用 Rust 写 JavaScript 编译器

前端性能及新技术实践

字节跳动 Dev Better 技术沙龙

Speaker: Boshen
github.com/boshen

ByteDance 字节跳动

内容

- 现代和未来的前端工具链介绍
- JavaScript 工具链为什么慢?
- Native 工具链为什么快?
- 尝试用 Rust 自研 JavaScript 编译器
- 挑战 ESLint - 20 多倍的性能提升
- 性能优化经验

前端工具链介绍

- Parser - 解析成 AST, 让后面的工具使用
  - babel, tsc
  - swc, esbuild
- Formatter - 格式化源代码
  - prettier
- Linter - 检测错误
  - eslint, tslint
- Transpiler - Down-level JavaScript
  - babel, tsc
  - esbuild, swc
- Minifier - 压缩代码
  - uglify, terser
  - Google Closure Compiler (Java 😂)
  - esbuild, (swc)
- Bundler - 打包代码
  - browserify, webpack, rollup
  - parcel, esbuild

前端工具链趋势

用 Native 语言一统江湖!

- Deno, Bun
  - 用 Rust / Zig 统一 Runtime
- swc, esbuild
  - 用 Rust / Go 重写 Bundler
- Rome Tools
  - 用 Rust 统一全家桶!

商业逻辑 🤔 ？

用性能抢占开发者流量，卖各种服务 💰 💰 💰

JS 工具链为什么慢 🐌?

- 语言自身 - Runtimes
  - Chrome, Node.js, Deno - V8
  - Firefox - SpiderMonkey
  - Safari, Bun - JavaScriptCore
- JIT (Just in Time)
  - 边解析边编译
  - 需要启动时间
  - 没有最优的 CPU 指令优化
- 内存 和 GC (Garbage Collection)
  - 运行时 GC 影响性能
- 架构
  - 算法问题
  - 插件系统
  - 没有重复使用 AST
  - 集成接口不兼容的工具

性能应用需要处理的核心问题

Concurrency vs Parallelism | 并发 vs 并行

- Concurrency - 并发
  - 同一时间执行更多的任务
  - 各种 async 模型
- Parallelism - 并行
  - 把一个任务拆分成多个子任务
  - 并且跑在多个 CPU 上

前端工具链核心逻辑

```
let asts = [];

for (let file of files) {
  let ast1 = parse(file);
  let ast2 = transpile(ast1);
  let ast3 = minifiy(ast2);
  asts.push(ast3);
}

const output = bundle(asts);
```

都是可以拆的任务， 可以并行

可以使用多核!

Native (Rust / Go) 工具链为什么快 ⚡?

- 不单单是多核
- 编译到最优 CPU 指令
  - Rust - LLVM
  - Go - 定制
- 优秀的内存管理
  - Rust - 生命周期
  - Go - 业界最先进的 GC
- 层层优化
  - 极致到每一个比特的使用
  - 系统 API 调用
  - 优秀的性能排查工具

尝试用 Rust 自研 JavaScript 编译器

- 过去半年时间自研 JavaScript / TypeScript Parser
- 跑通 99% Test262, babel 以及 TypeScript 语法测试
- 比 swc parser 快一些，同 esbuild 和 Rome 在一个性能级别

性能来自于架构

- 一体式架构，不考虑插件系统
- 为多核而设计
- 压榨 LLVM, 用更优的 CPU 指令集
- 内存使用优化到极致
  - 使用 Memory Arena 内存池
  - 差点以为自己在写游戏引擎
  - 用各种工具排查和优化内存使用情况

Guide: https://github.com/Boshen/javascript-parser-in-rust

有了自己的地基，才敢搭建应用

挑战 ESLint

- ESLint 性能问题渐渐成为了我们的瓶颈
- 极其复杂的配置系统
- 无法提升能力的插件系统
- 没有区分开代码风格和代码错误两种基本问题
- 没有静态类型分析，需要引入 TypeScript - 变得巨慢无比
- 难处理巨型 Monorepo - 十几万文件，几百万行代码

决定：自研企业级 ESLint

企业级 ESLint 功能

- Performance is feature
- 只检查正确性，不检查风格
- 开箱即用, 无需配置
- 更友好的错误信息

自研 ESLint 成果

```
⚠ eslint(no-self-compare): Comparing to itself is potentially pointless
   ╭─[./src/vs/editor/contrib/inlineCompletions/browser/ghostTextWidget.ts:161:1]
161 │
162 │             if (0 < 0) {
    ·                 ──┬──
    ·                   ╰── Comparing to itself is potentially pointless
163 │                 // Not supported at the moment, condition is always false.
    ╰────
```

```
⚠ eslint(no-dupe-class-members): Duplicated class element
  ╭─[test.js:1:1]
1 │ class A { [123n]() {} 123() {} }
  ·           ────┬─── ──┬──
  ·               │      ╰── It cannot be redeclared here
  ·               ╰── Class element has already been declared here
  ╰────
```

自研 ESLint 成果

- VSCode 仓库 (https://github.com/microsoft/vscode)
  - 3.2K 文件, 786K 行代码
  - 2 秒完成
- 公司内部最大 Monorepo
  - 500万行代码
  - eslint 15 分钟 vs 20 秒 - 45 倍的性能提升
- 多核处理每一个文件
- 多核处理每一条 Linter 规则

性能优化经验

- 性能不是白给的, 需要花时间打磨
- Cache Locality - 缓存访问局部性
  - 想方设法减少数据结构的内存，让它进 CPU / L1 / L2 缓存
  - 在 Hot Path 上面统计使用频率，决定是否引入跟高一层的缓存 (Heap, Disk)
- 申请和释放内存真的很耗时
  - 改为使用内存池后，性能提升 20%+
  - 一些系统 API 会无意间使用内存, 劲量统一申请，提前申请容量 (Capacity)
- 系统学习排查性能问题
  - 多看其他编译器源码，学习细节
  - 学会统计 Code Path 的执行次数，决定是否使用缓存

期待开源

关注 github.com/boshen

THANK YOU