Buenas Prácticas

Tipos generales

Number, String, Boolean, Symbol Y Object

No utilizar nunca los tipos Number, String, Boolean, Symbol, o Object estos tipos se refieren a objetos en caja no primitivos que casi nunca se usan apropiadamente en el código JavaScript.

```
/* WRONG */
function reverse(s: String): String;
```

No utilice los tipos number, string, boolean, y symbol.

```
/* OK */
function reverse(s: string): string;
```

En lugar de Object, use el tipo no primitivo object(agregado en TypeScript 2.2).

Genéricos

No siempre tiene un tipo genérico que no utiliza su parámetro de tipo. Vea más detalles en la página de preguntas frecuentes de TypeScript .

alguna

No lo use anycomo tipo a menos que esté en el proceso de migrar un proyecto de JavaScript a TypeScript. El compilador trata *efectivamente*any como "por favor, desactive la verificación de tipos para esto". Es similar a poner un @ts-ignorecomentario alrededor de cada uso de la variable. Esto puede ser muy útil cuando está migrando por primera vez un proyecto de JavaScript a TypeScript, ya que puede establecer el tipo para las cosas que aún no ha migrado any, pero en un proyecto de TypeScript completo está deshabilitando la verificación de tipos para cualquier parte de su programa que use eso.

En los casos en los que no sepa qué tipo desea aceptar, o cuando quiera aceptar algo porque lo pasará ciegamente sin interactuar con él, puede usar unknown.

Tipos de devolución de llamada

Tipos de devolución de llamadas

No use el tipo de retorno `any`para devoluciones de llamada cuyo valor se ignorará:

```
/* WRONG */
function fn(x: () => any) {
  x();
}
```

No utilizar el tipo de retorno `void`para las devoluciones de llamada cuyo valor será ignorado:

```
/* OK */
function fn(x: () => void) {
  x();
}
```

Por qué : Usar `void`es más seguro porque evita que uses accidentalmente el valor de retorno de `x`una manera no marcada:

```
function fn(x: () => void) {
  var k = x(); // oops! meant to do something else
  k.doSomething(); // error, but would be OK if the return type had been 'any'
}
```

Parámetros opcionales en devoluciones de llamada

No use parámetros opcionales en devoluciones de llamada a menos que realmente lo diga:

```
/* WRONG */
interface Fetcher {
  getObject(done: (data: any, elapsedTime?: number) => void): void;
}
```

Esto tiene un significado muy específico: la `done`devolución de llamada puede invocarse con 1 argumento o puede invocarse con 2 argumentos. El autor probablemente pretendía decir que a la devolución de llamada podría no importarle el `elapsedTime`parámetro, pero no es necesario que el parámetro sea opcional para lograr esto; siempre es legal proporcionar una devolución de llamada que acepte menos argumentos.

Hacer parámetros de devolución de llamada de escritura como no opcional:

```
/* OK */
interface Fetcher {
  getObject(done: (data: any, elapsedTime: number) => void): void;
}
```

Sobrecargas y devoluciones de llamada

No escriba sobrecargas separadas que difieran solo en la aridad de devolución de llamada:

```
/* WRONG */
declare function beforeAll(action: () => void, timeout?: number): void;
declare function beforeAll(
  action: (done: DoneFn) => void,
  timeout?: number
): void;
```

Hacer escribir una sola sobrecarga utilizando el máximo aridad:

```
/* OK */
declare function beforeAll(
  action: (done: DoneFn) => void,
  timeout?: number
): void;
```

Por qué : siempre es legal que una devolución de llamada ignore un parámetro, por lo que no hay necesidad de una sobrecarga más corta. Proporcionar primero una devolución de llamada más corta permite que se pasen funciones escritas incorrectamente porque coinciden con la primera sobrecarga.

Sobrecargas de funciones

Ordenar

No coloque sobrecargas más generales antes que sobrecargas más específicas:

```
/* WRONG */
declare function fn(x: any): any;
declare function fn(x: HTMLElement): number;
declare function fn(x: HTMLDivElement): string;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: any, wat?
```

Hacer sobrecargas ordenar poniendo las firmas más generales después de más firmas específicas:

```
/* OK */
declare function fn(x: HTMLDivElement): string;
declare function fn(x: HTMLElement): number;
declare function fn(x: any): any;

var myElem: HTMLDivElement;
```

```
var x = fn(myElem); // x: string, :)
```

Por qué : TypeScript elige la *primera sobrecarga coincidente* al resolver llamadas a funciones. Cuando una sobrecarga anterior es "más general" que una posterior, la última se oculta efectivamente y no se puede llamar.

Usar parámetros opcionales

No escriba varias sobrecargas que difieran solo en los parámetros finales:

```
/* WRONG */
interface Example {
  diff(one: string): number;
  diff(one: string, two: string): number;
  diff(one: string, two: string, three: boolean): number;
}
```

No utilizar parámetros opcionales siempre que sea posible:

```
/* OK */
interface Example {
  diff(one: string, two?: string, three?: boolean): number;
}
```

Tenga en cuenta que este colapso solo debe ocurrir cuando todas las sobrecargas tienen el mismo tipo de retorno.

Por qué : esto es importante por dos razones.

TypeScript resuelve la compatibilidad de firmas al ver si se puede invocar alguna firma del destino con los argumentos de la fuente, *y se permiten argumentos extraños* . Este código, por ejemplo, expone un error solo cuando la firma está escrita correctamente usando parámetros opcionales:

```
function fn(x: (a: string, b: number, c: number) => void) {}
var x: Example;
// When written with overloads, OK -- used first overload
// When written with optionals, correctly an error
fn(x.diff);
```

La segunda razón es cuando un consumidor utiliza la función de "comprobación nula estricta" de TypeScript. Debido a que los parámetros no especificados aparecen como `undefined`en JavaScript, generalmente está bien pasar un explícito `undefined`a una función con argumentos opcionales. Este código, por ejemplo, debería estar bien bajo nulos estrictos:

```
var x: Example;
// When written with overloads, incorrectly an error because of passing 'undefined' to 'string'
// When written with optionals, correctly OK
```

```
x.diff("something", true ? undefined : "hour");
```

Usar tipos de unión

No escriba sobrecargas que difieran según el tipo en una sola posición de argumento:

```
/* WRONG */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number): Moment;
  utcOffset(b: string): Moment;
}
```

Hacer uso de tipos de unión siempre que sea posible:

```
/* OK */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number | string): Moment;
}
```

Tenga en cuenta que aquí no lo hicimos opcional porque los tipos de devolución de las firmas son diferentes.

Por qué : esto es importante para las personas que están "transmitiendo" un valor a su función:

```
function fn(x: string): void;
function fn(x: number): void;
function fn(x: number | string) {
  // When written with separate overloads, incorrectly an error
  // When written with union types, correctly OK
  return moment().utcOffset(x);
}
```

Módulos fantasma

También llamado non-instantiated modules. En lugar de contaminar el espacio de nombres global con muchas interfaces, está bien crear un módulo que *solo* contenga *interfaces* . Esto no introduce una variable en el espacio de nombres global (consulte la seguridad en el ejemplo siguiente) y este módulo solo se puede utilizar para *tipos* .

```
// this pattern has 3 name in top level
interface NodeFoo { }
interface NodeBar { }
interface NodeBuzz { }

// this ghost module has 1 name in top level
declare namespace NodeJS {
    interface Foo { }
    interface Bar { }
```

```
    interface Buzz { }
}

// safety!
var n = NodeJS; // TS Error : Could not find symbol NodeJS
```

Esto también le permite abrir más personalización en módulos externos, ya que las interfaces declaradas *dentro de* las declaraciones de módulos externos no se pueden extender, por ejemplo, lo siguiente es bueno ya que las personas pueden personalizar `foo`más en otras definiciones de biblioteca.

```
// Usage when declaring an external module
declare module 'foo' {
    var foo: NodeJS.Foo;
    export = foo;
}
```

Ampliación de tipos integrados

No hay forma de agregar *miembros estáticos* a objetos nativos en este momento, ya que los `lib.d.ts` define como a `var Date:{/*members*/}`y `var` no son extensibles. Se proponen dos soluciones al equipo de TS. O bien utilizar interfaces en lugar de var en lib.d.ts (votación) y / o variables de maquillaje / clases de boca abierta (votación)

Para agregar miembros a *instancias* de tipos nativos, hay interfaces relevantes disponibles en, `lib.d.ts` por ejemplo,

```
// add members to Date instances
interface Date {
    newMember: number;
}

// usage
var foo = new Date();
foo.newMember = 123; // okay
```

Getter / Setter

En vez de :

```
declare function duration(value?: number): any;
```

mejor hacer:

```
declare function duration(): number;
declare function duration(value: number): void;
```

Fluido

Bastante autoexplicativo:

```
interface Something {
    foo(): Something;
    bar(): Something;
}
```

Firmas de devolución de llamada

No marque los argumentos de devolución de llamada como opcionales si el código de llamada los pasa cada vez. También deje la devolución como `any`si al código de llamada no le importara. Por ejemplo, en la siguiente declaración *buena*`foo` está el código de llamada que

estamos declarando que siempre llama con barY basy no le importa el valor de devolución de llamada:

```
declare function foo(callback: (bar: any, bas: any) => any): void;

// Usage is as expected by a JavaScript developer
foo(() => { });
foo((bar) => 123);
foo((bar, bas) => '');
```

Una forma *incorrecta* de modelarlo sería como se muestra a continuación, ya que impone restricciones que el código de llamada original no impone:

```
declare function foo(callback: (bar?: any, bas?: any) => void);
```

Sobrecarga de funciones

Un tipo de unión (anypor ahora) es necesario solo para bolsas de objetos de configuración. Para funciones / constructores, use la sobrecarga de funciones, por ejemplo

```
declare class Foo {
    constructor(foo: number);
    constructor(foo: string);
}

new Foo(123); // okay
new Foo('123'); // okay
new Foo(true); // Error
```

Orden de sobrecarga

El código con sobrecargas *debe* ordenarse manualmente desde la sobrecarga más ajustada / más específica hasta la más flexible. Vea el ejemplo a continuación:

```
interface Parent { x; }
interface Child extends Parent { y; }

function foo(p: Child): Child;
function foo(p: Parent): Parent;
function foo(p: any): any;
function foo(p: any) { return p; }

var a = foo({ x: 3, y: 4 }); // a: Child
var b = foo({ x: 5 }); // b: Parent

var y: any;
var c = foo(y); // c: any
```