浅谈柯里化

背景:

在 react 项目中使用 antd 表单的时候，遇到一些老项目，需要校验密码的强弱、校验输入的规则等，如果每次都是传正则和需要校验的字符串，有点麻烦。

```javascript
import React from "react";

const accountReg = /^[a-zA-Z0-9_-]{4,16}$/;
const passwordReg = /^(?=.*?[a-z])(?=.*?[A-Z])(?=.*?\d)(?=.*?[!@#$])[a-zA-Z\d!@#$]{10,16}$/;

const FormCom = () => {

  const checkReg = (reg, txt) => {
    return reg.test(txt)
  }

  //账号
  const checkAccount = (event) => {
    checkReg(accountReg, event.target.value);
    // 其他逻辑
  };

  //密码
  const checkPassword = (event) => {
    checkReg(passwordReg, event.target.value);
    // 其他逻辑
  };

  ...
  // 省去其他函数校验

  render() {
    return (
      <form>
        账号:
        <input onChange={checkAccount} type="text" name="account" />
        密码:
        <input onChange={checkPassword} type="password" name="password" />
      </form>
    );
  }
```

```
}

export default FormCom;
```

我们怎么解决类似的问题呢，我们可以使用柯里化函数来解决类似的问题。当然属于个人观点，如果其他方法，欢迎提出，进行学习

```javascript 复制代码
import React from "react";

const accountReg = /^[a-zA-Z0-9_-]{4,16}$/;
const passwordReg = /^(?=.*?[a-z])(?=.*?[A-Z])(?=.*?\d)(?=.*?[!@#$])[a-zA-Z\d!@#$]{10,16}$/;

const FormCom = () => {

  // 柯里化封装
  const curryCheck = (reg) => {
    return function(txt) {
        return reg.test(txt)
    }
  }

  //账号，这样就省去了一个参数的传递
  const checkAccount = curryCheck(accountReg);

  //密码，这样就省去了一个参数的传递
  const checkPassword = curryCheck(passwordReg);

  const checkAccountFn = () => {
    checkAccount(event.target.value);
    // 其他逻辑
  }

  const passwordFn = (event) => {
    checkPassword(event.target.value);
    // 其他逻辑
  };

  ...
  // 省去其他函数校验

  render() {
    return (
      <form>
        账号：
        <input onChange={checkAccountFn} type="text" name="account" />
        密码：
        <input onChange={passwordFn} type="password" name="password" />
```

```
      </form>
    );
  }
}

export default FormCom;
```

一、柯里化

在计算机科学中，柯里化（英语：Currying ），又译为卡瑞化或加里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数，而且返回结果的新函数的技术。这个技术由克里斯托弗·斯特雷奇以逻辑学家哈斯凯尔·加里命名的，尽管它是 Moses Schönfinkel 和戈特洛布·弗雷格发明的。柯里化其实也是函数式编程的思想。下面来举例说明什么是柯里化呢?

下面例子求和的过程，就是一个“乞丐版”的柯里化的过程

javascript 复制代码

```javascript
const addFn = (x, y, z) => {
  return x + y + z;
};

const addResultFn = addFn(1, 2, 3);
console.log("🚀 ~ file: preview.html ~ line 19 ~ addResultFn", addResultFn)

// 将上述过程转化为下面的实现过程就是柯里化

const sumFn = (x) => {
  return function(y) {
    return function(z) {
      return x + y + z;
    }
  }
}

const sum = sumFn(1)(2)(3);
```

上面的 sumFn 函数层层嵌套，肯定会被喷的，当然可以做一下简单的优化

dart 复制代码

```dart
const simplifySumFn = x => y => z => {
  return x + y + z;
}
```

当然，sumFn 这样调用，性能低下，若果层级过多，还会造成栈溢出，为什么还要这么做呢，当然有它自己用途以及好处。

柯里化的作用：

0. 单一原则：在函数式编程中，往往是让一个函数处理的问题尽可能单一，而不是一个函数处理多个任务。
1. 提高维护性以及降低代码的重复性

二、柯里化的场景

1、比如我们在求和中，以一定的数字为基数进行累加的时候，就用到了函数柯里化。当然函数柯里化感觉上是把简答的问题复杂化了，其实不然。比如：

```ini 复制代码
// 比如，基础分值是30 + 30；
const fractionFn = (x) => {
  const totalFraction =  x + x;
  return function(num) {
    return totalFraction + num;
  }
};

const baseFn = fractionFn(30);
const base1Fn = baseFn(1);
const base2Fn = baseFn(2);
```

这样来进行累加的话，是不是就简单、清晰明了呢。如果觉得这样的场景用到的不多的时候。别慌，那我在举一个例子。我们常用的日志输出，是不是都是具体的日期、时间以及加上具体的原因呢：

2、上述也可是实现打印日志的功能函数，细心的你不知道你发现了没，其实date, type每次还需要传参。是不是可以进行抽离呢，当然了，函数柯里化就可以完美的解决这个。

```typescript 复制代码
const date = new Date();

const logFn = (date, type, msg) => {
  console.log(`${date.getHours()} : ${date.getMinutes()} ${type} - ${msg}`);
}

logFn(date, 'warning', '声明的变量未使用');
logFn(date, 'warning', '暂未查询到数据');
```

```typescript 复制代码
const date = new Date();
const logFn = date => type => msg => {
  console.log(`${date.getHours()}:${date.getMinutes()} ${type} - ${msg}`);
}
const nowLogFn = logFn(date);
nowLogFn('warning')('声明的变量未被引用');
```

三、柯里化函数的实现

是不是比上面第一种的要清晰呢，但是还是有点不完美的地方，因为这个过程都是我们手动进行柯里化的，难道每次都要手动进行转换吗？我们程序员不就是来解决能程序解决的，绝不手动重复的吗？

```scss 复制代码
const selfCurryFn = (fn) => {
  const fnLen = fn.length;  // fn 接收的参数
  function curry(...args) {
    const argLen = args.length; // curry 接收的参数
    if(argLen >= fnLen) {
      return fn.apply(this, args); // 如果外面绑定 this 的话，直接绑定到fn上
    } else {
      // 参数个数没有达到时继续接收剩余的参数
      function otherCurry(...args2) {
        return curry.apply(this, args.concat(args2))
      }
      return otherCurry;
    }
  }
  return curry;
}
•
const selfAddFn = (x, y, z) => {
  return x + y + z;
}
•
const selfSum = selfCurryFn(selfAddFn);
•
console.log("🚀 ~ file: preview.html ~ line 80 ~ selfSum(1, 2, 3)", selfSum(1, 2, 3))
console.log("🚀 ~ file: preview.html ~ line 81 ~ selfSum(1, 2)(3)", selfSum(1, 2)(3))
console.log("🚀 ~ file: preview.html ~ line 81 ~ selfSum(1, 2)(3)", selfSum(1)(2)(3))
```

四、柯里化在其他库的应用

1、我们常用的 Redux，里面其实也用到了柯里化。Redux 的中间件提供的是位于 action 被发起之后数据流加上中间件后变成了

view -> action -> middleware -> reducer -> store 。在 middleware 这个节点可以进行一些“副作用”的 操作，比如打印日志等等。

```javascript
export default function applyMiddleware(...middlewares) {
    return createStore => (...args) => {
        // 利用传入的 createStore 和 reducer 和创建一个 store
        const store = createStore(...args)
        let dispatch = () => {
            throw new Error()
        }
        const middlewareAPI = {
            getState: store.getState,
            dispatch: (...args) => dispatch(...args)
        }
        // middlewareAPI 这个参数分别执行一遍
        const chain = middlewares.map(middleware => middleware(middlewareAPI))
        // 组装成一个新的函数，即新的 dispatch
        dispatch = compose(...chain)(store.dispatch)
        return {
            ...store,
            dispatch
        }
    }
}
```

2、组合函数是 JavaScript 开发过程中一种对函数的使用组合模式。

- 我们对某一个函数进行调用，执行 fn1、fn2，这两个函数是依次执行
- 每次我们都需要进行两个函数的调用，操作上就会显示的重复
- 那么我们是不是可以将 fn1、fn2 组合起来，自动一次调用呢？ 其实实现上述的过程就是组合函数（compose function）。

```dart
const fn1 = (num) => {
  return num + 10;
}

const fn2 = (num) => {
  return num * num;
}
```

```
const a = 10;
const result = fn2(fn1(a));
```

加入我们有很多类似的函数，需要这么调用，这样每次调用都比较麻烦，也比较冗余。当时我们可以给进行组合，然后在进行调用。

dart 复制代码

```
function compose (fn1, fn2) {
   return function(num) {
     return fn2(fn1(num));
   }
}

const fn1 = (num) => {
  return num + 10;
}

const fn2 = (num) => {
  return num * num;
}

const a = 10;
const newFn = compose(fn1, fn2);
const result = newFn(a);
```

通用组合函数的实现

ini 复制代码

```
function createCompose(...fns) {
  const length = fns.length;
  for(let i = 0; i < length; i++) {
    if(typeof fns[i] !== 'function') {
      throw new TypeError('arguments is not function');
    }
  }

  function compose(...args ) {
    let index = 0;
    let result = length ? fns[index].apply(this, args) : args;
    while(++index < length) {
      result = fns[index].call(this, result);
    }
  }
  return compose;
}
```

其实 redux 里的 compose 函数就是类似上面的实现过程，将多个函数进行聚合，然后在进行函数的执行。