Web 应用开发是怎么一步一步演变的?

前端印象 2022-06-24 22:09 发表于上海

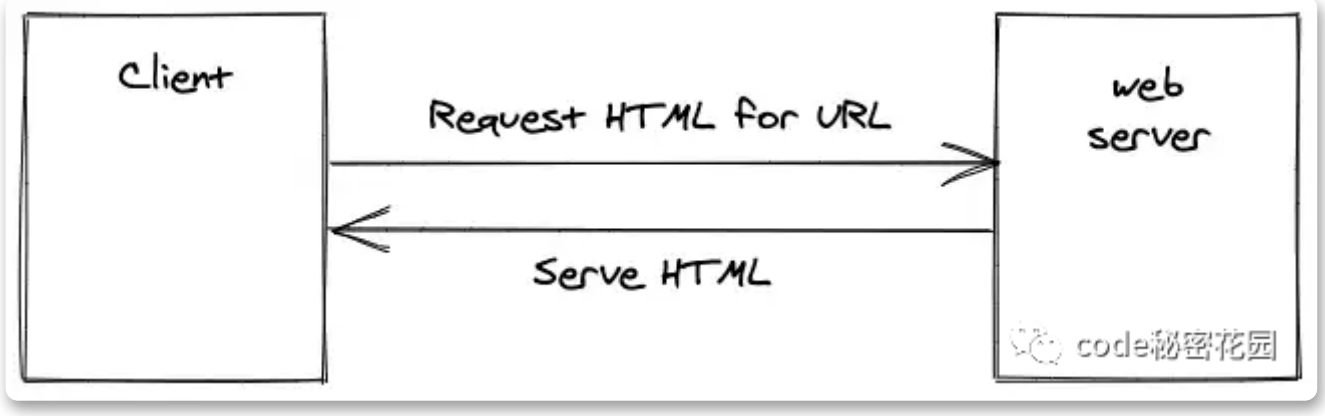
出处：code秘密花园

大家有时候有没有仔细想过，我们开发一个网站，本质上是在做什么呢？ **Web** 开发从刀耕火种的 **HTML** 时代，到现代的 **Web** 开发模式，巨鲸发生了怎么样的演变呢？我今天就带大家开一起看一下 ~

本文译自：https://www.robinwieruch.de/web-applications/

传统网站

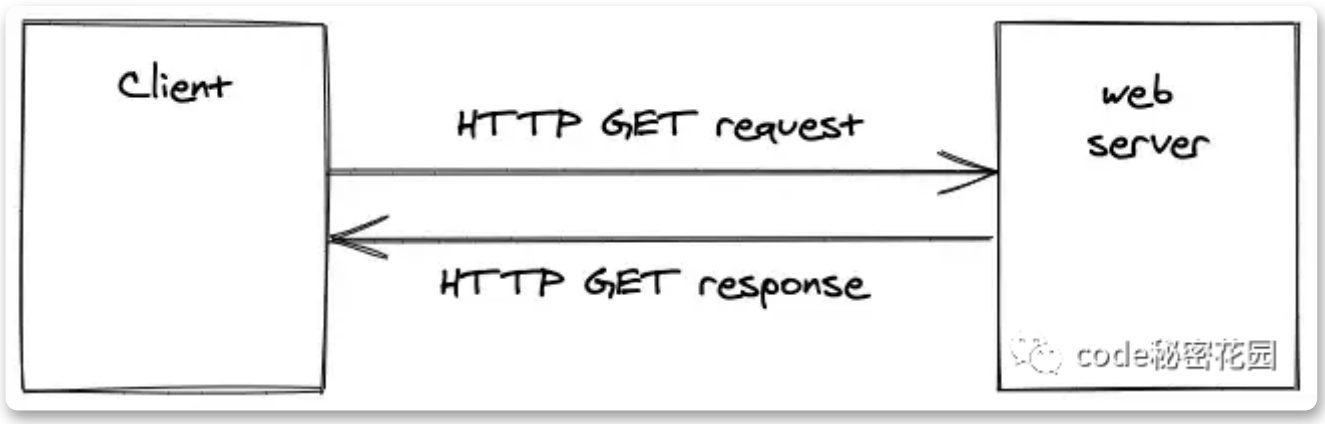
如果你刚刚开始学习 **Web** 开发，你很可能会从使用 **HTML** 开始。我们编写一个仅带有 **HTML** 的网站，没有样式（**CSS**）且没有任何逻辑（**JavaScript**）。



如果你在笔记本电脑或智能手机上的浏览器中导航到特定的 **URL**，浏览器会向负责该 **URL** 的 **Web** 服务器发出请求。如果 **Web** 服务器能够将请求与网站匹配，它会将网站的 **HTML** 文件返回给你的浏览器。

为了将网站传输到浏览器， 客户端和 **Web** 服务器之间请求和响应的通信工作由 **HTTP** 协议来承担。这就是为什么每个 **URL** 前面都有一个 **“http”**。

客户端和服务器之间的通信是异步的，这意味着你的网站不会立即就显示出来。从客户端向 **Web** 服务器发送请求、从 **Web** 服务器向客户端发送响应都需要一定时间。



HTTP 请求带有四种基本的 **HTTP** 方法，我想在这里处理它们：**GET、POST、PUT、DELETE** 。 **GET** 方法通常用于读取资源，其余方法通常用于写入资源 — 其中资源可以是从 **HTML** 到 **JSON** 的任何内容。所有四种方法都可以抽象为臭名昭著的 **CRUD** 操作：创建、读取、更新、删除。

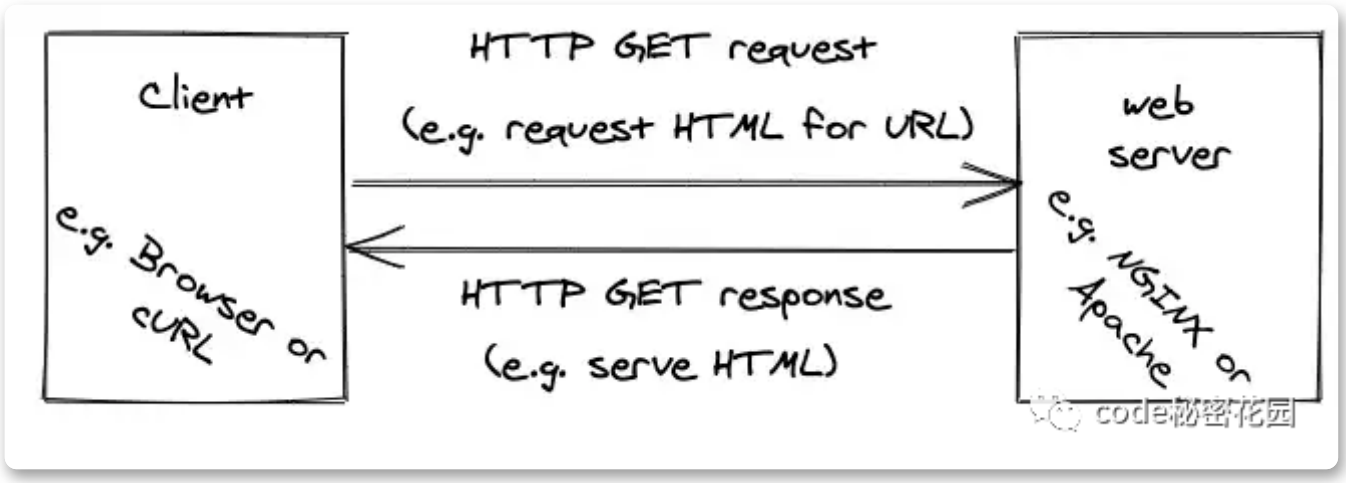
```
创建 -> HTTP POST
读取 -> HTTP GET
更新 -> HTTP PUT
删除 -> HTTP DELETE
```

在我们上面的网站示例中，通过访问浏览器中的 **URL** 从 **Web** 服务器向客户端提供服务，浏览器执行 **HTTP GET** 方法从 **Web** 服务器读取 **HTML** 文件。

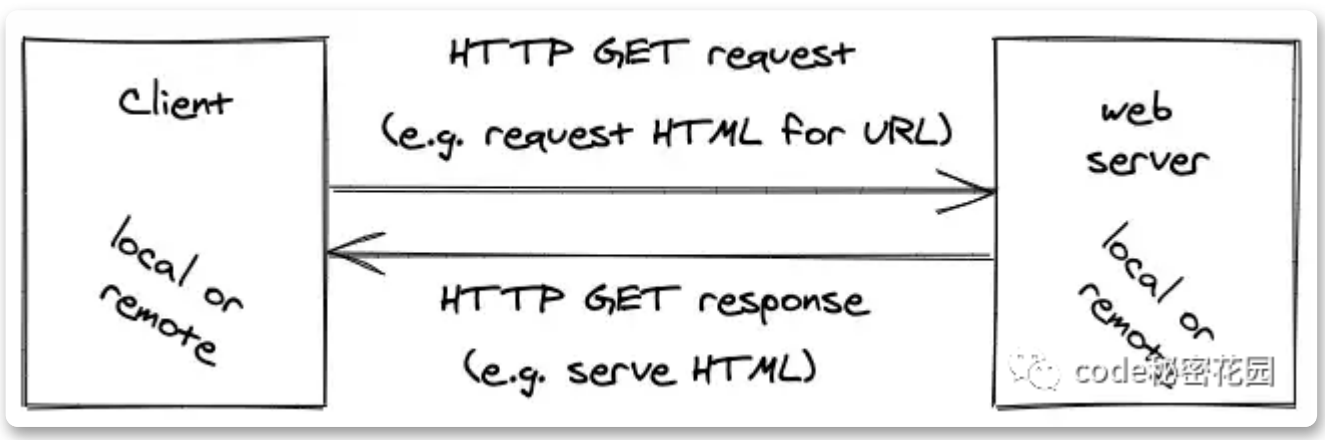
客户端和服务器有啥区别？

客户端是使用服务器的实体。它要么从服务器读取资源，要么将资源写入服务器。对于传统网站，客户端就是你的浏览器。如果你在浏览器中导航到特定的 URL，你的浏览器会与服务器通信以请求资源（例如 HTML）来为你显示网站。越过传统网站的思维，客户端其实也不一定是浏览器（例如 cURL）。

服务器是为客户端服务的实体。在传统意义上的网站中，服务器就是负责对客户端的请求做出反应的；要么回复来自 HTTP GET 请求的资源（例如 HTML、CSS、JavaScript），要么确认来自 HTTP POST、PUT、DELETE 请求的操作。作为一种特定类型的服务器，NGINX 或 Apache 都是现在比较流行的 Web 服务器。



可以说没有服务器就没有客户端，没有客户端就没有服务器。他们不需要在同一个位置也可以互相协作。例如，当你机器上的浏览器位于本地位置（例如北京）时，为网站提供服务的 Web 服务器也可以在一个远程位置（例如上海）。服务器 — 它只是另一台计算机，通常位于本地计算机之外的其他地方。为了开发一个服务器，你也可以在本地计算机上拥有一个服务器（localhost）。



Web 服务器和应用服务器有啥区别？

Web 服务器一般用于提供资源（例如 HTML、CSS、JavaScript），这些资源是可以通过 HTTP 传输的格式。当客户端从 Web 服务器请求资源时，Web 服务器通过将资源发送回客户端来满足请求。资源只是此服务器上的文件。如果一个 HTML 被发送到客户端，客户端（浏览器）会负责解析这个 HTML 然后把它渲染出来。

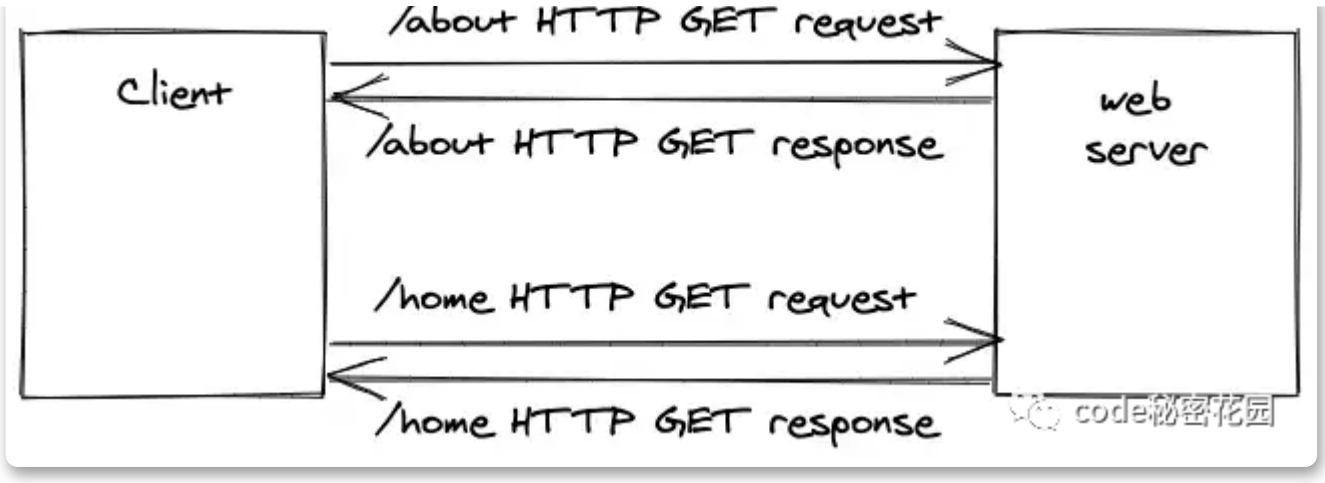
相比之下，应用服务器一般用于提供 HTML、CSS、JavaScript 之外的资源，例如 JSON 。Web 服务器主要使用 HTTP 协议，而应用服务器也可以使用其他协议（例如用于实时通信的 WebSockets ）。最重要的是，应用服务器可以在其服务端以特定编程语言（例如 JavaScript 与 Node.js 、PHP、Java、Ruby、C#、Go、Rust、Python ）编写特定逻辑。

Web 服务器和应用程序服务器都可以归类为服务器。所以一般谈到服务器，可能指的是这两者之一。但是，人们也会将服务器称为物理计算机，它运行在远程某处，它可以用作 Web 服务器或应用程序服务器。

还有两个术语可能会出现：部署（deploying）和托管（hosting）。我们简单理解一下：部署描述了在服务器上运行网站的行为，托管描述的是在服务器上持续为网站提供服务的行为。这就是为什么在你的电脑上开发一个网站时，你必须用 URL localhost 打开它，这只意味着你是这个网站的本地主机。

我们更改了 URL 路径会发生啥？

如果用户通过 URL 访问网站并在此域（例如 conardli.top ）上从路径（例如 /about ）导航到路径（/home ）会发生什么？在传统网站中，对于每个不同的 URL，都会从客户端向 Web 服务器发出一个新请求。



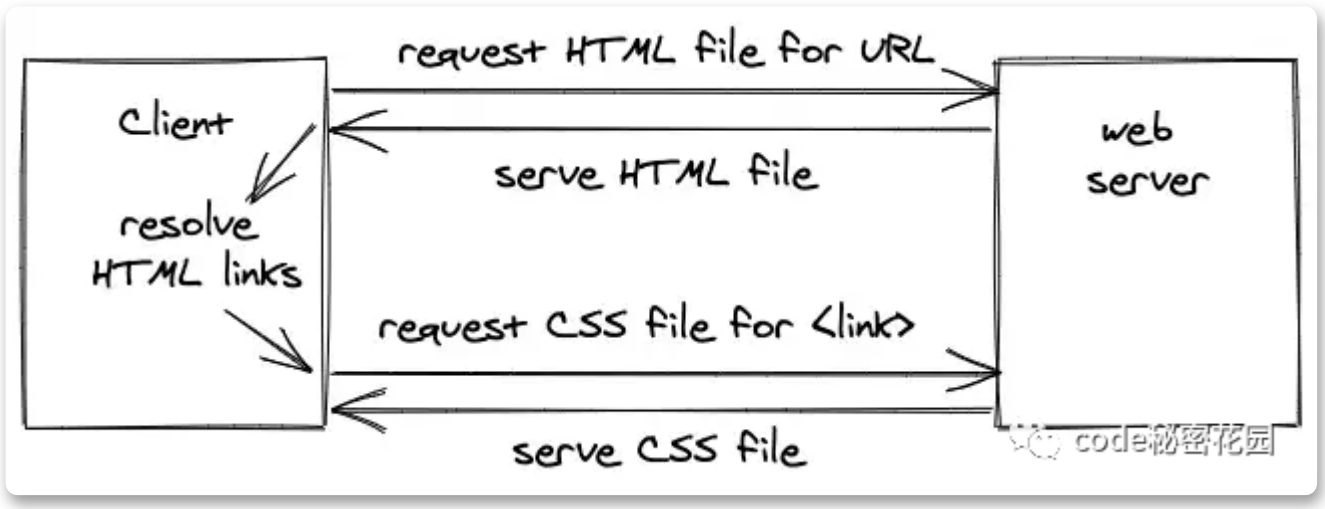
对于每个 URL，都会将不同的 HTTP GET 方法发送到专用 Web 服务器来完成请求。例如，当用户通过浏览器中的 /about 路径(也称为页面或路由)访问一个网站时，例如 http://www.conardli.top/about，Web 服务器将关于这个 URL 的所有信息发送回浏览器。这种行为称为服务器端路由，因为服务器决定在每个 URL 上将哪些资源发送给客户端。

当我的网站不仅仅是 HTML 时会发生啥?

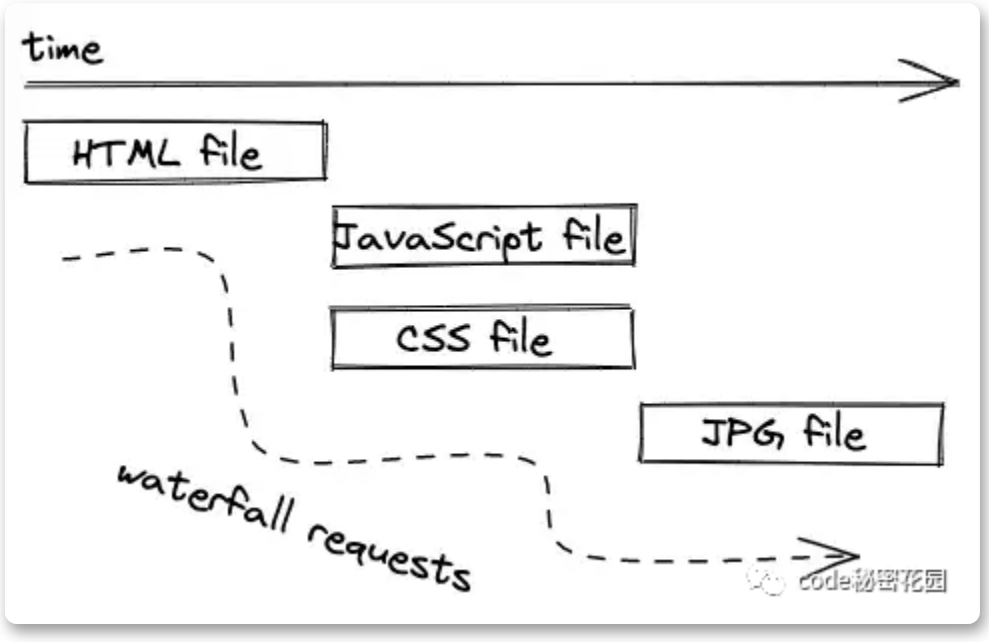
现代网站一般由 HTML（结构）、CSS（样式）和 JavaScript（逻辑）组成。没有 CSS，网站就不会有华丽的 UI，没有 JavaScript，网站不会有动态交互的能力。通常在使用 CSS 和 JavaScript 文件时，它们会被链接在一个 HTML 文件中:

```
link href="/media/examples/link-element-example.css" rel="stylesheet">

<h1>Home at /home route</p>

<p class="danger">Hello World code秘密花园</p>
```

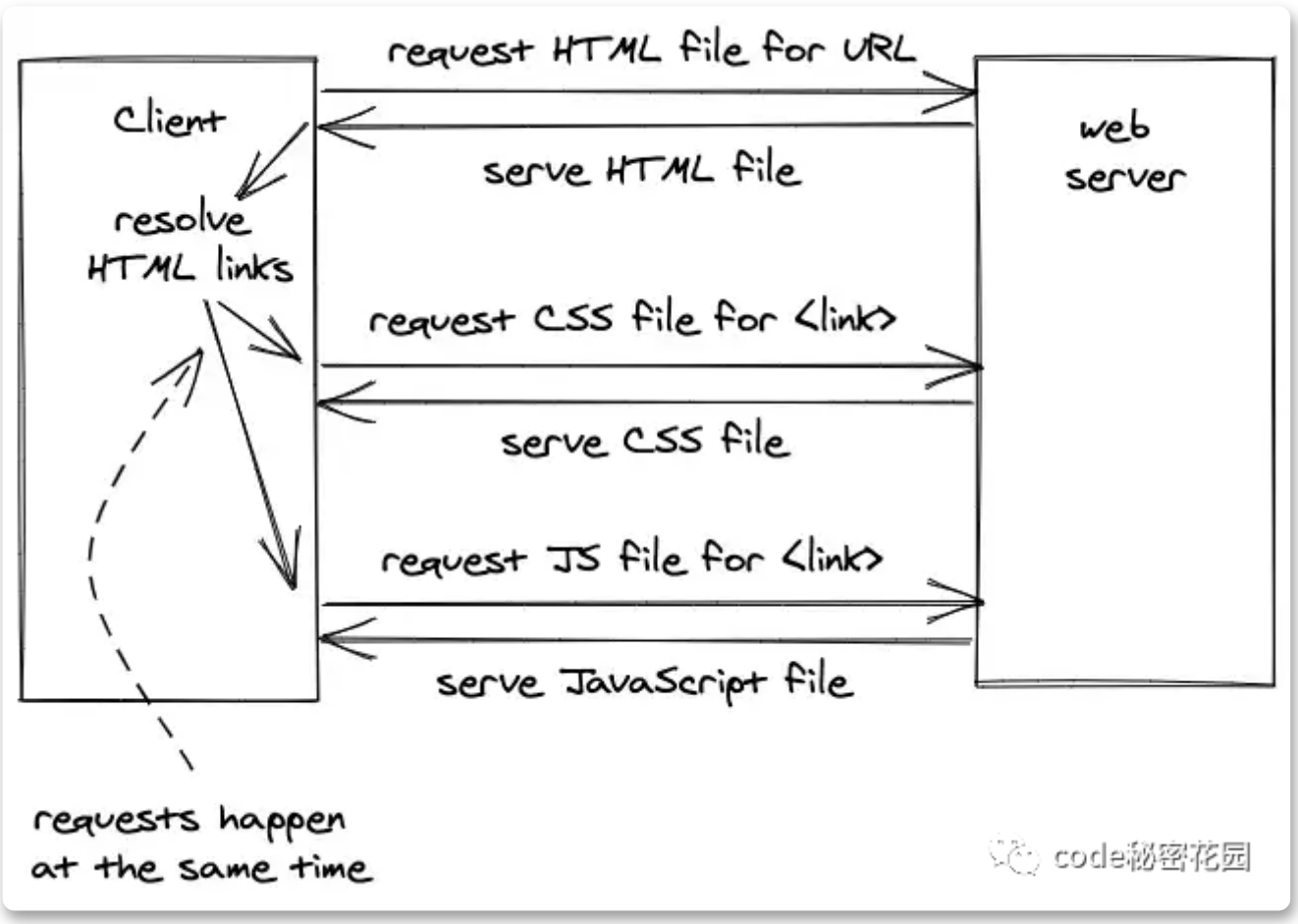
如果浏览器向 Web 服务器请求 URL 的 HTML，Web 服务器会返回 HTML 文件，其中可能包含链接到其他资源（如 CSS 或 JavaScript 文件）的 HTML 标签。对于每个资源，都会向 Web 服务器发出另一个请求。



这些也称为瀑布请求，因为一个请求必须等待另一个请求完成才能继续发送。在我们的示例中，浏览器不知道它需要在 HTML 文件与 HTML link 标签一起到达之前请求 CSS 文件。在下面的示例中，HTML 文件链接了 JavaScript 和 CSS 文件，而 CSS 文件链接了一个 JPG 文件（例如可以用作 CSS background）。



但是，如果一个文件中有多个引用，例如链接了 CSS 和 JavaScript 文件的初始 HTML 文件，这些资源将被并行请求和解析。

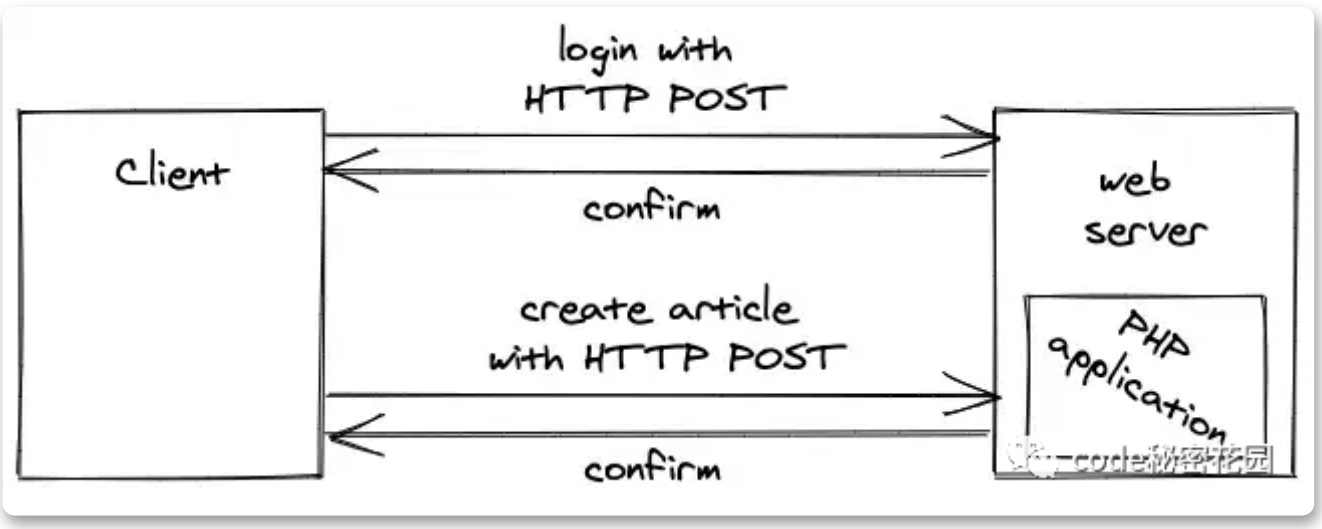


现在，浏览器拥有了特定 URL 下的所有资源(例如 HTML、CSS、JavaScript、png、jpg、svg)，并解析 HTML 及其包含的所有资源，为你渲染所需的结果。它已经准备好让你作为用户与它交互了。

Web 2.0：从网站到 Web 应用

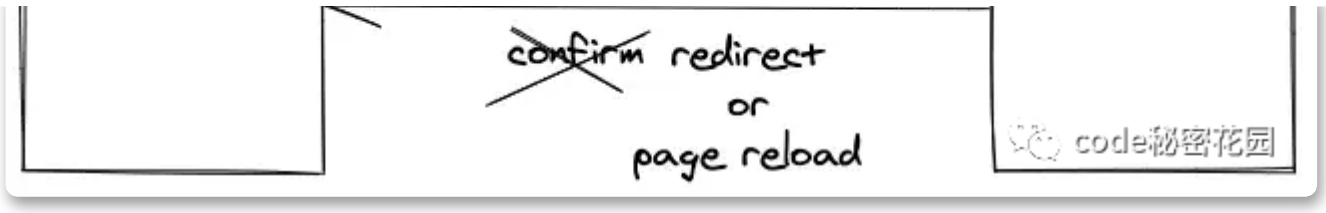
最终，人们不满足于仅仅从 Web 服务器提供静态内容。在 Web 2.0 （大约 2004 年）时代，用户不仅可以阅读内容，还可以创建内容，动态内容慢慢普及了。还记得之前的 HTTP 方法吗？刚才我们只看到了用于读取资源的 HTTP GET 方法，但是其他 HTTP 方法呢？

随着像 Wordpress 这样的内容管理系统的兴起， Web 服务器不仅可以支持用户查看资源，还可以让我们对其进行操作。例如，使用内容管理系统的用户可以进行登录、创建博客文章、更新博客文章、删除博客文章以及注销等操作。此时，编程语言 PHP 最适合这类动态网站开发。

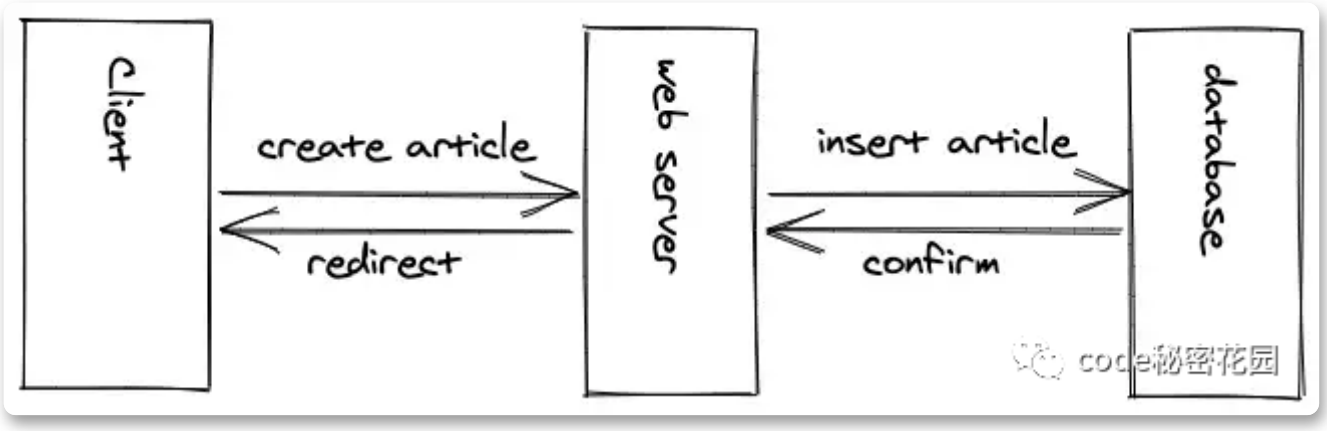


有了服务器端的逻辑，开发人员就可以处理来自用户的读写请求。如果用户想要创建博客文章（写入操作），用户必须在浏览器中编写博客文章并单击“保存”按钮将内容发送到运行在 Web 服务器上的服务端逻辑。这个逻辑会验证用户是否获得了授权，验证博客内容等，并将内容写入数据库。所有这些权限都不允许在客户端上进行，否则每个人都可以在未经授权的情况下操作数据库。

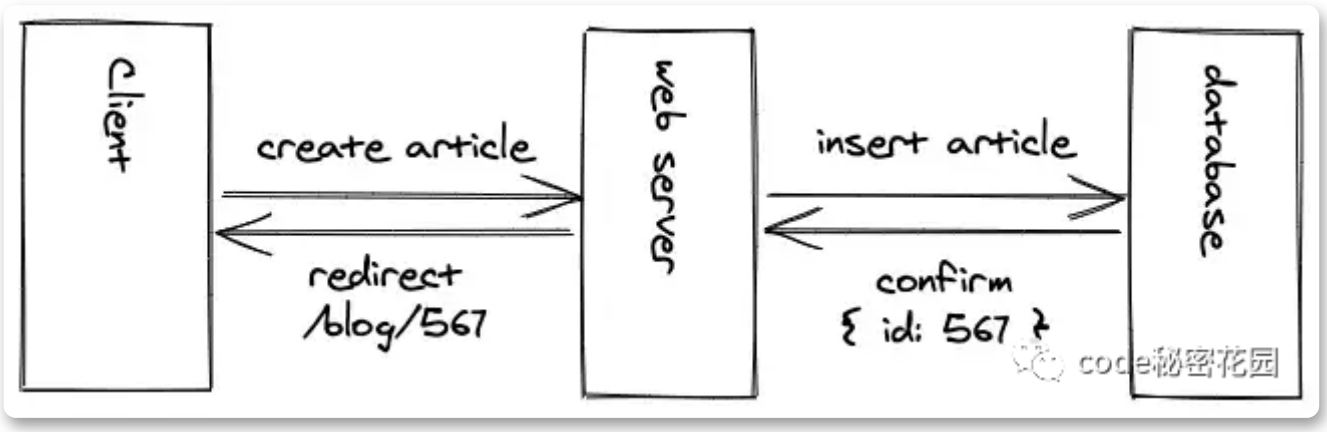
由于我们仍然有服务器端路由的能力，因此在成功创建博客文章后， Web 服务器能够将用户重定向到新页面。例如，重定向可以指向新发布的博客文章。如果没有重定向， HTTP POST/PUT/DELETE 请求通常会导致页面刷新/重新加载。



由于用户现在可以创建动态内容了，我们需要有一个数据库来存储这些数据。（可能在 **Web 2.0** 的早期阶段）数据库可以与 **Web** 服务器一样位于同一物理服务器（计算机）上，也可以在另一台远程计算机上（可能在 **Web** 开发的现代时代）。



一旦将博客文章插入数据库，就可以为该博客文章生成一个唯一标识符（ **id** ），这个 **id** 可以用于将用户重定向到新发布的博客文章的 **URL** 地址。所有这些仍然是异步发生的。



现在，在创建博客文章后，如果博客文章的数据不是静态的，而是存储在数据库中的，服务器如何发送 **HTML** 文件呢？这就是服务器端渲染（不要误认为是服务端路由）发挥作用的地方。

带有面向消费者的网站（静态内容）的 **Web 1.0** 和带有面向生产者的网站（动态内容）的 **Web 2.0** 从服务器返回 **HTML** 。用户导航到浏览器中的 **URL** 并为其请求 **HTML** 。但是，对于 **Web 2.0** 中的动态内容，发送给客户端的 **HTML** 不再是具有静态内容的静态 **HTML** 文件。相反，它会从服务器的数据库中插入动态内容：

```
<?php if ($expression == true): ?>
  This will show if the expression is true.
<?php else: ?>
  Otherwise this will show.
<?php endif; ?>
```

不同编程语言的模板引擎(例如， **Node.js** 上的 **JavaScript** 使用 **Pug** ， **PHP** 上的 **Twig** ， **Java** 上的 **JSP** ， **Python** 上的 **Django** ）可以让 **HTML** 和数据库里的动态数据直接进行交互。在服务端渲染的帮助下，通过在客户端请求时动态创建 **HTML** ，可以将用户生成的内容从服务器提供给客户端。

我们还在处理一个网站吗？从技术上讲是的，但是通过从带有数据库的 **Web** 服务器（或应用程序服务器）提供动态内容来超越静态内容的网站也可以称为 **Web** 应用程序。两种类型之间的界限没有那么清晰。

Web 2.0 这个术语及其流行度在 **2010** 年前后就逐渐减弱了，因为 **Web 2.0** 的特性变得无处不在，并失去了它们的新鲜感。

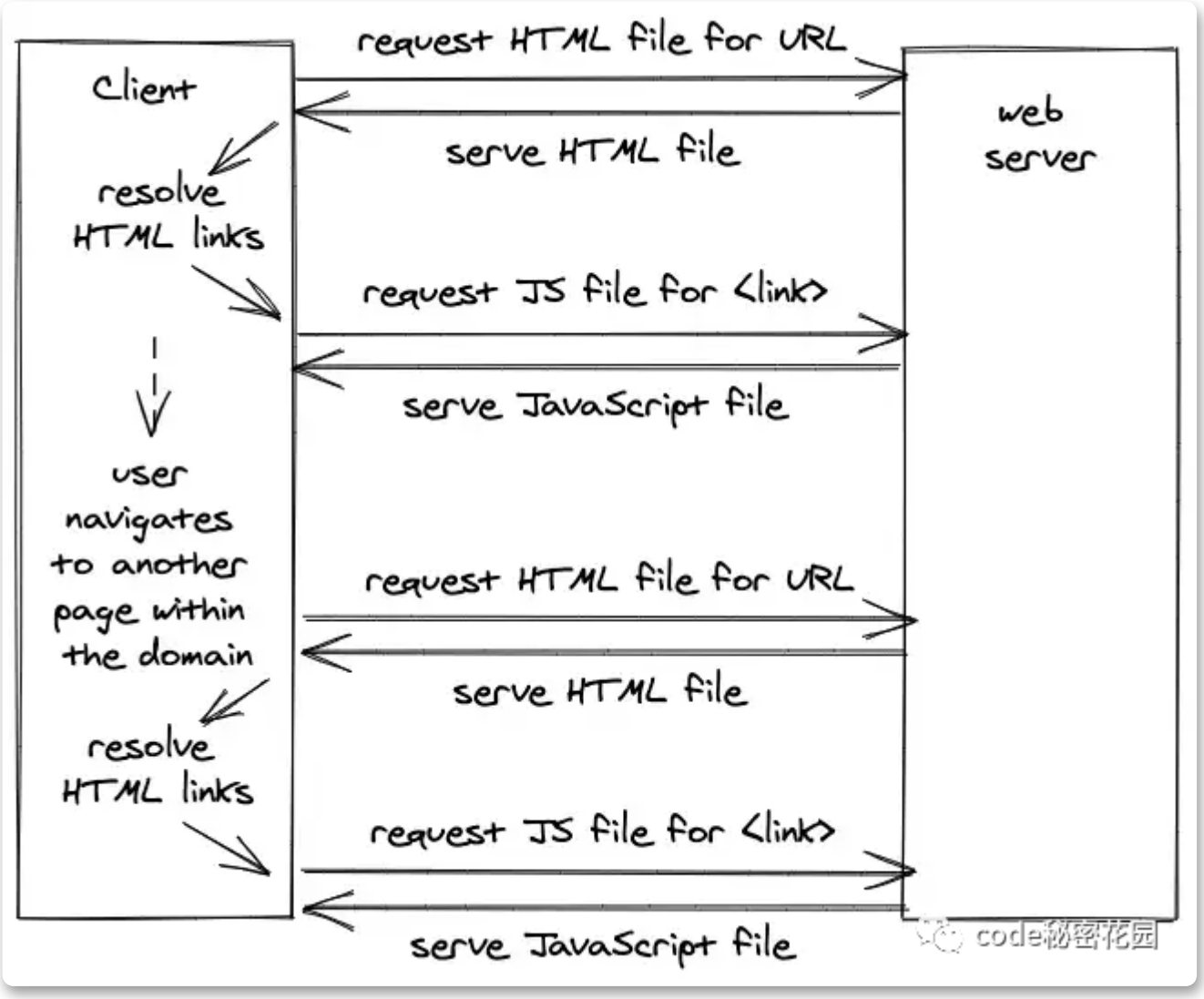
单页应用

2010 年后，单页应用程序（ **SPA**)的兴起使 **JavaScript** 流行起来。在这个时代之前，网站主要是用 **HTML** 加 **CSS** 和少量的 **JavaScript** 开发的。少量的 **JavaScript** 会用于动画或 **DOM** 操作（例如删除、添加、修改 **HTML** 元素），但除此之外没有太多

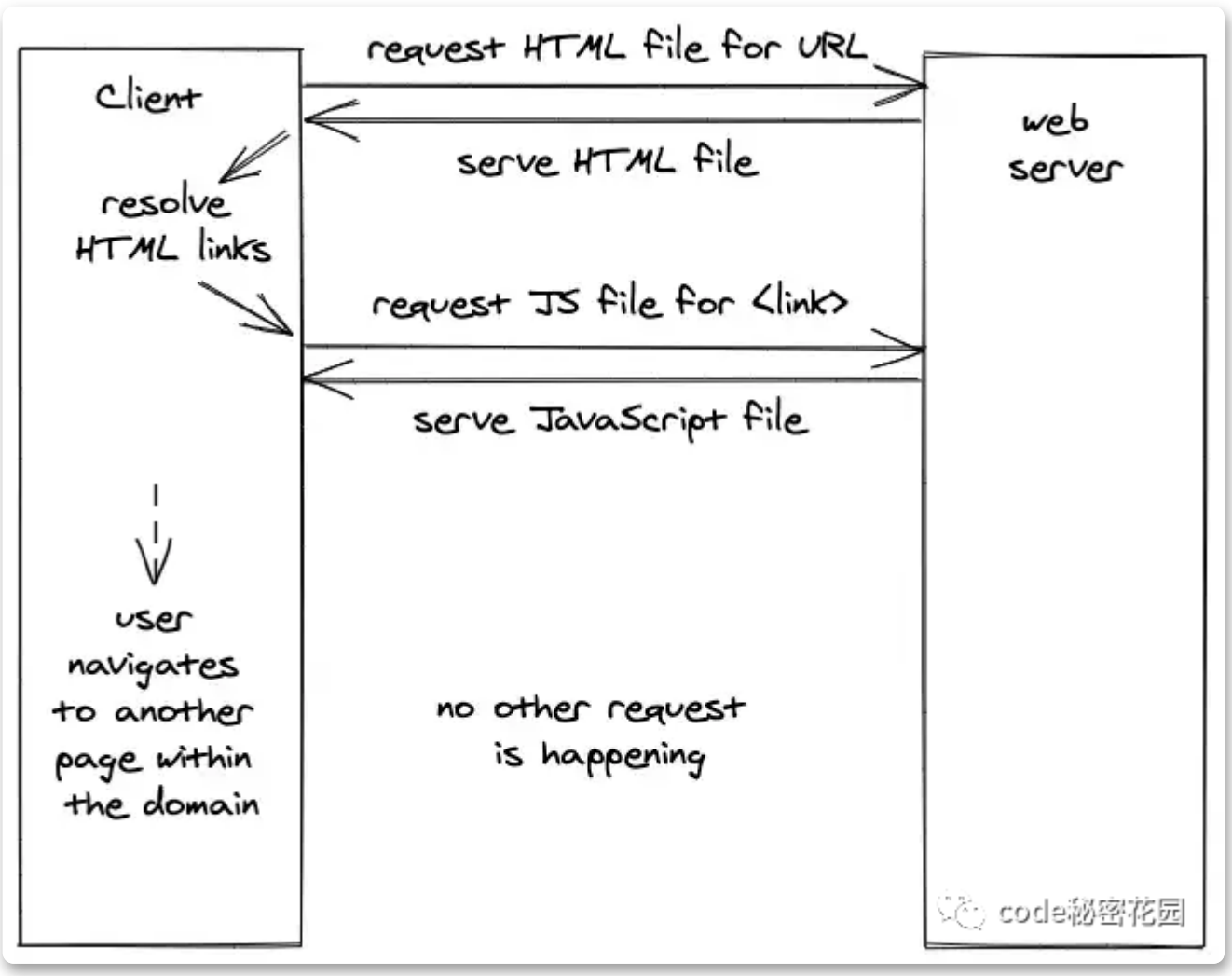
应用。 **jQuery** 是执行此类任务的最流行的库之一。

但是谁能想到整个应用程序都可以用 **JavaScript** 构建呢？ **Knockout.js、Ember.js** 和 **Angular.js** 这些都是早期的用 **JavaScript** 编写单页应用程序的库/框架；而 **React.js** 和 **Vue.js** 是后来才发布的。时至今日，它们中的大多数在现代 **Web** 应用程序中仍然非常活跃。

在单页应用程序出现之前，浏览器会从网站服务器请求 **HTML** 文件和所有链接的资源文件。如果用户碰巧在同一域（例如 **conardli.top**）内从页面（例如 **/home**）导航到页面（例如 **/about**），每次导航都会向 **Web** 服务器发出新请求。



相比之下，单页面应用程序主要用 **JavaScript** 封装整个应用程序，**JavaScript** 包含了如何使用 **HTML**（和 **CSS**）渲染以及渲染什么内容的所有知识。对于单页应用的最基本用法，浏览器只会对一个域请求一次带有一个 **JavaScript** 资源文件的 **HTML** 文件。



单页应用（这里是 **React** 应用）请求的 **HTML** 只是请求 **JavaScript** 应用（这里是 **bundle.js** ）的中间人，在客户端请求并解析之后，它将在 **HTML** 中渲染（ **id="app"** ）：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello code秘密花园 HTML File which executes a React Application</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="./bundle.js"></script>
  </body>
</html>
```

然后， **React** 会接手了这个来自 **./bundle.js** 的 **JavaScript** ：

```
import * as React from 'react';
import ReactDOM from 'react-dom';

const title = 'Hello React';

ReactDOM.render(
  <div>{title}</div>,
  document.getElementById('app')
);
```

在这个小的 **React** 应用程序中，只有一个名为 **title** 的变量显示在 **HTML div** 元素中。但是， **HTML div** 元素之间的所有内容都可以替换为使用 **React** 组件及其模板语法 **JSX** 构建的整个 **HTML** 结构。

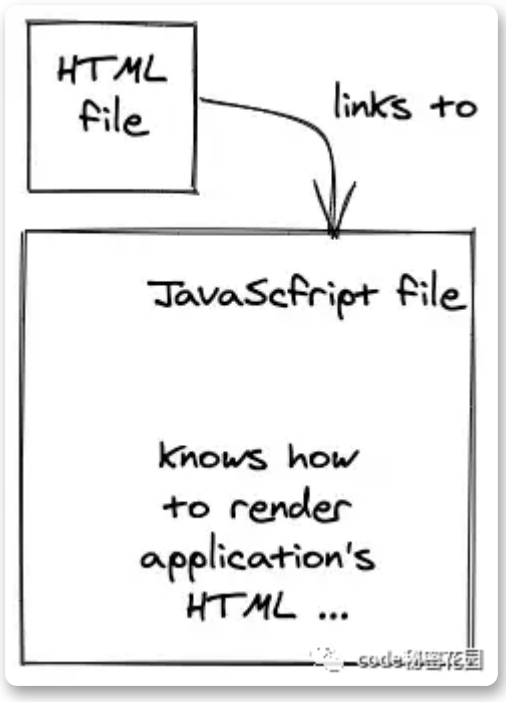
```
import * as React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  const [counter, setCounter] = React.useState(42);

  return (
    <div>
      <button onClick={() => setCounter(counter + 1)}>
        Increase
      </button>
      <button onClick={() => setCounter(counter - 1)}>
        Decrease
      </button>

      {counter}
    </div>
  );
};

ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

这本质上就是早期的模板引擎，但现在是在客户端而不是服务器上执行的，因此这不再是服务端渲染。

```
const App = () => {
```

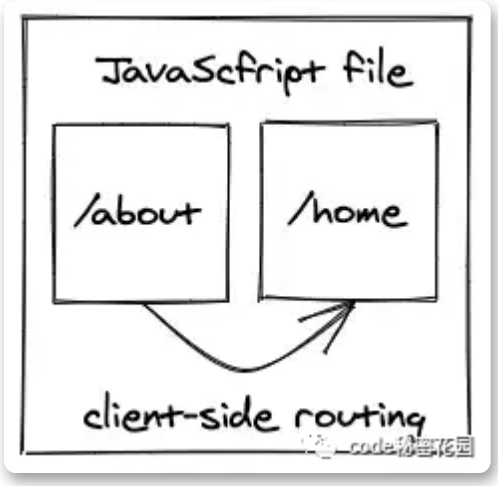
```
const [books, setBooks] = React.useState([
  'The Road to JavaScript',
  'The Road to React',
]);

const [text, setText] = React.useState('');

const handleAdd = () => {
  setBooks(books.concat(text));
  setText('');
};

return (
  <div>
    <input
      type="text"
      value={text}
      onChange={(event) => setText(event.target.value)}
    />
    <button
      type="button"
      onClick={handleAdd}
    >
      Add
    </button>

    <List list={books} />
  </div>
);
};

const List = ({ list }) => (
  <ul>
    {list.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
```

由于这种从服务器到客户端执行渲染的变化，我们现在称之为客户端渲染。换句话说：我们不是直接从 **Web** 服务器提供预渲染的 **HTML**，而是主要从 **Web** 服务器提供 **JavaScript**，它在客户端上执行，然后才渲染 **HTML**。通常，**SPA** 和 **客户端渲染** 的应用程序表达的是相同的意思。



如果 **SPA** 仅从 **Web** 服务器发送一个请求，当用户从一个页面导航到同一域中的另一个页面（例如 conardli.top/about 到 conardli.top/home）而不请求另一个 **HTML** 时，它是怎么工作的呢？

随着传统 SPA 的发展，我们也从服务端路由转移到了客户端路由。最初为基本 SPA 请求的 JavaScript 文件封装了网站的所有页面。从一个页面（例如 /about ）导航到另一个页面（例如 /home ）不会对 Web 服务器执行任何请求。相反，客户端路由（例如 React 的 React Router ）会负责从最初请求的 JavaScript 文件渲染适当的页面。

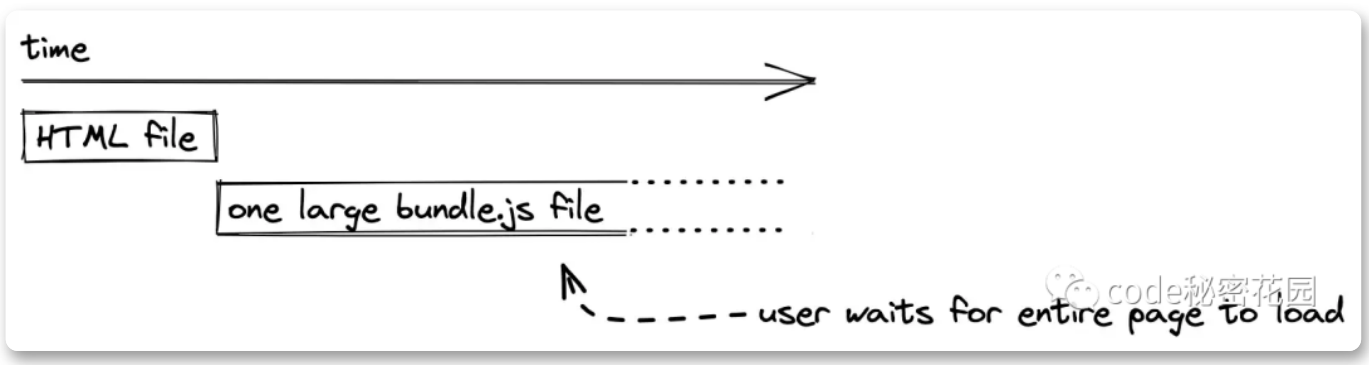


简而言之：一个基本的单页应用程序使用客户端渲染/路由而不是服务端渲染/路由，同时仅从 Web 服务器请求整个应用程序一次。它是一个页面，因为整个应用程序只有一个请求，它是一个链接到一个 JavaScript 文件的 HTML 页面；它封装了所有实际的 UI 页面并在客户端执行。

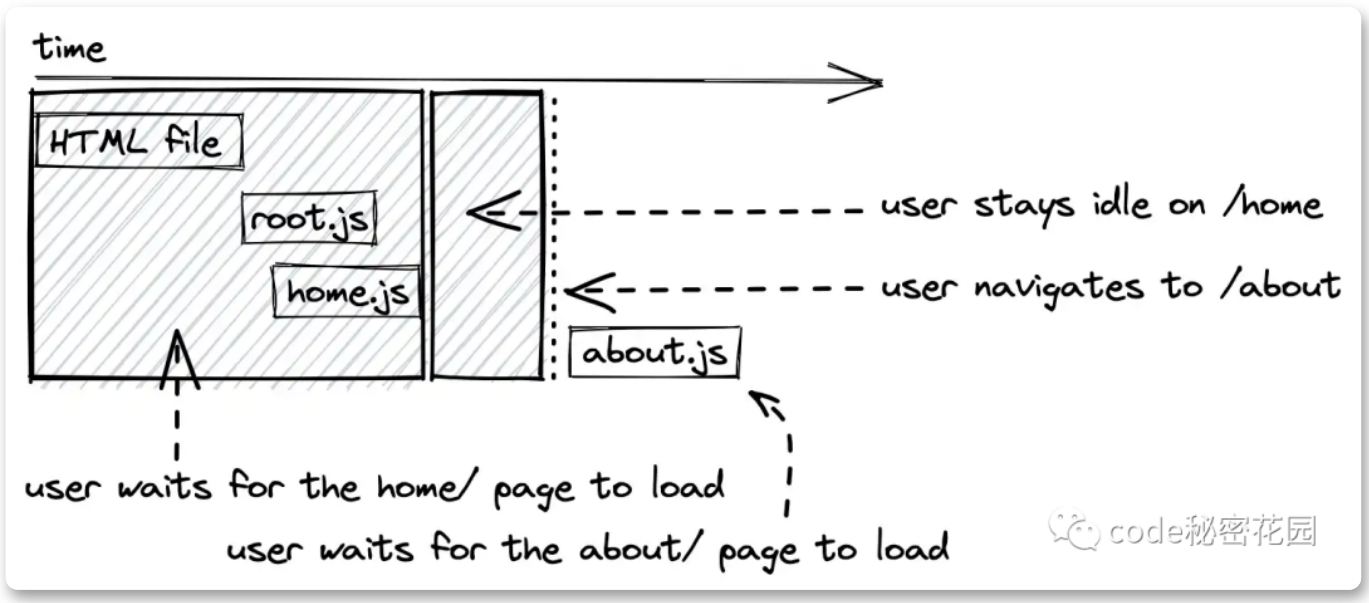
可以说，在我们拥有单页应用之前，我们一直在使用多页应用，因为对于每个页面（例如 /about ），都会向 Web 服务器发出一个新请求，以请求它所需的所有文件。然而，多页面应用并不是一个真正的术语，因为它是单页应用流行之前的默认设置。

代码拆分

我们了解到，SPA 默认以一个小的 HTML 文件和一个 JS 文件的形式提供。JavaScript 文件开始时很小，但随着你的应用程序变大，它的体积会逐渐增加，因为更多的 JavaScript 会打包在同一个 bundle.js 文件中。这会影响 SPA 的用户体验，因为将 JavaScript 文件从 Web 服务器传输到浏览器的初始加载时间会增加。加载完所有文件后，用户可以从一个页面导航到另一个页面而不会中断。但是，相比之下，首屏渲染和加载的时间会降低用户体验。



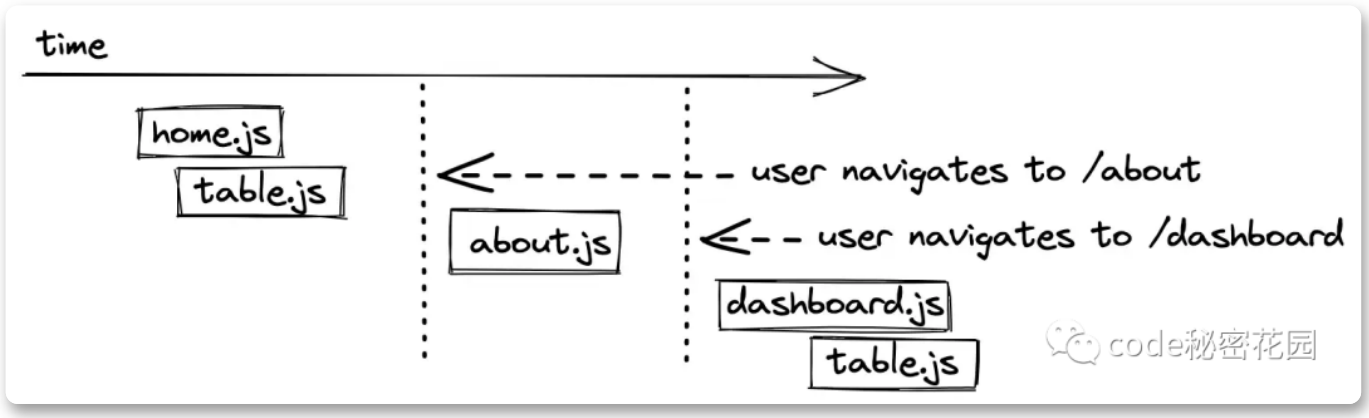
一旦应用程序变大，将整个应用程序打包到一个 JavaScript 文件就会成为一个缺点。对于更复杂的单页应用程序，诸如代码拆分（在 React + React Router 中也称为延迟加载）之类的技术仅用于为当前页面所需的应用程序的一小部分（例如 conardli.top/home）提供服务。当导航到下一页（例如 conardli.top/about）时，会向 Web 服务器发出另一个请求以请求该页面的部分



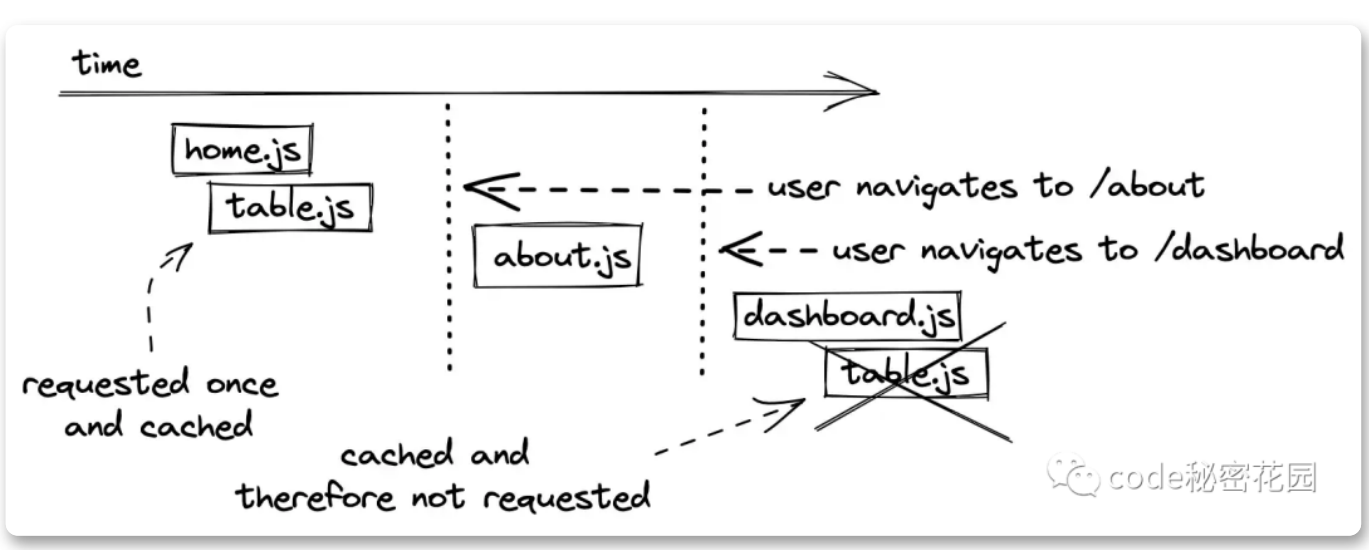
如果你回顾一下传统网站的工作方式，你会发现它与启用代码拆分的 SPA 非常相似。对于传统网站，每次用户导航到新路由时，都会加载一个新的 HTML 文件（带有可选的 CSS、JavaScript 和其他资源文件）。对于在路由级别进行代码拆分的 SPA ，每次导航都会请求新的 JavaScript 文件。

我们仍然可以调用这个单页应用还是回到多页应用程序？你会看到这些术语之间的界限会慢慢变得不太清晰了...

代码拆分不需要像之前的场景那样在路由级别发生。例如，也可以将较大的 `React` 组件提取到其独立的 `JavaScript` 包中，以便它只会在实际使用它的页面上加载。



但是，正如你所见，这会导致从 `Web` 服务器请求冗余的代码。当用户两次导航到代码拆分后的路由时也会发生同样的情况，因为它也会从 `Web` 服务器加载两次。因此，我们希望读取浏览器缓存结果。



现在，如果我们在表格中引入了新功能，打包后的 `table.js` 文件发生了变化，会发生什么呢？如果启用缓存，我们仍然会在浏览器中看到旧版本的 `Table` 组件。

作为此问题的解决方案，应用的每个新版本都会检查打包后的代码是否已更改。如果它发生了变化，它会收到一个基于时间戳的带有哈希的新文件名（例如 `table.hash123.js` 变为 `table.hash765.js` ）。当浏览器请求具有缓存文件名的文件时，它会使用缓存版本。但是，如果文件已更改并且也更新了 `hash` 值，浏览器就会请求新的文件。

另一个例子是第三方 `JavaScript` 库的代码拆分。例如，在为 `React` 安装带有 `Button` 和 `Dropdown` 等组件的 UI 库时，也可以进行代码拆分。每个组件都是一个独立的 `JavaScript` 文件。从 `UI` 库导入 `Button` 组件时，仅导入 `Button` 中的 `JavaScript` ，而不导入 `Dropdown` 中的 `JavaScript` 。

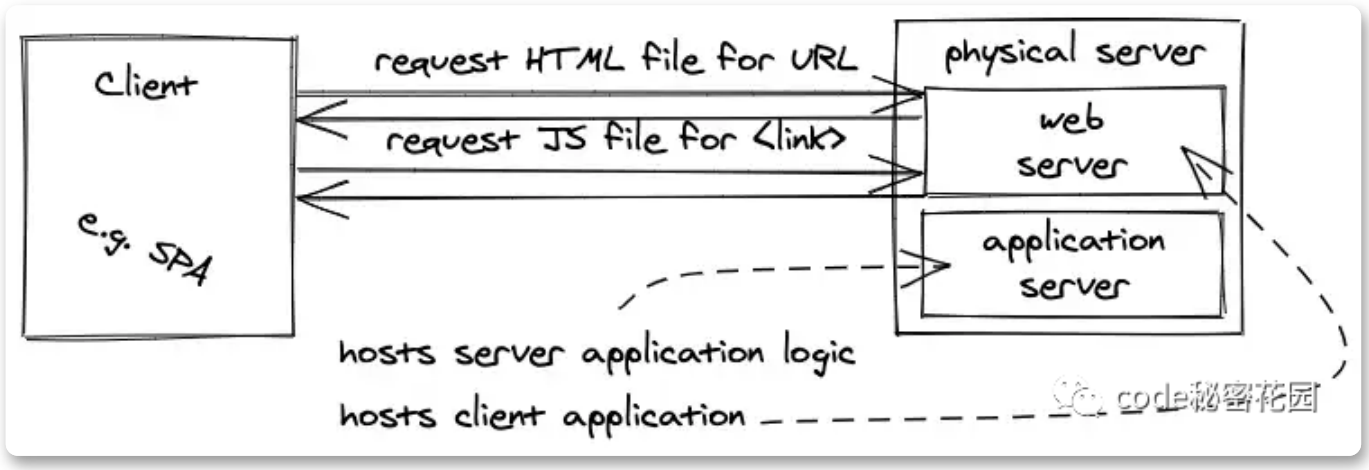
为了将 `React` 应用（或库）打包到一个或多个（带有代码拆分的） `JavaScript` 文件中，另一种称为 `tree shaking` 的技术开始发挥作用，它会帮助我们消除掉未使用过的代码，避免这些代码被打包。从历史上看， `JavaScript` 中使用了以下打包程序（从过去到最近）：

- Grunt (2012)
- Gulp (2013)
- Webpack (2014+)
- Rollup (mainly libraries)
- esbuild (2020+)

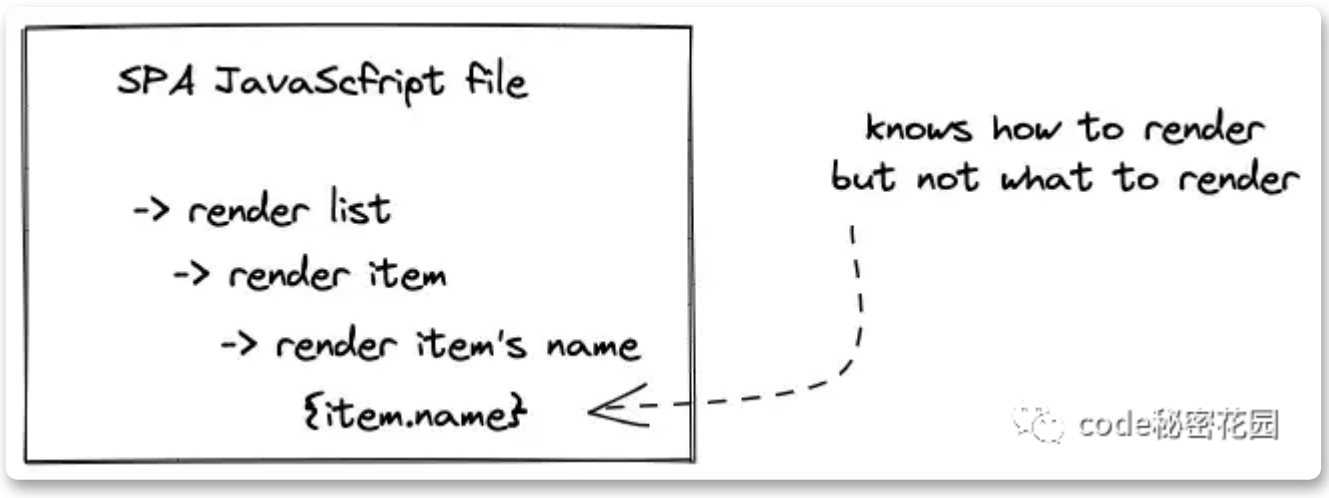
全栈应用

我们正在进入与 `SPA` 同时流行的全栈应用程序时代。全栈应用程序包括客户端（例如 `SPA` ）和服务器应用程序。如果公司正在寻找全栈开发者，他们通常希望这些人有能力同时写客户端和服务端。有时客户端和服务器可以使用相同的编程语言（例如，客户端上的 `JavaScript` 和 `React` ，服务器上的 `JavaScript` 和 `Node.js` ），但也没必要。

其实也是由于客户端单页应用的兴起，对全栈应用程序的需求应运而生。

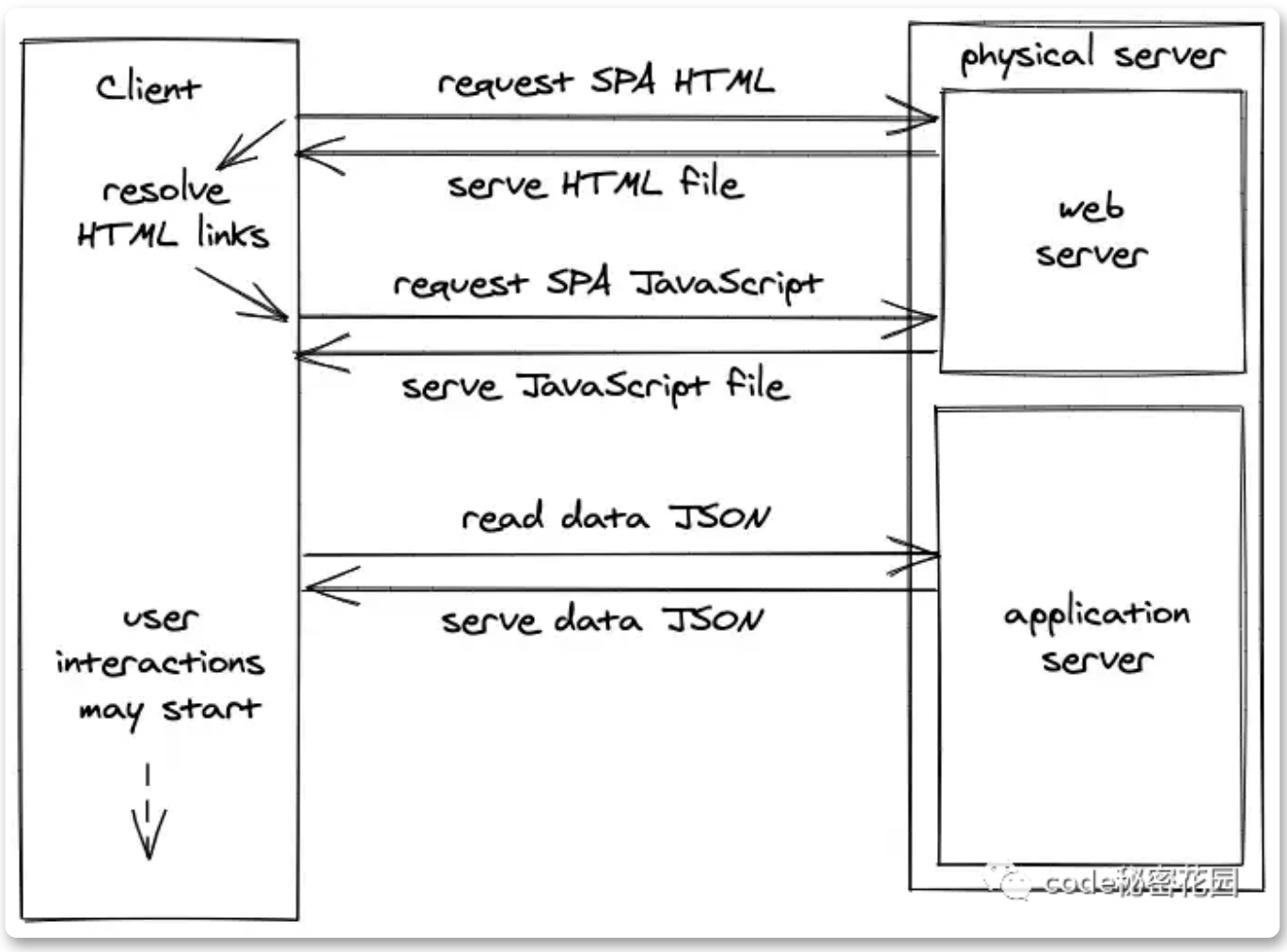


到目前为止，我们已经从使用 **HTML/CSS/JavaScript** 的传统网站发展到现代 **Web** 应用程序（例如 **React** 应用）。渲染静态内容很好，但我们如何渲染动态内容，如博客文章，如果只提供 **JavaScript**（和 **HTML**）如何将完全由客户端渲染接管的 **SPA** 时和 **Web** 服务器进行交互呢？

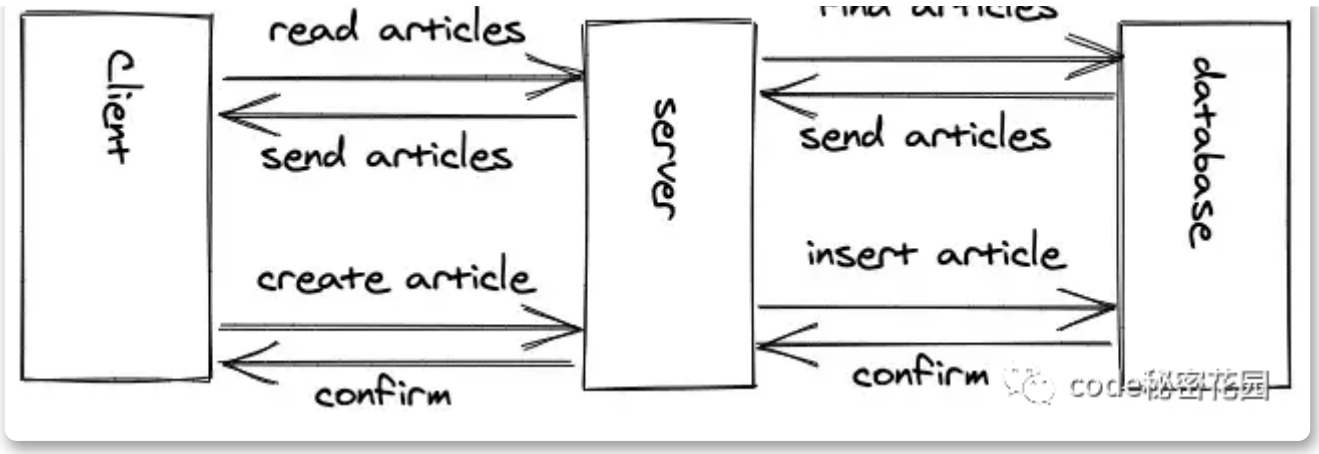


SPA 应用 — 封装在一个 **JavaScript** 文件中，没有任何用户特定的数据。这只是页面的逻辑：UI 以及它们在用户交互中的行为方式。实际数据并没有被加入进去，因为它们还在数据库里待着呢。这是从服务端渲染转移到客户端渲染时必须做出的权衡。

因此，我们必须从客户端向服务器（使用 **JavaScript/Node.js** 或其他编程语言编写的应用程序服务器）发出另一个请求，以请求这些缺失的的数据。客户端模板引擎（例如 **React** 中的 **JSX**）负责渲染内容（数据）。



在处理客户端渲染的应用程序时，基本上有两次请求往返：一次是用于 **JavaScript** 应用程序，另一次用于请求一些动态数据。在浏览器中渲染完所有内容后，用户就开始与应用程序交互 — 例如创建新的博客文章。**JSON** 是从客户端向服务器发送数据的首选格式。服务器通过读取或写入数据库来处理来自客户端的所有请求。



客户端渲染应用（**SPA**）需要注意的是，并不是从一开始就可以使用所有数据的。他们必须要等待一些异步的动态数据请求。作为浏览网页的最终用户，你会以两种方式注意到客户端渲染的应用程序：

- 首先，会加载一个大页面的 **Lodaing**，然后转换为很多小部件的加载 **Lodaing**（瀑布请求），因为请求数据是在渲染初始页面之后发生的。
- 然后，从路由到路由的导航是实时的（不包括代码拆分，因为由于对服务器的额外打包请求，它感觉有点慢）。这就是我们从 **SPA** 中获得的好处。

除了额外的数据获取请求之外，客户端渲染的应用程序还必须处理状态管理的问题，因为用户交互和数据需要在客户端的某个地方存储和管理。

使用 **SPA** 时考虑：用户以作者身份访问可以发布博客文章的网站。在当前页面，用户可以看到他们所有的博客文章，因此在加载此页面时需要获取所有这些博客文章。这些获取的文章在代码中会被保存为客户端内存中的状态。现在，当用户开始与页面及其数据进行交互时，每个文章的按钮允许用户单独删除它们。当用户单击删除按钮时会发生什么？

用户单击删除按钮，该按钮会向应用程序服务器发送一个请求，其中包含博客文章的标识符和删除它的指令（通常一个 **HTTP DELETE** 就足够了）。在服务器上的所有权限检查（例如用户是否授权、博客文章是否存在、博客文章是否属于用户）完成后，服务器会将操作委托给删除博客文章的数据库。数据库向服务器确认操作成功，服务器向客户端发送响应。现在，客户端要么从内存中的本地状态中删除博客文章，要么再次从服务器获取所有博客文章，并用更新的博客文章列表替换内存中的博客文章。

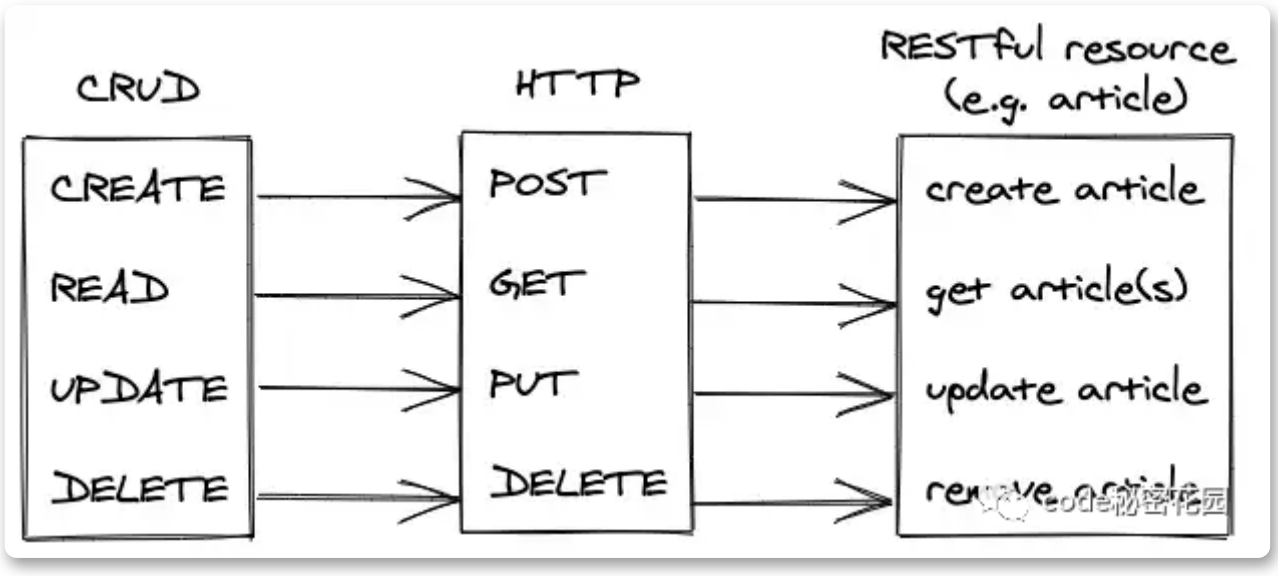
在执行客户端路由时，可以通过状态管理最小化对数据（例如文章）的请求。这意味着理想情况下，从一个页面导航到另一个页面然后返回初始页面的用户不应该触发对初始页面所需数据的第二次请求。相反，它应该已经通过状态管理缓存在客户端上了。

最后但同样重要的是，客户端和服务器之间的接口称为 **API**。在这种情况下，它是客户端和服务器之间的一种特定类型的 **API**，但是在编程中很多东西都称为 **API**。

客户端 - 服务器通信

传统的全栈应用程序使用 **REST** 作为其 **API** 规范；它采用 **HTTP** 方法进行 **CRUD** 操作。之前，我们已经在文件和用户交互之间使用 **HTTP** 方法进行 **CRUD** 操作了，但是没有遵循明确的约束 — 比如使用 **PHP** 等服务端语言创建文章。

但是，当使用 **REST API** 时，我们在 **RESTful** 资源上使用这些 **HTTP** 方法。例如，一个 **RESTful** 资源可以是一篇博客文章。用户可以使用 **HTTP GET** 从应用程序服务器读取博客文章，或者使用 **HTTP POST** 在应用程序服务器上创建新的博客文章。



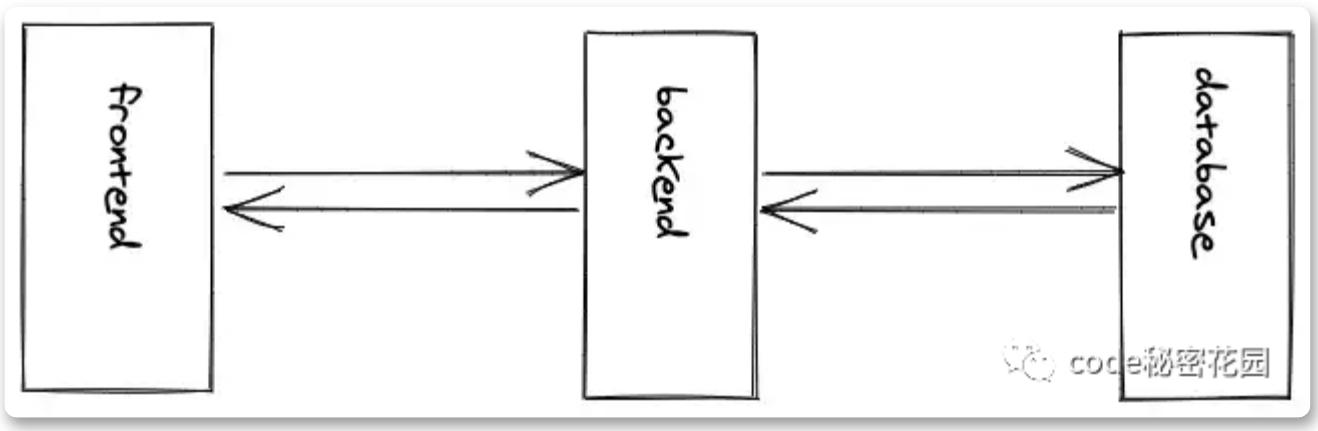
REST API 负责连接客户端和服务器应用程序，而无需使用相同的编程语言去实现。他们只需要提供一个用于发送和接收 **HTTP** 请求和响应的库。**REST** 是一种没有数据格式（过去是 **XML**，但现在是 **JSON**）和编程语言的通信范式。

REST 的现代替代方案是用于客户端和服务器之间 API 的 GraphQL 。GraphQL 也不需要绑定到数据格式，与未绑定到 HTTP 的 REST 相比，大多数情况下你也会看到这里使用的 HTTP 和 JSON 。

到目前为止讨论的技术，全栈应用程序将客户端和服务器应用程序分离。两者都通过精心挑选的 API （例如 REST 或 GraphQL ）进行通信。当客户端应用程序在浏览器中渲染 Web 应用程序所需的一切时，服务器应用程序处理来自客户端的读取和写入数据的请求。

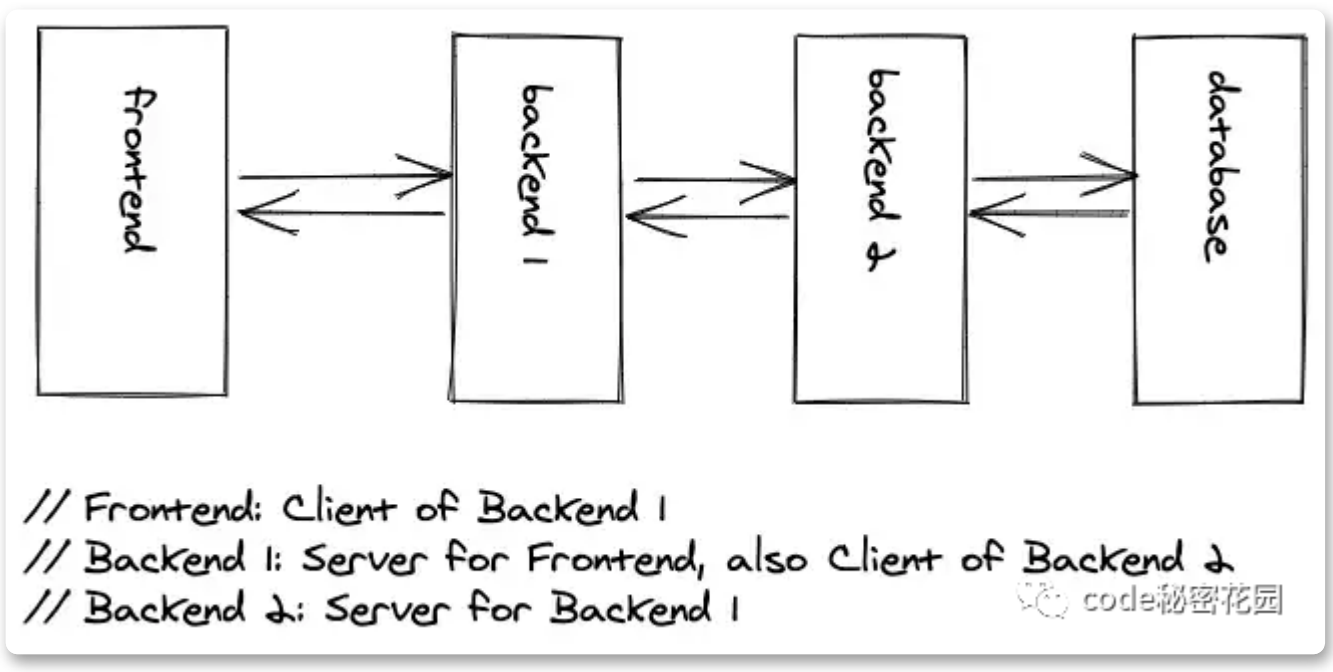
前端和后端

我们还没有讨论前端和后端这两个术语，因为我不想预先添加太多信息。前端应用程序可能是用户在浏览器中看到的所有内容（例如网站、Web 应用程序、SPA ）。因此，你会看到前端开发人员最常使用 HTML/CSS 或 React.js 之类的库。相比之下，后端通常是背后的逻辑：它是读取和写入数据库的逻辑，与其他应用程序交互的逻辑，通常是提供 API 的逻辑。



但是，不要将客户端应用程序始终误认为是前端，而将服务器应用程序始终误认为是后端。这些概念不能那么容易地交换。前端通常是在浏览器中看到的东西，而后端通常执行不应在浏览器中公开的业务逻辑，并且通常也连接到数据库。

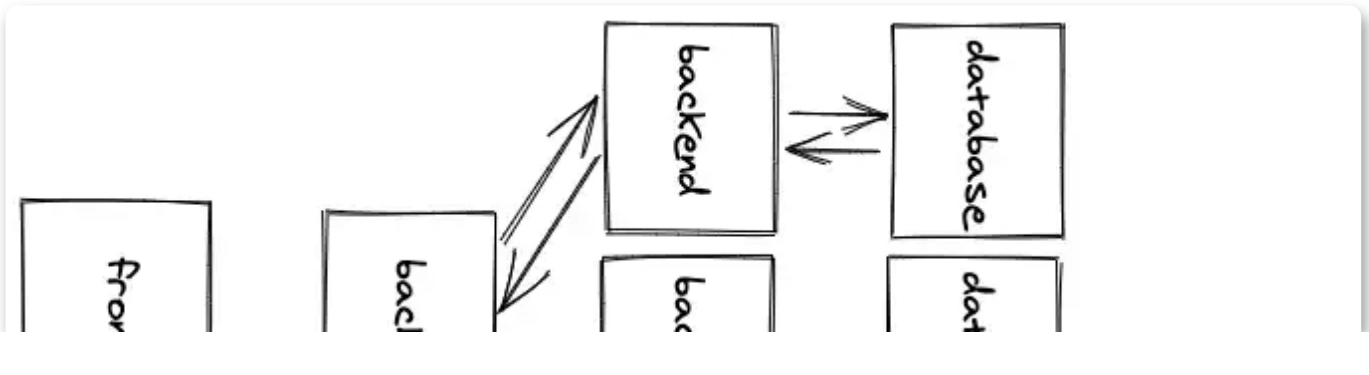
但是，相比之下，客户端和服务器是一个角度问题。使用另一个后端（Backend 2）的后端应用程序（Backend 1）成为服务器应用程序（Backend 2）的客户端应用程序（Backend 1）。但是，同一个后端应用程序（Backend 1）仍然是另一个客户端应用程序的服务器，即前端应用程序（Frontend）。

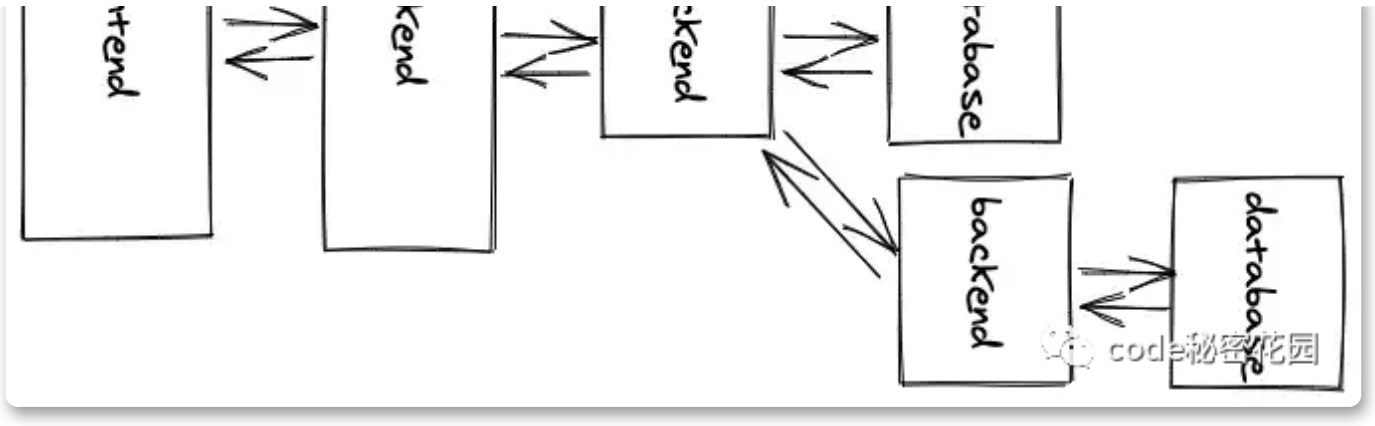


如果你想回答客户端-服务器问题，如果有人问你实体在客户端-服务器架构中扮演什么角色，请始终问自己谁（服务器）为谁（客户端）服务，谁（客户端）使用谁的（后端）功能？

微服务

例如，微服务是一种将一个大后端（也称为单体）拆分为较小后端（微服务）的架构。每个较小的后端可能具有一个特定于域的功能，但它们毕竟都服务于一个前端（或多个前端）。但是，一个后端也可以消费另一个后端，而前者的后端成为客户端，而后者的后端成为服务器。



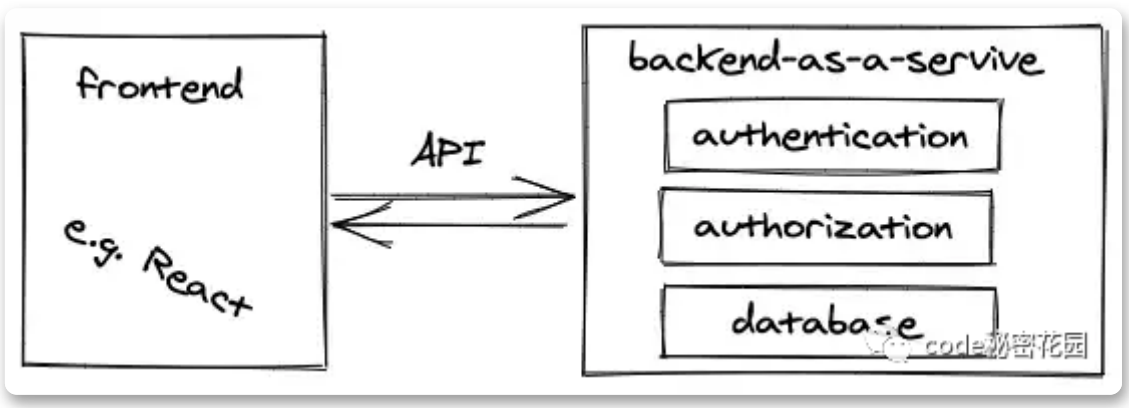


在微服务架构中，每个后端应用程序都可以使用不同的编程语言创建，而所有后端都可以通过 **API** 相互通信。他们选择哪种 **API** 规范无关紧要，无论是 **REST API** 还是 **GraphQL API**，只要与其服务器交互的客户端了解 **API** 规范即可。也可能出现前端不只与一个后端交互，而是与多个后端并行交互的情况。

后端即服务

在传统意义上，一个只为一个前端应用程序服务的后端应用程序通常连接到一个数据库。这是一个典型的全栈应用程序。但是，大多数情况下，后端应用程序除了读取和写入数据库、允许某些用户执行某些操作（授权）或首先验证（例如登录、注销、注册）用户之外，并没有做太多事情地方。如果是这种情况，通常不需要自己实现后端应用程序。

Firebase（由 **Google** 提供）是一种后端即服务解决方案，它提供数据库、身份验证和授权作为开箱即用的后端。开发人员只剩下实现需要连接到此后端即服务的前端应用程序（例如 **React** 应用程序）。



Firebase 等后端即服务（**BaaS**）允许开发人员快速启动并运行他们的前端应用程序。身份验证、授权和数据库的一切都为你完成。此外，大多数 **BaaS** 也提供托管服务，例如，你的 **React** 应用程序也可以使用 **Firebase** 托管。**Firebase** 会将你的 **React** 应用程序提供给你的客户端（浏览器），并让你的应用程序可以使用所有其他的功能（例如身份验证、数据库）。**Firebase** 的一个流行的开源替代品是 **Supabase**。

超越全栈应用

如果前面这些你还跟得上，我们来看看全栈应用程序的最新发展。随着从传统网站到全栈应用程序的所有发展，你可能已经注意到从 **X** 到 **Y** 的转变常常使事情变得更加复杂.....

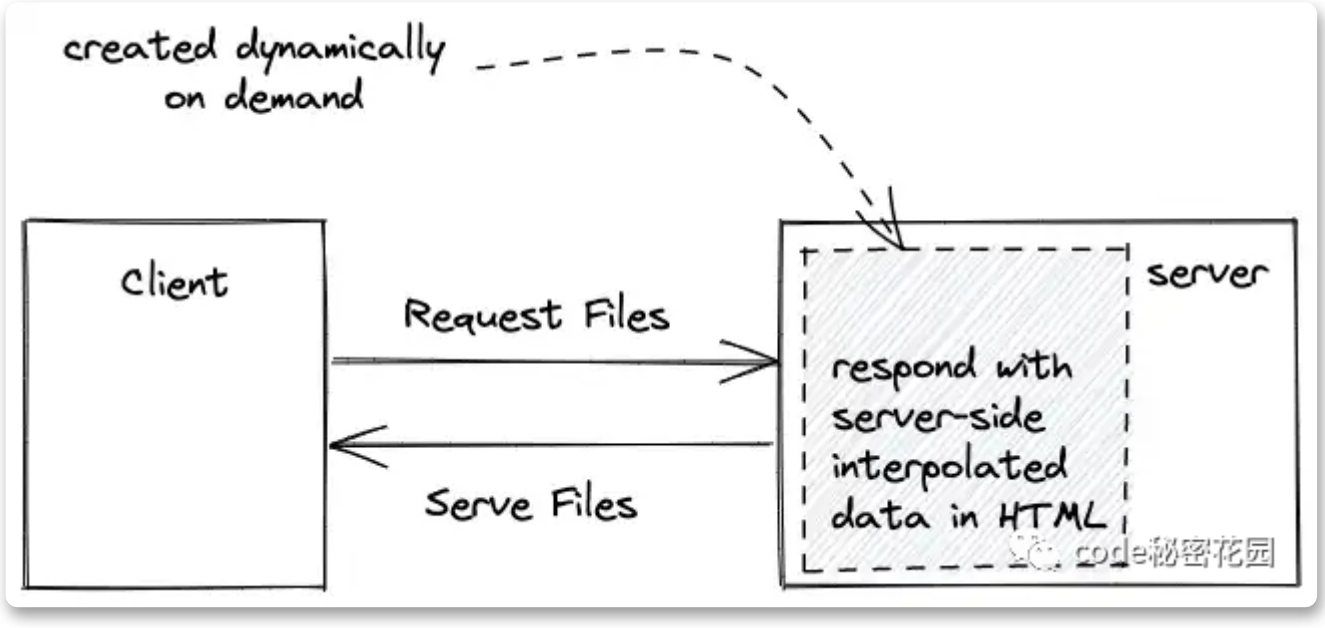
- 服务端路由 (X) 到客户端路由 (Y)
  - 带来了包体积问题，可以通过代码拆分来解决
- 服务端渲染 (X) 到客户端渲染 (Y)
  - 为开发者提供额外的数据获取和状态管理工作
  - 为最终用户提供大量加载 Loading
  - 增加了额外的数据请求

在下文中，我想向你介绍两种方法，它们的理念（**SSR**、**SSG**）并不新鲜，一些现代库（例如 **React**）和框架（例如 **Next.js**、**Gatsby.js**）使这些方法成为可能。

服务端渲染 2.0 (SSR)

我们之前已经了解了 **Web 2.0** 的服务端渲染。在后来的某个时间点，全栈应用将客户端和服务器解耦，并使用 **React** 等库引入了客户端渲染。那么，如果再退一步，使用 **React** 进行服务器端渲染呢？

当使用基于 React 之上的流行 Next.js 框架时，你仍在开发 React 应用程序。但是，你在 Next.js 中实现的所有内容都将在服务器端渲染。在 Next.js 中，你使用 React 实现每个页面（例如 /about、/home ）。当用户从一个页面导航到另一个页面时，只有一小部分服务器端渲染的 React 被发送到浏览器。它的强大之处在于：你可以请求一些动态的数据，使用 React 插入这些数据，并将其发送到客户端而不会有任何间隔。



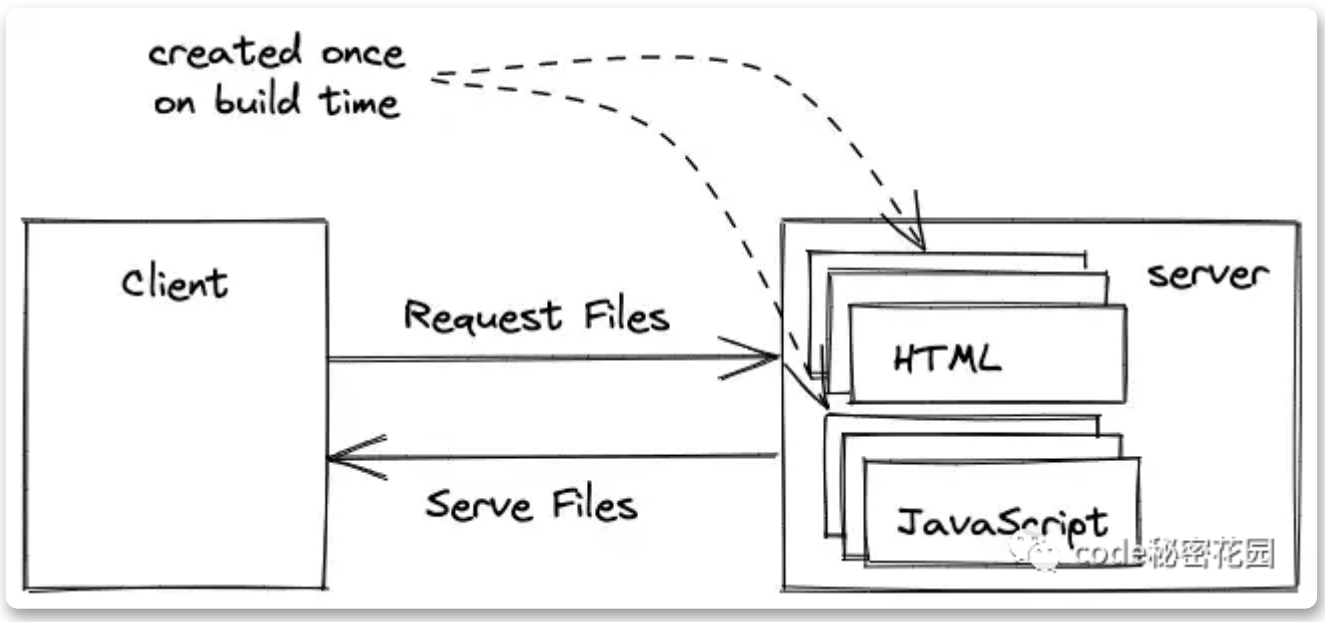
这与客户端渲染不同，因为 React 只在客户端管理，并且只有在客户端上没有数据的情况下或者最初渲染时才开始请求数据。使用 SSR React，你可以在服务器上插入 React 中的数据，也可以选择在应用程序渲染时在客户端获取数据。客户端渲染和服务器端渲染这两个选项可以混合使用。

- 优势：客户端收到的 HTML 已经渲染好了数据（ UX 和 SEO 的改进）
- 缺点：客户端可能需要等待更长时间，因为渲染好的 HTML 是在服务器上动态创建的（利用好 HTTP 缓存可以进行一些优化）。

静态站点生成 (SSG)

传统网站使用来自 Web 服务器的静态文件在浏览器上渲染。就像我们所了解的一样，没有应用程序服务器的参与，也没有服务端渲染的参与。传统网站的方法非常简单，因为 Web 服务器只托管你的文件，并且在用户访问你的浏览器的每个 URL 上都会发出请求以获取必要的文件。那么如果我们可以将 React 用于静态文件呢？

React 本身不适用于静态文件。相反，React 只是在客户端动态创建应用程序的 JavaScript 文件。但是，基于 React 之上的框架 Gatsby.js 可以用于为 React 应用程序生成静态站点。Gatsby 采用 React 应用程序并将其编译为静态 HTML 和 JavaScript 文件。然后所有这些文件都可以托管在 Web 服务器上。如果用户访问 URL，则将静态文件提供给浏览器。



与服务端渲染 React 相比，静态文件不会在用户请求时动态创建，而只会在构建时创建一次。对于数据经常变化的动态内容，这可能是一个缺点，但是，对于内容不经常变化的活动页或博客，只构建一次网站是完美的解决方案。

— end —