

2025-05-30

Tinymist 2024

Review of the Tinymist in 2024.

Contents

Tinymist 2024 - 语言服务器部分	2
1.0.1 社区定位与竞争	2
1.0.2 LSP 框架	2
1.0.3 LSP 协议格式库	2
1.1 LSP 引擎库	3
Tinymist 2024 - 语言服务部分	4
2.0.1 Tinymist 项目结构的设计	4
2.0.1.1 可变全局状态	4
2.0.1.2 可变后台状态	5
2.0.1.3 可变本地状态	5
2.0.2 Typst 工具的设计	5
2.0.3 tinymist.lock 文件设计	6
2.0.3.1 核心结构: InputSpec、OutputSpec 和 RouteSpec	6
2.0.4 textmate 语法支持	7

2025-05-23

Tinymist 2024 - 语言服务器部分

关于 tinymist, 一个 typst 的语言服务器的开发思考。

在 2024 年初的时候, 由 nvarner 编写的 typst-lsp 已经基本停止开发。怒其不争, 遂开发了 tinymist。就个人背景, 此前我的主要编程语言是 C++, Python 和 TypeScript; 并已经有开发过 clangd 的经验, 学过一些 lsp 相关的知识; 我只学过简单的 rust 语法, 并上手了 typst.ts 作为第一个项目。也就是说 tinymist 是我的第二个 rust 项目。

1.0.1 社区定位与竞争

一般来说, 一件事情整个社区最好只有一个仓库, 我还是换了一个仓库开发针对 typst 的语言服务, 这里面有好有坏。从结果来看, 我们的这一举措并没有对 typst 生态造成破坏, 这便很好了。

最初, 我已经是 typst-lsp 的贡献者了。在开发的过程中, 我发现很多问题。首先 nvarner 已经几乎不审 PR 了。其次, 由于 typst-lsp 的一些错误决定, 一些协议上的 bug 已经到了需要完全调整架构的程度, 我没有信心和耐心劝说 nvarner 修改 typst-lsp 几乎所有代码。

我认为 typst-lsp 这个名字并不好, nvarner 也认为取了这个名字 typst-lsp 就只专注于 lsp 相关开发。出于一些想法, 我认为对于语言服务, 我们要提供一个统一的 vscode 扩展和仓库。

在定位上, tinymist 与 typst-lsp 的区别是, tinymist 关注与编辑器的所有交互, lsp 只是其组成部分之一。lsp 只是用户无需关注的与用户交互的编辑器协议细节。

不多的人注意或在意这一点, 一方面因为 lsp 势大, 许多人认为 language support 等于 lsp。实际上, 还有很多其他有趣的功能不被 lsp 囊括。例如, 将来我们还能引入 tinymist-dap, tinymist-lsif 等。tinymist-lsp 才是等价于 typst-lsp 的部分。如果 nvarner 最终能回来, tinymist-lsp 应该能合并到 typst-lsp, 尽管这需要相当长的时间来协调 PR。

另一方面, 用户的知识有精英化的趋势, 许多精英更喜欢学习这些无关紧要的细节, 并享受为其一一配置的过程。而我认为, 为了方便不发声的大部分群体, 所有的这些最终都要统一组装起来交给用户, 而非散落成很多扩展让用户一一安装。这一点因 typst 的目标群体并非都是程序员而变得尤为重要。

1.0.2 LSP 框架

在事实上, rust 的 lsp 相关库应当分为两部分。一部分是 lsp 协议的格式, 这对应于 lsp-types; 另一部分是 lsp 协议的引擎框架, 这对应于 lsp-server。不幸的是, nvarner 选的这两方面的库都中了招。

1.0.3 LSP 协议格式库

lsp-types 是目前最广泛使用的关于 lsp 协议格式的库, 但它只是一个小众语言的子项目。它目前的设计已经足够几乎所有场景的使用, 但也有不好的地方。

首先, lsp-types 曾在 0.95 做出过错误的决定, 将 `url::Url` 替换成了 `fluent::Uri`, 揭露了 rust 在 `uri/url` 这个几乎是最常用格式上的混沌。`url::Url` 本身就有许多毛病。比如在解析 `file://` 时会给它加一个斜杠, 而 neovim 因此变得红温。`fluent::Uri` 则表示, 这些我干脆都不支持, 就没有 bug 了。但是大家实际上是需要 `url::Url` 上的许多方便方法 (method) 的。这直接使得几乎所有依赖 lsp-types 的语言服务无法升级 lsp-types 到更高版本。

其次，我觉得 lsp-types+lsp-server 在类型和格式的设计上有性能问题。比如，我认为 lsp-types 里所有的 String 都应该替换成 EcoString 等减少内存拷贝的特殊字符串类型。尤其是当在语言服务器有缓存的时候，许多 String 都只是简单拷贝并响应给编辑器。当然，简单观察后，我还发现由于 lsp-server 使用了 serde_json::Value 擦除类型，零拷贝已然成为了不可能。总的来看，我希望有新的 lsp 库来解决这些问题。尽管从直觉上，这并非性能上的主要问题，使得我个人不会急迫需要这方面的改进。

1.1 LSP 引擎库

在 lsp 引擎上，nvarner 的选择是 tower-lsp，但是这个库事实上并没有尊重 lsp 协议（2024 年的时候，tower-lsp 的情况如此）。lsp 在时序上希望你能保证按顺序处理请求，而 tower-lsp 收到请求上会乱序触发上层 service 的函数。这会导致 language server 状态在启动后一段时间与编辑器状态 desync。tower-lsp 的这一做法也使得允许上层 service 有一些“fancy”的写法，直接导致 typst-lsp 需要完全重写。

rust-analyzer 是怎么做的呢。rust-analyzer 使用了 lsp-server，这是一个底层完全同步的 lsp 引擎。每当有一个请求到来，都会触发一个获得 state: &mut State 的 handler。

一位群友为 nix 写的 nil 用了这位群友自研的 async-lsp。其接口要比 lsp-server 整洁和 neat 的多。每当有一个请求到来，async-lsp 也会触发获得全局可变状态的 handler，区别是这个 handler 是 async 的。

我是希望使用 async-lsp 的。在接入的过程中，再一次挖掘到了 rust 的混沌之处。我们做一个表格，lsp-server, async-lsp, tower-lsp 的区别如下：

Name	Order to Accept Requests	Type of Handler
tower-lsp	Out of order	Fn() -> Fut<Req>
lsp-server	Sequential	FnMut() -> Req
async-lsp	Sequential	FnMut() -> Fut<Req>

虽然 async-lsp 看似 async 了，但是别扭之处在于，它的 handler 无法使用 .await 语法，取而代之，必须返回一个不引用 state 的 async 闭包。这显然是 rust 的局限性。其次 async-lsp 将 stdio 的读写异步化了，而在 windows 上，这必须要借助 tokio 的 compat IO (correct me if I'm wrong, 因为我不是 async 专家)。对于 nix，这并非问题，因为 nix 只会在 unix 上运行 (correct me if I'm wrong, 因为我不是 nix 用户)。我觉得 nil 的作者不会为 windows 用户买单属于合情合理。

另外，为了方便测试，我有一些希望 async-lsp 改变的东西 (目前已经忘了)。出于以上原因，尽管 tinymist 已经完全做好了迁移到 async-lsp 的准备，我还是选择了继续保留对 lsp-server 的包装。总的来说，我希望将来要么 rust 改进对 async 借用的支持，要么有一个更好的引擎框架。

2025-11-09

Tinymist 2024 - 语言服务部分

关于 tinymist, 一个 typst 的语言服务器的开发思考。

受到一些项目的熏陶, 我近些年的项目设计越来越倾向于将一系列软件的代码集中在一起, 并减少组件之间的过度接口设计。tinymist 是我近些年对 monorepo 的首次尝试, 经过实践, monorepo 确实提高了少人合作情况下的开发效率。

2.0.1 Tinymist 项目结构的设计

从逻辑上, tinymist 多分为前后端。

- 语言服务: crates/tinymist (Rust 组件) 为后端, editor/vscode (TypeScript 组件) 为前端, 数据协议为 LSP。
- 预览服务: crates/typst-preview (Rust 组件) 为后端, tools/typst-preview-frontend (TypeScript 组件) 为前端, 传输协议为 WS, 数据协议以 SVG 为基础。
- 分析器: crates/tinymist-query (Rust 组件) 为后端, crates/tinymist 为前端, 接口基于 LSP。

可以发现, 所有这些前后端接口都是已成熟的二进制协议, 即数据对象结构稳定且可序列化。这非常有利于解耦和测试。

大组件由多个小组件组合, 且遵循命名规范。我们拿语言服务器作例子。其完成一次 LSP 请求的逻辑大多如下:

```
impl ServerState {
    fn serve(&mut self, req: LspRequest, id: RequestId) {
        let snapshot = change_and_prepare(self, req) // mut
        spawn(async move {
            let mut ctx = LocalContext(snapshot);
            let resp = handle(&mut ctx, req).await;
            ctx.send(id, resp);
        });
    }
}
```

tinymist 遵守本地可变, 共享 (全局) 不可变的设计。这是我认为语言服务器一种较为正确的设计方式。

2.0.1.1 可变全局状态

ServerState 是可变的, 意味着服务器串行开始所有的语言服务请求。但是这并不意味着全串行。ServerState 只是根据请求快速更新全局状态, 并仅创建快照 (或锁定) 一部分状态, 进入耗时部分。由于快照可以安全并发, 到 2025 年, tinymist 已经可以保证请求之间两两不阻塞, 尤其是耗时请求不阻塞延迟敏感的请求。

ServerState 是所有服务器状态的组合, 例如:

- ProjectState 保存了所有项目的状态, 其快照为 LspWorld。对 Typst 熟悉的人可以迅速反应过来, LspWorld 持有所有 Typst 编译资源, 是 Typst 任务访问操作系统的切面。
- PreviewState 保存了所有预览的状态。当启动预览时, 服务器启动一个 tokio 任务, 并返回一个 mpsc 的 channel 作为“快照”。tokio 任务持有可变状态, 响应来自其他不同线程的预览服务请求。

- 其他大大小小数十个 state，都具体而组合式地陈列在 `ServerState` 中。

对这些 state 单独阅读代码可以很清楚地知道 `tinymist` 如何管理所有可变全局状态，也方便定位相关代码逻辑。这与很多项目喜欢使用抽象类把实现隐藏而有所不同。

从传统意义角度，`ServerState` 是一个巨型管程。当开始服务请求时，直接，一次性锁定和修改所有子状态，并尽快拿到细粒度锁并解锁全局状态。最后，`tinymist` 在细粒度锁上完成耗时任务，以提高并行度。

这种设计参考了 Linux，并做出了简化。其简化的部分是对初始服务锁粒度的控制。Linux 对于某个 `syscall`，只会逐渐锁定一部分状态，从而有死锁和 TOCTOU 的风险，而 `tinymist` 则是先直接锁定所有状态。

`tinymist` 的简化虽然有降低吞吐量的缺点，然而这种缺点在语言服务器场景下被弱化了。一方面语言服务器并不需要超高并发，另一方面 `tinymist` 引入了快照机制。事实证明，`tinymist` 的这种设计方法非常适合高可靠服务。我认为经过优化，`tinymist` 这种模式能胜任 10k 或 100k OP/s 的服务强度。作为参考，`tinymist v0.14` 在冒烟测试中的服务速度为 11.04k OP/s。

2.0.1.2 可变后台状态

`PreviewState` 可以很好地反映 `tinymist` 如何应对复杂的生命周期任务。简单分析，用户会请求开始预览，中途发送多个预览请求，再请求终止预览。

当用户请求预览时，来到 `ServerState`。其创建一个 `PreviewActor`。`PreviewActor` 在后台，其本身又是可变的，串行服务所有的预览请求，实际上设计模式与 `ServerState` 相同，都是 Actor 设计模式。

从生命周期上来看，单次预览请求服务生命周期严格包含于 `PreviewActor` 生命周期，而 `PreviewActor` 生命周期又严格包含于 `ServerState` 生命周期。同时预览服务请求也和语言服务请求相同，依靠大管程和快照机制保证安全修改全局状态的同时保持高并发。

2.0.1.3 可变本地状态

在作分析请求的时候，我们可以发现，在本地栈上，`tinymist` 创建了一个 `LocalContext`：

```
let mut ctx = LocalContext(snapshot);
```

这个上下文对象依然是可变的。当分析要被缓存的时候，上下文对象首先优先查询本地缓存，其次再从全局缓存中拉取结果。这参考了 MLIR 的 `parametric storage` 设计。

2.0.2 Typst 工具的设计

`tinymist` 并非一个语言服务器，而是 `Typst` 的一个集成服务，或称为 `Typst` 的工具链 (toolchain)。目前已经有很多相关工具：

- `tinymist lsp`, LSP 协议服务。
- `tinymist preview`, 预览服务。
- `tinymist dap`, DAP 协议服务。
- `tinymist test`, 单元或渲染测试。
- `tinymist cov`, 覆盖率测试。
- `crityp`, 性能测试 (Benchmark)。
- `typlite`, 将 `typst` 转换为 `markdown`、`tex`、`docx` 等其他标记语言。

对如此之多的工具，`tinymist` 的抽象却相对简单。首先我们有：

```
#[derive(clap::Parser)]
pub struct CompileOnceArgs { ... }
```

这是一个实现了完全兼容 `typst-cli` 的命令行解析的结构体，同时实现了 `WorldProvider`：

```
pub trait WorldProvider {
    fn resolve(&self) -> Result<LspUniverse>;
}
```

`LspUniverse` 是一个可变结构体。开发者可以很方便地修改编译器资源，并随时创建一个安全多线程共享的 `LspWorld` 快照。

```
let verse = CompileOnceArgs::parse().resolve();
let doc = typst::compile(&verse.snapshot());
```

如果一个工具需要定制命令行参数，那么只需要利用 `clap flatten`：

```
#[derive(clap::Parser)]
pub struct ToolArgs {
    #[clap(flatten)]
    compile: CompileOnceArgs,
    #[clap(flatten)]
    extra: ToolExtraArgs,
}
```

这样，可以非常轻松地构建并开始 `Typst` 任务：

```
let ToolArgs { compile, extra } = ToolArgs::parse();
let verse = compile.resolve();
run_tool(verse, extra);
```

这里我认为我们的设计很好的遵循了减法：

- 命令行参数完全兼容 `typst-cli`，从而所有的工具在升级时随 `typst-cli` 平稳升级接口，也节省了用户学习负担。
- 将 `clap` 耦合到接口里，节省了开发者的设计开销。因为第一步很自然地是解析命令行，紧接着将开发者导向 `Universe` 等概念。
- 依然遵循我们上一节提到的“本地可变，共享（全局）不可变的设计”，允许高效的多线程任务。

2.0.3 `tinymist.lock` 文件设计

`tinymist.lock` 是一个**编译历史数据库**，记录了工作区内发生的编译事件。其灵感来源于 C++ 的 `compile_commands.json` 和 Rust 的锁文件机制。其主要目的是帮助语言服务器理解工作区中文档与源文件之间的复杂关系，特别是在多文件项目中，解决自动识别“主文件”的难题。其通过 `tinymist.projectResolution` 设置来切换项目管理模式。默认为 `singleFile`：将每个 `Typst` 文件视为独立文档。不生成或使用锁文件，适用于单文件或小型项目。可以设置为 `lockDatabase`：模仿 Rust 的项目管理方式，跟踪编译和预览历史。数据存储在 `tinymist.lock` 文件和缓存目录中，能根据历史上下文自动选择主文件。

2.0.3.1 核心结构：InputSpec、OutputSpec 和 RouteSpec

兼容 `typst-cli`，将输入拆分为三部分：`InputSpec` 决定如何创建一个 `LspUniverse`，`OutputSpec` 决定如何执行某个 `Typst` 任务，`RouteSpec` 决定该条目的优先级。

路由机制自不用多说。有多个输入来源，允许 `tinymist` 综合推断项目信息：

- CLI 命令：使用 `tinymist compile/preview --save-lock` 触发。
- LSP 命令：通过编辑器客户端推送。
- 外部工具：如测试框架。

`tinymist` 将设置多个具有不同持久性和优先级的数据库。

- 内存数据库，自动学习本次语言服务声明周期的所有输入来源。当服务终止时，丢失相关信息。
- 文件数据库，在项目的根可以保存一个 `tinymist.lock` 文件，允许多个进程共享输入来源信息。

总结：

- 避免用户额外学习和手动配置，让用户通过“成功的”编译，例如 `typst compile (tinymist compile)`，更新 `lock` 文件。
- 格式依然完全兼容 `typst-cli` 的命令行接口，从而保证了稳定性。这也方便其他工具也遵守相同的协议。
- 尽可能少的在文件系统中存储数据，从而有利于多进程合作。

2.0.4 `textmate` 语法支持

`typst` 的 `textmate` 较为艰巨。`textmate` 基于 `regex`，仅支持 `begin` 和 `while` 的模式。然而我们完成了一个可以解析 `typst/packages` 上所有包的语法实现，里面有一些有趣的准则：

- 通过放弃识别一些语法结构，达成 100% 的准确性。
- 通过假设良性语法结构，完成一些上下文敏感的语法解析。在一到两处极端情况，我们通过脚本生成了前探 6 个字符的正则表达式。

这个 `textmate` 目前已经被 GitHub 采用。