Cycle de vie d’un composant

Thèmes	React
Description	Comprendre le cycle de vie d’un composant.
Date de création	@16 mars 2025 16:46
Créée par	Kévin Wolff

Chapitres

Sur le même sujet

- Premier composant React
- Hook useEffect
- Hook useState

Introduction

Documentation officielle - cycle de vie

Un composant React suit un cycle de vie qui détermine comment il est créé, mis à jour et supprimé de l’interface. Ce cycle est essentiel pour optimiser les performances et gérer les effets secondaires (interactions avec des API, gestion des événements, etc.).

Le cycle de vie d’un composant est composé de trois phases :

1. Montage (mounting) : le composant est créé et ajouté au DOM.
2. Mise à jour (updating) : le composant est re-rendu lorsqu’il reçoit de nouvelles props ou que son état change.
3. Démontage (unmounting) : le composant est supprimé du DOM.

🚧 L’illustration du cycle de vie sera d’abord expliquée avec les composants en classe. L’approche moderne basée sur les hooks est plus concise mais abstrait certaines étapes du cycle de vie.

Comprendre les étapes du cycle de vie avec l’ancienne syntaxe permet de mieux appréhender comment React fonctionne sous le capot. Une fois ces bases acquises, vous découvrirez comment l’approche avec des “hooks”.

Point historique

À ses débuts, React utilisait une approche inspirée de la programmation orientée objet (POO) avec des composants en classe. Cependant, cette approche a été progressivement remplacée par des composants fonctionnels, plus lisibles et plus flexibles.

Grâce aux hooks, introduits avec React 16.8 (2019), les composants fonctionnels peuvent désormais gérer un état et des effets secondaires, rendant l’usage des classes obsolète dans la plupart des cas. Vous découvrirez cette approche moderne plus loin dans ce document.

Cycle de vie ancienne approche

Dans l'ancienne approche, un composant React était une classe qui étendait Component. Cela lui permettait d'accéder aux méthodes du cycle de vie, qui dictent son comportement.

```
import React, { Component } from "react";

class MyComponent extends Component {
 // Le constructeur pour initialiser l'état
 constructor(props) {
  super(props);
  this.state = { count: 0 };
  console.log("Constructor : initialisation");
 }

 // S'exécute juste après le montage du composant
 componentDidMount() {
  console.log("componentDidMount : le composant est monté !");
 }

 // S'exécute après chaque mise à jour (props ou state)
 componentDidUpdate(prevProps, prevState) {
  console.log("componentDidUpdate : le composant a été mis à jour !");
 }

 // S'exécute juste avant la suppression du composant
 componentWillUnmount() {
  console.log("componentWillUnmount : le composant va être supprimé !");
 }

 increment = () => {
  this.setState({ count: this.state.count + 1 });
 };

 render() {
  console.log("render : affichage du composant");
  return (
   <div>
    <p>Compteur : {this.state.count}</p>
    <button onClick={this.increment}>+1</button>
   </div>
  );
 }
}

export default MyComponent;
```

`constructor`	S'exécute à l'instanciation d'une classe, ici elle permet de gérer les props reçus et initialiser un compteur à zéro.

`componentDidMount`	S'exécute dès que le composant est monté dans le DOM, pour appeler une API ou démarrer un timer.
`componentDidUpdate`	S'exécute après chaque mise à jour, lorsque l'état du composant change après l'utilisation du compteur par exemple, ou si le composant reçoit de nouvelles props.
`componentWillUnmount`	S'exécute avant que le composant soit supprimé du DOM et détruit.
`render`	Devait être déclarée pour fournir le markup du composant.

C'est grâce à ce cycle de vie qu'un composant s'initialise correctement, gère ses changements d'états, se détruit et déclare son markup.

Cycle de vie nouvelle approche

Depuis React 16.8, l'utilisation des hooks a remplacé les méthodes du cycle de vie dans les composants fonctionnels. Les hooks permettent de gérer l'état et les effets secondaires sans avoir besoin de classes.

Le `constructor` n'est plus nécessaire à l'initialisation des propriétés du composant grâce au 📚 Hook useState.

Les méthodes `componentDidMount`, et `componentWillUnmount` ont étaient remplacées par le 📚 Hook useEffect.

La méthode `render` n'est plus nécessaire car le composant fonctionnel doit retourner son markup.

🚧 L'utilisation des hooks entraîne le re-rendering (le re-exécution) des composants, il est nécessaire de comprendre comment ce système fonctionne pour améliorer les performances de vos applications React. Voir chapitre suivant.

Le re-rendering

Le rendering correspond à l'exécution d'un composant pour la première fois lorsqu'il est monté dans l'interface utilisateur. Cela déclenche le cycle de vie du composant vu précédemment.

Le re-rendering (re-exécution) est un concept fondamental dans React. Il permet de mettre à jour l'interface en fonction des changements d'états ou des nouvelles props reçues d'un composant parent.

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

Ce composant déclare un état `count` initialisé à zéro. Lorsque l'on clique sur le bouton, la valeur de `count` est incrémentée. À chaque mise à jour de `count`, React re-render le composant pour afficher la nouvelle valeur.

```
function Parent() {
  const [value, setValue] = useState(0);

  return <Child number={value} />;
}

function Child({ number }) {
  return <p>Valeur : {number}</p>;
}
```

Ici le composant `Child` reçoit la valeur à afficher par le biais des props. Le composant parent `Parent` déclare un état `count` initialisé à zéro et fournis cette donnée au composant enfant `Child`.

Si l'état de `count` évolue au sein du composant `Parent` alors cette valeur est à nouveau fournis au composant enfant `Child` et entraîne par conséquent son re-rendering.

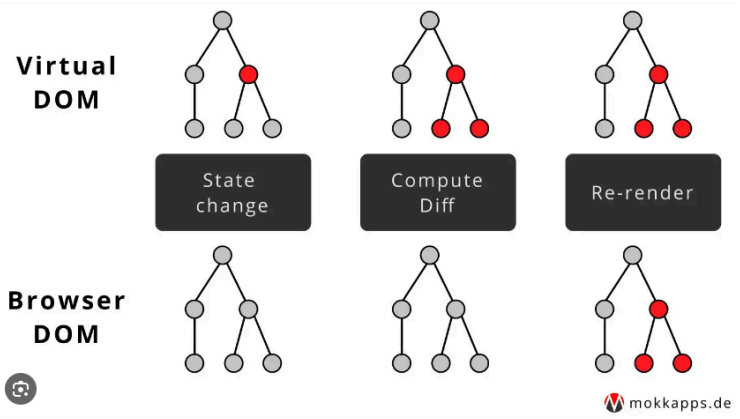
Notez que re-render un composant ne relance pas son cycle de vie, le composant n'est pas démonté puis recréé. Tous ses états sont conservés, seules les parties du cycle de vie associées aux mises à jours (`useEffect` avec dépendances) sont exécutées. Le rendu du composant est quant à lui re-exécuté pour refléter les changements dans l'interface.

Comment fonctionne le re-rendering

Lorsqu'un composant est monté pour la première fois, sa fonction de rendu est exécutée et un Virtual DOM est généré en mémoire. Ce Virtual DOM est une représentation de la structure des composants et de leur contenu, incluant les éléments HTML qui seront affichés à l'écran.

Lorsqu'un composant est re-render, React ne remonte pas tout depuis zéro. Il compare la nouvelle version du Virtual DOM avec la précédente pour identifier précisément les éléments qui ont été ajoutés, supprimés ou modifiés. Ce processus, appelé diffing, permet à React de cibler les changements nécessaires plutôt que de reconstruire l'ensemble de l'interface.

Plutôt que de mettre à jour directement tout le DOM réel, ce qui serait coûteux en performance, React applique uniquement les modifications détectées. Cette approche, connue sous le nom de reconciliation, optimise l'affichage en limitant les manipulations du DOM, garantissant ainsi une interface plus fluide et réactive.


Virtual DOM
State change
Compute Diff
Re-render
Browser DOM
mokkapps.de