二十行代码，搞懂 Observable

傅坦坦

上帝说，要有代码，于是沐浴焚香，大笔一挥，一个 Hello World

```
console.log('Hello world')
```

可是上帝对此并不高兴，他只想在需要的时候，才展示结果，于是我们把它放进了一个名为 `callback` 的**函数**盒子里：

```
function callback() {
  console.log('Hello world')
}
```

放在盒子里的程序不会自己执行，只有我们调用它的时候，才会执行，满足了上帝的需求，我们也学到一个知识点：

> 函数可以作为延迟代码执行的一种手段

演员就位

想象一个场景，我们需要一个提醒器，提醒我们该喝水了，于是可以像下面这样：

```
function reminder() {
  console.log('Time to drink water!') // 1
}
reminder() // Time to drink water!
```

看起来可行，但是我们更希望它只负责提醒，至于提醒之后做什么（行 1），由自己来决定，为了实现这个目的，可以将具体逻辑作为函数参数，传入：

```
function reminder(cb) {
  cb()
}
reminder(() => console.log('Time to drink water!')) // Time to drink water!
```

代码效果相同，但是这里我们可以自由地改变提醒事件发生之后的行为，传入不同的函数参数即可。

下面来玩儿一下 `reminder` 吧：

```
function reminder(cb) {
  cb()
  setTimeout(() => {
    cb()
  }, 1000)
}
reminder(() => console.log('Time to drink water!'))
```

上面的代码会先立即进行提醒，然后隔一秒钟再次提醒。

现实不总是那么尽如人意，我们的 `reminder` 可能会出错，我们也期望知道这一系列的提醒什么时候结束，但是现在 `reminder` 只有一个 `cb` 参数，已经满足不了我们了。最简单的方式是传入三个参数以供使用：

- 一个在提醒事件发生时，参数命名为 next
- 一个是错误发生时，命名为 error
- 一个是提醒结束时，命名为 complete

如果我们使用一个对象来包裹这三个参数：

```
function reminder(cb) {
  cb.next(1)
  cb.complete()
}
reminder({
  next: v => console.log('Time to drink water!', v),
  error: e => console.log(e),
  complete: () => console.log('Done'),
})
// Time to drink water! 1
// Done
```

我们来把 `reminder` 放进一个 `Reminder` 类里面，并重命名为 `remind` 方法：

```
class Reminder {
  remind(cb) {
    cb.next(1)
    cb.complete()
  }
}

const reminder = new Reminder()

reminder.remind({
  next: v => console.log('Time to drink water!', v),
```

```
  error: e => console.log(e),
  complete: () => console.log('Done'),
})
```

问题又来了，我们期望 `reminder` 的逻辑是可以动态注入的，一个解决方法是将函数体的内容通过构造参数传入：

```
class Reminder {
  constructor(behavior) {
    this.behavior = behavior
  }
  remind(cb) {
    this.behavior(cb)
  }
}

const reminder = new Reminder(cb => {
  cb.next(1)
  cb.complete()
})

reminder.remind({
  next: v => console.log('Time to drink water!', v),
  error: e => console.log(e),
  complete: () => console.log('Done'),
})
```

我们来做一些重命名

- `Reminder` → `Observable`
- `remind` → `subscribe`
- `cb` → `observer`
- `reminder` → `obs$`

```
class Observable {
  constructor(behavior) {
    this.behavior = behavior
  }
  subscribe(observer) {
    this.behavior(observer)
  }
}

const obs$ = new Observable(observer => {
  observer.next(1)
  observer.complete()
```

```
})
const observer = {
  next: v => console.log('Time to drink water!', v),
  error: e => console.log(e),
  complete: () => console.log('Done'),
}
obs$.subscribe(observer)
```

至此，20 行代码，甚至还包含一个文件末尾空行，我们实现了一个 `Observable`

思考

通过一系列的需求变更和拥抱变化，我们将最原始的 Hello World 重构成了一个具有基本功能的 `Observable`。再次观察和思考上面的代码，除了 `Observable` 的定义之外，存在三个语句：

- `obs$` 的定义，最重要的是定义了被观察者的行为
- `observer` 的定义，定义了在所观察的事件发生时候的行为
- `subscribe` 的调用，连接了 `Observable` 和 `observer`，如果没有这个连接，什么都不会发生

这三个语句职责单一且分明，提供了一种非常好的代码组织方式。

问题 1： `Observable` 是异步的吗?

看一下其内部实现，这完全取决于它的实现是否是异步调用了 `next`、`error` 和 `complete`。

问题 2: RxJS 和 Observable 的关系

RXJS 在 `Observable` 的基础上，提供了大量的操作符，来帮助实现复杂的 `Observable` 行为定义。

编辑于 2020-02-29