实现数据响应式

我们的 `mini-vue` 现在能展示初始页面，但是如果数据发生变化，还不会更新。

以前我们 `jQuery`，数据变化之后，我需要手动执行更新函数，里面写上各种 dom 操作。这是心智负担，也是工作量。前面我们说过 vue 的重要目标是 `数据驱动`，即数据变了，视图自动执行更新，用户不需要操心，要做到这一点需要修改数据的时候能检测到变化，然后自动调用组件更新函数。

这就需要实现 `数据响应式`。

数据响应式

下面我们实现一个数据响应式的方法 `reactive()`：

JavaScript

```
// reactive返回传入obj的代理对象，值更新时使app更新
export function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      return Reflect.get(target, key);
    },
    set(target, key, value) {
      return Reflect.set(target, key, value);
    },
    deleteProperty(target, key) {
      return Reflect.deleteProperty(target, key);
    },
  });
}
```

> Reflect:
> https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect

添加一些单测验证

```JavaScript
import { reactive } from "../index";

test("reactive should work", () => {
  const original = {foo: 'foo'}
  const observed = reactive(original)
  // 代理对象是一个全新对象
  expect(observed).not.toBe(original)
  // 能够访问代理对象的属性
  expect(observed.foo).toBe('foo')
  // 能够修改代理对象的属性
  observed.foo = 'fooooooo~'
  expect(original.foo).toBe('fooooooo~')
  // 能够新增代理对象的属性
  observed.bar = 'bar'
  expect(original.bar).toBe('bar');
  // 能够删除代理对象的属性
  delete observed.bar
  expect(original.bar).toBe(undefined);
});
```

使用 reactive

在我们的 demo 中，data 函数返回的对象需要做响应式处理，只有这样使用这些数据的组件 render 函数才能有机会和数据之间产生依赖关系。我们对 createRenderer 中的 render 函数做一个改造：

```JavaScript
// runtime-core/index.js
const render = (rootComponent, selector) => {
    const container = options.querySelector(selector);
    // 对data做响应式处理
    const observed = reactive(rootComponent.data());

    // 为组件定义一个更新函数
    const componentUpdateFn = () => {
      const el = rootComponent.render.call(observed);
```

```
      options.insert(el, container);
    };

    // 初始化执行一次
    componentUpdateFn();
}
```

可以测试一下，可以像之前一样正常运行。

依赖收集

但是前面改造并没有什么意义，因为如果数据变化，更新函数 componentUpdateFn 并不会再次执行。想要做到这一点，需要在 reactive()中添加依赖跟踪和触发的逻辑。

JavaScript

```
// reactivity/index.js
export function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      const value = Reflect.get(target, key);
      // 依赖跟踪
      track(target, key)
      return value
    },
    set(target, key, value) {
      const result = Reflect.set(target, key, value);
      // 依赖触发
      trigger(target, key)
      return result
    },
    deleteProperty(target, key) {
      const result = Reflect.deleteProperty(target, key);
      // 依赖触发
      trigger(target, key)
      return result
    },
  });
}
```

这里的 `track()` 就是依赖跟踪，目的是建立 `target`， `key` 和 `activeEffect` 之间的依赖关系。

JavaScript

```
// reactivity/index.js
const targetMap = new WeakMap()

function track(target, key) {
  if (activeEffect) {
    let depsMap = targetMap.get(target)

    if (!depsMap) {
      targetMap.set(target, (depsMap = new Map()))
    }

    let deps = depsMap.get(key)
    if (!deps) {
      depsMap.set(key, (deps = new Set()))
    }

    deps.add(activeEffect)
  }
}
```

这里面出现的 activeEffect 就是触发本次 get 调用的函数，在我们 demo 中就是 componentUpdateFn。

我们需要特地声明它，并添加一个设置函数：

JavaScript

```
// reactivity/index.js
let activeEffect;

// effect提供给其他模块用来设置activeEffect
export function effect(fn) {
  activeEffect = fn
}
```

最后完成依赖触发 `trigger`，它根据传入 `target`，`key` 获取相关联的所有副作用函数并执行它们。

JavaScript

```
// reactivity/index.js
function trigger(target, key) {
  const depsMap = targetMap.get(target)

  if (depsMap) {
    const deps = depsMap.get(key)

    if (deps) {
      deps.forEach(dep => dep())
    }
  }
}
```

下面在 demo 应用一下，触发依赖收集

JavaScript

```
// src/mini-vue/runtime-core/index.js
import { effect } from "../reactivity/index";

const render = (rootComponent, selector) => {
    // ...

    // 指定活动的副作用为该更新函数
    effect(componentUpdateFn);

    componentUpdateFn();
};
```

我们改一下数据

JavaScript

```
// src/main.js
createApp({
  mounted() {
    setTimeout(() => {
```

```
      this.title = 'wow, data change!'
    }, 2000);
  }
}).mount('#app')
```

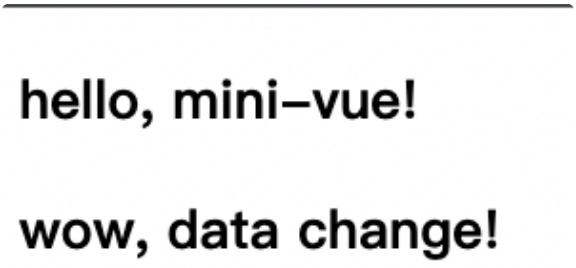
让 mounted 生效

JavaScript

```
// runtime-core/index.js
componentUpdateFn();

// 执行mounted
if (rootComponent.mounted) {
  rootComponent.mounted.call(observed)
}
```

大功告成，测试一下咱们的响应式是否生效！是不是很奇怪多了一行?



这是因为我们更新知识再次调用 componentUpdateFn，并没有清除之前剩下的内容。我们添加一个清除的方法：

JavaScript

```
// src/mini-vue/runtime-dom/index.js
const rendererOptions = {
  setElementText(el, text) {
    el.textContent = text;
  },
};
```

调用该方法

JavaScript

```
// src/mini-vue/runtime-core/index.js
const componentUpdateFn = () => {
  const el = rootComponent.render.call(observed);
```

```
  // 清除之前内容
  options.setElementText(container, '')
  options.insert(el, container);
};
```

关系图

最后理一下依赖收集过程中的各种关系，如下图：