TypeScript Sharing

分享人：沙鹏

奇怪的角度

拼写问题

TypeScript is **JavaScript with syntax for types.**

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

Duck Type

TypeScript 是一个结构化的类型系统，不同于 Java 等语言的标称类型系统，这种设计更符合我们平时开发 JavaScript 的习惯。

```
interface Point {
  x: number
  y: number
}

function printPointInfo(p: Point) {
  console.log(`x: ${p.x}, y: ${p.y}`)
}

const p = {
  x: 1,
  y: 2,
  z: 3,
}
// we can still use p
printPointInfo(p)
```

内容大纲

TypeScript: from zero to hero.

- 如何运行和使用 TypeScript
- 类型系统初探
- TypeScript 编译流程
- Challenge: TwoSum 类型体操

How to use TypeScript in our personal projects?

除了 babel 和 Webpack，如何使用现代化的工具开发 TypeScript。

- Client side
  - vite: 原生支持 TypeScript
- Server side
  - deno: 原生支持 TypeScript
  - tsx + unbuild: 使用 tsx 开发并且使用 unbuild / tsup 打包
- ESLint config
  - TypeScript + ESLint: @typescript-eslint
  - ESLint + prettier: eslint-config-prettier

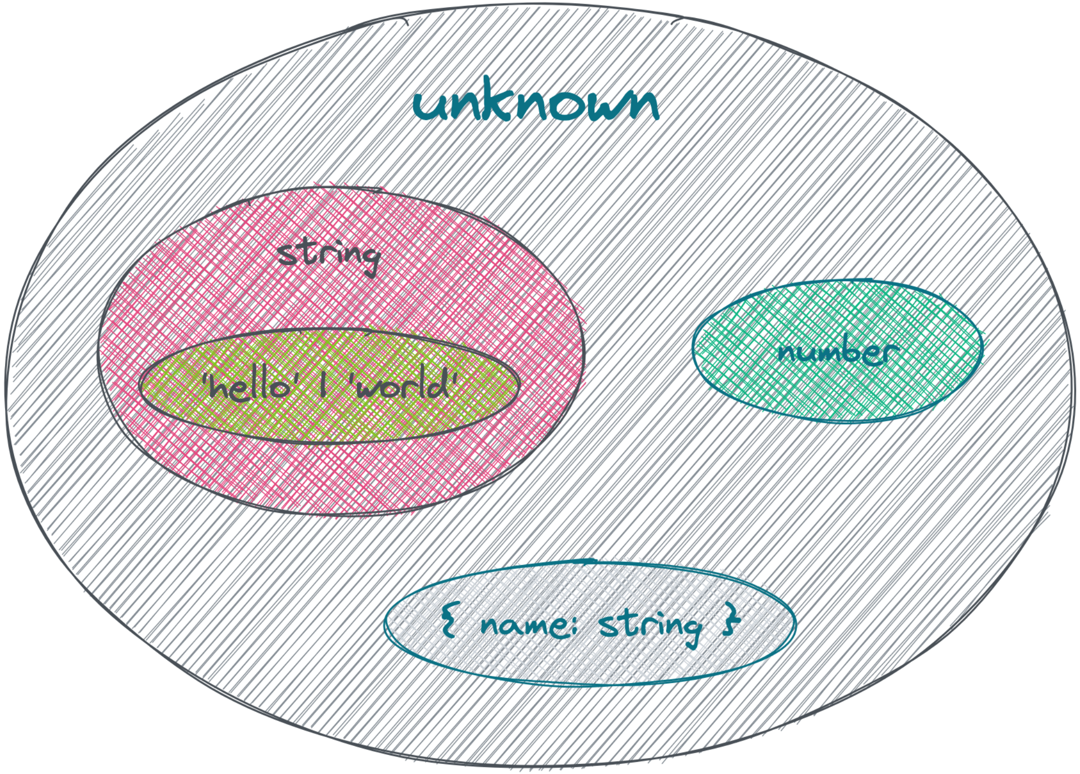
Data

TypeScript 类型编程不过是数据的转移，只不过这些数据都是类型罢了！

```
// 数据的组合与转移
type Primitives = number | boolean | string | undefined | null | symbol | bigint

type SomeLiterals = 20 | true | 'hello' | 10000n

type Add = (a: number, b: number) => number

type DataStructures =
  | { key: 'value' } // objects
  | [1, 2, 3] // tuples
  | number[] // lists

// 联合类型和交叉类型
type X = 'X'

type Y = 'Y'

type IntersectionAndUnions = (X & Y) | (X | Y)
```

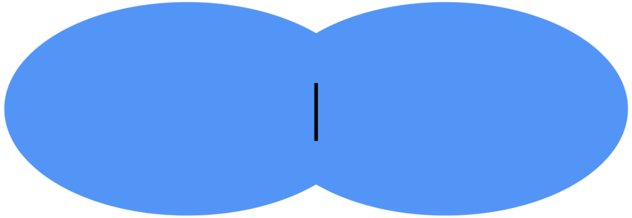
Types are Sets

TypeScript 中的类型本质上是一个集合！

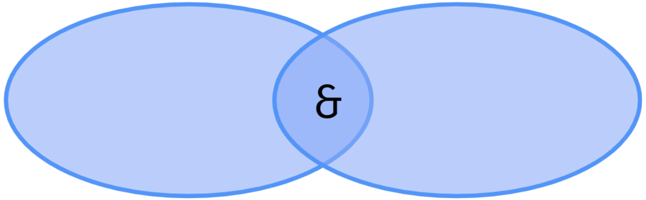


联合类型和交叉类型

`|` joins to set together.



`&` returns the intersection.



Functions and Code branching

范型就是类型系统的函数，我们可以用范型来构建 Utility Types。

```
type Func<A, B> = A | B

// 范型的类型约束
type Push<List extends any[], Item> = [...List, Item]

// 分支语句 本质上 A extends B 是在验证 A 是不是 B 的子类型 (subtype)
type If<A extends boolean, B, C> = A extends true ? B : C
```

Assignability

是否可分配的问题是我们在很多时候遇到最多的 TypeScript 报错了！

```
let a: number

// @ts-expect-error
a = 'hello world'
```

Assignability

我们继续来做个实验。。。

```
// if returns true, which means 'B is assignable to A'
// 说人话就是：父类型可以赋给子类型，反之则不行
// 我们用记号 A <: B 表示 A 是 B 的子类型
type IsSubtypeOf<A, B> = A extends B ? true : false

// 'hello world' <: number
type Test1 = IsSubtypeOf<'hello world', number> // false

// 'hello world' <: string
type Test2 = IsSubtypeOf<'hello world', string> // true

// let's see more example
type Test3 = IsSubtypeOf<{ hello: 'string' }, {}> // true
type Test4 = IsSubtypeOf<() => true, () => boolean> // true
type Test5 = IsSubtypeOf<(x: number) => void, (x: 1 | 2) => void> // why true?

// convariance vs contravariance
// 协变 vs 逆变
type F<A, B> = (x: A) => B
type Test6 = IsSubtypeOf<F<number, true>, F<1 | 2, boolean>>
```

The `infer` keyword

TypeScript 中的模式匹配！

```typescript
type GetTeam<U extends Record<string, unknown>> = U extends {
  name: string
  team: infer Team
}
  ? Team
  : never

type t = GetTeam<{ name: string; team: 'RNG' }>

// challenge
// implement Parameters and ReturnType
type p = Parameters<(x: number, y: string) => void> // [x: number, y: number]
type r = ReturnType<() => { hello: string }> // { hello: string }
type a = Awaited<Promise<boolean>> // boolean
```

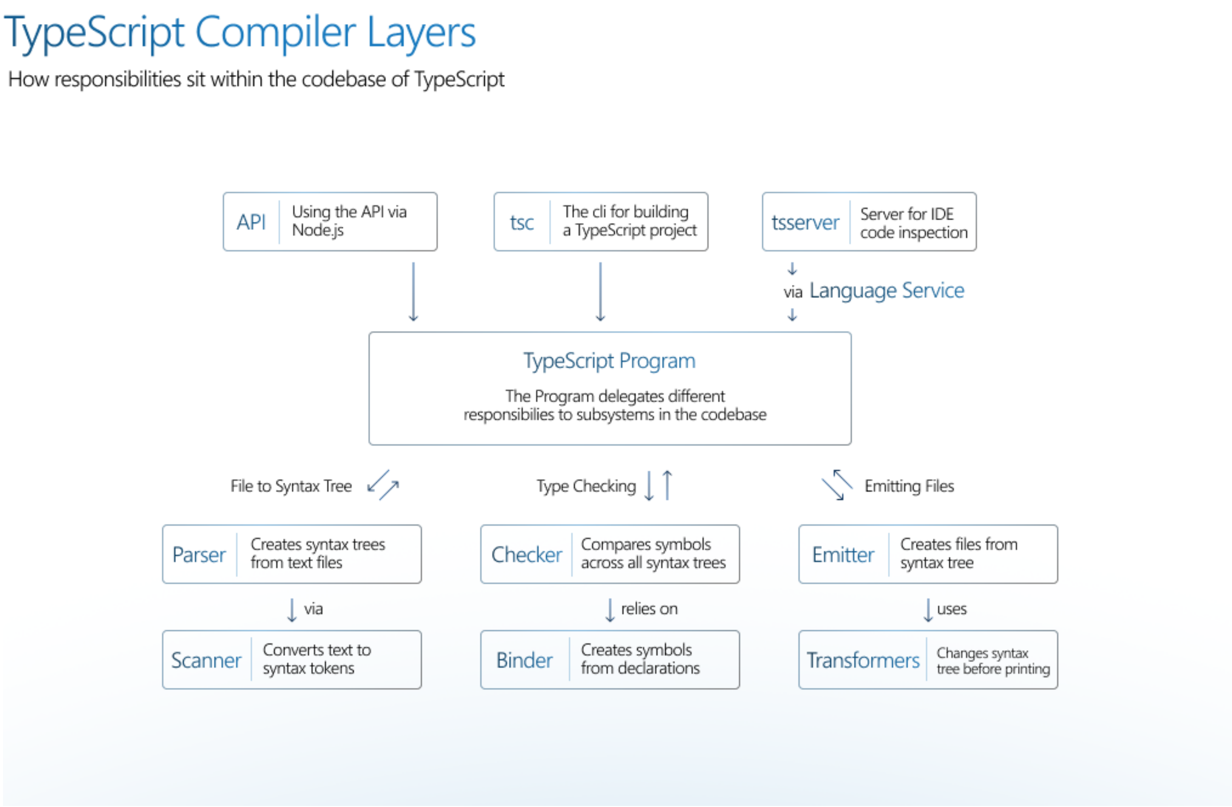
Loop

在类型系统中使用循环！

```
// Mapped Types
type OrNull<T extends Record<string, unknown>> = {
  [K in keyof T]: T[K] | null
}

type t = OrNull<{ a: number; b: number }>

// Recursive conditional types
type IsTwo<List extends any[]> = List extends [infer F, ...infer R] ? [F extends 2 ? true : false, ...IsTwo<R>] : []

type i = IsTwo<[1, 2, 3]>

// Mapping on Union Types
type Name = 'Alice' | 'Bob'

type NameToObject<Name> = Name extends string ? { name: Name } : never

type n = NameToObject<Name>
```

How does TypeScript compiler work?

The architecture!



TypeScript Compiler Layers

How responsibilities sit within the codebase of TypeScript

API – Using the API via Node.js

tsc – The cli for building a TypeScript project

tsserver – Server for IDE code inspection

via Language Service

TypeScript Program

The Program delegates different responsibilies to subsystems in the codebase

File to Syntax Tree

Type Checking

Emitting Files

Parser – Creates syntax trees from text files

via

Scanner – Converts text to syntax tokens

Checker – Compares symbols across all syntax trees

relies on

Binder – Creates symbols from declarations

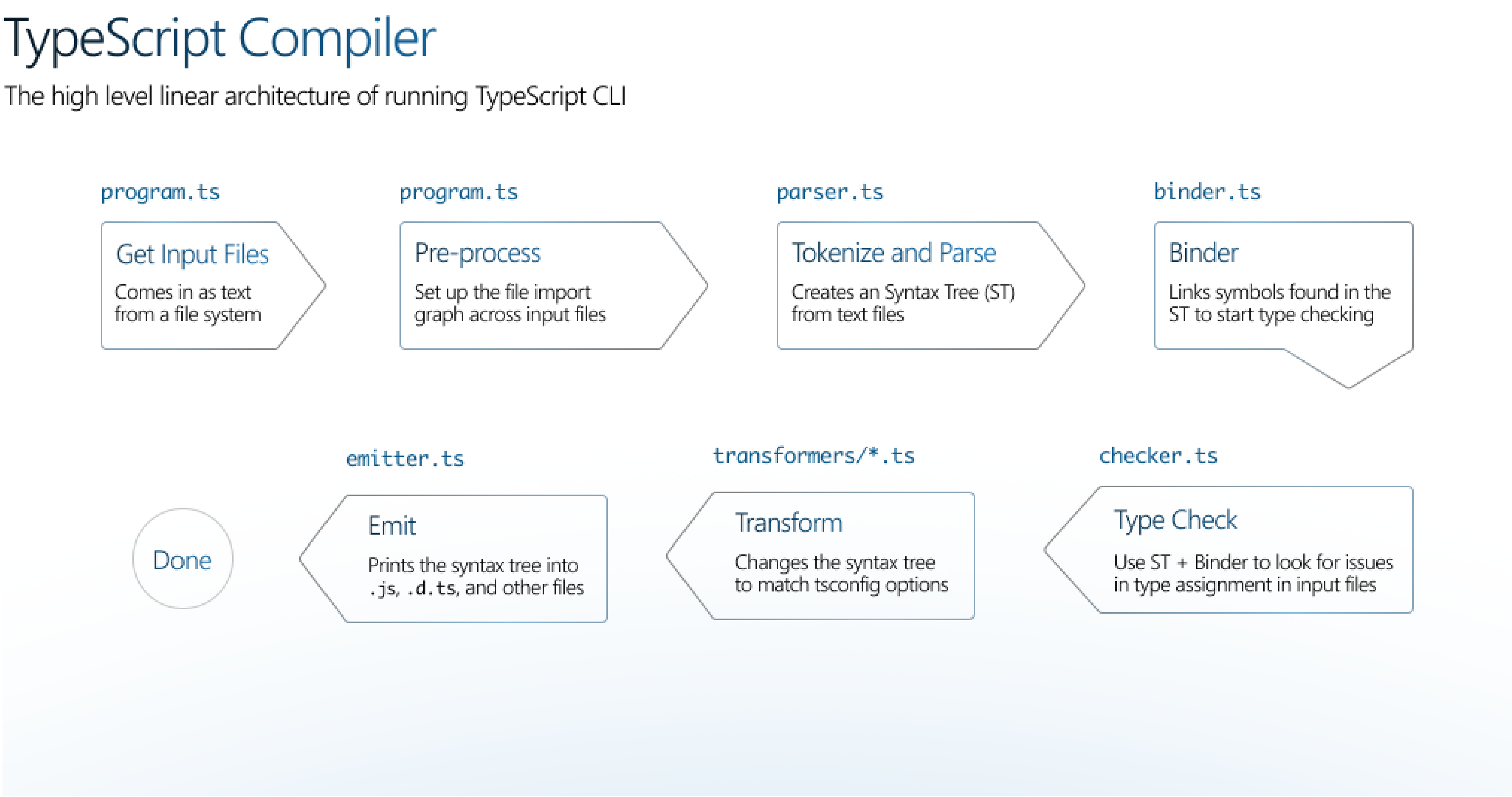
Emitter – Creates files from syntax tree

uses

Transformers – Changes syntax tree before printing

How does TypeScript compiler work?

CLI work flow!



TypeScript Compiler

The high level linear architecture of running TypeScript CLI

`program.ts`

Get Input Files

Comes in as text from a file system

`program.ts`

Pre-process

Set up the file import graph across input files

`parser.ts`

Tokenize and Parse

Creates an Syntax Tree (ST) from text files

`binder.ts`

Binder

Links symbols found in the ST to start type checking

`checker.ts`

Type Check

Use ST + Binder to look for issues in type assignment in input files

`transformers/*.ts`

Transform

Changes the syntax tree to match tsconfig options

`emitter.ts`

Emit

Prints the syntax tree into `.js`, `.d.ts`, and other files

Done

How does TypeScript compiler work?

Mini-TypeScript!

```
export function compile(s: string): [Module, Error[], string] {
  errors.clear()

  // scanner and parser
  const tree = parse(lex(s))

  // binder
  bind(tree)

  // checker
  check(tree)

  // transformer and emitter
  const js = emit(transform(tree.statements))

  return [tree, Array.from(errors.values()), js]
}
```

Some Tricks

基于上面我们学到的一些知识，简单分享一些实用的技巧👏

```
// module augmentation
declare module 'lodash' {
  export function uuid(): string
}

// as with literal string
type CapitalizeKey<T extends Record<string, unknown>> = {
  [K in keyof T as Capitalize<string & K>]: T[K]
}

type c = CapitalizeKey<{ hello: string }>

// how to implement Capitalize?
type MyCapitalize<T extends string> = T extends `${infer F}${infer R}` ? `${Uppercase<F>}${R}` : never

type h = MyCapitalize<'hello'>
```

TwoSum?

A little challenge for you guys.

```
// some utils
export type Expect<T extends true> = T
export type Equal<X, Y> = (<T>() => T extends X ? 1 : 2) extends <T>() => T extends Y ? 1 : 2 ? true : false

// our twosum
type TwoSum<T, U, Set> = any

// how to make the following things work
type cases = [
  Expect<Equal<TwoSum<[3, 3], 6>, true>>,
  Expect<Equal<TwoSum<[3, 2, 4], 6>, true>>,
  Expect<Equal<TwoSum<[2, 7, 11, 15], 15>, false>>,
  Expect<Equal<TwoSum<[2, 7, 11, 15], 9>, true>>,
  Expect<Equal<TwoSum<[1, 2, 3], 0>, false>>,
  Expect<Equal<TwoSum<[1, 2, 3], 1>, false>>,
  Expect<Equal<TwoSum<[1, 2, 3], 2>, false>>,
  Expect<Equal<TwoSum<[1, 2, 3], 3>, true>>,
  Expect<Equal<TwoSum<[1, 2, 3], 4>, true>>,
  Expect<Equal<TwoSum<[1, 2, 3], 5>, true>>,
  Expect<Equal<TwoSum<[1, 2, 3], 6>, false>>
]
```

Functional Programming

twosum 的函数式写法?

```
function twoSum(nums: number[], target: number, set: Set<number> = new Set()): boolean {
  if (nums.length === 0) {
    return false
  }

  return set.has(target - nums[0]) || twoSum(nums.slice(1), target, set.add(nums[0]))
}

// some more utils?
type ToTuple<L extends number, T extends unknown[] = []> = T['length'] extends L ? T : ToTuple<L, [...T, unknown]>

type Sub<A extends number, B extends number> = ToTuple<A> extends [...ToTuple<B>, ...infer Tail] ? Tail['length'] : -1

type Tail<T extends number[]> = T extends [unknown, ...infer Tail] ? Tail : []

type TwoSum<T extends number[], U extends number, Set = never> = any
```