31 React基本功（二）——深入组件通信机制

更新时间：2020-05-26 14:11:12

> 散步促进我的思想。我的身体必须不断运动，脑筋才会开动起来。—— 卢梭

大家知道，**组件化、数据驱动、单向数据流是 React 的重要特性**。

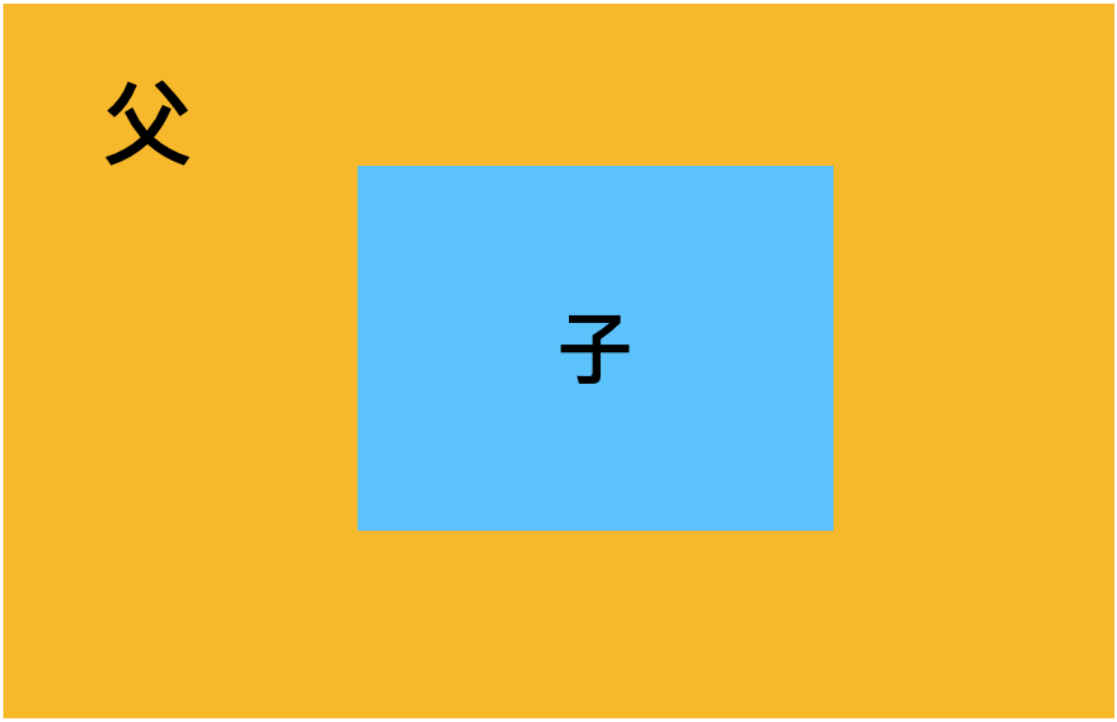
涉及 React 基础层次的面试，“通信机制”这个话题，会是一个绝佳的切入点。通过考察候选人对通信机制的理解，可以帮助面试官确认非常多的信息：比如你对组件关系的理解是否全面，对单向数据流的原则认知是否到位，比如你对 Context API 是否了解、对 Event Bus 是否了解等等等等。

简单来说——这是一道绝对的高频考题！

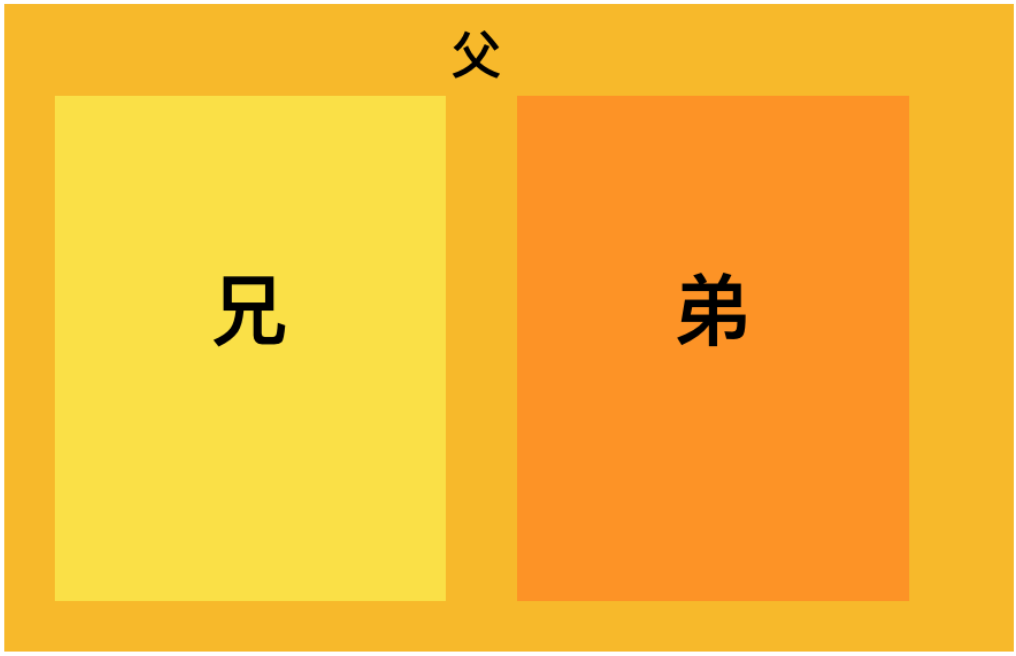
图解 React 组件间关系

在 React 世界里，组件间的关系基本可以由这三种来概括：

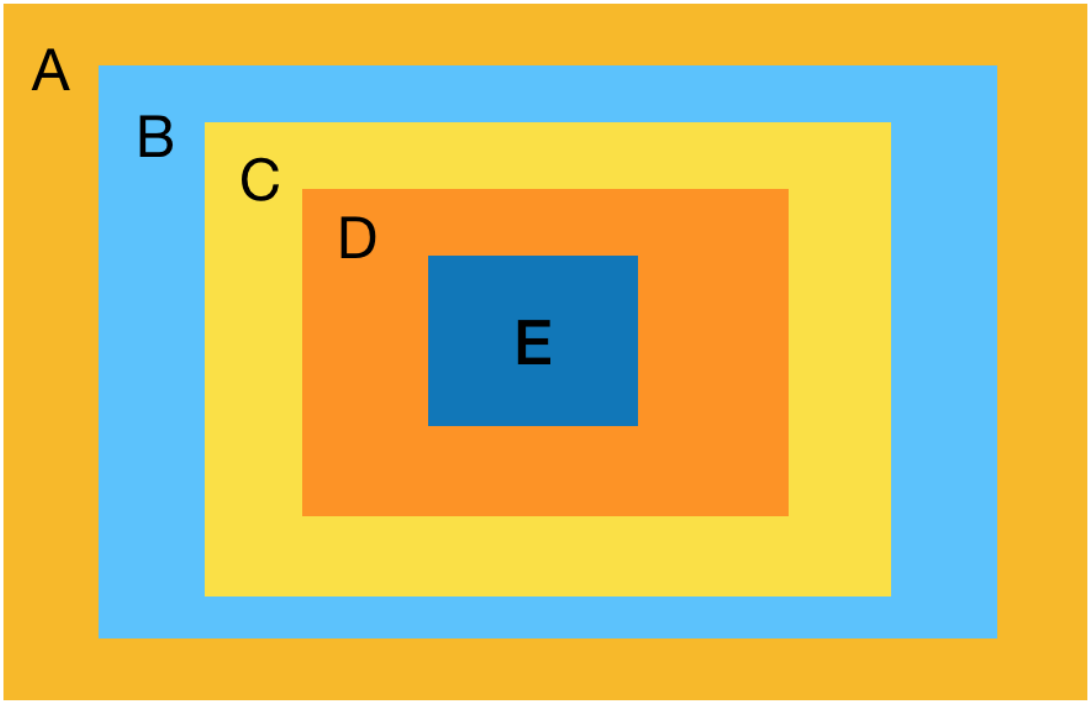
父子关系（嵌套）：



兄弟关系（平行）：



深层嵌套关系（子子孙孙无穷尽也）



父子组件通信

这是最常见、也最好解决的一个通信场景：react的数据流是单向的，最常见的操作就是通过props由父组件向子组件传递数据。

举个例子：

```
...(省略基础依赖的 import)
import Child from '子组件的引用路径'

class Father extends React.Component {
  constructor() {
    super()
    this.state = {
      // 父组件具有“组件标题颜色”信息，需要带给子组件
      titleColor: '#fff'
    }
  }

  changeColor = () => {
      // 点击后，父子组件标题同时变更颜色
      this.setState({
        titleColor: '#000'
      })
  }


  // 渲染
  render() {
    return (
      <div>
        <h1 style={{color: this.state.titleColor}}>我是父标题</h1>
        <button onClick={this.changeColor}>点我变色</button>
        // 子组件中以 props 的形式传入和父组件同步的 titleColor
        <Child titleColor={this.state.titleColor} />
      </div>
    );
  }
}
```

子组件中，可以通过 props 来拿到父组件的数据：

```
class Child extends React.Component {
  render() {
    const { titleColor } = this.props
    // 此处省略渲染细节
    return ...
  }
}
```

子组件向父组件通信

由于 React 数据流是单向的，子组件没法直接向父组件抛出数据。不过，可以通过父组件向子组件传递一个函数形式的 props 来达到我们的目的：父组件将作用域为自身的函数传递给子组件，子组件在调用该函数时，把目标数据以函数入参的形式交到父组件手中即可。

举个例子：

父组件中将方法传入子组件：

```
...(省略基础依赖的 import)
import Child from '子组件的引用路径'

class Father extends React.Component{
  state = {
    titleColor: '#fff'
  };

  // 父组件中存在一个允许自定义标题颜色的方法
  changeColor = (color) => {
    this.setState({
      titleColor
    });
  }


  render() {
    return (
      <div>
        <h1 style={{color: this.state.titleColor}}>我是父标题</h1>
        <button onClick={this.changeColor}>点我变色</button>
        // 子组件中以 props 的形式收到父组件中的 changeColor 方法
        <Child changeColor={color => this.state.changeColor(color)} />
      </div>
    );
  }
}
```

子组件在调用该函数时，把目标数据以函数入参的形式交到父组件手中：

```
class Child extends React.Component{
  componentDidMount() {
    setTimeout(() => {
      // 调用父组件的方法
      this.props.changeColor('#000')
    }, 1000)
  }

  render() {
    // 此处省略渲染细节
    return ...
  }
}
```

兄弟组件通信

兄弟组件乍一看没啥关联，但别忘了，它们有同一个爹。

结合咱们上面的分析，爹可以把话带给儿子，儿子也可以把话带给爹。那么儿子A说给爹的话，爹也可以带给儿子B。如此一来，兄弟组件通信就不是啥问题了。

我们梳理一下这个过程的实现流程：在父组件中定义一个作用域是它自己的方法（这个事件用来修改某个特定的 state），以 props 的形式传递给儿子 A，同时把这个事件里涉及的 state 值，以 props 的形式传递给儿子 B。这样当我们在 A 中调用父组件的方法、修改这个目标 state 值后，会触发父组件的 state 更新，进而自然就会触发儿子 B 的 props 传值更新。

我们用一个例子来理解一下这个过程：

首先是父组件：

```
...(省略基础依赖的 import)
import ChildA from '子组件A的引用路径'
import ChildB from '子组件B的引用路径'

class Father extends React.Component{
  state = {
    titleColor: '#fff'
  };

  // 父组件中存在一个允许自定义标题颜色的方法
  changeColor = (color) => {
    this.setState({
      titleColor
    });
  }


  render() {
    return (
      <div>
        <h1 style={{color: this.state.titleColor}}>我是父标题</h1>
        <button onClick={this.changeColor}>点我变色</button>
        // 子组件A中以 props 的形式收到父组件中的 changeColor 方法
        <ChildA changeColor={color => this.state.changeColor(color)} />
        // 子组件B中以 props 的形式传入和父组件同步的 titleColor
        <ChildB titleColor={this.state.titleColor} />
      </div>
    );
  }
}
```

子组件 A 中，触发父组件 state 的改变：

```
class ChildA extends React.Component{
  componentDidMount() {
    setTimeout(() => {
      // 调用父组件的方法
      this.props.changeColor('#000')
    }, 1000)
  }

  render() {
    // 此处省略渲染细节
    return ...
  }
}
```

子组件 B 中，对应的 props 就会同步为父组件新的 state 值：

```
class ChildB extends React.Component {
  render() {
    // 这个值也更新了
    const { titleColor } = this.props
    // 此处省略渲染细节
    return ...
  }
}
```

深层嵌套的组件间通信

在过去，涉及深层嵌套、或者说任意两个组件之间这样跨度较大的通信，我们一般是直接通过全局事件总线（Event Bus）或者引入 Redux 来解决。

不过随着 React16.3 版本的发布，在深层嵌套这个场景下，有了一个新的答案：**使用 Context API**。

大家注意，Context API 可不是随着 React16.3 的发布而出现的。事实上，React在很早就支持了context，只是官方并不推荐我们使用它：

> The vast majority of applications do not need to use context.
>
> If you want your application to be stable, don't use context. It is an experimental API and it is likely to break in future releases of React.
>
> (绝大多数应用程序不需要使用 context。如果你想让你的应用更稳定，不要使用context。因为这是一个实验性的API，在未来的React版本中可能会被更改。)

好消息是，从 React16.3 开始，Context API 得到了升级，不再作为不稳定的实验性能力存在。现在，我们可以放心地使用 Context API 了。

理解 **Context API**

Context API 是干嘛的？

> Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。

一般来说，在 React 应用中，数据只能通过 props 属性自上而下传递。

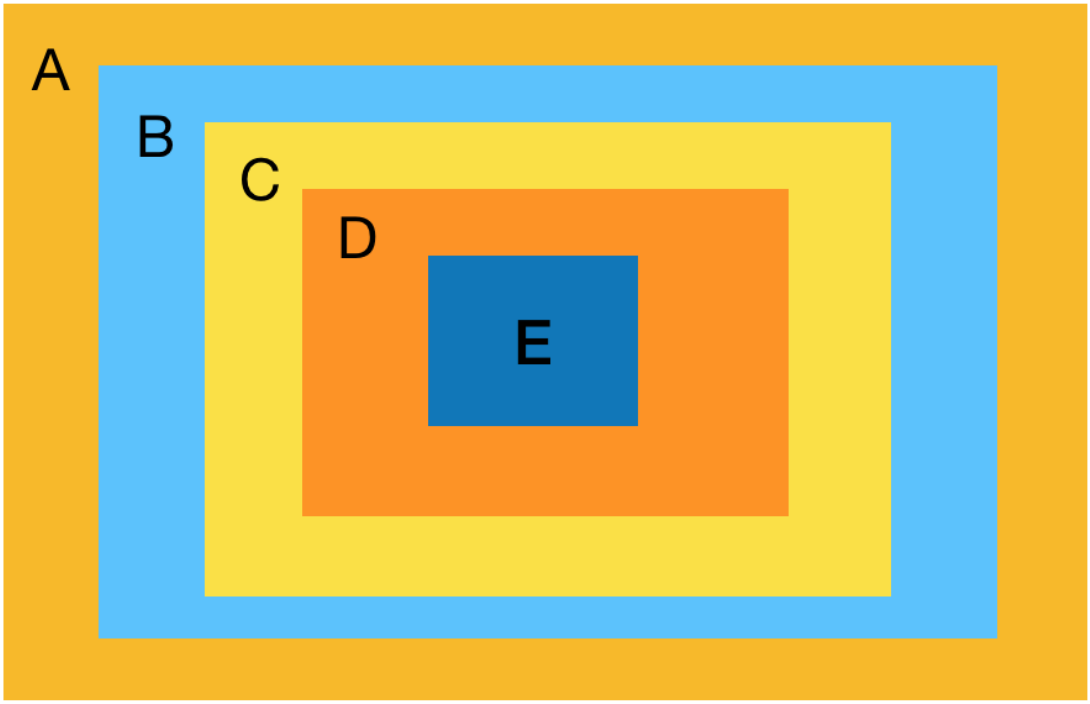
但当我们想要跨 N 个层级传递某个数据时，逐层传递就会变得非常繁琐，而且还会带来不必要的数据更新（比如说一些全局性质的数据，用户名、用户权限等）。

Context 面向这类场景，提供了一种在组件之间共享此类值的方式，它允许我们不必显式地通过组件树的逐层传递 props。

Context 的用法

我们直接通过一个实例来理解它的用法：

现在我们立足 A 组件，想要把 A 的标题颜色信息传递给千里之外的 E 组件，它们的嵌套关系正如下图一样，深不可测：




```
class A extends React.Component{
  state = {
    titleColor: '#fff'
  };

  render() {
    return (
      <div>
        // 省略 B 组件引入逻辑
      </div>
    );
  }
}
```

大家脑补一下，如果是使用 props，那么我需要在 A 组件中把 titleColor 传给 B，然后 B 传 C、C 传 D、D 传 E…这种操作蠢极了。

现在有了 Context API，我不必去理会那些无聊的“中间商”只需要把注意力放在数据的源头组件和数据的接收方上就可以了：

（注意源头组件（Provider）会接受一个 value 作为 props，这个值会修改你在创建 Context 时设定的默认值）

```
// 用 React.createContext 可以为当前的 titleColor 创建一个 context（“#fff”为默认值）。
// React.createContext 方法会返回一个带有 Provider 和 Consumer 的对象。
const TitleColorContext = React.createContext('#fff');

class A extends React.Component {

  render() {
    return (
      // 使用一个 Provider（提供者） 来将当前的 titleColor 传递给下面的组件树。
        <TitleColorContext.Provider value="#000">
       <div>
        // 省略 B 组件引入逻辑
       </div>
      </TitleColorContext.Provider>
    );
  }
}
```

随后我们可以在 E 组件里去访问这个 Context（注意 Consumer 表示消费者，它接受一个 render props 作为唯一的 children。render props 是一个函数，这个函数会接收到 Context 传递的数据作为参数，并且需要返回一个组件），形式上类似这样：

```
<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染*/}
</MyContext.Consumer>
```

对应到我们这个例子里，就是这样：

```
class E extends React.Component {

  render() {
    <TitleColorContext.Consumer>
      {
        (titleColor) => (
          <h1 style={{ color: titleColor }}>
              我是 E 标题
           </h1>
        )
      }
    </TitleColorContext.Consumer>
  }
}
```

大功告成！

任意组件间通信

这种场景需要我们召唤 Event-Bus。

关于 Event-Bus，用过它的同学会知道，这货简直是万金油。不夸张地说，它可以解决我们本文描述的所有场景下的通信问题。也正是因为它够强，一直是面试过程中的一个稳定热点。

不过，鉴于 Event-Bus 本身和 React 关系不大、同时更多地涉及了设计模式中的考点，我们后面会在“算法与设计模式”这一章为它单开小节来作介绍。在本章，我们仍然把篇幅留给 React。

}