前端必会面试题总结

http/https 协议总结

1.0 协议缺陷:

- 无法复用链接，完成即断开，重新慢启动和 `TCP 3` 次握手
- `head of line blocking`：线头阻塞，导致请求之间互相影响

1.1 改进:

- 长连接(默认 `keep-alive`)，复用
- `host` 字段指定对应的虚拟站点
- **新增功能:**
  - 断点续传
  - 身份认证
  - 状态管理
  - `cache` 缓存
    - `Cache-Control`
    - `Expires`
    - `Last-Modified`
    - `Etag`

2.0:

- 多路复用
- 二进制分帧层: 应用层和传输层之间
- 首部压缩
- 服务端推送

https: 较为安全的网络传输协议

- 证书(公钥)
- `SSL` 加密
- 端口 `443`

TCP:

- 三次握手
- 四次挥手
- 滑动窗口: 流量控制
- 拥塞处理
  - 慢开始
  - 拥塞避免
  - 快速重传
  - 快速恢复

缓存策略: 可分为 强缓存 和 协商缓存

- `Cache-Control/Expires`：浏览器判断缓存是否过期，未过期时，直接使用强缓存，`Cache-Control` 的 `max-age` 优先级高于 `Expires`
- 当缓存已经过期时，使用协商缓存
  - 唯一标识方案: `Etag`（`response` 携带) & `If-None-Match`（`request` 携带，上一次返回的 `Etag`）: 服务器判断资源是否被修改
  - 最后一次修改时间: `Last-Modified(response) & If-Modified-Since`（`request`，上一次返回的 `Last-Modified`）
    - 如果一致，则直接返回 304 通知浏览器使用缓存
    - 如不一致，则服务端返回新的资源
- `Last-Modified` 缺点：
  - 周期性修改，但内容未变时，会导致缓存失效
  - 最小粒度只到 `s`，`s` 以内的改动无法检测到
- `Etag` 的优先级高于 `Last-Modified`

常见的HTTP请求头和响应头

HTTP Request Header 常见的请求头：

- Accept:浏览器能够处理的内容类型
- Accept-Charset:浏览器能够显示的字符集
- Accept-Encoding：浏览器能够处理的压缩编码
- Accept-Language：浏览器当前设置的语言
- Connection：浏览器与服务器之间连接的类型
- Cookie：当前页面设置的任何Cookie

- Host：发出请求的页面所在的域
- Referer：发出请求的页面的URL
- User-Agent：浏览器的用户代理字符串

HTTP Responses Header 常见的响应头：

- Date：表示消息发送的时间，时间的描述格式由rfc822定义
- server:服务器名称
- Connection：浏览器与服务器之间连接的类型
- Cache-Control：控制HTTP缓存
- content-type:表示后面的文档属于什么MIME类型

常见的 Content-Type 属性值有以下四种：

（1）application/x-www-form-urlencoded：浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL转码。

（2）multipart/form-data：该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。

（3）application/json：服务器消息主体是序列化后的 JSON 字符串。

（4）text/xml：该种方式主要用来提交 XML 格式的数据。

常见的CSS布局单位

常用的布局单位包括像素（`px`），百分比（`%`），`em`，`rem`，`vw/vh`。

（1）像素（`px`）是页面布局的基础，一个像素表示终端（电脑、手机、平板等）屏幕所能显示的最小的区域，像素分为两种类型：CSS像素和物理像素：

- **CSS像素**：为web开发者提供，在CSS中使用的一个抽象单位；
- **物理像素**：只与设备的硬件密度有关，任何设备的物理像素都是固定的。

（2）百分比（`%`），当浏览器的宽度或者高度发生变化时，通过百分比单位可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果。一般认为子元素的百分比相对于直接父元素。

（3）em和rem相对于px更具灵活性，它们都是相对长度单位，它们之间的区别：em相对于父元素，rem相对于根元素。

- **em：** 文本相对长度单位。相对于当前对象内文本的字体尺寸。如果当前行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸(默认16px)。(相对父元素的字体大小倍数)。
- **rem：** rem是CSS3新增的一个相对单位，相对于根元素（html元素）的font-size的倍数。**作用**：利用rem可以实现简单的响应式布局，可以利用html元素中字体的大小与屏幕间的比值来设置font-size的值，以此实现当屏幕分辨率变化时让元素也随之变化。

（4）vw/vh是与视图窗口有关的单位，vw表示相对于视图窗口的宽度，vh表示相对于视图窗口高度，除了vw和vh外，还有vmin和vmax两个相关的单位。

- vw：相对于视窗的宽度，视窗宽度是100vw；
- vh：相对于视窗的高度，视窗高度是100vh；
- vmin：vw和vh中的较小值；
- vmax：vw和vh中的较大值；

vw/vh 和百分比很类似，两者的区别：

- 百分比（%）：大部分相对于祖先元素，也有相对于自身的情况比如（border-radius、translate等)
- vw/vm：相对于视窗的尺寸

水平垂直居中的实现

- 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过translate来调整元素的中心点到页面的中心。该方法需要**考虑浏览器兼容问题。**

```
.parent {    position: relative;} .child {    position: absolute;    left: 50%;    top: 50%;    trar
```

- 利用绝对定位，设置四个方向的值都为0，并将margin设置为auto，由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。该方法适用于**盒子有宽高**的情况：

```
.parent {
    position: relative;
}
```

```css
.child {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    margin: auto;
}
```

- 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过margin负值来调整元素的中心点到页面的中心。该方法适用于**盒子宽高已知**的情况

css 复制代码

```css
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -50px;     /* 自身 height 的一半 */
    margin-left: -50px;    /* 自身 width 的一半 */
}
```

- 使用flex布局，通过align-items:center和justify-content:center设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。该方法要**考虑兼容的问题**，该方法在移动端用的较多：

css 复制代码

```css
.parent {
    display: flex;
    justify-content:center;
    align-items:center;
}
```

如何根据设计稿进行移动端适配?

移动端适配主要有两个维度：

- **适配不同像素密度，** 针对不同的像素密度，使用 CSS 媒体查询，选择不同精度的图片，以保证图片不会失真；

- **适配不同屏幕大小，** 由于不同的屏幕有着不同的逻辑像素大小，所以如果直接使用 px 作为开发单位，会使得开发的页面在某一款手机上可以准确显示，但是在另一款手机上就会失真。为了适配不同屏幕的大小，应按照比例来还原设计稿的内容。

为了能让页面的尺寸自适应，可以使用 rem，em，vw，vh 等相对单位。

对 CSSSprites 的理解

CSSSprites（精灵图），将一个页面涉及到的所有图片都包含到一张大图中去，然后利用CSS的 background-image，background-repeat，background-position属性的组合进行背景定位。

优点：

- 利用 `CSS Sprites` 能很好地减少网页的http请求，从而大大提高了页面的性能，这是 `CSS Sprites` 最大的优点；
- `CSS Sprites` 能减少图片的字节，把3张图片合并成1张图片的字节总是小于这3张图片的字节总和。

缺点：

- 在图片合并时，要把多张图片有序的、合理的合并成一张图片，还要留好足够的空间，防止板块内出现不必要的背景。在宽屏及高分辨率下的自适应页面，如果背景不够宽，很容易出现背景断裂；
- `CSSSprites` 在开发的时候相对来说有点麻烦，需要借助 `photoshop` 或其他工具来对每个背景单元测量其准确的位置。
- 维护方面： `CSS Sprites` 在维护的时候比较麻烦，页面背景有少许改动时，就要改这张合并的图片，无需改的地方尽量不要动，这样避免改动更多的 `CSS` ，如果在原来的地方放不下，又只能（最好）往下加图片，这样图片的字节就增加了，还要改动 `CSS` 。

参考 前端进阶面试题详细解答

CSS3中有哪些新特性

- 新增各种CSS选择器 （: not(.input)：所有 class 不是“input”的节点）
- 圆角 （border-radius:8px）
- 多列布局 （multi-column layout）
- 阴影和反射 （Shadoweflect）

- 文字特效 （text-shadow）
- 文字渲染 （Text-decoration）
- 线性渐变 （gradient）
- 旋转 （transform）
- 增加了旋转,缩放,定位,倾斜,动画,多背景

对 CSS 工程化的理解

CSS 工程化是为了解决以下问题：

1. **宏观设计**：CSS 代码如何组织、如何拆分、模块结构怎样设计？
2. **编码优化**：怎样写出更好的 CSS？
3. **构建**：如何处理我的 CSS，才能让它的打包结果最优？
4. **可维护性**：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

以下三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践：

- 预处理器：Less、 Sass 等；
- 重要的工程化插件： PostCss；
- Webpack loader 等 。

基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

（1）预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码： 那为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这就和本来用 JS 也可以实现所有功能，但最后却写 React 的 jsx 或者 Vue 的模板语法一样——为了爽！要想知道有了预处理器有多爽，首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高，前端工程中对 CSS 提出了以下的诉求：

1. 宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；
2. 编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；
3. 可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器普遍会具备这样的特性：

- 嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系；
- 支持定义 css 变量；
- 提供计算函数；
- 允许对代码片段进行 extend 和 mixin；
- 支持循环语句的使用；
- 支持将 CSS 文件模块化，实现复用。

（2）PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss?

它和预处理器的不同就在于，预处理器处理的是 类CSS，而 PostCss 处理的就是 CSS 本身。Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多：

- 提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；
- 当我们的 CSS 代码需要适配低版本浏览器时，PostCss 的 Autoprefixer 插件可以帮助我们自动增加浏览器前缀；
- 允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

（3）Webpack 能处理 CSS 吗？如何实现？ Webpack 能处理 CSS 吗：

- **Webpack 在裸奔的状态下，是不能处理 CSS 的**，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；
- Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理：

- Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader
- 注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：
  - css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；
  - style-loader：创建style标签，把 CSS 内容写入标签。

在实际使用中，**css-loader 的执行顺序一定要安排在 style-loader 的前面。**因为只有完成了编译过程，才可以对 css 代码进行插入；若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的，它会无情报错。

对对象与数组的解构的理解

解构是 ES6 提供的一种新的提取数据的模式，这种模式能够从对象或数组里有针对性地拿到想要的数值。 **1）数组的解构** 在解构数组时，以元素的位置为匹配条件来提取想要的数据的：

javascript 复制代码

```
const [a, b, c] = [1, 2, 3]
```

最终，a、b、c分别被赋予了数组第0、1、2个索引位的值：

数组里的0、1、2索引位的元素值，精准地被映射到了左侧的第0、1、2个变量里去，这就是数组解构的工作模式。还可以通过给左侧变量数组设置空占位的方式，实现对数组中某几个元素的精准提取：

javascript 复制代码

```
const [a,,c] = [1,2,3]
```

通过把中间位留空，可以顺利地把数组第一位和最后一位的值赋给 a、c 两个变量：

2）对象的解构 对象解构比数组结构稍微复杂一些，也更显强大。在解构对象时，是以属性的名称为匹配条件，来提取想要的数据的。现在定义一个对象：

javascript 复制代码

```
const stu = {
  name: 'Bob',
  age: 24
}
```

假如想要解构它的两个自有属性，可以这样：

javascript 复制代码

```
const { name, age } = stu
```

这样就得到了 name 和 age 两个和 stu 平级的变量：

注意，对象解构严格以属性名作为定位依据，所以就算调换了 name 和 age 的位置，结果也是一样的：

```javascript 复制代码
const { age, name } = stu
```

其他值到布尔类型的值的转换规则?

以下这些是假值： • undefined • null • false • +0、-0 和 NaN • ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

null和undefined区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是**未定义**，null 代表的含义是**空对象**。一般变量声明了但还没有定义的时候会返回 undefined，null主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回 “object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

await 到底在等啥?

await 在等待什么呢? 一般来说，都认为 await 是在等待一个 async 函数完成。不过按语法说明，await 等待的是一个表达式，这个表达式的计算结果是 Promise 对象或者其它值（换句话说，就是没有特殊限定）。

因为 async 函数返回一个 Promise 对象，所以 await 可以用于等待一个 async 函数的返回值——这也可以说是 await 在等 async 函数，但要清楚，它等的实际是一个返回值。注意到

await 不仅仅用于等 Promise 对象，它可以等任意表达式的结果，所以，await 后面实际是可以接普通函数调用或者直接量的。所以下面这个示例完全可以正确运行：

javascript 复制代码

```javascript
function getSomething() {
    return "something";
}
async function testAsync() {
    return Promise.resolve("hello async");
}
async function test() {
    const v1 = await getSomething();
    const v2 = await testAsync();
    console.log(v1, v2);
}
test();
```

await 表达式的运算结果取决于它等的是什么。

- 如果它等到的不是一个 Promise 对象，那 await 表达式的运算结果就是它等到的东西。
- 如果它等到的是一个 Promise 对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

来看一个例子：

javascript 复制代码

```javascript
function testAsy(x){
   return new Promise(resolve=>{setTimeout(() => {
       resolve(x);
     }, 3000)
    }
   )
}
async function testAwt(){
  let result =  await testAsy('hello world');
  console.log(result);    // 3秒钟之后出现hello world
  console.log('cuger')   // 3秒钟之后出现cug
}
testAwt();
console.log('cug')  //立即输出cug
```

这就是 await 必须用在 async 函数中的原因。async 函数调用不会造成阻塞，它内部所有的阻塞都被封装在一个 Promise 对象中异步执行。await暂停当前async的执行，所以'cug''最先输

出，hello world'和'cuger'是3秒钟后同时出现的。

JavaScript脚本延迟加载的方式有哪些?

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- **defer 属性：** 给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。
- **async 属性：** 给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。
- **动态创建 DOM 方式：** 动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。
- **使用 setTimeout 延迟方法：** 设置一个定时器来延迟加载js脚本文件
- **让 JS 最后加载：** 将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

Unicode、UTF-8、UTF-16、UTF-32的区别?

（1）Unicode

在说 `Unicode` 之前需要先了解一下 `ASCII` 码：ASCII 码（`American Standard Code for Information Interchange`）称为美国标准信息交换码。

- 它是基于拉丁字母的一套电脑编码系统。
- 它定义了一个用于代表常见字符的字典。
- 它包含了"A-Z"(包含大小写)，数据"0-9" 以及一些常见的符号。
- 它是专门为英语而设计的，有128个编码，对其他语言无能为力

`ASCII` 码可以表示的编码有限，要想表示其他语言的编码，还是要使用 `Unicode` 来表示，可以说 `Unicode` 是 `ASCII` 的超集。

Unicode 全称 Unicode Translation Format，又叫做统一码、万国码、单一码。 Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

Unicode 的实现方式（也就是编码方式）有很多种，常见的是**UTF-8**、**UTF-16**、**UTF-32**和**USC-2**。

(2) UTF-8

UTF-8 是使用最广泛的 Unicode 编码方式，它是一种可变长的编码方式，可以是1—4个字节不等，它可以完全兼容 ASCII 码的128个字符。

注意： UTF-8 是一种编码方式， Unicode 是一个字符集合。

UTF-8 的编码规则：

- 对于**单字节**的符号，字节的第一位为0，后面的7位为这个字符的 Unicode 编码，因此对于英文字母，它的 Unicode 编码和 ACSII 编码一样。
- 对于**n字节**的符号，第一个字节的前n位都是1，第n+1位设为0，后面字节的前两位一律设为10，剩下的没有提及的二进制位，全部为这个符号的 Unicode 码 。

来看一下具体的 Unicode 编号范围与对应的 UTF-8 二进制格式：

编码范围（编号对应的十进制数）	二进制格式
0x00—0x7F （0-127）	0xxxxxxx
0x80—0x7FF （128-2047）	110xxxxx 10xxxxxx
0x800—0xFFFF （2048-65535）	1110xxxx 10xxxxxx 10xxxxxx
0x10000—0x10FFFF （65536以上）	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

那该如何通过具体的 Unicode 编码，进行具体的 UTF-8 编码呢？**步骤如下：**

- 找到该 Unicode 编码的所在的编号范围，进而找到与之对应的二进制格式
- 将 Unicode 编码转换为二进制数（去掉最高位的0）
- 将二进制数从右往左一次填入二进制格式的 X 中，如果有 X 未填，就设为0

来看一个实际的例子：“**马**”字的 Unicode 编码是： 0x9A6C ，整数编号是 39532 （1）首选确定了该字符在第三个范围内，它的格式是 1110xxxx 10xxxxxx 10xxxxxx （2）39532对应的二进制数为 1001 1010 0110 1100 （3）将二进制数填入X中，结果是： 11101001 10101001 10101100

（3）UTF-16

1. 平面的概念

在了解 UTF-16 之前，先看一下**平面**的概念： Unicode 编码中有很多很多的字符，它并不是一次性定义的，而是分区进行定义的，每个区存放**65536**（216）个字符，这称为一个**平面**，目前总共有17 个平面。

最前面的一个平面称为**基本平面**，它的码点从**0 — 216-1**，写成16进制就是 U+0000 — U+FFFF ，那剩下的16个平面就是**辅助平面**，码点范围是 U+10000–U+10FFFF 。

2. UTF-16 概念：

UTF-16 也是 Unicode 编码集的一种编码形式，把 Unicode 字符集的抽象码位映射为16位长的整数（即码元）的序列，用于数据存储或传递。 Unicode 字符的码位需要1个或者2个16位长的码元来表示，因此 UTF-16 也是用变长字节表示的。

3. UTF-16 编码规则：

- 编号在 U+0000–U+FFFF 的字符（常用字符集），直接用两个字节表示。
- 编号在 U+10000–U+10FFFF 之间的字符，需要用四个字节表示。

4. 编码识别

那么问题来了，当遇到两个字节时，怎么知道是把它当做一个字符还是和后面的两个字节一起当做一个字符呢?

UTF-16 编码肯定也考虑到了这个问题，在基本平面内，从 U+D800 — U+DFFF 是一个空段，也就是说这个区间的码点不对应任何的字符，因此这些空段就可以用来映射辅助平面的字符。

辅助平面共有 **220** 个字符位，因此表示这些字符至少需要 20 个二进制位。 UTF-16 将这 20 个二进制位分成两半，前 10 位映射在 U+D800 — U+DBFF ，称为**高位**（H），后 10 位映射在 U+DC00 — U+DFFF ，称为**低位**（L）。这就相当于，将一个辅助平面的字符拆成了两个基本平面的字符来表示。

因此，当遇到两个字节时，发现它的码点在 U+D800 —U+DBFF 之间，就可以知道，它后面的两个字节的码点应该在 U+DC00 — U+DFFF 之间，这四个字节必须放在一起进行解读。

5. 举例说明

以 "嫡" 字为例，它的 Unicode 码点为 0x21800 ，该码点超出了基本平面的范围，因此需要用四个字节来表示，步骤如下：

- 首先计算超出部分的结果： 0x21800 - 0x10000
- 将上面的计算结果转为20位的二进制数，不足20位就在前面补0，结果为： 0001000110 0000000000
- 将得到的两个10位二进制数分别对应到两个区间中
- U+D800 对应的二进制数为 1101100000000000 ， 将 0001000110 填充在它的后10 个二进制位，得到 1101100001000110 ，转成 16 进制数为 0xD846 。同理，低位为 0xDC00 ，所以这个字的 UTF-16 编码为 0xD846 0xDC00

(4) UTF-32

UTF-32 就是字符所对应编号的整数二进制形式，每个字符占四个字节，这个是直接进行转换的。该编码方式占用的储存空间较多，所以使用较少。

比如“**马**”字的Unicode编号是： U+9A6C ，整数编号是 39532 ，直接转化为二进制： 1001 1010 0110 1100 ，这就是它的UTF-32编码。

(5) 总结

Unicode、UTF-8、UTF-16、UTF-32有什么区别?

- Unicode 是编码字符集（字符集），而 UTF-8 、 UTF-16 、 UTF-32 是字符集编码（编码规则）；
- UTF-16 使用变长码元序列的编码方式，相较于定长码元序列的 UTF-32 算法更复杂，甚至比同样是变长码元序列的 UTF-8 也更为复杂，因为其引入了独特的**代理对**这样的代理机制；
- UTF-8 需要判断每个字节中的开头标志信息，所以如果某个字节在传送过程中出错了，就会导致后面的字节也会解析出错；而 UTF-16 不会判断开头标志，即使错也只会错一个字符，所以容错能力教强；
- 如果字符内容全部英文或英文与其他文字混合，但英文占绝大部分，那么用 UTF-8 就比 UTF-16 节省了很多空间；而如果字符内容全部是中文这样类似的字符或者混合字符中中文

占绝大多数，那么 UTF-16 就占优势了，可以节省很多空间；

Canvas和SVG的区别

（1）SVG： SVG可缩放矢量图形（Scalable Vector Graphics）是基于可扩展标记语言XML描述的2D图形的语言，SVG基于XML就意味着SVG DOM中的每个元素都是可用的，可以为某个元素附加Javascript事件处理器。在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变化，那么浏览器能够自动重现图形。

其特点如下：

- 不依赖分辨率
- 支持事件处理器
- 最适合带有大型渲染区域的应用程序（比如谷歌地图）
- 复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）
- 不适合游戏应用

（2）Canvas： Canvas是画布，通过Javascript来绘制2D图形，是逐像素进行渲染的。其位置发生改变，就会重新进行绘制。

其特点如下：

- 依赖分辨率
- 不支持事件处理器
- 弱的文本渲染能力
- 能够以 .png 或 .jpg 格式保存结果图像
- 最适合图像密集型的游戏，其中的许多对象会被频繁重绘

注：矢量图，也称为面向对象的图像或绘图图像，在数学上定义为一系列由线连接的点。矢量文件中的图形元素称为对象。每个对象都是一个自成一体的实体，它具有颜色、形状、轮廓、大小和屏幕位置等属性。

HTTP协议的性能怎么样

HTTP 协议是基于 TCP/IP，并且使用了**请求-应答**的通信模式，所以性能的关键就在这两点里。

- **长连接**

HTTP协议有两种连接模式，一种是持续连接，一种非持续连接。 （1）非持续连接指的是服务器必须为每一个请求的对象建立和维护一个全新的连接。 （2）持续连接下，TCP 连接默认不关闭，可以被多个请求复用。采用持续连接的好处是可以避免每次建立 TCP 连接三次握手时所花费的时间。

对于不同版本的采用不同的连接方式：

- 在HTTP/1.0 每发起一个请求，都要新建一次 TCP 连接（三次握手），而且是串行请求，做了无畏的 TCP 连接建立和断开，增加了通信开销。该版本使用的非持续的连接，但是可以在请求时，加上 Connection: keep-a live 来要求服务器不要关闭 TCP 连接。

- 在HTTP/1.1 提出了**长连接**的通信方式，也叫持久连接。这种方式的好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载。该版本及以后版本默认采用的是持续的连接。目前对于同一个域，大多数浏览器支持同时建立 6 个持久连接。

- **管道网络传输**

HTTP/1.1 采用了长连接的方式，这使得管道（pipeline）网络传输成为了可能。

管道（pipeline）网络传输是指：可以在同一个 TCP 连接里面，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。但是服务器还是按照顺序回应请求。如果前面的回应特别慢，后面就会有许多请求排队等着。这称为队头堵塞。

- **队头堵塞**

HTTP 传输的报文必须是一发一收，但是，里面的任务被放在一个任务队列中串行执行，一旦队首的请求处理太慢，就会阻塞后面请求的处理。这就是HTTP队头阻塞问题。

队头阻塞的解决方案： （1）并发连接：对于一个域名允许分配多个长连接，那么相当于增加了任务队列，不至于一个队伍的任务阻塞其它所有任务。 （2）域名分片：将域名分出很多二级域名，它们都指向同样的一台服务器，能够并发的长连接数变多，解决了队头阻塞的问题。

use strict是什么意思？使用它区别是什么?

use strict 是一种 ECMAscript5 添加的（严格模式）运行模式，这种模式使得 Javascript 在更严格的条件下运行。设立严格模式的目的如下：

- 消除 Javascript 语法的不合理、不严谨之处，减少怪异行为;

- 消除代码运行的不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 Javascript 做好铺垫。

区别：

- 禁止使用 with 语句。
- 禁止 this 关键字指向全局对象。
- 对象不能有重名的属性。

异步编程的实现方式?

JavaScript中的异步机制可以分为以下几种：

- **回调函数** 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。
- **Promise** 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。
- **generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。
- **async 函数** 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

浏览器乱码的原因是什么？如何解决?

产生乱码的原因：

- 网页源代码是 `gbk` 的编码，而内容中的中文字是 `utf-8` 编码的，这样浏览器打开即会出现 `html` 乱码，反之也会出现乱码；
- `html` 网页编码是 `gbk` ，而程序从数据库中调出呈现是 `utf-8` 编码的内容也会造成编码乱码；

- 浏览器不能自动检测网页编码，造成网页乱码。

解决办法：

- 使用软件编辑HTML网页内容；
- 如果网页设置编码是 `gbk` ，而数据库储存数据编码格式是 `UTF-8` ，此时需要程序查询数据库数据显示数据前进程序转码；
- 如果浏览器浏览时候出现网页乱码，在浏览器中找到转换编码的菜单进行转换。

谈谈你对React的理解

> React 是一个网页 UI 框架，通过组件化的方式解决视图层开发复用的问题，本质是一个组件化框架。

- 它的核心设计思路有三点，分别是 `声明式、组件化与 通用性` 。
- 声明式的优势在于直观与组合。
- 组件化的优势在于视图的拆分与模块复用，可以更容易做到高内聚低耦合。
- 通用性在于一次学习，随处编写。比如 React Native，React 360 等， 这里主要靠虚拟 DOM 来保证实现。
- 这使得 React 的适用范围变得足够广，无论是 Web、Native、VR，甚至 Shell 应用都可以进行开发。这也是 React 的优势。
- 但作为一个视图层的框架，React 的劣势也十分明显。它并没有提供完整的一揽子解决方案，在开发大型前端应用时，需要向社区寻找并整合解决方案。虽然一定程度上促进了社区的繁荣，但也为开发者在技术选型和学习适用上造成了一定的成本。
- 承接在优势后，可以再谈一下自己对于 React 优化的看法、对虚拟 DOM 的看法