Monad 视角下的 Promise

荀涧林

二〇二二年九月十一日

1 背景

随着 ES Promise 进入到标准库，人们越来越接受 Promise 作为异步的处理方式，加之 await、async 语法上的支持，Promise 写法上更加便利，不再出现以往的回调地狱，代码可读性进一步提高。Promise 除了表示一种数据结构，它还是有一种标准，称之为 Promise/A+，总体而言，Promise 是一种良好的抽象，我们可以从它身上实现更多有意义的设计模式。这种良好的抽象不单单是指对异步的抽象，Promise/A+ 并未定义一定只能处理异步任务，只要能满足标准的都可称之为 Promise，人们甚至讨论过 Promise 是不是一种 Monad [1]，当然，这不是我们讨论的重点，我们以 Promise 为切入点，利用它的某些性质，解决我们一些日常问题。

2 Promise Interface

在 typescript 语境中，Promise 有两种含义：

```
const a: Promise<number> = Promise.resolve(1);
```

代码出现了两个 Promise，前者是一个接口，后者就是具体的类型（或者对象），作为一个 Promise 对象，它自然满足所有 Promise 接口实现。Promise 还有一个标致性方法——then——等待任务完成，说实话，看到 then，很难不把它跟 Monad 的 bind 联系起来——都是表示计算顺序：

```
const a = 1;
const b = await Promise.resolve(a + 1)
return a + b; // Promise 3
```

```
let a = 1
b <- Just (a + 1)
return (a + b) -- Maybe 3
```

Promise 还能自动捕获异常，从这里抛出的异常，能够在 catch 中捕获。异常是非常自然的代码跳转语句，能够让我们从深层代码段中立即跳出，利用这一有用特性，能让我们实现一些便利的功能。

3 Maybe Promise

试想一下，如果 js 不再有空值指针，即 undefined、null 从语言中删除，那么我们怎么表示有和无呢？这个问题说来也简单不过，定义两个结构体，一个表示有，一个表示无，使用的时候判断一下就行了，跟直接使

用空指针差不多。

```
interface Some<T> {
  _tag: "some";
  _value: T;
}

interface None {
  _tag: 'none';
}

type Option<T> = Some<T> | None;
```

typescript 没有 ADT，我们只能依靠 _tag 区分 Some 和 None；Some 表示有值，None 相反。市面上的语言——Java、C++——都会有这样的数据类型提供使用，但一般是以库的形式提供，默认不启用，历史的原因，现在代码并不能完全抛舍空指针，C++11 甚至提出了 nullptr 关键字。

Haskell 的 Maybe 就是表示 Option，两者仅仅名称不同，实际作用完全一致。回到现实，从 js 的角度出发，我们依然需要处理空值，这是避免不了的；我们设计接口时，允许传递空值，一旦包装过 Maybe，我们希望之后不再对空值作显示处理，而且也不希望暴露出来——忘掉空值这回事。

```
interface Maybe<T> {
    map: <R>(f: (value: T) => R) => Maybe<R>;
}

const maybe = <T>(ma: T | null): Maybe<T> => {
    const map = <R>(f: (value: T) => R): Maybe<R> => {
        if (ma === null) {
            return maybe<R>(null);
        }
        else {
            return maybe(f(ma));
        }
    };

    return {
        map
    };
};
```

这个 maybe 就是我们的值构造器；map 函数能自动处理空值，参数 `f: (value: T) => R` 也不关心空值，一般而言，整个 Maybe 是对空值的抽象，我们不想显式处理空值时，用它是最适合不过。

```
const ma = maybe(1); // Maybe(1)
const ma_ = ma.map(x => x + 1); // Maybe(2)

const mb = maybe<number>(null); // Maybe(null)
const mb_ = mb.map(x => x + 1); // Maybe(null)
```

3.1 解构

正常情况我们不能看到 Maybe 里面的值，`console.log(ma)` 最后也只能得到一堆 Maybe 的属性。这是因为我们都是通过 maybe 构造的，而 maybe 这个又是个高阶函数，它的内部值就是它的参数，它最后也仅仅返回一堆函数：

```
const maybe = (a) => { // a不仅是输入的参数，它也是整个Maybe的内部状态
  // ...
  return {
    map
  };
};
```

为了拿出 a，我们可能需要多作一些准备：回想一下，我们说过要忘记空值，而此刻的 a 有可能是个空值，我们不能直接将它拿它出来使用——这样就违反了之前的约定。即然 Maybe 表示“有”与“无”，那我们只要构造出两个相关的解构函数就行了：

```
interface Maybe<T> {
    unwrap: <R>(cond: { Just: (value: T) => R, Nothing: () => R }) => R;
}

const maybe = <T>(a: T | null): Maybe<T> => {
    const unwrap = <R>(cond: { Just: (value: T) => R, Nothing: () => R }): R => {
        if (a === null) {
            return cond.Nothing();
        }
        else {
            return cond.Just(a);
        }
    };
};
```

我们用 cond 代表 Maybe 的模式匹配，这种匹配是严格匹配，Just 和 Nothing 都需要求严格求值：

```
const add1 = (x: number): number => x + 1;
const ma = maybe<number>(10);
const result = ma.map(add1);
result.unwrap({
    Just: x => console.log(`Get ${x}`),
    Nothing: () => console.log(`Get Nothing`)
});
```

至此，我们解决了 Maybe 打包、解包问题。接下去就要看看如何实现个 Maybe Promise。

3.2 Maybe Promise

then 跟 (»=) 十分相像，Maybe Promise 既然是 Maybe Monad 翻版，那么 then 的实现跟 (»=) 必须比较接近，它们都能自动处理 Nothing，遇到 Nothing 能够自动返回，不需手动干预。Haskell 的 ghc 能够自动优化这一过程，在 typescript 中，我们只能自己实现。为了能实现自动跳转，或者说提前中断，除了 return、

break……用来处理异常的 throw 也是十分好用的跳转命令，相较于前两者，throw 能直接向最外层跳转，至到被捕获或完全抛出。这是一个十分有用的特性，除了处理异常，我们能够实现提前跳转，一遇到 Nothing 提前结束后面的流程，直接返回 Nothing，实现 (»=) 类似的功能。

typescript 除了提供 Promise 接口，不同的版本，Promise 接口实现还不一样，es5 中只要实现 then、catch，其他版可能还需要多实现 finally，我们这里一切从简，只拿 then 和 catch。实际上，除的 Promise 接口，还有一个 PromiseLike 接口，它只有一个 then 实现，但我们不能用它，因为我们需要控制异常，这种版本的 Promise 无力处理异常，我们不能任由异常到处外抛。Promise 几个方法签名都比较复杂，实际上我们也并不需要重新实现一个 Promise，我们只是想利用 then 这个接口，从而能够直接使用 `await`；或者实现 catch 接口，达到控制异常目的。种种限制，都需求我们重新建构一个 Promise。Promise 本身也是带有异常状态，跟我们上面异常处理一拍即合。

```
interface Maybe<T> extends Promise<T> {
    // ...
}

class NilError {};

const maybe = <T>(a: T | null): Maybe<T> => {
    const then: PromiseLike<T>["then"] = (resolve, reject) => {
        return Promise.resolve(a)
            .then(a => {
                if (a === null) {
                    throw new NilError;
                }
                return a;
            })
            .then(resolve, reject);
    };

    const _catch: Promise<T>["catch"] = (f) => {
        return Promise.resolve(a)
            .then(a => {
                if (a === null) {
                    throw new NilError;
                }
                return a;
            })
            .catch(f);
    };
};
```

then 和 catch 实现非常接近，不管调哪个接口，都会构造新的 Promise，遇到 Nothing 即刻抛出 NilError。这种方式显而易见地容易，缺点就是从同步代码变成了异步。

```
await maybe(1) + await maybe(2) // maybe(3)

await maybe(1) + await maybe<number>(null) // NilError
```

除了第二条语句抛异常外，其他都挺好的。为了兜住这个异常，我们需要一个 Wrap：

```
interface AsyncMaybe<T> {
    unwrap: <R>(cond: { Just: (value: T) => R, Nothing: () => R }) => Promise<R>;
}

const asyncMaybe = <T>(f: () => Promise<T>): AsyncMaybe<T> => {
    const unwrap = <R>(cond: { Just: (value: T) => R, Nothing: () => R }): Promise<R> => {
        return f().then((x: T) => cond.Just(x))
            .catch(e => {
                if (e instanceof NilError) {
                    return cond.Nothing();
                }
                else {
                    throw e;
                }
            });
    };

    return {
        unwrap
    }
}
```

我们做了一个类似于 Maybe 的 AsyncMaybe，但它只有一个 unwrap 方法。由于 Maybe 变成 Promise，异致个结构都变成异步，所以 unwrap 最后的返回必然是 Promise。有了这个 wrap，我们就能自由写如下代码：

```
const ma = maybe<number>(10);

asyncMaybe<number>(async () => {
    const r = await ma + await ma;
    return r;
}).unwrap({
    Just: x => console.log(`Get ${x}`),
    Nothing: () => console.log("Nothing")
});
```

3.3 Maybe 小结

我们先是用 Maybe 作为空值的抽象结构，利用它尽力避免直接处理空值，我们使用 unwrap 分别处理“有”“无”。接下去为了向 Maybe Monad 靠近，利用 Promise 接口，实现了 await 接口，最表现的结果就是能直接用 await 对 Maybe 进行取值；为了实现空值提前跳转，我们又定义了 NilError 异常状态，在 Promise 实现中，遇到空值自动抛出该异常；为了避免 NilError 四处散播，我们又创建了 AsyncMaybe，专门用于处理这个异常。到目前为止，一切看起来都比较正常。

这里又暴露出一些问题：最重要的就是 Promise 是异步抽象，而 Monad 仅表示计算顺序，我们一步步抽象下来，最后不得不变成了 Promise；其次 Promise 是 js 时代的产物，它有灵活的一面，但在此处却变得极难处理，例如我们构造不出来 `Promise<Maybe<T>>` 这样的数据，最后 asyncMaybe 只能接受 Promise<T>。

这仅是一次尝试，可以看到 typescript 自身十分依赖 javascript 实现，typescript 作为类型标注，在多态实现上十分无力。

4 实现顺序的 web server

node 提供了 http 模块用于创建 web 服务，下面是最简单的例子：

```
import { createServer } from "http";

const server = createServer((req, res) => {
    console.log(req.url);
    res.end("hello world");
});

server.listen(3000);
```

用户不论输入什么地址，都只能看到“hello world”，为了区分不同路由，我们可以往里面加一些 if-else。

```
import { URL } from "url";

const server = createServer((req, res) => {
    const url = new URL(req.url ?? "", "http://localhost:3000");

    if (url.pathname === "/") {
        return res.end("hello world");
    }

    if (url.pathname === "/a") {
        return res.end("hello a");
    }

    res.end("not found");
});
```

4.1 MVC

目前来看，这个结构已经十分“顺序结构”了。但是一个普通的 web 服务都有个 MVC 结构，随着业务代码增加，这里的代码会急剧膨胀，所以最好能实现一个 MVC 结构出来。我们一般做法都是一条 url 绑定一个 Handler，一个网站必然有多条 url 绑定，用户无论输入什么网址，我们都会一条条寻找下去，直到找到为止，又或者直接抛出未找到信息。假如我们已经实现出来，那么会是这样光景：

```
const server = createServer(async (api, res) => {
    const r1 = await api.get("/", async req => {
        return "hello world";
    });

    const r2 = await api.get("/a", async req => {
        return `hello ${req.url}`;
    });
```

```
    res.end(r1 + r2);
});
```

当然，这上面是伪代码，无法运行。我们构造一个 api 的上下文，它用于绑定 url 与 handler，如果匹配到，那么执行这个 handler，需要注意的是，此时匹配过程并未中断，而是会往下继续执行。所以需要在 handler 里返回一个值，而不是立即调用 res.end 直接返回 http。假使 r2 也绑定到“/”，那么这两个路由都会匹配到，结果就会变成 `"hello world" + "hello ${req.url}"`；如果直接调用 res.end，它会执行两遍，这是不允许的行为，会内部异常。

4.2 API

我们已经知道了 api 使用方法，接下去就要去实现它。所明显，我们需要对 `(req, res)` 这对参数作一次包装，同时也要记录下当前访问 URL 对象，方便日后路由匹配。

```
import { createServer, IncomingMessage, Server } from "http";
import { URL } from "url";

interface Api {
    req: IncomingMessage;
    url: URL;
    get: (
        path: string,
        handler: (req: IncomingMessage) => Promise<string>
    ) => Promise<string>;
};

const useServer = (f: (api: Api) => Promise<string>): Server => {
    return createServer(async (req, res) => {
        const url = new URL(req.url ?? "", "http://localhost");
        const api: Api = {
            req,
            url,
            get: (path, handler) => {
                if (path === url.pathname) {
                    return handler(req);
                }
                else {
                    return Promise.resolve("");
                }
            }
        };

        const r = await f(api);
        res.end(r);
    });
};
```

Api 中的 get 就是 url 与 handler 的绑定，从实现上可以看出，如果匹配到该路由，那么会调用 handler，未匹配，则返回空字符串；useServer 要求必须返回 string，保证了无论是否匹配到路由，都能有个值可以用

于返回到用户。

useServer 本身没多大问题，但每写一个路由都要返回一个值，然后再集合起来返回，数量少还好，数量一多，这样的代码是不能维护的。我们必须想个办法，能够直接在 handler 中直接发送 http 响应。

4.3 Send

现在我们要实现能直接返回响应体的函数 send，上面提到了 res.end 不能调用多次，如果我们有这样的代码：

```
await api.get("/", _ => res.end("get /"));

res.end("not found");
```

访问首页会出问题：第一次会匹配到"/"，执行 `res.end("get /")`，但此时程序还是会往下执行，执行 `res.end("not found")` 出现异常，出现了两次调用。那么我们可以将 send 作为业务的最后一步，一旦调用该方法，最后只作返回响应，其他一概不管。为了能快速跳出，又得请上我们的老朋友——异常——了！

```
import { createServer, IncomingMessage, Server } from "http";
import { URL } from "url";

interface Api {
    req: IncomingMessage;
    url: URL;
    get: (
        path: string,
        handler: (req: IncomingMessage) => Promise<void>
    ) => Promise<void>;
    send: (msg: string) => void;
};

class Finish {}

const useServer = (f: (api: Api) => Promise<void>): Server => {
    return createServer(async (req, res) => {
        const url = new URL(req.url ?? "", "http://localhost");
        const api: Api = {
            req,
            url,
            get: async (path, handler) => {
                if (path === url.pathname) {
                    return handler(req);
                }
            },
            send: msg => {
                res.end(msg);
                throw new Finish;
            }
        };

        return f(api).catch(e => {
            if (e instanceof Finish) {
```

```
                return res.end("");
            }
            throw e;
        })
    });
};
```

新添了 send 方法，实现十分简单，res.end 之后直接抛出 Finsih，之后兜住它就行了。get 不再返回 Promise<string>，因为随时可以中断，我们对返回结果已经不关心了，只要 Promise<void> 就行。

最后使用起来会是这个样子：

```
const server = useServer(async api => {
  await api.get("/", async req => {
      api.send("hello world");
  });

  await api.get("/a", async req => {
      api.send("hello a");
  });

  api.send("not found");
});
```

4.4 小结

至此我们实现一个 Promise 形式的 web 服务，跟其他 web 服务不同在于，它不保存整个路由树，而是像同步代码一样，一条一条往下执行，直到遇到正确的路由。我们可以像普通逻辑一样写出 web，例如其他框架中的中间件，在这里仅仅是其中的一块代码而已：

```
await api.get("/", async req => {
  api.send("hello world");
});

const user: User | null = await api.getUser() // 假论已实现
if (user === null) {
  api.send("user not found");
}

// 以下都是用户已登录状态
await api.get("/user/info", _ => api.send(user.id));
```

api.send 带有中断功能，不用担心会往下执行。我们用普通方式表示出了其他框架的中间件功能，而且具有一定的类型安全，不会因为代码位置的不同而导致不同的状况。

5 总结

我们实现了 Maybe Promise 和 Handler Promise，它们都具有一定的相似功能，同时为了表达中断功能，都需要用到异常。使用异常一点问题没有，但我们不能管杀不管埋，我们需要对我们定义的异常作兜底处理，

不能任由它四处散播，如果让用户看到这个异常，就说明我们的封装是失败的，用户不该关心内部实现。

同时我们也该看到 Promise 一些缺陷，Promise 过于动态特性，导致无法构造出嵌套 Promise，例如我们无法构造出 `Maybe<Maybe<T>>` 。目前我们只实现了 Maybe Promise，Either Promise 自然也是不在话下。但 Promise 与 Monad 始终是不同的：Promise 的 then 和 Monad 的 (»=) 虽然都可以接受一个函数，但 then 是一次性回调，一个 then 里的函数一般最多调用一次，(»=) 却不然。List Monad 就是最典型的例子，某种程度上，(>>=) = $flatMap$，但 Promies.then 却做不到。像其他的 Reader Monad、Cont Monad 此类种种，Promise 一概无法实现。

Promise 性质虽然靠近 Monad，但却不是同样的抽象接口；Promise 是对异步抽象；Monad 则是更通用的抽象抽口，无论同步还是异步——例如同步的 Maybe Monad、异步的 TVar Monad——都能胜任。总而言之，Monad 在函数式编程中重要的一环，是因为它普适性，一律顺度的计算最后都需要实现出一种 Monad，例如 IO Monad。但在 typescript 中却不需要如此，因为它本身就是严格求值，求值顺序由代码顺序决定。

参考文献

[1] Zaitsev, D. Answer to "Why are Promises Monads?", May 2018.