Ts使用总结

declare module InstanceType

```ts
declare module "*.vue" {
  import type { DefineComponent } from "vue";
  const component: DefineComponent<{}, {}, any>;
  export default component;
}

import type PageSearch from "./PageSearch.vue";
export type PageSearchInstance = InstanceType<typeof PageSearch>;
```

CSSProperties

```ts
export type IObject = Record<string, any>;

import { CSSProperties } from "vue";
export type IToolsButton = {
  perm?: Array<string> | string; // 权限标识(可以是完整权限字符串如'sys:user:add'或操作权限如'add')
  attrs?: Partial<ButtonProps> & { style?: CSSProperties }; // 按钮属性
  render?: (row: IObject) => boolean; // 条件渲染
};

```

Partial Record Array<{}>

```ts
import type { FormProps, TableProps, ColProps, ButtonProps, CardProps } from "element-plus";

type DateComponent = "date-picker" | "time-picker" | "time-select" | "custom-tag" | "input-tag";
type InputComponent = "input" | "select" | "input-number" | "cascader" | "tree-select";
type OtherComponent = "text" | "radio" | "checkbox" | "switch" | "icon-select" | "custom";
export type IComponentType = DateComponent | InputComponent | OtherComponent;
```

```ts

export type IForm = Partial<Omit<FormProps, "model" | "rules">>;

export type IFormItems<T = IComponentType> = Array<{
  // 组件类型(如input,select,radio,custom等)
  type: T;
  // 标签提示
  tips?: string | IObject;
  // 组件属性
  attrs?: IObject;
  // 组件可选项(只适用于select,radio,checkbox组件)
  options?: Array<{ label: string; value: any; [key: string]: any }> | Ref<any[]>;
  // 验证规则
  rules?: FormItemRule[];
  // layout组件Col属性
  col?: Partial<ColProps>;
  // 组件事件
  events?: Record<string, (...args: any) => void>;
  // 初始化数据函数扩展
  initFn?: (item: IObject) => void;
}>;

export interface IModalConfig<T = any> {
  // dialog组件属性
  dialog?: Partial<Omit<DialogProps, "modelValue">>;
  // drawer组件属性
  drawer?: Partial<Omit<DrawerProps, "modelValue">>;
  // form组件属性
  form?: IForm;
  // 表单项
  formItems: IFormItems<IComponentType>;
  // 提交之前处理
  beforeSubmit?: (data: T) => void;
  // 提交的网络请求函数(需返回promise)
  formAction?: (data: T) => Promise<any>;
}
```

Omit 联合类型& Promise

```ts
type ToolbarTable = "edit" | "view" | "delete";
export type IToolsButton = {
  perm?: Array<string> | string; // 权限标识(可以是完整权限字符串如'sys:user:add'或操作权限如'add')
  attrs?: Partial<ButtonProps> & { style?: CSSProperties }; // 按钮属性
```

```
  render?: (row: IObject) => boolean; // 条件渲染
};

export interface IContentConfig<T = any> {
  // table组件属性
  table?: Omit<TableProps<any>, "data">;
  // 分页组件位置(默认: left)
  pagePosition?: "left" | "right";
  // pagination组件属性
  pagination?:
    | boolean
    | Partial<
        Omit<PaginationProps, "v-model:page-size" | "v-model:current-page" | "total" | "currentPage">
      >;
  // 列表的网络请求函数(需返回promise)
  indexAction: (queryParams: T) => Promise<any>;
  // 数据格式解析的回调函数
  parseData?: (res: any) => {
    list: IObject[];
    [key: string]: any;
  };
  // 修改属性的网络请求函数(需返回promise)
  modifyAction?: (data: {
    [key: string]: any;
    field: string;
    value: boolean | string | number;
  }) => Promise<any>;
  // 删除的网络请求函数(需返回promise)
  deleteAction?: (ids: string) => Promise<any>;
  // 后端导出的网络请求函数(需返回promise)
  exportAction?: (queryParams: T) => Promise<any>;
  // 前端全量导出的网络请求函数(需返回promise)
  exportsAction?: (queryParams: T) => Promise<IObject[]>;
  // 导入模板
  importTemplate?: string | (() => Promise<any>);
  // 后端导入的网络请求函数(需返回promise)
  importAction?: (file: File) => Promise<any>;
  // 前端导入的网络请求函数(需返回promise)
  importsAction?: (data: IObject[]) => Promise<any>;
  // 主键名(默认为id)
  pk?: string;
  // 表格工具栏(默认:add,delete,export,也可自定义)
  toolbar?: Array<ToolbarLeft | IToolsButton>;
  // 表格工具栏右侧图标(默认:refresh,filter,imports,exports,search)
  defaultToolbar?: Array<ToolbarRight | IToolsButton>;
  // table组件列属性(额外的属性templet,operat,slotName)
```

```
  cols: Array<{
    type?: "default" | "selection" | "index" | "expand";
    // 模板
    templet?:
      | "image"
      | "list"
      | "url"
      | "switch"
      | "input"
      | "price"
      | "percent"
      | "icon"
      | "date"
      | "tool"
      | "custom";
    // list模板相关参数
    selectList?: IObject;
    // tool模板相关参数
    operat?: Array<ToolbarTable | IToolsButton>;
    [key: string]: any;
    // 初始化数据函数
    initFn?: (item: IObject) => void;
  }>;
}
```

类型保护

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
     return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, ['Jarid]
```

泛型接口

```
// 写法1
interface ConfigFn {
        <T>(value:T):T;
}

var getData:ConfigFn = function<T>(value:T) : T {
        return value;
}

getData<string>('张三');
getData<string>(1243);  //错误


// 写法2:
interface ConfigFn<T> {
    (value:T):T;
}

var getData:ConfigFn<string> = function<T>(value:T) : T {
        return value;
}

getData('20');  /*正确*/


function getData<T>(value:T) : T {
        return value;
}
```

泛型类

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) {
        return x + y;
};
```

接口定义

```ts
interface Person {
 // 只读属性
    readonly id: number;
        // 确定属性
    name: string;
        // 可选属性
    age?: number;
        // 字符串索引
    [propName: string]: string;
         // 数字索引,类似数组
    [index: number]: string;
        // 函数
    (name: string, age: number): void;
        // 方法
    getName(id: number): string;
         // 构造函数
    new(name: string, age: number): Person;
}


interface Person {
  name: string;
  age: number;
}

type MyPerson = {
  [P in keyof Person as `get${P & string}`]: Person[P];
};

// getname: string, getnumber: number}
```

类的静态部分和实例部分

```ts

//  ClockConstructor 为构造函数所用
//  ClockInterface 为实例方法所用
//  createClock 的第一个参数是ClockConstructor类型，在createClock(AnalogClock, 7, 32)里，
// 会检查 AnalogClock 是否符合构造函数签名。
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}
```

```
12
13 // 第一个参数ctor的类型是接口 ClockConstructor，在这里就为类的静态部分指定需要实现的接口
14 function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {
15     return new ctor(hour, minute);
16 }
17
18 // 类 DigitalClock 实例化出来的对象（类的实例部分）应该满足这个接口的规则
19 class DigitalClock implements ClockInterface {
20     constructor(h: number, m: number) { }
21     tick() {
22         console.log("beep beep");
23     }
24 }
25 class AnalogClock implements ClockInterface {
26     constructor(h: number, m: number) { }
27     tick() {
28         console.log("tick tock");
29     }
30 }
31
32 let digital = createClock(DigitalClock, 12, 17);
33 let analog = createClock(AnalogClock, 7, 32);
34
```

混合类型

```
1  // 函数接口
2  interface Counter {
3    (start: number): string
4  }
5
6  let counter: Counter
7  counter = function (start: number) {
8    console.log(number)
9  }
10 // 调用
11 counter(12)
12
13
14 // 对象接口
15 interface Counter {
16     interval: number;
17     reset(): void;
```

```
}
```

```
// 下面的例子相当于上面2个接口声明合并
// 一个对象可以同时做为函数和对象使用，并带有额外的属性。如:下文中的变量c
interface Counter {
    (start: number): string; // 函数
    interval: number; // 对象属性
    reset(): void; // 对象方法
}

function getCounter(): Counter {
    // 通过类型断言，将函数对象转换为Counter类型，转换后的对象不但实现了函数接口的描述，使之成为一个函数，还具有interval属性和reset()方法
    let counter = <Counter>function (start: number) {
                console.log(number)
        };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

vue3源码中的使用

```
export interface ReactiveEffect<T = any> {
  (): T
  _isEffect: true
  active: boolean
  raw: () => T
  deps: Array<Dep>
  options: ReactiveEffectOptions
}

function createReactiveEffect<T = any>(
  fn: () => T,
  options: ReactiveEffectOptions
): ReactiveEffect<T> {
```

```
  const effect = function reactiveEffect(...args: unknown[]): unknown {
    return run(effect, fn, args)
  } as ReactiveEffect
  effect._isEffect = true
  effect.active = true
  effect.raw = fn
  effect.deps = []
  effect.options = options
  return effect
}
```

接口继承类

```
class Person {
  type: string // 这里是类的描述
}

interface Child extends Person { // Child 接口继承自 Person 类，因此规范了 type 属性
  log(): void
  // 这里其实有一个 type: string
}

// 上面的 Child 接口继承了 Person 对 type 的描述，还定义了 Child 接口本身 log 的描述

// 第一种写法
class Girl implements Child {
  type: 'child' // 接口继承自 Person 的
  log() {} // 接口本身规范的
}

// 第二种写法
class Boy extends Person implements Child { // 首先 extends 了 Person 类，然后还需满足 Child 接口的描述
  type: 'child'
  log() {}
}

// 当Person有 private属性时，只能如下写法，(不推荐使用)
 // 写法 (only)
class Boy extends Person implements Child { // 首先 extends 了 Person 类，然后还需满足 Child 接口的描述
  type: 'child'
```

```
  log() {}
}

```

重载

```
// java中的重载：同名函数，参数不一样. 允许一个函数接受不同数量或类型的参数时，作出不同的处理.
// typescript中的重载：通过为同一个函数提供多个函数类型定义，一个函数体实现多种功能的目的.
// ts为了兼容es5 以及 es6 重载的写法和java中有区别.

function reverse(x: number): number;  // 函数定义
function reverse(x: string): string;  // 函数定义
function reverse(x: number | string): number | string {  // 函数实现
  if (typeof x === 'number') {
      return Number(x.toString().split('').reverse().join(''));
  } else if (typeof x === 'string') {
      return x.split('').reverse().join('');
  }
}
```

扩展

getFirst

getLast

Pop

shift

Push

Unshift

StartsWith

Replace

TrimLeft

TrimRight

Trim

GetFunParameters

GetFunReturnType

GetValue

Zip

CapitalizeStr

CamelCase

DropSubStr

AppendArgument

Mapping

ReverseArr

GetOptional

GetRequired

StringToUnion

BuildArray

```
type getFirst<T extends unknown[]> = T extends [infer F, ...infer R] ? F : never;
type getLast<T extends unknown[]> = T extends [...infer R, infer L] ? L : never;
type Pop<T extends unknown[]> = T extends [...infer R, infer L] ? R : never;
type Shift<T extends unknown[]> = T extends [infer R, ...infer L] ? L : never;


```

```ts
type res = getFirst<arr>; // 1


type str = '00-test-end';

type StartsWith<Str extends string, Prefix extends string> = Str extends `${Prefix}${string}` ? true : false;

type res1 = StartsWith<str, '00-'>; // true
type res2 = StartsWith<str, '01-'>; // false


type Replace<Str extends string, From extends string, To extends string> = Str extends `${infer Prefix}${From}${infer Suffix}`
  ? `${Prefix}${To}${Suffix}`
  : Str;

type res1 = Replace<'abc', 'a', 'A'>; // Abc
type res2 = Replace<'abc', 'd', 'D'>; // abc


type TrimLeft<S extends string> = S extends `${' ' | '\n' | '\t'}${infer Rest}` ? TrimLeft<Rest> : S;
type TrimRight<S extends string> = S extends `${infer Rest}${' ' | '\n' | '\t'}` ? TrimRight<Rest> : S;
type Trim<S extends string> = TrimLeft<TrimRight<S>>;

type res1 = TrimLeft<' \n 123  '>; // '123  '
type res2 = TrimRight<'  123 '>; // '  123'
type res3 = Trim<'  123  '>; // '123'




type fun1 = (name: string, age: number) => void;

type GetFunParameters<F extends Function> = F extends (...args: infer Args) => unknown ? Args : never;
type res = GetFunParameters<fun1>; // [name: string, age: number]


type fun1<T> = (name: string, age: number) => T;

type GetFunReturnType<F extends Function> = F extends (...args: any[]) => infer ReturnType ? ReturnType : never;

```

```ts
type res = GetFunReturnType<fun1<{ code: number; value: string }>>; // [value: string, code: number]


type GetValue<Obj, K extends string> = K extends keyof Obj ? Obj[K] : never;

type res = GetValue<{ ref?: 1; value: 2 }, 'ref'>; // 1 | undefined


type Push<Arr extends unknown[], Ele> = [...Arr, Ele];
type res = Push<[0, 1, 2], 3>; // [0, 1, 2, 3]


type Unshift<Arr extends unknown[], Ele> = [Ele, ...Arr];
type res = Unshift<[0, 1, 2], 3>; // [3, 0, 1, 2]


type tuple1 = [1, 2];
type tuple2 = ['guang', 'dong'];

type tuple1 = [1, 2, 3, 4];
type tuple2 = ['guang', 'dong', 'bei', 'jing'];

type Zip<Arr1 extends unknown[], Arr2 extends unknown[]> = Arr1 extends [infer Arr1_1, ...infer Arr1_Other]
  ? Arr2 extends [infer Arr2_1, ...infer Arr2_Other]
    ? [[Arr1_1, Arr2_1], ...Zip<Arr1_Other, Arr2_Other>]
    : []
  : [];

type res = Zip<tuple1, tuple2>; // [[1, "guang"], [2, "dong"], [3, "bei"], [4, "jing"]]



type CapitalizeStr<Str extends string> = Str extends `${infer F}${infer R}` ? `${Uppercase<F>}${R}` : '';
type res = CapitalizeStr<'yang'>; // Yang


type CamelCase<Str extends string> = Str extends `${infer Left}_${infer Right}${infer Rest}` ? `${Left}${CapitalizeStr<Right>}${CamelCase<Rest>}` : Str;
```

```ts

type res = CamelCase<'yang_long_hi'>; // yangLongHi



type DropSubStr<Str extends string, SubStr extends string> = Str extends `${infer Prefix}${SubStr}${infer Suffix}`
  ? DropSubStr<`${Prefix}${Suffix}`, SubStr>
  : Str;

type res = DropSubStr<'yanglong~~~', '~'>; // yanglong




type AppendArgument<Fun extends Function, Arg> = Fun extends (...args: infer Args) => infer R ? (...args: [...Args, Arg]) => R : Fun;

type res = AppendArgument<(name: string) => void, number>; // (name: string, args_1: number) => void



type obj = {
  name: string;
  age: number;
  gender: boolean;
};
type Mapping<Obj extends object> = {
  readonly [k in keyof Obj]?: Obj[k];
};

type res = Mapping<obj>;
// { readonly name?: string | undefined; readonly age?: number | undefined; readonly gender?: boolean | undefined };


type obj = {
  readonly name: string;
  readonly age?: number;
  gender?: boolean;
};
type Mapping<Obj extends object> = {
  -readonly [k in keyof Obj]-?: Obj[k];
};

type res = Mapping<obj>; // { name: string; age: number; gender: boolean };
```

```ts

type obj = {
  name: string;
  age: number;
  gender: boolean;
  hobby: string[];
};
type Mapping<Obj extends Record<string, any>, ValueType> = {
  [K in keyof Obj as Obj[K] extends ValueType ? K : never]: Obj[K];
};

type res = Mapping<obj, string | number>; // { name: string; age: number; };



type ReverseArr<Arr extends unknown[]> = Arr extends [...infer Left, infer Value] ? [Value, ...ReverseArr<Left>] : [];

type res = ReverseArr<[1, 2, 3, 4]>; // [4, 3, 2, 1]


type IsEqual<A, B> = (A extends B ? true : false) & (B extends A ? true : false);
type Includes<Arr extends unknown[], Item> = Arr extends [...infer Left, infer Value]
  ? IsEqual<Value, Item> extends true
    ? true
    : Includes<Left, Item>
  : false;

type res1 = Includes<[1, 2, 3, 4], 5>; // false
type res2 = Includes<[1, 2, 3, 4], 4>; // true


type RemoveItem<Arr extends unknown[], Item> = Arr extends [...infer Left, infer Value]
  ? IsEqual<Value, Item> extends true
    ? [...Left]
    : [...RemoveItem<Left, Item>, Value]
  : [];

type res1 = RemoveItem<[1, 2, 3, 4], 2>; // [1, 3, 4]
type res2 = RemoveItem<[1, 2, 3, 4], 5>; // [1, 2, 3, 4]

// 另外一种思路
type RemoveItem2<Arr extends unknown[], Item, Res extends unknown[] = []> = Arr extends [infer Value, ...infer Left]
```

```ts
  ? IsEqual<Value, Item> extends true
    ? RemoveItem2<Left, Item, Res>
    : RemoveItem2<Left, Item, [...Res, Value]>
  : Res;



type BuildArray<Len extends number, Ele = unknown, Arr extends unknown[] = []> = IsEqual<Arr['length'], Len> extends false
  ? BuildArray<Len, Ele, [...Arr, Ele]>
  : Arr;

type res1 = BuildArray<3>; // [unknown, unknown, unknown]



type ReplaceAll<Str extends string, From extends string, To extends string> = Str extends `${infer Prefix}${From}${infer Suffix}`
  ? ReplaceAll<`${Prefix}${To}${Suffix}`, From, To>
  : Str;

type ReplaceAll2<Str extends string, From extends string, To extends string> = Str extends `${infer Prefix}${From}${infer Suffix}`
  ? `${Prefix}${To}${ReplaceAll<Suffix, From, To>}`
  : Str;

type res1 = ReplaceAll<'1,2,3', ',', ''>; // 123

type StringToUnion<Str extends string> = Str extends `${infer First}${infer Res}` ? First | StringToUnion<Res> : never;

type res1 = StringToUnion<'123456'>; // "1" | "2" | "3" | "4" | "5" | "6"



type isRequired<Key extends keyof Obj, Obj> = {} extends Pick<Obj, Key> ? never : Key;

type GetRequired<Obj extends Record<string, any>> = {
  [Key in keyof Obj as isRequired<Key, Obj>]: Obj[Key];
};



type GetOptional<Obj extends Record<string, any>> = {
```

```ts
  [Key in keyof Obj as {} extends Pick<Obj, Key> ? Key : never]: Obj[Key];
};

// 删除索引签名
type Dong = {
  [key: string]: any;
  sleep(): void;
};

type RemoveIndexSignature<Obj extends Record<string, any>> = {
  [Key in keyof Obj as Key extends `${infer Str}` ? Str : never]: Obj[Key];
};

type res1 = RemoveIndexSignature<Dong>; // { sleep: () => void };

```

[参考](https://juejin.cn/post/7448441576140095551)