Q&A

基础题

Q:盒模型是什么？

A:W3C标准盒模型中，content-box是所有元素box-sizing的默认置（非IE旧版本浏览器）参考：各个区域从内到外 content，padding，border，margin， width 在 border-box 中包含border，padding，content 三部分的宽度，content-box 中只包含content

Q: **Flex**布局

A:Flex是Flexible Box的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。 布局的传统解决方案，基于盒状模型，依赖display属性 + position属性 + float属性。它对于那些特殊布局非常不方便，比如，垂直居中就不容易实现。

简单的分为容器属性和元素属性 容器的属性：

flex-direction：决定主轴的方向（即子item的排列方法） .box { flex-direction: row | row-reverse | column | column-reverse; }

flex-wrap：决定换行规则 .box{ flex-wrap: nowrap | wrap | wrap-reverse; }

flex-flow： .box { flex-flow: <flex-direction> || <flex-wrap>; }

justify-content：对其方式，水平主轴对齐方式

align-items：对齐方式，竖直轴线方向

项目的属性（元素的属性）：

order属性：定义项目的排列顺序，顺序越小，排列越靠前，默认为0

flex-grow属性：定义项目的放大比例，即使存在空间，也不会放大

flex-shrink属性：定义了项目的缩小比例，当空间不足的情况下会等比例的缩小，如果定义个item的flow-shrink为0，则为不缩小

flex-basis属性：定义了在分配多余的空间，项目占据的空间。

flex：是flex-grow和flex-shrink、flex-basis的简写，默认值为0 1 auto。

align-self：允许单个项目与其他项目不一样的对齐方式，可以覆盖align-items，默认属性为auto，表示继承父元素的align-items

Q: 原型链

A: JavaScript是面向对象的，每个实例对象都有一个\_\_proto\_\_属性，该属性指向它的原型对象，这个实例对象的构造函数有一个原型属性prototype,与实例的proto属性指向同一个对象。当一个对象在查找一个属性的时候，自身没有就会根据\_\_proto\_\_向它的原型进行查找，如果都没有，则向它的原型的原型继续查找，直到查到Object.prototype.proto\_为null,这样也就形成了原型链

Q:new 操作符做了什么

A ： 1. 创建一个空的对象，即{}

1. 空对象的原型指向构造函数的原型，即设置该对象的构造函数
2. 让this指向新创建的空对象，即新创建的对象作为this的上下文
3. 判断返回值的类型，如果是值类型就是返回新的创建对象，如果是引用类型就返回引用类型的对象。（如果没有返回对象类型object包括Functoin, Array, Date, RegExg, Error），那么new表达式中的函数调用将返回该对象的引用）

Q:虚拟dom

A:  当数据发生变化，比如setState时，会引起组件重新渲染，整个UI都会以virtual dom的形式重新渲染

 然后收集差异也就是diff新的virtual dom和老的virtual dom的差异

 最后把差异队列里的差异，比如增加节点、删除节点、移动节点更新到真实的DOM上

Q: diff 算法

A: 把树形结构按照层级分解，只比较同级元素。不同层级的节点只有创建和删除操作  
  
·给列表结构的每个单元添加唯一的 key 属性，方便比较。  
  
·React 只会匹配相同 class 的component（这里面的 class 指的是组件的名字）  
  
·合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.  
  
·选择性子树渲染。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能

Q:如何理解react的单向数据流

A: 在 React 中，数据仅朝一个方向流动，即从父组件流向子组件。如果数据在兄弟子组件之间共享，那么数据应该存储在父组件，并同时传递给需要数据的两个子组件。

数据从父组件流向子组件，数据更新发送到父组件，父组件会进行实际的更改。

数据位于父组件中，并向下传递给子组件，但是父组件和子组件都可以使用数据，(父传子)

然而：

如果必须更新数据的话，则只有父组件应该进行更新；

如果子组件需要更改数据，它将会更新的数据发送给父组件，由父组件完成更改，父组件执行更改后，将会更新的数据传递给子组件。(子传父)

Q: **display:none**和**visibilty:hidden**的区别

**A:** 1. visibility：hidden，该元素隐藏起来了，但不会改变页面布局，但是不会触发该元素已经绑定的事 件

2. display：none，把元素隐藏起来，并且会改变页面布局，可以理解成在页面中把该元素删除掉。

Q : javascript 的基本类型有那些 ？ 引用类型呢 ？有啥区别 ？

A ：基本类型 Undefined、Null、Boolean、Number和String **symbo**（**ES6**）

复杂类型 Object

基本类型的变量是存放在栈内存（Stack）里的

基本类型的值是按值访问的

基本类型的比较是它们的值的比较

引用类型的值是保存在堆内存（Heap）中的对象（Object）

引用类型的值是按引用访问的

引用类型的值是可变的

引用类型的比较是引用的比较

Q ：如何判断一个一个变量是否是数组

**A** ：Object.prototype.call.toString()

Instanceof Array.isArray

Q ：空数组的布尔值是什么？

A ：true

Q : es6了解吗？ 用过哪些？ …

A ：promise，await/async，let、const、块级作用域、箭头函数、class、解构赋值

Q: 啥是promise

A: Promise是一个对象，保存着未来将要结束的事件，她有两个特征:

1、对象的状态不受外部影响，Promise对象代表一个异步操作，有三种状态，pending进行中，fulfilled已成功，rejected已失败，只有异步操作的结果，才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，这也就是promise名字的由来

2、一旦状态改变，就不会再变，promise对象状态改变只有两种可能，从pending改到fulfilled或者从pending改到rejected，只要这两种情况发生，状态就凝固了，不会再改变，这个时候就称为定型resolved,

api .then .catch .race .all .finally Promise.allSettled

Q : 事件监听

A ：addEventListener()方法，用于向指定元素添加事件句柄，它可以更简单的控制事件，语法为

element.addEventListener(event, function, useCapture);

第一个参数是事件的类型(如 "click" 或 "mousedown").

第二个参数是事件触发后调用的函数。

第三个参数是个布尔值用于描述事件是冒泡还是捕获。该参数是可选的。

事件传递有两种方式，冒泡和捕获

事件传递定义了元素事件触发的顺序，如果你将P元素插入到div元素中，用户点击P元素，

冒泡中，内部元素先被触发，然后再触发外部元素，

捕获中，外部元素先被触发，在触发内部元素，

Q: 事件委托(代理)

A: 子元素身上的事件会冒泡到父元素身上, 故监听父元素的事件，监听的event对象中有事件源

可用来分辨是哪一个子元素 提高效率， 不用大量监听子元素事件

Q: 数组和伪数组的区别 举个伪数组例子

A: 1. 伪数组拥有length属性，其它属性（索引）为非负整数(对象中的索引会被当做字符串来处理，这里

你可以当做是个非负整数串来理解)

2.不具有数组所具有的方法

常见的伪数组 arguments，DOM 对象列表（比如通过 document.getElementsByTags 得到的列表）

Q: get post 的区别

A:**get**是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。**post**是通过**HTTP** **post**机制，将表单内各个字段与其内容放置在HTML   HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。     2、对于**get**方式，服务器端用Request.QueryString获取变量的值，对于**post**方式，服务器端用Request.Form获取提交的数据。两种方式的参数都可以用Request来获得。     3、**get**传送的数据量较小，不能大于2KB。**post**传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。     4、**get**安全性非常低，**post**安全性较高。

通过**get**方法提交数据，可能会带来安全性的问题。比如一个登陆页面。当通过**get**方法提交数据时，用户名和密码将出现在URL上。如果：   １、 登陆页面可以被浏览器缓存；   ２、 其他人可以访问客户的这台机器。   那么，别人即可以从浏览器的历史记录中，读取到此客户的账号和密码。所以，在某些情况下，**get**方法会带来严重的安全性问题。

**post**是通过**HTTP** **post**机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。

Q: 请解释**React**中**props**和**state**的区别？

**A**： • 1.props

* + 函数组件的props就是一个函数的入参组件
  + 类组件：this.props 包括被该组件调用或者定义的props
* 2.state
  + 组件中的state包含了随时可以发生变化的数据
  + state由用户定义，它是一个普通javascript对象
* 3.区别
  + props是传递组件的(类似函数的形参)，而state是在组件内被组件自己管理的(类似在一个函数内声明变量)。
  + props是不可修改的，所有React组件必须向纯函数一样保护它们的props不被更改
  + state是在组件中创建的，一般在constructor中初始化state
  + state是多变的、可修改的，每次setState都异步更新的

Q: 面向对象的继承方式

**A**：原型链继承

核心： 将父类的实例作为子类的原型

特点：

非常纯粹的继承关系，实例是子类的实例，也是父类的实例

父类新增原型方法/原型属性，子类都能访问到

简单，易于实现

缺点：

要想为子类新增属性和方法，不能放到构造器中

无法实现多继承

来自原型对象的所有属性被所有实例共享

创建子类实例时，无法向父类构造函数传参

构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

特点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类传递参数

可以实现多继承（call多个父类对象）

缺点：

实例并不是父类的实例，只是子类的实例

只能继承父类的实例属性和方法，不能继承原型属性/方法

无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

实例继承

核心：为父类实例添加新特性，作为子类实例返回

特点：

不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

实例是父类的实例，不是子类的实例

不支持多继承

拷贝继承

特点：

支持多继承

缺点：

效率较低，内存占用高（因为要拷贝父类的属性）

组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

特点：

可以继承实例属性/方法，也可以继承原型属性/方法

既是子类的实例，也是父类的实例

不存在引用属性共享问题

可传参

函数可复用

寄生组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

中等题

Q: react 生命周期 以及每个生命周期的作用

A; componentWillMount、

Render、componentDidMount、componentWillRecieveProps、 shouldComponentUpdate, componentWillUpdate、render、componentDidUpdate、componentWillUnmount

* [1. 挂载卸载过程](#1-%E6%8C%82%E8%BD%BD%E5%8D%B8%E8%BD%BD%" \t "_blank)
  + [1.1.constructor()](#11constructor" \t "_blank)
  + [1.2.componentWillMount()](#12componentwillmount" \t "_blank)
  + [1.3.componentDidMount()](#13componentdidmount" \t "_blank)
  + [1.4.componentWillUnmount ()](#14componentwillunmount" \t "_blank)
* [2. 更新过程](#2-%E6%9B%B4%E6%96%B0%E8%BF%87%E7%A8%8B" \t "_blank)
  + [2.1. componentWillReceiveProps (nextProps)](#21-componentwillreceiveprops-nextprops" \t "_blank)
  + [2.2.shouldComponentUpdate(nextProps,nextState)](#22shouldcomponentupdatenextpropsnextsta" \t "_blank)
  + [2.3.componentWillUpdate (nextProps,nextState)](#23componentwillupdate-nextpropsnextstat" \t "_blank)
  + [2.4.componentDidUpdate(prevProps,prevState)](#24componentdidupdateprevpropsprevstate" \t "_blank)
  + [2.5.render()](#25render" \t "_blank)
* [3. React新增的生命周期(个人补充)](#3-react%E6%96%B0%E5%A2%9E%E7%9A%84%E7%9" \t "_blank)
  + [3.1. getDerivedStateFromProps(nextProps, prevState)](#31-getderivedstatefrompropsnextprops-pr" \t "_blank)
  + [3.2. getSnapshotBeforeUpdate(prevProps, prevState)](#32-getsnapshotbeforeupdateprevprops-pre" \t "_blank)

1. 挂载卸载过程

1.1.constructor()

constructor()中完成了React数据的初始化，它接受两个参数：props和context，当想在函数内部使用这两个参数时，需使用super()传入这两个参数。  
注意：只要使用了constructor()就必须写super(),否则会导致this指向错误。

1.2.componentWillMount()

componentWillMount()一般用的比较少，它更多的是在服务端渲染时使用。它代表的过程是组件已经经历了constructor()初始化数据后，但是还未渲染DOM时。

1.3.componentDidMount()

组件第一次渲染完成，此时dom节点已经生成，可以在这里调用ajax请求，返回数据setState后组件会重新渲染

1.4.componentWillUnmount ()

在此处完成组件的卸载和数据的销毁。

2. 更新过程

2.1. componentWillReceiveProps (nextProps)

1. 在接受父组件改变后的props需要重新渲染组件时用到的比较多
2. 接受一个参数nextProps
3. 通过对比nextProps和this.props，将nextProps的state为当前组件的state，从而重新渲染组件

2.2.shouldComponentUpdate(nextProps,nextState)

1. 主要用于性能优化(部分更新)
2. 唯一用于控制组件重新渲染的生命周期，由于在react中，setState以后，state发生变化，组件会进入重新渲染的流程，在这里return false可以阻止组件的更新
3. 因为react父组件的重新渲染会导致其所有子组件的重新渲染，这个时候其实我们是不需要所有子组件都跟着重新渲染的，因此需要在子组件的该生命周期中做判断

2.3.componentWillUpdate (nextProps,nextState)

shouldComponentUpdate返回true以后，组件进入重新渲染的流程，进入componentWillUpdate,这里同样可以拿到nextProps和nextState。

2.4.componentDidUpdate(prevProps,prevState)

组件更新完毕后，react只会在第一次初始化成功会进入componentDidmount,之后每次重新渲染后都会进入这个生命周期，这里可以拿到prevProps和prevState，即更新前的props和state。

2.5.render()

render函数会插入jsx生成的dom结构，react会生成一份虚拟dom树，在每一次组件更新时，在此react会通过其diff算法比较更新前后的新旧DOM树，比较以后，找到最小的有差异的DOM节点，并重新渲染。

3. React新增的生命周期(个人补充)

React新增生命周期

3.1. getDerivedStateFromProps(nextProps, prevState)

代替componentWillReceiveProps()。  
老版本中的componentWillReceiveProps()方法判断前后两个 props 是否相同，如果不同再将新的 props 更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。

这两者最大的不同就是:  
在 componentWillReceiveProps 中，我们一般会做以下两件事，一是根据 props 来更新 state，二是触发一些回调，如动画或页面跳转等。

1. 在老版本的 React 中，这两件事我们都需要在 componentWillReceiveProps 中去做。
2. 而在新版本中，官方将更新 state 与触发回调重新分配到了 getDerivedStateFromProps 与 componentDidUpdate 中，使得组件整体的更新逻辑更为清晰。而且在 getDerivedStateFromProps 中还禁止了组件去访问 this.props，强制让开发者去比较 nextProps 与 prevState 中的值，以确保当开发者用到 getDerivedStateFromProps 这个生命周期函数时，就是在根据当前的 props 来更新组件的 state，而不是去做其他一些让组件自身状态变得更加不可预测的事情。

3.2. getSnapshotBeforeUpdate(prevProps, prevState)

代替componentWillUpdate。  
常见的 componentWillUpdate 的用例是在组件更新前，读取当前某个 DOM 元素的状态，并在 componentDidUpdate 中进行相应的处理。  
这两者的区别在于：

1. 在 React 开启异步渲染模式后，在 render 阶段读取到的 DOM 元素状态并不总是和 commit 阶段相同，这就导致在  
   componentDidUpdate 中使用 componentWillUpdate 中读取到的 DOM 元素状态是不安全的，因为这时的值很有可能已经失效了。
2. getSnapshotBeforeUpdate 会在最终的 render 之前被调用，也就是说在 getSnapshotBeforeUpdate 中读取到的 DOM 元素状态是可以保证与 componentDidUpdate 中一致的。  
   此生命周期返回的任何值都将作为参数传递给componentDidUpdate（）。

Q: 跨域？解决？

A：同源策略/SOP（Same origin policy）是一种约定，由Netscape公司1995年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSFR等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

1. JsonP 原理 get？ Post? 安全性
2. Cros？
3. 代理？

Q: git pull 与 git fetch 之间有什么区别

A：简单来说，git pull 是 git fetch git merge。

当你使用 pull，Git 会试着自动为你完成工作。它是上下文（工作环境）敏感的，所以 Git 会把所有拉取的提交合并到你当前处理的分支中。pull 则是 自动合并提交而没有让你复查的过程。如果你没有细心管理你的分支，你可能会频繁遇到冲突。

当你 fetch，Git 会收集目标分支中的所有不存在的提交，并将这些提交存储到本地仓库中。但Git 不会把这些提交合并到当前分支中。这种处理逻辑在当你需要保持仓库更新，在更新文件时又希望处理可能中断的事情时，这将非常实用。而将提交合并到主分支中，则该使用 merge。

Q: async await 是什么？

A: es6 生成器 fn\* 的语法糖 async function 返回一个promise

化异步为同步 await 阻塞 async function

Q: typescript

A: 是js有了强类型校验 编译后还是js

Interface 跟 type 有啥区别？

如何理解泛型？

Q: event Loop;

A:Javascript在JS引擎中在主线程上顺序解析和执行，解析setTimeout会将回调函数放入“宏任务”消息队列，解析Promise会首先执行构造函数中的回调函数，执行resolve方法会将then中的回调函数放入微任务消息队列。引擎会先执行全部微任务，当全部微任务都执行完毕再执行宏任务。

优先级 new promise > promise() > setTImeout

Q: xss 与 csrf 是什么？ 如何防御

A:CSRF：跨站请求伪造，可以理解为攻击者盗用了用户的身份，以用户的名义发送了恶意请求，比如用户登录了一个网站后，立刻在另一个ｔａｂ页面访问量攻击者用来制造攻击的网站，这个网站要求访问刚刚登陆的网站，并发送了一个恶意请求，这时候CSRF就产生了，比如这个制造攻击的网站使用一张图片，但是这种图片的链接却是可以修改数据库的，这时候攻击者就可以以用户的名义操作这个数据库，防御方式的话：使用验证码，检查https头部的refer，使用token

XSS：跨站脚本攻击，是说攻击者通过注入恶意的脚本，在用户浏览网页的时候进行攻击，比如获取cookie，或者其他用户身份信息，可以分为存储型和反射型，存储型是攻击者输入一些数据并且存储到了数据库中，其他浏览者看到的时候进行攻击，反射型的话不存储在数据库中，往往表现为将攻击代码放在url地址的请求参数中，防御的话为cookie设置httpOnly属性，对用户的输入进行检查，进行特殊字符过滤

Q: 说一下**React.Component**和**React.PureComponent**的区别

**A**：React.Component需要自己实现shouldComponentUpdate()

React.PureComponent通过props和state的钱比较来实现shouldComponentUpdate()

如果定义了shouldComponentUpdate()，无论组件是否是PureComponent，都会执行shouldComponentUpdate()结果来判断是否update。如果组件未实现shouldComponentUpdate(),则判断该组件是否是PureComponent，如果是的话，会对新旧props、state进行比较，一旦新旧不一致，就会触发update

React.PureComponent优点:不需要开发者自己实现shouldComponentUpdate，就可以进行简单的判断来提升性能

React.PureComponent缺点：可能会因深层的数据不一致而产生错误的否定判断，从而shouldComponentUpdate结果返回false，界面得不到更新。

Q: function 组件有refs么？ 如何创建一个类似class组件的refs

A； const refs= {}

( ref => refs[xxx] = ref )

Q: **React**里**setState**到底是异步还是同步？

**A:** 两种情况都有

1. 1.setState只在合成事件和钩子函数中是"异步"的，在原生时间和setTimeout中是同步的
2. 2.setState的"异步"并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的"异步"，当然可以通过第二个参数setState(partialState,callback)中callback拿到更新后的结果。 partialState-->局部状态
3. 3.setState的批量更新优化也是建立在"异步"(合成事件、钩子函数)之上的，在原生事件和setTimeout中不会批量更新，在"异步"中如果对同一个值进行多次setState,setState的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时setState多个不同的值，在更新是会对其进行合并批量更新

Q: useLayoutEffect和useEffect的区别 react hook （react >= 16.8）

A: useEffect：渲染内容更新到DOM ->执行操作；

useLayoutEffect：执行操作 -> 渲染内容更新到DOM；

Q: this的指向 哪几种

A : 默认绑定：全局环境中，this默认绑定到window。

隐式绑定：一般地，被直接对象所包含的函数调用时，也称为方法调用，this隐式绑定到该直接对象。

隐式丢失：隐式丢失是指被隐式绑定的函数丢失绑定对象，从而默认绑定到window。

显式绑定：通过call()、apply()、bind()方法把对象绑定到this上，叫做显式绑定。

new绑定：如果函数或者方法调用之前带有关键字new，它就构成构造函数调用。对于this绑定来说，称为new绑定。

【1】构造函数通常不使用return关键字，它们通常初始化新对象，当构造函数的函数体执行完毕时，它会显式返回。在这种情况下，构造函数调用表达式的计算结果就是这个新对象的值。

【2】如果构造函数使用return语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值，同时使用这个新对象作为调用结果。

【3】如果构造函数显式地使用return语句返回一个对象，那么调用表达式的值就是这个对象。

Q:显示绑定 call apply bind的区别

A：Call和apply的作用是一模一样的，只是传参的形式有区别而已

apply，call方法会使函数立即执行 ，bind不会使函数立即执行

Q: useEffect 是在 render 前执行？ 还是 render 后执行？

A: 后

Q : useCallback 和 useMemo用过么？为什么在class组件内的 constructor 内要多次执行函数的bind操作？

A: 性能优化。减少重新赋值。

Q；class 组件 跟 function 有什么区别

A；function 无this 无生命周期函数 可用hook api……. 执行机制

Q: **react-router**里的**Link**标签和**a**标签有什么区别？

**A**：从最终渲染的DOM来看，这两者都是链接，都是a标签，区别是： Link标签是react-router里实现路由跳转的链接，一般配合Route使用，react-router接下了其默认的链接跳转行为，区别于传统的页面跳转，Link标签的"跳转"行为只会触发相匹配的Route对应的页面内容更新，而不会刷新整个页面

Link标签做的三件事情：

* 1.有onclick那就执行onclick
* 2.click的时候阻止a标签默认事件
* 3.根据跳转href(即使是to)，用history(web前端路由两种方式之一，history&hash)跳转，此时只是链接变了，并没有刷新页面

而标签就是普通的超链接了，用于从当前页面跳转到href指向的里一个页面(非锚点情况)

Q: **React**组件中怎么做事件代理？它的原理是什么？

**A**：React基于Virtual DOM实现了一个SyntheticEvent(合成事件层)，定义的事件处理器会接收到一个合成事件对象的实例，它符合W3C标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上

在React底层，主要对合成事件做了：事件委派和自动绑定

事件委派：React会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数

自动绑定：React组件中，每个方法的上下文都会指向该组件的实例，即自动绑定this为当前组件