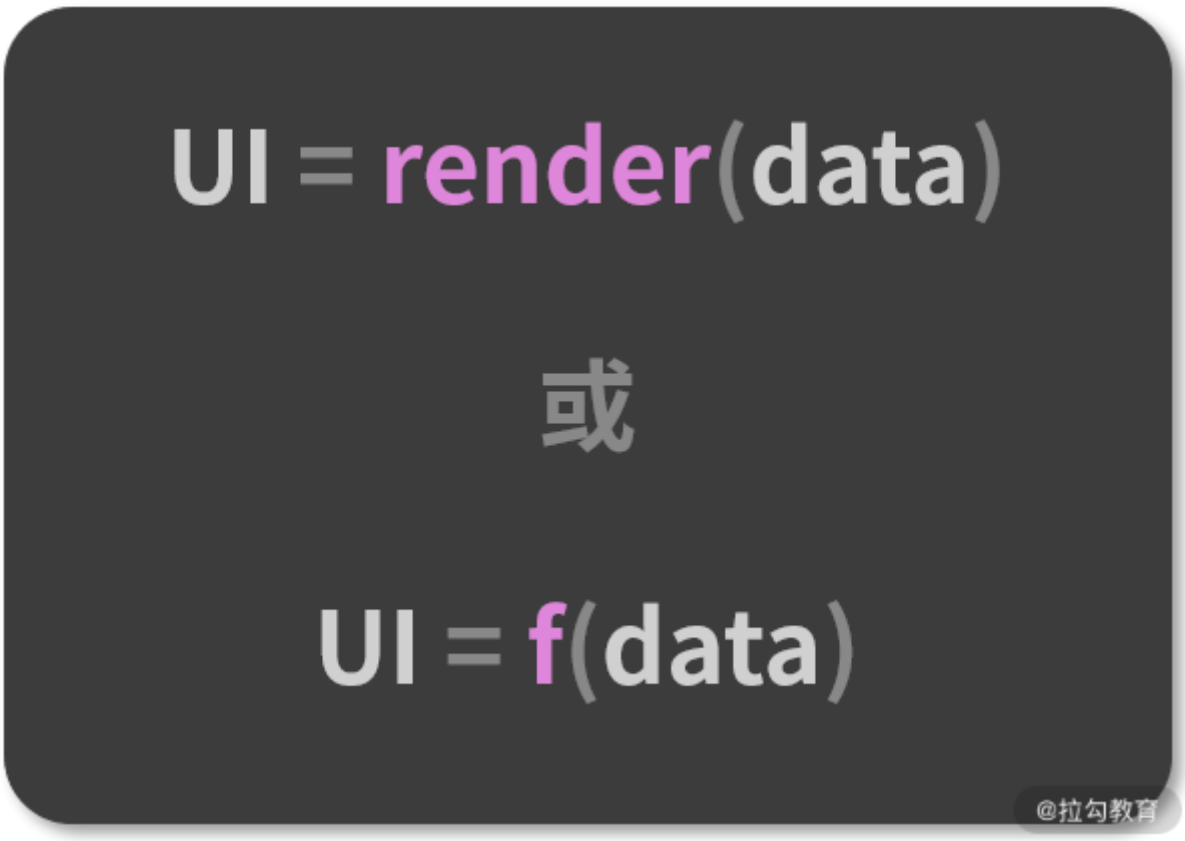
04 | 数据是如何在 React 组件之间流动的？（上）

2020/10/21 修言

通过前面 3 个课时的学习，相信你已经对 React 生命周期相关的“Why”“What”和“How”有了系统的理解和掌握。当我们谈论生命周期时，其实谈论的是组件的“内心世界”。但组件和人是一样的，它不仅需要拥有丰富的内心世界，还应该建立健全的“人际关系”，要学会**沟通和表达**。

从本课时开始，我们将一起探索蕴含在 React 组件中的“沟通与表达”的艺术。我们知道，React 的核心特征是“**数据驱动视图**”，这个特征在业内有一个非常有名的函数式来表达：



这个表达式有很多的版本，一些版本会把入参里的 data 替换成 state，但它们本质上都指向同一个含义，那就是**React 的视图会随着数据的变化而变化**。数据这个角色在 React 中的地位可见一斑。

在 React 中，如果说两个组件之间希望能够产生“耦合”（即 A 组件希望能够通过某种方式影响到 B 组件），那么毫无疑问，这两个组件必须先建立数据上的连接，以实现所谓的“组件间通信”。

“组件间通信”的背后是一套环环相扣的 React 数据流解决方案。虽然这套解决方案在业内已经有了比较成熟和稳定的结论，但**许多人仍然会因为知识的系统性和整体性不强而吃亏**。

在前面三个课时中，我们的学习思路是往纵深处去寻觅：铺陈大量的前置知识，然后一步一步地去询问生命周期背后的“Why”，最终揪出 Fiber 架构这个大 boss（不过学到这里，这个“纵深”我们才只挖到一半，专栏第二模块还有一大波 Fiber 原理等待我们继续寻觅）。

在接下来的第 04 和 05 课时中，我们要做的事情则更倾向于横向的“聚合”：我将用简单易懂的语言，帮你理解当下实践中 React 数据通信的四个大方向，并针对每个方向给出具体的场景和用例。**这些知识本身并不难，但摊子却可以铺得非常大，相关的问题在面试中也始终具备较高的区分度**。要想扎扎实实掌握，必须耐下心、沉住气，在学习过程中主动地去串联自己的知识链路。

基于 props 的单向数据流

既然 props 是组件的入参，那么组件之间通过修改对方的入参来完成数据通信就是天经地义的事情了。不过，这个“修改”也是有原则的——你必须确保所有操作都在“**单向数据流**”这个前提下。

所谓单向数据流，指的就是当前组件的 state 以 props 的形式流动时，**只能流向组件树中比自己层级更低的组件。** 比如在父-子组件这种嵌套关系中，只能由父组件传 props 给子组件，而不能反过来。

听上去虽然限制重重，但用起来却是相当的灵活。基于 props 传参这种形式，我们可以轻松实现父-子通信、子-父通信和兄弟组件通信。

父-子组件通信

原理讲解

这是最常见、也是最好解决的一个通信场景。React 的数据流是单向的，父组件可以直接将 this.props 传入子组件，实现父-子间的通信。这里我给出一个示例。

编码实现

- 子组件编码内容：

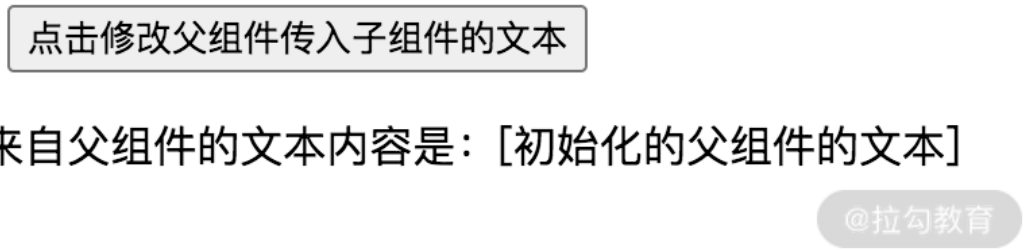
```
function Child(props) {
  return (
    <div className="child">
      <p>{`子组件所接收到的来自父组件的文本内容是：[${props.fatherText}]`}</p>
    </div>
  );
}
```

- 父组件编码内容：

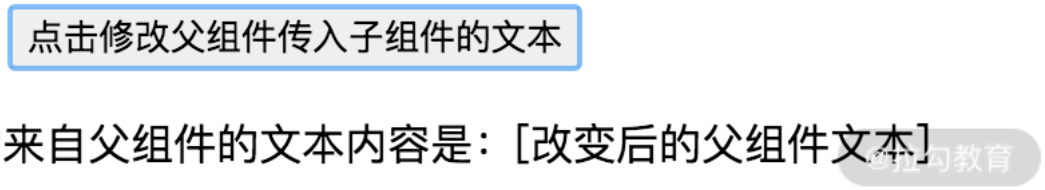
```
class Father extends React.Component {
  // 初始化父组件的 state
  state = {
    text: "初始化的父组件的文本"
  };
  // 按钮的监听函数，用于更新 text 值
  changeText = () => {
    this.setState({
      text: "改变后的父组件文本"
    });
  };
  // 渲染父组件
  render() {
    return (
      <div className="father">
        <button onClick={this.changeText}>
          点击修改父组件传入子组件的文本
        </button>
        {/* 引入子组件，并通过 props 下发具体的状态值实现父-子通信 */}
        <Child fatherText={this.state.text} />
      </div>
    );
  }
}
```

视图层验证

我们直接对父组件进行渲染，可以看到大致如下图所示的界面：



通过子组件顺利读取到父组件的 this.props.text，从这一点可以看出，父-子之间的通信是没有问题的。此时假如我们点击父组件中的按钮，父组件的 this.state.text 会发生变化，同时子组件读取到的 props.text 也会跟着发生变化（如下图所示），也就是说，父子组件的数据始终保持一致。



由此我们便充分验证了父-子组件基于 props 实现通信的可行性。

子-父组件通信

原理讲解

考虑到 props 是单向的，子组件并不能直接将自己的数据塞给父组件，但 props 的形式也可以是多样的。假如父组件传递给子组件的是一个**绑定了自身上下文的函数**，那么子组件在调用该函数时，就可以**将想要交给父组件的数据以函数入参的形式给出去**，以此来间接地实现数据从子组件到父组件的流动。

编码实现

这里我们只需对父-子通信中的示例稍做修改，就可以完成子-父组件通信的可行性验证。

首先是对子组件的修改。在 Child 中，我们需要增加对状态的维护，以及对 Father 组件传入的函数形式入参的调用。子组件编码内容如下，修改点我已在代码中以注释的形式标出：

```
class Child extends React.Component {
  // 初始化子组件的 state
  state = {
    text: '子组件的文本'
  }

  // 子组件的按钮监听函数
  changeText = () => {
    // changeText 中，调用了父组件传入的 changeFatherText 方法
    this.props.changeFatherText(this.state.text)
```

```
  }
  render() {
    return (
      <div className="child">
        {/* 注意这里把修改父组件文本的动作放在了 Child 里 */}
        <button onClick={this.changeText}>
          点击更新父组件的文本
        </button>
      </div>
    );
  }
}
```

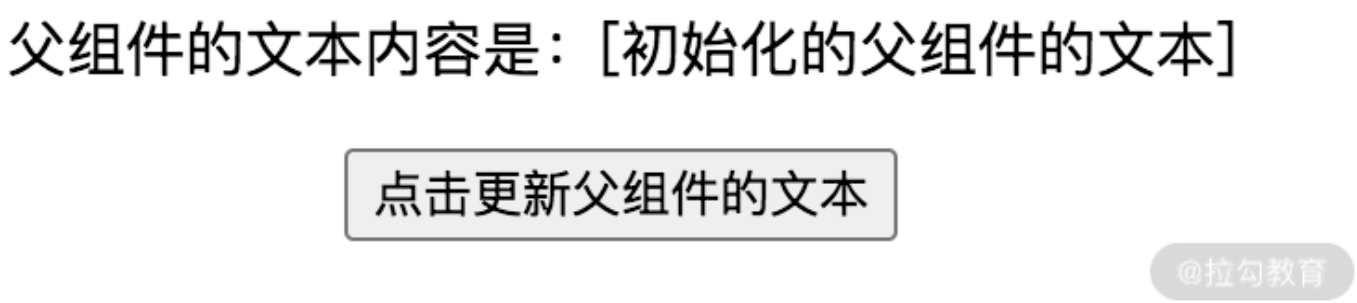
在父组件中，我们只需要在 changeText 函数上开一个传参的口子，作为数据通信的入口，然后把 changeText 放在 props 里交给子组件即可。父组件的编码内容如下：

复制代码

```
class Father extends React.Component {
  // 初始化父组件的 state
  state = {
    text: "初始化的父组件的文本"
  };
  // 这个方法会作为 props 传给子组件，用于更新父组件 text 值。newText 正是开放给子组件的数
  changeText = (newText) => {
    this.setState({
      text: newText
    });
  };
  // 渲染父组件
  render() {
    return (
      <div className="father">
        <p>{`父组件的文本内容是：[${this.state.text}]`}</p>
        {/* 引入子组件，并通过 props 中下发可传参的函数 实现子-父通信 */}
        <Child
          changeFatherText={this.changeText}
        />
      </div>
    );
  }
```

视图层验证

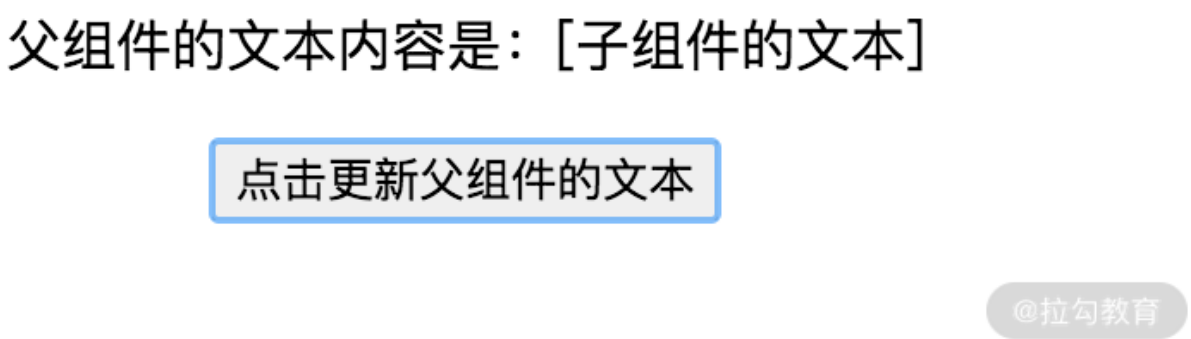
新的示例渲染后的界面大致如下图所示：



注意，在这个 case 中，我们将具有更新数据能力的按钮转移到了子组件中。

当点击子组件中的按钮时，会调用已经绑定了父组件上下文的 this.props.changeFatherText 方法，同时将子组件的 this.state.text 以函数入参的形式传入，由此便能够间接地用子组件的 state 去更新父组件的 state。

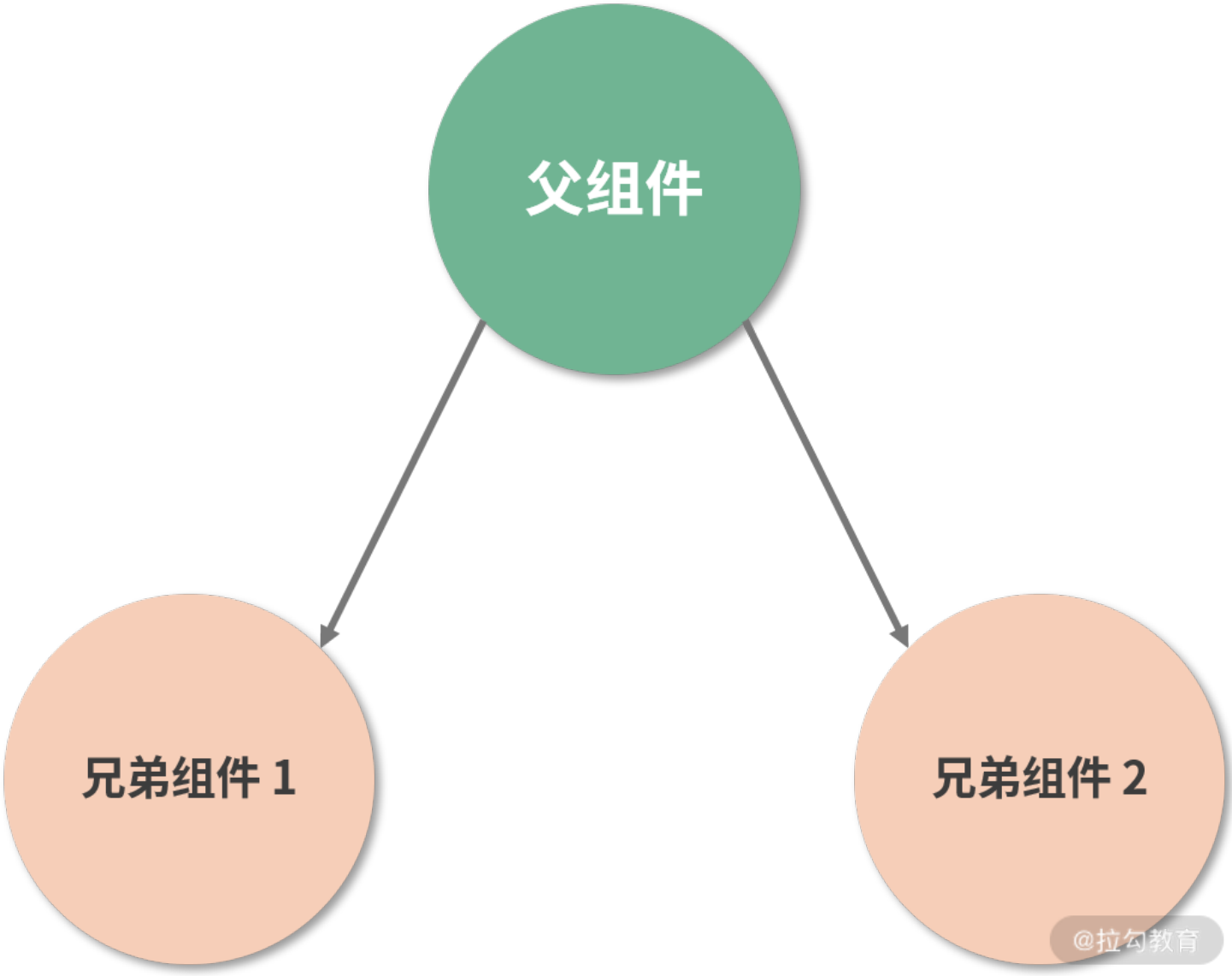
点击按钮后，父组件的文本会按照我们的预期被子组件更新掉，如下图所示：



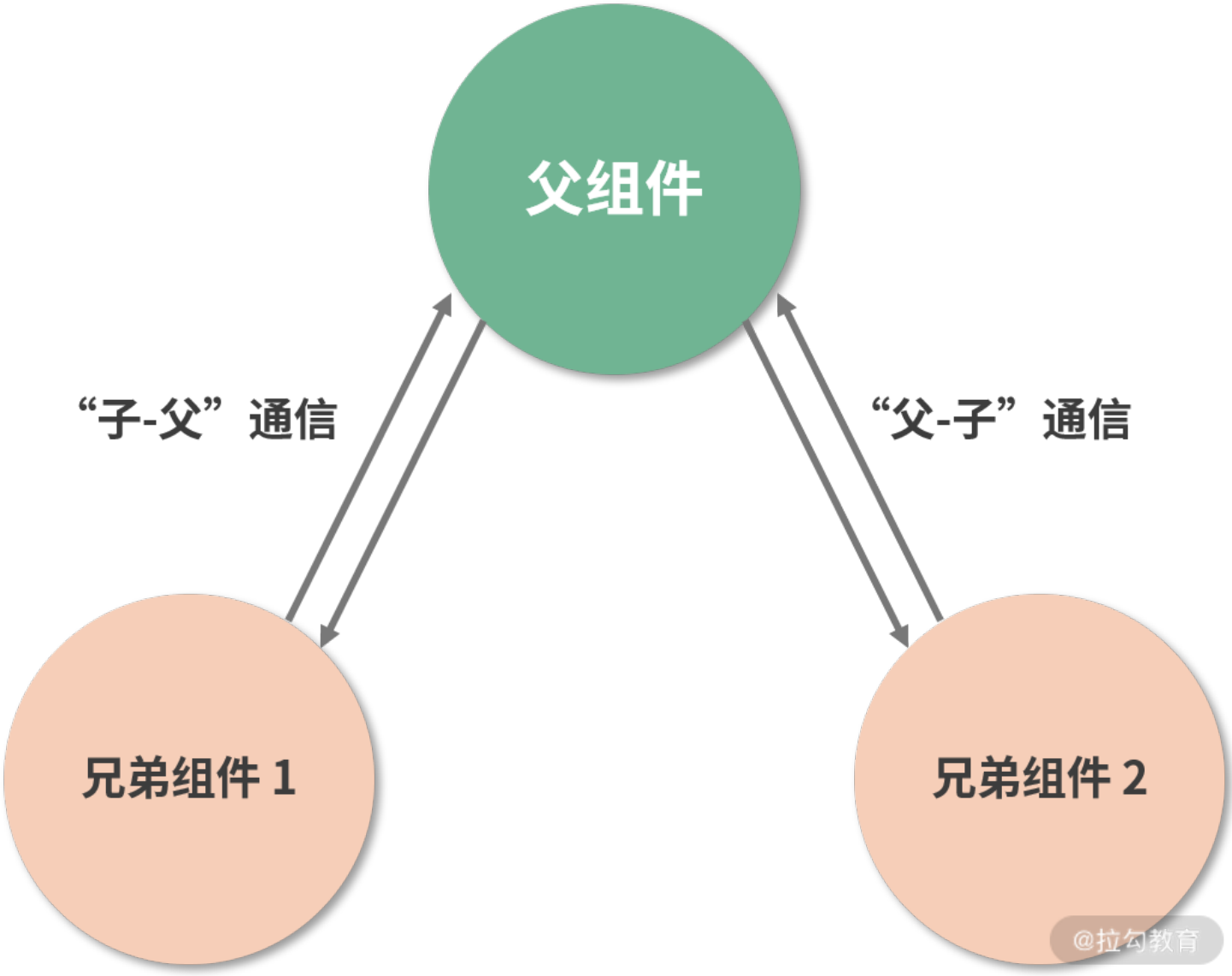
兄弟组件通信

原理讲解

兄弟组件之间共享了同一个父组件，如下图所示，这是一个非常重要的先决条件。



这个先决条件使得我们可以继续利用父子组件这一层关系，将“兄弟 1 → 兄弟 2”之间的通信，转化为“兄弟 1 → 父组件”（子-父通信）、“父组件 → 兄弟 2”（父-子）通信两个步骤，如下图所示，这样一来就能够巧妙地把“兄弟”之间的新问题化解为“父子”之间的旧问题。



编码实现

接下来我们仍然从编码的角度进行验证。首先新增一个 NewChild 组件作为与 Child 组件同层级的兄弟组件。NewChild 将作为数据的发送方，将数据发送给 Child。在 NewChild 中，我们需要处理 NewChild 和 Father 之间的关系。NewChild 组件编码如下：

复制代码

```
class NewChild extends React.Component {
  state = {
    text: "来自 newChild 的文本"
  };
  // NewChild 组件的按钮监听函数
  changeText = () => {
    // changeText 中，调用了父组件传入的 changeFatherText 方法
    this.props.changeFatherText(this.state.text);
  };
  render() {
    return (
      <div className="child">
```

```
        {/* 注意这里把修改父组件文本（同时也是 Child 组件的文本）的动作放在了 NewChild 里
        <button onClick={this.changeText}>点击更新 Child 组件的文本</button>
      </div>
    );
  }
}
```

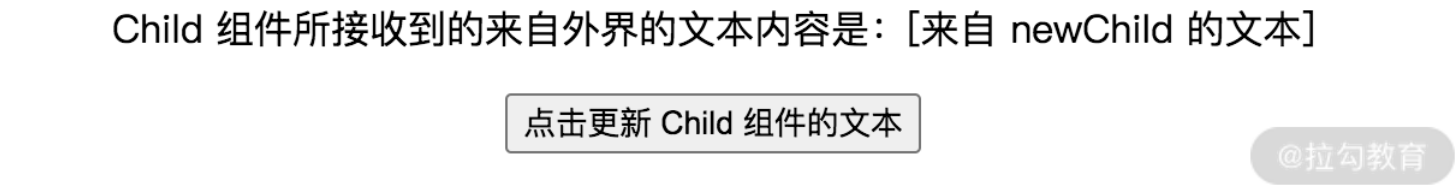
接下来看看 Father 组件。在 Father 组件中，我们通过 text 属性连接 Father 和 Child，通过 changeText 函数来连接 Father 和 NewChild。由此便把 text 属性的渲染工作交给了 Child，把 text 属性的更新工作交给 NewÇhild，以此来实现数据从 NewChild 到 Child 的流动。Father 组件编码如下：

复制代码

```
class Father extends React.Component {
  // 初始化父组件的 state
  state = {
    text: "初始化的父组件的文本"
  };
  // 传给 NewChild 组件按钮的监听函数，用于更新父组件 text 值（这个 text 值同时也是 Child
  changeText = (newText) => {
    this.setState({
      text: newText
    });
  };
  // 渲染父组件
  render() {
    return (
      <div className="father">
        {/* 引入 Child 组件，并通过 props 中下发具体的状态值 实现父-子通信 */}
        <Child fatherText={this.state.text} />
        {/* 引入 NewChild 组件，并通过 props 中下发可传参的函数 实现子-父通信 */}
        <NewChild changeFatherText={this.changeText} />
      </div>
    );
  }
}
```

视图层验证

编码完成之后，界面大致的结构如下图所示：



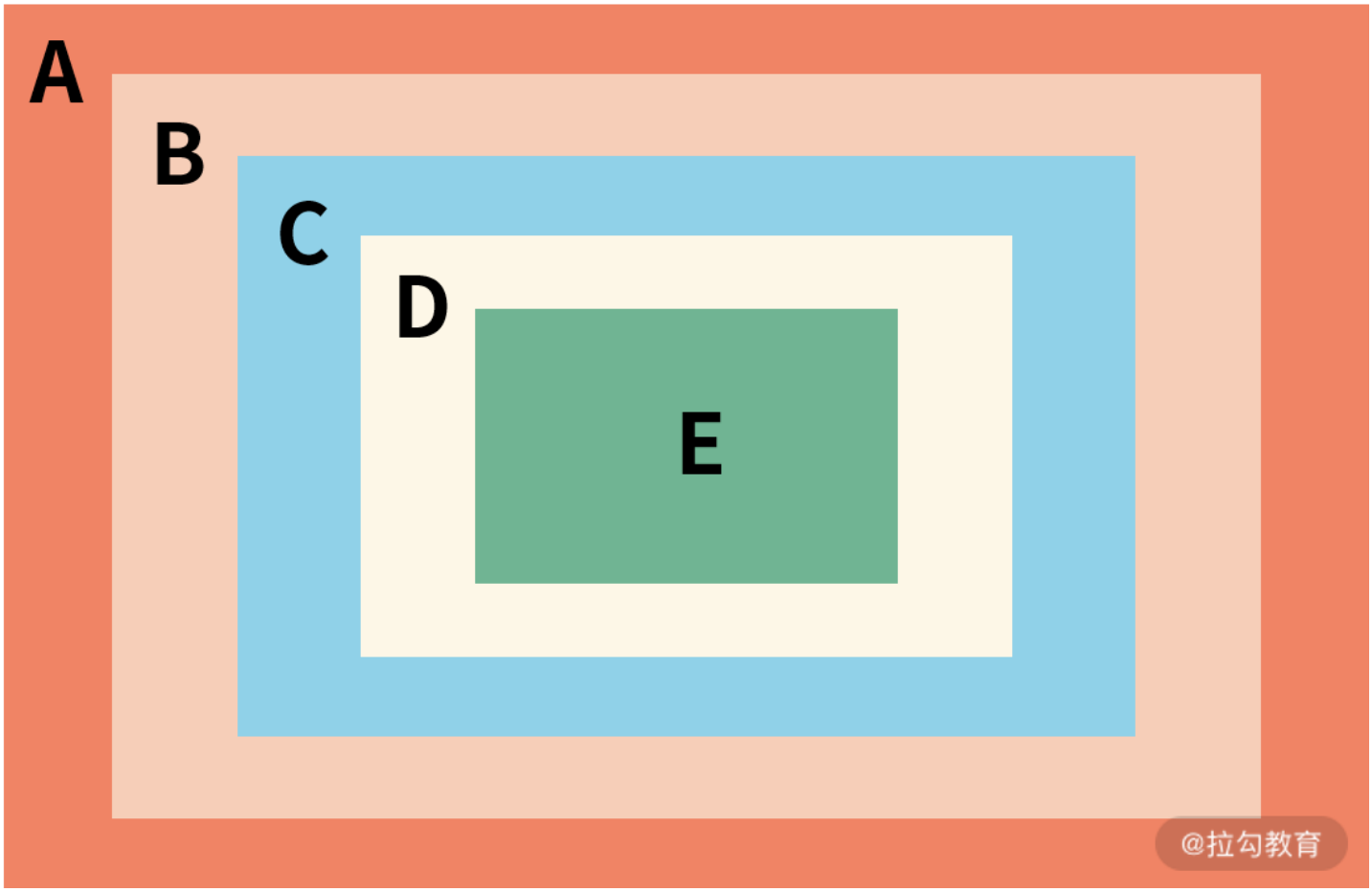
由于整体结构稍微复杂了一些，这里我把 Father、Child 和 NewChild 在图中的大致范围标一下：

- 红色所圈范围为 Father 组件，它包括了 Child 和 NewChild；
- 灰色圈住的按钮是 NewChild 组件的渲染结果，它可以触发数据的改变；
- 蓝色圈住的文本是 Child 组件的渲染结果，它负责感知和渲染数据。

现在我点击位于 NewChild 组件中的“点击更新 Child 组件的文本”按钮，就可以看到 Child 会跟着发生变化，如下图所示，进而验证方案的可行性。

为什么不推荐用 props 解决其他场景的需求

至此，我们给出了 props 传参这种形式比较适合处理的三种场景。尽管这并不意味着其他场景不能用 props 处理，但如果你试图用简单的 props 传递完成更加复杂的通信需求，往往会得不偿失。这里我给你举一个比较极端的例子：



如上图所示，可以看到这是一个典型的多层嵌套组件结构。A 组件倘若想要和层层相隔的 E 组件实现通信，就必须把 props 经过 B、C、D 一层一层地传递下去。在这个过程中，反反复复的 props 传递不仅会带来庞大的工作量和代码量，还会污染中间无辜的 B、C、D 组件的属性结构。

层层传递的优点是非常简单，用已有知识就能解决，但问题是会浪费很多代码，非常烦琐，中间作为桥梁的组件会引入很多不属于自己的属性。短期来看，写代码的人会很痛苦；长期来看，整个项目的维护成本都会变得非常高昂。因此，**层层传递 props 要不得**。

那有没有更加灵活的解决方案，能够帮我们处理“任意组件”之间的通信需求呢？答案是不仅有，而且姿势还很多。我先从最朴素的“发布-订阅”模式讲起。

利用“发布-订阅”模式驱动数据流

“发布-订阅”模式可谓是解决通信类问题的“万金油”，在前端世界的应用非常广泛，比如：

- 前两年爆火的 socket.io 模块，它就是一个典型的跨端发布-订阅模式的实现；
- 在 Node.js 中，许多原生模块也是以 EventEmitter 为基类实现的；
- 不过大家最为熟知的，应该还是 Vue.js 中作为常规操作被推而广之的“全局事件总线” EventBus。

这些应用之间虽然名字各不相同，但内核是一致的，也就是我们下面要讲到的“发布-订阅”模型。

理解事件的发布-订阅机制

发布-订阅机制早期最广泛的应用，应该是在浏览器的 DOM 事件中。 相信有过原生 JavaScript 开发经验的同学，对下面这样的用法都不会陌生：

```
target.addEventListener(type, listener, useCapture);
```

复制代码

通过调用 addEventListener 方法，我们可以创建一个事件监听器，这个动作就是“订阅”。比如我可以监听 click（点击）事件：

```
el.addEventListener("click", func, false);
```

复制代码

这样一来，当 click 事件被触发时，事件会被“发布”出去，进而触发监听这个事件的 func 函数。这就是一个最简单的发布-订阅案例。

使用发布-订阅模式的优点在于，**监听事件的位置和触发事件的位置是不受限的**，就算相隔十万八千里，只要它们在同一个上下文里，就能够彼此感知。这个特性，太适合用来应对“任意组件通信”这种场景了。

发布-订阅模型 API 设计思路

通过前面的讲解，不难看出发布-订阅模式中有两个关键的动作：**事件的监听（订阅）和事件的触发（发布）**，这两个动作自然而然地对应着两个基本的 API 方法。

- on()：负责注册事件的监听器，指定事件触发时的回调函数。
- emit()：负责触发事件，可以通过传参使其在触发的时候携带数据 。

最后，只进不出总是不太合理的，我们还要考虑一个 off() 方法，必要的时候用它来删除用不到的监听器：

- off()：负责监听器的删除。

发布-订阅模型编码实现

“发布-订阅”模式不仅在应用层面十分受欢迎，它更是面试官的心头好。在涉及设计模式的面试中，如果只允许出一道题，那么我相信大多数的面试官都会和我一样，会毫不犹豫地选择考察“发布-订阅模式的实现”。 接下来我就手把手带你来做这道题，写出一个同时拥有 on、emit 和 off 的 EventEmitter。

在写代码之前，先要捋清楚思路。这里我把“实现 EventEmitter”这个大问题，拆解为 3 个具体的小问题，下面我们逐个来解决。

- **问题一：事件和监听函数的对应关系如何处理?**

提到“对应关系”，应该联想到的是“映射”。在 JavaScript 中，处理“映射”我们大部分情况下都是用对象来做的。所以说在全局我们需要设置一个对象，来存储事件和监听函数之间的关系：

复制代码

```
constructor() {
  // eventMap 用来存储事件和监听函数之间的关系
  this.eventMap= {}
}
```

- **问题二：如何实现订阅?**

所谓“订阅”，也就是注册事件监听函数的过程。这是一个“写”操作，具体来说就是把事件和对应的监听函数写入到 eventMap 里面去：

复制代码

```
// type 这里就代表事件的名称
on(type, handler) {
  // hanlder 必须是一个函数，如果不是直接报错
  if(!(handler instanceof Function)) {
    throw new Error("哥 你错了 请传一个函数")
  }
  // 判断 type 事件对应的队列是否存在
  if(!this.eventMap[type]) {
   // 若不存在，新建该队列
    this.eventMap[type] = []
  }
  // 若存在，直接往队列里推入 handler
  this.eventMap[type].push(handler)
}
```

- **问题三：如何实现发布?**

订阅操作是一个“写”操作，相应的，发布操作就是一个“读”操作。发布的本质是触发安装在某个事件上的监听函数，我们需要做的就是找到这个事件对应的监听函数队列，将队列中的 handler 依次执行出队：

复制代码

```
// 别忘了我们前面说过触发时是可以携带数据的，params 就是数据的载体
emit(type, params) {
  // 假设该事件是有订阅的（对应的事件队列存在）
```

```
  if(this.eventMap[type]) {
    // 将事件队列里的 handler 依次执行出队
    this.eventMap[type].forEach((handler, index)=> {
      // 注意别忘了读取 params
      handler(params)
    })
  }
}
```

到这里，最最关键的 on 方法和 emit 方法就实现完毕了。最后我们补充一个 off 方法：

复制代码

```
off(type, handler) {
  if(this.eventMap[type]) {
    this.eventMap[type].splice(this.eventMap[type].indexOf(handler)>>>0,1)
  }
}
```

接着把这些代码片段拼接进一个 class 里面，一个核心功能完备的 EventEmitter 就完成啦：

复制代码

```
class myEventEmitter {
  constructor() {
    // eventMap 用来存储事件和监听函数之间的关系
    this.eventMap = {};
  }
  // type 这里就代表事件的名称
  on(type, handler) {
    // hanlder 必须是一个函数，如果不是直接报错
    if (!(handler instanceof Function)) {
      throw new Error("哥 你错了 请传一个函数");
    }
    // 判断 type 事件对应的队列是否存在
    if (!this.eventMap[type]) {
      // 若不存在，新建该队列
      this.eventMap[type] = [];
    }
    // 若存在，直接往队列里推入 handler
    this.eventMap[type].push(handler);
  }
  // 别忘了我们前面说过触发时是可以携带数据的，params 就是数据的载体
  emit(type, params) {
    // 假设该事件是有订阅的（对应的事件队列存在）
    if (this.eventMap[type]) {
      // 将事件队列里的 handler 依次执行出队
      this.eventMap[type].forEach((handler, index) => {
        // 注意别忘了读取 params
        handler(params);
      });
    }
  }
  off(type, handler) {
    if (this.eventMap[type]) {
      this.eventMap[type].splice(this.eventMap[type].indexOf(handler) >>> 0, 1);
```

```
    }
  }
}
```

下面我们对 myEventEmitter 进行一个简单的测试，创建一个 myEvent 对象作为 myEventEmitter 的实例，然后针对名为 “test” 的事件进行监听和触发：

复制代码

```
// 实例化 myEventEmitter
const myEvent = new myEventEmitter();
// 编写一个简单的 handler
const testHandler = function (params) {
  console.log(`test事件被触发了，testHandler 接收到的入参是${params}`);
};
// 监听 test 事件
myEvent.on("test", testHandler);
// 在触发 test 事件的同时，传入希望 testHandler 感知的参数
myEvent.emit("test", "newState");
```

以上代码会输出下面红色矩形框住的部分作为运行结果：

```
> // 实例化 myEventEmitter
const myEvent = new myEventEmitter();
// 编写一个简单的 handler
const testHandler = function (params) {
  console.log(`test事件被触发了，testHandler 接收到的入参是${params}`);
};

// 监听 test 事件
myEvent.on("test", testHandler);

// 在触发 test 事件的同时，传入希望 testHandler 感知的参数
myEvent.emit("test", "newState");
test事件被触发了，testHandler 接收到的入参是newState
```


VM537:5

由此可以看出，EventEmitter 的实例已经具备发布-订阅的能力，执行结果符合预期。

现在你可以试想一下，对于任意的两个组件 A 和 B，假如我希望实现双方之间的通信，借助 EventEmitter 来做就很简单了，以数据从 A 流向 B 为例。

我们可以在 B 中编写一个handler（记得将这个 handler 的 this 绑到 B 身上），在这个 handler 中进行以 B 为上下文的 this.setState 操作，然后将这个 handler 作为监听器与某个事件关联起来。比如这样：

复制代码

```
// 注意这个 myEvent 是提前实例化并挂载到全局的，此处不再重复示范实例化过程
const globalEvent = window.myEvent
class B extends React.Component {
  // 这里省略掉其他业务逻辑
  state = {
    newParams: ""
  };
  handler = (params) => {
    this.setState({
      newParams: params
    });
  };
  bindHandler = () => {
    globalEvent.on("someEvent", this.handler);
  };
  render() {
    return (
      <div>
        <button onClick={this.bindHandler}>点我监听A的动作</button>
        <div>A传入的内容是[{this.state.newParams}]</div>
      </div>
    );
  }
}
```

接下来在 A 组件中，只需要直接触发对应的事件，然后将希望携带给 B 的数据作为入参传递给 emit 方法即可。代码如下：

复制代码

```
class A extends React.Component {
  // 这里省略掉其他业务逻辑
  state = {
    infoToB: "哈哈哈哈我来自A"
  };
  reportToB = () => {
    // 这里的 infoToB 表示 A 自身状态中需要让 B 感知的那部分数据
    globalEvent.emit("someEvent", this.state.infoToB);
  };
  render() {
    return <button onClick={this.reportToB}>点我把state传递给B</button>;
  }
}
```

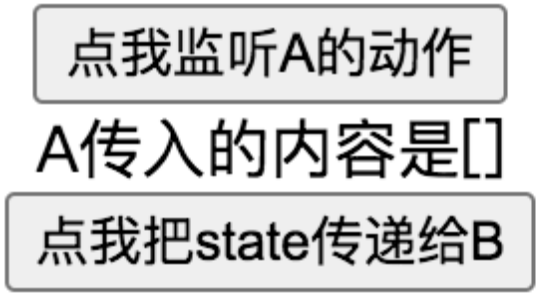
如此一来，便能够实现 A 到 B 的通信了。这里我将 A 与 B 编排为兄弟组件，代码如下：

复制代码

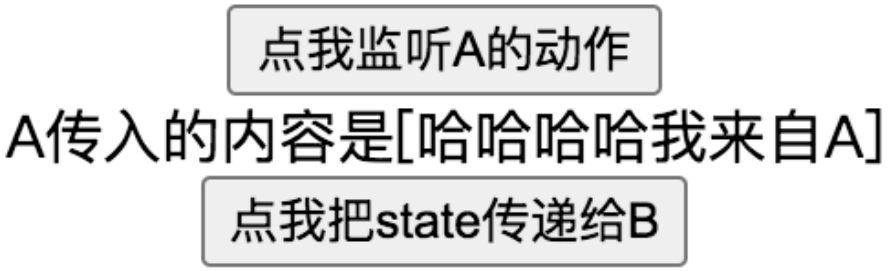
```
export default function App() {
  return (
    <div className="App">
      <B />
      <A />
    </div>
  );
```

```
8. }
```

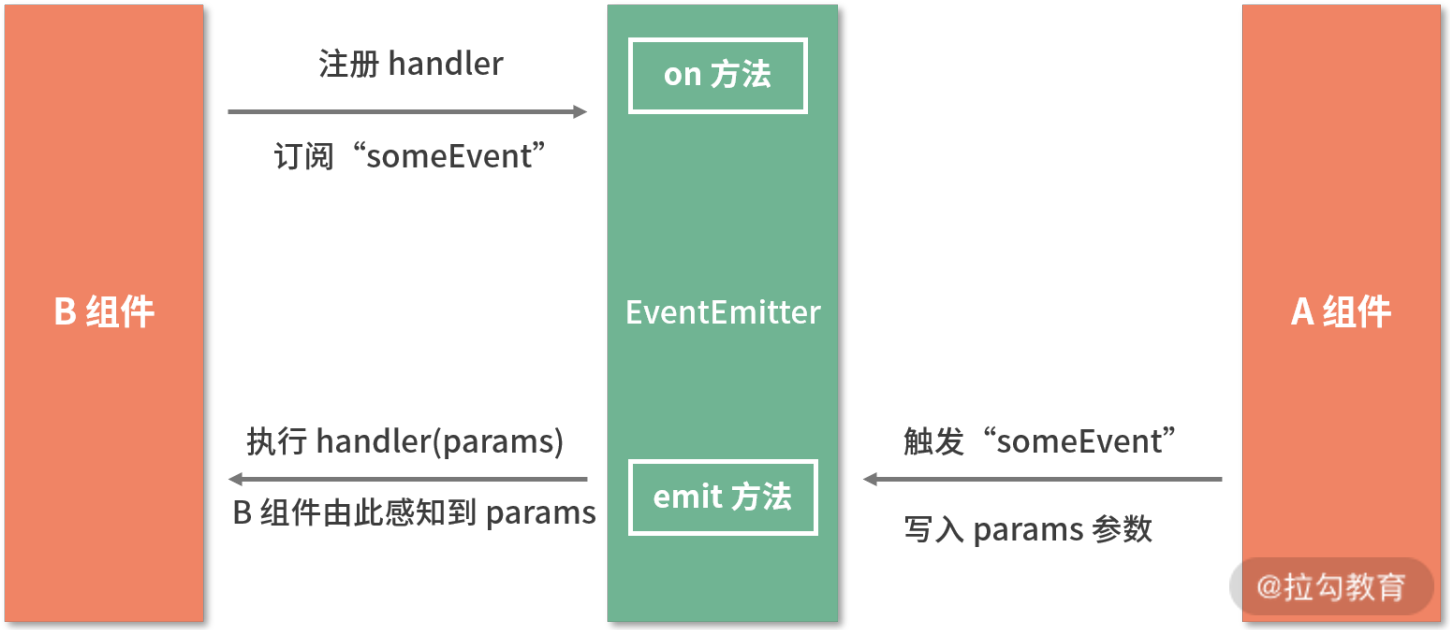
你也可以在自己的 Demo 里将 A 和 B 定义为更加复杂的嵌套关系，这里我给出的这个 Demo 运行起来会渲染出这样的界面，如下图所示：



依次点击顶部和底部的按钮，就可以实现对 someEvent 这个事件的监听和触发，进而观察到中间这行文本的改变，如下图所示：



由此我们便可以验证到发布-订阅模式驱动 React 数据流的可行性。为了强化你对过程的理解，我将 A 与 B 的通信过程梳理进了一张图里，供你参考：



总结

本课时，我们对 React 数据流管理方案中的前两个大方向进行了学习：

- 使用基于 Props 的单向数据流串联父子、兄弟组件；
- 利用“发布-订阅”模式驱动 React 数据在任意组件间流动。

这两个方向下的解决方案，单纯从理解上来看，难度都不高。**你需要把重点放在对编码的实现和理解上，尤其是基于“发布-订阅”模式实现的 EventEmitter，多年来一直是面试的大热点，务必要好好把握。**

这一课时就讲到这里了，下个课时，我们将继续站在“数据在 React 组件中的流动”这个视角，对 React 中的 Context API，以及第三方数据流管理框架中的“佼佼者” Redux 进行分析，相信一定能够为你带来不一样的理解和收获。