全栈修仙之路
重学TS 1.0

阿宝哥

前言

你好，我是阿宝哥。首先简单介绍下我自己，2017 年 3 月份开始在思否写 Angular 修仙之路专栏，目前已输出 **149** 篇原创文章，曾获得思否年度优秀文章作者及两季 Top Writer。

2020 年开始写 “重学 TS”、“玩转前端”、“你不知道的 XXX” 和 “一文读懂 XXX” 系列专题。平常活跃在各个开发社区，这里分享主要的社区地址：

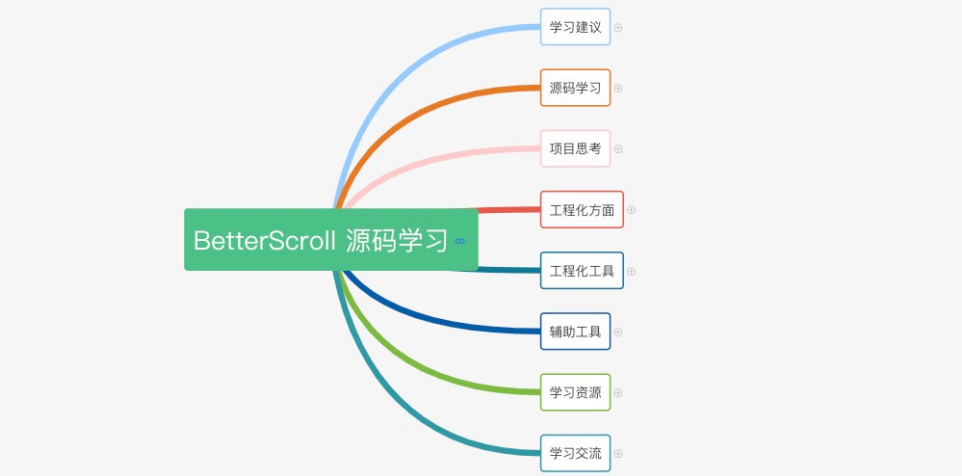
1、掘金（LV5）：https://juejin.im/user/764915822103079

2、思否（12.7K）：https://segmentfault.com/a/1190000008754631

3、个人博客：https://www.semlinker.com/

《重学TS v1.0》 是今年阿宝哥发布的第三本电子书，前两本分别是《前端进阶篇 v1.1》（下载量近 5900） 和《了不起的 TS 和 Deno》（下载量近 2100），这里衷心感谢大家对阿宝哥的认可与支持。

在学习 TS 的过程中，阿宝哥发现阅读优秀的 TS 开源项目是一种不错的进阶方式。因此在团队内策划了 TS 项目源码学习的专题，目前已经学完 4 个开源项目。以下是阿宝哥在学习 better-scroll（13.2K Stars） 开源项目整理的思维导图，感兴趣的小伙伴可以了解一下。



好的，回到正题。在阅读本书过程中，如果遇到问题或想进一步跟阿宝哥做技术交流，可以添加阿宝哥的个人微信号，备注 **“ts”**。之后阿宝哥会拉你进入本书的读者交流群，当然你也可以选择自助入群。

另外，如果你在学习、成长过程中遇到什么问题，也可以添加我的微信一起交流。

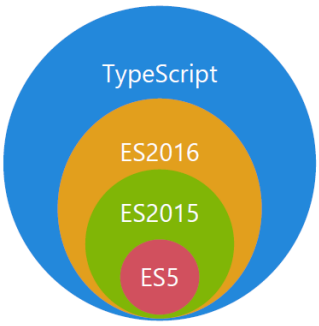
目录

第一章 TypeScript 快速入门

一、TypeScript 是什么

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。

TypeScript 提供最新的和不断发展的 JavaScript 特性，包括那些来自 2015 年的 ECMAScript 和未来的提案中的特性，比如异步功能和 Decorators，以帮助建立健壮的组件。下图显示了 TypeScript 与 ES5、ES2015 和 ES2016 之间的关系：



1.1 TypeScript 与 JavaScript 的区别

TypeScript	JavaScript
JavaScript 的超集用于解决大型项目的代码复杂性	一种脚本语言，用于创建动态网页
可以在编译期间发现并纠正错误	作为一种解释型语言，只能在运行时发现错误
强类型，支持静态和动态类型	弱类型，没有静态类型选项
最终被编译成 JavaScript 代码，使浏览器可以理解	可以直接在浏览器中使用
支持模块、泛型和接口	不支持模块，泛型或接口
社区的支持仍在增长，而且还不是很大	大量的社区支持以及大量文档和解决问题的支持

1.2 获取 TypeScript

命令行的 TypeScript 编译器可以使用 npm 包管理器来安装。

1.安装 TypeScript

```
$ npm install -g typescript
```

2.验证 TypeScript

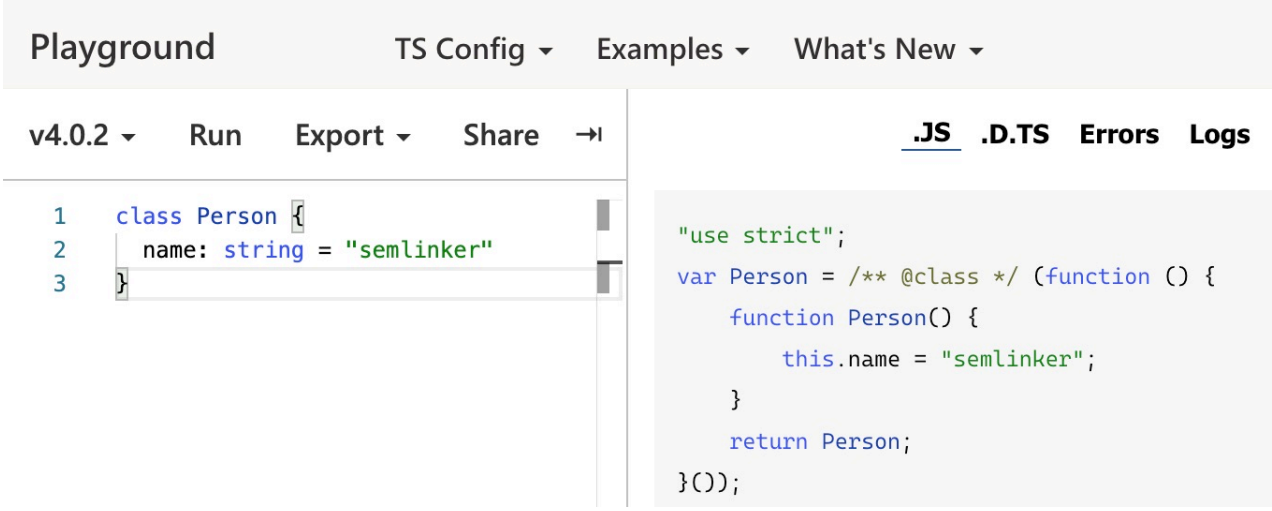
```
$ tsc -v
# Version 4.0.2
```

3.编译 TypeScript 文件

```
$ tsc helloworld.ts
# helloworld.ts => helloworld.js
```

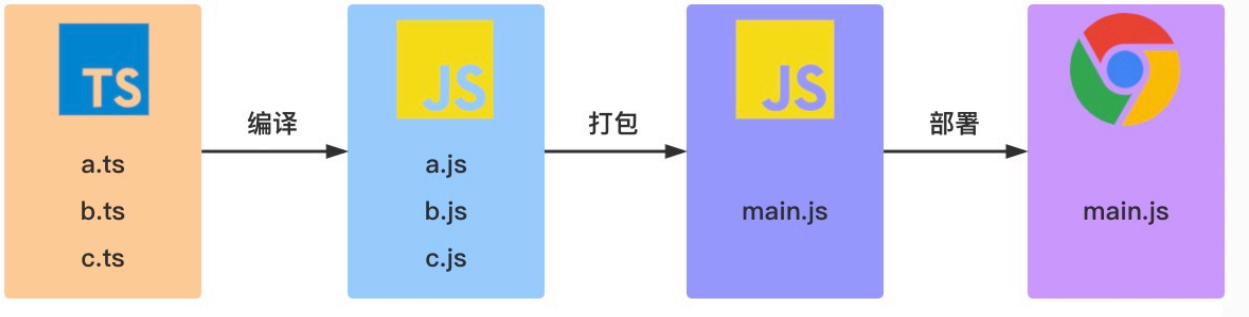
当然，对刚入门 TypeScript 的小伙伴来说，也可以不用安装 `typescript`，而是直接使用线上的 TypeScript Playground 来学习新的语法或新特性。通过配置 **TS Config** 的 Target，可以设置不同的编译目标，从而编译生成不同的目标代码。

下图示例中所设置的编译目标是 ES5：



（图片来源：https://www.typescriptlang.org/play）

1.3 典型 TypeScript 工作流程



如你所见，在上图中包含 3 个 ts 文件：a.ts、b.ts 和 c.ts。这些文件将被 TypeScript 编译器，根据配置的编译选项编译成 3 个 js 文件，即 a.js、b.js 和 c.js。对于大多数使用 TypeScript 开发的 Web 项目，我们还会对编译生成的 js 文件进行打包处理，然后在进行部署。

1.4 TypeScript 初体验

新建一个 `hello.ts` 文件，并输入以下内容：

```
function greet(person: string) {
  return 'Hello, ' + person;
}

console.log(greet("TypeScript"));
```

然后执行 `tsc hello.ts` 命令，之后会生成一个编译好的文件 `hello.js`：

```
"use strict";
function greet(person) {
  return 'Hello, ' + person;
}
console.log(greet("TypeScript"));
```

观察以上编译后的输出结果，我们发现 `person` 参数的类型信息在编译后被擦除了。TypeScript 只会在编译阶段对类型进行静态检查，如果发现有错误，编译时就会报错。而在运行时，编译生成的 JS 与普通的 JavaScript 文件一样，并不会进行类型检查。

二、TypeScript 基础类型

2.1 Boolean 类型

```
let isDone: boolean = false;
// ES5：var isDone = false;
```

2.2 Number 类型

```
let count: number = 10;
// ES5：var count = 10;
```

2.3 String 类型

```
let name: string = "semliker";
// ES5：var name = 'semlinker';
```

2.4 Symbol 类型

```
const sym = Symbol();
let obj = {
  [sym]: "semlinker",
};

console.log(obj[sym]); // semlinker
```

2.5 Array 类型

```
let list: number[] = [1, 2, 3];
// ES5：var list = [1,2,3];

let list: Array<number> = [1, 2, 3]; // Array<number>泛型语法
// ES5：var list = [1,2,3];
```

2.6 Enum 类型

使用枚举我们可以定义一些带名字的常量。 使用枚举可以清晰地表达意图或创建一组有区别的用例。TypeScript 支持数字的和基于字符串的枚举。

1.数字枚举

```
enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST,
}

let dir: Direction = Direction.NORTH;
```

默认情况下，NORTH 的初始值为 0，其余的成员会从 1 开始自动增长。换句话说，Direction.SOUTH 的值为 1，Direction.EAST 的值为 2，Direction.WEST 的值为 3。

以上的枚举示例经编译后，对应的 ES5 代码如下：

```
"use strict";
var Direction;
(function (Direction) {
  Direction[(Direction["NORTH"] = 0)] = "NORTH";
  Direction[(Direction["SOUTH"] = 1)] = "SOUTH";
  Direction[(Direction["EAST"] = 2)] = "EAST";
  Direction[(Direction["WEST"] = 3)] = "WEST";
})(Direction || (Direction = {}));
var dir = Direction.NORTH;
```

当然我们也可以设置 NORTH 的初始值，比如：

```
enum Direction {
  NORTH = 3,
  SOUTH,
  EAST,
  WEST,
}
```

2.字符串枚举

在 TypeScript 2.4 版本，允许我们使用字符串枚举。在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化。

```
enum Direction {
  NORTH = "NORTH",
  SOUTH = "SOUTH",
  EAST = "EAST",
  WEST = "WEST",
}
```

以上代码对应的 ES5 代码如下：

```
"use strict";
var Direction;
(function (Direction) {
    Direction["NORTH"] = "NORTH";
    Direction["SOUTH"] = "SOUTH";
    Direction["EAST"] = "EAST";
    Direction["WEST"] = "WEST";
})(Direction || (Direction = {}));
```

通过观察数字枚举和字符串枚举的编译结果，我们可以知道数字枚举除了支持 **从成员名称到成员值** 的普通映射之外，它还支持 **从成员值到成员名称** 的反向映射：

```
enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST,
}

let dirName = Direction[0]; // NORTH
let dirVal = Direction["NORTH"]; // 0
```

另外，对于纯字符串枚举，我们不能省略任何初始化程序。而数字枚举如果没有显式设置值时，则会使用默认规则进行初始化。

3.常量枚举

除了数字枚举和字符串枚举之外，还有一种特殊的枚举 —— 常量枚举。它是使用 `const` 关键字修饰的枚举，常量枚举会使用内联语法，不会为枚举类型编译生成任何 JavaScript。为了更好地理解这句话，我们来看一个具体的例子：

```
const enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST,
}

let dir: Direction = Direction.NORTH;
```

以上代码对应的 ES5 代码如下：

```
"use strict";
var dir = 0 /* NORTH */;
```

4.异构枚举

异构枚举的成员值是数字和字符串的混合：

```
enum Enum {
  A,
  B,
  C = "C",
  D = "D",
  E = 8,
  F,
}
```

以上代码对于的 ES5 代码如下：

```
"use strict";
var Enum;
(function (Enum) {
    Enum[Enum["A"] = 0] = "A";
    Enum[Enum["B"] = 1] = "B";
    Enum["C"] = "C";
    Enum["D"] = "D";
    Enum[Enum["E"] = 8] = "E";
    Enum[Enum["F"] = 9] = "F";
})(Enum || (Enum = {}));
```

通过观察上述生成的 ES5 代码，我们可以发现数字枚举相对字符串枚举多了 “反向映射”：

```
console.log(Enum.A) //输出：0
console.log(Enum[0]) // 输出：A
```

2.7 Any 类型

在 TypeScript 中，任何类型都可以被归为 any 类型。这让 any 类型成为了类型系统的顶级类型（也被称作全局超级类型）。

```
let notSure: any = 666;
notSure = "semlinker";
notSure = false;
```

`any` 类型本质上是类型系统的一个逃逸舱。作为开发者，这给了我们很大的自由：TypeScript 允许我们对 `any` 类型的值执行任何操作，而无需事先执行任何形式的检查。比如：

```
let value: any;

value.foo.bar; // OK
value.trim(); // OK
value(); // OK
new value(); // OK
value[0][1]; // OK
```

在许多场景下，这太宽松了。使用 `any` 类型，可以很容易地编写类型正确但在运行时有问题的代码。如果我们使用 `any` 类型，就无法使用 TypeScript 提供的大量的保护机制。为了解决 `any` 带来的问题，TypeScript 3.0 引入了 `unknown` 类型。

2.8 Unknown 类型

就像所有类型都可以赋值给 `any`，所有类型也都可以赋值给 `unknown`。这使得 `unknown` 成为 TypeScript 类型系统的另一种顶级类型（另一种是 `any`）。下面我们来看一下 `unknown` 类型的使用示例：

```
let value: unknown;

value = true; // OK
value = 42; // OK
value = "Hello World"; // OK
value = []; // OK
value = {}; // OK
value = Math.random; // OK
value = null; // OK
value = undefined; // OK
value = new TypeError(); // OK
value = Symbol("type"); // OK
```

对 `value` 变量的所有赋值都被认为是类型正确的。但是，当我们尝试将类型为 `unknown` 的值赋值给其他类型的变量时会发生什么?

```
let value: unknown;

let value1: unknown = value; // OK
let value2: any = value; // OK
let value3: boolean = value; // Error
let value4: number = value; // Error
let value5: string = value; // Error
let value6: object = value; // Error
let value7: any[] = value; // Error
let value8: Function = value; // Error
```

`unknown` 类型只能被赋值给 `any` 类型和 `unknown` 类型本身。直观地说，这是有道理的：只有能够保存任意类型值的容器才能保存 `unknown` 类型的值。毕竟我们不知道变量 `value` 中存储了什么类型的值。

现在让我们看看当我们尝试对类型为 `unknown` 的值执行操作时会发生什么。以下是我们在之前 `any` 章节看过的相同操作：

```
let value: unknown;

value.foo.bar; // Error
value.trim(); // Error
value(); // Error
new value(); // Error
value[0][1]; // Error
```

将 `value` 变量类型设置为 `unknown` 后，这些操作都不再被认为是类型正确的。通过将 `any` 类型改变为 `unknown` 类型，我们已将允许所有更改的默认设置，更改为禁止任何更改。

2.9 Tuple 类型

众所周知，数组一般由同种类型的值组成，但有时我们需要在单个变量中存储不同类型的值，这时候我们就可以使用元组。在 JavaScript 中是没有元组的，元组是 TypeScript 中特有的类型，其工作方式类似于数组。

元组可用于定义具有有限数量的未命名属性的类型。每个属性都有一个关联的类型。使用元组时，必须提供每个属性的值。为了更直观地理解元组的概念，我们来看一个具体的例子：

```
let tupleType: [string, boolean];
tupleType = ["semlinker", true];
```

在上面代码中，我们定义了一个名为 `tupleType` 的变量，它的类型是一个类型数组 `[string, boolean]`，然后我们按照正确的类型依次初始化 tupleType 变量。与数组一样，我们可以通过下标来访问元组中的元素：

```
console.log(tupleType[0]); // semlinker
console.log(tupleType[1]); // true
```

在元组初始化的时候，如果出现类型不匹配的话，比如：

```
tupleType = [true, "semlinker"];
```

此时，TypeScript 编译器会提示以下错误信息：

```
[0]: Type 'true' is not assignable to type 'string'.
[1]: Type 'string' is not assignable to type 'boolean'.
```

很明显是因为类型不匹配导致的。在元组初始化的时候，我们还必须提供每个属性的值，不然也会出现错误，比如：

```
tupleType = ["semlinker"];
```

此时，TypeScript 编译器会提示以下错误信息：

```
Property '1' is missing in type '[string]' but required in type '[string,
boolean]'.
```

2.10 Void 类型

某种程度上来说，void 类型像是与 any 类型相反，它表示没有任何类型。当一个函数没有返回值时，你通常会见到其返回值类型是 void：

```
// 声明函数返回值为void
function warnUser(): void {
  console.log("This is my warning message");
}
```

以上代码编译生成的 ES5 代码如下：

```
"use strict";
function warnUser() {
  console.log("This is my warning message");
}
```

需要注意的是，声明一个 void 类型的变量没有什么作用，因为在严格模式下，它的值只能为 `undefined`：

```
let unusable: void = undefined;
```

2.11 Null 和 Undefined 类型

TypeScript 里，`undefined` 和 `null` 两者有各自的类型分别为 `undefined` 和 `null`。

```
let u: undefined = undefined;
let n: null = null;
```

2.12 object, Object 和 {} 类型

1.object 类型

object 类型是：TypeScript 2.2 引入的新类型，它用于表示非原始类型。

```
// node_modules/typescript/lib/lib.es5.d.ts
interface ObjectConstructor {
  create(o: object | null): any;
  // ...
}

const proto = {};

Object.create(proto);     // OK
Object.create(null);      // OK
Object.create(undefined); // Error
Object.create(1337);      // Error
Object.create(true);      // Error
Object.create("oops");    // Error
```

2.Object 类型

Object 类型：它是所有 Object 类的实例的类型，它由以下两个接口来定义：

- Object 接口定义了 Object.prototype 原型对象上的属性；

```
// node_modules/typescript/lib/lib.es5.d.ts
interface Object {
  constructor: Function;
  toString(): string;
  toLocaleString(): string;
  valueOf(): Object;
  hasOwnProperty(v: PropertyKey): boolean;
  isPrototypeOf(v: Object): boolean;
  propertyIsEnumerable(v: PropertyKey): boolean;
}
```

- ObjectConstructor 接口定义了 Object 类的属性。

```
// node_modules/typescript/lib/lib.es5.d.ts
interface ObjectConstructor {
  /** Invocation via `new` */
  new(value?: any): Object;
  /** Invocation via function calls */
  (value?: any): any;
  readonly prototype: Object;
  getPrototypeOf(o: any): any;
  // ···
}

declare var Object: ObjectConstructor;
```

Object 类的所有实例都继承了 Object 接口中的所有属性。

3.{} 类型

{} 类型描述了一个没有成员的对象。当你试图访问这样一个对象的任意属性时，TypeScript 会产生一个编译时错误。

```
// Type {}
const obj = {};

// Error: Property 'prop' does not exist on type '{}'.
obj.prop = "semlinker";
```

但是，你仍然可以使用在 Object 类型上定义的所有属性和方法，这些属性和方法可通过 JavaScript 的原型链隐式地使用：

```
// Type {}
const obj = {};

// "[object Object]"
obj.toString();
```

2.13 Never 类型

`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。

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

在 TypeScript 中，可以利用 never 类型的特性来实现全面性检查，具体示例如下：

```
type Foo = string | number;

function controlFlowAnalysisWithNever(foo: Foo) {
  if (typeof foo === "string") {
    // 这里 foo 被收窄为 string 类型
  } else if (typeof foo === "number") {
    // 这里 foo 被收窄为 number 类型
  } else {
    // foo 在这里是 never
    const check: never = foo;
  }
}
```

注意在 else 分支里面，我们把收窄为 never 的 foo 赋值给一个显示声明的 never 变量。如果一切逻辑正确，那么这里应该能够编译通过。但是假如后来有一天你的同事修改了 Foo 的类型：

```
type Foo = string | number | boolean;
```

然而他忘记同时修改 `controlFlowAnalysisWithNever` 方法中的控制流程，这时候 else 分支的 foo 类型会被收窄为 `boolean` 类型，导致无法赋值给 never 类型，这时就会产生一个编译错误。通过这个方式，我们可以确保

`controlFlowAnalysisWithNever` 方法总是穷尽了 Foo 的所有可能类型。 通过这个示例，我们可以得出一个结论：**使用 never 避免出现新增了联合类型没有对应的实现，目的就是写出类型绝对安全的代码。**

三、TypeScript 断言

3.1 类型断言

有时候你会遇到这样的情况，你会比 TypeScript 更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。类型断言好比其他语言里的类型转换，但是不进行特殊的数据检查和解构。它没有运行时的影响，只是在编译阶段起作用。

类型断言有两种形式：

1.“尖括号” 语法

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

2.as 语法

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

3.2 非空断言

在上下文中当类型检查器无法断定类型时，一个新的后缀表达式操作符 `!` 可以用于断言操作对象是非 null 和非 undefined 类型。**具体而言，x! 将从 x 值域中排除 null 和 undefined 。**

那么非空断言操作符到底有什么用呢？下面我们先来看一下非空断言操作符的一些使用场景。

1.忽略 undefined 和 null 类型

```
function myFunc(maybeString: string | undefined | null) {
  // Type 'string | null | undefined' is not assignable to type 'string'.
  // Type 'undefined' is not assignable to type 'string'.
  const onlyString: string = maybeString; // Error
  const ignoreUndefinedAndNull: string = maybeString!; // Ok
}
```

2.调用函数时忽略 undefined 类型

```
type NumGenerator = () => number;

function myFunc(numGenerator: NumGenerator | undefined) {
  // Object is possibly 'undefined'.(2532)
  // Cannot invoke an object which is possibly 'undefined'.(2722)
  const num1 = numGenerator(); // Error
  const num2 = numGenerator!(); //OK
}
```

因为 `!` 非空断言操作符会从编译生成的 JavaScript 代码中移除，所以在实际使用的过程中，要特别注意。比如下面这个例子：

```
const a: number | undefined = undefined;
const b: number = a!;
console.log(b);
```

以上 TS 代码会编译生成以下 ES5 代码：

```
"use strict";
const a = undefined;
const b = a;
console.log(b);
```

虽然在 TS 代码中，我们使用了非空断言，使得 `const b: number = a!;` 语句可以通过 TypeScript 类型检查器的检查。但在生成的 ES5 代码中，`!` 非空断言操作符被移除了，所以在浏览器中执行以上代码，在控制台会输出 `undefined`。

3.3 确定赋值断言

在 TypeScript 2.7 版本中引入了确定赋值断言，即允许在实例属性和变量声明后面放置一个 `!` 号，从而告诉 TypeScript 该属性会被明确地赋值。为了更好地理解它的作用，我们来看个具体的例子：

```
let x: number;
initialize();
// Variable 'x' is used before being assigned.(2454)
console.log(2 * x); // Error

function initialize() {
  x = 10;
}
```

很明显该异常信息是说变量 x 在赋值前被使用了，要解决该问题，我们可以使用确定赋值断言：

```
let x!: number;
initialize();
console.log(2 * x); // Ok

function initialize() {
  x = 10;
}
```

通过 `let x!: number;` 确定赋值断言，TypeScript 编译器就会知道该属性会被明确地赋值。

四、类型守卫

类型保护是可执行运行时检查的一种表达式，用于确保该类型在一定的范围内。 换句话说，类型保护可以保证一个字符串是一个字符串，尽管它的值也可以是一个数值。类型保护与特性检测并不是完全不同，其主要思想是尝试检测属性、方法或原型，以确定如何处理值。

目前主要有四种的方式来实现类型保护：

4.1 in 关键字

```
interface Admin {
  name: string;
  privileges: string[];
}

interface Employee {
  name: string;
  startDate: Date;
}

type UnknownEmployee = Employee | Admin;

function printEmployeeInformation(emp: UnknownEmployee) {
  console.log("Name: " + emp.name);
  if ("privileges" in emp) {
    console.log("Privileges: " + emp.privileges);
  }
  if ("startDate" in emp) {
    console.log("Start Date: " + emp.startDate);
  }
}
```

4.2 typeof 关键字

```
function padLeft(value: string, padding: string | number) {
  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}'.`);
}
```

`typeof` 类型保护只支持两种形式：`typeof v === "typename"` 和 `typeof v !== "typename"`，`"typename"` 必须是 `"number"`，`"string"`，`"boolean"` 或 `"symbol"`。但是 TypeScript 并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

4.3 instanceof 关键字

```
interface Padder {
  getPaddingString(): string;
}
```

```
class SpaceRepeatingPadder implements Padder {
  constructor(private numSpaces: number) {}
  getPaddingString() {
    return Array(this.numSpaces + 1).join(" ");
  }
}

class StringPadder implements Padder {
  constructor(private value: string) {}
  getPaddingString() {
    return this.value;
  }
}

let padder: Padder = new SpaceRepeatingPadder(6);

if (padder instanceof SpaceRepeatingPadder) {
  // padder的类型收窄为 'SpaceRepeatingPadder'
}
```

4.4 自定义类型保护的类型谓词

```
function isNumber(x: any): x is number {
  return typeof x === "number";
}

function isString(x: any): x is string {
  return typeof x === "string";
}
```

五、联合类型和类型别名

5.1 联合类型

联合类型通常与 `null` 或 `undefined` 一起使用：

```
const sayHello = (name: string | undefined) => {
  /* ... */
};
```

例如，这里 `name` 的类型是 `string | undefined` 意味着可以将 `string` 或 `undefined` 的值传递给 `sayHello` 函数。

```
sayHello("semlinker");
sayHello(undefined);
```

通过这个示例，你可以凭直觉知道类型 A 和类型 B 联合后的类型是同时接受 A 和 B 值的类型。此外，对于联合类型来说，你可能会遇到以下的用法：

```
let num: 1 | 2 = 1;
type EventNames = 'click' | 'scroll' | 'mousemove';
```

以上示例中的 `1`、`2` 或 `'click'` 被称为字面量类型，用来约束取值只能是某几个值中的一个。

5.2 可辨识联合

TypeScript 可辨识联合（Discriminated Unions）类型，也称为代数数据类型或标签联合类型。**它包含 3 个要点：可辨识、联合类型和类型守卫。**

这种类型的本质是结合联合类型和字面量类型的一种类型保护方法。**如果一个类型是多个类型的联合类型，且多个类型含有一个公共属性，那么就可以利用这个公共属性，来创建不同的类型保护区块。**

1.可辨识

可辨识要求联合类型中的每个元素都含有一个单例类型属性，比如：

```
enum CarTransmission {
  Automatic = 200,
  Manual = 300
}

interface Motorcycle {
  vType: "motorcycle"; // discriminant
  make: number; // year
}

interface Car {
  vType: "car"; // discriminant
  transmission: CarTransmission
}

interface Truck {
  vType: "truck"; // discriminant
  capacity: number; // in tons
}
```

在上述代码中，我们分别定义了 `Motorcycle`、 `Car` 和 `Truck` 三个接口，在这些接口中都包含一个 `vType` 属性，该属性被称为可辨识的属性，而其它的属性只跟特性的接口相关。

2.联合类型

基于前面定义了三个接口，我们可以创建一个 `Vehicle` 联合类型：

```
type Vehicle = Motorcycle | Car | Truck;
```

现在我们就可以开始使用 `Vehicle` 联合类型，对于 `Vehicle` 类型的变量，它可以表示不同类型的车辆。

3.类型守卫

下面我们来定义一个 `evaluatePrice` 方法，该方法用于根据车辆的类型、容量和评估因子来计算价格，具体实现如下：

```
const EVALUATION_FACTOR = Math.PI;

function evaluatePrice(vehicle: Vehicle) {
  return vehicle.capacity * EVALUATION_FACTOR;
}

const myTruck: Truck = { vType: "truck", capacity: 9.5 };
evaluatePrice(myTruck);
```

对于以上代码，TypeScript 编译器将会提示以下错误信息：

```
Property 'capacity' does not exist on type 'Vehicle'.
Property 'capacity' does not exist on type 'Motorcycle'.
```

原因是在 Motorcycle 接口中，并不存在 `capacity` 属性，而对于 Car 接口来说，它也不存在 `capacity` 属性。那么，现在我们应该如何解决以上问题呢？这时，我们可以使用类型守卫。下面我们来重构一下前面定义的 `evaluatePrice` 方法，重构后的代码如下：

```
function evaluatePrice(vehicle: Vehicle) {
  switch(vehicle.vType) {
    case "car":
      return vehicle.transmission * EVALUATION_FACTOR;
    case "truck":
      return vehicle.capacity * EVALUATION_FACTOR;
    case "motorcycle":
      return vehicle.make * EVALUATION_FACTOR;
  }
}
```

在以上代码中，我们使用 `switch` 和 `case` 运算符来实现类型守卫，从而确保在 `evaluatePrice` 方法中，我们可以安全地访问 `vehicle` 对象中的所包含的属性，来正确的计算该车辆类型所对应的价格。

5.3 类型别名

类型别名用来给一个类型起个新名字。

```ts
type Message = string | string[];

let greet = (message: Message) => {
  // ...
};
```

六、交叉类型

在 TypeScript 中交叉类型是将多个类型合并为一个类型。通过 `&` 运算符可以将现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。

```ts
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };

let point: Point = {
  x: 1,
  y: 1
}
```

在上面代码中我们先定义了 `PartialPointX` 类型，接着使用 `&` 运算符创建一个新的 `Point` 类型，表示一个含有 x 和 y 坐标的点，然后定义了一个 `Point` 类型的变量并初始化。

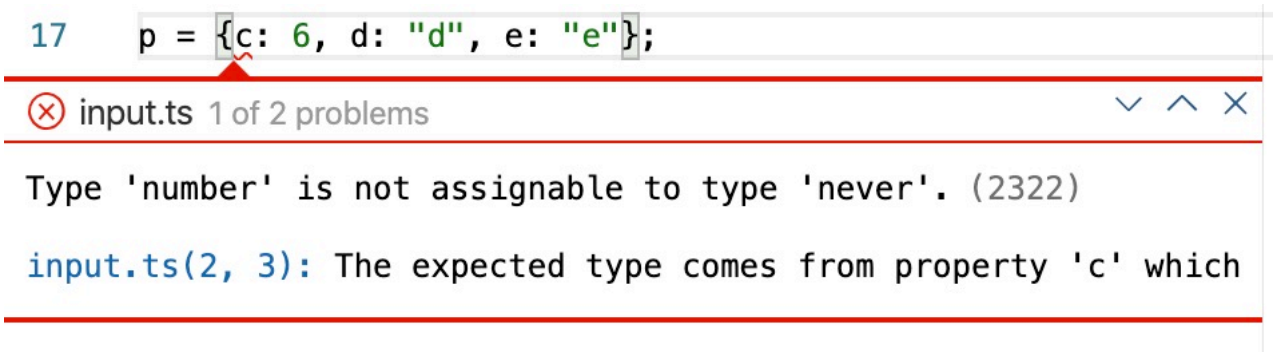
6.1 同名基础类型属性的合并

那么现在问题来了，假设在合并多个类型的过程中，刚好出现某些类型存在相同的成员，但对应的类型又不一致，比如：

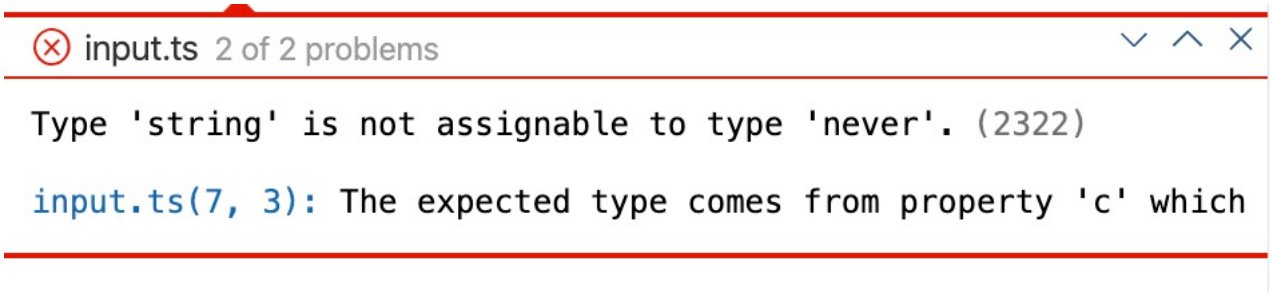
```ts
interface X {
  c: string;
  d: string;
}

interface Y {
  c: number;
  e: string
}

type XY = X & Y;
type YX = Y & X;

let p: XY;
let q: YX;
```

在上面的代码中，接口 X 和接口 Y 都含有一个相同的成员 c，但它们的类型不一致。对于这种情况，此时 XY 类型或 YX 类型中成员 c 的类型是不是可以是 `string` 或 `number` 类型呢？比如下面的例子：

```ts
p = { c: 6, d: "d", e: "e" };
```




```
q = { c: "c", d: "d", e: "e" };
```



为什么接口 X 和接口 Y 混入后，成员 c 的类型会变成 `never` 呢？这是因为混入后成员 c 的类型为 `string & number`，即成员 c 的类型既可以是 `string` 类型又可以是 `number` 类型。很明显这种类型是不存在的，所以混入后成员 c 的类型为 `never`。

6.2 同名非基础类型属性的合并

在上面示例中，刚好接口 X 和接口 Y 中内部成员 c 的类型都是基本数据类型，那么如果是非基本数据类型的话，又会是什么情形。我们来看个具体的例子：

```
interface D { d: boolean; }
interface E { e: string; }
interface F { f: number; }

interface A { x: D; }
interface B { x: E; }
interface C { x: F; }

type ABC = A & B & C;

let abc: ABC = {
  x: {
    d: true,
    e: 'semlinker',
    f: 666
  }
};

console.log('abc:', abc);
```

以上代码成功运行后，控制台会输出以下结果：

由上图可知，在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，那么是可以成功合并。

七、TypeScript 函数

7.1 TypeScript 函数与 JavaScript 函数的区别

TypeScript	JavaScript
含有类型	无类型
箭头函数	箭头函数（ES2015）
函数类型	无函数类型
必填和可选参数	所有参数都是可选的
默认参数	默认参数
剩余参数	剩余参数
函数重载	无函数重载

7.2 箭头函数

1.常见语法

```
myBooks.forEach(() => console.log('reading'));

myBooks.forEach(title => console.log(title));

myBooks.forEach((title, idx, arr) =>
  console.log(idx + '-' + title);
);

myBooks.forEach((title, idx, arr) => {
  console.log(idx + '-' + title);
});
```

2.使用示例

```
// 未使用箭头函数
function Book() {
  let self = this;
  self.publishDate = 2016;
  setInterval(function () {
    console.log(self.publishDate);
  }, 1000);
}

// 使用箭头函数
function Book() {
  this.publishDate = 2016;
  setInterval(() => {
    console.log(this.publishDate);
  }, 1000);
}
```

7.3 参数类型和返回类型

```
function createUserId(name: string, id: number): string {
  return name + id;
}
```

7.4 函数类型

```
let IdGenerator: (chars: string, nums: number) => string;

function createUserId(name: string, id: number): string {
  return name + id;
}

IdGenerator = createUserId;
```

7.5 可选参数及默认参数

```
// 可选参数
function createUserId(name: string, id: number, age?: number): string {
  return name + id;
}

// 默认参数
function createUserId(
  name = "semlinker",
  id: number,
  age?: number
): string {
  return name + id;
}
```

在声明函数时，可以通过 `?` 号来定义可选参数，比如 `age?: number` 这种形式。**在实际使用时，需要注意的是可选参数要放在普通参数的后面，不然会导致编译错误。**

7.6 剩余参数

```
function push(array, ...items) {
  items.forEach(function (item) {
    array.push(item);
  });
}

let a = [];
push(a, 1, 2, 3);
```

7.7 函数重载

函数重载或方法重载是使用相同名称和不同参数数量或类型创建多个方法的一种能力。

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;
function add(a: Combinable, b: Combinable) {
  // type Combinable = string | number;
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}
```

在以上代码中，我们为 add 函数提供了多个函数类型定义，从而实现函数的重载。在 TypeScript 中除了可以重载普通函数之外，我们还可以重载类中的成员方法。

方法重载是指在同一个类中方法同名，参数不同（参数类型不同、参数个数不同或参数个数相同时参数的先后顺序不同），调用时根据实参的形式，选择与它匹配的方法执行操作的一种技术。所以类中成员方法满足重载的条件是：在同一个类中，方法名相同且参数列表不同。下面我们来举一个成员方法重载的例子：

```
class Calculator {
  add(a: number, b: number): number;
  add(a: string, b: string): string;
  add(a: string, b: number): string;
  add(a: number, b: string): string;
  add(a: Combinable, b: Combinable) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
    return a + b;
  }
}

const calculator = new Calculator();
const result = calculator.add('Semlinker', ' Kakuqo');
```

这里需要注意的是，当 TypeScript 编译器处理函数重载时，它会查找重载列表，尝试使用第一个重载定义。 如果匹配的话就使用这个。 因此，在定义重载的时候，一定要把最精确的定义放在最前面。另外在 Calculator 类中，`add(a: Combinable, b: Combinable){ }` 并不是重载列表的一部分，因此对于 add 成员方法来说，我们只定义了四个重载方法。

八、TypeScript 数组

8.1 数组解构

```
let x: number; let y: number; let z: number;
let five_array = [0,1,2,3,4];
[x,y,z] = five_array;
```

8.2 数组展开运算符

```
let two_array = [0, 1];
let five_array = [...two_array, 2, 3, 4];
```

8.3 数组遍历

```
let colors: string[] = ["red", "green", "blue"];
for (let i of colors) {
  console.log(i);
}
```

九、TypeScript 对象

9.1 对象解构

```
let person = {
  name: "Semlinker",
  gender: "Male",
};

let { name, gender } = person;
```

9.2 对象展开运算符

```
let person = {
  name: "Semlinker",
  gender: "Male",
  address: "Xiamen",
};

// 组装对象
let personWithAge = { ...person, age: 33 };

// 获取除了某些项外的其它项
let { name, ...rest } = person;
```

十、TypeScript 接口

在面向对象语言中，接口是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类去实现。

TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。

10.1 对象的形状

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

let semlinker: Person = {
  name: "semlinker",
  age: 33,
};
```

10.2 可选 | 只读属性

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

只读属性用于限制只能在对象刚刚创建的时候修改其值。此外 TypeScript 还提供了 `ReadonlyArray<T>` 类型，它与 `Array<T>` 相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改。

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

10.3 任意属性

有时候我们希望一个接口中除了包含必选和可选属性之外，还允许有其他的任意属性，这时我们可以使用 **索引签名** 的形式来满足上述要求。

```ts
interface Person {
  name: string;
  age?: number;
  [propName: string]: any;
}

const p1 = { name: "semlinker" };
const p2 = { name: "lolo", age: 5 };
const p3 = { name: "kakuqo", sex: 1 }
```

10.4 接口与类型别名的区别

1.Objects/Functions

接口和类型别名都可以用来描述对象的形状或函数签名：

接口

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

interface SetPoint {
  (x: number, y: number): void;
}
```

类型别名

```ts
type Point = {
  x: number;
  y: number;
};

type SetPoint = (x: number, y: number) => void;
```

2.Other Types

与接口类型不一样，类型别名可以用于一些其他类型，比如原始类型、联合类型和元组：

```
// primitive
type Name = string;

// object
type PartialPointX = { x: number; };
type PartialPointY = { y: number; };

// union
type PartialPoint = PartialPointX | PartialPointY;

// tuple
type Data = [number, string];
```

3.Extend

接口和类型别名都能够被扩展，但语法有所不同。此外，接口和类型别名不是互斥的。接口可以扩展类型别名，而反过来是不行的。

Interface extends interface

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

Type alias extends type alias

```
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };
```

Interface extends type alias

```
type PartialPointX = { x: number; };
interface Point extends PartialPointX { y: number; }
```

Type alias extends interface

```
interface PartialPointX { x: number; }
type Point = PartialPointX & { y: number; };
```

4.Implements

类可以以相同的方式实现接口或类型别名，但类不能实现使用类型别名定义的联合类型：

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

```
}

class SomePoint implements Point {
  x = 1;
  y = 2;
}

type Point2 = {
  x: number;
  y: number;
};

class SomePoint2 implements Point2 {
  x = 1;
  y = 2;
}

type PartialPoint = { x: number; } | { y: number; };

// A class can only implement an object type or
// intersection of object types with statically known members.
class SomePartialPoint implements PartialPoint { // Error
  x = 1;
  y = 2;
}
```

5.Declaration merging

与类型别名不同，接口可以定义多次，会被自动合并为单个接口。

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

const point: Point = { x: 1, y: 2 };
```

十一、TypeScript 类

11.1 类的属性与方法

在面向对象语言中，类是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。

在 TypeScript 中，我们可以通过 `Class` 关键字来定义一个类：

```
class Greeter {
  // 静态属性
  static cname: string = "Greeter";
  // 成员属性
  greeting: string;
```

```
  // 构造函数 - 执行初始化操作
  constructor(message: string) {
    this.greeting = message;
  }

  // 静态方法
  static getClassName() {
    return "Class name is Greeter";
  }

  // 成员方法
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

那么成员属性与静态属性，成员方法与静态方法有什么区别呢？这里无需过多解释，我们直接看一下编译生成的 ES5 代码：

```
"use strict";
var Greeter = /** @class */ (function () {
    // 构造函数 - 执行初始化操作
    function Greeter(message) {
      this.greeting = message;
    }
    // 静态方法
    Greeter.getClassName = function () {
      return "Class name is Greeter";
    };
    // 成员方法
    Greeter.prototype.greet = function () {
      return "Hello, " + this.greeting;
    };
    // 静态属性
    Greeter.cname = "Greeter";
    return Greeter;
}());
var greeter = new Greeter("world");
```

11.2 ECMAScript 私有字段

在 TypeScript 3.8 版本就开始支持**ECMAScript 私有字段**，使用方式如下：

```
class Person {
  #name: string;
```

```
  constructor(name: string) {
    this.#name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}

let semlinker = new Person("Semlinker");

semlinker.#name;
//     ~~~~~
// Property '#name' is not accessible outside class 'Person'
// because it has a private identifier.
```

与常规属性（甚至使用 `private` 修饰符声明的属性）不同，私有字段要牢记以下规则：

- 私有字段以 `#` 字符开头，有时我们称之为私有名称；
- 每个私有字段名称都唯一地限定于其包含的类；
- 不能在私有字段上使用 TypeScript 可访问性修饰符（如 public 或 private）；
- 私有字段不能在包含的类之外访问，甚至不能被检测到。

11.3 访问器

在 TypeScript 中，我们可以通过 `getter` 和 `setter` 方法来实现数据的封装和有效性校验，防止出现异常数据。

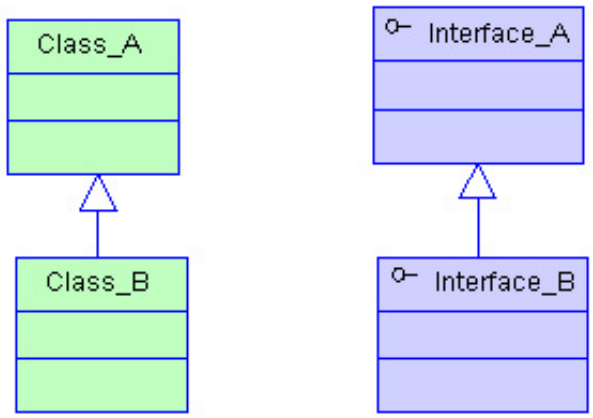
```
let passcode = "Hello TypeScript";

class Employee {
  private _fullName: string;

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (passcode && passcode == "Hello TypeScript") {
      this._fullName = newName;
    } else {
      console.log("Error: Unauthorized update of employee!");
    }
  }
}

let employee = new Employee();
employee.fullName = "Semlinker";
```

```
if (employee.fullName) {
  console.log(employee.fullName);
}
```

11.4 类的继承

继承（Inheritance）是一种联结类与类的层次模型。指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系。

继承是一种 is-a 关系：



在 TypeScript 中，我们可以通过 `extends` 关键字来实现继承：

```
class Animal {
  name: string;

  constructor(theName: string) {
    this.name = theName;
  }

  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) {
    super(name); // 调用父类的构造函数
  }

  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}
```

```
let sam = new Snake("Sammy the Python");
sam.move();
```

11.5 抽象类

使用 `abstract` 关键字声明的类，我们称之为抽象类。抽象类不能被实例化，因为它里面包含一个或多个抽象方法。所谓的抽象方法，是指不包含具体实现的方法：

```
abstract class Person {
  constructor(public name: string){}

  abstract say(words: string) :void;
}

// Cannot create an instance of an abstract class.(2511)
const lolo = new Person(); // Error
```

抽象类不能被直接实例化，我们只能实例化实现了所有抽象方法的子类。具体如下所示：

```
abstract class Person {
  constructor(public name: string){}

  // 抽象方法
  abstract say(words: string) :void;
}

class Developer extends Person {
  constructor(name: string) {
    super(name);
  }

  say(words: string): void {
    console.log(`${this.name} says ${words}`);
  }
}

const lolo = new Developer("lolo");
lolo.say("I love ts!"); // lolo says I love ts!
```

11.6 类方法重载

在前面的章节，我们已经介绍了函数重载。对于类的方法来说，它也支持重载。比如，在以下示例中我们重载了 `ProductService` 类的 `getProducts` 成员方法：

```
class ProductService {
    getProducts(): void;
    getProducts(id: number): void;
    getProducts(id?: number) {
```

```
        if(typeof id === 'number') {
            console.log(`获取id为 ${id} 的产品信息`);
        } else {
            console.log(`获取所有的产品信息`);
        }
    }
}

const productService = new ProductService();
productService.getProducts(666); // 获取id为 666 的产品信息
productService.getProducts(); // 获取所有的产品信息
```

十二、TypeScript 泛型

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。

泛型（Generics）是允许同一个函数接受不同类型参数的一种模板。相比于使用 any 类型，使用泛型来创建可复用的组件要更好，因为泛型会保留参数类型。

12.1 泛型语法

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。其实它没有什么特别，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。

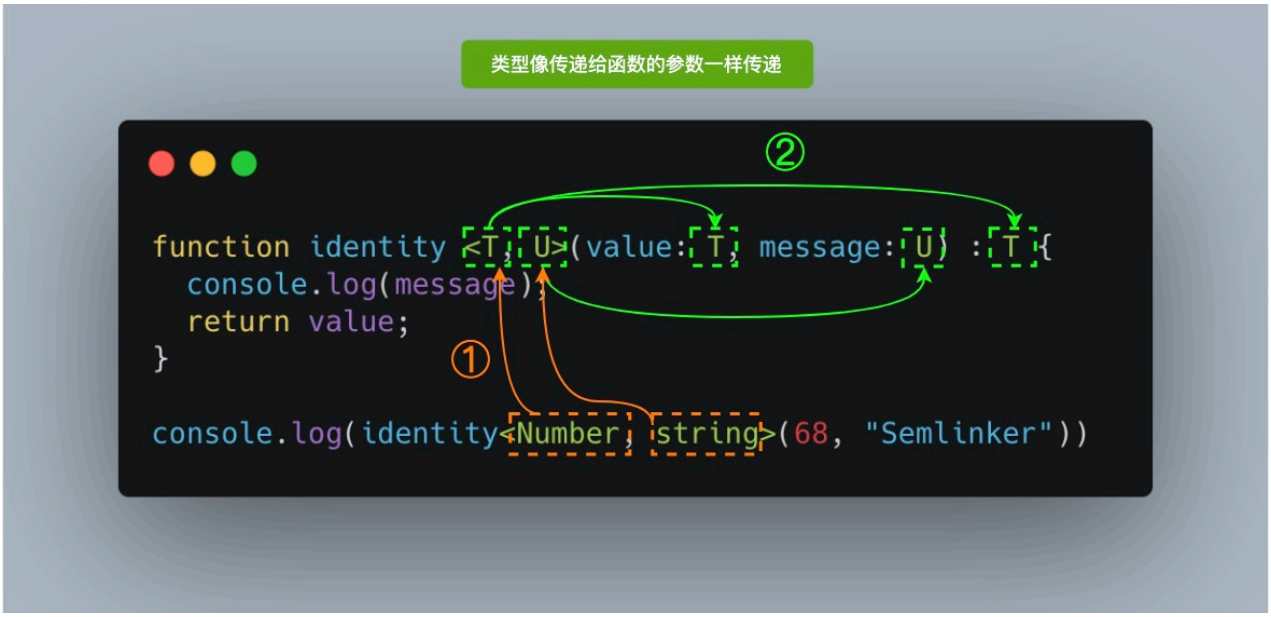
参考上面的图片，当我们调用 `identity<Number>(1)`，`Number` 类型就像参数 `1` 一样，它将在出现 `T` 的任何位置填充该类型。图中 `<T>` 内部的 `T` 被称为类型变量，它是我们希望传递给 identity 函数的类型占位符，同时它被分配给 `value` 参数用来代替它的类型：此时 `T` 充当的是类型，而不是特定的 Number 类型。

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"));
```



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。

12.2 泛型接口

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}
```

12.3 泛型类

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
  return x + y;
};
```

12.4 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 Partial 工具类型。不过在具体介绍之前，我们得先介绍一些相关的基础知识，方便读者自行学习其它的工具类型。

1.typeof

在 TypeScript 中，`typeof` 操作符可以用来获取一个变量声明或对象的类型。

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

const sem: Person = { name: 'semlinker', age: 33 };
type Sem= typeof sem; // -> Person

function toArray(x: number): Array<number> {
  return [x];
}

type Func = typeof toArray; // -> (x: number) => number[]
```

2.keyof

`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。

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

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

在 TypeScript 中支持两种索引签名，数字索引和字符串索引：

```
interface StringArray {
  // 字符串索引 -> keyof StringArray => string | number
  [index: string]: string;
}

interface StringArray1 {
  // 数字索引 -> keyof StringArray1 => number
  [index: number]: string;
}
```

为了同时支持两种索引类型，就得要求数字索引的返回值必须是字符串索引返回值的子类。**其中的原因就是当使用数值索引时，JavaScript 在执行索引操作时，会先把数值索引先转换为字符串索引**。所以 `keyof { [x: string]: Person }` 的结果会返回 `string | number`。

3.in

`in` 用来遍历枚举类型：

```
type Keys = "a" | "b" | "c"

type Obj =  {
  [p in Keys]: any
} // -> { a: any, b: any, c: any }
```

4.infer

在条件类型语句中，可以用 `infer` 声明一个类型变量并且对它进行使用。

```
type ReturnType<T> = T extends (
  ...args: any[]
) => infer R ? R : any;
```

以上代码中 `infer R` 就是声明一个变量来承载传入函数签名的返回值类型，简单说就是用它取到函数返回值的类型方便之后使用。

5.extends

有时候我们定义的泛型不想过于灵活或者说想继承某些类等，可以通过 extends 关键字添加泛型约束。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```

这时我们需要传入符合约束类型的值，必须包含必须的属性：

```
loggingIdentity({length: 10, value: 3});
```

6.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: "Learn TS",
  description: "Learn TypeScript",
};

const todo2 = updateTodo(todo1, {
  description: "Learn TypeScript Enum",
});
```

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;
}
```

十三、TypeScript 装饰器

13.1 装饰器是什么

- 它是一个表达式
- 该表达式被执行后，返回一个函数
- 函数的入参分别为 target、name 和 descriptor
- 执行该函数后，可能返回 descriptor 对象，用于配置 target 对象

13.2 装饰器的分类

- 类装饰器（Class decorators）
- 属性装饰器（Property decorators）
- 方法装饰器（Method decorators）
- 参数装饰器（Parameter decorators）

需要注意的是，若要启用实验性的装饰器特性，你必须在命令行或 `tsconfig.json` 里启用 `experimentalDecorators` 编译器选项：

命令行：

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json：

```
{
  "compilerOptions": {
     "target": "ES5",
     "experimentalDecorators": true
   }
}
```

13.3 类装饰器

类装饰器声明：

```
declare type ClassDecorator = <TFunction extends Function>(
  target: TFunction
) => TFunction | void;
```

类装饰器顾名思义，就是用来装饰类的。它接收一个参数：

- target: TFunction - 被装饰的类

看完第一眼后，是不是感觉都不好了。没事，我们马上来个例子：

```
function Greeter(target: Function): void {
  target.prototype.greet = function (): void {
    console.log("Hello Semlinker!");
  };
}

@Greeter
class Greeting {
  constructor() {
    // 内部实现
  }
}

let myGreeting = new Greeting();
(myGreeting as any).greet(); // console output: 'Hello Semlinker!';
```

上面的例子中，我们定义了 `Greeter` 类装饰器，同时我们使用了 `@Greeter` 语法糖，来使用装饰器。

> 友情提示：读者可以直接复制上面的代码，在 TypeScript Playground 中运行查看结果。

有的读者可能想问，例子中总是输出 `Hello Semlinker!` ，能自定义输出的问候语么？ 这个问题很好，答案是可以的。

具体实现如下：

```
function Greeter(greeting: string) {
  return function (target: Function) {
    target.prototype.greet = function (): void {
```

```
      console.log(greeting);
    };
  };
}

@Greeter("Hello TS!")
class Greeting {
  constructor() {
    // 内部实现
  }
}

let myGreeting = new Greeting();
(myGreeting as any).greet(); // console output: 'Hello TS!';
```

13.4 属性装饰器

属性装饰器声明：

```
declare type PropertyDecorator = (target:Object,
  propertyKey: string | symbol ) => void;
```

属性装饰器顾名思义，用来装饰类的属性。它接收两个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 被装饰类的属性名

趁热打铁，马上来个例子热热身：

```
function logProperty(target: any, key: string) {
  delete target[key];

  const backingField = "_" + key;

  Object.defineProperty(target, backingField, {
    writable: true,
    enumerable: true,
    configurable: true
  });

  // property getter
  const getter = function (this: any) {
    const currVal = this[backingField];
    console.log(`Get: ${key} => ${currVal}`);
    return currVal;
  };

  // property setter
  const setter = function (this: any, newVal: any) {
```

```
    console.log(`Set: ${key} => ${newVal}`);
    this[backingField] = newVal;
  };

  // Create new property with getter and setter
  Object.defineProperty(target, key, {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true
  });
}

class Person {
  @logProperty
  public name: string;

  constructor(name : string) {
    this.name = name;
  }
}

const p1 = new Person("semlinker");
p1.name = "kakuqo";
```

以上代码我们定义了一个 `logProperty` 函数，来跟踪用户对属性的操作，当代码成功运行后，在控制台会输出以下结果：

```
Set: name => semlinker
Set: name => kakuqo
```

13.5 方法装饰器

方法装饰器声明：

```
declare type MethodDecorator = <T>(target:Object, propertyKey: string | symbol,
  descriptor: TypePropertyDescript<T>) => TypedPropertyDescriptor<T> | void;
```

方法装饰器顾名思义，用来装饰类的方法。它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- descriptor: TypePropertyDescript - 属性描述符

废话不多说，直接上例子：

```
function log(target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
```

```
  let originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log("wrapped function: before invoking " + propertyKey);
    let result = originalMethod.apply(this, args);
    console.log("wrapped function: after invoking " + propertyKey);
    return result;
  };
}

class Task {
  @log
  runTask(arg: any): any {
    console.log("runTask invoked, args: " + arg);
    return "finished";
  }
}

let task = new Task();
let result = task.runTask("learn ts");
console.log("result: " + result);
```

以上代码成功运行后，控制台会输出以下结果：

```
"wrapped function: before invoking runTask"
"runTask invoked, args: learn ts"
"wrapped function: after invoking runTask"
"result: finished"
```

下面我们来介绍一下参数装饰器。

13.6 参数装饰器

参数装饰器声明：

```
declare type ParameterDecorator = (target: Object, propertyKey: string |
symbol,
  parameterIndex: number ) => void
```

参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- parameterIndex: number - 方法中参数的索引值

```
function Log(target: Function, key: string, parameterIndex: number) {
  let functionLogged = key || target.prototype.constructor.name;
  console.log(`The parameter in position ${parameterIndex} at ${functionLogged}
has
  been decorated`);
}

class Greeter {
  greeting: string;
  constructor(@Log phrase: string) {
  this.greeting = phrase;
  }
}
```

以上代码成功运行后，控制台会输出以下结果：

```
"The parameter in position 0 at Greeter has been decorated"
```

十四、TypeScript 4.0 新特性

TypeScript 4.0 带来了很多新的特性，这里我们只简单介绍其中的两个新特性。

14.1 构造函数的类属性推断

当 `noImplicitAny` 配置属性被启用之后，TypeScript 4.0 就可以使用控制流分析来确认类中的属性类型：

```
class Person {
  fullName; // (property) Person.fullName: string
  firstName; // (property) Person.firstName: string
  lastName; // (property) Person.lastName: string

  constructor(fullName: string) {
    this.fullName = fullName;
    this.firstName = fullName.split(" ")[0];
    this.lastName =   fullName.split(" ")[1];
  }
}
```

然而对于以上的代码，如果在 TypeScript 4.0 以前的版本，比如在 3.9.2 版本下，编译器会提示以下错误信息：

```
class Person {
  // Member 'fullName' implicitly has an 'any' type.(7008)
  fullName; // Error
  firstName; // Error
  lastName; // Error

  constructor(fullName: string) {
    this.fullName = fullName;
    this.firstName = fullName.split(" ")[0];
    this.lastName =   fullName.split(" ")[1];
  }
}
```

从构造函数推断类属性的类型，该特性给我们带来了便利。但在使用过程中，如果我们没法保证对成员属性都进行赋值，那么该属性可能会被认为是 `undefined`。

```
class Person {
   fullName;  // (property) Person.fullName: string
   firstName; // (property) Person.firstName: string | undefined
   lastName; // (property) Person.lastName: string | undefined

   constructor(fullName: string) {
     this.fullName = fullName;
     if(Math.random()){
       this.firstName = fullName.split(" ")[0];
       this.lastName =   fullName.split(" ")[1];
     }
   }
}
```

14.2 标记的元组元素

在以下的示例中，我们使用元组类型来声明剩余参数的类型：

```
function addPerson(...args: [string, number]): void {
  console.log(`Person info: name: ${args[0]}, age: ${args[1]}`)
}

addPerson("lolo", 5); // Person info: name: lolo, age: 5
```

其实，对于上面的 `addPerson` 函数，我们也可以这样实现：

```
function addPerson(name: string, age: number) {
  console.log(`Person info: name: ${name}, age: ${age}`)
}
```

这两种方式看起来没有多大的区别，但对于第一种方式，我们没法设置第一个参数和第二个参数的名称。虽然这样对类型检查没有影响，但在元组位置上缺少标签，会使得它们难于使用。为了提高开发者使用元组的体验，TypeScript 4.0 支持为元组类型设置标签：

```
function addPerson(...args: [name: string, age: number]): void {
  console.log(`Person info: name: ${args[0]}, age: ${args[1]}`);
}
```

之后，当我们使用 `addPerson` 方法时，TypeScript 的智能提示就会变得更加友好。

```
// 未使用标签的智能提示
// addPerson(args_0: string, args_1: number): void
function addPerson(...args: [string, number]): void {
  console.log(`Person info: name: ${args[0]}, age: ${args[1]}`)
}

// 已使用标签的智能提示
// addPerson(name: string, age: number): void
function addPerson(...args: [name: string, age: number]): void {
  console.log(`Person info: name: ${args[0]}, age: ${args[1]}`);
}
```

十五、编译上下文

15.1 tsconfig.json 的作用

- 用于标识 TypeScript 项目的根路径；
- 用于配置 TypeScript 编译器；
- 用于指定编译的文件。

15.2 tsconfig.json 重要字段

- files - 设置要编译的文件的名称；
- include - 设置需要进行编译的文件，支持路径模式匹配；
- exclude - 设置无需进行编译的文件，支持路径模式匹配；
- compilerOptions - 设置与编译流程相关的选项。

15.3 compilerOptions 选项

compilerOptions 支持很多选项，常见的有 `baseUrl`、 `target`、 `moduleResolution` 和 `lib` 等。

compilerOptions 每个选项的详细说明如下：

```
{
  "compilerOptions": {

    /* 基本选项 */
```

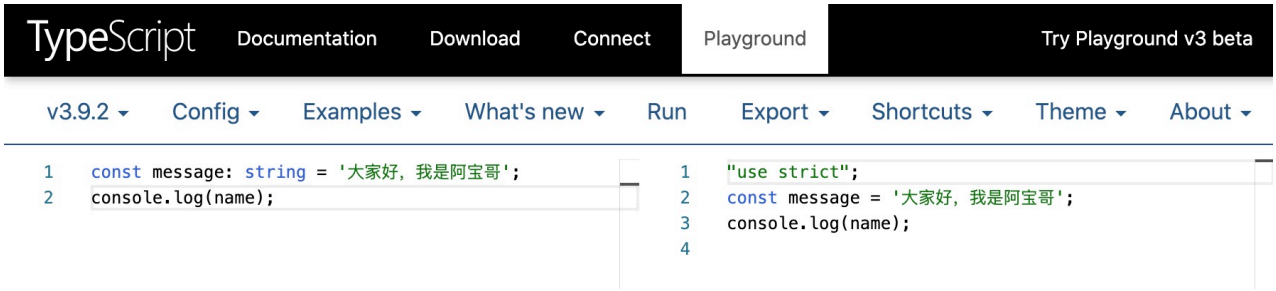
```
    "target": "es5",                          // 指定 ECMAScript 目标版本: 'ES3'
(default), 'ES5', 'ES6'/'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
    "module": "commonjs",                     // 指定使用模块: 'commonjs', 'amd',
'system', 'umd' or 'es2015'
    "lib": [],                                // 指定要包含在编译中的库文件
    "allowJs": true,                          // 允许编译 javascript 文件
    "checkJs": true,                          // 报告 javascript 文件中的错误
    "jsx": "preserve",                        // 指定 jsx 代码的生成: 'preserve',
'react-native', or 'react'
    "declaration": true,                      // 生成相应的 '.d.ts' 文件
    "sourceMap": true,                        // 生成相应的 '.map' 文件
    "outFile": "./",                          // 将输出文件合并为一个文件
    "outDir": "./",                           // 指定输出目录
    "rootDir": "./",                          // 用来控制输出目录结构 --outDir.
    "removeComments": true,                   // 删除编译后的所有的注释
    "noEmit": true,                           // 不生成输出文件
    "importHelpers": true,                    // 从 tslib 导入辅助工具函数
    "isolatedModules": true,                  // 将每个文件做为单独的模块 （与
'ts.transpileModule' 类似）.

    /* 严格的类型检查选项 */
    "strict": true,                           // 启用所有严格类型检查选项
    "noImplicitAny": true,                    // 在表达式和声明上有隐含的 any类型时报错
    "strictNullChecks": true,                 // 启用严格的 null 检查
    "noImplicitThis": true,                   // 当 this 表达式值为 any 类型的时候，生
成一个错误
    "alwaysStrict": true,                     // 以严格模式检查每个模块，并在每个文件里加
入 'use strict'

    /* 额外的检查 */
    "noUnusedLocals": true,                   // 有未使用的变量时，抛出错误
    "noUnusedParameters": true,               // 有未使用的参数时，抛出错误
    "noImplicitReturns": true,                // 并不是所有函数里的代码都有返回值时，抛出
错误
    "noFallthroughCasesInSwitch": true,       // 报告 switch 语句的 fallthrough 错
误。（即，不允许 switch 的 case 语句贯穿）

    /* 模块解析选项 */
    "moduleResolution": "node",               // 选择模块解析策略： 'node' (Node.js)
or 'classic' (TypeScript pre-1.6)
    "baseUrl": "./",                          // 用于解析非相对模块名称的基目录
    "paths": {},                              // 模块名到基于 baseUrl 的路径映射的列表
    "rootDirs": [],                           // 根文件夹列表，其组合内容表示项目运行时的
结构内容
    "typeRoots": [],                          // 包含类型声明的文件列表
    "types": [],                              // 需要包含的类型声明文件名列表
    "allowSyntheticDefaultImports": true,     // 允许从没有设置默认导出的模块中默认导入。

    /* Source Map Options */
```

```
    "sourceRoot": "./",                     // 指定调试器应该找到 TypeScript 文件而
不是源文件的位置
    "mapRoot": "./",                        // 指定调试器应该找到映射文件而不是生成文件
的位置
    "inlineSourceMap": true,                // 生成单个 soucemaps 文件，而不是将
sourcemaps 生成不同的文件
    "inlineSources": true,                  // 将代码与 sourcemaps 生成到一个文件
中，要求同时设置了 --inlineSourceMap 或 --sourceMap 属性

    /* 其他选项 */
    "experimentalDecorators": true,         // 启用装饰器
    "emitDecoratorMetadata": true           // 为装饰器提供元数据的支持
  }
}
```

十六、TypeScript 开发辅助工具

16.1 TypeScript Playground

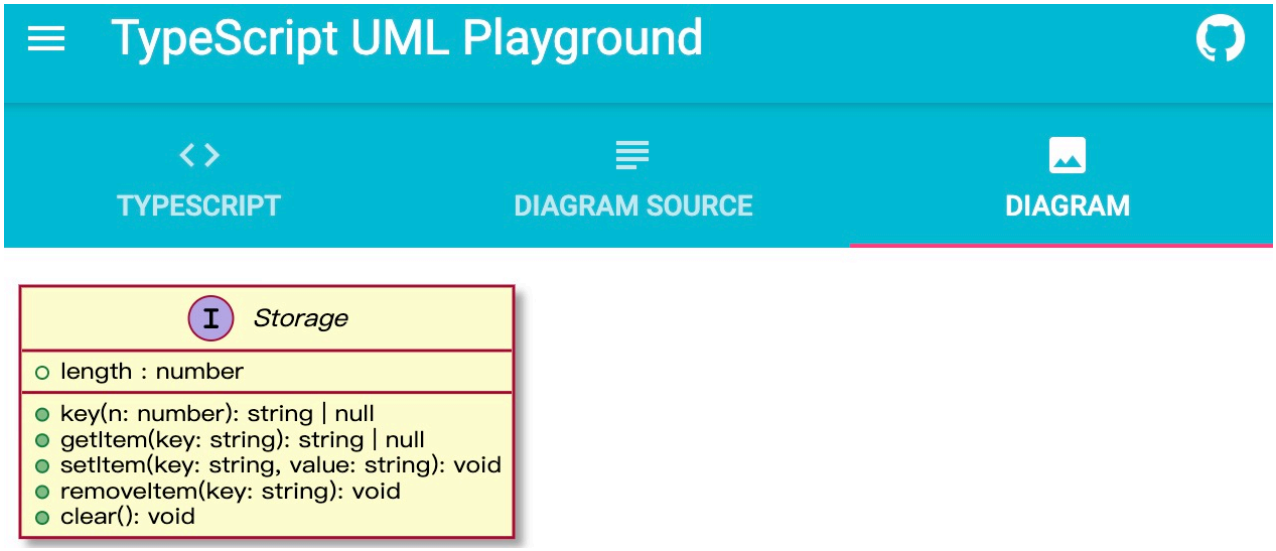
> 简介：TypeScript 官方提供的在线 TypeScript 运行环境，利用它你可以方便地学习 TypeScript 相关知识与不同版本的功能特性。
>
> 在线地址：https://www.typescriptlang.org/play/



除了 TypeScript 官方的 Playground 之外，你还可以选择其他的 Playground，比如 codepen.io、stackblitz 或 jsbin.com 等。

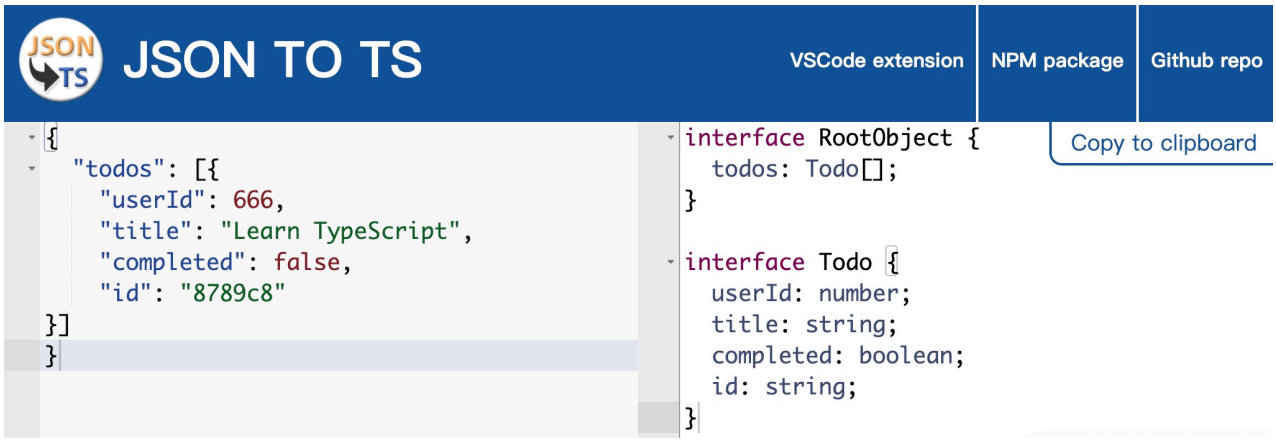
16.2 TypeScript UML Playground

> 简介：一款在线 TypeScript UML 工具，利用它你可以为指定的 TypeScript 代码生成 UML 类图。
>
> 在线地址：https://tsuml-demo.firebaseapp.com/



16.3 JSON TO TS

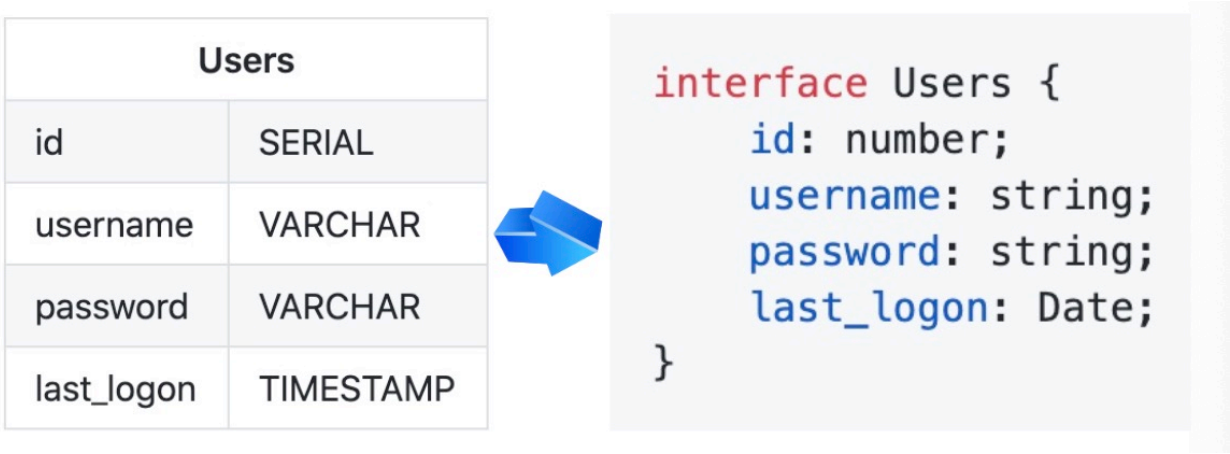
> 简介：一款 TypeScript 在线工具，利用它你可以为指定的 JSON 数据生成对应的 TypeScript 接口定义。
>
> 在线地址：http://www.jsontots.com/



除了使用 jsontots 在线工具之外，对于使用 VSCode IDE 的小伙们还可以安装 JSON to TS 扩展来快速完成 **JSON to TS** 的转换工作。

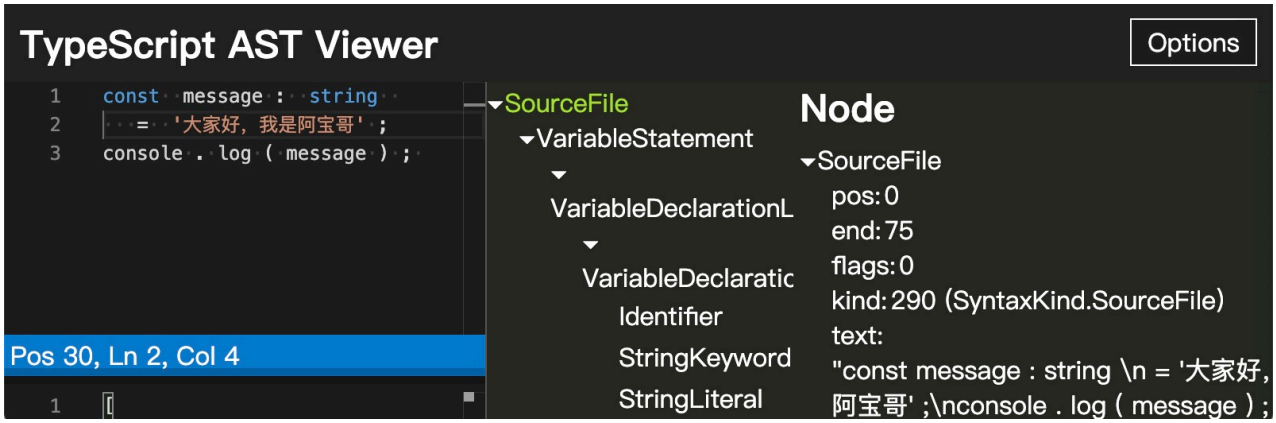
16.4 Schemats

> 简介：利用 Schemats，你可以基于（Postgres，MySQL）SQL 数据库中的 schema 自动生成 TypeScript 接口定义。
>
> 在线地址：https://github.com/SweetIQ/schemats



16.5 TypeScript AST Viewer

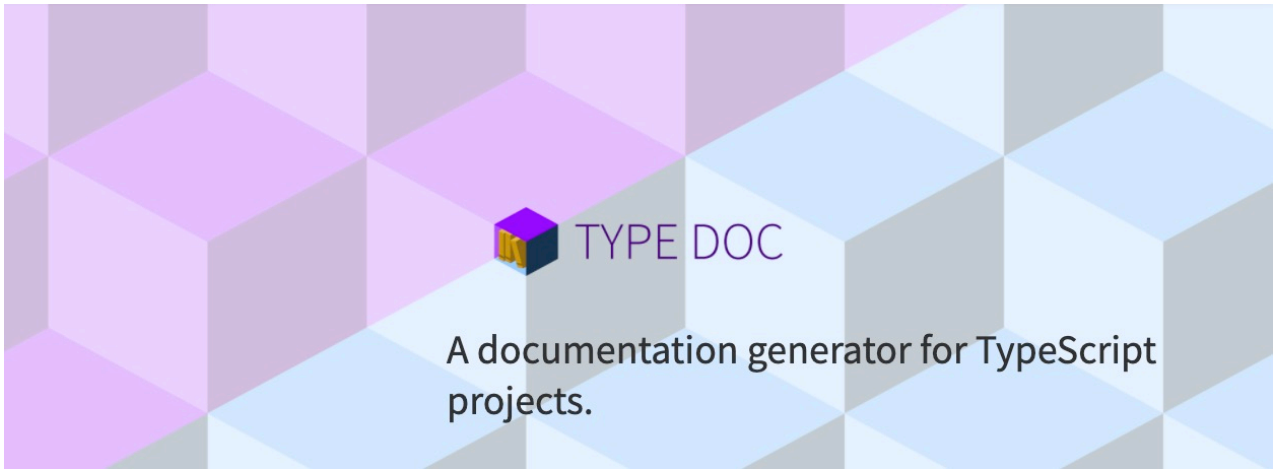
> 简介：一款 TypeScript AST 在线工具，利用它你可以查看指定 TypeScript 代码对应的 AST（Abstract Syntax Tree）抽象语法树。
>
> 在线地址：https://ts-ast-viewer.com/



对于了解过 AST 的小伙伴来说，对 astexplorer 这款在线工具应该不会陌生。该工具除了支持 JavaScript 之外，还支持 CSS、JSON、RegExp、GraphQL 和 Markdown 等格式的解析。

16.6 TypeDoc

> 简介：TypeDoc 用于将 TypeScript 源代码中的注释转换为 HTML 文档或 JSON 模型。它可灵活扩展，并支持多种配置。
>
> 在线地址：https://typedoc.org/



16.7 TypeScript ESLint

> 简介：使用 TypeScript ESLint 可以帮助我们规范代码质量，提高团队开发效率。
>
> 在线地址：https://typescript-eslint.io/

对 TypeScript ESLint 项目感兴趣且想在项目中应用的小伙伴，可以参考 “在Typescript项目中，如何优雅的使用ESLint和Prettier” 这篇文章。

能坚持看到这里的小伙伴都是 “真爱”，如果你还意犹未尽，那就来看看本人整理的 Github 上 1.8K+ 的开源项目：awesome-typescript。

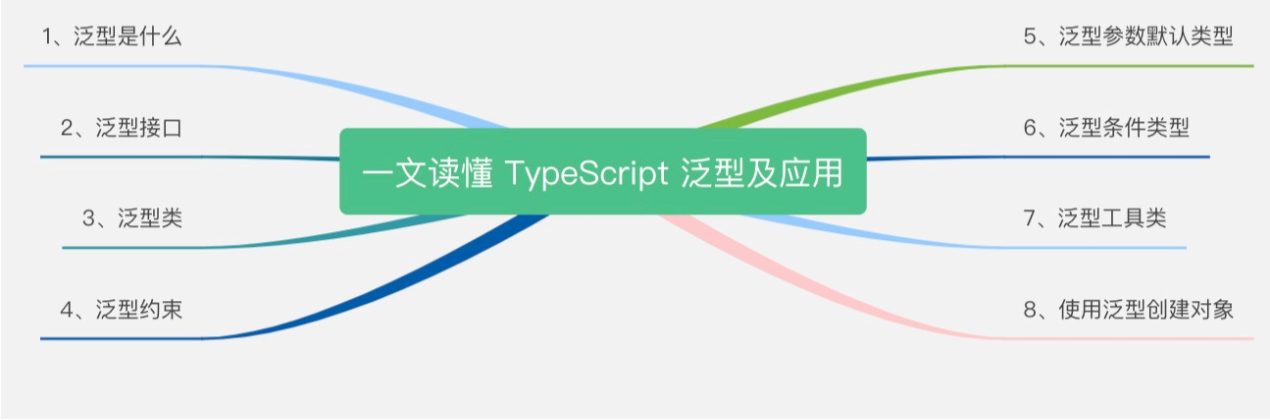
> https://github.com/semlinker/awesome-typescript

十七、参考资源

- mariusschulz - the-unknown-type-in-typescript
- 深入理解 TypeScript - 编译上下文
- TypeScript 4.0
- TypeScript Quickly

第二章 一文读懂 TypeScript 泛型及应用

本章将从八个方面入手，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：



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：

```ts
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。

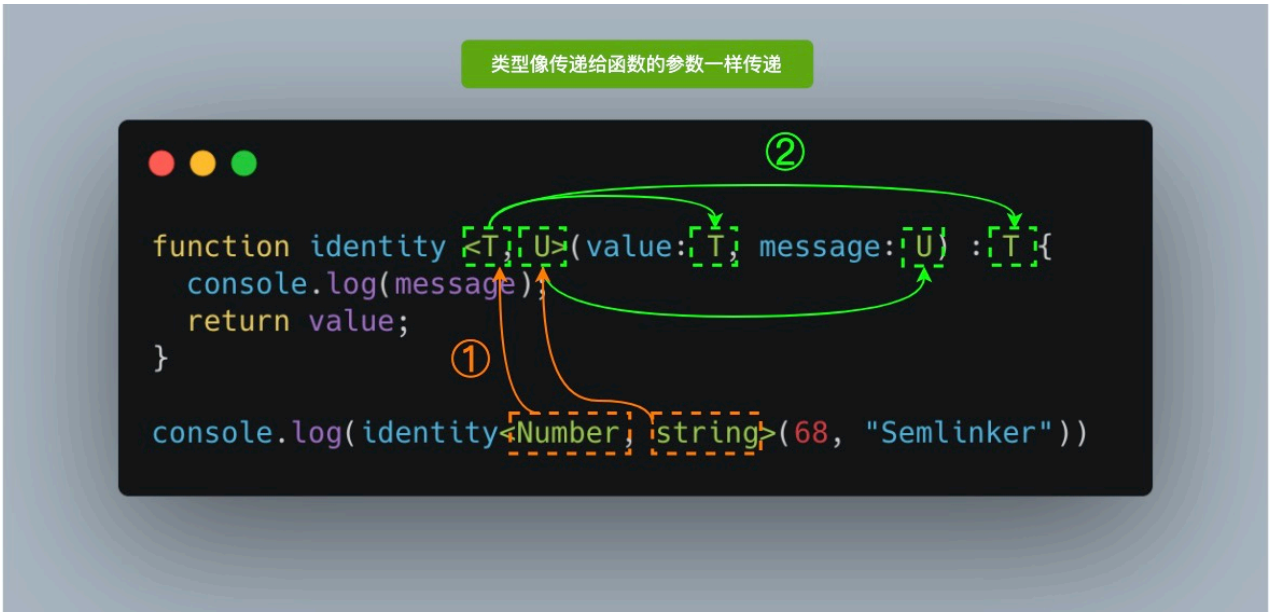
参考上面的图片，当我们调用 `identity<Number>(1)` ， `Number` 类型就像参数 `1` 一样，它将在出现 `T` 的任何位置填充该类型。图中 `<T>` 内部的 `T` 被称为类型变量，它是我们希望传递给 identity 函数的类型占位符，同时它被分配给 `value` 参数用来代替它的类型：此时 `T` 充当的是类型，而不是特定的 Number 类型。

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：

```ts
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。如果想直观地感受一下类型传递的过程，可以观看 这篇文章 中的动画。

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。

我们将在本章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，我们得来了解一下 `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(tsInfo, '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，我们通常还会结合 `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;
}
```

```ts
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：

```ts
/**
 * 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：

```ts
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：

```ts
// 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：

```ts
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：

```ts
// 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：

```ts
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、参考资源

- typescript-generics
- typescript-generics-explained
- typescript-tip-of-the-week-generics

第三章 细数 TS 中那些奇怪的符号

本章阿宝哥将分享这些年在学习 TypeScript 过程中，遇到的 9 大 “奇怪” 的符号。下面我们来开始介绍第一个符号 —— **! 非空断言操作符**。

一、! 非空断言操作符

在上下文中当类型检查器无法断定类型时，一个新的后缀表达式操作符 `!` 可以用于断言操作对象是非 null 和非 undefined 类型。**具体而言，x! 将从 x 值域中排除 null 和 undefined 。**

那么非空断言操作符到底有什么用呢？下面我们先来看一下非空断言操作符的一些使用场景。

1.1 忽略 undefined 和 null 类型

```
function myFunc(maybeString: string | undefined | null) {
  // Type 'string | null | undefined' is not assignable to type 'string'.
  // Type 'undefined' is not assignable to type 'string'.
  const onlyString: string = maybeString; // Error
  const ignoreUndefinedAndNull: string = maybeString!; // Ok
}
```

1.2 调用函数时忽略 undefined 类型

```
type NumGenerator = () => number;

function myFunc(numGenerator: NumGenerator | undefined) {
  // Object is possibly 'undefined'.(2532)
  // Cannot invoke an object which is possibly 'undefined'.(2722)
  const num1 = numGenerator(); // Error
  const num2 = numGenerator!(); //OK
}
```

因为 `!` 非空断言操作符会从编译生成的 JavaScript 代码中移除，所以在实际使用的过程中，要特别注意。比如下面这个例子：

```
const a: number | undefined = undefined;
const b: number = a!;
console.log(b);
```

以上 TS 代码会编译生成以下 ES5 代码：

```
"use strict";
const a = undefined;
const b = a;
console.log(b);
```

虽然在 TS 代码中，我们使用了非空断言，使得 `const b: number = a!;` 语句可以通过 TypeScript 类型检查器的检查。但在生成的 ES5 代码中，`!` 非空断言操作符被移除了，所以在浏览器中执行以上代码，在控制台会输出 `undefined`。

1.3 确定赋值断言

在 TypeScript 2.7 版本中引入了确定赋值断言，即允许在实例属性和变量声明后面放置一个 `!` 号，从而告诉 TypeScript 该属性会被明确地赋值。为了更好地理解它的作用，我们来看个具体的例子：

```
let x: number;
initialize();
// Variable 'x' is used before being assigned.(2454)
console.log(2 * x); // Error

function initialize() {
  x = 10;
}
```

很明显该异常信息是说变量 x 在赋值前被使用了，要解决该问题，我们可以使用确定赋值断言：

```
let x!: number;
initialize();
console.log(2 * x); // Ok

function initialize() {
  x = 10;
}
```

通过 `let x!: number;` 确定赋值断言，TypeScript 编译器就会知道该属性会被明确地赋值。

二、?. 运算符

TypeScript 3.7 实现了呼声最高的 ECMAScript 功能之一：可选链（Optional Chaining）。有了可选链后，我们编写代码时如果遇到 `null` 或 `undefined` 就可以立即停止某些表达式的运行。可选链的核心是新的 `?.` 运算符，它支持以下语法：

```
obj?.prop
obj?.[expr]
arr?.[index]
func?.(args)
```

这里我们来举一个可选的属性访问的例子：

```
const val = a?.b;
```

为了更好的理解可选链，我们来看一下该 `const val = a?.b` 语句编译生成的 ES5 代码：

```
var val = a === null || a === void 0 ? void 0 : a.b;
```

上述的代码会自动检查对象 a 是否为 `null` 或 `undefined`，如果是的话就立即返回 `undefined`，这样就可以立即停止某些表达式的运行。你可能已经想到可以使用 `?.` 来替代很多使用 `&&` 执行空检查的代码：

```
if(a && a.b) { }

if(a?.b){ }
/**
* if(a?.b){ } 编译后的ES5代码
*
* if(
*  a === null || a === void 0
*  ? void 0 : a.b) {
* }
*/
```

但需要注意的是，`?.` 与 `&&` 运算符行为略有不同，`&&` 专门用于检测 `falsy` 值，比如空字符串、0、NaN、null 和 false 等。而 `?.` 只会验证对象是否为 `null` 或 `undefined`，对于 0 或空字符串来说，并不会出现 “短路”。

2.1 可选元素访问

可选链除了支持可选属性的访问之外，它还支持可选元素的访问，它的行为类似于可选属性的访问，只是可选元素的访问允许我们访问非标识符的属性，比如任意字符串、数字索引和 Symbol：

```
function tryGetArrayElement<T>(arr?: T[], index: number = 0) {
  return arr?.[index];
}
```

以上代码经过编译后会生成以下 ES5 代码：

```
"use strict";
function tryGetArrayElement(arr, index) {
    if (index === void 0) { index = 0; }
    return arr === null || arr === void 0 ? void 0 : arr[index];
}
```

通过观察生成的 ES5 代码，很明显在 `tryGetArrayElement` 方法中会自动检测输入参数 arr 的值是否为 `null` 或 `undefined`，从而保证了我们代码的健壮性。

2.2 可选链与函数调用

当尝试调用一个可能不存在的方法时也可以使用可选链。在实际开发过程中，这是很有用的。系统中某个方法不可用，有可能是由于版本不一致或者用户设备兼容性问题导致的。函数调用时如果被调用的方法不存在，使用可选链可以使表达式自动返回 `undefined` 而不是抛出一个异常。

可选调用使用起来也很简单，比如：

```
let result = obj.customMethod?.();
```

该 TypeScript 代码编译生成的 ES5 代码如下：

```
var result = (_a = obj.customMethod) === null
  || _a === void 0 ? void 0 : _a.call(obj);
```

另外在使用可选调用的时候，我们要注意以下两个注意事项：

- 如果存在一个属性名且该属性名对应的值不是函数类型，使用 `?.` 仍然会产生一个 `TypeError` 异常。
- 可选链的运算行为被局限在属性的访问、调用以及元素的访问 —— 它不会沿伸到后续的表达式中，也就是说可选调用不会阻止 `a?.b / someMethod()` 表达式中的除法运算或 `someMethod` 的方法调用。

三、?? 空值合并运算符

在 TypeScript 3.7 版本中除了引入了前面介绍的可选链 `?.` 之外，也引入了一个新的逻辑运算符 —— 空值合并运算符 `??`。**当左侧操作数为 null 或 undefined 时，其返回右侧的操作数，否则返回左侧的操作数。**

与逻辑或 `||` 运算符不同，逻辑或会在左操作数为 falsy 值时返回右侧操作数。也就是说，如果你使用 || 来为某些变量设置默认的值时，你可能会遇到意料之外的行为。比如为 falsy 值（''、NaN 或 0）时。

这里来看一个具体的例子：

```
const foo = null ?? 'default string';
console.log(foo); // 输出："default string"

const baz = 0 ?? 42;
console.log(baz); // 输出：0
```

以上 TS 代码经过编译后，会生成以下 ES5 代码：

```
"use strict";
var _a, _b;
var foo = (_a = null) !== null && _a !== void 0 ? _a : 'default string';
console.log(foo); // 输出："default string"

var baz = (_b = 0) !== null && _b !== void 0 ? _b : 42;
console.log(baz); // 输出：0
```

通过观察以上代码，我们更加直观的了解到，空值合并运算符是如何解决前面 `||` 运算符存在的潜在问题。下面我们来介绍空值合并运算符的特性和使用时的一些注意事项。

3.1 短路

当空值合并运算符的左表达式不为 `null` 或 `undefined` 时，不会对右表达式进行求值。

```
function A() { console.log('A was called'); return undefined;}
function B() { console.log('B was called'); return false;}
function C() { console.log('C was called'); return "foo";}

console.log(A() ?? C());
console.log(B() ?? C());
```

上述代码运行后，控制台会输出以下结果：

```
A was called
C was called
foo
B was called
false
```

3.2 不能与 && 或 || 操作符共用

若空值合并运算符 `??` 直接与 AND（&&）和 OR（||）操作符组合使用 ?? 是不行的。这种情况下会抛出 SyntaxError。

```
// '||' and '??' operations cannot be mixed without parentheses.(5076)
null || undefined ?? "foo"; // raises a SyntaxError

// '&&' and '??' operations cannot be mixed without parentheses.(5076)
true && undefined ?? "foo"; // raises a SyntaxError
```

但当使用括号来显式表明优先级时是可行的，比如：

```
(null || undefined ) ?? "foo"; // 返回 "foo"
```

3.3 与可选链操作符 ?. 的关系

空值合并运算符针对 undefined 与 null 这两个值，可选链式操作符 `?.` 也是如此。可选链式操作符，对于访问属性可能为 undefined 与 null 的对象时非常有用。

```
interface Customer {
  name: string;
  city?: string;
}

let customer: Customer = {
  name: "Semlinker"
};

let customerCity = customer?.city ?? "Unknown city";
console.log(customerCity); // 输出：Unknown city
```

前面我们已经介绍了空值合并运算符的应用场景和使用时的一些注意事项，该运算符不仅可以在 TypeScript 3.7 以上版本中使用。当然你也可以在 JavaScript 的环境中使用它，但你需要借助 Babel，在 Babel 7.8.0 版本也开始支持空值合并运算符。

四、?: 可选属性

在面向对象语言中，接口是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类去实现。**TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。**

在 TypeScript 中使用 `interface` 关键字就可以声明一个接口：

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

let semlinker: Person = {
  name: "semlinker",
  age: 33,
};
```

在以上代码中，我们声明了 `Person` 接口，它包含了两个必填的属性 `name` 和 `age`。在初始化 Person 类型变量时，如果缺少某个属性，TypeScript 编译器就会提示相应的错误信息，比如：

```
// Property 'age' is missing in type '{ name: string; }' but required in type
'Person'.(2741)
let lolo: Person  = { // Error
  name: "lolo"
}
```

为了解决上述的问题，我们可以把某个属性声明为可选的：

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

let lolo: Person  = {
  name: "lolo"
}
```

4.1 工具类型

4.1.1 `Partial<T>`

在实际项目开发过程中，为了提高代码复用率，我们可以利用 TypeScript 内置的工具类型 `Partial<T>` 来快速把某个接口类型中定义的属性变成可选的：

```
interface PullDownRefreshConfig {
  threshold: number;
  stop: number;
}

/**
 * type PullDownRefreshOptions = {
 *   threshold?: number | undefined;
 *   stop?: number | undefined;
 * }
 */
type PullDownRefreshOptions = Partial<PullDownRefreshConfig>
```

是不是觉得 `Partial<T>` 很方便，下面让我们来看一下它是如何实现的：

```
/**
 * Make all properties in T optional
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

4.1.2 `Required<T>`

既然可以快速地把某个接口中定义的属性全部声明为可选，那能不能把所有的可选的属性变成必选的呢？答案是可以的，针对这个需求，我们可以使用 `Required<T>` 工具类型，具体的使用方式如下：

```
interface PullDownRefreshConfig {
  threshold: number;
  stop: number;
```

```
}

type PullDownRefreshOptions = Partial<PullDownRefreshConfig>

/**
 * type PullDownRefresh = {
 *   threshold: number;
 *   stop: number;
 * }
 */
type PullDownRefresh = Required<Partial<PullDownRefreshConfig>>
```

同样，我们来看一下 `Required<T>` 工具类型是如何实现的：

```
/**
 * Make all properties in T required
 */
type Required<T> = {
  [P in keyof T]-?: T[P];
};
```

原来在 `Required<T>` 工具类型内部，通过 `-?` 移除了可选属性中的 `?`，使得属性从可选变为必选的。

五、& 运算符

在 TypeScript 中交叉类型是将多个类型合并为一个类型。通过 `&` 运算符可以将现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。

```
type PartialPointX = { x: number; };
type Point = PartialPointX & { y: number; };

let point: Point = {
  x: 1,
  y: 1
}
```

在上面代码中我们先定义了 `PartialPointX` 类型，接着使用 `&` 运算符创建一个新的 `Point` 类型，表示一个含有 x 和 y 坐标的点，然后定义了一个 `Point` 类型的变量并初始化。

5.1 同名基础类型属性的合并

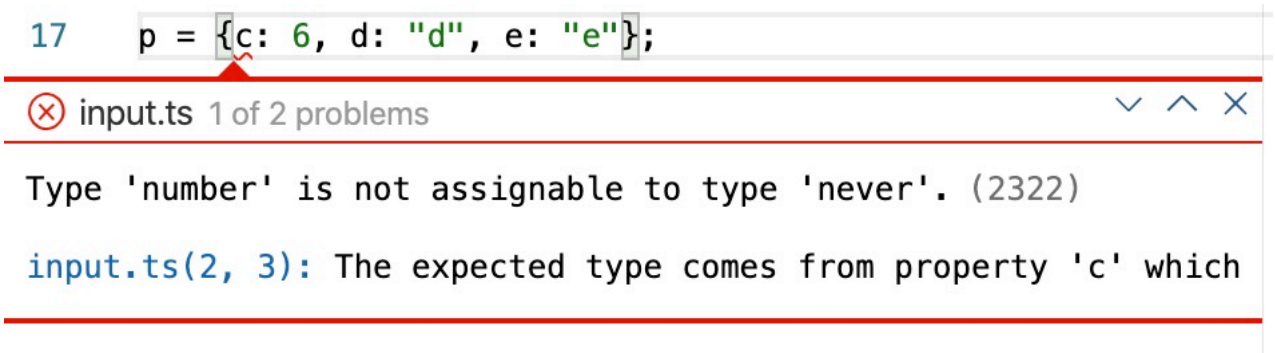
那么现在问题来了，假设在合并多个类型的过程中，刚好出现某些类型存在相同的成员，但对应的类型又不一致，比如：

```
interface X {
  c: string;
  d: string;
}
```

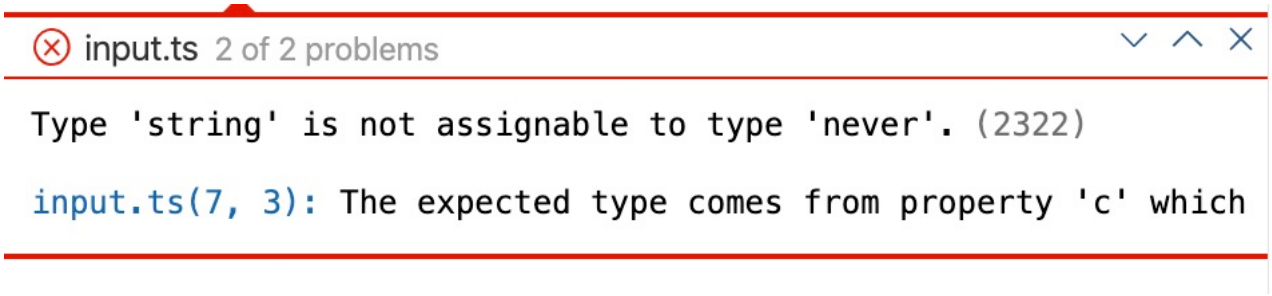
```ts
interface Y {
  c: number;
  e: string
}

type XY = X & Y;
type YX = Y & X;

let p: XY;
let q: YX;
```

在上面的代码中，接口 X 和接口 Y 都含有一个相同的成员 c，但它们的类型不一致。对于这种情况，此时 XY 类型或 YX 类型中成员 c 的类型是不是可以是 `string` 或 `number` 类型呢？比如下面的例子：

```ts
p = { c: 6, d: "d", e: "e" };
```




```ts
q = { c: "c", d: "d", e: "e" };
```



为什么接口 X 和接口 Y 混入后，成员 c 的类型会变成 `never` 呢？这是因为混入后成员 c 的类型为 `string & number`，即成员 c 的类型既可以是 `string` 类型又可以是 `number` 类型。很明显这种类型是不存在的，所以混入后成员 c 的类型为 `never`。

5.2 同名非基础类型属性的合并

在上面示例中，刚好接口 X 和接口 Y 中内部成员 c 的类型都是基本数据类型，那么如果是非基本数据类型的话，又会是什么情形。我们来看个具体的例子：

```ts
interface D { d: boolean; }
interface E { e: string; }
```

```ts
interface F { f: number; }

interface A { x: D; }
interface B { x: E; }
interface C { x: F; }

type ABC = A & B & C;

let abc: ABC = {
  x: {
    d: true,
    e: 'semlinker',
    f: 666
  }
};

console.log('abc:', abc);
```

以上代码成功运行后，控制台会输出以下结果：

由上图可知，在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，那么是可以成功合并。

六、| 分隔符

在 TypeScript 中联合类型（Union Types）表示取值可以为多种类型中的一种，联合类型使用 `|` 分隔每个类型。联合类型通常与 `null` 或 `undefined` 一起使用：

```ts
const sayHello = (name: string | undefined) => { /* ... */ };
```

以上示例中 `name` 的类型是 `string | undefined` 意味着可以将 `string` 或 `undefined` 的值传递给 `sayHello` 函数。

```ts
sayHello("semlinker");
sayHello(undefined);
```

此外，对于联合类型来说，你可能会遇到以下的用法：

```
let num: 1 | 2 = 1;
type EventNames = 'click' | 'scroll' | 'mousemove';
```

示例中的 `1`、`2` 或 `'click'` 被称为字面量类型，用来约束取值只能是某几个值中的一个。

6.1 类型保护

当使用联合类型时，我们必须尽量把当前值的类型收窄为当前值的实际类型，而类型保护就是实现类型收窄的一种手段。

类型保护是可执行运行时检查的一种表达式，用于确保该类型在一定的范围内。换句话说，类型保护可以保证一个字符串是一个字符串，尽管它的值也可以是一个数字。类型保护与特性检测并不是完全不同，其主要思想是尝试检测属性、方法或原型，以确定如何处理值。

目前主要有四种的方式来实现类型保护：

6.1.1 in 关键字

```
interface Admin {
  name: string;
  privileges: string[];
}

interface Employee {
  name: string;
  startDate: Date;
}

type UnknownEmployee = Employee | Admin;

function printEmployeeInformation(emp: UnknownEmployee) {
  console.log("Name: " + emp.name);
  if ("privileges" in emp) {
    console.log("Privileges: " + emp.privileges);
  }
  if ("startDate" in emp) {
    console.log("Start Date: " + emp.startDate);
  }
}
```

6.1.2 typeof 关键字

```
function padLeft(value: string, padding: string | number) {
  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}'.`);
}
```

`typeof` 类型保护只支持两种形式：`typeof v === "typename"` 和 `typeof v !== typename`，`"typename"` 必须是 `"number"`，`"string"`，`"boolean"` 或 `"symbol"`。但是 TypeScript 并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型保护。

6.1.3 instanceof 关键字

```
interface Padder {
  getPaddingString(): string;
}

class SpaceRepeatingPadder implements Padder {
  constructor(private numSpaces: number) {}
  getPaddingString() {
    return Array(this.numSpaces + 1).join(" ");
  }
}

class StringPadder implements Padder {
  constructor(private value: string) {}
  getPaddingString() {
    return this.value;
  }
}

let padder: Padder = new SpaceRepeatingPadder(6);

if (padder instanceof SpaceRepeatingPadder) {
  // padder的类型收窄为 'SpaceRepeatingPadder'
}
```

6.1.4 自定义类型保护的类型谓词（type predicate）

```ts
function isNumber(x: any): x is number {
  return typeof x === "number";
}

function isString(x: any): x is string {
  return typeof x === "string";
}
```

七、_ 数字分隔符

TypeScript 2.7 带来了对数字分隔符的支持，正如数值分隔符 ECMAScript 提案中所概述的那样。对于一个数字字面量，你现在可以通过把一个下划线作为它们之间的分隔符来分组数字：

```ts
const inhabitantsOfMunich = 1_464_301;
const distanceEarthSunInKm = 149_600_000;
const fileSystemPermission = 0b111_111_000;
const bytes = 0b1111_10101011_11110000_00001101;
```

分隔符不会改变数值字面量的值，但逻辑分组使人们更容易一眼就能读懂数字。以上 TS 代码经过编译后，会生成以下 ES5 代码：

```ts
"use strict";
var inhabitantsOfMunich = 1464301;
var distanceEarthSunInKm = 149600000;
var fileSystemPermission = 504;
var bytes = 262926349;
```

7.1 使用限制

虽然数字分隔符看起来很简单，但在使用时还是有一些限制。比如你只能在两个数字之间添加 `_` 分隔符。以下的使用方式是非法的：

```ts
// Numeric separators are not allowed here.(6188)
3_.141592 // Error
3._141592 // Error

// Numeric separators are not allowed here.(6188)
1_e10 // Error
1e_10 // Error

// Cannot find name '_126301'.(2304)
_126301  // Error
// Numeric separators are not allowed here.(6188)
126301_ // Error

// Cannot find name 'b111111000'.(2304)
// An identifier or keyword cannot immediately follow a numeric literal.(1351)
```

```
0_b111111000 // Error

// Numeric separators are not allowed here.(6188)
0b_111111000 // Error
```

当然你也不能连续使用多个 `_` 分隔符，比如：

```
// Multiple consecutive numeric separators are not permitted.(6189)
123__456 // Error
```

7.2 解析分隔符

此外，需要注意的是以下用于解析数字的函数是不支持分隔符：

- `Number()`
- `parseInt()`
- `parseFloat()`

这里我们来看一下实际的例子：

```
Number('123_456')
NaN
parseInt('123_456')
123
parseFloat('123_456')
123
```

很明显对于以上的结果不是我们所期望的，所以在处理分隔符时要特别注意。当然要解决上述问题，也很简单只需要非数字的字符删掉即可。这里我们来定义一个 `removeNonDigits` 的函数：

```
const RE_NON_DIGIT = /[^0-9]/gu;

function removeNonDigits(str) {
  str = str.replace(RE_NON_DIGIT, '');
  return Number(str);
}
```

该函数通过调用字符串的 `replace` 方法来移除非数字的字符，具体的使用方式如下：

```
removeNonDigits('123_456')
123456
removeNonDigits('149,600,000')
149600000
removeNonDigits('1,407,836')
1407836
```

八、@XXX 装饰器

8.1 装饰器语法

对于一些刚接触 TypeScript 的小伙伴来说，在第一次看到 `@Plugin({...})` 这种语法可能会觉得很惊讶。其实这是装饰器的语法，装饰器的本质是一个函数，通过装饰器我们可以方便地定义与对象相关的元数据。

```
@Plugin({
  pluginName: 'Device',
  plugin: 'cordova-plugin-device',
  pluginRef: 'device',
  repo: 'https://github.com/apache/cordova-plugin-device',
  platforms: ['Android', 'Browser', 'iOS', 'macOS', 'Windows'],
})
@Injectable()
export class Device extends IonicNativePlugin {}
```

在以上代码中，我们通过装饰器来保存 ionic-native 插件的相关元信息，而 `@Plugin({...})` 中的 `@` 符号只是语法糖，为什么说是语法糖呢？这里我们来看一下编译生成的 ES5 代码：

```
var __decorate = (this && this.__decorate) || function (decorators, target,
key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc =
Object.getOwnPropertyDescriptor(target, key) : desc, d;
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function")
r = Reflect.decorate(decorators, target, key, desc);
    else for (var i = decorators.length - 1; i >= 0; i--) if (d =
decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r) : d(target, key))
|| r;
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};

var Device = /** @class */ (function (_super) {
    __extends(Device, _super);
    function Device() {
        return _super !== null && _super.apply(this, arguments) || this;
    }
    Device = __decorate([
        Plugin({
            pluginName: 'Device',
            plugin: 'cordova-plugin-device',
            pluginRef: 'device',
            repo: 'https://github.com/apache/cordova-plugin-device',
            platforms: ['Android', 'Browser', 'iOS', 'macOS', 'Windows'],
        }),
        Injectable()
    ], Device);
```

```
    return Device;
}(IonicNativePlugin));
```

通过生成的代码可知，`@Plugin({...})` 和 `@Injectable()` 最终会被转换成普通的方法调用，它们的调用结果最终会以数组的形式作为参数传递给 `__decorate` 函数，而在 `__decorate` 函数内部会以 `Device` 类作为参数调用各自的类型装饰器，从而扩展对应的功能。

8.2 装饰器的分类

在 TypeScript 中装饰器分为类装饰器、属性装饰器、方法装饰器和参数装饰器四大类。

8.2.1 类装饰器

类装饰器声明：

```
declare type ClassDecorator = <TFunction extends Function>(
  target: TFunction
) => TFunction | void;
```

类装饰器顾名思义，就是用来装饰类的。它接收一个参数：

- target: TFunction - 被装饰的类

看完第一眼后，是不是感觉都不好了。没事，我们马上来个例子：

```
function Greeter(target: Function): void {
  target.prototype.greet = function (): void {
    console.log("Hello Semlinker!");
  };
}

@Greeter
class Greeting {
  constructor() {
    // 内部实现
  }
}

let myGreeting = new Greeting();
(myGreeting as any).greet(); // console output: 'Hello Semlinker!';
```

上面的例子中，我们定义了 `Greeter` 类装饰器，同时我们使用了 `@Greeter` 语法糖，来使用装饰器。

> 友情提示：读者可以直接复制上面的代码，在 TypeScript Playground 中运行查看结果。

8.2.2 属性装饰器

属性装饰器声明：

```
declare type PropertyDecorator = (target:Object,
  propertyKey: string | symbol ) => void;
```

属性装饰器顾名思义，用来装饰类的属性。它接收两个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 被装饰类的属性名

趁热打铁，马上来个例子热热身：

```
function logProperty(target: any, key: string) {
  delete target[key];

  const backingField = "_" + key;

  Object.defineProperty(target, backingField, {
    writable: true,
    enumerable: true,
    configurable: true
  });

  // property getter
  const getter = function (this: any) {
    const currVal = this[backingField];
    console.log(`Get: ${key} => ${currVal}`);
    return currVal;
  };

  // property setter
  const setter = function (this: any, newVal: any) {
    console.log(`Set: ${key} => ${newVal}`);
    this[backingField] = newVal;
  };

  // Create new property with getter and setter
  Object.defineProperty(target, key, {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true
  });
}

class Person {
  @logProperty
```

```
  public name: string;

  constructor(name : string) {
    this.name = name;
  }
}

const p1 = new Person("semlinker");
p1.name = "kakuqo";
```

以上代码我们定义了一个 `logProperty` 函数，来跟踪用户对属性的操作，当代码成功运行后，在控制台会输出以下结果：

```
Set: name => semlinker
Set: name => kakuqo
```

8.2.3 方法装饰器

方法装饰器声明：

```
declare type MethodDecorator = <T>(target:Object, propertyKey: string | symbol,
  descriptor: TypePropertyDescript<T>) => TypedPropertyDescriptor<T> | void;
```

方法装饰器顾名思义，用来装饰类的方法。它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- descriptor: TypePropertyDescript - 属性描述符

废话不多说，直接上例子：

```
function log(target: Object, propertyKey: string, descriptor:
PropertyDescriptor) {
  let originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log("wrapped function: before invoking " + propertyKey);
    let result = originalMethod.apply(this, args);
    console.log("wrapped function: after invoking " + propertyKey);
    return result;
  };
}

class Task {
  @log
  runTask(arg: any): any {
    console.log("runTask invoked, args: " + arg);
    return "finished";
  }
```

```
}

let task = new Task();
let result = task.runTask("learn ts");
console.log("result: " + result);
```

以上代码成功运行后，控制台会输出以下结果：

```
"wrapped function: before invoking runTask"
"runTask invoked, args: learn ts"
"wrapped function: after invoking runTask"
"result: finished"
```

8.2.4 参数装饰器

参数装饰器声明：

```
declare type ParameterDecorator = (target: Object, propertyKey: string |
symbol,
  parameterIndex: number ) => void
```

参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数：

- target: Object - 被装饰的类
- propertyKey: string | symbol - 方法名
- parameterIndex: number - 方法中参数的索引值

```
function Log(target: Function, key: string, parameterIndex: number) {
  let functionLogged = key || target.prototype.constructor.name;
  console.log(`The parameter in position ${parameterIndex} at ${functionLogged}
has
  been decorated`);
}

class Greeter {
  greeting: string;
  constructor(@Log phrase: string) {
  this.greeting = phrase;
  }
}
```

以上代码成功运行后，控制台会输出以下结果：

```
"The parameter in position 0 at Greeter has been decorated"
```

九、#XXX 私有字段

在 TypeScript 3.8 版本就开始支持 **ECMAScript 私有字段**，使用方式如下：

```
class Person {
  #name: string;

  constructor(name: string) {
    this.#name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}

let semlinker = new Person("Semlinker");

semlinker.#name;
//     ~~~~~
// Property '#name' is not accessible outside class 'Person'
// because it has a private identifier.
```

与常规属性（甚至使用 `private` 修饰符声明的属性）不同，私有字段要牢记以下规则：

- 私有字段以 `#` 字符开头，有时我们称之为私有名称；
- 每个私有字段名称都唯一地限定于其包含的类；
- 不能在私有字段上使用 TypeScript 可访问性修饰符（如 public 或 private）；
- 私有字段不能在包含的类之外访问，甚至不能被检测到。

9.1 私有字段与 private 的区别

说到这里使用 `#` 定义的私有字段与 `private` 修饰符定义字段有什么区别呢？现在我们先来看一个 `private` 的示例：

```
class Person {
  constructor(private name: string){}
}

let person = new Person("Semlinker");
console.log(person.name);
```

在上面代码中，我们创建了一个 Person 类，该类中使用 `private` 修饰符定义了一个私有属性 `name`，接着使用该类创建一个 `person` 对象，然后通过 `person.name` 来访问 `person` 对象的私有属性，这时 TypeScript 编译器会提示以下异常：

```
Property 'name' is private and only accessible within class 'Person'.(2341)
```

那如何解决这个异常呢？当然你可以使用类型断言把 person 转为 any 类型：

```
console.log((person as any).name);
```

通过这种方式虽然解决了 TypeScript 编译器的异常提示，但是在运行时我们还是可以访问到 `Person` 类内部的私有属性，为什么会这样呢？我们来看一下编译生成的 ES5 代码，也许你就知道答案了：

```
var Person = /** @class */ (function () {
    function Person(name) {
      this.name = name;
    }
    return Person;
}());

var person = new Person("Semlinker");
console.log(person.name);
```

这时相信有些小伙伴会好奇，在 TypeScript 3.8 以上版本通过 `#` 号定义的私有字段编译后会生成什么代码：

```
class Person {
  #name: string;

  constructor(name: string) {
    this.#name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}
```

以上代码目标设置为 ES2015，会编译生成以下代码：

```
"use strict";
var __classPrivateFieldSet = (this && this.__classPrivateFieldSet)
  || function (receiver, privateMap, value) {
    if (!privateMap.has(receiver)) {
      throw new TypeError("attempted to set private field on non-instance");
    }
    privateMap.set(receiver, value);
    return value;
};

var __classPrivateFieldGet = (this && this.__classPrivateFieldGet)
  || function (receiver, privateMap) {
    if (!privateMap.has(receiver)) {
```

```
        throw new TypeError("attempted to get private field on non-instance");
    }
    return privateMap.get(receiver);
};

var _name;
class Person {
    constructor(name) {
        _name.set(this, void 0);
        __classPrivateFieldSet(this, _name, name);
    }
    greet() {
        console.log(`Hello, my name is ${__classPrivateFieldGet(this, _name)}!`);
    }
}
_name = new WeakMap();
```

通过观察上述代码，使用 `#` 号定义的 ECMAScript 私有字段，会通过 `WeakMap` 对象来存储，同时编译器会生成 `__classPrivateFieldSet` 和 `__classPrivateFieldGet` 这两个方法用于设置值和获取值。

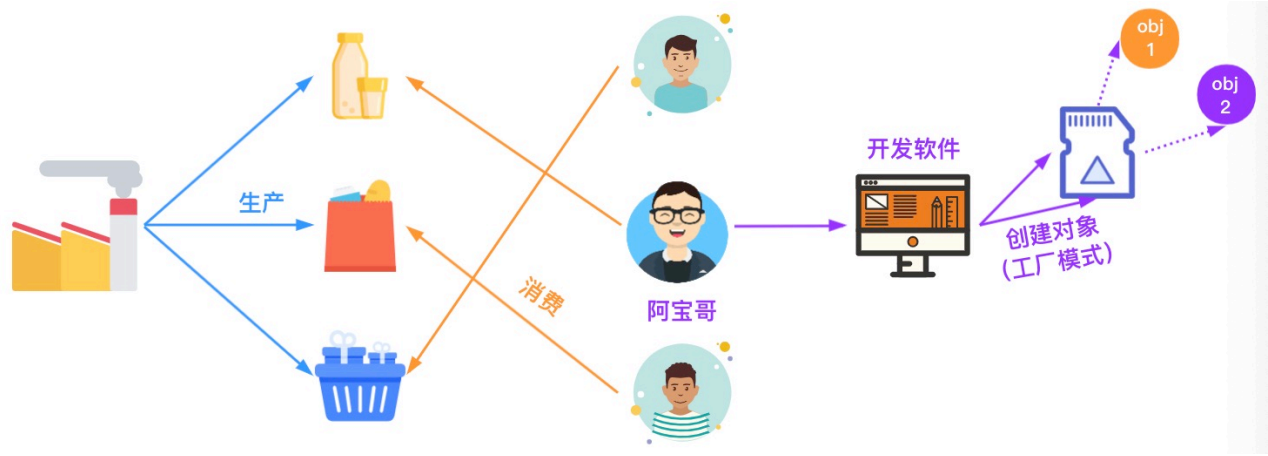
十、参考资源

- ES proposal: numeric separators
- typescriptlang.org

第四章 工厂方法模式

在现实生活中，工厂是负责生产产品的，比如牛奶、面包或礼物等，这些产品满足了我们日常的生理需求。此外，在日常生活中，我们也离不开大大小小的系统，这些系统是由不同的组件对象构成。

而作为一名 Web 软件开发工程师，在软件系统的设计与开发过程中，我们可以利用设计模式来提高代码的可重用性、可扩展性和可维护性。在众多设计模式当中，有一种被称为工厂模式的设计模式，它提供了创建对象的最佳方式。



工厂模式可以分为三类：

- 简单工厂模式（Simple Factory Pattern）
- 工厂方法模式（Factory Method Pattern）
- 抽象工厂模式（Abstract Factory Pattern）

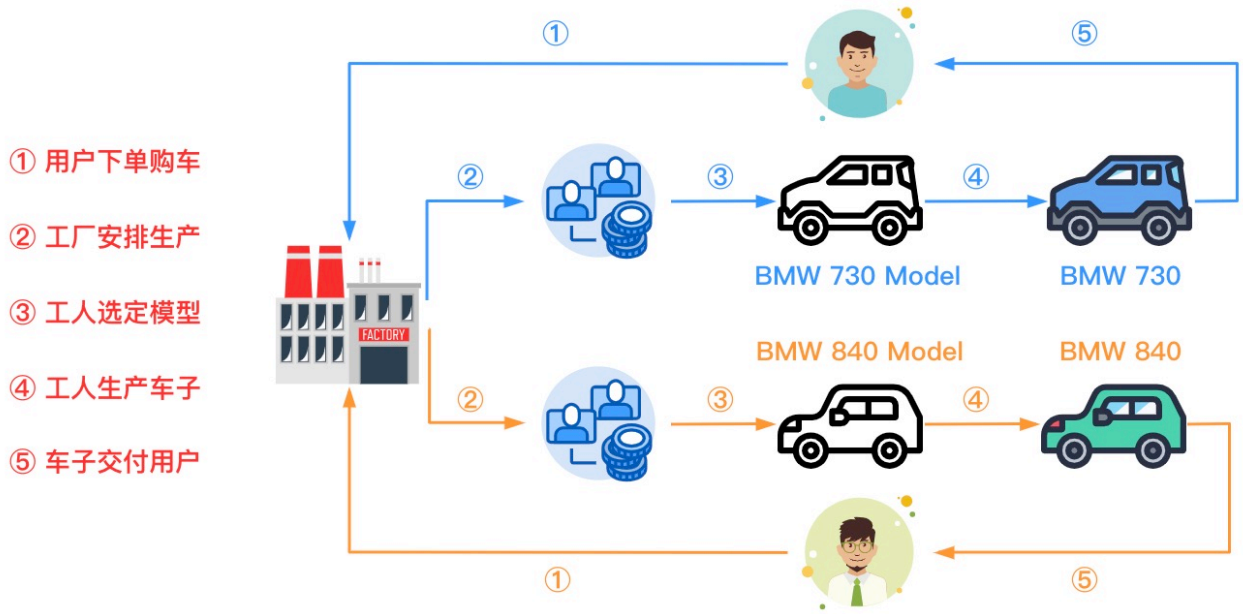
本章阿宝哥将介绍简单工厂模式与工厂方法模式，而抽象工厂模式将在后续的文章中介绍，下面我们先来介绍简单工厂模式。

一、简单工厂模式

1.1 简单工厂模式简介

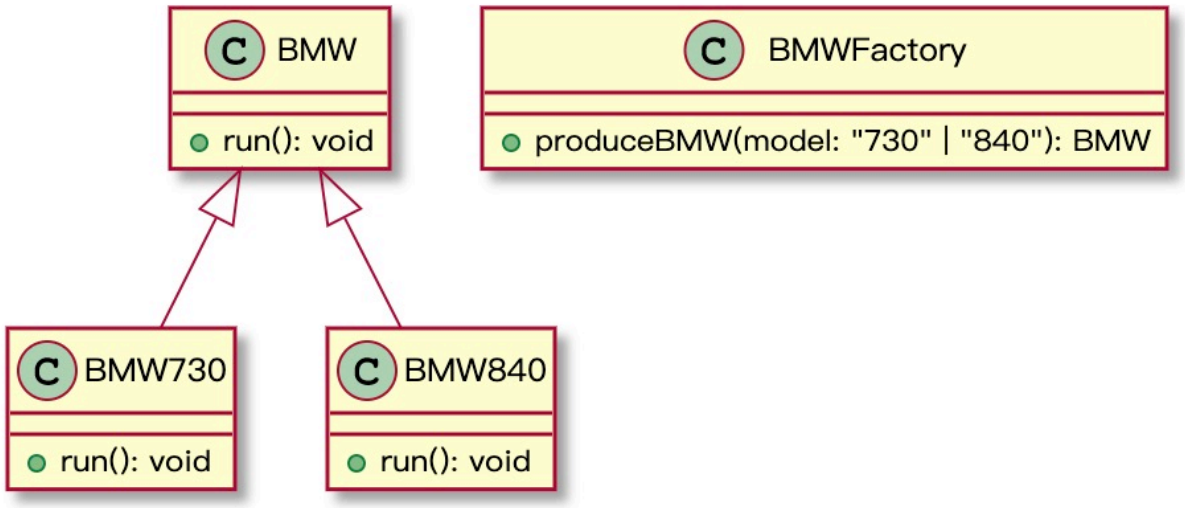
简单工厂模式又叫 **静态方法模式**，因为工厂类中定义了一个静态方法用于创建对象。简单工厂让使用者不用知道具体的参数就可以创建出所需的 ”产品“ 类，即使用者可以直接消费产品而不需要知道产品的具体生产细节。

相信对于刚接触简单工厂模式的小伙伴来说，看到以上的描述可能会觉得有点抽象。这里为了让小伙伴更好地理解简单工厂模式，阿宝哥以用户买车为例，来介绍一下 BMW 工厂如何使用简单工厂模式来生产🚗。



在上图中，阿宝哥模拟了用户购车的流程，pingan 和 qhw 分别向 BMW 工厂订购了 BMW730 和 BMW840 型号的车型，接着工厂按照对应的模型进行生产并在生产完成后交付给用户。接下来，阿宝哥将介绍如何使用简单工厂来描述 BMW 工厂生产指定型号车子的过程。

1.2 简单工厂模式实战



1. 定义 BMW 抽象类

```
abstract class BMW {
  abstract run(): void;
}
```

2. 创建 BMW730 类（BMW 730 Model）

```
class BMW730 extends BMW {
  run(): void {
    console.log("BMW730 发动咯");
  }
}
```

3. 创建 BMW840 类（BMW 840 Model）

```
class BMW840 extends BMW {
  run(): void {
    console.log("BMW840 发动咯");
  }
}
```

4. 创建 BMWFactory 工厂类

```
class BMWFactory {
  public static produceBMW(model: "730" | "840"): BMW {
    if (model === "730") {
      return new BMW730();
    } else {
      return new BMW840();
    }
  }
}
```

5. 生产并发动 BMW730 和 BMW840

```
const bmw730 = BMWFactory.produceBMW("730");
const bmw840 = BMWFactory.produceBMW("840");

bmw730.run();
bmw840.run();
```

以上代码运行后的输出结果为：

```
BMW730 发动咯
BMW840 发动咯
```

通过观察以上的输出结果，我们可以知道我们的 BMWFactory 已经可以正常工作了。在 BMWFactory 类中，阿宝哥定义了一个 `produceBMW()` 方法，该方法会根据传入的模型参数来创建不同型号的车子。

看完简单工厂模式实战的示例，你是不是觉得简单工厂模式还是挺好理解的。那么什么场景下使用简单工厂模式呢？要回答这个问题我们需要来了解一下简单工厂的优缺点。

1.3 简单工厂模式优缺点

1.3.1 优点

- 将创建实例与使用实例的任务分开，使用者不必关心对象是如何创建的，实现了系统的解耦；
- 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可。

1.3.2 缺点

- 由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。
- 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，也有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

了解完简单工厂的优缺点，我们来看一下它的应用场景。

1.4 简单工厂模式应用场景

在满足以下条件下可以考虑使用简单工厂模式：

- 工厂类负责创建的对象比较少：由于创建的对象比较少，不会造成工厂方法中业务逻辑过于复杂。
- 客户端只需知道传入工厂类静态方法的参数，而不需要关心创建对象的细节。

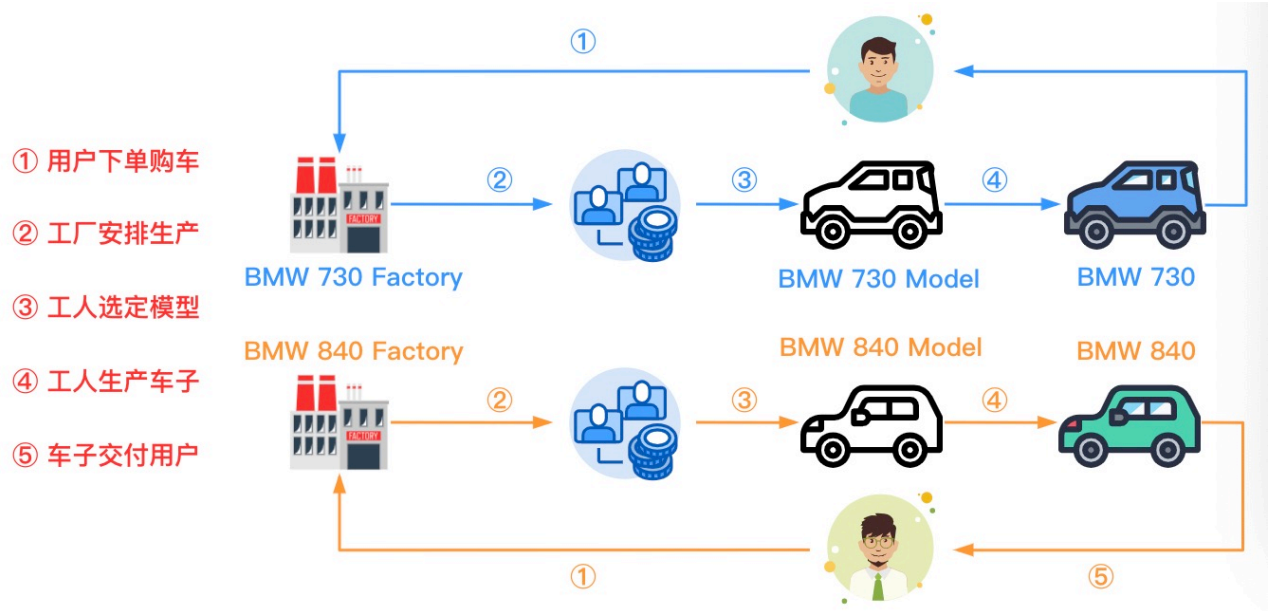
介绍完简单工厂模式，接下来我们来介绍本章的主角 **"工厂方法模式"**。

二、工厂方法模式

2.1 工厂方法简介

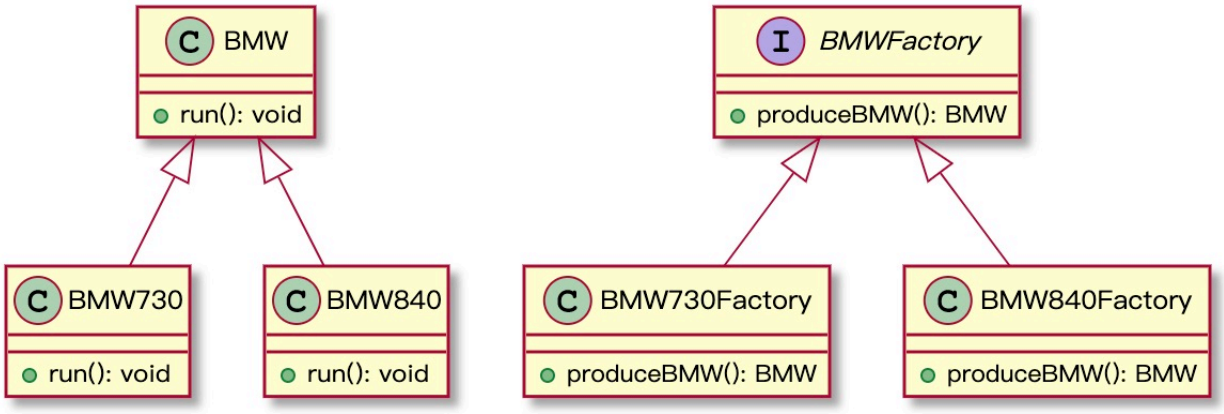
工厂方法模式（Factory Method Pattern）又称为工厂模式，也叫多态工厂（Polymorphic Factory）模式，它属于类创建型模式。

在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象， **这样做的目的是将产品类的实例化操作延迟到工厂子类中完成**，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。



在上图中，阿宝哥模拟了用户购车的流程，pingan 和 qhw 分别向 BMW 730 和 BMW 840 工厂订购了 BMW730 和 BMW840 型号的车型，接着工厂按照对应的模型进行生产并在生产完成后交付给用户。接下来，阿宝哥来介绍如何使用工厂方法来描述 BMW 工厂生产指定型号车子的过程。

2.2 工厂方法实战



1. 定义 BMW 抽象类

```
abstract class BMW {
  abstract run(): void;
}
```

2. 创建 BMW730 类（BMW 730 Model）

```
class BMW730 extends BMW {
  run(): void {
    console.log("BMW730 发动咯");
  }
}
```

3. 创建 BMW840 类（BMW 840 Model）

```
class BMW840 extends BMW {
  run(): void {
    console.log("BMW840 发动咯");
  }
}
```

4. 定义 BMWFactory 接口

```
interface BMWFactory {
  produceBMW(): BMW;
}
```

5. 创建 BMW730Factory 类

```
class BMW730Factory implements BMWFactory {
  produceBMW(): BMW {
    return new BMW730();
  }
}
```

6. 创建 BMW840Factory 类

```
class BMW840Factory implements BMWFactory {
  produceBMW(): BMW {
    return new BMW840();
  }
}
```

7. 生产并发动 BMW730 和 BMW840

```
const bmw730Factory = new BMW730Factory();
const bmw840Factory = new BMW840Factory();

const bmw730 = bmw730Factory.produceBMW();
const bmw840 = bmw840Factory.produceBMW();

bmw730.run();
bmw840.run();
```

通过观察以上的输出结果，我们可以知道我们的 BMW730Factory 和 BMW840Factory 工厂已经可以正常工作了。相比前面的简单工厂模式，工厂方法模式通过创建不同的工厂来生产不同的产品。下面我们来看一下工厂方法有哪些优缺点。

2.3 工厂方法优缺点

2.3.1 优点

- 在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，更加符合 “开闭原则”。而简单工厂模式需要修改工厂类的判断逻辑。
- 符合单一职责的原则，即每个具体工厂类只负责创建对应的产品。而简单工厂模式中的工厂类存在一定的逻辑判断。
- 基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。

2.3.2 缺点

- 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。
- 一个具体工厂只能创建一种具体产品。

最后我们来简单介绍一下工厂方法的应用场景。

2.4 工厂方法应用场景

- 一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
- 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

三、参考资源

- 简单工厂模式（SimpleFactoryPattern）
- design-patterns - simple_factory
- 工厂方法模式（Factory Method）

第五章 发布订阅模式

一、背景

作为一名 Web 开发者，在日常工作中，经常都会遇到消息通信的场景。比如实现组件间通信、实现插件间通信、实现不同的系统间通信。那么针对这些场景，我们应该怎么实现消息通信呢？本章阿宝哥将带大家一起来学习如何优雅的实现消息通信。

好的，接下来我们马上步入正题，这里阿宝哥以一个文章订阅的例子来拉开本章的序幕。小秦与小王是阿宝哥的两个好朋友，他们在阿宝哥的 “全栈修仙之路” 博客中发现了 TS 专题文章，刚好他们近期也打算系统地学习 TS，所以他们就开启了 TS 的学习之旅。

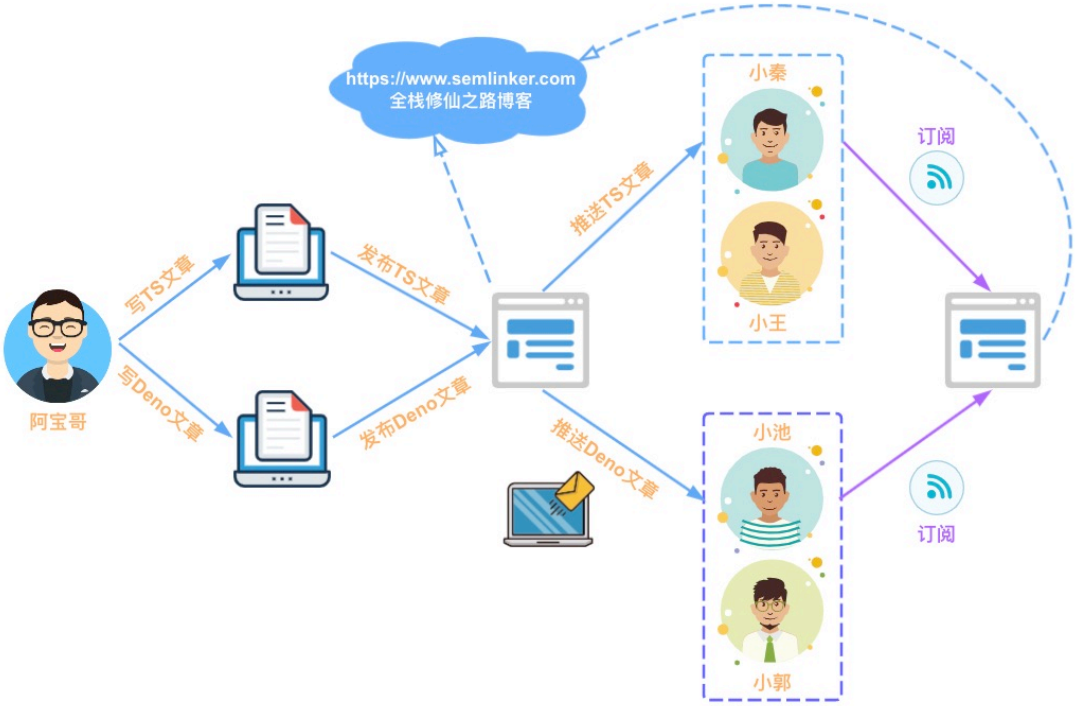
时间就这样过了半个月，小秦和小王都陆续找到了阿宝哥，说 “全栈修仙之路” 博客上的 TS 文章都差不多学完了，他们有空的时候都会到 “全栈修仙之路” 博客上查看是否有新发的 TS 文章。他们觉得这样挺麻烦的，看能不能在阿宝哥发完新的 TS 文章之后，主动通知他们。

好友提的建议，阿宝哥怎能拒绝呢？所以阿宝哥分别跟他们说：“我会给博客加个订阅的功能，功能发布后，你填写一下邮箱地址。以后发布新的 TS 文章，系统会及时给你发邮件”。此时新的流程如下图所示：

在阿宝哥的一顿 “操作” 之后，博客的订阅功能上线了，阿宝哥第一时间通知了小秦与小王，让他们填写各自的邮箱。之后，每当阿宝哥发布新的 TS 文章，他们就会收到新的邮件通知了。

阿宝哥是个技术宅，对新的技术也很感兴趣。在遇到 Deno 之后，阿宝哥燃起了学习 Deno 的热情，同时也开启了新的 Deno 专题。在写了几篇 Deno 专题文章之后，两个读者小池和小郭分别联系到我，说他们看到了阿宝哥的 Deno 文章，想跟阿宝哥一起学习 Deno。

在了解他们的情况之后，阿宝哥突然想到了之前小秦与小王提的建议。因此，又是一顿 “操作” 之后，阿宝哥为了博客增加了专题订阅功能。该功能上线之后，阿宝哥及时联系了小池和小郭，邀请他们订阅 Deno 专题。之后小池和小郭也成为了阿宝哥博客的订阅者。现在的流程变成这样：



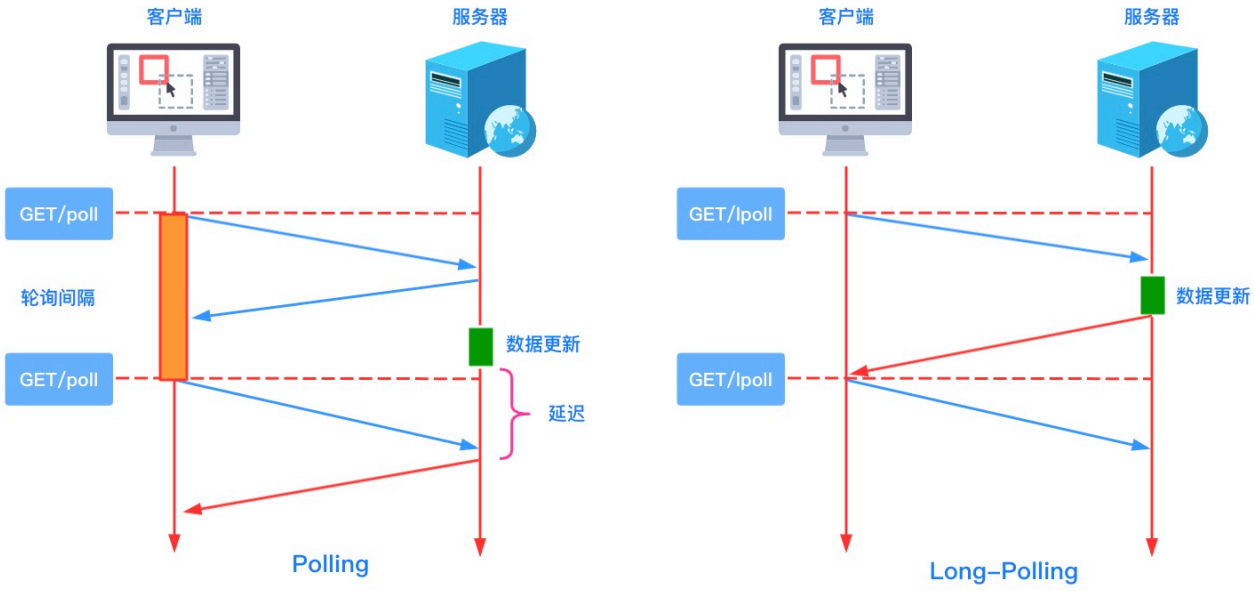
这个例子看起来很简单，但它背后却与一些设计思想和设计模式相关联。因此，接下来阿宝哥将分析以上三个场景与软件开发中一些设计思想和设计模式的关联性。

二、场景与模式

2.1 消息轮询模式

在第一个场景中，小秦和小王为了能查看阿宝哥新发的 TS 文章，他们需要不断地访问 “全栈修仙之路” 博客：

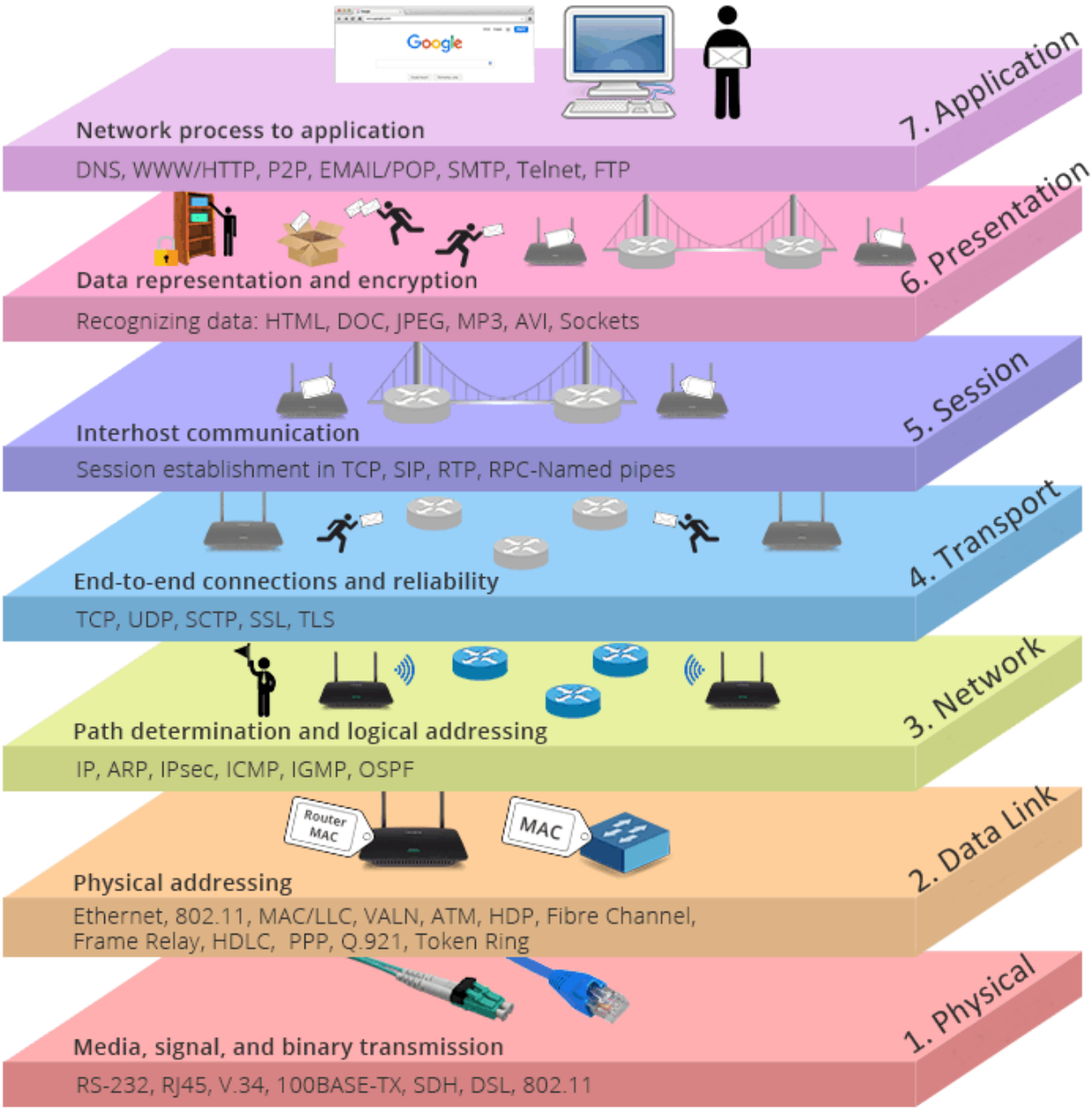
这个场景跟软件开发过程中的轮询模式类似。早期，很多网站为了实现推送技术，所用的技术都是轮询。轮询是指由浏览器每隔一段时间向服务器发出 HTTP 请求，然后服务器返回最新的数据给客户端。常见的轮询方式分为轮询与长轮询，它们的区别如下图所示：



这种传统的模式带来很明显的缺点，**即浏览器需要不断的向服务器发出请求，然而 HTTP 请求与响应可能会包含较长的头部，其中真正有效的数据可能只是很小的一部分，所以这样会消耗很多带宽资源**。为了解决上述问题 HTML5 定义了 WebSocket 协议，能更好的节省服务器资源和带宽，并且能够更实时地进行通讯。

WebSocket 是一种网络传输协议，可在单个 TCP 连接上进行全双工通信，位于 OSI 模型的应用层。WebSocket 协议在 2011 年由 IETF 标准化为 RFC 6455，后由 RFC 7936 补充规范。

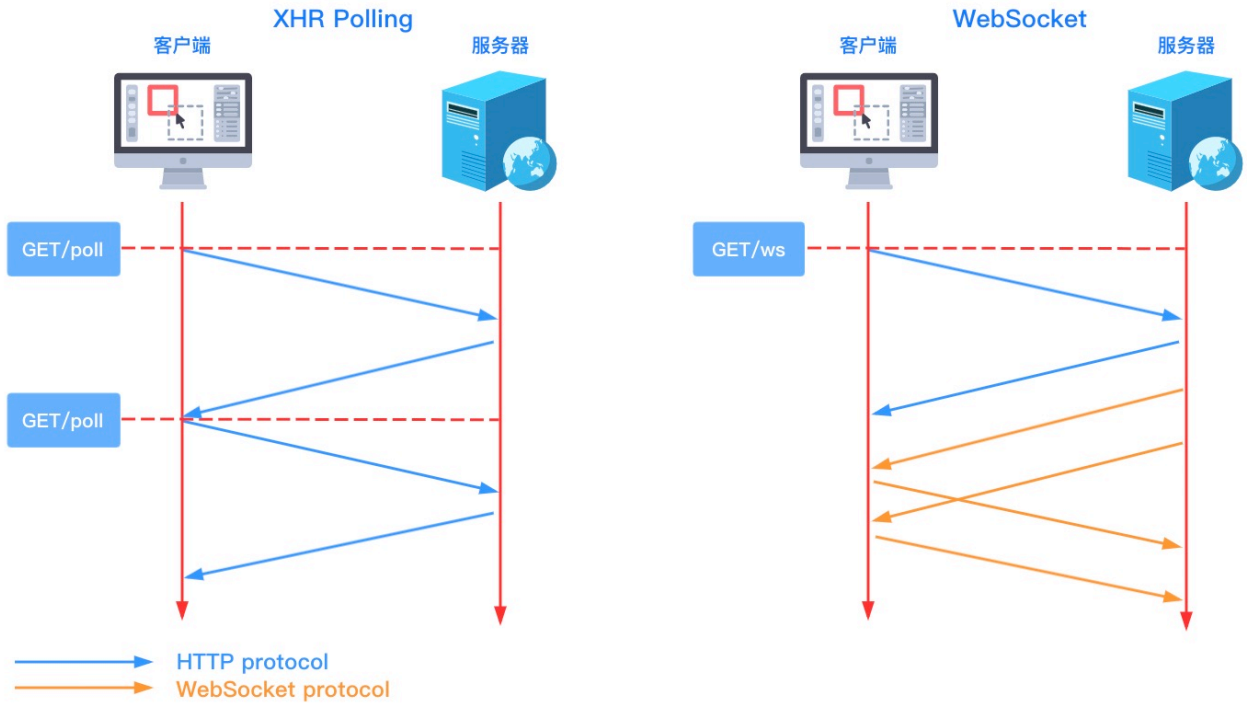
既然已经提到了 OSI（Open System Interconnection Model）模型，这里阿宝哥来分享一张很生动、很形象描述 OSI 模型的示意图：



（图片来源：https://www.networkingsphere.com/2019/07/what-is-osi-model.html）

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。**在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输。**

介绍完轮询和 WebSocket 的相关内容之后，接下来我们来看一下 XHR Polling 与 WebSocket 之间的区别：

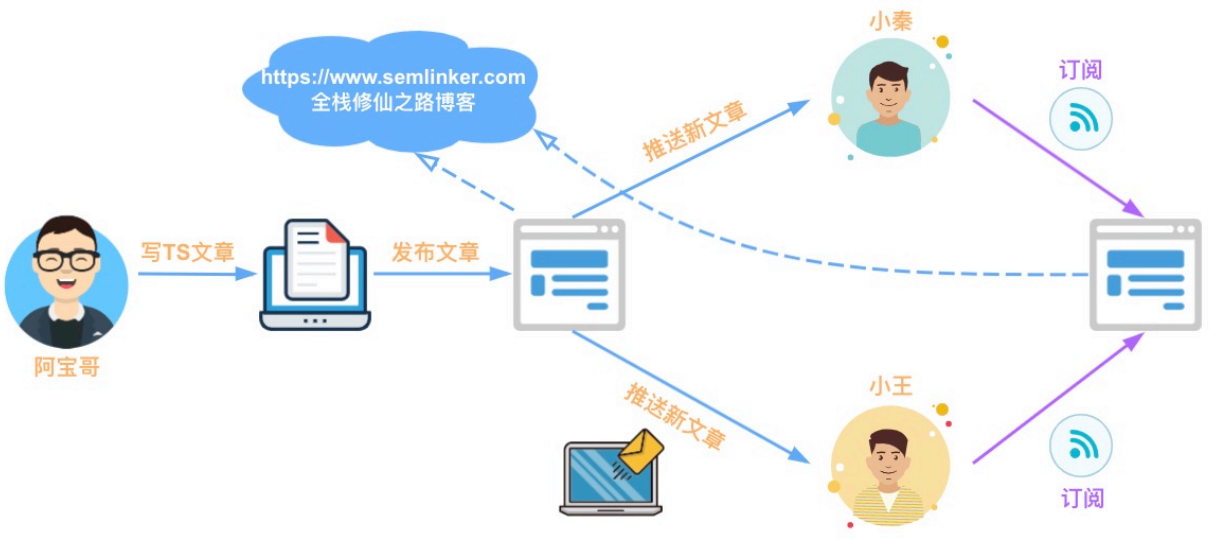


对于 XHR Polling 与 WebSocket 来说，它们分别对应了消息通信的两种模式，即 Pull（拉）模式与 Push（推）模式：

场景一我们就介绍到这里，对轮询和 WebSocket 感兴趣的小伙伴可以阅读阿宝哥写的 **“**[你不知道的 WebSocket](#)**”** 这一篇文章。下面我们来继续分析第二个场景。

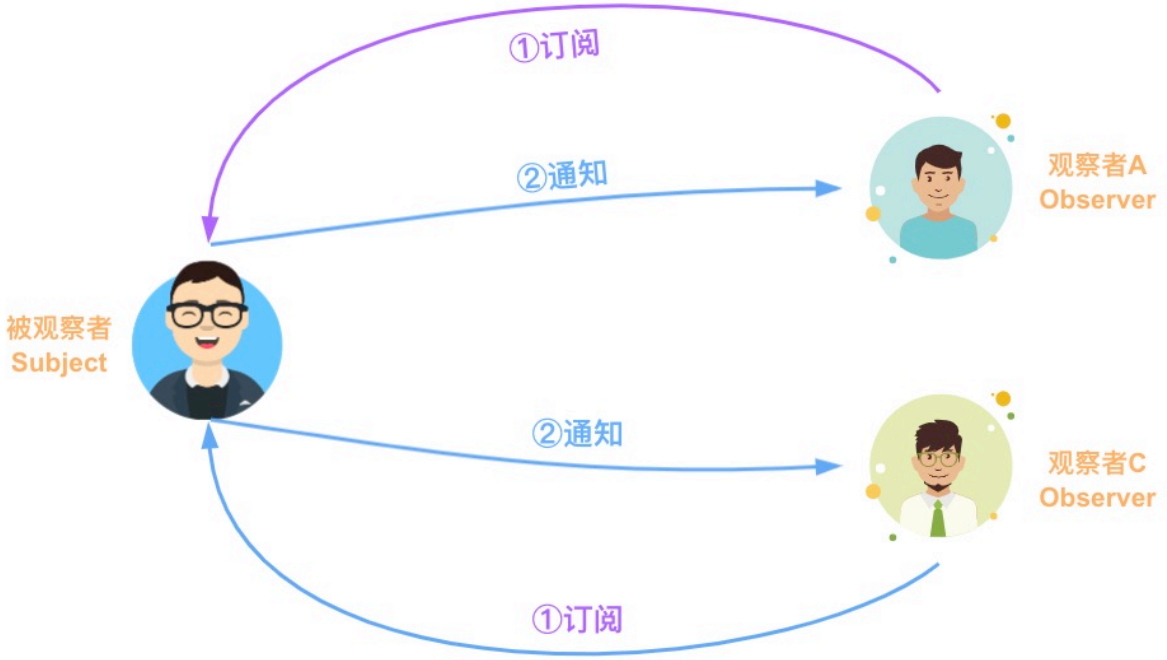
2.2 观察者模式

在第二个场景中，为了让小秦和小王能及时收到阿宝哥新发布的 TS 文章，阿宝哥给博客增加了订阅功能。这里假设阿宝哥博客一开始只发布 TS 专题的文章。



针对这个场景，我们可以考虑使用设计模式中观察者模式来实现上述功能。**观察者模式，它定义了一种一对多的关系，让多个观察者对象同时监听某一个主题对象，这个主题对象的状态发生变化时就会通知所有的观察者对象，使得它们能够自动更新自己。**

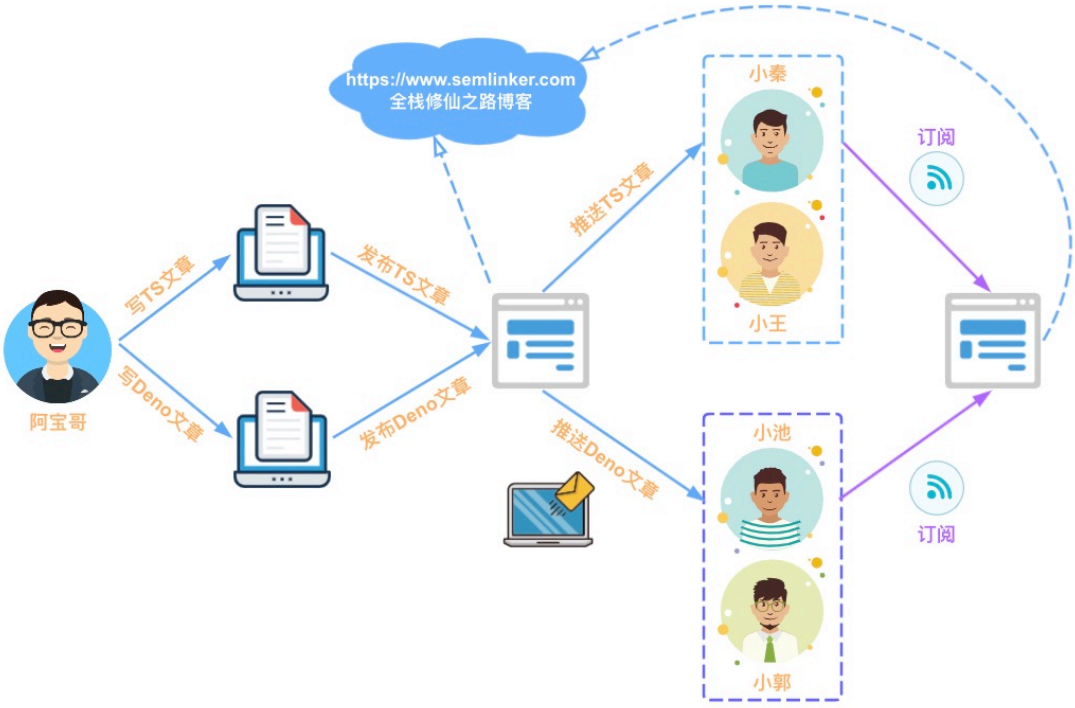
在观察者模式中有两个主要角色：Subject（主题）和 Observer（观察者）。



在第二个场景中，Subject（主题）就是阿宝哥的 TS 专题文章，而观察者就是小秦和小王。由于观察者模式支持简单的广播通信，当消息更新时，会自动通知所有的观察者。因此对于第二个场景，我们可以考虑使用观察者设计模式来实现上述的功能。接下来，我们来继续分析第三个场景。

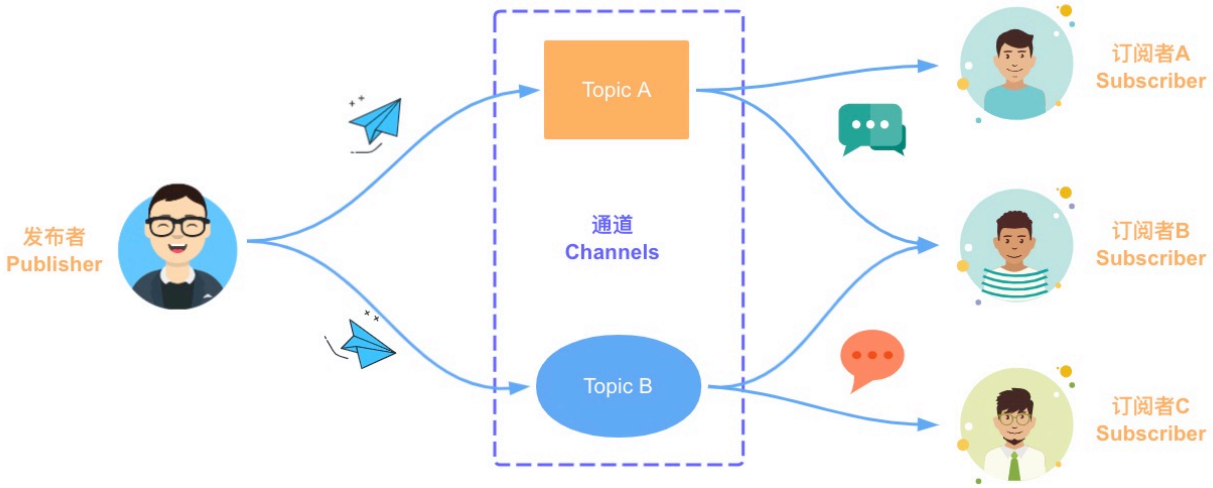
2.3 发布订阅模式

在第三个场景中，为了让小池和小郭能及时收到阿宝哥新发布的 Deno 文章，阿宝哥给博客增加了专题订阅功能。即支持为阿宝哥博客的订阅者分别推送新发布的 TS 或 Deno 文章。



针对这个场景，我们可以考虑使用发布订阅模式来实现上述功能。在软件架构中，发布 — 订阅是一种消息范式，消息的发送者（称为发布者）不会将消息直接发送给特定的接收者（称为订阅者）。而是将发布的消息分为不同的类别，然后分别发送给不同的订阅者。同样的，订阅者可以表达对一个或多个类别的兴趣，只接收感兴趣的消息，无需了解哪些发布者存在。

在发布订阅模式中有三个主要角色：Publisher（发布者）、Channels（通道）和 Subscriber（订阅者）。



在第三个场景中，Publisher（发布者）是阿宝哥，Channels（通道）中 Topic A 和 Topic B 分别对应于 TS 专题和 Deno 专题，而 Subscriber（订阅者）就是小秦、小王、小池和小郭。好的，了解完发布订阅模式，下面我们来介绍一下它的一些应用场景。

三、发布订阅模式的应用

3.1 前端框架中模块/页面间消息通信

在一些主流的前端框架中，内部也会提供用于模块间或页面间通信的组件。比如在 Vue 框架中，我们可以通过 `new Vue()` 来创建 EventBus 组件。而在 Ionic 3 中我们可以使用 `ionic-angular` 模块中的 Events 组件来实现模块间或页面间的消息通信。下面我们来分别介绍在 Vue 和 Ionic 中如何实现模块/页面间的消息通信。

3.1.1 Vue 使用 EventBus 进行消息通信

在 Vue 中我们可以通过创建 EventBus 来实现组件间或模块间的消息通信，使用方式很简单。在下图中包含两个 Vue 组件：Greet 和 Alert 组件。Alert 组件用于显示消息，而 Greet 组件中包含一个按钮，即下图中 "显示问候消息" 的按钮。当用户点击按钮时，Greet 组件会通过 EventBus 把消息传递给 Alert 组件，该组件接收到消息后，会调用 `alert` 方法把收到的消息显示出来。

以上示例对应的代码如下：

main.js

```
Vue.prototype.$bus = new Vue();
```

Alert.vue

```
<script>
export default {
  name: "alert",
  created() {
    // 监听alert:message事件
    this.$bus.$on("alert:message", msg => {
      this.showMessage(msg);
    });
  },
```

```
  methods: {
    showMessage(msg) {
      alert(msg);
    },
  },
  beforeDestroy: function() {
    // 组件销毁时，移除alert:message事件监听
    this.$bus.$off("alert:message");
  }
}
</script>
```

Greet.vue

```
<template>
  <div>
    <button @click="greet(message)">显示问候信息</button>
  </div>
</template>

<script>
export default {
  name: "Greet",
  data() {
    return {
      message: "大家好，我是阿宝哥",
    };
  },
  methods: {
    greet(msg) {
      this.$bus.$emit("alert:message", msg);
    }
  }
};
</script>
```

3.1.2 Ionic 使用 Events 组件进行消息通信

在 Ionic 3 项目中，要实现页面间消息通信很简单。我们只要通过构造注入的方式注入 `ionic-angular` 模块中提供的 Events 组件即可。具体的使用示例如下所示：

```
import { Events } from 'ionic-angular';

// first page (publish an event when a user is created)
constructor(public events: Events) {}
createUser(user) {
  console.log('User created!')
  this.events.publish('user:created', user, Date.now());
```

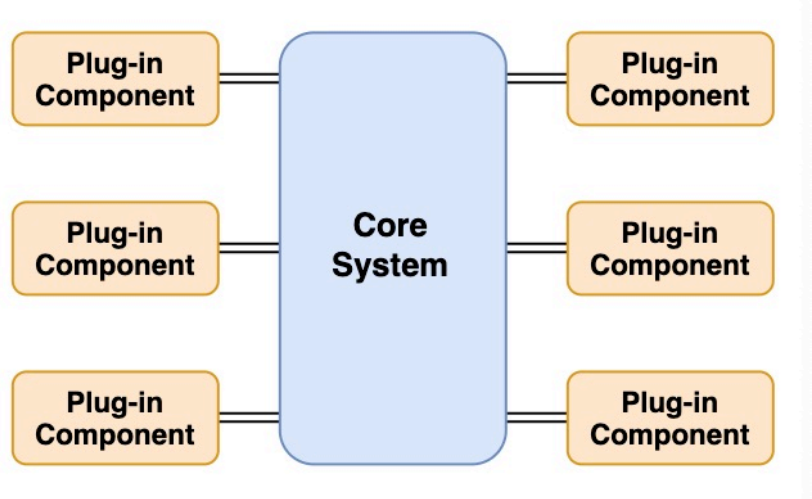
```
}


// second page (listen for the user created event after function is called)
constructor(public events: Events) {
  events.subscribe('user:created', (user, time) => {
    // user and time are the same arguments passed in `events.publish(user,
time)`
    console.log('Welcome', user, 'at', time);
  });
}
```

介绍完发布订阅模式在 Vue 和 Ionic 框架中的应用之后，接下来阿宝哥将介绍该模式在微内核架构中是如何实现插件通信的。

3.2 微内核架构中插件通信

微内核架构（Microkernel Architecture），有时也被称为插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于产品的应用。微内核架构模式允许你将其他应用程序功能作为插件添加到核心应用程序，从而提供可扩展性以及功能分离和隔离。

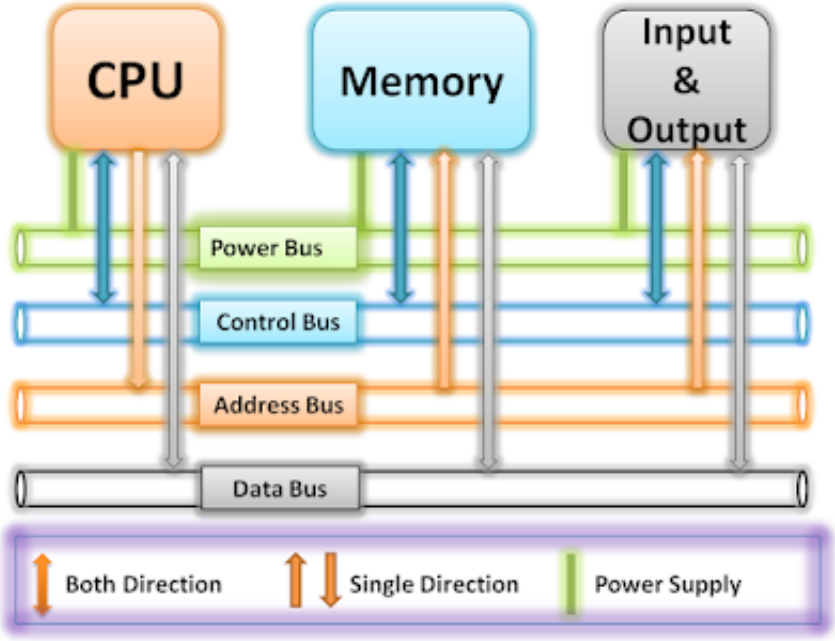
微内核架构模式包括两种类型的架构组件：核心系统（Core System）和插件模块（Plug-in modules）。**应用逻辑被分割为独立的插件模块和核心系统，提供了可扩展性、灵活性、功能隔离和自定义处理逻辑的特性。**



对于微内核的核心系统设计来说，它涉及三个关键技术：**插件管理、插件连接和插件通信**，这里我们重点来分析一下插件通信。

插件通信是指插件间的通信。虽然设计的时候插件间是完全解耦的，但实际业务运行过程中，必然会出现某个业务流程需要多个插件协作，这就要求两个插件间进行通信；**由于插件之间没有直接联系，通信必须通过核心系统，因此核心系统需要提供插件通信机制。**

这种情况和计算机类似，计算机的 CPU、硬盘、内存、网卡是独立设计的配置，但计算机运行过程中，CPU 和内存、内存和硬盘肯定是有通信的，计算机通过主板上的总线提供了这些组件之间的通信功能。



下面阿宝哥将以基于微内核架构设计的西瓜播放器为例，介绍它的内部是如何提供插件通信机制。在西瓜播放器内部，定义了一个 `Player` 类来创建播放器实例：

```
let player = new Player({
  id: 'mse',
  url: '//abc.com/**/*.mp4'
});
```

`Player` 类继承于 `Proxy` 类，而在 `Proxy` 类内部会通过构造继承的方式继承 `EventEmitter` 事件派发器：

```
import EventEmitter from 'event-emitter'

class Proxy {
  constructor (options) {
    this._hasStart = false;
    // 省略大部分代码
    EventEmitter(this);
  }
}
```

所以我们创建的西瓜播放器也是一个事件派发器，利用它就可以实现插件的通信。为了让大家能够更好地理解具体的通信流程，我们以内置的 poster 插件为例，来看一下它内部如何使用事件派发器。

poster 插件用于在播放器播放音视频前显示海报图，该插件的使用方式如下：

```
new Player({
  el:document.querySelector('#mse'),
  url: 'video_url',
  poster: '//abc.com/**/*.png' // 默认值""
});
```

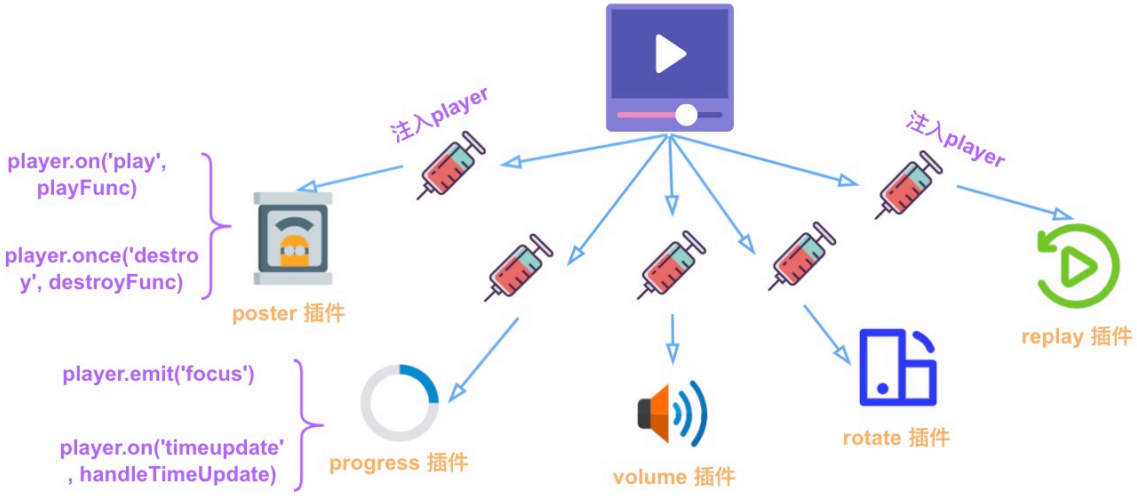
poster 插件的对应源码如下：

```
import Player from '../player'

let poster = function () {
  let player = this;
  let util = Player.util
  let poster = util.createDom('xg-poster', '', {}, 'xgplayer-poster');
  let root = player.root
  if (player.config.poster) {
    poster.style.backgroundImage = `url(${player.config.poster})`
    root.appendChild(poster)
  }

  // 监听播放事件，播放时隐藏封面图
  function playFunc () {
    poster.style.display = 'none'
  }
  player.on('play', playFunc)

  // 监听销毁事件，执行清理操作
  function destroyFunc () {
    player.off('play', playFunc)
    player.off('destroy', destroyFunc)
  }
  player.once('destroy', destroyFunc)
}

Player.install('poster', poster)
```

(https://github.com/bytedance/xgplayer/blob/master/packages/xgplayer/src/control/poster.js)

通过观察源码可知，在注册 poster 插件时，会把播放器实例注入到插件中。之后，在插件内部会使用 player 这个事件派发器来监听播放器的 `play` 和 `destroy` 事件。当 poster 插件监听到播放器的 `play` 事件之后，就会隐藏海报图。而当 poster 插件监听到播放器的 `destroy` 事件时，就会执行清理操作，比如移除已绑定的事件。

看到这里我们就已经很清楚了，西瓜播放器内部使用 `EventEmitter` 来提供插件通信机制，每个插件都会注入 player 这个全局的事件派发器，通过它就可以轻松地实现插件间通信了。



提到 `EventEmitter`，相信很多小伙伴对它并不会陌生。在 Node.js 中有一个名为 `events` 的内置模块，通过它我们可以方便地实现一个自定义的事件派发器，比如：

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('event', () => {
  console.log('大家好，我是阿宝哥!');
});

myEmitter.emit('event');
```

3.3 基于 Redis 实现不同系统间通信

在前面我们介绍了发布订阅模式在单个系统中的应用。其实，在日常开发过程中，我们也会遇到不同系统间通信的问题。接下来阿宝哥将介绍如何利用 Redis 提供的发布与订阅功能实现系统间的通信，不过在介绍具体应用前，我们得先熟悉一下 Redis 提供的发布与订阅功能。

3.3.1 Redis 发布与订阅功能

Redis 订阅功能

通过 Redis 的 subscribe 命令，我们可以订阅感兴趣的通道，其语法为：`SUBSCRIBE channel [channel …]`。

```
➜  ~ redis-cli
127.0.0.1:6379> subscribe deno ts
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "deno"
3) (integer) 1
1) "subscribe"
2) "ts"
3) (integer) 2
```

在上述命令中，我们通过 `subscribe` 命令订阅了 deno 和 ts 两个通道。接下来我们新开一个命令行窗口，来测试 Redis 的发布功能。

Redis 发布功能

通过 Redis 的 publish 命令，我们可以为指定的通道发布消息，其语法为：`PUBLISH channel message`。

```
➜  ~ redis-cli
127.0.0.1:6379> publish ts "pub/sub design mode"
(integer) 1
```

当成功发布消息之后，订阅该通道的客户端就会收到消息，对应的控制台就会输出如下信息：

```
1) "message"
2) "ts"
3) "pub/sub design mode"
```

了解完 Redis 的发布与订阅功能，接下来阿宝哥将介绍如何利用 Redis 提供的发布与订阅功能实现不同系统间的通信。

3.3.2 实现不同系统间的通信

这里我们使用 Node.js 的 Express 框架和 redis 模块来快速搭建不同的 Web 应用，首先创建一个新的 Web 项目并安装一下相关的依赖：

```
$ npm init --yes
$ npm install express redis
```

接着创建一个发布者应用：

publisher.js

```
const redis = require("redis");
const express = require("express");

const publisher = redis.createClient();
```

```
const app = express();

app.get("/", (req, res) => {
  const article = {
    id: "666",
    name: "TypeScript实战之发布订阅模式",
  };

  publisher.publish("ts", JSON.stringify(article));
  res.send("阿宝哥写了一篇TS文章");
});

app.listen(3005, () => {
  console.log(`server is listening on PORT 3005`);
});
```

然后分别创建两个订阅者应用：

subscriber-1.js

```
const redis = require("redis");
const express = require("express");

const subscriber = redis.createClient();

const app = express();

subscriber.on("message", (channel, message) => {
  console.log("小王收到了阿宝哥的TS文章： " + message);
});

subscriber.subscribe("ts");

app.get("/", (req, res) => {
  res.send("我是阿宝哥的粉丝，小王");
});

app.listen(3006, () => {
  console.log("server is listening to port 3006");
});
```

subscriber-2.js

```
const redis = require("redis");
const express = require("express");

const subscriber = redis.createClient();
```

```
// https://dev.to/ganeshmani/implementing-redis-pub-sub-in-node-js-application-12he
const app = express();

subscriber.on("message", (channel, message) => {
  console.log("小秦收到了阿宝哥的TS文章: " + message);
});

subscriber.subscribe("ts");

app.get("/", (req, res) => {
  res.send("我是阿宝哥的粉丝，小秦");
});

app.listen(3007, () => {
  console.log("server is listening to port 3007");
});
```

接着分别启动上面的三个应用，当所有应用都成功启动之后，在浏览器中访问 `http://localhost:3005/` 地址，此时上面的两个订阅者应用对应的终端会分别输出以下信息：

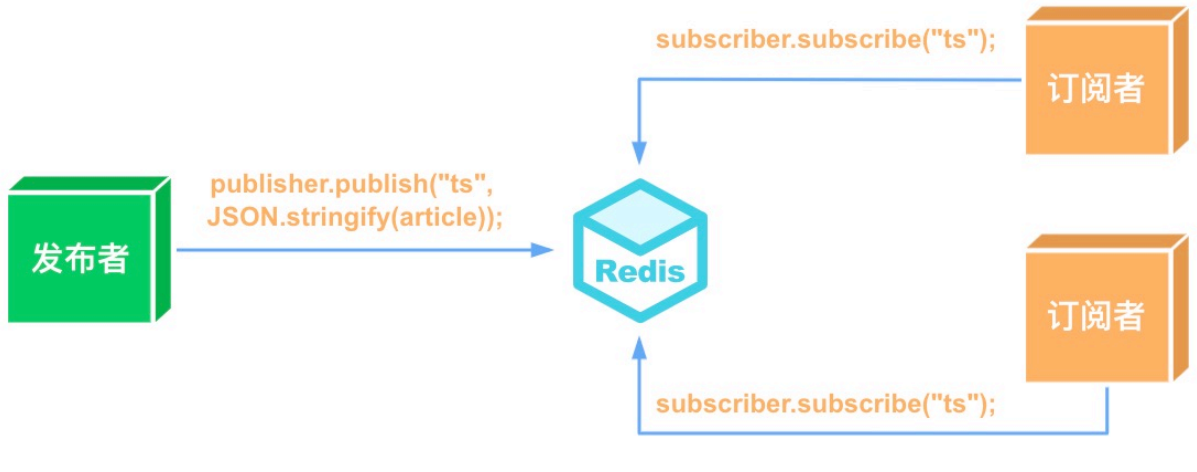
subscriber-1.js

```
server is listening to port 3006
小王收到了阿宝哥的TS文章: {"id":"666","name":"TypeScript实战之发布订阅模式"}
```

subscriber-2.js

```
server is listening to port 3007
小秦收到了阿宝哥的TS文章: {"id":"666","name":"TypeScript实战之发布订阅模式"}
```

以上示例对应的通信流程如下图所示：



到这里发布订阅模式的应用场景，已经介绍完了。最后，阿宝哥来介绍一下如何使用 TS 实现一个支持发布与订阅功能的 EventEmitter 组件。

四、发布订阅模式实战

4.1 定义 EventEmitter 类

```
type EventHandler = (...args: any[]) => any;

class EventEmitter {
  private c = new Map<string, EventHandler[]>();

  // 订阅指定的主题
  subscribe(topic: string, ...handlers: EventHandler[]) {
    let topics = this.c.get(topic);
    if (!topics) {
      this.c.set(topic, topics = []);
    }
    topics.push(...handlers);
  }

  // 取消订阅指定的主题
  unsubscribe(topic: string, handler?: EventHandler): boolean {
    if (!handler) {
      return this.c.delete(topic);
    }

    const topics = this.c.get(topic);
    if (!topics) {
      return false;
    }

    const index = topics.indexOf(handler);

    if (index < 0) {
      return false;
    }
    topics.splice(index, 1);
    if (topics.length === 0) {
      this.c.delete(topic);
    }
    return true;
  }

  // 为指定的主题发布消息
  publish(topic: string, ...args: any[]): any[] | null {
    const topics = this.c.get(topic);
    if (!topics) {
      return null;
    }
    return topics.map(handler => {
```

```
      try {
        return handler(...args);
      } catch (e) {
        console.error(e);
        return null;
      }
    });
  }
}
```

4.2 使用示例

```
const eventEmitter = new EventEmitter();
eventEmitter.subscribe("ts", (msg) => console.log(`收到订阅的消息：${msg}`) );

eventEmitter.publish("ts", "TypeScript发布订阅模式");
eventEmitter.unsubscribe("ts");
eventEmitter.publish("ts", "TypeScript发布订阅模式");
```

以上代码成功运行之后，控制台会输出以下信息：

```
收到订阅的消息：TypeScript发布订阅模式
```

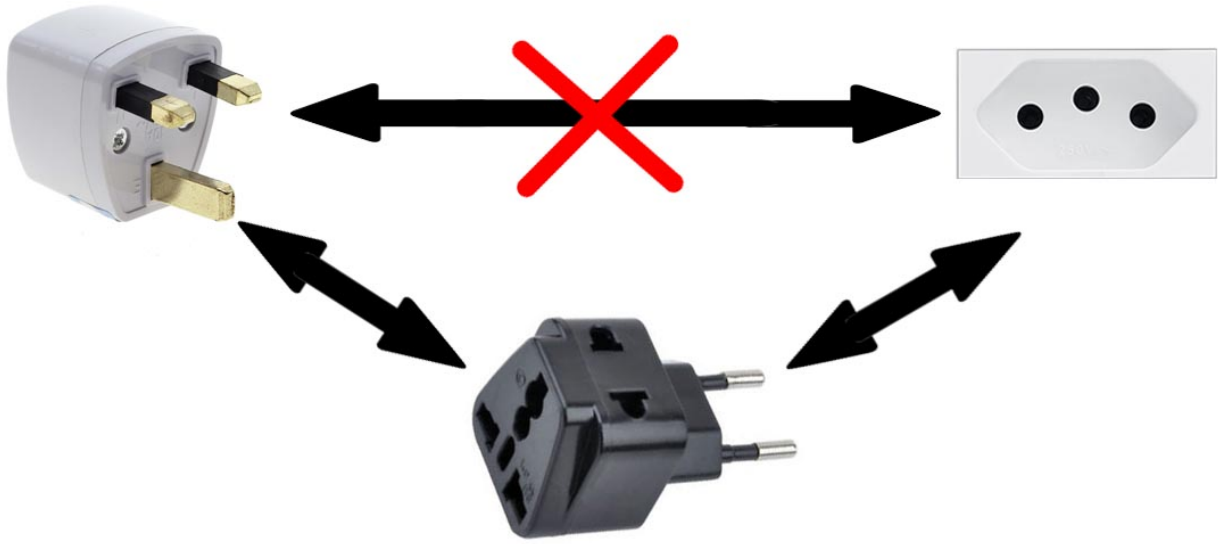
五、参考资源

- 维基百科 - 发布/订阅
- Ionic 3 - Events
- implementing-redis-pub-sub-in-node-js-application

第六章 适配器模式

一、简介

在实际生活中，也存在适配器的使用场景，比如：港式插头转换器、电源适配器和 USB 转接口。**而在软件工程中，适配器模式的作用是解决两个软件实体间的接口不兼容的问题。**使用适配器模式之后，原本由于接口不兼容而不能工作的两个软件实体就可以一起工作。



(图片来源 - https://meneguite.com/)

二、优缺点

优点

- 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。
- 增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
- 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，符合开闭原则。

缺点

- 过多地使用适配器，会让系统非常零乱，不易整体进行把握。

三、应用场景

- 系统需要使用现有的类，而这些类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

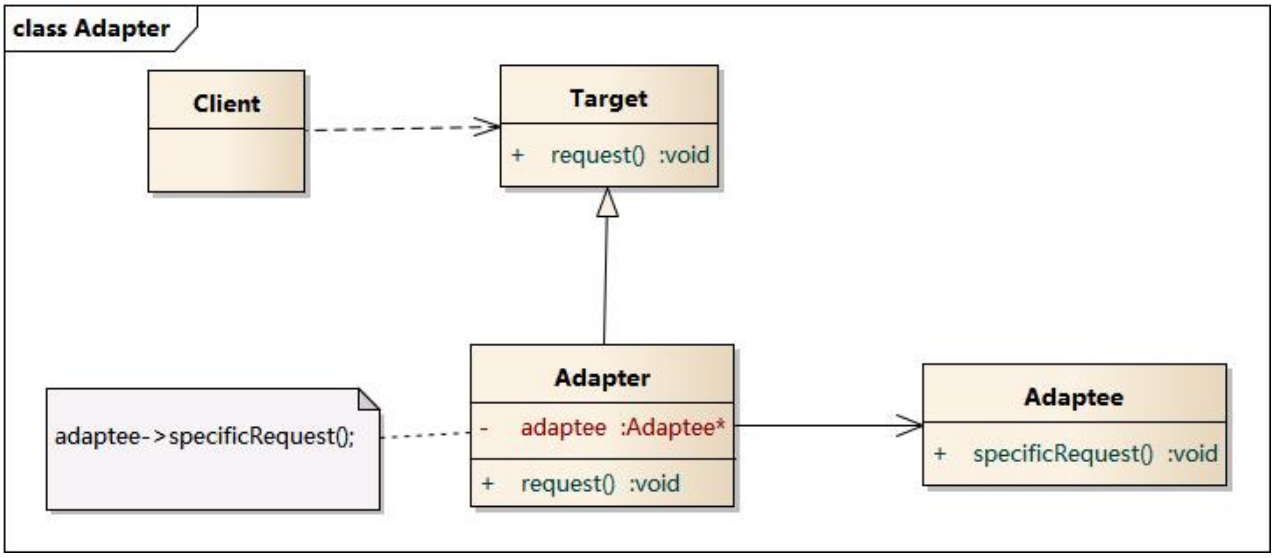
四、模式结构

适配器模式包含以下角色：

- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类
- Client：客户类

适配器模式有对象适配器和类适配器两种实现，这里我们主要介绍对象适配器。

对象适配器：



五、实战

具体实现

定义 Target 接口

```
interface Target {
    request(): void;
}
```

创建 Adaptee（适配者） 类

```
class Adaptee {
    public specificRequest(): void {
        console.log("specificRequest of Adaptee is being called");
    }
}
```

创建 Adapter（适配器）类

```
class Adapter implements Target {
    public request(): void {
        console.log("Adapter's request method is being called");
        var adaptee: Adaptee = new Adaptee();
        adaptee.specificRequest();
    }
}
```

使用示例

```
function show(): void {
    const adapter: Adapter = new Adapter();
    adapter.request();
}
```

为了更好地理解适配器模式的作用，我们来举一个实际的应用示例。假设你现在拥有一个日志系统，该日志系统会将应用程序生成的所有信息保存到本地文件，具体如下：

```
interface Logger {
  info(message: string): Promise<void>;
}

class FileLogger implements Logger {
  public async info(message: string): Promise<void> {
    console.info(message);
    console.info('This Message was saved with FileLogger');
  }
}
```

基于上述的 FileLogger 类，我们就可以在 NotificationService 通知服务中使用它：

```
class NotificationService {
  protected logger: Logger;

  constructor (logger: Logger) {
    this.logger = logger;
  }

  public async send(message: string): Promise<void> {
    await this.logger.info(`Notification sended: ${message}`);
  }
}

(async () => {
  const fileLogger = new FileLogger();
  const notificationService = new NotificationService(fileLogger);
  await notificationService.send('Hello Semlinker, To File');
```

```
})();
```

以上代码成功运行后会输出以下结果：

```
Notification sended: Hello Semlinker
This Message was saved with FileLogger
```

但是现在我们需要使用一种新的方式来保存日志，因为随着应用的增长，我们需要将日志保存到云服务器上，而不再需要保存到磁盘中。因此我们需要使用另一种实现，比如：

```
interface CloudLogger {
  sendToServer(message: string, type: string): Promise<void>;
}

class AliLogger implements CloudLogger {
  public async sendToServer(message: string, type: string): Promise<void> {
    console.info(message);
    console.info('This Message was saved with AliLogger');
  }
}
```

但这时对于我们来说，要使用这个新类，我们就可能需要重构旧的代码以使用新的日志存储方式。为了避免重构代码，我们可以考虑使用适配器来解决这个问题。

```
class CloudLoggerAdapter implements Logger {
  protected cloudLogger: CloudLogger;

  constructor (cloudLogger: CloudLogger) {
    this.cloudLogger = cloudLogger;
  }

  public async info(message: string): Promise<void> {
    await this.cloudLogger.sendToServer(message, 'info');
  }
}
```

在定义好 CloudLoggerAdapter 适配器之后，我们就可以这样使用：

```
(async () => {
  const aliLogger = new AliLogger();
  const cloudLoggerAdapter = new CloudLoggerAdapter(aliLogger);
  const notificationService = new NotificationService(cloudLoggerAdapter);
  await notificationService.send('Hello Kakuqo, To Cloud');
})();
```

以上代码成功运行后会输出以下结果：

```
Notification sended: Hello Kakuqo, To Cloud
This Message was saved with AliLogger
```

如你所见，适配器模式是一个非常有用的模式，对于任何开发人员来说，理解这种模式都是至关重要的。

日志系统适配器完整示例

接口定义

```
interface Logger {
  info(message: string): Promise<void>;
}

interface CloudLogger {
  sendToServer(message: string, type: string): Promise<void>;
}
```

日志实现类

```
class AliLogger implements CloudLogger {
  public async sendToServer(message: string, type: string): Promise<void> {
    console.info(message);
    console.info('This Message was saved with AliLogger');
  }
}
```

适配器

```
class CloudLoggerAdapter implements Logger {
  protected cloudLogger: CloudLogger;

  constructor (cloudLogger: CloudLogger) {
    this.cloudLogger = cloudLogger;
  }

  public async info(message: string): Promise<void> {
    await this.cloudLogger.sendToServer(message, 'info');
  }
}
```

通知服务类

```
class NotificationService {
  protected logger: Logger;

  constructor (logger: Logger) {
    this.logger = logger;
  }

  public async send(message: string): Promise<void> {
    await this.logger.info(`Notification sended: ${message}`);
  }
}
```

使用示例

```
(async () => {
  const aliLogger = new AliLogger();
  const cloudLoggerAdapter = new CloudLoggerAdapter(aliLogger);
  const notificationService = new NotificationService(cloudLoggerAdapter);
  await notificationService.send('Hello Kakuqo, To Cloud');
})();
```

六、参考资源

- design-patterns-com-typescript-adapter

第七章 享元模式

一、简介

享元模式就是运行共享技术有效地支持大量细粒度的对象，避免大量拥有相同内容的小类的开销（如耗费内存），使大家共享一个类。在享元模式中有两个重要的概念，即内部状态和外部状态：

- 内部状态：在享元对象内部不随外界环境改变而改变的共享部分。
- 外部状态：随着环境的改变而改变，不能够共享的状态就是外部状态。

由于享元模式区分了内部状态和外部状态，所以我们可以通过设置不同的外部状态使得相同的对象可以具备一些不同的特性，而内部状态设置为相同部分。

二、优缺点

优点

- 享元模式的优点在于它能够极大的减少系统中对象的个数。
- 享元模式由于使用了外部状态，外部状态相对独立，不会影响到内部状态，所以享元模式使得享元对象能够在不同的环境被共享。

缺点

- 由于享元模式需要区分外部状态和内部状态，使得应用程序在某种程度上来说更加复杂化了。
- 为了使对象可以共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。

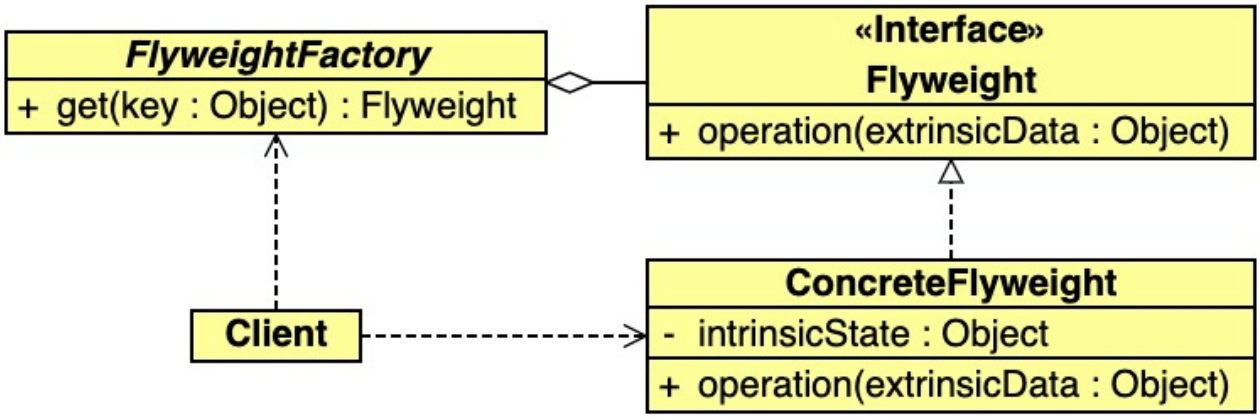
三、应用场景

- 一个程序中使用了大量的相似对象。
- 由于使用了大量对象，造成很大的内存开销。
- 对象的大多数状态都可以变为外部状态。
- 剥离出对象的外部状态之后，可以用相对较少的共享对象取代大量对象。

四、模式结构

享元模式包含以下角色：

- Client：调用 FlyweightFactory 获取享元对象。
- FlyweightFactory：
    - 创建和管理享元对象；
    - 当请求某个享元对象不存在时，它会创建一个新的享元对象；
    - 新创建的享元对象会被存储起来，用于下次请求。
- Flyweight：维护要在应用程序之间共享的固有数据。



五、实战

苹果公司批量生产 iPhone11，iPhone11 的大部分属性比如型号、屏幕都是一样，少部分属性比如内存有分 128、256G 等。未使用享元模式前，我们写如下代码：

```
class Iphone11 {
  constructor(model: string, screen: number, memory: number, sn: number) { }
}

const phones = [];
for (let i = 0; i < 10000; i++) {
  let memory = i % 2 == 0 ? 128 : 256;
  phones.push(new Iphone11("iPhone11", 6.1, memory, i));
}
```

在以上代码中，我们创建了一万个 iPhone11，每个 iPhone11 都独立占有一个内存空间。**但是我们仔细观察可以看到，大部分 iPhone11 都是类似的，只是内存和序列号不一样，如果是一个对性能要求比较高的程序，我们就要考虑去优化它。**

当存在大量相似对象的程序，我们就可以考虑用享元模式去优化它，我们分析出大部分的 iPhone11 的型号、屏幕、内存都是一样的，那么这部分数据就可以共用，这就是享元模式中的内在数据，因此定义 iPhone11 对应的享元类如下：

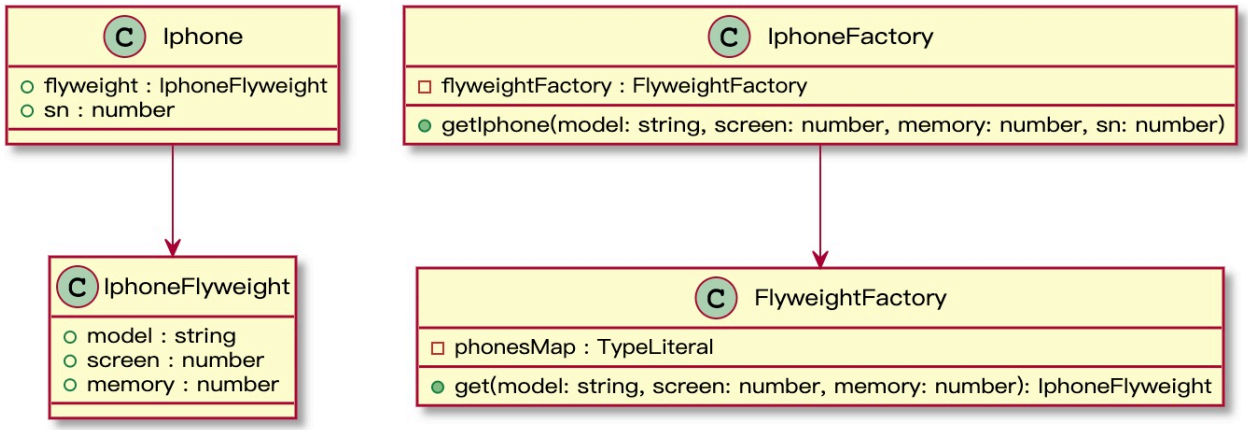
```
class IphoneFlyweight {
  constructor(model: string, screen: number, memory: number) {}
}
```

我们定义了 IphoneFlyweight 享元类，其中包含型号、屏幕、内存三个数据。我们还需要一个享元工厂来维护这些数据：

```
class FlyweightFactory {
    private phonesMap: { [s: string]: IphoneFlyweight } = {};

    public get(model: string, screen: number, memory: number): IphoneFlyweight
{
      const key = model + screen + memory;
      if (!this.phonesMap[key]) {
        this.phonesMap[key] = new IphoneFlyweight(model, screen, memory);
      }
      return this.phonesMap[key];
    }
}
```

在这个工厂中，我们定义了一个对象来保存享元对象，并提供一个方法根据参数来获取享元对象，如果 phonesMap 对象中有则直接返回，没有则创建一个返回。

具体实现



定义 IphoneFlyweight 类

```
/**
 * 内部状态：model, screen, memory
 * 外部状态：sn
 */
class IphoneFlyweight {
  constructor(model: string, screen: number, memory: number) { }
}
```

定义 FlyweightFactory 类

```
class FlyweightFactory {
  private phonesMap: { [s: string]: IphoneFlyweight } = {};

  public get(model: string, screen: number, memory: number): IphoneFlyweight {
    const key = model + screen + memory;
    if (!this.phonesMap[key]) {
      this.phonesMap[key] = new IphoneFlyweight(model, screen, memory);
    }
    return this.phonesMap[key];
  }
}
```

定义 Iphone 类

```
class Iphone {
  constructor(flyweight: IphoneFlyweight, sn: number) { }
}
```

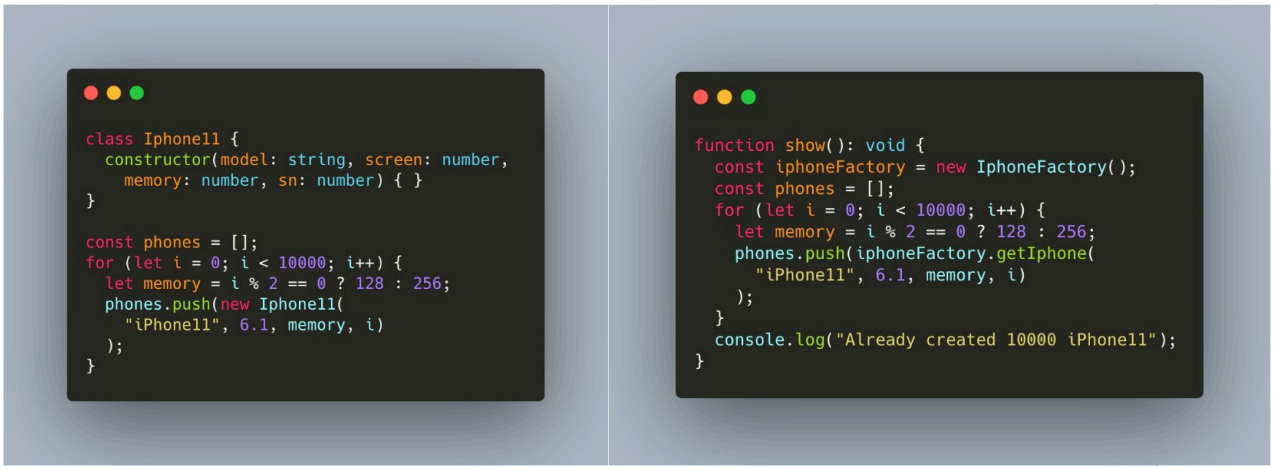
定义 IphoneFactory 类

```
class IphoneFactory {
  private static flyweightFactory: FlyweightFactory = new FlyweightFactory();

  public getIphone(
    model: string,
    screen: number,
    memory: number,
    sn: number
  ) {
    const flyweight: IphoneFlyweight = IphoneFactory.flyweightFactory.get(
      model,
      screen,
      memory
    );
    return new Iphone(flyweight, sn);
  }
}
```

使用示例

```
function show(): void {
  const iphoneFactory = new IphoneFactory();
  const phones = [];
  for (let i = 0; i < 10000; i++) {
    let memory = i % 2 == 0 ? 128 : 256;
    phones.push(iphoneFactory.getIphone("iPhone11", 6.1, memory, i));
  }
  console.log("Already created 10000 iPhone11");
}
```

最后我们来看一下未使用享元模式（左图）和使用享元模式（右图）的代码：



由于享元模式区分了内部状态和外部状态，所以我们可以通过设置不同的外部状态使得相同的对象可以具备一些不同的特性，而内部状态设置为相同部分。

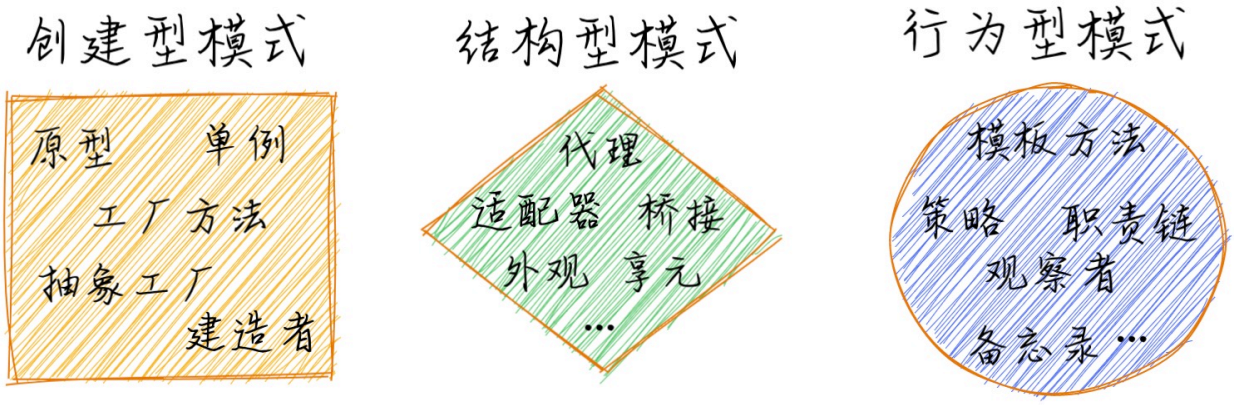
在前面的 iPhone 示例中，我们定义了 IphoneFlyweight 享元类，其中包含型号、屏幕、内存三个内部状态。而对于外部状态如手机编号 sn，我们重新定义另一个 Iphone 类来包含该外部状态。在创建 Iphone 对象时，在型号、屏幕和内存相同的情况下，会共享由 IphoneFlyweight 享元类创建的享元对象。

六、总结

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

第八章 图解常用的九种设计模式

在软件工程中，设计模式（Design Pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。根据模式的目的来划分的话，GoF（Gang of Four）设计模式可以分为以下 3 种类型：



1、创建型模式：用来描述 “如何创建对象”，它的主要特点是 “将对象的创建和使用分离”。包括单例、原型、工厂方法、抽象工厂和建造者 5 种模式。

2、结构型模式：用来描述如何将类或对象按照某种布局组成更大的结构。包括代理、适配器、桥接、装饰、外观、享元和组合 7 种模式。

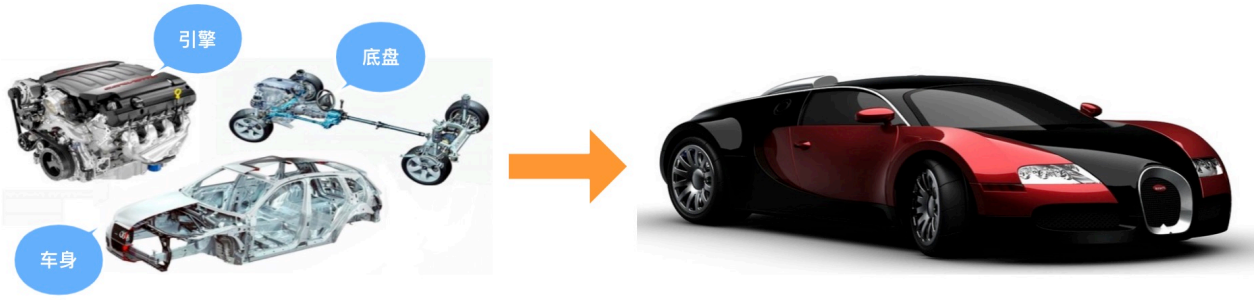
3、行为型模式：用来识别对象之间的常用交流模式以及如何分配职责。包括模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录和解释器 11 种模式。

接下来阿宝哥将结合一些生活中的场景并通过精美的配图，来向大家介绍 9 种常用的设计模式。

一、建造者模式

建造者模式（Builder Pattern）将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

一辆小汽车 🚗 通常由 **发动机、底盘、车身和电气设备** 四大部分组成。汽车电气设备的内部构造很复杂，简单起见，我们只考虑三个部分：引擎、底盘和车身。



在现实生活中，小汽车也是由不同的零部件组装而成，比如上图中我们把小汽车分成引擎、底盘和车身三大部分。下面我们来看一下如何使用建造者模式来造车子。

1.1 实现代码

```
class Car {
  constructor(
    public engine: string,
    public chassis: string,
    public body: string
  ) {}
}

class CarBuilder {
  engine!: string; // 引擎
  chassis!: string; // 底盘
  body!: string; // 车身

  addChassis(chassis: string) {
    this.chassis = chassis;
    return this;
  }

  addEngine(engine: string) {
    this.engine = engine;
    return this;
  }

  addBody(body: string) {
    this.body = body;
    return this;
  }

  build() {
    return new Car(this.engine, this.chassis, this.body);
  }
}
```

在以上代码中，我们定义一个 `CarBuilder` 类，并提供了 `addChassis`、`addEngine` 和 `addBody` 3 个方法用于组装车子的不同部位，当车子的 3 个部分都组装完成后，调用 `build` 方法就可以开始造车。

1.2 使用示例

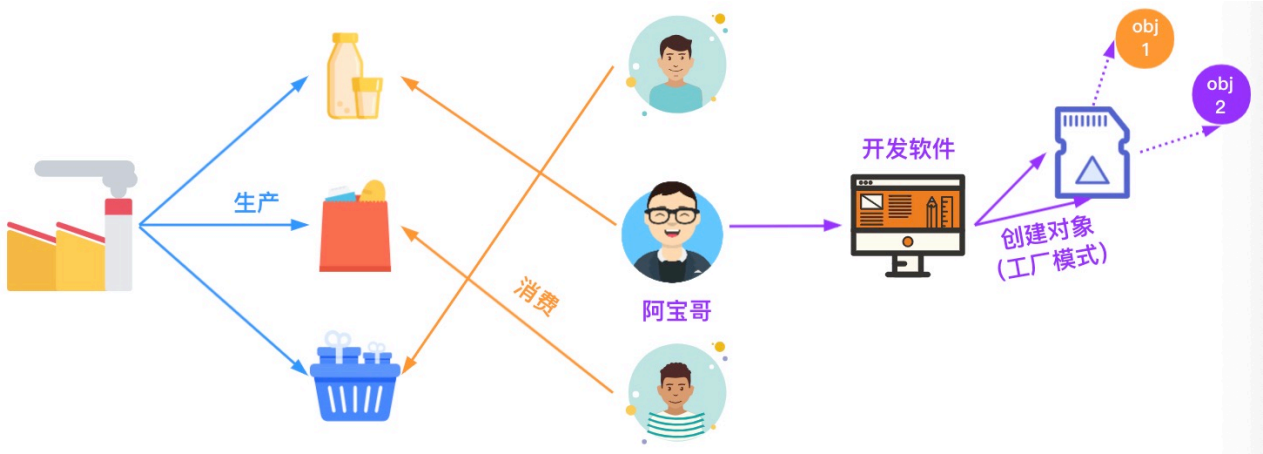
```
const car = new CarBuilder()
  .addEngine('v12')
  .addBody('镁合金')
  .addChassis('复合材料')
  .build();
```

1.3 应用场景及案例

- 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。
- 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。
- 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。
- Github - node-sql-query：https://github.com/dresende/node-sql-query

二、工厂模式

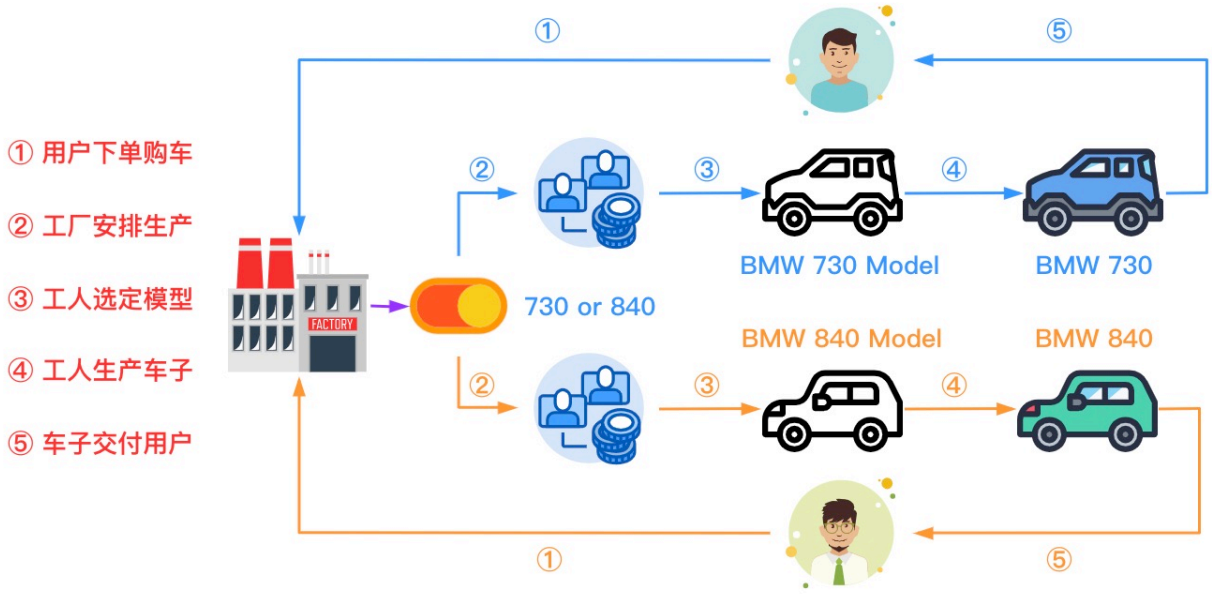
在现实生活中，工厂是负责生产产品的，比如牛奶、面包或礼物等，这些产品满足了我们日常的生理需求。



在众多设计模式当中，有一种被称为工厂模式的设计模式，它提供了创建对象的最佳方式。工厂模式可以分为：**简单工厂模式、工厂方法模式和抽象工厂模式**。

2.1 简单工厂

简单工厂模式又叫 **静态方法模式**，因为工厂类中定义了一个静态方法用于创建对象。简单工厂让使用者不用知道具体的参数就可以创建出所需的 ”产品“ 类，即使用者可以直接消费产品而不需要知道产品的具体生产细节。



在上图中，阿宝哥模拟了用户购车的流程，小王和小秦分别向 BMW 工厂订购了 BMW730 和 BMW840 型号的车型，接着工厂会先判断用户选择的车型，然后按照对应的模型进行生产并在生产完成后交付给用户。

下面我们来看一下如何使用简单工厂来描述 BMW 工厂生产指定型号车子的过程。

2.1.1 实现代码

```
abstract class BMW {
  abstract run(): void;
}

class BMW730 extends BMW {
  run(): void {
    console.log("BMW730 发动咯");
  }
}

class BMW840 extends BMW {
  run(): void {
    console.log("BMW840 发动咯");
  }
}

class BMWFactory {
  public static produceBMW(model: "730" | "840"): BMW {
    if (model === "730") {
      return new BMW730();
    } else {
      return new BMW840();
    }
  }
}
```

在以上代码中，我们定义一个 `BMWFactory` 类，该类提供了一个静态的 `produceBMW()` 方法，用于根据不同的模型参数来创建不同型号的车子。

2.1.2 使用示例

```
const bmw730 = BMWFactory.produceBMW("730");
const bmw840 = BMWFactory.produceBMW("840");

bmw730.run();
bmw840.run();
```

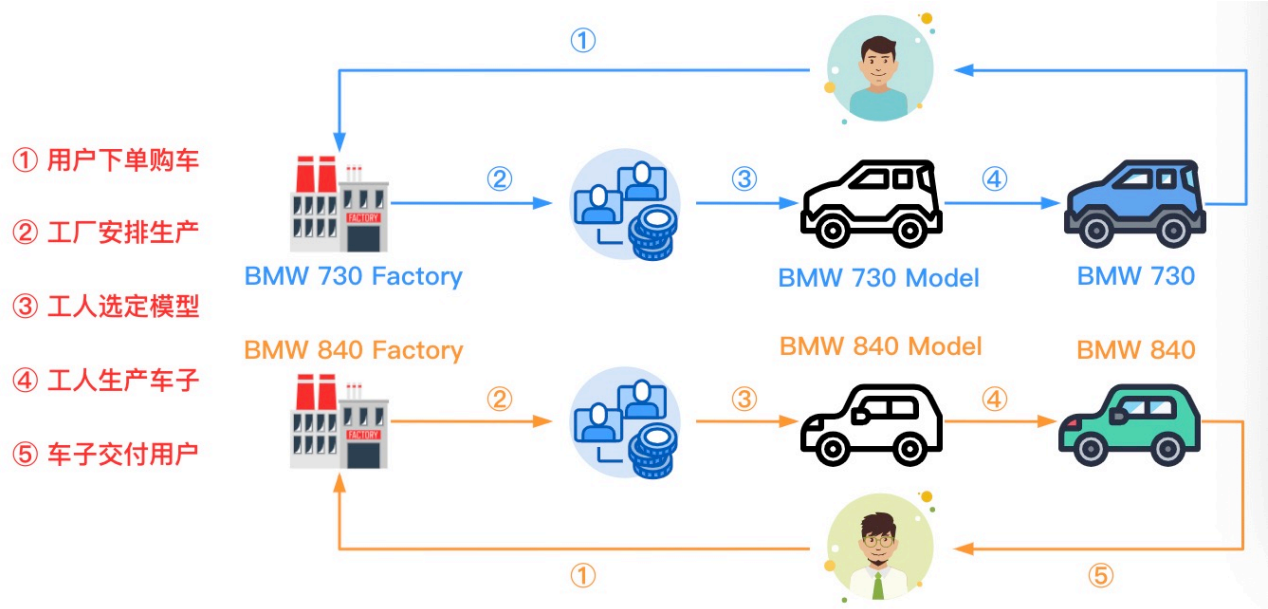
2.1.3 应用场景

- 工厂类负责创建的对象比较少：由于创建的对象比较少，不会造成工厂方法中业务逻辑过于复杂。
- 客户端只需知道传入工厂类静态方法的参数，而不需要关心创建对象的细节。

2.2 工厂方法

工厂方法模式（Factory Method Pattern）又称为工厂模式，也叫多态工厂（Polymorphic Factory）模式，它属于类创建型模式。

在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象， **这样做的目的是将产品类的实例化操作延迟到工厂子类中完成**，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。



在上图中，阿宝哥模拟了用户购车的流程，小王和小秦分别向 BMW 730 和 BMW 840 工厂订购了 BMW730 和 BMW840 型号的车子，接着工厂按照对应的模型进行生产并在生产完成后交付给用户。

同样，我们来看一下如何使用工厂方法来描述 BMW 工厂生产指定型号车子的过程。

2.2.1 实现代码

```
abstract class BMWFactory {
  abstract produceBMW(): BMW;
}

class BMW730Factory extends BMWFactory {
  produceBMW(): BMW {
    return new BMW730();
  }
}

class BMW840Factory extends BMWFactory {
  produceBMW(): BMW {
    return new BMW840();
  }
```

```
}
```

在以上代码中，我们分别创建了 `BMW730Factory` 和 `BMW840Factory` 两个工厂类，然后使用这两个类的实例来生产不同型号的车子。

2.2.2 使用示例

```
const bmw730Factory = new BMW730Factory();
const bmw840Factory = new BMW840Factory();

const bmw730 = bmw730Factory.produceBMW();
const bmw840 = bmw840Factory.produceBMW();

bmw730.run();
bmw840.run();
```

2.2.3 应用场景

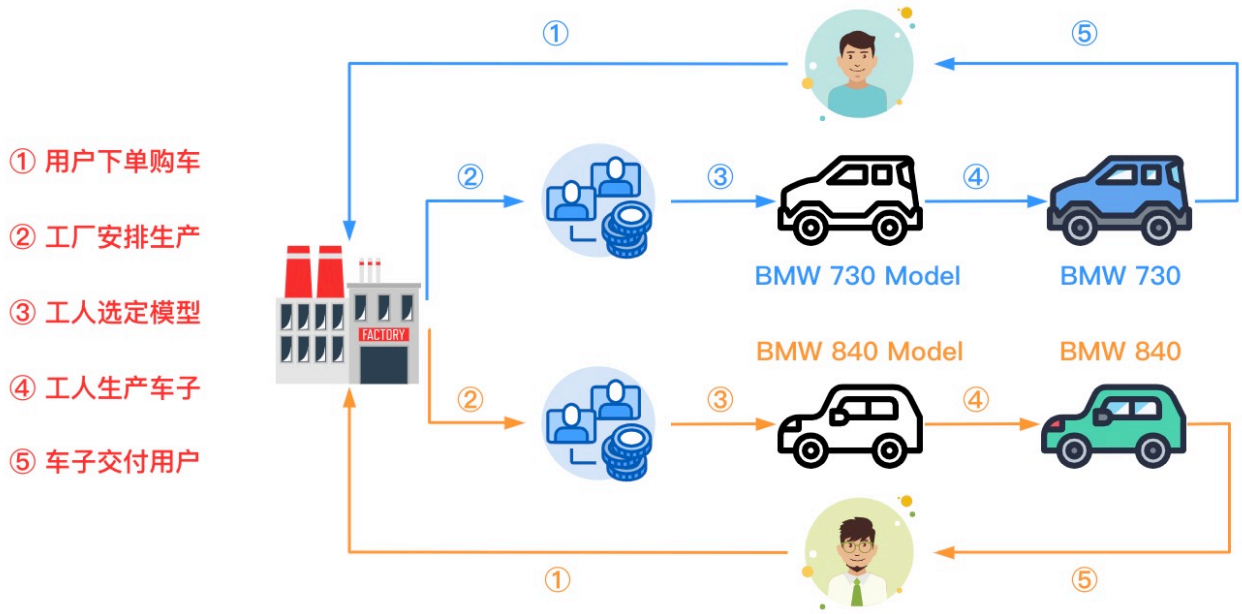
- 一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
- 一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

> 继续阅读：Typescript 设计模式之工厂方法

2.3 抽象工厂

抽象工厂模式（Abstract Factory Pattern），提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。 **但是有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。**



在上图中，阿宝哥模拟了用户购车的流程，小王向 BMW 工厂订购了 BMW730，工厂按照 730 对应的模型进行生产并在生产完成后交付给小王。而小秦向同一个 BMW 工厂订购了 BMW840，工厂按照 840 对应的模型进行生产并在生产完成后交付给小秦。

下面我们来看一下如何使用抽象工厂来描述上述的购车过程。

2.3.1 实现代码

```
abstract class BMWFactory {
  abstract produce730BMW(): BMW730;
  abstract produce840BMW(): BMW840;
}

class ConcreteBMWFactory extends BMWFactory {
  produce730BMW(): BMW730 {
    return new BMW730();
  }

  produce840BMW(): BMW840 {
    return new BMW840();
  }
}
```

2.3.2 使用示例

```
const bmwFactory = new ConcreteBMWFactory();

const bmw730 = bmwFactory.produce730BMW();
const bmw840 = bmwFactory.produce840BMW();

bmw730.run();
bmw840.run();
```

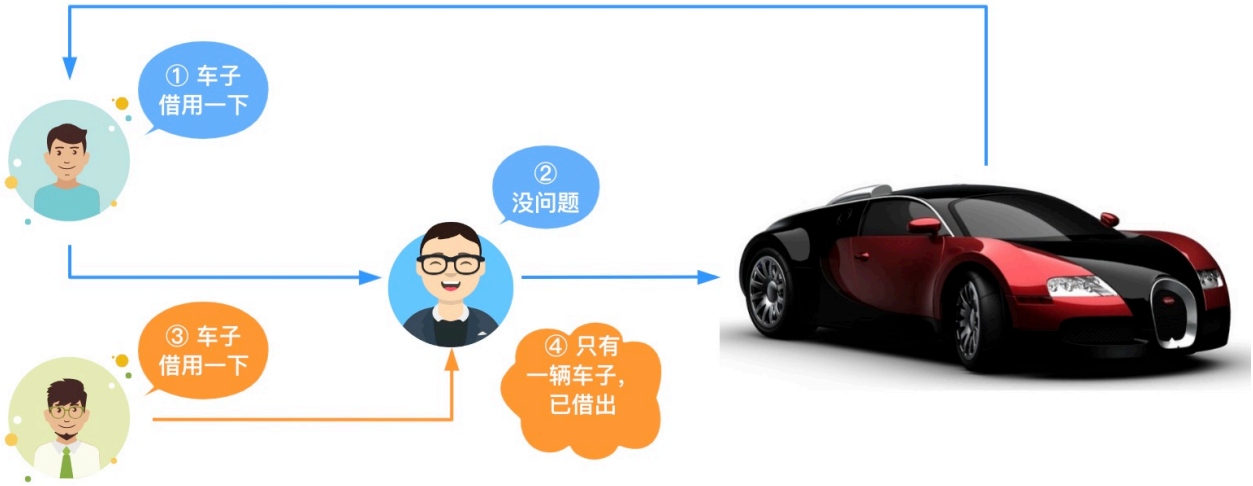
2.3.3 应用场景

- 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。
- 系统中有多于一个的产品族，而每次只使用其中某一产品族。
- 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

> 继续阅读：[创建对象的最佳方式是什么？]

三、单例模式

单例模式（Singleton Pattern）是一种常用的模式，有一些对象我们往往只需要一个，比如全局缓存、浏览器中的 window 对象等。单例模式用于保证一个类仅有一个实例，并提供一个访问它的全局访问点。



在上图中，阿宝哥模拟了借车的流程，小王临时有急事找阿宝哥借车子，阿宝哥家的车子刚好没用，就借给小王了。当天，小秦也需要用车子，也找阿宝哥借车，因为阿宝哥家里只有一辆车子，所以就没有车可借了。

对于车子来说，它虽然给生活带来了很大的便利，但养车也需要一笔不小的费用（车位费、油费和保养费等），所以阿宝哥家里只有一辆车子。在开发软件系统时，如果遇到创建对象时耗时过多或耗资源过多，但又经常用到的对象，我们就可以考虑使用单例模式。

下面我们来看一下如何使用 TypeScript 来实现单例模式。

3.1 实现代码

```
class Singleton {
  // 定义私有的静态属性，来保存对象实例
  private static singleton: Singleton;
  private constructor() {}

  // 提供一个静态的方法来获取对象实例
  public static getInstance(): Singleton {
    if (!Singleton.singleton) {
      Singleton.singleton = new Singleton();
    }
    return Singleton.singleton;
  }
}
```

3.2 使用示例

```
let instance1 = Singleton.getInstance();
let instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true
```

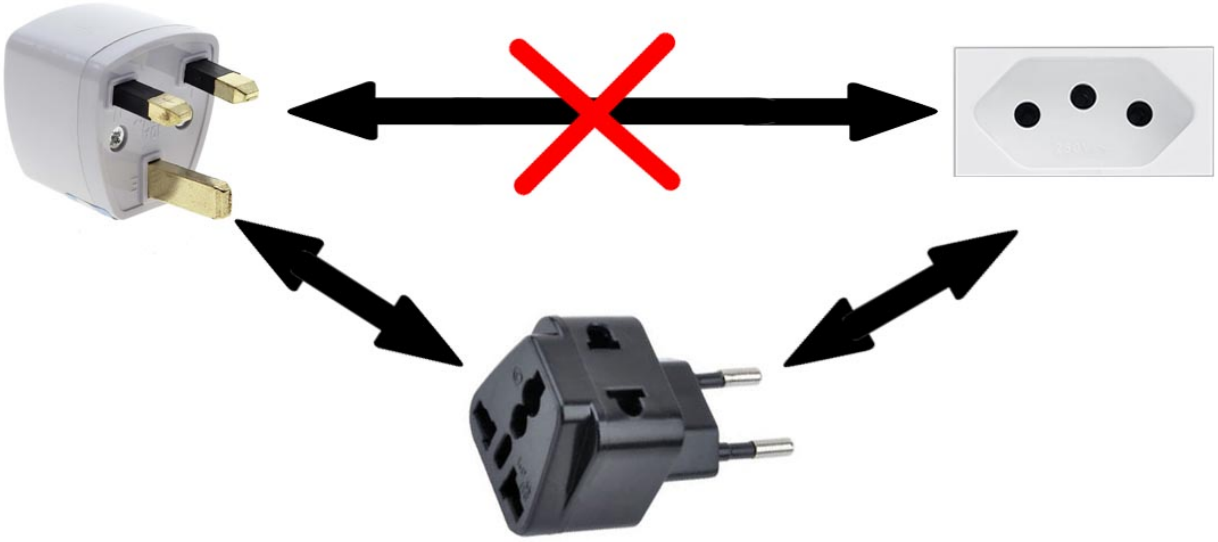
3.3 应用场景

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或耗资源过多，但又经常用到的对象。
- 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器或资源管理器，或者需要考虑资源消耗太大而只允许创建一个对象。

> 继续阅读：TypeScript 设计模式之单例模式

四、适配器模式

在实际生活中，也存在适配器的使用场景，比如：港式插头转换器、电源适配器和 USB 转接口。**而在软件工程中，适配器模式的作用是解决两个软件实体间的接口不兼容的问题。** 使用适配器模式之后，原本由于接口不兼容而不能工作的两个软件实体就可以一起工作。



4.1 实现代码

```
interface Logger {
  info(message: string): Promise<void>;
}

interface CloudLogger {
  sendToServer(message: string, type: string): Promise<void>;
}

class AliLogger implements CloudLogger {
  public async sendToServer(message: string, type: string): Promise<void> {
    console.info(message);
    console.info('This Message was saved with AliLogger');
  }
}

class CloudLoggerAdapter implements Logger {
  protected cloudLogger: CloudLogger;

  constructor (cloudLogger: CloudLogger) {
    this.cloudLogger = cloudLogger;
  }

  public async info(message: string): Promise<void> {
    await this.cloudLogger.sendToServer(message, 'info');
  }
}

class NotificationService {
  protected logger: Logger;

  constructor (logger: Logger) {
    this.logger = logger;
```

```
  }

  public async send(message: string): Promise<void> {
    await this.logger.info(`Notification sended: ${message}`);
  }
}
```

在以上代码中，因为 `Logger` 和 `CloudLogger` 这两个接口不匹配，所以我们引入了 `CloudLoggerAdapter` 适配器来解决兼容性问题。

4.2 使用示例

```
(async () => {
  const aliLogger = new AliLogger();
  const cloudLoggerAdapter = new CloudLoggerAdapter(aliLogger);
  const notificationService = new NotificationService(cloudLoggerAdapter);
  await notificationService.send('Hello semlinker, To Cloud');
})();
```

4.3 应用场景及案例

- 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
- 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。
- Github - axios-mock-adapter：https://github.com/ctimmerm/axios-mock-adapter

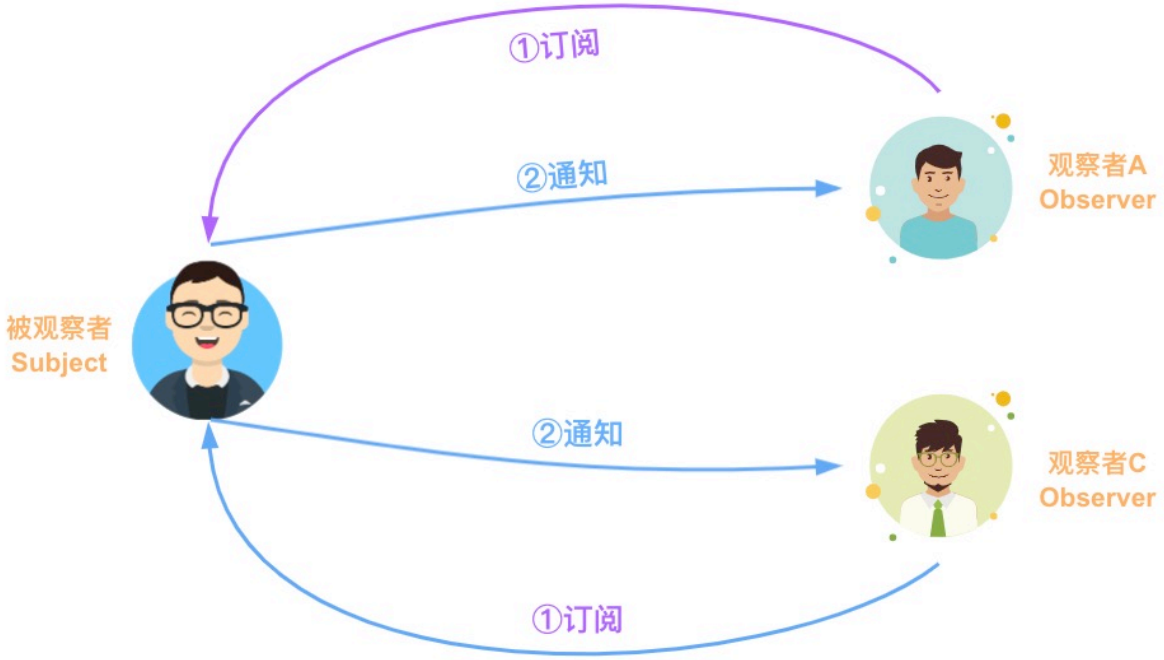
> 继续阅读：TypeScript 设计模式之适配器模式

五、观察者模式 & 发布订阅模式

5.1 观察者模式

观察者模式，它定义了一种一对多的关系，让多个观察者对象同时监听某一个主题对象，这个主题对象的状态发生变化时就会通知所有的观察者对象，使得它们能够自动更新自己。

在观察者模式中有两个主要角色：Subject（主题）和 Observer（观察者）。



在上图中，Subject（主题）就是阿宝哥的 TS 专题文章，而观察者就是小秦和小王。由于观察者模式支持简单的广播通信，当消息更新时，会自动通知所有的观察者。

下面我们来看一下如何使用 TypeScript 来实现观察者模式。

5.1.1 实现代码

```
interface Observer {
  notify: Function;
}

class ConcreteObserver implements Observer{
  constructor(private name: string) {}

  notify() {
    console.log(`${this.name} has been notified.`);
  }
}

class Subject {
  private observers: Observer[] = [];

  public addObserver(observer: Observer): void {
    console.log(observer, "is pushed!");
    this.observers.push(observer);
  }

  public deleteObserver(observer: Observer): void {
    console.log("remove", observer);
    const n: number = this.observers.indexOf(observer);
    n != -1 && this.observers.splice(n, 1);
  }
```

```
  public notifyObservers(): void {
    console.log("notify all the observers", this.observers);
    this.observers.forEach(observer => observer.notify());
  }
}
```

5.1.2 使用示例

```
const subject: Subject = new Subject();
const xiaoQin = new ConcreteObserver("小秦");
const xiaoWang = new ConcreteObserver("小王");
subject.addObserver(xiaoQin);
subject.addObserver(xiaoWang);
subject.notifyObservers();

subject.deleteObserver(xiaoQin);
subject.notifyObservers();
```

5.1.3 应用场景及案例

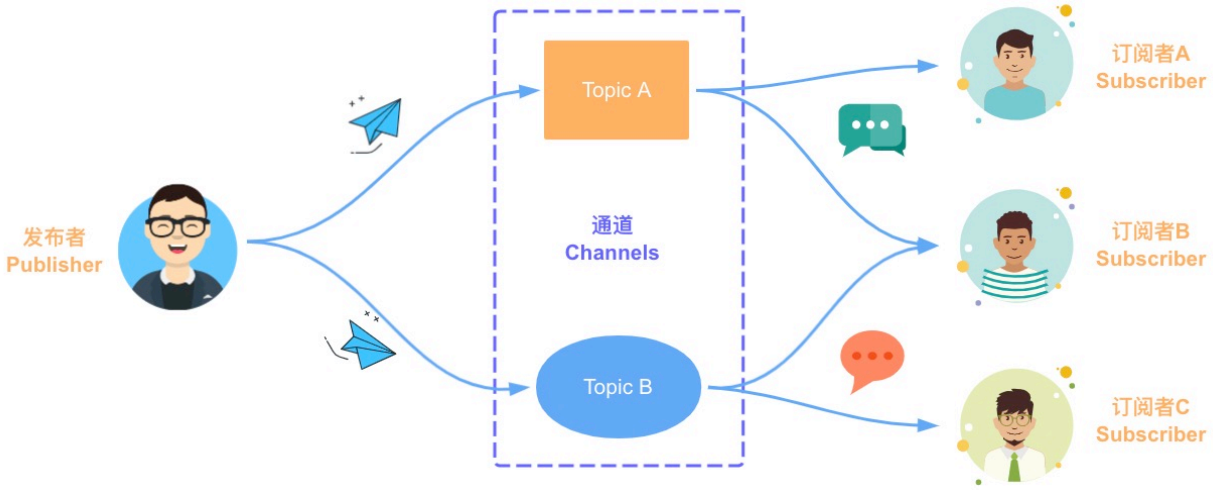
- 一个对象的行为依赖于另一个对象的状态。或者换一种说法，当被观察对象（目标对象）的状态发生改变时，会直接影响到观察对象的行为。
- RxJS Subject：https://github.com/ReactiveX/rxjs/blob/master/src/internal/Subject.ts
- RxJS Subject 文档：https://rxjs.dev/guide/subject

> 继续阅读：TypeScript 设计模式之观察者模式

5.2 发布订阅模式

在软件架构中，发布/订阅是一种消息范式，**消息的发送者（称为发布者）不会将消息直接发送给特定的接收者（称为订阅者）。而是将发布的消息分为不同的类别，然后分别发送给不同的订阅者。** 同样的，订阅者可以表达对一个或多个类别的兴趣，只接收感兴趣的消息，无需了解哪些发布者存在。

在发布订阅模式中有三个主要角色：Publisher（发布者）、Channels（通道）和 Subscriber（订阅者）。



在上图中，Publisher（发布者）是阿宝哥，Channels（通道）中 Topic A 和 Topic B 分别对应于 TS 专题和 Deno 专题，而 Subscriber（订阅者）就是小秦、小王和小池。

下面我们来看一下如何使用 TypeScript 来实现发布订阅模式。

5.2.1 实现代码

```
type EventHandler = (...args: any[]) => any;

class EventEmitter {
  private c = new Map<string, EventHandler[]>();

  // 订阅指定的主题
  subscribe(topic: string, ...handlers: EventHandler[]) {
    let topics = this.c.get(topic);
    if (!topics) {
      this.c.set(topic, topics = []);
    }
    topics.push(...handlers);
  }

  // 取消订阅指定的主题
  unsubscribe(topic: string, handler?: EventHandler): boolean {
    if (!handler) {
      return this.c.delete(topic);
    }

    const topics = this.c.get(topic);
    if (!topics) {
      return false;
    }

    const index = topics.indexOf(handler);

    if (index < 0) {
      return false;
    }
    topics.splice(index, 1);
    if (topics.length === 0) {
      this.c.delete(topic);
    }
    return true;
  }

  // 为指定的主题发布消息
  publish(topic: string, ...args: any[]): any[] | null {
    const topics = this.c.get(topic);
    if (!topics) {
      return null;
```

```
    }
    return topics.map(handler => {
      try {
        return handler(...args);
      } catch (e) {
        console.error(e);
        return null;
      }
    });
  }
}
```

5.2.2 使用示例

```
const eventEmitter = new EventEmitter();
eventEmitter.subscribe("ts", (msg) => console.log(`收到订阅的消息：${msg}`) );

eventEmitter.publish("ts", "TypeScript发布订阅模式");
eventEmitter.unsubscribe("ts");
eventEmitter.publish("ts", "TypeScript发布订阅模式");
```

5.2.3 应用场景

- 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
- 作为事件总线，来实现不同组件间或模块间的通信。
- BetterScroll - EventEmitter：https://github.com/ustbhuangyi/better-scroll/blob/dev/packages/shared-utils/src/events.ts
- EventEmitter 在插件化架构的应用：https://mp.weixin.qq.com/s/N4iw3bi0bxJ57J8EAp5ctQ

> 继续阅读：如何优雅的实现消息通信?

六、策略模式

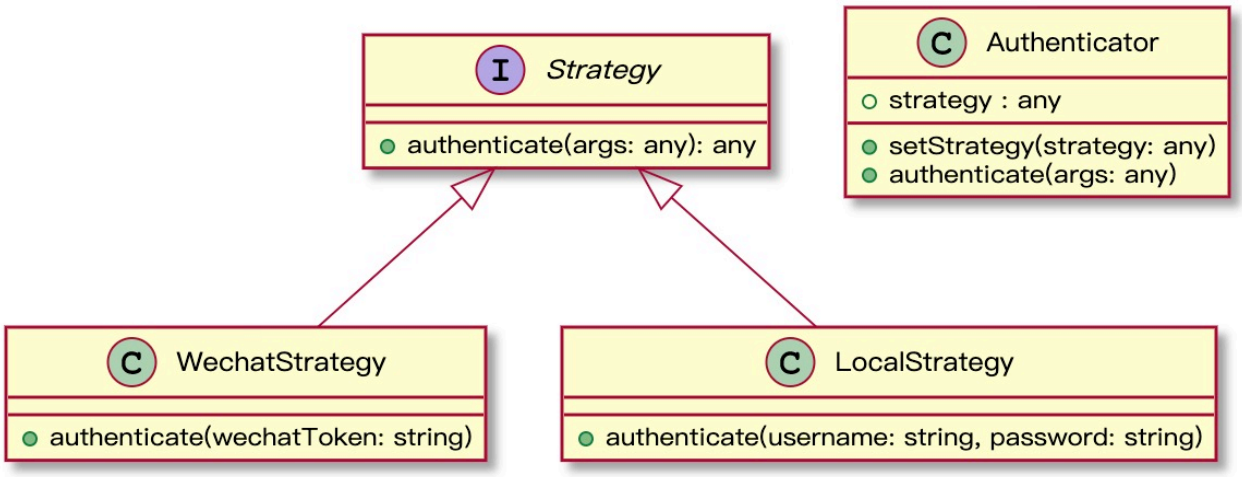
策略模式（Strategy Pattern）定义了一系列的算法，把它们一个个封装起来，并且使它们可以互相替换。策略模式的重心不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活、可维护、可扩展。

目前在一些主流的 Web 站点中，都提供了多种不同的登录方式。比如账号密码登录、手机验证码登录和第三方登录。为了方便维护不同的登录方式，我们可以把不同的登录方式封装成不同的登录策略。

下面我们来看一下如何使用策略模式来封装不同的登录方式。

6.1 实现代码

为了更好地理解以下代码，我们先来看一下对应的 UML 类图：




```
interface Strategy {
  authenticate(...args: any): any;
}

class Authenticator {
  strategy: any;
  constructor() {
    this.strategy = null;
  }

  setStrategy(strategy: any) {
```

```
    this.strategy = strategy;
  }

  authenticate(...args: any) {
    if (!this.strategy) {
      console.log('尚未设置认证策略');
      return;
    }
    return this.strategy.authenticate(...args);
  }
}

class WechatStrategy implements Strategy {
  authenticate(wechatToken: string) {
    if (wechatToken !== '123') {
      console.log('无效的微信用户');
      return;
    }
    console.log('微信认证成功');
  }
}

class LocalStrategy implements Strategy {
  authenticate(username: string, password: string) {
    if (username !== 'abao' && password !== '123') {
      console.log('账号或密码错误');
      return;
    }
    console.log('账号和密码认证成功');
  }
}
```

6.2 使用示例

```
const auth = new Authenticator();

auth.setStrategy(new WechatStrategy());
auth.authenticate('123456');

auth.setStrategy(new LocalStrategy());
auth.authenticate('abao', '123');
```

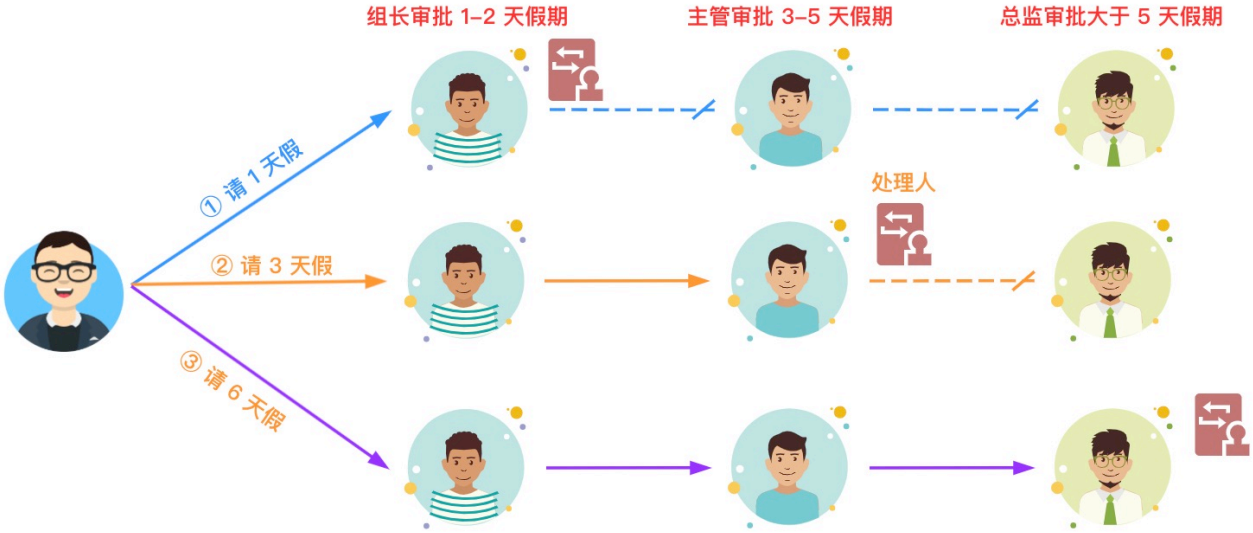
6.3 应用场景及案例

- 一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中。
- 多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句。
- Github - passport-local：https://github.com/jaredhanson/passport-local

- Github - passport-oauth2：https://github.com/jaredhanson/passport-oauth2
- Github - zod：https://github.com/vriad/zod/blob/master/src/types/string.ts

七、职责链模式

职责链模式是使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。在职责链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。

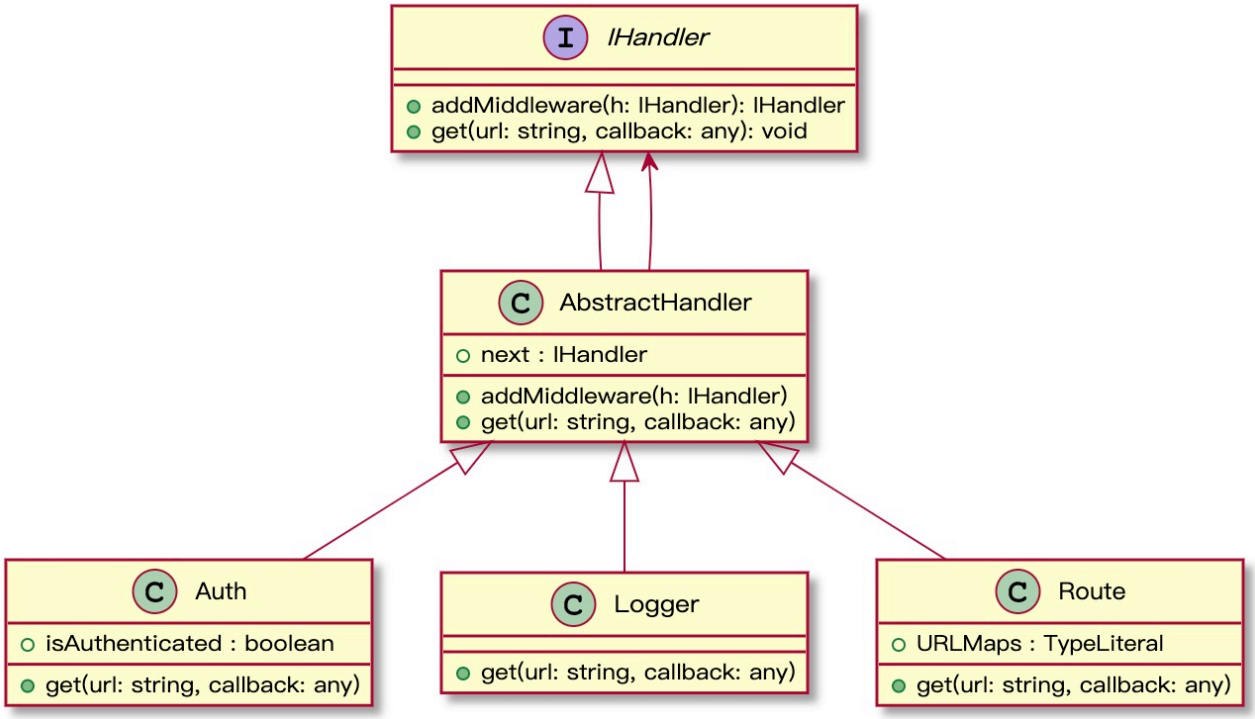


在公司中不同的岗位拥有不同的职责与权限。以上述的请假流程为例，当阿宝哥请 1 天假时，只要组长审批就可以了，不需要流转到主管和总监。如果职责链上的某个环节无法处理当前的请求，若含有下个环节，则会把请求转交给下个环节来处理。

在日常的软件开发过程中，对于职责链来说，一种常见的应用场景是中间件，下面我们来看一下如何利用职责链来处理请求。

7.1 实现代码

为了更好地理解以下代码，我们先来看一下对应的 UML 类图：




```
interface IHandler {
  addMiddleware(h: IHandler): IHandler;
  get(url: string, callback: (data: any) => void): void;
}

abstract class AbstractHandler implements IHandler {
  next!: IHandler;
  addMiddleware(h: IHandler) {
    this.next = h;
    return this.next;
  }

  get(url: string, callback: (data: any) => void) {
    if (this.next) {
      return this.next.get(url, callback);
    }
  }
}

// 定义Auth中间件
class Auth extends AbstractHandler {
  isAuthenticated: boolean;
  constructor(username: string, password: string) {
    super();

    this.isAuthenticated = false;
    if (username === 'abao' && password === '123') {
      this.isAuthenticated = true;
    }
  }
```

```
  get(url: string, callback: (data: any) => void) {
    if (this.isAuthenticated) {
      return super.get(url, callback);
    } else {
      throw new Error('Not Authorized');
    }
  }
}

// 定义Logger中间件
class Logger extends AbstractHandler {
  get(url: string, callback: (data: any) => void) {
    console.log('/GET Request to: ', url);
    return super.get(url, callback);
  }
}

class Route extends AbstractHandler {
  URLMaps: {[key: string]: any};
  constructor() {
    super();
    this.URLMaps = {
      '/api/todos': [{ title: 'learn ts' }, { title: 'learn react' }],
      '/api/random': Math.random(),
    };
  }

  get(url: string, callback: (data: any) => void) {
    super.get(url, callback);

    if (this.URLMaps.hasOwnProperty(url)) {
      callback(this.URLMaps[url]);
    }
  }
}
```

7.2 使用示例

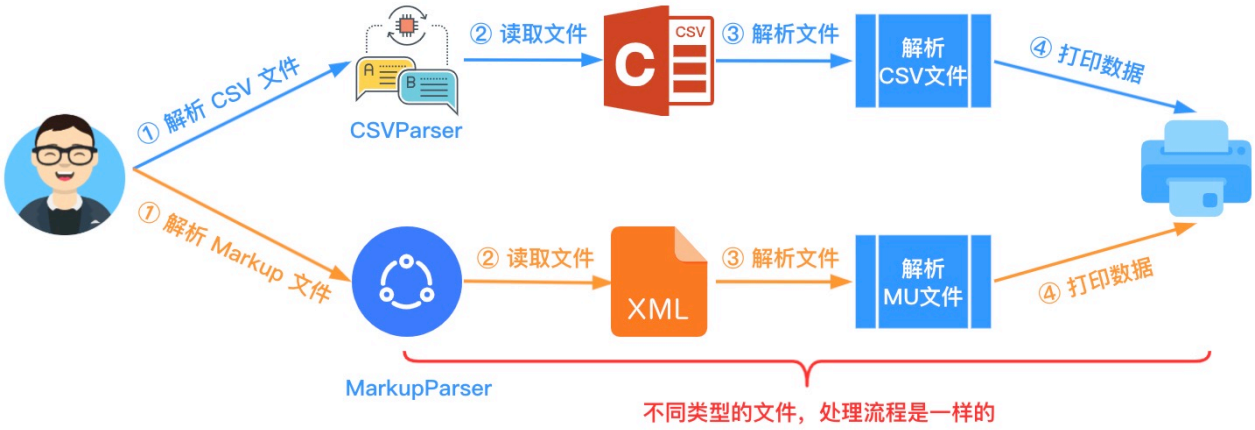
```
const route = new Route();
route.addMiddleware(new Auth('abao', '123')).addMiddleware(new Logger());

route.get('/api/todos', data => {
  console.log(JSON.stringify({ data }, null, 2));
});

route.get('/api/random', data => {
  console.log(data);
});
```

7.3 应用场景

- 可处理一个请求的对象集合应被动态指定。
- 想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 有多个对象可以处理一个请求，哪个对象处理该请求运行时自动确定，客户端只需要把请求提交到链上即可。

八、模板方法模式

模板方法模式由两部分结构组成：抽象父类和具体的实现子类。**通常在抽象父类中封装了子类的算法框架，也包括实现一些公共方法以及封装子类中所有方法的执行顺序**。子类通过继承这个抽象类，也继承了整个算法结构，并且可以选择重写父类的方法。

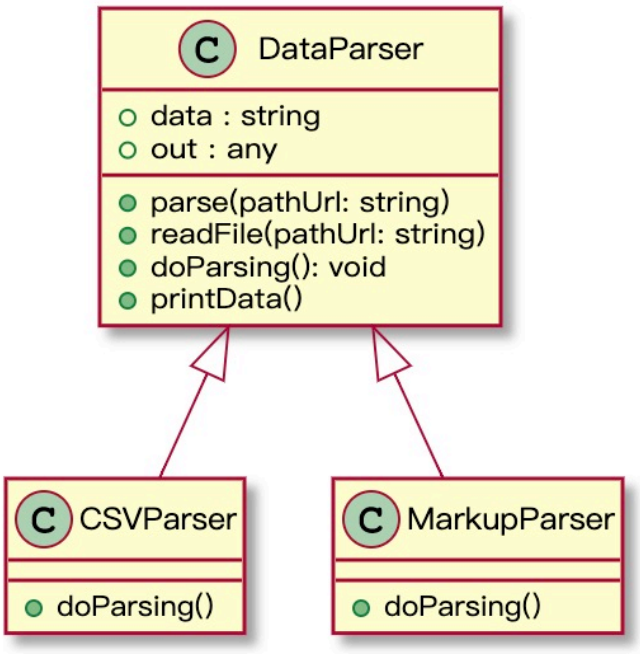


在上图中，阿宝哥通过使用不同的解析器来分别解析 CSV 和 Markup 文件。虽然解析的是不同的类型的文件，但文件的处理流程是一样的。这里主要包含读取文件、解析文件和打印数据三个步骤。针对这个场景，我们就可以引入模板方法来封装以上三个步骤的处理顺序。

下面我们来看一下如何使用模板方法来实现上述的解析流程。

8.1 实现代码

为了更好地理解以下代码，我们先来看一下对应的 UML 类图：




```
import fs from 'fs';

abstract class DataParser {
  data: string = '';
  out: any = null;

  // 这就是所谓的模板方法
  parse(pathUrl: string) {
    this.readFile(pathUrl);
    this.doParsing();
    this.printData();
  }

  readFile(pathUrl: string) {
    this.data = fs.readFileSync(pathUrl, 'utf8');
  }

  abstract doParsing(): void;

  printData() {
    console.log(this.out);
  }
}

class CSVParser extends DataParser {
  doParsing() {
    this.out = this.data.split(',');
  }
}

class MarkupParser extends DataParser {
```

```
  doParsing() {
    this.out = this.data.match(/<\w+>.*<\/\w+>/gim);
  }
}
```

8.2 使用示例

```
const csvPath = './data.csv';
const mdPath = './design-pattern.md';

new CSVParser().parse(csvPath);
new MarkupParser().parse(mdPath);
```

8.3 应用场景

- 算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。
- 当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

九、参考资源

- 维基百科 - 设计模式
- Java设计模式：23种设计模式全面解析
- Design Patterns Everyday

第九章 TypeScript 进阶之插件化架构

近期我们团队的小伙伴小池同学分享了 “**BetterScroll 2.0 发布：精益求精，与你同行**” 这篇文章到团队内部群，看到了 **插件化** 的架构设计，阿宝哥突然来了兴趣，因为之前阿宝哥在团队内部也做过相关的分享。既然已经来了兴趣，那就决定开启 BetterScroll 2.0 源码的学习之旅。

接下来本章的重心将围绕 **插件化** 的架构设计展开，不过在分析 BetterScroll 2.0 插件化架构之前，我们先来简单了解一下 BetterScroll。

一、BetterScroll 简介

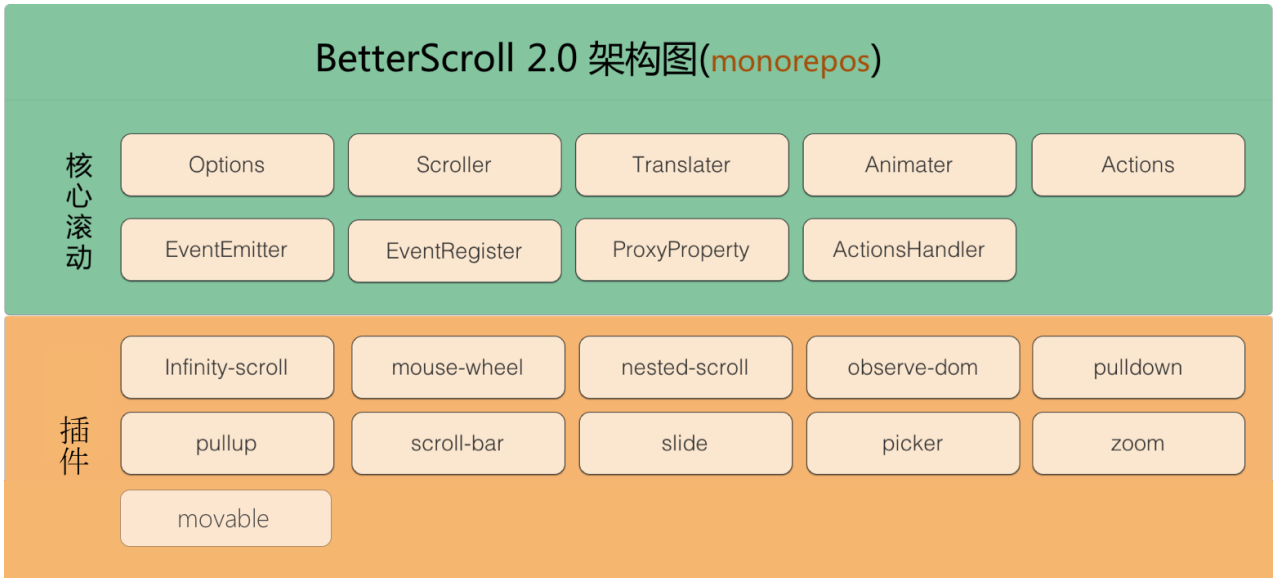
BetterScroll 是一款重点解决移动端（已支持 PC）各种滚动场景需求的插件。它的核心是借鉴的 iscroll 的实现，它的 API 设计基本兼容 iscroll，在 iscroll 的基础上又扩展了一些 feature 以及做了一些性能优化。

BetterScroll 1.0 共发布了 **30** 多个版本，npm 月下载量 **5** 万，累计 star 数 **12600+**。那么为什么升级 2.0 呢?

> 做 v2 版本的初衷源于社区的一个需求：
>
> - **BetterScroll 能不能支持按需加载?**
>
> 来源于：BetterScroll 2.0 发布：精益求精，与你同行

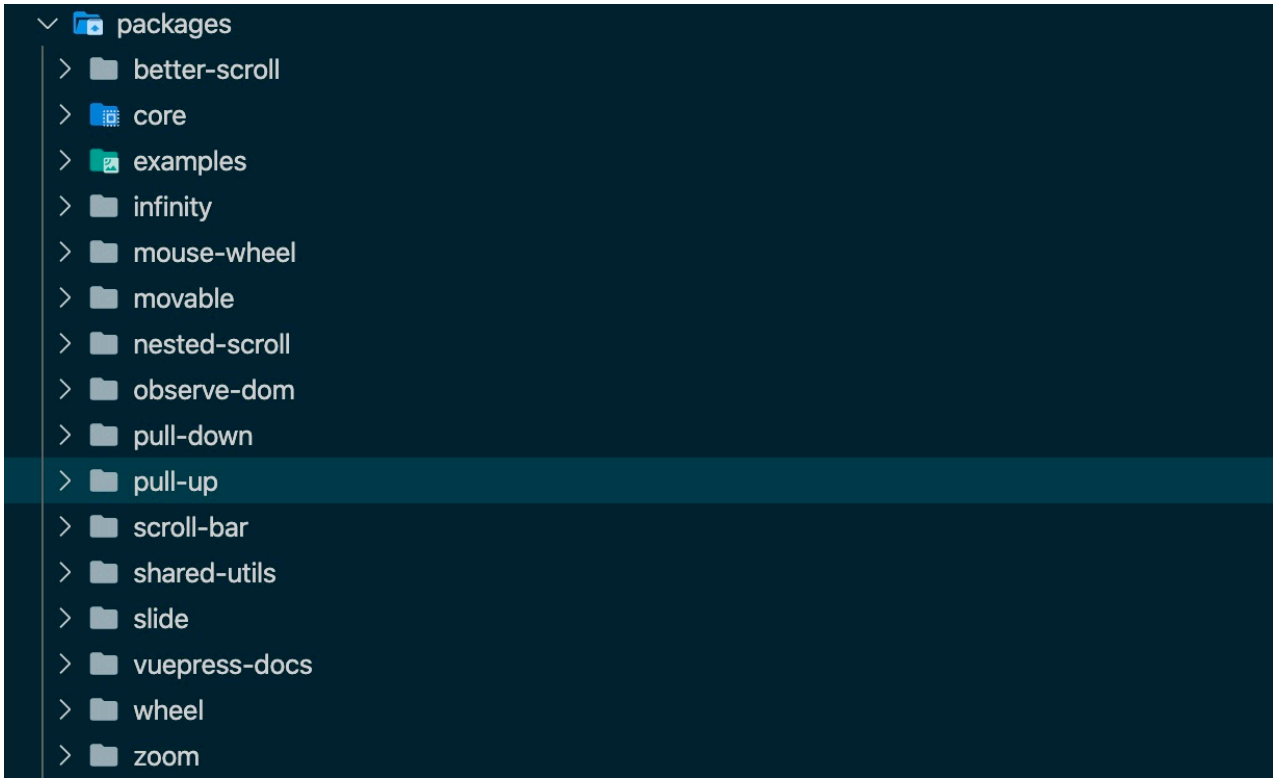
为了支持插件的按需加载，BetterScroll 2.0 采用了 **插件化** 的架构设计。CoreScroll 作为最小的滚动单元，暴露了丰富的**事件**以及**钩子**，其余的功能都由不同的插件来扩展，这样会让 BetterScroll 使用起来更加的灵活，也能适应不同的场景。

下面是 BetterScroll 2.0 整体的架构图：



（图片来源：https://juejin.im/post/6868086607027650573）

该项目采用的是 monorepos 的组织方式，使用 lerna 进行多包管理，每个组件都是一个独立的 npm 包：



与西瓜播放器一样，BetterScroll 2.0 也是采用 **插件化** 的设计思想，CoreScroll 作为最小的滚动单元，其余的功能都是通过插件来扩展。比如长列表中常见的上拉加载和下拉刷新功能，在 BetterScroll 2.0 中这些功能分别通过 `pull-up` 和 `pull-down` 这两个插件来实现。

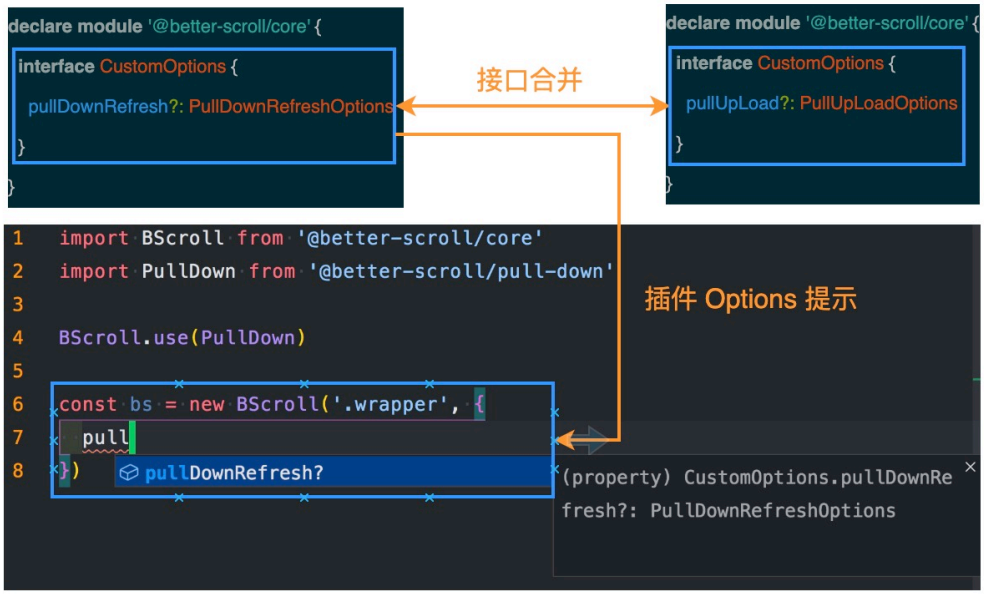
插件化的好处之一就是可以支持按需加载，此外把独立功能都拆分成独立的插件，会让核心系统更加稳定，拥有一定的健壮性。好的，简单介绍了一下 BetterScroll，接下来我们步入正题来分析一下这个项目中一些值得我们学习的地方。

二、开发体验方面

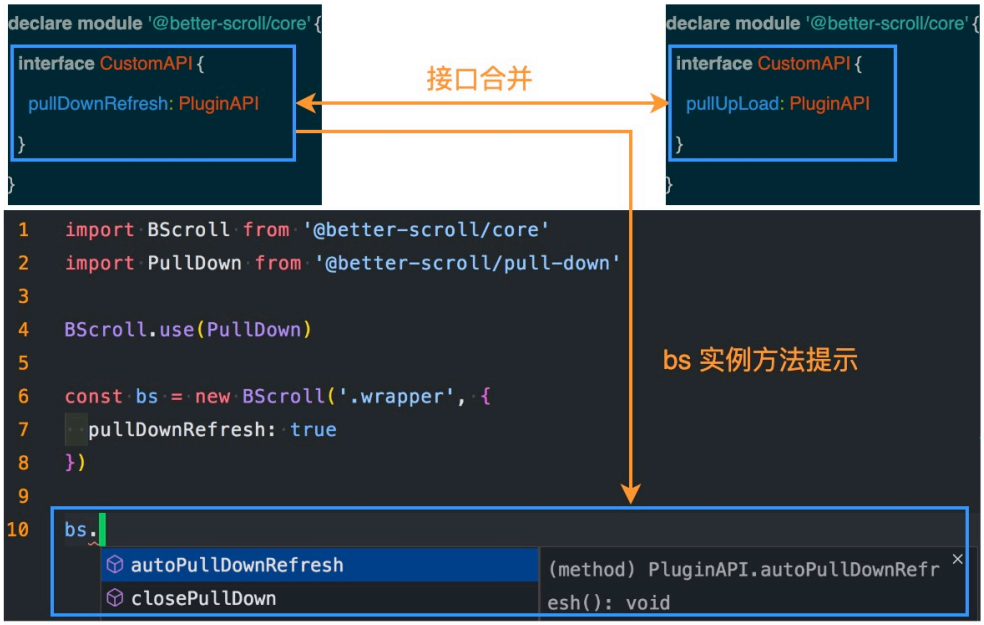
2.1 更好的智能提示

BetterScroll 2.0 采用 TypeScript 进行开发，为了让开发者在使用 BetterScroll 时能够拥有较好的智能提示，BetterScroll 团队充分利用了 TypeScript 接口自动合并的功能，让开发者在使用某个插件时，能够有对应的 Options 提示以及 bs（BetterScroll 实例）能够有对应的方法提示。

2.1.1 智能插件 Options 提示



2.1.2 智能 BetterScroll 实例方法提示



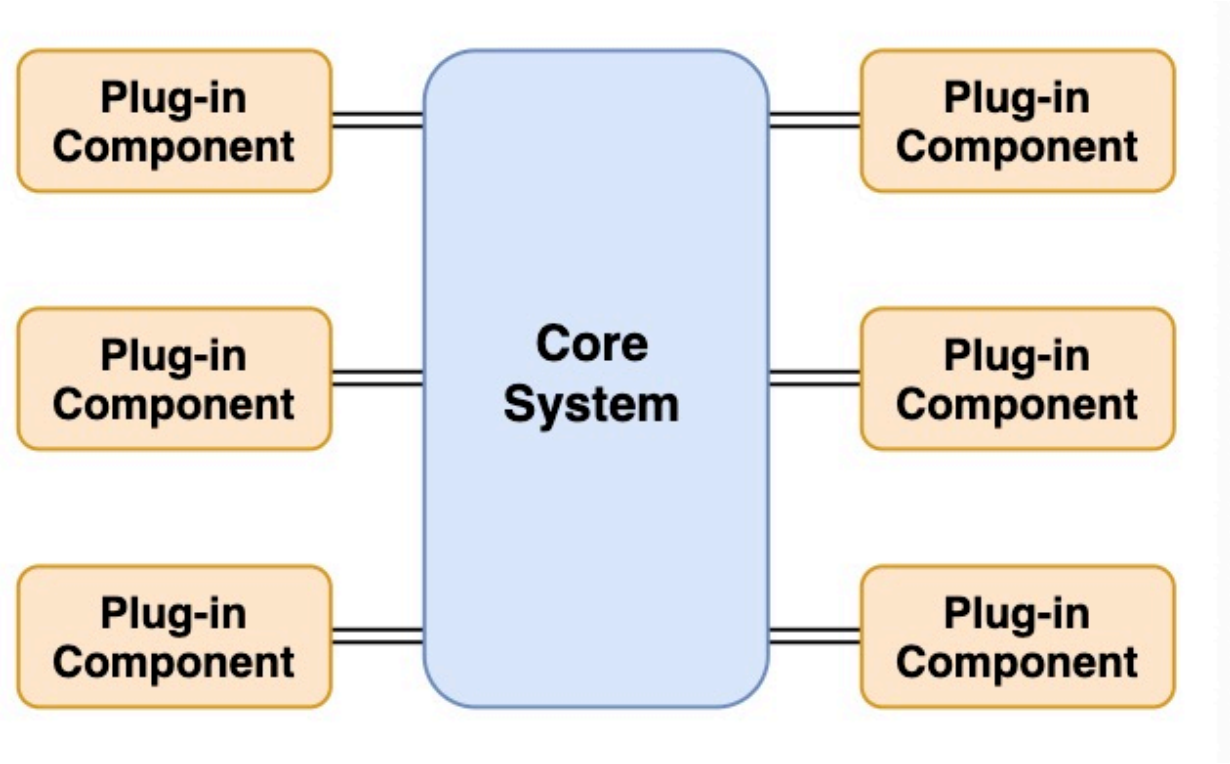
接下来，为了后面能更好地理解 BetterScroll 的设计思想，我们先来简单介绍一下插件化架构。

三、插件化架构简介

3.1 插件化架构的概念

插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于产品的应用。插件化架构模式允许你将其他应用程序功能作为插件添加到核心应用程序，从而提供可扩展性以及功能分离和隔离。

插件化架构模式包括两种类型的架构组件：**核心系统（Core System）和插件模块（Plug-in modules）**。应用逻辑被分割为独立的插件模块和核心系统，提供了可扩展性、灵活性、功能隔离和自定义处理逻辑的特性。



图中 Core System 的功能相对稳定，不会因为业务功能扩展而不断修改，而插件模块是可以根据实际业务功能的需要不断地调整或扩展。 **插件化架构的本质就是将可能需要不断变化的部分封装在插件中，从而达到快速灵活扩展的目的，而又不影响整体系统的稳定。**

插件化架构的核心系统通常提供系统运行所需的最小功能集。插件模块是独立的模块，包含特定的处理、额外的功能和自定义代码，来向核心系统增强或扩展额外的业务能力。 **通常插件模块之间也是独立的，也有一些插件是依赖于若干其它插件的。重要的是，尽量减少插件之间的通信以避免依赖的问题。**

3.2 插件化架构的优点

- 灵活性高：整体灵活性是对环境变化快速响应的能力。由于插件之间的低耦合，改变通常是隔离的，可以快速实现。通常，核心系统是稳定且快速的，具有一定的健壮性，几乎不需要修改。
- 可测试性：插件可以独立测试，也很容易被模拟，不需修改核心系统就可以演示或构建新特性的原型。
- 性能高：虽然插件化架构本身不会使应用高性能，但通常使用插件化架构构建的应用性能都还不错，因为可以自定义或者裁剪掉不需要的功能。

介绍完插件化架构相关的基础知识，接下来我们来分析一下 BetterScroll 2.0 是如何设计插件化架构的。

四、BetterScroll 插件化架构实现

对于插件化的核心系统设计来说，它涉及三个关键点：插件管理、插件连接和插件通信。下面我们将围绕这三个关键点来逐步分析 BetterScroll 2.0 是如何实现插件化架构。

4.1 插件管理

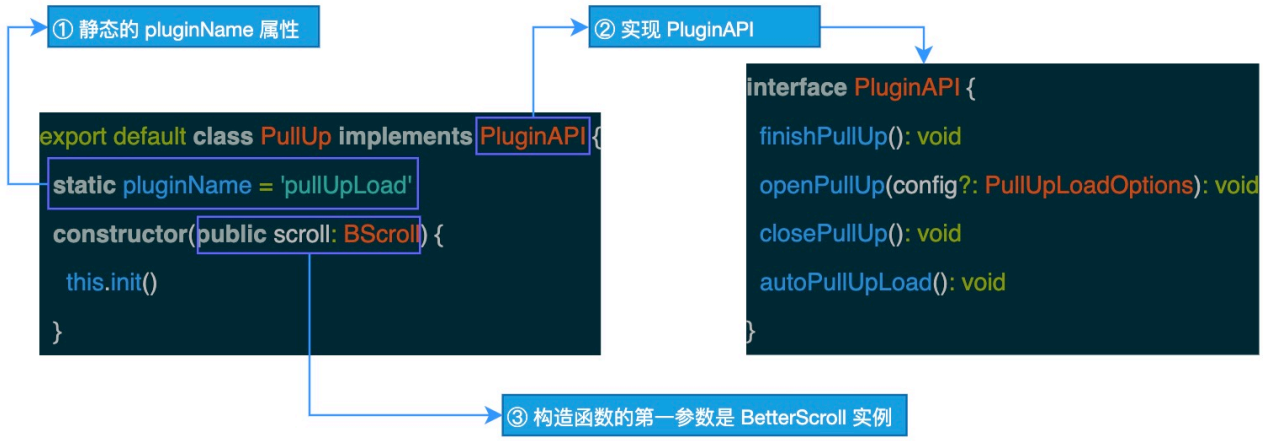
为了统一管理内置的插件，也方便开发者根据业务需求开发符合规范的自定义插件。BetterScroll 2.0 约定了统一的插件开发规范。 **BetterScroll 2.0 的插件需要是一个类，并且具有以下特性：**

1.静态的 pluginName 属性；

2.实现 PluginAPI 接口（当且仅当需要把插件方法代理至 bs）；

3.constructor 的第一个参数就是 BetterScroll 实例 `bs`，你可以通过 bs 的 **事件** 或者 **钩子** 来注入自己的逻辑。

这里为了直观地理解以上的开发规范，我们将以内置的 PullUp 插件为例，来看一下它是如何实现上述规范的。PullUp 插件为 BetterScroll 扩展上拉加载的能力。



顾名思义，静态的 `pluginName` 属性表示插件的名称，而 PluginAPI 接口表示插件实例对外提供的 API 接口，通过 PluginAPI 接口可知它支持 4 个方法：

- finishPullUp(): void：结束上拉加载行为；
- openPullUp(config?: PullUpLoadOptions): void：动态开启上拉功能；
- closePullUp(): void：关闭上拉加载功能；
- autoPullUpLoad(): void：自动执行上拉加载。

插件通过构造函数注入 BetterScroll 实例 `bs`，之后我们就可以通过 bs 的**事件**或者**钩子**来注入自己的逻辑。那么为什么要注入 bs 实例？如何利用 bs 实例？这里我们先记住这些问题，后面我们再来分析它们。

4.2 插件连接

核心系统需要知道当前有哪些插件可用，如何加载这些插件，什么时候加载插件。常见的实现方法是插件注册表机制。核心系统提供插件注册表（可以是配置文件，也可以是代码，还可以是数据库），插件注册表含有每个插件模块的信息，包括它的名字、位置、加载时机（启动就加载，或是按需加载）等。

这里我们以前面提到的 PullUp 插件为例，来看一下如何注册和使用该插件。首先你需要使用以下命令安装 PullUp 插件：

```
$ npm install @better-scroll/pull-up --save
```

成功安装完 pullup 插件之后，你需要通过 `BScroll.use` 方法来注册插件：

```
import BScroll from '@better-scroll/core'
import Pullup from '@better-scroll/pull-up'

BScroll.use(Pullup)
```

然后，实例化 BetterScroll 时需要传入 PullUp 插件的配置项。

```
new BScroll('.bs-wrapper', {
  pullUpLoad: true
})
```

现在我们已经知道通过 `BScroll.use` 方法可以注册插件，那么该方法内部做了哪些处理？要回答这个问题，我们来看一下对应的源码：

```
// better-scroll/packages/core/src/BScroll.ts
export const BScroll = (createBScroll as unknown) as BScrollFactory
createBScroll.use = BScrollConstructor.use
```

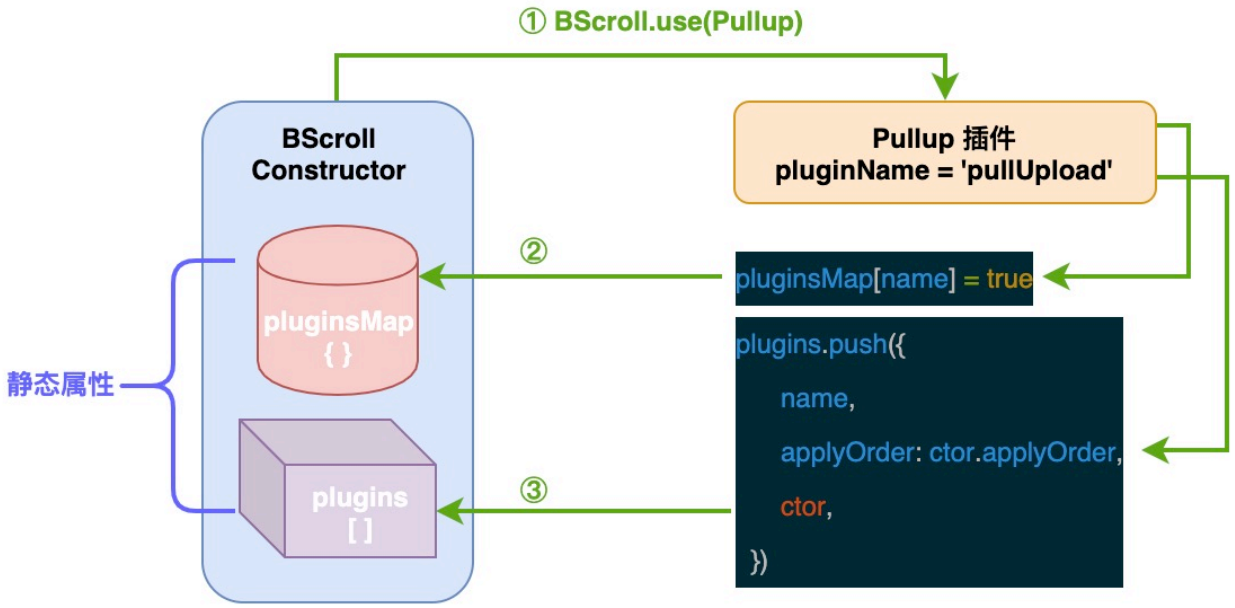
在 `BScroll.ts` 文件中， `BScroll.use` 方法指向的是 `BScrollConstructor.use` 静态方法，该方法的实现如下：

```
export class BScrollConstructor<O = {}> extends EventEmitter {
  static plugins: PluginItem[] = []
  static pluginsMap: PluginsMap = {}

  static use(ctor: PluginCtor) {
    const name = ctor.pluginName
    const installed = BScrollConstructor.plugins.some(
      (plugin) => ctor === plugin.ctor
    )
    // 省略部分代码
    if (installed) return BScrollConstructor
    BScrollConstructor.pluginsMap[name] = true
    BScrollConstructor.plugins.push({
      name,
      applyOrder: ctor.applyOrder,
      ctor,
    })
    return BScrollConstructor
  }
}
```

通过观察以上代码，可知 `use` 方法接收一个参数，该参数的类型是 `PluginCtor`，用于描述插件构造函数的特点。`PluginCtor` 类型的具体声明如下所示：

```
interface PluginCtor {
  pluginName: string
  applyOrder?: ApplyOrder
  new (scroll: BScroll): any
}
```

当我们调用 `BScroll.use(Pullup)` 方法时，会先获取当前插件的名称，然后判断当前插件是否已经安装过了。如果已经安装则直接返回 BScrollConstructor 对象，否则会对插件进行注册。即把当前插件的信息分别保存到 pluginsMap（{}） 和 plugins（[]） 对象中：



另外调用 `use` 静态方法后，会返回 `BScrollConstructor` 对象，这是为了支持链式调用：

```
BScroll.use(MouseWheel)
  .use(ObserveDom)
  .use(PullDownRefresh)
  .use(PullUpLoad)
```

现在我们已经知道 `BScroll.use` 方法内部是如何注册插件的，注册插件只是第一步，要使用已注册的插件，我们还需要在实例化 BetterScroll 时传入插件的配置项，从而进行插件的初始化。对于 PullUp 插件，我们通过以下方式进行插件的初始化。

```
new BScroll('.bs-wrapper', {
  pullUpLoad: true
})
```

所以想了解插件是如何连接到核心系统并进行插件初始化，我们就需要来分析一下 `BScroll` 构造函数：

```ts
// packages/core/src/BScroll.ts
export const BScroll = (createBScroll as unknown) as BScrollFactory

export function createBScroll<O = {}>(
  el: ElementParam,
  options?: Options & O
): BScrollConstructor & UnionToIntersection<ExtractAPI<O>> {
  const bs = new BScrollConstructor(el, options)
  return (bs as unknown) as BScrollConstructor &
    UnionToIntersection<ExtractAPI<O>>
}
```

在 `createBScroll` 工厂方法内部会通过 `new` 关键字调用 `BScrollConstructor` 构造函数来创建 BetterScroll 实例。因此接下来的重点就是分析 `BScrollConstructor` 构造函数：

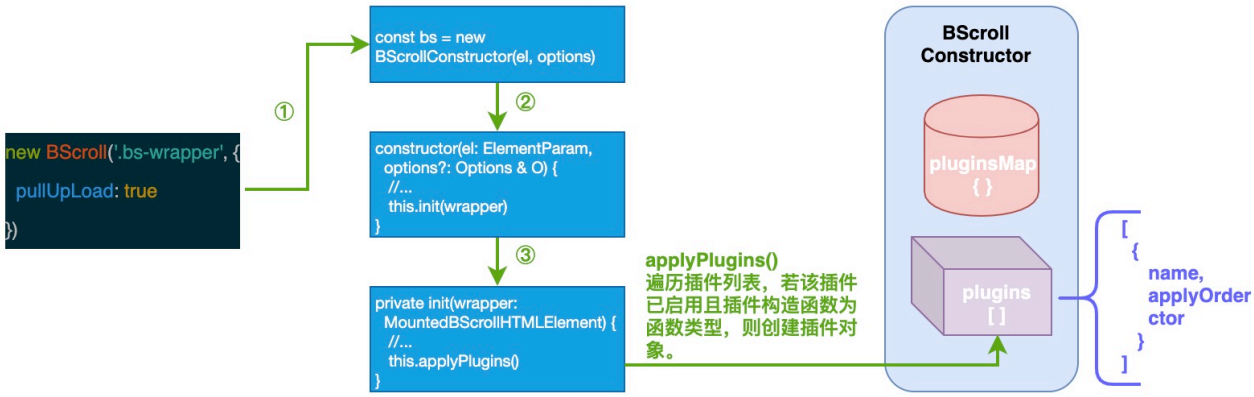
```ts
// packages/core/src/BScroll.ts
export class BScrollConstructor<O = {}> extends EventEmitter {
  constructor(el: ElementParam, options?: Options & O) {
    const wrapper = getElement(el)
    // 省略部分代码
    this.plugins = {}
    this.hooks = new EventEmitter([...])
    this.init(wrapper)
  }

  private init(wrapper: MountedBScrollHTMLElement) {
    this.wrapper = wrapper
    // 省略部分代码
    this.applyPlugins()
  }
}
```

通过阅读 BScrollConstructor 的源码，我们发现在 BScrollConstructor 构造函数内部会调用 `init` 方法进行初始化，而在 `init` 方法内部会进一步调用 `applyPlugins` 方法来应用已注册的插件：

```ts
// packages/core/src/BScroll.ts
export class BScrollConstructor<O = {}> extends EventEmitter {
  private applyPlugins() {
    const options = this.options
    BScrollConstructor.plugins
      .sort((a, b) => {
        const applyOrderMap = {
          [ApplyOrder.Pre]: -1,
          [ApplyOrder.Post]: 1,
        }
        const aOrder = a.applyOrder ? applyOrderMap[a.applyOrder] : 0
        const bOrder = b.applyOrder ? applyOrderMap[b.applyOrder] : 0
```

```
        return aOrder - bOrder
      })
      .forEach((item: PluginItem) => {
        const ctor = item.ctor
        // 当启用指定插件的时候且插件构造函数的类型是函数的话，再创建对应的插件
        if (options[item.name] && typeof ctor === 'function') {
          this.plugins[item.name] = new ctor(this)
        }
      })
  }
}
```

在 `applyPlugins` 方法内部会根据插件设置的顺序进行排序，然后会使用 `bs` 实例作为参数调用插件的构造函数来创建插件，并把插件的实例保存到 `bs` 实例内部的 plugins（{}） 属性中。

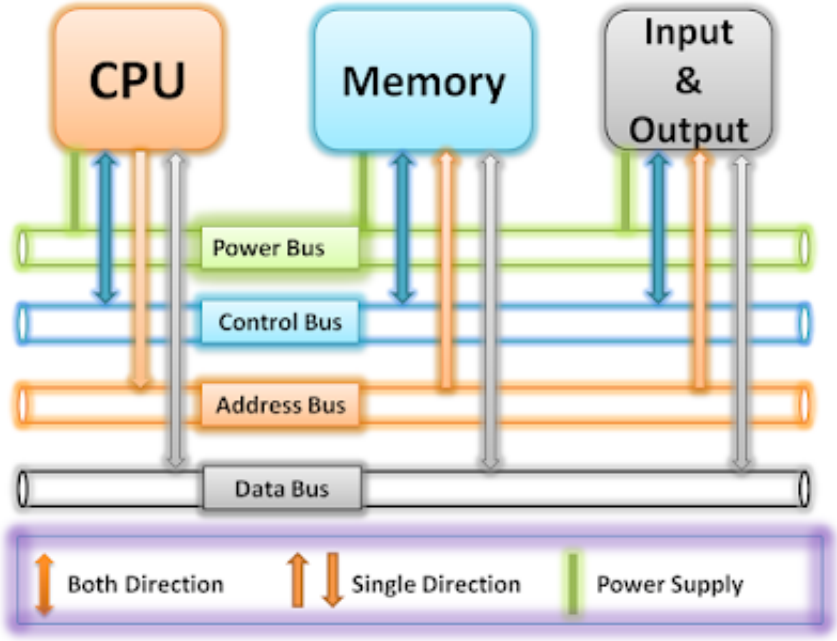


到这里我们已经介绍了插件管理和插件连接，下面我们来介绍最后一个关键点 —— 插件通信。

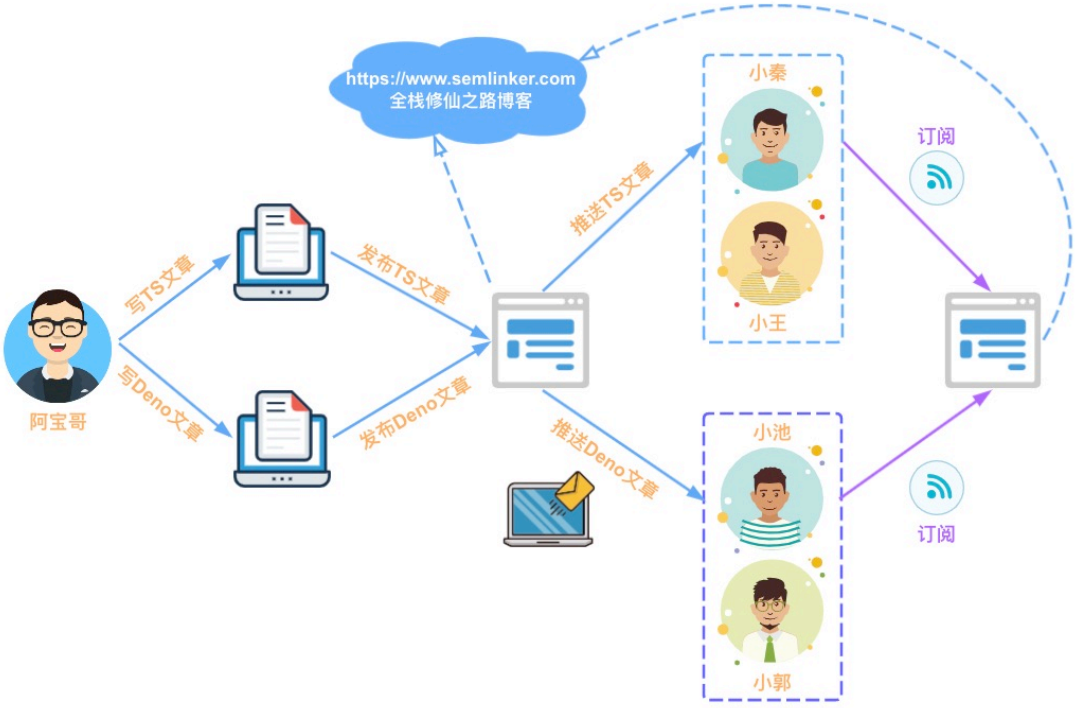
4.3 插件通信

插件通信是指插件间的通信。虽然设计的时候插件间是完全解耦的，但实际业务运行过程中，必然会出现某个业务流程需要多个插件协作，这就要求两个插件间进行通信； **由于插件之间没有直接联系，通信必须通过核心系统，因此核心系统需要提供插件通信机制**。

这种情况和计算机类似，计算机的 CPU、硬盘、内存、网卡是独立设计的配置，但计算机运行过程中，CPU 和内存、内存和硬盘肯定是有通信的，计算机通过主板上的总线提供了这些组件之间的通信功能。



同样，对于插件化架构的系统来说，通常核心系统会以事件总线的形式提供插件通信机制。提到事件总线，可能有一些小伙伴会有一些陌生。但如果说是使用了 **发布订阅模式** 的话，应该就很容易理解了。这里阿宝哥不打算在展开介绍发布订阅模式，只用一张图来回顾一下该模式。

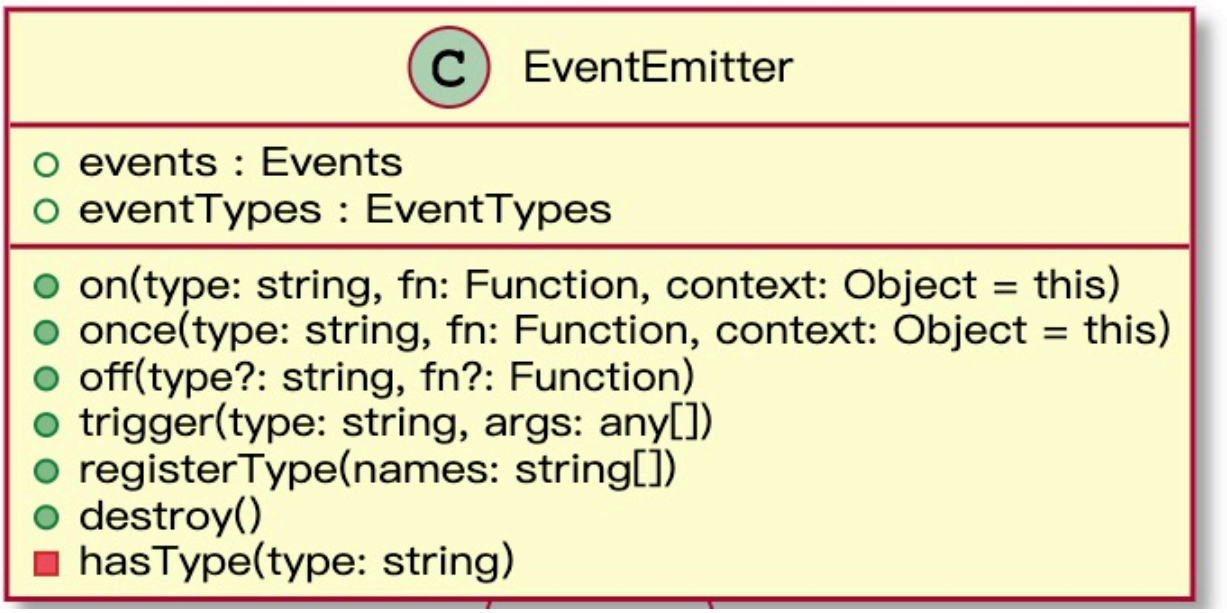


对于 BetterScroll 来说，它的核心是 `BScrollConstructor` 类，该类继承了 `EventEmitter` 事件派发器：

```
// packages/core/src/BScroll.ts
export class BScrollConstructor<O = {}> extends EventEmitter {
  constructor(el: ElementParam, options?: Options & O) {
    this.hooks = new EventEmitter([
      'refresh',
      'enable',
```

```
      'disable',
      'destroy',
      'beforeInitialScrollTo',
      'contentChanged',
    ])
    this.init(wrapper)
  }
}
```

EventEmitter 类是由 BetterScroll 内部提供的，它的实例将会对外提供事件总线的功能，而该类对应的 UML 类图如下所示：



讲到这里我们就可以来回答前面留下的第一个问题：“那么为什么要注入 bs 实例？”。因为 bs（BScrollConstructor）实例的本质也是一个事件派发器，在创建插件时，注入 bs 实例是为了让插件间能通过统一的事件派发器进行通信。

第一个问题我们已经知道答案了，接下来我们来看第二个问题：”如何利用 bs 实例？“。要回答这个问题，我们将继续以 PullUp 插件为例，来看一下该插件内部是如何利用 bs 实例进行消息通信的。

```
export default class PullUp implements PluginAPI {
  static pluginName = 'pullUpLoad'
  constructor(public scroll: BScroll) {
    this.init()
  }
}
```

在 PullUp 构造函数中，bs 实例会被保存到 PullUp 实例内部的 `scroll` 属性中，之后在 PullUp 插件内部就可以通过注入的 bs 实例来进行事件通信。比如派发插件的内部事件，在 PullUp 插件中，当距离滚动到底部小于 `threshold` 值时，触发一次 `pullingUp` 事件：

```
private checkPullUp(pos: { x: number; y: number }) {
  const { threshold } = this.options
  if (...) {
      this.pulling = true
      // 省略部分代码
      this.scroll.trigger(PULL_UP_HOOKS_NAME) // 'pullingUp'
  }
}
```

知道如何利用 bs 实例派发事件之后，我们再来看一下在插件内部如何利用它来监听插件所感兴趣的事件。

```
// packages/pull-up/src/index.ts
export default class PullUp implements PluginAPI {
  static pluginName = 'pullUpLoad'
  constructor(public scroll: BScroll) {
    this.init()
  }

  private init() {
    this.handleBScroll()
    this.handleOptions(this.scroll.options.pullUpLoad)
    this.handleHooks()
    this.watch()
  }
}
```

在 PullUp 构造函数中会调用 `init` 方法进行插件初始化，而在 `init` 方法内部会分别调用不同的方法执行不同的初始化操作，这里跟事件相关的是 `handleHooks` 方法，该方法的实现如下：

```
private handleHooks() {
  this.hooksFn = []
  // 省略部分代码
  this.registerHooks(
    this.scroll.hooks,
    this.scroll.hooks.eventTypes.contentChanged,
    () => {
      this.finishPullUp()
    }
  )
}
```

很明显在 `handleHooks` 方法内部，会进一步调用 `registerHooks` 方法来注册钩子：

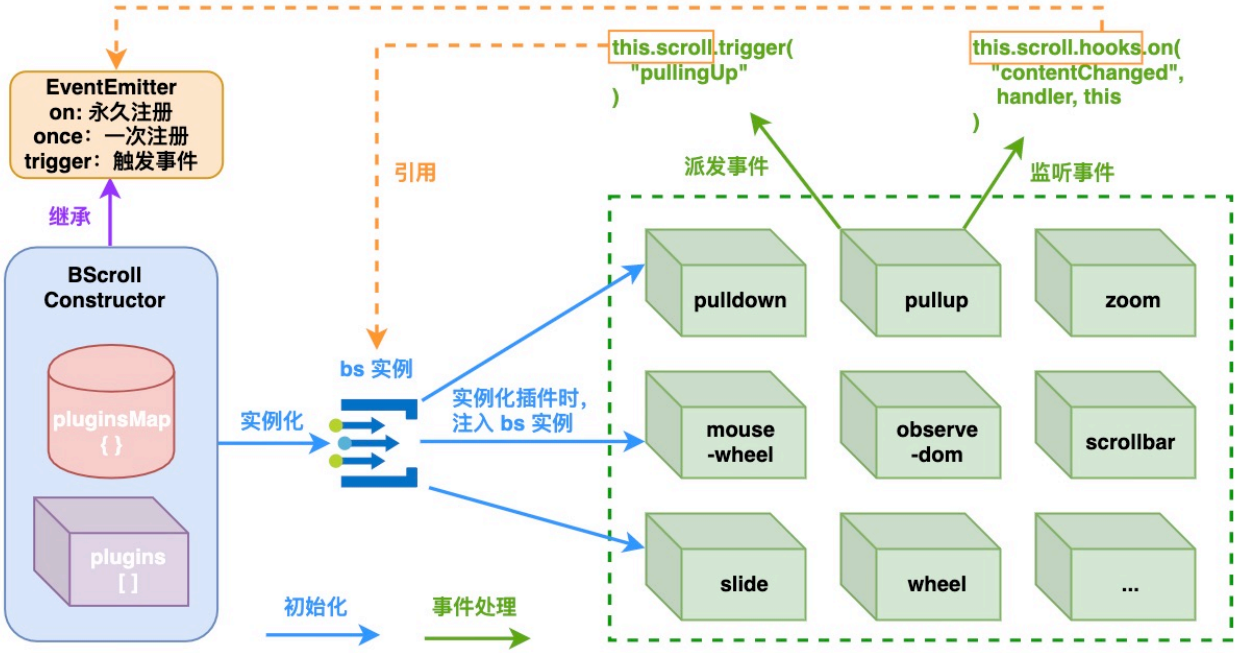
```
private registerHooks(hooks: EventEmitter, name: string, handler: Function) {
  hooks.on(name, handler, this)
  this.hooksFn.push([hooks, name, handler])
}
```

通过观察 `registerHooks` 方法的签名可知，它支持 3 个参数，第 1 个参数是 `EventEmitter` 对象，而另外 2 个参数分别表示事件名和事件处理器。在 `registerHooks` 方法内部，它就是简单地通过 `hooks` 对象来监听指定的事件。

那么 `this.scroll.hooks` 对象是什么时候创建的呢？在 `BScrollConstructor` 构造函数中我们找到了答案。

```
// packages/core/src/BScroll.ts
export class BScrollConstructor<O = {}> extends EventEmitter {
  constructor(el: ElementParam, options?: Options & O) {
    // 省略部分代码
    this.hooks = new EventEmitter([
      'refresh',
      'enable',
      'disable',
      'destroy',
      'beforeInitialScrollTo',
      'contentChanged',
    ])
  }
}
```

很明显 `this.hooks` 也是一个 `EventEmitter` 对象，所以可以通过它来进行事件处理。好的，插件通信的内容就先介绍到这里，下面我们用一张图来总结一下该部分的内容：



介绍完 BetterScroll 插件化架构的实现，最后我们来简单聊一下 BetterScroll 项目工程化方面的内容。

五、工程化方面

在工程化方面，BetterScroll 使用了业内一些常见的解决方案：

- lerna：Lerna 是一个管理工具，用于管理包含多个软件包（package）的 JavaScript 项目。
- prettier：Prettier 中文的意思是漂亮的、美丽的，是一个流行的代码格式化的工具。
- tslint：TSLint 是可扩展的静态分析工具，用于检查 TypeScript 代码的可读性，可维护性和功能性错误。
- commitizen & cz-conventional-changelog：用于帮助我们生成符合规范的 commit message。
- husky：husky 能够防止不规范代码被 commit、push、merge 等等。
- jest：Jest 是由 Facebook 维护的 JavaScript 测试框架。
- coveralls：用于获取 Coveralls.io 的覆盖率报告，并在 README 文件中添加一个不错的覆盖率按钮。
- vuepress：Vue 驱动的静态网站生成器，它用于生成 BetterScroll 2.0 的文档。

因为本章的重点不在工程化，所以上面阿宝哥只是简单罗列了 BetterScroll 在工程化方面使用的开源库。如果你对 BetterScroll 项目也感兴趣的话，可以看看项目中的 `package.json` 文件，并重点看一下项目中 **npm scripts** 的配置。当然 BetterScroll 项目还有很多值得学习的地方，剩下的就等大家去发掘吧，欢迎感兴趣的小伙伴跟阿宝哥一起交流与讨论。

六、参考资源

- BetterScroll 2.0 文档

第十章 TypeScript 进阶之控制反转和依赖注入

本章阿宝哥将从六个方面入手，全方位带你一起探索面向对象编程中 **IoC（控制反转）和 DI（依赖注入）** 的设计思想。阅读完本章，你将了解以下内容：

- IoC 是什么、IoC 能解决什么问题；
- IoC 与 DI 之间的关系、未使用 DI 框架和使用 DI 框架之间的区别；
- DI 在 AngularJS/Angular 和 NestJS 中的应用；
- 了解如何使用 TypeScript 实现一个 IoC 容器，并了解 **装饰器、反射** 的相关知识。

一、背景概述

在介绍什么是 IoC 容器之前，阿宝哥来举一个日常工作中很常见的场景，即创建指定类的实例。最简单的情形是该类没有依赖其他类，但现实往往是残酷的，我们在创建某个类的实例时，需要依赖不同类对应的实例。为了让小伙伴们能够更好地理解上述的内容，阿宝哥来举一个例子。

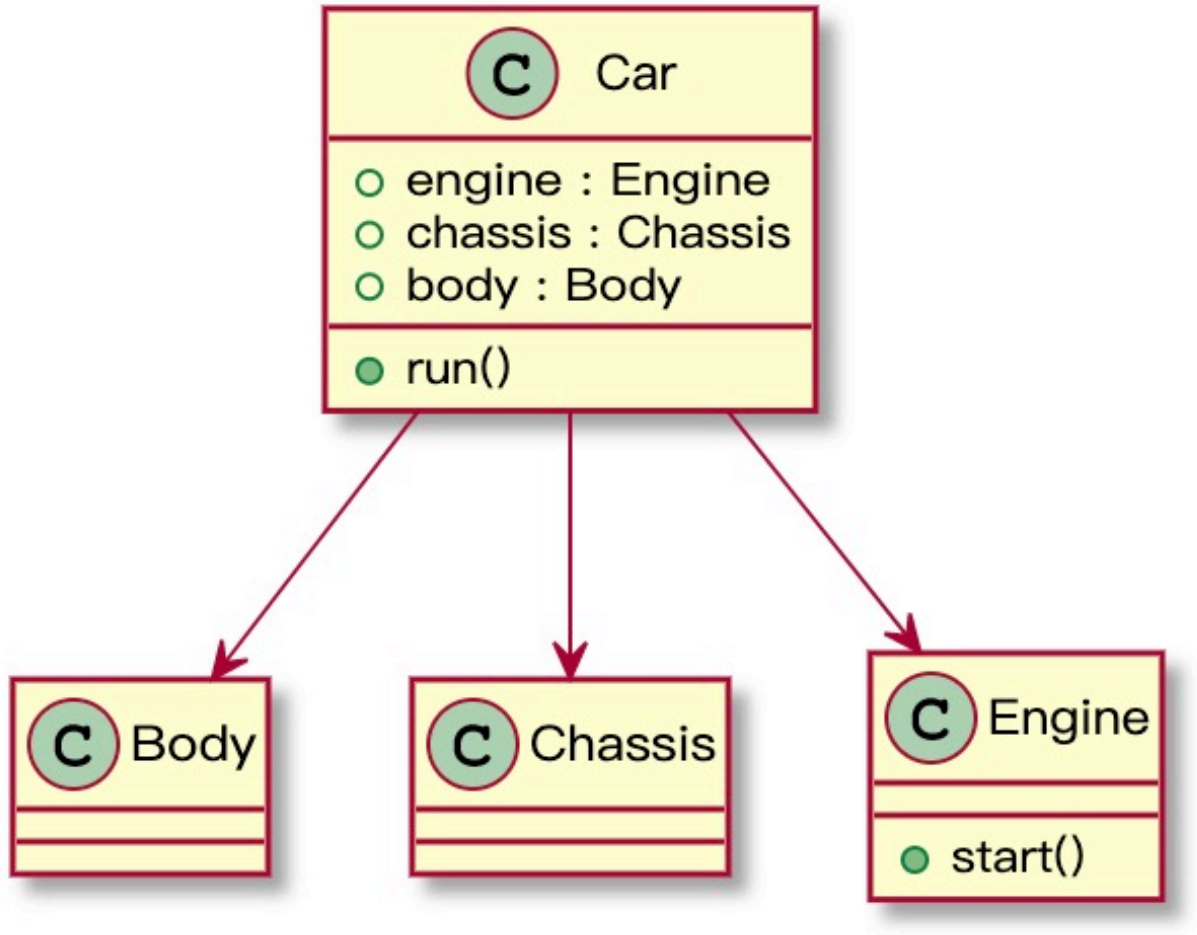
一辆小汽车 🚗 通常由 **发动机、底盘、车身和电气设备** 四大部分组成。汽车电气设备的内部构造很复杂，简单起见，我们只考虑三个部分：发动机、底盘和车身。

（图片来源：https://www.newkidscar.com/vehicle-construction/car-structure/）

在现实生活中，要造辆车还是很困难的。而在软件的世界中，这可难不倒我们。👇是阿宝哥要造的车子，有木有很酷。

（图片来源：https://pixabay.com/zh/illustrations/car-sports-car-racing-car-speed-49278/）

在开始造车前，我们得先看一下 “图纸”：



看完上面的 “图纸”，我们马上来开启造车之旅。第一步我们先来定义车身类：

1.定义车身类

```
export default class Body { }
```

2.定义底盘类

```
export default class Chassis { }
```

3.定义引擎类

```
export default class Engine {
  start() {
    console.log("引擎发动了");
  }
}
```

4.定义汽车类

```
import Engine from './engine';
import Chassis from './chassis';
import Body from './body';

export default class Car {
    engine: Engine;
    chassis: Chassis;
    body: Body;

    constructor() {
      this.engine = new Engine();
      this.body = new Body();
      this.chassis = new Chassis();
    }

    run() {
      this.engine.start();
    }
}
```

一切已准备就绪，我们马上来造一辆车：

```
const car = new Car(); // 阿宝哥造辆新车
car.run(); // 控制台输出：引擎发动了
```

现在虽然车已经可以启动了，但却存在以下问题：

- 问题一：在造车的时候，你不能选择配置。比如你想更换汽车引擎的话，按照目前的方案，是实现不了的。
- 问题二：在汽车类内部，你需要在构造函数中手动去创建汽车的各个部件。

为了解决第一个问题，提供更灵活的方案，我们可以重构一下已定义的汽车类，具体如下：

```
export default class Car {
    body: Body;
```

```
    engine: Engine;
    chassis: Chassis;

    constructor(engine, body, chassis) {
      this.engine = engine;
      this.body = body;
      this.chassis = chassis;
    }

    run() {
      this.engine.start();
    }
}
```

重构完汽车类，我们来重新造辆新车：

```
const engine = new NewEngine();
const body = new Body();
const chassis = new Chassis();

const newCar = new Car(engine, body, chassis);
newCar.run();
```

此时我们已经解决了上面提到的第一个问题，要解决第二个问题我们要来了解一下 IoC（控制反转）的概念。

二、IoC 是什么

IoC（Inversion of Control），即“控制反转”。在开发中， IoC 意味着你设计好的对象交给容器控制，而不是使用传统的方式，在对象内部直接控制。

如何理解好 IoC 呢？理解好 IoC 的关键是要明确 **“谁控制谁，控制什么，为何是反转，哪些方面反转了”**，我们来深入分析一下。

- 谁控制谁，控制什么：在传统的程序设计中，我们直接在对象内部通过 `new` 的方式创建对象，是程序主动创建依赖对象； **而 IoC 是有专门一个容器来创建这些对象，即由 IoC 容器控制对象的创建**；

  谁控制谁？当然是 IoC 容器控制了对象；控制什么？主要是控制外部资源（依赖对象）获取。

- 为何是反转了，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在程序中主动控制去获取依赖对象，也就是正转； **而反转则是由容器来帮忙创建及注入依赖对象**；

  为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转了；哪些方面反转了？依赖对象的获取被反转了。

三、IoC 能做什么

IoC 不是一种技术，只是一种思想，是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。

传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试； **有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器注入组合对象，所以对象之间是松散耦合。** 这样也便于测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实 IoC 对编程带来的最大改变不是从代码上，而是思想上，发生了 “主从换位” 的变化。应用程序本来是老大，要获取什么资源都是主动出击，但在 IoC 思想中，应用程序就变成被动了，被动的等待 IoC 容器来创建并注入它所需的资源了。

四、IoC 与 DI 之间的关系

对于控制反转来说，其中最常见的方式叫做 **依赖注入**，简称为 DI（Dependency Injection）。

组件之间的依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。 **依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。**

通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解 DI 的关键是 **“谁依赖了谁，为什么需要依赖，谁注入了谁，注入了什么”**：

- 谁依赖了谁：当然是应用程序依赖 IoC 容器；
- 为什么需要依赖：应用程序需要 IoC 容器来提供对象需要的外部资源（包括对象、资源、常量数据）；
- 谁注入谁：很明显是 IoC 容器注入应用程序依赖的对象；
- 注入了什么：注入某个对象所需的外部资源（包括对象、资源、常量数据）。

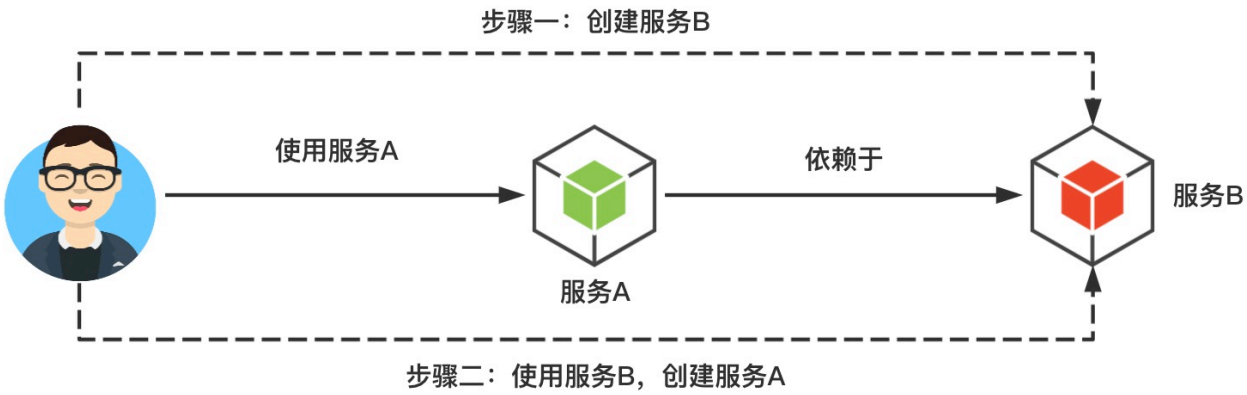
那么 IoC 和 DI 有什么关系？其实它们是同一个概念的不同角度描述，由于控制反转的概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护依赖关系），所以 2004 年大师级人物 Martin Fowler 又给出了一个新的名字：**“依赖注入”，相对 IoC 而言，“依赖注入” 明确描述了被注入对象依赖 IoC 容器配置依赖对象**。

总的来说， 控制反转（Inversion of Control）是说创建对象的控制权发生转移，以前创建对象的主动权和创建时机由应用程序把控，而现在这种权利转交给 IoC 容器，它就是一个专门用来创建对象的工厂，你需要什么对象，它就给你什么对象。 **有了 IoC 容器，依赖关系就改变了，原先的依赖关系就没了，它们都依赖 IoC 容器了，通过 IoC 容器来建立它们之间的关系**。

前面介绍了那么多的概念，现在我们来看一下未使用依赖注入框架和使用依赖注入框架之间有什么明显的区别。

4.1 未使用依赖注入框架

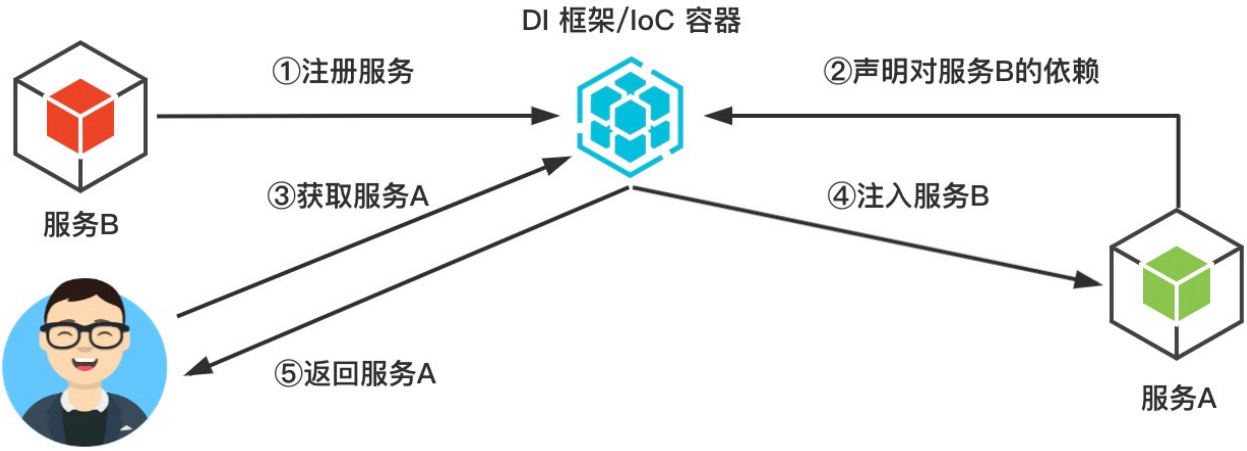
假设我们的服务 A 依赖于服务 B，即要使用服务 A 前，我们需要先创建服务 B。具体的流程如下图所示：



从上图可知，未使用依赖注入框架时，服务的使用者需要关心服务本身和其依赖的对象是如何创建的，且需要手动维护依赖关系。若服务本身需要依赖多个对象，这样就会增加使用难度和后期的维护成本。对于上述的问题，我们可以考虑引入依赖注入框架。下面我们来看一下引入依赖注入框架，整体流程会发生什么变化。

4.2 使用依赖注入框架

使用依赖注入框架之后，系统中的服务会统一注册到 IoC 容器中，如果服务有依赖其他服务时，也需要对依赖进行声明。当用户需要使用特定的服务时，IoC 容器会负责该服务及其依赖对象的创建与管理工作。具体的流程如下图所示：



到这里我们已经介绍了 IoC 与 DI 的概念及特点，接下来我们来介绍 DI 的应用。

五、DI 的应用

DI 在前端和服务端都有相应的应用，比如在前端领域的代表是 AngularJS 和 Angular，而在服务端领域是 Node.js 生态中比较出名的 NestJS。接下来阿宝哥将简单介绍一下 DI 在 AngularJS/Angular 和 NestJS 中的应用。

5.1 DI 在 AngularJS 中的应用

在 AngularJS 中，依赖注入是其核心的特性之一。在 AngularJS 中声明依赖项有 3 种方式：

```
// 方式一：  使用 $inject annotation 方式
let fn = function (a, b) {};
fn.$inject = ['a', 'b'];

// 方式二：  使用 array-style annotations 方式
let fn = ['a', 'b', function (a, b) {}];

// 方式三：  使用隐式声明方式
let fn = function (a, b) {}; // 不推荐
```

对于以上的代码，相信使用过 AngularJS 的小伙们都不会陌生。作为 AngularJS 核心功能特性的 DI 还是蛮强大的，但随着 AngularJS 的普及和应用的复杂度不断提高，AngularJS DI 系统的问题就暴露出来了。

这里阿宝哥简单介绍一下 AngularJS DI 系统存在的几个问题：

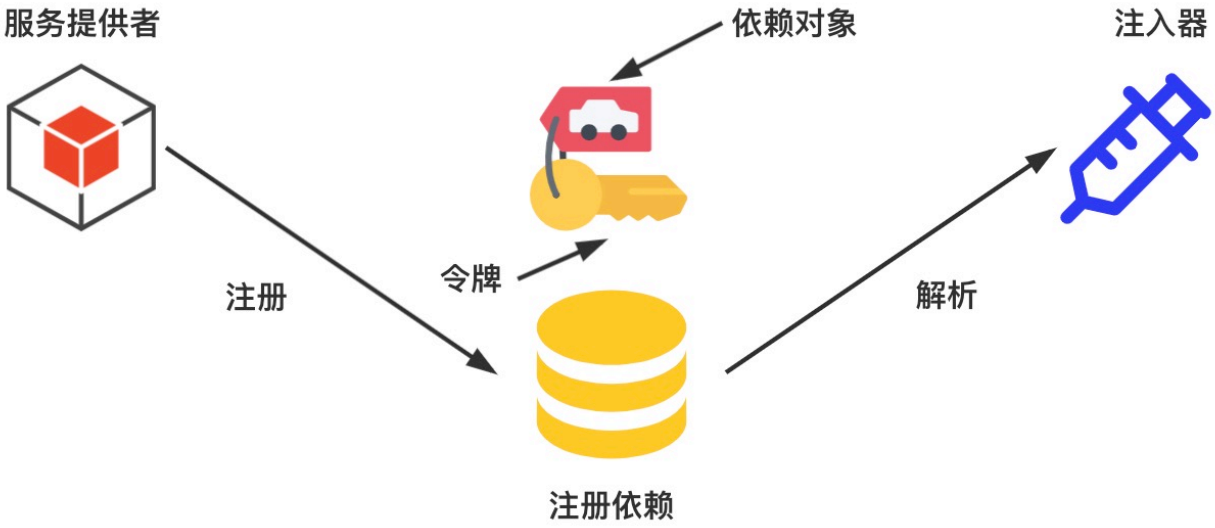
- 内部缓存： AngularJS 应用程序中所有的依赖项都是单例，我们不能控制是否使用新的实例；
- 命名空间冲突： 在系统中我们使用字符串来标识服务的名称，假设我们在项目中已有一个 CarService，然而第三方库中也引入了同样的服务，这样的话就容易出现混淆。

由于 AngularJS DI 存在以上的问题，所以在后续的 Angular 重新设计了新的 DI 系统。

5.2 DI 在 Angular 中的应用

以前面汽车的例子为例，我们可以把汽车、发动机、底盘和车身这些认为是一种 “服务”，所以它们会以服务提供者的形式注册到 DI 系统中。为了能区分不同服务，我们需要使用不同的令牌（Token）来标识它们。接着我们会基于已注册的服务提供者创建注入器对象。

之后，当我们需要获取指定服务时，我们就可以通过该服务对应的令牌，从注入器对象中获取令牌对应的依赖对象。上述的流程的具体如下图所示：



好的，了解完上述的流程。下面我们来看一下如何使用 Angular 内置的 DI 系统来 “造车”。

5.2.1 car.ts

```
// car.ts
import { Injectable, ReflectiveInjector } from '@angular/core';

// 配置Provider
@Injectable({
  providedIn: 'root',
})
export class Body {}

@Injectable({
  providedIn: 'root',
})
export class Chassis {}

@Injectable({
  providedIn: 'root',
})
export class Engine {
  start() {
    console.log('引擎发动了');
  }
}

@Injectable()
export default class Car {
  // 使用构造注入方式注入依赖对象
  constructor(
    private engine: Engine,
    private body: Body,
    private chassis: Chassis
  ) {}

  run() {
    this.engine.start();
  }
}

const injector = ReflectiveInjector.resolveAndCreate([
  Car,
  Engine,
  Chassis,
  Body,
]);

const car = injector.get(Car);
car.run();
```

在以上代码中我们调用 ReflectiveInjector 对象的 `resolveAndCreate` 方法手动创建注入器，然后根据车辆对应的 `Token` 来获取对应的依赖对象。通过观察上述代码，你可以发现，我们已经不需要手动地管理和维护依赖对象了，这些 “脏活”、“累活” 已经交给注入器来处理了。

此外，如果要能正常获取汽车对象，我们还需要在 `app.module.ts` 文件中声明 Car 对应 Provider，具体如下所示：

5.2.2 app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import Car, { Body, Chassis, Engine } from './car';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [{ provide: Car, deps: [Engine, Body, Chassis] }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

5.3 DI 在 NestJS 中的应用

NestJS 是构建高效，可扩展的 Node.js Web 应用程序的框架。 它使用现代的 JavaScript 或 TypeScript（保留与纯 JavaScript 的兼容性），并结合 OOP（面向对象编程），FP（函数式编程）和 FRP（函数响应式编程）的元素。

在底层，Nest 使用了 Express，但也提供了与其他各种库的兼容，例如 Fastify，可以方便地使用各种可用的第三方插件。

近几年，由于 Node.js，JavaScript 已经成为 Web 前端和后端应用程序的「通用语言」，从而产生了像 Angular、React、Vue 等令人耳目一新的项目，这些项目提高了开发人员的生产力，使得可以快速构建可测试的且可扩展的前端应用程序。 然而，在服务器端，虽然有很多优秀的库、helper 和 Node 工具，但是它们都没有有效地解决主要问题 —— 架构。

NestJS 旨在提供一个开箱即用的应用程序体系结构，允许轻松创建高度可测试，可扩展，松散耦合且易于维护的应用程序。 在 NestJS 中也为我们开发者提供了依赖注入的功能，这里我们以官网的示例来演示一下依赖注入的功能。

5.3.1 app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

5.3.2 app.controller.ts

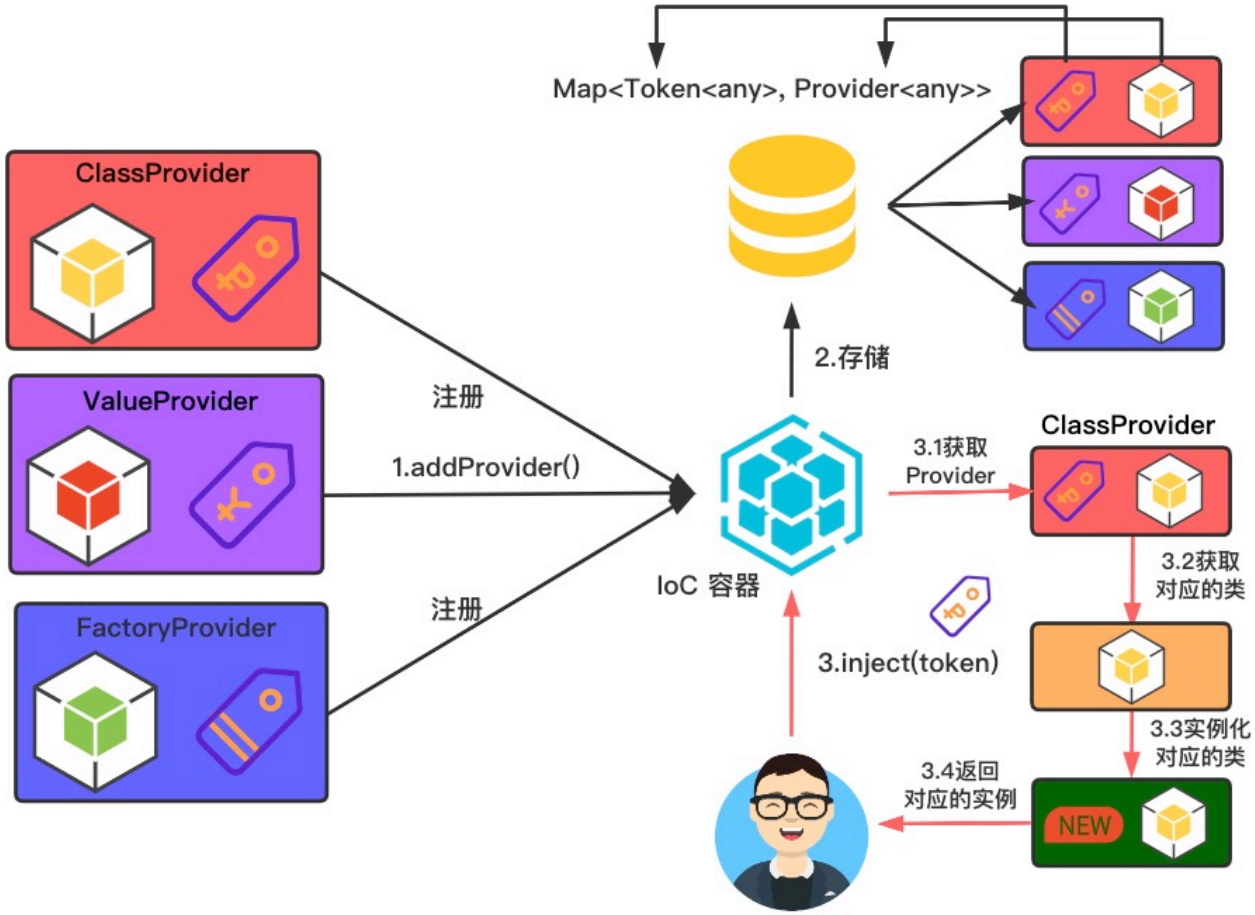
```
import { Get, Controller, Render } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  @Render('index')
  render() {
    const message = this.appService.getHello();
    return { message };
  }
}
```

在 AppController 中，我们通过构造注入的方式注入了 AppService 对象，当用户访问首页的时候，我们会调用 AppService 对象的 `getHello` 方法来获取 `'Hello World!'` 消息，并把消息返回给用户。当然为了保证依赖注入可以正常工作，我们还需要在 AppModule 中声明 providers 和 controllers，具体操作如下：

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

其实 DI 并不是 AngularJS/Angular 和 NestJS 所特有的，如果你想在其他项目中使用 DI/IoC 的功能特性，阿宝哥推荐你使用 InversifyJS，它是一个可用于 JavaScript 和 Node.js 应用，功能强大、轻量的 IoC 容器。

对 InversifyJS 感兴趣的小伙伴可以自行了解一下，阿宝哥就不继续展开介绍了。接下来，我们将进入本章的重点，即介绍如何使用 TypeScript 实现一个简单的 IoC 容器，该容器实现的功能如下图所示：



六、手写 IoC 容器

为了让大家能更好地理解 IoC 容器的实现代码，阿宝哥来介绍一些相关的前置知识。

6.1 装饰器

如果你有使用过 Angular 或 NestJS，相信你对以下的代码不会陌生。

```
@Injectable()
export class HttpService {
  constructor(
    private httpClient: HttpClient
  ) {}
}
```

在以上代码中，我们使用了 `Injectable` 装饰器。该装饰器用于表示此类可以自动注入其依赖项。其中 `@Injectable()` 中的 `@` 符号属于语法糖。

装饰器是一个包装类，函数或方法并为其添加行为的函数。这对于定义与对象关联的元数据很有用。装饰器有以下四种分类：

- 类装饰器（Class decorators）
- 属性装饰器（Property decorators）

- 方法装饰器（Method decorators）
- 参数装饰器（Parameter decorators）

前面示例中使用的 `@Injectable()` 装饰器，属于类装饰器。在该类装饰器修饰的 `HttpService` 类中，我们通过构造注入的方式注入了用于处理 HTTP 请求的 `HttpClient` 依赖对象。

6.2 反射

```
@Injectable()
export class HttpService {
  constructor(
    private httpClient: HttpClient
  ) {}
}
```

以上代码若设置编译的目标为 ES5，则会生成以下代码：

```
// 忽略__decorate函数等代码
var __metadata = (this && this.__metadata) || function (k, v) {
    if (typeof Reflect === "object" && typeof Reflect.metadata === "function")
      return Reflect.metadata(k, v);
};

var HttpService = /** @class */ (function () {
    function HttpService(httpClient) {
      this.httpClient = httpClient;
    }
    var _a;
    HttpService = __decorate([
        Injectable(),
        __metadata("design:paramtypes", [typeof (_a = typeof HttpClient !==
"undefined" && HttpClient)
            === "function" ? _a : Object])
    ], HttpService);
    return HttpService;
}());
```

通过观察上述代码，你会发现 `HttpService` 构造函数中 `httpClient` 参数的类型被擦除了，这是因为 JavaScript 是弱类型语言。那么如何在运行时，保证注入正确类型的依赖对象呢？这里 TypeScript 使用 reflect-metadata 这个第三方库来存储额外的类型信息。

reflect-metadata 这个库提供了很多 API 用于操作元信息，这里我们只简单介绍几个常用的 API：

```
// define metadata on an object or property
Reflect.defineMetadata(metadataKey, metadataValue, target);
Reflect.defineMetadata(metadataKey, metadataValue, target, propertyKey);
```

```
// check for presence of a metadata key on the prototype chain of an object or
property
let result = Reflect.hasMetadata(metadataKey, target);
let result = Reflect.hasMetadata(metadataKey, target, propertyKey);

// get metadata value of a metadata key on the prototype chain of an object or
property
let result = Reflect.getMetadata(metadataKey, target);
let result = Reflect.getMetadata(metadataKey, target, propertyKey);

// delete metadata from an object or property
let result = Reflect.deleteMetadata(metadataKey, target);
let result = Reflect.deleteMetadata(metadataKey, target, propertyKey);

// apply metadata via a decorator to a constructor
@Reflect.metadata(metadataKey, metadataValue)
class C {
  // apply metadata via a decorator to a method (property)
  @Reflect.metadata(metadataKey, metadataValue)
  method() {
  }
}
```

对于上述的 API 只需简单了解一下即可。在后续的内容中，我们将介绍具体如何使用。这里我们需要注意以下两个问题：

- 对于类或函数，我们需要使用装饰器来修饰它们，这样才能保存元数据。
- 只有类、枚举或原始数据类型能被记录。接口和联合类型作为 “对象” 出现。这是因为这些类型在编译后完全消失，而类却一直存在。

6.3 定义 Token 和 Provider

了解完装饰器与反射相关的基础知识，接下来我们来开始实现 IoC 容器。我们的 IoC 容器将使用两个主要的概念：令牌（Token）和提供者（Provider）。令牌是 IoC 容器所要创建对象的标识符，而提供者用于描述如何创建这些对象。

IoC 容器最小的公共接口如下所示：

```
export class Container {
  addProvider<T>(provider: Provider<T>) {} // TODO
  inject<T>(type: Token<T>): T {} // TODO
}
```

接下来我们先来定义 Token：

```
// type.ts
interface Type<T> extends Function {
  new (...args: any[]): T;
}

// provider.ts
class InjectionToken {
  constructor(public injectionIdentifier: string) {}
}

type Token<T> = Type<T> | InjectionToken;
```

Token 类型是一个联合类型，既可以是一个函数类型也可以是 InjectionToken 类型。AngularJS 中使用字符串作为 Token，在某些情况下，可能会导致冲突。因此，为了解决这个问题，我们定义了 InjectionToken 类，来避免出现命名冲突问题。

定义完 Token 类型，接下来我们来定义三种不同类型的 Provider：

- ClassProvider：提供一个类，用于创建依赖对象；
- ValueProvider：提供一个已存在的值，作为依赖对象；
- FactoryProvider：提供一个工厂方法，用于创建依赖对象。

```
// provider.ts
export type Factory<T> = () => T;

export interface BaseProvider<T> {
  provide: Token<T>;
}

export interface ClassProvider<T> extends BaseProvider<T> {
  provide: Token<T>;
  useClass: Type<T>;
}

export interface ValueProvider<T> extends BaseProvider<T> {
  provide: Token<T>;
  useValue: T;
}

export interface FactoryProvider<T> extends BaseProvider<T> {
  provide: Token<T>;
  useFactory: Factory<T>;
}

export type Provider<T> =
  | ClassProvider<T>
  | ValueProvider<T>
  | FactoryProvider<T>;
```

为了更方便的区分这三种不同类型的 Provider，我们自定义了三个类型守卫函数：

```
// provider.ts
export function isClassProvider<T>(
  provider: BaseProvider<T>
): provider is ClassProvider<T> {
  return (provider as any).useClass !== undefined;
}

export function isValueProvider<T>(
  provider: BaseProvider<T>
): provider is ValueProvider<T> {
  return (provider as any).useValue !== undefined;
}

export function isFactoryProvider<T>(
  provider: BaseProvider<T>
): provider is FactoryProvider<T> {
  return (provider as any).useFactory !== undefined;
}
```

6.4 定义装饰器

在前面我们已经提过了，对于类或函数，我们需要使用装饰器来修饰它们，这样才能保存元数据。因此，接下来我们来分别创建 **Injectable** 和 **Inject** 装饰器。

6.4.1 Injectable 装饰器

Injectable 装饰器用于表示此类可以自动注入其依赖项，该装饰器属于类装饰器。在 TypeScript 中，类装饰器的声明如下：

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction)
  => TFunction | void;
```

类装饰器顾名思义，就是用来装饰类的。它接收一个参数：`target: TFunction`，表示被装饰的类。下面我们来看一下 Injectable 装饰器的具体实现：

```
// Injectable.ts
import { Type } from "./type";
import "reflect-metadata";

const INJECTABLE_METADATA_KEY = Symbol("INJECTABLE_KEY");

export function Injectable() {
  return function(target: any) {
    Reflect.defineMetadata(INJECTABLE_METADATA_KEY, true, target);
    return target;
  };
}
```

在以上代码中，当调用完 Injectable 函数之后，会返回一个新的函数。在新的函数中，我们使用 reflect-metadata 这个库提供的 defineMetadata API 来保存元信息，其中 defineMetadata API 的使用方式如下所示：

```
// define metadata on an object or property
Reflect.defineMetadata(metadataKey, metadataValue, target);
Reflect.defineMetadata(metadataKey, metadataValue, target, propertyKey);
```

Injectable 类装饰器使用方式也简单，只需要在被装饰类的上方使用 `@Injectable()` 语法糖就可以应用该装饰器：

```
@Injectable()
export class HttpService {
  constructor(
    private httpClient: HttpClient
  ) {}
}
```

在以上示例中，我们注入的是 Type 类型的 HttpClient 对象。但在实际的项目中，往往会比较复杂。除了需要注入 Type 类型的依赖对象之外，我们还可能会注入其他类型的依赖对象，比如我们希望在 HttpService 服务中注入远程服务器的 API 地址。针对这种情形，我们需要使用 Inject 装饰器。

6.4.2 Inject 装饰器

接下来我们来创建 Inject 装饰器，该装饰器属于参数装饰器。在 TypeScript 中，参数装饰器的声明如下：

```
declare type ParameterDecorator = (target: Object,
  propertyKey: string | symbol, parameterIndex: number ) => void
```

参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数：

- target: Object —— 被装饰的类；
- propertyKey: string | symbol —— 方法名；

- parameterIndex: number —— 方法中参数的索引值。

下面我们来看一下 Inject 装饰器的具体实现：

```
// Inject.ts
import { Token } from './provider';
import 'reflect-metadata';

const INJECT_METADATA_KEY = Symbol('INJECT_KEY');

export function Inject(token: Token<any>) {
  return function(target: any, _: string | symbol, index: number) {
    Reflect.defineMetadata(INJECT_METADATA_KEY, token, target, `index-
${index}`);
    return target;
  };
}
```

在以上代码中，当调用完 Inject 函数之后，会返回一个新的函数。在新的函数中，我们使用 reflect-metadata 这个库提供的 defineMetadata API 来保存参数相关的元信息。这里是保存 index 索引信息和 Token 信息。

定义完 Inject 装饰器，我们就可以利用它来注入我们前面所提到的远程服务器的 API 地址，具体的使用方式如下：

```
const API_URL = new InjectionToken('apiUrl');

@Injectable()
export class HttpService {
  constructor(
    private httpClient: HttpClient,
    @Inject(API_URL) private apiUrl: string
  ) {}
}
```

6.5 实现 IoC 容器

目前为止，我们已经定义了 Token、Provider、Injectable 和 Inject 装饰器。接下来我们来实现前面所提到的 IoC 容器的 API：

```
export class Container {
  addProvider<T>(provider: Provider<T>) {} // TODO
  inject<T>(type: Token<T>): T {} // TODO
}
```

6.5.1 实现 addProvider 方法

addProvider() 方法的实现很简单，我们使用 Map 来存储 Token 与 Provider 之间的关系：

```
export class Container {
  private providers = new Map<Token<any>, Provider<any>>();

  addProvider<T>(provider: Provider<T>) {
    this.assertInjectableIfClassProvider(provider);
    this.providers.set(provider.provide, provider);
  }
}
```

在 addProvider() 方法内部除了把 Token 与 Provider 的对应信息保存到 providers 对象中之外，我们定义了一个 assertInjectableIfClassProvider 方法，用于确保添加的 ClassProvider 是可注入的。该方法的具体实现如下：

```
private assertInjectableIfClassProvider<T>(provider: Provider<T>) {
  if (isClassProvider(provider) && !isInjectable(provider.useClass)) {
    throw new Error(
        `Cannot provide ${this.getTokenName(
          provider.provide
     )} using class ${this.getTokenName(
          provider.useClass
     )}, ${this.getTokenName(provider.useClass)} isn't injectable`
   );
  }
}
```

在 assertInjectableIfClassProvider 方法体中，我们使用了前面已经介绍的 `isClassProvider` 类型守卫函数来判断是否为 ClassProvider，如果是的话，会判断该 ClassProvider 是否为可注入的，具体使用的是 isInjectable 函数，该函数的定义如下：

```
export function isInjectable<T>(target: Type<T>) {
  return Reflect.getMetadata(INJECTABLE_METADATA_KEY, target) === true;
}
```

在 isInjectable 函数中，我们使用 reflect-metadata 这个库提供的 getMetadata API 来获取保存在类中的元信息。为了更好地理解以上代码，我们来回顾一下前面 Injectable 装饰器：

```
const INJECTABLE_METADATA_KEY = Symbol("INJECTABLE_KEY");

export function Injectable() {
  return function(target: any) {
    Reflect.defineMetadata(INJECTABLE_METADATA_KEY, true, target);
    return target;
  };
}
```

如果添加的 Provider 是 ClassProvider，但 Provider 对应的类是不可注入的，则会抛出异常。为了让异常消息更加友好，也更加直观。我们定义了一个 `getTokenName` 方法来获取 Token 对应的名称：

```
private getTokenName<T>(token: Token<T>) {
  return token instanceof InjectionToken
    ? token.injectionIdentifier
    : token.name;
}
```

现在我们已经实现了 Container 类的 `addProvider` 方法，这时我们就可以使用它来添加三种不同类型的 Provider：

```
const container = new Container();
const input = { x: 200 };

class BasicClass {}
// 注册ClassProvider
container.addProvider({ provide: BasicClass, useClass:  BasicClass});
// 注册ValueProvider
container.addProvider({ provide: BasicClass, useValue: input });
// 注册FactoryProvider
container.addProvider({ provide: BasicClass, useFactory: () => input });
```

需要注意的是，以上示例中注册三种不同类型的 Provider 使用的是同一个 Token 仅是为了演示而已。下面我们来实现 Container 类中核心的 inject 方法。

6.5.2 实现 inject 方法

在看 inject 方法的具体实现之前，我们先来看一下该方法所实现的功能：

```
const container = new Container();
const input = { x: 200 };

container.addProvider({ provide: BasicClass, useValue: input });
const output = container.inject(BasicClass);
expect(input).toBe(output); // true
```

观察以上的测试用例可知，Container 类中 inject 方法所实现的功能就是根据 Token 获取与之对应的对象。在前面实现的 addProvider 方法中，我们把 Token 和该 Token 对应的 Provider 保存在 providers Map 对象中。所以在 inject 方法中，我们可以先从 providers 对象中获取该 Token 对应的 Provider 对象，然后在根据不同类型的 Provider 来获取其对应的对象。

好的，下面我们来看一下 inject 方法的具体实现：

```
inject<T>(type: Token<T>): T {
  let provider = this.providers.get(type);
  // 处理使用Injectable装饰器修饰的类
  if (provider === undefined && !(type instanceof InjectionToken)) {
    provider = { provide: type, useClass: type };
    this.assertInjectableIfClassProvider(provider);
  }
  return this.injectWithProvider(type, provider);
}
```

在以上代码中，除了处理正常的流程之外。我们还处理一个特殊的场景，即没有使用 `addProvider` 方法注册 Provider，而是使用 Injectable 装饰器来装饰某个类。对于这个特殊场景，我们会根据传入的 type 参数来创建一个 provider 对象，然后进一步调用 `injectWithProvider` 方法来创建对象，该方法的具体实现如下：

```
private injectWithProvider<T>(type: Token<T>, provider?: Provider<T>): T {
  if (provider === undefined) {
    throw new Error(`No provider for type ${this.getTokenName(type)}`);
  }
  if (isClassProvider(provider)) {
    return this.injectClass(provider as ClassProvider<T>);
  } else if (isValueProvider(provider)) {
    return this.injectValue(provider as ValueProvider<T>);
  } else {
    return this.injectFactory(provider as FactoryProvider<T>);
  }
 }
```

在 `injectWithProvider` 方法内部，我们会使用前面定义的用于区分三种不同类型 Provider 的类型守卫函数来处理不同的 Provider。这里我们先来看一下最简单 ValueProvider，当发现注入的是 ValueProvider 类型时，则会调用 `injectValue` 方法来获取其对应的对象：

```
// { provide: API_URL, useValue: 'https://www.semlinker.com/' }
private injectValue<T>(valueProvider: ValueProvider<T>): T {
  return valueProvider.useValue;
}
```

接着我们来看如何处理 FactoryProvider 类型的 Provider，如果发现是 FactoryProvider 类型时，则会调用 `injectFactory` 方法来获取其对应的对象，该方法的实现也很简单：

```
// const input = { x: 200 };
// container.addProvider({ provide: BasicClass, useFactory: () => input });
private injectFactory<T>(valueProvider: FactoryProvider<T>): T {
  return valueProvider.useFactory();
}
```

最后我们来分析一下如何处理 ClassProvider，对于 ClassProvider 类说，通过 Provider 对象的 useClass 属性，我们就可以直接获取到类对应的构造函数。最简单的情形是该类没有依赖其他对象，但在大多数场景下，即将实例化的服务类是会依赖其他的对象的。所以在实例化服务类前，我们需要构造其依赖的对象。

那么现在问题来了，怎么获取类所依赖的对象呢？我们先来分析一下以下代码：

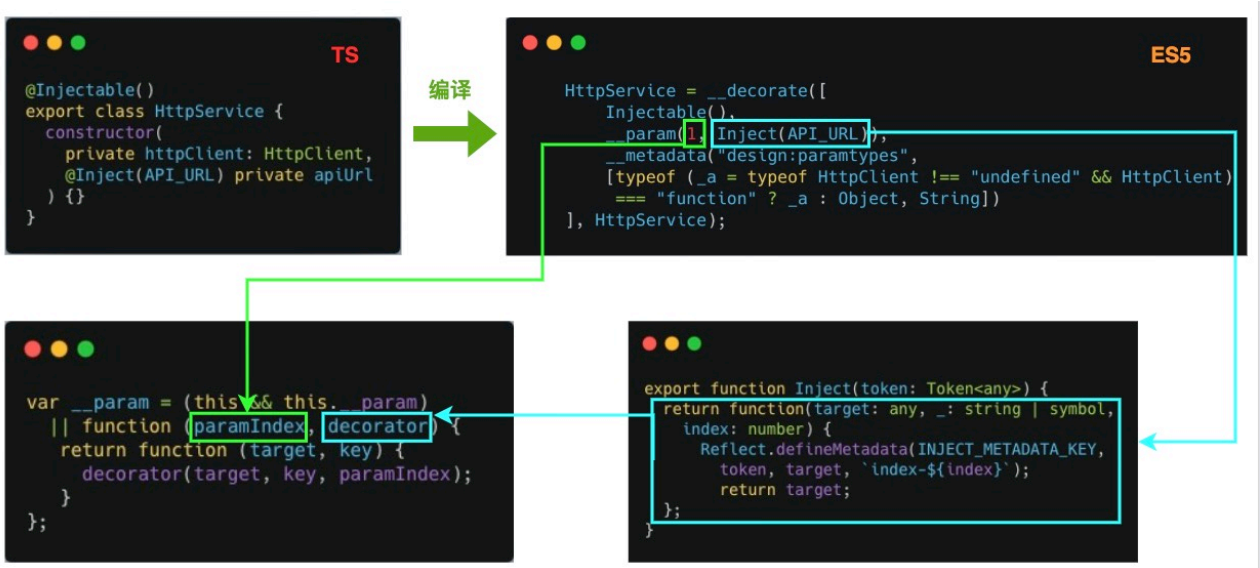
```
const API_URL = new InjectionToken('apiUrl');

@Injectable()
export class HttpService {
  constructor(
    private httpClient: HttpClient,
    @Inject(API_URL) private apiUrl: string
  ) {}
}
```

以上代码若设置编译的目标为 ES5，则会生成以下代码：

```
// 已省略__decorate函数的定义
var __metadata = (this && this.__metadata) || function (k, v) {
    if (typeof Reflect === "object" && typeof Reflect.metadata === "function")
return Reflect.metadata(k, v);
};

var __param = (this && this.__param) || function (paramIndex, decorator) {
    return function (target, key) { decorator(target, key, paramIndex); }
};

var HttpService = /** @class */ (function () {
    function HttpService(httpClient, apiUrl) {
        this.httpClient = httpClient;
        this.apiUrl = apiUrl;
    }
    var _a;
    HttpService = __decorate([
        Injectable(),
        __param(1, Inject(API_URL)),
        __metadata("design:paramtypes", [typeof (_a = typeof HttpClient !==
"undefined" && HttpClient)
          === "function" ? _a : Object, String])
    ], HttpService);
```

```
    return HttpService;
}());
```

观察以上的代码会不会觉得有点晕？不要着急，阿宝哥会逐一分析 HttpService 中的两个参数。首先我们先来分析 `apiUrl` 参数：



在图中我们可以很清楚地看到，`API_URL` 对应的 Token 最终会通过 Reflect.defineMetadata API 进行保存，所使用的 Key 是 `Symbol('INJECT_KEY')`。而对于另一个参数即 httpClient，它使用的 Key 是 `"design:paramtypes"`，它用于修饰目标对象方法的参数类型。

除了 `"design:paramtypes"` 之外，还有其他的 `metadataKey`，比如 `design:type` 和 `design:returntype`，它们分别用于修饰目标对象的类型和修饰目标对象方法返回值的类型。

由上图可知，HttpService 构造函数的参数类型最终会使用 `Reflect.metadata` API 进行存储。了解完上述的知识，接下来我们来定义一个 `getInjectedParams` 方法，用于获取类构造函数中声明的依赖对象，该方法的具体实现如下：

```
type InjectableParam = Type<any>;
const REFLECT_PARAMS = "design:paramtypes";
```

```
private getInjectedParams<T>(target: Type<T>) {
  // 获取参数的类型
  const argTypes = Reflect.getMetadata(REFLECT_PARAMS, target) as (
      | InjectableParam
      | undefined
  )[];
  if (argTypes === undefined) {
      return [];
  }
  return argTypes.map((argType, index) => {
    // The reflect-metadata API fails on circular dependencies, and will return
undefined
    // for the argument instead.
    if (argType === undefined) {
      throw new Error(
        `Injection error. Recursive dependency detected in constructor for type
${target.name}
           with parameter at index ${index}`
      );
    }
    const overrideToken = getInjectionToken(target, index);
    const actualToken = overrideToken === undefined ? argType : overrideToken;
    let provider = this.providers.get(actualToken);
    return this.injectWithProvider(actualToken, provider);
  });
}
```

因为我们的 Token 的类型是 `Type<T> | InjectionToken` 联合类型，所以在 `getInjectedParams` 方法中我们也要考虑 InjectionToken 的情形，因此我们定义了一个 `getInjectionToken` 方法来获取使用 `@Inject` 装饰器注册的 Token，该方法的实现很简单：

```
export function getInjectionToken(target: any, index: number) {
  return Reflect.getMetadata(INJECT_METADATA_KEY, target, `index-${index}`) as
Token<any> | undefined;
}
```

现在我们已经可以获取类构造函数中所依赖的对象，基于前面定义的 `getInjectedParams` 方法，我们就来定义一个 `injectClass` 方法，用来实例化 ClassProvider 所注册的类。

```
// { provide: HttpClient, useClass: HttpClient }
private injectClass<T>(classProvider: ClassProvider<T>): T {
  const target = classProvider.useClass;
  const params = this.getInjectedParams(target);
  return Reflect.construct(target, params);
}
```

这时 IoC 容器中定义的两个方法都已经实现了，我们来看一下 IoC 容器的完整代码：

```
// container.ts
type InjectableParam = Type<any>;

const REFLECT_PARAMS = "design:paramtypes";

export class Container {
  private providers = new Map<Token<any>, Provider<any>>();

  addProvider<T>(provider: Provider<T>) {
    this.assertInjectableIfClassProvider(provider);
    this.providers.set(provider.provide, provider);
  }

  inject<T>(type: Token<T>): T {
    let provider = this.providers.get(type);
    if (provider === undefined && !(type instanceof InjectionToken)) {
      provider = { provide: type, useClass: type };
      this.assertInjectableIfClassProvider(provider);
    }
    return this.injectWithProvider(type, provider);
  }

  private injectWithProvider<T>(type: Token<T>, provider?: Provider<T>): T {
    if (provider === undefined) {
      throw new Error(`No provider for type ${this.getTokenName(type)}`);
    }
    if (isClassProvider(provider)) {
      return this.injectClass(provider as ClassProvider<T>);
    } else if (isValueProvider(provider)) {
      return this.injectValue(provider as ValueProvider<T>);
    } else {
      // Factory provider by process of elimination
      return this.injectFactory(provider as FactoryProvider<T>);
    }
  }

  private assertInjectableIfClassProvider<T>(provider: Provider<T>) {
    if (isClassProvider(provider) && !isInjectable(provider.useClass)) {
      throw new Error(
        `Cannot provide ${this.getTokenName(
          provider.provide
        )} using class ${this.getTokenName(
          provider.useClass
        )}, ${this.getTokenName(provider.useClass)} isn't injectable`
      );
    }
  }
```

```
  private injectClass<T>(classProvider: ClassProvider<T>): T {
    const target = classProvider.useClass;
    const params = this.getInjectedParams(target);
    return Reflect.construct(target, params);
  }

  private injectValue<T>(valueProvider: ValueProvider<T>): T {
    return valueProvider.useValue;
  }

  private injectFactory<T>(valueProvider: FactoryProvider<T>): T {
    return valueProvider.useFactory();
  }

  private getInjectedParams<T>(target: Type<T>) {
    const argTypes = Reflect.getMetadata(REFLECT_PARAMS, target) as (
      | InjectableParam
      | undefined
    )[];
    if (argTypes === undefined) {
      return [];
    }
    return argTypes.map((argType, index) => {
      // The reflect-metadata API fails on circular dependencies, and will
return undefined
      // for the argument instead.
      if (argType === undefined) {
        throw new Error(
          `Injection error. Recursive dependency detected in constructor for
type ${target.name}
              with parameter at index ${index}`
        );
      }
      const overrideToken = getInjectionToken(target, index);
      const actualToken = overrideToken === undefined ? argType :
overrideToken;
      let provider = this.providers.get(actualToken);
      return this.injectWithProvider(actualToken, provider);
    });
  }

  private getTokenName<T>(token: Token<T>) {
    return token instanceof InjectionToken
      ? token.injectionIdentifier
      : token.name;
  }
}
```

最后我们来简单测试一下我们前面开发的 IoC 容器，具体的测试代码如下所示：

```
// container.test.ts
import { Container } from "./container";
import { Injectable } from "./injectable";
import { Inject } from "./inject";
import { InjectionToken } from "./provider";

const API_URL = new InjectionToken("apiUrl");

@Injectable()
class HttpClient {}

@Injectable()
class HttpService {
  constructor(
    private httpClient: HttpClient,
    @Inject(API_URL) private apiUrl: string
  ) {}
}

const container = new Container();

container.addProvider({
  provide: API_URL,
  useValue: "https://www.semlinker.com/",
});

container.addProvider({ provide: HttpClient, useClass: HttpClient });
container.addProvider({ provide: HttpService, useClass: HttpService });

const httpService = container.inject(HttpService);
console.dir(httpService);
```

以上代码成功运行后，控制台会输出以下结果：

```
HttpService {
  httpClient: HttpClient {},
  apiUrl: 'https://www.semlinker.com/' }
```

很明显该结果正是我们所期望的，这表示我们 IoC 容器已经可以正常工作了。当然在实际项目中，一个成熟的 IoC 容器还要考虑很多东西，如果小伙伴想在项目中使用的话，阿宝哥建议可以考虑使用 InversifyJS 这个库。

七、参考资源

- 维基百科 - 控制反转
- Stackblitz - Car-Demo
- Github - reflect-metadata
- Metadata Proposal - ECMAScript
- typescript-dependency-injection-in-200-loc

第十一章 TypeScript 进阶之装饰 Web 服务器

本章阿宝哥将以 Github 上的 OvernightJS 开源项目为例，来介绍一下如何使用 TypeScript 装饰器来装饰 Express，从而让你的 Express 好用得飞起来。

接下来本章的重心将围绕 **装饰器** 的应用展开，不过在分析装饰器在 OvernightJS 的应用之前，阿宝哥先来简单介绍一下 OvernightJS。

一、OvernightJS 简介

> TypeScript decorators for the ExpressJS Server.

OvernightJS 是一个简单的库，用于为要调用 Express 路由的方法添加 TypeScript 装饰器。此外，该项目还包含了用于管理 json-web-token 和打印日志的包。

1.1 OvernightJS 特性

OvernightJS 并不是为了替代 Express，如果你之前已经掌握了 Express，那你就可以快速地学会它。OvernightJS 为开发者提供了以下特性：

- 使用 `@Controller` 装饰器定义基础路由；
- 提供了把类方法转化为 Express 路由的装饰器（**比如 @Get，@Put，@Post，@Delete**）；
- 提供了用于处理中间件的 `@Middleware` 和 `@ClassMiddleware` 装饰器；
- 提供了用于处理异常的 `@ErrorMiddleware` 装饰器；
- 提供了 `@Wrapper` 和 `@ClassWrapper` 装饰器用于包装函数；
- 通过 `@ChildControllers` 装饰器支持子控制器。

出于篇幅考虑，阿宝哥只介绍了 OvernightJS 与装饰器相关的部分特性。了解完这些特性，我们来快速体验一下 OvernightJS。

1.2 OvernightJS 入门

1.2.1 初始化项目

首先新建一个 `overnight-quickstart` 项目，然后使用 `npm init -y` 命令初始化项目，然后在命令行中输入以下命令来安装项目依赖包：

```
$ npm i @overnightjs/core express -S
```

在 Express 项目中要集成 TypeScript 很简单，只需安装 `typescript` 这个包就可以了。但为了在开发阶段能够在命令行直接运行使用 TypeScript 开发的服务器，我们还需要安装 `ts-node` 这个包。要安装这两个包，我们只需在命令行中输入以下命令：

```
$ npm i typescript ts-node -D
```

1.2.2 为 Node.js 和 Express 安装声明文件

声明文件是预定义的模块，用于告诉 TypeScript 编译器的 JavaScript 值的形状。类型声明通常包含在扩展名为 `.d.ts` 的文件中。这些声明文件可用于所有最初用 JavaScript 而非 TypeScript 编写的库。

幸运的是，我们不需要重头开始为 Node.js 和 Express 定义声明文件，因为在 Github 上有一个名为 DefinitelyTyped 项目已经为我们提供了现成的声明文件。

要安装 Node.js 和 Express 对应的声明文件，我们只需要在命令行执行以下命令就可以了：

```
$ npm i @types/node @types/express -D
```

该命令成功执行之后，`package.json` 中的 `devDependencies` 属性就会新增 Node.js 和 Express 对应的依赖包版本信息：

```
{
  "devDependencies": {
     "@types/express": "^4.17.8",
     "@types/node": "^14.11.2",
     "ts-node": "^9.0.0",
     "typescript": "^4.0.3"
  }
}
```

1.2.3 初始化 TypeScript 配置文件

为了能够灵活地配置 TypeScript 项目，我们还需要为本项目生成 TypeScript 配置文件，在命令行输入 `tsc --init` 之后，项目中就会自动创建一个 `tsconfig.json` 的文件。对于本项目来说，我们将使用以下配置项：

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./build",
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "strict": true
  }
}
```

1.2.4 创建简单的 Web 服务器

在创建简单的 Web 服务器之前，我们先来初始化项目的目录结构。首先在项目的根目录下创建一个 `src` 目录及 `controllers` 子目录：

```
├── src
│   ├── controllers
│   │   └── UserController.ts
│   └── index.ts
```

接着新建 `UserController.ts` 和 `index.ts` 这两个文件并分别输入以下内容：

UserController.ts

```
import { Controller, Get } from "@overnightjs/core";
import { Request, Response } from "express";

@Controller("api/users")
export class UserController {
  @Get("")
  private getAll(req: Request, res: Response) {
    return res.status(200).json({
      message: "成功获取所有用户",
    });
  }
}
```

index.ts

```
import { Server } from "@overnightjs/core";
import { UserController } from "./controllers/UserController";

const PORT = 3000;

export class SampleServer extends Server {
  constructor() {
```

```
    super(process.env.NODE_ENV === "development");
    this.setupControllers();
  }

  private setupControllers(): void {
    const userController = new UserController();
    super.addControllers([userController]);
  }

  public start(port: number): void {
    this.app.listen(port, () => {
      console.log(`⚡[server]: Server is running at http://localhost:${PORT}`);
    });
  }
}

const sampleServer = new SampleServer();
sampleServer.start(PORT);
```

完成上述步骤之后，我们在项目的 `package.json` 中添加一个 `start` 命令来启动项目：

```
{
  "scripts": {
    "start": "ts-node ./src/index.ts"
  },
}
```

添加完 `start` 命令，我们就可以在命令行中通过 `npm start` 来启动 Web 服务器了。当服务器成功启动之后，命令行会输出以下消息：

```
> ts-node ./src/index.ts

⚡[server]: Server is running at http://localhost:3000
```

接着我们打开浏览器访问 http://localhost:3000/api/users 这个地址，你就会看到 `{"message":"成功获取所有用户"}` 这个信息。

1.2.5 安装 nodemon

为了方便后续的开发，我们还需要安装一个第三方包 `nodemon`。对于写过 Node.js 应用的小伙伴来说，对 `nodemon` 这个包应该不会陌生。`nodemon` 这个包会自动检测目录中文件的更改，当发现文件异动时，会自动重启 Node.js 应用程序。

同样，我们在命令行执行以下命令来安装它：

```
$ npm i nodemon -D
```

安装完成后，我们需要更新一下前面已经创建的 `start` 命令：

```
{
  "scripts": {
    "start": "nodemon ./src/index.ts"
  }
}
```

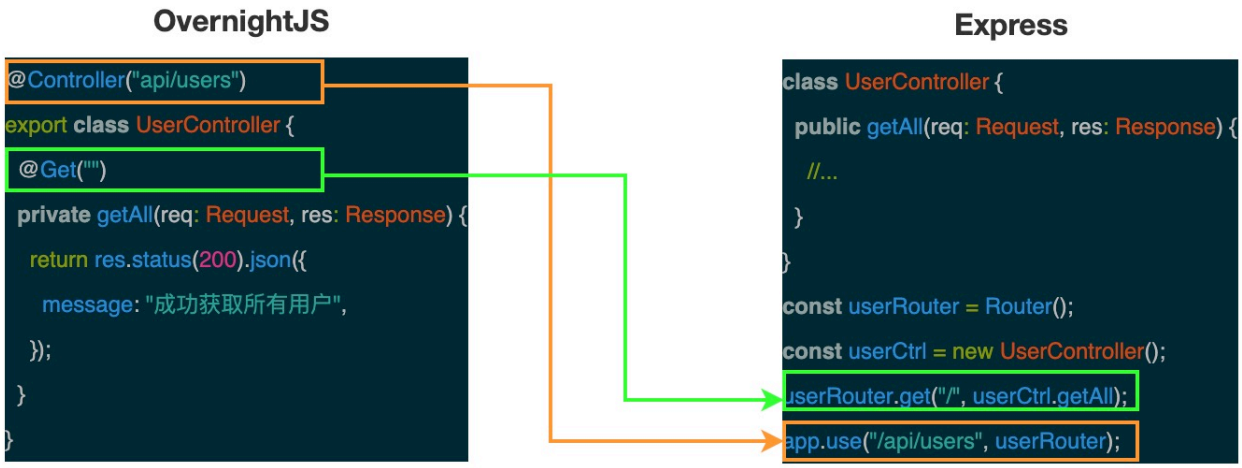
好的，现在我们已经知道如何使用 OvernightJS 来开发一个简单的 Web 服务器。接下来，阿宝哥将带大家一起来分析 OvernightJS 是如何使用 TypeScript 装饰器实现上述的功能。

二、OvernightJS 原理分析

在分析前面示例中 `@Controller` 和 `@Get` 装饰器原理前，我们先来看一下直接使用 Express 如何实现同样的功能：

```
import express, { Router, Request, Response } from "express";
const app = express();

const PORT = 3000;
class UserController {
  public getAll(req: Request, res: Response) {
    return res.status(200).json({
      message: "成功获取所有用户",
    });
  }
}

const userRouter = Router();
const userCtrl = new UserController();
userRouter.get("/", userCtrl.getAll);

app.use("/api/users", userRouter);

app.listen(PORT, () => {
  console.log(`⚡[server]: Server is running at http://localhost:${PORT}`);
});
```

在以上代码中，我们先通过调用 `Router` 方法创建了一个 `userRouter` 对象，然后进行相关路由的配置，接着使用 `app.use` 方法应用 `userRouter` 路由。下面我们用一张图来直观感受一下 OvernightJS 与 Express 在使用上的差异：

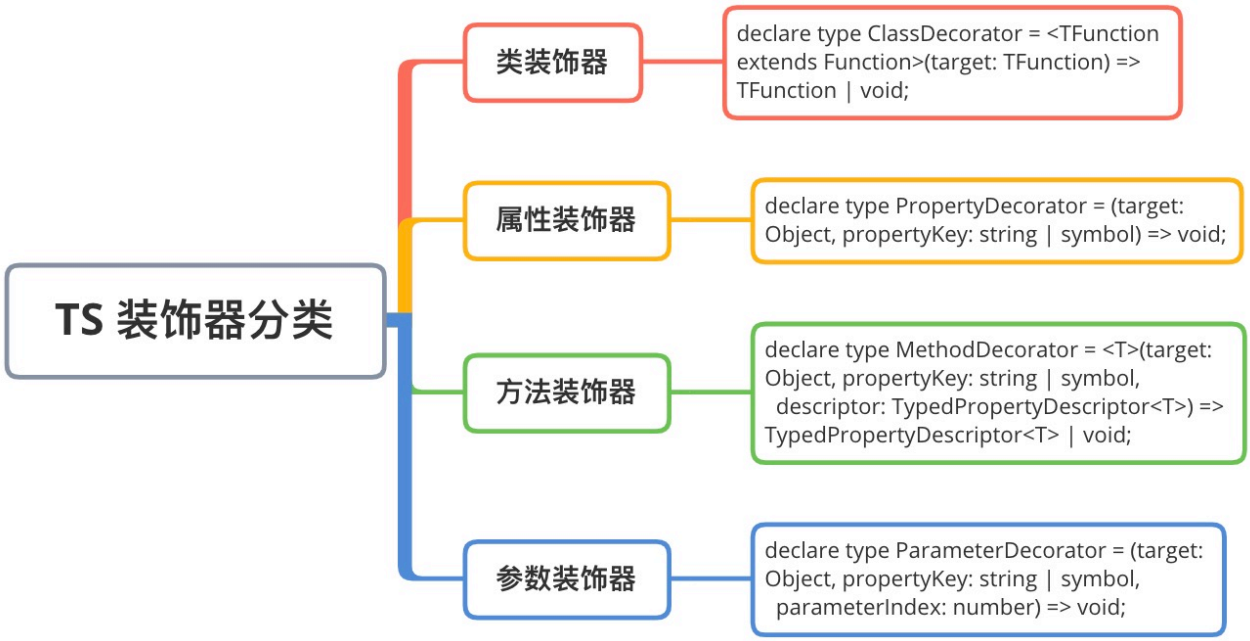


通过以上对比可知，利用 OvernightJS 提供的装饰器，可以让我们开发起来更加便捷。但大家要记住 OvernightJS 底层还是基于 Express，其内部最终还是通过 Express 提供的 API 来处理路由。

接下来为了能更好理解后续的内容，我们先来简单回顾一下 TypeScript 装饰器。

2.1 TypeScript 装饰器简介

装饰器是一个表达式，该表达式执行后，会返回一个函数。在 TypeScript 中装饰器可以分为以下 4 类：



需要注意的是，若要启用实验性的装饰器特性，你必须在命令行或 `tsconfig.json` 里启用 `experimentalDecorators` 编译器选项：

命令行：

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json：

```
{
  "compilerOptions": {
     "experimentalDecorators": true
   }
}
```

了解完 TypeScript 装饰器的分类，我们来开始分析 OvernightJS 框架中提供的装饰器。

2.2 @Controller 装饰器

在前面创建的简单 Web 服务器中，我们通过以下方式来使用 `@Controller` 装饰器：

```
@Controller("api/users")
export class UserController {}
```

很明显该装饰器应用在 `UserController` 类上，它属于类装饰器。OvernightJS 的项目结构很简单，我们可以很容易找到 `@Controller` 装饰器的定义：

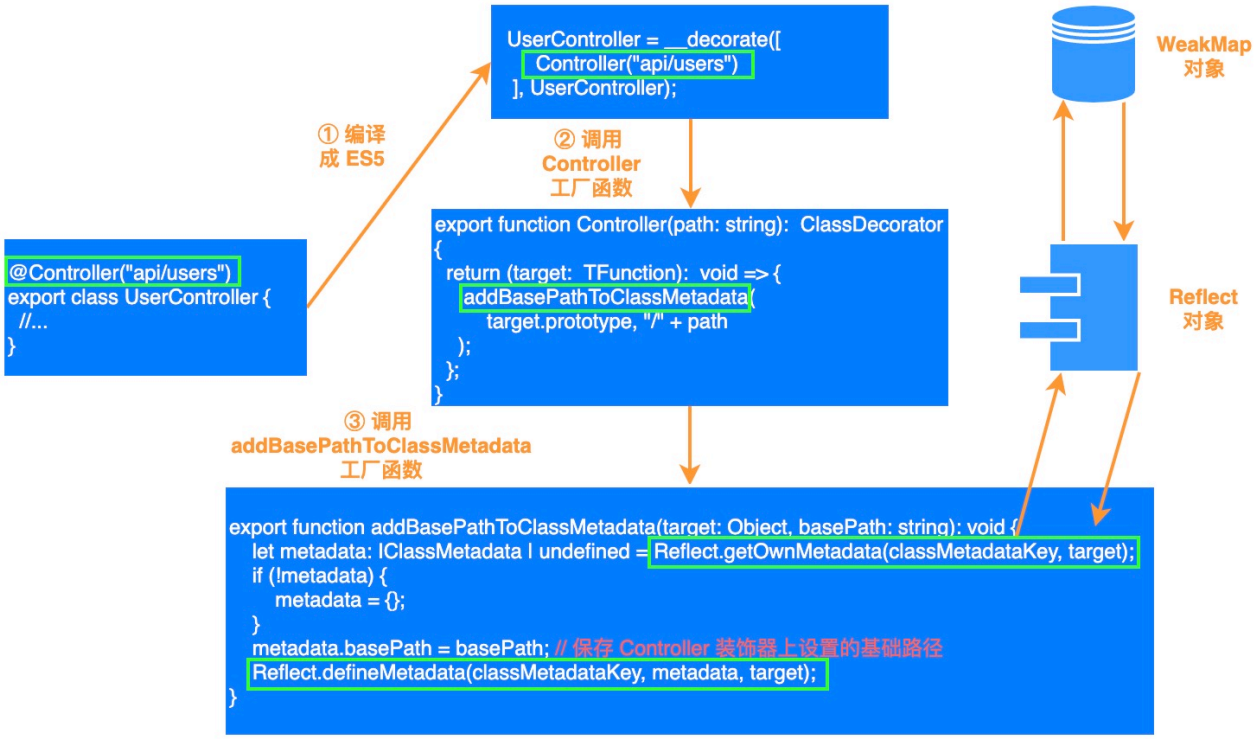
```
// src/core/lib/decorators/class.ts
export function Controller(path: string): ClassDecorator {
  return <TFunction extends Function>(target: TFunction): void => {
    addBasePathToClassMetadata(target.prototype, "/" + path);
  };
}
```

通过观察以上代码可知，Controller 函数是一个装饰器工厂，即调用该工厂方法之后会返回一个 ClassDecorator 对象。在 ClassDecorator 内部，会继续调用 `addBasePathToClassMetadata` 方法，把基础路径添加到类的元数据中：

```
// src/core/lib/decorators/class.ts
export function addBasePathToClassMetadata(target: Object, basePath: string):
void {
  let metadata: IClassMetadata | undefined =
Reflect.getOwnMetadata(classMetadataKey, target);
  if (!metadata) {
      metadata = {};
  }
  metadata.basePath = basePath;
  Reflect.defineMetadata(classMetadataKey, metadata, target);
}
```

`addBasePathToClassMetadata` 函数的实现很简单，主要是利用 Reflect API 实现元数据的存取操作。在以上代码中，会先获取 `target` 对象上已保存的 `metadata` 对象，如果不存在的话，会创建一个空的对象，然后把参数 `basePath` 的值添加该对象的 `basePath` 属性中，元数据设置完成后，在通过 `Reflect.defineMetadata` 方法进行元数据的保存。

下面我们用一张图来说明一下 `@Controller` 装饰器的处理流程：



在 OvernightJS 项目中，所使用的 Reflect API 是来自 reflect-metadata 这个第三方库。该库提供了很多 API 用于操作元数据，这里我们只简单介绍几个常用的 API：

```
// define metadata on an object or property
Reflect.defineMetadata(metadataKey, metadataValue, target);
Reflect.defineMetadata(metadataKey, metadataValue, target, propertyKey);

// check for presence of a metadata key on the prototype chain of an object or
property
let result = Reflect.hasMetadata(metadataKey, target);
let result = Reflect.hasMetadata(metadataKey, target, propertyKey);

// get metadata value of an own metadata key of an object or property
let result = Reflect.getOwnMetadata(metadataKey, target);
let result = Reflect.getOwnMetadata(metadataKey, target, propertyKey);

// get metadata value of a metadata key on the prototype chain of an object or
property
let result = Reflect.getMetadata(metadataKey, target);
let result = Reflect.getMetadata(metadataKey, target, propertyKey);

// delete metadata from an object or property
let result = Reflect.deleteMetadata(metadataKey, target);
let result = Reflect.deleteMetadata(metadataKey, target, propertyKey);
```

相信看到这里，可能有一些小伙伴会有疑问，通过 Reflect API 保存的元数据什么时候使用呢？这里我们先记住这个问题，后面我们再来分析它，接下来我们来开始分析 `@Get` 装饰器。

2.3 @Get 装饰器

在前面创建的简单 Web 服务器中，我们通过以下方式来使用 `@Get` 装饰器，该装饰器用于配置 Get 请求：

```
export class UserController {
  @Get("")
  private getAll(req: Request, res: Response) {
    return res.status(200).json({
      message: "成功获取所有用户",
    });
  }
}
```

`@Get` 装饰器应用在 `UserController` 类的 `getAll` 方法上，它属于方法装饰器。它的定义如下所示：

```
// src/core/lib/decorators/method.ts
export function Get(path?: string | RegExp): MethodDecorator &
PropertyDecorator {
  return helperForRoutes(HttpVerb.GET, path);
}
```

与 `Controller` 函数一样，`Get` 函数也是一个装饰器工厂，调用该函数之后会返回 `MethodDecorator & PropertyDecorator` 的交叉类型。除了 Get 请求方法之外，常见的 HTTP 请求方法还有 Post、Delete、Put、Patch 和 Head 等。为了统一处理这些请求方法，OvernightJS 内部封装了一个 `helperForRoutes` 函数，该函数的具体实现如下：

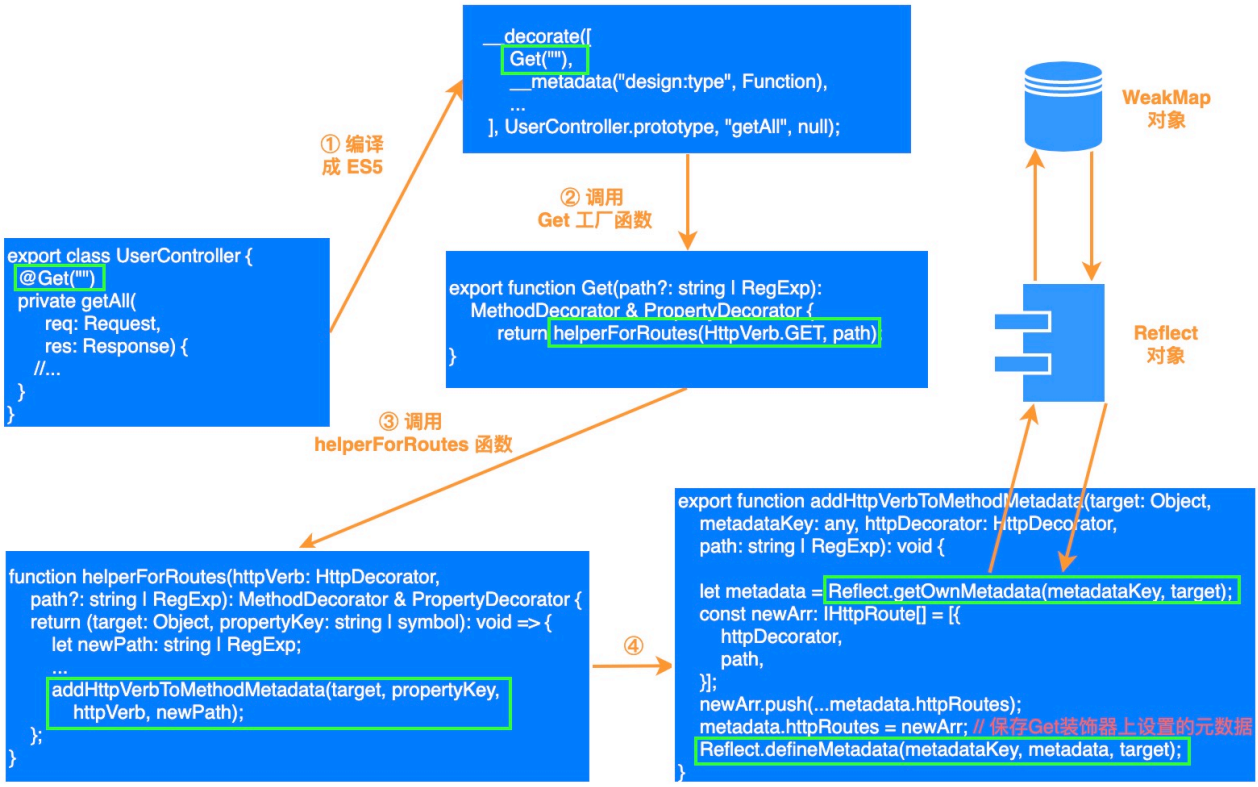
```
// src/core/lib/decorators/method.ts
function helperForRoutes(httpVerb: HttpDecorator, path?: string | RegExp):
MethodDecorator & PropertyDecorator {
  return (target: Object, propertyKey: string | symbol): void => {
      let newPath: string | RegExp;
      if (path === undefined) {
          newPath = '';
      } else if (path instanceof RegExp) {
          newPath = addForwardSlashToFrontOfRegex(path);
      } else { // assert (path instanceof string)
          newPath = '/' + path;
      }
      addHttpVerbToMethodMetadata(target, propertyKey, httpVerb, newPath);
    };
}
```

观察以上代码可知，在 `helperForRoutes` 方法内部，会继续调用 `addHttpVerbToMethodMetadata` 方法把请求方法和请求路径这些元数据保存起来。

```
// src/core/lib/decorators/method.ts
export function addHttpVerbToMethodMetadata(target: Object, metadataKey: any,
  httpDecorator: HttpDecorator, path: string | RegExp): void {
    let metadata: IMethodMetadata | undefined =
Reflect.getOwnMetadata(metadataKey, target);
    if (!metadata) {
        metadata = {};
    }
    if (!metadata.httpRoutes) {
        metadata.httpRoutes = [];
    }
    const newArr: IHttpRoute[] = [{
      httpDecorator,
      path,
    }];
    newArr.push(...metadata.httpRoutes);
    metadata.httpRoutes = newArr;
    Reflect.defineMetadata(metadataKey, metadata, target);
}
```

在 `addHttpVerbToMethodMetadata` 方法中，会先获取已保存的元数据，如果 `metadata` 对象不存在则会创建一个空的对象。然后会继续判断该对象上是否含有 `httpRoutes` 属性，没有的话会使用 `[]` 对象来作为该属性的属性值。而请求方法和请求路径这些元数据会以对象的形式保存到数组中，最终在通过 `Reflect.defineMetadata` 方法进行元数据的保存。

同样，我们用一张图来说明一下 `@Get` 装饰器的处理流程：



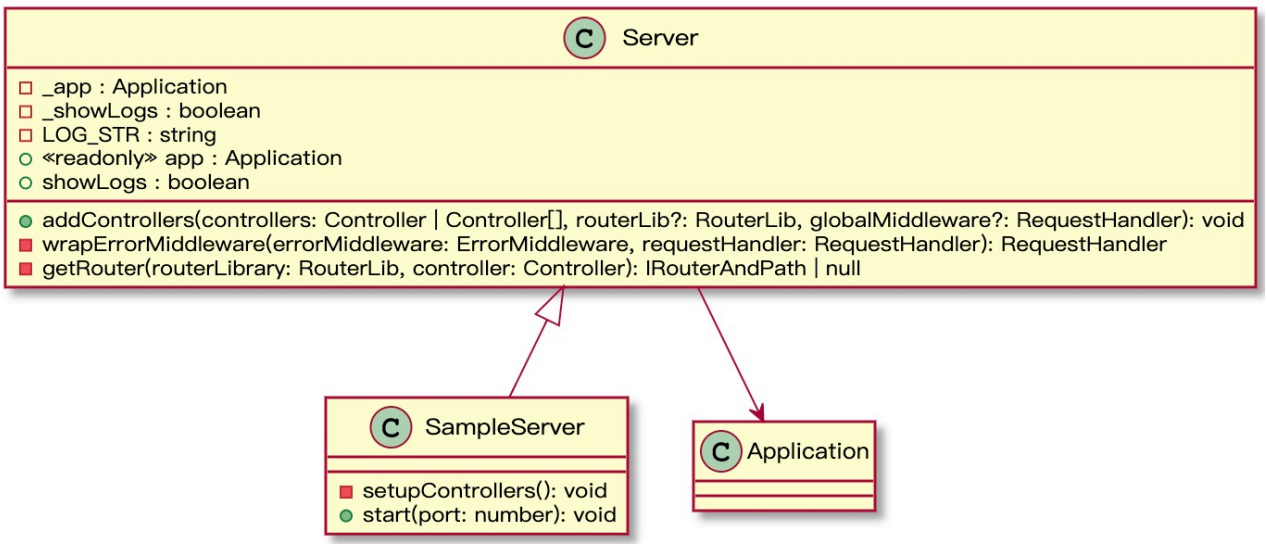
分析完 `@Controller` 和 `@Get` 装饰器，我们已经知道元数据是如何进行保存的。下面我们来回答 “通过 Reflect API 保存的元数据什么时候使用呢？” 这个问题。

2.4 元数据的使用

要搞清楚通过 Reflect API 保存的元数据什么时候使用，我们就需要来回顾一下前面开发的 `SampleServer` 服务器：

```
export class SampleServer extends Server {
  constructor() {
    super(process.env.NODE_ENV === "development");
    this.setupControllers();
  }

  private setupControllers(): void {
    const userController = new UserController();
    super.addControllers([userController]);
  }

  public start(port: number): void {
    this.app.listen(port, () => {
      console.log(`⚡[server]: Server is running at http://localhost:${PORT}`);
    });
  }
}

const sampleServer = new SampleServer();
sampleServer.start(PORT);
```

在以上代码中 `SampleServer` 类继承于 OvernightJS 内置的 `Server` 类，对应的 UML 类图如下所示：



此外，在 `SampleServer` 类中我们定义了 `setupControllers` 和 `start` 方法，分别用于初始化控制器和启动服务器。我们在自定义的控制器上使用了 `@Controller` 和 `@Get` 装饰器，因此接下来我们的重点就是分析 `setupControllers` 方法。该方法的内部实现很简单，就是手动创建控制器实例，然后调用父类的 `addControllers` 方法。

下面我们来分析 `addControllers` 方法，该方法位于 `src/core/lib/Server.ts` 文件中，具体实现如下：

```
// src/core/lib/Server.ts
export class Server {
  public addControllers(
    controllers: Controller | Controller[],
    routerLib?: RouterLib,
    globalMiddleware?: RequestHandler,
  ): void {
        controllers = (controllers instanceof Array) ? controllers :
[controllers];
        // ① 支持动态设置路由库
        const routerLibrary: RouterLib = routerLib || Router;
        controllers.forEach((controller: Controller) => {
          if (controller) {
              // ② 为每个控制器创建对应的路由对象
              const routerAndPath: IRouterAndPath | null =
this.getRouter(routerLibrary, controller);
              // ③ 注册路由
              if (routerAndPath) {
                   if (globalMiddleware) {
                       this.app.use(routerAndPath.basePath, globalMiddleware,
routerAndPath.router);
                   } else {
                       this.app.use(routerAndPath.basePath,
routerAndPath.router);
                   }
               }
            }
         });
    }
}
```

`addControllers` 方法的整个执行过程还是比较清晰，最核心的部分就是 `getRouter` 方法。在该方法内部就会处理通过装饰器保存的元数据。其实 `getRouter` 方法内部还会处理其他装饰器保存的元数据，简单起见我们只考虑与 `@Controller` 和 `@Get` 装饰器相关的处理逻辑。

```
// src/core/lib/Server.ts
export class Server {
 private getRouter(routerLibrary: RouterLib, controller: Controller):
IRouterAndPath | null {
        const prototype: any = Object.getPrototypeOf(controller);
        const classMetadata: IClassMetadata | undefined =
Reflect.getOwnMetadata(classMetadataKey, prototype);

        // 省略部分代码
        const { basePath, options, ...}: IClassMetadata = classMetadata;
```

```
        // ① 基于配置项创建Router对象
        const router: IRouter = routerLibrary(options);

        // ② 为路由对象添加路径和请求处理器
        let members: any = Object.getOwnPropertyNames(controller);
        members = members.concat(Object.getOwnPropertyNames(prototype));
        members.forEach((member: any) => {
            // ③ 获取方法中保存的元数据
            const methodMetadata: IMethodMetadata | undefined =
Reflect.getOwnMetadata(member, prototype);
            if (methodMetadata) {
                const { httpRoutes, ...}: IMethodMetadata = methodMetadata;
                let callBack: (...args: any[]) => any = (...args: any[]): any
=> {
                    return controller[member](...args);
                };
                // 省略部分代码
                if (httpRoutes) { // httpRoutes数组中包含了请求的方法和路径
                    // ④ 处理控制器类中通过@Get、@Post、@Put或@Delete装饰器保存的元数
据
                    httpRoutes.forEach((route: IHttpRoute) => {
                        const { httpDecorator, path }: IHttpRoute = route;
                        // ⑤ 为router对象设置对应的路由信息
                        if (middlewares) {
                            router[httpDecorator](path, middlewares, callBack);
                        } else {
                            router[httpDecorator](path, callBack);
                        }
                    });
                }
            }
        });
        return { basePath, router, };
    }
}
```

现在我们已经知道 OvernightJS 内部如何利用装饰器来为控制器类配置路由信息，这里阿宝哥用一张图来总结 OvernightJS 的工作流程：

在 OvernightJS 内部除了 `@Controller`、`@Get`、`@Post`、`@Delete` 等装饰器之外，还提供了用于注册中间件的 `@Middleware` 装饰器及用于设置异常处理中间件的 `@ErrorMiddleware` 装饰器。感兴趣的小伙伴可以参考一下阿宝哥的学习思路，自行阅读 OvernightJS 项目的源码。

三、参考资源

- Github - overnight
- expressjs.com

第十二章 编写高效 TS 代码的一些建议

本章阿宝哥将分享编写高效 TS 代码的 5 个建议，希望这些建议对大家编写 TS 代码能有一些帮助。

一、尽量减少重复代码

对于刚接触 TypeScript 的小伙伴来说，在定义接口时，可能一不小心会出现以下类似的重复代码。比如：

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate {
  firstName: string;
  lastName: string;
  birth: Date;
}
```

很明显，相对于 `Person` 接口来说，`PersonWithBirthDate` 接口只是多了一个 `birth` 属性，其他的属性跟 `Person` 接口是一样的。那么如何避免出现例子中的重复代码呢？要解决这个问题，可以利用 `extends` 关键字：

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate extends Person {
  birth: Date;
}
```

当然除了使用 `extends` 关键字之外，也可以使用交叉运算符（&）：

```
type PersonWithBirthDate = Person & { birth: Date };
```

另外，有时候你可能还会发现自己想要定义一个类型来匹配一个初始配置对象的「形状」，比如：

```
const INIT_OPTIONS = {
  width: 640,
  height: 480,
  color: "#00FF00",
  label: "VGA",
};

interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
```

其实，对于 Options 接口来说，你也可以使用 typeof 操作符来快速获取配置对象的「形状」：

```
type Options = typeof INIT_OPTIONS;
```

而在使用可辨识联合（代数数据类型或标签联合类型）的过程中，也可能出现重复代码。比如：

```
interface SaveAction {
  type: 'save';
  // ...
}

interface LoadAction {
  type: 'load';
  // ...
}

type Action = SaveAction | LoadAction;
type ActionType = 'save' | 'load'; // Repeated types!
```

为了避免重复定义 `'save'` 和 `'load'`，我们可以使用成员访问语法，来提取对象中属性的类型：

```
type ActionType = Action['type']; // 类型是 "save" | "load"
```

然而在实际的开发过程中，重复的类型并不总是那么容易被发现。有时它们会被语法所掩盖。比如有多个函数拥有相同的类型签名：

```
function get(url: string, opts: Options): Promise<Response> { /* ... */ }
function post(url: string, opts: Options): Promise<Response> { /* ... */ }
```

对于上面的 get 和 post 方法，为了避免重复的代码，你可以提取统一的类型签名：

```
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;

const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
```

二、使用更精确的类型替代字符串类型

假设你正在构建一个音乐集，并希望为专辑定义一个类型。这时你可以使用 `interface` 关键字来定义一个 `Album` 类型：

```
interface Album {
  artist: string; // 艺术家
  title: string; // 专辑标题
  releaseDate: string; // 发行日期：YYYY-MM-DD
  recordingType: string; // 录制类型："live" 或 "studio"
}
```

对于 `Album` 类型，你希望 `releaseDate` 属性值的格式为 `YYYY-MM-DD`，而 `recordingType` 属性值的范围为 `live` 或 `studio`。但因为接口中 `releaseDate` 和 `recordingType` 属性的类型都是字符串，所以在使用 `Album` 接口时，可能会出现以下问题：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  releaseDate: "November 31, 1991", // 与预期格式不匹配
  recordingType: "Studio", // 与预期格式不匹配
};
```

虽然 `releaseDate` 和 `recordingType` 的值与预期的格式不匹配，但此时 TypeScript 编译器并不能发现该问题。为了解决这个问题，你应该为 `releaseDate` 和 `recordingType` 属性定义更精确的类型，比如这样：

```
interface Album {
  artist: string; // 艺术家
  title: string; // 专辑标题
  releaseDate: Date; // 发行日期：YYYY-MM-DD
  recordingType: "studio" | "live"; // 录制类型："live" 或 "studio"
}
```

重新定义 `Album` 接口之后，对于前面的赋值语句，TypeScript 编译器就会提示以下异常信息：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  // 不能将类型“string”分配给类型“Date”。ts(2322)
  releaseDate: "November 31, 1991", // Error
  // 不能将类型“"Studio"”分配给类型“"studio" | "live"”。ts(2322)
  recordingType: "Studio", // Error
};
```

为了解决上面的问题，你需要为 `releaseDate` 和 `recordingType` 属性设置正确的类型，比如这样：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  releaseDate: new Date("1991-11-31"),
  recordingType: "studio",
};
```

另一个错误使用字符串类型的场景是设置函数的参数类型。假设你需要写一个函数，用于从一个对象数组中抽取某个属性的值并保存到数组中，在 Underscore 库中，这个操作被称为 “pluck”。要实现该功能，你可能最先想到以下代码：

```
function pluck(record: any[], key: string): any[] {
  return record.map((r) => r[key]);
}
```

对于以上的 `pluck` 函数并不是很好，因为它使用了 `any` 类型，特别是作为返回值的类型。那么如何优化 `pluck` 函数呢？首先，可以通过引入一个泛型参数来改善类型签名：

```
function pluck<T>(record: T[], key: string): any[] {
  // Element implicitly has an 'any' type because expression of type 'string'
can't be used to
  // index type 'unknown'.
  // No index signature with a parameter of type 'string' was found on type
'unknown'.(7053)
  return record.map((r) => r[key]); // Error
}
```

通过以上的异常信息，可知字符串类型的 key 不能被作为 unknown 类型的索引类型。要从对象上获取某个属性的值，你需要保证参数 key 是对象中的属性。

说到这里相信有一些小伙伴已经想到了 `keyof` 操作符，它是 TypeScript 2.1 版本引入的，可用于获取某种类型的所有键，其返回类型是联合类型。接着使用 `keyof` 操作符来更新一下 `pluck` 函数：

```
function pluck<T>(record: T[], key: keyof T) {
  return record.map((r) => r[key]);
}
```

对于更新后的 `pluck` 函数，你的 IDE 将会为你自动推断出该函数的返回类型：

```
function pluck<T>(record: T[], key: keyof T): T[keyof T][]
```

对于更新后的 `pluck` 函数，你可以使用前面定义的 `Album` 类型来测试一下：

```
const albums: Album[] = [{
  artist: "Michael Jackson",
  title: "Dangerous",
  releaseDate: new Date("1991-11-31"),
  recordingType: "studio",
}];

// let releaseDateArr: (string | Date)[]
let releaseDateArr = pluck(albums, 'releaseDate');
```

示例中的 `releaseDateArr` 变量，它的类型被推断为 `(string | Date)[]`，很明显这并不是你所期望的，它的正确类型应该是 `Date[]`。那么应该如何解决该问题呢？这时你需要引入第二个泛型参数 `K`，然后使用 `extends` 来进行约束：

```
function pluck<T, K extends keyof T>(record: T[], key: K): T[K][] {
  return record.map((r) => r[key]);
}

// let releaseDateArr: Date[]
let releaseDateArr = pluck(albums, 'releaseDate');
```

三、定义的类型总是表示有效的状态

假设你正在构建一个允许用户指定页码，然后加载并显示该页面对应内容的 Web 应用程序。首先，你可能会先定义 `State` 对象：

```
interface State {
  pageContent: string;
  isLoading: boolean;
  errorMsg?: string;
}
```

接着你会定义一个 `renderPage` 函数，用来渲染页面：

```
function renderPage(state: State) {
  if (state.errorMsg) {
    return `呜呜呜，加载页面出现异常了...${state.errorMsg}`;
  } else if (state.isLoading) {
    return `页面加载中~~~`;
  }
  return `<div>${state.pageContent}</div>`;
}

// 输出结果：页面加载中~~~
console.log(renderPage({isLoading: true, pageContent: ""}));
// 输出结果：<div>大家好，我是阿宝哥</div>
console.log(renderPage({isLoading: false, pageContent: "大家好，我是阿宝哥"}));
```

创建好 `renderPage` 函数，你可以继续定义一个 `changePage` 函数，用于根据页码获取对应的页面数据：

```
async function changePage(state: State, newPage: string) {
  state.isLoading = true;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const text = await response.text();
    state.isLoading = false;
    state.pageContent = text;
  } catch (e) {
    state.errorMsg = "" + e;
  }
}
```

对于以上的 `changePage` 函数，它存在以下问题：

- 在 catch 语句中，未把 `state.isLoading` 的状态设置为 `false`；
- 未及时清理 `state.errorMsg` 的值，因此如果之前的请求失败，那么你将继续看到错误消息，而不是加载消息。

出现上述问题的原因是，前面定义的 `State` 类型允许同时设置 `isLoading` 和 `errorMsg` 的值，尽管这是一种无效的状态。针对这个问题，你可以考虑引入可辨识联合类型来定义不同的页面请求状态：

```
interface RequestPending {
  state: "pending";
}

interface RequestError {
  state: "error";
  errorMsg: string;
```

```
}

interface RequestSuccess {
  state: "ok";
  pageContent: string;
}

type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: { [page: string]: RequestState };
}
```

在以上代码中，通过使用可辨识联合类型分别定义了 3 种不同的请求状态，这样就可以很容易的区分出不同的请求状态，从而让业务逻辑处理更加清晰。接下来，需要基于更新后的 `State` 类型，来分别更新一下前面创建的 `renderPage` 和 `changePage` 函数：

更新后的 renderPage 函数

```
function renderPage(state: State) {
  const { currentPage } = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case "pending":
      return `页面加载中~~~`;
    case "error":
      return `呜呜呜，加载第${currentPage}页出现异常了...${requestState.errorMsg}`;
    case "ok":
      `<div>第${currentPage}页的内容：${requestState.pageContent}</div>`;
  }
}
```

更新后的 changePage 函数

```
async function changePage(state: State, newPage: string) {
  state.requests[newPage] = { state: "pending" };
  state.currentPage = newPage;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`无法正常加载页面 ${newPage}: ${response.statusText}`);
    }
    const pageContent = await response.text();
    state.requests[newPage] = { state: "ok", pageContent };
  } catch (e) {
    state.requests[newPage] = { state: "error", errorMsg: "" + e };
  }
```

```
}
```

在 `changePage` 函数中，会根据不同的情形设置不同的请求状态，而不同的请求状态会包含不同的信息。这样 `renderPage` 函数就可以根据统一的 `state` 属性值来进行相应的处理。因此，通过使用可辨识联合类型，让请求的每种状态都是有效的状态，不会出现无效状态的问题。

四、选择条件类型而不是重载声明

假设你要使用 TS 实现一个 `double` 函数，该函数支持 `string` 或 `number` 类型。这时，你可能已经想到了使用联合类型和函数重载：

```
function double(x: number | string): number | string;
function double(x: any) {
  return x + x;
}
```

虽然这个 `double` 函数的声明是正确的，但它有一点不精确：

```
// const num: string | number
const num = double(10);
// const str: string | number
const str = double('ts');
```

对于 `double` 函数，你期望传入的参数类型是 number 类型，其返回值的类型也是 number 类型。当你传入的参数类型是 string 类型，其返回的类型也是 string 类型。而上面的 `double` 函数却是返回了 `string | number` 类型。为了实现上述的要求，你可能想到了引入泛型变量和泛型约束：

```
function double<T extends number | string>(x: T): T;
function double(x: any) {
  return x + x;
}
```

在上面的 `double` 函数中，引入了泛型变量 `T`，同时使用 `extends` 约束了其类型范围。

```
// const num: 10
const num = double(10);
// const str: "ts"
const str = double('ts');
console.log(str);
```

不幸的是，我们对精确度的追求超过了预期。现在的类型有点太精确了。当传递一个字符串类型时，`double` 声明将返回一个字符串类型，这是正确的。但是当传递一个字符串字面量类型时，返回的类型是相同的字符串字面量类型。这是错误的，因为 `ts` 经过 `double` 函数处理后，返回的是 `tsts`，而不是 `ts`。

另一种方案是提供多种类型声明。虽然 TypeScript 只允许你编写一个具体的实现，但它允许你编写任意数量的类型声明。你可以使用函数重载来改善 `double` 的类型：

```
function double(x: number): number;
function double(x: string): string;
function double(x: any) {
  return x + x;
}

// const num: number
const num = double(10);
// const str: string
const str = double("ts");
```

很明显此时 num 和 str 变量的类型都是正确的，但不幸的是，`double` 函数还有一个小问题。因为 `double` 函数的声明只支持 `string` 或 `number` 类型的值，而不支持 `string | number` 联合类型，比如：

```
function doubleFn(x: number | string) {
  // Argument of type 'string | number' is not assignable to
  // parameter of type 'number'.
  // Argument of type 'string | number' is not assignable to
  // parameter of type 'string'.
  return double(x); // Error
}
```

为什么会提示以上的错误呢？因为当 TypeScript 编译器处理函数重载时，它会查找重载列表，直到找一个匹配的签名。对于 `number | string` 联合类型，很明显是匹配失败的。

然而对于上述的问题，虽然可以通过新增 `string | number` 的重载签名来解决，但最好的方案是使用条件类型。在类型空间中，条件类型就像 `if` 语句一样：

```
function double<T extends number | string>(
  x: T
): T extends string ? string : number;
function double(x: any) {
  return x + x;
}
```

这与前面引入泛型版本的 `double` 函数声明类似，只是它引入更复杂的返回类型。条件类型使用起来很简单，与 JavaScript 中的三目运算符（?:）一样的规则。`T extends string ? string : number` 的意思是，如果 `T` 类型是 `string` 类型的子集，则 `double` 函数的返回值类型为 `string` 类型，否则为 `number` 类型。

在引入条件类型之后，前面的所有例子就可以正常工作了：

```ts
// const num: number
const num = double(10);
// const str: string
const str = double("ts");

// function f(x: string | number): string | number
function f(x: number | string) {
  return double(x);
}
```

五、一次性创建对象

在 JavaScript 中可以很容易地创建一个表示二维坐标点的对象：

```ts
const pt = {};
pt.x = 3;
pt.y = 4;
```

然而对于同样的代码，在 TypeScript 中会提示以下错误信息：

```ts
const pt = {};
// Property 'x' does not exist on type '{}'
pt.x = 3; // Error
// Property 'y' does not exist on type '{}'
pt.y = 4; // Error
```

这是因为第一行中 pt 变量的类型是根据它的值 {} 推断出来的，你只能对已知的属性赋值。针对这个问题，你可能会想到一种解决方案，即新声明一个 Point 类型，然后把它作为 pt 变量的类型：

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

// Type '{}' is missing the following properties from type 'Point': x, y(2739)
const pt: Point = {}; // Error
pt.x = 3;
pt.y = 4;
```

那么如何解决上述问题呢？其中一种最简单的解决方案是一次性创建对象：

```ts
const pt = {
  x: 3,
  y: 4,
}; // OK
```

如果你想一步一步地创建对象，你可以使用类型断言（as）来消除类型检查：

```
const pt = {} as Point;
pt.x = 3;
pt.y = 4; // OK
```

但是更好的方法是一次性创建对象并显式声明变量的类型：

```
const pt: Point = {
  x: 3,
  y: 4,
};
```

而当你需要从较小的对象来构建一个较大的对象时，你可能会这样处理，比如：

```
const pt = { x: 3, y: 4 };
const id = { name: "Pythagoras" };
const namedPoint = {};
Object.assign(namedPoint, pt, id);

// Property 'id' does not exist on type '{}'.(2339)
namedPoint.name; // Error
```

为了解决上述问题，你可以使用对象展开运算符 `...` 来一次性构建大的对象：

```
const namedPoint = {...pt, ...id};
namedPoint.name; // OK, type is string
```

此外，你还可以使用对象展开运算符，以一种类型安全的方式逐个字段地构建对象。关键是在每次更新时使用一个新变量，这样每个变量都会得到一个新类型：

```
const pt0 = {};
const pt1 = {...pt0, x: 3};
const pt: Point = {...pt1, y: 4}; // OK
```

虽然这是构建这样一个简单对象的一种迂回方式，但对于向对象添加属性并允许 TypeScript 推断新类型来说，这可能是一种有用的技术。要以类型安全的方式有条件地添加属性，可以使用带 `null` 或 `{}` 的对象展开运算符，它不会添加任何属性：

```
declare var hasMiddle: boolean;
const firstLast = {first: 'Harry', last: 'Truman'};
const president = {...firstLast, ...(hasMiddle ? {middle: 'S'} : {})};
```

如果在编辑器中鼠标移到 president，你将看到 TypeScript 推断出的类型：

```
const president: {
  middle?: string;
  first: string;
  last: string;
}
```

最终通过设置 `hasMiddle` 变量的值，你就可以控制 president 对象中 `middle` 属性的值：

```
declare var hasMiddle: boolean;
var hasMiddle = true;
const firstLast = {first: 'Harry', last: 'Truman'};
const president = {...firstLast, ...(hasMiddle ? {middle: 'S'} : {})};

let mid = president.middle
console.log(mid); // S
```

六、参考资源

- effective-typescript-specific-ways-improve

第十三章 让人眼前一亮的 10 大 TS 项目

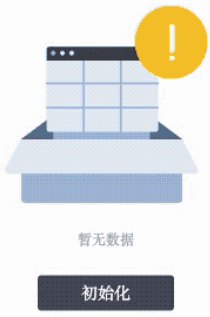
最后一章阿宝哥将跟大家分享一下 10 个不错的 TS 项目，感兴趣的小伙伴可以了解一下。

一、ava

> 🤖 A framework for automated visual analytics.
>
> https://github.com/antvis/AVA

AVA（A Visual Analytics）是为了更简便的可视分析而生的技术框架。 其名称中的第一个 **A** 具有多重涵义：它说明了这是一个出自阿里巴巴集团（Alibaba）技术框架，其目标是成为一个**自动化**（Automated）、**智能驱动**（AI driven）、**支持增强分析**（Augmented）的可视分析解决方案。

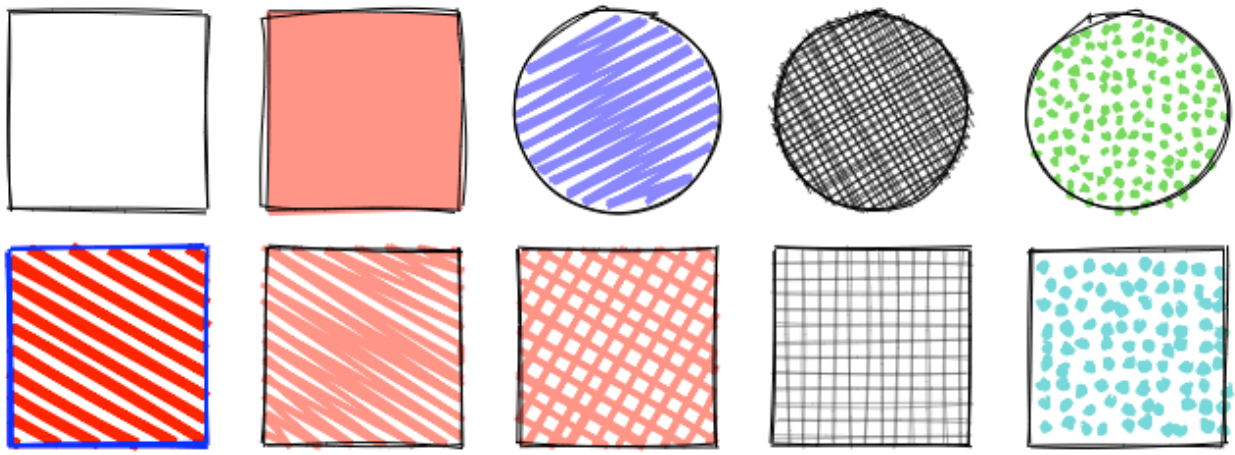
```
import { autoChart } from '@antv/chart-advisor';

const container = document.getElementById('mountNode');

const data = [
  {f1: 'a', f2: 101},
  {f1: 'b', f2: 104},
  {f1: 'c', f2: 105},
];

autoChart(container, data, {toolbar:true, development: true});
```



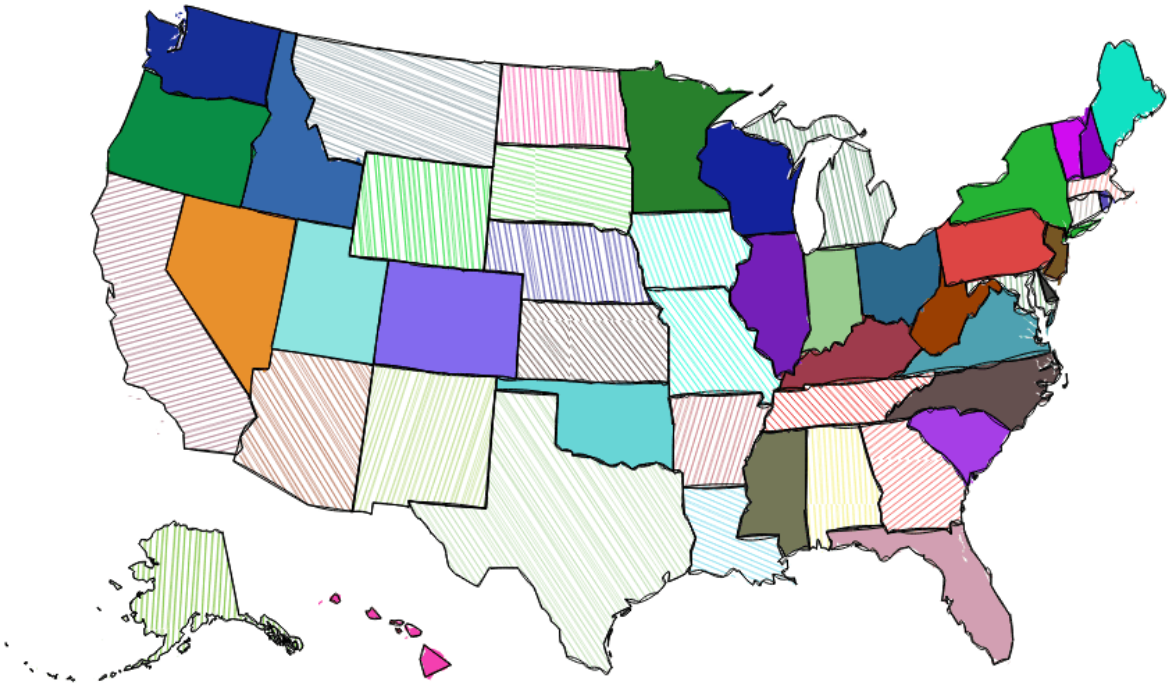
二、rough

> Create graphics with a hand-drawn, sketchy, appearance.
>
> https://github.com/pshihn/rough

Rough.js 是一个轻量的图形库（压缩后的 <9 kB），可以让你用手绘的方式绘制草图。该库提供绘制线条、曲线、弧线、多边形、圆形和椭圆的基础能力，同时支持绘制 SVG 路径。Rough.js 可同时支持 Canvas 和 SVG。



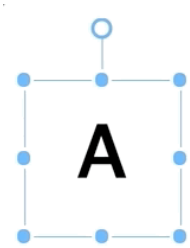
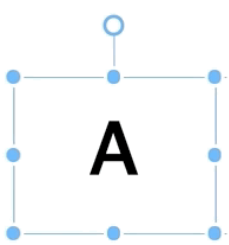
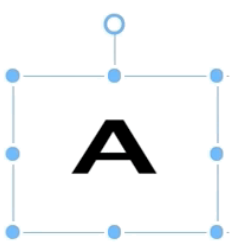
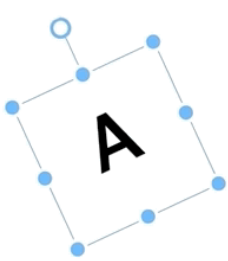
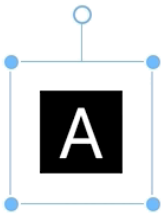
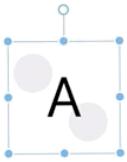
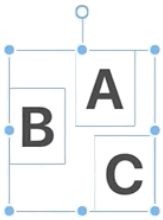
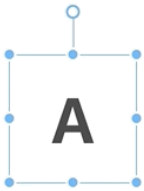
除了生成简单的图形之外，使用 Rough.js 也可以用来生成复杂的图形，比如手绘风格的地图：



三、moveable

> Moveable! Draggable! Resizable! Scalable! Rotatable! Warpable! Pinchable! Groupable! Snappable!
>
> https://github.com/daybrush/moveable

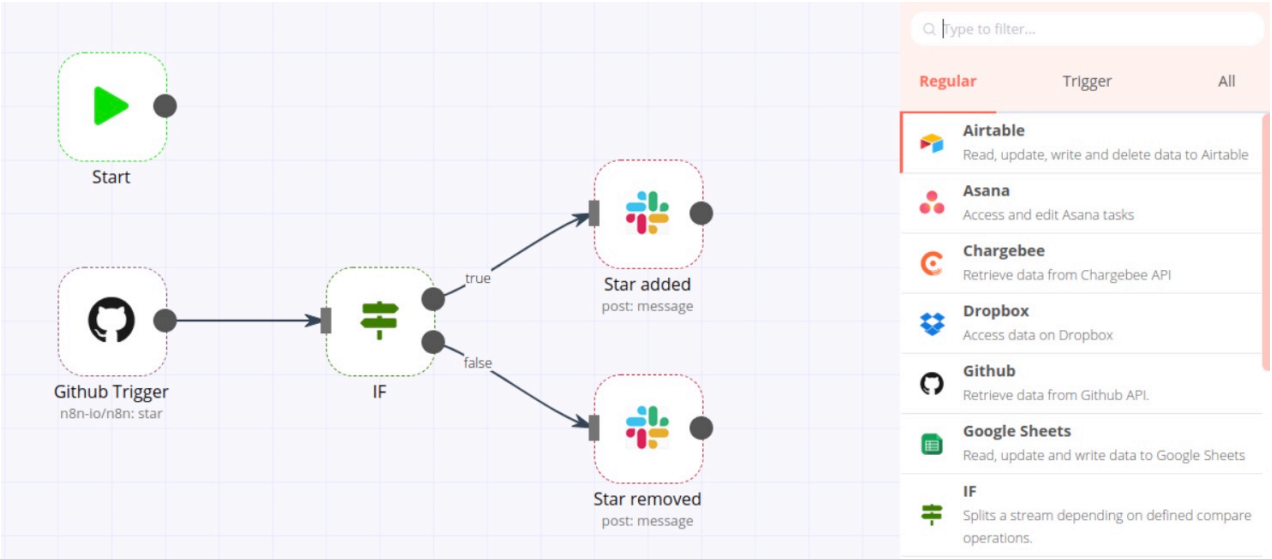
Moveable 可以让你把指定的元素，变成可拖动的，可调整大小的，可伸缩的，可旋转的或可组合的元素。

Draggable	Resizable	Scalable	Rotatable
			
Warpable	Pinchable	Groupable	Snappable
			

四、n8n

> Free and open fair-code licensed node based Workflow Automation Tool. Easily automate tasks across different services.
>
> https://github.com/n8n-io/n8n

n8n 是一个免费、开放、fair-code 许可，基于节点的工作流自动化工具。它可以自托管，很容易扩展，因此也可以与内部工具一起使用。n8n 类似 IFTTT、Zapier，可以互联互通包括 GitHub、Dropbox、Google、NextCLoud、RSS、Slack、Telegram 在内的 100 多个在线服务。利用 n8n 你可以方便地实现当 A 条件发生，触发 B 服务这样的自动工作流程。



> IFTTT 是一个被称为 “网络自动化神器” 的创新型互联网服务理念，它很实用而且概念很简单。IFTTT 全称是 **If this then that**，意思是如果满足 “this” 条件，则触发执行 “that” 动作。

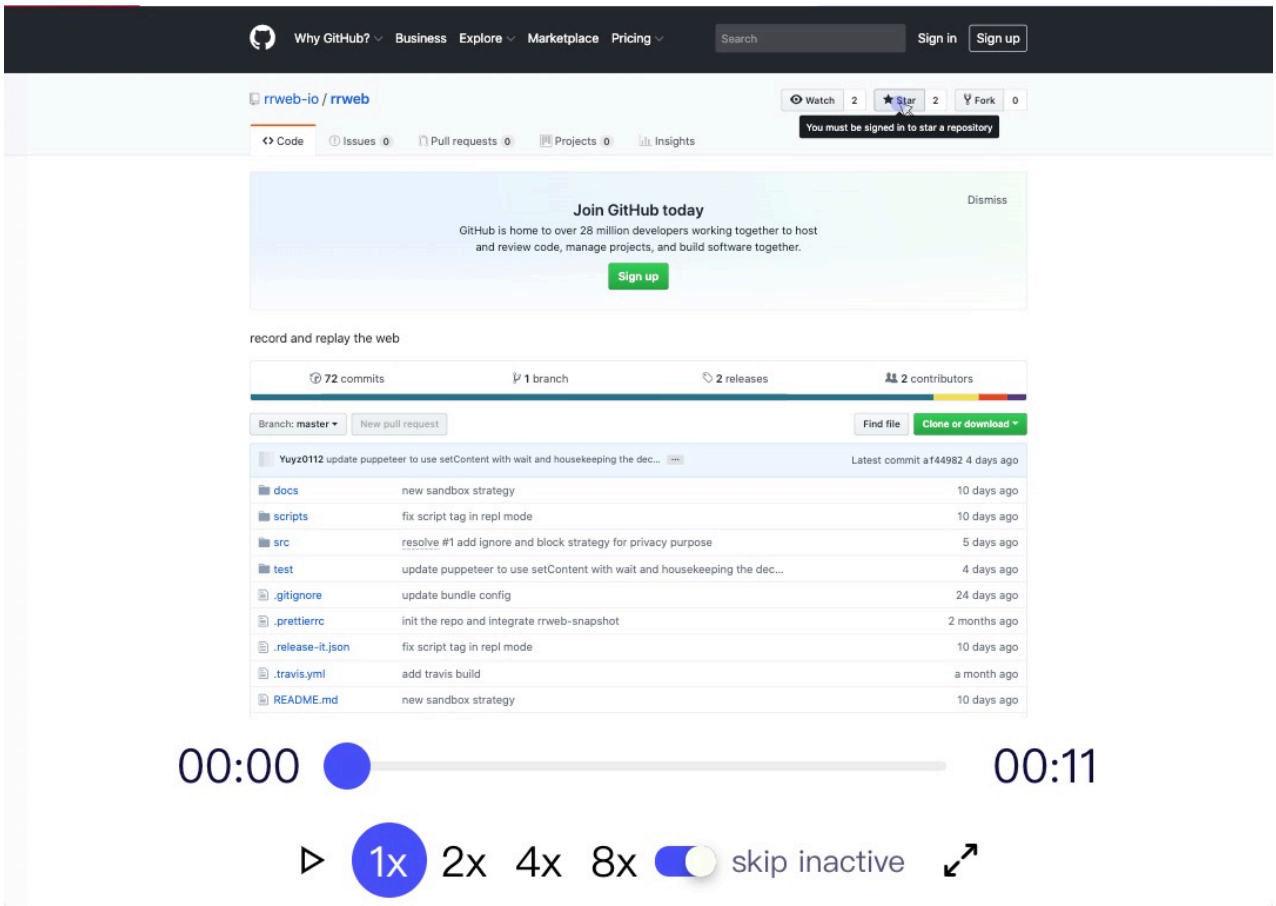
五、rrweb-io

> record and replay the web.
>
> https://github.com/rrweb-io/rrweb

rrweb 是 **'record and replay the web'** 的简写，旨在利用现代浏览器所提供的强大 API 录制并回放任意 Web 界面中的用户操作。

rrweb 主要由 3 部分组成：

- **rrweb-snapshot**，包含 snapshot 和 rebuild 两个功能。snapshot 用于将 DOM 及其状态转化为可序列化的数据结构并添加唯一标识；rebuild 则是将 snapshot 记录的数据结构重建为对应的 DOM。
- **rrweb**，包含 record 和 replay 两个功能。record 用于记录 DOM 中的所有变更（mutation）；replay 则是将记录的变更按照对应的时间一一重放。
- **rrweb-player**，为 rrweb 提供一套 UI 控件，提供基于 GUI 的暂停、快进、拖拽至任意时间点播放等功能。



如上图所示，在完成录制 Web 界面中的用户操作之后，就可以 **rrweb-player** 进行暂停、快进、拖拽至任意时间点等播放功能。看完之后，有些小伙伴是不是手痒了，rrweb 的作者也很贴心为我们提供了三个在线示例：

- Bootstrap checkout form（https://www.rrweb.io/demo/checkout-form）
- Conversational Form（https://www.rrweb.io/demo/chat）
- Tetris game（https://www.rrweb.io/demo/tetris?lan=en）

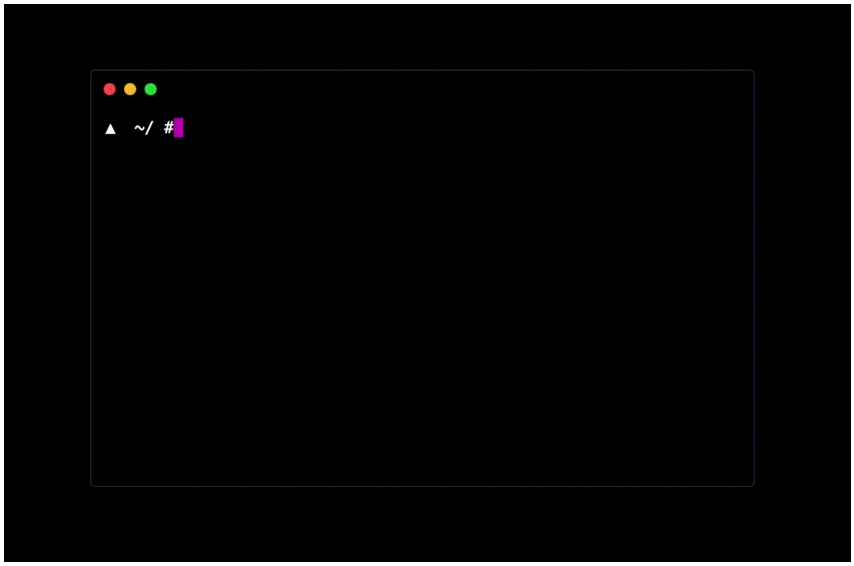
> Tetris 即俄罗斯方块，适用于所有电子游戏机和电脑操作系统，是一个最初由阿列克谢帕吉特诺夫在苏联设计和编程的益智类视频游戏。

俄罗斯方块（Tetris）这个游戏，勾起了本人对童年的无限回忆，一波回忆杀，有木有？

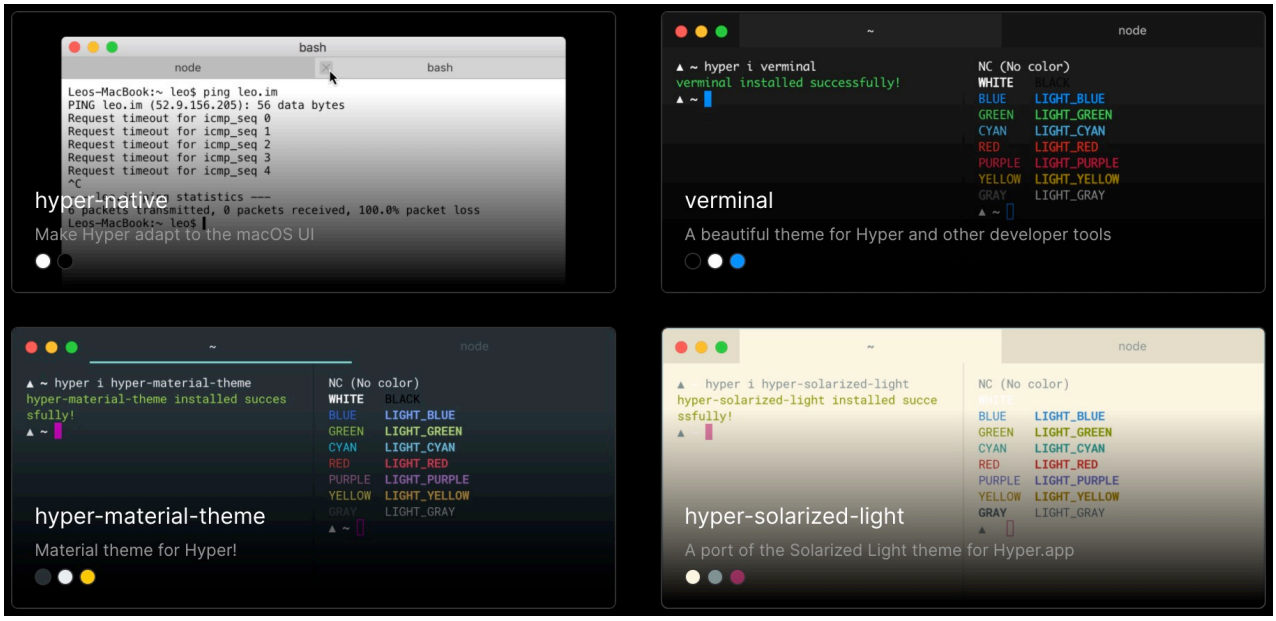
六、hyper

> A terminal built on web technologies.
>
> https://github.com/vercel/hyper

Hyper 是使用 Web 技术开发的命令行工具，它和 VS Code 一样，都是基于 Electron，提供实用的 Plugins 和 Themes。



开发者可以根据自己的喜好，在 Hyper 官网 —— https://hyper.is/themes 选择自己喜欢的主题，当然也可以与其他用户分享自己开发的主题：

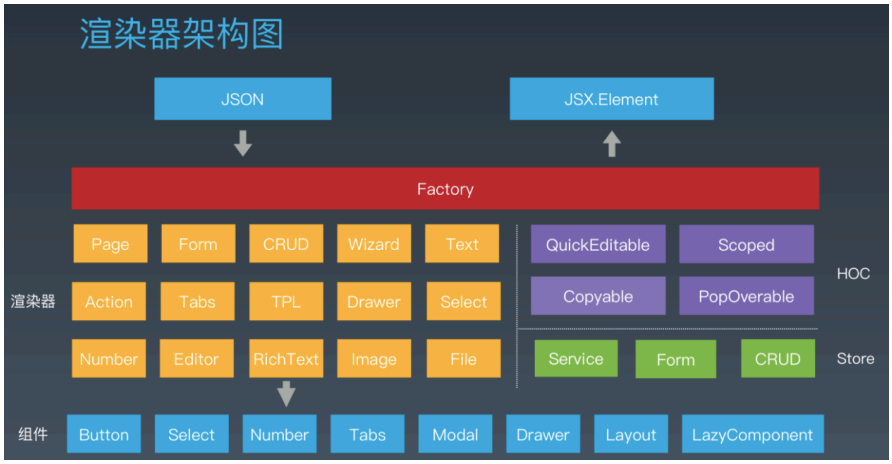


七、amis

> 前端低代码框架，通过 JSON 配置就能生成各种后台页面。
>
> https://github.com/baidu/amis

amis 百度开源的前端低代码框架，通过 JSON 配置就能生成各种后台页面，极大减少开发成本，甚至可以不需要了解前端。目前在百度广泛用于内部平台的前端开发，已有 **100+** 部门使用，创建了 **3w+** 页面。

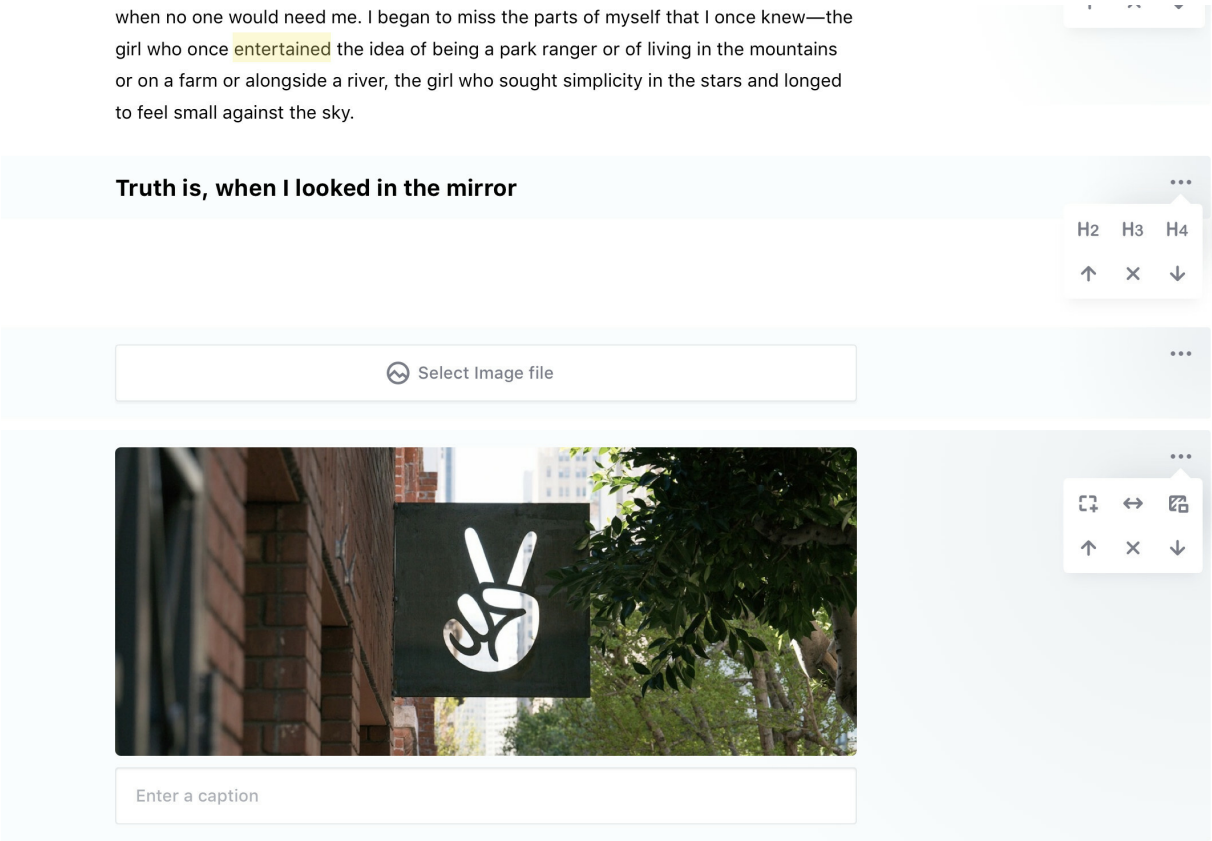
amis 渲染器架构图



八、editor.js

> A block-styled editor with clean JSON output.
>
> https://github.com/codex-team/editor.js

Editor.js 是一个块风格的编辑器。块是组成条目的结构单元。例如，段落，标题，图像，视频，列表都是块。每个块由插件表示。此外，Editor.js 还为开发者提供了许多现成的插件和一个用于创建新插件的简单 API。

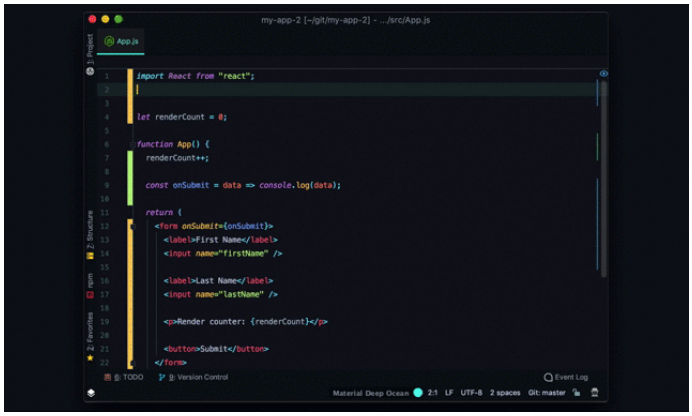


九、react-hook-form

> 📋 React hooks for forms validation without the hassle (Web + React Native)
>
> https://github.com/react-hook-form/react-hook-form

React Hook Form 是高性能、灵活、易拓展、易于使用的表单校验库。它支持以下特性：

- 使创建表单和集成更加便捷
- 非受控表单校验
- 以性能和开发体验为基础构建
- 迷你的体积而没有其他依赖
- 遵循 html 标准进行校验
- 与 React Native 兼容
- 支持 Yup, Joi, Superstruct 或自定义
- 支持浏览器原生校验



十、nest

> A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications on top of TypeScript & JavaScript (ES6, ES7, ES8) 🚀.
>
> https://github.com/nestjs/nest

Nest 是构建高效，可扩展的 Node.js Web 应用程序的框架。 它使用现代的 JavaScript 或 TypeScript（保留与纯 JavaScript 的兼容性），并结合 OOP（面向对象编程），FP（函数式编程）和 FRP（函数响应式编程）的元素。

在底层，Nest 使用了 Express，但也提供了与其他各种库的兼容，例如 Fastify，可以方便地使用各种可用的第三方插件。

近几年，由于 Node.js，JavaScript 已经成为 Web 前端和后端应用程序的「通用语言」，从而产生了像 Angular、React、Vue 等令人耳目一新的项目，这些项目提高了开发人员的生产力，使得可以快速构建可测试的且可扩展的前端应用程序。 然而，在服务器端，虽然有很多优秀的库、helper 和 Node 工具，但是它们都没有有效地解决主要问题 —— 架构。

Nest 旨在提供一个开箱即用的应用程序体系结构，允许轻松创建高度可测试，可扩展，松散耦合且易于维护的应用程序。

结尾

至此本书的内容已经介绍完了，非常感谢你的阅读。由于作者水平有限，书中可能会有一些描述不准确内容或出现一些错别字，请大家多多包涵。重学 TS 之路，阿宝哥与你同行，欢迎小伙们与我一起技术交流，共同学习进步。如果你还想学习 TS 方面的其它知识，可以关注阿宝哥的 “全栈修仙之路”，阅读近 50 篇的 TS 系列教程，还会不断更新哟。