# ChangeLog

## 0.3.8

### Feat

* 支持**表格视图**获取当前选中的所有记录 id
* 新增 Bridge.navigateToExtension 实现插件间跳转能力
* 新增 Bridge.onDataChange 事件，用于监听插件数据存储变更

### Fix

* 插件相关权限将受文档权限管控

## 0.3.6

### Feat

* 支持插件自定义存储能力，详细用法参考 Bridge.getData/setData
* 支持配置公式字段 Formula 表达式
* UI 模块新增 UI.showRecordDetailDialog 用于在全局弹出记录详情弹窗，支持更丰富的业务场景联动

### Fix

* 修复部分用户刷新页面接口偶现变慢问题

## 0.3.5

### Feat

#### UI 模块

* 新增 UI.selectRecordIdList 方法，该方法会在全局唤起指定数据表选择记录 Modal框，方便当前用户选择对应的记录并返回记录 id 列表
* 新增 UI.showToast 方法用于在全局弹出消息提示
* 新增 UI.switchToTable/switchToView 方法用于切换当前选中的数据表/视图

#### Table 模块

* TableMeta 添加同步表标识

#### View 模块

* 支持视图筛选/排序/分组能力
* 支持开启/关闭视图分享并获取记录分享链接
* 新增多种类型视图模块能力，如支持获取表格视图的列宽等

#### Field 模块

* 支持邮箱字段
* 查找引用字段支持获取引用的 TableId 和 FieldId

### Fix:

* 修复 CellValue Checkers断言过宽问题

## 0.3.0

### Feat

* 发布高阶 API，抽象不同类型字段，新增 Record、Cell 模块

## 0.2.4

### Feat

* 新增 Table.getRecordShareLink 用于获取指定记录的分享链接
* 自动编号字段返回值新增是否处于计算中状态
* 优化部分接口调用出错的错误提示

### Fix

* 修复 Table.addRecords 批量调用耗时过长问题

## 0.2.2

### Feat

* 批量操作记录接口上限提升至 5000
* 新增 Base.addTable/setTable/deleteTable 方法用于新增/设置/删除数据表
* 新增 Table.addView/setView/deleteView 方法用于新增/设置/删除视图

### Fix

* 修复创建/修改人，创建/修改时间计算延迟问题
* 修复字段类型转换失败问题

## 0.2.1

### Feat

* 新增 Table.getRecords 方法用于批量获取记录
* 字段 Meta 新增 isPrimary 标识当前字段是否索引字段
* 字段 Meta 支持字段描述读写能力
* 新增 Bridge.getInstanceId 用于获取当前插件的唯一标识

### Fix

* 修复 View.getVisibleRecordIdList 跨表场景 id 为空问题
* 修复 Table.getRecordById 获取记录数据部分字段缺失问题
* 修复 Table.getCellAttachmentUrls 接口 QPS 过低问题

## 0.2.0

首次发布至 NPM

# FAQ

## 1. 数据表 (Table) 和视图 (View) 的关系

* **数据表**：可以将数据表理解成一个数据源，它负责维护数据与数据之间的联系，并不涉及 UI 展示(如字段顺序、记录顺序等)
* **视图**：不同的视图（如表格视图 GridView、看板视图 KanbanView 等）均是以某种 UI 的呈现形式来消费这个数据源

## 2. 为什么 Table 模块中 getRecordIdList 和 getFieldIdList 方法拿到的记录列表和字段列表均是无序的？如何拿到有序的记录列表和字段列表？

原因解释可以参考上面问题的答案，不同类型的 View 模块中均存在 [getVisibleRecordIdList](./api/view/grid.md#getvisiblerecordidlist) 和 [getVisibleFieldIdList](./api/view/grid.md#getvisiblefieldidlist) 方法，如果希望拿到有序的记录和字段列表，请通过这两个方法获取。

## 3. [Bridge.getUserId](../zh/api/bridge.md#getuserid) 方法获取到的 userId 能直接在飞书开放平台消费使用吗？

**不可以**，该接口获取到的 userId 和飞书开放平台中的 userId 并非一一对应的关系，**两者相互独立，请勿混用**。

# 实现一个货币转换的插件

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

import axios from 'axios';  
  
interface ExchangeRatesResponse {  
 rates: {  
 [key: string]: number;  
 };  
 base: string;  
 date: string;  
}  
  
export async function getExchangeRate(base: string, target: string): Promise<number | undefined> {  
 try {  
 const response = await axios.get<ExchangeRatesResponse>(`https://api.exchangerate-api.com/v4/latest/${base}`);  
 const rate = response.data.rates[target];  
  
 if (!rate) {  
 throw new Error(`Exchange rate not found for target currency: ${target}`);  
 }  
  
 return rate;  
 } catch (error) {  
 console.info(`Error fetching exchange rate: ${(error as any).message}`);  
 }  
}

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

1. 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

import { CurrencyCode } from '@lark-base-open/js-sdk';  
  
export const CURRENCY = [  
 { label: 'CNY', value: CurrencyCode.CNY },  
 { label: 'USD', value: CurrencyCode.USD },  
 { label: 'EUR', value: CurrencyCode.EUR },  
 { label: 'AED', value: CurrencyCode.AED },  
 { label: 'BRL', value: CurrencyCode.BRL },  
 { label: 'CAD', value: CurrencyCode.CAD },  
 { label: 'CHF', value: CurrencyCode.CHF },  
 { label: 'HKD', value: CurrencyCode.HKD },  
 { label: 'INR', value: CurrencyCode.INR },  
 { label: 'JPY', value: CurrencyCode.JPY },  
 { label: 'MXN', value: CurrencyCode.MXN },  
];

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

1. 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

import { bitable } from '@lark-base-open/js-sdk'; //[!code --]  
import { bitable, CurrencyCode, FieldType, ICurrencyField, ICurrencyFieldMeta } from '@lark-base-open/js-sdk'; //[!code ++]  
import { CURRENCY } from './const'; //[!code ++]  
  
function LoadApp() {  
 const [info, setInfo] = useState('get table name, please waiting ....'); // [!code --]  
 const [infoType, setInfoType] = useState('info'); // [!code --]  
 const [currencyFieldMetaList, setMetaList] = useState<ICurrencyFieldMeta[]>([]) // [!code ++]  
 const [selectFieldId, setSelectFieldId] = useState<string>(); // [!code ++]  
 const [currency, setCurrency] = useState<CurrencyCode>(); // [!code ++]

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

useEffect(() => {  
 const fn = async () => {  
 const table = await bitable.base.getActiveTable();  
 const tableName = await table.getName(); // [!code --]  
 setInfo(`The table Name is ${tableName}`); // [!code --]  
 setInfoType('success'); // [!code --]  
 const fieldMetaList = await table.getFieldMetaListByType<ICurrencyFieldMeta>(FieldType.Currency); // [!code ++]  
 setMetaList(fieldMetaList); // [!code ++]  
 };  
 fn();  
}, []);

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

const formatFieldMetaList = (metaList: ICurrencyFieldMeta[]) => { // [!code ++]  
 return metaList.map(meta => ({ label: meta.name, value: meta.id })); // [!code ++]  
}; // [!code ++]  
  
return <div>  
 <Alert message={info} type={alertType}/> // [!code --]  
 <div style={{ margin: 10 }}> // [!code ++]  
 <div>Select Field</div> // [!code ++]  
 <Select style={{ width: 120 }} onSelect={setSelectFieldId} options={formatFieldMetaList(currencyFieldMetaList)}/> // [!code ++]  
 </div> // [!code ++]  
 <div style={{ margin: 10 }}>// [!code ++]  
 <div>Select Currency</div>// [!code ++]  
 <Select options={CURRENCY} style={{ width: 120 }} onSelect={setCurrency}/>// [!code ++]  
 </div>// [!code ++]  
</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

1. 实现货币转换的逻辑

我们先将获取汇率的 API 引入

import { CURRENCY } from './const';  
import { getExchangeRate } from './exchange-api'; // [!code ++]

然后准备一个转换交互按钮以及转换函数

const transform = async () => { // [!code ++]  
}// [!code ++]  
  
return <div>  
 <div style={{ margin: 10 }}>  
 <div>Select Field</div>  
 <Select style={{ width: 120 }} onSelect={setSelectFieldId} options={formatFieldMetaList(currencyFieldMetaList)}/>  
 </div>  
 <div style={{ margin: 10 }}>  
 <div>Select Currency</div>  
 <Select options={CURRENCY} style={{ width: 120 }} onSelect={setCurrency}/>  
 <Button style={{ marginLeft: 10 }} onClick={transform}>transform</Button>// [!code ++]  
 </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

const transform = async () => {  
 // 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  
 if (!selectFieldId || !currency) return;  
 const table = await bitable.base.getActiveTable();  
 // 获取货币字段，这里我们传入了一个 ICurrencyField // [!code focus]  
 // 来表明我们获取的是一个货币类型的字段 // [!code focus]  
 // 在使用 ts 的情况下，我们限制了这个字段的类型之后 // [!code focus]  
 // 在开发时就会获得很多类型提示，来帮我们进行开发 // [!code focus]  
 const currencyField = await table.getField<ICurrencyField>(selectFieldId); // [!code focus]  
 const currentCurrency = await currencyField.getCurrencyCode();  
 // 设置货币类型  
 await currencyField.setCurrencyCode(currency);  
 // 获取货币的汇率  
 const ratio = await getExchangeRate(currentCurrency, currency);  
 if (!ratio) return;  
 // 首先我们获取 recordId   
 const recordIdList = await table.getRecordIdList();  
 // 对 record 进行遍历  
 for (const recordId of recordIdList) {  
 // 获取当前的货币值// [!code focus]  
 const currentVal = await currencyField.getValue(recordId);// [!code focus]  
 // 通过汇率进行新值的运算// [!code focus]  
 await currencyField.setValue(recordId, currentVal \* ratio);// [!code focus]  
 }  
}

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](https://github.com/Lark-Base-Team/js-sdk-learn-demo/tree/feat-currency-plugin)

# 准备开发环境

目前我们提供两种开发方式给刚刚入门的同学： 1. Replit 官网开发 (推荐新手用户使用) 2. 本地编辑器开发 (推荐有一定编程经验的同学使用)

## Replit 官网开发

1. 点击进入这个 [Replit 项目](https://replit.com/@RocketNasa1/js-sdk-learn-demo?v=1)
2. Fork 这个仓库 (点击红框展示的 Fork 按钮)
3. 输入一个喜欢的名字以及描述，然后点击右下角的 Fork Repl
4. 稍等一会，就可以进入如下所示的页面，此时点击顶部的 Run 按钮，稍等一会，Replit 就会将项目运行起来
5. 然后点击图中所示的 tab，然后将展示的 URL 复制，在第 6 步中使用 复制下面所示的 URL
6. 打开/新建 一个飞书多维表格，点击此处的扩展脚本按钮
7. 点击新增脚本，将 5 中复制的 URl 粘贴进去，然后点击确定
8. 在完成 7 中的确认弹窗之后，多维表格就会开始加载对应的插件，加载完成之后就会展示如下的效果(有时候会因为网络以及部署等原因，部署较慢，需要耐心等待一下，插件正式上架之后就不会有这样的问题) 这个插件作用是：获取当前的 Table 名称，并展示出来，后面我们基于这个基础 Demo，来实现一个货币转换的插件

## 本地开发

1. 点击进入这个 [Github项目](https://github.com/Lark-Base-Team/js-sdk-learn-demo), Fork 此仓库，并 git clone 到本地
2. 按照项目中 Readme.md 的内容，将项目运行起来，并按照提示复制对应的 URL 例如下图中的地址：
3. 打开/新建 一个飞书多维表格，点击此处的扩展脚本按钮
4. 点击新增脚本，将 5 中复制的 URl 粘贴进去，然后点击确定
5. 在完成 4 中的确认弹窗之后，多维表格就会开始加载对应的插件，加载完成之后就会展示如下的效果
6. 这个插件作用是：获取当前的 Table 名称，并展示出来，后面我们基于这个基础 Demo，来实现一个货币转换的插件

# 发布插件

## 本地开发的插件

1. 首先，本地开发的插件需要用户使用 GitHub 进行管理，并提交代码，然后打开这个链接 [Replit](https://replit.com/~)，完成注册登陆之后，点击 Create Repl
2. 然后点击 Import from Github
3. 选择对应的仓库 (第一次使用的同学这里会展示一个 Github 链接，点击链接进行授权登陆之后就可以了)
4. 点击 Import from GitHub
5. 稍等一会，项目就会成功被导入到 Replit 中，然后输入一个让项目运行的指令（这里默认是 npm run start，replit 没有 pnpm 所以需要用 npm 来启动）
6. 点击顶部的项目名称，将会出现 publish 按钮
7. 在填完信息之后，会出现 Publish to Community 按钮，然后点击
8. 会出现 Repl published 的提示，并附带地址，请点击旁边的复制按钮，将这个地址复制下来，会在后一步中用到
9. 在项目设置为 public 之后填写这个[表单](https://bytedance.feishu.cn/share/base/form/shrcnfhuWeBHoWTicKLFybn92Lb)，我们会与最快的速度与您联系，帮助您的插件在应用中正式上架 其中第 8 步复制来的 URl 填入到这一项中： ## Replit 官网开发的插件
10. 点击顶部的项目名称，将会出现 publish 按钮
11. 在填完信息之后，会出现 Publish to Community 按钮，然后点击
12. 会出现 Repl published 的提示，并附带地址，请点击旁边的复制按钮，将这个地址复制下来，会在后一步中用到
13. 在项目设置为 public 之后填写这个[表单](https://bytedance.feishu.cn/share/base/form/shrcnfhuWeBHoWTicKLFybn92Lb)，我们会与最快的速度与您联系，帮助您的插件在应用中正式上架 其中第 8 步复制来的 URl 填入到这一项中：

# 最后

更多的 API 用法可以点击 [API](../api/guide.md) 来查看，在此页面可以通过点击顶部的搜索框来进行 API 的搜索。

欢迎来到 Base JS SDK 文档，本章节将为你介绍，如何开发一个 Base 插件，并掌握 Base 插件开发的 80% 概念以及设计的核心概念。 [NPM 地址](https://www.npmjs.com/package/@lark-base-open/js-sdk)

前置学习： + 前端开发基础知识：HTMl / CSS / JS + [React](https://react.docschina.org/learn) (不限定 UI 框架使用 Vue 等框架的同学也可以制作插件)

推荐学习： + [Typescript](https://www.typescriptlang.org/docs/) # 核心知识

我们先了解一下多维表格的核心概念以及相关知识

下图所示的是 API 的设计模型:

bitable是 API 的入口，base ui bridge 等模块都会作为属性挂载在上面，一般会这样使用:

import { bitable } from '@lark-base-open/js-sdk'  
const table = await bitable.base.getActiveTable();  
const attachmentFieldList = await table.getFieldListByType<IAttachmentField>(FieldType.Attachment);

其中，table field 等模块都都可以在下面这张图中找到对应的关系：

下面介绍各个模块的概念：

* base: base 模块，可以理解为多个 table的集合体，可以通过 API 来获取对应的table，其中还有文件上传等 API 应用层面的 API
* table: table 模块是数据的集合体，不涉及页面上UI的展示，其概念和数据库中的表类似:含有field(字段) record(记录)，在多维表格中，table 还会含有 view(视图)的概念（在 table 层面获取字段信息的时候，往往是无序的，因为table层面只有数据，无法知晓其在view(视图)层面的展示方式）
* field: field 模块中含有很多不同的字段类型，例如：IAttachmentField(附件字段) 以及后续我们要用到的ICurrencyField(货币字段)等，Field字段细化了很多字段的操作方法以及设置字段属性的方法，例如为多选/单选字段提供了新增选项的方法，详细方法可以看对应的 API 文档，同时我们也推荐用户从字段入手结合 record 来对数据进行增删改查，以及获取 table 的 cell 或者通过 createCell 方法来创建一个 cell
* record: record 模块主要是用来获取数据的，其可以和 field 共同作用来获取 cell
* cell: cell 模块是的单元格，是 record/field 中的某一个单元格的数据，同时支持用户通过调用field的 createCell 方法来创建一个单元格，创建好的单元格可以在 table.addRecord 时作为参数传入进去，从而新增一条数据，在调用 createCell 方法时，为了便于开 发者调用，会简化对应的数据结构，创建的单元格被插入到 table 之后，会与一条 record 自动关联起来，在与 record 关联之后，cell 的 getValue/setValue 方法就会与 table 的数据关联起来，从而具有实时性
* view: view 模块是视图模块，其最主要的作用就是在当前视图（表格视图/看板视图）决定 table 中 field(字段)/record(记录) 的展示方式（顺序/是否可见），因此在 view 层中，去获取 field(字段)/record(记录) 的字段信息/记录顺序，往往都是与当前展现的内容相符合的（table层去获取这些信息的时候，往往会因为 table 层无法决定展现形式而无序）
* ui: ui 模块负责插件展示窗口 UI 的 API，与插件本身的应用能力有关
* bridge: bridge 模块主要提供应用层的接口，提供类似主题切换事件通知等插件能力的 API，与 UI 模块类似，都与应用能力有关系

接下来，我们搭建开发环境。

# 旧版本升级指南

目前很多开发者的 JS SDK 版本为 0.2.x，目前有较大改动的 API 版本为 0.3.x，开发者可以直接升级到 0.3.x 版本，无 Break Change，目前 JS SDK 的 [npm 地址](https://www.npmjs.com/package/@lark-base-open/js-sdk) 对比新版，最大的不同就是通过细化了字段的一些操作，我们推荐开发者先获取字段，然后实现对应的能力，在获取字段的时候，也强烈推荐通过传入对应的字段类型来获取完善的语法提示，例如 下面例子中涉及到的 IAttachmentField

const attachmentField = await table.getField<IAttachmentField>(attachmentFieldId);

用户可以通过更新对应 npm 中 @lark-base-open/js-sdk 的包版本为 0.3.x 来进行升级

# 新版本对比旧版本便利性提升

## 获取 Table 方式优化

在旧版本获取 Table 的 API 中：

const { tableId } = await bitable.base.getSelection();  
const table = await bitable.base.getTableById(tableId);

很多开发者都写了很多类似的摸板化代码，在新的 API 中，我们优化了这个过程，可以通过调用：

const table = await bitable.base.getActiveTable()

一步获取对应的 Table

## 优化开发者对数据进行增删改查的操作

在旧版本中，想要实现对数据的增删改查是非常麻烦的，在新版中我们做了一个非常大的优化，以附件字段举例，我们可以这样添加一个附件数据

const attachmentField = await table.getField<IAttachmentField>(attachmentFieldId);  
const attachmentCell = await attachmentField.createCell(File);  
const recordId = await table.addRecord(attachmentCell);

这是一种全新的新增记录的方式，我们首先获取对应的附件字段(table.getField<IAttachmentField>(attachmentFieldId))，然后通过这个字段的 createCell 方法来创建一个附件单元格（调用这个方法时，可以直接传入文件，不需要先进行文件上传等操作），然后将这个单元格 通过 table 下的 addRecord 方法插入到 table 中，从而实现插入一个带有附件数据的一条记录。对于其他类型的字段，我们也做了类似的优化，详情可以看 [API](api/guide.md) 部分。

## 细化字段，优化字段的属性设置

在旧版本中，对于字段属性的设置非常复杂，而且往往无法获取很好的语法提示，在新版本中我们优化了这个过程，可以让用户更加方便的设置字段的属性，例如在新版中可以这样新增一个单选字段的选项：

const singleSelectField = await table.getField<ISingleSelectField>(singleSelectFieldId);  
await singleSelectField.addOption('option1');

与优化数据增删改查的操作类似，首先获取对应字段，然后调用这个字段特有的方法

## 总结

在新版的 API 中，我们从字段的角度优化了数据结构，更多的相关内容可以查看 [API](api/guide.md) 部分,欢迎用户体验 👏。

# Record 模块

更推荐开发者们从 [Field(字段)](field/guide.md) 字段角度来考虑对数据的增删改查

在 Table 中，可以通过 getRecordList 接口获取到 RecordList(Record 记录的一个数组集合，其中有当前 table 下所有的记录），RecordList 是可以遍历使用的，使用方式如下（不推荐，会有性能相关问题）:

const recordList = await table.getRecordList();  
for (const record of recordList) {  
 const cell = await record.getCellByField(fieldId);  
 const val = await cell.getValue();  
}

cell 可以简单类比为表格中的单元格，其中有对应的值(关于 [Cell](cell.md))

const record = await recordList.getRecordById(recordId);

不过，还是更推荐开发者在对数据进行增删改查时，从 [Field(字段)](field/guide.md) 来考虑

## getCellList

获取当前记录中所有的 Cell(关于 [Cell](cell.md))

getCellList: () => Promise<ICell[]>;

#### 示例

const recordList = await table.getRecordList();  
const cellList = await recordList[0].getCellList();

## getCellByField

通过 Field 来获取 Cell(关于 [Cell](cell.md))

getCellByField: (fieldOrId: IField | string) => Promise<ICell>;

#### 示例

const recordList = await table.getRecordList();  
const textField = await table.getField('多行文本');  
const cell = await recordList[0].getCellByField(textField);

# View 模块

视图 View 是数据表 Table 的呈现方式（例如字段的展示顺序/记录的显示或隐藏等），一个数据表至少有一个视图，可能有多个视图，每个视图都有唯一标识 viewId，viewId 在一个多维表格中唯一。

注意此处与 Table 模块的差异，在 View 模块获取字段/记录的顺序都是**有序**的。

View 模块可以在 Table 层通过 getViewById 的方式获取

const view = await table.getViewById(viewId);

View 可以通过下图在得知其在页面中是负责 UI 展示的，因此很多与 UI 展示形式相关的 API 都存在于 View 层，例如筛选/分组/排序等

## 不同类型的视图

目前支持以下 6 种不同类型的视图，不同类型的视图可用能力存在差异：

* [GridView](./view/grid.md)：表格视图
* [KanbanView](./view/kanban.md)：看板视图
* [FormView](./view/form.md)：表单视图
* [GalleryView](./view/gallery.md)：画册视图
* [GanttView](./view/gantt.md)：甘特视图
* [CalendarView](./view/calendar.md)：日历视图

## View 基础能力

视图中最基础的能力包括筛选、排序和分组，下面将简介其用法，相关 API 定义在具体类型的模块中，如 [GridView](./view/grid.md)。

### 筛选

视图根据筛选条件过滤出数据表中符合条件的记录，主要由 FilterInfoCondition 过滤条件 和 FilterConjunction 生效条件 两部分信息组成

interface IFilterInfo {  
 conjunction: FilterConjunction;  
 conditions: FilterInfoCondition[];  
}

#### FilterInfoCondition

FilterInfoCondition 代表过滤条件，每个 Condition 由字段 + 过滤操作符 + 匹配值三个基本元素组成。

interface FilterInfoCondition {  
 fieldId: string; // field 唯一标识  
 conditionId?: string; // condition 唯一标识，新增时可不传入  
 value: FieldValue; // 字段匹配值  
 operator: FilterOperation; // 匹配操作符  
}

#### FilterConjunction

FilterConjunction 代表过滤条件的生效条件，FilterConjunction.And 代表符合所有过滤条件，FilterConjunction.Or 代表符合任一过滤条件：

enum FilterConjunction {  
 And = 'and',  
 Or = 'or'  
}

不同的字段可匹配的过滤操作符和匹配值不同，具体类型如下：

|  | IFilterAttachmentCondition | IFilterCheckboxCondition | IFilterAutoNumberCondition | IFilterDateTimeCondition | IFilterCreatedTimeCondition | IFilterModifiedTimeCondition | IFilterUserCondition | IFilterCreatedUserCondition | IFilterModifiedUserCondition | IFilterDuplexLinkCondition | IFilterSingleLinkCondition | IFilterFormulaCondition | IFilterGroupChatCondition | IFilterLocationCondition | IFilterLookupCondition | IFilterMultiSelectCondition | IFilterSingleSelectCondition | IFilterPhoneCondition | IFilterTextCondition | IFilterNumberCondition | IFilterUrlCondition | IFilterCurrencyCondition | IFilterBarcodeCondition | IFilterProgressCondition | IFilterRatingCondition |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **operator** | FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | FilterOperator.Is | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.IsGreater \| FilterOperator.IsGreaterEqual \| FilterOperator.IsLess \| FilterOperator.IsLessEqual \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty; | FilterOperator.Is \| FilterOperator.IsGreater \| FilterOperator.IsLess \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | FilterOperator.Is \| FilterOperator.IsGreater \| FilterOperator.IsLess \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | FilterOperator.Is \| FilterOperator.IsGreater \| FilterOperator.IsLess \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | BaseFilterOperator | BaseFilterOperator | BaseFilterOperator | BaseFilterOperator | BaseFilterOperator | FilterOperator | BaseFilterOperator | BaseFilterOperator | FilterOperator | BaseFilterOperator | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.Contains \| FilterOperator.DoesNotContain \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | BaseFilterOperator | BaseFilterOperator | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.IsGreater \| FilterOperator.IsGreaterEqual \| FilterOperator.IsLess \| FilterOperator.IsLessEqual \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | BaseFilterOperator | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.IsGreater \| FilterOperator.IsGreaterEqual \| FilterOperator.IsLess \| FilterOperator.IsLessEqual \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | BaseFilterOperator | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.IsGreater \| FilterOperator.IsGreaterEqual \| FilterOperator.IsLess \| FilterOperator.IsLessEqual \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty | FilterOperator.Is \| FilterOperator.IsNot \| FilterOperator.IsGreater \| FilterOperator.IsGreaterEqual \| FilterOperator.IsLess \| FilterOperator.IsLessEqual \| FilterOperator.IsEmpty \| FilterOperator.IsNotEmpty |
| **value** | null | boolean \| null | number \| null | IFilterDateTimeValue = number \| FilterDuration \| null | number \| FilterDuration \| null | number \| FilterDuration \| null | string[] \| null | string[] \| null | string[] \| null | string[] \| null | string[] \| null | IFilterAll | string[] \| null | string \| null | IFilterAll | string[] \| null \| string | string[] \| string | string \| null | string \| null | number \| null | string \| null | number \| null | string \| null | number \| null | number \| null |

FilterOperator 定义如下：

enum FilterOperator {  
 /\*\* 等于 \*/  
 Is = 'is',  
 /\*\* 不等于 \*/  
 IsNot = 'isNot',  
 /\*\* 包含 \*/  
 Contains = 'contains',  
 /\*\* 不包含 \*/  
 DoesNotContain = 'doesNotContain',  
 /\*\* 为空 \*/  
 IsEmpty = 'isEmpty',  
 /\*\* 不为空 \*/  
 IsNotEmpty = 'isNotEmpty',  
 /\*\* 大于 \*/  
 IsGreater = 'isGreater',  
 /\*\* 大于或等于 \*/  
 IsGreaterEqual = 'isGreaterEqual',  
 /\*\* 小于 \*/  
 IsLess = 'isLess',  
 /\*\* 小于或等于 \*/  
 IsLessEqual = 'isLessEqual'  
}

FilterDuration 定义如下

enum FilterDuration {  
 /\*\* 今天 \*/  
 Today = "Today",  
 /\*\* 明天 \*/  
 Tomorrow = "Tomorrow",  
 /\*\* 昨天 \*/  
 Yesterday = "Yesterday",  
 /\*\* 过去7天 \*/  
 TheLastWeek = "TheLastWeek",  
 /\*\* 未来7天 \*/  
 TheNextWeek = "TheNextWeek",  
 /\*\* 过去30天 \*/  
 TheLastMonth = "TheLastMonth",  
 /\*\* 未来30天 \*/  
 TheNextMonth = "TheNextMonth",  
 /\*\* 本周 \*/  
 CurrentWeek = "CurrentWeek",  
 /\*\* 上周 \*/  
 LastWeek = "LastWeek",  
 /\*\* 本月 \*/  
 CurrentMonth = "CurrentMonth",  
 /\*\* 上个月 \*/  
 LastMonth = "LastMonth"  
}

BaseFilterOperator 定义如下：

type BaseFilterOperator =  
 FilterOperator.Is  
 | FilterOperator.IsNot  
 | FilterOperator.Contains  
 | FilterOperator.DoesNotContain  
 | FilterOperator.IsEmpty  
 | FilterOperator.IsNotEmpty;

### 排序

视图按照一定的规则将数据表中的记录进行排序，一条排序规则由 字段ID + 顺序 组成：

interface ISortInfo {  
 fieldId: string;  
 /\*\* false: 正序 A -> Z; true: 倒序 Z -> A \*/  
 desc: boolean;  
}

### 分组

视图按照一定的规则将数据表中的记录进行分组，一条分组规则由 字段ID + 顺序 组成：

interface IGroupInfo {  
 fieldId: string;  
 /\*\* false: 正序 A -> Z; true: 倒序 Z -> A \*/  
 desc: boolean;  
}

### 同步配置

View 模块中分组/筛选/排序等能力，在调用写入 API 之后如果**希望保存或者同步给其他用户**需要调用 applySetting 方法

### 分享

SDK 支持开启/关闭指定视图分享，并获取处在开启状态视图的分享链接。

#### enableSharing

开启视图分享。

enableSharing: () => Promise<boolean>;

#### enableSharing

关闭视图分享。

disableSharing: () => Promise<boolean>;

#### getSharingStatus

获取当前视图分享状态。

getSharingStatus: () => Promise<SharingStatus>;  
  
enum SharingStatus {  
 Enabled = 'Enabled',  
 Disabled = 'Disabled'  
}

#### getSharingStatus

获取当前视图分享链接。

获取分享链接的前置条件是视图分享状态是开启的。

getShareLink(): Promise<string>;

# Base 模块

当前多维表格实例，多维表格相关 API 的主入口。

import { bitable } from '@lark-base-open/js-sdk';  
  
const base = bitable.base;

## 读接口

### isEditable

是否可以编辑当前多维表格。

isEditable(): Promise<boolean>;

##### 示例

const isEditable = await base.isEditable();

### getSelection

获取当前多维表格激活的相关信息（当前文档 id、数据表 id、视图 id 等）。

getSelection: () => Promise<Selection>;  
  
interface Selection {  
 baseId: string | null,   
 tableId: string | null,  
 fieldId: string | null,  
 viewId: string | null,   
 recordId: string | null  
}

#### 示例

const selection = await bitable.base.getSelection();

### getActiveTable

获取当前选中的数据表 table。

getActiveTable: () => Promise<ITable>;

#### 示例

const table = await base.getActiveTable();

### getTable

获取指定数据表 table，支持传入 table 的 id 或名称。

getTable(idOrName: string): Promise<ITable>

#### 示例

// 传入 table id  
const table = await base.getTable('t\_idxxxx');  
// 传入 table name  
const table = await base.getTable('Table\_For\_Test');

### getTableById

通过数据表 id 来获取指定数据表。

getTableById(tableId: string): ITable;

#### 示例

const table = await base.getTableById('t\_idxxxx');

### getTableByName

通过数据表名称获取指定数据表。

getTableByName(name: string): Promise<ITable>

#### 示例

const table = await base.getTableByName('Table\_For\_Test');

### getTableList

获取当前多维表格下所有的数据表。

getTableList(): Promise<ITable[]>;

#### 示例

const tableList = await base.getTableList();

### getTableMetaById

通过 id 获取指定数据表的元信息。

getTableMetaById(tableId: string): Promise<ITableMeta>

##### 示例

const tableMeta = await base.getTableMetaById('t\_id');

### getTableMetaList

获取当前多维表格下所有数据表元信息。

getTableMetaList(): Promise<ITableMeta[]>

##### 示例

const tableMetaList = await base.getTableMetaList();

### getPermission

获取 Base、Table、Field、Record、Cell 等不同实体的权限，当返回 true 的时候表示有权限，返回为 false 的时候没有权限。

getPermission(params: GetPermissionParams): Promise<boolean>;

其中 GetPermissionParams 的类型定义如下：

type GetPermissionParams = BasePermissionParams | TablePermissionParams | RecordPermissionParams | FieldPermissionParams | CellPermissionParams;  
  
interface BasePermissionParams {  
 entity: PermissionEntity.Base;  
 type: BaseOperation;  
}  
  
interface TablePermissionParams {  
 entity: PermissionEntity.Table;  
 param: {  
 tableId?: string;  
 };  
 type: TableOperation;  
}  
  
interface RecordPermissionParams {  
 entity: PermissionEntity.Record;  
 param: {  
 tableId: string;  
 recordId?: string;  
 };  
 type: RecordOperation;  
}  
  
interface FieldPermissionParams {  
 entity: PermissionEntity.Field;  
 param: {  
 tableId: string;  
 fieldId?: string;  
 };  
 type: FieldOperation;  
}  
  
interface CellPermissionParams {  
 entity: PermissionEntity.Cell;  
 param: {  
 tableId: string;  
 recordId?: string;  
 fieldId?: string;  
 };  
 type: CellOperation;  
}

#### 示例

使用的时候，需要传入对应的配置，来查询对应的权限, 下面展示一个查询字段编辑权限的例子：

import { PermissionEntity, OperationType } from '@lark-base-open/js-sdk';  
  
const fieldInfo: FieldPermissionParams = {  
 entity: PermissionEntity.Field,  
 params: {  
 tableId,  
 fieldId,  
 },  
 type: OperationType.Editable,  
}  
const hasPermission = await base.getPermission(params);

## 写接口

### addTable

添加数据表，支持设置表名和字段，返回创建成功数据表的 id 和索引位置。 :::warning 暂不支持配置字段，可以创建数据表后通过 [Table 模块](./table.md)中的方法添加字段。 :::

addTable(config: IAddTableConfig): Promise<{ tableId: string, index: number }>;  
  
interface IAddTableConfig {  
 name: string; // 表名  
}

#### 示例

const { tableId, index } = await bitable.base.addTable({  
 name: '测试添加数据表'  
})

### setTable

修改数据表，目前仅支持修改数据表的名称，修改成功会返回当前被修改数据表 id。

setTable(tableId: string, config: ISetTableConfig): Promise<string>;  
  
interface ISetTableConfig {  
 name: string; // 表名  
}

#### 示例

const tableId = await bitable.base.setTable({  
 name: '修改数据表'  
})

### deleteTable

删除指定数据表。

deleteTable(tableId: string): Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
  
await bitable.base.deleteTable(table.id);

### batchUploadFile

批量上传文件，按序返回每个文件对应的 fileToken 列表，支持传入 [File](https://developer.mozilla.org/zh-CN/docs/Web/API/File) 数组或 [FileList](https://developer.mozilla.org/zh-CN/docs/Web/API/FileList) 对象。

batchUploadFile(file: File[] | FileList): Promise<string[]>;

#### 示例

// 文件上传限制  
// file name 长度不得大于 250  
// file size 不得大于 1024 \* 1024 \* 1024 \* 2  
  
const file = new File(['Hello, World!'], 'hello.txt', { type: 'text/plain' });  
const tokens =await bitable.base.batchUploadFile([file]); // 拿到的 token 可以用于设置附件字段  
console.log(tokens) // ['BcdqbMmW4ohD7ExUq9rcGtuVn8e']

## 事件

### onTableAdd

监听 Table 添加事件，将返回一个取消监听函数。

onTableAdd(callback: () => void): () => void;

#### 示例

const off = bitable.base.onTableAdd((event) => {  
 console.log('table added')  
})

### onTableDelete

监听 Table 删除事件，将返回一个取消监听函数。

onTableDelete(callback: () => void): () => void;

#### 示例

const off = bitable.base.onTableDelete((event) => {  
 console.log('table deleted')  
})

### onSelectionChange

监听当前选中（数据表、单元格、视图）改变事件，将返回一个取消监听函数。

onSelectionChange(callback: () => void): () => void;

#### 示例

const off = bitable.base.onSelectionChange((event: { data: Selection }) => {  
 console.log('current selection', event)  
})

# Table 模块

Table 即数据表，可以将数据表理解成一个数据源，它负责维护数据与数据之间的联系，并不涉及 UI 展示(如字段顺序、记录顺序等，这些顺序信息保存在 [View 模块](./view.md)中)。

通过 Base 获取到 Table 之后，就可以调用 Table 中的 API，可以通过 getActiveTable 方法来获取当前选中的数据表实例：

const table = await bitable.base.getActiveTable()

当然也可以通过数据表 id 或名称来获取指定的数据表实例：

const table = await bitable.base.getTable(tableId/tableName)

## id

数据表 id。

id: string;

## getName

获取数据表名。

getName: () => Promise<string>;

#### 示例

const name = await table.getName();

## getMeta

获取数据表元数据。

getMeta: () => Promise<ITableMeta>;  
  
interface ITableMeta {  
 id: string;  
 name: string  
 isSync: boolean; // 是否同步表  
}

#### 示例

const meta = await table.getMeta();

## 获取字段

### isFieldExist

判断指定字段 id 判断字段是否存在。

isFieldExist(fieldId: string): Promise<boolean>;

#### 示例

const isExist = await table.isFieldExist('fieldId');

### getField

getField: <T extends IField>(idOrName: string) => Promise<T>;

通过 id 或 name 来获取对应的 Field(字段)，建议传入 Field 类型(例如示例中的 <IAttachmentField>)，来获得更好的语法提示。

#### 示例

const field = await table.getField<IAttachmentField>(idOrName);

### getFieldById

通过 id 来获取对应的 Field(字段)，建议传入 Field 类型(例如示例中的 <IAttachmentField>)，来获得更好的语法提示。

getFieldById: <T extends IField>(id: string) => Promise<T>;

#### 示例

const field = await table.getFieldById<IAttachmentField>(idOrName);

### getFieldByName

通过 name 来获取对应的 Field(字段)，建议传入 Field 类型(例如示例中的 <IAttachmentField>)，来获得更好的语法提示

getFieldByName: <T extends IField>(name: string) => Promise<T>;

#### 示例

const field = await table.getFieldByName<IAttachmentField>(idOrName);

### getFieldList

获取当前 table 下所有的字段列表。

getFieldList: <T extends IField>() => Promise<T[]>;

#### 示例

const fieldList = await table.getFieldList();

### getFieldIdList

获取字段 id 列表。 :::warning 通过该方法获取的字段 id 列表是**无序的**，因为 table 不涉及 UI 展示层面的信息，如果需要获取有序的字段 id 列表，需要在 View 模块 调用 view.getVisibleFieldIdList 来获取有序的字段 id 列表 :::

getFieldIdList(): Promise<string[]>;

#### 示例

const fieldIdList = await table.getFieldIdList();

### getFieldMetaById

通过 id 获取对应的字段元信息。

getFieldMetaById(fieldId: string): Promise<IFieldMeta>;  
  
interface IFieldMeta {  
 id: string;  
 type: FieldType;  
 name: string;  
 isPrimary: boolean;  
 description: IBaseFieldDescription;  
}

#### 示例

const fieldMeta = await table.getFieldMetaById('f\_id');  
// { id: 'f\_id', name: 'text field', type: 1, isPrimary: true, description: { content: content, disableSyncToFormDesc: false } }

### getFieldMetaList

获取所有字段元信息。

通过该方法获取的字段 meta 列表是**无序的**，因为 table 不涉及 UI 展示层面的信息，如果需要获取有序的字段 meta 列表，需要在 View 模块 调用 view.getFieldMetaList 来获取有序的字段 id 列表

getFieldMetaList(): Promise<IFieldMeta[]>;  
  
interface IFieldMeta {  
 id: string;  
 type: FieldType;  
 property: IFieldProperty;  
 name: string;  
 isPrimary: boolean;  
 description: IBaseFieldDescription;  
}

#### 示例

const fieldMetaList = await table.getFieldMetaList();

### getFieldListByType

获取当前数据表下所有指定字段类型的字段列表。

getFieldListByType: <T extends IField>(type: FieldType) => Promise<T[]>;

#### 示例

// 获取 table 下所有的附件字段  
const attachmentFieldList = await table.getFieldListByType<IAttachmentField>(FieldType.Attachment);

### getFieldMetaListByType

获取当前数据表下所有指定字段类型的字段元信息列表。

getFieldMetaListByType: <T extends IFieldMeta>(type: FieldType) => Promise<T[]>;  
  
interface IFieldMeta {  
 id: string;  
 type: FieldType;  
 name: string;  
 property: IFieldProperty;  
 isPrimary: boolean;  
 description: IBaseFieldDescription;  
}

#### 示例

// 获取 table 下所有的附件字段的 Meta 列表  
const attachmentFieldMetaList = await table.getFieldMetaListByType<IAttachmentFieldMeta>(FieldType.Attachment)

## 新增字段

### addField

新增字段，并返回对对应的字段 id。

addField 支持直接配置字段属性，但推荐在新建字段之后通过对应的字段方法修改字段属性，更简便不易出错。

addField: (fieldConfig: IAddFieldConfig) => Promise<FieldId>;  
  
type IAddFieldConfig = {  
 type: FieldType;  
 property?: FieldProperty;  
 name?: string;  
 description?: { // 字段描述  
 content?: string;  
 /\*\* 是否禁止同步，如果为true，表示禁止同步该描述内容到表单的问题描述（只在新增、修改字段时生效）; 默认false \*/  
 disableSyncToFormDesc?: boolean;  
 };  
}  
  
type FieldId = string;

#### 示例

const field = await table.addField({ type: FieldType.SingleSelect });  
const singleSelectField = await table.getField<ISingleSelectField>(field);  
await singleSelectField.addOption('Option1');

如上所示的例子，我们先新增了一个单选字段，然后再在这个字段上新增了一个选项（推荐在获取字段的时候，指定对应的类型（如 <ISingleSelectField>） ，以获得更好的语法提示）

### onFieldAdd

监听 Field 添加事件，返回一个取消监听函数。

onFieldAdd(callback: (ev: IEventCbCtx) => void): () => void;

#### 示例

const off = table.onFieldAdd((event) => {  
 console.log('event:', event);  
})  
const fieldId = await table.addField({ // 新增一个多行文本类型的字段  
 type: FieldType.Text,  
 name: 'field\_test'  
})

## 修改字段

### setField

修改字段，如字段类型、字段名称和字段属性等。 :::tip 推荐从 Field 实例调用对应的字段方法来修改字段属性，更简便不易出错。 :::

setField(fieldId: string, fieldConfig: ISetFieldConfig): Promise<IFieldRes>;  
  
// 支持不传 name、type、property 等参数，不传参数时会合并原属性  
type ISetFieldConfig = {  
 type: FieldType;  
 property?: FieldProperty;  
 name?: string;  
 description?: { // 字段描述  
 content?: string;  
 /\*\* 是否禁止同步，如果为true，表示禁止同步该描述内容到表单的问题描述（只在新增、修改字段时生效）; 默认false \*/  
 disableSyncToFormDesc?: boolean;  
 };  
}

#### 示例

const field = await table.getField('f\_idxxx');  
  
const res = await table.setField(field.id, {  
 name: 'modify\_field\_name'  
})

### onFieldModify

监听字段修改事件，返回一个取消监听函数。

onFieldModify(callback: (ev: IEventCbCtx) => void): () => void;

#### 示例

const off = table.onFieldModify((event) => {  
 console.log('field modify:', event);  
})  
  
const fieldId = await table.addField({ // 新增一个多行文本类型的字段  
 type: FieldType.Text,  
 name: 'field\_test'  
})  
  
const fieldId = await table.setField({ // 修改字段名称  
 name: 'field\_modified'  
})

## 删除字段

### deleteField

删除指定字段。

deleteField: (fieldOrId: string | IField) => Promise<boolean>;

#### 示例

const attachmentField = await table.addField({ FieldType.Attachment });  
// 直接传递 field 实例  
await table.deleteField(attachmentField)  
// 或者传递 fieldId  
await table.deleteField(attachmentField.id);

### onFieldDelete

监听 Field 删除事件，返回一个取消监听函数。

onFieldDelete(callback: (ev: IEventCbCtx) => void): () => void;

#### 示例

const off = table.onFieldDelete((event) => {  
 console.log('field delete', event);  
})  
  
const fieldId = await table.addField({ // 新增多行文本类型的字段  
 type: FieldType.Text,  
 name: 'field\_test'  
})  
  
table.deleteField(fieldId) // 删除字段

## 获取记录

### getCellValue

获取指定单元格的取值。

getCellValue(fieldId: string, recordId: string): Promise<IOpenCellValue>;

#### 示例

// 光标选中数据表中的单元格  
const { fieldId, recordId } = await bitable.base.getSelection();  
const cellValue = table.getCellValue(fieldId, recordId);

### getRecordById

通过指定 id 去获取对应记录。

批量获取场景下，建议使用 [getRecords](./table.md#getrecords) 方法以获得更好的性能体验

getRecordById(recordId: string): Promise<IRecordValue>;  
  
type IRecordValue = {  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
};

#### 示例

const recordIdList = await table.getRecordIdList(); // 获取 recordId 列表  
  
const recordValue = await table.getRecordById(recordIdList[0]);

### getRecords

批量获取 record 数据。

单次获取上限 **5000** 条。

getRecords({ pageSize, pageToken, viewId }: IGetRecordsParams): Promise<IGetRecordsResponse>;

| 名称 | 数据类型 | 是否必填 | 描述 |
| --- | --- | --- | --- |
| pageSize | number | 是 | 分页页面大小 size，最大值：5000 |
| pageToken | string | 否 | 分页标记，第一次请求不填，表示从头开始遍历；分页查询结果还有更多项时会同时返回新的 page\_token，下次遍历可采用该 page\_token 获取查询结果 |
| viewId | string | 否 | 视图的唯一标识符，获取指定视图下的记录 |

相关类型定义如下：

interface IGetRecordsParams {  
 pageSize?: number; // 获取数量，默认 5000，最大不得超过 5000  
 pageToken?: string; // 分页标记，第一次请求不填，表示从头开始遍历；分页查询结果还有更多项时会同时返回新的 pageToken，下次遍历可采用该 pageToken 获取查询结果  
 viewId?: string; // 获取指定视图的 record  
}  
  
interface IGetRecordsResponse {  
 total: number; // 记录总数  
 hasMore: boolean; // 是否还有更多记录  
 records: IRecord[]; // 记录列表  
 pageToken?: string; // 分页标记  
}  
  
interface IRecord {  
 recordId: string;  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
}

#### 示例

// 首先使用 getActiveTable 方法获取了当前用户选择的 table（用户当前编辑的数据表）  
const table = await bitable.base.getActiveTable();  
const records = await table.getRecords({  
 pageSize: 5000  
})

### getRecordIdList

获取所有记录 id 列表。

通过该方法获取的记录 id 列表是**无序的**，因为 table 不涉及 UI 展示层面的信息，如果需要获取有序的字段 id 列表，需要在 View 模块 调用 view.getVisibleRecordIdList 来获取有序的记录 id 列表

getRecordIdList(): Promise<string[]>;

#### 示例

const recordIdList = await table.getRecordIdList();

### getRecordList

获取当前的记录列表，Record 模块中的相关方法可以查看 [Record 模块](./record.md)

getRecordsList(): Promise<Record>;

const recordList = await table.getRecordList();  
for (const record of recordList) {  
 const cell = await record.getCellByField(fieldId);  
 const val = await cell.getValue();  
}

### getCellAttachmentUrls

批量获取指定附件单元格中附件的 URL，参数中的 token 需要从附件字段所属的单元格中获取。(推荐通过 [AttachmentField](./field/attachment.md) 模块去获取)

接口返回的临时链接的有效时间是 10 分钟

getCellAttachmentUrls(tokens: string[], fieldId: string, recordId: string): Promise<string[]>;

#### 示例

const urls = await table.getCellAttachmentUrls(['token\_1', 'token\_2'], 'f\_id', 'r\_id');

### getRecordShareLink

获取指定记录的分享链接，获得链接的用户，将以多维表格的权限访问。

getRecordShareLink(recordId: string)

#### 示例

const recordShareLink = await table.getRecordShareLink('r\_Id')

## 新增记录

### addRecord

新增一条记录，新增成功后返回 recordId，支持直接传递 RecordValue 或单元格 Cell 实例。 :::tip 批量新增场景下，建议使用 [addRecords](./table.md#addrecords) 方法以获得更好的性能体验 :::

addRecord: (recordVale?: IRecordValue | ICell | ICell[]) => Promise<IRecordRes>;  
  
type IRecordValue = {  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
};  
  
type IRecordRes = string;

#### 示例

如果使用 RecordValue 来创建（不推荐）：

const field = await table.getField('多行文本'); // 选择某个多行文本字段  
  
const res = await table.addRecord({  
 fields: {  
 [field.id]: 'new text field value'  
 }  
});  
// 'recxxx' 新增的记录 id

更推荐通过组合 Cell 实例来插入一条记录，[Cell](./cell.md) 可以通过各个字段的 createCell 方法来创建，下面是一个例子:

const textField = await table.getField<TextField>('多行文本');  
  
const textCell = await textField.createCell('new text field value');  
const recordId = await table.addRecord(textCell);

### addRecords

新增多条记录，新增成功后返回 recordId 列表。

单次新增记录上限 **5000** 条

addRecords: (record?: IRecordValue[] | ICell[] | Array<ICell[]>) => Promise<IRecordRes[]>;  
  
type IRecordValue = {  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
};  
  
type IRecordRes = string;

#### 示例

如果使用 RecordValue 来创建（不推荐）：

const field = await table.getField('多行文本'); // 选择某个多行文本字段  
  
const res = await table.addRecords([  
 {  
 fields: {  
 [field.id]: 'new text field value1'  
 }  
 },  
 {  
 fields: {  
 [field.id]: 'new text field value2'  
 }  
 },  
]);

更推荐通过组合 Cell 实例来插入多条记录，[Cell](./cell.md) 可以通过各个字段的 createCell 方法来创建，下面是一个例子:

const textField = await table.getField('多行文本'); // 选择某个多行文本字段  
  
const textCell1 = await textField.createCell('new text field value1');  
const textCell2 = await textField.createCell('new text field value1');  
const recordIds = await table.addRecords([[textCell1],[textCell2]]);

### onRecordAdd

监听 Record 添加事件，返回一个取消监听方法。

onRecordAdd(callback: (ev: IEventCbCtx<[recordId: string]>) => void): () => void;

#### 示例

const field = await table.getField('多行文本'); // 根据字段名获取多行文本类型的字段  
const off = table.onRecordAdd((event) => { // 监听字段增加事件  
 console.log('record add', event);  
});  
  
const cell = field.createCell('new text field value');  
table.addRecord(cell);

## 修改记录

### setCellValue

修改指定单元格的值。(推荐通过 Field 来修改)

批量修改场景下，建议使用 [setRecords](./table.md#setrecords) 方法以获得更好的性能体验

setCellValue<T extends IOpenCellValue = IOpenCellValue>(fieldId: string, recordId: string, cellValue: T): Promise<boolean>;

#### 示例

const recordIds = await table.getRecordIdList();  
const field = await table.getField('多行文本');  
  
// 修改某个多行文本类型的字段  
const res = await table.setCellValue(field.id, recordIds[0], 'test setCellValue')  
// true

### setRecord

修改指定记录数据。

批量修改场景下，建议使用 [setRecords](./table.md#setrecords) 方法以获得更好的性能体验

setRecord(recordId: string, recordValue?: IRecordValue): Promise<IRecordRes>;  
  
type IRecordValue = {  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
};

#### 示例

const recordIds = await table.getRecordIdList(); // 获取所有记录 id  
const field = await table.getField('多行文本'); // 选择多行文本字段  
  
const res = await table.setRecord(recordIds[0], {  
 fields: {  
 [field.id]: 'test setRecord'  
 }  
})

### setRecords

批量修改记录数据。

单次修改记录上限 **5000** 条

setRecords(records?: IRecord[]): Promise<IRecordRes[]>;  
  
interface IRecord {  
 recordId: string;  
 fields: {  
 [fieldId: string]: IOpenCellValue;  
 };  
}

#### 示例

const recordIds = await table.getRecordIdList(); // 获取所有记录 id  
const field = await table.getField('多行文本'); // 选择多行文本字段  
  
await table.setRecords([  
 {  
 recordId: recordIds[0],  
 fields: {  
 [field.id]: 'test setRecords1'  
 }  
 },  
 {  
 recordId: recordIds[1],  
 fields: {  
 [field.id]: 'test setRecords2'  
 }  
 }  
])

### getCellString

获取单元格取值的**原始字符串形式**，如日期字段会返回具体的年月日字符串。

getCellString(fieldId: string, recordId: string): Promise<string>;

#### 示例

const recordIds = await table.getRecordIdList();  
const dateTimeField = await table.getField('日期');  
  
const res = await table.getCellString(dateTimeField.id, recordIds[0]);   
// 2023/10/01

### onRecordModify

监听 Record 修改事件，返回一个取消监听方法。如果记录修改前后并未发生变化，则不会触发回调函数。

onRecordModify(callback: (ev: IEventCbCtx<{  
 recordId: string;  
 fieldIds: string[];  
}>) => void): () => void;

#### 示例

const recordIds = await table.getRecordIdList();  
const field = await table.getFieldByName('多行文本')  
  
const off = table.onRecordModify((event) => { // 监听记录修改事件  
 console.log('record modify', event);  
})  
  
await table.setRecord(recordIds[0], { // 修改某条记录的多行文本字段  
 fields:{  
 [field.id]: 'modify value'  
 }  
})

## 删除记录

### deleteRecord

删除指定记录。 :::tip 批量删除场景下，建议使用 [deleteRecords](./table.md#deleterecords) 方法以获得更好的性能体验 :::

deleteRecord(recordId: string): Promise<boolean>;

#### 示例

const recordIdList = await table.getRecordIdList();  
  
await table.deleteRecord(recordIdList[0]);

### deleteRecords

批量删除记录。

单次删除记录上限 **5000** 条

deleteRecords(recordIdList: string[]): Promise<boolean>;

#### 示例

const recordIdList = await table.getRecordIdList();  
  
// 删除前100条记录  
await table.deleteRecords(recordIdList.slice(0, 100));

### onRecordDelete

监听 Record 删除事件，返回一个取消监听方法。

onRecordDelete(callback: (ev: IEventCbCtx<[recordId: string]>) => void): () => void;

#### 示例

const off = table.onRecordDelete((event) => {  
 console.log('record delete', event);  
})  
  
const recordIdList = await table.getRecordIdList();  
table.deleteRecord(recordIdList[0]);

## 获取视图

View 模块相关能力请参考 [视图模块](./view.md)。

### getActiveView

This method is under testing, please use the 0.3.5-alpha.4 version package for test

获取当前选择的 View 视图。

getActiveView: () => Promise<IView>;

#### 示例

const view = await table.getActiveView();

### isViewExist

通过 viewId 判断视图是否存在。

isViewExist(viewId: string): Promise<boolean>;

#### 示例

const isExist = await table.isViewExist('viewId');

### getViewById

通过 id 来获取 View 视图。

getViewById: (id: string) => Promise<IView>;

#### 示例

const view = await table.getViewById(viewId);

### getViewList

获取当前数据表的所有视图。

getViewList: () => Promise<IView[]>;

#### 示例

const viewList = await table.getViewList();

### getViewMetaById

通过 id 获取视图的元信息。

getViewMetaById(viewId: string): Promise<IViewMeta>;  
  
interface IViewMeta {  
 id: string;  
 name: string;  
 type: ViewType;  
 property: IViewProperty;  
}

#### 示例

const viewMeta = await table.getViewById(viewId);

### getViewMetaList

获取当前数据表下所有的视图元信息。

getViewMetaList(): Promise<IViewMeta[]>;  
  
interface IViewMeta {  
 id: string;  
 name: string;  
 type: ViewType;  
 property: IViewProperty;  
}

#### 示例

const viewMetaList = await table.getViewMetaList();

## 新增视图

### addView

给当前数据表添加视图。 ::: warning 目前仅支持设置 ViewType 和 name，推荐创建后通过 View 模块的 API 进行视图配置。 :::

addView(config: IAddViewConfig): Promise<IAddViewResult>;  
  
interface IAddViewConfig {  
 name?: string;  
 type: ViewType;  
}  
  
interface IAddViewResult {  
 viewId: string;  
 index: number; // 视图顺序  
}

#### 示例

await table.addView({ type: ViewType.Grid, name: 'test'});

## 修改视图

### setView

修改指定视图信息。 ::: warning 目前仅支持设置 ViewType 和 name，推荐创建后通过 View 模块的 API 进行视图配置。 :::

setView(viewId: string, config: ISetViewConfig): Promise<ViewId>;  
  
interface ISetViewConfig {  
 name?: string;  
}

#### 示例

await table.setView('v\_id', { name: 'modified name'});

## 删除视图

### deleteView

删除指定视图。

deleteView(viewId: string): Promise<boolean>;

#### 示例

await table.deleteView('v\_id');

# TextField 文本字段

类型定义 ITextField，使用方法示例：

const textField = await table.getField<ITextField>(fieldId);

文本字段可以承载多个段落 Segment，段落 Segment 存在多种不同类型，目前支持文本类型、URL 类型、人员类型和文档类型。相关数据类型定义：

type IOpenTextFieldValue = IOpenSegment[];  
  
type IOpenSegment = IOpenTextSegment | IOpenUrlSegment | IOpenUserMentionSegment | IOpenDocumentMentionSegment;  
  
/\*\* 文本类型 \*/  
interface IOpenTextSegment {  
 type: IOpenSegmentType.Text;   
 text: string   
};  
  
/\*\* URL 类型 \*/  
interface IOpenUrlSegment {  
 type: IOpenSegmentType.Url;  
 text: string;  
 link: string;  
};  
  
/\*\* 人员类型 \*/  
interface IOpenUserMentionSegment {  
 mentionType: 'User';  
 text: string;  
 token: string;  
 /\*\* 用户名 \*/  
 name: string;  
 enName?: string;  
 /\*\* 用户 id \*/  
 id: string;  
}  
  
/\*\* 文档类型 \*/  
interface IOpenDocumentMentionSegment {  
 mentionType: 'Doc' | 'Sheet' | 'Bitable' | '...';   
 link: string;  
 text: string;  
 token: string;  
}

## createCell

创建一个文本字段的 Cell，**对于文本和 URL 类型的段落，支持直接传递字符串**。

createCell: (val: TextFieldTransformVal) => Promise<ICell>;

#### 示例

// text segment  
await textField.createCell('test');  
// URL segment  
await textField.createCell('https://www.feishu.cn');

## getCell

获取指定记录对应的 Cell 单元格。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordList = await table.getRecordList();  
  
const cell = await textField.getCell(recordList[0]);

## setValue

设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: TextFieldTransformVal) => Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
await textField.setValue(recordIdList[0], 'modify value');

## getValue

获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenSegment[]>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
const cellValue = await textField.getValue(recordIdList[0]);

# Autonumber 自动编号字段

AutoNumber 自动编号字段**不支持手动往单元格写入值**。

类型定义 IAutonumberField，使用方法示例：

const autonumberField = await table.getField<IAutonumberField>(fieldId);

其中字段值的类型定义为：

export type IOpenAutoNumber = string;

## getValue

获取指定单元格自动编号的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenAutoNumber>;

### 示例

const val = await autonumberField.getValue(recordId);

## getCell

通过对应的 记录(Record) 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

### 示例

const autonumberField = await table.getField<IAutonumberField>(fieldId);  
const cell = await autonumberField.getCell(recordId);

# Email 邮箱字段

类型定义 IEmailField，使用方法示例：

const emailField = await table.getField<IEmailField>(fieldId);

其中字段值的类型定义为：

type IOpenEmail = string;

## createCell

创建一个邮箱字段的 Cell，**此处写入的字符串需要满足邮箱格式**。

createCell: (val: string) => Promise<ICell>;

#### 示例

await emailField.createCell('test@gmail.com');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await emailField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值，**此处写入的字符串需要满足邮箱格式**。

setValue: (recordOrId: IRecordType | string, val: string) => Promise<boolean>;

#### 示例

await emailField.setValue('r\_id', 'test@gmail.com');

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenEmail>;

#### 示例

await emailField.getValue('r\_id');

# DuplexLink 双向关联字段

类型定义 IDuplexLinkField，使用方法示例：

const duplexLinkField = await table.getField<IDuplexLinkField>(fieldId);

其中字段值的类型定义为：

type IOpenLink = {  
 recordIds: string[]; // 被关联的记录 id 列表  
 text: string; // 被关联记录的文本汇总值，用 , 连接  
 tableId: string; // 被关联表的 id  
};

## createCell

创建一个双向关联字段的 Cell。

createCell: (val: Partial<IOpenLink>) => Promise<ICell>;

#### 示例

await duplexLinkField.createCell({  
 recordIds: ['r\_id1', 'r\_id2'], // 关联字段配置的数据表下的记录 id 列表  
});

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await duplexLinkField.getCell('r\_id');

## setValue

通过 Record 来设置对应单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenLink) => Promise<boolean>;

#### 示例

await duplexLinkField.setValue({  
 recordIds: ['r\_id1', 'r\_id2'], // 关联字段配置的数据表下的记录 id 列表  
});

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenLink>;

#### 示例

await duplexLinkField.getValue('r\_id');

## setTableId

设置关联的数据表 Table。

setTableId: (tableId: string) => Promise<IFieldRes>;

#### 示例

await duplexLinkField.setTableId('t\_id');

## getTableId

获取关联的数据表 id。

getTableId: () => Promise<string>;

#### 示例

await duplexLinkField.getTableId();

## setMultiple

设置是否允许关联多条记录。

setMultiple: (multiple: boolean) => Promise<IFieldRes>;

#### 示例

await duplexLinkField.setMultiple(true);

## getMultiple

获取是否允许关联多条记录。

getMultiple: () => Promise<boolean>;

#### 示例

await duplexLinkField.getMultiple();

# Barcode 条码字段

类型定义 IBarcodeField，使用方法示例：

const barcodeField = await table.getField<IBarcodeField>(fieldId);

条码字段与文本字段的类型定义完全相同，详情可参考[文本字段](./text.md)。

## createCell

创建一个条码字段的 Cell，支持直接传入字符串。

createCell: (val: BarcodeTransformVal) => Promise<ICell>;

### 示例

const cell = await barcodeField.createCell('barcode');  
await table.addRecord(cell);

## getCell

通过对应的记录 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

### 示例

const cell = await barcodeField.getCell(recordId);

## setValue

通过对应的记录 Record 来设置对应的值，支持直接传入字符串。

setValue: (recordOrId: IRecordType | string, val: BarcodeTransformVal) => Promise<boolean>;

### 示例

await barcodeField.setValue('r\_id', 'barcode');

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenSegment[]>;

### 示例

const val = await barcodeField.getValue(recordId);

## setOnlyMobile

设置是否仅可通过移动端扫码录入, 为 true 时表示只运行移动端扫码录入。

setOnlyMobile: (onlyMobile: boolean) => Promise<boolean>;

### 示例

await barcodeField.setOnlyMobile(true);

## getOnlyMobile

获取是否仅可通过移动端扫码录入, 为 true 时表示只运行移动端扫码录入。

getOnlyMobile: () => Promise<boolean>;

### 示例

await barcodeField.getOnlyMobile();

# Group 群组字段

类型定义 IGroupField，使用方法示例：

const groupField = await table.getField<IGroupField>(fieldId);

其中字段值的类型定义为：

type IOpenGroupChat = {  
 id: string;  
 name: string;  
 avatarUrl: string; // 群头像  
 enName?: string;  
 linkToken?: string; // 群链接 token  
};

## createCell

创建一个群组字段的 Cell。

createCell: (val: IOpenGroupChat[]) => Promise<ICell>;

#### 示例

await groupField.createCell([  
 {  
 id: 'og\_xxx',  
 name: 'group\_name'  
 }  
]);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await groupField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenGroupChat[]) => Promise<boolean>;

#### 示例

await groupField.setValue('r\_id', [  
 {  
 id: 'og\_xxx',  
 name: 'group\_name'  
 }  
]);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenGroupChat[]>;

#### 示例

await groupField.getValue('r\_id');

## setMultiple

设置是否可以多选。

setMultiple: (multiple: boolean) => Promise<IFieldRes>;

#### 示例

await groupField.setMultiple(true);

## getMultiple

获取是否可以多选。

getMultiple: () => Promise<boolean>;

#### 示例

await groupField.getMultiple();

# Location 地理位置字段

类型定义 ILocationField，使用方法示例：

const locationField = await table.getField<ILocationField>(fieldId);

其中字段值的类型定义为：

type IOpenLocation = {  
 address: string;  
 adname: string;  
 cityname: string;  
 /\*\* 简短地址 \*/  
 name: string;  
 /\*\* 省 \*/  
 pname: string;  
 /\*\* 完整地址 \*/  
 fullAddress: string;  
 /\*\* 经纬度，格式为 "「精度」，「维度」" \*/  
 location: string;  
};

## createCell

创建一个地理位置字段的 Cell，**仅传入经纬度信息即可，底层会解析出该经纬度对应的位置信息**。

createCell: (val: Partial<IOpenLocation>) => Promise<ICell>;

#### 示例

await locationField.createCell({  
 location: '112.946927,21.529146'  
});

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await locationField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值，**仅传入经纬度信息即可，底层会解析出该经纬度对应的位置信息**。

setValue: (recordOrId: IRecordType | string, val: IOpenLocation) => Promise<boolean>;

#### 示例

await locationField.setValue('r\_id', {  
 location: '112.946927,21.529146'  
});

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenLocation>;

#### 示例

await locationField.getValue('r\_id');

## setInputType

设置允许输入地址的方式。

setInputType: (inputType: ELocationInputType) => Promise<IFieldRes>;

其中 ELocationInputType 的类型定义如下：

enum ELocationInputType {  
 ONLY\_MOBILE = 'ONLY\_MOBILE', // 只允许手机定位  
 NOT\_LIMIT = 'NOT\_LIMIT', // 可获取任意位置  
}

#### 示例

await locationField.setInputType(ELocationInputType.ONLY\_MOBILE);

## getInputType

获取输入地址的方式。

getInputType: () => Promise<ELocationInputType>;

#### 示例

await locationField.getInputType();

# SingleLink 单向关联字段

类型定义 ISingleLinkField，使用方法示例：

const singleLinkField = await table.getField<ISingleLinkField>(fieldId);

其中字段值的类型定义为：

type IOpenLink = {  
 recordIds: string[]; // 被关联的记录 id 列表  
 text: string; // 被关联记录的文本汇总值，用 , 连接  
 tableId: string; // 被关联表的 id  
};

## createCell

创建一个双向关联字段的 Cell。

createCell: (val: Partial<IOpenLink>) => Promise<ICell>;

#### 示例

await singleLinkField.createCell({  
 recordIds: ['r\_id1', 'r\_id2'], // 关联字段配置的数据表下的记录 id 列表  
});

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await singleLinkField.getCell('r\_id');

## setValue

通过 Record 来设置对应单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenLink) => Promise<boolean>;

#### 示例

await singleLinkField.setValue({  
 recordIds: ['r\_id1', 'r\_id2'], // 关联字段配置的数据表下的记录 id 列表  
});

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenLink>;

#### 示例

await singleLinkField.getValue('r\_id');

## setTableId

设置关联的数据表 Table。

setTableId: (tableId: string) => Promise<IFieldRes>;

#### 示例

await singleLinkField.setTableId('t\_id');

## getTableId

获取关联的数据表 id。

getTableId: () => Promise<string>;

#### 示例

await singleLinkField.getTableId();

## setMultiple

设置是否允许关联多条记录。

setMultiple: (multiple: boolean) => Promise<IFieldRes>;

#### 示例

await singleLinkField.setMultiple(true);

## getMultiple

获取是否允许关联多条记录。

getMultiple: () => Promise<boolean>;

#### 示例

await singleLinkField.getMultiple();

# UrlField 超链接字段

类型定义 IUrlField，使用方法示例：

const urlField = await table.getField<IUrlField>(fieldId);

其中字段值的类型定义为：

type IOpenUrlSegment = {  
 type: IOpenSegmentType.Url;  
 text: string;  
 link: string;  
};  
  
export type UrlTransformVal = string | IOpenUrlSegment | IOpenUrlSegment[];

## createCell

创建一个链接字段的 Cell，开发者只需要输入文本就可以指定转化为指定格式，其中 URL 链接会转化为对应的超链接。

createCell: (val: UrlTransformVal) => Promise<ICell>;

#### 示例

const cell = await urlField.createCell('https://lark-base-team.github.io/js-sdk-docs');

## getCell

通过指定 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

const recordIdList = await table.getRecordIdList();  
  
const cell = await urlField.getCell(recordIdList[0]);

## setValue

通过 Record 来设置对应单元格 Cell 的值。

setValue: (recordOrId: IRecordType | string, val: UrlTransformVal) => Promise<boolean>;

#### 示例

const recordIdList = await table.getRecordIdList();  
  
await urlField.setValue(recordIdList[0], 'https://lark-base-team.github.io/js-sdk-docs');

## getValue

通过 Record 来获取对应单元格 Cell 的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenUrlSegment[]>;

#### 示例

const recordIdList = await table.getRecordIdList();  
  
await urlField.getValue(recordIdList[0]);

# Attachment 附件字段

类型定义 IAttachmentField，使用方法示例：

const attachmentField = await table.getField<IAttachmentField>(fieldId);

其中字段值的类型定义为：

type IOpenAttachment = {  
 name: string;  
 size: number;  
 type: string; // mime  
 token: string;  
 timeStamp: number;  
 /\*\* 高级权限下附件接口依赖的信息，可能为空 \*/  
 permission?: {  
 tableId: string;  
 recordId: string;  
 fieldId: string;  
 }  
};  
  
type AttachmentTransformVal = File | File[] | FileList | IOpenAttachment | IOpenAttachment[];

## getAttachmentUrls

该接口返回的临时链接的有效时间是 **10 分钟**。

通过 记录 Record (id 或者 Record 对象) 信息，获取附件的 URL 地址(URL 有效期为 10 分钟)。

getAttachmentUrls: (recordOrId: IRecordType | string) => Promise<string[]>;

#### 示例

const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const attachmentUrls = await attachmentField.getAttachmentUrls(recordId);

## setOnlyMobile

设置是否只允许移动端上传，传入为 true 的时候设置为仅允移动端上传文件。

setOnlyMobile: (onlyMobile: boolean) => Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
await attachmentField.setOnlyMobile(true);

## getOnlyMobile

获取是否只允许移动端上传的属性值。

getOnlyMobile: () => Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const isOnlyMobile = await attachmentField.getOnlyMobile();

## createCell

创建一个附件单元格，可以直接传入文件 File 来实现构造一个单元格。

createCell: (val: AttachmentTransformVal) => Promise<ICell>;

#### 示例

const file = new File(['text'], 'file\_name.txt', { type: "text/plain" });  
  
const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const attachmentCell = await attachmentField.createCell(file);  
const recordId = await table.addRecord(attachmentCell);

## getCell

获取一个附件单元格，可以传入记录(record)的 id 或者实例。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>

#### 示例

const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const attachmentCell = await attachmentField.getCell(recordId);

## setValue

通过 Record 设置 Value，支持直接传入文件 File。

setValue: (recordOrId: IRecordType | string, val: AttachmentTransformVal) => Promise<boolean>;

#### 示例

const file = new File(['text'], 'file\_name.txt', { type: "text/plain" });  
  
const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
await attachmentField.setValue(recordId, file);

## getValue

通过 Record 获取指定单元格所有的附件。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenAttachment[]>;

#### 示例

const table = await bitable.base.getActiveTable();  
const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const val = await attachmentField.getValue(recordId);

# Date 日期字段

类型定义 IDateTimeField，使用方法示例：

const dateTimeField = await table.getField<IDateTimeField>(fieldId);

其中字段值的类型定义为：

type IOpenTimestamp = number; // 以毫秒为单位的 Unix 时间戳

## createCell

创建一个日期字段的 Cell。

createCell: (val: IOpenTimestamp) => Promise<ICell>;

#### 示例

await dateTimeField.createCell(Date.now());

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await dateTimeField.getCell('r\_id');

## setValue

通过 Record 来设置对应单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenTimestamp) => Promise<boolean>;

#### 示例

await dateTimeField.setValue('r\_id', Date.now());

## getValue

通过 Record 来获取对应单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenTimestamp>;

#### 示例

await dateTimeField.getValue('r\_id');

## setDateFormat

设置字段日期格式。

setDateFormat: (format: DateFormatter) => Promise<IFieldRes>;

其中 DateFormatter 的数据格式为：

enum DateFormatter {  
 DATE\_YMD\_WITH\_SLASH = 'yyyy/MM/dd', // 2021/01/30  
 DATE\_TIME = 'yyyy/MM/dd HH:mm', // 2021/01/30 14:00  
 DATE\_TIME\_WITH\_HYPHEN = 'yyyy-MM-dd HH:mm', // 2021-01-30 14:00  
 DATE\_YMD\_WITH\_HYPHEN = 'yyyy-MM-dd', // 2021-01-30  
 DATE\_MD\_WITH\_HYPHEN = 'MM-dd', // 01-30  
 DATE\_MMDD\_WITH\_SLASH = 'MM/dd/yyyy', // 01/30/2021  
 DATE\_DDMM\_WITH\_SLASH = 'dd/MM/yyyy', // 30/01/2021  
}

#### 示例

await dateTimeField.setDateFormat(DateFormatter.DATE\_TIME);

## getDateFormat

获取日期格式。

getDateFormat: () => Promise<DateFormatter>;

#### 示例

await dateTimeField.getDateFormat();

# SingleSelect 单选字段

类型定义 ISingleSelectField，使用方法示例：

const singleSelectField = await table.getField<ISingleSelectField>(fieldId);

其中字段值的类型定义为：

// 单选值类型定义  
type IOpenSingleSelect = {  
 id: string; // 选项 id  
 text: string;  
};  
  
type SingleSelectTransformVal = string | IOpenSingleSelect;

## createCell

创建一个单选字段的 Cell。

createCell: (val: SingleSelectTransformVal) => Promise<ICell>;

#### 示例

await singleSelectField.createCell('test option');

## getCell

获取指定记录对应的 Cell 单元格。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordList = await table.getRecordList();  
  
const cell = await singleSelectField.getCell(recordList[0]);

## setValue

设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: SingleSelectTransformVal) => Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
await singleSelectField.setValue(recordIdList[0], 'option\_id'); // 传入选项 id

## getValue

获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenSingleSelect>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
const cellValue = await singleSelectField.getValue(recordIdList[0]);

## addOption

新增选项，可指定选项名称和颜色。

addOption: (name: string, color?: number) => Promise<IFieldRes>;

#### 示例

await singleSelectField.addOption('new option');

## addOptions

新增多个选项，可指定选项名称和颜色。

addOptions: (optionList: { name: string, color?: number }[]) => Promise<IFieldRes>;

#### 示例

await singleSelectField.addOptions([  
 {   
 name: 'new option 1',  
 },  
 {   
 name: 'new option 2',  
 }  
]);

## getOptions

获取所有的选项，其中 ISelectFieldOption 的类型定义为：

getOptions: () => Promise<ISelectFieldOption[]>;  
  
interface ISelectFieldOption {  
 id: string;  
 name: string;  
 color: number;  
}

#### 示例

await singleSelectField.getOptions();

## deleteOption

通过选项 id 或者 name 删除选项。

deleteOption: (idOrName: string) => Promise<IFieldRes>;

#### 示例

const options = await singleSelectField.getOptions();  
  
await singleSelectField.deleteOption(options[0].id);

## setOption

通过选项 id 或者 name 设置选项，其中 OptionConfig 的类型定义为：

setOption: (nameOrId: string, config: OptionConfig) => Promise<IFieldRes>;  
  
export type OptionConfig = {  
 name?: string;  
 color?: number;  
};

#### 示例

const options = await singleSelectField.getOptions();  
  
await singleSelectField.setOption(options[0].id, {  
 name: 'modify option'  
});

## getOptionsType

获取选项类型，其中 SelectOptionsType 的类型定义为:

getOptionsType: () => Promise<SelectOptionsType>;  
  
enum SelectOptionsType {  
 STATIC, // 自定义选项  
 DYNAMIC, // 引用选项  
}

#### 示例

await singleSelectField.getOptionsType();

## setOptionsType

设置选项类型，其中 SelectOptionsType 的类型定义为:

setOptionsType: (type: SelectOptionsType) => Promise<IFieldRes>;  
  
enum SelectOptionsType {  
 STATIC, // 自定义选项  
 DYNAMIC, // 引用选项  
}

#### 示例

await singleSelectField.setOptionsType(SelectOptionsType.STATIC);

# Checkbox 复选框字段

类型定义 ICheckBoxField，使用方法示例：

const checkboxField = await table.getField<ICheckBoxField>(fieldId);

其中字段值的类型定义为：

export type IOpenCheckbox = boolean;

## createCell

创建一个复选框字段的 Cell。

createCell: (val: IOpenCheckbox) => Promise<ICell>;

### 示例

const cell = await checkBoxField.createCell(false);  
await table.addRecord(cell);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

### 示例

const cell = await checkBoxField.getCell(recordId);

## setValue

通过 Record 来设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenCheckbox) => Promise<boolean>;

### 示例

await checkBoxField.setValue(recordId, false);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenCheckbox>;

### 示例

await checkBoxField.getValue(recordId);

# Progress 进度字段

目前暂未支持配置进度字段相关属性。

类型定义 IProgressField，使用方法示例：

const progressField = await table.getField<IProgressField>(fieldId);

其中字段值的类型定义为：

type IOpenNumber = number;

## createCell

创建一个进度字段的 Cell。

createCell: (val: IOpenNumber) => Promise<ICell>;

#### 示例

await progressField.createCell(50);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await progressField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: IOpenNumber) => Promise<boolean>;

#### 示例

await progressField.setValue('r\_id', 50);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenNumber>;

#### 示例

await progressField.getValue('r\_id');

# Currency 货币字段

类型定义 ICurrencyField，使用方法示例：

const currencyField = await table.getField<ICurrencyField>(fieldId);

其中字段值的类型定义为：

type IOpenNumber = number;

## createCell

通过传入数值来创建对应的 Cell。

createCell: (val: IOpenNumber) => Promise<ICell>;

#### 示例

await currencyField.createCell('100');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await currencyField.getCell(recordId);

## setValue

通过 Record 来设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: number) => Promise<boolean>;

#### 示例

await currencyField.setValue(recordId, '1000');

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<number>;

#### 示例

await currencyField.getValue(recordId);

## setDecimalDigits

设置货币精度，**精度取值范围** 0~4。

setDecimalDigits: (decimalDigits: number) => Promise<IFieldRes>;

#### 示例

await currencyField.setDecimalDigits(4); // 0～4

## getDecimalDigits

获取货币精度。

getDecimalDigits: () => Promise<number>;

#### 示例

await currencyField.getDecimalDigits(); // 0～4

## setCurrencyCode

设置货币类型。

setCurrencyCode: (currencyCode: CurrencyCode) => Promise<IFieldRes>;

其中 CurrencyCode 的类型定义如下：

enum CurrencyCode {  
 CNY = 'CNY',  
 USD = 'USD',  
 EUR = 'EUR',  
 GBP = 'GBP',  
 AED = 'AED',  
 AUD = 'AUD',  
 BRL = 'BRL',  
 CAD = 'CAD',  
 CHF = 'CHF',  
 HKD = 'HKD',  
 INR = 'INR',  
 IDR = 'IDR',  
 JPY = 'JPY',  
 KRW = 'KRW',  
 MOP = 'MOP',  
 MXN = 'MXN',  
 MYR = 'MYR',  
 PHP = 'PHP',  
 PLN = 'PLN',  
 RUB = 'RUB',  
 SGD = 'SGD',  
 THB = 'THB',  
 TRY = 'TRY',  
 TWD = 'TWD',  
 VND = 'VND',  
}

#### 示例

await currencyField.setCurrencyCode(CurrencyCode.CNY);

## getCurrencyCode

获取货币类型。

getCurrencyCode: () => Promise<CurrencyCode>;

#### 示例

await currencyField.getCurrencyCode();

# MultipleSelect 多选字段

类型定义 IMultiSelectField，使用方法示例：

const multiSelectField = await table.getField<IMultiSelectField>(fieldId);

其中字段值的类型定义为：

// 多选值类型定义  
type IOpenMultiSelect = IOpenSingleSelect[];  
  
type IOpenSingleSelect = {  
 id: string; // 选项 id  
 text: string;  
};  
  
type MultiSelectTransformVal = string[] | string | IOpenMultiSelect | IOpenSingleSelect;

## createCell

创建一个多选字段的 Cell。

createCell: (val: MultiSelectTransformVal) => Promise<ICell>;

#### 示例

await multiSelectField.createCell('test option');

## getCell

获取指定记录对应的 Cell 单元格。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordList = await table.getRecordList();  
  
const cell = await multiSelectField.getCell(recordList[0]);

## setValue

设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: MultiSelectTransformVal) => Promise<boolean>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
await multiSelectField.setValue(recordIdList[0], ['option\_id1', 'option\_id2']); // 传入选项 id

## getValue

获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenMultiSelect>;

#### 示例

const table = await bitable.base.getActiveTable();  
const recordIdList = await table.getRecordIdList();  
  
const cellValue = await multiSelectField.getValue(recordIdList[0]);

## addOption

新增选项，可指定选项名称和颜色。

addOption: (name: string, color?: number) => Promise<IFieldRes>;

#### 示例

await multiSelectField.addOption('new option');

## addOptions

新增多个选项，可指定选项名称和颜色。

addOptions: (optionList: { name: string, color?: number }[]) => Promise<IFieldRes>;

#### 示例

await multiSelectField.addOptions([  
 {   
 name: 'new option 1',  
 },  
 {   
 name: 'new option 2',  
 }  
]);

## getOptions

获取所有的选项，其中 ISelectFieldOption 的类型定义为：

getOptions: () => Promise<ISelectFieldOption[]>;  
  
interface ISelectFieldOption {  
 id: string;  
 name: string;  
 color: number;  
}

#### 示例

await multiSelectField.getOptions();

## deleteOption

通过选项 id 或者 name 删除选项。

deleteOption: (idOrName: string) => Promise<IFieldRes>;

#### 示例

const options = await multiSelectField.getOptions();  
  
await multiSelectField.deleteOption(options[0].id);

## setOption

通过选项 id 或者 name 设置选项，其中 OptionConfig 的类型定义为：

setOption: (nameOrId: string, config: OptionConfig) => Promise<IFieldRes>;  
  
export type OptionConfig = {  
 name?: string;  
 color?: number;  
};

#### 示例

const options = await multiSelectField.getOptions();  
  
await multiSelectField.setOption(options[0].id, {  
 name: 'modify option'  
});

## getOptionsType

获取选项类型，其中 SelectOptionsType 的类型定义为:

getOptionsType: () => Promise<SelectOptionsType>;  
  
enum SelectOptionsType {  
 STATIC, // 自定义选项  
 DYNAMIC, // 引用选项  
}

#### 示例

await multiSelectField.getOptionsType();

## setOptionsType

设置选项类型，其中 SelectOptionsType 的类型定义为:

setOptionsType: (type: SelectOptionsType) => Promise<IFieldRes>;  
  
enum SelectOptionsType {  
 STATIC, // 自定义选项  
 DYNAMIC, // 引用选项  
}

#### 示例

await multiSelectField.setOptionsType(SelectOptionsType.STATIC);

# Formula 公式字段

Formula 公式字段**不支持手动往单元格写入值**。

类型定义 IFormulaField，使用方法示例：

const formulaField = await table.getField<IFormulaField>(fieldId);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await formulaField.getCell('r\_id');

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenFormulaCellValue>;

#### 示例

await formulaField.getValue('r\_id');

## setFormula

设置公式表达式，设置成功将返回该字段的 id。

公式的详细用法可查看 [公式字段概述](https://www.feishu.cn/hc/zh-CN/articles/360049067853-%E5%A4%9A%E7%BB%B4%E8%A1%A8%E6%A0%BC%E5%85%AC%E5%BC%8F%E5%AD%97%E6%AE%B5%E6%A6%82%E8%BF%B0)。

setFormula: (formula: string) => Promise<string>;

#### 示例

await formulaField.setFormula("多行文本"); // 公式列的值等于多行文本列  
await formulaField.setFormula("截止日期-开始日期"); // 计算截止日期和开始日期之间的天数  
await formulaField.setFormula('单价\*销量'); // 计算单价字段和销量字段的乘积  
await formulaField.setFormula('[销售统计].FILTER(CurrentValue.[成员] = [成员]).[销售额].SUM()'); // 在数据表 [销售额汇总] 中，计算 [成员] 字段下，每个员工的销售总额（从数据表 [销售统计] 中跨表引用销售数据）

# 字段模块引导

字段即多维表格中的列，字段类型决定了这一列的数据类型。

通常我们通过 [Table 模块](../table) 创建或获取字段，如下所示：

const singleSelectField = await table.getField<ISingleSelectField>(fieldNameOrId);

这里有一个很重要的地方，我们在调用 getField 方法时，传入了指定的类型 <ISingleSelectField>，我们非常推荐这样的用法，通过这样的方法获取的 Field，会有足够的类型提示, 例如我们可以很方便地为这个单选字段新增选项：

await singleSelectField.addOption('Option1');

除了设置字段的属性之外，我们也推荐开发者从字段角度来对值进行增删改查操作例如：

await singleSelectField.setValue(recordOrId, 'Option2');

基于列的角度对数据进行增删改查时非常简单和便利，我们为很多字段提供了便于开发者使用的方法 ，这里展示一个通过 [附件字段](./attachment.md) 来创建一条记录的例子：

const attachmentCell = await attachmentField.createCell(imageFile);  
await table.addRecord(attachmentCell);

目前我们已支持了多种字段类型，详细用法可以点击具体类型的字段模块中查看。

# Phone 电话字段

类型定义 IPhoneField，使用方法示例：

const phoneField = await table.getField<IPhoneField>(fieldId);

其中字段值的类型定义为：

type IOpenPhone = string;

## createCell

创建一个电话字段的 Cell。

createCell: (val: IOpenPhone) => Promise<ICell>;

#### 示例

await phoneField.createCell(123456789);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await phoneField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值，**此处写入的字符串不会校验电话格式**。

setValue: (recordOrId: IRecordType | string, val: IOpenPhone) => Promise<boolean>;

#### 示例

await phoneField.setValue('r\_id', 123456789);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenPhone>;

#### 示例

await phoneField.getValue('r\_id');

# ModifiedUser 更新人字段

ModifiedUser 更新人字段**不支持手动往单元格写入值**。

类型定义 IModifiedUserField，使用方法示例：

const modifiedUserField = await table.getField<IModifiedUserField>(fieldId);

其中字段值的类型定义为：

type IOpenUser = {  
 id: string;  
 name?: string;  
 enName?: string;  
 email?: string;  
};

## getValue

获取创建修改人的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenUser[]>;

#### 示例

await modifiedUserField.getValue('r\_id');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await modifiedUserField.getCell('r\_id');

# ModifiedTime 更新时间字段

ModifiedTime 更新时间字段**不支持手动往单元格写入值**。

类型定义 IModifiedTimeField，使用方法示例：

const modifiedTimeField = await table.getField<IModifiedTimeField>(fieldId);

其中字段值的类型定义为：

type IOpenTimestamp = number;

## getValue

获取指定单元格的更新时间。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenTimestamp>;

#### 示例

await modifiedTimeField.getValue('r\_id');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await modifiedTimeField.getCell('r\_id');

## setDateFormat

设置字段日期格式。

setDateFormat: (format: DateFormatter) => Promise<IFieldRes>;

其中 DateFormatter 的数据格式为：

enum DateFormatter {  
 DATE\_YMD\_WITH\_SLASH = 'yyyy/MM/dd', // 2021/01/30  
 DATE\_TIME = 'yyyy/MM/dd HH:mm', // 2021/01/30 14:00  
 DATE\_TIME\_WITH\_HYPHEN = 'yyyy-MM-dd HH:mm', // 2021-01-30 14:00  
 DATE\_YMD\_WITH\_HYPHEN = 'yyyy-MM-dd', // 2021-01-30  
 DATE\_MD\_WITH\_HYPHEN = 'MM-dd', // 01-30  
 DATE\_MMDD\_WITH\_SLASH = 'MM/dd/yyyy', // 01/30/2021  
 DATE\_DDMM\_WITH\_SLASH = 'dd/MM/yyyy', // 30/01/2021  
}

#### 示例

await modifiedTimeField.setDateFormat(DateFormatter.DATE\_TIME);

## getDateFormat

获取日期格式。

getDateFormat: () => Promise<DateFormatter>;

#### 示例

await modifiedTimeField.getDateFormat();

# UserField 人员字段

类型定义 IUserField，使用方法示例：

const userField = await table.getField<IUserField>(fieldId);

其中字段值的类型定义为：

export type IOpenUser = {  
 id: string; // open user\_id  
 name?: string; // 名称  
 enName?: string; // 英文名  
 email?: string; // 邮箱  
};  
  
type UserFieldTransformVal = IOpenUser | IOpenUser[];

## createCell

创建一个人员字段的 Cell。

createCell: (val: UserFieldTransformVal) => Promise<ICell>;

#### 示例

await userField.createCell({  
 id: 'ou\_xxxx'  
});

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await userField.getCell('r\_id');

## setValue

通过 Record 来设置对应的值。

setValue: (recordOrId: IRecordType | string, val: UrlTransformVal) => Promise<boolean>;

#### 示例

await userField.setValue([  
 {  
 id: 'ou\_xxxx1'  
 },  
 {  
 id: 'ou\_xxxx2'  
 }  
]);

## getValue

通过 Record 来获取对应的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenUser[]>;

#### 示例

await userField.getValue('r\_id');

## setMultiple

设置是否允许设置多个人员。

setMultiple: (multiple: boolean) => Promise<IFieldRes>;

#### 示例

await userField.setMultiple(false);

## getMultiple

获取是否允许设置多个人员属性值。

getMultiple: () => Promise<boolean>;

#### 示例

await userField.getMultiple();  
// true

# CreateTime 创建时间字段

CreateTime 创建时间字段**不支持手动往单元格写入值**。

类型定义 ICreateTimeField，使用方法示例：

const createTimeField = await table.getField<ICreateTimeField>(fieldId);

其中字段值的类型定义为：

type IOpenTimestamp = number;

## getValue

获取指定单元格的创建时间。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenTimestamp>;

#### 示例

await createTimeField.getValue('r\_id');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await createTimeField.getCell('r\_id');

## setDateFormat

设置字段日期格式。

setDateFormat: (format: DateFormatter) => Promise<IFieldRes>;

其中 DateFormatter 的数据格式为：

enum DateFormatter {  
 DATE\_YMD\_WITH\_SLASH = 'yyyy/MM/dd', // 2021/01/30  
 DATE\_TIME = 'yyyy/MM/dd HH:mm', // 2021/01/30 14:00  
 DATE\_TIME\_WITH\_HYPHEN = 'yyyy-MM-dd HH:mm', // 2021-01-30 14:00  
 DATE\_YMD\_WITH\_HYPHEN = 'yyyy-MM-dd', // 2021-01-30  
 DATE\_MD\_WITH\_HYPHEN = 'MM-dd', // 01-30  
 DATE\_MMDD\_WITH\_SLASH = 'MM/dd/yyyy', // 01/30/2021  
 DATE\_DDMM\_WITH\_SLASH = 'dd/MM/yyyy', // 30/01/2021  
}

#### 示例

await createTimeField.setDateFormat(DateFormatter.DATE\_TIME);

## getDateFormat

获取日期格式。

getDateFormat: () => Promise<DateFormatter>;

#### 示例

await createTimeField.getDateFormat();

# CreateUser 创建人字段

CreateUser 创建人字段**不支持手动往单元格写入值**。

类型定义 ICreateUserField，使用方法示例：

const createUserField = await table.getField<ICreateUserField>(fieldId);

其中字段值的类型定义为：

type IOpenUser = {  
 id: string;  
 name?: string;  
 enName?: string;  
 email?: string;  
};

## getValue

获取创建人的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenUser[]>;

#### 示例

await createUserField.getValue('r\_id');

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await createUserField.getCell('r\_id');

# Rating 评分字段

类型定义 IRatingField，使用方法示例：

const ratingField = await table.getField<IRatingField>(fieldId);

其中字段值的类型定义为：

type IOpenNumber = number;

## createCell

创建一个评分字段的 Cell，**写入的评分值需要落在字段配置的最小值和最大值之间**。

createCell: (val: IOpenNumber) => Promise<ICell>;

#### 示例

await ratingField.createCell(5);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await ratingField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值，**写入的评分值需要落在字段配置的最小值和最大值之间**。

setValue: (recordOrId: IRecordType | string, val: IOpenNumber) => Promise<boolean>;

#### 示例

// eg: min~max => 0~5  
await ratingField.setValue('r\_id', 4);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenNumber>;

#### 示例

await ratingField.getValue('r\_id');

## getMin

获取设置的评分最小值，**最小值取值范围 0~1**。

getMin: () => Promise<number>;

#### 示例

await ratingField.getMin();

## setMin

设置评分最小值，**最小值取值范围 0~1**。

setMin: (min: number) => Promise<IFieldRes>;

#### 示例

await ratingField.setMin(0);

## getMax

获取设置的评分最大值，**最大值取值范围 1~10**。

getMax: () => Promise<number>;

#### 示例

await ratingField.getMax();

## setMax

设置评分最大值，**最大值取值范围 1~10**。

setMax: (max: number) => Promise<IFieldRes>;

#### 示例

await ratingField.setMax(10);

## getRatingIcon

获取评分字段的 ICON。

getRatingIcon: () => Promise<RatingIconType>;

其中 RatingIconType 的值为：

enum RatingIconType {  
 STAR = 'star',  
 HEART = 'heart',  
 THUMBSUP = 'thumbsup',  
 FIRE = 'fire',  
 SMILE = 'smile',  
 LIGHTNING = 'lightning',  
 FLOWER = 'flower',  
 NUMBER = 'number',  
}

#### 示例

await ratingField.getRatingIcon();

## setRatingIcon

设置评分字段的 ICON。

setRatingIcon: (icon: RatingIconType) => Promise<IFieldRes>;

#### 示例

await ratingField.setRatingIcon(RatingIconType.FLOWER);

# Lookup 查找引用字段

Lookup 查找引用字段**不支持手动往单元格写入值**。

类型定义 ILookupField，使用方法示例：

const lookupField = await table.getField<ILookupField>(fieldId);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await lookupField.getCell('r\_id');

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenFormulaCellValue>;

#### 示例

await lookupField.getValue('r\_id');

# Number 数字字段

类型定义 INumberField，使用方法示例：

const numberField = await table.getField<INumberField>(fieldId);

其中字段值的类型定义为：

type IOpenNumber = number;

## createCell

创建一个数字字段的 Cell。

createCell: (val: number) => Promise<ICell>;

#### 示例

await numberField.createCell(123);

## getCell

通过对应的 Record 来获取对应的 Cell。

getCell: (recordOrId: IRecordType | string) => Promise<ICell>;

#### 示例

await numberField.getCell('r\_id');

## setValue

通过 Record 来设置指定单元格的值。

setValue: (recordOrId: IRecordType | string, val: number) => Promise<boolean>;

#### 示例

await numberField.setValue('r\_id', 123);

## getValue

通过 Record 来获取指定单元格的值。

getValue: (recordOrId: IRecordType | string) => Promise<IOpenNumber>;

#### 示例

await numberField.getValue('r\_id');

## setFormatter

设置数字的格式。

setFormatter: (formatter: NumberFormatter) => Promise<IFieldRes>;

其中 NumberFormatter 的类型定义如下：

enum NumberFormatter {  
 INTEGER = '0',  
 DIGITAL\_ROUNDED\_1 = '0.0',  
 DIGITAL\_ROUNDED\_2 = '0.00',  
 DIGITAL\_ROUNDED\_3 = '0.000',  
 DIGITAL\_ROUNDED\_4 = '0.0000',  
 DIGITAL\_THOUSANDS = '1,000',  
 DIGITAL\_THOUSANDS\_DECIMALS = '1,000.00',  
 PERCENTAGE\_ROUNDED = '0%',  
 PERCENTAGE = '0.00%',  
}

#### 示例

await numberField.setFormatter(NumberFormatter.INTEGER);

## getFormatter

获取当前的数字格式。

getFormatter: () => Promise<NumberFormatter>;

#### 示例

await numberField.getFormatter();

# API 引导

推荐先从 [快速入门](../start/core) 开始阅读

根据[快速入门](../start/core)中的知识，大家应该对多维表格的模型有了一个初步的了解，这篇文章，帮助大家简单强化理解一下这个模型，便于插件的设计以及 API 的查询。

bitable，是所有 API 的统一入口，不同的功能模块均会挂载至该统一入口。

## Base 模块

绝大部分的 API 调用方法都是从 base 调用的，因为对多维表格进行增删改查的 API 都位于 base 下，大部分 Case 如下所示：

import { bitable } from '@lark-base-open/js-sdk';  
  
const table = await bitable.base.getActiveTable();

getActiveTable 的作用是来获取当前页面已经选择的 table，在 base 上还有很多接口可以用来获取 table，可以在 Base 模块的[文档](./base)内查看。

## Table 模块与 Field 模块

获取到 table 之后，就可以做很多数据相关的增删改查操作。

推荐开发者从 field(字段) 的角度来对数据进行增删改差，因为数据每一个单元格内的数据是由其所属的字段类型决定的，因此在通过字段来实现增删改查的时候，会给予更多的类型提示，下面是一个例子：

const attachmentField = await table.getField<IAttachmentField>(fieldId);  
const attachmentCell = await attachmentField.createCell(file);  
await table.addRecord(attachmentCell);

在这个例子中，我们通过 fieldId 来获取了一个 attachmentField(附件字段)，在获取这个字段的时候，我们传入了一个很重要的东西 IAttachemntField，这是一个类型参数，告诉 ts 我们获取的是一个附件字段， 因此在后续调用 attachmentField.createCell(file) 的时候，我们是可以获得足够的类型提示的，告诉我们可以通过直接传入一个 File / File[] / FileList 来构造出一个附件单元格。

然后我们调用 table.addRecord(attachmentCell) 方法，将这个单元格插入到了表格之中（添加了一条记录），table.addRecord 方法还支持传入 Cell[] 参数，从而创建出一条更为完整的数据。

除此之外，IAttachmentField(附件字段) 还有一些便于开发者使用的 API：

const attachmentUrls = await attachmentField.getAttachmentUrls(recordId)

因为附件字段中存储的并不是真正的 URL，所以在获取真正的 URL 时，需要多步，但是从字段本身的角度去考虑，我们在实现这个 API 的时候，就可以把这些工作放在字段自身的方法里实现， 所以还是非常推荐用户对数据的增删改查以及字段属性的设置可以从字段角度去考量。

除了 IAttachmentField(附件字段) 以外，我们还细化了很多字段，可以从 [字段引导](field/guide.md) 阅读更多资料，Table 模块也有更多的 API 方法来供给开发者使用。

## Cell 模块

在上文中，通过 attachmentField.creatCell 方法构造出来的 Cell 也是一个非常重要的模块，在进行插入数据的操作时，我们推荐开发者通过 Field 来构造 Cell，来插入数据， 当一个 Cell 被成功插入到 Table 中去后，它会与一条数据进行关联，此时进行 getValue/setValue 时，都会与实时性的数据产生关联，按照上述的流程可以这样来改变对应单元格的值：

await attachmentCell.setValue(newFile)

在 setValue 顺利执行之后，表格中对应单元格的值会发生改变，更多的 API 用法可以查看对应[文档](cell.md)。

## Record 模块

Record 模块主要用来存储**行数据**，API 可以查看对应[文档](record.md)。

## UI 模块

UI 模块提供与用户交互相关的能力，如切换数据表、切换视图等，详细 API 可查看[UI 模块文档](ui.md)。

## Bridge 模块

Bridge 模块提供了一些通用能力，如获取用户 id，获取当前环境信息等，详细 API 可查看 [Bridge 模块文档](bridge.md)。

# CalendarView 日历视图

## id

当前视图的 id

## tableId

当前视图所属的数据表 id

## getName

getName(): Promise<string>;

获取视图名

## getType

获取视图类型

getType(): Promise<ViewType.Calendar>;

## getMeta

getMeta(): Promise<ICalendarViewMeta>;

获取日历视图元信息，其中 ICalendarViewMeta 的类型定义如下：

interface ICalendarViewMeta {  
 id: string;  
 name: string;  
 type: ViewType.Calendar;  
 property: {  
 filterInfo: IFilterInfo | null;  
 }  
}

## showField

showField: (fieldId: string | string[]) => Promise<boolean>;

显示字段

## hideField

hideField: (fieldId: string | string[]) => Promise<boolean>;

隐藏字段

## getFieldMetaList

getFieldMetaList(): Promise<IFieldMeta[]>;

获取字段信息的列表，因为 View 层涉及到了 UI 的展示，所以此时获取的字段信息是有序的

## getVisibleRecordIdList

getVisibleRecordIdList(): Promise<(string | undefined)[]>;

获取可见记录的 ID 列表

## getVisibleFieldIdList

getVisibleFieldIdList(): Promise<string[]>;

获取可见字段的 ID 列表

## applySetting

applySetting(): Promise<void>;

将设置的分组/筛选/排序等视图配置提交，同步给其他用户

## getFilterInfo

getFilterInfo(): Promise<IFilterInfo | null>;

获取当前的筛选信息([IFilterInfo 定义](../view.md#ifilterinfo))

## addFilterCondition

addFilterCondition: (param: IAddFilterConditionParams) => Promise<boolean>;

新增筛选条件，如果新增失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./calendar.md#applysetting) 方法)

## deleteFilterCondition

deleteFilterCondition: (conditionId: string) => Promise<boolean>;

删除筛选条件，如果删除失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./calendar.md#applysetting) 方法)

## updateFilterCondition

updateFilterCondition: (param: IUpdateFilterConditionParams) => Promise<boolean>;

更新筛选条件，如果更新失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./calendar.md#applysetting) 方法)

## setFilterConjunction

setFilterConjunction: (conjunction: FilterConjunction) => Promise<boolean>;

设置筛选条件之间的关系，其中 FilterConjunction 类型定义为:

enum FilterConjunction {  
 And = "and",  
 Or = "or"  
}

可以选择满足所有筛选条件或者其中某条件 (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./calendar.md#applysetting) 方法)

# FormView 表单视图

## id

当前视图的 id

## tableId

当前视图所属的数据表 id

## getName

getName(): Promise<string>;

获取视图名

## getType

获取视图类型

getType(): Promise<ViewType.Form>;

## getMeta

getMeta(): Promise<IFormViewMeta>;

获取视图元数据，其中 IFormViewMeta 的类型定义为：

interface IFormViewMeta {  
 id: string;  
 name: string;  
 type: ViewType;  
 property: null;  
}

# KanbanView 看板视图

## id

当前视图的 id

## tableId

当前视图所属的数据表 id

## getName

getName(): Promise<string>;

获取视图名

## getType

获取视图类型

getType(): Promise<ViewType.Kanban>;

## getMeta

getMeta(): Promise<IKanbanViewMeta>;

获取视图元数据，其中 IKanbanViewMeta 的类型定义为：

interface IKanbanViewMeta {  
 id: string;  
 name: string;  
 type: ViewType.Kanban;  
 property: {  
 filterInfo: IFilterInfo | null;  
 sortInfo: ISortInfo[];  
 };  
}

## showField

showField: (fieldId: string | string[]) => Promise<boolean>;

显示字段

## hideField

hideField: (fieldId: string | string[]) => Promise<boolean>;

隐藏字段

## getFieldMetaList

getFieldMetaList(): Promise<IFieldMeta[]>;

获取字段信息的列表，因为 View 层涉及到了 UI 的展示，所以此时获取的字段信息是有序的

## getVisibleRecordIdList

getVisibleRecordIdList(): Promise<(string | undefined)[]>;

获取可见记录的 ID 列表

## getVisibleFieldIdList

getVisibleFieldIdList(): Promise<string[]>;

获取可见字段的 ID 列表

## applySetting

applySetting(): Promise<void>;

将设置的分组/筛选/排序等视图配置提交，同步给其他用户

## getFilterInfo

getFilterInfo(): Promise<IFilterInfo | null>;

获取当前的筛选信息([IFilterInfo 定义](../view.md#ifilterinfo))

## addFilterCondition

addFilterCondition: (param: IAddFilterConditionParams) => Promise<boolean>;

新增筛选条件，如果新增失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## deleteFilterCondition

deleteFilterCondition: (conditionId: string) => Promise<boolean>;

删除筛选条件，如果删除失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## updateFilterCondition

updateFilterCondition: (param: IUpdateFilterConditionParams) => Promise<boolean>;

更新筛选条件，如果更新失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## setFilterConjunction

setFilterConjunction: (conjunction: FilterConjunction) => Promise<boolean>;

设置筛选条件之间的关系，其中 FilterConjunction 类型定义为:

enum FilterConjunction {  
 And = "and",  
 Or = "or"  
}

可以选择满足所有筛选条件或者其中某条件 (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()) ## getSortInfo

getSortInfo(): Promise<ISortInfo[]>;

获取当前的排序条件([sortInfo定义](../view.md#isortinfo))

## setAutoSort

setAutoSort(param: boolean): Promise<boolean>;

设置是否自动进行排序（在设置了排序条件之后，会自动设置为 true, 调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## addSort

addSort: (param: ISortInfo | ISortInfo[]) => Promise<boolean>;

新增排序条件（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## deleteSort

deleteSort: (param: ISortInfo | string) => Promise<boolean>;

删除排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## updateSort

updateSort: (param: ISortInfo) => Promise<boolean>;

更新排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

# GalleryView 画册视图

## id

当前视图的 id

## tableId

当前视图所属的数据表 id

## getName

getName(): Promise<string>;

获取视图名

## getType

获取视图类型

getType(): Promise<ViewType.Gallery>;

## getMeta

getMeta(): Promise<IGalleryViewMeta>;

获取视图元数据，其中 IGalleryViewMeta 的类型定义为：

interface IGalleryViewMeta {  
 id: string;  
 name: string;  
 type: ViewType.Gallery;  
 property: {  
 sortInfo: ISortInfo[];  
 filterInfo: IFilterInfo | null;  
 };  
}

## showField

showField: (fieldId: string | string[]) => Promise<boolean>;

显示字段

## hideField

hideField: (fieldId: string | string[]) => Promise<boolean>;

隐藏字段

## getFieldMetaList

getFieldMetaList(): Promise<IFieldMeta[]>;

获取字段信息的列表，因为 View 层涉及到了 UI 的展示，所以此时获取的字段信息是有序的

## getVisibleRecordIdList

getVisibleRecordIdList(): Promise<(string | undefined)[]>;

获取可见记录的 ID 列表

## getVisibleFieldIdList

getVisibleFieldIdList(): Promise<string[]>;

获取可见字段的 ID 列表

## applySetting

applySetting(): Promise<void>;

将设置的分组/筛选/排序等视图配置提交，同步给其他用户

## getFilterInfo

getFilterInfo(): Promise<IFilterInfo | null>;

获取当前的筛选信息([IFilterInfo 定义](../view.md#ifilterinfo))

## addFilterCondition

addFilterCondition: (param: IAddFilterConditionParams) => Promise<boolean>;

新增筛选条件，如果新增失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## deleteFilterCondition

deleteFilterCondition: (conditionId: string) => Promise<boolean>;

删除筛选条件，如果删除失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## updateFilterCondition

updateFilterCondition: (param: IUpdateFilterConditionParams) => Promise<boolean>;

更新筛选条件，如果更新失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## setFilterConjunction

setFilterConjunction: (conjunction: FilterConjunction) => Promise<boolean>;

设置筛选条件之间的关系，其中 FilterConjunction 类型定义为:

enum FilterConjunction {  
 And = "and",  
 Or = "or"  
}

可以选择满足所有筛选条件或者其中某条件 (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()) ## getSortInfo

getSortInfo(): Promise<ISortInfo[]>;

获取当前的排序条件([sortInfo定义](../view.md#isortinfo))

## setAutoSort

setAutoSort(param: boolean): Promise<boolean>;

设置是否自动进行排序（在设置了排序条件之后，会自动设置为 true, 调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## addSort

addSort: (param: ISortInfo | ISortInfo[]) => Promise<boolean>;

新增排序条件（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## deleteSort

deleteSort: (param: ISortInfo | string) => Promise<boolean>;

删除排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## updateSort

updateSort: (param: ISortInfo) => Promise<boolean>;

更新排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

# GridView 表格视图

## id

当前视图的 id。

## tableId

当前视图所属的数据表 id。

## getName

获取视图名。

getName(): Promise<string>;

## getType

获取视图类型。

getType(): Promise<ViewType.Grid>;

## getMeta

获取表格视图元信息。

getMeta(): Promise<IGridViewMeta>;

其中 IGridViewMeta 类型定义为：

interface IGridViewMeta {  
 id: string;  
 name: string;  
 type: ViewType.Grid;  
 property: {  
 hierarchyConfig: {  
 fieldId: string | undefined;  
 };  
 filterInfo: IFilterInfo | null;  
 sortInfo: ISortInfo[];  
 groupInfo: IGroupInfo[];  
 }  
}

## getFieldMetaList

获取字段信息的列表，因为 View 层涉及到了 UI 的展示，所以此时获取的字段信息是有序的。

getFieldMetaList(): Promise<IFieldMeta[]>;

## getVisibleRecordIdList

获取可见记录的 ID 列表。

getVisibleRecordIdList(): Promise<(string | undefined)[]>;

## getVisibleFieldIdList

获取可见字段的 ID 列表。

getVisibleFieldIdList(): Promise<string[]>;

## getSelectedRecordIdList

获取当前选中的所有记录 ID 列表。

getSelectedRecordIdList(): Promise<string[]>;

## applySetting

将设置的分组/筛选/排序等视图配置提交，同步给其他用户。

applySetting(): Promise<void>;

## getChildRecordIdList

获取指定记录的子记录 id 列表, undefined 则表示该记录无子记录。

getChildRecordIdList(parentRecordId: string): Promise<RecordId[] | undefined>;

## getFilterInfo

获取当前的筛选信息([IFilterInfo 定义](../view.md#ifilterinfo))。

getFilterInfo(): Promise<IFilterInfo | null>;

## addFilterCondition

新增筛选条件，如果新增失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting))。

addFilterCondition: (param: IAddFilterConditionParams) => Promise<boolean>;

## deleteFilterCondition

删除筛选条件，如果删除失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting))。

deleteFilterCondition: (conditionId: string) => Promise<boolean>;

## updateFilterCondition

更新筛选条件，如果更新失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting))。

updateFilterCondition: (param: IUpdateFilterConditionParams) => Promise<boolean>;

## setFilterConjunction

setFilterConjunction: (conjunction: FilterConjunction) => Promise<boolean>;

设置筛选条件之间的关系，其中 FilterConjunction 类型定义为:

enum FilterConjunction {  
 And = "and",  
 Or = "or"  
}

可以选择满足所有筛选条件或者其中某条件 (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting))

## getSortInfo

获取当前的排序条件([sortInfo定义](../view.md#isortinfo))。

getSortInfo(): Promise<ISortInfo[]>;

## setAutoSort

设置是否自动进行排序（在设置了排序条件之后，会自动设置为 true, 调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting))。

setAutoSort(param: boolean): Promise<boolean>;

## addSort

新增排序条件（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

addSort: (param: ISortInfo | ISortInfo[]) => Promise<boolean>;

## deleteSort

删除排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

deleteSort: (param: ISortInfo | string) => Promise<boolean>;

## updateSort

更新排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

updateSort: (param: ISortInfo) => Promise<boolean>;

## getGroupInfo

获取分组信息([IGroupInfo定义](../view.md#igroupinfo))。

getGroupInfo(): Promise<IGroupInfo[]>;

## addGroup

新增分组 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

addGroup: (param: IGroupInfo | IGroupInfo[]) => Promise<boolean>;

## deleteGroup

删除分组 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

deleteGroup: (param: string | IGroupInfo) => Promise<boolean>;

## updateGroup

更新分组（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

updateGroup: (param: IGroupInfo) => Promise<boolean>;

## showField

显示字段。

showField: (fieldId: string | string[]) => Promise<boolean>;

## hideField

隐藏字段。

hideField: (fieldId: string | string[]) => Promise<boolean>;

## getFieldWidth

获取字段宽度。

getFieldWidth(fieldId: string): Promise<number>;

## setFieldWidth

设置字段宽度。

setFieldWidth(fieldId: string, width: number): Promise<boolean>;

## setRowHeight

设置行高，目前行高按照从矮到高有以下几种 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 [view.applySetting](./grid.md#applysetting)）。

setRowHeight(rowHeight: RowHeightLevel): Promise<boolean>;

enum RowHeightLevel {  
 Short = 1,  
 Medium = 2,  
 Tall = 3,  
 ExtraTall = 4  
}

# GanttView 甘特视图

## id

当前视图的 id

## tableId

当前视图所属的数据表 id

## getName

getName(): Promise<string>;

获取视图名

## getType

获取视图类型

getType(): Promise<ViewType.Gantt>;

## getMeta

获取视图元数据，其中 IGanttViewMeta 的类型定义为：

getMeta(): Promise<IGanttViewMeta>;  
  
interface IGanttViewMeta {  
 id: string;  
 name: string;  
 type: ViewType.Gantt;  
 property: {  
 filterInfo: IFilterInfo | null;  
 sortInfo: ISortInfo[];  
 groupInfo: IGroupInfo[];  
 };  
}

## showField

showField: (fieldId: string | string[]) => Promise<boolean>;

显示字段

## hideField

hideField: (fieldId: string | string[]) => Promise<boolean>;

隐藏字段

## getFieldMetaList

getFieldMetaList(): Promise<IFieldMeta[]>;

获取字段信息的列表，因为 View 层涉及到了 UI 的展示，所以此时获取的字段信息是有序的

## getVisibleRecordIdList

getVisibleRecordIdList(): Promise<(string | undefined)[]>;

获取可见记录的 ID 列表

## getVisibleFieldIdList

getVisibleFieldIdList(): Promise<string[]>;

获取可见字段的 ID 列表

## applySetting

applySetting(): Promise<void>;

将设置的分组/筛选/排序等视图配置提交，同步给其他用户

## getFilterInfo

getFilterInfo(): Promise<IFilterInfo | null>;

获取当前的筛选信息([IFilterInfo 定义](../view.md#ifilterinfo))

## addFilterCondition

addFilterCondition: (param: IAddFilterConditionParams) => Promise<boolean>;

新增筛选条件，如果新增失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## deleteFilterCondition

deleteFilterCondition: (conditionId: string) => Promise<boolean>;

删除筛选条件，如果删除失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## updateFilterCondition

updateFilterCondition: (param: IUpdateFilterConditionParams) => Promise<boolean>;

更新筛选条件，如果更新失败，则会返回 false (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## setFilterConjunction

setFilterConjunction: (conjunction: FilterConjunction) => Promise<boolean>;

设置筛选条件之间的关系，其中 FilterConjunction 类型定义为:

enum FilterConjunction {  
 And = "and",  
 Or = "or"  
}

可以选择满足所有筛选条件或者其中某条件 (调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## getSortInfo

getSortInfo(): Promise<ISortInfo[]>;

获取当前的排序条件([sortInfo定义](../view.md#isortinfo))

## setAutoSort

setAutoSort(param: boolean): Promise<boolean>;

设置是否自动进行排序（在设置了排序条件之后，会自动设置为 true, 调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting())

## addSort

addSort: (param: ISortInfo | ISortInfo[]) => Promise<boolean>;

新增排序条件（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## deleteSort

deleteSort: (param: ISortInfo | string) => Promise<boolean>;

删除排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## updateSort

updateSort: (param: ISortInfo) => Promise<boolean>;

更新排序条件 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## getGroupInfo

getGroupInfo(): Promise<IGroupInfo[]>;

获取分组信息([IGroupInfo定义](../view.md#igroupinfo))

## addGroup

addGroup: (param: IGroupInfo | IGroupInfo[]) => Promise<boolean>;

新增分组 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## deleteGroup

deleteGroup: (param: string | IGroupInfo) => Promise<boolean>;

删除分组 （调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

## updateGroup

updateGroup: (param: IGroupInfo) => Promise<boolean>;

更新分组（调用该 API 时，并不会保存修改的设置，如果需要保存则需要额外调用 view.applySetting()）

# Bridge 模块

Bridge 模块主要提供了一些通用能力的 API，例如（获取当前语言信息/获取当前多维表格主题信息等), 获取方式为：

const bridge = bitable.bridge;

## setData

通过指定 key 存储当前插件自定义数据，该自定义数据在同一插件下是共享的。

setData<T>(key: string, data: T): Promise<boolean>;

#### 示例

await bitable.bridge.setData('test\_key1', 'hello world');  
await bitable.bridge.setData('test\_key2', 1);  
await bitable.bridge.setData('test\_key3', { key: 'value' });

## getData

通过指定 key 获取当前插件自定义数据。

getData<T>(key: string): Promise<T>;

#### 示例

await bitable.bridge.getData('test\_key1') // 'hello world'  
await bitable.bridge.getData('test\_key2') // 1  
await bitable.bridge.getData('test\_key3') // { key: 'value' }

## onDataChange

监听数据存储变化，任意存储 key 的变化都会触发回调。

onDataChange(callback: (ev: IEventCbCtx)) => void

#### 示例

bridge.onDataChange((event) => {  
 console.log('data change', event.data);  
})  
  
bridge.setData('test\_key', 233);

## getBitableUrl

生成多维表格链接。

getBitableUrl(options: GetBitableUrlOptions): Promise<string>;

其中 GetBitableUrlOptions 的类型参数定义为：

type GetBitableUrlOptions = {  
 tableId: string,  
 viewId: string,  
 // recordId 为空时打开表格，不为空时打开卡片  
 recordId?: string,  
 fieldId?: string,  
}

#### 示例

const res = await bitable.bridge.getBitableUrl({  
 tableId: 'tblkrAjKK1wEP4Nf',  
 viewId: 'vewboZNrq3',  
 fieldId: 'fldfd2ITyJ',  
});  
// 'https://bytedance.feishu.cn/base/QtTUxxxx?field=fldfd2ITyJ&table=tblkrAjKK1wEP4Nf&view=vewboZNrq3'

## getUserId

获取当前用户 ID。

该接口返回的用户 ID 与飞书开放平台的 OpenUserId 并不通用，请勿将其作为全平台的唯一 id。

getUserId(): Promise<string>;

#### 示例

const res = await bitable.bridge.getUserId();

## getTheme

获取当前主题。

getTheme(): Promise<ThemeModeType>;

其中 ThemeModeType 类型定义如下：

enum ThemeModeType {  
 LIGHT = "LIGHT",  
 DARK = "DARK"  
}

#### 示例

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

## onThemeChange

监听主题变化。

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

#### 示例

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

## getLocale

获取地区设置。

getLocale(): Promise<Locale>;

其中 Local 的类型定义如下:

type Locale = 'zh-CN' | 'zh-TW' | 'zh-HK' | 'en-US' | 'ja-JP' | 'fr-FR' | 'hi-IN' | 'id-ID' | 'it-IT' | 'ko-KR' | 'pt-BR' | 'ru-RU' | 'th-TH' | 'vi-VN' | 'de-DE' | 'es-ES';

#### 示例

const locale = await bitable.bridge.getLocale();  
// 'zh-CN'

## getLanguage

获取当前的语言信息。

getLanguage(): Promise<Language>;

其中 Language 的类型定义如下：

type Language = 'zh' | 'zh-TW' | 'zh-HK' | 'en' | 'ja' | 'fr' | 'hi' | 'id' | 'it' | 'ko' | 'pt' | 'ru' | 'th' | 'vi' | 'de' | 'es';

#### 示例

const language = await bitable.bridge.getLanguage();  
// 'zh'

## getTenantKey

获取当前租户 Id。

getTenantKey(): Promise<string>;

#### 示例

await bitable.bridge.getTenantKey();

## getEnv

获取当前的环境信息。

getEnv(): Promise<Env>;

其中 Env 的类型定义如下：

type Product = 'lark' | 'feishu';  
interface Env {  
 product: Product;  
}

#### 示例

const env = await bitable.bridge.getEnv();  
// { product: 'feishu' }

## getInstanceId

获取当前的插件的实例 id，每个插件的实例 id 全局唯一。

getInstanceId(): Promise<string>;

#### 示例

const instanceId = await bitable.bridge.getInstanceId();

## navigateToExtension

跳转至指定 id 对应的插件，需经过用户授权确认，用户拒绝授权或跳转失败时返回 false。

navigateToExtension(extensionId: string): Promise<boolean>;

#### 示例

await bitable.bridge.navigateToExtension('xxx\_id');

# Cell 模块

Cell 模块可以理解为表格视图中的单元格，通过 Cell 模块能够**以更细的粒度操作数据表**，可以通过如下几种方式来获取 Cell：

1. 通过字段 Field 来创建一个 Cell

**新创建的 Cell 还未插入 Table**，所以在给 Cell 赋值时都**不会对 Table 中的数据产生任何影响**，此时的 Cell 最大的用途是用来作为 addRecord/addRecords 的参数，只有通过 addRecord/addRecords 插入数据表之后的 Cell 才会和数据表中的数据产生联动。

const cell = await field.createCell(val);

1. 通过 Field 来获取一个 Cell

**这时的 Cell 已经存在于数据表中，因此 Cell 和数据表中的数据是相互联动的**。

const cell = await field.getCell(recordOrId)

1. 通过 Record 获取 Cell

同样的，**这时的 Cell 已经存在于数据表中，因此 Cell 和数据表中的数据是相互联动的**。

const cellList = await record.getCellList();  
const cell = await record.getCellByField(fieldOrId);

更推荐通过 Field 来对 Cell 进行操作，因为会有足够的类型提示，每一个 Cell 的类型定义都会有准确的补全，因此会有更好的语法提示

const attachmentCell = await attachmentField.createCell(imageFileList);  
const singleSelectCell = await singleSelectField.createCell('option1');  
const recordId = await table.addRecord([attachmentCell, singleSelectCell]);

## setValue

设置一个单元格的值，当单元格已经插入 Table 后，会实时改变 Table 中的值

setValue: (val: V) => Promise<void | boolean>;

#### 示例

const cell = await textField.createCell('test');  
cell.setValue('modify value');

## getValue

获取一个单元格的值，当单元格已经插入 Table 后，会获取 Table 中的值

getValue: () => Promise<R>;

#### 示例

const cell = await field.getCell(recordOrId);  
const value = await cell.getValue();

## getFieldId

获取当前单元格所属的字段的 id

getFieldId: () => string;

#### 示例

const fieldId = cell.getFieldId();

# UI 模块

UI 模块承载了用户交互相关的能力，获取方式为：

const ui = bitable.ui;

## switchToTable

切换当前选中的数据表。

switchToTable(tableId: string): Promise<boolean>;

#### 示例

await bitable.ui.switchToTable('table\_id');

## switchToView

切换至指定 Table(数据表) 下指定的 View(视图)，该视图必须从属于数据表，否则会调用失败。

switchToView(tableId: string, viewId: string): Promise<boolean>;

#### 示例

await bitable.ui.switchToView('table\_id', 'view\_id');

## selectRecordIdList

交互式选择记录，调用这个 API 时会在全局唤起选择记录的对话框，如下图所示。用户选择完记录后点击确定，接口返回值会返回已选择记录的记录 ID 列表。

|  |
| --- |
| 选择记录对话框 |

选择记录对话框

selectRecordIdList(tableId: string, viewId: string): Promise<string[]>;

#### 示例

const { tableId, viewId } = await bitable.base.getSelection();  
const recordIdList = await bitable.ui.selectRecordIdList(tableId, viewId);  
const table = await bitable.base.getActiveTable();  
const recordValList = [];  
for (const recordId of recordIdList) {  
 recordValList.push(await table.getRecord(recordId));  
}

## showToast

全局消息提示，调用这个 API 时会在全局提示一条消息，如下图所示。 toast

showToast(options: ShowToastOptions): Promise<boolean>;

相关类型定义如下：

type ShowToastOptions = {  
 toastType?: ToastType,  
 message: string,  
};  
  
enum ToastType {  
 info = 'info',  
 success = 'success',  
 warning = 'warning',  
 error = 'error',  
 loading = 'loading',  
}

#### 示例

await bitable.ui.showToast({  
 toastType: ToastType.info,  
 message: 'hello world'  
})

## showRecordDetailDialog

展示指定数据表指定记录的详情弹窗，默认展示所有字段，支持指定需要展示的字段列表。 showRecordDetailDialog

showRecordDetailDialog(params: { tableId: string, recordId: string, fieldIdList?: string[] } ): Promise<boolean>;

示例

await bitable.ui.showRecordDetailDialog({ tableId: 't\_id', recordId: 'r\_id' });

## getSelectOptionColorInfoList

获取多维表格内置的 **55** 种选择字段中的选项颜色信息，包括选项的背景色，选中态颜色等。

getSelectOptionColorInfoList(): Promise<ISelectOptionColor[]>;

其中 ISelectOptionColor 类型定义如下：

interface ISelectOptionColor {  
 /\*\* 颜色方案id，可用范围为0 - 54 \*/  
 id: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54;  
 /\*\* 同css 16进制颜色值，选项的背景色  
 \* @example '#ff0000' 纯红色  
 \*/  
 bgColor: string;  
 /\*\* 同css 16进制颜色值，文字的颜色  
 \* @example '#ff0000' 纯红色  
 \*/  
 textColor: string;  
 /\*\* 同css 16进制颜色值，表格中删除选项时的x图标的颜色  
 \* @example '#ff0000' 纯红色  
 \*/  
 iconColor: string;  
 /\*\* 同css 16进制颜色值，表格中删除选项时的x图标hover时候的颜色  
 \* @example '#ff0000' 纯红色  
 \*/  
 iconAltColor: string;  
}

#### 示例

const selectOptColorInfo = await bitable.ui.getSelectOptionColorInfoList();