

Signal vs Observable 有啥区别？

本文翻译自Angular作者MIŠKO HEVERY

Signals vs Observables



```
const answer = 42;
```



```
const answer = useSignal(42);
```



```
const answer = from([42]);
```

关于框架内的Signal已经有很多[讨论](#)。但一个自然要问的问题是Signal与Observable有何不同。嗯，这是一个好问题，也是本文的目标，所以请继续阅读！

如何理解Signal 和 Observable？

我们都知道值是什么：

```
const answer = 42;
```

那Signal是什么呢？

```
const answer = useSignal(42);
```

OK，那Observable又是啥呢？

```
const answer = from([42]);
```

看起来很相似，但是我们尝试用`console.log()`打印出来：

```
console.log('Answer', answer);
```

打印Signal的话：

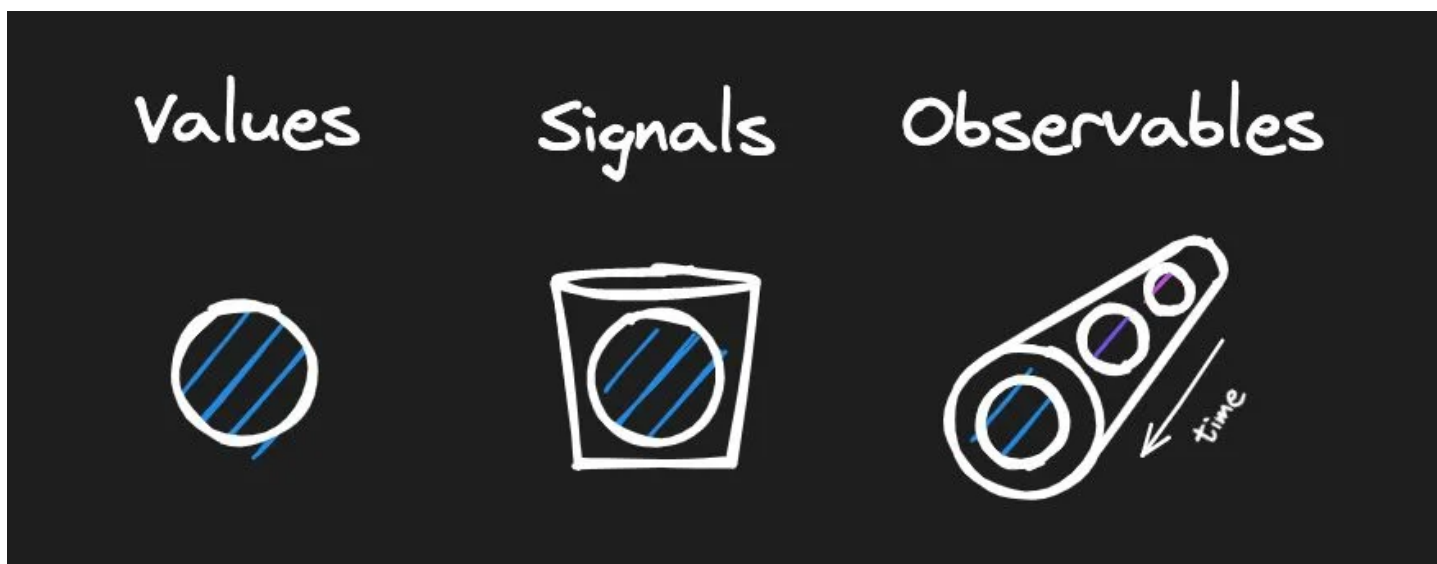
```
console.log('Answer', answer.value);
```

最后，打印Observable的话：

```
answer.subscribe((value) => console.log('Answer', value));
```

我认为Observable比较让人印象深刻，因为它是唯一一种我们无法直接获取到值的情况；相反，我们需要创建回调和订阅。这种差异是Signal与Observable差异的核心。

比喻



假设我们有一个由上图中的阴影圆圈表示的值。我们都知道如何与它打交道。它只是对值的引用。我们可以将值（或引用）传递到应用程序的不同部分。

应用程序的任何部分都可以读取该值来使用它。但值不是响应式的！“观察者”无法“知道”值何时发生变化。因此，虽然使用值很简单，但他们缺乏响应式！

Signal就像一个包含值的桶。你可以像将值传递给应用程序的其余部分一样传递存储桶。应用程序可以随时查看存储桶（读取）或更新存储桶中的内容（写入）。

但因为它是一个存储桶，所以存储桶“知道”何时发生读取或写入。因此，现在“观察者”可以“知道”何时发生读取或写入。能够“知道”何时读/写使得Signal具有响应式！

Observable与值或Signal有很大不同。Observable就像一个管道。在任何时候，管道都可以传递新的值。这有几个含义。首先，你不能只看管道是否适合你。相反，你必须注册一个回调并等待新值出现时调用回调。其次，随着时间的推移，Observable会带来值。这个“时间”概念是Observable的核心。（而Signal没有时间概念，它只是桶中的任何内容。）

这就是为什么在上面的示例中，我们可以直接读取值或Signal，但需要为Observable创建回调。这种差异（一个存储值的桶与随时间推移发布内容的管道）是Signal和Observable之间的核心差异，并且具有很多含义。那么让我们探讨一下这种差异的后果。

传递引用与时间通道




我们前面已经谈到了这一点，但让我们回顾一下以便更清楚。

值可以在你的应用程序中传递。这没什么特别的。

对Signal和Observable而言，我们不传递值，而是传递包含感兴趣值的容器。这是一个重要的区别，因为传递容器允许你在获得需要传递的值之前将其传递给应用程序的其余部分。本质上，传递容器允许你设置系统的“管道”，然后通过管道传递值。

Observable不是值的容器，而是随着时间的推移而变化的值的容器。时间成分对于Observable的概念非常重要。

由于Observable是随时间变化的值，因此你不能只“读取”当前值。没有“当前”值。这在Observable的世界中没有任何意义。相反，你必须注册一个回调，并且当值出现时Observable将调用回调。这就是为什么将Observable比喻成管道是更贴切的。

| |  Values |  Signals |  Observables |
|-----------|---|--|--|
| reference | value | container of value | container of values |
| time | (no concept) | (no concept) | value over time |

访问性

访问参考值是基本的。这就是普通的 JavaScript。

访问Signal的值也非常简单，但它需要某种形式的 getter 执行。从根本上来说，实际上只有两种选择：调用函数或访问属性（间接调用 getter 函数）

```
// retrieving function style:
const signalValue = signal();




// retrieving property getter style:
const signalValue = signal.value;
```

注意：讨论两种风格之间的权衡将成为未来一篇很好的文章，但这超出了本文的范围。

访问Observable的值有点复杂。人们不能只查看Observable，因为正如我们之前提到的，Observable是随时间变化的值。因此Observable要求我们注册一个回调函数。

回调函数有一些有趣的含义。这意味着我们的回调函数不会立即使用最后一个值（已经传递的值）进行调用。相反，在出现新值之前不会发生任何事情。对比Signal，你是可以在Signal中获取当前值的。

getter 与回调的含义是，Signal是基于pull的（代码可以同步读取Signal值），而Observable是基于push的（当值出现时，Observable将值传递给回调。）

| |  Values |  Signals |  Observables |
|-----------|--|---|---|
| reference | value | container of value | container of values |
| time | (no concept) | (no concept) | value over time |
| access | direct | getter -> pull | callback -> push |

订阅




通过调用 subscribe 方法并传入回调来订阅 Observable。所以我们称之为“显式”订阅。（是的，框架有时可以代替你调用订阅，但它本质仍然是订阅。）

虽然Signal也可以有“显式”订阅，但大多数都依赖于“隐式”订阅。让我们来看看它是如何工作的。

```
const firstName = useSignal('');
const lastName = useSignal('');
const fullName = useComputed$(() => {
  return lastName.value + ', ' + firstName.value;
});
```

注意：不同框架语法大同小异，但本质是一样的，比如Solid中的Reaction和Derivition。

在上面的示例中，useComputed\$() 闭包在特殊上下文中执行。该上下文“观察” Signal读取并记录它们。这本质上是“隐式”创建订阅，无需单独监听每个Signal。没有“显式订阅” API。（许多Signal的实现都没有“订阅” API。）

| |  Values |  Signals |  Observables |
|---------------------------|--|---|---|
| reference | value | container of value | container of values |
| time | (no concept) | (no concept) | value over time |
| access | direct | getter -> pull | callback -> push |
| subscription (cleanup) | - | implicit (typical) | explicit |




异步模型

Signal是一种同步执行模型。同步从Signal读取值或向Signal写入值。如果你想处理异步代码，许多Signal实现都有一些“Effect” API 允许你这样做。

Signal的同步性质是隐式的，因为它们创建“隐式” 订阅。如果“计算的” 回调是异步的，上下文就不可能观察读取，因为它无法知道未来应该观察哪些读取/写入。

Observable本质上是异步的。毕竟，它们是“随着时间的推移而产生的值”。时间意味着异步性。但Observable实现通常是同步和异步回调的组合。

因此，它是一种混合模型。将新值推送到 observable 中可能会也可能不会同步调用订阅。

| |  Values |  Signals |  Observables |
|---------------------------|--|---|---|
| reference | value | container of value | container of values |
| time | (no concept) | (no concept) | value over time |
| access | direct | getter -> pull | callback -> push |
| subscription (cleanup) | - | implicit (typical) | explicit |
| async model | - | synchronous (effects for async) | hybrid |

响应图




在Observable中，人们通常在“设置阶段”设置响应图，然后该图往往保持静态（有用于更改它的API，但这不是大多数人使用它的方式）。

Signal往往具有非常动态的响应图。以这段代码为例：

```
const location = useSignal('37°46'39"N 122°24'59"W');
const zipCode = useSignal('94103');
const preference = useSignal('location');

const weather = useComputed$(() => {
  switch (preference.value) {
    case 'location':
      return lookUpWetherByGeo(location.value);
    case 'zip':
      return lookUpWetherByZip(zipCode.value);
    default:
      return null;
  }
});
```

在上面的示例中，weather Signal可以订阅 `zipCode`，`location` 或两者都不订阅，具体取决于 preference 的值。这就是Signal的工作原理。

| |  Values |  Signals |  Observables |
|---------------------------|--|---|---|
| reference | value | container of value | container of values |
| time | (no concept) | (no concept) | value over time |
| access | direct | getter -> pull | callback -> push |
| subscription (cleanup) | - | implicit (typical) | explicit |
| async model | - | synchronous (effects for async) | hybrid |
| graph | - | dynamic | fixed |

合适的选择

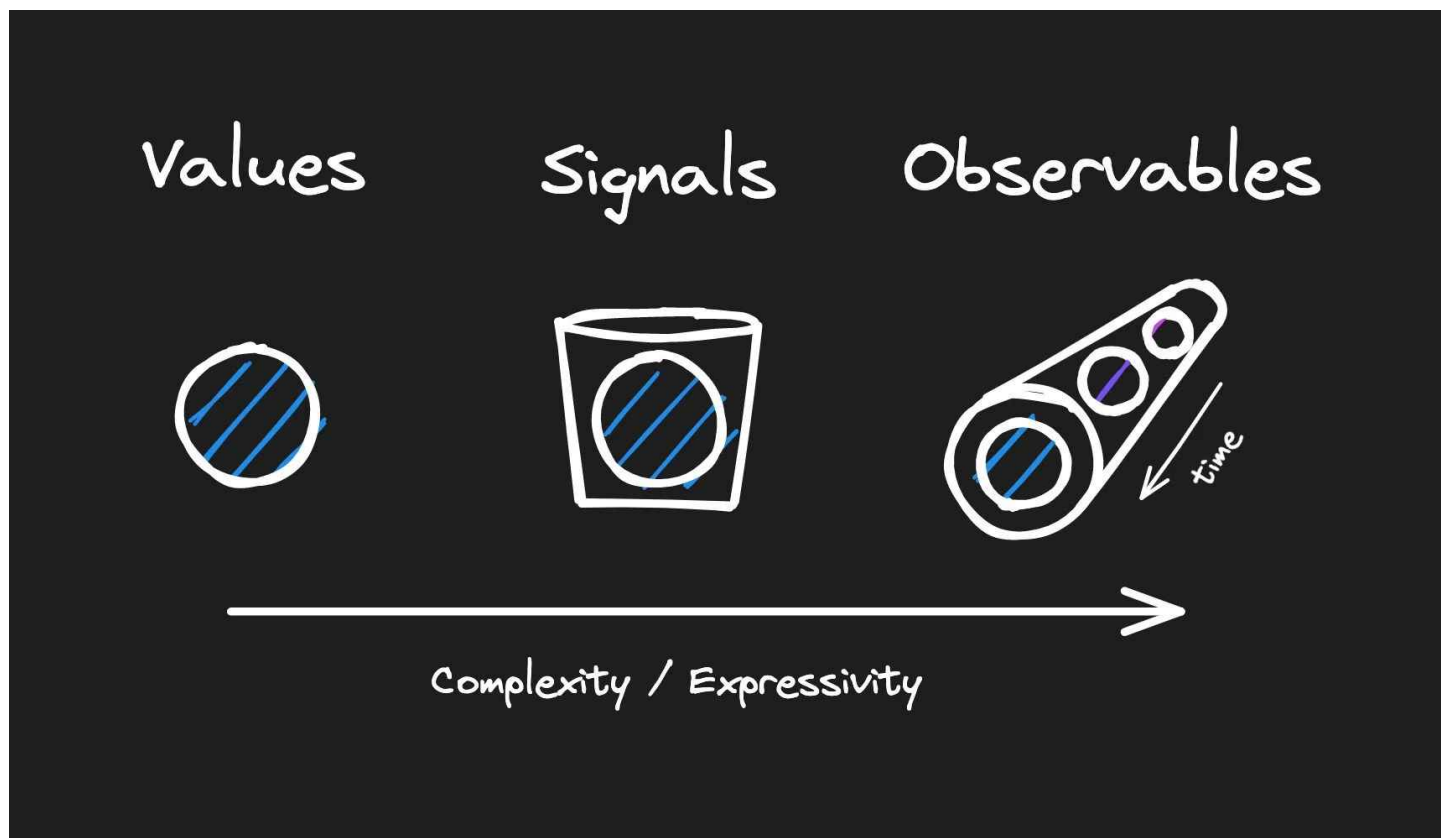
Signal比Observable更好吗？这是一个错误的问题。它们是两个不同的东西，是否选择Observable而不是Signal取决于你想要实现的目标。

如果随时间变化的值是你的问题域的关键概念，那么Observable是正确的选择。另一方面，如果Signal的“时间”分量不是你的问题域所需要的，那么使用Observable会带来不必要的复杂性。

我的想法是，它是一个平衡。一方面，你有值。它们很简单，但缺乏表达性，而表达性有时是有用的。

另一个极端是Observable，它可以做任何事情，但附带更复杂的 API。中间是Signal，它们不像Observable那么强大，但更直接。

因此，值、Signal和Observable是复杂性和表达性的权衡。你得根据你的问题领域选择适合的选项。



如果是UI的话，谁是更合适的选择？

如果你的问题领域是构建 UI，我认为Observable在大多数情况下都太重了。是的，有一些领域是适合用Observable，但对于大多数 UI 来说，Signal已经足够了，而且Signal具有较少的 API 接口这一事实在大多数情况下是正确的权衡，这就是为什么我认为Signal是 Web 框架的未来。

参考

- <https://www.builder.io/blog/signals-vs-observables>