React源码分析8-状态更新的优先级机制

为什么需要优先级

优先级机制最终目的是为了实现**高优先级任务优先执行，低优先级任务延后执行。**

实现这一目的的本质就是在低优先级任务执行时，有更高优先级任务进来的话，可以打断低优先级任务的执行。

同步模式下的react运行时

我们知道在同步模式下，从 `setState` 到 虚拟DOM遍历，再到真实DOM更新，整个过程都是同步执行且无法被中断的，这样可能就会出现一个问题 —— 用户事件触发的更新被阻塞。

什么是用户事件触发的更新被阻塞？如果 `React` 正在进行更新任务，此时用户触发了交互事件，且在事件回调中执行了 `setState`，在**同步模式**下，这个更新任务需要 `等待` 当前正在更新的任务完成之后，才会被执行。假如当前 `React` 正在进行的更新任务耗时比较久，用户事件触发的更新任务不能及时被执行，造成下个更新任务被阻塞，从而形成了卡顿。

这时候，我们就希望能够及时响应用户触发的事件，优先执行用户事件触发的更新任务，也就是我们说的**异步模式**

我们可以比较一下，**同步模式**下和**异步模式**(`优先级机制`)下更新任务执行的差异

```javascript 复制代码
import React from "react";
import "./styles.css";

export default class extends React.Component {
  constructor() {
    super();
    this.state = {
      list: new Array(10000).fill(1),
    };
    this.domRef = null;
  }
  componentDidMount() {
    setTimeout(() => {
```

```
      console.log("setTimeout 准备更新", performance.now());
      this.setState(
        {
          list: new Array(10000).fill(Math.random() * 10000),
          updateLanes: 16
        },
        () => {
          console.log("setTimeout 更新完毕", performance.now());
        }
      );
    }, 100);
    setTimeout(() => {
      this.domRef.click();
    }, 150);
  }

  render() {
    const { list } = this.state;
    return (
      <div
        ref={(v) => (this.domRef = v)}        className="App"        onClick={() => {          consc
        ))}      </div>
    );
  }
}
```

`click事件` 触发的更新，会比 `setTimeout` 触发的更新更优先执行，做到了及时响应用户事件，打断 `setTimeout` 更新任务(`低优先级任务`)的执行。

如何运用优先级机制优化react运行时

为了解决同步模式渲染下的缺陷，我们希望能够对 `react` 做出下面这些优化

1. 确定不同场景下所触发更新的优先级，以便我们可以决定优先执行哪些任务
2. 若有更高优先级的任务进来，我们需要打断当前进行的任务，然后执行这个高优先级任务
3. 确保低优先级任务不会被一直打断，在一定时间后能够被升级为最高优先级的任务

确定不同场景下的调度优先级

看过 `react` 源码的小伙伴可能都会有一个疑惑，为什么源码里面有那么多优先级相关的单词？？怎么区分他们呢？

其实在 `react` 中主要分为两类优先级， `scheduler` 优先级和 `lane` 优先级， `lane` 优先级下面又派生出 `event` 优先级

- lane 优先级：主要用于任务调度前，对当前正在进行的任务和被调度任务做一个优先级校验，判断是否需要打断当前正在进行的任务
- event 优先级：本质上也是lane优先级，lane优先级是通用的，event优先级更多是结合浏览器原生事件，对lane优先级做了分类和映射
- scheduler 优先级：主要用在时间分片中任务过期时间的计算

lane优先级

可以用赛道的概念去理解lane优先级，lane优先级有31个，我们可以用31位的二进制值去表示，值的每一位代表一条赛道对应一个lane优先级，**赛道位置越靠前，优先级越高**

优先级	十进制值	二进制值	赛道位置
NoLane	0	0000000000000000000000000000000	0
SyncLane	1	0000000000000000000000000000001	0
InputContinuousHydrationLane	2	0000000000000000000000000000010	1
InputContinuousLane	4	0000000000000000000000000000100	2
DefaultHydrationLane	8	0000000000000000000000000001000	3
DefaultLane	16	0000000000000000000000000010000	4
TransitionHydrationLane	32	0000000000000000000000000100000	5
TransitionLane1	64	0000000000000000000000001000000	6
优先级 十进制值 二进制值 赛道位置

TransitionLane2	128	00000000000000000000000010000000	7
TransitionLane3	256	00000000000000000000000100000000	8
TransitionLane4	512	00000000000000000000001000000000	9
TransitionLane5	1024	00000000000000000000010000000000	10
TransitionLane	2048	00000000000000000000100000000000	11
TransitionLane7	4096	00000000000000000001000000000000	12
TransitionLane8	8192	00000000000000000010000000000000	13
TransitionLane9	16384	00000000000000000100000000000000	14
TransitionLane10	32768	00000000000000001000000000000000	15
TransitionLane11	65536	00000000000000010000000000000000	16
TransitionLane12	131072	00000000000000100000000000000000	17
TransitionLane13	262144	00000000000001000000000000000000	18
TransitionLane14	524288	00000000000010000000000000000000	19
TransitionLane15	1048576	00000000001000000000000000000000	20
优先级	**十进制值**	**二进制值**	**赛道位置**

TransitionLane16	2097152	0000000001000000000000000000	21
RetryLane1	4194304	0000000010000000000000000000	22
RetryLane2	8388608	0000000100000000000000000000	23
RetryLane3	16777216	0000001000000000000000000000	24
RetryLane4	33554432	0000010000000000000000000000	25
RetryLane5	67108864	0000100000000000000000000000	26
SelectiveHydrationLane	134217728	0001000000000000000000000000	27
IdleHydrationLane	268435456	0010000000000000000000000000	28
IdleLane	536870912	0100000000000000000000000000	29
OffscreenLane	1073741824	1000000000000000000000000000	30

相关参考视频讲解：进入学习

event优先级

EventPriority		Lane	数值
DiscreteEventPriority	离散事件。click、keydown、focusin等，事件的触发不是连续，可以做到快速响应	SyncLane	1
ContinuousEventPriority	连续事件。drag、scroll、mouseover等，事件的是连续触发的，快速响应可能会阻塞渲染，优先级较离散事件低	InputContinuousLane	4
DefaultEventPriority	默认的事件优先级	DefaultLane	16
IdleEventPriority	空闲的优先级	IdleLane	536870912

scheduler优先级

SchedulerPriority	EventPriority	大于>17.0.2	小于>17.0.2
ImmediatePriority	DiscreteEventPriority	1	99
UserblockingPriority	Userblocking	2	98
NormalPriority	DefaultEventPriority	3	97
LowPriority	DefaultEventPriority	4	96
IdlePriority	IdleEventPriority	5	95
NoPriority		0	90

优先级间的转换

- lane优先级 转 event优先级（参考 `lanesToEventPriority` 函数）
    - 转换规则：以区间的形式根据传入的lane返回对应的 event 优先级。比如传入的优先级不大于 Discrete 优先级，就返回 Discrete 优先级，以此类推

- event优先级 转 scheduler优先级（参考 `ensureRootIsScheduled` 函数）
    - 转换规则：可以参考上面scheduler优先级表
- event优先级 转 lane优先级（参考 `getEventPriority` 函数）
    - 转换规则：对于非离散、连续的事件，会根据一定规则作转换，具体课参考上面 event 优先级表，

优先级机制如何设计

说到优先级机制，我们可能马上能联想到的是优先级队列，其最突出的特性是**最高优先级先出**，`react` 的优先级机制跟优先级队列类似，不过其利用了**赛道**的概念，配合**位与运算**丰富了队列的功能，比起优先级队列，读写速度更快，更加容易理解

设计思路

1. 合并赛道：维护一个队列，可以存储被占用的赛道
2. 释放赛道：根据优先级释放对应被占用赛道
3. 找出最高优先级赛道：获取队列中最高优先级赛道
4. 快速定位赛道索引：根据优先级获取赛道在队列中所在的位置
5. 判断赛道是否被占用：根据传入优先级判断该优先级所在赛道是否被占用

合并赛道

- **场景**
    - 比如当前正在调度的任务优先级是DefaultLane，用户点击触发更新，有一个高优先级的任务SyncLane产生，需要存储这个任务所占用的赛道
- **运算过程**
    - 运算方式：位或运算 - `a | b`
    - 运算结果：DefaultLane和SyncLane分别占用了第1条和第5条赛道

```javascript 复制代码
DefaultLane优先级为16，SyncLane优先级为1

16 | 1 = 17
```

```
17的二进制值为10001
16的二进制值为10000，1的二进制值为00001
```

释放赛道

- **场景**
  - SyncLane 任务执行完，需要释放占用的赛道
- **运算过程**
  - 运算方式：位与+位非 - `a & ~b`
  - 运算结果：SyncLane赛道被释放，只剩下DefaultLane赛道

```javascript 复制代码
17 & ~1 = 16
17的二进制值为10001

为什么用位非？
~1 = -2
2 的二进制是00010，-2的话符号位取反变为10010
10001和10010进行位与运算得到10000，也就是十进制的16
```

找出最高优先级赛道

- **场景**
  - 当前有 DefaultLane 和 SyncLane 两个优先级的任务占用赛道，在进入 ensureRootIsScheduled 方法后，我需要先调度优先级最高的任务，所以需要找出当前优先级最高的赛道
- **运算过程**
  - 运算方式：位与+符号位取反 - `a & -b`
  - 运算结果：找到了最高优先级的任务SyncLane，SyncLane任务为同步任务，Scheduler将以同步优先级调度当前应用根节点

```javascript 复制代码
17 & -17 = 1

17的二进制值为10001
-17的二进制值为00001
10001和00001进行位与运算得到1，也就是SyncLane
```

快速定位赛道索引

- **场景**
  - 饥饿任务唤醒：在发起调度前，我们需要对队列中的所有赛道进行一个判断，判断该赛道的任务是否过期，如果过期，就优先执行该过期任务。为此，需要维护一个长度为31的数组，数组的每个元素的下标索引与31个优先级赛道一一对应，数组中存储的是**任务的过期时间**，在判断时，我们希望能根据优先级快速找到该优先级在数组中对应的位置。
- **运算过程**
  - 运算方式：Math.clz32
  - 运算结果：找到了DefaultLane的索引位置为4，那就可以释放应用根节点上的eventTimes、expirationTimes，将其所在位置的值赋值为-1，然后执行对应的过期任务

```javascript 复制代码
// 找出 DefaultLane 赛道索引
31 - Math.clz32(16) = 4

16的二进制值为10000
索引4对应的就是第五个赛道
```

- **Math.clz32是用来干什么的?**
  - 获取一个十进制数字对应二进制值中开头0的个数。
  - 所以用31减去 `Math.clz32` 的值就能得到该赛道的索引

判断赛道是否被占用

异步模式下**会存在高优先级任务插队的情况，此情况下 `state` 的计算方式会跟**同步模式下**有些不同。

- **场景**
  1. 我们 setState 之后并不是马上就会更新 `state`，而是会根据 setState 的内容生成一个 `Update` 对象，这个对象包含了更新内容、更新优先级等属性。
  2. 更新 `state` 这个动作是在 `processUpdateQueue` 函数里进行的，函数里面会判断 `Update` 对象的优先级所在赛道是否被占用，来决定是否在此轮任务中计算这个 `Update` 对象的 `state`
     - 如果被占用，代表 `Update` 对象优先级和当前正在进行的任务相等，可以根据 `Update` 对象计算 `state` 并更新到 Fiber 节点的 `memoizedState` 属性上
     - 如果未被占用，代表当前正在进行的任务优先级比这个 `Update` 对象优先级高，相应的这个低优先级的 `Update` 对象将暂不被计算state，留到下一轮低优先级任

务被重启时再进行计算

- **运算过程**
  - 运算方式：位与 `(renderLanes & updateLanes) == updateLanes`
  - 运算结果：0代表当前调度优先级高于某个Update对象优先级

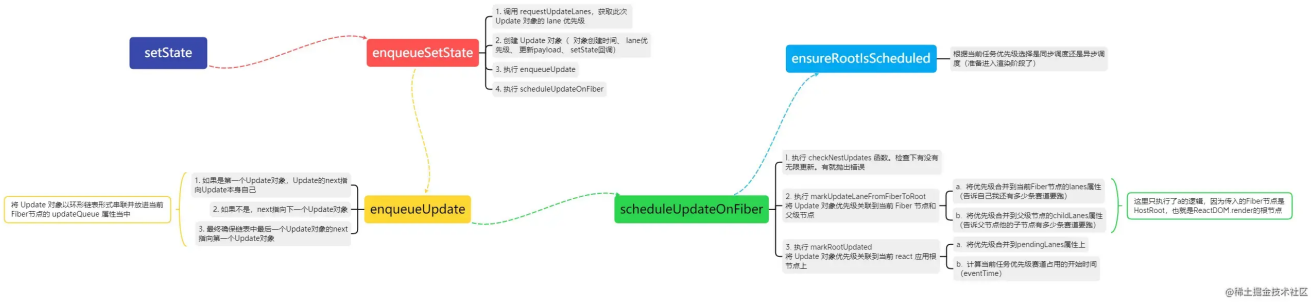
```javascript 复制代码
运算公式
(1 & 16) == 16

1的二进制值为00001
16的二进制值为10000
00001和10000进行位与运算得到0
```

如何将优先级机制融入React运行时

生成一个更新任务

生成任务的流程其实非常简单，入口就在我们常用的 `setState` 函数，先上图



`setState` 函数内部执行的就是 `enqueueUpdate` 函数，而 `enqueueUpdate` 函数的工作主要分为4步：

1. 获取本次更新的优先级。
2. 创建 `Update` 对象
3. 将本次更新优先级关联到当前Fiber节点、父级节点和应用根节点
4. 发起 `ensureRootIsScheduled` 调度。

步骤一：获取本次更新的优先级

步骤一的工作是调用 `requestUpdateLane` 函数拿到此次更新任务的优先级

1. 如果当前为非 `concurrent` 模式
    - 当前不在 render 阶段。返回 syncLane
    - 当前正在 render 阶段。返回 workInProgressRootRenderLanes 中最高的优先级（这里就用到上面的优先级运算机制，**找出最高优先级赛道**）
2. 如果当前为 `concurrent` 模式
    - 需要执行延迟任务的话，比如 `Suspend`、`useTransition`、`useDefferedValue` 等特性。在 `transition` 类型的优先级中寻找空闲的赛道。`transition` 类型的赛道有 16 条，从第 1 条到第 16 条，当到达第 16 条赛道后，下一次 `transition` 类型的任务会回到第 1 条赛道，如此往复。
    - 执行 `getCurrentUpdatePriority` 函数。获取当前更新优先级。如果不为 `NoLane` 就返回
    - 执行 `getCurrentEventPriority` 函数。返回当前的事件优先级。如果没有事件产生，返回 `DefaultEventPriority`

总的来说，`requestUpdateLane` 函数的优先级选取判断顺序如下：

```
SyncLane  >>  TransitionLane  >>  UpdateLane  >>  EventLane
```
复制代码

> 估计有很多小伙伴都会很困惑一个问题，为什么会有这么多获取优先级的函数，这里我整理了一下其他函数的职责

getCurrentUpdatePriority	返回当前任务优先级。 当前任务优先级在很多地方会设置（函数setCurrentUpdaterPriority）
getCurrentEventPriority	返回当前的事件优先级 a. 如果没有事件。返回 DefaultEventPriority b. 如果有。根据事件类型返回优先级
getNextLanes	从根节点的 pendingLanes 中找出最高优先级的赛道，里面调用了 getHighestPriorityLanes 函数
getHighestPriorityLanes	1. 先调用 getHighestPriorityLane 获取传入 lanes 中的最高优先级 2. 根据 getHighestPriorityLane 的返回值进入 switch 返回相应的优先级
getHighestPriorityLane	通过位与+符号位取反的方式计算 lanes 中最高的优先级

步骤二：创建 Update 对象

这里的代码量不多，其实就是将 setState 的参数用一个对象封装起来，留给 render 阶段用

```javascript 复制代码
function createUpdate(eventTime, lane) {
  var update = {
    eventTime: eventTime,
    lane: lane,
    tag: UpdateState,
    payload: null,
    callback: null,
    next: null
  };
  return update;
}
```

步骤三：关联优先级

在这里先解释两个概念，一个是 `HostRoot`，一个是 `FiberRootNode`

- `HostRoot`：就是 `ReactDOM.render` 的第一个参数，组件树的根节点。`HostRoot`可能会存在多个，因为 `ReactDOM.render` 可以多次调用
- `FiberRootNode`：react 的应用根节点，每个页面只有一个 react 的应用根节点。可以从 `HostRoot` 节点的 `stateNode` 属性访问

这里关联优先级主要执行了两个函数

1. `markUpdateLaneFromFiberToRoot`。该函数主要做了两个事情
    - 将优先级合并到当前 Fiber 节点的 lanes 属性中
    - 将优先级合并到父级节点的 childLanes 属性中（告诉父节点他的子节点有多少条赛道要跑） 但因为函数传入的 Fiber 节点是 `HostRoot`，也就是 `ReactDOM.render` 的根节点，也就是说没有父节点了，所以第二件事情没有做
2. `markRootUpdated`。该函数也是主要做了两个事情
    - 将待调度任务优先级合并到当前 react 应用根节点上
    - 计算当前任务优先级赛道占用的开始时间（eventTime）

由此可见，`react` 的优先级机制并不独立运行在每一个组件节点里面，而是依赖一个全局的 `react` 应用根节点去控制下面多个组件树的任务调度

> 将**优先级**关联到这些Fiber节点有什么用?

先说说他们的**区别**

- lanes：只存在非 react 应用根节点上，记录当前 Fiber 节点的 lane 优先级
- childLanes：只存在非 react 应用根节点上，记录当前 Fiber 节点下的所有子 Fiber 节点的 lane 优先级
- pendingLanes：只存在 react 应用根节点上，记录的是所有 `HostRoot` 的 lane 优先级

具体**应用场景**

1. 释放赛道。上面说的优先级运算机制提到了任务执行完毕会释放赛道，具体来说是在 commit 阶段结束之后释放被占用的优先级，也就是 `markRootFinished` 函数。
2. 判断赛道是否被占用。在 render 阶段的 `beginWork` 流程里面，会有很多判断 `childLanes` 是否被占用的判断

步骤四：发起调度

调度里面最关键的一步，就是 `ensureRootIsScheduled` 函数的调用，该函数的逻辑就是由下面两大部分构成，**高优先级任务打断低优先级任务** 和 **饥饿任务问题**

高优先级任务打断低优先级任务

该部分流程可以分为三部曲

1. cancelCallback
2. pop(taskQueue)
3. 低优先级任务重启

cancelCallback

```javascript
var existingCallbackNode = root.callbackNode;
var existingCallbackPriority = root.callbackPriority;
var newCallbackPriority = getHighestPriorityLane(nextLanes);
if (existingCallbackPriority === newCallbackPriority) {
    ...
    return;
}
if (existingCallbackNode != null) {
    cancelCallback(existingCallbackNode);
}

newCallbackNode = scheduleCallback(
```

```
    schedulerPriorityLevel,
    performConcurrentWorkOnRoot.bind(null, root)
);
root.callbackPriority = newCallbackPriority;
root.callbackNode = newCallbackNode;
```

上面是 `ensureRootIsScheduled` 函数的一些代码片段，先对变量做解释

- `existingCallbackNode`：当前 render 阶段正在进行的任务
- `existingCallbackPriority`：当前 render 阶段正在进行的任务优先级
- `newCallbackPriority`：此次调度优先级

这里会判断 `existingCallbackPriority` 和 `newCallbackPriority` 两个优先级是否相等，**如果相等**，此次更新合并到当前正在进行的任务中。**如果不相等**，代表此次更新任务的优先级更高，需要打断当前正在进行的任务

> 如何打断任务?

1. 关键函数 `cancelCallback(existingCallbackNode)`，`cancelCallback` 函数就是将 `root.callbackNode` 赋值为null
2. `performConcurrentWorkOnRoot` 函数会先把 `root.callbackNode` 缓存起来，在函数末尾会再判断 `root.callbackNode` 和开始缓存起来的值是否一样，如果不一样，就代表 `root.callbackNode` 被赋值为null了，有更高优先级任务进来。
3. 此时 `performConcurrentWorkOnRoot` 返回值为null

下面是 `performConcurrentWorkOnRoot` 代码片段

kotlin 复制代码

```
...
var originalCallbackNode = root.callbackNode;
...
// 函数末尾
if (root.callbackNode === originalCallbackNode) {
    return performConcurrentWorkOnRoot.bind(null, root);
}
return null;
```

由上面 `ensureRootIsScheduled` 的代码片段可以知道，`performConcurrentWorkOnRoot` 函数是被 `scheduleCallback` 函数调度的，具体返回后的逻辑需要到 `Scheduler` 模块去找

pop(taskQueue)

```javascript 复制代码
var callback = currentTask.callback;
if (typeof callback === 'function') {
  ...
} else {
  pop(taskQueue);
}
```

上面是 `Scheduler` 模块里面 `workLoop` 函数的代码片段，`currentTask.callback` 就是 `scheduleCallback` 的第二个参数，也就是 `performConcurrentWorkOnRoot` 函数

承接上个主题，如果 `performConcurrentWorkOnRoot` 函数返回了null，`workLoop` 内部就会执行 `pop(taskQueue)`，将当前的任务从 `taskQueue` 中弹出。

低优先级任务重启

上一步中说道一个低优先级任务从 `taskQueue` 中被弹出。那高优先级任务执行完毕之后，如何重启回之前的低优先级任务呢?

关键是在 `commitRootImpl` 函数

```javascript 复制代码
var remainingLanes = mergeLanes(finishedWork.lanes, finishedWork.childLanes);
markRootFinished(root, remainingLanes);

...

ensureRootIsScheduled(root, now());
```

`markRootFinished` 函数刚刚上面说了是释放已完成任务所占用的赛道，那也就是说未完成任务依然会占用其赛道，所以我们可以重新调用 `ensureRootIsScheduled` 发起一次新的调度，去重启低优先级任务的执行。我们可以看下重启部分的判断

```javascript 复制代码
var nextLanes = getNextLanes(
    root, root === workInProgressRoot ? workInProgressRootRenderLanes : NoLanes
);
// 如果 nextLanes 为 NoLanes，就证明所有任务都执行完毕了
if (nextLanes === NoLanes) {
    ...
    root.callbackNode = null;
    root.callbackPriority = NoLane;
```

```
    // 只要 nextLanes 为 NoLanes，就可以结束调度了
    return;
}
// 如果 nextLanes 不为 NoLanes，就代表还有任务未执行完，也就是那些被打断的低优先级任务
...
```

饥饿任务问题

上面说到，在高优先级任务执行完毕之后，低优先级任务就会被重启，但假设如果持续有高优先级任务持续进来，我的低优先级任务岂不是没有重启之日？

所以 react 为了处理解决饥饿任务问题，react 在 `ensureRootIsScheduled` 函数开始的时候做了以下处理：（参考 `markStarvedLanesAsExpired` 函数）

```javascript 复制代码
var lanes = pendingLanes;
while (lanes > 0) {
    var index = pickArbitraryLaneIndex(lanes);
    var lane = 1 << index;
    var expirationTime = expirationTimes[index];
    if (expirationTime === NoTimestamp) {
      if ((lane & suspendedLanes) === NoLanes || (lane & pingedLanes) !== NoLanes) {
        expirationTimes[index] = computeExpirationTime(lane, currentTime);
      }
    } else if (expirationTime <= currentTime) {
      root.expiredLanes |= lane;
    }
    lanes &= ~lane;
}
```

1. 遍历31条赛道，判断每条赛道的过期时间是否为 `NoTimestamp`，如果是，且该赛道存在待执行的任务，则为该赛道初始化过期时间
2. 如果该赛道已存在过期时间，且过期时间已经小于当前时间，则代表任务已过期，需要将当前优先级合并到 `expiredLanes`，这样在下一轮 render 阶段就会以同步优先级调度当前 `HostRoot`

可以参考 render 阶段执行的函数 `performConcurrentWorkOnRoot` 中的代码片段

```javascript 复制代码
var exitStatus = shouldTimeSlice(root, lanes) && ( !didTimeout) ?
                    renderRootConcurrent(root, lanes) :
                    renderRootSync(root, lanes);
```

可以看到只要 `shouldTimeSlice` 只要返回 false，就会执行 `renderRootSync`，也就是以同步优先级进入 render 阶段。而 `shouldTimeSlice` 的逻辑也就是刚刚的 `expiredLanes` 属性相关

```javascript
function shouldTimeSlice(root, lanes) {
  // 如果 expiredLanes 里面有东西，代表有饥饿任务
  if ((lanes & root.expiredLanes) !== NoLanes) {
    return false;
  }
  var SyncDefaultLanes = InputContinuousHydrationLane |
                         InputContinuousLane |
                         DefaultHydrationLane |
                         DefaultLane;

  return (lanes & SyncDefaultLanes) === NoLanes;
}
```

总结

`react` 的优先级机制在源码中并不是一个独立的，解耦的模块，而是涉及到了react整体运行的方方面面，最后回归整理下优先级机制在源码中的使用，让大家对优先级机制有一个更加整体的认知。

1. 时间分片。涉及到任务打断、根据优先级计算分片时长
2. setState 生成 Update 对象。每个 Update 对象里面都有一个 lane 属性，代表此次更新的优先级
3. 高优先级任务打断低优先级任务。每一次调度都会对正在进行任务和当前任务最高优先级做比较，如果不相等，就代表有高优先级任务进来，需要打断当前正在的任务。
4. 低优先级任务重启。协调 `(reconcile)` 的下一个阶段是渲染 `(renderer)`，也就是我们说的 commit 阶段，在此阶段末尾，会调用 `ensureRootIsScheduled` 发起一次新的调度，执行尚未完成的低优先级任务。
5. 饥饿任务唤醒。每次调度的开始，都会先检查下有没有过期任务，如果有的话，下一次就会以同步优先级进行 render 任务 `(reconcile)`，同步优先级就是最高的优先级，不会被打断