1. 协调不会长时间阻塞浏览器渲染：会给GUI渲染引擎留出足够的时间进行渲 染，即使是大数据量更新也可以分段更新。
2. 高优先级更新可以中断低优先级更新，优先渲染：可以把一些数据量大、紧 急度不高的更新通过低优先级api发起，优先渲染高优先级的数据内容。
3. 通过Suspense、SuspenseList、useTransition、useDeferredValue 这些 新的 API（低优先级api），给用户提供更好的交互体验。

### 在legacy模式下：

构建fiber树的协调过程是同步的，react一定要构建完整个fiber树完成整个更新任务的调度和协调过程才会去执行新的调度任务。Legacy 模式下所有的更新都分配的是相同的 lane-SyncLane，更新是按照先进先出的顺序来处理的，不存在更新中断的情形。

### concurrent模式下：

1. 任务注册

生成一个update时，根据上下文使用不同的**lane优先级**，update存放于fiber.updateQueue上，将这些lane收集起来放于**root.pendingLanes和fiber.lanes**，然后在ensureRootIsScheduled中为更新注册调度任务，调度器会根据上下文生成不同优先级的调度任务（**调度优先级**），并申请一个时间片（5ms）。

1. 任务执行

workLoop在每次执行时会判断时间片是否过期，同时也会判断下一个要处 理的task（使用调度优先级，优先级越高过期时间越短）是否过期：

如果过期，说明task要立即处理，此时就不会让出主线程；

如果没过期则让出浏览器的执行权。

高优先级更新打断低优先级更新、恢复中断协调：

低优先级任务执行后协调中 => 发生高优先级更新 =>

令时间片到期，让出主线程 =>为高优先级更新创建调度任务，获取新的时间片=>workLoop从任务栈中取任务执行，拿到时间片后会先判断此次协调的优先级（**renderLanes**）与上次的协调优先级是否一致，如果一致则**恢复中断协调；**如果不一致则说明有高优先级的更新=>处理高优先级的更新=>从根节点重新处理低优先级的更新。

workLoop的协调过程：

从**root.pendingLanes**中取优先级最高的**lanes**作为**renderLanes**， 判断**fiber.lanes**中是否存在**renderLanes，**如果存在则再通过 **renderLanes**去匹配**fiber.updateQueue**中的**update**。

匹配**fiber.updateQueue**过程（遍历）：

updateQueue中不匹配**renderLanes** 的update会被跳过。

例如，updateQueue中 update 为 A1 => B2 => C1 => D2， renderLanes 为 1， 那么第1、3 个update会被处理，第2、4个 update会被跳过，先输出AC。

其次，有些update会依赖上一个update对象的返回值，为了保证 状态的连续性，就需要记录上一次协调时第一次出现lane不配 renderLanes时update的位置以及当时的state。

在上面的例子中，第一次协调时，在第二个update这里第一次出现 lane不匹配renderLanes。此时，state为 A，不匹配的update及 后续update为B2=>C1=>D2。第二次协调时，renderLanes为2， state为A，那么最后输出为ABD。

期望的结果是ABCD，现在却是ABD，是哪里出现问题了？

之所以会这样，是因为中间有一个步骤被遗漏了。第一次协调时，A1 和 C1被处理，他们的 lane 会被置为 0，那么第二次协调时处理的是 B2=>C0=>D2。0 是可以匹配任何 renderLanes 的，所以在第二 次 协调时，C 会再被处理一遍，最后的结果也就变成了如我们所愿 的 ABCD。

### Suspense 以及 useTransition&startTransition：

1. Suspense：

在旧有版本中，Suspense仅是作为与React.lazy配合做组件懒加载 而使用的，现在它有了新的功能：

Suspense包裹异步组件，提供一个fallBack用作loading：

提供一个函数对promise若promise异步数据源进行分装，处于 pengding阶段 则抛出这个promise，rejected抛出错误，fulfilled返 回数据。

Suspense实现原理：

其实比较简单，利用react的错误边界（Error Boundaries）提供 的生命周期钩子函数componentDidCatch，捕捉异步子组件中的 异步请求抛出的对象（pending时为promise，rejected为错误）， 此时直接显示loading，然后进行回调检查这个对象是否是 promise， 如果是则执行promise的then方法，再次尝试显示异 步子组件，此时异步子组件会再次加载、调取异步数据，重复上面 的流程。



1. useTransition&startTransition：

利用 useTransition & startTransition来运行低优先级任务（过渡任 务），触发 concurrent更新，实现时间片分段的协调，适用于跟某些 高优先级任务（与用户体验高度关联）同时触发时，使用此方式来执 行低优先级任务。

1. useDeferredValue

useTransition是将一个任务（update）以低优先级运行（过渡任务）， 而useDeferredValue是将一个值通过过渡任务得到新的值。

1. componentWillReceiveProps、componentWillMount、 componentWillUpdate

在concurrent模式下不再安全且不能getDerivedStateFromProps 以及getSnapshotBeforeUpdate这样的新生命周期api共用。

不安全的情况：当前组件的协调源自于一个低优先级的更新，且使用了 componentWillXXX这样的不全的生命周期，此时会触发该生命周期； 如果此时发生了其他的高优先级的更新，会导致之前的低优先级的协 调被打断，再次恢复该协调时（从根节点重新协调），会再次触发 componentWillXXX生命周期。

componentWillXXX生命周期getDerivedStateFromProps 以及 getSnapshotBeforeUpdate的调用阶段：

beginWork=>updateClassComponent=>updateClassInstance=> callComponentWillReceiveProps。

1. update 重复处理的副作用

如果使用改正前的方式，state.count为3，因为如匹配fiber.updateQueue **过程（遍历）**所述，跳过低优先级更新（startTransition(() => dispatch("INCREMENT"))）时，会记录此时的state.count（跳过时为0）， 图中错误写法在于reducer计算时修改了state（react中都是浅克隆）， dispatch("INCREMENT") 会导致之前记录的state.count变为1，而等到 低优先级update协调时会把而低优先级更新协调时会把跳过 update及其之后**所有**的update以以记录的state.count（1）为基础重新 计算一 遍，就会得出3的结果。

如果使用改正后的写法，dispatch("INCREMENT")就不会影响记录的state， 低优先级update协调后得到的结果就是2。