14–2 如何实现 useCallback

`useCallback` 是一个允许你在多次渲染中缓存**函数**的 React Hook。

```JavaScript
const cachedFn = useCallback(fn, dependencies)
```

参数

- `fn`：想要缓存的函数。此函数可以接受任何参数并且返回任何值。React 将会在初次渲染而非调用时返回该函数。当进行下一次渲染时，如果 `dependencies` 相比于上一次渲染时没有改变，那么 React 将会返回相同的函数。否则，React 将返回在最新一次渲染中传入的函数，并且将其缓存以便之后使用。React 不会调用此函数，而是返回此函数。你可以自己决定何时调用以及是否调用。
- `dependencies`：有关是否更新 `fn` 的所有响应式值的一个列表。响应式值包括 props、state，和所有在你组件内部直接声明的变量和函数。如果你的代码检查工具 配置了 React，那么它将校验每一个正确指定为依赖的响应式值。依赖列表必须具有确切数量的项，并且必须像 `[dep1, dep2, dep3]` 这样编写。React 使用 `Object.is` 比较每一个依赖和它的之前的值。

返回值

在初次渲染时，`useCallback` 返回你已经传入的 `fn` 函数

在之后的渲染中，如果依赖没有改变，`useCallback` 返回上一次渲染中缓存的 `fn` 函数；否则返回这一次渲染传入的 `fn`。

基础使用

TypeScript

```
function FunctionComponent() {
  const [count1, setCount] = useReducer((x) => x + 1, 0);
  const [count2, setCount2] = useState(0);

  // const addClick = () => {
  //   //   ajax('xxx/'+count1)
  //   let sum = 0;
  //   for (let i = 0; i < count1; i++) {
  //     sum += i;
  //   }
  //   return sum;
  // };

  const addClick = useCallback(() => {
    let sum = 0;
    for (let i = 0; i < count1; i++) {
      sum += i;
    }
    return sum;
  }, [count1]);

  const expensive = useMemo(() => {
    //只有addClick变化，这里才重新执行
    console.log("compute");
    return addClick();
  }, [addClick]);

  return (
    <div className="border">
      <h1>函数组件</h1>
      <p>{expensive}</p>
      <button onClick={() => setCount()}>{count1}</button>
```

```
      <button onClick={() => setCount2(count2 + 1)}>{count2}</button>

      {/* <Child addClick={addClick} /> */}
    </div>
  );
}

// // memo 允许组件在 props 没有改变的情况下跳过重新渲染。
// const Child = memo(({ addClick }: { addClick: () => number }) => {
//   console.log("child render"); //sy-log
//   return (
//     <div>
//       <h1>Child</h1>
//       <button onClick={() => console.log(addClick())}>add</button>
//     </div>
//   );
// });
```

实现 useCallback

TypeScript

```
export function useCallback<T>(callback: T, deps: Array<any> | void |
  const hook = updateWorkInProgressHook();

  const nextDeps = deps === undefined ? null : deps;

  const prevState = hook.memoizedState;
  // 检查依赖项是否发生变化
  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps = prevState[1];
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        // 依赖项没有变化，返回上一次缓存的callback
        return prevState[0];
      }
    }
  }

  hook.memoizedState = [callback, nextDeps];
```

```
  return callback;
}
```