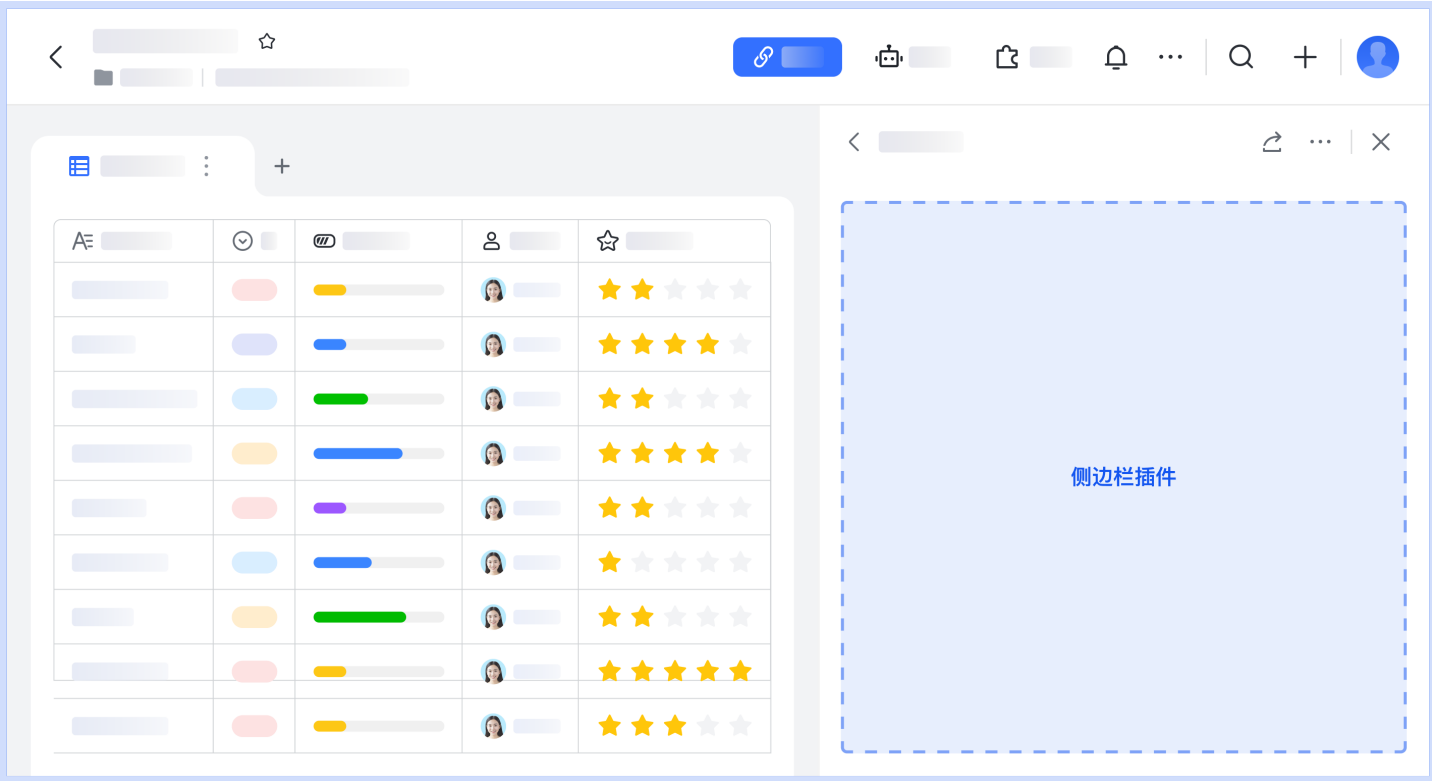


边栏插件开发指南


介绍

边栏插件是多维表格推出的一个灵活、便捷的开放能力。开发者可通过编程来实现自定义功能，扩展核心平台能力，构建更强大的业务系统，或将其发布到插件中心以供所有多维表格用户使用。



寻求帮助

如果在开发过程中遇到任何困难，或有任何反馈，请加入交流群，发起话题，与运营人员及其他开发者一起进行讨论。

 **多维表格插件交流群** 外部

群名片

如果你有特殊的需求又没有开发资源，可以向其他开发者 [提交插件需求](#)，开发者也可以 [查看需求汇总](#) 来认领需求。

开始开发

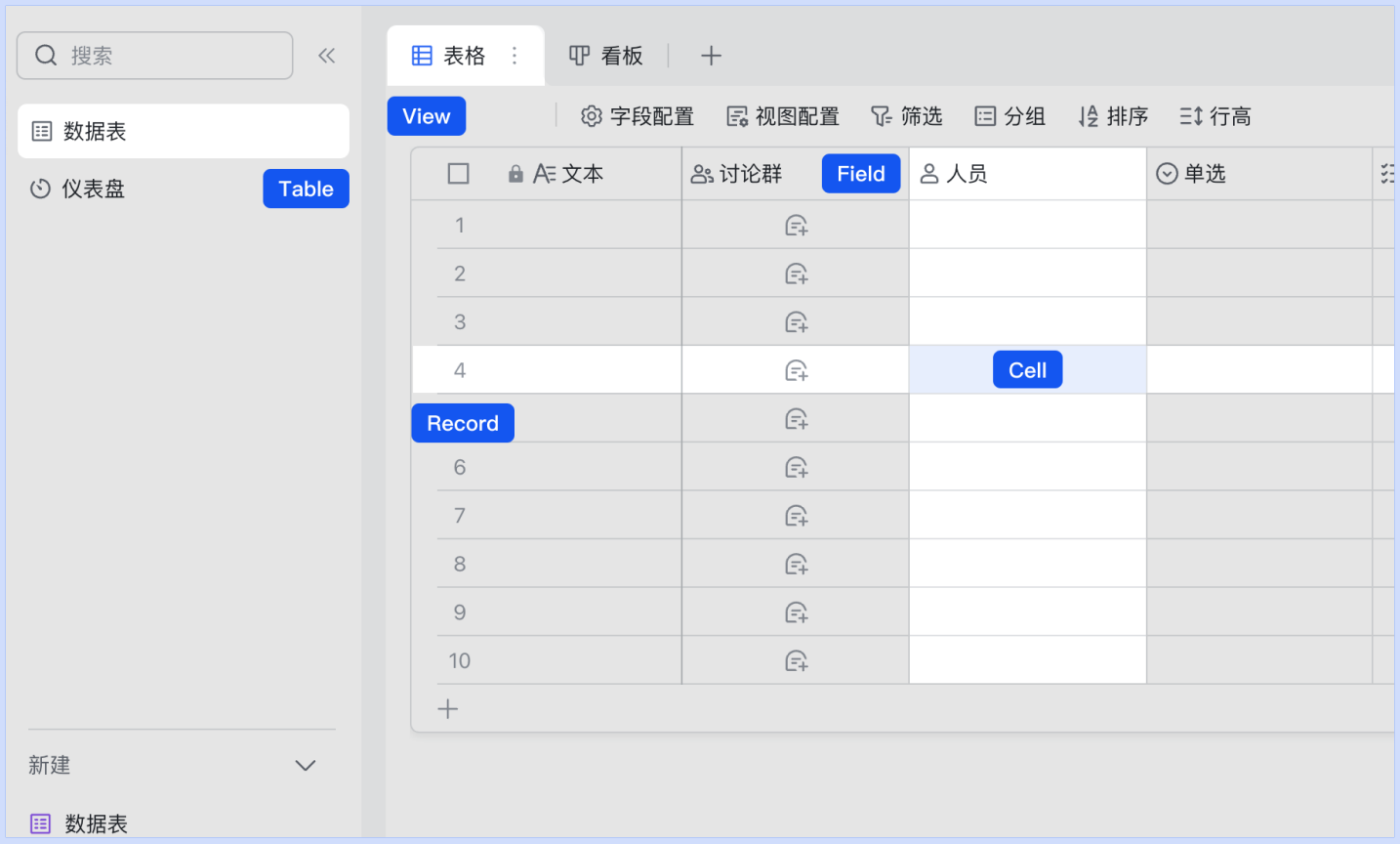
跟随示例，尝试动手完成一个插件的搭建，对插件的开发流程建立直观认知。无论是 Vercel、Github、localhost，还是你自己的服务器，只要部署了服务，插件都可以在多维表格中正常运行。你可以直接在多维表格的控制台中查看调试信息。

- 新建或打开任意多维表格，点击 插件 展开插件面板
- 点击 自定义插件 ，点击 +新增插件 ，在输入框内填入运行地址后点击 确定 添加并运行插件



多维表格数据模型

我们先了解一下多维表格的核心概念以及相关知识，多维表格的数据结构与常见的关系型数据库基本概念相通。下图所示的是 API 的设计模型：



AI 编程实践：多维表格插件

写了一份「AI 友好」的多维表格插件教程，主要提供了 Markdown 格式的资源，包括 API、开发指南、设计规范等等，方便 AI 读取和参考。

一个简单的从 0 编写字段捷径教程，作为演示。

实现一个前端插件

以 [Base JS SDK](#) 为例，演示如何开发一个前端插件。

准备开发环境

安装 SDK，或选择一个[模板](#)项目 GitHub 地址，Fork 此仓库并 clone 到本地，再按照 Readme.md 中说明运行项目

npm

```
1 npm i -S @lark-base-open/node-sdk
```

yarn

```
1 yarn add @lark-base-open/node-sdk
```

实现逻辑

在准备好开发环境的基础上，我们来开发一个货币转换插件，首先需要用户先插入一个货币字段，并填充一定的数据。

1. 安装完成之后，在 src 目录下新建 ts 文件取名为 `exchange-api.ts` 并复制以下内容。

```
1 import axios from 'axios';
2
3 interface ExchangeRatesResponse {
4   rates: {
5     [key: string]: number;
6   };
7   base: string;
8   date: string;
9 }
10
11 export async function getExchangeRate(base: string, target: string):
  Promise<number | undefined> {
12   try {
13     const response = await axios.get<ExchangeRatesResponse>
      (`https://api.exchangerate-api.com/v4/latest/${base}`);
14     const rate = response.data.rates[target];
15
16     if (!rate) {
17       throw new Error(`Exchange rate not found for target currency:
        ${target}`);
18     }
19
20     return rate;
21   } catch (error) {
22     console.info(`Error fetching exchange rate: ${(error as
      any).message}`);
23   }
24 }
```

```
23     }
24 }
```

这部分的代码逻辑是获取实时汇率，`base` 指的是当前的货币类型 `target` 指的是兑换的货币类型，通过这个 API 可以获取保留两位小数的汇率。

2. 在 `src` 目录下新建一个 `ts` 文件，取名为 `const.ts`，并将以下内容复制进去。

```
1  import { CurrencyCode } from '@lark-base-open/js-sdk';
2
3  export const CURRENCY = [
4    { label: 'CNY', value: CurrencyCode.CNY },
5    { label: 'USD', value: CurrencyCode.USD },
6    { label: 'EUR', value: CurrencyCode.EUR },
7    { label: 'AED', value: CurrencyCode.AED },
8    { label: 'BRL', value: CurrencyCode.BRL },
9    { label: 'CAD', value: CurrencyCode.CAD },
10   { label: 'CHF', value: CurrencyCode.CHF },
11   { label: 'HKD', value: CurrencyCode.HKD },
12   { label: 'INR', value: CurrencyCode.INR },
13   { label: 'JPY', value: CurrencyCode.JPY },
14   { label: 'MXN', value: CurrencyCode.MXN },
15 ];
```

这个文件是用来枚举可以进行转换的货币类型，因为只做 Demo 展示，所以枚举的数量有限。

3. 提供用户选择转换的货币字段能力。

首先，货币转换是在原本的字段进行货币值的转换，所以我们需要筛选当前 `table` 中的货币类型字段，来让用户进行选择，这里我们在交互上使用 `Select` 组件来实现选择这个动作，其中每一个选项都是当前 `table` 可以选择的货币字段。

我们修改 `index.tsx` 中的 `LoadApp` 函数：

定义货币字段信息的 `currencyFieldMetaList` 以及选择进行转换的字段 `selectFieldId` 和选择转换的货币类型 `currency`。

```
1  import { bitable, CurrencyCode, FieldType, ICurrencyField,
2    ICurrencyFieldMeta } from '@lark-base-open/js-sdk';
3
4  import { CURRENCY } from './const';
5
6  function LoadApp() {
```

```

5     const [currencyFieldMetaList, setMetaList] =
      useState<ICurrencyFieldMeta[]>([])
6     const [selectFieldId, setSelectFieldId] = useState<string>();
7     const [currency, setCurrency] = useState<CurrencyCode>();

```

修改 `useEffect` 函数，在页面完成渲染时获取当前 `table` 内的货币类型字段信息。

```

1  useEffect(() => {
2    const fn = async () => {
3      const table = await bitable.base.getActiveTable();
4      const fieldMetaList = await
        table.getFieldMetaListByType<ICurrencyFieldMeta>(FieldType.Currency);
5      setMetaList(fieldMetaList);
6    };
7    fn();
8  }, []);

```

按照顺序，我这里讲解一下用到的相关 API：

- `bitable.base.getActiveTable`：获取当前的 `table`，获取到了 `table` 之后就可以对数据进行操作
- `table.getFieldMetaListByType<ICurrencyFieldMeta>(FieldType.Currency)`：通过字段类型去获取对应的字段信息

然后我们修改渲染的组件，满足用户交互上的需求。

```

1  const formatFieldMetaList = (metaList: ICurrencyFieldMeta[]) => {
2    return metaList.map(meta => ({ label: meta.name, value: meta.id }));
3  };
4
5  return <div>
6    <div style={{ margin: 10 }}>
7      <div>Select Field</div>
8      <Select style={{ width: 120 }} onSelect={setSelectFieldId} options=
        {formatFieldMetaList(currencyFieldMetaList)}/>
9    </div>
10   <div style={{ margin: 10 }}>
11     <div>Select Currency</div>
12     <Select options={CURRENCY} style={{ width: 120 }} onSelect=
        {setCurrency}/>
13   </div>
14 </div>

```

这个时候，用户已经可以选择字段和想要转换的货币类型了，我们接下来实现转换货币值的逻辑。

4. 实现货币转换的逻辑，我们先将获取汇率的 API 引入。

```
1 import { CURRENCY } from './const';
2 import { getExchangeRate } from './exchange-api';
```

然后准备一个转换交互按钮以及转换函数。

```
1 const transform = async () => {
2   }
3
4 return <div>
5   <div style={{ margin: 10 }}>
6     <div>Select Field</div>
7     <Select style={{ width: 120 }} onSelect={setSelectFieldId} options=
      {formatFieldMetaList(currencyFieldMetaList)}/>
8   </div>
9   <div style={{ margin: 10 }}>
10    <div>Select Currency</div>
11    <Select options={CURRENCY} style={{ width: 120 }} onSelect=
      {setCurrency}/>
12    <Button style={{ marginLeft: 10 }} onClick=
      {transform}>transform</Button>
13  </div>
```

接下来实现我们最重要的一步：在 `transform` 函数中实现货币字段的货币类型转换以及数值转换。

```
1 const transform = async () => {
2   // 如果用户没有选择货币或者转换的字段，则不进行转换操作
3   if (!selectFieldId || !currency) return;
4   const table = await bitable.base.getActiveTable();
5   // 获取货币字段，这里我们传入了一个 ICurrencyField
6   // 来表明我们获取的是一个货币类型的字段
7   // 在使用 ts 的情况下，我们限制了这个字段的类型之后
8   // 在开发时就会获得很多类型提示，来帮我们进行开发
9   const currencyField = await table.getField<ICurrencyField>
    (selectFieldId);
10  const currentCurrency = await currencyField.getCurrencyCode();
11  // 设置货币类型
```

```

12     await currencyField.setCurrencyCode(currency);
13     // 获取货币的汇率
14     const ratio = await getExchangeRate(currentCurrency, currency);
15     if (!ratio) return;
16     // 首先我们获取 recordId
17     const recordIdList = await table.getRecordIdList();
18     // 对 record 进行遍历
19     for (const recordId of recordIdList) {
20         // 获取当前的货币值
21         const currentVal = await currencyField.getValue(recordId);
22         // 通过汇率进行新值的运算
23         await currencyField.setValue(recordId, currentVal * ratio);
24     }
25 }

```

在上面的例子中，我们在获取字段时传入了对其类型的限制，从而在后续的逻辑中得到了足够的类型提示，这一步非常重要，我们非常推荐开发者用类似的方法来获取字段，从而提高开发体验。

在修改货币类型时，可以直接调用 `CurrencyField.setCurrencyCode` 来改变对应的货币类型，这也是得益于在获取对应的字段时我们提供了类型（在这个基础上，需要修改单选/多选字段的选项时，也可以做到类似的效果）。

在设置货币值的时候，我们用 `CurrencyField.getValue` 来获取对应的数据，然后进行运算，回填的时候，也是调用了 `CurrencyField.setValue`，我们非常推荐 开发者在对值进行增删改查的时候从字段入手，我们细化了非常多的字段类型，从而优化开发者的使用体验（例如附件字段，在 `setValue` 时支持直接传入文件，来达到设置对应值的目的）。

[货币转换插件完整代码地址](#)

实现一个服务端插件（上架需自备服务）

以 [Base Node.js SDK](#) 为例，演示如何开发一个服务端插件。

准备开发环境

安装 SDK

npm

```
1 npm i -S @lark-base-open/node-sdk
```

yarn


```
1 yarn add @lark-base-open/node-sdk
```

实现逻辑

在准备好开发环境的基础上，我们来开发一个批量查找替换插件。

```
1  import { BaseClient } from '@lark-base-open/node-sdk';
2
3  // 新建 BaseClient, 填写需要操作的 appToken 和 personalBaseToken
4  const client = new BaseClient({
5    appToken: 'xxx',
6    personalBaseToken: 'xxx'
7  });
8
9  const TABLEID = 'xxx';
10
11 interface IRecord {
12   record_id: string;
13   fields: Record<string, any>
14 }
15
16 // 查找替换
17 async function searchAndReplace(from: string, to: string) {
18   // 获取当前表的字段信息
19   const res = await client.base.appTableField.list({
20     params: {
21       page_size: 100,
22     },
23     path: {
24       table_id: TABLEID,
25     }
26   });
27   const fields = res?.data?.items || [];
28   // 文本列
29   const textFieldNames = fields.filter(field => field.ui_type ===
'Text').map(field => field.field_name);
30
31   // 遍历记录
32   for await (const data of await
client.base.appTableRecord.listWithIterator({ params: { page_size: 50 },
path: { table_id: TABLEID } })) {
33     const records = data?.items || [];
```

```

34     const newRecords: IRecord[] = [];
35     for (const record of records) {
36         const { record_id, fields } = record || {};
37         const entries = Object.entries<string>(fields);
38         const newFields: Record<string, string> = {};
39         for (const [key, value] of entries) {
40             // 替换多行文本字段值
41             if ((textFieldNames.includes(key)) && value) {
42                 const newValue = value.replace(new RegExp(from, 'g'), to);
43                 // 把需要替换的字段加入 newFields
44                 newValue !== value && (newFields[key] = newValue);
45             }
46         }
47         // 需要替换的记录加入 newRecords
48         Object.keys(newFields).length && newRecords.push({
49             record_id,
50             fields: newFields,
51         })
52     }
53
54     // 批量更新记录
55     await client.base.appTableRecord.batchUpdate({
56         path: {
57             table_id: TABLEID,
58         },
59         data: {
60             records: newRecords
61         }
62     })
63 }
64 console.log('success')
65 }
66
67 searchAndReplace('abc', '23333333');
68
69 console.log('start')

```

SDK

我们提供了多个语言版本的 SDK，将所有冗长的接口逻辑内置处理，提供完备的类型系统、语义化的编程接口，提高开发者的编码体验。根据的业务场景选择适合的技术栈和插件运行形态，纯前端项目偏重通过界面交互实现对多维表格的数据操作，服务端插件具有脱离界面的运行能力，可兼顾部分自动化功能，或是两者结合。

前端 SDK

- [Base JS SDK](#)

服务端 SDK

- [Base Node.js SDK](#)
- [Base Python SDK](#)
- [Base Golang SDK](#)

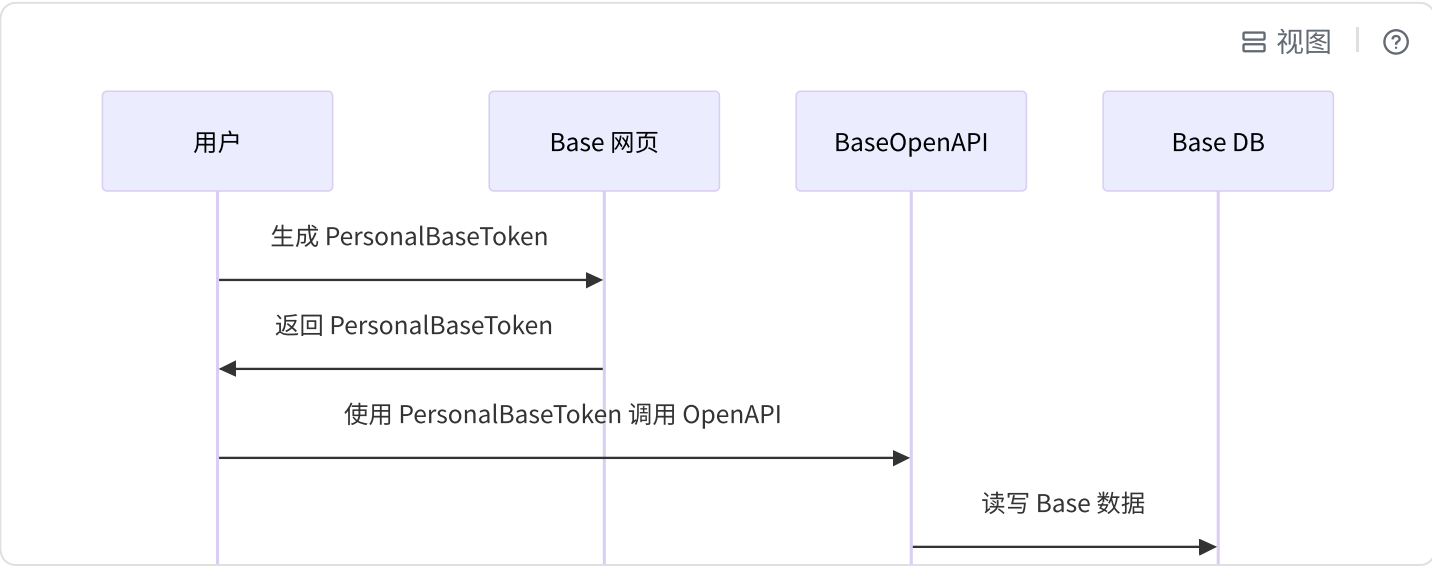
鉴权

前端插件

前端插件运行时将获取当前登录的用户身份，插件的权限范围与登录用户权限范围一致。

服务端插件

针对多维表格服务端 SDK，我们制定了独立的鉴权体系，获取和使用更为方便。开发者只需在网页端获取多维表格对应的 授权码 `PersonalBaseToken`，即可在服务端通过 SDK 操作多维表格数据。



- 多维表格的 **所有者 / 管理员** 有权限获取
- 通过授权码可以读写对应多维表格的数据，**请勿公开传播**
- 通过授权码调用服务端接口操作多维表格数据时，权限范围与 **授权码的生成者** 一致
- 授权码仅可操作对应的多维表格，每个多维表格的授权码需要独立生成和使用，互不影响
- 授权码默认 **永久有效**，除非在多维表格网页端手动关闭或更新

获取授权码



使用授权码

多维表格在新域名独立部署了一套 Base 业务的 OpenAPI，接口路径、接口定义和 [飞书开放平台](#) 完全一致，没有额外的学习成本，授权码仅能操作Base 相关的开放接口。

- 云文档 - 多维表格（Base）：[全部接口](#)
- 云文档 - 云空间（Drive）：[上传素材](#)、[下载素材](#)两个接口

以「列出记录」接口为例：	
协议和方法	HTTP GET
域名	https://base-api.feishu.cn
与 OpenAPI 不同	https://base-api.larksuite.com
Path 和 Seesion	API 路径：/open-apis/bitable/v1/apps/:app_token/tables/:table_id/records
Header	Authorization: Bearer [PersonalBaseToken]
QueryParam	<div><pre>1 filter:CurrentValue.[多行文本]="双向关联测试" 2 sort:["字段1 DESC"] 3 page_size:20</pre></div>
Response	<div><pre>1 2 { 3 "code": 0, 4 "data": { 5 "has_more": true, 6 "items": [7 { 8 "fields": { 9 "其他表": [10 { 11 "record_ids": [12 "rec2ltnYkQ", 13 "recZE5zqYP" 14], 15 "table_id": "tblvC2gefQet5bTV", 16 "text": "测试,测试2", 17 "text_arr": [18 "测试", 19 "测试2" 20], 21 "type": "text" 22 } 23], 24 },</pre></div>

```
25         "id": "rec0iAsbhD",
26         "record_id": "rec0iAsbhD"
27     }
28 ],
29     "page_token": "rec0iAsbhD",
30     "total": 9
31 },
32     "msg": "success"
33 }
```

UI builder

UI builder 将复杂的 UI 渲染简化成一行命令调用的渲染框架，以降低 UI 搭建成本。支持通过 SDK 引用，或直接 Fork [UIBuilder Template](#) ([github地址](#))，然后在 `src/runUIBuilder.tsx` 文件的 `main` 函数内调用 `UIBuilder` 的方法。详细使用方法可通过 [UIBuilder 模板使用指南](#) 了解。

使用模板

我们在 <https://replit.com>（可选，或者直接clone github 上模板（见下文中的“模板”）即可）中提供了一系列的模板帮助你快速开发，请根据你的业务场景和技术栈，选择对应的模板Fork 到你的 Replit 账号内，或导入 GitHub 进行开发。

前端插件模板

- 如果你准备开发表单UI，或者你是非前端开发同学，想降低 UI 搭建成本，建议使用以下模板：
 - a. [UIBuilder 模板](#)，[github地址](#) 入口为 `src/runUIBuilder.tsx` 文件，具体使用方法见 [UIBuilder 模板使用指南](#)
- 如果你有前端开发经验则可以使用以下模板来自由搭建 UI：
 - a. [HTML 模板](#)，[github地址](#) 入口为 `src/index.ts` 文件
 - b. [React 模板](#)，[github地址](#) 入口为 `src/App.tsx` 文件
 - c. [Vue 模板](#)，[github地址](#) 入口为 `src/App.vue` 文件

服务端插件模板

- [Nodejs 模板](#)
- [Python 模板](#)

前后端混合插件模板

nextjs无法静态部署，如需官方部署，请优先使用其他的模板。

1. [Nextjs 模板](#)，前端入口为 `pages/index.tsx` 文件，服务端入口为 `pages/api` 目录下的文件

参考项目

由其他开发者提交并同意开发源代码的项目被集中整合，开发者可通过查看 [参考项目](#)，阅读源代码来寻找灵感。

合法域名

我们没有对域名进行限制，只要是 HTTPS 协议连接都可以正常运行。如果出于安全合规方面原因希望限制某些特定域名的访问，可以填写 [申请表单](#) 将指定域名加为黑名单。

发布到插件中心

完成插件开发后，你可以将其 [发布到插件中心](#)，以供所有多维表格用户使用。插件发布到插件中心后，将由官方托管部署。在此之前你需要对插件的基本信息进行补充，我们对每个元素的价值及要求进行了说明，并提供了示例，以帮助你顺利完成发布前的准备。

发布表单：[发布到插件中心](#)

Check list

只要提交一下信息就可以将插件发布到市场，但更加完善的信息有助于插件被更多用户使用。

- 插件名
- 项目代码地址
- 简短描述
- 类别
- 使用录屏

简短描述

用户在浏览插件中心时会看到卡片上的简短描述，使用尽可能精简的语句描述该插件的功能及价值。推荐使用主动动词（如添加、实施、创建、更新、可视化等）撰写基于动作的描述。

- 必要项
- 最多 X 个字符
- 示例：按照一定条件查找重复的记录，并删除它们。



详细介绍

在插件介绍页展示，它应该具体阐释插件的功能，通过步骤介绍如何使用插件，以及出现使用问题时该如何寻求帮助，确保用户对插件有完整的了解。

- 非必要项，如开发者无法提供，我们的运营人员将通过 AI 为其生成
- 200 至 2,000 个字符
- 建议使用换行符或项目符号列表令版式更为美观
- 支持通过 Markdown 编辑器生成 Markdown 语句

我们建议遵循此结构：

第 1 段：突出显示插件的主要功能、解决的问题以及核心优势，确保用户仅用一段文字就能理解插件的功能。

第 2 段：分享更多用例并提供有关插件的更多背景信息。

第 3 段：提供一个用户可以寻求帮助的路径，例如帮助文档链接或联系方式。



插件名称

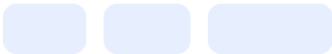
开发者 |



第 1 段：突出显示插件的主要功能、解决的问题以及核心优势，确保用户仅用一段文字就能理解插件的功能。

第 2 段：分享更多用例并提供有关插件的更多背景信息。

第 3 段：提供一个用户可以寻求帮助的路径，例如帮助文档链接或联系方式。



类别

插件中心允许用户根据类别筛选插件，从以下列表中选择插件所属的类别：

- 必要项
- 最多选择三个

批量处理	文本处理	图表	营销
内容转换	开发工具	导入导出	人事行政
提取解析	筛选查询	AI	进销存
附件处理	表结构处理	翻译	设计工具

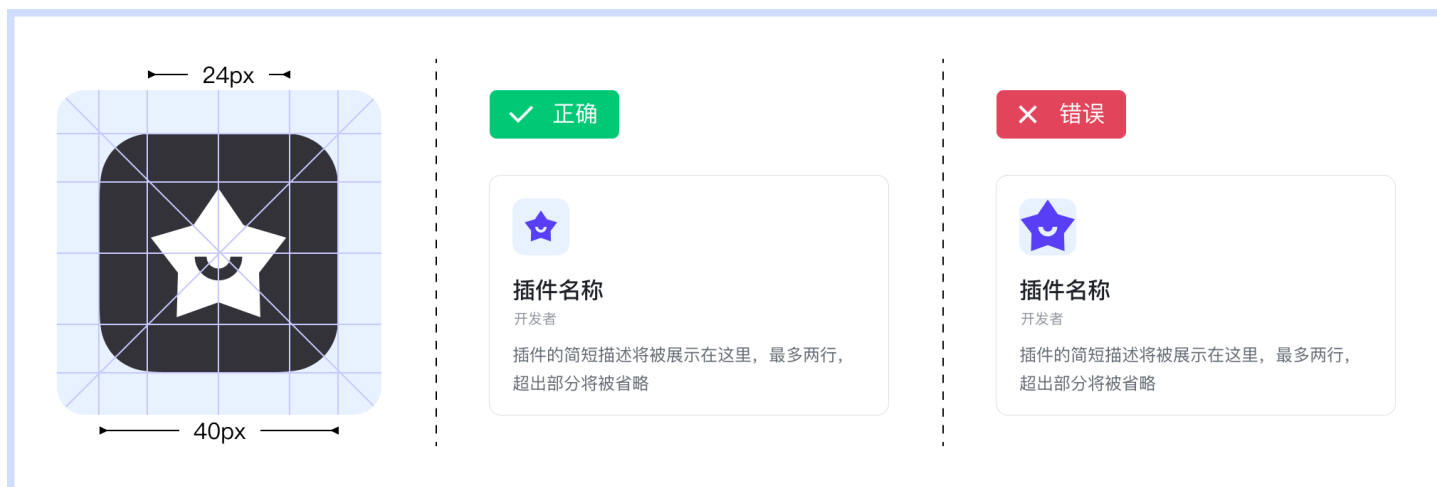
如果没找到适合的分​​类可通过 [交流群](#) 向我们反馈。

图标

使用图形语言尽可能的传达插件功能，避免出现复杂细节影响可识别性，并确保不存在版权风险，我们提供了 [remixicon](#) 和 [iconpark](#) 两套开源图标库可供使用。你可以使用此 [模板](#) 创建自己的图标，选择图标背景与元素的颜色搭配，并调整元素大小保持在框架内。

- 必要项，如开发者无法提供，将由我们的运营人员代为生成

- 推荐 SVG 格式
- 或 128 像素 x 128 像素 JPG / PNG



色板



介绍图片

通过若干静态图片来突出插件的主要特征、界面、品牌和标识。这些图片应将裁剪后的、重点突出的界面与简短文字说明结合起来。

- 非必要项
- 宽高比例必须与图片/其他视频一致
- 推荐 SVG 格式
- 或 1920 像素 x 960 像素 JPG / PNG



在视觉上突出插件的功能和特性，而不是简单地截图。建议使用彩色背景，以确保图像在所有主题（包括深色模式）中脱颖而出。每张图片都应侧重于介绍插件的一个功能点，使用户感受到价值。



介绍视频

使用此视频演示插件的特性、功能和用户界面，以帮助用户快速了解如何操作使用该插件。

- 必要项
- 宽高比例必须与图片/其他视频一致

- 不超过 20 秒
- MP4 或 GIF 图

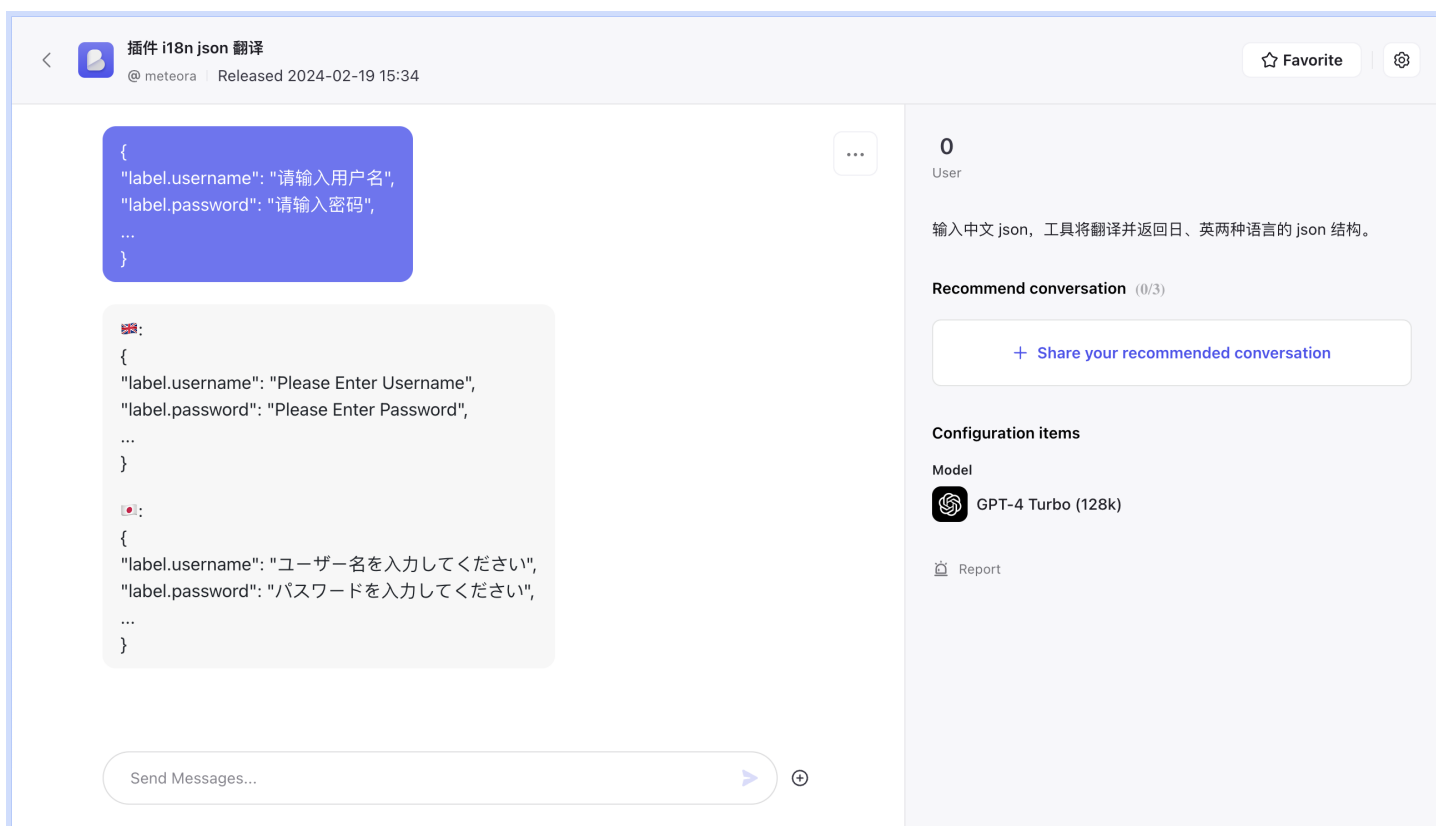


推荐使用桌面端即时消息截图工具，使用快捷键 **Alt + Shift + R**（Windows）或 **Option + Shift + R**（Mac），框选录屏区域。或是鼠标悬浮于 **截图** 按钮，选择 **录屏**，框选录屏区域，选择 **MP4** 或者 **GIF** 格式，点击 **开始录制** 即可。



国际化

由于国际化和市场团队的要求，发布到市场的插件必须通过 i18n 能力支持中、日、英三种语言。请务必使用 [插件 i18n json 翻译](#) 工具完成国际化，以确保专有名词的正确性。输入中文 json，工具将翻译并返回日、英两种语言的 json 结构。



示例：

./locales/zh.json

```
1  {
2    "label.username": "请输入用户名",
3    "label.password": "请输入密码",
4    ...
5  }
```

./locales/en.json

```
1  {
2    "label.username": "Please enter your username",
3    "label.password": "Please enter your password",
4    ...
5  }
```

./locales/jp.json

```
1  {
2    "label.username": "ユーザー名を入力してください",
3    "label.password": "パスワードを入力してください",
4    ...
5  }
```

./i18n.ts

```
1  import i18n from 'i18next';
2  import { initReactI18next } from 'react-i18next';
3
4  import translationEN from './locales/en.json';
5  import translationZH from './locales/zh.json';
6  import translationJP from './locales/jp.json';
7
8  // 设置支持的语言列表
9  const supportedLanguages = ['en', 'zh', 'jp'];
10
11
```

```
12 export function initI18n(lang: 'en' | 'zh' | 'jp'){
13   // 初始化 i18n
14   i18n.use(initReactI18next).init({
15     resources: {
16       en: {
17         translation: translationEN,
18       },
19       zh: {
20         translation: translationZH,
21       },
22     },
23     lng: lang, // 设置默认语言
24     fallbackLng: 'en', // 如果没有对应的语言文件，则使用默认语言
25     interpolation: {
26       escapeValue: false, // 不进行 HTML 转义
27     },
28   });
29
30 }
```

UI & 交互

出于视觉一致性考虑，对于主要由表单按钮等简单ui组成的插件，我们强烈建议开发者使用UIBuilder来搭建，如无法满足插件功能而需要自定义样式，也应符合基础的设计规范，确保插件的视觉质量。在插件审核时，我们会确保一些基础的设计规范符合标准，如果想进一步提高插件的品质，可以参阅[Base 开放设计规范](#)。



如果你使用 AI 编程，强烈推荐将以下「Base 开放设计规范」给到 AI 作为参考，以保证前端 UI 的美观性和一致性。

- AI 编程工具推荐：<https://trae.ai>、<https://www.trae.com.cn>、<https://www.cursor.com>



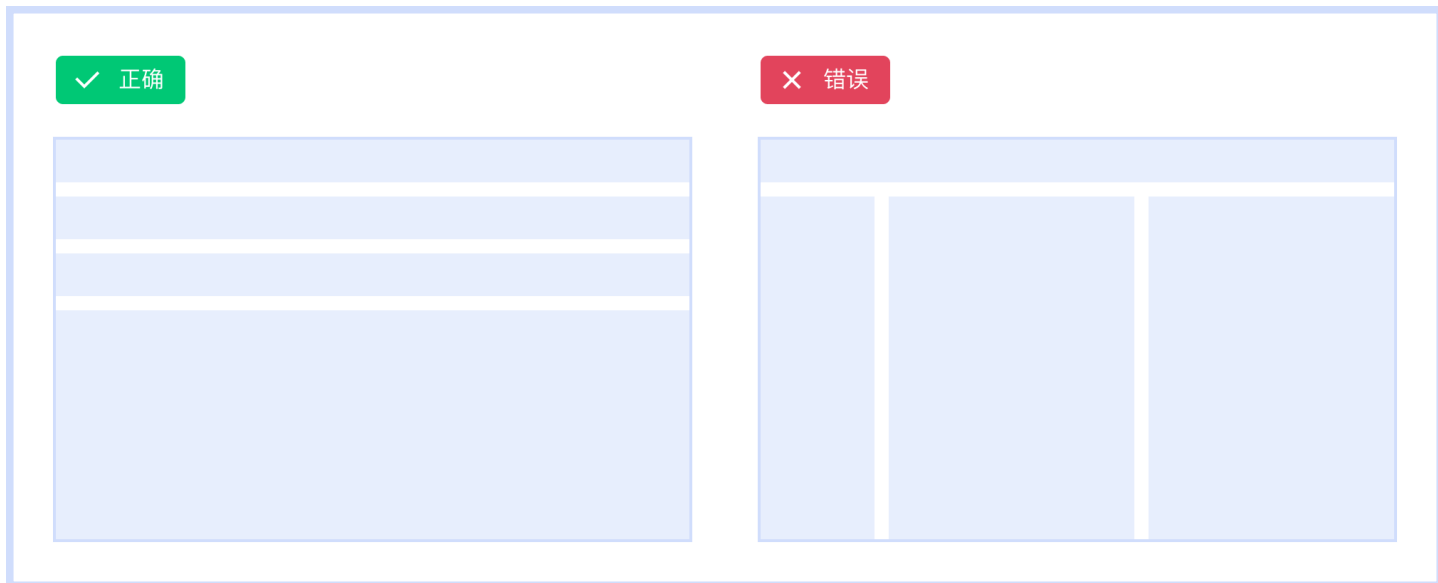
Base 开放设计规范.zip

12.68MB



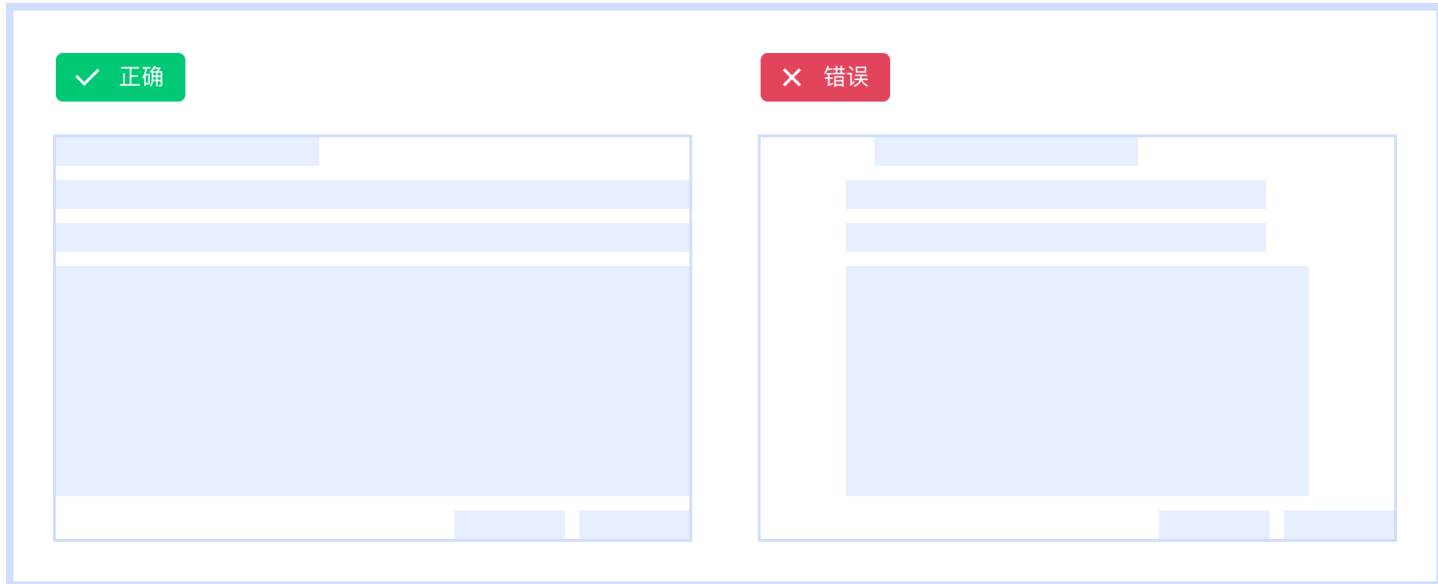
布局

由于侧边栏插件运行面板可拖拽改变宽度，因此我们建议采用垂直布局，以保证小宽度情况下的可用性。并在不同功能区及页面元素之间留有适当的间距。



自适应

在 CSS 样式处理上，应当注意使用动态单位及合理的对齐，确保页面元素在不同面板宽度下保持良好的自适应。侧边栏插件运行面板最小宽度为 410px，最大宽度是根据页面宽度动态计算得出，最大宽度=页面宽度-左侧侧边栏宽度-640px。



字体

优先使用系统默认的界面字体，同时提供一套备用字体库，来维护在不同平台以及浏览器的显示下，字体始终保持良好的易读性和可读性，建议开发者同样使用这套字体规则以保证兼容性。


```
2 font-family:-apple-system,BlinkMacSystemFont,Helvetica Neue,Tahoma,PingFang
  SC,Microsoft Yahei,Arial,Hiragino Sans GB,sans-serif,Apple Color Emoji,Segoe
  UI Emoji,Segoe UI Symbol,Noto Color Emoji;
3 //日文环境
4 font-family:"ヒラギノ角ゴ Pro W3", "Hiragino Kaku Gothic Pro", "Yu Gothic UI",
  "游ゴシック体", "Noto Sans Japanese", "Microsoft Jhenghei UI", "Microsoft Yahei
  UI", "MS Pゴシック", Arial, sans-serif,Apple Color Emoji,Segoe UI Emoji,Segoe
  UI Symbol,Noto Color Emoji;
```

主题色兼容

多维表格支持切换「浅色（light mode）」和「深色（dark mode）」两种外观模式，因此插件在视觉上也需要进行兼容。插件的 iframe 容器天然兼容两种主题色，因此开发者无需额外设置插件内元素的背景色，只需要关注元素本身的颜色即可。开发者可以通过 [Base JS SDK](#) 中的 `getTheme` 和 `onThemeChange` 方法来实现主题色切换。

获取当前主题 `getTheme`

```
1 getTheme(): Promise<ThemeModeType>;
```

`ThemeModeType` 类型定义

```
1 enum ThemeModeType {
2   LIGHT = "LIGHT",
3   DARK = "DARK"
4 }
```

示例

```
1 const theme = await bitable.bridge.getTheme();
2 // 'LIGHT'
```

监听主题变化 `onThemeChange`

```
1 onThemeChange(callback: (ev: IEventCbCtx<ThemeModeCtx>) => void): () => void;
```

示例

```
1 const theme = await bitable.bridge.onThemeChange((event) => {  
2   console.log('theme change', event.data.theme);  
3 });
```

以 Vue3 + Element Plus 技术栈为例，封装一个 hook，通过在 App.vue 中导入，实现整个插件的全局使用

useTheme.ts

```
1 import { bitable } from '@lark-base-open/js-sdk';  
2  
3 export const useTheme = () => {  
4   const theme = ref('');  
5  
6   const setThemeColor = () => {  
7     const el = document.documentElement;  
8  
9     // 处理主要样式  
10    const themeStyles = {  
11      LIGHT: {  
12        '--el-color-primary': 'rgb(20, 86, 240)',  
13        '--el-bg-color': '#fff',  
14        '--el-border-color-lighter': '#dee0e3',  
15      },  
16      DARK: {  
17        '--el-color-primary': '#4571e1',  
18        '--el-bg-color': '#252525',  
19        '--el-border-color-lighter': '#434343',  
20      },  
21    };  
22  
23    const currentThemeStyles = themeStyles[theme.value];  
24  
25    // 设置样式变量  
26    Object.entries(currentThemeStyles).forEach(([property, value]) => {  
27      el.style.setProperty(property, value);  
28    });  
29  };
```

```

30
31 // 挂载时处理
32 onMounted(async () => {
33     theme.value = await bitable.bridge.getTheme();
34     setThemeColor();
35 });
36
37 // 主题修改时处理
38 bitable.bridge.onThemeChange((event) => {
39     theme.value = event.data.theme;
40     setThemeColor();
41 });
42
43 // 抛出当前主题变量
44 return {
45     theme
46 };
47 };

```

App.vue

```

1  <script setup>
2    import Form from './components/Form.vue';
3    import { useTheme } from '@/hooks/useTheme';
4
5    // 使用 useTheme hook
6    useTheme();
7  </script>
8
9  <template>
10    <main>
11      <Form />
12    </main>
13  </template>

```

代码规范

在插件审核时，我们会对代码进行 review，以规避在数据安全和性能等方面存在的隐患。

插件上架到多维表格的步骤

1. 本地打包

我们将直接静态部署前端产物，以避免二次构建打包，首先需要指定构建产物目录，比如指定部署 `dist/index.html`，

在 `package.json` 中设置 `output` 属性值为 `"dist"` 即可，注意需要同时去掉 `.gitignore` 文件中的 `dist`，并在每次代码更新之后重新打包构建一下，然后将重新构建(`npm run build`)的 `dist` 上传即可。

代码块

```
1  {
2    "output": "dist" // 指定直接上传 dist 目录了
3  }
```

同时需要注意的是，打包产物的资源引用路径不可以使用绝对路径，请使用相对路径，如在 `vite.config.js` 中指定 `base: './'`：

代码块

```
1  import { defineConfig } from "vite";
2
3  export default defineConfig({
4    base: "./", // 使用相对路径
5    //....
6  });
7
```

2. 禁止使用 `history` 路由，请使用 `hash` 路由

3. 填写 [共享表单](#)

4. 多维表格审核通过后会 Fork 和部署你的项目初始化配置

插件运行时，需要基于插件功能遍历当前多维表格数据结构，进行初始化配置。如无法获取正确的数据结构，或必须依赖选中单元格，则需给出明确提示，引导用户如何操作，避免产生疑惑。

通用的初始化配置逻辑

- 遍历所有表的所有字段类型
- 遍历所有表的记录数
- 选取符合字段类型要求且记录数最多的表
- 如对字段类型无严格限制，则直接判断记录数
- 如所有表都没有记录，则用第一个张表
- 按返回序列为插件配置项匹配适合的字段

监听事件

前端项目应当实时监听 `base`、`table`、`view`、`field`、`record`、`cell` 的数据变化，以及选中状态变化。当上述维度发生改变时，插件应当即时响应，而无需用户手动刷新。

性能

在批量操作数据时，建议使用 `addRecords`、`setRecords`、`deleteRecords`、`getRecords` 等批量接口来增删改查行记录，而不是使用单次接口循环遍历。

数据安全

为确保数据安全，除插件功能必要的 API 请求外，禁止将多维表格数据向外部发送。

常见问题

如何获取 `appToken`

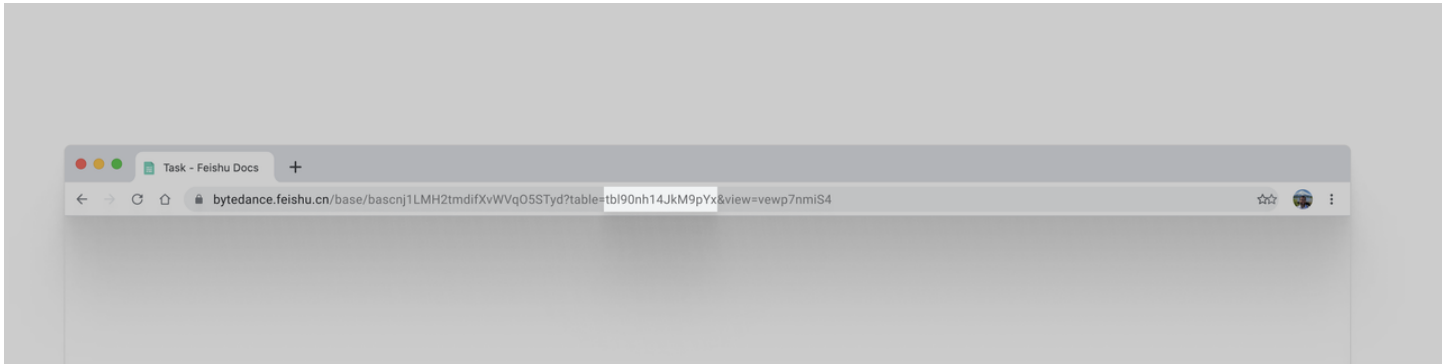
多维表格 URL 中如下图所示部分。（推荐使用「[开发工具](#)」插件获取）



！ 注意 URL 路径必须是 `base/` 后面获取的才是正确的 `appToken`，如果路径为 `wiki/` 则必须使用「[开发工具](#)」插件获取。

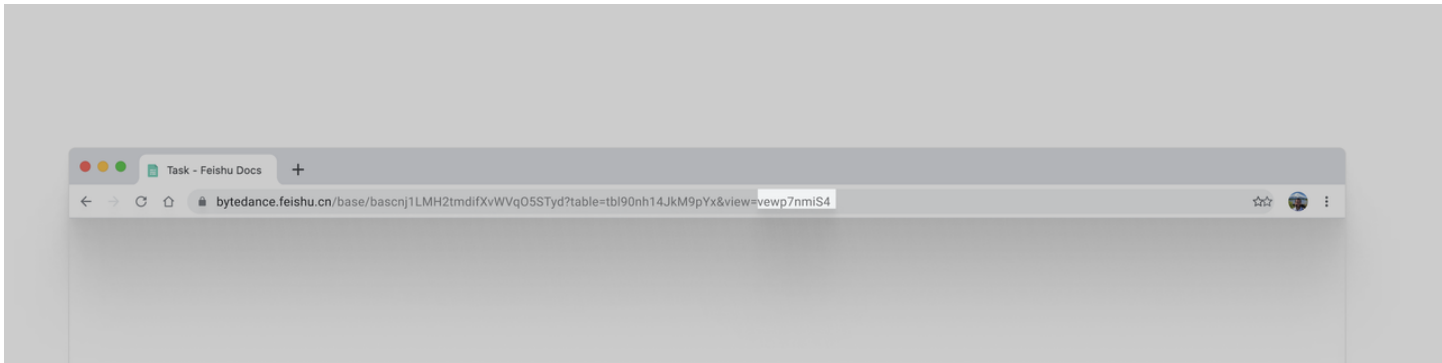
如何获取 `table_id`

多维表格 URL 中如下图所示部分（推荐使用「[开发工具](#)」插件获取）



如何获取 `view_id`

多维表格 URL 中如下图所示部分（推荐使用「[开发工具](#)」插件获取）



边栏插件相关的参数

可在多维表格 url 后添加这些参数，以实现特定功能

```
1  隐藏侧边栏
2  hideSidebar=1
3
4  设置侧边栏展开宽度
5  extension_market_spread_width={number}
6
7  打开插件市场
8  extension_market_spread=1
9
10 打开插件
11 extension_market_extension_id={id}
12 效果预览：Markdown
```

插件的部署和安全

多维表格插件由多维表格官方和第三方开发者提供，其中代码均由多维表格官方审核，部署在多维表格官方服务器以及认证 ISV 的服务器上，以保证插件的安全合规。

插件权限

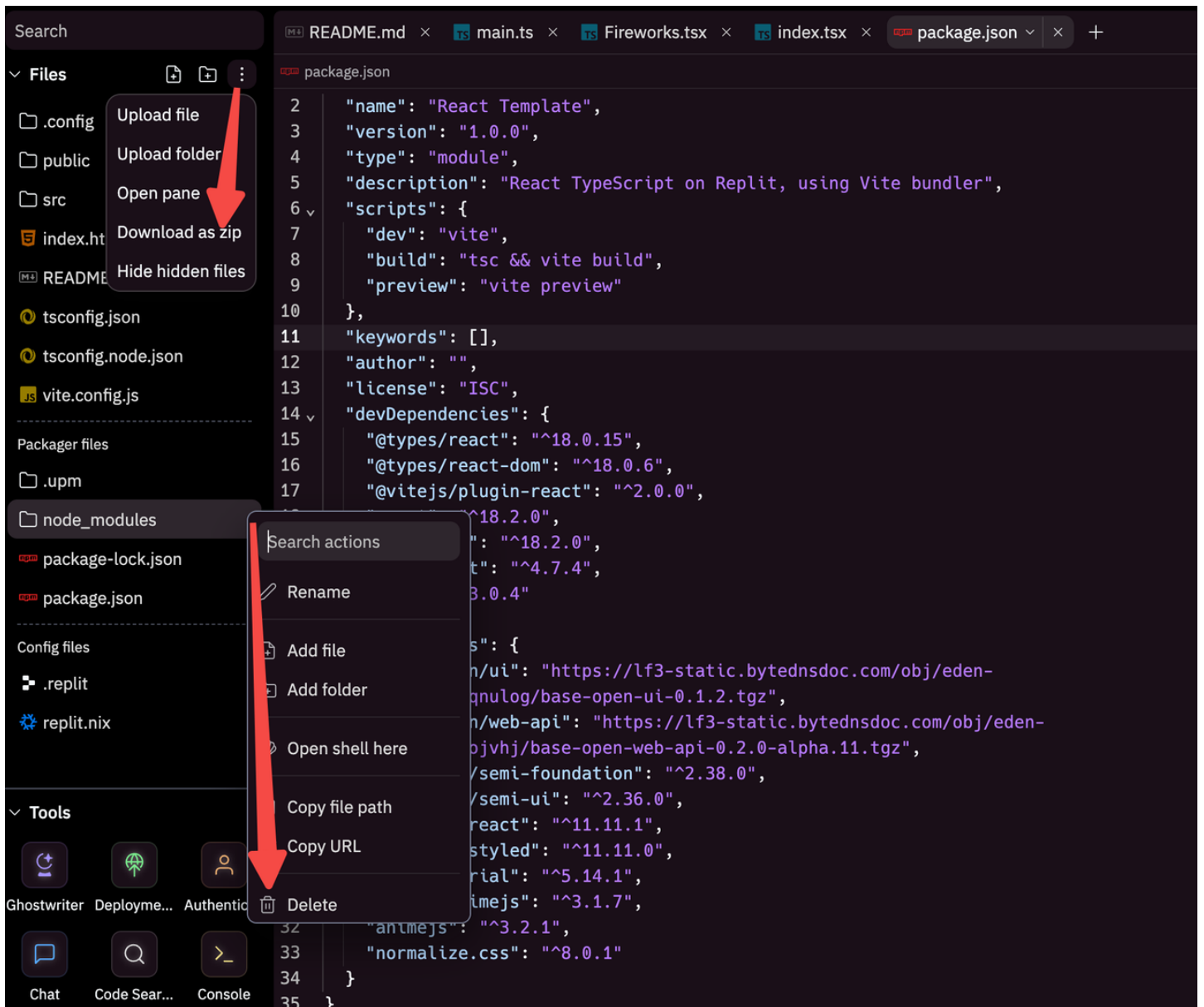
为了确保客户数据安全，插件权限是受限制的

1. 前端插件，接口的权限会跟随执行插件的人；（简而言之：如果该用户在多维表格界面上无权看某些数据，那么插件中也看不到）
2. 服务端插件，插件运行依赖文档所有者提供 PersonalBaseToken ，这个权限等于「文档所有人」的身份，但是必须由他/她亲自获取后提供

如何本地调试开发

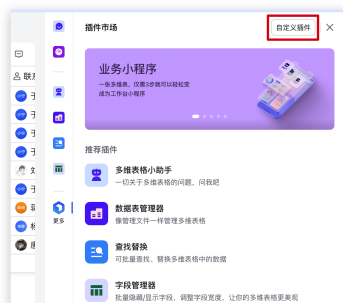
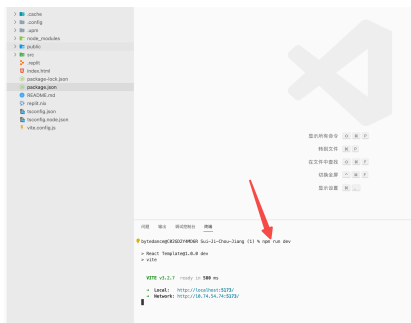
你可以将 fork 出来的模板的 node_modules 删掉（如果不删掉就直接下载，可能会有点慢），然后将代码下载到本地。

为了在本地运行代码，你需要安装 [nodejs](#) 以及 [vscode](#)。



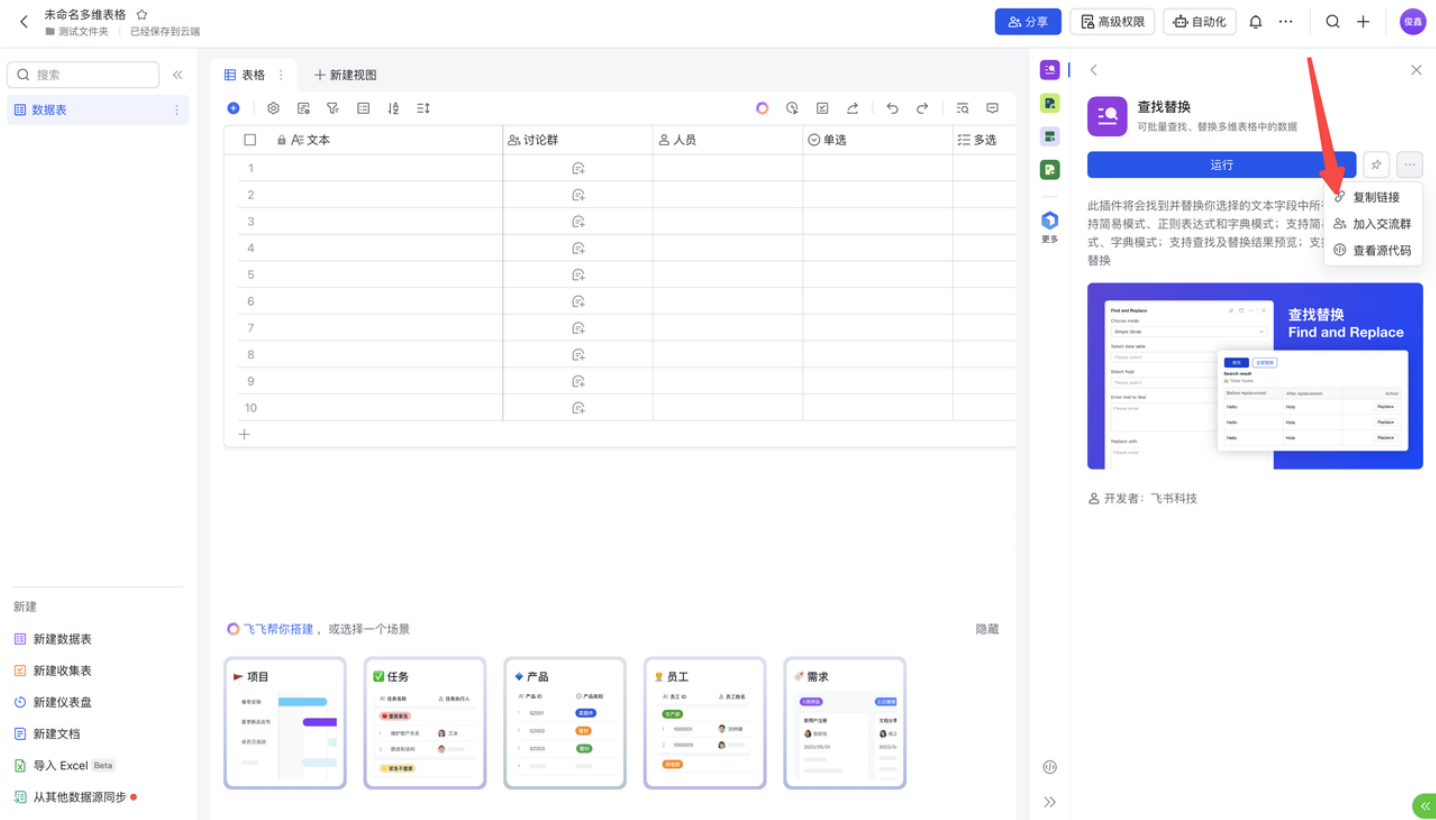
然后使用 vscode 打开下载的文件夹，然后新建终端，输入 `npm i` 并按回车；

安装完毕依赖包之后执行 `npm run dev`，将本地起的 localhost 项目链接粘贴到插件的预览地址输入框中，然后点击确定就可以看到本地预览的效果了。



如何将插件分享给其他用户

1. 在任意多维表格内访问插件详情，点击「复制链接」获取可分享的插件 URL 地址



2. 你可以将该地址分享给其他用户

https://bytedance.larkoffice.com/base/extension/replit_3c67c9577361a3e3