https://juejin.cn/post/6844903824683958286

开发中的常见问题

1.为何必须引用React
2.自定义的React组件为何必须大写
  注意：babel在编译时会判断JSX中组件的首字母，当首字母为小写时，其被认定为原生DOM标签，createElement的第一个变量被编译为字符串；当首字母为大写时，其被认定为自定义组件，createElement的第一个变量被编译为对象；
3.React如何防止XSS
4.React的Diff算法和其他的Diff算法有何区别
5.key在React中的作用
5.如何写出高性能的React组件

如果你对上面几个问题还存在疑问，说明你对React的虚拟DOM以及Diff算法实现原理还有所欠缺，那么请好好阅读本篇文章吧。

虚拟dom和原生dom

原生的JavaScript程序中，我们直接对DOM进行创建和更改，而DOM元素通过我们监听的事件和我们的应用程序进行通讯

1、React整个的渲染机制就是React会调用render()函数构建一棵Dom树，
2、在state/props发生改变的时候，render()函数会被再次调用渲染出另外一棵树，重新渲染所有的节点，构造出新的虚拟Dom tree跟原来的Dom tree用Diff算法进行比较，找到需要更新的地方批量改动，再渲染到真实的DOM上，由于这样做就减少了对Dom的频繁操作，从而提升的性能。

+ 当我们需要创建或更新元素时，React首先会让这个VitrualDom对象进行创建和更改，然后再将VitrualDom对象渲染成真实DOM；

+ 当我们需要对DOM进行事件监听时，首先对VitrualDom进行事件监听，VitrualDom会代理原生的DOM事件从而做出响应。

为何使用虚拟DOM

- 1.提高开发效率
- 2.跨浏览器兼容
  React基于VitrualDom自己实现了一套自己的事件机制，自己模拟了事件冒泡和捕获的过程，采用了事件代理，批量更新等方法，抹平了各个浏览器的事件兼容性问题。
- 3.跨平台兼容:VitrualDom为React带来了跨平台渲染的能力
  只需要告诉React你想让视图处于什么状态，React则通过VitrualDom确保DOM与该状态相匹配。你不必自己去完成属性操作、事件处理、DOM更新，React会替你完成这一切。

原因4：关于提升性能:如果是首次渲染，VitrualDom不具有任何优势，甚至它要进行更多的计算，消耗更多的内存。

直接操作DOM是非常耗费性能的，这一点毋庸置疑。但是React使用VitrualDom也是无法避免操作DOM的。

如果是首次渲染，VitrualDom不具有任何优势，甚至它要进行更多的计算，消耗更多的内存。

VitrualDom的优势在于React的Diff算法和批处理策略，React在页面更新之前，提前计算好了如何进行更新和渲染DOM。

实际上，这个计算过程我们在直接操作DOM时，也是可以自己判断和实现的，但是一定会耗费非常多的精力和时间，而且往往我们自己做的是不如React好的。所以，在这个过程中React帮助我们"提升了性能"。
所以，我更倾向于说，VitrualDom帮助我们提高了开发效率，在重复渲染时它帮助我们计算如何更高效的更新，而不是它比DOM操作更快。

JSX和createElement

所有的JSX代码最后都会转换成React.createElement(...)，Babel帮助我们完成了这个转换的过程。

```
// 第一种是使用JSX:JSX只是为 React.createElement(component, props, ...children)方法提供的语法糖。
class Hello extends Component {
  render() {
    return <div>Hello ConardLi</div>;
  }
}

// 第二种是直接使用React.createElement编写：

class Hello extends Component {
  render() {
    return React.createElement('div', null, `Hello ConardLi`);
  }
}
如下面的JSX
<div>
  <img src="avatar.png" className="profile" />
  <Hello />
</div>;
    |
    |
    |
    V
将会被Babel转换为:
React.createElement("div", null, React.createElement("img", {
  src: "avatar.png",
  className: "profile"
}), React.createElement(Hello, null));

注意：babel在编译时会判断JSX中组件的首字母，当首字母为小写时，其被认定为原生DOM标签，createElement的第一个变量被编译为字符串；当首字母为大写时，其被认定为自定义组件，createElement的第一个变量被编译为对象；
```

创建虚拟DOM

```
<div class="title">
      <span>Hello ConardLi</span>
      <ul>
        <li>苹果</li>
        <li>橘子</li>
      </ul>
</div>
```

```
    |
    |
    |
    v
```

在React可能存储为这样的JS代码：

```
const VitrualDom = {
  type: 'div',
  props: { class: 'title' },
  children: [
    {
      type: 'span',
      children: 'Hello ConardLi'
    },
    {
      type: 'ul',
      children: [
        { type: 'li', children: '苹果' },
        { type: 'li', children: '橘子' }
      ]
    }
  ]
}
```

那么React是如何将我们的代码转换成这个结构的呢，下面我们来看看createElement函数的具体实现（文中的源码经过精简）

```
ReactElement.createElement=function (type,config,children) {
    // body...
    var propName;
    var props = {}
    var key = null
    var ref = null
    var self=null
    var source = null
    if(config!=null){
        //1.处理props
    }
    //2.获取子元素
    if (type&&type.defaultProps) {
        //3.处理默认props
    }
    return
ReactElement(type,key,ref,self,source,ReaceCurrentOwner.current,props)
}
```

createElement函数内部做的操作很简单，将props和子元素进行处理后返回一个ReactElement对象，下面我们来逐一分析：

(1).处理props

```
/*
1.将特殊属性ref、key从config中取出并赋值
2.将特殊属性self、source从config中取出并赋值
3.将除特殊属性的其他属性取出并赋值给props
*/
if(config!=null){
```

```
    //1.处理props
    if(hasValidRef(config)){
        ref=config.ref
    }
    if(hasValidKey(config)){
        key = '' + config.key;
    }
    self = config.__self===undefined?null:config.__self;
    source = config.__source===undefined?null:config.__source;
    for (propName in config) {

if(hasOwnProperty.call(config,propName)&&!RESERVED_PROPS.hasOwnProperty(propName
)){
            props[propName]=config[propName]
        }
    }
}
```

(2)获取子元素

```
/*
1.获取子元素的个数 —— 第二个参数后面的所有参数
2.若只有一个子元素，赋值给props.children
3.若有多个子元素，将子元素填充为一个数组赋值给props.children
*/
var childrenLength = arguments.length - 2
if(childrenLength===1){
    props.children =children
}else if(childrenLength>1){
    var childArr = Array(childrenLength)
    for (var i = 0; i < childrenLength; i++) {
            childArr[i] =arguments[i+2]
    }
    props.children = childArr
}
```

(3)处理默认props

将组件的静态属性defaultProps定义的默认props进行赋值

```
if (type&&type.defaultProps) {
    var defaultProps = type.defaultProps
    for (propName in defaultProps) {
        if(props[propName] === undefined){
            props[propName] = defaultProps[propName]
        }
    }
}
```

ReactElement: ReactElement将传入的几个属性进行组合，并返回。

```
type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）
key：组件的唯一标识，用于Diff算法，下面会详细介绍
ref：用于访问原生dom节点
props：传入组件的props
owner：当前正在构建的Component所属的Component
```

ReactElement.isValidElement函数用来判断一个React组件是否是有效的，下面是它的具体实现。

```
ReactElement.isValidElement = function (object) {
  return typeof object === 'object' && object !== null && object.?typeof ===
REACT_ELEMENT_TYPE;
};
/*
可见React渲染时会把没有?typeof标识，以及规则校验不通过的组件过滤掉。

当你的环境不支持Symbol时，?typeof被赋值为0xeac7，至于为什么，React开发者给出了答案：
0xeac7看起来有点像React。

*/
```

self、source只有在非生产环境才会被加入对象中。

```
self指定当前位于哪个组件实例。
_source指定调试代码来自的文件(fileName)和代码行数(lineNumber)。
```

虚拟DOM转换为真实DOM

整个流程可分为4部分:
过程1：初始参数处理
过程2：批处理、事务调用
过程3：生成html
过程4：渲染html

1.初始参数处理

在编写好我们的React组件后，我们需要调用ReactDOM.render(element, container[, callback])将组件进行渲染。
render函数内部实际调用了_renderSubtreeIntoContainer，我们来看看它的具体实现：

```
  render: function (nextElement, container, callback) {
    return ReactMount._renderSubtreeIntoContainer(null, nextElement, container,
callback);
  },
```

1.将当前组件使用TopLevelWrapper进行包裹
　TopLevelWrapper只一个空壳，它为你需要挂载的组件提供了一个rootID属性，并在render函数中返回该组件。
　TopLevelWrapper只一个空壳，它为你需要挂载的组件提供了一个rootID属性，并在render函数中返回该组件。
2.判断根结点下是否已经渲染过元素，如果已经渲染过，判断执行更新或者卸载操作
3.处理shouldReuseMarkup变量，该变量表示是否需要重新标记元素
4.调用将上面处理好的参数传入*renderNewRootComponent，渲染完成后调用callback*
　*在*renderNewRootComponent中调用instantiateReactComponent对我们传入的组件进行分类包装:
根据组件的类型，React根据原组件创建了下面四大类组件，对组件进行分类渲染:
　ReactDOMEmptyComponent:空组件
　ReactDOMTextComponent:文本
　ReactDOMComponent:原生DOM
　ReactCompositeComponent:自定义React组件

他们都具备以下三个方法：
construct:用来接收ReactElement进行初始化。
mountComponent:用来生成ReactElement对应的真实DOM或DOMLazyTree。
unmountComponent:卸载DOM节点，解绑事件。
具体是如何渲染我们在过程3中进行分析。

2.批处理、事务调用

在_renderNewRootComponent中使用ReactUpdates.batchedUpdates调用batchedMountComponentIntoNode进行批处理。

```
ReactUpdates.batchedUpdates(batchedMountComponentIntoNode, componentInstance, container, shouldReuseMarkup, context);
```

在batchedMountComponentIntoNode中，使用transaction.perform调用mountComponentIntoNode让其基于事务机制进行调用。

```
transaction.perform(mountComponentIntoNode, null, componentInstance, container, transaction, shouldReuseMarkup, context);
```

关于批处理事务，在我前面的分析setState执行机制中有更多介绍。https://juejin.im/post/6844903781813993486

3.生成html:在mountComponentIntoNode函数中调用ReactReconciler.mountComponent生成原生DOM节点。

mountComponent内部实际上是调用了过程1生成的四种对象的mountComponent方法。首先来看一下ReactDOMComponent:

```
1.对特殊DOM标签、props进行处理。
2.根据标签类型创建DOM节点。
3.调用_updateDOMProperties将props插入到DOM节点，_updateDOMProperties也可用于props Diff，第一个参数为上次渲染的props，第二个参数为当前props，若第一个参数为空，则为首次创建。
4.生成一个DOMLazyTree对象并调用_createInitialChildren将孩子节点渲染到上面。
    那么为什么不直接生成一个DOM节点而是要创建一个DOMLazyTree呢？我们先来看看_createInitialChildren做了什么：
    判断当前节点的dangerouslySetInnerHTML属性、孩子节点是否为文本和其他节点分别调用DOMLazyTree的queueHTML、queueText、queueChild。

可以发现：DOMLazyTree实际上是一个包裹对象，node属性中存储了真实的DOM节点，children、html、text分别存储孩子、html节点和文本节点。
它提供了几个方法用于插入孩子、html以及文本节点，这些插入都是有条件限制的，当enableLazy属性为true时，这些孩子、html以及文本节点会被插入到DOMLazyTree对象中，当其为false时会插入到真实DOM节点中。
```

```
var enableLazy = typeof document !== 'undefined' &&
  typeof document.documentMode === 'number' ||
  typeof navigator !== 'undefined' &&
  typeof navigator.userAgent === 'string' &&
  /\bEdge\/\d/.test(navigator.userAgent);
```

可见：enableLazy是一个变量，当前浏览器是IE或Edge时为true。

在IE（8-11）和Edge浏览器中，一个一个插入无子孙的节点，效率要远高于插入一整个序列化完整的节点树。

所以lazyTree主要解决的是在IE（8-11）和Edge浏览器中插入节点的效率问题，在后面的过程4我们会分析到：若当前是IE或Edge，则需要递归插入DOMLazyTree中缓存的子节点，其他浏览器只需要插入一次当前节点，因为他们的孩子已经被渲染好了，而不用担心效率问题。

下面来看一下ReactCompositeComponent，由于代码非常多这里就不再贴这个模块的代码，其内部主要做了以下几步：

```
+ 处理props、contex等变量，调用构造函数创建组件实例
+ 判断是否为无状态组件，处理state
+ 调用performInitialMount生命周期，处理子节点，获取markup。
+ 调用componentDidMount生命周期
```

总结：

在performInitialMount函数中，首先调用了componentWillMount生命周期，由于自定义的React组件并不是一个真实的DOM，所以在函数中又调用了孩子节点的mountComponent。这也是一个递归的过程，当所有孩子节点渲染完成后，返回markup并调用componentDidMount。

4.渲染html

```
在mountComponentIntoNode函数中调用将上一步生成的markup插入container容器。
在首次渲染时，_mountImageIntoNode会清空container的子节点后调用
DOMLazyTree.insertTreeBefore：
```

```
var insertTreeBefore = function(parentNode,tree,referenceNode){
    //判断是否为fragment节点或者<object>插件：

if(tree.node.nodeType===DOCUMENT_FRAGMENT_NODE_TYPE||tree.node.nodeType===ELEMEN
T_NODE_TYPE&&tree.node.nodeName.toLowerCase()==='object'
        &&
(tree.node.namespaceURI==null||tree.node.namespaceURI===DOMNamespaces.html)){
        insertTreeChildren(tree)
        parentNode.insertBefore(tree.node,referenceNode)
    }else{
        parentNode.insertBefore(tree.node,referenceNode)
    }
}
/*
判断是否为fragment节点或者<object>插件：
+ 如果是以上两种，首先调用insertTreeChildren将此节点的孩子节点渲染到当前节点上，再将渲染完的
节点插入到html

+ 如果是其他节点，先将节点插入到插入到html，再调用insertTreeChildren将孩子节点插入到html。

+ 若当前不是IE或Edge，则不需要再递归插入子节点，只需要插入一次当前节点。
*/
function insertTreeChildren(tree){
    if(!enableLazy){
        //不是ie/bEdge
        return
    }
    var node = tree.node
    var children = tree.children
```

```
    if(children.length){
        //递归渲染子节点
        for(var i= 0;i<children.length;i++){
            insertTreeChildren(node,children[i],null)
        }
    }else if(tree.html!=null){
        //渲染html节点
        setInnerHTML(node,tree.html)
    }else if(tree.text!=null){
        //渲染文本节点
        setTextContext(node,tree.text)
    }
}
/*
+ 判断不是IE或bEdge时return
+ 若children不为空，递归insertTreeBefore进行插入
+ 渲染html节点
+ 渲染文本节点
*/
```

原生DOM事件代理

有关虚拟DOM的事件机制，我曾专门写过一篇文章，有兴趣可以👇【React深入】React事件机制: https://juejin.im/post/6844903790198571021

虚拟DOM原理、特性总结

React组件的渲染流程

```
+ 使用React.createElement或JSX编写React组件，实际上所有的JSX代码最后都会转换成React.createElement(...)，Babel帮助我们完成了这个转换的过程。

+ createElement函数对key和ref等特殊的props进行处理，并获取defaultProps对默认props进行赋值，并且对传入的孩子节点进行处理，最终构造成一个ReactElement对象（所谓的虚拟DOM）

+ ReactDOM.render将生成好的虚拟DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实DOM。
```

虚拟DOM的组成 即ReactElementelement对象，我们的组件最终会被渲染成下面的结构：

```
+ type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）
+ key：组件的唯一标识，用于Diff算法，下面会详细介绍
+ ref：用于访问原生dom节点
+ props：传入组件的props，chidren是props中的一个属性，它存储了当前组件的孩子节点，可以是数组（多个孩子节点）或对象（只有一个孩子节点）
+ owner：当前正在构建的Component所属的Component
+ self：（非生产环境）指定当前位于哪个组件实例
+ _source：（非生产环境）指定调试代码来自的文件(fileName)和代码行数(lineNumber)
```

防止XSS

ReactElement对象还有一个?typeof属性，它是一个Symbol类型的变量 Symbol.for('react.element')，当环境不支持Symbol时，?typeof被赋值为0xeac7。

这个变量可以防止XSS。如果你的服务器有一个漏洞，允许用户存储任意JSON对象， 而客户端代码需要一个字符串，这可能为你的应用程序带来风险。JSON中不能存储Symbol类型的变量，而React渲染时会把没有?typeof标识的组件过滤掉

针对性的性能优化

在IE（8-11）和Edge浏览器中，一个一个插入无子孙的节点，效率要远高于插入一整个序列化完整的节点树。

React通过lazyTree，在IE（8-11）和Edge中进行单个节点依次渲染节点，而在其他浏览器中则首先将整个大的DOM结构构建好，然后再整体插入容器。

并且，在单独渲染节点时，React还考虑了fragment等特殊节点，这些节点则不会一个一个插入渲染。