React - rendering

Cooper Duan

What is "Rendering"?

"Rendering" is a process React asking components to describe what they want the UI to look like, based on the current component's `props` and `state`.

- **React Element** - `JSX` will eventually be compiled into a call to `React.createElement` at compile time.
- **React tree** - the entire component tree
- **Reconciliation** - Diff the previous element tree vs the current element tree to decide what actual changes are necessary

```
// This JSX syntax:
return <MyComponent a={42} b="testing">Text here</MyComponent>

// is converted to this call:
return React.createElement(MyComponent, {a: 42, b: "testing"}, "Text Here")

// and that becomes this element object:
{type: MyComponent, props: {a: 42, b: "testing"}, children: ["Text Here"]}

// And internally, React calls the actual function to render it:
let elements = MyComponent({...props, children})

// For "host components" like HTML:
return <button onClick={() => {}}>Click Me</button>
// becomes
React.createElement("button", {onClick}, "Click Me")
// and finally:
{type: "button", props: {onClick}, children: ["Click me"]}
```

```
const element = React.createElement(
  'div' | HeaderComponent,      // type
  { className: 'container' },   // props
  'Hello, world!'               // children
);
```

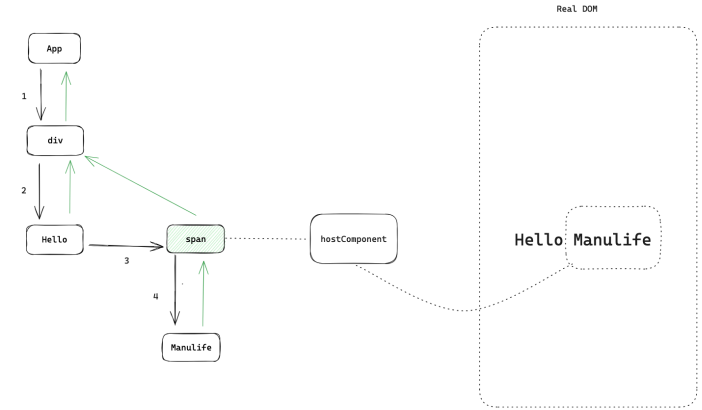
Render and Commit phases

Each rendering process is divided into two phases

Commit

Apply changes to the DOM and excute effects and lifecyles

- sync apply DOM updates
- Effects and class component lifecycles, run cleanup




```
functoin App() {

  useEffect(() => {
    console.log('2')
  })

  useLayoutEffect(() => {
    console.log('1')
  })

  return <div>...</div>
}
```

How to trigger rendering

Every React render starts with a state update call

```jsx
const App = () => {
  const[, forceRender] = useReducer((x) => x + 1, 0);
  return (
    <div>
      Hello <span className="text-red-500">World !</span>
    </div>
  );
};
```

Function components

- `useState`
- `useReducer`
- `useSyncExternalStore`

Class components

- `this.setState()`
- `this.forceUpdate()`

Standard render flow

Rendering a component will by default cause all components inside of it to be rendered.

TIPS:

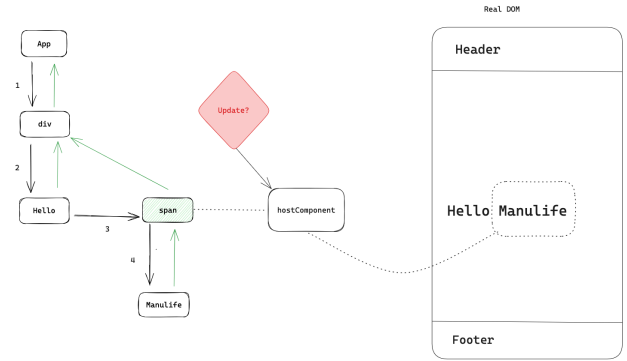
In normal rendering, React does not care whether "props changed", component will render just because parent component rendered.

```
A > B > C > D

A call setState
B render
C render cause by B
D render cause by C
```

Rendering vs DOM Updates

"Rendering" does not mean "apply updates to the DOM"

- A component may return an equivalent tree of elements as last time. So no changes need to be Update DOM here
- React have to go through the process of rendering and find which component need to be updated




```
function updateDOMProperties(domElement, updatePayload, lastProps, nextProps) {
  // 移除旧的属性和事件监听器
  for (const propKey in lastProps) {
    if (lastProps.hasOwnProperty(propKey) && !nextProps.hasOwnProperty(propKey)) {
      if (propKey === 'style') {
        const lastStyle = lastProps[propKey];
        for (const styleName in lastStyle) {
          domElement.style[styleName] = '';
        }
      } else if (propKey.startsWith('on')) {
        const eventType = propKey.toLowerCase().substring(2);
        domElement.removeEventListener(eventType, lastProps[propKey]);
      } else {
        domElement.removeAttribute(propKey);
      }
    }
  }
```

Rules of React Rendering

"Rendering" must be "pure" and not have any "side effects".
"side effects" means to break render things, like mutate a props is very bad.

Render logic can

- mutate objects that were newly created while rendering
- throw errorss
- "Lazy initialize"

Render logic can not

- mutate existing variables and objects
- make network requests

Fibers

React store component "instance" and metadata in internal "Fiber" object.

- props and state
- hooks and class instance
- Pointers to parent, sibling, and child components
- metadata that React uses to track rendering process

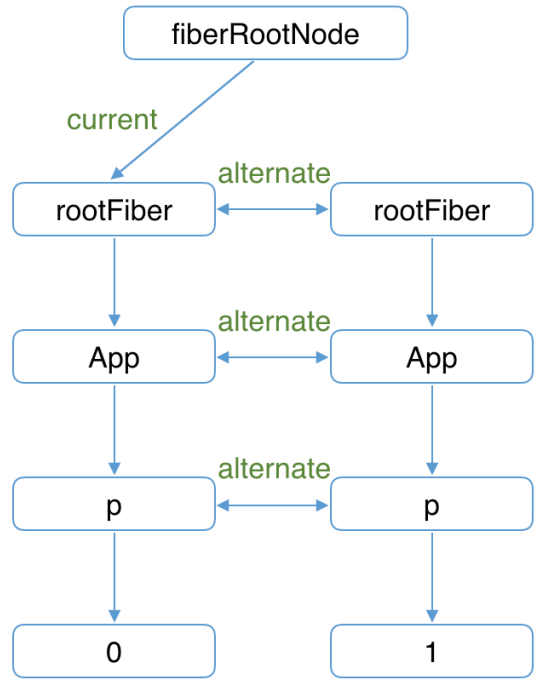
```
function FiberNode(
  tag: WorkTag,
  pendingProps: mixed,
  key: null | string,
  mode: TypeOfMode,
) {
  this.tag = tag;             // identifying the type of
  this.key = key;             // unique identifier of th
  this.type = null;           // the resolved function/c
  this.stateNode = null;      // the real DOM node for c
  // Used to connect other Fiber nodes to form a Fiber
  this.return = null;
  this.child = null;
  this.sibling = null;
  this.index = 0;

  this.ref = null;
  this.pendingProps = pendingProps;
  this.memoizedProps = null;    // memoizedProps is use
  this.updateQueue = null;      // is a queue of state
  this.memoizedState = null;    // is used to create th

  this.effectTag = NoEffect; // NoEffect | Update | Pla
  this.alternate = null; // Points to the fiber corresp


  // 指向下一个有副作用的 fiber
  this.nextEffect = null;
```

The two fiber tree

React keeps two fiber tree currentFiber and workInProgressFiber

mount:

render:



Rendering optimization

Component Types

React reuses existing component instances and DOM nodes as much as possible.
We can do something to speed up reconciliation

- reuse type: `if(prevElement.type !== currentElement.type)` React assumes sub tree will be different

it means React will destory that entire existing component tree including all DOM nodes.
we can return the same type every time to speed up rendering.

```
function ChildComponent() {
  return <div>child</div>;
}
const App = () => {
  return <ChildComponent />;
};
```

Rendering optimization

Key

React identifies component instances via `<App key="app-unique-id" />`

- `key` cannot pass from parent to child component

React implements fast list updates by detecting keys

```
const App = () => {
  return data.map((item) => <Item key={item.key} />)
}
```

```
// tips: The optimization will only take effect after t
const App = () => {
  return isActive
    ? <AComponent key="A" />
    : <BComponent key="B />
}
```

```
// tips: The optimization will only take effect after t
const App = () => {
  const data = useMemo(() => isActive ? <AComponent />
  return data
}
```

Async Rendering and Closures

Extremely common user mistake:
set new value, then try to log the existing variable.
It it not work in React.

Common answer:
"React rendering is async". Really two reasons:

- render will be sync, but the event handler is a "closures",
- and it still sees the original value the component last rendered.

```
import React, { useState } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  function onClickFactory(capturedCount) {
    return function() {
      setCount(capturedCount + 1);
      console.log("count: ", capturedCount);
    };
  }

  const onClick = onClickFactory(count);

  return (
    <div>
      <p>{count}</p>
      <button onClick={onClick}>Increment</button>
    </div>
  );
}

export default Example;
```

Training

What happen here? and how to fix it.

```
// 2. Add loading
function Avatar({ id }){
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setLoading(true); // Start loading
    fetchData(`/api/user/${id}`)
      .then((res) => {
        setData(res.data)
      })
      .finally(() => {
        setLoading(false);
      })
  }, [id])
  return data
}
```

```
// 3. Avoiding memory issues
// 4. Exception handling
function Avatar({ id }) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    let isMounted = true; // Track if component is mour
    const getData = async () => {
      setLoading(true); // Start loading
      try {
        const res = await fetch(`/api/user/${id}`).ther
        if (isMounted) {
          setData(res.data);
          setLoading(false); // Finish loading
        }
      } catch (error) {
        if (isMounted) {
          console.error("Error fetching data:", error);
          setLoading(false); // Finish loading even if
        }
      }
    };
    getData();
    return () => {
      isMounted = false; // Cleanup function
    };
  }, [id]); // Dependency array
```

SWR

You can use third-party libraries to simplify logic

```
function Avatar({ id }){
  const { data, isLoading, error }= useSWR(`/api/user/${id}`);
  }, [id])

  if (loading) return <div>Loading...</div>;
  if (!data) return <div>No data available</div>;
  return <div>{data}</div>;
}
```

Thank you

Cooper Duan