介绍

初识流程图

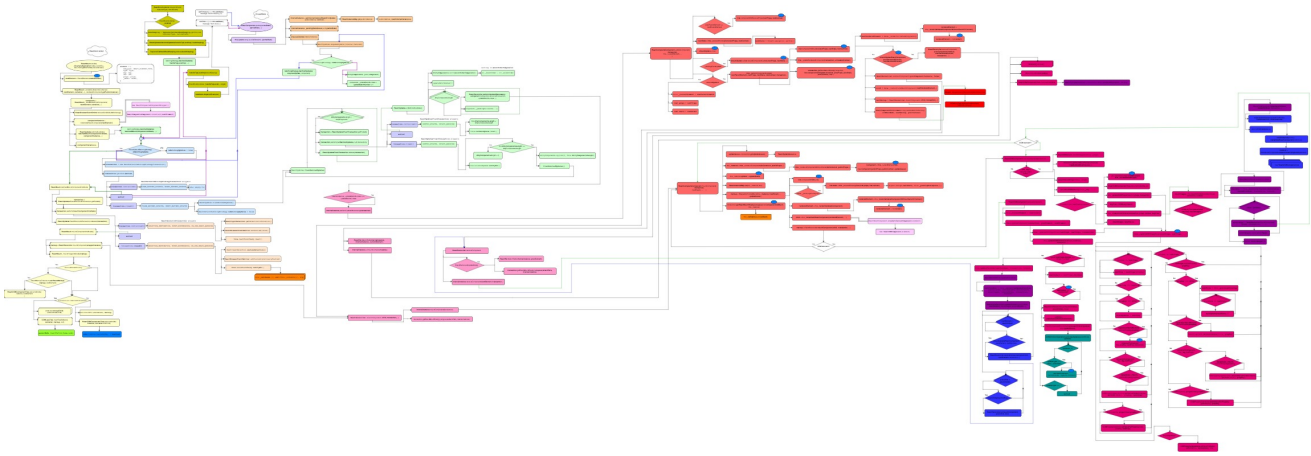


图 介绍-0：整体流程

你可以先花点时间看下整体的流程。虽然看起来很复杂，但它实际上只描述了两个流程：(组件的)挂载和更新。我跳过了卸载，因为它是一种“反向挂载”，而且删除这部分简化了流程图。另外，**这图并不是100%** 同源代码匹配，而只是描述架构的主要部分。总体来说，它大概是源代码的 60%，而另外的 40% 没有多少视觉价值，为了简单起见，我省略了那部分。

乍一看，你可能会注意到流程图中有很多颜色。每个逻辑项（流程图上的形状）都以其父模块的颜色高亮显示。例如，如果是从红色的 `模块 B` 调用 `方法 A`，那 `方法 A` 也是红色的。以下是流程图中模块的图例以及每个文件的路径。

src\renderers\dom\client**ReactMount**.js

src\renderers\shared\stack\reconciler**ReactUpdates**.js

src\renderers\shared\stack\reconciler**instantiateReactComponent**.js

src\renderers\shared\stack\reconciler**ReactDefaultBatchingStrategy**.js

src\renderers\shared\utils**Transaction**.js

src\renderers\dom\client**ReactReconcileTransaction**.js

src\renderers\shared\stack\reconciler**ReactReconciler**.js

src\renderers\dom\client\utils**DOMLazyTree**.js

src\renderers\dom\client**ReactDOMComponentTree**.js

src\renderers\shared\stack\reconciler**ReactCompositeComponent**.js

src\renderers\dom\shared**ReactDOMComponent**.js

src\renderers\shared\stack\reconciler**ReactMultiChild**.js

src\renderers\shared\utils**CallbackQueue**.js

src\renderers\shared\stack\reconciler**ReactChildReconciler**.js

src\isomorphic\modern\class**ReactComponent**.js

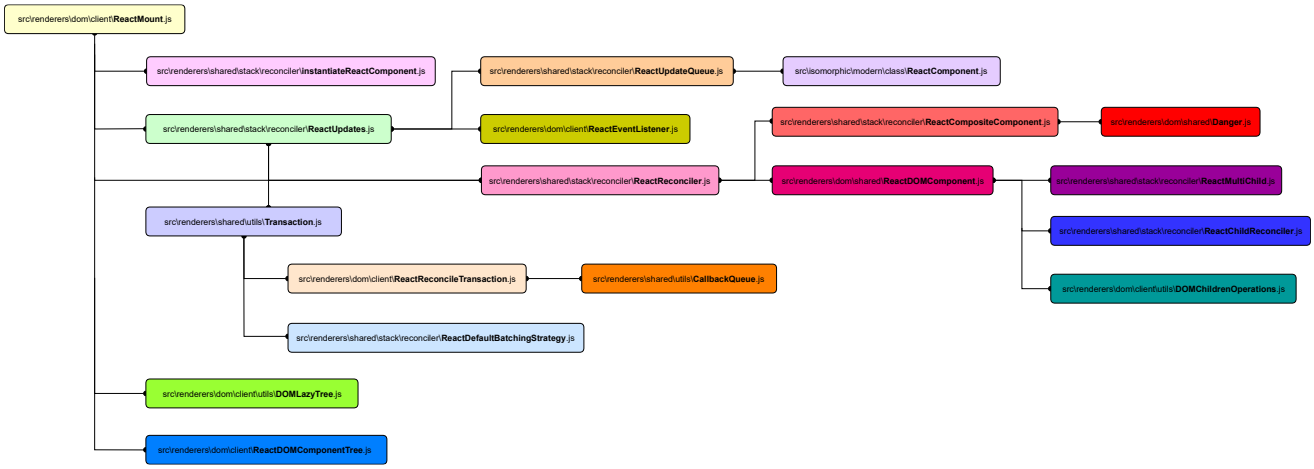
src\renderers\shared\stack\reconciler**ReactUpdateQueue**.js

src\renderers\dom\client**ReactEventListener**.js

src\renderers\dom\client\utils**DOMChildrenOperations**.js

src\renderers\dom\shared**Danger**.js

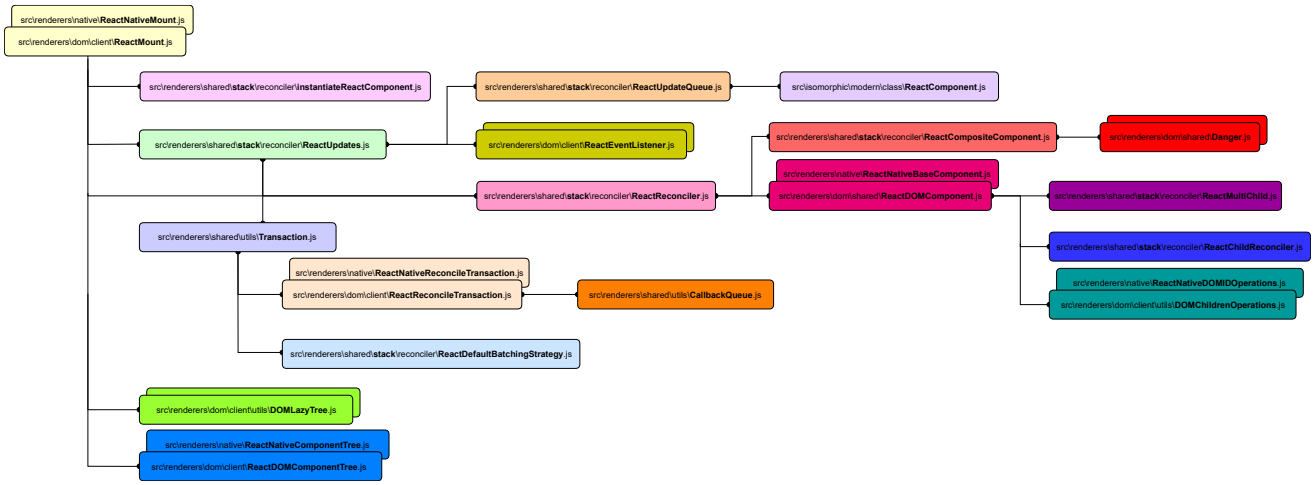
让我们把它们放在一张流程图中，看看**模块之间的依赖关系。**



你可能知道，React 是为**支持多种环境**而构建的。

- 移动端（**ReactNative**）
- 浏览器（**ReactDOM**）
- 服务端渲染
- **ReactART**（使用 React 绘制矢量图形）
- 其它

因此，一些文件实际上比上面流程图中列出的要更大。以下是包含多环境支持的相同的流程图。



如你所见，有些项似乎翻倍了。这表明它们对每个平台都有一个独立的实现。让我们来看一些简单例子，例如 ReactEventListener，显然，不同平台会有不同的实现。从技术上讲，你可以想象，这些依赖于平台的模块，应该以某种方式注入或连接到当前的逻辑流程中。实际上有很多这样的注入器，因为它们的用法是标准组合模式的一部分。同样，为了简单起见，我选择忽略它们。

让我们来学习下**常规浏览器**中 **React DOM** 的逻辑流程。这是最常用的平台，并完全覆盖了所有 React 的架构设计理念。

代码示例

学习框架或者库的源码的最佳方式是什么？没错，研读并调试源码。那好，我们将要调试这**两个流程**：**ReactDOM.render** 和 **component.setState** 这两者对应了组件的挂载和更新。让我们来看一下我们能编写一些什么样的代码来开始学习。我们需要什么呢？或许几个具有简单渲染的小组件就可以了，因为更容易调试。

```
class ChildCmp extends React.Component {
    render() {
        return <div> {this.props.childMessage} </div>
    }
}

class ExampleApplication extends React.Component {
    constructor(props) {
        super(props);
        this.state = {message: 'no message'};
    }

    componentWillMount() {
        //...
    }

    componentDidMount() {
        /* setTimeout(()=> {
            this.setState({ message: 'timeout state message' });
        }, 1000); */
    }

    shouldComponentUpdate(nextProps, nextState, nextContext) {
        return true;
    }

    componentDidUpdate(prevProps, prevState, prevContext) {
        //...
    }

    componentWillReceiveProps(nextProps) {
        //...
    }

    componentWillUnmount() {
        //...
    }

    onClickHandler() {
        /* this.setState({ message: 'click state message' }); */
    }

    render() {
        return <div>
            <button onClick={this.onClickHandler.bind(this)}> set state button </button>
            <ChildCmp childMessage={this.state.message} />
            And some text as well!
        </div>
    }
}

ReactDOM.render(
    <ExampleApplication hello={'world'} />,
```

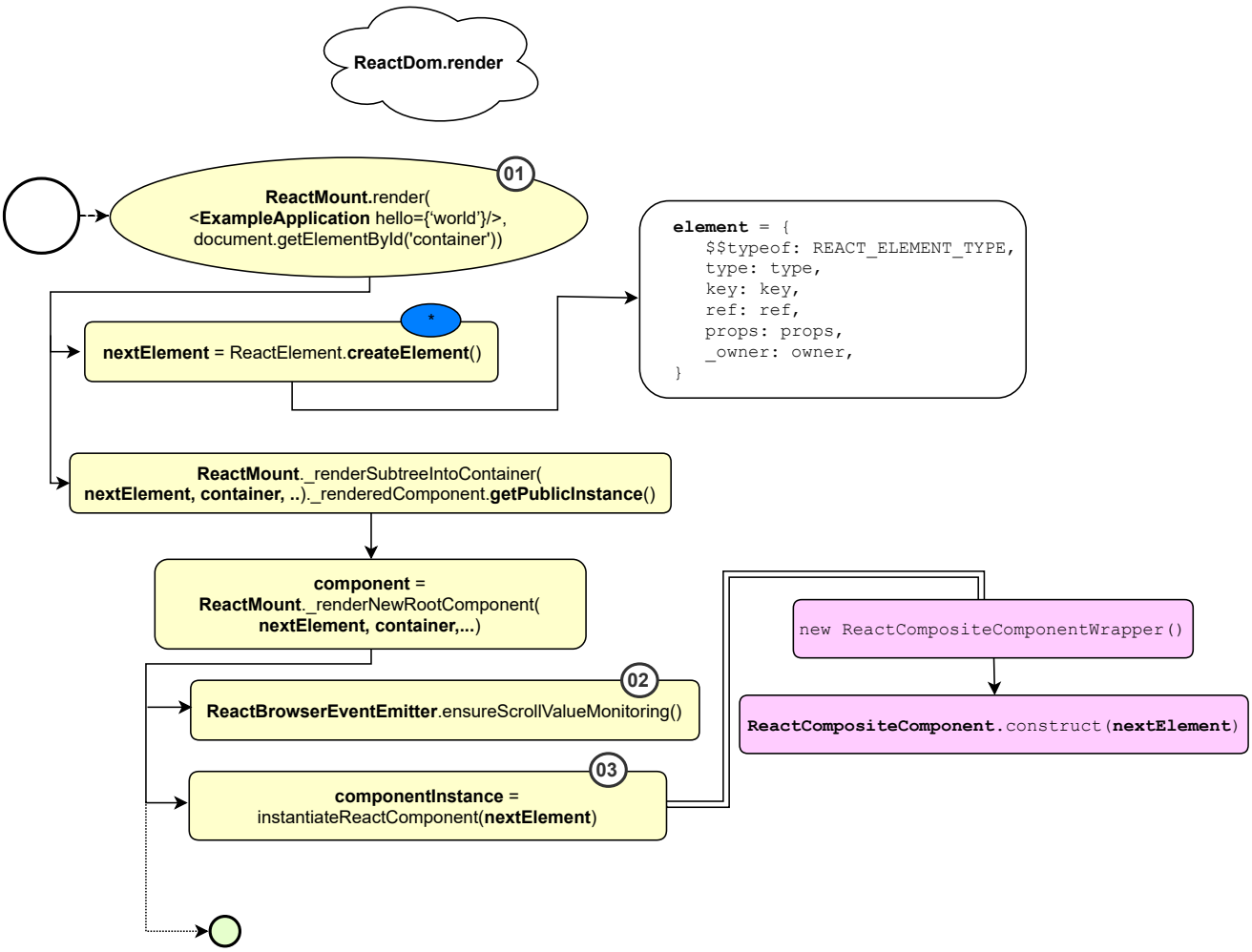
```
    document.getElementById('container'),
    function() {}
);
```

我们已经准备好开始学习了。让我们先来分析流程图中的第一部分。一个接一个，我们会将它们全部分析完。

下一页：Part 0 >>

首页

第 0 部分



ReactDOM.render

让我们从 ReactDOM.render 的调用开始。

入口点是 ReactDom.render，我们的应用程序是从这里开始渲染到 DOM 中的。为了方便调试，我创建了一个简单的 `<ExampleApplication />` 组件。因此，发生的第一件事就是 **JSX 会被转换成 React 元素(element)**。它们是简单的、直白的对象。具有简单的结构。它们仅仅展示从本组件渲染中返回的内容，没有其他了。一些字段应该是你已经熟悉的，像 props、key 和 ref。属性类型是指由 JSX 描述的标记对象。所以，在我们的例子中，它就是 `ExampleApplication` 类，但是它也可以仅仅是 Button 标签的 `button` 字符串等其他类。另外，在 React 组件创建过程中，它会将 `defaultProps` 与 `props` 合并（如果显式声明了），并验证 `propTypes`。

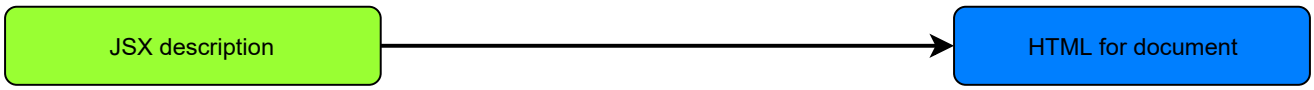
更多详细信息可参考源码：`src\isomorphic\classic\element\ReactElement.js`。

ReactMount

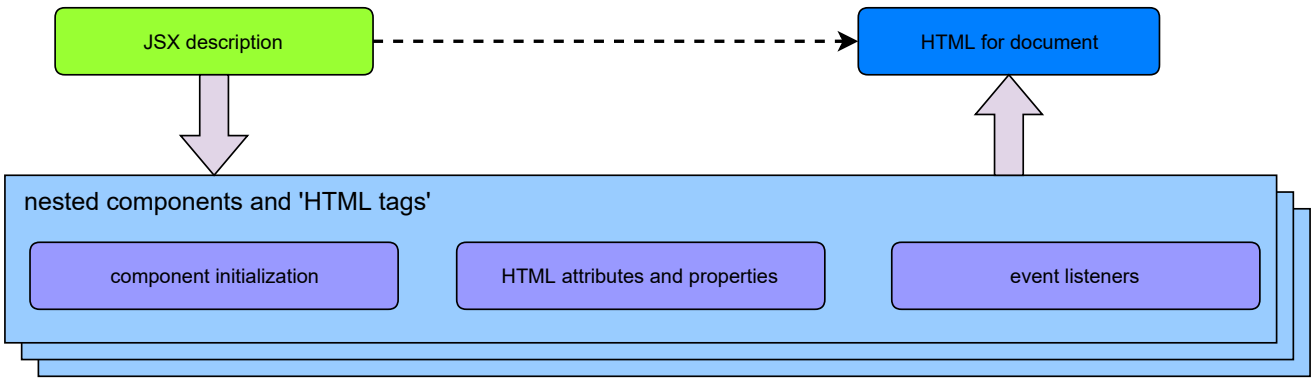
你可以看到一个叫做 `ReactMount`（01）的模块。它包含组件挂载的逻辑。实际上，在 `ReactDOM` 里面没有逻辑，它只是一个与 `ReactMount` 一起使用的接口，所以当你调用 `ReactDOM.render` 的时候，实际上调用了 `ReactMount.render`。那"挂载"指的是什么呢?

> 挂载是初始化 React 组件的过程。该过程通过创建组件所代表的 DOM 元素，并将它们插入到提供的 `container` 中来实现。

至少源码中的注释是这样描述的。那这真实的含义是什么呢？好吧，让我们想象一下下方的转换：



React 需要**将你的组件描述转换为 HTML** 以将其放入到 DOM 中。那怎样才能做到呢？没错，它需要处理所有的**属性、事件监听、内嵌的组件**和逻辑。它需要将你的高阶描述（组件）转换成实际可以放入到网页中的低阶数据（HTML）。这就是真正的挂载过程。



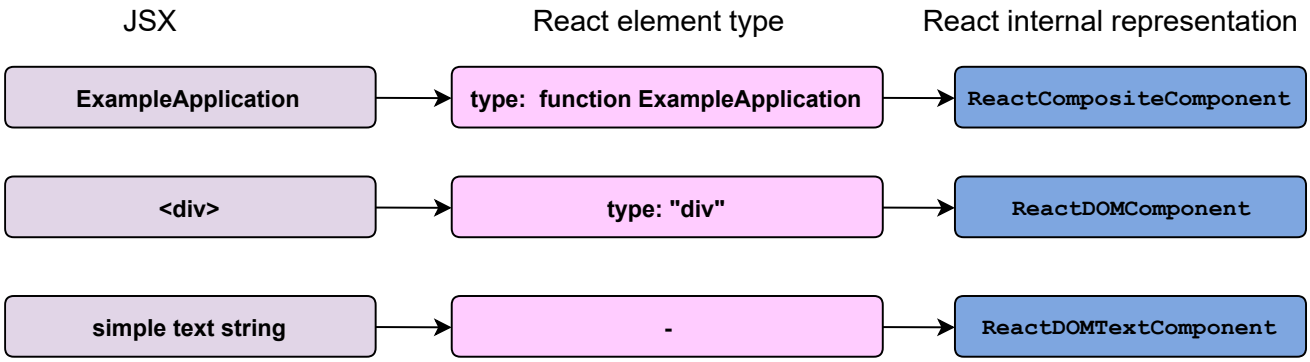
让我们继续深入下去。接下来是有趣的事实时间！是的，让我们在探索过程中添加一些有趣的东西，让它变得更"有趣"。

> 有趣的事实：确保滚动正在监听（02） ？？

> 有趣的是，在第一次渲染根组件时，React 初始化滚动监听并缓存滚动值，以便应用程序代码可以访问它们而不触发重排。实际上，由于浏览器渲染机制的不同，一些 DOM 值不是静态的，因此每次在代码中使用它们时都会进行计算。当然，这会影响性能。事实上，这只影响了不支持 `pageX` 和 `pageY` 的旧版浏览器。React 也试图优化这一点。可以看到，制作一个运行快速的工具需要使用很多技术，这个滚动就是一个很好的例子。

实例化 React 组件

看下流程图，在图中（03）处标明了一个创建的实例。在这里创建一个 `<ExampleApplication />` 的实例还为时过早。实际上该处实例化了 `TopLevelWrapper` （一个 React 内部的类）。让我们先来看看下面这个流程图。



你可以看到有三个部分，JSX 会被转换为 React 内部三种组件类型中的一种：`ReactCompositeComponent` （我们自定义的组件），`ReactDOMComponent` （HTML 标签）和 `ReactDOMTextComponent` （文本节点）。我们将略过描述 `ReactDOMTextComponent` 并将重点放在前两个。

内部组件？这很有趣。你已经听说过 **虚拟 DOM** 了吧？虚拟 DOM 是一种 DOM 的表现形式。 React 用虚拟 DOM 进行组件差异计算等过程。该过程中无需直接操作 DOM 。这使得 React 在更新视图时候更快。但在 React 的源码中没有名为“Virtual DOM”的文件或者类。这是因为 虚拟DOM 只是一个概念，一种如何操作真实 DOM 的方法。所以，有些人说 虚拟DOM 元素等同于 React 组件，但在我看来，这并不完全正确。我认为虚拟 DOM 指的是这三个类：`ReactCompositeComponent` 、 `ReactDOMComponent` 和 `ReactDOMTextComponent` 。后面你会知道到为什么。

好了，让我们在这里完成实例化过程。我们将创建一个 `ReactCompositeComponent` 实例，但实际上这并不是因为我们把 `<ExampleApplication />` 放在了 `ReactDOM.render` 里。React 总是从 `TopLevelWrapper` 开始渲染一棵组件的树。它几乎是一个空的包装器，其 `render` 方法（组件的 render）随后将返回 `<ExampleApplication />` 。

```
//src\renderers\dom\client\ReactMount.js#277
TopLevelWrapper.prototype.render = function () {
  return this.props.child;
};

```

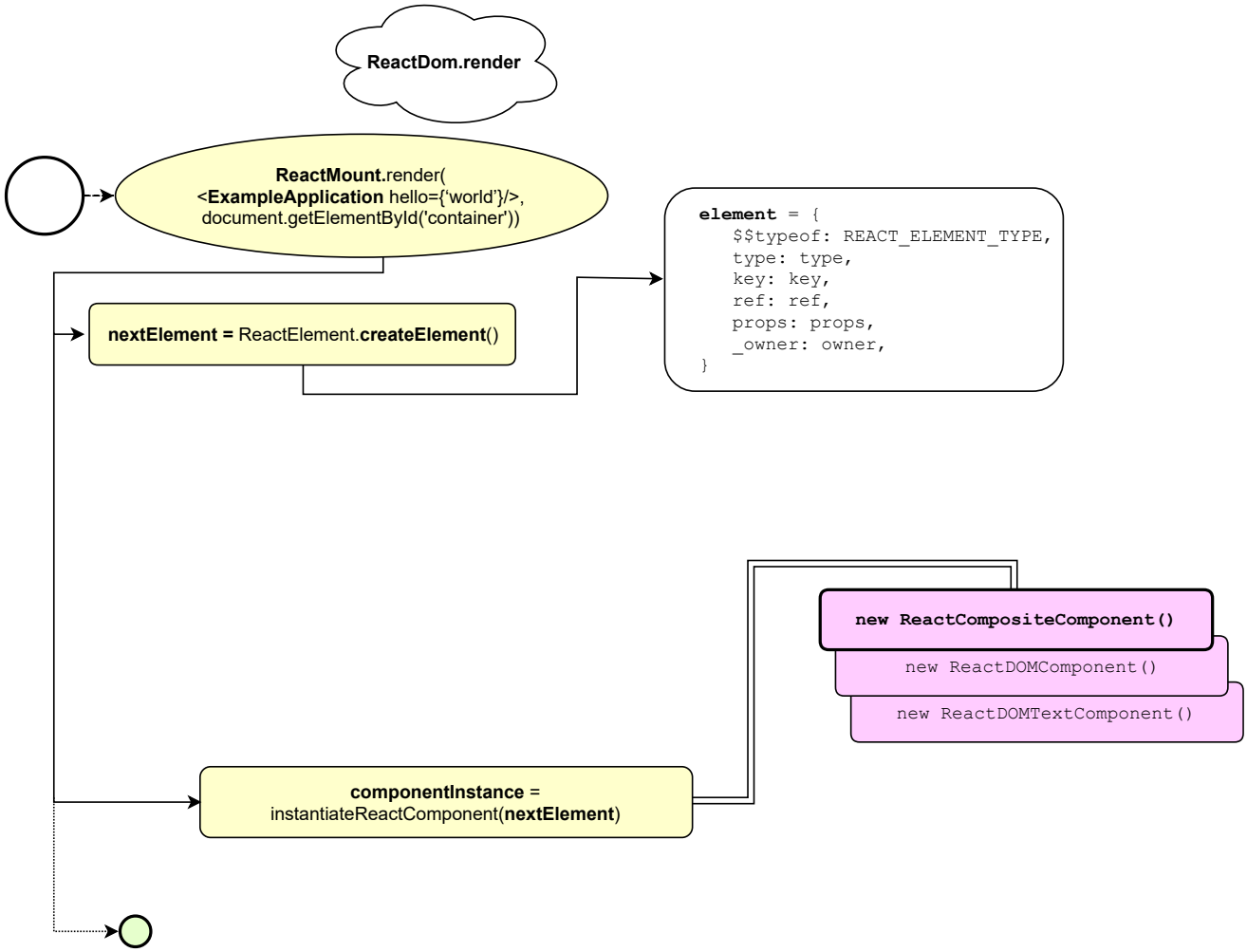
所以，目前为止只有 `TopLevelWrapper` 被创建了。但是……先看一下一个有趣的事实。

> 有趣的事实：验证 DOM 内嵌套　　react 中渲染组件会有一个 对组件嵌套的规则验证：

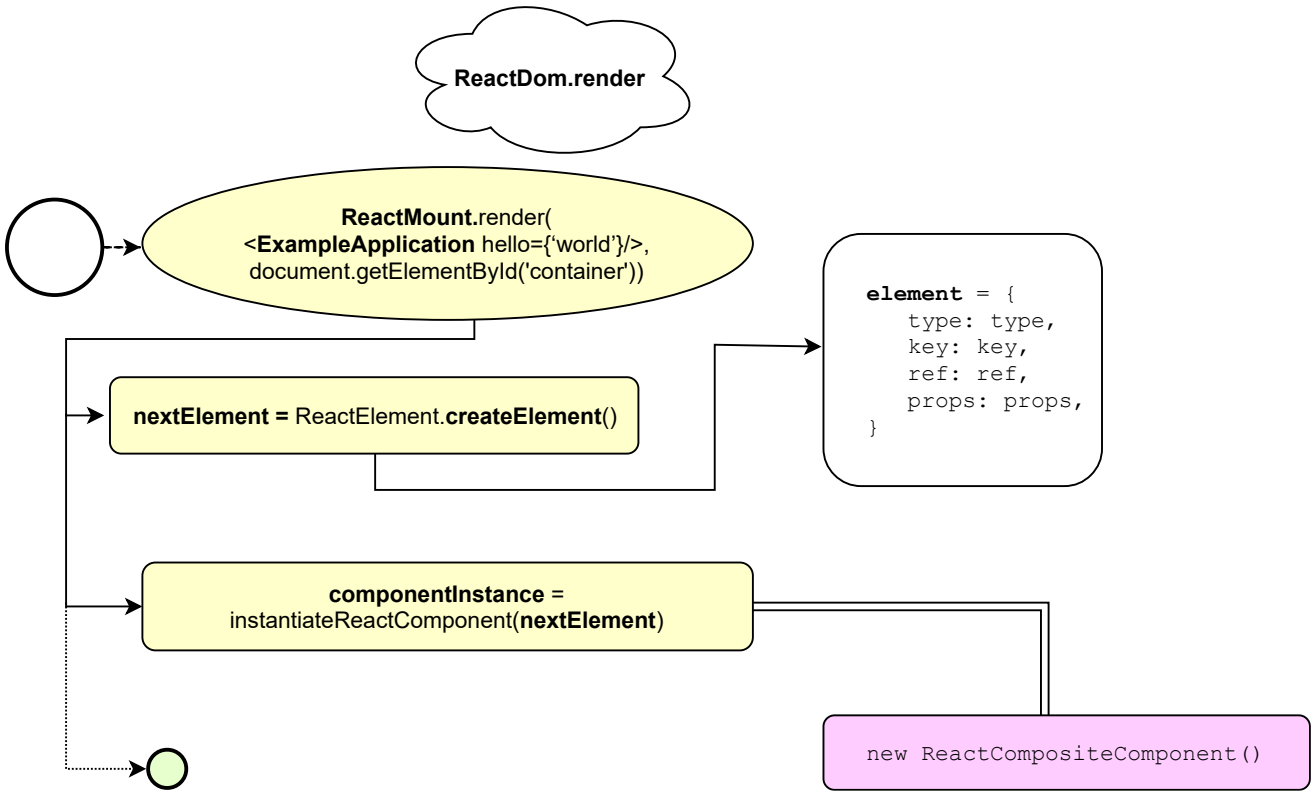
> 几乎每次内嵌的组件渲染时，都被一个专门用于进行 HTML 验证的 `validateDOMNesting` 模块验证。DOM 内嵌验证指的是 `子标签 -> 父标签` 的标签层级的验证。例如，如果父标签是 `<select>` ，则子标签应该是以下其中一个标签： `option` 、 `optgroup` 或者 `#text` 。这些规则实际上是在 https://html.spec.whatwg.org/multipage/syntax.html#parsing-main-electlect 中定义的。你可能已经看到过这个模块是如何工作的，它像这样报错： * cannot appear as a descendant of <p> .*

小结

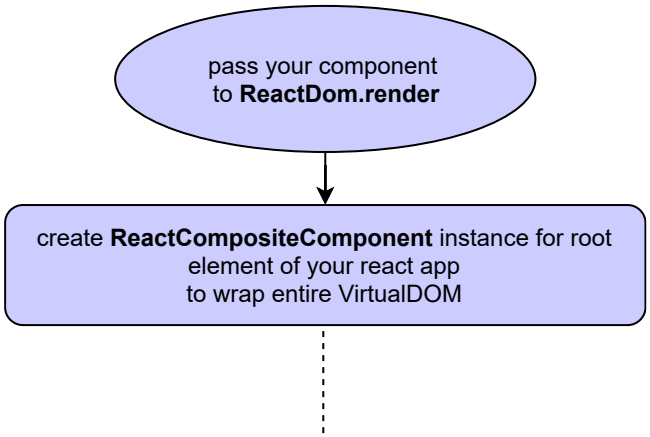
让我们回顾一下上面的内容。再看一下流程图，然后删除多余的不太重要的部分，变成下面这样：


ReactDom.render
ReactMount.render(
<ExampleApplication hello={'world'}/>,
document.getElementById('container'))
element = {
$$typeof: REACT_ELEMENT_TYPE,
type: type,
key: key,
ref: ref,
props: props,
_owner: owner,
}
nextElement = ReactElement.createElement()
new ReactCompositeComponent()
new ReactDOMComponent()
new ReactDOMTextComponent()
componentInstance =
instantiateReactComponent(nextElement)

再调整一下间距和对齐：



实际上，这就是本部分的所有内容。因此，我们可以从 **第 0 部分** 中得到重点，并将它用于最终的 `mounting` 流程中：



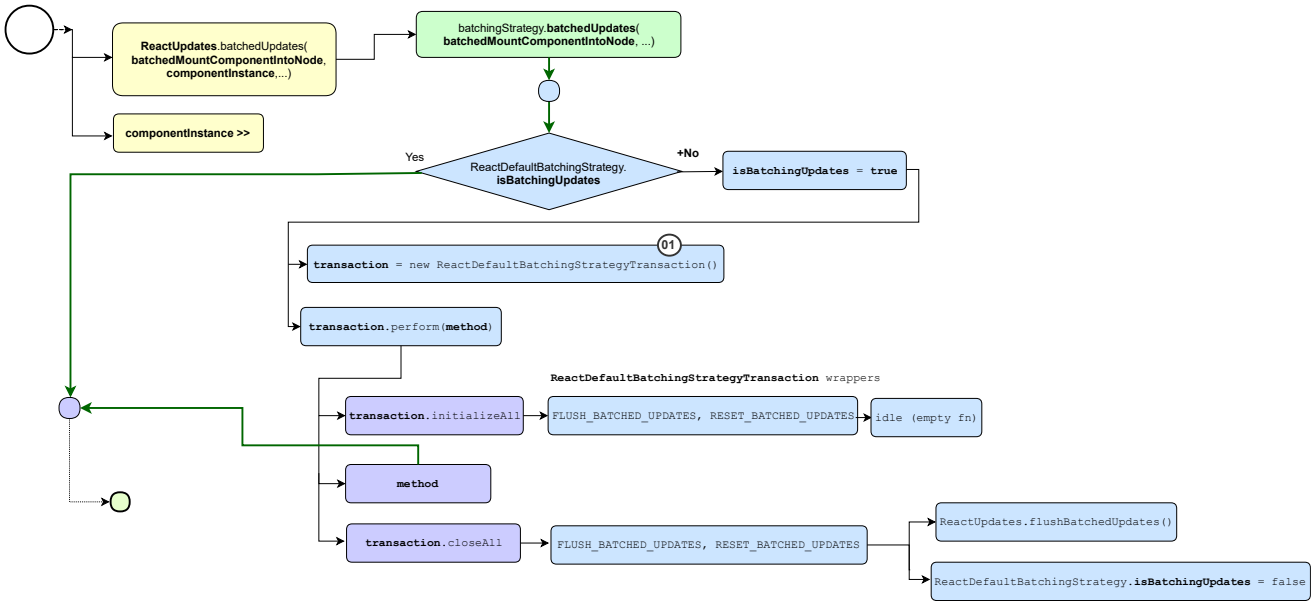
完成了！

[下一页：第 1 部分 >>]

[<< 上一页：介绍]

[首页]

第 1 部分



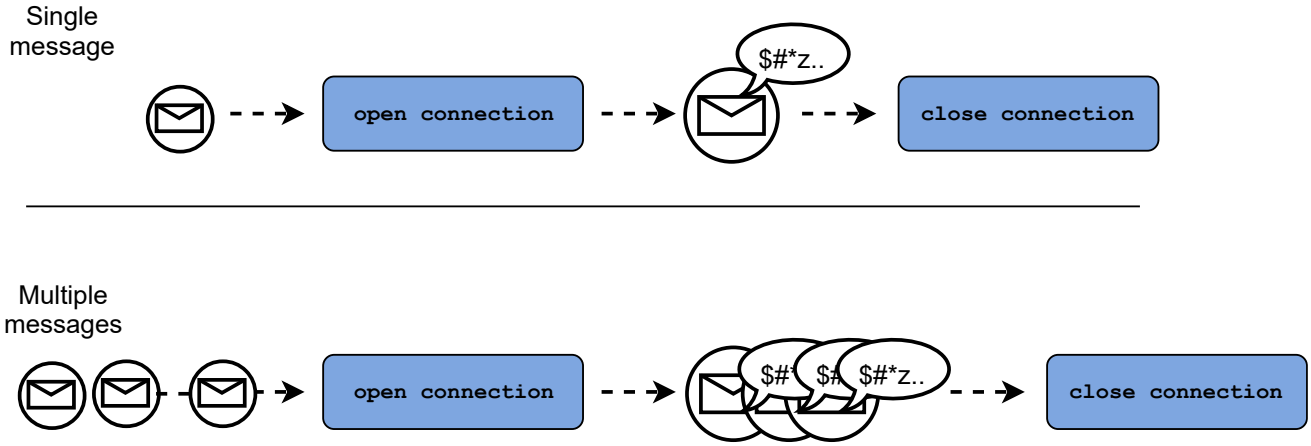
1.0 第 1 部分(点击查看大图)

事务

某一组件实例应该以某种方式**连接入**React的生态系统，并对该系统**产生一些影响**。有一个专门的模块名为 `ReactUpdates` 专职于此。 正如大家所知, **React 以块形式执行更新**，这意味着它会收集一些操作然后**统一**执行。这样做更好，因为这样允许为整个块只应用一次某些**前置条件**和**后置条件**，而不是为块中的每个操作都应用。

什么真正执行了这些前/后处理？对， **事务**！对某些人来说，**事务**可能是一个新术语，至少对UI方面来说是个新的含义。接下来我们从一个简单的例子开始再来谈一下它。

想象一下 `通信信道` 。你需要开启连接，发送消息，然后关闭连接。 如果你按这个方式逐个发送消息，就要每次发送消息的时候建立、关闭连接。不过，你也可以只开启一次连接，发送所有挂起的消息然后关闭连接。



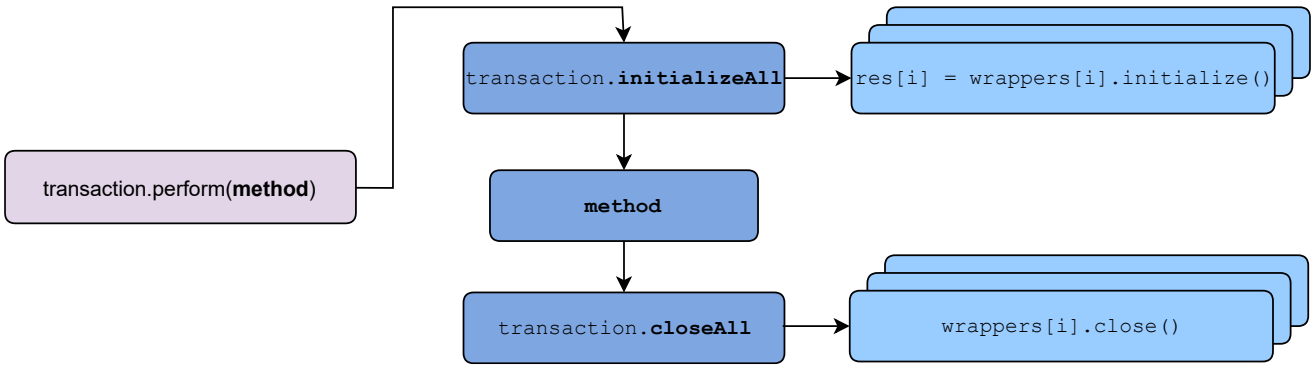
1.1 非常真实的事务示例(查看大图)

好的，让我们再想想更多抽象的东西。想象一下，在执行操作期间，“发送消息”是您要执行的任何操作，“打开/关闭连接”是预处理/后处理。 然后，再想想一下，你可以分别定义任何 open/close 对，并使用任何方法来使用它们（我们可以将它们命名为 `wrapper` ,因为事实上每一对都包装动作方法）。听起来很酷，不是吗?

我们回到 React。 事务是 React 中广泛使用的模式。除了包装行为外，事务允许应用程序重置事务流，如果某事务已在进行中则阻止同时执行，等等。有很多不同的事务类，它们每个都描述具体的行为，它们都继承自 `Transaction` 模块。事务类之间的主要区别是具体的事务包装器的列表的不同。包装器只是一个包含初始化和关闭方法的对象。

所以，**我的想法是**:

- 调用每个 wrapper.initialize 方法并缓存返回结果（可以进一步使用）
- 调用事务方法本身
- 调用每个 wrapper.close 方法



1.2 事务实现(点击查看大图)

我们来看看 React 中的一些**其他事务用例**:

- 在差分对比更新渲染步骤的前后，保留输入选取的范围，即使在发生意外错误的情况下也能保存。
- 在重排DOM时，停用事件，防止模糊/焦点选中，同时保证事件系统在 DOM 重排后重新启动。
- 在 worker 线程完成了差分对比更新算法后，将一组选定的 DOM 变化直接应该用到 UI 主线程上。
- 在渲染新内容后触发任何收集到的 `componentDidUpdate` 回调。

让我们回到具体案例。

正如我们看到的， React 使用 `ReactDefaultBatchingStrategyTransaction` (1)。我们前文提到过，事务最重要的是它的包装器。所以，我们可以看看包装器，并弄清楚具体被定义的事务。好，这里有两个包装器：`FLUSH_BATCHED_UPDATES`， `RESET_BATCHED_UPDATES`。我们来看它们的代码：

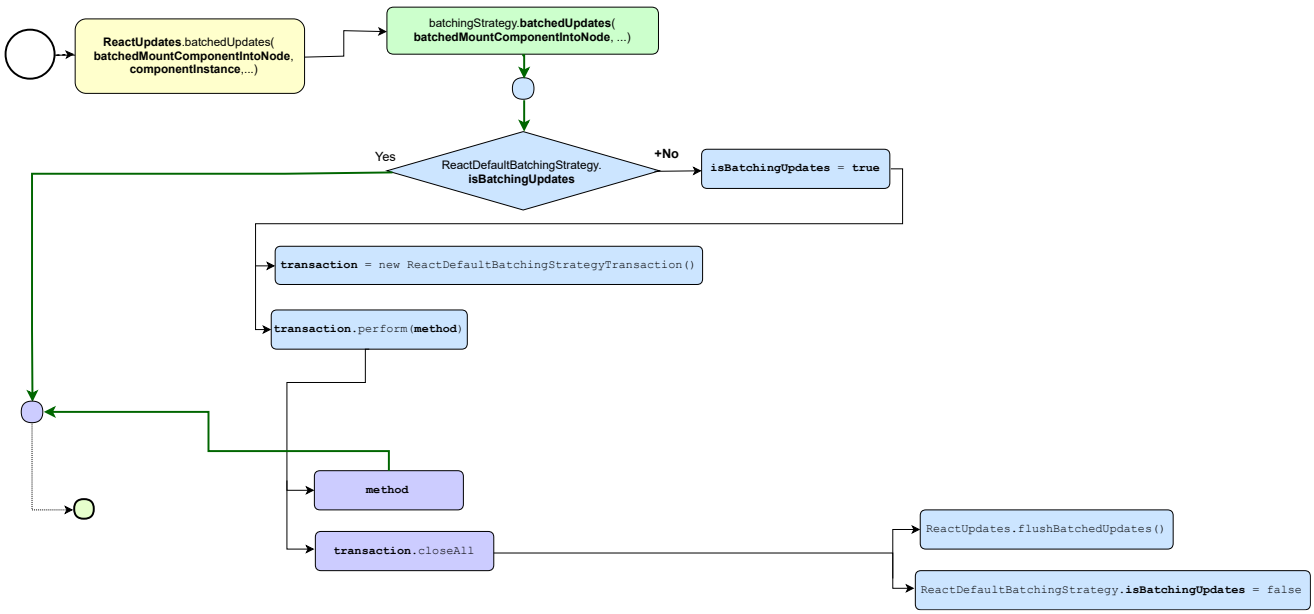
```
//\src\renderers\shared\stack\reconciler\ReactDefaultBatchingStrategy.js#19
var RESET_BATCHED_UPDATES = {
      initialize: emptyFunction,
      close: function() {
        ReactDefaultBatchingStrategy.isBatchingUpdates = false;
      },
};

var FLUSH_BATCHED_UPDATES = {
     initialize: emptyFunction,
     close: ReactUpdates.flushBatchedUpdates.bind(ReactUpdates),
}

var TRANSACTION_WRAPPERS = [FLUSH_BATCHED_UPDATES, RESET_BATCHED_UPDATES];
```

所以，你可以看看事务的写法。此代码中事务没有前置条件。 `initialize` 方法是空的,但其中一个 `close` 方法很有趣。它调用了`ReactUpdates.flushBatchedUpdates`。 这意味着什么? 它实际上对对脏组件的验证进一步重新渲染。所以，你理解了，对吗？我们调用 mount 方法并将其包装在这个事务中，因为在 mount 执行后，React 检查已加载的组件对环境有什么影响并执行相应的更新。

我们来看看包装在该事务中的方法。 事实上，它引发了另外一个事务...

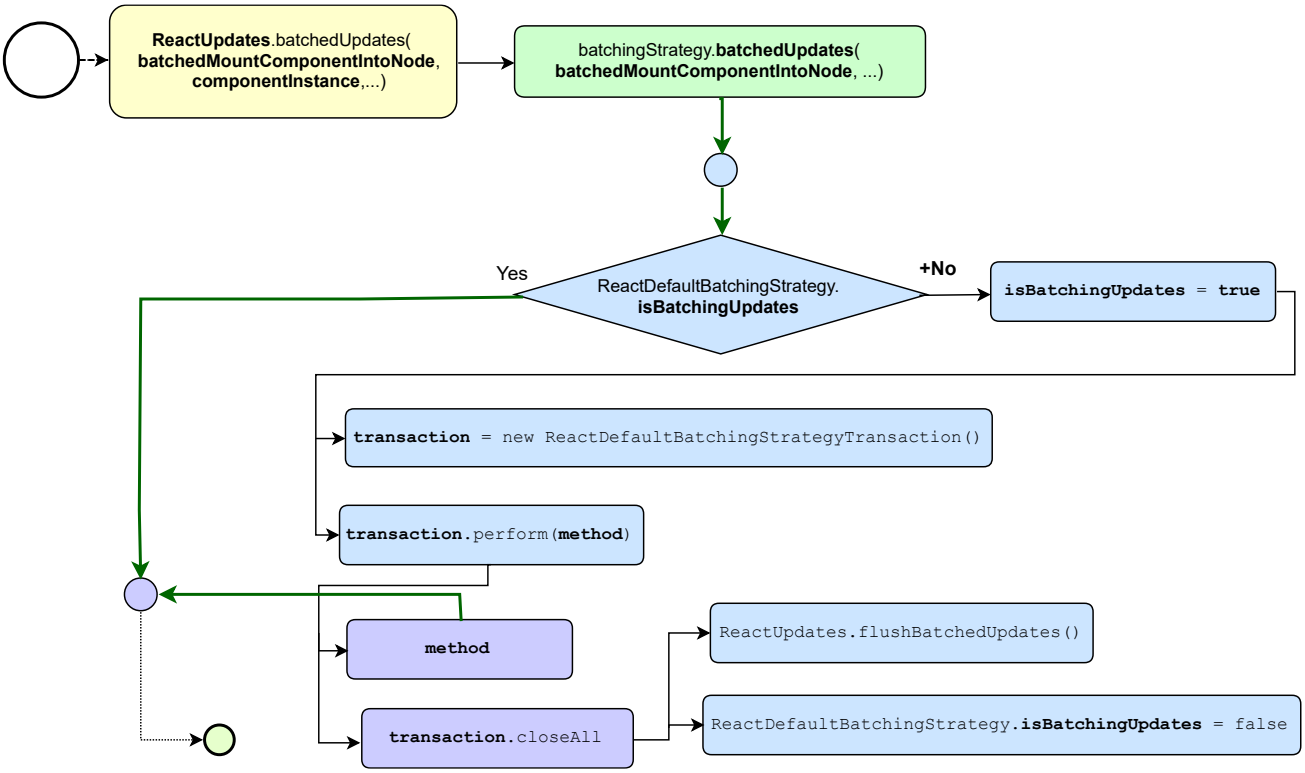
第 1 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



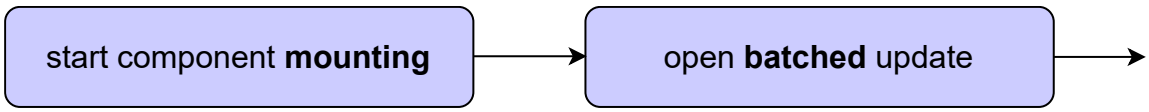
1.3 第 1 部分简化版 (点击查看大图)

然后我们适当再调整一下：



1.4 第 1 部分简化和重构(点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 1 部分**的本质，并将其画在最终的 `mount` （挂载） 方案里：



1.5 第 1 部分本质(点击查看大图)

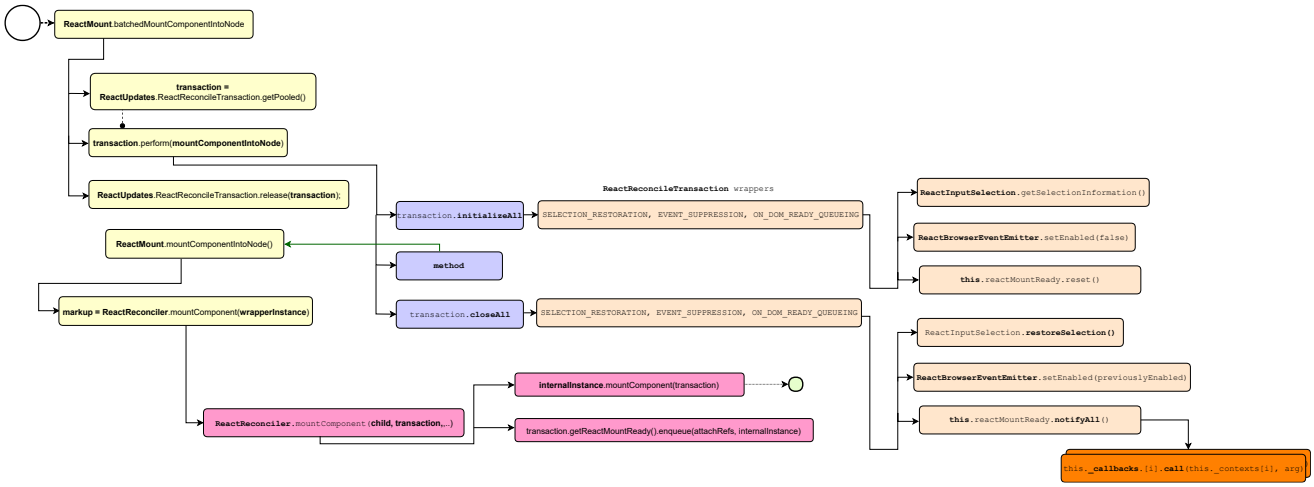
完成！

下一节: 第 2 部分 >>

<< 上一节 第 0 部分

主页

第二部分



2.0 第二部分

另一个事务

这次我们将讨论 `ReactReconcileTransaction` 事务。正如你所知道的，对我们来说主要感兴趣的是事务包装器。其中包括三个包装器：

```
//\src\renderers\dom\client\ReactReconcileTransaction.js#89
var TRANSACTION_WRAPPERS = [
  SELECTION_RESTORATION,
  EVENT_SUPPRESSION,
  ON_DOM_READY_QUEUEING,
];
```

我们可以看到，这些包装器主要用来 *保留实际状态*，React 将确保在事务的方法调用之前锁住某些可变值，调用完后再释放它们。举个例子，范围选择（输入当前选择的文本）不会被事务的方法执行干扰（在 `initialize` 时选中并在 `close` 时恢复）。此外，它阻止因为高级 DOM 操作（例如，临时从 DOM 中移除文本）而无意间触发的事件（例如模糊/选中焦点），React在 `initialize` 时 *暂时禁用* `ReactBrowserEventEmitter` 并在事务执行到 `close` 时重新启用。

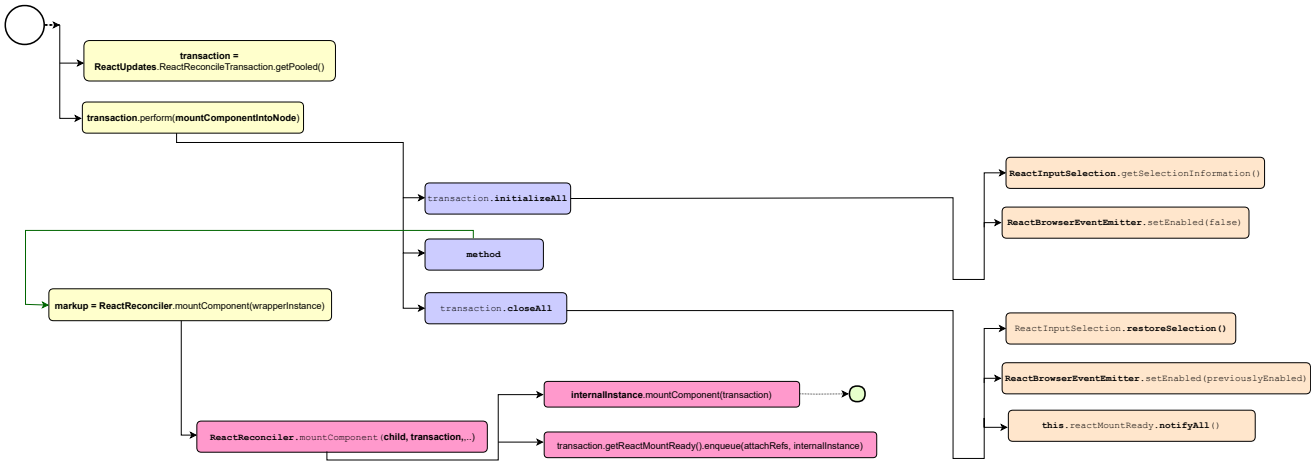
到这里，我们已经非常接近组件的挂载了，挂载将会把我们准备好的（HTML）标记插入到 DOM 中。实际上，`ReactReconciler.mountComponent` 只是一个包装，更准确的说，它是一个中介者。它将代理组件模块的挂载方法。这是一个重要的部分，画个重点。

> 在实现某些和平台相关的逻辑时，`ReactReconciler` 模块总是会被调用，例如这个确切的例子。挂载过程在每个平台上都是不同的，所以 “主模块” 会询问 `ReactReconciler`，`ReactReconciler` 知道下一步应该怎么做。

好的，让我们将目光移到组件方法 `mountComponent` 上。这可能是你已经听说过的方法了。它初始化组件，渲染标记以及注册事件监听函数。你看，千辛万苦我们终于看到了调用组件加载。调用加载之后，我们应该可以得到可以插入到文档中的 HTML 元素了。

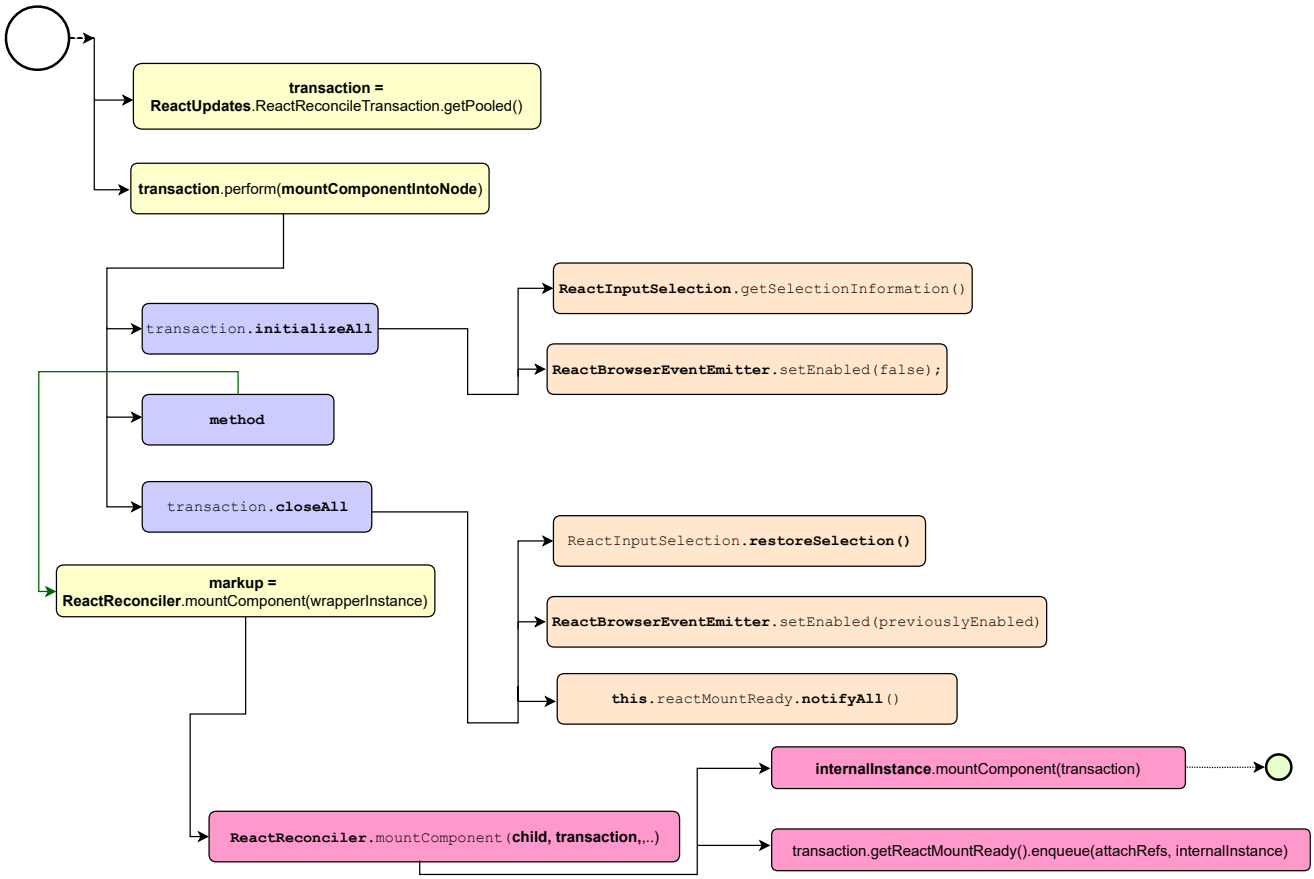
我们完成了 *第二部分*

让我们回顾一下这一部分，我们再一次流程图，然后删除一些不重要的信息，它将变成这样：



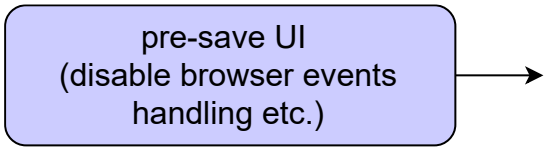
2.1 第二部分 简化

让我们优化一下排版：



2.2 第二部分 简化与重构

很好，其实这就是这一部分所发生的一切。我们可以从 *第一部分* 中取下必要的信息，然后完善 `mounting` （挂载） 的流程图：



2.3 第二部分 必要信息

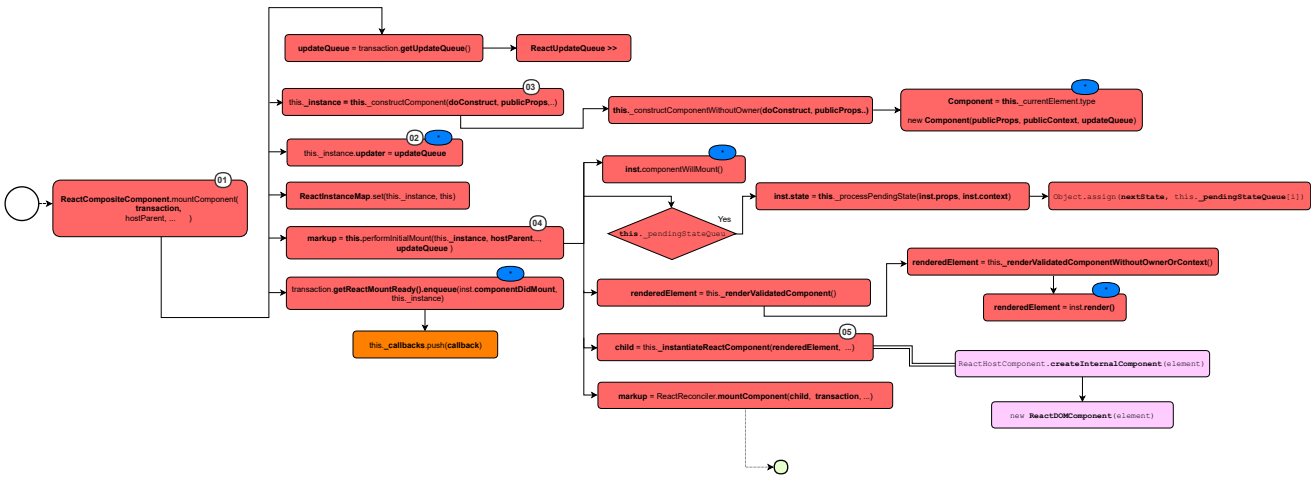
完成啦！

[下一页：第三部分 >>]

[<< 上一页：第一部分]

[首页]

第 3 部分



3.0 第 3 部分 (点击查看大图)

挂载

`componentMount` 方法是我们整个系列中极其重要的一个部分。如图，我们关注 `ReactCompositeComponent.mountComponent` (1) 方法。

如果你还记得，我曾提到过 **组件树的入口组件** 是 `TopLevelWrapper` 组件 (React 底层内部类)。我们准备挂载它。由于它实际上是一个空的包装器，调试起来非常枯燥并且对实际的流程而言没有任何影响，所以我们跳过这个组件从他的孩子组件开始分析。

把组件挂载到组件树上的过程就是先挂载父亲组件，然后他的孩子组件，然后他的孩子的孩子组件，依次类推。可以肯定，当 `TopLevelWrapper` 挂载后，他的孩子组件 (用来管理 `ExampleApplication` 的组件 `ReactCompositeComponent`) 也会在同一阶段注入。

现在我们回到步骤 (1) 观察这个方法的内部实现，有一些重要行为会发生，接下来让我们深入研究这些重要行为。

给实例赋值 updater

从 `transaction.getUpdateQueue()` 方法返回的 `updater` 见图中(2)， 实际上就是 `ReactUpdateQueue` 模块。 为什么要在这里赋值一个 `updater` 呢？因为我们正在研究的类 `ReactCompositeComponent` 是一个全平台的共用的类，但是 `updater` 却依赖于平台环境有不同的实现，所以我们在这里根据不同的平台动态的将它赋值给实例。

然而，我们现在并不马上需要这个 `updater` ，但是你要记住它是非常重要的，因为它很快就会应用于非常知名的组件内更新方法 **`setState`** 。

事实上在这个过程中，不仅仅 `updater` 被赋值给实例，组件实例（你的自定义组件）也获得了继承的 `props` , `context` , 和 `refs` 。

观察以下的代码:

```
// \src\renderers\shared\stack\reconciler\ReactCompositeComponent.js#255
// 这些应该在构造方法里赋值，但是为了
// 使类的抽象更简单，我们在它之后赋值。
inst.props = publicProps;
inst.context = publicContext;
inst.refs = emptyObject;
inst.updater = updateQueue;
```

因此，你才可以通过一个实例从你的代码中获得 `props`，比如 `this.props`。

创建 ExampleApplication 实例

通过调用步骤 (3) 的方法 `_constructComponent` 然后经过几个构造方法的作用后，最终创建了 `new ExampleApplication()`。这就是我们代码中构造方法第一次被执行的时机，当然也是我们的代码第一次实际接触到 React 的生态系统，很棒。

执行首次挂载

接着我们研究步骤 (4)，第一个即将发生的行为是 `componentWillMount` (当然仅当它被定义时) 的调用。这是我们遇到的第一个生命周期钩子函数。当然，在下面一点你会看到 `componentDidMount` 函数, 只不过这时由于它不能马上执行，而是被注入了一个事务队列中，在很后面执行。他会在挂载系列操作执行完毕后执行。当然你也可能在 `componentWillMount` 内部调用 `setState`，在这种情况下 `state` 会被重新计算但此时不会调用 `render`。(这是合理的，因为这时候组件还没有被挂载)

官方文档的解释也证明这一点:

> `componentWillMount()` 在挂载执行之前执行，他会在 `render()` 之前被调用，因此在这个过程中设置组件状态不会触发重绘。

观察以下的代码，进一步验证:

```
// \src\renderers\shared\stack\reconciler\ReactCompositeComponent.js#476
if (inst.componentWillMount) {
    //..
    inst.componentWillMount();

    // 当挂载时，在 `componentWillMount` 中调用的 `setState` 会执行并改变状态
    // `this._pendingStateQueue` 不会触发重渲染
    if (this._pendingStateQueue) {
        inst.state = this._processPendingState(inst.props, inst.context);
    }
}
```

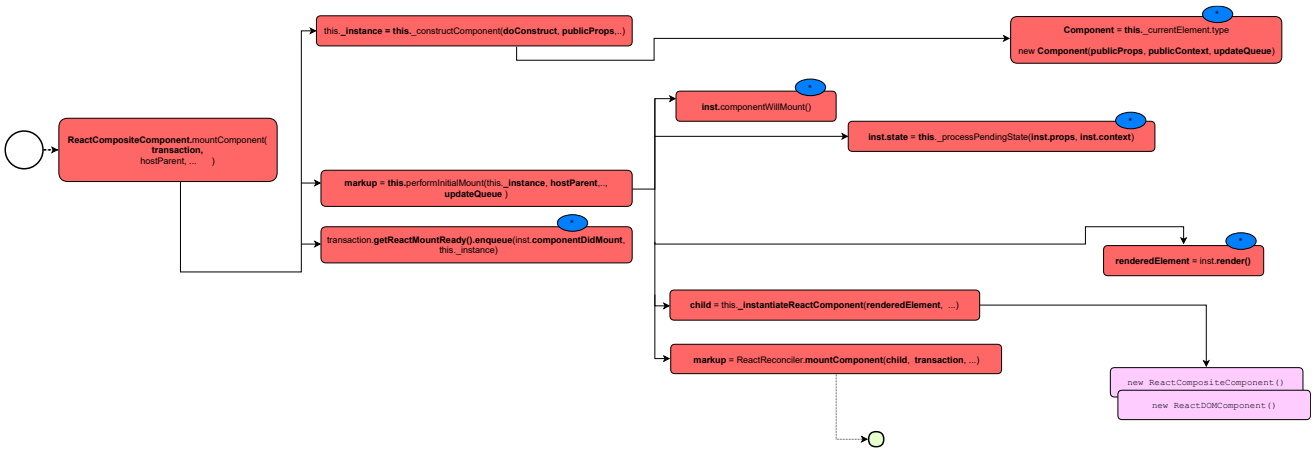
确实如此，但是当 state 被重新计算完成后，会调用我们在组件中申明的 render 方法。再一次接触 “我们的” 代码。

接下来下一步就会创建一个 React 的组件的实例。然后呢？我们已经看见过步骤 (5) `this._instantiateReactComponent` 的调用了，对吗？是的。在那个时候它为我们的 `ExampleApplication` 组件实例化了 `ReactCompositeComponent`，现在我们准备基于它的 `render` 方法获得的元素作为它的孩子创建 VDOM (虚拟 DOM) 实例。在我们的例子中，`render` 方法返回了一个 `div`，所以准确的 VDOM 元素是一个 `ReactDOMElement`。当该实例被创建后，我们会再次调用 `ReactReconciler.mountComponent`，但是这次我们传入刚刚新创建的 `ReactDOMComponent` 实例作为 `internalInstance`。

然后继续调用此类中的 `mountComponent` 方法，这样递归往下...

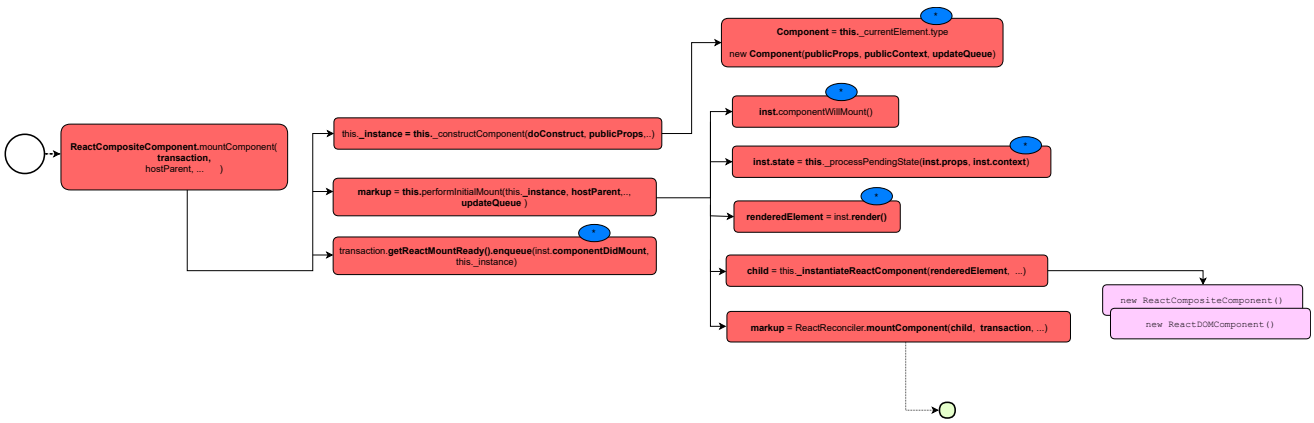
好，第 3 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



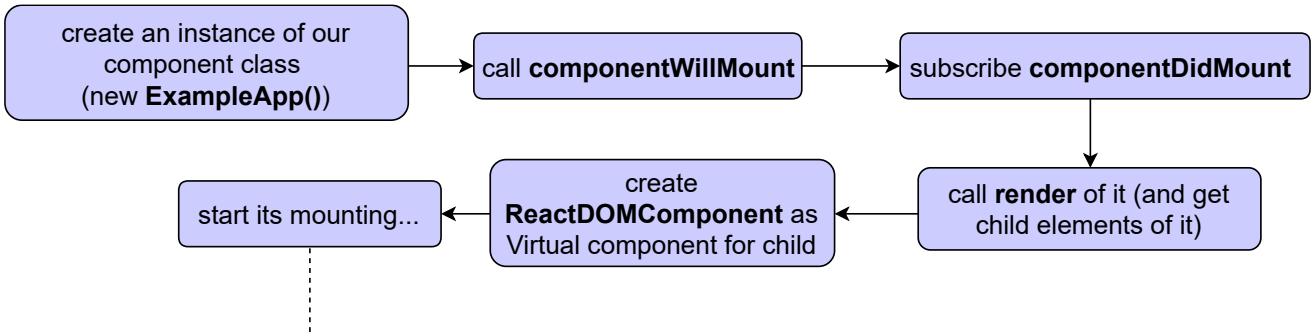
3.1 第 3 部分简化版 (点击查看大图)

让我们适度在调整一下:



3.2 第 3 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 3 部分**的本质，并将其用于最终的 `mount` 方案：



3.3 第 3 部分本质 (点击查看大图)

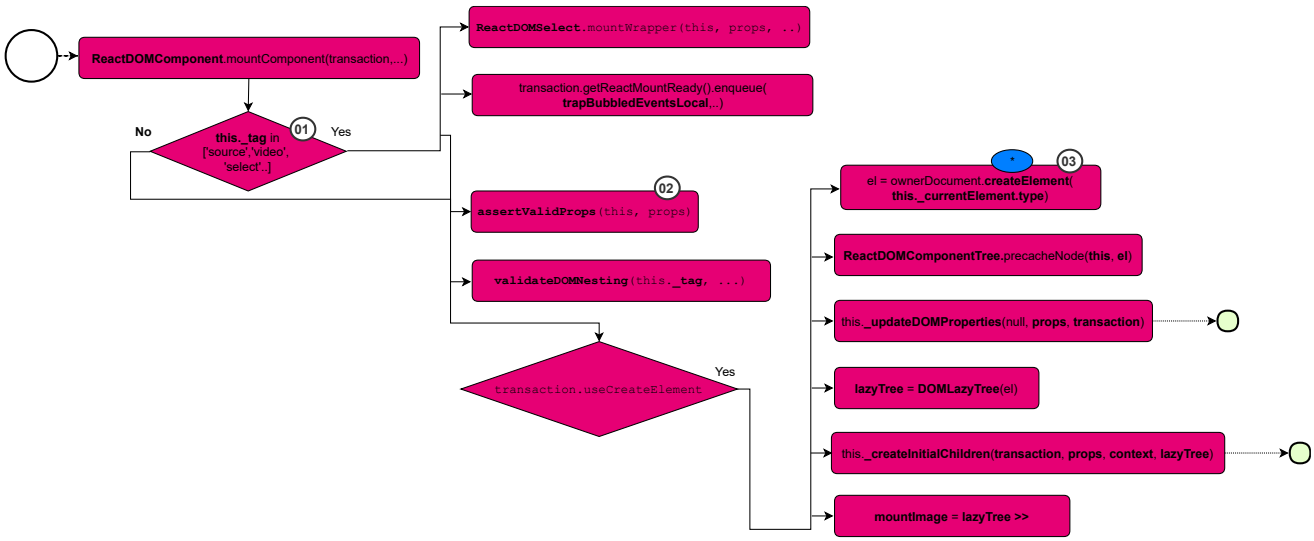
完成!

下一节: 第 4 部分 >>

<< 上一节: 第 2 部分

主页

第 4 部分



4.0 第 4 部分 (点击查看大图)

子元素挂载

已经入迷了对吗? 让我们接续研究 `mount` 方法。

如果步骤 (1) 的 `_tag` 包含一个复杂的标签，比如 `video`、`form`、`textarea` 等等，这些就需要更进一步的封装，对每个媒体事件需要绑上更多事件监听器，比如给 `audio` 标签增加 `volumechange` 事件监听，或者像 `select`、`textarea` 等标签只需要封装一些浏览器原生行为。

我们有很多封装器干这事，比如 `ReactDOMSelect` 和 `ReactDOMTextarea` 位于源码 (src\renderers\dom\client\wrappers\folder) 中。本文例子中只有简单的 `div` 标签。

Props 验证

接下来要讲解的验证方法是为了确保内部 `props` 被设置正确，不然它就会抛出异常。举个例子，如果设置了 `props.dangerouslySetInnerHTML` (经常在我们需要基于一个字符串插入 HTML 时使用)，但是它的对象健值 `__html` 忘记设置，那么将会抛出下面的异常：

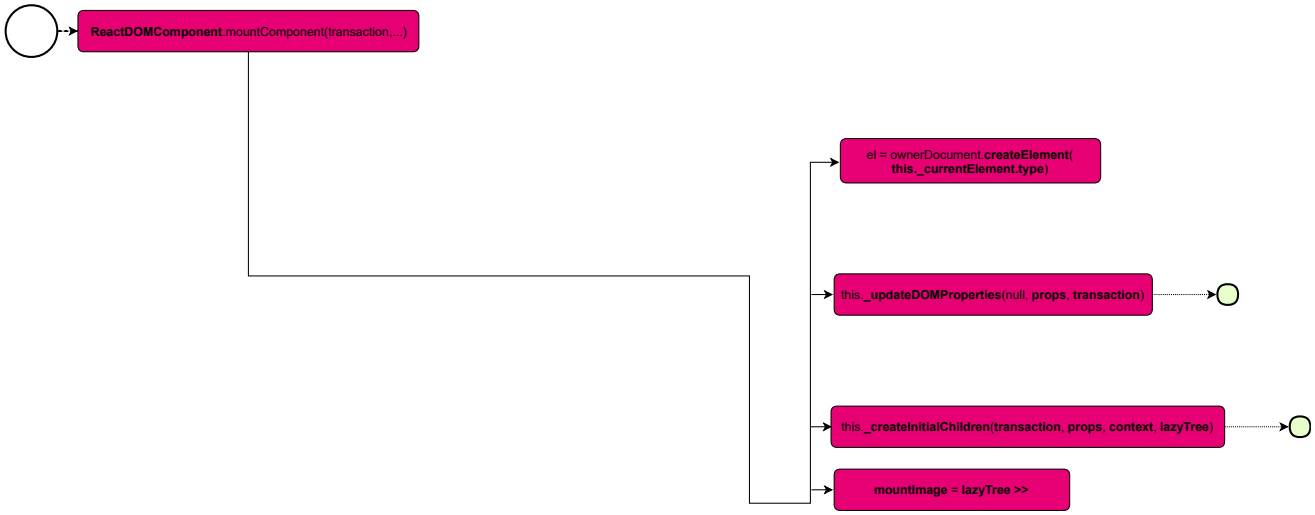
> `props.dangerouslySetInnerHTML` must be in the form `{__html: ...}`. Please visit https://fb.me/react-invariant-dangerously-set-inner-html for more information.
>
> (`props.dangerouslySetInnerHTML` 必须符合 `{__html: ...}` 的形式)

创建 HTML 元素

接着， `document.createElement` 方法会创建真实的 HTML 元素，实例出真实的 HTML `div`，在这一步之前我们只能用虚拟的表现形式表达，而现在你第一次能实际看到它了。

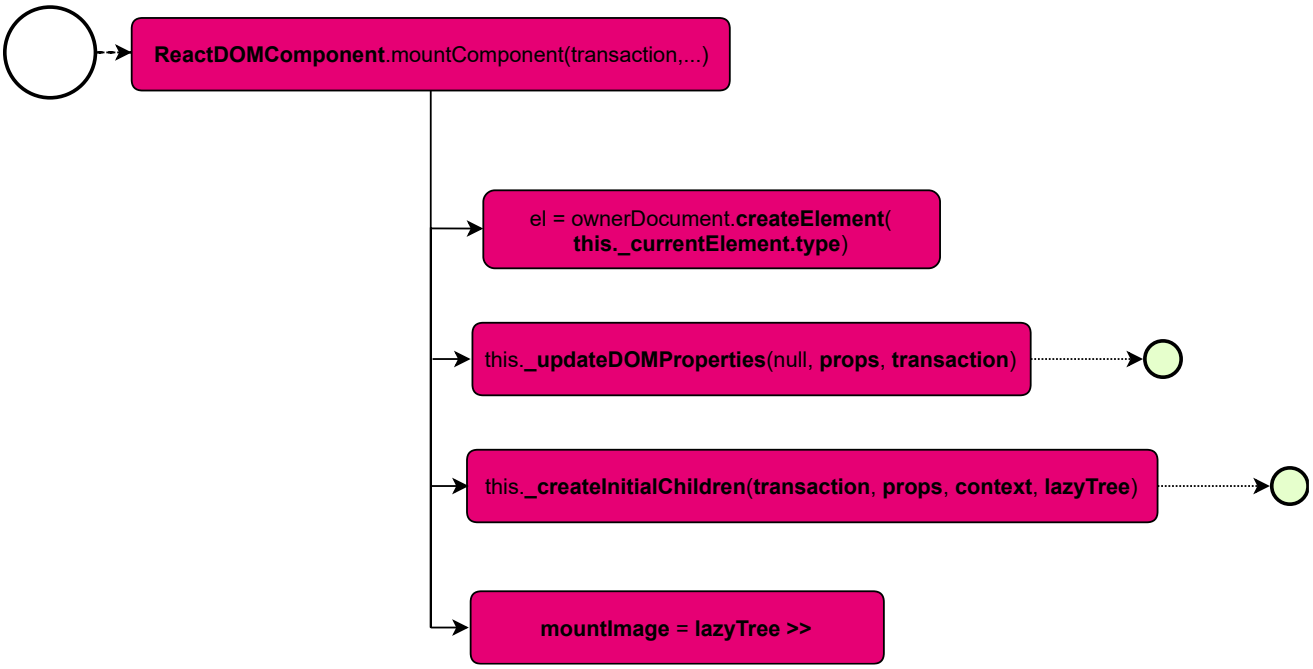
好，第 4 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



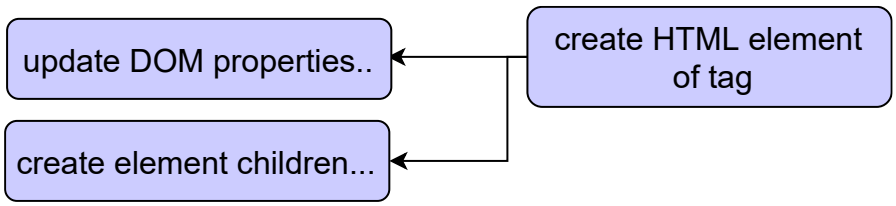
4.1 第 4 部分简化版 (点击查看大图)

让我们适度在调整一下：



4.2 第 4 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 4 部分**的本质，并将其用于最终的 `mount` 方案：



4.3 第 4 部分本质 (点击查看大图)

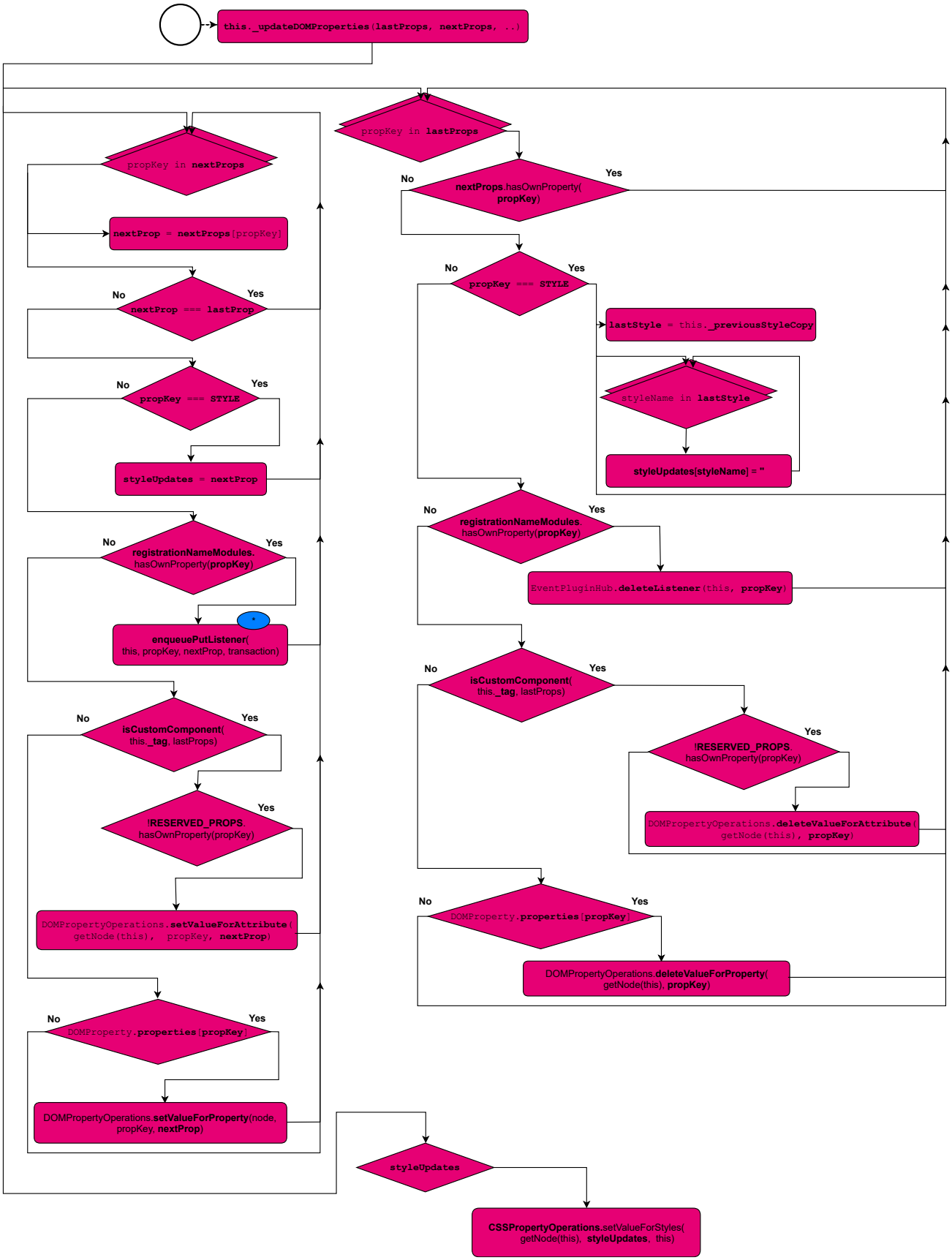
完成！

下一节: 第 5 部分>>

<< 上一节: 第 3 部分

主页

第 5 部分



5.0 第 5 部分(点击查看大图)

更新 DOM 属性

这图片看上去有点复杂？这里主要讲的是如何高效的把diff作用到新老 `props` 上。我们来看一下源码对这块的代码注释：

> 差分对比更新算法通过探测属性的差异并更新需要更新的 DOM。该方法可能是性能优化上唯一且极其重要的一环。

这个方法实际上有两个循环。第一个循环遍历前一个 `props`，后一个循环遍历下一个 `props`。在我们的挂载场景下，`lastProps` (前一个) 是空的。(很明显这是第一次给我们的 props 赋值)，但是我们还是来看看这里发生了什么。

`lastprops` 循环

第一步，我们检查 `nextProps` 对象是不是包含相同的 prop 值，如果相等的话，我们就跳过那个值，因为它之后会在 `nextProps` 循环中处理。然后我们重置样式的值，删除事件监听器 (如果监听器之前设置过的话)，然后去除 DOM 属性名以及 DOM 属性值。对于属性们，只要我们确定它们不是 `RESERVED_PROPS` 中的一员，而是实际的 `prop`，例如 `children` 或者 `dangerouslySetInnerHTML`。

`nextprops` 循环

该循环中，第一步检查 `prop` 是不是变化了，也就是检查下一个值是不是和老的值不同。如果相同，我们不做任何处理。对于 `styles` （你也许已经注意到我们会区别对待它）我们更新从 `lastProp` 到现在变化的部分值。然后我们添加事件监听器(比如 `onClick` 等等)。让我们更深入的分析它。

其中很重要的一点是，纵观 React app，所有的工作都会传入一个名叫 `synthetic` 的事件。没有一个例外。它其实是一些封装器来优化效率的。下一个重要部分是我们处理事件监听器的中介控制模块 `EventPluginHub` (位于源码中 `src\renderers\shared\stack\event\EventPluginHub.js`)。它包含一个 `listenerBank` 的映射来缓存并管控所有的监听器。我们准备好了添加我们自己的事件监听器，但是不是现在。这里的关键在于我们应该在组件和 DOM 元素已经准备好处理事件的时候才增加监听器。看上去在这里我们执行迟了。也你许会问，我们如何知道 DOM 已经准备好了？很好，这就引出了下一个问题！你是否还记得我们曾把 `transaction` 传递给每个方法和调用？这就对了，我们那样做就是因为在这种场景它可以很好的帮助我们。让我们从代码中寻找佐证：

```
//src\renderers\dom\shared\ReactDOMComponent.js#222
transaction.getReactMountReady().enqueue(putListener, {
    inst: inst,
    registrationName: registrationName,
    listener: listener,
});
```

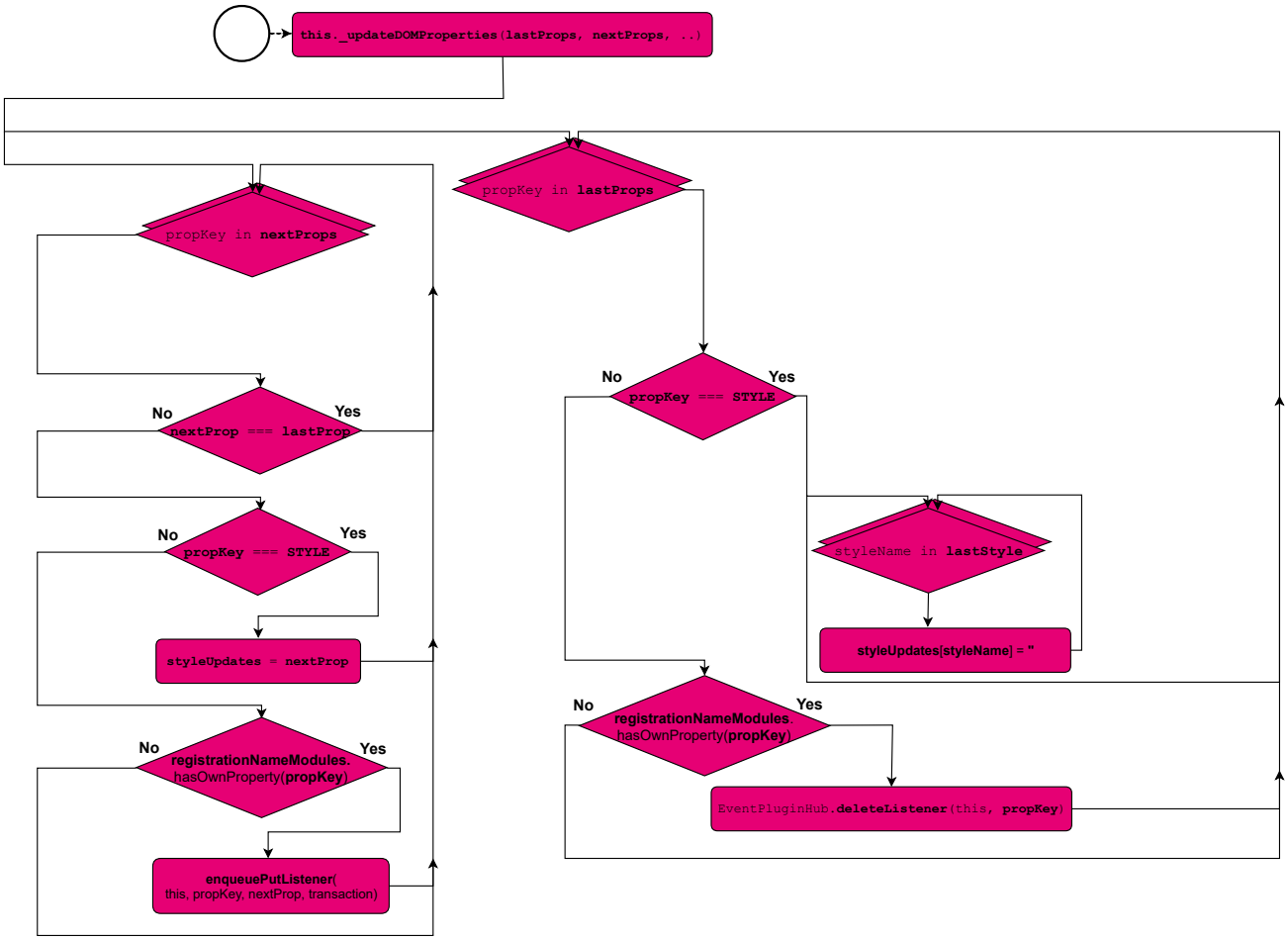
在处理完事件监听器，我们开始设置 DOM 属性名和 DOM 属性值。就像之前说的一样，对于属性们，我们确定他们不是 `RESERVED_PROPS` 中的一员，而是实际的 `prop`，例如 `children` 或者 `dangerouslySetInnerHTML`。

在处理前一个和下一个 props 的时候，我们会计算 `styleUpdates` 的配置并且现在把它传递给 `CSSPropertyOperations` 模块。

很好，我们已经完成了更新属性这一部分，让我们继续

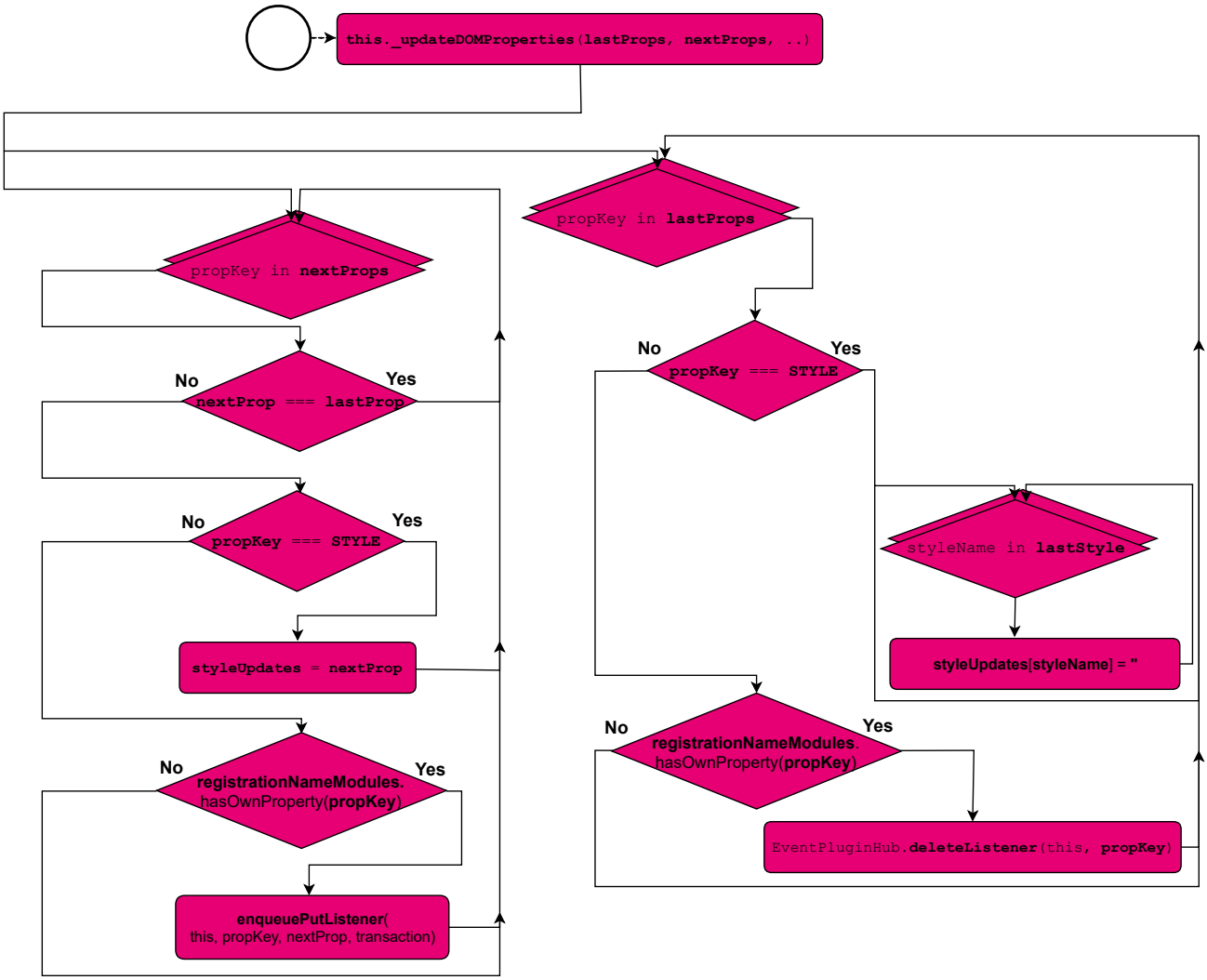
好, 第 5 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



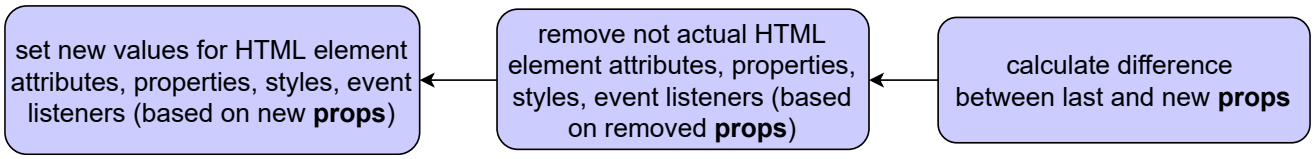
5.1 第 5 部分简化版 (点击查看大图)

然后我们适当再调整一下：



5.2 第 5 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 5 部分**的本质，并将其用于最终的 `mounting` 方案：



5.3 第 5 部分 本质 (点击查看大图)

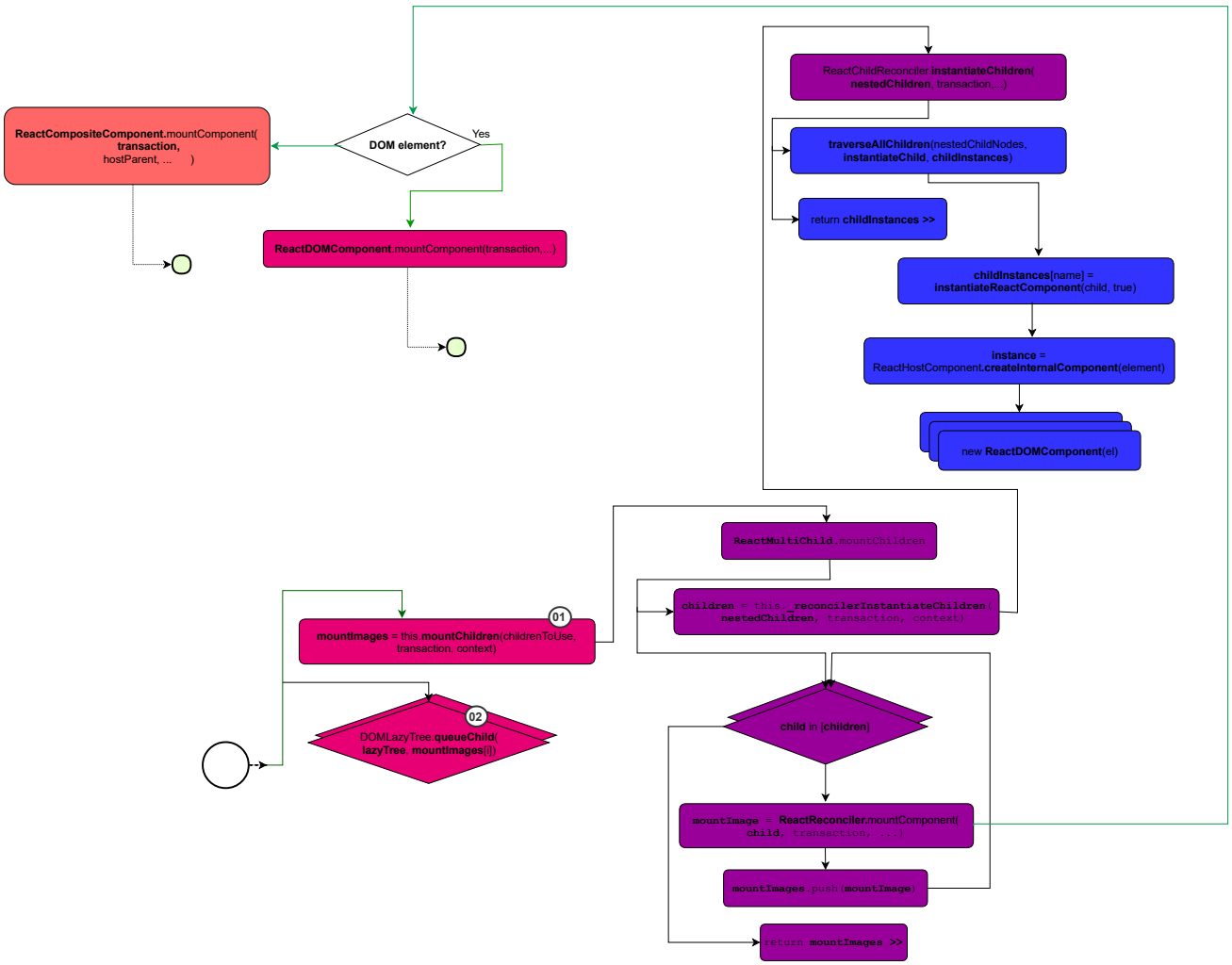
完成!

[下一节: 第 6 部分>>]

[<< 上一节: 第 4 部分]

[主页]

第 6 部分



6.0 第 6 部分（点击查看大图）

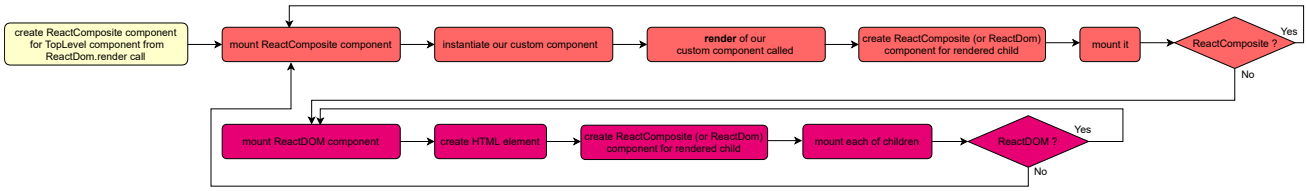
创建最初的子组件

好像组件本身已经创建完成了，现在我们可以继续创建它的子组件了。这个分为以下两步：（1）子组件应该由（`this.mountChildren`）加载，（2）并与它的父级通过（`DOMLazyTree.queueChild`）连接。我们来讨论一下子组件的挂载。

有一个单独的 `ReactMultiChild`（`src\renderers\shared\stack\reconciler\ReactMultiChild.js`）模块来操作子组件。我们来查看一下 `mountChildren` 方法。它包括两个主要任务。首先我们初始化子组件（使用 `ReactChildReconciler`）并加载他们。这里到底是什么子组件呢？它可能是一个简单的 HTML 标签或者一个其他自定义的组件。为了处理 HTML，我们需要初始化 `ReactDOMComponent`，对于自定义组件，我们使用 `ReactCompositeComponent`。加载流程也是依赖于子组件是什么类型。

再一次

如果你还在阅读这篇文章，那么现在可能是再一次阐述和整理整个过程的时候了。现在我们休息一下，重新整理下对象的顺序。



6.1 所有加载图示（点击查看大图）

1) 在React 中使用 `ReactCompositeComponent` 实例化你的自定义组件（通过使用像 `componentWillMount` 这类的组件生命周期钩子）并加载它。

2) 在加载过程中，首先会创建一个你自定义组件的实例（调用 `构造器` 函数）。

3) 然后，调用该组件的渲染函数（举个简单的例子，渲染返回的 `div`）并且 `React.createElement` 来创建 React 元素。它可以直接被调用或者通过Babel解析JSX后来替换渲染中的标签。但是，它可能不是我们所需要的，看看接下来是什么。

4) 我们对于 `div` 需要一个 DOM 组件。所以，在实例化过程中，我们从元素-对象（上文提到过）出发创建 `ReactDOMComponent` 的实例。

5) 然后，我们需要加载 DOM 组件。这实际上就意味者我们创建 DOM 元素，并加载了事件监听等。

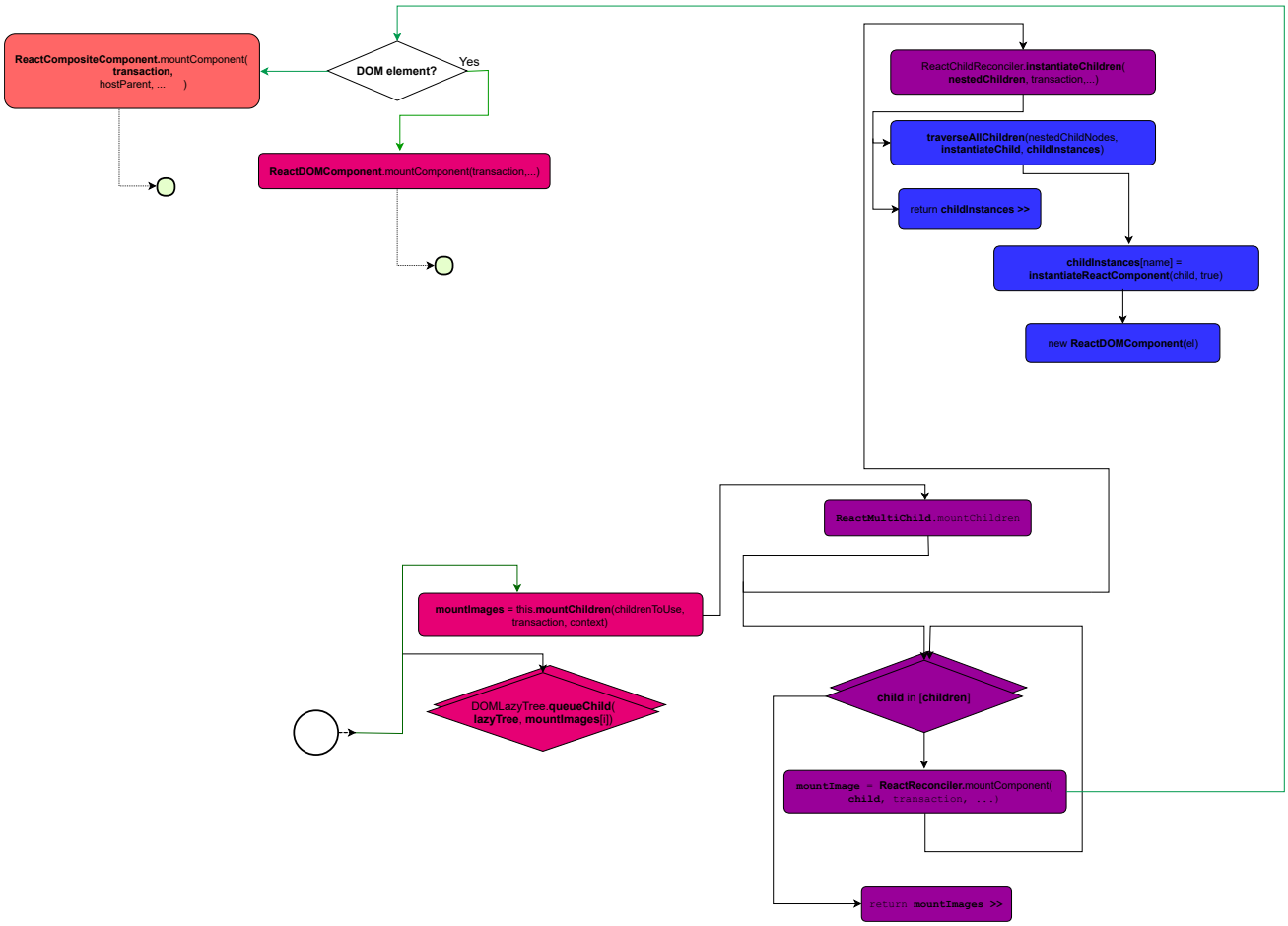
6) 然后，我们处理我们的DOM组件的直接子组件。我们创建它们的实例并且加载它们。根据子组件的是什么(自定义组件或只是HTML标签)，我们分别跳转到步骤1）或步骤5）。然后再一次处理所有的内嵌元素。

加载过程就是这个。就像你看到的一样非常直接。

加载基本完成。下一步是 `componentDidMount` 方法。大功告成。

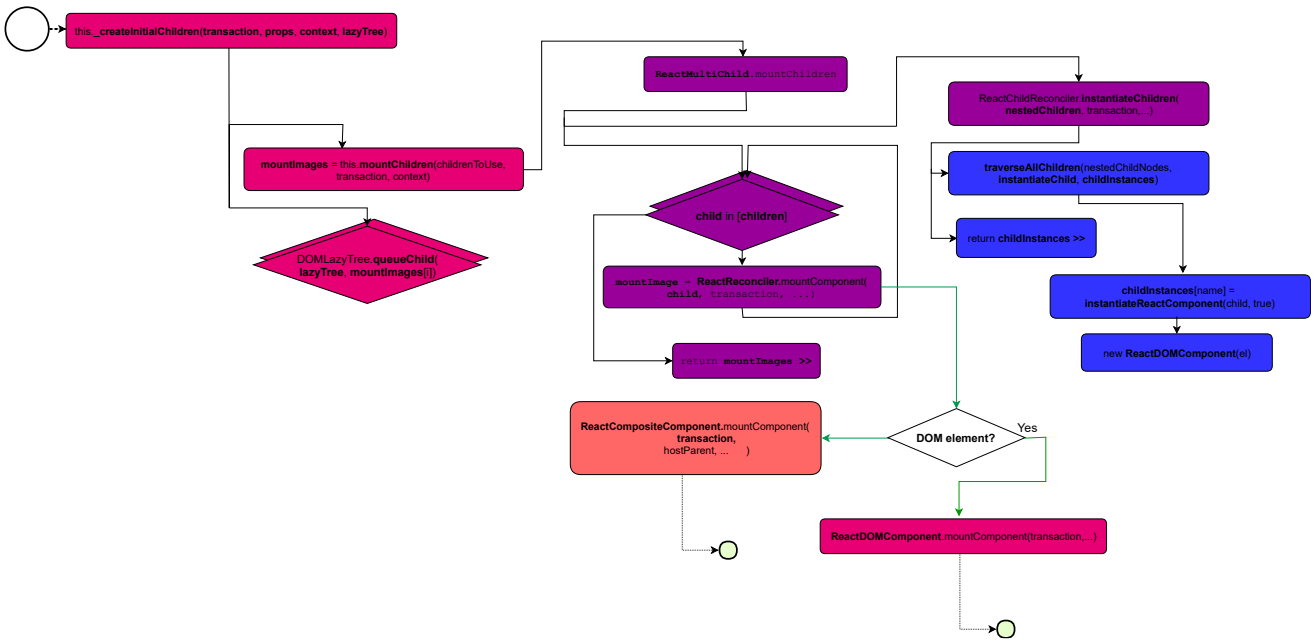
好的，我们已经完成了*第 6 部分*

让我们概括一下我们怎么到这里的。再一次看一下示例图，然后移除掉冗余的不那么重要的部分，它就变成了这样：



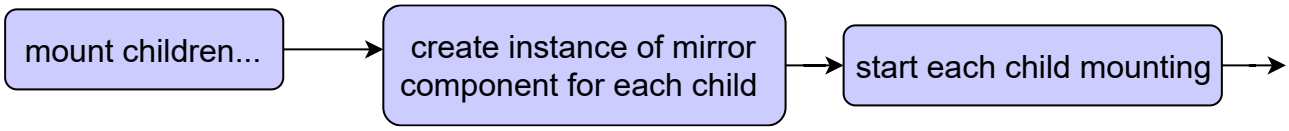
6.2 第 6 部分 简化（点击查看大图）

我们也应该尽可能的修改空格和对齐方式:



6.3 第 6 部分 简化和重构（点击查看大图）

很好。实际上它就是这儿所发生的一切。我们可以从*第 6 部分*中获得基本精髓，并将其用于最终的"加载"图表：



6.4 第 6 部分本质(点击查看大图)

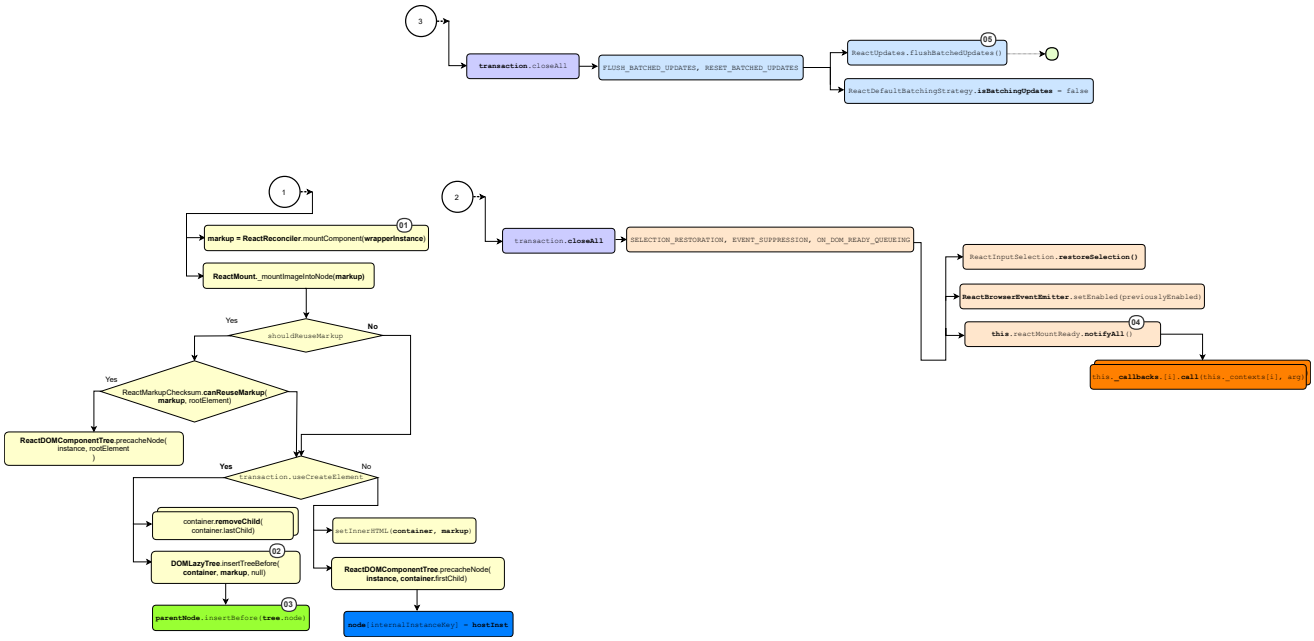
完成!

[下一节：第 7 部分 >>]

[<< 上一节：第 5 部分]

[主页]

第七部分



7.0 第七部分（可点击查看大图）

回到开始的地方

在执行加载后，我们就准备好了可以插入文档的 HTML 元素。实际上生成的是 `markup`，但是无论 `mountComponent` 是如何命名的，它们并非等同于 HTML 标记。它是一种包括子节点、节点（也就是实际 DOM 节点）等的数据结构。但是，我们最终将 HTML 元素放入在 `ReactDOM.render` 的调用中指定的容器中。在将其添加到 DOM 中时，React 会清除容器中的所有内容。 `DOMLazyTree` 是一个对树形结构执行一些操作的工具类，也是我们在使用 DOM 时实际在做的事。

最后一件事是 `parentNode.insertBefore(tree.node)`，其中 `parentNode` 是容器 `div` 节点，而 `tree.node` 实际上是 `ExampleAppliication` 的 div 节点。很好，加载创建的 HTML 元素终于被插入到文档中了。

那么，这就是所有？并未如此。也许你还记得， `mount` 的调用被包装到一个事务中。这意味着我们需要关闭这个事务。让我们来看看我们的 `close` 包装。多数情况下，我们应该恢复一些被锁定的行为，例如 `ReactInputSelection.restoreSelection()`， `ReactBrowserEventEmitter.setEnabled(previouslyEnabled)`，而且我们也需要使用 `this.reactMountReady.notifyAll` 来通知我们之前在 `transaction.reactMountReady` 中添加的所有回调函数。其中之一就是我们最喜欢的 `componentDidMount`，它将在 `close` 中被触发。

现在你对“组件已加载”的意思有了清晰的了解。恭喜！

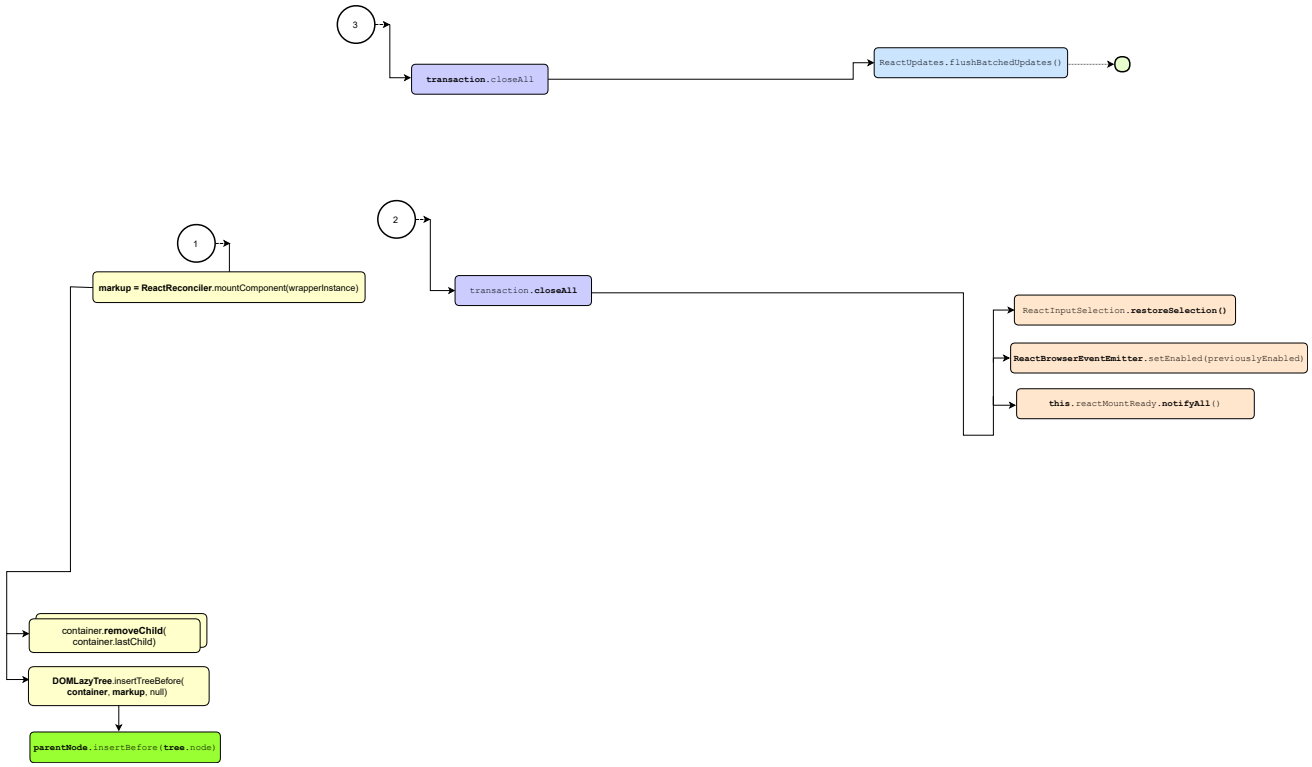
还有一个事务需要关闭

实际上，不止一个事务需要关闭。我们忘记了另一个用来包装 `ReactMount.batchedMountComponentIntoNode` 的事务。我们也需要关闭它。

这里我们需要检查将处理 `dirtyComponents` 的包装器 `ReactUpdates.flushBatchedUpdates`。听起来很有趣吗？那是好消息还是坏消息。我们只做了第一次加载，所以我们还没有脏组件。这意味着它是一个空置的调用。因此，我们可以关闭这个事务，并说批量策略更新已完成。

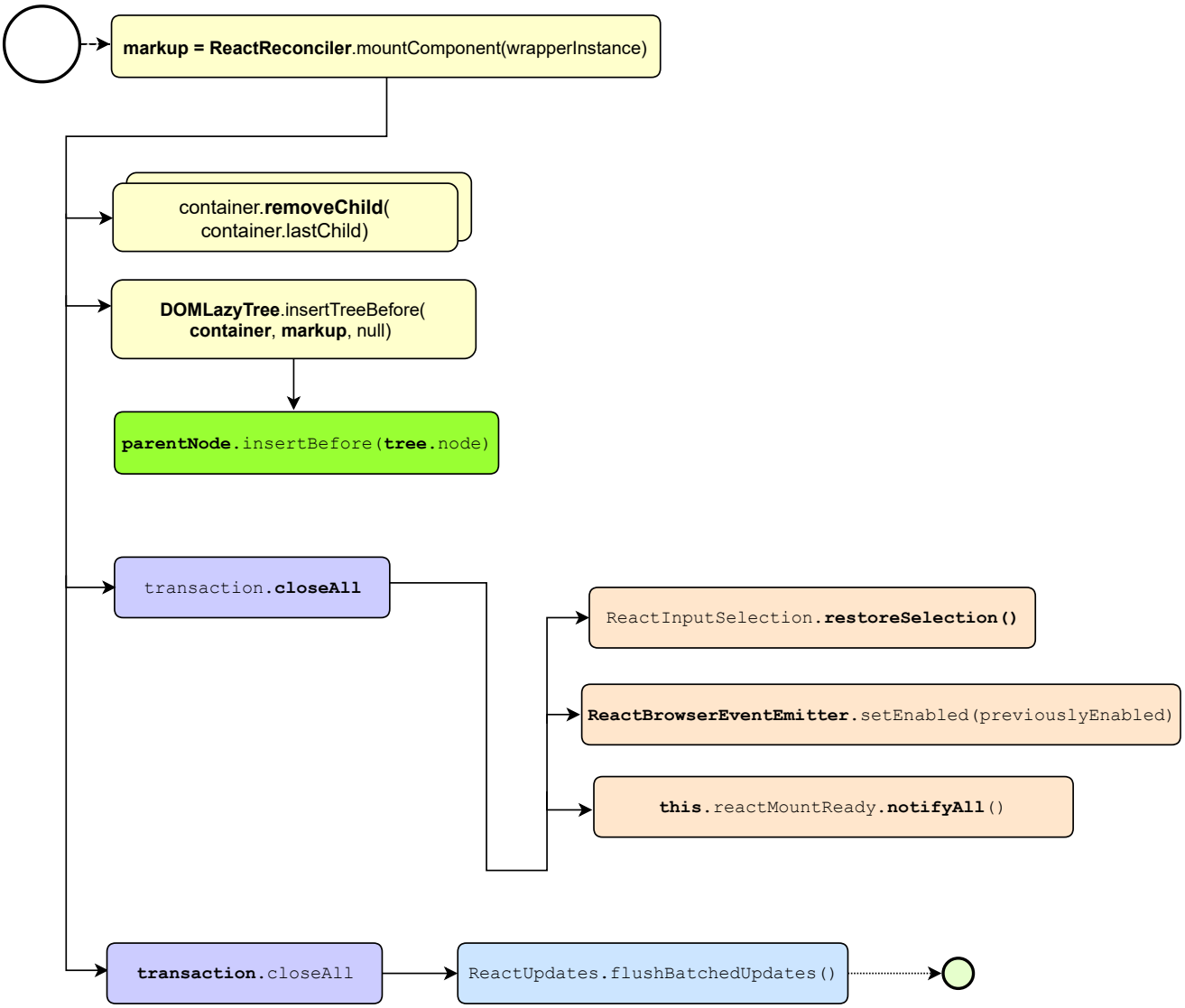
好的，我们已经完成了第 7 部分

让我们回顾一下我们是如何到达这里的。首先再看一下整体流程，然后去除多余的不太重要的部分，它就变成了：



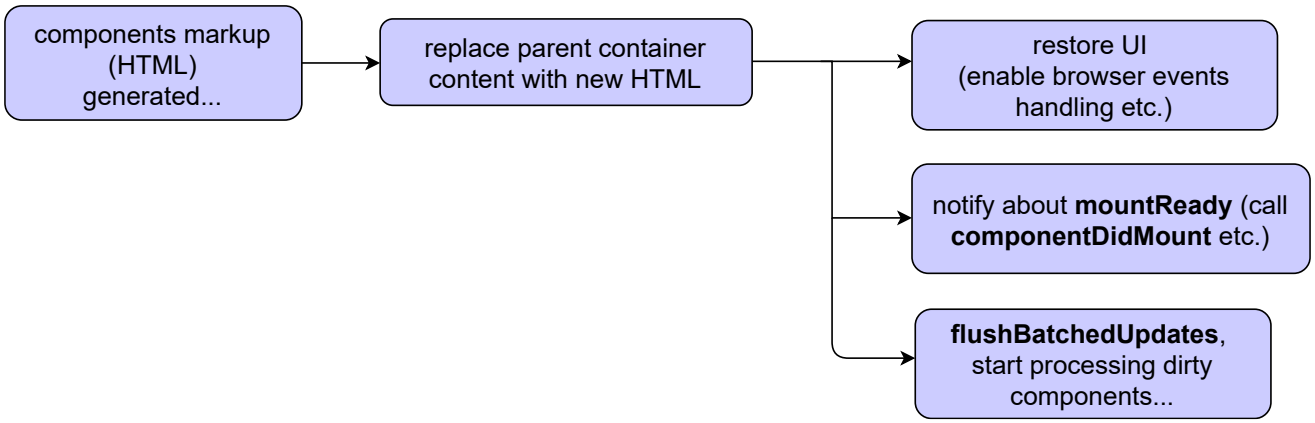
7.1 第 7 部分 简化（点击查看大图）

我们也应该修改空格和对齐：



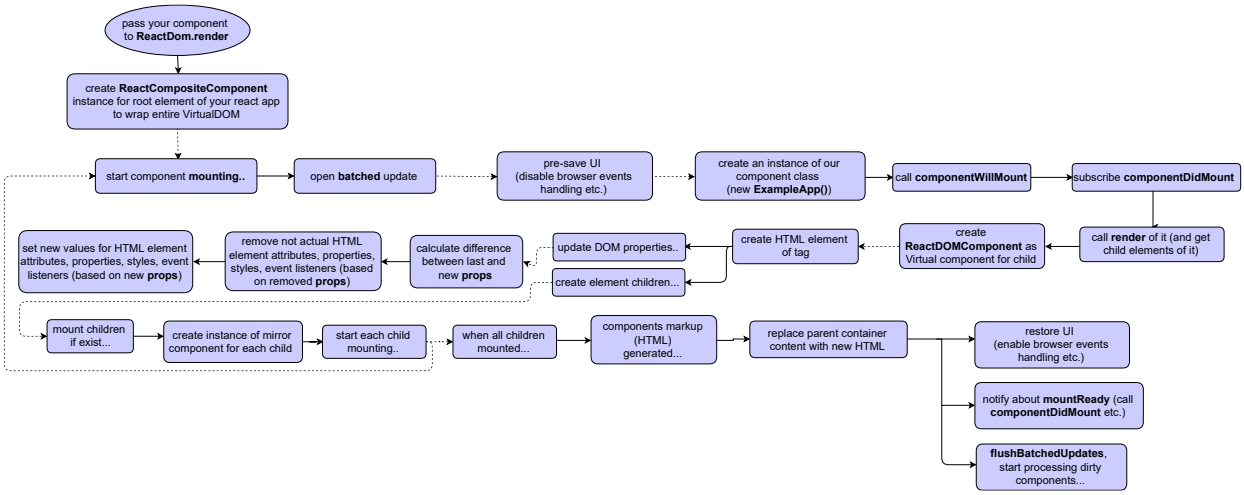
7.2 第 7 部分 简化并重构（点击查看大图）

其实这就是这里发生的所有。我们可以从第 7 部分中的重要部分来组成最终的 `mounting` 流程：



7.3 第 7 部分 基本价值（点击查看大图）

完成！其实我们完成了加载。让我们来看看下图吧！



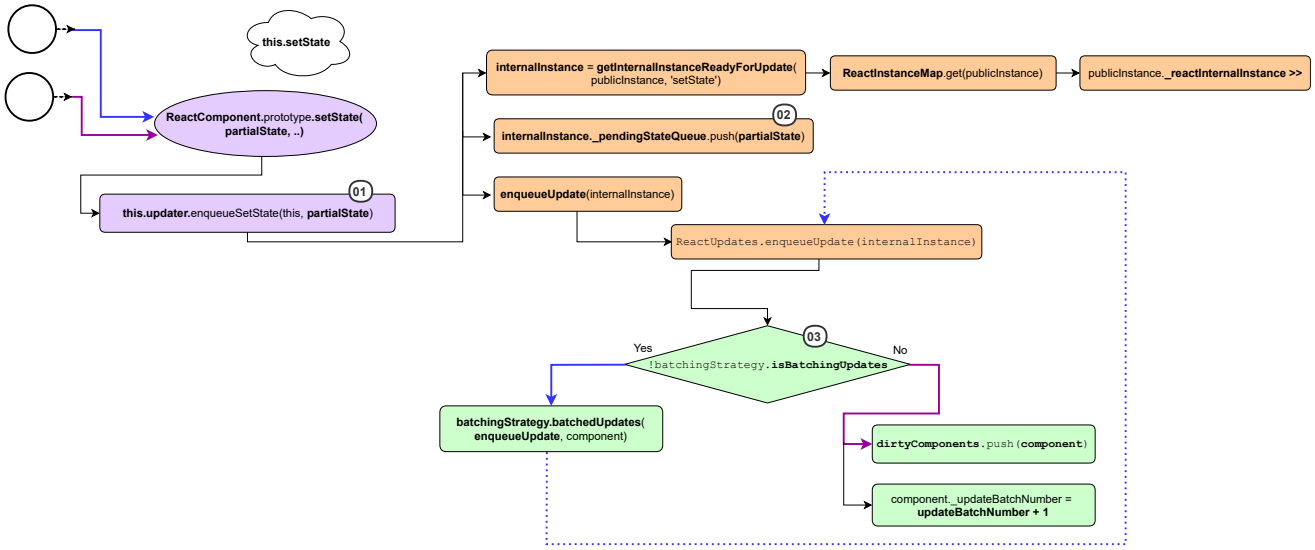
7.4 Mounting 过程（点击查看大图）

[下一节：第 8 部分 >>]

[<< 上一节：第 6 部分]

[主页]

第 8 部分



8.0 Part 8 (点击查看大图)

`this.setState`

我们已经学习了挂载的工作原理，现在从另一个角度来学习。嗯，比如 `setState` 方法，其实也很简单。

首先，为什么我们可以在自己的组件中调用 `setState` 方法呢？很明显我们的组件继承自 `ReactComponent`，这个类我们可以很方便的在 React 源码中找到。

```
//src\isomorphic\modern\class\ReactComponent.js#68
this.updater.enqueueSetState(this, partialState)
```

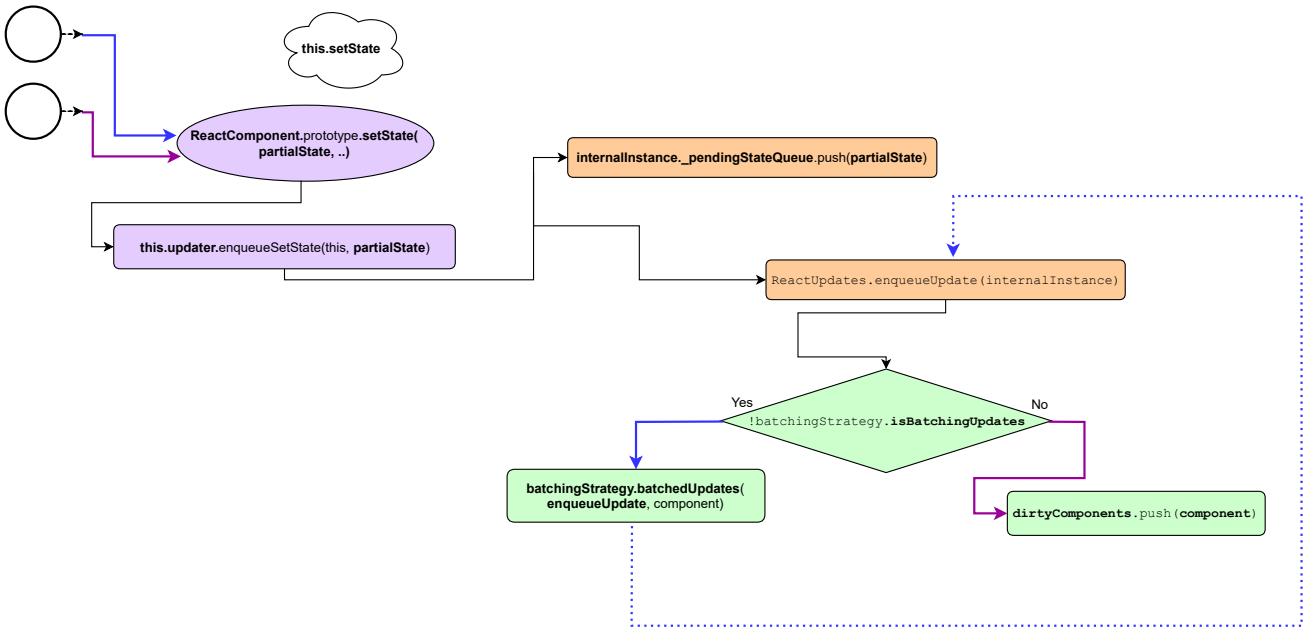
我们发现，这里有一些 `updater` 接口。什么是 `updater` 呢？在讲解挂载过程时我们讲过，在 `mountComponent` 过程中，实例会接受一个 `ReactUpdateQueue`（`src\renderers\shared\stack\reconciler\ReactUpdateQueue.js`）的引用作为 `updater` 属性。

很好，我们现在深入研究步骤 (1) 的 `enqueueSetState`。首先，它会往步骤 (2) 的 `_pendingStateQueue`（来自于内部实例。注意，这里我们说的外部实例是指用户的组件 `ExampleApplication`，而内部实例则挂载过程中创建的 `ReactCompositeComponent`）注入 `partialState`（这里的 `partialState` 就是指给 `this.setState` 传递的对象）。然后，执行 `enqueueUpdate`，这个过程会检查更新是否已经在进展中，如果是则把我们的组件注入到 `dirtyComponents` 列表中，如果不是则先初始化打开更新事务，然后把组件注入到 `dirtyComponents` 列表。

总结一下，每个组件都有自己的一组处于等待的"状态"的列表，当你在一次事务中调用 `setState` 方法，其实只是把那个状态对象注入一个队列里，它会在之后一个一个依次被合并到组件 `state` 中。调用此 `setState` 方法同时，你的组件也会被添加进 `dirtyComponents` 列表。也许你很好奇 `dirtyComponents` 是如何工作的，这就是另一个研究重点。

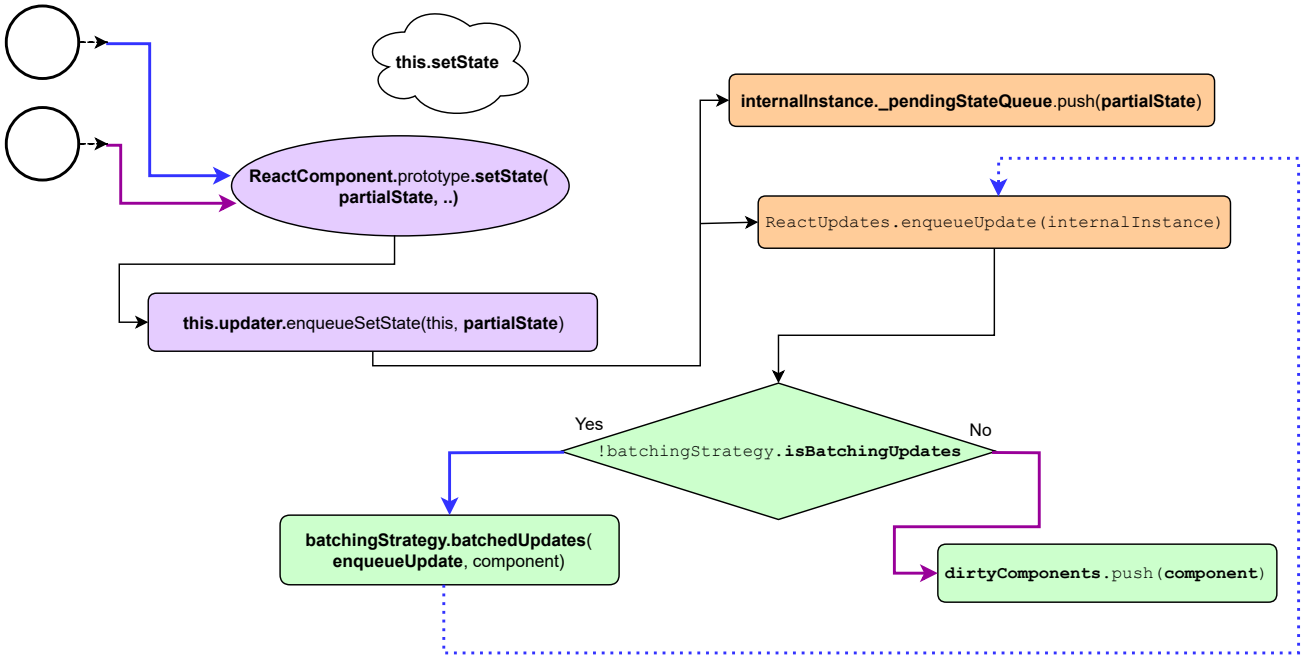
好，第 8 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



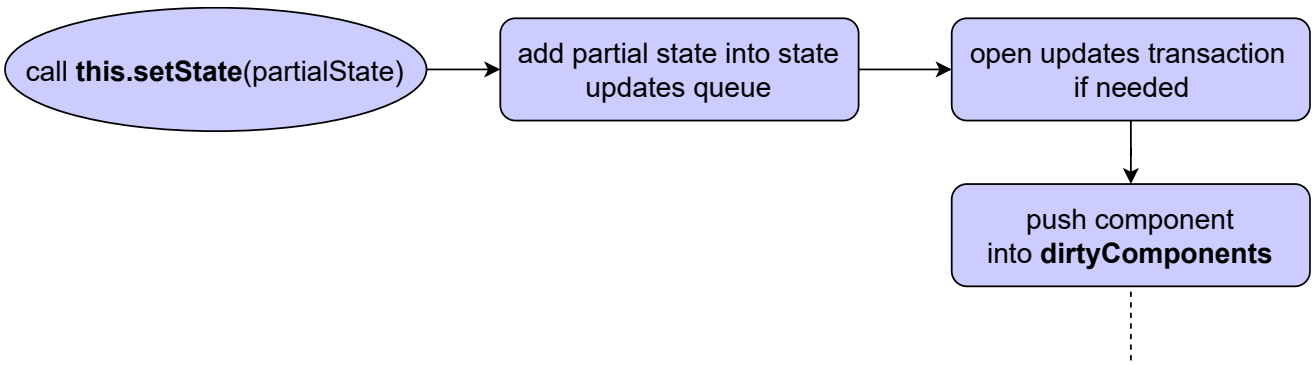
8.1 第 8 部分简化版（点击查看大图）

让我们适度在调整一下：



8.2 第 8 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 8 部分**的本质，并将其用于最终的 `updating` 方案：



8.3 Part 8 本质 (点击查看大图)

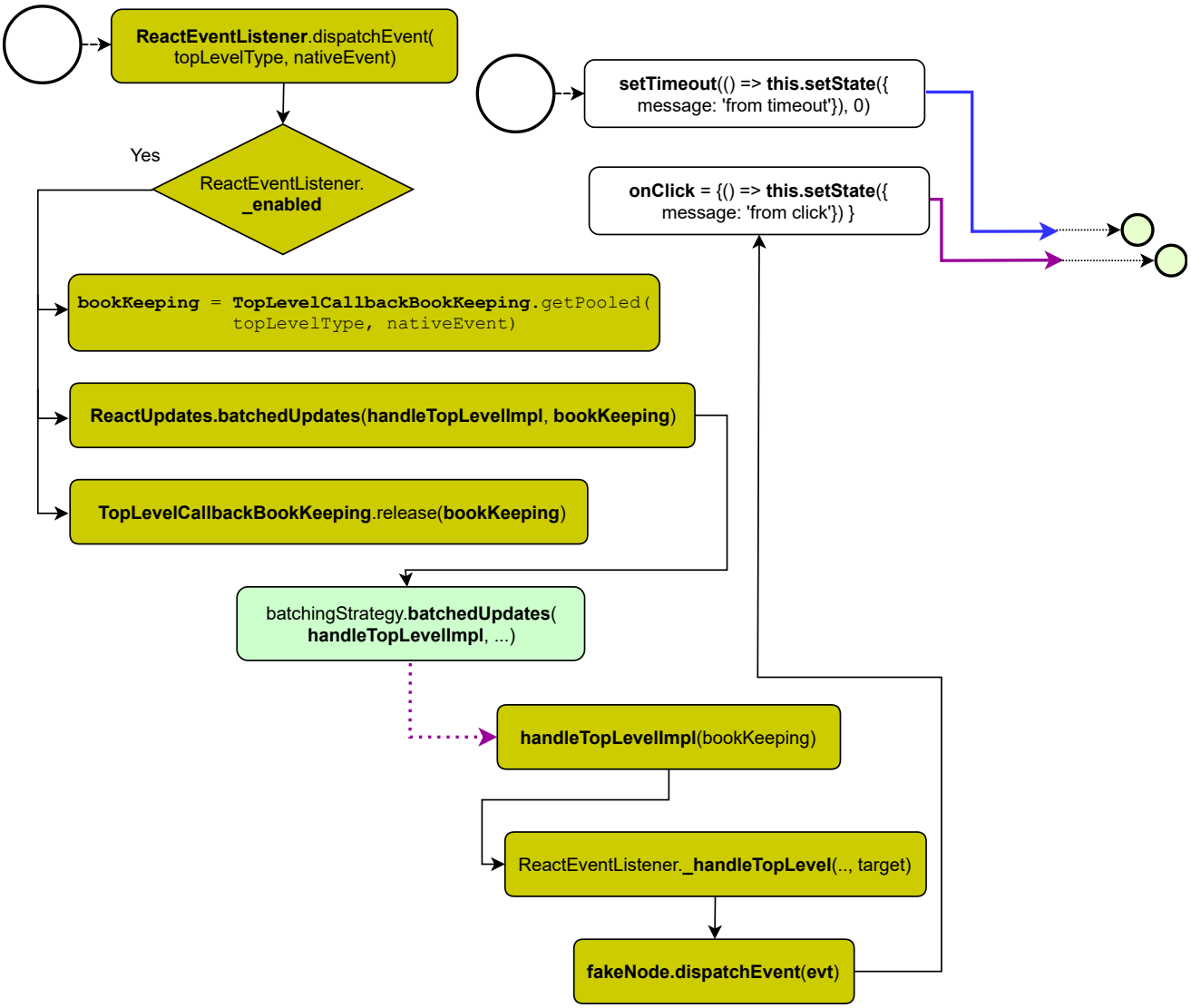
完成!

下一节: 第 9 部分>>

<< 上一节: 第 7 部分

主页

第 9 部分



9.0 第 9 部分(点击查看大图)

继续研究 `setState`

根据流程图我们发现，有很多方式来触发 `setState`。可以直接通过用户交互触发，也可能只是隐含在方法里触发。我们举两个例子：第一种情况下，它由用户的鼠标点击事件触发。而第二种情况，例如在 `componentDidMount` 里通过 `setTimeout` 调用来触发。

那么这两种方式有什么差异呢？如果你还记得 React 的更新过程是批量化进行的，这就意味着他先会收集这些更新操作，然后一起处理。当鼠标事件触发后，会被顶层先处理，然后经过多层封装器的作用，这个批更新操作才会开始。过程中你会发现，只有当步骤 (1) 的 `ReactEventListener` 是 `enabled` 的状态才会触发更新。然而你还记得在组件挂载过程中，`ReactReconcileTransaction` 中的一个封装器会使它 `disabled` 来确保挂载的安全。那么 `setTimeout` 案例是怎样的呢？这个也很简单，在把组件丢进 `dirtyComponents` 列表前，React会确保事务已经开始，那么，之后他应该会被关闭，然后一起处理列表中的组件。

就像你所知道的那样，React 有实现很多 “synthetic事件”，一些 “语法糖”，实际上包裹着原生事件。随后，他会表现为我们很熟悉的原生事件。你可以看下面的代码注释：

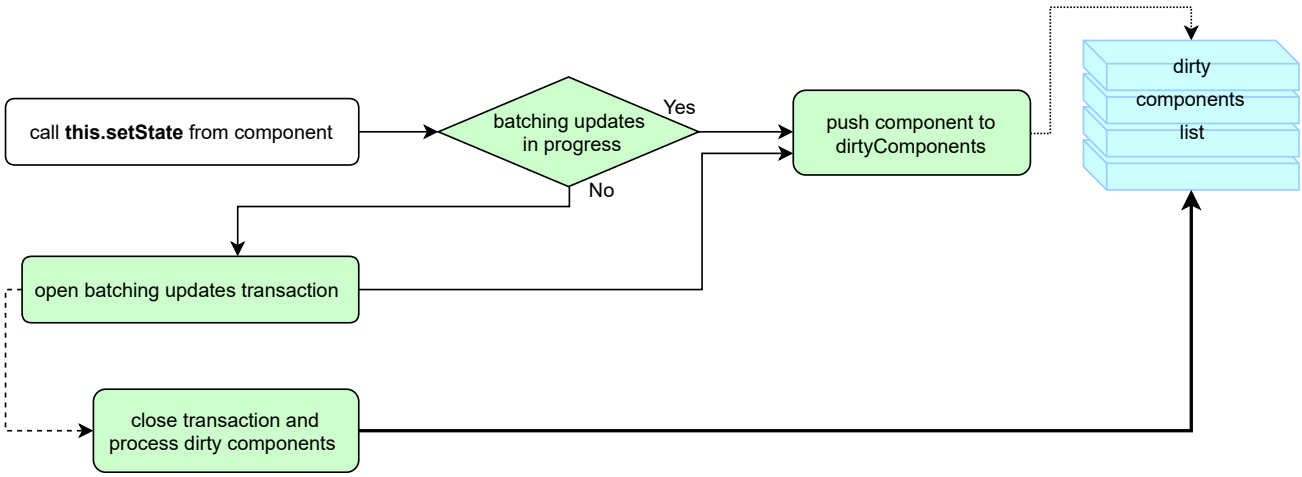
> 实验过程为了更方便和调试工具整合，我们模拟一个真实浏览器事件

```
var fakeNode = document.createElement('react');

```

```
ReactErrorUtils.invokeGuardedCallback = function (name, func, a) {
      var boundFunc = func.bind(null, a);
      var evtType = 'react-' + name;

      fakeNode.addEventListener(evtType, boundFunc, false);

      var evt = document.createEvent('Event');
      evt.initEvent(evtType, false, false);

      fakeNode.dispatchEvent(evt);
      fakeNode.removeEventListener(evtType, boundFunc, false);
};
```

好，回到我们的更新，让我们总结一下，整个过程是：

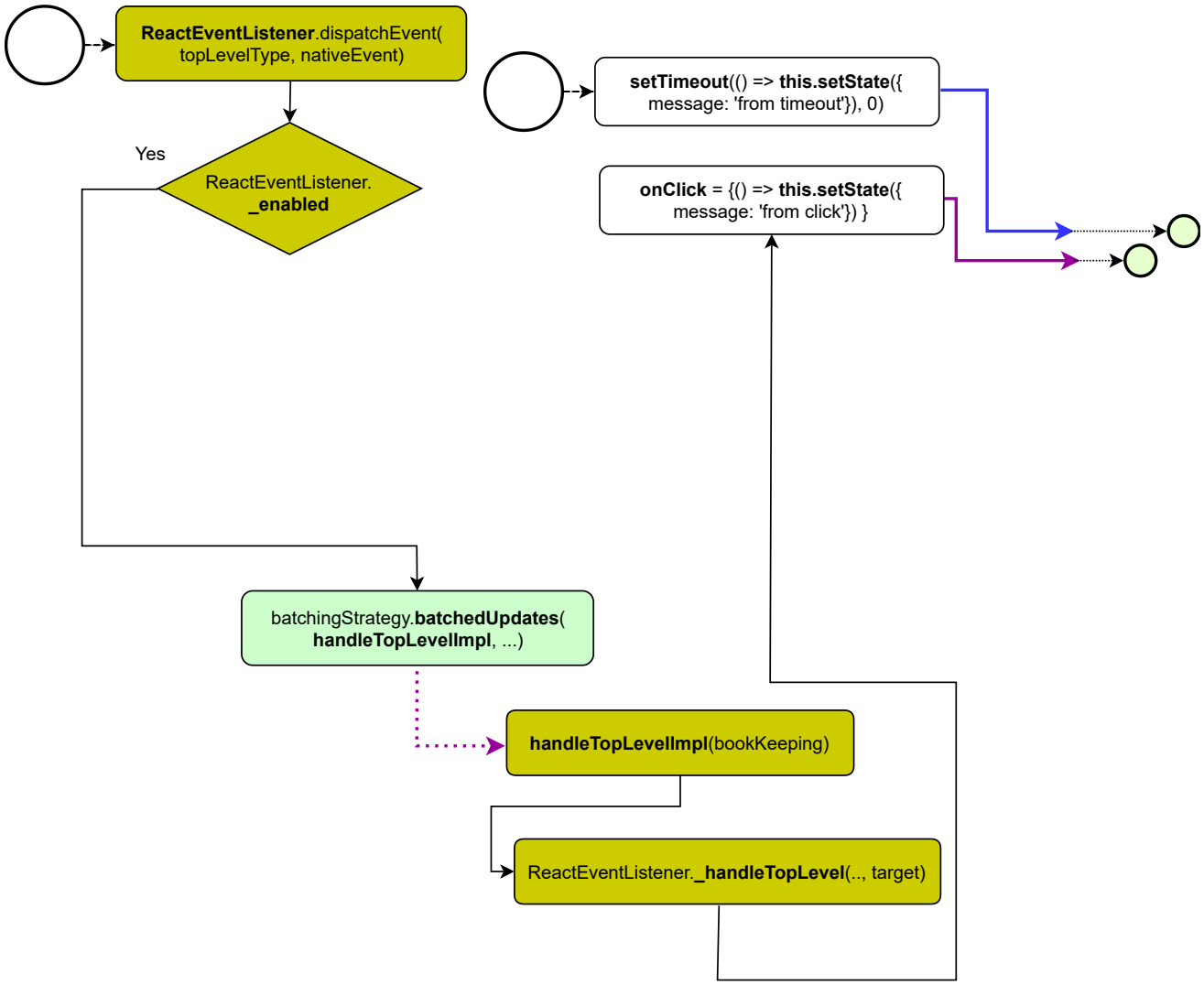
1. 调用 setState
2. 如果批处理事务没有打开，则打开
3. 把受影响的组件添加入 `dirtyComponents` 列表
4. 在调用 `ReactUpdates.flushBatchedUpdates` 的同时关闭事务，并处理在所有 `dirtyComponents` 列表中的组件



9.1 `setState` 执行过程(点击查看大图)

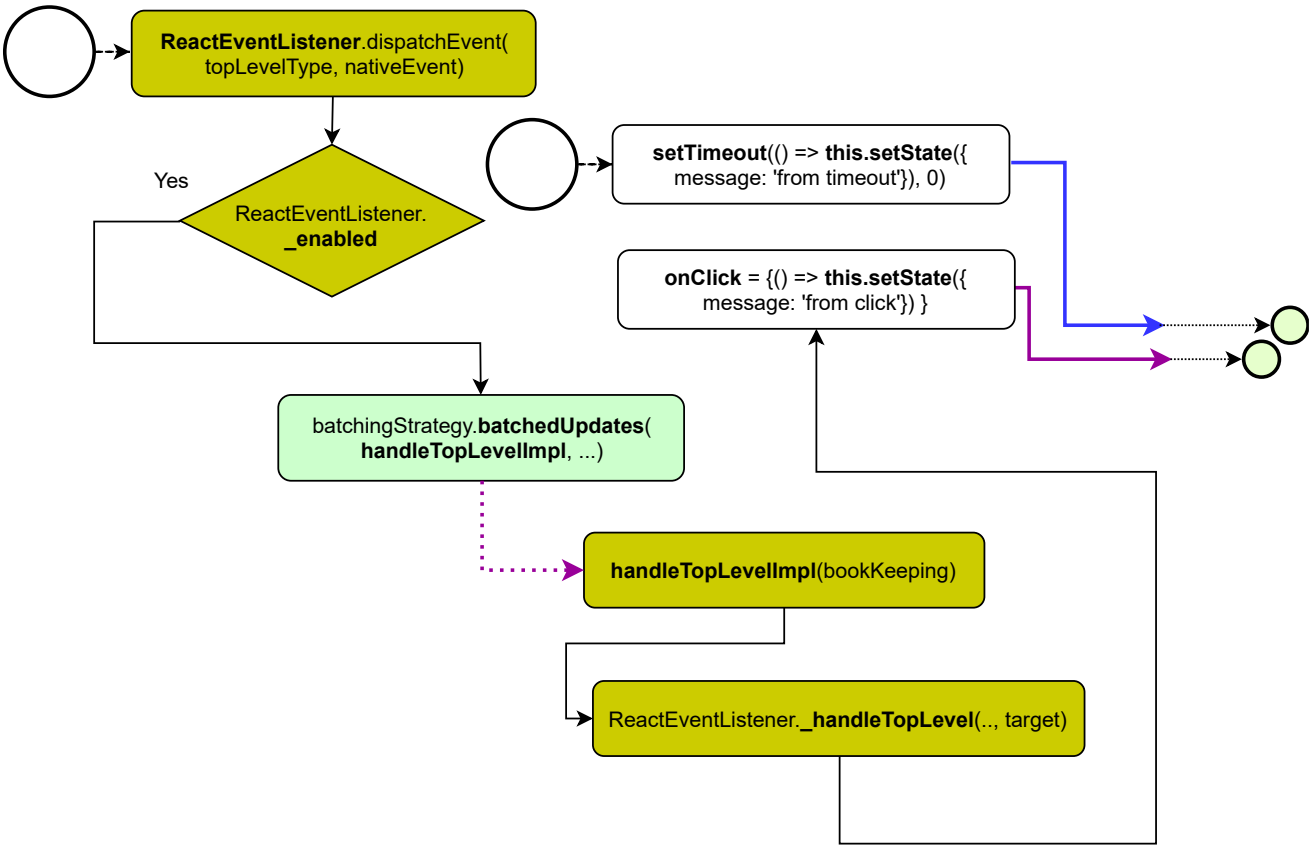
好，第 9 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



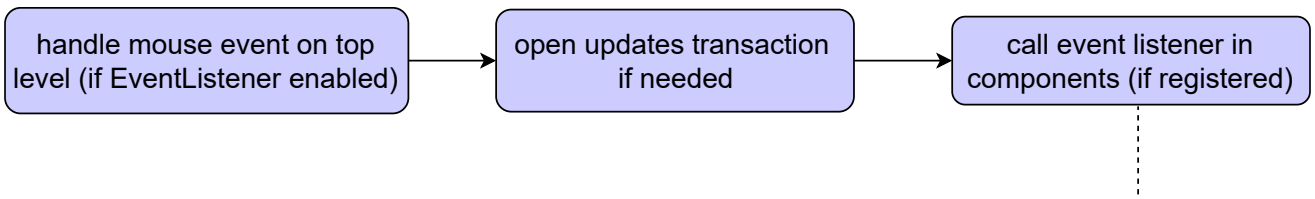
9.2 第 9 部分简化版 (点击查看大图)

然后我们适当再调整一下：



9.3 第 9 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 9 部分**的本质，并将其用于最终的 `updating` 方案：



9.4 第 9 部分本质 (点击查看大图)

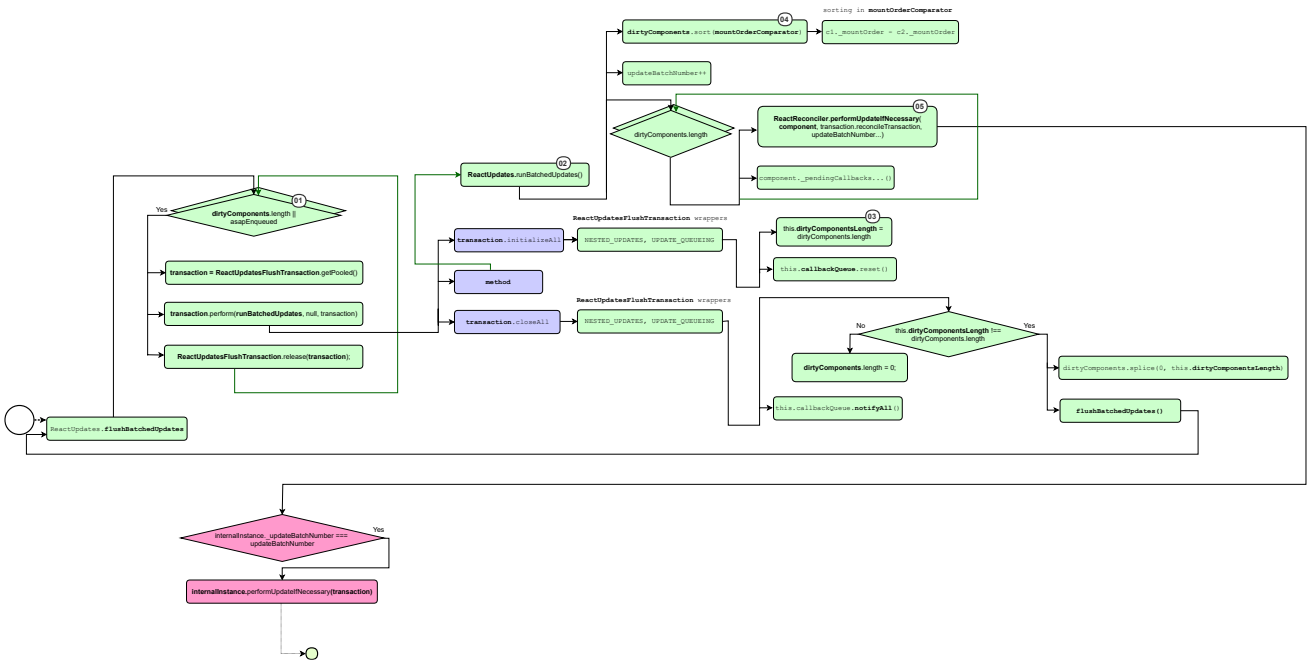
完成!

下一节: 第 10 部分>>

<< 上一节: 第 8 部分

主页

第 10 部分



10.0 第十部分 (点击查看大图)

脏组件

就像流程图所示那样，React 会遍历步骤 (1) 的 `dirtyComponents`，并且通过事务调用步骤 (2) 的 `ReactUpdates.runBatchedUpdates`。事务? 又是一个新的事务，它怎么工作呢，我们一起来看。

这个事务的类型是 `ReactUpdatesFlushTransaction`，之前我们也说过，我们需要通过事务包装器来理解事务具体干什么。以下是从代码注释中获得的启示：

> ReactUpdatesFlushTransaction 的封装器组会清空 dirtyComponents 数组，并且执行 mount-ready 处理器组压入队列的更新 (mount-ready 处理器是指那些在 mount 成功后触发的生命周期函数。例如 `componentDidUpdate`)

但是，不管怎样，我们需要证实它。现在有两个 `wrappers`： `NESTED_UPDATES` 和 `UPDATE_QUEUEING`。在初始化的过程中，我们存下步骤 (3) 的 `dirtyComponentsLength`。然后观察下面的 `close` 处，React 在更新过程中会不断检查对比 `dirtyComponentsLength`，当一批脏组件变更了，我们把它们从中数组中移出并再次执行 `flushBatchedUpdates`。 你看, 这里并没有什么黑魔法，每一步都清晰简单。

然而... 一个神奇的时刻出现了。 `ReactUpdatesFlushTransaction` 复写了 `Transaction.perform` 方法。因为它实际上是从 `ReactReconcileTransaction` (在挂载的过程中应用到的事务，用来保障应用 `state` 的安全) 中获得的行为。因此在 `ReactUpdatesFlushTransaction.perform` 方法里， `ReactReconcileTransaction` 也被使用到，这个事务方法实际上又被封装了一次。

因此，从技术角度看，它可能形如：

```
[NESTED_UPDATES, UPDATE_QUEUEING].initialize()
[SELECTION_RESTORATION, EVENT_SUPPRESSION, ON_DOM_READY_QUEUEING].initialize()

method -> ReactUpdates.runBatchedUpdates

[SELECTION_RESTORATION, EVENT_SUPPRESSION, ON_DOM_READY_QUEUEING].close()
[NESTED_UPDATES, UPDATE_QUEUEING].close()
```

我们之后会回到这个事务，再次理解它是如何帮助我们的。但是现在，让我们来看步骤 (2) `ReactUpdates.runBatchedUpdates` (`\src\renderers\shared\stack\reconciler\ReactUpdates.js#125`)。

我们要做的第一件事就是给 `dirtyComponets` 排序，我们来看步骤 (4)。怎么排序呢？通过 `mount order` (当实例挂载时组件获得的序列整数)，这将意味着父组件 (先挂载) 会被先更新，然后是子组件，然后往下以此类推。

下一步我们提升批号 `updateBatchNumber`，批号是一个类似当前差分对比更新状态的 ID。 代码注释中提到：

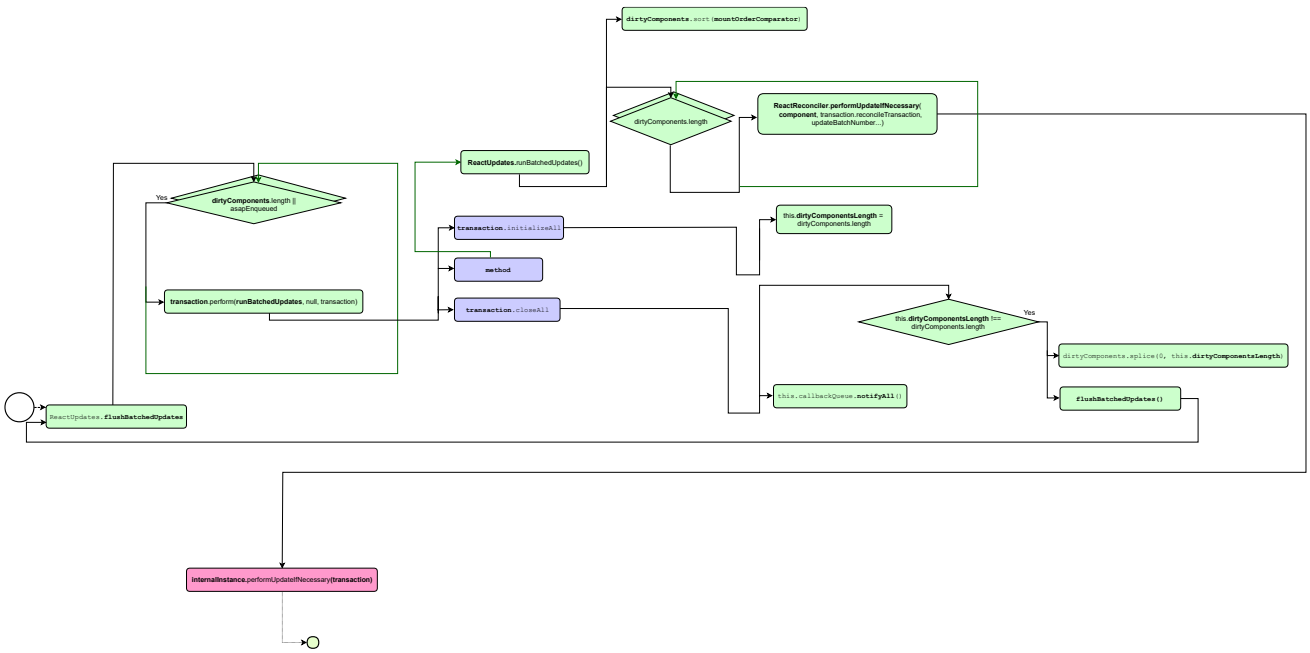
> '任何在差分对比更新过程中压入队列的更新必须在整个批处理结束后执行。 否则, 如果 dirtyComponents 为 [A, B]。 其中 A 有孩子 B 和 C, 那么如果 C 的渲染压入一个更新给 B，则 B 可能在一个批次中更新两次 (由于 B 已经更新了，我们应该跳过它，而唯一能感知的方法就是检查批号)。'

这将避免重复更新同一个组件。

非常好，最终我们遍历 `dirtyComponents` 并传递其每个组件给步骤 (5) 的 `ReactReconciler.performUpdateIfNecessary`，这也是 `ReactCompositeComponent` 实例里调用 `performUpdateIfNecessary` 的地方。然后，我们将继续研究 `ReactCompositeComponent` 代码以及它的 `updateComponent` 方法，在那里我们会发现更多有趣的事，让我们继续深入研究。

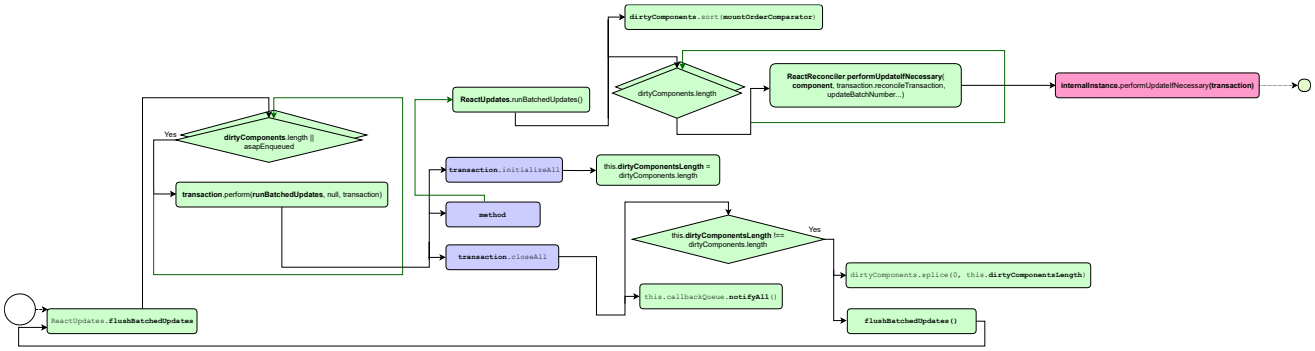
好, 第 10 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



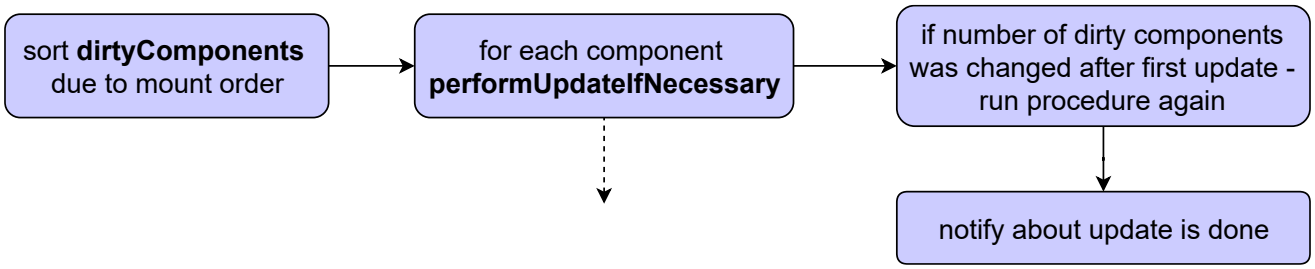
10.1 第 10 部分简化版 (点击查看大图)

让我们适度调整一下：



10.2 第 10 部分重构与简化 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 10 部分**的本质，并将其用于最终的 `updating` 方案：



10.3 第 10 部分 本质 (点击查看大图)

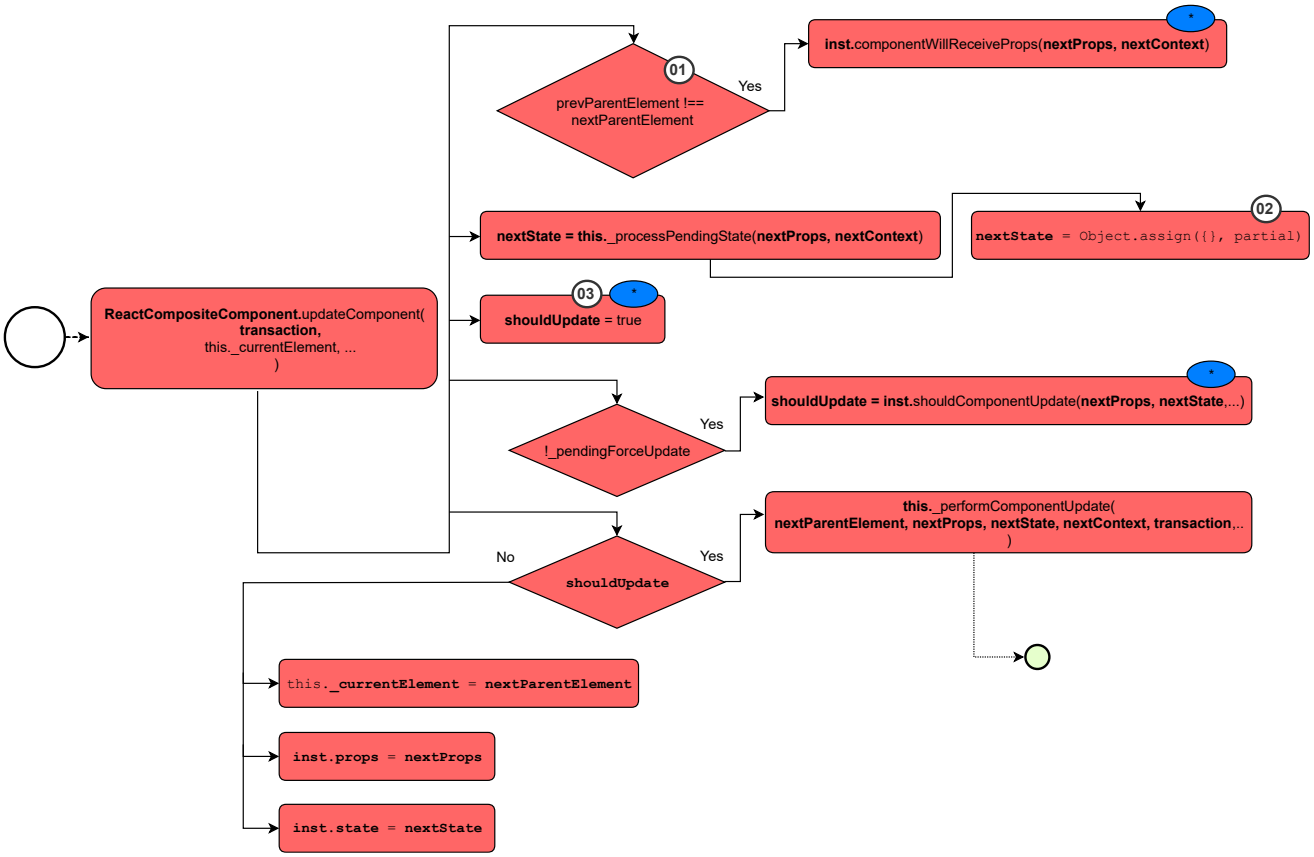
完成!

下一节: 第 11 部分>>

<< 上一节: 第 9 部分

主页

第 11 部分



11.0 第 11 部分(点击查看大图)

更新组件方法

源码中的注释是这样介绍这个方法的：

> 对一个已经挂载后的组件执行再更新操作的时候，`componentWillReceiveProps` 以及 `shouldComponentUpdate` 方法会被调用，然后 (假定这个更新有效) 调用其他更新中其余的生命周期钩子方法，并且需要变化的 DOM 也会被更新。默认情况下这个过程会使用 React 的渲染和差分对比更新算法。对于一些复杂的实现，客户可能希望重写这步骤。

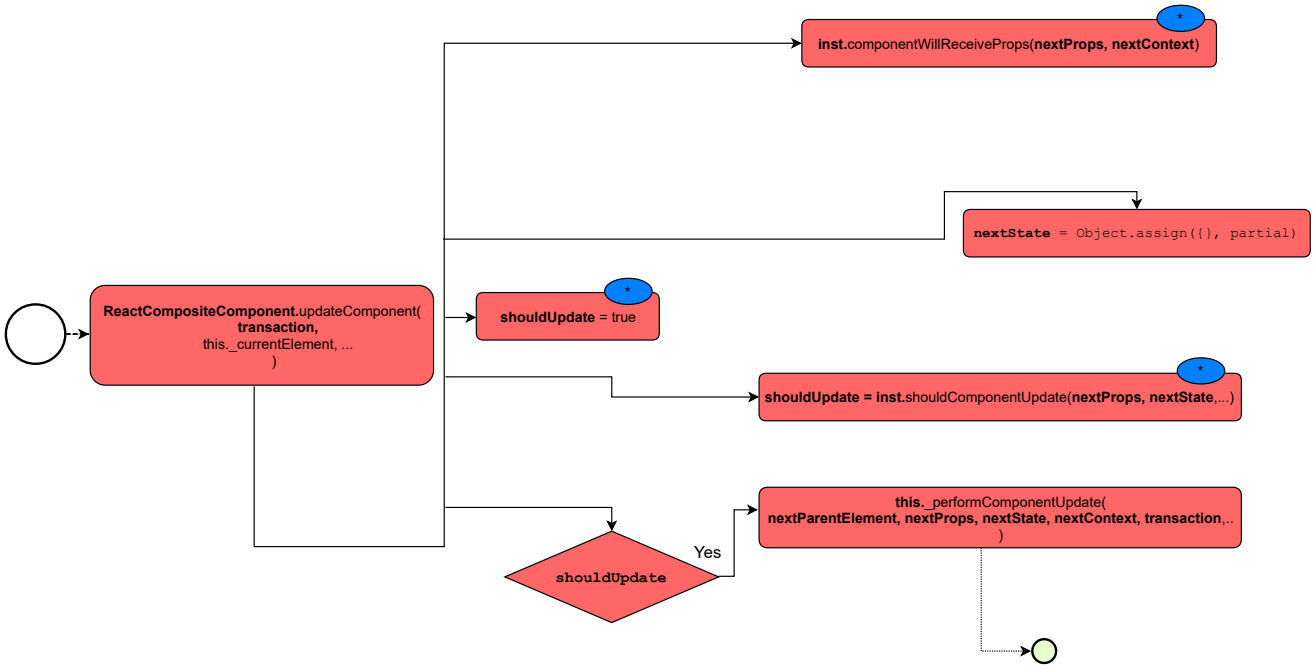
很好... 听起来很合理。

首先我们会去检查步骤 (1) 的 `props` 是否改变了，原理上讲，`updateComponent` 方法会在 setState 方法被调用或者 props 变化这两种情况下使用。如果 `props` 确实改变了，那么生命周期函数 `componentWillReceiveProps` 就会被执行. 接着, React 会根据 `pending state queue` (指我们之前设置的 `partialState` 队列，现在可能形如 [{ message: "click state message" }]) 重新计算步骤 (2) 的 `nextState`。当然在只有 props 更新的情况下， state 是不会受到影响的。

很好，下一步，我们把 `shouldUpdate` 初始化为步骤 (3) 的 `true`。这里可以看出即使 `shouldComponentUpdate` 没有申明，组件也会按照此默认行为更新。然后检查一下 `force update` 的状态，因为我们也可以在组件里调用 `forceUpdate` 方法，不管 `state` 和 `props` 是不是变化，都强制更新。当然，React 的官方文档不推荐这样的实践。在使用 `forceUpdate` 的情况下，组件将会被持久化的更新，否则， `shouldUpdate` 将会是 `shouldComponentUpdate` 的返回结果。如果 `shouldUpdate` 为否，组件不应该更新时，React 依然会设置新的 `props` and `state`, 不过会跳过更新的余下部分。

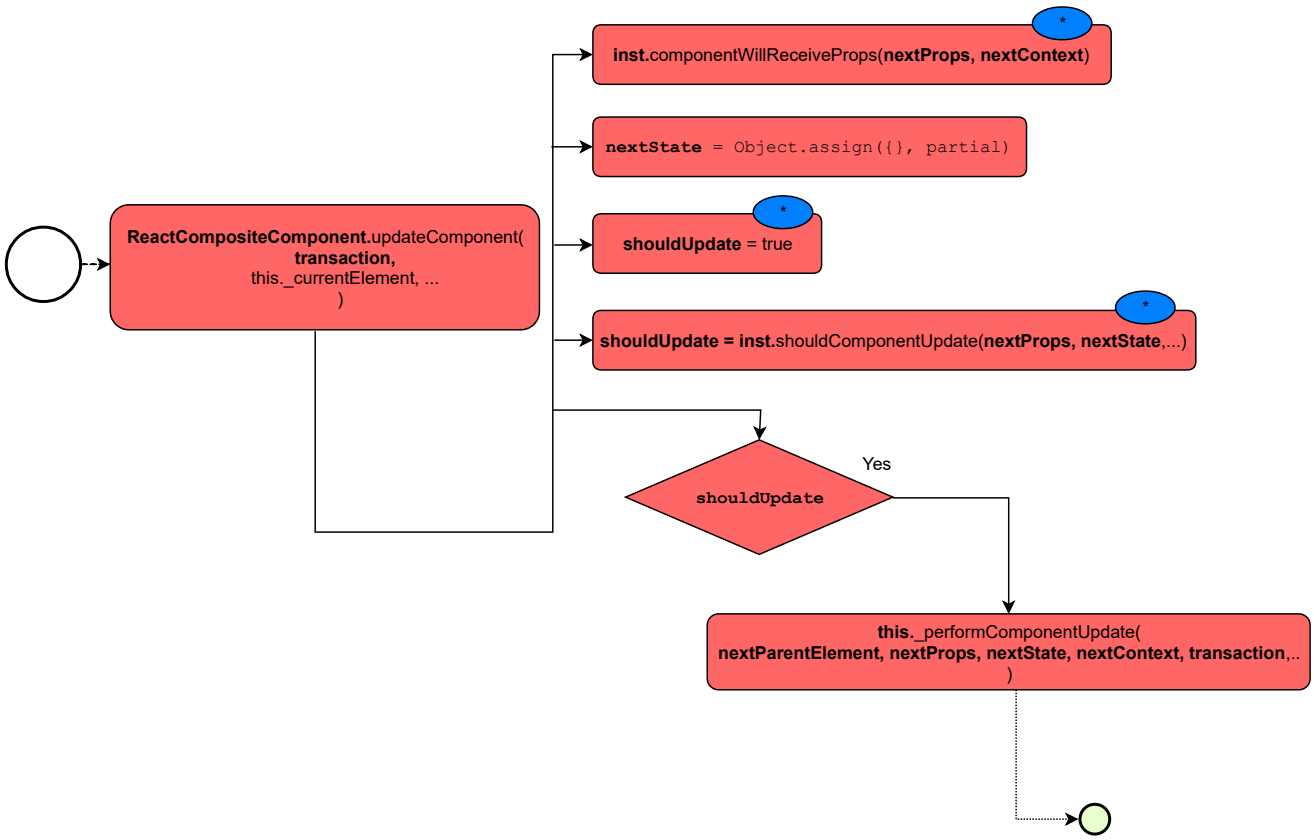
好, 第 11 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



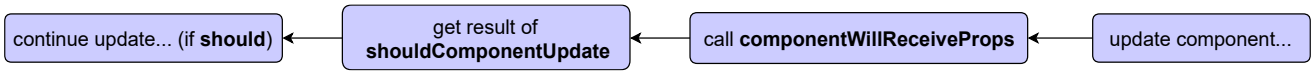
11.1 第 11 部分简化版（点击查看大图）

然后我们适当再调整一下：



11.2 第 11 部分简化和重构（点击查看大图）

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 11 部分**的本质，并将其用于最终的 `updating` 方案：



11.3 第 11 部分本质(点击查看大图)

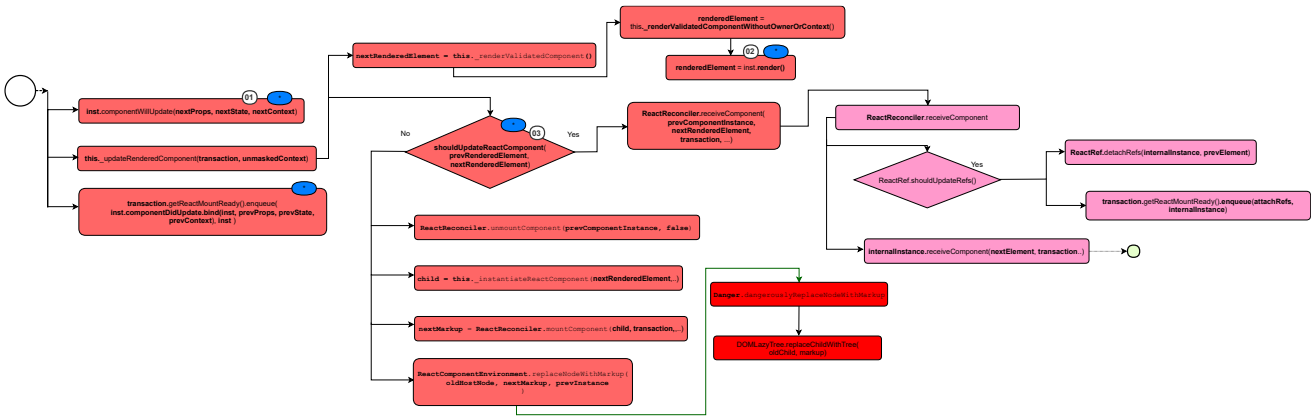
完成!

[下一节: 第 12 部分>>]

[<< 上一节: 第 10 部分]

[主页]

第 12 部分



12.0 第 12 部分(点击查看大图)

当组件确实需要更新...

现在我们已经到更新行为的开始点，此时应该先调用步骤 (1) 的 `componentWillUpdate` (当然必须声明过) 的生命周期钩子。然后重绘组件并且把另一个知名的方法 `componentDidUpdate` 的调用压入队列 (推迟是因为它应该在更新操作结束后执行)。那怎么重绘呢？实际上这时候会调用组件的 render 方法，并且相应的更新 DOM。所以第一步，调用实例 (`ExampleApplication`) 中步骤 (2) 的 `render` 方法, 并且存储更新的结果 (这里会返回 React 元素)。然后我们会和之前已经渲染的元素对比并决策出哪些 DOM 应该被更新。

这个部分是 React 杀手级别的功能，它避免冗余的 DOM 更新，只更新我们需要的部分以提高性能。

我们来看源码对步骤 (3) 的 `shouldUpdateReactComponent` 方法的注释：

> 决定现有实例的更新是部分更新，还是被移除还是被一个新的实例替换

因此，通俗点讲，这个方法会检测这个元素是否应该被彻底的替换, 在彻底替换掉情况下，旧的部分需要先被 `unmounted` (卸载) ，然后从 `render` 获取的新的部分应该被挂载，然后把挂载后获得的元素替换现有的。这个方法还会检测是否一个元素可以被部分更新。彻底替换元素的主要条件是当一个新的元素是空元素 (意即被 render 逻辑移除了)。或者它的标签不同，比如原先是一个 `div` ，然而是现在是其它的标签了。让我们来看以下代码，表达的非常清晰。

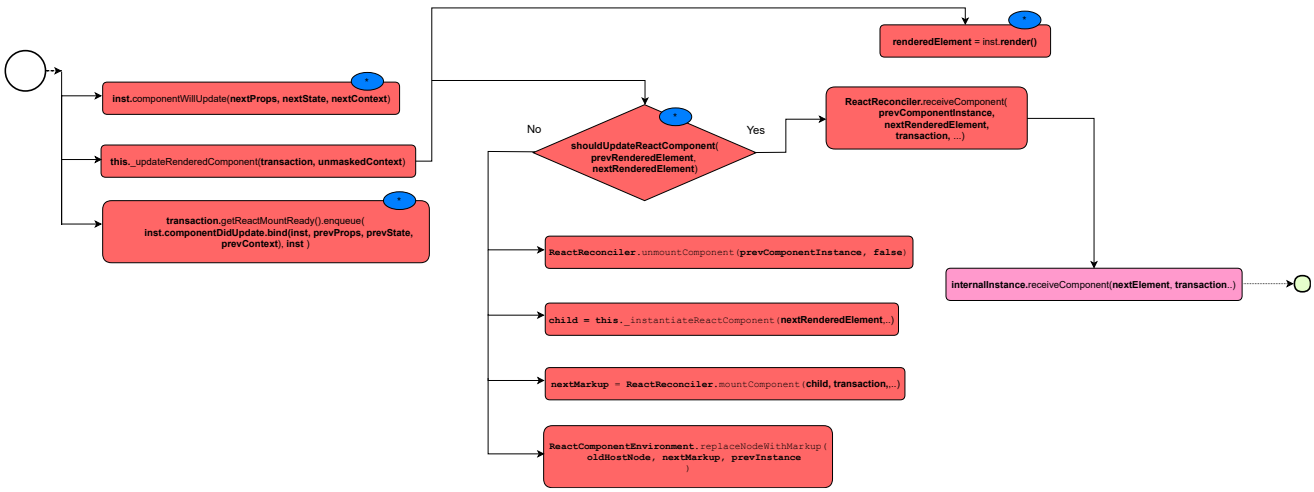
```
///src/renderers/shared/shared/shouldUpdateReactComponent.js#25

function shouldUpdateReactComponent(prevElement, nextElement) {
    var prevEmpty = prevElement === null || prevElement === false;
```

```
    var nextEmpty = nextElement === null || nextElement === false;
    if (prevEmpty || nextEmpty) {
        return prevEmpty === nextEmpty;
    }

    var prevType = typeof prevElement;
    var nextType = typeof nextElement;
    if (prevType === 'string' || prevType === 'number') {
        return (nextType === 'string' || nextType === 'number');
    } else {
        return (
            nextType === 'object' &&
            prevElement.type === nextElement.type &&
            prevElement.key === nextElement.key
        );
    }
}
```

很好，实际上我们的 `ExampleApplication` 实例仅仅更新了 state 属性，并没有怎么影响 `render`。到现在我们可以进入下一个场景，`update` 后的反应。

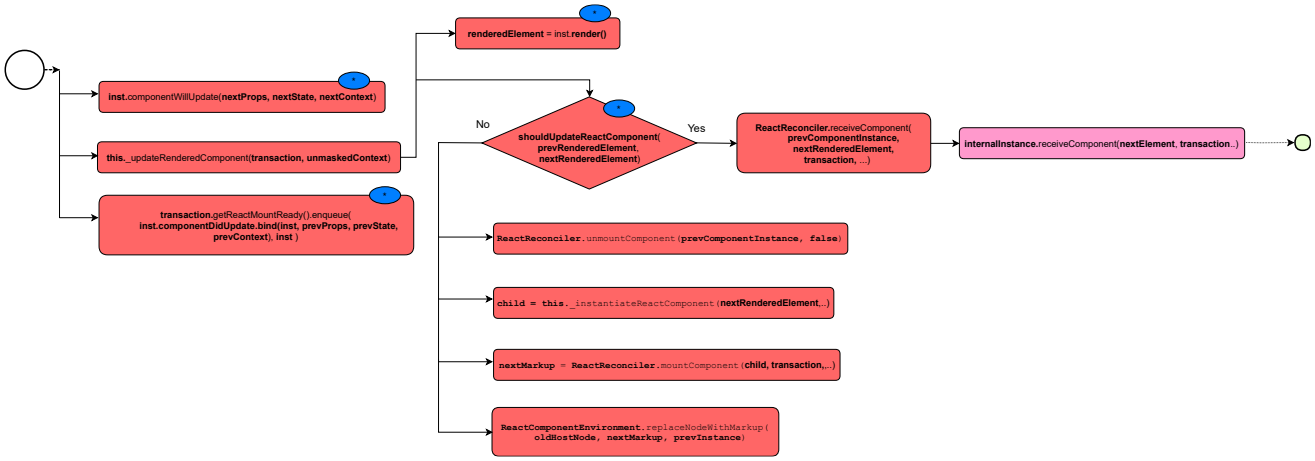
好, 第 12 部分我们讲完了

我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：



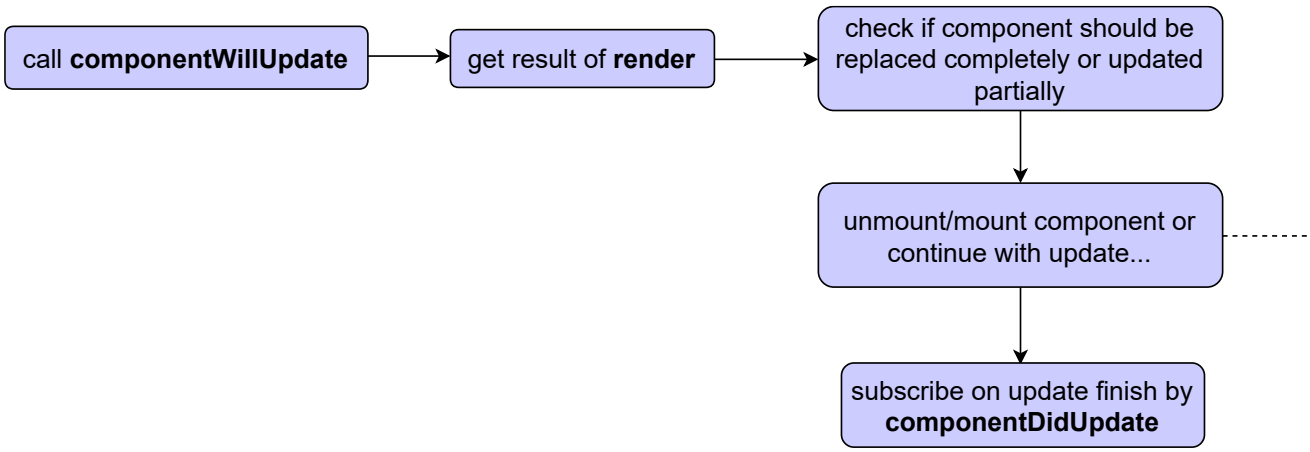
第 12 部分简化版 (点击查看大图)

然后我们适当再调整一下：



12.2 第 12 部分简化和重构 (点击查看大图)

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 12 部分**的本质，并将其用于最终的 `updating` 方案：



12.3 第 12 部分本质 (点击查看大图)

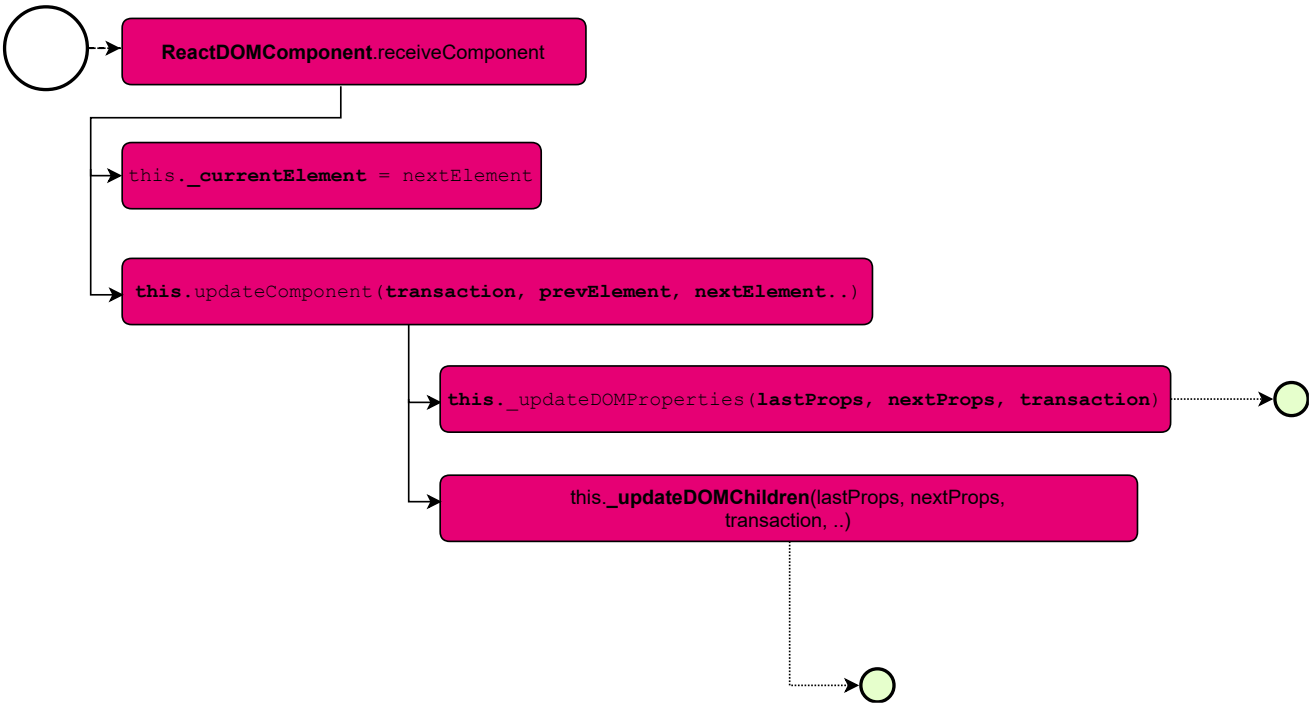
完成!

下一节: 第 13 部分>>

<< 上一节: 第 11 部分

主页

第 13 部分



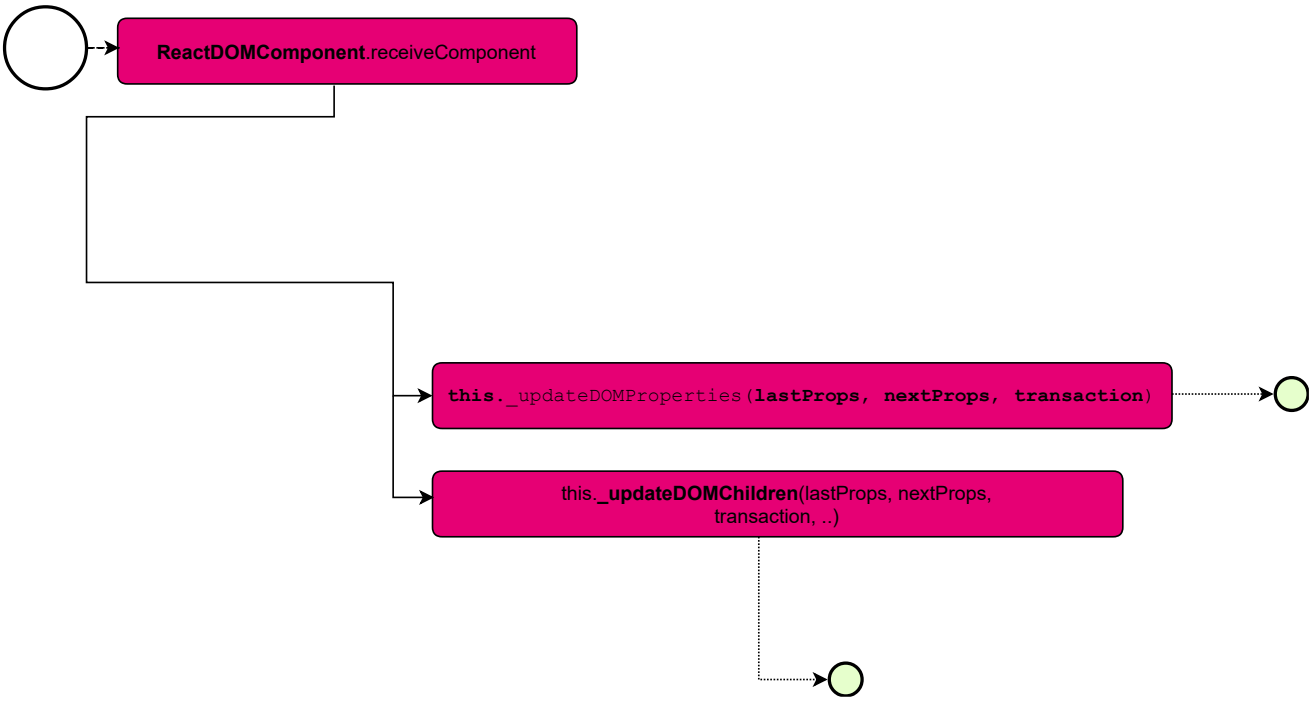
13.0 第 13 部分（点击查看大图）

接收组件（更精确的下一个元素）

通过 `ReactReconciler.receiveComponent`，React 实际上从 `ReactDOMComponent` 调用 `receiveComponent` 并传递给下一个元素。在 DOM 组件实例上重新分配并调用 update 方法。`updateComponent` 方法实际上主要是两步：基于 `prev` 和 `next` 的属性，更新 DOM 属性和 DOM 元素的子节点。好在我们已经分析了 `_updateDOMProperties`（`src\renderers\dom\shared\ReactDOMComponent.js#946`）方法。就像你记得的那样，这个方法大部分处理了 HTML 元素的属性和特质，计算样式以及处理事件监听等。剩下的就是 `_updateDOMChildren`（`src\renderers\dom\shared\ReactDOMComponent.js#1076`）方法了。

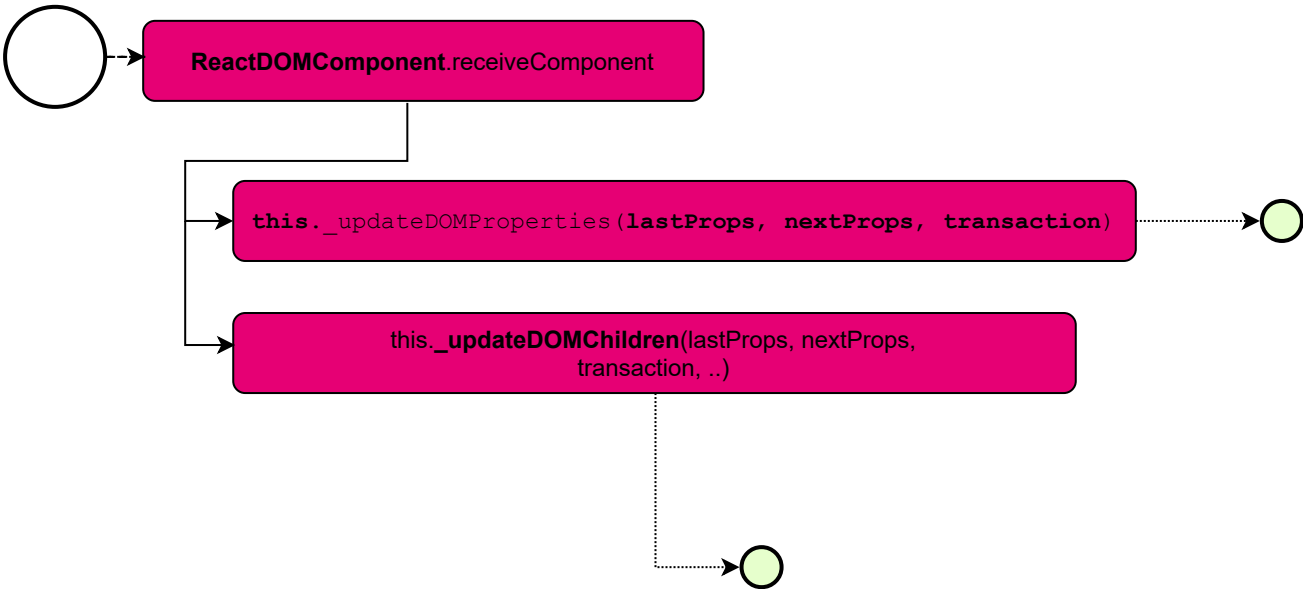
好了，我们已经完成了*第 13 部分*。好短的一章。

让我们概括一下我们怎么到这里的。再看一下这张图，然后移除掉冗余的不那么重要的部分，它就变成了这样：



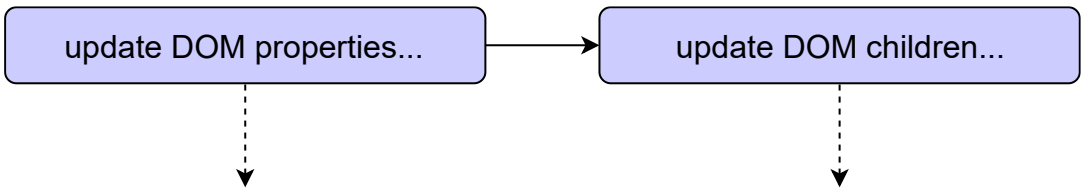
13.1 第 13 部分 简化（点击查看大图）

我们也应该尽可能的修改空格和对齐方式:



13.2 第 13 部分 简化和重构（点击查看大图）

很好。实际上它就是这儿所发生的一切。我们可以从*第 13 部分*中获得基本价值，并将其用于最终的"更新"图表:



13.3 第 13 部分本质（点击查看大图）

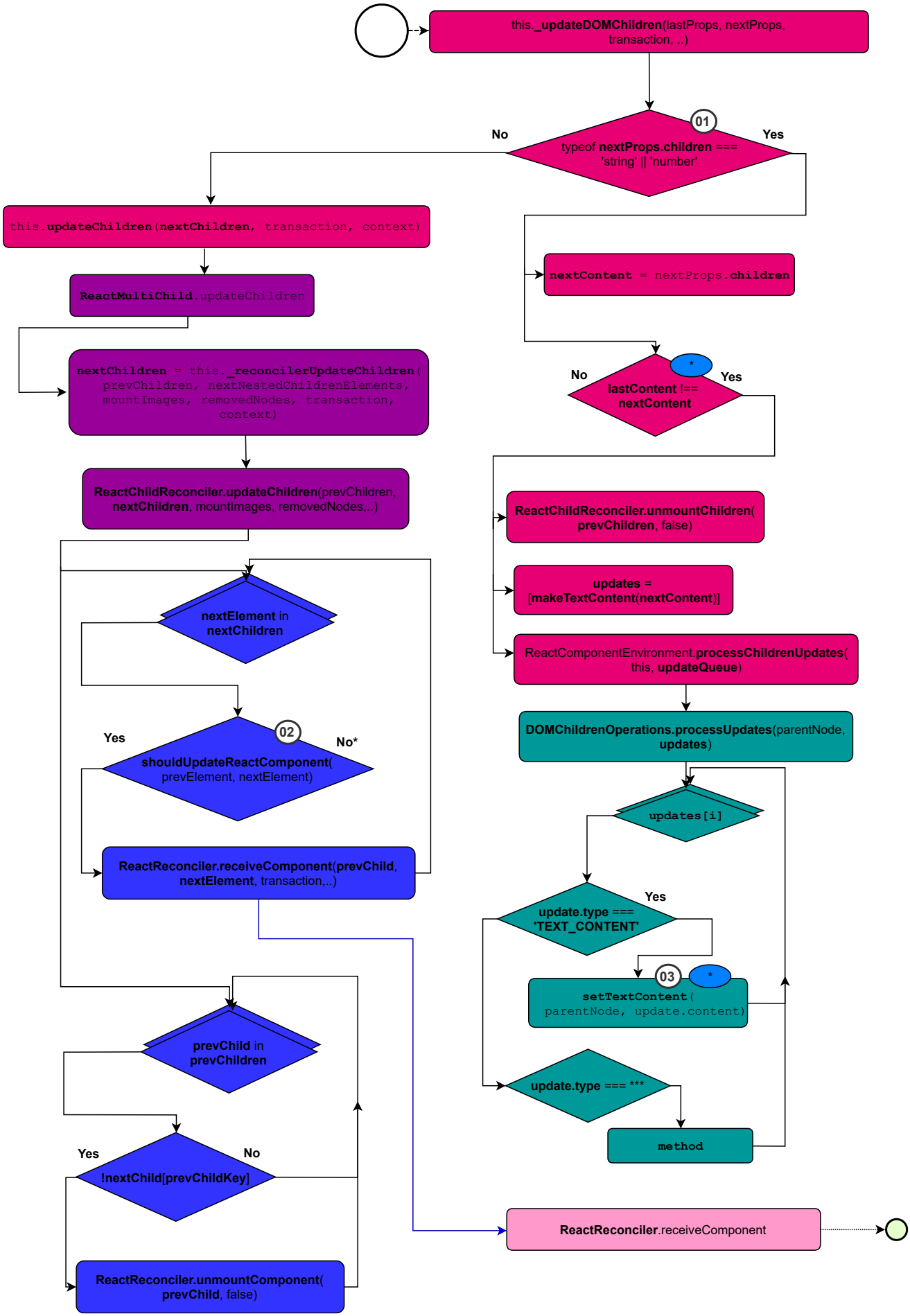
完成!

下一节：第 14 部分 >>

<< 上一节：第 12 部分

主页

第 14 部分


this._updateDOMChildren(lastProps, nextProps, transaction, ..)
01
typeof nextProps.children === 'string' || 'number'
No
Yes
this.updateChildren(nextChildren, transaction, context)
ReactMultiChild.updateChildren
nextChildren = this._reconcilerUpdateChildren(prevChildren, nextNestedChildrenElements, mountImages, removedNodes, transaction, context)
ReactChildReconciler.updateChildren(prevChildren, nextChildren, mountImages, removedNodes,..)
nextElement in nextChildren
02
Yes
No*
shouldUpdateReactComponent(prevElement, nextElement)
ReactReconciler.receiveComponent(prevChild, nextElement, transaction,..)
prevChild in prevChildren
Yes
No
!nextChild[prevChildKey]
ReactReconciler.unmountComponent(prevChild, false)
nextContent = nextProps.children
*
No
Yes
lastContent !== nextContent
ReactChildReconciler.unmountChildren(prevChildren, false)
updates = [makeTextContent(nextContent)]
ReactComponentEnvironment.processChildrenUpdates(this, updateQueue)
DOMChildrenOperations.processUpdates(parentNode, updates)
updates[i]
Yes
update.type === 'TEXT_CONTENT'
03
*
setTextContent(parentNode, update.content)
update.type === ***
method
ReactReconciler.receiveComponent

14.0 第 14 部分(点击查看大图)

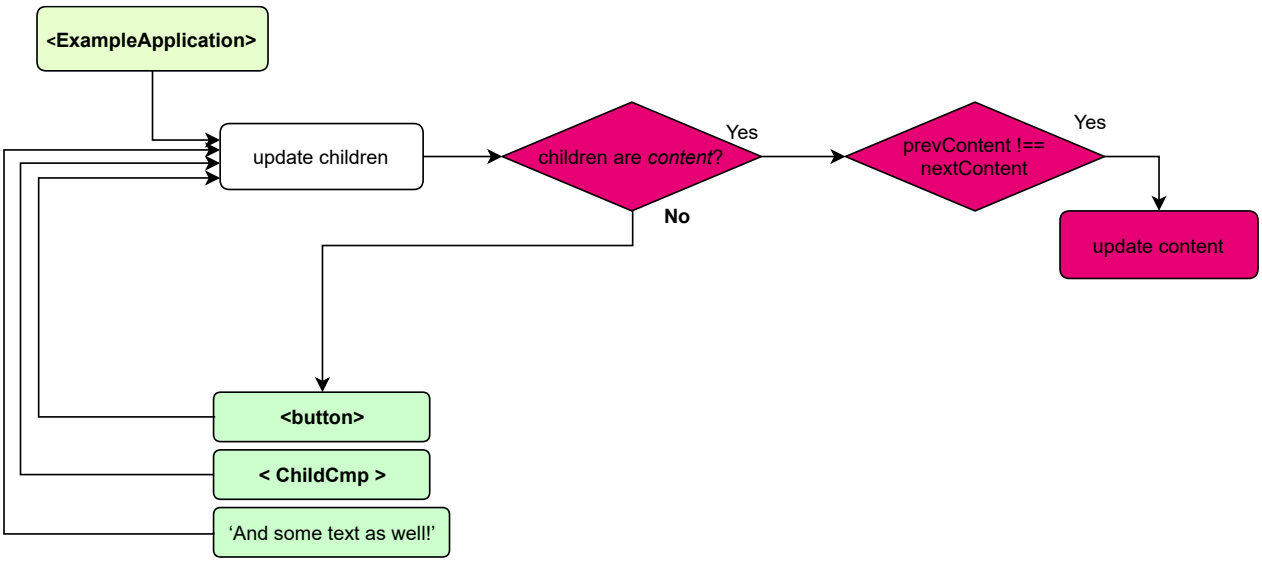
最后一章!

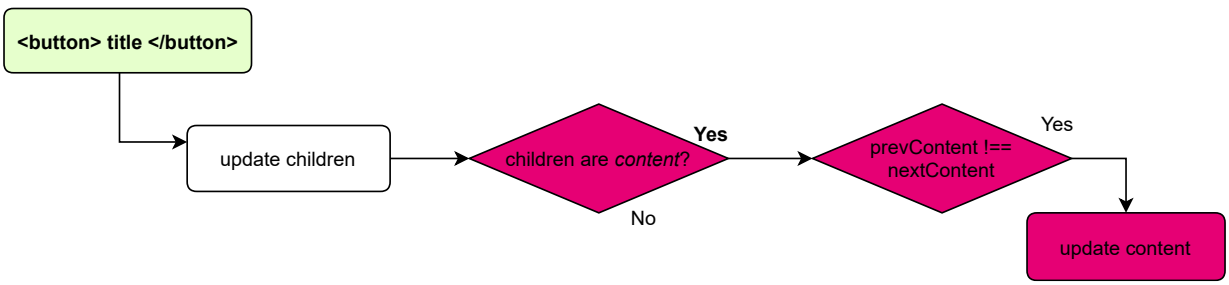
在发起子组件更新操作时会有很多属性影响子组件内容。这里有几种可能的情况，不过其实就只有两大主要情况。即子组件是不是 “复杂”。这里的复杂的含义是，它们是React组件，React应当通过它们不断递归直到触及内容层，或者，该子组件只是简单数据类型，比如字符串、数字。

这个判断条件就是步骤 (1) 的 `nextProps.children` 的类型，在我们的情形中，`ExampleApplication` 有三个孩子 `button`, `ChildCmp` 和 `text string`。

很好，现在让我们来看它的工作原理。

首先，在首次迭代时，我们分析 `ExampleApplication children`。很明显可以看出子组件的类型不是 “纯内容类型”，因此情况为 “复杂” 情况。然后我们一层层往下递归，每层都会判断 children 的类型。顺便说一下，步骤 (2) 的 `shouldUpdateReactComponent` 判断条件可能让你有些困惑，它看上去是在验证更新与否，但实际上它会检查类型是更新还是删除与创建（为了简化流程我们跳过此条件为否的情形，假定是更新）。当然接下来我们对比新旧子组件，如果有孩子被移除，我们也会去除挂载组件，并把它移除。





14.1 Children 更新 (点击查看大图)

在第二轮迭代时，我们分析 `button`，这是一个很简单的案例，由于它仅包含一个标题文字 `set state button`，它的孩子只是一个字符串。因此我们对比一下之前和现在的内容。很好，这些文字并没有变化，因此我们不需要更新 `button`？这非常的合理，因此所谓的 “虚拟 DOM”，现在听上去也不是那么的抽象，React 维护了一个对 DOM 的内部表达对象，并且在需要的时候更改真实 DOM，这样取得了很不错的性能。因此我想你应该已经了解了这个设计模式。那我们接着来更新 `ChildCmp`，然后它的孩子也到达我们可以更新的最底层。可以看到在这层的内容已经被修改了，当时我们通过 `click` 和 `setState` 的调用，`this.props.message` 已经更新成 `'click state message'` 了。

```
//...
onClickHandler() {
    this.setState({ message: 'click state message' });
}

render() {
    return <div>
        <button onClick={this.onClickHandler.bind(this)}>set state button</button>
        <ChildCmp childMessage={this.state.message} />
//...
```

从这里可以看出已经可以更新元素的内容，事实上也就是替换它。那么真正的行为是怎样的呢，其实它会生成一个"配置对象"并且其配置的动作会被相应地应用。在我们的场景下这个文字的更新操作可能形如：

```
{
  afterNode: null,
  content: "click state message",
  fromIndex: null,
  fromNode: null,
  toIndex: null,
  type: "TEXT_CONTENT"
}
```

我们可以看到很多字段是空，因为文字更新是比较简单的。但是它有很多属性字段，因为当你移动节点就会比仅仅更新字符串要复杂得多。我们来看这部分的源码加深理解。

```
//src\renderers\dom\client\utils\DOMChildrenOperations.js#172
processUpdates: function(parentNode, updates) {
    for (var k = 0; k < updates.length; k++) {
      var update = updates[k];

      switch (update.type) {
        case 'INSERT_MARKUP':
          insertLazyTreeChildAt(
            parentNode,
            update.content,
            getNodeAfter(parentNode, update.afterNode)
          );
          break;
        case 'MOVE_EXISTING':
          moveChild(
            parentNode,
            update.fromNode,
            getNodeAfter(parentNode, update.afterNode)
          );
          break;
        case 'SET_MARKUP':
          setInnerHTML(
            parentNode,
            update.content
```

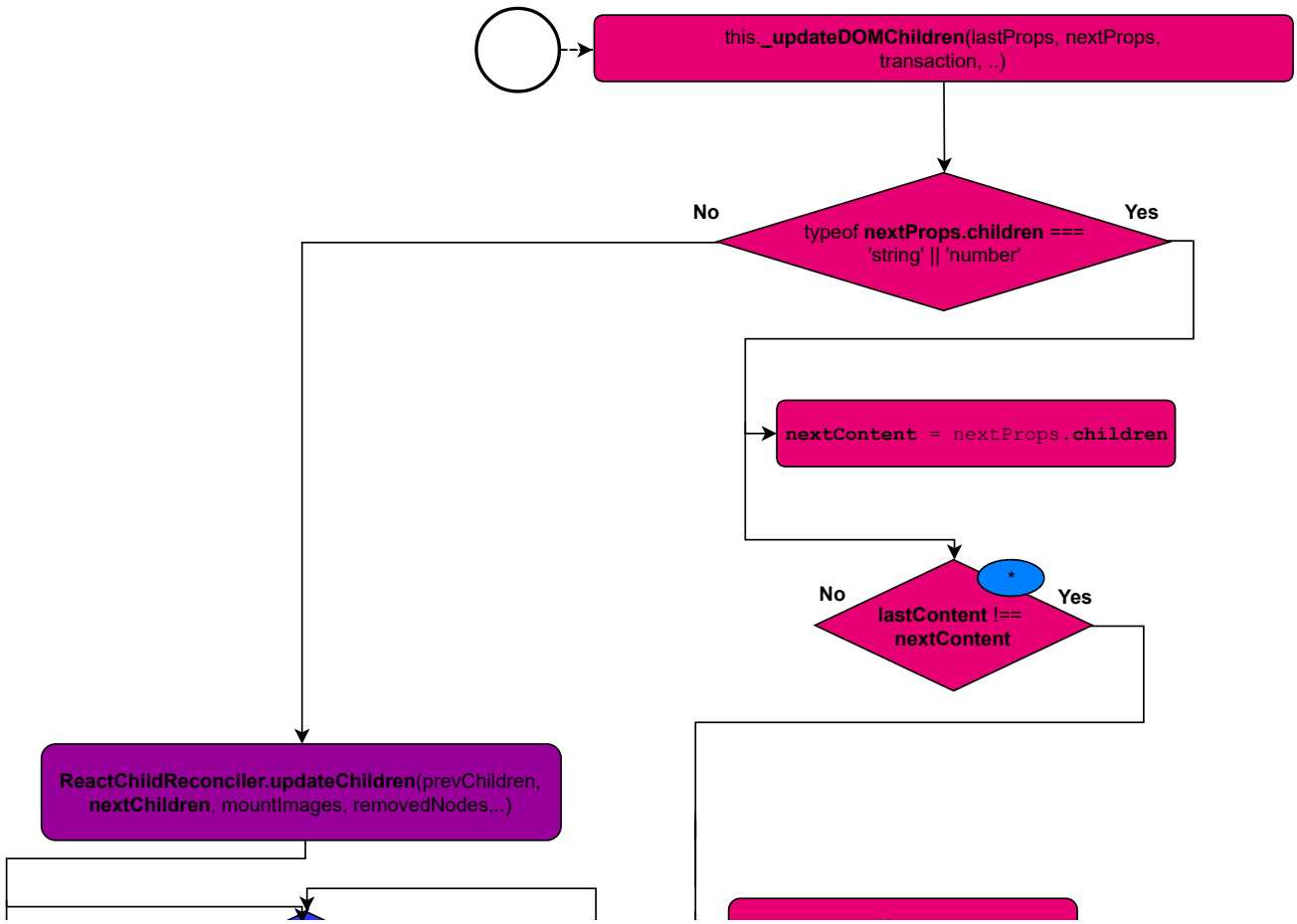
```
          );
          break;
        case 'TEXT_CONTENT':
          setTextContent(
            parentNode,
            update.content
          );
          break;
        case 'REMOVE_NODE':
          removeChild(parentNode, update.fromNode);
          break;
      }
    }
  }
```

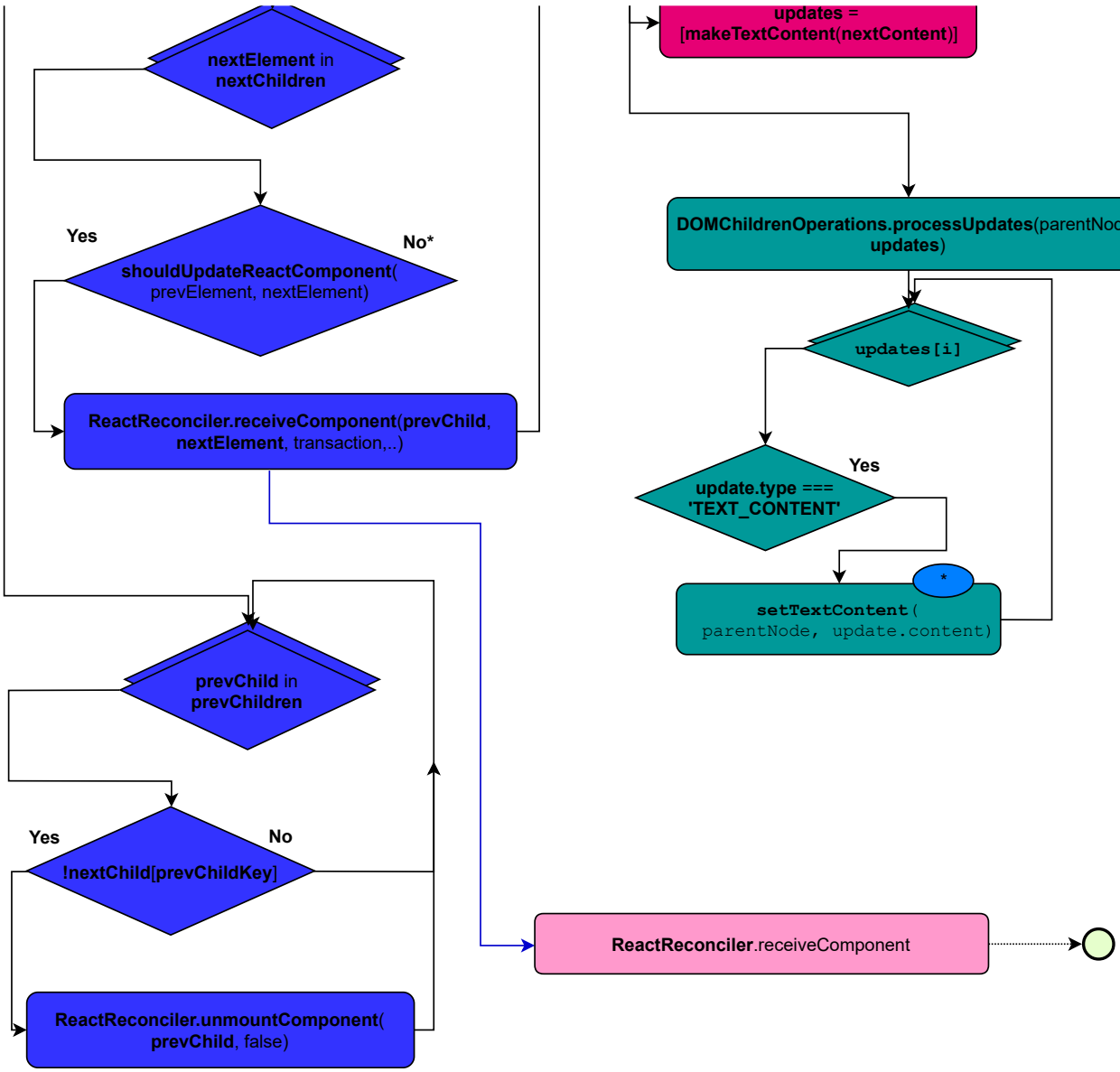
在我们的情况下，更新类型是 `TEXT_CONTENT`，因此实际上这是最后一步，我们调用步骤 (3) 的 `setTextContent` 方法并且更新 HTML 节点（从真实 DOM 中操作）。

非常好！内容已经被更新，界面上也做了重绘。我们还有什么遗忘的吗？让我们结束更新！这些事都做完了，我们的组件生命周期钩子函数 `componentDidUpdate` 会被调用。这样的延迟回调是怎么调用的呢？实际上就是通过事务的封装器。如果你还记得，脏组件的更新会被 `ReactUpdatesFlushTransaction` 封装器修饰，并且其中的一个封装器实际上包含了 `this.callbackQueue.notifyAll()` 逻辑，所以它回调用 `componentDidUpdate`。很好，现在看上去我们已经讲完了全部内容。

好, 第 14 部分我们讲完了

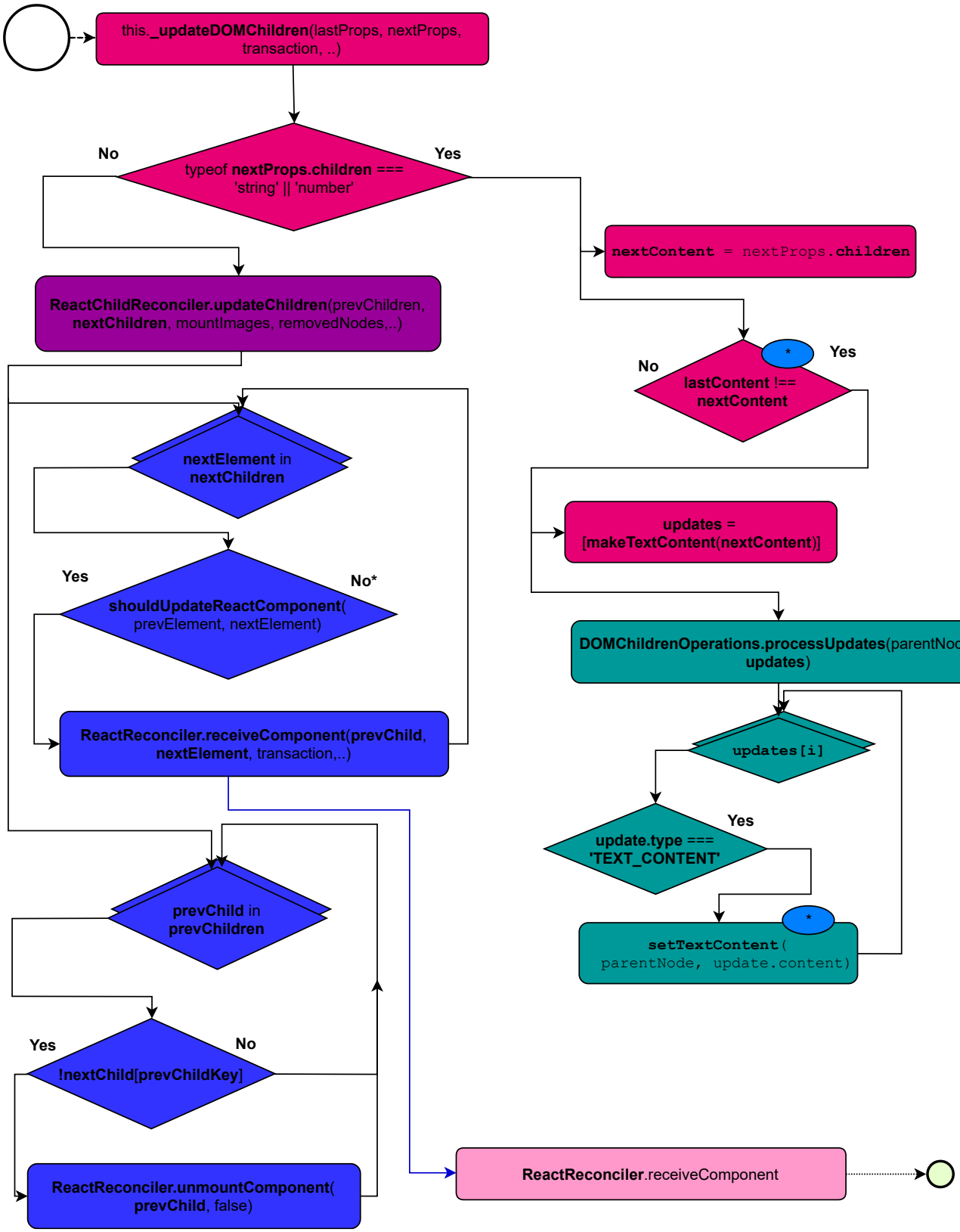
我们来回顾一下我们学到的。我们再看一下这种模式，然后去掉冗余的部分：





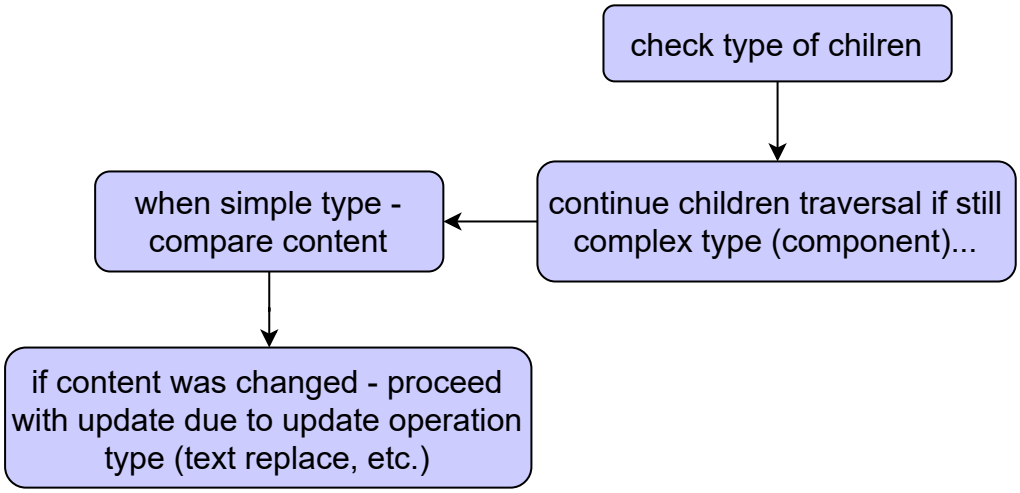
14.2 第 14 部分简化板 (点击查看大图)

然后我们适当再调整一下：



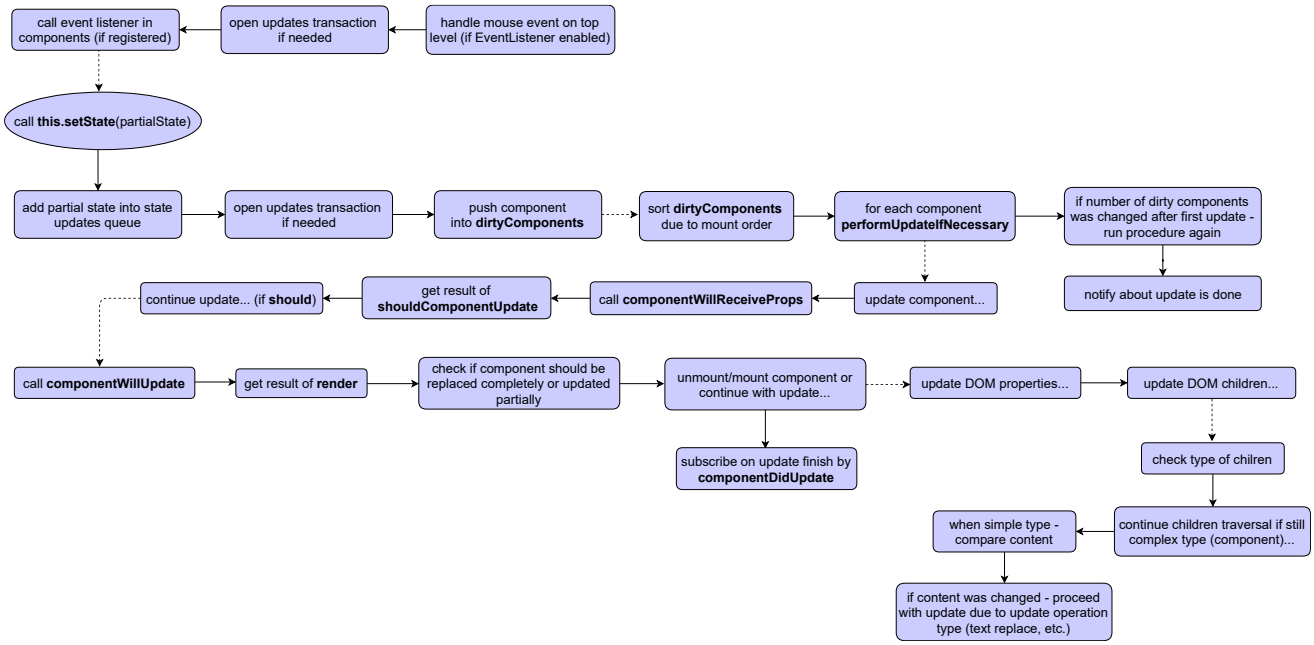
14.3 第 14 简化和重构（点击查看大图）

很好，实际上，下面的示意图就是我们所讲的。因此，我们可以理解**第 14 部分**的本质，并将其用于最终的 `updating` 方案：



14.4 第 14 部分 本质 (点击查看大图)

我们已经完成了更新操作的学习，让我们重头整理一下。



14.5 更新 (点击查看大图)

[<< 上一节: 第 13 部分]

[主页]