代码编辑器系列 #1 架构与解耦

2018, Apr 29 by Tesla Ice Zhang

在上一篇文章中我提到了两种代码编辑器—— Vim/VSCode 类和 IntelliJ IDEA/Visual Studio 类。 这两者分别是把代码分析的任务交给另一个程序，和自主进行代码分析。 微软已经为前者（ VSCode ）的这种策略所采用的协议起了一个名字，叫 Language Server Protocol，下文使用简称『**LSP**』。 然后自主代码分析的由于 JB 家的都属于标准的案例，就使用简称『**JB**』（和 VS 是一样的）。

代码编辑器一般都会考虑一个问题，即借助插件添加新语言支持（这里还涉及插件系统的设计，不过这暂时不是本文讨论的重点）。 显而易见的是，这里有一个代码分析器和编辑器前端的解耦问题。 编辑器开发者需要首先对代码进行抽象，把『代码分析』（本质就是一个 `[Library] -> Code -> ([Diagnostic], [QuickFix], [Highlight])` 的函数）交给插件开发者（以及内建语言支持），插件开发者进行代码分析，反馈分析结果，再由编辑器前端显示分析结果、代码高亮、提供 quick fix。

然后还可能有其他设施，比如往欢迎界面加东西，或者改右键菜单之类的杂七杂八的不难实现的特性。 这些东西就不说了，完全不难。

名词解释

上面提到了代码编辑器的两种解耦模式，分别是『编辑器只负责编辑，后台进程才看得懂代码语义』的可以不读完文件的 **LSP** 式，和『编辑器搞定一切』的必须读完文件的 **JB** 式。

这两种模式的定义是有绝对的界限的，但现在很多编辑器都可以说是同时具有两种特点（比如 Emacs 也有用编译器 server 的（ `idris-mode`，`agda2-mode`，`coq-mode` ）， IDEA 也有使用编译器 server 的插件），因此使用『是否自主进行代码分析』来定义是不准确的，目前我想到的一个比较好的区分点就是『是否可以不读完文件』。

这是因为，JB 式的编辑器显而易见可以使用 LSP（作为一个额外的代码分析器），而 LSP 式的编辑器则因为无法提供完整的代码而无法自主进行语义分析，所以我才使用『是否可以不读完文件』来区分二者，只是为了防止自己造出类似『函数式编程』这样的名词而已。

两种分析策略

LSP 和 JB 式分别将『代码』这一概念进行了两个不同层面的抽象。

根据上面给的 LSP 的 overview，可以看出 LSP 是通过给 language server 发送编辑造成的增量改动（ `range` 或者 `offset`, `text` ，分别对应插入和删除嘛）配合 language server 的增量 parsing 来实现更新的。 至于高亮（tokenize 同），要么从 language server 获取，要么用正则表达式或者 lexer。 可以看出，编辑器只知道插入和删除的 range ，代码对它来说只是一个一个的 token。

而 JB 式的编辑器会使用自己内置的 parser 对 AST 进行更新（可以做成增量的），代码对它来说是 AST，可以拿到元素的父子节点，知道代码的层级嵌套关系以及详细的语义。 如果能拿到完整的 code base（大部分情况下都能拿到，但也有特例，比如根据 vczh 的说法 office 的开发就拿不到，因为代码太多了。这种代码也不适合使用对代码进行完整分析的 IDE 开发，一般都是编辑器家族或者大型 IDE 关掉一些分析功能来写），就可以对代码进行完整的静态分析，比如 IDEA 可以根据函数里对参数的处理，推导库函数参数的 `@NotNull` 和 `@Nullable`（只要直接调用了上面的函数，或者传给了需要 `@NotNull` 的函数，那么就是 `@NotNull`；如果有什么 `if (参数 == null) return 默认值` 的语句，就会推断为 `@Nullable`，从而对错误的操作（比如传一个 `null` 给一个直接在这个参数上调用方法的函数，很明显的 `NullPointerException`）进行报警），寻找命令行报错信息中的文件名和行号进行并提供跳转到出错地点等。

LSP 的优点是，编辑器和代码分析是完全分离的， parsing 可以直接复用编译器（然后还需要实现增量更新 AST），而且可以轻易地保证 parsing 和编译器的行为完全一致。 不过，任何涉及语义的操作都必须传 json，遇到 parser 很慢的语言或者大量使用编译期计算的语言可能会把 language server 卡掉导致语义分析不可用。 重构功能也难以提供，因为涉及大量的 AST 操作。 而内存效率本身也不高，毕竟编辑器本身和 language server 都在读取代码、分析代码，保存了不必要的多份拷贝。

JB 系的优点是，插件可以轻松进行自主代码分析和重构。AST 是以结构化数据存在于内存中，读取成本很低。 比如，expansion selection（IDEA 默认快捷键 Ctrl + W）这样的操作实际上只是需要取 AST 的父子节点，而这个操作一般都是会

很快地连续多次进行的，在 LSP 式编辑器里就需要不停地收发 json，而 JB 就只是调用一个函数，访问一个变量而已。 在不编译的时候，在运行的程序就只有编辑器，代码只存在于编辑器里，没有多份拷贝。 遇到语法复杂的语言时，可以选择写一个轻量级的 parser，把一些原本属于语法分析的工作放到后台的代码分析进程里，可以在一定程度上缓解编辑时的体验（其实就是自己实现 parser 带来的灵活性）。 不过，要实现这样的功能，就要牺牲部分读取文件的优化，以及需要重新实现一个该语言的 parser，行为很可能和编译器本身是 parser 不一致，带来了坑的可能性。两个承担相同任务的 parser 本身也是不应该同时存在的，这是不优雅的架构。

两种解耦策略

LSP 眼中的代码，是一个线性的 token 序列。JB 眼中的代码，是一个树形的语法结构。 一个把全部代码分析交给插件，一个只把语法定义交给插件。

这就是见仁见智的两种解耦方式了。最优的是 parser 和 IDE 本身就使用同一个分析框架的语言，他们不仅只有一份 code base，而且实现了编译器和 IDE 行为一致，重构的时候也不需要进行高成本的通信，甚至编译的时候可以直接使用 IDE 拿到的 AST。 这类例子有很多， Roslyn + Visual Studio 是一种， Kotlin + IntelliJ IDEA 是一种，其他的我还没听说过不过不可能只有这俩。 由于社区的分离，绝大多数语言都不可能有这种待遇（只有微软和 JB 才出 IDE，其他搞语言的只做的起插件）。 不过我以后要是造语言，肯定会提供这样的套餐的。不然和咸鱼有什么区别呢。

本文完

机场写文章真爽。

成都，辣鸡二线小城市。真羡慕北京的教育条件，不过我这里也比山区的孩子好了。 希望我能把这一手不好不烂的牌打好点吧。

这个系列还会更新哦。 以后的方向主要是讲 JB 式编辑器的实现，以及一些文本编辑器的通识技巧。

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License