前端遇上Go

静态资源增量更新的新实践

美团金融 刘洋河

美团点评

自我介绍

刘洋河
美团金融 高级前端工程师

2014年 开始前端职业生涯
2017年 加入美团

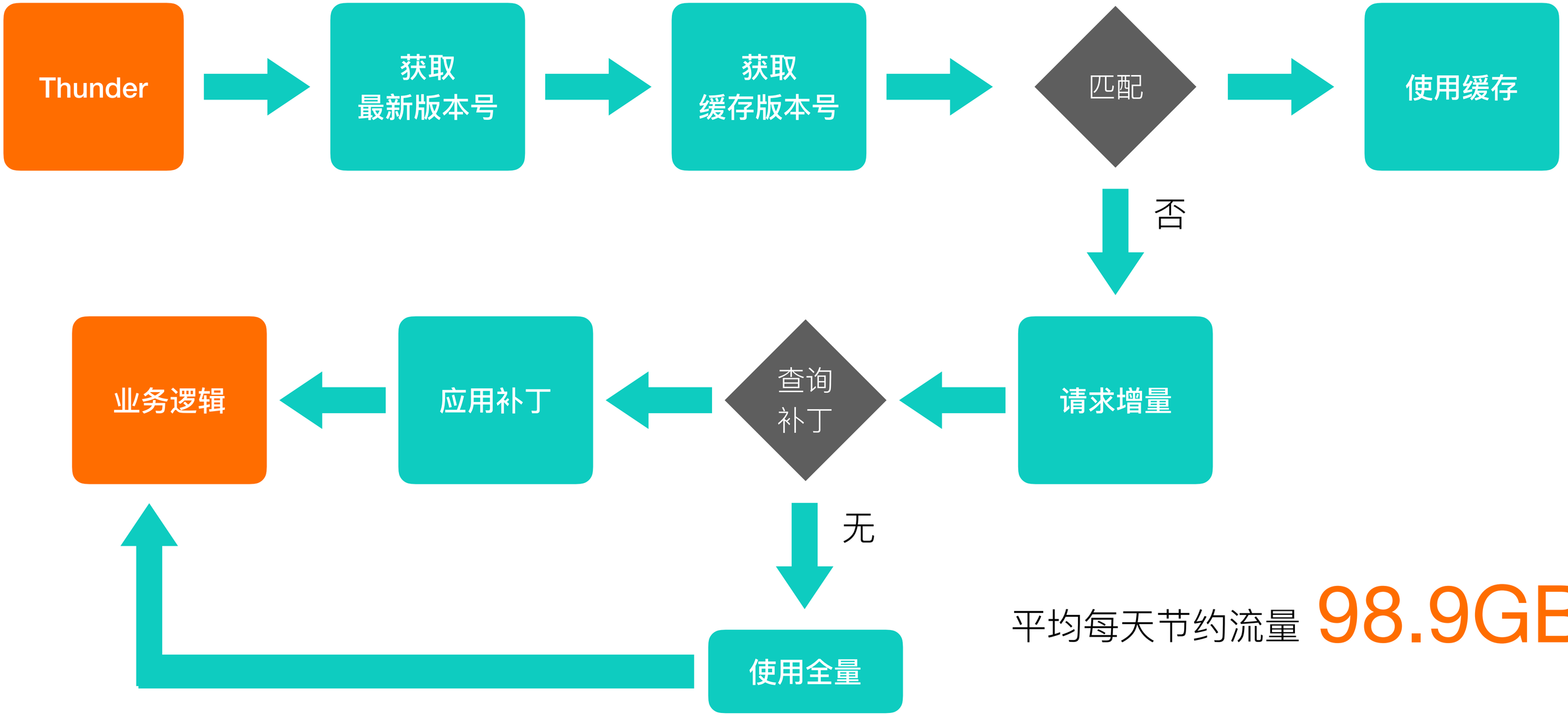
前端与静态资源

以扫码付为例

日均静态资源请求 620W　产生流量 314GB

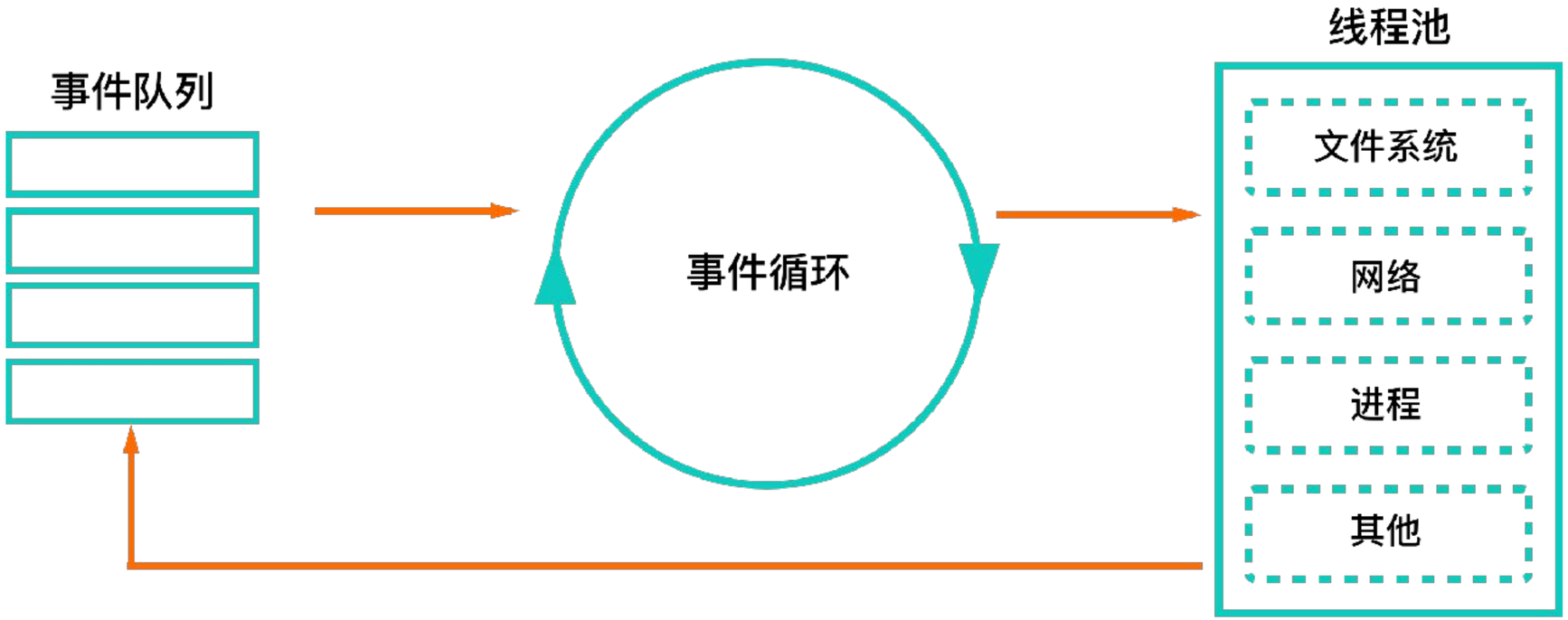
增量更新

Thunder JS



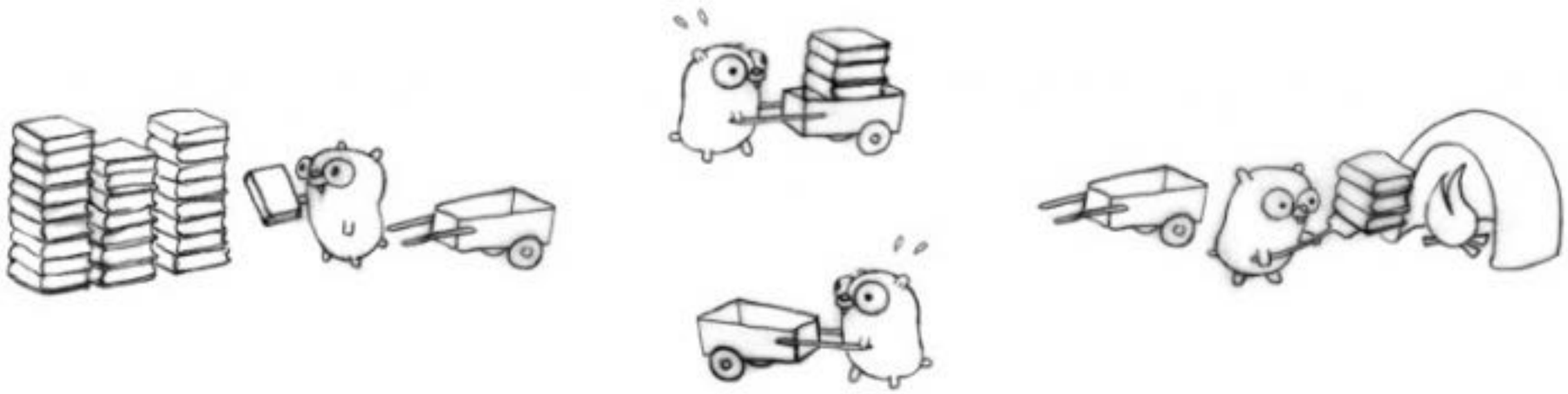
平均每天节约流量 98.9GB

唯快不破



- 字符串的 diff 计算本身是一项非常消耗 CPU 资源的工作
- 基于 Node.js 的 diff 计算，针对长达数百 KB 的资源，耗时可能长达数十秒
- Node.js 适合 I/O 密集型任务，对 CPU 密集型任务比较短板

JavaScript 的任务调度



- 最重的任务在主线程
- JS的优化原则是将任务尽可能拆细
- “拆”的成本有时很高

C++ Add-on / asm.js / WebAssembly

ADD-ON

Node.js 官方的方案

1. 设计给 Node.js 使用
2. 通常用 C++ 写成
3. Node.js 独家支持

asm.js

ASM.JS

Mozilla 的方案

1. 早期的跨浏览器提速尝试
2. 选取JS的子集
3. 时代的眼泪

WEB-ASM

W3C 的标准

1. ASM.js 死后的新方案
2. 更紧凑、接近汇编的机器码
3. 多个编译器支持

实现高性能的无阻塞方案困难

选择语言的考量

GC

类型系统

部署

运行速度

依赖管理

并发处理

调优难度

社区

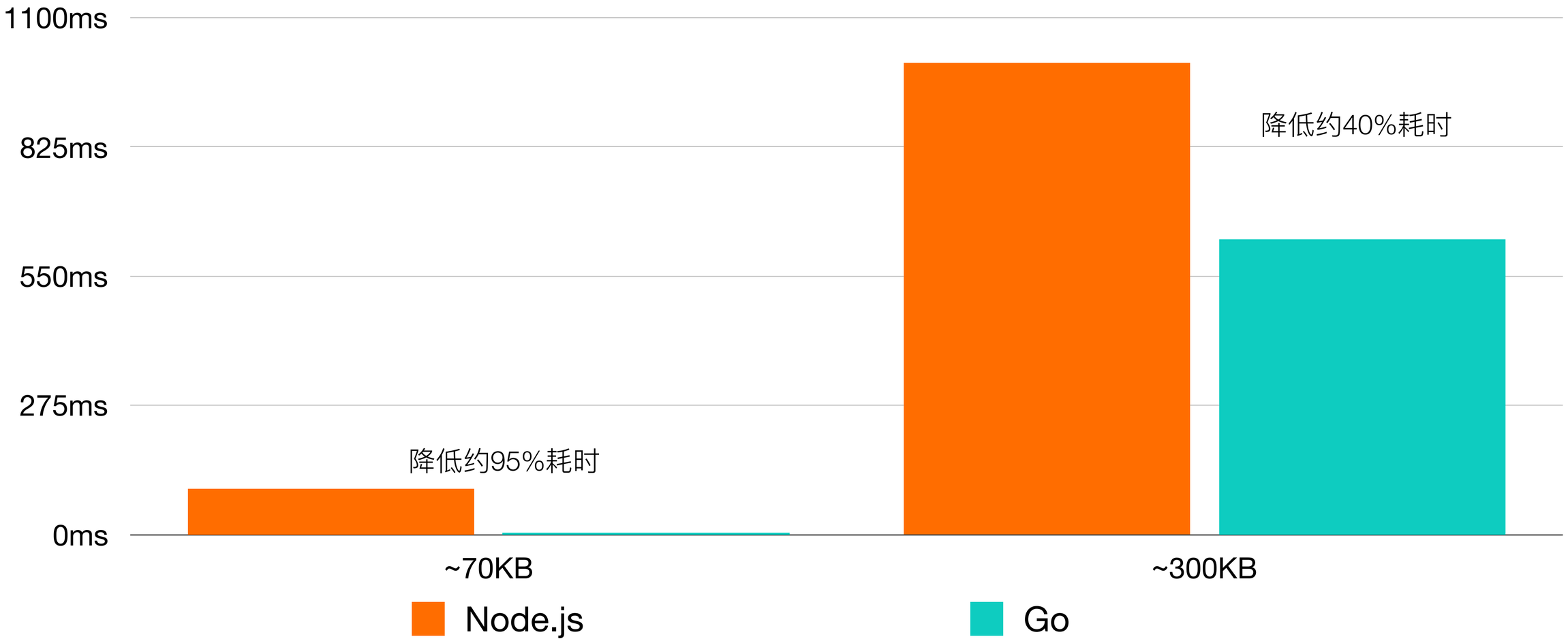
易学

Let's Go

- 静态语言，性能接近于C
- 适合进行I/O与计算并存的业务
- 有良好、易用的并发模型
- 静态分析即可帮助我们发现大量潜在问题
- 调优简单直观
- 有比较健康的社区生态

Go

执行速度



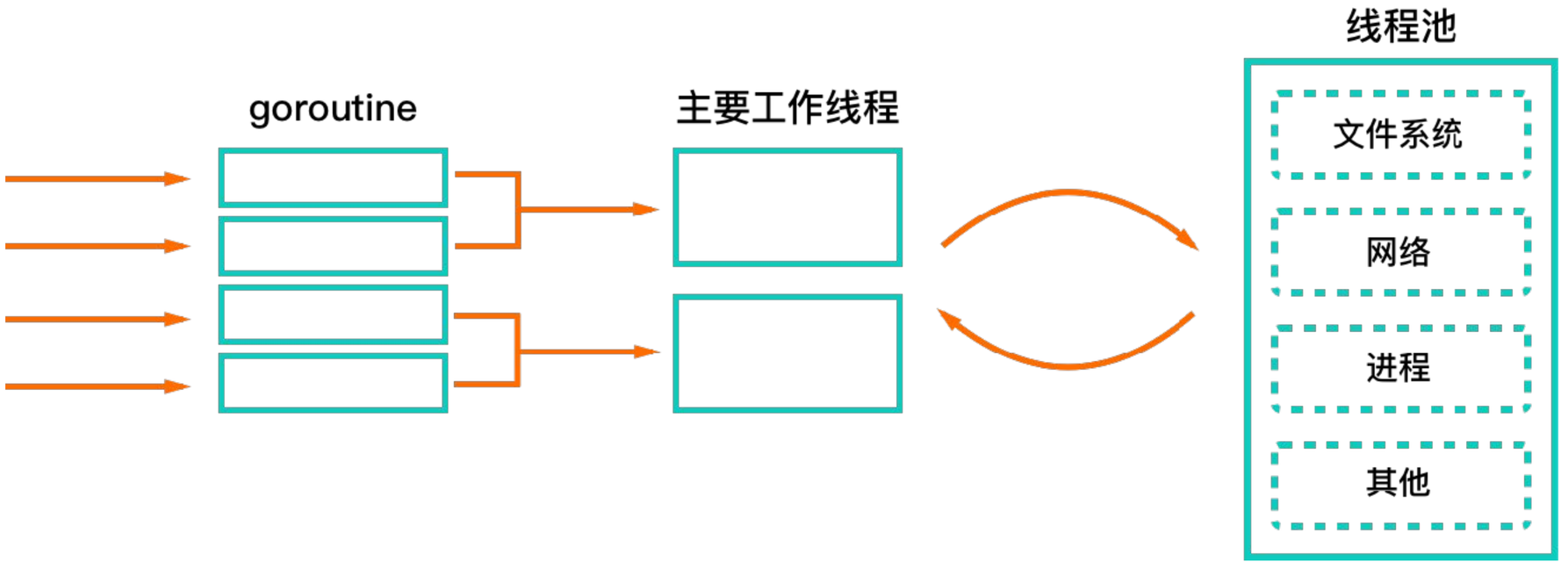
相同diff算法、文件样本下的耗时对比

Goroutine

```go
package main

import "time"
import "fmt"

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

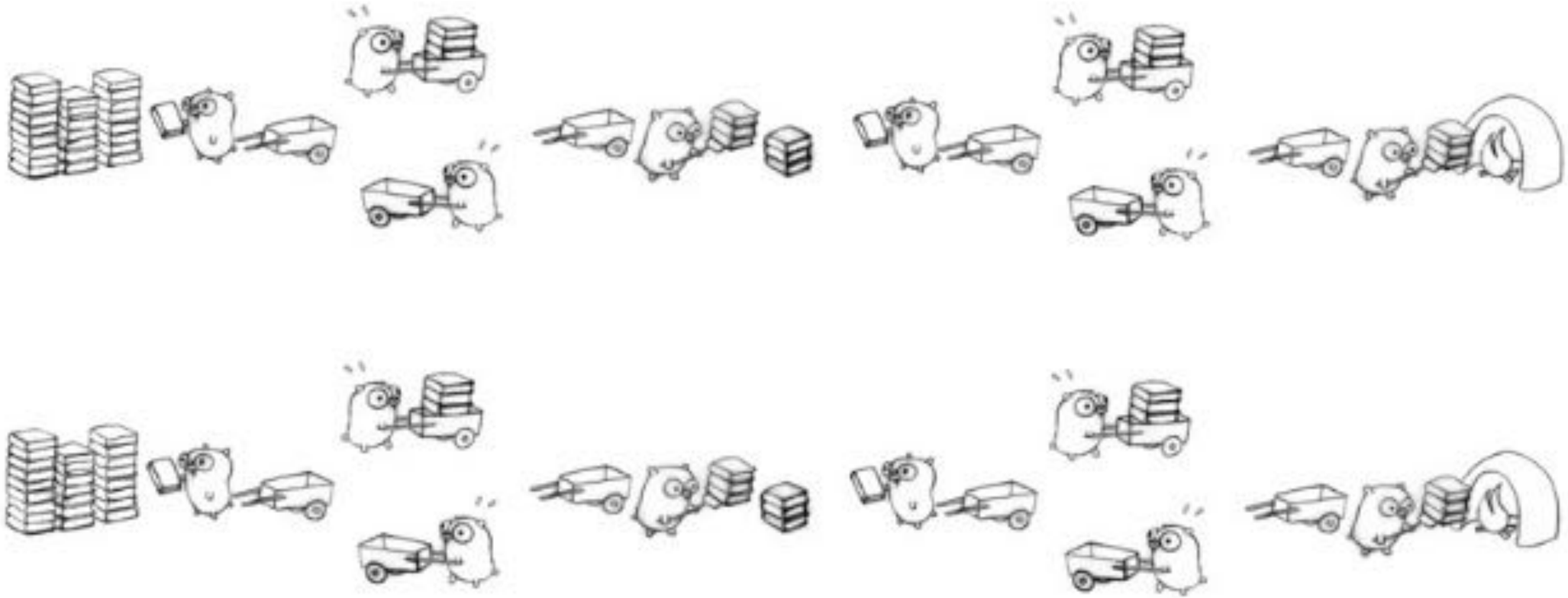
- 每个 goroutine 内部同步执行
- 多个 goroutine 和多线程类似
- 大部分 Web 场景下用不到锁
- 原子变量可以解决绝大部分“计数器”场景
- channel 机制既可以用来同步，也可以用来通信

Go 的并发



- Go 的轻量级线程能够更充分地利用多核资源
- 抢占式调度，避免单个任务影响过大

Go 的并发



类型系统

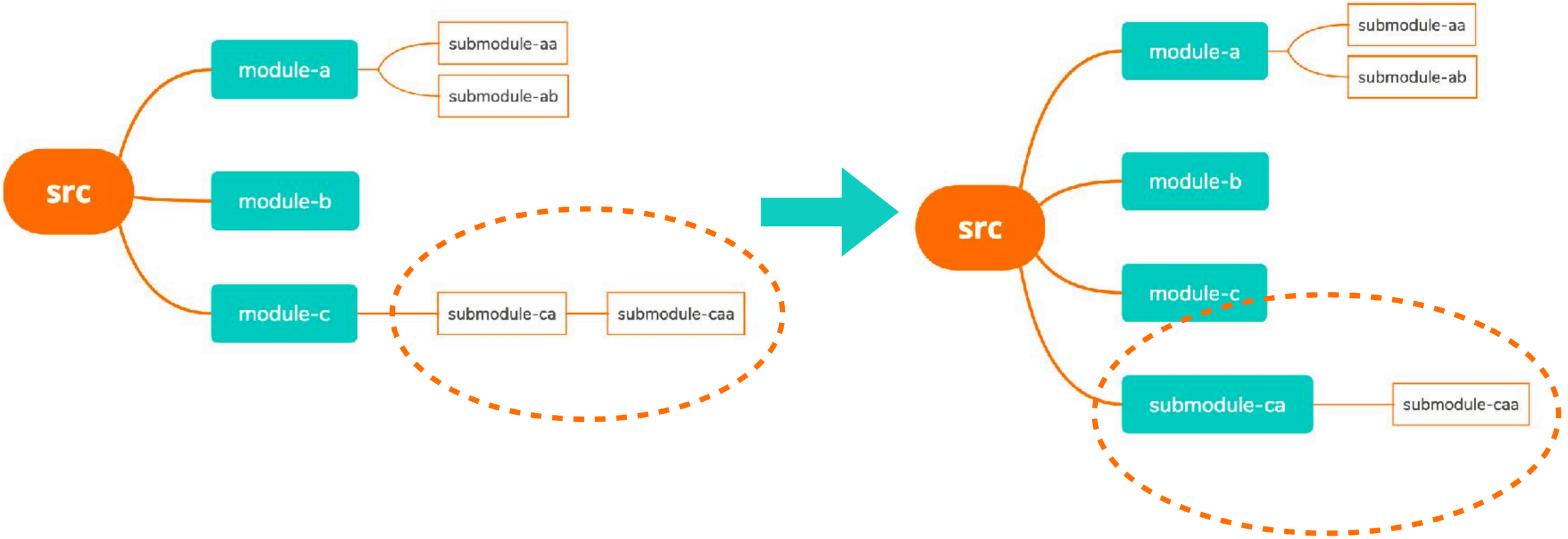
```javascript
var a = 1
var b = 2

const c = "asdasdasd"

console.log(a, b, c)
// 1 2 "asdasdasd"

var p = {
  Name: "Jerry",
  Age: 10,
}

console.log(p);
// {Name: "Jerry", Age: 10}
```

JavaScript

```go
var a int64 = 1
b := 2

const c string = "asdasdasd"

fmt.Println(a, b, c)
// 1 2 asdasdasd

type Person struct {
  Name string
  Age  int
}

p := Person{
  Name: "Jerry",
  Age: 10,
}

fmt.Println(p)
// {Jerry 10}
```

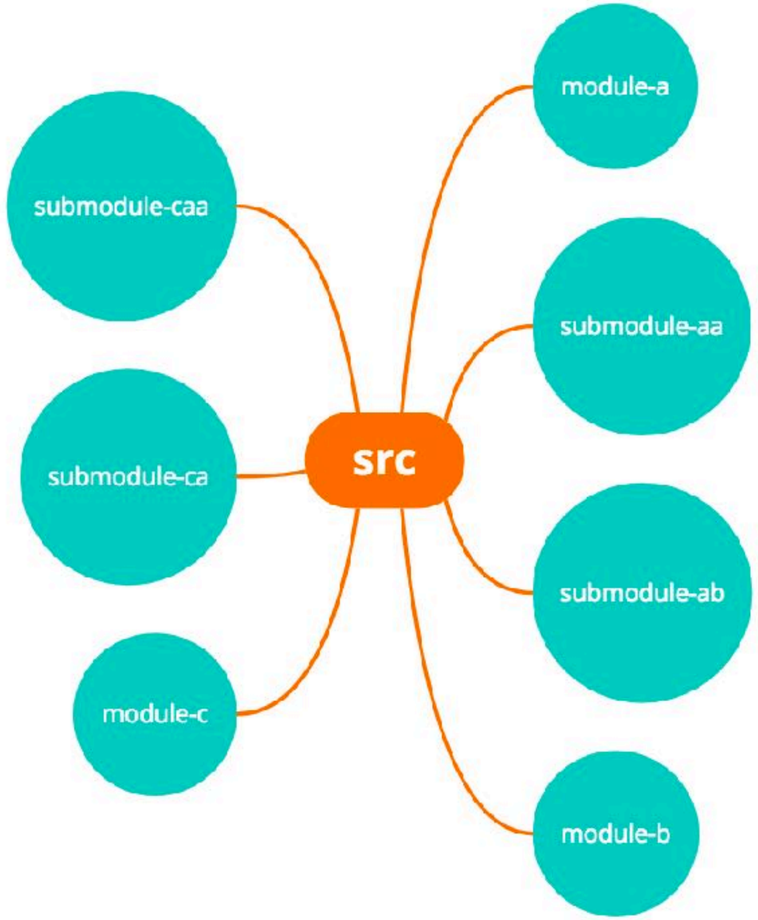
Go

模块



Node.js 模块关系

模块



Go 模块关系

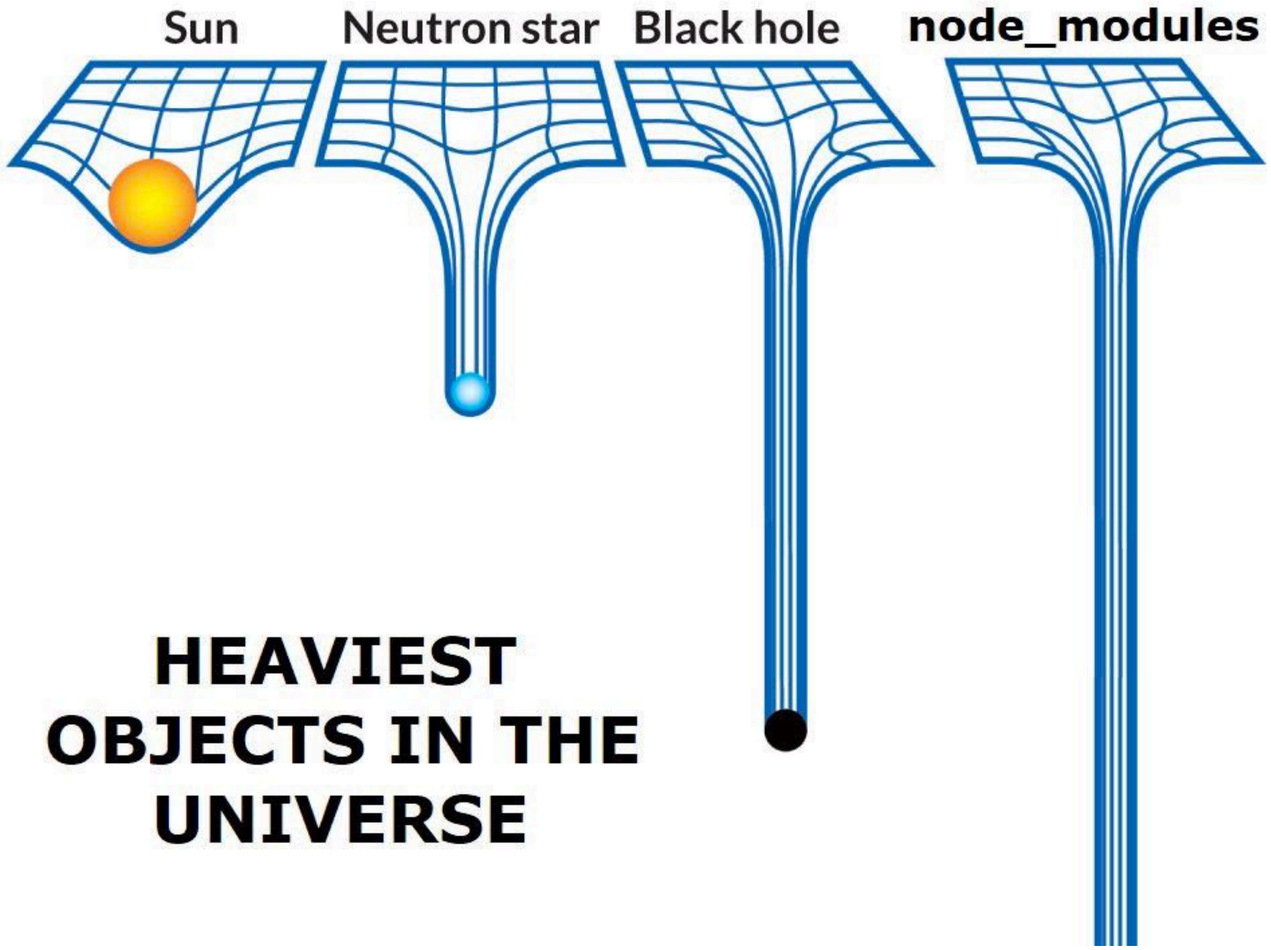
- Go 的官方模块直接按名称引用
- 自定义模块总是相对于根路径引用
- 自定义模块之间**不能用**相对路径引用
- 意味着移动模块需要更多的改动

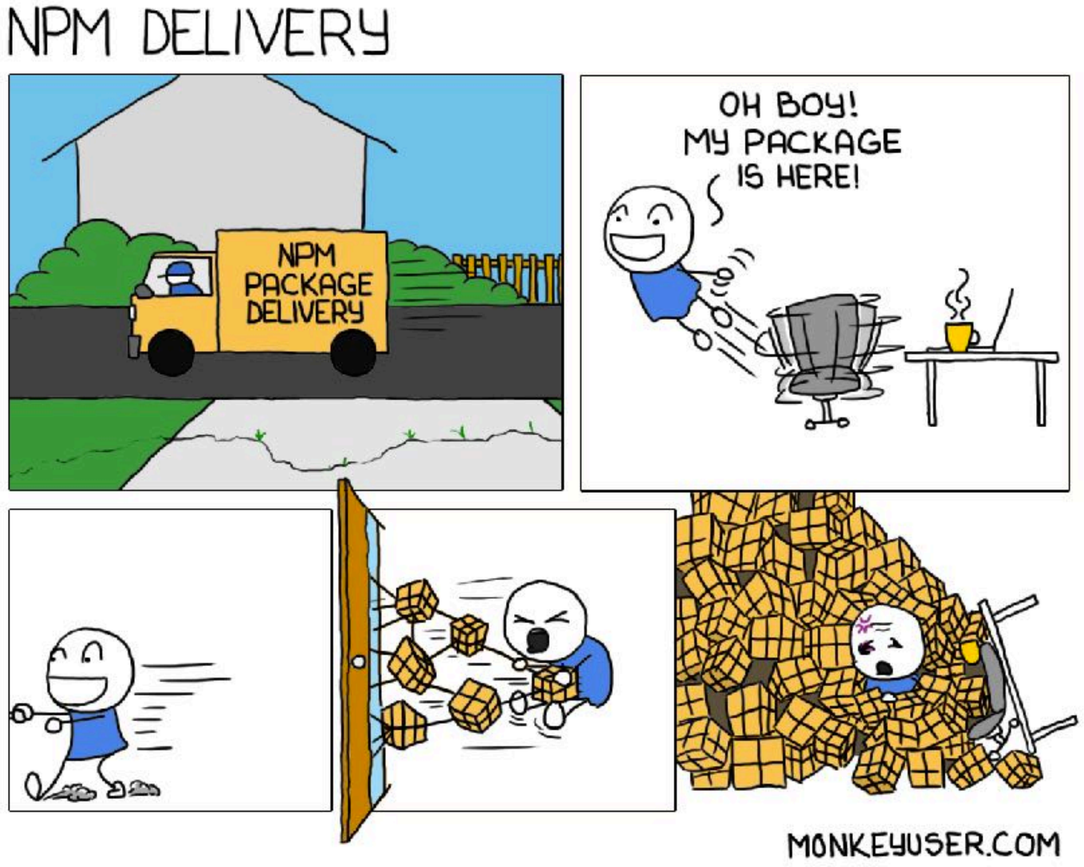
包管理

npm install something --save --registry=http://mirror.something.org

Node.js 的 npm

node_modules?





包管理

go get github.com/someuser/cool

- 没有版本描述文件
- 没有 npm scripts / npx 等价物
- 没有中心包管理节点 - 镜像？不存在的

Go 官方的 go-get

包管理

Package Management

Libraries for package and dependency management.

- dep - Go dependency tool.
- gigo - PIP-like dependency tool for golang, with support for private repositories and hashes.
- glide - Manage your golang vendor and vendored packages with ease. Inspired by tools like Maven, Bundler, and Pip.
- godep - dependency tool for go, godep helps build packages reproducibly by fixing their dependencies.
- gom - Go Manager - bundle for go.
- goop - Simple dependency manager for Go (golang), inspired by Bundler.
- gop - Build and manage your Go applications out of GOPATH
- gopm - Go Package Manager.
- govendor - Go Package Manager. Go vendor tool that works with the standard vendor file.
- gpm - Barebones dependency manager for Go.
- gvt - `gvt` is a simple vendoring tool made for Go native vendoring (aka GO15VENDOREXPERIMENT), based on gb-vendor.
- johnny-deps - Minimal dependency version using Git.
- nut - Vendor Go dependencies.
- VenGO - create and manage exportable isolated go virtual environments.

包管理

glide install github.com/someuser/cool

- 增加了版本描述文件，支持语义化版本号
- 自己写 shell 脚本
- 第三方依赖入库
- 期待官方给出更好的方案

Go 社区的 glide

基础设施与 Go

	Tair	Squirrel	美团云MSS	octo	thrift	内网镜像
Node.js	有npm包	有npm包	有npm包	有npm包	有npm包	有
Go	自研	无	AWS Go SDK	无	需二次开发	git

Tair、Squirrel、octo等公司常用的基础设施大多没有Go版本的SDK

Thrift虽然是一个开源技术方案，但公司的thrift服务实际对开源方案有所改动

部署

- npm install —production
- tar/zip
- dockerize

Node.js 应用的部署

- go build
- dockerize

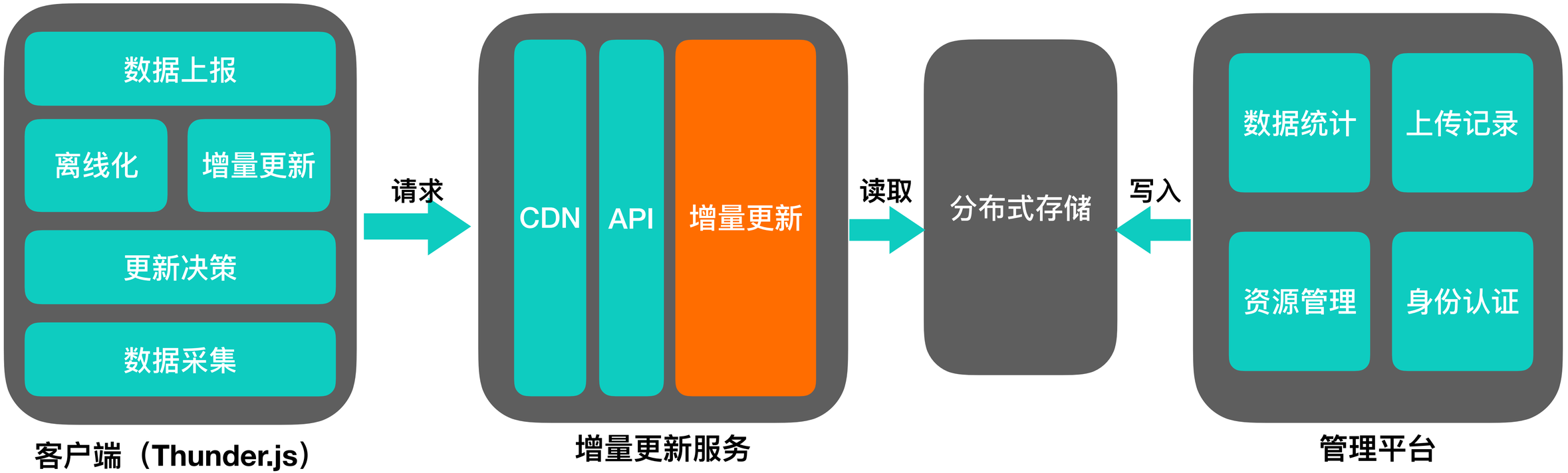
Go 应用的部署

增量计算面临的挑战

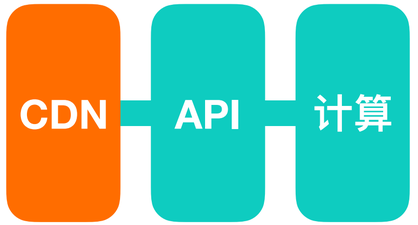
是否使用 Go 就解决了所有问题?

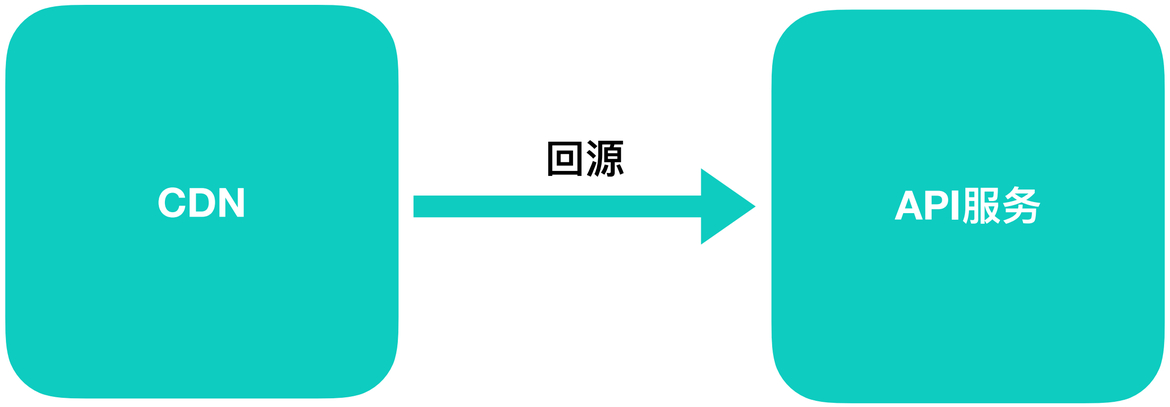
- 如何面对海量突发流量
- 如何容灾

架构设计


数据上报
离线化
增量更新
更新决策
数据采集
客户端（Thunder.js）
请求
CDN
API
增量更新
增量更新服务
读取
分布式存储
写入
数据统计
上传记录
资源管理
身份认证
管理平台

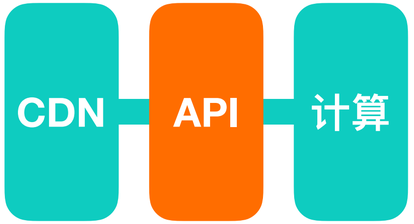
如何面对海量突发流量

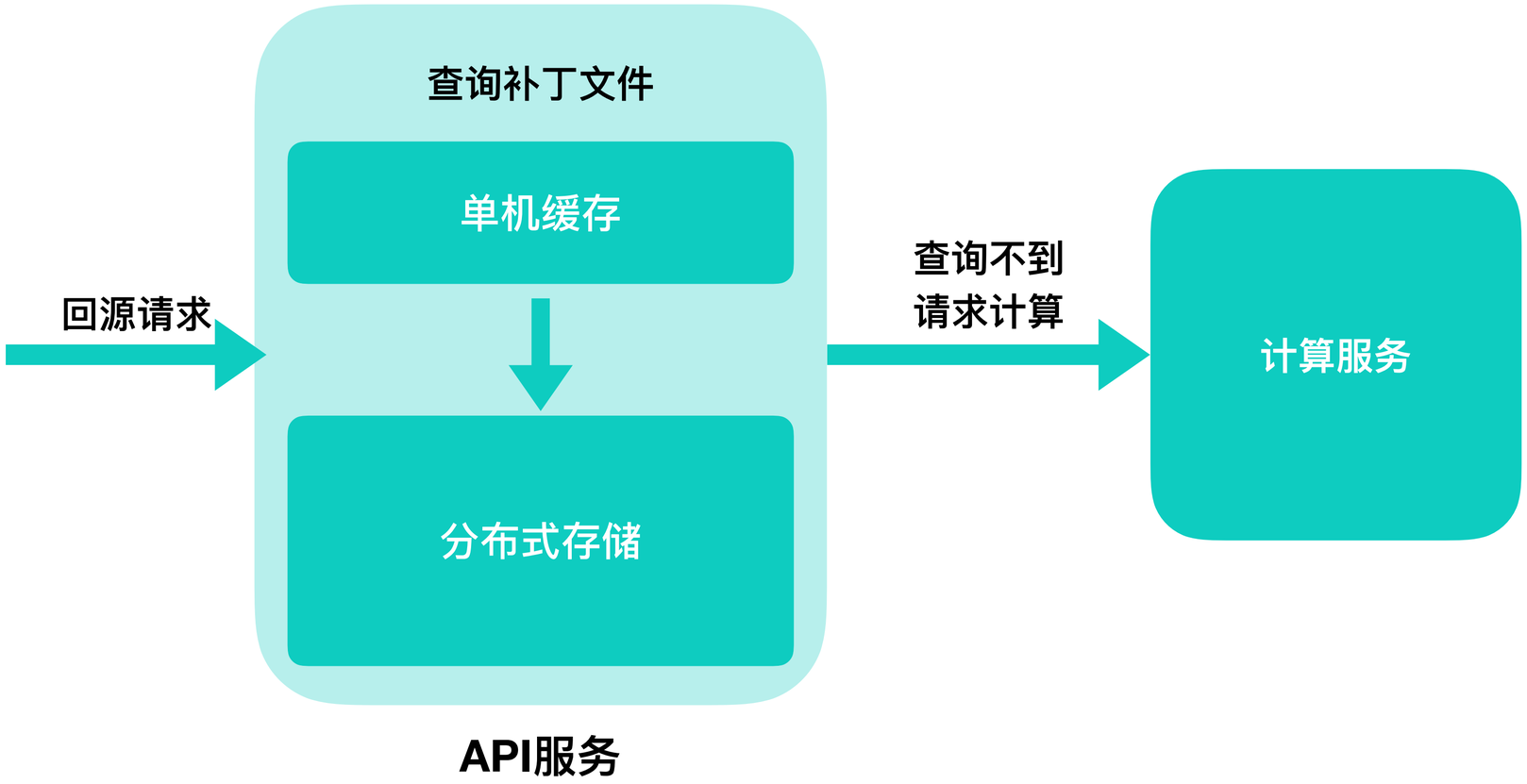




- 在增量更新服务的最外层有CDN进行保护
- 正常情况下，即使有突发流量，也会抵挡绝大部分请求

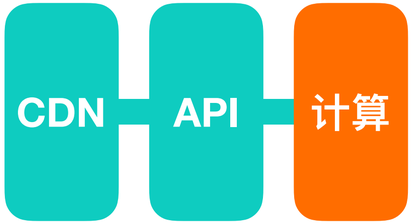
如何面对海量突发流量

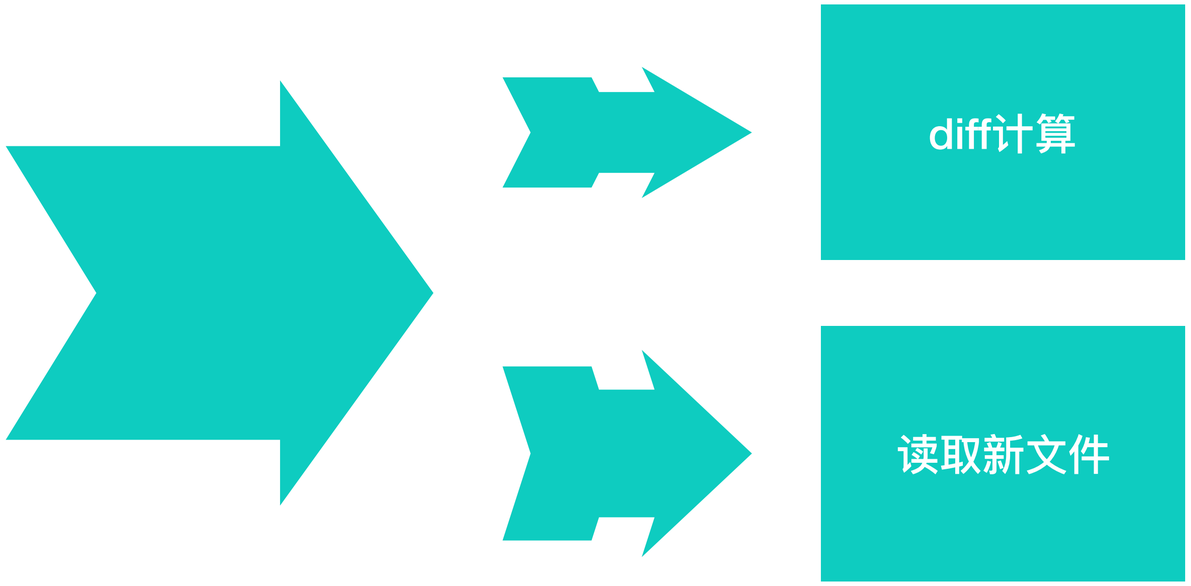




- 将对外的API拆解成单独的服务
- API服务保持只读，化解流量压力，使得计算服务可以专心做计算

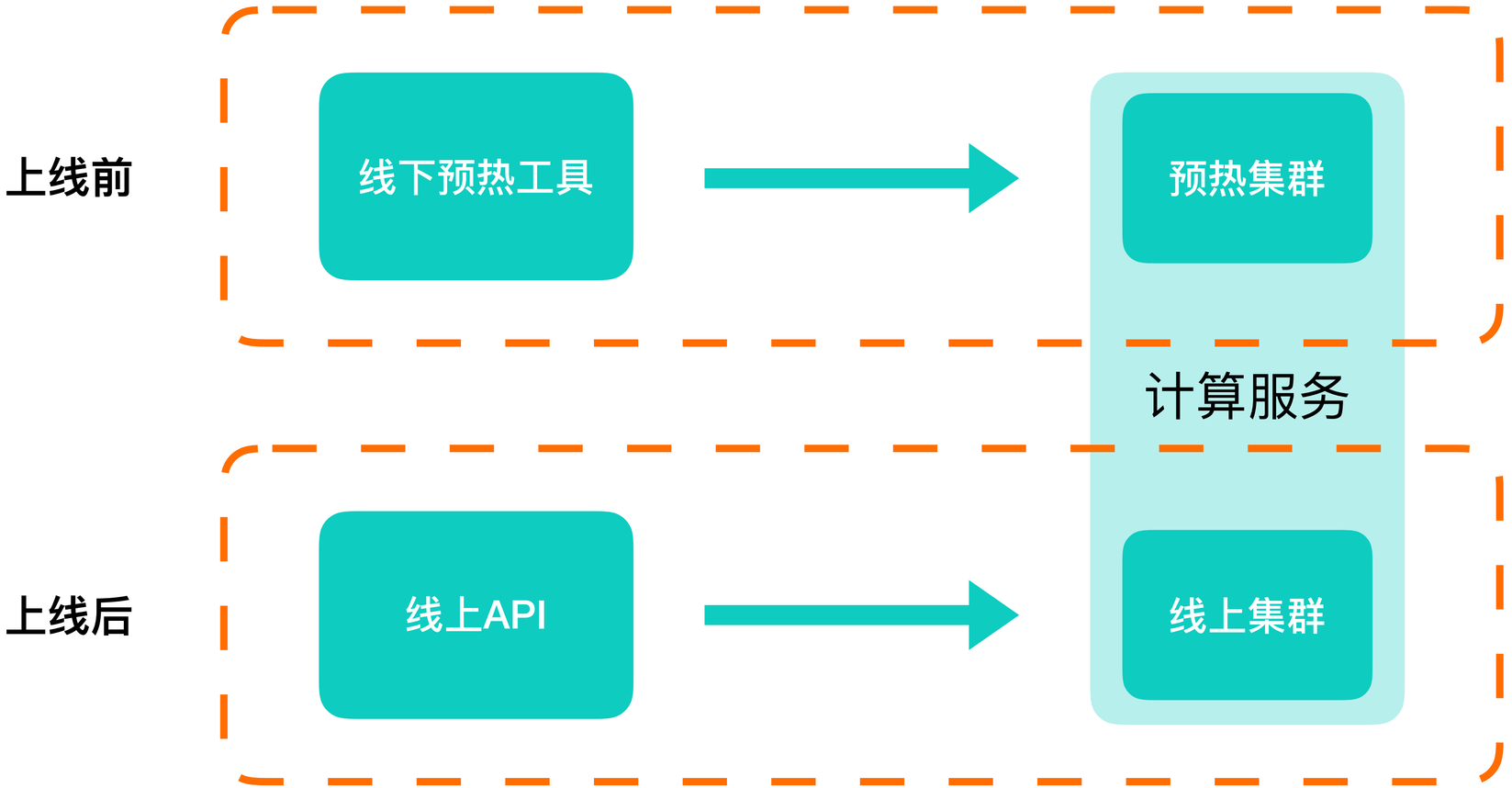
如何面对海量突发流量





- 以压测结果对单机QPS进行限制
- 对于超额流量，直接返回新文件
- 无论哪种结果，最终会缓存在CDN

如何面对海量突发流量



- 上线前对增量进行预热
- 预热服务与主服务隔离
- 只共用存储

如何容灾

- **线路故障**
  - 每一层都有单机内存Cache，减少网络故障影响
- **存储故障**
  - 利用公司稳定存储平台而不是自建存储，避免磁盘坏道等尴尬场面
  - 使用两个不同的分布式存储平台，互为替代
- **CDN故障**
  - 双CDN，隔离来自CDN故障的风险
- **系统故障**
  - CDN会缓存服务的返回，无论是否异常，避免击穿
  - 设计服务降级预案，增量系统瘫痪后请求全量文件，不影响用户使用

我们正在做的

一个易用、稳定、可扩展、数据驱动的静态资源托管服务

回顾与总结

- 不同的语言和工具有不同的用武之地，不要试图用锤子去锯木头
- 更换语言是一个重要的决定，在决定之前首先需要思考是否应当这么做
- 语言解决更多的是局部问题，架构解决更多的是系统问题
- 构建一个系统时，首先思考它是如何垮的

Q&A

- 美团金融持续招收大前端
- Node.js? Service Worker? WebAssembly?
- 欢迎一起实践