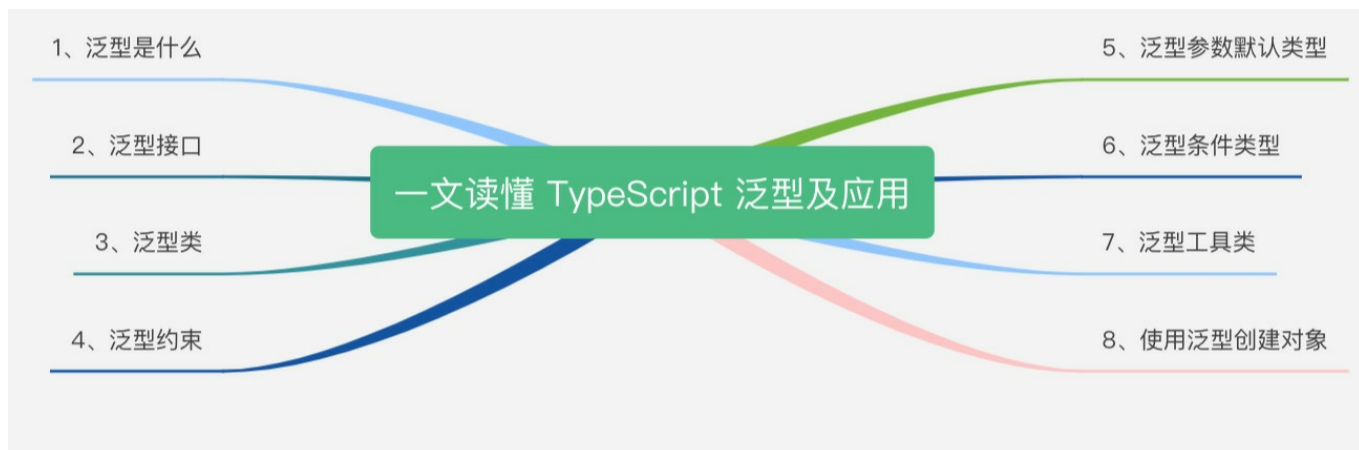


## 一文读懂 TypeScript 泛型及应用（7.8K字）

觉得 TypeScript 泛型有点难，想系统学习 TypeScript 泛型相关知识的小伙伴们看过来，**本文从八个方面入手，全方位带你一步步学习 TypeScript 中泛型**，详细的内容大纲请看下图：



**动静（图）结合**，在泛型学习之路助你一臂之力，还在犹豫什么，赶紧开启 TypeScript 泛型的学习之旅吧！

想入门 TypeScript 的小伙伴看过来，阿宝哥特意为你们准备的 —— [1.2W字 | 了不起的 TypeScript 入门教程](#)（1027+ 个👍）教程。

### 一、泛型是什么

软件工程中，我们不仅要创建一致的定义良好的 API，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

**在像 C# 和 Java 这样的语言中，可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。这样用户就可以以自己的数据类型来使用组件。**

设计泛型的关键目的是在成员之间提供有意义的约束，这些成员可以是：类的实例成员、类的方法、函数参数和函数返回值。

为了便于大家更好地理解上述的内容，我们来举个例子，在这个例子中，我们将一步步揭示泛型的作用。首先我们来定义一个通用的 identity 函数，该函数接收一个参数并直接返回它：

```
function identity (value) {  
  return value;  
}
```

```
console.log(identity(1)) // 1
```

现在，我们将 identity 函数做适当的调整，以支持 TypeScript 的 Number 类型的参数：

```
function identity (value: Number) : Number {  
  return value;  
}
```

```
console.log(identity(1)) // 1
```

这里 identity 的问题是我们将 Number 类型分配给参数和返回类型，使该函数仅可用于该原始类型。但该函数并不是可扩展或通用的，很明显这并不是我们所希望的。

我们确实可以把 Number 换成 any，我们失去了定义应该返回哪种类型的能力，并且在这个过程中使编译器失去了类型保护的作用。我们的目标是让 identity 函数可以适用于任何特定的类型，为了实现这个目标，我们可以使用泛型来解决这个问题，具体实现方式如下：

```
function identity <T>(value: T) : T {  
    return value;  
}
```

```
console.log(identity<Number>(1)) // 1
```

对于刚接触 TypeScript 泛型的读者来说，首次看到 <T> 语法会感到陌生。但这没什么可担心的，就像传递参数一样，我们传递了我们想要用于特定函数调用的类型。



参考上面的图片，当我们调用 `identity<Number>(1)`，Number 类型就像参数 1 一样，它将在出现 T 的任何位置填充该类型。图中 <T> 内部的 T 被称为类型变量，它是我们希望传递给 identity 函数的类型占位符，同时它被分配给 value 参数用来代替它的类型：此时 T 充当的是类型，而不是特定的 Number 类型。

其中 T 代表 **Type**，在定义泛型时通常用作第一个类型变量名称。但实际上 T 可以用任何有效名称代替。除了 T 之外，以下是常见泛型变量代表的意思：

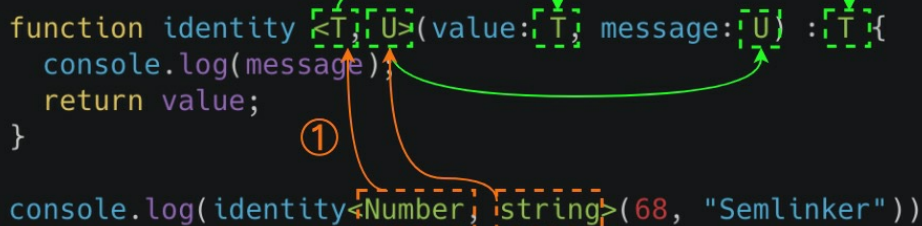
- K (Key)：表示对象中的键类型；
- V (Value)：表示对象中的值类型；
- E (Element)：表示元素类型。

其实并不是只能定义一个类型变量，我们可以引入希望定义的任何数量的类型变量。比如我们引入一个新的类型变量 U，用于扩展我们定义的 identity 函数：

```
function identity <T, U>(value: T, message: U) : T {  
    console.log(message);  
    return value;  
}
```

```
console.log(identity<Number, string>(68, "Semlinker"));
```

类型像传递给函数的参数一样传递



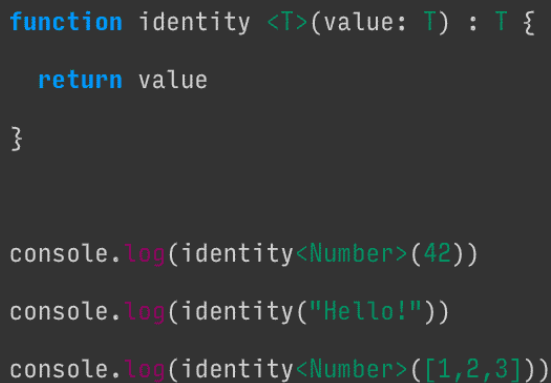
```
function identity <T, U>(value: T, message: U) : T {  
  console.log(message);  
  return value;  
}  
  
console.log(identity<Number, string>(68, "Semlinker"))
```

除了为类型变量显式设定值之外，一种更常见的做法是使编译器自动选择这些类型，从而使代码更简洁。我们可以完全省略尖括号，比如：

```
function identity <T, U>(value: T, message: U) : T {  
  console.log(message);  
  return value;  
}
```

```
console.log(identity(68, "Semlinker"));
```

对于上述代码，编译器足够聪明，能够知道我们的参数类型，并将它们赋值给 T 和 U，而不需要开发人员显式指定它们。下面我们来看张动图，直观地感受一下类型传递的过程：



```
function identity <T>(value: T) : T {  
  return value  
}  
  
console.log(identity<Number>(42))  
console.log(identity("Hello!"))  
console.log(identity<Number>([1,2,3]))
```

(图片来源: <https://medium.com/better-programming/typescript-generics-90be93d8c292>)

感谢 @仑 (前端搬砖党) 指出，该动图有bug。

动态图最后一句错了吗？`console.log(identity([1,2,3]))`这里注入类型应该是`number[]`吧？

如你所见，该函数接收你传递给它的任何类型，使得我们可以为不同类型创建可重用的组件。现在我们再来看一下 `identity` 函数：

```
function identity <T, U>(value: T, message: U) : T {  
  console.log(message);
```

```
    return value;
}
```

相比之前定义的 identity 函数，新的 identity 函数增加了一个类型变量 U，但该函数的返回类型我们仍然使用 T。如果我们想要返回两种类型的对象该怎么办呢？针对这个问题，我们有多种方案，其中一种就是使用元组，即为元组设置通用的类型：

```
function identity <T, U>(value: T, message: U) : [T, U] {
    return [value, message];
}
```

虽然使用元组解决了上述的问题，但有没有其它更好的方案呢？答案是有的，你可以使用泛型接口。

## 二、泛型接口

为了解决上面提到的问题，首先让我们创建一个用于的 identity 函数通用 Identities 接口：

```
interface Identities<V, M> {
    value: V,
    message: M
}
```

在上述的 Identities 接口中，我们引入了类型变量 V 和 M，来进一步说明有效的字母都可以用于表示类型变量，之后我们就可以将 Identities 接口作为 identity 函数的返回类型：

```
function identity<T, U> (value: T, message: U): Identities<T, U> {
    console.log(value + ": " + typeof (value));
    console.log(message + ": " + typeof (message));
    let identities: Identities<T, U> = {
        value,
        message
    };
    return identities;
}
```

```
console.log(identity(68, "Semlinker"));
```

以上代码成功运行后，在控制台会输出以下结果：

```
68: number
Semlinker: string
{value: 68, message: "Semlinker"}
```

泛型除了可以应用在函数和接口之外，它也可以应用在类中，下面我们就来看一下在类中如何使用泛型。

## 三、泛型类

在类中使用泛型也很简单，我们只需要在类名后面，使用 <T, ...> 的语法定义任意多个类型变量，具体示例如下：

```
interface GenericInterface<U> {
    value: U
    getIdentity: () => U
}
```

```

class IdentityClass<T> implements GenericInterface<T> {
  value: T

  constructor(value: T) {
    this.value = value
  }

  getIdentity(): T {
    return this.value
  }
}

const myNumberClass = new IdentityClass<Number>(68);
console.log(myNumberClass.getIdentity()); // 68

const myStringClass = new IdentityClass<string>("Semlinker!");
console.log(myStringClass.getIdentity()); // Semlinker!

```

接下来我们以实例化 myNumberClass 为例，来分析一下其调用过程：

- 在实例化 IdentityClass 对象时，我们传入 Number 类型和构造函数参数值 68；
- 之后在 IdentityClass 类中，类型变量 T 的值变成 Number 类型；
- IdentityClass 类实现了 GenericInterface<T>，而此时 T 表示 Number 类型，因此等价于该类实现了 GenericInterface<Number> 接口；
- 而对于 GenericInterface<U> 接口来说，类型变量 U 也变成了 Number。这里我有意使用不同的变量名，以表明类型值沿链向上传播，且与变量名无关。

泛型类可确保在整个类中一致地使用指定的数据类型。比如，你可能已经注意到在使用 Typescript 的 React 项目中使用了以下约定：

```

type Props = {
  className?: string
  ...
};

type State = {
  submitted?: bool
  ...
};

class MyComponent extends React.Component<Props, State> {
  ...
}

```

在以上代码中，我们将泛型与 React 组件一起使用，以确保组件的 props 和 state 是类型安全的。

相信看到这里一些读者会有疑问，我们在什么时候需要使用泛型呢？通常在决定是否使用泛型时，我们有以下两个参考标准：

- 当你的函数、接口或类将处理多种数据类型时；
- 当函数、接口或类在多个地方使用该数据类型时。

很有可能你没有办法保证在项目早期就使用泛型的组件，但是随着项目的发展，组件的功能通常会被扩展。这种增加的可扩展性最终很可能会满足上述两个条件，在这种情况下，引入泛型将比复制组件来满足一系列数据类型更干净。

我们将在本文的后面探讨更多满足这两个条件的用例。不过在这样做之前，让我们先介绍一下 Typescript 泛型提供的其他功能。

## 四、泛型约束

有时我们可能希望限制每个类型变量接受的类型数量，这就是泛型约束的作用。下面我们来举几个例子，介绍一下如何使用泛型约束。

### 4.1 确保属性存在

有时候，我们希望类型变量对应的类型上存在某些属性。这时，除非我们显式地将特定属性定义为类型变量，否则编译器不会知道它们的存在。

一个很好的例子是在处理字符串或数组时，我们会假设 `length` 属性是可用的。让我们再次使用 `identity` 函数并尝试输出参数的长度：

```
function identity<T>(arg: T): T {
  console.log(arg.length); // Error
  return arg;
}
```

在这种情况下，编译器将不会知道 `T` 确实含有 `length` 属性，尤其是在可以将任何类型赋给类型变量 `T` 的情况下。我们需要做的就是让类型变量 `extends` 一个含有我们所需属性的接口，比如这样：

```
interface Length {
  length: number;
}

function identity<T extends Length>(arg: T): T {
  console.log(arg.length); // 可以获取length属性
  return arg;
}
```

`T extends Length` 用于告诉编译器，我们支持已经实现 `Length` 接口的任何类型。之后，当我们使用不含有 `length` 属性的对象作为参数调用 `identity` 函数时，TypeScript 会提示相关的错误信息：

```
identity(68); // Error
// Argument of type '68' is not assignable to parameter of type 'Length'.(2345)
```

此外，我们还可以使用 `,` 号来分隔多种约束类型，比如：`<T extends Length, Type2, Type3>`。而对于上述的 `length` 属性问题来说，如果我们显式地将变量设置为数组类型，也可以解决该问题，具体方式如下：

```
function identity<T>(arg: T[]): T[] {
  console.log(arg.length);
  return arg;
}

// or
function identity<T>(arg: Array<T>): Array<T> {
```

```
    console.log(arg.length);
    return arg;
}
```

## 4.2 检查对象上的键是否存在

泛型约束的另一个常见的使用场景就是检查对象上的键是否存在。不过在看具体示例之前，我们得了解一下 `keyof` 操作符，**`keyof` 操作符是在 TypeScript 2.1 版本引入的，该操作符可以用于获取某种类型的所有键，其返回类型是联合类型。**“耳听为虚，眼见为实”，我们来举个 `keyof` 的使用示例：

```
interface Person {
    name: string;
    age: number;
    location: string;
}

type K1 = keyof Person; // "name" | "age" | "location"
type K2 = keyof Person[]; // number | "length" | "push" | "concat" | ...
type K3 = keyof { [x: string]: Person }; // string | number
```

通过 `keyof` 操作符，我们就可以获取指定类型的所有键，之后我们就可以结合前面介绍的 `extends` 约束，即限制输入的属性名包含在 `keyof` 返回的联合类型中。具体的使用方式如下：

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}
```

在以上的 `getProperty` 函数中，我们通过 `K extends keyof T` 确保参数 `key` 一定是对象中含有的键，这样就不会发生运行时错误。这是一个类型安全的解决方案，与简单调用 `let value = obj[key];` 不同。

下面我们来看一下如何使用 `getProperty` 函数：

```
enum Difficulty {
    Easy,
    Intermediate,
    Hard
}

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

let tsInfo = {
    name: "Typescript",
    supersetOf: "Javascript",
    difficulty: Difficulty.Intermediate
}

let difficulty: Difficulty =
    getProperty(tsInfo, 'difficulty'); // OK

let supersetOf: string =
    getProperty(tsInfo, 'superset_of'); // Error
```

在以上示例中，对于 `getProperty(tslInfo, 'superset_of')` 这个表达式，TypeScript 编译器会提示以下错误信息：

```
Argument of type '"superset_of"' is not assignable to parameter of type  
"difficulty" | "name" | "supersetOf".(2345)
```

很明显通过使用泛型约束，在编译阶段我们就可以提前发现错误，大大提高了程序的健壮性和稳定性。接下来，我们来介绍一下泛型参数默认类型。

## 五、泛型参数默认类型

在 TypeScript 2.3 以后，我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推断出类型时，这个默认类型就会起作用。

泛型参数默认类型与普通函数默认值类似，对应的语法很简单，即 `<T=Default Type>`，对应的使用示例如下：

```
interface A<T=string> {  
  name: T;  
}
```

```
const strA: A = { name: "Semlinker" };  
const numB: A<number> = { name: 101 };
```

泛型参数的默认类型遵循以下规则：

- 有默认类型的类型参数被认为是可选的。
- 必选的类型参数不能在可选的类型参数后。
- 如果类型参数有约束，类型参数的默认类型必须满足这个约束。
- 当指定类型实参时，你只需要指定必选类型参数的类型实参。未指定的类型参数会被解析为它们的默认类型。
- 如果指定了默认类型，且类型推断无法选择一个候选类型，那么将使用默认类型作为推断结果。
- 一个被现有类或接口合并的类或者接口的声明可以为现有类型参数引入默认类型。
- 一个被现有类或接口合并的类或者接口的声明可以引入新的类型参数，只要它指定了默认类型。

## 六、泛型条件类型

在 TypeScript 2.8 中引入了条件类型，使得我们可以根据某些条件得到不同的类型，这里所说的条件是类型兼容性约束。尽管以上代码中使用了 `extends` 关键字，也不一定要强制满足继承关系，而是检查是否满足结构兼容性。

条件类型会以一个条件表达式进行类型关系检测，从而在两种类型中选择其一：

`T extends U ? X : Y`

以上表达式的意思是：若 `T` 能够赋值给 `U`，那么类型是 `X`，否则为 `Y`。在条件类型表达式中，我们通常会结合 `infer` 关键字，实现类型抽取：

```
interface Dictionary<T = any> {  
  [key: string]: T;  
}
```

```
type StrDict = Dictionary<string>
```



```
type DictMember<T> = T extends Dictionary<infer V> ? V : never
type StrDictMember = DictMember<StrDict> // string
```

在上面示例中，当类型 T 满足 T extends Dictionary 约束时，我们会使用 infer 关键字声明了一个类型变量 V，并返回该类型，否则返回 never 类型。

在 TypeScript 中，never 类型表示的是那些永不存在的值的类型。例如，never 类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型。

另外，需要注意的是，没有类型是 never 的子类型或可以赋值给 never 类型（除了 never 本身之外）。即使 any 也不可以赋值给 never。

除了上述的应用外，利用条件类型和 infer 关键字，我们还可以方便地实现获取 Promise 对象的返回值类型，比如：

```
async function stringPromise() {
  return "Hello, Semlinker!";
}
```

```
interface Person {
  name: string;
  age: number;
}
```

```
async function personPromise() {
  return { name: "Semlinker", age: 30 } as Person;
}
```

```
type PromiseType<T> = (args: any[]) => Promise<T>;
type UnPromisify<T> = T extends PromiseType<infer U> ? U : never;
```

```
type extractStringPromise = UnPromisify<typeof stringPromise>; // string
type extractPersonPromise = UnPromisify<typeof personPromise>; // Person
```

## 七、泛型工具类型

为了方便开发者 TypeScript 内置了一些常用的工具类型，比如 Partial、Required、Readonly、Record 和 ReturnType 等。出于篇幅考虑，这里我们只简单介绍其中几个常用的工具类型。

### 7.1 Partial

Partial<T> 的作用就是将某个类型里的属性全部变为可选项？。

**定义：**

```
/**
 * node_modules/typescript/lib/lib.es5.d.ts
 * Make all properties in T optional
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

在以上代码中，首先通过 keyof T 拿到 T 的所有属性名，然后使用 in 进行遍历，将值赋给 P，最后通

过 T[P] 取得相应的属性值。中间的 ? 号，用于将所有属性变为可选。

### 示例：

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
  title: "organize desk",
  description: "clear clutter"
};

const todo2 = updateTodo(todo1, {
  description: "throw out trash"
});
```

在上面的 updateTodo 方法中，我们利用 Partial<T> 工具类型，定义 fieldsToUpdate 的类型为 Partial<Todo>，即：

```
{
  title?: string | undefined;
  description?: string | undefined;
}
```

## 7.2 Record

Record<K extends keyof any, T> 的作用是将 K 中所有的属性的值转化为 T 类型。

### 定义：

```
/**
 * node_modules/typescript/lib/lib.es5.d.ts
 * Construct a type with a set of properties K of type T
 */
type Record<K extends keyof any, T> = {
  [P in K]: T;
};
```

### 示例：

```
interface PageInfo {
  title: string;
}

type Page = "home" | "about" | "contact";

const x: Record<Page, PageInfo> = {
  about: { title: "about" },
```

```
    contact: { title: "contact" },
    home: { title: "home" }
};
```

### 7.3 Pick

Pick<T, K extends keyof T> 的作用是将某个类型中的子属性挑出来，变成包含这个类型部分属性的子类型。

**定义：**

```
// node_modules/typescript/lib/lib.es5.d.ts

/**
 * From T, pick a set of properties whose keys are in the union K
 */
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};
```

**示例：**

```
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
    title: "Clean room",
    completed: false
};
```

### 7.4 Exclude

Exclude<T, U> 的作用是将某个类型中属于另一个的类型移除掉。

**定义：**

```
// node_modules/typescript/lib/lib.es5.d.ts

/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;
```

如果 T 能赋值给 U 类型的话，那么就会返回 never 类型，否则返回 T 类型。最终实现的效果就是将 T 中某些属于 U 的类型移除掉。

**示例：**

```
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
```

```
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number
```

## 7.5 ReturnType

ReturnType<T> 的作用是用于获取函数 T 的返回类型。

**定义:**

```
// node_modules/typescript/lib/lib.es5.d.ts
```

```
/**
 * Obtain the return type of a function type
 */
type ReturnType<T extends (...args: any) => any> = T extends (...args: any) => infer R ? R : any;
```

**示例:**

```
type T0 = ReturnType<() => string>; // string
type T1 = ReturnType<(s: string) => void>; // void
type T2 = ReturnType<<T>() => T>; // {}
type T3 = ReturnType<<T extends U, U extends number[]>() => T>; // number[]
type T4 = ReturnType<any>; // any
type T5 = ReturnType<never>; // any
type T6 = ReturnType<string>; // Error
type T7 = ReturnType<Function>; // Error
```

简单介绍了泛型工具类型，最后我们来介绍如何使用泛型来创建对象。

## 八、使用泛型创建对象

### 8.1 构造签名

有时，泛型类可能需要基于传入的泛型 T 来创建其类型相关的对象。比如：

```
class FirstClass {
  id: number | undefined;
}

class SecondClass {
  name: string | undefined;
}

class GenericCreator<T> {
  create(): T {
    return new T();
  }
}
```

```
const creator1 = new GenericCreator<FirstClass>();
const firstClass: FirstClass = creator1.create();
```

```
const creator2 = new GenericCreator<SecondClass>();
const secondClass: SecondClass = creator2.create();
```

在以上代码中，我们定义了两个普通类和一个泛型类 `GenericCreator<T>`。在通用的 `GenericCreator` 泛型类中，我们定义了一个名为 `create` 的成员方法，该方法会使用 `new` 关键字来调用传入的实际类型的构造函数，来创建对应的对象。但可惜的是，以上代码并不能正常运行，对于以上代码，在 **TypeScript v3.9.2** 编译器下会提示以下错误：

'T' only refers to a type, but is being used as a value here.

这个错误的意思是：T 类型仅指类型，但此处被用作值。那么如何解决这个问题呢？根据 TypeScript 文档，为了使通用类能够创建 T 类型的对象，我们需要通过其构造函数来引用 T 类型。对于上述问题，在介绍具体的解决方案前，我们先来介绍一下构造签名。

在 TypeScript 接口中，你可以使用 `new` 关键字来描述一个构造函数：

```
interface Point {  
  new (x: number, y: number): Point;  
}
```

以上接口中的 `new (x: number, y: number)` 我们称之为构造签名，其语法如下：

*ConstructSignature:*

`new TypeParametersopt ( ParameterListopt ) TypeAnnotationopt`

在上述的构造签名中，`TypeParametersopt`、`ParameterListopt` 和 `TypeAnnotationopt` 分别表示：可选的类型参数、可选的参数列表和可选的类型注解。与该语法相对应的几种常见的使用形式如下：

```
new C  
new C ( ... )  
new C < ... > ( ... )
```

介绍完构造签名，我们再来介绍一个与之相关的概念，即构造函数类型。

## 8.2 构造函数类型

在 TypeScript 语言规范中这样定义构造函数类型：

An object type containing one or more construct signatures is said to be a **constructor type**. Constructor types may be written using constructor type literals or by including construct signatures in object type literals.

通过规范中的描述信息，我们可以得出以下结论：

- 包含一个或多个构造签名的对象类型被称为构造函数类型；
- 构造函数类型可以使用构造函数类型字面量或包含构造签名的对象类型字面量来编写。

那么什么是构造函数类型字面量呢？构造函数类型字面量是包含单个构造函数签名的对象类型的简写。具体来说，构造函数类型字面量的形式如下：

```
new < T1, T2, ... > ( p1, p2, ... ) => R
```

该形式与以下对象类型字面量是等价的：

```
{ new < T1, T2, ... > ( p1, p2, ... ) : R }
```

下面我们来举个实际的示例：

```
// 构造函数类型字面量
new (x: number, y: number) => Point
```

等价于以下对象类型字面量：

```
{
  new (x: number, y: number): Point;
}
```

### 8.3 构造函数类型的应用

在介绍构造函数类型的应用前，我们先来看个例子：

```
interface Point {
  new (x: number, y: number): Point;
  x: number;
  y: number;
}
```

```
class Point2D implements Point {
  readonly x: number;
  readonly y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}
```

```
const point: Point = new Point2D(1, 2);
```

对于以上的代码，TypeScript 编译器会提示以下错误信息：

```
Class 'Point2D' incorrectly implements interface 'Point'.
Type 'Point2D' provides no match for the signature 'new (x: number, y: number): Point'.
```

相信很多刚接触 TypeScript 不久的小伙伴都会遇到上述的问题。要解决这个问题，我们就需要把对前面定义的 Point 接口进行分离，即把接口的属性和构造函数类型进行分离：

```
interface Point {
  x: number;
  y: number;
}

interface PointConstructor {
  new (x: number, y: number): Point;
}
```

完成接口拆分之后，除了前面已经定义的 Point2D 类之外，我们又定义了一个 newPoint 工厂函数，该函数用于根据传入的 PointConstructor 类型的构造函数，来创建对应的 Point 对象。

```
class Point2D implements Point {
  readonly x: number;
  readonly y: number;
```

```

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

```

```

function newPoint(
    pointConstructor: PointConstructor,
    x: number,
    y: number
): Point {
    return new pointConstructor(x, y);
}

```

```

const point: Point = newPoint(Point2D, 1, 2);

```

## 8.4 使用泛型创建对象

了解完构造签名和构造函数类型之后，下面我们来开始解决上面遇到的问题，首先我们需要重构一下 create 方法，具体如下所示：

```

class GenericCreator<T> {
    create<T>(c: { new (): T }): T {
        return new c();
    }
}

```

在以上代码中，我们重新定义了 create 成员方法，根据该方法的签名，我们可以知道该方法接收一个参数，其类型是构造函数类型，且该构造函数不包含任何参数，调用该构造函数后，会返回类型 T 的实例。

如果构造函数含有参数的话，比如包含一个 number 类型的参数时，我们可以这样定义 create 方法：

```

create<T>(c: { new(a: number): T; }, num: number): T {
    return new c(num);
}

```

更新完 GenericCreator 泛型类，我们就可以使用下面的方式来创建 FirstClass 和 SecondClass 类的实例：

```

const creator1 = new GenericCreator<FirstClass>();
const firstClass: FirstClass = creator1.create(FirstClass);

const creator2 = new GenericCreator<SecondClass>();
const secondClass: SecondClass = creator2.create(SecondClass);

```

## 九、参考资料

- [typescript-generics](#)
- [typescript-generics-explained](#)
- [typescript-tip-of-the-week-generics](#)



微信号



关注  
全栈修仙之路，  
一起学习与成长

公众号

