React neu denken

Activities, automatische Optimierung und der Compiler

Jonas Herrmannsdörfer

Drei neue React APIs

- <Activity>
- useEffectEvent
- React Compiler

Live Coding 1: Activity

Activity: hidden

- DOM bleibt erhalten
- React State bleibt erhalten
- Effects werden bereinigt
- Updates laufen niedriger priorisiert

Activity Caveats

```
useEffect(() => {
  const subscription = createSubscription()
  subscription.connect()

  return () => {
    subscription.disconnect()
  };
}, []);
```

> Hidden hält keine unsichtbare Subscription am Leben.

Gut für: Modals, Drawers, Side Panels, Tabs.

Das Dependency-Problem

```
function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(roomId);

    connection.on("connected", () => {
      showNotification("Connected!", theme);
    });

    connection.connect();
    return () => connection.disconnect();
  }, [roomId, theme]);
}
```

Chat reconnectet bei Theme-Wechsel

```
Effect dependencies:
roomId + theme

roomId changes  -> reconnect
theme changes   -> reconnect
```

✅

❌

Das Dependency Array beschreibt gerade zu viel.

Live Coding 2: useEffectEvent

Reconnect nur bei Raum-Wechsel

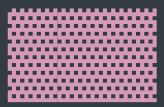
```
Effect lifetime:
roomId

Effect Event reads:
theme

roomId changes  -> reconnect
theme changes   -> latest notification style, no
reconnect
```

useEffectEvent

- verhält sich wie ein Event Handler
- hat immer Zugriff auf den neusten Wert von Props und State
- kann nur innerhalb von `useEffect`, `useLayoutEffect`, `useInsertionEffect` oder anderen `useEffectEvent` aufgerufen werden
- sollte nur für das Event eines Effects verwendet

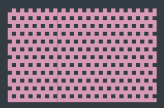
Live Coding 3: compiler

React Compiler

 Automatische Memoisierung zur Build-Zeit

- analysiert Control Flow, Data Flow und Mutability
- versteht die Rules of React
- memoiziert JSX-Werte, Callbacks und Berechnungen, wenn sicher
- erzeugt granulare und bedingte Memoisierung

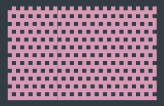
Was der Compiler optimiert



Primär Update-Performance

- weniger Cascading Re-Renders
- stabile JSX-Werte
- stabile Callbacks, wenn sicher
- teure Render-Berechnungen
- bedingte Memoisierung

Worauf muss ich achten?



Rules of React

- Components und Hooks müssen pure sein (Input => Output)
- Side Effects gehören in Effects
- Details: https://react.dev/reference/rules

> Je besser der Code diese Regeln einhält, desto mehr kann der Compiler sicher optimieren.

Was passiert mit memo, useMemo, useCallback?

- Bestehender Code darf bleiben
- `memo` in neuem Code nicht mehr nötig
- `useMemo` oder `useCallback` manchmal noch sinvoll, wenn Wert oder Funktion in Dependency Array verwendet werden muss

Das war's?

Könnten wir das nicht alles selbst machen?

Ja. Theoretisch.

Praktisch optimiert der Compiler Stellen, an die wir selten denken würden.

Er optimiert:

- feiner
- konsistenter

JSX ist auch nur ein Wert

```
const Card = memo(function Card({ children }) {
  return <section className="card">{children}</
section>;
});

function Page({ user, theme }) {
  const [count, setCount] = useState(0);

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button

      <Card>
        <Header user={user} />
        <ThemePreview theme={theme} />
      </Card>
    </>
  );
}
```

JSX ist auch nur ein Wert

```
// Manuell möglich, aber untypisch:
const cardContent = useMemo(() => (
  <>
    <Header user={user} />
    <ThemePreview theme={theme} />
  </>
), [user, theme]);

return <Card>{cardContent}</Card>;
```

> memo(Card) reicht nicht, wenn children jedes Mal neu sind.

Referenzen, an die niemand denkt

```
<Chart
  data={items.map(toPoint)}
  options={{ theme, showLegend: true }}
  onSelect={(point) => select(point.id)}
/>
```

```
const data = useMemo(() => items.map(toPoint), [
items]);
const options = useMemo(() => ({ theme,
showLegend: true }), [theme]);
const onSelect = useCallback((point) => select(
point.id), [select]);
```

> Manuell möglich, aber schnell unleserlich.

Bedingte Memoisierung

```
function Results({ kind, users, orders }) {
  if (kind === "users") {
    const rows = users.map(formatUser);
    return <Table rows={rows} />;
  }

  const rows = orders.map(formatOrder);
  return <Table rows={rows} />;
}
```

Bedingte Memoisierung

```
// Manuell: Branches aufteilen und einzeln
memoizieren
const UserResults = memo(function UserResults({
users }) {
  const rows = useMemo(() => users.map(formatUser
), [users]);
  return <Table rows={rows} />;
});

const OrderResults = memo(function OrderResults({
orders }) {
  const rows = useMemo(() => orders.map(
formatOrder), [orders]);
  return <Table rows={rows} />;
});
```

> Der Compiler kann entlang des Control Flows cachen.

Warum ist der Compiler also so besonders?

- useMemo und useCallback optimieren Stellen, an die wir gedacht haben
- React Compiler optimiert alle Stellen, die er beweisbar sicher optimieren kann

Wie implementiere ich den Compiler sicher?

- incremental Adoption
- erstmal nur auf einer Route testen und immer mehr aktivieren
  - https://react.dev/learn/react-compiler/incremental-adoption#directory-based-adoption
- oder aktiv Dateien mit Directive aktivieren
  - https://react.dev/learn/react-compiler/incremental-adoption#using-the-directive

Was der Compiler nicht löst

- API-Design
- Overfetching
- Underfetching
- JavaScript-Budget
- Caching
- Netzwerk-Latenz
- DOM-Größe
- Third-Party-Scripts

Persönlicher Blick aus echten Apps

> Selten ist fehlendes useMemo das größte Problem.

Häufiger:

- API-Form und Datenmenge
- falsche State Ownership
- zu viel Client-JavaScript

- fehlendes Caching

Takeaways

1. `<Activity>` lässt uns UI ausblenden und einblenden ohne State zu verlieren
2. `useEffectEvent` löst ein echtes Effect-Problem
3. Der Compiler optimiert eure Anwendung auch an den Stellen, an die wir nie denken würden.
4. Performance bleibt weiterhin Architekturarbeit, da

Slides + mehr Infos zu mir

herrmannsdoerfer.dev

Q&A