36讲插件开发（五）：工作台相关API

在插件部分的前几篇文章，我一直在介绍编辑器相关的 API。不过，除了拓展编辑器以外，我们还可以拓展 VS Code 的其他组件，这一类 API，我把它们叫做**工作台 API**。下面我们就来看看， 工作台相关的 API 有哪些种类，以及我们可以如何使用它们。

今天，我们依然使用 JavaScript 插件模板。

一、信息提示和 QuickPick

第一类，就是通过插件 API 在 VS Code 中调出对话框向用户询问问题，或者弹出信息提示以警示用户。其实这二者的本质都是一致的，都是跟用户进行信息的交互，以完成进一步的操作。

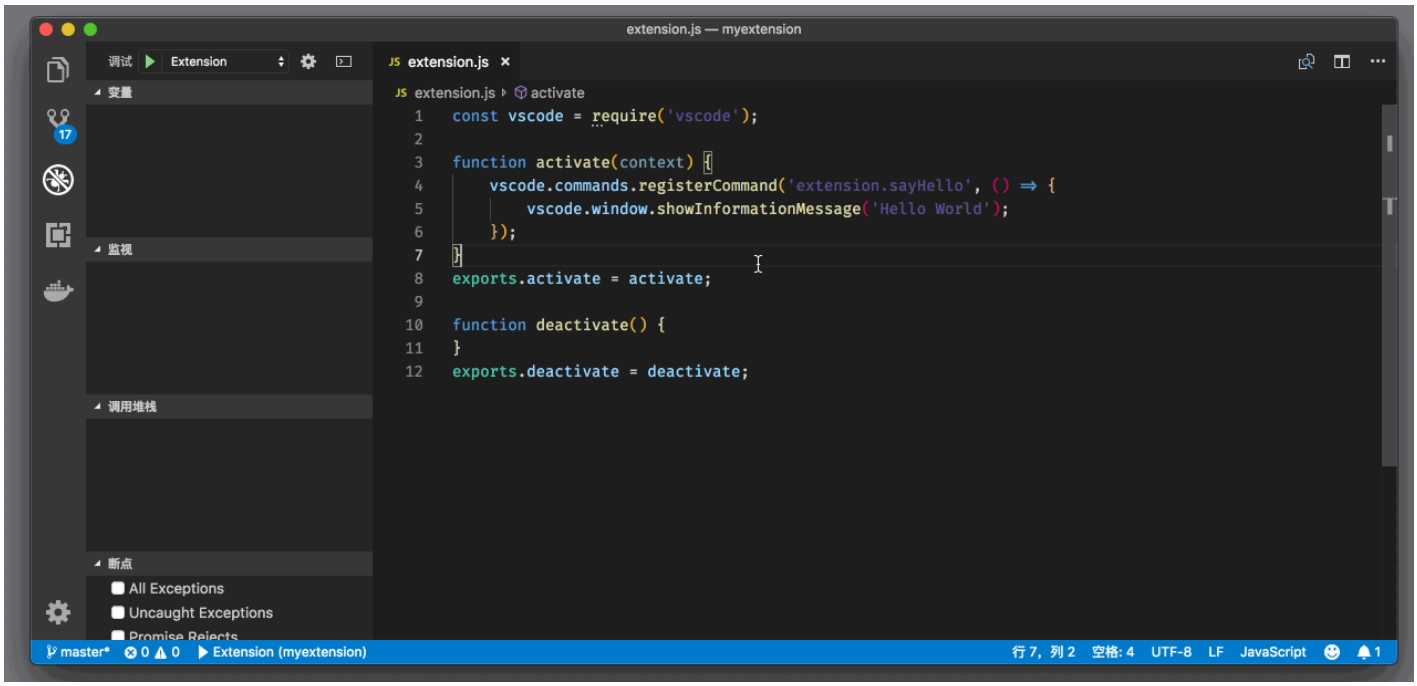
1、Information、Warning和Error 消息

首先我们来看看信息提示，这个 API 我们在之前的例子里已经使用过了。

下面的这段示例代码，我们注册了 extension.sayHello 这个命令，这个命令通过 vscode.window.showInformationMessage API，输出了一个提示 “Hello World”：

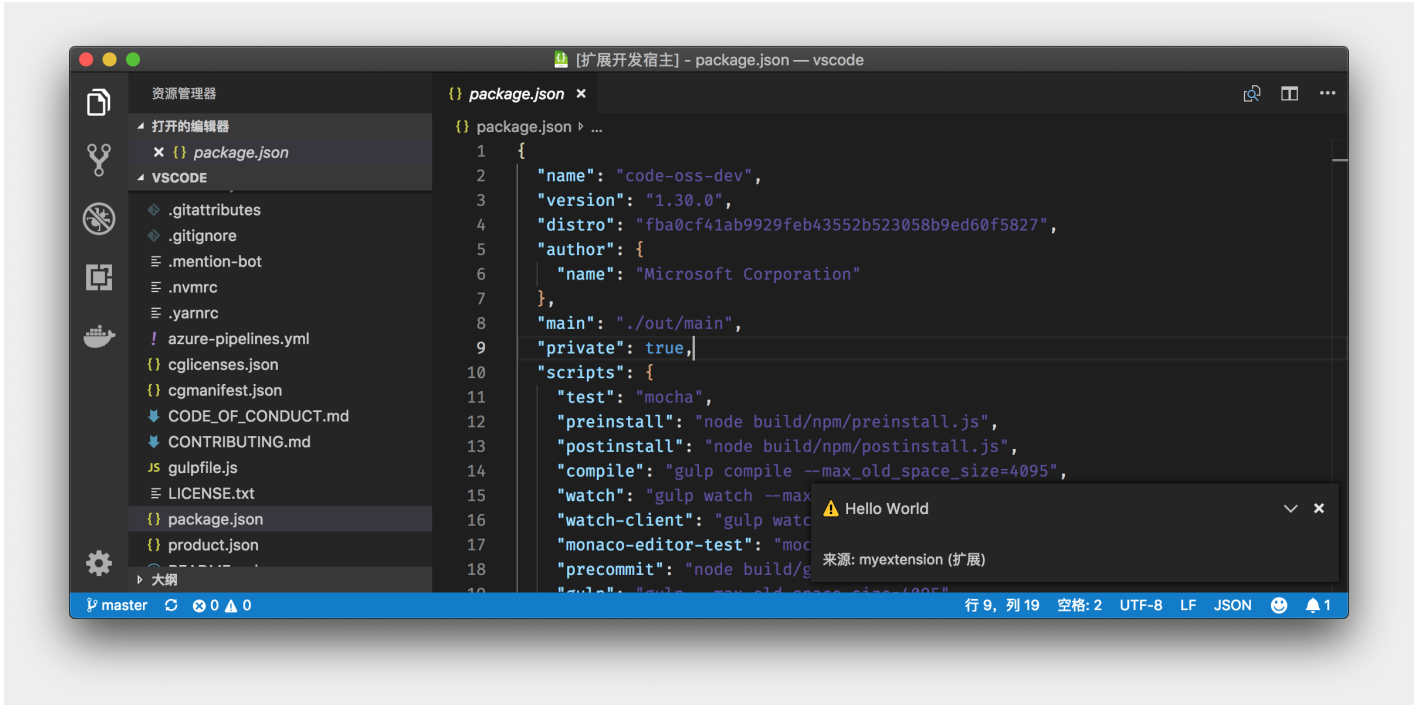
```
vscode.commands.registerCommand('extension.sayHello', () => {
    vscode.window.showInformationMessage('Hello World');
});
```

代码运行的效果如下：

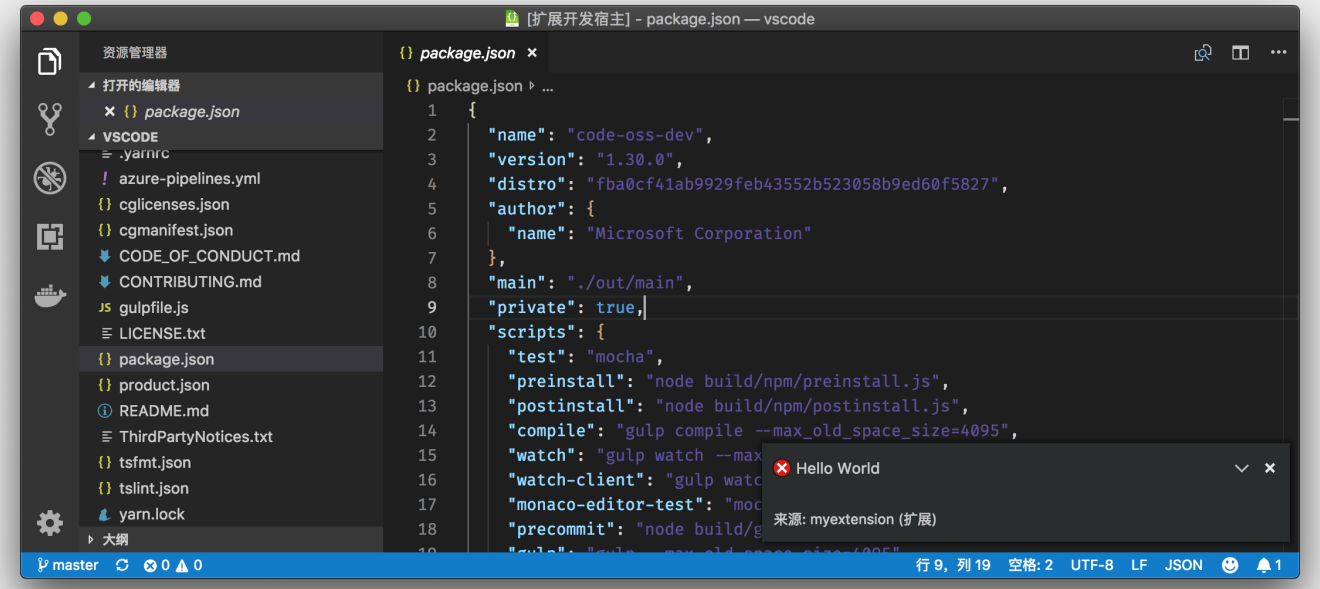


showInformationMessage

然后我们再来看看对话框。这里我们可以使用的 API 还有两种：showWarningMessage 和 showErrorMessage。使用方法都是一样的，不过呈现效果会不同，以体现 Information、Warning和Error 不同的重要程度。



showWarningMessage



showErrorMessage

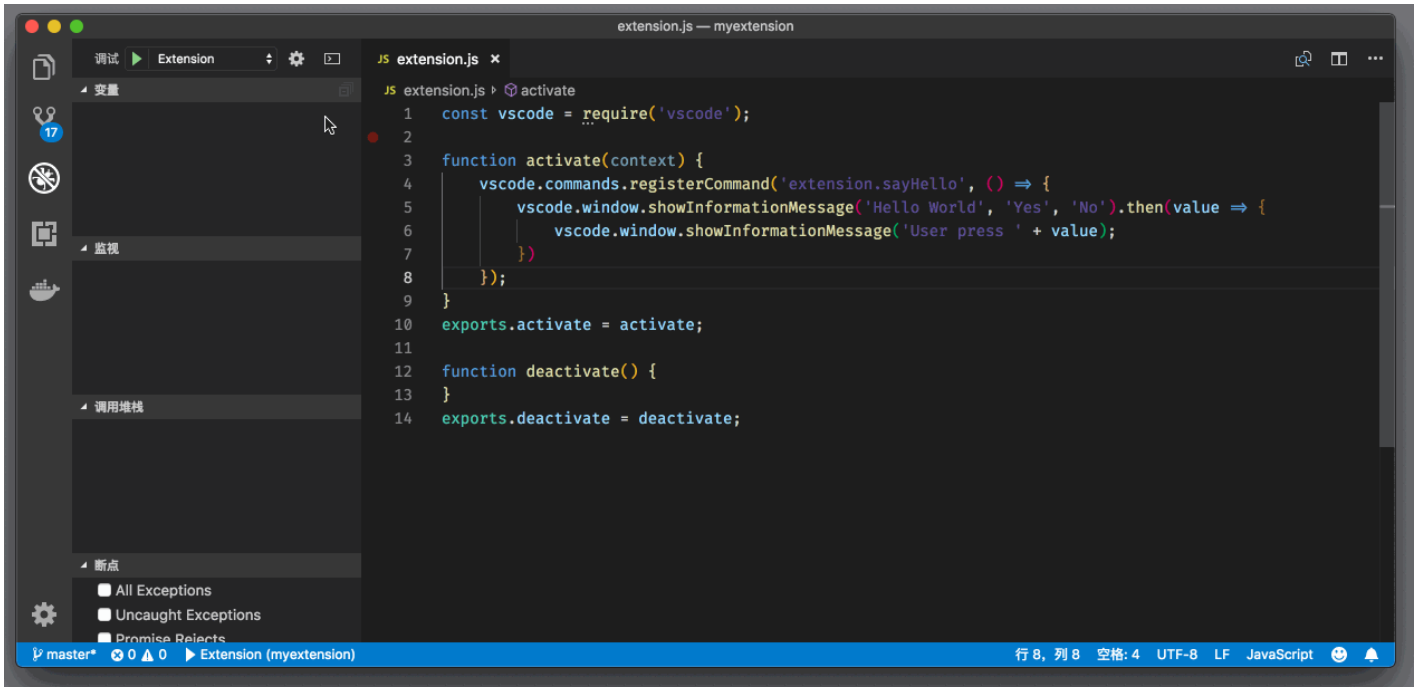
除了给用户展示信息以外，这三个 API 还允许我们和用户进行简单的交互。下面我们来看看 showInformationMessage 的定义：

```
/**
 * Show an information message to users. Optionally provide an array of items which will be presented as
 * clickable buttons.
 *
 * @param message The message to show.
 * @param items A set of items that will be rendered as actions in the message.
 * @return A thenable that resolves to the selected item or `undefined` when being dismissed.
 */
export function showInformationMessage(message: string, ...items: string[]): Thenable<string | undefined>;
```

除了传入消息 message，我们还可以传入一个数组，这个数组里的字符串，都会被渲染成按钮，当用户按下这些按钮，我们就能够收到反馈了。下面，我们不妨把样例代码修改成：

```
vscode.commands.registerCommand('extension.sayHello', () => {
    vscode.window.showInformationMessage('Hello World', 'Yes', 'No').then(value => {
        vscode.window.showInformationMessage('User press ' + value);
    })
});
```

我们给用户提供了 “Yes” 和 “No” 两个选项，当用户选择其中之一后，弹出一个新的信息框，显示用户点击了哪个。



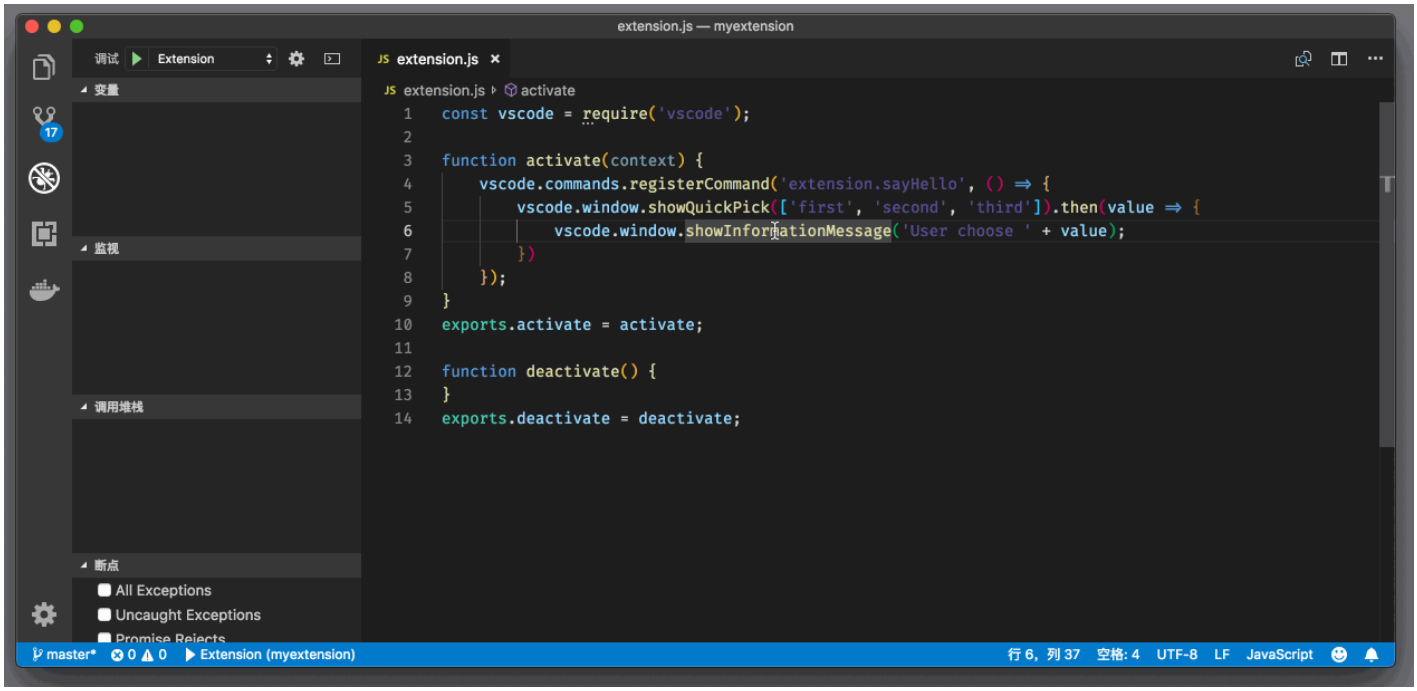
showInformationMessage 用户操作选项

2、QuickPick

接下来这个 API 就是 QuickPick 了，通过 `vscode.window.showQuickPick` 这个函数，给用户提供了一系列选项，然后根据用户选择的选项进行下一步的操作。比如下面的这段代码：

```
vscode.window.showQuickPick(['first', 'second', 'third']).then(value => {
   vscode.window.showInformationMessage('User choose ' + value);
})
```

我们给用户提供了三个选项：‘first’、‘second’、‘third’，然后将用户的选择以信息的方式弹出。



showQuickPick API 的第一个参数除了可以是一个字符串数组以外，还可以提供其他不同的类型。

比如**Promise**，这个参数可以为最终解析值为字符串数组的 Promise。有了这个类型，我们就能够异步地获取选项列表，等这个列表解析出来了再提供给用户，而用户则会在界面上看到滚动条。

另外，这个数组也可以是 **QuickPickItem** 对象数组。QuickPickItem 的结构为：

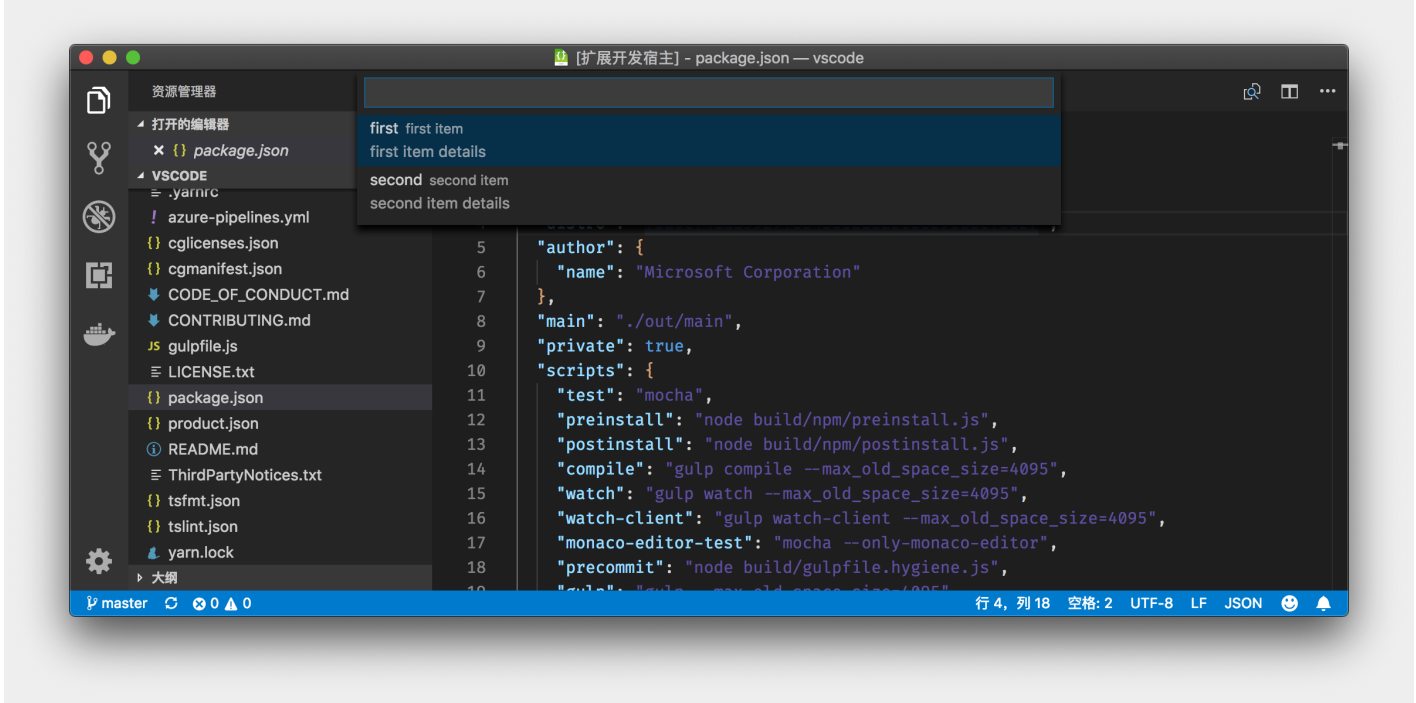
```
export interface QuickPickItem {
 /**
  * A human readable string which is rendered prominent.
  */
 label: string;
 description?: string;
 detail?: string;
 picked?: boolean;
 alwaysShow?: boolean;
}
```

其实，我们在上面的示例里面使用的数组，也可以用 QuickPickItem 来替代，我们只需要使用 QuickPickItem 的 label 属性，然后label 里的值就会被渲染在列表中。

除了 label，我们还可以通过 description 或者 detail 来提供更多的信息。比如说，如果我们使用下面的 QuickPickItem 数组：

```
vscode.commands.registerCommand('extension.sayHello', () => {
    vscode.window.showQuickPick([{
        label: 'first',
        description: 'first item',
        detail: 'first item details'
    }, {
        label: 'second',
        description: 'second item',
        detail: 'second item details'
    }]).then(value => {
        vscode.window.showInformationMessage('User choose ' + value.label);
    })
});
```

我们则会得到如下图所示的列表：



QuickPickItem

至于picked 这个属性，就非常好理解了。默认情况下列表里的第一个选项会被选中。如果你希望默认选中其他项的话，将它的 picked 属性改为 true 就好了。alwaysShow 这个属性则是使用于列表很长的情况，如果列表非常长，VS Code 不得不渲染出滚动条时，那么通过将某些项的 alwaysShow属性改为 true，这个选项就会一直出现在列表中，而不会受滚动条的影响。

整体来讲，在我们实现插件的过程中，很多命令或者操作的流程、信息，并不是完全确定的，我们往往需要用户来提供更多的信息，并且由用户来做出最终的决定。这个时候，通过信息提示和 QuickPick，我们将选择权交还给了用户。

但是一定要注意的是：信息提示和 QuickPick都是会打扰用户的正常工作。所以我们在使用这类 API 的时候一定要慎重，不然用户可能就会卸载我们的插件了。

二、面板 Panel

第二类就是面板里的信息了。默认情况下，面板中有以下几个组件：

- 问题面板
- 调试面板
- 输出面板
- 终端面板

除了调试面板是由调试插件控制的以外，其他的三个，都是可以通过普通的插件 API 来完成。而这里面，又属问题面板和输出面板使用最为频繁，所以我们着重来看看这二者的 API。

1、问题面板

当我们在书写代码时，VS Code 的各类插件，会把代码中出现的错误信息提供给问题面板。然后用户就可以通过问题面板，快速地查询问题并且进行代码的跳转。

问题面板相关的 API 存在于 vscode.languages 这个 namespace 下。要给问题面板提供相关的信息，我们要使用的 API 是 createDiagnosticCollection。

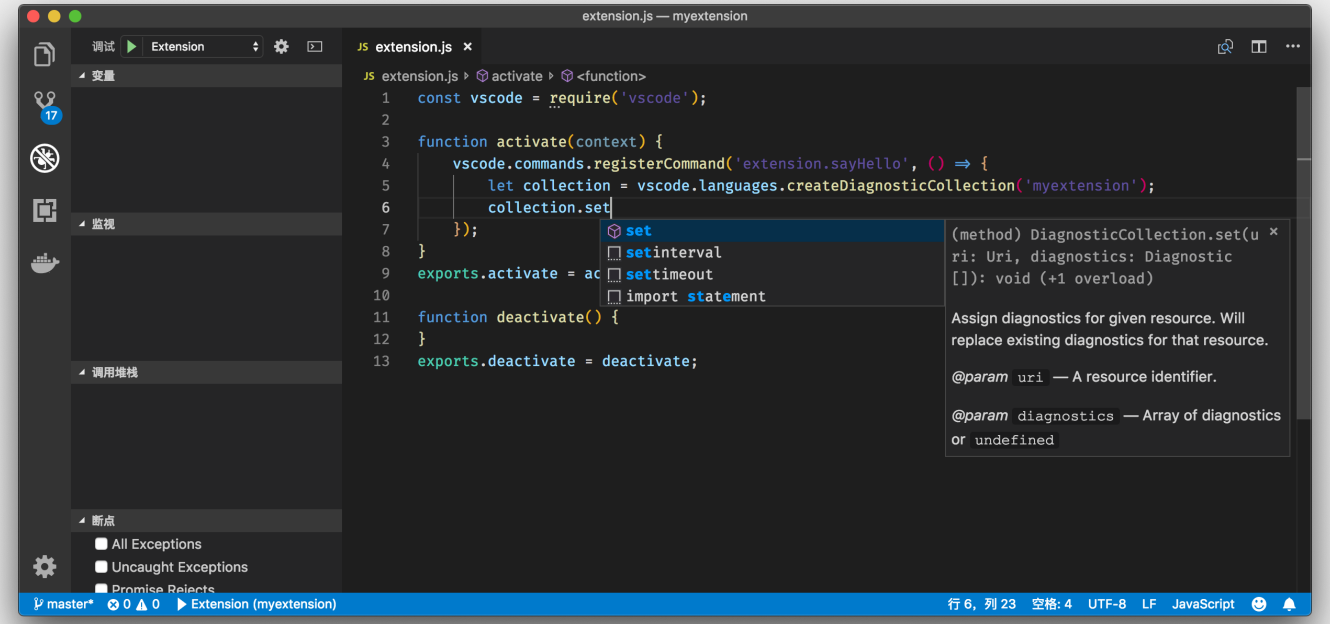
```
export namespace languages {
  /**
   * Create a diagnostics collection.
   *
   * @param name The name of the collection.
   * @return A new diagnostic collection.
   */
  export function createDiagnosticCollection(name?: string): DiagnosticCollection;
}
```

通过 vscode.languages.createDiagnosticCollection 创建出来的对象，就将是我们跟 VS Code 问题面板通讯的中介。

下面，我们使用如下的代码样例进行讲解。

```
vscode.commands.registerCommand('extension.sayHello', () => {
    let collection = vscode.languages.createDiagnosticCollection('myextension');
    let uri = vscode.window.activeTextEditor.document.uri;
    collection.set(uri, [
        {
            range: new vscode.Range(0, 0, 0, 1),
            message: 'We found an error'
        }
    ]);
});
```

在 collection 对象创建出来后，接着我们就要往这个 collection 里塞数据，这里要使用的 API 是 `set(uri: Uri, diagnostics: Diagnostic[] | undefined): void;` 。



collection.set

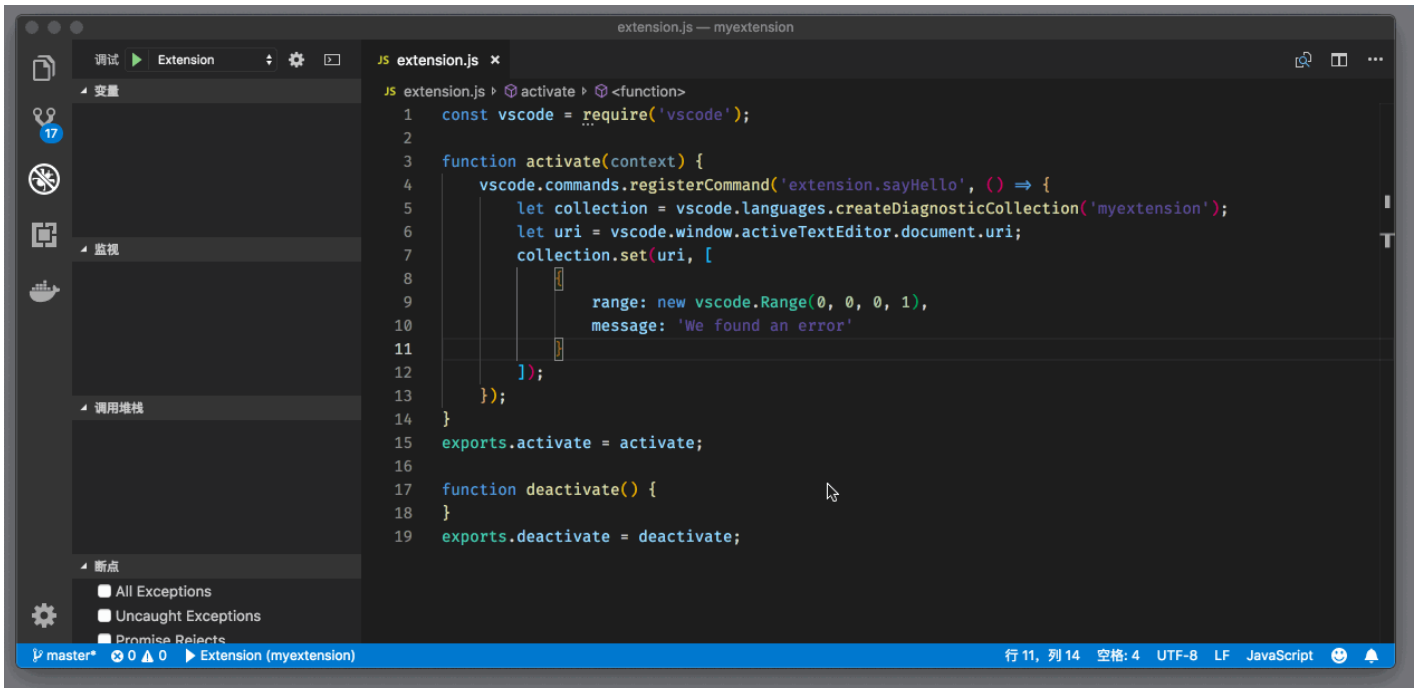
set 函数提供两个参数：第一个就是文档的地址 Uri，样例代码里使用了 vscode.window.activeTextEditor.document.uri ，也就是当前编辑器里的文档的地址 Uri；第二个就是我们在这个文档里发现的所有问题，每个问题的类型必须是 Diagnostic。

```
export class Diagnostic {
 /**
  * The range to which this diagnostic applies.
  */
 range: Range;
 /**
  * The human-readable message.
  */
 message: string;
 /**
  * The severity, default is error.
  */
 severity: DiagnosticSeverity;

 source?: string;
 code?: string | number;
 relatedInformation?: DiagnosticRelatedInformation[];
 tags?: DiagnosticTag[];
 constructor(range: Range, message: string, severity?: DiagnosticSeverity);
}
```

Diagnostic 对象必须要提供的两个属性就是 range 和 message，也就是问题所在的位置和问题相关的信息。我们还可以给

Diagnostic 对象提供诸如 severity 问题的程度、source 问题的来源等。

当我们把上面的代码运行起来后，在编辑器里执行 “Hello World” 命令，可以看到第一行第一列代码下出现了波浪线，同时问题面板里也多出了一个条目，点击它就能够跳转到编辑器中。



输出内容至问题面板

2、输出面板

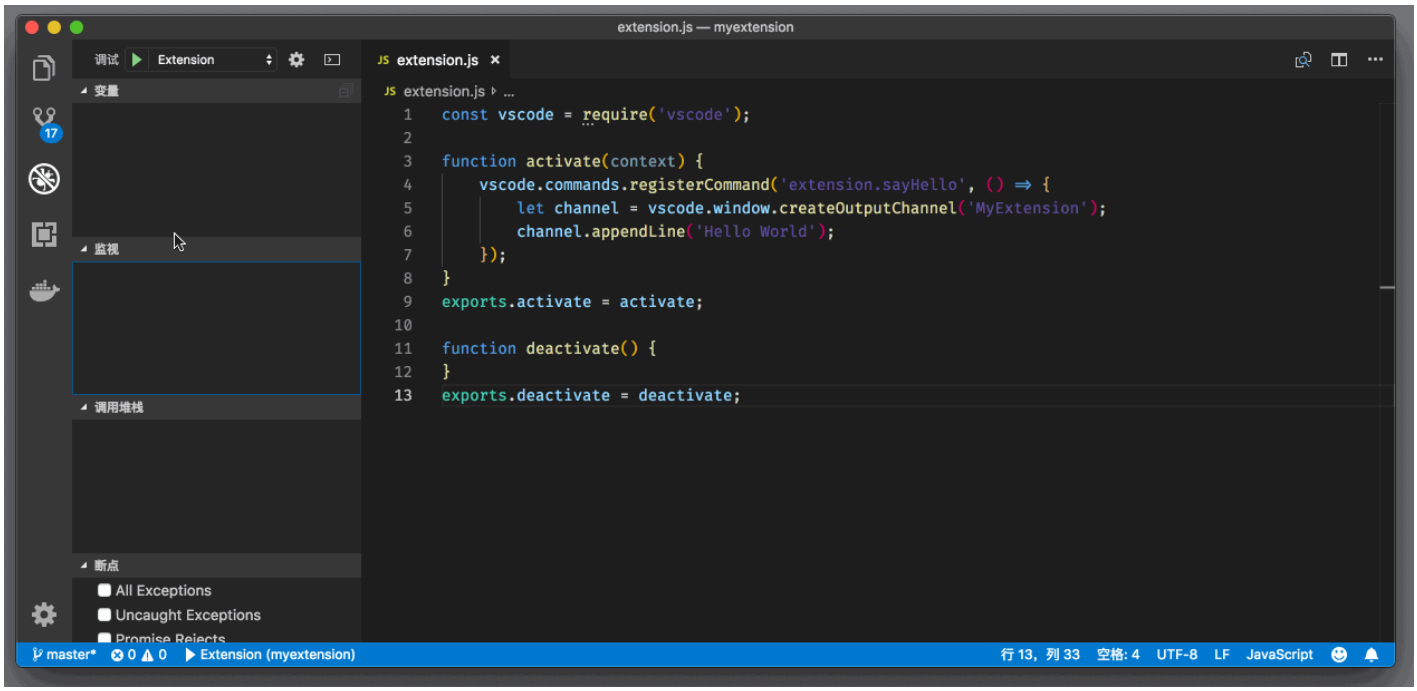
相比于问题面板，为输出面板提供内容的 API 要更简单一些。这一次，我们首先要创建一个 OutputChannel：

```
let channel = vscode.window.createOutputChannel('MyExtension');
```

我们只要提供一个名字即可。接着，我们就可以往这个对象中添加输出日志了：

```
channel.appendLine('Hello World');
```

有了这两行代码，就可以运行了。



输出内容至输出面板

通过上面的代码你可以发现，输出面板下拉框中现在出现了一个新的选项，叫做**MyExtension**，也就是我们创建的 OutputChannel。接着我们使用 channel.appendLine 输出的信息，就会被放在输出面板中。这套 API 非常像 console.log()，唯一不同的是，这套 API 将内容输出到了输出面板中。

这部分总体来说就是：问题面板的使用，跟语言服务结合到一起会很好，比如 Linting 信息、编译错误信息，甚至错别字检查信息，都可以塞到问题面板中。不过要注意，问题面板里的内容，意味着需要用户去修改代码。所以一些无关紧要的信息就不要放到这里面了。

而输出面板，大家完全可以把它当 log 日志来使用。大部分时间用户不需要去关心它，不过当用户遇到问题了，如果能够通过输出日志里的信息获得帮助，那么输出面板的目的就达到了。

三、视图 TreeView

第三类 API 则是跟视图有关。这一套 API 的最初需求是来自于 Visual Studio 用户，在 Visual Studio 中，我们可以在视图中看到项目、测试、云管理等，但是 VS Code 当时并没有 API 可以实现这种定制。于是 TreeView API 应运而生，通过实现这套 API，任何插件都可以实现类似于资源管理器的树形结构。

TreeView API 虽然是用于创建视图中树形结构的，但是它跟 VS Code 的其他 API 非常类似，都是给 VS Code 提供数据，然后 VS Code 来进行渲染。创建 TreeView 的 API 也非常简单：

```
export namespace window {
 export function registerTreeDataProvider<T>(viewId: string, treeDataProvider: TreeDataProvider<T>): Disposable
}
```

registerTreeDataProvider 一共有两个参数：第一个是这个 TreeView 的名字，第二个则是这个 TreeView 的数据来源 Data Provider。那么这个 Data Provider 长什么样呢?

```
export interface TreeDataProvider<T> {
 onDidChangeTreeData?: Event<T | undefined | null>;
 getTreeItem(element: T): TreeItem | Thenable<TreeItem>;
 getChildren(element?: T): ProviderResult<T[]>;
 getParent?(element: T): ProviderResult<T>;
}
```

这个 Data Provider 上，只有两个属性是必须的。第一个是 **getTreeItem**，通过这个函数，VS Code 就知道该怎么渲染某个树节点了。第二个就是 **getChildren**，这个也很好理解，就是返回一个树节点的所有子节点的数据。在介绍样例代码之前，我们再看看 TreeItem，也就是每个树节点的数据结构：

```
export class TreeItem {
 label?: string;
 id?: string;
 iconPath?: string | Uri | { light: string | Uri; dark: string | Uri } | ThemeIcon;
 resourceUri?: Uri;
 tooltip?: string | undefined;

 /**
  * The command that should be executed when the tree item is selected.
  */
 command?: Command;

 /**
  * TreeItemCollapsibleState of the tree item.
  */
 collapsibleState?: TreeItemCollapsibleState;
 contextValue?: string;
 constructor(label: string, collapsibleState?: TreeItemCollapsibleState);
 constructor(resourceUri: Uri, collapsibleState?: TreeItemCollapsibleState);
}
```

TreeItem 有两种创建方式：第一种，就是提供 label，也就是一个字符串，VS Code 会把这个字符串渲染在树形结构中；第二种就是提供 resourceUri，也就是一个资源地址，VS Code 则会像资源管理器里渲染文件和文件夹一样渲染这个节点的。

至于其他属性：

- iconPath 属性，是用于控制树节点前的图标的。如果说我们自己通过 TreeView API 来实现一个资源管理器的话，就可以使用 iconPath 来为不同的文件类型指定不同的图标。
- tooltip 属性，当我们把鼠标移动到某个节点上等待片刻，VS Code 就会显示出这个节点对应的 tooltip 文字。
- collapsibleState 是用于控制这个树节点是应该展开还是折叠。当然，如果这个节点没有子节点的话，这个属性就用不着了。

- command 属性，如果有这个属性的话，当我们点击这个树节点时，这个属性所指定的命令就会被执行了。

了解了以上几个属性，就能够实现一个简易的 TreeView 了。下面我们就来看看一段示例代码：

```
vscode.window.registerTreeDataProvider('myextension', {
    getChildren: (element) => {
        if (element) {
            return null;
        }

        return ['first', 'second', 'third'];
    },
    getTreeItem: (element) => {
        return {
            label: element,
            tooltip: 'my ' + element + ' item'
        }
    }
})
```

上面的这段代码，注册了一个名为 myextension 的 TreeView，这个 TreeView 只有一层节点，它们分别是 ‘first’、 ‘second’、 ‘third’。

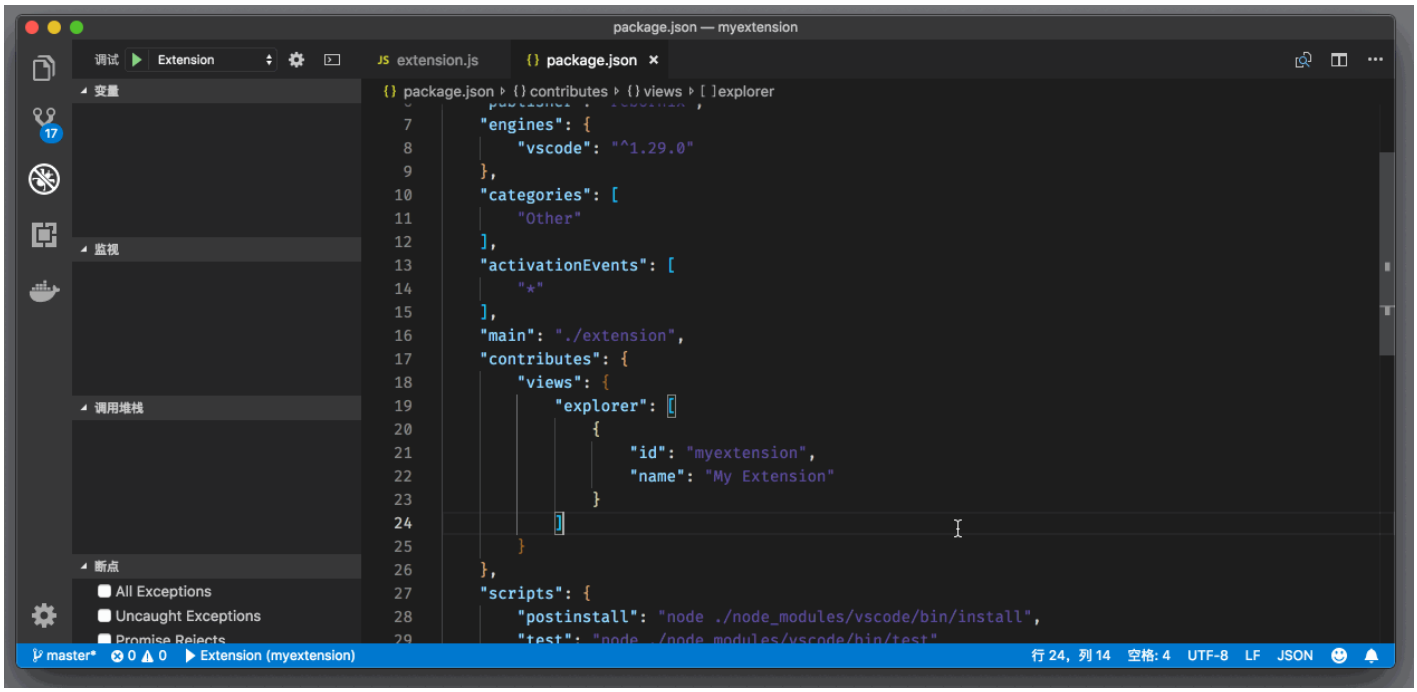
当我们将这段代码放入 extension.js 中时，运行插件，你会发现，VS Code 的视图里找不到这个名为 myextension 的 TreeView。这是为什么呢?

```
const vscode = require('vscode');

function activate(context) {
    vscode.window.registerTreeDataProvider('myextension', {
        getChildren: (element) => {
            if (element) {
                return null;
            }

            return ['first', 'second', 'third'];
        },
        getTreeItem: (element) => {
            return {
                label: element,
                tooltip: 'my ' + element + ' item'
            }
        }
    })
}
exports.activate = activate;

function deactivate() {
}
exports.deactivate = deactivate;
```

不知道你还记不记得，之前我们在 extension.js 里创建 extension.sayHello 命令，同时，我们还在 package.json 的 contributes 部分申明了这个命令。没错，**要想将这个 TreeView 成功地注册到 VS Code 中，你还得在 package.json 的 contributes 部分添加 TreeView 的申明**。修改后的 package.json 的 contributes 部分如下：

```
{
    "contributes": {
        "views": {
   "explorer": [
    {
     "id": "myextension",
     "name": "My Extension"
                }
            ]
        }
    }
}
```

这段 contributes 是说，我们把 myextension 这个 TreeView 注册到资源管理器中。好了，下面我们运行这个插件：

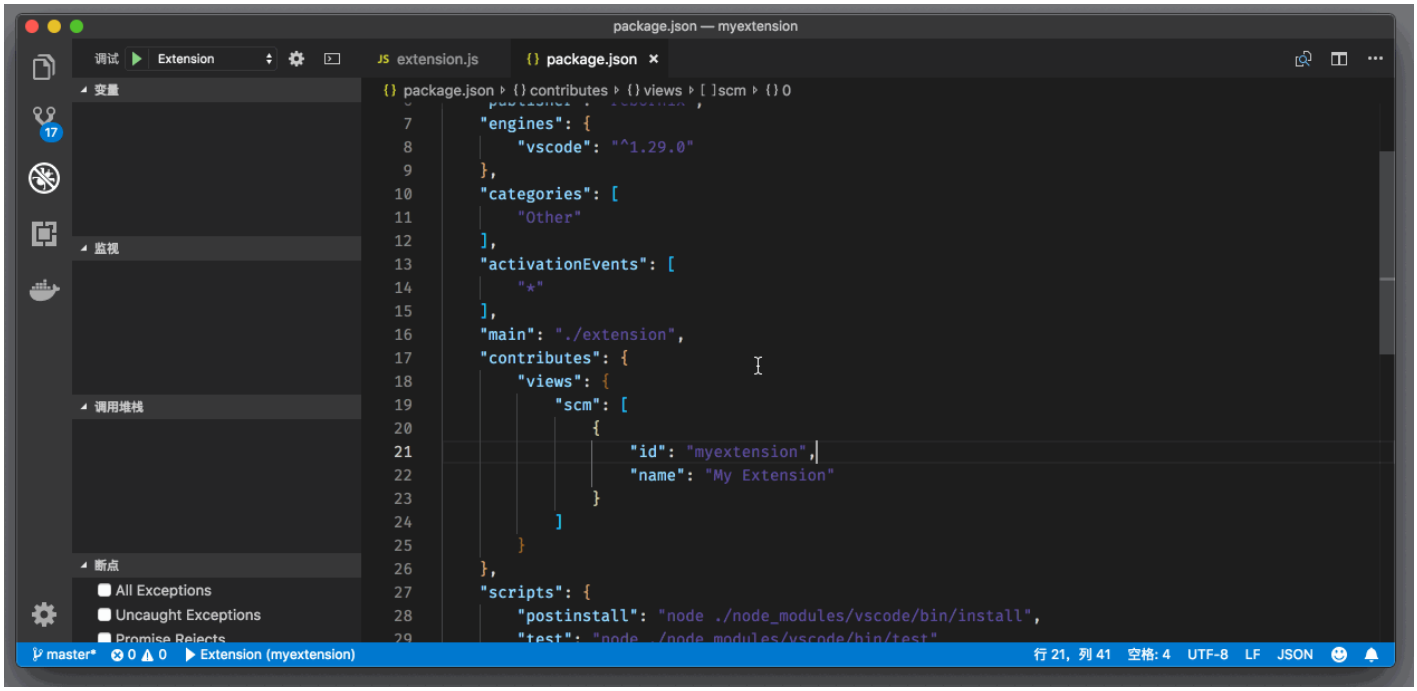


将 TreeView 注册到资源管理器中

除了将 TreeView 注册到资源管理器 Explorer 下以外，我们也可以将它注册到版本管理视图中，对应的 contributes 如下：

```
"contributes": {
    "views": {
        "scm": [
            {
                "id": "myextension",
                "name": "My Extension"
            }
        ]
    }
}
```

代码运行起来后，我们就能够在版本管理视图中看到这个 TreeView 了。



将 TreeView 注册到版本管理视图中

简言之，VS Code 的 TreeView API 使用了 Data Provider 的模式，插件提供数据，而 VS Code 负责渲染。至于数据长什么样、树形结构里的层级关系如何，这个就属于 Business Logic 了，需要大家发挥想象力。比如说 GitHub Pull Request 插件，用树形结构来展示所有的 Pull Requests 和每个 PR 里的代码改动，NPM Explorer 则将所有的 NPM 脚本展示在树形结构中。你也可以想想有什么别的有用的信息可以放在视图中呢?

小结

好了，以上就是今天内容的全部了。今天我们介绍了三类 API：信息提示和QuickPick，面板类 API， 以及视图 TreeView。

除此之外，VS Code 还有很多别的有趣的工作台相关的 API，比如你可以使用 WebView API 来生成任意的编辑器内容，可以使用 FileSystemProvider 或者 TextDocumentContentProvider 来为 VS Code 提供类似于本地文件的文本内容。虽然它们很小众也更高级，但是使用的方法，跟上面提到的几种并没有什么区别，建议你通过 VS Code 的 typings 文件找寻你想要使用的 API，多多尝试。

精选留言

谢mingmin
vs code的扩展API文档感觉目录结构很乱啊，有考虑优化一下结构吗?
2018-12-04 07:52

吉泓铭
可以增加一些应用场景的列举吗。
2018-12-04 11:50