TypeScript 全解

饥人谷出品

学习建议

看 思路

敲 代码

記 重点

基础好者看一次视频

基础弱就多看一两次

每节课写一篇博客

课程代码放在单独目录

VS Code一次只打开一个目录

课后问答群

TypeScript全解课后问答群

如果加不进去
直接加我微信 frank_fang
暗号：ts

正式内容

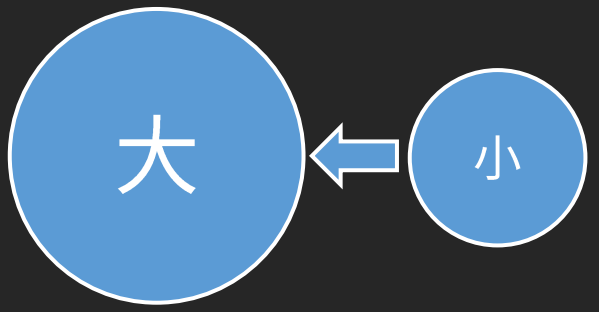
JS datatype

null undefined
string number boolean
bigint symbol
object（含 Array、Function、Date...）

TS datatype

以上所有，加上
void、never、enum、unknown、any
再加上自定义类型 type、interface

联合类型
交叉类型



a:A=b 类型兼容

深入对象和函数

我还是用 any 吧

高级

Generic Types
泛型

看不懂

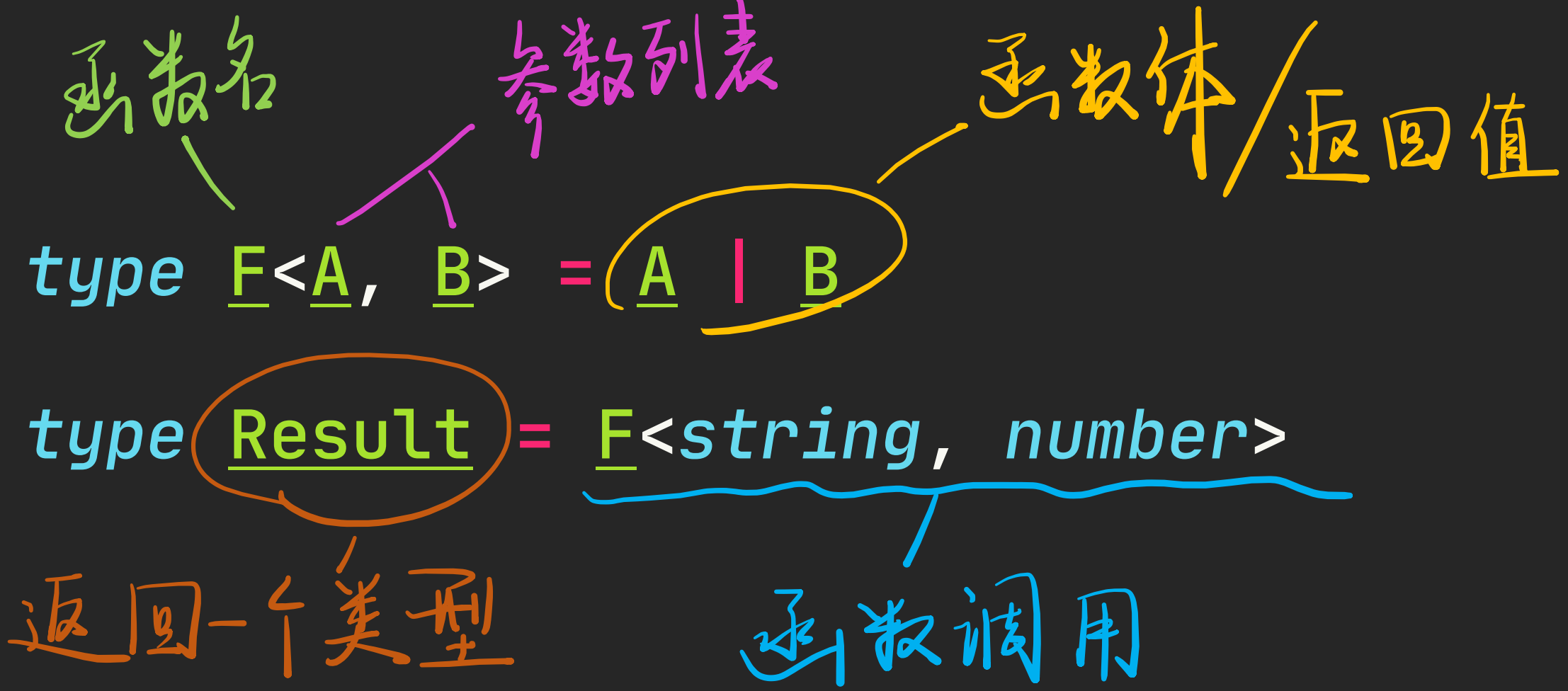
体操

```ts
const f = (a, b) => a + b
const result = f(1, 2)
//    ^--- 3
```

如果你能看懂JS的函数

那么你就能看懂TS的泛型

```ts
type F<A, B> = A | B
type Result = F<string, number>
//    ^---- string | number
```


函数名
参数列表
函数体/返回值
type F<A, B> = A | B
type Result = F<string, number>
返回一个类型
函数调用

思考题

函数的本质是什么？

函数的本质

推后执行的、部分待定的代码

```
// 立即执行
console.log(1)

// 推后一步执行
const f1 = () => console.log(1)
f1()
```

```
// 推后两步执行
const f2 = () => console.log(1)
console.log(2)
f2()
```

```
// 部分待定的
const fn = (n) => console.log(n) // 参数待定
fn(100)
const ff = (f) => f(1) // 函数待定
```

推后执行的部分待定的代码

举一反三

ZZ型的本质是什么？

泛型的本质

推后执行的、部分待定的~~代码~~类型

为什么需要这些？

```ts
function echo(whatever: number | string | boolean) {
  return whatever
}
```

参数类型 `number | string | boolean`

返回值类型 `number | string | boolean`

`number` `string` `boolean`

`number` `string` `boolean`

```
function echo(whatever: number | string | boolean) {
  switch (typeof whatever) {
    case 'number':
      return whatever
      break;
    case 'string':
      return whatever
      break;
    case 'boolean':
      return whatever
      break;
  }
}
```

没有泛型

有些奇怪的需求就无法满足

没有泛型的类型系统

就如同没有函数的编程语言

接下来的内容

很烧脑

如有不懂

如何提问

群里提问

Ready? Go!

难度 

```ts
type Union<A, B> = A | B
type Union3<A, B, C> = A | B | C

type Intersect<A, B> = A & B
type Intersect3<A, B, C> = A & B & C

interface List<A> {
  [index: number]: A
}
interface Hash<V> {
  [key: string]: V
}
```

难度 

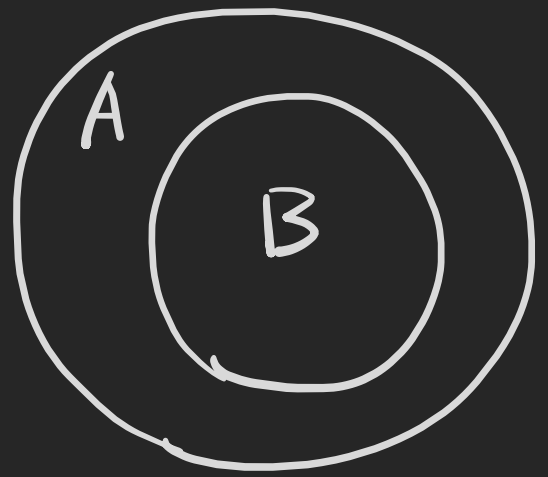
```ts
type Person = { name: string }

type LikeString<T> = ???
type LikeNumber<T> = ???
type LikePerson<T> = ???

type R1 = LikeString<'hi'> // true
type R2 = LikeString<true> // false
type S1 = LikeNumber<6666> // 1
type S2 = LikeNumber<false> // 2
type T1 = LikePerson<{ name: 'frank', xxx: 1 }>
// yes
type T2 = LikePerson<{ xxx: 1 }> // no
```

```ts
type LikeString<T> = T extends string ? true : false
type LikeNumber<T> = T extends number ? 1 : 2
type LikePerson<T> = T extends Person ? 'yes' : 'no'
```

extends 读作 继承 或 包含于



B extends A ✓

A extends A ✓

规则 ① 若T为 never，则表达式的值为 never *

② 若T为联合类型，则为分开计算 *

Conditional Types

条件类型

```ts
type ToArray<T> = T extends unknown ? T[] : never

type Result = ToArray<string | number>
```

请选择：

① Result 为 (string | number)[] 类型

② Result 为 string[] | number[] 类型

```ts
type ToArray<T> = T extends unknown ? T[] : never

type Result = ToArray<string | number>
// type Result = (string | number) extends unknown ? ...
// type Result = (string extends unknown ? ...)
//             | (number extends unknown ? ...)
// type Result = string[] | number[]
```

请选择：

① Result 为 (string | number)[] 类型

② Result 为 string[] | number[] 类型

```ts
type ToArray<T> = T extends unknown ? T[] : never

type Result = ToArray<never>
```

请选择：

① Result 为 never[] 类型

② Result 为 never 类型

```ts
type ToArray<T> = T extends unknown ? T[] : never

type Result = ToArray<never>
// type Result = never extends unknown ? ...
// type Result = 空集没有元素直接返回 never
// type Result = never
```

请选择：

① Result 为 never[] 类型

② Result 为 never 类型

好有一比 (string | number) extends ?

乘法的分配律 (A+B)XC=AXC+BXC

乘法中的零 0XC=0

注意：只对泛型有效

难度 

```ts
type Person = { name: string, age: number }
type GetKeys<T> = ???
type Result = GetKeys<Person>
//   ^-- 'name' | 'age'
```

```ts
type Person = { name: string, age: number }
type GetKeys<T> = keyof T
type Result = GetKeys<Person>
//   ^-- 'name' | 'age'
```

难度 

```ts
type Person = { name: string, age: number }
type GetKeyType<T, K ???> = ???
type Result1 = GetKeyType<Person, 'name'>
//   ^-- string
type Result2 = GetKeyType<Person, 'age'>
//   ^-- number
```

```ts
type Person = { name: string, age: number }
type GetKeyType<T, K extends keyof T> = T[K]
type Result1 = GetKeyType<Person, 'name'>
//   ^-- string
type Result2 = GetKeyType<Person, 'age'>
//   ^-- number
```

难度 

```ts
type X1 = Readonly<Person>
type X2 = Partial<Person>
type X3 = Required<Person>
type X4 = Record<string, number>
type X5 = Exclude<1 | 2 | 3, 1 | 2> // 3
type X6 = Extract<1 | 2 | 3, 2 | 4> // 2
type X7 = Omit<Person, 'name' | 'age'>
type X8 = Pick<Person, 'age'>

type Omit<T, Key> = {
  [K2 in keyof T as (K2 extends Key ? never : K2)]: T
}
```

```ts
type Mutable<Type> = {
  ??????? [Property in keyof Type]: Type[Property];
};
type Y = Mutable<Readonly<Person>>
```

```ts
type Mutable<Type> = {
  -readonly [Property in keyof Type]: Type[Property];
};
type X9 = Mutable<Readonly<Person>>
```

回顾

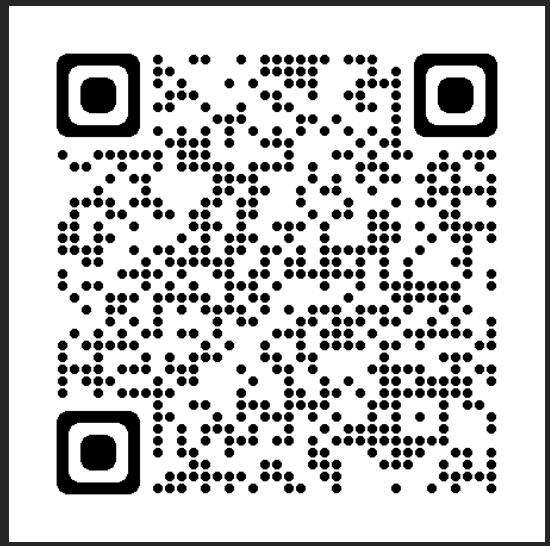
从类型中创造类型

- extends
- keyof
- T['name']
- Mapped Types
  - in keyof
  - -readonly[]
  - []-?
  - as

如何学习泛型

① 别用 any

② 做题



③ 与人交流

这就是美型体操吗？

体操比这难多了！

扫码失败请自行搜索

问与答