16-3 分析Provider与后代组件消费context value

本章节源码如无特别标记，路径均是来自 react-reconciler/src/ReactFiberNewContext.js

render 阶段

beginWork

pushProvider

TypeScript

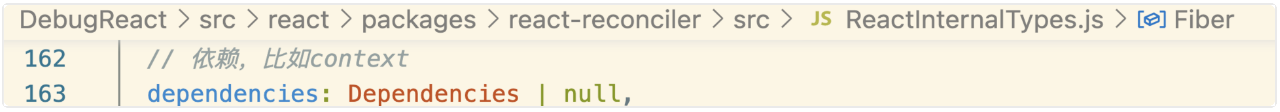
```
export function pushProvider<T>(
  providerFiber: Fiber,
  context: ReactContext<T>,
  nextValue: T,
): void {
    push(valueCursor, context._currentValue, providerFiber);
    context._currentValue = nextValue;
}
```

propagateContextChange

```JavaScript
export function propagateContextChange<T>(
  workInProgress: Fiber,
  context: ReactContext<T>,
  renderLanes: Lanes,
): void {
  propagateContextChange_eager(workInProgress, context, renderLanes);
}

function propagateContextChange_eager<T>(
  workInProgress: Fiber,
  context: ReactContext<T>,
  renderLanes: Lanes,
): void {

  let fiber = workInProgress.child;
  if (fiber !== null) {
    // Set the return pointer of the child to the work-in-progress fib
    fiber.return = workInProgress;
  }
  while (fiber !== null) {
    let nextFiber;

    // 深度优先遍历
    const list = fiber.dependencies; // fiber消费的context单链表
    if (list !== null) {
      nextFiber = fiber.child;

      let dependency = list.firstContext;
      // 如果有context消费，那么接下来遍历context单链表，找到匹配的context，然
      while (dependency !== null) {
        if (dependency.context === context) {
          // 找到匹配的context，那么调度更新。
          if (fiber.tag === ClassComponent) {
            // 如果是类组件，那么按照类组件的更新逻辑。因为类组件的更新逻辑与其它
            const lane = pickArbitraryLane(renderLanes);
            const update = createUpdate(lane);
            update.tag = ForceUpdate;


            const updateQueue = fiber.updateQueue;
            if (updateQueue === null) {
              // Only occurs if the fiber has been unmounted.
            } else {
              const sharedQueue: SharedQueue<any> = (updateQueue: any)
              const pending = sharedQueue.pending;
              if (pending === null) {
                // This is the first update. Create a circular list.
                update.next = update;
              } else {
                update.next = pending.next;
                pending.next = update;
              }
              sharedQueue.pending = update;
            }
          }

          fiber.lanes = mergeLanes(fiber.lanes, renderLanes);
          const alternate = fiber.alternate;
          if (alternate !== null) {
            alternate.lanes = mergeLanes(alternate.lanes, renderLanes)
          }
          // 更新所有祖先的childLanes
          scheduleContextWorkOnParentPath(
            fiber.return,
            renderLanes,
            workInProgress,
          );

          // Mark the updated lanes on the list, too.
          list.lanes = mergeLanes(list.lanes, renderLanes);

          // 找到一个匹配的context，就可以停止遍历context单链表。
          break;
        }
        dependency = dependency.next;
      }
    } else if (fiber.tag === ContextProvider) {
      // 如果这一个一个匹配的provider，那么不需要继续遍历。因为Provider会再次走
      nextFiber = fiber.type === workInProgress.type ? null : fiber.ch
    } else if (fiber.tag === DehydratedFragment) {
      // If a dehydrated suspense boundary is in this subtree, we don'
      // if it will have any context consumers in it. The best we can
      // mark it as having updates.
      const parentSuspense = fiber.return;

      if (parentSuspense === null) {
        throw new Error(
          'We just came from a parent so we must have had a parent. Th
        );
      }

      parentSuspense.lanes = mergeLanes(parentSuspense.lanes, renderLa
      const alternate = parentSuspense.alternate;
      if (alternate !== null) {
        alternate.lanes = mergeLanes(alternate.lanes, renderLanes);
      }

      scheduleContextWorkOnParentPath(
        parentSuspense,
        renderLanes,
        workInProgress,
      );
      nextFiber = fiber.sibling;
    } else {
      // 继续向下遍历
      nextFiber = fiber.child;
    }

    if (nextFiber !== null) {
      // Set the return pointer of the child to the work-in-progress f
      nextFiber.return = fiber;
    } else {
      // No child. Traverse to next sibling.
      // 深度优先遍历，没有子节点，继续遍历兄弟节点
      nextFiber = fiber;
      while (nextFiber !== null) {
        if (nextFiber === workInProgress) {
          // 如果遍历到了本subtree的root节点，退出
          nextFiber = null;
          break;
        }
        const sibling = nextFiber.sibling;
        if (sibling !== null) {
          // 如果找到兄弟节点，退出
          sibling.return = nextFiber.return;
          nextFiber = sibling;
          break;
        }
        // 没有兄弟节点，接下来遍历父节点的兄弟节点
        nextFiber = nextFiber.return;
      }
    }
    fiber = nextFiber;
  }
}
```

后代组件消费 context value

后代组件消费 context，这些 context 会存储在 `fiber.dependencies` 属性上。单链表结构。


DebugReact > src > react > packages > react-reconciler > src > JS ReactInternalTypes.js > Fiber

```
// 依赖，比如context
dependencies: Dependencies | null,
```

react/packages/react-reconciler/src/ReactInternalTypes.js

```TypeScript
export type ContextDependency<T> = {
  context: ReactContext<T>,
  next: ContextDependency<mixed> | null,
  memoizedValue: T,
  ...
};

export type Dependencies = {
  lanes: Lanes,
  firstContext: ContextDependency<mixed> | null,
  ...
};
```

准备读取 context value: prepareToReadContext

JavaScript

```javascript
let currentlyRenderingFiber: Fiber | null = null;
let lastContextDependency: ContextDependency<mixed> | null = null;
let lastFullyObservedContext: ReactContext<any> | null = null;

export function prepareToReadContext(
  workInProgress: Fiber,
  renderLanes: Lanes,
): void {
  currentlyRenderingFiber = workInProgress;
  lastContextDependency = null;
  lastFullyObservedContext = null;

  const dependencies = workInProgress.dependencies;
  if (dependencies !== null) {
    if (enableLazyContextPropagation) {
      // Reset the work-in-progress list
      dependencies.firstContext = null;
    } else {
      const firstContext = dependencies.firstContext;
      if (firstContext !== null) {
        if (includesSomeLane(dependencies.lanes, renderLanes)) {
          // Context list has a pending update. Mark that this fiber p
          markWorkInProgressReceivedUpdate();
        }
        // Reset the work-in-progress list
        dependencies.firstContext = null;
      }
    }
  }
}
```

readContext: 读取 context value

```javascript
function readContextForConsumer<T>(
  consumer: Fiber | null,
  context: ReactContext<T>,
): T {
  const value = context._currentValue

  if (lastFullyObservedContext === context) {
    // Nothing to do. We already observe everything in this context.
  } else {
    const contextItem = {
      context: ((context: any): ReactContext<mixed>),
      memoizedValue: value,
      next: null,
    };

    if (lastContextDependency === null) {
      if (consumer === null) {
        throw new Error(
          'Context can only be read while React is rendering. ' +
            'In classes, you can read it in the render method or getDe
            'In function components, you can read it directly in the f
            'inside Hooks like useReducer() or useMemo().',
        );
      }

      // This is the first dependency for this component. Create a new
      lastContextDependency = contextItem;
      consumer.dependencies = {
        lanes: NoLanes,
        firstContext: contextItem,
      };
      if (enableLazyContextPropagation) {
        consumer.flags |= NeedsPropagation;
      }
    } else {
      // Append a new context item.
      lastContextDependency = lastContextDependency.next = contextItem
    }
  }
  return value;
}
```

completeWork

popProvider

JavaScript

```javascript
export function popProvider(
  context: ReactContext<any>,
  providerFiber: Fiber,
): void {
  const currentValue = valueCursor.current;
  context._currentValue = currentValue;
  pop(valueCursor, providerFiber);
}
```

render 阶段最后

重置:

TypeScript

```typescript
export function resetContextDependencies(): void {
  currentlyRenderingFiber = null;
  lastContextDependency = null;
  lastFullyObservedContext = null;
}
```