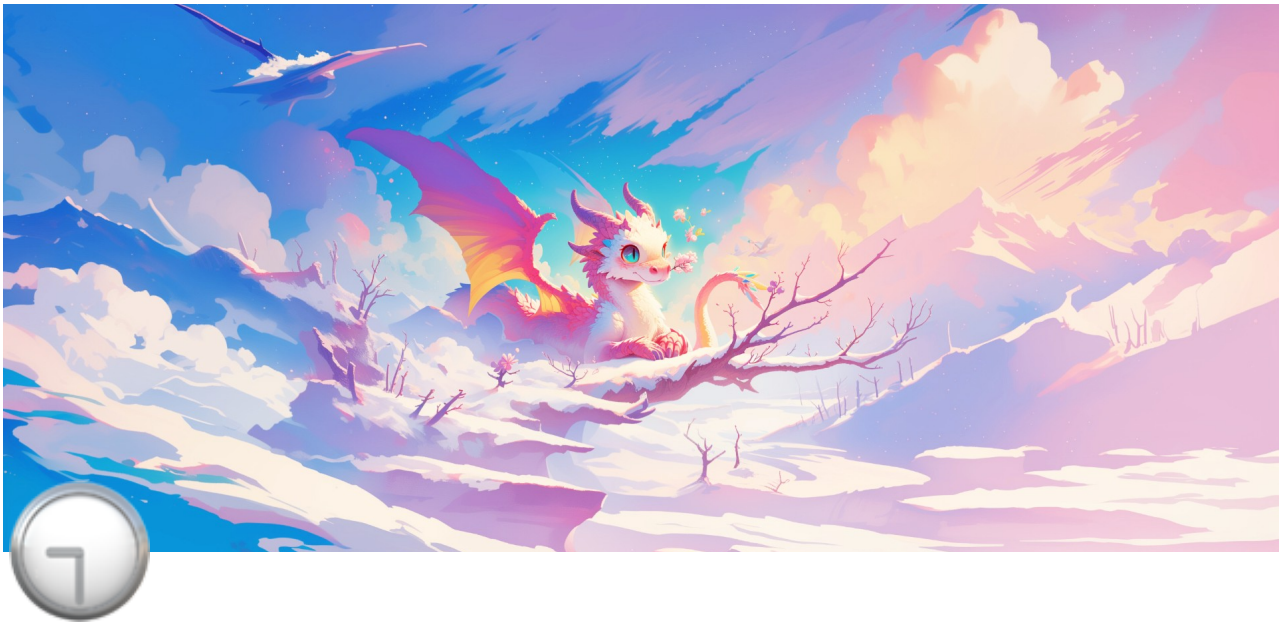

20-3 实现 lanes 模型

React18 把 update 分成两种：

- **Urgent updates** 紧急更新（普通更新），指直接交互，通常指的用户交互。如点击、输入等。这种更新一旦不及时，用户就会觉得哪里不对。
- **Transition updates** 过渡更新（非紧急更新），如 UI 从一个视图向另一个视图的更新。通常这种更新用户并不着急看到。

给 fiber 添加 lanes

packages/react-reconciler/src/ReactInternalTypes.ts

```
@@ -1,4 +1,5 @@
import type { Flags } from "./ReactFiberFlags";
+ import { LaneMap, Lanes, NoLanes } from "./ReactFiberLane";
import type { WorkTag } from "./ReactWorkTags";

export type Fiber = {
@@ -51,6 +52,9 @@ export type Fiber = {

  // 记录effect
  updateQueue: any;
+
+   lanes: Lanes;
+   childLanes: Lanes;
};

export type Container = Element | Document | DocumentFragment;
@@ -60,4 +64,5 @@ export type FiberRoot = {
  current: Fiber;
  // 一个准备提交 work-in-progress,  HostRoot
  finishedWork: Fiber | null;
+   pendingLanes: Lanes;
};
```

packages/react-reconciler/src/ReactFiber.ts

```
@@ -19,6 +19,7 @@ import {
  REACT_MEMO_TYPE,
  REACT_PROVIDER_TYPE,
} from "shared/ReactSymbols";
+ import { NoLanes } from "./ReactFiberLane";

// 创建一个fiber
export function createFiber(
@@ -69,6 +70,9 @@ function FiberNode(tag: WorkTag, pendingProps: any, key: null | string) {
  this.deletions = null;

  this.updateQueue = null;
+
+  this.lanes = NoLanes;
+  this.childLanes = NoLanes;
}

// 根据 ReactElement 创建Fiber
@@ -130,6 +134,8 @@ export function createWorkInProgress(current: Fiber, pendingProps: any): Fiber {
  }

  workInProgress.flags = current.flags;
+  workInProgress.childLanes = current.childLanes;
+  workInProgress.lanes = current.lanes;

  workInProgress.child = current.child;
  workInProgress.memoizedProps = current.memoizedProps;
```

FiberRootNode

packages/react-reconciler/src/ReactFiberRoot.ts

TypeScript

```typescript
export function FiberRootNode(containerInfo: Container) {
  this.containerInfo = containerInfo;
  this.current = null;
  this.finishedWork = null;
  this.pendingLanes = NoLanes;
}
```

优先级

1. 获取 update lane，即 requestUpdateLane

获取紧急 update 的 lane。

packages/react-reconciler/src/ReactFiberWorkLoop.ts

```TypeScript
export function requestUpdateLane(): Lane {
  const updateLane: Lane = getCurrentUpdatePriority();
  if (updateLane !== NoLane) {
    return updateLane;
  }
  const eventLane: Lane = getCurrentEventPriority();
  return eventLane;
}
```

getCurrentUpdatePriority

packages/react-dom-bindings/src/client/ReactFiberConfigDOM.ts

```TypeScript
import {
  DefaultEventPriority,
  EventPriority,
} from "react-reconciler/src/ReactEventPriorities";
import { getEventPriority } from "../events/ReactDOMEventListener";

export function getCurrentEventPriority(): EventPriority {
  const currentEvent = window.event;
  if (currentEvent === undefined) {
    // ? sy 页面初次渲染
    return DefaultEventPriority;
  }

  return getEventPriority(currentEvent.type as any);
}
```

lanesToEventPriority

packages/react-reconciler/src/ReactEventPriorities.ts

```TypeScript
```

```
export function lanesToEventPriority(lanes: Lanes): EventPriority {
  // 根据优先级最高的lane，返回对应的 EventPriority。这里对应Scheduler包中的优
  const lane = getHighestPriorityLane(lanes);
  if (!isHigherEventPriority(DiscreteEventPriority, lane)) {
    return DiscreteEventPriority;
  }
  if (!isHigherEventPriority(ContinuousEventPriority, lane)) {
    return ContinuousEventPriority;
  }
  if (includesNonIdleWork(lane)) {
    return DefaultEventPriority; // 2
  }
  return IdleEventPriority;
}
```

2. 获取 DeferredLane，即 requestDeferredLane

packages/react-reconciler/src/ReactFiberWorkLoop.ts

TypeScript

```
let workInProgressDeferredLane: Lane = NoLane;

export function requestDeferredLane(): Lane {
  if (workInProgressDeferredLane === NoLane) {
    workInProgressDeferredLane = claimNextTransitionLane();
  }

  return workInProgressDeferredLane;
}
```

claimNextTransitionLane

packages/react-reconciler/src/ReactFiberLane.ts

TypeScript

```
let nextTransitionLane: Lane = TransitionLane1;

export function claimNextTransitionLane(): Lane {
  // 循环遍历lanes，将每个新的transition分配到下一个lane。
  // 在大多数情况下，这意味着每个transition都有自己的lane，直到我们用完所有lane
```

```
  const lane = nextTransitionLane;
  nextTransitionLane <<= 1;
  if ((nextTransitionLane & TransitionLanes) === NoLanes) {
    nextTransitionLane = TransitionLane1;
  }
  return lane;
}
```

getNextLanes

packages/react-reconciler/src/ReactFiberLane.ts

TypeScript

```
export function getNextLanes(root: FiberRoot, wipLanes: Lanes): Lanes
  const pendingLanes = root.pendingLanes;
  if (pendingLanes === NoLanes) {
    return NoLanes;
  }

  let nextLanes = getHighestPriorityLanes(pendingLanes);

  if (nextLanes === NoLanes) {
    return NoLanes;
  }
  // 如果我们已经在render阶段中，切换lanes会中断当前渲染进程，导致丢失进度。
  // 只有当新lanes的优先级更高时，我们才应该这样做。
  if (wipLanes !== NoLanes && wipLanes !== nextLanes) {
    const nextLane = getHighestPriorityLane(nextLanes);
    const wipLane = getHighestPriorityLane(wipLanes);
    if (
      nextLane >= wipLane ||
      // Default priority updates不应中断transition。default updates和tr
      (nextLane === DefaultLane && (wipLane & TransitionLanes) !== NoL
    ) {
      // 继续完成正在进行中的树。不中断。
      return wipLanes;
    }
  }
```

```
  return nextLanes;
}
```

实现 useDeferredValue

TypeScript

```
// These are set right before calling the component.
let renderLanes: Lanes = NoLanes;

export function renderWithHooks<Props>(
  current: Fiber | null,
  workInProgress: Fiber,
  Component: any,
  props: Props,
  nextRenderLanes: Lanes
): any {
  renderLanes = nextRenderLanes;

  currentlyRenderingFiber = workInProgress;
  workInProgress.memoizedState = null;
  workInProgress.updateQueue = null;

  let children = Component(props);

  finishRenderingHooks();

  return children;
}

export function useDeferredValue<T>(value: T): T {
  const hook = updateWorkInProgressHook();

  const prevValue: T = hook.memoizedState;

  if (currentHook !== null) {
    // 更新阶段
    if (Object.is(value, prevValue)) {
      // 传入的值与当前渲染的值是相同的，因此我们可以快速bail out
      return value;
```

```
    } else {
      // 收到一个与当前数值不同的新值
      // 不是 只包括非紧急更新
      const shouldDeferValue = !includesOnlyNonUrgentLanes(renderLanes
      if (shouldDeferValue) {
        // sy-input
        // 这是一个紧急更新。由于数值已更改，可以继续使用先前的数值，并生成一个延迟
        // 调度一个延迟渲染。
        const deferredLane = requestDeferredLane();
        currentlyRenderingFiber!.lanes = mergeLanes(
          currentlyRenderingFiber!.lanes, // 0
          deferredLane // 128
        );

        // 复用先前的数值。我们不需要将其标记一个update，因为我们没有渲染新值。
        return prevValue;
      } else {
        // 只包括非紧急更新。没有其他紧急的更新，那么这个时候执行这个非紧急更新就
        hook.memoizedState = value;
        return value;
      }
    }
  }
  hook.memoizedState = value;

  return value;
}
```