一道React面试题把我整懵了

提问：react项目中的JSX里，onChange={this.func.bind(this)}的写法，为什么要比非bind的func = () => {}的写法效率高?

> 声明: 由于本人水平有限，有考虑不周之处，或者出现错误的，请严格指出，小弟感激不尽。这是小弟第一篇文章，有啥潜规则不懂的，你们就告诉我。小弟明天有分享，等分享完了之后，继续完善。

之前不经意间看到这道题，据说是阿里p5-p6级别的题目，我们先看一下这道题目，明面上是考察对react的了解深度，实际上涉及的考点很多：bind，arrow function，react各种绑定this的方法，优缺点，适合的场景，类的继承，原型链等等，所以综合性很强。

我们今天的主题就是由此题目，来总结一下相关的知识点，这里我会**着重分析题目中第二种绑定方案。**

五种this绑定方案的差异性

方案一: React.createClass

这是老版本React中用来声明组件的方式，在那个版本，没有引入class这种概念，所以通过这种方式来创建一个组件类（constructor） ES6的class相比createClass，移除了两点：一个是mixin 一个是this的自动绑定。前者可以用HOC替代，后者则是完完全全的没有，原因是FB认为这样可以避免和JS的语法产生混淆，所以去掉了。 使用这种方法，我们不需要担心this，它会自动绑定到组件实例身上，但是这个API已经废弃了，所以只需要了解。

javascript 复制代码

```
const App = React.createClass({
  handleClick() {
    console.log(this)
  },
  render() {
    return <div onClick={this.handleClick}>你好</div>
  }
})
```

方案二：在render函数中使用bind

```scala
class Test extends Component {
  handleClick() {
    console.log(this)
  }
  render() {
    return <div onClick={this.handleClick.bind(this)}></div>
  }
}
```

方案三：在render函数中使用箭头函数

```scala
class Test extends Component {

  handleClick() {
    console.log(this)
  }
  render() {
    return <div onClick={() => this.handleClick()}></div>
  }
}
```

> 这两个方案简洁明了,可以传参，但是也存在潜在的性能问题: **会引起不必要的渲染**

我们常常会在代码中看到这些场景: 更多演示案例请点击

```scala
class Test extends Component {
  render() {
    return <div>
      <Input />
      <button>添加<button>
      <List options={this.state.options || Immutable.Map()} data={this.state.data} onSelect={this.or
    </div>
  }
}
```

场景一：使用空对象/数组来做兜底方案，避免options没有数据时运行时报错。 场景二：使用箭头函数来绑定this。

可能在一些不需要关心性能的场景下这两种写法没有什么太大的坏处，但是如果我们正在考虑性能优化，譬如我们使用了 `PureComponent` 来去优化我们的渲染性能 这里面React有使用shallowEqual做第一层的比较，这个时候我们关注的可能是这个data(数据是否有变化从而影响渲染)，然而被我们忽视的options，onSelect却会直接导致PureComponent失效，然而我们找不到优化失败的原因。

而假设我们的核心data是 `Immutable` 的，这样其实优化了我们做diff相关的性能。当data为null时，此时我们期望的是不会重复渲染，然而当我们的Test组件有状态更新，触发了Test的重新渲染，此时render执行，List依旧会重新渲染。原因就是 `我们每次执行render，传递给子组件的options，onSelect是一个新的对象/函数。` 这样在做shallowEqual时，会认为有更新，所以会更新List组件。参考 前端进阶面试题详细解答

这个地方也有很多解决方案：

1. 不要直接在render函数里面做兜底，或者使用同一引用的数据源
2. 对于事件监听函数，我们可以事先做好绑定，使用方案4或者5，或者最新的hook（useCallback、useMemo）

javascript 复制代码

```
const onSelect = useCallback(() => {
  ... //和select相关的逻辑
}, []) // 第二个参数是相关的依赖，只有依赖变了，onSelect才会变，设置为空数组，表示永远不变
```

方案四：在构造函数中使用bind

scala 复制代码

```
class Test extends Component {
  constrcutor() {
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    console.log(this)
  }

  render() {
    return <Button onClick={this.handleClick}>测试</Button>
  }
}
```

这种方案是React推荐的方式，只在实例化组件的时候做一次绑定，之后传递的都是同一引用，没有方案二、三带来的负面效应。

但是这种写法相对2，3繁琐了许多：

> ~~1. 如果我们并不需要在构造函数里做什么的话，为了做函数绑定，我们需要手动声明构造函数；~~ 这里没有考虑到实例属性的新写法，直接在顶层赋值。感谢@Yes好2012指正。
>
> 1. 针对一些复杂的组件（要绑定的方法过多），我们需要多次重复的去写这些方法名；
> 2. 无法单独处理传参问题(这一点尤其重要，也限制了它的使用场景)。

方案五：使用箭头函数定义方法（class properties）

这种技术依赖于 `Class Properties` 提案，目前还在 `stage-2` 阶段，如果需要使用这种方案，我们需要安装 `@babel/plugin-proposal-class-properties`

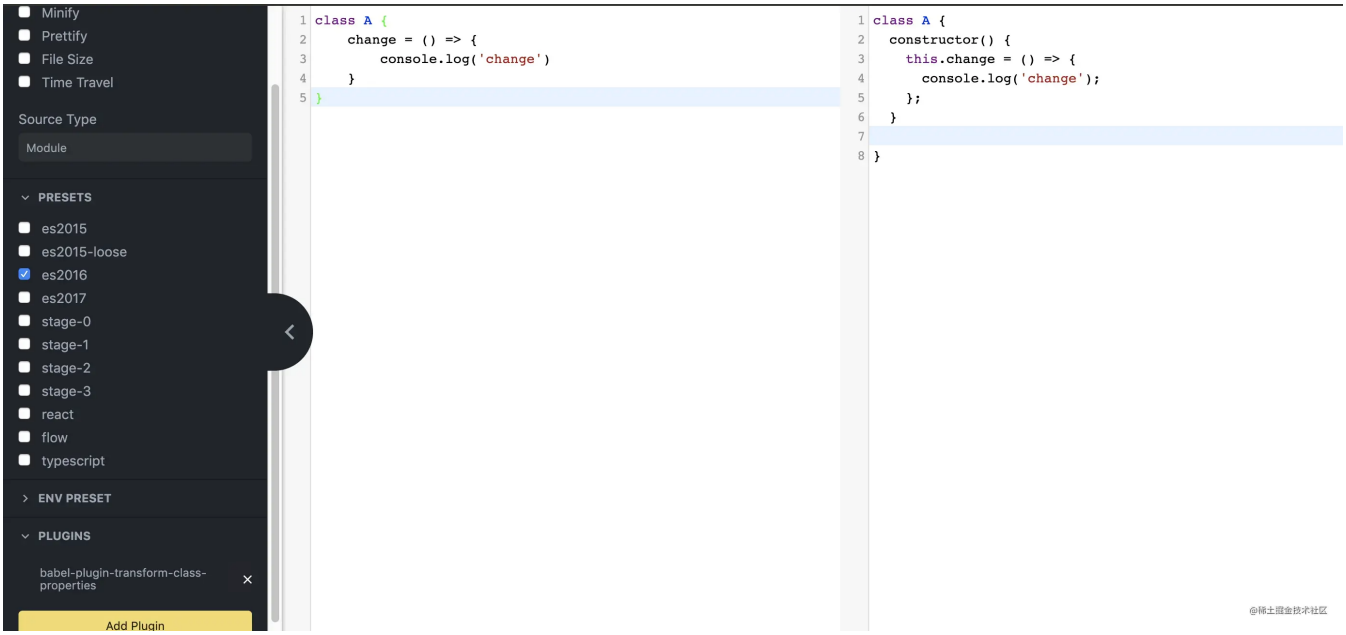
```scala
class Test extends Component {
  handleClick = () => {
    console.log(this)
  }

  render() {
    return <button onClick={this.handleClick}>测试</button>
  }
}
```

这也是我们面试题中提到的第二种绑定方案 先总结一下优点：

> 1. 自动绑定
> 2. 没有方案二、三所带来的渲染性能问题（只绑定一次，没有生成新的函数）；
> 3. 可以再封装一下，使用 `params => () => {}` 这种写法来达到传参的目的。

我们在babel上做一下编译：点击class-properties（选择ES2016或者更高，需要手动安装一下这个plugin `babel-plugin-transform-class-properties` 相比于 `@babel/plugin-proposal-class-properties` 更直观，前者是babel6命名方式，后者是babel7）



在使用plugin编译后的版本我们可以看到，这种方案其实就是直接在构造函数中定义了一个change属性，然后赋值为箭头函数，从而实现的对this的绑定，看起来很完美，很精妙。然而，正是因为这种写法，意味着由这个组件类实例化的所有组件实例都会分配一块内存来去存储这个箭头函数。而我们定义的普通方法，其实是定义在原型对象上的，被所有实例共享，牺牲的代价则是需要我们使用bind手动绑定，生成了一个新的函数。

我们看一下bind函数的polyfill：

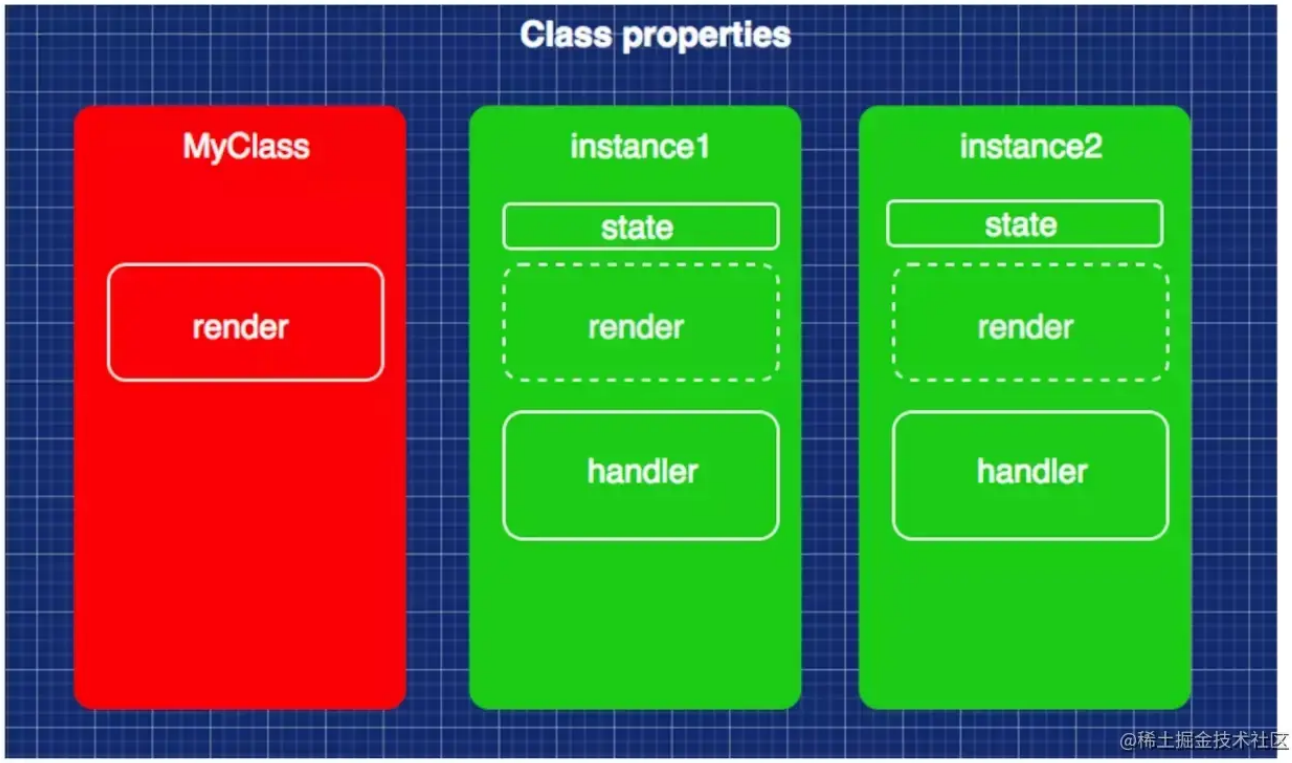
javascript 复制代码

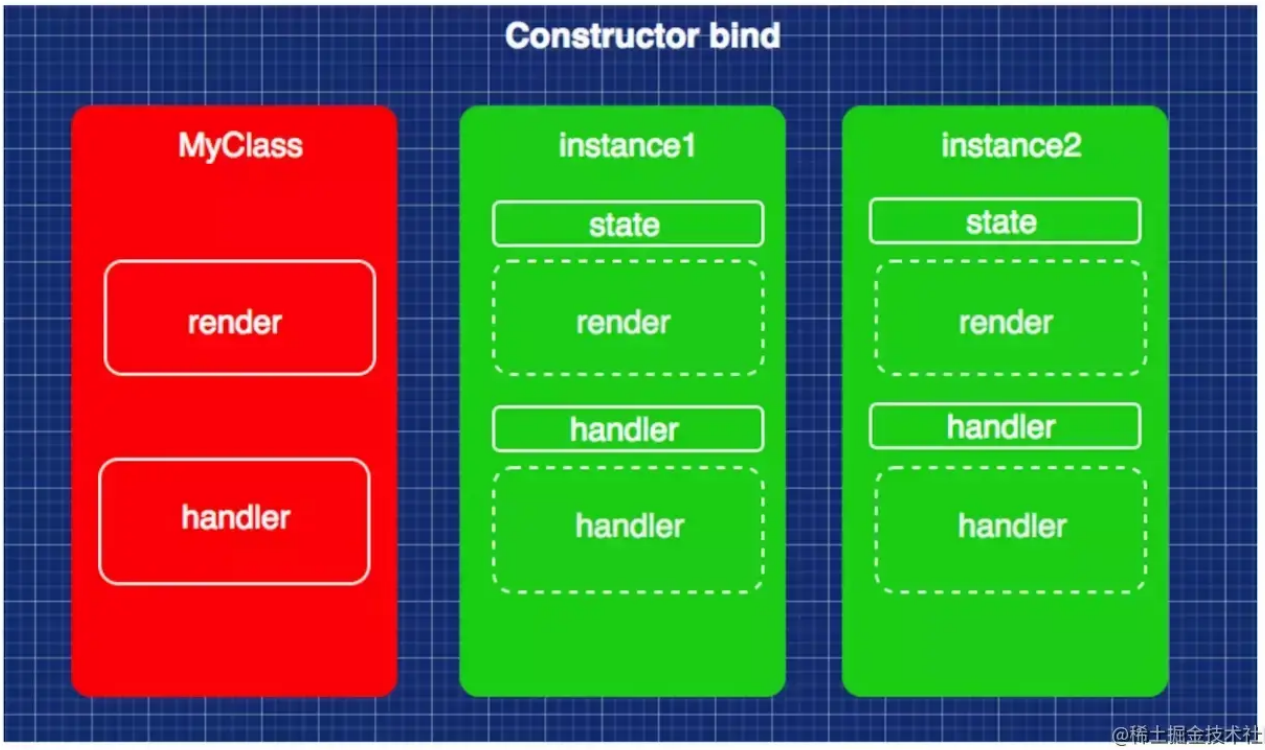
```
if (!Function.prototype.bind) {
    ... // do sth
    var fBound  = function() {
          // this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用
          return fToBind.apply(this instanceof fBound
                 ? this
                 : oThis,
                 // 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的
                 aArgs.concat(Array.prototype.slice.call(arguments)));
        };
    ... // do sth

    return fBound;
  };
}
```

如果在不支持bind的浏览器上，其实编译后，也就相当于新生成的函数的函数体就一条语句: `fToBind.apply(...)` 。

我们以图片的形式看一下差距：





> **注：** 图中，虚线框面积代表引用函数所节省的内存，实线框的面积代表消耗的内存。 图一：使用箭头函数做this绑定。只有render函数定义在原型对象上，由所有实例对象共享。其他内存消耗都是基于每个实例上的。 图二：在构造函数中做this绑定。render，handler都定义在原型对象上，实例上的handler实线框代表使用bind生成的函数所消耗的内存大小。

如果我们的handler函数体本身就很小，实例数量不多，绑定的方法不多。两种方案在内存占用上的差异性不大，但是一旦我们 要在handler里处理复杂的逻辑 ，或者该 组件可能会产生大量的实例 ，抑或是该 组件有大量的需要绑定方法 ，第一种的优势就突显出来了。

如果说上面这种绑定this的方案只用在React上，可能我们只需要考虑上面几点，但是如果我们使用上面的方法去创建一些工具类，可能注意的不止这些。

说到类，可能大家都会想到类的继承，如果我们需要重写某个基类的方法，运行下面，你会发现，和想象中的相差甚远。

javascript 复制代码

```
class Base {
  sayHello() {
    console.log('Hello')
  }

  sayHey = () => {
    console.log('Hey')
  }
}

class A extends Base {
  constructor() {
    super()
    this.name = 'Bitch'
  }

  sayHey() {
    console.log('Hey', this.name)
  }
}

new A().sayHello()  // 'Hello'
new A().sayHey() // 'Hey'
```

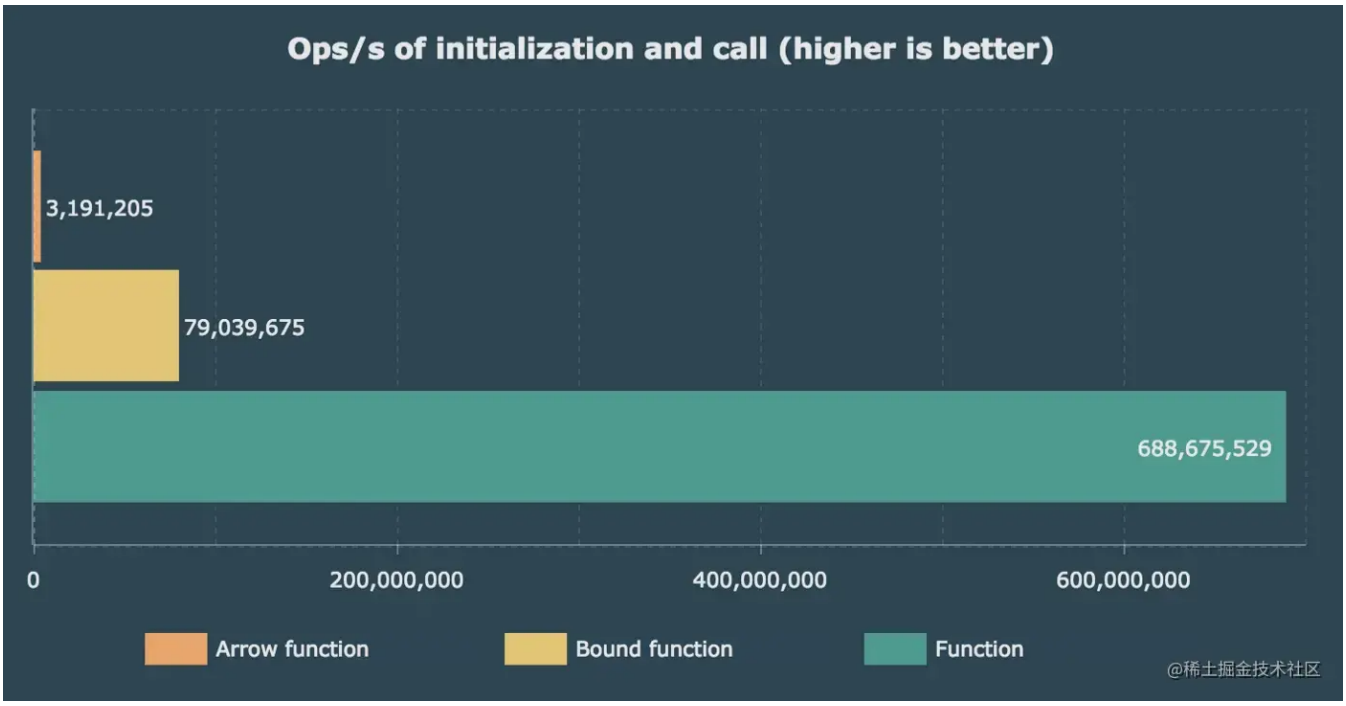
> **注：** 我们希望打印出 'Hello' 'Hey Bitch'，实际打印的是：'Hello' 'Hey'

原因很简单，在A的构造函数内，我们调用super执行了Base的构造函数，向A实例上添加属性，这个时候执行Base构造函数后，A实例上已经有了sayHey属性，它的值是一个箭头函数，打印出·Hey· 而我们重写的sayHey其实是定义在原型对象上的。所以最终执行的是在Base里定义的sayHey方法，但不是同一个方法。 据此，我们还可以推理一下假设我们要先执行Base的sayHey，然后在此基础上执增加逻辑我们又该怎么做？下面这种方案肯定是行不通的。

```
javascript 复制代码
```

```
sayHey() {
  super.sayHey() // 报错
  console.log('get off!')
}
```

多说一句： 有大佬认为这种方法的性能并不好，它考察的点是ops/s（每秒可以实例化多少个组件，越多越好），最终得出的结论是



但是就有人提出质疑，这些方法我们最终都会通过babel编译成浏览器能识别的代码，那么最终运行的版本所体现的差异性是否能够代表其真实的差异性。具体的我也没细看，有需要了解更多的，可以 看一下这篇文章Arrow Functions in Class Properties Might Not Be As Great As We Think

据此，我们已经cover了这道题多数考点，如果下次碰到这种题，或者想出这类题不妨从下面的角度去考虑下

1. 面试者的角度： 1.1 在回答这道题之前，写解释两种方案的原理，显然，面试官想要着重考察的是第二种的了解情况，他背后到底做了什么。然后谈谈他们一些常规的优缺点 1.2 回答关于效率的问题，前者每次bind，都会生成一个新的函数，但是函数体内代码量少，最重要的还是引用的原型上的handler,这个是共享的。但是后面这一种，他会在每个实例上生成一个函数，如果实例数量多，或者函数体大，或者是绑定函数过多，那么占用的内存就明显要超出第一种。
2. 面试官的角度: 考bind实现，考react的绑定策略，优缺点，考性能优化策略，考箭头函数， 考原型链，考继承。发散开来，真的很广。

总结：

每种绑定方案既然存在就有其存在的理由（除了第一种已经是过去），但是也会有相应的弊端，并没有绝对的谁好谁差，我们在使用时，可以根据实际场景做选择。 这道题目答到点不难，怎样让面试官觉得你懂得全面还是挺难的。

其次针对this绑定方案， **如果特别在意性能，牺牲一点代码量，可读性：推荐四其次，如果自己本身够细心，二三也可以使用，但是一定要注意新生成的函数是否会导致多余渲染； 如果想不加班：推荐五（如何传参文章中有提及）。**