link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=157 sentences=284, words=1858

1. monorepo管理

- Monorepo 是管理项目代码的一个方式，指在一个项目仓库(repo)中管理多个模块/包(package)
- monorepo 最主要的好处是统一的工作流和代码共享
- Lerna (https://github.com/lerna/lerna)是一个管理多个 npm 模块的工具,优化维护多包的工作流，解决多个包互相依赖，且发布需要手动维护多个包的问题
- yarn (https://classic.yarnpkg.com/en/docs/cli/)

1.1 MultiRepo

1.1.1 优点

- 各模块的管理自由度较高，可以自行选择构建工具、依赖管理、单元测试等配套设施
- 各模块的体积也不会太大

1.1.2 缺点

- 仓库分散不好找，分支管理混乱
- 版本更新繁琐，如果公共模块发生了变化，需要对所有的模块进行依赖更新
- CHANGELOG不好梳理，无法自动关联各个模块的变动

1.2 MonoRepo

1.2.1 优点

- 一个仓库维护多个模块，方便好找
- 方便版本管理和依赖管理，模块之间的引用调试都比较方便
- 方便统一生成CHANGELOG

1.2.2 缺点

- 统一构建工具, 需要构建工具能构建所的模块
- 仓库体积变大

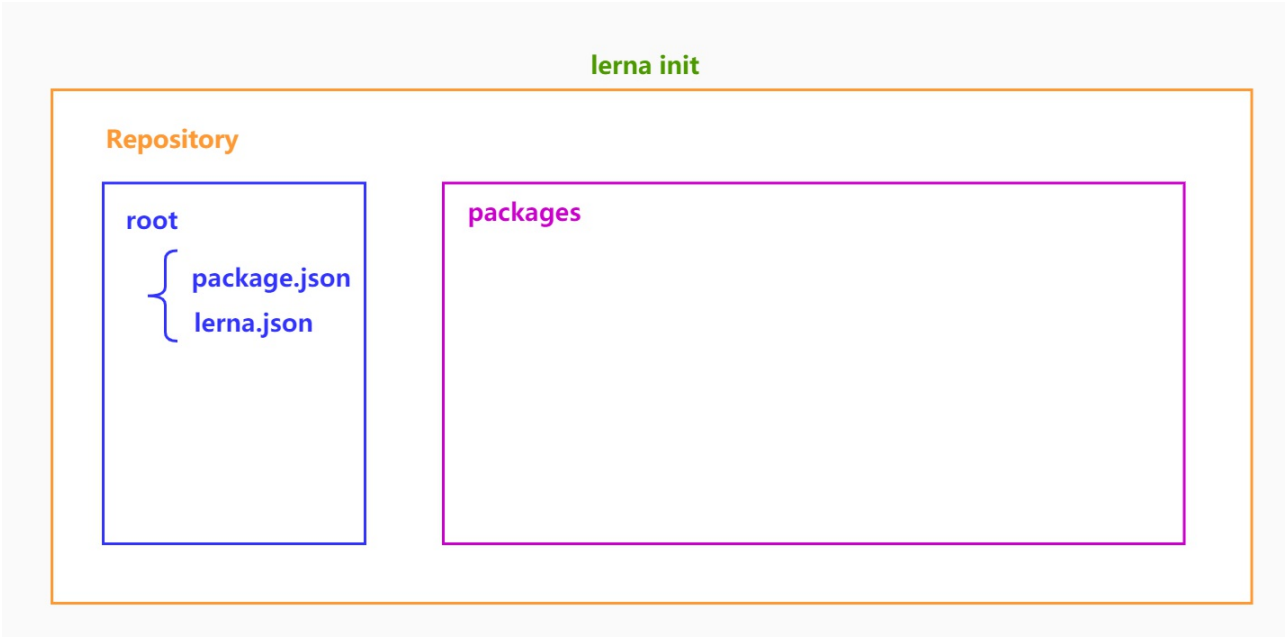
1.3 使用lerna

1.3.1 安装lerna

```
npm i lerna -g
```

1.3.2 初始化项目

```
mkdir lerna-project
cd lerna-project

lerna init
lerna notice cli v4.0.0
lerna info Initializing Git repository
lerna info Creating package.json
lerna info Creating lerna.json
lerna info Creating packages directory
lerna success Initialized Lerna files
```



1.3.3 package.json

package.json

```
{
  "name": "root",
  "private": true,
  "devDependencies": {
    "lerna": "^4.0.0"
  }
}
```

1.3.4 lerna.json

lerna.json

```
{
  "packages": [
    "packages/*"
  ],
  "version": "0.0.0"
}
```

1.4 yarn workspace

- yarn workspace允许我们使用 monorepo 的形式来管理项目
- 在安装 node_modules 的时候它不会安装到每个子项目的 node_modules 里面，而是直接安装到根目录下面，这样每个子项目都可以读取到根目录的 node_modules
- 整个项目只有根目录下面会有一份 yarn.lock 文件。子项目也会被 link 到 node_modules 里面，这样就允许我们就可以直接用 import 导入对应的项目
- yarn.lock文件是自动生成的,也完全Yarn来处理. yarn.lock锁定你安装的每个依赖项的版本，这可以确保你不会意外获得不良依赖

1.4.1 开启workspace

package.json

```
{
  "name": "root",
  "private": true,
+  "workspaces": [
+    "packages/*"
+  ],
  "devDependencies": {
    "lerna": "^3.22.1"
  }
}
```

1.3.2 创建子项目

- react是React核心，包含了 React.createElement等代码
- shared 存放各个模块公用的全局变量和方法
- scheduler 实现了优先级调度功能
- react-reconciler 提供了协调器的功能
- react-dom 提供了渲染到DOM的功能

```
lerna create react
lerna create shared
lerna create scheduler
lerna create react-reconciler
lerna create react-dom
```

1.3.3 添加依赖

- yampkg (https://classic.yampkg.com/en/docs/cli)
- lerna (https://github.com/lerna/lerna#readme)

1.3.3.1 设置加速镜像

```
yarn config get registry
yarn config set registry http://registry.npm.taobao.org/
yarn config set registry http://registry.npmjs.org/
```

1.3.4 常用命令

1.3.4.1 根空间添加依赖

```
yarn add chalk --ignore-workspace-root-check
```

1.3.4.2 给某个项目添加依赖

```
yarn workspace react add object-assign
```

```
yarn install
lerna bootstrap --npm-client yarn --use-workspaces
```

1.3.4.4 其它命令

作用 命令 查看工作空间信息 yam workspaces info 删除所有的 node_modules lerna clean 等于 yam workspaces run clean 重新获取所有的 node_modules yam install –force 查看缓存目录 yam cache dir 清除本地缓存 yam cache clean

2. 调试源码

2.1 下载代码

```
git clone https:
```

2.2 编译源码

- development-workflow (https://zh-hans.reactjs.org/docs/how-to-contribute.html#development-workflow)

```
cd react
yarn build react,shared,scheduler,react-reconciler,react-dom --type=NODE
cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link
```

```
create-react-app debug-react17
cd debug-react17
yarn link react react-dom
```

3. setState的更新是同步还是异步的？#

3.1 setState

- 在开发中我们并不能直接通过修改state的值来让界面发生更新
    - 因为修改了state之后，希望React根据最新的State来重新渲染界面，但是这种方式的修改React并不知道数据发生了变化
    - React并没有实现类似于Vue2中的Object.defineProperty或者Vue3中的Proxy的方式来监听数据的变化
    - 必须通过setState来告知React数据已经发生了变化

3.2 异步更新

- React在执行setState的时候会把更新的内容放入队列
- 在事件执行结束后会计算state的数据，然后执行回调
- 最后根据最新的state计算虚拟DOM更新真实DOM
- 优点
    - 保持内部一致性。如果改为同步更新的方式，尽管 setState 变成了同步，但是 props 不是
    - 为后续的架构升级启用并发更新,React 会在 setState 时，根据它们的数据来源分配不同的优先级，这些数据来源有：事件回调句柄、动画效果等，再根据优先级并发处理，提升渲染性能
    - setState设计为异步，可以显著的提升性能

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

class Counter extends React.Component{
  state = {number:0}
  buttonClick = ()=>{
    console.log('buttonClick');
    this.setState({number:this.state.number+1});
    console.log(this.state.number);
    this.setState({number:this.state.number+1});
    console.log(this.state.number);
  }
  divClick = ()=>{
    console.log('divClick');
  }
  render(){
    return (
      <div onClick={this.divClick} id="counter">
        <p>{this.state.number}p>
        <button onClick={this.buttonClick}>+button>
      div>
    )
  }
}
ReactDOM.render(<Counter/>,document.getElementById('root'));
```

3.3 回调执行

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

class Counter extends React.Component{
  state = {number:0}
  buttonClick = ()=>{
    console.log('buttonClick');
+   this.setState({number:this.state.number+1},()=>{
+     console.log(this.state.number);
+   });
+   this.setState({number:this.state.number+1},()=>{
+     console.log(this.state.number);
+   });
  }
  divClick = ()=>{
    console.log('divClick');
  }
  render(){
    return (

        {this.state.number}
        +

    )
  }
}
ReactDOM.render(,document.getElementById('root'));
```

3.4 函数更新

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

class Counter extends React.Component{
  state = {number:0}
  buttonClick = ()=>{
    console.log('buttonClick');
+   this.setState((state)=>({number:state.number+1}),()=>{
+     console.log(this.state.number);
+   });
+
+   this.setState((state)=>({number:state.number+1}),()=>{
+     console.log(this.state.number);
+   });

  }
  divClick = ()=>{
    console.log('divClick');
  }
  render(){
    return (

        {this.state.number}
        +

    )
  }
}
ReactDOM.render(,document.getElementById('root'));
```

3.5 同步执行

- 在React的生命周期函数和合成事件中可以修改批量更新的变量 isBatchingUpdates
- 可以设置为批量，其它地方如 addEventListener、 setTimeout和setInterval里无法设置

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

class Counter extends React.Component{
  state = {number:0}
  buttonClick = ()=>{
    console.log('buttonClick');// 2234
    this.setState((state)=>({number:state.number+1}),()=>{
      console.log(this.state.number);
    });

    this.setState((state)=>({number:state.number+1}),()=>{
      console.log(this.state.number);
    });
+   setTimeout(()=>{
+     this.setState((state)=>({number:state.number+1}),()=>{
+       console.log(this.state.number);
+     });
+
+     this.setState((state)=>({number:state.number+1}),()=>{
+       console.log(this.state.number);
+     });
    });

  }
  divClick = ()=>{
    console.log('divClick');
  }
  render(){
    return (

        {this.state.number}
        +

    )
  }
}
ReactDOM.render(,document.getElementById('root'));
```

3.6 强行批量更新

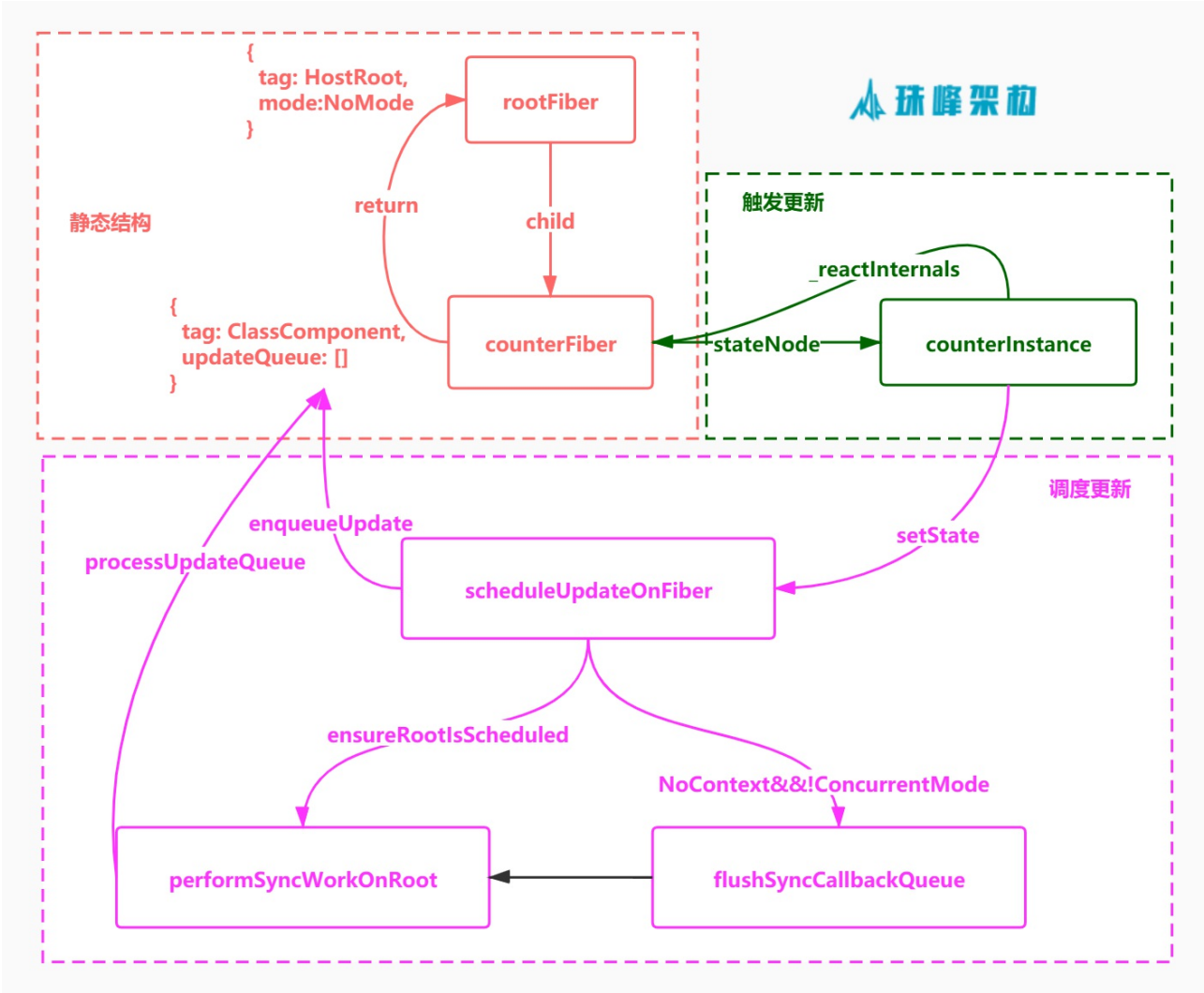
```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

class Counter extends React.Component{
  state = {number:0}
  buttonClick = ()=>{
    console.log('buttonClick');// 2234
    setTimeout(()=>{
+     ReactDOM.unstable_batchedUpdates(()=>{
+       this.setState((state)=>({number:state.number+1}));
+       console.log(this.state.number);
+     });
    });
  }
  divClick = ()=>{
    console.log('divClick');
  }
  render(){
    return (

        {this.state.number}
        +

    )
  }
}
ReactDOM.render(,document.getElementById('root'));
```

3.7 并发更新

- 启用 concurrent模式

```
+ReactDOM.unstable_createRoot(document.getElementById('root')).render();
+ReactDOM.createRoot(document.getElementById('root')).render();
```

4.实现setState



4.1 src\index.js

```
import { HostRoot, ClassComponent } from './ReactWorkTags';
import { Component } from './ReactFiberClassComponent';
import { NoMode, ConcurrentMode } from './ReactTypeOfMode';
import { batchedUpdates } from './ReactFiberWorkLoop';
class Counter extends Component {
  constructor() {
    super();
    this.state = { number: 0 };
  }
  onClick = (event) => {
    this.setState({ number: this.state.number + 1 });
    console.log('setState1', this.state.number);
    this.setState({ number: this.state.number + 1 });
    console.log('setState2', this.state.number);
    setTimeout(() => {
      this.setState({ number: this.state.number + 1 });
      console.log('setTimeout setState1', this.state.number);
      this.setState({ number: this.state.number + 1 });
      console.log('setTimeout setState2', this.state.number);
    });
  }
  render() {
    console.log('render', this.state.number);
    return this.state.number;
  }
}
let counterInstance = new Counter();
let mode = ConcurrentMode;
let rootFiber = { tag: HostRoot, updateQueue: [], mode }
let counterFiber = { tag: ClassComponent, updateQueue: [], mode }
counterFiber.stateNode = counterInstance;
counterInstance._reactInternals = counterFiber;
counterFiber.return = rootFiber;
rootFiber.child = counterFiber;

document.addEventListener('click',(nativeEvent)=>{
  let syntheticEvent = {nativeEvent};
  batchedUpdates(()=>counterInstance.onClick(syntheticEvent));
});
```

src\ReactWorkTags.js

```
export const HostRoot = 3;
export const ClassComponent = 1;
```

4.3 src\ReactTypeOfMode.js

src\ReactTypeOfMode.js

```
export const NoMode = 0b00000;
export const ConcurrentMode = 0b00100;
```

4.4 src\ReactFiberClassComponent.js

src\ReactFiberClassComponent.js

```
import {scheduleUpdateOnFiber} from './ReactFiberWorkLoop';
let classComponentUpdater = {
    enqueueSetState: function (inst, payload) {
        const fiber = get(inst);
        const update = createUpdate();
        update.payload = payload;
        enqueueUpdate(fiber, update);
        scheduleUpdateOnFiber(fiber);
    }
}
function get(inst){
    return inst._reactInternals;
}
function createUpdate(){
    return {};
}
function enqueueUpdate(fiber, update) {
    var updateQueue = fiber.updateQueue;
    updateQueue.push(update);
}

export class Component {
    constructor() {
        this.updater = classComponentUpdater;
    }
    setState(partialState) {
        this.updater.enqueueSetState(this, partialState);
    }
}
```

4.5 src\ReactFiberWorkLoop.js

src\ReactFiberWorkLoop.js

```
import { ClassComponent, HostRoot } from './ReactWorkTags';
import {NoMode,ConcurrentMode} from './ReactTypeOfMode'
let syncQueue;
let NoLanePriority = 0;
let SyncLanePriority = 12;
export let NoContext = 0;
export let BatchedContext = 1;
export let executionContext = NoContext;
function markUpdateLaneFromFiberToRoot(fiber) {
    let parent = fiber.return;
    while (parent) {
        fiber = parent;
        parent = parent.return;
    }
    if (fiber.tag === HostRoot) {
        return fiber;
    }
    return null;
}
export function getExecutionContext() {
    return executionContext;
}
export function batchedUpdates(fn) {
    const prevExecutionContext = executionContext;
    executionContext |= BatchedContext;
    try {
      return fn();
    } finally {
      executionContext = prevExecutionContext;
      if (executionContext === NoContext) {
        flushSyncCallbackQueue();
      }
    }
  }
export function scheduleUpdateOnFiber(fiber) {
    let root = markUpdateLaneFromFiberToRoot(fiber);
    ensureRootIsScheduled(root);
    if (executionContext === NoContext && (fiber.mode & ConcurrentMode) === NoMode) {
        flushSyncCallbackQueue();
    }
}

function ensureRootIsScheduled(root) {
    let existingCallbackPriority = root.callbackPriority;
    let newCallbackPriority = SyncLanePriority;
    if (existingCallbackPriority === newCallbackPriority) {
        return;
    }
    scheduleSyncCallback(performSyncWorkOnRoot.bind(null, root));
    queueMicrotask(flushSyncCallbackQueue);
    root.callbackPriority = newCallbackPriority;
}
export function flushSyncCallbackQueue() {
    if(syncQueue){
        for (let i = 0; i < syncQueue.length; i++) {
            let callback = syncQueue[i];
            do {
                callback = callback();
            } while (callback);
        }
        syncQueue = null;
    }
}
function scheduleSyncCallback(callback) {
    if (!syncQueue) {
        syncQueue = [callback];
    } else {
        syncQueue.push(callback);
    }
}
function performSyncWorkOnRoot(workInProgress) {
    let root = workInProgress;
    while (workInProgress) {
        if (workInProgress.tag === ClassComponent) {
            let inst = workInProgress.stateNode;
            inst.state = processUpdateQueue(inst,workInProgress);
            workInProgress.stateNode.render();
        }
        workInProgress = workInProgress.child;
    }
    commitRoot(root);
}
function processUpdateQueue(inst,workInProgress){
    return  workInProgress.updateQueue.reduce((state, update) => ({ ...state, ...update.payload }), inst.state);;
}
function commitRoot(root){
    root.callbackPriority = NoLanePriority;
}
```

5.React中的优先级

- 不同事件产生的更新优先级不同

5.1 React中的优先级

- 事件优先级：按照用户事件的交互紧急程度，划分的优先级
- 更新优先级：事件导致React产生的更新对象（update）的优先级（update.lane）
- 任务优先级：产生更新对象之后，React去执行一个更新任务，这个任务所持有的优先级
- 调度优先级：Scheduler依据React更新任务生成一个调度任务，这个调度任务所持有的优先级

> 前三者属于React的优先级机制，第四个属于 scheduler的优先级机制

5.2 事件优先级

- 离散事件（DiscreteEvent）：click、keydown等，这些事件的触发不是连续的，优先级为 0
- 用户阻塞事件（UserBlockingEvent）：drag、scroll、mouseover等，特点是连续触发，阻塞渲染，优先级为1
- 连续事件（ContinuousEvent）：canplay、error，优先级最高，为2

src\react\packages\shared\ReactTypes.js

```
export const DiscreteEvent = 0;
export const UserBlockingEvent = 1;
export const ContinuousEvent = 2;
```

src\react\packages\scheduler\src\Scheduler.js

```
function unstable_runWithPriority(priorityLevel, eventHandler) {
  switch (priorityLevel) {
    case ImmediatePriority:
    case UserBlockingPriority:
    case NormalPriority:
    case LowPriority:
    case IdlePriority:
      break;
    default:
      priorityLevel = NormalPriority;
  }
  var previousPriorityLevel = currentPriorityLevel;
  currentPriorityLevel = priorityLevel;
  try {
    return eventHandler();
  } finally {
    currentPriorityLevel = previousPriorityLevel;
  }
}
```

5.3 更新优先级

- setState本质上是调用enqueueSetState,生成一个update对象，这时候会计算它的更新优先级，即update.lane
- 首先找出Scheduler中记录的优先级 `schedulerPriority`，然后计算更新优先级

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

```
const TotalLanes = 31;

export const NoLanes: Lanes =  0b0000000000000000000000000000000;
export const NoLane: Lane =  0b0000000000000000000000000000000;

export const SyncLane: Lane =  0b0000000000000000000000000000001;
export const SyncBatchedLane: Lane =  0b0000000000000000000000000000010;

export const InputDiscreteHydrationLane: Lane =  0b0000000000000000000000000000100;
const InputDiscreteLanes: Lanes =  0b0000000000000000000000000011000;

const InputContinuousHydrationLane: Lane =  0b0000000000000000000000000100000;
const InputContinuousLanes: Lanes =  0b0000000000000000000000011000000;

export const DefaultHydrationLane: Lane =  0b0000000000000000000000100000000;
export const DefaultLanes: Lanes =  0b0000000000000000000111000000000;

const TransitionHydrationLane: Lane =  0b0000000000000000001000000000000;
const TransitionLanes: Lanes =  0b0000000001111111110000000000000;

const RetryLanes: Lanes =  0b0000011110000000000000000000000;

export const SomeRetryLane: Lanes =  0b0000010000000000000000000000000;

export const SelectiveHydrationLane: Lane =  0b0000100000000000000000000000000;

const NonIdleLanes =  0b0000111111111111111111111111111;

export const IdleHydrationLane: Lane =  0b0001000000000000000000000000000;
const IdleLanes: Lanes =  0b0110000000000000000000000000000;

export const OffscreenLane: Lane =  0b1000000000000000000000000000000;

let currentUpdateLanePriority = NoLanePriority;

export function getCurrentUpdateLanePriority() {
  return currentUpdateLanePriority;
}

export function setCurrentUpdateLanePriority(newLanePriority) {
  currentUpdateLanePriority = newLanePriority;
}
```

5.4 任务优先级

- update会被一个React的更新任务执行
- 任务优先级被用来区分多个更新任务的紧急程度
- 收敛同等优先级的任务调度
- 高优先级任务及时响应

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

```
export const SyncLanePriority: LanePriority = 15;
export const SyncBatchedLanePriority: LanePriority = 14;

const InputDiscreteHydrationLanePriority: LanePriority = 13;
export const InputDiscreteLanePriority: LanePriority = 12;

const InputContinuousHydrationLanePriority: LanePriority = 11;
export const InputContinuousLanePriority: LanePriority = 10;

const DefaultHydrationLanePriority: LanePriority = 9;
export const DefaultLanePriority: LanePriority = 8;

const TransitionHydrationPriority: LanePriority = 7;
export const TransitionPriority: LanePriority = 6;

const RetryLanePriority: LanePriority = 5;

const SelectiveHydrationLanePriority: LanePriority = 4;

const IdleHydrationLanePriority: LanePriority = 3;
const IdleLanePriority: LanePriority = 2;

const OffscreenLanePriority: LanePriority = 1;

export const NoLanePriority: LanePriority = 0;
```

5.5 调度优先级

- 一旦任务被调度，那么它就会进入scheduler
- 在Scheduler中，这个任务会被包装一下，生成一个属于Scheduler自己的task，这个task持有的优先级就是调度优先级

src\react\packages\react-reconciler\src\SchedulerWithReactIntegration.old.js

```
export const ImmediatePriority: ReactPriorityLevel = 99;
export const UserBlockingPriority: ReactPriorityLevel = 98;
export const NormalPriority: ReactPriorityLevel = 97;
export const LowPriority: ReactPriorityLevel = 96;
export const IdlePriority: ReactPriorityLevel = 95;
export const NoPriority: ReactPriorityLevel = 90;
```