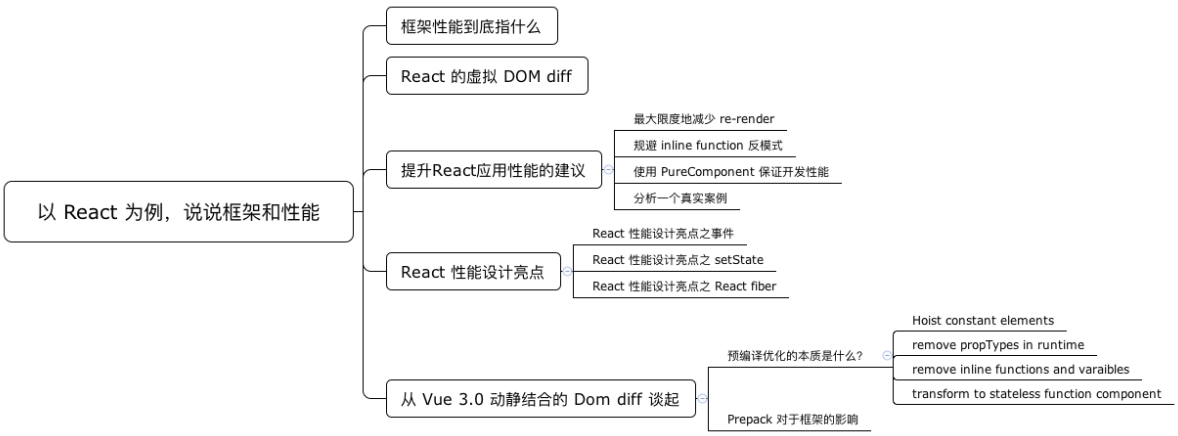
以 React 为例，说说框架和性能（1）

在上一节课中，我们提到了性能优化。在这个话题上，除了工程化层面的优化和语言层面的优化以外，框架性能也备受瞩目。这一节课，我们就来聊聊框架的性能话题，并以 React 为例进行分析。

主要知识点如下：



框架性能到底指什么

说起框架的性能话题，很多读者可能会想到「不要过早地做优化」这条原则。实际上，大部分应用的复杂度并不会对性能和产品体验构成挑战。毕竟在之前课程中我们学习到，现代化的框架凭借高效的虚拟 DOM diff 算法和（或）响应式理念，以及框架内部引擎，已经做得较为完美了，一般项目需求对性能的压力并不大。

但是对于一些极其复杂的需求，性能优化是无法回避的。如果你开发的是图形处理应用、DNA 检测实验应用、富文本编辑器或者功能丰富的表单型应用，则很容易触碰到性能瓶颈。同样，作为框架的使用者，也需要对性能优化有所了解，这对于理解框架本身也是有很大帮助的。

前端开发自然离不开浏览器，而性能优化大都在和浏览器打交道。我们知道，页面每一帧的变化都是由浏览器绘制出来的，并且这个绘制频率受限于显示器的刷新频率，因此一个重要的性能数据指标是每秒 60 帧的绘制频率。这样进行简单的换算之后，每一帧只有 16.6ms 的绘制时间。

如果一个应用对用户的交互响应处理过慢，则需要花费很长的时间来计算更新数据，这就造成了应用缓慢、性能低下的问题，被用户感知造成极差的用户体验。对于框架来说，以 React 为例，开发者不需要额外关注 DOM 层面的操作。因为 React 通过维护虚拟 DOM 及其高效的 diff 算法，可以决策出每次更新的最小化 DOM batch 操作。**但实际上，使用 React 能完成的性能优化，使用纯原生的 JavaScript 都能做到，甚至做得更好。只不过经过 React 统一处理后，大大节省了开发成本，同时也降低了应用性能对开发者优化技能的依赖。**

因此现代框架的性能表现，除了想办法缩减自身的 bundle size 之外，主要优化点就在于框架本身运行时对 DOM 层操作的合理性以及自身引擎计算的高效性。这一点我们会通过两节课程来慢慢展开。

React 的虚拟 DOM diff

React 主要通过以下几种方式来保证虚拟的 DOM diff 算法和更新的高效性能：

- 高效的 diff 算法
- Batch 操作
- 摒弃脏检测更新方式

当任何一个组件使用 setState 方法时，React 都会认为该组件变「脏」，触发组件本身的重新渲染（re-render）。同时因其始终维护两套虚拟的 DOM，其中一套是更新后的虚拟的 DOM；另一套是前一个状态的虚拟的 DOM。通过对这两套虚拟的 DOM 的 diff 算法，找到需要变化的最小单元集，然后把这个最小单元集应用在真实的 DOM 当中。

而通过 diff 算法找到这个最小单元集后，React 采用启发式的思路进行了一些假设，将两棵 DOM 树之间的 diff 成本由 O(n3) 缩减到 O(n)。

说到这里，你一定很想知道 React 的那些大胆假设吧：

DOM 节点跨层级移动忽略不计

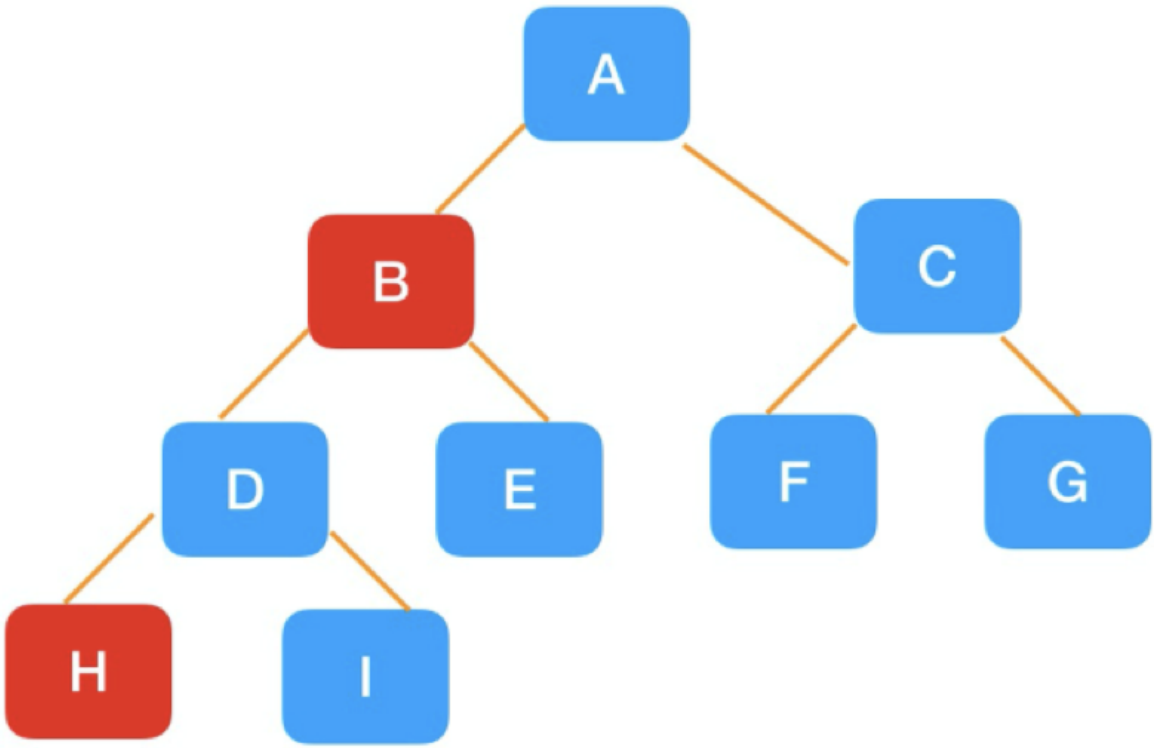
拥有相同类的两个组件生成相似的树形结构，拥有不同类的两个组件生成不同的树形结构

根据这些假设，ReactJS 采取的策略如下：

React 对组件树进行分层比较，两棵树只会对同一层级的节点进行比较

当对同一层级节点进行比较时，对于不同的组件类型，直接将整个组件替换为新类型组件

对于下图所示的组件结构，我们可以想象：如果子组件 B 和 H 的类型同时发生变化，当遍历到 B 组件时，直接进行新组件的替换，减少了不必要的消耗。



当对同一层级节点进行比较时，对于相同的组件类型，如果组件的 state 或 props 发生变化，则直接重新渲染组件本身。开发者可以尝试使用 shouldComponentUpdate 生命周期函数来规避不必要的渲染。

当对同一层级节点进行比较时，开发者可以使用 key 属性来「声明」同一层级节点的更新方式。

另外，setState 方法引发了「蝴蝶效应」，并通过创新的 diff 算法找到需要更新的最小单元集，但是这些变更也并不一定立即同步产生。实际上，React 会进行 setState 的 batch 操作，通俗地讲就是「积攒归并」一批变化后，再统一进行更新。显然这是出于对性能的考虑。

提升 React 应用性能的建议

我们知道，React 渲染真实的 DOM 节点的过程由两个主要过程组成：

对 React 内部维护的虚拟的 DOM 进行更新

前后两个虚拟 DOM 比对，并将 diff 所得结果应用于真实的 DOM 中的过程

这两步极其关键，设想一下，如果虚拟的 DOM 更新很慢，那么重新渲染势必会很耗时。本节我们就针对此问题，对症下药，来了解更多的性能优化小技巧。

最大限度地减少 re-render

为了提升 React 应用性能，我们首先想到的就是最大限度地规避不必要的 re-render。但是当状态发生变化时，重新渲染是 React 内部的默认行为，我们如何保证不必要的渲染呢?

最先想到的一定是使用 shouldComponentUpdate 生命周期函数，它旨在对比前后状态 state/props 是否出现了变更，根据是否变更来决定组件是否需要重新渲染。

实际上，还有很多方式，开发者都可以给 React 发送「不需要渲染」的信号。

比如，无状态组件返回同一个 element 实例：如果 render 方法返回同一个 element 实例，React 会认为组件并没有发生变化。请参考以下代码：

```
class MyComponent extends Component {
 text = "";
```

```
 renderedElement = null;
 _render() {
   return

{this.props.text}

 }
 render() {
      if (!this.renderedElement || this.props.text !==
this.text) {
     this.text = this.props.text;
     this.renderedElement = _render();
   }
   return this.renderedElement;
 }
```

熟悉 lodash 库的读者，可能会想到其带来的 memoize 函数，同样可以用来简化上述代码：

```
import memoize from 'lodash/memoize'

class MyComponent extends Component {
 _render = memoize((text) =>

{text}
)
 render() {
   return _render(this.props.text)
 }
}
```

在之前介绍的高阶组件的基础上，我们不妨设想这样一类高阶组件：它能够细粒度地控制组件的渲染行为。比如，某个组件仅仅在某一项 props 变化时才会触发 re-render。这样一来，开发者可以完全掌控组件渲染时机，更有针对性地进行渲染优化。

这样的方法有点类似于农业灌溉上的「滴灌」技术，它规避了代价昂贵的粗暴型灌溉，而是精准地定位需求，从而达到节约水资源的目的。

在社区中，优秀的 recompose 库恰好可以满足我们的需求。请参考如下代码：

```
@onlyUpdateForKeys(['prop1', 'prop2'])
class MyComponent2 extends Component {
 render() {
     //...
 }
}
```

使用 @onlyUpdateForKeys 修饰器，MyComponent2 组件只在 prop1 和 prop2 变化时才进行渲染；否则其他的 props 发生任何改变，都不会触发 re-render。

藏在 onlyUpdateForKeys 背后的「黑魔法」其实并不难理解，只需要在高阶组件中调用 shouldComponentUpdate 方法，在 shouldComponentUpdate 方法中比较对象由完整的 props 转为传入的指定 props 即可。有兴趣的读者，可以翻阅 recompose 源码进行了解，其实思路即是如此。

规避 inline function 反模式

我们需要注意一个「反模式」。当使用 render 方法时，要留意 render 方法内创建的函数或者数组等，这些创建可能是显式地，也可能是隐式生成。因为这些新生成的函数或数组，在量大时会造成一定的性能负担。同时 render 方法经常被反复执行多次，也就是说总有新的函数或数组被创建，这样造成内存无意义开销。往往性能更友好的做法只需要它们创建一次即可，而不是每次渲染都被创建。比如：

```
render() {
 return ;
}
```

或者：

```
render() {
 return this.props.update()} />;
}
```

对于 render 方法内产生数组或其他类型的情况，也存在类似问题：

```
render() {
 return
}
```

这样做会在每次渲染且 this.props.items 不存在时创建一个空数组。更好的做法是：

```
const EMPTY_ARRAY = []
render() {
   return
}
```

事实上，不得不说，这些性能副作用或者优化手段都「微乎其微」，并不是性能恶化的「罪魁祸首」。但是理解这些内容对我们编写出高质量的代码还是有帮助的。我们后续课程会针对这种情况进行框架层面上的启发式探索。

使用 PureComponent 保证开发性能

PureComponent 大体与 Component 相同，唯一不同的地方是 PureComponent 会自动帮助开发者使用 shouldComponentUpdate 生命周期方法。也就是说，当组件 state 或者 props 发生变化时，正常的 Component 都会自动进行 re-render，在这种情况下，shouldComponentUpdate 默认都会返回 true。但是 PureComponent 会先进行对比，即比较前后两次 state 和 props 是否相等。需要注意的是，这种对比是浅比较：

```
function shallowEqual (objA: mixed, objB: mixed) {
   if (is(objA, objB)) {
       return true;
   }
```

```
    if (typeof objA !== 'object' || objA === null ||
        typeof objB !== 'object' || objB === null) {
        return false;
    }

    const keysA = Object.keys(objA);
    const keysB = Object.keys(objB);

    if (keysA.length !== keysB.length) {
        return false;
    }

    for (let i = 0; i < keysA.length; i++) {
        if (
        !hasOwnProperty.call(objB, keysA[i]) ||
        !is(objA[keysA[i]], objB[keysA[i]])
        ) {
            return false;
        }
    }

    return true;
}
```

基于以上代码，我们总结出使用 PureComponent 需要注意如下细节：

> 既然是浅比较，也就是说，当与前一状态下的 props 和 state 比对时，如果比较对象是 JavaScript 基本类型，则会对其值是否相等进行判断；如果比较对象是 JavaScript 引用类型，比如 object 或者 array，则会判断其引用是否相同，而不会进行值比较；
>
> 开发者需要避免共享（mutate）带来的问题。

如果在一个父组件中对 object 进行了 mutate 的操作，若子组件依赖此数据，且采用 PureComponent 声明，那么子组件将无法进行更新。尽管 props 中的某一项值发生了变化，但是它的引用并没有发生变化，因此 PureComponent 的

shouldComponentUpdate 也就返回了 false。更好的做法是在更新 props 或 state 时，返回一个新的对象或数组。

分析一个真实案例

设想一下，如果应用组件非常复杂，含有一个具有很长 list 的组件，如果只是其中一个子组件发生了变化，那么使用 PureComponent 进行对比，有选择性地进行渲染，一定是比所有列表项目都重新渲染划算很多。

我们来看一个案例：简易实现一个采用 PureComponent 和不采用 PureComponent 的性能差别对比试验。假如在页面中需要渲染非常多的用户信息，所有的用户信息都被维护在一个 users 数组当中，数组的每一项为一个 JavaScript 对象，表示一个用户的基本信息。User 组件负责渲染每一个用户的信息内容：

```
import User from './User'
const Users = ({users}) =>



        {users.map(user => }
```

这样做存在的问题是：users 数组作为 Users 组件的 props 出现，users 数组的第 K 项发生变化时，users 数组即发生变化，Users 组件重新渲染导致所有的 User 组件都会进行渲染。某个 User 组件，即使非 K 项并没有发生变化，这个 User 组件不需要重新渲染，但也不得不必要的渲染。

在测试中，我们渲染了一个有 200 项的数组：

```
const arraySize = 200;
const getUsers = () =>
 Array(arraySize)
   .fill(1)
   .map((_, index) => ({
     name: 'John Doe',
```

```
    hobby: 'Painting',
    age: index === 0 ? Math.random() * 100 : 50
  }));
```

注意：在 getUsers 方法中，对 age 属性进行了判断，保证每次调用时，getUsers 返回的数组只有第一项的 age 属性不同，其余的全部为 50。在测试组件中，在 componentDidUpdate 中保证数组将会触发 400 次 re-render，并且每一次只改变数组第一项的 age 属性，其他的均保持不变。

```
  const repeats = 400;
 componentDidUpdate() {
   ++this.renderCount;
   this.dt += performance.now() - this.startTime;
   if (this.renderCount % repeats === 0) {
     if (this.componentUnderTestIndex > -1) {
       this.dts[componentsToTest[this.componentUnderTestIn
dex]] = this.dt;
       console.log(
         'dt',
         componentsToTest[this.componentUnderTestIndex],
         this.dt
       );
     }
     ++this.componentUnderTestIndex;
     this.dt = 0;
     this.componentUnderTest =
componentsToTest[this.componentUnderTestIndex];
   }
   if (this.componentUnderTest) {
     setTimeout(() => {
       this.startTime = performance.now();
       this.setState({ users: getUsers() });
     }, 0);
   }
   else {
     alert(`
       Render Performance ArraySize: ${arraySize} Repeats:
```

```
${repeats}
        Functional: ${Math.round(this.dts.Functional)} ms
        PureComponent:
${Math.round(this.dts.PureComponent)} ms
        Component: ${Math.round(this.dts.Component)} ms
      `);
    }
 }
```

下面对三种组件声明方式进行对比。

函数式方式

```
export const Functional = ({ name, age, hobby }) => (



    {name}
    {age}
    {hobby}


)
```

PureComponent 方式

```
export class PureComponent extends React.PureComponent {
 render() {
    const { name, age, hobby } = this.props;
    return (



        {name}
        {age}
        {hobby}
```

```
    )
  }
}
```

经典 class 方式

```
export class Component extends React.Component {
 render() {
   const { name, age, hobby } = this.props;
   return (



       {name}
       {age}
       {hobby}



    )
 }
```

在使用 PureComponent 声明的组件中，会自动在触发渲染前后进行 {name, age, hobby} 对象值比较。如果没有发生变化，则 shouldComponentUpdate 返回 false，以规避不必要的渲染。因此，使用 PureComponent 声明的组件性能明显优于其他方式。在不同的浏览器环境下，可以得出：

在 Firefox 下，PureComponent 收益 30%

在 Safari 下，PureComponent 收益 6%

在 Chrome 下，PureComponent 收益 15%

实际上，我们通过定义 changedItems 来表示变化数组的项目，array 表示所需渲染的数组。changedItems.length/array.length 的比值越小，表示数组中变化的元素也越少，React.PureComponent 涉及的性能优化也越有必要实施，因为

React.PureComponent 通过浅比较规避了不必要的更新过程，而浅比较自身的计算成本一般都不值一提，可以节约成本。

当然，PureComponent 也不是万能的，尤其是它的浅比较，需要开发者格外注意。因此在特定情况下，开发者根据需求自己实现 shouldComponentUpdate 中的比较逻辑，将是更高效的选择。

总结

性能优化是前端开发中一个永恒的话题，不同框架之间的性能对比也一直是各位开发者关注的方面。性能涉及方方面面，如前端工程化、浏览器解析和渲染、比较算法等。本章主要介绍了 React 框架在性能上的优劣、虚拟的 DOM 思想，以及在开发 React 应用时需要注意的性能优化环节和手段。 也许不是每个应用都会面临性能的问题，如同社区中所说的：「过早地进行性能优化是毫无必要的，但是开发者在性能优化方面的积累却要时刻先行。」同时，优化手段也在与时俱进，不断更新，需要开发者时刻保持学习。

点击查看下一节

以 React 为例，说说框架和性能（2）