antd 5.0 定制主题如此酷炫，我决定开启 @ant-design/cssinjs 阅读之旅

前言

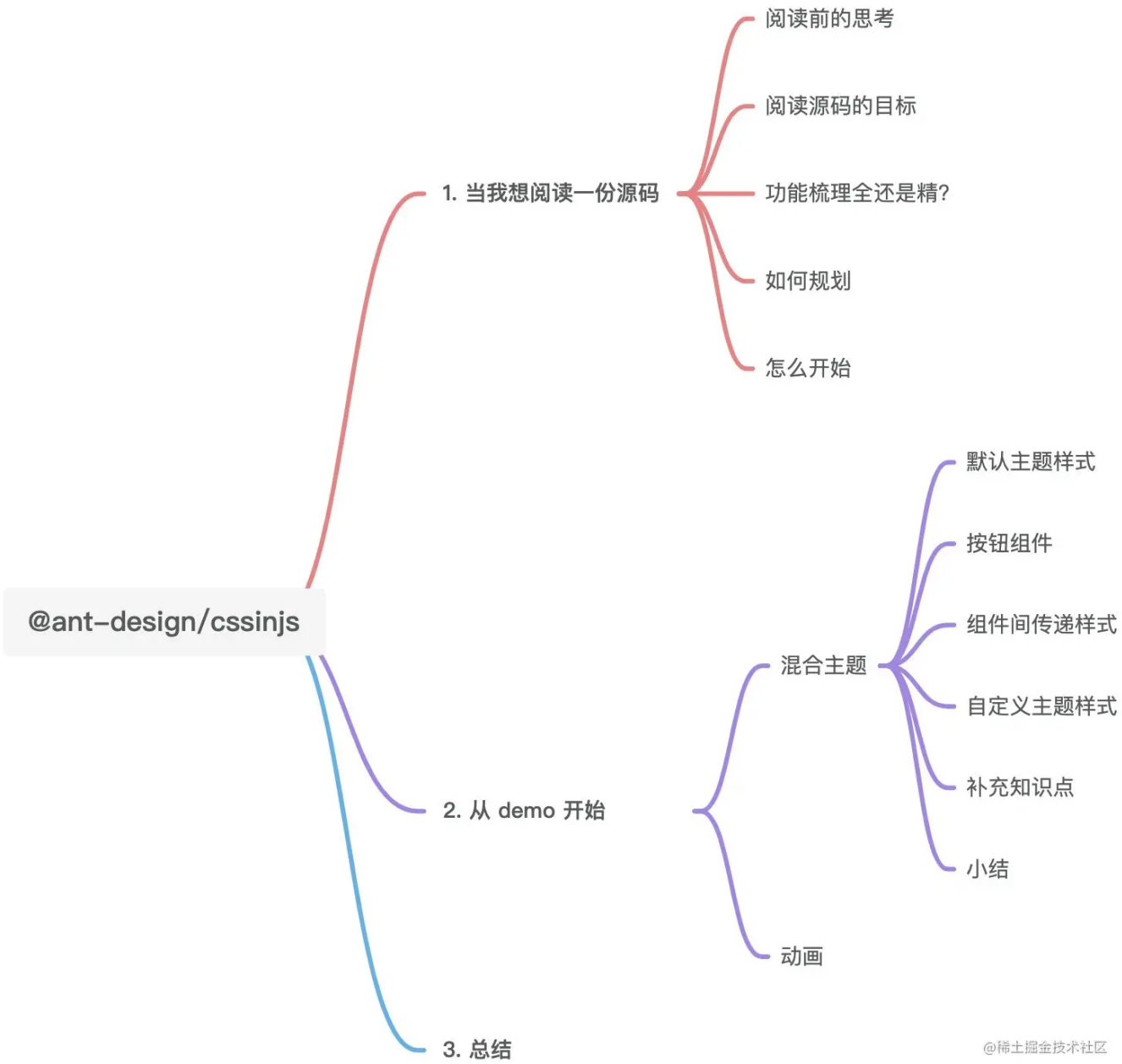
antd 5.0 正式发布已经有一段时间了，发布当天，一心二用的看完直播。很喜欢整个设计，有种简约快乐工作的感觉，某些功能设计初衷也给了我一些启发。

antd 5.0 提供的主题定制挺酷炫的，加之我最近对「CSS-in-JS」很感兴趣。于是迫不及待的打开了它的源码，准备研究一番 。

我大部分情况下都是通过碎片化的时间来研究技术，所以时间合理配置和任务合理分块，一直是我常采用的方式。加上对源码阅读的经验不足，所以此次的阅读之旅，我将详细记录阅读前的思考、阅读规划以及收获，并将破冰心得总结之后分享出来。

文章速读

阅读文章，可以有以下收获：



当我想阅读一份源码

我源码阅读的经验较少，大部分时候阅读源码的目的是寻找文档上没有写的参数或者API。

之前有过几个完整的阅读经验，但是源码项目相对简单，阅读过程快速且顺畅。

@ant-design/cssinjs 的源码，由于其文档内容较少，很多参数单纯靠代码内容无法确定其准确的结构，所以在进一步的研究之前，我进行了一场深思。

「阅读前的思考」

我认真思考了几个问题：

- 我阅读这个源码的目标是什么?

- 我要把所有功能梳理的一清二楚吗？
- 遇到复杂的源码，没有完整的时间，我应该怎么合理做规划？
- 明确方案之后，我具体应该怎么去实现？
- 源码阅读完之后呢？仅仅写篇文章，还是真的应用到工作中？

阅读源码的目标

对于 @ant-design/cssinjs，我的目标很明确，搞懂它是如何实现动态主题的。

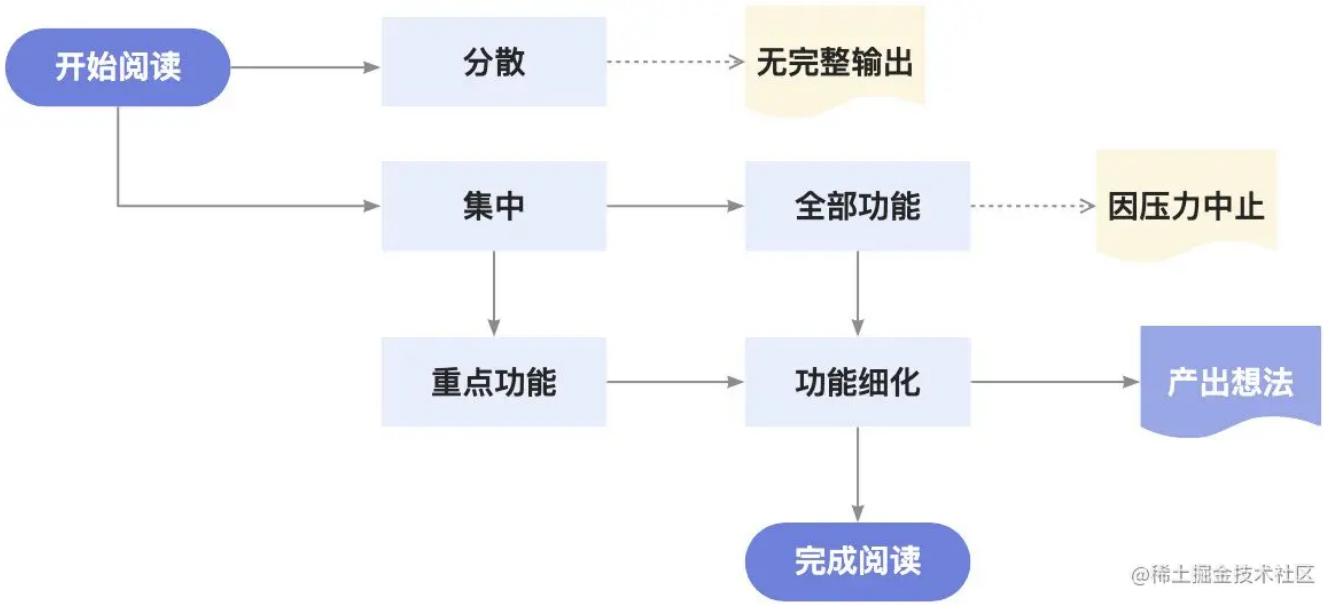
功能梳理全还是精？

如果我不明确这个问题，我有可能读的时候会分散，看到哪，读到哪。

如果我要梳理完全，可能会无形中给自己很多压力。

如果我只看重点功能，那我怎么确定哪些是重点功能呢？

如下为**功能梳理的全部情况流程图**



无论是全部读完，还是挑重点功能阅读，都推荐进行功能细化，将事情拆分成多个小项。

如何规划

我习惯分类归纳学习经验，按照技术类型区分。这次源码阅读虽然有之前的经验，但是之前的源码整体难度不是很高。后面可能还会阅读内容更为复杂的开源项目，现阶段还是一个积累经验的过程。

这次是一个很好的尝试，后面我会归纳出**一套完整的经验**。

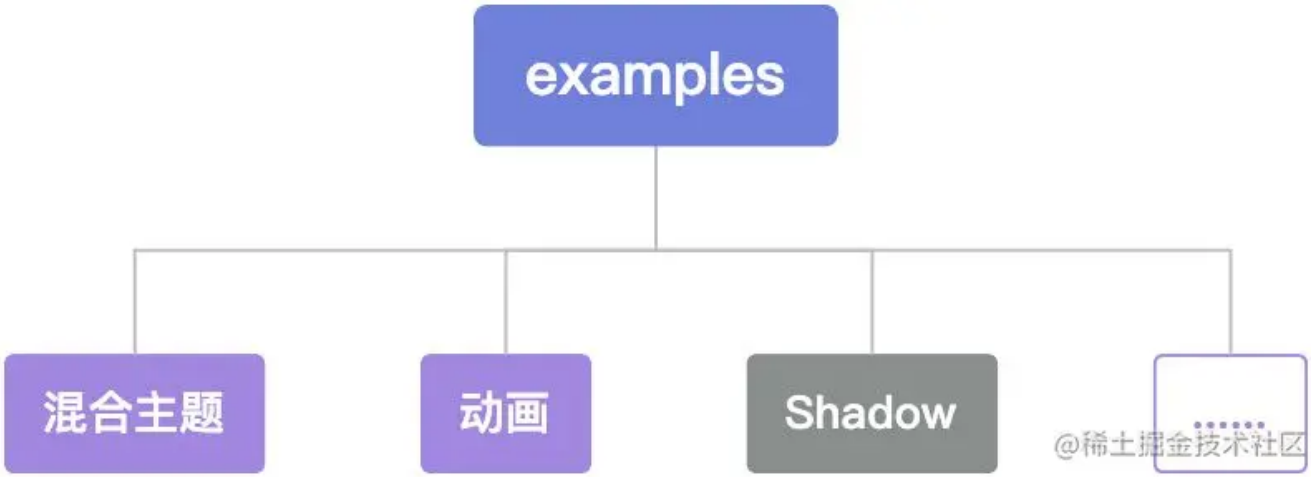
「怎么开始」

我本来尝试从测试用例开始，发现运行之后不是我想要的结果，我想到其实还有一条路可以试试，那就是**demo**。

通过 demo 既可以**查看效果**，又可以**打印各种参数**。

从 demo 开始

前期的准备工作就绪，正式开启阅读之旅。



「混合主题」

» 默认主题样式

scss 复制代码

```scss
// docs/examples/components/theme.tsx

// 默认主题样式
const defaultDesignToken: DesignToken = {
  primaryColor: '#1890ff',
  textColor: '#333333',
  reverseTextColor: '#FFFFFF',

  componentBackgroundColor: '#FFFFFF',

  borderRadius: 2,
  borderColor: 'black',
  borderWidth: 1,
};
```

这个默认主题样式，通过变量名大致能猜出来各自代表什么。但是具体**应用**在哪些元素上，还需要通过**使用设置**才能确定。

» 按钮组件

typescript 复制代码

```typescript
// docs/examples/components/Button.tsx

// 按钮组件
import React from 'react';
import classNames from 'classnames';
import { useToken } from './theme';
import type { DerivativeToken } from './theme';
import { useStyleRegister } from '../../../src/';
import type { CSSInterpolation, CSSObject } from '../../../src/';

// 通用框架
const genSharedButtonStyle = (
  prefixCls: string,
  token: DerivativeToken,
): CSSInterpolation => ({
  [`.${prefixCls}`]: {
    borderColor: token.borderColor, // 边框颜色
    borderWidth: token.borderWidth, // 边框宽度
    borderRadius: token.borderRadius, // 边框圆角

    cursor: 'pointer',

    transition: 'background 0.3s',
  },
});

// 实心底色样式
// 返回数组，第一个元素是通用样式，第二个元素是自定义样式，需要调用者传入
const genSolidButtonStyle = (
  prefixCls: string,
  token: DerivativeToken,
  postFn: () => CSSObject,
): CSSInterpolation => [
  genSharedButtonStyle(prefixCls, token),
  {
    [`.${prefixCls}`]: {
      ...postFn(),
    },
  },
];

// 默认样式
const genDefaultButtonStyle = (
```

```
  prefixCls: string,
  token: DerivativeToken,
): CSSInterpolation => {
  genSolidButtonStyle(prefixCls, token, () => ({
    backgroundColor: token.componentBackgroundColor, // 默认样式的背景颜色
    color: token.textColor, // 默认样式的字体颜色

    '&:hover': {
      borderColor: token.primaryColor, // 默认样式的经过时边框颜色
      color: token.primaryColor, // 默认样式的经过时字体颜色
    },
  }));
};

// 主色样式
const genPrimaryButtonStyle = (
  prefixCls: string,
  token: DerivativeToken,
): CSSInterpolation =>
  genSolidButtonStyle(prefixCls, token, () => ({
    backgroundColor: token.primaryColor, // 主色样式的背景颜色
    border: `${token.borderWidth}px solid ${token.primaryColor}`, // 主色样式的边框样式
    color: token.reverseTextColor, // 主色样式的字体样式

    '&:hover': {
      backgroundColor: token.primaryColorDisabled, // 主色样式的经过时背景颜色
    },
  }));

// 透明按钮
const genGhostButtonStyle = (
  prefixCls: string,
  token: DerivativeToken,
): CSSInterpolation => [
  genSharedButtonStyle(prefixCls, token),
  {
    [`.${prefixCls}`]: {
      backgroundColor: 'transparent', // 透明样式的背景颜色
      color: token.primaryColor, // 透明样式的字体颜色
      border: `${token.borderWidth}px solid ${token.primaryColor}`, // 透明样式的边框颜色

      '&:hover': {
        borderColor: token.primaryColor, // 透明样式的经过时背景颜色
        color: token.primaryColor,
      },
    },
  },
];
```

```tsx
interface ButtonProps
  extends Omit<React.ButtonHTMLAttributes<HTMLButtonElement>, 'type'> {
  type?: 'primary' | 'ghost' | 'dashed' | 'link' | 'text' | 'default';
}

const Button = ({ className, type, ...restProps }: ButtonProps) => {
  const prefixCls = 'ant-btn';

  // 【自定义】制造样式
  const [theme, token, hashId] = useToken();

  // default 添加默认样式选择器后可以省很多冲突解决问题
  const defaultCls = `${prefixCls}-default`;
  const primaryCls = `${prefixCls}-primary`;
  const ghostCls = `${prefixCls}-ghost`;

  // 全局注册，内部会做缓存优化
        // 目前是三种类型的样式：默认样式、主色样式、透明样式
  const wrapSSR = useStyleRegister(
    { theme, token, hashId, path: [prefixCls] },
    () => [
      genDefaultButtonStyle(defaultCls, token),
      genPrimaryButtonStyle(primaryCls, token),
      genGhostButtonStyle(ghostCls, token),
    ],
  );

  const typeCls =
    (
      {
        primary: primaryCls,
        ghost: ghostCls,
      } as any
    )[type as any] || defaultCls;

  return wrapSSR(
    <button
      className={classNames(prefixCls, typeCls, hashId, className)}
      {...restProps}
    />,
  );
};

export default Button;
```

按钮组件中，主要包含三种主题，样式做了**通用设置**。从目前的代码看 genGhostButtonStyle 的设置其实和另外两个是一样的。

而通用样式时的取值来自 **useToken**，useToken 则是采用组件树间进行数据传递的方式。

» 组件间传递样式

typescript 复制代码

```typescript
// docs/examples/components/theme.tsx

// 创建一个 Context 对象 ThemeContext
export const ThemeContext = React.createContext(createTheme(derivative));

// 创建一个 Context 对象 DesignTokenContext
export const DesignTokenContext = React.createContext<{
  token?: Partial<DesignToken>;
  hashed?: string | boolean;
}>({
  token: defaultDesignToken,
});

/**
 * 创建默认样式，并缓存 token
 * @returns 包含 theme, token, hashed 的数组对象
 */
export function useToken(): [Theme<any, any>, DerivativeToken, string] {
  // 订阅 DesignTokenContext
        // 将此处的 token 重命名为 rootDesignToken，并设置默认值 defaultDesignToken
  const { token: rootDesignToken = defaultDesignToken, hashed } =
    React.useContext(DesignTokenContext);
  // 订阅 ThemeContext
  const theme = React.useContext(ThemeContext);

  // 将 theme 派生的 token 缓存为全局共享 token
        // 实际 token 的取值
  const [token, hashId] = useCacheToken<DerivativeToken, DesignToken>(
    theme,
    [defaultDesignToken, rootDesignToken],
    {
      salt: typeof hashed === 'string' ? hashed : '',
    },
  );

  return [theme, token, hashed ? hashId : ''];
}
```

这里有一处处理需要注意📢，实际 token 不是从 DesignTokenContext 中取的，而是通过 useCacheToken 获得的。这是一个全局共享 token 的处理，token 中额外增加了参数。

> 注：具体为什么要这么处理以及怎么处理的，等我后续做进一步的研究之后，会持续输出分享。

» 自定义主题样式

除了默认主题，可以进行自定义主题的设置

javascript 复制代码

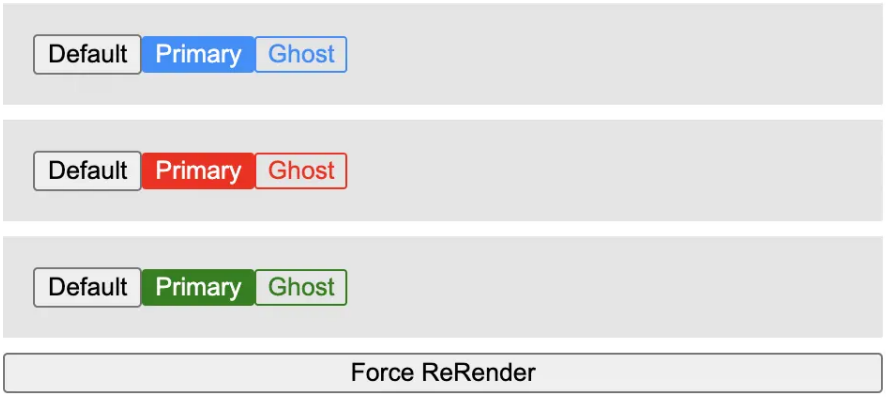
```javascript
// docs/examples/theme.tsx

import React from 'react';
import Button from './components/Button';
import { DesignTokenContext, ThemeContext } from './components/theme';
import type { DesignToken, DerivativeToken } from './components/theme';
import { createTheme } from '../../src';

function derivativeA(designToken: DesignToken): DerivativeToken {
  return {
    ...designToken,
    primaryColor: 'red', // 主色样式的背景颜色
    primaryColorDisabled: 'red', // 主色样式的经过时背景颜色
  };
}

function derivativeB(designToken: DesignToken): DerivativeToken {
  return {
    ...designToken,
    primaryColor: 'green',
    primaryColorDisabled: 'green',
  };
}

const ButtonList = () => (
  <div style={{ background: 'rgba(0,0,0,0.1)', padding: 16 }}>
    <Button>Default</Button>
    <Button type="primary">Primary</Button>
    <Button type="ghost">Ghost</Button>
  </div>
);

export default function App() {
  const [, forceUpdate] = React.useState({});

  return (
    <div style={{ display: 'flex', flexDirection: 'column', rowGap: 8 }}>
      <h3>混合主题</h3>

      <DesignTokenContext.Provider value={{ hashed: true }}>
        <ButtonList />
        {/*
         * 使用一个 Provider 来将当前的 theme 传递给以下的组件树。
         * 将 derivativeA 生成的 theme 作为当前值传下去
         * */}
        <ThemeContext.Provider value={createTheme(derivativeA)}>
          <ButtonList />
```

```
        </ThemeContext.Provider>

        <ThemeContext.Provider value={createTheme(derivativeB)}>
          <ButtonList />
        </ThemeContext.Provider>
      </DesignTokenContext.Provider>

      <button
        onClick={() => {
          forceUpdate({});
        }}
      >
        Force ReRender
      </button>
    </div>
  );
}
```

前面分析了按钮组件中样式设置，这里不难看出，自定义主题样式，更改了主色样式的背景颜色和经过时的背景颜色。

» 混合主题 UI

混合主题



» 补充知识点

createContext

使用 createContext 创建 Context 对象之后，当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 Provider 中读取到当前的 context 值。

useContext

接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值。

当组件上层最近的 <MyContext.Provider> 更新时，该 Hook 会触发重渲染，并使用最新传递给 MyContext provider 的 context value 值。

解构时的别名

解构时，可以使用冒号为变量赋予别名。

```ini
let DesignTokenContext = {
  token: 'rename',
};
const defaultDesignToken = 'default';
const { token: rootDesignToken = defaultDesignToken } = DesignTokenContext;
console.log(rootDesignToken, 'rootDesignToken');
```

此时打印结果

```arduino
// > rename
```

别名的实际应用

> 可以帮助动态取值的过程中避免相同变量名的值冲突。

» 小结

1. 在按钮组件中，主要包含三种主题。不同主题主要由通用样式和特定样式组成，源码中用 genSolidButtonStyle 函数进行了统一的设置。
2. Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。所以，目前常见 createContext 的方式管理组件间的数据。
3. 对于 token 的处理，还需要进一步的研究。
4. 刚开始梳理我花了不少时间，那会心有点急，后面稳下心神，思路逐渐明朗。**找准方向**，渐入佳境。

「动画」

动画的功能较主题简单一些，主要看看动画组件的功能。

» 动画组件

```typescript
// docs/examples/components/Spin.tsx
```

```tsx
import React from 'react';
import classNames from 'classnames';
import { useToken } from './theme';
import type { DerivativeToken } from './theme';
import { useStyleRegister, Keyframes } from '../../../src/';
import type { CSSInterpolation } from '../../../src/';

// 设置动画的类名和 style 值
const animation = new Keyframes('loadingCircle', {
  to: {
    transform: `rotate(360deg)`,
  },
});

// 通用框架
const genSpinStyle = (
  prefixCls: string,
  token: DerivativeToken,
  hashId: string,
): CSSInterpolation => [
  {
    [`.${prefixCls}`]: {
      width: 20,
      height: 20,
      backgroundColor: token.primaryColor,
        // getName 方法帮助组合最终的类名：hashId-name
      animation: `${animation.getName(hashId)} 1s infinite linear`,
    },
  },
  animation,
];

interface SpinProps extends React.HTMLAttributes<HTMLDivElement> {}

const Spin = ({ className, ...restProps }: SpinProps) => {
  const prefixCls = 'ant-spin';

  // 【自定义】制造样式
  const [theme, token, hashId] = useToken();

  // 全局注册，内部会做缓存优化
  const wrapSSR = useStyleRegister(
    { theme, token, hashId, path: [prefixCls] },
    () => [genSpinStyle(prefixCls, token, hashId)],
  );

  return wrapSSR(
    <div className={classNames(prefixCls, hashId, className)} {...restProps} />,
  );
```

```
};

export default Spin;
```

动画组件中，除了设置动画样式，也需要处理animation中 keyframe 的值来**绑定对应的选择器**。