什么是React：

React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决定自己写一套，用来架设 [Instagram](https://www.instagram.com/" \t "https://segmentfault.com/a/_blank) 的网站。做出来以后，发现这套东西很好用，就在2013年5月开源了。

1.声明式设计：React采用声明范式，可以轻松描述应用。

2.高效：React通过对DOM的模拟，最大限度地减少与DOM的交互。

3.灵活：React可以与已知的库或框架很好地配合。

什么是ES6：

ECMAScript 6（以下简称ES6）是JavaScript语言的下一代标准，已经在2015年6月正式发布了。它的目标，是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

标准的制定者有计划，以后每年发布一次标准，使用年份作为标准的版本。因为当前版本的ES6是在2015年发布的，所以又称ECMAScript 2015。也就是说，ES6就是ES2015，下一年应该会发布小幅修订的ES2016。

总结：ES6就像是ES5的“更新版”，添加了很多新特性。比如let const

从官网下载一个html文件，这个文件就能最简单的解释React的基本用法



先看引用：

1. react.development.js react核心库，主要用于创建UI

2.react-dom.development.js 提供与DOM相关的功能，用于将创建的UI渲染到浏览器中

3.babel.min.js babel是一个转换编译器，2015年6月，ES2015（ECMScript 6 简称ES6）正式发布，ES6中有很多的新特性，但是浏览器的兼容性并不好，所以才有了babel，他能将ES6转换成浏览器中可以运行的代码，并且内置了react jsx拓展。



官网的注释中有句话需要注意：不要在项目中直接使用这个babel.min.js

原因是：babel在讲jsx转换为js时比较耗费时间，实际生产过程中最好是放到服务器中完成。

再看<script type=”text/babel”> type的属性值text/babel是因为react独有的jsx语法，跟js不兼容，凡是用到jsx的地方都得使用这个属性。



继续看script中的代码：

ReactDOM是react最基础的用法，它主要的作用就是将模板转换为HTML语言并插入制定的DOM节点。

上述代码解释：

定义一个id为root的div，然后创建一个 Hello,world 为内容的h1标签，并将这个标签插入到id为root的div中。

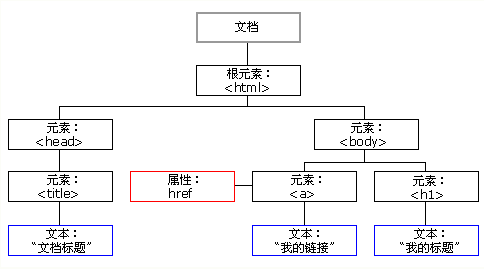
什么是虚拟DOM：

VirtualDOM是一个能够直接描述一段ＨＴＭＬ　ＤＯＭ结构的Ｊａｖａｓｃｒｉｐｔ对象，浏览器可以根据其结构按照一定规则创建出确定唯一的ＨＴＭＬ　ＤＯＭ结构。



为什么要有虚拟DOM：

先看一下HTML DOM节点树



浏览器每次在渲染HTML上的标签时，跟Android布局有很多类似之处，为了让标签最终呈现出我们希望的样子，需要经过大量的布局计算，在这样的一个过程中，性能消耗比较大。这也就是为什么HTMl以及android布局时都建议，尽量避免过多的层级嵌套。

而且一个什么内容都不包含的div如果按照标准DOM的方式创建，会自动继承大量的属性和方法。而虚拟dom则会要少很多。

（比如我们如果在android系统中很多的系统类，虽然我们用的时候感觉不到它有多少的继承，但如果一直追溯下去就能发现它可能继承自其他的类，而且其他类的方法跟属性也能继承下来，但如果我们自己写一个类，是不是我们可以想继承就继承，不继承只要你实现对应的功能就行了？这个我们自己创造的类是不是类似这种虚拟DOM，简化了一些没有必要的继承等信息）

什么时候创建虚拟DOM

浏览器在加载页面时，React根据内容先生成一个虚拟DOM（老的）

Js在对内容进行操作时，又会生成另外一个虚拟DOM（新的）

然后拿两个dom进行比较，一旦发现新的dom改了哪里，就对应去改老的dom，最后应用到实际的DOM中。

另外举个例子：

**var** A=document.getElementById('test');

A.style.backgroundColor = "black";

A.style.backgroundColor = "red";

A.style.backgroundColor = "black";

如上所示，在标准dom机制下，会对A节点进行三次的dom操作。  
而在react应用的事件函数中进行如上操作时，同样会在虚拟dom上进行三次dom的操作，但在真实dom中，它只会执行一次dom操作，即A.style.backgroundColor = "black";因为在react虚拟dom机制中，它会把所有的操作都会合并，只会对比刚开始的状态和最后操作的状态，两者中找出不同再同步到真实dom中，这就大大减少了真实dom的操作，而众所周知，dom操作是很耗性能的，这是react能做到极速渲染的原因之一。

当我们在使用React的时，我们可以这么理解，render()就是创建了一个React元素树。当state或者是props变更了之后，render()又会返回一个不同的React元素树。为了更加有效的让React更新UI，有一些通用的算法解决方案，即用最小操作数生成一棵树然后转换为另一棵树。

然而，最先进算法的复杂度为O(N^3)，其中n是树中的元素数。如果我们在React中使用这一点，那么显示1000个元素将需要10亿个比较的顺序。这个太耗费性能。相反，React实现了基于两个假设的启发式O(N)算法：两个不同类型的元素将产生不同的树。开发人员可以提示哪些子元素可以在不同的呈现过程中通过关键支柱保持稳定。在实践中，这些假设对于几乎所有实际用例都是有效的。

Diff算法简介：

深度优先

Diffing算法在区分两棵树的时候，首先对根元素进行比较。而比较后具体怎么操作则根据根元素的类型来定。

不同类型的元素时：

不管什么时候，只要根元素类型不同，React直接将老的元素删除，再创建一个新的。不管你的这个元素时从<a>标签到<img>还是<Article>到<Comment>，只要不同就会重建。

当删除掉一棵树时，旧的DOM节点就被会移除，组件实例就会调用componentWillUnmount()，当创建新树时，新的DOM节点则会被插入DOM中，组件实例则会先调用componentWillMount()然后是componentDidMount()。任何跟老树相关的状态都会丢失。

任何属于该根节点下方的组件也都会被卸载并销毁其状态。

比如：

<div>

<Counter /></div>

<span>

<Counter /></span>

Div变换成了span，那么里面对应的Counter也会被重新创建一个。

当比较两个相同类型的React DOM元素时，React只会去查找两个元素的属性，如果属性不同则之变更属性。

比如:

<div className="before" title="stuff" />

<div className="after" title="stuff" />

当要修改style时，React同样也知道仅仅修改他的属性值

比如：下图就只讲color从red改为green

<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />

处理完节点之后，继续选取子节点对比。

当组件，元素都是同一类型时，React会保持实例不变。但为了匹配新元素，React会更新基础实例组件，并调用componentWillReceiveProps() 和componentWillUpdate()方法。

循环比较子节点时：

默认情况下，React会同时对两个子列表进行迭代，当发现不一样时则触发“变化”。

<ul>

<li>first</li>

<li>second</li>

</ul>

<ul>

<li>first</li>

<li>second</li>

<li>third</li>

</ul>

比如在比较上面两个ul里的子节点时，React会直接先比较first跟second，然后比较到第三个时发现了不一样，则会插入third。

但如果出现下面这种情况就会出现“问题”了：

<ul>

<li>first</li>

<li>second</li>

</ul>

<ul>

<li>third</li>

<li>first</li>

<li>second</li>

</ul>

这种情况下，React在比较时会发现，第一个li跟第二li是不一样的，这也就会导致后面的对比都会出现问题。还没智能到知道第一个根元素下的两个子节点跟第二个根元素下的子节点的后面两个一致。

为了解决这一问题，React还支持一个关键属性，当子节点有键时，可以使用该键去进行对比，这样就能提高转换效率。

<ul>

<li key=”10”>first</li>

<li key=”20”>second</li>

</ul>

<ul>

<li key=”30”>third</li>

<li key=”10”>first</li>

<li key=”20”>second</li>

</ul>