TYPESCRIPT

rortegag

Análisis de control de flujo

Puntos clave

El análisis de control de flujo casi siempre toma un sindicato y reduce el número de tipos dentro de la unión basándose en lógica en su código.
La mayoría de las veces, el análisis de control de flujo funciona dentro de la lógica booleana, pero hay formas de definir sus propias funciones que afectan a cómo TypeScript reduce los tipos.

Declaraciones If

La mayor parte del estrechamiento proviene de expresiones dentro de sentencias if, donde diferentes operadores de tipo se estrechan dentro del nuevo ámbito.

typeof (para tipos primitivos)

```
const input = getUserInput()
input // string | number

if(typeof input === “string”) {
  input //string
}
```

instanceof (para clases)

```
const input = getUserInput()
input // number | number[]

if(input instanceof Array) {
  input // number[]
}
```

“property” en objeto (para objetos)

```
const input = getUserInput()
input // string | { error: ... }

if (“error” in input) {
  input // { error: ...
}
```

funciones type-guard (para cualquier cosa)

```
const input = getUserInput()
input // number | number[]

if (Array.isArray(input)) {
  input: // number[]
}
```

Expresiones

El estrechamiento también se produce en la misma línea de código, en operaciones booleanas.

```
const input = getUserInput()
input // string | number

const inputLength =
  (typeof input === “string” && input.length) || input
                              // input: string
```

Uniones discriminadas

Todos los miembros(status) de la unión tienen el mismo nombre de propiedad, el análisis de control de flujo puede discriminarlo.

```
type Responses =
  | { status: 200, data: any }
  | { status: 301, to: string }
  | { status: 400, error: Error }
```

Uso

```
const response = getResponse()
response // Responses

switch(response.status) {
  case 200: return response.data
  case 301: return redirect(response.to)
  case 400: return response.error
}
```

MÁS

Type Guards

Una función con un tipo de retorno que describe el cambio del análisis de control de flujo para un nuevo ámbito cuando es verdadero.

```
function isErrorResponse(obj: Response): obj is APIErrorResponse {
    return obj instanceof APIErrorResponse
}
```

Uso

```
const response = getResponse()
response // Response | APIErrorResponse

if (isErrorResponse(response)) {
  response // APIErrorResponse
}
```

Funciones de aserción

Una función que describe los cambios del análisis de control de flujo que afectan al ámbito actual porque lanza en lugar de devolver false.

```
function assertResponse(obj: any): asserts obj is SuccessResponse {
  if (!(obj instanceof SuccessResponse)) {
    throw new Error(“¡Error!”)
  }
}
```

Uso

Las funciones de aserción cambian o lanza el ámbito actual.

```
const response = getResponse()
response // SuccessResponse | ErrorResponse

assertResponse(res)
res // SuccessResponse
```

Para más información: https://www.typescriptlang.org/docs/

TYPESCRIPT

rortegag

Análisis de control de flujo

Asignación

Limitación de tipos mediante “as const”.
Los subcampos de los objetos se tratan como si pudieran mutar, y durante la asignación el tipo se ‘amplia’ a una versión no literal. El prefijo ‘as const’ bloquea todos los tipos a sus versiones literales.

```
const data1 = {
  name: “Zagreus”
}
```

→

```
const data1 = {
  name: “Zagreus”
}
```

```
const data2 = {
  name: “Zagreus”
} as const
```

→

```
const data2 = {
  name: “Zagreus”
} as const
```

Rastrea las variables relacionadas

```
const response = getResponse()
const isSuccessResponse
  = res instanceof SuccessResponse

if (isSuccessResponse)
  res.data // SuccessResponse
```

Tipos de actualizaciones de reasignación

```
let data: string | number = ...
data // string | nunber
data = “Hello”
data // string
```

Clases

Puntos clave

Una clase TypeScript tiene algunas extensiones específicas de tipo a las clases JavaScript de ES2015 y una o dos adiciones en tiempo de ejecución.

Crear una instancia de clase

```
class ABC { ... }
const abe = new ABC()
```

Los parámetros del nuevo ABC provienen de la función constructora.

private x versus #private

El prefijo private es una adición de tipo y no tiene efecto en tiempo de ejecución. El código de fuera de la clase puede llegar al elemento en el siguiente caso:

```
class Bag {
  private item: any
}
```

Vs #private que es runtimeprivate y tiene aplicación dentro del motor JavaScript que sólo es accesible dentro de la clase:

```
class Bag { #item: any }
```

‘this’ en clases

El valor de 'this' dentro de una función depende de cómo sea llamada la función. No se garantiza que sea siempre la instancia de la clase en otros lenguajes.
Puede utilizar los ‘parámetros this’, utilizar la función bind, o las funciones para solucionar el problema cuando ocurra.

Tipo y valor

Sorpresa, una clase puede utilizarse como tanto un tipo como un valor.

```
const a:Bag = new Bag()
          //Tipo      //Valor
```

Así que ten cuidado de no hacer esto:

```
class C implements Bag {}
```

MÁS

Para más información: https://www.typescriptlang.org/docs/

TYPESCRIPT

rortegag

Clases

Sintaxis común

```
class User extends Account implements Updatable, Serializable {
  id: string;                                  Un campo.
  displayName?: boolean;                       Un campo opcional.
  name!: string;                               Un campo “confía en mí, está ahí”.
  #attributes: Map<any, any>;                  Un campo privado.
  roles = [”user”];                            Un campo con un valor predeterminado.
  readonly createdAt = new Date()              Un campo de sólo lectura con un valor por defecto.

  constructor(id: string, email: string) {     El código llamado en “new”.
    super(id);
    this.email = email;                        En "strict: true" este código se comprueba frente a los
                                               campos para asegurar que está configurado correctamente.
    ...
  };
```

Formas de describir métodos de clase (y campos de función de flecha).

```
  setName(name: string) { this.name = name }
  verifyName = (name: string) => { ... }
```

Una función con 2 definiciones de sobrecarga.

```
  sync(): Promise<{ ... }>
  sync(cb: ((result: string) => void)): void
  sync(cb?: ((result: string) => void)): void | Promise<{ ... }> { ... }
```

Getters y Setters.

```
  get accountID() {}
  set accountID(value: string) {}
```

El acceso “private” es sólo a esta clase, “protected” permite a las subclases. Sólo se utiliza para la comprobación de tipos, “public” es el valor predeterminado.

```
  private makeRequest() { ... }
  protected handleRequest() { ... }
```

Campos / métodos estáticos.

```
  static #userCount = 0;
  static registerUser(user: User) { ... }
```

Bloques estáticos para configurar variables estáticos, ‘this’ se refiere a la clase estática.

```
  static { this.#userCount = -1 }
}
```

Genéricos

Declarar un tipo que puede cambiar en los métodos de su clase.

```
class Box<Type> {                  Tipo de clase parámetro.
  contents: Type
  constructor(value: Type) {       Se utiliza aquí.
    this.contents = value;
  }
}
const stringBox = new Box(”a package”)
```

MÁS

TYPESCRIPT

rortegag

Clases

Estas características son extensiones de lenguaje específicas de Typescript que quizá nunca lleguen a Javascript con la sintaxis actual.

Propiedades de parámetros

Una extensión específica de typescript para clases que establece automáticamente un campo de instancia al parámetro de entrada.

```
class Location {
  constructor(public x: number, public y: number) {}
}
const loc = new Location(20, 40);
loc.x // 20
loc.y // 40
```

Decoradores y atributos

Puede utilizar decoradores en clases, métodos de clases, accesorios, propiedades y parámetros de métodos.

```
import {
  Syncable, triggersSync, preferCache, required
} from "mylib"

@Syncable
class User {
  @triggersSync()
  save() { ... }

  @preferCache(false)
  get displayName() { ... }

  update(@required info: Partial<User>) { ... }
}
```

Puntos clave

Utilizado para describir la forma de los objetos, y puede ser ampliado por otros.
Casi todo en javascript es un objeto y la "interfaz" se construye para que coincida con su comportamiento en tiempo de ejecución.

Tipos primitivos incorporados

boolean, string, number, undefined, null,
any, unknown, never, void, bigint, symbol.

Objetos JS comunes incorporados

Date, Error, Array, Map, Set, Regexp, Promise.

Tipos literales

Objectos:
`{ campo: string }`
Funciones:
`(arg: number) => string`

Arrays:
`string[] or Array<string>`
Tupla:
`[string, number]`

Evita

Object, String, Number, Boolean

Clases abstractas

Una clase puede declararse como no implementable, pero como existente para ser subclasificada en el sistema de tipos. Al igual que los miembros de la clase.

```
abstract class Animal {
  abstract getName(): string;
  printName() {
    console.log("Hello, " + this.getName());
  }
}
class Dog extends Animal { getName(): { ... } }
```

Interfaces

Sintaxis común

Opcionalmente tomar propiedades de la interfaz o tipo existente.

```
interface JSONResponse extends Response, HTTPAble {
  version: number;
```

Comentario JSDoc adjunto para mostrar en los editores.

```
  /** En bytes */
  payloadSize: number;
```

Esta propiedad podría no estar en el objeto.

```
  outOfStock?: boolean;
```

Son dos formas de describir una propiedad que es una función.

```
  update: (retryTimes: number) => void;
  update(retryTimes: number): void;
```

Puede llamar a este objeto a través de () - { funciones en JS son objetos que pueden ser llamados }

```
  (): JSONResponse;
```

Puede utilizar new en el objeto que describe esta interfaz.

```
  new(s: string): JSONResponse;
```

Se supone que existe cualquier propiedad que no esté ya descrita, y todas las propiedades deben ser números.

```
  [key: string]: number;
```

Indica a typescript que una propiedad no puede ser modificada.

```
  readonly body: string;
}
```

MÁS

TYPESCRIPT

rortegag

Interfaces

Genéricos

Declarar un tipo que puede cambiar en su interfaz.

```
interface APICall<Response> parámetro de tipo {
  data: Response se utiliza aquí
}
```

Uso

```
const api: APICall<ArtworkCall> = ...
api.data // ArtworkCall
```

Puede restringir los tipos que se aceptan en el parámetro genérico mediante la palabra clave “extends”.

Establece una restricción en el tipo, lo que significa que sólo se pueden utilizar tipos con la propiedad ‘status’.

```
interface APICall<Response extends { status: number }> {
  data: Response
}
const api: APICall<ArtworkCall> = ...
api.data.status
```

Sobrecarga

Una interfaz invocable puede tener varias definiciones para distintos conjuntos de parámetros.

```
interface Expect {
  (matcher: boolean): string
  (matcher: string): boolean;
}
```

Get y Set

Los objetos pueden tener getters o setters personalizados.

```
interface Rules {
  get size(): number
  set size(value: number | string);
}
```

Uso

```
const r: Ruler = ...
r.size = 12
r.size = “36”
```

Ampliación por fusión

Las interfaces se fusionan, por lo que múltiples declaraciones añadirán nuevos campos a la definición del tipo.

```
interface APICall {
  data: Response
}
interface APICall {
  error?: Error
}
```

Conformidad de clases

Puede asegurarse de que una clase se ajusta a una interfaz mediante “implements”.

```
interface Syncable { sync(): void }
class Account implements Syncable { ... }
```

Tipos

Puntos clave

Su nombre completo es "type alias" y se utilizan para proporcionar nombres a los literales de tipo.
Soportan características del sistema de tipos más ricas que las interfaces.

Tipos vs Interfaces

- Las interfaces sólo pueden describir formas de objetos.
- Las interfaces pueden ampliarse declarándolas varias veces.
- En los tipos de rendimiento crítico las comprobaciones de comparación de interfaces pueden ser más rápidas.

MÁS

Pensar en los tipos como variables

Al igual que se pueden crear variables con el mismo nombre en diferentes ámbitos, un tipo tiene una semántica similar.

Construir con Utility Types

Typescript incluye muchos tipos globales que te ayudan a realizar tareas comunes en el sistema de tipos.

Para más información: https://www.typescriptlang.org/docs/

TYPESCRIPT

rortegag

Tipos

Sintaxis literal de objeto

```
type JSONResponse = {
  version: number;                          // Campo
  /** En bytes */                           // JSdocs adjuntos
  payloadSize: number;
  outOfStock?: boolean;                     // Campo opcional
  update: (retryTimes: number) => void;     // Campo función flecha
  update(retryTimes: number): void;         // Función
  (): JSONResponse                          // Tipo callback
  {key: string}: number;                    // Admite cualquier índice
  new (s: string): JSONResponse;            // Newable
  readonly body: string;                    // Propiedad de solo lectura
}
```

Primitivo

```
type SanitizedInput = string;
type MissingNo = 404;
```

Tupla

Una tupla es un array con una carcasa especial con tipos conocidos en índices específicos.

```
type Data = [
  location:  Location;
  timestamp: string;
];
```

Intersección

Una forma de fusionar/ampliar tipos.

```
type Location = { x: number } & { y: number }
```

A partir de valor

Reutilizar el tipo de un valor de tiempo de ejecución Javascript existente mediante el operador “typeof”.

```
const data = { ... }
type Data = typeof data
```

Retorno de función

Reutilizar el valor de retorno de una función como tipo.

```
const createFixtures = () => { ... }
type Fixtures = ReturnType<typeof createFixtures>

function test(fixture: Fixtures) {}
```

Objeto literal

```
type Location = {
  x:  number;
  y: number;
}
```

Unión

Describe un tipo que es una de muchas opciones, por ejemplo una lista de cadenas de textos conocidas.

```
type Size = “small” | “medium” | “large”
```

Indexación de tipos

Una forma de extraer y nombrar un subconjunto de un tipo.

```
type Response = { data: { ... } }
type Data = Response[”data”]
```

Desde módulo

```
const data: import(”./data”).data
```

MÁS

Para más información: https://www.typescriptlang.org/docs/

TYPESCRIPT

rortegag

Tipos

Estas funciones son excelentes para crear bibliotecas, describir código Javascript existente y es posible que rara vez recurra a ellas en aplicaciones basadas principalmente en Typescript.

Mapeados

Actúa como una sentencia map para el sistema de tipos, permitiendo que un tipo de entrada cambie la estructura del nuevo tipo.

```
const Artist = { name: string, bio: string}

type Subscriber<Type> = {
  Bucle a través de cada campo en el parámetro genérico de tipo "Type".
  [Property in keyof Type]:
  Establece el tipo como una función con el tipo original como parámetro.
  (newValue: Type[Property]) => void
}
type ArtistSub = Subscriber<Artist>
// { name: (nv: string) => void, bio: (nv: string) => void }
```

Condicionales

Actúan como "sentencias if" dentro del sistema de tipos. Se crean mediante genéricos y se utilizan habitualmente para reducir el número de opciones en una unión de tipos.

```
type HasFourLegs<Animal> = Animal extends { legs: 4 } ? Animal : never

type Animals = Bird | Dog | Ant | Wolf;
type FourLegs = HasFourLegs<Animals> //Dog | Wolf
```

Unión de plantillas

Una cadena de plantilla puede utilizarse para combinar y manipular texto dentro del sistema de tipos.

```
type SupportedLangs = “en” | “pt” | “zh”;
type FooterLocaleIDs = “header” | “footer”;

type AllLocaleIDs = `${SupportedLangs}_${FooterLocaleIDs}_id`;
  // “en_header_id” | “en_footer_id” | “pt_header_id” | “pt_footer_id” | “zh_header_id” | “zh_footer_id” |
```

Para más información: https://www.typescriptlang.org/docs/