Communication between components

RUJUAN XING

Communication between Components

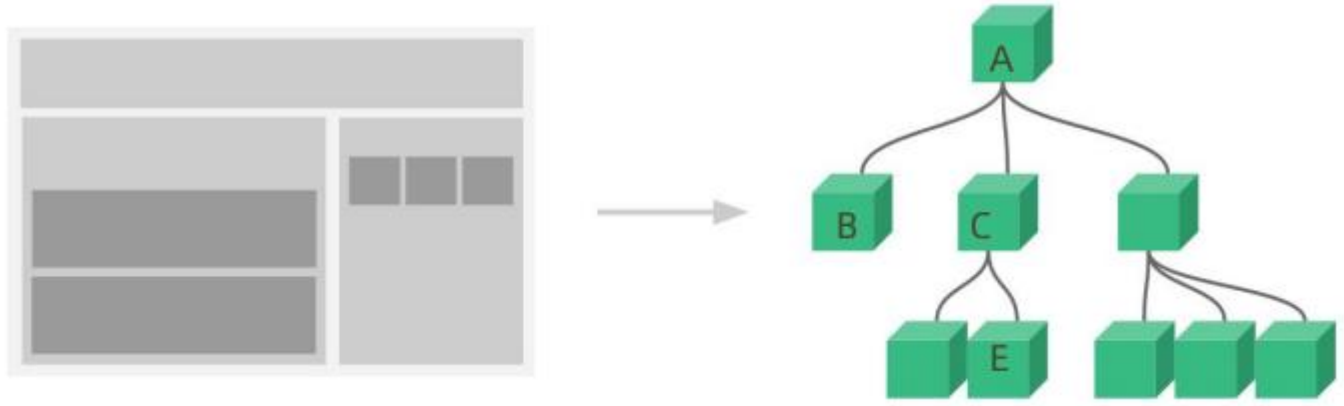
In modern front-end frameworks/libraries, the entire page is divided into multiple smaller components. Component-based architecture makes it easy to develop and maintain an application. During the development, you may come across a situation where you need to share the data with other components.

React provides several methods for handling the component communication, each with its appropriate use cases.

- Parent / Child communication
  - Parent to child communication
  - Child to parent communication
- Context API
- Centralized state with Redux
- …

Understanding Communication

Component communication refers to the transfer of data between components. Depending on the nesting relationship of the components, there are different communication methods.



A-B Parent-Child communication

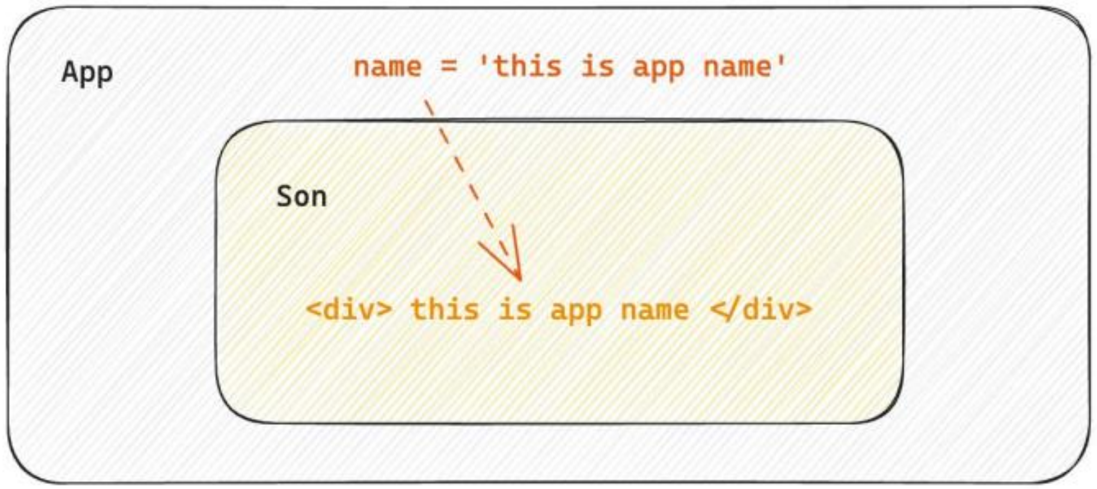
B-C Sibling communication

A-E Cross-layer communication

Parent to Child

Parent to Child communication

In React, parent components can communicate to child components using a special property defined by React called Props. Props are read-only and they're passed from parent to child component via HTML attributes.



Steps:

- Parent passes data - bind attributes to the child component.
- Child receives data - the child component receives data through `props` parameter.

Props

1. can pass any kind of data
   - number, string, boolean, array, object, function, JSX

```
<Son name={name}
      age={20}
      isTrue={false}
      list={['React', 'Angular']}
      obj={{ name: 'Jack' }}
      cb={() => console.log(123)}
      child={<span>this is a span child</span>}
/>
```

2. read only
   - The child component can only read the data in props and cannot directly modify it. The data in the parent component can only be modified by the parent component.

Props Children

When we nest content within the child component tags, the child component can get the content via the `children` property in props.

```
<Son>
     <span>This is a span in App</span>
</Son>
```

```
function Son(props: React.ReactNode) {
  return <div>this is son, {props.name}, {props.children}</div>
}
```

props & TypeScript

Use type or interface to add types to component props.

```
type Props = {
  className: string
}

function Button(props: Props){
  const {className} = props;
  return <button className={className}>Click
Me</button>
}
```

In the above example: The `Button` component can only accept a prop named '`className`', which must be of type `string` and is required.

Props & TypeScript - `children`

`children` is a special prop, supporting various types of data, requires internal `ReactNode` type for annotation.

`children` can be many types, such as: `React.ReactElement`, `string`, `number`, `React.ReactFragment`, `React.ReactPortal`, `Boolean`, `null`, `undefined`

```
<Son>
     <span>This is a span in App</span>
</Son>
```

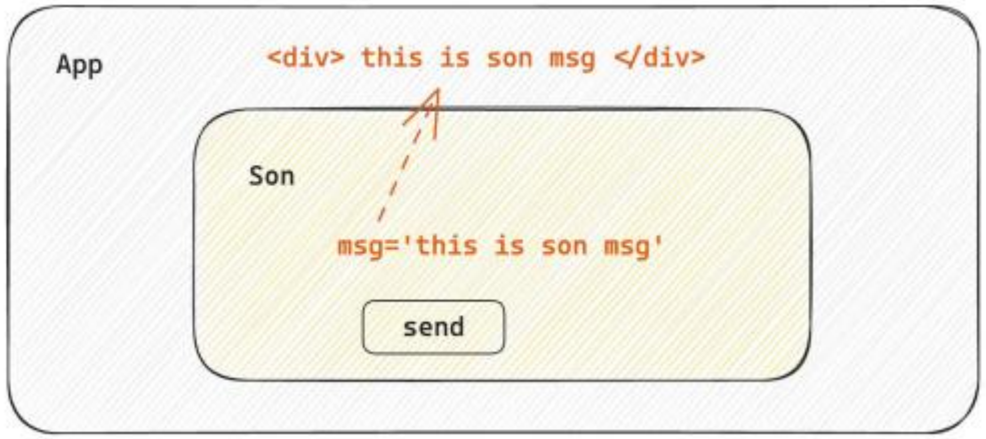
```
function Son(props: React.ReactNode) {
  return <div>this is son, {props.name}, {props.children}</div>
}
```

Child to Parent

Lifting State up:
Child to parent communication

Data from a child component to parent component can be passed using a callback. We can achieve this by following the below steps.

- Create a callback method in the parent component and pass it to the child using props.
- Child component can call this method using "`props.[callbackFunctionName]`" from the child and pass data as an argument.



Child to parent communication Demo

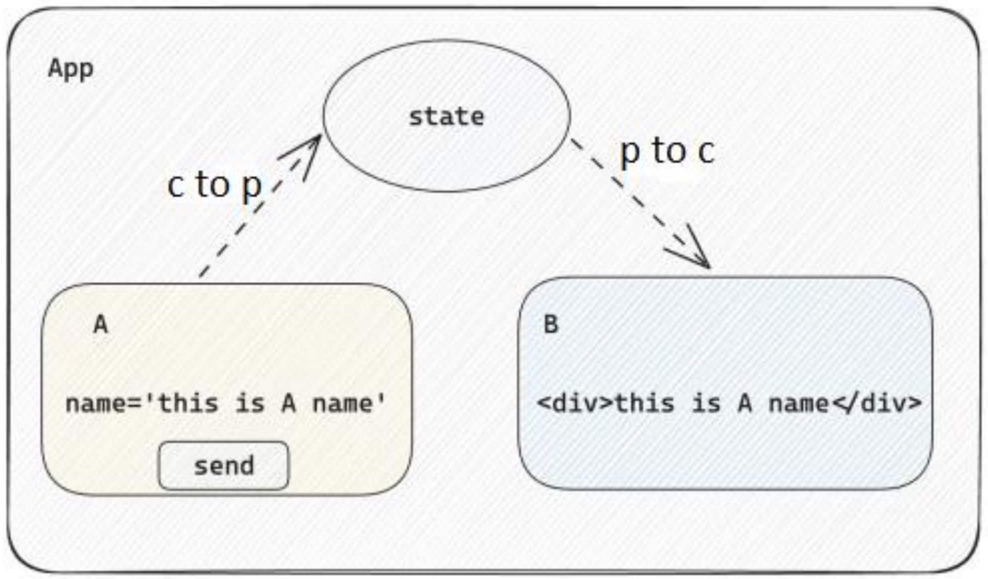
```
function App() {
  const [message, setMessage] = useState('');
  const getMsg = (msg:string) => setMessage(msg);
  return (
    <div>
      {message}
      <Son onGetMsg={getMsg}/>
    </div>
  );
}
```

```
function Son({onGetMsg}:{onGetMsg:(msg: string) => void}) {
  const sonMsg = 'This is Son Msg';
  return (
    <div>
      <button onClick={() => onGetMsg(sonMsg)}>Send</button>
    </div>
  );
}
```

Two Sibling

Two sibling components passing data to each other

Sibling communication is merely the combination of prop drilling and lifting state up.



1. The child A that wants to pass data to another sibling, sends it to its parent first using callback mechanism.
2. The parent receives the data from one child and then passes it to the second child B using props of that child.

Two sibling communication Demo

```
function A({onGetMsg}: {onGetMsg: (msg: string) => void}){
   const name = 'this is A name';
   return <button onClick={() => onGetMsg(name)}></button>
}
```

```
function App() {
  const [message, setMessage] = useState('');
  const getMsg = (msg:string) => setMessage(msg);
  return (
    <div>
      <A onGetMsg={getMsg}/>
      <B message={message}/>
    </div>
  );
}
```

```
function B({message}: {message: string}){
  return <div>{message}</div>
}
```

Communicate between components across different levels

The Problem - “Prop Drilling”

State should be held by the highest parent component in the stack that requires access to the state.

To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.

To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

```
export default function Component1() {
   const [user, setUser] = useState("Jesse Hall");

   return (
      <>
         <h1>{`Hello ${user}!`}</h1>
         <Component2 user={user} />
      </>
   );
}

function Component2({ user }) {
   return (
      <>
         <h1>Component 2</h1>
         <Component3 user={user} />
      </>
   );
}
```

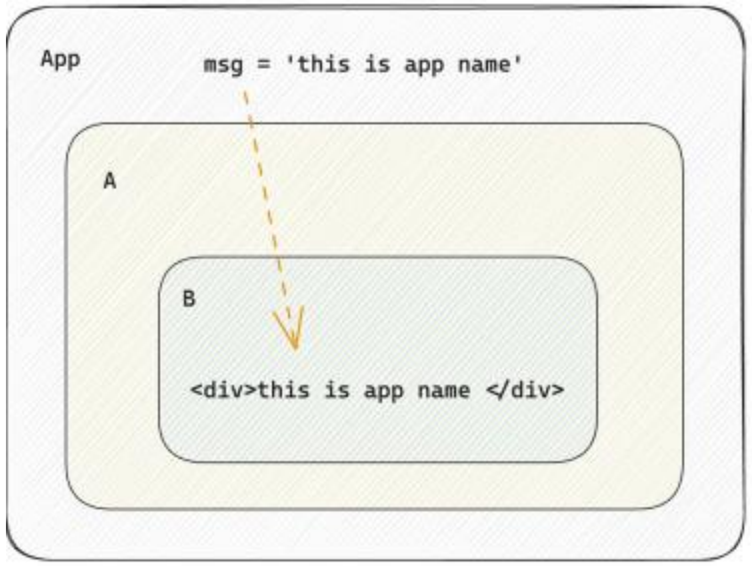
```
function Component3({ user }) {
   return (
      <>
         <h1>Component 3</h1>
         <Component4 user={user} />
      </>
   );
}

function Component4({ user }) {
   return (
      <>
         <h1>Component 4</h1>
         <Component5 user={user} />
      </>
   );
}

function Component5({ user }) {
   return (
      <>
         <h1>Component 5</h1>
         <h2>{`Hello ${user} again!`}</h2>
      </>
   );
}
```

React Context API

The Context API is designed to manage simple, globally interesting values. That is to say, values that are used by many components across the app.



Using Context

Using Context requires 3 simple steps:

1. creating the context
2. providing the context
3. consuming the context

Step 1: Creating the context

`createContext` lets you create a context that components can provide or read.

```
const someContext = createContext(defaultValue)
```

- `defaultValue`: The value that you want the context to have when there is no matching context provider in the tree above the component that reads context. If you don’t have any meaningful default value, specify null. The default value is meant as a “last resort” fallback. It is static and never changes over time.
- `someContext`: `createContext` returns a context object.

```
import {createContext} from "react";

const MsgContext = createContext("This is default value.");
```

Step 2: providing the context

`Context.Provider` component available on the context instance is used to provide the context to its child components, no matter how deep they are.

To set the value of context use the value prop available on the `<Context.Provider value={value} />`.

```
function App() {
  const msg = 'this is app message';
  return (
    <div>
      <MsgContext.Provider value={msg}>
        this is App
        <A />
      </MsgContext.Provider>
    </div>
  );
}
```

Step 3: Consuming the context

Consuming the context can be performed in 2 ways:

1. Use the `useContext(Context)` React hook:

```
const value = useContext(SomeContext)
```

- `SomeContext`: The context that you've previously created with `createContext`.
- `useContext` returns the context value for the calling component.

```
function B(){
  const msg = useContext(MsgContext);
  return (
    <div>
      this is B component, {msg}
    </div>
   )
}
```

Step 3: Consuming the context (cont.)

2. Use a render function supplied as a child to `Context.Consumer` special component available on the context instance

```
function B() {

  return (
    <MsgContext.Consumer>
      {
        msg => (
          <div>
            this is B component, {msg}
          </div>
        )
      }
    </MsgContext.Consumer>
  )
}
```