2022年我的前端面试准备

面试题

1. 圣杯和双飞翼实现方式的区别?

答：共同点：

1. 都使用了 `float`
2. 中间部分在文档前面，为了优先加载

区别：

- 圣杯： 左、中、右三个盒子在一个同一个盒子中，设置外侧盒子的 `padding` ，从而留出两侧盒子位置
- 双飞翼： 左、中、右三个盒子同级，在中间盒子里放一个小盒子，设置小盒子的 `margin` ，从而留出两侧盒子位置

元素居中的方式

答：

1. 使用 `margin` 进行固定长度的偏移

```css
/*关键样式代码*/
#father{
          overflow: hidden;
       }
   #son{
          margin:0 auto;/*水平居中*/
          margin-top: 50px;
        }
```

2. 使用绝对定位并进行偏移

css 复制代码

```css
/*关键样式代码*/
#father{
         position:relative;
       }
   #son{
         position: absolute;
         left:50%;
         margin-left: -50px;
         top:50%;
         margin-top: -50px;
        }

 /*优化代码（使用css样式中的计算公式）*/
#son{
        position: absolute;
        left:calc(50% - 50px);
        top:calc(50% - 50px);
        }
```

3. 使用绝对定位并 margin 自适应进行居中

css 复制代码

```css
/*关键样式代码*/
#father{
         position:relative;
       }
   #son{
         position: absolute;
         left: 0;
         top: 0;
         right: 0;
         bottom: 0;
         margin:auto;
        }
```

4. 使用 table-cell 进行居中显示

css 复制代码

```css
/*关键样式代码*/
 #father{
            display: table-cell;
            vertical-align: middle;
        }
    #son{
            margin: 0 auto;
        }
```

5. 使用弹性盒子来实现居中

```css
#father{ display: flex; justify-content: center; align-items: center; }
```

3.讲讲闭包以及在项目中的使用场景

闭包可以理解为定义在一个函数内部的函数

它可以实现变量私有化但同时容易造成内存泄漏

使用场景主要有返回值、函数赋值、自执行函数、迭代器等等

4.讲讲promise以及在项目中的使用场景

`ECMAscript 6` 原生提供了 `Promise` 对象。

`Promise` 对象代表了未来将要发生的事件，用来传递异步操作的消息。

Promise 对象有以下两个特点:

1、对象的状态不受外界影响。 `Promise` 对象代表一个异步操作，有三种状态：

- `pending`：初始状态，不是成功或失败状态。
- `fulfilled`：意味着操作成功完成。
- `rejected`：意味着操作失败。

只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 Promise 这个名字的由来，它的英语意思就是「承诺」，表示其他手段无法改变。

2、一旦状态改变，就不会再变，任何时候都可以得到这个结果。 `Promise` 对象的状态改变，只有两种可能：从 `Pending` 变为 `Resolved` 和从 `Pending` 变为 `Rejected` 。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（ `Event` ）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了 `Promise` 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外， `Promise` 对象提供统一的接口，使得控制异步操作更加容易。

`Promise` 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数， `Promise` 内部抛出的错误，不会反应到外部。第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

5.谈谈Promise.all方法，Promise.race方法以及使用

- **Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。（Promise.all 方法的参数不一定是数组，但是必须具有 iterator 接口，且返回的每个成员都是 Promise 实例。）**

- **顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。**

`Promise.all` 可以比作接力跑，必须都成功才能胜利

`Promise.race` 可以比作短跑，谁跑的快睡就胜利

使用场景

在前端开发请求数据的过程中，偶尔会遇到发送多个请求并根据请求顺序获取和使用数据的场景，使用Promise.all毫无疑问可以解决这个问题。

Promise.race则是在拼手速抢东西就可用到

6.谈谈你理解的vue(这个问题很广可以用react、Angular进行比较说明)

`Vue` 是个渐进式的轻量框架渐进式代表的含义是：主张最少。

每个框架都不可避免会有自己的一些特点，从而会对使用者有一定的要求，这些要求就是主张，主张有强有弱，它的强势程度会影响在业务开发中的使用方式。

比如说， `Angular` ，它两个版本都是强主张的，如果你用它，必须接受以下东西：

- 必须使用它的模块机制- 必须使用它的依赖注入
- 必须使用它的特殊形式定义组件（这一点每个视图框架都有，难以避免）

所以Angular是带有比较强的排它性的，如果你的应用不是从头开始，而是要不断考虑是否跟其他东西集成，这些主张会带来一些困扰。

比如React，它也有一定程度的主张，它的主张主要是函数式编程的理念，比如说，你需要知道什么是副作用，什么是纯函数，如何隔离副作用。它的侵入性看似没有Angular那么强，主要因为它是软性侵入。

Vue可能有些方面是不如 `React` ，不如 `Angular` ，但它是渐进的，没有强主张，你可以在原有大系统的上面，把一两个组件改用它实现，当 `jQuery` 用；也可以整个用它全家桶开发，当 `Angular` 用；还可以用它的视图，搭配你自己设计的整个下层用。你可以在底层数据逻辑的地方用OO和设计模式的那套理念，也可以函数式，都可以，它只是个轻量视图而已，只做了自己该做的事，没有做不该做的事，仅此而已。
渐进式的含义，我的理解是：没有多做职责之外的事。

7.关于Vue的组件传值有哪些?

1. 子组件 `props` 接受父组件传值，向父组件传值时需要使用 `vue` 中的 `$on` 和 `$emit`

2. `eventBus` 创建一个Vue的实例，让各个组件共用同一个事件机制。 传递数据方，通过一个事件触发 `eventBus.emit` （方法名， 传递的数据）。 接收数据方， 通过 `mounted( )` 触发 `eventBus.emit` (方法名，传递的数据)。 接收数据方，通过mounted() {}触发eventBus.emit(方法名，传递的数据)。接收数据方，通过 `mounted()` 触发 `eventBus.on(方法名，function(接收数据的参数){用该组件的数据接收传递过来的数据})`

3. `provide和inject`

8.谈谈provide和inject

成对出现： `provide和inject` 是成对出现的

作用：用于父组件向子孙组件传递数据

使用方法： `provide` 在父组件中返回要传给下级的数据， `inject` 在需要使用这个数据的子辈组

使用场景：由于 `vue` 有 `$parent` 属性可以让子组件访问父组件。但孙组件想要访问祖先组件就比较困难。通过 `provide/inject` 可以轻松实现跨级访问父组件的数据

`provider/inject` ：简单的来说就是在父组件中通过 `provider` 来提供变量，然后在子组件中通过 `inject` 来注入变量

需要注意的是这里不论子组件有多深，只要调用了inject那么就可以注入provider中的数据。而不是局限于只能从当前父组件的prop属性来获取数据。

```
// 孙组件
const SunChildComponent = {
  template: '<div>child component</div>',
  //跨级使用了父组件的数据
  inject: ['yeye'],
  mounted () {
    console.log(this.yeye)
  }
}
```

```
// 子组件
const ChildComponent = {
  name: 'comp',
  components: {
    SunChildComponent
  },
  template: `
    <div :style = "style">
      <slot :value = "value" :aaa = "aaa"></slot>
      <sun-child-component></sun-child-component>
    </div>
  `,
  data () {
    return {
      value: 'component val',
      aaa: 'component aaa'
    }
  }
}
```

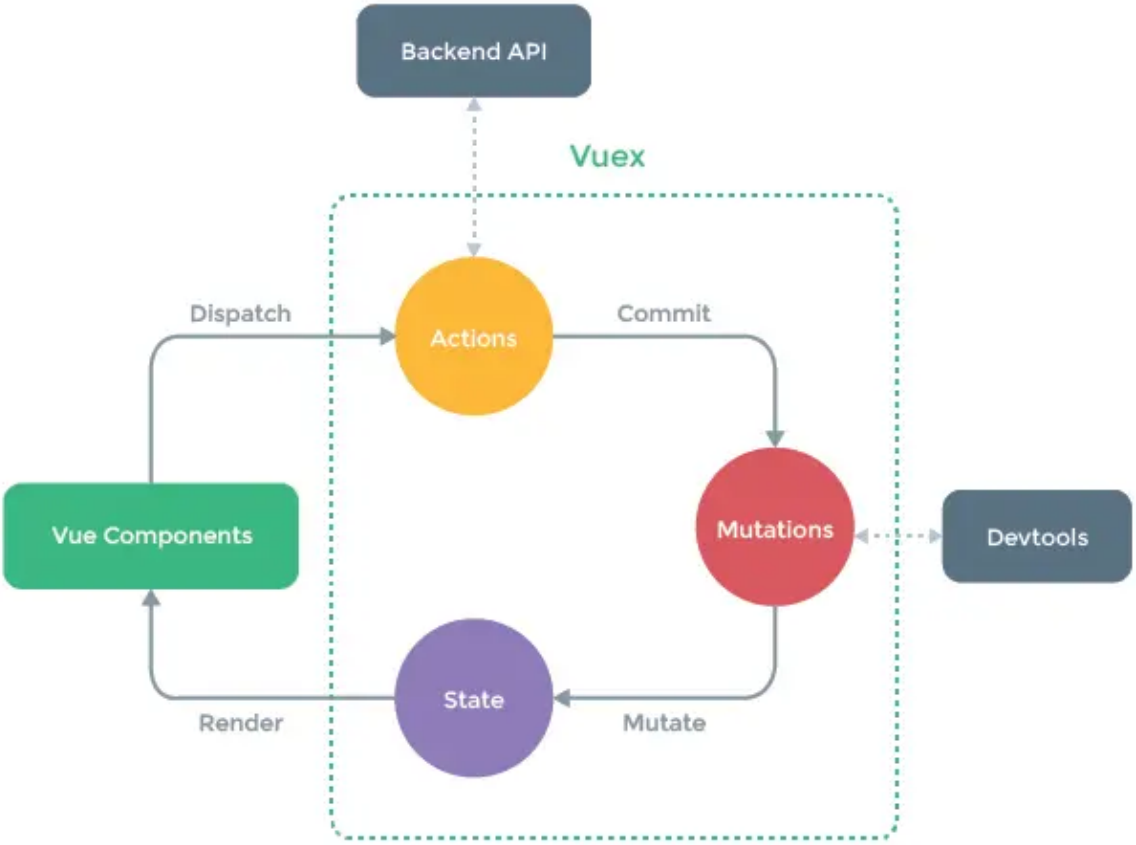
```
// 父组件
new Vue({
  components: {
    CompOne: ChildComponent
  },
  //父组件通过provide将自己的数据以对象形式传出去
  provide () {
    return {
      yeye: this
      value: this.value
    }
  },
  el: '#root',
  data () {
    return {
      value: '本组件的123'
    }
  },
  mounted () {
    console.log(this.$refs.comp.value)
    console.log(this.$refs.span)
  },
  template: `
    <div>
      <comp-one ref = "comp">
        <span slot-scope = "props" ref="span">{{props.value}} {{props.aaa}} {{value}
      </comp-one>
    </div>
  `
})
```


9.说说Vuex

`Vuex` 是一个专为 `Vue.js` 应用程序开发的**状态管理模式 + 库**。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

`Vuex` 可以帮助我们管理共享状态，并附带了更多的概念和框架。这需要对短期和长期效益进行权衡。

如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的。确实是如此——如果您的应用够简单，您最好不要使用 Vuex。一个简单的 store 模式就足够您所需了。但是，如果您需要构建一个中大型单页应用，您很可能会考虑如何更好地在组件外部管理状态，Vuex 将会成为自然而然的选择。



`store` 注入 vue的实例组件的方式，是通过 `vue` 的 `mixin` 机制，借助 `vue` 组件的生命周期 钩子 beforeCreate 完成的。即 每个 `vue` 组件实例化过程中，会在 `beforeCreate` 钩子前调用 `vuexInit` 方法。

10.路由守卫有哪些?

路由守卫又称导航守卫，指是路由跳转前、中、后过程中的一些钩子函数。官方解释是 `vue-router` 提供的导航守卫，要通过跳转或取消的方式来守卫导航。路由守卫分为三种，全局路由、组件内路由，路由独享。

全局路由钩子函数有：beforeEach、beforeResolve、afterEach（参数中没有next）

组件内路由的钩子函数有：beforeRouterEnter、beforeRouteUpdate、beforeRouteLeave

路由独享的钩子函数有：beforeEnter

11.箭头函数与普通函数的区别

- 箭头函数是匿名函数，不能作为构造函数，不能使用new
- 箭头函数不绑定 `arguments` ，取而代之用 `rest` 参数...解决
- 箭头函数不绑定 `this` ，会捕获其所在的上下文的this值，作为自己的this值
- 箭头函数通过 `call()` 或 `apply()` 方法调用一个函数时，只传入了一个参数，对 this 并没有影响。
- 箭头函数没有原型属性
- 箭头函数不能当做 `Generator` 函数,不能使用 `yield` 关键字

12.Git常用指令

- 检出仓库：$ git clone
- 查看远程仓库：$ git remote -v
- 添加远程仓库：$ git remote add [name] [url]
- 删除远程仓库：$ git remote rm [name]
- 修改远程仓库：$ git remote set-url --push [name] [newUrl]
- 拉取远程仓库：$ git pull [remoteName] [localBranchName]
- 推送远程仓库：$ git push [remoteName] [localBranchName]

- *如果想把本地的某个分支test提交到远程仓库，并作为远程仓库的master分支，或者作为另外一个名叫test的分支，如下：

$git push origin test:master　　// 提交本地test分支作为远程的master分支

- 查看本地分支：$ git branch
- 查看远程分支：$ git branch -r
- 创建本地分支：$ git branch [name] ----注意新分支创建后不会自动切换为当前分支
- 切换分支：$ git checkout [name]
- 创建新分支并立即切换到新分支：$ git checkout -b [name]
- 删除分支：$ git branch -d [name] ---- -d选项只能删除已经参与了合并的分支，对于未有合并的分支是无法删除的。如果想强制删除一个分支，可以使用-D选项
- 合并分支：$ git merge [name] ----将名称为[name]的分支与当前分支合并
- 创建远程分支(本地分支push到远程)：$ git push origin [name]
- 删除远程分支：*gitpushorigin* : *heads*/[*name*] 或 gitpush origin :[name]

13.Git代码回滚

- git reset --hard HEAD^ 回退到上个版本
- git reset --hard HEAD~3 回退到前3次提交之前，以此类推，回退到n次提交之前
- git reset --hard commit_id 退到/进到，指定commit的哈希码（这次提交之前或之后的提交都会回滚）

14.axios怎么去做请求拦截

```
// 请求拦截器
instance.interceptors.request.use(req=>{}, err=>{});
// 响应拦截器
instance.interceptors.reponse.use(req=>{}, err=>{});
```

1. 请求拦截器

```
    // 在发送请求前要做的事儿
    ...
    return req
}, err => {
    // 在请求错误时要做的事儿
    ...
    // 该返回的数据则是axios.catch(err)中接收的数据
    return Promise.reject(err)
})
```

2. 响应拦截器

```
// use(两个参数)
axios.interceptors.reponse.use(res => {
    // 请求成功对响应数据做处理
    ...
    // 该返回的数据则是axios.then(res)中接收的数据
    return res
}, err => {
    // 在请求错误时要做的事儿
    ...
    // 该返回的数据则是axios.catch(err)中接收的数据
    return Promise.reject(err)
})
```

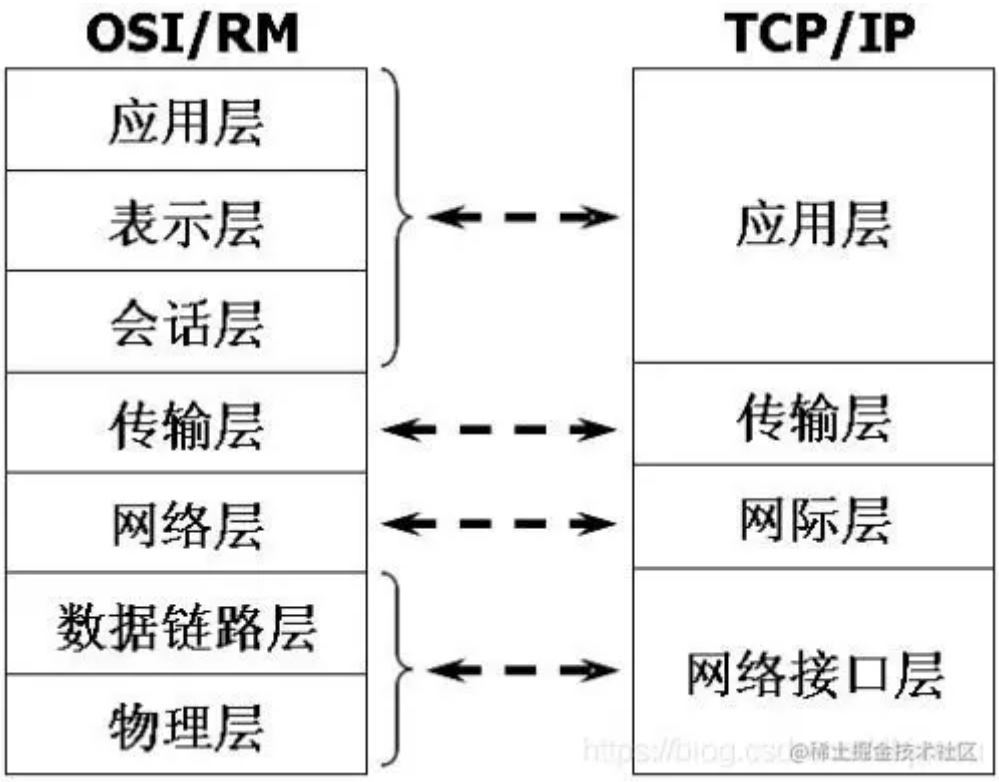
15.关于OSI七层模型和TCp四层模型

OSI 分层：应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

TCP/IP 模型：应用层、传输层、网络层、网络接口层

应用层协议(常用)： HTTP、RTSP、FTP

传输层协议： TCP、UDP



1、OSI的七层模型是什么?

ISO于1978年开发的一套标准架构ISO模型，被引用来说明数据通信协议的结构和功能。

OSI在功能上可以划分为两组：

网络群组：物理层、数据链路层、网络层

使用者群组：传输层、会话层、表示层、应用层

OSI 七层网络模型	TCP/IP 四层概念模型	对应网络协议
7：应用层	应用层	HTTP 、 RTSP TFTP（简单文本传输协议）、 FTP 、 NFS （数域筛法，数据加密）、 WAIS`（广域信息查询系统）
6：表示层	应用层	Telnet （internet远程登陆服务的标准协议）、 Rlogin 、 SNMP （网络管理协议）、Gopher
5：会话层	应用层	SMTP （简单邮件传输协议）、 DNS （域名系统）
4：传输层	传输层	TCP （传输控制协议）、 UDP （用户数据报协议））
3：网络层	网际层	ARP （地域解析协议）、 RARP 、 AKP 、 UUCP （Unix to Unix copy）
2：数据链路层	数据链路层	FDDI （光纤分布式数据接口）、 Ethernet、Arpanet、PDN （公用数据网）、 SLIP （串行线路网际协议） PPP （点对点协议，通过拨号或专线方建立点对点连接发送数据）
1：物理层	物理层	SMTP （简单邮件传输协议）、 DNS （域名系统）

其中高层（7、6、5、4层）定义了应用程序的功能，下面三层（3、2、1层）主要面向通过网络的端到端的数据流

2、 tcp/udp 属于哪一层?

传输层

3、 tcp/udp 有哪些优缺点?

(1) tcp 是面向连接的， udp 是面向无连接的

tcp 在通信之前必须通过三次握手机制与对方建立连接，而udp通信不必与对方建立连接，不管对方的状态就直接把数据发送给对方

(2) tcp 连接过程耗时， udp 不耗时

(3) tcp 连接过程中出现的延时增加了被攻击的可能，安全性不高，而 udp 不需要连接，安全性

(4) tcp 是可靠的，保证数据传输的正确性，不易丢包， udp 是不可靠的，易丢包

tcp 可靠的四大手段：

顺序编号： tcp 在传输文件的时候，会将文件拆分为多个tcp数据包，每个装满的数据包大小大约在1k左右， tcp 协议为保证可靠传输，会将这些数据包顺序编号

确认机制：当数据包成功的被发送方发送给接收方，接收方会根据 tcp 协议反馈给发送方一个成功接收的 ACK 信号，信号中包含了当前包的序号

超时重传：当发送方发送数据包给接收方时，会为每一个数据包设置一个定时器，当在设定的时间内，发送方仍没有收到接收方的 ACK 信号，会再次发送该数据包，直到收到接收方的ACK信号或者连接已断开

校验信息： tcp 首部校验信息较多， udp 首部校验信息较少

(5) tcp 传输速率较慢，实时性差， udp 传输速率较快

tcp 建立连接需要耗时，并且 tcp 首部信息太多，每次传输的有用信息较少，实时性差

(6) tcp 是流模式， udp 是数据包模式

tcp 只要不超过缓冲区的大小就可以连续发送数据到缓冲区上，接收端只要缓冲区上有数据就可以读取，可以一次读取多个数据包，而 udp 一次只能读取一个数据包，数据包之间独立

4、 tcp/udp 的使用场合?

(1)对数据可靠性的要求。 tcp 适用于可靠性高的场合，udp适用于可靠性低的场合

(2)应用的实时性。 tcp 有延时较大， udp 延时较小

(3)网络的可靠性。网络不好的情况下使用 tcp ，网络条件好的情况下，使用 udp

5、 PPP 协议属于哪一层协议?

数据链路层

16.从浏览器地址栏输入url到显示页面的步骤

详细版本

1. 在浏览器地址栏输入URL

2. 浏览器查看缓存，如果请求资源在缓存中并且新鲜，跳转到转码步骤

    1. 如果资源未缓存，发起新请求

    2. 如果已缓存，检验是否足够新鲜，足够新鲜直接提供给客户端，否则与服务器进行验证。

    3. 检验新鲜通常有两个HTTP头进行控制 `Expires` 和 `Cache-Control` ：

        - HTTP1.0提供Expires，值为一个绝对时间表示缓存新鲜日期
        - HTTP1.1增加了Cache-Control: max-age=,值为以秒为单位的最大新鲜时间

3. 浏览器解析URL获取协议，主机，端口，path

4. 浏览器组装一个HTTP（GET）请求报文

5. 浏览器获取主机ip地址，过程如下：

    1. 浏览器缓存
    2. 本机缓存
    3. hosts文件
    4. 路由器缓存
    5. ISP DNS缓存
    6. DNS递归查询（可能存在负载均衡导致每次IP不一样）

6. 打开一个socket与目标IP地址，端口建立TCP链接，三次握手如下：

    1. 客户端发送一个TCP的SYN=1，Seq=X的包到服务器端口
    2. 服务器发回SYN=1， ACK=X+1， Seq=Y的响应包
    3. 客户端发送ACK=Y+1， Seq=Z

7. TCP链接建立后发送HTTP请求

8. 服务器接受请求并解析，将请求转发到服务程序，如虚拟主机使用HTTP Host头部判断请求的服务程序

9. 服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码
10. 处理程序读取完整请求并准备HTTP响应，可能需要查询数据库等操作
11. 服务器将响应报文通过TCP连接发送回浏览器
12. 浏览器接收HTTP响应，然后根据情况选择关闭TCP连接或者保留重用，关闭TCP连接的四次握手如下：

```ini
1.  主动方发送Fin=1， Ack=Z， Seq= X报文
```

```ini
1.  被动方发送ACK=X+1， Seq=Z报文
1.  被动方发送Fin=1， ACK=X， Seq=Y报文
1.  主动方发送ACK=Y， Seq=X报文
```

12. 浏览器检查响应状态吗：是否为1XX，3XX， 4XX， 5XX，这些情况处理与2XX不同
13. 如果资源可缓存，进行缓存
14. 对响应进行解码（例如gzip压缩）
15. 根据资源类型决定如何处理（假设资源为HTML文档）
16. 解析HTML文档，构件DOM树，下载资源，构造CSSOM树，执行js脚本，这些操作没有严格的先后顺序，以下分别解释
17. 构建DOM树：

```css
1.  Tokenizing：根据HTML规范将字符流解析为标记
```

```markdown
1.  Lexing：词法分析将标记转换为对象并定义属性和规则
1.  DOM construction：根据HTML标记关系将对象组成DOM树
```

18. 解析过程中遇到图片、样式表、js文件，启动下载
19. 构建CSSOM树：

markdown 复制代码

1. Node：根据标记创建节点
1. CSSOM：节点创建CSSOM树

20. 根据 DOM 树和 CSSOM 树构建渲染树:

markdown 复制代码

1. 从DOM树的根节点遍历所有可见节点，不可见节点包括：1）`script`,`meta`这样本身不可见的标签。2)被css隐藏的
1. 对每一个可见节点，找到恰当的CSSOM规则并应用
1. 发布可视节点的内容和计算样式

21. js解析如下:

markdown 复制代码

1. 浏览器创建`Document`对象并解析HTML，将解析到的元素和文本节点添加到文档中，此时`document.readystate`为`
1. HTML解析器遇到没有`async`和`defer`的`script`时，将他们添加到文档中，然后执行行内或外部脚本。这些脚本会
1. 当解析器遇到设置了`async`属性的`script`时，开始下载脚本并继续解析文档。脚本会在它下载完成后尽快执行，但
1. 当文档完成解析，`document.readState`变成`interactive`
1. 所有`defer`脚本会按照在文档出现的顺序执行，延迟脚本能访问完整文档树，禁止使用`document.write()`
1. 浏览器在`Document`对象上触发`DOMContentLoaded`事件
1. 此时文档完全解析完成，浏览器可能还在等待如图片等内容加载，等这些内容完成载入并且所有异步脚本完成载入和执行

22. 显示页面（HTML解析过程中会逐步显示页面）

17.对深拷贝和浅拷贝的理解

基础

要深入理解浅拷贝和深拷贝的原理，那就要涉及到一些基本数据类型和引用数据类型的知识了；

基本数据类型：number，string，boolean，null，undefined，symbol等；

引用数据类型：object（{}对象，数组[]），function函数等；

因为拷贝是对数据进行操作，所以我们得了解一下这两类的数据存储方式；

基本数据类型一般存储在栈中；引用数据类型一般存放在堆中

浅拷贝

结果：当改变原对象或者新对象的值时 会影响另外一个对象的值

深拷贝

原理：针对引用类型数据拷贝的是该引用类型的值

结果：拷贝对象和被拷贝对象值不会互相影响

区别

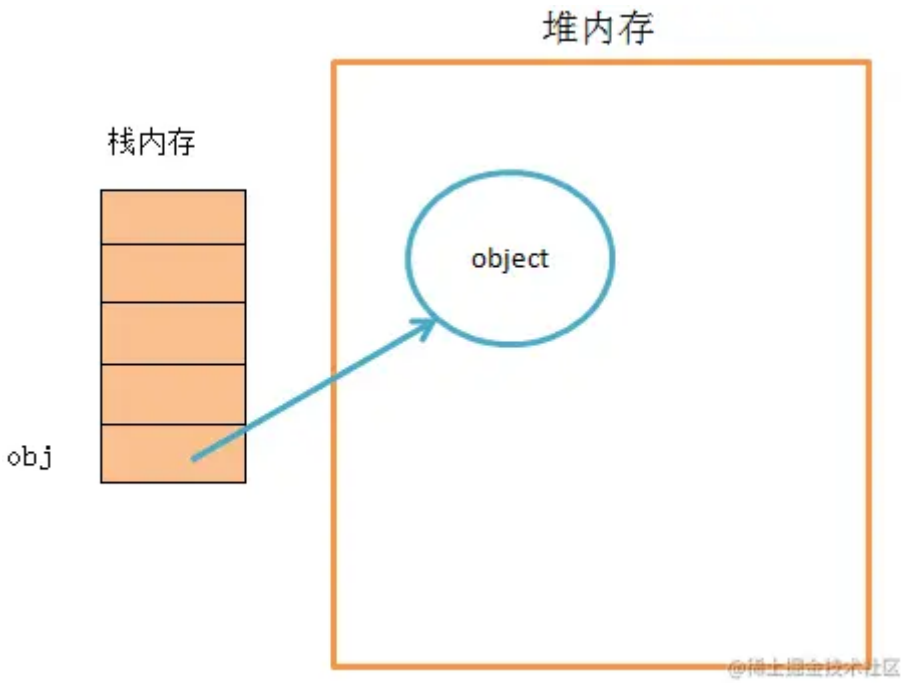
1. 深拷贝中既要拷贝基本数据类型也要拷贝引用类型的数据，也就是说拷贝一份完全一样的对象。

2. 浅拷贝中之拷贝基本数据类型，引用类型的数据只是拷贝了原来的引用，并没有把引用的数据也拷贝。

两种数据类型在内存中存放的位置

1 简单数据类型的值 存放在栈里边

2 引用类型的引用地址 存放在栈，值存在堆里面，然后这个引用地址的指针指向堆里边对应的值



什么是Dom?

DOM 是一项 W3C (World Wide Web Consortium) 标准。

DOM 定义了访问文档的标准：

> “W3C 文档对象模型（DOM）是中立于平台和语言的接口，它允许程序和脚本动态地访问、更新文档的内容、结构和样式。”

W3C DOM 标准被分为 3 个不同的部分：

- Core DOM - 所有文档类型的标准模型
- XML DOM - XML 文档的标准模型
- HTML DOM - HTML 文档的标准模型

什么是 HTML DOM?

HTML DOM 是 HTML 的标准*对象*模型和*编程接口*。它定义了：

- 作为*对象*的 HTML 元素
- 所有 HTML 元素的*属性*
- 访问所有 HTML 元素的*方法*
- 所有 HTML 元素的*事件*

换言之：HTML DOM 是关于如何获取、更改、添加或删除 HTML 元素的标准。

虚拟Dom是什么?

virtual DOM 虚拟 DOM，用普通 js 对象来描述 DOM 结构，它通过 JS 的 Object 对象模拟 DOM 中的节点，然后再通过特定的 render 方法将其渲染成真实的 DOM 节点。因为不是真实 DOM，所以称之为虚拟 DOM 。

总结来说一句话：virtual DOM 就是 Dom 的抽象化

为什么操作真实DOM的成本比较高?

(1) dom 树的实现模块和 js 模块是分开的这些跨模块的通讯增加了成本

(2) dom 操作引起的浏览器的回流和重绘，使得性能开销巨大。

原本在 pc 端是没有性能问题的，因为pc端的计算能力强，但是随着移动端的发展，越来越多的网页在智能手机上运行，而手机的性能参差不齐，会有性能问题。 我们之前用 jquery 在 pc 端写那些商城页面都没有问题，但放到移动端浏览器访问之后会发现除了首页会出现白屏之外在其他页面的操作并不流畅。

为何虚拟Dom性能更优?

虚拟 dom 是相对于浏览器所渲染出来的真实 dom 而言的，在 react ， vue 等技术出现之前，我们要改变页面展示的内容只能通过遍历查询 dom 树的方式找到需要修改的 dom 然后修改样式行为或者结构，来达到更新 ui 的目的。

这种方式相当消耗计算资源，因为每次查询 dom 几乎都需要遍历整颗 dom 树，如果建立一个与 dom 树对应的虚拟 dom 对象（ js 对象），以对象嵌套的方式来表示 dom 树及其层级结构，那么每次 dom 的更改就变成了对 js 对象的属性的增删改查，这样一来查找 js 对象的属性变化要比查询 dom 树的性能开销小。

Vue如何实现虚拟DOM的? （详解）

首先可以看看 vue 中 VNode 的结构

vnode.js

js 复制代码

```
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  ns: string | void;
  context: Component | void; // rendered in this component's scope
  functionalContext: Component | void; // only for functional component root nodes
  key: string | number | void;
  componentOptions: VNodeComponentOptions | void;
  componentInstance: Component | void; // component instance
  parent: VNode | void; // component placeholder node
  raw: boolean; // contains raw HTML? (server only)
  isStatic: boolean; // hoisted static node
  isRootInsert: boolean; // necessary for enter transition check
```

```
isOnce: boolean; // is a v-once node?

constructor (
  tag?: string,
  data?: VNodeData,
  children?: ?Array<VNode>,
  text?: string,
  elm?: Node,
  context?: Component,
  componentOptions?: VNodeComponentOptions
) {
  /*当前节点的标签名*/
  this.tag = tag
  /*当前节点对应的对象，包含了具体的一些数据信息，是一个VNodeData类型，可以参考VNodeData类型中的数据信息*
  this.data = data
  /*当前节点的子节点，是一个数组*/
  this.children = children
  /*当前节点的文本*/
  this.text = text
  /*当前虚拟节点对应的真实dom节点*/
  this.elm = elm
  /*当前节点的名字空间*/
  this.ns = undefined
  /*编译作用域*/
  this.context = context
  /*函数化组件作用域*/
  this.functionalContext = undefined
  /*节点的key属性，被当作节点的标志，用以优化*/
  this.key = data && data.key
  /*组件的option选项*/
  this.componentOptions = componentOptions
  /*当前节点对应的组件的实例*/
  this.componentInstance = undefined
  /*当前节点的父节点*/
  this.parent = undefined
  /*简而言之就是是否为原生HTML或只是普通文本，innerHTML的时候为true，textContent的时候为false*/
  this.raw = false
  /*静态节点标志*/
  this.isStatic = false
  /*是否作为跟节点插入*/
  this.isRootInsert = true
  /*是否为注释节点*/
  this.isComment = false
  /*是否为克隆节点*/
  this.isCloned = false
  /*是否有v-once指令*/
  this.isOnce = false
}
```

```
  // DEPRECATED: alias for componentInstance for backwards compat.
  /* istanbul ignore next https://github.com/answershuto/learnVue*/
  get child (): Component | void {
    return this.componentInstance
  }
}
```

这里对 VNode 进行稍微的说明：

- 所有对象的 context 选项都指向了 Vue 实例
- elm 属性则指向了其相对应的真实 DOM 节点

  vue 是通过 createElement 生成 VNode 再看看源码 create-element.js

js 复制代码

```
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}
```

上面可以看到 createElement 方法实际上是对 _createElement 方法的封装，对参数的传入进行了判断

js 复制代码

```
export function _createElement(
    context: Component,
    tag?: string | Class<Component> | Function | Object,
    data?: VNodeData,
    children?: any,
    normalizationType?: number
```

```
        process.env.NODE_ENV !== 'production' && warn(
            `Avoid using observed data object as vnode data: ${JSON.stringify(data)}\n` +
            'Always create fresh vnode data objects in each render!',
            context`
        )
        return createEmptyVNode()
    }
    // object syntax in v-bind
    if (isDef(data) && isDef(data.is)) {
        tag = data.is
    }
    if (!tag) {
        // in case of component :is set to falsy value
        return createEmptyVNode()
    }
    ...
    // support single function children as default scoped slot
    if (Array.isArray(children) &&
        typeof children[0] === 'function'
    ) {
        data = data || {}
        data.scopedSlots = { default: children[0] }
        children.length = 0
    }
    if (normalizationType === ALWAYS_NORMALIZE) {
        children = normalizeChildren(children)
    } else if ( === SIMPLE_NORMALIZE) {
        children = simpleNormalizeChildren(children)
    }
  // 创建VNode
    ...
}
```

可以看到 _createElement 接收5个参数：

context 表示 VNode 的上下文环境，是 Component 类型

tag 表示标签，它可以是一个字符串，也可以是一个 Component

data 表示 VNode 的数据，它是一个 VNodeData 类型

children 表示当前 VNode 的子节点，它是任意类型的

normalizationType 表示子节点规范的类型，类型不同规范的方法也就不一样，主要是参考

根据 normalizationType 的类型， children 会有不同的定义

```js
if (normalizationType === ALWAYS_NORMALIZE) {
    children = normalizeChildren(children)
} else if ( === SIMPLE_NORMALIZE) {
    children = simpleNormalizeChildren(children)
}
```

simpleNormalizeChildren 方法调用场景是 render 函数是编译生成的

normalizeChildren 方法调用场景分为下面两种：

render 函数是用户手写的 编译 slot、v-for 的时候会产生嵌套数组 无论是 simpleNormalizeChildren 还是 normalizeChildren 都是对 children 进行规范（使 children 变成了一个类型为 VNode 的 Array ），这里就不展开说了

规范化 children 的源码位置在： src/core/vdom/helpers/normalzie-children.js

在规范化 children 后，就去创建 VNode

```js
let vnode, ns
// 对tag进行判断
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // 如果是内置的节点，则直接创建一个普通VNode
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    // 如果是component类型，则会通过createComponent创建VNode节点
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
```

```
    vnode = createComponent(tag, data, context, children)
  }
```

稍微提下 `createComponent` 生成 `VNode` 的三个关键流程：

- 构造子类构造函数 `Ctor`
- `installComponentHooks` 安装组件钩子函数
- 实例化 `vnode`

小结

`createElement` 创建 VNode 的过程，每个 `VNode` 有 `children`， `children` 每个元素也是一个 `VNode`，这样就形成了一个虚拟树结构，用于描述真实的 `DOM` 树结构

19.数据类型存储以及堆栈内存是什么?

基本数据类型：直接存储在栈内存中，占据空间小，大小固定，属于被频繁使用的数据。指的是保存在栈内存中的简单数据段； `number` `string` `Boolean`

引用数据类型：同时存储在栈内存与堆内存中，占据空间大，大小不固定。

引用数据：类型将指针存在栈中，将值存在堆中。 当我们把对象值赋值给另外一个变量时，复制的是对象的指针，指向同一块内存地址，意思是，变量中保存的实际上只是一个指针，这个指针指向内存堆中实际的值，数组 对象

20.堆(heap)和栈(stack)有什么区别存储机制?

栈： 是一种连续储存的数据结构，具有先进后出后进先出的性质。

通常的操作有入栈（压栈），出栈和栈顶元素。想要读取栈中的某个元素，就是将其之间的所有元素出栈才能完成。

堆： 是一种非连续的树形储存数据结构，具有队列优先,先进先出； 每个节点有一个值，整棵树是经过排序的。特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。常用来实现优先队列，存取随意。

- `===` 属于严格判断，直接判断两者类型是否相同，如果两边的类型不一致时，不会做强制类型准换，不同则返回 `false` 如果相同再比较大小，不会进行任何隐式转换对于引用类型来说，比较的都是引用内存地址，所以 `===` 这种方式的比较，除非两者存储的内存地址相同才相等，反之 `false`

- `==` 二等表示值相等。判断操作符两边对象或值是否相等类型可以不同，如果两边的类型不一致，则会进行强制类型转化后再进行比较，使用 `Number()` 转换成 `Number` 类型在进行判断。**例外规则，** `null==undefined，null/undefined` **进行运算时不进行隐式类型转换。通常把值转为Boolean值，进行条件判断。**

  `Boolean(null)===Boolean(undefined)>false===false` 结果为 `true`

- `Object.is()` 在 `===` 基础上特别处理了 `NaN,-0,+0` ,保证 `-0` 与 `+0` 不相等，但NaN与NaN相等

js 复制代码

==操作符的强制类型转换规则
字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。
其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。
null 和 undefined 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。
对象和非对象之间的相等比较，对象先调用 ToPrimitive 抽象操作后，再进行比较。
如果一个操作值为 NaN ，则相等比较返回 false（ NaN 本身也不等于 NaN ）。
如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回true，

```
'1' == 1 // true
'1' === 1 // false
NaN == NaN //false
+0 == -0 //true
+0 === -0 // true
Object.is(+0,-0) //false
Object.is(NaN,NaN) //true
```

22. 函数柯里化（卡瑞化、加里化）？

概念：把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数而且返回结果的新函数的技术。 容易理解的概念：Currying概念其实很简单，只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数（主要是利用闭包实现的）。

特点：

①接收单一参数，将更多的参数通过回调函数来搞定；

②返回一个新函数，用于处理所有的想要传入的参数；

③需要利用 `call/apply` 与 `arguments` 对象收集参数；

④返回的这个函数正是用来处理收集起来的参数。

作用：能进行部分传值，而传统函数调用则需要预先确定所有实参。如果你在代码某一处只获取了部分实参，然后在另一处确定另一部分实参，这个时候柯里化和偏应用就能派上用场。

用途：我认为函数柯里化是对闭包的一种应用形式，延迟计算、参数复用、动态生成函数(都是闭包的用途)。

柯里化函数例子

柯里化函数：把一个多参数的函数转化为单参数函数的方法。并且返回接受余下的参数而且返回结果的新函数的技术。

我的理解就是将一个接受多个参数的函数，转化为接收一个参数，并且不改变输出结果的一种办法。我觉得这就是js的柯里化函数


```
// 简单的相加函数
var add = function (x,y) {
    return x + y
}
// 调用：
add(1,2)

// 柯里化以后
var add = function (x) { //柯里化函数(闭包)
    return function (y) {
        return x + y
    }
}
add(1)(2)
```

什么是高阶函数?

高阶函数只是 将函数作为参数 函数的返回值返回值是函数

```
function higherOrderFunction(param,callback){
    return callback(param);
}
```

23.关于构造函数

new的原理

new 实际上是在堆内存中开辟一个空间。

- ①创建一个空对象，构造函数中的this指向这个空对象；
- ②这个新对象被执行[[原型]]连接；
- ③执行构造函数方法，属性和方法被添加到 this 引用的对象中；
- ④如果构造函数中没有返回其它对象，那么返回 this ，即创建的这个的新对象，否则，返回构造函数中返回的对象。

```
function _new(){
    let target = {};   //创建的新对象
    let [constructor,...args] = [...arguments];
       //执行[[原型]]连接,target是constructor的实例
    target.__proto__ = constructor.prototype;
        //执行构造函数,将属性或方法添加到创建的空对象上
    let result = constructor.prototype;
    if(result && (typeof (result) == "object" || typeof (result) == "function")){
           //如果构造函数执行的结构返回的是一个对象,那么返回这个对象
        return result;
    }
       //如果构造函数返回的不是一个对象,返回创建的对象
    return target;
}
```

自己理解的 new ：

new 实际上是在堆内存中开辟一个新的空间。首先创建一个空对象 obj ，然后呢， 把这个空对象的原型（__proto__）和构造函数的原型对象（constructor.prototype）连接(说白了就是等于)； 然后执行函数中的代码，就是为这个新对象添加属性和方法。最后进行判断其返回值，如果构造函数返回的是一个对象， 那就返回这个对象，如果不是，那就返回我们创建的对象。

垃圾回收机制和内存机制

垃圾回收

浏览器的 `js` 具有自动垃圾回收机制，垃圾回收机制也就是自动内存管理机制，垃圾收集器会定期的找出那些不在继续使用的变量，然后释放内存。但是这个过程不是实时的，因为 `GC` 开销比较大并且时停止响应其他操作，所以垃圾回收器会按照固定的时间间隔周期性的执行。

内存泄露

如果 那些不再使用的变量，它们所占用的内存 不去清除的话就会造成内存泄漏

内存泄露其实就是我们的程序中已经动态分配的堆内存，由于某些原因没有得到释放，造成系统内存的浪费导致程序运行速度减慢甚至系统崩溃等严重后果。

比如说：

1、闭包：在闭包中引入闭包外部的变量时，当闭包结束时此对象无法被垃圾回收（GC）。

2、 `DOM` ： 当原有的 `DOM` 被移除时，子结点引用没有被移除则无法回收

3、 `Times` 计时器泄露

作用域

1、作用域

作用域就是一个变量可以使用的范围，主要分为全局作用域和函数作用域

全局作用域就是 `Js` 中最外层的作用域

函数作用域是js通过函数创建的一个独立作用域，函数可以嵌套，所以作用域也可以嵌套

`Es6` 中新增了块级作用域（由大括号包裹，比如： `if(){},for(){}` 等）

2、自由变量

当前作用域外的变量都是自由变量，一个变量在当前作用域没有定义，但是被使用了，就会向上级作用域，一层一层依次查找，直至找到为止，如果全局作用域都没有找到这个变量就会报

3、变量提升

每个 `var` 声明的变量， `function` 声明的函数存在变量提升。 `let const` 不存在变量提升

在js中声明之前未定义，会在 `js` 的最上方会形成一个预解析池，用来存储声明了但没有先定义的变量名

4、作用域链：

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，我们可以访问到外层环境的变量和 函数 ， 简单来说：内部函数访问外部函数的变量这种链式查找的机制被称为作用域链