♫ 类型安全的 web 框架 ♫

荀洪道

二〇二四年一一月一六日

介绍

ts 已经流行很长时间了，但推荐开启严格模式的框架少之又少，大部分都是披着 ts 的 js 框架， 只能提供十分有限的类型检查，无法发挥静态检查的优势； 可能还因为 ts 的限制，无法写出 js 轻易做到的代码，导致代码十分丑陋。

现在框架流行的做法

框架都需要扩展能力，不同框架做法不一致，但一般在`.d.ts`里声明新的扩展， 典型代表为 express.js：

```
declare global {
  namespace Express {
    export interface Request {
      language?: Language;
      user?: User;
    }
  }
}
```

hono 选择显式声明，通过泛型参数传入：

```
type Bindings = {
  TOKEN: string
}

type Variables = {
  user: User
}

const app = new Hono<{
  Bindings: Bindings
  Variables: Variables
}>()

app.use('/auth/', async (c, next) => {
  const token = c.env.TOKEN // token is `string`
  // ...
  c.set('user', user) // user should be `User`
  await next()
})
```

无论选择哪种方式，都是手动标注运行时的值，有可能标注与运行时类型对不上，造成难以察觉的 bug。 比如上面 hono 的例子，不 `set` 而是 `get` 一个 User，其运行时结果是 undefined， 然而根据标注却是 User，这就是一个隐患。

实现的原理

为了解决上面的问题，我们需要从全新角度审视。

一般的框架只有一种状态（全局状态，或一个请求的状态），无论哪里的调用链，都会受这个状态的钳制，例如上述的 `User`，本该是 undefined，却被提示是 User。 换言之，我们需要多种状态，根据每一

步的调用，会生成新的状态，影响到下一次的调用—— 我们不可避免地开始类型体操，不过不要紧，不很会复杂，仅会是类型的累加。

同时一个请求有可能在任意时间、任意地点返回，所以我们需要立刻返回的机制， 如果采用链式调用，`return` 是不够用的，它仅能从当前函数返回； js 没有 CPS，也无法 call/cc 快速返回； `throw` 虽然可行，但不“优雅”。 为了解决这个问题，需要引入 Either Monad。

阅读要求

基于上述原理，需要这些前置知识（要求从高到低）:

1. Reader Monad，这是重中之重，后面的抽象都基于此；不理解可以先阅读《typescript 上的 Reader》。
2. Monad Transformers 常用的 monad，如 Either 和 Maybe。下文用到的 Either 和 Maybe 皆出于 purify-ts。
3. 一点点的 Haskell 知识。
4. TypeScript 简单的类型体操，不会很复杂，知道 `A & B` 的含义即可。
5. 一点点微小的后端知识。

我们从最基础开始实现，随着更复杂的要求，会反覆修改原来的定义，到时需要前后对照。

中间件

中间件的抽象

开发一个 web 框架，本应关注到路由定义，但我们先换个角度，看一个应用很少、但又需要的流程：中间件。

很多框架都会提供中间件，express、hono 不外如是； 不同的框架的中间件表现形式不一样，但做的事是一样的，相当于某个阶段的统一处理流程； 中间件可以理解为处理某个业务的同样的代码，中间件之于框架，相当于函数之于程序。

我们具体看一下中间件的调用形式，就以 express 举例（因为 express 中间件简单、易懂）:

```
app.use((req, res, next) => {
  const user = getUser(req);
  if (!user) {
    return res.end("not found");
  }

  req.user = user;
  next();
});

app.use((req, res, next) => {
  const { user } = req; // 从上个中间件获取
  consle.log(user);
  next();
});
```

可以看到下面的中间件可以依靠着上个中间件的中间状态，如果我们把整个中间件定义放到外面：

```
const f1 = (req, res, next) => { ... };
const f2 = (req, res, next) => { ... };

app.use(f1);
app.use(f2);
```

此时的中间件就以具名函数表示（之前是匿名函数），它们的入参十分一致。稍微熟悉函数式的人，可以就想到组合 f1 和 f2 了。

```
const f3 = (req, res, next) => {
  f1(req, res, next);
  f2(req, res, next);
};

app.use(f3);
```

这种写法好像与上面等价，换言之，中间件也是可以组合的。这也是必然的，因为中间件归根结底，就是普通的函数。我们再回头看 f1 和 f2，他们的入参一致，好像可以随便调整调用顺序；我们调整 f1 和 f2 的位置，让 f2 排在 f1 前面，结果就会变成 undefined，严重一点会抛异常。出现这个问题的原因，在于 f2 强依赖于 `req.user`，而 `req.user` 是由 f1 赋值：调用顺序必须是 f1 在于 f2 前。既然出现了调用顺序，那跟普通的函数调用没什么两样，我们简化一下：

```
const env = { req, res, next };
const f4 = env => {
  const user = f1(env);
  f2(env, user);
};

app.use(f4);
```

这样看起来一目了然，从隐式的 `req.user` 赋值变成了函数间的组合。 f1 和 f2 都需要 env 这个参数，不仅在中间件，在所有请求处理的地方，都要方便获取 env，所以一个中间件，更符合一个 Reader: `env → IO a`。

中间件的状态

我们已经很清楚中间件就是 `Request → IO a` 的函数，r 必须携带 Request，除此之外，还有用户自定义的全局状态。所以我们可以写下这样的定义：

```
interface HttpState<S> {
  req: IncomingMessage;
  state: S;
}
```

需要特别说明一下 S，他并不是固定不变，他不仅可以是用户传入的全局状态，也可以是每次计算之后的新状态：S 表示每次计算的结果。如果还是不能理解，可以理解这个转换过程：`Reader (HttpState r) a → Reader (HttpState a) b`，从 `r` 到 `a` 的转变。

定义好全局状态，仅仅是解决了入参问题，出参格式还没着落呢。很明显，我们无法直接甩一个裸泛型参数当作返回值，因为一般的框架允许提前中断，我们无法从泛型参数知道更多信息。我们先看一段之前的代码：

```
const f = (req, res, next) => {
  const user = getUser(req);
  if (!user) {
    return res.end("not found");
  }

  req.user = user;
  next();
});
```

按照我们的设想，f 是一个普通的函数，它必定有返回值，按上面的语义，我们可以改写一下：

```
const f = (req) => {
  const user = getUser(req);
  if (!user) {
    return "not found";
  }

  return User;
});
```

这就很明显了，f 返回 `string | User`。但这种类型，我们无法辨别哪种是正确返回，哪个是错误（提前）返回。能够区别正确、错误的类型，很自然就联想到 Either，上面正好可以表示成 `Either<string, User>`。有了 Either 还不够，我们还得再定义出错误的类型，因为提前返回与出错返回是截然不同的行为。

```
class HttpError<E> {
  readonly code: number;
  readonly content: E;

  constructor(code: number, content: E) {
    this.code = code;
    this.content = content;
  }

  encodeToBody(): string {
    return JSON.stringify(this.content);
  }
}

interface ActionAbort {
  type: "abort";
  value: string;
}

interface ActionErr<E> {
  type: "err";
  err: HttpError<E>;
}

type ActionResult<E>
  = ActionAbort
  | ActionErr<E>
```

定义比 HttpState 长许多，但它要做的事只有两件： ActionAbort 用于提前返回正常响应值；ActionErr 用于处理业务错误、异常。 我们把正常返回与错误返回混在一起是个无奈这举，只因 js 没有 call/cc，而且也无法正常实现 Cont。 ActionErr 接受一个泛型参数 E，E 可以是任意值，我们无法决定用户有哪些异常，开放一个泛型参数是合理的。 ActionAbort 与 ActionErr 不同，它不是一个泛型结构，它的 value 是一个 string，表示响应体的内容。 ts 没有 typeclass 机制，无法实现通用的 toResponse 接口，反而转成 string 是个不错的折中法子。

好了，我们把这些定义填入 f，看看有什么效果：

```
const f = <S>(state: HttpState<S>): Either<ActionResult<string>, User> ⇒ {
  const user = getUser(state.req);
  if (!User) {
    return Left("not found");
  }
  return Right(user);
};
```

中间件的定义

f 的定义已经十分接近一个中间件的定义了，只需要把具体的类型换成泛型参数。

```
export interface Handler<S, E, R> extends
  Reader<HttpState<S>, EitherAsync<ActionResult<E>, R>> {
}
```

有些长，但不妨碍阅读：

```
type Handler s e r = ReaderT (HttpState s) (ExceptT (ActionResult e) IO) r
```

一个类型为 `Handler<S, E, R>`的中间件，它的入参是 `HttpState<S>`，要求返回 `EitherAsync<ActionResult<E>, R>`。既然是个 Reader，我们直接扩展原有定义：

```
interface HandlerCtx<T, E, ST = undefined> extends ReaderCtx<HttpState<T, ST>> {
  state: <K extends keyof T>(key: K) => T[K];
  send: <A>(value: A) => EitherAsync<ActionResult<E>, never>;
}

const handler = <S, E, R>(
  f: (ctx: HandlerCtx<S, E>) => EitherAsync<ActionResult<E>, R>
): Handler<S, E, R, ST> => {
  return reader((ctx: ReaderCtx<HttpState<S>>) => {
    const handlerCtx: HandlerCtx<S, E> = {
      ...ctx,
      state: key => ctx.asks(x => x.state[key]),
      send: <A>(v: A) => {
        const value: ActionResult<A> = {
          type: "abort",
          value: JSON.stringify(v)
        };
        return EitherAsync.liftEither(Left(value));
      }
    };

    return f(handlerCtx);
  });
```

HandlerCtx 在 ReaderCtx 基础上，扩展了一些方法。我们主要看 send。send 接受任意参数 A，并返回 `Either<ActionResult<E>, never>`，从签名就能看到，一旦调用 send，整个结果就会变成 `Left<ActionAbort>`，从而实现快速返回。我们继续重写 f，让它变成我们想要的样子：

```
const f: Handler<S, string, User> = handler(ctx => {
  // ask 是 ReaderCtx 的功能。
  const user = getUser(ctx.ask().req);
  if (!user) {
    // 让请求提前结束。
    return ctx.send("user not found");
  }

  return Right(user);
});
```

中间件的结合

众所周知，中间件也是个函数，既然是函数，那肯定就组合，于是我们扩展 Handler：

```
export interface Handler<S, E, R> extends
  Reader<HttpState<S>, EitherAsync<ActionResult<E>, R>> {
  bindPipe: <RA>(r: Handler<R, E, RA>) ⇒ Handler<S, E, RA>;
}
```

我们换种写法：

```
bindPipe :: Handler s e r → Handler r e ra → Handler s e ra
```

我们知道 s 是入参，r 是出参，e 可省略，我们再简化一下：`bindPipe :: (→ s r) → (r → a) → (→ s a)`。这明显就是 `flip (.)`。中间件的组合与普通函数的组合没有任何两样：

```
const bindPipe: Handler<S, E, R>["bindPipe"] = h ⇒
    handler(ctx ⇒ f(ctx).chain(nextState ⇒ {
      const state = ctx.ask();
      const nt: HttpState<R, ST> = {
        ...state,
        state: nextState,
      };

      return h.runReader(nt);
    }));
```

h 就是下一个中间件，类型为 `Handler<R, E, RA>`，调用 h 之前，我们需要得到当前中间件的返回值，即 R。为了得到 R，我们调用了 `f(ctx)`，该值就是当前中间件的返回值。

我们将之前的中间件组合一遍：

```
const f1: Handler<S, string, User> = handler(ctx ⇒ { ... });

const f2: Handler<User, stirng, void> = handler(ctx ⇒ { ... });

const f3: Handler<S, string, void> = f1.bindPipe(f3);
```

路由

我们说了很多中间件的事，如此大篇幅说中间件，是因为路由也是一种中间件。

例如我们在中间件中这样写：

```
app.use((req, res, next) ⇒ {
  if (req.url === "/abc" && req.method === "POST") {
    const r = f(req, res);
    res.end();
    return ;
  }
  next();
});
```

这样虽然没有问题，但也太过繁琐，我们势必要提供某种简便用法，方便用户定义安全的路由。

我们说路由是中件间，也就是路由也可以使用 Handler 作为定义。但是我们也要看看路由与中间件不一样的地方。 首先，路由的状态是可以根据不同定义，叠加出不同的状态的，我们举这样的路由：

```
app.post("/test1").body(schema1)
  .service(ctx ⇒ {
    const body = ctx.body; // typeof body ⩵ shcema1
    return {...};
  });

app.post("/test2").body(schema2)
  .service(ctx ⇒ {
    const body = ctx.body // typeof body ⩵ schema2
    return {...};
  });
```

两条路由各自定义 body 类型，它们的类型各不相同，都是由 `body(schema)`定义得到， 而且这两条路由的状态互相独立，不会其中一条的任何动作而改变其状态。 除了 body 之外，headers、method 等等都可以进行叠加，每进行一次定义，都会叠加上对应的状态。 `Handler<S, _, _>`的 S 已经表示计算的状态，那么我们需要额外的参数表示路由提取出来的状态：

```
interface HttpState<S, ST = undefined> {
  req: IncomingMessage;
  state: S;
  source: ST;
}

interface Handler<S, E, R, ST = undefined> extends Reader<HttpState<S, ST>,
EitherAsync<ActionResult<E>, R>> {
  bindPipe: <RA>(r: Handler<R, E, RA, ST>) ⇒ Handler<S, E, RA, ST>;
}
```

我们进一步扩展了 HttpState 和 Handler 的泛型参数，ST 表示路由累加状态， 当 ST = undefined，正好说明当前路由无任何附加状态，正好就是一个普通的中间件。 如果看到了定义 `Handler<S, E, R>`，极有可能是中间件； 如果看到了 `Handler<S, E, R, ST>`，基本上就是一条路由定义了。

我们好像为费吹灰之力就搞定了路由定义，可喜可贺。 但也只是完成了定义，路由的基本功还未实现。

路由匹配条件

路由与中间件最大不同，在于路由的匹配机制， 路由可以根据规则，选择性执行对应的逻辑， 如果未满足，可以继续往下匹配，还是直接中断退出。

我们以下面代码作说明：

```
app.get("/abc")
  .service(ctx ⇒ { ... });

app.post("/abc")
  .body(validate(schema))
  .service((ctx ⇒ {
    const path = ctx.path;
    const body = ctx.body;
    return { ... };
  });
```

假设有这么一个框架，允许定义同 path 的路由，可以根据 method 区分进入到哪个逻辑里： GET 方法我们估且称为※GET；POST 方法称为※POST。 从代码上看，路由会有一个匹配的过程，如果没有匹配到，并不会直接中断整个调用链， 匹配的过程并不会局限于上述代码中的 path、method， 自定义能力强的框架还能够允许用户写匹配规则，所以我们无法在代码中写死匹配逻辑。 假如接受到一个“POST /abc”的请求， 代码层面会先经过※GET，匹配到/abc 再去匹配 method，发现不一致，忽略当前路由，继续往下匹配； 代码到了※POST，发现在 url 和 method 都匹配，紧接着验证是否满足 schema， 未满足直接中断请求，提示 body 不合法，如果满足，进入 handler。 我们发现，即便是匹配的过程， 面对 url 和 method 的匹配，如果不满足条件，我们可以放行，进行下面路由的匹配； 面对 body 的匹配，我们一改称呼为“验证”，非法即刻中断提示，不会进行后面的匹配， 一旦验证成功，就会把这个 body 一起传入 handler。

路由匹配的定义

匹配路由，一边要做匹配条件，另一边又要把匹配到的数据传回到 handler。 我们之前往 Handler 塞了 ST 进去，ST 就是表示路由匹配出来的数据。 为了得到 ST，我们需要每次匹配时都携带这个状态。

```
interface Cond<S, ST, E, R> extends
  Reader<HttpState<S, ST>, EitherAsync<ActionResult<E>, Maybe<R>>> {
}
```

定义跟 Handler 差不多，唯一区别在于返回值：Handler 返回 R，Cond 返回 Maybe<R>。 返回 Maybe 的原因就是为了区分继续往下匹配，还是正常进入 handler。 还是以上面为例，“POST /abc”先去匹配※GET，因为 method 不满足，返回 Nothing； 继续往下匹配，发现都符合，开始验证 body，如果 body 合法，返回 Just body，如果不合法，直接中断。

我们对照它们两者的定义：

```
type Handler s st r = ReaderT (HttpState s st) (ExceptT (ActionResult e) IO) r
type Cond s st e r = ReaderT (HttpState s st) (ExceptT (ActionResult e) IO) (Maybe r)
```

一旦看懂类型含义，实现起来就简单了：

```
export const cond = <S, ST, E, R>(
  f: (ctx: CondCtx<S, ST>) ⇒ EitherAsync<ActionResult<E>, Maybe<R>>
): Cond<S, ST, E, R> ⇒ {
  return reader(f);
}
```

一句大白话。

路由匹配的组合性

跟 Handler 一样，Cond 也具备组合性：

```
interface Cond<S, ST, E, R> extends
  Reader<HttpState<S, ST>, EitherAsync<ActionResult<E>, Maybe<R>>> {
  bindPipe: <RA>(cond: Cond<S, R, E, RA>) ⇒ Cond<S, ST, E, RA>;
}
```

我们仔细看一下 bindPipe 的函数签名：

```
bindPipe :: Cond s st e r → Cond s r e ra → Cond s st e ra
-- 简化一下
bindPipe :: Cond _ st _ r → Cond _ r _ ra → Cond _ st _ ra
```

可以看到 Cond 的组合与 ST 有关，也就是说， 无论 Cond 怎么组合，它都保持着唯一全局状态（T），Cond 的组合性，也是临时状态（ST）的组合。

```
const bindPipe: Cond<S, TS, E, R>["bindPipe"] = g ⇒ {
  return cond(ctx ⇒ f(ctx).chain(x ⇒ x.caseOf({
    Just: r ⇒ {
      const state = ctx.ask();
      const rr: HttpState<S, R> = {
        ...state,
        source: r
      };
      return g.runReader(rr);
    },
    Nothing: () ⇒ EitherAsync.liftEither(Right(Nothing))
  })));
};
```

应用

单独的中间件和路由是无法直接使用的，我们需要将它们整合进来。

正如开头所言，整个应用依赖于 Reader，所以需要一个传入全局状态的入口，我们可以定义这样的函数：

```
interface HttpApplication<S> {
  fn: <E, R>(handler: Handler<S, E, R>) ⇒ HttpMiddleware<S, E, R>;
  listen: (port: number, callback: () ⇒ void) ⇒ void;
}

const application = <S>(state: S): HttpApplication<S> ⇒ {...}
```

fn 接受一个 `Handler<S, E, R>`，看到这个签名，它就是一个中间件， 然后返回了 `HttpMiddleware<S, E, R>`，看名字，应该是 Handler 的一层包装。

HttpMiddleware

`HttpMiddleware<S, E, R>`可以看成`Handler<S, E, R>`的一层包装，它持有当前的 Handler，可与下一个 Handler 进行合并，或者定义新路由。

```
interface HttpMiddleware<S, E, R> {
  fn: <RA>(handler: Handler<R, E, RA>) ⇒ HttpMiddleware<S, E, RA>;
  source: (path: string) ⇒ HttpDecl<S, { path: string }, E, R>;
  listen: (port: number, callback: () ⇒ void) ⇒ void;
}
```

生成一个 HttpMiddleware 也很简单，我把必要的参数一起传入：

```
const middleware = <S, E, R>(
  state: S,
  ha: Handler<S, E, R>
): HttpMiddleware<S, E, R> ⇒ {
  ...
}
```

这样看就更清楚了，如果 ha 直接应用 state，就能提到类型为 R 的结果。`HttpMiddlewar<S, E, R>`近似于`Handler<S, E, R>`。 fn 的签名`fn: <RA>(handler: Handler<R, E, RA>) ⇒ HttpMiddleware<S, E, RA>`十分类似于 Handler 的 bindPipe 操作，无非从 Handler 转变成了 HttpMiddleware：

```
const fn: HttpMiddleware<S, E, R>["fn"] = hb ⇒ {
  const h = ha.bindPipe(hb);
  return middleware(state, h);
};
```

HttpDecl

我们先不讲 Middleware.source 方法，先讲一下 HttpDecl。

HttpDecl 用于路由的定义，每经过一次路由定义，都会累计路由的状态，所以它需要记录下路由的状态：

```
interface HttpDecl<S, ST, E, R> {
  method: (method: string) ⇒ HttpDecl<S, ST & { method: string }, E, R>;
  body: <C>(codec: Codec<C>) ⇒ HttpDecl<S, ST & { body: C }, E, R>;
  service: <RA>(handler: Handler<R, E, RA, ST>) ⇒ HttpMiddleware<S, E, R>;
}

const httpDecl = <S, ST, E, R>(
  state: S,
  ca: Cond<S, undefined, E, ST>,
  ha: Handler<S, E, R>
): HttpDecl<S, ST, E, R> ⇒ {
  ...
}
```

跟 Middlware 一样，`HttpDecl<S, ST, E, R>`可以近似看成`Handler<S, E, R, ST>`。生成一个 HttpDecl，同样需要当前状态，及当前 Handler，还有一个路由匹配规则 ca。

method 很有意思，会生成新的 HttpDecl，它的状态变成了 `HttpDecl<S, ST & { method: string }, E, R>`。如果路由匹配到相同的 method，就会把`{ method }`传递下去，直到传到路由的 handle 里。

```
const method: HttpDecl<S, ST, E, R>["method"] = m ⇒ {
  const cb: Cond<S, ST, E, ST & { method: string }> = cond(ctx ⇒ {
    const st = ctx.ask();
    const x = Maybe.fromNullable(st.req.method)
      .map(s ⇒ s.toLowerCase())
      .filter(method ⇒ method ≡ m.toLowerCase())
      .map(method ⇒ ({
        ...st.source,
        method
      }));

    return EitherAsync.liftEither(Right(x));
  });

  const c = ca.bindPipe(cb);

  return httpDecl(state, c, ha);
};
```

我们利用了 Cond 的 bindPipe，方便实现这一功能，然后继续往下传递。

body 如法炮制，唯不同的是，如果遇到不合法的 body，需要直接中断，不能再往下走了。

```
const body: HttpDecl<S, ST, E, R>["body"] = <C>(codec: Codec<C>) => {
  const cb: Cond<S, ST, E, ST & { body: C }> = cond(ctx => {
    const st = ctx.ask();

    return EitherAsync.fromPromise(async () => {
      const bodyBuf = await new Promise<string>((resolve, reject) => {
        let bodyBuf = "";
        st.req.on("data", buf => bodyBuf += buf);
        st.req.on("end", () => resolve(bodyBuf));
        st.req.on("end", reject);
      });

      try {
        return Right(JSON.parse(bodyBuf));
      }
      catch (e) {
        return Left(mkActionInnerErr((e as Error).message));
      }
    })
      .chain(async x => {
        return codec.decode(x)
          .mapLeft(mkActionInnerErr)
      })
      .map(body => ({
        ...st.source,
        body
      }))
      .map(Just);
  });

  return httpDecl(state, ca.bindPipe(cb), ha);
};
```

body 方法接受一个 schema，待 request 的 body 收集完毕后，会进入 `codec.decode` 阶段， 如果合法，会封装成新的状态往下去；如果不合法，返回一个 Left 值，直接跳过正常逻辑，返回一个非法请求。

service 就是最终路由的定义，它会把当前收集到的所有状态传给 handler：

```
const service: HttpDecl<S, ST, E, R>["service"] = hb ⇒ {
  const h = handler<S, E, R>(ctx ⇒ {
    const st = ctx.ask();
    return ha.runReader(st).chain(a ⇒ {
      return ca.runReader(st).chain(x ⇒ x.caseOf({
        Just: r ⇒ {
          const nt: HttpState<R, ST> = {
            ...st,
            state: a,
            source: r
          };
          return hb.runReader(nt).chain(ctx.send)
        },
        Nothing: () ⇒ EitherAsync.liftEither(Right(a))
      }));
    });
  });

  return middleware(state, h);
};
```

路由 hb 需要 `HttpState<R, ST>`，R 是计算到当前的全局状态，也可能等于 S。R 由当前中间件 ha 计算得到；ST 由 ca 计算得到。有了这两个状态，hb 就能继续往下走。代码中需要判断 Maybe 值，这也就是在介绍路由匹配时，如果返回了 Nothing，表明没有匹配到，需要往下走下去，同是把 ha 的状态继续传下去；返回 Just，执行我们的 hb 路由，但同时调用 `ctx.send`，该方法直接返回结果，中断后续所有匹配。

Application

有了上面这些辅助定义，就可以很快把这些串在一起。我们再回过头看 HttpApplication，各个定义也就自然而然得出了：

```
const fn: HttpApplication<S>["fn"] = h ⇒ middleware(state, h);
```

同时的 HttpMiddlware.source 也能得出：

```
const source: HttpMiddleware<S, E, R>["source"] = path ⇒ {
  const c: Cond<S, undefined, E, { path: string }> = cond(ctx ⇒ {
    const req = ctx.prop("req");

    const x = Maybe.fromNullable(req.url)
      .filter(url ⇒ url === path)
      .map(url ⇒ ({ path: url }));

    return EitherAsync.liftEither(Right(x));
  });

  return httpDecl(state, c, ha);
};
```

source 比较直白，它是入口，所以 `ST = undefined`。一旦调用 source，就进入了 HttpDecl 定义阶段， 直到调用 service 才能再返回一个 HttpMiddleware，才能再定义其他路由。

HttpMiddlware.listen 到了最终应用阶段了：

```
const listen: HttpMiddleware<S, E, R>["listen"] = (port, callback) => {
  const srv = createServer(async (req, res) => {
    const httpState: HttpState<S> = {
      req,
      state,
      source: undefined
    };

    const result = await ha.runReader(httpState);
    const [content, code] = result.caseOf({
      Right: a => [JSON.stringify(a), 200],
      Left: e => {
        if (e.type === "abort") {
          return [e.value, 200];
        }
        if (e.type === "err") {
          return [e.err.encodeToBody(), e.err.code];
        }
        if (e.type === "innerErr") {
          return [e.err, e.code];
        }
        const _: never = e;
        return _;
      }
    });
    res.setHeader("Content-Type", "application/json");
    res.write(content);
    res.statusCode = code;
    res.end();
  });

  srv.listen(port, undefined, undefined, callback);
};
```

它只依赖于 state 和 ha，ha 就是最终整合到一起的 Handler， 不管是单独定义的 Handler，还是经由 HttpDecl 定义出来的 Handler， 所以的路由经过一系列的变形、组合，最终到达这里，一个 ha 就是组合不同中间件、路由的巨大函数。 同时也要在这里把所有异常处理了，不能让程序出现意外，导致意外退出。 在这里，可以看到 HttpState 的生成、ha 的应用、错误处理，可以说，这是整个 web 框架处理请求、响应结果的地方。 这里最关键的，就是 ha 如何得来，state 及错误没有调整的空间，ha 却可以根据用户自由组合出来。

一个示例

我们看一个简单例子：

```
const fn1: Handler<{ nameCount: number }, never, number> = handler(ctx ⇒ {
  const state = ctx.prop("state");
  return EitherAsync.liftEither(Right(state.nameCount + 1))
});

const fn2: Handler<number, never, { name: string, value: string }> = handler(ctx ⇒ {
  console.log("do this?");
  const value = ctx.prop("state");
  const nextState = {
    name: "hello",
    value: JSON.stringify(value)
  };
  return EitherAsync.liftEither(Right(nextState));
});

const fn3 = fn1.bindPipe(fn2);

application({ nameCount: 1 })
  .fn(fn3)
  .source("/abc").method("get").service(handler(_ ⇒ {
    return EitherAsync.liftEither(Right({
      a: 1,
      b: ["1", { a: 1 }]
    }))
  }))
  .source("/abc").method("post").body(CC.Codec.interface({ name: CC.string, code:
CC.number })).service(handler(ctx ⇒ {
    const body = ctx.source("body");
    return ctx.send(body);
  }))
  .fn(handler(_ ⇒ EitherAsync.liftEither(Right("not found"))))
  .listen(3000, () ⇒ console.log("start!"));
```

f1 和 f2 单独定义，f1 的入参是`{ nameCount: number }`，正好是 application 的入参； f2 的入参是 f2 的出参，满足 bindPipe 条件，所以 `f3 = f1.bindPipe(f2)`。

source 定义了两条路由，一个是 Get /abc，另一个 Post /abc； Get /abc 并没有声明 body，所以它无法使用 ctx.source(“body”)得到 body，编译期会直接报错； Post /abc 相反，可以得到相对应的 body。

每一层定义环环相扣，如果调整顺序，就会出现类型错误，这就是由类型检测带来的安全性。

不足之处

我们已经完成整体框架，使用也符合预期。 示例中，我们通过链式表达，定义出每个路由，这种写法对于小型应用还好，规模一大就会绞在一起。 还是群友形容得贴切：像一大串烤串。

假设我们编写出了另一种形式：

```
application(anything)
  .source(source("/abc").method("get").service({}))
  .source(source("/abc").method("post").service({}))
```

内部 source 可以像 handler 单独定义，这就能做到解耦。 不幸的是，这样的写法，导致类型出现问题： 我们要求 Handler<S, E, R>的泛型参数能够在上下文推断出来， 然后在此处，source 独立定义，所有泛型只能推断出 unknown， 导致要求 Handler<S, E, R>的地方接受 Handler<undefined, undefined, undefined>， 不满足要求编译错误。可以说这是 ts 上下文推导能力不足导致的。

另一个不足在于没会根据 url 的 prefix 分层，简单来说无法做到：

```
const tag = app.get("/");
const post = app.get("/");

app.ues("/tag", tag);
app.use("/post", post);
```

这是框架层面的不足。

总结

我们通过不断抽象，抽象出了中间件、路由共用表达式：Handler。 又利用这个抽象，进而再次抽象出匹配的规则 Cond。 不管它们的表现形式如果，归根结底它们都是 Reader 的一种变体，也就是一种函数的另一种形式。 函数具体组合性质，我们又将 Handler 与 Handler、Cond 与 Cond、Handler 与 Cond 结合起来， 得到一个更加复杂的 Handler……组合可以无限进行，直到满足用户为止。 每一次组合，都影响着整个框架的状态，影响着每次调用 handler 的类型限制。 由于类型的层层推进，才保证的 handler 的合法性，这样才能保证框架的安全。

最后的最后，虽然 ts 的类型体操可以制造出一个类型安全的框架， 但编译器本身的限制，做出来的框架使用体验不佳。

不如直接使用现有框架，打上一些补丁，也能具备一定的安全性。 就像纯函数式之于 IO。