React 应用设计之道 - curry 化妙用

Lucas HC
前端开发话题下的优秀答主
关注他

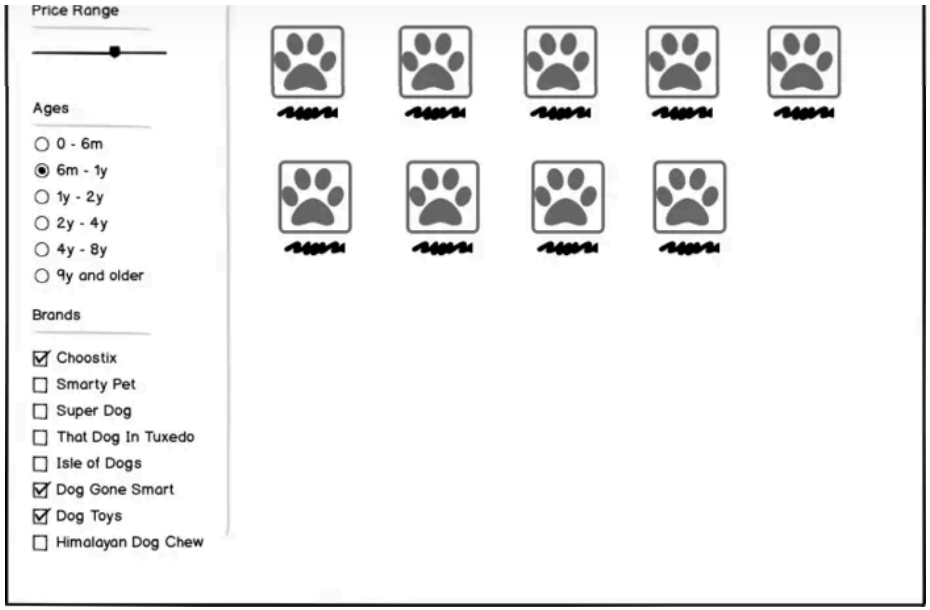
使用 React 开发应用，给予了前端工程师无限“组合拼装”快感。但在此基础上，组件如何划分，数据如何流转等应用设计都决定了代码层面的美感和强健性。

同时，在 React 世界里提到 curry 化，也许很多开发者会第一时间反应出 React-redux 库的 connect 方法。然而，如果仅仅机械化地停留于此，而没有更多灵活地应用，是非常可惜的。

这篇文章以一个真实场景为基础，从细节出发，分析 curry 化如何化简为繁，更优雅地实现需求。

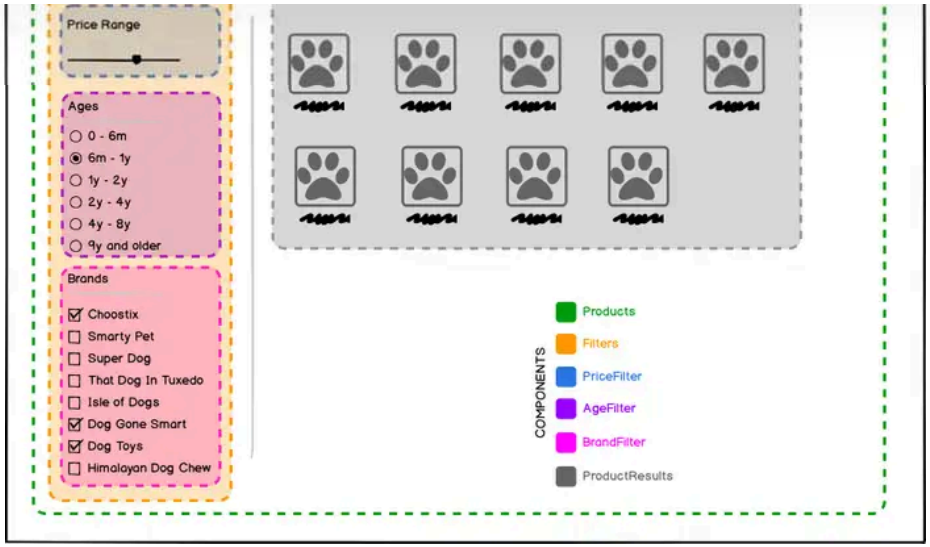
场景介绍

需求场景为一个卖食品的电商网站，左侧部分为商品筛选栏目，用户可以根据：价格区间，商品年



页面示意图

作为 React 开发者，我们知道 React 是组件化的，第一步将考虑根据 UE 图，进行组件拆分。这个过程比较简单直观，我们对拆分结果用下图表示：



组件设计

对应代码为：

```
<Products>
    <Filters>
        <PriceFilter/>
        <AgeFilter/>
        <BrandFilter/>
    </Filters>
    <ProductResults/>
</Products>
```

至此，我们对场景有了大致了解，这在互联网产品中较为常见。

初级实现

React 是基于数据状态的，紧接着第二步就要考虑应用状态。商品展现结果数据我们暂时不需要关心。这里主要考虑应用最重要的状态，即**过滤条件信息**。

我们使用命名为 filterSelections 的 JavaScript 对象表示过滤条件信息，如下：

知乎 首发于 颜海镜的博客

```
  ages: ...,
  brands: ...,
}
```

此数据需要在 Products 组件中进行维护。**因为 Products 组件的子组件 Filters 和 ProductResults 都将依赖这项数据状态。**

Filters 组件通过 prop 接收 filterSelections 状态，并拆解传递给它的三项筛选子组件：

```
class Filters extends React.Component {
  render() {
    return (
      <div>
        <PriceFilter price={this.props.filterSelections.price} />
        <AgeFilter ages={this.props.filterSelections.ages} />
        <BrandFilter brands={this.props.filterSelections.brands} />
      </div>
    );
  };
}
```

同样地，ProductResults 组件也通过 prop 接收 filterSelections 状态，进行相应产品的展示。

对于 Filters 组件，它一定不仅仅是接收 filterSelections 数据而已，同样也需要对此项数据进行更新。为此，我们在 Products 组件中设计相应的 handler 函数，对过滤信息进行更新，命名为 updateFilters，并将此处理函数作为 prop 下发给 Filters 组件：

```
class Products extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      filterSelections: {
        price: someInitialValue,
        ages: someInitialValue,
        brands: someInitialValue,
      }
    }
  }

  updateFilters = (newSelections) => {
```

```
  };

  render() {
    return(
      <div>
        <Filters
          filterSelections={this.state.filterSelections}
          selectionsChanged={this.updateFilters}
        />
        <Products filterSelections={this.state.filterSelections} />
      </div>
    );
  }
}
```

注意这里我们对 this 绑定方式。有兴趣的读者可以参考我的另一篇文章：从 React 绑定 this，看 JS 语言发展和框架设计。

作为 Filters 组件，同样也要对处理函数进行进一步拆分和分发：

```
class Filters extends React.Component {
  updatePriceFilter = (newValue) => {
    this.props.selectionsChanged({
      ...this.props.filterSelections,
      price: newValue
    })
  };

  updateAgeFilter = (newValue) => {
    this.props.selectionsChanged({
      ...this.props.filterSelections,
      ages: newValue
    })
  };

  updateBrandFilter = (newValue) => {
    this.props.selectionsChanged({
      ...this.props.filterSelections,
      brands: newValue
    })
  };
```

知乎 首发于 颜海镜的博客

```
          <PriceFilter
            price={this.props.filterSelections.price}
            priceChanged={this.updatePriceFilter}
          />
          <AgeFilter
            ages={this.props.filterSelections.ages}
            agesChanged={this.updateAgeFilter}
          />
          <BrandFilter
            brands={this.props.filterSelections.brands}
            brandsChanged={this.updateBrandFilter}
          />
        </div>
      );
    };
  }
```

我们根据 selectionsChanged 函数，通过传递不同类型参数，设计出 updatePriceFilter、updateAgeFilter、updateBrandFilter 三个方法，分别传递给 PriceFilter、AgeFilter、BrandFilter 三个组件。

这样的做法非常直接，然而运行良好。但是在 Filters 组件中，多了很多函数，且这些函数看上去做着相同的逻辑。如果将来又多出了一个或多个过滤条件，那么同样也要多出同等数量的“双胞胎”函数。这显然不够优雅。

currying 是什么

在分析更加优雅的解决方案之前，我们先简要了解一下 curry 化是什么。curry 化事实上是一种变形，它将一个函数 f 变形为 f'，f' 的参数接收原本函数 f 的参数，同时返回一个新的函数 f''，f'' 接收剩余的参数并返回函数 f 的计算结果。

这么描述无疑是抽象的，我们还是通过代码来理解。这是一个简单的求和函数：

```
add = (x, y) => x + y;
```

curried 之后：

```
}
```

所以，当执行 curriedAdd(1)(2) 之后，得到结果 3，curriedAdd(x) 函数有一个名字叫 partial application，curriedAdd 函数只需要原本 add(X, y) 函数的一部分参数。

> Currying a regular function let’ s us perform partial application on it.

简单了解了这些内容，我们看看 curry 化如何在上述场景中应用。

curry 化应用

再回到之前的场景，我们设计 curry 化函数：updateSelections，

```
updateSelections = (selectionType) => {
  return (newValue) => {
    this.props.selectionsChanged({
      ...this.props.filterSelections,
      [selectionType]: newValue,
    });
  }
};
```

进一步可以简化为：

```
updateSelections = (selectionType) => (newValue) => {
   this.props.selectionsChanged({
      ...this.props.filterSelections,
      [selectionType]: newValue,
   })
};
```

对于 updateSelections 的偏应用（即上面提到的 partial application）：

```
updateSelections('ages');
updateSelections('brands');
updateSelections('price');
```

知乎 首发于 颜海镜的博客

```
  updateSelections = (selectionType) => {
    return (newValue) => {
      this.props.selectionsChanged({
        ...this.props.selections,
        [selectionType]: newValue,  // new ES6 Syntax!! :)
      });
    }
  };

  render() {
    return (
      <div>
        <PriceFilter
          price={this.props.selections.price}
          priceChanged={this.updateSelections('price')}
        />
        <AgeFilter
          ages={this.props.selections.ages}
          agesChanged={this.updateSelections('ages')}
        />
        <BrandFilter
          brands={this.props.selections.brands}
          brandsChanged={this.updateSelections('brands')}
        />
      </div>
    );
  };
}
```

当然，**currying 并不是解决上述问题的唯一方案**。我们再来了解一种方法，进行对比消化，updateSelections 函数 uncurried 版本：

```
updateSelections = (selectionType, newValue) => {
  this.props.updateFilters({
    ...this.props.filterSelections,
    [selectionType]: newValue,
  });
}
```

这样的设计使得每一个 Filter 组件：PriceFilter、AgeFilter、BrandFilter 都要调用

```
  updateSelections = (selectionType, newValue) => {
    this.props.selectionsChanged({
      ...this.props.filterSelections,
      [selectionType]: newValue,
    });
  };

  render() {
    return (
      <div>
        <PriceFilter
          price={this.props.selections.price}
          priceChanged={(value) => this.updateSelections('price', value)}
        />
        <AgeFilter
          ages={this.props.selections.ages}
          agesChanged={(value) => this.updateSelections('ages', value)}
        />
        <BrandFilter
          brands={this.props.selections.brands}
          brandsChanged={(value) => this.updateSelections('brands', value)}
        />
      </div>
    );
  };
}
```

其实我认为，在这种场景下，关于两种方案的选择，可以根据开发者的偏好来决定。

总结

这篇文章内容较为基础，但从细节入手，展现了 React 开发编写和函数式理念相结合的魅力。文章译自这里，部分内容有所改动。

广告时间：
如果你对前端发展，尤其对 React 技术栈感兴趣：我的新书中，也许有你想看到的内容。关注作者 Lucas HC，新书出版将会有送书活动。

I

我的其他几篇关于React技术栈的文章：

从setState promise化的探讨 体会React团队设计思想

从setState promise化的探讨 体会React团队设计思想

通过实例，学习编写 React 组件的“最佳实践”

React 组件设计和分解思考

从 React 绑定 this，看 JS 语言发展和框架设计

做出Uber移动网页版还不够 极致性能打造才见真章**

React+Redux打造“NEWS EARLY”单页应用 一个项目理解最前沿技术栈真谛

编辑于 2018-04-18 06:57

前端工程师 前端开发 React

在 render 里引用对应的更新方法，正如「初级」实现中所做的那样，而不是每次 render 都生成一模一样的方法。

2018-04-19 回复 2

Lucas HC 作者

赞！inline functions 做法对性能影响确实一直在提，有的也再用 class 箭头方法再处理。jsx no lambda 我学习了解一下。

2018-04-19 回复 喜欢

知乎用户j6w4Nc

父组件每次 render 时执行 this.updateSelections('ages') 传输 props，会导致子组件对应 props 的引用变化，「===」的判断就不成立了，从而导致自组件无谓的 rerender 伤害性能。

另外 jsx-no-lambda 并不是站得住的 lint 规则，参考：cdb.reacttraining.com/r...

2018-04-19 回复 1

Cacivy

直接bind就行吧，this.updateSelections.bind(null, 'ages')

2018-04-18 回复 1

Lucas HC 作者

嗯 bind 是 curry 的代言人

2018-04-18 回复 喜欢

丁丁丁丁丁大树

curry过后，props上面的函数都是匿名的，组件在shouldComponentUpdate时，对比props，nextProps上的这个函数会return true，所以还需要在shouldComponentUpdate对这种情况做处理，避免不必要的更新

2018-04-18 回复 1

黄努努

代码写多了大家在最佳实践方面都会有一些共鸣，但是能把最佳实践总结好并写出这么一篇文章，真实令人佩服！！老师厉害

2018-04-18 回复 1

颜海镜

不明觉厉，期待新书，敢不敢把评论的筛选关了，ヘ(ˇ ^ ˇ)ヘ

2018-04-18 回复 1

Lucas HC 作者

不敢

2018-04-18 回复 喜欢

知乎 首发于 **颜海镜的博客**

updateSelections 函数里去判断 value 的类型，如果是对象就再取对象的属性？ 如果是这种情况，用文中刚开头的多写几个“双胞胎”函数会不会更好点？

2019-07-20 回复 喜欢

Lucas HC 作者

uncurried 可以设置成三个独立的函数。所以组件只要调用各自对应的函数就行了。不需要感知 type 类型，不需要添加参数。事件处理函数的话，合成事件对象和其他信息参数分别作为不同的参数次序。

2019-07-22 回复 喜欢

r怪兽

你好请问，“这样的设计使得每一个 Filter 组件：PriceFilter、AgeFilter、BrandFilter 都要调用 updateSelections 函数本身，并且要求组件本身感知 filterSelections 的属性名，以进行相应属性的更新。这就是一种耦合”指的仅仅是第二种 uncurry 的方式吗？如果是的话，为什么说 “要求组件本身感知 filterSelections 的属性名，以进行相应属性的更新”，属性名不是统一在 Filter 组件里面指定的吗？

2019-07-20 回复 喜欢

Lucas HC 作者

curry 化有这个情况。因为函数 curry 化之后 第一个参数 要传，表明 filter 哪种类型。这个类型就需要函数调用时感知。

2019-07-20 回复 喜欢

justjavac

这个链接让我醉了，link.jianshu.com/?... 我做了三次转码才把还原了

2018-04-18 回复 喜欢

justjavac ▸ **颜海镜**

循环重定向或者30x太多可能会导致被警告，或者被标记为有风险

2018-04-18 回复 1

justjavac ▸ **Lucas HC**

鸡汤太多

2018-04-18 回复 喜欢

查看全部 6 条回复 ›

蒋正

在 lint 里开启 jsx-no-lambda，天然就写成这样了。

2018-04-18 回复 喜欢

欢迎参与讨论

知乎 首发于 **颜海镜的博客**

颜海镜的博客
专注Web前端

推荐阅读

React 16 架构研究记录（文末有彩蛋

方正

面试系列之三：你真的了解React吗（中）组件间的通信...

本系列文章旨在完善你可能忽略的React相关的知识点，通过回答以下问题： React解决了什么问题如何设计一个好的组件？组件的Render函数在何时被调用？调用时DOM就一定会被更新吗？ 组件的生...

李熠 发表于技术圆桌

44 个 React 前端面试问题

素颜

解剖react组件的多种写法与演进

sessionboy