# React 实现自动上报 pv/click 的埋点 hooks

## 前言

此篇文章笔者将围绕 React 中自定义 hooks 给大家讲讲自定义 hooks 的概念以及我们要如何来设计编写自定义 hooks...

## 介绍

自定义 hooks 是基于 React Hooks 的一个拓展，我们可以根据业务需求制定满足业务需要的组合 hooks，更注重的是逻辑单元。怎样把一段逻辑封装起来，做到复用，这才是自定义 hooks 的初衷。

自定义 hooks 也可以说是 React Hooks 的聚合产物，其内部有一个或者多个 React Hooks 组成，用于解决一些复杂逻辑。

一个传统自定义 hooks 长下面这个样子：

```js 复制代码
function useXXX(参数A, 参数B, ...) {
  /*
    实现自定义 hooks 逻辑
    内部应用了其他 React Hooks
  */
  return [xxx, ...]
}
```

使用：

```js 复制代码
const [xxx, ...] = useXXX(参数A, 参数B, ...)
```

自定义 hooks **参数** 可能是以下内容：

> - `hooks` 初始化值
> - 一些副作用或事件的回调函数
> - 可以是 `useRef` 获取的 `DOM` 元素或者组件实例
> - 不需要参数

自定义 hooks **返回值** 可能是以下内容：

> - 负责渲染视图获取的状态
> - 更新函数组件方法，本质上是 `useState` 或者 `useReducer`
> - 一些传递给子孙组件的状态
> - 没有返回值

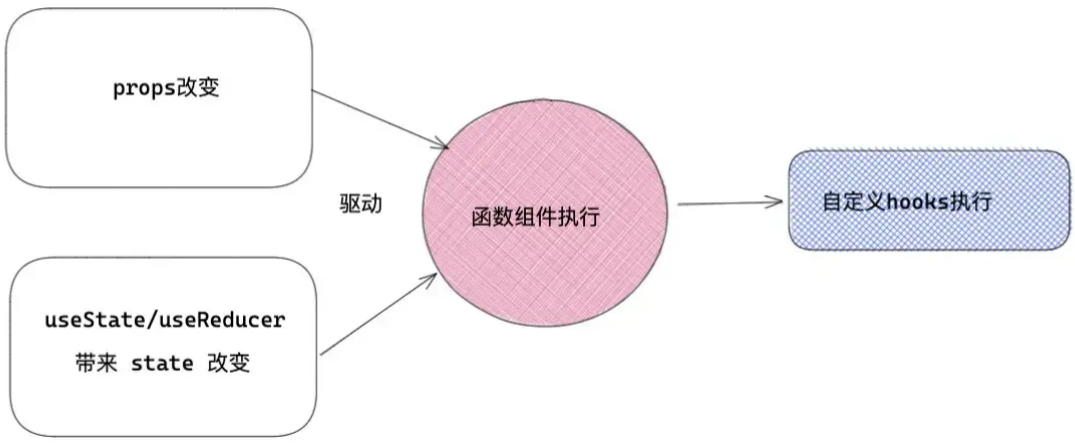
## 特性

首先我们要明白，开发者编写的自定义 hooks 本质上就是一个函数，而且在函数组件中被执行。**自定义 hooks 驱动本质上就是函数组件的执行**。

### 驱动条件

自定义 hooks 的驱动条件主要有两点：

> 1. `props` 改变带来的函数组件执行。
> 2. `useState` 或 `useReducer` 改变 `state` 引起函数组件的更新。




### 顺序原则

自定义 hooks 内部至少要有一个 React Hooks，那么自定义 hooks 也同样要遵循 React Hooks 的规则，**不能放在条件语句中，而且要保持执行顺序的一致性。** 这是为什么呢?

> 这是因为在更新过程中，如果通过 if 条件语句，增加或者删除 hooks，那么在复用 hooks 的过程中，会产生复用 hooks 状态和当前 hooks 不一致的问题。所以在开发时一定要注意 hooks 顺序的一致性。

## 实践

接下来我们来实现一个能够 **自动上报 页面浏览量 | 点击时间 的自定义 hooks** -- `useLog` 。

通过这个自定义 hooks，来 **控制监听 DOM 元素，分清楚依赖关系。**

编写自定义 hooks：

```js 复制代码
export const LogContext = createContext({});

export const useLog = () => {
  /* 定义一些公共参数 */
  const message = useContext(LogContext);
  const listenDOM = useRef(null);

  /* 分清依赖关系 */
  const reportMessage = useCallback(
    function (data, type) {
      if (type === "pv") {
        // 页面浏览量上报
        console.log("组件 pv 上报", message);
      } else if (type === "click") {
        // 点击上报
        console.log("组件 click 上报", message, data);
      }
    },
    [message]
  );

  useEffect(() => {
    const handleClick = function (e) {
      reportMessage(e.target, "click");
    };
    if (listenDOM.current) {
      listenDOM.current.addEventListener("click", handleClick);
    }
```

```
    return function () {
      listenDOM.current &&
        listenDOM.current.removeEventListener("click", handleClick);
    };
  }, [reportMessage]);

  return [listenDOM, reportMessage];
};
```

在上面的代码中，使用到了如下4个 React Hooks：

> - 使用 `useContext` 获取埋点的公共信息，当公共信息改变时，会统一更新。
> - 使用 `useRef` 获取 DOM 元素。
> - 使用 `useCallback` 缓存上报信息 `reportMessage` 方法，里面获取 `useContext` 内容。把 `context` 作为依赖项，当依赖项发生改变时，重新声明 `reportMessage` 函数。
> - 使用 `useEffect` 监听 DOM 事件，把 `reportMessage` 作为依赖项，在 `useEffect` 中进行事件绑定，返回的销毁函数用于解除绑定。

**依赖关系**：`context` 发生改变 -> 让引入 `context` 的 `reportMessage` 重新声明 -> 让绑定 DOM 事件监听的 `useEffect` 里面能够绑定最新的 `reportMessage`

使用自定义 hooks：

```js 复制代码
import React, { useState } from "react";
import { LogContext, useLog } from "./hooks/useLog";

const Home = () => {
  const [dom, reportMessage] = useLog();
  return (
    <div>
      {/* 监听内部点击 */}
      <div ref={dom}>
        <button> 按钮 1 (内部点击) </button>
        <button> 按钮 2 (内部点击) </button>
        <button> 按钮 3 (内部点击) </button>
      </div>
      {/* 外部点击 */}
      <button
        onClick={() => {
          console.log(reportMessage);
        }}
      >
```

```
        外部点击
      </button>
    </div>
  );
};
// 阻断 useState 的更新效应
const Index = React.memo(Home);

const App = () => {
  const [value, setValue] = useState({});
  return (
    <LogContext.Provider value={value}>
      <Index />
      <button onClick={() => setValue({ cat: "小猫", color: "棕色" })}>
        点击
      </button>
    </LogContext.Provider>
  );
};

export default App;
```

如上，当 `context` 发生改变时，能够达到正常上报的效果。小细节：使用 `React.memo` 来阻断 `App` 组件改变 `state` 给 `Home` 组件带来的更新效应。

## 效果

刚开始时依次点击按钮1，2，3，效果如下：

```
组件 click 上报 ▸{}   <button> 按钮 1 (内部点击) </button>   useLog.jsx:24
组件 click 上报 ▸{}   <button> 按钮 2 (内部点击) </button>   useLog.jsx:24
组件 click 上报 ▸{}   <button> 按钮 3 (内部点击) </button>   useLog.jsx:24
>
```



点击点击按钮后，再依次点击按钮1，2，3时，效果如下：