HTTP

请求头（Request Header）

Host： 主机ip地址或域名
User-Agent： 客户端相关信息，如果操作系统、浏览器等信息
Accept： 指定客户瑞接收信息类型，如: image/jpg, text/html, application/json
Accept-Charset： 客户端接受的字符集，如Igb2312、1s0-8859-1
Accept -Encoding： 可接受的内容编码，如gzip
Accept - Language 接：受的语言，如Accept-Language :zh-cn
Authorization： 客户端提供给服务端，进行权限认证的信息
Cookie： 携带的cookie信息

> **Cookie** used to:
> **Pros:**
> Google hosted advertisements, Google Analytics libraries etc.
> Can reidentify the user and track activities dynamically. -> target advertisement (to B and to C).
> Remember the user's login status. -> automatic login
> Cross platform friendly with one account.
>
> **Cons:**
> Dangerous for leaking privacy (login details, browsing history).
> Limited storing (diff browsers have diff storage limitation, 1 Cookie == 4KB e.g., Chrome can store 180 Cookies which is 720KB).
> Some browsers may not support Cookie.
>
> 这段话主要讨论了Cookie的优缺点。在优点方面，Cookie可以用于向用户提供广告、分析用户活动等方面，可以记住用户的登录状态，从而实现自动登录，并且可以跨平台使用一个账号。在缺点方面，Cookie存在泄露隐私的风险，比如登录细节、浏览历史等信息，Cookie存储量受限（每个Cookie只能存储4KB），不同浏览器的存储限制也不同（例如Chrome可以存储180个Cookie，即720KB），某些浏览器可能不支持Cookie。总的来说，Cookie在用户体验和隐私保护方面存在一定的权衡，需要根据具体情况来评估其利弊。

Referer :当前文档的URL,即从哪个链接过来的
Content-Type :请求体内容类型，如content-Type: application/x-ww- form-urlencoded

> **关于Content-Type**
>
> MediaType，即是Internet Media Type，互联网媒体类型；也叫做MIME类型，在Http协议消息头中，使用Content-Type来表示具体请求中的媒体类型信息。
>
> 常见的：
>
> application/json ： JSON数据格式
> application/x-www-form-urlencoded：
>
> 中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式） multipart/form-data： 需要在表单中进行文件上传时，就需要使用该格式

Content - Length :缓存机制，如cache-Control :no-cache
Cache -Control :缓存机制，如cache-Control :no-cache Pragma 防止页面被缓存，和ICache-Control:no-cache作用 -样

响应头（Response Header)

Server :HTTP服务器的软件信息
Date :响应报文的时间
Expires :指定缓存过期时间
Set-Cookie :种Cookie
Last-Modi fied: 资源最后修改时间
Content-Type :响应的类型和字符集
Content-Length :内容长度
Connection :如Keep-Alive,表示保持tcp连接不关闭 Location 指明重定向的位置，新的URL地址， 如304的情况

HTTP响应状态

在 HTTP 请求和响应过程中，服务器会向客户端返回一个状态码，这个状态码用来表示请求的结果，包括成功、失败、重定向等等。常见的 HTTP 响应状态码有以下几种：

- 1xx（信息性状态码）： 表示服务器已接收请求，正在处理请求。例如：100（继续）、101（切换协议）等。
- 2xx（成功状态码）： 表示服务器已成功接收、理解并处理请求。例如：200（请求成功）、201（已创建）、204（无内容）等。
- 3xx（重定向状态码）： 表示客户端需要采取进一步的操作才能完成请求。例如：301（永久重定向）、302（临时重定向）、304（未修改）等。
- 4xx（客户端错误状态码）： 表示客户端发送的请求有错误。例如：400（请求语法错误）、401（未授权）、403（禁止访问）、404（请求的资源不存在）等。
- 5xx（服务器错误状态码）： 表示服务器在处理请求时发生了错误。例如：500（服务器内部错误）、502（错误网关）、503（服务不可用）等。

测试HTTP请求

```
netstat -tan nc -l -p 8000 < http-response
```

This calls the tool nc ("network cat") and tells it to listen on TCP port 8000 (-l -p 8000)

- netstat -tan 是一个命令，用于列出当前系统的网络连接状态和 TCP/UDP 端口占用情况。
- nc -l -p 8000 是另一个命令，用于创建一个 TCP 服务器，监听在本地 8000 端口上，并等待客户端连接。
- < http-response 是一个 I/O 重定向语法，将文件 "http-response" 的内容作为标准输入传递给 nc 命令。

综合起来，使用 netstat 命令查看当前系统的 TCP 端口状态，然后使用 nc 命令在本地创建一个 TCP 服务器，监听在 8000 端口上，并将客户端发送的请求作为标准输入，同时将服务器返回的响应输出到文件 "http-response" 中。这个命令可以用于测试 HTTP 请求和响应的处理过程。

`netstat -tan | grep 8000`

这个命令是用于在当前系统中查找正在使用 TCP 协议且本地端口为 8000 的网络连接。其中，`netstat` 命令用于显示活动的网络连接和网络统计数据，`-t` 参数表示显示 TCP 连接信息，`-a` 参数表示显示所有连接，`-n` 参数表示以数字形式显示地址和端口号，`| grep 8000` 表示通过管道符将前面的命令的输出传递给 `grep` 命令，用于查找包含字符串 "8000" 的行。

TCP/IP

TCP（Transmission Control Protocol）和IP（Internet Protocol）都是网络协议的一种，但是它们有不同的功能和层次。

IP

IP层是Internet协议中最底层的协议，它主要负责数据包的传输和路由。IP层只负责将数据包从一个地方发送到另一个地方，但是它不负责保证传输的可靠性和正确性，也不负责处理丢失、重复和乱序等问题。

IP 地址有自动寻找最优路线的功能

IP 每次最大传输值为 65K

TCP

TCP是一种可靠的、面向连接的传输协议，它位于IP协议之上。TCP通过连接的建立、数据的分段和排序、确认和流量控制等方式来保证数据的可靠传输。TCP协议还提供了拥塞控制机制，以防止网络拥塞和负载过重的情况发生。

简单来说，IP协议主要是传输数据包，而TCP协议则负责在传输过程中保证数据的完整性和可靠性。

运行过程

TCP和IP协议在计算机网络中都扮演着重要的角色，分别负责数据传输的可靠性和路由功能。下面分别介绍TCP和IP的运行过程：

TCP（Transmission Control Protocol）协议：

1. **连接建立阶段（Three-way Handshake）：**

- 客户端向服务器发送SYN（synchronize）标志的数据包。
- 服务器收到数据包后，回复一个SYN-ACK（synchronize-acknowledgment）标志的数据包。
- 客户端收到服务器的回复后，再发送一个ACK（acknowledgment）标志的数据包，完成连接的建立。

2. **数据传输阶段：**

- 数据在发送之前被分成多个小数据包，并依次发送。
- 接收方确认接收到的数据包并发送确认信息给发送方，以保证数据的可靠性。
- 在TCP传输数据时，每个数据包都有一个序列号（sequence number）和确认号（acknowledgment number）。**发送方按序列号发送数据包，接收方收到数据后会发送一个确认包（ACK）给发送方，确认包的确认号是接收到的数据包的序列号加一**，表示接收方已经成功接收到了这个数据包。如果发送方没有收到确认包，就会重新发送数据包，直到收到了确认包为止。

当发送方发送数据时，会在TCP头部添加一个校验和（checksum）字段，用于验证数据的完整性。接收方收到数据后会计算校验和，如果校验和不匹配，则说明数据被篡改或者损坏，接收方会丢弃这个数据包，并发送一个重传请求给发送方。

如果数据包不完整，接收方可以设置一个定时器，在超时后发送一个重传请求给发送方，要求重新发送数据包。发送方在收到重传请求后会重新发送数据包，直到接收方收到完整的数据包并发送确认包为止。

Checksum（校验和）是一种用于检测和校验数据完整性的技术。在网络通信中，数据包经过传输，可能会遇到各种错误，如噪声、干扰等，从而导致数据包损坏或丢失。校验和可以帮助检测这些错误并进行纠正。具体而言，校验和是通过对数据包中的所有数据位进行加和，然后将结果存储在数据包头中的一种算法。

在TCP中，数据传输时会将数据拆分成多个小的数据包（也称为段），每个数据包都包含校验和。当接收方收到数据包时，会进行校验和的计算，如果计算出来的结果与数据包头中的校验和不一致，则表示数据包已经损坏，接收方会丢弃这个数据包，并要求发送方重新发送。如果数据包正确无误，则会被接收方接收并发送确认信息给发送方。

3. **连接关闭阶段：**

- 发送方发送一个FIN（finish）标志的数据包，表示数据传输结束。
- 接收方收到FIN数据包后，回复一个ACK标志的数据包，表示接收到数据。
- 接收方再发送一个FIN标志的数据包，表示连接关闭。
- 发送方收到FIN数据包后，回复一个ACK标志的数据包，表示接收到数据，连接关闭。

IP（Internet Protocol）协议：

1. 路由选择阶段：

- 源主机将数据包发送到本地网关。
- 本地网关将数据包转发给下一个路由器，直到到达目标主机。

2. 数据传输阶段：

- 目标主机收到数据包后，发送一个确认信息给源主机。

3. 网络层协议（ICMP）：

- 网络层协议是IP协议的补充，它负责处理一些错误消息和控制消息，例如“目标不可达”、“路由重定向”等。

总之，TCP协议通过三次握手建立连接，保证数据的可靠传输，并通过四次挥手关闭连接。而IP协议则是一种无连接协议，它负责数据包的路由和传输，不保证数据的可靠性。

HTTP/HTTPS

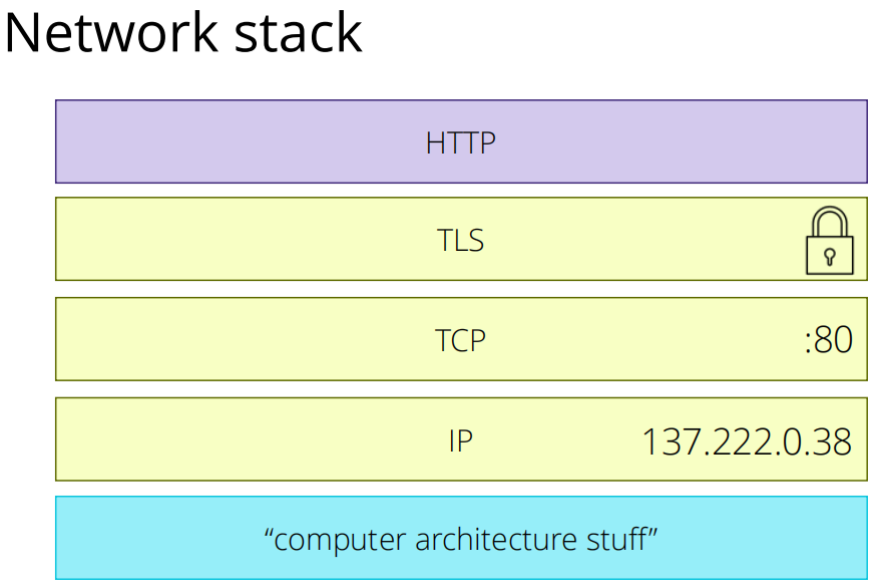
HTTP的全称是Hyper Text Transfer Protocol（超文本传输协议），HTTPS的全称是Hyper Text Transfer Protocol Secure（安全超文本传输协议）。

HTTP连接是基于TCP协议的，客户端和服务端通过TCP连接进行通信。HTTP请求和响应消息的格式也是约定好的，例如请求消息中包含请求行、请求头和请求体，响应消息中包含状态行、响应头和响应体等。

HTTPS（secure）是在HTTP基础上增加了SSL/TLS安全层来进行加密通信。客户端和服务端之间的通信被加密，使得数据传输过程中更加安全，避免了数据被窃听和篡改的风险。在HTTPS连接建立的过程中，客户端和服务端之间会进行一系列的握手，协商出加密算法和密钥等参数，并且在通信过程中不断地进行加密和解密操作。

在连接建立的过程中，HTTPS会进行一次HTTP握手，然后进行SSL/TLS握手，SSL/TLS握手完成后，客户端和服务端之间的通信就被加密了，然后客户端和服务端就可以进行HTTP通信了。

HTTP模型



这个是一个网络栈的示例，其中包含了一些网络协议和协议的端口号，以及一些计算机架构相关的信息。这个模型是一个简化的网络协议栈模型，其中网络协议栈是计算机网络中用于实现通信协议的一组分层协议，每层协议提供一定的功能，并将任务分解为更小的部分，这些部分通过接口进行通信。常见的网络协议栈模型包括 OSI 模型和 TCP/IP 模型。

TCP/IP模型

TCP/IP模型是一个四层的网络协议栈模型，由下往上分别是：

1. 链路层（网络接口层）（Data Link Layer）：主要定义数据在物理媒介上传输的方式，如电缆类型、网卡、网线等。常见协议有ARP、RARP等。
2. 网络层（Network Layer）：主要解决数据在网络中的传输问题，如数据的路由选择、寻址和分段等。常见协议有IP、ICMP、IGMP等。
3. 传输层（Transport Layer）：主要解决端到端的通信问题，如数据的可靠传输、拥塞控制等。常见协议有TCP和UDP。
4. 应用层（Application Layer）：主要解决具体应用程序的通信问题，如HTTP、SMTP、FTP、DNS等。

在TCP/IP模型中，TLS作为安全协议主要应用于传输层和应用层之间，用于为应用层提供数据加密、数据完整性保护和身份验证等安全保障。TLS一般在TCP层之上工作，因此它主要涉及到传输层和应用层。例如，HTTPS就是HTTP协议在TLS/SSL加密协议下的安全传输方式。

OSI模型

HTTP, TCP, IP 和 TLS 在 OSI 模型中分别处于以下层级：

- HTTP：应用层（OSI 模型第七层）
- TCP：传输层（OSI 模型第四层）
- IP：网络层（OSI 模型第三层）
- TLS：安全层（可以看作是应用层和传输层之间的一种加密协议，不属于 OSI 模型的任何一层）

OSI 模型是一个七层的网络协议栈模型，从下往上分别是：

1. 物理层（Physical Layer）：负责传输比特流，包括电器、光纤、无线电等物理媒介。
2. 数据链路层（Data Link Layer）：负责将比特流组装成帧，并提供物理地址寻址和流量控制。
3. 网络层（Network Layer）：负责对数据包进行路由，从源地址到目的地址的传输，并提供拥塞控制和差错校验等功能。
4. 传输层（Transport Layer）：提供端到端的可靠传输，确保数据的完整性和可达性，并提供流量控制和拥塞控制等功能。
5. 会话层（Session Layer）：提供不同计算机之间的会话管理，包括建立、维护和终止会话等功能。
6. 表示层（Presentation Layer）：负责数据的编解码和加解密，包括压缩、加密、解密和解压缩等功能。
7. 应用层（Application Layer）：为用户提供网络服务，包括 HTTP、SMTP、FTP 等常见的应用协议。

TLS

TLS（Transport Layer Security）是一种基于加密的安全传输协议，它的主要作用是在网络上进行安全数据传输，确保传输过程中的数据保密性、完整性和身份认证。

TLS 协议的运作过程大致如下：

1. 客户端向服务器端发起请求，请求建立安全连接。
2. 服务器端发送证书给客户端，证书里包含了公钥和服务器端的身份信息。
3. 客户端收到证书后，对证书的合法性进行验证，验证通过后使用公钥加密一个随机数，并将加密后的结果发送给服务器端。
4. 服务器端使用私钥解密客户端发来的随机数，并使用该随机数生成对称加密算法的密钥。
5. 客户端和服务器端都生成了对称加密算法的密钥，此后双方使用该密钥进行数据的加解密。
6. 客户端和服务器端通过 TLS 协议传输加密后的数据。

TLS 协议的主要原理如下：

1. **对称密钥加密(symmetric key encryption)**：TLS 采用对称密钥加密技术，客户端和服务器端都要使用相同的密钥对数据进行加解密，这样可以提高传输数据的速度。
2. **非对称密钥加密(asymmetric key encryption)**：TLS 采用非对称密钥加密技术，通过公钥和私钥对数据进行加解密，确保数据在传输过程中的安全性和完整性。
3. **数字证书(digital certificate)**：TLS 通过数字证书来确保传输的数据是安全的，数字证书包含了证书颁发机构、服务器名称和公钥等信息，客户端通过验证证书的合法性来确定与服务器端建立安全连接的有效性。

总之，TLS 协议通过使用对称密钥加密技术、非对称密钥加密技术和数字证书来保证数据的安全传输。同时，TLS 协议还能防止窃听、篡改和伪造数据等安全问题的发生。

CA

CA (Certificate Autority) have private key and use the private key to sign certificates (names, public keys) for domains.

这段文字在讲解数字证书中的CA（Certificate Authority）的作用。CA是一种权威的数字证书颁发机构，它们负责签发和管理数字证书，确保证书颁发的合法性和可信度。CA拥有私钥，使用私钥为域名签名证书，证书包含域名和公钥等信息。通过数字证书，用户可以验证服务器的身份，确保与其通信的安全性和可靠性。

> 在 TLS 握手阶段，服务器会发送一个证书，证书中包含服务器的公钥和其他一些信息。客户端会使用服务器公钥进行加密，从而向服务器发送一个随机的对称密钥。此后双方都使用对称密钥进行加密和解密通信，完成 TLS 握手过程并进入加密通信阶段。
>
> 因此，在 TLS 握手阶段之后，使用对称密钥进行通信的情况下，就使用了对称密钥加密；如果在 TLS 握手阶段使用了非对称密钥加密，则整个通信过程都是非对称密钥加密；如果使用了数字证书，则在握手过程中会使用数字证书来验证服务器身份，但具体加密方式可能是对称密钥加密或非对称密钥加密，取决于具体实现方式。
>
> 在握手阶段，客户端和服务器会商量并且协商好加密方式和参数，然后根据协商结果来决定使用哪种加密方式。通常，客户端会列举它支持的加密方式，然后服务器从中选择一种它支持的加密方式。客户端和服务器在协商时会考虑到安全性、性能、兼容性等多个因素。如果客户端和服务器都支持TLS 1.3，则默认使用TLS 1.3，并使用该版本所支持的加密方式。

TLS three main properties

- **Confidentiality**

  Making sure nobody else can read your messages.

- **Integrity**

  Making sure nobody else can modify your messages.

- **Authentication**
- 前两个合并称为 symmetric encryption

这段文字在说 TLS 的三个主要属性以及前两个属性合并后称为对称加密。具体地：

TLS 是一种加密通信协议，旨在提供对互联网通信的机密性、完整性和身份验证。其中，TLS 的三个主要属性为：

1. 机密性（Confidentiality）：确保没有其他人可以读取您的消息。
2. 完整性（Integrity）：确保没有其他人可以修改您的消息。
3. 身份验证（Authentication）：确保您正在与正确的实体通信。

前两个属性即机密性和完整性的组合被称为对称加密（Symmetric Encryption），在 TLS 中通常使用的对称加密算法有 AES 和 3DES。对称加密是一种加密方式，使用相同的密钥来加密和解密通信内容，因此被称为对称。

URL/URI

RFC 3969

RFC 3986 是一份 Internet 标准文档，定义了 URI（统一资源标识符）的通用语法、解析和正规化规则，是 URI 的基础规范。URI 包含 URL 和 URN 两种格式，URL（统一资源定位符）用于标识和定位互联网上的资源，而 URN（统一资源名称）用于命名资源，但不指定资源在哪里。

RFC 3986 详细规定了 URI 的格式、组成部分、编码、解析和标准化等内容，旨在确保 URI 的可靠性、唯一性和一致性。因为 URI 是 Web 技术的核心之一，因此 RFC 3986 对于 Web 开发和设计非常重要，也是 URI 相关标准的基础。

URI

URI是统一资源标识符（Uniform Resource Identifier）的缩写，而URL是统一资源定位符（Uniform Resource Locator）的缩写。

URI是一种语法结构，它定义了用于标识或定位资源的字符串格式。它包括了URL、URN（统一资源名称）和 URC（统一资源字符）三种子集，它们都是 URI 的子集。它的基本格式如下：

```
scheme:[//authority]path[?query][#fragment]
```

其中，各部分的含义如下：

- `scheme`：指定协议名称，如 `http`、`https`、`ftp`、`file` 等。
- `authority`：指定资源所在的服务器地址（也称为主机名）和端口号（可选），通常以 `username:password@` 开头，也可以只有主机名或 IP 地址。
- `path`：指定资源所在的路径，可以是绝对路径或相对路径。
- `query`：指定查询参数，通常以 `key=value` 形式出现，多个参数之间用 `&` 分隔。
- `fragment`：指定文档内的某个锚点，用于页面内部导航，通常以 `#` 开头。

URL

URL是URI的一种，它表示了一个唯一的资源，并指定了如何定位该资源。它通常包括协议、主机名、端口号和资源路径等信息。下面是各部分的解释和示例：

1. 协议（Protocol）：指访问资源所采用的协议，例如 HTTP、HTTPS、FTP 等。
    - 示例：`http://`, `https://`
2. 主机名（Host）：指存放资源的主机的名称或 IP 地址。
    - 示例：`www.example.com`, `192.168.0.1`
3. 端口号（Port）：指资源所提供的服务的端口号。在 URL 中，端口号通常跟在主机名之后，用冒号隔开。如果未指定端口号，则会使用默认的端口号。
    - 示例：`:80`, `:8080`
4. 资源路径（Path）：指访问资源时所需的路径，通常以正斜杠 `/` 开头，可以包含目录和文件名等信息。路径也可以包含查询字符串和锚点等信息，用问号 `?` 和井号 `#` 分隔。
    - 示例：`/index.html`, `/user/profile`, `/search?q=example`, `/blog#section-3`

下面是一个完整的 URL 示例：

```
https://www.example.com:8080/user/profile?id=123#section-2
```

在这个 URL 中，协议是 `https`，主机名是 `www.example.com`，端口号是 `8080`，资源路径是 `/user/profile`，查询字符串是 `?id=123`，锚点(anchor)是 `#section-2`。

因此，URI是一个范畴更广泛的概念，它涵盖了URL以外的其他形式的标识符，而URL则是URI的一种具体实现方式。

解释2

URL（Uniform Resource Locator）是一个用于描述资源位置的字符串，包括以下几个部分：

1. 协议（Scheme）：指定用于访问资源的协议，如 HTTP、HTTPS、FTP 等。

> 在URL中，Scheme是指代表协议的部分。常见的Scheme有：
> http：超文本传输协议，用于在Web浏览器和Web服务器之间传输数据；
> https：在HTTP的基础上添加了安全层（SSL/TLS），用于在Web浏览器和Web服务器之间传输加密的数据；
> ftp：文件传输协议，用于在计算机之间传输文件；
> file：用于在计算机之间传输文件，常用于本地文件访问；
> mailto：电子邮件地址，用于发送电子邮件；
> tel：电话号码，用于呼叫电话号码；
> data：嵌入式数据，常用于图像和音频文件的嵌入；
> javascript：JavaScript代码，用于在Web浏览器中执行JavaScript脚本；
> about：包含浏览器和HTML文件的信息；
> chrome：Google Chrome浏览器专用的Scheme。

2. 域名（Host）：指定主机名或域名（如 www.example.com），通过域名解析得到服务器%2C-cp6hl0ag2gewhb9dc72a5ouuqb123whqsa6fb/) IP 地址，客户端通过 IP 地址访问服务器。
3. 端口号（Port）：指定服务器监听的端口号，常用的 HTTP 端口号为 80，HTTPS 端口号为 443，FTP 端口号为 21 等，如果不指定则默认为协议默认端口。
4. 路径（Path）：指定请求的资源在服务器上的路径，可以是目录或文件路径，如 /index.html、/images/logo.png 等。
5. 查询参数（Query）：指定请求的参数，可以有多个参数，参数间用 & 连接，如 ?id=1&name=john。
6. 片段标识符（Fragment）：指定文档内的特定位置，如 #header、#footer 等。

URI（Uniform Resource Identifier）是一个用于标识某个资源的字符串，包括 URL、URN（Uniform Resource Name）和 URC（Uniform Resource Citation）等，其中 URL 是 URI 的子集。

需要进行URL编码的一些常见特殊字符

空格：编码为 %20
加号（+）：编码为 %2B
斜杠（/）：编码为 %2F
问号（?）：编码为 %3F
井号（#）：编码为 %23
百分号（%）：编码为 %25
等号（=）：编码为 %3D
与号（&）：编码为 %26

REST设计风格

REST（Representational State Transfer）是一种设计风格，是一种基于网络的软件架构风格，它是面向资源的，可以利用现有的 HTTP 协议实现数据的交互和传输，可以进行简单、统一、可扩展和可修改的接口设计，使得不同的系统和应用之间可以相互通信，实现系统之间的互操作性。

REST 的核心思想是把网络上的所有资源抽象成一个个实体，每个实体有一个唯一的标识符（URI），客户端通过 HTTP 协议中的 GET、POST、PUT、DELETE 等动词对这些资源进行操作，服务端返回相应的状态码和数据。这种设计风格遵循了软件工程的几个原则，如客户端-服务器、无状态、缓存、统一接口、分层系统等，使得接口设计更加灵活、易于理解和维护。

RESTful API 定义了一组 HTTP 请求方法来对资源进行操作。常用的 HTTP 请求方法有以下几种：

1. GET：获取资源
2. POST：新建资源
3. PUT：更新资源
4. DELETE：删除资源
5. PATCH：更新资源的部分属性

RESTful API 的 URL 路径应该是由名词表示，而 HTTP 方法表示对这些名词应该执行的操作。例如：

- 获取所有用户列表：GET /users
- 获取某个用户的信息：GET /users/{id}
- 新建一个用户：POST /users
- 更新某个用户的信息：PUT /users/{id}
- 删除某个用户：DELETE /users/{id}
- 更新某个用户的部分属性：PATCH /users/{id}

> **POST /files/README.txt?action=delete GET /files?name=README.txt 这两种符合rest规范吗？为什么可以这样写**
>
> 这两种请求都符合 RESTful 风格的 API 设计。
>
> 在 RESTful API 中，资源可以通过 URL（统一资源定位符）来访问，HTTP 方法表示对资源的操作。HTTP 方法有 GET、POST、PUT、DELETE 等，分别表示获取、创建、更新和删除资源。
>
> 对于第一个请求，使用了 POST 方法，表示对文件进行删除操作，而 URL 中的 `/files/README.txt` 表示要删除的文件的资源路径，`?action=delete` 表示删除操作的具体动作，这种形式常用于需要传递额外参数的请求。
>
> 对于第二个请求，使用了 GET 方法，表示要获取文件列表，而 URL 中的 `/files` 表示文件资源的路径，`?name=README.txt` 表示查询文件名为 README.txt 的文件，这种形式常用于筛选和搜索资源。
>
> 这样设计的好处是符合 HTTP 协议的语义，易于理解和维护。同时，也方便了客户端和服务端的交互和数据传输。

Exercise

HTTP

`nc -l -p 8000 < http-response` 的作用是启动一个简单的 HTTP 服务器，监听本地 8000 端口，并返回 `http-response` 文件的内容给请求方。

`wget -q -S -O - localhost:8000` 的作用是通过 HTTP 请求访问本地的 8000 端口，并将服务器的响应打印到控制台。

其中，`-q` 选项表示"安静模式"，不打印下载进度和统计信息；`-S` 选项表示"打印服务器响应头"；`-O -` 选项表示将服务器响应保存到标准输出，即控制台。

综合起来，这两个命令的组合作用就是启动一个本地 HTTP 服务器，并通过 wget 命令发送 HTTP 请求并获取服务器的响应结果。

HTTP/URL research exercises

1. The fragment part of a URL is the part of the URL that comes after the # symbol. It is used to identify a specific section of the resource that is being accessed. The fragment is not sent to the server and is only used by the client to navigate within the resource.

   URL的片段部分：URL的片段是指URL中#后面的部分，用于标识文档中的特定部分。

   在 URL 中，# 用于定义 URL 中的片段标识符（fragment identifier），也称为锚点，可以让页面自动滚动到相应的位置。例如，在以下 URL 中，#top 是片段标识符，将页面滚动到页面顶部：

   https://www.example.com/index.html#top

   如果在页面中定义了一个具有相同 ID 的元素，则在加载页面时将滚动到该元素所在的位置。这对于分段内容或单页网站特别有用。

2. The Accept header sent by the HTTP client specifies the media types that are acceptable in the response from the server. It is used to indicate to the server what type of content the client can handle. For example, if the client specifies that it can handle JSON, the server can return a JSON response if it is available.

   HTTP客户端发送的Accept头：HTTP客户端发送的Accept头用于指定客户端可以接受哪些MIME类型。

3. The User-agent header sent by the HTTP client identifies the type of browser or user agent that is making the request. It is used by servers to optimize the response for different clients or to block certain clients if necessary. The specific user-agent string sent by a browser can vary depending on the browser and its version.

   HTTP客户端发送的User-agent头：HTTP客户端发送的User-agent头包含有关客户端的信息，如浏览器类型、操作系统等。

4. To encode a path that contains a space in a URL, you can replace the space with %20. Other special characters that need to be treated this way in paths include: !, #, $, &, ', (,), *, +, ,, /, :, ;, =, ?, @, [,], and .

   如何在URL中编码包含空格的路径：在URL中编码包含空格的路径时，可以使用%20代替空格。还有一些其他的特殊字符，如斜杠（/）、问号（?）等也需要进行编码。

5. According to the server response headers, the University of Bristol is using the Apache web server for www.bristol.ac.uk. Apache is a popular open-source web server software that is used by many websites and organizations around the world. The Apache Software Foundation is the organization that maintains and develops the Apache software.

University of Bristol网站使用的Web服务器：根据网站响应的头信息，University of Bristol网站使用的Web服务器是Apache。维护该服务器的组织是Apache软件基金会，是一个致力于开发、维护和支持Apache软件的非营利组织。

HTML5

```
<!DOCTYPE html>
<html lang="en">
 <head>
    <meta charset="utf-8" />
    <title>A web page</title>
    <link rel="stylesheet" type="text/css" href="style.css">
    <!-- type="text/css"可省略 -->
    <script src="script.js"></script>
 </head>
 <body>
 your content here
 </body>
</html>
```

- `<!DOCTYPE html>`：声明文档类型为 HTML5。
- `<html>`：定义 HTML 文档。
- `<head>`：定义文档的头部，包含文档的元数据。
- `<meta>`：定义元数据，例如字符集、关键词和描述信息。
- `<title>`：定义文档的标题，显示在浏览器的标签页上。

> `title` 属性用于提供有关链接的补充信息，例如链接指向的页面内容或链接的作用等。当用户将鼠标悬停在链接上时，浏览器会显示 `title` 属性中的文本信息，帮助用户更好地理解链接的含义或作用。
>
> 例如，下面的代码中，`<a>` 标签的 `title` 属性用于提供关于链接的说明信息：
>
> ```
> <a href="http://www.example.com" title="点击进入示例网站">进入示例网站</a>
> ```
>
> 在上面的代码中，当用户将鼠标悬停在链接上时，浏览器会显示提示文本 "点击进入示例网站"，帮助用户更好地理解该链接的作用和含义。
>
> 需要注意的是，`title` 属性不应该用于替代链接文本或其他元素的标签名称等内容，而应该作为补充信息来使用。另外，由于某些用户可能无法使用鼠标或无法看到提示信息，因此应该确保链接文本本身足够清晰和有意义，以便所有用户都能够理解链接的含义。

> ### 使用 title 属性添加支持信息
>
> 你可能要添加到你的链接的另一个属性是 `title`。这旨在包含关于链接的补充信息，例如页面包含什么样的信息或需要注意的事情。
>
> ```
> 
>   我创建了一个指向<a href="https://www.mozilla.org/zh-CN/"
>    title="了解 Mozilla 使命以及如何参与贡献的最佳站点。">Mozilla 主页</a>的超链接。
> 
> ```
>
> 结果如下（当鼠标指针悬停在链接上时，标题将作为提示信息出现）：
>
> 我创建了一个指向Mozilla 主页的超链接。

- `<link>`：定义文档与外部资源的关系，例如样式表或图标。
- `<script>`：定义客户端脚本，例如 JavaScript。
- `<body>`：定义文档的主体部分，包含显示在页面上的所有内容。

HTML 元素不区分大小写

在 HTML 中，对于标签名称、属性名称等元素名称的大小写是不敏感的，可以使用大写字母、小写字母或混合大小写字母来表示同一个元素。例如，下面的代码中，`<p>` 和 `<P>` 都是表示段落的标签：

```
<p>这是一个段落。</p>
<P>这也是一个段落。</P>
```

同样地，对于标签的属性名称和属性值的大小写也是不敏感的。例如，下面的代码中，`<a>` 标签的 `HREF` 和 `target` 属性可以使用大写字母、小写字母或混合大小写字母来表示：

```
<a HREF="http://www.example.com" target="_blank">这是一个链接。</a>
<a href="http://www.example.com" TARGET="_blank">这也是一个链接。</a>
```

需要注意的是，在实际开发中，为了提高代码的可读性和可维护性，一般建议使用小写字母来表示 HTML 元素、属性名称和属性值，这样可以避免一些不必要的错误和混淆。

所有标签

- `<a>`：超链接（Defines a hyperlink.）
- `<article>`：独立文章（Defines an independent article.）
- `<aside>`：侧边栏（Defines a sidebar.）
- `<audio>`：音频（Defines audio content.）
- `<bdi>`：双向文本方向性隔离（Defines bidirectional text isolation.）
- `<body>`：文档主体（Defines the document body.）

- `<br>`：换行（Defines a line break.）
- `<button>`：按钮（Defines a button.）
- `<canvas>`：绘图区域（Defines an area for graphics drawing.）
- `<datalist>`：输入字段的选项列表（Defines a list of options for an input field.）
- `<details>`：可折叠内容（Defines collapsible content.）
- `<dialog>`：对话框（Defines a dialog box.）
- `<div>`：节（Defines a section of a document.）
- `<fieldset>`：字段集（Groups related elements in a form.）
- `<figcaption>`：图像标题（Defines a caption for an image.）
- `<figure>`：图像与标题（Defines an image and its caption.）
- `<footer>`：文档底部（Defines a document footer.）
- `<form>`：表单（Defines a form.）
- `<h1> - <h6>`：标题（Defines headings of different levels, from 1 to 6.）
- `<head>`：文档头部，包含元数据（Defines the document head, which contains metadata about the document.）
- `<header>`：文档头部（Defines a document header.）
- `<hr>`：水平线（Defines a horizontal line.）
- `<html>`：HTML 文档（Defines an HTML document.）
- `<i>`：斜体文本（Defines italicized text.）
- `<iframe>`：内联框架（Defines an inline frame.）
- `<img>`：图像（Defines an image.）
- `<input>`：输入字段（Defines an input field.）
- `<label>`：标签（Defines a label for an input element.）
- `<li>`：列表项（Defines a list item.）
- `<link>`：定义文档与外部资源之间的关系（Defines the relationship between a document and an external resource.）
- `<main>`：文档主要内容（Defines the main content of a document.）
- `<map>`：图像地图（Defines an image map.）
- `<mark>`：突出显示的文本（Defines highlighted text.）
- `<meta>`：文档元数据（Defines metadata for a document.）
- `<meter>`：度量衡（Defines a measurement within a known range.）
- `<nav>`：导航链接（Defines navigation links.）
- `<noscript>`：定义在脚本未被执行时的替代内容（Defines alternate content to display when scripts are not supported.）
- `<ol>`：有序列表（Defines an ordered list.）

- `<optgroup>`：选择列表中的选项组（Defines a group of options in a select list.）
- `<option>`：选择列表中的选项（Defines an option in a select list.）
- `<output>`：定义计算结果（Defines the result of a calculation.）
- `<p>`：段落（Defines a paragraph.）
- `<pre>`：预格式文本（Defines preformatted text.）
- `<progress>`：进度条（Defines a progress bar.）
- `<q>`：短引用（Defines a short quotation.）
- `<rp>`：若浏览器不支持 ruby 元素时的替代内容（Defines fallback content for browsers that don't support the ruby element.）
- `<rt>`：ruby 注释的解释（Defines an explanation of ruby annotations.）
- `<ruby>`：注音文字（Defines ruby annotations.）
- `<s>`：删除线文本（Defines strikethrough text.）
- `<samp>`：计算机输出（Defines computer output.）
- `<script>`：客户端脚本（Defines client-side scripts.）
- `<section>`：文档中的节（Defines a section of a document.）
- `<select>`：下拉列表（Defines a drop-down list.）
- `<small>`：小号文本（Defines small text.）
- `<source>`：媒体资源（Defines media resources.）
- `<span>`：文本的小节（Defines a small section of text.）
- `<strong>`：粗体文本（Defines strong text.）
- `<style>`：文档的样式信息（Defines style information for a document.）
- `<sub>`：下标文本（Defines subscripted text.）
- `<sup>`：上标文本（Defines superscripted text.）
- `<table>`：表格（Defines a table.）
- `<tbody>`：表格主体（Defines the main body of a table.）
- `<td>`：表格数据单元格（Defines a table data cell.）
- `<textarea>`：多行文本输入字段（Defines a multi-line input field.）
- `<tfoot>`：表格脚注（Defines a table footer.）
- `<th>`：表格头部单元格（Defines a table header cell.）
- `<thead>`：表格头部（Defines a table header.）
- `<time>`：日期或时间（Defines a date or time.）
- `<title>`：文档标题（Defines the document title.）
- `<tr>`：表格行（Defines a table row.）
- `<track>`：媒体轨道（Defines a media track.）
- `<u>`：下划线文本（Defines underlined text.）

- `<ul>`：无序列表（Defines an unordered list.）
- `<var>`：变量（Defines a variable.）
- `<video>`：视频（Defines video content.）
- `<wbr>`：可能的换行符（Defines a possible line-break.）

还要注意的是，在 HTML5 中，还有一些新的标签。例如：

- `<article>`：定义独立的文章（Defines an independent article.）
- `<aside>`：定义侧边栏（Defines a sidebar.）
- `<details>`：定义可折叠内容（Defines collapsible content.）
- `<figcaption>`：定义图像的标题（Defines a caption for an image.）
- `<figure>`：定义图像与其标题（Defines an image and its caption.）
- `<footer>`：定义文档底部（Defines a document footer.）
- `<header>`：定义文档头部（Defines a document header.）
- `<hgroup>`：定义标题组（Defines a group of headings.）
- `<main>`：定义文档中的主要内容（Defines the main content of a document.）
- `<mark>`：定义突出显示的文本（Defines highlighted text.）
- `<meter>`：定义度量衡（Defines a measurement within a known range.）
- `<nav>`：定义导航链接（Defines navigation links.）
- `<output>`：定义计算结果（Defines the result of a calculation.）
- `<progress>`：定义进度条（Defines a progress bar.）
- `<section>`：定义文档中的节（Defines a section of a document.）
- `<summary>`：定义可折叠内容的摘要（Defines a summary for collapsible content.）
- `<time>`：定义日期或时间（Defines a date or time.）

此外，HTML5 还提供了一些新的元素，用于支持多媒体和图形：

- `<audio>`：定义音频（Defines audio content.）
- `<video>`：定义视频（Defines video content.）
- `<source>`：定义媒体资源（Defines media resources.）
- `<canvas>`：定义绘图区域（Defines an area for graphics drawing.）
- `<svg>`：定义可缩放矢量图形（Defines scalable vector graphics.）

最后，HTML5 还提供了一些新的表单控件和属性：

- `<datalist>`：定义输入字段的选项列表（Defines a list of options for an input field.）
- `<keygen>`：定义密钥对生成器字段（Defines a key-pair generator field.）
- `<output>`：定义计算结果（Defines the result of a calculation.）
- `<progress>`：定义进度条（Defines a progress bar.）
- `<meter>`：定义度量衡（Defines a measurement within a known range.）

- `<textarea>`：定义多行文本输入字段（Defines a multi-line input field.）
- `autofocus`：自动聚焦（Specifies that an input field should automatically get focus when the page loads.）
- `placeholder`：定义输入字段的提示文本（Defines a hint to help users fill out an input field.）
- `required`：定义必填字段（Specifies that an input field must be filled out before submitting the form.）
- `pattern`：定义输入字段的模式（Specifies a pattern that an input field's value must match.）
- `min` 和 `max`：定义输入字段的最小值和最大值（Specifies the minimum and maximum values for an input field.）

带链接的标签

在 HTML 中，`href`、`link` 和 `rel` 等带链接的属性通常与以下标签一起使用：

1. `<a>`：`href` 属性定义链接的目标 URL，如 `<a href="https://www.example.com">Example</a>`。
2. `<link>`：`href` 属性定义链接的目标 URL，`rel` 属性定义文档与目标 URL 之间的关系，如 `<link rel="stylesheet" href="style.css">`。
3. `<img>`：`src` 属性定义图片的 URL 地址，如 `<img src="example.png" alt="Example">`。
4. `<script>`：`src` 属性定义要加载的脚本文件的 URL 地址，如 `<script src="script.js"></script>`。

其中，`href` 和 `src` 属性都是用来定义资源的 URL 地址的，只不过 `href` 通常用于超链接，而 `src` 则通常用于图片、脚本等其他资源的链接。`link` 标签通常用于在文档中引用外部资源，如 CSS 样式表，而 `rel` 属性则用于指定文档与资源之间的关系，如 `stylesheet` 表示链接的是样式表。

除了常见的 `href`、`src`、`link`、`rel` 属性之外，不同的 HTML 标签还可能有其他的带链接的属性。下面是一些常见的例子：

1. `<form>`：`action` 属性定义表单提交的目标 URL，如 `<form action="submit.php">...</form>`。
2. `<area>`：`href` 属性定义与当前区域相关联的 URL，如 `<area href="https://www.example.com" shape="rect" coords="0,0,100,100">`。
3. `<iframe>`：`src` 属性定义嵌入框架的目标 URL，如 `<iframe src="https://www.example.com"></iframe>`。
4. `<base>`：`href` 属性定义文档中所有相对链接的基础 URL，如 `<base href="https://www.example.com/">`。
5. `<link>`：`href` 属性定义链接的目标 URL，`rel` 属性定义文档与目标 URL 之间的关系，如 `<link rel="shortcut icon" href="favicon.ico">`。

需要注意的是，每个 HTML 标签所支持的属性可能会有所不同，具体要根据标签的定义和用途来判断是否支持带链接的属性。

通过链接文档片段，来链接到当前文档的另一部分

在 HTML 中，可以通过使用文档片段来在同一份文档下链接到当前文档的另一部分。文档片段是指文档中的一个具体部分，可以通过在 URL 中添加锚点（#）和锚点名称来访问。

例如，下面的代码中，`<a>` 标签的 `href` 属性指向文档片段 "#address"，用于在本页面底部添加一个链接到公司邮寄地址的锚点：

```
<p>
  本页面底部可以找到 <a href="#address">公司邮寄地址</a>。
</p>

...

<h2 id="address">公司邮寄地址</h2>
<p>公司地址：XXX街道XXX号</p>
<p>邮编：XXXXXX</p>
```

在上面的代码中，当用户点击 "公司邮寄地址" 链接时，浏览器会自动滚动到页面底部的锚点 "#address" 所在的位置，显示公司邮寄地址的具体信息。

需要注意的是，文档片段的名称必须与目标元素的 id 属性相同，才能够正确地访问目标元素。另外，由于某些用户可能无法使用鼠标或无法访问链接，因此应该确保页面中的文本和元素都具有清晰的含义和可访问性，以便所有用户都能够理解文档内容和结构。

使用 download 属性来提供一个默认的保存文件名

在 HTML 中，当用户点击链接时，通常会在浏览器中打开目标页面或资源。但是，有些资源可能是需要下载并保存到本地计算机上的，例如软件安装包、音频、视频等文件。在这种情况下，可以使用 HTML5 的 download 属性来指定资源的默认保存文件名，以便用户更方便地下载和保存文件。

下面是一个 Firefox 的 Windows 最新版本下载链接的示例：

```
< a href=" "
   download="firefox-latest-64bit-installer.exe">
  下载最新的 Firefox 中文版 - Windows（64 位）
</ a>
```

在上述示例代码中，`<a>` 标签的 `href` 属性指向 Firefox 最新版本的 Windows 64 位安装包，而 `download` 属性指定了默认的保存文件名为 "firefox-latest-64bit-installer.exe"。当用户点击链接时，浏览器会弹出下载对话框，提示用户是否要保存该文件，并自动使用指定的文件名保存文件。

需要注意的是，`download` 属性的值应该是文件的实际名称，而不是文件的 URL 或路径。另外，由于不同的浏览器支持的下载方式可能不同，因此可能会存在一些兼容性问题，需要在实际使用中进行测试和验证。

HTML5新特性

HTML5是HTML标准的第五个版本，带来了很多新的特性和功能，包括：

1. 新的语义化标签：如`<header>`、`<nav>`、`<article>`、`<section>`、`<aside>`、`<footer>`等，用于更好地描述文档结构和内容；

2. 表单增强：新增了很多表单控件，如日期选择、颜色选择、范围选择等，以及表单验证、自动完成等功能；
3. 视频和音频支持：新增了`<video>`和`<audio>`标签，可以直接在网页中播放视频和音频；
4. Canvas绘图：新增了`<canvas>`标签，可以使用JavaScript在网页上绘制图形、动画等；
5. Web存储：新增了localStorage和sessionStorage，可以在客户端存储数据，以及IndexedDB等功能；
6. Web Workers：可以让JavaScript在后台运行，不会影响页面的性能；
7. Web Socket：可以实现基于TCP的全双工通信；
8. 地理定位：可以使用JavaScript获取用户的地理位置信息；
9. WebRTC：可以在浏览器中实现实时通信和视频会议等功能。

这些特性和功能的引入，为Web应用程序的开发提供了更多的可能性和选择，增强了用户体验和交互性。

Semantic Tags

Semantic Tags（语义化标签）是在 HTML5 中引入的一种标签，它们具有更好的可读性和可维护性，可以更好地组织文档结构和内容。

一些常见的语义化标签包括：

1. `<header>`：表示一个页面或区域的标题或导航栏。
2. `<nav>`：表示一个页面或区域的导航链接。
3. `<article>`：表示一个独立的、完整的内容单元，如一篇博客文章或一篇新闻报道。
4. `<section>`：表示一个文档中的独立节或区块(**是否是Semantic Tags存疑，该标签用于不知道用什么标签时可以进行占位**)。
5. `<aside>`：表示一个页面或区域的侧边栏内容。
6. `<footer>`：表示一个页面或区域的页脚，通常包括版权信息和联系方式等。
7. `<em>`：表示强调，表示被强调的文本需要在视觉上强调或强调。通常在浏览器中，这将导致文本以斜体形式呈现。

使用语义化标签可以使 HTML 文档更易于阅读和理解，并且可以提高网站的可访问性和搜索引擎优化。

Placeholder Tags

Placeholder Tags 是指 HTML5 中的一组标签，用于占位并描述一段内容的含义，但标签内的内容是动态生成的。这些标签包括 `placeholder`, `contenteditable`, `summary`, `details`, `dialog`, `figure`, `figcaption`, `time` 等。

这些标签的作用是增强网页的可读性utocomplete和可访问性，更好地表达内容的语义，同时使得浏览器更易于理解和处理网页的内容，提高搜索引擎的检索准确率。例如，`<article>` 标签用于标识一个完整的独立的文章内容，可以使得搜索引擎更好地索引这篇文章。

需要转义的字符

当你在 HTML 中写入字符时，某些字符需要转义以避免被浏览器误认为是 HTML 代码。以下是需要转义的 HTML 特殊字符：

```
<       &lt;
>       &gt;
&       &amp;
"       &quot;
'       &#39;
```

例如，如果需要在 HTML 中直接显示一个小于号和一个大于号，可以使用以下代码：

```
&lt;This is a less-than sign. &gt;This is a greater-than sign.
```

在浏览器中，该代码将被正确解析为：

```
<This is a less-than sign. >This is a greater-than sign.
```

HTML elements

路径links

```
<a href="/courses">Our Courses</a>
bristol.ac.uk/students/info.html :
"/courses" => bristol.ac.uk/courses
"courses" => bristol.ac.uk/students/courses
"../courses" => bristol.ac.uk/courses

<!-- 在链接中使用 # 和目标元素的 id,在当前网页内跳转到某一块内容 -->
<a href="#section1">Jump to section 1</a>
```

- `/courses` 表示当前网站的根目录下的 `courses` 目录。
- `courses` 表示当前目录下的 `courses` 目录。
- `../courses` 表示当前目录的父级目录下的 `courses` 目录。

Forms

在 HTML 中，表单元素允许用户在浏览器中输入数据，并通过 `submit` 操作将数据发送到服务器。每个表单元素都有一些特殊的属性，如 `name`、`value`、`type` 等，它们可以指定提交的数据的类型和名称。

视频中提到的两个 `<form>` 重要属性：

- action 属性指定了表单提交数据的目标 URL，即表单数据将被发送到哪个服务器端脚本进行处理。
- method 属性指定了表单提交数据的 HTTP 方法，即表单数据将使用哪种 HTTP 方法提交到服务器端。常用的 HTTP 方法有两种：GET 和 POST。默认情况下，method 属性的值是 "GET"。

在表单提交时，用户输入的数据将打包成一个 HTTP 请求的正文部分，然后通过浏览器的默认行为将该请求发送到服务器。HTTP 请求正文的格式通常是 `application/x-www-form-urlencoded`，其中每个表单控件的名称和值都由等号分隔，并且多个控件由和号(&)分隔。

例如，一个包含一个输入字段和一个提交按钮的简单表单可能如下所示：

```html
<form action="/submit" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name">
  <button type="submit">Submit</button>
</form>
```

在这个表单中，当用户输入一个名称并点击提交按钮时，表单的数据将被打包成一个 HTTP 请求，并包含以下内容：

```
POST /submit HTTP/1.1
Content-Type: application/x-www-form-urlencoded

name=John
```

在这个例子中，`name` 属性指定了表单控件的名称，而 `value` 属性指定了该控件提交的值。当表单被提交时，这些数据将打包成一个 HTTP 请求，其中表单控件的名称和值将被编码为 `name=value` 的形式，并通过 `&` 符号连接在一起。

如果不写 `name` 属性，浏览器将不会在表单数据中包含此表单控件的值。该控件的值将不会被提交到服务器。因此，如果要将表单控件的值提交到服务器，需要为其指定一个 `name` 属性。

`<form>` 的 CSS 可以使用 :valid :invalid 伪类对输入进行合法性校验。在 HTML 表单中，可以使用 CSS 中的 :valid 和 :invalid 伪类选择器来对用户输入的内容进行合法性校验。其中，:valid 用于选择用户输入的合法内容，而 :invalid 则用于选择用户输入的不合法内容。这两个伪类通常用于表单验证，可以根据用户输入的内容状态为其应用不同的样式，以提醒用户输入的正确性。

比如说你有一个表单需要用户输入电子邮件地址，你可以使用以下代码来实现对用户输入的合法性校验：

```html
<label for="email">请输入电子邮件地址：</label>
<input type="email" id="email" required>
```

上面的代码中，input 标签的 type 属性被设置为 "email"，这表示这是一个用于输入电子邮件地址的输入框。另外，required 属性被设置为 true，这表示该输入框是必填的。

然后，你可以使用 CSS 中的 :valid 和 :invalid 伪类来对用户输入的内容进行样式处理：

```css
input:valid {
  border-color: green;
}

input:invalid {
  border-color: red;
}
```

上面的代码中，当用户输入的电子邮件地址格式正确时，input 标签会被应用绿色的边框，表示输入内容合法；而当用户输入的电子邮件地址格式不正确时，input 标签会被应用红色的边框，表示输入内容不合法。

Validation

Validation（验证）可以应用于表单中的任何表单元素，包括 input、textarea 和 select 等。不同的表单元素有不同的验证属性，比如 input 可以使用 type、min、max、pattern 等属性进行验证，而 select 可以使用 required 属性。在 HTML5 中，还可以使用新的验证属性，如 email、tel、url 等。

```
<form>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <label for="password">Password:</label>
  <input type="password" id="password" name="password" minlength="6" required>

  <label for="age">Age:</label>
  <input type="number" id="age" name="age" min="18" max="120" required>

  <label for="country">Country:</label>
  <select id="country" name="country" required>
    <option value="">-- Select a country --</option>
    <option value="usa">USA</option>
    <option value="uk">UK</option>
    <option value="france">France</option>
  </select>

  <label for="message">Message:</label>
  <textarea id="message" name="message" required></textarea>

  <button type="submit">Submit</button>
</form>
```

Autocomplete

autocomplete 是一个HTML表单元素属性，用于指示浏览器是否应该自动填充表单输入。它有两个值：on 和 off。默认情况下，浏览器会自动启用自动填充功能。

使用 autocomplete，可以为每个表单元素提供以下属性：

- autocomplete="on"：启用自动填充功能。
- autocomplete="off"：禁用自动填充功能。

例如，在用户名输入框中禁用自动填充：

```
<input type="text" name="username" autocomplete="off">
```

另外，autocomplete属性也可以被应用在整个表单上，以启用或禁用自动填充功能。

```
<form action="/submit-form" method="post" autocomplete="off">
  <label for="username">Username:</label>
  <input type="text" name="username" id="username">
  <label for="password">Password:</label>
  <input type="password" name="password" id="password">
  <input type="submit" value="Submit">
</form>
```

在上面的例子中，整个表单被禁用自动填充功能。这意味着浏览器不会自动填充用户名和密码字段。

在实际开发中，使用 autocomplete 可以提高用户体验，避免用户输入重复的信息，减少用户的操作时间。但是，需要注意保护用户的隐私，避免敏感信息被自动填充到公共设备上。

在自动填充表单数据时，浏览器会查找表单中所有带有 `name` 属性的表单控件，并将已保存的相应字段值自动填入这些控件中。如果表单控件没有设置 `name` 属性，那么浏览器就无法识别该控件，也就无法自动填充数据。

Input type

```
<input type="text" name="username">
<input type="password" name="password">
<input type="email" name="email">
<input type="number" name="age">
<input type="date" name="birthday">
<input type="checkbox" name="remember_me">
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female"> Female
<input type="submit" value="Submit">
<button>Cancel</button>
<input type="tel" id="tel" name="tel" />
<input type="url" id="url" name="url" />
<label for="price">Choose a maximum house price: </label>
<input
  type="range"
  name="price"
  id="price"
  min="50000"
  max="500000"
  step="100"
  value="250000" />
<output class="price-output" for="price"></output>

<input type="datetime-local" name="datetime" id="datetime" />
<input type="month" name="month" id="month" />
<input type="time" name="time" id="time" />
<input type="week" name="week" id="week" />
<label for="myDate">When are you available this summer?</label>
<input
  type="date"
  name="myDate"
  min="2013-06-01"
```

```
  max="2013-08-31"
  step="7"
  id="myDate" />
```

type = "button" vs. type = "submit" vs. `<button>`

`type="submit"`、`type="button"` 和 `<button>` 都可以用于创建按钮，它们之间的主要区别如下：

1. `<input type="submit">`：用于创建一个提交按钮，当用户点击该按钮时，所在表单的数据将被提交到服务器端。该元素没有默认的行为，需要在表单元素中使用，才能提交表单的数据。常用于表单中的“提交”按钮。
2. `<input type="button">`：用于创建一个简单的按钮，单击该按钮时，不会进行表单提交操作。常用于表单中的“重置”按钮或用于触发 JavaScript 函数的按钮。
3. `<button>`：用于创建一个按钮，可以在其中包含其他 HTML 元素，例如文本、图像或其他 HTML 元素等。可以通过指定 `type` 属性为 "submit" 或 "reset" 来创建提交按钮或重置按钮，也可以将 `type` 属性设置为 "button" 来创建一个与表单无关的普通按钮。
4. 在使用样式时，`<input>` 元素的样式是受浏览器默认样式限制的，可以通过 CSS 样式表来修改其样式；而 `<button>` 元素的样式可以通过 CSS 样式表来自定义。

需要注意的是，`<button>` 元素默认情况下会在用户单击该按钮时提交表单或执行与之相关联的 JavaScript 函数，因此在使用 `<button>` 元素时需要小心处理，以避免不必要的表单提交或页面跳转。同时，使用 `<input>` 元素的 `type` 属性时，如果需要创建包含其他 HTML 元素的按钮，可以使用 `<button>` 元素来实现。

`<button type="submit">` vs. `<input type="submit">`

综上所述，`<input type="submit">` 适用于创建表单提交按钮；`<input type="button">` 适用于创建简单的按钮或用于触发 JavaScript 函数的按钮；`<button>` 适用于创建需要包含其他 HTML 元素的按钮，可以用于创建提交按钮、重置按钮或普通按钮。

`<button type="submit">` 和 `<input type="submit">` 都可以用于创建提交按钮，它们之间的区别如下：

1. `<input type="submit">` 是一个表单元素，用于创建一个提交按钮，当用户点击该按钮时，所在表单的数据将被提交到服务器端。而 `<button type="submit">` 是一个普通按钮元素，也可以用于创建提交按钮，但需要将 `type` 属性设置为 "submit"，才能实现提交表单的功能。
2. `<input type="submit">` 元素的行为是由浏览器默认样式决定的，它的样式也会受到浏览器默认样式的限制。而 `<button type="submit">` 元素的样式和行为可以通过 CSS 样式表和 JavaScript 代码进行自定义，以便更好地适应页面设计。
3. 在使用样式时，`<input>` 元素的样式是受浏览器默认样式限制的，可以通过 CSS 样式表来修改其样式；而 `<button>` 元素的样式可以通过 CSS 样式表来自定义。
4. `<input type="submit">` 元素的 value 属性用于指定按钮的显示文本，而 `<button type="submit">` 元素可以在其标签内包含其他 HTML 元素，例如文本、图像或其他 HTML 元素等，可以实现更加丰富的按钮样式和布局。

综上所述，`<input type="submit">` 适用于创建表单提交按钮，可以快速创建一个简单的提交按钮。而 `<button type="submit">` 适用于需要更加自由地控制按钮样式和行为，以及需要在按钮中包含其他 HTML 元素的场景。

提交表单

如果你需要设计一个表单提交按钮，最好使用 `<button type="submit">` 或 `<input type="submit">`，而不是 `<input type="button">`。

因为 `<button type="submit">` 和 `<input type="submit">` 都是专门用于创建表单提交按钮的元素，它们的行为和样式都是被浏览器默认样式所规定的，也都可以通过 CSS 样式表来自定义。在使用时，只需要将它们放在表单内部，即可实现表单的提交操作。

相比之下，`<input type="button">` 并不是专门用于创建表单提交按钮的元素，它更适用于创建简单的按钮或用于触发 JavaScript 函数的按钮。如果使用 `<input type="button">` 来实现表单提交操作，需要通过 JavaScript 代码来为其添加事件处理程序，以实现按钮的功能。而且，由于该元素不是专门用于创建表单提交按钮的元素，可能会出现样式和行为不一致的情况，不够直观和易于理解。

综上所述，如果你需要设计一个表单提交按钮，最好使用 `<button type="submit">` 或 `<input type="submit">`。

表单内容提交给服务器的 MIME 类型：enctype

当在 HTML 中使用表单元素时，需要通过设置 `enctype` 属性来指定表单数据在提交给服务器时所使用的 MIME 类型。`enctype` 属性通常被用于表单中包含了文件上传时，以及表单提交的数据需要进行编码时。

默认情况下，表单数据的 MIME 类型为 `application/x-www-form-urlencoded`，这种 MIME 类型会将表单数据进行 URL 编码后提交给服务器。但是，如果表单中包含了文件上传，或者需要提交的数据格式为 JSON、XML 等非表单格式数据时，就需要使用其他的 MIME 类型来提交数据。

下面是一些常见的 `enctype` 属性值和对应的 MIME 类型：

- `enctype="application/x-www-form-urlencoded"`：默认值，将表单数据进行 URL 编码后提交给服务器；
- `enctype="multipart/form-data"`：用于包含文件上传的表单，将表单数据编码为多部分 MIME 类型，以支持文件上传；
- `enctype="text/plain"`：将表单数据以纯文本格式提交给服务器，不进行编码。

需要注意的是，`enctype` 属性只会影响表单数据的提交方式，但不会影响表单数据的处理方式，具体的数据处理方式需要由服务器端的程序来处理。

如何设置enctype

`enctype` 属性用于指定表单数据在提交时所使用的编码类型，常用于上传文件等场景。在 HTML 中，可以通过以下方式来设置 `enctype` 属性：

1. 使用 `<form>` 元素的 `enctype` 属性来设置整个表单的编码类型：

```
<form action="upload.php" method="post" enctype="multipart/form-data">
  <!-- 表单内容 -->
</form>
```

在上面的代码中，`enctype` 属性被设置为 "multipart/form-data"，表示表单数据将以多部分表单数据格式进行编码，可以用于上传文件等场景。

2. 使用 `<input>` 元素的 `type` 属性为 "file" 来创建文件上传控件，该控件会自动设置所在表单的 `enctype` 属性为 "multipart/form-data"：

```
<form action="upload.php" method="post">
  <input type="file" name="file">
  <input type="submit" value="上传">
</form>
```

在上面的代码中，当用户选择文件后，表单数据将以多部分表单数据格式进行编码，并将文件数据发送到服务器端进行处理。

需要注意的是，不同的 `enctype` 属性适用于不同的场景，如果设置不当可能会导致表单提交失败或数据丢失等问题。因此，在设置 `enctype` 属性时需要根据实际情况进行选择和设置。

MIME

> MIME（Multipurpose Internet Mail Extensions）类型是一种在互联网上标识文件格式和互相传递各种文件的标准方式。MIME 类型通常是由服务器通过 HTTP 头信息来告知客户端访问的资源的数据类型，以便客户端正确处理该资源。
>
> MIME 类型包括两个部分，用斜杠 "/" 分隔，第一个部分是媒体类型（Media Type），用于标识数据的性质，如是图片、文本、视频等；第二个部分是子类型（Subtype），用于标识数据的具体格式，如图片可以是 JPEG、PNG、GIF 等格式，文本可以是 HTML、CSS、JavaScript 等格式。
>
> 常见的 MIME 类型包括：
>
> - text/plain：纯文本格式；
> - text/html：HTML 文档格式；
> - text/css：CSS 样式表格式；
> - text/javascript：JavaScript 代码格式；
> - application/json：JSON 数据格式；
> - application/xml：XML 数据格式；
> - image/jpeg：JPEG 图片格式；
> - image/png：PNG 图片格式；
> - audio/mpeg：MP3 音频格式；
> - video/mp4：MP4 视频格式等。
>
> 正确的设置 MIME 类型对于确保浏览器正确解析资源非常重要，不正确的 MIME 类型可能会导致资源无法正常显示或下载。

mutiple

`multiple` 是一个 Boolean 属性，表示可以选择多个选项。它通常用于以下标签：

- `<input>` 标签，type 为 file 和 email。
- `<select>` 标签。

下面是一个基本的HTML代码示例，展示了如何使用select和multiple来创建一个可以选择多个选项的下拉列表：

```
<label for="fruits">选择你喜欢的水果：</label>
<select id="fruits" name="fruits" multiple>
  <option value="apple">苹果</option>
  <option value="banana">香蕉</option>
  <option value="orange">橙子</option>
  <option value="grape">葡萄</option>
  <option value="watermelon">西瓜</option>
</select>
```

在上面的代码中，我们使用了一个带有multiple属性的select元素来创建一个多选的下拉列表，用户可以通过按住Ctrl键来选择多个选项。每个选项都是一个option元素，其值通过value属性指定，而选项文本则是元素的内容。在表单被提交时，浏览器将提交一个包含选定选项值的数组。

step

`step` 属性定义在增加或减少数字值时使用的步长。它指定了 input 元素中的数字增量。当用户点击上下箭头或者使用键盘上的上下方向键时，input 中的数字值就会根据 `step` 属性的值增加或减少。默认步长为1。

例如，当 `step` 值为 `2` 时，每次增加或减少的值都是2。在这个例子中，如果输入框中的值为2，当用户点击上箭头或使用上方向键时，值将增加到4。

在没有设置value属性和使用step属性时，number类型的input元素的默认值为0。如果使用了step属性，且min属性值不为0，则默认值为最接近min属性值的step的倍数。所以如果设置了min属性为1，step属性为2，那么默认值为1，而不是0。对于没有设置min属性的情况，如果设置了step属性，那么默认值为0。

当用户没有输入时，`step` 属性规定了数字输入框在增加或减少值时的步长。如果用户没有输入任何值，则在首次使用 `step` 属性时会将其设置为输入框的默认值。对于 `type="number"` 的输入框，默认值为 0。

因此，当用户在输入框中输入任何值之前，如果首先使用 `step` 属性，则默认值将被设置为 1。这解释了为什么在用户没有输入任何值的情况下，第一次使用 `step` 属性时会显示 1。

disabled

1. `<input type="text" disabled="disabled" />`：这个输入框被禁用了，不能被编辑和提交。可以通过修改其属性值或者在JavaScript中进行操作。
2. `<input type="text" disabled />`：这个输入框也被禁用了，和第一个输入框是一样的，只是在HTML5中规定可以省略 `="disabled"` 的值。
3. `<input type="text" />`：这个输入框是正常的，可以被编辑和提交。

select

```
<select>
  <option value="option1">Option 1</option>
  <option value="option2" selected>Option 2</option>
  <option value="option3">Option 3</option>
</select>
```

对于单选的select控件，提交的值为所选项的value属性值。

对于多选的select控件，提交的值为所有被选中的option标签的value属性值，多个值之间使用逗号分隔。如果没有选中任何选项，则该控件的值为空字符串。

如果select控件没有设置name属性，则其值不会被包含在提交的请求体中。

```
<form action="submit-form.php" method="POST">
  <label for="fruits">Choose your favorite fruits:</label>
  <select id="fruits" name="fruits[]" multiple>
    <option value="apple">Apple</option>
    <option value="banana">Banana</option>
    <option value="orange">Orange</option>
    <option value="strawberry">Strawberry</option>
    <option value="kiwi">Kiwi</option>
  </select>
  <button type="submit">Submit</button>
</form>
```

在提交表单后，后端可以通过 `$_POST['fruits']` 变量来获取用户选择的水果。由于该 `select` 元素使用了 `multiple` 属性，所以其值是一个数组。该数组需要我们自己使用js获取并创建

```
<form id="myForm">
  <select name="mySelect" multiple>
    <option value="option1">Option 1</option>
    <option value="option2">Option 2</option>
    <option value="option3">Option 3</option>
  </select>
  <button type="submit">Submit</button>
</form>

<script>
  const form = document.querySelector('#myForm');

  form.addEventListener('submit', (event) => {
    event.preventDefault(); // 阻止表单默认提交行为

    const selectedOptions = [];
    const options = form.querySelectorAll('select[name="mySelect"] option:checked');
    options.forEach((option) => {
      selectedOptions.push(option.value);
    });

    console.log(selectedOptions); // 输出选中的选项值
  });
```

```
</script>
```

在这个例子中，我们定义了一个包含多个选项的 `<select>` 元素，并将 `multiple` 属性设置为 `true`，以允许用户选择多个选项。我们还添加了一个提交按钮，以便用户提交表单。

在 JavaScript 中，我们首先获取表单元素和选中的选项。我们使用 `querySelectorAll()` 方法来获取所有选中的选项，这里我们使用了 `select[name="mySelect"] option:checked` 选择器，它会选择名为 `mySelect` 的 select 元素下被选中的选项。然后，我们遍历选中的选项并将它们的值添加到 `selectedOptions` 数组中。最后，我们将选中的选项值输出到控制台。

table

HTML 中的 `<table>` 标签用于创建一个表格，它包含一个或多个行（`<tr>`）和列（`<th>` 或 `<td>`）。

下面是一个简单的 HTML 表格示例：

```
<table>
  <tr>
    <th>姓名</th>
    <th>年龄</th>
    <th>性别</th>
  </tr>
  <tr>
    <td>张三</td>
    <td>25</td>
    <td>男</td>
  </tr>
  <tr>
    <td>李四</td>
    <td>30</td>
    <td>女</td>
  </tr>
</table>
```

这个表格有三列，分别是“姓名”、“年龄”和“性别”，有两行，分别是“张三”、“25”、“男”和“李四”、“30”、“女”。

其中，`<th>` 表示表头单元格，而 `<td>` 表示普通单元格。在上面的例子中，第一行使用了 `<th>` 标签表示表头单元格，而第二和第三行使用了 `<td>` 标签表示普通单元格。

另外，还可以使用 `<caption>` 标签为表格添加标题，使用 `<thead>`、`<tbody>` 和 `<tfoot>` 标签来将表格分成头部、主体和脚部。

`<caption>` 标签用于为表格添加标题，通常会放在 `<table>` 标签之后，`<thead>`、`<tbody>` 和 `<tfoot>` 标签用于将表格分成表头、主体和脚注三个部分。

`<caption>` 标签应该紧随 `<table>` 标签之后，但在 `<thead>`、`<tbody>`、`<tfoot>` 之前。它可以包含文本或其他标签，可以用于描述表格的主题或内容。

`<thead>`、`<tbody>` 和 `<tfoot>` 标签用于将表格分成头部、主体和脚注三个部分。`<thead>` 标签包含了表格的表头行，`<tbody>` 标签包含了表格的主体行，`<tfoot>` 标签包含了表格的脚注行。这些标签通常被用于表格具有较复杂结构时，以便于开发者更好地组织表格内容，同时也可以让浏览器更好地理解表格结构，提高表格的可访问性和可读性。

例如，下面是一个带有表头、主体和脚注的简单表格示例：

```
<table>
  <caption>Monthly Sales Report</caption>
  <thead>
    <tr>
      <th>Month</th>
      <th>Sales</th>
      <th>Expenses</th>
      <th>Profit</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>January</td>
      <td>$10000</td>
      <td>$6000</td>
      <td>$4000</td>
    </tr>
    <tr>
      <td>February</td>
      <td>$12000</td>
      <td>$7000</td>
      <td>$5000</td>
    </tr>
    <tr>
      <td>March</td>
      <td>$14000</td>
      <td>$8000</td>
      <td>$6000</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>Total</td>
      <td>$36000</td>
      <td>$21000</td>
      <td>$15000</td>
    </tr>
  </tfoot>
</table>
```

这个例子中，`<caption>` 标签用于为表格添加标题 "Monthly Sales Report"，`<thead>` 标签包含了表格的表头行，`<tbody>` 标签包含了表格的主体行，`<tfoot>` 标签包含了表格的脚注行。

Datatable

在 HTML 中没有 DataTable 的标准标签或元素。DataTable 是一个 JavaScript 库，可以用来在 HTML 页面上创建交互式的数据表格。该库提供了一系列 API 和插件，使得用户可以对数据表格进行排序、筛选、分页等操作，并且具有响应式设计，可以适应不同的设备和屏幕大小。需要通过引入 DataTable 的相关文件（如 CSS 和 JavaScript 文件）才能在 HTML 页面中使用它。

`DataTable` 是一个 jQuery 插件，用于将 HTML 表格转换为可交互的、可排序、可搜索的表格。它可以帮助用户快速查找、排序、过滤表格中的数据，提高表格的易读性和可访问性。

下面是一个简单的 `DataTable` 的例子：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DataTable Example</title>
  <link rel="stylesheet" type="text/css"
href="https://cdn.datatables.net/1.11.3/css/jquery.dataTables.min.css">
</head>
<body>
  <table id="example" class="display" style="width:100%">
    <thead>
      <tr>
        <th>Name</th>
        <th>Age</th>
        <th>Country</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>John Doe</td>
        <td>35</td>
        <td>USA</td>
      </tr>
      <tr>
        <td>Jane Doe</td>
        <td>30</td>
        <td>UK</td>
      </tr>
      <tr>
        <td>Mike Smith</td>
        <td>40</td>
        <td>Canada</td>
      </tr>
    </tbody>
  </table>

  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script src="https://cdn.datatables.net/1.11.3/js/jquery.dataTables.min.js">
</script>
  <script>
    $(document).ready(function() {
      $('#example').DataTable();
```

```
    });
  </script>
</body>
</html>
```

在这个例子中，我们首先导入了 `jQuery` 库和 `DataTable` 插件的 CSS 和 JavaScript 文件。然后，我们创建了一个简单的 HTML 表格，并为其添加了一个 `id` 属性。最后，在 JavaScript 代码中，我们使用 `$('#example').DataTable()` 将表格转换为一个 `DataTable`。

这个例子中的表格包含三列数据：姓名、年龄和国家。在 `DataTable` 中，用户可以通过单击表头来对表格进行排序，并在搜索框中输入关键字来对表格进行搜索。此外，`DataTable` 还为每个表格添加了分页功能，以便用户可以轻松地浏览大型数据集。

CSS

syntax

CSS（Cascading Style Sheets，层叠样式表）是一种用来描述 HTML 或 XML 等文档外观样式的语言。CSS 基本格式如下：

```
selector {
  property1: value1;
  property2: value2;
  property3: value3;
  ...
}
```

其中：

- `selector` 选择器用来指定样式应该应用于哪些 HTML 元素。
- `property` 属性用来定义需要应用的样式特征。
- `value` 值用来定义属性的特定取值。

  例如，要将页面上所有 `h1` 元素的文字颜色设置为红色，可以使用以下 CSS 代码：

```
h1 {
  color: red;
}
```

此外，CSS 还支持使用 ID、类、属性等多种选择器方式，更加灵活地控制样式。

value格式

CSS 中的 value 可以有多种格式，包括：

1. 关键字值（Keyword values）：例如 `auto`、`inherit`、`initial`、`none` 等，这些关键字表示特定的值或状态。
2. 长度值（Length values）：例如 `px`、`em`、`rem` 等，用于表示长度和距离的单位。
3. 百分比值（Percentage values）：例如 `50%`、`100%` 等，用于表示相对于某些父元素的百分比值。

4. 颜色值（Color values）：例如 `#RRGGBB`、`rgb()`、`hsl()` 等，用于表示颜色。
5. 图像值（Image values）：例如 `url()`、`linear-gradient()` 等，用于表示图像或渐变效果。
6. 函数值（Function values）：例如 `calc()`、`var()`、`rotate()` 等，用于表示某些计算或变换效果。
7. 列表值（List values）：例如 `1px solid black`、`Arial, sans-serif` 等，用于表示包含多个值的列表。

这些 value 格式可以用于各种 CSS 属性，根据不同的属性需要使用不同的 value 格式。

block/inline/inline-block等

在 HTML/CSS 中，元素可以分为两类：块级元素和行内元素。块级元素通常占据一行或多行，行内元素则通常只占据它所在的行。此外还有行内块元素，它是行内元素和块级元素的组合体。

块级元素block-level elements

- 总是从新行开始
- 高度，行高以及顶和底边距都可控制
- 宽度缺省是它的容器的100%
- 可以容纳内联元素和其他块元素

常见的块级元素有：`<div>`, `<p>`, `<h1>-<h6>`, `<form>`, `<ul>`, `<ol>`, `<dl>`, `<table>`, `<header>`, `<footer>`, `<section>` 等。

行内元素inline elements

- 和相邻的行内元素在一行上
- 高，行高及顶和底边距不可改变
- 宽度只和它的内容有关
- 只能容纳文本或其他行内元素

常见的行内元素有：`<a>`, `<span>`, `<em>`, `<strong>`, `<label>`, `<input>`, `<button>`, `<select>`, `<textarea>` 等。

行内块元素inline-block elements

- 和相邻的行内元素在一行上
- 高度，行高以及顶和底边距都可控制
- 宽度缺省是它的内容的宽度
- 可以容纳其他行内元素和文本

常见的行内块元素有：`<img>`, `<button>`, `<input>` 等。

综上所述，块级元素和行内元素主要区别在于布局方式和容纳内容的类型，而行内块元素则是两者的组合体，同时具有它们各自的部分特性。

内联元素inline elements

内联元素是指在 HTML 页面上默认以行内方式呈现的元素，它们不会在前后插入换行符，而是排列在同一行上。内联元素的宽度和高度默认由其包含的内容决定，无法通过 CSS 显式地指定。

一些常见的内联元素包括：

- `<a>`：定义超链接
- `<span>`：定义文本的小节
- `<img>`：定义图像
- `<input>`：定义输入字段
- `<button>`：定义按钮
- `<label>`：定义标签

需要注意的是，这些元素本身不一定是内联元素，也可以通过 CSS 的 `display` 属性将块级元素转换为内联元素。例如，将 `<div>` 元素的 `display` 属性设置为 `inline` 或 `inline-block`，就可以将其转换为内联元素或内联块级元素。

选择器selectors

CSS选择器是一种用于选择特定HTML元素的语法，可以通过不同的选择器选取HTML中的元素，从而对其进行样式的设置。

以下是常用的CSS选择器：

1. 标签选择器：选取所有指定标签的元素，例如：p {}
2. 类选择器：选取具有指定类名的元素，例如：.classname {}
3. ID选择器：选取具有指定id的元素，例如：#idname {}
4. 后代选择器：选取指定元素内的子元素，例如：ul li {}
5. 子选择器：选取指定元素的子元素，例如：ul > li {}
6. 属性选择器：选取带有指定属性的元素，例如：input[type="text"] {}

```
[attribute]：选择带有指定属性的元素。
例如：[href] 选择带有 href 属性的元素。
[attribute=value]：选择带有指定属性值的元素。
例如：[href="#"] 选择 href 属性为 # 的元素。
[attribute^=value]：选择属性值以指定值开头的元素。
例如：[class^="top"] 选择 class 属性值以 top 开头的元素。
[attribute$=value]：选择属性值以指定值结尾的元素。
例如：[class$="btn"] 选择 class 属性值以 btn 结尾的元素。
[attribute*=value]：选择属性值包含指定值的元素。
例如：[class*="active"] 选择 class 属性值包含 active 的元素。
```

7. 伪类选择器：选取某些特殊状态下的元素，例如：a:hover {}

```
:hover：当元素被鼠标悬浮时
:active：当元素被激活时（比如当链接被点击时）
:focus：当元素获得焦点时（比如当文本框被点击时）
:visited：已访问链接的状态
:first-child：选取元素的第一个子元素
:last-child：选取元素的最后一个子元素
:nth-child(n)：选取元素的第 n 个子元素
:nth-last-child(n)：选取元素的倒数第 n 个子元素

ul li:hover - 当鼠标悬停在ul元素内的li元素上时触发样式。
.container :focus-within - 当焦点位于.container元素内任何一个元素上时触发样式。
input[type="checkbox"] + label:after - 当与<input>元素相邻的<label>元素被选中时触发样式。
nav a:not(:last-child) - 除了nav元素内最后一个<a>元素以外的所有<a>元素。
```

8. 通配选择器：选取所有元素，例如：* {}

```
.container p：选择 .container 元素内的所有 p 元素，包括嵌套的 p 元素。
.container > p：选择 .container 元素直接子级的所有 p 元素。
.container ~ p：选择 .container 元素后面所有同级的 p 元素。
.container + p：选择 .container 元素后面第一个同级的 p 元素。
```

选择器参数

- :nth-child(n)：选择每个父元素下的第 n 个子元素，最小值为 1，如果 n 是 0 则无效。
- :nth-of-type(n)：选择每个父元素下相同类型的第 n 个子元素，最小值为 1，如果 n 是 0 则无效。
- :first-child：选择每个父元素下的第一个子元素，最小值为 1。
- :last-child：选择每个父元素下的最后一个子元素，最小值为 1。
- :first-of-type：选择每个父元素下相同类型的第一个子元素，最小值为 1。
- :last-of-type：选择每个父元素下相同类型的最后一个子元素，最小值为 1。
- :not(selector)：选择不匹配指定选择器的元素，无最小值限制。
- :nth-last-child(n)：选择每个父元素下的倒数第 n 个子元素，最小值为 1，如果 n 是 0 则无效。
- :nth-last-of-type(n)：选择每个父元素下相同类型的倒数第 n 个子元素，最小值为 1，如果 n 是 0 则无效。
- :only-child：选择每个父元素下只有一个子元素的元素，最小值为 1。
- :only-of-type：选择每个父元素下只有一个相同类型子元素的元素，最小值为 1。

需要注意的是，如果选择器的参数无效（例如 :nth-child(0) 或 :nth-of-type(0)），则选择器不会匹配任何元素。

querySelector()

使用 querySelector() 或 querySelectorAll() 方法可以选择 id、class 或属性。

要选择 id，使用 # 符号，后面跟随 id 名称。例如，选择 id 为 "myId" 的元素：

```
const myElement = document.querySelector("#myId");
```

要选择 class，使用 . 符号，后面跟随 class 名称。例如，选择 class 为 "myClass" 的所有元素：

```
const myElements = document.querySelectorAll(".myClass");
```

要选择属性，使用方括号并将属性名称放入其中。例如，选择所有具有 "data-test" 属性的元素：

```
const myElements = document.querySelectorAll("[data-test]");
```

也可以选择带有特定属性值的元素。例如，选择具有 "data-test" 属性和值 "myValue" 的所有元素：

```
const myElements = document.querySelectorAll("[data-test='myValue']");
```

给标签添加属性值的不同方式

可以使用HTML的属性来添加标签的属性值，例如：

```
<input type="text" style="color: red;" disabled>
```

在这个例子中，`style`和`disabled`属性被添加到`<input>`标签中。其中，`style`属性被设置为`color: red;`，表示文本的颜色为红色；`disabled`属性不需要值，表示该输入框被禁用，无法编辑。

可以使用JavaScript来动态地添加标签的属性值，例如：

```
const inputElement = document.querySelector('input');
inputElement.style.color = 'red';
inputElement.disabled = true;
```

在这个例子中，首先使用`document.querySelector()`方法选择一个`<input>`标签，然后使用`.`操作符访问`style`和`disabled`属性，分别将文本颜色设置为红色，以及禁用该输入框。

在现有属性基础上增加

可以使用 JavaScript 中的 DOM 操作来增加新的 style 属性值。可以通过以下方式获取元素的 style 属性值：

```
const myElement = document.querySelector("#myElement");
const currentStyle = myElement.style;
```

然后可以使用 `currentStyle` 对象来访问和修改元素的 style 属性值。例如，要在现有样式的基础上添加一个新的样式，可以使用以下代码：

```
myElement.style.color += "; background-color: yellow";
```

这会在元素的当前样式的基础上添加一个新的 `background-color` 样式。

同样地，也可以使用类似的方式来设置 `disabled` 属性：

```
myElement.disabled = true;
```

或者，如果要通过 JavaScript 设置元素的属性，可以使用以下代码：

```
myElement.setAttribute("disabled", true);
```

这将把 `disabled` 属性设置为 `true`。

无序列表自定义图案

在 HTML 中，我们可以使用 CSS 来自定义无序列表（`<ul>`）的图案。具体来说，我们可以使用 `list-style-type` 属性来指定无序列表项目（`<li>`）的标记类型。

以下是一些常见的标记类型及其对应的值：

- 实心圆：`disc`
- 实心方块：`square`
- 空心圆：`circle`
- 小写字母：`lower-alpha`
- 大写字母：`upper-alpha`
- 小写罗马数字：`lower-roman`
- 大写罗马数字：`upper-roman`

如果以上的标记类型都无法满足需求，我们还可以使用自定义图片作为标记，具体步骤如下：

1. 创建一张大小为 16x16 像素的 PNG 图片，例如 `marker.png`。
2. 在 CSS 样式表中，使用 `list-style-image` 属性来指定图片 URL，如 `list-style-image: url(marker.png);`。
3. 将 `list-style-type` 属性设置为 `none`，以去除默认的标记。

以下是一个示例代码：

HTML：

```
<ul class="custom-list">
  <li>第一项</li>
  <li>第二项</li>
  <li>第三项</li>
</ul>
```

CSS：

```
.custom-list {
  list-style-type: none;
  padding-left: 0;
}

.custom-list li {
  list-style-image: url(marker.png);
  margin-bottom: 10px;
}
```

以上代码将无序列表的默认标记去除，并将自定义图片作为标记。注意，由于不同的浏览器可能对于自定义标记的显示效果支持程度不同，因此需要进行兼容性测试。

给标签定义style的不同方式

可以通过以下几种方式给一个标签定义 style：

1. 直接在 HTML 标签上定义 style 属性：

```
<div style="color: red; font-size: 16px;">Hello World</div>
```

2. 使用 style 标签在 HTML 文件内部定义样式：

```
<head>
  <style>
    div {
      color: red;
      font-size: 16px;
    }
  </style>
</head>
<body>
  <div>Hello World</div>
</body>
```

3. 使用外部 CSS 文件来定义样式，并通过 link 标签将其引入到 HTML 文件中：

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <div>Hello World</div>
</body>
```

4. 使用 JavaScript 在运行时动态地为标签添加样式：

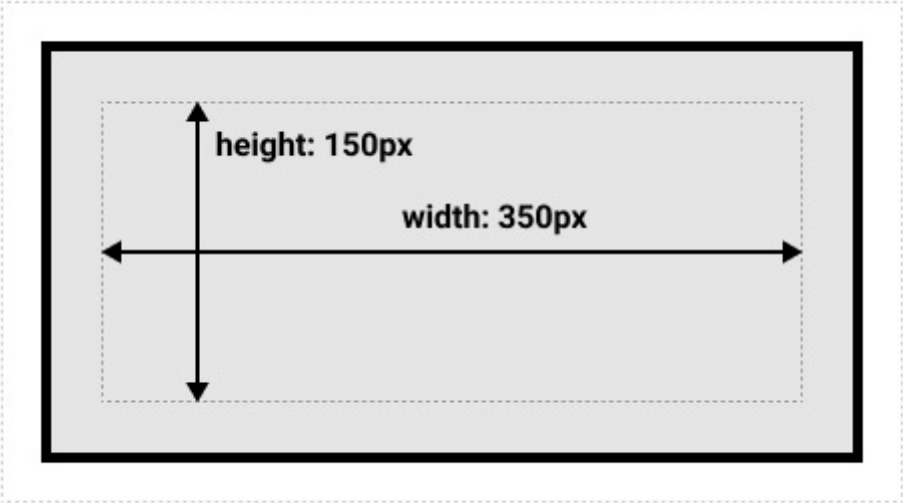
```
const div = document.createElement('div');
div.style.color = 'red';
div.style.fontSize = '16px';
div.textContent = 'Hello World';
document.body.appendChild(div);
```

Box Model

CSS Box Model 是 CSS 中一种重要的布局模型，它将每个 HTML 元素看作是一个矩形的盒子（box），并规定了这个盒子所包含的内容、内边距、边框和外边距，从而构成了一个完整的盒子模型。

在 CSS Box Model 中，每个盒子由以下四个部分组成：

1. Content（内容）：盒子所包含的内容，如文字、图片、媒体等。

> 在 CSS 的标准盒模型中，一个盒子的大小由四个部分组成：content box（内容区域）、padding box（内边距区域）、border box（边框区域）和 margin box（外边距区域）。**当你设置一个盒子的 `width` 和 `height` 属性时，实际上是在定义该盒子的 content box 的宽度和高度。**
>
> padding 和 border 的宽度会被添加到 content box 的宽度和高度上，以得到整个盒子的宽度和高度。所以，如果你设置了一个盒子的 `width` 和 `height`，那么盒子的总宽度和高度将会是 `width + padding-left + padding-right + border-left-width + border-right-width` 和 `height + padding-top + padding-bottom + border-top-width + border-bottom-width`。
>
> 
>
>
> **使用 `box-sizing: border-box;` 可以改变盒子的计算方式，使设置的宽度和高度包括了内边距和边框。**这样，元素的实际宽度和高度就是你设置的宽度和高度。
>
> 例如，如果我们有一个元素宽度为 200px，内边距为 10px，边框为 2px，那么元素的实际宽度在标准盒模型下为 214px（200px + 2 * 10px + 2 * 2px），而在使用 `box-sizing: border-box;` 的盒模型下为 200px。
>
> 这种盒模型更加符合实际开发中的需求，因为开发者可以更加直观地设置元素的宽度和高度，而不必再手动计算内边距和边框所占据的空间。同时，使用 `box-sizing: border-box;` 也有助于处理响应式布局和弹性布局等情况，使元素的大小更加稳定。

2. Padding（内边距）：盒子边缘与内容之间的空白区域，可以设置背景色、背景图等样式。
3. Border（边框）：盒子边缘的线条，可以设置颜色、宽度、样式等属性。
4. Margin（外边距）：盒子与周围元素之间的空白区域，不显示任何内容，可以控制元素的位置。

> 在 CSS 中，可以使用多种方式定义 margin 属性，包括：
>
> 1. 四个值：**上、右、下、左**。例如：`margin: 10px 20px 30px 40px;`
> 2. 三个值：**上、左右、下**。例如：`margin: 10px 20px 30px;`
> 3. 两个值：**上下、左右**。例如：`margin: 10px 20px;`
> 4. 单个值：**所有方向**。例如：`margin: 10px;`
> 5. 使用 margin-top、margin-right、margin-bottom、margin-left 分别定义
>
> 需要注意的是，如果只定义了一个值，其他方向会继承该值，如果没有定义值，所有方向的 margin 值默认为 0。

inline box vs. inline-block box

在 CSS 盒模型中，inline box（内联级盒子）和 inline-block box（内联块级盒子）是两种不同的盒子类型，它们具有不同的特性和用途。

inline box 是指与文本流同行显示的元素，它们的宽度和高度是由其包含的内容和内边距（padding）来决定的，同时也受到最小和最大宽度的限制。inline box 的典型代表包括 `<span>`、`<a>`、`<img>` 等元素。

而**inline-block box** 则是指既具有 inline 特性又具有 block 特性的元素，它们可以像 inline 元素一样在同一行显示，但也可以设置宽度和高度，并且可以应用盒模型的内边距、边框和外边距等属性。inline-block box 的典型代表包括 `<button>`、`<input>`、`<textarea>` 等表单元素。

对于 Block box（块级盒子），width 和 height 属性会起作用，可以用来设置该盒子的宽度和高度。Block box 是指独占一行的元素，例如 `<div>`、`<p>`、`<ul>`、`<ol>` 等。

而**对于 Inline box（内联级盒子），width 和 height 属性则不会起作用**，不能用来设置盒子的宽度和高度。Inline box 是指与文本流同行显示的元素，例如 `<span>`、`<a>`、`<img>` 等。

需要注意的是，可以通过将 inline box 转换为 block box 或 inline-block box，来使其支持 width 和 height 属性。

dpi

在 CSS 中，`dpi` 是指屏幕的像素密度，即每英寸（2.54 厘米）上的像素数。DPI 表示每英寸上的点数，通常被用于打印机等输出设备。而在 Web 开发中，我们更常用的是像素密度（Pixel Density），它是衡量屏幕显示效果的重要指标之一。

具体来说，像素密度是指屏幕上每英寸的像素数，它的单位通常是 PPI（Pixels Per Inch），也就是每英寸上的像素数。例如，一台分辨率为 1920x1080 的电脑显示器，它的屏幕尺寸为 21.5 英寸，那么它的像素密度就是 102.46 PPI（根据勾股定理计算得出）。

在 CSS 中，我们可以使用 `dpi` 和 `dppx` 两个单位来表示像素密度，其中 `dpi` 表示每英寸上的像素数，而 `dppx` 表示每像素上的设备像素数。例如，如果设备的像素密度为 2 倍（即 2 像素对应一个设备像素），那么它的 `dppx` 值为 2。在 CSS 中，我们可以使用媒体查询来根据不同的像素密度应用不同的样式，以保证在不同设备上的显示效果一致。

Units

CSS中的单位分为绝对单位和相对单位。

绝对单位：

1. px（pixel，像素）：像素是最基本的单位，是屏幕上显示的最小单位，通常用于固定大小的元素，如边框、尺寸等。
2. in（inch，英寸）：1英寸=96像素，屏幕大小可以通过英寸来指定，但是在不同设备上显示的大小可能会有所不同。
3. cm（centimeter，厘米）：1厘米=37.8像素，与英寸类似，用于屏幕大小的指定。

相对单位：

1. em：以父元素的字体大小作为基准，比如在一个16px的文本中设置1em的大小，就是16px。
2. rem：以根元素（即html元素）的字体大小作为基准，与em的区别在于，rem不会受到父元素的影响，更加稳定。如果没有定义根元素的字体大小，默认情况下，rem 将等于浏览器的默认字体大小，通常为 16 像素。
3. %：以父元素的宽度为基准，比如一个元素设置width: 50%就是相对于其父元素的宽度计算得到的。
4. vw（viewport width）：视口宽度的百分比，1vw表示视口宽度的1%。
5. vh（viewport height）：视口高度的百分比，1vh表示视口高度的1%。
6. vmin：视口宽度和高度中较小的那个值的百分比，1vmin表示视口宽度和高度中较小的那个值的1%。
7. vmax：视口宽度和高度中较大的那个值的百分比，1vmax表示视口宽度和高度中较大的那个值的1%。

此外，还有一些特殊的单位：

1. ex：基于当前字体的 x-height（字母 x 的高度）。
2. ch：基于 "0" 的宽度。
3. pt（point，点）：用于打印单位，1pt=1/72英寸。

需要注意的是，相对单位可以根据不同的使用场景灵活设置，而绝对单位是固定的。一般情况下，推荐使用相对单位，因为它们能够自适应不同的设备和屏幕大小，更加灵活。

基于最近的祖先单位

CSS 有一些单位是相对于最近的祖先元素来确定的，这些单位包括：

1. `em`：相对于父元素的字体大小。例如，如果一个元素的字体大小为 16px，其子元素使用 2em 作为宽度，则子元素的宽度将为 32px。
2. `rem`：相对于根元素的字体大小。例如，如果根元素的字体大小为 16px，一个元素使用 2rem 作为宽度，则该元素的宽度将为 32px。
3. `ex`：相对于字母 x 的高度。在大多数字体中，字母 x 的高度大约是字体大小的一半左右。
4. `ch`：相对于数字 0 的宽度。在大多数字体中，数字 0 的宽度与其他字符的宽度大致相等。
5. `vw` 和 `vh`：相对于视口的宽度和高度。1vw 表示视口宽度的 1%，1vh 表示视口高度的 1%。这些单位在创建响应式布局时非常有用。
6. `vmin` 和 `vmax`：相对于视口宽度和高度中的最小值和最大值。例如，1vmin 表示视口宽度和高度中的较小值的 1%。

7. `%`：相对于父元素的宽度。如果一级父元素没有定义宽度，则子元素的百分比宽度会根据其最近的有定义宽度的祖先元素来计算。如果没有找到有定义宽度的祖先元素，则百分比宽度将根据浏览器窗口宽度来计算。这个过程被称为“相对于包含块计算宽度”。

需要注意的是，这些相对单位的计算都基于最近的祖先元素。如果没有找到有效的祖先元素，则会默认使用根元素作为计算基础。因此，在进行布局时，我们需要特别注意每个元素的定位和宽高计算，以免出现意外的偏差。

如果所有祖先元素都没有定义相应的宽度或高度，则这些相对单位将根据默认值计算。例如，`em` 相对于其父元素的字体大小，如果祖先元素没有定义字体大小，则根据浏览器的默认值进行计算。`rem` 相对于根元素的字体大小，如果根元素没有定义字体大小，则根据浏览器的默认值进行计算。`%` 相对于其包含块的宽度或高度，如果所有祖先元素都没有定义宽度或高度，则根据包含块的默认值计算。

em

1. 如果在默认情况下，浏览器的默认字体大小为16px，那么1em就等于16px。如果当前元素的字体大小为1.5em，那么就是1.5 * 16 = 24px。
2. 如果在style所在的标签内设置了字体大小，比如在body标签内设置字体大小为20px，那么1em就等于20px。如果当前元素的字体大小为1.5em，那么就是1.5 * 20 = 30px。
3. 以下是一些带em的例子：

```
body {
  font-size: 20px;
}
h1 {
  font-size: 2em; /* 2em = 40px */
}
p {
  font-size: 1.2em; /* 1.2em = 24px (assuming default font size of 20px) */
  line-height: 1.5em; /* 1.5em = 30px (assuming default font size of 20px) */
}
```

在这个例子中，h1元素的字体大小是2em，因为它是body标签内的直接子元素，所以它的字体大小是相对于body标签内的字体大小计算的。p元素的字体大小是1.2em，它的行高是1.5em，都是相对于body标签内的字体大小计算的。

在 CSS 中，`em` 单位的值是相对于其父元素的字体大小而言的。也就是说，如果没有修改其父元素的字体大小，那么 `em` 单位的值将会与默认的字体大小（一般为 `16px`）有关系。当我们修改了父元素的字体大小，那么子元素的 `em` 值也会相应地被调整。如果在样式中直接设置了当前元素的字体大小，那么它的 `em` 值将会是相对于当前元素字体大小而言的。

在CSS中，使用百分比（%）来定义元素的宽度或高度时，可以根据其父元素的宽度或高度进行相对设置。例如，如果设置一个元素的宽度为50%，则它将占据其父元素宽度的50%。

具体来说，使用%来定义元素的宽度或高度时，所使用的百分比是相对于父元素的宽度或高度而言的。例如，如果一个子元素的宽度设置为50%，那么它的宽度将是其父元素宽度的50%。如果父元素的宽度发生变化，那么子元素的宽度也会随之变化。同样的道理也适用于使用百分比来定义元素的高度。

Some More Design Principles

Gestalt

Gestalt 是一种设计原则和方法论，主要用于帮助设计师和开发人员创建一致、整洁和易于使用的用户界面。它最初是在德国心理学家和哲学家Max Wertheimer的工作中提出的，用于解释人们是如何组织视觉元素的。

在设计中，Gestalt 原则可以用来指导以下方面的决策：

1. 对齐：将元素对齐，使它们在视觉上形成一组。
2. 重复：在设计中重复元素或样式，以增强视觉统一性。
3. 对比：使用颜色、大小、形状等元素之间的对比来突出重要信息。
4. 接近性：将相关的元素放在一起，以便在视觉上形成组或部分。
5. 简洁性：使用简单、明了的元素和排版方式，以增强可读性和易用性。

通过使用这些原则，设计师可以创建出更加清晰、整洁和易于使用的界面，提高用户的体验和满意度。

Whitespace

在 CSS 中，Whitespace（空白）指的是文本中的空格、换行符和制表符等空白字符。它们在文本的渲染和布局中扮演着重要的角色。

在默认情况下，浏览器会将连续的空白字符缩减为一个空格，并忽略行尾空格。但是，有时候我们需要保留这些空白字符，例如在代码中显示一段预格式化的文本，或者在段落之间添加额外的空白。此时，我们可以使用 CSS 中的 `white-space` 属性来控制空白的处理方式。

`white-space` 属性有以下几个取值：

- `normal`：默认值。连续的空白字符会被缩减为一个空格，并忽略行尾空格。
- `pre`：空白字符会被保留，并且文本中的换行符和制表符也会被保留。
- `nowrap`：连续的空白字符会被缩减为一个空格，但是不会忽略行尾空格。
- `pre-wrap`：空白字符会被保留，但是行尾空格会被忽略。
- `pre-line`：空白字符会被保留，但是连续的空白字符会被缩减为一个空格，并忽略行尾空格。

使用 `white-space` 属性可以控制空白字符的处理方式，从而更好地控制文本的渲染和布局。

```
pre {
  white-space: pre;
}
```

Responsive Design

CSS Grid

CSS Grid 是一种用于网格布局的 CSS 模块，可以用于创建二维的布局。网格布局通常用于构建复杂的、多列或多行的布局，例如网站的主要布局或者嵌套的组件。

Grid 使用一个网格容器（grid container）和一组网格项（grid item）来定义布局。容器是被赋予 display: grid 属性的元素，它定义了一个新的网格上下文，它包含了一个或多个网格行（grid row）和网格列（grid column），它们定义了网格项的位置。每个网格项是容器的直接子元素，通过 grid-template-rows 和 grid-template-columns 属性，指定其行数和列数，并通过 grid-row 和 grid-column 属性指定其在网格中的位置。

下面是一个使用 Grid 布局的示例：

```
<div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
</div>

.grid-container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 100px 100px;
  gap: 10px;
}

.grid-item {
  background-color: #ddd;
  text-align: center;
  padding: 20px;
}
```

在上面的例子中，我们定义了一个网格容器，它包含了 3 列和 2 行。每个网格项都被指定为具有 1 个跨度的行和列。通过 `gap` 属性，我们还定义了网格中每个项目之间的间距。

使用 Grid 布局可以轻松地实现响应式设计，例如通过媒体查询来定义不同的网格模板，以适应不同的设备屏幕尺寸。

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-gap: 10px;
}

.item {
  grid-column: 2 / 3;
  grid-row: 1 / 3;
}
```

Holy Grail

Holy Grail是指一种常见的网页布局方式，该布局具有三列，其中两个列为固定宽度，中间的列为自适应宽度，使得页面布局可以自适应各种不同的设备和屏幕大小。这种布局方式被称为Holy Grail，因为它可以很好地解决页面布局的许多挑战，就像圣杯一样珍贵和难以获得。

在Holy Grail布局中，通常会有一个header和footer，它们会固定在页面的顶部和底部。左侧和右侧的侧边栏通常有固定的宽度，中间的列则会自适应宽度，以适应不同的屏幕大小。

实现Holy Grail布局的常用方式是使用CSS Grid布局，通过在父元素上定义grid-template-areas，来指定每个子元素在网格中的位置。在这种布局中，通常会将中间的列设置为一个网格单元，并使用auto-fit和minmax函数来自适应宽度。这样可以确保中间的列会自适应父元素的宽度，并且在空间允许的情况下，尽可能地占用剩余的空间。

```
<body>
  <header>Header</header>
  <div class="container">
    <main>Main Content</main>
    <nav>Navigation</nav>
    <aside>Aside Content</aside>
  </div>
  <footer>Footer</footer>
</body>
```

```
body {
  display: grid;
  grid-template-rows: auto 1fr auto;
  grid-template-columns: 1fr minmax(0, 960px) 1fr;
  min-height: 100vh;
}

header,
footer {
  grid-column: 1 / -1;
}

.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-column: 2 / -2;
}

main {
  grid-column: 2 / 3;
}

nav {
  grid-column: 1 / 2;
}

aside {
```

```
  grid-column: 3 / 4;
}
```

`grid-row: span 2;` 定义了该元素跨越两个网格行。如果该元素所在的网格容器中定义了网格行的高度，则该元素的高度为两个网格行的高度之和；如果未定义，则由内容撑开。宽度则取决于其他属性的定义，如 `grid-column`、`width`、`min-width` 等。

`grid-template-columns` 是 CSS Grid 布局中用于定义网格列的属性，它有多种用法：

1. 固定宽度列：可以通过指定一个固定宽度值或百分比来定义每一列的宽度。例如，以下代码将创建一个具有三个等宽列的网格布局：

   ```
   cssCopy code
   grid-template-columns: 1fr 1fr 1fr;
   ```

2. 自动列：可以使用 `auto` 关键字来指定一个自动列，让其宽度自适应内容宽度。例如，以下代码将创建一个具有两个固定宽度列和一个自适应列的网格布局：

   ```
   arduinoCopy code
   grid-template-columns: 100px auto 200px;
   ```

3. 网格行线名称：可以使用方括号来为网格行定义名称，使得它们可以用来放置网格项。例如，以下代码将创建一个具有三列和两个行的网格布局，并将第一行命名为 `header`、第二行命名为 `main`：

   ```
   cssCopy codegrid-template-columns: 1fr 1fr 1fr;
   grid-template-rows: [header] 100px [main] auto;
   ```

4. 网格行线大小：可以使用长度、百分比或 `minmax()` 函数来定义行高。例如，以下代码将创建一个具有三列和两行的网格布局，并将第一行设置为 100 像素高，第二行设置为最小高度 200 像素，最大高度 1fr：

   ```
   cssCopy codegrid-template-columns: 1fr 1fr 1fr;
   grid-template-rows: 100px minmax(200px, 1fr);
   ```

5. 重复网格行：可以使用 `repeat()` 函数来定义一个重复的网格行模式。例如，以下代码将创建一个具有三列和五个行的网格布局，并将前两行设置为固定高度 100 像素，其余行设置为最小高度 200 像素，最大高度 1fr：

   ```
   cssCopy codegrid-template-columns: 1fr 1fr 1fr;
   grid-template-rows: repeat(2, 100px) minmax(200px, 1fr) repeat(2, 1fr);
   ```

总之，`grid-template-columns` 是一个非常强大和灵活的属性，可以使用多种方式来定义网格列。

Explicit grid

以下是一些常用的CSS Grid属性及其解释：

1. grid-template-rows/grid-template-columns：定义网格的行数和列数。 例如：

```
.grid-container {
  display: grid;
  grid-template-rows: 100px 200px 50px; /* 定义三行，高度分别为100px、200px、50px */
  grid-template-columns: 50px 100px 200px; /* 定义三列，宽度分别为50px、100px、200px */
}
```

1. grid-row-start/grid-row-end/grid-column-start/grid-column-end：定义元素在网格中的位置，从网格左上角开始计算。其中，grid-row-start/grid-column-start用于定义元素在哪个行/列开始，grid-row-end/grid-column-end用于定义元素在哪个行/列结束。例如：

```
.item {
  grid-row-start: 2; /* 元素开始于第二行 */
  grid-row-end: 4; /* 元素结束于第四行 */
  grid-column-start: 1; /* 元素开始于第一列 */
  grid-column-end: 3; /* 元素结束于第三列 */
}
```

1. grid-template-areas：用字符表示法来定义网格区域。例如：

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(3, 1fr);
  grid-template-areas:
    "header header header"
    "nav article aside"
    "footer footer footer";
}

.header {
  grid-area: header;
}

.nav {
  grid-area: nav;
}

.article {
  grid-area: article;
}

.aside {
  grid-area: aside;
}
```

```
.footer {
  grid-area: footer;
}
```

1. grid-gap：定义网格行和列之间的间隔。例如：

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(3, 1fr);
  grid-gap: 20px;
}
```

1. justify-items/align-items：定义网格容器内元素在网格中的对齐方式。例如：

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(3, 1fr);
  justify-items: center; /* 水平居中对齐 */
  align-items: center; /* 垂直居中对齐 */
}
```

1. justify-content/align-content：定义网格容器内所有网格的对齐方式。例如：

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(3, 1fr);
  justify-content: center; /* 所有列水平居中对齐 */
  align-content: center; /* 所有行垂直居中对齐 */
}
```

Implicit grid

在 CSS Grid 中，如果某一行或列没有被明确指定网格线，就会自动形成一个称为“隐式网格”的网格线，这样的网格线就是隐式网格线。可以通过下列属性来控制隐式网格线的行为：

1. grid-auto-rows：指定未指定的行的大小。
2. grid-auto-columns：指定未指定的列的大小。
3. grid-auto-flow：指定如何排列那些没有放入明确的网格中的项目，有行优先、列优先和密集优先等几种选项。

使用这些属性可以让我们更加自由地控制网格中未被明确指定的部分，让页面更加灵活自适应。

以下是一个简单的示例，展示了如何使用Implicit grid属性：

HTML代码：

```
<div class="container">
  <div class="item item1">1</div>
  <div class="item item2">2</div>
  <div class="item item3">3</div>
  <div class="item item4">4</div>
  <div class="item item5">5</div>
</div>
```

CSS代码：

```
.container {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-gap: 10px;
}

.item {
  background-color: #ddd;
  padding: 10px;
  text-align: center;
}

.item1 {
  grid-column-start: 1;
  grid-column-end: 3;
}

.item2 {
  grid-row-start: 2;
  grid-row-end: 4;
}

.item3 {
  grid-column-start: 3;
  grid-column-end: 4;
}

.item4 {
  grid-row-start: 3;
}

.item5 {
  /* No grid-row-start and grid-row-end values specified */
}
```

在这个示例中，我们定义了一个3列的grid布局，并使用grid-gap属性定义了元素之间的间距。然后我们定义了5个grid项目（即5个包含内容的元素），并使用grid-column-start，grid-column-end，grid-row-start和grid-row-end属性定义它们在grid中的位置。

注意，最后一个元素（即item5）没有指定grid-row-start和grid-row-end值，它将被自动放置在grid的下一个可用单元格中，这就是Implicit grid的作用。这个元素将自动创建一个新的grid行来容纳它。

grid-area

`grid-area` 是一个 CSS Grid 布局中用来指定网格项位置和大小的属性。它可以同时指定网格项的行、列起始和结束位置，相当于使用了 `grid-row-start`、`grid-column-start`、`grid-row-end` 和 `grid-column-end` 这四个属性。使用 `grid-area` 属性可以更为简洁地定义网格项的位置和大小。

`grid-area` 属性的语法为 `grid-area: <row-start> / <column-start> / <row-end> / <column-end>;`，其中 `<row-start>`、`<column-start>`、`<row-end>` 和 `<column-end>` 可以为数字、关键字 `span` 或者自定义的命名网格线。

例如，如果想让一个网格项占据第一行到第三行的第一列到第三列，可以这样写：

```
.item {
  grid-area: 1 / 1 / 4 / 4;
}
```

其中 `1 / 1 / 4 / 4` 表示该网格项的起始行为第一行，起始列为第一列，结束行为第四行，结束列为第四列。也可以用 `span` 关键字来代替数字，例如：

```
.item {
  grid-area: 1 / 1 / span 3 / span 3;
}
```

表示该网格项的起始行为第一行，跨越三行，起始列为第一列，跨越三列。

使用 `grid-area` 属性还可以通过自定义命名网格线来更方便地布局网格项，例如：

```
.grid {
  display: grid;
  grid-template-columns: [start] 1fr [mid] 2fr [end];
  grid-template-rows: [top] 100px [mid] 200px [bottom];
}

.item {
  grid-area: top / start / bottom / end;
}
```

其中 `grid-template-columns` 和 `grid-template-rows` 定义了自定义的网格线，而 `grid-area` 属性使用了这些自定义的网格线来指定网格项的位置和大小。

content和items区别

`justify-content` 和 `align-content` 属性分别是针对网格容器内容的行轴（水平方向）和列轴（垂直方向）进行对齐的，只有当网格容器的内容大于网格容器的尺寸时，这两个属性才会起作用。

而 `justify-items` 和 `align-items` 属性则是针对网格项目的行轴和列轴进行对齐的。

因此，`justify-content` 和 `align-content` 属性针对的是网格容器，而 `justify-items` 和 `align-items` 属性针对的是网格项目。如果将 `.grid-container` 中的 `justify-content` 和 `align-content` 属性改为 `justify-items` 和 `align-items` 属性，那么就是将对齐属性从网格容器级别改为了网格项目级别。这意味着，网格项目将会根据 `justify-items` 和 `align-items` 的值在网格单元格内对齐，而不再是对齐整个网格容器的内容。

假设我们有一个包含两个元素的网格容器和四个元素的网格项目。

```
<div class="grid-container">
  <div class="grid-item">Item 1</div>
  <div class="grid-item">Item 2</div>
  <div class="grid-item">Item 3</div>
  <div class="grid-item">Item 4</div>
</div>
```

在上面的代码中，`.grid-container`就是网格容器，它定义了一个网格的结构，决定了网格项目应该如何排列。而`.grid-item`就是网格项目，它是网格中的实际元素，被放置在网格中，根据网格容器的排列规则摆放。

简单来说，网格容器决定了网格结构，网格项目则是在网格中实际占据位置的元素。

Flexbox

```
.container {
  display: flex;
  flex-direction: row;
  justify-content: space-between;
  align-items: center;
}

.item {
  flex-basis: 20%;
  height: 50px;
}
```

在弹性布局中，这四个属性设置为center产生的效果如下：

justify-content: 在单行和多行中都是在主轴方向上整体居中；

justify-items：在弹性布局中没有效果，该属性会被忽略。

align-content: 只在多行情况下有效，多行元素会整体居中。

align-items: 单行和多行都是在所在行中居中，这里区别下整体居中（align-content）。

单行：justify-content 主轴居中，align-items 次轴居中

多行：justify-content 主轴居中， align-content 次轴整体居中, align-items 各行内居中

justify对应主轴，align对应次轴。content对应的是整体，items对应的是每个元素所在的那个周边区域。

max-width

`max-width` 是一个 CSS 属性，用于指定一个元素的最大宽度。当元素的实际宽度超过了 `max-width` 的值时，元素的宽度将被限制在 `max-width` 的值以内。

`max-width` 的作用主要是防止内容区域的宽度过大，导致内容无法适应不同尺寸的设备屏幕。通常情况下，应该使用 `max-width` 来控制元素的最大宽度，而不是使用 `width`，这可以确保元素不会超出其容器的边界。

一个必须使用 `max-width` 的情况是当你想要在不同尺寸的设备上显示相同的布局时。如果你使用一个固定的 `width` 值，那么当屏幕尺寸小于该值时，元素的内容可能会溢出或者被截断，导致显示不完整。使用 `max-width` 可以确保元素在不同屏幕尺寸下都能够正确地适应，并保持相同的布局。

`max-width` 的值可以是任何长度单位或百分比值。常见的值包括像素、百分比和 em 等。还可以使用 `none`，表示无限制最大宽度，或者使用 `initial`，表示使用元素的默认宽度作为最大宽度。

Media Queries

```
@media (max-width: 768px) {
  .container {
    display: flex;
    flex-direction: column;
  }

  .item {
    width: 100%;
  }
}
```

以下是几个常用的 Media queries 的例子：

1. 在屏幕宽度小于等于 768 像素时，应用以下样式：

```
@media (max-width: 768px) {
  /* 样式 */
}
```

1. 在屏幕宽度在 769 像素到 1024 像素之间时，应用以下样式：

```
@media (min-width: 769px) and (max-width: 1024px) {
  /* 样式 */
}
```

1. 在屏幕高度小于等于 480 像素时，应用以下样式：

```
@media (max-height: 480px) {
  /* 样式 */
}
```

1. 在设备是打印机时，应用以下样式：

```
@media print {
  /* 样式 */
}
```

这些 Media queries 可以根据需要进行修改，以适应不同的页面布局和设计需求。

Others

text-align和align-items的区别

`text-align` 和 `align-items` 是两个不同的 CSS 属性，分别用于控制文本或内容的水平或垂直对齐方式。

- `text-align` 用于控制文本在容器内的水平对齐方式，适用于容器内有文本的情况。可以接受的值包括 left、center、right、justify 等。
- `align-items` 用于控制容器内元素的垂直对齐方式，适用于容器内有多个元素的情况。可以接受的值包括 stretch、center、flex-start、flex-end、baseline 等。

它们的区别主要在于作用对象和作用方式不同。`text-align` 作用于文本，控制文本在容器内的水平对齐方式；`align-items` 作用于容器内的元素，控制元素在容器内的垂直对齐方式。

使用 `margin: 0; padding: 0;` 清除元素默认样式

使用 `margin: 0; padding: 0;` 的目的是为了清除元素的默认样式，以避免元素因为默认的 `margin` 和 `padding` 值而产生意料之外的效果，比如两个元素之间有不必要的空隙，或者元素的大小不符合预期。

如果我们想要完全控制元素的外观和布局，通常需要使用 `margin: 0; padding: 0;` 来重置元素的默认样式，然后再根据需要设置具体的 `margin` 和 `padding` 值。同时，使用这个方法也可以避免一些浏览器之间的差异和 bug，以保证样式的一致性。

清除元素的默认样式只会清除元素的默认外观属性，比如 `margin`、`padding`、`border`、`font-size`、`line-height` 等。它不会清除所有的样式，比如背景色、文本颜色、文本对齐等样式不会被清除。

border-collapse: collapse

`border-collapse` 属性用于控制表格中相邻单元格边框的重叠方式。默认情况下，相邻单元格的边框会分开绘制，而 `border-collapse: collapse;` 可以让相邻单元格的边框重合在一起，形成一个连续的边框线。

这个属性可以应用于 `<table>` 元素以及 `<td>`、`<th>` 等单元格元素。当应用于 `<table>` 元素时，会影响表格中所有单元格的边框重叠方式。当应用于单元格元素时，会影响该单元格与其相邻单元格的边框重叠方式。

需要注意的是，当单元格边框重合时，其宽度将变为两条边框的宽度之和。例如，两个相邻单元格的边框宽度都为 1px，则它们重合后的边框宽度将为 2px。

JavaScript

Execution Context

Execution Context 是 JavaScript 引擎在执行代码时创建的一个环境，其中包含了当前代码所需的变量、函数和上下文等信息。

当浏览器读取 JavaScript 代码时，它会逐行解释并在执行代码之前先创建一个全局 Execution Context。全局 Execution Context 中包含了 JavaScript 程序运行时所需要的全局变量、函数和其他信息，同时还会在内存中创建一个全局作用域链。当执行到函数时，JavaScript 引擎会为每个函数调用创建一个局部 Execution Context，并在内存中创建一个新的作用域链。

```
var a = 2;
function add (num) {
   var resultlocal = num + 10;
}
var resultglobal = add(a);
```

在上述例子中，全局 Execution Context 包含了变量 a 和函数 add。当执行到函数 add 时，JavaScript 引擎会为其创建一个局部 Execution Context，并在其中创建变量 resultlocal，用于存储 num + 10 的结果。在执行完 add 函数后，该函数的局部 Execution Context 将被销毁，并释放相应的内存空间。

值得注意的是，函数是具有局部作用域的，它们在全局作用域中是不可见的，只有在函数被调用时才会创建对应的局部 Execution Context，并在其中执行代码。

Hoisting

Hoisting（变量提升）是 JavaScript 中的一个概念，指的是在代码执行过程中，变量和函数的声明会被提升到它们所在作用域的顶部，不受代码书写顺序的影响。

具体来说，在 JavaScript 中，变量和函数的声明会被分离成两个阶段：编译阶段和执行阶段。在编译阶段，变量和函数的声明会被提升到当前作用域的顶部，形成一个变量和函数的"声明池"。在执行阶段，变量和函数的定义（也就是赋值）会按照代码书写顺序执行，不会被提升。

这个概念主要涉及两个方面：变量和函数。

对于变量，**变量声明会被提升，但是赋值操作不会被提升。**例如：

```
console.log(a); // 输出 undefined
var a = 10;
```

上面代码中，变量 `a` 的声明会被提升，但是赋值操作不会被提升，因此第一行输出 `undefined`。

对于函数，**函数声明会被提升，但是函数表达式不会被提升。**例如：

```
foo(); // 输出 "Hello"
function foo() {
  console.log("Hello");
}
bar(); // 报错
var bar = function() {
  console.log("World");
}
```

上面代码中，函数 `foo` 的声明会被提升，因此可以在函数声明之前调用 `foo` 函数。但是函数 `bar` 的定义是一个函数表达式，不会被提升，因此在定义之前调用 `bar` 函数会报错。

在使用变量和函数时，建议将它们的声明和定义放在使用它们的代码之前，以便避免因为 Hoisting 导致的错误。

Scope in JS

在 JavaScript 中，作用域（scope）是指在程序中定义变量的区域。变量在代码中定义的位置决定了它们在程序中的可见性和生命周期。

词法作用域（lexical scope）意味着变量的作用域在编写代码时就已经确定了。这意味着变量在创建时会沿着嵌套的作用域链向上查找它们的父级作用域，直到找到定义该变量的作用域。

```
function first(){
   console.log(a);
   function second(){
       var a = 10;
   }
}
first();
```

在上面的例子中，`first()` 函数包含了一个 `console.log(a)` 语句，但是 `a` 变量没有在函数内部声明。因此，当 `first()` 函数被调用时，它会尝试在其父级作用域中寻找 `a` 变量，即全局作用域。如果 `a` 变量在全局作用域中不存在，那么将抛出一个 ReferenceError 错误。

另外，`first()` 函数包含一个名为 `second()` 的子函数，但是在 `first()` 函数中并没有声明 `a` 变量，所以 `second()` 函数无法访问 `first()` 函数中的 `a` 变量，会在控制台输出未定义的错误。这是因为 JavaScript 中的作用域链是由函数的定义位置决定的。在这个例子中，`a` 变量的作用域是全局作用域，而 `second()` 函数是 `first()` 函数的子函数，所以 `second()` 函数无法访问 `first()` 函数中的 `a` 变量。

因此，JavaScript 中的作用域链是由函数定义的位置决定的，而不是函数被调用的位置决定的。

当一个函数被定义时，它的作用域被确定，与函数被调用的位置无关。例如，考虑以下代码：

```
var a = 1;

function foo() {
  var b = 2;

  function bar() {
    var c = 3;
    console.log(a, b, c);
  }

  return bar;
}

var fn = foo(); // 将 bar 函数赋值给变量 fn
fn(); // 输出 1, 2, 3
```

在这个例子中，函数 `bar` 内部嵌套在函数 `foo` 中，因此 `bar` 的作用域包含了 `foo` 的作用域和全局作用域。因此，在 `bar` 中，可以访问全局变量 `a` 和 `foo` 中定义的局部变量 `b`，但无法访问 `foo` 之外的变量。在函数 `foo` 中，将函数 `bar` 赋值给了变量 `fn`，并返回它。因此，在调用 `fn()` 时，实际上是在调用函数 `bar`，这时就能访问到 `foo` 中定义的变量 `b` 了。

这个例子说明，函数的作用域是在函数定义时就确定的，与函数的调用位置无关。函数定义的位置决定了函数作用域的范围和所包含的变量。

Scope chain 也是一个重要的概念，它代表了一个变量的可访问范围。当 JavaScript 引擎查找一个变量时，会先从当前作用域开始查找，如果没有找到就一直向上查找直到找到该变量或到达全局作用域为止。

在执行 `bar()` 函数时，它会先在自己的作用域内查找变量 `c`，如果找到了就使用该变量，否则就会向上查找。在这个例子中，它会找到包含它的 `foo()` 函数的作用域，查找变量 `b`，并使用它。如果在 `foo()` 的作用域中也找不到变量 `b`，就会继续向上查找，最终到达全局作用域，查找变量 `a`。

这个查找的过程就是 Scope Chain（作用域链），它是由函数定义时的位置决定的。在这个例子中，变量 `a` 的作用域是全局作用域，而变量 `b` 和 `c` 的作用域则是 `foo()` 和 `bar()` 函数的局部作用域。因此，当在 `bar()` 中访问变量 `a` 时，它可以直接找到全局作用域，但访问变量 `b` 时则需要通过 `foo()` 的作用域链来查找。

Call Stack

在 JavaScript 中，调用栈（Call Stack）是一个存储函数调用的栈结构，它记录了当前执行代码的位置以及代码执行期间调用的所有函数。每当一个函数被调用时，它会被添加到调用栈的顶部。当函数执行完毕后，它将被从调用栈中弹出，并控制权回到前一个函数中。

调用栈的操作可以通过 push 和 pop 实现。每当函数被调用时，它的上下文将被推入调用栈的顶部，当函数返回时，其上下文将从栈中弹出。由于调用栈的大小是有限的，当栈空间耗尽时，会引发“堆栈溢出”错误。

Call Stack 在不同的编程语言和计算机体系结构中可能会有不同的名称和实现方式，例如执行堆栈（Execution Stack）或计算机堆栈（Computer Stack）。

```
var a = 2;
function add (num) {
  var resultlocal = num+ 10;
  return resultlocal;
}
var b = 3;
function sub (num) {
  var resultlocal = num+ 10;
  return resultlocal;
}
let c = 4;
```

简单版

当这段代码执行时，Call Stack 将会按照以下顺序 push 和 pop 函数上下文：

1. 从全局作用域开始，首先 push 全局执行上下文；
2. 执行 var a = 2，将 a 赋值为 2；
3. 执行 function add()，将 add 函数上下文 push 到 Call Stack 中；

4. 执行 var b = 3，将 b 赋值为 3；
5. 执行 function sub()，将 sub 函数上下文 push 到 Call Stack 中；
6. 执行 let c = 4，将 c 赋值为 4；
7. 执行完毕 sub()，将 sub 函数上下文 pop 出 Call Stack；
8. 执行完毕 add()，将 add 函数上下文 pop 出 Call Stack；
9. 最后，pop 出全局执行上下文，Call Stack 变为空。

因此，Call Stack 的 push 和 pop 顺序是：全局执行上下文 -> add 函数上下文 -> sub 函数上下文，pop 顺序与 push 顺序相反。

执行顺序会被hoisting影响

变量和函数的声明会被提升到当前作用域的顶部，这被称为hoisting。所以在执行顺序上，JavaScript 引擎会先扫描当前作用域，把所有的变量和函数声明提升到作用域顶部，然后再执行代码。因此，在一些情况下，执行顺序会受到 hoisting 的影响。

详细版

当这段代码开始执行时，JavaScript 引擎会创建一个全局执行上下文并将其推入调用栈中。全局执行上下文包含了全局变量和函数声明。

然后，当执行到第一行 `var a = 2;` 时，JavaScript 引擎会将变量 `a` 添加到当前执行上下文中。

接下来，当执行到 `function add(num)` 时，JavaScript 引擎会创建一个新的执行上下文并将其推入调用栈顶部。这个新的执行上下文会包含变量 `num` 和 `resultlocal`。然后，当执行到第二行 `var resultlocal = num + 10;` 时，JavaScript 引擎会将变量 `resultlocal` 添加到当前执行上下文中。

接着，当执行到 `return resultlocal;` 时，JavaScript 引擎会将当前执行上下文从调用栈中弹出，并将返回值推入上一个执行上下文的栈顶。在这个例子中，返回值是 `resultlocal`。

然后，当执行到 `var b = 3;` 时，JavaScript 引擎会将变量 `b` 添加到当前执行上下文中。

接着，当执行到 `function sub(num)` 时，JavaScript 引擎会创建一个新的执行上下文并将其推入调用栈顶部。这个新的执行上下文会包含变量 `num` 和 `resultlocal`。然后，当执行到第二行 `var resultlocal = num + 10;` 时，JavaScript 引擎会将变量 `resultlocal` 添加到当前执行上下文中。

接下来，当执行到 `return resultlocal;` 时，JavaScript 引擎会将当前执行上下文从调用栈中弹出，并将返回值推入上一个执行上下文的栈顶。在这个例子中，返回值是 `resultlocal`。

最后，当执行到 `let c = 4;` 时，JavaScript 引擎会将变量 `c` 添加到当前执行上下文中。

当整个代码执行完毕后，JavaScript 引擎会将全局执行上下文从调用栈中弹出。

Syntax

在 JavaScript 中，语法是指编写代码时使用的结构和规则。JavaScript 解释器会根据语法规则来分析代码，并将其转换成可执行的指令。如果代码中存在语法错误，解释器会发出错误提示并停止执行。

语法错误包括拼写错误、不完整的语句、不正确的语句、缺少分号等等。在执行代码之前，JavaScript 引擎会通过解析器对代码进行解析，并尽可能地自动纠正错误。

在代码中，JavaScript 会自动转换类型。例如，在使用 `+` 运算符时，如果其中一个操作数是字符串，则另一个操作数也会自动转换为字符串，从而实现字符串拼接的效果。另外，使用 `prompt()` 函数获取用户输入的结果是字符串类型，如果需要使用整数，则需要使用 `parseInt()` 函数将其转换为整数类型。

在代码中，声明变量可以使用 `var`、`let` 和 `const` 关键字。其中，`var` 是早期的声明变量的方式，在 ES6 中新增了 `let` 和 `const` 关键字。

对于使用 let 和 const 声明的变量，如果没有显式进行初始化，则会默认初始化为 undefined，但是不能直接使用未初始化的变量。

在使用 `var` 声明变量时，如果没有进行初始化，其默认值为 `undefined`，而不会抛出语法错误。但如果在使用该变量之前没有给其赋值，那么其值仍为 `undefined`，如果尝试对其进行操作，可能会导致 `TypeError` 错误。

```
function test1() {
  let x;
  console.log(x);
}
test1();

function test2() {
  const y;
  console.log(y);
}
test2();

function test3() {
  var z;
  console.log(z);
}
test3();
```

在 `test1` 函数中，使用 `let` 声明变量 `x`，但是没有初始化，因此其默认值为 `undefined`。在第二行使用 `console.log` 输出 `x` 的值，由于 `x` 没有被赋值，因此其值为 `undefined`。因此，`test1` 函数输出 `undefined`。

在 `test2` 函数中，使用 `const` 声明变量 `y`，但是没有初始化，这会导致语法错误。因为 `const` 声明的变量必须在声明时进行初始化。因此，`test2` 函数会抛出一个语法错误，无法输出任何值。

在 `test3` 函数中，使用 `var` 声明变量 `z`，但是没有初始化，因此其默认值为 `undefined`。在第二行使用 `console.log` 输出 `z` 的值，由于 `z` 没有被赋值，因此其值为 `undefined`。因此，`test3` 函数输出 `undefined`。

操作符

Comparison operators（比较运算符）用于比较两个值，并返回一个布尔值，常用的比较运算符包括：

- `<`（小于）：如果左边的值小于右边的值，则返回 `true`；否则返回 `false`。
- `<=`（小于等于）：如果左边的值小于或等于右边的值，则返回 `true`；否则返回 `false`。
- `>`（大于）：如果左边的值大于右边的值，则返回 `true`；否则返回 `false`。

- `>=`（大于等于）：如果左边的值大于或等于右边的值，则返回 `true`；否则返回 `false`。
- `==`（等于）：如果两个值相等，则返回 `true`；否则返回 `false`。不同类型的值会进行类型转换后再比较。
- `===`（严格等于）：如果两个值不仅相等，而且类型也相同，则返回 `true`；否则返回 `false`。

Logical operators（逻辑运算符）用于组合两个或多个表达式，并返回一个布尔值，常用的逻辑运算符包括：

- `&&`（逻辑与）：如果两个表达式都为 `true`，则返回 `true`；否则返回 `false`。
- `||`（逻辑或）：如果两个表达式中至少有一个为 `true`，则返回 `true`；否则返回 `false`。
- `!`（逻辑非）：用于取反一个表达式的值。如果表达式的值为 `true`，则返回 `false`；如果表达式的值为 `false`，则返回 `true`。

let, var和const

`var`、`let` 和 `const` 是 JavaScript 中用于声明变量的关键字，它们有以下不同点：

1. 变量作用域：
   - 使用 `var` 声明的变量具有函数级作用域（function scope），即变量在函数内部定义的话，只能在函数内部访问，如果在函数外部定义，就会成为全局变量。
   - 使用 `let` 和 `const` 声明的变量具有块级作用域（block scope），即变量在定义所在的代码块中有效，包括函数、循环、条件语句等，不会污染全局变量。块级作用域内部定义的变量在外部不可访问。
2. 变量声明提升：
   - 使用 `var` 声明的变量会存在变量声明提升（hoisting）的特性，即变量可以在声明之前使用，但是值为 `undefined`。
   - 使用 `let` 和 `const` 声明的变量不会存在变量声明提升的特性，如果在声明之前使用，会抛出 `ReferenceError` 异常。
3. 变量重复声明：
   - 使用 `var` 声明的变量可以重复声明，后声明的变量会覆盖之前声明的变量。
   - 使用 `let` 和 `const` 声明的变量不允许重复声明，如果在同一个作用域内重复声明会抛出 `SyntaxError` 异常。
4. 变量赋值：
   - 使用 `var` 和 `let` 声明的变量可以修改其值。
   - 使用 `const` 声明的变量不能修改其值，它们被视为常量，一旦赋值就不能再更改。
5. 作用域链：
   - 当使用 `var` 声明的变量在函数内部访问时，它们可以通过作用域链（scope chain）访问到更外层的变量。
   - 当使用 `let` 和 `const` 声明的变量在函数内部访问时，它们只能在当前作用域内访问，不能通过作用域链访问到更外层的变量。

综上所述，`var`、`let` 和 `const` 在变量作用域、变量声明提升、变量重复声明、变量赋值和作用域链等方面存在着不同。建议使用 `let` 和 `const` 声明变量，尽量避免使用 `var`。在选择 `let` 和 `const` 时，如果变量的值需要修改，则使用 `let`；如果变量的值不需要修改，则使用 `const`。

Arrays

在 JavaScript 中，数组是一种数据结构，可以用来存储和操作一组有序的数据。数组可以包含各种类型的数据，包括数字、字符串、布尔值和对象等。

创建数组有多种方式。可以使用数组字面量，使用 Array 构造函数，或者使用 Array.from() 静态方法等。以下是几种常见的创建数组的方式：

使用数组字面量：

```
const numbers = [1, 2, 3, 4, 5];
const fruits = ['apple', 'banana', 'orange'];

const arr = [
  "apple",
  2,
  true,
  ["orange", "banana"],
  { name: "John", age: 30 },
  function sayHello() {
    console.log("Hello!");
  }
];
```

使用 Array 构造函数：

```
const numbers = new Array(1, 2, 3, 4, 5);
const fruits = new Array('apple', 'banana', 'orange');
```

使用 Array.from() 静态方法：

```
const numbers = Array.from([1, 2, 3, 4, 5]);
const fruits = Array.from(['apple', 'banana', 'orange']);
```

可以使用数组的下标来访问和修改数组中的元素。数组的下标从0开始，最大值为数组长度减1。

在 JavaScript 中，数组对象有许多方法，包括以下一些常用的方法：

1. push(): 将一个或多个元素添加到数组的末尾，并返回新数组的长度。通过对象调用。
2. pop(): 移除数组的最后一个元素，并返回该元素的值。通过对象调用。
3. unshift(): 将一个或多个元素添加到数组的开头，并返回新数组的长度。通过对象调用。
4. shift(): 移除数组的第一个元素，并返回该元素的值。通过对象调用。
5. splice(): 可以添加、删除和替换数组中的元素。通过对象调用。
6. slice(): 返回一个新数组，包含从开始到结束（不包含结束）的所有元素。通过对象调用。

7. concat(): 将两个或多个数组合并成一个新数组。通过对象调用。
8. join(): 将数组中的所有元素连接成一个字符串，并返回该字符串。通过对象调用。
9. reverse(): 反转数组中的元素顺序。通过对象调用。
10. sort(): 对数组中的元素进行排序。通过对象调用。
11. filter(): 返回一个新数组，其中包含符合条件的数组元素。通过对象调用。
12. map(): 返回一个新数组，其中包含调用函数处理过的每个元素。通过对象调用。
13. forEach(): 对数组中的每个元素执行指定操作。通过对象调用。
14. reduce(): 对数组中的元素进行累加或累计操作，返回计算结果。通过对象调用。
15. find(): 返回数组中第一个符合条件的元素。通过对象调用。
16. findIndex(): 返回数组中第一个符合条件的元素的索引。通过对象调用。
17. indexOf(): 返回数组中第一个匹配项的索引。通过对象调用。
18. lastIndexOf(): 返回数组中最后一个匹配项的索引。通过对象调用。
19. includes(): 判断数组中是否包含指定元素。通过对象调用。
20. fill(): 用指定值填充数组中的元素。通过对象调用。
21. keys(): 返回一个包含数组中所有键的迭代器。通过对象调用。
22. values(): 返回一个包含数组中所有值的迭代器。通过对象调用。
23. entries(): 返回一个包含数组中所有键值对的迭代器。通过对象调用。

除了数组对象的方法，JavaScript 还有一些内置数组方法，例如：

1. Array.from(): 将一个类数组对象或可迭代对象转换为一个新的数组对象。
2. Array.isArray(): 判断一个对象是否为数组对象。
3. Array.of(): 创建一个由指定参数组成的新数组对象。

这些方法可以通过 Arrays.方法() 的方式调用。

例如：

```
const numbers = [1, 2, 3];
numbers.push(4);
console.log(numbers); // [1, 2, 3, 4]

const fruits = ['apple', 'banana', 'orange'];
fruits.splice(1, 1, 'pear');
console.log(fruits); // ['apple', 'pear', 'orange']
```

除了以上常用的方法，数组还有很多其他的方法，如 forEach()、map()、filter()、reduce() 等，可以用来遍历和操作数组中的元素。

方括号和大括号

在 JavaScript 中，方括号通常用于创建和访问数组，而大括号则用于创建和访问对象。

使用方括号创建一个数组时，可以在方括号中指定数组的元素，用逗号分隔。例如：

```
let arr = [1, 2, 3, 4, 5];
```

要访问数组的元素，可以使用方括号和元素的索引值，例如：

```
console.log(arr[0]); // 输出 1
```

使用大括号创建一个对象时，需要在大括号中指定对象的属性和属性值，用冒号分隔。例如：

```
let obj = {name: 'Alice', age: 25, gender: 'female'};
```

要访问对象的属性，可以使用点运算符或方括号，例如：

```
console.log(obj.name); // 输出 'Alice'
console.log(obj['age']); // 输出 25
```

需要注意的是，如果属性名包含特殊字符或数字开头，只能使用方括号访问该属性，例如：

```
let obj = {'first-name': 'Alice', 'last-name': 'Smith', '123': '456'};
console.log(obj['first-name']); // 输出 'Alice'
console.log(obj['last-name']); // 输出 'Smith'
console.log(obj['123']); // 输出 '456'
```

For loop

`for...in` 和 `for...of` 都是 JavaScript 中的循环语句，但它们的作用有所不同。

`for...in` 语句用于枚举对象的属性，它遍历对象的可枚举属性，并将属性名赋给循环变量。通常用于遍历对象，不适合遍历数组，因为它会遍历对象自身和原型链上的所有属性，而数组的索引被视为对象的属性，所以 `for...in` 也会遍历数组的非数字属性，造成一定的性能损失。

示例代码：

```
const obj = { a: 1, b: 2, c: 3 };

for (let prop in obj) {
  console.log(prop, obj[prop]);
}
// 输出结果：a 1, b 2, c 3

const arr = [1, 2, 3];
for (let index in arr) {
  console.log(index, arr[index]);
}
// 输出结果：0 1, 1 2, 2 3
```

`for...of` 语句用于遍历可迭代对象，如数组、字符串、Map、Set 等等。它遍历可迭代对象的元素，并将元素值赋给循环变量。通常用于遍历数组等可迭代对象。

示例代码：

```
const arr = [1, 2, 3];

for (let item of arr) {
  console.log(item);
}
// 输出结果：1, 2, 3
```

需要注意的是，`for...of` 语句只能遍历可迭代对象的元素，不能遍历对象的属性。如果要遍历对象的属性，仍然需要使用 `for...in` 语句。

`for-in` 循环是用于对象的属性遍历，而不是数组的下标遍历。在循环过程中，变量会被赋值为属性的名称，而不是索引，因此输出的是属性名而不是索引。对于数组来说，属性名实际上就是数组的索引。因此在 `for-in` 循环中，输出的是数组元素的索引，而不是元素本身。

```
var example = [1, 'joe', 5.555];
for (var element in example) {
  console.log(element);
}
//输出 0 1 2
for (var element of example) {
  console.log(element);
}
//输出 1 joe 5.555
```

数组、Set、Map、字符串才可以使用for...of

Anonymous Functions

Anonymous Functions 是指没有具体名称的函数，通常使用匿名函数时，是将该函数作为另一个函数的参数或者是在自调用函数中使用。在 JavaScript 中，我们可以使用匿名函数来实现很多功能，比如在数组的 map() 方法中，就经常使用匿名函数。

举例来说，假设我们有一个数组 age，它包含了一组年龄数据 [20, 25, 30, 35]。现在我们需要将每个元素的值加上 10，我们可以使用 map() 方法和匿名函数来实现：

```
var age = [20, 25, 30, 35];
age = age.map(function(num) {
  return num + 10;
});
console.log(age); // 输出 [30, 35, 40, 45]
```

在这个例子中，我们使用了匿名函数来对数组的每个元素进行操作，匿名函数的参数 num 代表了数组中的每个元素，然后将其加上 10 后返回，最后使用 map() 方法将修改后的新数组赋值给 age 变量。

使用匿名函数可以让代码更加简洁，特别是当我们需要对一个数组中的所有元素都进行相同的操作时，可以将这个操作封装到匿名函数中，然后使用 map()、forEach() 等方法来执行。

Arrow Functions

Arrow Functions是ES6中新增的一种函数声明方式，它可以用更简洁的语法来定义函数。相比于普通函数，Arrow Functions有以下几个特点：

1. 箭头函数没有自己的this值，它们的this值继承自外层作用域中的this。
2. 箭头函数没有arguments对象，可以使用Rest Parameters代替。
3. 箭头函数不能用作构造函数。
4. 箭头函数语法更简洁，适用于一些只需要一行代码的函数。

下面是一个使用Arrow Functions的例子，将数组中的每个元素加上10并返回新的数组：

```
const arr = [1, 2, 3, 4];
const newArr = arr.map(item => item + 10);
console.log(newArr); // [11, 12, 13, 14]

result1 => {....} //等价于一个函数 function a(result1) {...}
```

在这个例子中，我们使用了箭头函数来定义map()方法的回调函数。箭头函数用`=>`符号表示，它的参数是一个item，箭头后面的表达式就是函数的返回值，这里我们将每个元素加上10并返回新的数组。

this值

在 JavaScript 中，this 关键字通常指向当前函数的执行上下文，即指向调用该函数的对象。而在箭头函数中，this 的指向与普通函数有所不同，它指向了定义该箭头函数的上下文。

在 React 组件中，通常情况下，this 指向组件实例对象。下面举几个例子：

1. 在类组件的构造函数中，this 指向当前组件实例对象：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    console.log(this); // MyComponent { props: {...}, state: {...}, ... }
  }
  // ...
}
```

2. 在事件处理函数中，this 指向当前组件实例对象：

```
class MyComponent extends React.Component {
  handleClick() {
    console.log(this); // MyComponent { props: {...}, state: {...}, ... }
  }
  render() {
    return (
      <button onClick={this.handleClick}>Click me</button>
    );
  }
}
```

3. 在 componentDidMount() 函数中，this 指向当前组件实例对象：

```
class MyComponent extends React.Component {
  componentDidMount() {
    console.log(this); // MyComponent { props: {...}, state: {...}, ... }
  }
  // ...
}
```

在这些情况下，使用箭头函数可以确保 this 指向组件实例对象，避免了使用 bind 方法或函数包装的麻烦。

: 或 =

在 JavaScript 中，箭头函数有两种语法形式：

1. 不带参数列表的箭头函数：

```
() => { ... }
```

2. 带参数列表的箭头函数：

```
(arg1, arg2, ...) => { ... }
```

在使用带参数列表的箭头函数时，可以选择在箭头函数前使用 `=` 或 `:` 来定义函数。这两种写法的效果是相同的，都是将箭头函数赋值给一个变量。

例如：

```
const func1 = (arg1, arg2) => { ... };
const func2: (arg1, arg2) => { ... };
const func3 = function(arg1, arg2) { ... };
```

其中，`func1` 和 `func2` 都是箭头函数，使用了不同的定义方式；`func3` 是普通的函数表达式。无论使用哪种方式定义函数，都可以通过变量名来调用函数。

在使用箭头函数时，可以选择在函数名前面使用冒号（:）或等号（=）。这两种方式的作用是类似的，都是用于声明一个函数。区别在于使用等号时，通常将函数赋值给一个变量，而使用冒号时，通常将函数作为对象的属性来定义。
例如，以下两种方式都定义了一个名为myFunction的箭头函数：

```
const myFunction = () => {
  // function body
}

const obj = {
  myFunction: () => {
    // function body
  }
}
```

第一种方式使用等号，将箭头函数赋值给了一个变量。第二种方式使用冒号，将箭头函数作为一个对象obj的属性来定义。

lambda和箭头函数

Lambda表达式和箭头函数有很多相似之处，但它们最大的不同在于它们的语法和使用场景。

Lambda表达式是一种函数式编程概念，它在很多编程语言中都有应用，比如Java和C#等。在JavaScript中，Lambda表达式可以通过箭头函数来实现。

箭头函数是一种简写函数表达式的语法，它可以让函数定义更加简洁，同时也改变了函数内部this的指向。

Lambda表达式在JavaScript中通常称为箭头函数，是一种特殊类型的函数表达式。与普通函数表达式不同，箭头函数使用 => 连接参数和函数体。

```
let add = function(x, y) {
  return x + y;
}
```

而箭头函数使用箭头符号 `=>` 来定义函数，比如：

```
let add = (x, y) => {
  return x + y;
}
```

Events

在网页上发生的所有操作都是事件。比如点击按钮、输入文本、悬停鼠标等。HTML 页面可以捕获这些事件，并将它们传递给 JavaScript 中的事件处理程序。通过事件处理程序，我们可以对这些事件做出响应，例如执行某个操作、修改页面内容等。

举个例子，比如在网页上有一个按钮，我们可以在 HTML 中为该按钮添加一个事件监听器，监听用户的点击事件。当用户点击按钮时，事件就会被捕获，并传递给 JavaScript 中的事件处理程序。事件处理程序可以根据具体情况执行一些操作，例如弹出一个提示框、修改页面内容等。

js可以通过事件对象（Event object）来传递值。当事件被触发时，浏览器会创建一个事件对象并传递给事件处理函数。事件对象包含了事件的相关信息，例如触发事件的元素、事件类型等等。事件处理函数可以通过访问事件对象来获取这些信息。

举个例子，假设有一个按钮元素，当用户点击该按钮时，需要获取按钮的文本内容。可以通过添加一个点击事件监听器来实现：

```
<button id="myButton">Click me!</button>
javascriptCopy codeconst myButton = document.getElementById('myButton');

myButton.addEventListener('click', function(event) {
  const buttonText = event.target.textContent;
  console.log(buttonText); // 输出 "Click me!"
});
```

在上面的代码中，`addEventListener()` 方法用来添加一个点击事件监听器。当按钮被点击时，会调用回调函数，并传递一个事件对象 `event`。通过访问事件对象的 `target` 属性，可以获取触发事件的元素，也就是按钮本身。然后通过访问该元素的 `textContent` 属性，可以获取按钮的文本内容。最后将文本内容输出到控制台中。

typeof()

`typeof` 是一个 JavaScript 运算符，用于检测一个值的数据类型。下面是不同类型变量在 `typeof` 运算符下的输出：

- `typeof undefined` 会输出 `"undefined"`，表示未定义的值。
- `typeof null` 会输出 `"object"`，这是一个历史遗留问题，实际上 `null` 是一个原始值而不是对象。
- `typeof true` 和 `typeof false` 分别会输出 `"boolean"`，表示布尔类型。
- `typeof 42` 会输出 `"number"`，表示数值类型。
- `typeof 'JavaScript'` 会输出 `"string"`，表示字符串类型。
- `typeof Symbol('foo')` 会输出 `"symbol"`，表示符号类型。
- `typeof {}` 和 `typeof []` 分别会输出 `"object"`，表示对象类型。
- `typeof function() {}` 会输出 `"function"`，表示函数类型。

下面是一些示例：

```
console.log(typeof undefined); // 输出： "undefined"
console.log(typeof null); // 输出： "object"
console.log(typeof true); // 输出： "boolean"
console.log(typeof 42); // 输出： "number"
console.log(typeof 'JavaScript'); // 输出： "string"
console.log(typeof Symbol('foo')); // 输出： "symbol"
console.log(typeof {}); // 输出： "object"
console.log(typeof []); // 输出： "object"
console.log(typeof function() {}); // 输出： "function"
```

需要注意的是，`typeof` 运算符并不总是能够准确地检测值的数据类型，特别是在处理对象类型时。在这种情况下，`typeof` 运算符只能识别出基本的对象类型（包括对象、数组、函数等），而不能识别出更具体的对象类型。例如，`typeof []` 和 `typeof {}` 都会输出 `"object"`，无法区分它们的具体类型。

在 `typeof` 运算符中使用变量名时，它会返回该变量的数据类型。如果变量未定义，则 `typeof` 运算符返回 `"undefined"`。下面是在 `var`、`let` 和 `const` 关键字下声明变量时，`typeof` 运算符输出的结果：

```
console.log(typeof x); // 输出： "undefined"

var x;
console.log(typeof x); // 输出： "undefined"

let x;
console.log(typeof x); // 输出： "undefined"

const x;
console.log(typeof x); // 报错： Missing initializer in const declaration
```

在这个例子中，我们分别使用 `var`、`let` 和 `const` 关键字声明了变量 `x`，并在 `typeof` 运算符中使用了该变量名。在所有情况下，`typeof x` 都会输出 `"undefined"`，因为这些变量被声明但未被初始化，因此其值为 `undefined`。

需要注意的是，当我们在使用 `const` 关键字声明变量时，必须同时对其进行初始化。如果我们在 `const` 声明语句中省略了初始化器，就会导致语法错误，代码无法执行。

NaN

`NaN` 是 JavaScript 中的一个特殊值，表示“不是数字”（Not a Number）。它通常在以下情况下输出：

1. 对于数学运算中的错误或未定义的操作，例如将非数值型的值与数值相加或相除，或对负数执行平方根或自然对数运算。

   ```
   console.log('abc' / 2); // 输出： NaN
   console.log(Math.sqrt(-1)); // 输出： NaN
   console.log(Math.log(-1)); // 输出： NaN
   ```

2. 在执行字符串转换为数值的操作时，如果字符串无法被解析为数字，就会输出 `NaN`。例如：

```
console.log(parseInt('abc')); // 输出： NaN
console.log(parseFloat('def')); // 输出： NaN
```

3. `NaN` 与任何其他值进行比较，包括自身，结果都是 `false`。例如：

```
console.log(NaN == NaN); // 输出： false
console.log(NaN === NaN); // 输出： false
```

需要注意的是，虽然 `NaN` 表示“不是数字”，但它实际上是一个数值类型的值。可以通过 `typeof` 运算符来检测其数据类型，输出结果为 `"number"`。

Propmt

`prompt()` 是 JavaScript 中的一个内置函数，它会弹出一个包含输入框的对话框，允许用户输入一些内容并点击确定或取消按钮。该函数通常用于获取用户输入数据，例如获取用户名、密码等。

`prompt()` 函数接受一个参数，表示将显示在对话框中的文本。该参数是可选的，如果未提供，则默认显示一个空字符串。函数返回一个字符串，表示用户在对话框中输入的内容。如果用户单击了取消按钮，则该函数返回 `null`。

`prompt()` 函数有两个参数，第一个参数是弹出窗口中显示的文本字符串，第二个参数是可选的默认输入字符串。如果用户在弹出窗口中输入了字符串并单击了“确定”按钮，则 `prompt()` 函数返回该字符串。如果用户单击了“取消”按钮，则返回 `null`。

默认输入字符串是可以被删掉的。用户可以直接删除默认输入字符串，然后输入自己的内容。

如果用户没有输入内容或删除默认字符串，点击确定，则会返回空字符串。如果点击取消，则返回 null。

Elements

在 JavaScript 中，事件是与浏览器交互的重要方式之一，通过事件我们可以在文档结构中定义特定的交互行为。事件是 DOM 的一部分，其目的是在发生某些事情时执行特定的代码。

事件的类型有很多种，比如鼠标事件、键盘事件、表单事件等。事件处理程序则是被绑定到 HTML 元素上的函数，当某个事件发生时就会被执行。

以下是一些常见的事件类型：

- 鼠标事件：click、dblclick、mousedown、mouseup、mousemove、mouseover、mouseout、mouseenter、mouseleave 等。
- 键盘事件：keydown、keyup、keypress 等。
- 表单事件：submit、reset、focus、blur、change 等。
- 文档事件：DOMContentLoaded、readystatechange、load、unload 等。

JavaScript 通过事件监听器来处理事件。事件监听器是指通过 JavaScript 绑定到元素的一个函数，当该元素发生指定的事件时，该函数就会被执行。有两种主要的方式可以添加事件监听器：HTML 属性和 DOM 事件处理程序。

HTML 属性是在 HTML 元素上直接添加的属性，其中包含了 JavaScript 代码，以实现与该元素关联的事件处理程序。例如：

```
<button onclick="alert('Hello world!')">Click me</button>
```

这里将 onclick 属性设置为一个 JavaScript 函数，当用户单击按钮时，就会调用该函数并显示一个弹窗。

DOM 事件处理程序则是通过 JavaScript 代码添加到 HTML 元素上的事件处理程序。这种方式的优点在于可以将 JavaScript 代码与 HTML 元素分离，提高代码的可维护性和可重用性。例如：

```
<button id="myBtn">Click me</button>
const btn = document.getElementById('myBtn');
btn.addEventListener('click', function() {
  alert('Hello world!');
});
```

这里使用 addEventListener 方法将一个事件处理程序添加到按钮元素上，当用户单击按钮时，就会调用该处理程序并显示一个弹窗。

除了上面的方式之外，还有一些其他的方式可以处理事件，比如使用 jQuery 或 React 等库来添加事件监听器。无论使用哪种方式，事件处理程序都是 JavaScript 中处理事件的核心机制之一。

Json

```
{
  "name": "John",
  "age": 30,
  "city": "New York",
  "hobbies": ["reading", "swimming", "traveling"],
  "isMarried": true,
  "education": {
    "degree": "Bachelor's",
    "major": "Computer Science"
  }
}
```

js对象 vs. json

JavaScript 对象和 JSON 对象都可以用来表示复杂数据结构，但它们的格式有所不同。下面是 JavaScript 对象和 JSON 对象的格式对比以及示例：

JavaScript 对象格式：

```
let person = {
  name: "John",
  age: 30,
  hobbies: ["reading", "running"],
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "CA",
    zip: "12345"
  }
};
```

JSON 对象格式：

```
{
  "name": "John",
  "age": 30,
  "hobbies": ["reading", "running"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

JavaScript 对象使用花括号 `{}` 括起来，包含一个或多个键值对，每个键值对用冒号 `:` 分隔。键值对之间用逗号 `,` 分隔。键是字符串，也可以是数字或其他类型，值可以是任何 JavaScript 数据类型，包括对象、数组、字符串、数字、布尔值、函数等。

在 JavaScript 对象中，属性名可以是字符串或标识符。如果属性名是标识符，可以直接写成变量名的形式。例如：

```
const person = {
  name: "Alice",
  age: 25
};

console.log(person.name); // 输出 "Alice"
console.log(person.age); // 输出 25

const propertyName = "age";
console.log(person[propertyName]); // 输出 25
```

在上面的例子中，`name` 和 `age` 都是标识符，它们可以直接用作属性名。我们可以通过点符号或中括号来访问这些属性。当属性名是变量时，需要使用中括号来访问对象的属性。

如果属性名不是标识符，例如包含空格或中文字符的字符串，就必须将其用引号括起来。在这种情况下，我们只能使用中括号来访问属性，不能使用点符号。例如：

```
const person = {
  "full name": "Alice Smith",
  "年龄": 25
};

console.log(person["full name"]); // 输出 "Alice Smith"
console.log(person["年龄"]); // 输出 25
```

在上面的例子中，属性名包含了空格和中文字符，因此必须使用引号将其括起来。我们只能通过中括号来访问这些属性。

JSON 对象使用花括号 `{}` 括起来，包含一个或多个键值对，每个键值对用冒号 `:` 分隔。键和值都必须是字符串，并且字符串必须使用双引号 `"` 包括。键值对之间用逗号 `,` 分隔。值可以是任何 JSON 数据类型，包括对象、数组、字符串、数字、布尔值和 `null`。

在 JSON 中，所有的键名都必须使用双引号 `"` 包裹，而且值也必须使用双引号、单引号或者不使用引号包裹。具体来说，**以下三种类型的值不需要使用双引号**：

1. 数字（包括整数和浮点数）；
2. 布尔值（true 和 false）；
3. null 值。

下面是一个示例 JSON 对象，其中包含了数字、布尔值和 null 值：

```
{
  "name": "Alice",
  "age": 25,
  "isStudent": true,
  "score": 92.5,
  "languages": ["JavaScript", "Python", "Java"],
  "address": null
}
```

需要注意的是，虽然 JSON 不需要使用分号 `;` 分隔每个键值对，但是建议在每个键值对之后加上逗号，以增加代码的可读性。

JavaScript 对象可以直接在 JavaScript 代码中使用，而 JSON 对象则通常用于在不同平台之间传递数据，例如通过 AJAX 请求从服务器获取数据。

下面是一个将 JavaScript 对象转换为 JSON 格式的示例：

```
let person = {
  name: "John",
  age: 30,
  hobbies: ["reading", "running"],
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "CA",
    zip: "12345"
  }
```

```
};

let json = JSON.stringify(person);
console.log(json); // 输出：{"name":"John","age":30,"hobbies":
["reading","running"],"address":{"street":"123 Main
St","city":"Anytown","state":"CA","zip":"12345"}}
```

在这个例子中，我们使用 `JSON.stringify()` 方法将 JavaScript 对象 `person` 转换为 JSON 格式的字符串。需要注意的是，JSON 字符串中的所有键都必须使用双引号 `"` 包括，因为这是 JSON 格式的规定。

js 与json互转

可以使用 JSON.stringify() 方法将 JavaScript 对象转换为 JSON 字符串。该方法接受一个 JavaScript 对象，然后将其序列化为一个 JSON 字符串。例如：

```
const data = { name: 'Alice', age: 28 };
const jsonString = JSON.stringify(data);
console.log(jsonString); // 输出 {"name":"Alice","age":28}
```

如果需要将一个 JSON 字符串转换为一个 JavaScript 对象，可以使用 JSON.parse() 方法。该方法接受一个 JSON 字符串，然后将其解析为一个 JavaScript 对象。例如：

```
const jsonString = '{"name":"Alice","age":28}';
const data = JSON.parse(jsonString);
console.log(data); // 输出 { name: 'Alice', age: 28 }
```

xml vs. json

JSON（JavaScript Object Notation）和 XML（eXtensible Markup Language）都是数据交换格式。下面是它们的一些主要区别和优劣：

1. 语法结构不同：JSON使用JavaScript对象和数组语法，而XML使用自定义标签和属性。
2. 传输效率不同：JSON更加简洁轻便，传输速度更快。XML则相对冗长，传输速度较慢。
3. 解析和操作方式不同：JSON使用内置的JSON.parse()和JSON.stringify()方法解析和操作，而XML需要使用专门的解析器和库进行操作。
4. 可读性不同：JSON相对于XML来说更加易读易懂，因为它使用JavaScript的语法，更加直观。
5. 扩展性不同：XML相对于JSON来说更具扩展性，因为XML标签和属性可以自定义，而JSON则需要在结构中进行预定义。
6. 应用场景不同：JSON通常用于Web应用程序中，因为它更加轻便易用，而XML则更加适合用于复杂的数据存储和数据交换。

综上，JSON更适合在轻量级的Web应用程序中使用，而XML则更适合在大型、复杂的数据交换和存储中使用。

Template literals

在 JavaScript 中，模板字符串是一种特殊的字符串字面量，使用**反引号**包裹起来，可以在其中嵌入变量或表达式。模板字符串使用 `${}` 语法来插入变量或表达式，使用 `${expression}` 来代替传统的字符串拼接方式。模板字符串可以包含多行文本，不需要使用转义字符或连接符。例如：

```
const name = 'Alice';
const age = 25;
const message = `My name is ${name} and I am ${age} years old.`;
console.log(message); // 输出 "My name is Alice and I am 25 years old."
```

在这个例子中，我们使用了模板字符串来创建一个包含变量的字符串。`${name}` 会被替换成 `Alice`，`${age}` 会被替换成 `25`，最终得到一个完整的字符串。

如果字符串中包含了变量、表达式、函数调用等需要计算的部分，就可以使用 `${}` 将它们嵌入到字符串中。需要注意的是，在 `${}` 中只能包含表达式，而不能包含语句。

如果将 JavaScript 对象放到模板字符串中，则会将该对象的 toString() 方法的返回值作为字符串输出。如果该对象没有重写 toString() 方法，则输出默认的 "[object Object]"。如果想要输出 JavaScript 对象的属性值，可以在模板字符串中使用对象的属性名，使用 `${}` 包裹起来

假设有一个对象 `person`，包含 `name` 和 `age` 两个属性：

```
const person = { name: 'Alice', age: 25 };
```

如果将该对象直接放到模板字符串中输出，会得到以下结果：

```
console.log(`${person}`); // [object Object]
```

因为默认情况下，将对象放入模板字符串中会调用对象的 `toString()` 方法，而 `toString()` 方法返回的是一个类似于 `"[object Object]"` 的字符串。如果要输出对象的属性，可以使用模板字符串的变量替换功能，例如：

```
console.log(`Name: ${person.name}, Age: ${person.age}`); // Name: Alice, Age: 25
```

这样就可以输出对象的属性值了。

Object Oriented Paradigm

JavaScript 支持面向对象编程（Object Oriented Paradigm），主要是通过使用对象（Objects）和原型（Prototypes）来实现的。

JavaScript 中的对象是一组由键值对构成的无序集合。每个键（key）都是一个字符串，每个值（value）可以是任意类型的数据，包括其他对象。对象可以通过“属性（property）”和“方法（method）”来操作和访问其数据。

JavaScript 中的原型（Prototype）是一种用于实现继承的机制。每个 JavaScript 对象都有一个原型对象，它定义了对象的默认属性和方法。当我们访问一个对象的属性或方法时，如果该对象本身没有该属性或方法，JavaScript 引擎会沿着该对象的原型链（Prototype Chain）向上查找，直到找到该属性或方法或到达原型链的顶端。这个过程就是继承。

JavaScript 中的类（Class）是基于原型机制实现的，它们被称为构造函数（Constructor）。构造函数可以用来创建对象的实例，每个实例都继承了该构造函数的原型链。类中的"静态方法（static method）"和"实例方法（instance method）"可以被用来操作类和实例的数据。

相比于传统的面向对象编程语言，JavaScript 的面向对象编程模型有一些独特的特点和优势：

1. 基于原型的继承模型更加灵活，可以实现更加复杂的对象关系和组合。
2. JavaScript 中的对象和函数可以相互转换，这为面向对象和函数式编程提供了更多的选择和灵活性。
3. JavaScript 语言本身的灵活性和动态性，使得它适合于快速原型开发和构建动态、交互性的 Web 应用程序。

class

在 JavaScript 中，类是通过构造函数（constructor）来实现的。构造函数是一个特殊的方法，用于创建并初始化类的对象。在创建一个新对象时，构造函数被调用，将值和属性分配给该对象，从而初始化该对象。实例化一个类会创建一个新的对象，该对象拥有该类定义的所有属性和方法。

与其他语言不同的是，JavaScript 中的类并不是预定义的，而是使用函数和对象来创建的。在函数中定义属性和方法，使用 `this` 关键字来引用当前对象的属性和方法。通过使用 `new` 关键字来实例化类，即创建一个新的对象，该对象拥有该类定义的所有属性和方法。

在 JavaScript 中，类并不存在于内存中，而是通过构造函数创建的实例对象才存在于内存中。因此，在 JavaScript 中创建类的方式不同于传统的面向对象编程语言，但它仍然具有面向对象编程的核心思想，即封装、继承和多态性。

举个例子，下面是一个简单的构造函数和使用它创建对象的例子：

```
// 构造函数
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// 使用构造函数创建对象
const person1 = new Person('Alice', 25);
const person2 = new Person('Bob', 30);

console.log(person1); // 输出：Person { name: 'Alice', age: 25 }
console.log(person2); // 输出：Person { name: 'Bob', age: 30 }
```

在上面的例子中，我们定义了一个名为 Person 的构造函数，它接受两个参数：name 和 age。构造函数使用 this 关键字将这些参数存储为新对象的属性。我们随后使用 new 关键字来创建两个 Person 对象，并将它们存储在变量 person1 和 person2 中。

通过这种方式，我们可以使用构造函数来创建任意数量的对象，每个对象都有自己的属性和值。这是 JavaScript 中面向对象编程的一种基本方式。

Object对象

JavaScript 中对象是轻量级的，可以通过字面量（literals）创建对象，也可以使用构造函数创建对象。不需要使用类（class）就可以创建对象。字面量创建对象的方式如下：

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA'
  }
};
```

函数构造器创建对象的方式如下：

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}

const person = new Person('John', 'Doe', 30);
```

常用方法

根据JavaScript 对象类型不同，可调用方法也不同：

1. Object：JavaScript 的基本对象类型，用于表示一般对象。 常用方法：Object.keys()、Object.values()、Object.entries()、Object.assign()、Object.create()、Object.defineProperty()、Object.defineProperties()、Object.getOwnPropertyDescriptor()、Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()、Object.getPrototypeOf()、Object.is()、Object.freeze()、Object.seal()、Object.preventExtensions()、Object.setPrototypeOf() 等。
2. Array：用于表示数组。 常用方法：push()、pop()、shift()、unshift()、concat()、slice()、splice()、sort()、reverse()、join()、indexOf()、lastIndexOf()、filter()、map()、reduce()、forEach() 等。
3. String：用于表示字符串。 常用方法：charAt()、charCodeAt()、concat()、indexOf()、lastIndexOf()、match()、replace()、search()、slice()、split()、substr()、substring()、toLowerCase()、toUpperCase() 等。
4. Number：用于表示数字。 常用方法：toFixed()、toPrecision()、toString()、valueOf() 等。
5. Boolean：用于表示布尔值。 常用方法：toString()、valueOf() 等。
6. Date：用于表示日期和时间。 常用方法：getDate()、getDay()、getFullYear()、getHours()、getMilliseconds()、getMinutes()、getMonth()、getSeconds()、getTime()、setDate()、setFullYear()、setHours()、setMilliseconds()、setMinutes()、setMonth()、setSeconds()、setTime() 等。

7. RegExp：用于表示正则表达式。 常用方法：exec()、test() 等。
8. Function：用于表示函数。 常用方法：apply()、call()、bind() 等。
9. Math：用于执行数学运算。 常用方法：abs()、ceil()、floor()、max()、min()、pow()、random()、round()、sqrt()、trunc() 等。

需要注意的是，不同的对象类型可能会有一些共用的方法，如 toString()、valueOf() 等，也可能有各自独有的方法，具体可以参考相应对象类型的文档。

以下是 JavaScript 对象常用的方法及作用：

obj.方法名() 其中 obj 是要检查的对象

- obj.hasOwnProperty(prop)：判断对象自身是否具有指定属性（不包括原型链上的属性）。

```
const person = { name: 'Alice', age: 30 };
console.log(person.hasOwnProperty('name')); // true
console.log(person.hasOwnProperty('toString')); // false
```

1. obj.toString()：返回对象的字符串表示。

```
const person = { name: 'Alice', age: 30 };
console.log(person.toString()); // [object Object]
```

2. obj.valueOf()：返回对象的原始值表示。

```
const num = new Number(10);
console.log(num.valueOf()); // 10
```

3. obj.hasOwnProperty()：判断对象是否拥有指定属性，不考虑原型链上的属性。

```
const person = { name: 'Alice', age: 30 };
console.log(person.hasOwnProperty('name')); // true
console.log(person.hasOwnProperty('toString')); // false
```

4. obj.isPrototypeOf(obj)：判断当前对象是否是指定对象的原型。

```
function Person(name) {
  this.name = name;
}

const person = new Person('Alice');
console.log(Person.prototype.isPrototypeOf(person)); // true
```

5. obj.propertyIsEnumerable(prop)：判断对象自身是否具有指定属性且该属性是否可枚举。

```
const person = { name: 'Alice', age: 30 };
console.log(person.propertyIsEnumerable('name')); // true
console.log(person.propertyIsEnumerable('toString')); // false
```

6. obj.toLocaleString()：返回对象的本地化字符串表示。

```
const num = new Number(10);
console.log(num.toLocaleString()); // "10"
```

7. obj.valueOf()：返回对象的原始值表示。

```
const num = new Number(10);
console.log(num.valueOf()); // 10
```

Object.方法名()

- `Object.assign(target, source1, source2, ...)`：将一个或多个对象的属性复制到目标对象中。

  ```
  const target = { a: 1, b: 2 };
  const source1 = { b: 3, c: 4 };
  const source2 = { c: 5, d: 6 };
  const result = Object.assign(target, source1, source2);
  console.log(result); // { a: 1, b: 3, c: 5, d: 6 }
  ```

- `Object.create(proto, [propertiesObject])`：创建一个新对象，使用现有对象作为原型。

  ```
  const person = { name: 'Alice' };
  const employee = Object.create(person, {
    salary: { value: 50000 },
  });
  console.log(employee.name); // 'Alice'
  console.log(employee.salary); // 50000
  ```

- `Object.defineProperty(obj, prop, descriptor)`：定义或修改对象的属性。

  ```
  const person = {};
  Object.defineProperty(person, 'name', {
    value: 'Alice',
    writable: false,
    enumerable: true,
    configurable: true,
  });
  console.log(person.name); // 'Alice'
  person.name = 'Bob'; // throws an error in strict mode
  ```

- `Object.defineProperties(obj, props)`：定义或修改对象的多个属性。

```
const person = {};
Object.defineProperties(person, {
  name: { value: 'Alice' },
  age: { value: 25 },
});
console.log(person.name); // 'Alice'
console.log(person.age); // 25
```

- `Object.entries(obj)`：返回一个给定对象自身可枚举属性的键值对数组，即一个由数组组成的数组。

```
const person = { name: 'Alice', age: 25 };
const entries = Object.entries(person);
console.log(entries); // [['name', 'Alice'], ['age', 25]]
```

- `Object.freeze(obj)`：冻结对象，使其属性不能被修改、添加或删除。

```
const person = { name: 'Alice' };
Object.freeze(person);
person.name = 'Bob'; // fails silently in non-strict mode
console.log(person.name); // 'Alice'
```

- `Object.getOwnPropertyDescriptor(obj, prop)`：获取对象的属性描述符。

```
const person = {};
Object.defineProperty(person, 'name', {
  value: 'Alice',
  writable: false,
});
const descriptor = Object.getOwnPropertyDescriptor(person, 'name');
console.log(descriptor);
// { value: 'Alice', writable: false, enumerable: false, configurable: false }
```

- `Object.getOwnPropertyNames(obj)`：返回对象自身的所有属性的名称。

```
const person = { name: 'Alice', age: 25 };
const properties = Object.getOwnPropertyNames(person);
console.log(properties); // ['name', 'age']
```

- `Object.getOwnPropertySymbols(obj)`：返回对象自身所有的 Symbol 类型属性的数组。

```
const key = Symbol('key');
const person = { [key]: 'value' };
const symbols = Object.getOwnPropertySymbols(person);
console.log(symbols); // [Symbol(key)]
```

- `Object.getPrototypeOf(obj)`：获取对象的原型。

```
const person = { name: 'John' };
const prototypeOfPerson = Object.getPrototypeOf(person);
console.log(prototypeOfPerson); // Output: {}
```

- `Object.is(obj1, obj2)`：判断两个值是否严格相等。

```
console.log(Object.is(5, 5)); // Output: true
console.log(Object.is('hello', 'hello')); // Output: true
console.log(Object.is(5, '5')); // Output: false
```

- `Object.isFrozen(obj)`：判断对象是否被冻结。

```
const person = { name: 'John' };
Object.freeze(person);
console.log(Object.isFrozen(person)); // Output: true
```

- `Object.isSealed(obj)`：判断对象是否被封闭。

```
const person = { name: 'John' };
Object.seal(person);
console.log(Object.isSealed(person)); // Output: true
```

- `Object.keys(obj)`：返回一个数组，包含对象中所有可枚举属性的键名。

```
const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // Output: ["name", "age"]
```

- `Object.preventExtensions(obj)`：防止一个对象能够添加新的属性。

```
const person = { name: 'John' };
Object.preventExtensions(person);
person.age = 30; // This line will have no effect as the object is prevented
from adding new properties.
console.log(person); // Output: { name: 'John' }
```

- `Object.seal(obj)`：封闭对象，使其属性不能被删除，但可以修改。

```
const person = { name: 'John', age: 30 };
Object.seal(person);
delete person.age; // This line will have no effect as the object is sealed from
deleting properties.
person.name = 'Jane'; // This line will modify the 'name' property.
console.log(person); // Output: { name: 'Jane', age: 30 }
```

- `Object.setPrototypeOf(obj, prototype)`：设置对象的原型。

```
const person = { name: 'John' };
const proto = { age: 30 };
Object.setPrototypeOf(person, proto);
console.log(person.age); // Output: 30
```

- `Object.values(obj)`：返回一个数组，包含对象中所有可枚举属性的值。

```
const person = { name: 'John', age: 30 };
console.log(Object.values(person)); // Output: ["John", 30]
```

除此之外，JavaScript 对象还有其他方法可供使用，但这些方法是常用的并涵盖了大部分对象操作需求。

查看js对象是否有某个属性

可以使用 `in` 运算符来检查一个 JavaScript 对象是否拥有某个属性，语法如下：

```
property in object
```

其中，`property` 为属性名称，`object` 为要检查的对象。

例如，如果要检查对象 `flowers` 是否拥有 `description` 属性，可以使用以下代码：

```
if ('description' in flowers) {
  // 对象有 description 属性
} else {
  // 对象没有 description 属性
}
```

或者可以使用 `Object.hasOwnProperty()` 方法来检查对象是否拥有某个属性，语法如下：

```
object.hasOwnProperty(property)
```

其中，`object` 为要检查的对象，`property` 为属性名称。该方法返回一个布尔值，表示该对象是否拥有指定的属性。

例如，如果要检查对象 `flowers` 是否拥有 `description` 属性，可以使用以下代码：

```
if (flowers.hasOwnProperty('description')) {
  // 对象有 description 属性
} else {
  // 对象没有 description 属性
}
```

prototype

在 JavaScript 中，每个对象都有一个称为原型（prototype）的内部属性。原型可以被理解为一个模板或者蓝图，它定义了对象的属性和方法。对象通过原型继承属性和方法。当对象需要访问一个属性或方法时，如果本身没有，就会沿着原型链往上查找，直到找到为止。

prototype 对象的主要作用是**实现继承。**在 JavaScript 中，使用构造函数创建对象时，对象会继承它的原型中的属性和方法。原型中定义的属性和方法可以被所有实例共享，避免了在每个实例中都定义同样的属性和方法，从而减小了内存的消耗。

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function() {
  return this.firstName + ' ' + this.lastName;
};

const john = new Person('John', 'Doe');
console.log(john.getFullName()); // Output: "John Doe"
```

在这个例子中，我们首先定义了一个 `Person` 构造函数，用于创建一个具有 `firstName` 和 `lastName` 属性的对象。然后，我们通过 `Person.prototype` 对象添加了一个 `getFullName` 方法。由于 `getFullName` 方法被添加到了 `Person` 构造函数的原型中，因此在使用 `new` 运算符创建 `Person` 对象时，该方法也将成为对象的属性，我们可以通过 `john.getFullName()` 访问它。这种方法不仅使代码更加可读和易于维护，而且还避免了为每个对象添加方法所需的额外开销。

可以直接给实例对象添加方法，例如：

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.getFullName = function() {
    return this.firstName + ' ' + this.lastName;
  };
}

const john = new Person('John', 'Doe');
console.log(john.getFullName());
```

但是这样的做法会导致每个实例对象都有自己的 `getFullName` 方法，这会浪费内存空间。使用原型定义方法可以让所有实例对象共享同一个方法，节省内存空间。此外，原型链的机制使得即使在实例对象中没有找到对应的方法，JavaScript引擎也会在原型链上查找该方法，从而实现方法的复用。因此，通常建议使用原型定义方法。

通过原型修改某个属性会影响到该原型下的所有对象。因为JavaScript的原型继承机制是基于引用的，所以如果多个对象共享同一个原型，那么它们的属性都指向同一个内存地址。因此，如果通过一个对象修改原型上的属性，其他对象也会看到这个变化。这一点需要特别注意，避免出现不必要的副作用。

Inheritance

在 JavaScript 中，继承通常通过原型链来实现。每个对象都有一个指向它的原型对象的内部链接，也就是原型链。如果对象无法在自身属性中找到所需的属性或方法，它会继续在其原型对象中查找，直到找到该属性或方法为止。

JavaScript 中的继承有两种方式：原型继承和构造函数继承。

1. **原型继承**： 原型继承的实现方式是让一个构造函数的原型对象指向另一个构造函数的实例，从而实现子类继承父类的属性和方法。代码如下：

```
// 父类
function Animal(name) {
  this.name = name;
}
Animal.prototype.eat = function() {
  console.log(`${this.name} is eating.`);
}

// 子类
function Dog(name) {
  this.name = name;
}
Dog.prototype = new Animal();  // 原型继承
Dog.prototype.bark = function() {
  console.log(`${this.name} is barking.`);
}

const dog = new Dog('Tom');
dog.eat(); // Tom is eating.
dog.bark(); // Tom is barking.
```

2. **构造函数继承**： 构造函数继承的实现方式是在子类构造函数中通过 `call` 或 `apply` 方法调用父类构造函数，并将子类实例作为上下文。这种方式可以实现子类独有的属性和方法，但无法继承父类原型链上的属性和方法。代码如下：

```
// 父类
function Animal(name) {
  this.name = name;
}
Animal.prototype.eat = function() {
  console.log(`${this.name} is eating.`);
}

// 子类
function Dog(name, breed) {
  Animal.call(this, name); // 构造函数继承
  this.breed = breed;
}
Dog.prototype.bark = function() {
  console.log(`${this.name} is barking.`);
```

```
}

const dog = new Dog('Tom', 'Golden Retriever');
console.log(dog.name); // Tom
console.log(dog.breed); // Golden Retriever
dog.bark(); // Tom is barking.
```

函数与函数对象

函数可以被赋值给变量、作为参数传递给其他函数、作为其他函数的返回值等等。函数对象则是函数在 JavaScript 中的一种数据类型，它可以包含一些属性和方法，例如函数名、参数、返回值等等。

可以这么理解，函数是行为，函数对象是数据结构。函数对象存储函数本身的信息，例如代码块、参数列表等等，而函数则是可以执行的代码块。

举个例子，假设有以下的代码：

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

const myGreet = greet;
```

在这里，`greet` 是一个函数，它定义了打印问候语的行为。`myGreet` 则是一个函数对象，它指向了 `greet` 函数，并且也可以被用来执行打印问候语的行为，例如：

```
myGreet('John'); // 输出 "Hello, John!"
```

因此，函数对象是可以包含函数行为的一种数据类型。

解构赋值

```
const [id, name] = [w.record.fields.ward_id, w.record.fields.name];
<!-- 等价于 -->
const id = w.record.fields.ward_id;
const name = w.record.fields.name;
```

以下是几个解构赋值的示例：

1. 数组解构赋值

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

2. 对象解构赋值

```
const user = {
  name: 'Alice',
  age: 30,
  email: 'alice@example.com'
};
const { name, age, email } = user;
console.log(name); // 'Alice'
console.log(age); // 30
console.log(email); // 'alice@example.com'
```

3. 嵌套的数组解构赋值

```
const numbers = [1, 2, [3, 4]];
const [a, b, [c, d]] = numbers;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(d); // 4
```

4. 函数参数解构赋值

```
function greet({ name, age }) {
  console.log(`Hello, my name is ${name} and I'm ${age} years old.`);
}
const person = { name: 'Alice', age: 30 };
greet(person); // "Hello, my name is Alice and I'm 30 years old."
```

5. 剩余元素的解构赋值

```
const numbers = [1, 2, 3, 4, 5];
const [a, b, ...rest] = numbers;
console.log(a); // 1
console.log(b); // 2
console.log(rest); // [3, 4, 5]
```

这些示例只是解构赋值的基础用法，实际上解构赋值还有更多的用法和技巧。

```
Promise.all([fetch('first-URL'), fetch('second-URL')])
  .then([response1, response2] => Promise.all([response1.json(), response2.json()]))
  .then([data1, data2] => {
    // ... here you can use both data1 and data2 ...
  });

<!-- 等价于 -->
Promise.all([fetch('first-URL'), fetch('second-URL')])
  .then(responses => Promise.all(responses.map(r => r.json())))
  .then([data1, data2] => {
    // ... here you can use both data1 and data2 ...
  });
```

```
/*
这两段代码的功能是相同的，只是第二段代码使用了更简洁的方式来实现。在第一段代码中，使用了解构数组语法来获取包含两个响应对象的数组，然后在第二个 then 中再次使用 Promise.all 来获取解析后的 JSON 数据。

而第二段代码直接在第一个 then 中使用 Array.prototype.map 方法来遍历响应对象数组并将每个响应对象转换为对应的 JSON 数据，然后使用 Promise.all 方法等待所有异步操作完成并返回一个包含所有解析后 JSON 数据的数组，以便在第二个 then 中进行进一步处理。
*/
```

方括号表示解构，将 `Promise.all` 返回的结果中的第一个元素赋值给 `response1`，第二个元素赋值给 `response2`。因此，这里的 `[response1, response2]` 是一个数组，用于保存解构的结果。

这段代码的执行顺序如下：

1. 创建一个 Promise.all，接收两个 fetch 请求作为参数，并返回一个 Promise 对象。
2. 当两个 fetch 请求都返回结果时，Promise.all 的 Promise 对象会被 resolve，返回两个 Response 对象组成的数组。
3. 然后在 Promise.all 的回调函数中，将两个 Response 对象的 json 数据解析出来，返回一个包含两个 Promise 对象的数组。
4. 当两个 Promise 都 resolve 时，Promise.all 的 Promise 对象会被 resolve，返回两个 json 数据组成的数组。
5. 最后在第二个 then 中，可以拿到两个 json 数据并进行处理。

Callback

在 JavaScript 中，Callback 和 Promises 是用于处理异步操作的两种常见方式。

Callback 是 JavaScript 中一种常见的处理异步操作的方式。它是一种函数，被传递给另一个函数，并且在特定的事件或操作完成时被调用。这个被调用的函数被称为回调函数。例如，setTimeout() 函数接受一个回调函数作为其第一个参数，该函数将在一定的时间后被调用。

Callback可以类比为C语言中函数指针作为参数传入函数。在JavaScript中，callback是指把一个函数作为参数传递给另一个函数，在后者执行完毕后再调用前者。这种方式使得代码可以异步执行，提高了程序的性能和效率。

下面是一个使用 Callback 的示例：

```
function loadData(url, callback) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url, true);
  xhr.onload = function() {
    if (xhr.status === 200) {
      callback(xhr.responseText);
    }
  };
  xhr.send();
}
```

```
loadData('https://example.com/data', function(response) {
  console.log(response);
});
```

Callback是否异步取决于之前的函数:

```
function getData(callback) {
  const data = fetchDataSync();
  callback(data);
}
```

如果`fetchDataSync()`是同步获取数据的，那么`callback`函数也会在`getData`函数内同步执行；如果`fetchDataSync()`是异步获取数据的，那么`callback`函数就会异步执行。

至于为什么Promise属于微任务，而Callback却不一定，是因为Promise是ECMAScript规范中定义的一种异步模式，它有自己的执行顺序和行为。而Callback函数并没有特别的规范定义，它的执行方式完全取决于传入的函数。在Node.js中，许多Callback函数都是异步的，而在浏览器端，许多Callback函数都是同步的。因此，Callback函数无法像Promise一样被统一定义为微任务。

同步与异步

在 JavaScript 中，同步和异步是指代码执行的方式。

同步指的是代码按照顺序依次执行，每一行代码执行完之后再执行下一行代码，直到所有代码都执行完毕。

异步指的是代码执行不是按照顺序依次执行，而是在某些操作完成之后再执行。比如在使用 AJAX 发送请求时，JavaScript 不会一直等待服务器响应，而是继续执行下面的代码。当服务器响应后，JavaScript 才会执行回调函数。

JavaScript 中通常使用回调函数、Promise、async/await 等方式来实现异步编程。

Callback Hell or Pyramid of doom

Callback Hell 或者 Pyramid of Doom 是指一个嵌套过多的回调函数结构，导致代码难以阅读和维护的情况。这种情况通常出现在需要执行多个异步操作，每个操作完成后需要执行下一个操作的场景中，代码结构类似于金字塔形，回调函数不断嵌套，代码难以理解和调试。

例如，下面的代码就是一个 Callback Hell 的例子：

```
doSomething(function(result1) {
  doSomethingElse(result1, function(result2) {
    doAnotherThing(result2, function(result3) {
      doFinalThing(result3, function(finalResult) {
        console.log('Got the final result: ' + finalResult);
      }, failureCallback);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

可以看到，每次执行异步操作，都需要嵌套一个回调函数，代码的结构类似于金字塔形，导致代码难以维护和扩展。为了解决这个问题，可以使用 Promise 或 async/await 等方式来处理异步操作，使代码更加清晰和易读。

Promise

Promise 是一种更现代的处理异步操作的方式。Promise 是一个对象，表示异步操作的最终完成或失败，并且可以获取其结果。Promise 可以使代码更加清晰和简洁，并且可以更好地处理错误和异常情况。Promise 具有以下三种状态：

- Pending：异步操作正在进行中，尚未完成。
- Fulfilled：异步操作已成功完成并返回结果。
- Rejected：异步操作已失败并返回一个错误。

下面是一个使用 Promise 的示例：

```
function loadData(url) {
  return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onload = function() {
      if (xhr.status === 200) {
        resolve(xhr.responseText);
      } else {
        reject(Error(xhr.statusText));
      }
    };
    xhr.onerror = function() {
      reject(Error('Network Error'));
    };
    xhr.send();
  });
}

loadData('https://example.com/data')
  .then(function(response) {
    console.log(response);
  })
  .catch(function(error) {
    console.log(error);
  });
```

在这个例子中，loadData() 函数返回一个 Promise 对象。当异步操作成功完成时，调用 resolve() 方法并传递结果。当异步操作失败时，调用 reject() 方法并传递错误对象。然后可以使用 then() 和 catch() 方法来处理异步操作的结果或错误。

链式调用

Promise 有许多链式调用方法，如下所示：

1. then()：定义 Promise 实例状态从“Pending”到“Fulfilled”或“Rejected”时，分别执行的回调函数。
2. catch()：定义 Promise 实例状态为“Rejected”时执行的回调函数。
3. finally()：定义 Promise 实例状态无论是“Fulfilled”还是“Rejected”都要执行的回调函数，类似于 try...catch...finally 中的 finally。
4. all()：接收一个 Promise 实例数组，返回一个新的 Promise 实例，当数组中所有 Promise 实例都状态为“Fulfilled”时，返回的 Promise 实例状态为“Fulfilled”，否则返回的 Promise 实例状态为“Rejected”。
5. race()：接收一个 Promise 实例数组，返回一个新的 Promise 实例，当数组中任意一个 Promise 实例状态为“Fulfilled”或“Rejected”时，返回的 Promise 实例的状态就会跟随它。
6. any()：接收一个 Promise 实例数组，返回一个新的 Promise 实例，当数组中任意一个 Promise 实例状态为“Fulfilled”时，返回的 Promise 实例状态为“Fulfilled”，否则返回的 Promise 实例状态为“Rejected”。
7. allSettled()：接收一个 Promise 实例数组，返回一个新的 Promise 实例，等待数组中所有 Promise 实例都完成（无论状态是“Fulfilled”还是“Rejected”），返回一个包含所有 Promise 实例状态和值的数组。

此外，forEach 方法并不是 Promise 实例的方法，而是数组的方法。在 Promise 中，可以使用 then() 方法来对数组进行操作。then() 方法接收一个函数作为参数，该函数会在 Promise 实例状态为“Fulfilled”时被调用，并将 Promise 实例的结果传递给该函数。可以在该函数中使用 forEach 方法来对结果数组进行操作。

Fetch

Fetch 是一种现代的 Web API，用于从服务器获取资源。它提供了一种比传统的 XMLHttpRequest（XHR）对象更简洁、更易于使用的方式来发出网络请求。

Fetch 可以用于获取各种资源，例如文本、JSON、HTML 和二进制数据。通过使用 Promise 对象，可以处理成功和失败的响应，并在获取响应后对其进行操作。

Fetch会返回一个Promise对象

调用 `fetch` 函数后，会返回一个 `Promise` 对象，其中的状态为 `pending`，并在内部创建一个 `Response` 对象来表示请求的响应。当接收到响应数据时，会将其传递给 `Response` 对象，并将 `Promise` 对象的状态更新为 `resolved`。此时，可以在 `then` 方法中访问该响应，并对其进行处理。如果请求发生错误或者响应的状态码不是 `200 OK`，则会将 `Promise` 对象的状态更新为 `rejected`，并在 `catch` 方法中处理该错误。

可以使用箭头函数或普通函数来实现，在函数的参数中传入响应对象，然后对响应进行处理。下面是两个示例：

使用箭头函数：

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

使用普通函数：

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(function(response) {
    return response.json();
  })
  .then(function(data) {
    console.log(data);
  })
  .catch(function(error) {
    console.error(error);
  });
```

Fetch 还允许您设置请求头、使用 POST 方法发送数据以及通过使用其他配置选项来自定义请求。

.json()和Json.parse()

`.json()` 和 `JSON.parse()` 都是用于将 JSON 数据转换为 JavaScript 对象的方法，但它们的使用场景有所不同。

`.json()` 是 `fetch()` 方法返回的响应对象的一个方法，它会将响应体解析为 JSON 格式的字符串，然后返回对应的 JavaScript 对象，因此它只能用于处理网络请求返回的 JSON 数据。

`.json()` 方法会将 JSON 格式的响应数据解析为 JavaScript 对象，并返回一个 Promise 对象，该 Promise 对象的解决值是一个 JavaScript 对象。因此，可以使用 `.then()` 方法来获取该对象，并进一步处理。

例如，可以使用以下代码将一个远程的 JSON 数据转换为 JavaScript 对象：

```
fetch('https://example.com/data.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

`JSON.parse()` 是全局对象 `JSON` 的一个方法，可以将任意格式的 JSON 字符串转换为对应的 JavaScript 对象，因此它可以用于处理从任何来源获取到的 JSON 数据，包括通过网络请求获取的 JSON 数据，以及保存在本地的 JSON 文件。

例如，可以使用以下代码将一个 JSON 字符串转换为 JavaScript 对象：

```
const jsonString = '{"name": "Alice", "age": 25}';
const data = JSON.parse(jsonString);
console.log(data);
```

需要注意的是，`.json()` 方法返回一个 Promise 对象，而 `JSON.parse()` 方法直接返回一个 JavaScript 对象，因此在使用时需要根据具体的场景选择使用哪种方法。

宏任务微任务

执行顺序：宏任务 => 微任务 => 渲染 => 宏任务

宏任务

常见的宏任务包括：

1. setTimeout
2. setInterval
3. setImmediate
4. requestAnimationFrame
5. I/O操作（例如读取文件）
6. UI渲染
7. MessageChannel
8. postMessage

在浏览器环境中，DOM事件也是一种宏任务。在Node.js环境中，process.nextTick()也是一种宏任务。

微任务

常见的微任务有以下几种：

1. Promise回调函数：在Promise的resolve或reject函数中执行的回调函数会被放入微任务队列中。
2. process.nextTick：Node.js环境中的微任务，使用process.nextTick方法注册的回调函数会被放入微任务队列中。
3. MutationObserver：当DOM发生变化时，注册的回调函数会被放入微任务队列中。
4. Object.observe：已经被废弃，使用Proxy代替。
5. IntersectionObserver：当被观察的元素进入或离开视窗时，注册的回调函数会被放入微任务队列中。

需要注意的是，不同的JavaScript引擎和运行环境可能对微任务的实现方式有所不同，因此可能会存在微任务执行顺序等方面的差异。

Dom

DOM (Document Object Model) 是一种将 HTML、XML 和 SVG 文档视为对象的方法，它定义了一种标准的方式来访问和操作文档的内容和结构。通过 DOM，开发者可以以编程方式访问和修改文档的元素、属性和文本内容。DOM 中的每个元素都是一个节点（Node），节点可以是元素节点（Element Node）、文本节点（Text Node）、注释节点（Comment Node）或者其他节点类型。

以下是一些常用的 DOM 方法和属性：

- document.getElementById(id)：通过 ID 获取一个元素。
- document.getElementsByTagName(name)：通过标签名获取元素。
- document.getElementsByClassName(name)：通过 class 名称获取元素。
- element.innerHTML：设置或获取一个元素的 HTML 内容。
- element.style.property：设置或获取一个元素的 CSS 样式。
- element.getAttribute(name)：获取一个元素的属性值。
- element.setAttribute(name, value)：设置一个元素的属性值。

- element.addEventListener(event, function)：为一个元素添加事件监听器，第二个参数必须是无参回调函数。
- element.appendChild(newChild)：在一个元素的末尾添加一个子元素。
- element.append(childNode1, childNode2, ...)：在元素末尾添加多个子元素
- element.removeChild(child)：从一个元素中移除一个子元素。
- element.classList.add(className)：为一个元素添加一个 class 名称。
- element.classList.remove(className)：从一个元素中移除一个 class 名称。
- element.classList.toggle(className)：如果一个元素包含某个 class 名称，则移除该名称，否则添加该名称。

除了上述方法和属性，还有许多其他的 DOM 方法和属性，可以根据实际需要选择使用。在开发过程中，使用合适的 DOM 方法和属性可以提高开发效率和代码可读性，同时也能够使代码更加简洁、优雅。

创建dom

通过DOM方法可以创建新的DOM节点，常用的创建方法包括：

1. createElement(tagName)：创建一个指定标签名的新元素节点。

示例：

```
const newDiv = document.createElement('div');
```

2. createTextNode(text)：创建一个包含指定文本的新文本节点。

示例：

```
const newText = document.createTextNode('Hello, world!');
```

3. createDocumentFragment()：创建一个新的空白的文档片段。

示例：

```
const newFragment = document.createDocumentFragment();
```

创建完成之后，还需要使用DOM方法将创建的节点插入到文档中，比如使用appendChild方法将新节点添加到已有的节点中。

示例：

```
const parent = document.getElementById('parent');
parent.appendChild(newDiv);
```

这将在id为“parent”的节点中添加一个新的div元素节点。

常见的 DOM API 来创建元素和节点的方法有以下几种：

1. `document.createElement(tagName)`：创建一个指定标签名的元素节点，例如 `document.createElement('div')` 将创建一个 `<div>` 元素。

2. `document.createTextNode(text)`：创建一个包含指定文本的文本节点，例如 `document.createTextNode('Hello World!')` 将创建一个文本内容为 'Hello World!' 的文本节点。
3. `element.appendChild(childElement)`：将一个元素节点或文本节点作为子节点添加到指定的父节点中，例如 `parentElement.appendChild(childElement)` 将子元素或文本节点添加到父元素中。
4. `element.insertBefore(newElement, referenceElement)`：将一个新元素节点插入到指定参考节点之前，例如 `parentElement.insertBefore(newElement, referenceElement)` 将新元素插入到参考元素之前。
5. `document.createDocumentFragment()`：创建一个空的文档片段节点，可以将多个元素节点和文本节点添加到文档片段中，最后再一次性地将文档片段节点添加到 DOM 树中，以提高性能。

使用这些 DOM API 可以避免使用 `innerHTML` 时出现安全漏洞，同时也可以提高代码的可读性和性能。

querySelector

`querySelector` 和 `querySelectorAll` 方法可以用于选择DOM元素，它们是DOM API提供的方法之一。其中，`querySelector` 返回符合CSS选择器的第一个元素，`querySelectorAll` 则返回所有符合条件的元素。这两个方法可以接受任何有效的CSS选择器作为参数，例如：

```
// 选择id为"myElement"的元素
const myElement = document.querySelector('#myElement');

// 选择class为"myClass"的所有元素
const myElements = document.querySelectorAll('.myClass');

// 选择所有<p>元素
const allPTags = document.querySelectorAll('p');
```

另外，DOM API还提供了很多其他的方法和属性，用于操作和查询DOM元素。例如：

- `getElementById`：通过元素ID选择一个元素
- `getElementsByClassName`：通过class选择多个元素
- `getElementsByTagName`：通过标签名选择多个元素
- `createElement`：创建一个新的元素
- `appendChild`：在一个元素的末尾添加一个子元素
- `removeChild`：从一个元素中删除一个子元素
- `classList`：获取或设置元素的class列表，可以用来添加或删除class
- `innerHTML`：获取或设置元素的HTML内容
- `innerText`：获取或设置元素的文本内容（不包括HTML标记）

搭配通配符使用

使用 `querySelector` 和通配符来选取元素时，通配符可以用 `*` 表示匹配任何元素。

以下是一些使用 `querySelector` 和通配符的例子：

1. 选取文档中的所有元素：

```
const allElements = document.querySelectorAll('*');
```

2. 选取文档中所有的 `<p>` 元素：

```
const allParagraphs = document.querySelectorAll('p');
```

3. 选取文档中所有的带有 `class` 属性的元素：

```
const allWithClass = document.querySelectorAll('[class]');
```

4. 选取文档中所有的带有 `data-*` 属性的元素：

```
const allWithDataAttribute = document.querySelectorAll('[data-*]');
```

5. 选取文档中所有的 `<input>` 元素和 `<textarea>` 元素：

```
const allInputs = document.querySelectorAll('input, textarea');
```

Others

innerHTML 和 textContent 的区别及用法

`innerHTML` 和 `textContent` 都是 JavaScript 中常用的用于操作 HTML 元素的属性。

`innerHTML` 是一个可读写的属性，它可以获取或设置 HTML 元素的内容。当获取元素的 `innerHTML` 属性时，它会返回 HTML 标签以及它们包含的内容。当设置元素的 `innerHTML` 属性时，它会将字符串插入到元素中，从而改变元素的内容。

例如，假设我们有一个 `<div>` 元素，我们可以使用以下代码获取和设置它的 `innerHTML` 属性：

```
const divElement = document.querySelector('#myDiv');

// 获取元素的innerHTML属性
const htmlContent = divElement.innerHTML;

// 设置元素的innerHTML属性
divElement.innerHTML = '<p>Hello World!</p>';
```

`textContent` 属性也是一个可读写的属性，它可以获取或设置 HTML 元素的文本内容。当获取元素的 `textContent` 属性时，它会返回元素的文本内容，而不会包含任何 HTML 标签。当设置元素的 `textContent` 属性时，它会将字符串插入到元素中，从而改变元素的文本内容。

例如，假设我们有一个 `<p>` 元素，我们可以使用以下代码获取和设置它的 `textContent` 属性：

```
const paragraphElement = document.querySelector('#myParagraph');

// 获取元素的textContent属性
const textContent = paragraphElement.textContent;

// 设置元素的textContent属性
paragraphElement.textContent = 'Hello World!';
```

需要注意的是，当设置元素的 `textContent` 属性时，它会替换所有子节点的文本内容。而当设置元素的 `innerHTML` 属性时，它会替换整个元素的内容，包括所有子节点。

总之，`innerHTML` 和 `textContent` 都是常用的操作 HTML 元素的属性。在选择使用哪个属性时，需要根据具体的需求来决定，一般来说，如果需要处理 HTML 标签，就应该使用 `innerHTML` 属性，如果只需要处理文本内容，则可以使用 `textContent` 属性。

innerText和textContent区别

innerText 和 textContent 都是用来获取或设置元素内容的属性，它们的主要区别在于对待 HTML 和 CSS 的方式不同。

- textContent 是 W3C 标准属性，它返回元素中包含的所有文本内容，包括所有嵌套的元素、注释和空格。在获取元素内容时，会忽略 HTML 标签和 CSS 样式，返回的是纯文本内容。
- innerText 不是 W3C 标准属性，它返回元素中包含的所有文本内容，但会受到 CSS 样式的影响。在获取元素内容时，会保留 HTML 标签和 CSS 样式，但会忽略 display:none 的元素。

因此，当需要获取元素中的纯文本内容时，应该使用 textContent 属性；当需要获取带有样式的文本内容时，应该使用 innerText 属性。

使用 document.createElement('div'); 创建新的 DOM 分支之后，想要为其加入内容，应该用哪个操作?

使用 DOM API 中的 `appendChild()` 或 `insertBefore()` 方法可以将创建的 DOM 元素加入到文档中，然后使用 `textContent` 或 `innerHTML` 属性为其添加内容。例如：

```
// 创建新的 div 元素
const newDiv = document.createElement('div');

// 将新元素加入到文档中
document.body.appendChild(newDiv);

// 使用 textContent 或 innerHTML 添加内容
newDiv.textContent = 'Hello, world!';
```

Crawling the Web

Wget

wget 是一种非常流行的自由软件，用于从互联网上下载文件。它支持 HTTP、HTTPS 和 FTP 协议，可以在命令行下使用，也可以用于编写脚本。下面是一些 wget 常用的选项：

1. -r, --recursive 递归下载，下载指定链接的所有文件及文件夹。例如，`wget -r http://example.com` 会递归下载 `http://example.com` 中的所有文件和文件夹。
2. -l, --level=depth 设置递归下载的深度。例如 `-l 2` 表示最多下载两层深度的链接。例如，`wget -r -l 2 http://example.com` 会下载 `http://example.com` 及其子页面和链接，但不会进一步递归下载子页面和链接的子页面和链接。
3. -k, --convert-links 转换链接，将指向互联网的链接转换为本地链接，以便离线浏览。例如，`wget -k http://example.com` 会下载 `http://example.com` 中的所有页面和链接，并将其中的互联网链接转换为本地链接。
4. -p, --page-requisites 下载页面所有必要的资源，如图片、样式表和脚本等。例如，`wget -p http://example.com` 会下载 `http://example.com` 中的所有页面和链接，并下载其中的图片、样式表和脚本等资源。
5. -i, --input-file=file 从指定文件中读取链接列表，下载列表中所有链接。例如，`wget -i links.txt` 会从 `links.txt` 文件中读取链接列表，并下载列表中的所有链接。
6. -nc, --no-clobber 不覆盖已存在的文件，如果文件已经存在，则不下载该文件。例如，`wget -nc http://example.com/file.txt` 会下载 `http://example.com/file.txt` 文件，但如果本地已经存在该文件，则不会下载。
7. -U, --user-agent=agent-string 指定用户代理字符串，模拟不同的浏览器。例如，`wget -U Mozilla http://example.com` 会使用 Mozilla 浏览器的用户代理字符串来访问 `http://example.com`。
8. -O, --output-document=file 将下载的文件重命名为指定的文件名。例如，`wget -O file.txt http://example.com/file.html` 会下载 `http://example.com/file.html` 文件，并将其保存为 `file.txt`。
9. -c, --continue 继续下载上次未完成的文件，而不是重新下载整个文件。例如，`wget -c http://example.com/file.zip` 会继续下载 `file.zip` 文件，而不是重新下载整个文件。
10. -t, --tries=num 设定最多重试连接次数，默认为 20 次。例如，`wget -t 5 http://example.com` 会尝试最多 5 次连接 `http://example.com`。
11. -T, --timeout=seconds 设置超时时间，如果下载时间超过指定时间，则 wget 中止。例如，`wget -T 10 http://example.com` 设置超时时间为 10 秒，如果下载时间超过 10 秒，则 wget 中止。例如：wget -q http://example.com/ #静默下载，不输出下载进度
12. -q, --quiet 不输出下载进度信息，只显示错误信息。
13. -b, --background 后台下载模式，将下载放入后台执行。例如： wget -b http://example.com/ #后台下载
14. -S, --server-response 显示服务器的响应头信息。例如： wget -S http://example.com/ #显示服务器的响应头信息
15. -N, --timestamping 只下载比本地文件更新的文件。例如： wget -N http://example.com/ #只下载比本地文件更新的文件
16. -e, --execute=command 执行指定的 shell 命令，在 wget 下载完成后执行。例如： wget -e "wget --spider http://example.com/" http://example.com/ #下载完成后执行--spider命令
17. --no-check-certificate 禁止检查 SSL 证书。例如： wget --no-check-certificate https://example.com/ #禁止检查 SSL 证书

18. --spider 只获取网页的信息，不下载页面的内容。例如： wget --spider http://example.com/ #只获取网页的信息，不下载页面的内容

其中，--spider 命令用于模拟搜索引擎爬虫，只获取网页的信息，不下载页面的内容。

这些命令的顺序一般没有影响，只要各个命令的选项参数正确就可以了。不过，有些选项可能会依赖于其他选项，比如使用 `-k` 选项时，可能需要同时使用 `-p` 选项，否则可能无法正确转换链接。

另外，有些选项参数的顺序可能有影响，比如使用 `-U` 选项时，应该先指定该选项，再指定要下载的链接。因为 `-U` 选项是针对整个命令行的，如果在指定链接之前先指定了 `-U` 选项，可能会导致下载的链接使用了错误的用户代理字符串。

以上只是 wget 命令的一部分，更多选项可以查看 wget 的帮助文档（man wget）。总的来说，wget 是一个非常方便的下载工具，可以帮助我们从互联网上下载需要的文件，方便离线浏览或者后续处理。

```
wget -r -l 2 <url> 下载网页下所有链接（包括指向外部网站的链接）的资源（包括html，js，图片等）
wget -l 2 <url> 只下载本站点下的hmtl文件，如果链接指向外部网页，wget -l 2 不会下载该外部网页。
```

当使用 -r 参数时，wget 会递归下载链接中的内容，包括指向外部网站的链接，除非使用了 --exclude-domain 或 --span-hosts 参数进行限制。

如果网站有robots.txt文件，那么默认情况下，wget会遵循该文件中的规则，即不下载被禁止的内容。如果需要下载被禁止的内容，可以添加`--execute robots=on`参数，如下所示：

```
wget -p --execute robots=on <url>
```

如果存在多个robots.txt或者robots.txt与其他同名文件，wget会默认下载所有同名文件，但只遵循第一个robots.txt的规则，其他同名文件不会生效。

-r 和 -p

`-r` 选项会递归下载指定网站的所有链接，包括 HTML 文件、CSS 文件、JavaScript 文件、图片、视频等等。但是，`-r` 不会自动设置适当的选项来保存所有必要的资源，例如，它不会递归下载所引用的图片和样式表文件，也不会创建适当的目录结构以容纳下载的文件。

`-p` 选项会自动下载所有必要的资源，包括 HTML 文件、CSS 文件、JavaScript 文件、图片、视频等等，并且会自动设置适当的选项来保存这些资源。但是，`-p` 不会递归下载链接，所以只会下载指定网站的根页面和指定的资源，而不会下载其他链接。

因此，当需要递归下载指定网站的所有链接，并且需要保留所有必要的资源以及适当的目录结构时，可以同时使用 `-r` 和 `-p` 选项。

--user-agent

在使用 wget 发起请求时，可以使用 --user-agent 参数来指定请求头中的 User-Agent 字段，从而模拟不同的浏览器。

比如，要模拟 Chrome 浏览器的 User-Agent，可以使用如下命令：

```
wget --user-agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3" http://example.com/
```

这里指定了 User-Agent 为 Chrome 58.0.3029.110 在 Windows 10 上的版本。如果要模拟其他浏览器，可以找到该浏览器的 User-Agent 字符串并替换上述命令中的值即可。

命令示例

假设我们要下载一个网站 https://example.com/ 下的所有内容，包括html页面、图片、CSS、JavaScript等。

1. 下载整个网站：`wget -r -p https://example.com/`

这个命令会递归下载 example.com 网站上的所有内容，并且保留目录结构。使用 `-p` 选项，它会同时下载所有的 CSS、JavaScript 等文件，以保证网站完整地保存在本地。

1. 限制递归深度：`wget -r -l 2 https://example.com/`

如果我们只想下载 example.com 网站上的两层内容，可以使用 `-l` 选项，指定递归深度为 2。

2. 下载指定类型文件：`wget -r -A jpg,jpeg,png,gif https://example.com/`

如果我们只想下载 example.com 网站上的图片，可以使用 `-A` 选项，指定要下载的文件类型。

3. 下载指定目录：`wget -r -np -nH --cut-dirs=1 https://example.com/images/`

如果我们只想下载 example.com 网站上的 `/images` 目录下的内容，可以使用 `--cut-dirs` 选项，删除 URL 中的前缀目录。使用 `-np` 选项，它不会递归爬行到父目录，以保持目录结构简洁。使用 `-nH` 选项，它不会在本地保存主机名目录，以保证目录结构更加简洁。

4. 断点续传：

```
`wget -c https://example.com/file.zip`
```

如果下载过程中网络中断了，我们可以使用 `-c` 选项恢复之前的下载进度，从上次中断的地方继续下载。

5. 后台下载

```
wget -b https://example.com/file.zip`
```

如果我们希望 wget 在后台运行，可以使用 `-b` 选项。在后台运行时，wget 会把输出信息保存到一个日志文件中。

6. 下载单个文件：

```
wget http://example.com/file.zip
```

7. 下载多个文件：

```
wget http://example.com/file1.zip http://example.com/file2.zip
http://example.com/file3.zip
```

8. 下载一个网站：

```
wget --recursive --no-clobber --page-requisites --html-extension --convert-links --restrict-file-names=windows --domains website.org --no-parent http://website.org
```

9. 下载一个文件并重命名：

```
wget -O newfile.zip http://example.com/file.zip
```

10. 限速下载：

```
wget --limit-rate=50k http://example.com/file.zip
```

11. 使用代理下载：

```
wget --proxy-user=username --proxy-password=password http://example.com/file.zip
```

12. 下载一个文件并重命名为另一个文件名：

```
wget -O newfilename https://example.com/oldfilename
```

13. 限制下载速度为 100KB/s：

```
wget --limit-rate=100k https://example.com/file.zip
```

14. 下载文件，但只下载文件大小大于 100MB 的文件：

```
wget --spider --reject "size<100M" https://example.com/
```

15. 下载整个网站并保存到本地，同时将所有网址转换为本地链接：

```
wget --mirror --convert-links https://example.com/
```

16. 下载整个网站，但排除某些文件或目录：

```
wget --mirror --exclude-directories /tmp,/proc https://example.com/
```

17. 下载文件并使用 HTTP 用户名和密码进行身份验证：

```
wget --user=username --password=password https://example.com/file.zip
```

18. 从多个 URL 中下载文件：

```
wget -i urls.txt
```

其中，`urls.txt` 是包含要下载文件的 URL 的文本文件。

19. 下载文件并继续从上次停止的地方继续下载：

```
wget -c https://example.com/largefile.zip
```

20. 下载文件并保存到指定目录：

```
wget -P /path/to/save http://example.com/file.zip
```

21. 设置下载速度限制：

```
wget --limit-rate=200k http://example.com/file.zip
```

22. 递归下载并跳过指定文件类型：

```
wget -r -R "*.jpg,*.png" http://example.com/
```

23. 下载并将输出保存到日志文件中：

```
wget -o log.txt http://example.com/
```

24. 可以使用wget的-O选项来指定下载内容的保存文件名，例如：

```
wget -O output.html http://example.com/
```

这样就会将下载内容保存为output.html文件。

25. 要修改下载内容的根目录名，可以使用--directory-prefix选项，例如：

```
wget --directory-prefix=/path/to/folder http://example.com/
```

这样就会将下载内容保存在/path/to/folder目录下。

Robots.txt

robots.txt是一个文本文件，位于网站根目录下，用于向搜索引擎爬虫说明哪些页面可以访问、哪些不可以访问。网站管理员可以使用robots.txt文件控制搜索引擎爬虫访问他们的网站中哪些内容。通过禁止搜索引擎爬虫访问不必要的内容，可以节省服务器资源并提高网站安全性。

Recursive downloading

`wget -r -l N <url>`的作用是递归地下载指定URL中的文件，同时限制递归的深度为N级别。其中，`-r`表示递归下载，`-l`表示限制递归深度。N可以是数字或`inf`（表示无限递归）。

```
wget -r -l N <url>
-l N: The level of recursion to permit. (Default if omitted is 5).
```

在这个命令中，`-l` 选项指定了递归深度。递归深度是指在下载一个网站的时候，要下载多少级链接。例如，如果您使用命令 `wget -r -l 2 http://example.com`，则 `wget` 将下载 `example.com` 的主页，并下载其所有链接和子链接（如果有）一级深度。然后，对于每个链接，它将下载所有链接的链接（如果有），以此类推，直到达到指定的递归深度为止。

例如，假设 `example.com` 主页上有三个链接： `http://example.com/link1`，`http://example.com/link2`, and `http://example.com/link3`。如果递归深度为 1，那么 `wget` 将下载主页以及这三个链接。如果递归深度为 2，那么 `wget` 将下载主页、这三个链接和它们链接的任何东西。如果递归深度为 3，那么 `wget` 将下载主页、这三个链接、链接的任何内容以及任何进一步链接的内容，以此类推，直到达到指定的深度为止。

Mirroring

`wget -m <url>`是一个wget命令，用于递归下载指定URL的所有页面、链接、图片、样式表等内容，以便将其完全镜像到本地文件系统中。

-m选项表示镜像模式（mirror），会递归下载指定URL的所有链接，并将其保存到本地文件系统中，以便离线浏览和使用。

使用方式如下：

```
wget -m <url>
```

其中，`<url>`是指要下载的URL链接，可以是HTTP或FTP链接。

下载的内容保存在当前工作目录中，保存的文件夹名称以所下载的URL的主机名命名，例如，如果下载的URL是`http://example.com`，则保存的文件夹名称为`example.com`。

在镜像模式下，还可以使用一些其他的选项，比如`-r`用于指定递归下载深度，`-np`用于不递归下载父级链接，`-nc`用于不覆盖已经存在的文件等。

格式示例：

```
wget -m -r -np -nc http://example.com/
```

该命令将递归地下载`http://example.com/`及其下面的所有链接，不包括父级链接，同时不会覆盖已经存在的文件。

exercise

1. Read `man wget` to understand what the `-i` `--force-html` and `--spider` options do. Download a copy of this webpage (the one you are currently reading) and use `wget` to test all the links on the page. Are there any broken links?

   Q: 请阅读man wget文档，了解-i、--force-html和--spider选项的含义。下载这个网页（你正在阅读的这个），并使用wget测试页面上的所有链接。有没有失效的链接?

   A: -i：从指定的文件中读取要下载的URL列表，可以与 -B、-L、--input-file 一起使用。--force-html：将文件当作HTML文件对待。--spider：只检查URL是否存在。 下载这个网页并测试链接可以通过以下命令完成：

```
wget --spider -r -p -l 1 -o wget.log https://openai.com/
```

上述命令会测试页面上所有链接，并输出结果到日志文件wget.log。如果页面上有失效的链接，会在日志文件中显示。

2. Tell `wget` to use a different user agent string in a request to your server running on localhost. Check what the request looks like to your server.

   Q: 告诉wget在向运行在localhost上的服务器发出请求时使用不同的用户代理字符串。检查请求对你的服务器的请求是什么样子的。

   A: 可以使用 -U 选项来指定用户代理。例如，以下命令指定了一个Mozilla Firefox浏览器的用户代理字符串：

   ```
   wget -U Mozilla/5.0 http://localhost:8080/
   ```

   这会让服务器认为请求来自于Mozilla Firefox浏览器。

3. How would `wget -l 1 http://example.com` differ from `wget -p http://example.com`? *(Hint: think about external resources)*.

   Q: wget -l 1 http://example.com和wget -p http://example.com有什么区别？（提示：考虑外部资源）

   A: 1 表示只递归下载一个层级的页面，而 -p 表示下载页面所需的所有资源，包括外部资源（例如JavaScript文件、样式表等）。因此，wget -p会下载页面所需的所有资源，包括外部资源，而wget -l 1只会下载页面本身。

4. Look for 'Recursive Accept/Reject options' in the `wget` manpage. How would you get `wget` to crawl pages from multiple different domains?

   Q: 在wget manpage中查找“递归接受/拒绝选项”。如何让wget爬取来自多个不同域的页面?

   A: 可以使用 --domains 和 --exclude-domains选项来控制wget递归的域名。例如，以下命令将递归下载www.example.com和www.example.org，但排除www.example.net：

   ```
   wget --recursive --domains=www.example.com,www.example.org --exclude-
   domains=www.example.net http://www.example.com/
   ```

5. Look up what `-nc` does. What is clobbering, and why would or wouldn't you want to do it?

   Q: 查找-nc的含义。什么是覆盖，为什么会或不会想要这样做?

   A: -nc 表示“不要覆盖”，即如果已经存在相同的文件，则不要覆盖它。覆盖是指将一个文件覆盖另一个文件，以至于原始文件被删除。有些情况下，我们可能不想覆盖文件，因为这会导致数据丢失。例如，如果我们在下载文件时使用-w选项指定了一个等待时间，并且在等待期间我们希望停止下载，则不覆盖文件可能会很有用，因为我们可以稍后恢

BeautifulSoup

BeautifulSoup是一个Python库，用于从HTML或XML文件中提取数据。它是一个解析器，可以帮助开发者以编程方式解析HTML或XML文档，并从中提取所需的数据。

BeautifulSoup的主要功能是将HTML或XML文档解析为Python对象，并且可以轻松地遍历和搜索该对象，以提取所需的信息。使用BeautifulSoup可以方便地获取文档中的标签、文本、属性等信息，帮助开发者快速地获取所需的数据，从而方便地进行数据分析和处理。

BeautifulSoup支持多种解析器，包括Python标准库中的HTML解析器和XML解析器，以及第三方库lxml等。开发者可以根据实际情况选择合适的解析器来进行文档解析。

下面是使用BeautifulSoup的一些基本步骤：

1. 安装BeautifulSoup库：可以使用pip或conda安装，比如：

```
pip install beautifulsoup4
```

1. 导入BeautifulSoup库：

```
from bs4 import BeautifulSoup
```

1. 读取HTML文件或HTML字符串，将其转化为BeautifulSoup对象：

```
# 读取HTML文件
with open('example.html', 'r') as file:
    html = file.read()

# 将HTML字符串转化为BeautifulSoup对象
soup = BeautifulSoup(html, 'html.parser')
```

1. 使用BeautifulSoup对象进行解析和数据提取。可以使用BeautifulSoup对象的find()和find_all()方法查找HTML标签，使用get()方法获取标签属性值，使用text属性获取标签内容等等。例如：

```
# 查找第一个<p>标签
p_tag = soup.find('p')

# 查找所有<a>标签
a_tags = soup.find_all('a')

# 获取第一个<a>标签的href属性值
href = a_tags[0].get('href')

# 获取第一个<p>标签的内容
p_content = p_tag.text
```

这些是BeautifulSoup的一些基本使用方法，更多细节请参考官方文档。

exercise

1. Take a look at some other examples from the BeautifulSoup documentation, in particular regarding the use of the `.find_all()` method.

题目要求查看BeautifulSoup文档中的其他示例，特别是使用**.find_all()方法**的示例。BeautifulSoup是一个Python库，用于从HTML和XML文件中提取数据。.find_all()方法是BeautifulSoup库中的一个函数，用于根据标签名，属性和内容搜索文档中的元素。

> Beautiful Soup find_all() Signature: find_all(name, attrs, recursive, string, limit, **kwargs) The find_all() method looks through a tag's descendants and retrieves all descendants that match your filters. I gave several examples in Kinds of filters, but here are a few more:find_all() 方法查看标签的后代并检索与您的过滤器匹配的所有后代。我在 Kinds of filters 中给出了几个示例，但这里还有一些： soup.find_all("title") # [] soup.find_all("p", "title") # [
>
> **The Dormouse's story**
>
>] find
>
> tag with title class soup.find_all("a") # [Elsie</ a>, # Lacie</ a>, # Tillie</ a>] soup.find_all(id="link2") # [Lacie</ a>] import re //regular expression soup.find(string=re.compile("sisters")) # u'Once upon a time there were three little sisters; and their names were\n'

这段文字是 Beautiful Soup 库的 find_all() 方法的说明文档。find_all() 方法用于查找指定标签的后代，并返回所有符合条件的后代元素。该方法的参数包括：

- `name`：要查找的标签名称，可以是字符串、正则表达式、列表等多种类型。
- `attrs`：要查找的标签属性及属性值，可以使用字典、正则表达式等多种方式进行匹配。
- `recursive`：是否递归查找标签的后代，默认为 True。
- `string`：要查找的字符串，可以用于查找包含指定文本的标签。
- `limit`：返回的结果数量限制。

该文档提供了一些 find_all() 方法的示例，例如：

- `soup.find_all("title")`：查找所有的 title 标签。
- `soup.find_all("p", "title")`：查找所有 class 属性为 title 的 p 标签。
- `soup.find_all("a")`：查找所有的 a 标签。
- `soup.find_all(id="link2")`：查找所有 id 属性为 link2 的元素。
- `soup.find(string=re.compile("sisters"))`：查找包含字符串 "sisters" 的文本。

这些示例展示了如何使用不同的过滤器来查找标签及其后代元素，其中包括标签名称、属性、文本内容等多种方式

2. Use your interpreter to access a list of all the `<strong>` elements in the webpage, and figure out how to print out just the text contained within them.

   使用以下代码可以找到网页中所有的**元素并打印出它们的文本内容**：

```
from bs4 import BeautifulSoup
import requests

# 使用requests库获取网页的内容
url = 'http://example.com/'
response = requests.get(url)
```

```
# 将网页内容解析为BeautifulSoup对象
soup = BeautifulSoup(response.text, 'html.parser')

# 使用.find_all()方法找到所有的<strong>元素
strong_tags = soup.find_all('strong')

# 循环遍历<strong>元素列表，打印其中的文本内容
for tag in strong_tags:
    print(tag.text)
```

3. **How would you use `.find_all()` to find all `<div>` elements with a particular class value (e.g., 'container')? Would your method work if the div had multiple classes?**

   可以使用以下代码来查找所有具有特定类值（例如“container”）的

   元素，并且即使
   具有多个类，该方法也是有效的：

```
# 使用.find_all()方法查找所有具有特定类值（例如“container”）的<div>元素
div_tags = soup.find_all('div', {'class': 'container'})

# 循环遍历<div>元素列表，打印其中的文本内容
for tag in div_tags:
    print(tag.text)
```

4. **解释一下wget的`Recursive Accept/Reject options`，并举例**

   `Recursive Accept/Reject options`是wget命令中的选项，用于控制递归下载时哪些文件应该被接受或拒绝下载。

   具体来说，以下是wget中递归接受/拒绝选项的说明：

   - **`--accept`：接受指定的文件类型。可以使用通配符匹配多个文件类型，例如--accept=jpg,gif表示接受jpg和gif类型的文件。**
   - **`--reject`：拒绝指定的文件类型。可以使用通配符匹配多个文件类型，例如--reject=txt表示拒绝txt类型的文件。**
   - **`--accept-regex`：接受匹配正则表达式的文件名。例如，`--accept-regex='.jpg$'`表示接受以.jpg为扩展名的文件。**
   - **`--reject-regex`：拒绝匹配正则表达式的文件名。例如，`--reject-regex='(index|home).html'`表示拒绝包含`index.html`或`home.html`的文件。**

   下面是一个wget命令的例子，该命令将下载网站中所有的jpg和png文件，但排除了所有的html和txt文件：

   `wget -r -l 2 --accept=jpg,png --reject=html,txt http://example.com/`

   另一个例子，该命令将下载网站中所有包含"news"关键字的html文件，但排除了所有的png文件：

   `wget -r -l 2 --accept=html --reject=png --accept-regex='news' http://example.com/`

5. **我要如何从multiple different domains下载页面？**

如果你要从多个不同的域名下载页面并控制下载哪些文件，你可以使用wget命令中的`--domains`选项指定下载的域名。

以下是一个示例命令，它将从三个域名（example1.com、example2.com、example3.com）下载页面：

`wget -r -l 2 --domains=example1.com,example2.com,example3.com http://example.com/`

“--exclude-domains"用于排除指定的域名，以避免递归下载这些域名下的页面和资源。

例如，使用以下命令可以递归下载指定网站及其链接指向的所有资源，但排除指定的域名（这里是example.com和example.org）：`wget -r -l 2 --exclude-domains example.com,example.org http://example.net/`

请注意，如果同时指定了"--exclude-domains"和"--domains"选项，则"--exclude-domains"选项将优先于"--domains"选项，即排除指定的域名将覆盖下载指定的域名。

请注意，如果你只想下载特定的域名而不是列出多个域名，你可以使用"--span-hosts"选项，它将允许wget跨越不同的域名进行递归下载。

`wget -r -l 2 --span-hosts http://example.com/`

React

虚拟DOM

在 React 中，通常使用 JSX 来描述组件的外观和结构。在组件被渲染时，React 会将 JSX 转换为虚拟 DOM 元素树，然后根据该虚拟 DOM 元素树来生成真实 DOM。

虚拟 DOM 是一个 JavaScript 对象，它描述了实际 DOM 元素的层次结构、属性和事件处理函数等信息。每当组件状态或属性发生变化时，React 会重新渲染虚拟 DOM，并使用算法比较新旧虚拟 DOM 的差异，然后仅更新有差异的部分的真实 DOM，从而提高了渲染效率。

React 是通过虚拟 DOM 来管理真实 DOM 的，通过比较虚拟 DOM 和上一次渲染的虚拟 DOM 的差异，来最小化真实 DOM 的操作次数，提高页面渲染的性能和效率。

以 React 组件的更新为例，当组件的状态或属性发生变化时，React 会生成一个新的虚拟 DOM 树，然后与上一次渲染的虚拟 DOM 树进行比较，找出它们之间的差异，这个过程叫做“协调”（reconciliation）。比较完成后，React 就知道了需要进行哪些操作来更新页面，比如添加、修改、删除节点等，这些操作会转化为真实 DOM 操作，更新页面的内容。

React 采用的是一种“声明式编程”的方式，开发者只需要关心页面需要展示的内容，而不需要手动管理页面的 DOM 元素。React 会根据开发者提供的虚拟 DOM 树来计算需要更新的部分，然后进行真实 DOM 的更新操作，这个过程是自动完成的，开发者不需要手动介入。

在 React 中，可以通过使用 JSX 和组件的方式来构建虚拟 DOM 树。每个组件对应着一个虚拟 DOM 树，组件的状态或属性发生变化时，React 会重新生成一个新的虚拟 DOM 树，并与之前的虚拟 DOM 树进行比较，然后计算出需要更新的部分，最终更新真实 DOM，实现页面的更新。

总之，React 中的 DOM 是由虚拟 DOM 生成的，虚拟 DOM 是一个 JavaScript 对象，用于描述实际 DOM 元素的层次结构、属性和事件处理函数等信息。我们通常不直接操作真实 DOM，而是通过状态和属性的改变来触发虚拟 DOM 的重新渲染。

虚拟DOM vs. 真实DOM

虚拟 DOM（Virtual DOM）和真实 DOM（Real DOM）是两种不同的 DOM（文档对象模型）表现形式。

真实 DOM 是浏览器提供的一种接口，开发者通过它可以操作网页中的各种元素。但是，当页面中的 DOM 发生变化时，浏览器需要重新计算元素的位置和样式，这个过程很耗费时间和性能，会导致页面卡顿、闪烁等问题。

虚拟 DOM 是使用 JavaScript 对象来描述页面上的元素和属性，它是一种轻量级的 DOM，因为它只存在于内存中，不需要和浏览器进行交互。当虚拟 DOM 中的元素发生变化时，React 会通过比较算法找出需要更新的元素，然后再将这些元素更新到真实 DOM 上，避免了频繁的 DOM 操作，提高了页面的性能。

虚拟 DOM 和真实 DOM 的主要区别可以总结如下：

1. **维护方式不同：虚拟 DOM 是通过 JavaScript 对象维护的，真实 DOM 是浏览器提供的接口。**
2. **操作方式不同：操作虚拟 DOM 只需要更新 JavaScript 对象，而操作真实 DOM 需要进行 DOM 操作，性能上有差异。**
3. **更新方式不同：虚拟 DOM 通过算法比较新旧虚拟 DOM 差异，只更新差异部分到真实 DOM，而真实 DOM 需要全部更新。**

在 React 中，虚拟 DOM 是 React 的核心机制之一，它可以大大提高应用的性能。由于虚拟 DOM 是 React 内部实现的机制，开发者只需要关注页面中的组件和数据，不需要直接操作虚拟 DOM，可以简化开发流程，提高代码的可维护性。

创建react项目

使用 npm 和 create-react-app 工具创建 React 项目的一般流程：

1.安装 Node.js 和 npm

Node.js 和 npm 可以在 Node.js 官网下载安装包，然后按照安装程序的步骤安装即可。

2.安装 create-react-app 工具

在命令行终端运行以下命令，使用 npm 安装 create-react-app 工具：

```
npm install -g create-react-app
```

这会在全局安装 create-react-app 工具，以便你可以在任何地方创建新的 React 项目。

3.创建 React 项目

在命令行终端进入要创建 React 项目的目录，并运行以下命令：

```
npx create-react-app my-app
```

其中 `my-app` 是你要创建的项目名称。运行这个命令会自动创建一个新的 React 项目，其中包含了所需的文件和文件夹结构。

4.启动开发服务器

进入项目文件夹，并运行以下命令来启动开发服务器：

```
cd my-app
npm start
```

这会启动开发服务器并在默认的 Web 浏览器中打开应用程序，你可以看到应用的初始界面。在运行 `npm start` 命令时，开发服务器将监视你对源代码的更改，并自动重新加载应用程序。

至此，你已经创建了一个基本的 React 应用程序，可以使用你的文本编辑器对其进行修改，并通过浏览器预览你的更改。

修改启动端口

在 React 项目中，可以通过修改 `package.json` 文件中的 `start` 命令来修改启动端口。打开 `package.json` 文件，找到 `scripts` 中的 `start` 命令，将其修改为：

```
"scripts": {
  "start": "PORT=3000 react-scripts start"
},
```

上述代码中将启动端口修改为 `3000`。你可以将其改为任意你想要的端口号。

修改完成后，通过命令行进入到项目目录下，执行 `npm start` 命令即可启动项目并监听修改后的端口号。

npm/npx

npm是Node.js自带的包管理工具，它的主要作用是用来安装和管理Node.js模块。npm有一个全局安装的机制，可以通过`npm install -g`安装全局模块，全局模块安装在系统的全局目录下，可以在命令行中直接使用。npm也可以用来初始化和管理项目的依赖，生成和管理项目的package.json文件，以及执行一些脚本。

npx是npm5.2.0版本之后新增的工具，用来执行临时安装的包。它的主要作用是解决了全局安装包可能导致的版本冲突问题，npx会在执行命令时临时下载安装所需的包，并执行它们，执行完成后就会被删除。在使用npx时，可以直接指定包的名称和版本号，不需要提前全局安装它们。npx还可以用来运行项目中安装的命令，而不需要使用全局安装的命令。

因此，可以认为npm是用来管理模块和依赖的，而npx是用来运行命令和工具的。

项目目录结构

React项目目录结构通常包含以下主要部分：

1. `public`文件夹：用于存放静态资源文件，如HTML文件、图片等。
2. `src`文件夹：包含了项目的源代码，其中最重要的是`index.js`，它是项目的入口文件。
3. `components`文件夹：用于存放React组件，每个组件通常包括JSX文件、CSS文件以及其他相关资源文件。
4. `pages`文件夹：用于存放项目中的页面，每个页面通常由多个组件组成。
5. `utils`文件夹：存放一些通用的工具函数或方法。
6. `services`文件夹：用于处理数据请求和响应，例如通过Axios发送HTTP请求。
7. `redux`文件夹：用于存放Redux相关的代码，包括actions、reducers、store等。

8. `routes` 文件夹：用于存放路由相关的代码，例如定义路由和路由组件等。
9. `styles` 文件夹：存放项目的公共样式文件，包括全局样式和局部样式。
10. `assets` 文件夹：用于存放项目的静态资源文件，如图片、音频等。

以上只是React项目目录结构的一种常见配置，具体可以根据项目需求进行调整。

1. `package.json` 是这个项目的配置文件，类似于Java/Maven中的 `pom.xml`。
2. 被服务的页面是 `public/index.html`，虽然在开发中 `%PUBLIC_URL%` 被替换为空字符串。这个页面的内容实际上只有一个 `<div id="root"></div>`，但是当 React 加载完成后会添加一行引用 JavaScript 文件的代码，将 App 组件的内容渲染到这个 `root` div 上。
3. 代码都在 `src/` 目录下，主要是 `src/App.js` 文件。默认的代码使用的是函数式组件，而不是类组件，但是在 JSX 代码中看到的元素就是你在浏览器中在 `root` div 内看到的元素。

node_modules

在 React 项目中，`node_modules` 是一个包含所有依赖库的文件夹。当你在创建 React 项目时，React 会自动为你安装一些必要的依赖库，比如 React、React-DOM、Babel 等。这些依赖库都会被安装到 `node_modules` 文件夹中，你可以在代码中引入这些库。

在开发过程中，你可能需要安装其他的依赖库，比如 React-Router、Redux 等。安装这些库的方式是使用 npm 或 yarn，这些库也会被安装到 `node_modules` 文件夹中。

`node_modules` 文件夹的作用类似于一个本地仓库，存储了项目所需的所有依赖库。这样做的好处是可以使得项目依赖的库互不干扰，也便于管理和维护。同时，如果你需要部署你的 React 项目到其他机器上，只需要将项目代码和 `node_modules` 文件夹一起复制到目标机器上即可。

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
      <App></App>
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

`import React from 'react';`：导入 React 库。

`import ReactDOM from 'react-dom/client';`：导入 ReactDOM 库的 client 模块, for handling the virtual DOM。

`import './index.css';`：导入项目的 CSS 样式表。

`import App from './App';`：导入 App.js 文件中默认导出的组件 App。

`import reportWebVitals from './reportWebVitals';`：导入 reportWebVitals.js 文件中默认导出的 reportWebVitals 函数。

`const root = ReactDOM.createRoot(document.getElementById('root'));`：获取名为 "root" 的 DOM 元素，并用 createRoot 方法创建一个渲染根。

`root.render( ... );`：渲染 root 的内容。在 StrictMode 组件下，将 App 组件和 Test 组件渲染到 root 中。

`reportWebVitals();`：调用 reportWebVitals 函数。

`<React.StrictMode>`是React 16.3新增的一个组件，它是用来检测应用程序中潜在问题的工具。

`<React.StrictMode>`可以帮助我们识别出应用程序中的潜在问题，并给出一些提示，以便更好地进行调试和修复。它会检查一些常见的问题，比如：

- 识别不安全的生命周期函数使用
- 检查已弃用的API的使用情况
- 检查警告（例如关于使用过时的API的警告）是否被正确地传递给上游组件

在代码中，`<React.StrictMode>`是一个组件，用于将子组件包裹起来。在上述代码中，`<React.StrictMode>`包裹了`<App>`和`<Test>`两个组件，表示在这两个组件内部启用严格模式，从而帮助我们识别潜在的问题。

Components

React 中的 class 组件和 function 组件都是用来定义组件的方式，但是它们之间存在一些区别。

使用 class 组件时，需要继承自 React.Component，并实现 render 方法来定义组件。在 render 方法中，通过返回 JSX 语法来表示组件的结构和渲染逻辑。class 组件可以管理状态（state），并在状态变化时重新渲染。

使用 function 组件时，直接使用函数来定义组件。在函数体内部，通过返回 JSX 语法来表示组件的结构和渲染逻辑。function 组件可以接收 props 作为参数，并使用 props 来控制组件的渲染结果。function 组件不能管理状态，但可以使用 React Hooks 来实现类似的效果。

因此，class 组件和 function 组件的主要区别在于管理状态的能力和代码复杂度。如果组件需要管理状态或者逻辑比较复杂，建议使用 class 组件。如果组件只需要简单的渲染逻辑，并且不需要管理状态，可以使用 function 组件。

```
class NameGreeter extends React.Component {
  render() {
    if (this.props.name === "") {
      return (
        <p>Hello!</p>
      )
```

```
    } else {
      return (
        <p>Hello, {this.props.name}!</p>
      )
    }
  }
}

<!-- 等价于 -->

function NameGreeter(props) {
  if (props.name === "") {
    return (
      <p>Hello!</p>
    )
  } else {
    return (
      <p>Hello, {props.name}!</p>
    )
  }
}
```

这个函数组件接收一个名为 `props` 的参数，并根据 `props.name` 的值来渲染不同的内容。

将 React 组件从类组件重写为函数组件通常会导致以下变化：

1. **组件定义方式：类组件使用 class 关键字，而函数组件使用 function 或箭头函数。**
2. **组件的生命周期方法：函数组件没有生命周期方法，只能使用 React Hooks。**
3. **组件内部的 this：在类组件中，可以使用 this 关键字来访问组件内部的状态和属性，但是在函数组件中无法使用 this，需要使用 React Hooks 中提供的 useState、useEffect 等 Hooks 来访问组件状态和生命周期方法。**

在这个例子中，我们将 `render()` 方法改为了一个普通的函数，同时去掉了类定义和 `this` 关键字。由于函数组件没有生命周期方法，因此不需要定义 `constructor` 或 `componentDidMount` 等方法。

导出组件

```
// MyComponent.js

export default class MyComponent extends React.Component {
  // ...
}


// App.js

import MyComponent from './MyComponent';

function App() {
  return (
    <div>
```

```
      <MyComponent />
    </div>
  );
}

export default App;
```

在上面的例子中，`MyComponent` 类被定义在 `MyComponent.js` 文件中，并通过 `export default` 导出为一个模块。在 `App.js` 文件中，使用 `import MyComponent from './MyComponent'` 语句导入了 `MyComponent` 模块，并在组件中使用了 `MyComponent` 类。最后，通过 `export default` 导出了 `App` 组件。

export default

在 React 中，`export default` 可以用于导出以下几种内容：

1. **函数组件**

```
function MyComponent() {
  // ...
}

export default MyComponent;
```

2. **类组件**

```
class MyComponent extends React.Component {
  // ...
}

export default MyComponent;
```

3. **变量、对象、函数等**

```
const message = 'Hello, world!';
export default message;

export const PI = 3.14;
export function add(a, b) {
  return a + b;
}
```

需要注意的是，一个模块中只能使用一次 `export default`，但可以使用多次 `export` 导出多个变量、对象、函数等。在导入时，可以使用 `import` 加花括号 `{}` 导入多个变量，或者直接导入默认导出的内容，例如：

```
// 导入多个变量,当你导入非默认导出时，你必须使用它的确切名称。
import { PI, add } from './utils';

// 导入默认导出的内容
import message from './message';
```

export和export default

`export` 和 `export default` 是用于将模块中的内容导出的关键字，它们的区别在于：

1. `export` 可以导出多个内容，每个导出的内容都需要使用 `{}` 括起来，使用时需要使用相同的名字，例如：`export { a, b, c }`，而 `export default` 只能导出一个默认值，使用时可以使用任意的名字，例如：`export default myFunction`。
2. 导入时，`export` 导出的变量需要使用 `{}` 括起来，并使用相同的名字，例如：`import { a, b, c } from './myModule'`，而 `export default` 导出的变量可以使用任意名字，并且不需要使用 `{}` 括起来，例如：`import myFunction from './myModule'`。

例如，在一个 `math.js` 模块中，使用 `export` 和 `export default` 分别导出模块中的内容：

```
export const PI = 3.14;

export function add(a, b) {
  return a + b;
}

export default function subtract(a, b) {
  return a - b;
}
```

在另一个模块中，使用 `import` 导入这些内容：

```
import { PI, add } from './math';
import mySubtractFunction from './math';

console.log(PI); // 3.14
console.log(add(1, 2)); // 3

console.log(mySubtractFunction(5, 2)); // 3
```

需要注意的是，`export default` 只能导出一个默认值，而 `export` 可以导出多个值，包括函数、类、常量、变量等。

生命周期

React 组件的生命周期是指从组件实例化、渲染到销毁整个过程中会发生的一系列事件和方法调用。

React 组件的生命周期分为三个阶段：Mounting(挂载)、Updating(更新) 和 Unmounting(卸载)。

挂载阶段(Mounting)

在 React 中，组件挂载指的是组件第一次被创建并添加到 DOM 中的过程

1. **constructor(): 组件的构造函数，用于初始化组件的状态和绑定方法；**
2. **static getDerivedStateFromProps(): 从组件的属性(props)中派生出 state，返回更新后的 state，或者返回 null 表示不更新；**
3. **render(): 渲染组件，返回组件的 HTML，React 会根据这个 HTML 创建真正的 DOM 元素；**
4. **componentDidMount(): 组件挂载后的生命周期函数，可以在这里请求数据、添加监听器等操作。**

这个过程分为两个阶段：

1. **`constructor` 阶段：组件实例被创建，但还未被添加到 DOM 中；**
2. **`componentDidMount` 阶段：组件实例被添加到 DOM 中。**

在组件挂载的过程中，React 会依次调用以下方法：

1. **`constructor` 构造函数；**
2. **`static getDerivedStateFromProps` 静态函数（可选）；**
3. **`render` 函数；**
4. **`componentDidMount` 生命周期函数。**

更新阶段(Updating)

当组件挂载完成后，组件会进入到更新阶段，此时若发生组件状态的变化或父组件重新渲染，React 会触发组件更新，即重新调用 `render` 函数，并依次调用 `shouldComponentUpdate`、`getSnapshotBeforeUpdate` 和 `componentDidUpdate` 生命周期函数。

1. **static getDerivedStateFromProps(): 同挂载阶段，当组件接收到新的属性(props)时触发；**
2. **shouldComponentUpdate(): 当组件的状态(state)或属性(props)发生变化时，判断是否需要重新渲染组件，返回 true 或 false；**
3. **render(): 同挂载阶段；**
4. **getSnapshotBeforeUpdate(): 组件更新之前触发，可以用来保存组件更新之前的状态；**
5. **componentDidUpdate(): 组件更新后触发，可以在这里操作 DOM 元素，发起请求等操作。**

卸载阶段(Unmounting)

1. **componentWillUnmount(): 组件卸载时触发，可以在这里清除定时器、取消订阅等操作。**

React 还提供了一些其它的生命周期函数，包括错误处理和子组件更新等，具体可以参考官方文档。需要注意的是，在 React 16 版本之后，一些生命周期函数被废弃了，可以使用其它替代方法代替，具体可以参考官方文档。

一次调用和多次调用

在 React 组件生命周期中，有一些函数是只会被调用一次的，也有一些函数是可以被多次调用的。其中 `render()` 方法会在每次组件发生变化时被调用，比如组件的初始化挂载和更新时。而 `constructor()` 方法只会在组件创建时被调用一次，它用于设置组件的默认值。

以下是只会被调用一次的生命周期函数：

1. `constructor`：组件的构造函数，在组件实例化时只会调用一次。
2. `componentDidMount`：组件挂载后的生命周期函数，在组件挂载到 DOM 中后只会调用一次。
3. `componentWillUnmount`：组件卸载前的生命周期函数，在组件卸载前只会调用一次。
4. `getDerivedStateFromProps`：从组件的属性(props)中派生出 state，每次组件接收新的属性时都会调用，但在组件挂载时也会调用一次。

以下是可以被多次调用的生命周期函数：

1. `shouldComponentUpdate`：组件更新前的生命周期函数，每次组件更新时都会调用，可以通过返回 true 或 false 决定是否需要重新渲染组件。
2. `getSnapshotBeforeUpdate`：组件更新前的生命周期函数，在组件更新之前每次调用，可以用来保存组件更新之前的状态。
3. `componentDidUpdate`：组件更新后的生命周期函数，在组件更新之后每次调用。

需要注意的是，在 React 16 版本之后，一些生命周期函数被废弃了，具体可以参考官方文档。

示例

我们以一个计数器组件为例来说明 React 的生命周期。

首先，我们先来实现这个计数器组件：

```
import React from 'react';

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    console.log('Counter constructor');
  }

  componentDidMount() {
    console.log('Counter componentDidMount');
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('Counter shouldComponentUpdate');
    return true;
  }

  componentDidUpdate(prevProps, prevState) {
    console.log('Counter componentDidUpdate');
  }

  componentWillUnmount() {
    console.log('Counter componentWillUnmount');
```

```
  }

  handleClick = () => {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }

  render() {
    console.log('Counter render');
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

在这个组件中，我们实现了以下生命周期方法：

- `constructor`：**构造函数，用于初始化组件的状态和绑定方法。**
- `componentDidMount`：**组件挂载后的生命周期函数，可以在这里请求数据、添加监听器等操作。**
- `shouldComponentUpdate`：**组件更新前的生命周期函数，用于判断组件是否需要更新，返回 true 或 false。**
- `componentDidUpdate`：**组件更新后的生命周期函数，可以在这里操作 DOM 元素，发起请求等操作。**
- `componentWillUnmount`：**组件卸载前的生命周期函数，可以在这里清除定时器、取消订阅等操作。**

我们在这些生命周期方法中加入了 `console.log`，可以看到它们在组件的生命周期中被调用的情况。

示例1

```
import './App.css';
import React from 'react';
import NumButton from "./NumButton";
import Counter from "./Counter";

class App extends React.Component {
  render() {
    return (
        <div>
            <Counter/>
        </div>
    );
  }
}
```

```
export default App;
```

运行这个项目后，控制台会输出以下内容：

```
<!-- 项目启动或页面刷新时 -->
Counter constructor
Counter render
Counter componentDidMount
<!-- 点击按钮后 -->
Counter shouldComponentUpdate
Counter render
Counter componentDidUpdate
```

这是因为当 `App` 组件被渲染到 DOM 中时，它会渲染 `Counter` 组件，此时会调用 `Counter` 组件的构造函数 `constructor`，所以会输出两次 `Counter constructor`，因为在 `App` 组件中只渲染了一个 `Counter` 组件。

接着，`Counter` 组件被渲染到 DOM 中时，会调用 `render` 函数，此时会输出 `Counter render`。紧接着，`componentDidMount` 生命周期方法会被调用，输出 `Counter componentDidMount`。

注意，因为在项目中使用了 `ReactDOM.createRoot` 来渲染 `App` 组件，所以控制台输出的顺序可能会有所不同。

React.StrictMode中双重渲染

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
      <App></App>
  </React.StrictMode>
);

reportWebVitals();
```

使用严格模式后的输出：

```
<!-- 项目启动或页面刷新时 -->
Counter constructor
Counter constructor
Counter render
Counter render
Counter componentDidMount
Counter componentWillUnmount
Counter componentDidMount
<!-- 点击按钮后 -->
```

```
Counter shouldComponentUpdate
Counter shouldComponentUpdate
Counter render
Counter render
Counter componentDidUpdate
```

React 严格模式会在开发环境下检测不安全的生命周期函数和函数式组件中的不安全用法，以便帮助开发人员进行调试和排查问题。因此，在严格模式下，React 会在多个生命周期函数中触发两次组件的更新，以确保不安全用法被正确地检测出来。

具体来说，在严格模式下，React 在渲染过程中会启用两个渲染器并同时进行渲染，以便检测到任何可能的问题。

渲染阶段生命周期方法

从 React 16.3 开始，React 官方文档中将 `componentWillMount`、`componentWillReceiveProps`、`componentWillUpdate` 都标记为过时，并且在未来版本中可能会被移除。在现代 React 应用中，可以使用其它替代方法来代替这些生命周期函数。

React 16.3 渲染阶段生命周期包括以下 class 组件方法

- **constructor()**
- **static getDerivedStateFromProps**
- **shouldComponentUpdate**
- **render()**
- **setState 更新函数（第一个参数）**
- **getSnapshotBeforeUpdate**
- **componentDidUpdate**

其中，`componentWillMount`、`componentWillReceiveProps`、`componentWillUpdate` 可以使用 `componentDidMount`、`static getDerivedStateFromProps`、`shouldComponentUpdate`、`render`、`setState`、`getSnapshotBeforeUpdate`、`componentDidUpdate` 等方法来替代。

在 React 18.2 版本中，渲染阶段的生命周期包括以下 class 组件方法

- **constructor()**
- **getDerivedStateFromProps**
- **shouldComponentUpdate**
- **render()**
- **setState 更新函数（第一个参数）**

这个列表与之前 React 16 版本不同，去除了 componentWillMount、componentWillReceiveProps 和 componentWillUpdate 三个生命周期函数，也没有 UNSAFE_componentWillMount、UNSAFE_componentWillReceiveProps 和 UNSAFE_componentWillUpdate 等同名的 UNSAFE 版本。

示例2

下面是一个使用计数器组件的示例：

```
import React from 'react';
import Counter from './Counter';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      showCounter: true
    };
    console.log('App constructor');
  }

  componentDidMount() {
    super.componentDidMount();
    console.log('App componentDidMount');
  }

  handleClick = () => {
    this.setState(prevState => ({
      showCounter: !prevState.showCounter
    }));
  }

  render() {
    console.log('App render');
    return (
      <div>
        <button onClick={this.handleClick}>Toggle Counter</button>
        {this.state.showCounter && <Counter />}
      </div>
    );
  }
}

export default App;
```

在这个示例中，我们通过点击按钮来控制是否显示计数器组件，可以看到组件的生命周期方法被调用的情况。

在初始渲染时，以下生命周期方法被调用：

- `App constructor`
- `App render`
- `Counter constructor`
- `Counter render`
- `Counter componentDidMount`

- `App componentDidMount`

当我们点击按钮隐藏计数器组件时，以下生命周期方法被调用：

- `Counter componentWillUnmount`

当我们再次点击按钮显示计数器组件时，以下生命周期方法被调用：

- `Counter constructor`
- `Counter render`
- `Counter componentDidMount`

当我们点击计数器组件中的增加按钮时，会触发 `handleIncrement` 方法。在这个方法中，我们会调用 `setState` 方法来更新组件的 `count` 状态。

在调用 `setState` 方法后，React 会触发更新阶段。首先，React 会调用 `shouldComponentUpdate` 方法来判断组件是否需要更新。在这个例子中，我们没有重写 `shouldComponentUpdate` 方法，因此默认情况下 React 会认为组件需要更新。如果我们重写了 `shouldComponentUpdate` 方法，并且返回 `false`，那么 React 将会跳过更新过程。

接下来，React 会调用 `render` 方法来重新渲染组件。在 `render` 方法中，我们会根据更新后的 `count` 状态来生成新的 JSX，React 会将新的 JSX 转换为真正的 DOM 元素，并将其更新到页面中。

最后，React 会调用 `componentDidUpdate` 方法来完成组件更新过程。在这个方法中，我们可以访问到更新前和更新后的组件状态，并且可以执行一些其他操作，比如发送请求或更新组件之外的 DOM 元素。

总的来说，当我们点击计数器组件中的增加按钮时，React 会依次执行以下方法：

1. **`handleIncrement` 方法；**
2. **`setState` 方法；**
3. **`shouldComponentUpdate` 方法（如果有）；**
4. **`render` 方法；**
5. **`componentDidUpdate` 方法。**

以上是 React 组件的生命周期中的一部分，具体可以根据实际情况选择实现需要的生命周期方法来完成组件的开发。

JSX

JSX（JavaScript XML）是 React 中的一种语法扩展，它允许在 JavaScript 代码中直接书写类似于 HTML 的标记，以声明式地描述应用程序界面。这样可以使得开发者更加方便地构建 UI，并且代码可读性更高。

在 JSX 中，我们可以使用 HTML 标签、React 组件以及自定义组件等，同时也可以在标签中添加属性和事件等，这些属性和事件的命名方式与 HTML 中的相似，但也有一些不同之处，例如 React 中使用 `className` 代替 HTML 中的 `class`。

下面是一个简单的 JSX 代码示例：

```
import React from 'react';

function App() {
  return (
    <div className="app">
      <h1>Welcome to my app</h1>
      <p>Here is some content</p>
    </div>
  );
}
```

在上面的代码中，我们使用 `import` 关键字导入了 React 模块，并创建了一个名为 `App` 的函数组件，返回一个包含 `div`、`h1` 和 `p` 标签的 JSX 代码。其中，`className` 属性被用来设置 `div` 标签的样式类名。

JSX中有一些属性名与JavaScript中的关键字不同，需要特别注意，例如：

- `className`：用于指定DOM元素的class，因为`class`是JavaScript中的关键字，所以在JSX中需要使用`className`来代替。
- `htmlFor`：用于指定表单元素的`id`属性，因为`for`是JavaScript中的关键字，所以在JSX中需要使用`htmlFor`来代替。
- `tabIndex`：用于指定DOM元素的tab键顺序，因为`tabindex`是JavaScript中的关键字，所以在JSX中需要使用`tabIndex`来代替。

需要注意的是，在React中，如果希望为DOM元素添加其他属性，可以使用小写字母的属性名，例如`data-`开头的属性可以通过`data-`+属性名的方式添加。

<> 或 React.Fragment

当定义组件时，必须有一个根元素。在某些情况下，可以使用<React.Fragment>或简写的<>作为根元素，以避免多余的代码块。

以下代码使用React.Fragment作为根元素，返回多个代码块，而不会在DOM中添加额外的标记。

```
function MyComponent() {
  return (
    <React.Fragment>
      <h1>Hello</h1>
      <p>World</p>
    </React.Fragment>
  );
}
```

jsx中双大括号的作用

在 React 中，组件的 JSX 语法中，样式的定义需要使用双大括号（`{{ }}`），因为第一层大括号用于将 JSX 转换为 JavaScript 对象，第二层大括号用于表示 JavaScript 对象的字面量形式，即样式对象的定义。

例如，要为一个元素添加样式，可以使用 `style` 属性，如下所示：

```
<div style={{ color: 'red', fontSize: '20px' }}>Hello, World!</div>
```

另外，还有一些其他属性也需要使用双大括号的，如 `className`、`onMouseDown`、`onMouseUp` 等事件处理属性。这是因为这些属性需要传递 JavaScript 代码，而双大括号可以使其成为 JavaScript 表达式。例如：

```
<button className={isActive ? 'active' : ''} onMouseDown={() => console.log('Mouse Down')}>Click Me</button>
```

在上面的例子中，`className` 属性使用了双大括号将表达式 `{isActive ? 'active' : ''}` 包裹起来，使其成为了一个 JavaScript 表达式。同时，`onMouseDown` 属性使用了箭头函数作为事件处理函数的定义。

需要注意的是，并不是所有的属性都需要使用双大括号，只有需要传递 JavaScript 代码的属性才需要。大多数属性，如 `id`、`name`、`value` 等等，只需要使用字符串即可。

举例来说，当需要将一个变量的值作为 `className` 属性的一部分时，需要使用大括号：

```
const myClass = "highlight";
const myElement = <div className={myClass}>Hello World</div>;
```

而当需要设置一个固定的属性值时，则不需要大括号：

```
const myElement = <input type="text" name="email" />;
```

需要注意的是，`style` 属性是一个例外。即使只需要设置一个固定的样式属性，也需要使用双大括号将样式对象包裹起来。例如：

```
const myStyle = { color: "red", fontSize: "16px" };
const myElement = <div style={myStyle}>Hello World</div>;
```

属性名转换为连字符格式

当在 JSX 中使用内联样式时，React 会将样式对象的属性名转换为连字符格式，并将属性值转换为字符串，生成一个包含 CSS 样式规则的字符串，例如：

```
const styles = {
  backgroundColor: "red",
  fontSize: "20px"
}

function MyComponent() {
  return <div style={styles}>Hello World</div>;
}
```

在此示例中，样式对象 `styles` 中的属性名 `backgroundColor` 和 `fontSize` 会被转换为 `background-color` 和 `font-size`，并且属性值会被转换为字符串。因此，最终生成的 HTML 代码如下：

```
<div style="background-color:red;font-size:20px;">Hello World</div>
```

这里的 `style` 属性值是一个字符串，包含了两条 CSS 样式规则，其中属性名用连字符格式表示，属性值用字符串表示。通过这种方式，React 可以将 JSX 中的样式对象转换为普通 HTML 中的样式规则，实现内联样式的效果。

Adding styles

在 React 项目中，对样式的应用可以与传统 HTML 相同，使用外部 CSS 文件、内联样式或 CSS 模块化等方式。但是由于 React 的组件化开发模式，有时候也会使用 CSS-in-JS 的方式，即将 CSS 代码直接嵌入到组件的 JavaScript 代码中。

在使用外部 CSS 文件的情况下，可以在 HTML 文件中通过 `<link>` 标签引入 CSS 文件，或者使用 CSS 预处理器（如 SASS 或 LESS）将多个 CSS 文件打包成一个文件。在 React 中使用外部 CSS 文件时，可以在组件的 JSX 中使用 `className` 属性来指定 CSS 类名。

在使用内联样式的情况下，可以在 JSX 中通过 `style` 属性设置样式。这里的 `style` 属性需要传入一个 JavaScript 对象，对象中的属性名为 CSS 属性名，属性值为 CSS 属性值。例如：

```
<div style={{ backgroundColor: 'red', fontSize: '20px' }}>Hello world!</div>
```

在使用 CSS 模块化的情况下，可以使用 webpack 等构建工具将 CSS 文件打包成一个独立的模块，并通过 `import` 关键字引入。这种方式可以避免全局 CSS 污染和命名冲突的问题。使用 CSS 模块化时，可以在组件的 JSX 中使用 `className` 属性来引用 CSS 模块。例如：

```
import styles from './styles.module.css';

function MyComponent() {
  return (
    <div className={styles.container}>
      <p className={styles.text}>Hello world!</p>
    </div>
  );
}
```

这里的 `styles.container` 和 `styles.text` 分别指的是 `styles.module.css` 文件中定义的 CSS 类名。

Conditional rendering

```
let content;
if (isLoggedIn) {
  content = <AdminPanel />;
} else {
  content = <LoginForm />;
}
return (
  <div>
    {content}
  </div>
);
```

```
<!-- or -->

<div>
  {isLoggedIn ? (
    <AdminPanel />
  ) : (
    <LoginForm />
  )}
</div>
```

Rendering list

在 React 中，Rendering lists 是指在渲染页面时，使用一个 JavaScript 数组映射为一个列表，可以使用 `map()` 函数来遍历数组，然后将数组中的每个元素转换成 React 组件。

例如，假设我们有一个包含商品信息的数组，我们想要将每个商品的标题显示在一个列表中：

```
const products = [
  { id: 1, title: 'Product 1' },
  { id: 2, title: 'Product 2' },
  { id: 3, title: 'Product 3' }
];

function ProductList() {
  const listItems = products.map(product =>
    <li key={product.id}>
      {product.title}
      style={{
        color: product.title === 'Product 1' ? 'magenta' : 'darkgreen'
      }}
    </li>
  );

  return (
    <ul>{listItems}</ul>
  );
}
```

在上面的例子中，我们首先定义了一个包含商品信息的数组 `products`，然后在 `ProductList` 组件中使用 `map()` 函数遍历 `products` 数组，并将每个商品转换为一个 `li` 元素。最后，我们将 `listItems` 渲染到一个 `ul` 元素中。

当 `products` 数组发生变化时，React 会重新渲染 `ProductList` 组件，并重新生成 `listItems` 数组，从而更新列表的内容。这样就可以轻松地将动态数据映射为一个列表，并随着数据的变化动态更新页面内容。

Responding to events

在 React 中，事件处理程序是响应事件的函数。当用户与组件交互时（例如，单击按钮或输入表单字段），组件将触发一个事件，而事件处理程序将负责对事件进行响应。通过编写事件处理程序，您可以更改组件状态，执行异步操作或调用其他函数等。

React中的事件处理程序与原生JavaScript事件处理程序不同，因为它们使用了合成事件（SyntheticEvent）和事件池（Event Pool），以提高性能并避免常见的问题，例如跨浏览器兼容性问题和内存泄漏。

在 React 中，通过在组件上定义事件处理程序属性（例如 onClick，onChange，onSubmit 等），可以指定在发生特定事件时要调用的函数。这些属性与原生DOM事件属性类似，但使用 camelCase 命名约定而不是小写字母命名约定，并接受一个函数作为值。

SyntheticEvent

在 React 的事件处理函数中，事件对象 e 是一个合成事件对象（SyntheticEvent），它是基于原生的浏览器事件对象进行封装的。它提供了与原生事件对象相同的接口，但是它的实现是跨浏览器的，而且可以在组件中进行重用。

在这个例子中，e.target 表示被点击的 DOM 元素。具体来说，当按钮被点击时，React 会创建一个合成事件对象，其中包含了一些关于该事件的信息，例如事件类型、被点击的元素等。在 handleClick 函数中，我们通过 e.target 来获取被点击的元素，并将它输出到控制台中。

```
class Example extends React.Component {
  handleClick(e) {
    console.log(e.target); // 输出点击的 DOM 元素
  }

  render() {
    return (
      <button onClick={this.handleClick.bind(this)}>点击我</button>
    );
  }
}
```

函数组件不需要this绑定

在函数组件中，不需要使用 this 关键字和 bind() 方法来访问属性和方法，是因为函数组件本身就是一个函数，它的作用域已经被隐式地绑定了，可以直接访问函数内部的变量和函数。

下面是几个例子：

1. 访问 props 变量

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

在这个例子中，函数组件 Greeting 接收一个 props 参数，并直接在函数体内访问了 props.name 变量。

2. 定义内部状态

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

在这个例子中，函数组件 Counter 使用 useState 钩子定义了一个内部状态 count，并定义了一个 handleClick 函数，用来更新 count 状态。在 handleClick 函数中，可以直接访问 count 状态，而不需要使用 this 关键字。

总之，由于函数组件的特殊性质，可以直接在函数体内访问函数作用域内的变量和函数，而不需要使用 this 关键字或 bind() 方法。

class组件中的事件绑定

```
class App extends React.Component {
  constructor() {
    super();
    this.onNameChange = this.onNameChange.bind(this);
  }

  onNameChange(newName) {
        this.setState({name: newName})
  }

  // state and render method here, same as before
}
```

在 React 中，当你定义一个 class 组件时，你必须将 this 绑定到它的方法中。否则，当你尝试在该方法中使用 this 时，会出现 undefined 的错误。

这就是为什么代码中需要使用 `this.onNameChange = this.onNameChange.bind(this)` 的原因，因为它将组件的 `this` 绑定到了 `onNameChange` 方法中。

如果没有这行代码，当你在 `onNameChange` 方法中使用 `this.setState` 时，`this` 将是未定义的，因此 `setState` 也会失败。

```
import React, {Component} from 'react'

class Test extends React.Component {
```

```
    constructor (props) {
        super(props)
        this.state = {message: 'Allo!'}
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick (e) {
        console.log(this.state.message)
    }

    render () {
        return (
            <div>
                <button onClick={ this.handleClick }>Say Hello</button>
            </div>
        )
    }
}
```

这里的 `this.handleClick = this.handleClick.bind(this)` 是用来将 `handleClick` 方法中的 `this` 绑定到 `Test` 组件实例上的。这样做是因为在 `handleClick` 方法中，`this` 默认会指向触发事件的 DOM 元素，而我们希望在 `handleClick` 中能够访问到 `Test` 组件实例中的 `state` 属性，因此需要将 `this` 绑定到 `Test` 组件实例上。

如果没有这行代码，那么 `console.log(this.state.message)` 中的 `this` 会指向触发点击事件的 DOM 元素，而 `<button onClick={ this.handleClick }>Say Hello</button>` 中的 `this` 也会指向触发点击事件的 DOM 元素。这样会导致无法访问到 `Test` 组件实例中的 `state` 属性，从而无法更新组件的渲染结果。

```
import React, {Component} from 'react'

class Test extends React.Component {
    constructor (props) {
        super(props)
        this.state = {message: 'Allo!'}
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick (e) {
        console.log(this.state.message)
    }

    render () {
        return (
            <div>
                <button onClick={ this.handleClick }>Say Hello</button>
            </div>
        )
    }
}
```

在React中，类的方法默认情况下不会自动绑定this，如果直接将一个类方法作为回调函数传递给onClick，那么this将指向undefined，这通常会导致TypeError: Cannot read property 'state' of undefined错误。

但是，在React的构造函数中，我们可以手动绑定方法并将其作为实例属性。在这种情况下，我们可以将方法作为回调函数传递给onClick，React将自动绑定它的this指向组件实例。所以在这段代码中，虽然没有显式地绑定this，但是在React内部仍然对handleClick方法进行了绑定，因此能够正常输出Allo。

```
class Test extends React.Component {
    constructor (props) {
        super(props)
        this.state = {message: 'Allo!'}
    }

    handleClick (e) {
        console.log(this.state.message)
    }

    render () {
        return (
            <div>
                <button onClick={ ()=>{ this.handleClick() } }>Say Hello</button>
            </div>
        )
    }
}
```

在这个代码中， ()=>{ this.handleClick() } 是一个箭头函数，它的作用是将 this 指向 Test 类的实例，而不是 DOM 元素。这是因为箭头函数不会创建自己的 this，而是继承了外层作用域的 this。因此，通过这种方式可以确保在 handleClick 方法中，this 指向 Test 类的实例。

React中在JSX中传递的事件不是一个字符串，而是一个函数（如:onClick={this.handleClick}），此时onClick即是中间变量，所以处理函数中的this指向会丢失。解决这个问题就是给调用函数时bind(this)，从而使得无论事件处理函数如何传递，this指向都是当前实例化对象。
当然，如果不想使用bind(this)，我们可以在声明函数时使用箭头函数将函数内容返回给一个变量，并在调用时直接使用this.变量名即可。示例代码如下：

```
import React from 'react';
export default class Life extends React.Component{
    constructor(props){
        super(props);
        this.state = {
            count:4
        };
    }
    render(){
        var style = {
            padding:'10px',
            color:'red',
```

```
            fontSize:'30px'
        }
        return (
            <div style={style}>{/*注意js语法使用一个括号{}去表示,style使用两个括号,原因里面其实是一个对象*/}
                <p>React生命周期介绍</p>
                <button onClick={this.handleAdd}>无bind点击一下</button>
                <button onClick={this.handleClick.bind(this)}>有bind点击一下</button>
                <p>{this.state.count}</p>
            </div>
        )
    }
    //此时this指向是当前实例对象
    handleAdd = ()=> {
        console.log(this)
        this.setState({
            count:5
        })
    }
    handleClick(){
        this.setState({
            count:6
        })
    }
}
```

Using hooks

React Hook 是 React 16.8 版本引入的新特性，它可以让你在函数组件中使用 state 和其他 React 特性，而不需要将组件转换为 class。使用 Hook 可以使代码更简洁、易懂，并提高代码复用性。

在函数组件中使用 Hook 非常简单，只需要在函数组件的顶部使用 useState() 或其他 Hook 函数即可。例如，使用 useState() 来声明状态：

```
import React, { useState } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

上述代码中，useState() 返回一个数组，第一个元素为当前状态值，第二个元素为更新状态值的函数。在例子中，count 表示当前状态值，setCount 表示更新 count 值的函数。

除了 useState()，React 还提供了许多其他的 Hook，例如 useEffect()、useContext()、useReducer() 等等，它们分别用于处理副作用、使用上下文、使用 reducer 等场景。

需要注意的是，Hook 只能在函数组件的顶层使用，不可以在循环、条件语句等内部使用。此外，在使用 Hook 时也需要遵守其使用规则，例如在条件分支中使用 Hook 时必须使用相同的顺序调用它们，避免在多个 Hook 之间跳过调用。

useState

React 中的 useState() 是一个用于创建并更新 state（状态）的 Hook（钩子）函数。它通常用于函数式组件中，可以让组件拥有状态，从而实现数据的管理和传递。

在组件的顶层使用 useState hook 来声明状态变量。useState() 接收一个初始状态值，返回一个数组，包含两个元素：当前状态的值和一个更新状态的函数，我们可以使用这个函数来更新状态的值。通常，我们使用数组的解构语法来获取这两个值。

```
const [count, setCount] = useState(0);
//count是 当前状态的值
//setCount是 一个更新状态的函数
```

如果您将一个函数作为 useState 的初始状态值参数传递，它将在渲染期间调用，返回一个初始状态值。如果您需要一个初始化代码的值，请使用惰性初始化。

> 惰性初始化是一种延迟初始化的策略，即在需要时才进行初始化，而不是在一开始就初始化。这种策略可以减少不必要的计算和内存消耗，提高程序的性能。
>
> 在编程中，我们可以使用惰性初始化来延迟创建对象或变量，直到它们真正需要使用时才进行创建。例如，在使用 React 的 useState 钩子时，可以使用惰性初始化来延迟初始化状态，以避免不必要的计算和渲染。
>
> 下面是一个在 React 中使用惰性初始化的示例：
>
> ```
> import { useState } from 'react';
>
> function MyComponent() {
> const [myState, setMyState] = useState(() => {
>  // 惰性初始化，只有在第一次使用状态时才执行
>  console.log('Initializing state...');
>  return 0;
> });
>
> // 点击按钮后更新状态
> const handleClick = () => {
>  setMyState(myState + 1);
> };
>
> return (
>  
>    <h1>My State: {myState}</h1>
> ```

```
    <button onClick={handleClick}>Increment</button>
  </div>
);
}
```

在上面的示例中，我们使用了 useState 的第二个参数，即惰性初始化函数，它只在第一次渲染组件时执行，用于初始化状态。在每次状态更新时，惰性初始化函数不会再次执行。

更新状态值的函数由 useState 返回的数组的第二个元素提供，它接受一个新的状态值或一个函数，并将状态值更新为下一个状态。更新函数是异步执行的，且不保证立即更新状态。调用 set 函数后读取状态变量，您仍然会得到调用前的旧值。这是因为 React 会在渲染结束之前将所有的更新函数合并，并只渲染一次。

> 当我们调用 `set` 函数更新组件的状态时，React 并不会立即执行渲染操作。相反，它将所有的状态更新函数加入到一个队列中。在组件更新之前，React 会批量处理这些更新操作并计算出新的状态。最终，只有在更新队列中所有的操作都执行完毕后，React 才会执行组件的渲染操作。
>
> 因此，如果在调用 `set` 函数之后立即读取状态变量的值，你可能会得到调用之前的旧值。这是因为状态更新函数还没有被执行，新的状态还没有被计算出来。只有在整个渲染周期结束后，批量处理操作完成之后，才能得到最新的状态值。
>
> 下面是一个示例代码，用来说明这个过程：

```
function Counter() {
const [count, setCount] = useState(0);

console.log("Before update:", count); // 打印当前状态
setCount(count + 1);
console.log("After update:", count); // 依然打印当前状态，不是更新后的状态

return <div>Count: {count}</div>;
}
```

> 在这个示例中，当我们调用 `setCount` 函数更新状态时，由于更新操作还没有被执行，所以第二个 `console.log` 语句输出的仍然是调用之前的状态值。只有在整个组件渲染周期结束后，才能得到更新后的状态值。

React 可以跟踪更新状态值的变化，并在必要时重新渲染组件及其子组件。如果您提供的新值与当前状态值相同，则 React 将跳过重新渲染组件及其子组件。这是为了提高性能而进行的优化。在极少数情况下，React 可能仍需要在跳过子组件之前调用您的组件，但它应该不会影响您的代码。

React 在处理所有事件处理程序之后更新屏幕，以防止在单个事件中多次重新渲染。在某些情况下，您可能需要强制 React 更早地更新屏幕，例如在访问 DOM 时。您可以使用 flushSync 来实现这一点。

> 举个例子，假设你的React组件有一个按钮，当你点击它时，它会通过设置state来更新屏幕中的一些元素。但是，如果您想在更新屏幕之前访问某些DOM元素，那么您需要在调用setState之前使用flushSync函数强制更新屏幕。
>
> 下面是一个示例：

```
import React, { useState, flushSync } from 'react';

function MyComponent() {
```

```
const [count, setCount] = useState(0);

function handleClick() {
 flushSync(() => {
   // 访问DOM
   const element = document.getElementById('my-element');
   console.log('Element:', element);

   // 强制更新屏幕
   setCount(count + 1);
 });
}

return (
 <div>
   <button onClick={handleClick}>Click me</button>
   <p>You clicked the button {count} times.</p>
   <div id="my-element">Some element</div>
 </div>
);
}

export default MyComponent;
```

在这个例子中，我们在点击按钮时访问了一个DOM元素，并使用flushSync函数强制更新屏幕。这样，在设置新状态之前，我们可以在控制台中查看DOM元素。

在 React 中，只允许在渲染期间从当前渲染组件中调用 set 函数。如果在 render 函数以外的地方调用 set 函数，React 将放弃其输出，并立即尝试以新状态再次渲染它。

> **在 React 中，只有在渲染函数中（如函数组件的主体或类组件的 render 方法）中，才能调用 `set` 函数更新组件的状态。如果在其他地方（如类组件的生命周期方法、事件处理程序或普通函数）中调用 `set` 函数，React 会放弃该函数的输出，并立即尝试使用当前状态重新渲染组件。这可能会导致不可预期的结果。**
>
> **例如，以下代码尝试在组件外部的函数中调用 `set` 函数来更新状态：**

```
function MyComponent() {
const [count, setCount] = useState(0);

function handleClick() {
 setCount(count + 1); // This is invalid!
}

//不在render()内
return (
 <div>
   <p>Count: {count}</p>
   <button onClick={handleClick}>Increment</button>
 </div>
);
}
```

这是无效的，因为 `handleClick` 函数不是渲染函数，不能调用 `set` 函数。如果要在事件处理程序中更新状态，应该使用函数式更新（使用回调函数）或使用 `useReducer`。例如，以下代码使用函数式更新：

```
function MyComponent() {
const [count, setCount] = useState(0);

function handleClick() {
 setCount(prevCount => prevCount + 1); // This is valid!
}

return (
 <div>
   <p>Count: {count}</p>
   <button onClick={handleClick}>Increment</button>
 </div>
);
}
```

在这个例子中，我们使用函数式更新来更新状态，因为 `handleClick` 函数不是渲染函数。这将确保 React 在更新状态之前将其与先前的状态合并，从而避免意外的行为。在 React 中，只有在渲染函数中（如函数组件的主体或类组件的 render 方法）中，才能调用 `set` 函数更新组件的状态。如果在其他地方（如类组件的生命周期方法、事件处理程序或普通函数）中调用 `set` 函数，React 会放弃该函数的输出，并立即尝试使用当前状态重新渲染组件。这可能会导致不可预期的结果。

例如，以下代码尝试在组件外部的函数中调用 `set` 函数来更新状态：

```
function MyComponent() {
const [count, setCount] = useState(0);

function handleClick() {
 setCount(count + 1); // This is invalid!
}

return (
 <div>
   <p>Count: {count}</p>
   <button onClick={handleClick}>Increment</button>
 </div>
);
}
```

这是无效的，因为 `handleClick` 函数不是渲染函数，不能调用 `set` 函数。如果要在事件处理程序中更新状态，应该使用函数式更新（使用回调函数）或使用 `useReducer`。例如，以下代码使用函数式更新：

```
function MyComponent() {
const [count, setCount] = useState(0);
```

```
function handleClick() {
 setCount(prevCount => prevCount + 1); // This is valid!
}

return (
 <div>
   <p>Count: {count}</p>
   <button onClick={handleClick}>Increment</button>
 </div>
);
}
```

> 在这个例子中，我们使用函数式更新来更新状态，因为 `handleClick` 函数不是渲染函数。这将确保 React 在更新状态之前将其与先前的状态合并，从而避免意外的行为。

在严格模式下，React 将调用更新函数两次，以帮助您找到意外杂质。这是一个开发环境中的行为，不会影响生产环境。如果您的更新函数是纯的（即不依赖于外部变量或函数），则不应该有任何影响。其中一个调用的结果将被忽略。

调用set函数不会改变已执行的代码中的当前状态,只影响 useState 从下一个渲染开始返回的内容

```
function handleClick() {

 setName('Robin');

 console.log(name); // Still "Taylor"!

}
```

当使用对象或数组作为状态时，应该避免直接修改原始对象或数组，而应该使用一个新对象或数组来替换旧的状态对象。这可以通过使用展开运算符和合并对象来实现。例如，如果有一个在state里的form对象，应该使用新对象代替旧对象：

```
// Don't mutate an object in state like this:
form.firstName = 'Taylor';

//通过创建一个新对象来替换整个对象：
// Replace state with a new object
setForm({
  ...form,
  firstName: 'Taylor'
});
```

> *...form* 是 ES6 中的展开语法，它将 *form* 对象的属性和值展开为一个新的对象。当表单输入框的值发生变化时，使用展开语法创建一个新的对象并更新 *form* 对象中相应的属性，以确保 React 组件的状态始终是最新的。

避免在使用 React 的 useState hook 时重复调用初始值函数。

当你在使用 useState 声明一个状态变量时，你可以给它一个初始值。如果这个初始值是一个函数，那么这个函数只会在组件第一次渲染的时候被调用，而不会在组件重渲染时重复调用。

比如，下面的例子中，createInitialTodos 是一个函数，它在组件第一次渲染时被调用，它返回一个初始的待办事项列表。这个列表将作为状态变量 todos 的初始值。

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);

  function createInitialTodos() {
    return [
      { id: 1, text: 'Buy groceries', completed: false },
      { id: 2, text: 'Walk the dog', completed: true },
      { id: 3, text: 'Do laundry', completed: false },
    ];
  }

  // ...
}
```

但是，如果你在传递初始值函数的时候不小心把函数名加上了一对括号，那么实际上传递给 useState 的就不是一个函数了，而是这个函数的返回值。这样的话，在组件重渲染时，每次调用 useState 都会再次执行这个函数，这可能会导致不必要的性能开销。

比如，下面的例子中，在 createInitialTodos 后面加上了一对括号，这样 createInitialTodos 就被调用了，它返回的列表就作为 todos 的初始值。

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos()); // 注意这里的括号

  function createInitialTodos() {
    return [
      { id: 1, text: 'Buy groceries', completed: false },
      { id: 2, text: 'Walk the dog', completed: true },
      { id: 3, text: 'Do laundry', completed: false },
    ];
  }

  // ...
}
```

为了避免这个问题，你应该传递一个函数作为初始值，而不是在这个函数后面加上一对括号。这样，这个函数就不会在组件重渲染时被重复调用了。

示例

以下是一个使用 useState() 的示例：

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);
```

```
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

在这个例子中，useState(0) 的返回值 [count, setCount] 被解构为了两个变量 count 和 setCount。count 是当前状态值，初始值为 0。setCount 是用于更新 count 的函数，我们使用它来更新 count 的值。当用户点击按钮时，我们使用 setCount(count + 1) 更新 count 的值。

注意，使用 useState() 时，React 会自动保留状态值，并在状态值发生变化时重新渲染组件。这就是 React 的响应式特性所在，让我们可以轻松地更新 UI，而不需要手动处理 DOM 元素。

此外，useState() 还支持传递一个函数作为初始状态值，该函数会在组件初始化时执行一次，返回的值即为初始状态值。这对于需要根据其他变量计算初始状态值的场景非常有用。例如：

```
import React, { useState } from 'react';

function Counter({ initialValue }) {
  const [count, setCount] = useState(() => {
    return initialValue * 2;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

在这个例子中，我们在 useState() 中传递了一个函数作为初始状态值，这个函数会在组件初始化时执行一次，计算出一个初始状态值 initialValue * 2。

useState创建的setCount只能接收一个值，但是在使用时，可以在里面调用一个有返回值的函数

```
<button onClick={() => setCount(currentCount => {
  // 一些复杂的逻辑
  return currentCount + 1;
})}>
```

useEffect

用于处理函数式组件中的副作用，例如发起 AJAX 请求、添加和移除事件监听器等。useEffect 接受一个回调函数和一个数组参数，表示需要监控的变量，只有在这些变量发生改变时才会触发回调函数。例如：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

`useEffect` 函数可以传入两个参数，第一个参数是副作用操作的函数，第二个参数是一个可选的依赖项数组。

- **如果 useEffect 只传入一个参数，那么每当组件挂载、更新或卸载时，都会执行这个副作用操作。**
- **如果传入了第二个参数，那么只有当依赖项发生变化时才会执行副作用操作。**
  - **如果 useEffect 的第二个参数是一个空数组（[]），则第一个参数函数只会在组件挂载时执行一次，执行时机类似于 componentDidMount。**
  - **如果 useEffect 的第二个参数是一个包含变量的数组，那么只有在这些变量发生改变时，第一个参数函数才会执行，执行时机类似于 componentDidUpdate。**
  - **如果 useEffect 的第二个参数是一个函数，这个函数会在组件卸载时执行，执行时机类似于 componentWillUnmount。**

在 useEffect() 中的 return** 语句用于定义清理操作，该语句内的代码会在组件被卸载或重新渲染时执行，以清理组件中产生的遗留垃圾或资源，例如清除计时器、取消订阅等操作。清除函数是一个可选的函数，您可以根据需要选择是否使用它。**

`useEffect` 函数的完整用法如下：

```
useEffect(effectFunction, dependencyArray)
```

其中，`effectFunction` 是副作用操作的函数，`dependencyArray` 是依赖项数组。例如：

```
useEffect(() => {
  // 副作用操作
}, [dependency1, dependency2]);
```

在这个例子中，当 `dependency1` 或 `dependency2` 发生变化时，就会执行副作用操作函数。如果 `dependencyArray` 为空数组，则只有在组件挂载和卸载时才会执行副作用操作函数。

代码执行顺序

useEffect的return执行时机

1. **首先渲染，并不会执行useEffect中的 return**
2. **变量修改后，导致的重新render，会先执行 useEffect 中的 return，再执行useEffect内除了return部分代码。**
3. **return 内的回调，可以用来清理遗留垃圾，比如订阅或计时器 ID 等占用资源的东西。**

下面是一个示例代码，用于说明 useEffect() 中各部分代码的执行顺序：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // 定义 useEffect() 钩子函数
  useEffect(() => {
    console.log('执行 useEffect()');

    // 定义清理函数
    return () => {
      console.log('执行 useEffect() 的清理函数');
    };
  }, [count]); // 依赖 count 变量

  console.log('渲染组件');

  // 定义点击事件处理函数
  const handleClick = () => {
    console.log('执行 setCount()前');
    setCount(count + 1);
    console.log('执行 setCount()后');
  };

  // 渲染组件，当count ===2 时，卸载组件
    return(
        <div>
            <p>你点击了 {count} 次</p>
            <button onClick={handleClick}>点击增加</button>
        </div>
    );
}
```

在上述示例代码中，当组件首次渲染时，会先执行组件的渲染代码，然后执行 useEffect() 钩子函数内的代码，打印 "执行 useEffect()"。此时并不会执行 useEffect() 的清理函数。

- **初次渲染时控制台输出**

```
渲染组件
执行 useEffect()
//不会执行 useEffect() 的清理函数
```

当用户点击按钮，导致 count 变量发生改变，组件会重新渲染。此时，会先执行 useEffect() 的清理函数，打印 "执行 useEffect() 的清理函数"，然后再执行 useEffect() 钩子函数内的代码，打印 "执行 useEffect()"。

- **在用户点击按钮后，控制台会依次输出以下内容：**

```
执行 setCount()前
执行 setCount()后
渲染组件
//先执行 useEffect() 的清理函数
执行 useEffect() 的清理函数
//再执行 useEffect() 钩子函数内的代码
执行 useEffect()
```

如果在组件被卸载时，也会执行 useEffect() 的清理函数，以清理组件中产生的遗留垃圾或资源。

- **组件被卸载时**

```
执行 useEffect() 的清理函数
```

需要注意的是，由于 useEffect() 钩子函数是异步执行的，因此在渲染组件时，可能会先执行一些 useEffect() 钩子函数的清理函数，然后再执行其它 useEffect() 钩子函数的代码，具体的执行顺序可能会有所不同，需要根据实际情况进行分析和验证。

可以通过以下代码测试卸载UseEffectButton组件时的输出

```
import React, { useState } from 'react';
import UseEffectButton from './UseEffectButton';

function TestUseEffectButton() {
    const [isUnmounted, setIsUnmounted] = useState(false);

    const handleUnmount = () => {
        setIsUnmounted(true);
    };

    return (
        <div>
            <button onClick={handleUnmount}>卸载子组件</button>
            {!isUnmounted && <UseEffectButton />}
        </div>
    );
```

```
}

export default TestUseEffectButton;
```

示例

下面是一个使用 `useEffect` 清除函数的例子：

```
import React, { useState, useEffect } from 'react';

function UseEffectTimer() {
    const [count, setCount] = useState(0);

    // 这段代码第二个参数为空数组，因此不会在count更新时执行清理函数
    // useEffect(() => {
    //     const timer = setInterval(() => {
    //         console.log('Hello');
    //     }, 1000);
    //
    //     return () => {
    //         console.log('clear');
    //         clearInterval(timer);
    //     };
    // }, []);

    useEffect(() => {
        const timer = setInterval(() => {
            console.log('使用useEffect()');
        }, 1000);

        return () => {
            console.log('执行 useEffect() 的清理函数');
            clearInterval(timer);
        };
    }, [count]);

    const handleClick = () => {
        setCount(count + 1);
    };

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={handleClick}>Click me</button>
        </div>
    );
}

export default UseEffectTimer;
```

在这个例子中，我们使用了 `useState()` 钩子函数来创建一个状态变量 `count`，并在组件中渲染其当前值。我们还定义了一个按钮点击事件处理函数 `handleClick()`，每次点击按钮时将 `count` 的值加 1。

在组件的 `useEffect()` 钩子函数中，我们使用 `setInterval()` 方法创建了一个每隔一秒钟输出一次 "Hello" 的定时器，并在返回值函数中清除该定时器。由于依赖数组为空，钩子函数只会在组件首次渲染时执行一次，因此 "Hello" 会每隔一秒钟输出一次，直到组件卸载。

你可以在控制台中查看输出信息，并在页面上点击按钮来测试组件的重新渲染。

需要注意的是，如果副作用操作不需要清除，那么可以不返回清除函数。同时，如果副作用操作需要清除，但是没有返回清除函数，那么组件在卸载时可能会出现内存泄漏的问题。

useContext

用于从父组件向子组件传递数据，避免了 props 层层传递的繁琐过程。

使用 `useContext` 需要先创建一个 Context 对象，然后将这个 Context 对象传递给某个组件或应用程序的上下文。在子组件中使用 `useContext` 来获取这个 Context 对象，从而得到 Context 中存储的数据。

例如，假设有一个名为 `UserContext` 的 Context 对象，存储着当前用户的信息，可以通过以下方式在组件中使用它：

```
import React, { useContext } from 'react';

// 创建一个 Context 对象
const UserContext = React.createContext();

function App() {
  // 将用户信息传递给 UserContext.Provider
  return (
    <UserContext.Provider value={{ name: 'John', age: 30 }}>
      <User />
    </UserContext.Provider>
  );
}

function User() {
  // 使用 useContext 获取 UserContext 中存储的数据
  const user = useContext(UserContext);

  return (
    <div>
      <h2>{user.name}</h2>
      <p>{user.age}</p>
    </div>
  );
}
```

在上述代码中，`UserContext` 是一个 Context 对象，存储着当前用户的信息。在 `App` 组件中，使用 `UserContext.Provider` 将用户信息传递给下级组件。在 `User` 组件中，使用 `useContext` 获取 `UserContext` 中存储的数据，并进行渲染。

需要注意的是，`useContext` 只能用于函数式组件中，如果要在类组件中使用 Context，则需要使用 `static contextType` 或 `Consumer`。

useReducer

在 React 应用中，reducer 主要是用于管理组件的状态(state)。它是一个纯函数，接收旧的 state 和一个 action，返回新的 state。

使用 reducer 首先要定义一个 reducer 函数，它接收两个参数，分别是当前的状态和一个 action 对象，然后根据 action 的 type 属性来更新状态，最后返回新的状态。

举个例子：

```
function counterReducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1
      };
    case 'DECREMENT':
      return {
        count: state.count - 1
      };
    default:
      return state;
  }
}
```

这是一个计数器的 reducer 函数，它有两个 action：INCREMENT 和 DECREMENT。分别对应计数器加一和减一的操作。

接下来，我们可以使用 useReducer 来创建一个计数器组件，并将上面定义的 reducer 函数作为参数传入：

```
import React, { useReducer } from 'react';

function Counter() {
  const [state, dispatch] = useReducer(counterReducer, { count: 0 });

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
}
```

在这个例子中，我们使用了 useReducer 来创建了一个计数器组件。在组件中，我们首先调用了 useReducer，传入了我们定义的计数器 reducer 函数和初始状态 { count: 0 }。

接下来，我们在组件中返回了一个 div 元素，包含一个 h1 元素和两个 button 元素。当点击增加或减少按钮时，我们使用 dispatch 方法分别调用了 INCREMENT 和 DECREMENT 两个 action。

当 dispatch 方法被调用时，React 会自动调用计数器 reducer 函数，并将旧的状态和 action 作为参数传入。根据 action 的类型，reducer 函数会返回一个新的状态，然后这个新的状态会被 React 存储，并更新组件的视图。

这就是使用 reducer 管理组件状态的基本过程。需要注意的是，reducer 必须是一个纯函数，也就是说，它不能改变原来的状态，而是要返回一个新的状态。同时，reducer 也必须是可预测的，也就是说，给定相同的输入，它应该返回相同的输出。

useRef

`useRef` 是 React 提供的一个 Hook 函数，用于创建一个可变的 ref 对象，可以用来存储组件中的一个可变值，该值在组件更新时不会改变，可以跨越组件生命周期存储数据。

使用 `useRef` 需要先通过调用该函数创建一个 ref 对象：

```
const myRef = useRef(initialValue);
```

其中 `initialValue` 是可选的初始值，可以为任何值，也可以省略不传。

然后可以将该 ref 对象赋值给组件中的一个元素或其他对象的属性，或者直接使用该 ref 对象来存储数据。

以下是一个简单的例子，使用 `useRef` 存储计时器的初始时间戳，并在组件更新时不改变该值：

```
import { useRef, useEffect, useState } from 'react';

function Timer() {
  const startTimeRef = useRef(Date.now());
  const [elapsedTime, setElapsedTime] = useState(0);

  useEffect(() => {
    const timerId = setInterval(() => {
      setElapsedTime(Date.now() - startTimeRef.current);
    }, 1000);

    return () => {
      clearInterval(timerId);
    };
  }, []);

  return (
    <div>
      <p>Start Time: {startTimeRef.current}</p>
      <p>Elapsed Time: {elapsedTime}</p>
    </div>
  );
}
```

在上面的例子中，我们通过 `useRef` 创建了一个名为 `startTimeRef` 的 ref 对象，并将当前时间戳赋值给该对象。在计时器的更新中，我们使用了 `startTimeRef.current` 来获取计时器的起始时间戳，而这个值在组件的更新过程中不会改变。

Sharing data between components

在 React 中，可以使用多种方式在多个组件之间传递和共享数据。下面是几种常见的方法：

1. Props：将数据作为 props 传递给子组件，使其能够在组件之间共享数据。这是 React 中最常见的方法。
2. Context：Context 是一种在组件树中共享数据的方式。它允许您通过使用 Provider 和 Consumer 组件在组件树中传递数据，而不必通过 props 一级一级地手动传递。
3. Redux：Redux 是一个流行的状态管理库，它允许您在整个应用程序中管理共享状态。
4. React Router：React Router 是一个流行的路由库，它允许您在不同的组件之间切换，并通过路由参数共享数据。

在实际开发中，您可能需要根据应用程序的特定需求选择不同的方法来传递和共享数据。

```
import { useState } from 'react';

export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```

这段代码通过使用 `useState` 钩子在父组件中创建了一个 `count` 状态，并通过将 `count` 和 `handleClick` 作为 props 传递给 `MyButton` 子组件来实现了组件之间的数据共享。在 `MyButton` 组件中，通过接收 `count` 和 `onClick` props，可以渲染出一个按钮，同时在按钮被点击时调用 `onClick` 回调函数来更新父组件的 `count` 状态。由于 `MyButton` 组件被父组件多次渲染，因此每个按钮的 `count` props 都是相同的，从而实现了组件之间的数据共享。

props

在 React 中，props 是指父组件传递给子组件的属性。props 可以包含任意类型的数据，包括基本数据类型、函数、对象等。

在React中，父组件可以向子组件传递数据和方法，但子组件无法直接访问祖先组件的数据和方法。如果需要访问Grand组件的属性或方法，可以通过将它们作为props传递给Father和Son组件来实现。

如果在Grand、Father和Son组件中有同名的属性或方法，React会从当前组件开始向上查找，找到第一个匹配的属性或方法并使用它。如果没有找到匹配的属性或方法，则会报错。如果需要在子组件中访问父组件中的同名属性或方法，可以使用别名来区分它们。例如，在Father组件中将Grand组件的属性名改为grandTitle，然后将其传递给Son组件，可以使用以下代码：

```
class Grand extends React.Component{
  /* 一些属性和方法 */
  render() {
    return (
      <Father grandTitle={this.props.title} />
    );
  }
}

class Father extends React.Component {
  /* 一些属性和方法 */
  render() {
    return (
      <Son fatherTitle={this.props.title} grandTitle={this.props.grandTitle} />
    );
  }
}

class Son extends React.Component {
  /* 一些属性和方法 */
  render() {
    return (
      <div>
        <h1>{this.props.grandTitle}</h1>
        <h2>{this.props.fatherTitle}</h2>
        <p>{this.props.content}</p>
      </div>
    );
  }
}
```

这样，Son组件就可以通过this.props.grandTitle和this.props.fatherTitle访问两个不同的属性值。

示例2

```
class Parent extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('Button clicked!');
  }

  render() {
    return (
      <div>
        <Child onClick={this.handleClick} />
      </div>
    );
  }
}

class Child extends React.Component {
  render() {
    return (
      <div>
        <button onClick={this.props.onClick}>Click me</button>
      </div>
    );
  }
}
```

在`Child`组件中，我们使用了`this.props.onClick`，这样就可以正确地调用`handleClick`方法，从而输出`Button clicked!`。如果我们尝试使用`this.handleClick`，则会出现错误，因为`Child`组件中并没有定义该方法。

在 React 中，props 是指父组件传递给子组件的属性。props 可以包含任意类型的数据，包括基本数据类型、函数、对象等。

在组件中，可以通过`this.props`访问 props 对象，如：

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.title}</h1>
        <p>{this.props.content}</p>
      </div>
    );
  }
}
```

在上面的例子中，`this.props.title`和`this.props.content`分别引用了父组件传递给子组件的`title`和`content`属性。

当然，如果父组件传递给子组件的属性中包含了函数类型的属性，那么子组件也可以通过 props 来引用这些函数，并在合适的时候调用它们。

在 React 中，父组件可以通过 props 向子组件传递任何类型的数据。而子组件也可以通过 props 来引用其父组件传递下来的属性或方法，不管这些属性或方法是来自一级父组件还是更深层次的父组件。

state

this.state 是 React 组件内部存储可变数据的地方。当组件状态改变时，React 会自动重新渲染组件，以确保它们反映最新的状态。在组件的生命周期方法和事件处理程序中，可以使用 this.setState() 方法更新组件状态。通过使用 this.setState() 方法，React 将比较先前的状态与新状态，计算出最小的 DOM 更新，并仅仅更新必要的部分，以提高性能和减少资源消耗。

React中，组件的状态(state)可以在构造函数(constructor)中初始化，并通过调用setState方法来更改。

初始化状态：

初始化state必须在构造函数内部进行。在构造函数外部声明和初始化state将导致React无法对其进行管理和更新。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
      </div>
    );
  }
}
```

在上面的代码中，MyComponent组件的状态(count)被初始化为0。

更改状态：

要更改组件的状态，需要调用setState方法。例如，下面的代码在组件被点击时增加状态count的值：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      count: this.state.count + 1
```

```
    });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}
```

在上面的代码中，handleClick方法调用了setState方法，以更新状态count的值。在render方法中，我们通过`{this.state.count}`来显示最新的状态值。

无限渲染

“无限渲染”（Infinite Render）问题通常是由于某个组件在渲染过程中不断地更新自身或调用父组件的函数而导致的，从而陷入了一个无限循环的状态。这种问题会严重影响应用程序的性能和稳定性，需要尽早发现和解决。在 React 开发中，可以通过使用 useEffect Hook 来避免这种问题的发生。

示例

一种常见的无限次重新渲染的问题是，当我们将一个函数作为 prop 传递给一个组件，并且在该组件内部将该函数作为事件处理函数绑定到元素上时，如果我们不小心传递了一个无限循环的函数，那么就会导致组件无限次重新渲染，从而造成程序崩溃。

例如，假设我们有一个 App 组件和一个 Square 组件，App 组件将一个 handleClick 函数作为 prop 传递给 Square 组件，并且 Square 组件将 handleClick 函数作为事件处理函数绑定到按钮元素上。如果我们在 handleClick 函数中调用了 setState 函数，那么就会导致组件无限次重新渲染，从而造成程序崩溃。

React 中，每个组件都有自己的生命周期，在组件挂载、更新、卸载等不同的阶段，会调用对应的生命周期方法。当组件的 state 或 props 发生变化时，React 会检查是否需要重新渲染该组件。如果需要重新渲染，React 会调用组件的 render 方法来生成新的虚拟 DOM，然后对比新旧虚拟 DOM 的差异并进行最小化的更新。

在上述的情况中，当 Square 组件接收到的 props 有变化时，React 会进行重新渲染该组件。但由于传递给 Square 组件的函数对象并不会重新创建，因此每次渲染时传递给 Square 组件的函数对象并没有发生变化，也就不会导致重新渲染。

以下是一个可能导致无限次重新渲染的示例代码：

```
function App() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
```

```
      <p>Count: {count}</p>
      <Square handleClick={handleClick} />
    </div>
  );
}

function Square({ handleClick }) {
  return (
    <button onClick={handleClick}>Click me</button>
  );
}
```

当我们使用 `handleClick` 函数时，实际上我们是将函数作为回调函数传递给 `onClick` 事件。而回调函数必须是一个函数对象，如果直接传递 `handleClick` 函数，那么每次渲染 Square 组件时都会调用 `handleClick` 函数，导致无限次重新渲染，进而导致报错。因此，我们需要将 `handleClick` 函数包装在一个无参的函数中，并将该函数作为 prop 传递给 Square 组件，这样在渲染 Square 组件时就不会直接调用 `handleClick` 函数，而是调用我们包装后的函数。

具体实现时，我们可以使用箭头函数，如下所示：

```
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
```

这里，我们将 `handleClick` 函数包装在了一个无参的箭头函数中，并将该函数作为 `onSquareClick` prop 传递给了 `Square` 组件，这样每次渲染 `Square` 组件时都会调用我们包装后的箭头函数，而不是直接调用 `handleClick` 函数，从而避免了无限次重新渲染的问题。

以下是修改后的示例代码：

```
function App() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <Square handleClick={() => handleClick()} />
    </div>
  );
}

function Square({ handleClick }) {
  return (
    <button onClick={handleClick}>Click me</button>
  );
}
```

```
...
<Square value={squares[0]} onSquareClick={() => handleClick(0)}/>
<Square value={squares[1]} onSquareClick={handleClick}/>
<Square value={squares[2]} onSquareClick={handleClick(0)}/>
...
```

第一行和第二行都是将一个无参的箭头函数作为props传递给Square组件，而这个箭头函数的作用是在Square组件内部调用handleClick函数，这样可以避免因为函数的重新创建而导致的无限渲染问题。

第二行代码的 `handleClick` 是一个函数对象，而不是函数的调用结果。当传递函数对象作为 props 时，React 会自动将其绑定到组件实例的 `this` 上，因此在组件内部使用 `this.props.handleClick` 调用该函数时不会导致无限渲染的问题。

而第三行代码中，onSquareClick prop 接收的是一个具体的函数调用，传递的是一个函数的调用结果 `handleClick(0)`，而不是函数对象，这就意味着每次渲染Square组件时，都会调用该函数，重新计算该值，导致了无限次的重新渲染。

Tic-Tac-Toe

```
import { useState } from 'react';

function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}

function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }

  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (xIsNext ? 'X' : 'O');
  }
```

```
  return (
    <>
      <div className="status">{status}</div>
      <div className="board-row">
        <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
        <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
        <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
      </div>
      <div className="board-row">
        <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
        <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
        <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
      </div>
      <div className="board-row">
        <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
        <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
        <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
      </div>
    </>
  );
}

export default function Game() {
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xIsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }

  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = 'Go to move #' + move;
    } else {
      description = 'Go to game start';
    }
    return (
      <li key={move}>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });
```

```
  return (
    <div className="game">
      <div className="game-board">
        <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
      </div>
      <div className="game-info">
        <ol>{moves}</ol>
      </div>
    </div>
  );
}

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

Cloud

Cloud Computing

Cloud Computing（云计算）是指基于互联网的一种计算模型，用户可以通过互联网访问可扩展的计算资源，如虚拟机、存储、应用程序和服务，而无需拥有这些资源的物理副本或在本地管理这些资源。简单来说，云计算就是将计算资源作为服务交付给用户，用户只需要按需使用并支付所消费的资源费用，而无需自己购买、配置和维护这些资源。

云计算具有以下特点：

1. **虚拟化：云计算资源通常是以虚拟化的方式提供，这意味着多个虚拟机可以在同一物理服务器上运行，从而提高硬件利用率。**
2. **弹性扩展：用户可以根据需要快速地增加或减少计算资源的数量，以适应业务需求的变化。**
3. **自助服务：用户可以自己管理和控制他们所使用的资源，包括创建、启动、停止和监控虚拟机等。**
4. **按需计费：云计算资源的费用通常是按使用时间和消耗量计算的，用户只需为所使用的资源支付相应的费用。**

云计算的优点包括：

1. **灵活性：云计算可以根据业务需求的变化快速地增加或减少计算资源。**
2. **可扩展性：云计算可以扩展到数百或数千台服务器，以满足大规模计算的需求。**
3. **可靠性：云计算提供了高可用性和容错性，能够确保业务连续性。**
4. **成本效益：云计算的按需计费模式可以减少硬件设备和维护费用，并提高IT资源的利用率。**
5. **全球性：云计算可以通过全球范围的数据中心提供服务，使用户可以在全球范围内访问资源。**

云计算在各个行业的应用也越来越广泛，例如电子商务、金融、医疗、制造业等。由于云计算的优点，越来越多的企业和组织选择使用云计算来支持他们的业务需求。

以下是一些常见的云计算应用实例：

1. **云存储：云存储是一种基于云计算的数据存储服务，可以帮助用户以高可用性、可扩展性和低成本的方式存储和管理数据。例如，Google Cloud Storage、Amazon S3 等。**
2. **云数据库：云数据库是一种基于云计算的数据库服务，可以帮助用户以高可用性、可扩展性和低成本的方式存储和管理数据。例如，Google Cloud SQL、Amazon RDS 等。**
3. **云计算资源租用：云计算资源租用是一种基于云计算的服务，可以帮助用户以弹性、按需和低成本的方式租用计算资源。例如，Amazon EC2、Google Compute Engine 等。**
4. **虚拟桌面：虚拟桌面是一种基于云计算的服务，可以帮助用户在云端管理和交付虚拟桌面。例如，Amazon WorkSpaces、Microsoft Windows Virtual Desktop 等。**
5. **云计算应用程序：云计算应用程序是一种基于云计算的应用程序，可以帮助用户在云端运行和管理应用程序。例如，Google Cloud Run、Amazon Elastic Beanstalk 等。**

这些都是云计算技术在实际应用中的例子，通过使用云计算，用户可以以弹性、可扩展和低成本的方式获取计算资源和服务，并将其应用于各种应用场景中。

Cloud Economics

Cloud Economics（云经济学）是指分析和优化云计算中成本、性能和可用性之间的平衡。随着云计算的普及和发展，企业和组织越来越关注在云计算平台上运行应用程序的成本和效益问题。Cloud Economics 旨在通过提供成本优化的策略、工具和技术，帮助组织最大限度地提高其在云中运行应用程序的价值，同时降低成本。

Cloud Economics 主要关注以下几个方面：

1. **成本管理：如何最小化云计算平台上的成本，包括服务器和存储等基础设施成本、网络和带宽费用、管理和监控等操作成本等。**
2. **性能优化：如何提高在云计算平台上运行的应用程序的性能，以提高用户体验和减少停机时间，包括通过自动化调度、负载平衡和容器化等技术来优化应用程序性能。**
3. **可用性：如何确保云计算平台上的应用程序始终可用，包括通过自动化监控、容错和备份等机制来保证应用程序的高可用性。**

在 Cloud Economics 的框架下，组织可以通过采用一系列策略和技术来降低在云计算平台上运行应用程序的成本、提高性能和可用性，并实现业务目标。这些策略和技术包括根据需求动态调整服务器数量、使用云端存储和容器化、使用自动化工具来优化应用程序性能等等。通过实施这些策略和技术，组织可以最大限度地提高在云计算平台上运行应用程序的价值，并同时降低成本。

以下是一些 Cloud Economics 的例子：

1. 成本管理：云计算提供了一种灵活的方式，可以根据实际使用情况调整资源，并按需付费。这样可以降低公司的成本，避免因为不必要的资源浪费而产生不必要的费用。
2. 弹性扩容：云计算提供了弹性扩容的能力，可以根据业务需求增加或减少资源。这样可以避免因为业务峰值期间资源不足而影响业务的情况，也可以在业务低谷期间减少资源使用，避免资源浪费。
3. 无需资本投资：使用云计算，企业不需要购买昂贵的服务器和网络设备，不需要投入大量的资本成本。这可以使企业在更短的时间内启动项目，并快速适应市场需求。
4. 全球化部署：云计算服务提供商可以在全球范围内提供数据中心和服务节点，可以轻松地将应用程序和数据部署到多个地理位置。这可以使企业更轻松地进入全球市场，并提高用户的响应速度和服务质量。
5. 高可用性和容错性：云计算服务提供商可以提供高可用性和容错性的服务，使企业的应用程序可以在多个数据中心和服务器上运行。这可以保障企业的业务连续性和数据安全性。

这些例子都是基于云计算的 Cloud Economics，通过云计算提供的灵活性、弹性扩容、无需资本投资、全球化部署以及高可用性和容错性等优势，帮助企业提高效率、降低成本、提升业务响应速度和质量等方面的表现。

As A Service

"As A Service" 是一种服务模式，即将某种服务提供给客户以供按需使用，客户可以通过网络访问这些服务。通常情况下，这些服务提供商会通过云计算和互联网来提供服务，使客户可以通过自己的设备访问这些服务，而无需拥有自己的硬件和基础架构。

一些常见的 "As A Service" 服务包括：

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)
- Backend as a Service (BaaS)
- Functions as a Service (FaaS)
- Database as a Service (DBaaS)
- Identity as a Service (IDaaS)
- Security as a Service (SECaaS)
- Monitoring as a Service (MaaS)

这些服务提供商可以提供许多不同的服务，例如计算、存储、网络、数据库、身份验证、安全和监控等，使客户可以按需使用这些服务，而不需要自己拥有和维护硬件和软件基础架构。

IaaS, PaaS, and SaaS are three different types of cloud computing services.

IaaS、PaaS 和 SaaS 是云计算服务的三种不同类型。

- IaaS (Infrastructure as a Service): IaaS provides users with virtualized computing resources such as servers, storage, and networking infrastructure over the internet. This allows users to rent these resources on a pay-as-you-go basis, rather than purchasing and maintaining physical hardware. Examples of IaaS providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform.

IaaS（基础设施即服务）：IaaS 为用户提供虚拟化计算资源，例如服务器、存储和网络基础设施，可通过互联网租用这些资源，而不是购买和维护物理硬件。 IaaS 提供商包括 Amazon Web Services（AWS）、Microsoft Azure 和 Google Cloud Platform。

- PaaS (Platform as a Service): PaaS provides users with a platform to build, test, and deploy web applications without the need for infrastructure management. This includes operating systems, databases, web servers, and development tools. PaaS providers take care of the infrastructure and allow users to focus on developing and deploying their applications. Examples of PaaS providers include Heroku, Google App Engine, and Microsoft Azure.

  PaaS（平台即服务）：PaaS 为用户提供一个平台来构建、测试和部署 Web 应用程序，无需进行基础设施管理，包括操作系统、数据库、Web 服务器和开发工具。 PaaS 提供商负责基础设施，并允许用户专注于开发和部署应用程序。 PaaS 提供商包括 Heroku、Google App Engine 和 Microsoft Azure。

- SaaS (Software as a Service): SaaS provides users with access to software applications over the internet, typically on a subscription basis. The software is hosted on a remote server and accessed through a web browser or API. Users do not need to install or maintain the software, as it is provided as a service. Examples of SaaS applications include Google Workspace, Salesforce, and Dropbox.

  SaaS（软件即服务）：SaaS 为用户提供通过互联网访问软件应用程序的服务，通常是按订阅方式提供。软件托管在远程服务器上，可通过 Web 浏览器或 API 访问。用户无需安装或维护软件，因为它是作为服务提供的。SaaS 应用程序的例子包括 Google Workspace、Salesforce 和 Dropbox。

下面是一些常见的 IaaS、PaaS 和 SaaS 服务的例子：

IaaS:

- Amazon Web Services (AWS)
- Microsoft Azure
- Google Cloud Platform
- DigitalOcean

PaaS:

- Heroku
- Microsoft Azure App Service
- Google App Engine
- AWS Elastic Beanstalk

SaaS:

- Salesforce
- Dropbox
- Slack
- Google Workspace (G Suite)