

前言

基于 mapcn 官方文档的系统性实战指南

WorkBuddy 文档生成团队

2026 年 8 月

目录

前言	4
第一部分 入门与环境	4
第 01 章 初识 mapcn: 地图组件的另一种可能	4
1.1 mapcn 是什么	5
1.2 为什么不直接用 MapLibre	5
1.3 mapcn 与同类方案的对比	5
1.4 它能兼容哪些底图	5
1.5 适用边界与本书路线	6
本章小结	6
第 02 章 环境准备与安装	6
2.1 前置条件	6
2.2 安装地图组件	7
2.3 验证安装结果	7
2.4 关于 import 路径	7
2.5 常见问题	8
本章小结	8
第二部分 地图基础与交互	8
第 03 章 你的第一张地图	8
3.1 最小的地图	8
3.2 在 shadcn Card 中展示	9
3.3 Next.js 下的 “use client”	9
3.4 视口: center、zoom、bearing、pitch	9
3.5 受控模式	10
3.6 主题自动切换	11
本章小结	11

第 04 章 地图控件与交互	11
4.1 一行加全套控件	11
4.2 控制位置	12
4.3 定位回调	12
4.4 受控之外的交互底色	12
4.5 自定义控件示例（预览）	12
本章小结	13
第 05 章 标记点 Markers	13
5.1 组件结构	13
5.2 基础标记	13
5.3 文本标签	14
5.4 自定义标记内容	15
5.5 可拖拽标记	15
5.6 性能边界	16
本章小结	16
第 06 章 弹窗与提示框	16
6.1 MarkerPopup：依附于标记	16
6.2 富交互弹窗	17
6.3 关闭按钮	17
6.4 MarkerTooltip：轻量提示	17
6.5 MapPopup：独立弹窗	17
本章小结	18
第三部分 图形与数据	19
第 07 章 路线与路径规划	19
7.1 基础路线	19
7.2 路线 + 站点标记	19
7.3 接入真实导航：OSRM	20
本章小结	22
第 08 章 弧线连接 Arcs	22
8.1 数据形态	22
8.2 弯曲程度与采样	23
8.3 用 MapLibre 表达式按类别着色	23
8.4 进阶样式入口	24
本章小结	24
第 09 章 GeoJSON 与区域地图	24
9.1 为什么需要 blank 底图	25
9.2 在普通底图上叠加区域	25
9.3 填充与描边的精细控制	26
9.4 数据来源：内联、URL 或类型化	27
本章小结	27

第 10 章 海量点聚合 Clusters	27
10.1 一行启用聚合	27
10.2 关键属性	28
10.3 泛型与类型安全	29
10.4 与海量标记方案的对比	29
本章小结	29
第四部分 进阶与实战	29
第 11 章 深入 MapLibre 实例	29
11.1 用 ref 拿到底层 map	29
11.2 用 useMap 钩子	30
11.3 示例：自定义控件	31
11.4 示例：自定义 GeoJSON 图层	31
11.5 事件清理的重要性	33
本章小结	33
第 12 章 主题、样式与自定义底图	33
12.1 主题自动切换	33
12.2 替换底图：styles 属性	33
12.3 空白画布 blank	34
12.4 投影与 3D 地球	34
12.5 组件外观的 Tailwind 化	34
12.6 小结：样式优先级	34
本章小结	34
第 13 章 性能与最佳实践	35
13.1 三种渲染方式的取舍	35
13.2 受控模式的代价	35
13.3 事件监听必须清理	35
13.4 图层增删先判定	35
13.5 大数据量的取数策略	35
13.6 容器与尺寸	36
13.7 可访问性与降级	36
13.8 一份检查清单	36
本章小结	36
第 14 章 实战：构建一个门店地图应用	36
14.1 数据结构	37
14.2 主组件骨架	37
14.3 联动逻辑解析	39
14.4 还能怎么扩展	39
14.5 收尾：你已经学到了什么	39
本章小结	39
版权与免责声明	40

前言

在 Web 前端，“给页面加一张可交互的地图”似乎从来都不是一件轻松的事。开发者往往要在「引入一个庞大的地图 SDK」「配置 API Key」「与一堆命令式 API 搏斗」之间反复横跳。mapcn 试图把这件事变得更像写普通 UI：组件化、可组合、可定制，且代码归你所有。

mapcn 是一套**免费、开源、可复制粘贴**的 React 地图组件库。它构建于 MapLibre GL 之上，使用 Tailwind CSS 进行样式设计，并刻意契合 shadcn/ui 的使用习惯。你不必把整座地图引擎“安装”进项目，只需要把需要的组件文件复制进来，它们就是你的代码——这正是它区别于传统地图 SDK 的根本之处。

本书的由来，是想给中文世界的 React 开发者一本“既能入门、又能进阶”的 mapcn 专著。它的内容完全建立在 mapcn 官方文档的基础之上——我们通读了项目文档，并按“由浅入深、面面俱到”的原则重新编排了章节结构。你读到的每一处组件 API、每一个属性、每一项用法，都尽量回溯到官方文档的对应出处；涉及版本差异或行为约束处，也如实标注。

全书分为四个部分、共十四章：

- **第一部分 入门与环境 (第 1-2 章)**：从 mapcn 是什么、它和主流方案有何不同，到环境准备、安装与项目脚手架，为后续打下地基。
- **第二部分 地图基础与交互 (第 3-6 章)**：渲染第一张地图，掌握地图控件与视口交互，使用标记点 (Marker)、弹窗 (Popup) 与提示框 (Tooltip)。
- **第三部分 图形与数据 (第 7-10 章)**：用路线 (Route) 与弧线 (Arc) 连接点位，用 GeoJSON 绘制区域地图，以及海量点的聚合 (Cluster) 渲染。
- **第四部分 进阶与实战 (第 11-14 章)**：深入 MapLibre 实例与 useMap，自定义主题与底图，性能优化与最佳实践，并以一个完整的“门店地图应用”收尾。

阅读建议：完全的初学者建议从第 1 章顺序读到第 6 章，把基础打牢；有一定 React 或地图开发经验的读者，可以跳过前几章的入门铺垫，直接从交互或图形数据部分切入；希望理解“组件之下发生了什么”的读者，第四部分值得精读。每章末尾都配有“本章小结”，便于复习与速查。

书中的代码示例以“可运行、可复现”为第一原则，默认基于 mapcn 当前文档所对应版本；涉及版本差异处均显式标注。建议读者在自己的项目中动手敲一遍代码——mapcn 的“复制即拥有”特性，让这种即时实验的成本几乎为零。

最后说明：本书由 AI 辅助文档生成流程编撰而成，内容力求准确、并已尽量对照官方文档核验；但在涉及生产环境的关键决策（如性能调优、数据安全、版本升级）时，仍请以官方文档与社区的最新结论为准。

第一部分 入门与环境

第 01 章 初识 mapcn：地图组件的另一种可能

在 Web 前端，给页面加一张可交互的地图，似乎从来都不是一件轻松的事。开发者往往要在「引入一个庞大的地图 SDK」「配置 API Key」「与一堆命令式 API 搏斗」之间反复横跳。mapcn 试图把这件事变得更像写普通 UI：组件化、可组合、可定制，且代码归你所有。

本章将回答三个问题：mapcn 是什么、它和主流方案有何不同、以及我们应当怎样看待它适用的边界。

1.1 mapcn 是什么

mapcn 是一套**免费、开源、可复制粘贴**的 React 地图组件库。它构建于 MapLibre GL 之上，使用 Tailwind CSS 进行样式设计，并刻意契合 shadcn/ui 的使用习惯。

与「安装一个 npm 包、然后 import 它导出的黑盒组件」不同，mapcn 沿用了 shadcn 的哲学：**组件源码直接复制进你的项目**。这意味着：

- 你拥有全部源码，可以逐行修改；
- 没有运行时锁定，升级、裁剪、分叉都由你决定；
- 组件天然融入你既有的 Tailwind 与 shadcn 设计令牌 (design tokens)。

一句话概括：mapcn 把「地图」从外部依赖，变成了你代码库里的一等公民。

1.2 为什么不直接用 MapLibre

MapLibre GL JS 本身是优秀的开源地图引擎，但它是**命令式**的：你拿到一个 Map 实例，然后调用 `setCenter()`、`flyTo()` 这样的 API。在 React 的声明式世界里，这种写法容易与组件生命周期、状态同步产生摩擦。

mapcn 的价值，正是架在 MapLibre 与 React 之间那层薄而稳的桥：

- 用 `Map`、`MapControls`、`MapMarker` 这类声明式组件描述「想要什么」；
- 把 MapLibre 的初始化、主题切换、图层增删、事件清理等脏活封装起来；
- 当你确实需要底层能力时，`useMap` 钩子与 `MapRef` 又能让你随时下沉到原生实例。

它不是要取代 MapLibre，而是让你**默认用 React 的方式思考，必要时才用 MapLibre 的方式**。

1.3 mapcn 与同类方案的对比

维度	传统地图 SDK	图表库附带地图	mapcn
使用方式	命令式 API	配置对象	声明式组件
代码归属	黑盒依赖	黑盒依赖	复制进项目，完全拥有
主题切换	手动适配	有限	浅/深自动切换
与 shadcn/Tailwind	割裂	割裂	原生契合
底层控制	直接	受限	<code>useMap</code> / <code>map</code> 直达

理解这张表，有助于判断 mapcn 是否适合你的项目：它最适合「想快速交付、又不愿 放弃控制权」的团队，尤其是已经采用 shadcn/ui 与 Tailwind 的 React 应用。

1.4 它能兼容哪些底图

mapcn 不绑定任何特定服务商。只要符合 MapLibre 样式规范的瓦片都能用，包括 Open-StreetMap、Carto、MapTiler、Stadia Maps、Thunderforest 等。默认情况下，mapcn 使用 Carto 的免费底图瓦片，且无需任何 API Key。

提示：本书后续所有示例默认使用 mapcn 自带的 Carto 底图。当你需要品牌化样式或离线底图时，再用 `style` 属性替换即可（见第 12 章）。

1.5 适用边界与本书路线

mapcn 并非银弹。以下场景需要你清醒评估：

- **超大规模实时数据**（十万级以上动态要素）：应更多依赖原生 MapLibre 图层与 WebGL 能力，必要时绕过组件封装；
- **复杂的地图编辑/绘制编辑器**：mapcn 提供基础能力，但专业编辑器仍需自行扩展；
- **非 React 技术栈**：mapcn 是 React 组件库，不适用于 Vue、Svelte 等。

本书的路线由浅入深：

1. **入门（第 2-6 章）**：环境、安装、第一张地图、控件、标记与弹窗；
2. **图形与数据（第 7-10 章）**：路线、弧线、GeoJSON、聚合点；
3. **进阶（第 11-13 章）**：深入实例、主题样式、性能实践；
4. **实战（第 14 章）**：综合构建一个门店地图应用。

本章小结

- mapcn 是构建于 MapLibre 之上的、可复制粘贴的 React 地图组件库；
- 它的核心优势是「声明式 + 代码归你所有 + 与 shadcn/Tailwind 契合」；
- 它与 MapLibre 是协作而非替代关系，Map / useMap 提供底层出口；
- 它兼容任意 MapLibre 样式规范的底图，默认使用免费 Carto 瓦片；
- 适合快速交付且不愿放弃控制权的 React 团队，超大规模或专业编辑器场景需另行评估。

下一章，我们动手把 mapcn 装进你的项目。

第 02 章 环境准备与安装

任何组件库的第一步都是把环境搭好。mapcn 对运行环境的要求很低：一个现代 React 项目、Tailwind CSS，以及 shadcn/ui 的基础配置即可。本章带你逐项确认前置条件，并完成组件的安装。

2.1 前置条件

在开始之前，请确认你的项目满足以下条件：

- **React 18+**：mapcn 使用了 `use()`、并发特性等现代 React API；
- **Tailwind CSS**：组件样式完全基于 Tailwind 工具类；
- **shadcn/ui 基础**：需要 `cn()` 工具函数（通常位于 `lib/utils.ts`）与一套 shadcn 设计令牌（`--background`、`--foreground`、`--border`、`--muted` 等 CSS 变量）。

如果你的项目还没有 shadcn/ui，可先运行官方初始化：

```
npx shadcn@latest init
```

该命令会创建 `components.json`、`lib/utils.ts` 以及 `cn()` 中的设计令牌，这些正是 mapcn 组件所依赖的。

说明： mapcn 组件内部引用了 `cn()` (来自 `lib/utils.ts`) 与一系列语义化颜色令牌。只要你的项目沿用了 shadcn 的约定，组件即可零改动使用。

2.2 安装地图组件

mapcn 以 shadcn registry 的形式分发。运行以下命令，即可把 Map 组件及其依赖 复制进项目：

```
npx shadcn@latest add https://mapcn.dev/r/map.json
```

执行后会发生两件事：

1. 安装运行时依赖 `maplibre-gl`；
2. 在 `components/ui/map.tsx` 写入组件源码（即 `map.tsx` 的内容）。

为什么是「复制」而不是「安装」？ 因为 mapcn 的组件源码直接进入你的仓库，你之后对它的任何修改都不会被上游覆盖。这也是它与普通 npm 包最本质的区别。

2.3 验证安装结果

安装完成后，项目结构应多出以下关键文件：

```
your-project/
├── components/
│   ├── ui/
│   │   └── map.tsx          # mapcn 地图组件（你拥有的源码）
│   └── ...
├── lib/
│   └── utils.ts            # cn() 工具
├── package.json            # 已包含 maplibre-gl
└── ...
```

你可以用一个最小示例确认一切就绪（完整代码见第 3 章）。若浏览器中能渲染出地图，说明安装成功。

2.4 关于 import 路径

本书所有示例统一使用 shadcn 约定路径：

```
import { Map, MapControls } from "@/components/ui/map";
```

其中 `@` 是项目配置的模块别名（指向 `src/`）。若你的项目使用其它别名（如 `~/`），按实际配置调整即可——组件本身不关心路径，只关心导出了哪些符号。

mapcn 导出的顶层符号包括：

- `Map`、`MapControls`：容器与控件；
- `MapMarker`、`MarkerContent`、`MarkerPopup`、`MarkerTooltip`：标记相关；
- `MapPopup`：独立弹窗；
- `MapRoute`、`MapArc`、`MapGeoJSON`、`MapClusterLayer`：图形图层；
- `useMap`：访问底层实例的钩子；

- 类型：MapRef、MapViewport、MapArcDatum 等。

2.5 常见问题

Q: 需要申请 API Key 吗? 不需要。默认底图使用 Carto 的免费瓦片，无需密钥。若切换到 MapTiler 等需要 密钥的服务，则按服务商要求配置。

Q: 能用 Next.js / Vite / CRA 吗? 可以。mapcn 是纯客户端组件库。在 Next.js App Router 下，包含交互逻辑的组件需加 "use client" 指令（详见第 3 章）。

Q: maplibre-gl 的 CSS 去哪了? 组件内部已引入 maplibre-gl 的样式，你无需手动引入。

本章小结

- 前置条件：现代 React + Tailwind CSS + shadcn/ui 基础（提供 cn() 与设计令牌）；
- 安装命令：npx shadcn@latest add https://mapcn.dev/r/map.json；
- 安装结果：运行时依赖 maplibre-gl + 源码 map.tsx；
- 组件以「复制进项目」的方式分发，源码归你所有、可自由修改；
- 统一通过 @/components/ui/map 引入，导出了 Map、各类图层与 useMap 钩子。

下一章，我们渲染出第一张真正可交互的地图。

第二部分 地图基础与交互

第 03 章 你的第一张地图

安装就绪后，第一步自然是让地图出现在页面上。mapcn 的 Map 组件把 MapLibre 的初始化、主题、上下文都封装好了，因此渲染一张地图只需要几行代码。

3.1 最小的地图

新建一个组件，引入 Map，给它一个高度合适的容器即可：

```
import { Map } from "@/components/ui/map";

export function BasicMap() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[-74.006, 40.7128]} zoom={12} />
    </div>
  );
}
```

这里有两个关键点：

- center 是 [经度, 纬度]（注意顺序是**经度在前**），本例指向纽约；

- zoom 是缩放级别，12 大致是「城市街区」的尺度。

地图容器必须有**明确的高度**，否则地图会塌陷为 0 高度而看不见。常见做法是用 `h-[420px]` 这类 Tailwind 工具类，或把地图放进一张 Card 里。

3.2 在 shadcn Card 中展示

把地图放进卡片，是文档站中最常见的布局：

```
import { Map, MapControls } from "@/components/ui/map";
import { Card } from "@/components/ui/card";

export function MapInCard() {
  return (
    <Card className="h-[320px] p-0 overflow-hidden">
      <Map center={[-74.006, 40.7128]} zoom={11}>
        <MapControls />
      </Map>
    </Card>
  );
}
```

Card 提供了圆角与裁剪 (rounded)，MapControls 则叠加了缩放等交互控件（详见第 4 章）。

3.3 Next.js 下的 “use client”

在 Next.js App Router 等「默认服务端渲染」的框架里，任何用到状态、事件、浏览器 API 的组件都必须标注为客户端组件：

```
"use client";

import { Map } from "@/components/ui/map";

export function ClientMap() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[-74.006, 40.7128]} zoom={12} />
    </div>
  );
}
```

原则上，只要组件链中**任何一个**用到了 `useState`、`useEffect`、事件处理或浏览器 API，其所在文件（及所有交互子组件）都应加 `"use client"`。纯展示的静态地图在严格的服务端环境下也能渲染外壳，但加上它最省心。

3.4 视口: center、zoom、bearing、pitch

Map 暴露了完整（且类型安全）的视口属性，常用的有：

属性	含义	典型取值
center	中心坐标 [经度, 纬度]	[-74, 40.7]
zoom	缩放级别 (0= 世界, 22= 楼栋)	12
bearing	方位角 (顺时针旋转, 度)	0
pitch	倾角 (2D→3D, 度)	0

例如, 让地图带一点 3D 倾角:

```
<Map center={[-73.9857, 40.7484]} zoom={15} pitch={60} bearing={-20} />
```

3.5 受控模式

当 viewport 与 onViewportChange 同时传入时, 地图进入**受控模式**: 视口完全 由你的状态驱动, 地图的平移、缩放、旋转都会回调给你。

```
"use client";

import { useState } from "react";
import { Map, type MapViewport } from "@components/ui/map";

export function ControlledMap() {
  const [viewport, setViewport] = useState<MapViewport>({
    center: [-74.006, 40.7128],
    zoom: 8,
    bearing: 0,
    pitch: 0,
  });

  return (
    <div className="relative h-[420px] w-full">
      <Map viewport={viewport} onViewportChange={setViewport} />
      <div className="absolute top-2 left-2 rounded border px-2 py-1 font-mono text-xs">
        <div>lng: {viewport.center[0].toFixed(3)}</div>
        <div>lat: {viewport.center[1].toFixed(3)}</div>
        <div>zoom: {viewport.zoom.toFixed(1)}</div>
      </div>
    </div>
  );
}
```

受控模式的典型用途: 把地图状态写进 URL 以便分享、与外部滑块联动、或做「地图与 列表双向高亮」。非受控场景下, 仅用 onViewportChange 也能单纯「监听」变化, 不 接管控制权。

3.6 主题自动切换

Map 会读取文档根元素的 `dark / light class` (兼容 `next-themes`), 并监听 系统偏好变化, 自动在浅色/深色底图间切换。你**无需写任何额外代码**。

若想强制指定主题, 可传 `theme` 属性:

```
<Map theme="dark" center={[ -74.006, 40.7128]} zoom={11} />
```

本章小结

- 渲染地图只需 Map 组件 (`<Map center={ ... } zoom={ ... } />`), 容器必须有确定高度;
- 把地图放进 Card 是常见且优雅的布局;
- Next.js 等环境下, 含交互逻辑的组件需加 `"use client"`;
- 视口属性 `center/zoom/bearing/pitch` 类型安全, 受控模式用 `viewport + onViewportChange`;
- 主题默认根据文档 `class` 与系统偏好自动切换, 可用 `theme` 覆盖。

下一章, 我们为地图加上交互控件。

第 04 章 地图控件与交互

一张静态地图只能看, 加上控件后才能「用」。mapcn 的 `MapControls` 把缩放、指北针、定位、全屏等常用控件打包成一个声明式组件, 你可以按需在任意角落开关它们。

4.1 一行加全套控件

```
import { Map, MapControls } from "@components/ui/map";

export function MapWithControls() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[2.3522, 48.8566]} zoom={10}>
        <MapControls
          position="top-right"
          showZoom
          showCompass
          showLocate
          showFullscreen
        />
      </Map>
    </div>
  );
}
```

四个开关含义如下:

- `showZoom` : 放大/缩小按钮 (默认开启);
- `showCompass` : 复位方位角的指北针 (点击后 `bearing` 归零、`pitch` 归零);
- `showLocate` : 调用浏览器地理定位, 飞到用户所在位置;
- `showFullscreen` : 切换地图容器的全屏状态。

4.2 控制位置

`position` 决定控件落在哪个角, 可选值:

```
"top-left" | "top-right" | "bottom-left" | "bottom-right" // 默认 bottom-right
```

多数产品把缩放放在右下, 把「定位/全屏」放在右上, 避免与底部版权信息重叠。

4.3 定位回调

当用户点击定位按钮并授权后, `MapControls` 会回调其坐标, 便于你做进一步处理 (例如记录、或在列表中高亮):

```
<MapControls
  showLocate
  onLocate={coords} => {
    console.log(" 用户位于", coords.longitude, coords.latitude);
  }
/>
```

注意: 定位依赖浏览器的 Geolocation API, 必须在 HTTPS (或 `localhost`) 环境下、且用户明确授权后才会成功。被拒绝时按钮会自动复位, 不会卡死。

4.4 受控之外的交互底色

`MapControls` 本质上是对 `MapLibre` 原生控件的轻封装。例如放大按钮调用的是 `map.zoomIn()`, 全屏则操作地图容器的 `requestFullscreen()`。理解了这一点, 当你需要完全自定义控件时, 就能自然地过渡到第 11 章的「直接操作 `map` 实例」。

4.5 自定义控件示例 (预览)

如果你想要文档站里那种「3D 视角 / 复位」按钮, 思路是: 用 `useMap` 拿到 `map`, 在按钮里调用 `map.easeTo()`。完整代码见第 11 章, 这里先看个骨架:

```
"use client";

import { Map, useMap } from "@components/ui/map";
import { Button } from "@components/ui/button";

function Controller() {
  const { map, isLoading } = useMap();
  if (!isLoading) return null;
  return (
```

```

        <Button size="sm" onClick={() => map?.easeTo({ pitch: 60, duration: 800 })}>
          3D 视角
        </Button>
      );
    }
  }
}

```

本章小结

- MapControls 一行即可叠加缩放/指南针/定位/全屏控件；
- 用 position 控制角落，show* 系列开关控制显示哪些；
- onLocate 在用户授权定位后回调坐标；
- 控件是对 MapLibre 原生能力的封装，自定义控件应走第 11 章的 useMap 路线。

下一章，我们在地图上放置标记点。

第 05 章 标记点 Markers

标记 (Marker) 是地图最常见的元素：它把现实世界中的一个点，钉在地图的某个经纬度上。mapcn 用一组可组合的组件描述一个标记——内容、提示、弹窗、标签各司其职，你 按需嵌套即可。

5.1 组件结构

一个完整标记由外到内是这样组合的：

```

<MapMarker longitude={ ... } latitude={ ... }>
  <MarkerContent>      {/* 标记的可视内容（默认是蓝点） */}
    <MarkerLabel />    {/* 可选：文本标签 */}
  </MarkerContent>
  <MarkerTooltip>      {/* 可选：悬停提示框 */}
    提示文字
  </MarkerTooltip>
  <MarkerPopup>        {/* 可选：点击弹窗 */}
    弹窗内容
  </MarkerPopup>
</MapMarker>

```

- MapMarker：标记容器，负责定位与事件；
- MarkerContent：渲染标记本体，不传子组件时显示默认蓝点；
- MarkerLabel：在标记上方/下方显示文字；
- MarkerTooltip：悬停时出现的轻量提示；
- MarkerPopup：点击标记时弹出的卡片。

5.2 基础标记

一个带提示框与弹窗的标记列表：

```

import {
  Map,
  MapMarker,
  MarkerContent,
  MarkerPopup,
  MarkerTooltip,
} from "@components/ui/map";

const locations = [
  { id: 1, name: " 帝国大厦", lng: -73.9857, lat: 40.7484 },
  { id: 2, name: " 中央公园", lng: -73.9654, lat: 40.7829 },
  { id: 3, name: " 时代广场", lng: -73.9855, lat: 40.758 },
];

export function MarkersExample() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[-73.98, 40.76]} zoom={12}>
        {locations.map((loc) => (
          <MapMarker key={loc.id} longitude={loc.lng} latitude={loc.lat}>
            <MarkerContent>
              <div className="bg-primary size-4 rounded-full border-2 border-white shadow-lg" />
            </MarkerContent>
            <MarkerTooltip>{loc.name}</MarkerTooltip>
            <MarkerPopup>
              <div className="space-y-1">
                <p className="font-medium">{loc.name}</p>
                <p className="text-muted-foreground text-xs">
                  {loc.lat.toFixed(4)}, {loc.lng.toFixed(4)}
                </p>
              </div>
            </MarkerPopup>
          </MapMarker>
        ))}
      </Map>
    </div>
  );
}

```

`bg-primary` 之类的类来自 `shadcn` 设计令牌，因此标记颜色会随你的主题自动协调。

5.3 文本标签

`MarkerLabel` 在标记旁显示文字，默认在上方，可用 `position` 改为 `bottom`：

```

<MarkerContent>
  <div className="size-5 rounded-full bg-rose-500 border-2 border-white" />

```

```

    <MarkerLabel position="bottom"> 我的位置 </MarkerLabel>
  </MarkerContent>

```

5.4 自定义标记内容

MarkerContent 的子组件可以是任何 React 节点。配合 MapPin 图标，可以做出「图钉」「店铺」等富样式标记：

```

import { MapPin } from "lucide-react";

<MarkerContent>
  <MapPin className="fill-black stroke-white dark:fill-white" size={28} />
</MarkerContent>

```

5.5 可拖拽标记

给 MapMarker 加 draggable，并通过 onDrag 把最新坐标写回状态：

```

"use client";

import { useState } from "react";
import { Map, MapMarker, MarkerContent, MarkerPopup } from "@/components/ui/map";

export function DraggableMarker() {
  const [pos, setPos] = useState({ lng: -73.98, lat: 40.75 });

  return (
    <div className="h-[420px] w-full">
      <Map center={[-73.98, 40.75]} zoom={12}>
        <MapMarker
          draggable
          longitude={pos.lng}
          latitude={pos.lat}
          onDrag={(lngLat) => setPos({ lng: lngLat.lng, lat: lngLat.lat })}
        >
          <MarkerContent>
            <div className="cursor-move">□</div>
          </MarkerContent>
          <MarkerPopup>
            <p className="text-xs tabular-nums">
              {pos.lat.toFixed(4)}, {pos.lng.toFixed(4)}
            </p>
          </MarkerPopup>
        </MapMarker>
      </Map>
    </div>
  );
}

```

```

    );
  }

```

MapMarker 还提供 onClick、onMouseEnter、onMouseLeave、onDrag、onDragEnd 等事件回调，覆盖大多数交互需求。

5.6 性能边界

MapMarker 是**基于 DOM** 的：每个标记都是一个真实 DOM 节点。在几百个标记以内 它表现良好；但当数据量达到几千、上万时，DOM 标记会让浏览器卡顿。此时应改用 第 10 章介绍的图层方式（GeoJSON 点图层）来渲染海量点。

本章小结

- 标记由 MapMarker 容器 + MarkerContent（本体）+ 可选的 MarkerLabel / MarkerTooltip / MarkerPopup 组合而成；
- MarkerContent 不传子组件时显示默认蓝点，传任意节点即可自定义外观；
- 加 draggable 与 onDrag 可实现可拖拽标记；
- MapMarker 支持点击、悬停、拖拽等多类事件回调；
- DOM 标记适合数百量级，更大规模请转向 GeoJSON 图层方案。

下一章，我们深入弹窗与提示框的细节。

第 06 章 弹窗与提示框

弹窗（Popup）是地图与用户对话的主要媒介：点击店铺看详情、悬停标记看名称，都离不开它。mapcn 区分两种弹窗——**依附于标记的弹窗**（MarkerPopup）与**独立于 标记的弹窗**（MapPopup）——本章逐一讲解。

6.1 MarkerPopup: 依附于标记

MarkerPopup 必须放在 MapMarker 内部，点击标记时弹出：

```

<MapMarker longitude={lng} latitude={lat}>
  <MarkerContent>
    <div className="size-4 rounded-full bg-blue-500 border-2 border-white" />
  </MarkerContent>
  <MarkerPopup>
    <p className="font-medium"> 帝国大厦 </p>
    <p className="text-muted-foreground text-xs"> 纽约 · 观景台 </p>
  </MarkerPopup>
</MapMarker>

```

它的内容是任意 React 节点，因此可以嵌入图片、评分、按钮等富内容，与 shadcn/ui 组件无缝结合。

6.2 富交互弹窗

借助 shadcn 的 Button、Star 与 Navigation 图标，可以拼出信息卡风格的弹窗：

```
import { Button } from "@components/ui/button";
import { Star, Navigation } from "lucide-react";

<MarkerPopup className="w-62">
  <div className="space-y-2 p-3">
    <h3 className="font-semibold"> 大都会艺术博物馆 </h3>
    <div className="flex items-center gap-1 text-sm">
      <Star className="size-3.5 fill-amber-400 text-amber-400" />
      <span className="font-medium">4.8</span>
      <span className="text-muted-foreground">(12,453)</span>
    </div>
    <Button size="sm" className="w-full">
      <Navigation className="size-3.5" /> 导航
    </Button>
  </div>
</MarkerPopup>
```

样式提示：mapcn 已重置 MapLibre 弹窗的默认样式以避免冲突，请使用组件自身的 className 与 shadcn 令牌来定制外观，而不是覆盖 MapLibre 的 maplibre-gl 默认样式 / 弹窗容器。

6.3 关闭按钮

默认弹窗不带关闭按钮（点击地图其它处即可消失）。需要显式关闭按钮时，传 closeButton：

```
<MarkerPopup closeButton>
  <p> 可关闭的内容 </p>
</MarkerPopup>
```

6.4 MarkerTooltip: 轻量提示

MarkerTooltip 在悬停时出现、移出即消失，适合展示极短的信息（名称、状态）：

```
<MarkerTooltip>{loc.name}</MarkerTooltip>
```

它是对 MarkerPopup 的封装，但刻意排除了 closeButton，因为提示框本就不该 被点击关闭。

6.5 MapPopup: 独立弹窗

MapPopup 不依赖标记，可在地图任意经纬度放置，并通过状态编程控制。它必须 放在 Map 内部。

```
"use client";

import { useState } from "react";
```

```

import { Map, MapPopup } from "@components/ui/map";
import { Button } from "@components/ui/button";

export function StandalonePopup() {
  const [show, setShow] = useState(true);

  return (
    <div className="relative h-[420px] w-full">
      <Map center={{-74.006, 40.7128}} zoom={13}>
        {show && (
          <MapPopup
            longitude=-74.006
            latitude=40.7128
            onClose={() => setShow(false)}
            closeButton
            closeOnClick={false}
          >
            <div className="space-y-2">
              <h3 className="font-semibold"> 纽约市 </h3>
              <p className="text-sm text-muted-foreground"> 人口: 830 万 </p>
              <Button size="sm" variant="outline" className="w-full"
                onClick={() => setShow(false)}> 关闭 </Button>
            </div>
          </MapPopup>
        )}
      </Map>
      {!show && (
        <Button className="absolute bottom-4 left-4"
          onClick={() => setShow(true)}> 显示弹窗 </Button>
      )}
    </div>
  );
}

```

常见用法：点击列表项 → 在对应坐标弹出 MapPopup；或聚合点被点击 → 在聚合中心 弹出详情（见第 10 章）。

本章小结

- MarkerPopup 依附于标记，点击弹出，内容可任意富化；
- 用 closeButton 开启关闭按钮，样式走 className + shadow 令牌；
- MarkerTooltip 是悬停提示，轻量、自动消失；
- MapPopup 独立、可编程，靠 longitude/latitude 定位，适合列表联动；
- 两者都基于 MapLibre Popup，但排除了会引发样式冲突的字段。

下一章，我们绘制连接坐标的路线。

第三部分 图形与数据

第 07 章 路线与路径规划

在地图上把一系列坐标连成线，就能表达「路线、轨迹、管线、迁徙路径」等概念。mapcn 的 MapRoute 用一行声明画出一条线；结合外部路由服务，还能做真实的驾车导航。

7.1 基础路线

MapRoute 接收 coordinates (一串 [经度, 纬度]) 即可画线：

```
import { Map, MapRoute } from "@components/ui/map";

const route: [number, number][] = [
  [-74.006, 40.7128], // 市政厅
  [-73.9857, 40.7484], // 帝国大厦
  [-73.9772, 40.7527], // 中央车站
  [-73.9654, 40.7829], // 中央公园
];

export function RouteExample() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[-73.98, 40.75]} zoom={11.2}>
        <MapRoute coordinates={route} color="#3b82f6" width={4} opacity={0.8} />
      </Map>
    </div>
  );
}
```

可调样式包括：

属性	说明	默认值
color	线条颜色 (CSS 颜色)	#3b82f6
width	线宽 (像素)	4
opacity	不透明度 0-1	0.8
dashArray	虚线 [实, 虚]	—
interactive	是否响应点击/悬停	true

例如虚线：<MapRoute dashArray={[2, 2]} />。

7.2 路线 + 站点标记

把 MapRoute 与 MapMarker 组合，是「带有序号站点的路线」的标准写法：

```

import {
  Map,
  MapMarker,
  MarkerContent,
  MarkerTooltip,
  MapRoute,
} from "@components/ui/map";

const stops = [
  { name: " 市政厅", lng: -74.006, lat: 40.7128 },
  { name: " 帝国大厦", lng: -73.9857, lat: 40.7484 },
  { name: " 中央公园", lng: -73.9654, lat: 40.7829 },
];

export function RouteWithStops() {
  return (
    <div className="h-[420px] w-full">
      <Map center={[-73.98, 40.75]} zoom={11.2}>
        <MapRoute coordinates={stops.map((s) => [s.lng, s.lat]) as [number, number][]} />
        {stops.map((stop, i) => (
          <MapMarker key={stop.name} longitude={stop.lng} latitude={stop.lat}>
            <MarkerContent>
              <div className="flex size-4.5 items-center justify-center rounded-full border-2 border-gray-200">
                {i + 1}
              </div>
            </MarkerContent>
            <MarkerTooltip>{stop.name}</MarkerTooltip>
          </MapMarker>
        ))}
      </Map>
    </div>
  );
}

```

7.3 接入真实导航: OSRM

MapRoute 只负责「画线」, 不负责「算路」。要得到真实的驾车路线, 需要调用路由 服务, 例如免费的 OSRM。下面这个例子获取阿姆斯特丹 到鹿特丹的备选路线, 并允许点击切换:

```

"use client";

import { useEffect, useState } from "react";
import { Map, MapMarker, MarkerContent, MapRoute, MarkerLabel } from "@components/ui/map";

const start = { name: " 阿姆斯特丹", lng: 4.9041, lat: 52.3676 };
const end = { name: " 鹿特丹", lng: 4.4777, lat: 51.9244 };

```

```

export function OsrmsRoute() {
  const [routes, setRoutes] = useState<[number, number][][]>([]);
  const [selected, setSelected] = useState(0);

  useEffect(() => {
    const url =
      `https://router.project-osrm.org/route/v1/driving/` +
      `${start.lng},${start.lat};${end.lng},${end.lat}` +
      `?overview=full&geometries=geojson&alternatives=true`;
    fetch(url)
      .then((r) => r.json())
      .then((data) => {
        if (data.routes?.length) {
          setRoutes(data.routes.map((r: any) => r.geometry.coordinates));
        }
      })
      .catch((e) => console.error(" 路线获取失败", e));
  }, []);

  return (
    <div className="h-[500px] w-full">
      <Map center={[4.69, 52.14]} zoom={8.5}>
        {routes.map((coords, i) => (
          <MapRoute
            key={i}
            coordinates={coords}
            color={i === selected ? "#6366f1" : "#94a3b8"}
            width={i === selected ? 6 : 5}
            onClick={() => setSelected(i)}
          />
        ))}
        <MapMarker longitude={start.lng} latitude={start.lat}>
          <MarkerContent><div className="size-5 rounded-full bg-green-500 border-2 border-white" />
          <MarkerLabel position="top">{start.name}</MarkerLabel>
        </MapMarker>
        <MapMarker longitude={end.lng} latitude={end.lat}>
          <MarkerContent><div className="size-5 rounded-full bg-red-500 border-2 border-white" />
          <MarkerLabel position="bottom">{end.name}</MarkerLabel>
        </MapMarker>
      </Map>
    </div>
  );
}

```

要点：

- OSRM 返回的 `geometry.coordinates` 正是 `[number, number][]` 格式，直接喂给 `MapRoute` 的 `coordinates`；
- 用 `selected` 状态高亮当前路线，`onClick` 切换；
- 这是「前端取数 + mapcn 画线」的典型分层：路由算法交给专业服务，渲染交给组件。

本章小结

- `MapRoute` 用 `coordinates` 一串坐标即可画线，可设颜色/线宽/虚线；
- 与 `MapMarker` 组合可表达「带站点的路线」；
- `MapRoute` 只画线、不算路；真实导航需调用 OSRM 等路由服务；
- 典型分层：路由算法交给后端/服务，渲染交给 mapcn。

下一章，我们用弧线表达起点—终点连接。

第 08 章 弧线连接 Arcs

直线适合「路线」，但当你表现**起点—终点的连接关系**——航线、物流通道、资金流向——一根稍微弯曲的弧线往往比直线更有信息张力。mapcn 的 `MapArc` 专门为这类场景设计：它在经纬度空间里用二次贝塞尔曲线把两点连成弧。

8.1 数据形态

每条弧线是一个 `MapArcDatum`：唯一 `id` + `from` + `to`（均为 `[number, number]`）。把多条弧线组成数组传给 `data`：

```
import { Map, MapArc, MapMarker, MarkerContent, MarkerLabel } from "@components/ui/map";

const hub = { name: " 伦敦", lng: -0.1276, lat: 51.5074 };
const destinations = [
  { name: " 纽约", lng: -74.006, lat: 40.7128 },
  { name: " 圣保罗", lng: -46.6333, lat: -23.5505 },
  { name: " 东京", lng: 139.6917, lat: 35.6895 },
];

const arcs = destinations.map((d) => ({
  id: d.name,
  from: [hub.lng, hub.lat] as [number, number],
  to: [d.lng, d.lat] as [number, number],
}));

export function ArcExample() {
  return (
    <div className="h-[420px] w-full">
      <Map center={ [hub.lng, hub.lat] } zoom={1} projection={{ type: "globe" }}>
        <MapArc data={arcs} paint={{ "line-color": "#3b82f6", "line-dasharray": [2, 2] }} interact={...}>
        </Map>
      </div>
    )
  }
```

```

    </div>
  );
}

```

注意这里用了 `projection={{ type: "globe" }}`，把地图切换为 3D 地球——弧线在地球上看起来最自然。

8.2 弯曲程度与采样

属性	说明	默认值
<code>curvature</code>	弧线偏离直线的程度，0 为直线，负值向另一侧弯	0.5
<code>steps</code>	每条弧线的采样点数，越大越平滑	60
<code>id</code>	底层 source/图层前缀	自动

想让弧线更「拱」，就调大 `curvature`；想要笔直连线，设 0。

8.3 用 MapLibre 表达式按类别着色

MapArc 的 `paint` 接受任意 `line-*` 字段，因此可以用 MapLibre 表达式按数据字段着色。下面的例子用 `match` 表达式，把「空运」染成紫色、「海运」染成绿色：

```

"use client";

import { useState } from "react";
import type { ExpressionSpecification } from "maplibre-gl";
import { Map, MapArc, MapPopup, type MapArcDatum } from "@components/ui/map";

interface Lane extends MapArcDatum {
  origin: string;
  destination: string;
  mode: "air" | "sea";
}

const lanes: Lane[] = [
  { id: "dxb-jfk", origin: " 迪拜", destination: " 纽约", from: [55.2708, 25.2048], to: [-74.006, 40.7128] },
  { id: "sin-rtm", origin: " 新加坡", destination: " 鹿特丹", from: [103.8198, 1.3521], to: [4.4777, 51.924] }
];

const colorExpr: ExpressionSpecification = [
  "match", ["get", "mode"], "air", "#a78bfa", "sea", "#34d399", "#888",
];

export function ColoredArcs() {
  const [sel, setSel] = useState<Lane | null>(null);
  return (

```

```

<div className="h-[420px] w-full">
  <Map center={[20, 20]} zoom={0.8}>
    <MapArc<Lane>
      data={lanes}
      paint={{ "line-color": colorExpr, "line-width": 1.5 }}
      hoverPaint={{ "line-width": 3, "line-opacity": 1 }}
      onHover={e => setSel(e ? e.arc : null)}
    />
    {sel && (
      <MapPopup longitude={sel.from[0]} latitude={sel.from[1]}>
        {sel.origin} → {sel.destination}
      </MapPopup>
    )}
  </Map>
</div>
);
}

```

这里出现了两个进阶属性：

- `hoverPaint`：悬停某条弧线时叠加的样式（通过 `feature-state` 实现，只影响当前弧）；
- `onHover`：悬停变化时的回调，可用于联动弹窗或侧栏。

8.4 进阶样式入口

`MapArc` 暴露了完整的 `paint` 与 `hoverPaint` 属性，等价于直接配置一个 `MapLibre line` 图层。常见可用字段：

- `line-color` / `line-width` / `line-opacity` / `line-dasharray`
- （虚线）
- `line-cap` / `line-join`（连接与端点样式）

当内置属性不够用时，使用第 11 章的 `useMap` 直接操作底层图层是最灵活的退路。

本章小结

- `MapArc` 用 `data`（每条含 `from/to`）绘制经纬度空间的贝塞尔弧线；
- `curvature` 控制弯曲程度，`steps` 控制平滑度；
- `paint` 接受完整 `MapLibre` 表达式，可按字段着色（如空运/海运）；
- `hoverPaint` + `onHover` 实现悬停高亮与联动；
- 样式不足时，可下沉到 `useMap` 直接操作 `line` 图层。

下一章，我们用 `GeoJSON` 渲染面状区域。

第 09 章 GeoJSON 与区域地图

点、线之外，地图最有力的表达是面——国家边界、行政区、商圈、学区。这类数据几乎总是以 `GeoJSON` 格式存在。`mapcn` 的 `MapGeoJSON` 把任意 `GeoJSON` 渲染成「填充 + 描边」图

层，开箱即用、主题感知，是做分级着色图（choropleth）与区域地图的主力。

9.1 为什么需要 blank 底图

上一章的弧线、路线都画在默认街道底图上。而国家/区域地图通常需要一个**干净的画布**——不希望街道、注记干扰图形本身。这时给 Map 加 blank 属性：

```
import { Map, MapGeoJSON } from "@components/ui/map";

// 假设 WORLD_GEOJSON 是一个 FeatureCollection (国家边界)
export function WorldMap() {
  return (
    <div className="h-[420px] w-full">
      <Map blank center={[10, 25]}>
        <MapGeoJSON data={WORLD_GEOJSON} linePaint={false} />
      </Map>
    </div>
  );
}
```

blank 是一块透明的、无瓦片画布；单独使用它什么都不显示，必须叠加自己的图层。linePaint={false} 表示只画填充、不画边界线，得到一个纯色的「世界色块」。

9.2 在普通底图上叠加区域

更多时候，你想在真实街道底图上**高亮一块区域**（如某商圈）。这时不用 blank：

```
import { Map, MapGeoJSON } from "@components/ui/map";

const area: GeoJSON.FeatureCollection = {
  type: "FeatureCollection",
  features: [
    {
      type: "Feature",
      properties: {},
      geometry: {
        type: "Polygon",
        coordinates: [
          [
            [-122.42, 37.78], [-122.398, 37.785], [-122.392, 37.768],
            [-122.412, 37.758], [-122.43, 37.77], [-122.42, 37.78],
          ],
        ],
      },
    },
  ],
};

export function OverlayExample() {
```

```

return (
  <div className="h-[420px] w-full">
    <Map center={[-122.41, 37.772]} zoom={12}>
      <MapGeoJSON
        data={area}
        fillPaint={{ "fill-color": "#3b82f6", "fill-opacity": 0.25 }}
        linePaint={{ "line-color": "#2563eb", "line-width": 2 }}
      />
    </Map>
  </div>
);
}

```

9.3 填充与描边的精细控制

MapGeoJSON 的关键样式属性：

属性	说明
fillPaint	多边形填充 paint；传 false 可省略填充层
linePaint	描边 paint；传 false 可省略描边层
fillHoverPaint	悬停要素的填充样式（需 interactive）
promoteId	提升为要素 id 的属性，悬停/点击联动所需
interactive	是否响应鼠标事件（默认 false）

默认填充是「主题感知的单色」，在浅色/深色下都清晰可读。你可以用 MapLibre 表达式做**数据驱动着色**，例如按人口上色：

```

<MapGeoJSON
  data={provinces}
  promoteId="id"
  interactive
  fillPaint={{
    "fill-color": [
      "interpolate", ["linear"], ["get", "population"],
      1000000, "#fee5d9",
      10000000, "#fc9272",
      50000000, "#de2d26",
    ],
  }}
  fillHoverPaint={{ "fill-opacity": 0.95 }}
  onHover={{(e) => e && setHovered(e.feature.properties.name)}}
/>

```

这里 promoteId 把每个要素的 id 属性提升为要素 id，从而使 onHover 的悬停高亮能精确定位到具体省；onHover 还能把省名同步到侧栏。

9.4 数据来源：内联、URL 或类型化

MapGeoJSON 接受四种形式：

1. 内联 `FeatureCollection` / `Feature` / `Geometry` (如上一节)；
2. 一个 URL 字符串，组件会自动 `fetch`；
3. 类型化泛型 `MapGeoJSON<T>`，获得属性类型提示；
4. 通过 `beforeId` 控制图层插入顺序 (`z-index`)。

例如直接拉取在线 GeoJSON：

```
<MapGeoJSON
  data="https://example.com/api/countries.geojson"
  promoteId="iso_code"
  interactive
  fillHoverPaint={{ "fill-color": "#1d4ed8" }}
/>
```

本章小结

- MapGeoJSON 把任意 GeoJSON 渲染为「填充 + 描边」图层；
- `blank` 底图适合纯区域/数据地图，`linePaint={false}` 可只画线；
- `fillPaint` / `linePaint` 接受 MapLibre 表达式，可做数据驱动分级着色；
- `promoteId` + `interactive` + `onHover` 实现悬停高亮与联动；
- MapGeoJSON 支持内联、URL、类型化泛型与 `z-order` 控制。

下一章，我们处理海量点的聚合。

第 10 章 海量点聚合 Clusters

当数据点从「几十个」增长到「成千上万个」——地震震中、充电桩、门店分布——逐一渲染标记既卡又乱。地图的经典解法是**聚合 (clustering)**：低缩放时把邻近点收拢成「数字气泡」，放大后再展开为单点。mapcn 的 `MapClusterLayer` 直接封装了 MapLibre 原生的聚合能力。

10.1 一行启用聚合

```
"use client";

import { useState } from "react";
import { Map, MapClusterLayer, MapPopup, MapControls } from "@components/ui/map";

interface QuakeProps {
  mag: number;
  place: string;
  tsunami: number;
}

export default function ClusterExample() {
```

```

const [point, setPoint] = useState<{
  coordinates: [number, number];
  properties: QuakeProps;
} | null>(null);

return (
  <div className="h-[420px] w-full">
    <Map center={[-103.59, 40.66]} zoom={3.4}>
      <MapClusterLayer<QuakeProps>
        data="https://maplibre.org/maplibre-gl-js/docs/assets/earthquakes.geojson"
        clusterRadius={50}
        clusterMaxZoom={14}
        onPointClick={(feature, coordinates) => setPoint({ coordinates, properties: feature.properties })}
      />
      {point && (
        <MapPopup
          longitude={point.coordinates[0]}
          latitude={point.coordinates[1]}
          onClose={() => setPoint(null)}
          closeOnClick={false}
          closeButton
        >
          <p> 震级: {point.properties.mag}</p>
          <p> 海啸: {point.properties.tsunami} ≡ 1 ? " 是" : " 否"</p>
        </MapPopup>
      )}
      <MapControls />
    </Map>
  </div>
);
}

```

点击聚合气泡会自动放大展开；点击未聚合的单点则触发 `onPointClick`，我们在回调里 弹出一个 `MapPopup` 显示详情。

10.2 关键属性

属性	说明	默认值
<code>data</code>	GeoJSON 点集合，或用于拉取的 URL	—
<code>clusterMaxZoom</code>	进行聚合的最大缩放级别	14
<code>clusterRadius</code>	聚合半径（像素）	50
<code>clusterColors</code>	簇圆配色（浅 → 深，按点数分级）	自动
<code>color</code>	未聚合单点颜色	自动
<code>onPointClick</code>	单点被点击的回调	—
<code>onClusterClick</code>	簇被点击的回调（不提供则放大）	—

簇圆的大小与颜色会随包含点数，按 `clusterColors` 分级变化——点数越多，圆越大、颜色越深。

10.3 泛型与类型安全

`MapClusterLayer<T>` 让 `data` 拥有类型，回调里能直接访问 `properties` 而不必任何类型断言。这是 mapcn 对 GeoJSON 图层一贯的类型化思路（与第 9 章 `MapGeoJSON` 一致）。

10.4 与海量标记方案的对比

方案	适用规模	特点
<code>MapMarker</code> (DOM)	数百	富样式、易交互，但 DOM 开销大
<code>MapGeoJSON</code> 点图层	数千 ~ 数万	WebGL 渲染，性能好，样式较统一
<code>MapClusterLayer</code>	数千 ~ 数十万	自动聚合，最佳可读性

经验法则：点数过千就应考虑聚合；需要「每个点有独立 DOM 样式」时才用 `MapMarker`。

本章小结

- `MapClusterLayer` 封装 MapLibre 原生聚合，低缩放收拢、放大展开；
- 通过 `clusterRadius` / `clusterMaxZoom` / `clusterColors` / `color` 调聚合外观；
- `onPointClick` / `onClusterClick` 分别处理单点与簇的交互；
- 泛型 `MapClusterLayer<T>` 提供属性类型安全；
- 点数过千优先聚合，富样式单点才用 DOM 标记。

下一章，我们深入底层 MapLibre 实例。

第四部分 进阶与实战

第 11 章 深入 MapLibre 实例

声明式组件覆盖了 90% 的场景，但总有 10% 需要直接操作地图引擎——自定义动画、监听原生事件、增删任意图层。mapcn 为此留了两个出口：`ref` 与 `useMap` 钩子。本章把它们讲透。

11.1 用 ref 拿到底层 map

`Map` 的 `ref` 指向原生的 `MapRef` 实例（`MapRef` 类型即别名）。在事件处理函数或副作用里调用它即可：

```
import { Map, type MapRef } from "@components/ui/map";
import { useRef } from "react";

function FlyToButton() {
  const mapRef = useRef<MapRef>(null);
```

```

return (
  <>
    <button onClick={() => mapRef.current?.flyTo({ center: [-74, 40.7], zoom: 12 })}>
      飞往纽约
    </button>
    <Map ref={mapRef} center={[-74, 40.7]} zoom={10} />
  </>
);
}

```

flyTo()、setCenter()、getSource()、addLayer()、removeLayer().....凡是 MapLibre Map 上的方法，这里都能直接用。

11.2 用 useMap 钩子

当你的自定义组件渲染在 Map 内部时，用 useMap 更自然——它返回 { map, isLoading }，无需手动传 ref：

```

"use client";

import { Map, useMap } from "@components/ui/map";
import { useEffect } from "react";

function MapEventListener() {
  const { map, isLoading } = useMap();

  useEffect(() => {
    if (!map || !isLoading) return;
    const handleClick = (e: any) => console.log(" 点击位置:", e.lngLat);
    map.on("click", handleClick);
    return () => map.off("click", handleClick);
  }, [map, isLoading]);

  return null;
}

export function Demo() {
  return (
    <Map center={[-74, 40.7]} zoom={10}>
      <MapEventListener />
    </Map>
  );
}

```

关键：useMap 必须在 Map 内调用，否则会抛错。isLoading 表示地图 样式已加载就绪——在它变 true 之前，不要急着 map.on()。

11.3 示例：自定义控件

用 useMap 做一个「3D 视角 / 复位」控制器，实时显示 pitch 与 bearing：

```
"use client";

import { useEffect, useState } from "react";
import { Map, useMap } from "@components/ui/map";
import { Button } from "@components/ui/button";

function Controller() {
  const { map, isLoading } = useMap();
  const [pitch, setPitch] = useState(0);
  const [bearing, setBearing] = useState(0);

  useEffect(() => {
    if (!map || !isLoading) return;
    const onMove = () => {
      setPitch(Math.round(map.getPitch()));
      setBearing(Math.round(map.getBearing()));
    };
    map.on("move", onMove);
    return () => map.off("move", onMove);
  }, [map, isLoading]);

  if (!isLoading) return null;

  return (
    <div className="absolute top-3 left-3 flex flex-col gap-2">
      <Button size="sm" variant="secondary"
        onClick={() => map?.easeTo({ pitch: 60, bearing: -20, duration: 1000 })}>
        3D 视角
      </Button>
      <Button size="sm" variant="secondary"
        onClick={() => map?.easeTo({ pitch: 0, bearing: 0, duration: 1000 })}>
        复位
      </Button>
      <div className="font-mono text-xs">pitch: {pitch}° / bearing: {bearing}°</div>
    </div>
  );
}
```

11.4 示例：自定义 GeoJSON 图层

当内置的 MapGeoJSON 不够用时，直接在 map 里增删 MapLibre 原生图层。下面 给纽约的两座公园加填充 + 描边：

```

"use client";

import { useEffect } from "react";
import { Map, useMap } from "@components/ui/map";

const parks = {
  type: "FeatureCollection" as const,
  features: [
    { type: "Feature" as const, properties: { name: " 中央公园" },
      geometry: { type: "Polygon" as const, coordinates: [[
        [-73.9731, 40.7644], [-73.9819, 40.7681], [-73.958, 40.8006],
        [-73.9493, 40.7969], [-73.9731, 40.7644],
      ]] } },
  ],
};

function ParkLayer() {
  const { map, isLoading } = useMap();

  useEffect(() => {
    if (!map || !isLoading) return;
    if (!map.getSource("parks")) {
      map.addSource("parks", { type: "geojson", data: parks });
    }
    if (!map.getLayer("parks-fill")) {
      map.addLayer({
        id: "parks-fill",
        type: "fill",
        source: "parks",
        paint: { "fill-color": "#22c55e", "fill-opacity": 0.4 },
      });
    }
    if (!map.getLayer("parks-line")) {
      map.addLayer({
        id: "parks-line",
        type: "line",
        source: "parks",
        paint: { "line-color": "#16a34a", "line-width": 2 },
      });
    }
  }, [map, isLoading]);

  return null;
}

```

要点：先判断 `getSource` / `getLayer` 是否存在再添加，避免重复添加报错；所有 增删都应

在 `isLoading` 为真后进行。

11.5 事件清理的重要性

自定义图层与事件监听**不会被** `mapcn` 自动清理。组件卸载时务必移除，否则会造成 内存泄漏或「幽灵图层」：

```
useEffect(() => {
  if (!map || !isLoading) return;
  const handler = () => { /* ... */ };
  map.on("click", handler);
  return () => map.off("click", handler); // 必须清理
}, [map, isLoading]);
```

本章小结

- 两个底层出口：`ref`（指向原生 `MapRef`）与 `useMap`（内部组件用）；
- `useMap` 返回 `{ map, isLoading }`，且必须在 `Map` 内调用；
- 自定义控件/图层都在 `isLoading` 为真后操作 `map` 实例；
- 增删图层前先判断是否存在；事件监听务必在卸载时清理。

下一章，我们谈主题、样式与自定义底图。

第 12 章 主题、样式与自定义底图

地图不是孤立的视觉元素——它要融入你的整体设计系统。本章讲清三件事：浅/深主题 如何自动切换、如何替换底图、以及 3D 地球等投影玩法。

12.1 主题自动切换

`Map` 默认读取文档根元素的 `dark / light class`（兼容 `next-themes` 的 `dark` 与 `light` 两种模式），并监听系统偏好变化，在 `Carto` 的浅色（`Positron`）与深色（`Dark Matter`）底图间无缝切换。你**无需写任何** 切换代码。

若想强制指定：

```
<Map theme="dark" center={[-74.006, 40.7128]} zoom={11} />
```

12.2 替换底图: `styles` 属性

`styles` 接受一个分别针对浅/深主题的样式对象，覆盖默认 `Carto` 底图：

```
const myStyles = {
  light: "https://tiles.openfreemap.org/styles/bright",
  dark: "https://tiles.openfreemap.org/styles/liberty",
};

<Map styles={myStyles} center={[-0.1276, 51.5074]} zoom={15} />
```

也可以使用完整的 MapLibre `StyleSpecification` 对象（而非 URL），便于完全程序化构建样式。注意：一旦显式提供 `styles`，`style` 属性会被忽略。

12.3 空白画布 blank

`blank` 用一块透明无瓦片的画布替换街道底图，是数据可视化的理想基底（分级着色图、世界弧线、点阵图都在此之上绘制自己的图层）：

```
<Map blank center={[10, 25]}>
  <MapGeoJSON data={WORLD_GEOJSON} />
</Map>
```

透明背景会让下方的主题容器透出，因此浅色/深色模式下都自然协调。

12.4 投影与 3D 地球

`projection` 属性控制地图投影。传入 `{ type: "globe" }` 即可切换到 3D 地球视图，弧线与全球航线在地球上最具表现力：

```
<Map center={[0, 20]} zoom={1} projection={{ type: "globe" }}>
  <MapArc data={globalArcs} />
</Map>
```

也可以在用户交互后动态切换投影（需地图样式已加载）：

```
map.setProjection({ type: "globe" });
```

12.5 组件外观的 Tailwind 化

`mapcn` 组件的外观完全由 Tailwind 类驱动，因此「换肤」就是改类：把标记的蓝点改成品牌色，只需换 `bg-primary` → `bg-rose-500`；把弹窗内边距调小，直接在 `className` 上改 `p-3`。没有独立的「主题配置文件」需要维护。

12.6 小结：样式优先级

当多个样式来源冲突时，优先级如下：

1. 显式 `style/styles` 属性（最高）；
2. `blank` 透明画布；
3. 默认 Carto 浅/深底图（随主题自动）。

图层类的样式（`MapRoute` 的 `paint`、`MapGeoJSON` 的 `fillPaint`）始终叠加在底图之上，不受底图切换影响。

本章小结

- 主题默认按文档 `class` 与系统偏好自动切换，可用 `theme` 覆盖；
- `styles` 分别指定浅/深底图（URL 或 `StyleSpecification`），会屏蔽 `style`；
- `blank` 是透明画布，适合数据可视化；
- `projection` 切换 3D 地球；

- 组件外观即 Tailwind 类，换肤 = 改类，无需主题配置文件。

下一章，我们聚焦性能与最佳实践。

第 13 章 性能与最佳实践

地图应用的性能坑往往在使用量上来后才暴露。本章把前文散落的性能要点收敛成一份可操作的清单，帮你在「功能」与「流畅」之间拿稳平衡。

13.1 三种渲染方式的取舍

方式	渲染机制	适用规模	交互粒度
MapMarker (DOM)	真实 DOM 节点	数百	每点独立、富样式
MapGeoJSON 点图层	WebGL	数千 ~ 数万	统一样式、批量
MapClusterLayer	WebGL + 聚合	数千 ~ 数十万	聚合/单点

经验法则：点数过千，优先 MapClusterLayer 或 MapGeoJSON 点图层；只有需要「每个标记都是独特 DOM 组件」时才用 MapMarker。

13.2 受控模式的代价

受控模式 (viewport + onViewportChange) 会在**每一次**平移、缩放、旋转时触发回调，频繁 onViewportChange 可能带来重渲染压力。建议：

- 仅在确实需要「地图状态」与「应用状态」同步时启用；
- 若只想「监听」变化，单独用 onViewportChange 而不传 viewport，地图仍是非受控的，回调不会驱动重渲染；
- 对高频回调做节流 (throttle)，或只记录最终值。

13.3 事件监听必须清理

自定义 map.on() 监听要在组件卸载时 map.off() (见第 11 章)。未清理的监听会随组件反复挂载而累积，导致内存泄漏与诡异行为。

13.4 图层增删先判定

直接用 map 增删图层时，先 getSource / getLayer 判断存在性再添加，避免重复添加报错；卸载时移除自己创建的图层与 source。mapcn 的内置图层 (Route/Arc/ GeoJSON/ Cluster) 已自动管理生命周期，无需你清理。

13.5 大数据量的取数策略

- **静态大数据：**优先用 URL 让组件 fetch，或内联一次性数据，避免每次渲染重建大型对象；

- **动态数据**：用 `useMemo` 缓存 GeoJSON 结构，仅在数据真正变化时更新，减少 `fetch` 调用；
- **表达式优于循环**：用 MapLibre 表达式（如 `interpolate / match`）按字段着色，而不是为每个要素单独建图层。

13.6 容器与尺寸

- 地图容器**必须有确定高度**，否则塌陷为 0；
- 频繁改变容器尺寸时，MapLibre 会自动重排，但应避免在动画帧里反复改尺寸；
- 移动端注意 `100vh` 在地址栏收起时的抖动，必要时用 `100dvh` 或固定高度。

13.7 可访问性与降级

- 给交互控件加 `aria-label`（`mapcn` 已为内置按钮加好）；
- 地图本身对屏幕阅读器不友好，关键数据应同时在列表/表格中以文本形式提供；
- 地理定位等依赖用户授权的能力，要在被拒绝时优雅降级（见第 4 章 `showLocate`）。

13.8 一份检查清单

上线前对照：

- ☐ 点数规模是否选对了渲染方式？
- ☐ 受控模式是否必要？回调是否节流？
- ☐ 所有 `map.on()` 都有对应 `map.off()`？
- ☐ 自定义图层是否在卸载时移除？
- ☐ 大数据是否用 `useMemo` / `URL` / 表达式优化？
- ☐ 容器高度是否确定、移动端是否稳定？

本章小结

- 渲染方式按规模取舍：数百用 DOM 标记，数千以上用图层/聚合；
- 受控模式按需启用，监听变化可用单向 `onViewportChange`；
- 事件监听与自定义图层务必清理；
- 用 `useMemo`、`URL` 取数、表达式着色优化大数据；
- 容器高度确定、关键数据文本化，是可访问与稳定的底线。

下一章，我们用一个完整案例把所学串起来。

第 14 章 实战：构建一个门店地图应用

从前 13 章走到这里，你已经掌握了 `mapcn` 的全部核心能力。本章把它们串成一个完整可运行的「门店地图」：左侧是可筛选的门店列表，右侧是地图，二者通过选中状态双向联动；地图用聚合呈现全部门店，点击单点弹出详情；并带搜索与主题切换。

14.1 数据结构

```
interface Store {
  id: string;
  name: string;
  category: string;    // 餐饮 / 零售 / 服务
  lng: number;
  lat: number;
  address: string;
  rating: number;
}
```

把若干门店组织成 GeoJSON FeatureCollection，便于交给 MapClusterLayer：

```
const storesToFC = (stores: Store[]): GeoJSON.FeatureCollection => ({
  type: "FeatureCollection",
  features: stores.map((s) => ({
    type: "Feature",
    properties: { id: s.id, name: s.name, category: s.category, rating: s.rating },
    geometry: { type: "Point", coordinates: [s.lng, s.lat] },
  })),
});
```

14.2 主组件骨架

```
"use client";

import { useMemo, useState } from "react";
import { Map, MapClusterLayer, MapPopup, MapControls } from "@components/ui/map";
import { Card } from "@components/ui/card";

export function StoreMap({ stores }: { stores: Store[] }) {
  const [selected, setSelected] = useState<Store | null>(null);
  const [query, setQuery] = useState("");

  const filtered = useMemo(
    () => stores.filter((s) => s.name.includes(query) || s.category.includes(query)),
    [stores, query],
  );

  const fc = useMemo(() => storesToFC(filtered), [filtered]);

  return (
    <div className="grid h-[600px] grid-cols-[320px_1fr] gap-4">
      { /* 左侧列表 */ }
      <Card className="overflow-auto p-3">
        <input
          value={query}

```

```

      onChange={(e) => setQuery(e.target.value)}
      placeholder=" 搜索门店..."
      className="mb-3 w-full rounded border px-2 py-1 text-sm"
    />
    {filtered.map((s) => (
      <button
        key={s.id}
        onClick={() => setSelected(s)}
        className="block w-full rounded px-2 py-1.5 text-left text-sm hover:bg-accent"
      >
        <div className="font-medium">{s.name}</div>
        <div className="text-muted-foreground text-xs">{s.category} · {s.rating}</div>
      </button>
    ))}
  </Card>

  {/* 右侧地图 */}
  <Card className="relative overflow-hidden p-0">
    <Map center={[116.4, 39.9]} zoom={10}>
      <MapClusterLayer<Store>
        data={fc}
        onPointClick={(f) => {
          const p = f.properties;
          setSelected(stores.find((s) => s.id === p.id) ?? null);
        }}
      />
      {selected && (
        <MapPopup
          longitude={selected.lng}
          latitude={selected.lat}
          onClose={() => setSelected(null)}
          closeButton
          closeOnClick={false}
        >
          <div className="space-y-1">
            <p className="font-semibold">{selected.name}</p>
            <p className="text-muted-foreground text-xs">{selected.address}</p>
            <p className="text-xs">{selected.category} · {selected.rating}</p>
          </div>
        </MapPopup>
      )}
      <MapControls />
    </Map>
  </Card>
</div>
);

```

```
}
```

14.3 联动逻辑解析

- **列表 → 地图**：点击列表项 `setSelected(s)`，地图中对应的 `MapPopup` 在 `selected` 坐标处弹出；
- **地图 → 列表**：点击聚合中的单点触发 `onPointClick`，用要素 `id` 反查 `stores` 得到完整对象并 `setSelected(...)`，列表可用 `selected` 高亮当前项（此处略去高亮样式）；
- **搜索**：`query` 过滤 `filtered`，`useMemo` 派生 `fc`，聚合层随之更新——这正符合第 13 章「大数据用 `useMemo` 缓存」的建议。

14.4 还能怎么扩展

这个骨架已经覆盖了真实项目最常见的模式，接下来可自然延伸：

- **分类筛选**：用 `paint` 表达式为不同 `category` 着色（见第 8 章）；
- **路由规划**：选中两个门店后，调用 `OSRM` 画出驾车路线（见第 7 章）；
- **详情弹窗富化**：把 `MapPopup` 内容换成带图片、评分、按钮的卡片（见第 6 章）；
- **性能**：门店过万时，`MapClusterLayer` 的聚合天然扛得住，无需改动结构。

14.5 收尾：你已经学到了什么

回顾全书：

1. 第 1-2 章：理解 `mapcn` 的定位并完成安装；
2. 第 3-4 章：渲染第一张地图、加上控件；
3. 第 5-6 章：标记、弹窗、提示框；
4. 第 7-10 章：路线、弧线、`GeoJSON`、聚合；
5. 第 11-12 章：深入实例、主题与自定义底图；
6. 第 13 章：性能与最佳实践；
7. 第 14 章：用一个完整案例落地。

你现在已经具备用 `mapcn` 构建生产级地图应用的能力。

本章小结

- 门店地图 = `MapClusterLayer`（聚合）+ `MapPopup`（详情）+ 列表（侧栏）的联动；
- 列表与地图通过共享 `selected` 状态双向联动；
- 搜索用 `useMemo` 派生 `GeoJSON`，聚合层自动更新；
- 骨架可直接扩展为分类着色、路线规划、富弹窗等；
- 至此，你已掌握从入门到进阶的全部核心能力。

愿你的地图，既好看，又好用。

版权与免责声明

本书《Mapcn 从入门到进阶》由 AI 辅助文档生成流程编撰，内容以 mapcn 项目官方文档为基线整理而成，仅供学习与技术参考之用。

mapcn 基于 MapLibre GL 构建，相关商标与版权归原权利人所有。本书不主张对 mapcn 或 MapLibre 的任何知识产权。

免责声明：尽管本书已尽量对照官方文档核验事实与语法，但软件版本演进迅速，文中个别参数、默认值或行为可能随版本更新而变化。在涉及生产环境的关键决策时，请务必以官方文档与社区的最新结论为准。因使用本书内容所导致的任何直接或间接后果，编撰方不承担责任。

——编撰团队，2026 年 8 月