On this page

`connect()`

La `connect()` fonction connecte un composant React à un magasin Redux.

Il fournit à son composant connecté les éléments de données dont il a besoin du magasin et les fonctions qu'il peut utiliser pour envoyer des actions au magasin.

Il ne modifie pas la classe de composant qui lui est transmise ; à la place, il renvoie une nouvelle classe de composants connectés qui encapsule le composant que vous avez transmis.

```
function connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)
```

Le `mapStateToProps` et `mapDispatchToProps` traite respectivement de et de votre magasin `state` Redux `dispatch`. `state` et `dispatch` sera fourni à vos fonctions `mapStateToProps` ou `mapDispatchToProps` comme premier argument.

Les retours de `mapStateToProps` et `mapDispatchToProps` sont désignés en interne par `stateProps` et `dispatchProps`, respectivement. Ils seront fournis à `mergeProps`, s'ils sont définis, en tant que premier et deuxième arguments, où le troisième argument sera `ownProps`. Le résultat combiné, communément appelé `mergedProps`, sera alors fourni à votre composant connecté.

`connect()`

`connect` accepte quatre paramètres différents, tous facultatifs. Par convention, on les appelle :

1. `mapStateToProps?: Function`
2. `mapDispatchToProps?: Function | Object`
3. `mergeProps?: Function`
4. `options?: Object`

`mapStateToProps?: (state, ownProps?) => Object`

Si une `mapStateToProps` fonction est spécifiée, le nouveau composant wrapper s'abonnera aux mises à jour du magasin Redux. Cela signifie que chaque fois que le magasin est mis à jour, `mapStateToProps` sera appelé. Le résultat de `mapStateToProps` doit être un objet simple, qui sera fusionné avec les accessoires du composant enveloppé. Si vous ne souhaitez pas vous abonner aux mises à jour de la boutique, passez `null` ou `undefined` à la place de `mapStateToProps` .

1. `state: Object`
2. `ownProps?: Object`

Une `mapStateToProps` fonction prend au maximum deux paramètres. Le nombre de paramètres de fonction déclarés (alias arité) affecte le moment où elle sera appelée. Cela détermine également si la fonction recevra ownProps. Voir les notes ici .

`state`

If your `mapStateToProps` function is declared as taking one parameter, it will be called whenever the store state changes, and given the store state as the only parameter.

```
const mapStateToProps = (state) => ({ todos: state.todos })
```

`ownProps`

If your `mapStateToProps` function is declared as taking two parameters, it will be called whenever the store state changes *or* when the wrapper component receives new props (based on shallow equality comparisons). It will be given the store state as the first parameter, and the wrapper component's props as the second parameter.

The second parameter is normally referred to as `ownProps` by convention.

```
const mapStateToProps = (state, ownProps) => ({
  todo: state.todos[ownProps.id],
})
```

Returns

Your `mapStateToProps` functions are expected to return an object. This object, normally referred to as `stateProps`, will be merged as props to your connected component. If you define `mergeProps`, it will be supplied as the first parameter to `mergeProps`.

The return of the `mapStateToProps` determine whether the connected component will re-render (details here).

For more details on recommended usage of `mapStateToProps`, please refer to our guide on using `mapStateToProps`.

> You may define `mapStateToProps` and `mapDispatchToProps` as a factory function, i.e., you return a function instead of an object. In this case your returned function will be treated as the real `mapStateToProps` or `mapDispatchToProps`, and be called in subsequent calls. You may see notes on Factory Functions or our guide on performance optimizations.

`mapDispatchToProps?: Object | (dispatch, ownProps?) => Object`

Conventionally called `mapDispatchToProps`, this second parameter to `connect()` may either be an object, a function, or not supplied.

Your component will receive `dispatch` by default, i.e., when you do not supply a second parameter to `connect()`:

```
// do not pass `mapDispatchToProps`
connect()(MyComponent)
connect(mapState)(MyComponent)
connect(mapState, null, mergeProps, options)(MyComponent)
```

If you define a `mapDispatchToProps` as a function, it will be called with a maximum of two parameters.

Parameters

1. `dispatch: Function`
2. `ownProps?: Object`

`dispatch`

If your `mapDispatchToProps` is declared as a function taking one parameter, it will be given the `dispatch` of your `store`.

```
const mapDispatchToProps = (dispatch) => {
  return {
    // dispatching plain actions
    increment: () => dispatch({ type: 'INCREMENT' }),
    decrement: () => dispatch({ type: 'DECREMENT' }),
    reset: () => dispatch({ type: 'RESET' }),
  }
}
```

`ownProps`

If your `mapDispatchToProps` function is declared as taking two parameters, it will be called with `dispatch` as the first parameter and the props passed to the wrapper component as the second parameter, and will be re-invoked whenever the connected component receives new props.

The second parameter is normally referred to as `ownProps` by convention.

```
// binds on component re-rendering
<button onClick={() => this.props.toggleTodo(this.props.todoId)} />

// binds on `props` change
const mapDispatchToProps = (dispatch, ownProps) => ({
  toggleTodo: () => dispatch(toggleTodo(ownProps.todoId)),
})
```

The number of declared function parameters of `mapDispatchToProps` determines whether they receive ownProps. See notes here.

Returns

Your `mapDispatchToProps` functions are expected to return an object. Each fields of the object should be a function, calling which is expected to dispatch an action to the store.

The return of your `mapDispatchToProps` functions are regarded as `dispatchProps`. It will be merged as props to your connected component. If you define `mergeProps`, it will be supplied as the second parameter to `mergeProps`.

```
const createMyAction = () => ({ type: 'MY_ACTION' })
const mapDispatchToProps = (dispatch, ownProps) => {
  const boundActions = bindActionCreators({ createMyAction }, dispatch)
  return {
    dispatchPlainObject: () => dispatch({ type: 'MY_ACTION' }),
    dispatchActionCreatedByActionCreator: () => dispatch(createMyAction()),
    ...boundActions,
    // you may return dispatch here
    dispatch,
  }
}
```

For more details on recommended usage, please refer to our guide on using `mapDispatchToProps`.

> You may define `mapStateToProps` and `mapDispatchToProps` as a factory function, i.e., you return a function instead of an object. In this case your returned function will be treated as the real `mapStateToProps` or `mapDispatchToProps`, and be called in subsequent calls. You may see notes on Factory Functions or our guide on performance optimizations.

Object Shorthand Form

`mapDispatchToProps` may be an object where each field is an action creator.

```
import { addTodo, deleteTodo, toggleTodo } from './actionCreators'

const mapDispatchToProps = {
  addTodo,
  deleteTodo,
  toggleTodo,
}

export default connect(null, mapDispatchToProps)(TodoApp)
```

In this case, React-Redux binds the `dispatch` of your store to each of the action creators using `bindActionCreators`. The result will be regarded as `dispatchProps`, which will be either directly merged to your connected components, or supplied to `mergeProps` as the second argument.

```
// internally, React-Redux calls bindActionCreators
// to bind the action creators to the dispatch of your store
bindActionCreators(mapDispatchToProps, dispatch)
```

We also have a section in our `mapDispatchToProps` guide on the usage of object shorthand form here.

`mergeProps?: (stateProps, dispatchProps, ownProps) => Object`

If specified, defines how the final props for your own wrapped component are determined. If you do not provide `mergeProps`, your wrapped component receives `{ ...ownProps, ...stateProps, ...dispatchProps }` by default.

Parameters

`mergeProps` should be specified with maximum of three parameters. They are the result of `mapStateToProps()`, `mapDispatchToProps()`, and the wrapper component's `props`, respectively:

1. `stateProps`
2. `dispatchProps`
3. `ownProps`

The fields in the plain object you return from it will be used as the props for the wrapped component. You may specify this function to select a slice of the state based on props, or to bind action creators to a particular variable from props.

Returns

The return value of `mergeProps` is referred to as `mergedProps` and the fields will be used as the props for the wrapped component.

`options?: Object`

```
{
  context?: Object,
  pure?: boolean,
  areStatesEqual?: Function,
```

```
  areOwnPropsEqual?: Function,
  areStatePropsEqual?: Function,
  areMergedPropsEqual?: Function,
  forwardRef?: boolean,
}
```

`context: Object`

> Note: This parameter is supported in >= v6.0 only

React-Redux v6 allows you to supply a custom context instance to be used by React-Redux. You need to pass the instance of your context to both `<Provider />` and your connected component. You may pass the context to your connected component either by passing it here as a field of option, or as a prop to your connected component in rendering.

```
// const MyContext = React.createContext();
connect(mapStateToProps, mapDispatchToProps, null, { context: MyContext })(
  MyComponent
)
```

`pure: boolean`

- default value: `true`

Assumes that the wrapped component is a "pure" component and does not rely on any input or state other than its props and the selected Redux store's state.

When `options.pure` is true, `connect` performs several equality checks that are used to avoid unnecessary calls to `mapStateToProps`, `mapDispatchToProps`, `mergeProps`, and ultimately to `render`. These include `areStatesEqual`, `areOwnPropsEqual`, `areStatePropsEqual`, and `areMergedPropsEqual`. While the defaults are probably appropriate 99% of the time, you may wish to override them with custom implementations for performance or other reasons.

We provide a few examples in the following sections.

`areStatesEqual: (next: Object, prev: Object) => boolean`

- default value: `strictEqual: (next, prev) => prev === next`

When pure, compares incoming store state to its previous value.

Example 1

```
const areStatesEqual = (next, prev) =>
  prev.entities.todos === next.entities.todos
```

You may wish to override `areStatesEqual` if your `mapStateToProps` function is computationally expensive and is also only concerned with a small slice of your state. The example above will effectively ignore state changes for everything but that slice of state.

Example 2

If you have impure reducers that mutate your store state, you may wish to override `areStatesEqual` to always return false:

```
const areStatesEqual = () => false
```

This would likely impact the other equality checks as well, depending on your `mapStateToProps` function.

`areOwnPropsEqual: (next: Object, prev: Object) => boolean`

- default value: `shallowEqual: (objA, objB) => boolean` (returns `true` when each field of the objects is equal)

When pure, compares incoming props to its previous value.

You may wish to override `areOwnPropsEqual` as a way to whitelist incoming props. You'd also have to implement `mapStateToProps`, `mapDispatchToProps` and `mergeProps` to also whitelist props. (It may be simpler to achieve this other ways, for example by using recompose's mapProps.)

`areStatePropsEqual: (next: Object, prev: Object) => boolean`

- type: `function`
- default value: `shallowEqual`

When pure, compares the result of `mapStateToProps` to its previous value.

`areMergedPropsEqual: (next: Object, prev: Object) => boolean`

- default value: `shallowEqual`

When pure, compares the result of `mergeProps` to its previous value.

You may wish to override `areStatePropsEqual` to use `strictEqual` if your `mapStateToProps` uses a memoized selector that will only return a new object if a relevant prop has changed. This would be a very slight performance improvement, since would avoid extra equality checks on individual props each time `mapStateToProps` is called.

You may wish to override `areMergedPropsEqual` to implement a `deepEqual` if your selectors produce complex props. ex: nested objects, new arrays, etc. (The deep equal check may be faster than just re-rendering.)

`forwardRef: boolean`

> Note: This parameter is supported in >= v6.0 only

If `{forwardRef : true}` has been passed to `connect`, adding a ref to the connected wrapper component will actually return the instance of the wrapped component.

`connect()` Returns

The return of `connect()` is a wrapper function that takes your component and returns a wrapper component with the additional props it injects.

```
import { login, logout } from './actionCreators'

const mapState = (state) => state.user
const mapDispatch = { login, logout }

// first call: returns a hoc that you can use to wrap any component
const connectUser = connect(mapState, mapDispatch)

// second call: returns the wrapper component with mergedProps
// you may use the hoc to enable different components to get the same behavior
const ConnectedUserLogin = connectUser(Login)
const ConnectedUserProfile = connectUser(Profile)
```

In most cases, the wrapper function will be called right away, without being saved in a temporary variable:

```
import { login, logout } from './actionCreators'
```

```
const mapState = (state) => state.user
const mapDispatch = { login, logout }

// call connect to generate the wrapper function, and immediately call
// the wrapper function to generate the final wrapper component.

export default connect(mapState, mapDispatch)(Login)
```

Example Usage

Because `connect` is so flexible, it may help to see some additional examples of how it can be called:

- Inject just `dispatch` and don't listen to store

```
export default connect()(TodoApp)
```

- Inject all action creators (`addTodo`, `completeTodo`, ...) without subscribing to the store

```
import * as actionCreators from './actionCreators'

export default connect(null, actionCreators)(TodoApp)
```

- Inject `dispatch` and every field in the global state

> Don't do this! It kills any performance optimizations because `TodoApp` will rerender after every state change. It's better to have more granular `connect()` on several components in your view hierarchy that each only listen to a relevant slice of the state.

```
// don't do this!
export default connect((state) => state)(TodoApp)
```

- Inject `dispatch` and `todos`

```
function mapStateToProps(state) {
  return { todos: state.todos }
}

export default connect(mapStateToProps)(TodoApp)
```

- Inject `todos` and all action creators

```
import * as actionCreators from './actionCreators'

function mapStateToProps(state) {
  return { todos: state.todos }
}

export default connect(mapStateToProps, actionCreators)(TodoApp)
```

- Inject `todos` and all action creators (`addTodo`, `completeTodo`, ...) as `actions`

```
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos` and a specific action creator (`addTodo`)

```
import { addTodo } from './actionCreators'
import { bindActionCreators } from 'redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators({ addTodo }, dispatch)
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos` and specific action creators (`addTodo` and `deleteTodo`) with shorthand syntax

```
import { addTodo, deleteTodo } from './actionCreators'

function mapStateToProps(state) {
  return { todos: state.todos }
}

const mapDispatchToProps = {
  addTodo,
  deleteTodo,
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos`, `todoActionCreators` as `todoActions`, and `counterActionCreators` as `counterActions`

```
import * as todoActionCreators from './todoActionCreators'
import * as counterActionCreators from './counterActionCreators'
import { bindActionCreators } from 'redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return {
    todoActions: bindActionCreators(todoActionCreators, dispatch),
    counterActions: bindActionCreators(counterActionCreators, dispatch),
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos`, and todoActionCreators and counterActionCreators together as `actions`

```
import * as todoActionCreators from './todoActionCreators'
import * as counterActionCreators from './counterActionCreators'
import { bindActionCreators } from 'redux'

function mapStateToProps(state) {
```

```
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(
      { ...todoActionCreators, ...counterActionCreators },
      dispatch
    ),
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos`, and all `todoActionCreators` and `counterActionCreators` directly as props

```
import * as todoActionCreators from './todoActionCreators'
import * as counterActionCreators from './counterActionCreators'
import { bindActionCreators } from 'redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators(
    { ...todoActionCreators, ...counterActionCreators },
    dispatch
  )
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

- Inject `todos` of a specific user depending on props

```
import * as actionCreators from './actionCreators'

function mapStateToProps(state, ownProps) {
  return { todos: state.todos[ownProps.userId] }
}

export default connect(mapStateToProps)(TodoApp)
```

- Inject `todos` of a specific user depending on props, and inject `props.userId` into the action

```
import * as actionCreators from './actionCreators'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mergeProps(stateProps, dispatchProps, ownProps) {
  return Object.assign({}, ownProps, {
    todos: stateProps.todos[ownProps.userId],
    addTodo: (text) => dispatchProps.addTodo(ownProps.userId, text),
  })
}

export default connect(mapStateToProps, actionCreators, mergeProps)(TodoApp)
```

Notes

The Arity of `mapToProps` Functions

The number of declared function parameters of `mapStateToProps` and `mapDispatchToProps` determines whether they receive `ownProps`

> Note: `ownProps` is not passed to `mapStateToProps` and `mapDispatchToProps` if the formal definition of the function contains one mandatory parameter (function has length 1). For example, functions defined like below won't receive `ownProps` as the second argument. If the incoming value of `ownProps` is `undefined`, the default argument value will be used.

```
function mapStateToProps(state) {
  console.log(state) // state
  console.log(arguments[1]) // undefined
}

const mapStateToProps = (state, ownProps = {}) => {
  console.log(state) // state
  console.log(ownProps) // {}
}
```

Functions with no mandatory parameters or two parameters*will receive `ownProps`.

```
const mapStateToProps = (state, ownProps) => {
  console.log(state) // state
  console.log(ownProps) // ownProps
}

function mapStateToProps() {
  console.log(arguments[0]) // state
  console.log(arguments[1]) // ownProps
}

const mapStateToProps = (...args) => {
  console.log(args[0]) // state
  console.log(args[1]) // ownProps
}
```

Factory Functions

If your `mapStateToProps` or `mapDispatchToProps` functions return a function, they will be called once when the component instantiates, and their returns will be used as the actual `mapStateToProps`, `mapDispatchToProps`, functions respectively, in their subsequent calls.

The factory functions are commonly used with memoized selectors. This gives you the ability to create component-instance-specific selectors inside the closure:

```
const makeUniqueSelectorInstance = () =>
  createSelector([selectItems, selectItemId], (items, itemId) =>
items[itemId])
const makeMapState = (state) => {
  const selectItemForThisComponent = makeUniqueSelectorInstance()
  return function realMapState(state, ownProps) {
    const item = selectItemForThisComponent(state, ownProps.itemId)
    return { item }
  }
}
export default connect(makeMapState)(SomeComponent)
```

Legacy Version Docs

While the `connect` API has stayed almost entirely API-compatible between all of our major versions, there have been some small changes in options and behavior from version to version.

Pour plus de détails sur les anciennes versions 5.x et 6.x, veuillez consulter ces fichiers archivés dans le référentiel React Redux :

- `connect` Référence API 5.x
- `connect` Référence API 6.x

Modifier cette page

*Dernière mise à jour le**03/11/2021***