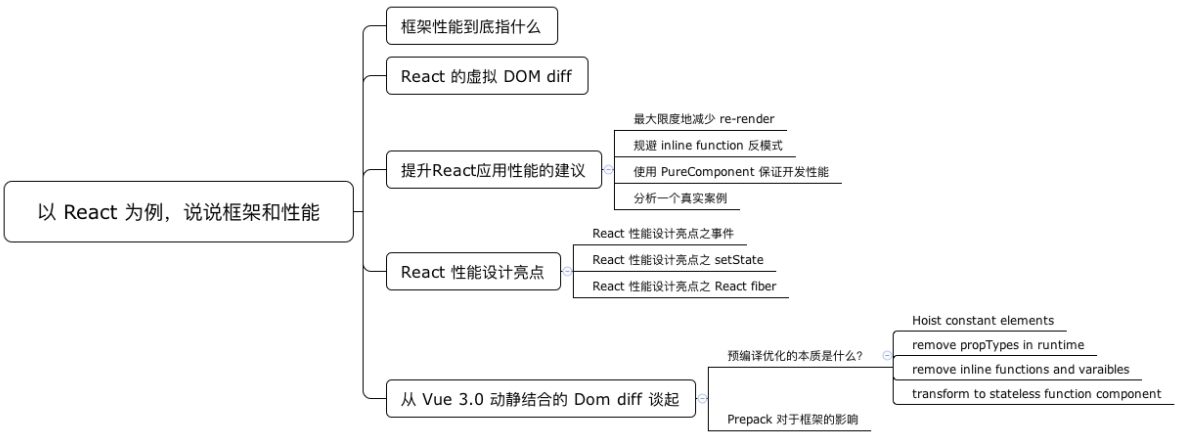
以 React 为例，说说框架和性能（2）

在上一讲中，我们提到了框架性能优化的一些基本概念，并分析了以 React 框架为代表的常用优化手段。但是这些内容还不够，需要了解更多框架设计底层的性能相关话题。这一讲，我将会以 Vue（未来新版本 3.0）和 React 为主，分析这两个框架在设计层面，而非使用层面的性能考量。

相关知识点如下图所示：



React 性能设计亮点

React 设计上的性能亮点非常多，除了「老生常谈」的虚拟 DOM 之外，还有很多不为人知的细节，比如事件机制（合成和池化）、React fiber 设计。

React 性能设计亮点之事件

React 事件机制我们前面已经有所介绍，总结一下性能亮点的体现有：

将所有事件挂载到 document 节点上，利用事件代理实现优化；

采用合成事件，在原生事件的基础上包装合成事件，并结合池化思路实现内存保护。

前面课程《第 4-2 课：你真的懂 React 吗？》已经介绍过相关内容，这里不再展开。

React 性能设计亮点之 setState

setState 这个谜之 API 我们也有所介绍，其异步（或者叫做 batch 合并）设计也是出于性能的考虑。这种优化思路已经被很多框架所借鉴，Vue 当中也是有类似的设计。

React 性能设计亮点之 React fiber

前面两个「亮点」我们在以往的课程中已经有所涉及，这里来重点说一下 React fiber。

通过课程《第 2-1 和 2-2 课：异步不可怕「死记硬背」+ 实战拿下》，我们知道在浏览器主线程中，JavaScript 代码在调用栈 call stack 执行时，可能会调用浏览器的 APIs，对 DOM 进行操作；也可能执行一些异步任务：这些异步任务如果是以回调的方式处理，那么往往会被添加到 event queue 当中；如果是以 promise 处理，就会先放到 job queue 当中。这个涉及到宏任务和微任务，这些异步任务和渲染任务将会在下一个时序当中由调用栈处理执行。

理解了这些，大家就会明白：如果调用栈 call stack 运行一个很耗时的脚本，比如解析一个图片，call stack 就会像北京上下班高峰期的环路入口一样，被这个复杂任务堵塞。主线程其他任务都要排队，进而阻塞 UI 响应。这时候用户点击、输入、页面动画等都没有了响应。

这样的性能瓶颈，就如同阿喀琉斯之踵一样，在一定程度上限制着 JavaScript 的发挥。

我们一般有两种方案突破上文提到的瓶颈，其中之一就是将耗时高、成本高、易阻塞的长任务切片，分成子任务，并异步执行。

这样一来，这些子任务会在不同的 call stack tick 周期执行，进而主线程就可以在子任务间隙当中执行 UI 更新操作。设想一个常见的场景：如果我们需要渲染

一个由十万条数据组成的列表，那么相比一次性渲染全部数据，我们可以将数据分段，使用 setTimeout API 去分步处理，构建渲染列表的工作就被分成了不同的子任务在浏览器中执行。在这些子任务间隙，浏览器得以处理 UI 更新。

React 在 JavaScript 执行层面花费的时间较多，这是因为下面一系列复杂过程所造成的：

Virtual DOM 构建 → 计算 DOM diff → 生成 render patch

也就是说，在一定程度上：React 著名的调度策略 -- stack reconcile 是 React 的性能瓶颈。因为 React stack reconcile 过程会深度优先遍历所有的 Virtual DOM 节点，进行 diff。整棵 Virtual DOM 树计算完成之后，将任务出栈释放主线程。因此，浏览器主线程被 React 更新状态任务占据的时候，用户与浏览器进行任何交互都不能得到反馈，只有等到任务结束，才能得到浏览器的响应。

我们来看一个典型的场景，来自文章：React 的新引擎—React Fiber 是什么？

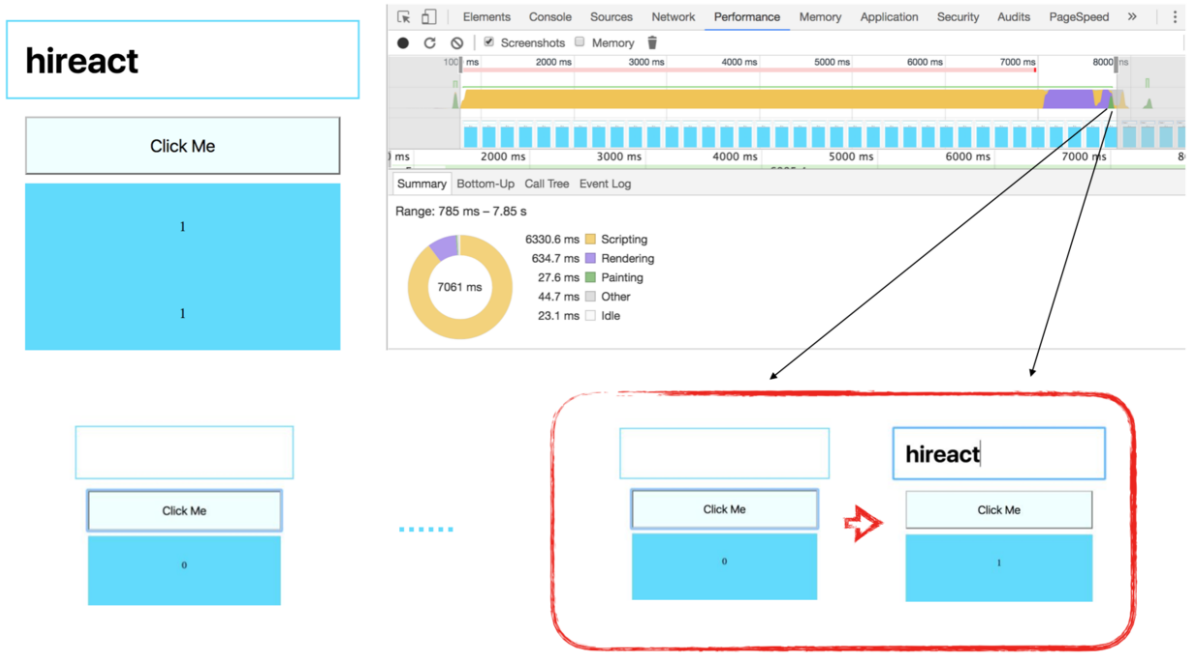
这个例子会在页面中创建一个输入框、一个按钮、一个 BlockList 组件。BlockList 组件会根据 NUMBER_OF_BLOCK 数值渲染出对应数量的数字显示框，数字显示框显示点击按钮的次数。

在这个例子中，我们可以设置 NUMBER_OF_BLOCK 的值为 100000，表示渲染 100000 个矩形框。这时候点击按钮，触发 setState，页面开始更新。此时点击输入框，输入一些字符串，比如 「hi，react」，可以看到：页面没有任何响

应；等待 7s 之后，输入框中突然出现了之前输入的 「hireact」 。同时，BlockList 组件也更新了。

显而易见，这样的用户体验并不好。

浏览器主线程在这 7s 的 performance 如下图所示：



黄色部分：是 JavaScript 执行时间，也是 React 占用主线程的时间。

紫色部分：是浏览器重新计算 DOM Tree 的时间。

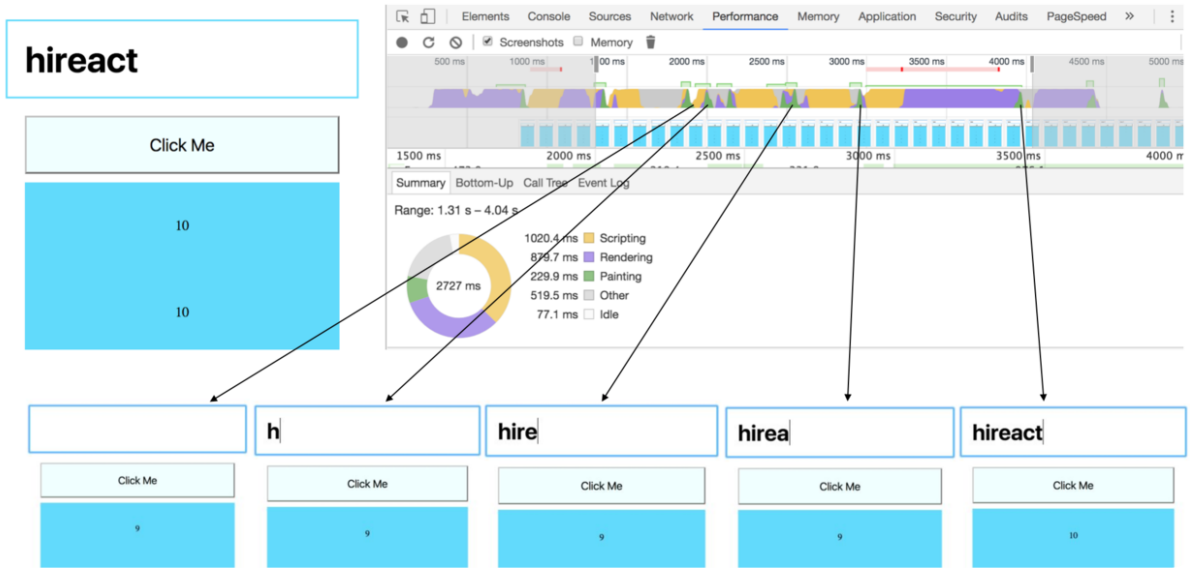
绿色部分：是浏览器绘制页面的时间。

这三种任务，总共占用浏览器主线程 7s 的时间，此时间内浏览器无法与用户交互。主要是黄色部分执行时间较长，占用了 6s，即 React 较长时间占用主线程，导致主线程无法响应用户输入。这就是一个典型的例子。

React 核心团队很早之前就预知性能风险的存在，并且持续探索可解决的方式。基于浏览器对 requestIdleCallback 和 requestAnimationFrame 这两个 API 的支持，React 团队实现新的调度策略 —— Fiber reconcile。

在应用 React Fiber 的场景下，重复刚才的例子，不会再出现页面卡顿，交互自然而顺畅。

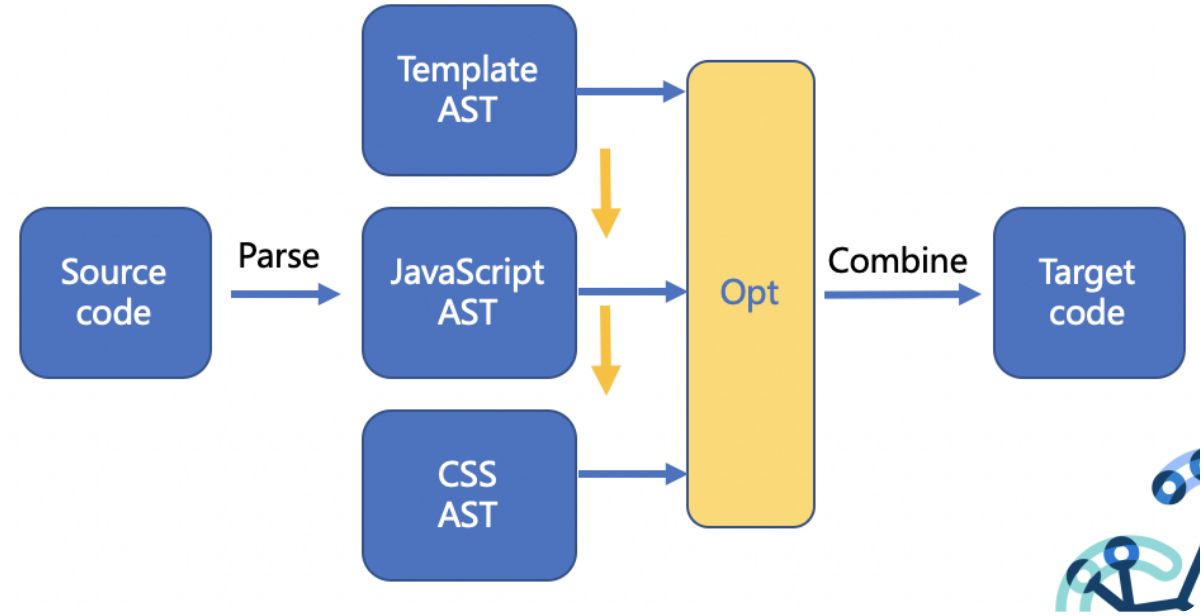
浏览器主线程的 performance 如下图所示：



可以看到：在黄色 JavaScript 执行过程中，也就是 React 占用浏览器主线程期间，浏览器也在重新计算 DOM Tree，并且进行重绘。直观来看，黄色和紫色等互相交替，同时页面截图显示，用户输入得以及时响应。简单说，在 React 占用浏览器主线程期间，浏览器也在与用户交互。这显然是「更好的性能」表现。

从 Vue 3.0 动静结合的 Dom diff 谈起

Vue3.0 提出的动静结合的 DOM diff 思想，我个人认为是 Vue 近几年在「创新」上的一个很好体现。之所以能够做到动静结合的 DOM diff，或者把这个问题放得更大：之所以能够做到预编译优化，是因为 Vue core 可以静态分析 template，在解析模版时，整个 parse 的过程是利用正则表达式顺序解析模板，当解析到开始标签、闭合标签和文本的时候都会分别执行对应的回调函数，来达到构造 AST 树的目的。



这个过程换成代码如下：

```
const ast = parse(template, options)

optimize(ast, options)

const code = generate(ast, options)
```

借助预编译过程，Vue 可以做到的预编译优化就很强大了。比如在预编译时标记出模版中可能变化的组件节点，再次进行渲染前 diff 时就可以跳过「永远不会变化的节点」，而只需要对比「可能会变化的动态节点」。这也就是动静结合的 DOM diff 将 diff 成本与模版大小正相关优化到与动态节点正相关的理论依据。

类似地，我们也可以标记出来一些「快速通道（fast path）」。比如某个复杂的组件之所以 className 发生变化（这个场景很常见，我们根据变量，通过更改 className 来应用不同的样式）。针对这种场景，我们在预编译阶段进行特定的标记，在重新渲染 diff 时只需要更新新的 className 即可。

预编译优化的本质是什么？

我关心的是：React 能否像 Vue 那样进行预编译优化？

Vue 需要做数据双向绑定，需要进行数据拦截或代理，那它就需要在预编译阶段静态分析模版，分析出视图依赖了哪些数据，进行响应式处理。而 React 就是局部重新渲染，React 拿到的或者说掌管的，所负责的就是一堆递归 React.createElement 的执行调用，它无法从模版层面进行静态分析。

比如这样的 JSX：

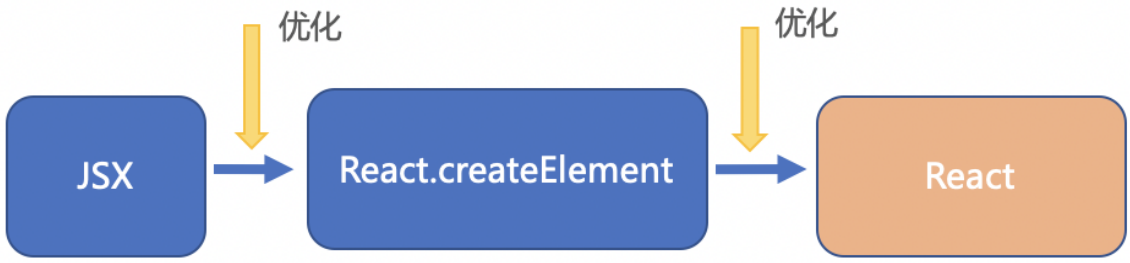
```
        This is a test
```

将会被编译为：

```
React.createElement(
 "div", null,
 React.createElement(
   "p", null,
   React.createElement(
     "span", null, "This is a test"
   )
 )
)
```

因此 React JSX 过度的灵活性导致运行时可以用于优化的信息不足。但是，在 React 框架之外，我们作为开发者还是可以通过工程化手段达到类似的目的，因为我们能够接触到 JSX 编译成 React.createElement 的整个过程。开发者在项目中开发 babel 插件，实现 JSX 编译成 React.createElement，那么优化手段就是是从编写 babel 插件开始：

如图：



那么到底开发者应该怎么做，实现预编译优化呢?

为此我挑出了一些具有代表性的案例，这些案例都是由开发者开发 Babel plugin 实现的 React 预编译手段。

Hoist constant elements

将静态不变的节点在预编译阶段就抽象成函数或者静态变量，这个和 Vue 框架内所做的一样，不过需要开发者实现，这样一来就不需要在每次重新渲染时生成多余实例，只需要调用 _ref 变量即可。

```
const _ref = Hello World

class MyComponent extends React.Component {
 render() {
   return (



       {_ref}


   )
 }
}
```

remove propTypes in runtime

PropTypes 提供了许多验证工具，用来帮助确定 React 组件中 props 数据的有效性。但是，React v15.5 后就被移除了 PropTypes ，因此现在使用 prop-

types 库代替。

propTypes 对于业务开发非常有用，帮助我们弥补了 JS 数据类型检查的不足。但是在线上代码中，propTypes 是多余的。

因此在运行时代码删除 propTypes 就变的比较有必要了。

remove inline functions and varaibles

第三个优化场景是这样的：我们知道组件内如果存在函数生成（箭头函数定义，bind 使用）或者闭包变量的情况下，组件每一次刷新，都会生成一个新的函数或者闭包变量。我们将这种不必要的函数称为 inline functions。

比如下面这段代码中，transformeData 和 onClick 对应的匿名函数，都会随着组件渲染重新生成一个全新的引用。

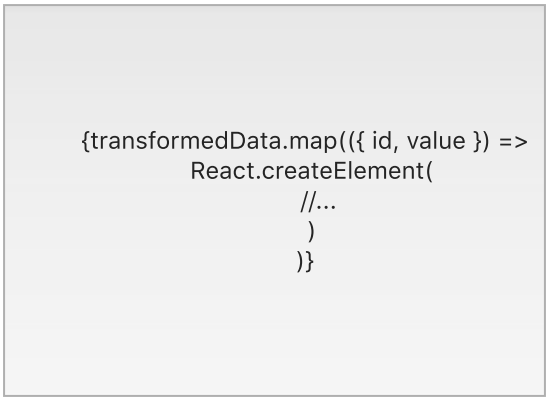
```
export default ({ data, sortComparator, filterPredicate, history }) => {

 const transformedData = data
   .filter(filterPredicate)
   .sort(sortComparator)

 return (

                    className="back-btn"
                  onClick={() => history.pop()}
                    />



      {transformedData.map(({ id, value }) => (

                    ))}

 )
```

```
}
```

反复生成这些 inline functions 或者数据，这对于 React 运行时性能或多或少会有一点影响，也带来了 GC 压力。

我们在工程中，可以通过插件对 inline functions 或者变量进行内存持久化处理。最终经过预编译优化后的代码为：

```
 let _anonymousFnComponent

export default ({ data, sortComparator, filterPredicate,
history }) => {

 const transformedData = React.useMemo(
   () =>
   data.filter(filterPredicate).sort(sortComparator),
   [data, data.filter, filterPredicate, sortComparator]
 )

 return React.createElement(_anonymousFnComponent =
_anonymousFnComponent || (() => {

   const _onClick2 = React.useCallback(
     () => history.pop(),
     [history, history.pop]
   )

   return (
```




```
    )
  }), null)
}
```

我们使用了 React 新特性 useMemo 和 useCallback 将这些变量包裹。useMemo 和 useCallback 都会在组件第一次渲染的时候执行，之后会在其依赖的变量，也就是 useMemo 和 useCallback 的第二个参数数组，数组内的数值发生改变时再次执行；这两个 hooks 都返回缓存的值，useMemo 返回缓存的变量，useCallback 返回缓存的函数。

我们看代码，transformeData 在其数据源：data,data.filter,filterPredicate,sortComparator 发生变化时才会更新，才会重新生成一份 transformeData，函数渲染时只要依赖的 data,data.filter,filterPredicate,sortComparator 不变，不会重新生成 transformeData，而是使用缓存的值。onClick 也使用了 useCallback 将函数引用持久化保存，道理一样。

这样一来就避免了在组件重新渲染时，总是生成不必要的 inline functions 和闭包变量的困扰。

transform to stateless function component

我们知道函数式组件虽然未来会比 class 声明的组件性能更好，并且函数不管是从性能上、可组合性上还是 TS 契合度上，都要要优于 class 使用。

这个例子，我们将符合条件的 class 声明组件自动在预编译阶段转化为函数式组件。

我们的目标是：

```
class MyComponent extends React.Component {
 static propTypes = {
   className: React.PropTypes.string.isRequired
 }

 render() {
   return (


       Hello World


   )
 }
}
```

在预编译阶段优化为：

```
const MyComponent = props =>


   Hello World


MyComponent.propTypes = {
 className: React.PropTypes.string.isRequired
}
```

在这里我们展开实现一下 Babel plugin 的编写，其中会涉及到一些 AST 的内容，读者只需明白思想方向即可。

```
module.exports = function({ types: t }) {
 return {
   visitor: {
     Class(path) {
       const state = {
         renderMethod: null,
         properties: [],
         thisProps: [],
         isPure: true
       }

       path.traverse(bodyVisitor, state)

       let replacement = []

       state.thisProps.forEach(function(thisProp) {
         thisProp.replaceWith(t.identifier('props'))
         thisProp.replaceWith(t.identifier('props'))
       })

       replacement.push(
         t.functionDeclaration(
           id,
           [t.identifier('props')],
           state.renderMethod.node.body
         )
       )

       state.properties.forEach(prop => {
         replacement.push(t.expressionStatement(
           t.assignmentExpression('=',
             t.MemberExpression(id, prop.node.key),
             prop.node.value
           )
         ))
       })
```

```
      if (t.isExpression(path.node)) {
        replacement.push(t.returnStatement(id))

        replacement = t.callExpression(
          t.functionExpression(null, [],
            t.blockStatement(replacement)
          ),
          []
        )
      }

      path.replaceWithMultiple(
        replacement
      )
    }
  }
}

const bodyVisitor = {
  ClassMethod(path) {
    if (path.node.key.name === 'render') {
      this.renderMethod = path
    } else {
      this.isPure = false
      path.stop()
    }
  },

  ClassProperty(path) {
    const name = path.node.key.name

    if (path.node.static && (
      name === 'propTypes' ||
      name === 'defaultProps'
    )) {
      this.properties.push(path)
    } else {
```

```
        this.isPure = false
        this.isPure = false
      }
    },

    MemberExpression(path) {
      this.thisProps.push(path)
    },

    JSXIdentifier(path) {
      if (path.node.name === 'ref') {
        this.isPure = false
        path.stop()
      }
    }
  }
}
```

代码分析：我们先明确，什么样的 class 组件，具备转换成函数式组件的条件?

首先，class 组件不能具有 this.state 的引用，组件不能出现任何生命周期方法，也不能出现 createRef，因为这些特性在函数式组件中并不存在。

满足这样的条件时，我们在进行 JSX 转换过程进行组件替换：通过 AST 进行遍历， 首先在遍历过程中找到符合条件的 class 组件，是否符合条件我们用 isPure 来进行标记， 同时在遍历时，对每一个符合条件的 class 组件，储存 render 方法，作为转换函数式组件的返回值；储存 propTypes 和 defaultProps 静态属性，之后会挂载在函数组件函数属性上；同时对 this.props 的用法转为 props, props 作为函数式组件的参数出现 最后在按照上述规则，修改 AST 树，新的 AST 树相关组件节点会生成函数式组件。

Prepack 对于框架的影响

Prepack 同样是 FaceBook 团队的作品。它让你编写普通的 JavaScript 代码，它在构建阶段就试图了解代码将做什么，然后生成等价的代码，减少了运行时的计算量。

我们看一个 fibonacci 数列求和的例子，再经过 prepack 处理之后，直接输出结果，运行时就是一个 610 这么一个结果。这么看 prepack 是一个 JavaScript 的部分求值器（Partial Evaluator），可在编译时执行原本在运行时的计算过程，并通过重写 JavaScript 代码来提高其执行效率。

我就用 Prepack 结合 React 尝了个鲜：

```
(function () {

  const Bar = (props) => {
    return <div>{props.text}</div>;
  }

  const Foo = props => {
    var list = ['1', '2']

    return (
      <div>
        {list.map(item => <Bar text={item} /> )}
      </div>
    );
  }

  // __optimizeReactComponentTree(Foo)
  Foo
  window.Foo = Foo
})()
```

```
(function () {
  var _$0 = this;

  var _1 = props => {
    var list = ['1', '2'];
    return <div>
        {list.map(item => <_2 text={item} />)}
      </div>;
  };

  var _2 = props => {
    return <div>{props.text}</div>;
  };

  _$0.Foo = _1;
}).call(this);
```

```
(function () {

  const Bar = (props) => {
    return <div>{props.text}</div>;
  }

  const Foo = props => {
    var list = ['1', '2']

    return (
      <div>
        {list.map(item => <Bar text={item} /> )}
      </div>
    );
  }

  __optimizeReactComponentTree(Foo)
  // Foo
  window.Foo = Foo
})()
```

```
(function () {
  var _$0 = this;

  var _1 = (props, context) => {
    _4 === void 0 && $f_0();
    return _2;
  };

  var $f_0 = function () {
    _4 = <div>1</div>;
    _7 = <div>2</div>;
    _2 = <div>{_4}{_7}</div>;
  };

  var _4;

  var _7;

  var _2;

  _$0.Foo = _1;
}).call(this);
```

上图左边部分是我编写的代码，在不使用 prepack 情况下，运行时代码如右边所示：经过编译之后右边的代码仍然是对数组 list 进行 map，逐条渲染出数组内容。

经过 preack 优化后，运行时代码已经非常轻量了。运行时就减少 map 的计算等，直接用生成的组件内容作为运行时结果。

点击查看下一节

揭秘前端设计模式（1）