- 类型索引
- 类型约束
- 映射类型
- 条件类型

3.2.1. 交叉类型

通过 `&` 将多个类型合并为一个类型，包含了所需的所有类型的特性，本质上是一种并的操作

语法如下：

TypeScript | 复制代码

```typescript
T & U
```

适用于对象合并场景，如下将声明一个函数，将两个对象合并成一个对象并返回：

TypeScript | 复制代码

```typescript
function extend<T , U>(first: T, second: U) : T & U {
    let result: <T & U> = {}
    for (let key in first) {
        result[key] = first[key]
    }
    for (let key in second) {
        if(!result.hasOwnProperty(key)) {
            result[key] = second[key]
        }
    }
    return result
}
```

3.2.2. 联合类型

联合类型的语法规则和逻辑“或”的符号一致，表示其类型为连接的多个类型中的任意一个，本质上是一个交的关系

语法如下：

TypeScript | 复制代码

```typescript
T | U
```

例如 `number` | `string` | `boolean` 的类型只能是这三个的一种，不能共存

如下所示：

TypeScript 复制代码

```typescript
function formatCommandline(command: string[] | string) {
  let line = '';
  if (typeof command === 'string') {
    line = command.trim();
  } else {
    line = command.join(' ').trim();
  }
}
```

3.2.3. 类型别名

类型别名会给一个类型起个新名字，类型别名有时和接口很像，但是可以作用于原始值、联合类型、元组以及其它任何你需要手写的类型

可以使用 `type SomeName = someValidTypeAnnotation` 的语法来创建类型别名：

TypeScript 复制代码

```typescript
type some = boolean | string

const b: some = true // ok
const c: some = 'hello' // ok
const d: some = 123 // 不能将类型“123”分配给类型“some”
```

此外类型别名可以是泛型：

TypeScript 复制代码

```typescript
type Container<T> = { value: T };
```

也可以使用类型别名来在属性里引用自己：

TypeScript 复制代码

```typescript
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

可以看到，类型别名和接口使用十分相似，都可以描述一个对象或者函数

两者最大的区别在于，`interface` 只能用于定义对象类型，而 `type` 的声明方式除了对象之外还可以定义交叉、联合、原始类型等，类型声明的方式适用范围显然更加广泛

3.2.4. 类型索引

`keyof` 类似于 `Object.keys` ，用于获取一个接口中 Key 的联合类型。

TypeScript | 复制代码

```typescript
interface Button {
    type: string
    text: string
}

type ButtonKeys = keyof Button
// 等效于
type ButtonKeys = "type" | "text"
```

3.2.5. 类型约束

通过关键字 `extend` 进行约束，不同于在 `class` 后使用 `extends` 的继承作用，泛型内使用的主要作用是对泛型加以约束

TypeScript | 复制代码

```typescript
type BaseType = string | number | boolean

// 这里表示 copy 的参数
// 只能是字符串、数字、布尔这几种基础类型
function copy<T extends BaseType>(arg: T): T {
  return arg
}
```

类型约束通常和类型索引一起使用，例如我们有一个方法专门用来获取对象的值，但是这个对象并不确定，我们就可以使用 `extends` 和 `keyof` 进行约束。

```typescript
function getValue<T, K extends keyof T>(obj: T, key: K) {
  return obj[key]
}

const obj = { a: 1 }
const a = getValue(obj, 'a')
```

3.2.6. 映射类型

通过 `in` 关键字做类型的映射，遍历已有接口的 `key` 或者是遍历联合类型，如下例子：

```typescript
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

interface Obj {
  a: string
  b: string
}

type ReadOnlyObj = Readonly<Obj>
```

上述的结构，可以分成这些步骤：

- keyof T：通过类型索引 keyof 的得到联合类型 'a' | 'b'
- P in keyof T 等同于 p in 'a' | 'b'，相当于执行了一次 forEach 的逻辑，遍历 'a' | 'b'

所以最终 `ReadOnlyObj` 的接口为下述：

```typescript
interface ReadOnlyObj {
    readonly a: string;
    readonly b: string;
}
```

3.2.7. 条件类型

条件类型的语法规则和三元表达式一致，经常用于一些类型不确定的情况。

```typescript
T extends U ? X : Y
```

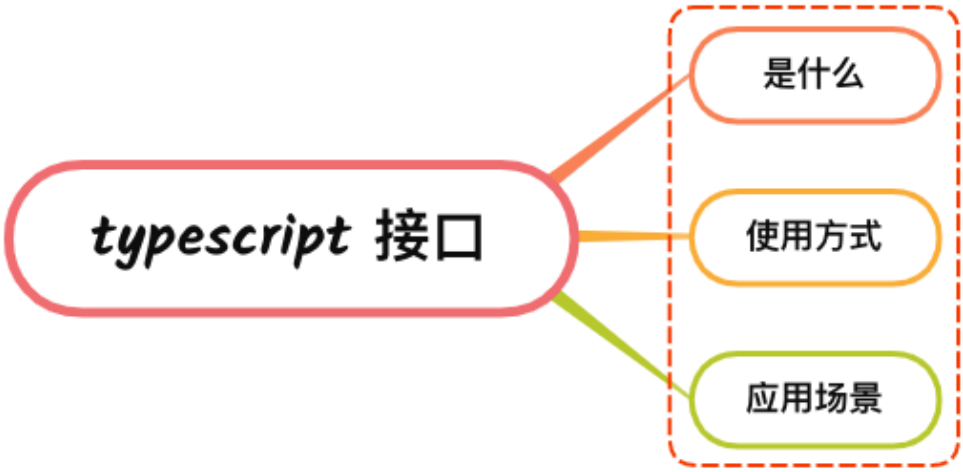
上面的意思就是，如果 T 是 U 的子集，就是类型 X，否则为类型 Y

3.3. 总结

可以看到，如果只是掌握了 `typeScript` 的一些基础类型，可能很难游刃有余的去使用 `typeScript`，需要了解一些 `typescript` 的高阶用法

并且 `typescript` 在版本的迭代中新增了很多功能，需要不断学习与掌握

4. 说说你对 TypeScript 中接口的理解？应用场景？



4.1. 是什么

接口是一系列抽象方法的声明，是一些方法特征的集合，这些方法都应该是抽象的，需要由具体的**类**去实现，然后第三方就可以通过这组抽象方法调用，让具体的类执行具体的方法

简单来讲，一个接口所描述的是一个对象相关的属性和方法，但并不提供具体创建此对象实例的方法

`typescript` 的核心功能之一就是对类型做检测，虽然这种检测方式是“鸭式辨型法”，而接口的作用就是为为这些类型命名和为你的代码或第三方代码定义一个约定

4.2. 使用方式

接口定义如下

```TypeScript
interface interface_name {
}
```

例如有一个函数，这个函数接受一个 `User` 对象，然后返回这个 `User` 对象的 `name` 属性:

```TypeScript
const getUserName = (user) => user.name
```

可以看到，参数需要有一个 `user` 的 `name` 属性，可以通过接口描述 `user` 参数的结构

```TypeScript
interface User {
    name: string
    age: number
}

const getUserName = (user: User) => user.name
```

这些属性并不一定全部实现，上述传入的对象必须拥有 `name` 和 `age` 属性，否则 `typescript` 在编译阶段会报错，如下图:

如果不想要 `age` 属性的话，这时候可以采用**可选属性**，如下表示:

```typescript
interface User {
    name: string
    age?: number
}
```

这时候 `age` 属性则可以是 `number` 类型或者 `undefined` 类型

有些时候，我们想要一个属性变成只读属性，在 `typescript` 只需要使用 `readonly` 声明，如下：

```typescript
interface User {
    name: string
    age?: number
    readonly isMale: boolean
}
```

当我们修改属性的时候，就会出现警告，如下所示：

这是属性中有一个函数，可以如下表示：

```typescript
interface User {
    name: string
    age?: number
    readonly isMale: boolean
    say: (words: string) => string
}
```

如果传递的对象不仅仅是上述的属性，这时候可以使用：

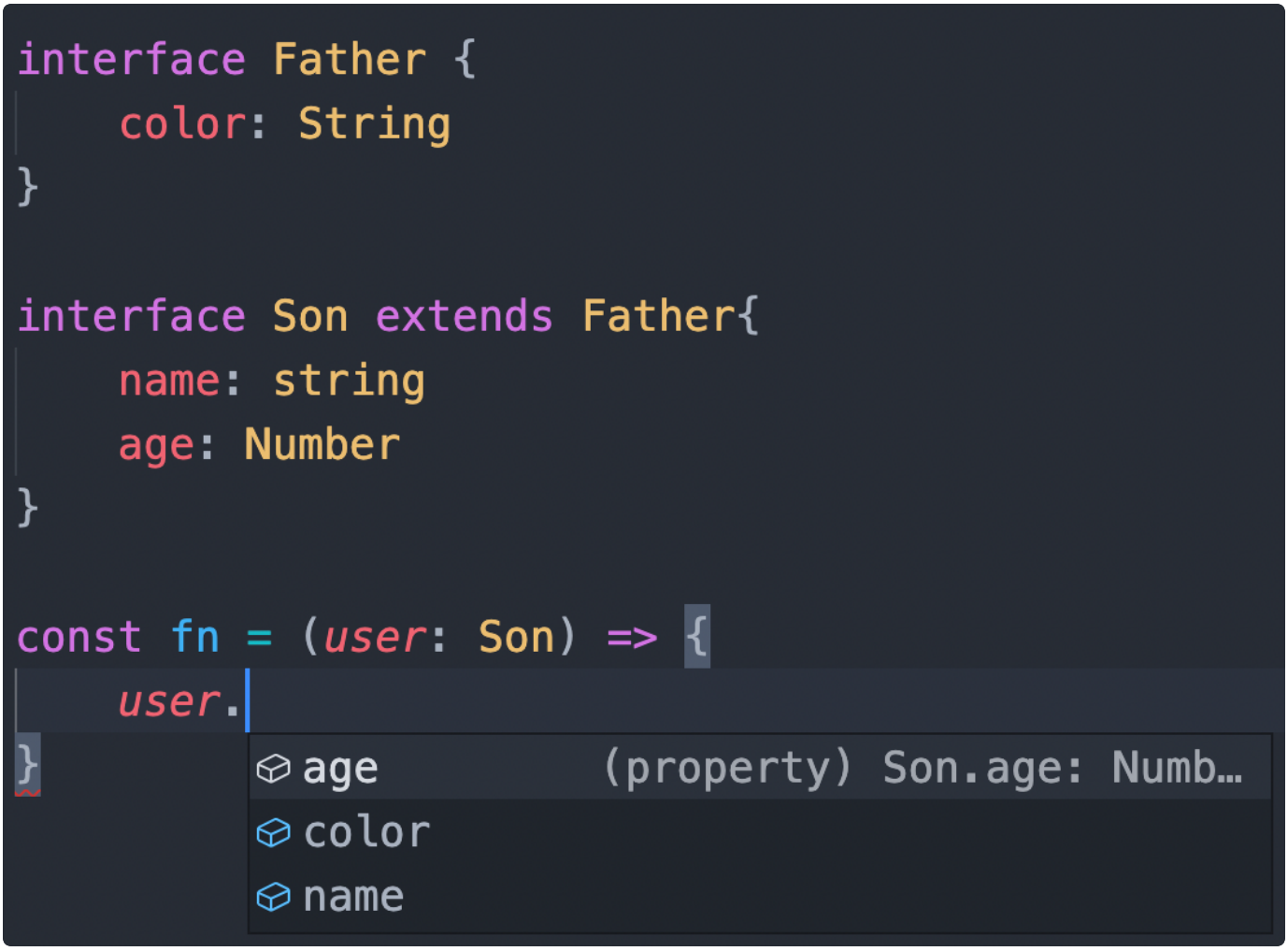
- 类型推断

```plain
interface User {
    name: string
    age: number
}

const getUserName = (user: User) => user.name
getUserName({color: 'yellow'} as User)
```

- 给接口添加字符串**索引签名**

```typescript
interface User {
    name: string
    age: number
    [propName: string]: any;
}
```

接口还能实现继承，如下图：



也可以继承多个，父类通过逗号隔开，如下：

TypeScript | 复制代码

```typescript
interface Father {
    color: String
}

interface Mother {
    height: Number
}

interface Son extends Father,Mother{
    name: string
    age: Number
}
```

4.3. 应用场景

例如在 `javascript` 中定义一个函数，用来获取用户的姓名和年龄：

JavaScript | 复制代码

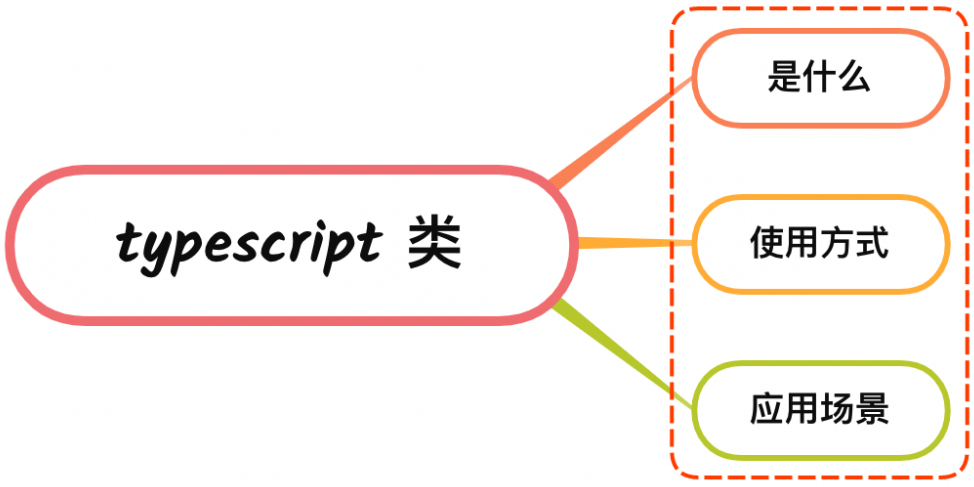
```javascript
const getUserInfo = function(user) {
    // ...
    return name: ${user.name}, age: ${user.age}
}
```

如果多人开发的都需要用到这个函数的时候，如果没有注释，则可能出现各种运行时的错误，这时候就可以使用接口定义参数变量：

TypeScript | 复制代码

```typescript
// 先定义一个接口
interface IUser {
  name: string;
  age: number;
}

const getUserInfo = (user: IUser): string => {
  return `name: ${user.name}, age: ${user.age}`;
};

// 正确的调用
getUserInfo({name: "koala", age: 18});
```

5. 说说你对 TypeScript 中类的理解？应用场景？



5.1. 是什么

类（Class）是面向对象程序设计（OOP，Object-Oriented Programming）实现信息封装的基础

> 类是一种用户定义的引用数据类型，也称类类型

传统的面向对象语言基本都是基于类的， `JavaScript` 基于原型的方式让开发者多了很多理解成本

在 `ES6` 之后， `JavaScript` 拥有了 `class` 关键字，虽然本质依然是构造函数，但是使用起来已经方便了许多

但是 `JavaScript` 的 `class` 依然有一些特性还没有加入，比如修饰符和抽象类

`TypeScript` 的 `class` 支持面向对象的所有特性，比如 类、接口等

5.2. 使用方式

定义类的关键字为 `class` ，后面紧跟类名，类可以包含以下几个模块（类的数据成员）：

- **字段** ： 字段是类里面声明的变量。字段表示对象的有关数据。
- **构造函数**： 类实例化时调用，可以为类的对象分配内存。
- **方法**： 方法为对象要执行的操作

如下例子：

TypeScript 复制代码

```typescript
class Car {
    // 字段
    engine:string;

    // 构造函数
    constructor(engine:string) {
        this.engine = engine
    }

    // 方法
    disp():void {
        console.log("发动机为 :   "+this.engine)
    }
}
```

5.2.1. 继承

类的继承使用过 `extends` 的关键字

```typescript
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

`Dog` 是一个 派生类，它派生自 `Animal` 基类，派生类通常被称作子类，基类通常被称作 超类

`Dog` 类继承了 `Animal` 类，因此实例 `dog` 也能够使用 `Animal` 类 `move` 方法

同样，类继承后，子类可以对父类的方法重新定义，这个过程称之为方法的重写，通过 `super` 关键字是对父类的直接引用，该关键字可以引用父类的属性和方法，如下：

```typescript
class PrinterClass {
   doPrint():void {
      console.log("父类的 doPrint() 方法。")
   }
}

class StringPrinter extends PrinterClass {
   doPrint():void {
      super.doPrint() // 调用父类的函数
      console.log("子类的 doPrint()方法。")
   }
}
```

5.2.2. 修饰符

可以看到，上述的形式跟 `ES6` 十分的相似， `typescript` 在此基础上添加了三种修饰符：

- 公共 public：可以自由的访问类程序里定义的成员
- 私有 private：只能够在该类的内部进行访问
- 受保护 protect：除了在该类的内部可以访问，还可以在子类中仍然可以访问

5.2.3. 私有修饰符

只能够在该类的内部进行访问，实例对象并不能够访问

```
class Father {
    private name: String
    constructor(name: String) {
        this.name = name
    }
}

const father = new Father('huihui')

        属性“name”为私有属性，只能在类“Father”中访问。 ts(2341)
        (property) Father.name: String
        查看问题 (⌥F8)    没有可用的快速修复
father.name
```

并且继承该类的子类并不能访问，如下图所示：

```
class Father {
    private name: String
    constructor(name: String) {
        this.name = name
    }
}

class Son extends Father{
    say() {
        console.log(`my name is ${this.name}`)
    }
}
```

属性“name”为私有属性，只能在类“Father”中访问。 ts(2341)
(property) Father.name: String
查看问题 (⌥F8)　没有可用的快速修复

5.2.4. 受保护修饰符

跟私有修饰符很相似，实例对象同样不能访问受保护的属性，如下：

有一点不同的是 `protected` 成员在子类中仍然可以访问

```
class Father {
    protected name: String
    constructor(name: String) {
        this.name = name
    }
}

class Son extends Father{
    say() {
        console.log(`my name is ${this.name}`)
    }
}
```

除了上述修饰符之外，还有只读**修饰符**

5.2.4.1. 只读修饰符

通过 `readonly` 关键字进行声明，只读属性必须在声明时或构造函数里被初始化，如下：

除了实例属性之外，同样存在静态属性

5.2.5. 静态属性

这些属性存在于类本身上面而不是类的实例上，通过 `static` 进行定义，访问这些属性需要通过 类型.静态属性 的这种形式访问，如下所示：

TypeScript | 复制代码

```typescript
class Square {
    static width = '100px'
}

console.log(Square.width) // 100px
```

上述的类都能发现一个特点就是，都能够被实例化，在 `typescript` 中，还存在一种抽象类

5.2.6. 抽象类

抽象类做为其它派生类的基类使用，它们一般不会直接被实例化，不同于接口，抽象类可以包含成员的实现细节

`abstract` 关键字是用于定义抽象类和在抽象类内部定义抽象方法，如下所示：

```typescript
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log('roaming the earch...');
    }
}
```

这种类并不能被实例化，通常需要我们创建子类去继承，如下：

```typescript
class Cat extends Animal {

    makeSound() {
        console.log('miao miao')
    }
}

const cat = new Cat()

cat.makeSound() // miao miao
cat.move() // roaming the earch...
```

5.3. 应用场景

除了日常借助类的特性完成日常业务代码，还可以将类（class）也可以作为接口，尤其在 `React` 工程中是很常用的，如下：

```typescript
export default class Carousel extends React.Component<Props, State> {}
```

由于组件需要传入 `props` 的类型 `Props` ，同时有需要设置默认 `props` 即 `defaultProps` ，这时候更加适合使用 `class` 作为接口

先声明一个类，这个类包含组件 `props` 所需的类型和初始值：

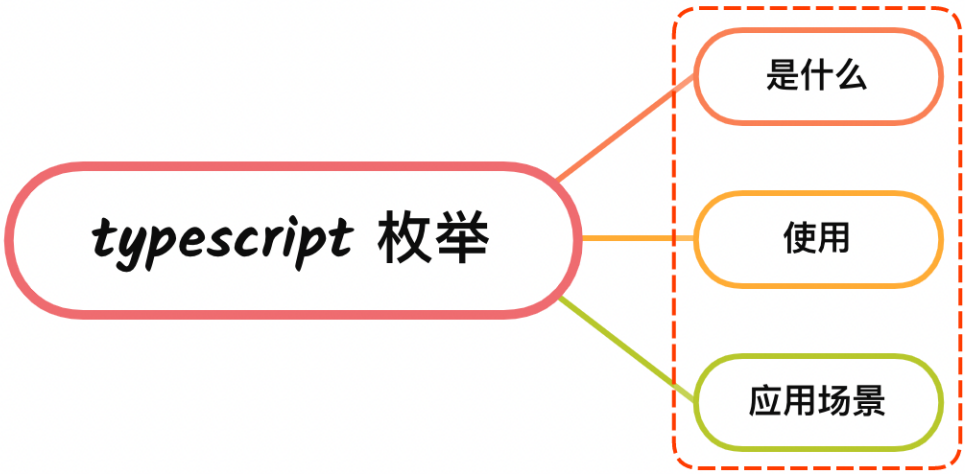
```typescript
// props的类型
export default class Props {
  public children: Array<React.ReactElement<any>> | React.ReactElement<any> | never[] = []
  public speed: number = 500
  public height: number = 160
  public animation: string = 'easeInOutQuad'
  public isAuto: boolean = true
  public autoPlayInterval: number = 4500
  public afterChange: () => {}
  public beforeChange: () => {}
  public selesctedColor: string
  public showDots: boolean = true
}
```

当我们需要传入 `props` 类型的时候直接将 `Props` 作为接口传入，此时 `Props` 的作用就是接口，而当需要我们设置 `defaultProps` 初始值的时候，我们只需要:

```typescript
public static defaultProps = new Props()
```

`Props` 的实例就是 `defaultProps` 的初始值，这就是 `class` 作为接口的实际应用，我们用一个 `class` 起到了接口和设置初始值两个作用，方便统一管理，减少了代码量

6. 说说你对 TypeScript 中枚举类型的理解？应用场景?



6.1. 是什么

枚举是一个被命名的整型常数的集合，用于声明一组命名的常数,当一个变量有几种可能的取值时,可以将它定义为枚举类型

通俗来说，枚举就是一个对象的所有可能取值的集合

在日常生活中也很常见，例如表示星期的SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY就可以看成是一个枚举

枚举的说明与结构和联合相似，其形式为：

Plain Text | 复制代码

```
enum 枚举名{
    标识符①[=整型常数],
    标识符②[=整型常数],
    ...
    标识符N[=整型常数],
}枚举变量;
```

6.2. 使用

枚举的使用是通过 `enum` 关键字进行定义，形式如下：

TypeScript | 复制代码

```
enum xxx { ... }
```

声明关键字为枚举类型的方式如下：

```typescript
// 声明d为枚举类型Direction
let d: Direction;
```

类型可以分成：

- 数字枚举
- 字符串枚举
- 异构枚举

6.2.1. 数字枚举

当我们声明一个枚举类型是,虽然没有给它们赋值,但是它们的值其实是默认的数字类型,而且默认从0开始依次累加:

```typescript
enum Direction {
    Up,   // 值默认为 0
    Down, // 值默认为 1
    Left, // 值默认为 2
    Right // 值默认为 3
}

console.log(Direction.Up === 0); // true
console.log(Direction.Down === 1); // true
console.log(Direction.Left === 2); // true
console.log(Direction.Right === 3); // true
```

如果我们将第一个值进行赋值后，后面的值也会根据前一个值进行累加1：

```typescript
enum Direction {
    Up = 10,
    Down,
    Left,
    Right
}

console.log(Direction.Up, Direction.Down, Direction.Left, Direction.Right);
 // 10 11 12 13
```

6.2.2. 字符串枚举

TypeScript | 复制代码

```
枚举类型的值其实也可以是字符串类型：

enum Direction {
    Up = 'Up',
    Down = 'Down',
    Left = 'Left',
    Right = 'Right'
}

console.log(Direction['Right'], Direction.Up); // Right Up
```

如果设定了一个变量为字符串之后，后续的字段也需要赋值字符串，否则报错：

TypeScript | 复制代码

```
enum Direction {
  Up = 'UP',
  Down, // error TS1061: Enum member must have initializer
  Left, // error TS1061: Enum member must have initializer
  Right // error TS1061: Enum member must have initializer
}
```

6.2.3. 异构枚举

即将数字枚举和字符串枚举结合起来混合起来使用，如下：

TypeScript | 复制代码

```
enum BooleanLikeHeterogeneousEnum {
    No = 0,
    Yes = "YES",
}
```

通常情况下我们很少会使用异构枚举

6.2.4. 本质

现在一个枚举的案例如下：

```typescript
enum Direction {
    Up,
    Down,
    Left,
    Right
}
```

通过编译后，`javascript` 如下：

```typescript
var Direction;
(function (Direction) {
    Direction[Direction["Up"] = 0] = "Up";
    Direction[Direction["Down"] = 1] = "Down";
    Direction[Direction["Left"] = 2] = "Left";
    Direction[Direction["Right"] = 3] = "Right";
})(Direction || (Direction = {}));
```

上述代码可以看到，`Direction[Direction["Up"] = 0] = "Up"` 可以分成

- Direction["Up"] = 0
- Direction[0] = "Up"

所以定义枚举类型后，可以通过正反映射拿到对应的值，如下：

```typescript
enum Direction {
    Up,
    Down,
    Left,
    Right
}

console.log(Direction.Up === 0); // true
console.log(Direction[0]); // Up
```

并且多处定义的枚举是可以进行合并操作，如下：

```typescript
enum Direction {
    Up = 'Up',
    Down = 'Down',
    Left = 'Left',
    Right = 'Right'
}

enum Direction {
    Center = 1
}
```

编译后，`js` 代码如下：

```javascript
var Direction;
(function (Direction) {
    Direction["Up"] = "Up";
    Direction["Down"] = "Down";
    Direction["Left"] = "Left";
    Direction["Right"] = "Right";
})(Direction || (Direction = {}));
(function (Direction) {
    Direction[Direction["Center"] = 1] = "Center";
})(Direction || (Direction = {}));
```

可以看到，`Direction` 对象属性回叠加

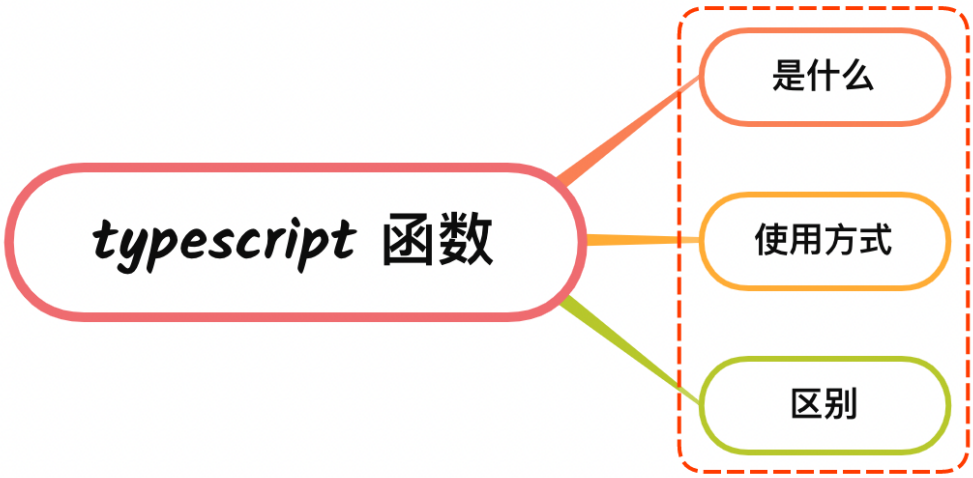
6.3. 应用场景

就拿回生活的例子，后端返回的字段使用 0 – 6 标记对应的日期，这时候就可以使用枚举可提高代码可读性，如下：

```typescript
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 0); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true
```

包括后端日常返回0、1 等等状态的时候，我们都可以通过枚举去定义，这样可以提高代码的可读性，便于后续的维护

7. 说说你对 TypeScript 中函数的理解？与 JavaScript 函数的区别？



7.1. 是什么

函数是 `JavaScript` 应用程序的基础，帮助我们实现抽象层、模拟类、信息隐藏和模块

在 `TypeScript` 里，虽然已经支持类、命名空间和模块，但函数仍然是主要定义行为的方式，`TypeScript` 为 `JavaScript` 函数添加了额外的功能，丰富了更多的应用场景

函数类型在 `TypeScript` 类型系统中扮演着非常重要的角色，它们是可组合系统的核心构建块

7.2. 使用方式

跟 `javascript` 定义函数十分相似，可以通过 `funciton` 关键字、箭头函数等形式去定义，例如下面一个简单的加法函数：

```TypeScript
const add = (a: number, b: number) => a + b
```

上述只定义了函数的两个参数类型，这个时候整个函数虽然没有被显式定义，但是实际上 `TypeScript` 编译器是能够通过类型推断到这个函数的类型，如下图所示：

```
1
2           const add: (a: number, b: number) => number
3   const add = (a: number, b: number) => a + b
```

当鼠标放置在第三行 `add` 函数名的时候，会出现完整的函数定义类型，通过 `:` 的形式来定于参数类型，通过 `=>` 连接参数和返回值类型

当我们没有提供函数实现的情况下，有两种声明函数类型的方式，如下所示：

TypeScript | 复制代码

```typescript
// 方式一
type LongHand = {
  (a: number): number;
};

// 方式二
type ShortHand = (a: number) => number;
```

当存在函数重载时，只能使用方式一的形式

7.2.1. 可选参数

当函数的参数可能是不存在的，只需要在参数后面加上 `?` 代表参数可能不存在，如下：

TypeScript | 复制代码

```typescript
const add = (a: number, b?: number) => a + (b ? b : 0)
```

这时候参数 `b` 可以是 `number` 类型或者 `undefined` 类型，即可以传一个 `number` 类型或者不传都可以

7.2.2. 剩余类型

剩余参数与 `JavaScript` 的语法类似，需要用 `...` 来表示剩余参数

如果剩余参数 `rest` 是一个由 `number` 类型组成的数组，则如下表示：

TypeScript | 复制代码

```typescript
const add = (a: number, ...rest: number[]) => rest.reduce(((a, b) => a + b)
, a)
```

7.2.3. 函数重载

允许创建数项名称相同但输入输出类型或个数不同的子程序，它可以简单地称为一个单独功能可以执行多项任务的能力

关于 `typescript` 函数重载，必须要把精确的定义放在前面，最后函数实现时，需要使用 `|` 操作符或者 `?` 操作符，把所有可能的输入类型全部包含进去，用于具体实现

这里的函数重载也只是多个函数的声明，具体的逻辑还需要自己去写， `typescript` 并不会真的将你的多个重名 `function` 的函数体进行合并

例如我们有一个add函数，它可以接收 `string` 类型的参数进行拼接，也可以接收 `number` 类型的参数进行相加，如下：

TypeScript | 复制代码

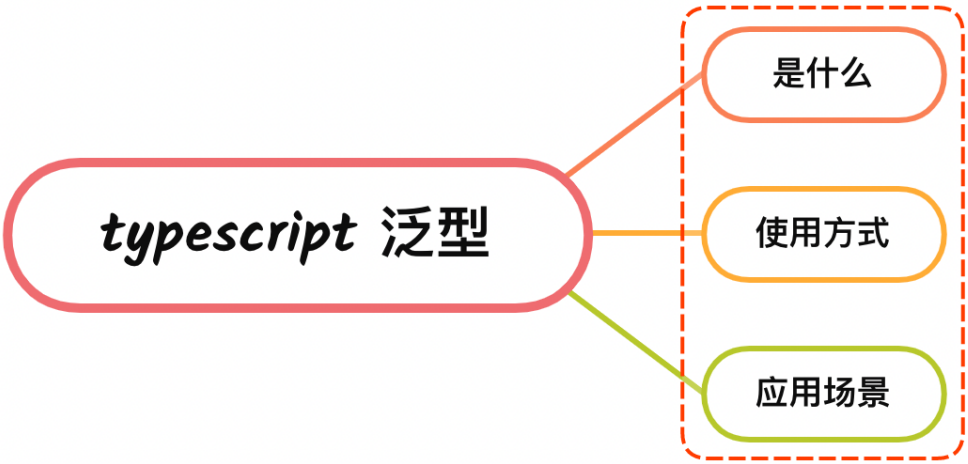
```typescript
// 上边是声明
function add (arg1: string, arg2: string): string
function add (arg1: number, arg2: number): number
// 因为我们在下边有具体函数的实现，所以这里并不需要添加 declare 关键字

// 下边是实现
function add (arg1: string | number, arg2: string | number) {
  // 在实现上我们要注意严格判断两个参数的类型是否相等，而不能简单的写一个 arg1 + arg2
  if (typeof arg1 === 'string' && typeof arg2 === 'string') {
    return arg1 + arg2
  } else if (typeof arg1 === 'number' && typeof arg2 === 'number') {
    return arg1 + arg2
  }
}
```

7.3. 区别

从上面可以看到：

- 从定义的方式而言，typescript 声明函数需要定义参数类型或者声明返回值类型
- typescript 在参数中，添加可选参数供使用者选择
- typescript 增添函数重载功能，使用者只需要通过查看函数声明的方式，即可知道函数传递的参数个数以及类型

8. 说说你对 TypeScript 中泛型的理解？应用场景？



8.1. 是什么

泛型程序设计（generic programming）是程序设计语言的一种风格或范式

泛型允许我们在强类型程序设计语言中编写代码时使用一些以后才指定的类型，在实例化时作为参数指明这些类型

在 `typescript` 中，定义函数，接口或者类的时候，不预先定义好具体的类型，而在使用的时候在指定类型的一种特性

假设我们用一个函数，它可接受一个 `number` 参数并返回一个 `number` 参数，如下写法：

TypeScript | 复制代码

```typescript
function returnItem (para: number): number {
    return para
}
```

如果我们打算接受一个 `string` 类型，然后再返回 `string` 类型，则如下写法：

TypeScript | 复制代码

```typescript
function returnItem (para: string): string {
    return para
}
```

上述两种编写方式，存在一个最明显的问题在于，代码重复度比较高

虽然可以使用 `any` 类型去替代，但这也并不是很好的方案，因为我们的目的是接收什么类型的参数返回什么类型的参数，即在运行时传入参数我们才能确定类型

这种情况就可以使用泛型，如下所示：