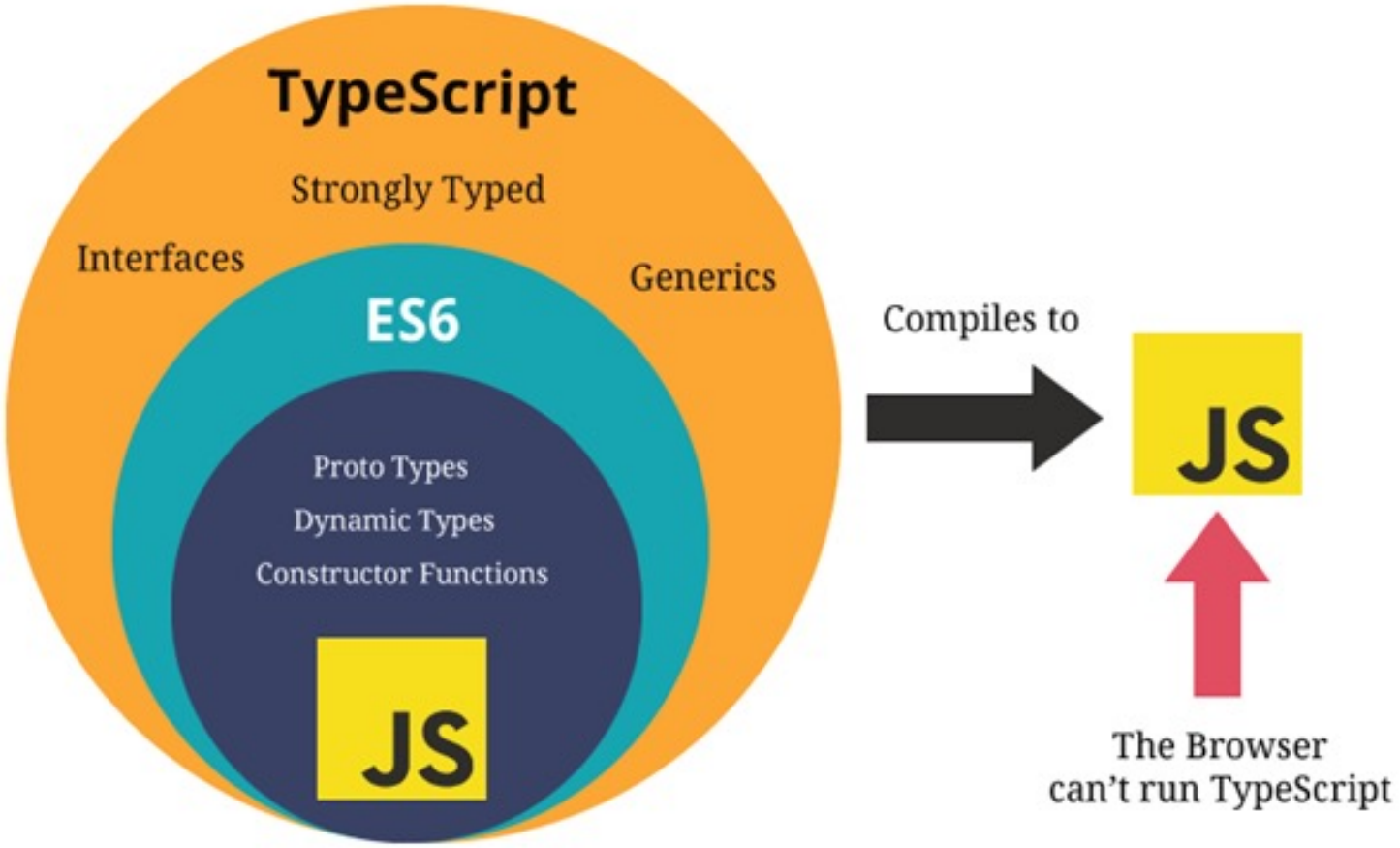
TypeScript语法精讲（二）

王红元 coderwhy

JavaScript和TypeScript的数据类型

- 我们经常说TypeScript是JavaScript的一个超级：



JavaScript类型 – number类型

- 数字类型是我们开发中经常使用的类型，TypeScript和JavaScript一样，不区分整数类型（int）和浮点型（double），统一为number类型。

```
let num = 100;
num = 20;
num = 6.66;
```

- 如果你学习过ES6应该知道，ES6新增了二进制和八进制的表示方法，而TypeScript也是支持二进制、八进制、十六进制的表示：

```
// 2.其他进制表示
num = 100; // 十进制
num = 0b110; // 二进制
num = 0o555; // 八进制
num = 0xf23; // 十六进制
```

JavaScript类型 – boolean类型

- boolean类型只有两个取值：true和false，非常简单

```
// boolean类型的表示
let flag: boolean = true;
flag = false;
flag = 20 > 30;
```

JavaScript类型 – string类型

- string类型是字符串类型，可以使用单引号或者双引号表示：

```
// string类型表示
let message: string = "Hello World";
message = 'Hello TypeScript';
```

- 同时也支持ES6的模板字符串来拼接变量和字符串：

```
const name = "why";
const age = 18;
const height = 1.88;

const info = `my name is ${name}, age is ${age}, height is ${height}`;
console.log(info);
```

JavaScript类型 – Array类型

- 数组类型的定义也非常简单，有两种方式：

```
const names: string[] = ["abc", "cba", "cba"]
const names2: Array<string> = ["abc", "cba", "nba"]

names.push("why")
names2.push("why")
```

- 如果添加其他类型到数组中，那么会报错：

```
names.push( Argument of type 'number' is not assignable to parameter of type
names2.push 'string'. ts(2345)

View Problem (⌥F8)    No quick fixes available

names.push(123)
names2.push(123)
```

JavaScript类型 – Object类型

- object对象类型可以用于描述一个对象：

```
const myInfo: object = {
  name: "why",
  age: 18,
  height: 1.88
}
```

- 但是从myinfo中我们不能获取数据，也不能设置数据：

```
myInfo["name"] = "coderwhy"
console.log(myInfo["age"])
```

```
01_JavaScript数据类型.ts:16:1 - error TS7053: Element implicitly has an 'any' type because expression of type '"name"' can't be
used to index type '{}'.
  Property 'name' does not exist on type '{}'.

16 myInfo["name"] = "coderwhy"
   ~~~~~~~~~~~~~~

01_JavaScript数据类型.ts:17:13 - error TS7053: Element implicitly has an 'any' type because expression of type '"age"' can't be
used to index type '{}'.
  Property 'age' does not exist on type '{}'.

17 console.log(myInfo["age"])
               ~~~~~~~~~~~~~
```

JavaScript类型 – Symbol类型

- 在ES5中，如果我们是不可以在对象中添加相同的属性名称的，比如下面的做法：

```
const person = {
  identity: "程序员",
  identity: "老师",
}
```

- 通常我们的做法是定义两个不同的属性名字：比如identity1和identity2。
- 但是我们也可以通过symbol来定义相同的名称，因为Symbol函数返回的是不同的值：

```
const s1: symbol = Symbol("title")
const s2: symbol = Symbol("title")

const person = {
  [s1]: "程序员",
  [s2]: "老师"
}
```

JavaScript类型 – null和undefined类型

- 在 JavaScript 中，undefined 和 null 是两个基本数据类型。
- 在TypeScript中，它们各自的类型也是undefined和null，也就意味着它们既是实际的值，也是自己的类型：

```
let n: null = null
let u: undefined = undefined
```

TypeScript类型 - any类型

- 在某些情况下，我们确实无法确定一个变量的类型，并且可能它会发生一些变化，这个时候我们可以使用any类型（类似于Dart语言中的dynamic类型）。
- any类型有点像一种讨巧的TypeScript手段：
  - 我们可以对any类型的变量进行任何的操作，包括获取不存在的属性、方法；
  - 我们给一个any类型的变量赋值任何的值，比如数字、字符串的值；

```
let a: any = "why";
a = 123;
a = true;

const aArray: any[] = ["why", 18, 1.88];
```

- 如果对于某些情况的处理过于繁琐不希望添加规定的类型注解，或者在引入一些第三方库时，缺失了类型注解，这个时候我们可以使用any：
  - 包括在Vue源码中，也会使用到any来进行某些类型的适配；

TypeScript类型 - unknown类型

- unknown是TypeScript中比较特殊的一种类型，它用于描述类型不确定的变量。
- 什么意思呢？我们来看下面的场景：

```ts
function foo(): string {
  return 'foo'
}

function bar(): number {
  return 123
}
```

```ts
const flag = true
let result: unknown

if (flag) {
  result = foo()
} else {
  result = bar()
}

if (typeof result === 'string') {
  console.log(result.length)
}

export {}
```

TypeScript类型 - void类型

- void通常用来指定一个函数是没有返回值的，那么它的返回值就是void类型：
  - 我们可以将null和undefined赋值给void类型，也就是函数可以返回null或者undefined

```
function sum(num1: number, num2: number) {
  console.log(num1 + num2)
}
```

- 这个函数我们没有写任何类型，那么它默认返回值的类型就是void的，我们也可以显示的来指定返回值是void：

```
function sum(num1: number, num2: number): void {
  console.log(num1 + num2)
}
```

TypeScript类型 - never类型

- never 表示永远不会发生值的类型，比如一个函数：
  - 如果一个函数中是一个死循环或者抛出一个异常，那么这个函数会返回东西吗？
  - 不会，那么写void类型或者其他类型作为返回值类型都不合适，我们就可以使用never类型；

```ts
function loopFun(): never {
  while(true) {
    console.log("123")
  }
}

function loopErr(): never {
  throw new Error()
}
```

```ts
function handleMessage(message: number|string) {
  switch (typeof message) {
    case 'string':
      console.log('foo')
      break
    case 'number':
      console.log('bar')
      break
    default:
      const check: never = message
  }
}
```

- never有什么样的应用场景呢？这里我们举一个例子，但是它用到了联合类型，后面我们会讲到：

TypeScript类型 - tuple类型

- tuple是元组类型，很多语言中也有这种数据类型，比如Python、Swift等。

```
const tInfo: [string, number, number] = ["why", 18, 1.88];
const item1 = tInfo[0]; // why, 并且知道类型是string类型
const item2 = tInfo[1]; // 18, 并且知道类型是number类型
```

- 那么tuple和数组有什么区别呢？
  - 首先，数组中通常建议存放相同类型的元素，不同类型的元素是不推荐放在数组中。（可以放在对象或者元组中）
  - 其次，元组中每个元素都有自己特性的类型，根据索引值获取到的值可以确定对应的类型；

```
const info: (string|number)[] = ["why", 18, 1.88]
const item1 = info[0] // 不能确定类型

const tInfo: [string, number, number] = ["why", 18, 1.88]
const item2 = tInfo[0] // 一定是string类型
```

Tuple的应用场景

- 那么tuple在什么地方使用的是最多的呢？
  - tuple通常可以作为返回的值，在使用的时候会非常的方便；

```
function useState<T>(state: T): [T, (newState: T) => void] {
  let currentState = state
  const changeState = (newState: T) => {
    currentState = newState
  }

  return [currentState, changeState]
}

const [counter, setCounter] = useState(10)
```

函数的参数类型

- 函数是JavaScript非常重要的组成部分，TypeScript允许我们指定函数的参数和返回值的类型。
- **参数的类型注解**
  - 声明函数时，可以在每个参数后添加类型注解，以声明函数接受的参数类型：

```
function greet(name: string) {
  console.log("Hello " + name.toUpperCase())
}

// Argument of type 'number' is not assignable to parameter of type 'string'
greet(123)

// Expected 1 arguments, but got 2.ts(2554)
greet("abc", "cba")
```

函数的返回值类型

- 我们也可以添加返回值的类型注解，这个注解出现在函数列表的后面：

```
function sum(num1: number, num2: number): number {
  return num1 + num2
}
```

- 和变量的类型注解一样，我们通常情况下不需要返回类型注解，因为TypeScript会根据 return 返回值推断函数的返回类型：
  - 某些第三方库处于方便理解，会明确指定返回类型，但是这个看个人喜好；

匿名函数的参数

- 匿名函数与函数声明会有一些不同：
  - 当一个函数出现在TypeScript可以确定该函数会被如何调用的地方时；
  - 该函数的参数会自动指定类型；

```
const names = ["abc", "cba", "nba"]
names.forEach(item => {
  console.log(item.toUpperCase())
})
```

- 我们并没有指定item的类型，但是item是一个string类型：
  - 这是因为TypeScript会根据forEach函数的类型以及数组的类型推断出item的类型；
  - 这个过程称之为上下文类型（*contextual typing*），因为函数执行的上下文可以帮助确定参数和返回值的类型；

对象类型

- 如果我们希望限定一个函数接受的参数是一个对象，这个时候要如何限定呢？
  - 我们可以使用对象类型；

```
function printCoordinate(point: {x: number, y: number}) {
  console.log("x坐标:", point.x)
  console.log("y坐标:", point.y)
}

printCoordinate({x: 10, y: 30})
```

- 在这里我们使用了一个对象来作为类型：
  - 在对象我们可以添加属性，并且告知TypeScript该属性需要是什么类型；
  - 属性之间可以使用，或者；来分割，最后一个分隔符是可选的；
  - 每个属性的类型部分也是可选的，如果不指定，那么就是any类型；

可选类型

- 对象类型也可以指定哪些属性是可选的，可以在属性的后面添加一个?：

```
function printCoordinate(point: {x: number, y: number, z?: number}) {
  console.log("x坐标:", point.x)
  console.log("y坐标:", point.y)
  if (point.z) {
    console.log("z坐标:", point.z)
  }
}

printCoordinate({x: 10, y: 30})
printCoordinate({x: 20, y: 30, z: 40})
```

联合类型

- TypeScript的类型系统允许我们使用多种运算符，从现有类型中构建新类型。
- 我们来使用第一种组合类型的方法：联合类型（Union Type）
  - 联合类型是由两个或者多个其他类型组成的类型；
  - 表示可以是这些类型中的任何一个值；
  - 联合类型中的每一个类型被称之为联合成员（union's *members*）；

```
function printId(id: number| string) {
  console.log("你的id是:", id)
}

printId(10)
printId("abc")
```

使用联合类型

- 传入给一个联合类型的值是非常简单的：只要保证是联合类型中的某一个类型的值即可
  - 但是我们拿到这个值之后，我们应该如何使用它呢？因为它可能是任何一种类型。
  - 比如我们拿到的值可能是string或者number，我们就不能对其调用string上的一些方法；
- 那么我们怎么处理这样的问题呢？
  - 我们需要使用缩小（narrow）联合（后续我们还会专门讲解缩小相关的功能）；
  - TypeScript可以根据我们缩小的代码结构，推断出更加具体的类型；

```
function printId(id: number| string) {
  if (typeof id === 'string') {
    // 确定id是string类型
    console.log("你的id是:", id.toUpperCase())
  } else {
    // // 确定id是number类型
    console.log("你的id是", id)
  }
}
```

可选类型补充

- 其实上，可选类型可以看做是 类型 和 undefined 的联合类型：

```
function print(message?: string) {
  console.log(message)
}

print()
print("Coderwhy")
print(undefined)

// error: Argument of type 'null' is not assignable to parameter of type 'string | undefined'.
print(null)
```

类型别名

- 在前面，我们通过在类型注解中编写 对象类型 和 联合类型，但是当我们想要多次在其他地方使用时，就要编写多次。
- 比如我们可以给对象类型起一个别名：

```ts
type Point = {
  x: number
  y: number
}

function printPoint(point: Point) {
  console.log(point.x, point.y)
}

function sumPoint(point: Point) {
  console.log(point.x + point.y)
}

printPoint({x: 20, y: 30})
sumPoint({x: 20, y: 30})
```

```ts
type ID = number | string

function printId(id: ID) {
  console.log("您的id:", id)
}
```

类型断言as

- 有时候TypeScript无法获取具体的类型信息，这个我们需要使用类型断言（Type Assertions）。
  - 比如我们通过 document.getElementById，TypeScript只知道该函数会返回 HTMLElement ，但并不知道它具体的类型：

```
const myEl = document.getElementById("my-img") as HTMLImageElement

myEl.src = "图片地址"
```

- TypeScript只允许类型断言转换为 更具体 或者 不太具体 的类型版本，此规则可防止不可能的强制转换：

```
myEl.src = "图
// Conversion
overlaps with
const name = "coderwhy" as number;
```

Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first. ts(2352)

View Problem (⌥F8) Quick Fix... (⌘.)

```
const name = ("coderwhy" as unknown) as number;
```

非空类型断言!

- 当我们编写下面的代码时，在执行ts的编译阶段会报错：
  - 这是因为传入的message有可能是为undefined的，这个时候是不能执行方法的；

```
function printMessage(message?: string) {
  // error TS2532: Object is possibly 'undefined'
  console.log(message.toUpperCase())
}

printMessage("hello")
```

- 但是，我们确定传入的参数是有值的，这个时候我们可以使用非空类型断言：
  - 非空断言使用的是！，表示可以确定某个标识符是有值的，跳过ts在编译阶段对它的检测；

```
function printMessage(message?: string) {
  console.log(message!.toUpperCase())
}
```

可选链的使用

- 可选链事实上并不是TypeScript独有的特性，它是ES11（ES2020）中增加的特性：
  - 可选链使用可选链操作符 ?.；
  - 它的作用是当对象的属性不存在时，会短路，直接返回undefined，如果存在，那么才会继续执行；
  - 虽然可选链操作是ECMAScript提出的特性，但是和TypeScript一起使用更版本；

```ts
type Person = {
  name: string
  friend?: {
    name: string
    age?: number,
    girlFriend?: {
      name: string
    }
  }
}
```

```ts
const info: Person = {
  name: "why",
  friend: {
    name: "kobe",
    girlFriend: {
      name: "lily"
    }
  }
}
```

```ts
console.log(info.friend?.name)
console.log(info.friend?.age)
console.log(info.friend?.girlFriend?.name)
```

??和!!的作用

- 有时候我们还会看到 !! 和 ?? 操作符，这些都是做什么的呢？
- !!操作符：
  - 将一个其他类型转换成boolean类型；
  - 类似于Boolean(变量)的方式；
- ??操作符：
  - 它是ES11增加的新特性；
  - **空值合并操作符（??）是一个逻辑操作符，当操作符的左侧是 null 或者 undefined 时，返回其右侧操作数，否则返回左侧操作数；**

```
const message = ""

let flag1 = Boolean(message)
let flag2 = !!message
```

```
const message = "321"
const result = message ?? "123"

console.log(result)
```

字面量类型

- 除了前面我们所讲过的类型之外，也可以使用字面量类型（literal types）：

```
let message: "Hello World" = "Hello World"
// Type '"你好啊, 李银河"' is not assignable to type '"Hello World"'.
message = "你好啊, 李银河"
```

- 那么这样做有什么意义呢？
  - 默认情况下这么做是没有太大的意义的，但是我们可以将多个类型联合在一起；

```
type Alignment = 'left' | 'right' | 'center'
function changeAlign(align: Alignment) {
  console.log("修改方向:", align)
}

changeAlign("left")
```

字面量推理

- 我们来看下面的代码：

```ts
const info = {
  url: "https://coderwhy.org/abc",
  method: "GET"
}

function request(url: string, method: "GET" | "POST") {
  console.log(url, method)
}

request(info.url, info.method)
```

- 这是因为我们的对象再进行字面量推理的时候，info其实是一个 {url: string, method: string}，所以我们没办法将一个 string 赋值给一个 字面量 类型。

```ts
// 方式一：
request(info.url, info.method as "GET")
```

```ts
const info = {
  url: "https://coderwhy.org/abc",
  method: "GET"
} as const
```