Etat et cycle de vie

Sommaire

I. Concept d'état et de cycle de vie dans un composant React.

Vous pouvez trouver une référence d'API de composant détaillée ici .

Nous appelons `ReactDOM.render()` pour changer la sortie rendue :

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Dans cette section, nous allons apprendre à rendre le composant `Clock` réellement réutilisable et encapsulé.

Il établira son propre chronomètre et se mettra à jour toutes les secondes.

Nous pouvons commencer par encapsuler à quoi ressemble l'horloge :

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Cependant, il manque une exigence cruciale : le fait de configurer Clock Timer et de mettre à jour l'interface utilisateur toutes les secondes devrait être un détail de la mise en œuvre du Clock.

Idéalement, nous voulons écrire ceci une fois et disposer à mettre à jour : Clock

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Pour implémenter cela, nous devons ajouter « state » au composant Clock.

State est un objet du composant, mais il est privé et entièrement contrôlé par le composant.

II. Conversion d'une fonction en classe

Vous pouvez convertir un composant de fonction comme une classe Clock en cinq étapes :

1. Créez une classe ES6 , avec le même nom, qui s'étend React.Component.
2. Ajoutez une seule méthode vide à elle appelée render().
3. Déplacez le corps de la fonction dans la méthode render().
4. Remplacer props par this.props dans le contenu render().
5. Supprimez la déclaration de fonction vide restante.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Clock est maintenant défini comme une classe plutôt que comme une fonction.

La méthode render sera appelée à chaque fois qu'une mise à jour se produit, mais tant que le rendu s'effectue <Clock />dans le même nœud DOM, une seule instance de la classe Clock sera utilisée. Cela nous permet d'utiliser des fonctionnalités supplémentaires telles que les méthodes d'état local et de cycle de vie.

III. Ajout d'un état local à une classe

Nous allons passer de l'objet date à l'état en trois étapes:

1. Remplacer `this.props.date` par `this.state.date` dans la méthode `render()`:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. Ajoutez un constructeur de classe qui attribue l'initiale `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Notez comment nous passons props au constructeur de base:

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Les composants de classe doivent toujours appeler le constructeur de base avec props.

3. Retirer l'objet date de l'élément <Clock /> :

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Nous ajouterons plus tard le code du minuteur au composant lui-même.

Le résultat ressemble à ceci :

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Ensuite, nous allons configurer `Clock` notre propre minuteur et le mettre à jour toutes les secondes.

IV. Ajout de méthodes de cycle de vie à une classe

Dans les applications comportant de nombreux composants, il est très important de libérer les ressources utilisées par les composants lorsqu'ils sont détruits.

Nous voulons « configurer un Timer » à chaque fois que le Clock est effectué dans le DOM pour la première fois. Cela s'appelle «montage» dans React.

Nous voulons également « stopper un Timer » chaque fois que le DOM produit par le Clock est supprimé. Cela s'appelle « démonter » dans React.

Nous pouvons déclarer des méthodes spéciales sur la classe de composant pour exécuter du code lorsqu'un composant monte et décompose :

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Ces méthodes sont appelées « méthodes de cycle de vie ».

La méthode componentDidMount() est exécutée une fois que la sortie du composant a été rendue dans le DOM. C'est un bon endroit pour configurer une minuterie:

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

Notez comment nous sauvegardons l'identifiant this de la minuterie.

Quand l'objet this.props est configuré par React lui-même et this.state a une signification particulière. Vous êtes libre d'ajouter des champs supplémentaires

à la classe manuellement si vous devez stocker quelque chose qui ne fait pas partie du flux de données (comme un ID de temporisateur).

Nous allons abattre la minuterie dans la méthode componentWillUnmount() du cycle de vie:

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

Enfin, nous allons implémenter une méthode appelée tick() que le composant Clock sera exécuté toutes les secondes.

Il utilisera this.setState() pour planifier les mises à jour de l'état local du composant:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }
```

```
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
{this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Maintenant, l'horloge sonne à chaque seconde.

Récapitulons rapidement ce qui se passe et l'ordre dans lequel les méthodes sont appelées :

Quand <Clock />est passé à ReactDOM.render(), React appelle le constructeur du composant Clock. Comme il doit afficher l'heure actuelle, il initialise this.state avec un objet, y compris l'heure actuelle.

Nous mettrons ultérieurement à jour cet état.

React appelle ensuite la méthode render() du composant Clock. C'est ainsi que React apprend ce qui doit être affiché à l'écran. React met ensuite à jour le DOM pour qu'il corresponde à la sortie du rendu de Clock.

Lorsque la sortie Clock est insérée dans le DOM, React appelle la component avec la méthode componentDidMount() du cycle de vie. À l'intérieur, le composant Clock demande au navigateur de configurer une minuterie pour appeler la méthode tick() du composant une fois par seconde.

Chaque seconde, le navigateur appelle la méthode tick(). À l'intérieur, le composant Clock planifie une mise à jour de l'interface utilisateur en appelant setState() avec un objet contenant l'heure actuelle. Grâce à l'appel setState(), React sait que l'état a changé et appelle à render()nouveau la méthode pour savoir ce qui devrait apparaître à l'écran. Cette fois, this.state.date dans la méthode render() sera différente et la sortie de rendu inclura l'heure mise à jour. React met à jour le DOM en conséquence.

Si le Clockcomposant est jamais supprimé du DOM, React appelle la componentWillUnmount()méthode de cycle de vie afin que le minuteur soit arrêté.

V. Utiliser correctement l'état

Il y a trois choses que vous devriez savoir `setState()`.

Ne pas modifier directement l'état

Par exemple, cela ne rendra pas de nouveau un composant :

```
// Wrong
this.state.comment = 'Hello';
```

Au lieu de cela, utilisez `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

Le seul endroit où vous pouvez affecter this.state est le constructeur.

VI. Les mises à jour d'état peuvent être asynchrones

React peut regrouper plusieurs setState() appels en une seule mise à jour pour améliorer les performances.

Etant donné que this.props et this.state pouvant être mis à jour de manière asynchrone, vous ne devez pas vous fier à leurs valeurs pour calculer le prochain état.

Par exemple, ce code peut ne pas réussir à mettre à jour le compteur :

```
 // Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Pour résoudre ce problème, utilisez une seconde forme `setState()`qui accepte une fonction plutôt qu'un objet. Cette fonction recevra l'état précédent comme

premier argument et les accessoires au moment de l'application de la mise à jour comme second argument :

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Nous avons utilisé une « arrow function » ci-dessus, mais cela fonctionne également avec les fonctions habituelles :

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

VII.Les mises à jour d'état sont fusionnées

Lorsque vous appelez setState(), React fusionne l'objet que vous fournissez dans l'état actuel.

Par exemple, votre état peut contenir plusieurs variables indépendantes :

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

Ensuite, vous pouvez les mettre à jour indépendamment avec des appels séparés de `setState()`:

```
  componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
    });
```

```
    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

La fusion est superficielle, this.setState({comments}) laissant ainsi les données this.state.posts intactes, mais les remplace complètement par this.state.comments.

VIII. Les données s'écoulent

Ni les composants parents ni les composants enfants ne peuvent savoir si un composant donné a un état ou non, et ils ne devraient pas se soucier de savoir s'il est défini en tant que fonction ou classe.

C'est pourquoi l'état est souvent appelé local ou encapsulé. Il n'est accessible à aucun composant autre que celui qui le possède et le définit.

Un composant peut choisir de transmettre son état comme accessoire à ses composants enfants :

```
 <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

Cela fonctionne également pour les composants définis par l'utilisateur :

```
<FormattedDate date={this.state.date} />
```

Le composant `FormattedDate` recevrait l'objet date sans savoir s'il provenait de l'état de `Clock` :

```
function FormattedDate(props) {
  return <h2>It is {
props.date.toLocaleTimeString()}.</h2>;
}
```

Ceci est communément appelé flux de données « top-down » ou «unidirectionnel». Tout état appartient toujours à un composant spécifique, et

toute donnée ou interface utilisateur dérivée de cet état ne peut affecter que les composants "situés au-dessous" de ceux-ci dans l'arborescence.

Si vous imaginez une arborescence de composants comme une cascade d'accessoires, l'état de chaque composant est comme une source d'eau supplémentaire qui la rejoint à un point arbitraire, mais qui s'écoule également vers le bas.

Pour montrer que tous les composants sont vraiment isolés, nous pouvons créer un composant App qui rend trois <Clock>s:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Chacun configure son composant Clock par son propre minuteur et se met à jour indépendamment.

Dans les applications React, le fait qu'un composant soit dynamique ou sans état est considéré comme un détail d'implémentation du composant susceptible de changer avec le temps. Vous pouvez utiliser des composants sans état dans des composants avec état, et inversement.