ISSN: 2710-0588

Practical Approaches to Developing High-performance Web Applications Based on React

Yiting Gu
Publicis Sapient, 6021 Connection Dr, Irving, TX, 75063, US

Abstract

This article takes an in-depth look at how to improve the performance of Web applications based on the React framework, and analyzes common performance bottlenecks, including single-page application performance degradation, UI and data synchronization complexity, state management challenges, and component rerendering issues. Some strategies such as lazy application loading, interface virtualization, optimizing Redux performance and shouldComponentUpdate are proposed to enhance the response speed and user interaction experience of React applications. This research provides React developers with practical performance optimization techniques and has significant application value.

Keywords

React; High performance; Web application; Optimization strategy; Performance bottleneck.

1. Introduction

As Web applications continue to become more complex, improving performance has become a major challenge in the programming process. As a popular front-end framework, React is widely used in building single page applications (SPAs). But when building large applications, React often faces performance bottlenecks, such as slow load times, frequent rendering, and data synchronization challenges. This article aims to analyze the common performance challenges of React applications and give corresponding optimization strategies to help developers get practical guidance on optimizing the performance of web applications.

2. React's role in high-performance Web development

As a front-end framework with declarative programming and componentized design at its core, React plays an integral role in creating high-performance Web applications. Its notable feature is the use of virtual DOM mechanism, which improves rendering performance by reducing the operation of actual DOM elements. React can intelligently compare the differences before and after DOM update, update only the changed part, avoid global refresh, and improve the response time of the page. The componentized development pattern makes the structure of the application more modular, which improves code reusability, increases code elasticity, and helps with performance optimization. In the React framework, its lifecycle management and asynchronous rendering (i.e. React Fiber architecture) effectively reduce the frequency of page rendering, but also alleviate the interface latency phenomenon. The tool set around React, such as React Router and Redux, also brings performance improvements for data processing and routing switching. Taken together, React, through efficient rendering, componentized design, and rich ecological support, greatly facilitates developers to create superior performance and responsive web applications.

React has won the favor of developers with its key features such as virtual DOM, lifecycle management, and asynchronous rendering, but also gives developers great optimization

ISSN: 2710-0588

potential in terms of flexibility and scalability. In the React framework, developers can control the rendering cycle of components and use React optimization methods such as shouldComponentUpdate, React.memo, useMemo, and useCallback to avoid unnecessary component rerendering and further improve performance. React.memo can be based on the comparison of components before and after props, effectively preventing multiple rendering when props are unchanged, reducing the rendering burden. With useMemo and useCallback, the storage of calculation results and functions can be achieved to avoid unnecessary recalculation in each rendering, which is of great benefit to improving the performance of scenes with complex logic and frequent rendering. In addition, React has efficient error handling functions, and the ErrorBoundary component can intercept and process JavaScript exceptions thrown by sub-components to avoid program crashes.

3. Challenges in React high-performance Web application development

3.1. Long runtime and performance degradation in single-page applications

Single page applications (SPAs) rely on JavaScript to update content in real time after the first page load, eliminating the need for frequent page reloads. As application usage time increases, especially when application state changes frequently, performance bottlenecks begin to become apparent. Long-run SPAs can experience performance degradation, especially when dealing with large data interactions and complex component rendering. As states and components increase, the rendering time of the page gradually increases, which may cause the application to be slow and affect the user's operating experience. The rerendering of components and the continuous accumulation of DOM nodes will increase the rendering pressure of the browser and affect the flow of the application. Table 1 below shows how the performance of a single page application varies over time:

Table 1. Table of single-page application performance degradation issues

Use period	Render time (ms)	Number of state changes	The number of times the component rerenders	Page response speed (seconds)
0-1Hour (s)	50	100	20	0.1
1-3Hour (s)	100	300	60	0.3
3-6Hour (s)	200	600	120	0.6
6More than hours	500	1200	240	1.5

3.2. Complexity of UI and data synchronization

In React applications, synchronizing the interface with data is a complex challenge, especially in multi-component and state management frameworks such as Redux. When multiple components depend on the same data source, frequent state changes can cause unnecessary rerendering that affects performance and may cause flickering or inconsistent displays. At the same time, the diverse requirements of different components for the same data also make synchronization more complex, especially when dealing with large amounts of data or frequent updates. Without the right optimization strategy, the responsiveness and smoothness of your app can be compromised. Table 2 below shows how UI rendering and performance changes under different data synchronization strategies, reflecting the challenges posed by the complexity of synchronization:

ISSN: 2710-0588

Table 2. Comparison of UI rendering time and performance under different data synchronization strategies

Synchronization strategy	Data update frequency	UI Render time (ms)	Performance impact (response delay)
Frequent synchronization (no optimization)	high	180	high
Optimized using useEffect	In the	120	In the
Optimize with Redux	low	80	low

3.3. Performance Bottlenecks in Status Management

Dealing with state can be particularly tricky in the face of complex React applications. Especially when building a large application, the state interaction between components is frequent, which is easy to cause unnecessary rendering and affect performance. For example, if there are many subcomponents that depend on a state, a change in that state will cause updates for all related components, even if the actual rendering of those components is not affected by the state. This problem is especially prominent in large projects, and the efficiency and accuracy of state processing are directly related to the response speed of the program and the interactive experience of the user.

Redux, as a popular state management framework, does a great job of managing application state, but its performance limitations are equally obvious. When the dispatch operation is triggered frequently, the global state tree is fully updated, and the mapStateToProps function transmits state changes to the individual components, causing a total recalculation and rendering. This repetitive rendering process consumes computing power and can make the user interface unresponsive. Especially when the component structure is complex and the state level is various, the performance bottleneck is more significant. Worse, if the state structure is too nested, a change in state can spread to multiple components, and the re-rendering of those components can cause more components to update, creating an inefficient feedback loop.

In addition, while hooks like React's useState and useReducer provide convenience in state management, they can still cause unnecessary rendering due to frequent state changes. When your application is dealing with large amounts of data or frequently updating the UI, over-reliance on this mode of state change can significantly degrade your application's performance. During React development, it is critical to explore how to efficiently manage state and optimize updates to reduce rerendering of unrelated components.

3.4. Overfrequency of component rerendering

During the development of React, overrendering of components was a key factor causing performance issues. Every time a component's state or props change, React will perform a rerendering operation, and if this operation is too frequent, especially when building complex interfaces or large applications, it will significantly slow down the performance of the application. Frequent rendering will occupy a large amount of computing resources and memory, but also reduce the interactive reaction speed of the user interface, affecting the final quality of the experience. Overrendering can occur for a variety of reasons, including unnecessary state updates, unnecessary child component updates due to parent component rerendering, and incorrect use of setState and useEffect. Table 3 below shows the performance impact of different component render frequencies:

ISSN: 2710-0588

Table 3. Effects of component rerender frequency on performance

Component type	Render frequency	Performance impact
High-frequency update components	high	Significant decline
Low-frequency update components	low	Good performance
Unnecessary rerendering	high	Serious performance degradation

4. React Practice Optimization Strategy for High-Performance Web Application Development

4.1. Implementation of lazy loading and on-demand loading

Lazy loading and on-demand loading are key strategies for optimizing the performance of React applications, especially when building large Web applications. The essence of these two strategies is to reduce the resource burden of startup loading, and speed up the display efficiency of the page by asynchronously loading the code and resources in a timely manner. Lazy loading is to delay the loading time of components and only perform loading when components must be displayed in the view. This can prevent all resources from being loaded at once and reduce the loading time of the home page. Through the user's operation or page needs to introduce the corresponding module or page content in a timely manner, this on-demand loading avoids the full loading of the entire application. With this strategy, developers are able to subdivide the application into many micro-modules, and the corresponding code is activated and loaded only when the user is actually interacting with the page, which significantly reduces the loading time at app launch and optimizes the user's operating experience. To quantify the optimization effect of lazy versus on-demand loading, the following formula can be used:

$$T_{optimized} = T_{initial} + \sum_{i=1}^{n} T_{lazy_i} \quad (1)$$

Where, $T_{optimized}$ is the total load time after optimization, $T_{initial}$ is the initial load time without optimization, T_{lazy_i} is the loading time of the i th lazy load module, and n is the number of modules that need to be lazy loaded. With these strategies, React applications can selectively load a large number of unnecessary resources, reduce redundant network requests and rendering burdens, improve page loading efficiency and response speed, and enhance user access experience. This combination of on-demand and lazy loading enhances performance and reduces application resource consumption, ensuring that React applications operate smoothly under heavy load conditions.

4.2. Dynamic rendering and virtualization technology

Dynamic rendering and virtualization are essential to optimize the performance of React applications, especially when dealing with large data sets or building complex user interfaces. This technology can load and display page components in real time according to user interaction, effectively avoiding unnecessary DOM operations, and greatly improving rendering efficiency. Virtualization technology also has the same wonderful, it only for the user is currently visible part of the interface rendering processing, the rest of the need to load or clear from the memory, so that both reduce the rendering burden, reduce memory consumption, to achieve significant performance enhancement. In the react framework, virtual rendering is often performed with the help of third-party libraries such as React-Window or React-

ISSN: 2710-0588

Virtualized to improve performance. These libraries can efficiently calculate the elements in the current viewport and update the display in real time based on the user's scrolling actions and view size, thereby reducing the unnecessary immediate loading of a large number of DOM nodes. When dealing with long sequences, continuous scrolling, or complex data lists, virtualization technology is particularly suitable, it can significantly reduce the total number of elements required by the browser to draw, relieve browser pressure, and speed up the response of the page. To quantify the impact of virtualization on performance, the optimization can be evaluated using the following formula:

$$T_{optimized} - T_{initial} - \left(\sum_{i=1}^{m} T_{rendered}\right) \tag{2}$$

Where $T_{optimized}$ is the total render time after optimization, $T_{initial}$ is the total render time without optimization, $T_{rendered}$ is the render time of each rendered visual area element, and m is the actual number of elements rendered during each view update. With these techniques, React applications can significantly reduce the number of unnecessary renderings to optimize page performance and user experience. When dealing with large amounts of data or building complex page scenes, these techniques can significantly increase the efficiency of rendering.

4.3. Optimizing Redux Performance

As React applications continue to expand, performance issues tend to become more prominent when Redux is used for state management. As the application grows in size, status updates are frequent and accompanied by unnecessary component rendering, which can make the application less responsive. This is especially true when many components simultaneously depend on a single state. To overcome these challenges, it is necessary to optimize Redux's status update process, reduce the number of redundant redraws, and increase the speed of data processing. For example, components are automatically re-rendered whenever state changes, but sometimes this rendering does not result in any substantial change to the user interface. To avoid unnecessary rendering, you can optimize component update frequency with React.memo or shouldComponentUpdate to prevent unnecessary interface overloading. Furthermore, building a memorized selector with the reselect library can reduce redundant computation tasks and improve application performance by ensuring that recalculations and renderings are only performed when the relevant data is updated. At the level of optimized data storage, the state update process can be made more efficient by splitting the state and reducing the size of the data involved in each state update. In addition, batch updates can also greatly improve performance, because it prevents separate updates from being triggered for each distribution operation, reducing the frequency of rendering. Assuming the time required for each render is T_render, the number of components is N, and the number of updates is U, then the total render time T_total is:

$$T_{total} = T_{render} \times N \times U \tag{4}$$

After optimization, the render times are reduced to N', the update times are reduced to U', and the optimized render time is:

$$T_{total\ \ optimized} = T_{render} \times N' \times U' \tag{5}$$

ISSN: 2710-0588

The optimized rendering time is significantly reduced, resulting in improved application responsiveness and user experience.

4.4. shouldComponentUpdate Should be Used to optimize rendering

In React framework, shouldComponentUpdate is an important means to improve optimized rendering. This method gives the developer the discretion to determine whether a component needs to be redrawn to prevent unnecessary updates from affecting program performance. React typically performs redraws whenever a component's props or state changes, although some changes do not necessarily affect the component's user interface. shouldComponentUpdate allows developers to improve application performance by avoiding unnecessary rendering. This function takes nextProps and nextState as input arguments and outputs a result of type Boolean. When the function returns a true value, the component performs a redraw. Conversely, if a false value is returned, the component skips the redrawing process. For performance optimization purposes, the shouldComponentUpdate function should return true only when a component's properties or state have actually changed to reduce unnecessary redraws.

In this way, developers can reduce unnecessary rendering, reduce computing resource consumption, and improve page responsiveness. shouldComponentUpdate should be used to calculate the difference between the optimized render time and the initial render time. The optimization formula is as follows:

$$T_{optimized} = T_{initial} - \left(\sum_{i=1}^{m} \Delta T_{rendered}\right) \tag{6}$$

Where $T_{optimized}$ represents the optimized rendering time, $T_{initial}$ represents the unoptimized rendering time, $\Delta T_{rendered}$ is the time reduced per rendering, and m is the number of actual updates. With this strategy, the performance of React applications can be significantly improved.

Conclusion: The core of improving the performance of React Web applications is to reduce unnecessary rendering and speed up page response. Adopting lazy loading, virtualization, and optimized state management can help prevent performance bottlenecks and increase operational efficiency. At the same time, moderately adjust the frequency of component updates, and adopt the appropriate performance optimization strategy, so that the application can maintain a smooth experience even when dealing with complex business and large amounts of data. As the React technology ecosystem continues to improve, more advanced technologies are expected to emerge to drive performance optimization.

References

[1] Yamanaka I .Measurement and Experimental Procedures in Electrosynthesis Reaction for Low Molecular Weight Reactants[J].Denki Kagaku, 2022, 90(4):351-356.

[2] Zhang J , Yin Q , Hu W ,et al.EPA : The effective pipeline architecture for CNN accelerator with high performance and computing efficiency based on FPGA[J].Concurrency and Computation Practice and Experience, 2021(2).

[3] Xu H , Wang C R , Berres A ,et al.Interactive Web Application for Traffic Simulation Data Management and Visualization:[J].Transportation Research Record, 2022, 2676(1):274-292.

[4] Qiu Y , Yin W , Wang L .A High-Performance and Scalable NVMe Controller Featuring Hardware Acceleration[J].IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems: A publication of the IEEE Circuits and Systems Society, 2022(5):41.

ISSN: 2710-0588

[5] Zhang H , Liu Z , Hasan S ,et al.Joint optimization strategy of heterogeneous resources in multi-MEC-server vehicular network[J].Wireless Networks, 2022, 28(2):765-778.

[6] Min B ,Ly H .How do gold and oil react to the COVID-19 pandemic: A review[J].Energy & Environment, 2023,34(7):2876-2902.

[7] Qin M H ,Takayuki K ,Michael C , et al.Reacting Mn3O4 powders with quaternary ammonium hydroxides to form two-dimensional birnessite flakes[J].Ceramics International, 2023, 49(21): 33537-33545.