# 什么是uniCloud

uniCloud 是 DCloud 联合阿里云、腾讯云，为 uni-app 的开发者提供的基于 serverless 模式和 js 编程的云开发平台。

**从**[HBuilderX 2.5.8](https://www.dcloud.io/hbuilderx.html)**起支持。目前仅支持阿里云，春节后开放腾讯云。**

### [uniCloud 的价值](https://uniapp.dcloud.io/uniCloud/README?id=unicloud-%e7%9a%84%e4%bb%b7%e5%80%bc)

* 对于程序员，从此你又get一个新技能，用熟悉的js，轻松搞定前后台整体业务。
* 对于开发商：
  1. 开发成本大幅下降。不用再雇佣php或java等服务器工程师，每年至少节省几十万；
  2. 你只需专注于你的业务，其他什么服务器运维、弹性扩容、防DDoS攻击，全都不需要操心;
  3. 对于敏捷性业务，前后端分离的沟通成本实在没有必要。可以考虑按业务负责分工，而不是按前后台分工。

### [uniCloud 的运行原理](https://uniapp.dcloud.io/uniCloud/README?id=unicloud-%e7%9a%84%e8%bf%90%e8%a1%8c%e5%8e%9f%e7%90%86)

**开发和运行流程**

* 开发者在HBuilderX里为项目新建 uniCloud 云环境（可选择阿里云或腾讯云），在云函数目录下编写js代码，上传部署云函数到阿里云或腾讯云的 serverless 环境中。
* 前端代码通过uniCloud.callFunction()方法调用云函数。
* 云函数中可执行js运算、读写云化数据库（NoSQL）、读写存储和CDN、操作网络，给前端返回数据

开发过程，连接DCloud服务器；运行过程是手机端直连阿里云或腾讯云 serverless 环境，不通过DCloud服务器中转。

uniCloud 的底层环境，和微信小程序云开发、支付宝小程序云开发的基建环境相同。功能、性能、稳定性有足够的保障。腾讯云提供基础 serverless 能力，微信团队基于该能力封装了微信云开发，而DCloud团队基于该能力封装了 uniCloud。阿里云类似。

小程序云开发已蔚然成风，微信小程序、支付宝小程序、百度小程序均提供了云开发。微信公布已有50万以上的开发者在使用云开发了，微信自己的生活缴费、乘车码等大日活应用均使用云开发。不过这个流行技术一直无法跨端，它们都只支持各自的小程序。uniCloud解决了跨端问题，让uni-app的所有端应用，都可以使用云开发这个利器。

**uniCloud 基建部分主要包括如下3部分：**

* 云函数：在云端运行的js代码。运行在定制过的node环境中，有良好的性能和强大的功能。serverless环境无需自行加购服务器处理增容，云函数按量付费，不管多大的并发都扛得住（阿里云serverless已经经过了双11的考验）。
* 数据库：可在云函数中读写的、基于 NoSQL 的 JSON 数据库。这种数据库对于前端工程师更自然，不需要学习SQL、不需要理解关系型和设计主键。
* 存储和CDN：不管在前端还是云函数中，都可以操作存储和CDN。在uniCloud提供的安全机制下，可以实现应用前端直传CDN，避免服务器中转的耗时和带宽占用，且不会发生盗传。

uniCloud 目前处于公测阶段，公测期间免费使用。公测结束后的收费也会低于传统云主机租用费用。

### [serverless和云的发展趋势](https://uniapp.dcloud.io/uniCloud/README?id=serverless%e5%92%8c%e4%ba%91%e7%9a%84%e5%8f%91%e5%b1%95%e8%b6%8b%e5%8a%bf)

serverless是目前很火的概念，它是下一代云技术，是真正的“云”。

之前的云服务，让开发者免于购买实体服务器硬件，改为购买虚拟机。但开发者仍然要自己装操作系统、web服务器、数据库，自己处理热备，自己新购服务器来应对高并发，自己抗DDOS攻击...

这不够“云”！

下一代基于serverless的云，真正的把计算、存储的能力进行了云化，开发者只需要按量租用这些计算和存储能力，再也不用买虚拟机，自己装服务器了。

* serverless，让一个不懂服务器运维的开发者，可以只处理自己的业务，再不用关心热备、负载、增容、DDOS等事情。
* serverless，让一个大学生，也可以享受世界最顶级的IT基础设置。

广义的serverless，是泛语言的，PHP、JAVA、Node.js都可以用 serverless。但基于 js 的 serverless，更被业内所看中。

nodejs的性能本身就高于PHP，serverless又降低了js工程师进入后端领域的门槛，所以基于 Node.js 的 serverless 取得更大的发展，并被称为云开发。

在 serverless 成熟后，紧接着出现了小程序云开发。微信、支付宝、百度都上线了自己的云开发，以帮助开发者云端一体的完成业务。根据微信公开的数据，已经有50万开发者在使用微信云开发了。

不过过去这些云开发，都无法跨端。使用它们，就意味着只能做一个端。

uniCloud，在腾讯云、阿里云的 serverless 基础能力之上，实现了真正的跨端。让 uni-app 的开发者使用js开发一云多端的整套业务。并对云开发的功能扩展、开发体验做了很多强化，是下一代开发技术的典型代表。

# 快速上手

## [创建uniCloud项目](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e5%88%9b%e5%bb%baunicloud%e9%a1%b9%e7%9b%ae)

* 在 [HBuilderX 2.5.8+](https://www.dcloud.io/hbuilderx.html" \t "_blank) 新建项目，选择uni-app项目，并勾选启用uniCloud
* 在右侧选择服务供应商（目前仅支持阿里云，春节后会开放腾讯云）



* 对于老的uni-app项目，也可以对项目点右键，菜单中选择“创建uniCloud云开发环境”
* 新建uni-app项目的模板中，有一个Hello uniCloud项目模板，演示了各种云函数的使用。

uniCloud云开发环境创建成功后，项目下会有一个带有云图标的特殊目录，名为“cloudfunctions”。

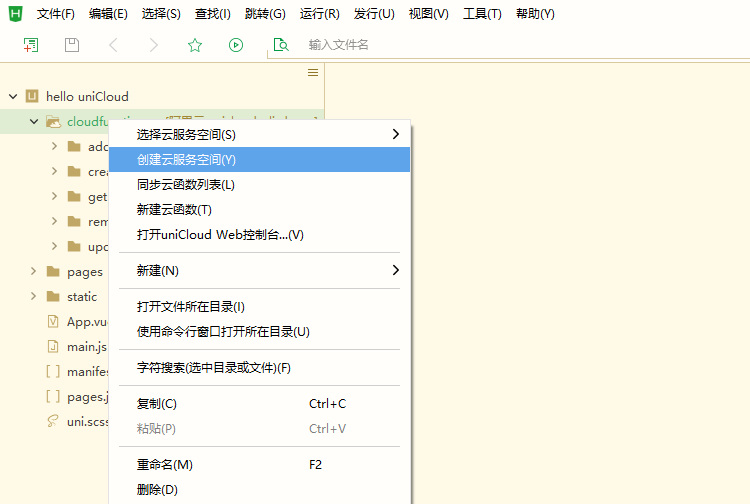
## [创建和绑定服务空间](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e5%88%9b%e5%bb%ba%e5%92%8c%e7%bb%91%e5%ae%9a%e6%9c%8d%e5%8a%a1%e7%a9%ba%e9%97%b4)

项目环境建好后，需要为这个项目选择一个服务空间。如果开发者账户没有实名认证，首先需要实名认证（这是法定要求，也是阿里云、腾讯云等云服务商的要求）。

一个开发者可以拥有多个服务空间，每个服务空间都是一个独立的serverless云环境，不同服务空间之间的云函数、数据库、存储都是隔离的。

服务空间和手机端项目是多对多绑定关系。同账号下，一个项目可以关联到多个服务空间。一个服务空间也可以被多个项目访问。

* 在云函数目录cloudfunctions右键菜单创建服务空间，会打开web控制台[https://unicloud.dcloud.net.cn](https://unicloud.dcloud.net.cn/) 进行创建

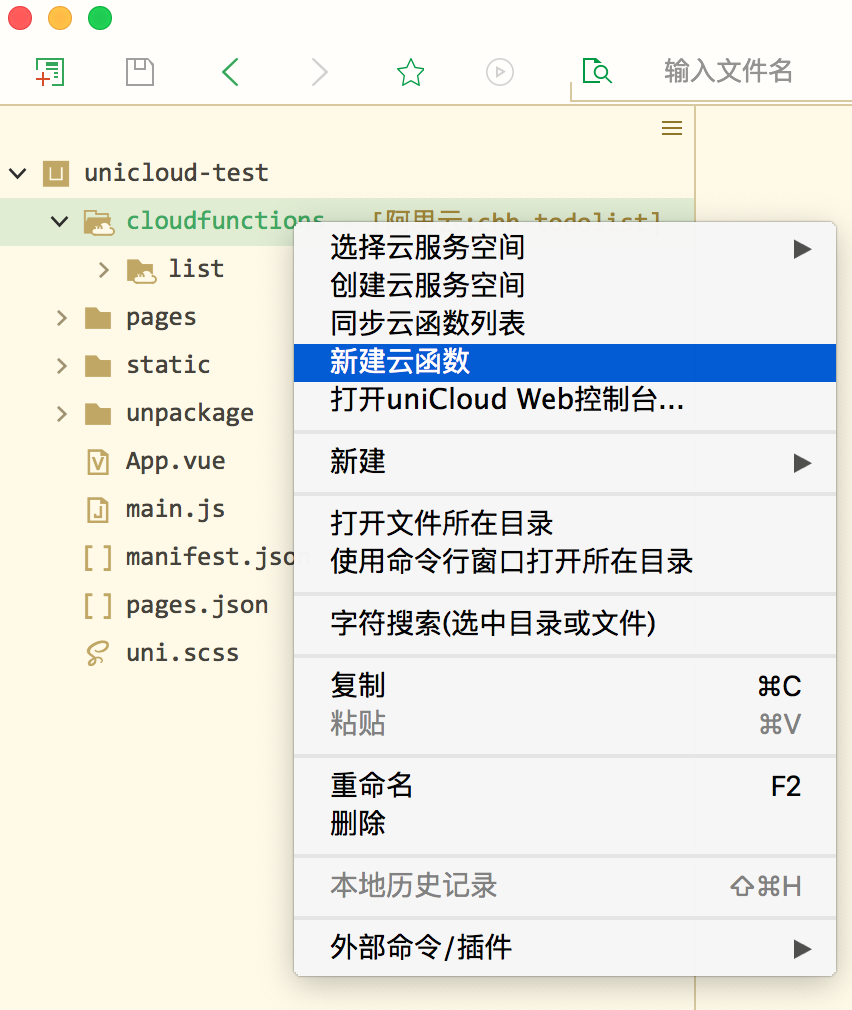


* 创建好服务空间后，对目录cloudfunctions点右键，菜单中点击选择云服务空间，绑定你之前创建的服务空间。

**说明**

* 如果未进行实名认证，会跳转至实名认证页面进行实名认证，等待实名认证审核之后可以开通服务空间
* 创建服务空间可能需要几分钟的时间，可以在控制台查看是否创建完成
* 一个应用，可以在[dev.dcloud.net.cn](https://dev.dcloud.net.cn/)设置协作者（选择应用->设置项目成员），实现多人共同使用一个云服务空间。（需 HBuilderX 2.5.9+）
* 如果一个项目只对应一个服务空间，此时前端可直接使用这个服务空间。如果一个项目绑定了多个服务空间，则需要先做初始化，具体参考：<https://uniapp.dcloud.io/uniCloud/init>

## [创建云函数](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e5%88%9b%e5%bb%ba%e4%ba%91%e5%87%bd%e6%95%b0)

uniCloud项目创建并绑定服务空间后，开发者可以在cloudfunctions目录右键创建云函数。 

创建后会以云函数名称为名生成一个特殊目录，该目录下自动生成index.js，是该云函数的入口文件，不可改名。如果该云函数还需要引入其他js，可在index.js入口文件中引用。

**注意**

* 不同项目使用同一个服务空间时，不可使用同名云函数，可以在uniCloud的web控制台手动删除重名云函数释放函数名。
* 创建时目前版本不校验重名，如果新云函数与服务器上已存在同名云函数，会用新函数覆盖。请务必注意。后续会修复此问题。
* 单个云函数大小限制为10M（包含node\_modules）

## [编写云函数](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e7%bc%96%e5%86%99%e4%ba%91%e5%87%bd%e6%95%b0)

云函数的语法，是在普通的Node.js基础上补充了uniCloud的专用API。可参考API开发文档编写，也可以直接新建项目时选择hello uniCloud模板体验。

HBuilderX为uniCloud开发提供了良好的语法提示和转到定义支持，对于代码中的API，选中并按下F1，也可以直接查看相应的文档。

如下为一个云函数示例

'use strict';

const db = uniCloud.database()

exports.main = async (event, context) => {

//event为客户端上传的参数

const collection = db.collection('unicloud-test') // 获取表'unicloud-test'的集合对象

const res = await collection.limit(10).get() // 获取表中的10条数据，结果为json格式

return res // 返回json给客户端

};

## [运行和调试云函数](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e8%bf%90%e8%a1%8c%e5%92%8c%e8%b0%83%e8%af%95%e4%ba%91%e5%87%bd%e6%95%b0)

编写云函数后，在项目管理器里右键点击该云函数的目录，在弹出菜单中可选择“上传部署云函数”、“上传并运行测试云函数”。如下图所示：

前者仅完成部署，后者会在部署后同时运行，并打印日志出来。

在云函数编辑器里，按Ctrl+r运行快捷键，或点工具栏的运行，还会直接看到上传并运行云函数的快捷指令。Ctrl+r然后回车或选0，即可高效的在控制台看到运行结果和日志输出。

云函数目前无法断点debug，只能打印console.log看日志。





## [手机端调用云函数](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e6%89%8b%e6%9c%ba%e7%ab%af%e8%b0%83%e7%94%a8%e4%ba%91%e5%87%bd%e6%95%b0)

在uni-app的前端代码中，通过uniCloud.callFunction方法调用云函数。详见[callFunction文档](https://uniapp.dcloud.io/uniCloud/functions?id=callfunction)

如下代码中，调用了名为test的云函数，并发送了data的json数据作为上行参数。

// promise方式

uniCloud.callFunction({

name: 'test',

data: { a: 1 }

})

.then(res => {});

// callback方式

uniCloud.callFunction({

name: 'test',

data: { a: 1 },

success(){},

fail(){},

complete(){}

});

## [手机端看日志](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e6%89%8b%e6%9c%ba%e7%ab%af%e7%9c%8b%e6%97%a5%e5%bf%97)

uni-app运行在各端，均可查看手机端日志。额外的，在App真机运行模式下，在HBuilderX的自带控制台也会打印云函数输出的console.log。

**App端真机调试输出云函数日志，如下图所示**

所执行云函数代码

'use strict';

exports.main = async (event, context) => {

console.log('------------');

console.log('云函数日志输出');

console.log('------------');

return {

action: 'log demo'

}

};

日志输出



* 运行到H5，需要在浏览器的控制台查看日志，但仅包含前端日志，不包含云函数内部的console.log。
* 运行到小程序，需要在小程序开发工具的控制台查看日志，但仅包含前端日志，不包含云函数内部的console.log。

目前uniCloud的web控制台还不能查看运行日志，后续会提供此功能。

## [小程序中使用uniCloud](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e5%b0%8f%e7%a8%8b%e5%ba%8f%e4%b8%ad%e4%bd%bf%e7%94%a8unicloud)

小程序中使用uniCloud需要在相应的小程序管理后台设置request合法域名、uploadFile合法域名。这些域名均为阿里云或腾讯云自有域名，并非DCloud所属域名。

| **服务提供商** | **request合法域名** | **uploadFile合法域名** |
| --- | --- | --- |
| 阿里云 | api.bspapp.com | bsppub.oss-cn-shanghai.aliyuncs.com |

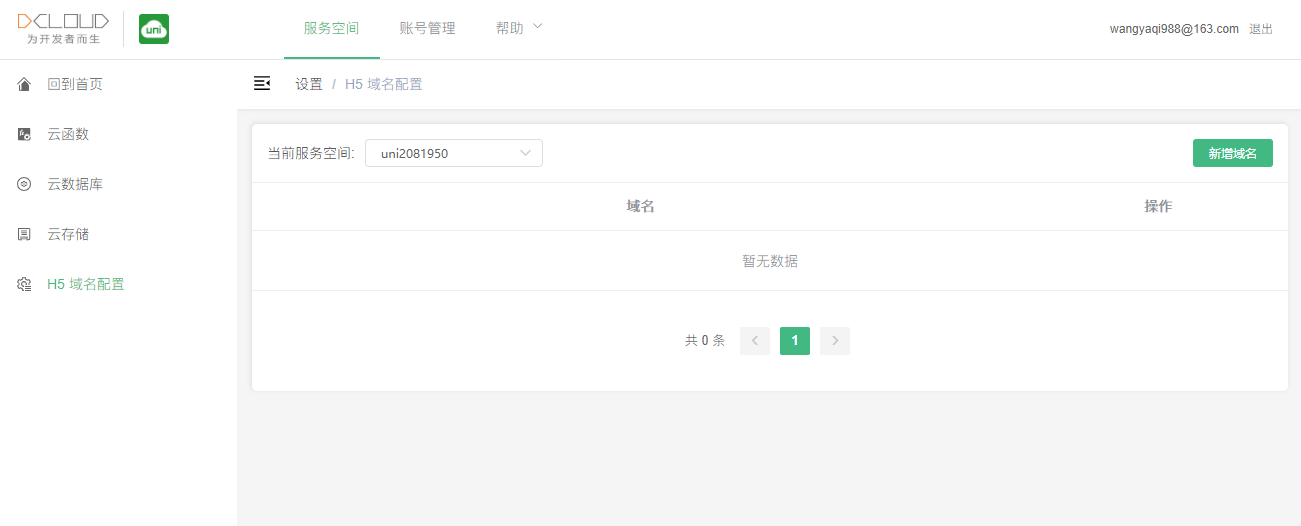
小程序开发工具的真机预览功能，必须添加上述域名白名单，否则无法调用云函数。模拟器的PC端预览、真机调试不受此影响。

## [H5中使用uniCloud](https://uniapp.dcloud.io/uniCloud/quickstart?id=h5%e4%b8%ad%e4%bd%bf%e7%94%a8unicloud)

H5前端js访问云函数，涉及跨域问题，导致前端js无法连接云函数服务器。处理方式如下：。

* 运行到H5端时，使用HBuilderX内置浏览器，可以忽略跨域问题（mac版需2.5.10+）。
* 发行到H5端时，需要在uniCloud后台操作，绑定安全域名，否则会因为跨域问题而无法访问。（在cloudfunctions目录右键可打开uniCloud后台）

**uniCloud后台配置安全域名：**



* 如果运行时，想使用外部浏览器运行，方案如下：
  + 方式1：在uniCloud web控制台绑定测试期的地址为安全域名，如配置：localhost:8080、192.168.0.1:8080
  + 方式2：在外部浏览器安装跨域插件，详见：<https://ask.dcloud.net.cn/article/35267>。要跨域的地址，详见上述文档中小程序配置安全域名章节。

**H5前端页面部署问题**

阿里云Serverless暂未支持H5前端页面部署，需开发者自行准备web服务器，在HBuilderX中点发行菜单，生成H5，将生成的前端文件部署在Nginx等web服务器下。

然后自行注册或使用已有域名，在域名服务商处处理好域名的解析，指向你的Nginx等服务器ip。

最后将该域名通过uniCloud后台配置为安全域名，即可在浏览器中访问。

**m3w.cn二级域名申请**

若为新冠抗疫需紧急上线H5，来不及注册域名，可申请使用DCloud提供的m3w.cn的二级域名，示例：[hellounicloud.m3w.cn](https://hellounicloud.m3w.cn/) 。此时请使用你注册DCloud账户的邮箱向service@dcloud.io发邮件申请，提供你的appid、计划使用的二级域名名称、解析的ip地址、应用的使用用途。

如果不发布H5，使用uniCloud不需要自己申请或准备域名。App和小程序里直接调用云函数即可。

## [使用db\_init.json初始化项目数据库](https://uniapp.dcloud.io/uniCloud/quickstart?id=%e4%bd%bf%e7%94%a8db_initjson%e5%88%9d%e5%a7%8b%e5%8c%96%e9%a1%b9%e7%9b%ae%e6%95%b0%e6%8d%ae%e5%ba%93)

自HBuilderX 2.5.11起uniCloud提供了db\_init.json来方便开发者快速进行数据库的初始化操作。

**使用说明**

* 在cloudfucntions目录右键即可创建db\_init.json，
* 在db\_init.json上右键初始化数据库。

**db\_init.json形式如下**

{

"collection\_test": { // 集合（表名）

"data": [ // 数据

{

"\_id": "da51bd8c5e37ac14099ea43a2505a1a5",

"name": "tom"

}

],

"index": [{ // 索引

"IndexName": "index\_a", // 索引名称

"MgoKeySchema": { // 索引规则

"MgoIndexKeys": [{

"Name": "index", // 索引字段

"Direction": 1 // 索引方向，1：ASC-升序，-1：DESC-降序 ，暂时使用数字后续此字段会修改为字符串形式

}],

"MgoIsUnique": false // 索引是否唯一

}

}]

}

}

**Bug&Tips**

* 早期阿里云的云函数的初次冷启动较慢，表现为某个云函数第一次被调用时联网时间较长，可能要5秒左右。第二次即可正常。此问题阿里云已修复，需重新上传部署云函数后生效。
* web控制台网址：[http://unicloud.dcloud.net.cn](http://unicloud.dcloud.net.cn/)，在HX中对云函数目录点右键，或者在帮助菜单中，均有入口链接。

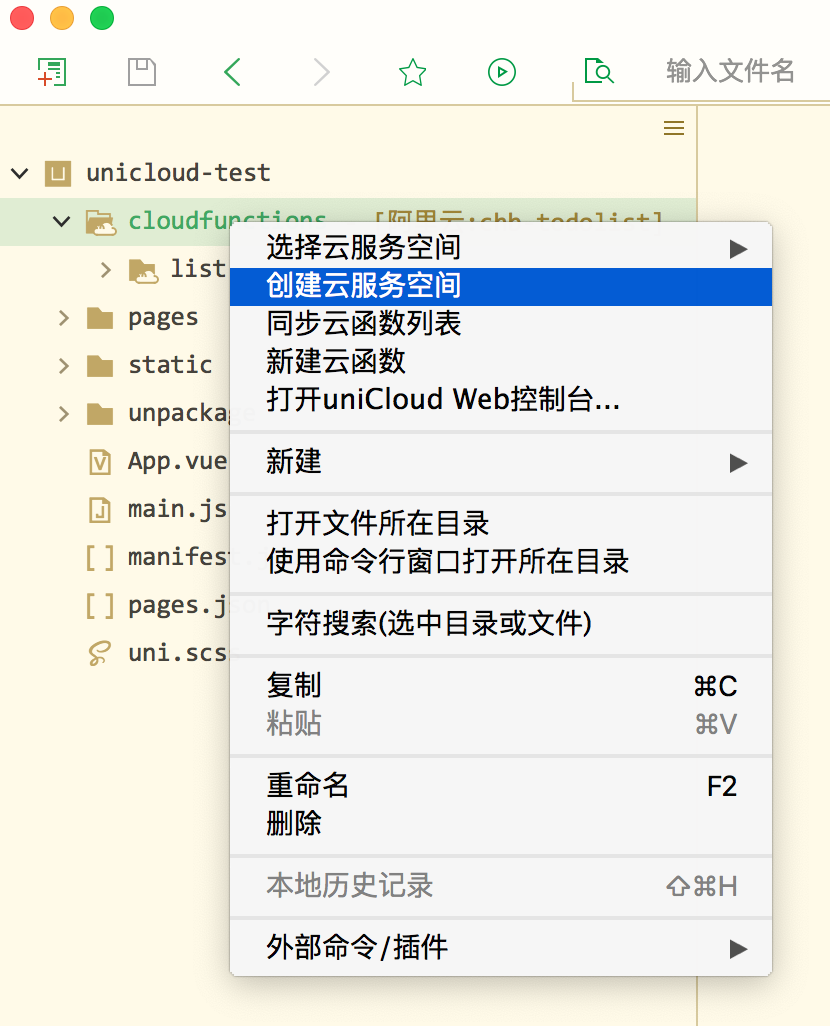
# 基本概念

## 服务空间

一个服务空间对应一整套独立的云开发资源，包括数据库、存储空间、云函数等资源。服务空间之间彼此隔离。

每个服务空间都有一个全局唯一的space ID。未来会支持一个应用同时连接多个服务空间，届时需要在代码中声明space ID以告诉代码要连接哪个云服务空间。

开发者可在 HBuilderX 中新建服务空间，如下：



或者在uniCloud的web控制台[https://unicloud.dcloud.net.cn](https://unicloud.dcloud.net.cn/" \t "_blank) 创建服务空间。

开发者需先为项目绑定服务空间，然后才能上传云函数、操作服务空间下的数据库、存储等资源。

## 数据库

uniCloud提供了一个 JSON 格式的文档型数据库，数据库中的每条记录都是一个 JSON 格式的对象。一个数据库可以有多个集合（相当于关系型数据中的表），集合可看做一个 JSON 数组，数组中的每个对象就是一条记录，记录的格式是 JSON 对象。

关系型数据库和 JSON 文档型数据库的概念对应关系如下表：

| **关系型** | **JSON 文档型** |
| --- | --- |
| 数据库 database | 数据库 database |
| 表 table | 集合 collection |
| 行 row | 记录 record / doc |
| 列 column | 字段 field |

更多云数据库介绍参考[规范](https://uniapp.dcloud.io/uniCloud/cf-database)。

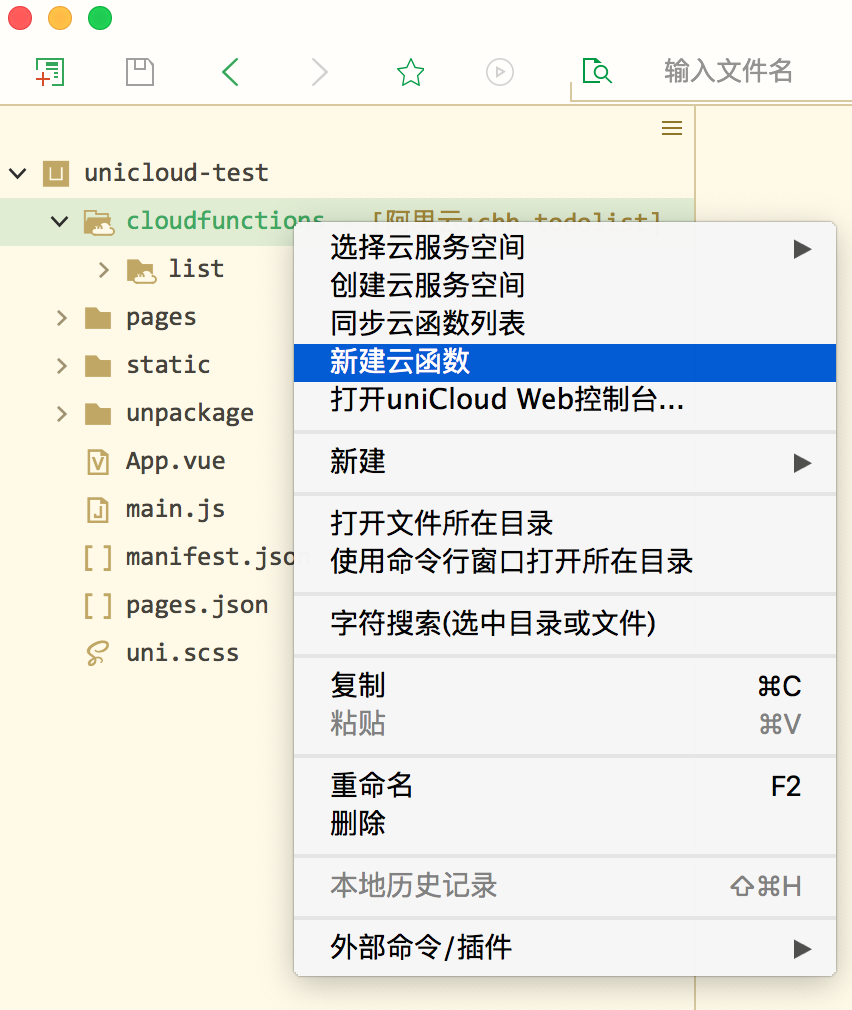
## 云函数

云函数即在云端（服务器端）运行的函数。

开发者无需购买、搭建服务器，只需编写函数代码并部署到云端即可在客户端（App/H5/小程序等）调用，同时云函数之间也可互相调用。

一个云函数的写法与一个在本地定义的 JavaScript 方法无异，代码运行在云端 Node.js 中。当云函数被客户端调用时，定义的代码会被放在 Node.js 运行环境中执行。

开发者可以如在 Node.js 环境中使用 JavaScript 一样在云函数中进行网络请求等操作，而且还可以通过云函数服务端 SDK 搭配使用多种服务，比如使用云函数 SDK 中提供的数据库和存储 API 进行数据库和存储的操作，这部分可参考数据库和存储后端 API 文档。

开发者可在HBuiderX中，在cloudfunctions目录上右键、新建云函数，如下： 

云函数修改后，需上传到云端，方可生效。

更多云函数介绍参考[规范](https://uniapp.dcloud.io/uniCloud/cf-functions)。

# 云函数

### [简介](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e7%ae%80%e4%bb%8b)

云函数是运行在云端的 JavaScript 代码，和普通的Node.js开发一样，熟悉Node.js的开发者可以直接上手。

如下是将传入的两个参数求和并返回客户端的云函数代码示例：

'use strict';

exports.main = async (event, context) => {

//event为客户端上传的参数

return {

sum:event.a + event.b

}

}

云函数的传入参数有两个，一个是event对象，一个是context对象。event指的是触发云函数的事件，当客户端调用云函数时，event就是客户端调用云函数时传入的参数。context 对象包含了此处调用的调用信息和运行状态，可以用它来了解服务运行的情况。uniCloud会自动将客户端的操作系统（os）、运行平台（platform）、应用信息（appid）等注入context中，开发者可通过context获取每次调用的上下文，如下是一个示例：

'use strict';

exports.main = async (event, context) => {

//event为客户端上传的参数

...

//context中可获取客户端调用的上下文

let os = context.OS //客户端操作系统，返回值：android、ios 等

let platform = context.PLATFORM //运行平台，返回值为 mp-weixin、app-plus等

let appid = context.APPID // manifest.json中配置的appid

... //其它业务代码

}

### [访问数据库](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e8%ae%bf%e9%97%ae%e6%95%b0%e6%8d%ae%e5%ba%93)

云函数中支持访问本服务空间下的数据库，调用方式详见[规范](https://uniapp.dcloud.io/uniCloud/cf-database)

### [访问HTTP服务](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e8%ae%bf%e9%97%aehttp%e6%9c%8d%e5%8a%a1)

uniCloud提供了uniCloud.httpclient供开发者使用。无需额外依赖，就可以请求任何 HTTP 和 HTTPS 协议的 Web 服务。uniCloud.httpclient返回的是一个[urllib实例](https://github.com/node-modules/urllib)。

**uniCloud.httpclient.request(URL,requestOptions)**

**requestOptions参数说明**

| **参数名** | **类型** | **是否必填** | **默认值** | **说明** |
| --- | --- | --- | --- | --- |
| method | String | - | GET | HTTP 请求方法, 默认为：GET. 可选值： GET, POST, DELETE, PUT |
| data | Object | - | - | 发送的数据 |
| dataAsQueryString | Boolean | - | true | 是否强制转换data为queryString |
| content | String | Buffer | - | - | 手动设置请求的payload，设置后会忽略data |
| files | Array<ReadStream|Buffer|String> | Object | ReadStream | Buffer | String | - | - | 上传的文件，设置后将会使用 multipart/form-data 格式。如果未设置method，将会自动将method设置为POST |
| contentType | String | - | - | 上传数据的格式，设为json会自动在header内设置Content-Type: application/json |
| nestedQuerystring | Boolean | - | - | 转换data为queryString时默认不支持嵌套Object，此选项设置为true则支持转换嵌套Object |
| dataType | String | - | - | 返回的数据格式 |
| headers | Object | - | - | 请求头 |
| timeout | Number | Array | - | - | 超时时间设置。设置为数组时第一项为请求超时，第二项为返回超时。设置为数字时相当于同时设置请求超时和返回超时，即timeout:3000效果等于timeouut:[3000,3000] |

**示例代码**

const res = await uniCloud.httpclient.request(apiUrl, {

method: 'POST',

data: {

test: 'testValue'

},

dataType: 'json'

})

console.log(res)

### [使用npm](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e4%bd%bf%e7%94%a8npm)

在云函数中我们可以引入第三方依赖来帮助我们更快的开发。云函数的运行环境是 Node.js，因此我们可以使用 npm 安装第三方依赖。

注意：鉴于阿里云的限制，目前仅支持全量上传云函数（整个 node\_modules文件夹全部上传），因此提醒大家，精简依赖，否则可能会每次上传时间很慢，影响开发体验。

Tips:

* 目前每个云函数上传包大小限制为10M。

### [客户端调用云函数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e5%ae%a2%e6%88%b7%e7%ab%af%e8%b0%83%e7%94%a8%e4%ba%91%e5%87%bd%e6%95%b0)

前端代码（H5前端、App、小程序），不再执行uni.request联网，而是通过uniCloud.callFunction调用云函数，callFunction定义如下：

#### [请求参数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e8%af%b7%e6%b1%82%e5%8f%82%e6%95%b0)

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| name | String | 是 | 云函数名称 |
| data | Object | 否 | 客户端需要传递的参数 |

#### [响应参数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e5%93%8d%e5%ba%94%e5%8f%82%e6%95%b0)

| **字段** | **类型** | **说明** |
| --- | --- | --- |
| result | Object | 云函数执行结果 |
| requestId | String | 请求序列号，用于错误排查 |

#### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81)

// promise方式

uniCloud.callFunction({

name: 'test',

data: { a: 1 }

})

.then(res => {});

// callback方式

uniCloud.callFunction({

name: 'test',

data: { a: 1 },

success(){},

fail(){},

complete(){}

});

目前仅支持客户端调用云函数，即将支持云函数调用云函数。

### [注意事项](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e6%b3%a8%e6%84%8f%e4%ba%8b%e9%a1%b9)

云函数是运行在云端的代码，运行环境由云服务器弹性调配，这是和传统Node.js应用很大的区别。

换言之，云函数每次执行的宿主环境（可简单理解为虚拟机或服务器硬件）可能相同，也可能不同，因此传统Node.js开发中将部分信息存储本地硬盘或内存的方案就不再适合，建议通过云数据库或云存储的方案替代。

另一方面，鉴于云函数的弹性调配机制，为提高运行性能，部分云厂商在一定时间周期内会复用相同的云函数实例，此时为避免数据污染，建议使用无状态的风格编写函数代码。

以如下代码为例，count作为全局变量，当多次调用该云函数时，可能会出现变量累加的情况（实例未复用时，每次返回0，若实例被复用，则可能返回1、2、3等各种意外情况）

**云函数中使用的时区是 UTC+0，而不是 UTC+8，在云函数中使用时间时需特别注意。**

let count = 0;

module.exports = async (event) => {

return count++

//此示例为错误示例

//云函数实例未复用时，每次返回0

//若实例被复用，则可能返回1、2、3等各种意外情况

}

## [云函数中调用云函数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e4%ba%91%e5%87%bd%e6%95%b0%e4%b8%ad%e8%b0%83%e7%94%a8%e4%ba%91%e5%87%bd%e6%95%b0)

用法同客户端调用云函数，不支持callback形式

#### [请求参数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e8%af%b7%e6%b1%82%e5%8f%82%e6%95%b0-1)

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| name | String | 是 | 云函数名称。 |
| data | Object | 否 | 云函数参数。 |

#### [响应参数](https://uniapp.dcloud.io/uniCloud/cf-functions?id=%e5%93%8d%e5%ba%94%e5%8f%82%e6%95%b0-1)

| **字段** | **类型** | **必备** | **说明** |
| --- | --- | --- | --- |
| errCode | String | 否 | 状态码，操作成功则不返回。 |
| errMsg | String | 否 | 错误描述。 |
| result | Object | 否 | 云函数执行结果。 |
| requestId | String | 否 | 请求序列号，用于错误排查。 |

**示例代码**

let callFunctionResult = await uniCloud.callFunction({

name: "test",

data: { a: 1 }

})

# 云数据库

## [云数据库简介](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e4%ba%91%e6%95%b0%e6%8d%ae%e5%ba%93%e7%ae%80%e4%bb%8b)

uniCloud提供了一个 JSON 格式的文档型数据库，数据库中的每条记录都是一个 JSON 格式的对象。一个数据库可以有多个集合（相当于关系型数据中的表），集合可看做一个 JSON 数组，数组中的每个对象就是一条记录，记录的格式是 JSON 对象。

关系型数据库和 JSON 文档型数据库的概念对应关系如下表：

| **关系型** | **JSON 文档型** |
| --- | --- |
| 数据库 database | 数据库 database |
| 表 table | 集合 collection |
| 行 row | 记录 record / doc |
| 列 column | 字段 field |

uniCloud云函数中可访问云数据库。

鉴于安全问题，暂不支持客户端直接访问数据库。

**阿里云使用的mongoDB数据库版本为3.4，腾讯云使用的版本是4.0。此差异可能会导致本文档内的部分功能不能在阿里云使用，我们会进行标注，如果发现有遗漏欢迎向我们反馈**

## [获取数据库的引用](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%8e%b7%e5%8f%96%e6%95%b0%e6%8d%ae%e5%ba%93%e7%9a%84%e5%bc%95%e7%94%a8)

const db = uniCloud.database();

## [获取集合的引用](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%8e%b7%e5%8f%96%e9%9b%86%e5%90%88%e7%9a%84%e5%bc%95%e7%94%a8)

// 获取 `user` 集合的引用

const collection = db.collection('user');

### [集合 Collection](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e9%9b%86%e5%90%88-collection)

通过 db.collection(name) 可以获取指定集合的引用，在集合上可以进行以下操作

| **类型** | **接口** | **说明** |
| --- | --- | --- |
| 写 | add | 新增记录（触发请求） |
| 计数 | count | 获取符合条件的记录条数 |
| 读 | get | 获取集合中的记录，如果有使用 where 语句定义查询条件，则会返回匹配结果集 (触发请求) |
| 引用 | doc | 获取对该集合中指定 id 的记录的引用 |
| 查询条件 | where | 通过指定条件筛选出匹配的记录，可搭配查询指令（eq, gt, in, ...）使用 |
|  | skip | 跳过指定数量的文档，常用于分页，传入 offset |
|  | orderBy | 排序方式 |
|  | limit | 返回的结果集(文档数量)的限制，有默认值和上限值 |
|  | field | 指定需要返回的字段 |

查询及更新指令用于在 where 中指定字段需满足的条件，指令可通过 db.command 对象取得。

### [记录 Record / Document](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%ae%b0%e5%bd%95-record-document)

通过 db.collection(collectionName).doc(docId) 可以获取指定集合上指定 id 的记录的引用，在记录上可以进行以下操作

| **接口** | **说明** |  |
| --- | --- | --- |
| 写 | set | 覆写记录 |
|  | update | 局部更新记录(触发请求) |
|  | remove | 删除记录(触发请求) |
| 读 | get | 获取记录(触发请求) |

### [查询筛选指令 Query Command](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9f%a5%e8%af%a2%e7%ad%9b%e9%80%89%e6%8c%87%e4%bb%a4-query-command)

以下指令挂载在 db.command 下

| **类型** | **接口** | **说明** |
| --- | --- | --- |
| 比较运算 | eq | 字段 == |
|  | neq | 字段 != |
|  | gt | 字段 > |
|  | gte | 字段 >= |
|  | lt | 字段 < |
|  | lte | 字段 <= |
|  | in | 字段值在数组里 |
|  | nin | 字段值不在数组里 |
| 逻辑运算 | and | 表示需同时满足指定的所有条件 |
|  | or | 表示需同时满足指定条件中的至少一个 |

### [字段更新指令 Update Command](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%ad%97%e6%ae%b5%e6%9b%b4%e6%96%b0%e6%8c%87%e4%bb%a4-update-command)

以下指令挂载在 db.command 下

| **类型** | **接口** | **说明** |
| --- | --- | --- |
| 字段 | set | 设置字段值 |
|  | remove | 删除字段 |
|  | inc | 加一个数值，原子自增 |
|  | mul | 乘一个数值，原子自乘 |
|  | push | 数组类型字段追加尾元素，支持数组 |
|  | pop | 数组类型字段删除尾元素，支持数组 |
|  | shift | 数组类型字段删除头元素，支持数组 |
|  | unshift | 数组类型字段追加头元素，支持数组 |

## [支持的数据类型](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%94%af%e6%8c%81%e7%9a%84%e6%95%b0%e6%8d%ae%e7%b1%bb%e5%9e%8b)

数据库提供以下几种数据类型：

* String：字符串
* Number：数字
* Object：对象
* Array：数组
* Bool：布尔值
* GeoPoint：地理位置点
* GeoLineStringL: 地理路径
* GeoPolygon: 地理多边形
* GeoMultiPoint: 多个地理位置点
* GeoMultiLineString: 多个地理路径
* GeoMultiPolygon: 多个地理多边形
* Date：时间
* Null

**注意**

* 阿里云数据库在存入emoji表情时会导致uniCloud控制台无法获取数据列表，目前阿里正在处理此问题，开发者可以先自行过滤一下

以下对几个特殊的数据类型做个补充说明

### [时间 Date](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%97%b6%e9%97%b4-date)

Date 类型用于表示时间，精确到毫秒，可以用 JavaScript 内置 Date 对象创建。需要特别注意的是，用此方法创建的时间是客户端时间，不是服务端时间。如果需要使用服务端时间，应该用 API 中提供的 serverDate 对象来创建一个服务端当前时间的标记，当使用了 serverDate 对象的请求抵达服务端处理时，该字段会被转换成服务端当前的时间，更棒的是，我们在构造 serverDate 对象时还可通过传入一个有 offset 字段的对象来标记一个与当前服务端时间偏移 offset 毫秒的时间，这样我们就可以达到比如如下效果：指定一个字段为服务端时间往后一个小时。

那么当我们需要使用客户端时间时，存放 Date 对象和存放毫秒数是否是一样的效果呢？不是的，我们的数据库有针对日期类型的优化，建议大家使用时都用 Date 或 serverDate 构造时间对象。

//服务端当前时间

new db.serverDate()

//服务端当前时间加1S

new db.serverDate({

offset: 1000

})

**Tips**

* 使用阿里云作为服务提供商时，如需存入日期类型，需要2020-02-10T04:59:05.579Z形式，即可以在云函数中使用new Date().toISOString()得到。

### [地理位置](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%9c%b0%e7%90%86%e4%bd%8d%e7%bd%ae)

**阿里云暂不支持地理位置类型**

### [Null](https://uniapp.dcloud.io/uniCloud/cf-database?id=null)

Null 相当于一个占位符，表示一个字段存在但是值为空。

## [新增文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%96%b0%e5%a2%9e%e6%96%87%e6%a1%a3)

方法1： collection.add(data)

示例：

| **参数** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| data | object | 是 | {\_id: '10001', 'name': 'Ben'} \_id 非必填 |

collection.add({

name: 'Ben'

}).then((res) => {

});

**Tips**

* 云服务商为阿里云时，若集合不存在，调用add方法会自动创建集合

方法2： collection.doc().set(data)

也可通过 set 方法新增一个文档，需先取得文档引用再调用 set 方法。 如果文档不存在，set 方法会创建一个新文档。

collection.doc().set({

name: "Hey"

});

## [查询文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9f%a5%e8%af%a2%e6%96%87%e6%a1%a3)

支持 where()、limit()、skip()、orderBy()、get()、update()、field()、count() 等操作。

只有当调用get() update()时才会真正发送请求。 注：默认取前100条数据，最大取前100条数据。

### [添加查询条件](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%b7%bb%e5%8a%a0%e6%9f%a5%e8%af%a2%e6%9d%a1%e4%bb%b6)

collection.where() 参数

设置过滤条件 where 可接收对象作为参数，表示筛选出拥有和传入对象相同的 key-value 的文档。比如筛选出所有类型为计算机的、内存为 8g 的商品：

db.collection('goods').where({

category: 'computer',

type: {

memory: 8,

}

})

如果要表达更复杂的查询，可使用高级查询指令，比如筛选出所有内存大于 8g 的计算机商品：

const dbCmd = db.command // 取指令

db.collection('goods').where({

category: 'computer',

type: {

memory: dbCmd.gt(8), // 表示大于 8

}

})

where 可以使用正则表达式来查询文档，比如一下示例查询所有name字段以ABC开头的用户

db.collection('user').where({

name: new RegExp('^ABC')

})

### [获取查询数量](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%8e%b7%e5%8f%96%e6%9f%a5%e8%af%a2%e6%95%b0%e9%87%8f)

collection.count()

参数

db.collection('goods').where({

category: 'computer',

type: {

memory: 8,

}

}).count().then(function(res) {

})

响应参数

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| code | string | 否 | 状态码，操作成功则不返回 |
| message | string | 否 | 错误描述 |
| total | Integer | 否 | 计数结果 |
| requestId | string | 否 | 请求序列号，用于错误排查 |

### [设置记录数量](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%ae%be%e7%bd%ae%e8%ae%b0%e5%bd%95%e6%95%b0%e9%87%8f)

collection.limit()

参数说明

| **参数** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| value | Integer | 是 | 限制展示的数值 |

使用示例

collection.limit(1).get().then(function(res) {

});

### [设置起始位置](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%ae%be%e7%bd%ae%e8%b5%b7%e5%a7%8b%e4%bd%8d%e7%bd%ae)

collection.skip()

参数说明

| **参数** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| value | Integer | 是 | 跳过展示的数据 |

使用示例

collection.skip(4).get().then(function(res) {

});

### [对结果排序](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%af%b9%e7%bb%93%e6%9e%9c%e6%8e%92%e5%ba%8f)

collection.orderBy()

参数说明

| **参数** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| field | string | 是 | 排序的字段 |
| orderType | string | 是 | 排序的顺序，升序(asc) 或 降序(desc) |

使用示例

collection.orderBy("name", "asc").get().then(function(res) {

});

### [指定返回字段](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%8c%87%e5%ae%9a%e8%bf%94%e5%9b%9e%e5%ad%97%e6%ae%b5)

collection.field()

参数说明

| **参数** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| - | object | 是 | 要过滤的字段，不返回传false，返回传true |

使用示例

collection.field({ 'age': true })

备注：只能指定要返回的字段或者不要返回的字段。即{'a': true, 'b': false}是一种错误的参数格式

### [查询指令](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9f%a5%e8%af%a2%e6%8c%87%e4%bb%a4)

#### [eq](https://uniapp.dcloud.io/uniCloud/cf-database?id=eq)

表示字段等于某个值。eq 指令接受一个字面量 (literal)，可以是 number, boolean, string, object, array。

比如筛选出所有自己发表的文章，除了用传对象的方式：

const myOpenID = 'xxx'

db.collection('articles').where({

\_openid: myOpenID

})

还可以用指令：

const dbCmd = db.command

const myOpenID = 'xxx'

db.collection('articles').where({

\_openid: dbCmd.eq(openid)

})

注意 eq 指令比对象的方式有更大的灵活性，可以用于表示字段等于某个对象的情况，比如：

// 这种写法表示匹配 stat.publishYear == 2018 且 stat.language == 'zh-CN'

db.collection('articles').where({

stat: {

publishYear: 2018,

language: 'zh-CN'

}

})

// 这种写法表示 stat 对象等于 { publishYear: 2018, language: 'zh-CN' }

const dbCmd = db.command

db.collection('articles').where({

stat: dbCmd.eq({

publishYear: 2018,

language: 'zh-CN'

})

})

#### [neq](https://uniapp.dcloud.io/uniCloud/cf-database?id=neq)

字段不等于。neq 指令接受一个字面量 (literal)，可以是 number, boolean, string, object, array。

如筛选出品牌不为 X 的计算机：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

brand: dbCmd.neq('X')

},

})

#### [gt](https://uniapp.dcloud.io/uniCloud/cf-database?id=gt)

字段大于指定值。

如筛选出价格大于 2000 的计算机：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

price: dbCmd.gt(2000)

})

#### [gte](https://uniapp.dcloud.io/uniCloud/cf-database?id=gte)

字段大于或等于指定值。

#### [lt](https://uniapp.dcloud.io/uniCloud/cf-database?id=lt)

字段小于指定值。

#### [lte](https://uniapp.dcloud.io/uniCloud/cf-database?id=lte)

字段小于或等于指定值。

#### [in](https://uniapp.dcloud.io/uniCloud/cf-database?id=in)

字段值在给定的数组中。

筛选出内存为 8g 或 16g 的计算机商品：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

memory: dbCmd.in([8, 16])

}

})

#### [nin](https://uniapp.dcloud.io/uniCloud/cf-database?id=nin)

字段值不在给定的数组中。

筛选出内存不是 8g 或 16g 的计算机商品：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

memory: dbCmd.nin([8, 16])

}

})

#### [and](https://uniapp.dcloud.io/uniCloud/cf-database?id=and)

表示需同时满足指定的两个或以上的条件。

如筛选出内存大于 4g 小于 32g 的计算机商品：

流式写法：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

memory: dbCmd.gt(4).and(dbCmd.lt(32))

}

})

前置写法：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

memory: dbCmd.and(dbCmd.gt(4), dbCmd.lt(32))

}

})

#### [or](https://uniapp.dcloud.io/uniCloud/cf-database?id=or)

表示需满足所有指定条件中的至少一个。如筛选出价格小于 4000 或在 6000-8000 之间的计算机：

流式写法：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

price:dbCmd.lt(4000).or(dbCmd.gt(6000).and(dbCmd.lt(8000)))

}

})

前置写法：

const dbCmd = db.command

db.collection('goods').where({

category: 'computer',

type: {

price: dbCmd.or(dbCmd.lt(4000), dbCmd.and(dbCmd.gt(6000), dbCmd.lt(8000)))

}

})

如果要跨字段 “或” 操作：(如筛选出内存 8g 或 cpu 3.2 ghz 的计算机)

const dbCmd = db.command

db.collection('goods').where(dbCmd.or(

{

type: {

memory: dbCmd.gt(8)

}

},

{

type: {

cpu: 3.2

}

}

))

### [正则表达式查询](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%ad%a3%e5%88%99%e8%a1%a8%e8%be%be%e5%bc%8f%e6%9f%a5%e8%af%a2)

#### [db.RegExp](https://uniapp.dcloud.io/uniCloud/cf-database?id=dbregexp)

根据正则表达式进行筛选

例如下面可以筛选出 version 字段开头是 "数字+s" 的记录，并且忽略大小写：

// 可以直接使用正则表达式

db.collection('articles').where({

version: /^\ds/i

})

// 或者

db.collection('articles').where({

version: new db.RegExp({

regex: '^\\ds' // 正则表达式为 /^\ds/，转义后变成 '^\\ds'

options: 'i' // i表示忽略大小写

})

})

## [删除文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%88%a0%e9%99%a4%e6%96%87%e6%a1%a3)

**方式1 通过指定文档ID删除**

collection.doc(\_id).remove()

// 清理全部数据

collection.get()

.then((res) => {

const promiseList = res.data.map(document => {

return collection.doc(document.id).remove();

});

Promise.all(promiseList);

})

.catch((e) => {

});

**方式2 条件查找文档然后直接批量删除**

collection.where().remove()

// 删除字段a的值大于2的文档

const dbCmd = db.command

collection.where({

a: dbCmd.gt(2)

}).remove().then(function(res) {

})

## [更新文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9b%b4%e6%96%b0%e6%96%87%e6%a1%a3)

### [更新指定文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9b%b4%e6%96%b0%e6%8c%87%e5%ae%9a%e6%96%87%e6%a1%a3)

collection.doc().update()

collection.doc('doc-id').update({

name: "Hey",

count: {

fav: 1

}

});

// 更新前

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 0,

follow: 0

}

}

// 更新后

{

\_id: 'xxx',

name: "Hey",

count: {

fav: 1,

follow: 0

}

}

### [更新文档，如果不存在则创建](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9b%b4%e6%96%b0%e6%96%87%e6%a1%a3%ef%bc%8c%e5%a6%82%e6%9e%9c%e4%b8%8d%e5%ad%98%e5%9c%a8%e5%88%99%e5%88%9b%e5%bb%ba)

collection.doc().set()

**注意：**

* 此方法会覆写已有字段，需注意与update表现不同，比如以下示例执行set之后follow字段会被删除

collection.doc('doc-id').set({

name: "Hey",

count: {

fav: 1

}

}).then(function(res) {

});

// 更新前

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 0,

follow: 0

}

}

// 更新后

{

\_id: 'xxx',

name: "Hey",

count: {

fav: 1

}

}

### [批量更新文档](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%89%b9%e9%87%8f%e6%9b%b4%e6%96%b0%e6%96%87%e6%a1%a3)

collection.update()

const dbCmd = db.command

collection.where({name: dbCmd.eq('hey')}).update({

age: 18,

}).then(function(res) {

});

### [更新指令](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9b%b4%e6%96%b0%e6%8c%87%e4%bb%a4)

#### [set](https://uniapp.dcloud.io/uniCloud/cf-database?id=set)

更新指令。用于设定字段等于指定值。这种方法相比传入纯 JS 对象的好处是能够指定字段等于一个对象：

const dbCmd = db.command

db.collection('photo').doc('doc-id').update({

count: dbCmd.set({

fav: 1,

follow: 1

})

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 0,

follow: 0

}

}

// 更新后

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 1,

follow: 1

}

}

#### [inc](https://uniapp.dcloud.io/uniCloud/cf-database?id=inc)

更新指令。用于指示字段自增某个值，这是个原子操作，使用这个操作指令而不是先读数据、再加、再写回的好处是：

1. 原子性：多个用户同时写，对数据库来说都是将字段加一，不会有后来者覆写前者的情况
2. 减少一次网络请求：不需先读再写

之后的 mul 指令同理。

如给收藏的商品数量加一：

const dbCmd = db.command

db.collection('user').where({

\_id: 'my-doc-id'

}).update({

count: {

fav: dbCmd.inc(1)

}

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 0,

follow: 0

}

}

// 更新后

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 1,

follow: 0

}

}

#### [mul](https://uniapp.dcloud.io/uniCloud/cf-database?id=mul)

更新指令。用于指示字段自乘某个值。

以下示例将count内的fav字段乘10

const dbCmd = db.command

db.collection('user').where({

\_id: 'my-doc-id'

}).update({

count: {

fav: dbCmd.mul(10)

}

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 2,

follow: 0

}

}

// 更新后

{

\_id: 'xxx',

name: "Hello",

count: {

fav: 20,

follow: 0

}

}

#### [remove](https://uniapp.dcloud.io/uniCloud/cf-database?id=remove)

更新指令。用于表示删除某个字段。如某人删除了自己一条商品评价中的评分：

const dbCmd = db.command

db.collection('comments').doc('comment-id').update({

rating: dbCmd.remove()

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

rating: 5,

comment: 'xxxx'

}

// 更新后

{

\_id: 'xxx',

comment: 'xxxx'

}

#### [push](https://uniapp.dcloud.io/uniCloud/cf-database?id=push)

向数组尾部追加元素，支持传入单个元素或数组

const dbCmd = db.command

db.collection('comments').doc('comment-id').update({

// users: dbCmd.push('aaa')

users: dbCmd.push(['c', 'd'])

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

users: ['a','b']

}

// 更新后

{

\_id: 'xxx',

users: ['a','b','c','d']

}

#### [pop](https://uniapp.dcloud.io/uniCloud/cf-database?id=pop)

删除数组尾部元素

const dbCmd = db.command

db.collection('comments').doc('comment-id').update({

users: dbCmd.pop()

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

users: ['a','b']

}

// 更新后

{

\_id: 'xxx',

users: ['a']

}

#### [unshift](https://uniapp.dcloud.io/uniCloud/cf-database?id=unshift)

向数组头部添加元素，支持传入单个元素或数组。使用同push

const dbCmd = db.command

db.collection('comments').doc('comment-id').update({

// users: dbCmd.push('aaa')

users: dbCmd.unshift(['c', 'd'])

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

users: ['a','b']

}

// 更新后

{

\_id: 'xxx',

users: ['c','d','a','b']

}

#### [shift](https://uniapp.dcloud.io/uniCloud/cf-database?id=shift)

删除数组头部元素。使用同pop

const dbCmd = db.command

db.collection('comments').doc('comment-id').update({

users: dbCmd.shift()

}).then(function(res) {

})

// 更新前

{

\_id: 'xxx',

users: ['a','b']

}

// 更新后

{

\_id: 'xxx',

users: ['b']

}

## [聚合操作](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%81%9a%e5%90%88%e6%93%8d%e4%bd%9c)

获取数据库集合的聚合操作实例

db.collection('scores').aggregate()

### [addFields](https://uniapp.dcloud.io/uniCloud/cf-database?id=addfields)

聚合阶段。添加新字段到输出的记录。经过 addFields 聚合阶段，输出的所有记录中除了输入时带有的字段外，还将带有 addFields 指定的字段。

**API 说明**

addFields 等同于同时指定了所有已有字段和新增字段的 project 阶段。

**addFields 的形式如下：**

addFields({

<新字段>: <表达式>

})

addFields 可指定多个新字段，每个新字段的值由使用的表达式决定。

如果指定的新字段与原有字段重名，则新字段的值会覆盖原有字段的值。注意 addFields 不能用来给数组字段添加元素。

**示例 1：连续两次 addFields**

假设集合 scores 有如下记录：

{

\_id: 1,

student: "Maya",

homework: [ 10, 5, 10 ],

quiz: [ 10, 8 ],

extraCredit: 0

}

{

\_id: 2,

student: "Ryan",

homework: [ 5, 6, 5 ],

quiz: [ 8, 8 ],

extraCredit: 8

}

应用两次 addFields，第一次增加两个字段分别为 homework 和 quiz 的和值，第二次增加一个字段再基于上两个和值求一次和值。

const $ = db.command.aggregate

db.collection('scores').aggregate()

.addFields({

totalHomework: $.sum('$homework'),

totalQuiz: $.sum('$quiz')

})

.addFields({

totalScore: $.add(['$totalHomework', '$totalQuiz', '$extraCredit'])

})

.end()

返回结果如下：

{

"\_id" : 1,

"student" : "Maya",

"homework" : [ 10, 5, 10 ],

"quiz" : [ 10, 8 ],

"extraCredit" : 0,

"totalHomework" : 25,

"totalQuiz" : 18,

"totalScore" : 43

}

{

"\_id" : 2,

"student" : "Ryan",

"homework" : [ 5, 6, 5 ],

"quiz" : [ 8, 8 ],

"extraCredit" : 8,

"totalHomework" : 16,

"totalQuiz" : 16,

"totalScore" : 40

}

**示例 2：在嵌套记录里增加字段**

可以用点表示法在嵌套记录里增加字段。假设 vehicles 集合含有如下记录：

{ \_id: 1, type: "car", specs: { doors: 4, wheels: 4 } }

{ \_id: 2, type: "motorcycle", specs: { doors: 0, wheels: 2 } }

{ \_id: 3, type: "jet ski" }

可以用如下操作在 specs 字段下增加一个新的字段 fuel\_type，值都设为固定字符串 unleaded：

db.collection('vehicles').aggregate()

.addFields({

'spec.fuel\_type': 'unleaded'

})

.end()

返回结果如下：

{ \_id: 1, type: "car",

specs: { doors: 4, wheels: 4, fuel\_type: "unleaded" } }

{ \_id: 2, type: "motorcycle",

specs: { doors: 0, wheels: 2, fuel\_type: "unleaded" } }

{ \_id: 3, type: "jet ski",

specs: { fuel\_type: "unleaded" } }

**示例 3：设置字段值为另一个字段**

可以通过 $ 加字段名组成的字符串作为值的表达式来设置字段的值为另一个字段的值。

同样用上一个集合示例，可以用如下操作添加一个字段 vehicle\_type，将其值设置为 type 字段的值：

db.collection('vehicles').aggregate()

.addFields({

vehicle\_type: '$type'

})

.end()

返回结果如下：

{ \_id: 1, type: "car", vehicle\_type: "car",

specs: { doors: 4, wheels: 4, fuel\_type: "unleaded" } }

{ \_id: 2, type: "motorcycle", vehicle\_type: "motorcycle",

specs: { doors: 0, wheels: 2, fuel\_type: "unleaded" } }

{ \_id: 3, type: "jet ski", vehicle\_type: "jet ski",

specs: { fuel\_type: "unleaded" } }

### [bucket](https://uniapp.dcloud.io/uniCloud/cf-database?id=bucket)

聚合阶段。将输入记录根据给定的条件和边界划分成不同的组，每组即一个 bucket。

**API 说明**

每组分别作为一个记录输出，包含一个以下界为值的 \_id 字段和一个以组中记录数为值的 count 字段。count 在没有指定 output 的时候是默认输出的。

bucket 只会在组内有至少一个记录的时候输出。

**bucket 的形式如下：**

bucket({

groupBy: <expression>,

boundaries: [<lowerbound1>, <lowerbound2>, ...],

default: <literal>,

output: {

<output1>: <accumulator expr>,

...

<outputN>: <accumulator expr>

}

})

groupBy 是一个用以决定分组的表达式，会应用在各个输入记录上。可以用 $ 前缀加上要用以分组的字段路径来作为表达式。除非用 default 指定了默认值，否则每个记录都需要包含指定的字段，且字段值必须在 boundaries 指定的范围之内。

boundaries 是一个数组，每个元素分别是每组的下界。必须至少指定两个边界值。数组值必须是同类型递增的值。

default 可选，指定之后，没有进入任何分组的记录将都进入一个默认分组，这个分组记录的 \_id 即由 default 决定。default 的值必须小于 boundaries 中的最小值或大于等于其中的最大值。default 的值可以与 boundaries 元素值类型不同。

output 可选，用以决定输出记录除了 \_id 外还要包含哪些字段，各个字段的值必须用累加器表达式指定。当 output 指定时，默认的 count 是不会被默认输出的，必须手动指定：

output: {

count: $.sum(1),

...

<outputN>: <accumulator expr>

}

使用 bucket 需要满足以下至少一个条件，否则会抛出错误：

每一个输入记录应用 groupBy 表达式获取的值都必须是一个在 boundaries 内的值

指定一个 default 值，该值在 boundaries 以外，或与 boundaries 元素的值不同的类型。

**示例**

假设集合 items 有如下记录：

{

\_id: "1",

price: 10

}

{

\_id: "2",

price: 50

}

{

\_id: "3",

price: 20

}

{

\_id: "4",

price: 80

}

{

\_id: "5",

price: 200

}

对上述记录进行分组，将 [0, 50) 分为一组，[50, 100) 分为一组，其他分为一组：

const $ = db.command.aggregate

db.collection('items').aggregate()

.bucket({

groupBy: '$price',

boundaries: [0, 50, 100],

default: 'other',

output: {

count: $.sum(),

ids: $.push('$\_id')

}

})

.end()

返回结果如下：

[

{

"\_id": 0,

"count": 2,

"ids": [

"1",

"3"

]

},

{

"\_id": 50,

"count": 2,

"ids": [

"2",

"4"

]

},

{

"\_id": "other",

"count": 22,

"ids": [

"5"

]

}

]

### [bucketAuto](https://uniapp.dcloud.io/uniCloud/cf-database?id=bucketauto)

聚合阶段。将输入记录根据给定的条件划分成不同的组，每组即一个 bucket。与 bucket 的其中一个不同之处在于无需指定 boundaries，bucketAuto 会自动尝试将记录尽可能平均的分散到每组中。

**API 说明** 每组分别作为一个记录输出，包含一个以包含组中最大值和最小值两个字段的对象为值的 \_id 字段和一个以组中记录数为值的 count 字段。count 在没有指定 output 的时候是默认输出的。

**bucketAuto 的形式如下：**

bucketAuto({

groupBy: <expression>,

buckets: <number>,

granularity: <string>,

output: {

<output1>: <accumulator expr>,

...

<outputN>: <accumulator expr>

}

})

groupBy 是一个用以决定分组的表达式，会应用在各个输入记录上。可以用 $ 前缀加上要用以分组的字段路径来作为表达式。除非用 default 指定了默认值，否则每个记录都需要包含指定的字段，且字段值必须在 boundaries 指定的范围之内。

buckets 是一个用于指定划分组数的正整数。

granularity 是可选枚举值字符串，用于保证自动计算出的边界符合给定的规则。这个字段仅可在所有 groupBy 值都是数字并且没有 NaN 的情况下使用。枚举值包括：R5、R10、R20、R40、R80、1-2-5、E6、E12、E24、E48、E96、E192、POWERSOF2。

output 可选，用以决定输出记录除了 \_id 外还要包含哪些字段，各个字段的值必须用累加器表达式指定。当 output 指定时，默认的 count 是不会被默认输出的，必须手动指定：

output: {

count: $.sum(1),

...

<outputN>: <accumulator expr>

}

在以下情况中，输出的分组可能会小于给定的组数：

输入记录数少于分组数

* groupBy 计算得到的唯一值少于分组数
* granularity 的间距少于分组数
* granularity 不够精细以至于不能平均分配到各组

**granularity 详细说明**

granularity 用于保证边界值属于一个给定的数字序列。

**Renard 序列**

Renard 序列是以 10 的 5 / 10 / 20 / 40 / 80 次方根来推导的、在 1.0 到 10.0 (如果是 R80 则是 10.3) 之间的数字序列。

设置 granularity 为 R5 / R10 / R20 / R40 / R80 就把边界值限定在序列内。如果 groupBy 的值不在 1.0 到 10.0 (如果是 R80 则是 10.3) 内，则序列数字会自动乘以 10。

**E 序列**

E 序列是以 10 的 6 / 12 / 24 / 48 / 96 / 192 次方跟来推导的、带有一个特定误差的、在 1.0 到 10.0 之间的数字序列。

**1-2-5 序列**

1-2-5 序列 表现与三值 Renard 序列一样。

**2的次方序列**

由 2 的各次方组成的序列数字。

**示例**

假设集合 items 有如下记录：

{

\_id: "1",

price: 10.5

}

{

\_id: "2",

price: 50.3

}

{

\_id: "3",

price: 20.8

}

{

\_id: "4",

price: 80.2

}

{

\_id: "5",

price: 200.3

}

对上述记录进行自动分组，分成三组：

const $ = db.command.aggregate

db.collection('items').aggregate()

.bucket({

groupBy: '$price',

buckets: 3,

})

.end()

返回结果如下：

{

"\_id": {

"min": 10.5,

"max": 50.3

},

"count": 2

}

{

"\_id": {

"min": 50.3,

"max": 200.3

},

"count": 2

}

{

"\_id": {

"min": 200.3,

"max": 200.3

},

"count": 1

}

### [count](https://uniapp.dcloud.io/uniCloud/cf-database?id=count)

聚合阶段。计算上一聚合阶段输入到本阶段的记录数，输出一个记录，其中指定字段的值为记录数。

**API 说明**

**count 的形式如下：**

count(<string>)

是输出记录数的字段的名字，不能是空字符串，不能以 $ 开头，不能包含 . 字符。

count 阶段等同于 group + project 的操作：

const $ = db.command.aggregate

db.collection('items').aggregate()

.group({

\_id: null,

count: $.sum(1),

})

.project({

\_id: 0,

})

.end()

上述操作会输出一个包含 count 字段的记录。

**示例**

假设集合 items 有如下记录：

{

\_id: "1",

price: 10.5

}

{

\_id: "2",

price: 50.3

}

{

\_id: "3",

price: 20.8

}

{

\_id: "4",

price: 80.2

}

{

\_id: "5",

price: 200.3

}

找出价格大于 50 的记录数：

const $ = db.command.aggregate

db.collection('items').aggregate()

.match({

price: $.gt(50)

})

.count('expensiveCount')

.end()

返回结果如下：

{

"expensiveCount": 3

}

### [group](https://uniapp.dcloud.io/uniCloud/cf-database?id=group)

聚合阶段。将输入记录按给定表达式分组，输出时每个记录代表一个分组，每个记录的 \_id 是区分不同组的 key。输出记录中也可以包括累计值，将输出字段设为累计值即会从该分组中计算累计值。

**API 说明**

**group 的形式如下：**

group({

\_id: <expression>,

<field1>: <accumulator1>,

...

<fieldN>: <accumulatorN>

})

\_id 参数是必填的，如果填常量则只有一组。其他字段是可选的，都是累计值，用 $.sum 等累计器(const $ = db.command.aggregate)，但也可以使用其他表达式。

累计器必须是以下操作符之一：

详细使用方法见[累计器操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%b4%af%e8%ae%a1%e5%99%a8%e6%93%8d%e4%bd%9c%e7%ac%a6)

| **操作符** | **说明** |
| --- | --- |
| addToSet | 向数组中添加值，如果数组中已存在该值，不执行任何操作 |
| avg | 返回一组集合中，指定字段对应数据的平均值 |
| sum | 计算并且返回一组字段所有数值的总和 |
| first | 返回指定字段在一组集合的第一条记录对应的值。仅当这组集合是按照某种定义排序（ sort ）后，此操作才有意义。 |
| last | 返回指定字段在一组集合的最后一条记录对应的值。仅当这组集合是按照某种定义排序（ sort ）后，此操作才有意义。 |
| max | 返回一组数值的最大值 |
| min | 返回一组数值的最小值 |
| push | 在 group 阶段，返回一组中表达式指定列与对应的值，一起组成的数组 |
| stdDevPop | 返回一组字段对应值的标准差 |
| stdDevSamp | 计算输入值的样本标准偏差。如果输入值代表数据总体，或者不概括更多的数据，请改用 db.command.aggregate.stdDevPop |
| mergeObjects | 将多个文档合并为单个文档 |

**内存限制**

该阶段有 100M 内存使用限制。

**示例 1：按字段值分组**

假设集合 avatar 有如下记录：

{

\_id: "1",

alias: "john",

region: "asia",

scores: [40, 20, 80],

coins: 100

}

{

\_id: "2",

alias: "arthur",

region: "europe",

scores: [60, 90],

coins: 20

}

{

\_id: "3",

alias: "george",

region: "europe",

scores: [50, 70, 90],

coins: 50

}

{

\_id: "4",

alias: "john",

region: "asia",

scores: [30, 60, 100, 90],

coins: 40

}

{

\_id: "5",

alias: "george",

region: "europe",

scores: [20],

coins: 60

}

{

\_id: "6",

alias: "john",

region: "asia",

scores: [40, 80, 70],

coins: 120

}

const $ = db.command.aggregate

db.collection('avatar').aggregate()

.group({

\_id: '$alias',

num: $.sum(1)

})

.end()

返回结果如下：

{

"\_id": "john",

"num": 3

}

{

"\_id": "authur",

"num": 1

}

{

"\_id": "george",

"num": 2

}

**示例 2：按多个值分组**

可以给 \_id 传入记录的方式按多个值分组。还是沿用上面的示例数据，按各个区域（region）获得相同最高分（score）的来分组，并求出各组虚拟币（coins）的总量：

const $ = db.command.aggregate

db.collection('avatar').aggregate()

.group({

\_id: {

region: '$region',

maxScore: $.max('$scores')

},

totalCoins: $.sum('$coins')

})

.end()

返回结果如下：

{

"\_id": {

"region": "asia",

"maxScore": 80

},

"totalCoins": 220

}

{

"\_id": {

"region": "asia",

"maxScore": 100

},

"totalCoins": 100

}

{

"\_id": {

"region": "europe",

"maxScore": 90

},

"totalCoins": 70

}

{

"\_id": {

"region": "europe",

"maxScore": 20

},

"totalCoins": 60

}

### [limit](https://uniapp.dcloud.io/uniCloud/cf-database?id=limit)

聚合阶段。限制输出到下一阶段的记录数。

**示例**

假设集合 items 有如下记录：

{

\_id: "1",

price: 10

}

{

\_id: "2",

price: 50

}

{

\_id: "3",

price: 20

}

{

\_id: "4",

price: 80

}

{

\_id: "5",

price: 200

}

返回价格大于 20 的记录的最小的两个记录：

const $ = db.command.aggregate

db.collection('items').aggregate()

.match({

price: $.gt(20)

})

.sort({

price: 1,

})

.limit(2)

.end()

返回结果如下：

{

"\_id": "3",

"price": 20

}

{

"\_id": "4",

"price": 80

}

### [lookup](https://uniapp.dcloud.io/uniCloud/cf-database?id=lookup)

聚合阶段。联表查询。与同个数据库下的一个指定的集合做 left outer join(左外连接)。对该阶段的每一个输入记录，lookup 会在该记录中增加一个数组字段，该数组是被联表中满足匹配条件的记录列表。lookup 会将连接后的结果输出给下个阶段。

**API 说明**

lookup 有两种使用方式

#### [相等匹配](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%9b%b8%e7%ad%89%e5%8c%b9%e9%85%8d)

将输入记录的一个字段和被连接集合的一个字段进行相等匹配时，采用以下定义：

lookup({

from: <要连接的集合名>,

localField: <输入记录的要进行相等匹配的字段>,

foreignField: <被连接集合的要进行相等匹配的字段>,

as: <输出的数组字段名>

})

**参数详细说明**

| **参数字段** | **说明** |
| --- | --- |
| from | 要进行连接的另外一个集合的名字 |
| localField | 当前流水线的输入记录的字段名，该字段将被用于与 from 指定的集合的 foreignField 进行相等匹配。如果输入记录中没有该字段，则该字段的值在匹配时会被视作 null |
| foreignField | 被连接集合的字段名，该字段会被用于与 localField 进行相等匹配。如果被连接集合的记录中没有该字段，该字段的值将在匹配时被视作 null |
| as | 指定连接匹配出的记录列表要存放的字段名，这个数组包含的是匹配出的来自 from 集合的记录。如果输入记录中本来就已有该字段，则该字段会被覆写 |

这个操作等价于以下伪 SQL 操作：

SELECT \*, <output array field>

FROM collection

WHERE <output array field> IN (SELECT \*

FROM <collection to join>

WHERE <foreignField>= <collection.localField>);

**例子：**

* 指定一个相等匹配条件
* 对数组字段应用相等匹配
* 组合 mergeObjects 应用相等匹配

#### [自定义连接条件、拼接子查询](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%87%aa%e5%ae%9a%e4%b9%89%e8%bf%9e%e6%8e%a5%e6%9d%a1%e4%bb%b6%e3%80%81%e6%8b%bc%e6%8e%a5%e5%ad%90%e6%9f%a5%e8%af%a2)

**此用法阿里云暂不支持**

如果需要指定除相等匹配之外的连接条件，或指定多个相等匹配条件，或需要拼接被连接集合的子查询结果，那可以使用如下定义：

lookup({

from: <要连接的集合名>,

let: { <变量1>: <表达式1>, ..., <变量n>: <表达式n> },

pipeline: [ <在要连接的集合上进行的流水线操作> ],

as: <输出的数组字段名>

})

**参数详细说明**

| **参数字段** | **说明** |
| --- | --- |
| from | 要进行连接的另外一个集合的名字 |
| let | 可选。指定在 pipeline 中可以使用的变量，变量的值可以引用输入记录的字段，比如 let: { userName: '$name' } 就代表将输入记录的 name 字段作为变量 userName 的值。在 pipeline 中无法直接访问输入记录的字段，必须通过 let 定义之后才能访问，访问的方式是在 expr 操作符中用 $$变量名 的方式访问，比如 $$userName。 |
| pipeline | 指定要在被连接集合中运行的聚合操作。如果要返回整个集合，则该字段取值空数组 []。在 pipeline 中无法直接访问输入记录的字段，必须通过 let 定义之后才能访问，访问的方式是在 expr 操作符中用 $$变量名 的方式访问，比如 $$userName。 |
| as | 指定连接匹配出的记录列表要存放的字段名，这个数组包含的是匹配出的来自 from 集合的记录。如果输入记录中本来就已有该字段，则该字段会被覆写 |

该操作等价于以下伪 SQL 语句：

SELECT \*, <output array field>

FROM collection

WHERE <output array field> IN (SELECT <documents as determined from the pipeline>

FROM <collection to join>

WHERE <pipeline> );

**例子**

* 指定多个连接条件
* 拼接被连接集合的子查询

**示例**

**指定一个相等匹配条件**

假设 orders 集合有以下记录：

[

{"\_id":4,"book":"novel 1","price":30,"quantity":2},

{"\_id":5,"book":"science 1","price":20,"quantity":1},

{"\_id":6}

]

books 集合有以下记录：

[

{"\_id":"book1","author":"author 1","category":"novel","stock":10,"time":1564456048486,"title":"novel 1"},

{"\_id":"book3","author":"author 3","category":"science","stock":30,"title":"science 1"},

{"\_id":"book4","author":"author 3","category":"science","stock":40,"title":"science 2"},

{"\_id":"book2","author":"author 2","category":"novel","stock":20,"title":"novel 2"},

{"\_id":"book5","author":"author 4","category":"science","stock":50,"title":null},

{"\_id":"book6","author":"author 5","category":"novel","stock":"60"}

]

以下聚合操作可以通过一个相等匹配条件连接 orders 和 books 集合，匹配的字段是 orders 集合的 book 字段和 books 集合的 title 字段：

const db = cloud.database()

db.collection('orders').aggregate()

.lookup({

from: 'books',

localField: 'book',

foreignField: 'title',

as: 'bookList',

})

.end()

.then(res => console.log(res))

.catch(err => console.error(err))

结果：

[

{

"\_id": 4,

"book": "novel 1",

"price": 30,

"quantity": 2,

"bookList": [

{

"\_id": "book1",

"title": "novel 1",

"author": "author 1",

"category": "novel",

"stock": 10

}

]

},

{

"\_id": 5,

"book": "science 1",

"price": 20,

"quantity": 1,

"bookList": [

{

"\_id": "book3",

"category": "science",

"title": "science 1",

"author": "author 3",

"stock": 30

}

]

},

{

"\_id": 6,

"bookList": [

{

"\_id": "book5",

"category": "science",

"author": "author 4",

"stock": 50,

"title": null

},

{

"\_id": "book6",

"author": "author 5",

"stock": "60",

"category": "novel"

}

]

}

]

对数组字段应用相等匹配 假设 authors 集合有以下记录：

[

{"\_id": 1, "name": "author 1", "intro": "Two-time best-selling sci-fiction novelist"},

{"\_id": 3, "name": "author 3", "intro": "UCB assistant professor"},

{"\_id": 4, "name": "author 4", "intro": "major in CS"}

]

books 集合有以下记录：

[

{"\_id":"book1","authors":["author 1"],"category":"novel","stock":10,"time":1564456048486,"title":"novel 1"},

{"\_id":"book3","authors":["author 3", "author 4"],"category":"science","stock":30,"title":"science 1"},

{"\_id":"book4","authors":["author 3"],"category":"science","stock":40,"title":"science 2"}

]

以下操作获取作者信息及他们分别发表的书籍，使用了 lookup 操作匹配 authors 集合的 name 字段和 books 集合的 authors 数组字段：

const db = cloud.database()

db.collection('authors').aggregate()

.lookup({

from: 'books',

localField: 'name',

foreignField: 'authors',

as: 'publishedBooks',

})

.end()

.then(res => console.log(res))

.catch(err => console.error(err))

结果

[

{

"\_id": 1,

"intro": "Two-time best-selling sci-fiction novelist",

"name": "author 1",

"publishedBooks": [

{

"\_id": "book1",

"title": "novel 1",

"category": "novel",

"stock": 10,

"authors": [

"author 1"

]

}

]

},

{

"\_id": 3,

"name": "author 3",

"intro": "UCB assistant professor",

"publishedBooks": [

{

"\_id": "book3",

"category": "science",

"title": "science 1",

"stock": 30,

"authors": [

"author 3",

"author 4"

]

},

{

"\_id": "book4",

"title": "science 2",

"category": "science",

"stock": 40,

"authors": [

"author 3"

]

}

]

},

{

"\_id": 4,

"intro": "major in CS",

"name": "author 4",

"publishedBooks": [

{

"\_id": "book3",

"category": "science",

"title": "science 1",

"stock": 30,

"authors": [

"author 3",

"author 4"

]

}

]

}

]

**组合 mergeObjects 应用相等匹配**

假设 orders 集合有以下记录：

[

{"\_id":4,"book":"novel 1","price":30,"quantity":2},

{"\_id":5,"book":"science 1","price":20,"quantity":1},

{"\_id":6}

]

books 集合有以下记录：

[

{"\_id":"book1","author":"author 1","category":"novel","stock":10,"time":1564456048486,"title":"novel 1"},

{"\_id":"book3","author":"author 3","category":"science","stock":30,"title":"science 1"},

{"\_id":"book4","author":"author 3","category":"science","stock":40,"title":"science 2"},

{"\_id":"book2","author":"author 2","category":"novel","stock":20,"title":"novel 2"},

{"\_id":"book5","author":"author 4","category":"science","stock":50,"title":null},

{"\_id":"book6","author":"author 5","category":"novel","stock":"60"}

]

以下操作匹配 orders 的 book 字段和 books 的 title 字段，并将 books 匹配结果直接 merge 到 orders 记录中。

var db = cloud.database()

var $ = db.command.aggregate

db.collection('orders').aggregate()

.lookup({

from: "books",

localField: "book",

foreignField: "title",

as: "bookList"

})

.replaceRoot({

newRoot: $.mergeObjects([ $.arrayElemAt(['$bookList', 0]), '$$ROOT' ])

})

.project({

bookList: 0

})

.end()

.then(res => console.log(res))

.catch(err => console.error(err))

结果

[

{

"\_id": 4,

"title": "novel 1",

"author": "author 1",

"category": "novel",

"stock": 10,

"book": "novel 1",

"price": 30,

"quantity": 2

},

{

"\_id": 5,

"category": "science",

"title": "science 1",

"author": "author 3",

"stock": 30,

"book": "science 1",

"price": 20,

"quantity": 1

},

{

"\_id": 6,

"category": "science",

"author": "author 4",

"stock": 50,

"title": null

}

]

**指定多个连接条件**

假设 orders 集合有以下记录：

[

{"\_id":4,"book":"novel 1","price":300,"quantity":20},

{"\_id":5,"book":"science 1","price":20,"quantity":1}

]

books 集合有以下记录：

[

{"\_id":"book1","author":"author 1","category":"novel","stock":10,"time":1564456048486,"title":"novel 1"},

{"\_id":"book3","author":"author 3","category":"science","stock":30,"title":"science 1"}

]

以下操作连接 orders 和 books 集合，要求两个条件：

* orders 的 book 字段与 books 的 title 字段相等
* orders 的 quantity 字段大于或等于 books 的 stock 字段
* const db = cloud.database()
* const $ = db.command.aggregate
* db.collection('orders').aggregate()
* .lookup({
* from: 'books',
* let: {
* order\_book: '$book',
* order\_quantity: '$quantity'
* },
* pipeline: $.pipeline()
* .match(\_.expr($.and([
* $.eq(['$title', '$$order\_book']),
* $.gte(['$stock', '$$order\_quantity'])
* ])))
* .project({
* \_id: 0,
* title: 1,
* author: 1,
* stock: 1
* })
* .done(),
* as: 'bookList',
* })
* .end()
* .then(res => console.log(res))

.catch(err => console.error(err))

结果：

[

{

"\_id": 4,

"book": "novel 1",

"price": 300,

"quantity": 20,

"bookList": []

},

{

"\_id": 5,

"book": "science 1",

"price": 20,

"quantity": 1,

"bookList": [

{

"title": "science 1",

"author": "author 3",

"stock": 30

}

]

}

]

**拼接被连接集合的子查询**

假设 orders 集合有以下记录：

[

{"\_id":4,"book":"novel 1","price":30,"quantity":2},

{"\_id":5,"book":"science 1","price":20,"quantity":1}

]

books 集合有以下记录：

[

{"\_id":"book1","author":"author 1","category":"novel","stock":10,"time":1564456048486,"title":"novel 1"},

{"\_id":"book3","author":"author 3","category":"science","stock":30,"title":"science 1"},

{"\_id":"book4","author":"author 3","category":"science","stock":40,"title":"science 2"}

]

在每条输出记录上加上一个数组字段，该数组字段的值是对 books 集合的一个查询语句的结果：

const db = cloud.database()

const $ = db.command.aggregate

db.collection('orders').aggregate()

.lookup({

from: 'books',

let: {

order\_book: '$book',

order\_quantity: '$quantity'

},

pipeline: $.pipeline()

.match({

author: 'author 3'

})

.project({

\_id: 0,

title: 1,

author: 1,

stock: 1

})

.done(),

as: 'bookList',

})

.end()

.then(res => console.log(res))

.catch(err => console.error(err))

结果

[

{

"\_id": 4,

"book": "novel 1",

"price": 30,

"quantity": 20,

"bookList": [

{

"title": "science 1",

"author": "author 3",

"stock": 30

},

{

"title": "science 2",

"author": "author 3",

"stock": 40

}

]

},

{

"\_id": 5,

"book": "science 1",

"price": 20,

"quantity": 1,

"bookList": [

{

"title": "science 1",

"author": "author 3",

"stock": 30

},

{

"title": "science 2",

"author": "author 3",

"stock": 40

}

]

}

]

### [match](https://uniapp.dcloud.io/uniCloud/cf-database?id=match)

聚合阶段。根据条件过滤文档，并且把符合条件的文档传递给下一个流水线阶段。

**API 说明**

**match 的形式如下：**

match(<查询条件>)

查询条件与普通查询一致，可以用普通查询操作符，注意 match 阶段和其他聚合阶段不同，不可使用聚合操作符，只能使用查询操作符。

// 直接使用字符串

match({

name: 'Tony Stark'

})

// 使用操作符

const \_ = db.command

match({

age: \_.gt(18)

})

**示例**

假设集合 articles 有如下记录：

{ "\_id" : "1", "author" : "stark", "score" : 80 }

{ "\_id" : "2", "author" : "stark", "score" : 85 }

{ "\_id" : "3", "author" : "bob", "score" : 60 }

{ "\_id" : "4", "author" : "li", "score" : 55 }

{ "\_id" : "5", "author" : "jimmy", "score" : 60 }

{ "\_id" : "6", "author" : "li", "score" : 94 }

{ "\_id" : "7", "author" : "justan", "score" : 95 }

**匹配**

下面是一个直接匹配的例子：

db.collection('articles')

.aggregate()

.match({

author: 'stark'

})

.end()

这里的代码尝试找到所有 author 字段是 stark 的文章，那么匹配如下：

{ "\_id" : "1", "author" : "stark", "score" : 80 }

{ "\_id" : "2", "author" : "stark", "score" : 85 }

**计数**

match 过滤出文档后，还可以与其他流水线阶段配合使用。

比如下面这个例子，我们使用 group 进行搭配，计算 score 字段大于 80 的文档数量：

const \_ = db.command

const $ = \_.aggregate

db.collection('articles')

.aggregate()

.match({

score: \_.gt(80)

})

.group({

\_id: null,

count: $.sum(1)

})

.end()

返回值如下：

{ "\_id" : null, "count" : 3 }

### [project](https://uniapp.dcloud.io/uniCloud/cf-database?id=project)

聚合阶段。把指定的字段传递给下一个流水线，指定的字段可以是某个已经存在的字段，也可以是计算出来的新字段。

**API 说明**

**project 的形式如下：**

project({

<表达式>

})

表达式可以有以下格式：

| **格式** | **说明** |
| --- | --- |
| <字段>: <1 或 true> | 指定包含某个已有字段 |
| \_id: <0 或 false> | 舍弃 \_id 字段 |
| <字段>: <表达式> | 加入一个新字段，或者重置某个已有字段 |
| <字段>: <0 或 false> | 舍弃某个字段（如果你指定舍弃了某个非 \_id 字段，那么在此次 project 中，你不能再使用其它表达式） |

**指定包含字段**

\_id 字段是默认包含在输出中的，除此之外其他任何字段，如果想要在输出中体现的话，必须在 project 中指定； 如果指定包含一个尚不存在的字段，那么 project 会忽略这个字段，不会加入到输出的文档中；

**指定排除字段**

如果你在 project 中指定排除某个字段，那么其它字段都会体现在输出中； 如果指定排除的是非 \_id 字段，那么在本次 project 中，不能再使用其它表达式；

**加入新的字段或重置某个已有字段**

你可以使用一些特殊的表达式加入新的字段，或重置某个已有字段。

**多层嵌套的字段**

有时有些字段处于多层嵌套的底层，我们可以使用点记法：

"contact.phone.number": <1 or 0 or 表达式>

也可以直接使用嵌套的格式：

contact: { phone: { number: <1 or 0 or 表达式> } }

**示例**

假设我们有一个 articles 集合，其中含有以下文档：

{

"\_id": 666,

"title": "This is title",

"author": "Nobody",

"isbn": "123456789",

"introduction": "......"

}

**指定包含某些字段**

下面的代码使用 project，让输出只包含 \_id、title 和 author 字段：

db.collection('articles')

.aggregate()

.project({

title: 1,

author: 1

})

.end()

输出如下：

{ "\_id" : 666, "title" : "This is title", "author" : "Nobody" }

**去除输出中的 \_id 字段**

\_id 是默认包含在输出中的，如果不想要它，可以指定去除它：

db.collection('articles')

.aggregate()

.project({

\_id: 0, // 指定去除 \_id 字段

title: 1,

author: 1

})

.end()

输出如下：

{ "title" : "This is title", "author" : "Nobody" }

**去除某个非 \_id 字段**

我们还可以指定在输出中去掉某个非 \_id 字段，这样其它字段都会被输出：

db.collection('articles')

.aggregate()

.project({

isbn: 0, // 指定去除 isbn 字段

})

.end()

输出如下，相比输入，没有了 isbn 字段：

{

"\_id" : 666,

"title" : "This is title",

"author" : "Nobody",

"introduction": "......"

}

**加入计算出的新字段**

假设我们有一个 students 集合，其中包含以下文档：

{

"\_id": 1,

"name": "小明",

"scores": {

"chinese": 80,

"math": 90,

"english": 70

}

}

下面的代码，我们使用 project，在输出中加入了一个新的字段 totalScore：

const { sum } = db.command.aggregate

db.collection('students')

.aggregate()

.project({

\_id: 0,

name: 1,

totalScore: sum([

"$scores.chinese",

"$scores.math",

"$scores.english"

])

})

.end()

输出为：

{ "name": "小明", "totalScore": 240 }

**加入新的数组字段**

假设我们有一个 points 集合，包含以下文档：

{ "\_id": 1, "x": 1, "y": 1 }

{ "\_id": 2, "x": 2, "y": 2 }

{ "\_id": 3, "x": 3, "y": 3 }

下面的代码，我们使用 project，把 x 和 y 字段，放入到一个新的数组字段 coordinate 中：

db.collection('points')

.aggregate()

.project({

coordinate: ["$x", "$y"]

})

.end()

输出如下：

{ "\_id": 1, "coordinate": [1, 1] }

{ "\_id": 2, "coordinate": [2, 2] }

{ "\_id": 3, "coordinate": [3, 3] }

### [replaceRoot](https://uniapp.dcloud.io/uniCloud/cf-database?id=replaceroot)

聚合阶段。指定一个已有字段作为输出的根节点，也可以指定一个计算出的新字段作为根节点。

**API 说明**

**replaceRoot 使用形式如下：**

replaceRoot({

newRoot: <表达式>

})

表达式格式如下：

| **格式** | **说明** |
| --- | --- |
| <字段名> | 指定一个已有字段作为输出的根节点（如果字段不存在则报错） |
| <对象> | 计算一个新字段，并且把这个新字段作为根节点 |

**示例**

**使用已有字段作为根节点**

假设我们有一个 schools 集合，内容如下：

{

"\_id": 1,

"name": "SFLS",

"teachers": {

"chinese": 22,

"math": 18,

"english": 21,

"other": 123

}

}

下面的代码使用 replaceRoot，把 teachers 字段作为根节点输出：

db.collection('schools')

.aggregate()

.replaceRoot({

newRoot: '$teachers'

})

.end()

输出如下：

{

"chinese": 22,

"math": 18,

"english": 21,

"other": 123

}

**使用计算出的新字段作为根节点**

假设我们有一个 roles 集合，内容如下：

{ "\_id": 1, "first\_name": "四郎", "last\_name": "黄" }

{ "\_id": 2, "first\_name": "邦德", "last\_name": "马" }

{ "\_id": 3, "first\_name": "牧之", "last\_name": "张" }

下面的代码使用 replaceRoot，把 first\_name 和 last\_name 拼在一起：

const { concat } = db.command.aggregate

db.collection('roles')

.aggregate()

.replaceRoot({

newRoot: {

full\_name: concat(['$last\_name', '$first\_name'])

}

})

.end()

输出如下：

{ "full\_name": "黄四郎" }

{ "full\_name": "马邦德" }

{ "full\_name": "张牧之" }

### [sample](https://uniapp.dcloud.io/uniCloud/cf-database?id=sample)

聚合阶段。随机从文档中选取指定数量的记录。

**API 说明**

**sample 的形式如下：**

sample({

size: <正整数>

})

请注意：size 是正整数，否则会出错。

**示例**

假设文档 users 有以下记录：

{ "name": "a" }

{ "name": "b" }

**随机选取**

如果现在进行抽奖活动，需要选出一名幸运用户。那么 sample 的调用方式如下：

db.collection('users')

.aggregate()

.sample({

size: 1

})

.end()

返回了随机选中的一个用户对应的记录，结果如下：

{ "\_id": "696529e4-7e82-4e7f-812e-5144714edff6", "name": "b" }

### [skip](https://uniapp.dcloud.io/uniCloud/cf-database?id=skip)

聚合阶段。指定一个正整数，跳过对应数量的文档，输出剩下的文档。

**示例**

db.collection('users')

.aggregate()

.skip(5)

.end()

这段代码会跳过查找到的前 5 个文档，并且把剩余的文档输出。

### [sort](https://uniapp.dcloud.io/uniCloud/cf-database?id=sort)

聚合阶段。根据指定的字段，对输入的文档进行排序。

**API 说明**

**形式如下：**

sort({

<字段名1>: <排序规则>,

<字段名2>: <排序规则>,

})

<排序规则>可以是以下取值：

* 1 代表升序排列（从小到大）；
* -1 代表降序排列（从大到小）；

**示例**

升序/降序排列

假设我们有集合 articles，其中包含数据如下：

{ "\_id": "1", "author": "stark", "score": 80, "age": 18 }

{ "\_id": "2", "author": "bob", "score": 60, "age": 18 }

{ "\_id": "3", "author": "li", "score": 55, "age": 19 }

{ "\_id": "4", "author": "jimmy", "score": 60, "age": 22 }

{ "\_id": "5", "author": "justan", "score": 95, "age": 33 }

db.collection('articles')

.aggregate()

.sort({

age: -1,

score: -1

})

.end()

上面的代码在 students 集合中进行聚合搜索，并且将结果排序，首先根据 age 字段降序排列，然后再根据 score 字段进行降序排列。

输出结果如下：

{ "\_id": "5", "author": "justan", "score": 95, "age": 33 }

{ "\_id": "4", "author": "jimmy", "score": 60, "age": 22 }

{ "\_id": "3", "author": "li", "score": 55, "age": 19 }

{ "\_id": "1", "author": "stark", "score": 80, "age": 18 }

{ "\_id": "2", "author": "bob", "score": 60, "age": 18 }

### [sortByCount](https://uniapp.dcloud.io/uniCloud/cf-database?id=sortbycount)

聚合阶段。根据传入的表达式，将传入的集合进行分组（group）。然后计算不同组的数量，并且将这些组按照它们的数量进行排序，返回排序后的结果。

**API 说明**

**sortByCount 的调用方式如下：**

sortByCount(<表达式>)

表达式的形式是：$ + 指定字段。请注意：不要漏写 $ 符号。

**示例**

**统计基础类型**

假设集合 passages 的记录如下：

{ "category": "Web" }

{ "category": "Web" }

{ "category": "Life" }

下面的代码就可以统计文章的分类信息，并且计算每个分类的数量。即对 category 字段执行 sortByCount 聚合操作。

db.collection('passages')

.aggregate()

.sortByCount('$category')

.end()

返回的结果如下所示：Web 分类下有2篇文章，Life 分类下有1篇文章。

{ "\_id": "Web", "count": 2 }

{ "\_id": "Life", "count": 1 }

**解构数组类型**

假设集合 passages 的记录如下：tags 字段对应的值是数组类型。

{ "tags": [ "JavaScript", "C#" ] }

{ "tags": [ "Go", "C#" ] }

{ "tags": [ "Go", "Python", "JavaScript" ] }

如何统计文章的标签信息，并且计算每个标签的数量？因为 tags 字段对应的数组，所以需要借助 unwind 操作解构 tags 字段，然后再调用 sortByCount。

下面的代码实现了这个功能：

db.collection('passages')

.aggregate()

.unwind(`$tags`)

.sortByCount(`$tags`)

.end()

返回的结果如下所示：

{ "\_id": "Go", "count": 2 }

{ "\_id": "C#", "count": 2 }

{ "\_id": "JavaScript", "count": 2 }

{ "\_id": "Python", "count": 1 }

### [unwind](https://uniapp.dcloud.io/uniCloud/cf-database?id=unwind)

聚合阶段。使用指定的数组字段中的每个元素，对文档进行拆分。拆分后，文档会从一个变为一个或多个，分别对应数组的每个元素。

**API 说明**

使用指定的数组字段中的每个元素，对文档进行拆分。拆分后，文档会从一个变为一个或多个，分别对应数组的每个元素。

**unwind 有两种使用形式：**

**参数是一个字段名**

unwind(<字段名>)

**参数是一个对象**

unwind({

path: <字段名>,

includeArrayIndex: <string>,

preserveNullAndEmptyArrays: <boolean>

})

| **字段** | **类型** | **说明** |
| --- | --- | --- |
| path | string | 想要拆分的数组的字段名，需要以 $ 开头。 |
| includeArrayIndex | string | 可选项，传入一个新的字段名，数组索引会保存在这个新的字段上。新的字段名不能以 $ 开头。 |
| preserveNullAndEmptyArrays | boolean | 如果为 true，那么在 path 对应的字段为 null、空数组或者这个字段不存在时，依然会输出这个文档；如果为 false，unwind 将不会输出这些文档。默认为 false。 |

**示例**

**拆分数组**

假设我们有一个 products 集合，包含数据如下：

{ "\_id": "1", "product": "tshirt", "size": ["S", "M", "L"] }

{ "\_id": "2", "product": "pants", "size": [] }

{ "\_id": "3", "product": "socks", "size": null }

{ "\_id": "4", "product": "trousers", "size": ["S"] }

{ "\_id": "5", "product": "sweater", "size": ["M", "L"] }

我们根据 size 字段对这些文档进行拆分

db.collection('products')

.aggregate()

.unwind('$size')

.end()

输出如下：

{ "\_id": "1", "product": "tshirt", "size": "S" }

{ "\_id": "1", "product": "tshirt", "size": "M" }

{ "\_id": "1", "product": "tshirt", "size": "L" }

{ "\_id": "4", "product": "trousers", "size": "S" }

{ "\_id": "5", "product": "sweater", "size": "M" }

{ "\_id": "5", "product": "sweater", "size": "L" }

**拆分后，保留原数组的索引**

我们根据 size 字段对文档进行拆分后，想要保留原数组索引在新的 index 字段中。

db.collection('products')

.aggregate()

.unwind({

path: '$size',

includeArrayIndex: 'index'

})

.end()

输出如下：

{ "\_id": "1", "product": "tshirt", "size": "S", "index": 0 }

{ "\_id": "1", "product": "tshirt", "size": "M", "index": 1 }

{ "\_id": "1", "product": "tshirt", "size": "L", "index": 2 }

{ "\_id": "4", "product": "trousers", "size": "S", "index": 0 }

{ "\_id": "5", "product": "sweater", "size": "M", "index": 0 }

{ "\_id": "5", "product": "sweater", "size": "L", "index": 1 }

**保留字段为空的文档**

注意到我们的集合中有两行特殊的空值数据：

...

{ "\_id": "2", "product": "pants", "size": [] }

{ "\_id": "3", "product": "socks", "size": null }

...

如果想要在输出中保留 size 为空数组、null，或者 size 字段不存在的文档，可以使用 preserveNullAndEmptyArrays 参数

db.collection('products')

.aggregate()

.unwind({

path: '$size',

preserveNullAndEmptyArrays: true

})

.end()

输出如下：

{ "\_id": "1", "product": "tshirt", "size": "S" }

{ "\_id": "1", "product": "tshirt", "size": "M" }

{ "\_id": "1", "product": "tshirt", "size": "L" }

{ "\_id": "2", "product": "pants", "size": null }

{ "\_id": "3", "product": "socks", "size": null }

{ "\_id": "4", "product": "trousers", "size": "S" }

{ "\_id": "5", "product": "sweater", "size": "M" }

{ "\_id": "5", "product": "sweater", "size": "L" }

### [end](https://uniapp.dcloud.io/uniCloud/cf-database?id=end)

标志聚合操作定义完成，发起实际聚合操作

**返回值**

Promise.<Object>

| **属性** | **类型** | **说明** |
| --- | --- | --- |
| list | Array.<any> | 聚合结果列表 |

**示例代码**

const $ = db.command.aggregate

db.collection('books').aggregate()

.group({

// 按 category 字段分组

\_id: '$category',

// 让输出的每组记录有一个 avgSales 字段，其值是组内所有记录的 sales 字段的平均值

avgSales: $.avg('$sales')

})

.end()

.then(res => console.log(res))

.catch(err => console.error(err))

## [聚合操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e8%81%9a%e5%90%88%e6%93%8d%e4%bd%9c%e7%ac%a6)

### [算术操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%ae%97%e6%9c%af%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [abs](https://uniapp.dcloud.io/uniCloud/cf-database?id=abs)

聚合操作符。返回一个数字的绝对值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e)

语法如下：

db.command.aggregate.abs(<number>)

abs 传入的值除了数字常量外，也可以是任何最终解析成一个数字的表达式。

如果表达式解析为 null 或者指向一个不存在的字段，则 abs 的结果是 null。如果值解析为 NaN，则结果是 NaN。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81)

假设集合 ratings 有如下记录：

{ \_id: 1, start: 5, end: 8 }

{ \_id: 2, start: 4, end: 4 }

{ \_id: 3, start: 9, end: 7 }

{ \_id: 4, start: 6, end: 7 }

··· 可以用如下方式求得各个记录的 start 和 end 之间的绝对差异大小：

const $ = db.command.aggregate

db.collection('ratings').aggregate()

.project({

delta: $.abs($.subtract(['$start', '$end']))

})

.end()

返回结果如下：

{ "\_id" : 1, "delta" : 3 }

{ "\_id" : 2, "delta" : 0 }

{ "\_id" : 3, "delta" : 2 }

{ "\_id" : 4, "delta" : 1 }

#### [add](https://uniapp.dcloud.io/uniCloud/cf-database?id=add)

聚合操作符。将数字相加或将数字加在日期上。如果数组中的其中一个值是日期，那么其他值将被视为毫秒数加在该日期上。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-1)

语法如下：

db.command.aggregate.add([<表达式1>, <表达式2>, ...])

表达式可以是形如 $ + 指定字段，也可以是普通字符串。只要能够被解析成字符串即可。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-1)

假设集合 staff 有如下记录：

{ \_id: 1, department: "x", sales: 5, engineer: 10, lastUpdate: ISODate("2019-05-01T00:00:00Z") }

{ \_id: 2, department: "y", sales: 10, engineer: 20, lastUpdate: ISODate("2019-05-01T02:00:00Z") }

{ \_id: 3, department: "z", sales: 20, engineer: 5, lastUpdate: ISODate("2019-05-02T03:00:00Z") }

**数字求和**

可以用如下方式求得各个记录人数总数：

const $ = db.command.aggregate

db.collection('staff').aggregate()

.project({

department: 1,

total: $.add(['$sales', '$engineer'])

})

.end()

返回结果如下：

{ \_id: 1, department: "x", total: 15 }

{ \_id: 2, department: "y", total: 30 }

{ \_id: 3, department: "z", total: 25 }

**增加日期值**

如下操作可以获取各个记录的 lastUpdate 加一个小时之后的值：

const $ = db.command.aggregate

db.collection('staff').aggregate()

.project({

department: 1,

lastUpdate: $.add(['$lastUpdate', 60\*60\*1000])

})

.end()

返回结果如下：

{ \_id: 1, department: "x", lastUpdate: ISODate("2019-05-01T01:00:00Z") }

{ \_id: 2, department: "y", lastUpdate: ISODate("2019-05-01T03:00:00Z") }

{ \_id: 3, department: "z", lastUpdate: ISODate("2019-05-02T04:00:00Z") }

#### [ceil](https://uniapp.dcloud.io/uniCloud/cf-database?id=ceil)

聚合操作符。向上取整，返回大于或等于给定数字的最小整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-2)

语法如下：

db.command.aggregate.ceil(<number>)

<number> 可以是任意解析为数字的表达式。如果表达式解析为 null 或指向一个不存在的字段，则返回 null，如果解析为 NaN，则返回 NaN。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-2)

假设集合 sales 有如下记录：

{ \_id: 1, sales: 5.2 }

{ \_id: 2, sales: 1.32 }

{ \_id: 3, sales: -3.2 }

可以用如下方式取各个数字的向上取整值：

const $ = db.command.aggregate

db.collection('sales').aggregate()

.project({

sales: $.ceil('$sales')

})

.end()

返回结果如下：

{ \_id: 1, sales: 6 }

{ \_id: 2, sales: 2 }

{ \_id: 3, sales: -3 }

#### [divide](https://uniapp.dcloud.io/uniCloud/cf-database?id=divide)

聚合操作符。传入被除数和除数，求商。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-3)

语法如下：

db.command.aggregate.divide([<被除数表达式>, <除数表达式>])

表达式可以是任意解析为数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-3)

假设集合 railroads 有如下记录：

{ \_id: 1, meters: 5300 }

{ \_id: 2, meters: 64000 }

{ \_id: 3, meters: 130 }

可以用如下方式取各个数字转换为千米之后的值：

const $ = db.command.aggregate

db.collection('railroads').aggregate()

.project({

km: $.divide(['$meters', 1000])

})

.end()

返回结果如下：

{ \_id: 1, km: 5.3 }

{ \_id: 2, km: 64 }

{ \_id: 3, km: 0.13 }

#### [exp](https://uniapp.dcloud.io/uniCloud/cf-database?id=exp)

聚合操作符。取 e（自然对数的底数，欧拉数） 的 n 次方。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-4)

语法如下：

db.command.aggregate.exp(<exponent>)

<exponent> 可以是任意解析为数字的表达式。如果表达式解析为 null 或指向一个不存在的字段，则返回 null，如果解析为 NaN，则返回 NaN。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-4)

假设集合 math 有如下记录：

{ \_id: 1, exp: 0 }

{ \_id: 2, exp: 1 }

{ \_id: 3, exp: 2 }

const $ = db.command.aggregate

db.collection('math').aggregate()

.project({

result: $.exp('$exp')

})

.end()

返回结果如下：

{ \_id: 1, result: 1 }

{ \_id: 2, result: 2.71828182845905 }

{ \_id: 3, result: 7.38905609893065 }

#### [floor](https://uniapp.dcloud.io/uniCloud/cf-database?id=floor)

聚合操作符。向下取整，返回大于或等于给定数字的最小整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-5)

语法如下：

db.command.aggregate.floor(<number>)

<number> 可以是任意解析为数字的表达式。如果表达式解析为 null 或指向一个不存在的字段，则返回 null，如果解析为 NaN，则返回 NaN。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-5)

假设集合 sales 有如下记录：

{ \_id: 1, sales: 5.2 }

{ \_id: 2, sales: 1.32 }

{ \_id: 3, sales: -3.2 }

可以用如下方式取各个数字的向下取整值：

const $ = db.command.aggregate

db.collection('sales').aggregate()

.project({

sales: $.floor('$sales')

})

.end()

返回结果如下：

{ \_id: 1, sales: 5 }

{ \_id: 2, sales: 1 }

{ \_id: 3, sales: -6 }

#### [ln](https://uniapp.dcloud.io/uniCloud/cf-database?id=ln)

聚合操作符。计算给定数字在自然对数值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-6)

语法如下：

db.command.aggregate.ln(<number>)

<number> 可以是任意解析为非负数字的表达式。

ln 等价于 log([<number>, Math.E])，其中 Math.E 是 JavaScript 获取 e 的值的方法。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-6)

##### [db.command.aggregate.ln](https://uniapp.dcloud.io/uniCloud/cf-database?id=dbcommandaggregateln)

聚合操作符。计算给定数字在自然对数值。

语法如下：

db.command.aggregate.ln(<number>)

<number> 可以是任意解析为非负数字的表达式。

ln 等价于 log([<number>, Math.E])，其中 Math.E 是 JavaScript 获取 e 的值的方法。

#### [log](https://uniapp.dcloud.io/uniCloud/cf-database?id=log)

聚合操作符。计算给定数字在给定对数底下的 log 值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-7)

语法如下：

db.command.aggregate.log([<number>, <base>])

<number> 可以是任意解析为非负数字的表达式。<base> 可以是任意解析为大于 1 的数字的表达式。

如果任一参数解析为 null 或指向任意一个不存在的字段，log 返回 null。如果任一参数解析为 NaN，log 返回 NaN。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-7)

假设集合 curve 有如下记录：

{ \_id: 1, x: 1 }

{ \_id: 2, x: 2 }

{ \_id: 3, x: 3 }

计算 log2(x) 的值：

const $ = db.command.aggregate

db.collection('staff').aggregate()

.project({

log: $.log(['$x', 2])

})

.end()

返回结果如下：

{ \_id: 1, log: 0 }

{ \_id: 2, log: 1 }

{ \_id: 3, log: 1.58496250072 }

#### [log10](https://uniapp.dcloud.io/uniCloud/cf-database?id=log10)

聚合操作符。计算给定数字在对数底为 10 下的 log 值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-8)

语法如下：

db.command.aggregate.log(<number>)

<number> 可以是任意解析为非负数字的表达式。

log10 等同于 log 方法的第二个参数固定为 10。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-8)

##### [db.command.aggregate.log10](https://uniapp.dcloud.io/uniCloud/cf-database?id=dbcommandaggregatelog10)

聚合操作符。计算给定数字在对数底为 10 下的 log 值。

语法如下：

db.command.aggregate.log(<number>)

<number> 可以是任意解析为非负数字的表达式。

log10 等同于 log 方法的第二个参数固定为 10。

#### [mod](https://uniapp.dcloud.io/uniCloud/cf-database?id=mod)

聚合操作符。取模运算，取数字取模后的值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-9)

语法如下：

db.command.aggregate.mod([<dividend>, <divisor>])

第一个数字是被除数，第二个数字是除数。参数可以是任意解析为数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-9)

假设集合 shopping 有如下记录：

{ \_id: 1, bags: 3, items: 5 }

{ \_id: 2, bags: 2, items: 8 }

{ \_id: 3, bags: 5, items: 16 }

各记录取 items 除以 bags 的余数（items % bags）：

const $ = db.command.aggregate

db.collection('shopping').aggregate()

.project({

overflow: $.mod(['$items', '$bags'])

})

.end()

返回结果如下：

{ \_id: 1, log: 2 }

{ \_id: 2, log: 0 }

{ \_id: 3, log: 1 }

#### [multiply](https://uniapp.dcloud.io/uniCloud/cf-database?id=multiply)

聚合操作符。取传入的数字参数相乘的结果。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-10)

语法如下：

db.command.aggregate.multiply([<expression1>, <expression2>, ...])

参数可以是任意解析为数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-10)

假设集合 fruits 有如下记录：

{ "\_id": 1, "name": "apple", "price": 10, "quantity": 100 }

{ "\_id": 2, "name": "orange", "price": 15, "quantity": 50 }

{ "\_id": 3, "name": "lemon", "price": 5, "quantity": 20 }

求各个水果的的总价值：

const $ = db.command.aggregate

db.collection('fruits').aggregate()

.project({

name: 1,

total: $.multiply(['$price', '$quantity']),

})

.end()

返回结果如下：

{ "\_id": 1, "name": "apple", "total": 1000 }

{ "\_id": 2, "name": "orange", "total": 750 }

{ "\_id": 3, "name": "lemo", "total": 100 }

#### [pow](https://uniapp.dcloud.io/uniCloud/cf-database?id=pow)

聚合操作符。求给定基数的指数次幂。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-11)

语法如下：

db.command.aggregate.pow([<base>, <exponent>])

参数可以是任意解析为数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-11)

假设集合 stats 有如下记录：

{ "\_id": 1, "x": 2, "y": 3 }

{ "\_id": 2, "x": 5, "y": 7 }

{ "\_id": 3, "x": 10, "y": 20 }

求 x 和 y 的平方和：

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

sumOfSquares: $.add([$.pow(['$x', 2]), $.pow(['$y', 2])]),

})

.end()

返回结果如下：

{ "\_id": 1, "sumOfSquares": 13 }

{ "\_id": 2, "sumOfSquares": 74 }

{ "\_id": 3, "sumOfSquares": 500 }

#### [sqrt](https://uniapp.dcloud.io/uniCloud/cf-database?id=sqrt)

聚合操作符。求平方根。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-12)

语法如下：

db.command.aggregate.sqrt([<number>])

参数可以是任意解析为非负数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-12)

假设直角三角形集合 triangle 有如下记录：

{ "\_id": 1, "x": 2, "y": 3 }

{ "\_id": 2, "x": 5, "y": 7 }

{ "\_id": 3, "x": 10, "y": 20 }

假设 x 和 y 分别为两直角边，则求斜边长：

const $ = db.command.aggregate

db.collection('triangle').aggregate()

.project({

len: $.sqrt([$.add([$.pow(['$x', 2]), $.pow(['$y', 2])])]),

})

.end()

返回结果如下：

{ "\_id": 1, "len": 3.605551275463989 }

{ "\_id": 2, "len": 8.602325267042627 }

{ "\_id": 3, "len": 22.360679774997898 }

#### [subtract](https://uniapp.dcloud.io/uniCloud/cf-database?id=subtract)

聚合操作符。将两个数字相减然后返回差值，或将两个日期相减然后返回相差的毫秒数，或将一个日期减去一个数字返回结果的日期。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-13)

语法如下：

db.command.aggregate.subtract([<expression1>, <expression2>])

参数可以是任意解析为数字或日期的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-13)

假设集合 scores 有如下记录：

{ "\_id": 1, "max": 10, "min": 1 }

{ "\_id": 2, "max": 7, "min": 5 }

{ "\_id": 3, "max": 6, "min": 6 }

求各个记录的 max 和 min 的差值。：

const $ = db.command.aggregate

db.collection('scores').aggregate()

.project({

diff: $.subtract(['$max', '$min'])

})

.end()

返回结果如下：

{ "\_id": 1, "diff": 9 }

{ "\_id": 2, "diff": 2 }

{ "\_id": 3, "diff": 0 }

#### [trunc](https://uniapp.dcloud.io/uniCloud/cf-database?id=trunc)

聚合操作符。将数字截断为整形。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-14)

语法如下：

db.command.aggregate.trunc(<number>)

参数可以是任意解析为数字的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-14)

假设集合 scores 有如下记录：

{ "\_id": 1, "value": 1.21 }

{ "\_id": 2, "value": 3.83 }

{ "\_id": 3, "value": -4.94 }

const $ = db.command.aggregate

db.collection('scores').aggregate()

.project({

int: $.trunc('$value')

})

.end()

返回结果如下：

{ "\_id": 1, "value": 1 }

{ "\_id": 2, "value": 3 }

{ "\_id": 3, "value": -4 }

### [数组操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%95%b0%e7%bb%84%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [arrayElemAt](https://uniapp.dcloud.io/uniCloud/cf-database?id=arrayelemat)

聚合操作符。返回在指定数组下标的元素。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-15)

语法如下：

db.command.aggregate.arrayElemAt([<array>, <index>])

<array> 可以是任意解析为数字的表达式。

<index> 可以是任意解析为整形的表达式。如果是正数，arrayElemAt 返回在 index 位置的元素，如果是负数，arrayElemAt 返回从数组尾部算起的 index 位置的元素。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-15)

假设集合 exams 有如下记录：

{ "\_id": 1, "scores": [80, 60, 65, 90] }

{ "\_id": 2, "scores": [78] }

{ "\_id": 3, "scores": [95, 88, 92] }

求各个第一次考试的分数和和最后一次的分数：

const $ = db.command.aggregate

db.collection('exams').aggregate()

.project({

first: $.arraElemAt(['$scores', 0]),

last: $.arraElemAt(['$scores', -1]),

})

.end()

返回结果如下：

{ "\_id": 1, "first": 80, "last": 90 }

{ "\_id": 2, "first": 78, "last": 78 }

{ "\_id": 3, "first": 95, "last": 92 }

#### [arrayToObject](https://uniapp.dcloud.io/uniCloud/cf-database?id=arraytoobject)

聚合操作符。将一个数组转换为对象。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-16)

语法可以取两种：

第一种：传入一个二维数组，第二维的数组长度必须为 2，其第一个值为字段名，第二个值为字段值

db.command.aggregate.arrayToObject([

[<key1>, <value1>],

[<key2>, <value2>],

...

])

第二种：传入一个对象数组，各个对象必须包含字段 k 和 v，分别指定字段名和字段值

db.command.aggregate.arrayToObject([

{ "k": <key1>, "v": <value1> },

{ "k": <key2>, "v": <value2> },

...

])

传入 arrayToObject 的参数只要可以解析为上述两种表示法之一即可。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-16)

假设集合 shops 有如下记录：

{ "\_id": 1, "sales": [ ["max", 100], ["min", 50] ] }

{ "\_id": 2, "sales": [ ["max", 70], ["min", 60] ] }

{ "\_id": 3, "sales": [ { "k": "max", "v": 50 }, { "k": "min", "v": 30 } ] }

求各个第一次考试的分数和和最后一次的分数：

const $ = db.command.aggregate

db.collection('shops').aggregate()

.project({

sales: $.arrayToObject('$sales'),

})

.end()

返回结果如下：

{ "\_id": 1, "sales": { "max": 100, "min": 50 } }

{ "\_id": 2, "sales": { "max": 70, "min": 60 } }

{ "\_id": 3, "sales": { "max": 50, "min": 30 } }

#### [concatArrays](https://uniapp.dcloud.io/uniCloud/cf-database?id=concatarrays)

聚合操作符。将多个数组拼接成一个数组。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-17)

语法如下：

db.command.aggregate.arrayToObject([ <array1>, <array2>, ... ])

参数可以是任意解析为数组的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-17)

假设集合 items 有如下记录：

{ "\_id": 1, "fruits": [ "apple" ], "vegetables": [ "carrot" ] }

{ "\_id": 2, "fruits": [ "orange", "lemon" ], "vegetables": [ "cabbage" ] }

{ "\_id": 3, "fruits": [ "strawberry" ], "vegetables": [ "spinach" ] }

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

list: $.concatArrays(['$fruits', '$vegetables']),

})

.end()

返回结果如下：

{ "\_id": 1, "list": [ "apple", "carrot" ] }

{ "\_id": 2, "list": [ "orange", "lemon", "cabbage" ] }

{ "\_id": 3, "list": [ "strawberry", "spinach" ] }

#### [filter](https://uniapp.dcloud.io/uniCloud/cf-database?id=filter)

聚合操作符。根据给定条件返回满足条件的数组的子集。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-18)

语法如下：

db.command.aggregate.filter({

input: <array>,

as: <string>,

cond: <expression>

})

| **字段** | **说明** |
| --- | --- |
| input | 一个可以解析为数组的表达式 |
| as | 可选，用于表示数组各个元素的变量，默认为 this |
| cond | 一个可以解析为布尔值的表达式，用于判断各个元素是否满足条件，各个元素的名字由 as 参数决定（参数名需加 $$ 前缀，如 $$this） |

参数可以是任意解析为数组的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-18)

假设集合 fruits 有如下记录：

{

"\_id": 1,

"stock": [

{ "name": "apple", "price": 10 },

{ "name": "orange", "price": 20 }

],

}

{

"\_id": 2,

"stock": [

{ "name": "lemon", "price": 15 },

],

}

const \_ = db.command

const $ = db.command.aggregate

db.collection('fruits').aggregate()

.project({

stock: $.filter({

input: '$stock',

as: 'item',

cond: $.gte(['$$item.price', 15])

})

})

.end()

返回结果如下：

{ "\_id": 1, "stock": [ { "name": "orange", "price": 20} ] }

{ "\_id": 2, "stock": [ { "name": "lemon", "price": 15 } ] }

#### [in](https://uniapp.dcloud.io/uniCloud/cf-database?id=in-1)

聚合操作符。给定一个值和一个数组，如果值在数组中则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-19)

语法如下：

db.command.aggregate.in([<value>, <array>])

<value> 可以是任意表达式。

<array> 可以是任意解析为数组的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-19)

假设集合 shops 有如下记录：

{ "\_id": 1, "topsellers": ["bread", "ice cream", "butter"] }

{ "\_id": 2, "topsellers": ["ice cream", "cheese", "yagurt"] }

{ "\_id": 3, "topsellers": ["croissant", "cucumber", "coconut"] }

标记销量最高的商品包含 ice cream 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

included: $.in(['ice cream', '$topsellers'])

})

.end()

返回结果如下：

{ "\_id": 1, "included": true }

{ "\_id": 2, "included": true }

{ "\_id": 3, "included": false }

#### [indexOfArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=indexofarray)

聚合操作符。在数组中找出等于给定值的第一个元素的下标，如果找不到则返回 -1。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-20)

语法如下：

db.command.aggregate.indexOfArray([ <array expression>, <search expression>, <start>, <end> ])

| **字段** | **类型** | **说明** |
| --- | --- | --- |
| - | string | 一个可以解析为数组的表达式，如果解析为 null，则 indexOfArray 返回 null |
| - | string | 对数据各个元素应用的条件匹配表达式 |
| - | integer | 可选，用于指定搜索的开始下标，必须是非负整数 |
| - | integer | 可选，用于指定搜索的结束下标，必须是非负整数，指定了 时也应指定 ，否则 默认当做 |

参数可以是任意解析为数组的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-20)

假设集合 stats 有如下记录：

{

"\_id": 1,

"sales": [ 1, 6, 2, 2, 5 ]

}

{

"\_id": 2,

"sales": [ 4, 2, 1, 5, 2 ]

}

{

"\_id": 3,

"sales": [ 2, 5, 3, 3, 1 ]

}

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

index: $.indexOfArray(['$sales', 2, 2])

})

.end()

返回结果如下：

{ "\_id": 1, "index": 2 }

{ "\_id": 2, "index": 4 }

{ "\_id": 3, "index": -1 }

#### [isArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=isarray)

聚合操作符。判断给定表达式是否是数组，返回布尔值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-21)

语法如下：

db.command.aggregate.isArray(<expression>)

参数可以是任意表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-21)

假设集合 stats 有如下记录：

{

"\_id": 1,

"base": 10,

"sales": [ 1, 6, 2, 2, 5 ]

}

{

"\_id": 2,

"base": 1,

"sales": 100

}

计算总销量，如果 sales 是数字，则求 sales \* base，如果 sales 是数组，则求数组元素之和与 base 的乘积。

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

sum: $.cond({

if: $.isArray('$sales'),

then: $.multiply([$.sum(['$sales']), '$base']),

else: $.multiply(['$sales', '$base']),

})

})

.end()

返回结果如下：

{ "\_id": 1, "index": 160 }

{ "\_id": 2, "index": 100 }

#### [map](https://uniapp.dcloud.io/uniCloud/cf-database?id=map)

聚合操作符。类似 JavaScript Array 上的 map 方法，将给定数组的每个元素按给定转换方法转换后得出新的数组。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-22)

语法如下：

db.command.aggregate.map({

input: <expression>,

as: <string>,

in: <expression>

})

| **字段** | **说明** |
| --- | --- |
| input | 一个可以解析为数组的表达式 |
| as | 可选，用于表示数组各个元素的变量，默认为 this |
| in | 一个可以应用在给定数组的各个元素上的表达式，各个元素的名字由 as 参数决定（参数名需加 $$ 前缀，如 $$this） |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-22)

假设集合 stats 有如下记录：

{

"\_id": 1,

"sales": [ 1.32, 6.93, 2.48, 2.82, 5.74 ]

}

{

"\_id": 2,

"sales": [ 2.97, 7.13, 1.58, 6.37, 3.69 ]

}

将各个数字截断为整形，然后求和

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

truncated: $.map({

input: '$sales',

as: 'num',

in: $.trunc('$$num'),

})

})

.project({

total: $.sum('$truncated')

})

.end()

返回结果如下：

{ "\_id": 1, "index": 16 }

{ "\_id": 2, "index": 19 }

#### [objectToArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=objecttoarray)

聚合操作符。将一个对象转换为数组。方法把对象的每个键值对都变成输出数组的一个元素，元素形如 { k: <key>, v: <value> }。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-23)

语法如下：

db.command.aggregate.objectToArray(<object>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-23)

假设集合 items 有如下记录：

{ "\_id": 1, "attributes": { "color": "red", "price": 150 } }

{ "\_id": 2, "attributes": { "color": "blue", "price": 50 } }

{ "\_id": 3, "attributes": { "color": "yellow", "price": 10 } }

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

array: $.objectToArray('$attributes')

})

.end()

返回结果如下：

{ "\_id": 1, "array": [{ "k": "color", "v": "red" }, { "k": "price", "v": 150 }] }

{ "\_id": 2, "array": [{ "k": "color", "v": "blue" }, { "k": "price", "v": 50 }] }

{ "\_id": 3, "array": [{ "k": "color", "v": "yellow" }, { "k": "price", "v": 10 }] }

#### [range](https://uniapp.dcloud.io/uniCloud/cf-database?id=range)

聚合操作符。返回一组生成的序列数字。给定开始值、结束值、非零的步长，range 会返回从开始值开始逐步增长、步长为给定步长、但不包括结束值的序列。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-24)

语法如下：

db.command.aggregate.range([<start>, <end>, <non-zero step>])

| **字段** | **说明** |
| --- | --- |
| start | 开始值，一个可以解析为整形的表达式 |
| end | 结束值，一个可以解析为整形的表达式 |
| non-zero step | 可选，步长，一个可以解析为非零整形的表达式，默认为 1 |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-24)

假设集合 stats 有如下记录：

{ "\_id": 1, "max": 52 }

{ "\_id": 2, "max": 38 }

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

points: $.range([0, '$max', 10])

})

.end()

返回结果如下：

{ "\_id": 1, "points": [0, 10, 20, 30, 40, 50] }

{ "\_id": 2, "points": [0, 10, 20] }

#### [reduce](https://uniapp.dcloud.io/uniCloud/cf-database?id=reduce)

聚合操作符。类似 JavaScript 的 reduce 方法，应用一个表达式于数组各个元素然后归一成一个元素。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-25)

语法如下：

db.command.aggregate.reduce({

input: <array>

initialValue: <expression>,

in: <expression>

})

| **字段** | **说明** |
| --- | --- |
| input | 输入数组，可以是任意解析为数组的表达式 |
| initialValue | 初始值 |
| in | 用来作用于每个元素的表达式，在 in 中有两个可用变量，value 是表示累计值的变量，this 是表示当前数组元素的变量 |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-25)

**简易字符串拼接**

假设集合 player 有如下记录：

{ "\_id": 1, "fullname": [ "Stephen", "Curry" ] }

{ "\_id": 2, "fullname": [ "Klay", "Thompsom" ] }

获取各个球员的全名，并加 Player: 前缀：

const $ = db.command.aggregate

db.collection('player').aggregate()

.project({

info: $.reduce({

input: '$fullname',

initialValue: 'Player:',

in: $.concat(['$$value', ' ', '$$this']),

})

})

.end()

返回结果如下：

{ "\_id": 1, "info": "Player: Stephen Curry" }

{ "\_id": 2, "info": "Player: Klay Thompson" }

获取各个球员的全名，不加前缀：

const $ = db.command.aggregate

db.collection('player').aggregate()

.project({

name: $.reduce({

input: '$fullname',

initialValue: '',

in: $.concat([

'$$value',

$.cond({

if: $.eq(['$$value', '']),

then: '',

else: ' ',

}),

'$$this',

]),

})

})

.end()

返回结果如下：

{ "\_id": 1, "name": "Stephen Curry" }

{ "\_id": 2, "name": "Klay Thompson" }

#### [reverseArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=reversearray)

聚合操作符。返回给定数组的倒序形式。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-26)

语法如下：

db.command.aggregate.reverseArray(<array>)

参数可以是任意解析为数组表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-26)

假设集合 stats 有如下记录：

{

"\_id": 1,

"sales": [ 1, 2, 3, 4, 5 ]

}

取 sales 倒序：

const $ = db.command.aggregate

db.collection('stats').aggregate()

.project({

reversed: $.reverseArray('$sales'),

})

.end()

返回结果如下：

{ "\_id": 1, "reversed": [5, 4, 3, 2, 1] }

#### [size](https://uniapp.dcloud.io/uniCloud/cf-database?id=size)

聚合操作符。返回数组长度。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-27)

语法如下：

db.command.aggregate.size(<array>)

<array> 可以是任意解析为数组的表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-27)

假设集合 shops 有如下记录：

{ "\_id": 1, "staff": [ "John", "Middleton", "George" ] }

{ "\_id": 2, "staff": [ "Steph", "Jack" ] }

计算各个商店的雇员数量：

const $ = db.command.aggregate

db.collection('staff').aggregate()

.project({

totalStaff: $.size('$staff')

})

.end()

返回结果如下：

{ "\_id": 1, "totalStaff": 3 }

{ "\_id": 2, "totalStaff": 2 }

#### [slice](https://uniapp.dcloud.io/uniCloud/cf-database?id=slice)

聚合操作符。类似 JavaScritp 的 slice 方法。返回给定数组的指定子集。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-28)

语法有两种：

返回从开头或结尾开始的 n 个元素：

db.command.aggregate.slice([<array>, <n>])

返回从指定位置算作数组开头、再向后或向前的 n 个元素：

db.command.aggregate.slice([<array>, <position>, <n>])

<array> 可以是任意解析为数组的表达式。

<position> 可以是任意解析为整形的表达式。如果是正数，则将数组的第 <position> 个元素作为数组开始；如果 <position> 比数组长度更长，slice 返回空数组。如果是负数，则将数组倒数第 <position> 个元素作为数组开始；如果 <position> 的绝对值大于数组长度，则开始位置即为数组开始位置。

<n> 可以是任意解析为整形的表达式。如果 <position> 有提供，则 <n> 必须为正整数。如果是正数，slice 返回前 n 个元素。如果是负数，slice 返回后 n 个元素。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-28)

假设集合 people 有如下记录：

{ "\_id": 1, "hobbies": [ "basketball", "football", "tennis", "badminton" ] }

{ "\_id": 2, "hobbies": [ "golf", "handball" ] }

{ "\_id": 3, "hobbies": [ "table tennis", "swimming", "rowing" ] }

统一返回前两个爱好：

const $ = db.command.aggregate

db.collection('fruits').aggregate()

.project({

hobbies: $.slice(['$hobbies', 2]),

})

.end()

返回结果如下：

{ "\_id": 1, "hobbies": [ "basketball", "football" ] }

{ "\_id": 2, "hobbies": [ "golf", "handball" ] }

{ "\_id": 3, "hobbies": [ "table tennis", "swimming" ] }

#### [zip](https://uniapp.dcloud.io/uniCloud/cf-database?id=zip)

聚合操作符。把二维数组的第二维数组中的相同序号的元素分别拼装成一个新的数组进而组装成一个新的二维数组。如可将 [ [ 1, 2, 3 ], [ "a", "b", "c" ] ] 转换成 [ [ 1, "a" ], [ 2, "b" ], [ 3, "c" ] ]。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-29)

语法如下：

db.command.aggregate.zip({

inputs: [<array1>, <array2>, ...],

useLongestLength: <boolean>,

defaults: <array>

})

inputs 是一个二维数组（inputs 不可以是字段引用），其中每个元素的表达式（这个可以是字段引用）都可以解析为数组。如果其中任意一个表达式返回 null，<inputs> 也返回 null。如果其中任意一个表达式不是指向一个合法的字段 / 解析为数组 / 解析为 null，则返回错误。

useLongestLength 决定输出数组的长度是否采用输入数组中的最长数组的长度。默认为 false，即输入数组中的最短的数组的长度即是输出数组的各个元素的长度。

defaults 是一个数组，用于指定在输入数组长度不一的情况下时采用的数组各元素默认值。指定这个字段则必须指定 useLongestLength，否则返回错误。如果 useLongestLength 是 true 但是 defaults 是空或没有指定，则 zip 用 null 做数组元素的缺省默认值。指定各元素默认值时 defaults 数组的长度必须是输入数组最大的长度。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-29)

假设集合 stats 有如下记录：

{ "\_id": 1, "zip1": [1, 2], "zip2": [3, 4], "zip3": [5, 6] ] }

{ "\_id": 2, "zip1": [1, 2], "zip2": [3], "zip3": [4, 5, 6] ] }

{ "\_id": 3, "zip1": [1, 2], "zip2": [3] ] }

**只传 inputs**

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

zip: $.zip({

inputs: [

'$zip1', // 字段引用

'$zip2',

'$zip3',

],

})

})

.end()

返回结果如下：

{ "\_id": 1, "zip": [ [1, 3, 5], [2, 4, 6] ] }

{ "\_id": 2, "zip": [ [1, 3, 4] ] }

{ "\_id": 3, "zip": null }

**设置 useLongestLength**

如果设 useLongestLength 为 true：

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

zip: $.zip({

inputs: [

'$zip1', // 字段引用

'$zip2',

'$zip3',

],

useLongestLength: true,

})

})

.end()

返回结果如下：

{ "\_id": 1, "zip": [ [1, 3, 5], [2, 4, 6] ] }

{ "\_id": 2, "zip": [ [1, 3, 4], [2, null, 5], [null, null, 6] ] }

{ "\_id": 3, "zip": null }

**设置 defaults**

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

zip: $.zip({

inputs: [

'$zip1', // 字段引用

'$zip2',

'$zip3',

],

useLongestLength: true,

defaults: [-300, -200, -100],

})

})

.end()

返回结果如下：

{ "\_id": 1, "zip": [ [1, 3, 5], [2, 4, 6] ] }

{ "\_id": 2, "zip": [ [1, 3, 4], [2, -200, 5], [-300, -200, 6] ] }

{ "\_id": 3, "zip": null }

### [布尔操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%b8%83%e5%b0%94%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [and](https://uniapp.dcloud.io/uniCloud/cf-database?id=and-1)

聚合操作符。给定多个表达式，and 仅在所有表达式都返回 true 时返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-30)

语法如下：

db.command.aggregate.and([<expression1>, <expression2>, ...])

如果表达式返回 false、null、0、或 undefined，表达式会解析为 false，否则对其他返回值都认为是 true。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-30)

假设集合 price 有如下记录：

{ "\_id": 1, "min": 10, "max": 100 }

{ "\_id": 2, "min": 60, "max": 80 }

{ "\_id": 3, "min": 30, "max": 50 }

求 min 大于等于 30 且 max 小于等于 80 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

fullfilled: $.and([$.gte(['$min', 30]), $.lte(['$max', 80])])

})

.end()

返回结果如下：

{ "\_id": 1, "fullfilled": false }

{ "\_id": 2, "fullfilled": true }

{ "\_id": 3, "fullfilled": true }

#### [not](https://uniapp.dcloud.io/uniCloud/cf-database?id=not)

聚合操作符。给定一个表达式，如果表达式返回 true，则 not 返回 false，否则返回 true。注意表达式不能为逻辑表达式（and、or、nor、not）。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-31)

语法如下：

db.command.aggregate.not(<expression>)

如果表达式返回 false、null、0、或 undefined，表达式会解析为 false，否则对其他返回值都认为是 true。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-31)

假设集合 price 有如下记录：

{ "\_id": 1, "min": 10, "max": 100 }

{ "\_id": 2, "min": 60, "max": 80 }

{ "\_id": 3, "min": 30, "max": 50 }

求 min 不大于 40 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

fullfilled: $.not($.gt(['$min', 40]))

})

.end()

返回结果如下：

{ "\_id": 1, "fullfilled": true }

{ "\_id": 2, "fullfilled": false }

{ "\_id": 3, "fullfilled": true }

#### [or](https://uniapp.dcloud.io/uniCloud/cf-database?id=or-1)

聚合操作符。给定多个表达式，如果任意一个表达式返回 true，则 or 返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-32)

语法如下：

db.command.aggregate.or([<expression1>, <expression2>, ...])

如果表达式返回 false、null、0、或 undefined，表达式会解析为 false，否则对其他返回值都认为是 true。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-32)

假设集合 price 有如下记录：

{ "\_id": 1, "min": 10, "max": 100 }

{ "\_id": 2, "min": 60, "max": 80 }

{ "\_id": 3, "min": 30, "max": 50 }

求 min 小于 40 且 max 大于 60 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

fullfilled: $.or([$.lt(['$min', 30]), $.gt(['$max', 60])])

})

.end()

返回结果如下：

{ "\_id": 1, "fullfilled": true }

{ "\_id": 2, "fullfilled": false }

{ "\_id": 3, "fullfilled": true }

### [比较操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%af%94%e8%be%83%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [cmp](https://uniapp.dcloud.io/uniCloud/cf-database?id=cmp)

聚合操作符。给定两个值，返回其比较值：

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-33)

如果第一个值小于第二个值，返回 -1 如果第一个值大于第二个值，返回 1 如果两个值相等，返回 0

语法如下：

db.command.aggregate.cmp([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-33)

假设集合 price 有如下记录：

{ "\_id": 1, "shop1": 10, "shop2": 100 }

{ "\_id": 2, "shop1": 80, "shop2": 20 }

{ "\_id": 3, "shop1": 50, "shop2": 50 }

求 shop1 和 shop2 的各个物品的价格对比。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

compare: $.cmp(['$shop1', '$shop2']))

})

.end()

返回结果如下：

{ "\_id": 1, "compare": -1 }

{ "\_id": 2, "compare": 1 }

{ "\_id": 3, "compare": 0 }

#### [eq](https://uniapp.dcloud.io/uniCloud/cf-database?id=eq-1)

聚合操作符。匹配两个值，如果相等则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-34)

语法如下：

db.command.aggregate.eq([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-34)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

求 value 等于 50 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.eq(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": false }

{ "\_id": 2, "matched": false }

{ "\_id": 3, "matched": true }

#### [gt](https://uniapp.dcloud.io/uniCloud/cf-database?id=gt-1)

聚合操作符。匹配两个值，如果前者大于后者则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-35)

语法如下：

db.command.aggregate.gt([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-35)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

判断 value 是否大于 50。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.gt(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": false }

{ "\_id": 2, "matched": true }

{ "\_id": 3, "matched": false }

#### [gte](https://uniapp.dcloud.io/uniCloud/cf-database?id=gte-1)

聚合操作符。匹配两个值，如果前者大于或等于后者则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-36)

语法如下：

db.command.aggregate.gte([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-36)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

判断 value 是否大于或等于 50。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.gte(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": false }

{ "\_id": 2, "matched": true }

{ "\_id": 3, "matched": true }

#### [lt](https://uniapp.dcloud.io/uniCloud/cf-database?id=lt-1)

聚合操作符。匹配两个值，如果前者小于后者则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-37)

语法如下：

db.command.aggregate.lt([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-37)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

判断 value 是否小于 50。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.lt(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": true }

{ "\_id": 2, "matched": false }

{ "\_id": 3, "matched": false }

#### [lte](https://uniapp.dcloud.io/uniCloud/cf-database?id=lte-1)

聚合操作符。匹配两个值，如果前者小于或等于后者则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-38)

语法如下：

db.command.aggregate.lte([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-38)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

判断 value 是否小于 50。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.lte(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": true }

{ "\_id": 2, "matched": false }

{ "\_id": 3, "matched": true }

#### [neq](https://uniapp.dcloud.io/uniCloud/cf-database?id=neq-1)

聚合操作符。匹配两个值，如果不相等则返回 true，否则返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-39)

语法如下：

db.command.aggregate.neq([<value1>, <value2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-39)

假设集合 price 有如下记录：

{ "\_id": 1, "value": 10 }

{ "\_id": 2, "value": 80 }

{ "\_id": 3, "value": 50 }

求 value 不等于 50 的记录。

const $ = db.command.aggregate

db.collection('price').aggregate()

.project({

matched: $.neq(['$value', 50])

})

.end()

返回结果如下：

{ "\_id": 1, "matched": true }

{ "\_id": 2, "matched": true }

{ "\_id": 3, "matched": false }

### [条件操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%9d%a1%e4%bb%b6%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [cond](https://uniapp.dcloud.io/uniCloud/cf-database?id=cond)

聚合操作符。计算布尔表达式，返回指定的两个值其中之一。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-40)

cond 的使用形式如下：

cond({ if: <布尔表达式>, then: <真值>, else: <假值> })

或者：

cond([ <布尔表达式>, <真值>, <假值> ])

两种形式中，三个参数（if、then、else）都是必须的。

如果布尔表达式为真，那么 $cond 将会返回 <真值>，否则会返回 <假值>

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-40)

假设集合 items 的记录如下：

{ "\_id": "0", "name": "item-a", "amount": 100 }

{ "\_id": "1", "name": "item-b", "amount": 200 }

{ "\_id": "2", "name": "item-c", "amount": 300 }

我们可以使用 cond，根据 amount 字段，来生成新的字段 discount：

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

name: 1,

discount: $.cond({

if: $.gte(['$amount', 200]),

then: 0.7,

else: 0.9

})

})

.end()

输出如下：

{ "\_id": "0", "name": "item-a", "discount": 0.9 }

{ "\_id": "1", "name": "item-b", "discount": 0.7 }

{ "\_id": "2", "name": "item-c", "discount": 0.7 }

#### [ifNull](https://uniapp.dcloud.io/uniCloud/cf-database?id=ifnull)

聚合操作符。计算给定的表达式，如果表达式结果为 null、undefined 或者不存在，那么返回一个替代值；否则返回原值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-41)

ifNull 的使用形式如下：

ifNull([ <表达式>, <替代值> ])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-41)

假设集合 items 的记录如下：

{ "\_id": "0", "name": "A", "description": "这是商品A" }

{ "\_id": "1", "name": "B", "description": null }

{ "\_id": "2", "name": "C" }

我们可以使用 ifNull，对不存在 desc 字段的文档，或者 desc 字段为 null 的文档，补充一个替代值。

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

\_id: 0,

name: 1,

description: $.ifNull(['$description', '商品描述空缺'])

})

.end()

输出如下：

{ "name": "A", "description": "这是商品A" }

{ "name": "B", "description": "商品描述空缺" }

{ "name": "C", "description": "商品描述空缺" }

#### [switch](https://uniapp.dcloud.io/uniCloud/cf-database?id=switch)

聚合操作符。根据给定的 switch-case-default 计算返回值、

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-42)

switch 的使用形式如下：

switch({

branches: [

case: <表达式>, then: <表达式>,

case: <表达式>, then: <表达式>,

...

],

default: <表达式>

})

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-42)

假设集合 items 的记录如下：

{ "\_id": "0", "name": "item-a", "amount": 100 }

{ "\_id": "1", "name": "item-b", "amount": 200 }

{ "\_id": "2", "name": "item-c", "amount": 300 }

我们可以使用 switch，根据 amount 字段，来生成新的字段 discount：

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

name: 1,

discount: $.switch({

branches: [

{ case: $.gt(['$amount', 250]), then: 0.8 },

{ case: $.gt(['$amount', 150]), then: 0.9 }

],

default: 1

})

})

.end()

输出如下：

{ "\_id": "0", "name": "item-a", "discount": 1 }

{ "\_id": "1", "name": "item-b", "discount": 0.9 }

{ "\_id": "2", "name": "item-c", "discount": 0.8 }

### [日期操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e6%97%a5%e6%9c%9f%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [dateFromParts](https://uniapp.dcloud.io/uniCloud/cf-database?id=datefromparts)

聚合操作符。给定日期的相关信息，构建并返回一个日期对象。

**平台差异说明**

| **阿里云** | **腾讯云** |
| --- | --- |
| × | √ |

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-43)

语法如下：

db.command.aggregate.dateFromParts({

year: <year>,

month: <month>,

day: <day>,

hour: <hour>,

minute: <minute>,

second: <second>,

millisecond: <ms>,

timezone: <tzExpression>

})

你也可以使用 ISO 8601 的标准：

db.command.aggregate.dateFromParts({

isoWeekYear: <year>,

isoWeek: <week>,

isoDayOfWeek: <day>,

hour: <hour>,

minute: <minute>,

second: <second>,

millisecond: <ms>,

timezone: <tzExpression>

})

**说明**

* timezone字段请参考[Olson Timezone Identifier](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones)，形式类似：Asia/Shanghai

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-43)

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

date: $.dateFromParts({

year: 2017,

month: 2,

day: 8,

hour: 12,

timezone: 'America/New\_York'

}),

})

.end()

输出如下：

{

"date": ISODate("2017-02-08T17:00:00.000Z")

}

#### [dateFromString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datefromstring)

聚合操作符。将一个日期/时间字符串转换为日期对象

**平台差异说明**

| **阿里云** | **腾讯云** |
| --- | --- |
| × | √ |

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-44)

语法如下：

db.command.aggregate.dateFromString({

dateString: <dateStringExpression>,

timezone: <tzExpression>

})

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-44)

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

date: $.dateFromString({

dateString: "2019-05-14T09:38:51.686Z"

})

})

.end()

输出如下：

{

"date": ISODate("2019-05-14T09:38:51.686Z")

}

#### [dateToString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datetostring)

聚合操作符。根据指定的表达式将日期对象格式化为符合要求的字符串。

**平台差异说明**

| **阿里云** | **腾讯云** |
| --- | --- |
| × | √ |

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-45)

dateToString 的调用形式如下：

db.command.aggregate.dateToString({

date: <日期表达式>,

format: <格式化表达式>,

timezone: <时区表达式>,

onNull: <空值表达式>

})

下面是四种表达式的详细说明：

| **名称** | **描述** |
| --- | --- |
| 日期表达式 | 必选。指定字段值应该是能转化为字符串的日期。 |
| 格式化表达式 | 可选。它可以是任何包含“格式说明符”的有效字符串。 |
| 时区表达式 | 可选。指明运算结果的时区。它可以解析格式为 [UTC Offset](https://en.wikipedia.org/wiki/List_of_UTC_time_offsets) 或者 [Olson Timezone Identifier](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones) 的字符串。 |
| 空值表达式 | 可选。当 <日期表达式> 返回空或者不存在的时候，会返回此表达式指明的值。 |

下面是格式说明符的详细说明：

| **说明符** | **描述** | **合法值** |
| --- | --- | --- |
| %d | 月份的日期（2位数，0填充） | 01 - 31 |
| %G | ISO 8601 格式的年份 | 0000 - 9999 |
| %H | 小时（2位数，0填充，24小时制） | 00 - 23 |
| %j | 一年中的一天（3位数，0填充） | 001 - 366 |
| %L | 毫秒（3位数，0填充） | 000 - 999 |
| %m | 月份（2位数，0填充） | 01 - 12 |
| %M | 分钟（2位数，0填充） | 00 - 59 |
| %S | 秒（2位数，0填充） | 00 - 60 |
| %w | 星期几 | 1 - 7 |
| %u | ISO 8601 格式的星期几 | 1 - 7 |
| %U | 一年中的一周（2位数，0填充） | 00 - 53 |
| %V | ISO 8601 格式的一年中的一周 | 1 - 53 |
| %Y | 年份（4位数，0填充） | 0000 - 9999 |
| %z | 与 UTC 的时区偏移量 | +/-[hh][mm] |
| %Z | 以分钟为单位，与 UTC 的时区偏移量 | +/-mmm |
| %% | 百分号作为字符 | % |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-45)

假设集合 students 有如下记录：

{ "date": "1999-12-11T16:00:00.000Z", "firstName": "Yuanxin", "lastName": "Dong" }

{ "date": "1998-11-10T16:00:00.000Z", "firstName": "Weijia", "lastName": "Wang" }

{ "date": "1997-10-09T16:00:00.000Z", "firstName": "Chengxi", "lastName": "Li" }

**格式化日期**

下面是将 date 字段的值，格式化成形如 年份-月份-日期 的字符串：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

formatDate: $.dateToString({

date: '$date',

format: '%Y-%m-%d'

})

})

.end()

返回的结果如下：

{ "formatDate": "1999-12-11" }

{ "formatDate": "1998-11-10" }

{ "formatDate": "1997-10-09" }

**时区时间**

下面是将 date 字段值格式化为上海时区时间的例子：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

formatDate: $.dateToString({

date: '$date',

format: '%H:%M:%S',

timezone: 'Asia/Shanghai'

})

})

.end()

返回的结果如下：

{ "formatDate": "00:00:00" }

{ "formatDate": "00:00:00" }

{ "formatDate": "00:00:00" }

**缺失情况的默认值**

当指定的 <日期表达式> 返回空或者不存在的时候，可以设置缺失情况下的默认值：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

formatDate: $.dateToString({

date: '$empty',

onNull: 'null'

})

})

.end()

返回的结果如下：

{ "formatDate": "null" }

{ "formatDate": "null" }

{ "formatDate": "null" }

#### [dayOfMonth](https://uniapp.dcloud.io/uniCloud/cf-database?id=dayofmonth)

聚合操作符。返回日期字段对应的天数（一个月中的哪一天），是一个介于 1 至 31 之间的数字。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-46)

语法如下：

db.command.aggregate.dayOfMonth(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-46)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 dayOfMonth() 对 date 字段进行投影，获取对应的日期：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

dayOfMonth: $.dayOfMonth('$date')

})

.end()

输出如下：

{

"dayOfMonth": 14

}

#### [dayOfWeek](https://uniapp.dcloud.io/uniCloud/cf-database?id=dayofweek)

聚合操作符。返回日期字段对应的天数（一周中的第几天），是一个介于 1（周日）到 7（周六）之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-47)

注意：周日是每周的第 1 天\*

语法如下：

db.command.aggregate.dayOfWeek(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-47)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 dayOfWeek() 对 date 字段进行投影，获取对应的天数（一周中的第几天）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

dayOfWeek: $.dayOfWeek('$date')

})

.end()

输出如下：

{

"dayOfWeek": 3

}

#### [dayOfYear](https://uniapp.dcloud.io/uniCloud/cf-database?id=dayofyear)

聚合操作符。返回日期字段对应的天数（一年中的第几天），是一个介于 1 到 366 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-48)

语法如下：

db.command.aggregate.dayOfYear(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-48)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 dayOfYear() 对 date 字段进行投影，获取对应的天数（一年中的第几天）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

dayOfYear: $.dayOfYear('$date')

})

.end()

输出如下：

{

"dayOfYear": 134

}

#### [hour](https://uniapp.dcloud.io/uniCloud/cf-database?id=hour)

聚合操作符。返回日期字段对应的小时数，是一个介于 0 到 23 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-49)

语法如下：

db.command.aggregate.hour(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-49)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 hour() 对 date 字段进行投影，获取对应的小时数：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

hour: $.hour('$date')

})

.end()

输出如下：

{

"hour": 9

}

#### [isoDayOfWeek](https://uniapp.dcloud.io/uniCloud/cf-database?id=isodayofweek)

聚合操作符。返回日期字段对应的 ISO 8601 标准的天数（一周中的第几天），是一个介于 1（周一）到 7（周日）之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-50)

语法如下：

db.command.aggregate.month(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-50)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 month() 对 date 字段进行投影，获取对应的 ISO 8601 标准的天数（一周中的第几天）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

isoDayOfWeek: $.isoDayOfWeek('$date')

})

.end()

输出如下：

{

"isoDayOfWeek": 2

}

#### [isoWeek](https://uniapp.dcloud.io/uniCloud/cf-database?id=isoweek)

聚合操作符。返回日期字段对应的 ISO 8601 标准的周数（一年中的第几周），是一个介于 1 到 53 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-51)

根据 ISO 8601 标准，周一到周日视为一周，本年度第一个周四所在的那周，视为本年度的第 1 周。

例如：2016 年 1 月 7 日是那年的第一个周四，那么 2016.01.04（周一）到 2016.01.10（周日） 即为第 1 周。同理，2016 年 1 月 1 日的周数为 53。

语法如下：

db.command.aggregate.isoWeek(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-51)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 isoWeek() 对 date 字段进行投影，获取对应的 ISO 8601 标准的周数（一年中的第几周）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

isoWeek: $.isoWeek('$date')

})

.end()

输出如下：

{

"isoWeek": 20

}

#### [isoWeekYear](https://uniapp.dcloud.io/uniCloud/cf-database?id=isoweekyear)

聚合操作符。返回日期字段对应的 ISO 8601 标准的天数（一年中的第几天）。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-52)

此处的“年”以第一周的周一为开始，以最后一周的周日为结束。

语法如下：

db.command.aggregate.isoWeekYear(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-52)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 isoWeekYear() 对 date 字段进行投影，获取对应的 ISO 8601 标准的天数（一年中的第几天）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

isoWeekYear: $.isoWeekYear('$date')

})

.end()

输出如下：

{

"isoWeekYear": 2019

}

#### [millisecond](https://uniapp.dcloud.io/uniCloud/cf-database?id=millisecond)

聚合操作符。返回日期字段对应的毫秒数，是一个介于 0 到 999 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-53)

语法如下：

db.command.aggregate.millisecond(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-53)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 millisecond() 对 date 字段进行投影，获取对应的毫秒数：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

millisecond: $.millisecond('$date'),

})

.end()

输出如下：

{

"millisecond": 686

}

#### [minute](https://uniapp.dcloud.io/uniCloud/cf-database?id=minute)

聚合操作符。返回日期字段对应的分钟数，是一个介于 0 到 59 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-54)

语法如下：

db.command.aggregate.minute(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-54)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 minute() 对 date 字段进行投影，获取对应的分钟数：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

minute: $.minute('$date')

})

.end()

输出如下：

{

"minute": 38

}

#### [month](https://uniapp.dcloud.io/uniCloud/cf-database?id=month)

聚合操作符。返回日期字段对应的月份，是一个介于 1 到 12 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-55)

语法如下：

db.command.aggregate.month(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-55)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 month() 对 date 字段进行投影，获取对应的月份：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

month: $.month('$date')

})

.end()

输出如下：

{

"month": 5

}

#### [second](https://uniapp.dcloud.io/uniCloud/cf-database?id=second)

聚合操作符。返回日期字段对应的秒数，是一个介于 0 到 59 之间的整数，在特殊情况下（闰秒）可能等于 60。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-56)

语法如下：

db.command.aggregate.second(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-56)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 second() 对 date 字段进行投影，获取对应的秒数：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

second: $.second('$date')

})

.end()

输出如下：

{

"second": 51

}

#### [week](https://uniapp.dcloud.io/uniCloud/cf-database?id=week)

聚合操作符。返回日期字段对应的周数（一年中的第几周），是一个介于 0 到 53 之间的整数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-57)

每周以周日为开头，**每年的第一个周日**即为 week 1 的开始，这天之前是 week 0。

语法如下：

db.command.aggregate.week(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-57)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 week() 对 date 字段进行投影，获取对应的周数（一年中的第几周）：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

week: $.week('$date')

})

.end()

输出如下：

{

"week": 19

}

#### [year](https://uniapp.dcloud.io/uniCloud/cf-database?id=year)

聚合操作符。返回日期字段对应的年份。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-58)

语法如下：

db.command.aggregate.year(<日期字段>)

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-58)

假设集合 dates 有以下文档：

{

"\_id": 1,

"date": ISODate("2019-05-14T09:38:51.686Z")

}

我们使用 year() 对 date 字段进行投影，获取对应的年份：

const $ = db.command.aggregate

db

.collection('dates')

.aggregate()

.project({

\_id: 0,

year: $.year('$date')

})

.end()

输出如下：

{

"year": 2019

}

#### [subtract](https://uniapp.dcloud.io/uniCloud/cf-database?id=subtract-1)

见[subtract](https://uniapp.dcloud.io/uniCloud/cf-database?id=subtract)

### [常量操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%b8%b8%e9%87%8f%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [literal](https://uniapp.dcloud.io/uniCloud/cf-database?id=literal)

聚合操作符。直接返回一个值的字面量，不经过任何解析和处理。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-59)

literal 使用形式如下：

literal(<值>)

如果 <值> 是一个表达式，那么 literal **不会**解析或者计算这个表达式，而是直接返回这个表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-59)

比如我们有一个 items 集合，其中数据如下：

{ "\_id": "0", "price": "$1" }

{ "\_id": "1", "price": "$5.60" }

{ "\_id": "2", "price": "$8.90" }

**以字面量的形式使用 $**

下面的代码使用 literal，生成了一个新的字段 isOneDollar，表示 price 字段是否严格等于 "$1"。

注意：我们这里无法使用 eq(['$price', '$1'])，因为 "$1" 是一个表达式，代表 "1" 字段对应的值，而不是字符串字面量 "$1"。

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

isOneDollar: $.eq(['$price', $.literal('$1')])

})

.end()

输出如下：

{ "\_id": "0", "isOneDollar": true }

{ "\_id": "1", "isOneDollar": false }

{ "\_id": "2", "isOneDollar": false }

**投影一个字段，对应的值为 1**

下面的代码使用 literal，投影了一个新的字段 amount，其值为 1。

const $ = db.command.aggregate

db.collection('items').aggregate()

.project({

price: 1,

amount: $.literal(1)

})

.end()

输出如下：

{ "\_id": "0", "price": "$1", "amount": 1 }

{ "\_id": "1", "price": "$5.60", "amount": 1 }

{ "\_id": "2", "price": "$8.90", "amount": 1 }

### [对象操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%af%b9%e8%b1%a1%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [mergeObjects](https://uniapp.dcloud.io/uniCloud/cf-database?id=mergeobjects)

聚合操作符。将多个文档合并为单个文档。

**平台差异说明**

| **阿里云** | **腾讯云** |
| --- | --- |
| × | √ |

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-60)

使用形式如下： 在 group() 中使用时：

mergeObjects(<document>)

在其它表达式中使用时：

mergeObjects([<document1>, <document2>, ...])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-60)

**搭配 group() 使用**

假设集合 sales 存在以下文档：

{ "\_id": 1, "year": 2018, "name": "A", "volume": { "2018Q1": 500, "2018Q2": 500 } }

{ "\_id": 2, "year": 2017, "name": "A", "volume": { "2017Q1": 400, "2017Q2": 300, "2017Q3": 0, "2017Q4": 0 } }

{ "\_id": 3, "year": 2018, "name": "B", "volume": { "2018Q1": 100 } }

{ "\_id": 4, "year": 2017, "name": "B", "volume": { "2017Q3": 100, "2017Q4": 250 } }

下面的代码使用 mergeObjects()，将用相同 name 的文档合并：

const $ = db.command.aggregate

db.collection('sales').aggregate()

.group({

\_id: '$name',

mergedVolume: $.mergeObjects('$volume')

})

.end()

输出如下：

{ "\_id": "A", "mergedVolume": { "2017Q1": 400, "2017Q2": 300, "2017Q3": 0, "2017Q4": 0, "2018Q1": 500, "2018Q2": 500 } }

{ "\_id": "B", "mergedVolume": { "2017Q3": 100, "2017Q4": 250, "2018Q1": 100 } }

**一般用法**

假设集合 test 存在以下文档：

{ "\_id": 1, "foo": { "a": 1 }, "bar": { "b": 2 } }

{ "\_id": 2, "foo": { "c": 1 }, "bar": { "d": 2 } }

{ "\_id": 3, "foo": { "e": 1 }, "bar": { "f": 2 } }

下面的代码使用 mergeObjects()，将文档中的 foo 和 bar 字段合并为 foobar：

const $ = db.command.aggregate

db.collection('sales').aggregate()

.project({

foobar: $.mergeObjects(['$foo', '$bar'])

})

.end()

输出结果如下：

{ "\_id": 1, "foobar": { "a": 1, "b": 2 } }

{ "\_id": 2, "foobar": { "c": 1, "d": 2 } }

{ "\_id": 3, "foobar": { "e": 1, "f": 2 } }

#### [objectToArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=objecttoarray-1)

见[objectToArray](https://uniapp.dcloud.io/uniCloud/cf-database?id=objecttoarray)

### [集合操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e9%9b%86%e5%90%88%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [allElementsTrue](https://uniapp.dcloud.io/uniCloud/cf-database?id=allelementstrue)

聚合操作符。输入一个数组，或者数组字段的表达式。如果数组中所有元素均为真值，那么返回 true，否则返回 false。空数组永远返回 true。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-61)

语法如下：

allElementsTrue([<expression>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-61)

假设集合 test 有如下记录：

{ "\_id": 1, "array": [ true ] }

{ "\_id": 2, "array": [ ] }

{ "\_id": 3, "array": [ false ] }

{ "\_id": 4, "array": [ true, false ] }

{ "\_id": 5, "array": [ 0 ] }

{ "\_id": 6, "array": [ "stark" ] }

下面的代码使用 allElementsTrue()，判断 array 字段中是否均为真值：

const $ = db.command.aggregate

db.collection('price')

.aggregate()

.project({

isAllTrue: $.allElementsTrue(['$array'])

})

.end()

返回结果如下：

{ "\_id": 1, "isAllTrue": true }

{ "\_id": 2, "isAllTrue": true }

{ "\_id": 3, "isAllTrue": false }

{ "\_id": 4, "isAllTrue": false }

{ "\_id": 5, "isAllTrue": false }

{ "\_id": 6, "isAllTrue": true }

#### [anyElementTrue](https://uniapp.dcloud.io/uniCloud/cf-database?id=anyelementtrue)

聚合操作符。输入一个数组，或者数组字段的表达式。如果数组中任意一个元素为真值，那么返回 true，否则返回 false。空数组永远返回 false。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-62)

语法如下：

anyElementTrue([<expression>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-62)

假设集合 test 有如下记录：

{ "\_id": 1, "array": [ true ] }

{ "\_id": 2, "array": [ ] }

{ "\_id": 3, "array": [ false ] }

{ "\_id": 4, "array": [ true, false ] }

{ "\_id": 5, "array": [ 0 ] }

{ "\_id": 6, "array": [ "stark" ] }

下面的代码使用 anyElementTrue()，判断 array 字段中是否含有真值：

const $ = db.command.aggregate

db.collection('price')

.aggregate()

.project({

isAnyTrue: $.anyElementTrue(['$array'])

})

.end()

返回结果如下：

{ "\_id": 1, "isAnyTrue": true }

{ "\_id": 2, "isAnyTrue": false }

{ "\_id": 3, "isAnyTrue": false }

{ "\_id": 4, "isAnyTrue": true }

{ "\_id": 5, "isAnyTrue": false }

{ "\_id": 6, "isAnyTrue": true }

#### [setDifference](https://uniapp.dcloud.io/uniCloud/cf-database?id=setdifference)

聚合操作符，输入两个集合，输出只存在于第一个集合中的元素。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-63)

使用形式如下：

setDifference([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-63)

假设集合 test 存在以下数据：

{ "\_id": 1, "A": [ 1, 2 ], "B": [ 1, 2 ] }

{ "\_id": 2, "A": [ 1, 2 ], "B": [ 2, 1, 2 ] }

{ "\_id": 3, "A": [ 1, 2 ], "B": [ 1, 2, 3 ] }

{ "\_id": 4, "A": [ 1, 2 ], "B": [ 3, 1 ] }

{ "\_id": 5, "A": [ 1, 2 ], "B": [ ] }

{ "\_id": 6, "A": [ 1, 2 ], "B": [ {}, [] ] }

{ "\_id": 7, "A": [ ], "B": [ ] }

{ "\_id": 8, "A": [ ], "B": [ 1 ] }

下面的代码使用 setDifference，找到只存在于 B 中的数字：

db.collection('test')

.aggregate()

.project({

isBOnly: $.setDifference(['$B', '$A'])

})

.end()

{ "\_id": 1, "isBOnly": [] }

{ "\_id": 2, "isBOnly": [3] }

{ "\_id": 3, "isBOnly": [3] }

{ "\_id": 4, "isBOnly": [5] }

{ "\_id": 5, "isBOnly": [] }

{ "\_id": 6, "isBOnly": [{}, []] }

{ "\_id": 7, "isBOnly": [] }

{ "\_id": 8, "isBOnly": [1] }

#### [setEquals](https://uniapp.dcloud.io/uniCloud/cf-database?id=setequals)

聚合操作符，输入两个集合，判断两个集合中包含的元素是否相同（不考虑顺序、去重）。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-64)

使用形式如下：

setEquals([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-64)

假设集合 test 存在以下数据：

{ "\_id": 1, "A": [ 1, 2 ], "B": [ 1, 2 ] }

{ "\_id": 2, "A": [ 1, 2 ], "B": [ 2, 1, 2 ] }

{ "\_id": 3, "A": [ 1, 2 ], "B": [ 1, 2, 3 ] }

{ "\_id": 4, "A": [ 1, 2 ], "B": [ 3, 1 ] }

{ "\_id": 5, "A": [ 1, 2 ], "B": [ ] }

{ "\_id": 6, "A": [ 1, 2 ], "B": [ {}, [] ] }

{ "\_id": 7, "A": [ ], "B": [ ] }

{ "\_id": 8, "A": [ ], "B": [ 1 ] }

下面的代码使用 setEquals，判断两个集合中包含的元素是否相同：

db.collection('test')

.aggregate()

.project({

sameElements: $.setEquals(['$A', '$B'])

})

.end()

{ "\_id": 1, "sameElements": true }

{ "\_id": 2, "sameElements": true }

{ "\_id": 3, "sameElements": false }

{ "\_id": 4, "sameElements": false }

{ "\_id": 5, "sameElements": false }

{ "\_id": 6, "sameElements": false }

{ "\_id": 7, "sameElements": true }

{ "\_id": 8, "sameElements": false }

#### [setIntersection](https://uniapp.dcloud.io/uniCloud/cf-database?id=setintersection)

聚合操作符，输入两个集合，输出两个集合的交集。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-65)

使用形式如下：

setIntersection([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-65)

假设集合 test 存在以下数据：

{ "\_id": 1, "A": [ 1, 2 ], "B": [ 1, 2 ] }

{ "\_id": 2, "A": [ 1, 2 ], "B": [ 2, 1, 2 ] }

{ "\_id": 3, "A": [ 1, 2 ], "B": [ 1, 2, 3 ] }

{ "\_id": 4, "A": [ 1, 2 ], "B": [ 3, 1 ] }

{ "\_id": 5, "A": [ 1, 2 ], "B": [ ] }

{ "\_id": 6, "A": [ 1, 2 ], "B": [ {}, [] ] }

{ "\_id": 7, "A": [ ], "B": [ ] }

{ "\_id": 8, "A": [ ], "B": [ 1 ] }

下面的代码使用 setIntersection，输出两个集合的交集：

db.collection('test')

.aggregate()

.project({

commonToBoth: $.setIntersection(['$A', '$B'])

})

.end()

输出如下：

{ "\_id": 1, "commonToBoth": [ 1, 2 ] }

{ "\_id": 2, "commonToBoth": [ 1, 2 ] }

{ "\_id": 3, "commonToBoth": [ 1, 2 ] }

{ "\_id": 4, "commonToBoth": [ 1 ] }

{ "\_id": 5, "commonToBoth": [ ] }

{ "\_id": 6, "commonToBoth": [ ] }

{ "\_id": 7, "commonToBoth": [ ] }

{ "\_id": 8, "commonToBoth": [ ] }

#### [setIsSubset](https://uniapp.dcloud.io/uniCloud/cf-database?id=setissubset)

聚合操作符，输入两个集合，判断第一个集合是否是第二个集合的子集。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-66)

使用形式如下：

setIsSubset([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-66)

假设集合 test 存在以下数据：

{ "\_id": 1, "A": [ 1, 2 ], "B": [ 1, 2 ] }

{ "\_id": 2, "A": [ 1, 2 ], "B": [ 2, 1, 2 ] }

{ "\_id": 3, "A": [ 1, 2 ], "B": [ 1, 2, 3 ] }

{ "\_id": 4, "A": [ 1, 2 ], "B": [ 3, 1 ] }

{ "\_id": 5, "A": [ 1, 2 ], "B": [ ] }

{ "\_id": 6, "A": [ 1, 2 ], "B": [ {}, [] ] }

{ "\_id": 7, "A": [ ], "B": [ ] }

{ "\_id": 8, "A": [ ], "B": [ 1 ] }

下面的代码使用 setIsSubset，判断第一个集合是否是第二个集合的子集：

db.collection('test')

.aggregate()

.project({

AisSubsetOfB: $.setIsSubset(['$A', '$B'])

})

.end()

{ "\_id": 1, "AisSubsetOfB": true }

{ "\_id": 2, "AisSubsetOfB": true }

{ "\_id": 3, "AisSubsetOfB": true }

{ "\_id": 4, "AisSubsetOfB": false }

{ "\_id": 5, "AisSubsetOfB": false }

{ "\_id": 6, "AisSubsetOfB": false }

{ "\_id": 7, "AisSubsetOfB": true }

{ "\_id": 8, "AisSubsetOfB": true }

#### [setUnion](https://uniapp.dcloud.io/uniCloud/cf-database?id=setunion)

聚合操作符，输入两个集合，输出两个集合的并集。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-67)

使用形式如下：

setUnion([<expression1>, <expression2>])

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-67)

假设集合 test 存在以下数据：

{ "\_id": 1, "A": [ 1, 2 ], "B": [ 1, 2 ] }

{ "\_id": 2, "A": [ 1, 2 ], "B": [ 2, 1, 2 ] }

{ "\_id": 3, "A": [ 1, 2 ], "B": [ 1, 2, 3 ] }

{ "\_id": 4, "A": [ 1, 2 ], "B": [ 3, 1 ] }

{ "\_id": 5, "A": [ 1, 2 ], "B": [ ] }

{ "\_id": 6, "A": [ 1, 2 ], "B": [ {}, [] ] }

{ "\_id": 7, "A": [ ], "B": [ ] }

{ "\_id": 8, "A": [ ], "B": [ 1 ] }

下面的代码使用 setUnion，输出两个集合的并集：

db.collection('test')

.aggregate()

.project({

AB: $.setUnion(['$A', '$B'])

})

.end()

输出如下：

{ "\_id": 1, "AB": [ 1, 2 ] }

{ "\_id": 2, "AB": [ 1, 2 ] }

{ "\_id": 3, "AB": [ 1, 2, 3 ] }

{ "\_id": 4, "AB": [ 1, 2, 3 ] }

{ "\_id": 5, "AB": [ 1, 2 ] }

{ "\_id": 6, "AB": [ 1, 2, {}, [] ] }

{ "\_id": 7, "AB": [ ] }

{ "\_id": 8, "AB": [ 1 ] }

### [字符串操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%ad%97%e7%ac%a6%e4%b8%b2%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [concat](https://uniapp.dcloud.io/uniCloud/cf-database?id=concat)

聚合操作符。连接字符串，返回拼接后的字符串。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-68)

concat 的语法如下：

db.command.aggregate.concat([<表达式1>, <表达式2>, ...])

表达式可以是形如 $ + 指定字段，也可以是普通字符串。只要能够被解析成字符串即可。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-68)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 concat 可以拼接 lastName 和 firstName 字段，得到每位学生的名字全称：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

fullName: $.concat(['$firstName', ' ', '$lastName'])

})

.end()

返回的结果如下：

{ "fullName": "Yuanxin Dong" }

{ "fullName": "Weijia Wang" }

{ "fullName": "Chengxi Li" }

#### [dateFromString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datefromstring-1)

见[dateFromString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datefromstring)

#### [dateToString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datetostring-1)

见[dateToString](https://uniapp.dcloud.io/uniCloud/cf-database?id=datetostring)

#### [indexOfBytes](https://uniapp.dcloud.io/uniCloud/cf-database?id=indexofbytes)

聚合操作符。在目标字符串中查找子字符串，并返回第一次出现的 UTF-8 的字节索引（从0开始）。如果不存在子字符串，返回 -1。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-69)

indexOfBytes 的语法如下：

db.command.aggregate.indexOfBytes([<目标字符串表达式>, <子字符串表达式>, <开始位置表达式>, <结束位置表达式>])

下面是 4 种表达式的详细描述：

| **表达式** | **描述** |
| --- | --- |
| 目标字符串表达式 | 任何可以被解析为字符串的表达式 |
| 子字符串表达式 | 任何可以被解析为字符串的表达式 |
| 开始位置表达式 | 任何可以被解析为非负整数的表达式 |
| 结束位置表达式 | 任何可以被解析为非负整数的表达式 |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-69)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 indexOfBytes 查找字符 "a" 在字段 firstName 中的位置：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

aStrIndex: $.indexOfBytes(['$firstName', 'a'])

})

.end()

返回的结果如下：

{ "aStrIndex": 2 }

{ "aStrIndex": 5 }

{ "aStrIndex": -1 }

#### [indexOfCP](https://uniapp.dcloud.io/uniCloud/cf-database?id=indexofcp)

聚合操作符。在目标字符串中查找子字符串，并返回第一次出现的 UTF-8 的 code point 索引（从0开始）。如果不存在子字符串，返回 -1。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-70)

code point 是“码位”，又名“编码位置”。这里特指 Unicode 包中的码位，范围是从0（16进制）到10FFFF（16进制）。

indexOfCP 的语法如下：

db.command.aggregate.indexOfCP([<目标字符串表达式>, <子字符串表达式>, <开始位置表达式>, <结束位置表达式>])

下面是 4 种表达式的详细描述：

| **表达式** | **描述** |
| --- | --- |
| 目标字符串表达式 | 任何可以被解析为字符串的表达式 |
| 子字符串表达式 | 任何可以被解析为字符串的表达式 |
| 开始位置表达式 | 任何可以被解析为非负整数的表达式 |
| 结束位置表达式 | 任何可以被解析为非负整数的表达式 |

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-70)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 indexOfCP 查找字符 "a" 在字段 firstName 中的位置：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

aStrIndex: $.indexOfCP(['$firstName', 'a'])

})

.end()

返回的结果如下：

{ "aStrIndex": 2 }

{ "aStrIndex": 5 }

{ "aStrIndex": -1 }

#### [split](https://uniapp.dcloud.io/uniCloud/cf-database?id=split)

聚合操作符。按照分隔符分隔数组，并且删除分隔符，返回子字符串组成的数组。如果字符串无法找到分隔符进行分隔，返回原字符串作为数组的唯一元素。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-71)

split 的语法如下：

db.command.aggregate.split([<字符串表达式>, <分隔符表达式>])

字符串表达式和分隔符表达式可以是任意形式的表达式，只要它可以被解析为字符串即可。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-71)

假设集合 students 的记录如下：

{ "birthday": "1999/12/12" }

{ "birthday": "1998/11/11" }

{ "birthday": "1997/10/10" }

通过 split 将每条记录中的 birthday 字段对应值分隔成数组，每个数组分别由代表年、月、日的3个元素组成：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

birthday: $.split(['$birthday', '/'])

})

.end()

返回的结果如下：

{ "birthday": [ "1999", "12", "12" ] }

{ "birthday": [ "1998", "11", "11" ] }

{ "birthday": [ "1997", "10", "10" ] }

#### [strLenBytes](https://uniapp.dcloud.io/uniCloud/cf-database?id=strlenbytes)

聚合操作符。计算并返回指定字符串中 utf-8 编码的字节数量。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-72)

strLenBytes 的语法如下：

db.command.aggregate.strLenBytes(<表达式>)

只要表达式可以被解析成字符串，那么它就是有效表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-72)

假设集合 students 的记录如下：

{ "name": "dongyuanxin", "nickname": "心谭" }

借助 strLenBytes 计算 name 字段和 nickname 字段对应值的字节长度：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

nameLength: $.strLenBytes('$name'),

nicknameLength: $.strLenBytes('$nickname')

})

.end()

返回结果如下：

{ "nameLength": 11, "nicknameLength": 6 }

#### [strLenCP](https://uniapp.dcloud.io/uniCloud/cf-database?id=strlencp)

聚合操作符。计算并返回指定字符串的UTF-8 [code points](http://www.unicode.org/glossary/#code_point) 数量。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-73)

strLenCP 的语法如下：

db.command.aggregate.strLenCP(<表达式>)

只要表达式可以被解析成字符串，那么它就是有效表达式。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-73)

假设集合 students 的记录如下：

{ "name": "dongyuanxin", "nickname": "心谭" }

借助 strLenCP 计算 name 字段和 nickname 字段对应值的UTF-8 [code points](http://www.unicode.org/glossary/#code_point)的数量：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

nameLength: $.strLenCP('$name'),

nicknameLength: $.strLenCP('$nickname')

})

.end()

返回结果如下：

{ "nameLength": 11, "nicknameLength": 2 }

#### [strcasecmp](https://uniapp.dcloud.io/uniCloud/cf-database?id=strcasecmp)

聚合操作符。对两个字符串在不区分大小写的情况下进行大小比较，并返回比较的结果。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-74)

strcasecmp 的语法如下：

db.command.aggregate.strcasecmp([<表达式1>, <表达式2>])

只要 表达式1和 表达式2 可以被解析成字符串，那么它们就是有效的。

返回的比较结果有1，0和-1三种：

* 1：表达式1 解析的字符串 > 表达式2 解析的字符串 - 0：表达式1 解析的字符串 = 表达式2 解析的字符串 - -1：表达式1 解析的字符串 < 表达式2 解析的字符串

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-74)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 strcasecmp 比较 firstName 字段值和 lastName 字段值的大小：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

result: $.strcasecmp(['$firstName', '$lastName']),

})

.end()

返回结果如下：

{ "result": 1 }

{ "result": 1 }

{ "result": -1 }

#### [substr](https://uniapp.dcloud.io/uniCloud/cf-database?id=substr)

聚合操作符。返回字符串从指定位置开始的指定长度的子字符串。它是 db.command.aggregate.substrBytes 的别名，更推荐使用后者。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-75)

substr 的语法如下：

db.command.aggregate.substr([<表达式1>, <表达式2>, <表达式3>])

表达式1 是任何可以解析为字符串的有效表达式，表达式2 和 表达式3 是任何可以解析为数字的有效表达式。

如果 表达式2 是负数，返回的结果为 ""。

如果 表达式3 是负数，返回的结果为从 表达式2 指定的开始位置以及之后其余部分的子字符串。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-75)

假设集合 students 的记录如下：

{ "birthday": "1999/12/12", "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "birthday": "1998/11/11", "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "birthday": "1997/10/10", "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 substr 可以提取 birthday 中的年、月、日信息，代码如下：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

year: $.substr(['$birthday', 0, 4]),

month: $.substr(['$birthday', 5, 2]),

day: $.substr(['$birthday', 8, -1])

})

.end()

返回的结果如下：

{ "day": "12", "month": "12", "year": "1999" }

{ "day": "11", "month": "11", "year": "1998" }

{ "day": "10", "month": "10", "year": "1997" }

#### [substrBytes](https://uniapp.dcloud.io/uniCloud/cf-database?id=substrbytes)

聚合操作符。返回字符串从指定位置开始的指定长度的子字符串。子字符串是由字符串中指定的 UTF-8 字节索引的字符开始，长度为指定的字节数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-76)

substrBytes 的语法如下：

db.command.aggregate.substrBytes([<表达式1>, <表达式2>, <表达式3>])

表达式1 是任何可以解析为字符串的有效表达式，表达式2 和 表达式3 是任何可以解析为数字的有效表达式。

如果 表达式2 是负数，返回的结果为 ""。

如果 表达式3 是负数，返回的结果为从 表达式2 指定的开始位置以及之后其余部分的子字符串。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-76)

假设集合 students 的记录如下：

{ "birthday": "1999/12/12", "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "birthday": "1998/11/11", "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "birthday": "1997/10/10", "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 substrBytes 可以提取 birthday 中的年、月、日信息，代码如下：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

year: $.substrBytes(['$birthday', 0, 4]),

month: $.substrBytes(['$birthday', 5, 2]),

day: $.substrBytes(['$birthday', 8, -1])

})

.end()

返回的结果如下：

{ "day": "12", "month": "12", "year": "1999" }

{ "day": "11", "month": "11", "year": "1998" }

{ "day": "10", "month": "10", "year": "1997" }

#### [substrCP](https://uniapp.dcloud.io/uniCloud/cf-database?id=substrcp)

聚合操作符。返回字符串从指定位置开始的指定长度的子字符串。子字符串是由字符串中指定的 UTF-8 字节索引的字符开始，长度为指定的字节数。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-77)

substrCP 的语法如下：

db.command.aggregate.substrCP([<表达式1>, <表达式2>, <表达式3>])

表达式1 是任何可以解析为字符串的有效表达式，表达式2 和 表达式3 是任何可以解析为数字的有效表达式。

如果 表达式2 是负数，返回的结果为 ""。

如果 表达式3 是负数，返回的结果为从 表达式2 指定的开始位置以及之后其余部分的子字符串。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-77)

假设集合 students 的记录如下：

{ "name": "dongyuanxin", "nickname": "心谭" }

借助 substrCP 可以提取 nickname 字段值的第一个汉字：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

firstCh: $.substrCP(['$nickname', 0, 1])

})

.end()

返回的结果如下：

{ "firstCh": "心" }

#### [toLower](https://uniapp.dcloud.io/uniCloud/cf-database?id=tolower)

聚合操作符。将字符串转化为小写并返回。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-78)

toLower 的语法如下：

db.command.aggregate.toLower(表达式)

只要表达式可以被解析成字符串，那么它就是有效表达式。例如：$ + 指定字段。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-78)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 toLower 将 firstName 的字段值转化为小写：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

result: $.toLower('$firstName'),

})

.end()

返回的结果如下：

{ "result": "yuanxin" }

{ "result": "weijia" }

{ "result": "chengxi" }

#### [toUpper](https://uniapp.dcloud.io/uniCloud/cf-database?id=toupper)

聚合操作符。将字符串转化为大写并返回。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-79)

toUpper 的语法如下：

db.command.aggregate.toUpper(表达式)

只要表达式可以被解析成字符串，那么它就是有效表达式。例如：$ + 指定字段。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-79)

假设集合 students 的记录如下：

{ "firstName": "Yuanxin", "group": "a", "lastName": "Dong", "score": 84 }

{ "firstName": "Weijia", "group": "a", "lastName": "Wang", "score": 96 }

{ "firstName": "Chengxi", "group": "b", "lastName": "Li", "score": 80 }

借助 toUpper 将 lastName 的字段值转化为大写：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.project({

\_id: 0,

result: $.toUpper('$lastName'),

})

.end()

返回的结果如下：

{ "result": "DONG" }

{ "result": "WANG" }

{ "result": "LI" }

### [累计器操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%b4%af%e8%ae%a1%e5%99%a8%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [addToSet](https://uniapp.dcloud.io/uniCloud/cf-database?id=addtoset)

聚合操作符。聚合运算符。向数组中添加值，如果数组中已存在该值，不执行任何操作。它只能在 group stage 中使用。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-80)

addToSet 语法如下：

db.command.aggregate.addToSet(<表达式>)

表达式是形如 $ + 指定字段 的字符串。如果指定字段的值是数组，那么整个数组会被当作一个元素。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-80)

假设集合 passages 的记录如下：

{ "category": "web", "tags": [ "JavaScript", "CSS" ], "title": "title1" }

{ "category": "System", "tags": [ "C++", "C" ], "title": "title2" }

**非数组字段**

每条记录的 category 对应值的类型是非数组，利用 addToSet 统计所有分类：

const $ = db.command.aggregate

db

.collection('passages')

.aggregate()

.group({

\_id: null,

categories: $.addToSet('$category')

})

.end()

返回的结果如下：

{ "\_id": null, "categories": [ "System", "web" ] }

**数组字段**

每条记录的 tags 对应值的类型是数组，数组不会被自动展开：

const $ = db.command.aggregate

db

.collection('passages')

.aggregate()

.group({

\_id: null,

tagsList: $.addToSet('$tags')

})

.end()

返回的结果如下：

{ "\_id": null, "tagsList": [ [ "C++", "C" ], [ "JavaScript", "CSS" ] ] }

#### [avg](https://uniapp.dcloud.io/uniCloud/cf-database?id=avg)

聚合操作符。返回一组集合中，指定字段对应数据的平均值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-81)

avg 的语法如下：

db.command.aggregate.avg(<number>)

avg 传入的值除了数字常量外，也可以是任何最终解析成一个数字的表达式。它会忽略非数字值。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-81)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

借助 avg 可以计算所有记录的 score 的平均值：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.group({

\_id: null,

average: $.avg('$score')

})

.end()

返回的结果如下：

{ "\_id": null, "average": 90 }

#### [first](https://uniapp.dcloud.io/uniCloud/cf-database?id=first)

聚合操作符。返回指定字段在一组集合的第一条记录对应的值。仅当这组集合是按照某种定义排序（ sort ）后，此操作才有意义。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-82)

first 的语法如下：

db.command.aggregate.first(<表达式>)

表达式是形如 $ + 指定字段 的字符串。

first 只能在 group 阶段被使用，并且需要配合 sort 才有意义。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-82)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

如果需要得到所有记录中 score 的最小值，可以先将所有记录按照 score 排序，然后取出第一条记录的 first。

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.sort({

score: 1

})

.group({

\_id: null,

min: $.first('$score')

})

.end()

返回的数据结果如下：

{ "\_id": null, "min": 80 }

#### [last](https://uniapp.dcloud.io/uniCloud/cf-database?id=last)

聚合操作符。返回指定字段在一组集合的最后一条记录对应的值。仅当这组集合是按照某种定义排序（ sort ）后，此操作才有意义。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-83)

last 的语法如下：

db.command.aggregate.last(<表达式>)

表达式是形如 $ + 指定字段 的字符串。

last 只能在 group 阶段被使用，并且需要配合 sort 才有意义。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-83)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

如果需要得到所有记录中 score 的最大值，可以先将所有记录按照 score 排序，然后取出最后一条记录的 last。

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.sort({

score: 1

})

.group({

\_id: null,

max: $.last('$score')

})

.end()

返回的数据结果如下：

{ "\_id": null, "max": 100 }

#### [max](https://uniapp.dcloud.io/uniCloud/cf-database?id=max)

聚合操作符。返回一组数值的最大值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-84)

max 的语法如下：

db.command.aggregate.max(<表达式>)

表达式是形如 $ + 指定字段 的字符串。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-84)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

借助 max 可以统计不同组（ group ）中成绩的最高值，代码如下：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.group({

\_id: '$group',

maxScore: $.max('$score')

})

.end()

返回的数据结果如下：

{ "\_id": "b", "maxScore": 100 }

{ "\_id": "a", "maxScore": 96 }

```.

#### [mergeObjects](https://uniapp.dcloud.io/uniCloud/cf-database?id=mergeobjects-1)

见[mergeObjects](https://uniapp.dcloud.io/uniCloud/cf-database?id=mergeobjects)

#### [min](https://uniapp.dcloud.io/uniCloud/cf-database?id=min)

聚合操作符。返回一组数值的最小值。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-85)

min 的语法如下：

db.command.aggregate.min(<表达式>)

表达式是形如 $ + 指定字段 的字符串。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-85)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

借助 min 可以统计不同组（ group ）中成绩的最低值，代码如下：

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.group({

\_id: '$group',

minScore: $.min('$score')

})

.end()

返回的数据结果如下：

{ "\_id": "b", "minScore": 80 }

{ "\_id": "a", "minScore": 84 }

#### [push](https://uniapp.dcloud.io/uniCloud/cf-database?id=push-1)

聚合操作符。在 group 阶段，返回一组中表达式指定列与对应的值，一起组成的数组。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-86)

push 语法如下：

db.command.aggregate.push({

<字段名1>: <指定字段1>,

<字段名2>: <指定字段2>,

...

})

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-86)

假设集合 students 的记录如下：

{ "group": "a", "name": "stu1", "score": 84 }

{ "group": "a", "name": "stu2", "score": 96 }

{ "group": "b", "name": "stu3", "score": 80 }

{ "group": "b", "name": "stu4", "score": 100 }

借助 push 操作，对不同分组( group )的所有记录，聚合所有数据并且将其放入一个新的字段中，进一步结构化和语义化数据。

const $ = db.command.aggregate

db

.collection('students')

.aggregate()

.group({

\_id: '$group',

students: $.push({

name: '$name',

score: '$score'

})

})

.end()

输出结果如下：

{ "\_id": "b", "students": [{ "name": "stu3", "score": 80 }, { "name": "stu4", "score": 100 }] }

{ "\_id": "a", "students": [{ "name": "stu1", "score": 84 }, { "name": "stu2", "score": 96 }] }

#### [stdDevPop](https://uniapp.dcloud.io/uniCloud/cf-database?id=stddevpop)

聚合操作符。返回一组字段对应值的标准差。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-87)

stdDevPop 的使用形式如下：

db.command.aggregate.stdDevPop(<表达式>)

表达式传入的是指定字段，指定字段对应的值的数据类型必须是 number ，否则结果会返回 null。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-87)

假设集合 students 的记录如下：a 组同学的成绩分别是84和96，b组同学的成绩分别是80和100。

{ "group":"a", "score":84 }

{ "group":"a", "score":96 }

{ "group":"b", "score":80 }

{ "group":"b", "score":100 }

可以用 stdDevPop 来分别计算 a 和 b 两组同学成绩的标准差，以此来比较哪一组同学的成绩更稳定。代码如下：

const $ = db.command.aggregate

db.collection('students').aggregate()

.group({

\_id: '$group',

stdDev: $.stdDevPop('$score')

})

.end()

返回的数据结果如下：

{ "\_id": "b", "stdDev": 10 }

{ "\_id": "a", "stdDev": 6 }

#### [stdDevSamp](https://uniapp.dcloud.io/uniCloud/cf-database?id=stddevsamp)

聚合操作符。计算输入值的样本标准偏差。如果输入值代表数据总体，或者不概括更多的数据，请改用 db.command.aggregate.stdDevPop。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-88)

stdDevSamp 的使用形式如下：

db.command.aggregate.stdDevSamp(<表达式>)

表达式传入的是指定字段，stdDevSamp 会自动忽略非数字值。如果指定字段所有的值均是非数字，那么结果返回 null。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-88)

假设集合 students 的记录如下：

{ "score": 80 }

{ "score": 100 }

可以用 stdDevSamp 来计算成绩的标准样本偏差。代码如下：

const $ = db.command.aggregate

db.collection('students').aggregate()

.group({

\_id: null,

ageStdDev: $.stdDevSamp('$score')

})

.end()

返回的数据结果如下：

{ "\_id": null, "ageStdDev": 14.142135623730951 }

如果向集合 students 添加一条新记录，它的 score 字段类型是 string：

{ "score": "aa" }

用上面代码计算标准样本偏差时，stdDevSamp 会自动忽略类型不为 number 的记录，返回结果保持不变。

#### [sum](https://uniapp.dcloud.io/uniCloud/cf-database?id=sum)

聚合操作符。计算并且返回一组字段所有数值的总和。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-89)

sum 的使用形式如下：

db.command.aggregate.sum(<表达式>)

表达式可以传入指定字段，也可以传入指定字段组成的列表。sum 会自动忽略非数字值。如果字段下的所有值均是非数字，那么结果返回 0。若传入数字常量，则当做所有记录该字段的值都给给定常量，在聚合时相加，最终值为输入记录数乘以常量。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-89)

假设代表商品的集合 goods 的记录如下：price 代表商品销售额，cost 代表商品成本

{ "cost": -10, "price": 100 }

{ "cost": -15, "price": 1 }

{ "cost": -10, "price": 10 }

**单独字段**

借助 sum 可以计算所有商品的销售总和，代码如下：

const $ = db.command.aggregate

db

.collection('goods')

.aggregate()

.group({

\_id: null,

totalPrice: $.sum('$price')

})

.end()

返回的数据结果如下：销售额是 111

{ "\_id": null, "totalPrice": 111 }

**字段列表**

如果需要计算所有商品的利润总额，那么需要将每条记录的 cost 和 price 相加得到此记录对应商品的利润。最后再计算所有商品的利润总额。

借助 sum，代码如下：

const $ = db.command.aggregate

db

.collection('goods')

.aggregate()

.group({

\_id: null,

totalProfit: $.sum(

$.sum(['$price', '$cost'])

)

})

.end()

返回的数据结果如下：利润总额为 76

{ "\_id": null, "totalProfit": 76 }

### [变量操作符](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e5%8f%98%e9%87%8f%e6%93%8d%e4%bd%9c%e7%ac%a6)

#### [let](https://uniapp.dcloud.io/uniCloud/cf-database?id=let)

聚合操作符。自定义变量，并且在指定表达式中使用，返回的结果是表达式的结果。

##### [API 说明](https://uniapp.dcloud.io/uniCloud/cf-database?id=api-%e8%af%b4%e6%98%8e-90)

let 的语法如下：

db.command.aggregate.let({

vars: {

<变量1>: <变量表达式>,

<变量2>: <变量表达式>,

...

},

in: <结果表达式>

})

vars 中可以定义多个变量，变量的值由 变量表达式 计算而来，并且被定义的变量只有在 in 中的 结果表达式 才可以访问。

在 in 的结果表达式中访问自定义变量时候，请在变量名前加上双美元符号( $$ )并用引号括起来。

##### [示例代码](https://uniapp.dcloud.io/uniCloud/cf-database?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-90)

假设代表商品的集合 goods 的记录如下：price 代表商品价格，discount 代表商品折扣率，cost 代表商品成本

{ "cost": -10, "discount": 0.95, "price": 100 }

{ "cost": -15, "discount": 0.98, "price": 1 }

{ "cost": -10, "discount": 1, "price": 10 }

借助 let 可以定义并计算每件商品实际的销售价格，并将其赋值给自定义变量 priceTotal。最后再将 priceTotal 与 cost 进行取和( sum )运算，得到每件商品的利润。

代码如下：

const $ = db.command.aggregate

db

.collection('goods')

.aggregate()

.project({

profit: $.let({

vars: {

priceTotal: $.multiply(['$price', '$discount'])

},

in: $.sum(['$$priceTotal', '$cost'])

})

})

.end()

返回的数据结果如下：

{ "profit": 85 }

{ "profit": -14.02 }

{ "profit": 0 }

# 云存储

开发者使用uniCloud的云存储，无需再像传统模式那样单独去购买存储空间、CDN映射、流量采购等；uniCloud云存储支持文本、图片和其他由用户生成的内容存储到云端，并提供CDN下载地址，

开发者可在客户端使用云存储API，文件上传成功后，系统会自动生成一个资源链接，开发者需保存该文件地址供后续业务下载使用。

即将支持云函数中使用云存储功能。

## [uploadFile(Object object)](https://uniapp.dcloud.io/uniCloud/storage?id=uploadfile)

上传文件到云存储，**阿里云单文件大小限制为100M，腾讯云单文件最大为5G**

#### [请求参数](https://uniapp.dcloud.io/uniCloud/storage?id=%e8%af%b7%e6%b1%82%e5%8f%82%e6%95%b0)

**Object object**

| **参数名** | **类型** | **必填** | **默认值** | **说明** | **平台差异说明** |
| --- | --- | --- | --- | --- | --- |
| filePath | String | 是 | - | 要上传的文件对象 | - |

#### [响应参数](https://uniapp.dcloud.io/uniCloud/storage?id=%e5%93%8d%e5%ba%94%e5%8f%82%e6%95%b0)

| **字段** | **类型** | **说明** |
| --- | --- | --- |
| code | String | 状态码，操作成功则不返回 |
| message | String | 错误描述 |
| fileID | String | 文件唯一 ID，用来访问文件，建议存储起来 |
| requestId | String | 请求序列号，用于错误排查 |

#### [示例代码](https://uniapp.dcloud.io/uniCloud/storage?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81)

uni.chooseImage({

count: 1,

success(res) {

console.log(res);

if (res.tempFilePaths.length > 0) {

let filePath = res.tempFilePaths[0]

//进行上传操作

// promise

const result = await uniCloud.uploadFile({

filePath: filePath

});

// callback

uniCloud.uploadFile({

filePath: filePath

},

success() {},

fail() {},

complete() {}

});

}

}

});

**Tips**

* 阿里云返回的fileID为链接形式

## [deleteFile(Object object)](https://uniapp.dcloud.io/uniCloud/storage?id=deletefile)

删除云端文件

#### [请求参数](https://uniapp.dcloud.io/uniCloud/storage?id=%e8%af%b7%e6%b1%82%e5%8f%82%e6%95%b0-1)

**Object object**

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| fileList | <Array>.String | 是 | 要删除的文件 ID 组成的数组，**阿里云只支持一次删除一个文件** |

#### [响应参数](https://uniapp.dcloud.io/uniCloud/storage?id=%e5%93%8d%e5%ba%94%e5%8f%82%e6%95%b0-1)

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| code | String | 否 | 状态码，操作成功则不返回 |
| message | String | 否 | 错误描述 |
| fileList | <Array>.Object | 否 | 删除结果组成的数组 |
| requestId | String | 否 | 请求序列号，用于错误排查 |

**fileList定义**

| **字段** | **类型** | **必填** | **说明** |
| --- | --- | --- | --- |
| code | String | 否 | 删除结果，成功为 SUCCESS |
| fileID | String | 是 | 文件 ID |

#### [示例代码](https://uniapp.dcloud.io/uniCloud/storage?id=%e7%a4%ba%e4%be%8b%e4%bb%a3%e7%a0%81-1)

// promise

uniCloud

.deleteFile({

fileList: ['cloud://jimmytest-088bef/1534576354877.jpg']

})

.then(res => {});

// callback

uniCloud.deleteFile(

{

fileList: ['cloud://jimmytest-088bef/1534576354877.jpg'],

success(){},

fail(){},

complete(){}

}

);

# 日志输出

## [普通日志](https://uniapp.dcloud.io/uniCloud/cf-logger?id=%e6%99%ae%e9%80%9a%e6%97%a5%e5%bf%97)

开发者在云函数内也可以使用console.log、console.info、console.warn、console.error四种方式打印日志。

HBuilderX中查看日志的教程在 [快速上手章节](https://uniapp.dcloud.io/quickstart?id=%e8%bf%90%e8%a1%8c%e5%92%8c%e8%b0%83%e8%af%95%e4%ba%91%e5%87%bd%e6%95%b0)

## [高级日志](https://uniapp.dcloud.io/uniCloud/cf-logger?id=%e9%ab%98%e7%ba%a7%e6%97%a5%e5%bf%97)

高级日志包括以下四种，以此方式输出的日志会持久化存储（有效期30天）。

| **接口** | **描述** |
| --- | --- |
| uniCloud.logger.log | 以 log 日志等级输出日志 |
| uniCloud.logger.info | 以 info 日志等级输出日志 |
| uniCloud.logger.warn | 以 warn 日志等级输出日志 |
| uniCloud.logger.error | 以 error 日志等级输出日志 |

# 同时连多服务器空间

**若项目仅连接一个服务空间，uniCloud框架会自动绑定服务空间，开发者无需手动做初始化工作（可理解为类调用）。只有存在多服务空间时，才需要根据本文进行初始化**

//项目仅连接了一个服务空间，则无需初始化

//可通过uniCloud直接调用云开发的API

uniCloud.callFunction()

uniCloud.uploadFile()

若项目连接了多个服务空间，uniCloud无法自动绑定；需开发者在客户端代码中，手动调用初始化方法uniCloud.init，绑定服务空间。

uniCloud.init方法会返回一个uniCloud实例，之后云开发API的调用都需要通过该uniCloud实例发起（类似实例调用）。

uniCloud.init方法定义如下：

function init(options):uniCloud

uniCloud.init方法接受一个options参数，返回uniCloud实例，uniCloud实例可调用云函数、云存储相关API。

**options 参数说明**

| **参数名** | **类型** | **必填** | **默认值** | **说明** |
| --- | --- | --- | --- | --- |
| provider | String | 是 | - | 目前仅支持aliyun |
| spaceId | String | 是 | - | 服务空间ID，**注意是服务空间ID，不是服务空间名称** |
| clientSecret | String | 是 | - | 可以在[uniCloud控制台](https://unicloud.dcloud.net.cn/" \t "_blank)服务空间列表中查看 |

**示例代码**

//开发者创建了多个服务空间，则需手动初始化

const myCloud = uniCloud.init({

provider: 'aliyun',

spaceId: 'xxxx-yyy',

clientSecret: 'xxxx'

});

//通过uniCloud实例调用云开发的API

myCloud.callFunction()

myCloud.uploadFile()

**Tips：**

* 云函数会自动识别自己所属的服务空间，无需初始化。
* HBuilderX 2.5.8版本在main.js使用uniCloud时，运行到H5端会报错，可以先使用其他平台进行调试。此问题会在HBuilderX 2.5.9修复

# 提示及源码

如下是使用 uniCloud 的插件市场项目或开源案例。欢迎大家提交插件或开源项目，可以以pr方式直接编辑本文档。

* [Hello uniCLoud]：在HBuilderX 2.5.8+版本，新建uni-app项目的项目模板中有 hello unicloud模板，展示了uniCloud的云函数基本用法、cdn使用方式。它对应的H5演示地址是：[https://hellounicloud.m3w.cn/](https://hellounicloud.m3w.cn/" \t "_blank)
* [uniCloud基础登录示例](https://ext.dcloud.net.cn/plugin?id=1268)：包含账号密码登录注册，微信小程序登录，token验证功能的uniCloud示例
* [uni抗疫开源项目汇总](https://gitee.com/dcloud/xinguan2020)：这是一个项目集合汇总，里面有大量与抗疫项目的开源项目，均基于uniCloud。包括外来人员登记系统、学生健康报备系统、员工疫情筛查工具、消毒检查登记系统、物资管理系统等。这些项目具备一定通用性，可以稍加改造用于其他行业应用。

# 常见问题

**uniCloud和微信小程序云开发、支付宝小程序云开发有何区别？**

微信、支付宝、百度的小程序，均提供了云开发。但它们都仅支持自家小程序，无法跨端。

uniCloud和微信小程序云开发、支付宝小程序云开发使用相同的基础建设平台，微信小程序云开发背后是腾讯云的TCB团队，支付宝小程序云开发背后是阿里小程序云团队。uniCloud是DCloud和阿里小程序云团队、腾讯云的TCB团队展开深层次合作，在他们底层资源的基础上进行二次封装，提供的跨端云开发方案。

简单来说，uniCloud和微信小程序云开发、支付宝小程序云开发一样稳定健壮，但可以跨更多平台。不管你在uniCloud里选择了阿里还是腾讯的serverless，均可以跨端使用。

**uniCloud稳定吗？DCloud服务器异常会影响我的线上业务吗？**

uniCloud是 DCloud 和阿里云、腾讯云等成熟云厂商合作推出的云服务产品，阿里云、腾讯云等提供云端基础资源，DCloud提供前端框架的封装、IDE工具支持、插件生态等服务，开发者的云函数直接托管在阿里云等服务商平台，用户终端App运行时，直连云服务商平台，不会经过DCloud服务器，开发者无需担心因DCloud服务器负载而影响自己业务的问题。

**云函数 和 传统 Node.js 开发有何区别？**

云函数相当于 Node.js + serverless + DCloud改进。

* 传统Node.js开发需要购买服务器，安装Node.js环境，部署 pm2 等守护进程；云函数无需考虑服务器环境，只需专心实现业务代码，然后将云函数一键上传，云服务商负责云函数运行环境的准备。
* 传统Node.js开发模式，开发者需监控服务器参数，比如硬盘使用率，避免服务器负载过高导致业务中断；云函数模式下，开发者无需关心云函数运行的宿主环境，云厂商会实现服务调配及硬件监控。
* 用户量较大时，传统Node.js开发需考虑购买更多服务器并实现负载均衡；云函数模式下，云服务商自动弹性扩容，开发者无需担心服务器扛不住压力。
* 传统Node.js开发模式，需考虑安全防护，比如DDos攻击；云函数模式，云厂商的API网关会做拦截防护，开发者无需关心，并可节省高防IP等费用

总结一下，前端同学即便可熟练编写Node.js代码，但对于DB优化、弹性扩容、攻击防护、灾备处理等方面还是有经验欠缺的，但uniCloud将这些都封装好了，真正做到仅专注业务实现，其它都委托云厂商服务。

另外，在 Node.js 代码实现上，云函数每次执行的宿主环境（可简单理解为虚拟机或服务器硬件）可能相同，也可能不同，因此传统Node.js开发中将部分信息存储本地硬盘或内存的方案就不再适合，建议通过云数据库或云存储的方案替代。

**uniCloud只支持uni-app，怎么开发web界面？**

uni-app可以开发web界面，注意3个问题：

1. 如果要多端适配界面，使用css的媒体查询处理适配。
2. 网上有三方库可以替换touch的拖动为pc上的drag。比如hello uni-app的h5示例使用的touch-emulator.js。
3. uni-app的内置组件和api仅适配了webkit内核浏览器，ie和firefox可能有兼容问题。如有问题需自己写额外css或js适配。

后续DCloud会进一步强化内置组件和uni-ui对PC浏览器的适配。

**腾讯、阿里的serverless有什么大案例？**

* 微信小程序云开发，已经有50万开发者，包括腾讯自有的很多大日活应用都构建在腾讯云serverless上，如微信生活缴费、乘车码、微信读书、腾讯新闻、腾讯相册等。
* 2019年双11，阿里部分业务已经迁移在serverless上。支付宝小程序也提供了云开发功能。

**uniCloud费用贵不贵？**

目前uniCloud处于公测期，是免费的。未来uniCloud的租用费用，也会低于租用传统云主机的费用。

# 更新日志

# [uniCloud 更新日志](https://uniapp.dcloud.io/uniCloud/release?id=unicloud-%e6%9b%b4%e6%96%b0%e6%97%a5%e5%bf%97)

======================================

#### [2020-02-26](https://uniapp.dcloud.io/uniCloud/release?id=_2020-02-26)

* 【服务端SDK】
  + 阿里云 新增 云数据库支持 add 批量添加数据 [详情](https://uniapp.dcloud.io/uniCloud/cf-database?id=add)

#### [2020-02-24](https://uniapp.dcloud.io/uniCloud/release?id=_2020-02-24)

* 【服务端SDK】
  + 阿里云 新增 云函数互调功能 [详情](https://uniapp.dcloud.io/uniCloud/cf-functions?id=callbyfunction)