GOTC

Rspack 新一代前端构建工具

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE

目录📚

Rspack 简介

构建工具简介

```
<!DOCTYPE html>
<html>
<body>
  ...
</body>
<script src="./static/1.js"></script>
<script src="./static/2.js"></script>
<script src="./static/3.js"></script>
<!-- ...-->
<script src="./static/18.js"></script>
<script src="./static/19.js"></script>
<script src="./static/20.js"></script>
</html>
```

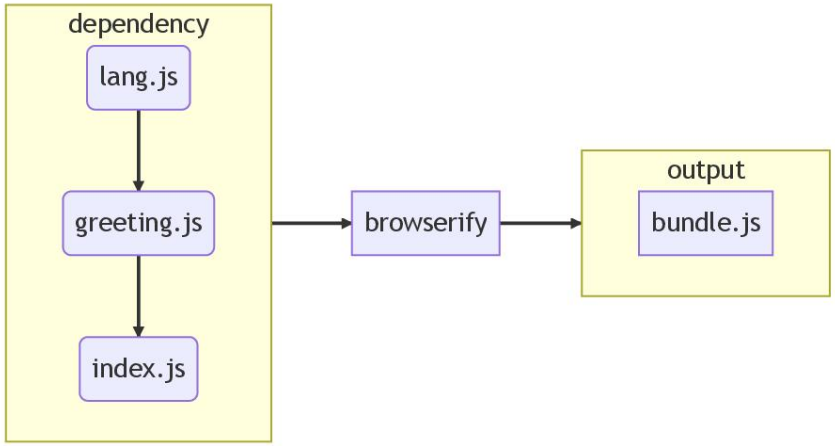
大量外联脚本存在的问题

- 文件依赖关系不清晰
- 可能存在重复脚本引入, 需要人为保证脚本加载顺序以得到正确的执行结果
- IDE 不友好, 很难从中检索出补全信息

构建工具简介

```
1   // lang.js
2
3   module.exports.hello = "hello world";
4
5   // file greeting.js
6   var hello = require('./lang.js')
7   var helloInLang = {
8       en: hello,
9   };
10
11  var sayHello = function (lang) {
12      return helloInLang[lang];
13  }
14
15  module.exports.sayHello = sayHello;
16
17  // file index.js
18  var sayHello = require('./greeting.js').sayHello;
19  var phrase = sayHello('en');
20  console.log(phrase);
```

browserify 构建流程



Web 2.0 中期，工具链井喷式发展

传统单一类型模块构建工具,无法满足开发者的需求

CSS相关的DSL

1. Scss
2. Less
3. Stylus
4. CSS modules

...

HTML(模板)相关的DSL

1. HandleBars
2. Mustache
3. Pug
4. EJS
5. Angular template
6. Vue template

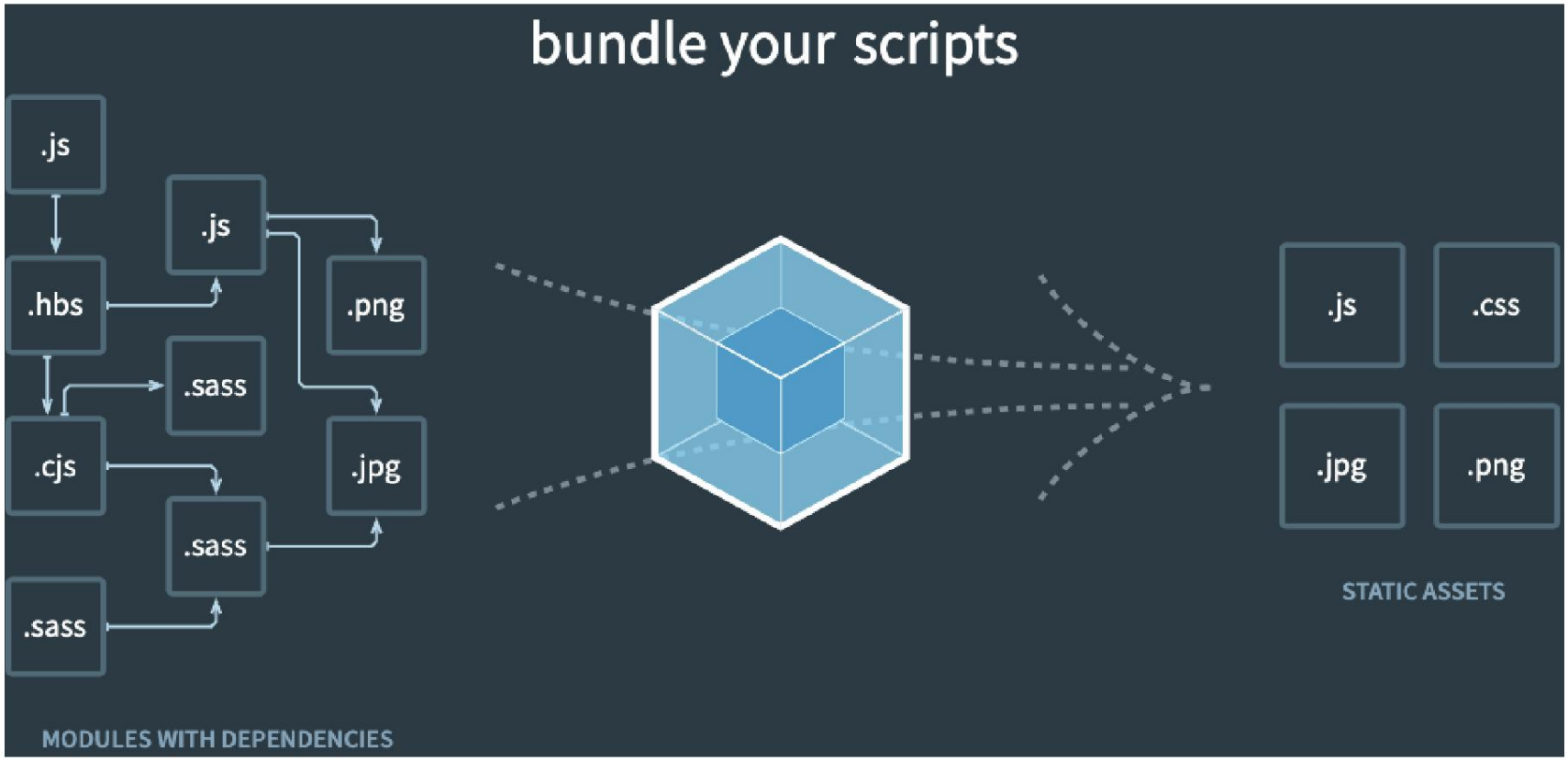
...

JavaScript 相关的方言或DSL

1. TypeScript
2. Jsx
3. Elm
4. CoffieScript
5. Flow

...

Webpack 诞生


bundle your scripts
.js
.hbs
.cjs
.sass
.js
.png
.sass
.jpg
.sass
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

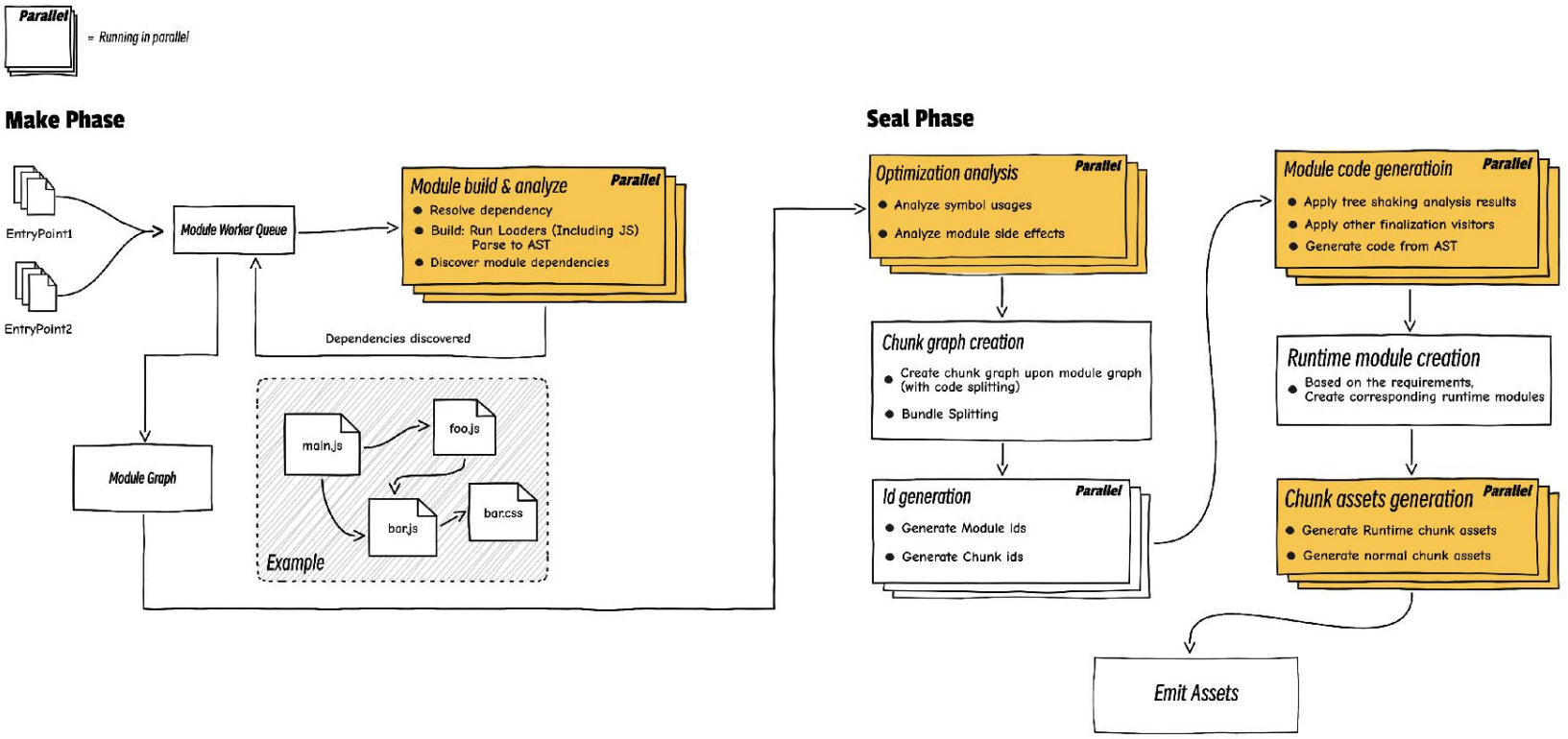
Rspack 诞生

Rspack

A fast Rust-based web bundler

Build a high-performance frontend toolchain

Rspack 架构简介


Parallel
= Running in parallel
Make Phase
EntryPoint1
EntryPoint2
Module Worker Queue
Module build & analyze
Parallel
Resolve dependency
Build: Run Loaders (Including JS)
Parse to AST
Discover module dependencies
Dependencies discovered
Module Graph
main.js
foo.js
bar.js
bar.css
Example
Seal Phase
Optimization analysis
Parallel
Analyze symbol usages
Analyze module side effects
Chunk graph creation
Create chunk graph upon module graph (with code splitting)
Bundle Splitting
Id generation
Parallel
Generate Module Ids
Generate Chunk ids
Module code generatioin
Parallel
Apply tree shaking analysis results
Apply other finalization visitors
Generate code from AST
Runtime module creation
Based on the requirements, Create corresponding runtime modules
Chunk assets generation
Parallel
Generate Runtime chunk assets
Generate normal chunk assets
Emit Assets

技术选型

技术选型

目标

- 尽可能的保证与 Webpack api 以及插件的兼容
- 尽可能的提高构建速度

实现策略

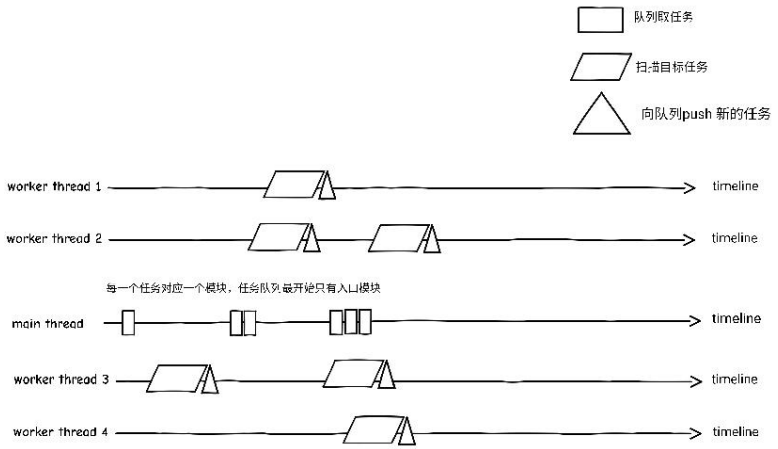
- 按照 Webpack 原架构进行移植
- 改进原来的算法以尽可能的并行化

为什么不用JavaScript + Nodejs

1. Webpack 经过这么多年的迭代和优化，单线程优化潜力不高，很难有成倍的提升。
2. 使用 Nodejs 多线程编程，有以下缺点：
   - Nodejs 是单线程的，只有一个主线程，因此其本身不能直接进行多线程编程。
   - Nodejs 提供了 Worker 线程，但是它是通过创建新的 V8 实例来模拟多线程，数据共享的效率不高。
   - 并发原语支持差，并发编程生态匮乏
   - 综上，多线程 Node.js 程序性能很难达到理想效果

为什么不用JavaScript + Nodejs

简单的多线程基准测试




```
'./d0/f0.jsx'
'./d0/f1.jsx'
'./d0/f2.jsx'
'./d0/f3.jsx'
'./d0/f4.jsx'
'./d0/f5.jsx'
'./d0/f6.jsx'
'./d0/f7.jsx'
'./d0/f8.jsx'
```

为什么不用JavaScript + Nodejs

测试结果 (Rust VS Nodejs)

```
Benchmark 1: node packages/pcp/index.js
  Time (mean ± σ):     264.9 ms ±  15.6 ms    [User: 1803.4 ms, System: 644.4 ms]
  Range (min … max):   242.8 ms … 285.4 ms    11 runs

Benchmark 2: ./target/release/pcp-benchmark
  Time (mean ± σ):      10.2 ms ±   8.7 ms    [User: 23.1 ms, System: 169.0 ms]
  Range (min … max):     6.7 ms …  55.4 ms    340 runs
```

为什么不用golang

1. golang 在性能方面是满足我们需求的
2. 由于语言定位，和本身的生态原因 (对 napi 支持的不好)，golang 在 Webpack API 兼容这一方面无法让我们满意。
3. esbuild 没有提供操作 AST 的 API，不支持转译 JavaScript 到 ES5,
4. 综上, golang 没有成为我们最终的开发语言。

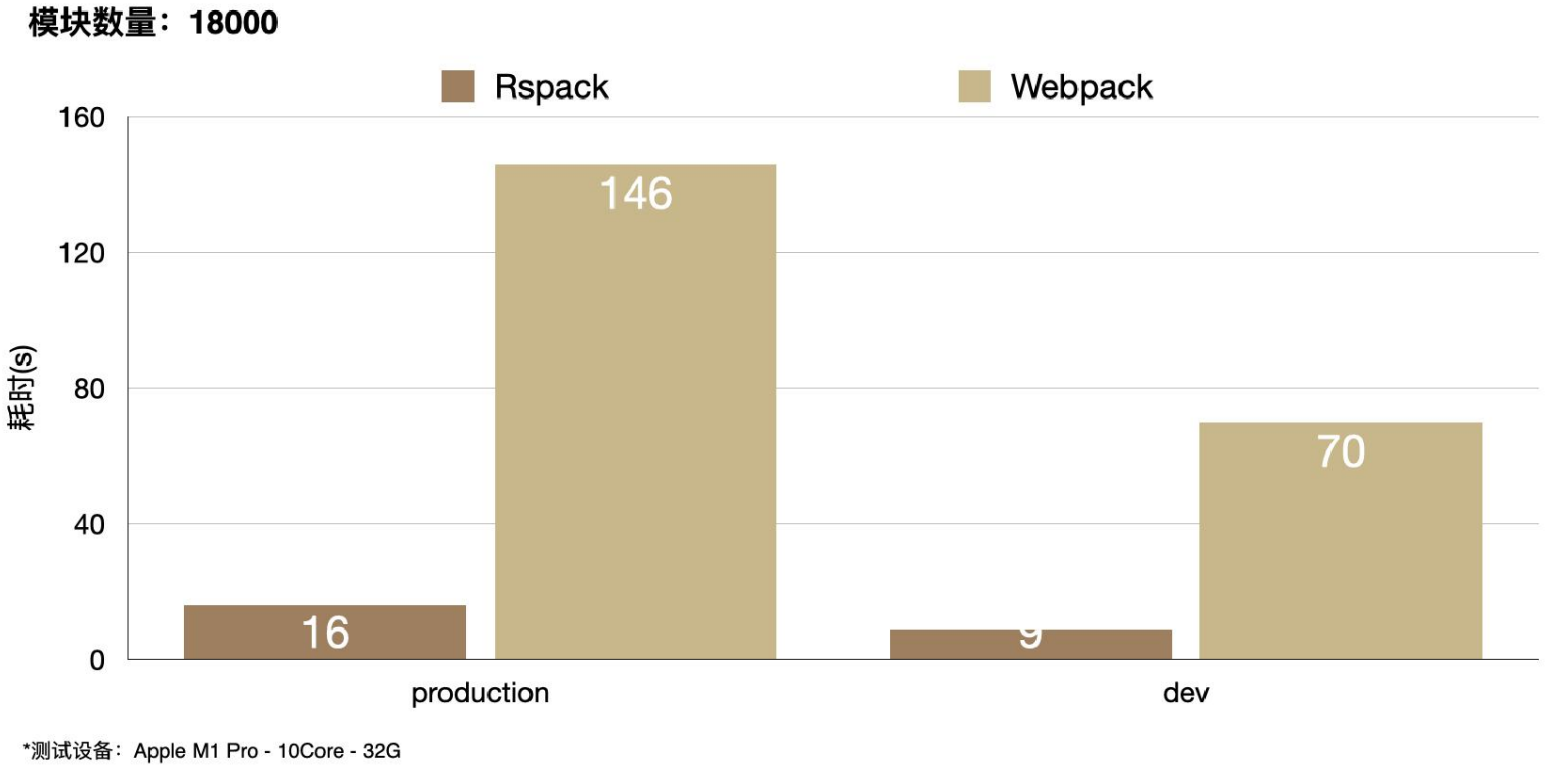
为什么用 Rust

1. 性能优秀，和 C，CPP 一个等级
2. napi 支持良好，可以基于此更好的兼容 Webpack api, 得益于 Rust 宏的支持，我们可以少写很多样板代码。
3. Rust 作为 WebAssembly 的一等公民, WebAssembly 的新特性，基本上都能很快的得到支持, 你甚至可以看到 WASM proposal落地后推着 Rust proposal 前进的 issue,

   > Said otherwise, is it a reasonable requirement to say that every possible backend must support this? Historically WASM didn't, so the answer was no. If WASM tail call support is landing, then it might be acceptable

4. Rust 生态中 swc 提供丰富的 AST 操作 API, 同时支持转译 JavaScript 到 ES5

Rspack 性能收益



*测试设备：Apple M1 Pro - 10Core - 32G

遇到的问题以及解决方案

多线程性能优化

如何解决 swc 并发解析性能差

多线程性能优化

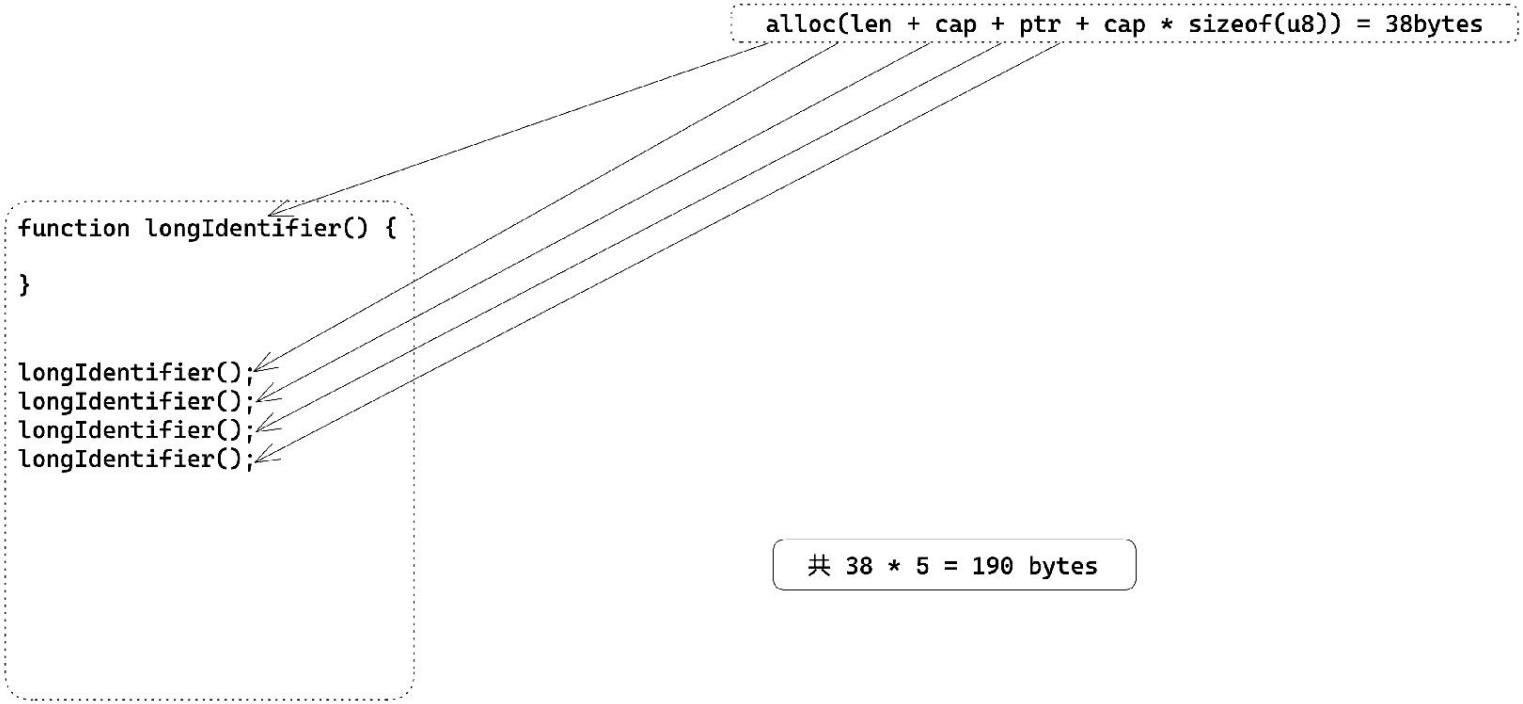
如何解决 swc 并发解析性能差

- dev 模式下不会做过多的优化,通过 profiler 发现 parsing 成为该阶段的主要瓶颈, 具体原因是 parsing 的时候有大量锁的系统调用

多线程性能优化

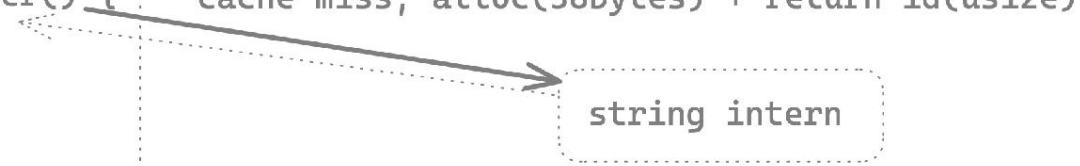
如何解决 swc 并发解析性能差

- dev 模式下不会做过多的优化,通过 profiler 发现 parsing 成为该阶段的主要瓶颈, 具体原因是 parsing 的时候有大量锁的系统调用
- 最后发现是 swc 使用了一个 string-intern 库 string-cache导致的

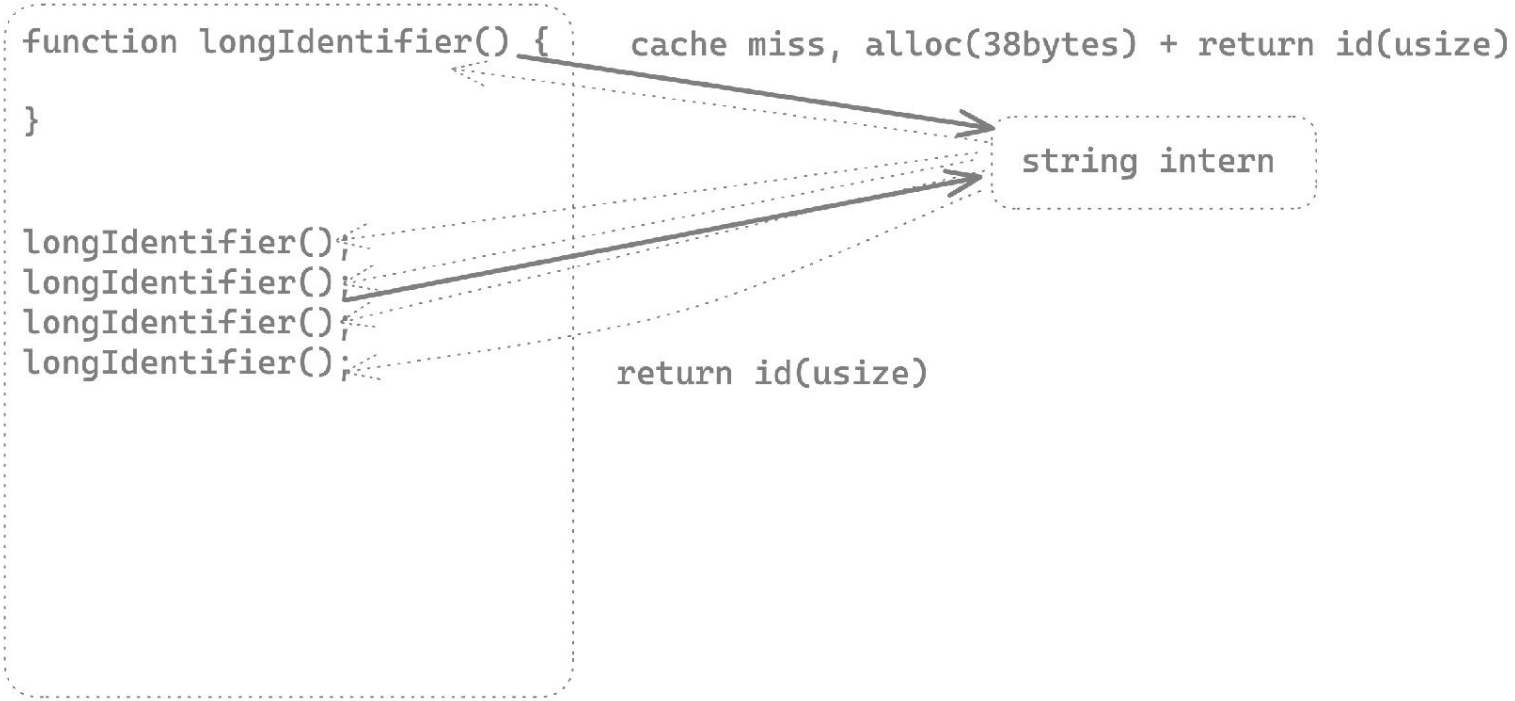

alloc(len + cap + ptr + cap * sizeof(u8)) = 38bytes
function longIdentifier() {
}
longIdentifier();
longIdentifier();
longIdentifier();
longIdentifier();
共 38 * 5 = 190 bytes

string intern 简介

```
function longIdentifier() {

}


longIdentifier();
longIdentifier();
longIdentifier();
longIdentifier();
```

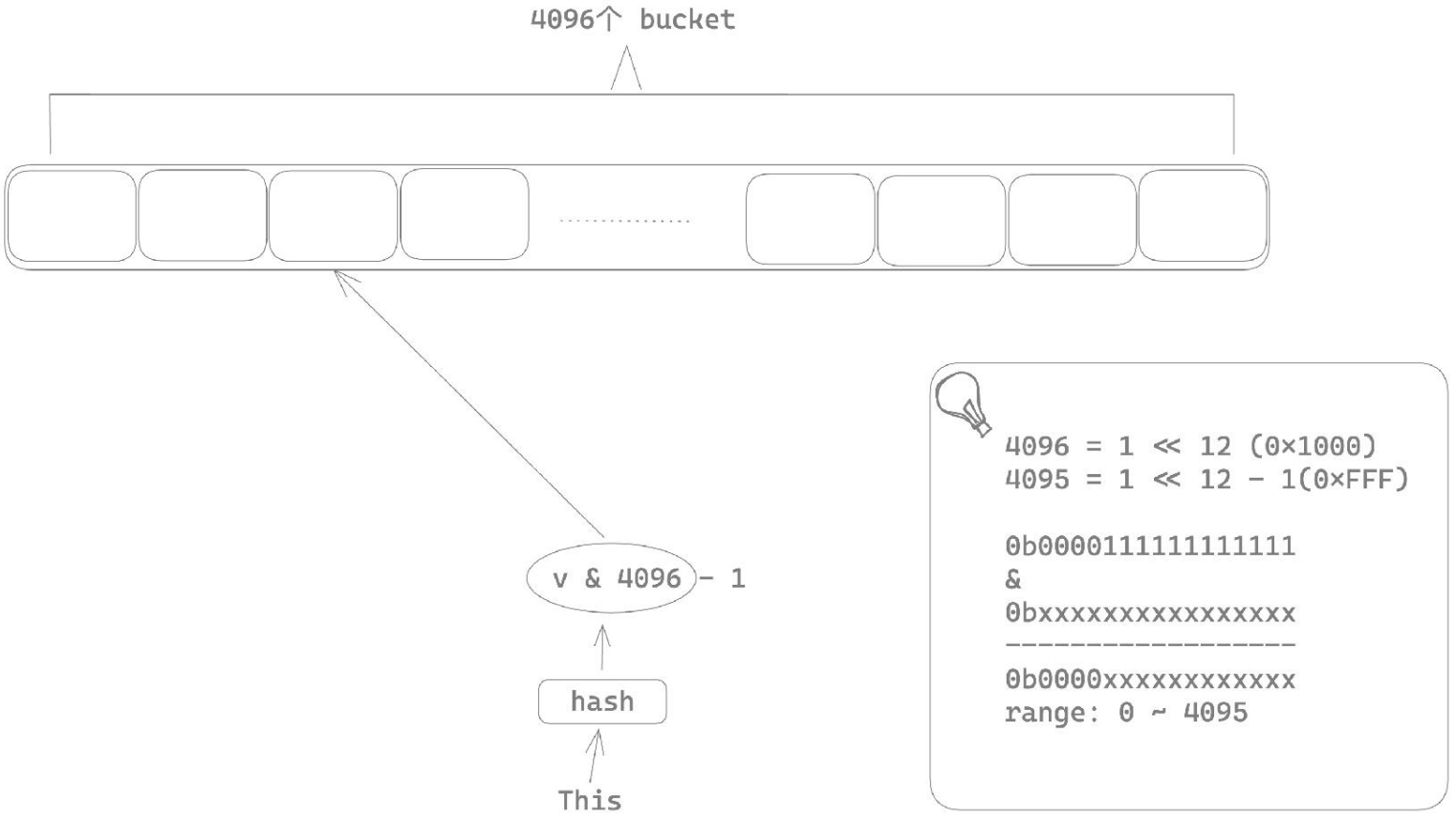


string intern 简介



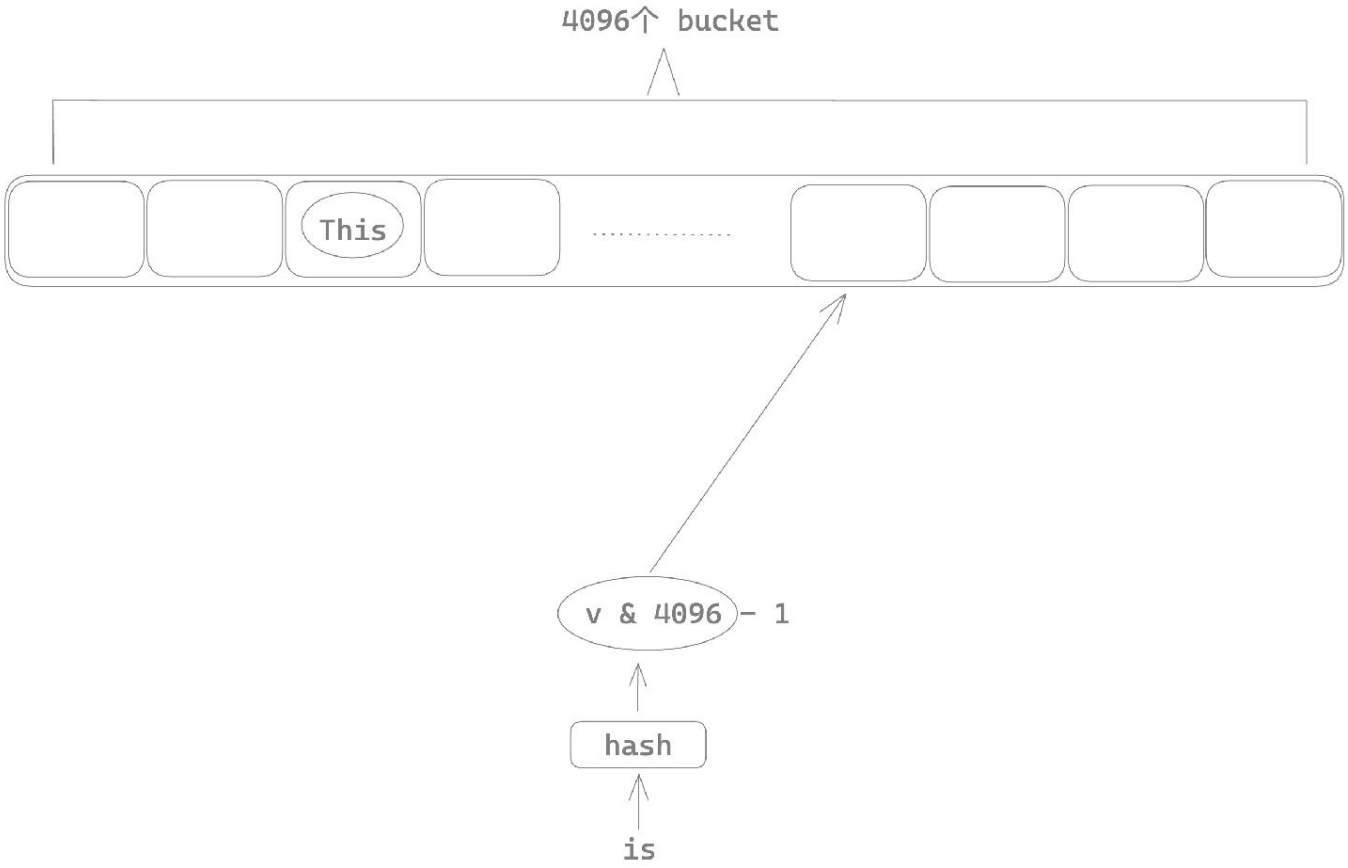
- 节省内存: 5 * 38 - 38 = 152bytes (节省80%内存)

string-cache 简介

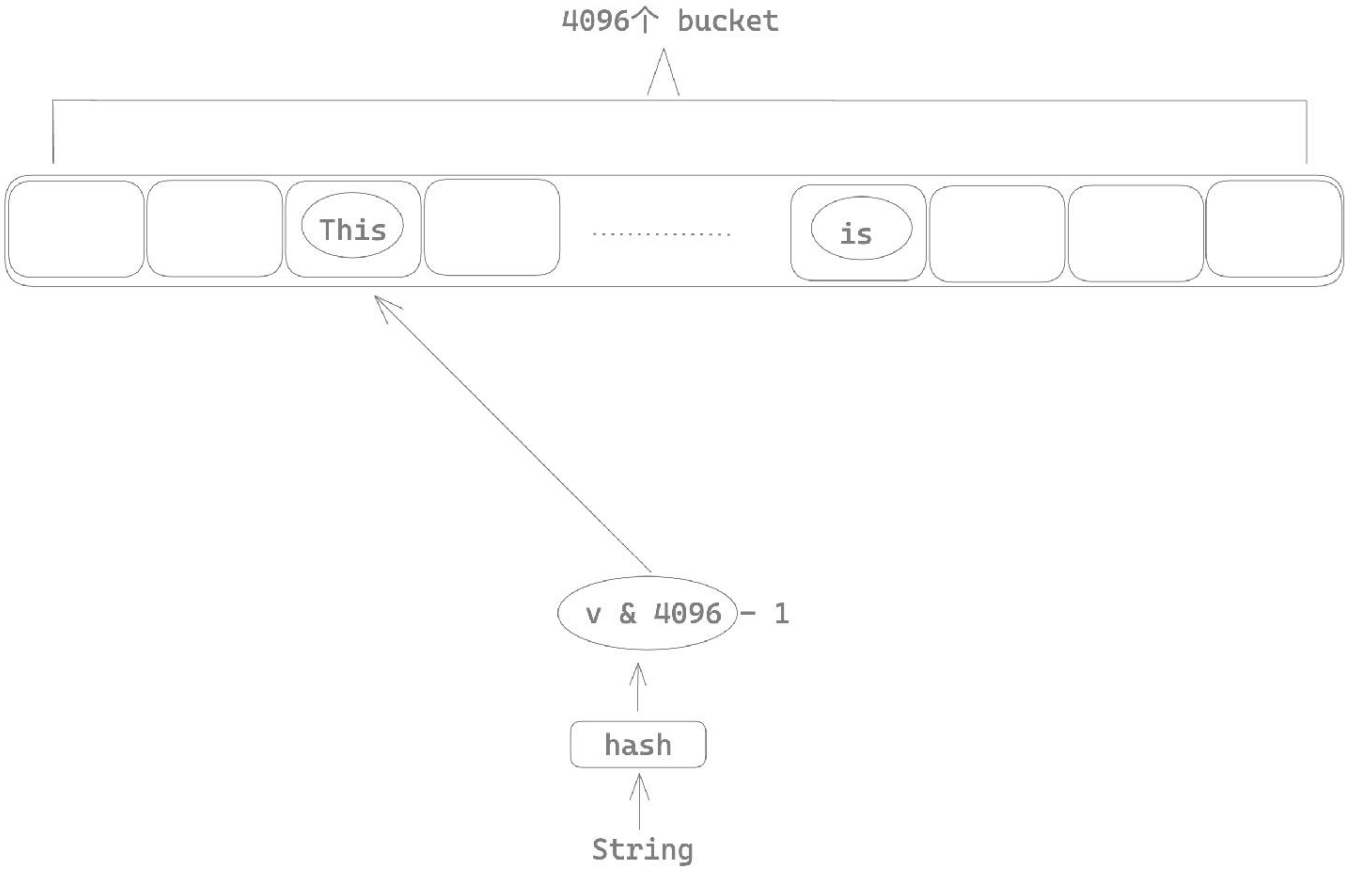



```
4096 = 1 << 12 (0x1000)
4095 = 1 << 12 - 1(0xFFF)

0b0000111111111111
&
0bxxxxxxxxxxxxxxxx
------------------
0b0000xxxxxxxxxxxx
range: 0 ~ 4095
```

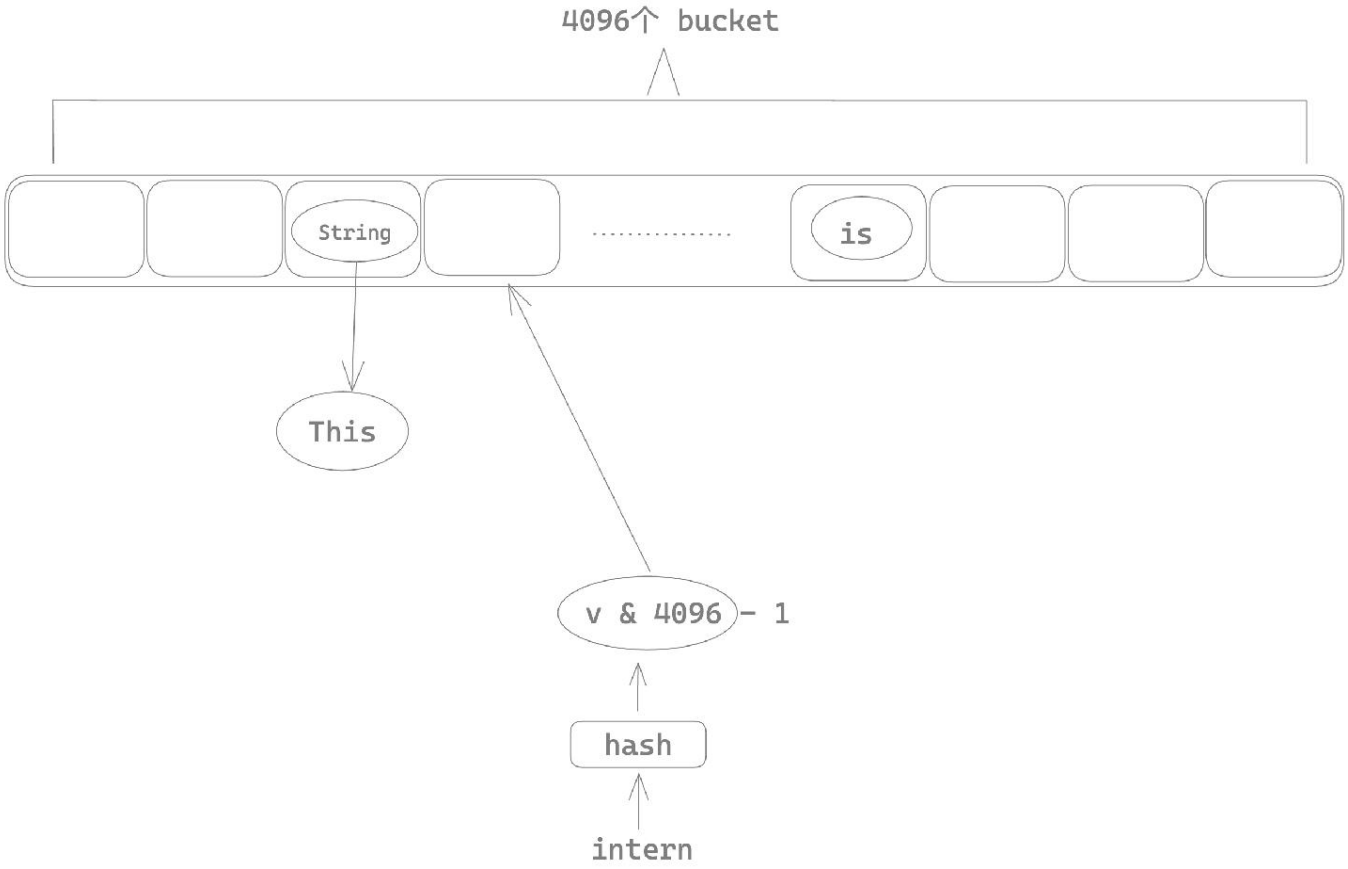
string-cache 简介



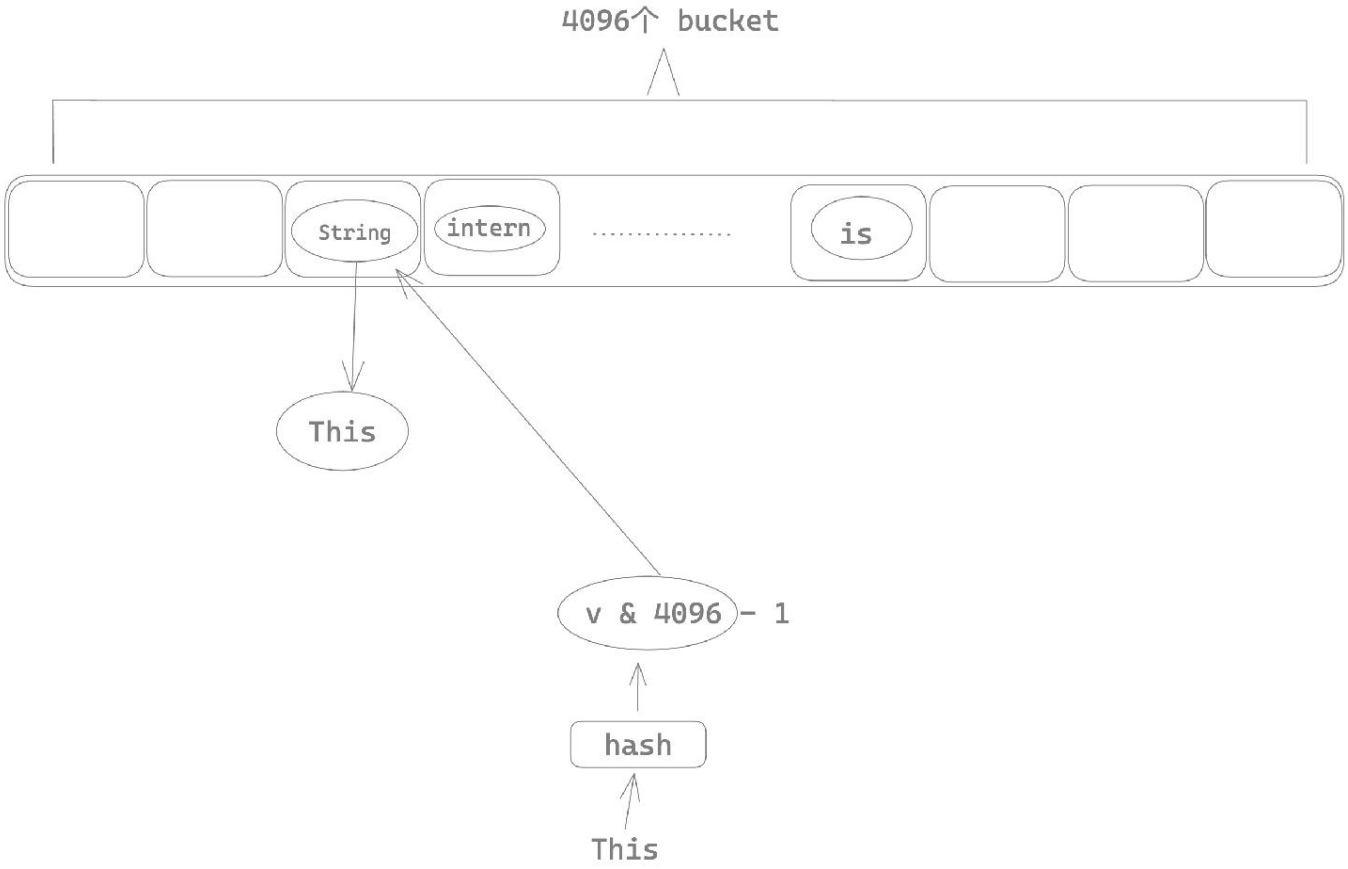
string-cache 简介



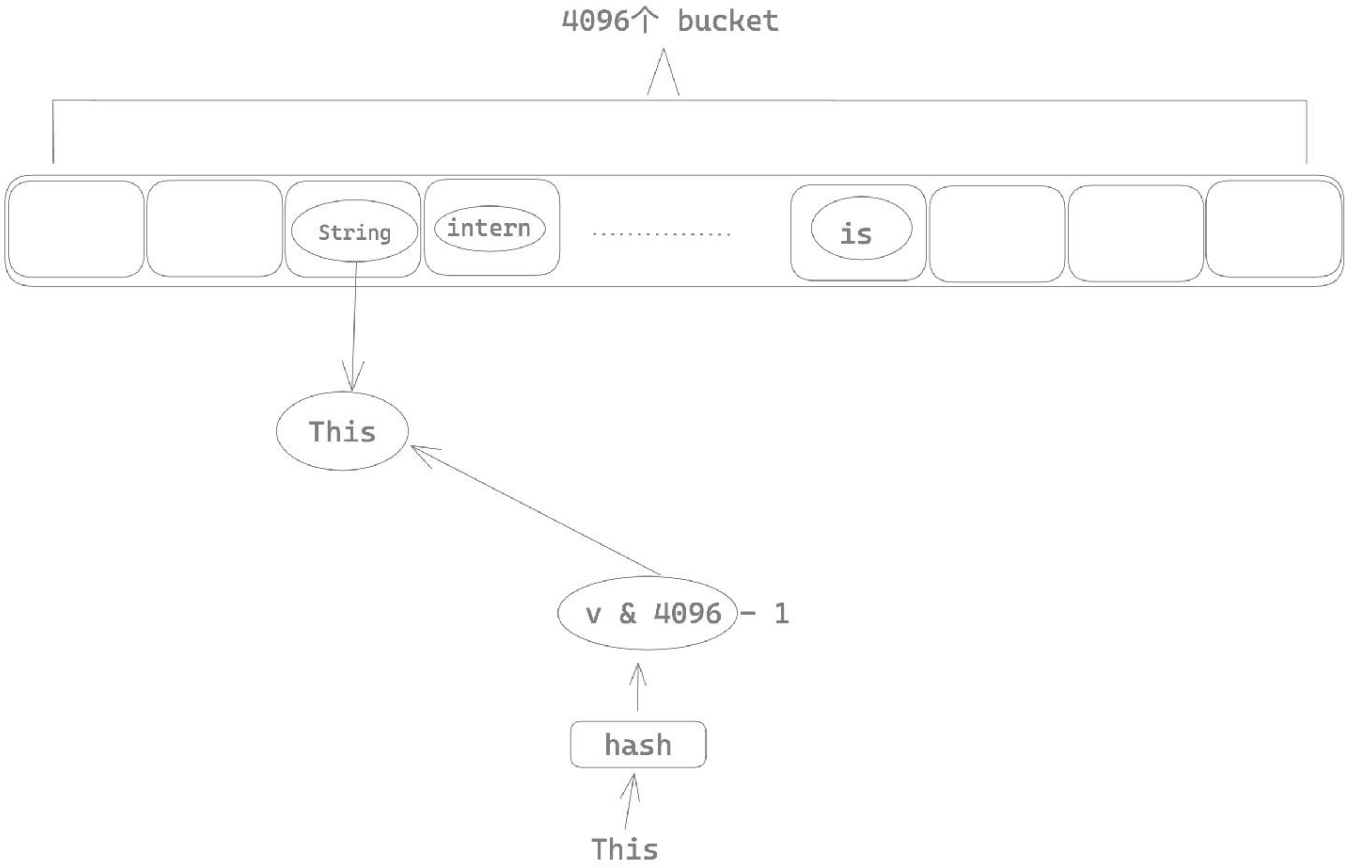
string-cache 简介



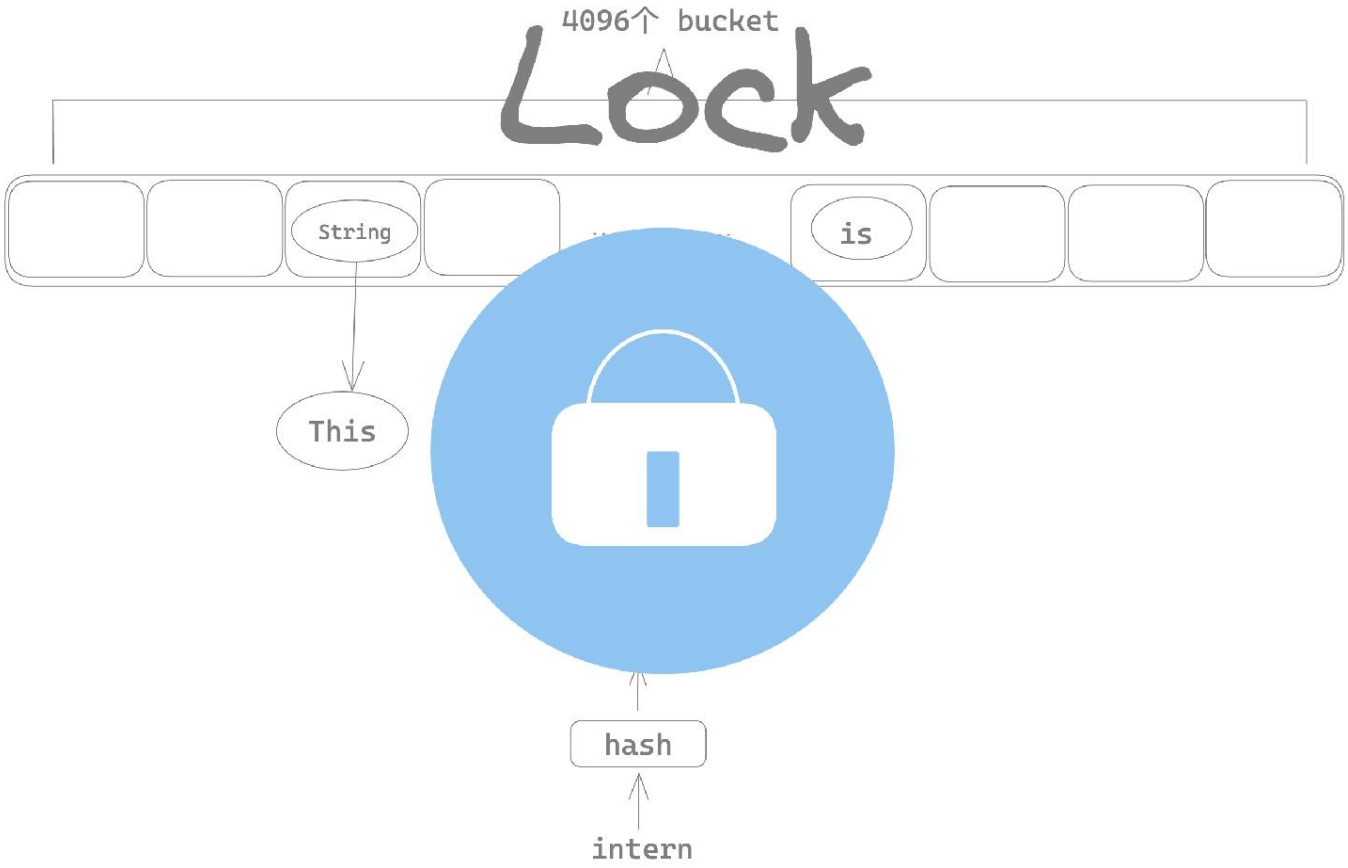
string-cache 简介

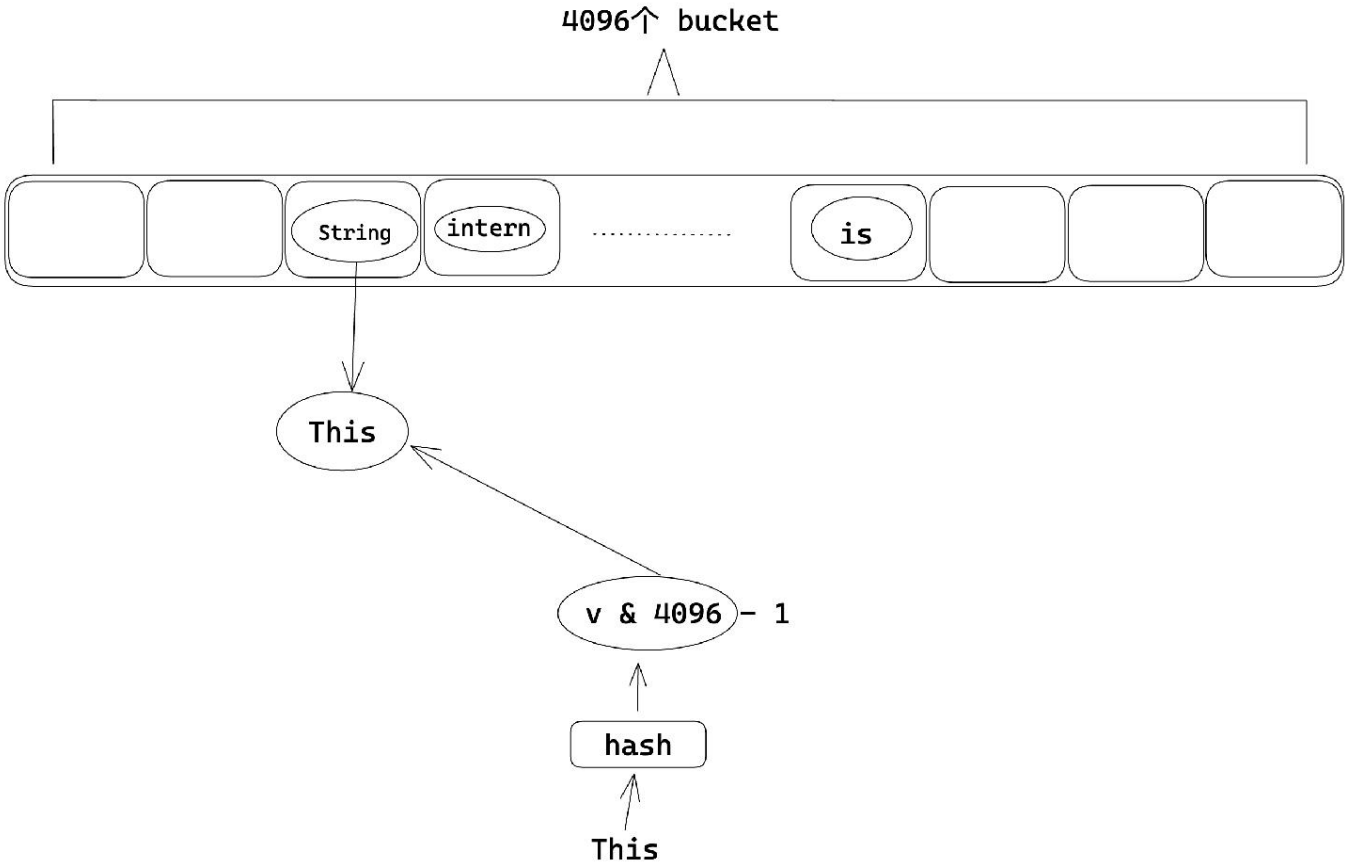


string-cache 简介

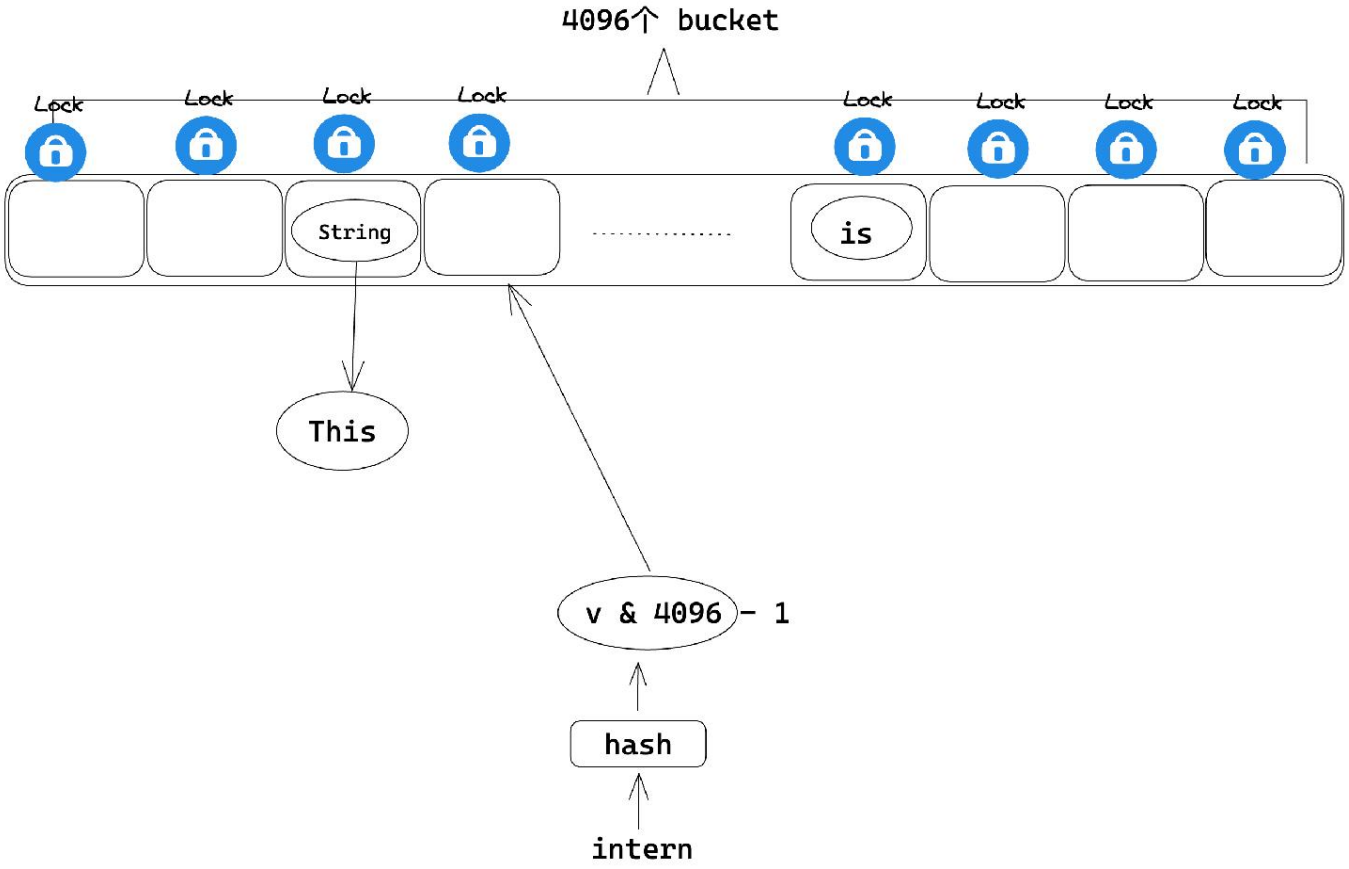


string-cache 并发瓶颈




4096个 bucket
String
intern
.....
is
This
v & 4096 - 1
hash
This

string-cache 性能优化



string-cache 优化结果对比

github-actions bot commented on Feb 17 — Contributor

Benchmark Results

```
group                                                baseline                                pr
-----                                                --------                                --
criterion_benchmark/ten_copy_of_threejs              1.68    539.9±5.42ms        ? ?/sec     1.00    320.8±2.
high_cost_benchmark/ten_copy_of_threejs_production   1.04       4.8±0.02s        ? ?/sec     1.00       4.6±0.
```

- development 模式: 41%
- production 模式: 4%

总结

- 核心, 降低锁的使用, 最大化 CPU 利用率
- 使用无锁的数据结构 (crossbeam等)
- rayon (`iter` -> `par_iter`)
- 降低锁的粒度, 减少不必要的临界区

算法优化

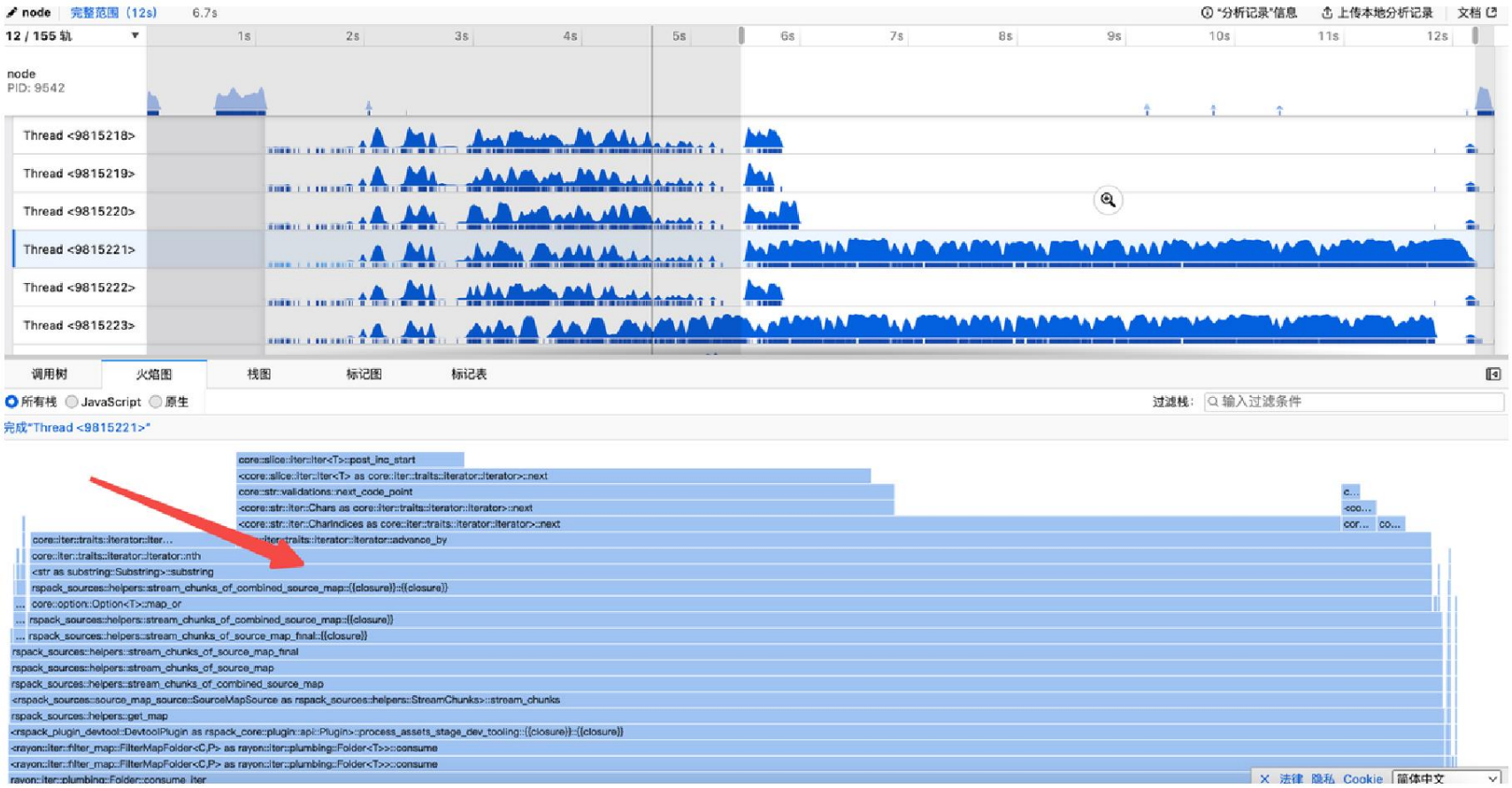
挑选一个趁手的 profile 工具

- Instruments
- samply
- tracing (tokio tracing + perfetto / chrome-tracing)
- perf
- flamegraph

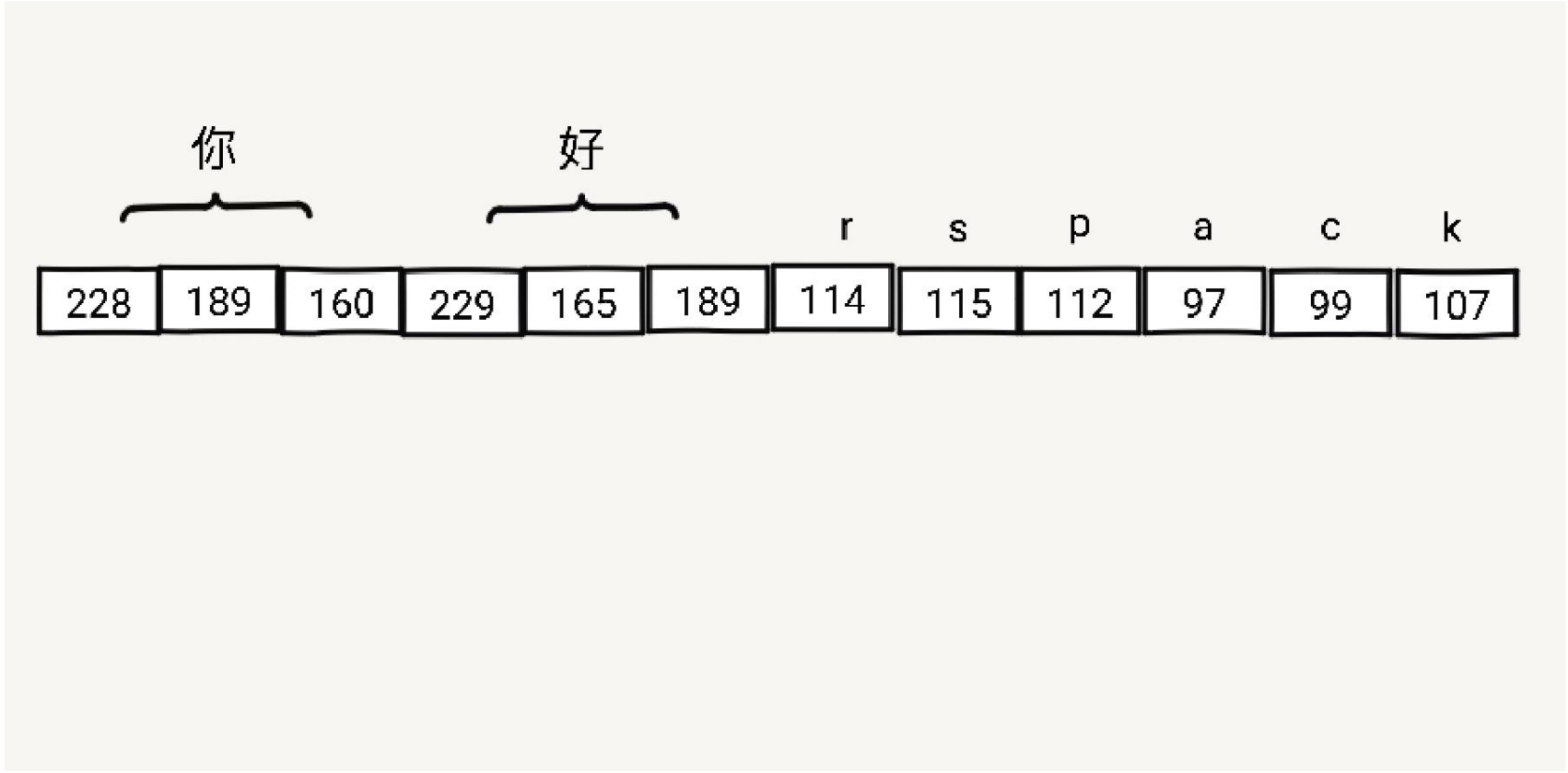
不慎引入 O(n^2) 算法导致性能问题

1. 业务方反馈开启 source-map 和不开启在生产环境有很大性能差异.

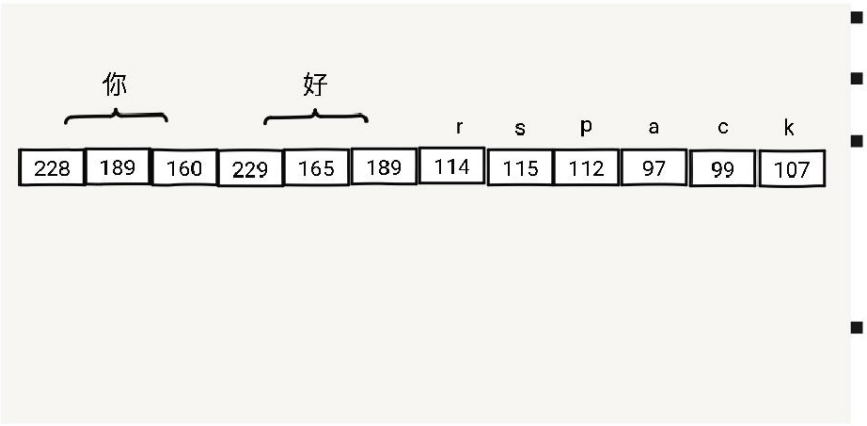
使用 samply 生成 profile



String 在 Rust 中的存储形式



String 在 Rust 中的存储形式



以"好rspac" 为例

- 获取到"好" byte offset
- "你".len_utf8() = 3
- 获取到"k" byte offset
- `['你', '好', 'r', 's', 'p', 'a', 'c'].iter().map(|ch| ch.len_utf8()).sum()` = 11
- 使用 range `3..11` 获取 string slice

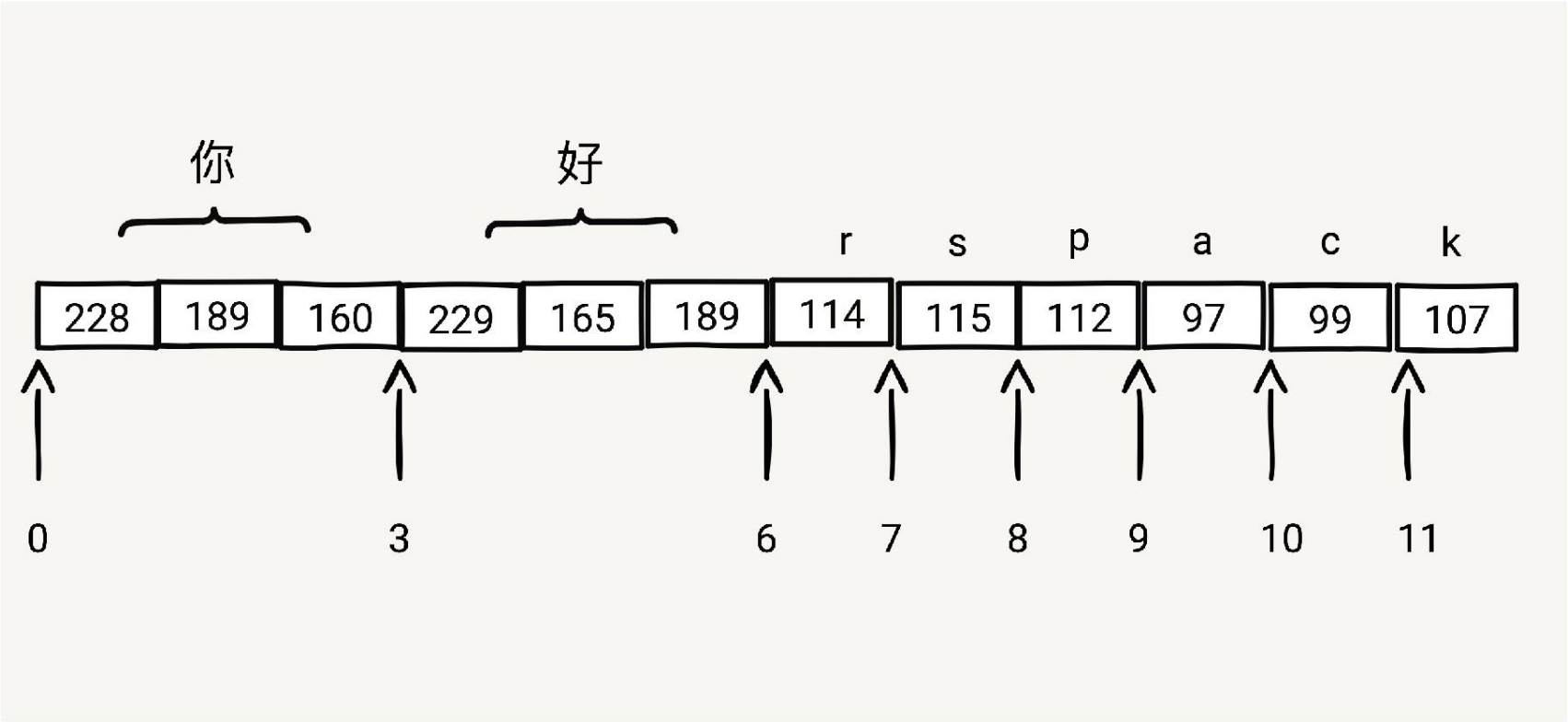
`substring::Substring` 简介

```rust
fn substring(&self, start_index: usize, end_index: usize) -> &str {
    if end_index <= start_index {
        return "";
    }

    let mut indices = self.char_indices();

    let obtain_index = |(index, _char)| index;
    let str_len = self.len();

    unsafe {
        // SAFETY: Since `indices` iterates over the `CharIndices` of `self`, we can guarantee
        // that the indices obtained from it will always be within the bounds of `self` and they
        // will always lie on UTF-8 sequence boundaries.
        self.slice_unchecked(
            indices.nth(start_index).map_or(str_len, &obtain_index),
            indices
                .nth(end_index - start_index - 1)
                .map_or(str_len, &obtain_index),
        )
    }
}
```

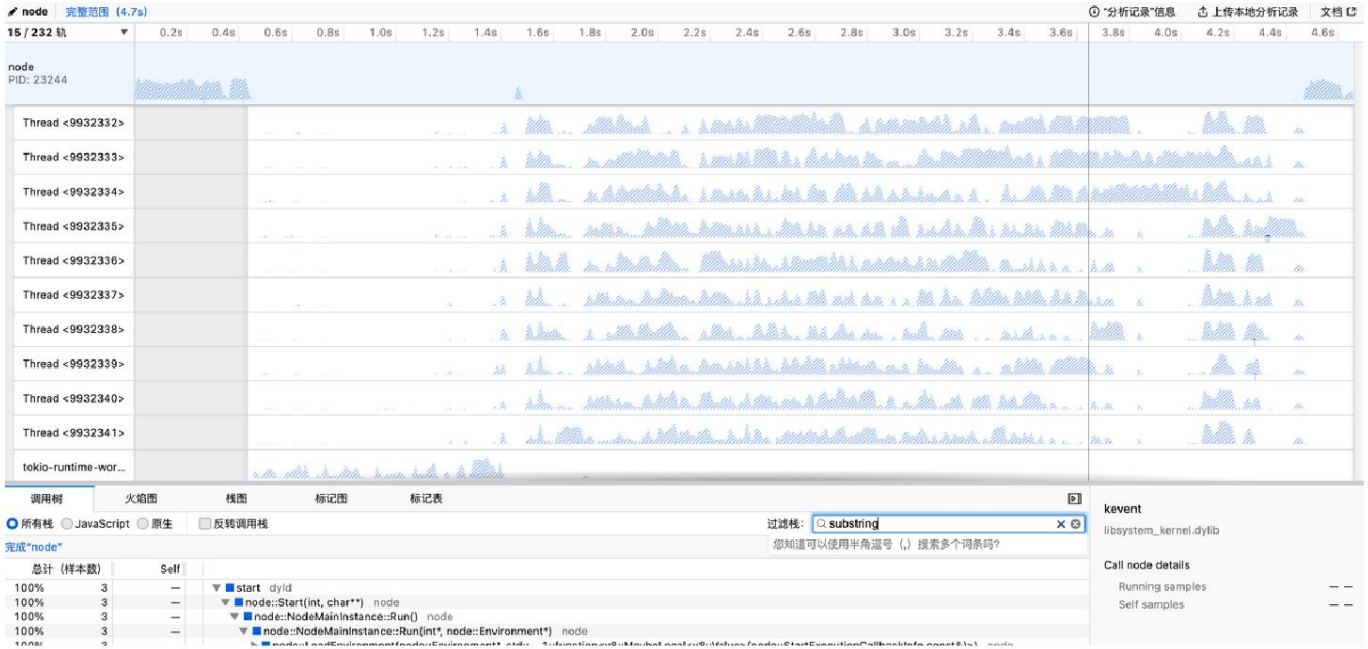
性能瓶颈产生的原因

1. `substring` 被调用的是次数和 `mapping` 数量成正比
2. 通常 `mapping` 数量可控, 所以在大部分情况下,该实现没有性能问题
3. 在 minify 场景下, `mapping` 数量级与压缩产物大小只差常数倍数
4. 该过程的时间复杂度约为, O(n^2), n 为压缩后产物大小

性能优化


你
好
r
s
p
a
c
k
228
189
160
229
165
189
114
115
112
97
99
107
0
3
6
7
8
9
10
11

优化收益



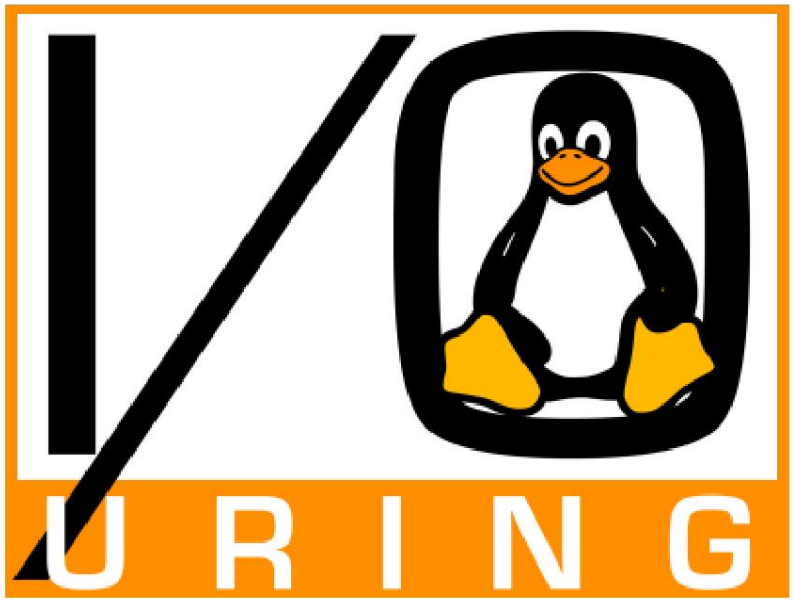
针对不同大小的产物有, 30% ~ 1000% 的提升

总结

- 好的 profile 工具能让你事半功倍
- 任何时间复杂度的算法在的数据规模小的时候差距都不大
- 做算法时间复杂度分析时不能只统计可见的代码,还需要统计函数以及库.

未来展望

使用 io-uring 提高 IO 速度



借鉴 salsa-rs 优化增量构建速度

Native plugin(使用Rust 写高性能的Plugin)

GOTC

THANKS

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE