# 19-3 React Lanes 模型常用工具函数

## 基本运算函数

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

JavaScript

```
// 是否是transitionLanes
export function isTransitionLane(lane: Lane): boolean {
  return (lane & TransitionLanes) !== NoLanes;
}

// 获取优先级最高的lane
// 因为在 lane 的值中，值越小，代表的优先级越高。即
// 获取最低位的1，如4194240&-4194240就是64
// 负数原码转换为补码的方法：符号位保持1不变，数值位按位求反，末位加1
export function getHighestPriorityLane(lanes: Lanes): Lane {
  return lanes & -lanes;
}
// 是否包含某个lane或者某些lane(lanes)
export function includesSomeLane(a: Lanes | Lane, b: Lanes | Lane): bo
  return (a & b) !== NoLanes;
}

// set是否包含subset，和 includesSomeLane 不同，includesSomeLane检查的是a和
```

```
// 而这里的 isSubsetOfLanes 检查的是 subset 是否是 set 的子集
export function isSubsetOfLanes(set: Lanes, subset: Lanes | Lane): boo
  return (set & subset) === subset;
}

// 合并两个lane或者lanes
export function mergeLanes(a: Lanes | Lane, b: Lanes | Lane): Lanes {
  return a | b;
}
// 移除某个lane或者lanes，比如执行完节点的 Update 操作之后，则需要移动 fiber.f
export function removeLanes(set: Lanes, subset: Lanes | Lane): Lanes {
  return set & ~subset;
}

// 与 includesSomeLane 不同，includesSomeLane返回的是是否有交叉，即结果是boo
// intersectLanes 返回交叉的lanes
export function intersectLanes(a: Lanes | Lane, b: Lanes | Lane): Lane
  return a & b;
}

// 返回优先级较高的lane。如果 a < b, 则说明a的优先级高于b，因为lane越小，优先级
export function higherPriorityLane(a: Lane, b: Lane): Lane {
  return a !== NoLane && a < b ? a : b;
}

// 其实就是返回比特位上最右边1的位置下标，(从左边往右边数，从0开始)
// 这个常函数用在获取一个lanes上优先级最高的lane，如这里返回值为index，那么1<<ir
function pickArbitraryLaneIndex(lanes: Lanes) {
  return 31 - clz32(lanes);
}

// 返回比特位上最右边1的位置下标，(从左边往右边数，从0开始)
function laneToIndex(lane: Lane) {
  return pickArbitraryLaneIndex(lane);
}
```

# 检查饿死现象

## FiberRoot

`FiberRoot` 上有个属性值 `expirationTimes`，用于记录的过期时间，其值类型定义：

```javascript
export type LaneMap<T> = Array<T>;

expirationTimes: LaneMap<number>, 。
```

初始化的时候：

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRoot.js

```javascript
function FiberRootNode(
  this: $FlowFixMe,
  containerInfo: any,
  // $FlowFixMe[missing-local-annot]
  tag,
  hydrate: any,
  identifierPrefix: any,
  onRecoverableError: any,
  formState: ReactFormState<any, any> | null,
) {
  this.tag = tag;
  this.containerInfo = containerInfo;
  this.pendingChildren = null;
  this.current = null;
  this.pingCache = null;
  this.finishedWork = null;
  this.timeoutHandle = noTimeout;
  this.cancelPendingCommit = null;
  this.context = null;
  this.pendingContext = null;
  this.next = null;
  this.callbackNode = null;
  this.callbackPriority = NoLane;
  this.expirationTimes = createLaneMap(NoTimestamp);
```

初始化函数：

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

```javascript
```

```
export const NoTimestamp = -1;

export function createLaneMap<T>(initial: T): LaneMap<T> {
  // Intentionally pushing one by one.
  // https://v8.dev/blog/elements-kinds#avoid-creating-holes
  const laneMap = [];
  for (let i = 0; i < TotalLanes; i++) {
    laneMap.push(initial);
  }
  return laneMap;
}
```

## markStarvedLanesAsExpired

这个函数在调度更新的过程中会被调用。

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

```
DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberRootScheduler.js > scheduleTaskForRootDuringMicrotask
322  function scheduleTaskForRootDuringMicrotask(
323    root: FiberRoot,
324    currentTime: number,
325  ): Lane {
326    // 这个函数总是在microtask中被调用，或者在渲染任务的最后，在我们将控制权交还给主线程之前被调用。它绝对不应该被同步调用
327    // This function is always called inside a microtask, or at the very end of a
328    // rendering task right before we yield to the main thread. It should never be
329    // called synchronously.
330    //
331    // TODO: Unless enableDeferRootSchedulingToMicrotask is off. We need to land
332    // that ASAP to unblock additional features we have planned.
333    //
334    // This function also never performs React work synchronously; it should
335    // only schedule work to be performed later, in a separate task or microtask.
336
337    // Check if any lanes are being starved by other work. If so, mark them as
338    // expired so we know to work on those next.
339    // 检查是否有任何lanes被其他work饿死。如果是，将它们标记为过期，这样我们就知道下一个要处理的是它们。      You, 28分
340    markStarvedLanesAsExpired(root, currentTime);
```

用处就是检查是否有 lanes 挨饿，如果有，则标记他们过期，以便下次执行。

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

```javascript
export function markStarvedLanesAsExpired(
  root: FiberRoot,
  currentTime: number,
): void {
  const pendingLanes = root.pendingLanes;
  const suspendedLanes = root.suspendedLanes;
```

```
  const pingedLanes = root.pingedLanes;
  const expirationTimes = root.expirationTimes;

  // 遍历pending lanes，并检查是否已经达到它们的过期时间。
  // 如果是，我们就认为这个update挨饿了，并将其标记为已过期，以强制其完成。
  let lanes = enableRetryLaneExpiration
    ? pendingLanes //  ? sy
    : pendingLanes & ~RetryLanes;

  while (lanes > 0) {
    // 下面两行代码的作用是找到lanes中最低位的1，即优先级最
    const index = pickArbitraryLaneIndex(lanes);
    // 把1左移index位，即得到一个只有第index位为1的子掩码
    const lane = 1 << index;

    const expirationTime = expirationTimes[index];
    if (expirationTime === NoTimestamp) {
      // 如果这个 pending lane 没有过期时间
      // 如果它没有被挂起且需要更新，我们就认为它是CPU密集型操作。
      // 用当前时间计算出一个新的过期时间给它。
      // CPU bound  / IO Bound
      if (
        (lane & suspendedLanes) === NoLanes ||
        (lane & pingedLanes) !== NoLanes
      ) {
        // 假设timestamps(时间戳)是单调递增的
        expirationTimes[index] = computeExpirationTime(lane, currentTi
      }
    } else if (expirationTime <= currentTime) {
      // 这个 pending lane 已经过期了
      root.expiredLanes |= lane;
    }
    // 把lane从lanes中移除，计算下一个lane
    lanes &= ~lane;
  }
}
```

## 通过 Lane 计算过期时间

策略和 scheduler 相同，根据优先级判断过期时间，优先级越大，值越小。

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

```javascript
function computeExpirationTime(lane: Lane, currentTime: number) {
  switch (lane) {
    case SyncHydrationLane:
    case SyncLane:
    case InputContinuousHydrationLane:
    case InputContinuousLane:
      // 交互行为应该早点被执行，因此过期时间会比较小
      return currentTime + syncLaneExpirationMs; // + 250;
    case DefaultHydrationLane:
    case DefaultLane:
    case TransitionHydrationLane:
    case TransitionLane1:
    case TransitionLane2:
    case TransitionLane3:
    case TransitionLane4:
    case TransitionLane5:
    case TransitionLane6:
    case TransitionLane7:
    case TransitionLane8:
    case TransitionLane9:
    case TransitionLane10:
    case TransitionLane11:
    case TransitionLane12:
    case TransitionLane13:
    case TransitionLane14:
    case TransitionLane15:
      return currentTime + transitionLaneExpirationMs; // + 5000
    case RetryLane1:
    case RetryLane2:
    case RetryLane3:
    case RetryLane4:
      return enableRetryLaneExpiration
        ? currentTime + retryLaneExpirationMs // + 5000
        : NoTimestamp;
    case SelectiveHydrationLane:
    case IdleHydrationLane:
    case IdleLane:
```

```
    case OffscreenLane:
    case DeferredLane:
      // 任何空闲优先级或更低的任务都不应该过期。
      return NoTimestamp;
    default:
      if (__DEV__) {
        console.error(
          'Should have found matching lanes. This is a bug in React.',
        );
      }
      return NoTimestamp;
  }
}
```

## getNextLanes

// 获取下一个 pending lanes

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

JavaScript

```
export function getNextLanes(root: FiberRoot, wipLanes: Lanes): Lanes
  // Early bailout if there's no pending work left.
  // 没有pending lanes，直接返回。后续进入bailout
  const pendingLanes = root.pendingLanes;
  if (pendingLanes === NoLanes) {
    return NoLanes;
  }

  let nextLanes = NoLanes;

  const suspendedLanes = root.suspendedLanes;
  const pingedLanes = root.pingedLanes;

  // 在所有 non-idle work 完成之前，不要处理任何idle work，即使这个work是susp
  const nonIdlePendingLanes = pendingLanes & NonIdleLanes;
  if (nonIdlePendingLanes !== NoLanes) {
    // 有非空闲的pending lanes， 注意不处理suspendedLanes
```

```
    const nonIdleUnblockedLanes = nonIdlePendingLanes & ~suspendedLane
    if (nonIdleUnblockedLanes !== NoLanes) {
      // 获取非idle非suspended的lanes的最高优先级，返回值是个lanes，也包括单个
      nextLanes = getHighestPriorityLanes(nonIdleUnblockedLanes);
    } else {
      const nonIdlePingedLanes = nonIdlePendingLanes & pingedLanes;
      if (nonIdlePingedLanes !== NoLanes) {
        nextLanes = getHighestPriorityLanes(nonIdlePingedLanes);
      }
    }
  } else {
    // 只剩下了idle work
    const unblockedLanes = pendingLanes & ~suspendedLanes;
    if (unblockedLanes !== NoLanes) {
      nextLanes = getHighestPriorityLanes(unblockedLanes);
    } else {
      if (pingedLanes !== NoLanes) {
        nextLanes = getHighestPriorityLanes(pingedLanes);
      }
    }
  }

  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 &&
    // 如果我们已经延迟暂停了，那么中断是可以的。不必等到root完成。
    (wipLanes & suspendedLanes) === NoLanes
  ) {
    const nextLane = getHighestPriorityLane(nextLanes);
    const wipLane = getHighestPriorityLane(wipLanes);
    if (
      // Tests whether the next lane is equal or lower priority than t
      // one. This works because the bits decrease in priority as you
      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;
}
```

## getHighestPriorityLanes

获取优先级最高的 lanes。其实也包括单个 lane，只是从类型上来说，lane 也算 lanes。

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

```javascript
function getHighestPriorityLanes(lanes: Lanes | Lane): Lanes {
  if (enableUnifiedSyncLane) {

    const pendingSyncLanes = lanes & SyncUpdateLanes;
    if (pendingSyncLanes !== 0) {
      // ? sy
      // 将DefaultLane、SyncLane和ContinuousLane统一为SyncLane，并在根上使
      // https://github.com/facebook/react/pull/25524
      return pendingSyncLanes;
    }
  }
  switch (getHighestPriorityLane(lanes)) {
    case SyncHydrationLane:
      return SyncHydrationLane;
    case SyncLane:
      return SyncLane;
    case InputContinuousHydrationLane:
      return InputContinuousHydrationLane;
    case InputContinuousLane:
      return InputContinuousLane;
    case DefaultHydrationLane:
      return DefaultHydrationLane;
    case DefaultLane:
```

```
      return DefaultLane;
    case TransitionHydrationLane:
      return TransitionHydrationLane;
    case TransitionLane1:
    case TransitionLane2:
    case TransitionLane3:
    case TransitionLane4:
    case TransitionLane5:
    case TransitionLane6:
    case TransitionLane7:
    case TransitionLane8:
    case TransitionLane9:
    case TransitionLane10:
    case TransitionLane11:
    case TransitionLane12:
    case TransitionLane13:
    case TransitionLane14:
    case TransitionLane15:
      return lanes & TransitionLanes;
    case RetryLane1:
    case RetryLane2:
    case RetryLane3:
    case RetryLane4:
      return lanes & RetryLanes;
    case SelectiveHydrationLane:
      return SelectiveHydrationLane;
    case IdleHydrationLane:
      return IdleHydrationLane;
    case IdleLane:
      return IdleLane;
    case OffscreenLane:
      return OffscreenLane;
    case DeferredLane:
      // This shouldn't be reachable because deferred work is always e
      // with something else.
      return NoLanes;
    default:
      if (__DEV__) {
        console.error(
          'Should have found matching lanes. This is a bug in React.',
        );
```

```
    }
    // This shouldn't be reachable, but as a fallback, return the en
    return lanes;
  }
}
```