndex, 1);
    }
    for (let i = 0; i < 4; i++) {
        // 在被选中的前 4 个位置放置数字 2
        const position = selectedPositions[i];
        array[position.x][position.y] = 2;
    }
    for (let i = 4; i < 6; i++) {
        // 在被选中的后两个位置随机放置数字 4 或 8
        const position = selectedPositions[i];
        const randomValue = getRandomInt(1, 2) === 1 ? 4 : 8;
        array[position.x][position.y] = randomValue;
    }
    return array;
}
```

上述代码有 5 个关键点，具体如下。

- ☑ 初始化一个 4 × 4 的全零矩阵。
- ☑ 将矩阵所有元素的坐标（共 16 个）存入 positions 数组，格式为“{x: 行索引, y: 列索引}”。
- ☑ 通过 getRandomInt()从 positions 中随机选取一个坐标，移出并存入 selectedPositions 数组中，重复 6 次，从而随机选择 6 个位置，通过 splice 确保 6 个位置不重复。
- ☑ 将前 4 个选中的位置的值设置为 2。
- ☑ 后两个位置随机选择 4 或 8，概率各 50%。

1.6.2 旋转矩阵

设计游戏逻辑模块的第 2 步是定义一个 rotate90Clockwise()函数，该函数用于将一个初始化后的矩阵顺时针旋转 90°。矩阵旋转 90°的核心逻辑是矩阵的转置加列反转，原矩阵中位置是“(i, j)”的元素先经转置后会移动到“(j, i)”的位置，再经列反转后会移动到“(j, n − i − 1)”的位置。代码如下：

```
const rotate90Clockwise = (matrix) => {
    const n = matrix.length;                                    // 获取矩阵的长度（假设是方阵）
    let rotatedMatrix = Array.from({ length: n }, () => []);    // 初始化一个空的旋转后的矩阵，长度与原矩阵相同

    // 遍历原矩阵的各个元素
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            rotatedMatrix[j][n - i - 1] = matrix[i][j];         // 将原矩阵的元素放到旋转后的新位置上
        }
    }
```

```
    return rotatedMatrix;                                    // 返回旋转后的矩阵
}
```

说明

“(i, j)”“(j, i)”和“(j, n − i − 1)”对应的格式是“(行索引, 列索引)”。

上述代码有 3 个关键点，具体如下。

☑ 通过 matrix.length 获取原矩阵的长度。

☑ 创建一个与原矩阵具有相同长度的空矩阵。

☑ 将原矩阵顺时针旋转 90°，原矩阵中位置是“(i, j)”的元素会成为位置是“(j, n − i − 1)”的元素。

1.6.3 向上合并数字

设计游戏逻辑模块的第 3 步是定义一个 addNum()函数，该函数用于实现“挑战 2048 小游戏”中向上合并数字的逻辑。具体的实现步骤包括：深复制输入数组、初始化表示“得分”的变量、遍历矩阵的每一行、遍历矩阵中当前行的每个元素、向上合并相同数字、处理非零元素、更新总得分并返回处理后的矩阵。代码如下。

```
const addNum = (arr) => {
    let copiedArray = JSON.parse(JSON.stringify(arr));    // 深复制传入的数组，避免修改原始数据
    let defen = 0;                                        // 初始化分数变量为 0

    for (let i = 0; i < copiedArray.length; i++) {        // 遍历每一行
        for (let j = 0; j < copiedArray[i].length; j++) { // 遍历当前行的每个元素
            if (copiedArray[i][j] !== 0) {                // 如果当前元素不为 0
                // 从当前元素的上边开始检查并合并相同元素
                for (let p = 0; p < j; p++) {
                    if (copiedArray[i][p] == copiedArray[i][j]) {
                        // 合并相同元素，并计算得分
                        copiedArray[i][p] = copiedArray[i][j] + copiedArray[i][p];
                        defen = defen + copiedArray[i][p] / 2;
                        copiedArray[i][j] = 0;            // 将当前元素置为 0，表示已合并
                        break;
                    }
                    // 如果找到第一个 0 的位置，就将当前元素移到该位置
                    if (copiedArray[i][p] == 0) {
                        copiedArray[i][p] = copiedArray[i][j];
                        copiedArray[i][j] = 0;
                        break;
                    }
                }
            }
        }
    }
    total.value = total.value + defen;                    // 更新总得分
    return copiedArray;                                   // 返回处理后的矩阵
}
```

上述代码有两个关键点，具体如下。

☑ 合并数字的逻辑：先向上检查、合并相同的数字，再将非零元素向上移动（如[2, 0, 2] → [0, 0, 4]），如图 1.9 所示。

☑ 得分统计：表示为合并后的数字的一半，例如两个 2 合并为一个 4，则玩家的总分会增加 2，如图 1.9 所示。

1.6.4 添加随机数字

设计游戏逻辑模块的第 4 步是定义一个 addRandomNumbersToZeros()函数，该函数用于在矩阵中的两个随机空白位置（元素为 0 的位置）插入随机数字（2、4 或 8）。该函数首先收集所有空白位置的行索引和列索引，然后随机选择两个空白位置，接着向其中插入随机数字，如图 1.10 所示，最后更新全局变量 newArr.value 以记录插入随机数字的位置，并返回添加随机数字后的矩阵。

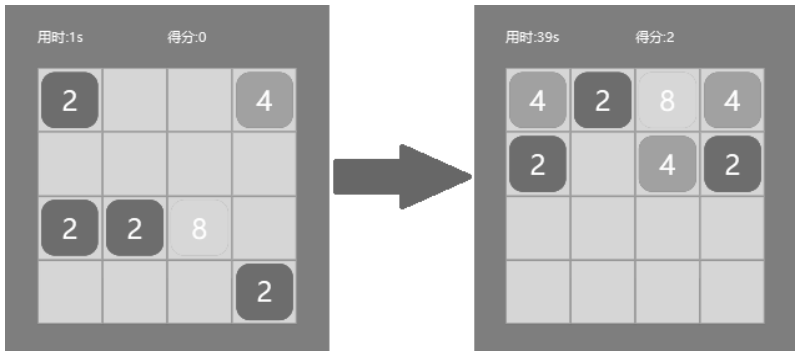


图 1.9 向上合并数字

图 1.10 添加随机数字

添加随机数字的实现代码如下。

```
const addRandomNumbersToZeros = (arr) => {
    let matrix = JSON.parse(JSON.stringify(arr));          // 创建矩阵的深复制以避免修改原数组

    let zeroIndices = [];                                  // 存储所有 0 的位置索引
    for (let i = 0; i < matrix.length; i++) {
        for (let j = 0; j < matrix[i].length; j++) {
            // 如果当前元素为 0，记录其索引
            if (matrix[i][j] === 0) {
                zeroIndices.push([i, j]);
            }
        }
    }

    // 如果 0 的数量少于两个，游戏结束
    if (zeroIndices.length < 2) {
        gameOver();
        return;
    }

    // 随机选择两个 0 的位置并生成一个包含这些位置的数组
    let randomIndices = zeroIndices.sort(() => 0.5 - Math.random()).slice(0, 2);

    let randomNumbers = [2, 4, 8];                         // 可选插入的数字（2、4 或 8）

    // 在选中的两个位置中插入随机数字
    for (let index of randomIndices) {
        let [row, col] = index;
        let randomNumber = randomNumbers[Math.floor(Math.random() * randomNumbers.length)];
        matrix[row][col] = randomNumber;
    }

    newArr.value = randomIndices;                          // 更新新数组的值

    return matrix;                                         // 返回修改后的矩阵
}
```

上述代码有两个关键点，具体如下。

☑ 随机性：空白位置和插入的数字均随机选择，数字 2、4、8 被选择的概率相同。

☑ 游戏终止条件：当空白位置少于两个时，会触发 gameOver()，使得程序结束。

1.6.5 方向移动

设计游戏逻辑模块的第 5 步是定义一个 moveAndMerge()函数，该函数用于实现“挑战 2048 小游戏”的方向移动与合并数字的逻辑。在 1.6.3 节中，已经阐明了合并数字的逻辑。但是，“挑战 2048 小游戏”如果仅支持向上合并数字，那么会不符合 2048 的游戏规则（通常需要支持上、下、左、右 4 个方向）。为此，moveAndMerge()函数的核心逻辑是根据方向调用 addNum()函数，通过矩阵旋转（即调用 rotate90Clockwise()函数）将不同方向的移动统一转化为“向上合并数字”的逻辑，合并后调用 addRandomNumbersToZeros()函数在空白位置随机插入 2、4 或 8，如图 1.11、图 1.12、图 1.13 和图 1.14 所示。

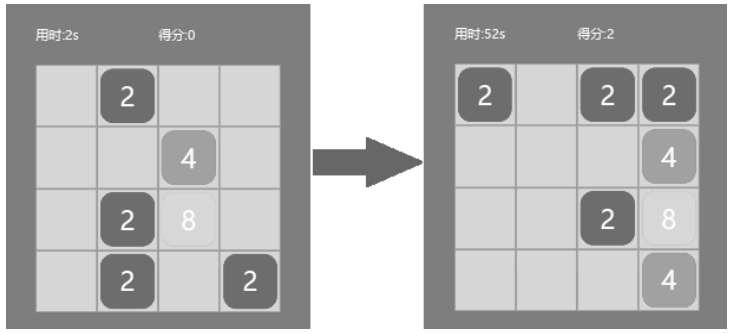


图 1.11 向右移动

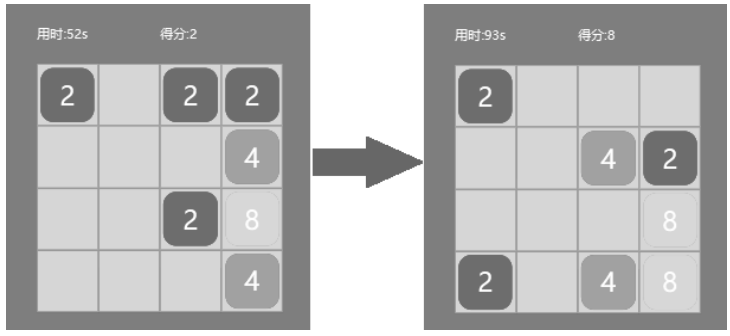


图 1.12 向下移动

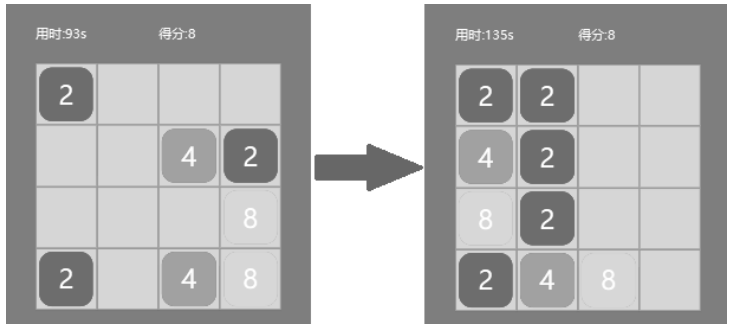


图 1.13 向左移动

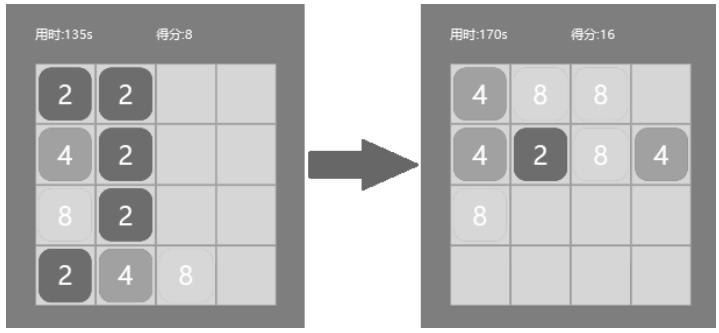


图 1.14　向上移动

实现方向移动的代码如下。

```
const moveAndMerge = (dir) => {
    // 根据方向移动和合并数字
    if (dir == 'shang') {
        // 向上移动并合并
        gameBoard.value = addNum(gameBoard.value);
    } else if (dir == 'zuo') {
        // 向左移动并合并
        let newArr = JSON.parse(JSON.stringify(gameBoard.value));
        newArr = rotate90Clockwise(addNum(rotate90Clockwise(rotate90Clockwise(rotate90Clockwise(newArr)))));
        gameBoard.value = newArr;
    } else if (dir == 'you') {
        // 向右移动并合并
        let newArr = JSON.parse(JSON.stringify(gameBoard.value));
        newArr = rotate90Clockwise(rotate90Clockwise(rotate90Clockwise(addNum(rotate90Clockwise(newArr)))));
        gameBoard.value = newArr;
    } else if (dir == 'xia') {
        // 向下移动并合并
        let newArr = JSON.parse(JSON.stringify(gameBoard.value));
        newArr = rotate90Clockwise(rotate90Clockwise(addNum(rotate90Clockwise(rotate90Clockwise(newArr)))));
        gameBoard.value = newArr;
    }

    // 在移动和合并后，在空白位置添加随机数字
    gameBoard.value = addRandomNumbersToZeros(gameBoard.value);
}
```

上述代码有 3 个关键点，具体如下。

☑ 使用 JSON.parse(JSON.stringify())进行深复制，避免修改原数组。

☑ 通过旋转游戏板将不同方向的移动统一转化为向上移动。

☑ 每次移动、合并后都会调用 addRandomNumbersToZeros()函数在两个随机的空白位置上添加随机数字。

下面讲解实现各个方向移动的逻辑，具体如下。

☑ 向上移动（'shang'）：直接调用 addNum()函数处理；这是基础方向，其他方向通过旋转游戏板转换为向上移动。

☑ 向左移动（'zuo'）：将游戏板顺时针旋转 270°（相当于逆时针旋转 90°），调用 addNum()向上移动（此时相当于原游戏板的向左移动），将结果继续旋转 90°恢复原方向。

☑ 向右移动（'you'）：将游戏板顺时针旋转 90°，调用 addNum()向上移动（此时相当于原游戏板的向右移动），将结果继续旋转 270°恢复原方向。

☑ 向下移动（'xia'）：将游戏板顺时针旋转 180°，调用 addNum()向上移动（此时相当于原游戏板的向下移动），将结果继续旋转 180°恢复原方向。

1.7 事件处理模块设计

在“挑战 2048 小游戏”的事件处理模块设计中，需要执行以下操作：跟踪游戏的当前状态，判断游戏是否结束，更新游戏的得分和时间，以及处理玩家的移动操作。下面将依次介绍上述操作的实现过程。

1.7.1 游戏开始

定义一个 gameStart()函数，该函数用于重置游戏状态、分数、时间，并生成初始游戏面板。其中，定义一个变量 total，用于管理游戏分数，在游戏开始时需要清空历史分数；定义一个变量 allTime，用于记录游戏已进行的时间（秒），计时器启动后会每秒更新该变量的值（相当于每秒更新游戏时间）；定义一个布尔变量 gameStatus，true 表示游戏进行中，false 表示游戏结束；调用 numInit()函数，初始化游戏面板。代码如下。

```
const gameStart = () => {
    total.value = 0;                                    // 初始化总分为 0
    allTime.value = 0;                                  // 初始化总时间为 0
    gameStatus.value = true;                            // 设置游戏状态为进行中
    gameBoard.value = numInit()                         // 初始化游戏板
    // 每秒更新一次游戏总时间
    timer1.value = setInterval(() => {
        allTime.value = allTime.value + 1;
    }, 1000)
}
```

需要说明的是，setInterval()函数用于创建周期性任务，即每隔 1000 毫秒（即 1 秒）执行一次回调函数。回调函数使得变量 allTime 自增 1，实现游戏时间的秒级更新。

1.7.2 游戏结束

定义一个 gameOver()函数，用于在游戏结束时执行清理操作，包括状态重置、计时器清除和数据初始化。其中，将布尔变量 gameStatus 的值设置为 false，表示游戏未开始或游戏结束；停止通过 setInterval()函数创建的计时器；重置计时器变量，释放对计时器的引用；初始化临时数据 newArr 为默认值，确保其在下次游戏开始时可用。代码如下。

```
const gameOver = () => {
    gameStatus.value = false;                           // 设置游戏状态为未开始
    clearInterval(timer1.value);                        // 清除计时器
    timer1.value = null;                                // 重置计时器变量
    // 重置新生成的方块数组
    newArr.value = Array.from({ length: 2 }, () => Array(2).fill(null));
}
```

1.7.3 方向控制

分别定义上、下、左、右 4 个方向的控制函数，这 4 个函数用于处理“挑战 2048 小游戏”中不同方向

的数字移动和合并操作。其中，将方向操作委托给 moveAndMerge()函数处理，通过字符串参数（'shang'、'xia'、'zuo'、'you'）区分移动方向。代码如下。

```
const shang = () => {
    moveAndMerge('shang');                    // 向上移动和合并
}

const xia = () => {
    moveAndMerge('xia');                      // 向下移动和合并
}

const zuo = () => {
    moveAndMerge('zuo');                      // 向左移动和合并
}

const you = () => {
    moveAndMerge('you');                      // 向右移动和合并
}
```

说明

所有方向操作均通过调用 moveAndMerge()函数实现，其优势在于能够有效地避免重复代码。

1.8 时间管理模块设计

在“挑战 2048 小游戏”的时间管理模块设计中，定义了一个游戏计时器功能，该计时器用于在游戏开始后实时统计游戏进行的时间，如图 1.15 所示。下面将介绍计时器的实现过程。

图 1.15 计时器

1.8.1 Vue 3 的 ref

在 Vue 3 中，ref 是响应式系统的核心 API，用于将数据转换为响应式引用。它能够将基本类型的数据（如数字、字符串）或复杂类型的数据（如对象、数组）包装为具有响应性的引用对象。当数据发生变化时，Vue 会自动触发视图的更新。

ref 可以包装任何类型的数据（包括对象），使用 ref 创建响应式数据的代码如下。

```
const count = ref(0);                         // 基本类型
const user = ref({ name: 'Alice', age: 25 });  // 对象或数组
```

在响应式数据被创建完毕后，通过.value 属性能够对已经创建的响应式数据进行访问和修改。代码如下。

```
console.log(count.value);                     // 读取值 → 0
count.value++;                                // 修改值 → 触发更新
user.value.name = 'Bob';                      // 修改对象属性
```

需要特别注意的是，“.value”属性显式标记了响应式变量，并为其提供了一致的访问方式，当.value 属性被修改时，Vue 会自动检测该变化，并更新依赖它的视图或计算属性。

说明

响应式变量是一种特殊的数据容器，当它的值发生变化时，依赖它的代码（如视图渲染、计算属性）会自动重新执行，保持数据与 UI 的同步。

综上所述，ref 是 Vue 3 响应式系统的基石，适用于管理任何类型的响应式数据。.value 属性是访问和修改响应式数据的必要方式。

1.8.2 计时器模块设计

定义两个响应式变量 allTime 和 timer1，allTime 用于存储游戏累计进行的时间（单位：秒），每秒自动加 1，初始值为 0；timer1 用于保存 setInterval()函数返回的计时器 ID，便于后续清除计时器。代码如下。

```
const allTime = ref(0)                    // 定义游戏总时间，初始为 0 秒
const timer1 = ref()                      // 定义计时器变量
```

在 gameStart()函数中初始化计时器，setInterval()函数接收一个回调函数作为参数，每隔 1000 毫秒（1 秒）执行一次该回调函数，并让 allTime.value 自增 1，实现秒级计时。代码如下。

```
// 每秒更新一次游戏总时间
timer1.value = setInterval(() => {
    allTime.value = allTime.value + 1;
}, 1000)
```

从上述代码可知，gameStart()函数被调用时，计时器就会被启动。

说明

setInterval()函数用于按照固定的时间间隔（毫秒）无限循环调用指定的代码（或函数），直到被显式停止。

在游戏结束时，gameOver()函数会被调用，此时将清除计时器。代码如下。

```
clearInterval(timer1.value);              // 清除计时器
```

说明

clearInterval()函数用于停止由 setInterval()函数创建的周期性任务。

1.9 项 目 运 行

通过前述步骤，我们设计并完成了“挑战 2048 小游戏”项目的开发。下面运行本项目，检验一下我们的开发成果。

打开 HBuilderX 开发工具，依次选择“文件”→“导入”→“从本地目录导入”选项，如图 1.16 所示。

单击“从本地目录导入”选项后弹出“打开目录”窗口，进入该项目所在的文件夹，如图 1.17 所示。选择该项目文件夹，单击窗口中的“选择文件夹”按钮即可将该项目文件夹导入 HBuilderX 中。

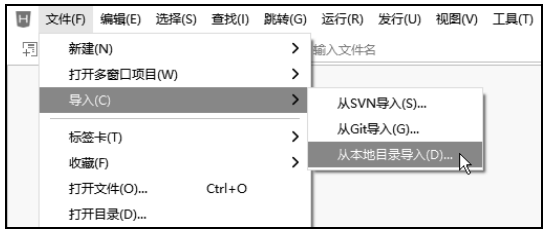


图 1.16　从本地目录导入项目

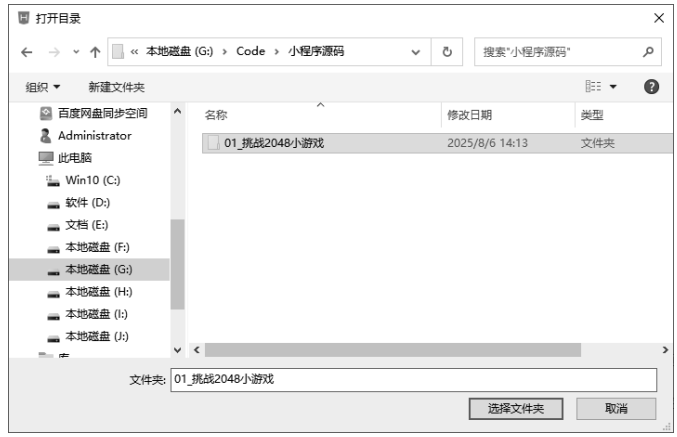


图 1.17　选择项目文件夹

选择要运行的项目文件夹，依次选择“运行”→“运行到浏览器”→“Chrome”选项，如图 1.18 所示。

图 1.18　运行项目

单击“Chrome”选项后即可在 Chrome 浏览器中运行项目。运行项目后会进入“挑战 2048 小游戏”的游戏界面，如图 1.19 所示。

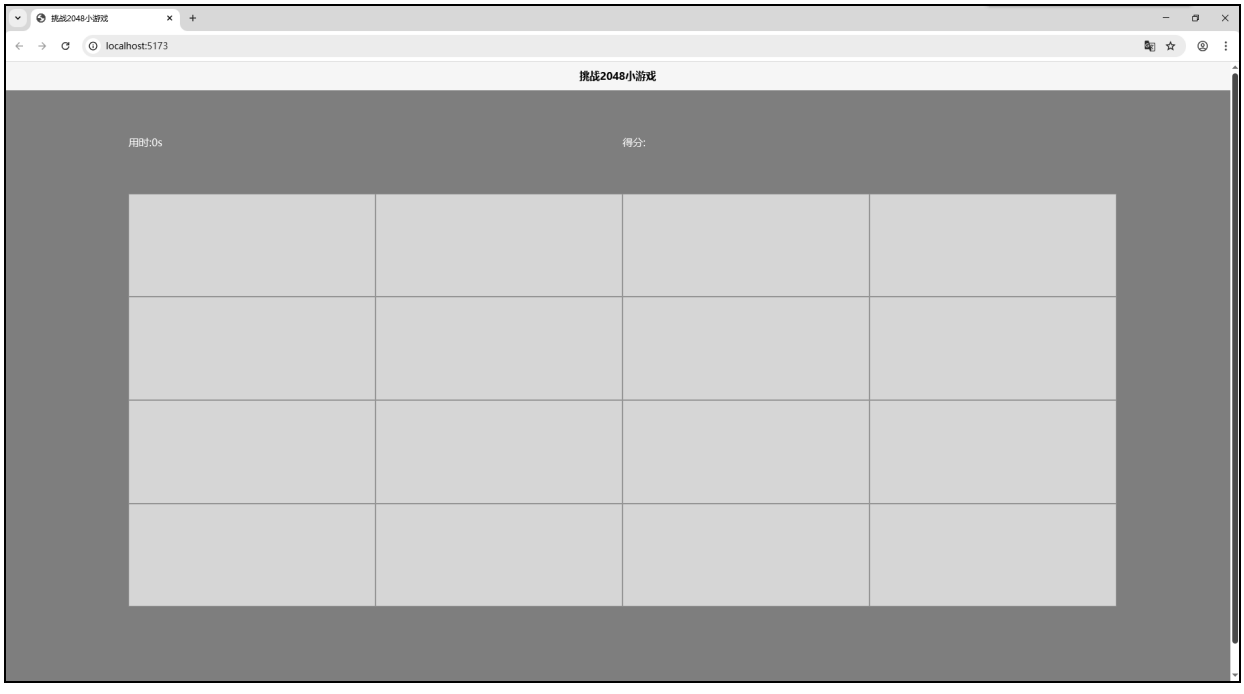


图 1.19　“挑战 2048 小游戏”项目的游戏界面

图 1.19 的游戏界面与图 1.2 的游戏界面不相符，因此需要在键盘上按 F12 快捷键予以调整，调整后的效果如图 1.20 所示。

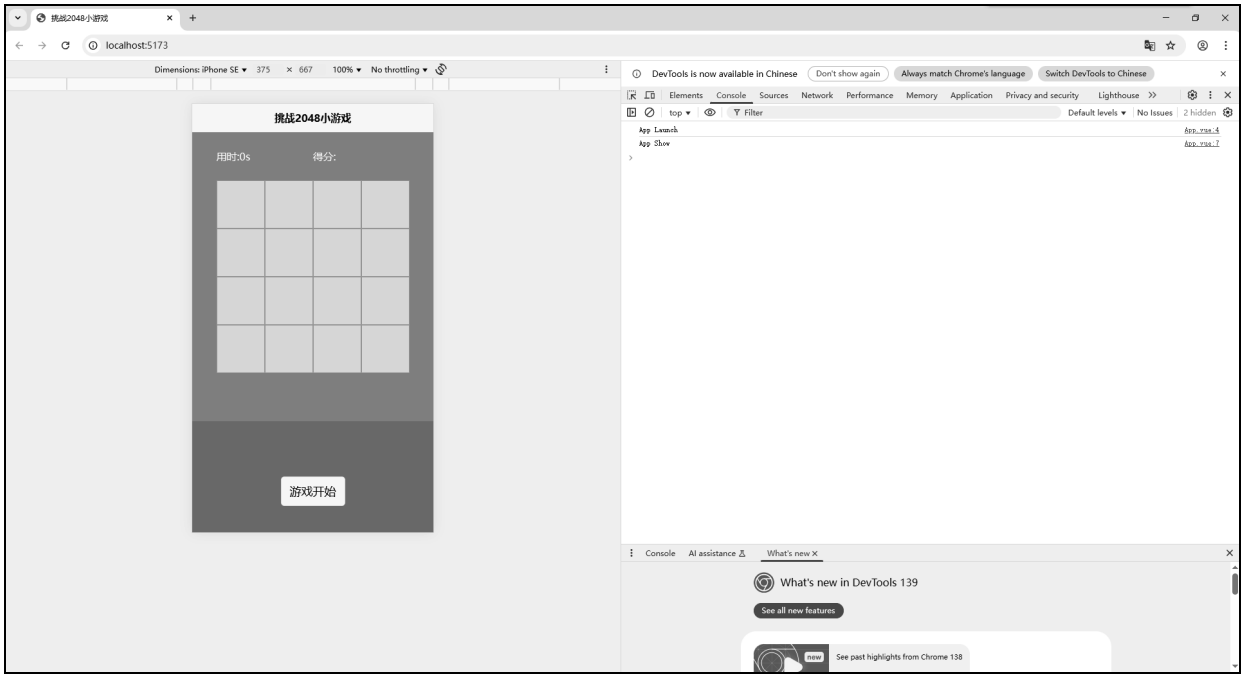


图 1.20　通过 F12 快捷键调整游戏界面

单击游戏界面上的“游戏开始”按钮，这时游戏会在一个空的 4 × 4 网格中随机生成 6 个数字（4 个 2，1 个 4 或 8，1 个 4 或 8），这些数字分布在不同的网格中，游戏开始后的效果如图 1.21 所示。

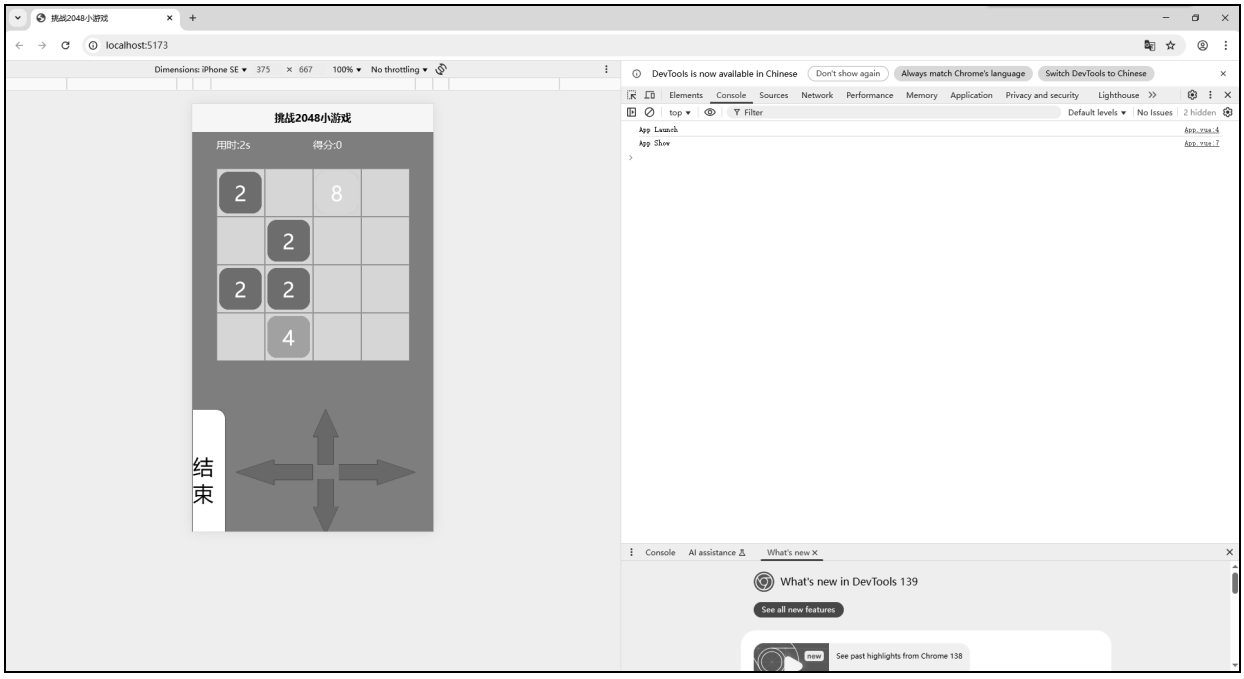


图 1.21　游戏开始后的效果图

玩家可以通过单击上、下、左、右 4 个方向的按钮对网格中的数字执行移动操作。每执行一次移动操作，所有数字都会按照指定方向移动至所在行或列的末端。在移动的过程中，如果行或列有两个相同数字，那么它们将合并为一个数字。每次执行完移动操作后，都会在两个随机的空白网格中插入随机数字（2、4 或 8）。当空白位置少于两个时，游戏结束。这样，我们就成功地检验了本项目的运行。

本项目基于 Vue 3 + TypeScript + uni-app 实现。本项目采用了模块化设计的思路，运用“响应式数据”和“将游戏操作转化为矩阵操作”等编程理念实现了游戏的核心逻辑，使游戏具有计时、计分、自动生成新数字、方向控制等功能。本项目的关键函数包括 gameStart()函数（初始化游戏）、numInit()函数（初始化矩阵）、rotate90Clockwise()函数（旋转矩阵）、addNum()函数（向上合并数字），addRandomNumbersToZeros()函数（添加随机数字）和 moveAndMerge()函数（方向移动）等。此外，本项目还通过合理的架构设计保证了代码的可维护性。

1.10 源 码 下 载

虽然本章详细地讲解了如何编码实现“挑战 2048 小游戏”项目的各个功能，但文中给出的代码都是代码片段，而非完整的源码。为了方便读者学习，本书提供了完整的项目源码，读者扫描右侧二维码即可下载。

源码下载