TypeScript

TypeScript

- TypeScript 是一门由微软开发的 JavaScript 的超集，提供了类型系统和其他高级特性。
- 2024 年，JavaScript 世界中的绝大多数东西，都是经过 TypeScript 编译的。

- 官网：https://www.typescriptlang.org/
- 官方教程：https://www.typescriptlang.org/docs/handbook/intro.html
- GitHub：https://github.com/microsoft/TypeScript

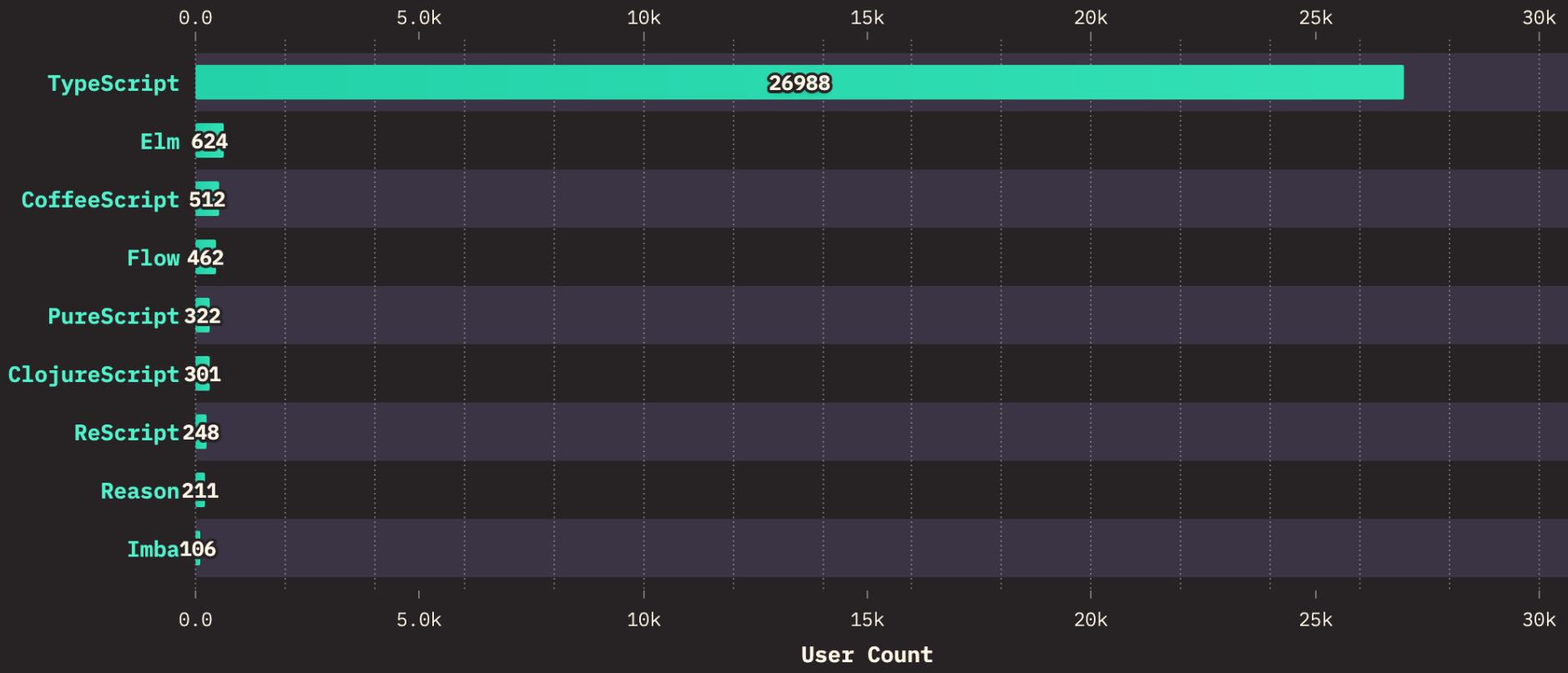
超集？

我们当然知道集合的超集，但是语言也有超集嘛？

- 语言的方言是指在原有语言的基础上进行扩展，以满足特定需求的语言变种。
  - HTML: ~~Pug~~, (EJS, Handlebars, Vue SFC, Svelte, …)
  - CSS: Sass/Scss, Less, ~~Stylus~~
  - JavaScript: TypeScript, JSX/TSX, ~~Flow, CoffeeScript~~
- 超集是一种特殊的方言，它包含了原有语言的所有特性，并且在此基础上进行了扩展。
  - 在编译时，只需要简单进行类型擦除，就获得了预期的 JavaScript 代码。
- 这些方言需要通过编译器转换为原有语言或 JavaScript，才能在浏览器或者 Node.js 中运行。

JAVASCRIPT FLAVORS

Languages that compile to JavaScript



JavaScript 很奇怪

奇怪的 `==` (事实上，工业界中几乎都是禁止使用 `==` 的):

```
if ("" == 0) {
  // 他们相等！但是为什么呢?
}
```

奇怪的隐式类型转换:

```
if (1 < x < 3) {
  // x 是任何值都为真!
}
```

任意访问对象上的属性不报错:

```
const obj = { width: 10, height: 15 }
// 为什么是 NaN? 拼写好难!
const area = obj.width * obj.heigth
```

为什么 TypeScript 会如此流行?

这一切都要从类型说起。

- JavaScript 是一门 **动态类型** (弱类型) 的语言。
- 人们需要一门 **静态类型** (强类型) 的语言。

什么是静态/动态类型?

```
var a;
a = 1;
a = 'foo';
```

什么是强/弱类型? (当然不管采用哪种定义，JavaScript 都应该是弱类型的)

> “我花了几个星期……试着弄清楚“强类型”、“静态类型”、“安全”等术语，但我发现这异常的困难……
> 这些术语的用法不尽相同，所以也就近乎无用。”
>
> ——Benjamin C. Pierce

类型是什么？

以我们熟悉的 C 语言为例。

- 原始类型（`short`，`int`，`long`，`float`，`double`，`char`，`void`，...）
- 结构体 / 类
- 枚举
- 指针
- 数组
- 函数
- ...

当我们写出一个变量名时，我们知道这个变量名对应的内存空间有多长，并且以怎样的形式存储数据。

类型是**数据结构**。

类型有什么用?

- **安全性**：提前检测出潜在的错误 (类型安全)
- **优化**：生成更高效的机器指令
- **可读性**：代码更易于理解；在 IDE 中提供更好的提示；自动文档生成
- **抽象化**：允许程序设计者对程序以较高层次的方式思考 (接口、模块化、协议)

事情并不总是朝着预想的方向发展

对于一门高级语言，类型如果就是单纯的数据结构会有什么问题?

- **安全性**：并不安全，因为程序员被迫使用各种指针和强制类型转换
- **优化**：没有受过专业训练的程序员无法理解编译器的优化策略，反而容易写出更低效的代码
- **可读性**：各种底层操作的可读性都不高
- **抽象化**：对于高级特性的表达能力不足

理想的情况是什么?

- ~~当我们写出一个变量名时，我们知道这个变量名对应的内存空间有多长，并且以怎样的形式存储数据。~~
- 当我们写出一个变量名时，我们不关心背后的内存布局，只想知道这个变量可能的行为是什么。

“鸭子类型”

当我们写出一个变量名时，我们不关心背后的内存布局，只想知道这个变量可能的行为是什么。

- number：我可以加减乘除，位运算，比较大小
- string：我可以拼接，截取，查找，替换
- function：我可以调用，并且知道参数和返回值的类型

对于对象呢?

- object：我希望能知道这个对象有哪些属性 (方法)

  如果两个对象有着完全相同的属性，那么它们应该就是同一种类型。

类型是集合！

类型是所有能赋值给有这一类型的变量的值的集合。

```
// 这里写的不是严格的 TypeScript 代码，只是为了说明概念
type NaturalNumber = ?

let a: NaturalNumber

a = 1 // OK
a = 10 // OK
a = -1 // Error
a = 'foo' // Error
```

原始类型

`number`,`string`,`boolean`

```
let a: number

a.toFixed() // OK
a.toUpperCase() // Error

let b: string

b.toUpperCase() // OK
b.toFixed() // Error
```

函数：参数类型

普通的函数和箭头函数都可以声明参数类型。

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

const greet = (name: string) => {
  console.log("Hello, " + name.toUpperCase() + "!")
}

greet(42) // Error
greet('world') // OK
```

函数：返回值类型

返回值类型是默认推断的，但是可以显式声明。

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

const double = (x: number) => x * 2 // OK
const double = (x: number): number => x * 2 // OK
const double = (x: number): string => x * 2 // Error
```

回调函数：参数类型推断

回调函数的参数类型可以通过上下文推断。

```
const names = ["Alice", "Bob", "Eve"]

names.forEach(function (s) {
  console.log(s.toUpperCase())
})

names.forEach((s) => {
  console.log(s.toUpperCase())
})
```

函数的类型

可以通过 `type` 关键字定义函数的类型。

```
type Double = (x: number) => number

// x 的类型可以自动推断
const double: Double = (x) => x * 2
```

数组

```
let a: string[] // 注意不是 [string]

a = [] // OK
a = ['foo'] // OK
a = ['foo', 'bar'] // OK

a = [1] // Error
a = ['foo', 1] // Error
```

数组上有很多方法都接受回调函数，比如 `map`，`find`，`filter`，`reduce` 等。

```
a.map(s => s.toUpperCase()) // 能够推断出 s 的类型是 string，以及整个表达式的类型是 string[]
```

对象字面量

通过花括号声明一个对象字面量类型。

```
function printCoord(point: { x: number; y: number }) {
  console.log("The coordinate's x value is " + point.x)
  console.log("The coordinate's y value is " + point.y)
  console.log("The coordinate's z value is " + point.z) // Error
}

printCoord({ x: 3, y: 7 }) // OK
printCoord({ x: 3, y: '7' }) // Error
printCoord({ x: 3 }) // Error
```

可选属性

通过`?:`声明可选属性。

```
function printCoord(point: { x: number; y?: number }) {
  console.log("The coordinate's x value is " + point.x)
  if (point.y !== undefined) {
    console.log("The coordinate's y value is " + point.y)
  }
}

printCoord({ x: 3, y: 7 }) // OK
printCoord({ x: 3, y: '7' }) // Error
printCoord({ x: 3 }) // Now OK!
```

类型别名与接口

有两种方式可以定义一个对象类型。

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

interface Point {
  x: number
  y?: number
}

function printCoord(point: Point) { ... }
```

它们有哪些区别？(建议：对象用 `interface`，其他情况用 `type`)

- `interface` 在错误显示上可能更友好
- `interface` 支持定义合并 (后面会讲)
- `type` 可以用于定义任何类型，`interface` 只能用于定义对象类型

并集

类型的并集相当于集合的并集，可以通过 `|` 运算符实现。

```
let a: string | number

// number
a = 1 // OK
a = -0.5 // OK

// string
a = 'foo' // OK

// other
a = true // Error
```

类型收窄

基于控制流分析。

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

类型分配

来看一个特别的例子。

```
// Return type is inferred as number[] | string
function getFirstThree(x: number[] | string) {
  // 如果 x: number[]，应当返回 number[]
  // 如果 x: string，应当返回 string
  return x.slice(0, 3)
}
```

类型断言

不完全是强制类型转换。严格了，但是没完全严格。

```
'foo' as string // OK, 虽然你不用写这个 as
'foo' as number // Error
'foo' as unknown as number // OK

let a: string | number = 1
let b = a as string // OK

let x: string = 'foo'
let y = x as string | number // OK
```

字面量类型

`let` 和 `const` 的区别。

```
let a = 1 // a: number

const b = 2 // b: 2

let c = 3 as const
let c: 3 = 3

c = 4 // Error
```

这种写法的主要用处是可以实现类似枚举的效果。

```
function printText(s: string, alignment: "left" | "right" | "center") { ... }

printText("Hello, world", "left") // OK
printText("G'day, mate", "centre") // Error
```

尽管 TypeScript 也有枚举类型，但是字面量类型的并集有时更加灵活。

子类型

子类型允许我们传入一个比定义更具体的类型的值。

```
function printCoord(point: { x: number; y?: number }) {
  console.log("The coordinate's x value is " + point.x)
  if (point.y !== undefined) {
    console.log("The coordinate's y value is " + point.y)
  }
}

printCoord({ x: 3, y: 7, z: 9 }) // OK!
```

回顾一下这个例子，本质上也是子类型的应用。

```
function printText(s: string, alignment: "left" | "right" | "center") { ... }

printText("Hello, world", "left" as "left") // OK
printText("Hello, world", "left" as string) // Error
```

泛型

泛型是一种参数化类型的方式。

```
function add<T>(x: T, y: T): T { ... }

add(1, 2) // 3
add('foo', 'bar') // foobar
```

再看一个例子：

```
function last<T>(arr: T[]): T | undefined {
  return arr[arr.length - 1]
}

last([1, 2, 3]) // 3
last(['foo', 'bar']) // bar
last([]) // undefined
```

泛型约束

`extends` 用于声明泛型的约束。

```
function longer<T extends { length: number }>(a: T, b: T) {
  if (a.length >= b.length) {
    return a
  } else {
    return b
  }
}

longer([1, 2], [1, 2, 3]) // [1, 2, 3]
longer("alice", "bob") // "alice"
longer(10, 100) // Error
```

交集

有了并集，自然也有交集。

```ts
type A = { a: number }
type B = { b: string }
type C = A & B

let c: C

c = { a: 1, b: 'foo' } // OK
c = { a: 1 } // Error
c = { b: 'foo' } // Error
```

TypeScript 强吗?

- TypeScript 的类型系统本身是图灵完备的
  - 这意味着你能用它来实现四则运算、实现小游戏、甚至写个 JSON 解析器
  - https://github.com/type-challenges/type-challenges
- TypeScript 提供了一种“类型是集合”视角下的经典实现
  - 支持类型的交集、并集、子类型、泛型、字面量类型等特性
  - 支持比较健全的类型推导、类型收窄 (和一定程度上的控制流分析)

类型系统

一门编程语言的类型系统就好比一棵技能树。

- 函数
- 异常
- 引用 / 借用
- 元组 / 单元类型 / 积类型
- 记录 / 对象类型
- 无交并 / 余积类型
- 枚举 / 变体 / 归纳类型
- 子类型
  - 顶 / 底类型
  - 交 / 并类型
  - 继承 / 接口
- 递归类型
- 多态
  - 参数化多态 / 泛型
  - 子类型多态
  - 特设多态 / 重载 / 特征
  - 协变 / 逆变 / 不变
- 高阶类型
- 依值类型
- 亚结构类型
  - 所有权 / 内存安全
  - 异步 / 并发安全
- 作用类型
- 模块
- 类型推断

Curry-Howard 对应

类型是命题！

- 假设类型 P 对应了一个命题，$p : P$ 对应了这个命题的一个证明
- 积类型 / 二元组类型 $P \times Q$ 对应了命题 $P \wedge Q$
- 无交并 / 余积类型 $P + Q$ 对应了命题 $P \vee Q$
- 函数类型 $P \to Q$ 对应了命题 $P \to Q$

类型是什么?

你在第几层?

0. 类型是数据结构
1. 类型是集合
2. 类型是命题
3. 类型是群胚 / 空间

我心目中的评分 (综合考虑各项语言特性，并不是分数越高就越好):

- C: 0
- Java/Go/Modern C++: 0.5
- TypeScript/Rust: 1
- Haskell/OCaml: 1.5
- Lean/Coq/Agda: 2~3

Koishi

Koishi.js

创建跨平台、可扩展、高性能的机器人

即刻起步 了解更多

Koishi

- Koishi 是一个跨平台、可扩展、高性能的聊天机器人框架。
- 基于 TypeScript 开发。
- 官方文档：https://koishi.chat/

聊天机器人有什么用?

- 企业功能
  - 任务提醒
  - 会议安排
- 个人助手
  - 消息通知、备忘录
  - 接入 GPT 等模型
- 群聊管理
  - 自动验证、关键词回复
  - 权限管理
- 娱乐功能
  - 互动玩法
  - 游戏辅助

插件

插件市场

官方插件一览

排序

综合

按评分

按下载量

按创建时间

按更新时间

筛选

官方认证 67

不安全 65

开发中 24

快速体验 107

插件市场

当前共有 1363 个可用于 v4 版本的插件 (4/7/2024, 10:49:30 AM)

输入想要查询的插件名

1 2 3 4 ... 57

adapter-satori

Satori 协议适配器

官方适配的平台

- 开发需要企业身份
  - 微信公众号
  - 企业微信
  - QQ (官方 API)
- 在企业内使用
  - 飞书 / Lark
  - 钉钉
- 其他国内平台
  - QQ (第三方登录方案)
  - 开黑啦 / KOOK
- 国外平台
  - Discord
  - Telegram
  - Slack
  - LINE
  - WhatsApp
- 自建方案
  - Matrix
  - Zulip
- 无平台
  - **测试沙盒 (推荐使用这个开发)**
  - 内置 IM
  - 网页内对话

安装

https://koishi.chat/zh-CN/manual/starter/boilerplate.html

插件系统

插件化是模块化的一种延伸，它使得一个系统能够在保持本体相对轻量的同时，允许用户扩展更多的功能。

实际上，大多数框架的插件系统都存在着一些问题：

1. **不完全的插件化**。这些框架往往仅仅将部分外围功能下放到了插件中，而核心功能仍然是硬编码的。
2. **不可逆的插件化**。这些框架往往不支持插件在运行时停用 (回收插件占用的资源)。

Koishi 基于名为 Cordis 的框架，同时解决了这两个问题。

- Koishi 目前有超过 1300 个插件。一个 Koishi 实例可能装载了上百个插件。
- 几乎所有功能都是通过插件实现的：适配器、数据库、业务功能、用户界面、……
- 插件自身的副作用和插之间的依赖关系都能够被 Koishi 妥善处理，确保了资源安全性。

可逆的插件系统

Koishi 的插件系统有什么特别之处?

任意进行加载和卸载插件操作后，最终行为仅与最终启用的插件相关；与中间是否重复加载过插件、插件之间的加载或卸载顺序都无关。你也可以简单理解为「路径无关」。包括：

- 任意次加载并卸载一个插件后，内存占用不会增加。
- 任意次加载并卸载一个插件后，不会残留对其他插件的影响。
- 如果插件之间有依赖关系，依赖的插件会自动在被依赖的插件之后加载，并自动在被依赖的插件之前卸载，即确保插件的生命周期由依赖关系而非加载顺序决定。

可逆的插件系统

实现了可逆性以后有什么好处?

- **启动速度**：插件的生命周期仅由依赖关系决定，冷启动时可以最大化并发性能
- **可扩展性**：任何服务都可以被替换实现，确保了可扩展性
- **无感开发**：所有的 API 产生的副作用都是自动回收的
- **热重载、滚动更新**：所有插件都可以在运行时加载、卸载和重载

发布插件

- 注册一个 npm 账号：http://npmjs.com/
- https://koishi.chat/zh-CN/guide/develop/publish.html

如果你把它作为大作业 (**这不是必须的**)，你应当：

- 任选主题 (我们只会评价你的代码，不会关注功能，写你想写的都行)
- 将插件发布到 GitHub 和 npm，并且能证明你是插件的作者
- 在作业中包含一份书面的报告，介绍你开发的插件

关于 OSPP 2024 活动 (**跟这门课程的评价没有任何关系**)：https://summer-ospp.ac.cn/

- Koishi 作为组织参加了这个活动，会发布 2-3 个项目，每个项目可以有 1 个名额
- 如果你有兴趣，可以先通过大作业熟悉 Koishi 的开发流程，再联系我 (不保证能选上)
- 你也可以报名参加其他组织的选题 (去年北大 Linux 俱乐部也有参加)

Questions?

任何跟 TypeScript 开发、Koishi 插件开发相关的问题都可以问我。