The fundamental principles behind MobX

MobX 背后的基本原则

原文博客地址

A few weeks ago Bertalan Miklos wrote a very interesting blog in which he compared the proxy based NX-framework with MobX. That blog is not only interesting because it proves the viability of proxies, but even more because it touches some very fundamental concepts in MobX and transparent reactivity (automatic reactivity) in general. Concepts I probably did not elaborate enough on so far. So let me share the mental model behind some of the unique features of MobX.

几周前，Bertalan Miklos 写了一篇非常有趣的博客，他将基于代理的 NX 框架与 MobX 进行了比较。那个博客很有趣，不仅因为它证明了代理的可行性，更因为它涉及到 MobX 中一些非常基本的概念和一般的透明反应性（自动反应性）。到目前为止，我可能没有详细说明的概念。因此，让我分享一下 MobX 的一些独特功能背后的心智模型。

Why MobX runs all derivations synchronously

为什么 MobX 同步运行所有派生

The article touches a very remarkable feature (imho) of MobX: in MobX all derivations are run synchronously. Which is quite unusual. Most UI frameworks don’t do this (if any at all). (Reactive stream libraries like RxJS run by default synchronous as well, but they lack transparent tracking so the situation is not entirely comparable).

这篇文章触及了 MobX 的一个非常显着的特性（恕我直言）：在 MobX 中，所有派生都是同步运行的。这是很不寻常的。大多数 UI 框架不这样做（如果有的话）。（像 RxJS 这样的响应式流库也默认

同步运行，但是它们缺乏透明的跟踪，所以情况并不完全可比）

Before starting MobX, I did quite some research on how existing libraries were perceived by developers. Frameworks like Meteor, Knockout, Angular, Ember and Vue all expose reactive behavior similar to MobX. And they exist already for a long time. So why did I built MobX? When digging through the issues and comments of people that were unhappy with these libraries, it occurred to me that there was one recurring theme which caused the chasm between the promise of reactivity and the nasty issues one had to deal with in practice.

在开始 MobX 之前，我对开发人员如何看待现有库进行了相当多的研究。 Meteor、Knockout、Angular、Ember 和 Vue 等框架都暴露了类似于 MobX 的反应行为。它们已经存在很长时间了。那么我为什么要构建 MobX？ 在挖掘对这些库不满意的人的问题和评论时，我突然想到，有一个反复出现的主题导致了反应性承诺与实践中必须处理的讨厌问题之间的鸿沟。

> *That recurring theme is “predictability”. If a framework runs your code twice, or with a delay, it becomes hard to debug it. Or to reason about it. Even ‘simple’ abstractions like promises are notoriously hard to debug due to their async nature.*

> 这个反复出现的主题是“可预测性”。如果一个框架运行你的代码两次，或者有延迟，就很难调试它。或者去推理它。由于它们的异步性质，即使是像 Promise 这样的“简单”抽象也很难调试。

I believe unpredictability to be, rightfully, one of the most important reasons for the popularity of Flux patterns and especially Redux: it addresses exactly this issue of predictability when scaling up. There is no magic scheduler at work.

我相信不可预测性理所当然地成为 Flux 模式尤其是 Redux 流行的最重要原因之一：它在扩展时正好解决了这个可预测性问题。工作中没有神奇的调度程序。

MobX however took another approach; instead of leaving behind the whole concept of automatically tracking and running functions, it tries to address the root causes. So that we can still reap the benefits of this model. Transparent reactivity is declarative, high-level and concise. It does this by adding two constraints:

然而，MobX 采取了另一种方法。它没有放弃自动跟踪和运行功能的整个概念，而是试图解决根本原因。这样我们仍然可以从这种模式中获益。透明的反应性是声明性的、高级的和简洁的。它通过添加两个约束来做到这一点：

1. Make sure that for any given set of mutations, any affected derivation will run exactly once.
2. Derivations are never stale, and their effects are immediately visible to any observer.

1. 确保对于任何给定的突变集，任何受影响的派生将只运行一次。
2. 推导永远不会过时，任何观察者都可以立即看到它们的效果。

Constraint 1. addresses so called “double runs”. It makes sure that if one derived value depends on another derived value, these derivations run in the right order. So that none of them accidentally read a stale value. How that exactly works is described in great detail in an earlier blog post.

约束 1. 解决所谓的“双重运行”。它确保如果一个派生值依赖于另一个派生值，则这些派生以正确的顺序运行。这样他们都不会意外读取过时的值。在之前的博客文章中详细描述了它的工作原理。

The second constraint; derivations are never stale; is a lot more interesting. Not only does it prevent so called “glitches” (temporary inconsistencies). It requires a fundamental different approach to scheduling derivations.

第二个约束；衍生品永远不会过时；有趣得多。它不仅可以防止所谓的“故障”（临时不一致）。它需要一种根本不同的方法来调度推导。

So far UI libraries have always taken the easy way out when it comes to scheduling derivations: Mark derivations as dirty, and re-run them on the next tick after all the state has been updated.

到目前为止，UI 库在调度派生方面总是采取简单的方法：将派生标记为脏，并在所有状态更新后的下一个滴答时重新运行它们。

This is simple and straight forward. It is a fine approach if your only concern is updating the DOM. The DOM usually lags a bit ‘behind’ and we hardly read data from it programmatically. So temporary staleness is not really an issue. Yet temporary staleness kills the general applicability of a reactive library. Take for example the following snippet:

这很简单直接。如果您唯一关心的是更新 DOM，这是一个很好的方法。 DOM 通常有点“落后”，我们几乎不会以编程方式从中读取数据。因此，暂时的陈旧性并不是真正的问题。然而，暂时的陈旧性扼杀了反应式库的普遍适用性。以下面的代码片段为例：

```
const user = observable({
  firstName: “Michel”,
  lastName: “Weststrate”,   // MobX computed attribute
  fullName: computed(function() {
    return this.firstName + " " + this.lastName
  })
})

user.lastName = “Vaillant”

sendLetterToUser(user)
```

The interesting question now is: When the `sendLetterToUser(user)` runs, will it see an *updated* or a *stale* version of the `fullName` of the user? When using the MobX the answer is always “updated”: Because MobX guarantees that any derivation is updated synchronously. This does not only prevent a lot of nasty surprises, it also makes debugging much simpler as a derivation always has the causing mutation in its stack.

现在有趣的问题是：当 sendLetterToUser(user) 运行时，它会看到用户全名的更新版本还是陈旧版本？使用 MobX 时，答案总是“更新的”：因为 MobX 保证任何派生都是同步更新的。这不仅防止了很多令人讨厌的意外，而且还使调试变得更加简单，因为派生总是在其堆栈中具有导致突变的原因。

So if you are wondering why a derivation is running, simply walking up the stack brings you back to the action that caused the derivation to be invalidated. If MobX would be using async scheduling / processing of derivations these advantages are lost. The library would not be as generally applicable as it is know.

因此，如果您想知道为什么派生正在运行，只需沿着堆栈向上走，您就会回到导致派生无效的操作。如果 MobX 使用派生的异步调度/处理，这些优势就会丢失。该库不会像已知的那样普遍适用。

When I started on MobX, there was a lot of skepticism on whether this could be done efficiently enough: Ordering the derivation tree and running derivations with each mutation.

当我开始使用 MobX 时，有很多人怀疑这是否可以足够有效地完成：对派生树排序并针对每个突变运行派生。

But here we are, with a system that is out of the box often more efficient than manually optimized code, as reported for example [here]() and [here]().

但是在这里，我们使用了一个开箱即用的系统，通常比手动优化的代码更有效，如此处和此处所报告的示例。

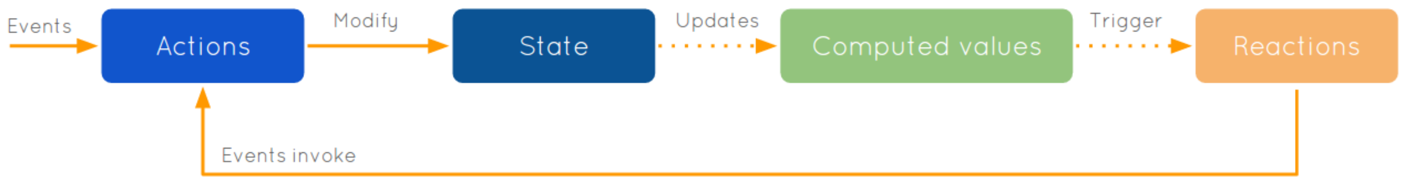
Transactions & Actions(事务和行为)

Yes, there is a small payoff to pay; mutations should be grouped in *transactions* to process multiple changes atomically. Transactions postpone the execution of derivations to the end of the transaction, but still runs them synchronously. Even cooler; if you use a computed value before the transaction has ended, MobX will ensure you get an updated value of that derivation nonetheless!

是的，需要支付少量回报；突变应该在事务中分组以原子地处理多个更改。事务将派生的执行推迟到事务结束，但仍同步运行它们。更酷；如果您在交易结束之前使用计算值，MobX 将确保您获得该推导的更新值！

In practice nobody uses transactions explicitly, they are even deprecated in MobX 3. Because actions apply transaction automatically. Actions are conceptually much nicer; an action indicates a function that will *update* state. They are the inverse of reactions, which *respond* to state updates.

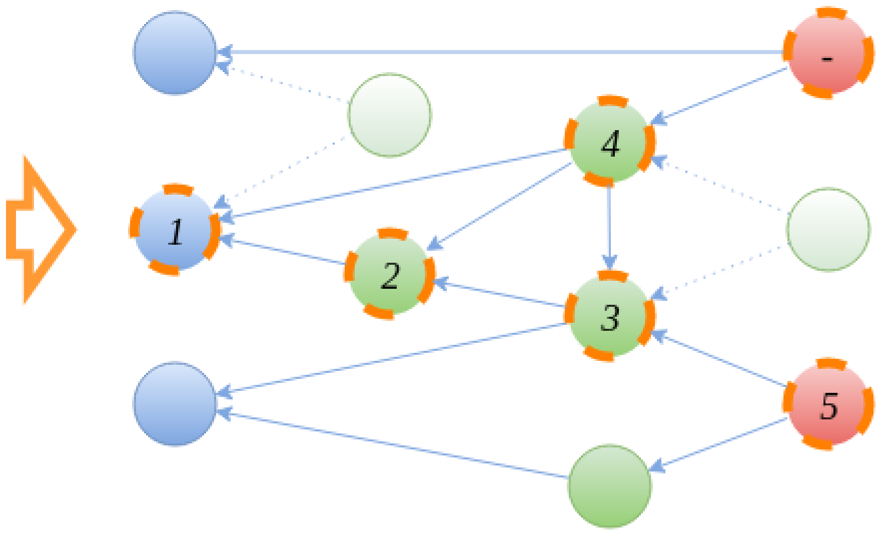
在实践中，没有人明确使用事务，它们甚至在 MobX 3 中被弃用。因为动作会自动应用事务。动作在概念上要好得多；动作表示将更新状态的功能。它们是对状态更新作出响应的反应的逆反应。



Computed values and reactions（计算和响应）

Another thing MobX focuses strongly on is the separation between values that can be derived (*computed values*), and side effects that should be triggered automatically if state changes (*reactions*). The separation of these concepts is very important and fundamental to MobX.

MobX 重点关注的另一件事是可导出的值（计算值）与状态变化时应自动触发的副作用（反应）之间的分离。这些概念的分离对于 MobX 来说非常重要和基础。



Example derivation graph. Observable state(blue), computed values (green) and reactions (red). Computed values will suspend (light green) if not observed (indirectly) by a reaction. MobX makes sure that each derivation runs only once and in the most optimal order after a mutation.

示例推导图。可观察状态（蓝色）、计算值（绿色）和反应（红色）。如果反应没有（间接）观察到，计算值将暂停（浅绿色）。 MobX 确保每个派生只运行一次，并且在突变后以最佳顺序运行。

Computed values should always be preferred over reactions.

计算值应始终优先于反应。

For several reasons:（有几个原因：）

1. They provide a lot of conceptual clarity. Computed values should always expressed purely in terms of other observable values. This results in a clean derivation graph of computations, rather then an unclear chain of reactions that trigger each other.

它们提供了很多概念上的清晰度。计算值应始终纯粹用其他可观察值表示。这会产生一个清晰的计算推导图，而不是一个相互触发的不清楚的反应链。

2. In other words, reaction that trigger more reactions, or reactions that update state: They are both considered anti patterns in MobX. Chained reactions lead to a hard to follow chain of events and should be avoided.

换句话说，触发更多反应的反应，或更新状态的反应：它们都被认为是 MobX 中的反模式。连锁反应会导致难以跟踪的事件链，应该避免。

3. For computed values MobX can be determine whether they are in use somewhere. This means that computed values can automatically be suspended and garbage collected. This saves tons of boilerplate and has a significant positive impact on performance.

对于计算值，MobX 可以确定它们是否在某处使用。这意味着计算值可以自动挂起和垃圾收集。这可以节省大量样板文件，并对性能产生显着的积极影响。

4. Computed values are enforced to be side-effect free. Because computed values are not allowed to have side effects, MobX can safely reorder the execution order of computed values to guarantee a minimal amount of re-runs. It can decide to only lazily run the computation if it is not observed.

计算值被强制为无副作用。因为计算值不允许有副作用，MobX 可以安全地重新排序计算值的执行顺序，以保证最少的重新运行。如果没有观察到，它可以决定只懒惰地运行计算。

5. Computed values are cached automatically. This means that reading a computed value will not re-run the computation as long as none of the involved observables did changes.

计算值会自动缓存。这意味着只要所涉及的 observables 没有发生变化，读取计算值就不会重新运行计算。

In the end, every software system needs side effects. For the simple reason we always need to bridge from reactive to imperative code. For example to make network requests or flush the DOM. However, often these reactions can be put away in clean abstractions like React `observer` components.

 最后，每个软件系统都需要副作用。出于简单的原因，我们总是需要从响应式代码过渡到命令式代码。例如发出网络请求或刷新 DOM。然而，这些反应通常可以放在像 React 观察者组件这样的干净抽象中。

So MobX goes great lengths to make sure stale values can never be observed, and derivations do not run more often then one would expect. In fact, computations are not even run at all if nobody is actively observing. In practice this makes all the difference. There is often some initial resistance to MobX, as the concepts remind people of the unpredictable behavior of MVVM frameworks. However, I believe the semantical clarity of actions, computed values and reactions, the fact that no stale values can be observed, the fact that all derivations run on the same stack as the causing action, makes all the difference here.

因此，MobX 竭尽全力确保永远不会观察到陈旧的值，并且派生不会比人们预期的更频繁地运行。事实上，如果没有人积极观察，甚至根本不会运行计算。在实践中，这一切都不同。 MobX 最初通常会遇到一些阻力，因为这些概念提醒人们 MVVM 框架的不可预测行为。然而，我相信动作、计算值和反应的语义清晰性、不能观察到陈旧值的事实、所有派生与引发动作在同一个堆栈上运行的事实，都在此处产生了重大影响。

Proxies & MobX

MobX is used heavily in production, and therefor has made a commitment to run on every ES5 environment. This makes it possible to target practical any browser with MobX, but also makes it unacceptable at this moment to rely on Proxies. For this reason MobX has some rough edges, like not fully supporting expando properties and using faux-arrays. It is no secret that the plan always have been to ultimately move to a Proxy based implementation. MobX 3 already has some changes that prepare for proxy usage, and the first opt-in Proxy based features can be expected soon. However the core will remain Proxy free until the vast majority of devices and browsers support them.

 MobX 在生产中大量使用，因此承诺在每个 ES5 环境中运行。这使得任何带有 MobX 的浏览器都可以成为实际的目标，但也使得目前依赖代理变得不可接受。出于这个原因，MobX 有一些粗糙的边缘，比如不完全支持 expando 属性和使用仿数组。计划始终是最终转向基于代理的实施，这已不是什么秘密。 MobX 3 已经有了一些为代理使用做准备的变化，预计很快就会推出第一个基于代理的可选功能。然而，在绝大多数设备和浏览器支持它们之前，核心将保持免费代理。

The case for shallow data structures（浅层数据结构的案例）

Regardless the future migration to proxies, the modifiers / shallow observable concept will remain in some form in MobX.

无论未来迁移到代理，修饰符/浅可观察概念将在 MobX 中以某种形式保留。

The reason for the modifiers mechanism is not performance; it is interoperability.

修改器机制的原因不是性能；它是互操作性。

As long as all data in the state of your application is under your control, auto-observability is very convenient. MobX started with just that. But at some point you discover that the world is not as ideal as you want it to be. In each application there are many libraries, each playing their own role, and each making their own assumptions. Modifiers and shallow collections were introduced in MobX to be able to make it clear what data MobX is allowed to manage.

只要您的应用程序状态中的所有数据都在您的控制之下，自动可观察性就非常方便。 MobX 就是这样开始的。但是在某些时候，您会发现世界并不像您希望的那样理想。在每个应用程序中都有许多库，每个库都扮演着自己的角色，并且各自做出自己的假设。 MobX 中引入了修饰符和浅层集合，以便能够明确允许 MobX 管理哪些数据。

For example, sometimes you need to store references to external concepts. However, automatic conversion to observables is often not desirable for objects that are managed by some external library already (for example JSX or DOM elements). It can easily interfere with the internal assumptions of that library. One can find quite some issues in the MobX issue tracker where turning objects unintentionally into observables lead to unexpected behavior. Modifiers are not meant to signal “please make this fast”, but to signal “only observe references to the object, consider the object itself a black box as it is not under our control”.

例如，有时您需要存储对外部概念的引用。但是，对于已经由某些外部库管理的对象（例如 JSX 或 DOM 元素），通常不希望自动转换为可观察对象。它很容易干扰该库的内部假设。人们可以在 MobX 问题跟踪器中找到相当多的问题，在这些问题中，无意中将对象转换为可观察对象会导致意外行为。修饰符不是为了表示“请加快速度”，而是表示“只观察对对象的引用，将对象本身视为黑匣子，因为它不受我们控制”。

This concept fits working with immutable data structures very nicely as well. It makes it possible to create, for example, an observable map of messages, where the messages themselves are considered immutable data structures.

这个概念也非常适合与不可变数据结构一起工作。例如，它可以创建一个可观察的消息映射，其中消息本身被认为是不可变的数据结构。

A second problem is that auto-observable collections always create ‘clones’, which is not always acceptable. Proxies always produce a new object and they work only in “one direction”. So if a the original of a proxied object is modified by the library that originally distributed it, proxies won’t detect that change. To illustrate:

第二个问题是自动可观察集合总是创建“克隆”，这并不总是可以接受的。代理总是产生一个新的对象，它们只在“一个方向”工作。因此，如果代理对象的原始文件被最初分发它的库修改，代理将不会检测到该更改。为了显示：

```
const target = { x: 3 }
const proxy = createObservableProxy(target)observe(() => {
 console.log(proxy.x)
})target.x = 4
// proxy.x is now 4, but no log statement will be written,
//as the proxy setter hook is not fired!
```

 Modifiers provide the necessary flexibility to deal with these situations. For now, since MobX uses property descriptors, it can actually enhance existing objects, so that data mutation can work in two directions if the need really arises.

修饰符为处理这些情况提供了必要的灵活性。目前，由于 MobX 使用属性描述符，它实际上可以增强现有对象，因此如果确实需要，数据变异可以在两个方向上起作用。

That being said, the way NX produces observable proxies on the fly during reads is super interesting. I’ m not sure how it handles referential transparency yet, but so far it seems a very smart thing to do. Avoid modifiers by reading / writing through `$raw` is a very interesting approach. I’ m not sure if it is cleaner to read or write, but it is definitely saves the introduction of some concepts like shallow which is great.

话虽如此，NX 在读取过程中动态生成可观察代理的方式非常有趣。我还不确定它是如何处理引用透明度的，但到目前为止，这似乎是一件非常聪明的事情。通过 $raw 读/写来避免修饰符是一种非常有趣的方法。我不确定它是读还是写更干净，但它绝对省去了一些概念的介绍，比如浅，这很棒。

What does untracked do?(未跟踪有什么作用？)

A small note has to be made on the semantics of `untracked`, unlike suggested, it doesn’ t relate to `$raw` updates in NX. In MobX there is simply no way to update data without notifying observers. Allowing this deliberately introduces the possibility of stale data in an application. Which goes against the principles of MobX. People sometimes think they need such a mechanism, but I never encountered a real life use case where there was not a conceptually cleaner solution.

必须对 untracked 的语义做一点说明，与建议的不同，它与 NX 中的 $raw 更新无关。在 MobX 中，根本无法在不通知观察者的情况下更新数据。故意允许这样做会在应用程序中引入陈旧数据的可能性。这违背了 MobX 的原则。人们有时认为他们需要这样的机制，但我从未遇到过没有概念上更清洁的解决方案的现实用例。

`untracked` works the other way around: It is not about undetectable *writes.* Instead, it just causes *reads* to be untracked. In other words, it is way of saying that we are not interested in any future updates of the data we use. Like `transaction`, nobody uses this API in practice, but it is a mechanism baked into *actions* because it makes conceptually a lot of sense: Actions run in response to a (user) event, not in response to state changes, so they shouldn’t be tracking which data they consume. That is what *reactions* are for.

untracked 则相反：它与不可检测的写入无关。相反，它只会导致读取未被跟踪。换句话说，就是说我们对我们使用的数据的任何未来更新不感兴趣。与事务一样，没有人在实践中使用此 API，但它是一种融入动作的机制，因为它在概念上很有意义：动作响应（用户）事件而运行，而不是响应状态变化，所以它们不应该跟踪他们使用的数据。这就是反应的目的。

Conclusion(结论)

MobX is designed to be a generally applicable reactivity library. It’s not just an utility to re-render the UI at the right moment.

MobX 被设计成一个普遍适用的反应性库。它不仅仅是在适当的时候重新渲染 UI 的实用程序。

Instead, it generalizes the concept of efficiently working (both in terms of performance and effort) with data that is just the project of some other data. At Mendix for example MobX is used in backend processes as well. Running derivations synchronously and the separation between computed values and reactions is very fundamental to MobX. It leads to a much clearer mental picture of the application state.

相反，它概括了有效工作的概念（在性能和工作量方面），数据只是一些其他数据的项目。例如，在 Mendix，MobX 也用于后端进程。同步运行推导以及计算值和反应之间的分离对于 MobX 来说是非常基础的。它导致应用程序状态的更清晰的心理画面。

Finally, nx-observe proves that proxies are very viable foundation for a transparent reactive programming library. Both conceptually and performance wise.

最后，nx-observe 证明代理是透明的反应式编程库非常可行的基础。无论是概念上还是性能上。