前端元编程——使用注解加速你的前端开发

> https://zhuanlan.zhihu.com/p/274328551

无论你用 React，Vue，还是 Angular，你还是要一遍一遍写相似的 CRUD 页面，一遍一遍，一遍一遍，一遍又一遍……

“天下苦秦久矣”~ ~

前端开发的“痛点”在哪里？

现在的前端开发，我们有了世界一流的 UI 库 React，Vue，Angular，有了样式丰富的 UI 组件库 Tea (腾讯云 UI 组件库，类似 Antd Design)， 有了方便强大的脚手架工具(例如，create react app)。但是我们在真正业务代码之前，通常还免不了写大量的样板代码。

现在的 CRUD 页面代码通常：

1. 太轻的“Model”或着“Service”，大多时候只是一些 API 调用的封装。
2. 胖“View“，View 页面中有展示 UI 逻辑，生命周期逻辑，CRUD 的串联逻辑，然后还要塞满业务逻辑代码。
3. 不同的项目业务逻辑不同，但是列表页，表单，搜索这三板斧的样板代码，却要一遍一遍占据着前端工程师的宝贵时间。

特别是 CRUD 类应用的样板代码受限于团队风格，后端 API 风格，业务形态等，通常内在逻辑相似书写上却略有区别，无法通过一个通用的库或者框架来解决（上图中背景越深，越不容易有一个通用的方案）。

说好的“**数据驱动的前端开发**”呢?

对于这个“痛点”——怎么尽可能的少写模版代码，就是本文尝试解决的问题。

我们尝试使用 JavaScript 新特性`Decorator`和`Reflect`元编程来解决这个问题。

前端元编程

> 从 ECMAScript 2015 开始，JavaScript 获得了 `Proxy` 和 `Reflect` 对象的支持，允许你拦截并定义基本语言操作的自定义行为（例如，属性查找，赋值，枚举，函数调用等）。借助这两个对象，你可以在 JavaScript 元级别进行编程。 MDN

在正式开始之前，我们先复习下`Decorator`和`Reflect`。

Decorator

这里我们简单介绍 Typescript 的 `Decorator`，ECMAScript 中 `Decorator` 尚未定稿，但是不影响我们日常的业务开发(*Angular 同学就在使用 Typescript 的Decorator*)。

简单来说，`Decorator` 是可以**标注修改类及其成员**的新语言特性，使用 `@expression` 的形式，可以附加到，类、方法、访问符、属性、参数上。

TypeScript 中需要在 `tsconfig.json` 中增加 `experimentalDecorators` 来支持：

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

比如可以使用类修饰器来为类扩展方法。

```
// offer type
abstract class Base {
  log() {}
}

function EnhanceClass() {
  return function (Target) {
    return class extends Target {
      log() {
        console.log("---log---");
      }
    };
  };
}
@EnhanceClass()
class Person extends Base {}

const person = new Person();
person.log();

// ---log---
```

更多查看 typescript 官方的文档。

Handbook - Decorators

Reflect

Reflect 是 ES6 中就有的特性，大家可能对它稍微陌生，Vue3 中依赖 Reflect 和 Proxy 来重写它的响应式逻辑。

简单来说，`Reflect` 是一个人内置的对象，提供了拦截 JavaScript 操作的方法。

```
const _list = [1, 2, 3];
```

```
const pList = new Proxy(_list, {
  get(target, key, receiver) {
    console.log("get value reflect:", key);
    return Reflect.get(target, key, receiver);
  },
  set(target, key, value, receiver) {
    console.log("set value reflect", key, value);
    return Reflect.set(target, key, value, receiver);
  },
});
pList.push(4);
// get value reflect:push
// get value reflect:length
// set value reflect 3 4
// set value reflect length 4
```

Reflect Metadata

Reflect Metadata 是 ES7 的一个提案，Typescript 1.5+就有了支持。要使用需要：

- `npm i reflect-metadata --save`
- 在 `tsconfig.json` 里配置 `emitDecoratorMetadata` 选项

简单来说，**Reflect Metadata 能够为对象添加和读取元数据。**

如下可以使用内置的`design:key`拿到属性类型：

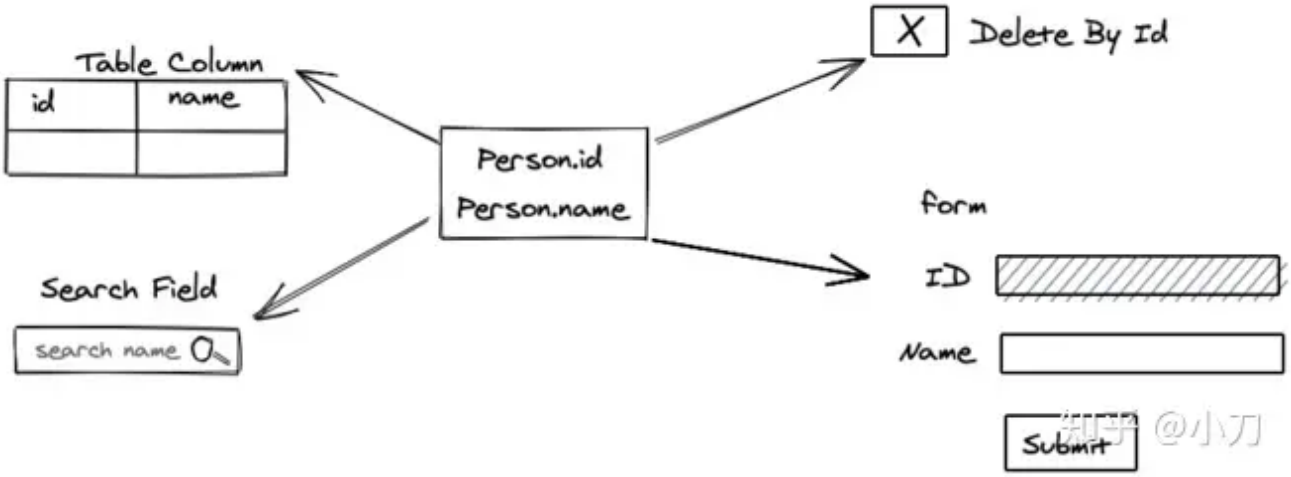
```
function Type(): PropertyDecorator {
  return function (target, key) {
    const type = Reflect.getMetadata("design:type", target, key);
    console.log(`${key} type: ${type.name}`);
  };
}

class Person extends Base {
  @Type()
  name: string = "";
}
// name type: String
```

使用 Decorator,Reflect 减少样板代码

回到正题——使用 Decorator 和 Reflect 来减少 CRUD 应用中的样板代码。

什么是 CRUD 页面?



CRUD 页面无需多言，列表页展示，表单页修改 ……包括 API 调用， 都是围绕某个数据结构(图中`Person`)展开，增、删、改、查。

基本思路

基本思路很简单，就像上图，Model 是中心，我们就是**借助Decorator和Reflect将 CRUD 页面所需的样板类方法属性元编程在 Model 上。**进一步延伸**数据驱动 UI**的思路。

1. 借助 Reflect Matadata 绑定 CRUD 页面信息到 Model 的属性上
2. 借助 Decorator 增强 Model，生成 CRUD 所需的夜班代码

Show Me The Code

下文，我们用**TypeScript**和**React**为例，组件库使用腾讯Tea component 解说这个方案。

首先我们有一个函数来生成不同业务的属性装饰函数。

```
function CreateProperDecoratorF<T>() {
  const metaKey = Symbol();
  function properDecoratorF(config: T): PropertyDecorator {
    return function (target, key) {
      Reflect.defineMetadata(metaKey, config, target, key);
    };
  }
  return { metaKey, properDecoratorF };
}
```

一个类装饰器，处理通过数据装饰器收集上来的元数据。

```
export function EnhancedClass(config: ClassConfig) {
  return function (Target) {
    return class EnhancedClass extends Target {};
  };
}
```

API Model 映射

TypeScript 项目中第一步自然是将后端数据安全地转换为`type`，`interface`或者`Class`，这里 Class 能在编译后在 JavaScript 存在，我们选用`Class`。

```
export interface TypePropertyConfig {
  handle?: string | ServerHandle;
}

const typeConfig = CreateProperDecoratorF<TypePropertyConfig>();
export const Type = typeConfig.properDecoratorF;

@EnhancedClass({})
export class Person extends Base {
  static sexOptions = ["male", "female", "unknow"];

  @Type({
    handle: "ID",
  })
  id: number = 0;

  @Type({})
  name: string = "";

  @Type({
    handle(data, key) {
      return parseInt(data[key] || "0");
    },
  })
  age: number = 0;

  @Type({
    handle(data, key) {
```

```
      return Person.sexOptions.includes(data[key]) ? data[key] : "unknow";
    },
  })
  sex: "male" | "female" | "unknow" = "unknow";
}
```

重点在`handle?: string | ServerHandle`函数，在这个函数处理 API 数据和前端数据的转换，然后在`constructor`中集中处理。

```
export function EnhancedClass(config: ClassConfig) {
  return function (Target) {
    return class EnhancedClass extends Target {
      constructor(data) {
        super(data);
        Object.keys(this).forEach((key) => {
          const config: TypePropertyConfig = Reflect.getMetadata(
            typeConfig.metaKey,
            this,
            key
          );
          this[key] = config.handle
            ? typeof config.handle === "string"
              ? data[config.handle]
              : config.handle(data, key)
            : data[key];
        });
      }
    };
  };
}
```

列表页 TablePage

列表页中一般使用 Table 组件，无论是 Tea Component 还是 Antd Design Component 中，样板代码自然就是写那一大堆 Colum 配置了，配置哪些 key 要展示，表头是什么，数据转化为显示数据……

首先我们收集 Tea Table 所需的`TableColumn`类型的 column 元数据。

```
import { TableColumn } from "tea-component/lib/table";
export type EnhancedTableColumn<T> = TableColumn<T>;
export type ColumnPropertyConfig = Partial<EnhancedTableColumn<any>>;

const columnConfig = CreateProperDecoratorF<ColumnPropertyConfig>();
export const Column = columnConfig.properDecoratorF;

@EnhancedClass({})
export class Person extends Base {
  static sexOptions = ["male", "female", "unknow"];

  id: number = 0;

  @Column({
```

```
    header: "person name",
  })
  name: string = "";

  @Column({
    header: "person age",
  })
  age: number = 0;

  @Column({})
  sex: "male" | "female" | "unknow" = "unknow";
}
```

然后在 EnhancedClass 中收集，生成 column 列表。

```
function getConfigMap<T>(
  F: any,
  cachekey: symbol,
  metaKey: symbol
): Map<string, T> {
  if (F[cachekey]) {
    return F[cachekey]!;
  }
  const item = new F({});
  F[cachekey] = Object.keys(item).reduce((pre, cur) => {
    const config: T = Reflect.getMetadata(metaKey, item, cur);
    if (config) {
      pre.set(cur, config);
    }
    return pre;
  }, new Map<string, T>());
  return F[cachekey];
}

export function EnhancedClass(config: ClassConfig) {
  const cacheColumnConfigKey = Symbol("cacheColumnConfigKey");
  return function (Target) {
    return class EnhancedClass extends Target {
      [cacheColumnConfigKey]: Map<string, ColumnPropertyConfig> | null;
      /**
       * table column config
       */
      static get columnConfig(): Map<string, ColumnPropertyConfig> {
        return getConfigMap<ColumnPropertyConfig>(
          EnhancedClass,
          cacheColumnConfigKey,
          columnConfig.metaKey
        );
      }

      /**
       * get table colums
       */
```

```
      static getColumns<T>(): EnhancedTableColumn<T>[] {
        const list: EnhancedTableColumn<T>[] = [];
        EnhancedClass.columnConfig.forEach((config, key) => {
          list.push({
            key,
            header: key,
            ...config,
          });
        });
        return list;
      }
    };
  };
}
```

Table 数据一般是分页，而且调用方式通常很通用，也可以在 EnhancedClass 中实现。

```
export interface PageParams {
  pageIndex: number;
  pageSize: number;
}

export interface Paginabale<T> {
  total: number;
  list: T[];
}
export function EnhancedClass(config: ClassConfig) {
  return function (Target) {
    return class EnhancedClass extends Target {
      static async getList<T>(params: PageParams): Promise<Paginabale<T>> {
        const result = await getPersonListFromServer(params);
        return {
          total: result.count,
          list: result.data.map((item) => new EnhancedClass(item)),
        };
      }
    };
  };
}
```

自然我们封装一个更简易的 Table 组件。

```
import { Table as TeaTable } from "tea-component/lib/table";
import React, { FC, useEffect, useState } from "react";
import { EnhancedTableColumn, Paginabale, PageParams } from "./utils";
import { Person } from "./person.service";

function Table<T>(props: {
  columns: EnhancedTableColumn<T>[];
  getListFun: (param: PageParams) => Promise<Paginabale<T>>;
}) {
  const [isLoading, setIsLoading] = useState(false);
```

```
  const [recordData, setRecordData] = useState<Paginabale<T>>();
  const [pageIndex, setPageIndex] = useState(1);
  const [pageSize, setPageSize] = useState(20);
  useEffect(() => {
    (async () => {
      setIsLoading(true);
      const result = await props.getListFun({
        pageIndex,
        pageSize,
      });
      setIsLoading(false);
      setRecordData(result);
    })();
  }, [pageIndex, pageSize]);
  return (
    <TeaTable
      columns={props.columns}
      records={recordData ? recordData.list : []}
      addons={[
        TeaTable.addons.pageable({
          recordCount: recordData ? recordData.total : 0,
          pageIndex,
          pageSize,
          onPagingChange: ({ pageIndex, pageSize }) => {
            setPageIndex(pageIndex || 0);
            setPageSize(pageSize || 20);
          },
        }),
      ]}
    />
  );
}

export default Table;
```

1. `getConfigMap<T>(F: any, cachekey: symbol,metaKey: symbol): Map<string,T>` 收集元数据到 Map
2. `static getColumns<T>(): EnhancedTableColumn<T>[]` 得到 table 可用 column 信息。

```
const App = () => {
  const columns = Person.getColumns<Person>();
  const getListFun = useCallback((param: PageParams) => {
    return Person.getList<Person>(param);
  }, []);
  return <Table<Person> columns={columns} getListFun={getListFun} />;
};
```

效果很明显，不是吗？ 7 行写一个 table page。

Form 表单页

表单，自然就是字段的 name，label，require，validate，以及提交数据的转换。

Form 表单我们使用Formik + Tea Form Component + yup(数据校验)。Formik 使用 React Context 来提供表单控件所需的各种方法数据，然后借助提供的 Field 等组件，你可以很方便的封装你的业务表单组件。

```
import React, { FC } from "react";
import { Field, Form, Formik, FormikProps } from "formik";
import { Form as TeaForm, FormItemProps } from "tea-component/lib/form";
import { Input, InputProps } from "tea-component/lib/input";
import { Select } from "tea-component/lib/select";

type CustomInputProps = Partial<InputProps> &
  Pick<FormItemProps, "label" | "name">;

type CustomSelectProps = Partial<InputProps> &
  Pick<FormItemProps, "label" | "name"> & {
    options: string[];
  };

export const CustomInput: FC<CustomInputProps> = (props) => {
  return (
    <Field name={props.name}>
      {({
        field, // { name, value, onChange, onBlur }
        form: { touched, errors }, // also values, setXXXX, handleXXXX, dirty, isValid,
status, etc.
        meta,
      }) => {
        return (
          <TeaForm.Item
            label={props.label}
            required={props.required}
            status={meta.touched && meta.error ? "error" : undefined}
            message={meta.error}
          >
            <Input
              type="text"
              {...field}
              onChange={(value, ctx) => {
                field.onChange(ctx.event);
              }}
            />
          </TeaForm.Item>
        );
      }}
    </Field>
  );
};

export const CustomSelect: FC<CustomSelectProps> = (props) => {
  return (
    <Field name={props.name}>
      {({
        field, // { name, value, onChange, onBlur }
```

```
        form: { touched, errors }, // also values, setXXXX, handleXXXX, dirty, isValid,
status, etc.
        meta,
      }) => {
        return (
          <TeaForm.Item
            label={props.label}
            required={props.required}
            status={meta.touched && meta.error ? "error" : undefined}
            message={meta.error}
          >
            <Select
              {...field}
              options={props.options.map((value) => ({ value }))}
              onChange={(value, ctx) => {
                field.onChange(ctx.event);
              }}
            />
          </TeaForm.Item>
        );
      }}
    </Field>
  );
};
```

照猫画虎，我们还是先收集 form 所需的元数据

```
import * as Yup from "yup";

export interface FormPropertyConfig {
  validationSchema?: any;
  label?: string;
  handleSubmitData?: (data: any, key: string) => { [key: string]: any };
  required?: boolean;
  initValue?: any;
  options?: string[];
}

const formConfig = CreateProperDecoratorF<FormPropertyConfig>();
export const Form = formConfig.properDecoratorF;

@EnhancedClass({})
export class Person extends Base {
  static sexOptions = ["male", "female", "unknow"];

  @Type({
    handle: "ID",
  })
  id: number = 0;

  @Form({
    label: "Name",
    validationSchema: Yup.string().required("Name is required"),
```

```
    handleSubmitData(data, key) {
      return {
        [key]: (data[key] as string).toUpperCase(),
      };
    },
    required: true,
    initValue: "test name",
  })
  name: string = "";

  @Form({
    label: "Age",
    validationSchema: Yup.string().required("Age is required"),
    handleSubmitData(data, key) {
      return {
        [key]: parseInt(data[key] || "0"),
      };
    },
    required: true,
  })
  age: number = 0;

  @Form({
    label: "Sex",
    options: Person.sexOptions,
  })
  sex: "male" | "female" | "unknow" = "unknow";
}
```

有了元数据，我们可以在 EnhancedClass 中生成 form 所需：

- initialValues
- 数据校验的 validationSchema
- 各个表单组件所需的，name，label，required 等
- 提交表单的数据转换 handle 函数

```
export type FormItemConfigType<T extends any> = {
  [key in keyof T]: {
    validationSchema?: any;
    handleSubmitData?: FormPropertyConfig["handleSubmitData"];
    form: {
      label: string;
      name: string;
      required: boolean;
      message?: string;
      options: string[];
    };
  };
};

export function EnhancedClass(config: ClassConfig) {
  return function (Target) {
    return class EnhancedClass extends Target {
```

```
  [cacheTypeConfigkey]: Map<string, FormPropertyConfig> | null;
  /**
   * table column config
   */
  static get formConfig(): Map<string, FormPropertyConfig> {
    return getConfigMap<FormPropertyConfig>(
      EnhancedClass,
      cacheTypeConfigkey,
      formConfig.metaKey
    );
  }

  /**
   * get form init value
   */
  static getFormInitValues<T extends EnhancedClass>(item?: T): Partial<T> {
    const data: any = {};
    const _item = new EnhancedClass({});
    EnhancedClass.formConfig.forEach((config, key) => {
      if (item && key in item) {
        data[key] = item[key];
      } else if ("initValue" in config) {
        data[key] = config.initValue;
      } else {
        data[key] = _item[key] || "";
      }
    });
    return data as Partial<T>;
  }

  static getFormItemConfig<T extends EnhancedClass>(overwriteConfig?: {
    [key: string]: any;
  }): FormItemConfigType<T> {
    const formConfig: any = {};
    EnhancedClass.formConfig.forEach((config, key) => {
      formConfig[key] = {
        form: {
          label: String(config.label || key),
          name: String(key),
          required: !!config.validationSchema,
          options: config.options || [],
          ...overwriteConfig,
        },
      };
      if (config.validationSchema) {
        formConfig[key].validationSchema = config.validationSchema;
      }
      if (config.handleSubmitData) {
        formConfig[key].handleSubmitData = config.handleSubmitData;
      }
    });
    return formConfig as FormItemConfigType<T>;
  }
```

```
    static handleToFormData<T extends EnhancedClass>(item: T) {
      let data = {};
      EnhancedClass.formConfig.forEach((config, key) => {
        if (item.hasOwnProperty(key)) {
          data = {
            ...data,
            ...(EnhancedClass.formConfig.get(key).handleSubmitData
              ? EnhancedClass.formConfig.get(key).handleSubmitData(item, key)
              : {
                  [key]: item[key] || "",
                }),
          };
        }
      });
      return data;
    }
  };
};
```

在 FormPage 中使用

```
export const PersonForm: FC<{
  onClose: () => void
}> = (props) => {
  const initialValues = Person.getFormInitValues<Person>();
  const formConfig = Person.getFormItemConfig<Person>();
  const schema = Object.entries(formConfig).reduce((pre, [key, value]) => {
    if (value.validationSchema) {
      pre[key] = value.validationSchema;
    }
    return pre;
  }, {});
  const validationSchema = Yup.object().shape(schema);

  function onSubmit(values) {
    const data = Person.handleToFormData(values);
    setTimeout(() => {
      console.log("---send to server", data);
      props.onClose();
    }, 10000);
  }
  return (
    <Formik
      initialValues={initialValues}
      onSubmit={onSubmit}
      validationSchema={validationSchema}
    >
      {(formProps: FormikProps<any>) => {
        return (
          <TeaForm>
            <CustomInput {...formConfig.name.form} />
```

```
            <CustomInput {...formConfig.age.form} />
            <CustomSelect {...formConfig.sex.form} />
            <Button
              type="primary"
              htmlType="submit"
              onClick={() => {
                formProps.submitForm();
              }}
            >
              Submit
            </Button>
          </TeaForm>
        );
      }}
    </Formik>
  );
};
```

40 行，我们有了个一个功能完备表单页

ts-model-decorator demots-model-decorator.stackblitz.io/

元编程减少样板代码Demo

yijian166/ts-model-decoratorgithub.com/yijian166/ts-model-decorator

效果

上文包含了不少的代码，但是大部头在如何将元数据转换成为页面组件可用的数据，也就是元编程的部分。

而业务页面，7 行的 Table 页面，40 行的 Form 页面，已经非常精简功能完备了。根据笔者实际项目中估计，可以节省至少 40%的代码量。

元编程 vs. 配置系统

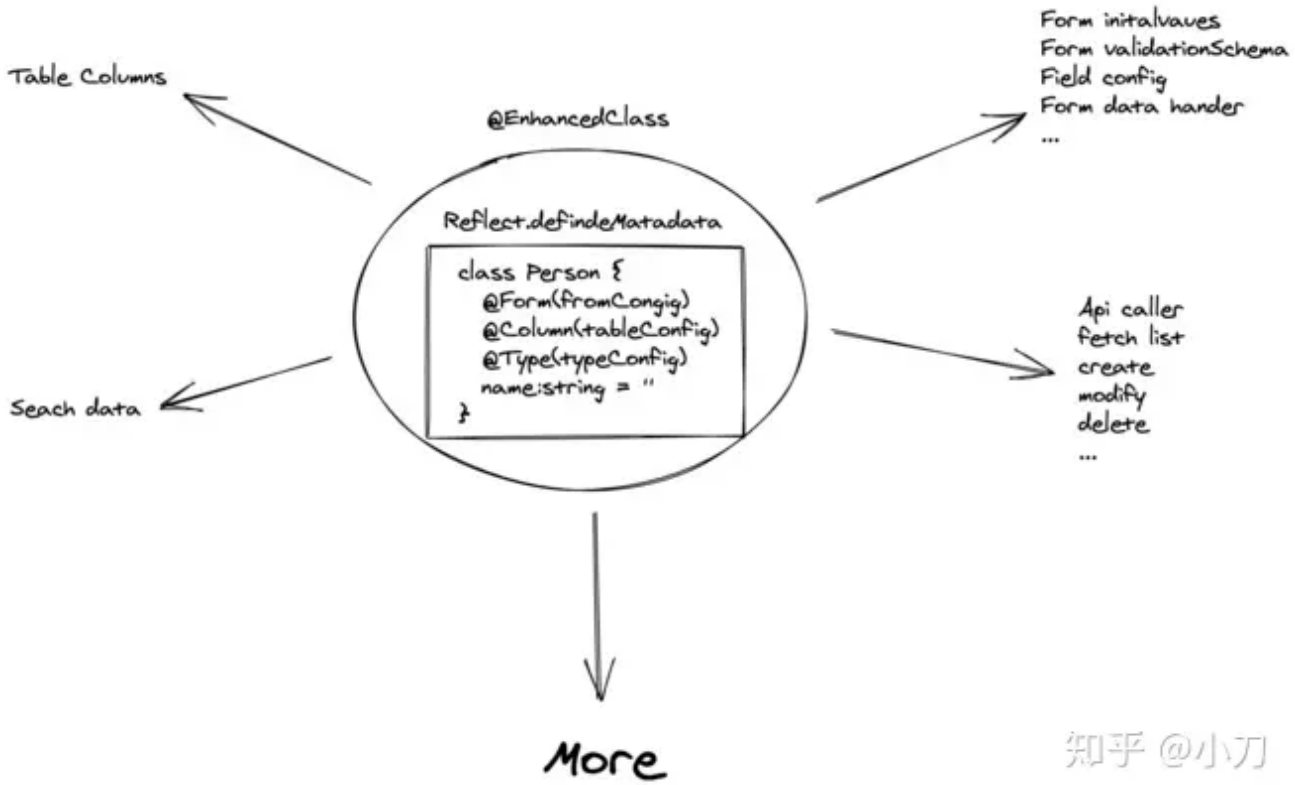
写到尾声，你大概会想到某些配置系统，前端 CRUD 这个从古就有的需求，自然早就有方案，用的最多的就是配置系统，在这里不会过多讨论。

简单来说，就是一个单独的系统，配置类似上文的元信息，然后使用固定模版生成代码。

思路实际上和本文的元编程类似，只是元编程成本低，你不需要单独做一个系统，更加轻量灵活，元编程代码在运行时，想象空间更大……

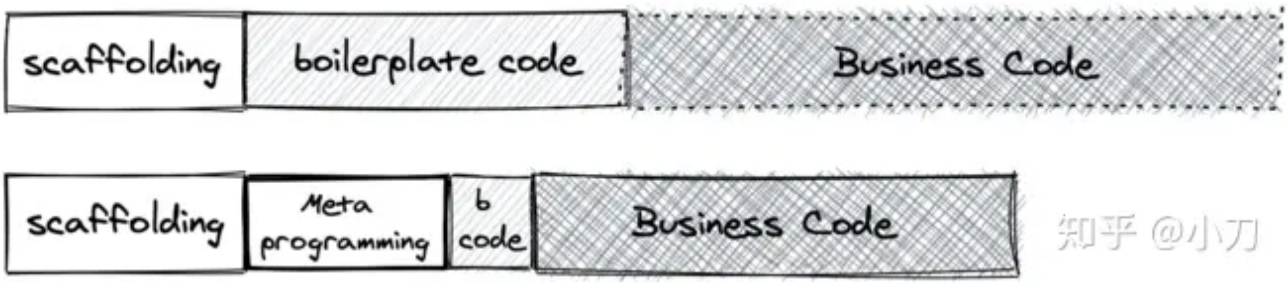
总结

上面只是 table，form 页面的代码展示，由此我们可以引申到很多类似的地方，甚至 API 的调用代码都可以在元编程中处理。



元编程——将元数据转换成为页面组件可用的数据，这部分恰恰可以在团队内非常好共享也需要共同维护的部分，带来的好处也很明显：

- 最大的好处自然就是**生产效率的提高**了，而且是低成本的实现效率的提升(相比配置系统)。一些简单单纯的 CURD 页面甚至都不用写代码了。
- 更易维护的代码:
- “瘦 View“，专注业务，
- 更纯粹的 Model，你可以和 redux，mobx 配合，甚至，你可以从 React，换成 Angular）
- 最后更重要的是，元编程是一个低成本，灵活，渐进的方案。它是一个运行时的方案，你不需要一步到罗马，徐徐图之 …… - ……



前端元编程，较少你的样板代码，加速前端开发

最后，本文更多是一次实践，一种思路，一种元编程在前端开发中的应用场景，最重要的还是抛砖引玉，希望前端小伙伴们能形成自己团队的的元编程实践，来解放生产力，更快搬砖～～