

构建在 Lens 之上的状态管理

荀润林

二〇二三年九月二十五日

1 背景

编写 `drifloon` 并没考虑到状态管理，内部仅提供了用于表达可变的 `IORef`，随着项目增大，渐渐发现处处可变，状态管理随之失控，样板代码充斥其间，十分不美观。`IORef` 跟 `let` 定出来的变量无任何不同，导致更新局部状态十分棘手：说我们有一个大状态 `State<A>`，`A` 表示我们要更新的部分，一个组件刚好只需要 `A`，于是我们传入 `IORef<A>` 或 `A`，组件内的更新都无法影响到 `State`，导致每次更新都要手动翻新整个 `State`。

由于上述原因，我们做不到双向绑定。此刻亟须实现一个统一、自动更新的状态管理。

2 选择 Lens

JS 传统，实现“自响应”的状态，大多会考虑 `Proxy`，但它的可扩展性不强——我们不仅要处理原始类型，还要能处理用户自定义类型——无法提供一个接口用于扩展；JS 还有响应式的 `rxjs`，`mithril` 也提供简易版的响应式，但响应式自有它的复杂性，同时它不是常规的状态管理，贸然使用徒增难度。

基本上否定了 JS 社区方案，于是目光转向了 Haskell。最初的考虑是 `profunctors` 或 `contravariant`，因为从这几个类型签名就感觉很满足条件：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Contravariant where
  contramap :: (a -> b) -> f b -> f a

class Profunctor f where
  dimap :: (a -> b) -> (c -> d) -> f b c -> f a d
```

但这些都太复杂了，回到最初，我们仅需要两个基本方法：

```
get :: s -> a
set :: s -> a -> s
```

不管多深的结构，我们仅仅组合以上两个函数就能得到所有数据。

例如我们有这样的结构体:

```
interface B {
  c: C
}

interface A {
  b: B
}
```

同时我们已经实现以下函数:

```
getB :: A -> B
setB :: A -> B -> A

getC :: B -> C
setC :: B -> C -> B
```

我们要实现 $A \rightarrow C$ 这样的函数, 只要组合 `getB` 和 `getC` 即可得到: $(A \rightarrow C) = (B \rightarrow C).(A \rightarrow B)$ 。是的, 就此, 我们发现了 Lens 的定义:

```
data Lens s t = Lens { get :: s -> t
                      , set :: s -> t -> s
                      }
```

3 Lens 是什么?

Lens 到底是什么?

一句话可概括:

Costate Comonad Coalgebra is equivalent of Java's member variable update technology for Haskell

Lens 就是为了解决不可变数据更新的问题, 我们从 JS 考虑这一问题: 我们依然有 State 这样的数据, 想要更新它比较深位置的数据, 该怎么做呢?

```
interface State {
  a: A;
}

interface A {
  b: B;
}

interface B: {
  c: C
}
```

为了维持不可变性, 我们需要每一层每一层、层层、一路更新上来:

```
const state: State;
const state = {
  ...state,
  a: {
    ...state.a.b,
    b: {
      ...state.a.b.c,
      c: c
    }
  }
}
```

```

}
};

```

这样的手工作业，效率极低，代码枯燥，容易出错；Lens 为此类问题，提供了统一接口，可以像命令式语言（JavaScript）直接“赋值”。

```

const state = mutable<State>();
const c = state.lens(`a`).lens(`b`).lens(`c`);
c.set(c);

```

离 `state.a.b.c = c` 这样的赋值有些距离，至少比较接近了。

4 实现我们的 Lens

知晓了 Lens 作用，我们将要着手实现了！

4.1 set、get

开始之前，不着急一步到位，先实现一个只有 set、get 功能的数据结构。这样的结构，当然可以作用于所有类型：

```

interface Mutable<T> {
  get: () => T;
  set: (value: T) => void;
}

const mutable = <T>(state: T): Mutable<T> => {
  const get = (): T => state;
  const set = (value: T) => { state = value; };

  return { get, set };
};

```

于是不管是 `mutable<string>('hello world')` 还是 `mutable<Array<number>>([])`，都可以使用 set 更新状态。

接下来，我们要实现取子数据的操作，例如 `state.lens(`a`).lens(`b`)`，依然能够使用 get、set。

我们目前知道，get 可以取得当前的状态，如果取 a，则可 `getA = () => get()['a']`；如果取 b，则 `getB = () => getA()['b']`。按此规律，无论多深的嵌套结构，我们都能以此得到对应的 get 方法。set 也是同理，说我们有这样的 set，能够更新当前状态，如果更新 a，则 `setA = a => set({ ...get(), a })`；如果更新 b，则 `setB = b => setA({ ...getA(), b })`。同样层层推进，每当使用 lens 方法，都需要重新构造新的 get、set，我们将需要将这对 get、set 传递下去：

```

interface Lens<T> {
  get: () => T;
  set: (value: T) => void;
}

```

```

}

type MutableRef<T> = Mutable<T>;

const mutable = <T>(state: T): Mutable<T> => {
  const lens = <K extends keyof T>(key: K): MutableRef<T[K]> => {
    const sublens: Lens<T[K]> = {
      get: () => get()[key],
      set: v => set({ ...get(), [key]: v })
    };

    return mutable(sublens);
  };
};

```

Lens<T> 的 get、set 说明，T 表示当前的状态，在取 lens 函数时，sublens 的类型为 Lens<T[K]>，正是说了它下面操作的子数据的状态。

注意： lens 最后调用了 mutable 自身，大家可能会有疑问，mutable 不是接受 T 吗，sublens 又是 Lens<_>，不管怎么看，类型都不匹配。没错的，这里偷了个懒，实际应该返回 MutableRef<T[K]>，但最后你会发现，Mutable 和 MutableRef 是等价的，只要有一个 $T \rightarrow \text{Lens}\langle T \rangle$ 的方法，就可以调用同样的接口。

至此，一个可以任意 get、set 的状态就完成了。我们可以试试效果。

```

const state = mutable<State>({ ... });
const alens = state.lens('a');
const blens = state.lens('a').lens('b'); // 或者 alens.lens('b');

a.set(a);
// 更新了state.a

b.set(b)
// 更新了state.a.b

state.get();
// 查看最后的状态

```

4.2 Prism

经过我们一番操作，如果不出意外，它已经是一个尽职尽责的状态管理：从使用上看，可以任意读取、更新状态；从功能上讲，我们可以随便拿取一部分数据，传递到任何地方，任由他人随便读取、更改。目前为止我们只遇到了简单数据，对于复合型的数据，我们依然照样处理吗？考虑一下这样数据：Array<{ name: string }> 和 Maybe<{ name: string }>，我们有办法利用 state.lens('name') 取出吗？我们再往下想一想，为什么遇到 Array 或 Maybe 整条链条就断裂？Array 表示零或任意个元素，如果我们要取 i 位置的元素，还要判断 i 是否在 Array 范围内——也就是我们可能会取到意料之外的值；Maybe = $\text{Just}T|\text{Nothing}$ ，表示有与无，当 Maybe 的值为空时，我们也无法取到想要的值。我们发现它们都有一个共同点：往下取值时，中间某处可能会出现意外之值。为了应对这样的状况，我们得加些什么。

从 Array 和 Maybe 的思考, 我们得到它们的取值未必必定成功, 所以需要有一个表示失败的 get, 以及能处理失败的 set:

```
interface Prism<S, T> {
  get: (state: S) => Maybe<T>;
  put: (state: S, value: T) => Maybe<S>;
}
```

对比 Lens, Prism 复杂了许多。首先 get 和 put 都显式接受 state, 这是因为这个接口需要向用户开放, 我们不能假定用户仅使用我们预想中的类型, 我们也需要允许用户自定义 prism, 所以用户必须知道当前处理的状态 state。最后是返回值都是 Maybe, Maybe 就是用来表示取值是否成功: Just 为成功取到值; Nothing 取值失败。作为演示, 我们以 Prism<string, string> 为例, 当我们遇到空字符串时, 取值失败, 返为 Nothing, 其余返回 Just:

```
const _notEmptyStr: Prism<string, string> = {
  get: s => {
    if (s.length === 0) {
      return Nothing;
    }
    else {
      return Just(s);
    }
  },
  put: (_, value) => {
    const s = value.trim();
    if (s.length === 0) {
      return Nothing;
    }
    else {
      return Just(s);
    }
  }
};
```

get 很好理解, put 我们忽略了原状态 (state), 用户更新值的时候, 如果更新的值不为空, 直接更新, 反之不更新, 所以忽略了 state。假设我们已经实现了接受任意 Prism<S, T> 的接口 prism, 预想中的行为:

```
const state = mutable<{ name: string }>({ name: "" });
const name = state.lens("name");
const namePrism = name.prism(_notEmptyStr);

namePrism.get(); // Nothing
name.get(); // ""

name.set("hello")
namePrism.get(); // Just "hello"
namePrism.put("")
name.get(); // "get"
```

namePrism 的 get 和 put 都受到了对应限制, 限制内容就是我们定义的 _notEmptyStr。一旦调用 prism, get 结果一定为 Maybe<T>, 而我们之前实现的 get 结果为 T, 并不对应: 很明显了, prism 调用后, 结果不

能是 `Mutable<T>`，而只能是其他。这个结构我们假定为 `Traversable<T>`，它同样也支持 `lens`、`get`、`put` 等操作，只是类型与 `Mutable<T>` 不一一对应，要重新实现。

```
interface Traversable<T> {
  get: () => Maybe<T>;
  put: (value: T) => void;
  lens: <K extends keyof T>(key: K) => Traversable<T[K]>;
  optic: <R>(op: Optic<T, R>) => Traversable<R>;
}
```

还有一个问题未解决，如何从 `Prism<S, T>` 生成 `Traversable<T>` 呢？`Prism<S, T>` 可是从了一个 `S`，该参数又该从哪里来呢？我们回头看看 `mutable` 的实现，它给了一个实现的方向，`mutable` 接受一个 `Lens<T>`，而 `Lens` 本身就能获取、更新 `S` 的能力，所以我们也只要构造一个合适的 `Lens`，传递进去，一切就顺理成章了。

由之前讨论可得出，`prism` 的 `get` 结果一定为 `Maybe`，那么可以传递 `Lens<Maybe<T>>`。我们可以从类型上解释这样做的合理法：`Lens<Maybe<T>>.get: () => Maybe<T>` 与 `Prism<S, T>.get: (state: S) => Maybe<T>`，类型十分接近；`Lens<Maybe<T>>.put: (value: Maybe<T>) => void` 与 `Prism<S, T>.put: (state: S, value: T) => Maybe<S>`，这部分复杂些，说 `Prism.put` 返回的 `Maybe`，那么 `Lens.get` 正好该类型参数。

于是我们可以动手实现了：

```
const traversable = <T>(op: Lens<Maybe<T>>): Traversable<T> => {
  const get: Traversable<T>["get"] = op.get;
  const put: Traversable<T>["put"] = v => {
    const a = op.get().map(_ => v);
    op.put(a);
  };

  const modify: Traversable<T>["modify"] = f =>
    op.get().ap(Just(f));

  const lens: Traversable<T>["lens"] = key => {
    type K = keyof key;
    const subOp: Lens<Maybe<T[K]>> = {
      get: () => op.get().map(s => s[key]),
      put: v => {
        return op.get()
          .chain(state => v.map(v => ({
            ...state,
            [key]: v
          })))
          .ifJust(put);
      }
    };
  };

  return traversable(subOp);
};

const prism: Traversable<T>["prism"] = _op => {
  type R = ReturnType<typeof _op>["get"];
```

```

const subOp: Lens<R> = {
  get: () => op.get().chain(_op.get),
  put: v => {
    return op.get()
      .chain(state => v.map(v => _op.put(state, v)))
      .ifJust(op.put);
  }
};
return traversable(subOp);
};
};

```

每个方法实现都比 `Mutable<T>` 复杂，因为都要考虑不正常取值情况，实现要注意一个问题，传入的 `op` 是唯一读取、更新父状态的 interface，一定要适时调用，不然不会更新父级状态。

同理，`Mutable<T>` 的 `prism` 实现如下：

```

const prism: Mutable<T>["prism"] = op => {
  type R = ReturnType<typeof op>["get"];
  const subOp: Lens<R> = {
    get: () => op.get(state),
    put: v => {
      v.chain(s => op.put(state, s))
        .ifJust(put);
    }
  };

  return traversable(subOp);
};

```

4.3 Primse 扩展

经过千辛万苦的摸索，终于得到了一个称心满意的 `prism`，这个接口十分开放，能允许用户实现任意类型的 `Prism`，例如上述的 `Prism<string, string>`，不一定非要复合型类型。这一节我们实现几个比较有用的 `Prism`。

```

const _notNil = <T>(): Prism<T | undefined, T> => ({
  get: Maybe.fromNullable,
  put: (state, v) => {
    return Maybe.fromNullable(state).map(_ => v);
  }
});

const _ix = <T>(index: number): Prism<Array<T>, T> => ({
  get: state => List.at(index, state),
  put: (state, x) => {
    if (index >= state.length) {
      return Nothing;
    }
    else {
      const xs = [...state];

```

```
        xs[index] = x;
        return Just(xs);
    }
}
});
```

`_notNil` 表示取非 `undefined` 值, 懂的都懂, 已不知受尽多少 `undefined` 苦楚了, 该 `Prism` 只取非 `undefined` 值, 遇到 `undefined` 则中止; `_ix` 表示取数组中某一位元素, 未取到则中断。我们直接看下面这个例子:

```
interface State {
  name: string;
  emails?: Array<{
    name: string;
    weight: number
  }>;
}

const state = mutable<State>();
const first = state.lens("emails").prism(_notNil).prism(_ix(1)).lens("name");
first.get();
first.put();
```

`prism` 可以在任意位置使用, 上面代码表示且当 `emails` 不为 `undefined`, 且 `emails` 的第二个元素存在, 那么 `get`、`put` 才能如愿得到、设置 `name` 值。

5 总结

我们从头实现一个偏函数式状态管理, 说偏函数式, 全文只有一处 (顶部) 更新状态, 其余各处都是重新生成新的状态, 一层一层传递上来。我们又开放了 `prism` 接口, 一方面用于处理可能中断的值, 另一方面允许用户扩展自定义 `Prism`, 让整个状态处理更加灵活。

`Lens` 给我们提供十分有用的抽象, Haskell 的 `Lens` 类型是 `Lens s t a b`, `Lens'` 才是 `Lens' s a`, 因为我们并不是要实现整个 `Lens`, 而是利用 `Lens` 实现状态的 `get`、`set`, 所以 `Lens'` 足够了。`Lens` 和 `Prism` 都是来自 Haskell `Lens`, `Lens` 表示聚焦一个子状态, 必定能得到一个值; `Prism` 也表示聚焦一个子状态, 它可能取得零个或一个值, 也就是我们上面表示的 `Maybe`。借此两个概念, 我们实现的状态必要的功能。

除此之外的 `Fold`、`Traversable` 我们都未实现, 这些可以交由标准库实现, 而且它们只有 `get`, 没有 `set` 功能。