代码编辑器系列 #0 两种编辑器

2018, Apr 27 by Tesla Ice Zhang

在看了那么久 IntelliJ IDEA 的架构后，自己心里总是痒痒的，想自己实现一个差不多的文本编辑器。 现在基础设施已经差不多全了（补全、基于 Parser 的带语义分析的和基于 Lexer 的基于 Token 的高亮、插件系统、撤销-重做系统）， 华而不实的东西也做了一些（拖放打开文件、背景图片、可以热更新的设置、自定义字体、 AST 阅读器（即 PsiViewer），还有一个初中小朋友写的查找替换、多行注释/取消注释等）， 项目叫 DevKt，代码在这里，可以直接在 release 里面下载试用。

在做了这个文本编辑器之后，我又学到了很多非平凡的常识（就是直觉感觉不出来，不去做一个你就不知道的常识，反正就是各种又硬核又冷门的知识）。 鉴于之前写过的几篇关于 IntelliJ IDEA / Eclipse 的文章实际上都比较水（看了只能浪费时间，笑一笑，看完并不能帮助你写出一个 DevKt 那样的文本编辑器，就像各种微信攻粽耗里的『Python 人工智能』教程一样），现在我决定出一个比较正经的系列来补偿一下之前被我浪费了时间的读者。 

正文

我眼中的文本编辑器分两大类， Office 那样的富文本编辑器和 IntelliJ IDEA 那样的代码编辑器。

前者自然是一个巨复杂的怪物（排版方面的），我不是这方面的专家所以就略过不谈了。

后者是一个很复杂的怪物，它也有很多分类，在我看来应该分为 Vim/VSCode 这类『只负责文本编辑』的和 IntelliJ IDEA/Emacs (由于不清楚 Visual Studio 实现，暂时不列入。根据我的个人推测，VS 也属于这一梯队) 这类。

代码编辑器

他们的主要区别在于，前者本质只是一个文本编辑器，对文本文件的读取可以进行优化，比如只读取一部分文本内容（按需读取），在用户滚动的时候继续读取并缓存高亮结果之类云云，可以做到打开大文件不卡等神奇功效。 这看起来当然是非常非常合理的优化，不过稍加思考就能发现它的缺陷——无法进行完整的代码分析。因为 Parsing 是需要拿到完整的文本的，而只读取一部分代码就无法正确地对代码进行分析了。举个很简单的例子，如果 Parser 拿到的都是注释的内容，上面和下面都拿不到，那么它是无法确定自己在解析什么的（Parsing 的结果是树形的而不是线性的）。

好吧，我现在感觉这个例子没什么意义。

这样的文本编辑器如果需要进行完整的代码分析，就需要借助其他程序（典型例子：VSCode 的 LSP ，Vim/Emacs 的 Agda/Idris 编辑模式）在读取了完整的文件的情况下进行语义分析，将分析结果借助 ffi 或者 http 等各种协议发到编辑器里，再进行 AST 叶子节点、报错的代码的高亮渲染，提供 quick fix 、补全。 这样

的代码分析无疑是非常低效的，不过可以带来十分不错的编辑体验。

不过在不需要这样的语义分析的时候，这种文本编辑器反而变成了最高效的（比如 VSCode 的 Kotlin 插件，只是提供了一个基于正则的高亮而已，菜的不像是人写出来的东西，但是就可以做到非常 efficient）。 假设现在插件开发者希望高亮是基于语义分析的，那么高亮的性能就会爆炸了。 

而 IntelliJ IDEA/Emacs/DevKt 这类编辑器选择一次性把文本全部读入，也就是同时承担了文本编辑器和前文所说的『其他程序』的工作。他们的高亮本来就是基于语义分析的，直接放弃了不完全的文件读取，保证最大化语义分析器的性能。定义跳转、quick fix 等都不需要经过各种协议传输，直接在代码里面传就是了，也减少了 `JSON.stringify` 等操作的成本。 这样的编辑功能会在性能上受到语义分析的拖累，造成一定程度的卡顿，可以通过将语义分析放进守护进程的方式解决编辑的卡顿问题，以及增量渲染在一定程度上解决绘制的性能问题（IntelliJ IDEA 绝对是这么做的，因为我现在使用比它稍微平凡一点的做法，遇到大文件都卡的不行）。

这种级别的文本编辑器对编辑器控件的要求非常高，比如增量的渲染之类的功能，还需要暴露非常多的底层 API（增量高亮有太多需要考虑的问题了。。。），比如 `JTextPane` 就只能在一定程度上满足需求。我们可以看到各家 Java 写的文本编辑器都自己造了轮子，IDEA 的我还没看过就暂时不说了，jEdit 的文本编辑器控件 `org.gjt.sp.jedit.textarea.TextArea` 就是直接继承了 `JComponent` 写的， 160 多 kb，控件源码可以在他们的 SourceForge 的 SVN 仓库 找到，不过代码风格不敢恭维（逃。

而这样的文本编辑器可以做到更实时的语义分析，更精准的高亮，更丰富的重构操作（毕竟 AST 都在内存里，可以直接找到在编辑器里的位置，很方便），以及最重要的，更好用的缓存。IntelliJ Platform 自己实现了一个文件系统，叫『虚拟文件系统』（Virtual File System），可以解决文件的外部改动和内部缓存的同步问题，使得缓存文件的 AST 成为可能（没有改动过的文件可以直接复用上次打开时构建的 AST，VSCode 没有这种功能，因此同样是巨大文件， IDEA 和 VSCode 第一次打开的时候都卡，但是如果文件没有改动，IDEA 之后打开就不卡了，VSCode 还是卡）。 如果没有这个设施，IDE 就无法保证内存里的 AST 和以后打开文件时的文件内容是否匹配（即，有可能文件会有外部改动），就不敢乱复用 AST，至少要对文件进行一次 Reparse，带来不必要的时间成本。

这样的文本编辑器当然也是可以借助外部工具进行代码分析的，比如 agda2-mode/idris-mode 。

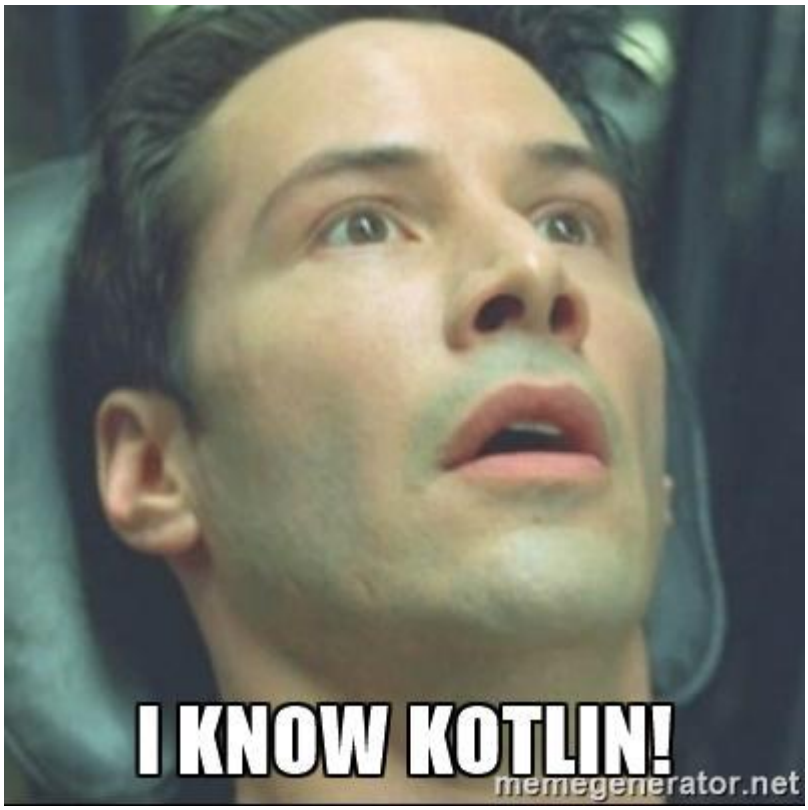
本文完

下面是吹水时间。

推荐阅读： VSCode、阿童木这些文本编辑器的代码实现中有哪些奇技淫巧？ justjavac 的知乎回答

里面提到了 VSCode 的 Lexer 的增量优化，但这其实比起 IDEA 那些增量更新就是小巫见大了（类似的优化 IDEA 是做不了的，因为在更新了 Token 串后还要对树形的 AST 进行增量更新，Reparse 肯定是免不了的了，对 AST 节点进行增量更新和语义信息进行增量更新才是最重要的。 Resolve reference/Index File Stub 的成本比 Parsing / Lexing 高多了，在有更大的问题时只能暂时忽略小的）

最后，能这么快构建原型，光靠有代码编辑器方面的先决知识是不够的，你还需要一门好用的工业语言。



（逃

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License