React 设计思想 https://github.com/react-guide/react-...

React 设计思想

> 译者序：本文是 React 核心开发者、有 React API 终结者之称的 Sebastian Markbåge 撰写，阐述了他设计 React 的初衷。阅读此文，你能站在更高的高度思考 React 的过去、现在和未来。原文地址：https://github.com/reactjs/react-basic

我写此文是想正式地阐述我心中 React 的心智模型。目的是解释为什么我们会这样设计 React，同时你也可以根据这些论点反推出 React。

不可否认，此文中的部分论据或前提尚存争议，而且部分示例的设计可能存在 bug 或疏忽。这只是正式确定它的最初阶段。如果你有更好的完善它的想法可以随时提交 pull request。本文不会介绍框架细节中的奇技淫巧，相信这样能提纲挈领，让你看清 React 由简单到复杂的设计过程。

React.js 的真实实现中充满了具体问题的解决方案，渐进式的解法，算法优化，历史遗留代码，debug 工具以及其他一些可以让它真的具有高可用性的内容。这些代码可能并不稳定，因为未来浏览器的变化和功能权重的变化随时面临改变。所以具体的代码很难彻底解释清楚。

我偏向于选择一种我能完全 hold 住的简洁的心智模型来作介绍。

变换（Transformation）

设计 React 的核心前提是认为 UI 只是把数据通过映射关系变换成另一种形式的数据。同样的输入必会有同样的输出。这恰好就是纯函数。

```
function NameBox(name) {
  return { fontWeight: 'bold', labelContent: name };
}
```

```
'Sebastian Markbåge' ->
{ fontWeight: 'bold', labelContent: 'Sebastian Markbåge' };
```

抽象（Abstraction）

你不可能仅用一个函数就能实现复杂的 UI。重要的是，你需要把 UI 抽象成多个隐藏内部细节，又可复用的函数。通过在一个函数中调用另一个函数来实现复杂的 UI，这就是抽象。

```
function FancyUserBox(user) {
  return {
```

```
    borderStyle: '1px solid blue',
    childContent: [
      'Name: ',
      NameBox(user.firstName + ' ' + user.lastName)
    ]
  };
}
```

```
{ firstName: 'Sebastian', lastName: 'Markbåge' } ->
{
  borderStyle: '1px solid blue',
  childContent: [
    'Name: ',
    { fontWeight: 'bold', labelContent: 'Sebastian Markbåge' }
  ]
};
```

组合（Composition）

为了真正达到重用的特性，只重用叶子然后每次都为他们创建一个新的容器是不够的。你还需要可以包含其他抽象的容器再次进行组合。我理解的“组合”就是将两个或者多个不同的抽象合并为一个。

```
function FancyBox(children) {
  return {
    borderStyle: '1px solid blue',
    children: children
  };
}

function UserBox(user) {
  return FancyBox([
    'Name: ',
```

```
    NameBox(user.firstName + ' ' + user.lastName)
  ]);
}
```

状态（State）

UI 不单单是对服务器端或业务逻辑状态的复制。实际上还有很多状态是针对具体的渲染目标。举个例子，在一个 text field 中打字。它不一定要复制到其他页面或者你的手机设备。滚动位置这个状态是一个典型的你几乎不会复制到多个渲染目标的。

我们倾向于使用不可变的数据模型。我们把可以改变 state 的函数串联起来作为原点放置在顶层。

```
function FancyNameBox(user, likes, onClick) {
  return FancyBox([
    'Name: ', NameBox(user.firstName + ' ' + user.lastName),
    'Likes: ', LikeBox(likes),
    LikeButton(onClick)
  ]);
}

// 实现细节

var likes = 0;
function addOneMoreLike() {
  likes++;
  rerender();
}

// 初始化

FancyNameBox(
  { firstName: 'Sebastian', lastName: 'Markbåge' },
  likes,
  addOneMoreLike
);
```

注意：本例更新状态时会带来副作用（addOneMoreLike 函数中）。我实际的想法是当一个"update"传入时我们返回下一个版本的状态，但那样会比较复杂。此示例待更新

Memoization

对于纯函数，使用相同的参数一次次调用未免太浪费资源。我们可以创建一个函数的 memorized 版本，用来追踪最后一个参数和结果。这样如果我们继续使用同样的值，就不需要反复执行它了。

```
function memoize(fn) {
  var cachedArg;
  var cachedResult;
  return function(arg) {
    if (cachedArg === arg) {
      return cachedResult;
    }
    cachedArg = arg;
    cachedResult = fn(arg);
    return cachedResult;
  };
}

var MemoizedNameBox = memoize(NameBox);

function NameAndAgeBox(user, currentTime) {
  return FancyBox([
    'Name: ',
    MemoizedNameBox(user.firstName + ' ' + user.lastName),
    'Age in milliseconds: ',
    currentTime - user.dateOfBirth
  ]);
}
```

列表（Lists）

大部分 UI 都是展示列表数据中不同 item 的列表结构。这是一个天然的层级。

为了管理列表中的每一个 item 的 state ，我们可以创造一个 Map 容纳具体 item 的 state。

```
function UserList(users, likesPerUser, updateUserLikes) {
  return users.map(user => FancyNameBox(
    user,
    likesPerUser.get(user.id),
    () => updateUserLikes(user.id, likesPerUser.get(user.id) + 1)
  ));
}

var likesPerUser = new Map();
function updateUserLikes(id, likeCount) {
  likesPerUser.set(id, likeCount);
  rerender();
}

UserList(data.users, likesPerUser, updateUserLikes);
```

注意：现在我们向 FancyNameBox 传了多个不同的参数。这打破了我们的 memoization 因为我们每次只能存储一个值。更多相关内容在下面。

连续性（Continuations）

不幸的是，自从 UI 中有太多的列表，明确的管理就需要大量的重复性样板代码。

我们可以通过推迟一些函数的执行，进而把一些模板移出业务逻辑。比如，使用“柯里化”（JavaScript 中的 `bind`）。然后我们可以从核心的函数外面传递 state，这样就没有样板代码了。

下面这样并没有减少样板代码，但至少把它从关键业务逻辑中剥离。

```
function FancyUserList(users) {
  return FancyBox(
    UserList.bind(null, users)
  );
}

const box = FancyUserList(data.users);
const resolvedChildren = box.children(likesPerUser, updateUserLikes);
const resolvedBox = {
  ...box,
  children: resolvedChildren
};
```

State Map

之前我们知道可以使用组合避免重复执行相同的东西这样一种重复模式。我们可以把执行和传递 state 逻辑挪动到被复用很多的低层级的函数中去。

```
function FancyBoxWithState(
  children,
  stateMap,
  updateState
) {
  return FancyBox(
    children.map(child => child.continuation(
      stateMap.get(child.key),
      updateState
    ))
  );
}
```

```
function UserList(users) {
  return users.map(user => {
    continuation: FancyNameBox.bind(null, user),
    key: user.id
  });
}

function FancyUserList(users) {
  return FancyBoxWithState.bind(null,
    UserList(users)
  );
}

const continuation = FancyUserList(data.users);
continuation(likesPerUser, updateUserLikes);
```

Memoization Map

一旦我们想在一个 memoization 列表中 memoize 多个 item 就会变得很困难。因为你需要制定复杂的缓存算法来平衡调用频率和内存占有率。

还好 UI 在同一个位置会相对的稳定。相同的位置一般每次都会接受相同的参数。这样以来，使用一个集合来做 memoization 是一个非常好用的策略。

我们可以用对待 state 同样的方式，在组合的函数中传递一个 memoization 缓存。

```
function memoize(fn) {
  return function(arg, memoizationCache) {
    if (memoizationCache.arg === arg) {
      return memoizationCache.result;
    }
    const result = fn(arg);
    memoizationCache.arg = arg;
    memoizationCache.result = result;
```

```
    return result;
  };
}

function FancyBoxWithState(
  children,
  stateMap,
  updateState,
  memoizationCache
) {
  return FancyBox(
    children.map(child => child.continuation(
      stateMap.get(child.key),
      updateState,
      memoizationCache.get(child.key)
    ))
  );
}

const MemoizedFancyNameBox = memoize(FancyNameBox);
```

代数效应（Algebraic Effects）

多层抽象需要共享琐碎数据时，一层层传递数据非常麻烦。如果能有一种方式可以在多层抽象中快捷地传递数据，同时又不需要牵涉到中间层级，那该有多好。React 中我们把它叫做“context”。

有时候数据依赖并不是严格按照抽象树自上而下进行。举个例子，在布局算法中，你需要在实现他们的位置之前了解子节点的大小。

现在，这个例子有一点超纲。我会使用 代数效应 这个由我发起的 ECMAScript 新特性提议。如果你对函数式编程很熟悉，它们在避免由 monad 强制引入的仪式一样的编码。

```
function ThemeBorderColorRequest() { }

function FancyBox(children) {
  const color = raise new ThemeBorderColorRequest();
  return {
    borderWidth: '1px',
    borderColor: color,
    children: children
  };
}

function BlueTheme(children) {
  return try {
    children();
  } catch effect ThemeBorderColorRequest -> [, continuation] {
    continuation('blue');
  }
}

function App(data) {
  return BlueTheme(
    FancyUserList.bind(null, data.users)
  );
}
```