Typescript

extends

```
// 泛型约束
function identity<T extends { name: string }>(arg: T): T {
    console.log(arg.name);
    return arg;
}

// 条件类型
type IsString<T> = T extends string ? "yes" : "no";
type Result = IsString<"hello">;  // Result类型为"yes"
```

对象定义

```
// 接口定义
interface Person {
      name: string;
      age: number;
  }
  function greet(person: Person) {
      return "Hello " + person.name;
  }

  // 类型别名
  type Person = {
      name: string;
      age: number;
  };
  function greet(person: Person) {
      return "Hello " + person.name;
  }

```

属性可选定义

```ts
interface PaintOptions {
    shape: Shape;
    xPos?: number;
    yPos?: number;
}

// readonly 不能写入
interface SomeType {
    readonly prop: string;
}

// 索引签名
// 索引签名的属性类型必须是 string 或者是 number。
interface StringArray {
    [index: number]: string;
}

interface NumberOrStringDictionary {
    [index: string]: number | string;
    length: number; // ok, length is a number
    name: string; // ok, name is a string
}

// 继承
interface BasicAddress {
    name?: string;
    street: string;
    city: string;
    country: string;
    postalCode: string;
}
interface AddressWithUnit extends BasicAddress {
    unit: string;
}

// 继承多个
interface Colorful {
    color: string;
}
interface Circle {
    radius: number;
}
interface ColorfulCircle extends Colorful, Circle {
}

const cc: ColorfulCircle = {
    color: "red",
```

```
    radius: 42,
};


// 继承类型互斥会报错
interface Colorful {
    color: string;
}
interface ColorfulSub extends Colorful {
    color: number
}
// 交叉类型不会报错，但类型取交集 never
interface Colorful {
    color: string;
}
type ColorfulSub = Colorful & {
    color: number
}
```

泛型

```
interface Box<Type> {
  contents: Type;
}
let box: Box<string>;


function setContents<Type>(box: Box<Type>, newContents: Type) {
    box.contents = newContents;
}


function identity<Type>(arg: Type): Type {
    return arg;
}

let myIdentity: <Type>(arg: Type) => Type = identity;
// 对象类型
let myIdentity: { <Type>(arg: Type): Type } = identity;


interface GenericIdentityFn<Type> {
    (arg: Type): Type;
}
// 将泛型参数作为整个接口的参数 GenericIdentityFn
```

```
25   let myIdentity: GenericIdentityFn<number> = identity;
```

泛型类

```
1  class GenericNumber<NumType> {
2    zeroValue: NumType;
3    add: (x: NumType, y: NumType) => NumType;
4  }
5
6  let myGenericNumber = new GenericNumber<number>();
7  myGenericNumber.zeroValue = 0;
8  myGenericNumber.add = function (x, y) {
9    return x + y;
10 };
```

泛型约束

```
1  interface Lengthwise {
2    length: number;
3  }
4   // 通过 extend 实现约束
5  function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
6    // Now we know it has a .length property, so no more error
7    console.log(arg.length);
8    return arg;
9  }
```

使用类型参数

```
1  function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
2    return obj[key];
3  }
4
5  let x = { a: 1, b: 2, c: 3, d: 4 };
6
7  getProperty(x, "a");
8
9  // Argument of type '"m"' is not assignable to parameter of type '"a" | "b" |
   "c" | "d"'.
10 getProperty(x, "m");
11
```

使用类类型

```
function create<Type>(c: { new (): Type }): Type {
  return new c();
}
```

```
class BeeKeeper {
  hasMask: boolean = true;
}
class ZooKeeper {
  nametag: string = "Mikle";
}
class Animal {
  numLegs: number = 4;
}
class Bee extends Animal {
  keeper: BeeKeeper = new BeeKeeper();
}
class Lion extends Animal {
  keeper: ZooKeeper = new ZooKeeper();
}

function createInstance<A extends Animal>(c: new () => A): A {
  return new c();
}

createInstance(Lion).keeper.nametag;
createInstance(Bee).keeper.hasMask;
```

keyof

```
type Arrayish = { [n: number]: unknown };
type A = keyof Arrayish;
// type A = number

type Mapish = { [k: string]: boolean };
type M = keyof Mapish;
// JavaScript 对象的属性名会被强制转为一个字符串，所以数字类型也可以
// type M = string | number
```

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

type PersonKeys = keyof Person;  // "name" | "age"

// 使用PersonKeys
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];  // 在这里，key的类型安全地限制在了T的所有键中
}

const person: Person = { name: "Alice", age: 30 };
const name = getProperty(person, "name");  // 正确
// const error = getProperty(person, "notExist");  // 错误：类型“"notExist"”的参数不能赋给类型“"name" | "age"”的参数。
```

数字字面量联合

```ts
const NumericObject = {
  [1]: "一号",
  [2]: "二号",
  [3]: "三号"
};

type result = keyof typeof NumericObject

// typeof NumbericObject 的结果为:
// {
//   1: string;
//   2: string;
//   3: string;
// }
// 所以最终的结果为:
// type result = 1 | 2 | 3
```

typeof

`typeof` 只能对标识符和属性使用

```ts
function f() {
```

```
  return { x: 10, y: 3 };
}
type P = ReturnType<typeof f>;

// type P = {
//     x: number;
//     y: number;
// }
```

对对象使用

```
const person = { name: "kevin", age: "18" }
type Kevin = typeof person;

// type Kevin = {
//       name: string;
//       age: string;
// }
```

对函数使用

```
function identity<Type>(arg: Type): Type {
  return arg;
}

type result = typeof identity;
// type result = <Type>(arg: Type) => Type
```

对enum使用

```
enum UserResponse {
  No = 0,
  Yes = 1,
 }

 type result = typeof UserResponse;

// ok
const a: result = {
      "No": 2,
      "Yes": 3
```

```
}

//result 类型类似于:
// {
//    "No": number,
//   "YES": number
// }
```

infer

```
// 定义一个条件类型，用于获取函数返回值的类型
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

// 使用示例
function getString(): string {
    return "hello";
}

function getNumber(): number {
    return 123;
}

type StringReturnType = ReturnType<typeof getString>;  // string
type NumberReturnType = ReturnType<typeof getNumber>;  // number

// 通过infer R，我们能够在不具体指定函数返回类型的情况下，推断出函数的返回类型。
// 这对于处理高阶函数或者类型封装时特别有用。
```

----------高级类型函数实现 ----------

Record<K, T>

创建一个对象类型，其属性键为K，属性值为T。

```
type Property = 'u1' | 'u2'
type User = Record<Property, string>
const u: User = {
    u1: 'xx',
    u2: 'bb'
}
```

Partial<T>

将某个类型的所有属性变为可选

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

// 使用Partial使Person接口中的属性都变为可选
type PartialPerson = Partial<Person>;
```

Required<T>

将所有属性变为必选

```ts
interface Person {
    name?: string;
    age?: number;
}

// 使用Required使Person接口中的属性都变为必选
type RequiredPerson = Required<Person>;
```

Pick<T, K>

从类型 `T` 中选取一组属性 `K` 来构造类型

```ts
interface Person {
    name: string;
    age: number;
    location: string;
}

// 使用Pick选取Person中的'name'和'age'属性
type PersonBasics = Pick<Person, 'name' | 'age'>;
//{
//    name: string;
//    age: number;
//}
```

Omit<T, K>

从类型 `T` 中排除一组属性 `K` 来构造类型。

```ts
interface Person {
    name: string;
    age: number;
    location: string;
}

// 使用Omit排除Person中的'location'属性
type PersonWithoutLocation = Omit<Person, 'location'>;
//{
//    name: string;
//    age: number;
//}
```

Exclude<T,U>

从类型T中排除那些可赋值给U的类型

```ts
type PrimitiveTypes = string | number | boolean;

type NonBooleanTypes = Exclude<PrimitiveTypes, boolean>;
// NonBooleanTypes是string | number
```

Extract<T, U>

从类型T中提取那些可赋值给U的类型

```ts
type PossibleValues = "a" | "b" | "c" | 1 | 2;

type StringValues = Extract<PossibleValues, string>;
// StringValues是"a" | "b" | "c"
```

NonNullable

从类型T中排除null和undefined

```ts
type MaybeNumber = number | null | undefined;
```

```
2
3 type JustNumber = NonNullable<MaybeNumber>;
4 // JustNumber是number，已经排除了null和undefined
```

ReturnType<T>

用于获取函数T的返回类型

```
1 function getUser() {
2   return { name: "John", age: 30 };
3 }
4
5 type User = ReturnType<typeof getUser>;
6 // User的类型是{name: string; age: number;}
```

Parameters<T>

用于获取函数T的参数类型，以元组的形式返回

```
1 function greet(name: string, age: number): string {
2   return `Hello, ${name}. You are ${age}.`;
3 }
4
5 type GreetParameters = Parameters<typeof greet>;
6 // GreetParameters的类型是[string, number]
```

ConstructorParameters<T>

用于获取构造函数类型的所有参数类型，以元组的形式返回

```
1 class Person {
2   constructor(public name: string, public age: number) {}
3 }
4
5 type PersonConstructorParameters = ConstructorParameters<typeof Person>;
6 // PersonConstructorParameters的类型是[string, number]
```

----------最佳实践----------

动态属性访问

```typescript
interface Employee {
  id: number;
  name: string;
  department: string;
}

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const employee: Employee = {
  id: 1,
  name: "John Doe",
  department: "HR",
};

// 安全访问
const name: string = getProperty(employee, "name");
console.log(name); // 输出: John Doe
```

条件类型

```typescript
type ApiResponse<T> = T extends "user" ? User : T extends "settings" ? Settings
 : never;

function fetchApi<T extends "user" | "settings">(endpoint: T): ApiResponse<T> {
  // 模拟API调用
  // 实际应用中，这里会是异步请求逻辑
  return undefined as any;
}

// 根据不同的参数，返回值类型会自动匹配
const user: User = fetchApi("user");
const settings: Settings = fetchApi("settings");
```

使用映射类型改造旧代码

```typescript
interface OldInterface {
  prop1: string;
```

```
  prop2: number;
  prop3: boolean;
}

type PartialInterface = Partial<OldInterface>;

// 现在PartialInterface的所有属性都是可选的
const update: PartialInterface = {
  prop1: "new value",
  // prop2和prop3是可选的，可以不提供
};
```

利用高级类型进行错误处理

```
type Result<T> = { success: true; value: T } | { success: false; error: Error };

function safeParse<T>(json: string): Result<T> {
  try {
    return { success: true, value: JSON.parse(json) };
  } catch (error) {
    return { success: false, error: error instanceof Error ? error : new Error(String(error)) };
  }
}

const result = safeParse<{ name: string }>('{"name":"John"}');
if (result.success) {
  console.log(result.value.name); // 安全访问
} else {
  console.error(result.error.message);
}
```

定义对象优先使用interface

联合类型、交叉类型或其他复杂类型操作时,应该使用type