手撸一个响应式库

> 本文来源于Solid.js作者**Ryan Carniato，讲解如何手撸一个响应式库。**

在之前的文章「细粒度响应式实践介绍」中，我通过示例讲述了细粒度响应式背后的理念。现在我们来聊聊如何自己创建一个响应式库。

当你看到它运行时，总有一些东西看起来有点神奇，但从机制上来说它并不那么复杂。反应性之所以如此神奇，是因为一旦到位，即使在动态场景下，它也会自行发挥作用。真正的声明式方法，因为只要遵守约定，如何实现并不重要。

我们手撸的这个响应式库并不会像MobX、Vue、Solid那样复杂，但它应该能作为一个很好的示例来了解其工作原理。

Signal

Signal是响应式系统的核心，也是我们开始撸的正确起点。它们包含一个 getter 和 setter，所以我们可以从这样的开始：

```
export function createSignal(value) {
  const read = () => value;
  const write = (nextValue) => value = nextValue;
  return [read, write];
}
```

这还没有做太多事情，但我们可以看到我们现在有一个简单的容器来保存我们的值。

```
const [count, setCount] = createSignal(3);
console.log("Initial Read", count());

setCount(5);
console.log("Updated Read", count());

setCount(count() * 2);
console.log("Updated Read", count());
```

```
Initial Read 3
Updated Read 5
Updated Read 10
```

那么我们还缺少什么呢？管理订阅。Signal是事件发射器。

```
const context = [];

function subscribe(running, subscriptions) {
  subscriptions.add(running);
  running.dependencies.add(subscriptions);
}

export function createSignal(value) {
  const subscriptions = new Set();

  const read = () => {
    const running = context[context.length - 1];
    if (running) subscribe(running, subscriptions);
    return value;
  };

  const write = (nextValue) => {
    value = nextValue;

    for (const sub of [...subscriptions]) {
      sub.execute();
    }
  };
  return [read, write];
}
```

这里有些需要解释的地方，有2个事情我们需要管理。在文件最上方是一个全局的context栈，用来追踪任意运行的Reaction 和 Derivation。除此之外，每个Signal有它自己的订阅者列表（subscriptions）。

这2件事构成了自动依赖性跟踪的整体基础。执行中的Reaction 或 Derivation将自身推送到context堆栈上。它将被添加到执行期间读取任何Siganl的订阅列表（subscriptions）中。我们还将Signal添加到运行上下文中用于进行清理操作，这将在下一节中介绍。

最后，在 Signal 的write 中，除了更新值之外，我们还要执行所有的订阅。我们克隆了该列表，以便在此执行过程中添加的新订阅不会影响此运行。

这是我们完成的Signal，但它只是等式的一半。

Reaction 和 Derivation

既然你已经看到了一半，你也许就能猜出另一半是什么样子了。让我们创建一个基本的Reaction（或Effect）。

```
function cleanup(running) {
  for (const dep of running.dependencies) {
    dep.delete(running);
  }
  running.dependencies.clear();
}

export function createEffect(fn) {
  const execute = () => {
    cleanup(running);
    context.push(running);
    try {
      fn();
    } finally {
      context.pop();
    }
  };

  const running = {
    execute,
    dependencies: new Set()
  };

  execute();
}
```

我们在这里创建的是我们推送到context中的对象。它具有Reaction监听的依赖（Signal）列表以及我们跟踪和重新运行的函数表达式。

每个执行周期我们都会从所有Signal中取消订阅Reaction并清除依赖项列表以开始新的。这就是我们存储反向链表的原因。这允许我们在每次运行时动态创建依赖项。然后我们将 Reaction 压入堆栈并执行用户提供的函数。

这 50 行代码可能看起来不多，但我们现在可以用它重新创建上一篇文章中的第一个演示。

```
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);
```

```
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
```

添加简单的Derivation并不需要更多的工作，只需使用与 createEffect 几乎相同的代码。在 MobX、Vue 或 Solid 等真正的响应式库中，我们将构建推/拉机制并跟踪图表以确保我们没有做额外的工作，但出于演示目的，我将使用Reaction。

> 注意：如果你对这个推/拉算法的实现感兴趣，我推荐你阅读Becoming Fully Reactive: An in-depth explanation of MobX

```
export function createMemo(fn) {
  const [s, set] = createSignal();
  createEffect(() => set(fn()));
  return s;
}
```

有个createMemo的实现，让我们重新用它创建带条件渲染的示例：

```
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
4. Set showFullName: true
My name is John Legend
```

正如你所看到的，因为当我们不再监听lastName时，我们会重新构建依赖关系图，因此在lastName更新时，不会重新执行Derivation。

结论

这些只是响应式系统的基础知识。当然，我们的库没有批处理、自定义处理方法或防止无限递归的保护措施，而且也不是没有故障的。但它包含了所有核心部分。这就是 2010 年代初期的 类似 KnockoutJS 等库的工作原理。

出于所有提到的原因，我不建议使用这个库。但只需大约 50 行代码，你就拥有了一个简单的反应式库的所有要素。当你考虑可以用它建模多少行为时，你应该更理解为什么像 Svelte 和 Solid 这样的库以及编译器可以生成如此小的包。

这么少的代码却蕴藏着巨大的力量。你确实可以用它来解决各种问题。**只需几行代码即可成为你选择的框架的状态库，并且只需几十行即可成为框架本身。**

希望通过这个练习，你现在可以更好地理解和欣赏细粒度响应式库中的自动跟踪如何工作，并且我们已经揭开了一些魔法。

如果你对 Solid 是如何利用它并创建一个完整的渲染库感兴趣。请查看 SolidJS: Reactivity to Rendering。

FAQ

示例没有类型，看起来没有头绪?

是的，很多人都会有这样的看法。有时，清晰度来自于代码里变量所选择的名称。做一些改变即可让示例更加易于理解：

Change #1:

```
// A Dependency has many different Subscribers depending on it
// A particular Subscriber has many Dependencies it depends on
type Dependency = Set<Subscriber>;
type Subscriber = {
  execute(): void;
  dependencies: Set<Dependency>;
};
```

- Dependency。每个Dependency有许多个不同的订阅者依赖它。
- Subscriber。每个Subscriber依赖于许多不同的依赖项。

Change #2:

```
  const subscriptions: Dependency = new Set();
```

Change #3:

```
function cleanup(running: Subscriber) {
```

Change #4:

```
  const running: Subscriber = {
    execute,
    dependencies: new Set(),
  };
```

改动完之后代码如下：

```
// https://www.youtube.com/watch?v=vHy7GRpTpm8&list=LL&index=3&t=514s
// https://codesandbox.io/s/0xyqf?file=/reactive.js:0-1088

// A Dependency has many different Subscribers depending on it
// A particular Subscriber has many Dependencies
type Dependency = Set<Subscriber>;
type Subscriber = {
  execute(): void;
  dependencies: Set<Dependency>;
};

type Signal<T> = [() => T, (value: T) => void];

const context: Subscriber[] = [];

function createSignal<T>(value: T): Signal<T> {
  const subscriptions: Dependency = new Set();
```

```
18
19    const read = (): T => {
20      const running = context[context.length - 1];
21
22      if (running) {
23        subscriptions.add(running);
24
25        running.dependencies.add(subscriptions);
26      }
27
28      return value;
29    };
30
31    const write = (nextValue: T) => {
32      value = nextValue;
33
34      for (const sub of [...subscriptions]) {
35        sub.execute();
36      }
37    };
38
39    return [read, write];
40  }
41
42  function cleanup(running: Subscriber) {
43    for (const dep of running.dependencies) {
44      dep.delete(running);
45    }
46
47    running.dependencies.clear();
48  }
49
50  function createEffect(effect: () => void) {
51    const execute = () => {
52      cleanup(running);
53
54      context.push(running);
55
56      try {
57        effect();
58      } finally {
59        context.pop();
60      }
61    };
62
63    const running: Subscriber = {
64      execute,
```

```
    dependencies: new Set(),
  };

  execute();
}

function createMemo<T>(fn: () => T): () => T {
  const [read, write] = createSignal<T>(null as any);

  createEffect(() => write(fn()));

  return read;
}

// util
function log(...args: unknown[]) {
  console.log(args.join(' '));
}

// index
log('1. Create');
const [firstName, _setFirstName] = createSignal('John');
const [lastName, setLastName] = createSignal('Smith');
const [showFullName, setShowFullName] = createSignal(true);

const displayName = createMemo(() =>
  showFullName() ? `${firstName()} ${lastName()}` : firstName()
);

createEffect(() => log('My name is', displayName()));

log('2. Set showFullName: false ');
setShowFullName(false);

log('3. Change lastName');
setLastName('Legend');

log('4. Set showFullName: true');
setShowFullName(true);

log('5. Change lastName while showFullName: true');
setLastName('Who');
```

完整的代码可在TypeScipt Playground体验。

为什么代码里Clean up是在下次执行 之前做而不是之后?

> 比如像下面这样：

```
export function createEffect(fn) {
  const execute = () => {
    context.push(running);
    try {
      fn();
    } finally {
      context.pop();
      cleanup(running);
    }
  };

  const running = {
    execute,
    dependencies: new Set()
  };

  execute();
}
```

被clean up的实际上是上一次的，而不是这次的，看起来类似这样

```
// run 1
cleanup(); // no-op
fn(); // execute 1

// run 2
cleanup(); // cleanup 1
fn(); // execute 2

// run 3
cleanup(); // cleanup 2
fn(); // execute 3

// parent removes child
cleanup(); // cleanup 3
```

为什么context要使用栈的形式?

> 把context换成普通的字符串看起来也能行，这是为什么呢? Codesandbox

```
let context = ""
```

```
const read = () => {
    const running = context;
    if (running) subscribe(running, subscriptions);
    return value;
};
```

```
const execute = () => {
    cleanup(running);
    context = running;
    try {
      fn();
    } finally {
      context = "";
    }
};
```

A： 因为嵌套响应性。一般来说，你可以将Effect或Derivition嵌套在它自己内部。这对于 Solid 的渲染方式很重要，但我认为这个简单的示例不需要。

> Q：在context为字符串的情况下，使用嵌套看起来也没啥问题呢？Codesandbox

```
log("1. Create");
const [value, setValue] = createSignal(0);
const [name, setName] = createSignal("jack");

createEffect(() => {
  log("name is ", name());
  createEffect(() => {
    log("value is ", value());
  });
});

setValue(1);
setName("vivi");
```

A: 上述例子中，如果在嵌套Effect的后面进行订阅，则将不会获得追踪：

```
createEffect(() => {
  createEffect(() => {
    log("value is ", value());
  });
  log("name is ", name());
});
```

请记住，这篇文章的实现只是一个简单的近似。我实际上并没有在我的库中使用数组来实现，而是只是用它来存储外部上下文，然后恢复它。

参考

- https://dev.to/ryansolid/building-a-reactive-library-from-scratch-1i0p