**面向实时协同Web前端开发环境的代码重构机制与工具**

1352856 马奥宇

目录

1. 项目简介
   1. 项目环境简介
   2. 项目目标与用途
2. 项目背景及选型
   1. web前端开发环境
   2. 实时协同开发环境
   3. 代码重构
   4. 项目选型
3. 项目架构

3.1 开发准备：atom插件使用简介

3.2 生成插件框架

3.3 项目结构简介

3.4 函数调用重构

3.5 文件引用重构

3.6 集成实时协同环境

1. 项目实现与技术细节
2. 实验

5.1 实验环境

5.2 功能测试

5.3 性能测试

1. 总结
2. 项目简介
   1. 项目环境简介

实时协同编程环境是解决多个程序员协同开发同一项目的方案之一，相比现在主流的版本控制方案，实时协同环境的操作传送与集成都是由底层系统自动完成，无需程序员手动执行相关操作，也不会发生版本控制方案里多个程序员在一次提交中同时编辑了同一文件而导致需要手动解决冲突的额外开销。这种新的方法与技术使程序员只关注开发相关事项不用关注代码合并等繁琐事项，能显著提高软件项目的生产效率与质量。

随着Web 2.0概念的普及和W3C组织的推广，网站的前端由此发生了翻天覆地的变化，DHTML的影响力正以惊人的速度增长，相应的web前端开发需求越来越多也越来越受到重视，因此web前端开发环境也从便捷易用开始上升到追求功能强大。

在软件项目的开发与优化中，代码重构是不可避免的，而方便实用的代码重构功能将大大提升程序员的开发效率，同时保证重构后的代码质量，因此一个优秀的代码重构工具对于开发者来说是必不可少且重要的。

* 1. 项目目标与用途

本毕设项目考虑以web前端开发环境为基本使用场景，以多个程序员在实时协同编程环境下可能的代码重构需求为具体使用场景，分析每种具体场景下的代码自动关联更新机制，并在一种具体的前端开发常用IDE下以插件的形式在实时协同编程环境基础上实现这些重构功能。

Web 2.0时代后，用户对页面的美观与交互易用、开发者对前端项目的架构理解与模式设计都发生了翻天覆地的变化。前端技术与新框架层出不穷，前端开发变得越来越复杂也越来越受到重视，而不再是与后端开发无法相提并论的“初级入门”技术。在这种技术快速迭代的趋势下，前端开发的IDE虽然种类繁多但都不像后端开发IDE那样“重”本身已集成好各种功能。因此，我们考虑选择一个合适的主流前端开发IDE，在实时协同编程环境下，分析前端开发可能的代码重构场景，以此研究抽象出前端代码重构机制并在该IDE上实现。这个代码重构工具对于前端开发人员来说是非常有意义的，可以显著提升前端项目的开发效率与质量。

1. 项目背景及选型

2.1实时协同开发环境

实时协同编程环境允许一组程序员以紧密耦合的方式并行编辑同一组源代码文件与目录，且多名程序员对代码的修改可以被即时传送与合并；而且，这些实时的操作传送与集成是由底层系统自动完成的，无需程序员手动执行相关操作。在实时协同编程过程中，多名程序员可以在同一时刻并行访问与编辑共享源代码副本的任意部分，甚至并行编辑同一个源代码文件的内容。实时协同编程是一种新的方法与技术，具备一系列益处，例如减少程序设计错误、产出更好的设计与代码、加速解决问题的进程、促使程序员更加享受工作的快乐，以至于提高软件项目的生产效率与质量。

在实时协同编程过程中，每位程序员的编辑操作都会被实时传送至其他协作站点，作用于所有客户端的源代码副本；在此过程中，每位程序员都可以实时看到其他程序员对代码的最新改动（如同所有程序员使用同一台计算机开展编程工作）。

2.2 web前端开发环境

web（World Wide Web）即全球广域网也称万维网，它是一种基于超文本和HTTP的、全球性的、动态交互的、跨平台的分布式图形信息系统[1]。随着移动终端的普及与web2.0的强大特性，web应用因其跨平台的特性越来越受到欢迎，不论是windows系统还是Mac系统、不论是dell个人电脑还是iPhone同一个web应用通过浏览器都可以完美运行，这极大减轻了开发者的负担。

Web开发从视图与数据的角度来看也分为前端和后端，前端负责视图的呈现和用户交互，后端负责业务逻辑处理和与数据库交互。随着前端项目的复杂化以及用户终端硬件的变革，开发者对于前端开发设计模式的思考也在经历着快速变革，前端代码的模块化、框架化、按需加载、依赖管理等等都在规范化，前端开发的功能与效率都在不断提升，因此Web前端越来越受到开发者的喜爱和重视。

Web前端项目开发的日益复杂化迫使人们不断提出架构和设计模式的改变，从传统mvc架构到angularJS框架为代表的mvvc、React框架为代表的Virtual DOM等等，而这些框架的出现一方面促进了前端开发的发展提升了前端开发的效率，另一方面也使前端工程像后端工程一样走向了模块化、工程化。相应地，前端项目对于IDE的需求也在逐渐变得更高，这种发展趋势让前端项目也有了代码重构的需求。

2.3 代码重构

web前端自身的发展趋势给IDE提出了代码重构的需求，同时这也是实时协同环境能正常运行的一个重要保证。实时协同环境在提供一系列好处的同时，也带来一个严重问题：每当一位程序员对源代码的某个代码域（code region）作了符合语法规则的更改但由于其他部分的源代码对该代码域有依赖而产生语法错误时，其他程序员无法开展正常的编译与调试工作，只能持续地编辑源代码，无法尽早通过编译与调试发现问题，背离了程序员应有的工作方式，给编程工作的质量与效率带来负面影响。

所以本项毕业设计提出以实时协同Web前端开发环境为应用场景（一组程序员实时协同编辑HTML、CSS、JavaScript等文档），设计一套针对上述问题的源代码智能重构机制，并以此为基础实现一个具体的软件工具。该源代码智能重构机制的核心是自动关联更新（automatic linked update）：每当程序员对源代码的某个代码域作了符合语法规则的更改，协同工作系统自动地对那些依赖该代码域的源代码执行关联更新。

具体场景主要考虑两类：一是当某位程序员改动了其正在编辑的方法的名称，源代码中（无论是同一源代码文件还是其他源代码文件）所有依赖该方法的代码元素（例如调用该方法的语句）都被自动更新，以保证该项更改不会导致源代码产生语法错误。二是当某位程序员需要重构项目结构而对某个文件路径进行改变时(更改文件名或更改文件位置)，整个项目所有源代码中对该文件的引用都被自动更新，以保证该文件名字或者位置的更改不会导致源代码产生引用错误。

* 1. 项目选型

完成实时协同Web前端开发环境场景下的源代码智能重构机制的分析与设计后，以此为基础，本项毕业设计将以代码编辑器插件的具体形式实现一个软件工具，以此帮助前端开发者更好地在实时协同环境下进行代码重构。

我们选择了4种现在前端开发主流的IDE：Sublime Text、Visual Studio Code、Adobe Brackets和Atom作为该软件工具候选的代码编辑器，主要从编辑器是否开源、提供的API与文档完善度、开源社区活跃性、编辑器性能等几个方面分析比较如下：



Sublime Text优点是性能很好运行速度快，但是不开源且收费，对插件开发者不友好，不利用本项毕业设计。VSCode是微软基于atom改进的编辑器，其设计理念是提供用户一个性能强大、功能完备、质量保证的IDE，而不是鼓励用户自己编写插件扩展所需功能，官方很晚才提供插件开发API，文档与社区支持性都一般，所以VSCode从单纯用户使用角度是很有竞争力的，但是定位与本毕业设计的插件开发也不特别相符。Adobe Brackets 是Adobe公司推出的一款开源编辑器，对Adobe系列软件支持很好很适合偏设计的前端开发人员使用，如可以用Brackets 直接打开一个Psd文件，然后进行颜色和宽度的选择等。Brackets本身提供了丰富的API和详细的文档供插件开发者使用，但是社区活跃性一般不如atom，因此最终本项毕业设计选择Github公司开源的编辑器atom作为插件开发平台。

Atom 原先是Github公司内部使用的编辑器，现在全部开源且鼓励用户按需使用或编写自己所需插件以获得更适合自己的功能。Atom被官方自称为“为 21 世纪创造的可配置的编辑器”，其最显著也是最重要的特点就是插件化架构，对 Atom 而言最重要的是灵活而又完备的插件 API，多次官方新版本发布都在不断改进API与相应文档的支持。atom开发维护团队不仅提供了强大易用的API与使用文档，同时因为与Github深度整合让开源的力量发展迅速，开源社区成长速度极快且十分活跃，包括中文社区都是同类编辑器中最活跃的。社区驱动的方式让atom搭乘上了开源的快车，用户不仅可以共享每年数以万计发布的各类功能的插件，而且还可以按自己喜好改进甚至很方便地开发自己需要的插件。综上，基于atom对Web平台开发者友好对前端更友好，Hackable的显著特色让任何人都能简单贡献自己的插件，最终选择在atom平台下完成实时协同Web前端环境下的源代码重构插件开发。

1. 项目架构

3.1 开发准备

3.1.1 atom插件使用简介

在开发插件之前，我们首先得明白atom上插件是如何被安装关联到编辑器。下载好atom后，我们还需要同时安装上apm(atom package manager)才能对atom插件包进行管理。命令行输入apm –v查看apm是否正确安装：



然后在自己开发或者其他开发者开发的插件代码工程目录下使用apm link命令，即可将此插件link到atom的插件库中(~/.atom/packages)，从而将插件关联到编辑器，实现atom的功能扩展。安装成功后，在atom上打开manage packages的setting页，即可看到installed packages的数量相应增加，同时在community packages目录下可以看到安装的非atom自带插件的详细信息与插件管理设置功能。最后启动插件，atom提供4种方式用于启动插件：一是快捷键启动，二是atom控制面板输入相应启动命令启动，三是通过导航栏菜单选择相应插件启动，四是通过右击菜单选择相应插件启动。也可在插件管理页面将插件设置为随编辑器自启动，但是要注意不同插件之间启动命令不能相互冲突。

3.1.2 插件开发工具与语言简介

Atom 文本编辑器是基于 WEB 技术（Chromium+Node.js）开发的，可以理解为编辑器本身就是一个跑在本地的网页。因此我们将完全使用 JavaScript 的 EcmaScript 6 规范来制作插件，至于插件实现功能所需要的样式(如弹出框尺寸、光标选中某代码域后改变该代码域颜色等)可用css或less实现。

另外，前面所述本毕业设计考虑的两大重构场景中的函数名重构场景需要对代码进行词法分析与语法分析，考虑本项目的工作量合理安排与项目质量保证，该parse功能我们使用atom社区成熟的开源方案’refactor’，本项毕业设计再根据需要重构语言的特性与具体使用场景实现该开源方案的接口，完成重构功能。

3.2 生成插件框架

atom是如此地鼓励社区进行插件开发，因此对于插件开发者不仅提供完备易用的API与文档，甚至于已经把插件开发框架集成到atom自身，所以我们可以很方便地使用 Atom 提供的工具创建一个新的 package（软件包）来开始插件的开发。

启动atom编辑器后，打开控制面板(command+shift+p)，输入“Package Generator: Generate Package”并点击列表中正确的条目，然后在输入提示框中输入该插件软件包的名称，即可生成插件开发的框架代码。

另外，在atom插件开发中需要注意的一点是，每当需要测试我们对软件包的改动的时候，都要重新加载当前窗口以确保 Atom 执行的是我们最新的源代码。atom强大的控制面板允许我们很方便地重载编辑器，在控制面板输入window:reload即可完成。

3.3 项目结构简介

通过atom自带的package generate生成的插件工程基本结构如下：



3.3.1 定义主文件位置

主文件是 Atom 软件包的入口文件，Atom 通过设置 package.json 里的main属性来找到主文件的位置：



3.3.2 keymaps文件夹

用于配置插件快捷键，对应3.1.1部分插件管理的第一种方式。在keymaps文件夹下自动生成了一个json配置文件co-refactor.json，然后在该json文件指定快捷键与对应指令即可，如以下配置可以让用户通过 Ctrl+Alt+O（Windows/Linux） 或 Cmd+Alt+O（MacOS） 来触发指令"co-refactor:toggle"(对atom指令的介绍见3.3.4)：

{

"atom-workspace": {

"ctrl-alt-o": "co-refactor:toggle"

}

}

3.3.3 menus文件夹

用于配置右击菜单和导航栏菜单，对应3.1.1部分插件管理的第三、第四种方式。在menus文件夹下自动生成了一个json配置文件co-refactor.json，其中主要有两项配置内容，一是“context-menu” 对象可以让我们定义右击菜单的一些新条目，每一个条目都是通过一个显示在菜单的 label 属性和一个点击后执行的命令的 command 属性来定义的，如定义右击菜单的“Toggle co-refactor”条目对应指令"co-refactor:toggle"；二是“menu”  对象用来定义插件的自定义应用菜单，通过 label 属性和submenu属性定义该自定义应用菜单层级，最后通过label 属性和command属性定义点击该菜单后执行的命令，如图：

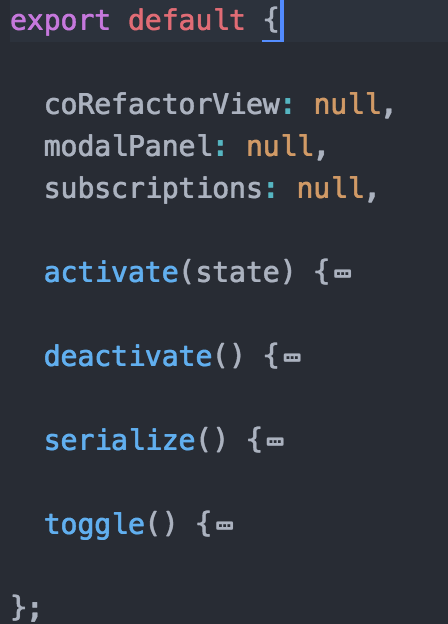




3.3.4 lib文件夹

主要包括插件主文件和处理业务逻辑的js文件，一般而言在项目package.json文件中通过main属性定义的插件主文件都在该文件夹下，如3.3.1所述该generate框架项目就是在package.json文件中通过"main": "./lib/co-refactor"定义该插件主文件为lib文件夹下的co-refactor.js。

主文件co-refactor.js导出一个带有生命周期函数（Atom 在特定的事件发生时调用的处理函数）的对象，主要包括4个模块：



* activate: 在 Atom 初次加载软件包的时候调用，用来初始化一些诸如软件包所需的用户界面元素的对象，以及订阅软件包命令的处理函数。如定义'co-refactor:toggle'命令对应'toggle'函数：

this.subscriptions.add(atom.commands.add('atom-workspace', {

'co-refactor:toggle': () => this.toggle()

}));

* deactivate: 在软件包停用的时候调用，例如，当用户关闭或者刷新编辑器的时候。
* serialize: Atom 会在使用软件包的过程中不断调用它以保存软件包的当前状态，然后将返回值在 Atom 下一次加载软件包的时候作为一个参数传递给上述activate模块。
* toggle: 为activate模块中定义的'co-refactor:toggle'指令对应函数，也就是插件启动入口函数，后续插件功能扩展由此开始。

3.4 代码调用重构

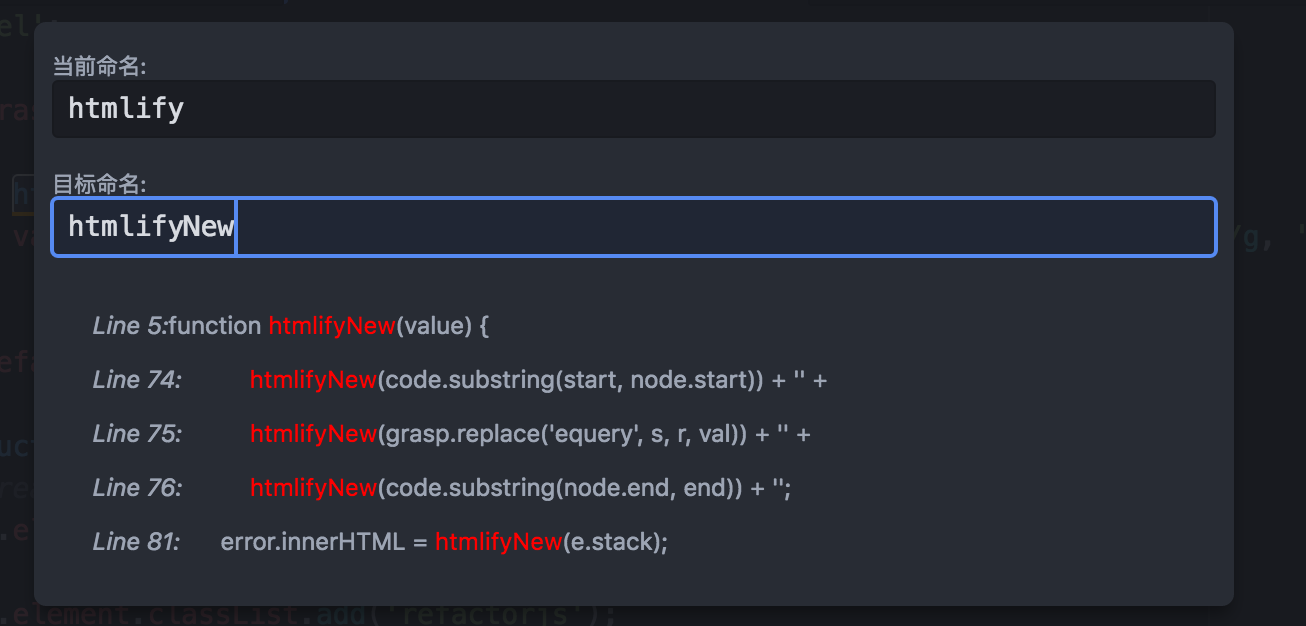
代码调用重构功能主要包括两部分：函数调用重构和变量重构，即用户选择某个函数或变量进行重构时，该文件中所有对这个函数或变量的调用都会自动关联更新。代码调用重构主要包括指令触发、获取所选重构对象、构造重构构件、搜索重构对象等模块，功能架构如下：

3.4.1定义指令触发方式

考虑用户更改代码中函数名或变量名时的使用场景，使用快捷键的方式是最方便，因此我们选择用快捷键的方式启动重构指令与提交重构指令（注意不同插件之间快捷键不要冲突）。其中用户选中重构对象后使用快捷键Cmd+Alt+R启动重构指令，显示重构构件；用户输入目标重构命名后，使用快捷键Cmd+Enter启动提交指令，执行重构。

3.4.2 构造重构构件

atom接收到代码重构指令后，生成一个重构构件用于接收和显示重构信息。该构件包括最基本的两个label显示重构对象当前命名和目标命名，为了增强这个工具的可视化，我们还增加一个list控件实时显示当前重构对象及其上下文：



3.4.3 使用grasp协助重构

grasp[4]是一个分析js代码的工具，用于协助开发者对js代码进行完全语法树(the abstract syntax tree)的构建与分析，可以很方便且高效地完成js的结构化分析与查询。在本毕业设计中我们使用grasp来协助完成对js代码的重构，包括查询重构对象在代码中的位置及上下文，以及完成最后的替换重构。

3.4.4 提交重构请求

用户在重构构件中输入重构对象目标命名后，构件中的list模块就会实时显示与此重构对象有关联的代码上下文并相应改动，但是这些改动只是为了方便用户直观掌握代码结构而在这个控件中进行的，事实上程序还没有真正去项目工程里重构代码。

若用户确认进行这次重构，还需要再进行一次提交操作，3.4.1中我们已经定义了使用快捷键Cmd+Enter启动提交指令，即对工程里代码真正进行重构；当然若用户想放弃这次重构，也可以使用esc健退出这次操作。

3.5 文件引用重构

文件引用重构功能是指，当用户改变一个文件的文件名或位置时，工程项目所有引用了该文件的代码都会相应被重构。文件引用重构功能主要包括指令触发、获取当前文件路径、构造重命名构件、状态记录、文件移动、代码引用重构几个模块，功能架构如下：

3.5.1 定义指令触发方式

考虑用户更改文件名或文件位置时的使用场景，将“文件引用重构”指令的触发方式定义到右击文件后的菜单的”refactor”项：



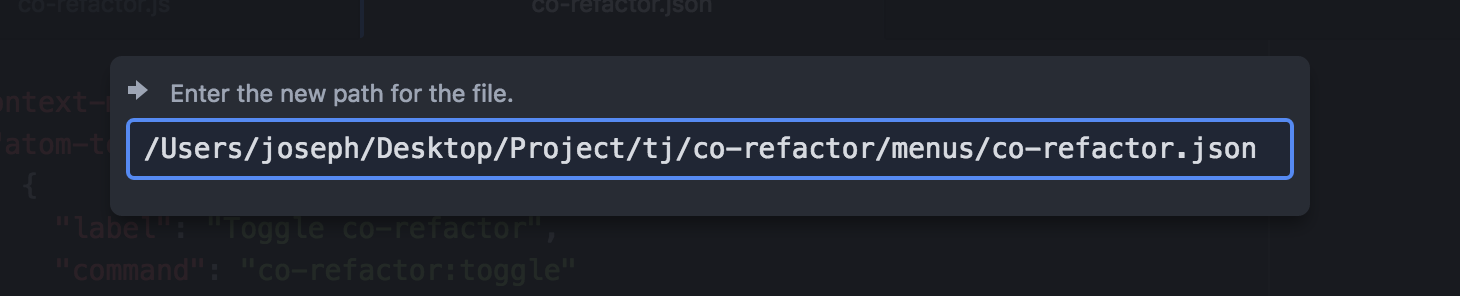
用户选择要重构的文件，右击该文件后选择Refactor选项即可调用文件引用重构功能。

3.5.2 获取当前文件路径

选择右击菜单的Refactor选项后，首先应该获取到该文件的路径并记录保存，具体实现见4.2.2。

3.5.3 构造重命名控件

获取到当前文件路径后，生成一个重命名控件PathRenameForm接收更改路径信息，默认显示当前路径并允许用户更改。



3.5.4 记录状态与触发重命名函数

用户输入目标路径后，若选择确认则触发核心的重命名函数。

重构的过程中，首先要记录当前重构的文件路径与欲更改的目标路径，然后与工程里所有文件交互匹配、重构所有引用，因此使用Redux生成store对象记录每个状态与对应视图。

重构开始时传入'refactor-start'状态到store，重构状态为正在进行，然后执行核心函数renamePaths，与工程所有文件匹配结束后，传入'refactor-end'状态到store，重构状态为结束。

3.5.5 重命名函数架构

重命名函数renamePaths是本毕业设计的核心函数，按业务处理逻辑可以分为4部分：

1.判断重命名的对象：若重命名的对象是文件夹，则遍历该文件夹下所有文件，生成一个json数组保存该文件夹下所有文件的当前路径和目标路径；若重命名的对象是单个文件，则只将该文件当前路径和目标路径存放到json数组。

2.判断目标路径是否存在：遍历上一步得到的json数组，若存在某个文件的目标路径已经存在，则无法进行重构，提示相应错误信息；

3.移动文件：若第2步中所有目标路径都有效可用，则调用atom的文件移动API：'fs-plus'，根据第1步的json数组将所有文件移动到目标路径。

4.重构所有引用语句：json数组里所有文件都被移动到目标路径位置后，使用Facebook开源的构件Codemod和运行 codemod 的工具包jscodeshift，对整个项目工程进行代码结构分析，并相应更改项目里所有被重构文件的引用。

3.6 集成实时协同环境

在3.4与3.5中完成web前端开发环境的代码调用重构和文件引用重构模块后，接下来还需要把实时协同环境集成到项目中，以到达实时协同环境下的代码重构。

本毕业设计使用指导老师另外一位学生的项目co-editor[6]作为实时协同环境的基础，本毕设与该实时协同项目为同一课题下的两个子项目，都是基于atom插件的形式进行开发，因此在基本环境集成方面没有大问题，主要难点还是在集成过程中的操作转换(Operation Transformation)上。

操作转换技术是实时协同系统最常用的技术之一，在实时协同中多个终端用户可能同时编辑一个文件，这时候操作执行的顺序就尤为重要，如用户a执行删除第6行的操作与用户b执行在第5行后新添一行的操作同时进行，若执行顺序不对则可能把用户b新添的行误删了。在将重构系统与实时协同系统整合的过程中，也同样存在这样的问题，需要考虑操作的转换顺序。

如用户a重构某函数的命名时，a的终端会向协同服务器同步启动重构的快捷键变化，导致用户b同步这个指令后也在b的终端执行重构；然而，用户a的重构完成后，代码域的改动又会同步到服务区导致用户b的代码域又一次改动。因此在集成整合实时协同系统时，我们要保证操作的转换不影响重构的正确性，即重构的指令和结果不能相互覆盖或重写，为避免这种情况的发生，我们在实时协同系统中就只同步重构的结果(text)，不同步重构指令的事件(event)。

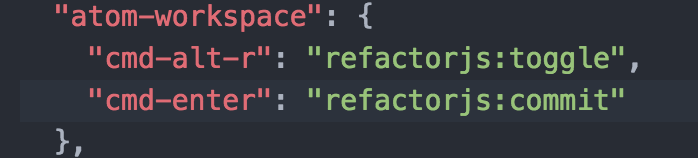
3.7 测试

1. 项目实现与技术细节

4.1代码调用重构

4.1.1定义指令触发方式

如3.3.2所述，对快捷键的配置在keymaps文件夹下的json文件里，指定相应快捷键对应的指令即可：



然后在入口主文件的activate函数里定义这两个指令的对应执行函数，接下来就可以在对应执行函数里编写所需业务逻辑代码了。

4.1.2 构造重构构件

为了方便，定义执行一次快捷键组合Cmd+Alt+R后启动插件，再执行一次退出插件。因此首先通过一个三目运算符this.modalPanel.isVisible() ?exit():start()实现根据插件当前运行状态来启动或退出，若启动则生成重构构件RefactorjsView，显示和输入重构信息。

该重构控件由两部分组成，一是显示重构对象的当前命名和目标命名的form，另外还有由重构对象在代码中出现位置上下文组成的list。

Form控件包括4项：searchText、searchInput、replaceText和replaceInput。其中searchText为一个提示message显示'当前命名:'，searchInput显示通过atom的editor API获取的重构对象：

var editor = atom.workspace.getActivePane().getActiveEditor();

var s = editor.getSelectedText() || editor.getWordUnderCursor();

this.searchInput.getModel().setText(s);

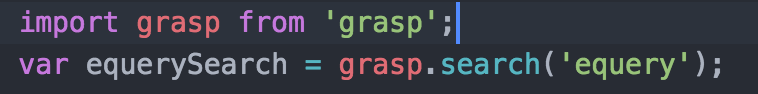
replaceText为一个提示message显示'目标命名:'，replaceInput为一个可编辑框接收重构对象目标命名，默认显示重构对象当前命名，为了在后面的list组件实时显示重构对象在代码中的位置及上下文，还要将replaceInput的改动事件绑定到list组件：

this.replaceEvent = this.replaceInput.getModel().onDidChange(this.search.bind(this));

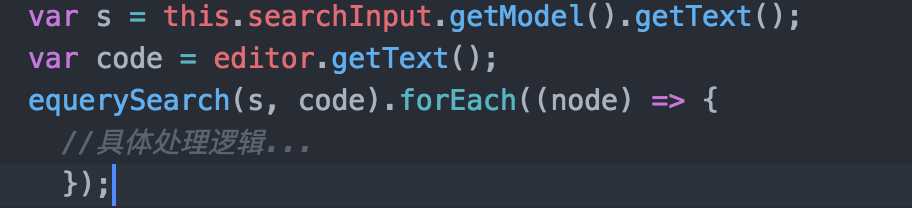
最后根据form控件中replaceInput的输入来实时匹配代码，并将重构对象出现的代码上下文实时显示，具体实现细节见4.1.3。

4.1.3 使用grasp协助重构

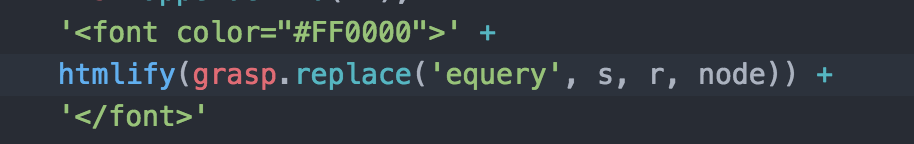
使用grasp生成的完全语法树可以很高效地进行js代码的分析，当前最新版本的grasp支持js和css两种语言，因此首先根据API文档选择js的语法树构造模式equery：



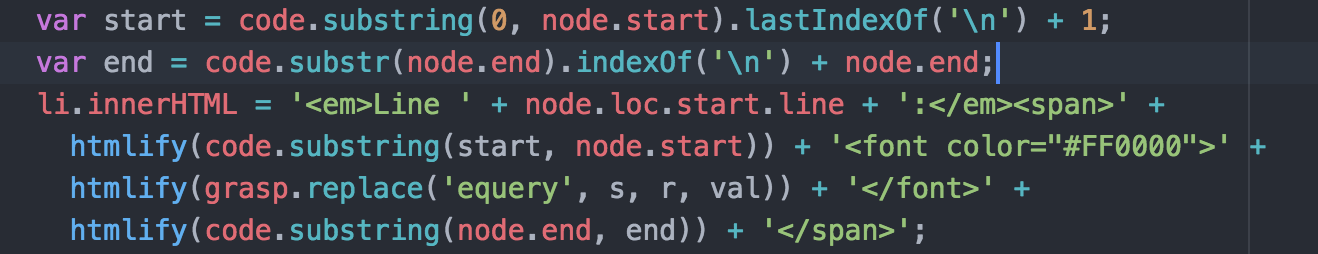
然后获取前面replaceInput输入框的值s和整个文件的代码code，再使用grasp的js模式equerySearch构造出目标代码的完全语法树并得到目标对象s出现的每一个相关上下文node:



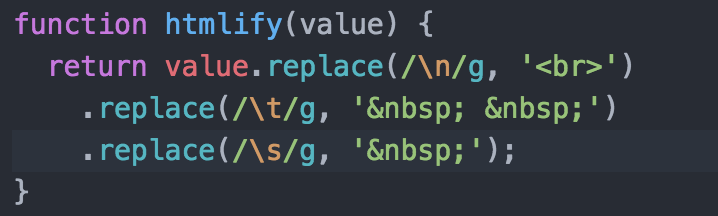
通过grasp得到重构对象在整个代码域中的上下文node后，首先根据接口文档使用node.loc.start.line方法得到该上下文在整个代码中的位置；然后再使用grasp的全局重构方法replace，在equery模式下根据目标对象原始命名s和replaceInput输入的值r实时更改重构对象，为了更直观我们将重构对象颜色设置为亮红色：



再把重构对象所在代码行相应上下文加入一起显示，方便用户查看该重构对象在代码中的情况：



此外，因为atom是基于web技术开发的，在把代码显示到控件时要先把代码html化包括换行符的转换等，我们使用一个htmlify函数协助完成转换代码到html格式：

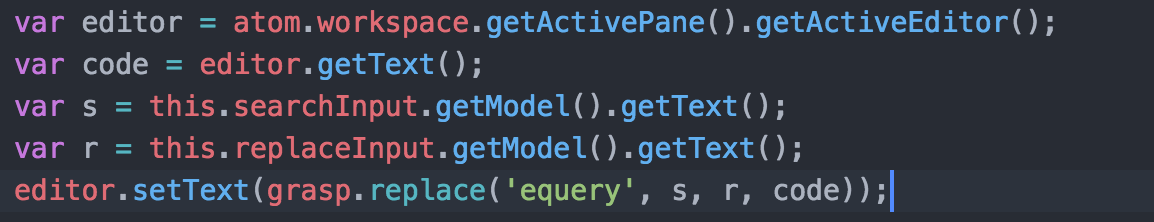


最后把分析每个重构对象出现的代码块node得到的结果都加入到list控件，即可在重构控件里实时显示重构对象所有出现的位置及上下文，方便用户分析、重构代码。

4.1.4 提交重构请求

在主文件中，我们已经指定了重构提交指令Cmd+Enter对应commit函数，因此提交请求相关的逻辑处理都在这个函数里实现。

首先获取atom下当前的编辑器editor及editor的整个代码域，然后再根据重构构件中searchInput和replaceInput控件的值得到重构对象原始命名与目标命名，最后在equery模式下调用grasp的全局重构方法replace对代码域完成重构，并将改动设置到editor：



4.2文件引用重构

4.2.1 指令触发方式实现

查阅atom的context-menu API可得在文件树状导航栏右击的代码为".tree-view .file, .tree-view .directory"，因此在menus/co-refactor.json里添加:

"context-menu": {

".tree-view .file, .tree-view .directory": [

{

"label": "Refactor",

"command": "co-refactor:refactor"

}

]

}

即实现将"Refactor"加入到右击树状导航栏中的文件出现的菜单选项中，并将该选项与指令"co-refactor:refactor"关联。

4.2.2 获取当前文件路径

atom是基于 WEB 技术（Chromium+Node.js）开发的，因此每个元素本质上都是一个HTMLElement，将点击的文件作为一个 HTMLElement传入相应API即可获取文件路径。

4.2.3实现重命名控件

PathRenameForm主要由一个提示输入的label和一个AtomInput组件构成，默认显示传入的路径并允许用户编辑目标路径，然后根据操作选择触发取消或确认函数。

Atom官方鼓励使用React来实现这样的构件，这个构件应包括获取的当前路径、取消操作、确认操作3个属性。Refactor操作触发后在主文件中使用ReactDOM.render将这3个属性传递给该控件：

this.el = document.createElement('div');

this.modalPanel = atom.workspace.addModalPanel({ item: this.el });

ReactDOM.render(

<PathRenameForm

previousPath={previousPath}

onClose={onClose}

onRename={onRename}

/>

, this.el);

PathRenameForm控件接收这3个属性后，首先调用构造函数，生成atom输入框并将属性里面的当前路径设为默认值显示；然后调用onChange函数记录用户当前输入为目标路径，最后根据用户操作选择执行传入属性的确认函数或取消函数：

'core:confirm': () => this.props.onRename({

previousPath: this.props.previousPath,

nextPath: this.state.path,

}),

'core:cancel': () => this.props.onClose()

这部分的主要技术难点是参考atom官方使用了React+Redux的函数式编程（FP）范式。区别于传统的面向对象模式，React+Redux的函数式编程（FP）范式更适用于多交互、多数据源的使用场景，而我们在进行文件名重构的过程正是要记录当前重构的文件路径与欲更改的目标路径，然后与工程里所有文件交互匹配、重构所有引用，所以使用函数式编程更易于实现功能。

4.2.4用Redux实现状态记录

首先通过Redux提供的createStore生成Store容器来保存数据：

import { createStore } from 'redux';

this.store = createStore(reducer);

其中reducer是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State指明重构是否完成：

export default function reducer(state, action) {

switch (action.type) {

case 'refactor-start':

refactorInProgress: true

case 'refactor-end':

refactorInProgress: false

}

}

在Redux中视图(View)通过store.dispatch() 方法发出 Action到前面的reducer，获取新的状态以更新视图。在本项目中，重构开始时发出'refactor-start'，结束时发送'refactor-end'，视图就会根据reducer返回的状态变化，以匹配所有文件。

4.2.5重命名函数实现

4.2.5.1判断重命名的对象

用户可选择对单个文件或者整个文件夹进行重构，所以首先要判断重命名的对象并根据所选对象不同执行不同操作。

Atom官方提供的'fs-plus'包[3]囊括了所有文件的基本操作API，查询API文档可得接口**isDirectorySync(directoryPath)**判断所给路径是否存在且是一个文件目录则返回true，否则返回false，应该可用于判断用户的选择是文件目录还是单个文件：

**if (isDirectorySync(previousPath)) {**

**//用户选择文件目录重构**

**}**

**else{**

**//用户选择单个文件重构**

**}**

若用户选择文件目录重构，则首先传入当前目录previousPath，使用'fs-plus'包下的接口**listTreeSync(rootPath)[3]**遍历该目录得到该目录下所有文件路径，然后拼接目标目录nextPath和当前目录下所有文件的相对路径，构成一个由所有原始文件路径prevFilePath和所有目标文件路径nextFilePath组成的数组：

return listTreeSync(previousPath).map(prevFilePath => ({

prevFilePath, nextFilePath:join(nextPath,prevFilePath.slice(previousPath.length)),

}));

若用户选择单个文件重构，则直接将该文件原始路径和目标路径加入数组即可。

4.2.5.2判断目标路径是否存在

得到存放所有目标路径的数组后，同样我们使用'fs-plus'包下接口**isFileSync(filePath)**，该接口判断若所给路径存在且是一个文件则返回true。

遍历数组里的所有目标路径，若**isFileSync(filePath)**返回了true则说明该目标路径下已有其它文件，不可进行重构；否则检查通过，进行下一步移动文件。

4.2.5.3 移动文件

查阅'fs-plus'包文档，atom提供了方便的接口**moveSync(source, target)**用于移动文件。上一步的路径存在检测保障了数组里所有目标路径都是可用的，因此直接遍历路径数组根据prevFilePath和nextFilePath，将所有原始文件移动到目标位置。

4.2.5.4 重构所有引用语句

被重构文件成功移动到目标位置后，最后一步需要做的就是将项目工程里所有引用了这些文件的引用语句重构。

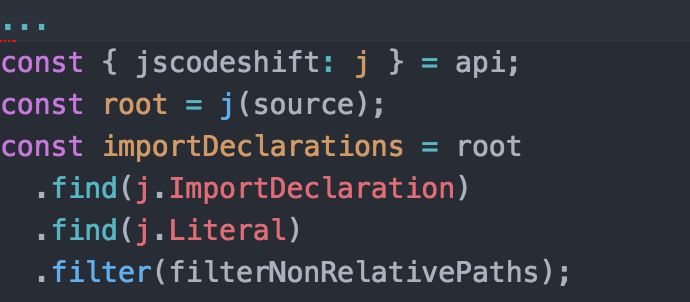
随着前端界的风起云涌，新功能、新框架、甚至 JavaScript 本身都在快速进化，为了让开发团队保持高效，业界也提出了很多使用重构件（Codemod）加速 JavaScript 开发的想法与框架，其中最优秀的当数Facebook的jscodeshift[5]。jscodeshift 是一个用于在多 JS 文件运行 codemod 的工具包，可用于协助大型代码库的自动化重构。此工具包将 JavaScript 解析为一棵 抽象语法树 ，并在其上进行变换，然后输出符合指定代码风格的新 JavaScript 代码，而且转换过程是用 JavaScript 本身实现的，它提供：

* 一个 runner，它可在每个传递给它的文件之间提供转换，还能输出转换文件的数量。
* recast 的包装，提供不同的 API。Recast 是一个 AST-to-AST 转换工具，并且会尽可能地保护代码的原有风格。

因此我们使用该工具包协助完成引用转换，根据jscodeshift提供的API我们可以很方便地完成代码里的声明引用重构：

首先引入runner：import Runner from 'jscodeshift/dist/Runner';

然后调用jscodeshift的声明引用API(ImportDeclaration)，封装为runner函数需要的fileInfo参数：



最后将声明引用API、路径参数、用户参数传递给runner函数，jscodeshift即可根据路径参数完成工程里声明引用的重构：

const result = Runner.run(transform, roots, options);

return Promise.resolve(result);

4.3 集成实时协同环境

实时协同项目co-editor使用ShareDB[7]作为项目的OT框架，ShareDB框架基于 Node.JS 实现，整合了前后端，可以很方便地自行搭建服务端和客户端。为了保证实时协同过程中重构的指令和结果不要相互覆盖或重写，ShareDB的客户端只监听重构的结果，不监听重构指令的事件：



启动实时协同环境前，首先需要指定server address、session id和client id，其中server address表示同步服务器的IP地址；session id用于标示文件路径，即第一位编辑某个工程项目的用户对该工程项目设置一个session id，后面的用户输入相同的session id就可以加入到这个工程的实时协同编辑中；client id用于标示用户，当多个用户编辑同一个文件时，光标上不同的client id就表示相应用户。

指定以上参数后，用户就可以开启或加入一个实时协同环境，在这个环境下用户可以进行自己的编辑，同时可以看到有黄色光标表示其他用户也在同时编辑这个文件：



1. 实验
2. 总结

参考文献：

w3c中国[http://www.chinaw3c.org/about.html]

https://github.com/atom/fs-plus

https://github.com/facebook/codemod/

<https://github.com/facebook/jscodeshift>

https://github.com/gkz/grasp/