HOW REACT WORKS

- Overview:

- HTML, DOM, JSX

- createElement

- Fiber

- Q&A

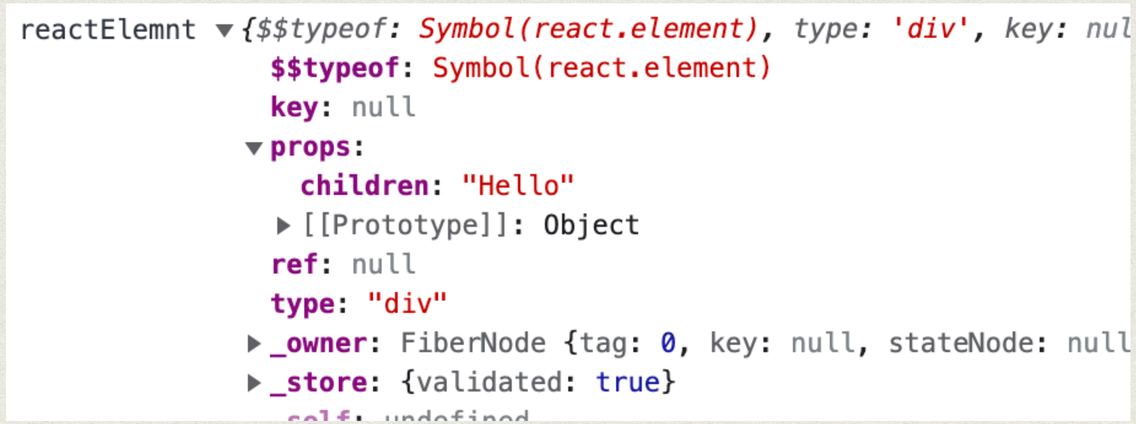
- Bonus

HTML, DOM, JSX

```jsx
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

```jsx
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```



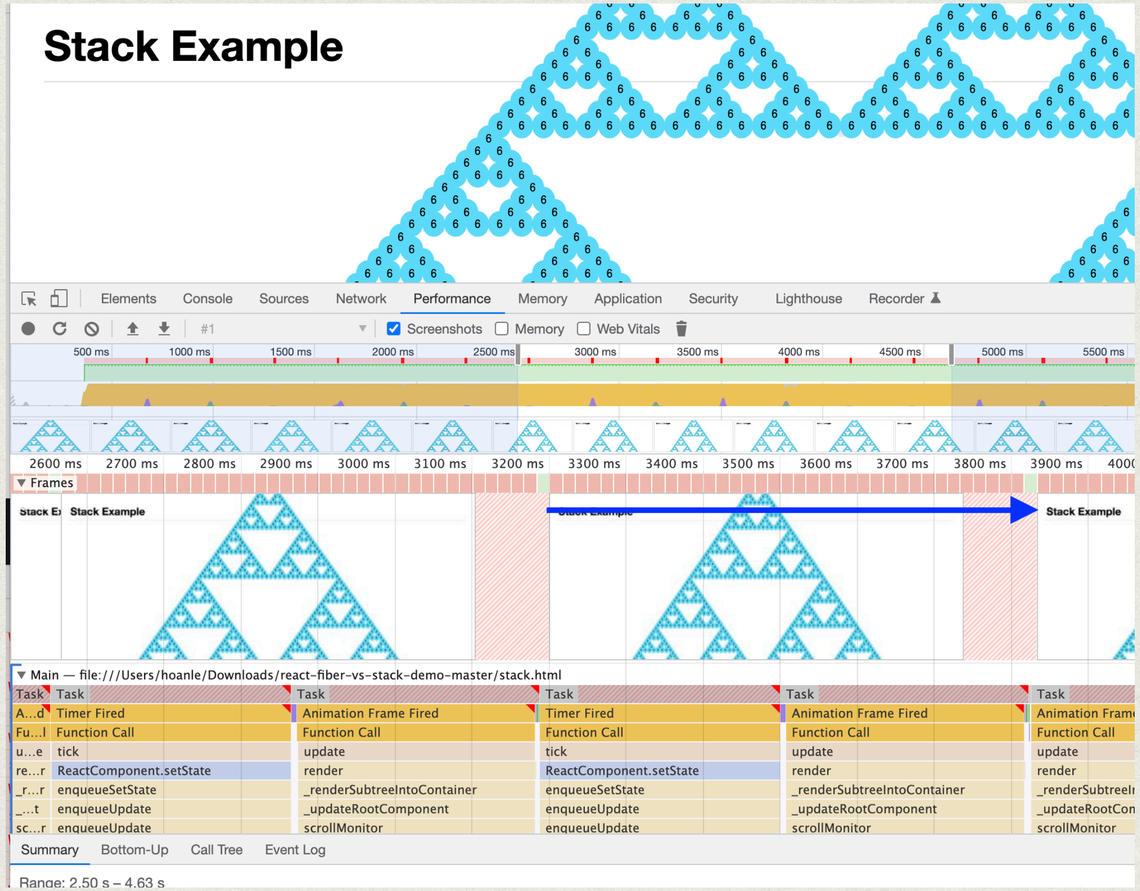
JSX-> DOM

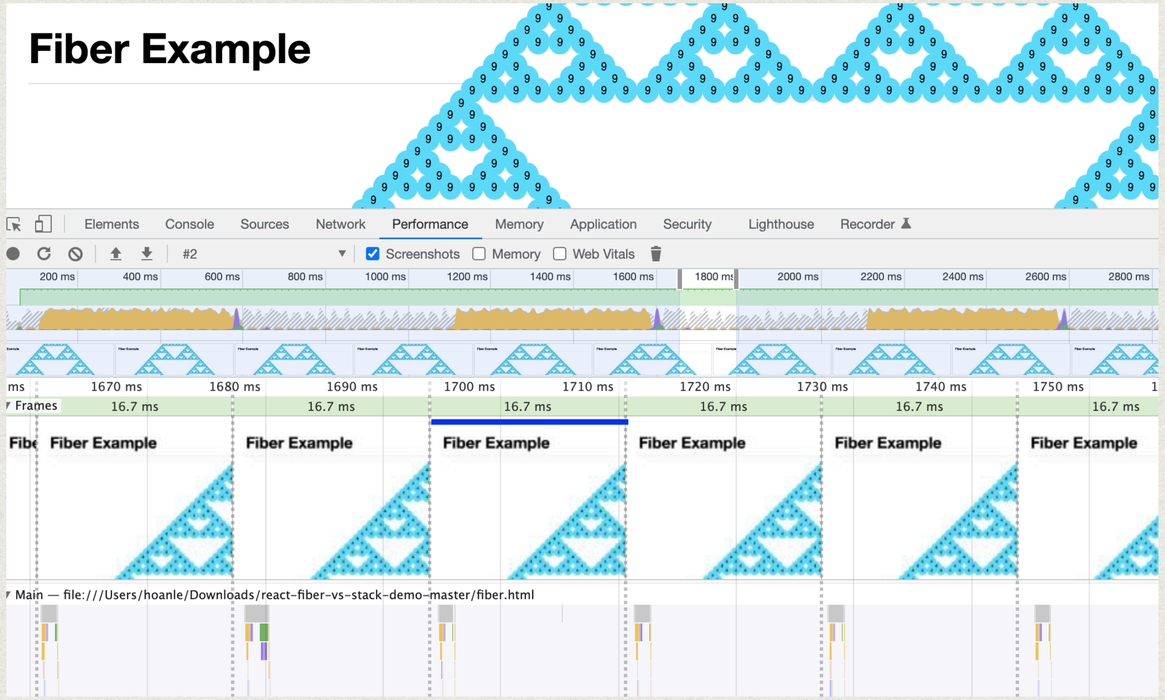
- From the Object => how to build it into Dom

STACK RECONCILER

- Old react version ~v15
- Traverse and process a ReactElement tree recursively
- Diff and Path are the same time during traversing ReactElement tree
- Process synchronously
- Leads to block the UI thread

STACK VS FIBER





REQUESTIDLECALLBACK

- `window.requestIdleCallback()` method queues a function to be called during a browser's idle periods.
- This enables developers to perform background and low priority work on the main event loop,

FIBER RECONCILER

- Traverse and process a ReactElement tree through Fiber objects
- Render Phase and Commit Phase
- Fiber can suspend and resume in its Render Phase

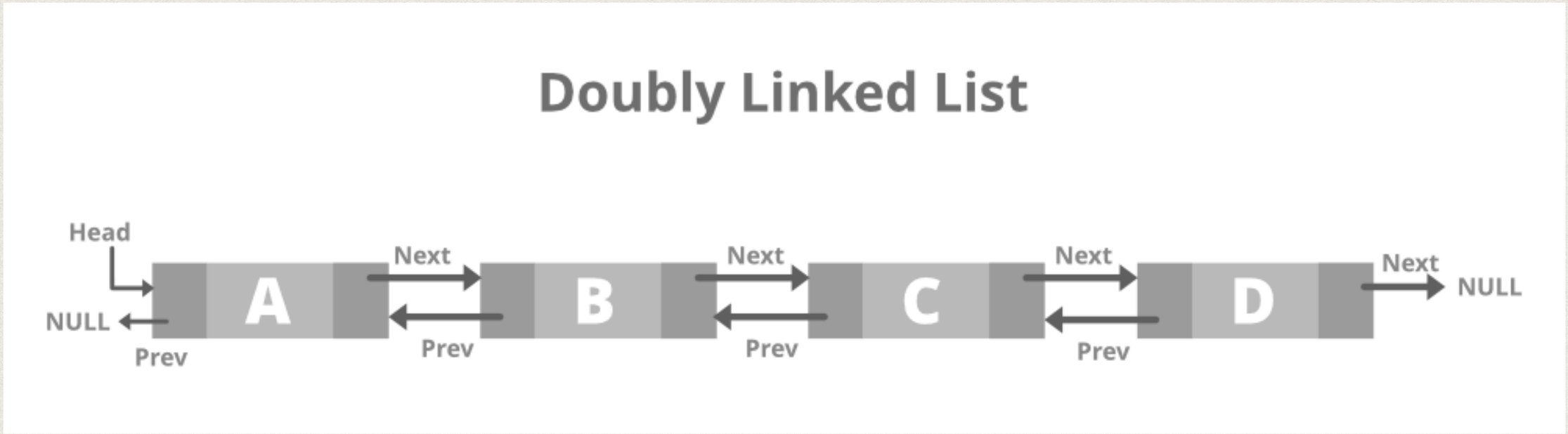
FIBER

- Fiber is an unit of work
- A ReactElement tree becomes a Linked list of Fibers
- A Fiber has an alternate Fiber (workInProgress <-> current)

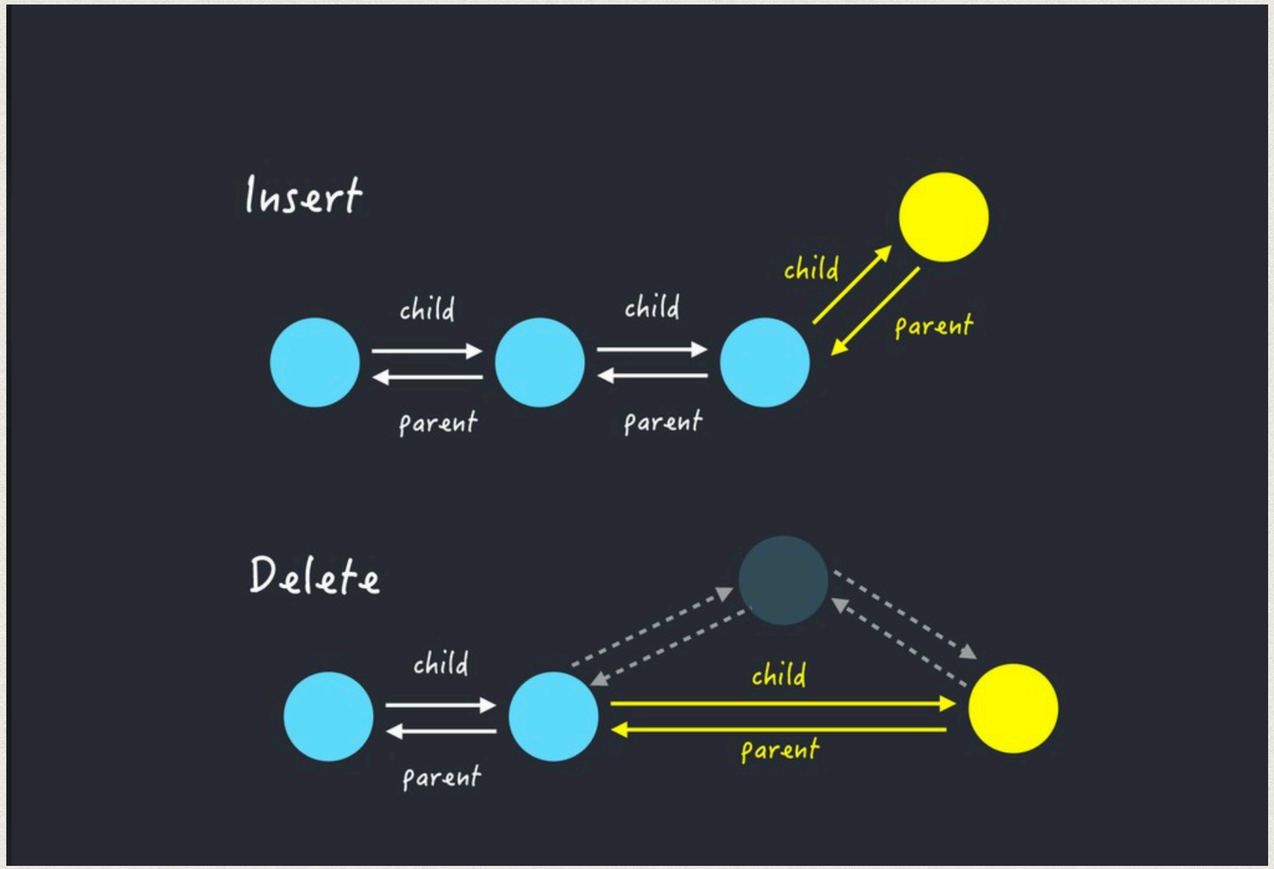
LINKED LIST

- A ReactElement tree becomes A Linked list of fibers
- Fiber process sequentially
- Search O(n)
- Insert/Delete O(1)

LINKED LIST

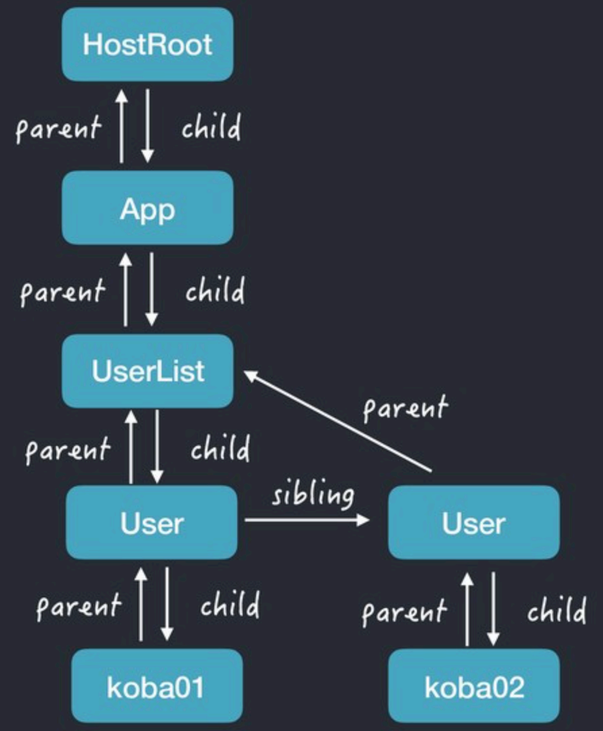

Doubly Linked List
Head
NULL
Prev
A
Next
Prev
B
Next
Prev
C
Next
Prev
D
Next
NULL

LINKED LIST


Insert
child
child
child
parent
parent
parent
Delete
child
child
parent
parent

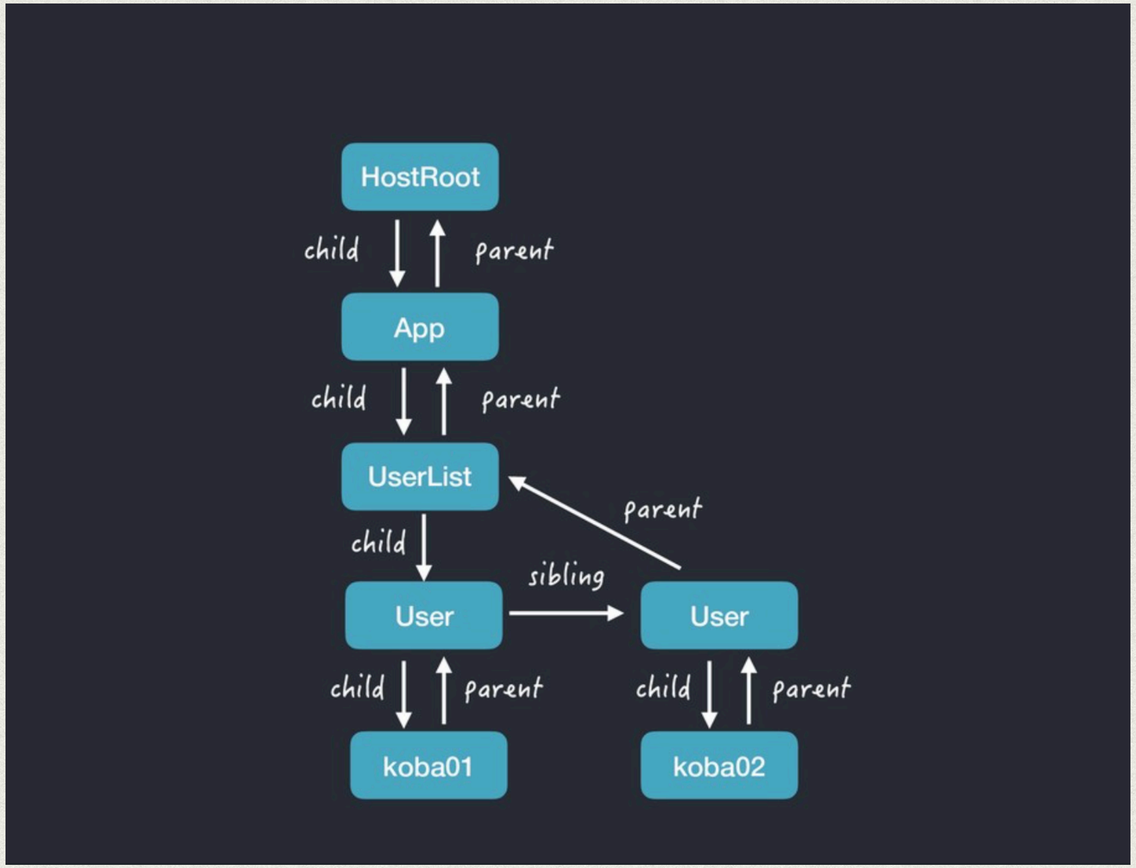
FIBER TREE

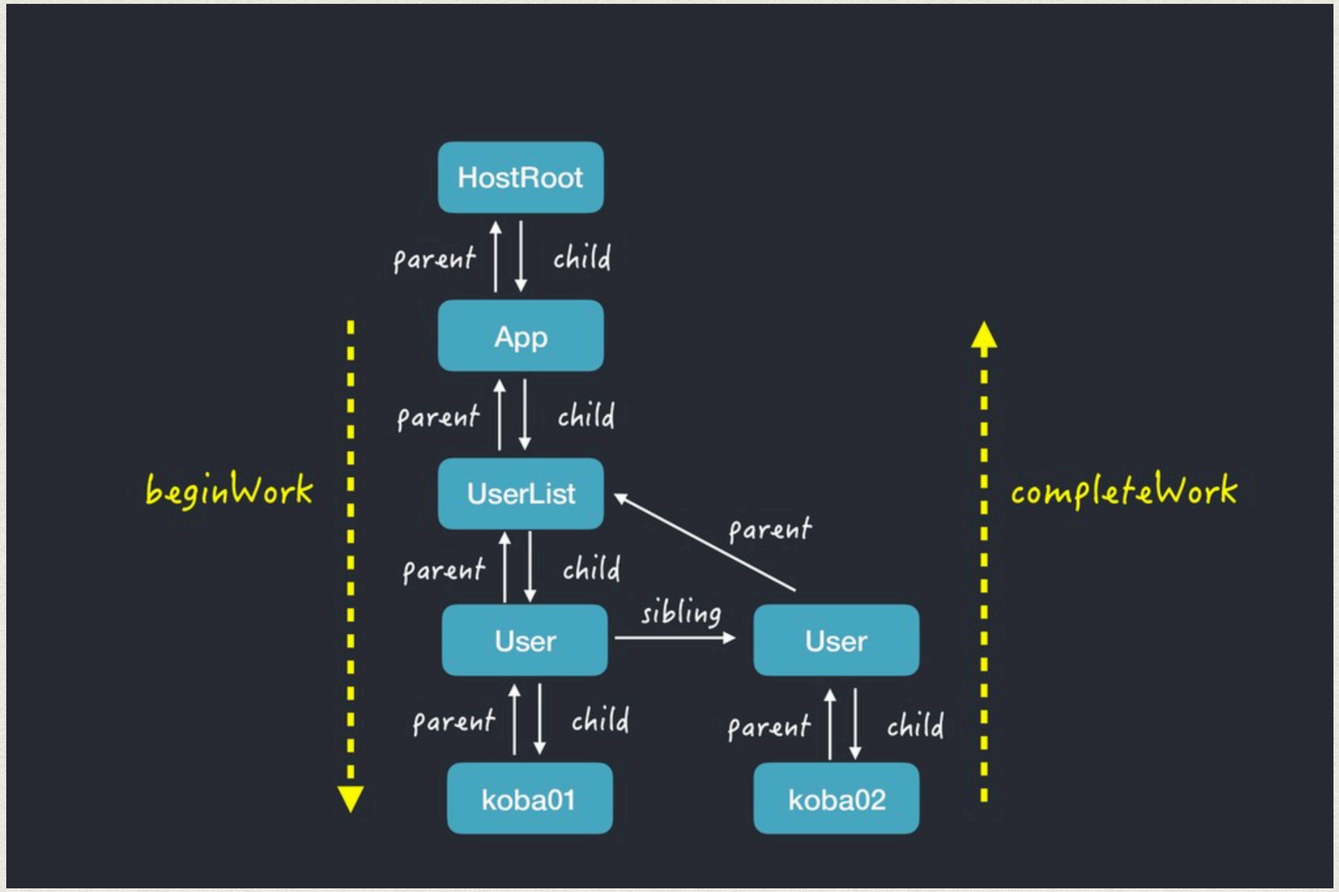
```
<App>
  <UserList>
    <User name="koba01" />
    <User name="koba02" />
  </UserList>
</App>
```



TRAVERSE FIBER TREE

```
function performUnitOfWork(fiber) {
  const isFunctionComponent = fiber.type instanceof Function
  if (isFunctionComponent) {
    updateFunctionComponent(fiber)
  } else {
    updateHostComponent(fiber)
  }
  if (fiber.child) {
    return fiber.child
  }
  let nextFiber = fiber
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling
    }
    nextFiber = nextFiber.parent
  }
}
```




HostRoot
parent
child
App
parent
child
beginWork
UserList
completeWork
parent
parent
child
sibling
User
User
parent
child
parent
child
koba01
koba02

Q&A

- Q & A

HOOKS

- Why can not put hooks in condition
- useEffect(()=> callback)) => call every time
- useEffect(()=> callback),[]) => call didmount
- useEffect(()=> callback),[a,b])

BITWISE IN REACT

- What is that magic numbers :
- 65536, 128 , 131072, ... ??
- |= 65536 , &= -52805 ??

```
7088  function markSuspenseBoundaryShouldCapture(
7089    suspenseBoundary,
7090    returnFiber,
7091    sourceFiber,
7092    root,
7093    rootRenderLanes
7094  ) {
7095    if (0 === (suspenseBoundary.mode & 1))
7096      return (
7097        suspenseBoundary === returnFiber
7098          ? (suspenseBoundary.flags |= 65536)
7099          : ((suspenseBoundary.flags |= 128),
7100            (sourceFiber.flags |= 131072),
7101            (sourceFiber.flags &= -52805),
7102            1 === sourceFiber.tag &&
7103              (null === sourceFiber.alternate
7104                ? (sourceFiber.tag = 17)
7105                : ((returnFiber = createUpdate(-1, 1)),
7106                  (returnFiber.tag = 2),
7107                  enqueueUpdate(sourceFiber, returnFiber))),
7108            (sourceFiber.lanes |= 1)),
7109        suspenseBoundary
7110      );
```

BITWISE IN REACT

- + Doing with bitwise always faster
- - sometime it’s hard to read and debug
- Other frameworks , libraries also use bitwise a lot. Ex: Angular

HOW REACT COMPARE VALUE

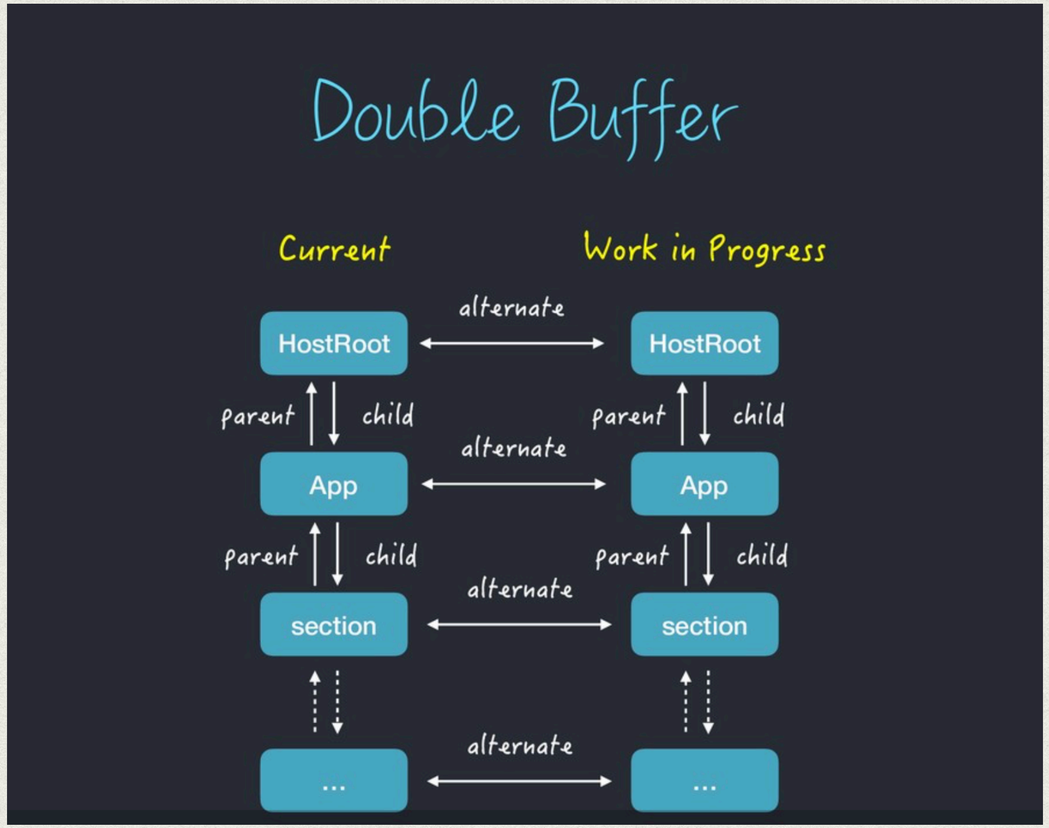
```
> NaN === NaN
< false
> <div> {NaN} </div>
  update every time ???
```

```
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (is(objA, objB)) {
    return true;
  }

  if (
    typeof objA !== 'object' ||
    objA === null ||
    typeof objB !== 'object' ||
    objB === null
  ) {
    return false;
  }

  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);

  if (keysA.length !== keysB.length) {
    return false;
  }

  // Test for A's keys different from B.
  for (let i = 0; i < keysA.length; i++) {
    const currentKey = keysA[i];
    if (
      !hasOwnProperty.call(objB, currentKey) ||
      !is(objA[currentKey], objB[currentKey])
    ) {
      return false;
    }
  }

  return true;
}
```

OBJECT.IS()

- Object.is('foo', 'foo'); // true
- Object.is('NaN', 'NaN'); // true
- Object.is(-1,1); // false
- Object.is(-0,0); // false

DOUBLE BUFFER



The Pattern

A **buffered class** encapsulates a **buffer**: a piece of state that can be modified. This buffer is edited incrementally, but we want all outside code to see the edit as a single atomic change. To do this, the class keeps *two* instances of the buffer: a **next buffer** and a **current buffer**.

When information is read *from* a buffer, it is always from the *current* buffer. When information is written *to* a buffer, it occurs on the *next* buffer. When the changes are complete, a **swap** operation swaps the next and current buffers instantly so that the new buffer is now publicly visible. The old current buffer is now available to be reused as the new next buffer.

SYMBOL

- $$typeof
- Symbol.for(‘react.element’)

This works because you can’t just put `Symbol`s in JSON. **So even if the server has a security hole and returns JSON instead of text, that JSON can’t include `Symbol.for('react.element')`**. React will check `element.$$typeof`, and will refuse to process the element if it’s missing or invalid.