细粒度响应式实践介绍

> 本文翻译自Solid的作者**Ryan Carniato**。

响应式编程已经存在了数十年，但它似乎有时候比较流行又有时候比较沉寂（come in and out of fashion）。在 JavaScript 前端中，它在过去几年中再次呈上升（upswing）趋势。它超越（transcend）了框架，是任何开发人员都需要熟悉的有用主题。

然而，它并不总是那么容易。首先，对于初学者而言响应式有不同类型。这个术语和命名常常充斥着相同的词，对不同的人来说意味着不同的事情。其次，它有时看起来像魔术。事实并非如此，但在理解它“是什么”之前，很难不被“它是如何做到”分散注意力。这使得通过实际例子进行教学成为一项挑战，同时也成为一种谨慎的平衡，以防止过于理论化。

这篇文章不会聚焦“它是如何做到”，我将尝试对 MobX、Vue、Svelte、Knockout 和 Solid 等库使用的细粒度反应性方法进行最浅显易懂的介绍。

> 注意：这里的响应式可能与你所熟悉的响应式（如RxJS）有所不同。它们是相关的并且有相似之处，但它们并不完全相同

虽然这篇文章的目标群体是对细粒度响应式或者一般的响应式不熟悉的人，但是它仍然是一个中级的主题，需要了解 JavaScript 并熟悉一些计算机科学入门主题。我会尽力详细解释，但请随时在评论中留下问题。

我会在Codesandbox中贴一些片段。我会使用我自己开发的Solid库来完成这些示例，并且这篇文章将使用Solid的语法。但是其实它们在不同的库里面使用起来大同小异。你可以点击相关的链接在完全交互式的环境中体验这些示例。

基本的语法

细粒度响应式是通过基本的网络原语构建的。我所说的原语是指像 Promise 这样的简单结构，而不是像字符串或数字这样的 JavaScript 原始值。

每个原语都充当网络图中的一个节点。你可以将它们视为理想化的电路。任何更改都会同时应用于所有节点。它所解决的问题是在某个时间单个节点状态的同步。这是我们在构建用户界面时经常遇到的问题。

让我们从了解不同类型的语法开始。

Signal

Signal是响应式系统中最基本的部分。它由getter、setter和value组成。经过通常在论文中它被称为Signal，它也常常被称为Observable，Atom，Subject 或 Ref。

当然，仅仅这个还不足以有趣。这些多多少少只是可以存储任何内容的值。重要的细节是 `get` 和 `set` 都可以运行任意代码。这对于传播更新非常重要。

函数是执行此操作的主要方法，但您可能已经看到通过对象 getter 或代理完成的这个效果：

```
const [count, setCount] = createSignal(0);

// read a value
console.log(count()); // 0

// set a value
setCount(5);
console.log(count()); //5
```

或者通过编译器完成：

```
// Svelte
let count = 0;
// read a value
console.log(count); // 0

// set a value
count = 5;
```

Signal的核心是事件发射器，但区别在于事件订阅的管理方式。

Reaction

如果没有合作伙伴Reaction，Signal不会很有趣。Reaction也被称为Effect，Autorun，Watch 或 Computed，通过观测Signal，当值发生改变时，重新运行这些Reaction。

它们被包裹在函数表达式中，会在初次以及Signal更新时被运行。

```
console.log("1. Create Signal");
const [count, setCount] = createSignal(0);

console.log("2. Create Reaction");
createEffect(() => console.log("The count is", count()));

console.log("3. Set count to 5");
setCount(5);

console.log("4. Set count to 10");
setCount(10);
```

执行结果如下（Codesandbox）：

```
1. Create Signal
2. Create Reaction
The count is 0
3. Set count to 5
The count is 5
4. Set count to 10
The count is 10
```

乍看起来比较像黑魔法，但这就是我们的Signal需要 `getter` 的原因。每当Signal被执行（setter）时，包装函数都会检测到它并自动订阅它。我们将继续解释有关此行为的更多信息。

重要的是，这些Signal可以携带任何类型（any sort of）的数据并且Reaction可以基于它做任何事情。在Codesandbox的例子里我创建了一个自定义的日志函数往页面DOM上追加日志。我们可以用这些完成任何更新

另外，这个更新过程是同步完成的。在我们打印出下一条指令之前，Reaction已经执行了。

就是这样。我们拥有细粒度响应式所需的所有组成。Signal和Reaction。被观察者和观察者。实际上，你只需要通过这2个语法就能完成大多数响应式行为。然而，我们也有必要谈谈另一个原语。

Derivation

通常（More often than not）我们需要用不同的方式呈现我们的数据并且在多个Reaction中使用同样的Signal。我们可以将这个逻辑写在Reaction中，或者更进一步提取一个辅助函数。

```
console.log("1. Create Signals");
const [firstName, setFirstName] = createSignal("John");
const [lastName, setLastName] = createSignal("Smith");
const fullName = () => {
  console.log("Creating/Updating fullName");
  return `${firstName()} ${lastName()}`
};

console.log("2. Create Reactions");
createEffect(() => console.log("My name is", fullName()));
createEffect(() => console.log("Your name is not", fullName()));

console.log("3. Set new firstName");
setFirstName("Jacob");
```

```
1. Create Signals
2. Create Reactions
Creating/Updating fullName
My name is John Smith
Creating/Updating fullName
Your name is not John Smith
3. Set new firstName
Creating/Updating fullName
My name is Jacob Smith
Creating/Updating fullName
Your name is not Jacob Smith
```

> 注意：在这个例子里面 `fullName` 是一个辅助函数。这是因为为了在Effect下读取Signal，我们需要延迟到Effect运行时执行它。如果我们只是使用一个值，如果它只是一个值，则没有机会获得跟踪效果或得到重新运行的效果。这与React显然不一样

```
function playground() {
  log("1. Create Signals");
  const [firstName, setFirstName] = createSignal("John");
  const [lastName, setLastName] = createSignal("Smith");
  const fullName = `${firstName()} ${lastName()}`;

  log("2. Create Reactions");
  createEffect(() => log("My name is", fullName));
  createEffect(() => log("Your name is not", fullName));

  log("3. Set new firstName");
  setFirstName("Jacob");
}
```

```
1 1. Create Signals
2 2. Create Reactions
3 My name is John Smith
4 Your name is not John Smith
5 3. Set new firstName
```

但是有时候计算推导值的过程是昂贵的并且我们不想每次都进行计算。因此，我们有第三个基本原语，其作用类似于函数记忆，将中间计算存储为自己的Signal。这些被称为“Derivation”，但也被称为“Memo”、“Computed”、“Pure Computed”。

我们可以对比下将前面的辅助函数 `fullName` 作为Derivation的例子：

```
console.log("1. Create Signals");
const [firstName, setFirstName] = createSignal("John");
const [lastName, setLastName] = createSignal("Smith");

console.log("2. Create Derivation");
const fullName = createMemo(() => {
  console.log("Creating/Updating fullName");
  return `${firstName()} ${lastName()}`
});

console.log("3. Create Reactions");
createEffect(() => console.log("My name is", fullName()));
createEffect(() => console.log("Your name is not", fullName()));

console.log("4. Set new firstName");
setFirstName("Jacob");
```

```
1. Create Signals
2. Create Derivation
Creating/Updating fullName // 这里会有一次执行，即初次执行
3. Create Reactions
// 这里不会有Creating/Updating fullName
My name is John Smith
// 这里不会有Creating/Updating fullName
Your name is not John Smith
4. Set new firstName
Creating/Updating fullName // 只会执行一次
My name is Jacob Smith
// 这里不会有Creating/Updating fullName
Your name is not Jacob Smith
```

这次 `fullName` 会在创建时立即计算它的值并且在Reaction读取的时候不会重新执行。当我们更新最初始的Signal的时候它才会再次执行，但也只会在这个更新传播到Reaction时只执行一次。

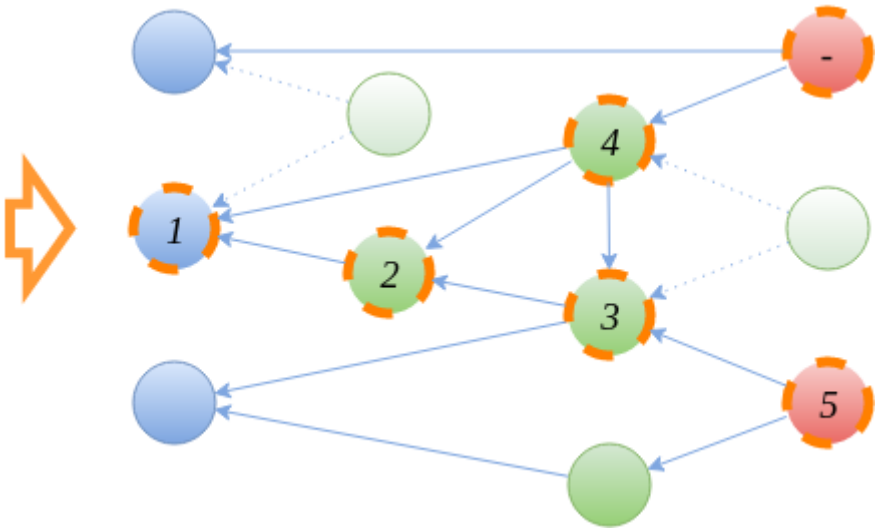
尽管计算 `fullName` 远算不上复杂的计算但是我们能看到Derivation是如何通过将值缓存在独立执行的表达式中来节省我们的工作的，该表达式本身是可跟踪的。

更重要的是，因为它们是派生的，所以它们保证是同步的。在任何时候，我们都可以确定它们的依赖关系并评估它们是否可能已过时（stale）。使用Reaction写入其他信号可能看起来等效，但不能带来

这种保证。这些Reaction不是Signal的显式依赖性（因为Signal没有依赖）。我们将在下一节中更多地了解依赖关系的概念。

> 注意：有些库会延迟计算Derivation因为他们只需要在被读取时才进行计算，并且它允许更激进地处置当前未读取的派生。这些方法之间存在一些权衡，超出了本文的范围。

响应式生命周期



细粒度响应式维持着许多响应式节点之间的连接。在任何地方发生变化，部分图将重新评估，并可以创建和删除连接。

> 注意：像Svelte或Marko之类的预编译库不会使用与此相同的运行时追踪技术，而是静态分析依赖关系。因此，他们对响应式表达式何时重新运行的控制较少，因此它们可能会过度执行，但管理订阅的开销较少。

考虑当条件改变时我们推导出的数据：

```
console.log("1. Create");
const [firstName, setFirstName] = createSignal("John");
const [lastName, setLastName] = createSignal("Smith");
const [showFullName, setShowFullName] = createSignal(true);

const displayName = createMemo(() => {
  if (!showFullName()) return firstName();
  return `${firstName()} ${lastName()}`
});

createEffect(() => console.log("My name is", displayName()));

console.log("2. Set showFullName: false ");
setShowFullName(false);

console.log("3. Change lastName");
setLastName("Legend");

console.log("4. Set showFullName: true");
setShowFullName(true);
```

```
1. Create
My name is John Smith
2. Set showFullName: false
My name is John
3. Change lastName
// 这里不会打印My name is xxx，因为依赖未变
4. Set showFullName: true
My name is John Legend
```

需要注意的是，当我们在步骤 3 中更改 `lastName` 时，我们不会得到新的日志。这是因为每次我们重新运行响应式表达式时，我们都会重建它的依赖关系。简单地说，当我们更改 `lastName` 时，没有人在观察它。

正如我们将 `showFullName` 设置回 true 时观察到的那样，该值确实发生了变化。然而，没有任何通知。这是一种安全的交互，因为为了让 `lastName` 再次被跟踪，`showFullName` 必须更改并且被跟踪。

依赖关系是响应式表达式读取以生成其值的信号。反过来，这些信号持有许多响应式表达式的订阅。当他们更新时，他们会通知依赖他们的订阅者。

我们在每次执行时构建这些订阅/依赖项。并在每次重新运行响应式表达式或最终释放它们时释放它们。你可以使用 onCleanup 帮助程序查看该时序：

```
console.log("1. Create");
const [firstName, setFirstName] = createSignal("John");
const [lastName, setLastName] = createSignal("Smith");
const [showFullName, setShowFullName] = createSignal(true);

const displayName = createMemo(() => {
  console.log("### executing displayName");
  onCleanup(() =>
    console.log("### releasing displayName dependencies")
  );
  if (!showFullName()) return firstName();
  return `${firstName()} ${lastName()}`
});

createEffect(() => console.log("My name is", displayName()));

console.log("2. Set showFullName: false ");
setShowFullName(false);

console.log("3. Change lastName");
setLastName("Legend");

console.log("4. Set showFullName: true");
setShowFullName(true);
```

```
1. Create
### executing displayName
My name is John Smith
2. Set showFullName: false
### releasing displayName dependencies
### executing displayName
My name is John
3. Change lastName
```

```
4. Set showFullName: true
### releasing displayName dependencies
### executing displayName
My name is John Legend
```

同步执行

细粒度反应系统同步且立即执行其更改。他们的目标是无故障（*glitch-free*），因为永远不可能观察到不一致的状态。这带来了可预测性，因为对于任何给定的更改代码只会运行一次。

当我们不能相信观察到的结果时，不一致的状态可能会导致意外的行为。

演示其工作原理的最简单的方法是同时应用 2 个更改，这些更改将输入到执行Reaction的 Derivation 中。我们将使用 `batch` （批处理）助手来演示。 `batch` 将更新包装在事务中，该事务仅在完成执行表达式时应用更改。

```
console.log("1. Create");
const [a, setA] = createSignal(1);
const [b, setB] = createSignal(2);
const c = createMemo(() => {
  console.log("### read c");
  return b() * 2;
});

createEffect(() => {
  console.log("### run reaction");
  console.log("The sum is", a() + c());
});

console.log("2. Apply changes");
batch(() => {
  setA(2);
  setB(3);
});
```

```
1. Create
### read c
### run reaction
The sum is 5
```

```
2. Apply changes
### run reaction
### read c
The sum is 8 // 也就是看到的是最终态
```

在这个示例中，代码按照你的预期自上而下地创建。但是，批量更新会反转run/read日志的输出顺序。

> 注意：当我们把更新B放在更新A前面，顺序又会有些差异

```
function playground() {
  log("1. Create");
  const [a, setA] = createSignal(1);
  const [b, setB] = createSignal(2);
  const c = createMemo(() => {
    log("### read c");
    return b() * 2;
  });

  createEffect(() => {
    log("### run reaction");
    log("The sum is", a() + c());
  });

  log("2. Apply changes");
  batch(() => {
    setB(3);
    setA(2);

  });
}
```

```
1. Create
### read c
### run reaction
The sum is 5
2. Apply changes
```

```
### read c
### run reaction
The sum is 8
```

当我们更新值时，即使 A 和 B 同时应用，我们也需要从某个地方开始，所以我们首先运行 A 的依赖项。因此，Reaction首先运行，但检测到 C 已过时，我们立即在读取时运行它，所有内容都会执行一次并正确计算。

当然，你可能会想到一种按顺序解决此静态情况的方法，但请记住依赖关系可能会在任何运行中发生变化。细粒度响应式库使用混合推/拉方法来保持一致性。它们不像事件/流那样纯粹“推”，也不像生成器那样纯粹“拉”。

结论

这篇文章涵盖了很多内容。我们介绍了核心原语，并触及了细粒度响应性的定义特征，包括依赖关系解析和同步执行。
如果主题看起来还不完全清楚，也没关系。你可复习文章并尝试修改示例。这些旨在以最简单的方式展示这些想法。但这其实已经是关于细粒度响应式的大部分内容。通过一些练习，你也将能够了解如何以精细的方式对数据进行建模。