第一篇 基础篇

第1章

现代Web应用开发：全新的纪元

自 20 世纪 90 年代中期万维网流行以来，万维网发生了巨大的变化。浏览器一开始没有显示图像的功能，后来服务器端动态 Web 页面逐步崭露头角，JavaScript 被引入，带来了客户端强大的交互灵活性，曾经的移动端页面让位于响应式网页。

互联网技术的蓬勃发展，让越来越多的人对开发 Web 应用程序产生了兴趣，并吸引更多的访问者访问 Web 资源。相信任何有抱负的开发人员在尝试开发 Web 应用程序之前都可能遇到的挑战之一就是选择 Web 应用程序体系结构和组件模型。本章作为全书的开篇，将逐步分享一些实践经验，以期带给大家一些启示。

面对日益繁多的互联网 Web 应用、上亿级的高频在线交易、无处不在的即时社交应用程序，尽管网络之间的通信模式依然离不开经典的 TCP/IP 四层模型，但因使用场景的不同，如 Web 应用之间、Web 应用与客户端之间、Web 应用与数据库之间，其交互的细节也不断改进，并衍生出多种不同的解决方案（正像如今应用广泛的 HTTP 2）。

并发一直都是 Web 领域关注的话题，但再也不仅仅只是简单的多进程 / 线程开发。计算机科学家们在过去几十年所发表的相关学术论文，实践到各个新语言及技术中焕发了新生。比较主流的并发模型如 Actor、CSP（Communicating Sequential Processes，通信顺序进程模型）、STM（Software Transactional Memory，软件事务存储模型）以及数据流并发模型（Dataflow Concurrency）。本书会在之后的章节中和大家进一步探讨这些内容。

1.1 单页应用概述

长期以来，当单击页面时，浏览器刷新并加载页面内容。随着单页应用（Single Page Application，SPA）的出现，这种情况发生了改变。根据用户交互产生的数据，动态更新页面，而无须刷新页面，这样大大增强了用户的体验性。本章将重点介绍 SPA。

1.1.1 从此不必刷新浏览器

一个网站的传统交互方式通常是由客户端（这里指浏览器）向服务器发出初始请求，服务器

处理请求并返回 HTML，任何后续的请求都会按照相同的过程被处理，即 HTML 页面都会被发送到 Web 服务器，服务器加载页面、处理事件，并将新 HTML 呈现到客户端，所以不得不刷新页面，如图 1.1 所示。

SPA 同样会发起初始请求，然而不同的是，每个后续请求都会通过 AJAX 向服务器端发送及接收 JSON 格式的数据，最后数据平滑同步地更新至 HTML 页面中而不再需要刷新页面，如图 1.2 所示。

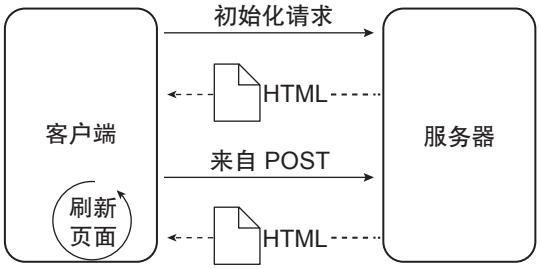

图 1.1　传统网页的生命周期

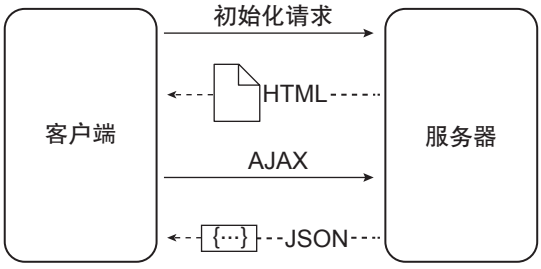

图 1.2　SPA 网页的生命周期

在目前非常注重即时满足的社会中，SPA 似乎很有意义：整个页面只需加载一次，并且一旦加载完成，进一步的交互将非常快；客户端可以完成更多的处理，减少服务器端的负载。尽管 SPA 技术已经存在了一段时间，但大多数网站还在使用传统网页机制，因此 SPA 仍然令人惊叹。

1.1.2　前端需要深入业务

过去传统网站技术流主要侧重于服务器端技术，网站的大部分功能是使用后端语言编写的。然而，前端开发仅限于构建模板、样式化和一堆奇怪、晦涩难懂的 JavaScript 代码片段。虽然这已经是一段时间以来的最佳解决方案，但是 Web 浏览器性能不佳，无法处理运行现代网站所需的复杂操作，造成了前端工程师在过去更多地只是负责一些页面样式，甚至部分美工的工作，而完全不了解数据及业务的逻辑。

然而，从 Microsoft 的 IE 7 到 Google 在 Chrome 中推出的 V8 引擎，浏览器已经取得了长足的进步。它们已经从局限于显示内容和基本 DOM 操作的简单页面发展成为操作系统的扩展，如 Node.js。

功能更强大的 Web 浏览器的出现，允许浏览器拥有更多的处理能力，从而催生了许多诸如 TypeScript、WebAssembly、SPA 等革新产物，将原本需要服务器端进行数据处理及与 HTML 模板整合等表现逻辑转移到了浏览器。

因此，前端开发和后端开发就能明确被分离，摆脱了原先紧密耦合的模型。通过将前端开发和后端开发解耦，还可以消除技术限制，这样在客户端和服务器上完全可以使用不同的技术栈。例如，使用 Python 或 Java、Golang 构建后端，而使用 Vue、React 或 Angular 构建前端，这在以前是根本不可能实现的。由于浏览器承担了越来越复杂的业务逻辑，用户体验的重要性被提到了前所未有的高度，在越来越多的互联网产品中，前端逐步成为影响项目的主导因素。

1.1.3 SPA 如何工作

作为加载单个 HTML 页面并在用户与应用程序交互时动态更新的 Web 应用程序，SPA 的核心是可以根据导航操作（如单击链接）重新呈现内容，而不需要向服务器发出获取新 HTML 的请求。

虽然 SPA 程序的实现机制各不相同，但其大多数依赖相同的浏览器行为和原生 API 来实现核心功能。如果读者之前接触过类似 React 或 Vue.js 等 SPA 框架，先暂时忘却它们，了解以下内容是掌握 SPA 程序工作机制的关键。

SPA 程序可以使用外部源（即 URL location）获取或传递状态，也可以在内部跟踪状态。本文将重点介绍基于 URL location 的 SPA 程序。倘若读者对这些概念有些云里雾里，请继续看下面的例子。

对于 SPA 来说，一定只有一个入口，内部跟踪状态在这方面有某些限制。单个入口意味着访问应用程序总是从首页或根页面开始。在使用内部跟踪状态的应用程序中导航某个页面，不存在路径表示。如果想对外分享一些内容，那么访问该内容的另一个人会从该程序的根页面开始访问，所以不得不向对方解释如何访问想要的内容，如图 1.3 所示。

基于 URL location 的 SPA 可以共享一个链接，并确信打开该链接的任何人都会看到相同的内容，因为在导航某个页面时，位置总是在更新（假设用户具有相同的权限来查看内容），如图 1.4 所示。

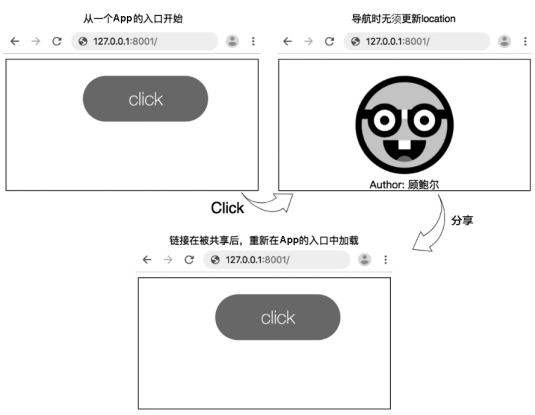

图 1.3 基于内部跟踪状态的 SPA 总是从程序的根页面加载

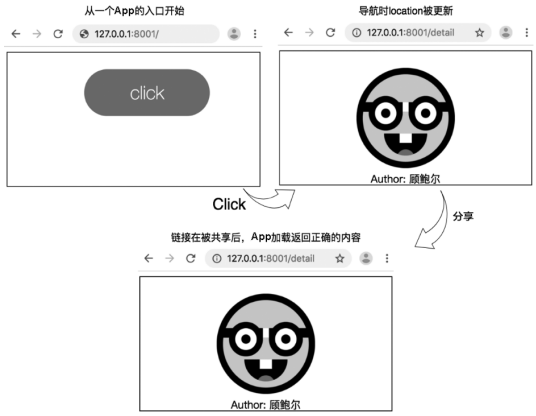

图 1.4 基于 location 的 SPA 可以立即呈现所需的内容

下面从几个方面来介绍 SPA 的工作机制。

1. location 入门

通常用户是与浏览器地址栏中的 URL 进行直接交互的，而不必关心 SPA 内部通过 window.location 来实现不同路径的解析和跳转。

如图 1.5 所示，location 对象只有 3 个属性对 SPA 很重要：路径名（pathname）、散列（hash）

和搜索项（search，通常称为 QueryString）。对于 SPA 而言，导航到应用程序中的任何位置都是定期执行的（如单击一个链接并让浏览器处理它），因此可以忽略主机名和协议。路径名通常是这 3 个属性中最重要的一个，因为它决定页面呈现什么内容。

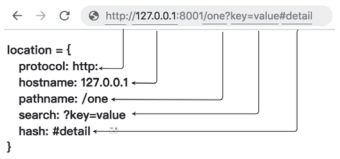

图 1.5　从 URL 直接映射到 window.location 的属性中

搜索项和散列对提供额外的数据更有用。例如，在 URL /images?of=emoji 中，/images 路径名指定页面应该呈现的图像，而 ?of=emoji 作为搜索项，在这里表示页面呈现相应的图像类型。

2. 路由匹配

SPA 通常依赖于路由器。路由器由路由组成，路由描述了应该匹配的 location。路径可以是静态（/about）路径或动态（/album/:id，其中，id 的值可以是任意数值）路径。第三方库 path-to-regexp 是创建路径的一个非常流行的解决方案。目前市面上的 SPA 框架（如 React 和 Vue.js），其路径匹配的实现都是用 path-to-regexp 作为核心引擎。例如：

```
const routes = [
   { path: '/' },
   { path: '/about' },
   { path: '/album/:id },
];
```

路由匹配是将当前 location（通常只比较路径名）与路由器的路由进行比较，以找到匹配的路径。匹配路由后，路由器将触发应用程序重新呈现内容。在实际情况下，这方面的实现在很大程度上取决于路由器。例如，路由器可能会使用“观察者模式”。在该模式中，开发者为路由器提供一个函数，该函数知道如何触发应用程序并重新呈现内容，路由器将在匹配路由后调用该函数，如图 1.6 所示。

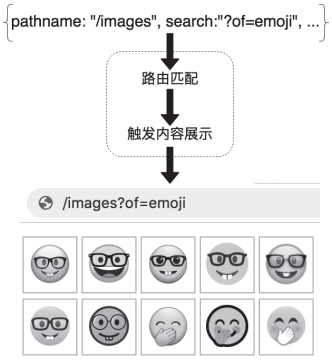

图 1.6　应用程序根据与 location 匹配的路由呈现内容

3. 浏览器如何处理 location

页面导航有一个很有趣的现象。例如，当单击一个链接时，浏览器会自动将这个行为附加到 Click 事件来触发导航，也可以阻止其默认行为并附加自己的处理事件（在现代浏览器中使用 event.preventDefault()）。在了解单击处理如何触发导航之前，我们需要了解一下浏览器通常如何处理导航。

每个浏览器的工具栏中都有两个按钮来控制页面的“后退”和“前进”，以方便用户浏览页面上下文。其主要维护着一个会话历史（session history）记录。这个会话历史记录本质上就是一个数组，存储着有关位置的信息：URL、关联文档、序列化状态和其他一些属性。当浏览页面时，每次浏览记录都作为一个条目记录在该数组中，用索引来指定其位置，并且用户还可以跟踪当前浏览的状态，如图 1.7 所示。

这里还需要进一步针对条目中的信息做一些补充。如图 1.8 所示，当浏览器进行导航时，请求会被发送到服务器，浏览器会根据其响应创建文档对象（document），该对象提供了多种用于交互的方法。例如，用户可以通过 window.document 属性对其进行访问。

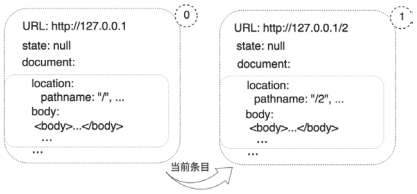

图 1.7　每次浏览记录作为条目组成的数组

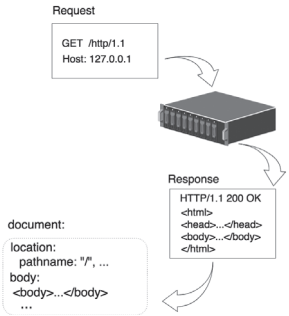

图 1.8　从 Response 创建文档

接下来继续探讨一下浏览器的会话历史记录。当用户单击链接并在页面中导航时，浏览器内部就会构建一个会话历史记录。每次导航都向服务器发出请求并创建一个新条目（包括一个新文档），如图 1.9 所示。

这里需要注意的是，每次导航所显示的不同的文字意味着不同的文档。当单击浏览器的“后退”按钮时，浏览器会通过 current.index–1 将导航后的当前条目作为新条目进行加载，如图 1.10 所示。

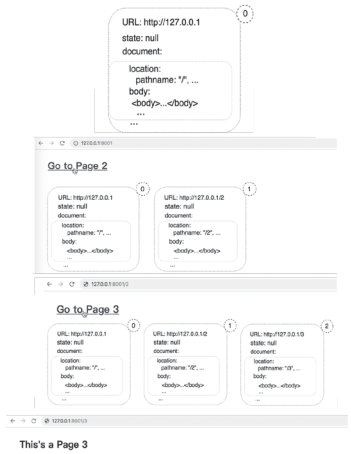

图 1.9　每次导航都作为新的条目附加进会话历史记录中

图 1.10　单击“后退”按钮将切换条目 (或文档), “前进”按钮具有相同的行为

单击页面中的某个外部链接是用户浏览网站时很常见的动作，那么此时浏览器会怎么处理会话历史记录呢？如图 1.11 所示，当前浏览条目之后的所有记录都会被删除。

然而，如果导航到与当前 location 完全相同的位置（即路径名、搜索项和散列都相等），则将替换当前条目，而不会影响后续的条目记录。

如图 1.12 所示，为了让大家能直观地观察到这一变化，Page 2 的网页内容较之前有了一些变化，这个变化主要体现在 document 中，路径名、搜索项和散列并没有改变。以上就是基于 URL location 的工作方式，但还没有结束，下面有必要探讨一下 History API，因为它是浏览器实现这些机制的关键。

图 1.11　当单击外部链接时，将删除当前条目之后的所有记录

图 1.12　单击与当前位置完全相同的链接，将替换当前条目

4. History API

早期的 SPA 依赖这样一个情况：直接修改 window.location，以改变 location 中的信息，使得浏览器只会创建一个新的 location 条目，无须向服务器发送请求。这虽然可以实现，但这种刻意粗暴的做法最终达到的效果不尽如人意，因此 HTML 5 加入了全新的 History API，目的是更好地支持 SPA。与为每个 location 创建新文档不同，History API 还支持更新当前 document 来反映新位置，从而也能复用 document 的特性，如图 1.13 所示。

图 1.13　通过 History API 导航时，可以复用当前 document

History API 有 3 个核心函数：pushState()、replaceState() 和 go()。这些函数及 API 的其他部分可以通过 window.history 访问。

注意：尽管目前几乎所有主要的现代浏览器都支持 History API，但还有一些特例需要关注，如 IE 9 及其以下版本浏览器不支持这一特性。不过值得庆幸的是，Microsoft 已经决定放弃 IE 9，更直接地说应该是放弃整个 IE 而全力转向 Edge（就在本书撰写时，基于 Chromium 内核的 Edge 正在公测中）；此外，Opera Mini 在客户端上无法运行 JavaScript，因此它也不支持 History API。

pushState() 与 replaceState() 具有相同的函数和参数。

第一个参数是 state，如果不想传递任何状态，则传递 null。这里保持应用程序状态似乎看起来比较合理，但是有一些注意事项将在后面讨论。

第二个参数是 title，但浏览器其实并没有用到它。

第三个参数是 path，定义导航的目标路径。在这里，path 可以是完整的 URL、绝对路径或相对路径，但它必须位于相同的应用程序中（即相同的协议和主机名）；否则，程序就会抛出 DOMException 错误。例如：

```
history.pushState(null, '', '/next-location');
history.replaceState(null, '', '/replace-location');
history.pushState({ msg: 'Hi!' }, '', '/greeting');
history.pushState(null, '', 'https://www.google.com');
// 抛出 DOMException
```

history.pushState 在当前条目之后向会话历史记录添加一个条目。如果当前条目之后有条目，则在推送新位置时会丢失这些条目。这是单击锚点时的正常行为，如图 1.14 所示。

replaceState() 替换修改会话历史记录中的当前条目。当前条目之后的任何条目都不会受影响，如图 1.15 所示。

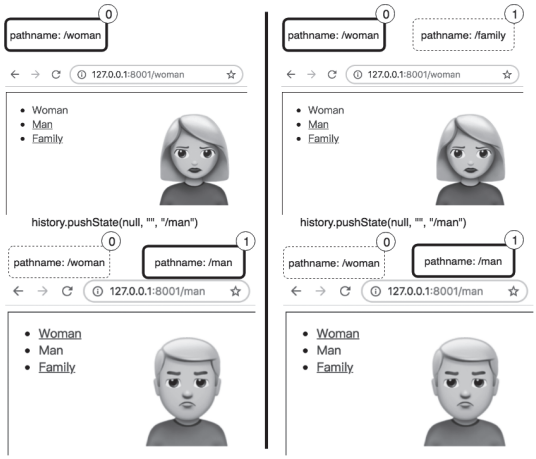

图 1.14 pushState() 在当前条目之后推送一个条目

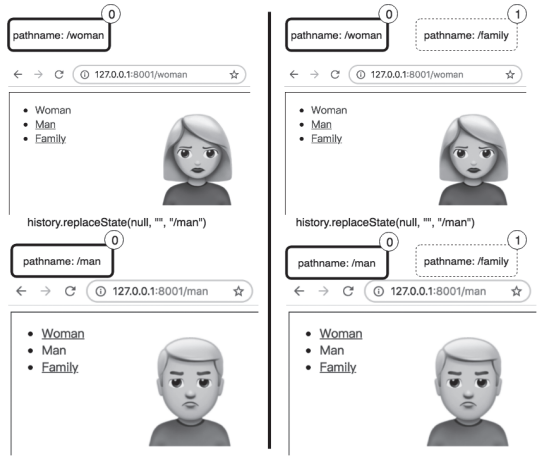

图 1.15 replaceState() 替换当前条目

go() 是用于控制浏览器“前进”按钮和“后退”按钮的任务编程接口。例如，其关闭一个基

于 URL location 的模态对话框之后，页面随即返回至前一页时会很有用，如图 1.16 所示。

go() 接收一个整数类型的参数，即当前要导航的条目的数量。参数为正数，表示向前导航；参数为负数，表示向后导航；参数为 0（或未定义），表示重新加载页面。

```
go(-1); // 后退
go(1); // 前进
go(-10); // 返回
go(0); // 刷新
go(); // 刷新
```

另外，还有 history.back() 和 history.forward() 方法，它们的用途分别与 history.go(−1)、history.go(1) 的用途一样。

location 条目的属性之一是 state（状态）。pushState() 和 replaceState() 方法中也有参数 state。状态是附加到任何条目的数据，它不会因为页面被导航后而丢失，所以如果向条目添加状态，然后导航，再返回该条目，该状态仍然存在。状态会附加到带有 pushState() 和 replaceState() 的条目。这里可以使用 history.state 访问当前条目的状态，如图 1.17 所示。

图 1.16　go() 在知道可以跳转回以前的位置时非常有用

图 1.17　状态可以附加到 location

状态也会存在一些限制。首先，它必须是可序列化的，这意味着数据可以转换为字符串。其次，它有大小限制。开发人员在开发的过程中可能不需要担心这些限制，但必须清楚它们的存在。最后，当用户直接导航到某个位置时，它的状态为 null，如果页面导航依赖于内部跟踪状态来呈现，那么直接导航将出现问题。

一个好的经验法则是，在设计页面路由导航时，基于内部跟踪状态来导航的机制应该用于一些不方便直接通过 URL 的路径就能访问的内容。例如，某个页面通过 URL location 类型进行页面导航，其路径是“/detail/:id”，或许有各种各样的原因不希望用户通过 id 直接访问，就可以考虑把该 id 存入 state 属性。在处理一些现实场景时，如一些邮箱系统（Hotmail、Gmail 等）的页面导航切换，也会采用该方案。

SPA 程序如何利用 History API？ SPA 程序可以使用 event.preventDefault() 覆盖原生行为的锚添加一个自定义的单击处理程序。处理程序可以调用 history.pushState() 或 history.replaceState() 来执行一些导航策略，从而不必触发服务器请求。

但是，History API 只更新会话历史记录，因此处理程序还需要与路由器交互，让路由器知道新的 location 在哪里。许多路由器使用 History API 来封装这些步骤，像 React 或 Vue.js 等 SPA 程序框架，都会通过封装提供多种方式将处理程序添加到锚中。下面笔者使用 Vue.js 尝试编写一个特殊的 link 组件，并将单击处理程序直接附加到组件上。

```
# Vue example
const Link = {
  name: 'router-link',
  props: {
    to: {   // 该组件接收一个类型为“字符串”的必填参数 to 属性
      type: String,
      required: true
    }
  },
  template: '<a @click="navigate" :href="to">{{ to }}</a>',
  // 指定 URL 从 to 属性中获得
  methods: {
    navigate(evt) {  // 定义一个 navigate 方法
      evt.preventDefault();  // 阻止 click 事件默认打开 URL 的行为
      // pushState() 在不刷新浏览器的情况下，在队列中创建新的条目并将其附加至 to 属性
      window.history.pushState(null, null, this.to);
    }
  }
};
// <router-link to="/vue-example"></router-link>
```

用户可以为单击事件添加一个全局事件监听器，用于检测应用程序内导航、覆盖默认行为并使用 History API 调用替换它。实际情况稍微复杂一些，因为并不是所有的单击事件都应该被覆盖。History API 与覆盖本地单击行为相结合，使应用程序内导航变得容易。但是，我们还需要担心另一种类型的导航：用户单击浏览器的“前进”按钮和“后退”按钮。

单击“后退”按钮和“前进”按钮（以及调用 history.go()）时，浏览器会发出一个 popstate 事件。为了检测这些，用户需要向 window 对象添加一个事件监听器。当事件监听器被调用时，会话历史记录被更新，所以用户需要做的就是让路由器知道位置已经改变。如果用户使用地址栏手动更新位置，该导航将创建一个新 document。

History API 只防止使用共享同一个 document 的条目重新加载，这意味着调用 history.go() 并单击“前进”按钮 /“后退”按钮，将导致整个页面重新加载。利用 SPA 程序来控制导航，这样就可以复用 document，而不是向服务器发送请求。

简而言之，在 SPA 程序内，导航是通过 History API 实现的，即用户单击链接覆盖锚的默认单击行为，以使用 pushState() 或 replaceState() 进行 location 条目的变更，而 popstate 事件监听器用于检测浏览器的“前进”按钮 /“后退”按钮的状态，一旦触发，单击处理程序和事件监听器都会将导航通知给路由器，当位置发生变化时，路由器就会进行路由匹配，并最后重新呈现相关内容。

本书并不会讨论关于 History API 的更多细节，但正所谓万卷不离其宗，了解本节内容能帮助读者理解 SPA 程序的运作机制，为读者日后学习各种 SPA 框架打下基础。读者倘若对本节内

容有兴趣，可以看一下 Mozilla 的 History API 文档；如果想亲自动手，可以阅读 WHATWG（Web Hypertext Application Technology Working Group，网页超文本应用技术工作小组）History 实现规范，其提供了很多有用的信息。

另外，市面上也有许多封装 History API 的第三方库，用户可以查看它们是如何工作的。它们通常具有比本文介绍的更多的功能。例如，History 包可以被 React Router 和其他项目使用。Vue.js 的 vue-router 也有自己的 History API 相关实现。

1.1.4　闲聊 MVVM 设计模式

在现代 Web 开发架构中，SPA 的技术发展并不是一蹴而就的。在了解 SPA 与 MVVM（Model-View-ViewModel，模型—视图—视图模型）设计模式的关系之前，有必要简单回顾一下 MVC（Model-View-Controller，模型—视图—控制器）的设计模式及其演进的过程。作为 MV* 框架系列（这里泛指 MVC、MVT、MVP、MVVM）的鼻祖，MVC 设计模式对目前主流的 MVVM 框架实现的 SPA 所产生的技术变革有着非常深远的影响。在很长的一段时间内，我们会在服务器端采用基于 MVC 设计模式的 Web 框架与客户端浏览器进行页面交互，其大致的流程可以简单概括如下。

- Model：用于获取及处理数据。
- View：用于页面的渲染及展示，原则上不处理任何与数据有关的逻辑。
- Controller：不同的页面路径会对应不同的Controller，然后Controller会绑定相应的View进行页面渲染。

用户访问网站的某一页面时，浏览器都会通过路径传递到服务器端的 Controller，并路由到相应的 View，View 从 Model 中获取已经整理好的数据，最后生成完整的 HTML 文档返回浏览器进行显示。或许用户每次都向服务器端请求获取完整的 HTML 文档，对于大型网站来说，其性能和用户交互性是非常糟糕的，但在很早以前，MVC 的出现彻底让业务逻辑和 UI 解耦，代码结构的分离也使开发大型而复杂的 Web 应用成为可能。当软件工程的问题得以解决之后，人们开始关注性能所带来的更好的用户体验。

其后，Microsoft 率先发明了 AJAX（Asynchronous JavaScript and XML，异步的 JavaScript 和 XML）技术。得益于 Google 的大力推广，AJAX 技术彻底打破了原先必须通过 Form 表单与服务器端进行数据交互时页面不停刷新的尴尬局面。这种支持异步请求交互数据的方式在当时可以说是一场重大的技术革命，也为后续 SPA 技术的日渐成熟奠定了基础。

人类的伟大之处就在于人类永远不会安于现状，过去前端与后端交互的频繁页面刷新带来的糟糕体验、多而重复的服务器资源请求导致的资源浪费，让人们慢慢发现是否可以在页面之间无缝切换而无须再向服务器端进行请求，那么前端 MVC 的解决方案就自然而然地应运而生。较具有代表性的框架有 Ember.js、AngularJS 等。

MVC 在前端技术应用上的理念和服务器端是一致的。如果抛开一些框架的理论，可以简单地把 HTML 看作 View，JavaScript 作为 Controller 用于和 HTML 的 DOM 节点交互来改变显示内

容，而 Model 主要由 AJAX 负责与服务器端进行数据的传递。事实上，从前端技术的角度来看，MVVM 的理念与 MVC 并没有太多的变化，但 Controller 的机制发生了改变并被 ViewModel 概念替代。

那什么是 ViewModel 呢？每次改变 HTML 的内容渲染都需要用 JavaScript 手工操作 DOM 节点，而 ViewModel 可以通过数据绑定的机制让这一切自动实现（对于不同的 MVVM 框架来说，实现方式会有所不同，这在以后的 Vue 章节中会详细介绍 ViewModel 的机制），这样一来，开发者可以把更多的精力用在关注实现业务逻辑上。

MVVM 与其他设计模式的区别，并不是本书关注的内容，笔者也并不提倡对设计模式的盲目崇拜，尤其像 SPA 的技术实现，无论是 MVC、MVVM 还是其他 MV* 设计模式，只要遵循良好的软件工程实践，确保代码可读性和扩展性，设计模式也就形成了。从目前的工程经验来看，对于绝大多数开发者来说，MVVM 框架也许是目前实现 SPA 程序较好的方式。

1.1.5 与服务器端通信

SPA 的发展得益于 AJAX 技术的成熟和普及，HTML 5 赋予了原生 API 对页面动态切换的完美支持，进一步降低了客户端与服务器端交互频繁所带来的性能问题。接下来，本节将和大家探讨 AJAX 技术在 SPA 客户端与服务器端进行数据交互通信时的改进和变化。

AJAX 可以说是当今 Web 应用程序的主干，也是用于客户机和服务器之间的重要通信方式。在 JSON 未普及之前，用于服务器端和客户端交互的数据交换格式主要以 XML 为主。因此，了解整个 AJAX 是如何工作的、有哪些挑战及如何解决问题是读者必须关注的课题。

笔者会安排如下几个议题循序渐进地分享 AJAX 的核心技术。

- 调用AJAX进行工作。
- 处理回调。
- 使用Promise。

1. 调用 AJAX 进行工作

调用 AJAX 比较简单。通过下面三行代码就能实现与服务器端的通信。

```
var xhr = new XMLHttpRequest();
xhr.open(methodType, URL, async);
xhr.send();
```

（1）创建 XMLHttpRequest 的实例。这个实例可以触发 XHR 调用并获得响应。

```
var xhr = new XMLHttpRequest();
```

（2）使用方法类型 methodType 连接到 URL，本书稍后将讨论 async。方法类型有四种，即 GET、POST、PUT 和 DELETE。其中，GET、POST、PUT 通常是大家接触较多的方法类型。

URL 表示目标服务器地址。

```
xhr.open(methodType, URL, async);
```

此时，浏览器将向服务器发出 HTTPRequest。xhr 将保存关于该请求的所有信息，如 HTTPConnection 状态、HTTPResponse 状态和代码。

```
xhr.send();
```

如果需要执行两个 AJAX 调用，则需要重复以上 3 步骤。

为了处理服务器发送的响应，这里需要在 xhr 实例上添加回调函数。

```
xhr.onreadystatechange = function(){
if (xhr.readyState === 4){
       if (xhr.state === 200){
           console.log("received the response", xhr.responseText);
       } else {
           console.log("error in processing the request");
       }
    } else {
        //console.log("waiting for the response to come");
   }
}
```

每当 HTTPConnection 发生更改时，相应程序都会在 xhr 实例上调用 onreadystatechange。这里需要注意 3 件事。

readyState 表示连接的状态。它的值为 0 ～ 4。当连接和请求发生更改时，该值也会随着发生变化。该值达到 3，意味着数据开始从服务器发出；该值达到 4，表示所有数据都已到达，服务器已处理完请求，连接已关闭。此时可以检查请求是否被成功处理。

status 可以告知该请求是否被成功处理。如果状态是 200，说明请求被成功处理。任何其他代码都表示请求失败。

xhr 实例的 responseText 属性将保存服务器返回的值。服务器可以发送任何类型的数据，如 XML、JSON、纯文本、二进制数据等。根据不同的数据格式，我们可以自行处理该值。

为了让代码变得更实用一些，笔者做了一些简单的封装来获取 GitHub 上的用户信息。代码如下：

```
function makeAjaxCall(url, methodType){
    var xhr = new XMLHttpRequest();
       xhr.open(methodType, url, true);
       xhr.send();
```

```
        xhr.onreadystatechange = function(){
          if (xhr.readyState === 4){
             if (xhr.status === 200){
                console.log("xhr done successfully");
                var resp = xhr.responseText;
                var respJson = JSON.parse(resp);
             } else {
               console.log("xhr failed");
             }
          } else {
             console.log("xhr processing going on");
          }
        }
        console.log("request sent succesfully");
      }
      // 通过 GitHub 提供的 API 来获取用户详情信息
      var URL = "https://api.github.com/users/boylegu";
      makeAjaxCall(URL, "GET");
```

请注意 JSON.parse() 函数（它可将字符串转换为 JavaScript 对象）及 JSON.stringify() 函数（其作用与 JSON.parse() 正好相反）。

2．处理回调

开始接触回调会让人感到很困惑。不过理解了回调的概念之后，学习回调就非常简单了，笔者准备了两个小问题逐一破解。

什么是回调？假设这里有一个函数 A1，它内部调用函数 B。函数 B 正在做一些异步操作，如 AJAX。A1 想知道 B 函数中 AJAX 调用的结果。现在 A1 将另一个函数 A2 作为额外参数传递给函数 B，函数 B 将在处理 AJAX 请求后把结果返回给函数 A2。更进一步地说，假设 A1 希望从函数 B 中获取结果，它就向函数 A2 提供需求细节，当函数 B 完成服务时，函数 B 通过使用一些额外的数据调用 A2 通知 A1，如图 1.18 所示。

为什么需要在 AJAX 中进行回调？因为如果没有回调，开发人员每次都需要重复地编写大量且相同的 AJAX 代码。这时候就可以创建一个通用的 AJAX 函数，该函数将 AJAX 细节和回调引用作为输入。在完成调用之后，它调用回调函数，以便调用者可以使用 AJAX 调用的结果继续调用。在前面的代码示例中，使用 makeAjaxCall 函数获取用户详细信息。

现在假设要求显示该用户在 GitHub 上的所有代码仓库，为此，需要使用 GitHub 提供的另一个 API 来获取代码仓库列表。显然，谁都不愿意再编写另一个类似 makeAjaxCall 函数的函数来执行服务器调用，而是继续使用 makeAjaxCall 函数。下面这段代码将显示完整的回调机制。

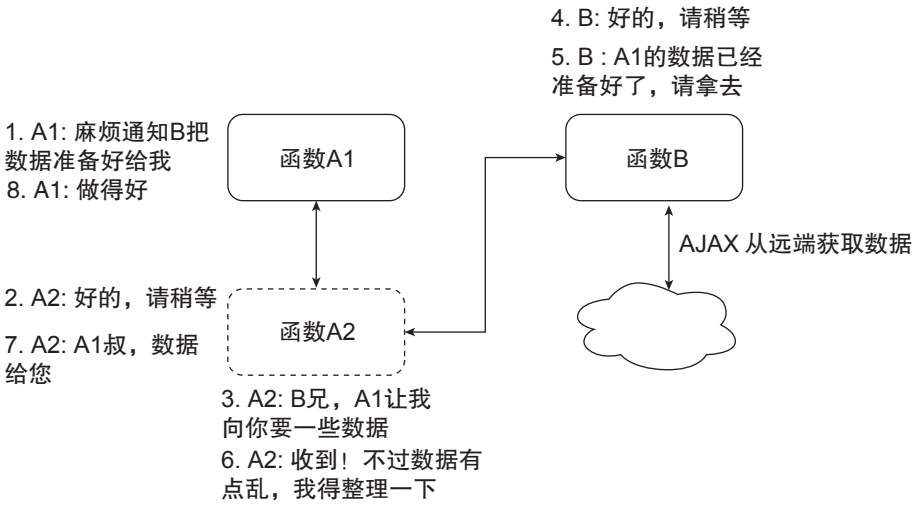

图 1.18　回调机制

```
function makeAjaxCall(url, methodType, callback){
   var xhr = new XMLHttpRequest();
      xhr.open(methodType, url, true);
      xhr.send();
      xhr.onreadystatechange = function(){
        if (xhr.readyState === 4){
           if (xhr.status === 200){
              console.log("xhr done successfully");
              var resp = xhr.responseText;
              var respJson = JSON.parse(resp);
              callback(respJson);
           } else {
              console.log("xhr failed");
           }
        } else {
           console.log("xhr processing going on");
        }
      }
      console.log("request sent succesfully");
   }
document.getElementById("userDetails").addEventListener("click", function(){
 var userId = document.getElementById("userId").value;
 var URL = "https://api.github.com/users/"+userId;
 makeAjaxCall(URL, "GET", processUserDetailsResponse);
```

```
});
document.getElementById("repoList").addEventListener("click", function(){
 var userId = document.getElementById("userId").value;
 var URL = "https://api.github.com/users/"+userId+"/repos";
 makeAjaxCall(URL, "GET", processRepoListResponse);
});
function processUserDetailsResponse(userData){
 console.log("render user details", userData);
}
function processRepoListResponse(repoList){
 console.log("render repo list", repoList);
}
```

在上面的示例中，调用 makeAjaxCall 函数的地方有两个。对于这两个场景，服务器响应的处理是不同的。makeAjaxCall 函数在这里作为一种服务函数，它接受 AJAX 细节和回调引用。当完成 AJAX 调用时，它通过调用回调引用来通知调用者，使用回调引用可以创建一个可重用的独立函数，该函数只关注 AJAX 调用。

回调函数可以自行处理任何数据，如显示用户详细信息或 GitHub 仓库列表。通过传递诸如 URL、方法和回调引用等 AJAX 调用细节，可以在 *n* 个位置使用 AJAX 服务函数。回调是将 AJAX 的核心逻辑与业务逻辑分离比较常用的办法。

不幸的是，当进行一系列 AJAX 调用时，其中一个调用依赖于前一个调用，因此处理回调变得非常困难，在维护多个回调引用和处理多个成功和错误条件时可能会遇到困难。Promise 是管理多个 AJAX 调用的更好方法，也是目前前端技术开发较常用的技术之一。

3．使用 Promise

Promise 用于解决具有多个回调的问题，并提供更好的方法来管理成功和错误条件。它本身就是一个异步函数（如 AJAX）返回的对象，有 3 种状态。

- Pending：该异步操作正在进行中。
- Resovled：该异步操作成功完成。
- Rejected：该异步操作完成但发生了异常错误。

使用 Promise 对象可以分为两个部分。

第一部分：将其封装为异步函数。

- 创建Promise对象。
- Async函数返回Promise对象。

如果异步操作成功，则通过调用 Promise 对象的 resolve 方法去解析它；如果异步操作错误，则通过调用 Promise 对象的 reject 方法直接拒绝。

下面是这部分 Promise 的代码。

```
/ 第一部分 将其封装为异步函数
function makeSomeAsyc(){
  // 创建 Promise 对象
  var promiseObj = new Promise(function(fullfill, reject){
      ... // 在这里添加 Async 代码
      ... // 成功时，调用 fullfill 方法进行解析
      ... // 出现错误时，调用 reject 方法来拒绝
  });
  //Returns Promise object
  return promiseObj;
}
function handleSuccess() {   ...   }
function handleError() {   ...   }
```

第二部分：外部调用。

- 调用函数并获得Promise对象。
- 使用then方法在Promise对象上附加成功处理程序和错误处理程序。

下面是这部分 Promise 的代码。

```
// 第二部分：外部调用
var pobj = makeSomeAsyc();
// 将成功处理逻辑附加到 Promise
pobj.then(handleSuccess, handleError);
```

将 Promise 与前面实现的 makeAjaxCall 函数结吉，看看两者会擦出怎样的火花。

```
function makeAjaxCall(url, methodType){
      var promiseObj = new Promise(function(resolve, reject){
            var xhr = new XMLHttpRequest();
            xhr.open(methodType, url, true);
            xhr.send();
            xhr.onreadystatechange = function(){
            if (xhr.readyState === 4){
               if (xhr.status === 200){
                  console.log("xhr done successfully");
                  var resp = xhr.responseText;
                  var respJson = JSON.parse(resp);
                  resolve(respJson);
               } else {
                  reject(xhr.status);
                  console.log("xhr failed");
```

```
                }
            } else {
                console.log("xhr processing going on");
            }
        }
        console.log("request sent succesfully");
    });
    return promiseObj;
}
document.getElementById("userDetails").addEventListener("click", function(){
 var userId = document.getElementById("userId").value;
 var URL = "https://api.github.com/users/"+userId;
 makeAjaxCall(URL, "GET").then(processUserDetailsResponse, errorHandler);
});
document.getElementById("repoList").addEventListener("click", function(){
 var userId = document.getElementById("userId").value;
 var URL = "https://api.github.com/users/"+userId+"/repos";
 makeAjaxCall(URL, "GET").then(processRepoListResponse, errorHandler);
});
function processUserDetailsResponse(userData){
 console.log("render user details", userData);
}
function processRepoListResponse(repoList){
 console.log("render repo list", repoList);
}
function errorHandler(statusCode){
 console.log("failed with status", status);
}
```

注意，makeAjaxCall 函数返回一个 Promise 对象，并且不接受任何回调，且只能被解析或拒绝一次，因此，我们可以在 Promise 对象上添加多个成功和错误处理代码，每一个都将按照注册时的顺序调用。

在编写复杂的异步代码与服务器端进行交互时，回调会变得混乱，尤其是构建大型 SPA 程序时，Promise 是目前不得不掌握的主流技术。

1.1.6 SPA 的优点和缺点

以上内容介绍了 SPA 程序的方方面面，在这里，笔者结合自己的经验简单概括一下 SPA 的优点和缺点。

1. SPA 的优点

（1）极佳的用户体验。

因为大多数资源（HTML、CSS、JavaScript）在应用程序的整个生命周期中只加载一次、速

度极快，所以用户体验的过程就像使用脱机版的本地程序一样流畅。

（2）减轻服务器的压力。

无须通过服务器呈现页面，剩下的就只有数据和服务器端来回传输。

（3）更容易调试。

使用调试工具可以更容易地监视网络操作，研究与之相关的页面元素和数据。

（4）大幅提高开发效率。

更彻底的前、后端分离，使开发人员可以为 Web 应用程序、移动应用程序复用相同的代码，大幅提高开发效率。

2. SPA 的缺点

然而，在很多时候，SPA 程序并非是万金良药，它的缺点也不少，主要有以下几个方面。

（1）无法很好地支持 SEO。

SPA 程序可以看成只有一个页面，而目前主要搜索引擎 (如谷歌、百度、必应) 支持索引的页面数量也被限制在一个页面，这意味着所有的网站 SEO 排名将被困在网站的单个页面。总体来说，只想对几个关键词进行排名可能比较容易，但如果希望支持大量关键词，那么针对单个页面进行排名就会非常困难。

（2）学习门槛不低。

对于之前长期依赖使用 jQuery 或刚接手前端开发的开发者来说，理解并使用 SPA 框架或许一开始并不容易，再者，业务逻辑从后端向前端的转移，导致对前端技术的能力要求也进一步提高。尽管像 Vue 这类的框架已经被设计得足够简单，但笔者看到过很多开发者会耗费大量的时间来研究如何在 SPA 中引入 jQuery，对 SPA 程序所赋予的组件复用、组件间的通信、状态等特性嗤之以鼻，因此很难从中获益。

（3）严重依赖 JavaScript。

由于 JavaScript 及 API 受限于各个浏览器版本，不能强制要求所有访问 Web 应用的用户群总是使用最新版本的浏览器，在风险不可评估的情况下，SPA 程序所带来的良好用户体验也会适得其反。

（4）扩展网站内容的能力十分有限。

SPA 程序所呈现的内容主题或风格较为单一，可能在开始时这并不是问题，但如果企业决定扩展内容更多、更复杂的设置，SPA 程序就不能很好地满足这些内容扩展。虽然可以简单地扩展单页（主页），让它有越来越多的内容，但事实上无法做得太多。如果计划向网站添加大量内容，SPA 程序可能不是最佳选择。

综上所述，相信大家对 SPA 程序的优缺点已经有所了解，然而，由于国内的搜索引擎算法及技术相比国外起步较晚，而国内很多针对 ToC 的产品又严重依赖于 SEO，并且包含着各式繁杂的营销类页面和视觉炫酷的广告，即使多页应用（MultiPage Application,MPA）程序会在用户体验、页面加载性能上带来诸多弊端，但对于绝大多数的普通用户来说，若没有在 SPA 程序上获益，也

就自然不在乎它们在体验方面的差距。

另外，从 MAP 架构转为 SPA 程序，对开发者的技术能力也是一种挑战，好在目前的 AJAX 足够强大，以至于可以在 MPA 的部分页面与服务器之间传输大量数据，传统的技术方案自然也有较为成熟的设计模式去解决工程化的问题。

总体而言，在收益和投入没有达到一定比例的情况下，国内很多互联网公司以及电商平台项目依然采用 MPA 程序技术，而一些具有特定应用场景的 Web 项目反而比较适合 SPA 程序，如电子邮箱系统、内部管理系统、社交类应用等。

1.2 异步与协程

降低高昂的信息化成本，合理应用服务器资源，打造一款具备良好性能的企业级应用程序，一直是大家所追求的目标，对于如何“榨干”服务器的“每一滴油”，在计算机科学领域也是一门极具魅力的综合性学科。

常被作为饭后谈资、面试必考的话题就是“并发”，这也是让众多计算机编程高手们相互切磋、流连忘返的“侠客岛”。然而，在现代 Web 应用程序开发中，语言、技术框架已经屏蔽了许多技术复杂度，越来越多的开发者被挡在了通往高手之路的门外，成为“优秀的 API 工程师”。于是，平庸和优秀之间逐步形成了技术阶级的壁垒。由此，本节将重新为读者拾遗补缺，更系统地介绍现代软件并发编程的知识。

1.2.1 程序、进程、线程与协程

您是否经常听到与计算机程序相关的“线程”这个术语，但又不了解它的确切含义？您可能知道线程与程序、进程有着某种密切的关系，但如果不是计算机科学专业的学生，那么您的理解或许仅限于此。

假设读者是一名程序员，那么了解这些术语的含义是绝对有必要的，当然它们对于普通计算机用户也相当有用，因为利用这些术语可以查看和理解 MAC 上的活动监视器、Windows 上的任务管理器或 Linux 上的 top，可以帮助排除哪些程序在计算机上造成了问题，或者确定是否需要安装更多内存才能使系统运行得更好。

接下来，笔者将简化和概括这些概念，用较为通俗易懂的方式进行介绍。

1. 程序

大家可能知道，程序就是存储在计算机上的代码，用于完成特定的任务。程序有许多类型，如帮助计算机运行的程序和操作系统的一部分程序，以及完成特定任务的其他程序。这些基于特定任务的程序也称为应用程序，包括文字处理、Web 浏览或向另一台计算机发送邮件等程序。

程序通常以计算机执行的形式存储在磁盘或非易失性内存中。在此之前，它们使用 C、LISP、Pascal 等编程语言创建，或者使用包含逻辑、数据、设备操作、递归和用户交互的指令构建。最终

的结果是文本文件的代码被编译成二进制形式，以便在计算机上运行。

另一种类型的程序称为解释程序，它不是为了运行而预先编译的，而是在运行时即可解释为可执行代码。一些常见的解释型编程语言有 Python、PHP、JavaScript、Ruby 等。然而，最终的结果是相同的，当程序运行时，它们以二进制形式加载到内存中。计算机的 CPU（中央处理单元）只理解二进制指令，所以二进制形式是程序运行时的必要的形式。

二进制语言是计算机的母语，因为电路的基本状态有开和关两种，所以在计算机中分别用 1 或 0 表示这两种状态（现在有一种量子计算机，它超越了计算中只有 1 和 0 的概念）。

2. 程序与进程是如何一起工作的

进程就是程序，它以二进制的形式加载到计算机内存中，计算机知道如何处理这些二进制代码，结合内存和各种操作系统资源才能运行。操作系统是分配这些资源背后的大脑，并且有着不同的风格，如 macOS、iOS、Microsoft Windows、Linux、Android 等。它负责管理着程序转换为可运行进程所需要的资源和环境。

每个进程都包括一些基本信息，如寄存器、程序计数器和堆栈。寄存器是 CPU 的数据存储位置。寄存器可以保存进程所需的指令、存储地址或其他类型的数据。程序计数器也称为“指令指针”，用于跟踪计算机在其程序序列中的位置。堆栈是一种数据结构，用于存储有关计算机程序的活动子程序的信息。它与堆的进程动态分配内存不同，如图 1.19 所示。

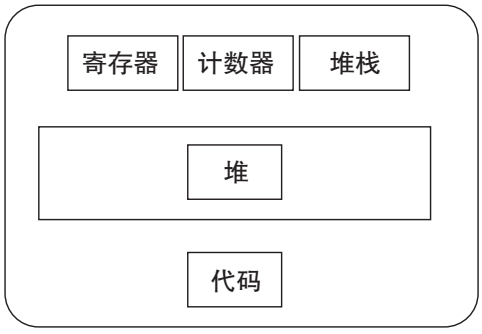

图 1.19　计算机中的进程

一个程序可以有多个实例，每个正在运行的程序实例都是一个进程。每个进程都有一个独立的内存地址空间，这意味着一个进程独立运行，并与其他进程隔离。因此，一个进程不能直接访问其他进程中的共享数据。从一个进程切换到另一个进程需要一些时间来保存和加载寄存器、内存映射和其他资源。

进程的独立性很高，因为操作系统会尽可能地隔离进程，这就意味着当计算机上的某个应用程序冻结或出现问题时，该程序能够在不影响其他程序的情况下退出。

3. 线程是如何工作的

线程是进程中的执行单元。一个进程可以有一个或多个线程，如图 1.20 所示。

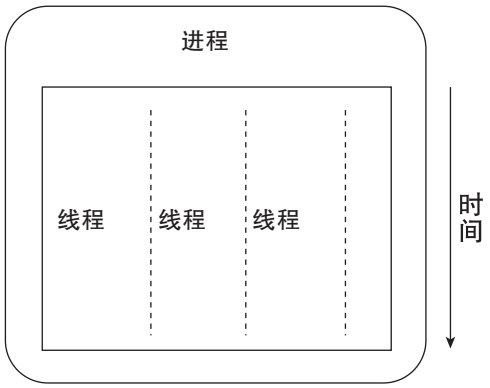

图 1.20　进程与线程

当进程启动时，它被分配内存和资源。进程中的每个线程共享内存和资源。在单线程进程中，进程包含一个线程。过程和线程是一回事，并且只有一件事在发生。

在多线程进程中，进程包含多个线程，并且进程同时完成许多事情（从技术上讲，有时几乎是同时进行的），如图 1.21 所示。

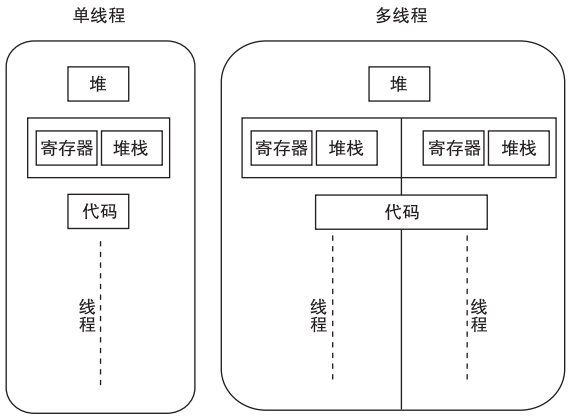

图 1.21　进程中的单线程和多线程

我们讨论了进程或线程可用的两种内存类型：堆栈和堆。区分这两种类型的进程内存非常重要，因为每个线程都有自己的堆栈，但是进程中的所有线程都将共享堆。

线程有时称为轻量级进程，因为它有自己的堆栈，但可以访问共享数据。由于线程与进程和进程内的其他线程共享相同的地址空间，线程之间通信的操作成本很低，这是一个优势。缺点是一个进程中一个线程的问题肯定会影响其他线程和进程本身的生存能力。

4. 线程与进程

下面主要总结一下线程与进程的优缺点，如表 1.1 所示。

表 1.1　线程与进程的优缺点对比

进　　程	线　　程
进程消耗的资源较多	线程更轻量级
每个进程都有各自的内存空间	线程可以使用它们所属进程的内存
进程之间的通信很慢，因为内存地址不同	线程间通信可能比进程间通信更快，因为同一进程的线程与它们所属的进程共享内存
进程之间的上下文切换开销更大	在相同进程的线程之间切换上下文的开销更小
进程之间无法共享同一个内存	线程与同一进程的其他线程共享内存

5. 协程

尽管线程比进程所占用的内存空间更小，但它在多并发的状态下仍旧通过 CPU 进行上下文切换及调度。产生大量的连接数时，频繁切换所导致的内存开销不可小觑，并且其调度的控制权完全依靠 CPU 进行，当程序发生问题时，不利于进行调试和排错，甚至就像数以百计脱了缰的野马到处驰骋，而你可能也束手无策。

协程的出现改善了这些问题，目前协程已经是 Web 应用程序的标配，那它与线程有什么区别呢？协程的上下文切换不再受 CPU 控制，这一控制权被下放给用户，在多并发的情况下，通过编程很容易在单线程下进行多个用户栈之间的上下文切换，在节省内存的情况下，提高吞吐量，更容易对程序进行控制，如图 1.22 所示。

在单个线程中开启多少个协程并没有限制，当然这也取决于系统的内存，好在协程的内存消耗比线程更小，因此单线程可以处理更多的连接数，另外通过扩展进程，协程的威力更是在单机并发上达到非常强悍的地步。

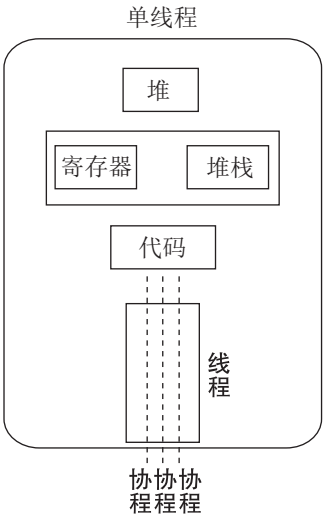

图 1.22　一个进程中单线程和协程的关系图

由于协程是通过用户态实现的，其实现方式也因不同的编程语言支持程度而不同，它的运作方式其实就是由一种事件驱动模型衍生出来的，所以了解事件驱动模型是了解协程技术的重要基础，但目前暂且放一放，因为还有一些关于并发的基本概念需要和读者再聊一聊。

1.2.2　并发基础

读者可能会问，进程或线程是否可以同时运行？答案视情况而定。具有多个处理器或 CPU 内核的系统（与现代处理器一样）可以并行执行多个进程或线程。但是，单个处理器不可能同时真正执行进程或线程。在本例中，使用进程调度算法在正在运行的进程或线程之间共享 CPU，该算法分配 CPU 的时间并产生并行执行的假象。给每个任务分配的时间称为时间片。

在不同任务之间来回进行上下文切换，其速度非常之快，通常是察觉不到的。并行性（真正的同步执行）和并发性（及时交错进程以呈现同步执行）区分了两种类型的实际或近似的同步操作，如图 1.23 所示。

那么，当程序员创建一个希望同时执行多个任务的程序时，如何在进程和线程之间做出选择呢？上面已经讨论了一些差异，下面来看一个实际的例子，这个例子中有一个很多人都使用的程序——谷歌 Chrome 浏览器。

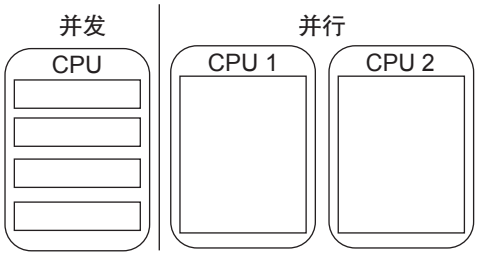

图 1.23　并发与并行

当谷歌设计 Chrome 浏览器时，需要决定如何处理同时需要计算机、通信和网络资源的许多不同任务。每个浏览器窗口或选项卡都与 Internet 上的多个服务器通信，以检索文本、程序、图形、音频、视频和其他资源，并呈现这些数据以供显示和与用户交互。此外，浏览器可以打开许多窗口，每个窗口都有许多任务。

谷歌必须决定如何处理任务的分离。选择在 Chrome 中以一个单独的进程运行每个浏览器窗口，而不是像其他浏览器一样运行一个线程或多个线程，这样做给谷歌带来了很多好处。将每个窗口作为一个进程运行可以保护整个应用程序不受呈现引擎中的 Bug 和故障的影响，并限制从每个呈现引擎进程对其他进程和系统其他部分的访问。在进程中隔离 JavaScript 程序可以防止它占用太多的 CPU 时间和内存，并使整个浏览器没有响应。

谷歌对多处理设计进行了权衡。为每个浏览器窗口启动一个新进程在内存和资源上的固定成本要高于使用线程。谷歌认为这种方法最终将减少内存膨胀。

当内存不足时，使用进程而不是线程还可以更好地利用内存。不活动的窗口被操作系统视为较低的优先级，当其他进程需要内存时，可以将其交换到磁盘。这有助于使用户可见的窗口响应速度更快。如果窗口是线程化的，那么将使用的内存和未使用的内存清晰地分开将更加困难，导致系统性能下降。

图 1.24 所示为运行在 MacBook Pro 上的谷歌 Chrome 进程，其中打开了许多选项卡。一些 Chrome 进程占用了相当多的 CPU 时间和资源，而另一些进程占用的 CPU 资源非常少。从图 1.24 中可以看到，每个进程都包含许多线程。

图 1.24　Chrome 浏览器在 osx 操作系统中的进程占有情况

1.2.3　I/O 漫聊

尽管目前越来越多的技术框架把并发代码的实现封装得足够简单、容易，可稍有不注意仍会带来可怕的后果，尤其是要谨记曾经在世界上发生过的臭名昭著的软件灾难之一——俗称杀人机器的“Therac-25”，就是由并发过程中竞态问题所引起的。本节将和大家继续探讨应用程序的 I/O，这些概念太重要了。

I/O 的本质就是操作一堆文件描述符。首先，在考虑并发技术的方向时应该清楚当下开发的应用程序属于计算密集型还是 I/O 密集型。表 1.2 基本概括了绝大多数的应用场景。

表 1.2　计算密集型与 I/O 密集型的应用场景区分

计算密集型（优先考虑多进程）	I/O 密集型（优先考虑多线程）
科学计算	磁盘读写
数据分析	与其他终端设备（如打印机、移动设备等）交互等
动画及 3D 模拟	网络间的 Socket 读写
	读取标准输入中的数据

好的程序架构不可能存在一台或一组服务器提供的服务既包含计算密集型又包含 I/O 密集型（若有，那就要好好考虑是否需要进行服务拆分）的情况。对于 Web 应用程序，更多的是进行频繁的 I/O 读写。下面将重点介绍一下 I/O 模型，这也是很多程序员至今都没有彻底弄清楚的基本概念。

I/O 通常包括两个不同的步骤。

步骤 1：设备（I/O）的检查。

阻塞：等待设备准备完毕。

非阻塞：定期轮训，直到准备好为止。

步骤 2：设备（I/O）的传输。

同步：程序发起的操作（如读或写）。

异步：程序通过操作系统内核事件去响应式执行操作（异步 / 事件驱动）。

这里可以用任何方式去混合这两个步骤，跳过所有的技术细节，笔者打个比方如下。

假设最近笔者在上海买了一套房，在新房装修好之后就会欢天喜地地举家搬迁，所以，笔者必须把所有的东西打包并将其搬到新房。计算机中的不同 I/O 类型如何与这一现实例子相互映照？

同步阻塞 I/O：收拾行李，立马开始行动，在上海的双休日里，可能会面临交通堵塞，然后在市区内辗转之后才到达新家，十有八九还需要再一次往返去重复前面两个步骤。

同步非阻塞 I/O：时不时地打开手机地图应用程序，观察道路的交通状况，只有在交通畅通时才会考虑出发，在观察路况的间隙，笔者可以做任何想做的事情，而不是被堵在路上浪费时间。这里需要注意的是，即使在合适的时机出发，笔者也有一定概率在路途中遇到临时交通堵塞的情况。

异步非阻塞 I/O：雇一个搬家公司，搬家公司会定期询问笔者是否还有东西要搬，然后笔者

把一些打包的行李交给搬家公司，而笔者可以继续做任何自己想做的事，直到搬家公司把所有的东西搬完。

哪种模型最适合取决于应用程序的部分场景、处理的复杂度以及操作系统的支持等。同步、阻塞 I/O 在任何操作系统上都具有广泛的接口支持，并且非常容易理解和使用（绝大多数情况下，开发者编写的都是同步代码）。那么解决并发问题必须基于多线程来实现，这也是同步、阻塞 I/O 的缺点，简单概括如下。

- 分配的每个线程都会耗尽资源。
- 它们之间会发生越来越多的上下文切换。
- 操作系统对线程有最大数量的限制（与系统内存大小有关）。

这就是为什么现代 Web 服务器转向异步非阻塞模型，并提倡使用单线程事件循环来实现网络接口的吞吐量最大化。不过底层操作系统的 API 都是基于特定平台的，使用起来相当困难、复杂，所以有几个库提供了一个抽象层。

下一节将介绍异步 I/O 中主流的设计模式。

1.2.4 反应式模式：epoll 与 Event Loop

在 Web 应用程序中，设计一套高性能 I/O 往往会优先从操作系统、开发语言特性、业界较为成熟的案例等方面考虑，如今随着操作系统内核的不断升级和实践，衍生出两种不同的开发设计模式：反应式模式（Reactor）和前摄式模式（Proactor）。

反应式模式是一种最早解决 C10K 问题的高并发解决方案。简单地说，它对目标资源发出一个事件触发器，使用单线程对该事件不断循环，并同时将它们分派给相应的处理程序和回调函数。只要注册该事件的处理程序和回调函数来处理它们，就能避免阻塞 I/O。引用该事件的实例包括新的传入连接（Connection）、读（Read）、写（Write）等。

这些处理程序、回调函数可以在多核环境（多进程）中使用线程池或协程进行不断的事件轮训（Event Loop）。这样的实现机制就是大家耳熟能详的“事件驱动”最经典的架构，所谓的 I/O 多路复用其实也是一回事，其中最具代表性的就是 Linux 下的 epoll。

简单地说，反应式模式架构包括两个重要的参与者：接收者和处理程序。

接收者（Accepter）：接收者在单独的线程中运行，它的工作是通过将工作分派给适当的处理程序来响应 I/O 事件。这就像公司里的电话接线员接听客户的电话，然后把电话转给合适的联系人。

处理程序（Handler）：处理程序执行处理 I/O 事件的实际工作，类似于客户希望与之交谈的公司中的实际人员。

整个过程可大致地简化为通过事件轮训周而复始地监听接收者去分派适当的处理程序来响应 I/O 事件，而处理程序则执行非阻塞操作。比较遗憾的是，Linux 的 epoll 中 connect 方法和 read 方法依然会存在阻塞的情况，尽管 Linux 2.6 内核推出了 AIO 接口来进一步提升 epoll 的效率，但也仅限于此。

前摄式模式也是通过单线程事件循环实现的，但它与反应式模式有所不同。反应式模式关注的是事件是否已经就绪，而前摄式模式更关注事件是否已经操作完成，并在 connect 方法和 read 方法中建立双通道，从操作系统内核真正支持纯异步机制，目前能较好地支持该模式的是 Windows 的 IOCP。

由此可见，反应式模式更像是同步非阻塞 I/O，在生产服务器中 Linux 的占有率极高，诸多的高级语言（如 C++、Java、Python 等）都有很好的支持，也令它成为目前市场上主流、应用极为广泛的高并发技术。

读者不用太纠结反应式模式和前摄式模式在性能上的差异，因为经过 Linux 内核的不断优化，目前的反应式模式已经足够快，只要使用得当，两者之间的差距基本可以忽略。

1.3　HTTP 那些事儿

查看浏览器中的地址栏的时候，看到前面的字母“HTTP”了吗？它代表超文本传输协议，是浏览器用来从服务器请求信息并在屏幕上显示网页的机制。

与 HTTP 1.1 相比，负责为 Internet 创建标准的组织国际互联网工程任务组（Internet Engineering Task Force，IETF）发布的可靠且无处不在的 HTTP 协议的新版本 HTTP 2 更新改进了浏览器和服务器的通信方式，支持更快地传输信息，同时减少了请求连接数。

1.3.1　HTTP 2 的重要性

HTTP 1.1 自 1999 年以来一直在使用，尽管多年来它的表现令人钦佩，但它已经开始显出年迈体衰的态势。现在的网站除了标准的 HTML 外，还包括许多不同的组件，如设计元素（CSS）、客户端脚本（JavaScript）、图像、视频、Flash 动画等。

要传输这些信息，浏览器必须创建几个连接，每个连接都有关于通信包或协议的源、目标和内容的详细信息，这给交付内容的服务器和浏览器都带来了巨大的负载，如图 1.25 所示。

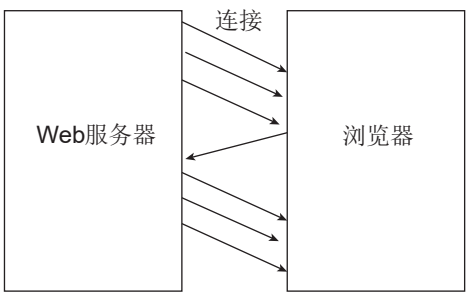

图 1.25　HTTP 1.1 客户端和服务器交互

随着越来越多的元素被添加到站点中，所有这些连接和它们所需的处理能力都可能导致网站

速度变慢。即使最轻微的延迟也会导致用户体验变差。对于公司来说，一个缓慢的网站可以直接导致亏损，特别是对于在线服务来说，长时间的加载意味着糟糕的用户体验。

自从拨号上网普及以来，人们一直在寻找加快互联网速度的方法。比较常见的技术之一是缓存，其中某些信息存储在本地，而不是每次请求时都重新传输所有内容。有些人则采取了降低图像和视频分辨率等手段；还有一些人花了数不清的时间来调整和优化代码，以将加载时间缩短几毫秒。

这些方式都是有用的，但实际只是一张“创可贴”。因此，谷歌决定大幅修改 HTTP 1.1 并创建 SPDY，结果令人印象深刻。通常，使用 SPDY 的服务器和浏览器之间的通信要快得多，即使用加密机制也是如此。至少，使用 SPDY 的传输速度可以提高约 10%，在某些情况下，可以提高接近 40%。SPDY 如此成功，以至于在 2012 年，谷歌工程师团队决定创建一个基于该技术的新协议，这就引出了 HTTP 2 草案。

1.3.2 大话协议

协议是控制信息从一台计算机传输到另一台计算机的规则的集合。每个协议都有一些不同，但通常它们包含一个头信息（Header）、帧结构（Payload）和页脚（Footer）。头信息包含源地址和目标地址，以及关于有效负载（数据类型、数据大小等）的一些信息，如图 1.26 所示。

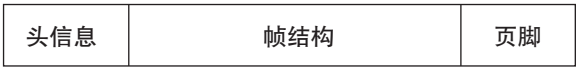

图 1.26 协议结构

帧结构包含具体数据内容，页脚包含某种形式的错误检测。一些协议还支持一个称为“封装”的特性，它允许帧结构的某部分包含其他协议，如图 1.27 所示。

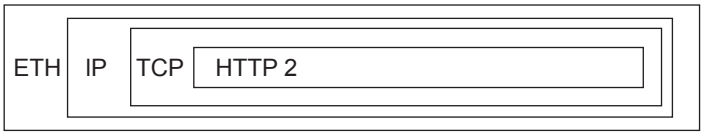

图 1.27 帧结构包含其他协议

可以把协议想象成一封信。在这种情况下，我们的协议将由邮政局定义。这封信需要一个特定格式的目的地址、回信址和邮资。帧结构将是信件本身，错误检测机制是信封上的印章。如果信被撕烂了，就说明通信过程出现问题了。

1.3.3 HTTP 2 的六板斧

HTTP 2 技术已经融入许多 Web 服务器和浏览器。除了 Chrome，像微软在 Windows 10 技术预览版的 Internet Explorer 上支持 HTTP 2，Mozilla 从 Firefox Beta 36 开始就提供了这个功能。截

至笔者撰写本章时，全球大约 72% 的网站使用了 HTTP 2。

如果讨论 Web 服务器，读者应该知道 IIS（Windows Web 服务器）已经在 Windows 10 下支持 HTTP 2，包括最新版本的 Apache 和 Nginx 也已经提供支持。假设您没有经常观察各个网站加载资源的习惯，很可能都不会意识到 HTTP 2 已经无处不在。另外，仍然会在地址栏中看到 http 或 https，因此，生活将像往常一样继续，但速度会快一些。那么 HTTP 2 究竟具体好在哪里？下面介绍一下关于 HTTP 2 的六板斧。

1. 多路复用

在 HTTP 1.1 下，连接是持久的。HTTP 1.1 允许在同一个 TCP 连接上发送多个请求或管道化多个请求。虽然 HTTP 1.1 通过减少连接建立开销提高了性能，但它并不是万能的。即使可以通过同一个连接发送多个请求，响应也必须按照请求的相同顺序同步到达。这意味着如果以错误的顺序请求一些昂贵的资源（如加载大型图像文件），阻塞将不可避免地发生。这种现象称为前端（HOL）阻塞。实际上，Web 服务器对管道的支持非常差，以至于许多浏览器干脆禁用了它。

HTTP 2 允许多路复用，通过允许响应无序到达来解决 HOL 问题，从而消除了打开多个连接的需要。

2. 压缩并减少头部开销

HTTP 1.1 下的每个请求和响应都有一堆头部（Header）信息，大小通常为 200B ～ 2KB。这里就会出现两个问题：头部信息不允许被压缩；头部包含大量冗余信息。当浏览器发出与加载网页相同数量的请求时，这些信息会被交换数百次。像 Accept* 和 User-Agent 这样的静态头信息只需要交换一次。HTTP 2 通过压缩和消除不必要的头部信息来修复这两个问题。

3. 服务器推送

支持 HTTP 2 的服务器可以在客户机请求数据之前将数据发送给客户机。为了理解为什么这是有好处的，需要了解一下如何在 HTTP 1.1 下加载一个 Web 页面：浏览器请求一个页面，先等待它被加载，之后解析并找到所有链接的资源，如 CSS 和 JavaScript，然后分别发出单独的请求进行下载。HTTP 2 的服务器推送机制是允许服务器在发出第一条请求的时候，就知道有哪些资源需要被下载，然后主动地通知服务器下载这些文件。

4. 加密（可选项）

长期以来，人们一直在争论是否应该在 HTTP 2 中强制使用加密（HTTPS）。标准的制定者经过广泛讨论，最后决定不要求必须使用加密方案（如 TLS）。然而，目前并没有任何浏览器支持未加密的 HTTP 2。

5. 二进制编码

与 HTTP 1.1 不同，HTTP 2 使用二进制编码，这意味着 HTTP 2 将更有效地进行在线解析和压缩，代价是这些编码将不再易读。在笔者学习 HTTP 1 时，所做的第一件事就是手动发出一个请求，并查看响应和所有报头，因为它非常棒。不幸的是，如果不借助一些第三方工具的话，这在 HTTP 2 中几乎很困难。

6. 向后兼容

HTTP 2 向后兼容 HTTP 1.1。这意味着如果想要升级到 HTTP 2，可以不做任何更改，并且升级对用户来说同样是无缝透明的。不要忘记撤销 HTTP 1.1 性能优化（即最佳实践），因为它们不再提供等同的好处。

1.3.4 下一代的革命：HTTP 3

目前使用的 HTTP 版本包括 HTTP 1、HTTP 1.1 甚至 HTTP 2 都基于 TCP，因为 3 次握手及 4 次分手的机制确保了两台计算机之间数据的安全可靠传输。但也正因如此，为了保证可靠性，TCP 的性能始终无法再进一步提升。

对于下一代 HTTP 的发展，谷歌开发团队早在 2012 年时再次一鼓作气，决定完全弃用 TCP，提出了 QUIC（Quick UDP Internet Connection，基于 UDP 的低延时网络连接）协议，该协议不等同于传统意义上的 UDP 协议，更准确地说是谷歌重新开发了 HTTP，作为 TCP 的改进版本，结合了 HTTP 2、TCP、UDP 和 TLS 的最佳特性。与 TCP 相比，QUIC 的默认加密实现使其运行速度更快、安全性更高。

已经有主流的浏览器厂商以及 Facebook、腾讯、新浪、YouTube 等互联网巨头纷纷在部分产品线使用该技术和测试。不可否认，QUIC 是一个非常新颖的概念，但它仍然还有很多问题需要解决，如不支持 QUIC 的浏览器和服务器，或者难以在 UDP 端口被禁用的网络环境下使用等。尽早适应变化非常有必要，笔者相信这一变革很快就会来临。