React API
Подходы проектирования компонент

Кирилл Талецкий

TeachMeSkills
9 Октября 2023

Оптимизируем компоненты

Memo API

useMemo

Задача

- Внутри рендер цикла компоненты нам нужно делать сложные вычисления
- Например, пробегаться по списку фильмов и отображать его в удобный для рендера формат данных
- Мы не хотим делать дорогие вычисления на каждый рендер, а только на изменение ключевых данных

```tsx
const ImagesList: FC<{ items: { src: string; id: string }[] }> = ({
  items,
}) => {
  return (
    <ul>
      {items.map(({ src, id }) => (
        <li>
          <img key={id} src={src} alt={"Doggo"} />
        </li>
      ))}
    </ul>
  );
};

export const App: FC = () => {
  const [images, setImages] = useState<string[]>([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then((res) => res.json())
      .then(({ message }) => setImages(message));
  }, []);

  const mapped = images.map((src) => ({
    src,
    id: src.split("/").at(-1) || "",
  }));

  return <ImagesList items={mapped} />;
};
```

useMemo()

Кэширует данные между ре-рендерами

```
const mapped = useMemo(() => {
    return images.map((src) => ({
      src,
      id: src.split("/").at(-1) || "",
    }))
  }, [images]);
```

Вычисляем и возвращаем значение из колбэка

Массив зависимостей

Следствия

- useMemo сохраняет ссылку на вычисленное значение постоянной
  - Это позволяет использовать вычисленное значение как зависимость в useEffect и не беспокоиться, что useEffect будет вызываться слишком часто
  - Это позволяет использовать более простое сравнение по ссылке в React.memo()

Следствия

- Важно: useMemo — это всего лишь оптимизация. Реакт *не гарантирует* что значение никогда не будет вычислено, если зависимости неизменны, поэтому вы не можете на это полагаться.
- Если вам нужно положиться на неизменность пропсов, то скорее всего вы что-то делаете не так

useCallback()

Вспомним React.memo()

Принимаем решение о ре-рендере компонент

```tsx
const HeavyCompoment: FC = () => {
  // этот компонент нужно рендерить как можно реже
  return <div />;
};

// функция, для принятия решения о перерендере на основании пропсов
// когда `return true`, компонент НЕ будет перерисован
function arePropsEqual<P>(prevProps: P, nextProps: P) {
  // ваша логика сравнения пропсов
  return true;
}

// новый компонент, который будет перерисовываться только на изменение пропсов
export const MemoisedComponent = React.memo(HeavyCompoment, arePropsEqual);
```

Проблема

- Мы мемоизировали компонент, но пробрасываем в него пропсом функцию, которая создаётся прямо в рендер цикле

- Функция — это объект, который пересоздаётся на каждый рендер, ссылка на него меняется.
- React будет считать, что пропсы всегда новые и будет пере-рендерить компонент, не смотря на то, что функция могла никак не измениться
- React.memo() становится бесполезным!

```
const TextEditor: FC<{ onFocus: () => void }> = React.memo(
  ({ onFocus }) => {
    return (
      <textarea
        style={{ transition: "background-color 0.2s" }}
        onFocus={onFocus}
      />
    );
  },
);

export const App = () => {
  const [privacy, setPrivacy] = useState<Privacy>("public");
  const [isBold, setIsBold] = useState<boolean>(false);

  const onEditorFocus = () => {
    console.log("focused");
  };

  return (
    <>
      <header>
        <h1>My App</h1>
      </header>
      <main>
        <PostPrivacy value={privacy} onChange={setPrivacy} />
        <FontWeightControl isBold={isBold} setIsBold={setIsBold} />
        <TextEditor isBold={isBold} onFocus={onEditorFocus} />
      </main>
    </>
  );
};
```

Подходы проектирования

Uncontrolled / Controlled Component

Введение

- В стандарте HTML есть компоненты, которые умеют запоминать значения:
  - <input />
  - <select />
  - <textarea />
  - И др.

- Это stateful компоненты, то есть у них есть своё внутреннее состояние, без всякого реакта.

Uncontrolled Component

- Давайте используем такой элемент в React:
  - Мы можем добавить начальное значение.
  - Мы можем реагировать на изменение *внутреннего* состояния элемента через обработчики событий

```
const UncontrolledInput: FC = () => {
  const onChange = (e: ChangeEvent<HTMLInputElement>) => {
    console.log(`Пользователь ввёл: ${e.target.value}`);
  };

  return <input defaultValue={"Значение по умолчанию"} onChange={onChange} />;
};
```

- Но мы не имеем контроля над его состоянием — мы не можем изменить его когда нам это необходимо
- Такой компонент называется *неконтролируемым*.

Controlled Component

- Теперь давайте сделаем компонент *контролируемым*:
  - Для этого нам нужно завести собственное состояние и сделать его источником истины для HTML элемента

```tsx
const ControlledComponent: FC = () => {
  const [value, setValue] = useState<string>("");

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

  return <input value={value} onChange={onChange} />;
};
```

- Теперь у нас всегда есть возможность изменить состояние, в любой момент

Presentational and Container components

Разделение ответственности

- Пусть нам надо сделать компонент, который загружает и отображает картинки с собаками

- Мы можем организовать компоненты так, чтобы они разделяли зоны ответственности:
  - Container компонент (aka Smart Component) — должен отвечать за загрузку данных, их трансформацию, хранение и передачу в
  - Presentational компонент (aka View, Dumb) — должен отвечать только за логику отображения списка и за финальную вёрстку.

```tsx
/**
 * Presentational компонент. Может иметь своё состояние и логику, но
 * только необходимые для контроля отрисовки элементов.
 */
const DogImagesView: FC<{ loading?: boolean; sources: string[] }> = ({
  loading,
  sources,
}) => {
  if (loading) return <h1>Loading...</h1>;

  return (
    <ul>
      {sources.map((src) => (
        <li key={src}>
          <img src={src} alt="Dog" />
        </li>
      ))}
    </ul>
  );
};

/**
 * Container компонент. Содержит в себе все состояния и функции, отвечающие
 * за бизнес-логику приложения. НЕ содержит HTML компоненты и стили.
 */
const DogImages: FC = () => {
  const [isLoading, setIsLoading] = useState(false);
  const [images, setImages] = useState<string[]>([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then((res) => res.json())
      .then(({ message }) => setImages(message))
      .finally(() => setIsLoading(false));
  }, []);

  return <DogImagesView loading={isLoading} sources={images} />;
};
```

Анализ

- Плюсы разделения
  - Простая отладка — логика сконцентрирована в нескольких ключевых компонентах, а не размазана ровным слоем по всему дереву
  - Presentational компоненты легко переиспользовать, легко тестировать, часто с ними можно работать даже не будучи разработчиком
- Минусы разделения
  - Дополнительная и не всегда нужная по-отдельности сущность — компонент-отображение

```tsx
/**
 * Presentational компонент. Может иметь своё состояние и логику, но
 * только необходимые для контроля отрисовки элементов.
 */
const DogImagesView: FC<{ loading?: boolean; sources: string[] }> = ({
  loading,
  sources,
}) => {
  if (loading) return <h1>Loading...</h1>;

  return (
    <ul>
      {sources.map((src) => (
        <li key={src}>
          <img src={src} alt="Dog" />
        </li>
      ))}
    </ul>
  );
};

/**
 * Container компонент. Содержит в себе все состояния и функции, отвечающие
 * за бизнес-логику приложения. НЕ содержит HTML компоненты и стили.
 */
const DogImages: FC = () => {
  const [isLoading, setIsLoading] = useState(false);
  const [images, setImages] = useState<string[]>([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then((res) => res.json())
      .then(({ message }) => setImages(message))
      .finally(() => setIsLoading(false));
  }, []);

  return <DogImagesView loading={isLoading} sources={images} />;
};
```

Компромиссный подход

- Если заранее не известно, понадобится ли отдельный компонент-отображение, сделайте так, чтобы Presentational компонент можно было легко выделить в отдельный когда это понадобится.

- Можете использовать хуки, чтобы выделить отделить логику и интерфейсы данных

- Это позволит в дальнейшем легко выделять View компонент:
  - Если нам потребуется сделать переиспользуемый компонент список картинок
  - Если требования к списку усложнятся — сложное отображение, JS анимации, виртуализация
  - И т.д.

```
/**
 * Выделяем всю бизнес-логику в отдельный хук
 */
const useDogImages = () => {
  const [isLoading, setIsLoading] = useState(false);
  const [images, setImages] = useState<string[]>([]);

  useEffect(() => {
    fetch("https://dog.ceo/api/breed/labrador/images/random/6")
      .then((res) => res.json())
      .then(({ message }) => setImages(message))
      .finally(() => setIsLoading(false));
  }, []);

  return {isLoading, images}
}

/**
 * Используем один или несколько хуков с бизнес логикой
 * прямо внутри компоненты-отображения
 */
const DogImages: FC = () => {
  const {isLoading, images} = useDogImages();
   const {} = useMyComponentController()

  if (isLoading) return <h1>Loading...</h1>;

  return (
    <ul>
      {images.map((src) => (
        <li key={src}>
          <img src={src} alt="Dog" />
          <MyComponent />
        </li>
      ))}
    </ul>
  );
};

/**
 * При необходимости, отдельный Presentational компонент будет очень
 * легко вынести, просто убрав из DogImages хуки и добавив пропсы
 */
```

Higher Order Component

Компонент высшего порядка

- Функция, которая принимает на вход компонент, и возвращает такой же компонент

- Добавляет новый функционал в переданный компонент
  - Пропсы
  - Состояния
  - Обработчики событий
  - и т.д.

```tsx
interface PropsWithStyle {
  style?: CSSProperties;
}

/** Higher order component */
function withStyles<P extends PropsWithStyle>(
  Component: ComponentType<P>,
): ComponentType<P> {
  return (props) => {
    const style = { color: "red" };
    return <Component {...props} style={{ ...style, ...props.style }} />;
  };
}

const Button: FC<PropsWithChildren<PropsWithStyle>> = (props) => (
  <button {...props} />
);
const Text: FC<PropsWithChildren<PropsWithStyle>> =
  (props) => <p {...props} />;

const StyledButton = withStyles(Button);
const StyledText = withStyles(Text);

export const App: FC = () => {
  return (
    <div>
      <Button>I am not styled :(</Button>
      <Text>I am not styled :(</Text>

      <StyledButton>I am styled!</StyledButton>
      <StyledText>I am styled!</StyledText>
    </div>
  );
};
```

Error boundary

Ловим ошибки в дереве

- Если при рендере одной из наших компонент произойдёт ошибка, пользователь увидит лишь белый экран вместо всего приложения

- Это произойдёт потому, что ошибка никак не обработана, и весь рендер цикл завершится с ошибкой, не позволяя реакту закончить отрисовку

```
const ErroredComponent: FC = () => {
  throw new Error("Component crashed");
  return <h3>This component will crash!</h3>;
};

export const App: FC = () => {
  return (
    <div>
      <h1>Hello world</h1>
      <ErroredComponent />
    </div>
  );
};
```

Ловим ошибки в дереве

- Для обработок ошибок рендера компонент можно создавать специальные компоненты — <ErrorBoundary />

- Они с помощью методов жизненного цикла будут отлавливать ошибку в процессе рендера

- После того, как ошибка была поймана, вы можете отрендерить запасной компонент (fallback), чтобы улучшить пользовательский опыт

```
const ErroredComponent: FC = () => {
  throw new Error("Component crashed");
  return <h3>This component will crash!</h3>;
};

export const App: FC = () => {
  return (
    <div>
      <h1>Hello world</h1>
      <ErroredComponent />
    </div>
  );
};
```

React Router