♫ TypeScript 上的 Reader ♫

荀洪道

二〇二四年一〇月二九日

背后的动机

“这个世界已经被 Java 统治了。”

身为 node 码农，眼看 node 已有被 java 严重渗透趋势：仿佛不用 java 那套笨重的方法便解决不了问题。 java 的注解，迁移到 typescript 就成了装饰器，本质上还是比较接近，都是为了实现依赖注入， 然而它们的实现都依赖运行时注入，无法在编译期完成检查；.Net 也有同样问题。

typescript 从类型角度上讲，还是有可取之处，尤其 structural type，可以做一些简单的类型合并(&)，可以说，ts 也有了个简易版的 dependent type；虽然造出不通用的 Monad，但特化的某一 Monad 还是没问题的。

基于 typescript 编译期的计算、检查，就可以造出类型安全的依赖注入。

阅读要求

1. 懂一点 Haskell，至少能看懂函数签名。
2. 懂 typescript 类型计算，不需要复杂类型体操，能懂基本操作即可。

Reader

reader 是什么？

这个问题很好回答，reader 就是一个普通的函数，同时，它正是一个依赖注入。 在 haskell，一条函数可以用`(→ r a)`表示，或者写成`(r → a)`，`r` 正是 reader，也可以需要注入的环境、变量。 乍一看没什么特别的，实际上`(→ r)`构成了一个 Monad，于是我们可以使用 `bind` 连接多个 reader。

```haskell
inc :: Int → Int
inc = (+ 1)

add10 :: Int → Int
add10 = (+ 10)

f :: Int → String
f = do
    a ← inc
    b ← add10
    pure $ show a ++ " " ++ show b

main :: IO ()
main = do
    print $ f 1     -- 2 11
    print $ f 11  -- 12 21
```

我们重点看 `f`，它的引用了 `inc` 和 `add10`，同时将这两个函数的返回值拼接成 String： `f` 接受参数 1 时，它调用 `inc` 时也将 1 传了过去，`add10` 同理， 所以可以得到 a 和 b 的值分别为 2 和 11，最后拼成 String。 我们并没有显式传值，但 `inc` 和 `add10` 依然能知道当前上下文—— 这个上下文就是我们所说的依赖—— 这是因为 Monad 的 do 记法隐藏了传值过程。

这也是叫 Reader 的原因，因为只是读取环境、变量，不会去改动环境、变量的值。

reader 在 typescript 中的表示

reader 就是一个普通函数，这在 ts 很容易，我们同样可以写上 `inc` 和 `add10`。

```
const inc = (n: number): number ⇒ n + 1;
const add10 = (n: number): number ⇒ n + 10;

const f = (n: number): string ⇒ ?;
```

到这 f 这里，我们好像写不出来 Haskell 那股味道了：inc 和 add10 必须显示传值。 我们好像没有办法做到隐式传值了？

函数的另一种形式

我们先扯一些别的话题：说到函数，你会想到什么？

大家的直觉就是 function 定义出来的就是函数。

那么再换个角度：一个可以调用的对象，可以是函数吗？

如果说一个对象可调用，我们为一个对象实现 Callable，是不是这个对象就成了函数？这样一讲，是不是很像 CPP 的 Functor（跟 Haskell 的 Functor 不是一个东西）。

```
Functor f = new Functor(10);
f();

Functor g = new Functor();
g(10);
```

f 和 g 都可视为函数的用法，唯一区别在于入参的位置：f 在对象创建时入参就确定，g 需要到调用时才确定。 基于依赖注入的要求，f 更符合传递依赖的要求，因为 f 在创建时就获取了环境，那么内部可以任意处理这个状态了， 不用担心这个环境何时传入、哪里传入。那么我们就可以写出一个普通的 Functor（Reader）了。

```
abstract class Reader<T, R> {
    protected readonly env: T;

    constructor(env: T) {
        this.env = env;
    }

    abstract run(): R;
}

class Inc extends Reader<number, number> {
    run(): number {
        return this.env + 1;
    }
}

const inc = new Inc(10);
inc.run(); // 11
```

看起来很不错，Inc 只要继承自 Reader，相当于隐式获取了 number（T = number）这个 env， 用 run 得到这个函数的返回值 number（R = number）。

如果一个地方返回了 Inc 对象，我们能否说返回了一个高阶函数回来呢？

对象的缺点

说到大的缺点也是没有的，既然对象能变成函数，那么该有的作用，它也是有的，唯一问题没有高阶对象， 我们再考虑一种情况：

```
const f = (n: number): ? => {
    return class Int extends Reader<number, number> {
        run(): number {
            return this.env + n;
        }
    }
};
```

f 的返回值无法确定了，它需要先在函数内创建 Int 才能知道具体类型，而返回需要预先知道， 所以 f 这个函数类型写不来。 而且本身也会报错，env 在匿名类中，不会再变成 protected，可能会被任意篡改。

闭包表示法

当我们抛弃了对象，那么我们只能继续使用函数本身。 上面说到单纯的函数，是无法做到隐式传参的，所以我们可以往函数上加一层：

```
interface Reader<T, R> {
    runReader: (env: Readonly<T>) => R;
}
```

构造成这个结构， 一方面不失原来函数本身的作用，调用 runReader 依然可以完成函数调用， 另一方面，可以方便日后扩展。

数据结构定下，那么我们就可以将普通的函数转化成 Reader：

```
const reader = <T, R>(f: (env: Readonly<T>) => R): Reader<T, R> => {
    const runReader: Reader<T, R>["runReader"] = env => f(env);

    return {
        runReader
    }
};

const f: Reader<number, number> = reader(n => n + 1);

f.runReader(1); // 2
```

实现完整的 Reader

Reader 的实现

我们已经构造了最简单的 Reader，接下去需要完成 Reader 间的引用—— 如果仅仅是普通引用，这跟普通函数调用没有任何区别—— 这就需要一个中间层，能记录当前 env，同时能自动调用其它 Reader。对于这个中间层，我们尚且命名为 ReaderCtx，因为保留了当前 env，所以它需要保留下 T 这个入参类型。

```
interface ReaderCtx<T> {
    ask: () ⇒ Readonly<T>;
    bindFrom: <R>(r: Reader<T, R>) ⇒ R;
}
```

ask 用于获取当前 env，上面的 `reader(n ⇒ n + 1)`的 n，即我们所说的 env，此时要用 ask 获取得到； bindFrom 调用其他 Reader，从签名可以知道，r 的入参要跟原先一致，但返回值可以是任意的， 这里的 `R` 与 `Reader<T, R>`不是同一个 `R`。 那么原先的 `reader` 也需要改：

```
const reader = <T, R>(f: (ctx: ReaderCtx<T>) ⇒ R): Reader<T, R> ⇒ {
    const runReader: Reader<T, R>["runReader"] = env ⇒ {
        const ctx: ReaderCtx<T> = {
            ask: () ⇒ env,
            bindFrom: r ⇒ r.runReader(env)
        };
        return f(ctx);
    }

    return {
        runReader
    }
};
```

`reader` 的 f 从 `T ⇒ R` 变成了 `ReaderCtx<T> ⇒ R`， 可以认为 ReaderCtx 是 T 的一个 wrapper，同时提供更多的便捷操作。 我们回头看 bindFrom 实现：`r ⇒ r.runReader(env)`。 很简单的一句话，就是将当前的 env 传给另一个 Reader，并得到结果。 有了这个“暗箱操作”，我们就可以写出这样的代码：

```
const f: Reader<number, number> = reader(ctx ⇒ ctx.ask() + 1);

const g: Reader<number, number> = reader(ctx ⇒
    ctx.bindFrom(f) + ctx.bindFrom(f));

g.runReader(10); // 22
```

g 中直接调用 f，因 ReactCtx.bindFrom 会自动传参给 f，所以可以直接省却。

Reader 的组合

到目前为止，已经完成了我们心中的 Reader Monad：不仅能当成普通函数使用，还能相互间调用。但我们还是可以继续前进一步，完成 Reader 的 compose 操作。 这时 Reader 留下的 interface 就有可扩展空间，我们改一下 Reader 结构：

```
interface Reader<T, R> {
    runReader: (env: Readonly<T>) ⇒ R;
    pipe: <RA>(r: Reader<R, RA>) ⇒ Reader<T, RA>;
}
```

大家都很了解函数间的组合： 已知 F：`A ⇒ B`，G：`B ⇒ C`，那么 `F ▷ G` 即为 `A ⇒ C`。

同理可得知道，`Reader<T, R>`组合 `Reader<R, RA>`，结果自然也是 `Reader<T, RA>`， 这也是 pipe 签名的含义。 我们就要去实现它：

```
const pipe: Reader<T, R>["pipe"] = r => {
    return reader(ctx => {
        const envR = f(ctx);
        return r.runReader(envR);
    })
};
```

因为需要重新组合成新的 Reader，所以调用 `reader` 方便生成新 Reader。 有了这个组合方法，我们又可以操作一番：

```
const f: Reader<number, number> = reader(ctx => ctx.ask() + 1);

const g: Reader<number, number> = reader(ctx => {
    const add3 = f.pipe(f).pipe(f);
    const add5 = f.pipe(f).pipe(f).pipe(f).pipe(f);

    return ctx.bindFrom(add3) + ctx.bindFrom(add5);
});

g.runReader(10); // 28
```

上面的代码已经不用再说明了，不言自明。

扩展 Reader

经过我们一番抽象行为，这个 Reader 基本能满足大部分场景， 但我们还是要有自定义 Reader 的，第三方的 Reader 永远是第三方，自己定义出来的 Reader 才是自己的。

假设我们需要 zip 两个 Reader，在不改变 Reader 的基础上，我们可以扩展出自己的 interface：

```
interface ZipReader<T, R> extends Reader<T, R> {
    zip: <RA>(r: ZipReader<T, RA>) => ZipReader<T, [R, RA]>;
};
```

这段很容易理解，`A zip B = [A, B]`。我们接下去扩展原来的 reader：

```
const zipReader = <T, R>(f: (ctx: ReaderCtx<T>) => R): ZipReader<T, R> => {
    const theReader = reader(f);

    const zip: ZipReader<T, R>["zip"] = r => {
        return zipReader(ctx => {
            const envA = f(ctx);
            const envB = r.runReader(ctx.ask());
            return [envA, envB];
        });
    };

    return {
        ...theReader,
        zip
    }
};

const f: ZipReader<number, number> = zipReader(ctx => ctx.ask() + 1);
const g: ZipReader<number, number> = zipReader(ctx => {
    const k = f.zip(f);
    const [a, b] = ctx.bindFrom(k);
    return a + b;
});

g.runReader(1); // 4
```

我们依然调用 reader 生成最原始的 Reader，之后额外定义一个 zip，再与 `theReader` 合并，这样就得到我们新的 ZipReader。

因为我们抛弃了对象，也舍弃了 OOP 语法上的继承，只能通过 JS 对象上的合并达到继承。但好在也是能实现，只是样板代码要多一些。

结尾

经过我们一番艰苦卓越的操作，终于实现了 typescript 版的 Reader，虽然上面都在讲如何实现，实际上它已经能做到完整的类型检查。

正如开篇所讲，一个 Reader 其实就一个普通函数，Reader 能做的，与一个函数没有区别：正因如此，单独看 Reader 反而不如函数实用。我们在开篇时也提到了依赖注入，Reader 能实现依赖注入的前提，就是能显示传递依赖到下层函数，经过我们改版后 Reader 是能做到的，所以从实用角度上讲，Reader 比普通函数更适合用作依赖注查。