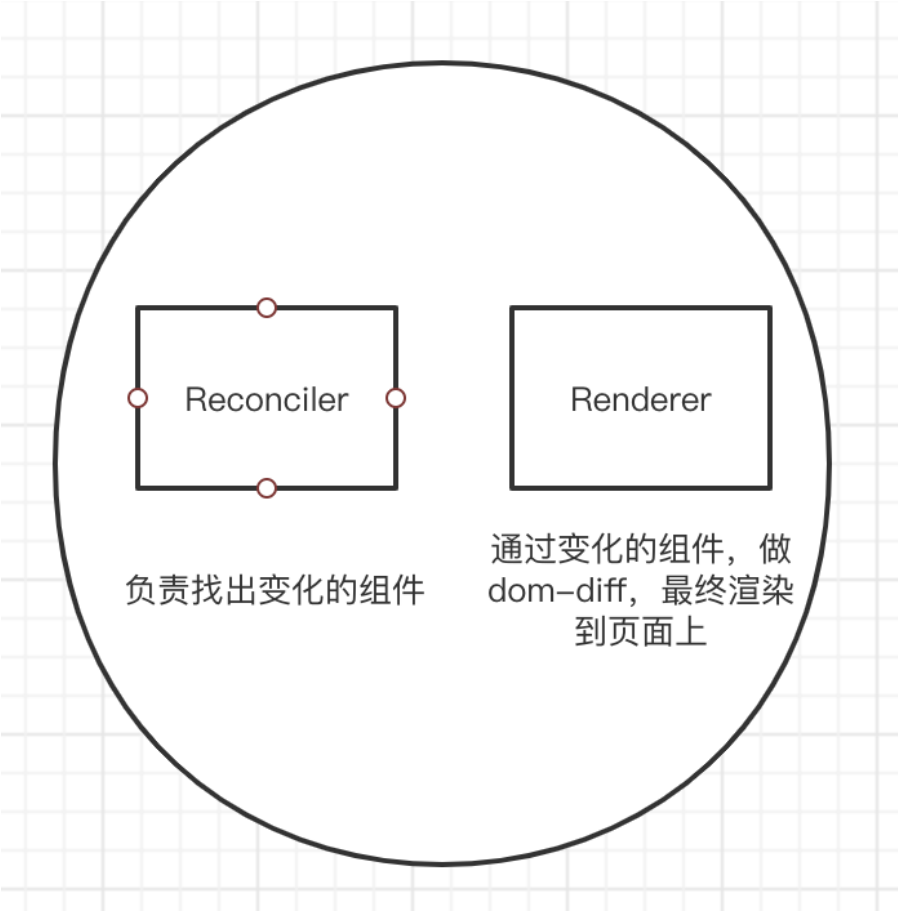
# react17.0 源码分析大纲

## react15的整体架构

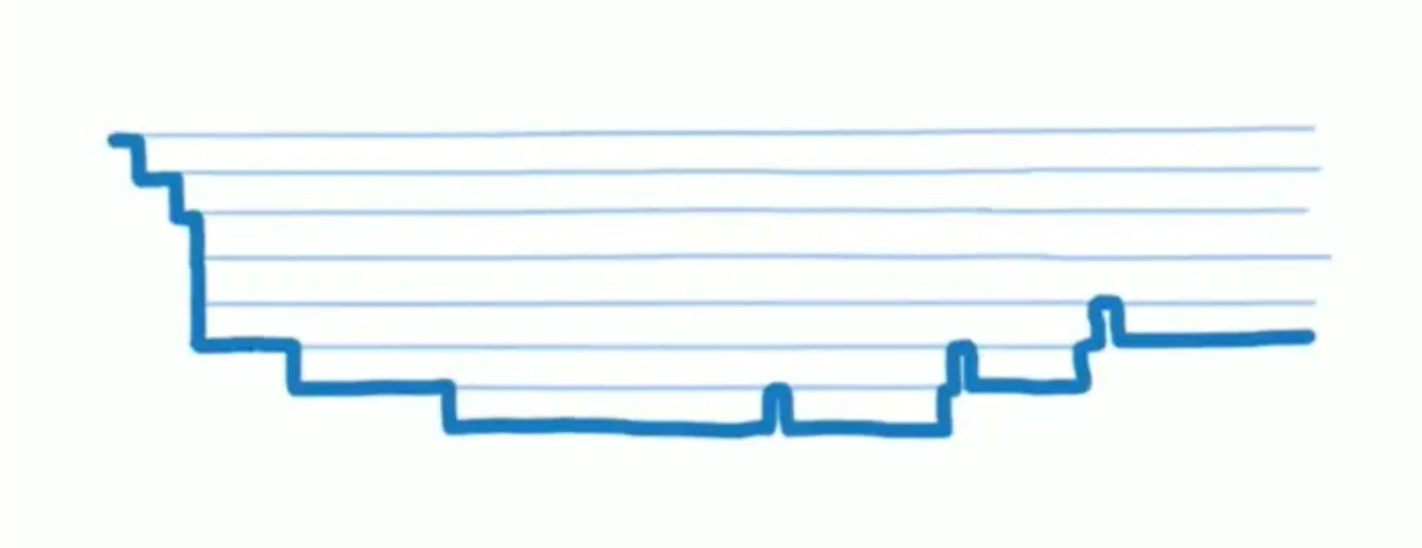
a. Reconciler 找出更新的组件
b. Renderer 渲染到页面

```
// 批处理的优化
this.setState({
    "a": ""
});
this.setState({
    "b": ""
});
// 只发起一次更新
// react内部默认批处理
// 不批处理 unBatchUpdate
// 1. ReactDOM.render() 页面还是白的。unBatchUpdate
```



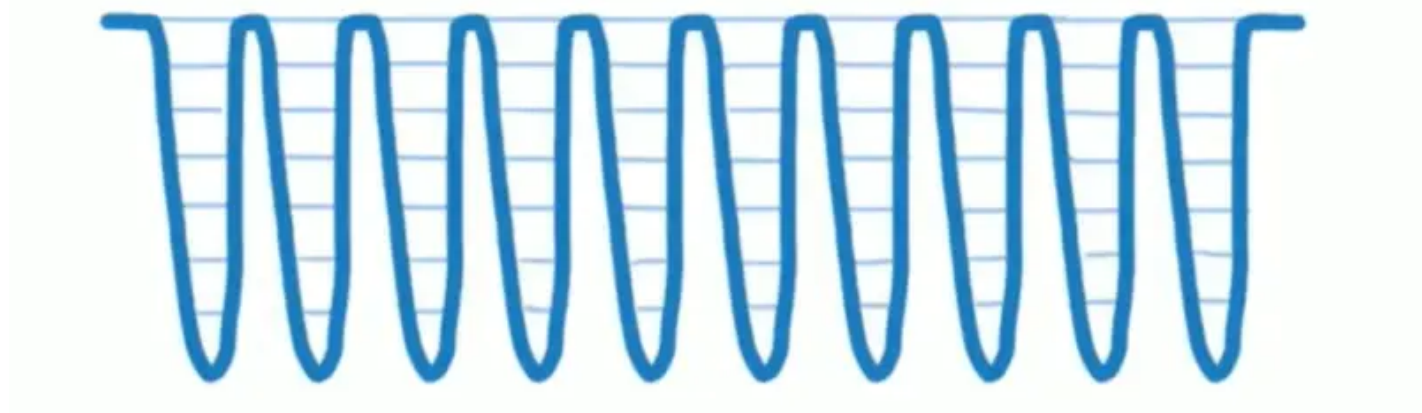
## react15架构的缺点

c. react15架构师递归的，一个长任务，会导致阻塞用户后续交互，会卡顿

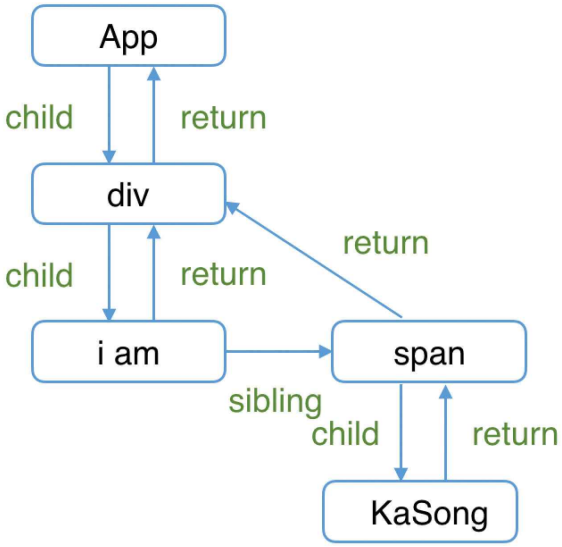


## react16 Fiber架构的做法

执行异步的调度任务会在宏任务中执行，这样可以保证，不会让用户失去响应



同时react16对所有的更新都做了一个优先级的绑定，当出现多个更新同时需要处理时，可以中断低优先级更新，先执行高优先级的更新。新增了Scheduler模块，来调度任务的优先级。




比如：

1. 当span发起了一次优先级为C的更新，还没有更新完成，我们记为第一次更新。
2. 紧接着i am 发起来一次优先级为A的更新，记为第二次更新。优先级为A的更新会打断优先级为C的更新，先执行第二次更新。
3. 再对KaSong 发起一个优先级为A的更新，记为第三次更新。由于第一次更新的优先级会随着时间的流逝也会提升，如果这时第一次更新的优先级和第三次更新一致了，就会一起执行。否则，还是会先执行第三次更新，再执行第一次更新。但是由于优先级会随着时间变化，所以最后也还是会执行。

高优先级cpu任务
低优先级cpu任务
低优先级io任务
低优先级cpu任务

## react17的扩展

### 对优先级的扩展。

从指定一个优先级到指定到指定一个连续的优先级区间。
仔细定义这个区间

## Scheduler（调度器）

调度任务的优先级，高优任务优先进入 **Reconciler**

### 了解react里的优先级 （和时间对应）

- 生命周期方法：同步执行
- 受控的用户输入：比如输入框内输入文字，同步执行

- 交互事件：比如动画，高优先级执行
- 其他：比如数据请求，低优先级执行

## 大致调度逻辑

### Fiber的结构

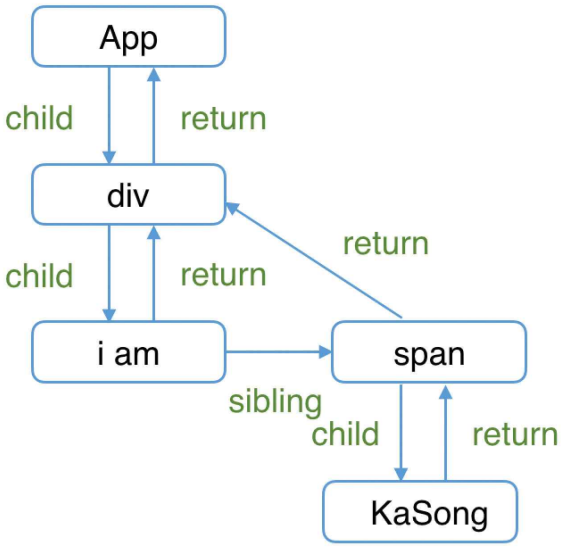
container._reactRootContainer => root
root._internalRoot  => fiberRoot
FiberRoot => FiberNode:

```
// switch(type){
    // case "functionComponent":
    //     ...
    // case "class":
    //     ...
// }
function FiberNode(){
  // Instance 作为静态数据结构的属性
  // Fiber对应组件的类型 Function/Class/Host
  this.tag = tag;
  // 标志你的这个节点的唯一性
  this.key = key;
  this.elementType = null;
  // 对于 FunctionComponent，指函数本身，对于ClassComponent，指class，对于HostComponent，指DOM节点tagName。
  this.type = null;
  // Fiber对应的真实DOM节点 <div></div>
  this.stateNode = null;
  // Fiber 用于连接其他Fiber节点形成Fiber树
  this.return = null;
  this.child = null;
  this.sibling = null;
  // 保存本次更新造成的状态改变相关信息
  this.pendingProps = pendingProps;
  this.memoizedProps = null;
  this.updateQueue = null;
  this.memoizedState = null;
  this.dependencies = null;
  // SideEffects, 标志更新的类型。删除，新增，更改属性
  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;
  // 调度优先级相关,以前这里是expirationTime.较之于优先级系统，有两个优势:
  // 1. 即较高优先级的IO约束任务会阻止较低优先级的CPU约束任务无法完成
```

```
  // 2. 能代表有限的一组的多个不同任务
  // 详情看这个文档：https://github.com/facebook/react/pull/18796
  this.lanes = NoLanes;
  this.childLanes = NoLanes;
  // 指向 workInProgress Fiber，其实就是上一次构建的Fiber镜像。
  this.alternate = null;
}
```

Fiber Tree

可以看到Fiber 与 Fiber之间是以链表的形式来连接的，这种结构可以方便中断。




**大致调度逻辑**

1. 根据优先级区分同步任务和异步任务，同步任务立即同步执行，最快渲染出来。异步任务走scheduler
2. 计算得到expirationTime，expirationTime = currentTime(当前时间) + timeout (不同优先级的时间间隔，时间越短，优先级越大)
3. 对比startTime和currentTime，将任务分为及时任务和延时任务。
4. 及时任务当即执行
5. 延时任务需要等到currentTime >= expirationTime的时候才会执行。
6. 及时任务执行完后，也会去判断是否有延时任务到了该执行之时，如果是，就执行延时任务
7. 每一批任务的执行在不同的宏任务中，不阻塞页面用户的交互

**调度具体代码分析**

1. 根据优先级区分同步任务和异步任务，同步任务立即同步执行，最快渲染出来。异步任务走scheduler

```
export function scheduleUpdateOnFiber(
```

```
  fiber: Fiber,
  lane: Lane,
  eventTime: number,
) {
  ...
   // 获得当前更新的优先级
  const priorityLevel = getCurrentPriorityLevel();
  // 同步任务，立即更新
  if (lane === SyncLane) {
    if (
      // 处于unbatchedUpdates， 且不在Renderer渲染阶段， 立即执行
      // Check if we're inside unbatchedUpdates
     // 执行上下文
      (executionContext & LegacyUnbatchedContext) !== NoContext &&
      // Check if we're not already rendering
      // CommitContext: 表示渲染到页面的那个逻辑
      (executionContext & (RenderContext | CommitContext)) === NoContext
    ) {
      // Register pending interactions on the root to avoid losing traced
interaction data.
      schedulePendingInteractions(root, lane);
      // This is a legacy edge case. The initial mount of a ReactDOM.render-ed
      // root inside of batchedUpdates should be synchronous, but layout updates
      // should be deferred until the end of the batch.
      performSyncWorkOnRoot(root);
    } else {
      ...
     // 包含异步调度逻辑，和中断逻辑
      ensureRootIsScheduled(root, eventTime);
    }
  } else {
    ...
    // Schedule other updates after in case the callback is sync.
    ensureRootIsScheduled(root, eventTime);
    schedulePendingInteractions(root, lane);
  }
  mostRecentlyUpdatedRoot = root;
}

// A: ensureRootIsScheduled  root.callbackNode已被赋值
// B: ensureRootIsScheduled  existingCallbackNode = root.callbackNode 是否存在
function ensureRootIsScheduled(root: FiberRoot, currentTime: number) {
  // root.callbackNode的存活周期是从ensureRootIsScheduled开始-->到commitRootImpl截止
  const existingCallbackNode = root.callbackNode;
```

```
  // 检查是否存在现有任务。 我们也许可以重用它。
  // Check if there's an existing task. We may be able to reuse it.
  if (existingCallbackNode !== null) {
    const existingCallbackPriority = root.callbackPriority;
    // 优先级没有改变。 我们可以重用现有任务，现有任务的优先级和下一个任务的优先级相同。比如input连续的输入，优先级相同，可以执行用之前的任务
    // 由于获取更新是从root开始，往下找到在这个优先级内的所有update.
    // 比如存在连续的setState，会执行这个逻辑，不会新建一个新的update
    // this.setState({
    //     "a": ""
    // });
    // this.setState({
    //    "b": ""
    // });
    // 不需要重新发起一个调度，用之前那个就可以了
    if (existingCallbackPriority === newCallbackPriority) {
      // The priority hasn't changed. We can reuse the existing task. Exit.
      return;
    }
    // 17 以前是判断 优先级的高低 lane
    // 优先级变了，先cancel掉，后续重新发起一个，// 中断逻辑
    // The priority changed. Cancel the existing callback. We'll schedule a new
    // one below.
    cancelCallback(existingCallbackNode);
  }
  // Schedule a new callback.
  // 发起一个新callBack
  let newCallbackNode;
  ...
  newCallbackNode = scheduleCallback(
      schedulerPriorityLevel,
      performConcurrentWorkOnRoot.bind(null, root),
  );
  root.callbackPriority = newCallbackPriority;
  // root.callbackNode的存活周期是从ensureRootIsScheduled开始——>到commitRootImpl截止
  root.callbackNode = newCallbackNode;
}
```

2. 计算得到expirationTime，expirationTime = currentTime(当前时间) + timeout (不同优先级的时间间隔，时间越短，优先级越大)

```
var currentTime = getCurrentTime();
  // 得到startTime，根据优先级的不同分别加上不同的间隔时间，构成expirationTime；当expirationTime越接近真实的时间，优先级越高
```

```
  // 根据startTime 是否大于当前的currentTime，将任务分为了及时任务和延时任务。延时任务还不会立即执行，它会在currentTime接近startTime的时候，才会执行
  var startTime;
  if (typeof options === 'object' && options !== null) {
    var delay = options.delay;
    if (typeof delay === 'number' && delay > 0) {
      startTime = currentTime + delay;
    } else {
      startTime = currentTime;
    }
  } else {
    startTime = currentTime;
  }
  var timeout;
  // 根据优先级增加不同的时间间隔
  switch (priorityLevel) {
    // ImmediatePriority = -1;
    case ImmediatePriority:
      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
      break;
    case UserBlockingPriority:
      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
      break;
    case IdlePriority:
      timeout = IDLE_PRIORITY_TIMEOUT;
      break;
    case LowPriority:
      timeout = LOW_PRIORITY_TIMEOUT;
      break;
    case NormalPriority:
    default:
      timeout = NORMAL_PRIORITY_TIMEOUT;
      break;
  }
  var expirationTime = startTime + timeout;
```

3. 对比startTime和currentTime，将任务分为及时任务和延时任务

```
if (startTime > currentTime) {
    push(timerQueue, newTask);
    // 当没有及时任务的时候
      // Schedule a timeout.
      // 在间隔时间之后，调用一个handleTimeout，主要作用是把timerQueue的任务加到taskQueue队列里来，然后调用requestHostCallback
      // 执行那个延时任务
```

```
      // setTimeOut(handleTimeout, startTime - currentTime)
      requestHostTimeout(handleTimeout, startTime - currentTime);
    }
  } else {
    push(taskQueue, newTask);
    // Schedule a host callback, if needed. If we're already performing work,
    // wait until the next time we yield.
    if (!isHostCallbackScheduled && !isPerformingWork) {
      isHostCallbackScheduled = true;
       // 这里会调度及时任务
      requestHostCallback(flushWork);
    }
  }
  return newTask;
}
```

4. 及时任务当即执行,但是为了不阻塞页面的交互，因此在宏任务中执行

```
// 第一次调用 scheduleCallback
 //    1.  把任务放在timeQueue 不会立即执行,等待
// 第二次调用 scheduleCallback
//      1.  把任务放在TaskQuene
//      2.  执行new MessageChannel() 的 port.postMessage。如果主线程还有任务，那就还不会到performWorkUntilDeadline。
// 第三次调用 scheduleCallback
//      1.1. 把任务放在TaskQuene
//      2. 执行new MessageChannel() 的 port.postMessage
// 主线程没有任务
// 执行微任务
// 执行宏任务列表
// 执行第二次调用发起的performWorkUntilDeadline
// performWorkUntilDeadline 去取得TaskQuene中的任务，发起 PerformanceSyncWorkOnRoot
// 判断TimeQueue中是否有到期的任务，如果有就加到TaskQuene来
// 主线程了
// 微任务
// 下一个宏任务
const channel = new MessageChannel();
  const port = channel.port2;
  channel.port1.onmessage = performWorkUntilDeadline;
// 在MessageChannel宏任务里执行真正的调度逻辑，可以保证任务与任务之间不是连续执行的，这样就不会因为要一次性执行的任务多而阻塞用户的操作
  requestHostCallback = function(callback) {
    scheduledHostCallback = callback;
    if (!isMessageLoopRunning) {
      isMessageLoopRunning = true;
```

```
      port.postMessage(null);
      // setTimeout(() => {
      //  进入宏任务
      // }, 0)
    }
  };
```

5. 延时任务需要等到currentTime >= expirationTime的时候才会执行。每次调度及时任务的时候，都会去判断延时任务的执行时间是否到了，如果判断为true，则添加到及时任务中来。

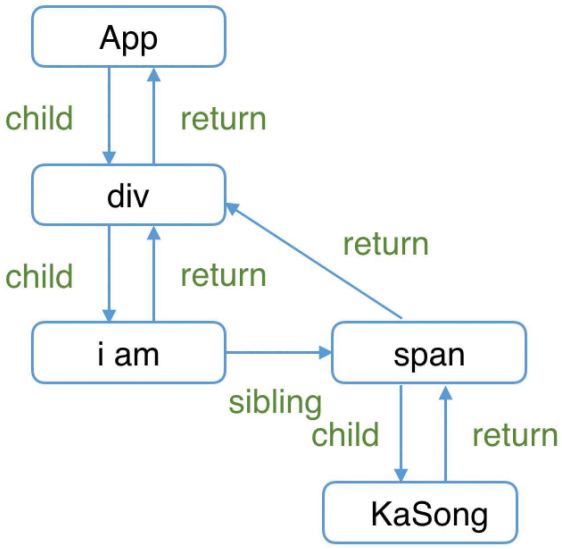
```
function advanceTimers(currentTime) {
  // Check for tasks that are no longer delayed and add them to the queue.
  let timer = peek(timerQueue);
  while (timer !== null) {
    if (timer.callback === null) {
      // Timer was cancelled.
      pop(timerQueue);
    } else if (timer.startTime <= currentTime) {
      // Timer fired. Transfer to the task queue.
      pop(timerQueue);
      timer.sortIndex = timer.expirationTime;
      // 把timerQueue中的添加到taskQueue中来
      push(taskQueue, timer);
      if (enableProfiling) {
        markTaskStart(timer, currentTime);
        timer.isQueued = true;
      }
    } else {
      // Remaining timers are pending.
      return;
    }
    timer = peek(timerQueue);
  }
}
```

## Reconciler（协调器）

主要作用是负责找出变化的组件。在react16以上，为了方便打断，数据结构几乎都是链表的格式。会做dom-diff。也会把dom元素生成。但是并不会渲染到页面。而是先打上一个标记。
等在下一个commit阶段才会真正的渲染到页面。

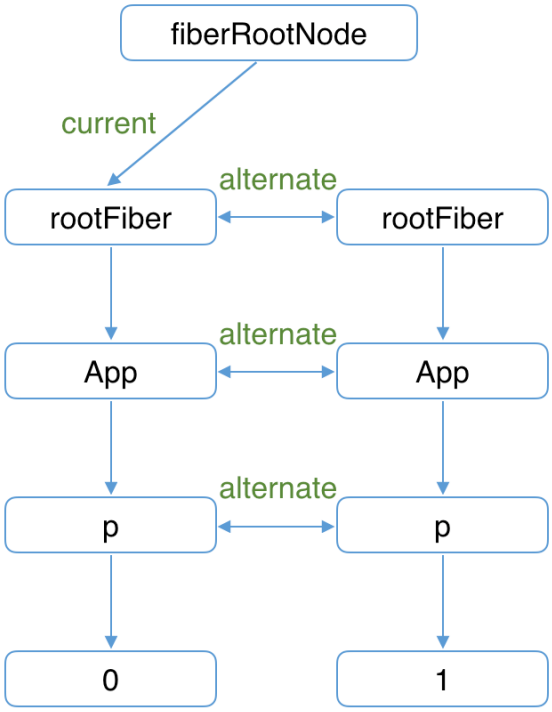
### 大致的找出变化的组件的逻辑

react发生一次更新的时候，比如ReactDOM.render/setState，都会从Fiber Root开始从上往下遍历，然后逐一找到变化的节点。构建完成会形成一颗Fiber Tree。在react内部会同时存在两棵`Fiber树`。




### 双缓存结构

在`React`中最多会同时存在两棵`Fiber树`。当前屏幕上显示内容对应的`Fiber树`称为`current Fiber树`，正在内存中构建的`Fiber树`称为`workInProgress Fiber树`。
`current Fiber树`中的`Fiber节点`被称为`current fiber`，`workInProgress Fiber树`中的`Fiber节点`被称为`workInProgress fiber`，他们通过`alternate`属性连接。
如果之前没有Fiber Tree就逐级创建Fiber Tree；如果存在Fiber Tree，会构建一个WorkInProgress Tree, 这个tree的Fiber节点可以复用Current Tree上没有发生变化的节点数据。




**为什么是双缓存结构?**

1. 可以很快的找到之前对应的Fiber
2. 在某些情况下可以直接复用fiber
3. 更新完毕后 current直接指向workInProgress root，完成了Fiber tree的更新

## 构建Fiber Tree

```
return <div>
i am
<span>kaSong</span>
</div>
```

Reconciler的代码大致从 renderRootSync 函数开始，从优先级最高的Fiber Root开始递归。

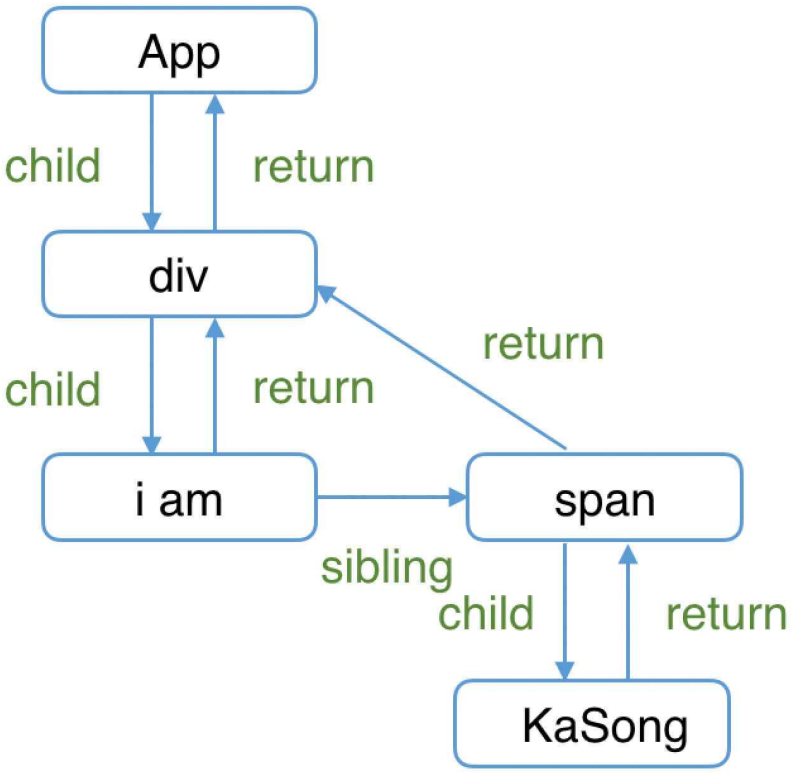
```
function workLoopSync() {
  // Already timed out, so perform work without checking if we need to yield.
  // workInProgress:当前正在处理的节点
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
    // if (next === null) {
        // If this doesn't spawn new work, complete the current work.
        // completeUnitOfWork(unitOfWork);
        //do{
            // sibliing存在的时候,  return
            // xxx
        // } while (completedWork !== null);
    // }
```

```
  }
}

function performUnitOfWork(unitOfWork: Fiber): void {
  // The current, flushed, state of this fiber is the alternate. Ideally
  // nothing should rely on this, but relying on it here means that we don't
  // need an additional field on the work in progress.
  const current = unitOfWork.alternate;
  ...
  // 会创建一个Fiber，赋值给workInProgress.child并返回workInProgress.child.注    意，sibilig Fiber，此处暂时还没处理
  // 返回 next = workInProgress.child
  next = beginWork(current, unitOfWork, subtreeRenderLanes);
  ...
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next === null) {
    // If this doesn't spawn new work, complete the current work.
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
  ReactCurrentOwner.current = null;
}

第一轮：调用 beginWork创建出App，workInProgress = App;
第二轮：
```

在beginWork函数里：只创建了这副图中的App， div， iam 3个Fiber。当没有子节点时，进入到completeUnitOfWork里执行，执行completeUnitOfWork后如果存在兄弟Fiber节点，就从兄弟Fiber节点这里接着执行BeginWork.执行完了，又接着执行completeUnitOfWork。这就是Fiber Tree构建的整体流程。




介绍了Fiber Tree 的整体流程，接下来我们可以稍微更具体来看下beginWork和completeWork这两个函数还做了哪些事情。由于beginWork里的情况特别多，我们选取其中一种来加以分析，其他的也都差不多。

### beginWork

1. 判断Fiber 节点是否可以复用
2. 根据不同的Tag，生成不同的Fiber节点（调用reconcileChildren）
    a. Mount 阶段：创建Fiber 节点
    b. Update阶段 和现在的Fiber节点做对比，生成新的Fiber节点
        i. 单节点Diff
        ii. 多节点Diff
3. 给存在变更的Fiber节点打上标记 newFiber.flags = Placement|Update|Deletion|...
4. 创建的Fiber节点赋给WorkInProgress.child,返回WorkInProgress.child. 继续下一次的循环

```
// {} === {} false
// extra = <div> extra </div>;
// render() => {
// return <div>
  //    i'm
  //   {this.extra}
  //   <Extra />
  // </div>
// }
```

```
// function Extra(){
//     return <div> extra </div>
// }

function beginWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes,
): Fiber | null {
  const updateLanes = workInProgress.lanes;
  // current tree 存在，不是初次构建
  if (current !== null) {
    const oldProps = current.memoizedProps;
    const newProps = workInProgress.pendingProps;
    if (
      oldProps !== newProps ||
      hasLegacyContextChanged()
    ) {
      // If props or context changed, mark the fiber as having performed work.
      // This may be unset if the props are determined to be equal later (memo).
      didReceiveUpdate = true;
    } else if (!includesSomeLane(renderLanes, updateLanes)) {
      // 更新的优先级和current tree的优先级是否有一致的，不一致才触发bailout
      didReceiveUpdate = false;
      // This fiber does not have any pending work. Bailout without entering
      // the begin phase. There's still some bookkeeping we that needs to be done
      // in this optimized path, mostly pushing stuff onto the stack.
      ...
      // didReceiveUpdate = false; 可以复用上次的fiber
      return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
    } else {
      if ((current.flags & ForceUpdateForLegacySuspense) !== NoFlags) {
        // This is a special case that only exists for legacy mode.
        // See https://github.com/facebook/react/pull/19216.
        didReceiveUpdate = true;
      } else {
        // An update was scheduled on this fiber, but there are no new props
        // nor legacy context. Set this to false. If an update queue or context
        // consumer produces a changed value, it will set this to true. Otherwise,
        // the component will assume the children have not changed and bail out.
        didReceiveUpdate = false;
      }
    }
  } else {
```

```
    didReceiveUpdate = false;
  }
  // Before entering the begin phase, clear pending update priority.
  // TODO: This assumes that we're about to evaluate the component and process
  // the update queue. However, there's an exception: SimpleMemoComponent
  // sometimes bails out later in the begin phase. This indicates that we should
  // move this assignment out of the common path and into each branch.
  workInProgress.lanes = NoLanes;
  switch (workInProgress.tag) {
    case IndeterminateComponent:
    ...
    case LazyComponent:
    ...
    case FunctionComponent:
      const Component = workInProgress.type;
      const unresolvedProps = workInProgress.pendingProps;
      const resolvedProps =
        workInProgress.elementType === Component
          ? unresolvedProps
          : resolveDefaultProps(Component, unresolvedProps);
      // 1. 调用renderWithHooks方法，注入hooks上下文，执行function函数体
      // 2. 判断节点是否可以复用，能复用则调bailoutHooks方法复用节点
      // 3. 设置flags
      // 4. 调用 reconcileChildren，得到子Fiber
      return updateFunctionComponent(
        current,
        workInProgress,
        Component,
        resolvedProps,
        renderLanes,
      );
    case ClassComponent: {
      const Component = workInProgress.type;
      const unresolvedProps = workInProgress.pendingProps;
      const resolvedProps =
        workInProgress.elementType === Component
          ? unresolvedProps
          : resolveDefaultProps(Component, unresolvedProps);
      // 执行render()等生命周期，reconcileChildren
      return updateClassComponent(
        current,
        workInProgress,
        Component,
        resolvedProps,
```

```
        renderLanes,
      );
    }
    // ReactDOM.render(<App />)
    case HostRoot:
      // 会调到reconcileChildren
      return updateHostRoot(current, workInProgress, renderLanes);
     }
    ...
}


export function reconcileChildren(
  current: Fiber | null,
  workInProgress: Fiber,
  nextChildren: any,
  renderLanes: Lanes,
) {
  if (current === null) {
    // If this is a fresh new component that hasn't been rendered yet, we
    // won't update its child set by applying minimal side-effects. Instead,
    // we will add them all to the child before it gets rendered. That means
    // we can optimize this reconciliation pass by not tracking side-effects.
    // Mount阶段，创建Fiber节点
    workInProgress.child = mountChildFibers(
      workInProgress,
      null,
      nextChildren,
      renderLanes,
    );
  } else {
    // If the current child is the same as the work in progress, it means that
    // we haven't yet started any work on these children. Therefore, we use
    // the clone algorithm to create a copy of all the current children.
    // If we had any progressed work already, that is invalid at this point so
    // let's throw it out.
    // Update节点，diff后更新Fiber节点
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren,
      renderLanes,
    );
  }
```

```
}
 //reconcileChildFibers里会判断变更的类型是什么？比如有新增，删除，更新等类型。每一种类型的变更，调用不同的方法，赋予flags一个值.在commit阶段，会直接根据flags来做dom操作。
  function placeSingleChild(newFiber: Fiber): Fiber {
    // This is simpler for the single child case. We only need to do a
    // placement for inserting new children.
    if (shouldTrackSideEffects && newFiber.alternate === null) {
      newFiber.flags = Placement;
    }
    return newFiber;
  }
```

### diff算法

#### diff的瓶颈以及React如何应对

React Diff 会预设几个规则：

1. 只对同级节点，进行比较.
2. 节点变化，直接删除，然后重建
3. 存在key值，对比key值一样的节点

```
// 代码在ReactChildFiber.new.js 下 reconcileChildFibers函数
// 判断节点是不是react 节点
    // Handle object types
    const isObject = typeof newChild === 'object' && newChild !== null;
    if (isObject) {
      // 根据不同的类型，处理不同的节点对比
      switch (newChild.$$typeof) {
        case REACT_ELEMENT_TYPE:
          return placeSingleChild(
            reconcileSingleElement(
              returnFiber,
              currentFirstChild,
              newChild,
              lanes,
            ),
          );
        ...
      }
    }
    if (typeof newChild === 'string' || typeof newChild === 'number') {
      return placeSingleChild(
        reconcileSingleTextNode(
          returnFiber,
```

```
          currentFirstChild,
          '' + newChild,
          lanes,
        ),
      );
    }
    // 多节点数组
    if (isArray(newChild)) {
      return reconcileChildrenArray(
        returnFiber,
        currentFirstChild,
        newChild,
        lanes,
      );
    }
```

**单节点diff**

1. 判断存在对应节点，key值是否相同，节点类型一致，可以复用
2. 存在对应节点，key值是否相同，节点类型不一致，标记删除
3. 存在对应节点，key值不同，标记删除
4. 不存在对应节点，创建新节点

```
function reconcileSingleElement(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
    element: ReactElement,
    lanes: Lanes,
  ): Fiber {
    const key = element.key;
    let child = currentFirstChild;
    // 是否存在对应节点
    while (child !== null) {
      // 比较key是否相同
      if (child.key === key) {
        switch (child.tag) {

          default: {
            // 节点类型一致，可以复用
            if (
              child.elementType === element.type) {
              deleteRemainingChildren(returnFiber, child.sibling);
              const existing = useFiber(child, element.props);
              existing.ref = coerceRef(returnFiber, child, element);
              existing.return = returnFiber;
```

```
              return existing;
            }
            break;
          }
        }
        // div => p
        // 节点类型不一致才会到这里，标记为删除
        // Didn't match.
        deleteRemainingChildren(returnFiber, child);
        break;
      } else {
        // key不同，将该fiber标记为删除 flags
        deleteChild(returnFiber, child);
      }
      child = child.sibling;
    }
    // 不存在对应节点，创建
   const created = createFiberFromElement(element, returnFiber.mode, lanes);
   created.ref = coerceRef(returnFiber, currentFirstChild, element);
   created.return = returnFiber;
   return created;
  }
```

### 多节点diff

```
// 1. 对比新旧children相同index的对象的key是否相等，如果是，返回该对象，如果不是，返回null
// 2. key值不等，不用对比下去了，节点不能复用，跳出
// 3. 判断节点是否存在移动，存在则返回新位置
// 4. 但可能存在新的数组小于老数组的情况，即老数组后面有剩余的，所以要删除
// 5. 新数组存在新增的节点，创建新阶段
// 6. 创建一个existingChildren代表所有剩余没有匹配掉的节点，然后新的数组根据key从这个 map 里面查找，如果有则复用，没有则新建
function reconcileChildrenArray(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
    newChildren: Array<*>,
    lanes: Lanes,
  ): Fiber | null {
    let resultingFirstChild: Fiber | null = null;
    let previousNewFiber: Fiber | null = null;
    let oldFiber = currentFirstChild;
    let lastPlacedIndex = 0;
    let newIdx = 0;
    let nextOldFiber = null;
```

```
    for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
      // oldIndex 大于 newIndex，那么需要旧的 fiber 等待新的 fiber，一直等到位置相同
      if (oldFiber.index > newIdx) {
        nextOldFiber = oldFiber;
        oldFiber = null;
      } else {
        nextOldFiber = oldFiber.sibling;
      }
      // 对比新旧children相同index的对象的key是否相等，如果是，返回该对象，如果不是，返回null
      const newFiber = updateSlot(
        returnFiber,
        oldFiber,
        newChildren[newIdx],
        lanes,
      );
      // key值不等，不用对比下去了，节点不能复用，跳出
      if (newFiber === null) {
        if (oldFiber === null) {
          oldFiber = nextOldFiber;
        }
        break;
      }
      // 判断节点是否存在移动，存在则返回新位置
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
      oldFiber = nextOldFiber;
    }
    // 第一个循环完毕
    // newIdx === newChildren.length 说明中途没有跳出的情况
    // 但可能存在新的数组小于老数组的情况，即老数组后面有剩余的，所以要删除
    if (newIdx === newChildren.length) {
      // We've reached the end of the new children. We can delete the rest.
      deleteRemainingChildren(returnFiber, oldFiber);
      return resultingFirstChild;
    }
    // oldFiber === null 说明数组中所有的都可以复用
    if (oldFiber === null) {
      // 新数组存在新增的节点，创建新阶段
      for (; newIdx < newChildren.length; newIdx++) {
        const newFiber = createChild(returnFiber, newChildren[newIdx], lanes);
        if (newFiber === null) {
          continue;
        }
        // 添加到sibiling
```

```
62          lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
63        }
64        return resultingFirstChild;
65      }
66      // 创建一个existingChildren代表所有剩余没有匹配掉的节点，然后新的数组根据key从这个
    map 里面查找，如果有则复用，没有则新建
67      const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
68      // Keep scanning and use the map to restore deleted items as moves.
69      for (; newIdx < newChildren.length; newIdx++) {
70          // 对比是否还有可以复用的
71        const newFiber = updateFromMap(
72          existingChildren,
73          returnFiber,
74          newIdx,
75          newChildren[newIdx],
76          lanes,
77        );
78        if (newFiber !== null) {
79          // 判断节点是否存在移动，存在则返回新位置
80          lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
81        }
82      }
83      return resultingFirstChild;
84    }
```

~~打上Effect Tag~~ flags

## completeUnitOfWork

## commit阶段——负责将变化的组件渲染到页面上

分为3个阶段：

### commitBeforeMutationEffects （DOM操作前）

1. 处理`DOM节点`渲染/删除后的`autoFocus`、`blur`逻辑。
2. 调用`getSnapshotBeforeUpdate`生命周期钩子。
3. 调度`useEffect`。

```
1  class A extrends Component{
2      getSnapshotBeforeUpdate(){
3            console.log(1);
4      }
5      render(){
```

```
        return <B> </B>
    }
}
class B extrends Component{
    getSnapshotBeforeUpdate(){
        console.log(2);
    }
    render(){
        return <div> </div>
    }
}

function commitBeforeMutationEffects(firstChild: Fiber) {
  let fiber = firstChild;
  while (fiber !== null) {
    // 处理需要删除的fiber autoFocus、blur 逻辑。
    if (fiber.deletions !== null) {
      commitBeforeMutationEffectsDeletions(fiber.deletions);
    }
    // 递归调用处理子节点
    if (fiber.child !== null) {
       commitBeforeMutationEffects(fiber.child);
    }
    try {
        // 调用 getSnapshotBeforeUpdate 生命周期
        // 异步调度useEffect
        commitBeforeMutationEffectsImpl(fiber);
      } catch (error) {
        captureCommitPhaseError(fiber, fiber.return, error);
      }    // 返回兄弟节点，接着循环
    fiber = fiber.sibling;
  }
}

// 在beginWork里，存在getSnapshotBeforeUpdate的时候，fiber.flags |= Snapshot;
function commitBeforeMutationEffectsImpl(fiber: Fiber) {
  const current = fiber.alternate;
  const flags = fiber.flags;
  if ((flags & Snapshot) !== NoFlags) {
    // 调用 getSnapshotBeforeUpdate 生命周期
    commitBeforeMutationEffectOnFiber(current, fiber);
  }
  // tags
  if ((flags & Passive) !== NoFlags) {
```

```
    // If there are passive effects, schedule a callback to flush at
    // the earliest opportunity.
    if (!rootDoesHavePassiveEffects) {
      rootDoesHavePassiveEffects = true;
      // 异步调度 useEffect 的回调并不是在 dom渲染前执行的。
      // useLayoutEffect  在dom操作后同步执行回调
      // useEffect  异步执行回调。不想阻塞主线程。可以先尝试使用useEffect。不行的话再使用useLayoutEffect。
      scheduleCallback(NormalSchedulerPriority, () => {
        flushPassiveEffects();
        return null;
      });
    }
}
```

## commitMutationEffects(执行DOM操作)

1. 遍历finishedWork，执行DOM操作
2. 对于删除的组件，会执行componentWillUnMount生命周期

```
// 结构和上面的一样
function commitMutationEffects(
  firstChild: Fiber,
  root: FiberRoot,
  renderPriorityLevel: ReactPriorityLevel,
) {
  let fiber = firstChild;
  while (fiber !== null) {
    const deletions = fiber.deletions;
    if (deletions !== null) {
      // 需要卸载的组件，会调用componentWillUnMount
      commitMutationEffectsDeletions(
        deletions,
        fiber,
        root,
        renderPriorityLevel,
      );
    }
    if (fiber.child !== null) {
      // 仍然是递归调用处理子Fiber
      commitMutationEffects(fiber.child, root, renderPriorityLevel);
    }
   try {
        // 区分不同情Flag 执行不同的Dom操作
```

```
        // 已经构造好了dom元素了，存放在stateNode节点的。新增，删除，替换?
        commitMutationEffectsImpl(fiber, root, renderPriorityLevel);
        // 页面就终于渲染出来了
      } catch (error) {
        captureCommitPhaseError(fiber, fiber.return, error);
      }
    // 处理兄弟节点
    fiber = fiber.sibling;
  }
}
```

## recursivelyCommitLayoutEffects(DOM操作后)

在这个阶段前current tree也发生变化了，指向了最新构建的workInProgress tree。

1. `layout阶段`也是深度优先遍历`effectList`，调用生命周期，didMount/didUpdate；执行useEffect
2. 赋值 ref
3. 处理ReactDom.render 回调

```
// A,B两个组件，生命周期getSnapshotBeforeUpdate，componentDidMount，componentWillMount，
// componentWillUnMount 问整体的执行顺序?
// 哪几个是在commit阶段执行的? 哪几个是在BeginWork里执行的?
class A extrends Component{
    componentDidMount(){
          console.log(1);
    }
    componentDidUpdate(){
          console.log(1);
    }
    render(){
        return <B> </B>
    }
}
class B extrends Component{
    componentDidMount(){
          console.log(2);
    }
    componentDidUpdate(){
          console.log(2);
    }
    render(){
        return <div> </div>
    }
```

```
}

ReactDOM.render(<App/>, $("#app"), () => {
    console.log('');
});


function recursivelyCommitLayoutEffects(
  finishedWork: Fiber,
  finishedRoot: FiberRoot,
) {
  const {flags, tag} = finishedWork;
  switch (tag) {
    ...
    default: {
      let child = finishedWork.child;
      while (child !== null) {
        ...
        try {
           // 仍然是递归
          recursivelyCommitLayoutEffects(child, finishedRoot);
        } catch (error) {
          captureCommitPhaseError(child, finishedWork, error);
        }
        child = child.sibling;
      }
      const primaryFlags = flags & (Update | Callback);
      if (primaryFlags !== NoFlags) {
        switch (tag) {
          case FunctionComponent:
          case ForwardRef:
          case SimpleMemoComponent:
          case Block: {
            ...
            // 执行useEffect的回调
            commitHookEffectListMount(
              HookLayout | HookHasEffect,
              finishedWork,
            );
            break;
          }
          case ClassComponent: {
            // 执行componentDidMount/didUpdate 生命周期
            // NOTE: Layout effect durations are measured within this function.
```

```
            commitLayoutEffectsForClassComponent(finishedWork);
            break;
          }
          ...
      }
        ...
      // 赋值ref
        if (flags & Ref && tag !== ScopeComponent) {
          commitAttachRef(finishedWork);
        }
      break;
    }
  }
}
```

## 更新流程

### ReactDOM.render流程

是走的unBatchUpdate ，所以是没有走schedule调度的，直接就到了Reconciler阶段了。
unBatchUpdate 不批处理
batchUpdate 批处理

### this.setState流程

## Hooks源码解读

见ppt。

## React 17的扩展

### 对优先级的扩展

为了解决react16的不足：
1. 高优先级IO操作会阻塞低优先级CPU操作
2. 只能指定一个优先级

升级为从指定一个优先级到指定到指定一个连续的优先级区间。扩展了原本优先级，可支持的优先级更多，同时也可以指定多个优先级为当前优先级
https://github.com/facebook/react/pull/18796

### 剥离了JSX

可以单独引入jsx，也可以不再引用React这整个空间

https://zh-hans.reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html