React Native的傻瓜式优化

React Native中文网 晴明
https://reactnative.cn

与算法、架构、业务无关的性能优化

- Pure Render
- Replacement
- Unlocking

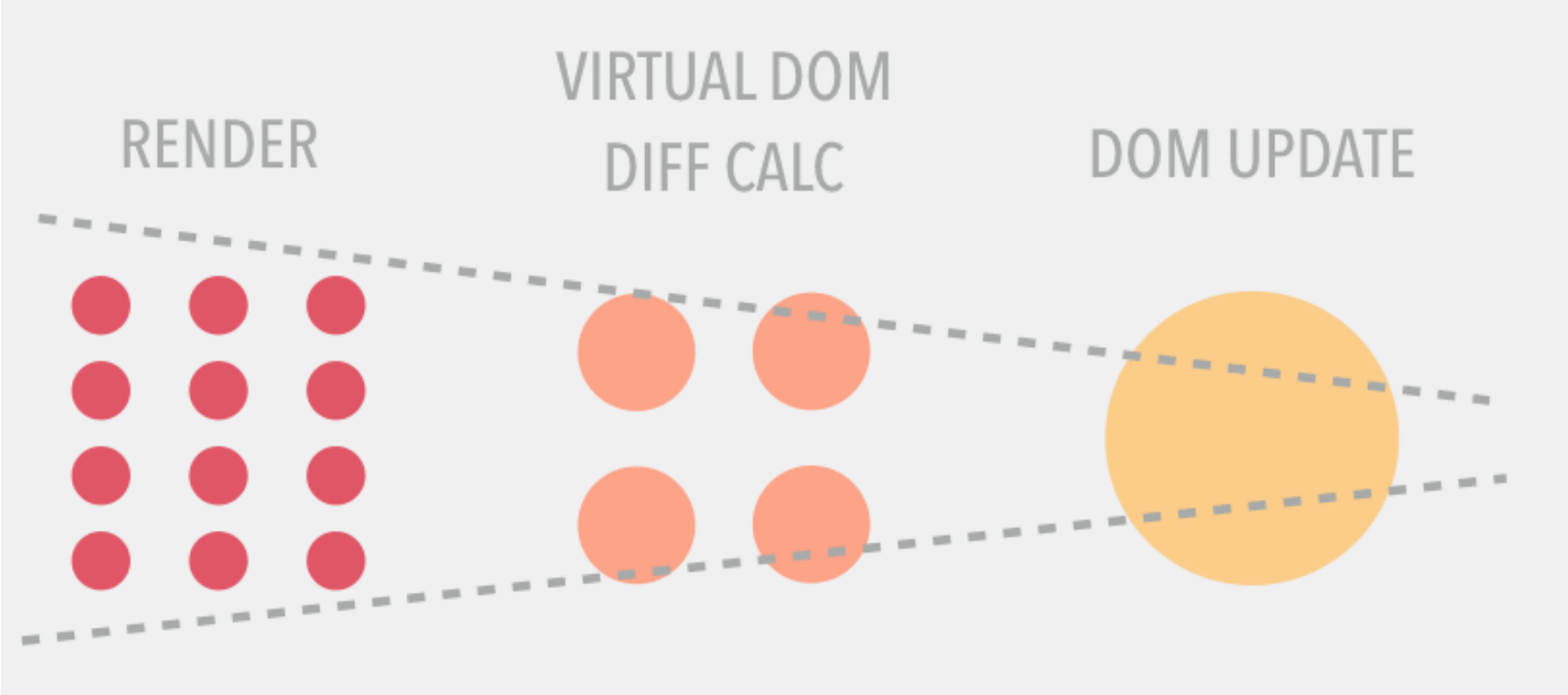
Pure Render

万物生长靠太阳，界面变化靠状态state

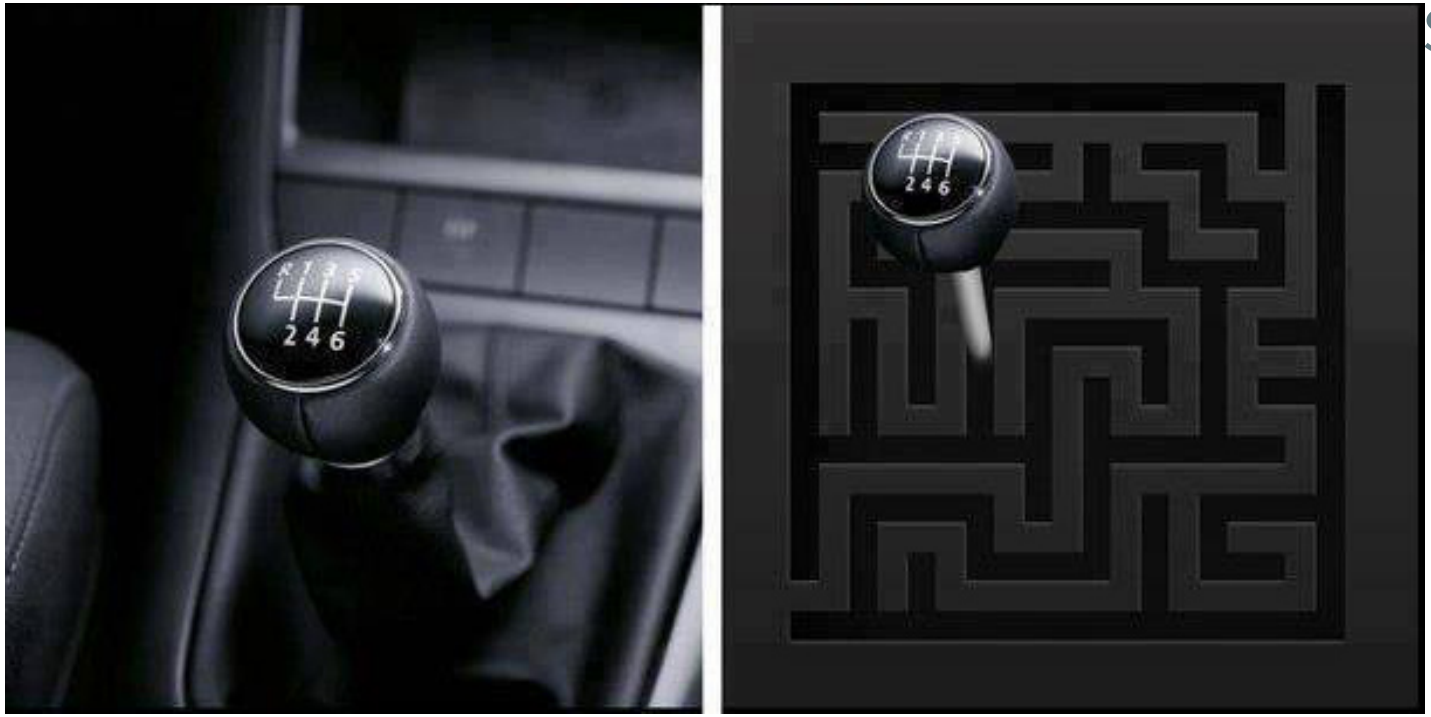
- 一切界面变化都是**状态state变化**
- React会自动计算状态的差异部分，以**最小的差异**去重新渲染
- 在内存中“打草稿计算差异”的这一概念便称为VIRTUAL DOM

然而，“打草稿”也是开销

所以，能不打就不打


RENDER
VIRTUAL DOM
DIFF CALC
DOM UPDATE

Avoid Reconciliation



State)

增强型“自动档” PureComponent

从react 15.3(react native 0.32)版本开始
引入了新的PureComponent（代替老的PureRenderMixin）

```
shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, ne
    // 对props和state进行浅比较
}
```

如果你的props和state都是值/原始类型（ primitive data type ）
即这6种： string, number, boolean, null, undefined, symbol
那么直接把Component替换为PureComponent即可

引用类型和immutable原则

引用类型即Object，而Object又包含有array、function等。

引用类型的浅比较实际为比较其指针地址。
修改引用类型的值，其指针地址不会变化。

如果使用PureComponent，而props/state又包含引用类型，
则需要遵循immutable原则——不修改原值。

immutable的典型用例

复制原值后再修改

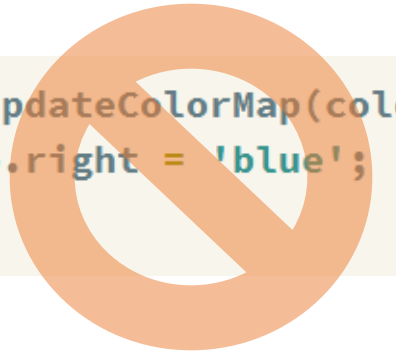
```
handleClick() {
  // This section is bad style and causes a bug
  const words = this.state.words;
  words.push('marklar');
  this.setState({words: words});
}
```

```
handleClick() {
  this.setState(prevState => ({
    words: prevState.words.concat(['marklar'])
  }));
}
```

```
handleClick() {
  this.setState(prevState => ({
    words: [...prevState.words, 'marklar'],
  }));
};
```

immutable的典型用例

复制原值后再修改

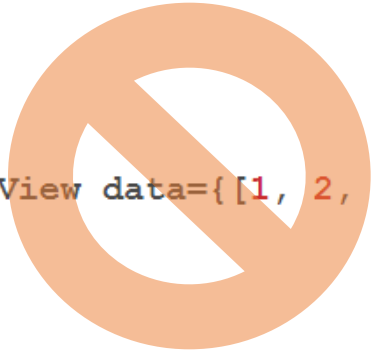


```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

immutable的典型用例

注意内联属性、函数



```
render() {
  return (
    <CustomView data={[1, 2, 3]} />
  )
}
```

```
render() {
  return (
    <CustomView data={this.data} />
  )
}
```

```
render() {
  return (
    <Button onPress={() => alert('welcome')} />
  )
}
```

```
onPress = () => {
  alert('welcome')
}
render() {
  return (
    <Button onPress={this.onPress} />
  )
}
```

Benchmark

```
class Common extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>
          C
          {this.props.data.text}
        </Text>
      </View>
    );
  }
}
```

```
class Pure extends PureComponent {
  render() {
    return (
      <View style={styles.container}>
        <Text>
          P
          {this.props.data.text}
        </Text>
      </View>
    );
  }
}
```

```
const isPureRender = false;  // 切换两种组件的开关
const CustomView = isPureRender ? Pure : Common;
const iterate = 100;  // 遍历次数
class Root extends Component {
  state = {
    // 生成长为100的数组，填充简单的对象，作为数据源
    dataSource: new Array(iterate).fill({ text: 1 })
  };
  start = iterate;  // 设定遍历的起始值，从100递减到0结束
  render() {
    const { dataSource } = this.state;
    // 生成100个子组件
    return (
      <View style={styles.root}>
        {
          dataSource.map((item, index) => <CustomView key={index} data={item} />)
        }
      </View>
    );
  }
  //...
```


```
componentDidMount() {
  this.startTime = new Date(); // 初次渲染完成后开始计时
  this.nextUpdate(); // 开始100次对state的修改
}
nextUpdate = () => {
  if(this.start > 0) {
    this.start -= 1;
    this.setState(({ dataSource }) => { // 取出state中的dataSource
      if (isPureRender) {
        // Pure组件遵循immutable原则，给予新的对象，以便浅比较能返回false
        dataSource[this.start] = { text: 2 };
      } else {
        // Common组件可以直接修改原值
        dataSource[this.start].text = 2;
      }
      return { dataSource }; // 返回修改后的state值
    });
  } else {
    alert(new Date() - this.startTime);  // 100次修改结束后查看消耗的时长
  }
};
componentDidUpdate() {
  this.nextUpdate();  // 父组件重渲染完成后进行下一次对state的修改
}
```

Benchmark

https://github.com/sunnylqm/PureRenderBenchmark

测试均在release版本下进行，每次都完全退出进程后重新运行，取三次结果的平均值

React Native 0.43.3 React 16.0.0-alpha.6	Component	PureComponent
iPhone6 (iOS 10.3.1)	1339ms	340ms
红米2 (Android 4.4.4)	2740ms	791ms

Replacement

优化重构的全新列表FlatList

从react native 0.43版本开始引入
用于代替老的ListView组件，
以解决长久以来被人诟病的滚动性能和内存占用问题

新特性：

- 简化API设计
  - 抛弃晦涩繁琐的dataSource，改用普通数组
- 在纯JS层面实现了Cell的复用机制
- 实现了跨平台的粘性section header
- 支持了scrollToEnd、scrollToIndex

优化重构的全新列表FlatList

征得58无线技术的蒋宏伟的同意，引用其测试结果如下：
(完整原文发表在其微信公众号上，题为“高性能RN列表的组件”)

测试机型：魅族 MX5	ListView	FlatList
1000条时内存	350M	180M
2000条时内存	/	230M
js-fps	4~6 fps	8~20 fps

启用了原生动画的react-navigation

在官方早期的性能文档中，提到了老的Navigator性能不佳的原因：

Navigator的动画是由JS线程所控制的。想象一下“从右边推入”这个场景的切换：

每一帧中，新的场景从右向左移动，从屏幕右边缘开始（不妨认为是320单位宽的的x轴偏移），最终移动到x轴偏移为0的屏幕位置。

切换过程中的每一帧，JS线程都需要发送一个新的x轴偏移量给主线程。如果JS线程卡住了，它就无法处理这项事情，因而这一帧就无法更新，动画就被卡住了。

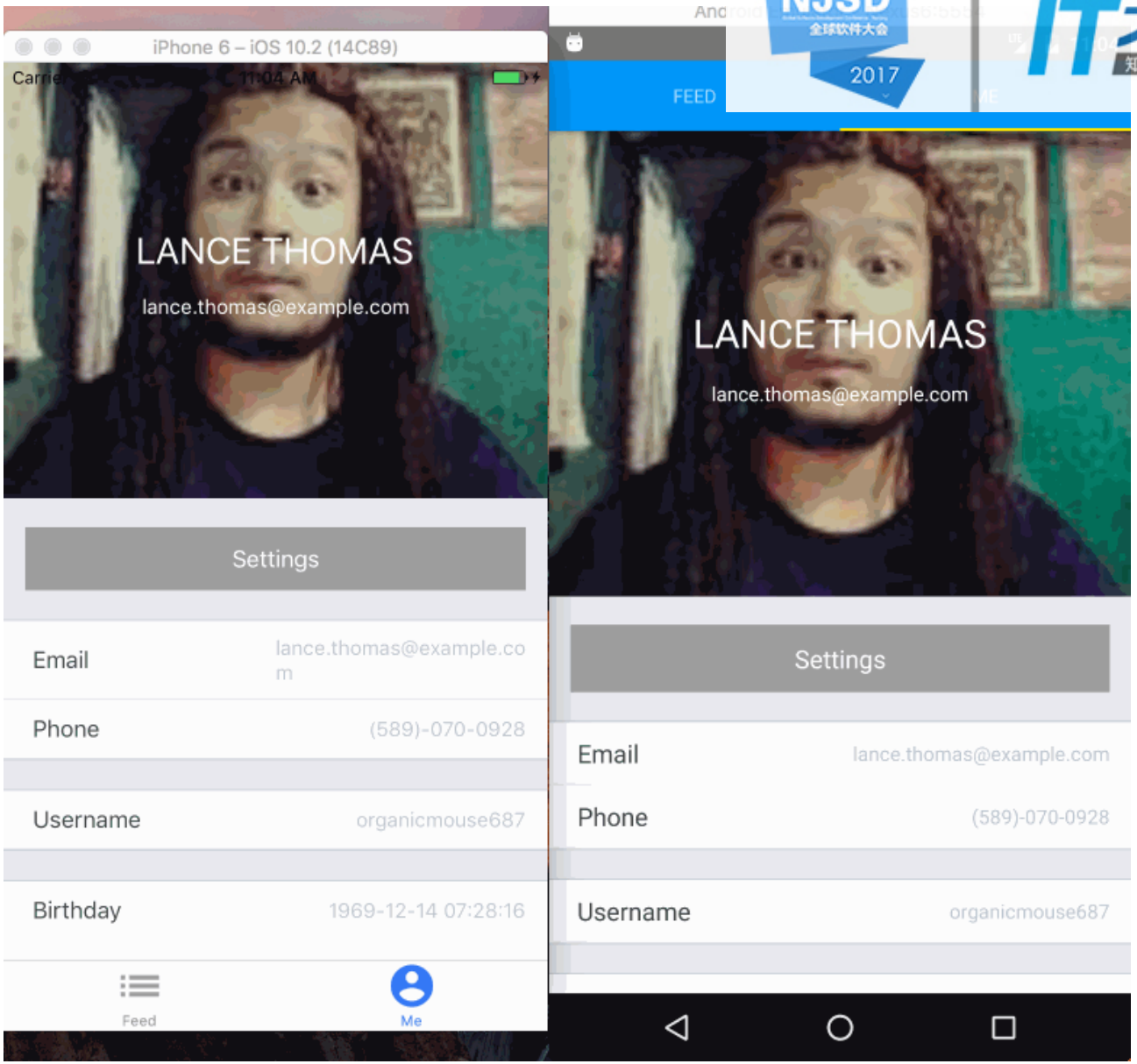
启用了原生动画的react-navigation

长远的解决方法，是要把基于JS的动画从主线程中分离。

我们可以在切换动画开始时计算出所有新场景需要的x轴偏移量，然后一次发送到主线程，以某种优化的方式执行。

此时JS线程不用再负担给主线程更新x轴偏移量的职责，因此JS线程中的掉帧就没什么影响了——切换动作不会再因此而中断等待，用户将会看到流畅的动画。

新的React Navigation库就是这一思想的实现。React Navigation中的视图是原生组件，同时用到了运行在原生线程上的Animated动画库，因而性能表现十分流畅。



Unlocking

全新的实验性Android布局实现Flatui/Nodes

计划从17年年中的版本开始启用

当前版本的解锁方式：
在MainApplication.java的ReactNativeHost中添加如下代码：

```
@Override
protected UIImplementationProvider getUIImplementationProvider() {
  return new FlatUIImplementationProvider();
}
```

全新的实验性Android布局实现Flatui/Nodes

优势：

- 在Android上支持overflow:visible样式属性（终于！）
- 可以更高效地生

已知缺陷：

- 尚不支持LayoutAnimation布局动画
- 尚不支持zIndex

尴尬的现状

由于官方在最近几个版本上对布局引擎Yoga大刀阔斧的修改，导致在最近的版本(0.42、0.43)上FlatUI处于“崩坏”状态。

咨询官方开发者得到的回复是，根据当前的工作计划，修复/完成FlatUI的时间安排在17年下半年。


简言之：如果你有overflow需求且使用0.41版本，可以一试

全新的核心算法React Fiber

预计从react 16.0正式版本开始启用

全新的Fiber旨在实现细粒度的更新任务调度：

- 可以暂停当前工作，并在合适的时机恢复
- 可以给不同任务赋予不同的优先级
- 可以复用已经完成的工作
- 可以中断不再需要的工作

理论上来说，可以提升动画等交互的体验，更新帧率可以更平滑等等。但目前我们仍非常缺乏来自官方的详细说明。

全新的核心算法React Fiber

官方开发者@janicduplessis在推上透露了目前解锁的方法：

react-native/Libraries/Renderer/src/
renderers/native/ReactNative.js
其中require('ReactNativeStack')改为require('ReactNativeFiber')

同目录下的ReactNativeFeatureFlags.js
其中useFiber: **false**改为useFiber: **true**

还是尴尬的现状

然而目前(0.43)强行开启react fiber，并没有性能上的提升。

React Native是一个日新月异的活跃的技术社区。

(两年多来日均10+ commits)

这里充满了严谨的热情、开放的智慧和不拘一格的灵感。

欢迎你加入！

https://github.com/facebook/react-native

https://reactnative.cn

THANKS!