自动化黑盒测试

bifnudozhao@tencent.com

概要

- 背景
  - 例子
  - 痛点
- 思路
  - 分析
  - 解决办法
- 技术实现
  - 概念
  - 自动生成测试用例
  - 自动化测试
  - 可持续迭代闭环

背景 - 例子

前端UI界面的状态是复杂的，而且这种复杂性在代码层面是非常不直观，也非常难模拟。

比如一个商品卡片，可以有非常多的状态。如下图所示（实际上这个卡片有20+种状态）。

背景 - 例子

在目前市面上的方法下，我们如果想测试这个组件，我们只有下面的办法。

1. 模拟各种状态下的商品数据
2. 将数据传入组件并渲染出来
3. 人眼观察展示是否正确
4. 并且这个过程不能同时做，只能一个个状态模拟

背景 - 痛点

测试用例搭建

- mock数据手写和构造难度大
- 测试用例编码量大
- 测试用例debug成本大
- 整个搭建所需时间长

测试用例迭代

- 需要理解之前的测试用例
- 需要有针对性修改mock数据
- 虽然可以自动执行，但需要人工验证结果

“有没有办法自动生成测试用例？”
“有没有办法自动比对用例运行结果？”

思路 - 分析

对 UI 界面的最高层次抽象，可以看成是一个 `黑盒函数` 。

```
const UI = render(params)
```

这个渲染函数可以接受任意的参数，然后将界面渲染出来。根据不同的输入，会渲染出不同的 UI 界面。

```
const PRODUCT_UI1 = renderProduct(product1: Product)
const PRODUCT_UI2 = renderProduct(product2: Product)
const PRODUCT_UI3 = renderProduct(product3: Product)
```

需要注意到的是，这些入参是具有 `同一类型` 的，在商品卡片这个例子里，所有的输入都必须是一个商品信息。假设我们能暴力枚举所有商品信息，那理论上我们就能拿到所有可能的界面。

```
const UIs = []
while (canGenerateProduct()) {
  const product = generateProduct()
  const UI = renderProduct(product)
  UIs.push(UI)
}
```

思路 - 分析

有了一堆可能出现的界面之后，我们可以对这些界面进行正确性的判别，然后将这些**正确的界面**，以及产生这些正确界面的**输入**保存起来。

```
const testCases = [
  { params: product1, UI: PRODUCT_UI1 },
  { params: product2, UI: PRODUCT_UI2 },
  ...,
  { params: productN, UI: PRODUCT_UI_N },
]
```

我们可以把这些看成是我们的测试用例，当我们进行代码改动了之后，我们再取出正例里的 `params` 丢给我们的渲染函数，如果函数输出的 UI 和记录的一致，那我们就可以说代码通过了一个测试用例。

```
check(testCases, fn)
```

其中 `fn` 是被测函数。

技术实现 - 概念

通过上面的分析，我们先定义一组概念

- **测试用例**: 输入是数据，输出是 UI 界面，`{ input: params, output: image }`
- **测试用例正确性**: 测试用例输出的结果和正确的 UI 是否一致
- **测试用例覆盖率**: 分支覆盖率
- **有效用例**: 可以使覆盖率增加的用例
- **类型定义**: 一个数据结构的定义，包含其字段以及类型
- **元数据**: 一个用于描述数据结构的数据结构
- **数据生成器**: 给定一个类型描述，输出对应类型的数据

技术实现 - 自动生成测试用例

数据类型定义

在商品卡片例子中，商品卡片组件的输入数据是商品数据，我们需要随机生成商品数据，就需要一个商品数据生成器，可以根据类型元数据来构造**数据生成器**。

```
interface Product {
  title: string,          // 商品标题
  currentPrice: number,   // 商品价格
  originalPrice: number, // 商品原价
  tags: string[],         // 商品标签
  image: string[],        // 商品图片
  viewNumber: number,     // 商品热度
  saleNumber: number,     // 商品销量
}
```

数据元数据

有了这种类型定义，我们可以将其转化为**元数据**。

```
const ProductSchema = {
  name: 'Product',
  fields: [
    { name: 'title', type: 'string' },
    { name: 'currentPrice', type: 'number' },
    { name: 'originalPrice', type: 'number' },
    { name: 'tags', type: 'array[string]' },
    { name: 'image', type: 'array[image]' },
    { name: 'viewNumber', type: 'number' },
    { name: 'saleNumber', type: 'number' },
  ]
}
```

数据生成器

- 数据生成器接受一个数据元数据 `model`，输出对应数据元数据的数据

```
type Generator = (schema) => data
```

- 随机数据的每个字段，是由对应类型的随机生成器生成
- 生成的数据是一个符合类型的随机数据
- 生成数据的随机性，可以通过每种类型的生成器进行控制
- 复合类型可嵌套

```
export const productGenerator = (schema: ProductSchema) => {
  const randomData = {}

  _.each(schema.fields, field => {
    let value
    switch (field.type) {
      case Types.Number:
        value = randomeNumber()
        break
      case Types.String:
        value = randomString()
        break
      case Types.Boolean:
        value = randomBoolean()
        break
      // 各种类型调用对应的随机生成器
    }
    randomData[field.name] = value
  })

  return randomData
}
```

技术实现 - 自动生成测试用例

“上面生成的都是随机用例，我们怎样得到有用的用例？”

技术实现 - 自动生成测试用例

技术实现 - 自动生成测试用例

收集正例

在进行测试的时候，我们实际上是用测试结果来与正例进行对比，所以我们需要一个正例库

由于通过随机生成的方式我们可以得到满足一定覆盖率的测试用例集，而这些用例的输出实际上是图片，所以通过人工的方式建立正例库成本并不大

正例库可能是如下形式的

```
const correctCases = [
  { input: params_1, output: UI_1 },
  { input: params_2, output: UI_2 },
  { input: params_3, output: UI_3 },
  // ...
  { input: params_N, output: UI_N },
]
```

UI 比对获得测试通过率

有了正例库，我们便可以执行正例库里的用例

```
const runCases = (correctCases, fn) => {
  let correctCount = 0
  // 检查每个正例
  _.each(correctCases, case => {
    // 使用正例里的数据作为输入
    const UI = fn(case.input)
    // 如果被测函数的输出与标准输出一致，则通过
    if (isCorrect(UI, case.output)) {
      correctCount += 1
    }
  })

  // 计算正确比率
  return correctCount / correctCases.length
}
```

技术实现 - 自动化测试

前面使用到的 `isCorrect` 也是一个可以灵活配置的组件，其目标在于判断两个被测函数的输出是否一致。在当前场景下被测函数的输出是图片，所以对比图片是否一致有很多办法。

- 精确匹配：像素级匹配，由于输入是固定的，如果代码改动正确，理论上输出也应该一致
- 模糊匹配：可以使用 AI 方法，使用正例进行训练，得到一个有容错并相对智能的图片匹配器

技术实现 - 可持续迭代闭环

最开始提到的一些痛点消失了

之前	现在
测试用例建立成本很大	测试用例自动生成
测试用例结果需要人工检查	测试用例结果可以自动比对

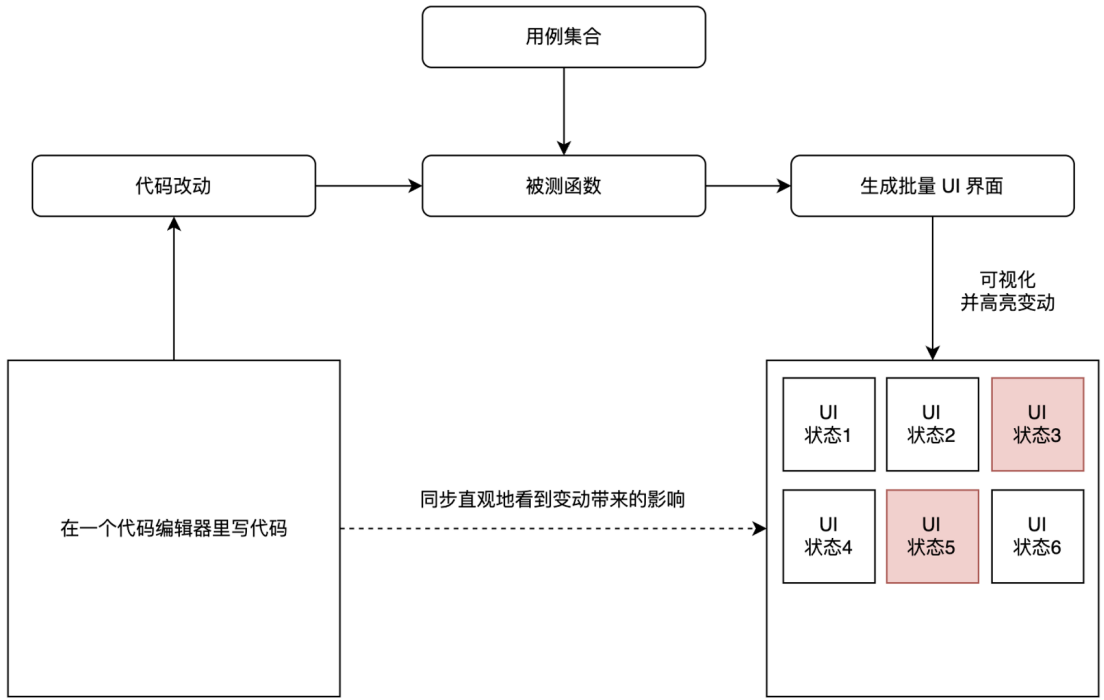
“那在迭代的过程中，如何使用以及更新这些用例呢？”

需求与代码改动

对于多状态组件，我们在面对代码的时候实际上是很抽象的

- 不清楚某一行代码是否会引起某些界面的变动
- 不清楚它是否会引入问题

有了一个组件的所有用例以及批量渲染UI的方式，我们可以实现下面的工作流。



技术实现 - 可持续迭代闭环

Q & A