

一份 2.5k star 的《React 开发思想纲领》

0. 介绍

《React 开发思想纲领》是：

- 我开发 **React** 时的一些思考
- 每当我 review 他人或自己的代码时自然而然会思考的东西
- 仅仅作参考和建议，并非严格的要求
- 会随着我的经验不断更新
- 大多数技术点是基础的[重构方法论](#)，[SOLID 原则](#)以及[极限编程](#)等思想的变体，仅仅是在 **React** 中的实践而已 😊

你可能会觉得我写的这些非常基础。但以下示例都来自一些复杂大型项目的线上代码。

《React 开发思想纲领》的灵感来源于我实际开发中遇到的各种场景。

1. 最低要求

1.1 计算机比你更「智能」

1. 使用 [ESLint](#) 来静态分析你的代码，开启 [rule-of-hooks](#) 和 [exhaustive-deps](#) 这两个规则来捕获 **React** 错误。
2. 开启 JS [严格模式](#) 吧，都 2022 年了。
3. [直面依赖](#)，解决在 [useMemo](#)，[useCallback](#) 和 [useEffect](#) 上 [exhaustive-deps](#) 规则提示的 warning 或 error 问题。可以将最新的值挂在 [ref](#) 上来保证这些 hook 在回调中拿到的都是最新的值，同时避免不必要的重新渲染。
4. 使用 [map](#) 批量渲染组件时，都加上 [key](#)。
5. 只在最顶层使用 [hook](#)，不要在循环、条件或嵌套语句中使用 hook。
6. 理解“不能对已经卸载的组件执行状态更新”的控制台警告。[facebook/react/pull/22114](#)
7. 给不同层级的组件都添加[错误边界 \(Error Boundary\)](#) 来防止[白屏](#)，还可以用它来向错误监控平台（比如 [Sentry](#)）上报错误，并设置报警。
8. 不要忽略了控制台中打印的错误和警告。
9. 记得要 [tree-shaking](#) !

10. 使用 [Prettier](#) 来保证代码的格式化一致性!
11. 使用 [Typescript](#) 和 [NextJS](#) 这样的框架来提升开发体验。
12. 强烈推荐 [Code Climate](#) (或其他类似的) 开源库。这类工具会自动检测代码异味 (Code Smell, 代码中的任何可能导致深层次问题的症状), 它可以促使我去处理项目里留下的技术债。

1.2 Code is just a necessary evil (代码只是一种必要的邪恶)

译者注: 程序员的目标是解决客户的问题, 代码只是副产品

"The best code is no code at all. Every new line of code you willingly bring into the world is code that has to be debugged, code that has to be read and understood, code that has to be supported." - Jeff Atwood

"最好的代码就是无代码。每一行新的代码都需要被调试, 需要被阅读和理解, 与支持。"

"One of my most productive days was throwing away 1000 lines of code." - Eric S. Raymond

"我最有效率的一天就是扔掉 1000 行代码"

参考: [Write Less Code \(少写代码\) - Rich Harris](#), [Code is evil \(代码是邪恶的\) - Artem Sapegin](#)

1.2.1 先思考, 再加依赖

依赖加的越多, 提供给浏览器的代码就越多。扪心问问自己, 你是否真的使用了某个库的 feature?

 **你真的需要它吗?** 看看这些你可能不需要的依赖

1. 你是否真的需要 [Redux](#)? 有可能需要, 但其实 React 本身也是一个[状态管理库](#)。
2. 你是否真的需要 [Apollo client](#)? [Apollo client](#) 有许多很强大的功能, 比如数据规范化。但使用的同时也会显著提高包体积。如果你的项目使用的并非是 [Apollo client](#) 特有的 feature, 可以考虑使用一些轻量的库来替代, 比如 [react-query](#) 或 [SWR](#) (或者根本不用)。

3. [Axios](#) 呢？Axios 是一个很棒的库，它的一些特性不容易通过原生的 [fetch](#) API 来复刻。但是如果使用 [Axios](#) 只是因为它有更好的 API，完全可以考虑在 [fetch](#) 上做一层封装（比如 [redaxios](#) 或自己实现）。取决于你的 App 是否真正地使用了 [Axios](#) 的核心 feature。
4. [Decimal.js](#) 呢？或许 [Big.js](#) 或者[其他轻量的库](#)就足够了。
5. [Lodash](#) / [underscoreJS](#) 呢？推荐你看看【[你不需要系列之“你不需要 Lodash/Underscore”](#)】。
6. [MomentJS](#) 呢？【[你不需要系列之“你不需要 Momentjs”](#)】
7. 你不需要为了主题（[浅色](#) / [深色](#) 模式）而使用 [Context](#)，考虑下用 [css 变量](#) 代替。
8. 你甚至不需要 [Javascript](#)，CSS 也足够强大。【[你不需要系列之“你不需要 JavaScript”](#)】

1.2.2 不要自作聪明，提前设计

"What could happen with my software in the future? Oh yeah, maybe this and that. Let's implement all these things since we are working on this part anyway. That way it's future-proof."

"我们的软件在未来会如何迭代？可能会这样或者那样，如果在当下就开始往这些方向进行代码设计，这就叫 future-proof（防过时，面向未来编程）。"

不要这样搞！ 应该在面临需求的时候再去实现相应功能，而不是在你预见到可能需要的时候。代码应该越少越好！（[Martin Fowler: YAGNI](#), [C2 Wiki: You Arent Gonna Need It!](#)）

相关部分: [2.4 复制比错误的抽象要“便宜”的多](#)

1.3 发现了就优化它

1.3.1 检测代码异味（Code Smell），并在必要时对其进行处理。

当你意识到某个地方出现了问题，那就马上处理掉。但如果当前不容易修复，或者没有时间，那请至少添加一条注释（[FIXME](#) 或者 [TODO](#)），附上对该问题的简要描述。来让项目里的每个人都知道这里有问题，让他们意识到当他们遇到这样的情况时也该这样做。

 **来看看这些容易发现的** [代码异味](#)

- ❌ 定义了很多参数的函数或方法
- ❌ 难以理解的，返回 Boolean 值的逻辑
- ❌ 单个文件中代码行数太多
- ❌ 在语法上可能相同（但格式化可能不同）的重复代码
- ❌ 可能难以理解的函数或方法
- ❌ 定义了大量函数或方法的类/组件
- ❌ 单个函数或方法中的代码行数太多
- ❌ 具有大量返回语句的函数或方法
- ❌ 不完全相同但代码结构类似的重复代码（比如变量名可能不同）

切记，代码异味并不一定意味着代码需要修改，它只是告诉你，你应该可以想出更好的方式来实现相同的功能。

1.3.2 无情的重构。简单比复杂好。

Is the CL more complex than it should be? Check this at every level of the CL—are individual lines too complex? Are functions too complex? Are classes too complex? “Too complex” usually means “can’t be understood quickly by code readers.” It can also mean “developers are likely to introduce bugs when they try to call or modify this code.”- Google Engineering Practices: What to look for in a code review

当 reviewer 在 Code Review (CR) 代码时，ChangeList (CL) 是否比想象中更复杂？某些行是否太复杂？函数是否太复杂？类是否太复杂？“太复杂”意味着“代码的读者不能快速理解”，也意味着“当别人试图变更此处时，容易产生 bug”——[谷歌工程实践：CR 时应该做什么](#)

 **小技巧：简化复杂的条件语句，最好能提前 return。**

tsx 复制代码

```
# ❌ 不太好

if (loading) {
  return <LoadingScreen />
} else if (error) {
  return <ErrorScreen />
} else if (data) {
  return <DataScreen />
} else {
  throw new Error('This should be impossible')
}
```

 推荐

```
if (loading) {  
  return <LoadingScreen />  
}  
  
if (error) {  
  return <ErrorScreen />  
}  
  
if (data) {  
  return <DataScreen />  
}  
  
throw new Error('This should be impossible')
```

 **小技巧: 比起传统的循环语句, 链式的高阶函数更优雅**

如果没有明显的性能差异, 尽量使用链式的高阶函数(`map`, `filter`, `find`, `findIndex`, `some` 等) 来代替传统的循环语句。 [Stackoverflow: 使用函数来代替循环语句的优点是什么?](#)

1.4 你可以做的更好

 **小技巧: 可以在 `setState` 时传入回调函数, 所以没必要把 `state` 作为一个依赖项**

你不用把 `setState` 和 `dispatch` 放在 `useEffect` 和 `useCallback` 这些 hook 的依赖数组中。ESLint 也不会给你提示, 因为 React 已经确保了它们不会出错。

tsx 复制代码

 不太好

```
const decrement = useCallback(() => setCount(count - 1), [setCount, count])  
const decrement = useCallback(() => setCount(count - 1), [count])
```

 推荐

```
const decrement = useCallback(() => setCount(count => (count - 1)), [])
```

 **小技巧: 如果你的 `useMemo` 或 `useCallback` 没有任何依赖, 那你可能用错了**

tsx 复制代码

 不太好

```
const MyComponent = () => {  
  const functionToCall = useCallback(x: string => `Hello ${x}!`, [])  
  const iAmAConstant = useMemo(() => { return {x: 5, y: 2} }, [])  
  /* 接下来可能会用到 functionToCall 和 iAmAConstant */  
}
```

✅ 推荐

```
const I_AM_A_CONSTANT = { x: 5, y: 2 }
const functionToCall = (x: string) => `Hello ${x}!`
const MyComponent = () => {
  /* 接下来可能会用到 functionToCall 和 I_AM_A_CONSTANT */
}
```

🧑 小技巧: 巧用 hook 封装自定义的 context, 会提升 API 可读性

它不仅看起来更清晰, 而且你只需要 import 一次, 而不是两次。

❌ 不太好

```
// 你每次需要 import 两个变量
import { useContext } from 'react';
import { SomethingContext } from 'some-context-package';

function App() {
  const something = useContext(SomethingContext); // 看起来 ok, 但可以更好
  // ...
}
```

tsx 复制代码

✅ 推荐

```
// 在另一个文件中, 定义这个 hook
function useSomething() {
  const context = useContext(SomethingContext);
  if (context === undefined) {
    throw new Error('useSomething must be used within a SomethingProvider');
  }
  return context;
}

// 你只需要 import 一次
import { useSomething } from 'some-context-package';

function App() {
  const something = useSomething(); // 看起来会更清晰
  // ...
}
```

tsx 复制代码

🧑 小技巧: 在写组件之前, 先思考该怎么用它

设计 API 很难, [README 驱动开发 \(RDD\)](#) 是个很有用的办法, 可以帮助你设计出更好的 API。并不是说应该无脑使用 [RDD](#), 但它背后的思想是很值得学习的。我自己发现, 在设计实现组件 API 之前, 使用 RDD 通常比不用时设计地更好。

2. 面向幸福设计

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

"任何傻瓜都能写出计算机能够理解的代码。优秀的程序员写出人能够理解的代码"

"The ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read." - Robert C. Martin (Not saying I agree with his political views)

"花在阅读和写作上的时间比例远远超过 10: 1。作为编写新代码工作的一部分, 我们不断地阅读旧代码。因此如果你想快速完成开发, 让代码可维护性强, 那就提高它的可读性。"

太长不看版

1. ❤️ 通过删除冗余的状态来减少状态管理的复杂性。
2. ❤️ "传递香蕉, 而不是拿着香蕉的大猩猩和整个丛林" (意思是组件要什么传什么, 不要传大对象)。
3. ❤️ 让你的组件小而简单 —— 单一职责原则。
4. ❤️ 复制比错误的抽象要"便宜"的多 (避免提早/不恰当的设计)。
5. 避免 prop 层层传递 (又叫 prop 钻取, prop drilling) ([Michael Jackson](#))。Context 不是解决状态共享问题的银弹。
6. 将巨大的 `useEffect` 拆分成独立的小 `useEffect`。([KCD: Myths about useEffect \(useEffect 谬见\)](#))
7. 将逻辑提取出来都放到 hook 和工具函数中。
8. `useCallback`, `useMemo` 和 `useEffect` 依赖数组中的依赖项最好都是基本类型。
9. 不要在 `useCallback`, `useMemo` 和 `useEffect` 中放入太多的依赖项。
10. 为了简单起见, 如果你的状态依赖其他状态和上次的值, 考虑使用 `useReducer`, 而不是使用很多个 `useState`。

11. **Context** 不一定要放在整个 app 的全局。把 **Context** 放在组件树中尽可能低的位置。同样的道理，你的变量，注释和状态（和普通代码）也应该放在靠近他们被使用的地方。

💖 2.1 删除冗余的状态来减少状态管理的复杂性

冗余的状态指可以通过其他状态经过推导得到的状态，不需要单独维护（类似 Vue computed），当你有冗余的状态时，一些状态可能会丢失同步性，在面对复杂交互的场景时，你可能会忘记更新它们。

删除这些冗余的状态，除了避免同步错误外，这样的代码也更容易维护和推理，而且代码更少。

参考：[KCD: Don't Sync State. Derive It!（不要手动维护状态，最好是通过计算推导！）](#)，[Tic-Tac-Toe（井字棋游戏）](#)

🐵 示例一

📝 业务需求/问题描述

你需要展示一个直角三角形的某些属性：

- 三条边的长度
- 周长
- 面积

从 API 获取了三角形的数据，是一个 `{a: number, b: number}` 类型的对象，表示这个直角三角形的两条短边。

❌ 一个不太好的代码设计

tsx 复制代码

```
const TriangleInfo = () => {
  const [triangleInfo, setTriangleInfo] = useState<{
    a: number;
    b: number;
  } | null>(null);
  const [hypotenuse, setHypotenuse] = useState<number | null>(null);
```



```

const [perimeter, setPerimeter] = useState<number | null>(null);
const [areas, setArea] = useState<number | null>(null);

useEffect(() => {
  fetchTriangle().then((t) => setTriangleInfo(t));
}, []);

useEffect(() => {
  if (!triangleInfo) {
    return;
  }

  const { a, b } = triangleInfo;
  const h = computeHypotenuse(a, b);
  setHypotenuse(h);
  const newArea = computeArea(a, b);
  setArea(newArea);
  const p = computePerimeter(a, b, h);
  setPerimeter(p);
}, [triangleInfo]);

if (!triangleInfo) {
  return null;
}

/**/ 展示信息 *****/
};

```

更好的设计

tsx 复制代码

```

const TriangleInfo = () => {
  const [triangleInfo, setTriangleInfo] = useState<{
    a: number;
    b: number;
  } | null>(null);

  useEffect(() => {
    fetchTriangle().then((r) => setTriangleInfo(r));
  }, []);

  if (!triangleInfo) {
    return;
  }

  const { a, b } = triangleInfo;
  const area = computeArea(a, b);
  const hypotenuse = computeHypotenuse(a, b);

```

```
const perimeter = computePerimeter(a, b, hypotenuse);

/** 展示信息 */
};
```

示例二

业务需求/问题描述

假设你被安排去设计一个组件：

1. 从 API 中获取一组二维坐标 (point) 列表
 2. 组件内包含一个按钮，点击后会将这些 point 按 `x` 或 `y` 值来排序 (升序)
 3. 组件内包含一个按钮，用来调整 `maxDistance` (每次增加 `10`，初始值为 `100`)
 4. 只展示距离原点 `(0, 0)` 不超过 `maxDistance` 的 point
 5. 假设这个列表只有 100 项 (不需要担心性能问题)。如果需要处理长列表，你可以使用 `useMemo` 来记忆计算结果
-

✗ 一个不太好的代码设计

tsx 复制代码

```
type SortBy = 'x' | 'y'

const toggle = (current: SortBy): SortBy => current === 'x' ? 'y' : 'x'

const Points = () => {
  const [points, setPoints] = useState<{x: number, y: number}[]>([])
  const [filteredPoints, setFilteredPoints] = useState<{x: number, y: number}[]>([])
  const [sortedPoints, setSortedPoints] = useState<{x: number, y: number}[]>([])
  const [maxDistance, setMaxDistance] = useState<number>(100)
  const [sortBy, setSortBy] = useState<SortBy>('x')

  useEffect(() => {
    fetchPoints().then(r => setPoints(r))
  }, [])

  useEffect(() => {
    const sorted = sortPoints(points, sortBy)
    setSortedPoints(sorted)
  }, [sortBy, points])
```

```

useEffect(() => {
  const filtered = sortedPoints.filter(p => getDistance(p.x, p.y) < maxDistance)
  setFilteredPoints(filtered)
}, [sortedPoints, maxDistance])

const otherSortBy = toggle(sortBy)
const pointToDisplay = filteredPoints.map(
  p => <li key={` ${p.x} | ${p.y}`} >{p.x}, {p.y}</li>
)

return (
  <>
    <button onClick={() => setSortBy(otherSortBy)}>
      排序: {otherSortBy}
    </button>
    <button onClick={() => setMaxDistance(maxDistance + 10)}>
      增加 max distance
    </button>
    仅显示距离原点(0, 0)小于 {maxDistance} 单位的点
    当前排序: '{sortBy}' (升序)
    <ol>{pointToDisplay}</ol>
  </>
)
}

```

更好的设计

tsx 复制代码

```

// NOTE: 也可以使用 useReducer 代替
type SortBy = 'x' | 'y'
const toggle = (current: SortBy): SortBy => current === 'x' ? 'y' : 'x'

const Points = () => {
  const [points, setPoints] = useState<{x: number, y: number}[]>([])
  const [maxDistance, setMaxDistance] = useState<number>(100)
  const [sortBy, setSortBy] = useState<SortBy>('x')

  useEffect(() => {
    fetchPoints().then(r => setPoints(r))
  }, [])

  const otherSortBy = toggle(sortBy)
  const filteredPoints = points.filter(p => getDistance(p.x, p.y) < maxDistance)
  const pointToDisplay = sortPoints(filteredPoints, sortBy).map(
    p => <li key={` ${p.x} | ${p.y}`} >{p.x}, {p.y}</li>
  )
}

```

```

return (
  <>
    <button onClick={() => setSortBy(otherSortBy)}>
      排序: {otherSortBy} <button>
    <button onClick={() => setMaxDistance(maxDistance + 10)}>
      增加 max distance
    <button>
      仅显示距离原点(0, 0)小于 {maxDistance} 单位的点
      当前排序: '{sortBy}' (升序)
    <ol>{pointToDisplay}</ol>
  </>
)
}

```

🍌 2.2 “传递香蕉，而不是拿着香蕉的大猩猩和整个丛林”

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle." - Joe Armstrong

"你想要的是香蕉，但你得到的是一只拿着香蕉的大猩猩和整个丛林。"

为了避免掉入这种坑，最好将基本类型（`boolean`，`string`，`number` 等）作为 props 传递。
(传递基本类型也能更好的让你使用 `React.memo` 进行优化)

组件应该仅仅只了解和它运作相关的内容就足够了。应该尽可能地与其他组件产生协作，而不需要知道它们是什么或做什么。

这样做的好处是，组件间的耦合会更松散，依赖程度会更低。低耦合更利于组件修改，替换和移除，而不会影响到其他组件。[stackoverflow:在面向对象中，低耦合和高聚合有什么区别？](#)

🐒 示例

📝 业务需求/问题描述

创建一个 `人员卡片 (MemberCard)` 组件，其中展示两个组件：`简介 (Summary)` 和 `查看更多 (SeeMore)`。

`MemberCard` 有 `id` 的 prop, 内部使用了 hook `useMember`, 该 hook 接收 `id` 并返回对应的 `Member` 信息。

ts 复制代码

```
type Member = {
  id: string;
  firstName: string;
  lastName: string;
  title: string;
  imgUrl: string;
  webUrl: string;
  age: number;
  bio: string;
  /***** 有 100 个字段 *****/
};
```

`SeeMore` 组件用于展示这个人的 年龄 (`age`) 和 简历 (`bio`)。还有一个按钮来切换这两个信息的显示/隐藏。

`Summary` 组件用于展示他的照片。同时还展示员工的 `title`, `firstName` 和 `lastName` (比如 `Mr. Vincenzo Cassano`)。点击 `member` 的名字会跳转到他的个人网站。`Summary` 组件还有一些其他功能 (比如, 每当这个组件被点击, 字体, 图片尺寸和背景色会随机变换... (简单起见, 我们叫“随机样式特性”)。

✗ 一个不太好的代码设计

tsx 复制代码

```
const Summary = ({ member } : { member: Member }) => {
  /** 包括 “随机样式特性” */
  return (
    <>
      <img src={member.imgUrl} />
      <a href={member.webUrl} >
        {member.title}. {member.firstName} {member.lastName}
      </a>
    </>
  )
}

const SeeMore = ({ member } : { member: Member }) => {
  const [seeMore, setSeeMore] = useState<boolean>(false)
  return (
    <>
```

```

        <button onClick={() => setSeeMore(!seeMore)}>
          See {seeMore ? "less" : "more"}
        </button>
      {seeMore && <>AGE: {member.age} | BIO: {member.bio}</>}
    </>
  )
}

const MemberCard = ({ id }: { id: string }) => {
  const member = useMember(id)
  return <><Summary member={member} /><SeeMore member={member} /></>
}

```

✅ 更好的设计

tsx 复制代码

```

const Summary = ({
  imgUrl,
  webUrl,
  header,
}: {
  imgUrl: string;
  webUrl: string;
  header: string;
}) => {
  /** 包括 "随机样式特性" */
  return (
    <>
      <img src={imgUrl} />
      <a href={webUrl}>{header}</a>
    </>
  );
};

const SeeMore = ({ componentToShow }: { componentToShow: ReactNode }) => {
  const [seeMore, setSeeMore] = useState<boolean>(false);
  return (
    <>
      <button onClick={() => setSeeMore(!seeMore)}>
        See {seeMore ? 'less' : 'more'}
      </button>
      {seeMore && <>{componentToShow}</>}
    </>
  );
};


const MemberCard = ({ id }: { id: string }) => {

```

```

const { title, firstName, lastName, webUrl, imgUrl, age, bio } =
  useMember(id);
const header = `${title}. ${firstName} ${lastName}`;
return (
  <>
    <Summary {...{ imgUrl, webUrl, header }} />
    <SeeMore
      componentToShow={
        <>
          AGE: {age} | BIO: {bio}
        </>
      }
    />
  </>
);
};

```

注意在  更好的设计中，`SeeMore` 和 `Summary` 组件不仅可以被 `Member` 组件使用，也可以被其他任何需要这些能力的组件复用，比如 `CurrentUser`，`Pet`，`Post` 等。

💖 2.3 让你的组件小而简单

什么是「单一职责原则」？

一个组件应该有且只有一个职责。应该尽可能的简单且实用，只有完成其职责的责任。

具有各种职责的组件很难被复用。几乎不可能只复用它的部分能力，很容易与其他代码耦合在一起。那些抽离了逻辑的组件，改起来负担不大而且复用性更强。

如何判断一个组件是否符合单一职责？

可以试着用一句话来描述这个组件。如果它只负责一个职责，描述起来会很简单。如果描述中出现了“和”或“或”，那么这个组件很大概率不是单一职责的。

检查组件的 `state`，`props` 和 `hooks`，以及组件内部声明的变量和方法（不应该太多）。问问自己：是否这些内容必须组合到一起这个组件才能工作？如果有些不需要，可以考虑把它们抽离到其他地方，或者把这个大组件拆解成小组件。

示例

需求是要展示一些不同的按钮，用户可以单击这些按钮来购买特定分类的商品。比如包，椅子和食物。

- 每个按钮会打开一个弹窗，可以用来选择和“保存 (save)”商品
 - 如果用户已经保存了某个分类的商品，那么该分类已经被“预定 (booked)”
 - 如果分类下已经有预定，按钮会有一个复选标记
 - 允许用户编辑 (添加/删除) 已经预定的商品
 - 当 hover 到按钮上，会展示 **WavingHand** (挥手) 组件
 - 当一个分类下没有任何商品可购买，按钮要展示 disabled 态
 - 当 hover 到 disabled 态的按钮上，会展示 **Not available** (不可用) 的 tooltip
 - 如果一个分类下没有任何商品可购买，按钮背景色是 **grey**
 - 如果一个分类下已经有预定，按钮背景色是 **green**
 - 如果一个分类下有可以预定的商品，按钮背景色是 **red**
 - 对于每个分类，按钮的 icon 和文案都不一样
-

一个不太好的代码设计

tsx 复制代码

```
type ShopCategoryTileProps = {
  isBooked: boolean
  icon: ReactNode
  label: string
  componentInsideModal?: ReactNode
  items?: {name: string, quantity: number}[]
}

const ShopCategoryTile = ({
  icon,
  label,
  items
  componentInsideModal,
}: ShopCategoryTileProps) => {
  const [openDialog, setOpenDialog] = useState(false)
  const [hover, setHover] = useState(false)
  const disabled = !items || items.length === 0
  return (
    <>
      <Tooltip title="Not Available" show={disabled}>
```



```

        <StyledButton
          className={disabled ? "grey" : isBooked ? "green" : "red" }
          disabled={disabled}
          onClick={() => disabled ? null : setOpenDialog(true) }
          onMouseEnter={() => disabled ? null : setHover(true)}
          onMouseLeave={() => disabled ? null : setHover(false)}
        >
          {icon}
          <StyledLabel>{label}</StyledLabel>
          {!disabled && isBooked && <FaCheckCircle/>}
          {!disabled && hover && <WavingHand />}
        </StyledButton>
      </Tooltip>
      {componentInsideModal &&
        <Dialog open={openDialog} onClose={() => setOpenDialog(false)}>
          {componentInsideModal}
        </Dialog>
      }
    </>
  )
}

```

更好的设计

tsx 复制代码

// 拆分成两个更小的组件！

```

const DisabledShopCategoryTile = ({ icon, label }: { icon: ReactNode, label: string }) => {
  return (
    <Tooltip title="Not available">
      <StyledButton disabled={true} className="grey">
        {icon}
        <StyledLabel>{label}</StyledLabel>
      </Button>
    </Tooltip>
  )
}

```

```

type ShopCategoryTileProps = {
  icon: ReactNode
  label: string
  isBooked: boolean
  componentInsideModal: ReactNode
}

```

```

const ShopCategoryTile = ({
  icon,
  label,

```

```

    isBooked,
    componentInsideModal,
  }: ShopCategoryTileProps ) => {
    const [openDialog, setOpenDialog] = useState(false)
    const [hover, setHover] = useState(false)

    return (
      <>
        <StyledButton
          disabled={false}
          className={isBooked ? "green" : "red"}
          onClick={() => setOpenDialog(true)}
          onMouseEnter={() => setHover(true)}
          onMouseLeave={() => setHover(false)}
        >
          {icon}
          <StyledLabel>{label}</StyledLabel>
          {isBooked && <FaCheckCircle/>}
          {hover && <WavingHand />}
        </StyledButton>
        {openDialog &&
          <Dialog onClose={() => setOpenDialog(false)}>
            {componentInsideModal}
          </Dialog>
        }
      </>
    )
  }
}

```

上面的例子是我在线上代码中看到的一个简化组件的真实案例

✗ 一个不太好的代码设计

tsx 复制代码

```

const ShopCategoryTile = ({
  item,
  offers,
}: {
  item: ItemMap;
  offers?: Offer;
}) => {
  const dispatch = useDispatch();
  const location = useLocation();
  const history = useHistory();
  const { items } = useContext(OrderingFormContext);
  const [openDialog, setOpenDialog] = useState(false);
  const [hover, setHover] = useState(false);
  const isBooked =

```

```

    !item.disabled && !items?.some((a: Item) => a.itemGroup === item.group);
const isDisabled = item.disabled || !offers;
const RenderComponent = item.component;

useEffect(() => {
    if (openDialog && !location.pathname.includes('item')) {
        setOpenDialog(false);
    }
}, [location.pathname]);
const handleClose = useCallback(() => {
    setOpenDialog(false);
    history.goBack();
}, []);

return (
    <GridStyled
        xs={6}
        sm={3}
        md={2}
        item
        booked={isBooked}
        disabled={isDisabled}
    >
        <Tooltip
            title="Not available"
            placement="top"
            disableFocusListener={!isDisabled}
            disableHoverListener={!isDisabled}
            disableTouchListener={!isDisabled}
        >
            <PaperStyled
                disabled={isDisabled}
                elevation={isDisabled ? 0 : hover ? 6 : 2}
            >
                <Wrapper
                    onClick={() => {
                        if (isDisabled) {
                            return;
                        }
                        dispatch(push(ORDER__PATH));
                        setOpenDialog(true);
                    }}
                    disabled={isDisabled}
                    onMouseEnter={() => !isDisabled && setHover(true)}
                    onMouseLeave={() => !isDisabled && setHover(false)}
                >
                    {item.icon}
                    <Typography variant="button">{item.label}</Typography>
                    <CheckIconWrapper>

```

```

        {isBooked && <FaCheckCircle size="26" />}
      </CheckIconWrapper>
    </Wrapper>
  </PaperStyled>
</Tooltip>
<Dialog fullscreen open={openDialog} onClose={handleClose}>
  {RenderComponent && (
    <RenderComponent item={item} offer={offers} onClose={handleClose} />
  )}
</Dialog>
</GridStyled>
);
};

```

💖 2.4 复制比错误的抽象要“便宜”的多

避免过早/不恰当的设计。如果你实现一个简单功能需要巨大的成本，可以尝试下其他的方案。我强烈推荐你阅读 [Sandi Metz: The Wrong Abstraction \(错误的抽象\)](#) .

A particular type of complexity is over-engineering, where developers have made the code more generic than it needs to be, or added functionality that isn't presently needed by the system. Encourage developers to solve the problem they know needs to be solved now, not the problem that the developer speculates might need to be solved in the future. The future problem should be solved once it arrives and you can see its actual shape and requirements in the physical universe. - Google Engineering Practices: What to look for in a code review

过度设计是一种特殊的复杂度，开发者会让代码设计的比它需要的更加通用，或者添加目前根本不需要的能力。应该鼓励开发者解决他们目前需要处理的问题而不是将来可能遇到的问题。一旦未来出现新的问题，应该马上处理，到时候你可以站在全局的角度再来看看这个问题实际是什么样的。—— [谷歌工程实践：CR 应该做什么](#)

参考: [KCD: AHA Programming \(AHA 编程\)](#) , [C2 Wiki: Contrived Interfaces/The Expensive Setup Smell/Premature Generalization](#)

👤 3. 性能优化技巧

"Premature optimization is the root of all evil" - Tony Hoare

"过早的优化是万恶之源"

"One accurate measurement is worth a thousand expert opinions" - Grace Hopper

"一次准确的测试胜过一千个专家的意见"

1. 如果你觉得应用速度慢，就应该做一次基准测试 (benchmark) 来证明。 "面对模棱两可的情况，拒绝猜测。" 多使用[Chrome 插件 - React 开发者工具](#)的 profiler!
2. `useMemo` 主要用在大开销的计算上。
3. 如果你打算使用 `React.memo` , `useMemo` , 和 `useCallback` 来减少重新渲染，它们不该有过多的依赖项，且这些依赖项最好都是基本类型。
4. 确保你清楚代码里 `React.memo` , `useCallback` 或 `useMemo` 它们都是为了什么而使用的 (是否真的能防止重新渲染？是否能证明在这些场景中真的可以显著提高性能？[Memoization 有时会起到反作用](#)，所以需要关注！)
5. [优先修复慢渲染，再修复重新渲染](#)
6. 把状态尽可能地放在它被使用的地方，一方面让代码读起来更顺，另一方面，能让你的 app 更快(state colocation (状态托管))
7. `Context` 应该按逻辑分开，不要在一个 provider 中管理多个 value。如果其中某个值变化了，所有使用该 context 的组件 (即便没有用到这个值) ，都会重新渲染。
8. 可以通过拆分 `state` 和 `dispatch` 来优化 `context` 。
9. 了解下 [lazy loading \(懒加载\)](#) 和 [bundle/code splitting \(代码分割\)](#)
10. 长列表请使用 [tannerlinsley/react-virtual](#) 或其它类似的库。
11. 包体积越小，app 越快。你可以使用 [source-map-explorer](#) 或者 [@next/bundle-analyzer](#) (用于 NextJS) 来进行包体积分析。
12. 关于表单的库，推荐使用 [react-hook-forms](#) , 它在性能和开发体验各方面都做的比较好。

推荐阅读: KCD 有关性能的文章

- [KCD: State Colocation will make your React app faster \(状态托管会使 React 更快\)](#)
- [KCD: When to `useMemo` and `useCallback` \(什么时候使用 `useMemo` 和 `useCallback` \)](#)
- [KCD: Fix the slow render before you fix the re-render \(优先修复慢渲染，再修复重新渲染\)](#)
- [KCD: Profile a React App for Performance \(检测一个 React App 的性能\)](#)
- [KCD: How to optimize your context value \(如何优化 context\)](#)
- [KCD: How to use React Context effectively \(如何高效使用 context\)](#)

- [KCD: One React Mistake that is slowing you down](#) (一个影响性能的 react 错误)
- [KCD: One simple trick to optimize React re-renders](#) (一个优化 React 重新渲染的小技巧)

4. 测试原则

"Write tests. Not too many. Mostly integration." - Guillermo Rauch

我们需要测试，但不用太多（不需要追求 100% 覆盖率），精力应该侧重于集成测试（因为在速度、消耗和可靠性之间取得了一定的平衡）

1. 测试应该始终与软件的使用方式相似。
2. 确保不是在测试一些边界细节（用户不会使用，看不到甚至感知不到的内容）。
3. 如果你的测试不能让你对自己的代码产生信任，那测试就是无意义的。
4. 如果你正在重构某个代码，且最后实现的功能都是完全一致的，其实几乎不需要修改测试，而且可以通过测试结果来判定你正确的重构了。
5. 对于前端来说，不需要 100% 的测试覆盖率，70% 就足够了。测试应该提升你的开发效率，虽然维护测试会暂时地阻塞你目前的开发，但当你不断地增加测试，会在不同阶段得到不同的回报。
6. 我个人喜欢使用 [Jest](#), [React testing library](#), [Cypress](#), 和 [Mock service worker](#)。