GOTC

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE

# OPEN SOURCE , OPEN WORLD #

# 「大前端新趋势」专场

本期议题：探索类型安全的 Node.js Web 框架

马天琦 2021年08月01日

# 自我介绍

GOTC

探索类型安全的 Node.js Web 框架

## 马天琦

- 字节跳动 Web Infra 团队成员
- Farrow 核心开发者

# 探索类型安全的 Node.js Web 框架

GOTC

- 类型安全
- Node.js Web 框架现状
- 当前 API 设计中的类型问题
- Farrow 的类型安全方案
- Farrow 未来的规划
- 总结

# 类型安全 What & Why?

# 类型安全

GOTC

## 什么是类型安全

定义：变量的行为与它的类型相匹配，不存在运行时的类型错误

- 例子 0：访问 null 的属性（null.foo()）
- 例子 1：将 string 类型当作 number 类型做运算
- 例子 2：调用对象中不存在的方法

# 类型安全

## 为什么要追求类型安全

Well typed programs cannot go wrong. ——《A Theory of Type Polymorphism in Programming》Robin Milner 1978.

- 尽可能在编译期通过类型检查提前捕获可能的程序错误，提高代码的健壮性
- 配合编辑器类型提示，类型检查是比单元测试反馈更快、更早、覆盖更全面的实时测试
- 符合类型安全准则的代码，往往是设计更合理、质量更高、编写更优雅的、表达更清晰的

GOTC

# Node.js Web 框架现状

# 现状

Node.js Web 框架

# 当前 API 设计中的类型问题

# Express

GOTC

```
import Express from 'express'

const app = Express()

app.use((req, res, next) => {
  res.end('Hello World')
})

app.listen(3000, () => {
  console.log('Listening on localhost:3000')
})
```

# 类型问题

## Hanging Request

- 可以不响应
- 可以远远挂起请求
- 无法在编译期得到约束和提醒

```
app.use((req, res, next) => {
  // do nothing
});
```

# 类型问题

## Wrong Response

- 无法在编译期约束 header -> body 的响应次序
- 无法在编译期约束只发送一次 body

```
app.use((req, res, next) => {
  // body 提前发送
  res.end('xxx');
  // header 必须在 body 之前发送，这里报错或 warning
  res.header('Content-Type', 'text/html');
  // body 只能发送一次，这里报错或 warning
  res.json({ error: true });
});
```

# 类型问题

## Monkey Patching

- 修改 req/res 污染全链路中间件的类型
- 动态追加属性或方法，与静态标注的类型有本质矛盾
- 静态类型决定能否赋值给属性，而非属性赋值决定是否包含特定类型

```
app.use((req, res, next) => {
  // what type of res.locals
  res.locals.a = 1;
  next();
});

app.use((req, res, next) => {
  // what type of res.locals
  console.log(res.locals.a);
});
```

# 类型问题

## No Runtime Validation

- TypeScript 类型在编译后都被抹去
- 在 Input/Output，parse/stringify 中需要运行时类型检查

```
app.use((req, res, next) => {
  if (req.body && typeof req.body === 'object') {
    if (req.body.id && typeof req.body.id === 'string') {
      // do something

    }
  }
});
```

# 类型问题

## Poor Type Inference

- 类型推导不友好
- 自动类型转换不友好
- 需要编写繁琐的类型转换和防御逻辑

```
app.get('/user/:userId', (req, res, next) => {
  req.params.userId; // no type infer
  const params = req.params as { userId: string };

  // 必须每次手动 transform
  const userId = Number(params.userId);
});
```

# 类型问题

## 总结

- Hanging Request（请求意外挂起）
- Wrong Response（错误响应内容）
- Monkey Patching（篡改对象属性）
- No Runtime Validation（无运行时验证）
- Poor Type Inference（不友好的类型推导）
- etc...

# 类型问题

GOTC

## 总结

- Hanging Request（请求意外挂起）
- Wrong Response（错误响应内容）
- Monkey Patching（篡改对象属性）
- No Runtime Validation（无运行时验证）
- Poor Type Inference（不友好的类型推导）
- etc...

只靠 *.d.ts，并不能获得充分的类型友好和类型安全特性

# 类型问题

GOTC

## 总结

- 基于 Express/Koa 可以用打补丁的方式解决一两种类型问题，但不能从根本上解决问题
- Fastify 提供了基于 JSON Schema 的运行时校验请求内容的方案，但方案与类型系统不贴合
- 要充分解决系统性问题，则需要基于 TypeScript 做全盘的思考
- 类型优先开发（Type-First Development）
- 类型驱动开发（Type-Driven Development）

# 类型问题

## 类型安全的服务端框架设计目标

- Prevent Hanging Request（阻止请求意外挂起）
- Refuse Wrong Response（拒绝错误响应内容）
- No need to Monkey-Patching（无需篡改对象属性）
- Embedded Runtime-Validation（内置运行时验证）
- Excellent Type Inference（出色的类型推导）
- etc...

# 类型问题

GOTC

## 类型安全的服务端框架设计目标

Farrow作者：做到之前做不到的，做好之前能做到的

- Prevent Hanging Request（阻止请求意外挂起）
- Refuse Wrong Response（拒绝错误响应内容）
- No need to Monkey-Patching（无需篡改对象属性）
- Embedded Runtime-Validation（内置运行时验证）
- Excellent Type Inference（出色的类型推导）
- etc...

# Farrow 作者介绍

GOTC

## 工业聚

- 前端架构师
- 函数式编程爱好者
- 开源实践者

- react-lite: React v15.0 的轻量级实现(2016)
- react-imvc：React SSR 框架(2017)
- graphql-bff：基于 GraphQL 的 BFF 框架(2019)
- pure-model: 面向 Model 层的逻辑跨端框架(2020)
- farrow：类型安全的 Node.js Web Framework(2021)

GOTC

# Farrow 的类型安全方案

# 类型安全方案：farrow-http

GOTC

## Prevent Hanging Request & Refuse Wrong Response

- Response as Return Type

```
import { Http, Response } from 'farrow-http';

const http = Http();

http.use((request) => {
  // response is return type
  return Response.text(request.pathname);
});
```

# 类型安全方案：farrow-http

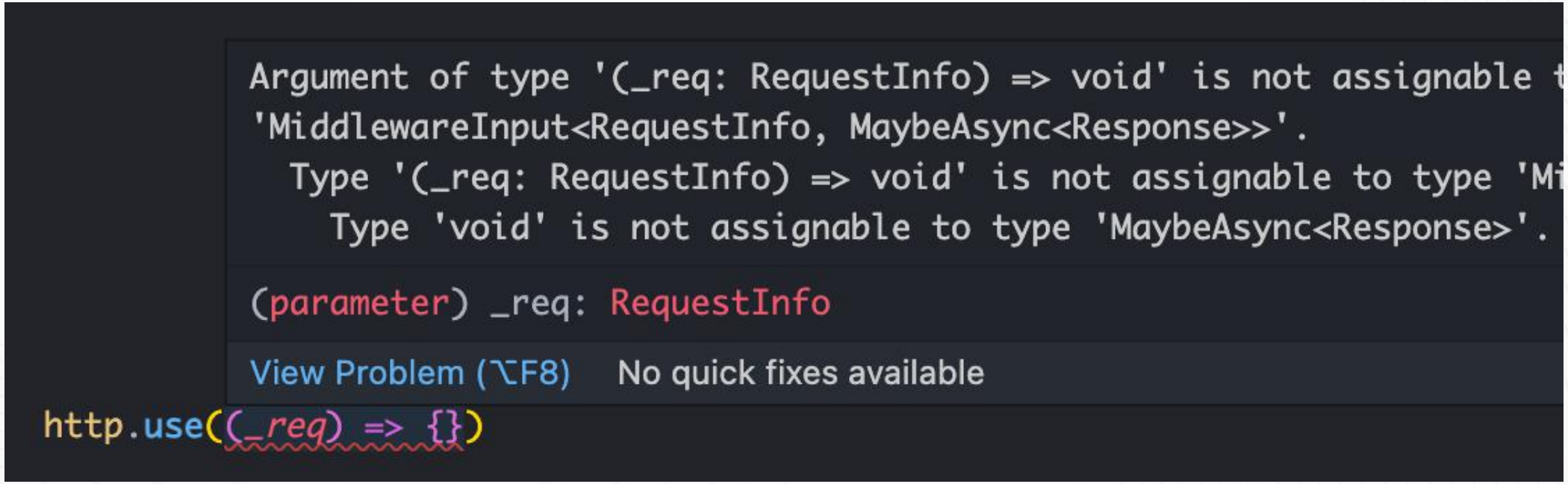
## Prevent Hanging Request & Refuse Wrong Response

- TypeScript 能够检查函数返回值类型是否满足约束
- 忘记响应或者响应错误的类型，都能得到类型检查

# 类型安全方案：farrow-http

GOTC

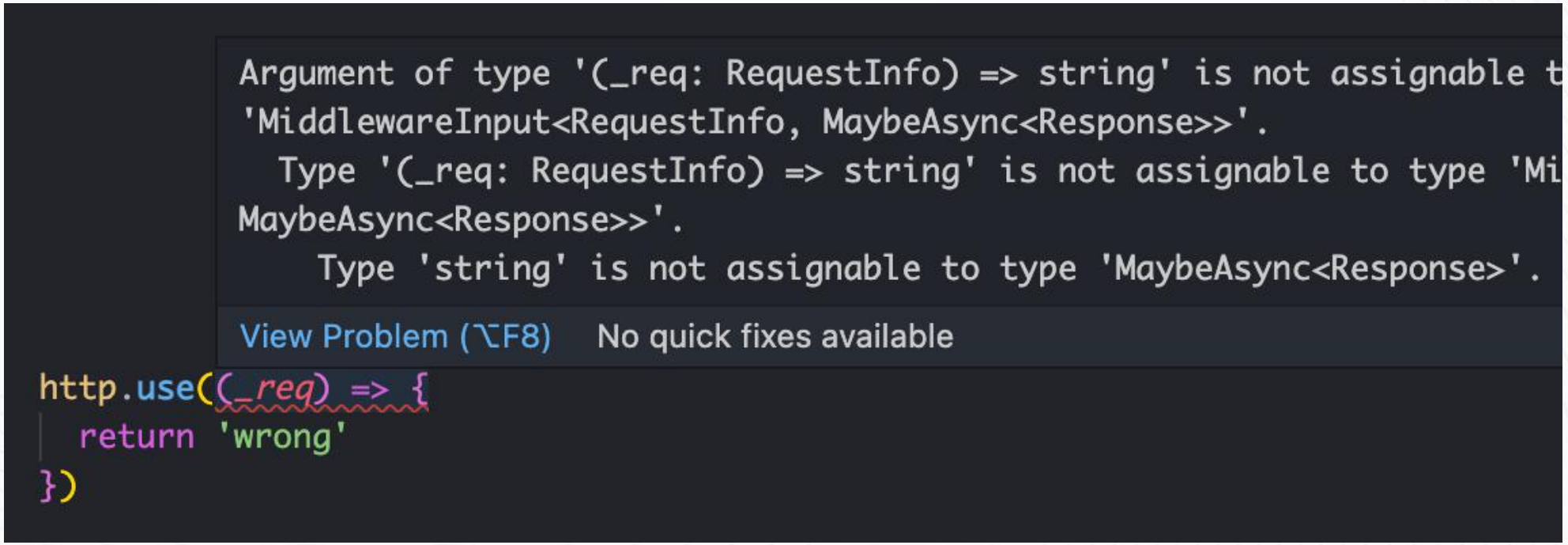
## Prevent Hanging Request & Refuse Wrong Response



# 类型安全方案：farrow-http

GOTC

## Prevent Hanging Request & Refuse Wrong Response



# 类型安全方案：farrow-http

GOTC

## Prevent Wrong Response

- 新的响应对象模型
- Response.text 等方法构造数据，多次使用将合并
- 最终统一按照正确方式处理 header 和 body

```
import { Http, Response } from 'farrow-http';

const http = Http();

http.use((request) => {
  // response is return type
  return Response.text(request.pathname)
    .header('abc', 'efg')
    .text('changed text');
});
```

# 类型安全方案：farrow-http

GOTC

## No need to Monkey-Patching

- 新的请求对象结构
- request 参数并非原生 req 对象，

  而是从中提取的 plain data
- next(newRequest) 可以向后传递新

  的 request 对象，无需修改原对象

```
import { Http, Response } from 'farrow-http';

const http = Http();

http.use((request, next) => {
  return next({
    ...request,
    pathname: '/another/pathname',
  });
});

http.use((request) => {
  request.pathname; // is equal to /another/pathname
});
```

# 类型安全方案：farrow-http

GOTC

## No need to Monkey-Patching

- next() 返回下游中间件的 response 对象
- 无需修改 response
- Immutable 比 Mutable 更加类型友好

```
import { Http, Response } from 'farrow-http';

const http = Http();

http.use((request, next) => {
  const response = await next();
  // 合并，组合，过滤，拼装新的 response
  return Response.header('abc', 'efg').merge(response);
});

http.use((request) => {
  return Response.text('hello world!');
});
```

# 类型安全方案：farrow-http

GOTC

No need to Monkey-Patching

- Context + Hooks
- 跨中间件传递 Context Data
- 无需挂载到 req/res 对象
- 利用 Async Hooks 让 Context + Hooks 更强大的

```ts
import { Http, Response, createContext } from 'farrow-http';

const http = Http();

// 创建 Context
const AuthContext = createContext<Auth | null>(null);

// 更新 Context
http.use(async (request, next) => {
  AuthContext.set(await getAuth(request));
  return next();
});

// 不管中间插入多少中间件，request/response 类型都不会污染

// 消费 Context
http.use((request) => {
  // 跨中间件访问 context 数据
  const auth = AuthContext.get();
  return Response.text('hello world!');
});
```

```ts
import Express from 'express'

const app = Express()

app.use(async (req, res, next) => {
  res.locals.auth = await getAuth(req)
  next()
})

app.use((req, res) => {
  const auth = res.locals.auth as AuthType
  res.end('hello world!')
})
```

```ts
import { Http, Response, createContext } from 'farrow-http';

const http = Http();

const AuthContext = createContext<Auth | null>(null);

http.use(async (request, next) => {
  AuthContext.set(await getAuth(request));
  return next();
});

http.use((request, next) => {
  const auth = AuthContext.get();
  return Response.text('hello world!');
});
```

# 类型安全方案：farrow-http

GOTC

## Embedded Runtime-Validation & Excellent Type Inference

- 提供基于 Schema 的 Runtime Validation 机制
- 通过 Schema 描述 request 的 Shape
- 通过 type infer 推导出 request type

```ts
import { Http, Response } from 'farrow-http';
import { Int } from 'farrow-schema';

const http = Http();

http
  .match({
    pathname: '/user',
    method: 'post',
    body: {
      userId: Int,
      userName: String,
    },
  })
  .use((request, next) => {
    // request.body is { userId, userName }
    console.log('userId', request.body.userId);
    console.log('userName', request.body.userName);
    return Response.text('hello world!')
  });
```

# 类型安全方案：farrow-http

GOTC

## Embedded Runtime-Validation & Excellent Type Inference

- 基于 TypeScript 4.1 的 Template Literal Type
- 从 URL 模式中提取类型
- 自动识别 params 和 query
- 自动将 String 转换成标记的 Int、Boolean 等类型

```
import { Http, Response } from 'farrow-http';

const http = Http();

http
  .get('/greet/<name:string>?<age:int>&farrow=type-safety')
  .use((request, next) => {
    // type infer for request from url
    console.log('name', request.params.name);
    console.log('age', request.query.age);
    console.log('farrow', request.query.farrow);
    return Response.text('hello world!');
  });
```

# 类型安全方案：farrow-http

GOTC

Farrow 运用 Type-First Development 思想和 React 启发的函数式/immutable 理念，系统性地提升了 Web Framework 的类型安全水平

- Prevent Hanging Request（阻止请求意外挂起）√
- Refuse Wrong Response（拒绝错误响应内容）√
- No need to Monkey-Patching（无需篡改对象属性）√
- Embedded Runtime-Validation（内置运行时验证）√
- Excellent Type Inference（出色的类型推导）√
- etc...

# 类型安全方案

GOTC

## 新的挑战

以上方案只优化了 Service Side 的类型安全，只解决了一半问题

- End-to-end Typing 将是一个新问题
- Client Side 如何复用 Service Side 的类型?
- Client Side 类型如何跟 Service Side 保持一致和同步?

# 类型安全方案

GOTC

## 新的挑战

重新思考 BFF（Backend For Frontend） 应该为前端提供什么？

- 传统 BFF：为前端提供 Data
- 现代 BFF：为前端提供 Data 和 Type
- 下一代 BFF：为前端提供 Data，Type 和 Code

# 类型安全方案：farrow-api

GOTC

## 新的挑战

Introspection + Codegen

- 提供类似 GraphQL 的 Introspection 机制
- 支持拉取 API 的 Schema 数据
- 通过 Codegen 生成 Typescript Type 和 HTTP Client Code

# 类型安全方案：farrow-api

GOTC

## Introspection + Codegen

```
const Input  = Struct({
  id: String
})

const Todo = Struct({
  content: String,
  createTime: Number
})

const Output = Struct({
  type: Literal('GetTodosSuccess'),
  todos: List(Todo)
})

const getTodos = Api({
  input: Input,
  output: Output,
})
```

生成

```
type Input  = {
  id: string
}

type Todo = {
  content: string,
  createTime: number
}

type Output = {
  type: 'GetTodosSuccess',
  todos: Array<Todo>
}

const getTodos = (
  input: Input
): Promise<Output> => {
  /** */
}
```

# 类型安全方案：farrow-api

GOTC

## 定义 Schema

```
export const getTodos = Api({
  input: GetTodosInput,
  output: GetTodosOutput,
})
```

```
export const GetTodosInput  = Struct({
  id: String
})

export const Todo = Struct({
  content: String,
  createTime: Number
})

export const GetTodosSuccess = Struct({
  type: Literal('GetTodosSuccess'),
  todos: List(Todo)
})

export const UnknownID = Struct({
  type: Literal('UnknownID'),
  message: String
})

export const GetTodosOutput = Union(
  GetTodosSuccess,
  UnknownID,
)
```

# 类型安全方案：farrow-api

GOTC

开发接口

```
getTodos.use(({ id }) => {
  const limit = LimitContext.get()

  if (mockData[id]) {
    return {
      type: 'GetTodosSuccess',
      todos: mockData[id].slice(0, limit),
    }
  } else {
    return {
      type: 'UnknownID',
      message: 'unknown id',
    }
  }
})
```

# 类型安全方案：farrow-api

GOTC

## 客户端调用

```
import { api } from './client'

api.getTodos({ id: 'foo' }).then((res) => {
  console.log(res)
})
```

# 类型安全方案：farrow-api

GOTC

## 客户端生成代码

```ts
import { createApiPipelineWithUrl, ApiInvokeOptions } from 'farrow-api-client'

export type GetTodosInput = {/** */}

export type GetTodosSuccess = {/** */}

export type UnknownID = {/** */}

export type Todo = {/** */}

export const url = 'http://localhost:3000/api'

export const apiPipeline = createApiPipelineWithUrl(url)

export const api = {
  /**
   * @remarks get note content
   */
  getTodos: (input: GetTodosInput, options?: ApiInvokeOptions) =>
    apiPipeline.invoke(
      { type: 'Single', path: ['getTodos'], input },
      options
    ) as Promise<GetTodosSuccess | UnknownID>,
}
```

# 类型安全方案：farrow-api

GOTC

## 客户端调用

```ts
import { api } from './client'

api.getTodos({ id: 'foo' }).then((res) => {
  console.log(res)
})
```

```ts
api.getTodos({ id: 'foo' }).then((res) => {
  console.log(res)
})
```

```ts
(parameter) res: GetTodosSuccess | UnknownID
```

GOTC

# Farrow 的未来规划

# 未来规划

GOTC

## 生态

- 业务框架现状
- 丰富 Middleware/Hooks
- Demo + 最佳实践
- Playground

## 基础能力

- Federation —— 接口聚合
- 结合 ORM，进一步提升类型安全
- 提供强大便捷的 Mock 能力

# 类型安全方案

GOTC

## Bonus: farrow-express & farrow-koa

通过 adapter 可复用 Express/Koa 等生态

- farrow-express：将 Farrow 运行在 Express App 上
- farrow-koa：将 Farrow 运行在 Koa App 上

# 总结

GOTC

- 类型安全的定义及其价值
- 当前 Node.js Web 框架中存在的类型问题
- Farrow 如何通过类型优先和函数式的思路，系统性地改善类型问题
- Farrow 如何贯通前后端类型
- 现在立刻能就在 Express/Koa 等应用中使用 Farrow 的方式
- Farrow 以及其它追求类型安全的框架将来要解决的问题

GOTC

# THANKS

全球开源技术峰会

THE GLOBAL OPENSOURCE TECHNOLOGY CONFERENCE