2025-11-09

Myriad Dreamin Blog 2025-11

Archive of Blog posts in November 2025.

Contents

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。

1.0.1 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。

1.0.1.1 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。

1.0.1.2 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。

1.0.1.3 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。

1.0.2 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：

```rust
#[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`：

```rust
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。

```rust
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：

```rust
#[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：

```rust
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。

1.0.3 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。

1.0.3.1 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。

1.0.4 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。