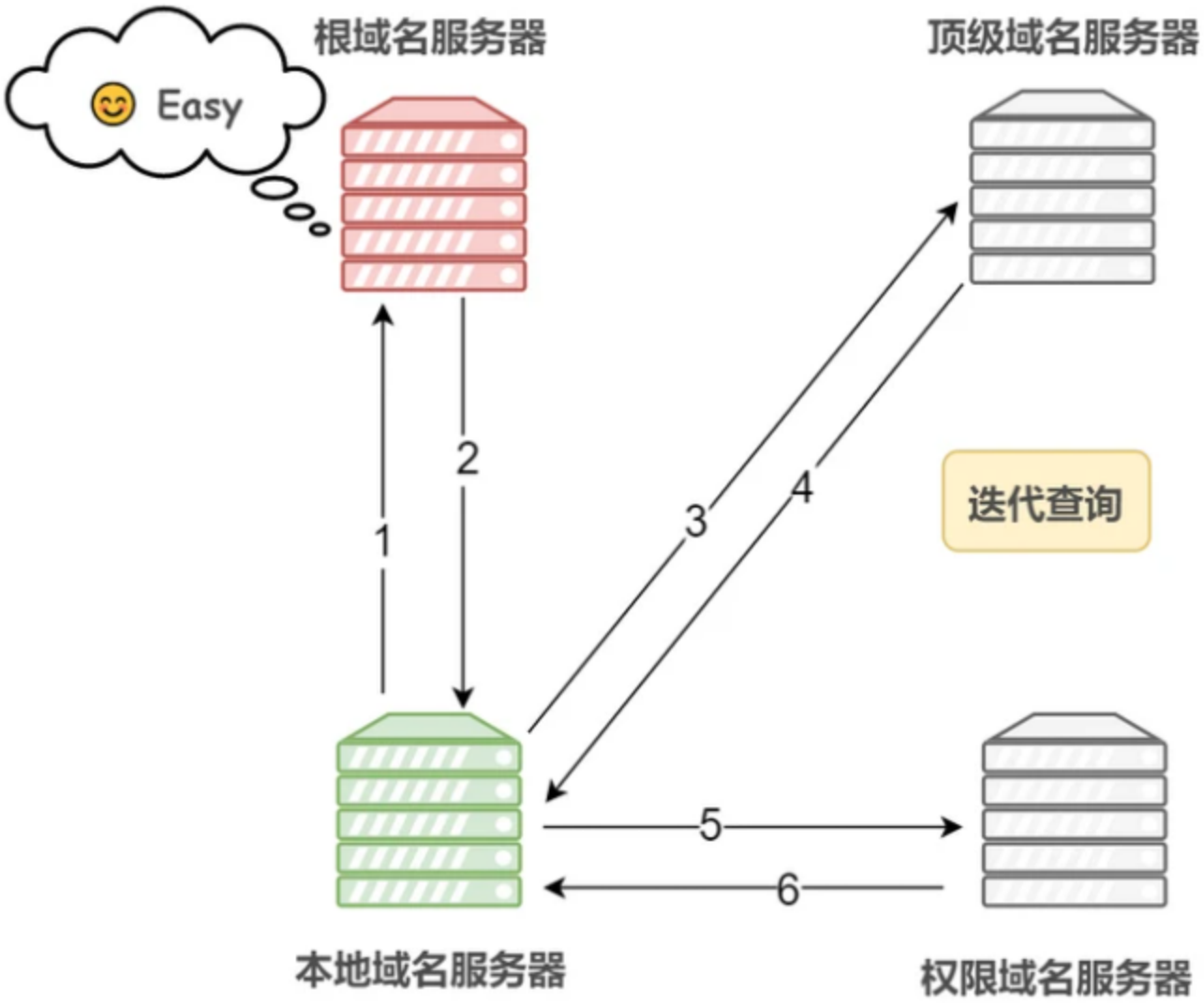


12.4. 域名缓存

在域名服务器解析的时候，使用缓存保存域名和 `IP` 地址的映射

计算机中 `DNS` 的记录也分成了两种缓存方式：

- 浏览器缓存：浏览器在获取网站域名的实际 IP 地址后会对其进行缓存，减少网络请求的损耗
- 操作系统缓存：操作系统的缓存其实是用户自己配置的 `hosts` 文件

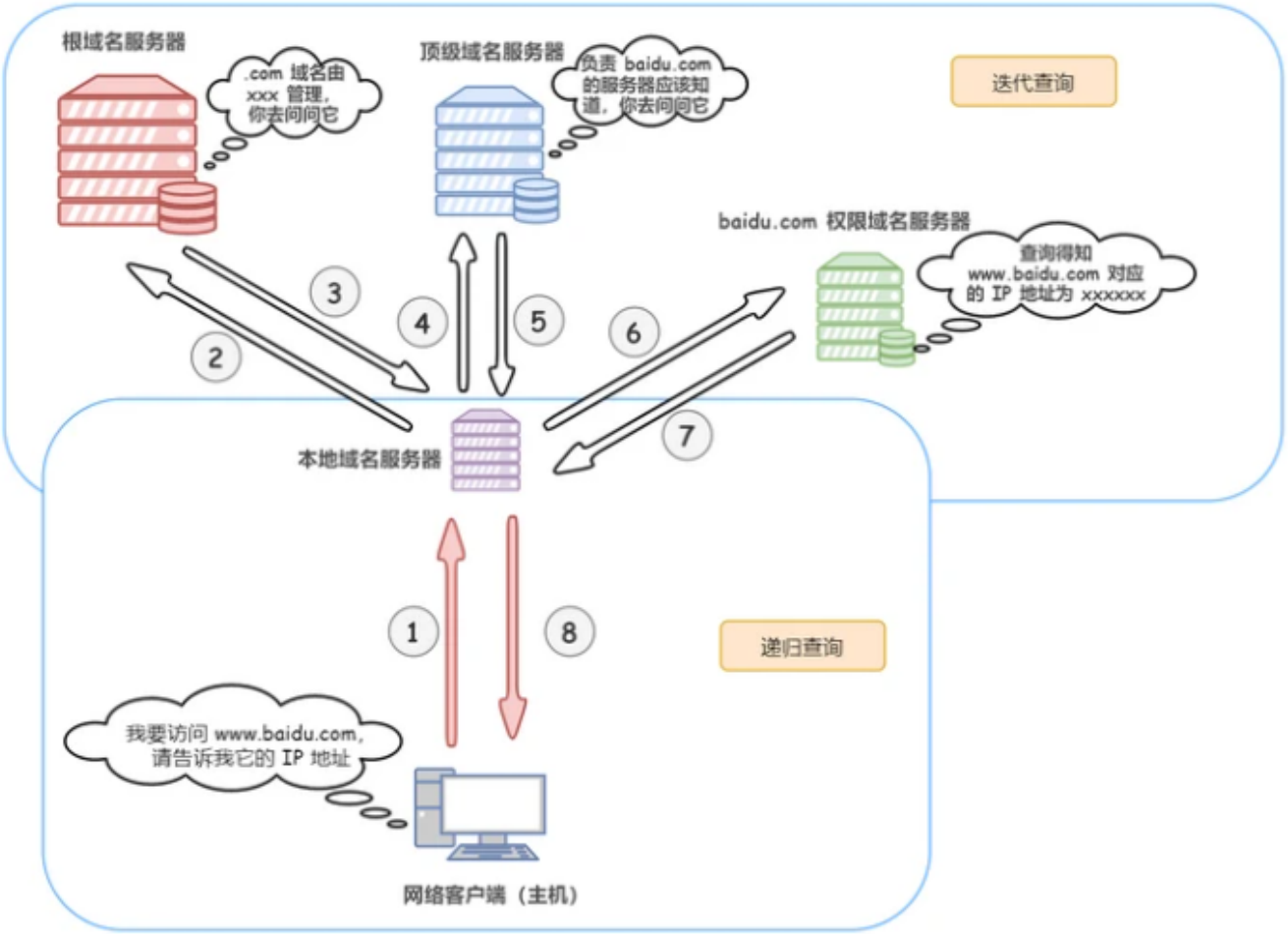
12.5. 查询过程

解析域名的过程如下：

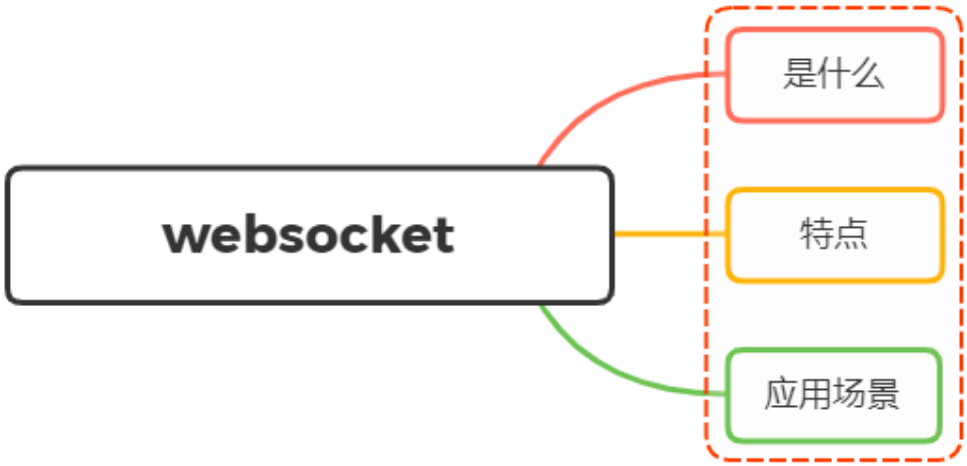
- 首先搜索浏览器的 DNS 缓存，缓存中维护一张域名与 IP 地址的对应表

- 若没有命中，则继续搜索操作系统的 DNS 缓存
- 若仍然没有命中，则操作系统将域名发送至本地域名服务器，本地域名服务器采用递归查询自己的 DNS 缓存，查找成功则返回结果
- 若本地域名服务器的 DNS 缓存没有命中，则本地域名服务器向上级域名服务器进行迭代查询
    - 首先本地域名服务器向根域名服务器发起请求，根域名服务器返回顶级域名服务器的地址给本地服务器
    - 本地域名服务器拿到这个顶级域名服务器的地址后，就向其发起请求，获取权限域名服务器的地址
    - 本地域名服务器根据权限域名服务器的地址向其发起请求，最终得到该域名对应的 IP 地址
- 本地域名服务器将得到的 IP 地址返回给操作系统，同时自己将 IP 地址缓存起来
- 操作系统将 IP 地址返回给浏览器，同时自己也将 IP 地址缓存起
- 至此，浏览器就得到了域名对应的 IP 地址，并将 IP 地址缓存起

流程如下图所示：



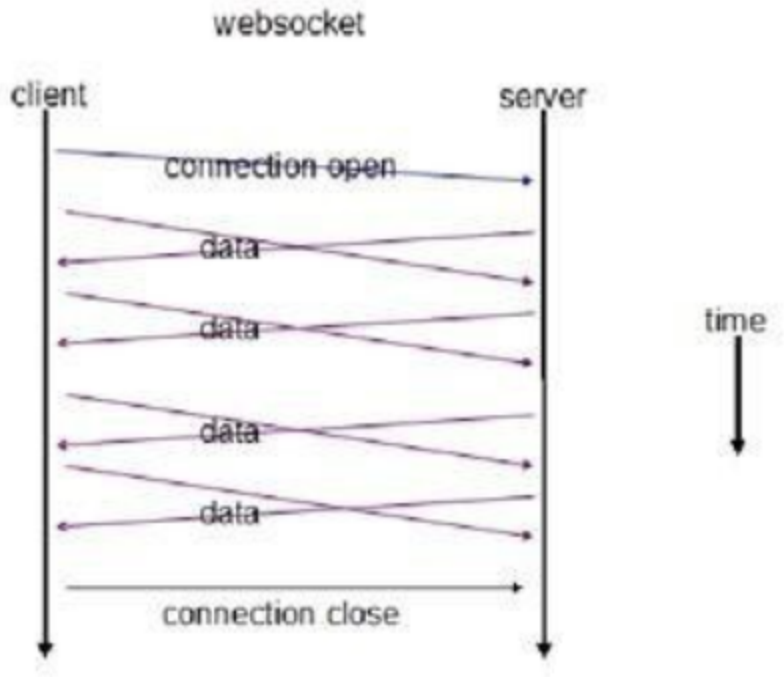
13. 说说对WebSocket的理解？应用场景？



13.1. 是什么

WebSocket，是一种网络传输协议，位于 `OSI` 模型的应用层。可在单个 `TCP` 连接上进行全双工通信，能更好的节省服务器资源和带宽并达到实时通迅

客户端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



从上图可见， `websocket` 服务器与客户端通过握手连接，连接成功后，两者都能主动的向对方发送或接受数据

而在 `websocket` 出现之前，开发实时 `web` 应用的方式为轮询

不停地向服务器发送 HTTP 请求，问有没有数据，有数据的话服务器就用响应报文回应。如果轮询的频率比较高，那么就可以近似地实现“实时通信”的效果

轮询的缺点也很明显，反复发送无效查询请求耗费了大量的带宽和 `CPU` 资源

13.2. 特点

13.2.1. 全双工

通信允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合

例如指 A→B 的同时 B→A ，是瞬时同步的

13.2.2. 二进制帧

采用了二进制帧结构，语法、语义与 HTTP 完全不兼容，相比 `http/2` ， `WebSocket` 更侧重于“实时通信”，而 `HTTP/2` 更侧重于提高传输效率，所以两者的帧结构也有很大的区别

不像 `HTTP/2` 那样定义流，也就不存在多路复用、优先级等特性

自身就是全双工，也不需要服务器推送

13.2.3. 协议名

引入 `ws` 和 `wss` 分别代表明文和密文的 `websocket` 协议，且默认端口使用80或443，几乎与 `http` 一致

HTTP | 复制代码

```
ws://www.chrono.com
ws://www.chrono.com:8080/srv
wss://www.chrono.com:445/im?user_id=xxx
```

13.2.4. 握手

`WebSocket` 也要有一个握手过程，然后才能正式收发数据

客户端发送数据格式如下：

```http
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

- Connection：必须设置Upgrade，表示客户端希望连接升级
- Upgrade：必须设置Websocket，表示希望升级到Websocket协议
- Sec-WebSocket-Key：客户端发送的一个 base64 编码的密文，用于简单的认证秘钥。要求服务端必须返回一个对应加密的“Sec-WebSocket-Accept应答，否则客户端会抛出错误，并关闭连接
- Sec-WebSocket-Version ： 表示支持的Websocket版本

服务端返回的数据格式：

```http
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=Sec-WebSocket-Protocol: chat
```

- HTTP/1.1 101 Switching Protocols：表示服务端接受 WebSocket 协议的客户端连接
- Sec-WebSocket-Accep：验证客户端请求报文，同样也是为了防止误连接。具体做法是把请求头里“Sec-WebSocket-Key”的值，加上一个专用的 UUID，再计算摘要

13.2.5. 优点

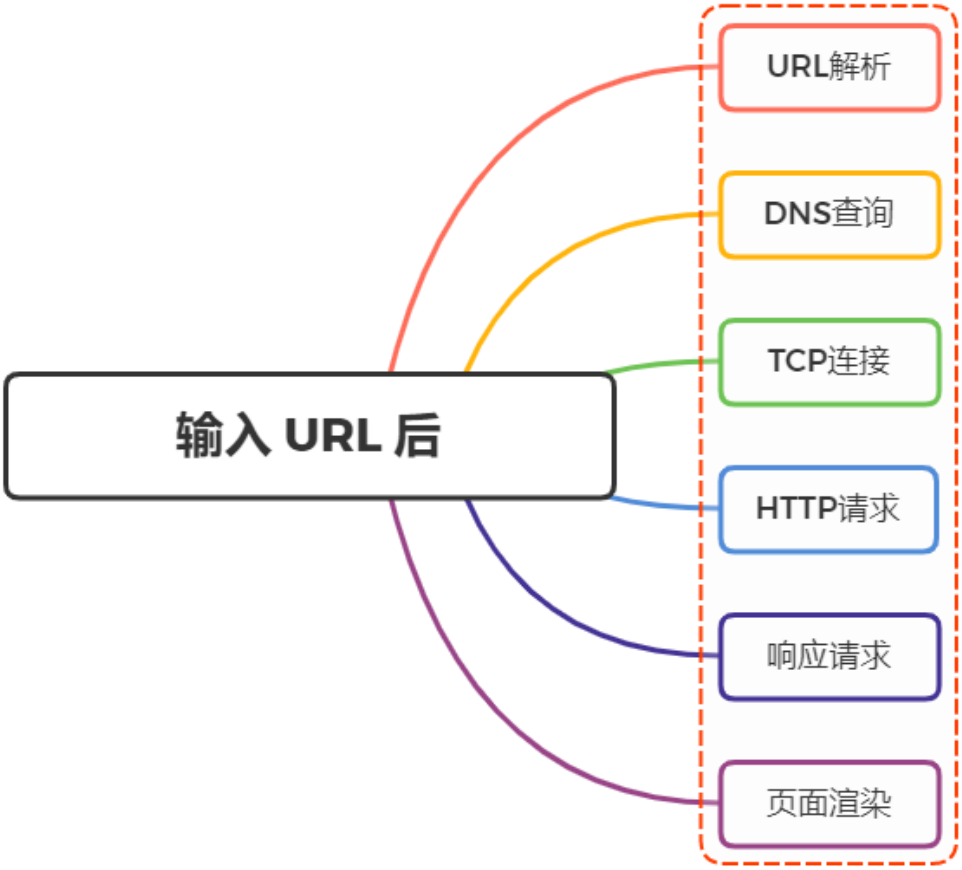
- 较少的控制开销：数据包头部协议较小，不同于http每次请求需要携带完整的头部
- 更强的实时性：相对于HTTP请求需要等待客户端发起请求服务端才能响应，延迟明显更少
- 保持创连接状态：创建通信后，可省略状态信息，不同于HTTP每次请求需要携带身份验证
- 更好的二进制支持：定义了二进制帧，更好处理二进制内容
- 支持扩展：用户可以扩展websocket协议、实现部分自定义的子协议
- 更好的压缩效果：Websocket在适当的扩展支持下，可以沿用之前内容的上下文，在传递类似的数据时，可以显著地提高压缩率

13.3. 应用场景

基于 `websocket` 的事实通信的特点，其存在的应用场景大概有：

- 弹幕
- 媒体聊天
- 协同编辑
- 基于位置的应用
- 体育实况更新
- 股票基金报价实时更新

14. 说说地址栏输入 URL 敲下回车后发生了什么?



14.1. 简单分析

简单的分析，从输入 `URL` 到回车后发生的行为如下：

- URL解析

- DNS 查询
- TCP 连接
- HTTP 请求
- 响应请求
- 页面渲染

14.2. 详细分析

14.2.1. URL解析

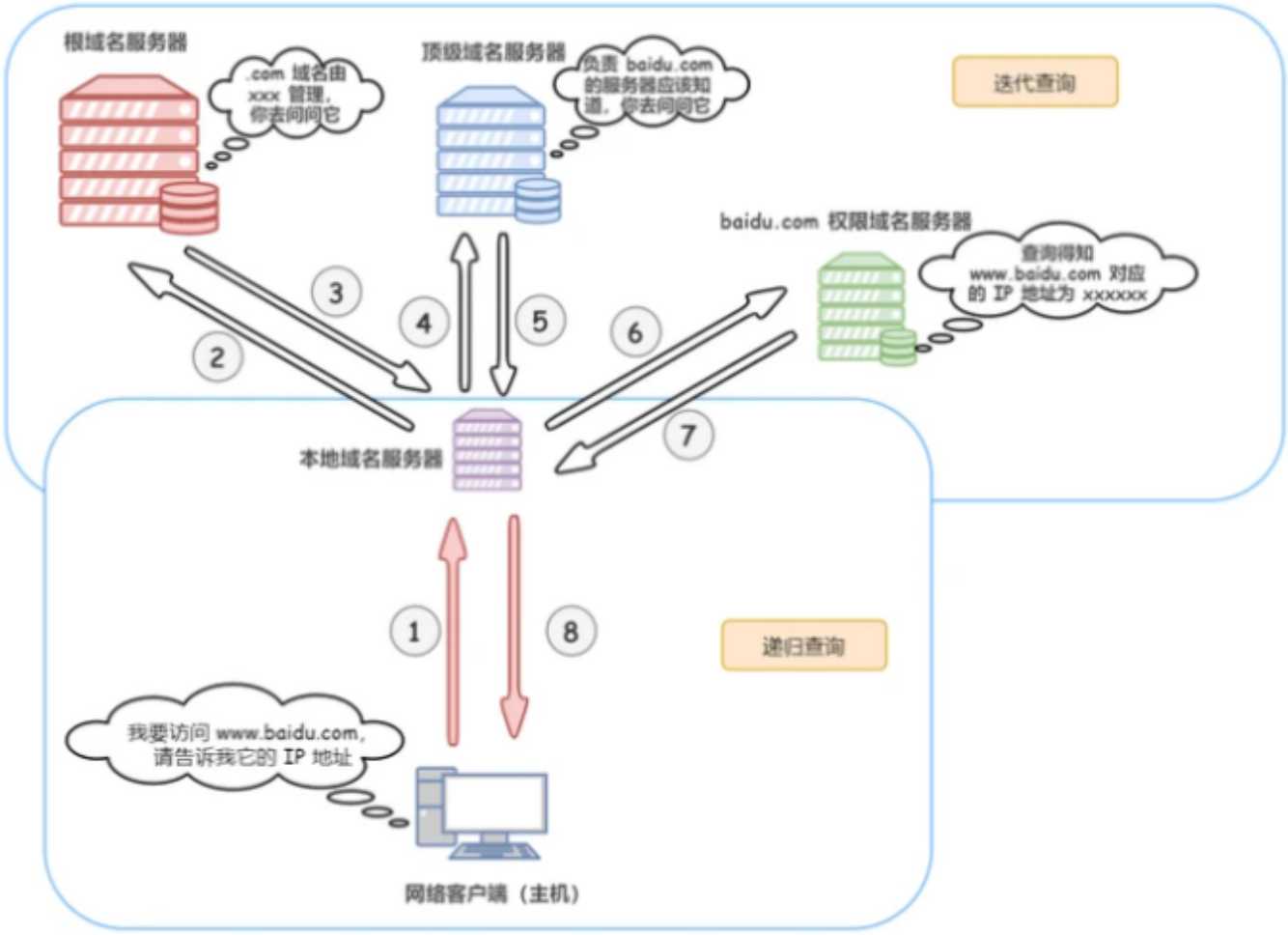
首先判断你输入的是一个合法的 `URL` 还是一个待搜索的关键词，并且根据你输入的内容进行对应操作

`URL` 的解析第过程中的第一步，一个 `url` 的结构解析如下：

14.2.2. DNS查询

在之前文章中讲过 `DNS` 的查询，这里就不再讲述了

整个查询过程如下图所示：

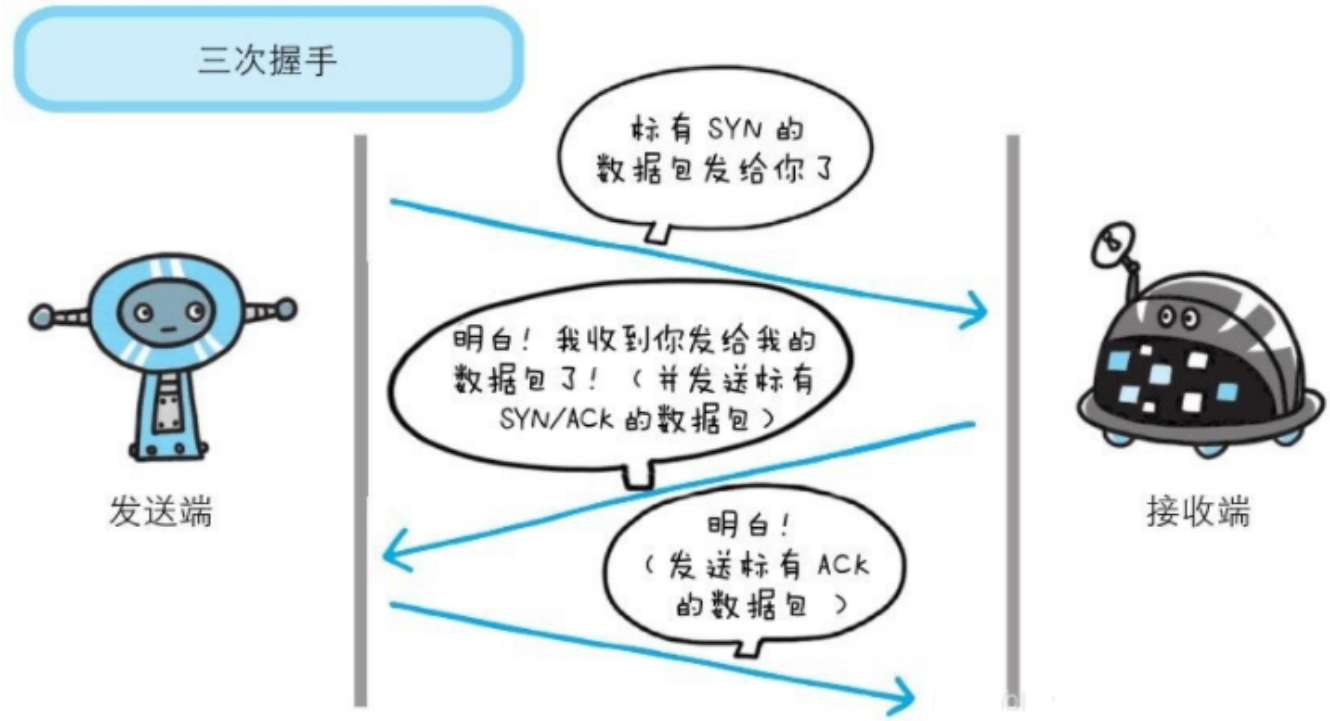


最终，获取到了域名对应的目标服务器 `IP` 地址

14.2.3. TCP连接

在之前文章中，了解到 `tcp` 是一种面向有连接的传输层协议

在确定目标服务器服务器的 `IP` 地址后，则经历三次握手建立 `TCP` 连接，流程如下：



14.2.4. 发送 http 请求

当建立 `tcp` 连接之后，就可以在这基础上进行通信，浏览器发送 `http` 请求到目标服务器

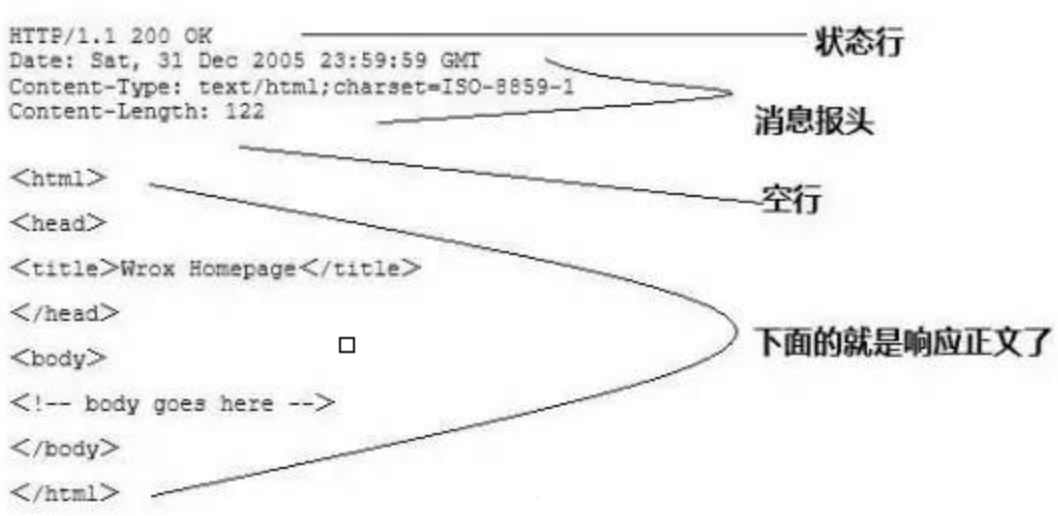
请求的内容包括：

- 请求行
- 请求头
- 请求主体

14.2.5. 响应请求

当服务器接收到浏览器的请求之后，就会进行逻辑操作，处理完成之后返回一个 `HTTP` 响应消息，包括：

- 状态行
- 响应头
- 响应正文



在服务器响应之后，由于现在 `http` 默认开始长连接 `keep-alive` ，当页面关闭之后， `tcp` 链接则会经过四次挥手完成断开

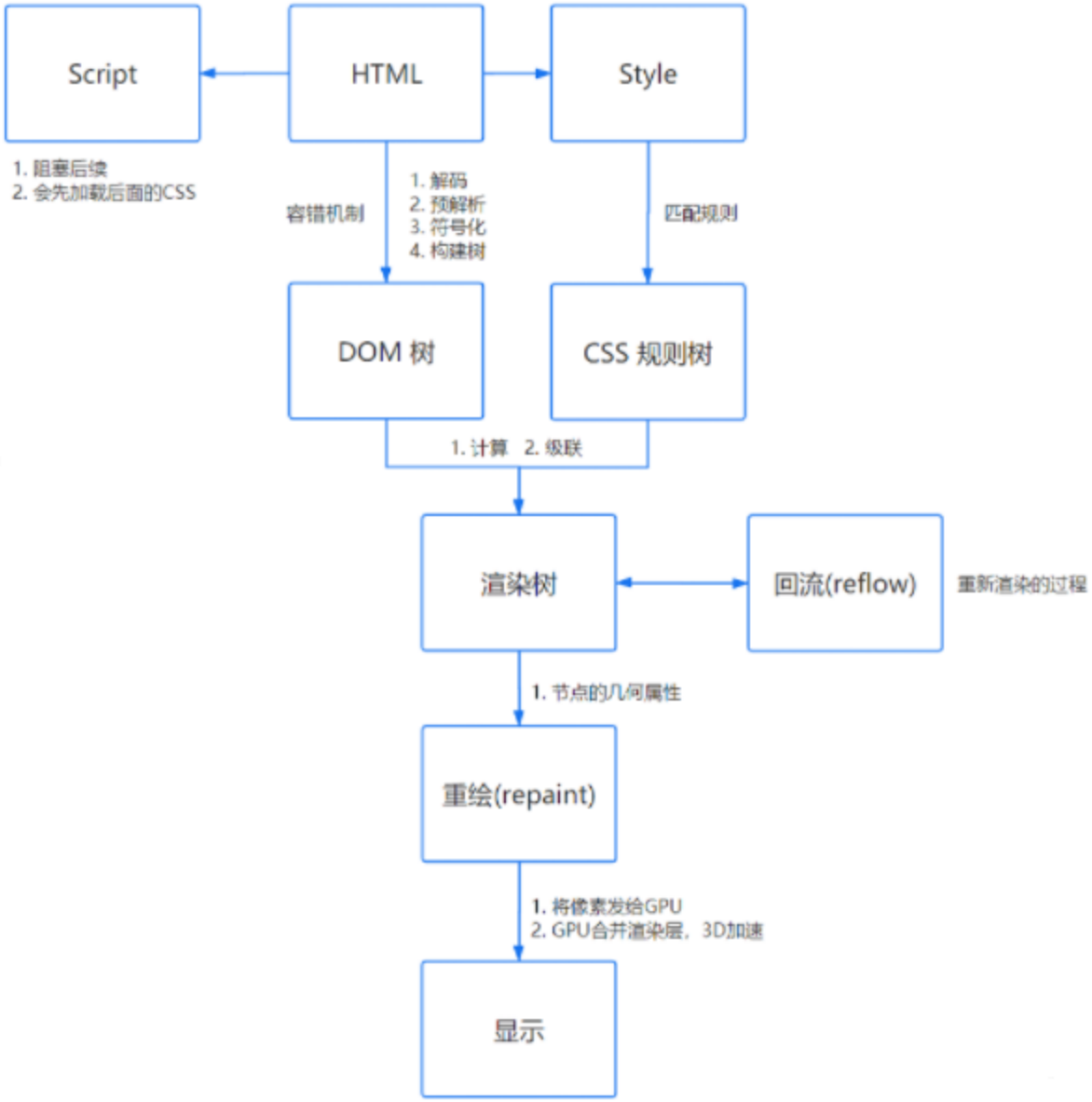
14.2.6. 页面渲染

当浏览器接收到服务器响应的资源后，首先会对资源进行解析：

- 查看响应头的信息，根据不同的指示做对应处理，比如重定向，存储cookie，解压gzip，缓存资源等等
- 查看响应头的 Content-Type的值，根据不同的资源类型采用不同的解析方式

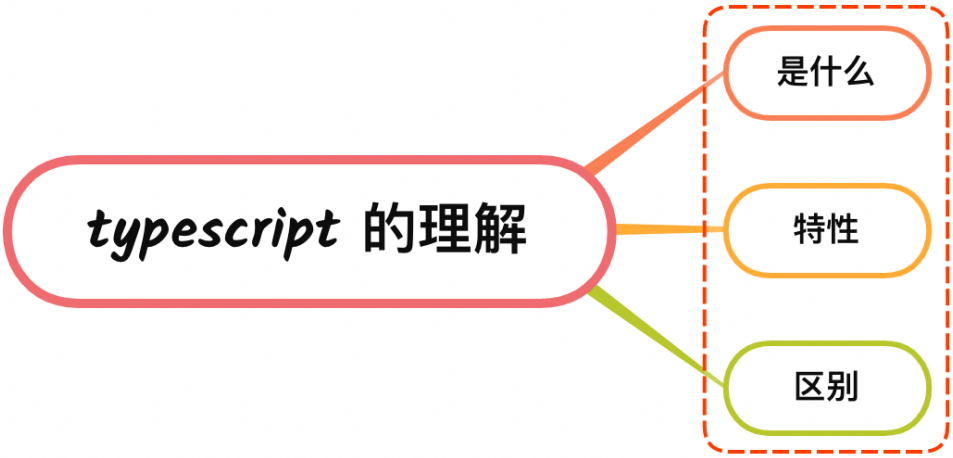
关于页面的渲染过程如下：

- 解析HTML，构建 DOM 树
- 解析 CSS ，生成 CSS 规则树
- 合并 DOM 树和 CSS 规则，生成 render 树
- 布局 render 树（ Layout / reflow ），负责各元素尺寸、位置的计算
- 绘制 render 树（ paint ），绘制页面像素信息
- 浏览器会将各层的信息发送给 GPU，GPU 会将各层合成（ composite ），显示在屏幕上


Script
HTML
Style
1. 阻塞后续
2. 会先加载后面的CSS
容错机制
1. 解码
2. 预解析
3. 符号化
4. 构建树
匹配规则
DOM 树
CSS 规则树
1. 计算 2. 级联
渲染树
回流(reflow)
重新渲染的过程
1. 节点的几何属性
重绘(repaint)
1. 将像素发给GPU
2. GPU合并渲染层，3D加速
显示

Typescript面试真题(12题)

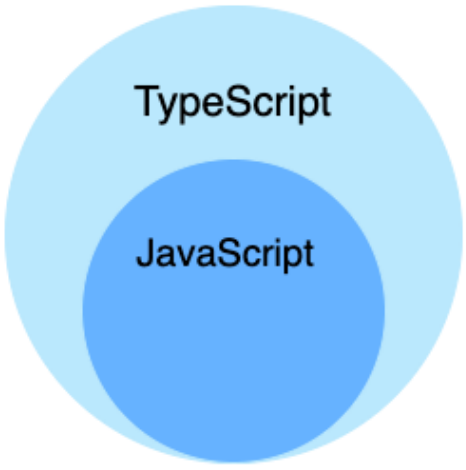
1. 说说你对 TypeScript 的理解？与 JavaScript 的区别？

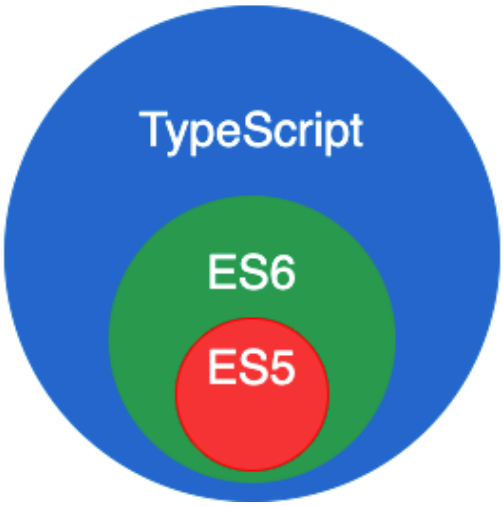


1.1. 是什么

`TypeScript` 是 `JavaScript` 的类型的超集，支持 `ES6` 语法，支持面向对象编程的概念，如类、接口、继承、泛型等

> 超集，不得不说另外一个概念，子集，怎么理解这两个呢，举个例子，如果一个集合 A 里面的的所有元素集合 B 里面都存在，那么我们可以理解集合 B 是集合 A 的超集，集合 A 为集合 B 的子集





其是一种静态类型检查的语言，提供了类型注解，在代码编译阶段就可以检查出数据类型的错误

同时扩展了 `JavaScript` 的语法，所以任何现有的 `JavaScript` 程序可以不加改变的在 `TypeScript` 下工作

为了保证兼容性， `TypeScript` 在编译阶段需要编译器编译成纯 `JavaScript` 来运行，是为大型应用之开发而设计的语言，如下：

`ts` 文件如下：

TypeScript | 复制代码

```typescript
const hello: string = "Hello World!";
console.log(hello);
```

编译文件后：

JavaScript | 复制代码

```javascript
const hello = "Hello World!";
console.log(hello);
```

1.2. 特性

`TypeScript` 的特性主要有如下：

- **类型批注和编译时类型检查** ： 在编译时批注变量类型
- **类型推断**：ts 中没有批注变量类型会自动推断变量的类型
- **类型擦除**：在编译过程中批注的内容和接口会在运行时利用工具擦除
- **接口**：ts 中用接口来定义对象类型
- **枚举**：用于取值被限定在一定范围内的场景
- **Mixin**：可以接受任意类型的值
- **泛型编程**：写代码时使用一些以后才指定的类型
- **名字空间**：名字只在该区域内有效，其他区域可重复使用该名字而不冲突
- **元组**：元组合并了不同类型的对象，相当于一个可以装不同类型数据的数组
- ...

1.2.1. 类型批注

通过类型批注提供在编译时启动类型检查的静态类型，这是可选的，而且可以忽略而使用 `JavaScript` 常规的动态类型

```tsx
function Add(left: number, right: number): number {
  return left + right;
}
```

对于基本类型的批注是 `number` 、 `bool` 和 `string` ，而弱或动态类型的结构则是 `any` 类型

1.2.2. 类型推断

当类型没有给出时，TypeScript 编译器利用类型推断来推断类型，如下：

```typescript
let str = "string";
```

变量 `str` 被推断为字符串类型，这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时

如果缺乏声明而不能推断出类型，那么它的类型被视作默认的动态 `any` 类型

1.2.3. 接口

接口简单来说就是用来描述对象的类型 数据的类型有 `number` 、 `null` 、 `string` 等数据格式，对象的类型就是用接口来描述的

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

let tom: Person = {
  name: "Tom",
  age: 25,
};
```

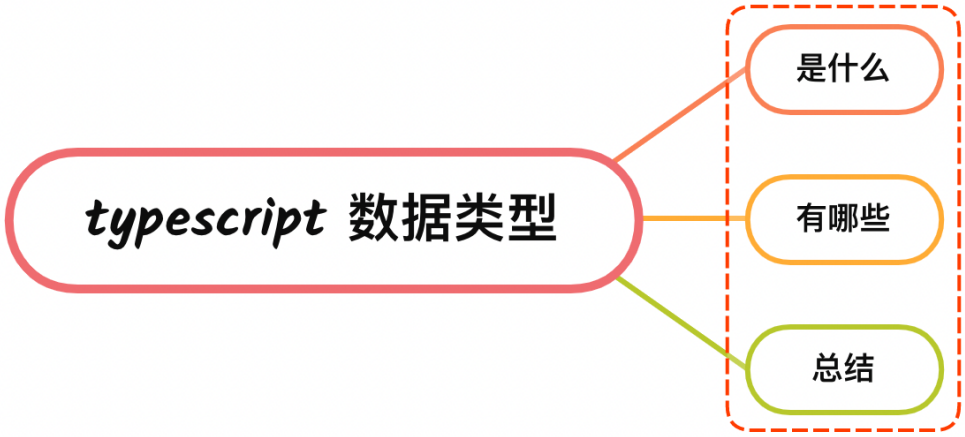
1.3. 区别

- TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法
- TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译
- TypeScript 文件的后缀名 .ts （.ts，.tsx，.dts），JavaScript 文件是 .js
- 在编写 TypeScript 的文件的时候就会自动编译成 js 文件

更多的区别如下图所示：

	JavaScript	TypeScript
语言	脚本语言	面向对象编程语言
学习难度	灵活易学	需要有脚本编程经验
类型	轻量级解释编程语言	强类型的面向对象编程语言
客户端/服务	客户端服务端都有	侧重客户端
拓展名	.js	.ts 或 .tsx
耗时	更快	编译代码需要些时间
数据绑定	没有类型和接口的概念	使用类型和接口表示数据
语法	所有的语句都写在脚本标签内。浏览器将脚本标签内的文本识别为脚本	一个 TypeScript 程序由模块、方法、变量、语句、表达式和注释构成
静态类型	JS 中没有静态类型的概念	支持静态类型
模块支持	不支持模块	支持模块
接口	没有接口	支持接口
可选参数方法	不支持	支持
原型	没有这种特性	支持原型特性

2. 说说 typescript 的数据类型有哪些?



2.1. 是什么

`typescript` 和 `javascript` 几乎一样，拥有相同的数据类型，另外在 `javascript` 基础上提供了更加实用的类型供开发使用

在开发阶段，可以为明确的变量定义为某种类型，这样 `typescript` 就能在编译阶段进行类型检查，当类型不合符预期结果的时候则会出现错误提示

2.2. 有哪些

`typescript` 的数据类型主要有如下：

- boolean（布尔类型）
- number（数字类型）
- string（字符串类型）
- array（数组类型）
- tuple（元组类型）
- enum（枚举类型）
- any（任意类型）
- null 和 undefined 类型
- void 类型
- never 类型
- object 对象类型

2.2.1. boolean

布尔类型

```tsx
let flag:boolean = true;
// flag = 123; // 错误
flag = false;  //正确
```

2.2.2. number

数字类型，和 `javascript` 一样， `typescript` 的数值类型都是浮点数，可支持二进制、八进制、十进制和十六进制

```tsx
let num:number = 123;
// num = '456'; // 错误
num = 456;  //正确
```

进制表示：

```tsx
let decLiteral: number = 6; // 十进制
let hexLiteral: number = 0xf00d; // 十六进制
let binaryLiteral: number = 0b1010; // 二进制
let octalLiteral: number = 0o744; // 八进制
```

2.2.3. string

字符串类型，和 `JavaScript` 一样，可以使用双引号（ `"` ）或单引号（ `'` ）表示字符串

```tsx
let str:string = 'this is ts';
str = 'test';
```

作为超集，当然也可以使用模版字符串``进行包裹，通过 ${} 嵌入变量

```tsx
let name: string = `Gene`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ name }
```

2.2.4. array

数组类型，跟 `javascript` 一致，通过 `[]` 进行包裹，有两种写法：

方式一：元素类型后面接上 `[]`

```tsx
 let arr:string[] = ['12', '23'];
 arr = ['45', '56'];
```

方式二：使用数组泛型， `Array<元素类型>` ：

```tsx
let arr:Array<number> = [1, 2];
arr = ['45', '56'];
```

2.2.5. tuple

元祖类型，允许表示一个已知元素数量和类型的数组，各元素的类型不必相同

```tsx
let tupleArr:[number, string, boolean];
tupleArr = [12, '34', true]; //ok
typleArr = [12, '34'] // no ok
```

赋值的类型、位置、个数需要和定义（生明）的类型、位置、个数一致

2.2.6. enum

`enum` 类型是对JavaScript标准数据类型的一个补充，使用枚举类型可以为一组数值赋予友好的名字

```tsx
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

2.2.7. any

可以指定任何类型的值，在编程阶段还不清楚类型的变量指定一个类型，不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查，这时候可以使用 `any` 类型

使用 `any` 类型允许被赋值为任意类型，甚至可以调用其属性、方法

```tsx
let num:any = 123;
num = 'str';
num = true;
```

定义存储各种类型数据的数组时，示例代码如下：

```tsx
let arrayList: any[] = [1, false, 'fine'];
arrayList[1] = 100;
```

2.2.8. null 和 和 undefined

在 `JavaScript` 中 `null` 表示 "什么都没有"，是一个只有一个值的特殊类型，表示一个空对象引用，而 `undefined` 表示一个没有设置值的变量

默认情况下 `null` 和 `undefined` 是所有类型的子类型， 就是说你可以把 `null` 和 `undefined` 赋值给 `number` 类型的变量

```tsx
let num:number | undefined; // 数值类型 或者 undefined
console.log(num); // 正确
num = 123;
console.log(num); // 正确
```

但是 `ts` 配置了 `--strictNullChecks` 标记， `null` 和 `undefined` 只能赋值给 `void` 和它们各自

2.2.9. void

用于标识方法返回值的类型，表示该方法没有返回值。

```tsx
function hello(): void {
    alert("Hello Runoob");
}
```

2.2.10. never

`never` 是其他类型 （包括 `null` 和 `undefined` ）的子类型，可以赋值给任何类型，代表从不会出现的值

但是没有类型是 never 的子类型，这意味着声明 `never` 的变量只能被 `never` 类型所赋值。

`never` 类型一般用来指定那些总是会抛出异常、无限循环

```tsx
let a:never;
a = 123; // 错误的写法

a = (() => { // 正确的写法
  throw new Error('错误');
})()

// 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);
}
```

2.2.11. object

对象类型，非原始类型，常见的形式通过 `{}` 进行包裹

```tsx
let obj:object;
obj = {name: 'Wang', age: 25};
```

2.3. 总结

和 `javascript` 基本一致，也分成：

- 基本类型
- 引用类型

在基础类型上， `typescript` 增添了 `void` 、 `any` 、 `emum` 等原始类型

3. 说说你对 TypeScript 中高级类型的理解？有哪些？

3.1. 是什么

除了 `string` 、 `number` 、 `boolean` 这种基础类型外，在 `typescript` 类型声明中还存在一些高级的类型应用

这些高级类型，是 `typescript` 为了保证语言的灵活性，所使用的一些语言特性。这些特性有助于我们应对复杂多变的开发场景

3.2. 有哪些

常见的高级类型有如下：

- 交叉类型
- 联合类型
- 类型别名