实现 React 应用与钱包插件、链上合约交互

最近在开发一个 NFT 二创平台，其中包含了很多概念和技术。我会更新一个系列的文章来总结和沉淀在这个过程中的一些知识与思考。

本文是对 ethers.js 进行一个全方位介绍，非常适合 web3 入门学习。

什么是 ethers.js?

Web3 中的各类 DApp，都需要与智能合约进行交互。如果用原生 JS 来做这些事会很麻烦。这时就需要使用专属的 SDK。

目前 JS 环境中有两个主流的库可以用来和智能合约进行交互，一个是 web3.js，另一个是 ethers.js。

ethers.js VS web3.js

web3.js 比 ethers.js 出现的更早。但是目前 ethers.js 更受欢迎。

主要原因有如下两点：

1. 体积：ethers.js 体积仅有 116.5kb，web3.js 有 590.6kb。相差 5 倍左右。
2. 设计理念：由于设计理念不同。web3.js 认为用户会在本地部署以太坊节点，私钥和网络连接状态由这个节点管理。但实际上大部分人都不会在本地部署以太坊节点。ethers.js 充分考虑了这一点，它是用 Provider 管理网络连接状态，用 Wallet 管理密钥，更加安全和灵活。而且原生支持 ENS。

ethers.js 基本使用介绍

节点即服务

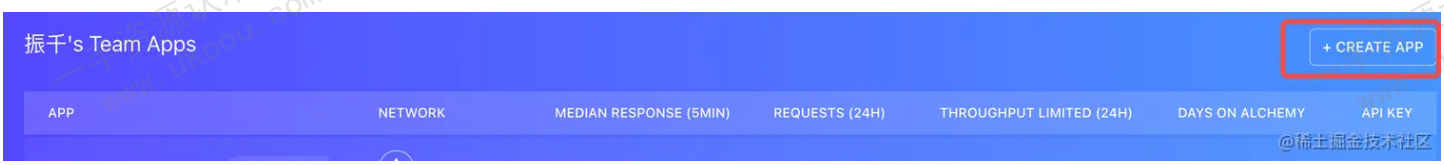
由于在本地部署一个区块链节点的成本并不低，很少会有人真的部署一个节点，而是选择使用节点即服务。

这类服务有很多，比如老牌的 Alchemy、Infura、Moralis 以及今年估值 102 亿美金的新秀 Tenderly。

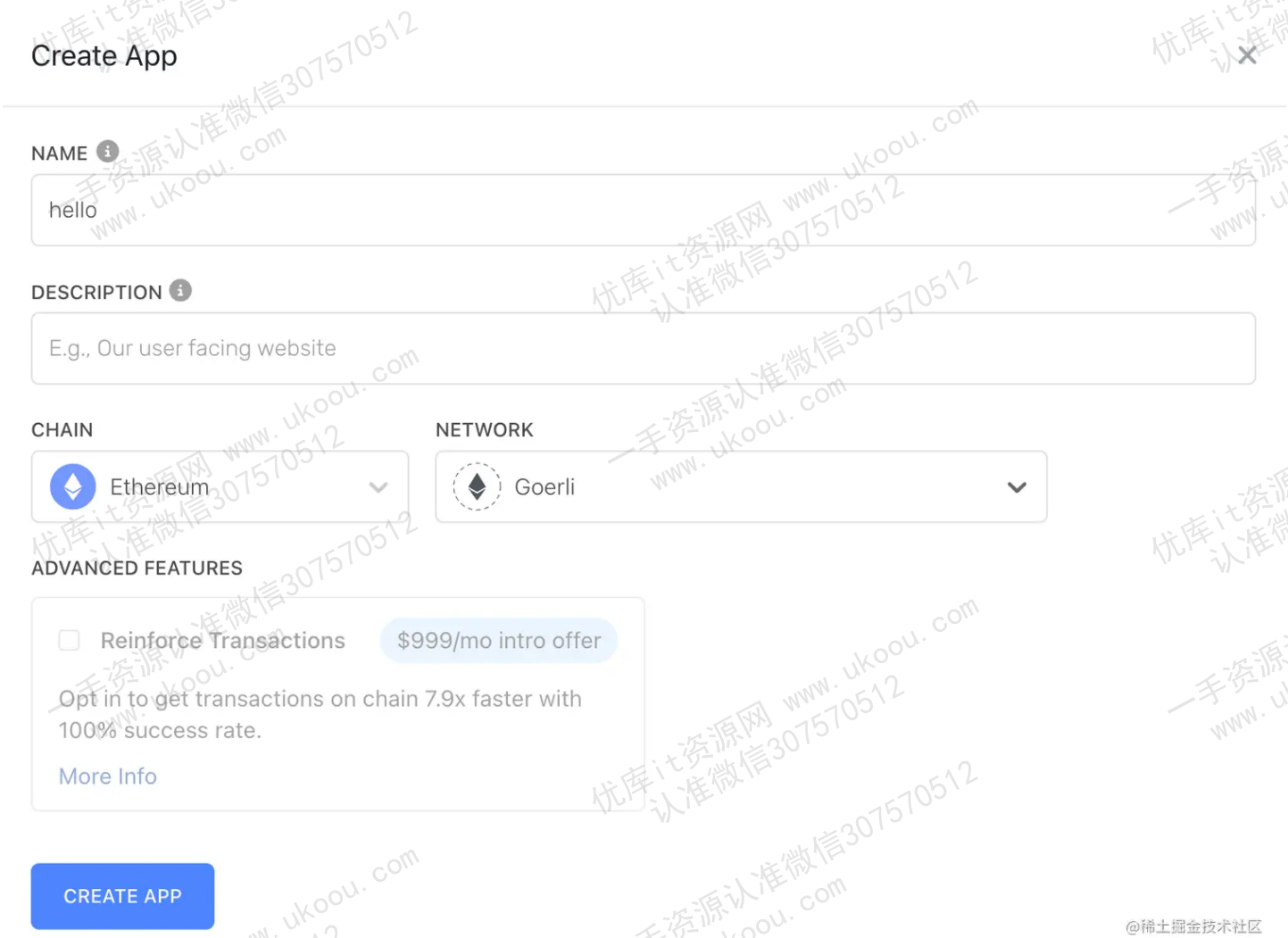
在这里我们选择 Alchemy，目前它的市场占有率是最高的。

alchemy 的网址在这：dashboard.alchemy.com/。具体的注册登陆就不讲了。

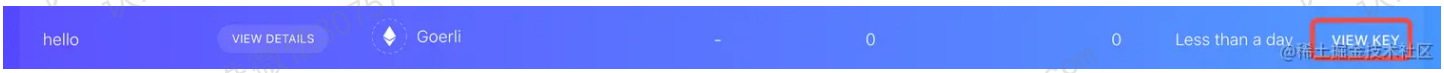
登陆之后我们创建一个 App。



Chain 选择 Ethereum，Network 选择 Goerli。



这样就成功创建了一个 App，点击后面的 view key，就可以查看 key。



我们把它复制下来，后面会用到。

构造合约

在对合约进行读取或交互之前，我们首先需要构造一个合约对象。

合约对象有三个构造参数，第一个是合约地址，是一个字符串。第二个是合约的 abi，可以是一个 JSON，第三个参数是 provider 对象，它用于管理网络连接状态。

下面的例子中就是使用 alchemy 提供的 JSON RPC 接口作为网络连接。

```
ini
复制代码
const rpc = `https://eth-goerli.g.alchemy.com/v2/${process.env.NEXT_PUBLIC_ALCHEMY_API_KEY}`;const provider = new ethers.providers.JsonRpcProvider(rpc);const contract = new ethers.Contract(contractAddress, abi, provider)
```

除了 JsonRpcProvider 以外，ethers 还有 IpcProvider、InfuraProvider、AlchemyProvider、Web3Provider 等多种 Provider。

读取合约信息

abi 中的方法会直接挂载到 contract 对象上，我们可以直接调用。

不过需要注意，所有的操作都是异步的。

```
ini
复制代码
(async () => {  const owner = await contract.owner();  console.log(owner);})();
```

连接钱包

由于和合约交互需要支付 gas 费用，所以必须有一个数字钱包。

钱包有很多种，比如 MetaMask、Rainbow、Coinbase Wallet 等。其中 MetaMask 是最常用的一种数字钱包。

这里主要介绍如何连接到 MetaMask。

MetaMask 有一个浏览器插件，如果用户安装了该插件，在 window 对象下会有一个 ethereum 对象。我们可以调用 ethereum.request 方法发起请求，参数是一个对象，对象的 method 描述该次请求的操作。

ethereum.request 方法是异步的，会返回一个数组，该数组是所有登陆钱包的账户地址字符串，第一个账户就是当前激活的账户。如果返回的数组长度为 0，则意味着没有登陆任何账户。

```
javascript
复制代码
(async ()=> {  const accounts = await ethereum.request({ method: 'eth_requestAccounts' })  if(accounts.length === 0) {    throw Error('未登录任何账户')  }    const activeAccount = accounts[0]  console.log(activeAccount)})()
```

我们还可以通过 ethereum.request 方法获取当前的网络状态。

```
1 javascript
2 复制代码
3 (async ()=> {  const chainId = await ethereum.request({ method: 'eth_chainId'
  })  console.log(chainId)})()
```

它会返回一个字符串。0x1 表示以太网主网；0x5 表示 Goerli 测试网，更多网络的 chainId 可以在这个网站查看：chainlist.org/。

下面是使用 ethers.js 来连接 MetaMask 的代码。

```
1 dart
2 复制代码
3 (async ()=> {  const provider = new
  ethers.providers.Web3Provider(window.ethereum)  const accounts = await
  provider.send("eth_requestAccounts", [])  const activeAccount = accounts[0]})()
```

如果使用 MetaMask 作为 provider，那么就不需要再使用 alchemy 了。

钱包

在转账交易之前，我们需要创建一个 Wallet 实例，它的作用是对交易和消息进行签名。

创建 Wallet 对象的方法有三种。

通过 Wallet.createRandom 创建随机钱包

这种方式创建的是一个单机钱包，需要连接网络。

```
1 ini
2 复制代码
3 const wallet = ethers.Wallet.createRandom()wallet.connect(provider)
```

通过助记词创建

```
1 arduino
2 复制代码
3 const wallet = new ethers.Wallet.fromMnemonic(mnenonic.phrase)
```

通过私钥和 provider 创建

```
1 arduino
```

```
2 复制代码
3 const wallet = new ethers.Wallet(privateKey, provider)
```

钱包信息

我们可以在创建好的钱包上面获取很多有用的信息，比如钱包地址、助记词、私钥、交易次数等。

```
1 javascript
2 复制代码
3 console.log(wallet.address)console.log(await
  wallet.getAddress())console.log(wallet.mnemonic)console.log(wallet.privateKey)c
  onsole.log(wallet.getTransactionCount())
```

转账

一旦又了钱包，我们就可以向其他人发起转账交易。

创建一个 tx 对象，它最少需要两个属性，to 和 value，分别表示接受钱包地址和转账额度。

然后使用 wallet.sendTransaction 方法发送转账，它会返回一个 receipt 对象。这个对象有一个异步的 wait 方法，当交易上链后会返回。

```
1 javascript
2 复制代码
3 const tx = {  to: address,  value:
  ethers.utils.parseEther("0.1"),}console.log('开始转账')const receipt = await
  wallet.sendTransaction(tx)await receipt.wait()console.log('完成转账')
```

通过合约转账交易

交易需要使用 Wallet 对象。再通过 wallet 作为合约的第三个构造参数创建 Contract 对象。

调用合约的 transfer 方法，进行转账交易。该方法需要两个参数，转入的钱包地址字符串和转入的数量。

transfer 会返回一个 tx 对象，该对象有一个异步的 wait 方法，会在交易完成后执行。

```
1 dart
2 复制代码
3 const wallet = new ethers.Wallet(privateKey, provider)const contract = new
  ethers.Contract(contractAddress, abi, wallet)(async ()=> {  const tx = await
  contract.transfer(toAddress, ethers.utils.parseEther("1"))  await tx.wait()})()
```

通过 signer 进行转账

通常我们无法直接拿到 privateKey，但是可以通过 signer 对象间接使用 privateKey。只需要进行签名就可以进行交易。这也是最常用的交易方式。

```ini
复制代码
const signer = walletProvider.getSigner();const tx = {  to,  value,};const receipt = await signer.sendTransaction(tx);await receipt.wait();
```

使用 React 和 ethers.js 开发加密钱包

接下来我们开发一个最简单的加密钱包，具备最基础的转账功能和查询余额功能。

创建项目

我们首先创建一个 Next.js 项目。

```lua
复制代码
npx create-next-app
```

需要选择 TypeScript。

安装依赖

安装 ethers.js

```css
复制代码
npm i ethers
```

安装 tailwindcss

```csharp
复制代码
npm install -D tailwindcss postcss autoprefixernpx tailwindcss init -p
```

修改 tailwind.cinfig.js。

```
1 css
2 复制代码
3 /** @type {import('tailwindcss').Config} */module.exports = {  content: [
"./pages/**/*.{js,ts,jsx,tsx}",    "./components/**/*.{js,ts,jsx,tsx}",  ],
  theme: {    extend: {},  },  plugins: [],}
```

修改 styles/globals.css。

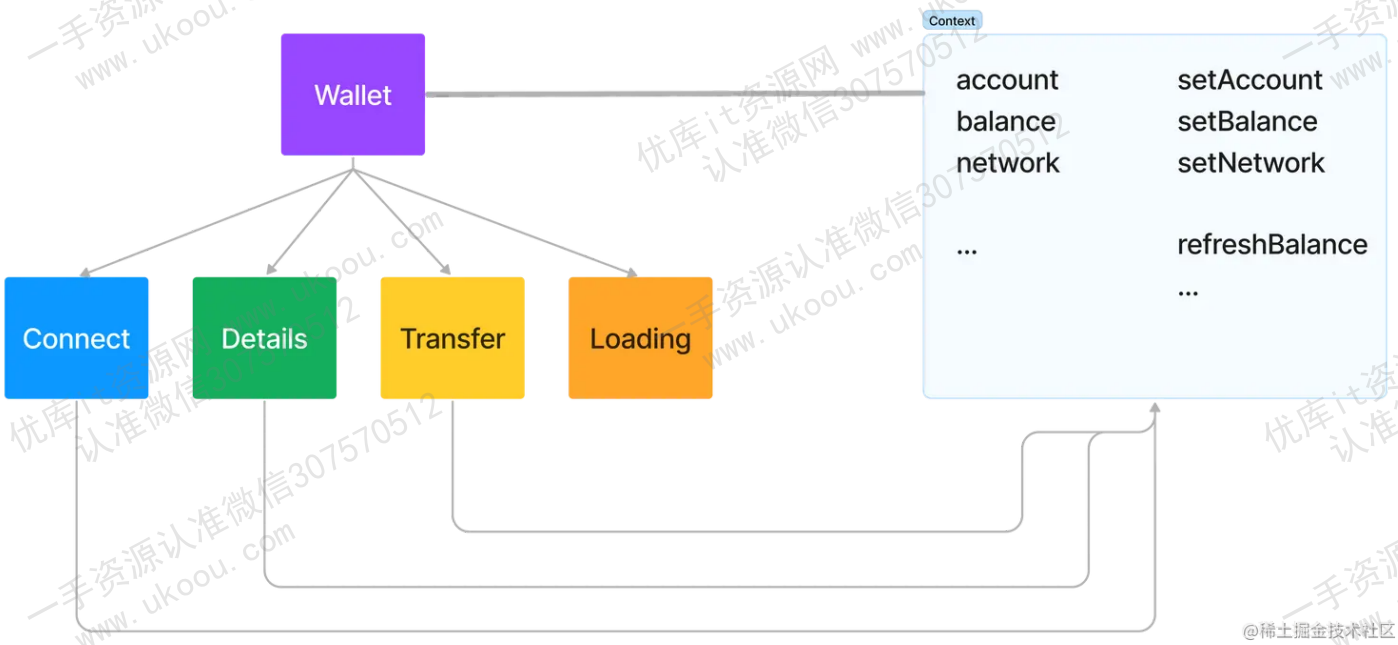
```
1 less
2 复制代码
3 @tailwind base;@tailwind components;@tailwind utilities;
```

安装 headless-ui

```
1 bash
2 复制代码
3 npm install @headlessui/react
```

整体架构设计

由于业务并不复杂，我们可以将它简单划分为几个组件。使用 Context 足够应对这个场景，而不需要额外导入状态管理库来增加复杂性。



Wallet 是根组件，内部维护了很多 state，以 Context 的方式将数据和操作注入到子组件。

Connect 负责连接钱包和断开连接。

Details 负责显示钱包信息，是纯展示型组件。

Transfer 负责向其他账户进行转账。

Loading 负责渲染加载动画，是纯展示型组件。

创建上下文

```
typescript
复制代码
type IWalletCtx = {  walletProvider: any;  setWalletProvider: (walletProvider: any) => void;  msgIsOpen: boolean;  setMsgIsOpen: (msgIsOpen: boolean) => void;  msg: string;  setMsg: (msg: string) => void;  account: string;  setAccount: (account: string) => void;  networkName: string;  setNetworkName: (networkName: string) => void;  balance: string;  setBalance: (balance: string) => void;  showMessage: (message: string) => void;  refresh: boolean;  setRefresh: (refresh: boolean) => void;};const WalletCtx = createContext<IWalletCtx>({} as IWalletCtx);
```

通过初始化一个对象，然后断言为 IWalletCtx 的方式，可以避免在使用 WalletCtx 时添加是否为 undefined 或 null 的判断。因为我们一定会注入数据。

Loading 组件

Loading 作为纯展示型组件，是最简单的组件。SVG 的代码是直接从 tailwindcss 文档中搬运过来的。仅仅是添加了一个 size 属性，用来展示不同大小的尺寸。

```
ini
复制代码
function Loading({ size = "md" }: { size?: "sm" | "md" | "lg" | "xl" }) {  const sizes = {    sm: "h-3 w-3",    md: "h-5 w-5",    lg: "h-7 w-7",    xl: "h-9 w-9",  };  return (    <svg      className={`animate-spin -ml-1 mr-3 ${sizes[size]} text-black`}      xmlns="http://www.w3.org/2000/svg"      fill="none"      viewBox="0 0 24 24"    >      <circle        className="opacity-25"        cx="12"        cy="12"        r="10"        stroke="currentColor"        strokeWidth="4"      ></circle>      <path        className="opacity-75"        fill="currentColor"        d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014 12H0c0 3.042 1.135 5.824 3 7.938l3-2.647z"      ></path>    </svg>  );}
```

Wallet 组件

在 Wallet 组件中创建这些 state，并注入到 context 中。

```
typescript
复制代码
export default function Wallet() {  const [walletProvider, setWalletProvider]
= useState<any(null);  const [msgIsOpen, setMsgIsOpen] = useState(false);
const [msg, setMsg] = useState("");  const [account, setAccount] =
useState<string("");  const [networkName, setNetworkName] =
useState<string("");  const [balance, setBalance] = useState<string("");  const
 [refresh, setRefresh] = useState<boolean(false);  useEffect(() => {
setWalletProvider(new ethers.providers.Web3Provider(window.ethereum));  },
[]);  const showMessage = (message: string) => {    setMsg(message);
setMsgIsOpen(true);    setTimeout(() => {      setMsg("");
setMsgIsOpen(false);    }, 2000);  };  return (    <WalletCtx.Provider
value={{      walletProvider,        setWalletProvider,        msgIsOpen,
    setMsgIsOpen,        msg,        setMsg,        account,
setAccount,        networkName,        setNetworkName,        balance,
setBalance,        showMessage,        refresh,        setRefresh,      }}
>      <div className="flex flex-col gap-4 p-4"        <Dialog open=
{msgIsOpen} as={"div"} onClose={() => setMsgIsOpen(false)}>          <div
className="fixed flex justify-center items-center w-full top-2"
<Dialog.Panel className="inline-flex flex-col bg-green-400 text-slate-600 p-4
shadow-xl rounded-3xl"              <Dialog.Title{msg}</Dialog.Title
 </Dialog.Panel          </div        </Dialog        <Connect />
<Details />        <Transfer />      </div    </WalletCtx.Provider  );}
```

Wallet 组件基本上没有什么逻辑，它的主要作用有三个：

- 向 context 注入数据。
- 创建 ethers.provider。
- 使用 Dialog 组件作为全局消息提示。

Connect 组件

在 styles/globals.css 中添加按钮样式。

```
less
复制代码
@layer components {  .btn {    @apply bg-black text-white py-2 px-4 rounded-
3xl;  }}
```

在 index.tsx 中编写逻辑。

```
ini
复制代码
function Connect() {  const {    walletProvider,    account,    setAccount,
setNetworkName,    setBalance,    showMessage,    refresh,  } =
useContext(WalletCtx);  const refreshBalance = useCallback(async () => {    if
(!walletProvider || !account) return;    const balance = await
walletProvider.getBalance(account);
setBalance(ethers.utils.formatEther(balance));  }, [setBalance,
walletProvider, account]);  useEffect(() => {    refreshBalance();  },
[refresh, refreshBalance]);  const connectToMetamask = async () => {    try {
    await window.ethereum.enable();    const accounts = await
walletProvider.send("eth_requestAccounts", []);    const network = await
walletProvider.getNetwork();    const balance = await
walletProvider.getBalance(accounts[0]);    setAccount(accounts[0]);
setNetworkName(network.name);
setBalance(ethers.utils.formatEther(balance));    } catch (error) {
console.log(error);    showMessage("failed to connect to metamask");    }
};  const disconnect = async () => {    setAccount("");  };  if (!account) {
 return (    <div className="flex justify-end p-4"    {walletProvider ?
(        <button className="btn" onClick={connectToMetamask}>
connect to metamask      </button>      ) : (        <Loading />
 )}    </div>  );  }  return (    <div className="flex justify-end items-
center gap-2"    <h1 className="text-end"Hello, {account}</h1>    <button
className="btn" onClick={disconnect}>    disconnect    </button>
</div>  );}
```

我们连接钱包后会获取 3 个重要的信息：钱包账户地址、连接的网络和余额。

分别通过 walletProvider.listAccounts()、walletProvider.getNetwork() 和 walletProvider.getBalance(accounts[0]) 来获取，需要注意它们都是异步操作。

Details 组件

Details 作为纯展示型组件没有什么逻辑，主要是一些样式。

```
javascript
复制代码
function Details() {  const { account, networkName, balance } =
useContext(WalletCtx);  if (!account) {    return null;  }  return (    <div
className="flex flex-col gap-4 w-full bg-slate-800 text-white p-4 rounded-md"
    <div className="flex justify-between"      <div className="text-2xl font-
thin"balance</div      <divnetwork: {networkName}</div    </div    <div
```

```
className="flex items-end gap-2"        <div className="text-2xl"{balance}
</div        <divETH</div      </div     </div  );}
```

现在我们看一下 Connect 和 Details 组件一起使用的效果。

Transfer 组件

Transfer 的功能比较简单，它在 UI 上仅仅包含两个输入框和一个 send 按钮。

两个输入框分别可以输入 to 和 value，表示转账的钱包地址和转账金额。

```ini
复制代码
function Transfer() {  const { walletProvider, account, showMessage, refresh, setRefresh } =    useContext(WalletCtx);  const [to, setTo] = useState<string>("");  const [amount, setAmount] = useState<string>("");  const [transferring, setTransferring] = useState<boolean>(false);  const transfer = async () => {  try {      const value = ethers.utils.parseEther(amount);      const signer = walletProvider.getSigner();      const tx = {        to,        value,      };      setTransferring(true);      const receipt = await signer.sendTransaction(tx);      await receipt.wait();      setTo("");      setAmount("");      showMessage("successfully transferred");    } catch (error) {      console.log(error);      showMessage("failed to transfer");    } finally {      setTransferring(false);      setRefresh(!refresh);    }  };  if (!account) {    return null;  }  return (    <div className="flex flex-col gap-4 mt-4"      <div className="font-bold text-4xl"Transfer</div>      {transferring ? (        <div className="flex flex-col items-center gap-4"          <div className="text-3xl"transferring...</div>          <Loading
```

```
size="xl" />      </div>      ) : (      <div className="flex flex-col gap-
2">      <input      className="input"      value={to}
onInput={(e: any) => setTo(e.target.value)}      type="text"
placeholder="address"      />      <input
className="input"      value={amount}      onInput={(e: any) =>
setAmount(e.target.value)}      type="number"
placeholder="amount"      />      <button className="btn" onClick=
{transfer}>      send      </button>      </div>      )}    </div>
);}
```

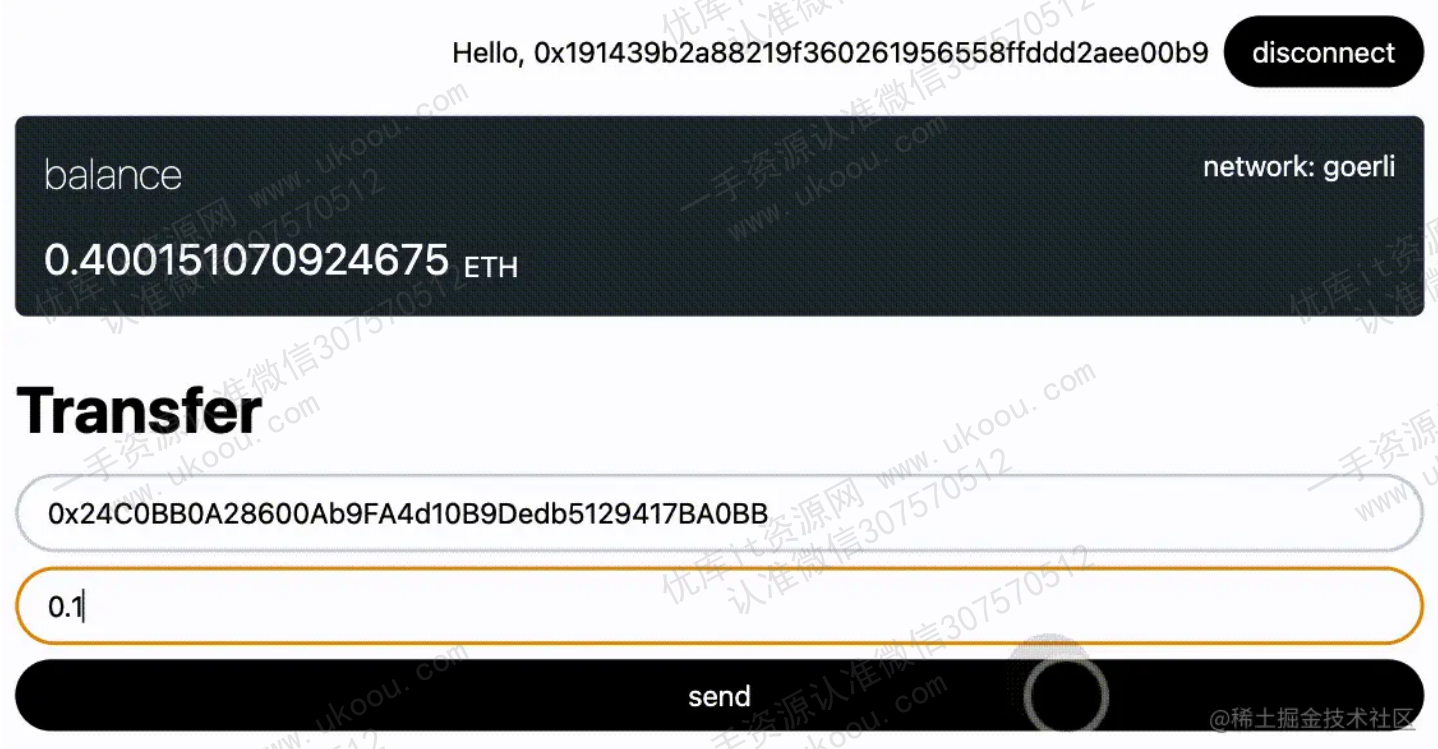
转账是通过 signer.sendTransaction 方法进行的，它会返回收据对象 receipt。

在转账时使用到了 ethers.utils.parseEther，因为 value 默认的单位是 wei，它非常小，10 的 18 次方才是一个 ehter。在 JS 中需要使用 BigInt 类型表示，并不方便操作，而我们更喜欢用 ether 来描述货币。所以这个 API 可以帮我们转换货币单位。

需要注意，在测试时需要选择 goerli 网或者其他测试网，否则会浪费 gas 费。

你至少要有两个钱包账户，这样可以从一个钱包账户转到另一个钱包账户。

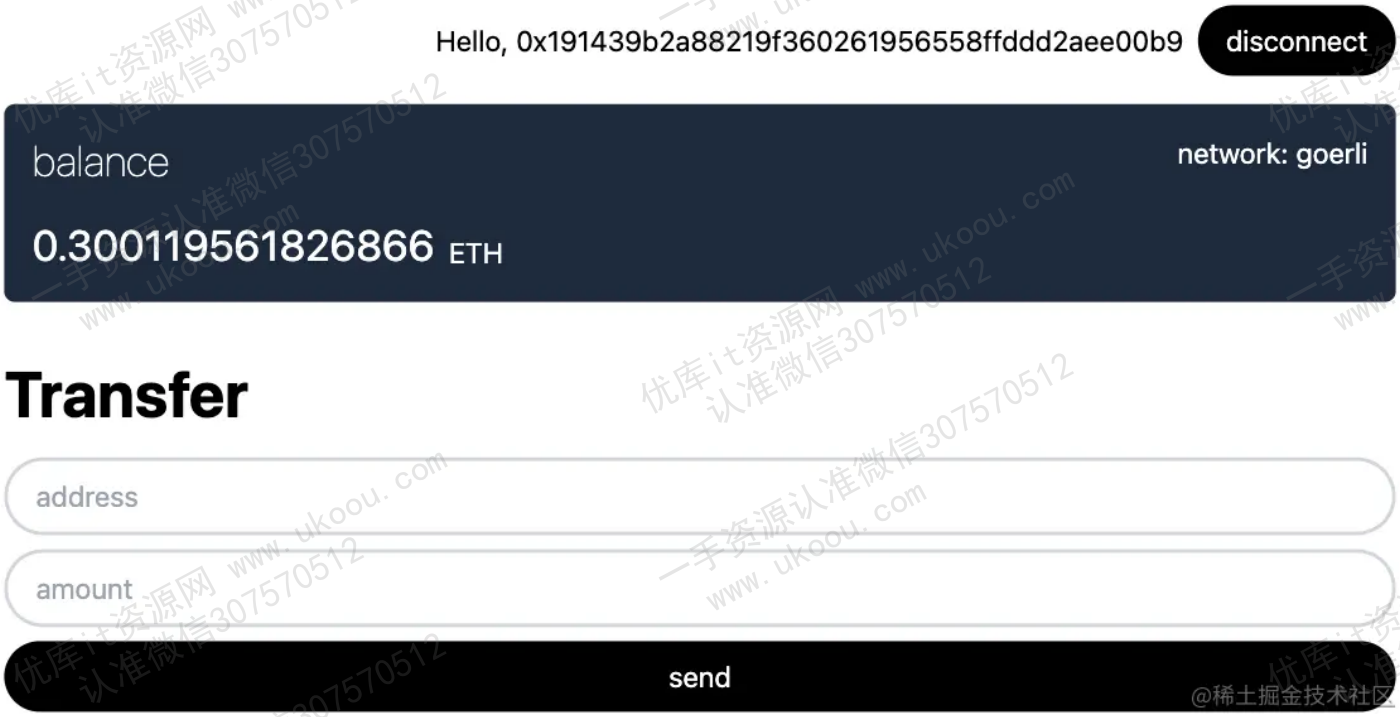
下面我们来测试一下转账。



这时 MetaMask 钱包会弹出来签名界面。

点击确认后，需要等待一段时间上链，大约 1 分钟，或者更久。

转账成功后，当前账户的余额会刷新。



完成

现在一个简单的加密钱包就完成了。通过这个项目的学习，相信你已经学会了 ethers.js 常规 API 的使用。

源码链接：