我们应该如何优雅的处理 React 中受控与非受控

引言

大家好，我是19组清风。有段时间没有和大家见面了，最近因为有一些比较重要的事情（陪女朋友和换了新公司）在忙碌所以销声匿迹了一小段时间，

后续会陆陆续续补充之前构建 & 编译系列中缺失的部分，提前预祝大伙儿圣诞节快乐！

受控 & 非受控

今天来和大家简单聊聊 React 中的**受控**和**非受控**的概念。

提到受控和非受控相信对于使用过 React 的朋友已经老生常谈了，在开始正题之前惯例先和大家聊一些关于受控 & 非受控的基础概念。

当然，已经有基础的小伙伴可以略过这个章节直接往下进行。

受控

> 在 HTML 中，表单元素（如 `<input>`、 `<textarea>` 和 `<select>`）通常自己维护 state，并根据用户输入进行更新。而在 React 中，可变状态（mutable state）通常保存在组件的 state 属性中，并且只能通过使用 ➤ `setState()` 来更新。
>
> 我们可以把两者结合起来，使 React 的 state 成为“唯一数据源”。渲染表单的 React 组件还控制着用户输入过程中表单发生的操作。被 React 以这种方式控制取值的表单输入元素就叫做“受控组件”。

上述的描述来自 React 官方文档，其实受控的概念也非常简单。**通过组件内部可控的 state 来控制组件的数据改变从而造成视图渲染。**

这种模式更像是 Vue 中在表单元素中的常用处理模式，举一个简单的例子，比如：

```ts
import { FC } from 'react';

interface InputProps<T = string> {
  value: T;
  onChange: (value?: T) => void;
}

const Input: FC<InputProps> = (props) => {
  const { onChange, value = '', ...rest } = props;

  const _onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const value = e.target.value;
    onChange && onChange(value);
  };

  return <input value={value} onChange={_onChange} {...rest} />;
};

export default Input;
```

上述的代码非常简单，我们声明了一个名为 Input 的自定义输入框组件，**但是 Input 框中的值是由组件中的 `controllerState` 进行控制的**。

这也就意味着，如果组件外部的状态并不改变（这里指组件的 props 中的 value）时，即使用户在页面上展示的 input 如何输入 input 框中渲染的值也是不会发生任何改变的。

当然，无论是通过 props 还是通过 state 只要保证表单组件的 value 接受的是一个非 undefined 的状态值，那么该表单元素就可以被称为受控（表单中的值是通过组件状态控制渲染的）。

非受控

既然存在受控组件，那么一定存在相反非受控的概念。

> 在大多数情况下，我们推荐使用 ➤ 受控组件 来处理表单数据。在一个受控组件中，表单数据是由 React 组件来管理的。另一种替代方案是使用非受控组件，这时表单数据将交由 DOM 节点来处理。

熟悉 Ant-Design 等存在表单校验的 React 组件库的朋友，可以稍微回忆下它们的表单使用。

ts 复制代码

```ts
// ant-design 官方表单使用示例
import React from 'react';
import { Button, Checkbox, Form, Input } from 'antd';

const App: React.FC = () => {
  const onFinish = (values: any) => {
    console.log('Success:', values);
  };

  const onFinishFailed = (errorInfo: any) => {
    console.log('Failed:', errorInfo);
  };

  return (
    <Form
      name="basic"
      labelCol={{ span: 8 }}
      wrapperCol={{ span: 16 }}
      initialValues={{ remember: true }}
      onFinish={onFinish}
      onFinishFailed={onFinishFailed}
      autoComplete="off"
    >
      <Form.Item
        label="Username"
        name="username"
        rules={[{ required: true, message: 'Please input your username!' }]}
      >
        <Input />
      </Form.Item>

      <Form.Item
        label="Password"
        name="password"
        rules={[{ required: true, message: 'Please input your password!' }]}
      >
        <Input.Password />
      </Form.Item>

      <Form.Item name="remember" valuePropName="checked" wrapperCol={{ offset: 8, span: 16 }}>
        <Checkbox>Remember me</Checkbox>
      </Form.Item>

      <Form.Item wrapperCol={{ offset: 8, span: 16 }}>
        <Button type="primary" htmlType="submit">
          Submit
        </Button>
      </Form.Item>
```

```
    </Form>
  );
};

export default App;
```

虽然说 React 官方推荐使用受控组件来处理表单数据，但如果每一个表单元素都需要使用方通过受控的方式来使用的话对于调用方来说的确是过于繁琐了。

所以大多数 React Form 表单我们都是通过非受控的方式来处理，那么所谓的非受控究竟是什么意思呢。我们一起来看看。

所谓非受控简单来说也就指的是**表单元素渲染并不通过内部状态数据的改变而渲染，而是交由源生表单内部的 State 来进行自由渲染。**

这其实是一种和受控组件完全相反的概念，比如：

ts 复制代码

```ts
import { FC } from 'react';

interface InputProps<T = string> {
  defaultValue?: T;
}

const Input: FC<InputProps> = (props) => {
  const { defaultValue } = props;

  return <input defaultValue={defaultValue} />;
};

export default Input;
```

上述我们重新定义了一个名为 Input 的非受控组件，此时当你在使用该 Input 组件时，由于 defaultValue 仅会在 input 元素初始化时进行一次数据的初始化。

之后当用户在页面上的 input 元素中输入任何值表单值都会跟随用户输入而实时变化而并不受任何组件状态的控制，这就被称为非受控组件。

当然相较于受控组件获取值的方式，非受控组件获取的方式就会稍微显得繁琐一些，非受控组件需要通过组件实例也就是配合 ref 属性来获取对应组件/表单中的值，比如：

ts 复制代码

```ts
import { FC, useRef } from 'react';
```

```tsx
interface InputProps<T = string> {
  defaultValue?: T;
}

const Input: FC<InputProps> = (props) => {
  const { defaultValue } = props;

  const instance = useRef<HTMLInputElement>(null);

  const getInstanceValue = () => {
    if (instance.current) {
      alert(instance.current.value);
    }
  };

  return (
    <div>
      <input ref={instance} defaultValue={defaultValue} />

      <button onClick={() => getInstanceValue()}>获取input中的值</button>
    </div>
  );
};

export default Input;
```

上边的代码中，我们需要获取 unController input 的值。需要通过 ref 获得对应 input 的实例之后获得 input 中的值。

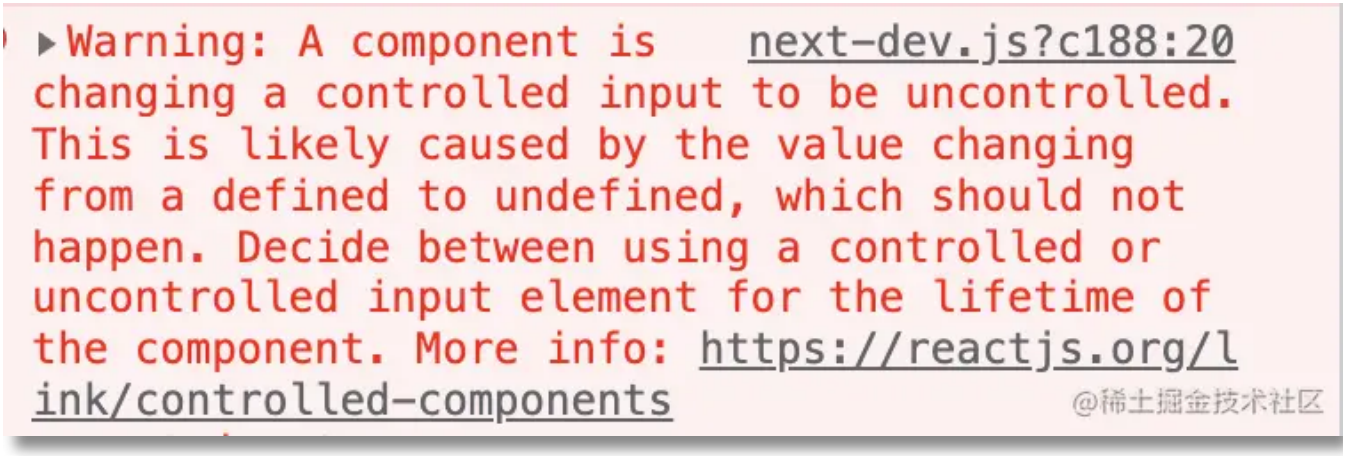
重要区分点

上边我们聊到了 React 中的受控和非受控的概念，在 React 中区分受控组件和非受控组件有一个最重要的 point 。

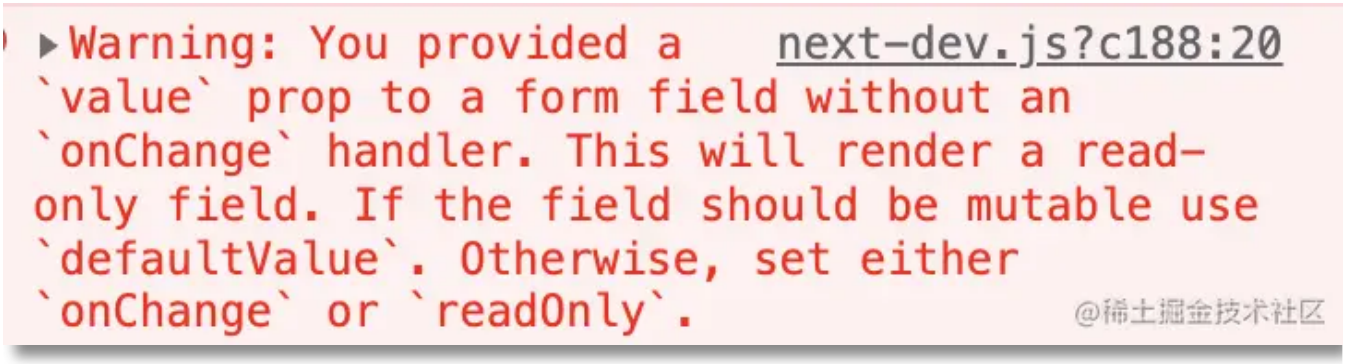
在 React 中当一个表单组件，我们显式的声明了它的 value （并不为 undefined 或者 null 时）那么该表单组件即为受控组件。

相反，当我们为它的 value 传递为 undefined 或者 null 时，那么该组件会变为非受控(unController)组件。

相信使用过 React 的小伙伴的同学或多或少都碰到过相关的 Warning ：



input 组件的 value 从非 undefeind 变为 undefined （从受控强行改变为非受控组件），这是不被 React 推荐的做法。



当并未受控组件提供 onChange 选项时，此时也就意味着用户永远无法改变该 input 中的值。

当然，还有诸如此类非常多的 Warining 警告。相信大家在搞清楚受控 & 非受控的概念后这些对于大家来说都是小菜一碟。

> 当然在绝大多数社区组件库中都是将 undefined 作为了区分受控和非受控的标志。

useMergedState

在我们了解了 React 中的受控 & 非受控的基础概念后，趁热打铁我们再来聊聊 rc-util 中的一个 useMergedState Hook。

这个 Hook 其实并没有多少难度，大家完全不用担心看不懂它的代码哈哈。

在开始阅读它的代码之前，我会一步一步带你了解它的运作方式。

作用

首先，我们先来看看 useMergedState 这个 Hook 的作用。

通常在我们开发一些表单组件时，需要基于多层属性来传递 props 给基层的 input 之类的表单控件。

由于是公用的基础表单控件，所以无疑仅提供受控或者非受控单一的一种方式来说对于调用者并不是那么优雅和便捷。

所以此时，针对于表单控件的开发我们需要提供给开发者受控和非受控两种方式的支持。

类似 Ant-Design 中的 Input 组件。它既接收显示传入 value 和 onChange 的组合方式，同时也支持传入 defaultValue 的非受控方式实现。

所谓的 useMergedState 即是这样的一个作用：**通过该 Hook 你可以自由定义表单控件的受控和非受控状态。**

这么说其实稍微有点含糊，我们先来看看它的类型定义吧：

```ts
export default function useMergedState<T, R = T>(defaultStateValue: T | (() => T), option?: {
    defaultValue?: T | (() => T);
    value?: T;
    onChange?: (value: T, prevValue: T) => void;
    postState?: (value: T) => T;
}): [R, Updater<T>];
```

这个 hook 接收两个形参，分别为 `defaultStateValue` 和 `option`：

- defaultStateValue 这个参数表示传入的默认 value 值，当传入参数不存在 option 中的 value 或者 defaultValue 时就会 defaultStateValue 来作为初始值。
- option
  - defaultValue 可选，表示接收非受控的初始化默认值，它的优先级高于 defaultStateValue。
  - value 可选，表示作为受控时的 value props，它的优先级高于 defaultValue 和 defaultStateValue。
  - onChange 可选，当内部值改变后会触发该函数。
  - postState 可选，表示对于传入值的 format 函数。

乍一看其实挺多的参数，相信没有了解过该函数的同学多多少少都会有些懵。

没关系，接下来我们会先抛开这个 Hook ，先自己来一步一步尝试如何来实现这样的组合受控 & 非受控的业务 Hook。

实现

接下来我们就先按照自己的思路来实现这个 Hook 。

首先，我们以一个 Input 组件为为例，假使我们需要编写一个 Input 输入框组件。

```ts
interface TextField
  extends Omit<InputHTMLAttributes<HTMLInputElement>, 'onchange'> {
  /**
   * onChange 函数
   */
  onChange: (e: React.ChangeEvent<HTMLInputElement>) => void;
}

const TextField: React.FC<TextField> = (props) => {
  const { value, defaultValue, onChange, ...rest } = props;

  return <input />;
};
```

非受控处理

上述，我们编写了一个基础的 Input 组件的模版。

此时，让我们先来考虑传入该组件的非受控处理，也就是所谓的接受 defaultValue 作为非受控的 props 传入。

我们利用 defaultValue 作为 input 框非受控的值传递，以及配合 onChange 仅做事件的传递。

```ts
interface TextField
  extends Omit<InputHTMLAttributes<HTMLInputElement>, 'onchange'> {
  /**
   * onChange 函数
   */
  onChange: (e: React.ChangeEvent<HTMLInputElement>) => void;
}
```

```ts
const TextField: React.FC<TextField> = (props) => {
  const { defaultValue, onChange, ...rest } = props;

  return <input defaultValue={defaultValue} onChange={onChange} {...rest} />;
};
```

看起来非常简单对吧，此时当调用者使用我们的组件时。只需要传入 defaultValue 的值就可以使用非受控状态的 input 。

受控处理

上述我们用非常简单的代码实现了非受控的 Input 输入框，此时我们再来看看如何兼顾受控状态的值。

我们提到过，在 React 中如果需要受控状态的表单控件是需要显式传入 value 和对应的 onChange 作为配合的，此时很容易我们想到这样改造我们的组件：

ts 复制代码

```ts
interface TextField
  extends Omit<InputHTMLAttributes<HTMLInputElement>, 'onchange'> {
  /**
   * onChange 函数
   */
  onChange: (e: React.ChangeEvent<HTMLInputElement>) => void;
}

const TextField: React.FC<TextField> = (props) => {
  const { defaultValue, value, onChange, ...rest } = props;

  return (
    <input
      value={value}
      defaultValue={defaultValue}
      onChange={onChange}
      {...rest}
    />
  );
};

export default TextField;
```

有些同学会很容易想到我们将 defaultValue 和 value 同时进行透传进来不就完成了吗。

没错，这样的确可以完成基础的需求。可是这对于一个组件来说并不是一种良好的做法，假如调用方这样使用我们的组件：

```ts
export default function App({ Component, pageProps }: AppProps) {
  const [state, setState] = useState('');

  const onChange = (e: ChangeEvent<HTMLInputElement>) => {
    const value = e.target.value;
    setState(value);
  };

  return (
    <TextField value={state} defaultValue={'hello world'} onChange={onChange} />
  );
}
```

上述我们在 App 页面中同时传入了 value 和 defaultValue 的值，虽然在使用上并没有任何问题。但是在开发模式下 React 会给予我们这样的警告：

它的大概意思是在说 React 无法解析出当前 TextField 中的 input 表单控件为受控还是非受控，因为我们同时传入了 value 和 defaultValue 的值。(但是它最终仍会将该 input 当做受控处理，因为 value 的优先级高于 defaultValue)

兼容两种模式

接下来就让我们来处理上述的 Warning 警告。

目前 TextField 内部 input 控件可以分别接受 value 和 defaultValue 两个值，这两个值完全由用户传入，显然是不太合理的。

我们先来思考下，我们需要解决这个警告的途径的思路：**我们将 TextField 处理为无论外部传入的是 value 还是 defaultValue 都在 TextField 内部通过受控处理。**

换句话说，无论调用者传入 defaultValue 还是 value ，对于调用方来说该表单控件是存在对应非受控和受控两种状态的。

但是对于 TextField 内部来说，我们会将外部传入的值全部当作受控来处理。

此时，我们来稍微改造改造我们的 TextField：

```ts
// ...
function fixControlledValue<T>(value: T) {
  if (typeof value === 'undefined' || value === null) {
    return ''
  }
  return String(value)
}

const TextField: React.FC<TextField> = (props) => {
  const { defaultValue, value, onChange, ...rest } = props;

  // 内部为受控状态控制 input 控件
  const [_value, setValue] = useState(() => {
    if (typeof value !== 'undefined') {
      return value;
    } else {
      return defaultValue;
    }
  });

  /**
   * onChange 函数
   * @param e
   */
  const _onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const inputValue = e.target.value;
    // 当 onChange 触发时，需要判断
    // 1. 如果当前外部传入 value === undefined ，此时表示为非受控模式。那么组件内部应该直接进行控件 value
    // 2. 相反，如果组件外部传入 value !== undefined，此时表示为受控模式。那么组件内部的值应该由外部的 pro
    if (typeof value === 'undefined') {
      setValue(inputValue);
    }
    onChange && onChange(e);
  };

  return <input value={(fixControlledValue(_value))} onChange={_onChange} {...rest} />;
};

export default TextField;
```

基于上述的思路，我们做了以下几点的小改造：

1. 将 TextField 内部之前基于外部传入的 value 和 defaultValue 全部通过内部 State 来进行初始化，在 TextField 内部进行受控处理。
2. 在 onChange 时，如果传入的 value 如果为非 undefined 那么表示外部希望该组件模式为受控模式，此时我们并不会改变内部的 state 。
3. 同时，我们定义了一个 fixedController 函数，保证内部 input 传入的 value 不为 undefined 或者 null，保证内部 input 是决定由受控状态改变的。

完成了上述功能点后，此时当我们传入 defaultValue 调用非受控的 TextField 时已经可以满足基础的功能点了：

```ts
// ...
<TextField  defaultValue={'hello world'} onChange={onChange} />
```

当外部传入 value 使用受控的情况时：

```ts
export default function App({ Component, pageProps }: AppProps) {
  const [state, setState] = useState('');

  const onChange = (e: ChangeEvent<HTMLInputElement>) => {
    const value = e.target.value;
    setState(value);
  };

  return (
    <TextField value={state}  onChange={onChange} />
  );
}
```

即使我们如何在页面的 input 中进行输入，此时传入的 onChange 的确会被触发同时通知 state 的值改变。

但是由于组件内部 useState 的值已经进行过初始化了，并不会由于组件的 props 改变而重新初始化组件内部的 state 状态。

```ts
// ...
  const [_value, setValue] = useState(() => {
    if (typeof value !== 'undefined') {
      return value;
    } else {
      return defaultValue;
```

```
    }
  });
```

此时就会造成，无论我们如何在页面上输入 onChange 会触发，外部 State 的值也会变化。

但是由于 TextField 中的 input 表单控件 value 是永远不会被改变，所以，页面不会发生任何变化。

那么，解决这个问题其实也非常简单。当 TextField 组件为受控状态时，内部表单的 value 值并不会跟随组件内部的 onChange 而改变表单的值。

而是，每当 props 中的 value 改变时，我们就需要及时改变对应表单的内部状态。

在 React 中我们不难想到这种场景应该利用的副作用函数，接下来我们再来为之前的 TextField 内部添加一个副作用 Hook：

```ts
const TextField: React.FC<TextField> = (props) => {
  const { defaultValue, value, onChange, ...rest } = props;

  // ...

  /** 当外部 props.value 改变时，修改对应内部的 State  */
  useLayoutEffect(() => {
    setValue(value);
  }, [value]);

  return (
    <input value={fixControlledValue(_value)} onChange={_onChange} {...rest} />
  );
};
```

此时，上述 TextField 的受控状态我们也完成了。

当我们再次传入 defaultValue 和 value 时，由于内部统一作为了组件内部 state 来处理所以自然也不会出现对应的 Warning 警告了。

其实，这也就是所谓 useMergedState 的源码核心思路。

它无非是基于上述的思路多做了一些边界状态的处理以及一些额外辅助参数的支持。接下来，我们来一起看看看这个 Hook 的源码。

源码

相信在经过上述的章节后，对于 React 中的受控和非受控 Hook 大家已经可以了解到其中的核心思路。

现在，让我们来一起进入 react-component 中 useMergedState 的源码来一探究竟吧。

初始化

首先，我们来看看顶部的这段逻辑：

ts 复制代码

```ts
import * as React from 'react';
import useEvent from './useEvent';
import useLayoutEffect, { useLayoutUpdateEffect } from './useLayoutEffect';
import useState from './useState';


enum Source {
  INNER,
  PROP,
}

type ValueRecord<T> = [T, Source, T];

/** We only think `undefined` is empty */
function hasValue(value: any) {
  return value !== undefined;
}

/**
 * Similar to `useState` but will use props value if provided.
 * Note that internal use rc-util `useState` hook.
 */
export default function useMergedState<T, R = T>(
  defaultStateValue: T | (() => T),
  option?: {
    defaultValue?: T | (() => T);
    value?: T;
    onChange?: (value: T, prevValue: T) => void;
    postState?: (value: T) => T;
  },
): [R, Updater<T>] {
  const { defaultValue, value, onChange, postState } = option || {};
```

```
  // ====================== Init ======================
  const [mergedValue, setMergedValue] = useState<ValueRecord<T>>(() => {
    let finalValue: T = undefined;
    let source: Source;

    // 存在 value 受控
    if (hasValue(value)) {
      finalValue = value;
      source = Source.PROP;
    } else if (hasValue(defaultValue)) {
      // 存在 defaultValue
      finalValue =
        typeof defaultValue === 'function'
          ? (defaultValue as any)()
          : defaultValue;
      source = Source.PROP;
    } else {
      // 两个都不存在
      finalValue =
        typeof defaultStateValue === 'function'
          ? (defaultStateValue as any)()
          : defaultStateValue;
      source = Source.INNER;
    }

    return [finalValue, source, finalValue];
  });

  const chosenValue = hasValue(value) ? value : mergedValue[0];
  const postMergedValue = postState ? postState(chosenValue) : chosenValue;

  // ...
}
```

上述的这段初始化逻辑其实和我们刚才差不多，对于传入的参数在内部使用 useState 进行初始化。

1. 首先判断是否存在 value ， 存在 value 则作为受控处理同时将 source 置为 prop 处理。
2. 其次，如果不存在有效 value ， 则判断是否存在 defaultValue ， 同时将 source 置为 prop 处理。
3. 最后，如果 value 和 defaultValue 都不存在有效参数那么将会使用第一个参数 defaultStateValue 初始化内部 state 同时将 source 作为 inner 处理。

其次：

1. chosenValue 表示使用的 value ， props 中如果存在传入 value 的话，表示受控模式直接取 `props.value` 。否则取内部的 `mergedValue[0]` 。
2. postMergedValue 表示，如果传入了 postState 方法，会在每次执行前格式化 chosenValue。

相信上面的初始化逻辑对于大家来讲都是轻松拿捏，我们继续往下看。

Sync & Update

```ts
export default function useMergedState<T, R = T>(
  defaultStateValue: T | (() => T),
  option?: {
    defaultValue?: T | (() => T);
    value?: T;
    onChange?: (value: T, prevValue: T) => void;
    postState?: (value: T) => T;
  },
): [R, Updater<T>] {
  // ...

  // ====================== Sync ======================
  useLayoutUpdateEffect(() => {
    setMergedValue(([prevValue]) => [value, Source.PROP, prevValue]);
  }, [value]);

  // ====================== Update ======================
  const changeEventPrevRef = React.useRef<T>();

  const triggerChange: Updater<T> = useEvent((updater, ignoreDestroy) => {
    setMergedValue(prev => {
      const [prevValue, prevSource, prevPrevValue] = prev;

      const nextValue: T =
        typeof updater === 'function' ? (updater as any)(prevValue) : updater;

      // Do nothing if value not change
      if (nextValue === prevValue) {
        return prev;
      }

      // Use prev prev value if is in a batch update to avoid missing data 解决批处理丢失上一次value问
      const overridePrevValue =
        prevSource === Source.INNER &&
          changeEventPrevRef.current !== prevPrevValue
          ? prevPrevValue
          : prevValue;
```

```
      return [nextValue, Source.INNER, overridePrevValue];
    }, ignoreDestroy);
  });

  // ...
}
```

接下来我们在看看所谓的同步和更新阶段。

同步 Sync

在同步阶段做的事情非常简单，它和我们上述自己写的 Demo 是一模一样的，是受控模式的特殊处理。

每当外部传入的 `props.value` 变化时，会调用 `setMergedValue` 同步更新 Hook 内部的 state 。

关于 `useLayoutUpdateEffect` 这个 Hook 也是 rc-util 中的一个辅助 hook：

ts 复制代码

```ts
export const useLayoutUpdateEffect: typeof React.useEffect = (
  callback,
  deps,
) => {
  const firstMountRef = React.useRef(true);

  useLayoutEffect(() => {
    if (!firstMountRef.current) {
      return callback();
    }
  }, deps);

  // We tell react that first mount has passed
  useLayoutEffect(() => {
    firstMountRef.current = false;
    return () => {
      firstMountRef.current = true;
    };
  }, []);
};
```

这个 Hook 的作为也非常简单，内部利用 ref 结合 useLayoutEffect 做到了**仅在依赖值更新时调用 callback 首次渲染并不执行。**

更新 Update

之后我们再来看看 Update 的逻辑。

```ts
const changeEventPrevRef = React.useRef<T>();

const triggerChange: Updater<T> = useEvent((updater, ignoreDestroy) => {
  setMergedValue(prev => {
    const [prevValue, prevSource, prevPrevValue] = prev;

    const nextValue: T =
      typeof updater === 'function' ? (updater as any)(prevValue) : updater;

    // Do nothing if value not change
    if (nextValue === prevValue) {
      return prev;
    }

    // Use prev prev value if is in a batch update to avoid missing data
    const overridePrevValue =
      prevSource === Source.INNER &&
        changeEventPrevRef.current !== prevPrevValue
        ? prevPrevValue
        : prevValue;

    return [nextValue, Source.INNER, overridePrevValue];
  }, ignoreDestroy);
});
```

首先，Update 的开头利用 changeEventPrevRef 这个 ref 值来确保每次更新时，获取到正确的 React 批处理的 prevValue。

这个值也许有些同学目前不太理解，没关系。我们会在稍后的 Tips 中结合实例来讲解它，目前如果你通过代码仍然不太理解它的话可以暂时不用过于在意。

useEvent

之后我们定义了一个 triggerChange 的方法，这个方法是利用 useEvent 来包裹的，首先我们先来 useEvent 是个什么东西：

```ts
import * as React from 'react';

export default function useEvent<T extends Function>(callback: T): T {
```

```
  const fnRef = React.useRef<any>();
  fnRef.current = callback;

  const memoFn = React.useCallback<T>(
    ((...args: any) => fnRef.current?.(...args)) as any,
    [],
  );

  return memoFn;
}
```

这个 useEvent 其实非常简单，它的作用仍然是使用 ref 和 useCallback 进行配合从而保证传入的 onChange 函数放在 fnRef 中。

从而确保每次 ReRender 时直接调用 fnRef.current 而无需在 Hook 重新生成一份传入的 onChange 定义。

同时这样的好处是，**虽然 useCallback 依赖的是一个 `[]` 但是由于 ref 的引用类型关系，即是外部 props.onChang 重新定义，内部 useEvent 包裹的 onChange 也会跟随生效。**

它算作是一个小的优化点而已。

setState 中的 ignoreDestroy

其次，我们再来看看函数内部的操作。可以看到定义的 triggerChange 函数接受两个参数，分别为 updater 和 ignoreDestroy 。

这里我们先忽略 ignoreDestroy 以免造成干扰。

我们先来看看函数内部的逻辑:

```ts 复制代码
const triggerChange: Updater<T> = useEvent((updater, ignoreDestroy) => {
  setMergedValue(prev => {
    // 结构出 state 中的值，分别为
    // prevValue 上一次的 value
    // prevSource 上一次的更新类型
    // 以及 prevPrevValue 上上一次的 value
    const [prevValue, prevSource, prevPrevValue] = prev;
    // 判断传入的是否为函数，如果是的话传入 prevValue 调用得到 nextValue
    const nextValue: T =
      typeof updater === 'function' ? (updater as any)(prevValue) : updater;

    // Do nothing if value not change
```

```
      if (nextValue === prevValue) {
        return prev;
      }

      // 确保 Patch 处理获得正确上一次的值 稍后结合实例来看
      // ...
    }, ignoreDestroy);
  });
```

相信上述的代码对于大家来说都是非常简单的，无非就是针对于每次调用 triggerChange 时进行参数的冲载。

如果是函数那么传入 prevValue ，非函数就获得对应的 nextValue 以及进行值相同不更新的操作。

不过，细心的小伙伴可能发现了，当我们调用 setMergedValue 时还接受了第二个参数 ignoreDestroy 。

我们再来回忆下 Init 阶段所谓的 setMergedValue 是从哪里来的：

```ts
import useState from './useState';
```

注意，Hook 中的 useState 并非来自 React 的 useState 而是 Rc-util 中自定义的 useState。

之所以 useState 接受第二个参数 ignoreDestroy 也正是 rc-util 自定义的 hook 支持第二个参数。

```javascript
// ...
// rc-util useState.ts 文件
export default function useSafeState<T>(
  defaultValue?: T | (() => T),
): [T, SetState<T>] {
  const destroyRef = React.useRef(false);
  const [value, setValue] = React.useState(defaultValue);

  // 每次 Render 后将 destroyRef.current 变为 false
  React.useEffect(() => {
    destroyRef.current = false;

    // 同时卸载后会将 destroyRef.current 变为 true
    return () => {
      destroyRef.current = true;
    };
  }, []);
```

```
  // 安全更新函数
  function safeSetState(updater: Updater<T>, ignoreDestroy?: boolean) {
    // 如果不为强制要求 ignoreDestroy 显示指定为 true
    // 同时组件已经卸载 destroyRef.current 为 true
    if (ignoreDestroy && destroyRef.current) {
      // 那么调用更新 state 的函数没有任何作用
      return;
    }

    setValue(updater);
  }

  return [value, safeSetState];
}
```

上述为 rc-util useState.ts 文件，它的用法和 React 中的 useState 类型。

不过是 `setState` 额外接收一个 `ignoreDestroy` 参数确保销毁后不会在被调用 setState 设置已销毁的状态。

这样做的好处其实也是一个针对于 React 中内存泄漏的优化点而已。

批处理更新处理

搞清楚了上述的小 Tips 后，我们继续来看看所谓的针对于批处理更新的 `changeEventPrevRef` 作用。

首先，在 Init 阶段我们针对于每一种传入的方式，比如 value、defaultValue 以及 defaultValueState 都定义了不同的类型。

定义了他们究竟是来自于 INNER 还是 PROP，忘记了的同学可以翻阅 Init 阶段在稍稍回忆下。

之后我们提到过在 `Sync` 同步阶段，每次 value 变化时，都会执行这个 Effect：

```ts 复制代码
  useLayoutUpdateEffect(() => {
    setMergedValue(([prevValue]) => [value, Source.PROP, prevValue]);
  }, [value]);
```

当我们为该 Hook 传入 value 表示为受控时，此时每次 value 变化都会直接调用 setMergedValue 方法并且保证 value 的类型为 `Source.PROP` 。

自然，changeEventPrevRef 和受控模式也没有任何关系。

那么当传入 defaultValueState 和 defaultValue 时，Hook 中表示为非受控处理时。

每次内部 mergeValue 改变就会触发对应的 triggerChange 从而触发对应的 setMergedValue 。

这里我们首先明确 changeEventPrevRef 是和非受控状态相关的一个 ref 变量。

其次，在 React 中存在一个➤ 批处理更新(Batch Updating)的概念。

同时，不要忘记在 useMergeState 第二个 option 参数中接收一个名为 onChange 的函数。

我们来结合 useMergeState 中 update 更新的代码来看看：

ts 复制代码

```ts
// ...
const changeEventPrevRef = React.useRef<T>();
const triggerChange: Updater<T> = useEvent((updater, ignoreDestroy) => {
 setMergedValue(prev => {
   // 结构出 state 中的值，分别为
   // prevValue 上一次的 value
   // prevSource 上一次的更新类型
   // 以及 prevPrevValue 上上一次的 value
   const [prevValue, prevSource, prevPrevValue] = prev;
   // 判断传入的是否为函数，如果是的话传入 prevValue 调用得到 nextValue
   const nextValue: T =
     typeof updater === 'function' ? (updater as any)(prevValue) : updater;

   // Do nothing if value not change
   if (nextValue === prevValue) {
     return prev;
   }

   // Use prev prev value if is in a batch update to avoid missing data
   // 确保非受控状态下的 onChange 函数多次同一队列中获得正确的 preValue 值
   const overridePrevValue =
     prevSource === Source.INNER &&
       changeEventPrevRef.current !== prevPrevValue
       ? prevPrevValue
       : prevValue;

   return [nextValue, Source.INNER, overridePrevValue];
 }, ignoreDestroy);
});
```

比如这样的使用场景：

```ts
const InputComponent: React.FC = (props) => {
  const [mergeState, setMergeState] = useMergedState('default value', {
    onChange: (currentValue, preValue) => {
      // log "[inputValue] 2"
      console.log(currentValue, '当前value');
      // 这里的preValue仍然为上一次的 inputValue 而非 inputValue + '1'
      console.log(preValue, '上一次value');
    },
  });

  const _onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const inputValue = e.target.value;
    // 调用三次 setMergeState
    setMergeState(inputValue);
    setMergeState(inputValue + '1');
    setMergeState(inputValue + '2');
  };

  return <input value={mergeState} onChange={_onChange} />;
};

export default InputComponent;
```

上述的 overridePrevValue 正是保证传入的 onChange 函数在内部多次 patch Updaing 后仍然可以通过 `changeEventPrevRef` 拿到正确的 prevPrevValue 值。

Change

最后，我们再来看看 Hook 最后的 Change 阶段：

```ts
  // ...
  // ====================== Change ======================
  useLayoutEffect(() => {
    // 每次 render mergedValue 改变时
    const [current, source, prev] = mergedValue;
    // 当前 current !== prev 同时 source === Source.INNER （非受控状态下）时才会触发 onChangeFn
    if (current !== prev && source === Source.INNER) {
      onChangeFn(current, prev);
      // 同时再次更新 changeEventPrevRef.current 为 prev(overridePrevValue)
      changeEventPrevRef.current = prev;
    }
  }, [mergedValue]);
  // ...
```

上述的代码其实看上去就非常简单了。

当每次 mergedValue 的值更新时，会触发对应的 useLayoutEffect 。

同时判断如果 `source === Source.INNER` 表示非受控状态下内部值改变同时 `current !== prev` 为一次有效的变化时。

会触发对应外部传入的 `onChangeFn(current,prev)` ，同时更新内部 ref `changeEventPrevRef.current` prev。

至此，整个 useMergedState 的源码我们就已经逐行解读完毕了。

如果仍有哪些地方你仍不是特别理解，那么你可以翻阅回去再次看看或者直接查阅它的➤ 代码。

结尾

这次的分享稍微显得有一些基础，不过我们可以发现一个看起非常简单的受控和非受控的概念在 useMergedState 中也的确藏着不少的知识点。

希望这篇文章可以在日常工作中对大家有所帮助。大家，加油!