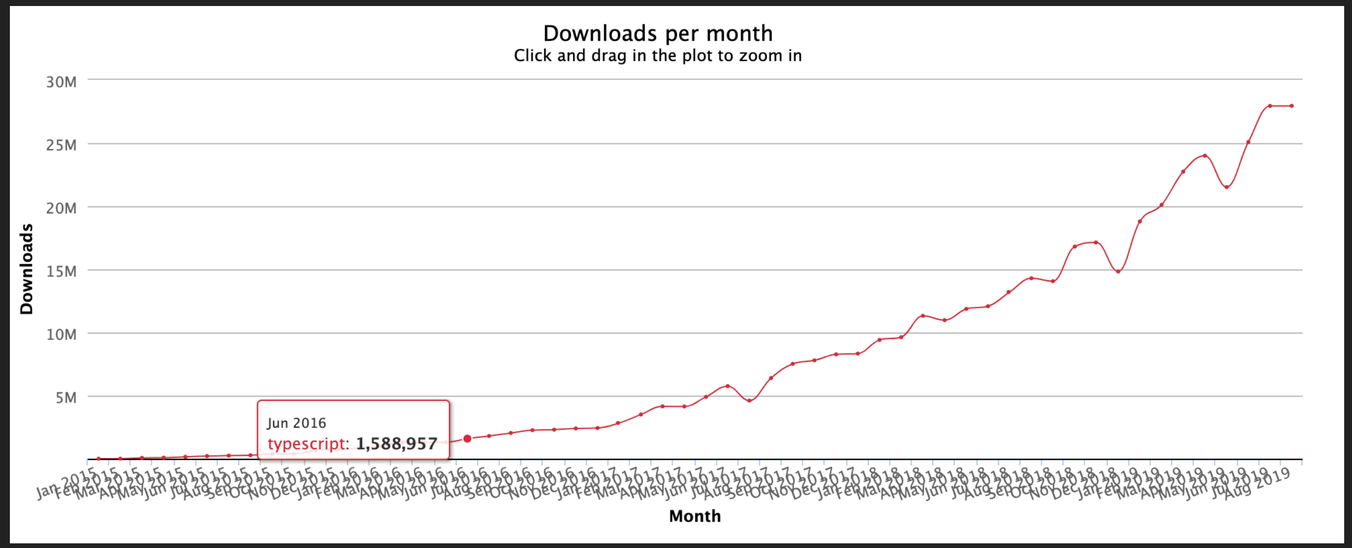
TYPESCRIPT

进阶知识分享与实践交流 — 前端组 姜康

技术选择

- 生态
- 语言排行榜
- 工程角度
- 成熟度



类型系统

- 基础类型：number、string、boolean、Array、Enum、Tuple、void、null、undefined、never、any、object
- 高级类型：联合类型、交叉类型、索引类型、映射类型、条件类型
- 泛型
- 类
- 接口
- 函数

基础类型 - 枚举

```ts
enum AnimalFlags {
  None      = 0,
  HasOne    = 1 << 0
}
enum AnimalFlags {
  None,
  HasOne
}
```

```ts
enum AnimalFlags {
  NONE = 'none',
  HAS = 'has'
}
```

```ts
export enum Weekday {
  Monday,
  Tuseday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

namespace Weekday {
  export function isBusinessDay(day: Weekday) {
    switch (day) {
      case Weekday.Saturday:
      case Weekday.Sunday:
        return false;
      default:
        return true;
    }
  }
}

const mon = Weekday.Monday;
const sun = Weekday.Sunday;

console.log(Weekday.isBusinessDay(mon)); // true
console.log(Weekday.isBusinessDay(sun)); // false
```

接口与类

```
interface SquareConfig {
  color?: string;
  readonly width?: number;
  [propName: string]: any;
  getArea(width: number, height?: number): number;
}

interface ReactAngularConfig extends SquareConfig {
  readonly height: number;
}

class Square implements SquareConfig {
  getArea = (width: number, height?: number): number => {
    return width * height ? height : width;
  }
}
```

高级类型 - 联合类型

```
// function padLeft(value: string, padding: string | number) {
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
      return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
      return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

console.log(padLeft("Hello world", 4)); // returns "    Hello world"
console.log(padLeft("Hello world", {})); // throw Error
```

两个类型选取其一

高级类型 - 交叉类型

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

class Person {
  constructor(public name: string) { }
}
interface Loggable {
  log(): void;
}
class ConsoleLogger implements Loggable {
  log() {
      // ...
  }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

两个类型合二为一

高级类型 - 索引类型

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

获取对象的属性集合

高级类型 - 映射类型

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

type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

改变类型中属性的特征

TS 对这部分特性进行了标准化，成为语言的一部分

高级类型 - 条件类型

```ts
type TypeName<T> =
    T extends string ? "string" :
    T extends number ? "number" :
    T extends boolean ? "boolean" :
    T extends undefined ? "undefined" :
    T extends Function ? "function" :
    "object";

type T0 = TypeName<string>;  // "string"
type T1 = TypeName<"a">;  // "string"
type T2 = TypeName<true>;  // "boolean"
type T3 = TypeName<() => void>;  // "function"
type T4 = TypeName<string[]>;  // "object"
```

需要讲一下 extends

T extends U ? X : Y，若 T 能够赋值给 U，那么类型为 X，否则为 Y

泛型

```
class Utility {
  reverse<T>(items: T[]): string[] {
    const toreturn: string[] = [];
    for (let i = items.length-1; i >= 0; i--) {
      toreturn.push(String(items[i]));
    }
    return toreturn;
  }
}

let utils = new Utility();
console.log(utils.reverse<number>([1,2,34]));

const getJSON = <T>(config: { url: string; headers?: { [key: string]: string } }):
Promise<T> => {
  const fetchConfig = {
    method: 'GET',
    Accept: 'application/json',
    'Content-Type': 'application/json',
    ...(config.headers || {})
  };
  return fetch(config.url, fetchConfig).then<T>(response => response.json());
};
```

到这里为止，基础知识介绍完毕

提供成员之间的类型关联

类型保护

- `typeof x === 'string'`
- `person instanceof Person`
- `'color' in Shape`
- `obj.property === 'someone'`
- `function isType(arg: TypeA | TypeB): arg is Type {  }`

提供一些判断来保证正确的调用对象中的类型

虽然方法多样，但是某些场景下，还是有更容易的方式来做到类型安全的方式下来使用对象成员。

类型断言

DEMO1：

```
function returnNumber (arg: number): number {
  let n = 1;
  return n;
}
```

DEMO2：

```
const bar = [1, 2];
let [a, b] = bar;

a = 'hello';
```

通过一些简单的规则判断出变量的类型

技巧：使用“！”符号进行 null 与 undefined 进行断言！

其他方式：

类型保护

xxx as Type

<Type>xxx

虽然方法多样，但是某些场景下，还是有更容易的方式来做到类型安全的方式下来使用对象成员。

互动：
联合类型下：判断为空，如何判断?

变体的名词解释

假定：A、B 表示类型，A -> B 表示 A为参数类型，B 为返回类型为B的函数类型，<= 表示子类型关系。

协变（参数）：A <= B，(T -> A) <= (T -> B)

协变（返回值）：A <= B，(B -> T) <= (A -> T)

不变：类型不同则不兼容

双向协变：即是逆变又是协变（TS 的选择）

泛型

类型转换后的继承关系

类型兼容

```ts
interface Named {
  name: string;
}

let x: Named;
let y = { name: 'Alice', location: 'Seattle' };
x = y;
```

剩余参数、可选参数等

关键词 - type

```ts
type Tuple = [number, string];
const tuple: Tuple = [2, 'sir'];

type Size = 'small' | 'default' | 'big' | number;
const size: Size = 24;
```

1. 用于创建别名
2. 对比 interface 支持更复杂的类型
3. 不支持 function 属性
4. 不支持类型兼容特性，编译阶段提示错误

可以代码演示下

关键词 - extend

```ts
interface A {
  a: number
}

interface B extends A {
  b: string
}

// 与上一种等价
type AB = A & {
  b: string
}
```

虽然方法多样，但是某些场景下，还是有更容易的方式来做到类型安全的方式下来使用对象成员。

关键词 - infer

```ts
type ParamType<T> = T extends (param: infer P) => any ? P : T;

interface User {
  name: string;
  age: number;
}

type Func = (user: User) => void;

type Param = ParamType<Func>; // Param = User
type AA = ParamType<string>; // string
```

用于表示在 extends 语句中待推断的类型

同态下的映射类型实践

```ts
// 设置对象所有属性 key 为可选项
type Partial<T> = { [P in keyof T]?: T[P] };

// 设置对象所有属性 key 为必填项项
type Required<T> = { [P in keyof T]-?: T[P] };

// 设置对象所有属性为只读项
type Readonly<T> = { readonly [P in keyof T]: T[P] };
```

同态：属性未变化

其他工具类型

```ts
// 从对象 T 中挑选出 属性 key 为 K 的属性
type Pick<T, K extends keyof T> = { [P in K]: T[P] };

// 把对象 K 所有属性设置为 T 类型
type Record<K extends keyof any, T> = { [P in K]: T };

// 把对象 T 中，与 U 存在交集的属性排除掉
type Exclude<T, U> = T extends U ? never : T;

// 从对象 T 中，挑选出与 U 存在交集的属性
type Extract<T, U> = T extends U ? T : never;

// 从对象 T 中排除属性 key 为 K 的 key 的属性
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;

// 获得 T 的返回值类型
type ReturnType<T extends (...args: any) => any> = T extends (...args: any) => infer R ? R : any;
```

命名空间与模块

```ts
namespace LogService {
  export function log(msg) {
    console.log(msg);
  }
  export function error(msg) {
    console.log(msg);
  }
}
```

```ts
var LogService;
(function (Utils) {
  function log(msg) {
      console.log(msg);
  }
  Utils.log = log;
  function error(msg) {
      console.log(msg);
  }
  Utils.error = error;
})(Utils || (Utils = {}));
```

1. 早期 TS 的模块化手段，现可以直接用 ES Module 进行代替
2. 命名空间仍然可用，仍然是 TS 组织代码的一种方式
3. Angular 中命名空间可以省略 Namespace 的前缀
4. React 中不建议使用

声明文件是什么

```
declare interface Something {
  x: string;
};
```

1. 描述库中方法、变量等成员，提供使用的方式
2. declare 为顶级的“声明”，大于“局部”的变量定义
3. d.ts 文件
4. 第三方库适配
5. 有时候也需要 namespace 介入

三斜线

```
/// <reference path="custom-typings.d.ts">
```

导入引用文件，确定关联文件

新特性

安全导航符：

```
// safe navigation operator，最初在 Angular 框架中引用，现已进入 TC39 stage 3 阶段（Babel6 已支持语法）。
// 支持 3 种使用方式：
obj?.props        对象中的属性访问   => obj == null ? undefined : obj.props
obj?.[props]     数组中的值访问       => obj == null ? undefined : obj[props]
func?.(...args) 函数调用                  => obj == null ? undefined : func(...args)
```

Lint:

官方已经在 ts 库中实践了 eslint 规则，并给了 AST 解析的包，2019 路线图中明确了完全使用 eslint 代替 tslint 的规划。

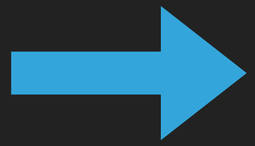
const断言：

```
// Type '10'
let x1 = 10 as const;

// Type 'readonly [10, 20]'
let y2 = [10, 20] as const;

// Type '{ readonly text: "hello" }'
let z3 = { text: "hello" } as const;
```

新特性

将参数类型转换为析构对象：

```
function updateOptions(
  hue?: number,
  saturation?: number,
  brightness?: number,
  positionX?: number,
  positionY?: number,
  positionZ?: number,) {
  // ....
}
```



```
interface Options {
  hue?: number,
  saturation?: number,
  brightness?: number,
  positionX?: number,
  positionY?: number,
  positionZ?: number,
}

function updateOptions(options: Options = {}) {

  // ....
}
```

解决参数类型不匹配的问题，保证了"类型"安全

新特性

类型安全的新类型 **unknown**：

```
function prettyPrint(x: unknown): string {
  if (Array.isArray(x)) {
    return "[" + x.map(prettyPrint).join(", ") + "]"
  }
  if (typeof x === "string") {
    return `"${x}"`
  }
  if (typeof x === "number") {
    return String(x)
  }
  return "etc."
}
```

与 any 一样，任何值都可以赋给 unknown，但是当没有类型断言或基于控制流的类型细化时 unknown 不可以赋值给其它类型，除了它自己和any外。同样地，在unknown 没有被断言或细化到一个确切类型之前，是不允许在其上进行任何操作的。

REACT 中的一些实践

```
<Input placeholder="请输入任务名称"
 value={taskName}
 onChange={
  (e: any) => setTaskName(e.target.value)}
/>
```

如何把 any 替换掉呢?

any => Event（BasicSyntheticEvent）?

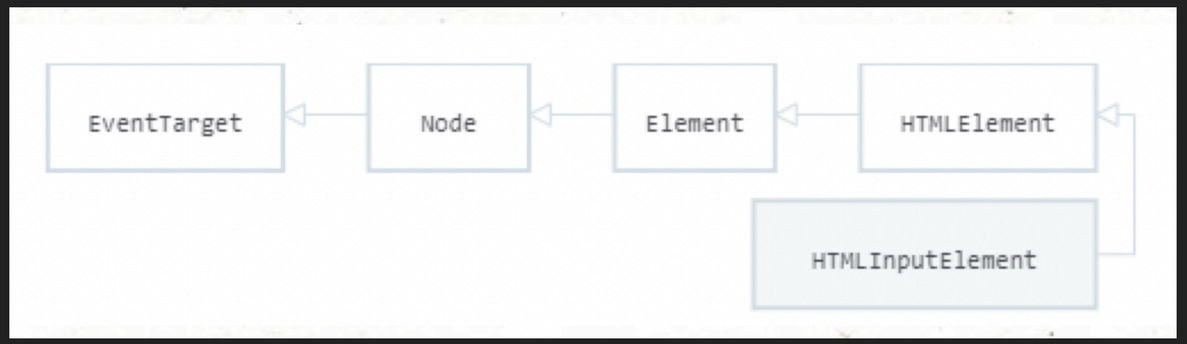
target.value 不存在

```
interface BaseSyntheticEvent<E = object, C = any, T = any> {
      nativeEvent: E;
      currentTarget: C;
      target: T;
      bubbles: boolean;
      cancelable: boolean;
      defaultPrevented: boolean;
      eventPhase: number;
      isTrusted: boolean;
      preventDefault(): void;
      isDefaultPrevented(): boolean;
      stopPropagation(): void;
      isPropagationStopped(): boolean;
      persist(): void;
      timeStamp: number;
      type: string;
    }

interface SyntheticEvent<T = Element, E = Event>
   extends BaseSyntheticEvent<E, EventTarget & T, EventTarget> {}
```

REACT 中的一些实践

查看 Event 标准，target 解释

Used to indicate the `EventTarget` to which the event was originally dispatched.




```
// Introduced in DOM Level 2:
interface Event {
  // PhaseType
  const unsigned short      CAPTURING_PHASE    = 1;
  const unsigned short      AT_TARGET          = 2;
  const unsigned short      BUBBLING_PHASE     = 3;

  readonly attribute DOMString         type;
  readonly attribute EventTarget      target;
  readonly attribute EventTarget      currentTarget;
  readonly attribute unsigned short   eventPhase;
  readonly attribute boolean          bubbles;
  readonly attribute boolean          cancelable;
  readonly attribute DOMTimeStamp     timeStamp;
  void     stopPropagation();
  void     preventDefault();
  void     initEvent(in DOMString eventTypeArg,
                     in boolean canBubbleArg,
                     in boolean cancelableArg);
};

// Introduced in DOM Level 2:
interface EventTarget {
  void  addEventListener(in DOMString type,
                         in EventListener listener,
                         in boolean useCapture);
  void  removeEventListener(in DOMString type,
                         in EventListener listener,
                         in boolean useCapture);
  boolean  dispatchEvent(in Event evt)
                        raises(EventException);
};
```

REACT 中的一些实践

解决方案：

```
(event: SyntheticEvent) {
  const {target} = event;
  if (!(target instanceof window.HTMLInputElement)) {
    return;
  }
}
```

```
React.ChangeEvent<HTMLInputElement>
```

```
(event.target: window.HTMLInputElement).value
```