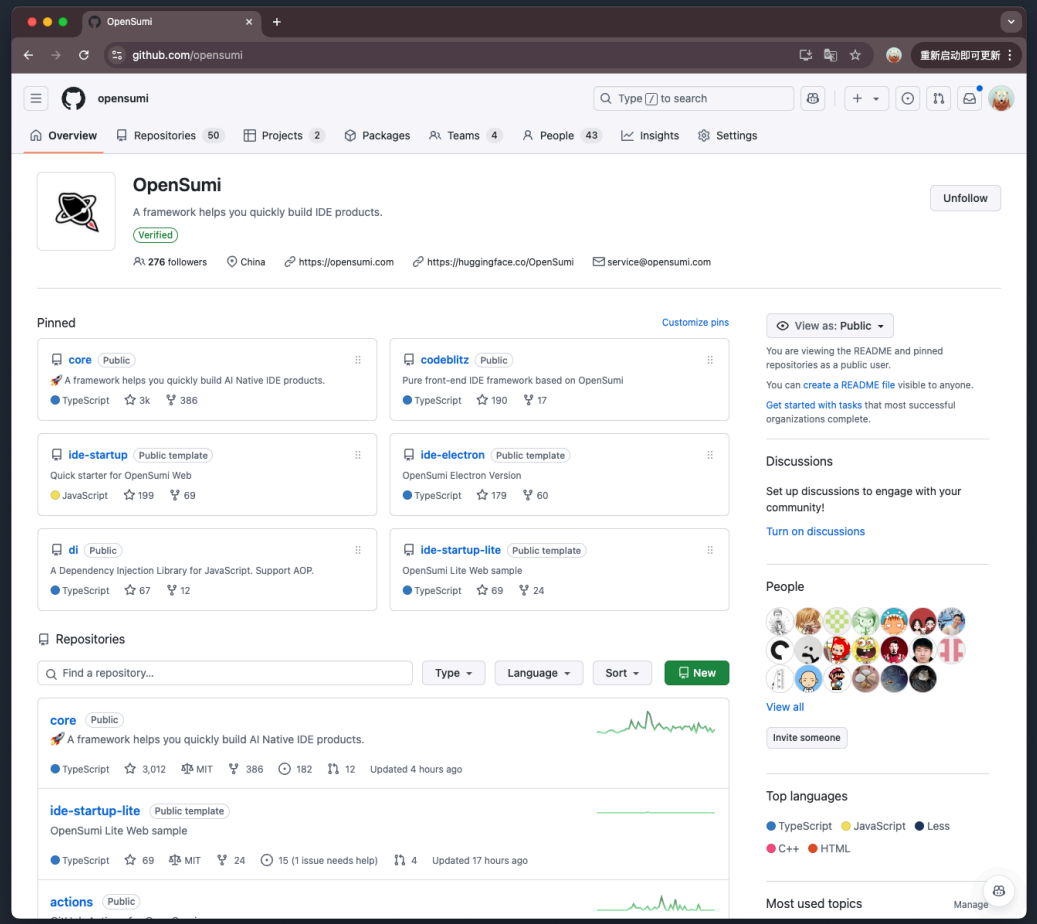
如何打造属于自己的 Cursor

谢俊鸿

FEDAY 2024



OpenSumi IDE 框架的负责人

整体架构设计和核心功能研发

目前主要在探索 AI 和 IDE 结合的玩法并将其实现

https://github.com/opensumi/core

Part1. 智能研发时代的机会和痛点

百家争鸣的智能研发时代

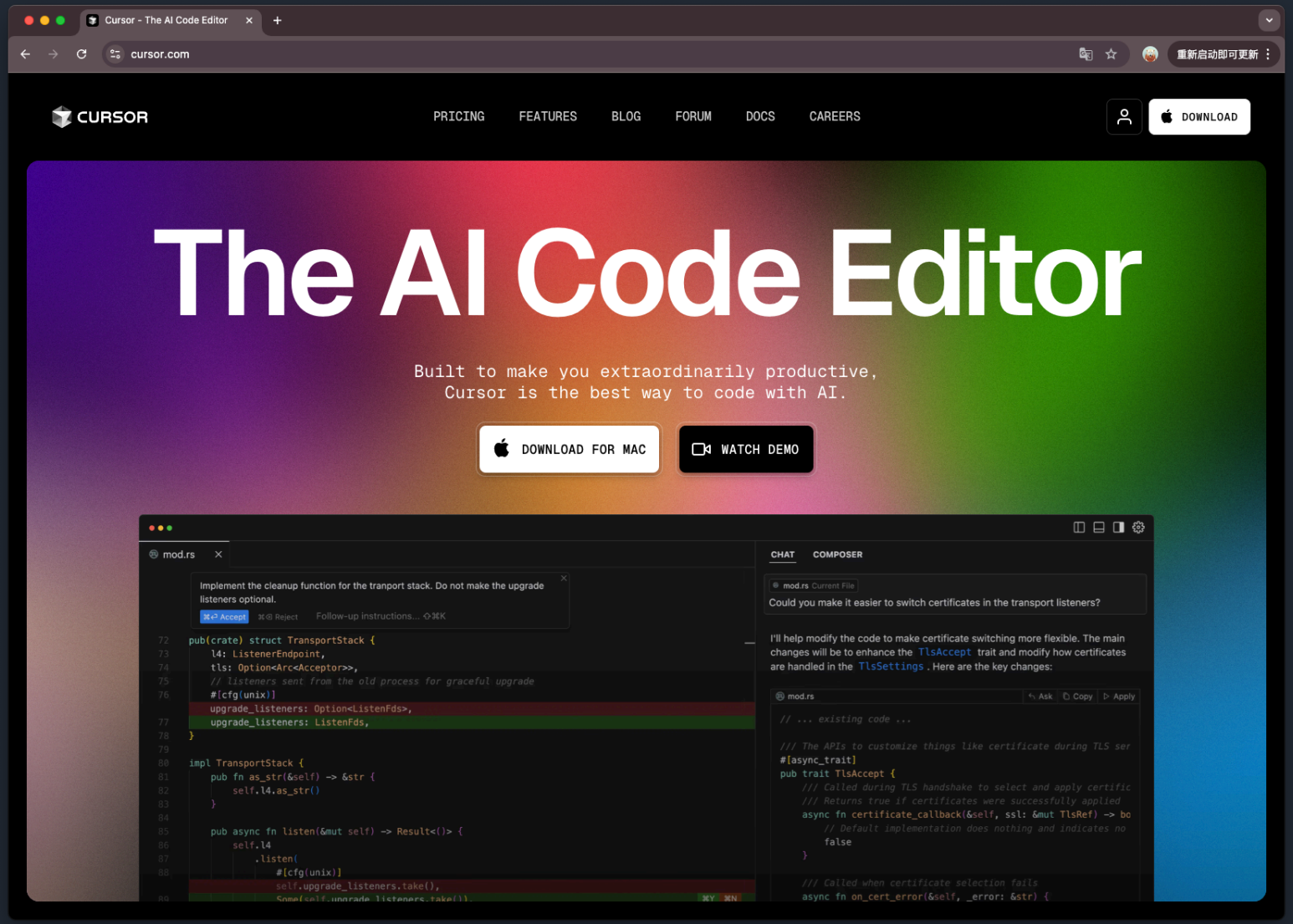
VS Code 给 GitHub Copilot 特供的插件 API

限制了其他厂 Copilot 插件的发展

- 体验割裂
- 难以突破插件形式的天花板

AI Native

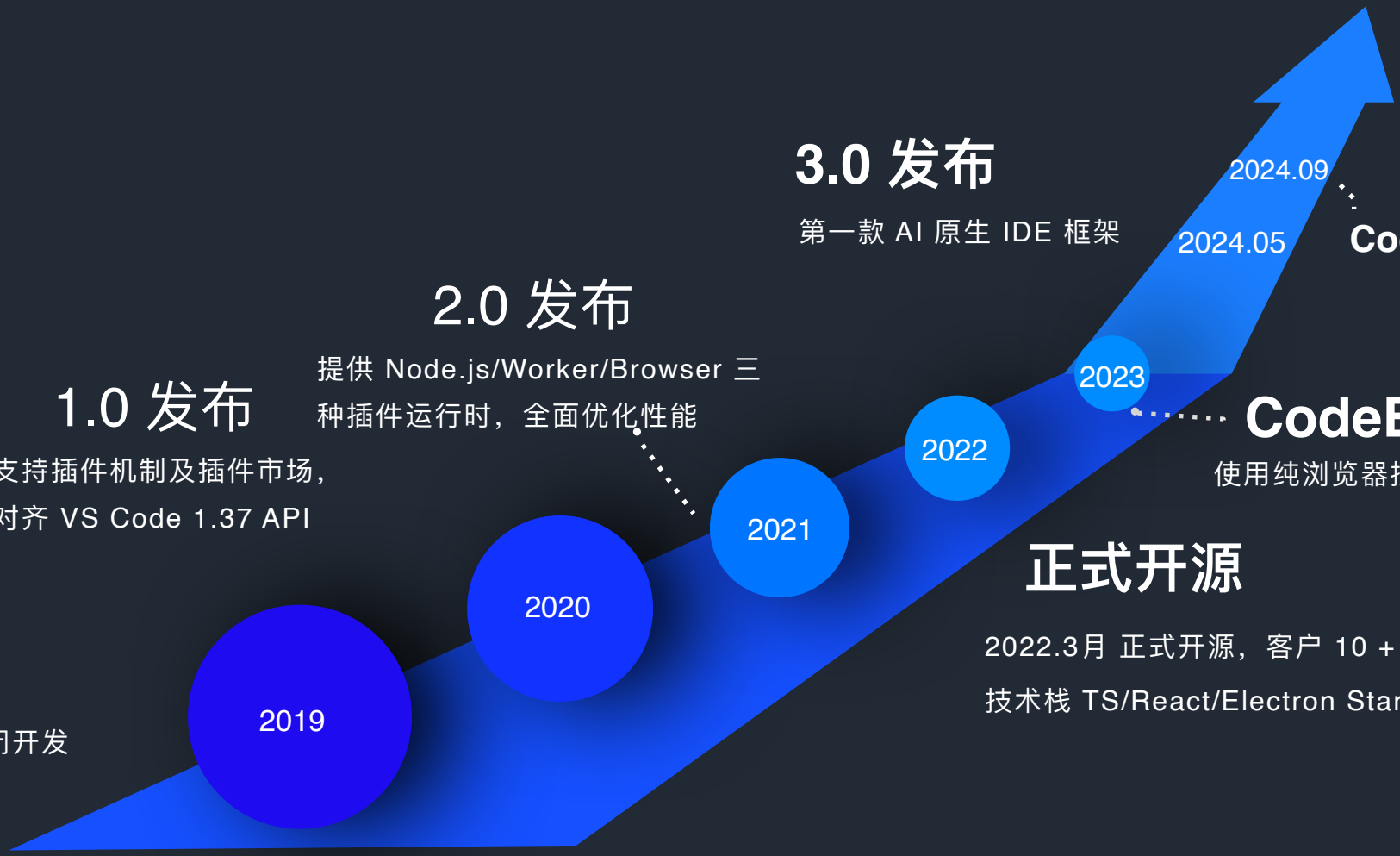
IDE?


Cursor - The AI Code Editor
cursor.com
重新启动即可更新
CURSOR
PRICING
FEATURES
BLOG
FORUM
DOCS
CAREERS
DOWNLOAD
The AI Code Editor
Built to make you extraordinarily productive,
Cursor is the best way to code with AI.
DOWNLOAD FOR MAC
WATCH DEMO
mod.rs
Implement the cleanup function for the tranport stack. Do not make the upgrade listeners optional.
Accept
Reject
Follow-up instructions... ⇧⌘K
pub(crate) struct TransportStack {
l4: ListenerEndpoint,
tls: Option<Arc<Acceptor>>,
// listeners sent from the old process for graceful upgrade
#[cfg(unix)]
upgrade_listeners: Option<ListenFds>,
upgrade_listeners: ListenFds,
}
impl TransportStack {
pub fn as_str(&self) -> &str {
self.l4.as_str()
}
pub async fn listen(&mut self) -> Result<()> {
self.l4
.listen(
#[cfg(unix)]
self.upgrade_listeners.take(),
CHAT
COMPOSER
mod.rs Current File
Could you make it easier to switch certificates in the transport listeners?
I'll help modify the code to make certificate switching more flexible. The main changes will be to enhance the TlsAccept trait and modify how certificates are handled in the TlsSettings . Here are the key changes:
mod.rs
Ask
Copy
Apply
// ... existing code ...
/// The APIs to customize things like certificate during TLS ser
#[async_trait]
pub trait TlsAccept {
/// Called during TLS handshake to select and apply certific
/// Returns true if certificates were successfully applied
async fn certificate_callback(&self, ssl: &mut TlsRef) -> bo
// Default implementation does nothing and indicates no
false
}
/// Called when certificate selection fails
async fn on_cert_error(&self, _error: &str) {

功能项	AI 原生 IDE	AI 插件
仓库级补全	✅	✅
多行补全	✅	❌
智能改写	✅	❌
下一次补全预测	✅	❌
编辑器内通过自然语言生成代码	✅	❌
编辑器内快速问答	✅	❌
智能终端	✅	❌
聊天面板	✅	✅

Part2. OpenSumi AI Native 模块

让 IDE 定制化研发变的简单丝滑



项目组成立

蚂蚁集团X阿里集团齐聚黄龙封闭开发

1.0 发布

支持插件机制及插件市场，
对齐 VS Code 1.37 API

2.0 发布

提供 Node.js/Worker/Browser 三种插件运行时，全面优化性能

正式开源

2022.3月 正式开源，客户 10 +

技术栈 TS/React/Electron Star: 2795 协议：MIT

3.0 发布

第一款 AI 原生 IDE 框架

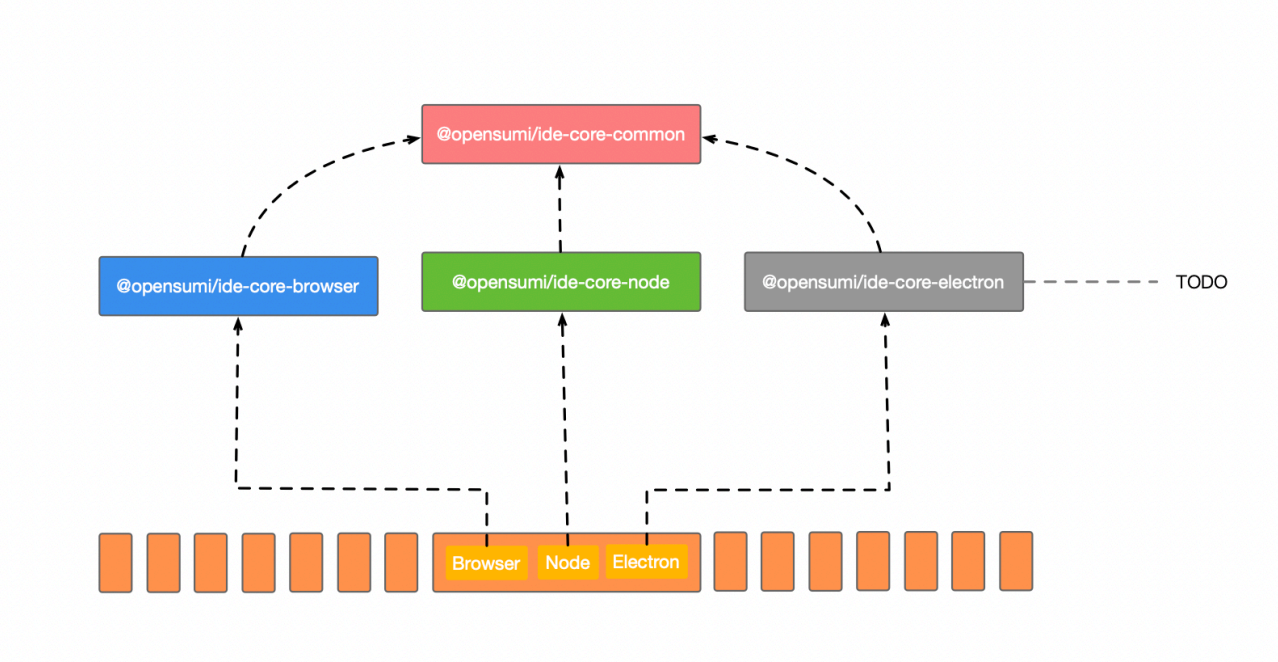
CodeBlitz 开源

使用纯浏览器技术的 IDE 解决方案

CodeBlitz 2.0


支付宝小程序云 Cloud IDE
配置同步
容器初始化
双容器适配
支付宝小程序 IDE
工具栏
模拟器
版本更新
淘宝 O2 IDE
配置同步
容器初始化
空间切换
日志上报
...
生命周期
initialize
onRendered
onStart
onWillStop
onStop
实例统一管理
核心贡献点
命令贡献点
快捷键贡献点
菜单贡献点
生命周期贡献点
配置贡献点
模块能力注册通过贡献点机制管控
核心模块
编辑器
文件树
终端服务
搜索服务
Git 管理
输出面板
插件市场
插件进程
共有 56 个核心模块
基础模块
文件管理
插件管理
终端管理
视图管理
存储管理
配置管理
通信机制
事件机制
国际化
……
core-browser
core-node
core-common
模块通用能力
IDE 组件
badge
notification
recycle-tree
recycle-list
tooltip
dialog
menu
底层依赖
Monaco
LSP
DAP
……

模块的概念



- Browser 层：负责前端视图层的实现
- Node 层：负责后端的实现
- Electron 层：仅在 Electron 客户端中的实现

根据模块本身的职责可以仅存在
browser 层（例如 components 模块）
或者
node 层（例如 file-search 模块）。

模块的概念 – DI 机制

```ts
@Injectable()
export class BasicModule {
  @Autowired(INJECTOR_TOKEN)
  protected injector: Injector;

  providers?: Provider[];
  /**
   * providers only avaiable in electron
   */
  electronProviders?: Provider[];
  /**
   * providers only avaiable in web
   */
  webProviders?: Provider[];
  backServices?: BackService[];
}
```

- injector：管理依赖注入
- providers：存放贡献点

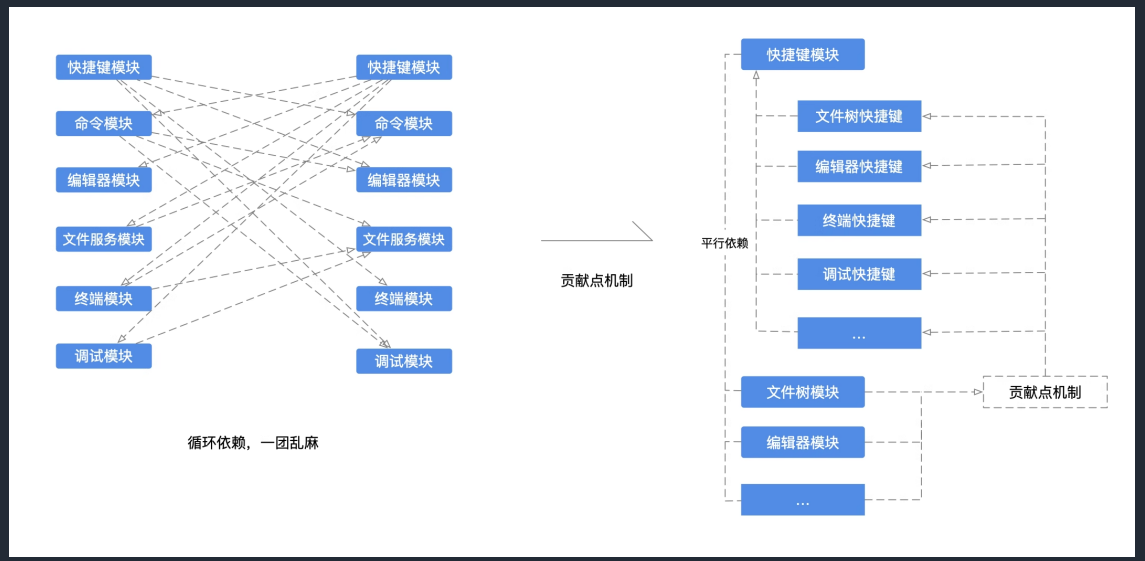
```ts
import { IDialogService } from '@opensumi/ide-overlay';

@Injectable()
class DemoService {
  @Autowired(IDialogService)
  private readonly dialogService: IDialogService;

  run() {
    this.dialogService.info('Hello OpenSumi');
  }
}
```

- 不需要依赖其他模块的具体实现
- 只需要依赖其显示声明的 Token
- 减少模块间的耦合

模块的概念 -贡献点机制

通过贡献点定义，可以让一个能力的完整实现

分散到各个子模块当中。



例如快捷键注册

```ts
import {
  Domain,
  KeybindingContribution,
  KeybindingRegistry
} from '@opensumi/ide-core-browser';

@Domain(KeybindingContribution)
export class DemoContribution implements KeybindingContribution {
  registerKeybindings(keybindings: KeybindingRegistry): void {
    keybindings.registerKeybinding({
      command: quickFileOpen.id,
      keybinding: 'ctrlcmd+p'
    });
  }
}
```

- 通过公共的前端模块引入 KeybindingContribution
- 注册 registerKeybindings 方法

模块的概念 -贡献点机制

- 完整的生命周期管理
- 在特定阶段执行自定义模块的逻辑

```ts
import {
  Domain,
  ClientAppContribution
} from '@opensumi/ide-core-browser';

@Domain(ClientAppContribution)
export class DemoContribution implements ClientAppContribution {
  initialize() {
    // 在初始化阶段执行该函数
  }

  onStart() {
    // 在应用启动阶段执行该函数
  }

  onDidStart() {
    // 在大部分模块启动完成阶段执行该函数
  }

  onWillStop() {
    // 在应用即将关闭前执行该函数，如果返回内容为 true，则关闭的行为将会被中断
  }

  onStop() {
    // 关闭阶段执行
  }
}
```

AI Native 模块的使用

```ts
// browser 层
import { AINativeModule } from '@opensumi/ide-ai-native/lib/browser';
import { DesignModule } from '@opensumi/ide-design/lib/browser';

renderApp({
  modules: [...CommonBrowserModules, DesignModule, AINativeModule]
});

// server 层
import { AINativeModule } from '@opensumi/ide-ai-native/lib/node';

startServer({
  modules: [...CommonNodeModules, AINativeModule]
});
```

1. 在 browser 层和node层都引入 AI Native 模块

```ts
import { AINativeCoreContribution } from '@opensumi/ide-ai-native/lib/browser/types';

@Domain(AINativeCoreContribution)
export class DemoContribution implements AINativeCoreContribution {
  // 在这里注册各种 AI 能力
}
```

2. 引入 AI Native 贡献点，此时就可以注册各种 AI 功能

AI Native 模块的使用

```ts
import { AINativeCoreContribution } from '@opensumi/ide-ai-native/lib/browser/types';

@Domain(AINativeCoreContribution)
export class DemoContribution implements AINativeCoreContribution {
  // 注册代码智能补全
  registerIntelligentCompletionFeature(registry: IIntelligentCompletionsRegistry): void {
    registry.registerInlineCompletionsProvider(async (editor, position, bean, token) => ({
      items: [{ insertText: 'Hello OpenSumi' }],
    }));
  }
}
```

3. 以代码补全为例，注册一个实现代码补全的 API

AI Native 模块的使用

4. 注册的功能我想调模型接口该怎么办?

有两种方式

- 如果你是纯前端的服务，直接 http 请求即可
- 如果考虑安全性，避免 token 信息泄漏，可以实现一个 ai back service 后端服务

在这个服务里你可以调用任意的大模型接口，openai、deepseek、通义等都行

```ts
import {
  CancellationToken,
  ChatReadableStream,
  IAIBackService,
  IAIBackServiceOption,
  IAIBackServiceResponse,
  INodeLogger,
  sleep,
} from '@opensumi/ide-core-node';

@Injectable()
export class AiBackService implements IAIBackService<ReqeustResponse, ChatReadableStream> {
  // 在这里可以跟任何的大模型 API 做通信交互

  // request 可以一次性返回大模型的结果
  async request(
    input: string,
    options: IAIBackServiceOption,
    cancelToken?: CancellationToken
  ) {
    // TODO
  }

  // requestStream 需要返回一个 ChatReadableStream, 以便前端可以流式读取模型的回答
  async requestStream(
    input: string,
    options: IAIBackServiceOption,
    cancelToken?: CancellationToken
  ) {
    const chatReadableStream = new ChatReadableStream();
    cancelToken?.onCancellationRequested(() => {
      chatReadableStream.end();
    });

    setTimeout(() => {
      chatReadableStream.emitData({ kind: 'content', content: 'Hello' });
      await sleep(10);
      chatReadableStream.emitData({ kind: 'content', content: 'OpenSumi!' });
    }, 1000);

    return chatReadableStream;
  }
}
```

AI Native 模块的使用

```ts
import { AIBackSerivceToken } from '@opensumi/ide-core-common';

const injector = new Injector();
injector.addProviders({
  token: AIBackSerivceToken,
  useClass: AiBackService
});
```

5. 通过 DI 机制覆盖服务实现

```ts
import { AIBackSerivcePath } from '@opensumi/ide-core-common';
import { AINativeCoreContribution } from '@opensumi/ide-ai-native/lib/browser/types';

@Domain(AINativeCoreContribution)
export class DemoContribution implements AINativeCoreContribution {
  // 通过 AIBackSerivcePath 拿到注册好的后端服务
  // 此时就能直接 RPC 调用后端服务提供的函数
  @Autowired(AIBackSerivcePath)
  private readonly aiBackService: IAIBackService;

  // 注册代码智能补全
  registerIntelligentCompletionFeature(registry: IIntelligentCompletionsRegistry): void {
    registry.registerInlineCompletionsProvider(async (editor, position, bean, token) => {
      const crossCode = this.getCrossCode(editor);
      const prompt = `Optimize the code:\n\`\`\`\n ${crossCode}\`\`\``;
      // 在这里就可以通过调用后端服务的 request 方法拿到大模型返回的结果
      const result = await this.aiBackService.request(prompt, {}, token);

      // TODO
    });
  }
}
```

6. 最终，你就可以在 browser 层直接调用这个 ai back service 服务的方法进行 api 请求

AI 功能的不同 scope

行

片段

文件

仓库

代码补全增强 – always visible

当内联补全与下拉补全同时出现时
会因为下拉补全的特性而导致内联补全的提示代码消失
OpenSumi 新增了配置开关，允许你的内联补全总是保持显示

代码补全增强 – always visible

当内联补全与下拉补全同时出现时
会因为下拉补全的特性而导致内联补全的提示代码消失
OpenSumi 新增了配置开关，允许你的内联补全总是保持显示

代码补全增强 – always visible

当内联补全与下拉补全同时出现时
会因为下拉补全的特性而导致内联补全的提示代码消失
OpenSumi 新增了配置开关，允许你的内联补全总是保持显示

代码补全增强 – always visible

利用 monaco 本身的 observable 观察者模式，做一层拦截

核心思路

1.拿到 InlineCompletionsController

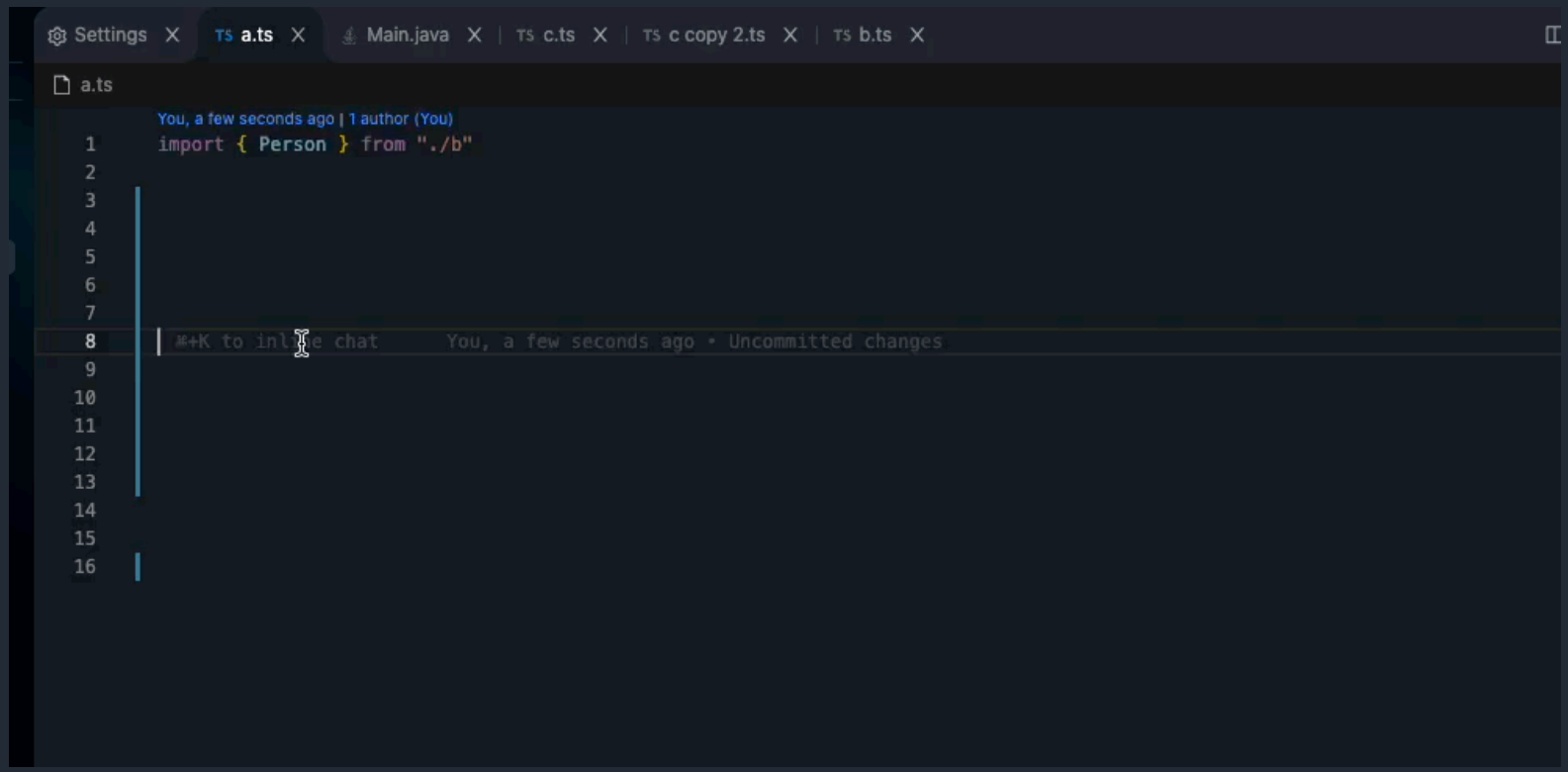
2.从 Controller 当中拿到 suggest widget（这是专门用来处理下拉补全的适配器），里面有一个 selectedItem 这个可观察的值

3.配合 autorun，去观察 selectedItem（也就是 read 函数）

4.如果有值，monaco 会把内联补全给隐藏掉，利用这点，我们可以重新给 selectedItem 赋值为 undefined

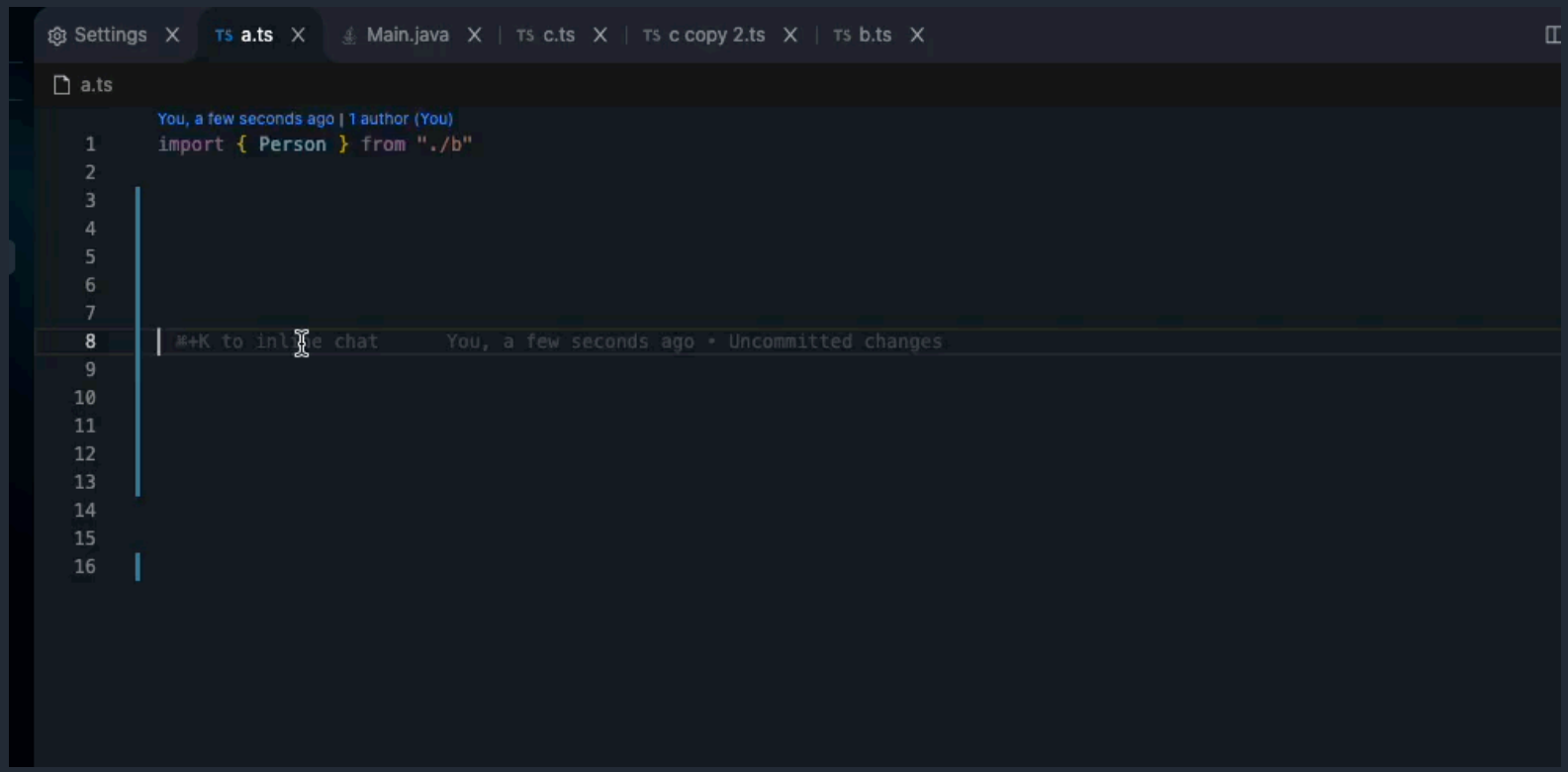
```ts
const inlineCompletionsController = InlineCompletionsController.get(this.monacoEditor);

alwaysVisibleDisposable.add(
  autorun((reader) => {
    /**
     * SuggestWidgetAdaptor 是 inline completions 模块专门用来处理 suggest widget 的适配器
     * 主要控制当下拉补全出现时阴影字符串的显示与隐藏
     * 当 selectedItem 有值的时候（也就是选中下拉补全列表项时），会把原来的 inline completions 本身的阴影字符给屏蔽掉
     * 所以可以利用这点，把 selectedItem 重新置为空即可
     */
    const suggestWidgetAdaptor = inlineCompletionsController['_suggestWidgetAdaptor'] as SuggestWidgetAdaptor;
    const selectedItemObservable = suggestWidgetAdaptor.selectedItem as ObservableValue<
      SuggestItemInfo | undefined
    >;
    const selectedItem = selectedItemObservable.read(reader);

    if (selectedItem) {
      transaction((tx) => {
        selectedItemObservable.set(undefined, tx);
      });
    }
  }),
);
```

试



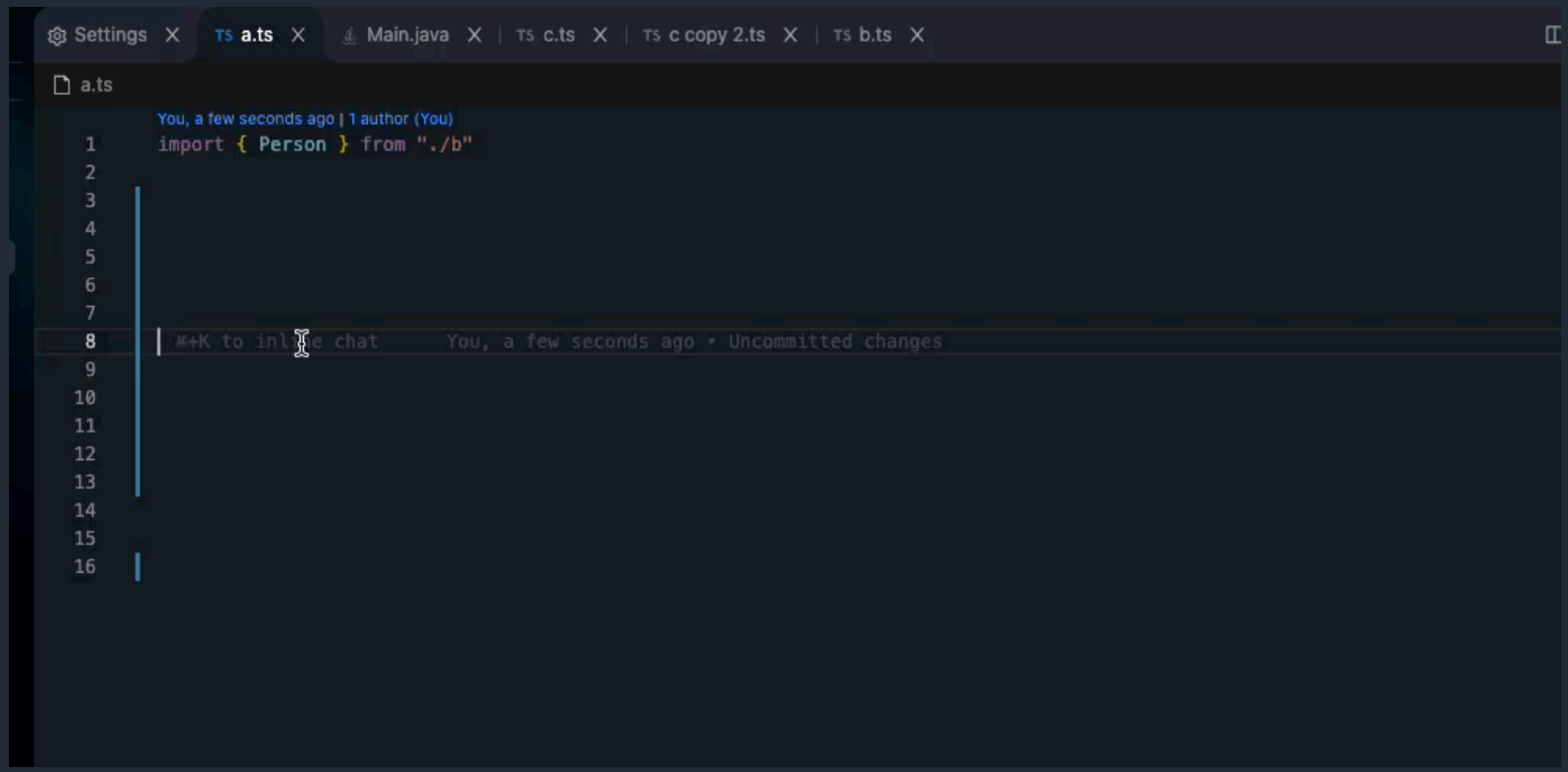
通常代码补全字符都是大一片的灰色字体
OpenSumi 尝试过给内联补全增加代码高亮的特性。

试



通常代码补全字符都是大一片的灰色字体
OpenSumi 尝试过给内联补全增加代码高亮的特性。

试



通常代码补全字符都是大一片的灰色字体
OpenSumi 尝试过给内联补全增加代码高亮的特性。

试

Inline completion 由两部分组成

1. 光标当前行通过 monaco 的 decoration api 渲染
2. 存在换行符时，其余内容通过 monaco 的 zone widget api 渲染

试

高亮的问题如何解决?

monaco 有一个机制叫 tokenization

他是将文本分解成更小的、具有语义意义的单元（tokens）的过程

每一个 token 通常代表着一个词、符号或者是其他语法元素，不同的语言可以有不同的 tokenization 规则。

然后每个 token 包含了其在文本中的位置、类型和颜色等元信息

```ts
// monaco.d.ts

export interface ITextModel {
  // ...
  /**
   * @internal
   */
  readonly tokenization: ITokenizationTextModelPart;
}

/**
 * Provides tokenization related functionality of the text model.
*/
export interface ITokenizationTextModelPart {
  // ...

  /**
   * Get the tokens for the line `lineNumber`.
   * The tokens might be inaccurate. Use `forceTokenization` to ensure accurate tokens.
   * @internal
   */
  getLineTokens(lineNumber: number): LineTokens;
}
```

我们通过构建的手段把 monaco 内部一些不公开的私有属性（带有 @internal 注释）的类型声明都编译出来

试

1. 创建虚拟的 model，并复用当前编辑器的 languageId，使其激活对应语言的高亮系统

2. 把补全内容填充到虚拟 model 中，模拟采纳后的结果。这一步的目的是为了能从虚拟 model 中拿到补全代码的 token 信息

3. 提取 token 信息然后转换成带有高亮色值的 dom 结构，appendChild 到 zone widget 视图中

```
const modelService = StandaloneServices.get(IModelService);
const languageSelection: ILanguageSelection = {
  languageId: textModel.getLanguageId(),
  onDidChange: Event.None
};

const virtualModel = modelService.createModel('', languageSelection);
// 将当前编辑器内容先填充到虚拟 model 中
virtualModel.setValue(textModel.getValue());

/**
 * 再将补全的内容推送到 monaco 的编辑操作中，mock 采纳后的代码
 */
virtualModel.pushEditOperations(null, [replacement], () => null);

return virtualModel;
```

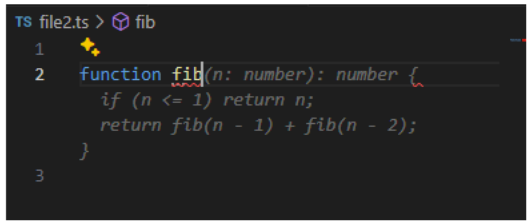
试

Investigate syntax highlighting for inline completions #225400

Closed #227547

hediet opened on Aug 12 · edited by hediet Edits

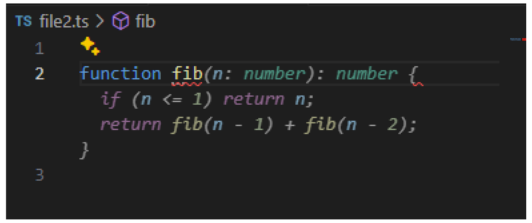
Before:



After:

```
"editor.inlineSuggest.syntaxHighlightingEnabled": true
```

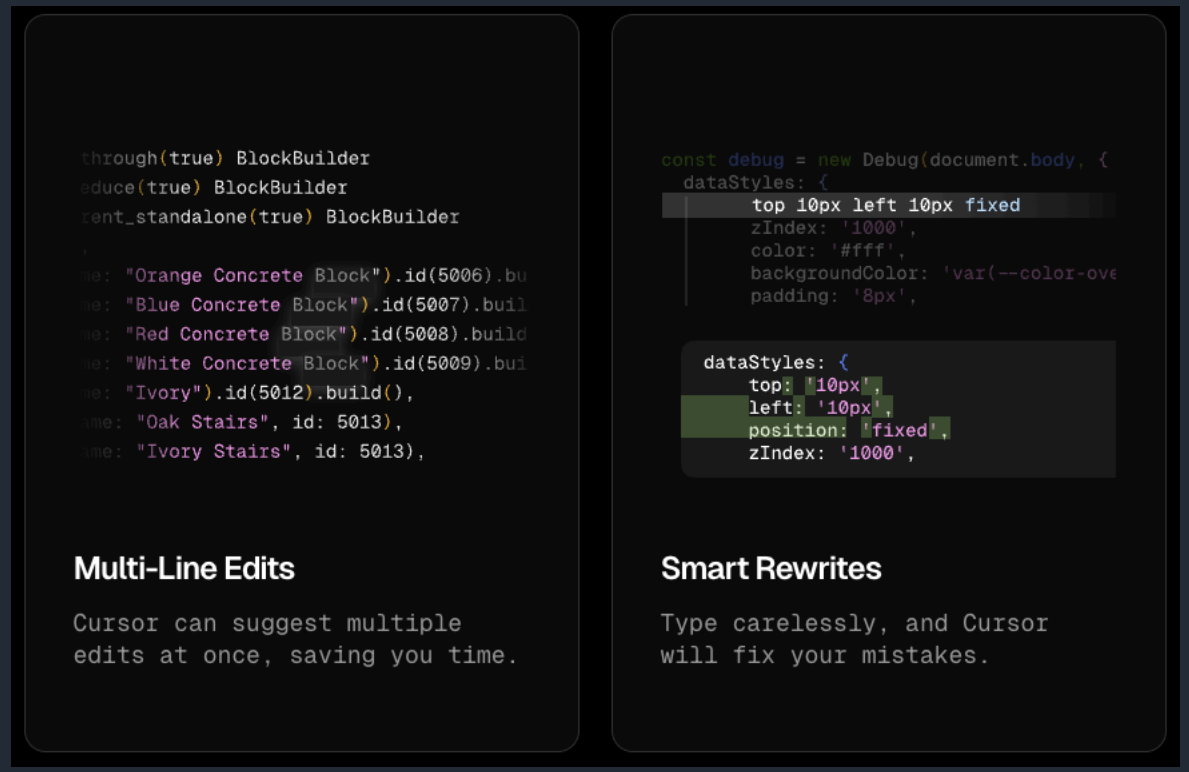
(needs setting change)



🚀 1

😢

写



在 IDE 上的表现形式有两种

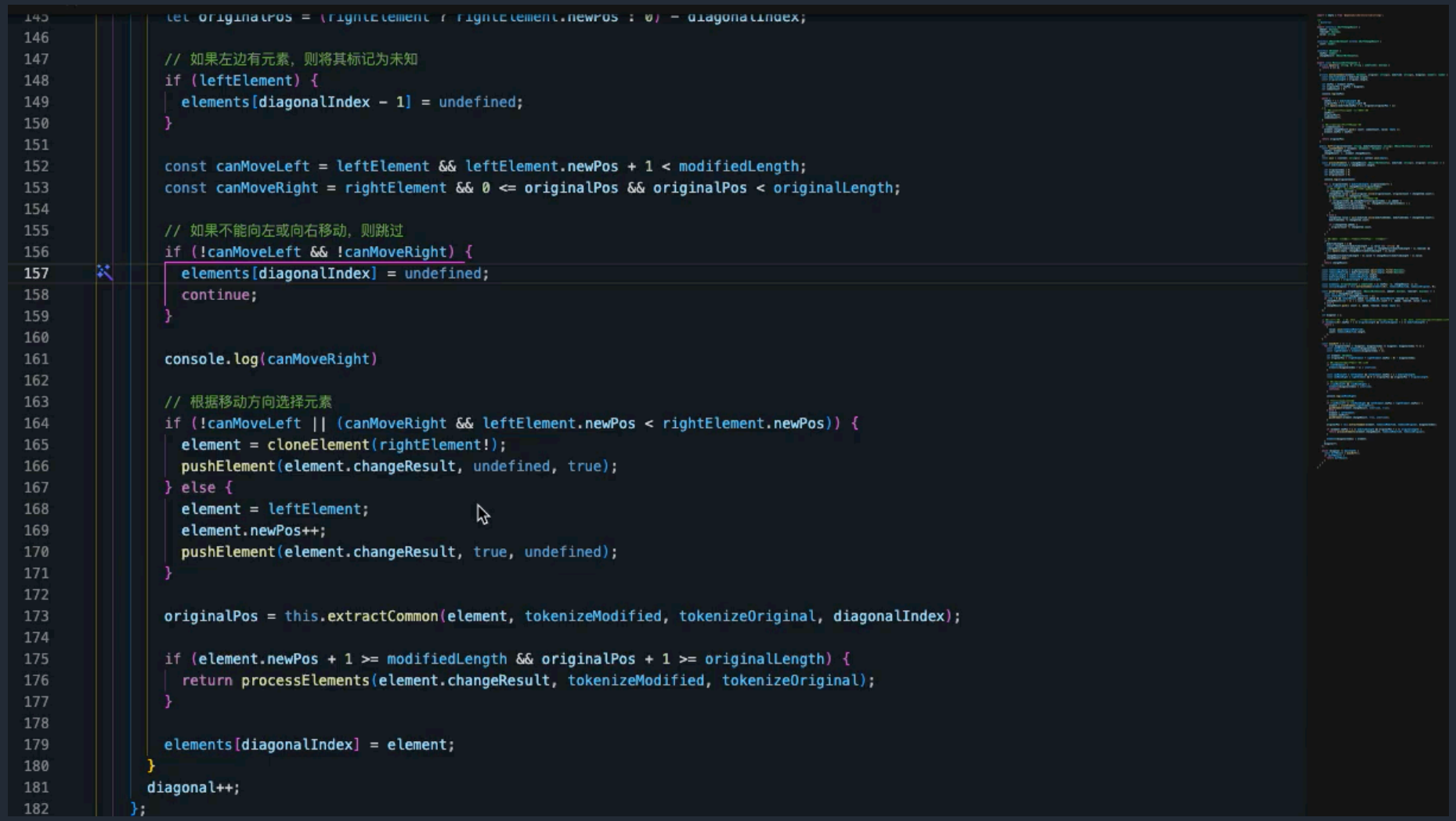
1. 同时在多行代码当中显示内联补全
2. 光标右侧悬浮窗口

写

```
registerIntelligentCompletionFeature(registry: IIntelligentCompletionsRegistry): void {
    registry.registerCodeEditsProvider(async (editor, position, bean, token) => {
      const model = editor.getModel();
      const lineMaxColumn = model.getLineMaxColumn(maxLine) ?? 1;
      // TODO
      return {
          items: [
            {
              // 要插入的代码
              insertText: 'Hello OpenSumi',
              // 要改写的范围，例如从光标开始到往下 3 行结束
              range: Range.fromPositions(
                { lineNumber: position.lineNumber, column: 1 },
                { lineNumber: position.lineNumber + 3, column: lineMaxColumn ?? 1 },
              ),
            },
          ],
        };
    });
  }
```

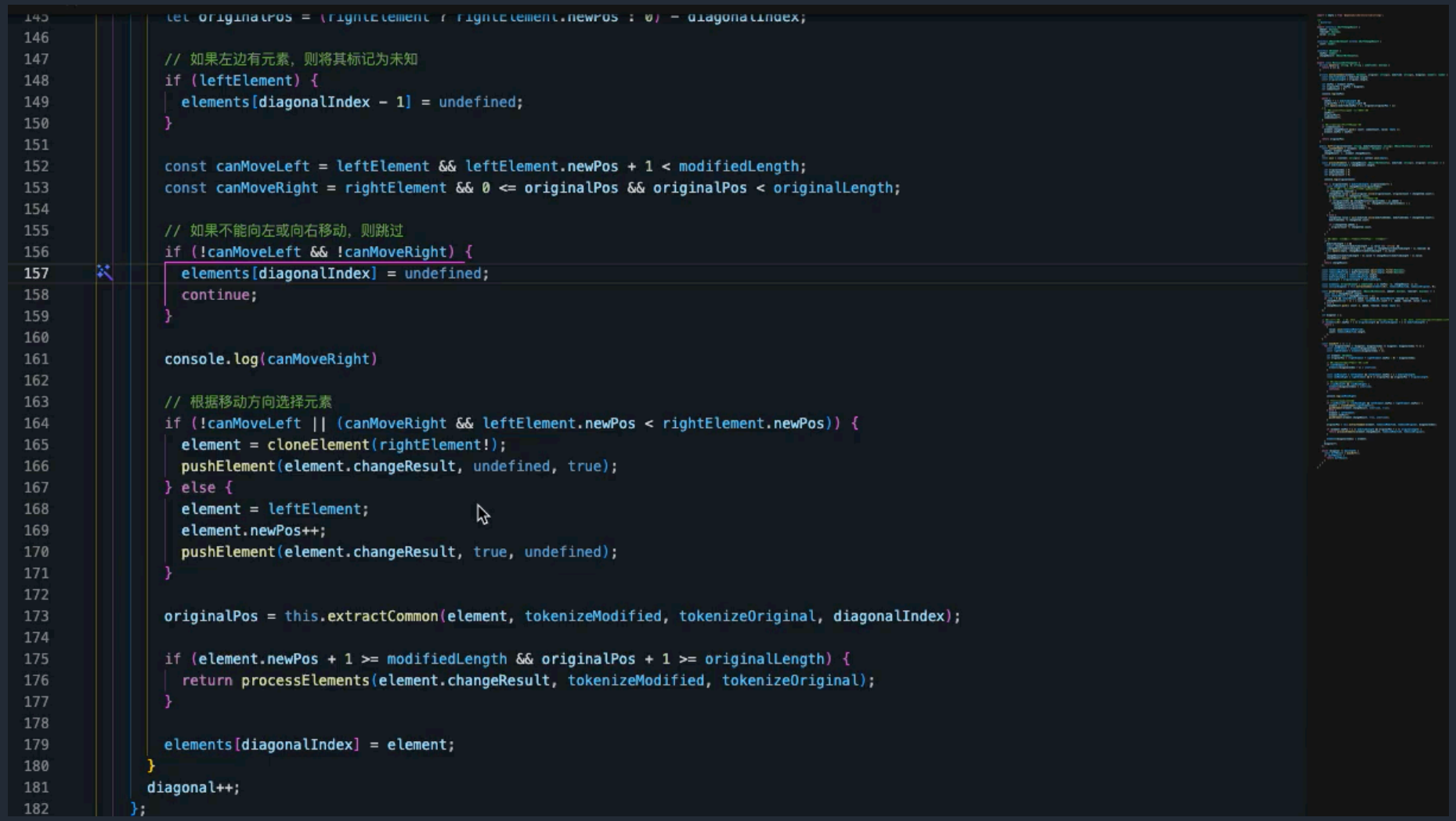
OpenSumi Code Edits API

写



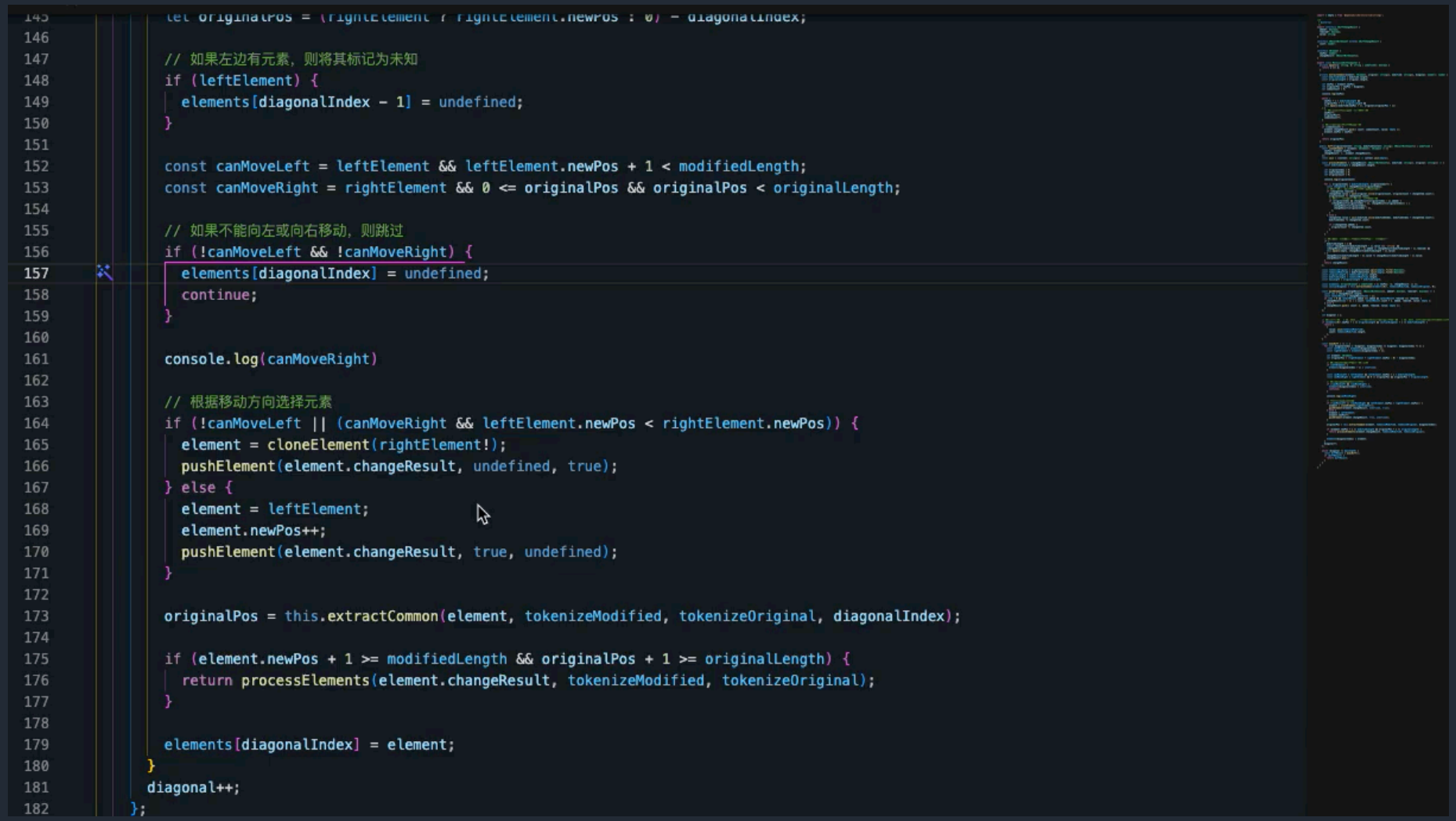
```
let originalPos = (rightElement ? rightElement.newPos : 0) - diagonalIndex;

// 如果左边有元素，则将其标记为未知
if (leftElement) {
  elements[diagonalIndex - 1] = undefined;
}

const canMoveLeft = leftElement && leftElement.newPos + 1 < modifiedLength;
const canMoveRight = rightElement && 0 <= originalPos && originalPos < originalLength;

// 如果不能向左或向右移动，则跳过
if (!canMoveLeft && !canMoveRight) {
  elements[diagonalIndex] = undefined;
  continue;
}

console.log(canMoveRight)

// 根据移动方向选择元素
if (!canMoveLeft || (canMoveRight && leftElement.newPos < rightElement.newPos)) {
  element = cloneElement(rightElement!);
  pushElement(element.changeResult, undefined, true);
} else {
  element = leftElement;
  element.newPos++;
  pushElement(element.changeResult, true, undefined);
}

originalPos = this.extractCommon(element, tokenizeModified, tokenizeOriginal, diagonalIndex);

if (element.newPos + 1 >= modifiedLength && originalPos + 1 >= originalLength) {
  return processElements(element.changeResult, tokenizeModified, tokenizeOriginal);
}

elements[diagonalIndex] = element;
}
diagonal++;
};
```

写



```
let originalPos = (rightElement ? rightElement.newPos : 0) - diagonalIndex;

// 如果左边有元素，则将其标记为未知
if (leftElement) {
  elements[diagonalIndex - 1] = undefined;
}

const canMoveLeft = leftElement && leftElement.newPos + 1 < modifiedLength;
const canMoveRight = rightElement && 0 <= originalPos && originalPos < originalLength;

// 如果不能向左或向右移动，则跳过
if (!canMoveLeft && !canMoveRight) {
  elements[diagonalIndex] = undefined;
  continue;
}

console.log(canMoveRight)

// 根据移动方向选择元素
if (!canMoveLeft || (canMoveRight && leftElement.newPos < rightElement.newPos)) {
  element = cloneElement(rightElement!);
  pushElement(element.changeResult, undefined, true);
} else {
  element = leftElement;
  element.newPos++;
  pushElement(element.changeResult, true, undefined);
}

originalPos = this.extractCommon(element, tokenizeModified, tokenizeOriginal, diagonalIndex);

if (element.newPos + 1 >= modifiedLength && originalPos + 1 >= originalLength) {
  return processElements(element.changeResult, tokenizeModified, tokenizeOriginal);
}

elements[diagonalIndex] = element;
}
diagonal++;
};
```

写



```
let originalPos = (rightElement ? rightElement.newPos : 0) - diagonalIndex;

// 如果左边有元素，则将其标记为未知
if (leftElement) {
  elements[diagonalIndex - 1] = undefined;
}

const canMoveLeft = leftElement && leftElement.newPos + 1 < modifiedLength;
const canMoveRight = rightElement && 0 <= originalPos && originalPos < originalLength;

// 如果不能向左或向右移动，则跳过
if (!canMoveLeft && !canMoveRight) {
  elements[diagonalIndex] = undefined;
  continue;
}

console.log(canMoveRight)

// 根据移动方向选择元素
if (!canMoveLeft || (canMoveRight && leftElement.newPos < rightElement.newPos)) {
  element = cloneElement(rightElement!);
  pushElement(element.changeResult, undefined, true);
} else {
  element = leftElement;
  element.newPos++;
  pushElement(element.changeResult, true, undefined);
}

originalPos = this.extractCommon(element, tokenizeModified, tokenizeOriginal, diagonalIndex);

if (element.newPos + 1 >= modifiedLength && originalPos + 1 >= originalLength) {
  return processElements(element.changeResult, tokenizeModified, tokenizeOriginal);
}

elements[diagonalIndex] = element;
}
diagonal++;
};
```

写

```
165        pushElement(element.changeResult, undefined, true);
166      } else {
167        element = leftElement;
168        element.newPos++;
169        pushElement(element.changeResult, true, undefined);
170      }
171
172      originalPos = this.extractCommon(element, tokenizeModified, tokenizeOriginal, diagonalIndex);
```

```
            element = rightElement;
            element.newPos--;
            pushElement(element.changeResult, false, undefined);
          }
```

实现这个功能有三个核心要素

1. 用哪种 diff 算法?
2. 多行的灰色补全提示如何实现?
3. 出现在光标右侧的悬浮窗口如何实现?

1. 结合 myers 算法提取字符级和单词级的差异
2. monaco 的 decoration api 渲染
3. monaco 的 content widget api 渲染

写

```
165         pushElement(element.changeResult, undefined, true);
166       } else {
167         element = leftElement;
168         element.newPos++;
169         pushElement(element.changeResult, true, undefined);
170       }
171
172       originalPos = this.extractCommon(element, tokenizeModified, tokenizeOriginal, diagonalIndex);
```

```
            element = rightElement;
            element.newPos--;
            pushElement(element.changeResult, false, undefined);
          }
```

可以发现悬浮窗口内是一块需要高亮和部分标绿的代码块

1. 自然用 monaco 渲染
2. 创建虚拟 model，把补全内容填充进去，模拟采纳后的代码
3. 标绿部分用虚拟 model 的 decoration api 渲染
4. 最终把虚拟 model 直接挂载到 content widget 中渲染，并做截取

Inline Chat

既可以选中代码编辑
也能在空白行生成新代码

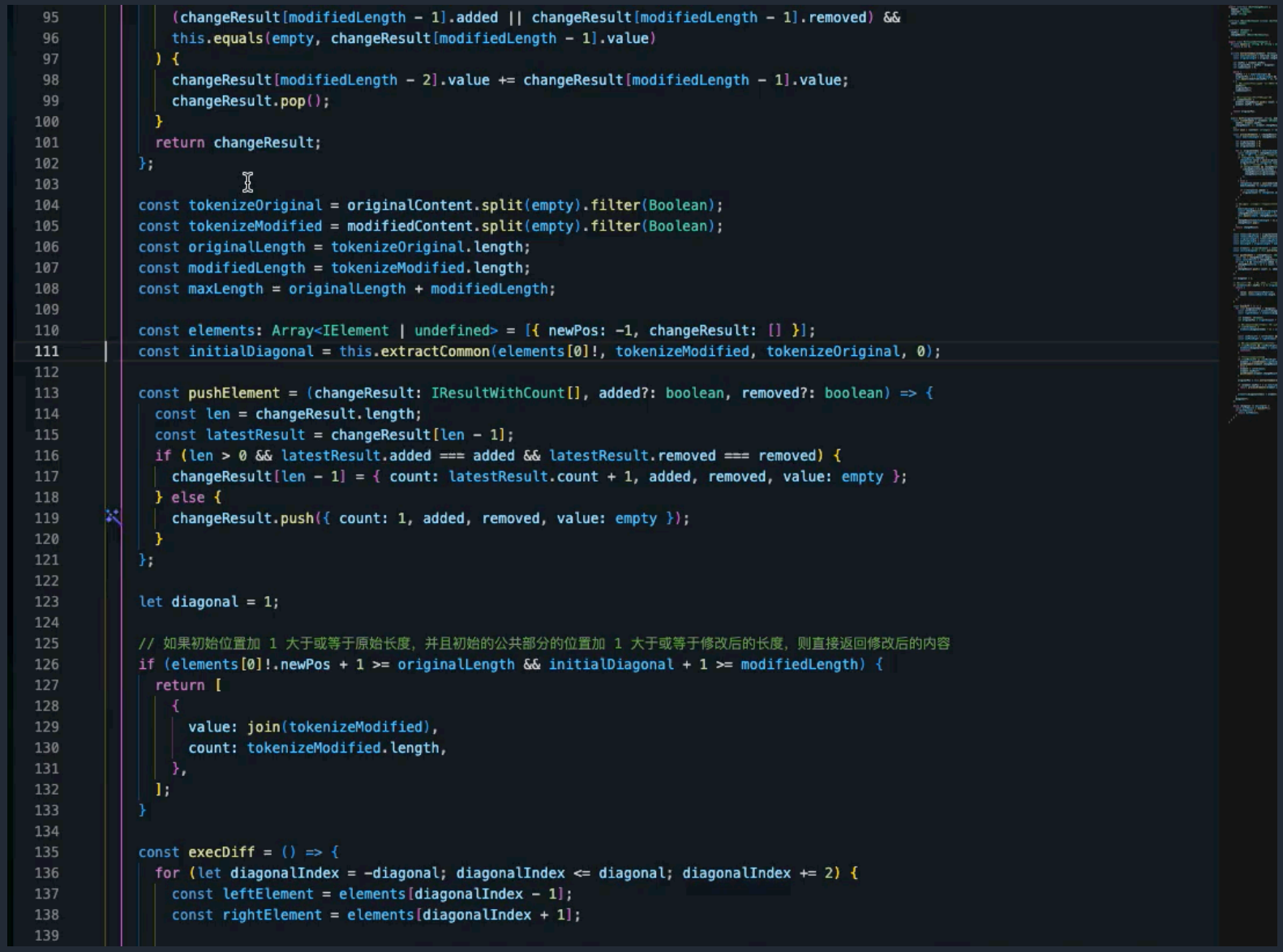
Inline Chat

OpenSumi Inline Chat API

1. 支持注册对话框
2. 支持注册快捷按钮

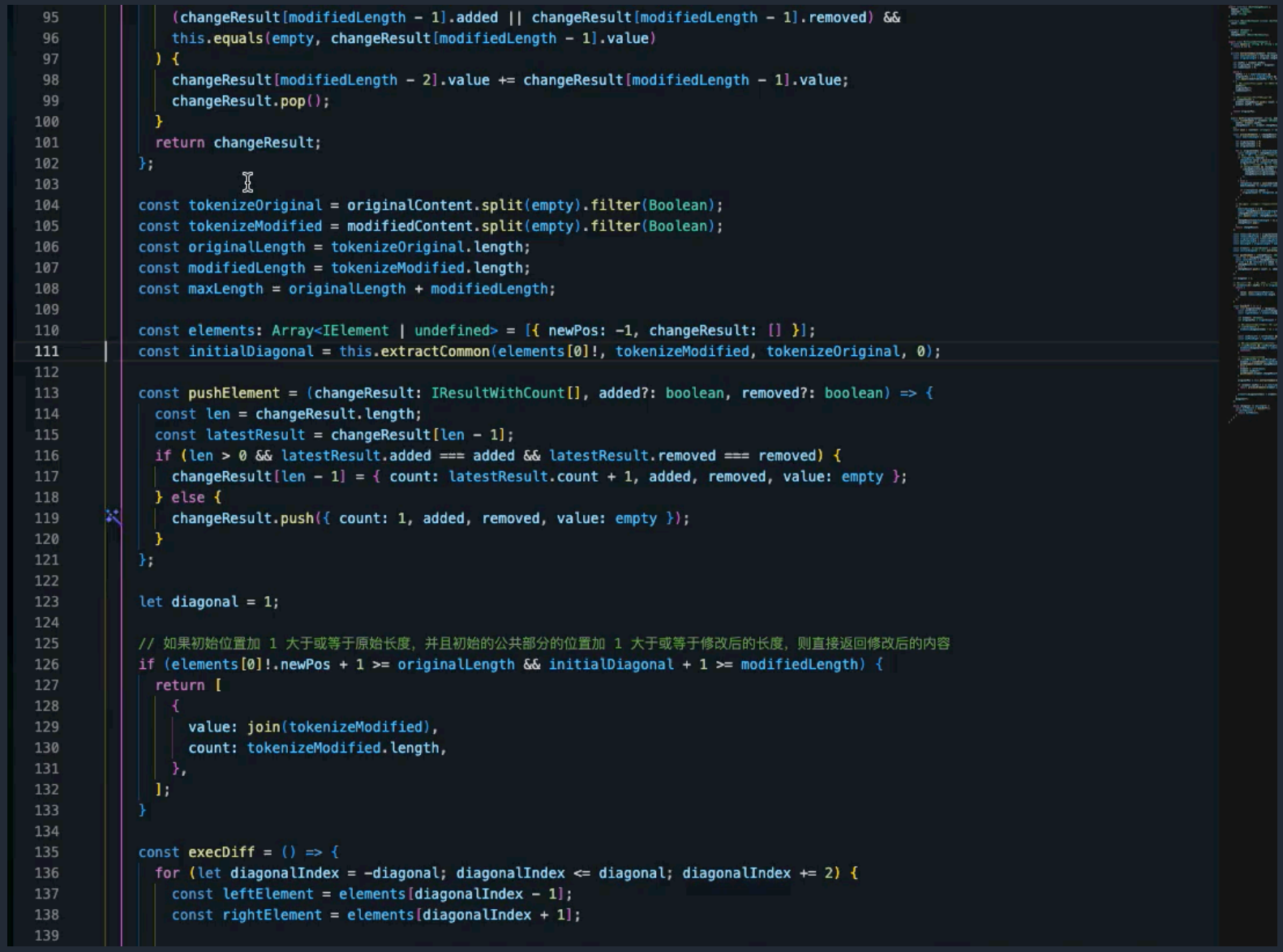
```
registerInlineChatFeature(registry: IInlineChatFeatureRegistry) {
  /**
  * 注册 inline chat 的对话输入框
  */
  registry.registerInteractiveInput(
    {
      providePreviewStrategy: async (editor, value, token) => {
        const crossCode = this.getCrossCode(editor);
        const prompt = `Comment the code: \`\`\`\n ${crossCode}\`\`\`. It is required to return only
the code results without explanation.`;
        const controller = new InlineChatController({ enableCodeblockRender: true });
        const stream = await this.aiBackService.requestStream(prompt, {}, token);
        controller.mountReadable(stream);

        return controller;
      },
    },
  );

  /**
  * 注册 inline chat 的快捷按钮
  */
  registry.registerEditorInlineChat(
    {
      id: 'ai-comments',
      name: 'Comments',
      title: 'add comments',
    },
    {
      providePreviewStrategy: async (editor: ICodeEditor, token) => {
        const crossCode = this.getCrossCode(editor);
        const prompt = `Comment the code: \`\`\`\n ${crossCode}\`\`\`. It is required to return only
the code results without explanation.`;

        const controller = new InlineChatController({ enableCodeblockRender: true });
        const stream = await this.aiBackService.requestStream(prompt, {}, token);
        controller.mountReadable(stream);

        return controller;
      },
    },
  );
}
```

Inline Chat



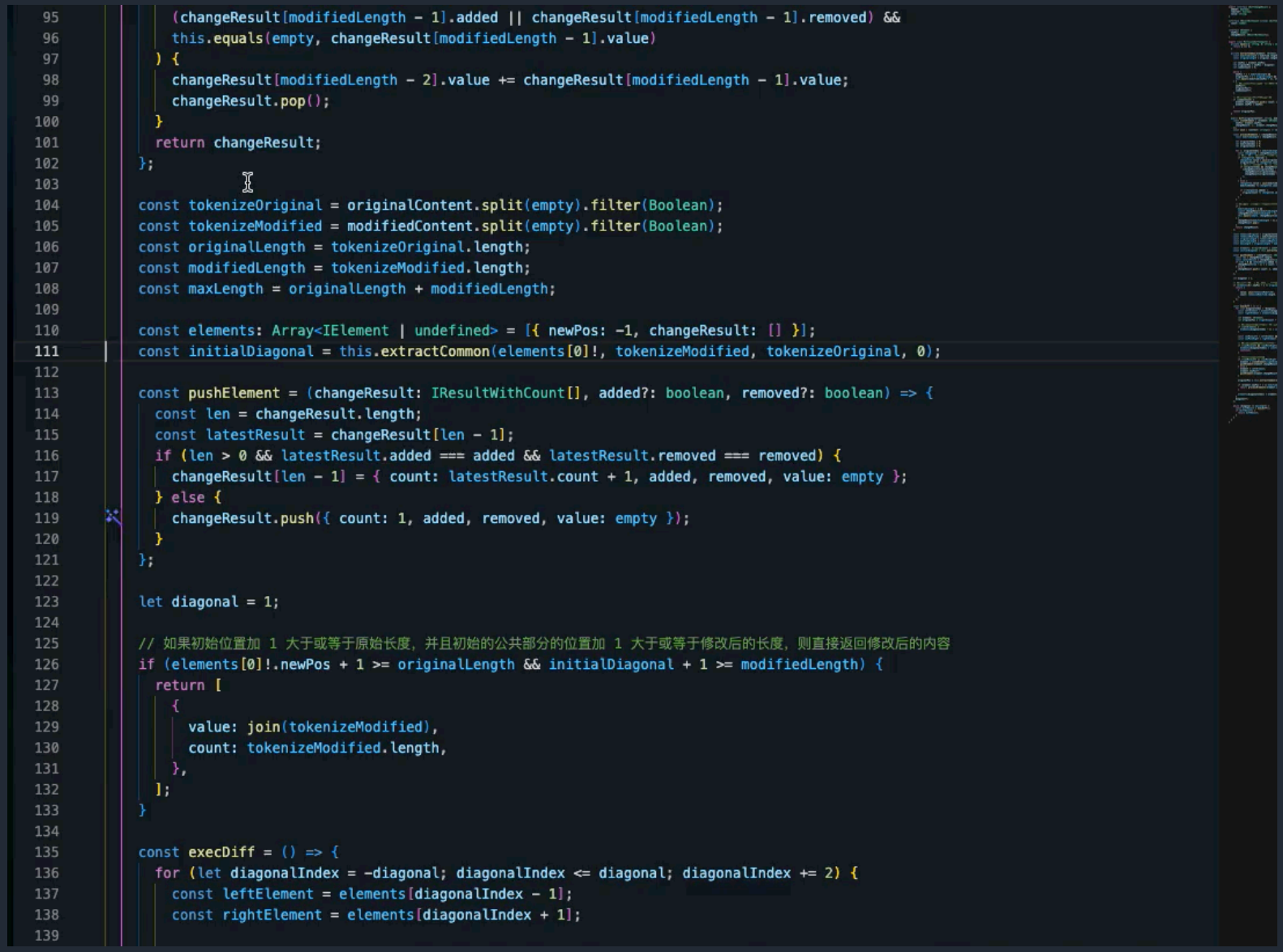
```
    (changeResult[modifiedLength - 1].added || changeResult[modifiedLength - 1].removed) &&
    this.equals(empty, changeResult[modifiedLength - 1].value)
  ) {
    changeResult[modifiedLength - 2].value += changeResult[modifiedLength - 1].value;
    changeResult.pop();
  }
  return changeResult;
};

const tokenizeOriginal = originalContent.split(empty).filter(Boolean);
const tokenizeModified = modifiedContent.split(empty).filter(Boolean);
const originalLength = tokenizeOriginal.length;
const modifiedLength = tokenizeModified.length;
const maxLength = originalLength + modifiedLength;

const elements: Array<IElement | undefined> = [{ newPos: -1, changeResult: [] }];
const initialDiagonal = this.extractCommon(elements[0]!, tokenizeModified, tokenizeOriginal, 0);

const pushElement = (changeResult: IResultWithCount[], added?: boolean, removed?: boolean) => {
  const len = changeResult.length;
  const latestResult = changeResult[len - 1];
  if (len > 0 && latestResult.added === added && latestResult.removed === removed) {
    changeResult[len - 1] = { count: latestResult.count + 1, added, removed, value: empty };
  } else {
    changeResult.push({ count: 1, added, removed, value: empty });
  }
};

let diagonal = 1;

// 如果初始位置加 1 大于或等于原始长度，并且初始的公共部分的位置加 1 大于或等于修改后的长度，则直接返回修改后的内容
if (elements[0]!.newPos + 1 >= originalLength && initialDiagonal + 1 >= modifiedLength) {
  return [
    {
      value: join(tokenizeModified),
      count: tokenizeModified.length,
    },
  ];
}

const execDiff = () => {
  for (let diagonalIndex = -diagonal; diagonalIndex <= diagonal; diagonalIndex += 2) {
    const leftElement = elements[diagonalIndex - 1];
    const rightElement = elements[diagonalIndex + 1];
```

Inline Chat



```
    (changeResult[modifiedLength - 1].added || changeResult[modifiedLength - 1].removed) &&
    this.equals(empty, changeResult[modifiedLength - 1].value)
  ) {
    changeResult[modifiedLength - 2].value += changeResult[modifiedLength - 1].value;
    changeResult.pop();
  }
  return changeResult;
};

const tokenizeOriginal = originalContent.split(empty).filter(Boolean);
const tokenizeModified = modifiedContent.split(empty).filter(Boolean);
const originalLength = tokenizeOriginal.length;
const modifiedLength = tokenizeModified.length;
const maxLength = originalLength + modifiedLength;

const elements: Array<IElement | undefined> = [{ newPos: -1, changeResult: [] }];
const initialDiagonal = this.extractCommon(elements[0]!, tokenizeModified, tokenizeOriginal, 0);

const pushElement = (changeResult: IResultWithCount[], added?: boolean, removed?: boolean) => {
  const len = changeResult.length;
  const latestResult = changeResult[len - 1];
  if (len > 0 && latestResult.added === added && latestResult.removed === removed) {
    changeResult[len - 1] = { count: latestResult.count + 1, added, removed, value: empty };
  } else {
    changeResult.push({ count: 1, added, removed, value: empty });
  }
};

let diagonal = 1;

// 如果初始位置加 1 大于或等于原始长度，并且初始的公共部分的位置加 1 大于或等于修改后的长度，则直接返回修改后的内容
if (elements[0]!.newPos + 1 >= originalLength && initialDiagonal + 1 >= modifiedLength) {
  return [
    {
      value: join(tokenizeModified),
      count: tokenizeModified.length,
    },
  ];
}

const execDiff = () => {
  for (let diagonalIndex = -diagonal; diagonalIndex <= diagonal; diagonalIndex += 2) {
    const leftElement = elements[diagonalIndex - 1];
    const rightElement = elements[diagonalIndex + 1];
```

Inline Chat



```
    (changeResult[modifiedLength - 1].added || changeResult[modifiedLength - 1].removed) &&
    this.equals(empty, changeResult[modifiedLength - 1].value)
  ) {
    changeResult[modifiedLength - 2].value += changeResult[modifiedLength - 1].value;
    changeResult.pop();
  }
  return changeResult;
};

const tokenizeOriginal = originalContent.split(empty).filter(Boolean);
const tokenizeModified = modifiedContent.split(empty).filter(Boolean);
const originalLength = tokenizeOriginal.length;
const modifiedLength = tokenizeModified.length;
const maxLength = originalLength + modifiedLength;

const elements: Array<IElement | undefined> = [{ newPos: -1, changeResult: [] }];
const initialDiagonal = this.extractCommon(elements[0]!, tokenizeModified, tokenizeOriginal, 0);

const pushElement = (changeResult: IResultWithCount[], added?: boolean, removed?: boolean) => {
  const len = changeResult.length;
  const latestResult = changeResult[len - 1];
  if (len > 0 && latestResult.added === added && latestResult.removed === removed) {
    changeResult[len - 1] = { count: latestResult.count + 1, added, removed, value: empty };
  } else {
    changeResult.push({ count: 1, added, removed, value: empty });
  }
};

let diagonal = 1;

// 如果初始位置加 1 大于或等于原始长度，并且初始的公共部分的位置加 1 大于或等于修改后的长度，则直接返回修改后的内容
if (elements[0]!.newPos + 1 >= originalLength && initialDiagonal + 1 >= modifiedLength) {
  return [
    {
      value: join(tokenizeModified),
      count: tokenizeModified.length,
    },
  ];
}

const execDiff = () => {
  for (let diagonalIndex = -diagonal; diagonalIndex <= diagonal; diagonalIndex += 2) {
    const leftElement = elements[diagonalIndex - 1];
    const rightElement = elements[diagonalIndex + 1];
```

Inline Chat

有个难题：如何做到像 Cursor 一样流畅的 inline diff 效果?

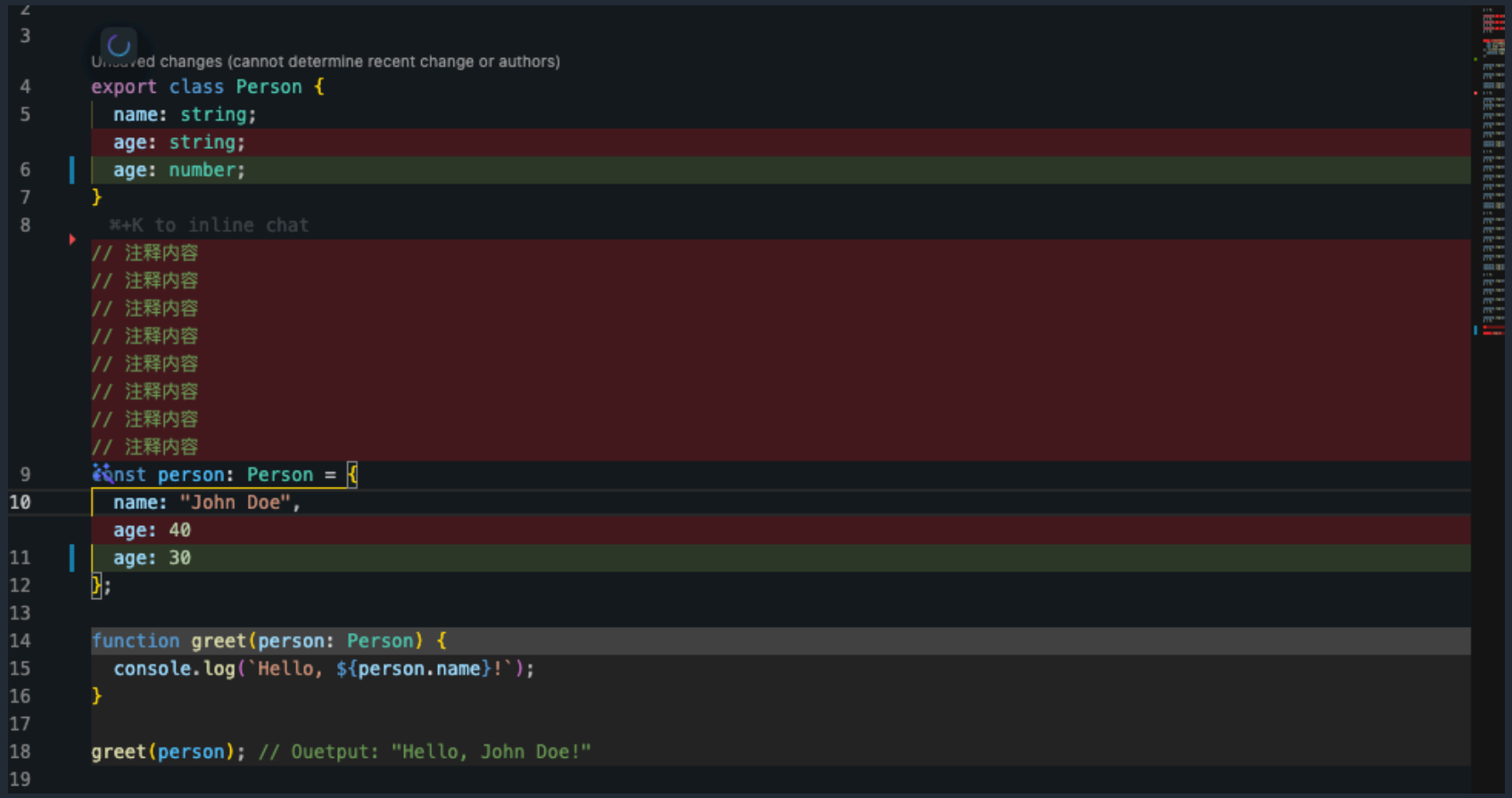
Inline diff 的好处：响应速度快、可以中途停止、体验流畅

问题拆解

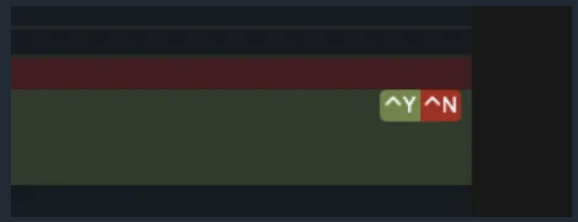
1. 如何找到合适的 diff 算法去做计算?
2. 拿到 diff 结果后如何在 monaco 当中进行渲染?

Inline Chat

1. 红色删除区域
2. 绿色新增区域
3. 深灰色“进度条”
4. 浅灰色“待处理”区域
5. 采纳/弃用 按钮



截取流式过程中的某一帧



Inline Chat

```
/**
 * 临时存储被删除之前的每一行 tokens 信息
 */
this.rawOriginalTextLinesTokens = this.rawOriginalTextLines.map((_, index) => {
  const lineNumber = startPosition.lineNumber + index;
  this.originalModel.tokenization.forceTokenization(lineNumber);
  const lineTokens = this.originalModel.tokenization.getLineTokens(lineNumber);
  return lineTokens;
});
```

红色删除区域

1. 先用 zone widget api 撑开面板
2. 结合 tokenization 机制提前存储被删除代码的 token 信息
3. 流式过程中将 token 信息转成 dom 结构 appendChild 到 zone widget 中

```
const dom = document.createElement('div');
renderLines(
  dom,
  this.editor.getOption(EditorOption.tabIndex),
  // 传递 lineTokens 信息
  this.textLines.map(({ text: content, lineTokens }) => ({
    content,
    decorations: [],
    lineTokens,
  })),
  this.editor.getOptions(),
);

// 把 dom appendChild 到 zone widget 中
this.root.render(<RemovedWidgetComponent dom={dom} />);
```

剩余的其他区域都可以使用 decoration api 进行着色渲染

Inline Chat – Diff 算法

Monaco 自带两种 diff 算法

- LegacyLinesDiffComputer：经典算法 LCS，计算行级差异
- DefaultLinesDiffComputer：Myers 算法，基于图的方式求 LCS，性能更好

Inline Chat – Diff 算法

原代码

```ts
export class Person {
  name: string;
}

// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
const person: Person = {
  name: "John Doe",
  age: 40
};

function greet(person: Person) {
  console.log(`Hello, ${person.name}!`, 66666);
}

greet(person); // Output: "Hello, John Doe!"
```

新代码

```ts
export class Person {
  name: string;
  age: number;
}

const person: Person = {
  name: "John Doe",
  age: 30
```

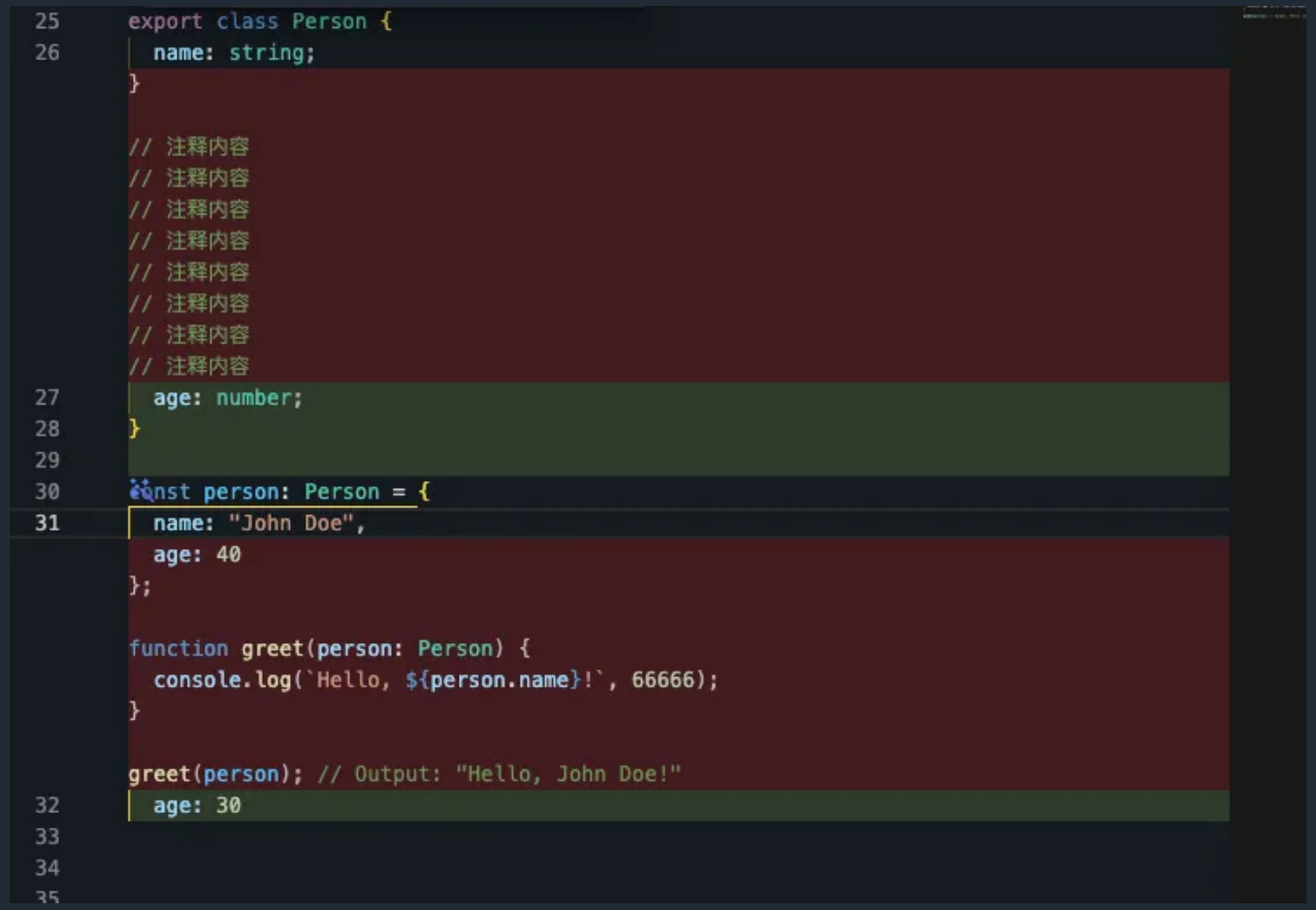
计算结

```
[
    {
        "original": {
            "startLineNumber": 3,
            "endLineNumberExclusive": 13
        },
        "modified": {
            "startLineNumber": 3,
            "endLineNumberExclusive": 6
        },
    },
    {
        "original": {
            "startLineNumber": 15,
            "endLineNumberExclusive": 23
        },
        "modified": {
            "startLineNumber": 8,
            "endLineNumberExclusive": 9
        }
    }
]
```

只需要两步

1. 先将左边的第 3 行开始到第 12 行结束（endLineNumberExclusive 字段减一）替换成右边的第 3 行开始到第 5 行结束
2. 第二步，将左边的第 15 行开始到第 23 行结束替换成右边的第 8 行

Inline Chat – Diff 算法

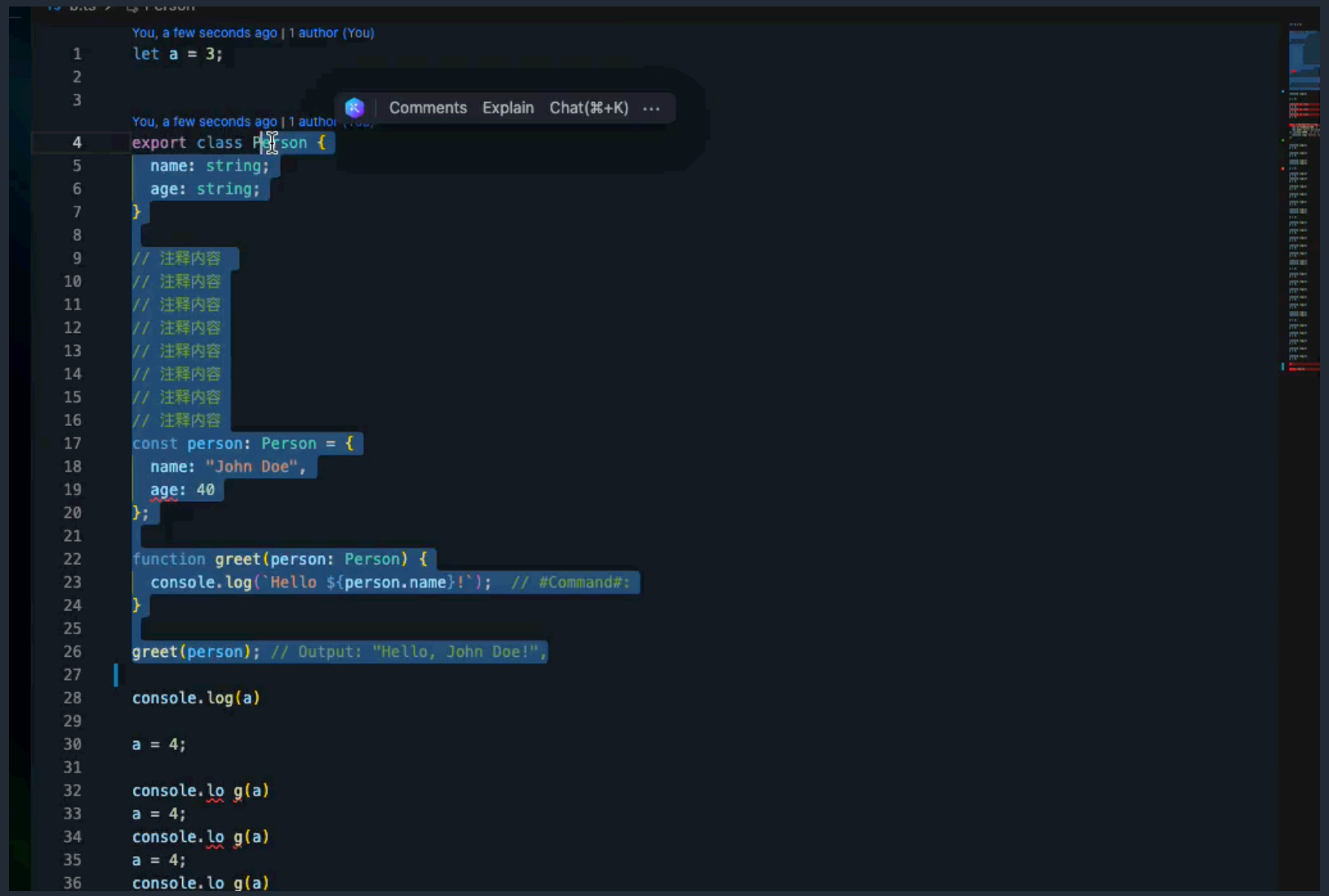


因为此时流式的过程是还在继续的

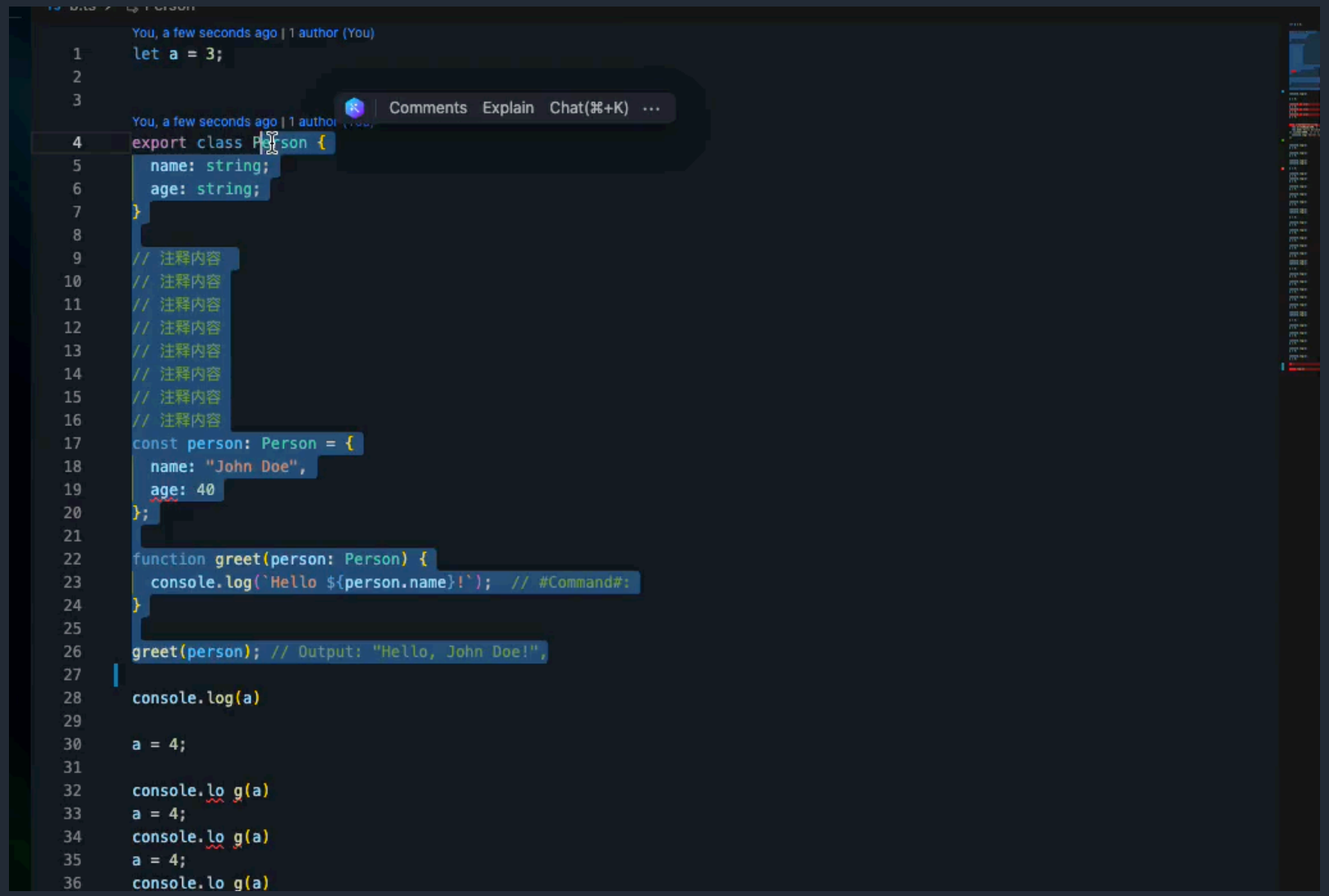
最后一块 diff 区域并不能直接进行替换操作

而是需要将其变成“待处理”区域，等待着下一次流式的结果进来

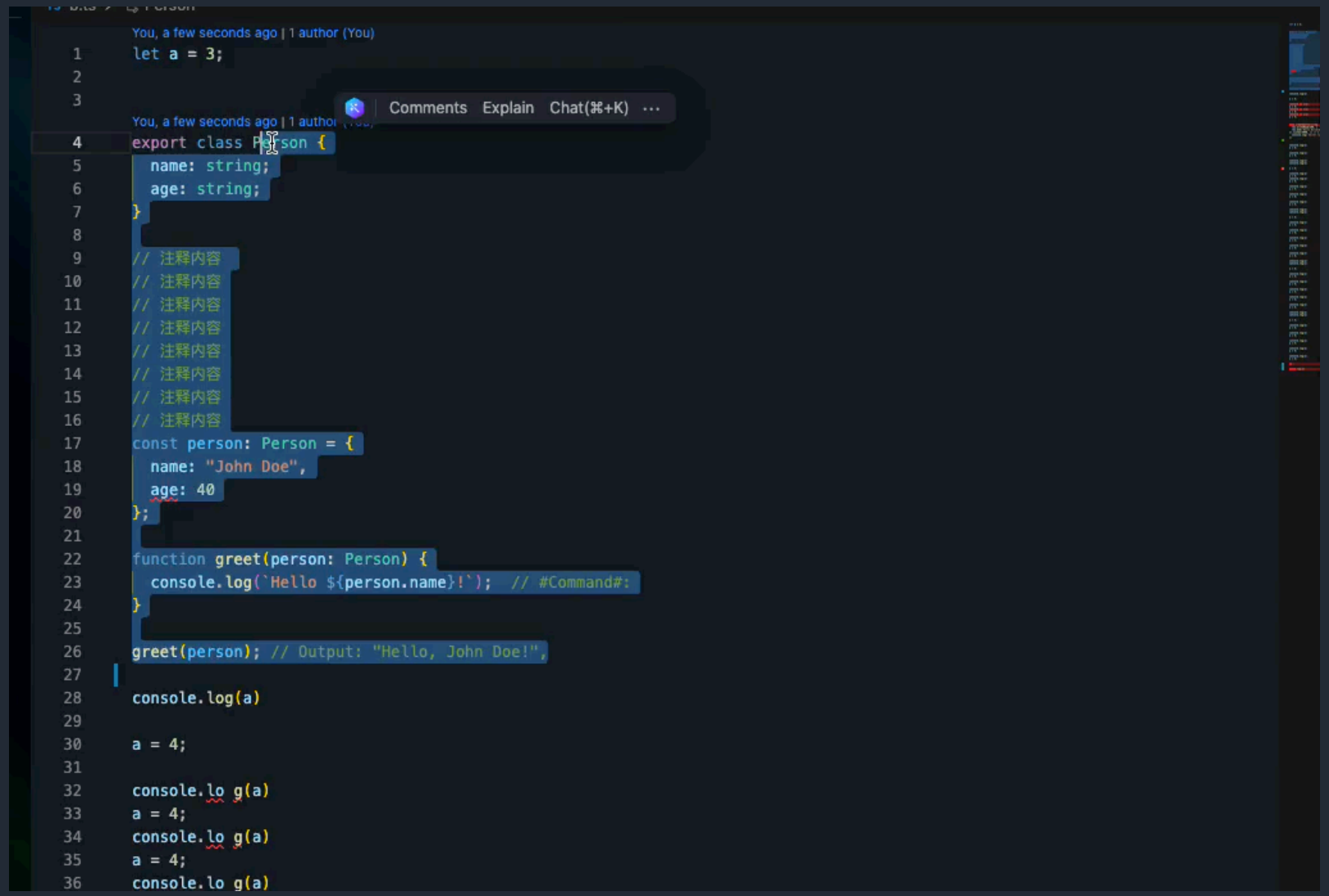
Inline Chat – Diff 算法



Inline Chat – Diff 算法



Inline Chat – Diff 算法



Inline Chat – Diff 算法

🤔 问题出在哪儿?

```
export class Person {
  name: string;
}

// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
// 注释内容
const person: Person = {
  name: "John Doe",
  age: 40
};

function greet(person: Person) {
  console.log(`Hello, ${person.name}!`, 66666);
}

greet(person); // Output: "Hello, John Doe!"
```

```
export class Person {
  name: string;
  age: number;
}

const person: Person = {
  name: "John Doe",
  age: 30
```

通过肉眼观察

1. 应该先插入第 3 行
2. 再删除 5 – 12 行
3. 最后替换 15 行

抖动的原因在于 LCS，因为流式的过程中最长公共子序列是会变的
他有长有短，反应到前端那就是红色删除区域一会长一会儿短

Inline Chat – Diff 算法

并不是求 “最少编辑次数”

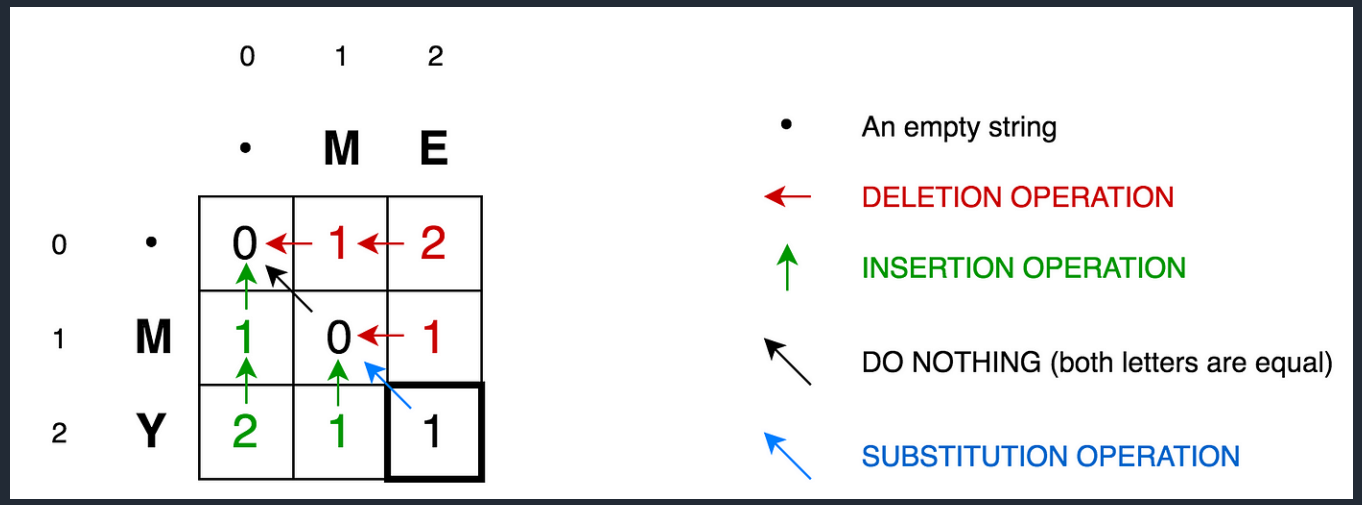
而是要求 “最短编辑距离”

也就是将一个单词修改为另一个单词所需的最少字符的编辑次数（插入、删除或替换）

典型的 Levenshtein 算法

Inline Chat – Levenshtein 算法

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$



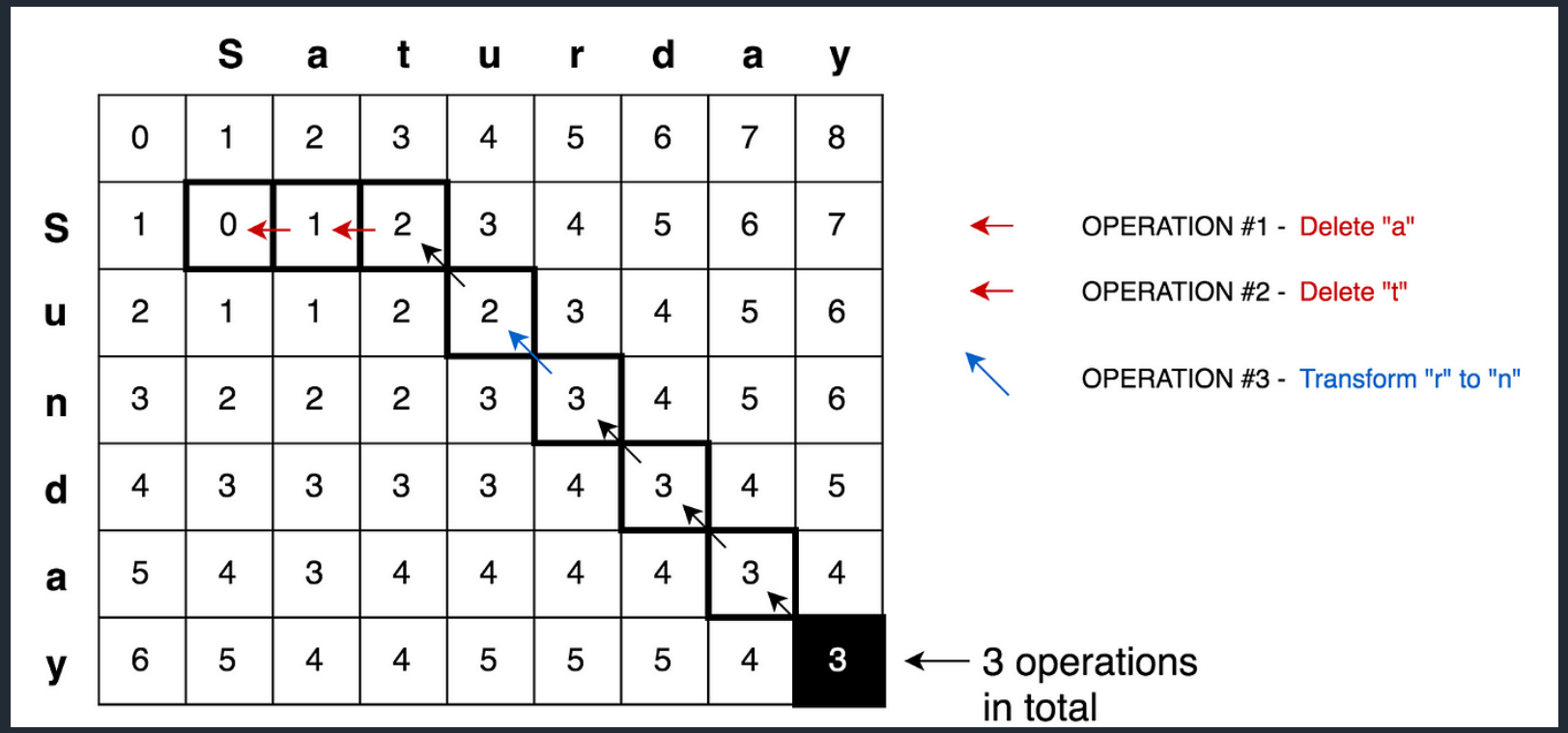
创建一个 3 x 3 的矩阵

每个格子 = 从以下结果中取最小

- 左边 + 1
- 上边 + 1
- 字符相等？ 左上角 ： （左上角 + 1 ）

Inline Chat – Levenshtein 算法

Saturday -> Sunday



- 字符相等：指向左上角
- 字符不等：min（左上，左，上）
  - 左：删除
  - 上：插入
  - 左上：替换

从最右下角开始逆向回溯到起点

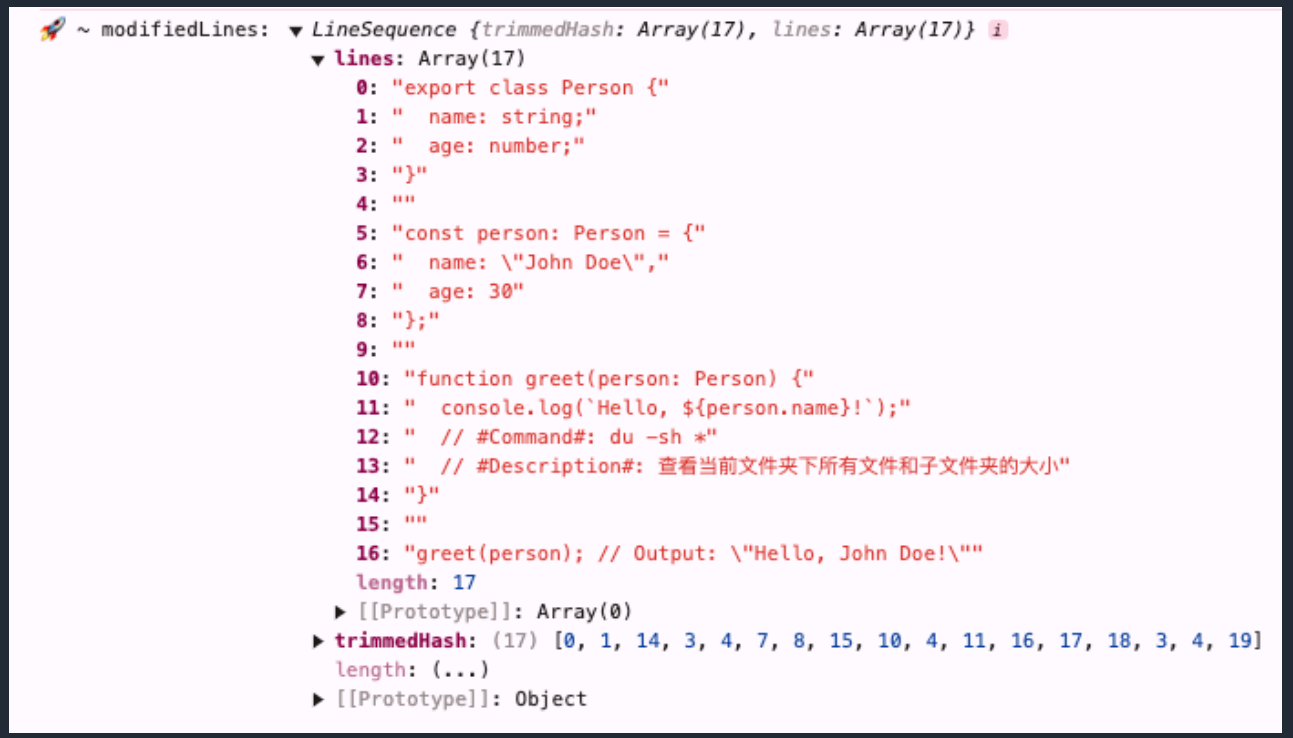
Inline Chat – Levenshtein 算法

- 单词 -> 整段代码
- 字符 -> 行

根据 levenshtein 算法先填充矩阵结果

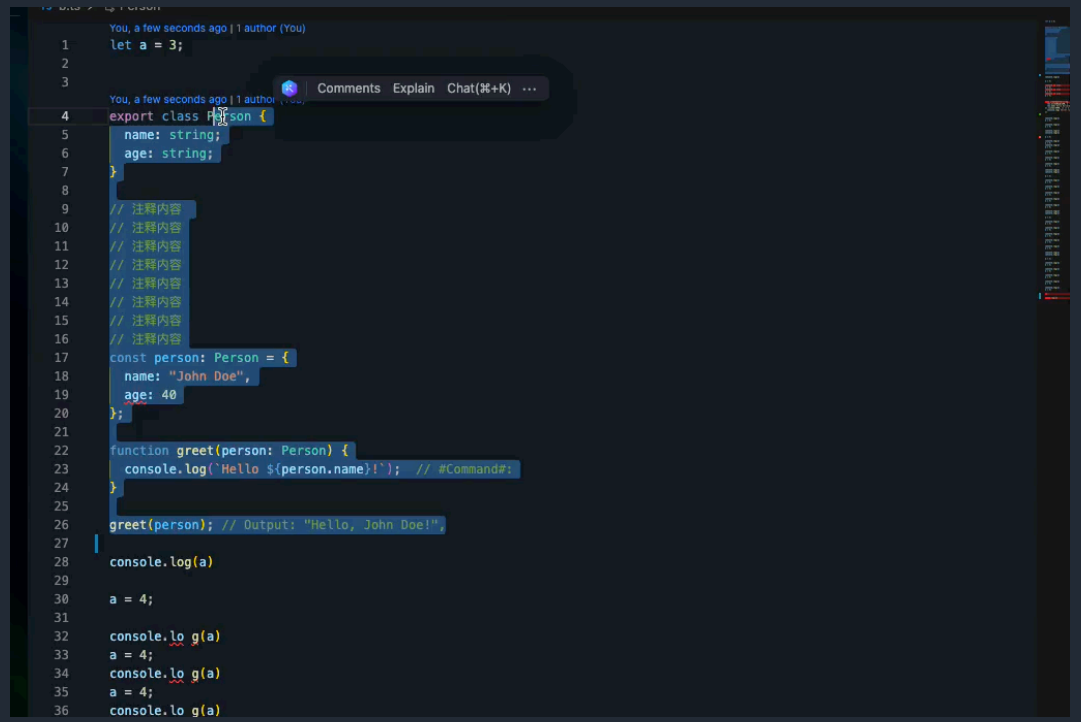
回溯时构建差异序列用于记录位置信息

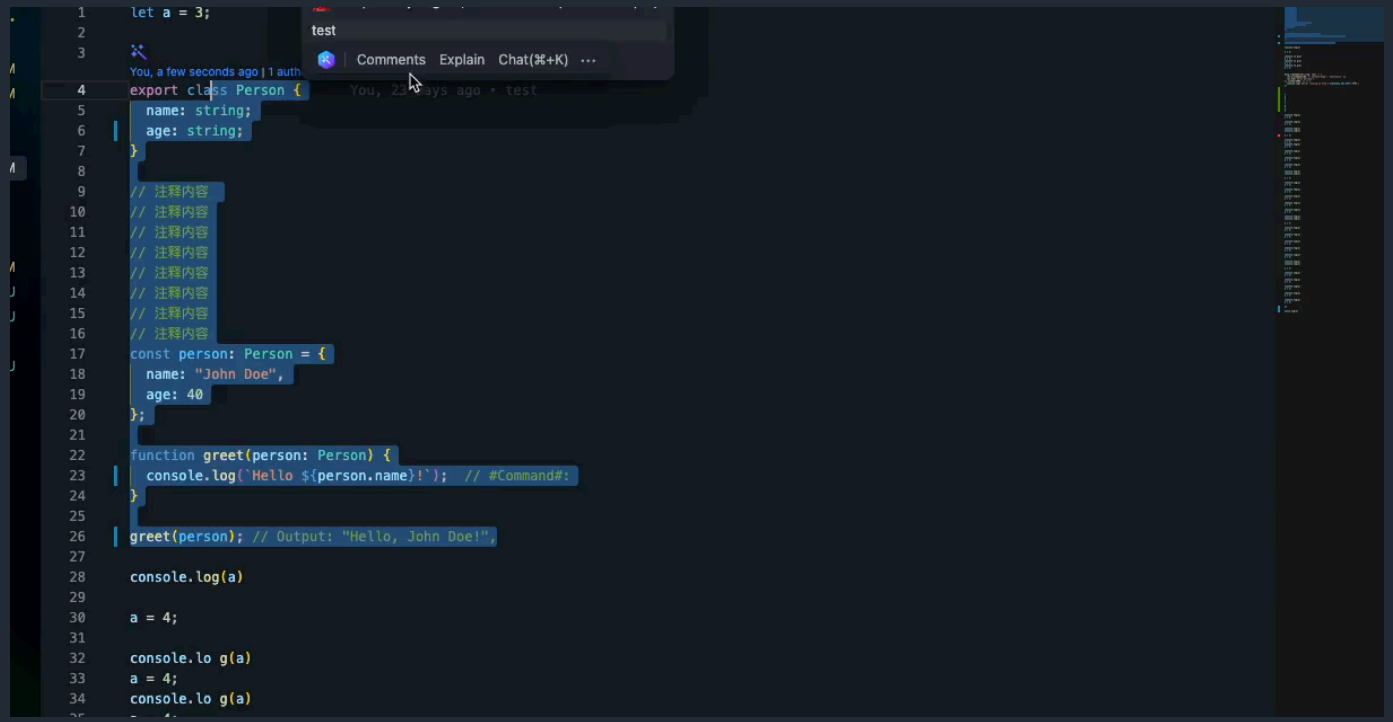
原代码



新代码

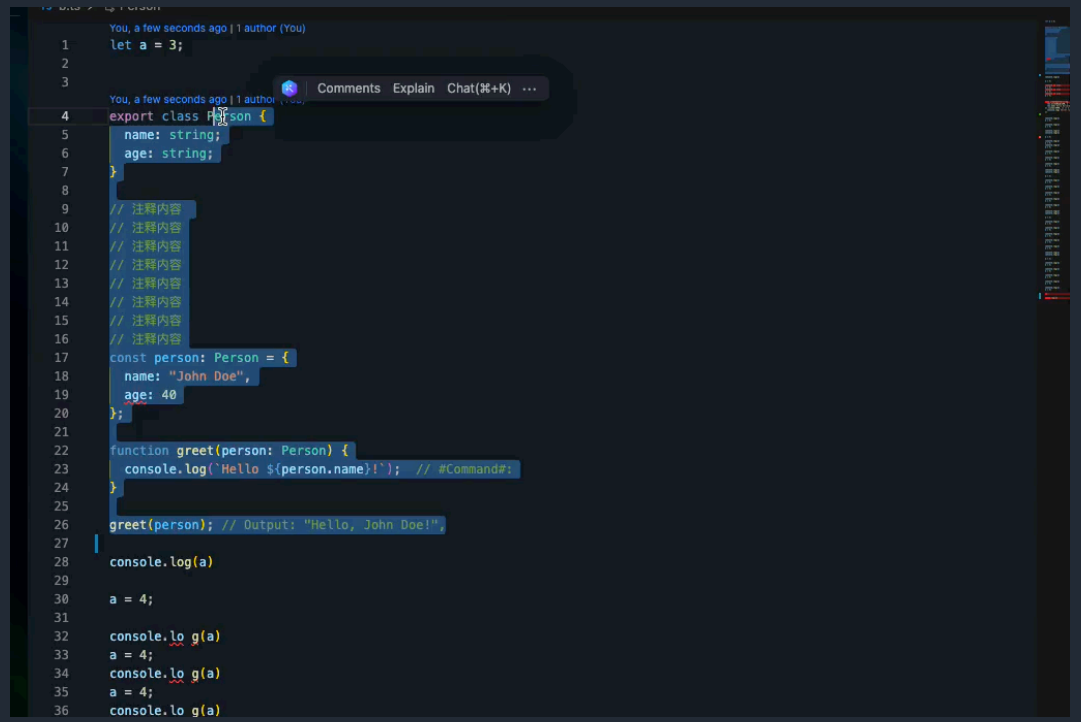
Inline Chat – Levenshtein 算法

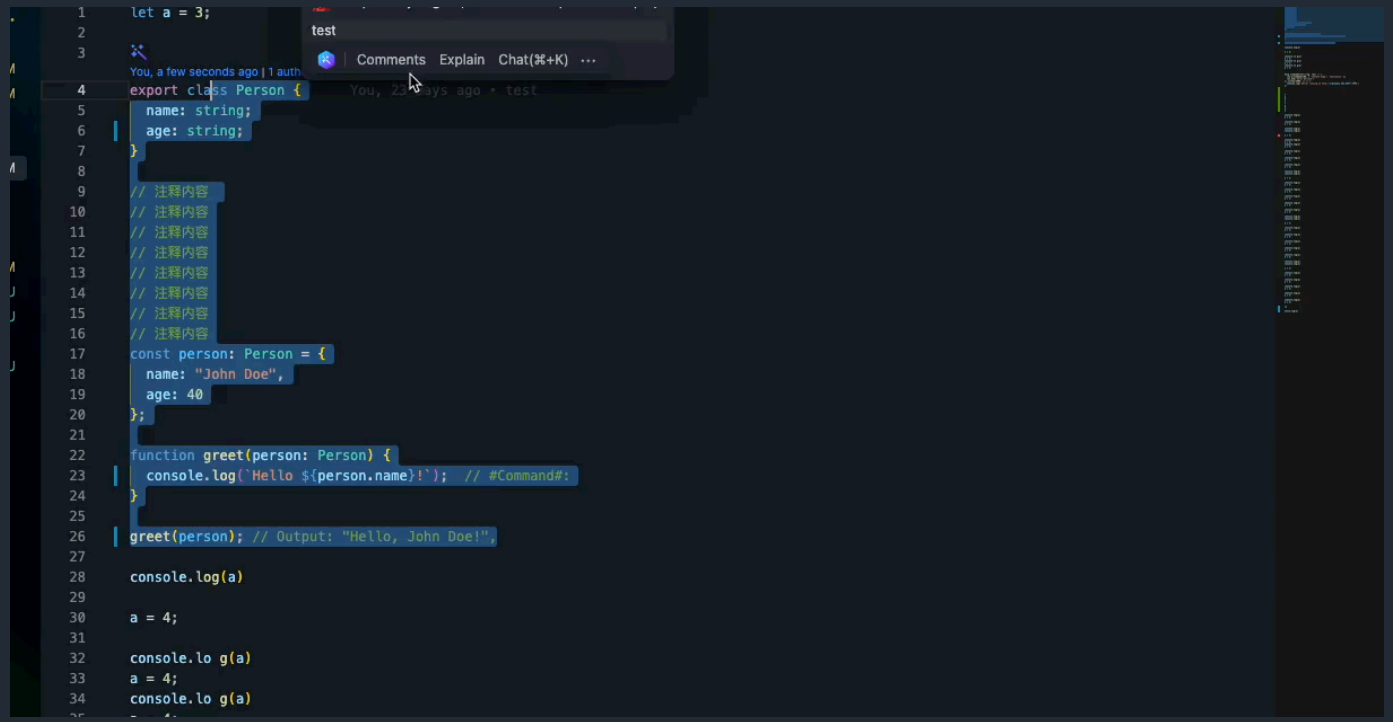




😔 😄

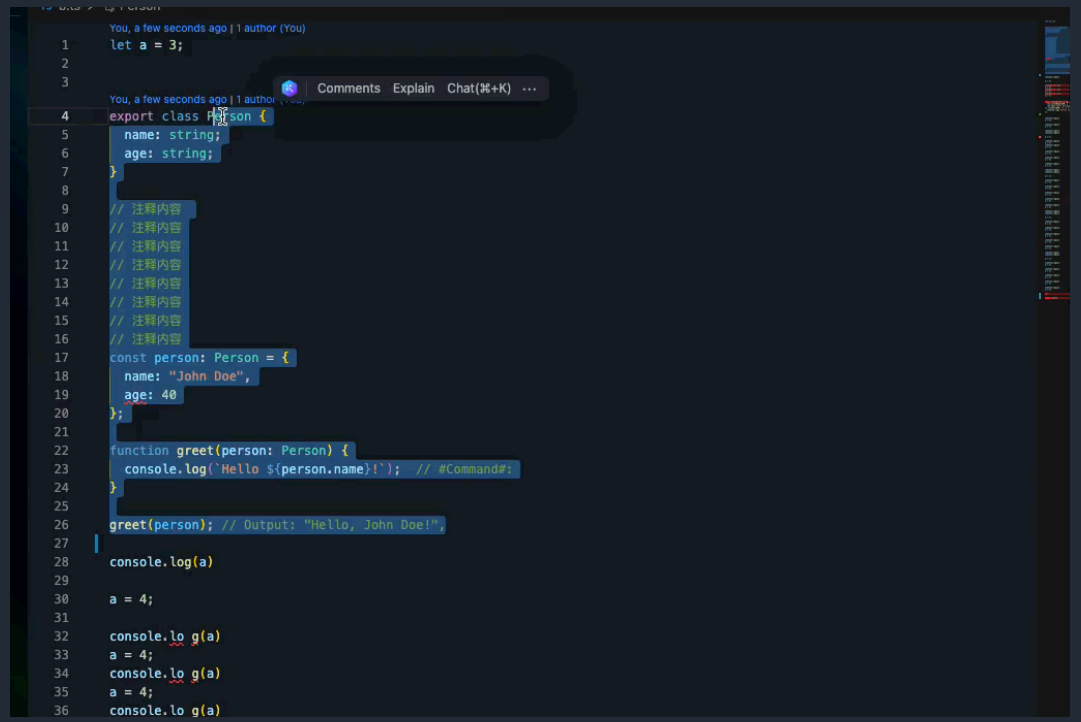
Inline Chat – Levenshtein 算法

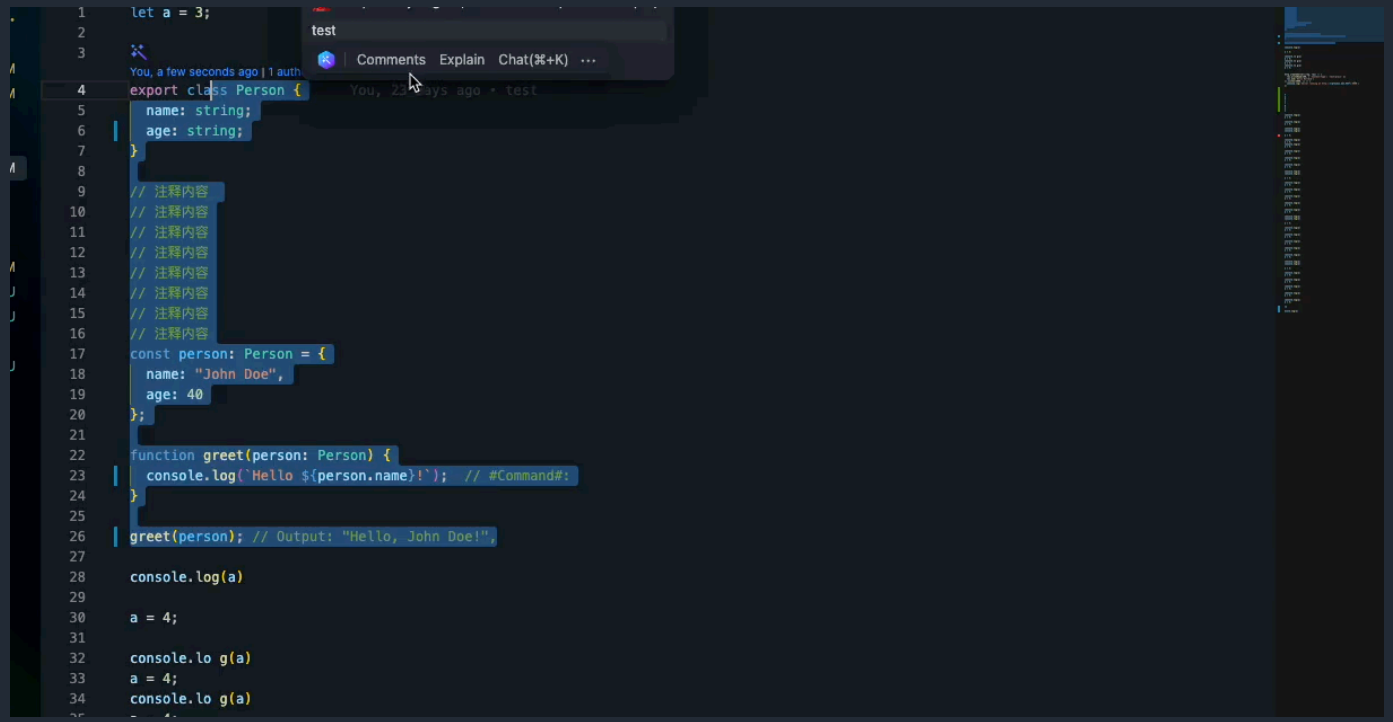




😔 😄

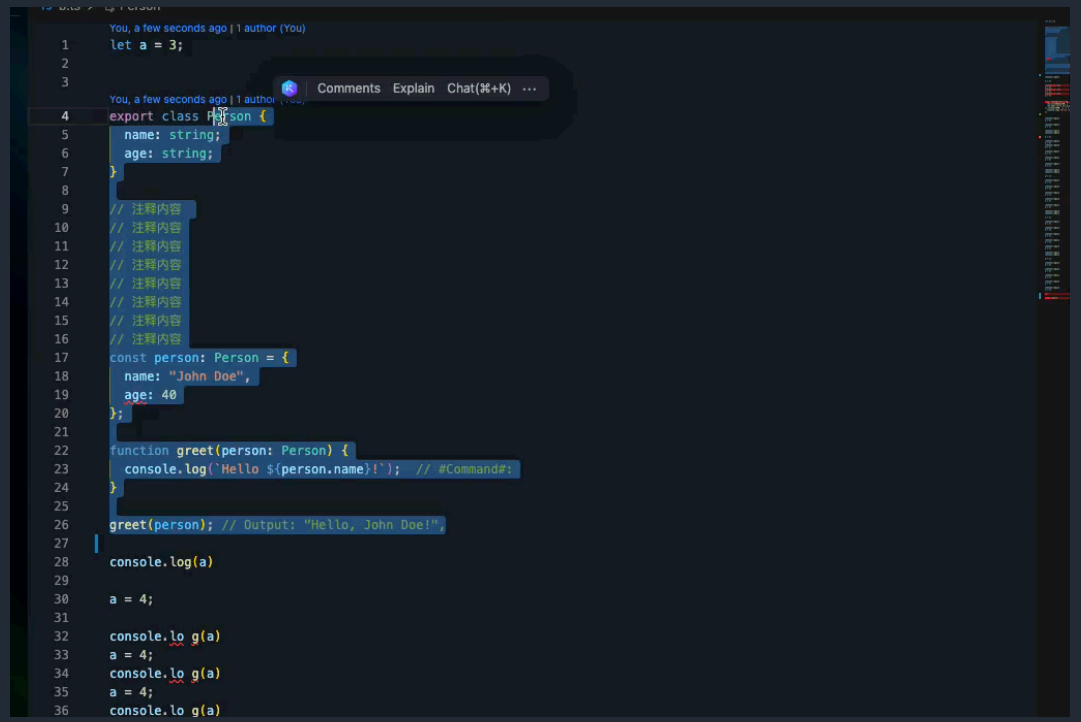
Inline Chat – Levenshtein 算法

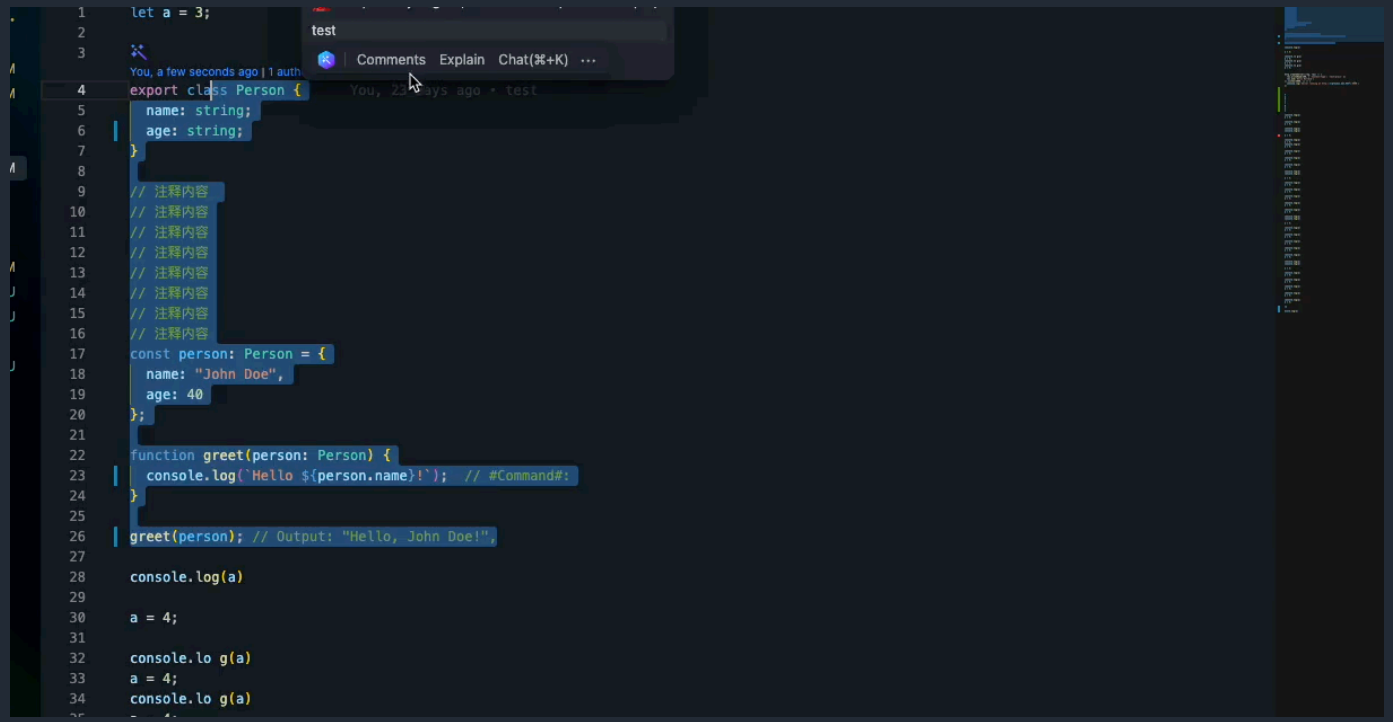




😔 😄

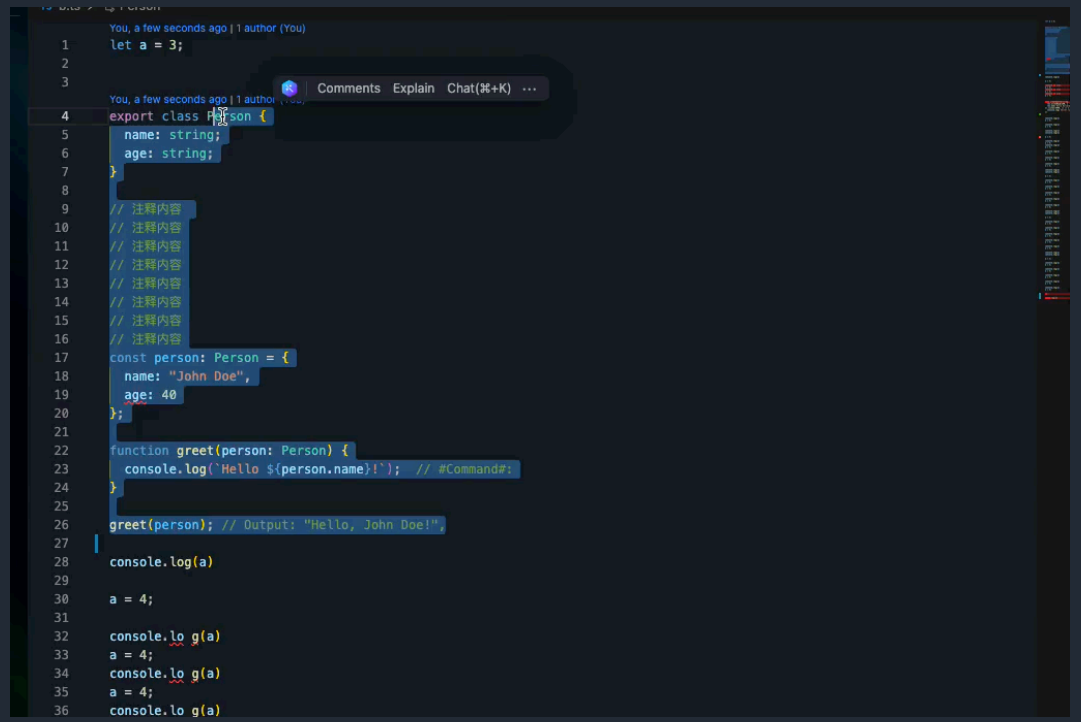
Inline Chat – Levenshtein 算法

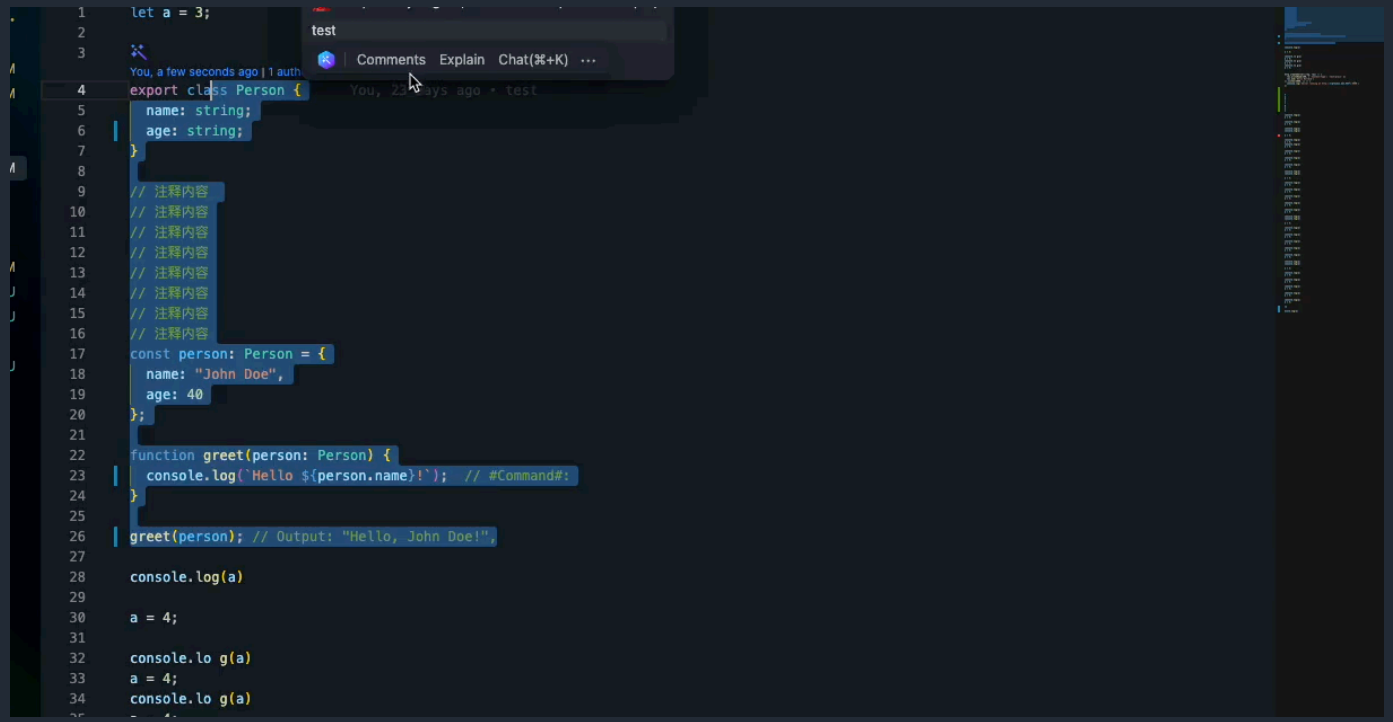




😔 😄

Inline Chat – Levenshtein 算法

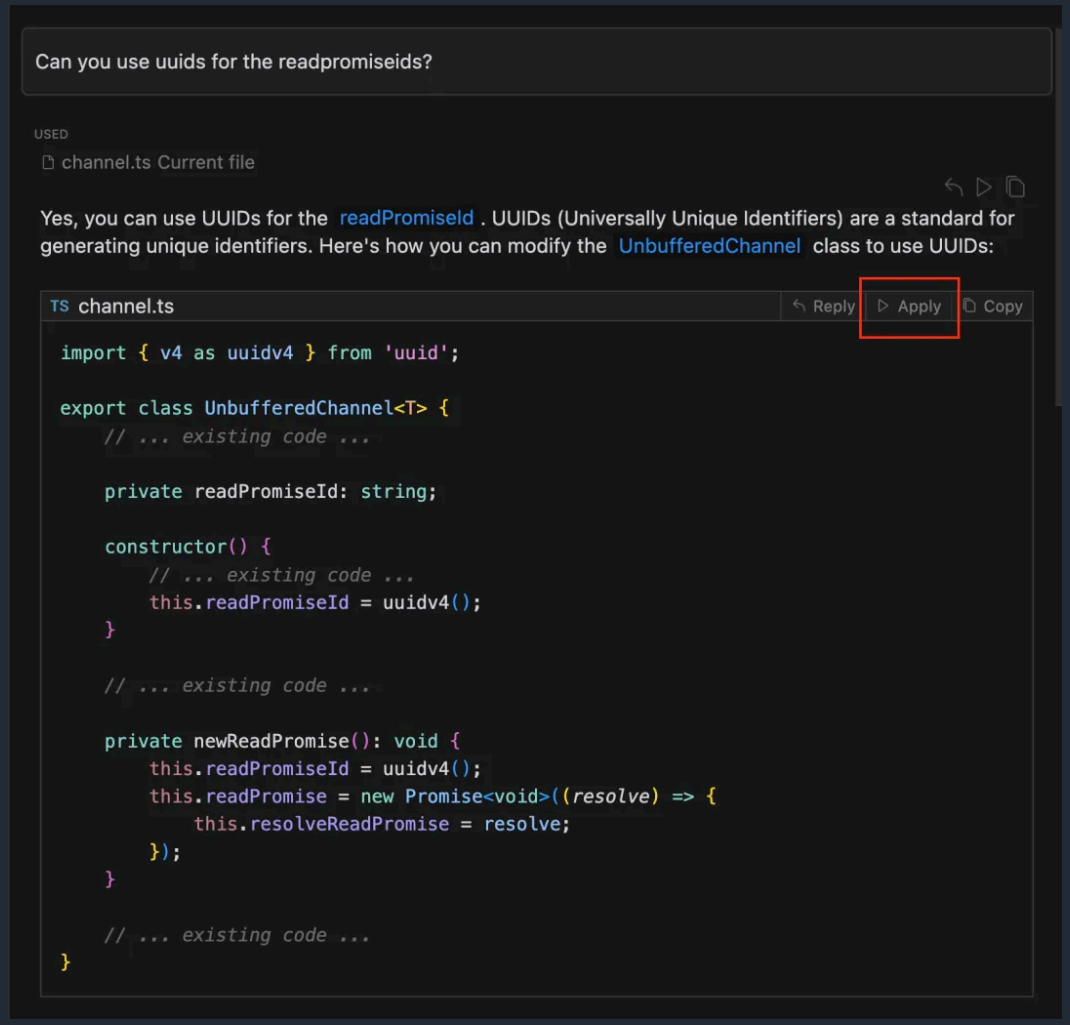




😔 😄

Instant-apply

- 训练 “Fast Apply” 模型
- 数据集设计：结合真实数据和合成数据，通过 GPT-4 生成 Chat 对话，以 1:4 的比例整合数据集。
- 训练：基于 Llama 3 模型，对小文件、文件名等进行下采样。
- 评估：设计一个包含 450 个全文件编辑的评估集，使用 Claude-3 Opus 作为评分标准。
- 因为局部修改存在上下文信息不足、Diff 不符合模型分布、行号问题等难点，采用全文重写。
- 采用 Speculative Decoding 推测解码技术加快推理，速度比传统方法快9倍



仓库级补全 – 用 rust 实现 js/ts 的本地索引

Agent 本地索引工具

构建相似性索引

遍历仓库中的所有文件（后缀 in java, rs, py, ts, js, go, cpp, tsx, jsx…）
对每个文件建索引操作：

1. 根据滑动窗口切分文件，比如窗口大小是 5 的话，整个文件会分为 0-5 行，6-10 行这样
2. 将每个窗口的代码进行 tokenizer，获取到分词 id 列表+ token 列表
3. 构造保存的键值，保存到本地数据库（ kv + json 存储）

构建相关性索引

遍历整个仓库文件，根据后缀判断语言，从而调用不同语言的解析策略
负责解析出每个文件中的代码信息，定义 CodeInfo 结构体

查询工作流

对不同语言实现匹配相关代码的工作流。找出相关代码后组装 prompt 给到大模型

仓库级补全 – 用 rust 实现 js/ts 的本地索引

用 OXC 解析 js/ts 的 AST 并保存成 CodeInfo 结构体

```rust
#[derive(Debug, Serialize, Deserialize, Clone, PartialEq)]
pub struct CodeInfo {
    //代码基础结构
    pub code_struct: String,
    //全限定名
    pub full_qualified_name: Option<String>,
    //import信息. 注意: java代码,如果处在同一个文件夹..没有import
    pub import_set: Option<HashSet<String>>,
    //代码类型
    pub class_type: Option<String>,
    //类名
    pub class_name: Option<String>,
    //继承父类的全限定名
    pub extend_class_name: Option<String>,
    //实现接口的全限定名集合
    pub implements_class_name_set: Option<HashSet<String>>,
    //类变量的名字-类型映射
    pub fild_name_class_map: Option<HashMap<String, String>>,
    /// 如果一个代码块内有多个类, 存在 class_list 中
    pub class_list: Option<Vec<ClassInfo>>,
}
```

大致流程：

1. 解析 AST，并用 ModuleRecordBuilder 构建依赖项（分析 requi
   import/export）
2. 用 oxc-resolver 将 import "./x.ts” => workspace/x.ts
3. 对于 node_module，分析的是 x.d.ts
4. 分析 AST 中的 class/function 等保存成 CodeInfo
5. 查询工作流的策略主体思想是类似语言服务的 Find Reference

最后

- AI 辅助编码是大模型最早落地的应用之一，也是最具有实用性和商业价值的场景之一
- Github Copilot、Cursor 的成功表明：除了模型以外，AI 原生的交互也很重要
- OpenSumi 3.0 完成 AI 原生 IDE 框架的升级，企业或个人可以对接任意模型，定制 AI IDE
- 团队招人，内部有很多 AI x IDE 相关的产品

qingyi.xjh@antgroup.com