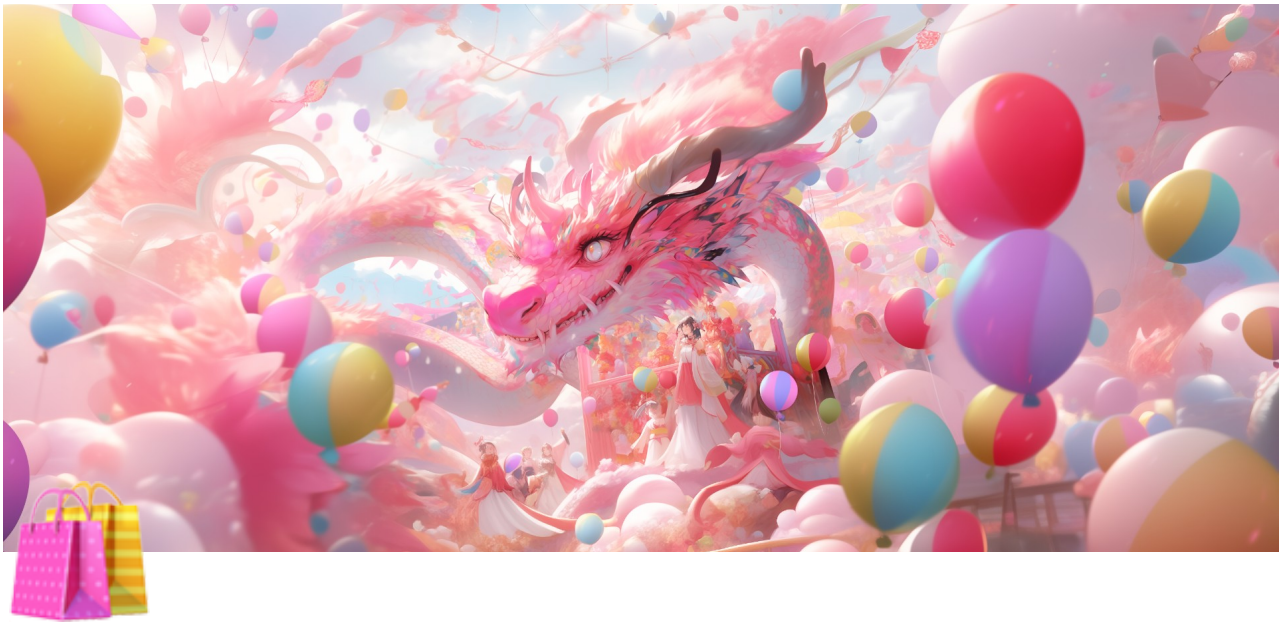

12-4 初步实现多个节点的React VDOM DIFF

非页面初次渲染。

TypeScript

```
function FunctionComponent() {
  const [count1, setCount1] = useReducer((x) => x + 1, 0);
  const arr = count1 % 2 === 0 ? [0, 1, 2, 3, 4] : [0, 1, 2, 3];

  // 0 删除
  return (
    <div className="border">
      <h3>函数组件</h3>
      <button
        onClick={() => {
          setCount1();
        }}
      >
        {count1}
      </button>
      <ul>
        {arr.map((item) => (
          <li key={"li" + item}>{item}</li>
```

```
        ))}
      </ul>
    </div>
  );
}
```

流程

1. 从左边往右遍历，比较新老节点，如果节点可以复用，继续往右，否则就停止

2.1 新节点没了，（老节点还有）。则删除剩余的老节点即可

2.2 (新节点还有)，老节点没了

3. 新老节点都还有节点，但是因为老 fiber 是链表，不方便快速 get 与 delete，
    因此把老 fiber 链表中的节点放入 Map 中，后续操作这个 Map 的 get 与 delete

4. 如果是组件更新阶段，此时新节点已经遍历完了，能复用的老节点都用完了，
    则最后查找 Map 里是否还有元素，如果有，则证明是新节点里不能复用的，也就是要被删除的元素，此时删除这些元素就可以了

源码实现

TypeScript

```
function reconcileChildrenArray(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChildren: Array<any>
) {
  let resultingFirstChild: Fiber | null = null; // 头结点
  let previousNewFiber: Fiber | null = null;
  let oldFiber = currentFirstChild;
  let nextOldFiber = null; //暂存oldFiber
  let newIdx = 0;
  let lastPlacedIndex = 0;

  // ! 1. 从左边往右遍历，比较新老节点，如果节点可以复用，继续往右，否则就停止
  for (; oldFiber !== null && newIdx < newChildren.length; newIdx++)
    if (oldFiber.index > newIdx) {
      nextOldFiber = oldFiber;
      oldFiber = null;
    } else {
      nextOldFiber = oldFiber.sibling;
    }

    const newFiber = updateSlot(returnFiber, oldFiber, newChildren[n

    if (newFiber === null) {
      if (oldFiber === null) {
        oldFiber = nextOldFiber;
      }
      break;
    }

    if (shouldTrackSideEffects) {
      if (oldFiber && newFiber.alternate === null) {
        deleteChild(returnFiber, oldFiber);
      }
    }

    lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
```

```
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
      oldFiber = nextOldFiber;
    }

    // !2.1 新节点没了，（老节点还有）。则删除剩余的老节点即可
    if (newIdx === newChildren.length) {
      deleteRemainingChildren(returnFiber, oldFiber);
      return resultingFirstChild;
    }
    // ! 2.2 (新节点还有)，老节点没了
    if (oldFiber === null) {
      for (; newIdx < newChildren.length; newIdx++) {
        const newFiber = createChild(returnFiber, newChildren[newIdx])
        if (newFiber === null) {
          continue;
        }
        lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx
        if (previousNewFiber === null) {
          // 第一个节点，不要用newIdx判断，因为有可能有null，而null不是有效fib
          resultingFirstChild = newFiber;
        } else {
          previousNewFiber.sibling = newFiber;
        }
        previousNewFiber = newFiber;
      }
      return resultingFirstChild;
    }

    // !3. 新老节点都还有节点，但是因为老fiber是链表，不方便快速get与delete，
    // !   因此把老fiber链表中的节点放入Map中，后续操作这个Map的get与delete

    // !4. 如果是组件更新阶段，此时新节点已经遍历完了，能复用的老节点都用完了，
    // ! 则最后查找Map里是否还有元素，如果有，则证明是新节点里不能复用的，也就是要
    return resultingFirstChild;
  }
```

updateSlot

TypeScript

```
function updateSlot(
  returnFiber: Fiber,
  oldFiber: Fiber | null,
  newChild: any
) {
  // 判断节点是否可以复用
  const key = oldFiber !== null ? oldFiber.key : null;
  if (isText(newChild)) {
    if (key !== null) {
      // 新节点是文本，老节点不是文本
      return null;
    }
    // 有可能可以复用
    return updateTextNode(returnFiber, oldFiber, newChild + "");
  }

  if (typeof newChild === "object" && newChild !== null) {
    if (newChild.key === key) {
      return updateElement(returnFiber, oldFiber, newChild);
    } else {
      return null;
    }
  }
}
```

updateElement

TypeScript

```
function updateElement(
  returnFiber: Fiber,
  current: Fiber | null,
  element: ReactElement
) {
  const elementType = element.type;
```

```
  if (current !== null) {
    if (current.elementType === elementType) {
      // 类型相同
      const existing = useFiber(current, element.props);
      existing.return = returnFiber;
      return existing;
    }
  }

  const created = createFiberFromElement(element);
  created.return = returnFiber;
  return created;
}
```

updateTextNode

TypeScript

```
function updateTextNode(
  returnFiber: Fiber,
  current: Fiber | null,
  textContent: string
) {
  if (current === null || current.tag !== HostText) {
    // 老节点不是文本
    const created = createFiberFromText(textContent);
    created.return = returnFiber;
    return created;
  } else {
    // 老节点是文本
    const existing = useFiber(current, textContent);
    existing.return = returnFiber;
    return existing;
  }
}
```

placeChild

TypeScript

```
function placeChild(
  newFiber: Fiber,
  lastPlacedIndex: number, // 记录的是新fiber在老fiber上的位置
  newIndex: number
) {
  newFiber.index = newIndex;

  if (!shouldTrackSideEffects) {
    return lastPlacedIndex;
  }

  // 判断节点位置是否发生相对位置变化，是否需要移动
  const current = newFiber.alternate;
  if (current !== null) {
    const oldIndex = current.index;
    if (oldIndex < lastPlacedIndex) {
      // 0 1 2
      // 0 2 1
      // 节点需要移动位置
      newFiber.flags |= Placement;
      return lastPlacedIndex;
    } else {
      return oldIndex;
    }
  } else {
    // 节点是新增
    newFiber.flags |= Placement;
    return lastPlacedIndex;
  }
}
```

ReactChildFiber

packages/react-reconciler/src/ReactChildFiber.ts

TypeScript

```
import {
  createFiberFromElement,
  createFiberFromText,
  createWorkInProgress,
} from "./ReactFiber";
import type { Fiber } from "./ReactInternalTypes";
import { REACT_ELEMENT_TYPE } from "shared/ReactSymbols";
import { ChildDeletion, Placement } from "./ReactFiberFlags";
import type { ReactElement } from "shared/ReactTypes";
import { isArray } from "shared/utils";
import { HostText } from "./ReactWorkTags";

type ChildReconciler = (
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChild: any
) => Fiber | null;

export const reconcileChildFibers: ChildReconciler =
  createChildReconciler(true);
export const mountChildFibers: ChildReconciler = createChildReconciler

// wrapper function
// 协调子节点
function createChildReconciler(shouldTrackSideEffects: boolean) {
  function deleteChild(returnFiber: Fiber, childToDelete: Fiber) {
    if (!shouldTrackSideEffects) {
      return;
    }
    const deletions = returnFiber.deletions;
    if (deletions === null) {
      returnFiber.deletions = [childToDelete];
      returnFiber.flags |= ChildDeletion;
    } else {
      returnFiber.deletions!.push(childToDelete);
    }
  }

  function deleteRemainingChildren(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null
```

```
) {
  if (!shouldTrackSideEffects) {
    return;
  }

  let childToDelete = currentFirstChild;
  while (childToDelete !== null) {
    deleteChild(returnFiber, childToDelete);
    childToDelete = childToDelete.sibling;
  }

  return null;
}
// 给fiber节点添加flags
function placeSingleChild(newFiber: Fiber) {
  if (shouldTrackSideEffects && newFiber.alternate === null) {
    newFiber.flags |= Placement;
  }
  return newFiber;
}

// 文本
function reconcileSingleTextNode(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null, // todo 更新
  textContent: string
) {
  const created = createFiberFromText(textContent);
  created.return = returnFiber;
  return created;
}

function useFiber(fiber: Fiber, pendingProps: any) {
  const clone = createWorkInProgress(fiber, pendingProps);
  clone.index = 0;
  clone.sibling = null;
  return clone;
}
// 协调单个节点，对于页面初次渲染，创建fiber，不涉及对比复用老节点
// new (1)
// old 2 [1] 3 4
```

```
function reconcileSingleElement(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  element: ReactElement
) {
  // 节点复用的条件
  // ! 1. 同一层级下 2. key相同 3. 类型相同
  const key = element.key;
  let child = currentFirstChild;

  while (child !== null) {
    if (child.key === key) {
      const elementType = element.type;
      if (child.elementType === elementType) {
        // todo 后面其它fiber可以删除了
        const existing = useFiber(child, element.props);
        existing.return = returnFiber;
        return existing;
      } else {
        // 前提：React不认为同一层级下有两个相同的key值
        deleteRemainingChildren(returnFiber, child);
        break;
      }
    } else {
      // 删除单个节点
      deleteChild(returnFiber, child);
    }
    // 老fiber节点是单链表
    child = child.sibling;
  }

  let createdFiber = createFiberFromElement(element);
  createdFiber.return = returnFiber;
  return createdFiber;
}

function createChild(returnFiber: Fiber, newChild: any): Fiber | nul
  if (isText(newChild)) {
    const created = createFiberFromText(newChild + "");
    created.return = returnFiber;
    return created;
```

```
  }

  if (typeof newChild === "object" && newChild !== null) {
    switch (newChild.$$typeof) {
      case REACT_ELEMENT_TYPE: {
        const created = createFiberFromElement(newChild);
        created.return = returnFiber;
        return created;
      }
    }
  }

  return null;
}

function updateTextNode(
  returnFiber: Fiber,
  current: Fiber | null,
  textContent: string
) {
  if (current === null || current.tag !== HostText) {
    // 老节点不是文本
    const created = createFiberFromText(textContent);
    created.return = returnFiber;
    return created;
  } else {
    // 老节点是文本
    const existing = useFiber(current, textContent);
    existing.return = returnFiber;
    return existing;
  }
}

function updateElement(
  returnFiber: Fiber,
  current: Fiber | null,
  element: ReactElement
) {
  const elementType = element.type;
  if (current !== null) {
    if (current.elementType === elementType) {
```

```
      // 类型相同
      const existing = useFiber(current, element.props);
      existing.return = returnFiber;
      return existing;
    }
  }

  const created = createFiberFromElement(element);
  created.return = returnFiber;
  return created;
}
function updateSlot(
  returnFiber: Fiber,
  oldFiber: Fiber | null,
  newChild: any
) {
  // 判断节点是否可以复用
  const key = oldFiber !== null ? oldFiber.key : null;
  if (isText(newChild)) {
    if (key !== null) {
      // 新节点是文本，老节点不是文本
      return null;
    }
    // 有可能可以复用
    return updateTextNode(returnFiber, oldFiber, newChild + "");
  }

  if (typeof newChild === "object" && newChild !== null) {
    if (newChild.key === key) {
      return updateElement(returnFiber, oldFiber, newChild);
    } else {
      return null;
    }
  }
}

function placeChild(
  newFiber: Fiber,
  lastPlacedIndex: number, // 记录的是新fiber在老fiber上的位置
  newIndex: number
) {
```

```
    newFiber.index = newIndex;

    if (!shouldTrackSideEffects) {
      return lastPlacedIndex;
    }

    // 判断节点位置是否发生相对位置变化，是否需要移动
    const current = newFiber.alternate;
    if (current !== null) {
      const oldIndex = current.index;
      if (oldIndex < lastPlacedIndex) {
        // 0 1 2
        // 0 2 1
        // 节点需要移动位置
        newFiber.flags |= Placement;
        return lastPlacedIndex;
      } else {
        return oldIndex;
      }
    } else {
      // 节点是新增
      newFiber.flags |= Placement;
      return lastPlacedIndex;
    }
  }
  function reconcileChildrenArray(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
    newChildren: Array<any>
  ) {
    let resultFirstChild: Fiber | null = null; // 头结点
    let previousNewFiber: Fiber | null = null;
    let oldFiber = currentFirstChild;
    let nextOldFiber = null; // oldFiber.sibling
    let newIdx = 0;
    let lastPlacedIndex = 0;

    // * 大多数实际场景下，节点相对位置不变
    // old 0 1 2 3 4
    // new 0 1 2 3
```

```
    // new 0 2 1 3

    // 0 2 newIndex(1)<2

    // new 3 2 0 4 1
    // ! 1. 从左往右遍历，按位置比较，如果可以复用，那就复用。不能复用，退出本轮
    for (; oldFiber !== null && newIdx < newChildren.length; newIdx++)
      if (oldFiber.index > newIdx) {
        nextOldFiber = oldFiber;
        oldFiber = null;
      } else {
        nextOldFiber = oldFiber.sibling;
      }
      const newFiber = updateSlot(returnFiber, oldFiber, newChildren[n
      if (newFiber === null) {
        if (oldFiber === null) {
          oldFiber = nextOldFiber;
        }
        break;
      }

      if (shouldTrackSideEffects) {
        if (oldFiber && newFiber?.alternate === null) {
          deleteChild(returnFiber, oldFiber);
        }
      }

      // 判断节点在DOM的相对位置是否发生变化
      //   组件更新阶段，判断在更新前后的位置是否一致，如果不一致，需要移动
      lastPlacedIndex = placeChild(newFiber as Fiber, lastPlacedIndex,

      if (previousNewFiber === null) {
        // 第一个节点，不要用newIdx判断，因为有可能有null，而null不是有效fiber
        resultFirstChild = newFiber as Fiber;
      } else {
        previousNewFiber.sibling = newFiber as Fiber;
      }
      previousNewFiber = newFiber as Fiber;

      oldFiber = nextOldFiber;
    }
```

```
  // * Vue 1.2 从右往左遍历，按位置比较，如果可以复用，那就复用。不能复用，退出

  // ! 2.1 老节点还有，新节点没了。删除剩余的老节点
  if (newIdx === newChildren.length) {
    deleteRemainingChildren(returnFiber, oldFiber);
    return resultFirstChild;
  }

  // ! 2.2 新节点还有，老节点没了。剩下的新节点新增就可以了
  // 包括页面初次渲染
  if (oldFiber === null) {
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(returnFiber, newChildren[newIdx])
      if (newFiber === null) {
        continue;
      }

      //  组件更新阶段，判断在更新前后的位置是否一致，如果不一致，需要移动
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx
      if (previousNewFiber === null) {
        // 第一个节点，不要用newIdx判断，因为有可能有null，而null不是有效fib
        resultFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultFirstChild;
  }

  // !3. 新老节点都还有
  // todo
  return resultFirstChild;
}
function reconcileChildFibers(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChild: any
) {
  if (isText(newChild)) {
```

```
      return placeSingleChild(
        reconcileSingleTextNode(returnFiber, currentFirstChild, newChi
      );
    }

    // 检查newChild类型，单个节点、文本、数组
    if (typeof newChild === "object" && newChild !== null) {
      switch (newChild.$$typeof) {
        case REACT_ELEMENT_TYPE: {
          return placeSingleChild(
            reconcileSingleElement(returnFiber, currentFirstChild, new
          );
        }
      }
    }

    // 子节点是数组
    if (isArray(newChild)) {
      return reconcileChildrenArray(returnFiber, currentFirstChild, ne
    }
    // todo
    return null;
  }

  return reconcileChildFibers;
}

function isText(newChild: any) {
  return (
    (typeof newChild === "string" && newChild !== "") ||
    typeof newChild === "number"
  );
}
```