[Draft][Tech]RN 性能优化技术方案

渲染优化

一、正确理解 useMemo、useCallback、memo 的使用场景

在我们平时的开发中很多情况下我们都在滥用 useMemo、useCallback这两个 hook， 实际上很多情况下我们不需要甚至说是不应该使用，因为这两个 hook 在首次 render 时需要做一些额外工作来提供缓存，

同时既然要提供缓存那必然需要额外的内存来进行缓存，综合来看这两个 hook 其实并不利于页面的首次渲染甚至会拖慢首次渲染，这也是我们常说的“不要在一开始就优化你的组件，出现问题的时候再优化也不迟”的根本原因。

那什么时候应该使用呢，无非以下两种情况：

1. 缓存 useEffect 的引用类型依赖；
2. 缓存子组件的引用类型依赖。

1． 缓存 useEffect 的引用类型依赖

```
import { useEffect } from 'react'
export default () => {
  const msg = {
    info: 'hello world',
  }
  useEffect(() => {
    console.log('msg:', msg.info)
  }, [msg])
}
```

此时 msg 是一个对象该对象作为了 useEffect 的依赖，这里本意是 msg 变化的时候打印 msg 的信息。但是实际上每次组件在render 的时候 msg 都会被重新创建，msg 的引用每次render 时都是不一样的，所以这里 useEffect 在每次render 的时候都会

重新执行，和我们预期的不一样，此时 useMemo 就可以派上用场了：

```
import { useEffect, useMemo } from 'react'
export default () => {
  const msg = useMemo(() => {
    return {
      info: 'hello world',
    }
  }, [])
  const print = msg => {
    console.log(msg)
  }
  useEffect(() => {
    console.log('msg:', msg.info)
  }, [msg])
}
```

同理对于函数作为依赖的情况，我们可以使用 useCallback：

```
import { useCallback, useEffect } from 'react'
export default props => {
  const print = useCallback(() => {
    console.log('msg', props.msg)
  }, [props.msg])
  useEffect(() => {
    print()
  }, [print])
}
```

2．缓存子组件的引用类型依赖

做这一步的目的是为了防止组件非必要的重新渲染造成的性能消耗，所以首先要明确组件在什么情况下会重新渲染。

1. 组件的 props 或 state 变化会导致组件重新渲染
2. 父组件的重新渲染会导致其子组件的重新渲染

这一步优化的目的是：在父组件中跟子组件没有关系的状态变更导致的重新渲染可以不渲染子组件，造成不必要的浪费。

大部分时候我们是明确知道这个目的的，但是很多时候却并没有达到目的，存在一定的误区：

误区	说明

`import { useCallback, useState } from 'react'`
`import { Text } from 'react-native'`
 	
`const Child = props => {}`
`export default () => {`
`  const handleChange = useCallback(() => {}, [])`
`  const [count, setCount] = useState(0)`
`  return (`
`    <>`
`      <Text onPress={() => {`
`        setCount(count++)`
`      }}>`
`        increase`
`      </Text>`
`      <Child cb={handleChange} />`
`    </>`
`  )`
`}`
	项目中有很多地方存在这样的代码，实际上完全不起作用，因为只要父组件重新渲染，Child 组件也会跟着重新渲染。

`import { memo, useState } from 'react'`
`import { Text } from 'react-native'`
 	
`const Child = memo (props => {})`
`export default () => {`
`  const handleChange = () => {}`
`  const [count, setCount] = useState (0)`
`  return (`
`  <>`
`  <Text`
`  onPress={() => {`
`  setCount (count++)`
`  }}`
`  >`
`  increase`
`  </Text>`
`  <Child cb= {handleChange} />`
`  </>`
`  )`
`}`
	给 Child 加上 memo 并不能完事大吉，如果 Child 的 props 属性中存在引用类型，这些引用类型却没有加上缓存，那父组件每次重新渲染的时候都会导致这些引用类型的数据重新创建 memo 在进行浅比较的时候发现引用不一样了也会重新渲染 Child 组件

正确的使用方式：

```
import { memo,
useCallback,
useMemo, useState
} from 'react'
import { Text }
from 'react-native'

const Child = memo
(props => {})
export default ()
=> {
  const
handleChange =
useCallback(() =>
{}, [])
  const list =
useMemo(() => {
    return []
  }, [])
  const [count,
setCount] =
useState(0)
  return (
    <>
      <Text
        onPress=
{() => {
          setCount
(count++)
        }}
      >
        increase
      </Text>
      <Child cb=
{handleChange}
list={list} />
    </>
  )
}
```

为了防止子组件因为父组件的渲染导致重复渲染，首先我们应该使用 memo 函数对子组件进行包装，包装完毕后子组件在每次重新渲染前会对 props 里面的属性进行浅比较，如果存在引用类型则比较引用类型的引用，所以一定要对引用类型使用 useCallback 或 useMemo 进行包装，否则除了让代码更复杂以外起不了任何作用。

二、useContext 使用注意事项

现在项目中我们已经重度依赖于 useContext 这个 api 来做状态管理，这也引入了一些问题。

useContext 的机制是使用这个 hook 的组件在 context 发生变化时都会重新渲染。这样导致的问题是：

1. memo 优化直接被穿透，不在起作用
2. 其他不依赖于这个 context 的组件也会被同时重新渲染
3. 如果把太多相关性不大的状态放在一起那也会导致

具体可以看这里的讨论 https://github.com/facebook/react/issues/15156