30 | AI Coding：不可阻挡的“前端工业革命”

月影 · 跟月影学前端智能体开发

你好，我是月影。

前面的课程里，我们以写代码通过 Ling 框架调用 API 和用 Coze 低代码搭建两种方式为主，详细讲解了如何用前端相关的技术栈来开发 AI 智能体。

这一节课，我们要讨论另外一个话题，也是前端工程师，或者说几乎所有的程序员目前都关注和关心的一个话题—— **AI 辅助编程**。

到今天，AI 辅助编程已经不是一个非常新的东西了。早在 2023 年，我就使用 GitHub Copilot 的 VS Code 插件来进行辅助编程。

最开始的时候，我感觉似乎 Copilot 也没什么，因为本身像 VS Code 这样的 IDE 就具备有很强的代码自动补全能力。当时我以为 AI 不过是在这方面进行了一些补充。

但是，当我深入用 Copilot 的方式编写代码之后，它逐渐令我惊讶，也让我越来越觉得这种方式在未来对软件开发底层逻辑将带来巨大的改变。

AI Copilot vs. AutoCompletion

与传统的 AutoCompletion 相比，AI 辅助编程的代码补全，就好比是传统的智能中文输入法和 AI 大模型的区别。AutoCompletion 就像中文输入法一样，利用大数据和项目本身的代码，通过概率和搜索来进行有限的推理，而 AI Coding 则是颠覆性地利用生成式 AI，让代码辅助具备智能，它具有远比传统模式更大的上下文窗口，所以能够极大地提高输出的准确性和连贯性。

AI Copilot 与 AutoCompletion 的功能差异

为了方便你直观对比传统 IDE 跟 AI Copilot 辅助编程方式的区别，我列了一张表格。

功能维度	传统 IDE 自动补全	AI Copilot 辅助编程
补全粒度	单词级、语法片段	整行、函数、类、文件级别
上下文理解范围	当前文件或作用域	跨文件、跨项目，理解更深层次的上下文
自然语言支持	无	支持从注释或描述生成代码
学习能力	静态，依赖预定义规则	动态，能够从用户的代码风格中学习并适应
多语言支持	受限于 IDE 的支持范围	支持多种编程语言，包括但不限于 TypeScript、Python、Go 等

如上表所示，AI 辅助编程与传统 IDE 自动补全相比，能够进行函数、类甚至文件级别的推理和补全。

```
export function detectMimeType(buffer: Buffer): string | null {
  function isWebm(): boolean {
    return header[0] === 0x1A && header[1] === 0x45 && header[2] === 0xDF && header[3] === 0xA3;
  }

  function isMpeg(): boolean {
    return header[0] === 0x00 && header[1] === 0x00 && header[2] === 0x01
        && (header[3] === 0xBA || header[3] === 0xB3);
  }

  function isAvi(): boolean {
    const r   const riff: string   ode(...header.slice(0, 4));
    const a                        de(...header.slice(8, 11));
    return riff === 'RIFF' && avi === 'AVI';
  }

  function isPdf(): boolean {
    return String.fromCharCode(...header.slice(0, 4)) === '%PDF';
  }

  if (isJpeg()) return 'image/jpeg';
```

比如上面的代码，是我在实现 Coze 的插件基础库中，根据文件内容来判断 mimeType 的函数，它在某些特定情况下，通过文件扩展名、URL 和 http header 都获取不到加载文件类型时，通过文件内容的头部来判断文件的 mimeType 类型。

以前的编程方式下，我必须手工查找所有要支持类型的判断方式，然后一个一个地添加。而现在，我只要实现几个 isXXX 方法之后，AI 大模型就能够自动添加和补全剩余的函数。甚至自动推测出下一步你要做什么：

这些能力，是传统的 AutoCompletion 完全做不到的。

除此之外，对于我们前端来说，AI Copilot 具有极强的类型推断能力，它可以通过函数功能的具体实现、注释等信息来反推和完善代码的类型声明。

下面的三段代码展示了 AI Copilot 如何一步一步地完善 ViduResult 的类型声明。

```
interface ViduResult {
  task_id: string;
  state: 'pending' | 'success' | 'failed' | 'timeout';
  creations: { id: string, cover_url: string, url: string, watermarked_url: string }[];
}

export async function getViduResult(apiKey: string, taskId: string, timeout = 180000): Promise<ViduResult> {
  const headers = {
    'Content-Type': 'application/json',
    Authorization: `Token ${apiKey}`,
  };
  const stateUrl = `https://api.vidu.cn/ent/v2/tasks/${taskId}/creations`;

  const startTime = Date.now();
  let errorMsg = '';
  let creations: { id: string, cover_url: string, url: string, watermarked_url: string }[] = [];
  let state = 'pending';
```

```
interface ViduResult {
  task_id: string;
  state: 'created' | 'queueing' | 'processing' | 'success' | 'failed' | 'timeout';
  creations: {
    id: string;
    cover_url: string;
    url: string;
    watermarked_url: string;
  }[];
  errorMsg: string;
}
```

```typescript
enum ViduResultState {
  created = 'created',
  queueing = 'queueing',
  processing = 'processing',
}

interface ViduResult {
  task_id: string;
  state: 'created' | 'queueing' | 'processing' | 'success' | 'failed' | 'timeout';
  creations: {
    id: string;
    cover_url: string;
    url: string;
    watermarked_url: string;
  }[];
  errorMsg: string;
}
```

大概在 2023 年之前，除非项目明确要求，否则我是几乎不写 TypeScript 代码的。因为那时的我，觉得定义类型比较繁琐，影响了我写代码的速度，虽然类型检查能带来一些好处，但是也付出了代价，牺牲了写代码的效率。

但从 2023 年下半年开始，我毅然决然地拥抱了 TypeScript，原因就是 AI Copilot 让写类型声明这个事情变得极其简单，付出的成本变得微不足道，而类型完备的代码则进一步提高了 AI 推理的确定性，让代码的生成变得更加高效准确，从而更大程度上提升了开发效率。

而到如今，我越来越离不开 TypeScript，TypeScript 已经完全取代了 JavaScript 成为我开发项目的唯一首选编程语言。除了前面说的这个原因外，还有一个非常重要的原因就是类似 Cursor、Trae 这样内置 AI 智能体的编辑器，通过 AI 智能体的 Builder，让创建、配置 TypeScript 项目变得更简单，也让 JavaScript 老代码几乎可以零成本与 TypeScript 彻底打通。

这也是我接下来想要和你分享的话题，那就是除了 AI Copilot 外，AI Builder 正在更加彻底地颠覆前端开发的底层范式。

AI Builder：AI Coding 的“工业革命”

记得我第一次使用 Cursor 的时候，我也仅仅是将它当成**更好用的、不需要插件内置了 AI 能力的 VS Code** 使用，对它的 AI 聊天 Coding 模式并不是很依赖，但是随着使用的深入，我发现当它通过配置工具有了通过对话操作项目的能力之后，一切都变得不一样了。

我相信过去很多的前端同学，尤其是初学者，对前端众多的工程工具、各种配置非常的头疼。有时候大家抱怨说前端技术发展太快，学不过来，并不是指那些底层核心技术，而是指各种层出不穷的工程工具和工程框架。

比如大家都用 Webpack，才刚用熟练，突然又有了新的 Rollup、parcel，没过几天 vite 又出来了，同样是构建工具，有一大堆选择，每种选择都各有利弊，很令人头疼。整个社区也是不断变化，今天一个前端最佳实践，没过几天又变了。各种工具本来就多，再组合起来更加不得了，直到今天，前端项目比较常见的工具组合还有三、四种，比如：

Webpack + Babel

Rollup + esbuilder

Vite + Vue/React

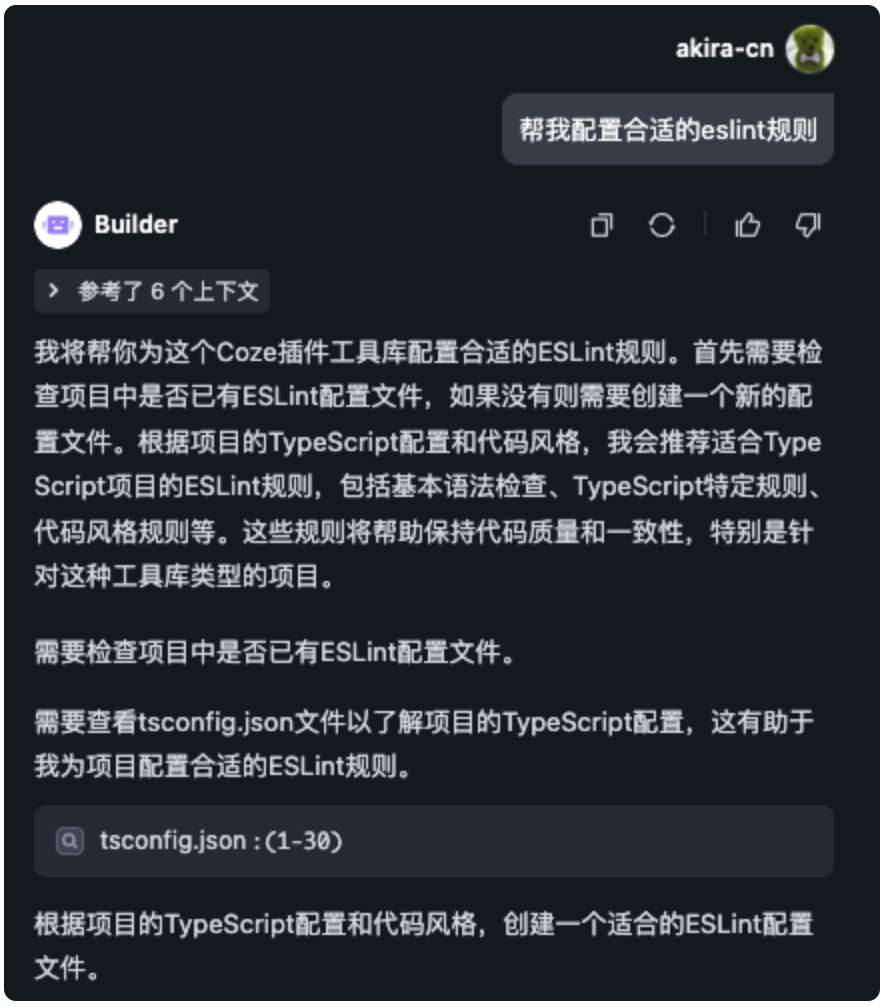
这还不包括要引入 ESLint、CSS 预处理器等等，别的不说，光 CSS 预处理器就有好几种，比如 Sass/SCSS、Less、Stylus、PostCSS，ESLint 也有很多推荐的规则，这些东西搞起来就足够让人头大。进行到这里，我们甚至还没真正开始写一行业务代码，仅仅是搞定了一些前置配置而已。

然而现在有了 AI Builder，一切都改变了。上述繁琐而令人头大的工作，我基本上完全丢给了 AI 处理：

我只需要给 AI 提基础需求，AI 完全能够帮我完成项目的初始配置。

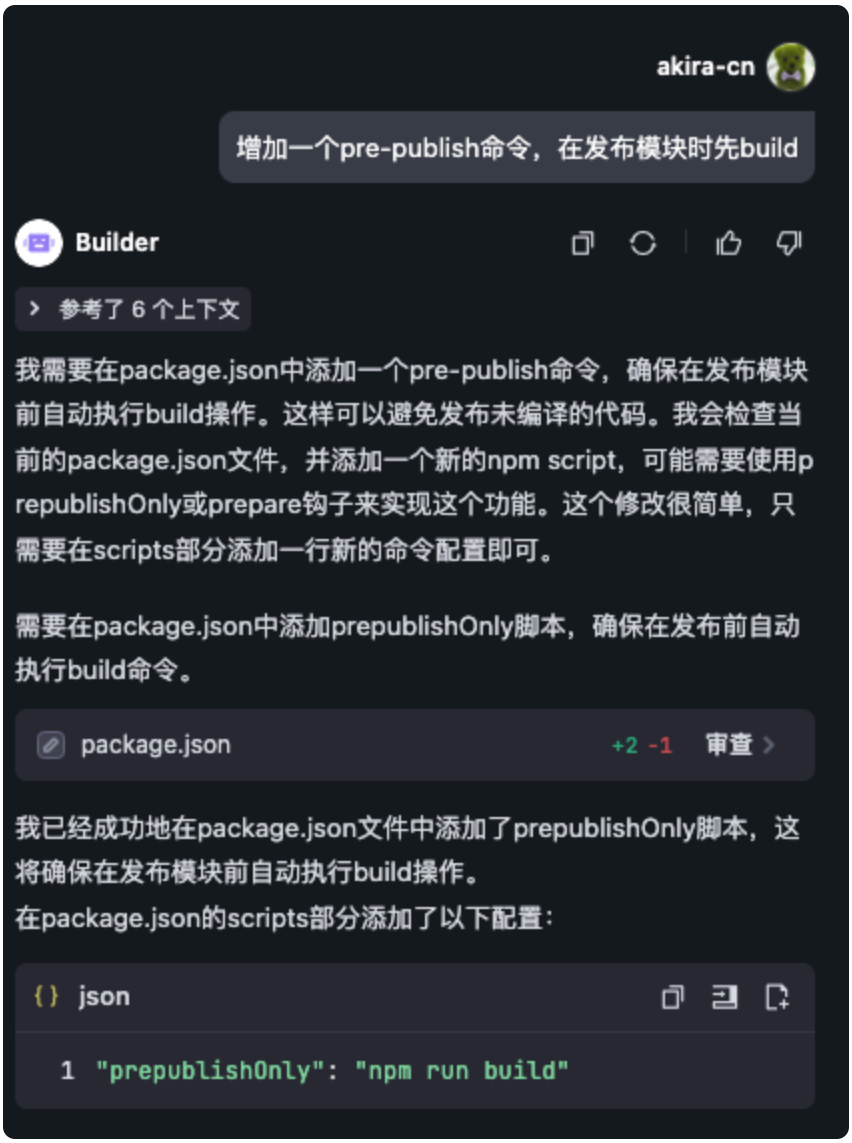
而且在后续，我们还可以随时沟通，及时调整我们的配置：

同样地可以配置代码规范，让 AI 帮我们生成 eslint。

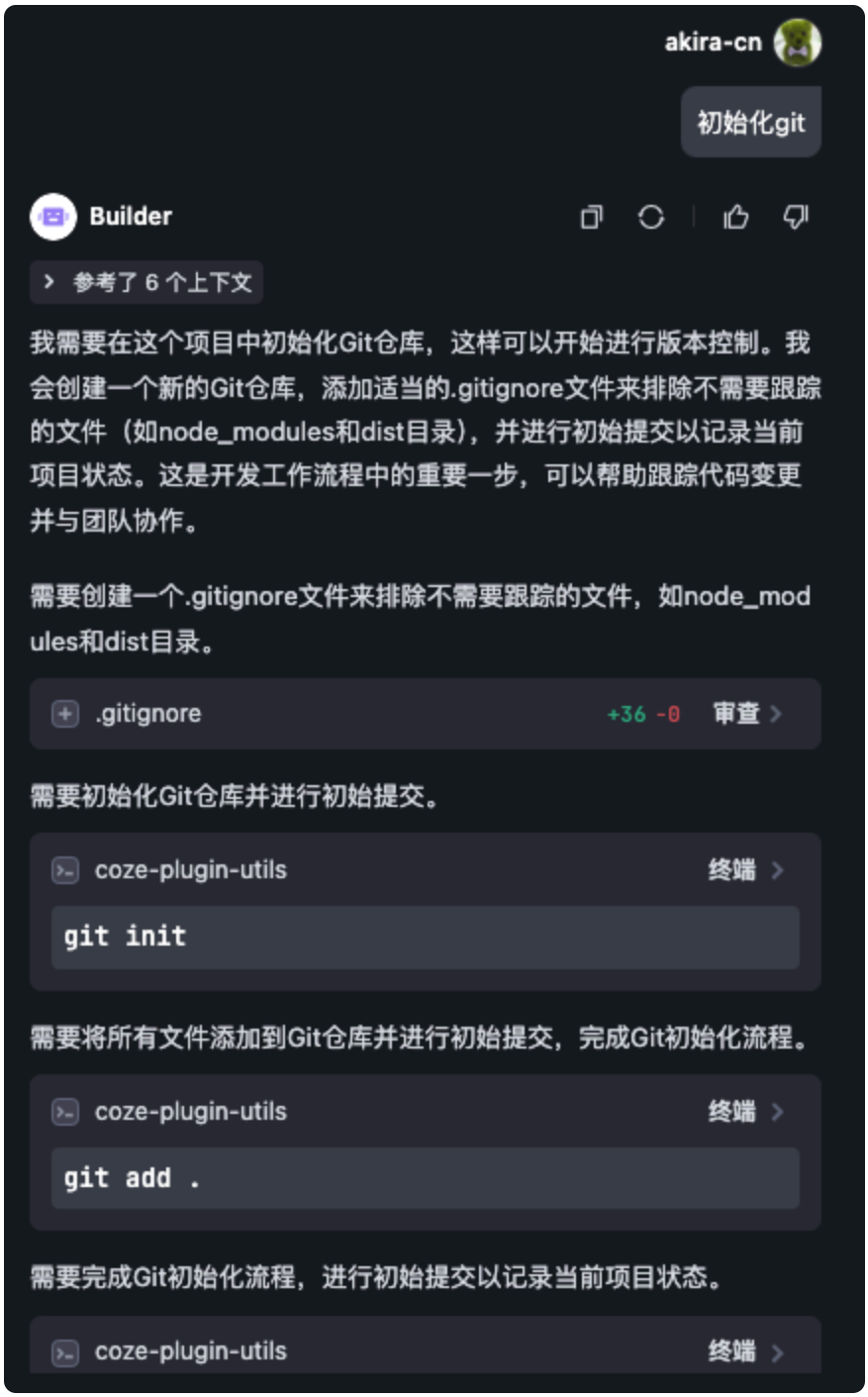


可以沟通改变 eslint 的配置方式，以及指定使用的规则：

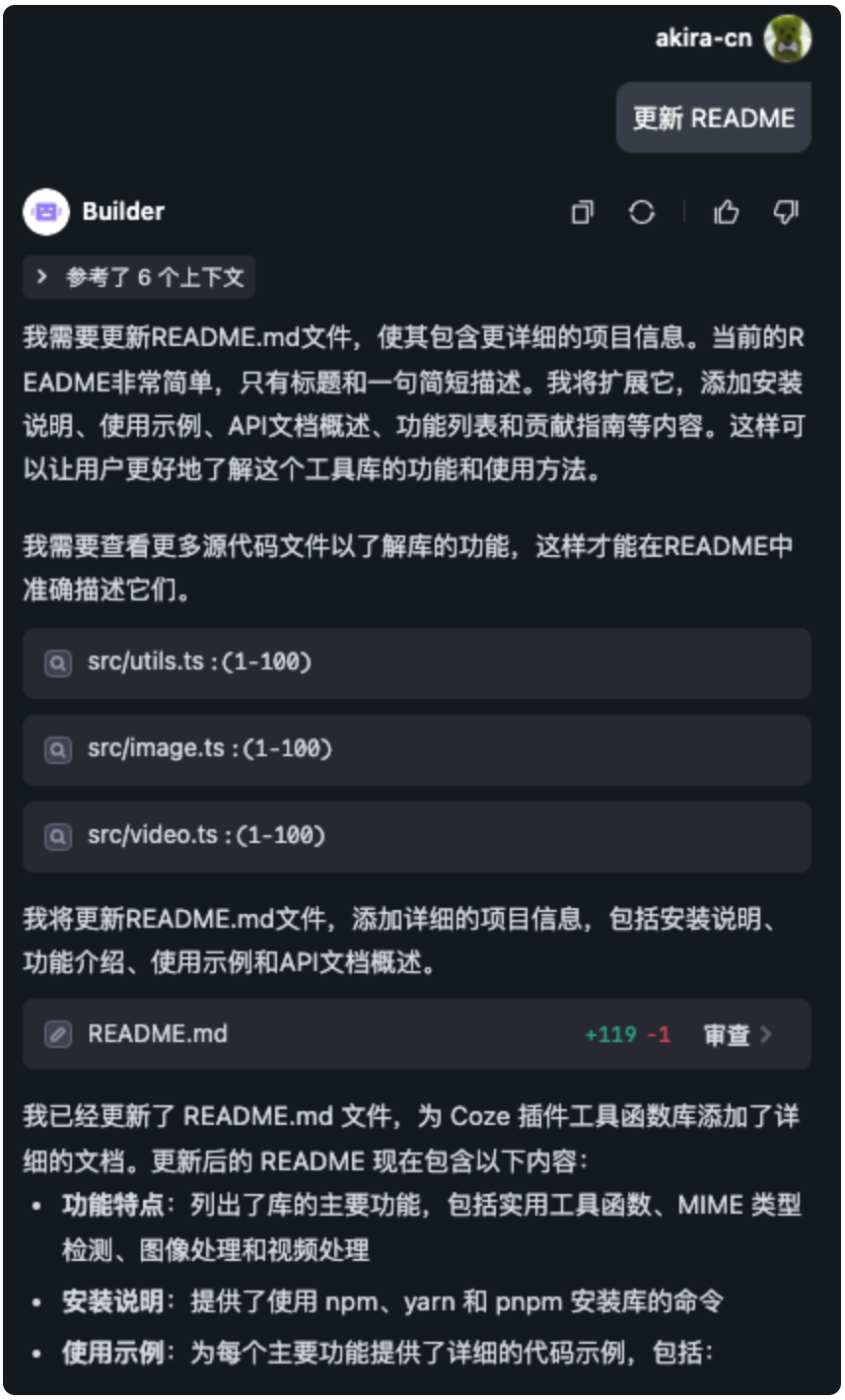
还可以让 AI 帮我们自动添加 pre-publish 命令，保证在发布代码前先完成编译构建：


akira-cn
增加一个pre-publish命令，在发布模块时先build
Builder
参考了 6 个上下文
我需要在package.json中添加一个pre-publish命令，确保在发布模块前自动执行build操作。这样可以避免发布未编译的代码。我会检查当前的package.json文件，并添加一个新的npm script，可能需要使用prepublishOnly或prepare钩子来实现这个功能。这个修改很简单，只需要在scripts部分添加一行新的命令配置即可。
需要在package.json中添加prepublishOnly脚本，确保在发布前自动执行build命令。
package.json
+2 -1
审查
我已经成功地在package.json文件中添加了prepublishOnly脚本，这将确保在发布模块前自动执行build操作。
在package.json的scripts部分添加了以下配置：
{} json
1 "prepublishOnly": "npm run build"

还有初始化 git 仓库：



生成和更新 README：



改变 TypeScript 的编译配置：


akira-cn
帮我改成编译为cjs
Builder
参考了 6 个上下文
Workspace
package.json 8-8
package.json 1-44
README.md 2-6
eslint.config.mjs 1-92
index.ts 1-11
我需要修改项目配置，将编译目标从ESM模块格式改为CommonJS格式。这需要更新tsconfig.json中的模块设置，并可能需要调整package.json中的相关配置。我会检查当前的配置并进行必要的修改，确保项目能够正确编译为CJS格式，同时保持类型声明和导出功能正常工作。
需要修改tsconfig.json文件，将module设置从NodeNext改为CommonJS，以支持CJS格式编译。
tsconfig.json
+2 -2
审查
需要修改package.json文件，移除ESM相关配置并更新导出设置以支持CommonJS格式。

增加开源协议：

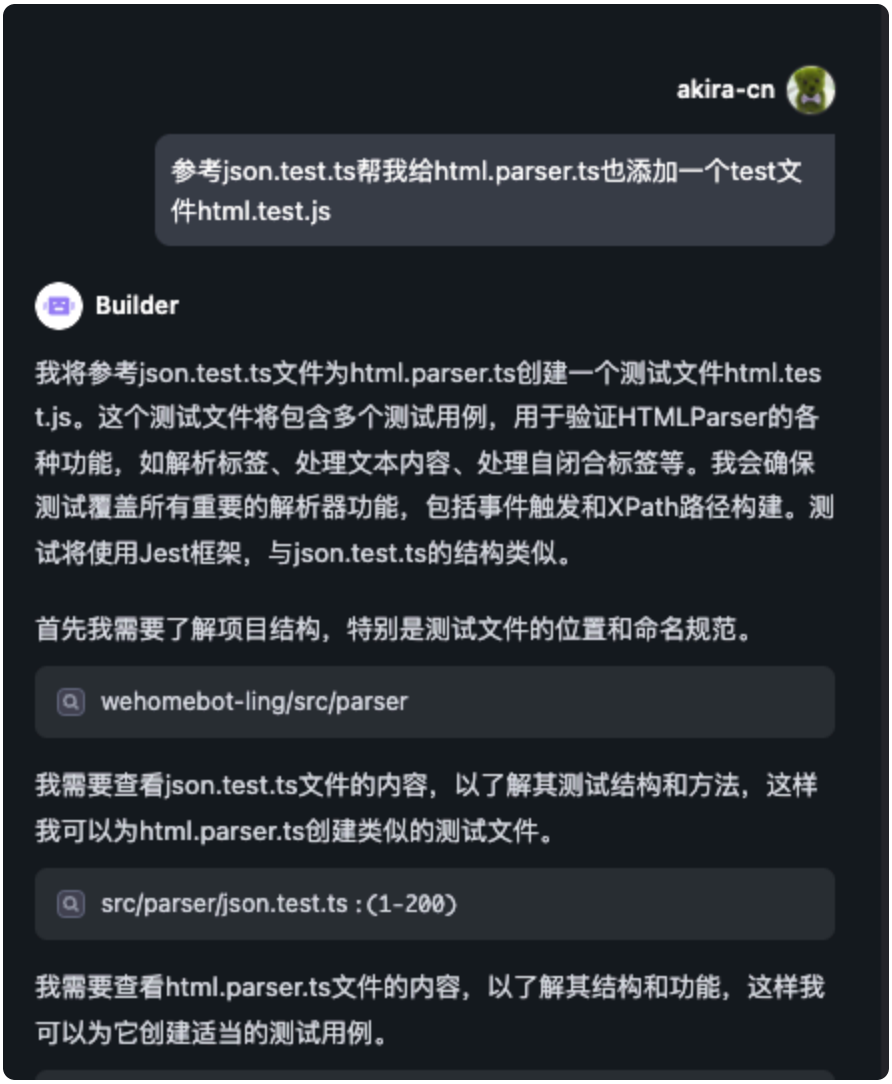
你会发现，过去这些工作几乎每个项目都要接触到，它们和代码逻辑本身毫无关系，但却是耗费精力的“体力活”。现在这些工作交给 AI，显然大大解放了我们的生产力，能够让我们将更多精力投入到业务代码本身，这本质上就是一种**由机器代替人力**的生产力根本变革。

而且，AI Builder 能做的事情远不止这些，它还能做到更多。比如代码的改写和迁移。

在 Ling 框架中，我们最近的升级让它支持了新的 HTML Parser，使得它不仅能流式解析 JSON 文件，还能流式解析 HTML、SVG 和 XML。而这个 Parser 并不是从零开始开发的，而是直接用一个 GitHub 仓库 https://github.com/snappyjs/node-xml-stream 的代码进行改造的。

node-xml-stream 这个仓库的核心代码 https://github.com/snappyjs/node-xml-stream/blob/master/src/parser.js 是一个 XML 的流式解析器，而它是 JavaScript 写的，而且缺少某些对 Ling 来说必要的核心功能。比如解析时不仅要返回标签和属性的信息，还需要返回这个元素的 path，以往这些问题得我们自己解决，比如将它改写成 TypeScript，并且添加这些我们要的功能，这往往容易耗费大量的时间，如果再算上对原版代码的理解和学习的时间，甚至不如我自己从头写起来得快。

但是现在有了 AI，我直接用 Builder 让 AI 把 JavaScript 代码改写成 TypeScript 代码，然后增加计算 path 等需要的功能，甚至参考已有代码给它增加单元测试用例，这些问题 AI 都能在极短的时间内帮我解决，所以大大提升了效率。



实际上 Ling 的 JSON Parser 我从开发、测试到第一版完成，用了大概一天时间，而将这 XML Parser 改写为符合我需求的 HTML Parser，在 Trae AI Builder 的辅助下，只用了一个多小时！

所以我们会直观感受到 AI Builder 对我们开发效率的提升，确实是类似于工业革命机器解放人们的生产力。

另外，我们可以让 AI 去做过去我们不愿意做或不敢做的事情，比如对老项目进行框架改造和升级，过去这种技术债绝大多数工程师和技术团队不愿意去做。

这背后的原因有很多，比如客观上老代码文档缺失、业务复杂、模块耦合严重、缺乏自动化测试、架构设计落后等；而主观上，许多工程师害怕在缺乏完整文档和测试情况下进行改动引发问题，导致系统不稳定，也缺乏必要的重构信心与经验，加上时间与资源优先级的冲突，很多技术债就这样被无限期搁置了，直到马上爆雷才不得不重构，甚至花大力气重新开发。

然而，如今随着 AI 技术的发展，工程师和技术团队完全可以借助 AI 工具辅助进行老项目的框架改造和升级。通过 AI Builder 分析复杂的遗留代码，生成、补全、升级文档和测试用例，拆解重构方法，制定重构方案，辅助工程师进行老代码重构，从而极大降低技术债务，提升系统的可维护性和可扩展性，带来巨大的长期收益。

另外，我们还有机会改变过去传统的工具和组件对框架的依赖性，通过构建面向 AI 的新组件和设计系统（Design System），我们有机会改变传统工具和组件对特定框架的依赖性，转而使用提示词（Prompt）取代硬编码。这种方式使得组件的行为和样式可以通过自然语言进行定义和调整，提高了系统的灵活性和可扩展性。

AI Coding 的未来可能性

在未来，我们前端开发可能不再选择类似 Ant Design、Arco Design 之类的设计系统，以及基于特定框架（如 Vue/React）的组件库，转而选择对 AI 更加友好的秉承 Headless 设计理念的 UI 组件库，例如 shadcn/ui、Headless UI、ZUI 3 等。

甚至更为激进一点，新的 UI 组件连代码都不需要，只描述可配置参数的系统提示词，完全在项目中由 AI 根据框架提示词来生成具体的组件代码，比如一个 UI 组件库中，可能存在以下按钮（Button）组件，“代码”如下：

复制代码

```
你是一个前端 UI 设计师，以下是一个按钮组件的定义，请根据用户需求生成组件提示词：

输出格式如下：
``
组件类型：按钮（Button）
显示文字：{label}
风格：{variant} 样式
图标：{icon}（如果为空则不展示）
尺寸：{size}
点击行为：{action}
```

```
## 组件详细信息：
- 这个按钮的用途、外观风格、尺寸特点，并说明它点击后会触发什么操作。
``
```

然后让 AI Builder 通过这个提示词先生成组件的业务描述，比如当用户输入以下内容：

复制代码

```
这是一个中等尺寸的主按钮，文本为“下载”，左侧带有一个下载图标。它采用主色调设计，醒目易点击。当用
```

最终 AI 生成如下内容：

复制代码

```
组件类型：按钮（Button）
显示文字：下载
风格：primary 样式
图标：download（位于左侧）
尺寸：medium
点击行为：开始下载文件

## 组件详细信息：
- 这个按钮用于执行“下载文件”的操作，位于界面中显著位置。
- 它采用 primary 主色调设计，颜色鲜明，视觉上具有较强的点击引导性。
- 尺寸为中等（medium），适用于大多数布局中，无论是表单还是工具栏。
- 按钮左侧展示一个“下载”图标，增强用户识别度。
- 点击按钮将触发下载逻辑，开始获取并保存文件。
```

然后，我们再让 AI Builder 进一步根据这个内容生成实际代码，这个代码生成过程里，AI Builder 会自己根据项目的技术栈进行调整，用项目技术栈中的技术方案来生成。例如，当前我们使用的是 React 框架，那么生成的代码可能会是如下：

复制代码

```
import { FaDownload } from 'react-icons/fa'; // 使用 react-icons 中的下载图标
import React from 'react';

const DownloadButton = () => {
  const handleDownload = () => {
    // 模拟下载逻辑
```

```
    console.log('开始下载文件...');
    // 你可以在这里替换成实际的下载逻辑
  };

  return (
    <button
      onClick={handleDownload}
      className="flex items-center px-4 py-2 bg-blue-600 text-white rounded-md ho
    >
      <FaDownload className="mr-2" />
      下载
    </button>
  );
};

export default DownloadButton;
```

而如果是 Vue3 框架，则可能会是下面这样：

复制代码

```
<template>
  <button
    @click="handleDownload"
    class="flex items-center px-4 py-2 bg-blue-600 text-white rounded-md hover:bg
  >
    <FaDownload class="mr-2" />
    下载
  </button>
</template>

<script setup>
import { FaDownload } from 'react-icons/fa'

const handleDownload = () => {
  // 模拟下载逻辑
  console.log('开始下载文件...')
  // 这里可以改为真实下载逻辑，如 window.open(url) 或发起接口请求
}
</script>
```

这样我们就实现了完全与框架和技术栈无关的、面向 AI 的组件库，甚至可以基于这个思想，逐步实现完整的新一代设计系统。

当然这些都只是初步设想，前端结合 AI Coding 的未来具体怎样，让我们拭目以待。

要点总结

在这一节，我们主要介绍了 AI Coding 工具的发展以及它为前端带来的变化。其中 AI Copilot 在前端代码生成方面已经能极大提升前端工程师的工作效率，同时在它的辅助下，让写 TypeScript 的成本也大大降低了。而更强大的 AI Coding，不仅能够通过辅助完成项目构建、代码规范、单元测试、文档管理和代码发布管理极大解放我们的生产力，还促使我们去审视过去的开发范式，寻找新的面向 AI 的未来开发范式和最佳实践。

历史的车轮滚滚向前，AI 带来变化已不可阻挡，学习 AI、拥抱 AI 固然让我们的工程工具、开发模式和底层范式发生翻天覆地的变化，促使我们离开过去的舒适区，但它带来的改变，显然是好处远大于不确定性的，因为它毫无疑问地解放了生产力，让我们更聚焦于程序设计的本质，让优秀的工程师变得更加优秀。

这一讲我们主要围绕的还是业务代码周边的东西，并未让 AI 实际参与到业务代码的开发，在后续的课程中，我们将通过实际例子，进一步了解如何用 Trae Builder 更加深入地介入实际项目开发，共同探讨其中的问题和方法，尝试寻找出面向 AI 编程的最佳实践。

课后练习

安装 Trae 或 Cursor，用一个真实的项目，去实际体验这一节课讲的内容，把你的收获分享到评论区吧。

AI智能总结

1. AI Copilot在前端工程师中备受关注，其功能超越传统IDE自动补全，能进行函数、类甚至文件级别的推理和补全。
2. AI Copilot具有强大的类型推断能力，能通过函数功能的具体实现、注释等信息来反推和完善代码的类型声明，简化了类型声明的编写过程。
3. AI Copilot让写类型声明变得简单，提高了代码生成的准确性和开发效率，使得TypeScript逐渐取代JavaScript成为前端开发的首选语言。
4. AI Builder通过AI智能体的Builder，让创建、配置TypeScript项目变得更简单，也让JavaScript老代码可以零成本与TypeScript打通。

5. AI Builder能够帮助前端开发者完成项目的初始配置，并在后续进行代码规范配置、添加pre-publish命令、初始化git仓库等工作，从而提高开发效率.
6. AI Builder还能辅助进行代码的改写和迁移，大大提升了效率，类似于工业革命机器解放人们的生产力。
7. AI Builder可以辅助进行老项目的框架改造和升级，通过分析复杂的遗留代码，生成、补全、升级文档和测试用例，从而降低技术债务，提升系统的可维护性和可扩展性。
8. AI Builder还有机会改变传统工具和组件对特定框架的依赖性，通过构建面向AI的新组件和设计系统，提高系统的灵活性和可扩展性。
9. 前端开发可能不再选择特定框架的组件库，而是选择对AI更加友好的UI组件库，甚至可能出现只描述可配置参数的系统提示词，由AI根据框架提示词来生成具体的组件代码.

全部留言 (1)

最新　精选

轩爷
2025-07-03 来自北京

在 AI 辅助编程的加持下，有几个点我觉得非常棒：
1、让 AI builder 帮助我们搭建框架，使我们更专注于业务代码的开发；
2、开发时产生的问题，AI 结合上下文给出解决办法；
3、代码补全，不仅是写法的补全，还能给出编写建议，能够提供开发思路；
4、代码审查，给出优化建议；
5、给出需求，让 AI 自动编码；
综上，AI 辅助编程极大提高了生产力，提升了代码质量。对个人来说，有更充裕的时间去学点别的东西。

缺点：
1、对个人能力要求更高，尤其是整合能力；
2、生成代码 1 分钟，调试代码俩小时。我觉得这一点，各个 AI Builder 都在改进，越来越智能。

总之，拥抱变化，AI 时代，大胆一点，就像老师 【AI Coding 的未来可能性】这部分对 UI 组件未来的设想一样，又提供了一种思路，赞👍