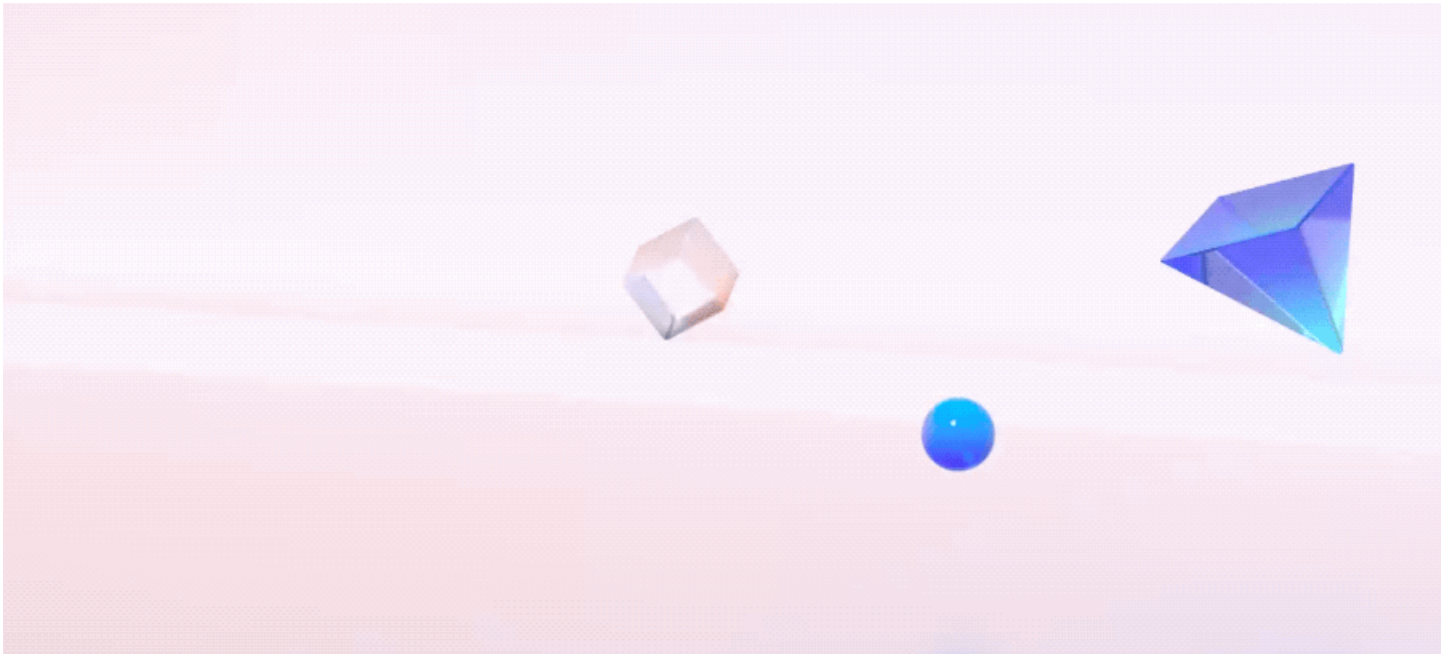


Becoming fully reactive: an in-depth explanation of MobX

https://medium.com/hackernoon/becoming-fully-reactive-an-in-depth-explanation-of-mobservable-55995262a254

Due to popular demand (and to have a cool story for my grand-children), these are the inner workings of MobX. A lot of people are surprised how consistent and fast MobX is. But rest assured, there is no magic in play!

由于大众的需求，这些是 MobX 的内部运作。很多人都对 MobX 的一致性和速度感到惊讶。但请放心，没有魔法在起作用！

First, let’ s define the core concepts of MobX:

首先，让我们定义 MobX 的核心概念

1 ***Observable state***. Any value that can be mutated and might serve as source for computed values is state. MobX can make most types of values (primitives, arrays, classes, objects, etc.) and even (potentially cyclic) references observable out of the box.

可观察状态。任何可以变异并可能用作计算值来源的值都是状态。 MobX 可以使大多数类型的值（基元、数组、类、对象等）甚至（可能是循环的）引用开箱即用。

2.***Computed values.*** Any value that can be computed by using a function that purely operates on other observable values. Computed values can range from the concatenation of a few strings up to deriving complex object graphs and visualizations. Because computed values are observable themselves, even the rendering of a complete user interface can be derived from the observable state. Computed values might evaluate either lazily or in reaction to state changes.

计算值。可以通过使用纯粹对其他可观察值进行操作的函数来计算的任何值。计算值的范围可以从几个字符串的串联到派生复杂的对象图和可视化。因为计算值本身是可观察的，所以即使是完整的用户界面的渲染也可以从可观察状态派生出来。计算值可能会延迟评估或响应状态更改。

3.***Reactions.*** A reaction is a bit similar to a computed value, but instead of producing a new value it produces a side effect. Reactions bridge reactive and imperative programming for things like printing to the console, making network requests, incrementally updating the React component tree to patch the DOM, etc.

反应。反应有点类似于计算值，但它不会产生新值，而是产生副作用。反应桥接反应式和命令式编程，例如打印到控制台、发出网络请求、增量更新 React 组件树以修补 DOM 等。

4.***Actions.*** Actions are the primary means to modify the state. Actions are not a reaction to state changes but take sources of change, like user events or incoming web-socket connections, to modify the observable state.

行动。动作是修改状态的主要手段。动作不是对状态变化的反应，而是利用变化源（如用户事件或传入的 Web 套接字连接）来修改可观察状态。

Computed values and *reactions* are both referred to as *derivations* in the remainder of this blog-post. So far, this might all sound a bit academic so let’ s make it concrete! In a spreadsheet all data cells that have values would form the *observable state*. Formulas and charts are *computed values* that can be derived from the data cells and other formulas. Drawing the output of a data cell or a formula on the screen is a *reaction*. Changing a data cell or formula is an *action*.

在这篇博文的其余部分中，计算值和反应都被称为推导。到目前为止，这听起来可能有点学术性，所以让我们把它具体化！在电子表格中，所有具有值的数据单元格都将形成可观察状态。公式和图表是可以从数据单元格和其他公式中得出的计算值。在屏幕上绘制数据单元格或公式的输出是一种反应。更改数据单元格或公式是一项操作。

Anyway, here are all four concepts in a small example that uses MobX and React:

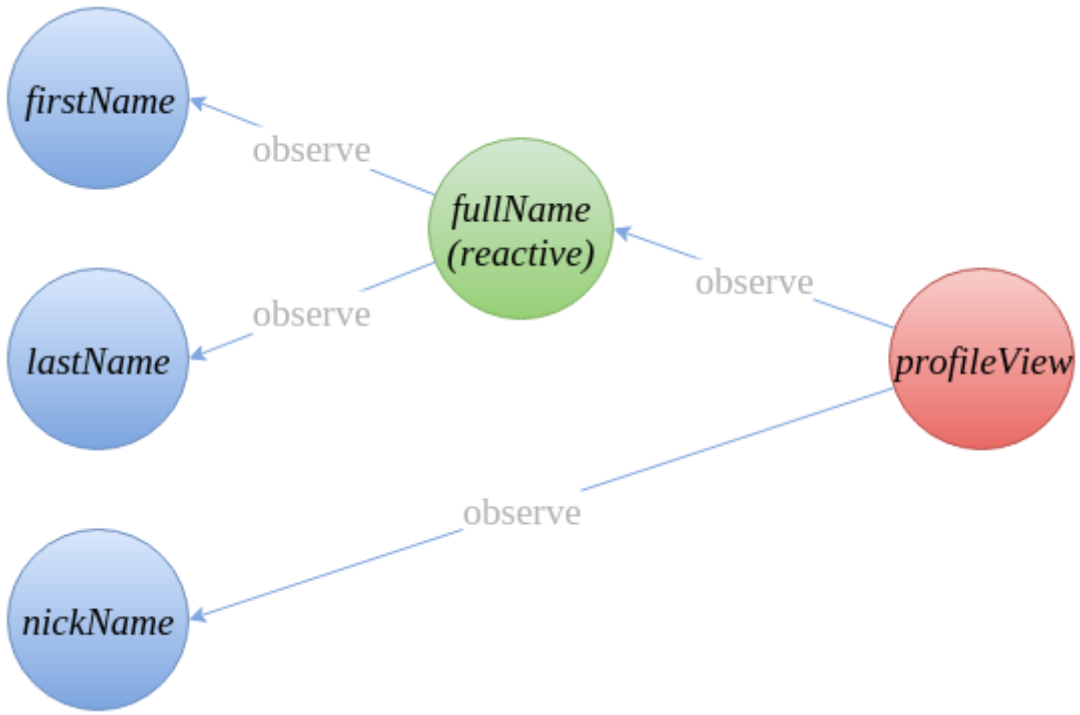
不管怎样，下面是一个使用 MobX 和 React 的小例子中的所有四个概念：

```
1 class Person {
2   @observable firstName = "Michel";
```

```
  @observable lastName = "Weststrate";
  @observable nickName;

  @computed get fullName() {
    return this.firstName + " " + this.lastName;
  }
}

const michel = new Person();

// Reaction: log the profile info whenever it changes
autorun(() => console.log(person.nickName ? person.nickName : person.fullName));

// Example React component that observes state
const profileView = observer(props => {
  if (props.person.nickName)
    return <div>{props.person.nickName}</div>
  else
    return <div>{props.person.fullName}</div>
});

// Action:
setTimeout(() => michel.nickName = "mweststrate", 5000)

React.render(React.createElement(profileView, { person: michel }), document.body)
// This snippet is runnable in jsfiddle: https://jsfiddle.net/mweststrate/049r6jo
```

We could draw a dependency tree based on the above listing. Intuitively it will look as follows:

我们可以根据上面的清单绘制一个依赖树。直观地说，它将如下所示：

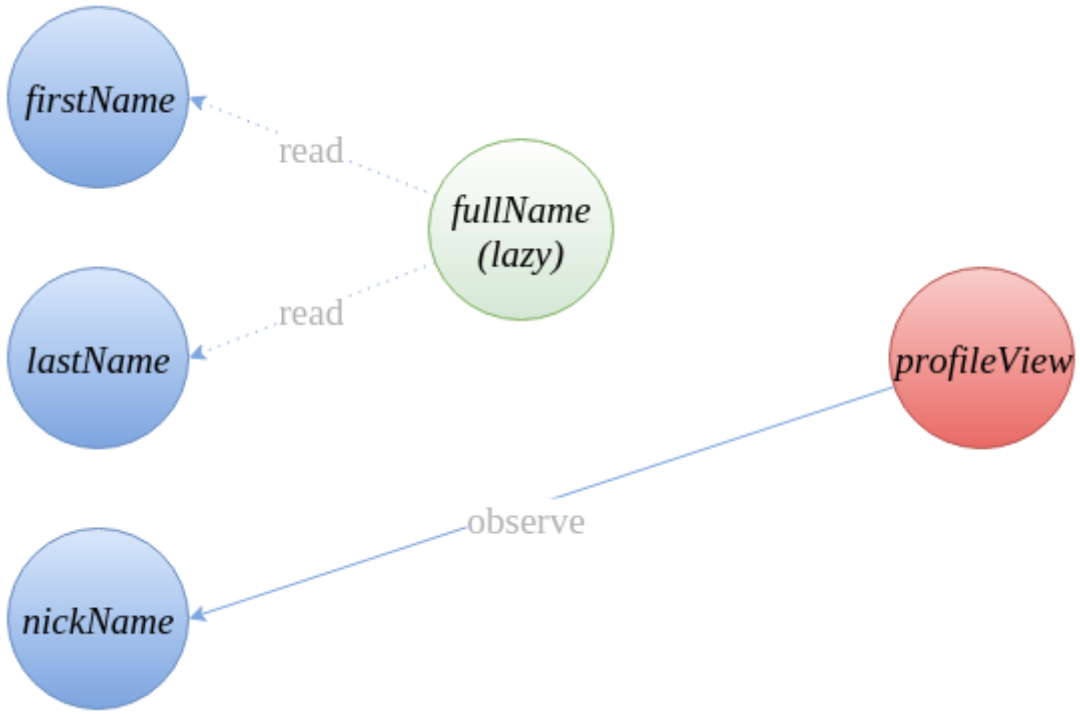


The *state* of this applications is captured in the *observable* properties (blue). The green *computed value* fullName can be derived from the state automatically by observing the firstName and the lastName. Similarly the rendering of the profileView can be derived from the nickName and the fullName. The profileView will react to state changes by producing a side effect: it updates the React component tree.

此应用程序的状态在可观察属性（蓝色）中捕获。绿色计算值 fullName 可以通过观察 firstName 和 lastName 自动从状态导出。类似地，profileView 的呈现可以从 nickName 和 fullName 派生。profileView 将通过产生副作用来响应状态更改：它更新 React 组件树。

When using MobX the dependency tree is minimally defined. For example, as soon as the person being rendered has a nickname, the rendering will no longer be affected by the output of the fullName value, nor the first- or lastName (see listing 1). All observer relations between those values can be cleaned up and MobX will automatically simplify the dependency tree accordingly:

使用 MobX 时，依赖关系树的定义最少。例如，一旦被渲染的人有了昵称，渲染将不再受 fullName 值输出的影响，也不会受到 first- 或 lastName 的影响（参见清单 1）。这些值之间的所有观察者关系都可以清除，MobX 会自动相应地简化依赖树：



MobX will always try to minimize the number of computations that are needed to produce a consistent state. In the rest of this blog post, I will describe several strategies used to achieve this goal. But before diving into the magic of how computed values and reactions are kept in sync with the state, let’ s first describe the principle behind MobX:

MobX 总是会尽量减少产生一致状态所需的计算数量。在这篇博文的其余部分，我将描述用于实现这一目标的几种策略。但在深入探讨计算值和反应如何与状态保持同步之前，让我们先描述一下 MobX 背后的原理：

Reacting to state changes is always better then acting on state changes.

对状态变化做出反应总是比对状态变化采取行动更好。

Automatically updating application state is an anti-pattern. Derive data instead

自动更新应用程序状态是一种反模式。改为派生数据

Any imperative action that an application takes in response to a state change usually creates or updates some values. In other words, most actions manage a local cache. Triggering the user interface to update? Updating aggregated values? Notifying the back-end? These can all be thought of as cache invalidations in disguise. To ensure these caches will stay in sync, you need to subscribe to future state changes that will enable your actions to be triggered again.

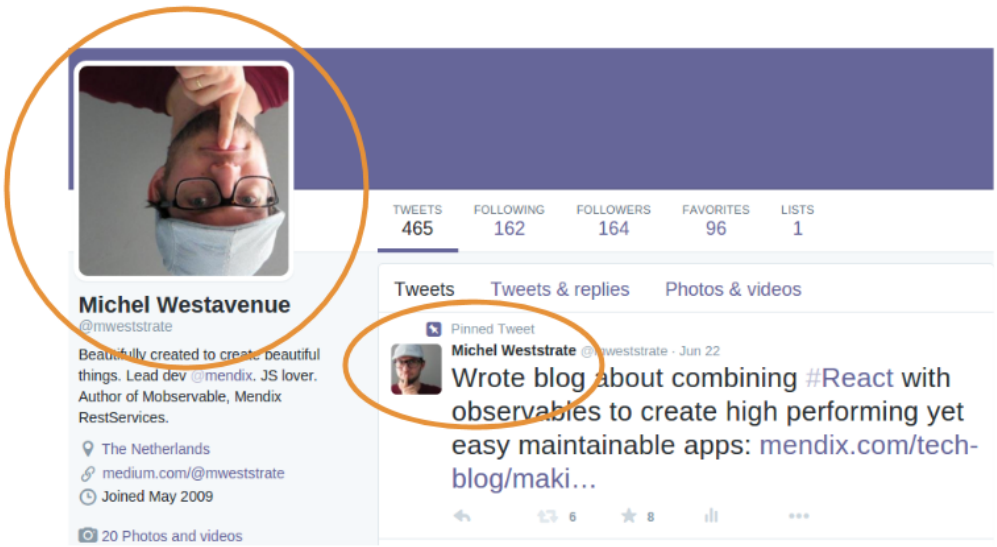
应用程序为响应状态更改而采取的任何命令性操作通常会创建或更新一些值。换句话说，大多数操作都管理本地缓存。触发用户界面更新？更新聚合值？通知后端？这些都可以被认为是变相的缓存失效。为确保这些缓存保持同步，您需要订阅未来的状态更改，以便再次触发您的操作。

But working with subscriptions (or cursors, lenses, selectors, connectors, etc) has a fundamental problem: as your app evolves, you will make mistakes in managing those subscriptions and either oversubscribe (continue subscribing to a value or store that is no longer used in a component) or undersubscribe (forgetting to listen for updates leading to subtle staleness bugs).

但是使用订阅（或光标、镜头、选择器、连接器等）存在一个根本问题：随着应用程序的发展，您将在管理这些订阅时出错，并且要么超额订阅（继续订阅不再使用的值或存储）在组件中）或订阅不足（忘记收听导致细微陈旧错误的更新）

In other words; when using manual subscriptions, your app will eventually be inconsistent.

换句话说;使用手动订阅时，您的应用最终会出现不一致的情况。



The above image is a nice example of the Twitter UI being inconsistent. As explained in my Reactive2015 talk, there can only be two causes for this: Either there is no subscription that tells tweets to re-render if the profile of the associated author has changed. Or the data was normalized and the author of a tweet doesn’ t even relate to the profile of the currently logged-in user, despite the fact that both pieces of data try to describe the same properties of the same person.

上图是 Twitter UI 不一致的一个很好的例子。正如我在 Reactive2015 演讲中所解释的，这只有两个原因：如果相关作者的个人资料发生变化，则没有订阅告诉推文重新呈现。或者数据被标准化，推文的作者甚至与当前登录用户的个人资料无关，尽管这两条数据都试图描述同一个人的相同属性。

Coarse grained subscriptions like Flux-style store subscriptions are very susceptible to oversubscribing. When using React, you can simply tell whether your components are oversubscribing by printing wasted renderings. MobX will reduce this number to zero. The idea is simple yet counterintuitive: More subscriptions result in fewer recomputations. MobX manages many thousands of observers for you. You can effectively tradeoff memory for CPU cycles.

像 Flux 风格的商店订阅这样的粗粒度订阅非常容易受到超额订阅的影响。使用 React 时，您可以通过打印浪费的渲染来简单地判断您的组件是否超额订阅。 MobX 会将这个数字减少到零。这个想法很简单但违反直觉：更多的订阅导致更少的重新计算。 MobX 为您管理成千上万的观察者。您可以有效地权衡内存以换取 CPU 周期。

Note that oversubscribing also exists in very subtle forms. If you subscribe to data that is used, but not under *all* conditions, you are still oversubscribing. For example, if the profileView component subscribes to the fullName of a person that has a nickName, it is oversubscribing (see listing 1). So an important principle behind the design MobX is:

请注意，超额订阅也以非常微妙的形式存在。如果您订阅了已使用的数据，但并非在所有情况下，您仍然超额订阅。例如，如果 profileView 组件订阅了一个拥有 nickName 的人的 fullName，那么它就是超额订阅（参见清单 1）。所以 MobX 设计背后的一个重要原则是：

A minimal, consistent set of subscriptions can only be achieved if subscriptions are determined at run-time.

只有在运行时确定订阅时，才能实现最小的、一致的订阅集

The second important idea behind MobX is that for any app that is more complex than TodoMVC, you will often need a data graph, instead of a normalized tree, to store the state in a mentally manageable yet optimal way. Graphs enable referential consistency and avoid data duplication so that it can be guaranteed that derived values are never stale.

MobX 背后的第二个重要理念是，对于任何比 TodoMVC 更复杂的应用程序，您通常需要一个数据图而不是规范化树，以一种易于管理但最佳的方式存储状态。图实现了引用一致性并避免了数据重复，因此可以保证派生值永远不会过时。

How MobX keeps all derivations efficiently in a consistent state

MobX 如何有效地将所有派生保持在一致的状态

The solution: don’t cache, derive instead. People ask: “isn’t that extremely expensive?” No, it is actually very efficient! The reason for that is, as explained above: MobX doesn’t run all derivations, but ensures that only *computed values* that are involved in some *reaction* are kept in sync with the observable state. Those derivations are called to be *reactive*. To draw the parallel with spreadsheets again: only those formulas that are currently visible or that are used indirectly by a visible formula, need to re-compute when one of the observed data cells change.

解决方案：不要缓存，而是派生。人们问：“那不是很贵吗？”不，它实际上非常有效！原因是，如上所述：MobX 不会运行所有派生，而是确保只有参与某些反应的计算值与可观察状态保持同步。这些推导被称为反应式。再次与电子表格进行比较：只有那些当前可见或被可见公式间接使用的公式，需要在观察到的数据单元格之一发生变化时重新计算。

Lazy versus reactive evaluation

惰性与反应性评估

So what about computations that aren’t used directly or indirectly by a reaction? You can still inspect the value of a computed value like fullName at any time. The solution is simple: if a computed value is not reactive, it will be evaluated on demand (lazily), just like a normal getter function. Lazy derivations (which never observe anything) can simply be garbage collected if they run out of scope. Remember that computed values should always be *pure functions* of the observable app state? This is the reason why: For pure functions it doesn’t matter whether they are evaluated lazily or eagerly; the evaluation of the function always yields the same result given the same observable state.

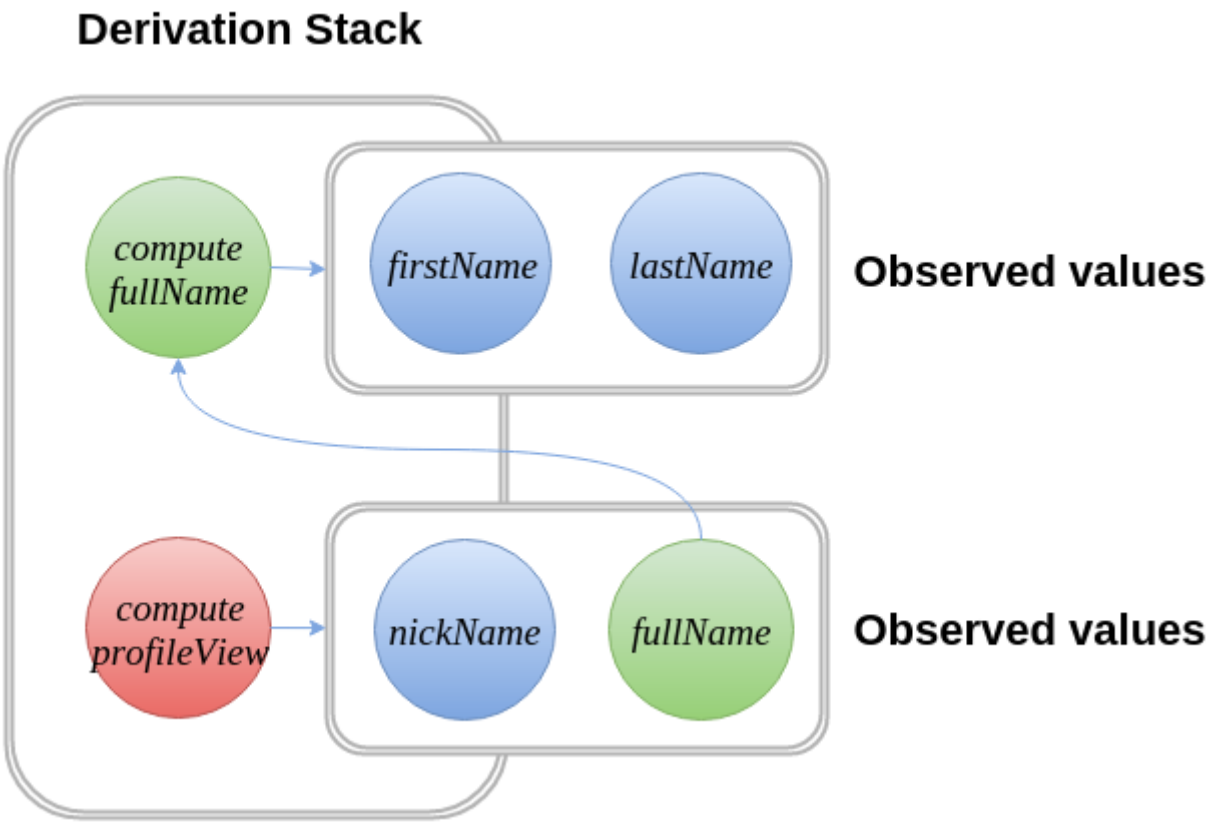
那么反应不直接或间接使用的计算呢？您仍然可以随时检查计算值（如 fullName）的值。解决方案很简单：如果计算值不是响应式的，它将按需（惰性）求值，就像普通的 getter 函数一样。惰性派生（从不观察任何东西）如果超出范围，可以简单地被垃圾收集。还记得计算值应该始终是可观察应用程序状态的纯函数吗？这就是为什么：对于纯函数，无论是惰性求值还是急切求值都无关紧要；给定相同的可观察状态，函数的评估总是产生相同的结果。

Running computations

运行计算

Reactions and computed values are both run by MobX in the same manner. When a recomputation is triggered the function is pushed onto the *derivation stack*; a function stack of currently running derivations. As long as a computation is running, every observable that is accessed will register itself as a dependency of the topmost function of the derivation stack. If the value of a computed value is needed, the value can simply be the last known value if the computed value is already in the reactive state. And otherwise it will push itself on the derivation-stack, switch to reactive mode and start computing as well.

MobX 以相同的方式运行反应和计算值。当触发重新计算时，函数被压入派生堆栈；当前运行的派生的函数堆栈。只要计算正在运行，每个被访问的 observable 都会将自己注册为派生堆栈最顶层函数的依赖项。如果需要计算值的值，如果计算值已经处于反应状态，则该值可以简单地是最后一个已知值。否则它将把自己推到派生堆栈上，切换到反应模式并开始计算。

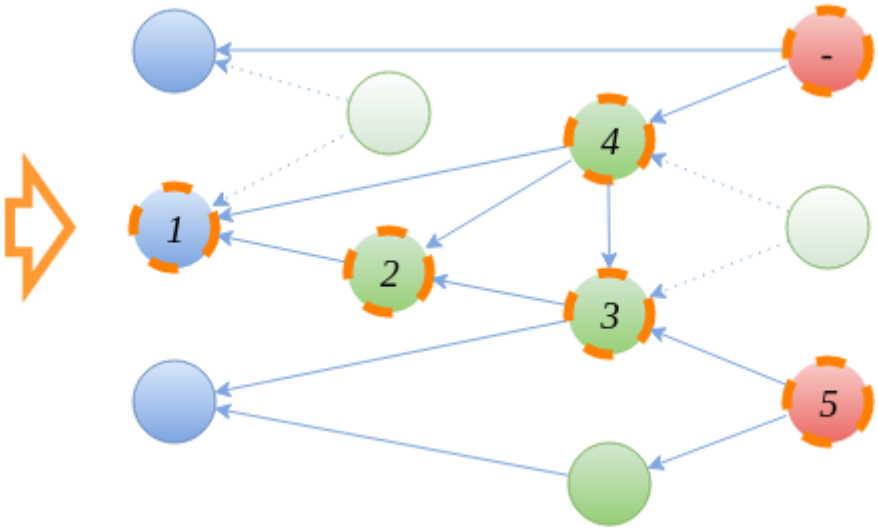


When a computation completes, it will have obtained a list of observables that were accessed during execution. In the profileView for example, this list will either just contain the nickName property, or the nickName and fullName properties. This list is diffed against the previous list of observables. Any removed items will be unobserved (computed values might go back from reactive to lazy mode at this point) and any added observables will be observed until the next computation. When the value of for example firstname is changed in the future, it knows that fullName needs to be recomputed. Which in turn will cause profile view to recomputed. The next paragraph explains this process in more detail.

当计算完成时，它将获得执行期间访问的可观察对象列表。例如，在 profileView 中，这个列表要么只包含 nickName 属性，要么包含 nickName 和 fullName 属性。这个列表与之前的 observables 列表不同。任何删除的项目都将不被观察（此时计算值可能会从反应模式返回到惰性模式），并且任何添加的可观察对象都将被观察到，直到下一次计算。当例如 firstname 的值在未来发生变化时，它知道 fullName 需要重新计算。这反过来将导致重新计算配置文件视图。下一段更详细地解释了这个过程。

Propagating state changes

传播状态变化



Derivations will react to state changes automatically. All reactions happen *synchronously* and more importantly *glitch-free*. When an observable value is modified the following algorithm is performed:

派生将自动对状态更改做出反应。所有反应同步发生，更重要的是无故障。当修改可观察值时，将执行以下算法：

1. The observable value sends a *stale notification* to all its observers to indicate that it has become stale. Any affected computed values will recursively pass on the notification to their observers. As a result, a part of the dependency tree will be marked as *stale*. In the example dependency tree of figure 5, the observers that will become stale when value ‘1’ is changed are marked with an orange, dashed border. These are all the derivations that *might* be affected by the changing value.
2. After sending the *stale notification* and storing the new value, a *ready notification* will be sent. This message also indicates whether the value did actually change.
3. As soon as a derivation has received a ready notification for every stale notification received in step 1, it knows that all the observed values are stable and it will start to recompute. Counting the number of ready / stale messages will ensure that, for example, computed value ‘4’ will only re-evaluate after computed value ‘3’ has become stable.

4. If none of the ready messages indicate that a value was changed, the derivation will simply tell its own observers that it is ready again, but without changing its value. Otherwise the computation will recompute and send a ready message to its own observers. This results in the order of execution as displayed in figure 5. Note that (for example) the last reaction (marked with '-') will never execute if computed value '4' did re-evaluate but didn't produce a new value.

1.可观察值向其所有观察者发送过时通知，以指示它已过时。任何受影响的计算值都会递归地将通知传递给它们的观察者。结果，依赖关系树的一部分将被标记为过时的。在图 5 的示例依赖树中，当值“1”更改时将变得陈旧的观察者用橙色虚线边框标记。这些都是可能受变化值影响的推导。
2.发送过时通知并存储新值后，将发送就绪通知。此消息还指示该值是否确实发生了变化。
3.一旦派生收到步骤 1 中收到的每个陈旧通知的就绪通知，它就知道所有观察到的值都是稳定的，它将开始重新计算。计算就绪/陈旧消息的数量将确保，例如，计算值“4”只会在计算值“3”变得稳定后重新评估。
4.如果没有任何就绪消息表明值已更改，则派生将简单地告诉其自己的观察者它再次准备好，但不更改其值。否则，计算将重新计算并向其自己的观察者发送就绪消息。这导致执行顺序如图 5 所示。请注意（例如）如果计算值“4”确实重新评估但没有产生新值，则最后一个反应（标有“-”）将永远不会执行.

The previous two paragraph summarize how dependencies between observable values and derivations are tracked at run-time and how changes are propagated through the derivations. At this point you might also realize that a *reaction* is basically a *computed value* that is always in *reactive* mode. It is important to realize that this algorithm can be implemented very efficiently without closures and just with a bunch of pointer arrays. Additionally, MobX applies a number of other optimizations which are beyond the scope of this blog post.

前两段总结了在运行时如何跟踪可观察值和派生之间的依赖关系，以及如何通过派生传播更改。此时您可能还意识到，反应基本上是一个始终处于反应模式的计算值。重要的是要意识到这个算法可以在没有闭包的情况下非常有效地实现，并且只需要一堆指针数组。此外，MobX 还应用了许多其他优化，这些优化超出了本文的范围。

Synchronous execution

同步执行

People are often surprised that MobX runs everything synchronously (like RxJs and unlike knockout). This has two big advantages: First it becomes simply impossible to ever observe stale derivations. So a derived value can be used immediately after changing a value that influences it. Secondly it makes stack-traces and debugging easier as it avoids the useless stack-traces that are typical to Promise / async libraries.

人们常常对 MobX 同步运行一切感到惊讶（就像 RxJs 和不一样的淘汰赛）。这有两大优势：首先，根本不可能观察到陈旧的推导。因此，可以在更改影响它的值后立即使用派生值。其次，它使堆栈跟踪

和调试更容易，因为它避免了 Promise / async 库典型的无用堆栈跟踪

```
transaction(() => {
  michel.firstName = "Mich";
  michel.lastName = "W.";
});
```

However, synchronous execution also introduces the need for transactions. If several mutations are applied in immediate succession, it is preferable to re-evaluate all derivations after all changes has been applied. Wrapping an action in a *transaction* block achieves this. Transactions simply postpone all *ready notifications* until the transaction block has completed. Note that transactions still runs and updates everything synchronously.

然而，同步执行也引入了对事务的需求。如果立即连续应用多个突变，则最好在应用所有更改后重新评估所有派生。在事务块中包装一个动作可以实现这一点。事务只是推迟所有就绪通知，直到事务块完成。请注意，事务仍会同步运行并更新所有内容。

That summarizes the most essential implementation details of MobX. We haven’ t covered everything yet, but it is good to know for example that you can compose computed values. By *composing reactive computations* it is even possible to automatically transform one graph of data into another graph of data and keep this derivation up to date with the minimum number of patches. This makes it easy to implement complex patterns like map-reduce, state tracking using immutable shared data, or sideways data loading. But more on that in a next blog post.

这总结了 MobX 最重要的实现细节。我们还没有涵盖所有内容，但是很高兴知道例如您可以组合计算值。通过组合反应式计算，甚至可以自动将一个数据图转换为另一个数据图，并以最少的补丁数量使这种推导保持最新。这使得实现复杂模式变得容易，例如 map-reduce、使用不可变共享数据的状态跟踪或横向数据加载。但在下一篇博客文章中会详细介绍。

1. The application state of complex applications can best be expressed using *graphs* to achieve referential consistency and stay close to the mental model of a problem domain.
2. One should not imperatively act on state changes by using manually defined subscriptions or cursors. This will inevitably lead to bugs as a result of under- or oversubscribing.
3. Use *runtime analysis* to determine the *smallest possible set* of observer → observable relationships. This leads to a computational model where it can be guaranteed that the minimum amount of derivations are run without ever observing a stale value.
4. Any derivation that is not needed to achieve an active side effect can be optimized away completely.

1.复杂应用程序的应用程序状态可以最好地使用图来表示，以实现引用一致性并保持接近问题域的心理模型。
2.不应通过使用手动定义的订阅或游标来强制执行状态更改。由于订阅不足或过度订阅，这将不可避免地导致错误。
3.使用运行时分析来确定最小可能的观察者→可观察关系集。这导致了一个计算模型，在该模型中可以保证在没有观察到陈旧值的情况下运行最小数量的推导。
4。任何不需要实现积极副作用的推导都可以完全优化掉。