普通类型

Number， String， Boolean和Object

*不要*使用如下类型`Number`，`String`，`Boolean`或`Object`。
这些类型指的是非原始的装盒对象，它们几乎没在JavaScript代码里正确地使用过。

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

*应该*使用类型`number`，`string`，and `boolean`。

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

使用非原始的`object`类型来代替`Object` （[TypeScript 2.2新增](../../release-notes/TypeScript 2.2.md#object-type)）

泛型

*不要*定义一个从来没使用过其类型参数的泛型类型。
了解详情TypeScript FAQ page。

回调函数类型

回调函数返回值类型

*不要*为返回值被忽略的回调函数设置一个`any`类型的返回值类型：

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

*应该*给返回值被忽略的回调函数设置`void`类型的返回值类型：

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

为什么：使用`void`相对安全，因为它防止了你不小心使用`x`的返回值：

```
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
}
```

回调函数里的可选参数

*不要*在回调函数里使用可选参数除非你真的要这么做：

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

这里有一种特殊的意义：`done`回调函数可能以1个参数或2个参数调用。
代码大概的意思是说这个回调函数不在乎是否有`elapsedTime`参数，
但是不需要把这个参数当成可选参数来达到此目的 --
因为总是允许提供一个接收较少参数的回调函数。

*应该*写出回调函数的非可选参数：

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

重载与回调函数

*不要*因为回调函数参数个数不同而写不同的重载：

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

*应该*只使用最大参数个数写一个重载：

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

为什么：回调函数总是可以忽略某个参数的，因此没必要为参数少的情况写重载。
参数少的回调函数首先允许错误类型的函数被传入，因为它们匹配第一个重载。

函数重载

顺序

*不要*把一般的重载放在精确的重载前面：

```
/* 错误 */
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?
```

*应该*排序重载令精确的排在一般的之前：

```
/* 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, :)
```

为什么：TypeScript会选择*第一个匹配到的重载*当解析函数调用的时候。
当前面的重载比后面的“普通”，那么后面的被隐藏了不会被调用。

使用可选参数

*不要*为仅在末尾参数不同时写不同的重载：

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

*应该*尽可能使用可选参数：

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

注意这在所有重载都有相同类型的返回值时会不好用。

为什么：有两种生要的原因。

TypeScript解析签名兼容性时会查看是否某个目标签名能够使用源的参数调用，
且允许外来参数。
下面的代码暴露出一个bug，当签名被正确的使用可选参数书写时：

```
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);
```

第二个原因是当使用了TypeScript“严格检查null”特性时。
因为没有指定的参数在JavaScript里表示为`undefined`，通常显示地为可选参数传入一个`undefined`。
这段代码在严格null模式下可以工作：

```
var x: Example;
// When written with overloads, incorrectly an error because of passing 'undefi
// When written with optionals, correctly OK
x.diff("something", true ? undefined : "hour");
```

使用联合类型

*不要*为仅在某个位置上的参数类型不同的情况下定义重载：

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

*应该*尽可能地使用类型类型：

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

注意我们没有让`b`成为可选的，因为签名的返回值类型不同。

为什么：This is important for people who are "passing through" a value to your function:

```
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);
}
```