

尚硅谷大数据技术之 InfluxDB

(作者：尚硅谷研究院)

版本：V1.0

第1章 认识 InfluxDB

1.1 InfluxDB 的使用场景

InfluxDB 是一种时序数据库，时序数据库通常被用在监控场景，比如运维和 IOT（物联网）领域。这类数据库旨在存储时序数据并实时处理它们。

比如。我们可以写一个程序将服务器上 CPU 的使用情况每隔 10 秒钟向 InfluxDB 中写入一条数据。接着，我们写一个查询语句，查询过去 30 秒 CPU 的平均使用情况，然后让这个查询语句也每隔 10 秒钟执行一次。最终，我们配置一条报警规则，如果查询语句的执行结果>xxx，就立刻触发报警。

上述就是一个指标监控的场景，在 IOT 领域中，也有大量的指标需要我们监控。比如，机械设备的轴承震动频率，农田的湿度温度等等。

1.2 为什么不用关系型数据库

1.2.1 写入性能

关系型数据库也是支持时间戳的，也能够基于时间戳进行查询。但是，从我们的使用场景出发，需要注意数据库的写入性能。通常，关系型数据库会采用 B+树数据结构，在数据写入时，有可能会触发叶裂变，从而产生了对磁盘的随机读写，降低写入速度。

当前市面上的时序数据库通常都是采用 LSM Tree 的变种，顺序写磁盘来增强数据的写入能力。网上有不少关于性能测试的文章，同学们可以自己去参考学习，通常时序数据库都会保证在单点每秒数十万的写入能力。

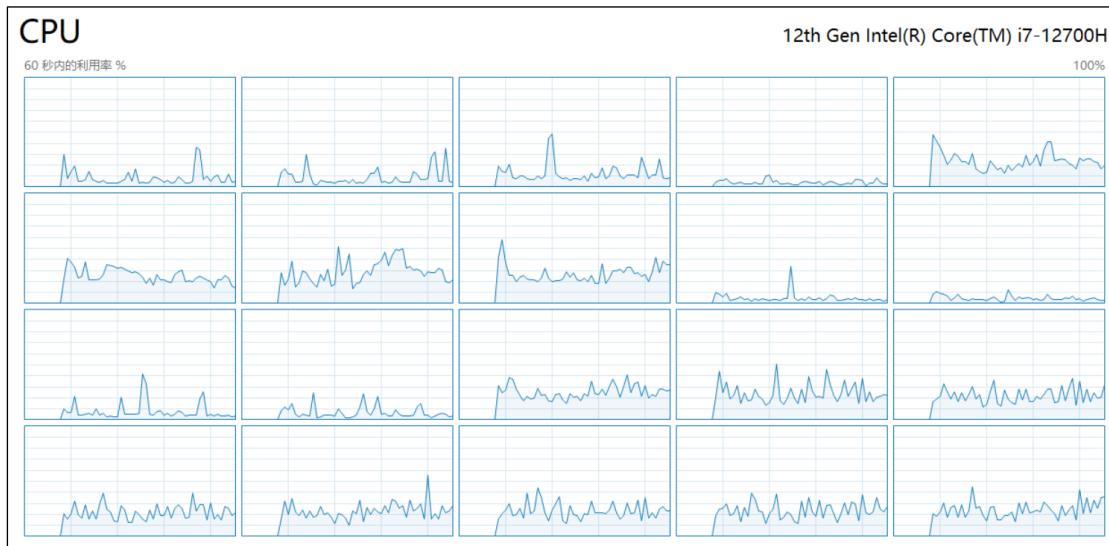
1.2.2 数据价值

我们之前说，时序数据库一般用于指标监控场景。这个场景的数据有一个非常明显的特点就是冷热差别明显。通常，指标监控只会使用近期一段时间的数据，比如我只查询某个设备最近 10 分钟的记录，10 分钟前的数据我就不再用了。那么这 10 分钟前的数据，对我们来说就是冷数据，应该被压缩放到磁盘里去来节省空间。而热数据因为经常要用，数

据库就应该让它留在内存里，等待查询。而市面上的时序数据库大都有类似的设计。

1.2.3 时间不可倒流，数据只写不改

时序数据是描述一个实体在不同时间所处的不同状态。



就像是我们打开任务管理器，查看 CPU 的使用情况。我发现 CPU 占用率太高了，于是杀死了其中一个进程，但 10 秒前的数据不会因为我关闭进程再发生改变了。

这是时序数据的一大特点。与之相应，时序数据库基本上是插入操作较多，而且还没有什么更新需求。

1.3 1.X 的 TICK 技术栈与 2.X 的进一步融合

根据上文的介绍，我们首先可以知道时序数据一般用在监控场景。大体上，数据的应用可以分为 4 步走。

- (1) 数据采集
- (2) 存储
- (3) 查询（包括聚合操作）
- (4) 报警

这样一看，只给一个数据库其实只能完成数据的存储和查询功能，上游的采集和下游的报警都需要自己来实现。因此 InfluxData 在 InfluxDB1.X 的时候推出了 TICK 生态来推出 start 全套的解决方案。

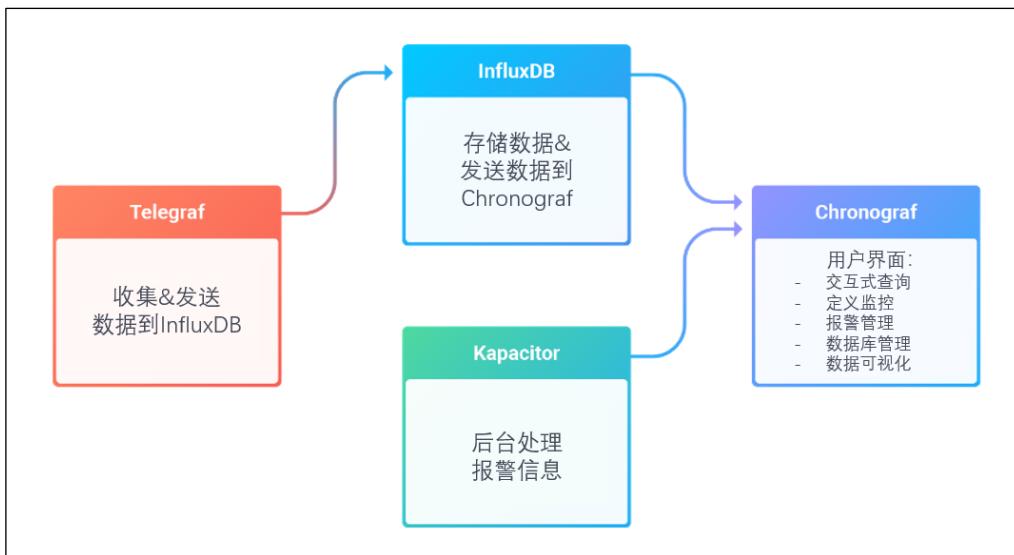
TICK4 个字母分别对应 4 个组件。

● T : Telegraf - 数据采集组件，收集&发送数据到 InfluxDB。

● I : InfluxDB - 存储数据&发送数据到 Chronograf。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- C : Chronograf - 总的用户界面，起到总的管理功能。
- K : Kapacitor - 后台处理报警信息。



到了 2.x, TICK 进一步融合, ICK 的功能全部融入了 InfluxDB, 仅需安装 InfluxDB 就能得到一个管理页面, 而且附带了定时任务和报警功能。

1.4 influxDB 版本比较与选型

1.4.1 版本特性比较

2020 年 InfluxDB 推出了 2.0 的正式版。2.x 同 1.x 相比, 底层引擎原理相差不大, 但会涉及一些概念的转变 (例如 db/rp 换成了 org/bucket)。另外, 对于 TICK 生态来说, 1.x 需要自己配置各个组件。2.x 则是更加方便集成, 有很棒的管理页面。

另外, 在查询语言方面, 1.x 是使用 InfluxQL 进行查询, 它的风格近似 SQL。2.x 推出了 FLUX 查询语言, 可以使用函数与管道符, 是一种更符合时序数据特性的更具表现力的查询语言。

1.4.2 选型, 本文档使用 InfluxDB 2.4

- 市场现状: 目前企业里面用 InfluxDB 1.X 和 InfluxDB 2.X 都有人在用, 数量上 InfluxDB1.X 占多一些。
- 易用性: 在开发中, InfluxDB 1.X 集成生态会比较麻烦, InfluxDB 2.X 相对来说更加便利。
- 性能: InfluxDB 1.X 和 2.X 的内核原理基本一致, 性能上差距不大。
- 集群: InfluxDB 从 0.11 版本开始, 就闭源了集群功能的代码。也就是说, 你只能免

费试用 InfluxDB 的单节点版（开源），想要集群等功能就需要购买企业版。不过就 InfluxDB 1.8 来说，有开源项目根据 0.11 的代码思路提供了 InfluxDB 开源的集群方案。也有开源项目给 InfluxDB 2.3 增加了反向代理功能，让我们可以横向拓展 InfluxDB 的服务能力。项目参考地址：

InfluxDB Cluster 对应 1.8.10: <https://github.com/chengshiwen/influxdb-cluster>

InfluxDB Proxy 对应 1.2 - 1.8: <https://github.com/chengshiwen/influx-proxy>

InfluxDB Proxy 对应 2.3: <https://github.com/chengshiwen/influx-proxy/tree/influxdb-v2>

● FLUX 语言支持：自 InfluxDB 1.7 和 InfluxDB 2.0 以来，InfluxDB 推出了一门独立的新的查询语言 FLUX，而且作为一个独立的项目来运作。InfluxData 公司希望 FLUX 语言能够成为一个像 SQL 一样的通用标准，而不仅仅是查询 InfluxDB 的特定语言。而且不管是你是选择 InfluxDB 1.X 还是 2.X 最终都会接触到 FLUX。不过 2.X 对 FLUX 的支持性要更好一些。

● InfluxDB 产品概况：

- InfluxDB 1.8 在小版本上还在更新，主要是修复一些 BUG，不再添加新特性
- InfluxDB 2.4 这是 InfluxDB 较新的版本，仍然在增加新的特性。
- InfluxDB 企业版 1.9 需要购买，相比开源版，它有集群功能。
- InfluxDB Cloud，免部署，跑在 InfluxData 公司的云服务器上，你可以使用客户端来操作。功能上对应开源版的 2.4

● 2.x 与 1.x 的主要区别：两个版本的内核原理基本一致，性能上的差别不大。差别主要是在，权限管理方式不同，2.x TICK 的集成性比 1.x 好，1.x 中的 database 到了 2.x 中变成了 bucket 等。

最终，本课程选择 Influx 2.4 来进行教学，学会使用 InfluxDB 2.4 后应当也能胜任 InfluxDB 1.7 及以上版本的开发。授课过程遇到与 InfluxDB 1.8 不同的地方，会做一些提醒。

第2章 安装部署 InfluxDB

2.1 下载安装

在 linux 环境下有两种安装方式

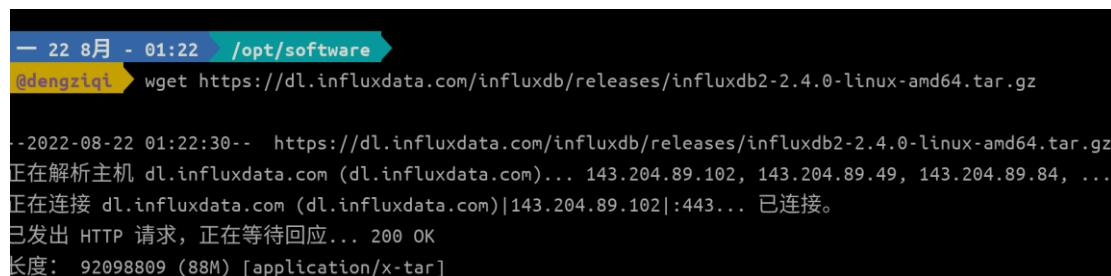
- 通过包管理工具安装，比如 apt 和 yum
- 直接下载可执行二进制程序的压缩包

本课程选用第二种方式，你可以使用下面的命令下载程序包。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
wget https://dl.influxdata.com/influxdb/releases/influxdb2-2.4.0-  
linux-amd64.tar.gz
```

如果下载失败的话，你也可以在课程资料中获取此安装包。课程资料可以通过关注尚硅谷微信公众号，回复“大数据”关键字领取。



A terminal window showing the command `wget https://dl.influxdata.com/influxdb/releases/influxdb2-2.4.0-linux-amd64.tar.gz` being run. The output shows the download progress, including the URL, host information, connection details, and file size (92098809 bytes).

```
--2022-08-22 01:22:30-- https://dl.influxdata.com/influxdb/releases/influxdb2-2.4.0-linux-amd64.tar.gz  
正在解析主机 dl.influxdata.com (dl.influxdata.com)... 143.204.89.102, 143.204.89.49, 143.204.89.84, ...  
正在连接 dl.influxdata.com (dl.influxdata.com)|143.204.89.102|:443... 已连接。  
已发出 HTTP 请求, 正在等待回应... 200 OK  
长度: 92098809 (88M) [application/x-tar]
```

压缩包下载好后，将其解压到目标路径。

```
tar -zxvf influxdb2-2.4.0-linux-amd64.tar.gz -C /opt/module
```

Go 语言开发的项目一般来说会只打包成单独的二进制可执行文件，也就是解压后目录下的 `influxd` 文件，这一文件中全是编译后的本地码，可以直接跑在操作系统上，不需要安装额外的运行环境或者依赖。

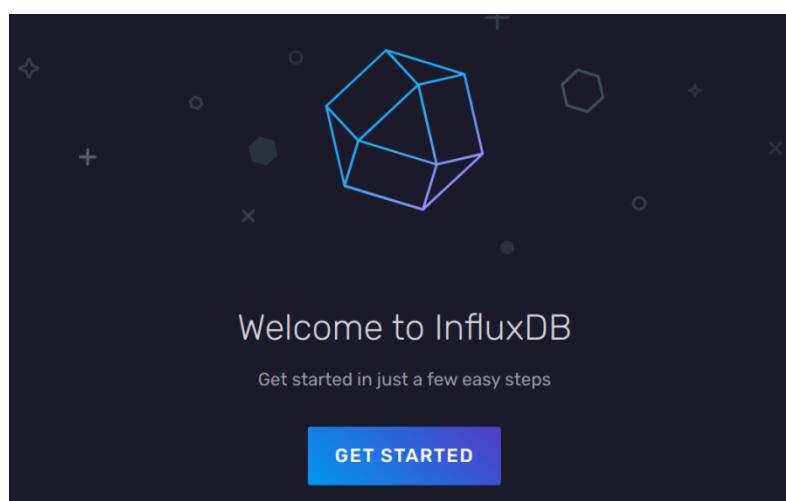
```
-rwxr-xr-x. 1 atguigu atguigu 150164784 Aug 19 03:44 influxd  
-rw-rw-r--. 1 atguigu atguigu       1067 Aug 19 03:44 LICENSE  
-rw-rw-r--. 1 atguigu atguigu      9830 Aug 19 03:44 README.md
```

现在，可以运行使用下面的命令，正式开启 InfluxDB 服务进程。

```
./influxd
```

2.2 进行初始化配置

使用浏览器访问 <http://hadoop102:8086>。如果是安装后的首次使用，InfluxDB 会返回一个初始化的引导界面。按照给定步骤完成操作就好。



2.2.1 创建用户和初始化存储桶

点击 GET STARTED 按钮，进入下一个步骤（添加用户）。如图所示，你需要填写、组织名称、用户名、用户密码。

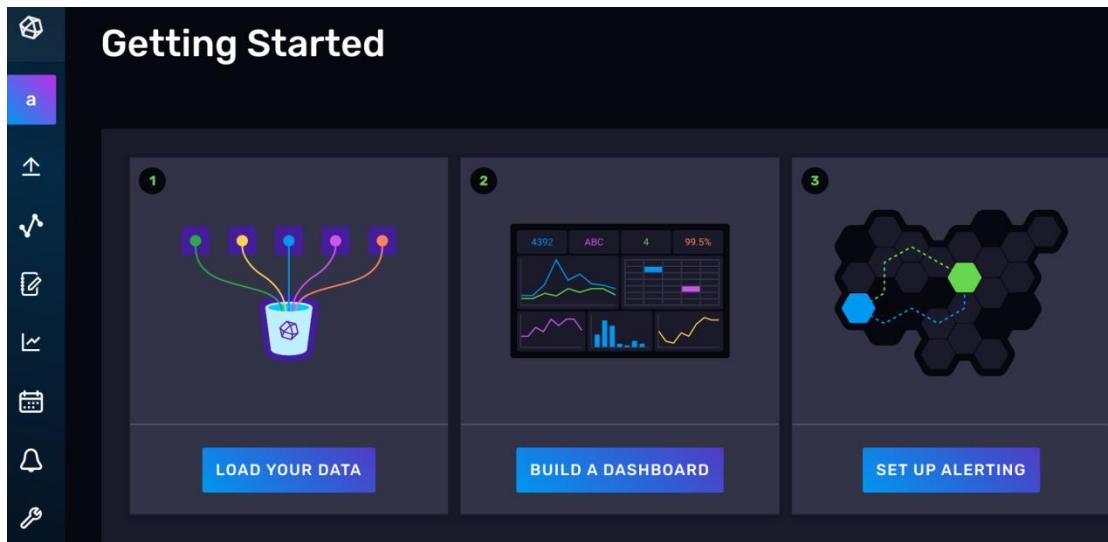


The screenshot shows the 'Setup Initial User' page. It has a dark background with white text and input fields. Red arrows point to specific fields with labels: '用户名' (Username) points to the 'Username' field containing 'tony'; '密码' (Password) points to the 'Password' field containing '.....'; '组织名称' (Initial Organization Name) points to the 'Initial Organization Name' field containing 'atguigu'; and '初始化的存储桶，相当于一个Database' (Initial Bucket Name) points to the 'Initial Bucket Name' field containing 'test_init'. A blue 'CONTINUE' button is at the bottom right.

填写完后点击 CONTINUE 按钮进入下一步。

2.2.2 配置完成

看到如图所示的页面，说明我们已经开始使用 tony 这一用户身份和 InfluxDB 交互了。



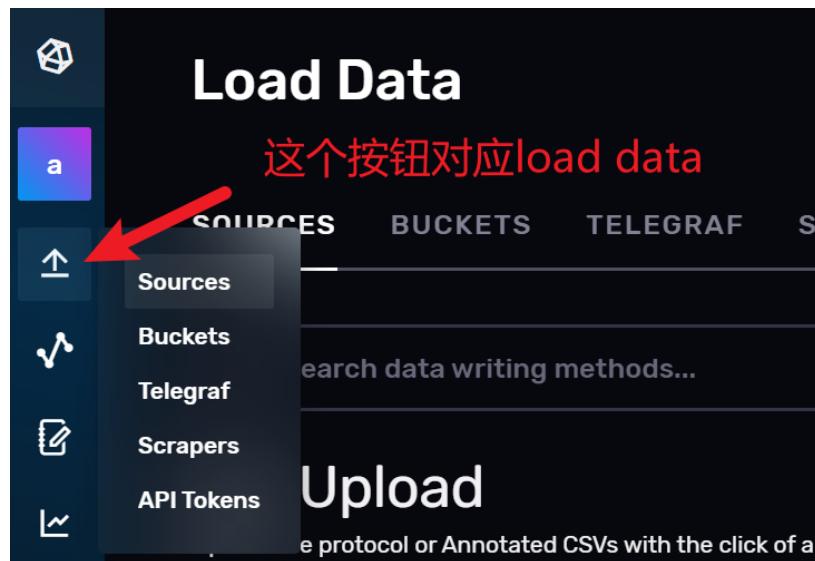
第3章 InfluxDB 入门（借助 Web UI）

借助 Web UI，我们可以更好地理解 InfluxDB 的功能划分。接下来，我们就从 Web UI 入手，先了解 InfluxDB 的基本功能。

3.1 数据源相关

3.1.1 Load Data（加载数据）

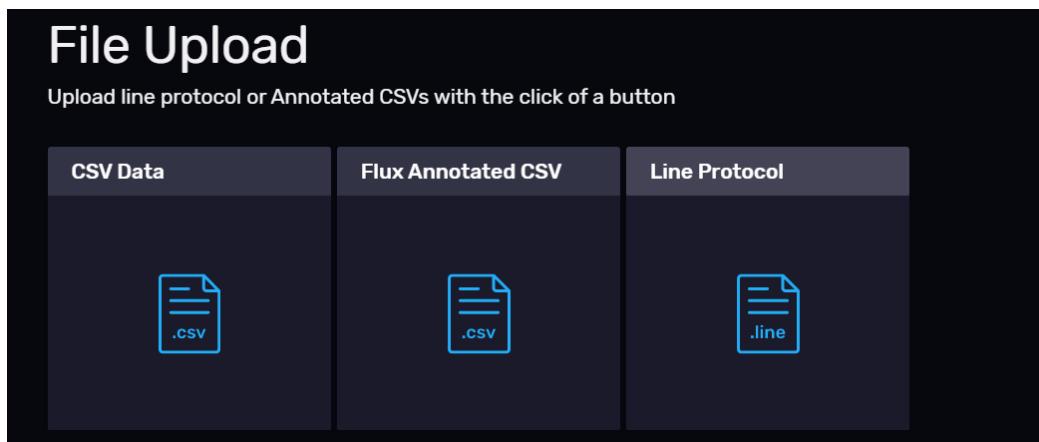
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网



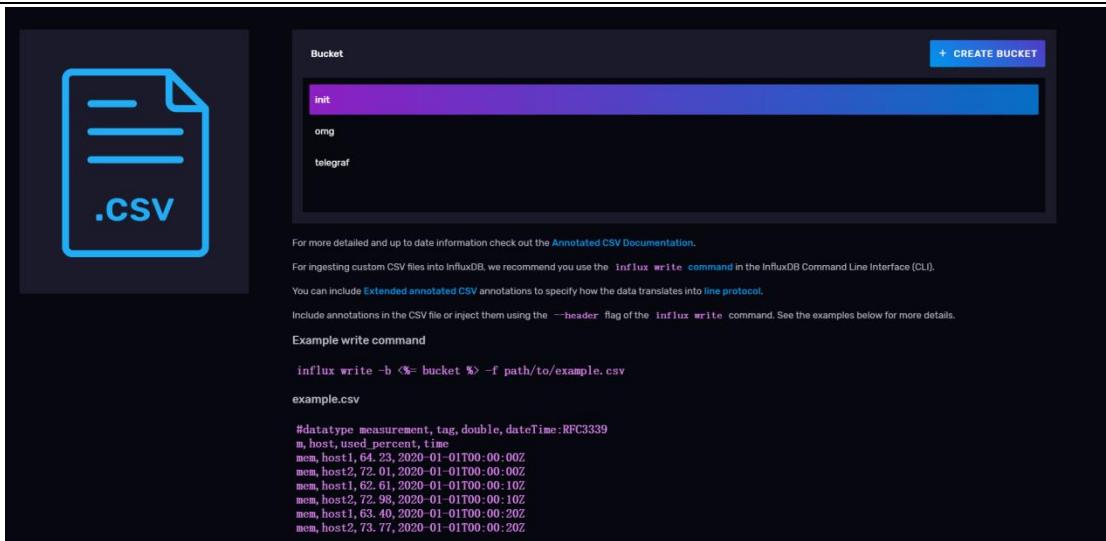
如图所示，页面上左侧的向上箭头，对应着 InfluxDB Web UI 的 Load Data（加载数据）页面。

3.1.1.1 上传数据文件

在 Web UI 上，你可以用文件的方式上传数据，前提是文件中的数据符合 InfluxDB 支持的类型，包括 CSV、带 Flux 注释的 CSV 和 InfluxDB 行协议。

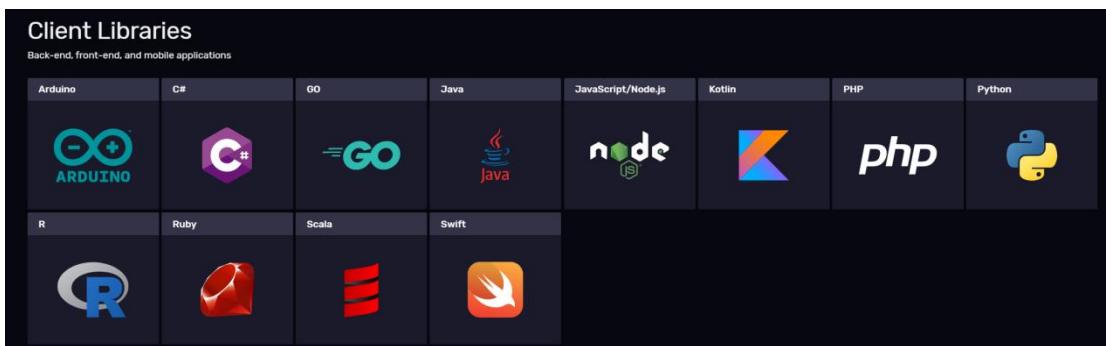


点击其中任意一个按钮，将进入数据的上传页面，页面中包含了详细的说明文档，包含你的数据应该符合什么格式，你要把数据放到哪个存储桶里，还包括用命令行来上传数据的命令模板。

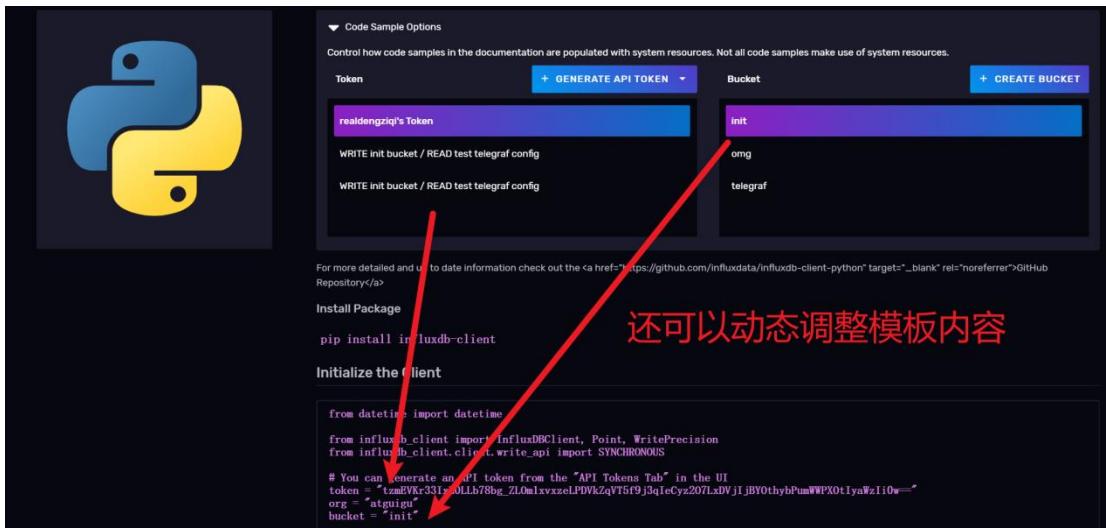


3.1.1.2 写入 InfluxDB 的代码模板

InfluxDB 提供了各种编程语言的连接库，你甚至可以在前端嵌入向 InfluxDB 写入数据的代码，因为 InfluxDB 向外提供了一套功能完整的 REST API。



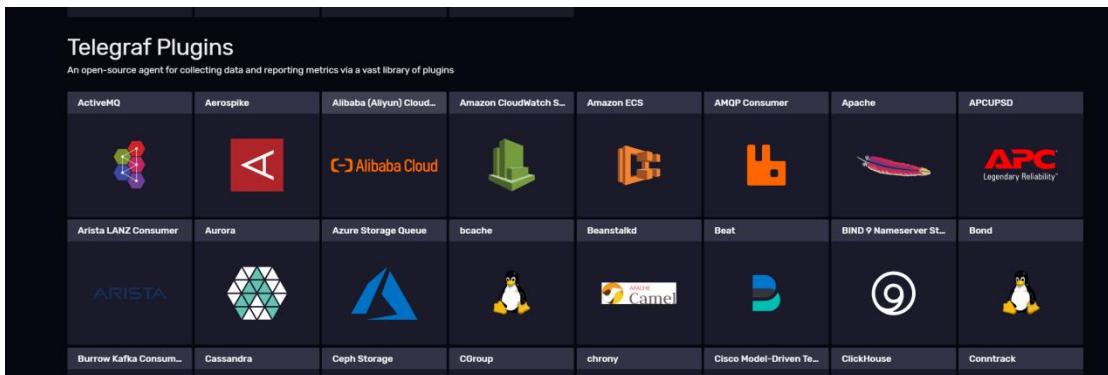
点击任何一个语言的 LOGO，你会看到使用这门语言，将数据写入到 InfluxDB 的代码模板。



建议从这里拷贝初始化客户端的代码。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

配置 Telegraf 的输入插件



Telegraf 是一个插件化的数据采集组件，在这里你可以找一下没有对应你的目标数据源的插件，点击它的 logo。可以看到这个插件配置的写法，但是关于这方面的内容，还是建议参考 Telegraf 的官方文档，那个更细更全一些。

Apache

Apache Input Plugin

The Apache plugin collects server performance information using the `mod_status` module of the Apache HTTP Server.

Typically, the `mod_status` module is configured to expose a page at the `/server-status?auto` location of the Apache server. The `ExtendedStatus` option...

Configuration:

```
# Read Apache status information (mod_status)
[[inputs.apache]]
  ## An array of URLs to gather from, must be directed at the machine
  ## readable version of the mod_status page including the auto query string.
  ## Default is "http://localhost/server-status?auto".
  urls = ["http://localhost/server-status?auto"]

  ## Credentials for basic HTTP authentication.
  # username = "myuser"
  # password = "mypassword"

  ## Maximum time to receive response.
  # response_timeout = "5s"

  ## Optional TLS Config
```

3.1.2 管理存储桶

你可以将 InfluxDB 中的 bucket 理解为普通关系型数据库中的 database。在 Load data 页面上，点击上访的 BUCKETS 选项卡，就可以进入 bucket 管理页面了。

Load Data

SOURCES BUCKETS TELEGRAF SCRAPERS API TOKENS

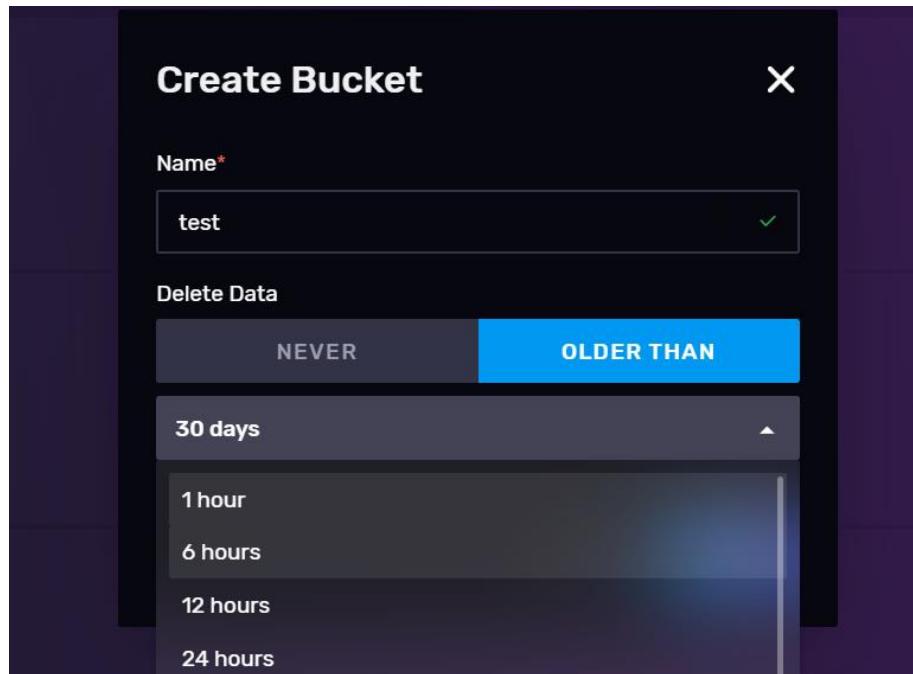
Filter buckets... Sort by Name (A → Z)

+ CREATE BUCKET

init Retention: Forever ID: 4a7b929a7cfb234b + Add a label + ADD DATA SETTINGS	omg Retention: Forever ID: d60e8d2ec8cc69ca + Add a label + ADD DATA SETTINGS	What is a Bucket? A bucket is a named location where time series data is stored. All buckets have a Retention Policy, a duration of time that each data point persists. Here's how to write data into your bucket.
--	---	---

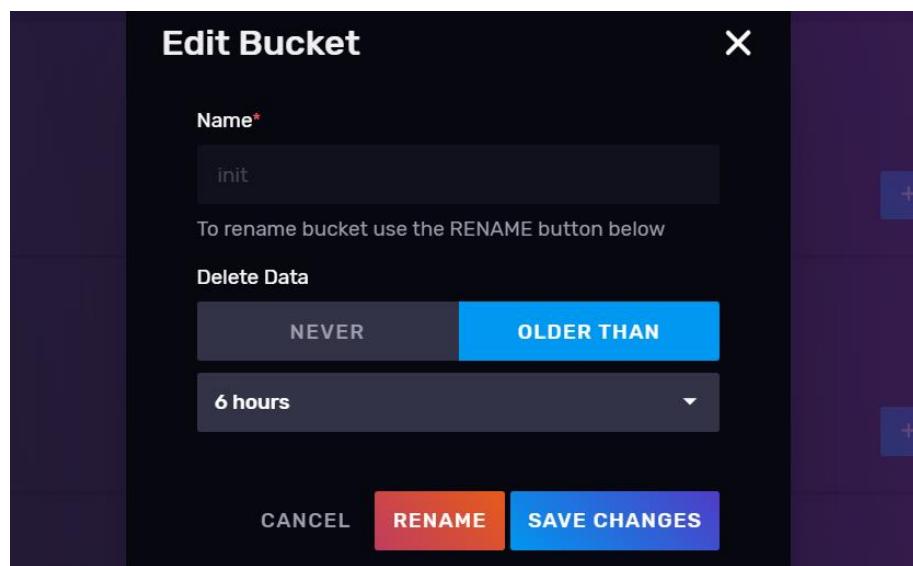
3.1.2.1 创建 Bucket

点击右上角的 CREATE BUCKET 按钮，会有一个创建存储桶的弹窗，这里你可以给 bucket 指定一个名称和数据的过期时间。比如，你设置过期时间为 6 小时，那 InfluxDB 就会自动把这个存储桶中距离当前时间超过 6 小时的数据删除。



3.1.2.2 调整 Bucket 的设置

存储桶的过期时间的名称都是可以修改的，点击任一 Bucket 信息卡的 SETTINGS 按钮会弹出一个调整设置的对话框。

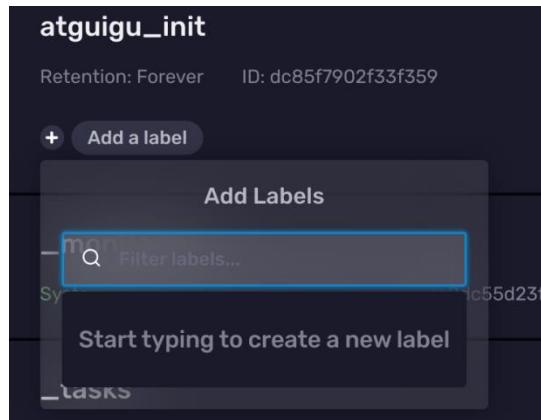


重命名是 InfluxDB 不建议的操作，因为大量的代码和 InfluxDB 定时任务都需要通过指定 Bucket 的名称来进行连接，贸然更改 Bucket 的名称可能导致这些程序无法正常工作。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

3.1.2.3 设置 Label

在每个 Bucket 信息卡的左下方都有一个 Add a label 按钮，点击这个按钮，你可以为 Bucket 添加一个标签。不过这个功能一般很少用



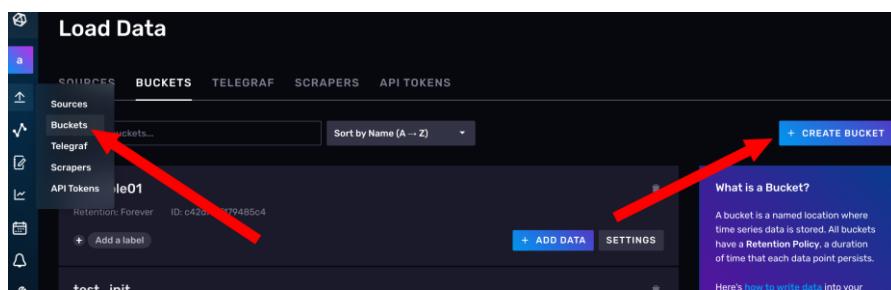
3.1.2.4 向 Bucket 添加数据

每个存储桶信息卡的右边都有一个添加数据按钮，点击这个按钮可以快速导入一些数据。这里还可以创建一个抓取任务（被抓取的数据在格式上必须符合 prometheus 数据格式）

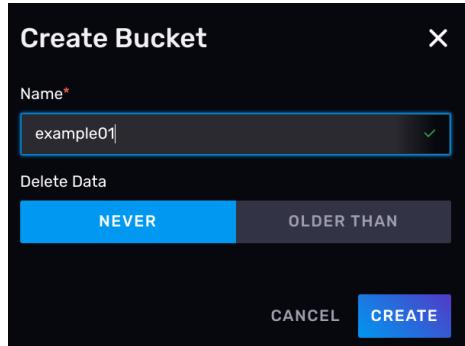
3.1.3 示例 1：创建 Bucket 并从文件导入数据

3.1.3.1 创建 Bucket

(1) 将鼠标悬停在 左侧的按钮上，点击 Buckets，进入 Bucket 的管理页面。



(2) 点击 CREATE BUCKET 按钮，指定一个名称，这里我们将其设为 example01，删除策略保留默认的 NEVER，表示永远不会删除数据



(3) 点击 CREATE 按钮，可以看到我们的 Buckets 已经创建成功了。

3.1.3.2 进入上传数据引导页面

在 Load Data 页面，点击 Line Protocol 进入 InfluxDB 行协议格式数据的上传引导页面。

3.1.3.3 录入数据

(1) 点击选择存储桶

(2) 选择 ENTER MANUALLY, 手动输入数据

(3) 将数据粘到输入框

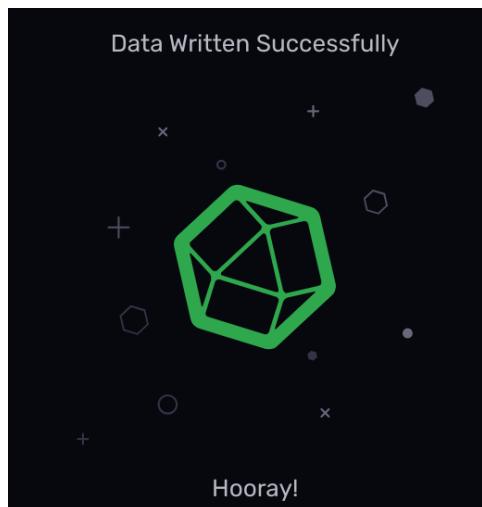
(4) 在右侧指明时间精度, 包括纳秒、微秒、毫秒和秒

数据如下:

```
people, name=tony age=12
people, name=xiaohong age=13
people, name=xiaobai age=14
people, name=xiaohei age=15
people, name=xiaohua age=12
```

当前我们写的数据格式叫做 InfluxDB 行协议。你可以查看附录 2 来了解这一数据格式的知识。

最后点击 WRITE DATA, 将数据写到 InfluxDB。如果出现 Data Written Successfully, 那么说明数据写入成功。

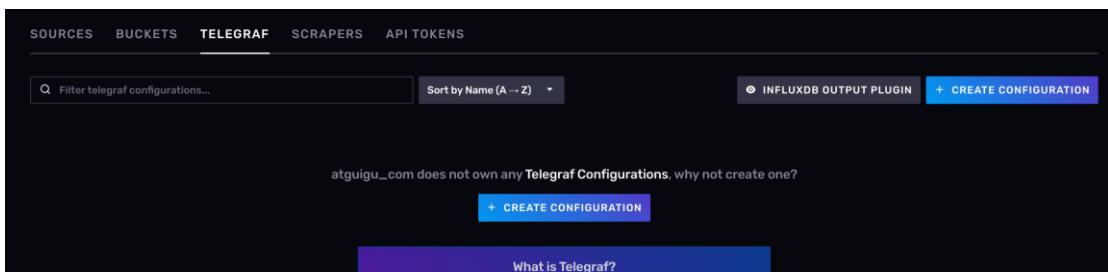


3.1.3.4 小结

InfluxDB 是一个无模式的数据库, 也就是除了在输入数据之前需要显示创建存储桶(数据库), 你不需要手动创建 measurement 或者指定各个 field 都是什么类型, 你甚至可以前后在同一个 measurement 下插入 filed 不同的数据。

3.1.4 管理 Telegraf 数据源

点击 Load Data 页面的 TELEGRAF 选项卡, 可以快速生成一些 Telegraf 配置文件。并向外暴露一个端口, 允许 telegraf 远程使用 InfluxDB 中生成的配置。

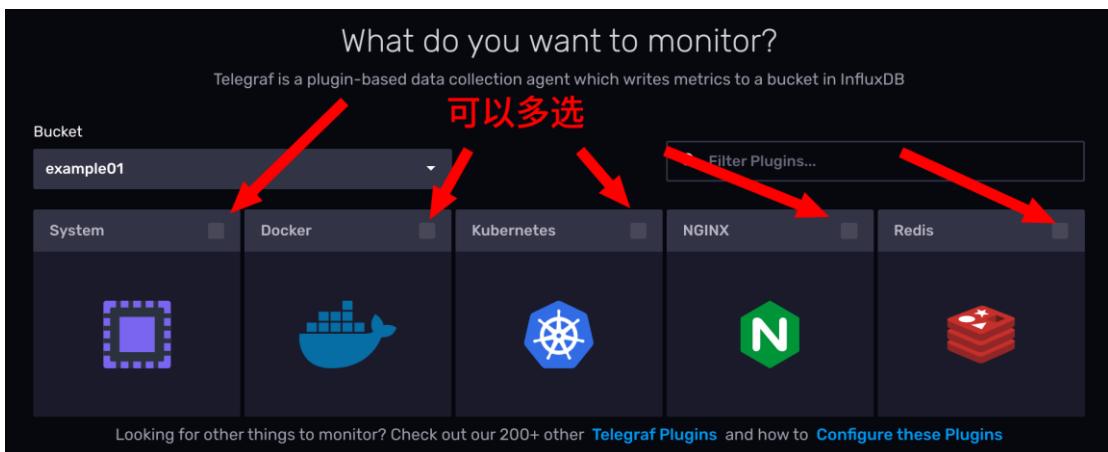


3.1.4.1 什么是 Telegraf

Telegraf 是 InfluxDB 生态中的一个数据采集组件，它可以将各种时序数据自动采集到 InfluxDB。现在，Telegraf 不仅仅是 InfluxDB 的数据采集组件了，很多时序数据库都支持与 Telegraf 进行协作，不少类似的时序数据收集组件选择在 Telegraf 的基础上二次开发。所以，我们将 Telegraf 录成了一门专门的课，大家可以到 B 站上找尚硅谷的 Telegraf 课程，将课程看到示例 3，就可以理解本课程中使用到的关于 Telegraf 的知识点了。

3.1.4.2 创建 Telegraf 配置文件

InfluxDB 的 Web UI 为我们提供了几种最常用的 telegraf 配置模板，包括监控主机指标、云原生容器状态指标，nginx 和 redis 等。



通过页面，你可以勾选几种监控目标，然后一步步操作去创建一个 Telegraf 的配置文件出来。

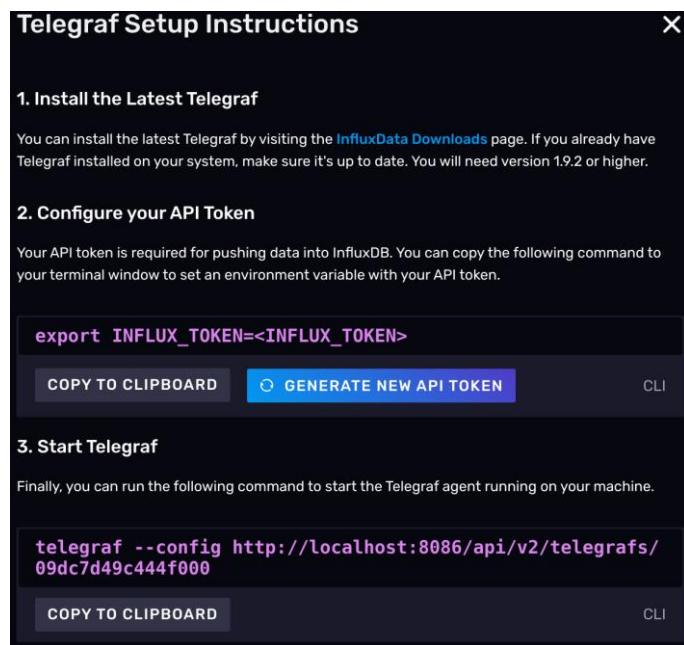
3.1.4.3 管理 Telegraf 配置文件接口

完成 Telegraf 的配置后，页面上会多出一个关于 telegraf 实例的信息卡。如图所示：



点击蓝色的 Setup Instructions。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网



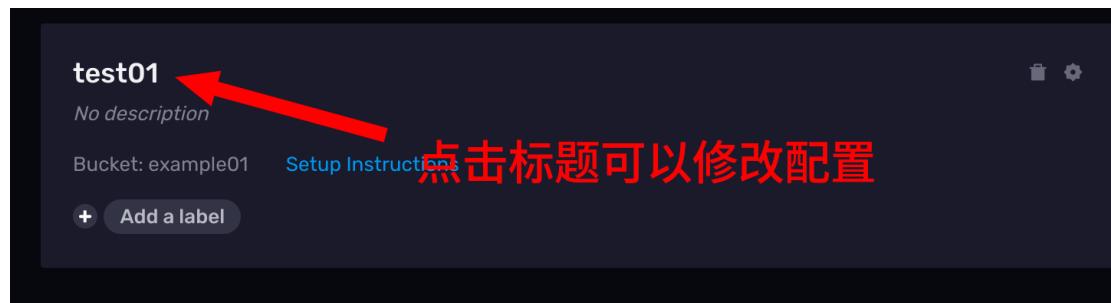
会弹出一个对话框，引导你完成 telegraf 的配置。可以看到第三步的命令。

```
telegraf --config  
http://localhost:8086/api/v2/telegrafs/09dc7d49c444f000
```

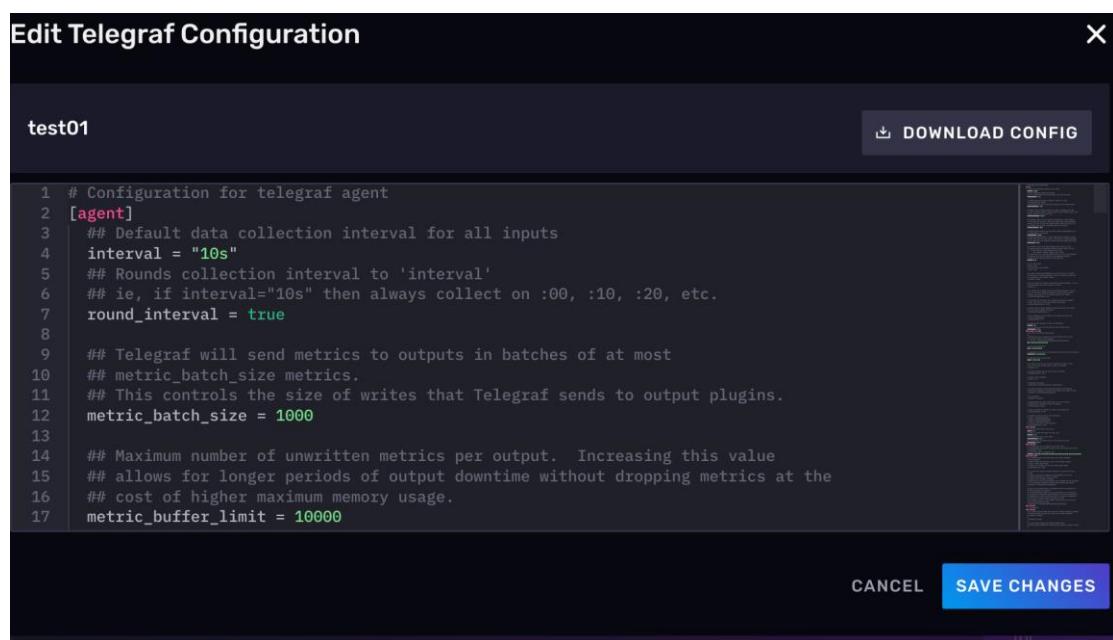
这个命令中有一个 URL，其实意思也就是 InfluxDB 向外提供了一个 API，通过这个 API 你可以访问到刚才生成的配置文件。

3.1.4.4 修改 Telegraf 配置

已经生成的配置文件如何去修改呢？你可以点击卡片的标题。



这个时候，会弹出一个配置文件的编辑页面，不过这个时候没有交互式的选项了，你需要自己直接面对配置文件。



修改完配置文件后，记得点击右方的 SAVE CHANGES 保存修改。

3.1.5 示例 2：使用 Telegraf 将数据收集到 InfluxDB

在本示例中，我们会使用 Telegraf 这个工具将一台机器上的 CPU 使用情况转变成时序数据，写到我们的 InfluxDB 中。

3.1.5.1 下载 Telegraf

可以使用下面的命令下载 telegraf，也可以在本课程的配套资料中获取（关注尚硅谷微信公众号，回复“大数据”）。

```
 wget https://dl.influxdata.com/telegraf/releases/telegraf-1.23.4_linux_amd64.tar.gz
```

3.1.5.2 解压压缩包

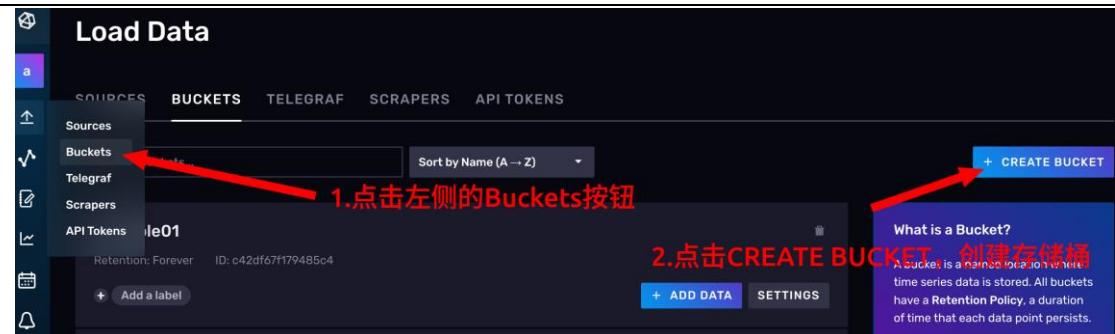
将 telegraf 解压到目标路径。

```
 tar -zxvf telegraf-1.23.4_linux_amd64.tar.gz -C /opt/module/
```

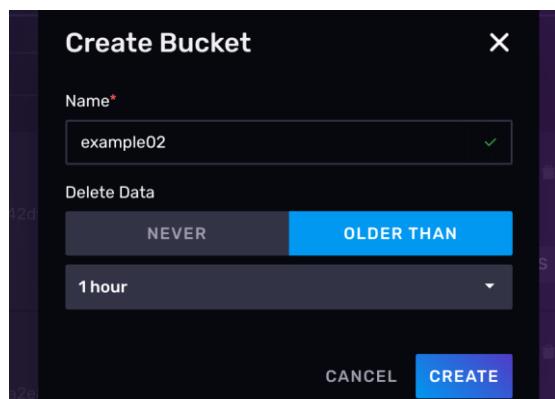
3.1.5.3 创建一个新的 Bucket

回到 Web UI 界面

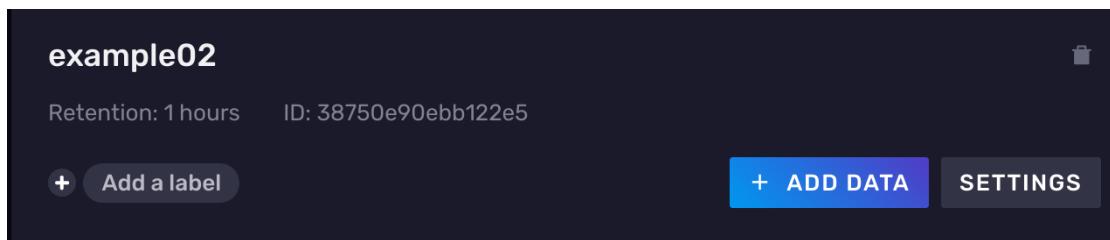
- (1) 点击左侧工具栏中的 Buckets 按钮
- (2) 点击右侧蓝色的 CREATE BUCKET 按钮



(3) 创建一个名为 example02 的 buckets，因为是演示，所以这里将过期时间为 1 小时。设置好后点击 CREATE



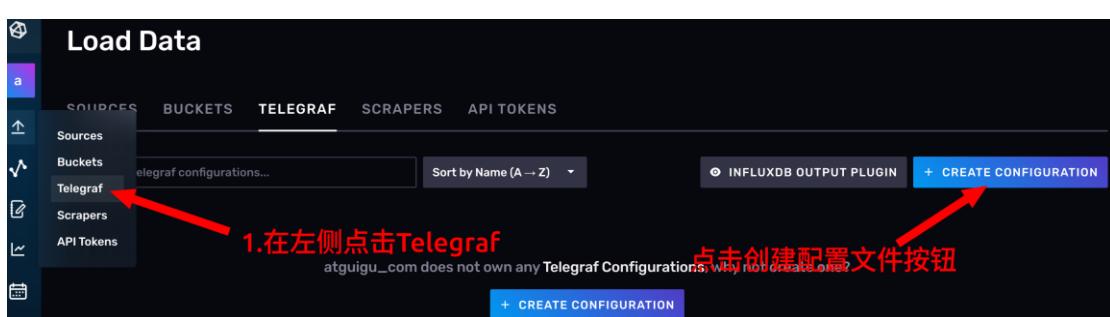
(4) 如果出现相应的 example02 的卡片，说明存储桶已经创建成功。



3.1.5.4 在 Web UI 上创建 telegraf 配置文件

(1) 在左侧的工具栏上点击 Telegraf 按钮。

(2) 点击右侧蓝色的 CREATE CONFIGURATION 创建 telegraf 配置文件



(3) 在 Bucket 栏选择 example02，表示让 telegraf 将抓取到的数据写到 example02 存储桶中，下面的选项卡勾选 System。点击 CONTINUE。



(4) 点击 CONTINUE 按钮后，会进入一个配置插件的页面。你可以自己决定是否启用这些插件。这里需要给生成的 Telegraf 配置起一个名字，方便管理。



(5) 点击 CREATE AND VERIFY 按钮，这个时候其实 Telegraf 的配置就已经创建好了，你会进入一个 Telegraf 的配置引导界面，如图所示：

Create a Telegraf Configuration

TEST YOUR CONFIGURATION

Start Telegraf and ensure data is being written to InfluxDB

1. Install the Latest Telegraf

You can install the latest Telegraf by visiting the [InfluxData Downloads](#) page. If you already have Telegraf installed on your system, make sure it's up to date. You will need version 1.9.2 or higher.

2. Configure your API Token

Your API token is required for pushing data into InfluxDB. You can copy the following command to your terminal window to set an environment variable with your API token.

```
export INFLUX_TOKEN=-RczDbY-fQtx0frfK_UbMfQtwydjPITY2nbg19z8JJdnC4GVt58CSUWE0I5vpXfx0s7pbI8ic_c0wY-yJA_e
tQ==
```

COPY TO CLIPBOARD **GENERATE NEW API TOKEN** **CLI**

3. Start Telegraf

Finally, you can run the following command to start the Telegraf agent running on your machine.

```
telegraf --config http://localhost:8086/api/v2/telegrafs/09dcf4afcf90000
```

COPY TO CLIPBOARD **CLI**

LISTEN FOR DATA

FINISH

3.1.5.5 声明 Telegraf 环境变量

按照 Web UI 上的建议，首先，你要在部署 Telegraf 的主机上声明一个环境变量叫 INFLUX_TOKEN，它是用来赋予 Telegraf 向 InfluxDB 写数据权限的。这里我们就不配环境变量了，请在单一的 shell 会话下完成后面的操作。

所以到你下载好 Telegraf 的机器上，执行下面的命令。（注意！TOKEN 是随机生成的，请按照自己的情况修改命令）

```
export INFLUX_TOKEN=v4TsUzzWtqgot18kt_adS1r-
7PTsMIQkbhEQ7oqLCP2TQ5Q-PcUP6RMyTHLy4IryPl_2riamNarsNqDc_S_eA==
```

3.1.5.6 启动 Telegraf

首先 cd 到我们解压的 telegraf 目录。

```
cd /opt/module/telegraf-1.23.4
```

```
— 22 8月 - 15:39 /opt/module/telegraf-1.23.4
@atguigu ll
总用量 12K
drwxr-xr-x 4 atguigu atguigu 4.0K 8月 17 03:42 etc/
drwxr-xr-x 4 atguigu atguigu 4.0K 8月 17 03:42 usr/
drwxr-xr-x 3 atguigu atguigu 4.0K 8月 17 03:42 var/
```

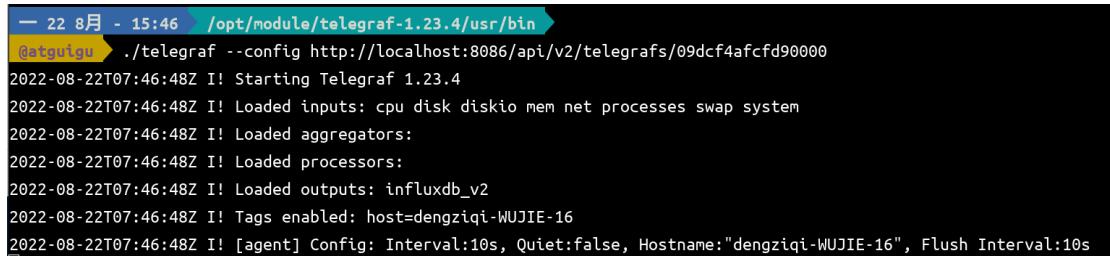
telegraf 的可执行文件在 ./usr/bin 目录下。cd 过去。

```
cd ./usr/bin
```

从 Web UI 中复制运行 telegraf 的命令，修改 host 然后执行，老师的 telegraf 和 InfluxDB 在同一台机器上，所以可以使用 localhost。最终命令如下。

```
telegraf --config
http://localhost:8086/api/v2/telegrafs/09dcf4afcf90000telegraf
```

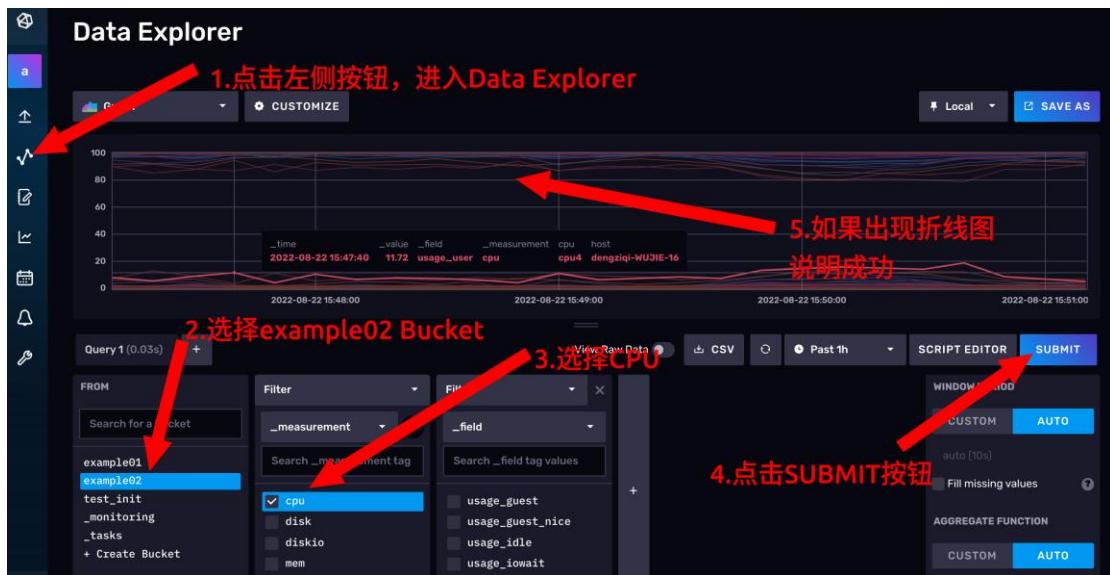
运行效果如下图所示。



```
— 22 8月 - 15:46 /opt/module/telegraf-1.23.4/usr/bin
@atguigu ./telegraf --config http://localhost:8086/api/v2/telegrafs/09dcf4afcf90000
2022-08-22T07:46:48Z I! Starting Telegraf 1.23.4
2022-08-22T07:46:48Z I! Loaded inputs: cpu disk diskio mem net processes swap system
2022-08-22T07:46:48Z I! Loaded aggregators:
2022-08-22T07:46:48Z I! Loaded processors:
2022-08-22T07:46:48Z I! Loaded outputs: influxdb_v2
2022-08-22T07:46:48Z I! Tags enabled: host=dengziqi-WUJIE-16
2022-08-22T07:46:48Z I! [agent] Config: Interval:10s, Quiet:false, Hostname:"dengziqi-WUJIE-16", Flush Interval:10s
```

3.1.5.7 验证数据采集结果

- (1) 点击左侧  按钮进入 Data Explorer 页面。
- (2) 在左下角第一个选项卡选择 example02，表示要从 example02 这个存储桶中查数据。
- (3) 点击好第一个选项卡后，会自动弹出第二个选项卡，勾选 cpu。
- (4) 点击右上方的 SUBMIT 按钮。
- (5) 如果出现折线图，说明我们成功地使用 Telegraf 把数据导进来了。



3.1.5.8 编写启停脚本

后面我们很多时候都要使用 telegraf 抓取的主机监控数据来进行查询演示。为了方便启停，我们编写一个 shell 脚本来管理 telegraf 任务。

- (1) 首先 cd 到~/bin 路径下，如果~路径下没有 bin，就创建 bin 这个目录。通常，

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

~/bin 是 PATH 环境变量包含的一个目录。

```
cd ~  
mkdir bin  
cd ~/bin
```

(2) 到~/bin 路径下创建一个文件 host_tel.sh

```
vim host_tel.sh
```

(3) 键入如下内容

```
#!/bin/bash

is_exist(){
    pid=`ps -ef | grep telegraf | grep -v grep | awk '{print $2}'`  

    # 如果不存在返回 1, 存在返回 0  

    if [ -z "${pid}" ]; then  

        return 1  

    else  

        return 0  

    fi
}

stop(){
    is_exist  

    if [ $? -eq "0" ]; then  

        kill ${pid}  

        if [ $? -eq "0" ]; then  

            echo "进程号:${pid},弄死你"  

        else  

            echo "进程号:${pid},没弄死"  

        fi
    else  

        echo "本来没有 telegraf 进程"  

    fi
}

start(){
    is_exist  

    if [ $? -eq "0" ]; then  

        echo "跑着呢, pid 是${pid}"  

    else  

        export INFLUX_TOKEN=v4TsUzzWtqgot18kt_adS1r-  

        7PTsMIQkbnhE7oqLCP2TQ5Q-PcUP6RMyTHLy4IryP1_2rIamNarsNqDc_S_eA==  

        /opt/module/telegraf-1.23.4/usr/bin/telegraf --config  

        http://localhost:8086/api/v2/telegrafs/09dcf4afcf90000  

    fi
}

status(){
    is_exist  

    if [ $? -eq "0" ]; then  

        echo "telegraf 跑着呢"  

    else  

        echo "telegraf 没有跑"  

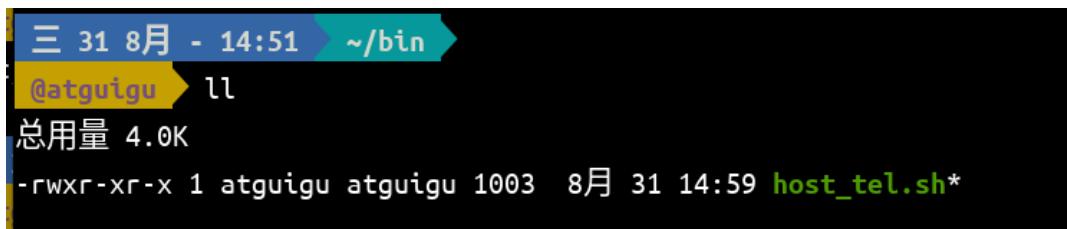
    fi
}
```

```
usage() {
    echo "哦! 请你 start 或 stop 或 status"
    exit 1
}

case "$1" in
    "start")
        start
        ;;
    "stop")
        stop
        ;;
    "status")
        status
        ;;
    *)
        usage
        ;;
esac 最后
```

(4) 最后给这个脚本加上一个执行权限, 你可以执行下面的代码。

```
chmod 755 ./host_tel.sh
```



A terminal window showing the file permissions for `host_tel.sh`. The command `ll` is run, displaying the following output:

```
三 31 8月 - 14:51 ~ /bin
@atguigu > ll
总用量 4.0K
-rwxr-xr-x 1 atguigu atguigu 1003 8月 31 14:59 host_tel.sh*
```

3.1.5.9 小结

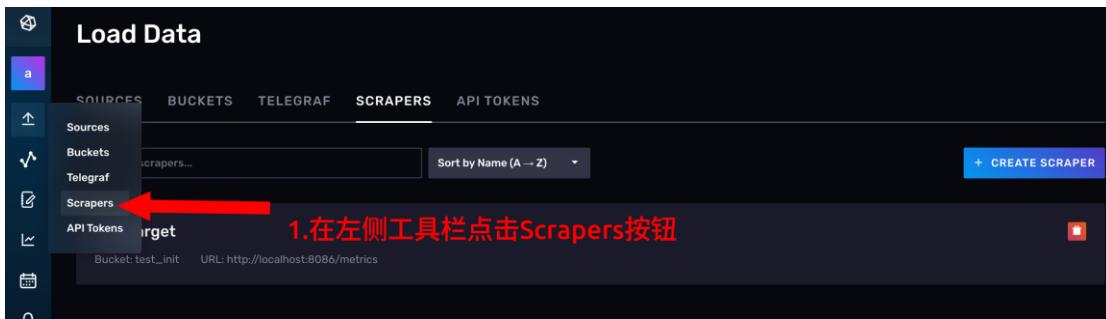
最终需要注意。InfluxDB 只是帮你管理了一下 Telegraf 的配置文件。InfluxDB 并不能管理 Telegraf 的启停和运行状态。如何运行 Telegraf 还是需要开发者手动或者编写脚本来维护的。

3.1.6 管理抓取任务

3.1.6.1 什么是抓取任务

抓取任务就是你给定一个 URL, InfluxDB 每隔一段时间去访问这个链接, 把访问到的数据入库。

在 InfluxDB 1.x 的时候, 类似的任务只能由 Telegraf 来实现。在 InfluxDB 2.x 中, 内置了抓取功能 (但是定制性上不如 Telegraf, 比如轮询间隔只能是 10 秒)



另外，目标 URL 暴露出来的数据格式必须得是 Prometheus 数据格式。关于 Prometheus 数据格式的详细介绍同学们可以参考本文档的[附录 3](#)

3.1.6.2 InfluxDB 自身暴露的监控接口

你可以访问 <http://localhost:8086/metrics> 来查看 InfluxDB 暴露出来的性能数据。这里有，InfluxDB 的 GC 情况

```
localhost:8086/metrics

# HELP boltdb_reads_total Total number of boltdb reads
# TYPE boltdb_reads_total counter
boltdb_reads_total 6018
# HELP boltdb_writes_total Total number of boltdb writes
# TYPE boltdb_writes_total counter
boltdb_writes_total 37
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.7819e-05
go_gc_duration_seconds{quantile="0.25"} 7.2565e-05
go_gc_duration_seconds{quantile="0.5"} 0.000115633
go_gc_duration_seconds{quantile="0.75"} 0.00016729
go_gc_duration_seconds{quantile="1"} 0.005084312
go_gc_duration_seconds_sum 0.015866833
go_gc_duration_seconds_count 88
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 1222
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.18.5"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
```

以及各个 API 的使用情况，如图所示，说的是各个 API 被谁请求过多少次。

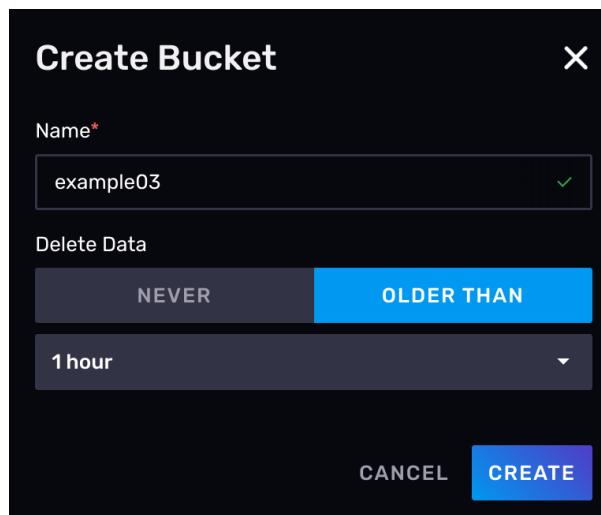
```
http_api_requests_total{handler="platform",method="GET",path="/6d375d8acf.png",response_code="200",status="2XX",user_agent="Edg"} 1
http_api_requests_total{handler="platform",method="GET",path="/9763d95516.png",response_code="200",status="2XX",user_agent="Edg"} 1
http_api_requests_total{handler="platform",method="GET",path="/fallback_path",response_code="200",status="2XX",user_agent="Chrome"} 1
http_api_requests_total{handler="platform",method="GET",path="/fallback_path",response_code="200",status="2XX",user_agent="Edg"} 1
http_api_requests_total{handler="platform",method="GET",path="/file_name.js",response_code="200",status="2XX",user_agent="Edg"} 23
http_api_requests_total{handler="platform",method="GET",path="/file_name.svg",response_code="200",status="2XX",user_agent="Edg"} 161
http_api_requests_total{handler="platform",method="GET",path="/file_name.wasm",response_code="200",status="2XX",user_agent="Edg"} 1
http_api_requests_total{handler="platform",method="GET",path="/file_name.woff2",response_code="200",status="2XX",user_agent="Edg"} 7
http_api_requests_total{handler="platform",method="GET",path="/api/v2/authorizations",response_code="200",status="2XX",user_agent="Edg"} 3
http_api_requests_total{handler="platform",method="GET",path="/api/v2/buckets",response_code="200",status="2XX",user_agent="Edg"} 10
http_api_requests_total{handler="platform",method="GET",path="/api/v2/dashboards",response_code="200",status="2XX",user_agent="Chrome"} 13
http_api_requests_total{handler="platform",method="GET",path="/api/v2/dashboard",response_code="200",status="2XX",user_agent="Edg"} 1
http_api_requests_total{handler="platform",method="GET",path="/api/v2/flags",response_code="200",status="2XX",user_agent="Chrome"} 9
http_api_requests_total{handler="platform",method="GET",path="/api/v2/flags",response_code="200",status="2XX",user_agent="Edg"} 2
http_api_requests_total{handler="platform",method="GET",path="/api/v2/labels",response_code="200",status="2XX",user_agent="Chrome"} 13
http_api_requests_total{handler="platform",method="GET",path="/api/v2/labels",response_code="200",status="2XX",user_agent="Edg"} 2
http_api_requests_total{handler="platform",method="GET",path="/api/v2/me",response_code="200",status="2XX",user_agent="Chrome"} 19
http_api_requests_total{handler="platform",method="GET",path="/api/v2/me",response_code="200",status="2XX",user_agent="Edg"} 124
http_api_requests_total{handler="platform",method="GET",path="/api/v2/orgs",response_code="200",status="2XX",user_agent="Chrome"} 9
http_api_requests_total{handler="platform",method="GET",path="/api/v2/orgs",response_code="200",status="2XX",user_agent="Edg"} 2
http_api_requests_total{handler="platform",method="GET",path="/api/v2/scrapers",response_code="200",status="2XX",user_agent="Edg"} 3
http_api_requests_total{handler="platform",method="GET",path="/api/v2/setup",response_code="200",status="2XX",user_agent="Chrome"} 10
http_api_requests_total{handler="platform",method="GET",path="/api/v2/setup",response_code="200",status="2XX",user_agent="Edg"} 5
http_api_requests_total{handler="platform",method="GET",path="/api/v2/tasks",response_code="200",status="2XX",user_agent="Chrome"} 1
http_api_requests_total{handler="platform",method="GET",path="/api/v2/telegrafs",response_code="200",status="2XX",user_agent="Edg"} 2
```

3.1.7 示例 3：让 InfluxDB 主动拉取数据

3.1.7.1 创建一个存储桶

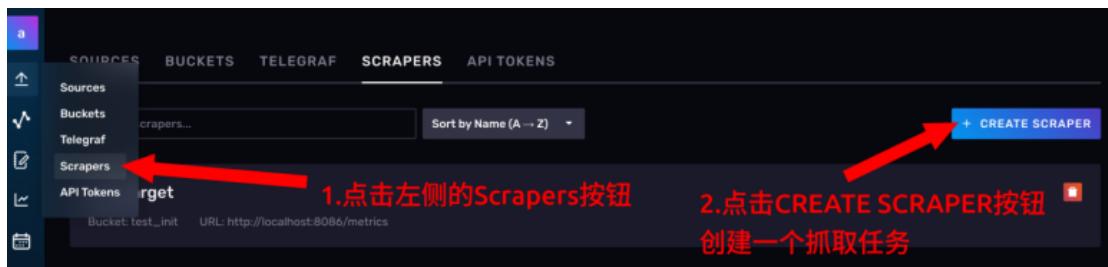
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

如图所示，我们创建了一个名为 example03 的存储桶。数据的过期时间为 1 小时。

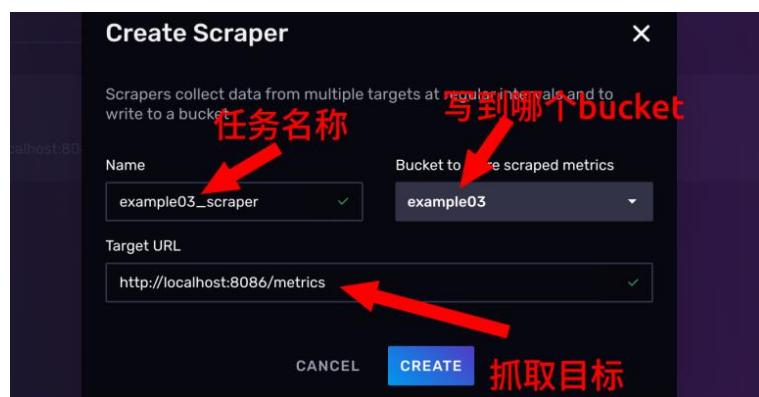


3.1.7.2 创建抓取任务

- (1) 进入抓取任务的管理页面
- (2) 点击 CREATE SCRAPER 按钮，创建抓取任务。



- (3) 在对话框上，给抓取任务起一个名字，此处命名为 example03_scraping
- (4) 右方的下拉框上，选择我们刚才创建的存储桶，example03。
- (5) 最下方设置一下目标路径，最后点击 CREATE



- (6) 如果页面上出现新的卡片，说明配置成功。接下来去看一下数据有没有进来。

The screenshot shows the InfluxDB interface for managing scrapers. At the top, there's a search bar labeled 'Filter scrapers...' and a dropdown menu 'Sort by Name (A → Z)'. On the right, there's a blue button '+ CREATE SCRAPER'. Below this, a card for 'example03_scraping' is displayed, showing the bucket 'example03' and URL 'http://localhost:8086/metrics'. There's also a small trash can icon.

3.1.7.3 验证抓取结果

- (1) 点击左侧的按钮，打开 Data Explorer
- (2) 在左下角第一个卡片选择要从哪个存储桶抽取数据，本例对应的是 example03
- (3) 第一个卡片选择好后，会自动弹出第二个卡片，你可以选择任意一个指标名称。
- (4) 点击右侧的 SUBMIT 按钮，提交查询。
- (5) 如果折线图成功加载，说明有数据了，抓取成功！

The screenshot shows the InfluxDB Data Explorer. It has several sections: a left sidebar with icons; a top navigation bar with 'Data Explorer', 'Graph', 'CUSTOMIZE', 'Local', and 'SAVE AS'; a main area with a line chart and a table of data; and a bottom section for querying. Red arrows point to specific steps: 1. Open Data Explorer (left sidebar), 2. Select storage bucket (FROM dropdown), 3. Select a measurement name (measurement dropdown), 4. Click submit (SUBMIT button), and 5. If data is visible, it's successful! (text overlay).

_time	_value	_field	_measurement
2022-08-23 11:19:20	52	count	go_gc_duration_seconds
2022-08-23 11:19:20	5.027μs	sum	go_gc_duration_seconds
2022-08-23 11:19:20	1.494μm	avg	go_gc_duration_seconds
2022-08-23 11:19:20	136.7μ	0.75	go_gc_duration_seconds
2022-08-23 11:19:20	95.01μ	0.5	go_gc_duration_seconds
2022-08-23 11:19:20	30.99μ	0.25	go_gc_duration_seconds
2022-08-23 11:19:20	21.16μ	0	go_gc_duration_seconds

3.1.7.4 补充

1) InfluxDB 的监控数据默认会被抓取到初始化的存储桶中

抓取任务管理面板上，我们发现自己还没创建什么东西呢，就有一个抓取任务。

The screenshot shows the InfluxDB Task Management interface. It lists a single task named 'new target' under the heading 'Bucket: test_init URL: http://localhost:8086/metrics'. A red arrow points to this entry with the text '发现一上来就有一个抓取任务' (Discover that there is one scheduled task from the start).

这个抓取任务是 InfluxDB 自动为我们创建的，它会把我们刚才访问 /metrics 拿到的数据写到 test_init 这个存储桶中去，而 test_init 这个存储桶是我们首次登录的时候为了初始化而创建的。所以大家要知道 test_init 中的一些监控数据是怎么产生的。

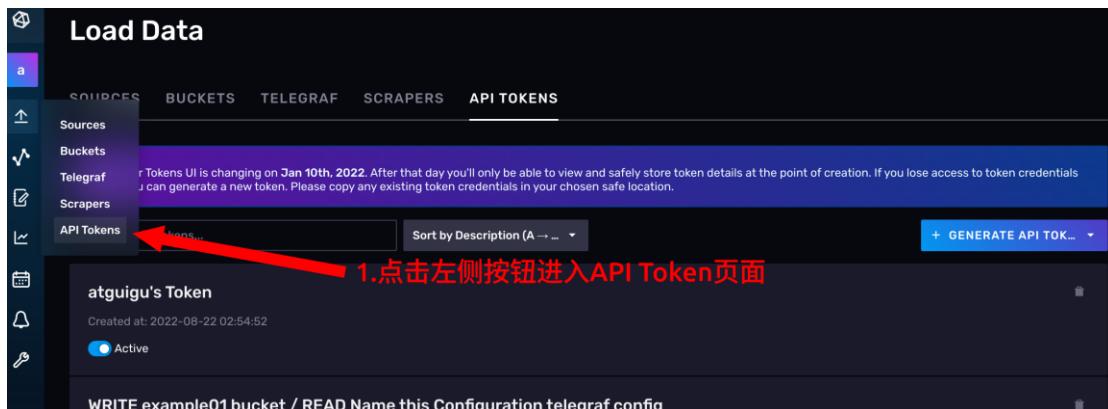
2) InfluxDB 的抓取任务都是 10 秒一次，无法自定义设置

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

至少截至目前（2.4 版本），用户无法去自定义抓取间隔。InfluxDB 会每隔 10 秒一次去抓取数据，这一点需要注意。

3.1.8 管理 API Token

点击左侧的 API Tokens 按钮，进入 API Token 的管理页面。



The screenshot shows the 'Load Data' interface with the 'API TOKENS' tab selected. A red arrow points to the 'API Tokens' button in the sidebar. The main area displays a single token entry:

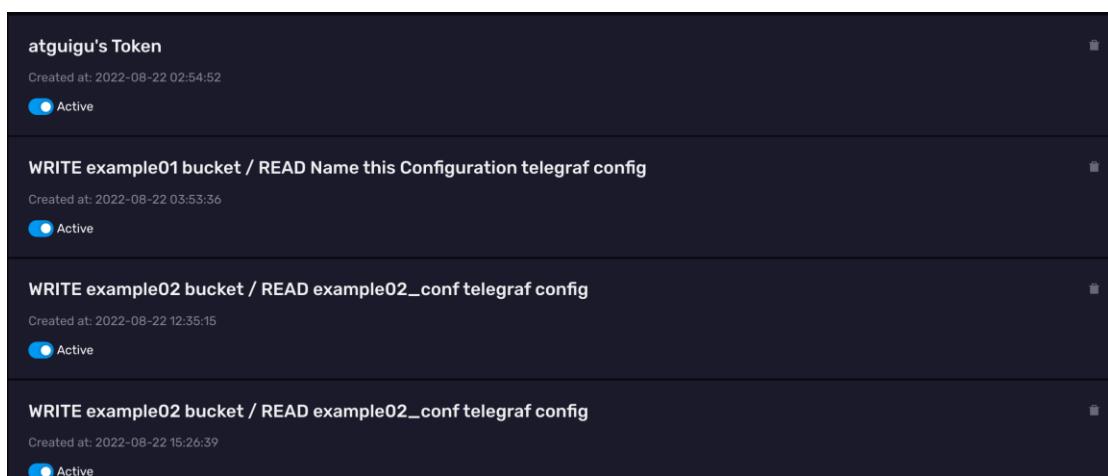
atguigu's Token
Created at: 2022-08-22 02:54:52
<input checked="" type="radio"/> Active
WRITE example01 bucket / READ Name this Configuration telegraf config

3.1.8.1 API Token 是干什么用的

简单来说，influxdb 会向外暴露一套 HTTP API。我们后面要学的命令行工具什么的，其实都是封装的对 influxdb 的 http 请求。所以，在 InfluxDB 中，对权限的管理主要就体现在 API 的 Tokens 上。客户端会将 token 放到 http 的请求头上，influxdb 服务端就根据客户端发来的请求头部的 token，来判断你能不能对某个存储桶读写，能不能删除存储桶，创建仪表盘等。

3.1.8.2 查看 API Token 权限

截至目前，我们还没有自己手动创建过 API Token。但是可以看到页面上已经有一些 Token 了，这些 Token 是由我们之前示例里面的操作自动生成的。

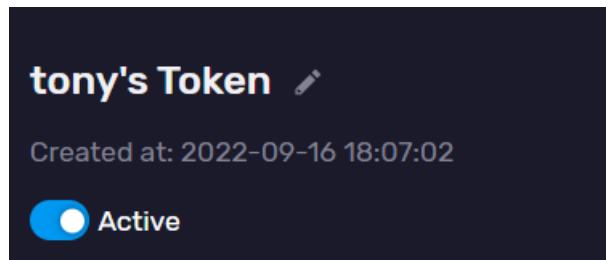


The screenshot shows the 'Load Data' interface with the 'API TOKENS' tab selected. It lists three tokens:

atguigu's Token
Created at: 2022-08-22 02:54:52
<input checked="" type="radio"/> Active
WRITE example01 bucket / READ Name this Configuration telegraf config
atguigu's Token
Created at: 2022-08-22 03:53:36
<input checked="" type="radio"/> Active
WRITE example02 bucket / READ example02_conf telegraf config
atguigu's Token
Created at: 2022-08-22 12:35:15
<input checked="" type="radio"/> Active
WRITE example02 bucket / READ example02_conf telegraf config
atguigu's Token
Created at: 2022-08-22 15:26:39
<input checked="" type="radio"/> Active
WRITE example02 bucket / READ example02_conf telegraf config

3.1.8.3 了解 tony's Token

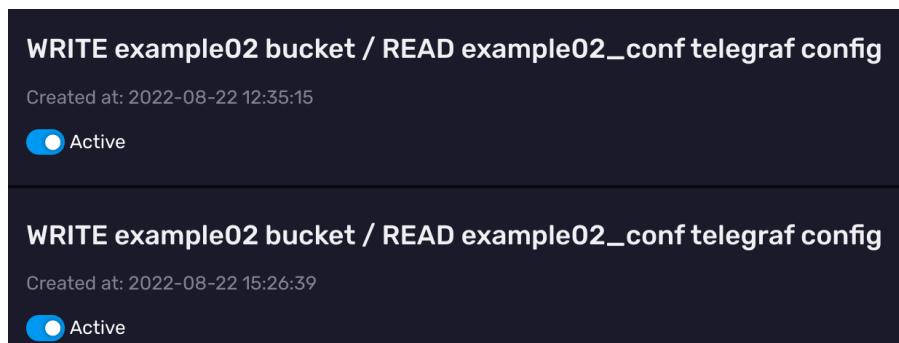
现在，我们围绕着 InfluxDB 中已有的 Token 来学习相关的知识，我们的 InfluxDB 上现在只有初始化时创建的 tony 账户，在 Token 列表中，我们可以看到有一个名为 tony's Token 的 token。



1) 修改 token 的名称

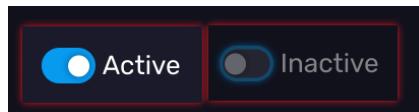
点击 token 右边的 符号，可以修改 token 名称。

- 没有客户端会用 token 的名称来调用 token，所以修改 token 名称不会影响已经部署的应用。
- InfluxDB 从未要求 token 的名称必须全局唯一，所以名称重复也是可以的。如图：



2) token 可以临时关停、也可以删除

正如你说看到，token 卡片下面的 Active 按钮是一个开关，可以在启用和停用之间进行切换。



同时，你也可以删除 token，但是这可能对你已经部署的应用产生不可挽回的影响。

3) 查看 Token 权限

点击 token 的名称，可以看到这个 token 具体有哪些权限。

这里我们比较两个 token，可以看到 tony's Token 的权限很高。



下面这个 Token 是我们前面示例，生成 Telegraf 配置的时候自动生成的 token。

WRITE example02 bucket / READ example02_conf telegraf config

Created at: 2022-08-22 12:35:15

Active

点开看一下它的权限。

WRITE example02 bucket / READ example02_conf telegraf config

-RczDbY-fQtx0frfK_UbMfQtwydjPITY2nbg19z8JJdnC4GVt58CSUWE0I5vpXFx0s7pbI8ic_c0wY-yJA_etQ==

COPY TO CLIPBOARD

Summary of access permissions

buckets-example02 write → 只能向example02存储桶写数据

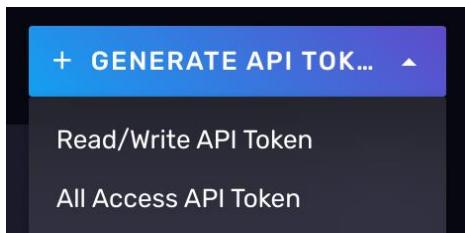
telegrafs read → 可以读取Telegraf配置

Summary of access permissions	
buckets-example02	write
telegrafs	read

可以看到这个 token 的权限就小得多了，它只能向一个存储桶里写数据，查的权限都没有呢。

3.1.8.4 创建 API Token

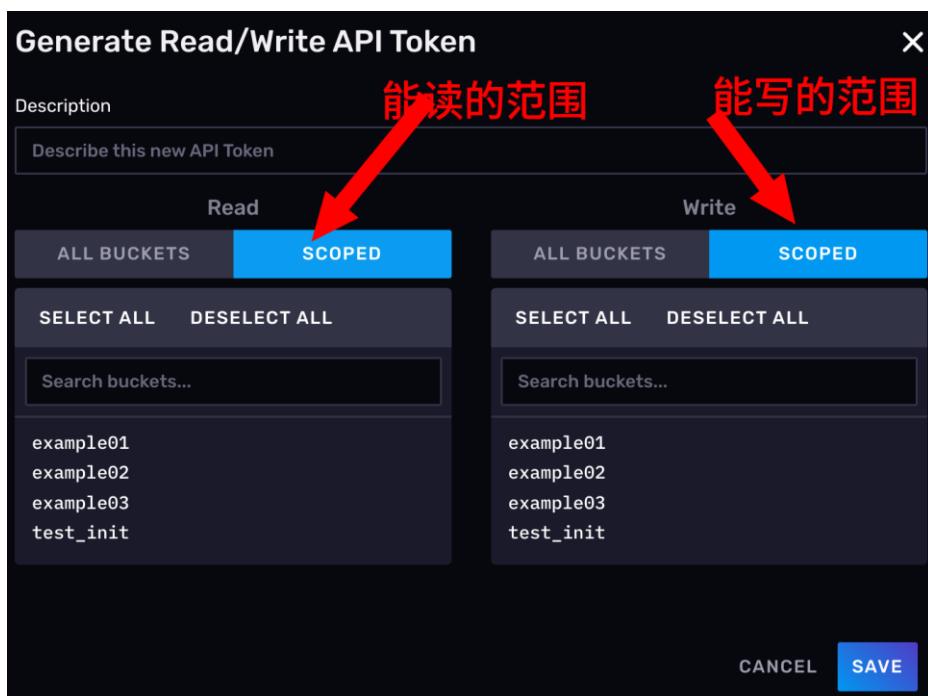
页面的右方有一个 GENERATE API TOKEN。点一下会出来一个下拉菜单，这其实是 Web UI 上的权限模板



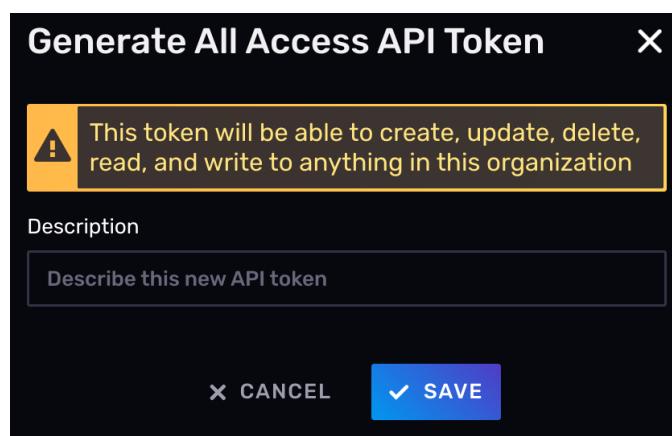
在 Web UI 上，有两种类型的模板让你可以快速创建 token。

- Read/Write API Token 仅读写存储桶的 Token

创建 Token 时还可以限定这个 Token 能操作哪些存储桶。



- All Access API Token 生成带所有权限的 Token



注意！InfluxDB 的 Token 是可以进行更细的管理的，Web UI 上给的只是生成 Token 的模板，准备了用户的常用需求，但不代表它的全部功能。

3.2 查询工具

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

如果继续后面内容的学习，最好先掌握附录 4：时序数据库中的数据模型 中的知识。

3.2.1 前言

关于 InfluxDB 的查询，需要用户掌握一门叫 FLUX 的语言。本节暂时不讲解 FLUX 语言的知识，而是先了解 InfluxDB 重要的两个开发工具——Data Explorer 和 Notebook。

3.2.2 了解 Data Explorer

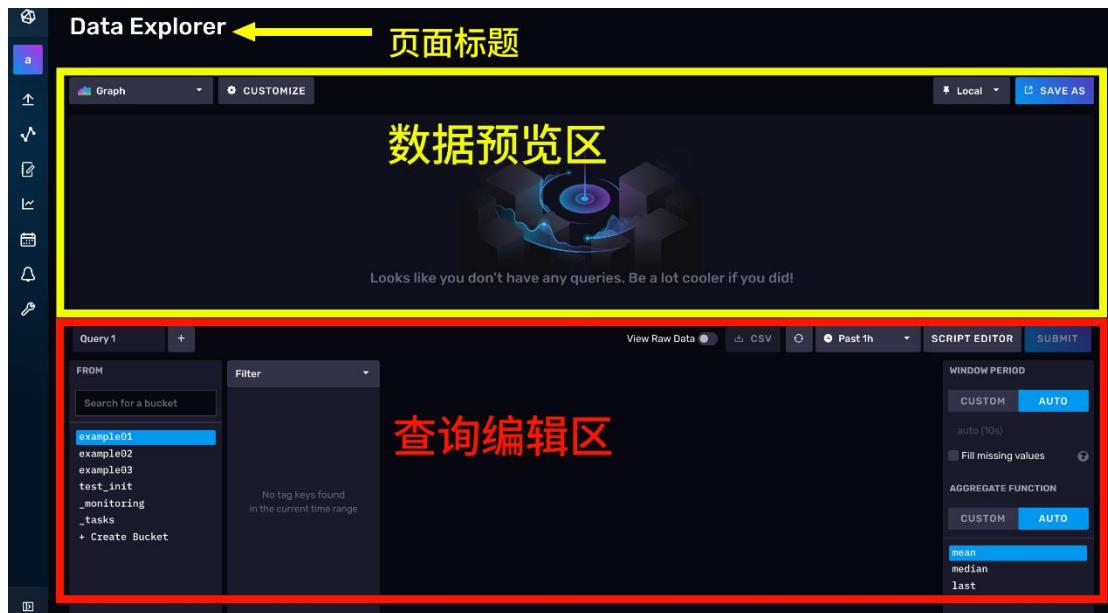
3.2.2.1 什么是 Data Explorer

explorer，探险家、探索者的意思。所以正如其字面意思，你可以使用 Data Explorer 探索数据，理解数据。说白了，就是你可以尝试性地写写 FLUX 查询语言（InfluxDB 独创的一门独立查询语言，课程后面会讲解），看一下数据的效果。开发过程中，你可以将它作为一个 FLUX 语言的 IDE。但是，目前我们不会向大家讲解 FLUX 语言。后面会为这门语言起一个专门的章节。

3.2.2.2 认识 Data Explorer 的页面

点击左边的  图标，进入 Data Explorer。

我们可以将 Data Explorer 的界面简单分为两个区域，上半部分为**数据预览区**，下半部分为**查询编辑区**。



3.2.2.3 查询编辑区

查询编辑区为你提供了两种查询工具，一个是查询构造器，一个是 FLUX 脚本编辑器。

(1) 查询构造器

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

你一进入 Data Explorer 页面，默认会打开查询构造器。使用查询构造器，你可以通过点按的方式完成查询。它背后的原理其实是根据你的设置，自动生成一条 FLUX 语句，提交给数据库完成查询。

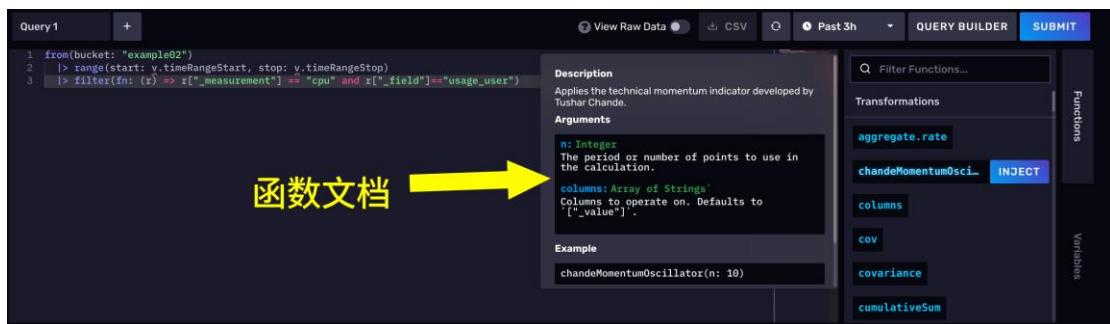
能够出现查询构造器这种东西，说明时序数据的查询之间遵循着某种规律。不同业务之间的查询步骤可能高度相似。



如上图，这是查询构造器的极简介绍。在后面的示例中，我们会详细讲解它的使用

(2) FLUX 脚本编辑器

你可以手动将查询构造器切换为 **FLUX 脚本编辑器**。然后愉快地编写 FLUX 脚本，实现各种奇葩查询。编辑器十分友好，还带自动提示和函数文档。



3.2.2.4 数据预览区

数据预览区可以将你的数据展示出来。下图是一个效果图。



默认情况下，数据预览区会将你的数据展示为一个折线图。不过除此之外，你还可以让数据展示为散点图、饼图或者查看原始数据等等。

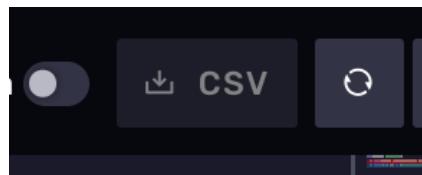
3.2.2.5 其他功能

除了查询和展示数据的功能外。

Data Explorer 还有一些拓展功能

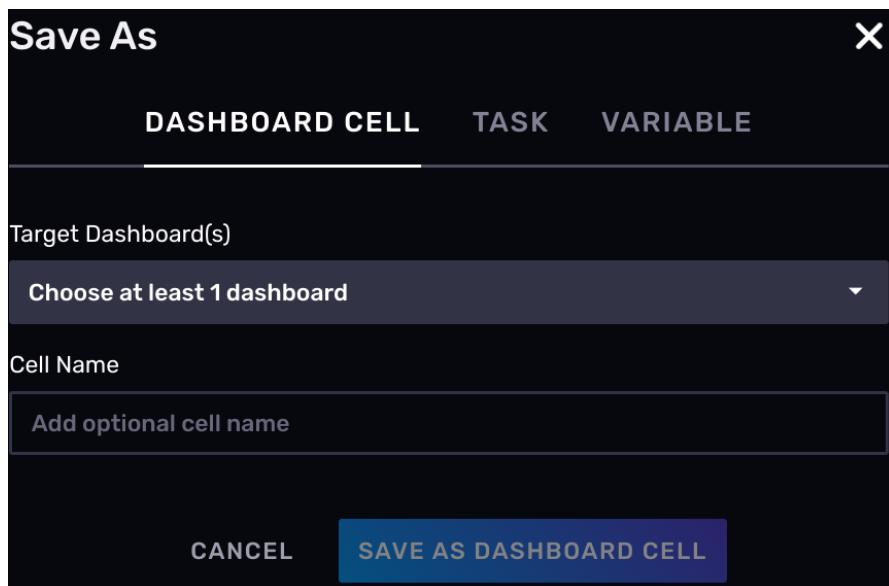
1) 将数据导出为 CSV

在执行查询之后，DataExplorer 允许你快速地将数据导出为一个 CSV 文件。

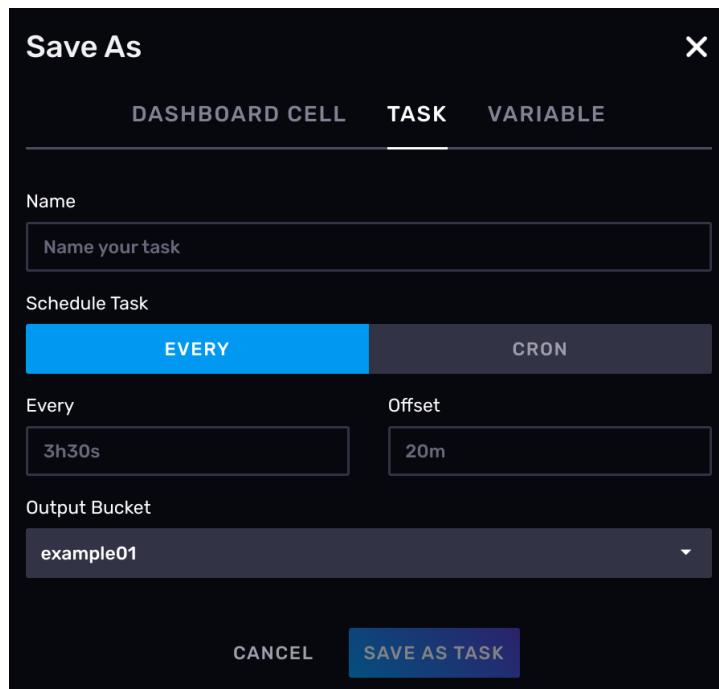


2) 将当前查询和可视化效果保存为仪表盘的一个单元

你可以将当前的查询逻辑和图形展示保存为某个仪表盘的一部分。这个功能需要在查询逻辑已经实现的前提下，点击右上角的 SAVE AS 触达。



3) 创建定时任务



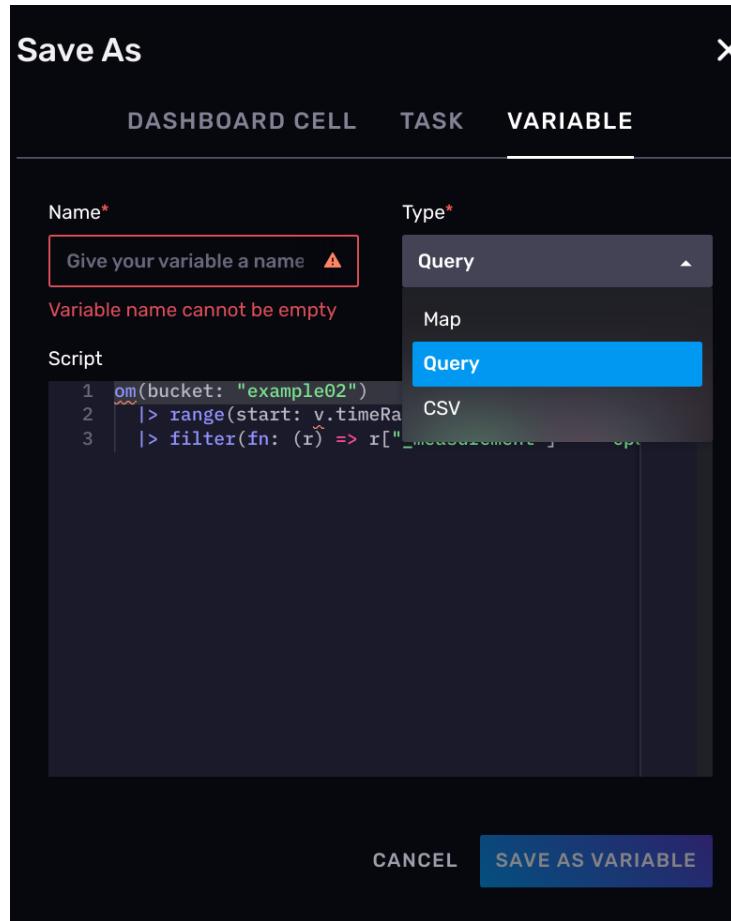
Data Explorer 中的查询逻辑可以保存为一个定时任务，也就是 TASK。这里提前说一下 InfluxDB 中的 TASK 是什么。TASK 其实是一个定时执行的 FLUX 语言写的脚本。因为 FLUX 是一个脚本语言，所以它其实有一定的 IO 能力。可以使用 http 与外面的系统进行通信，还可以将计算完的数据回写给 InfluxDB。所以通常 TASK 有两种使用场景。

(1) 数据检查与报警。对查询后的结果进行一下条件判断，如果不合规，就使用 http 向外通知报警。

(2) 聚合操作。在 InfluxDB 里开窗完成聚合计算，计算后的数据再写回到 InfluxDB，这样下游 BI（数据看板）可以直接去查询聚合后的数据了，而不是每次都把数据从 InfluxDB 里拉出来重新计算。这样可以减少 IO，不过会增加 InfluxDB 的压力。生产环境下需要根据实际情况进行取舍。

4) 定义全局变量

在 DataExplorer 里，你可以声明一些全局变量。全局变量的类型可以是 Map（键值对）、CSV 和 FLUX 脚本。这样，将来你可以直接引用这些变量，比如你的数据里有地区编码。你就可以将编码到地区名称的映射保存为一个全局 Map，供以后每次查询时使用。



3.2.3 示例 4：在 Data Explorer 使用查询构造器进行查询和可视化

3.2.3.1 打开 Data Explorer

点击左侧的  按钮，进入 Data Explorer 页面。

The screenshot shows the Data Explorer interface. On the left is a sidebar with various icons for navigation. The main workspace has a central area with a 'Create a query. Go on!' button. Below this are several panels: 'Query 1' (with a '+'), 'FROM' (listing buckets like example01, example02, example03, test, test_init, _monitoring, _tasks, + Create Bucket), 'Filter' (listing measurements like _measurement, _time, m1, m2), and configuration panels for 'SCRIPT EDITOR' (containing a query: 'SELECT mean("cpu_usage_percent") FROM "example01" WHERE _time > now() - 1h'), 'WINDOW PERIOD' (set to 'AUTO'), and 'AGGREGATE FUNCTION' (set to 'mean').

3.2.3.2 设置查询条件

我们现在要查询的是 test_init 存储桶下的 go_goroutines 测量，这个测量反应的是我们 InfluxDB 进程中的 goroutines（轻量级线程）数量。

首先，在左下角的查询构造器的 FROM 选项卡，选择 test_init 存储桶



接着会弹出一个 Filter 选项卡，默认情况下这里是选择_measurement，此处我们选择 go_goroutines。

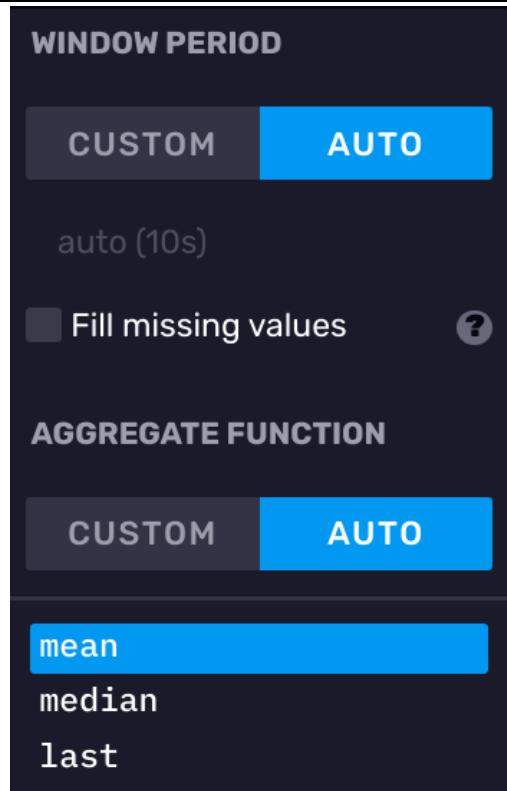
3.2.3.3 注意查询时间范围

右上角有一个带时钟符号的下拉菜单，这个菜单可以帮你纵向选择要查询数据的时间范围，通常默认是 1h。如下图所示：



3.2.3.4 注意右侧的窗口聚合选项

在查询构造器的最右边，有一个开窗聚合选项卡。使用查询构造器进行查询，就必须使用开窗聚合。默认情况下，DataExplorer 会根据你设置的查询时间范围，自动调整窗口大小，此处查询范围 1h 对应窗口大小 10s。

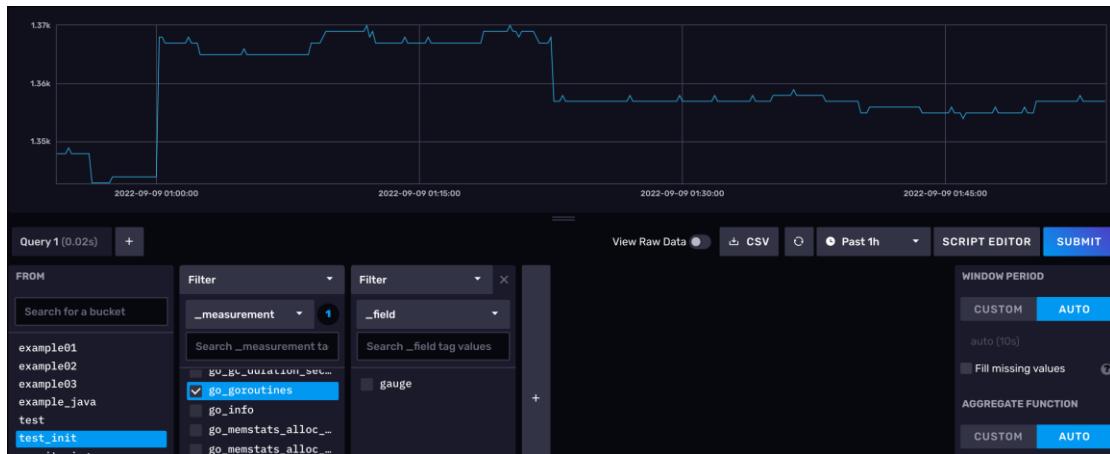


同时，聚合方式默认是平均值。

3.2.3.5 提交查询

点击右侧的 SUBMIT 按钮可以立刻提交查询。之后，数据展示区会出现相应的折线图。

如下图所示：



点击 View Raw Data，可以看到原始数据。

table _measurement _field _value _start _stop _time
 MEAN GROUP STRING DOUBLE GROUP DATETIME:RFC3339 GROUP DATETIME:RFC3339 INFLUXDB DATETIME:RFC3339
 0 go_goroutines gauge 1348 2022-09-08T16:54:06.275Z 2022-09-08T17:54:06.275Z 2022-09-08T16:54:20.000Z
 0 go_goroutines gauge 1348 2022-09-08T16:54:06.275Z 2022-09-08T17:54:06.275Z 2022-09-08T16:54:30.000Z
 0 go_goroutines gauge 1348 2022-09-08T16:54:06.275Z 2022-09-08T17:54:06.275Z 2022-09-08T16:54:40.000Z
 0 go_goroutines gauge 1348 2022-09-08T16:54:06.275Z 2022-09-08T17:54:06.275Z 2022-09-08T16:54:50.000Z
 0 go_goroutines gauge 1349 2022-09-08T16:54:06.275Z 2022-09-08T17:54:06.275Z 2022-09-08T16:55:00.000Z

点击View Raw Data查看原始数据

3.2.3.6 查询原理

我们使用查询构造器进行查询，其实是 Web UI 根据我们指定的查询条件生成了一套 FLUX 查询脚本。点击 SCRIPT EDITOR 按钮，可以看到查询构造器生成的 FLUX 脚本。

```
1 from(bucket: "test_init")
2 |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
3 |> filter(fn: (r) => r[_measurement] == "go_goroutines")
4 |> aggregateWindow(every: v.windowPeriod, fn: mean, createEmpty: false)
5 |> yield(name: "mean")
```

3.2.3.7 可视化原理

其实默认情况下的可视化，是依据返回数据中的_value 来展示的，但是有些时候，你想查询的数据可能字段名不会被判别为_value。它会安静地躺在原始数据中。



3.2.4 了解 Notebook

3.2.4.1 什么是 Notebook

Notebook 是 InfluxDB2.x 推出的功能，交互上模仿了 Jupyter NoteBook。它可以用于开发、文档编写、运行代码和展示结果。

你可以将 InfluxDB 笔记本视为按照顺序处理数据的集合。每个步骤都由一个“单元格”表示。一个单元格可以执行查询、可视化、处理或将数据写入存储桶等操作。Notebook 可以帮你完成下述操作

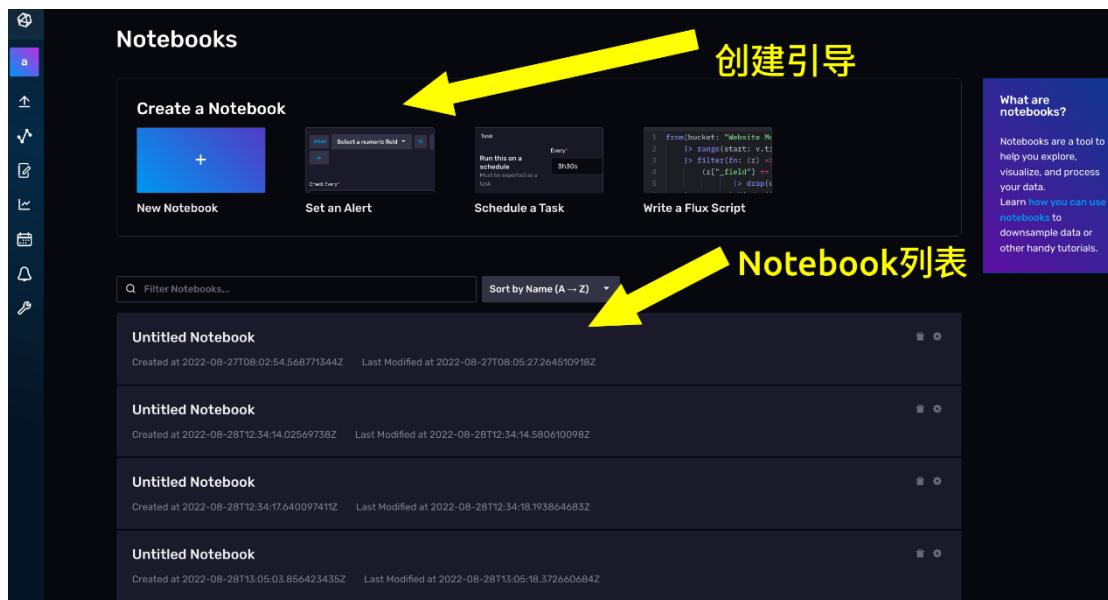
- 执行 FLUX 代码、可视化数据和添加注释性的片段
- 创建报警或者计划任务
- 对数据进行降采样或者清洗

- 生成要和团队分享的 Runbooks
- 将数据回写到存储桶

Notebook 和 DataExplorer 相比，主要是交互风格上的不同。DataExplorer 倾向于一锤子买卖，而 Notebook 可以将数据展示拆分为一个又一个具体的步骤。另外，NoteBook 可以用来开发告警任务 DataExplorer 则不能。

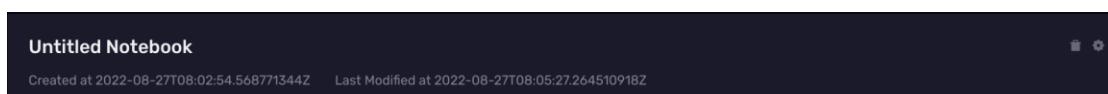
3.2.4.2 进入 Notebook 的导航界面

点击左侧的  按钮，即可进入 Notebook 的导航页面。



导航页面分两个部分：

- 上面是创建引导，除了创建一个空白的 Notebook，InfluxDB 还为你提供了 3 个模板。分别是 Set an Alert（设置一个报警）、Schedule a Task（调度一个任务）、write a Flux Script（写一个 Flux 脚本）。
- 下面是 Notebook 列表，过去你创建过的 NoteBook 再这里都会展示出来。

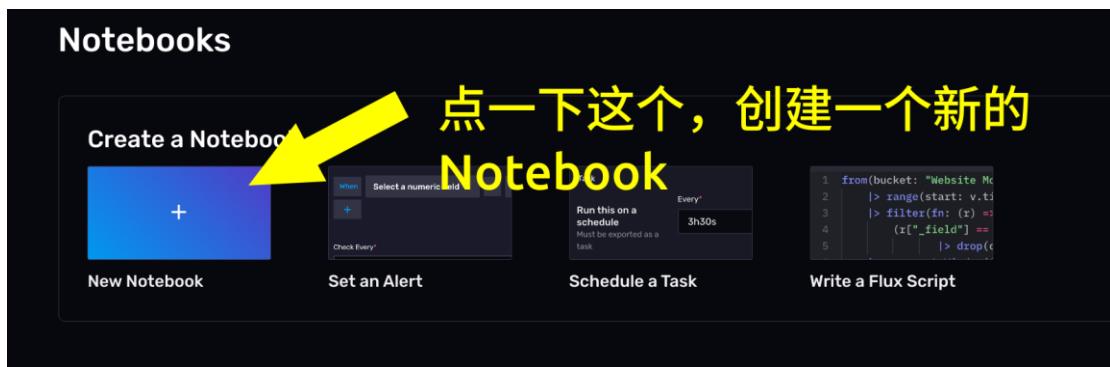


卡片上还有这个 Notebook 对应的创建时间和修改时间。通过卡片你可以对一个 Notebook 重命名，还可以将它复制和删除。

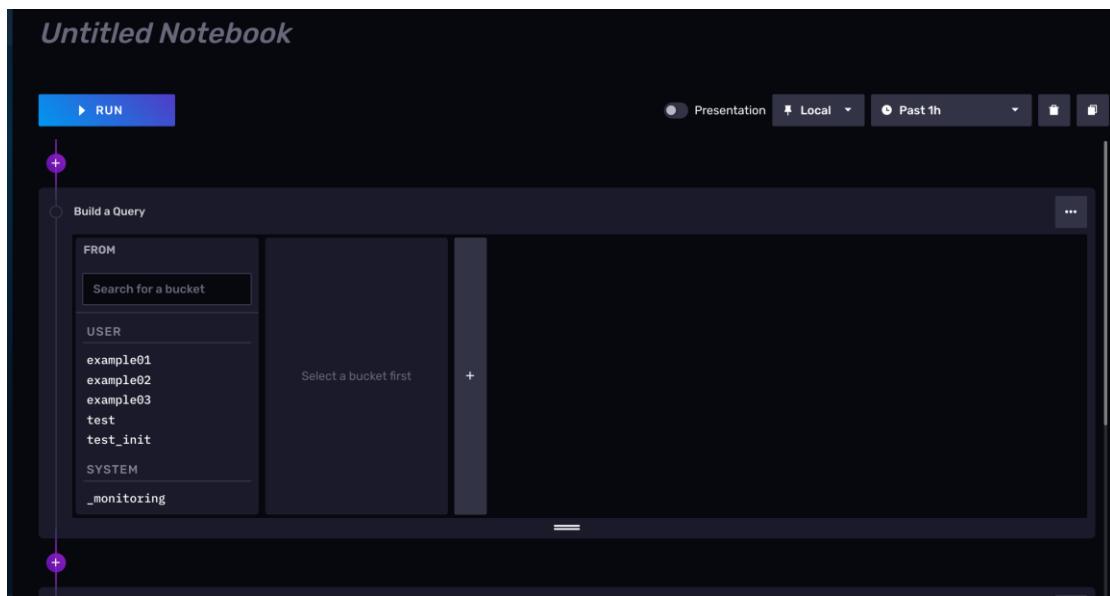
3.2.4.3 创建一个空白的 notebook

想要继续后面的步骤，我们必须先创建一个 Notebook。如下图所示，在页面上方点击 New Notebook 按钮即可。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

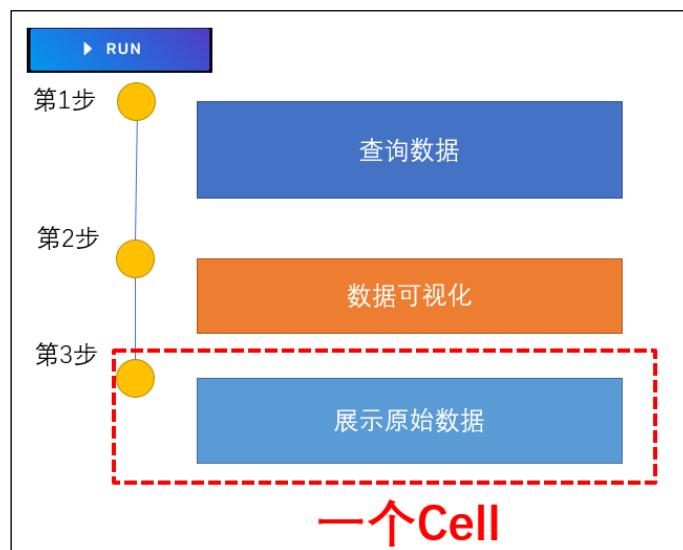


现在，你看到的就是 Notebook 的操作页面了。



3.2.4.4 NoteBook 工作流

目前你看到的页面应当是如下图所示的样子。

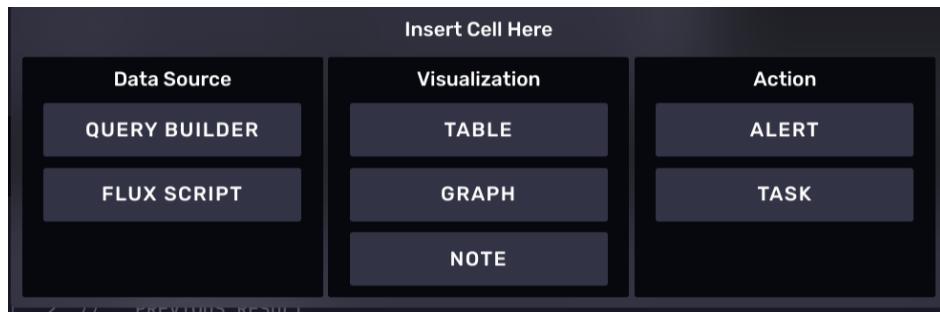


我们在页面中看到的一个又一个卡片，在 NoteBook 中叫做 Cell。一个 NoteBook 工作

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

流就是多个 Cell 按照先后顺序组合起来的执行流程。这些 Cell 中间随时可以插入别的 Cell，而且 Cell 和 Cell 还可以调换顺序。

按照 Cell 功能，Cell 可以按照下面的方式分类。



- 数据源相关的 Cell
 - 查询构造器
 - 直接编写 FLUX 脚本
 - 可视化相关的 Cell
 - 将数据展示为一个 Table
 - 将数据展示为一张图
 - 添加笔记。
 - 行为 Cell
 - 进行报警
 - 定时任务设定

3.2.4.5 工作流范式

在 NoteBook 里编写工作流通常是有套路可循的。



通常一个 notebook 工作流以查询数据开始，后面的 Cell 跟上把数据展示出来，当数据需要进一步修改的时候，可以再加一个 FLUX 脚本 cell，notebook 为我们留了一个接口，通过这种方式，后面的 Flux cell 可以将前面的数据作为数据源进行查询。

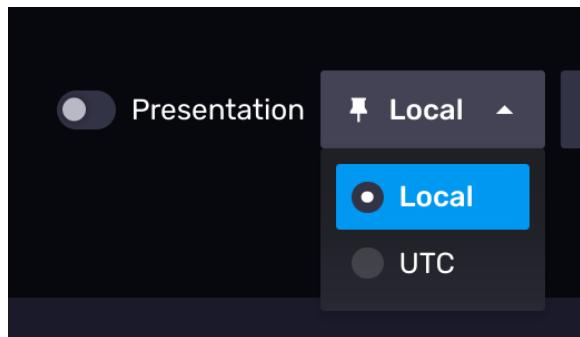
最终，notebook 工作流可以以任务设置或者报警操作作为整个工作流的终点，当然这不是强制要求。

3.2.4.6 NoteBook 控件

在 notebook 上存在下述几种控件

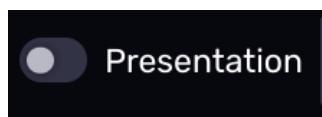
1) 时区转换

右上角有一个 Local 按钮，通过这个按钮，你可以选择将日期时间显示为系统所设时区还是 UTC 时间。



2) 仅显示可视化

点击 Presentation 按钮，可以选择是否仅显示数据展示的 cell。如果开启这个选项，那么查询构造器和 FLUX 脚本的 Cell 就会被折叠。



3) 删除按钮

点击确定后，可以删除整个 notebook。



4) 复制按钮

右上角的复制按钮可以立刻为当前 NoteBook 创建一个副本。



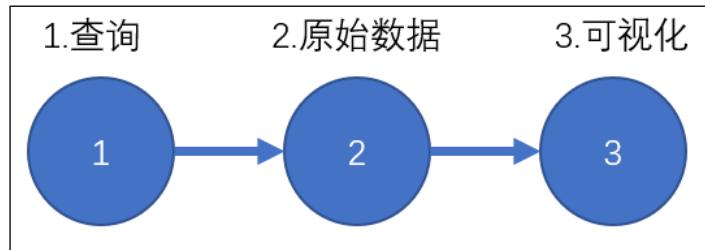
5) 运行按钮

RUN 按钮可以快速地执行 Notebook 中的查询操作并重新渲染其中的可视化 Cell。

3.2.5 示例 5：使用 NoteBook 查询和可视化数据

3.2.5.1 使用查询构造器记性查询

默认情况下，你创建的空白 NoteBook，自带 3 个 cell。



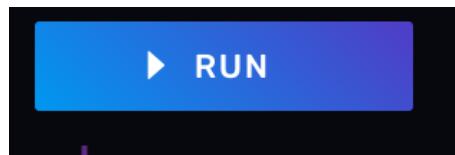
第一个 cell， 默认是一个查询构造器，相对于 DataExplorer 来说， notebook 的查询构造器不同的地方在于它没有开窗聚合操作。

此处，同样还是查询 test_init 中的 go_goroutines 测量。

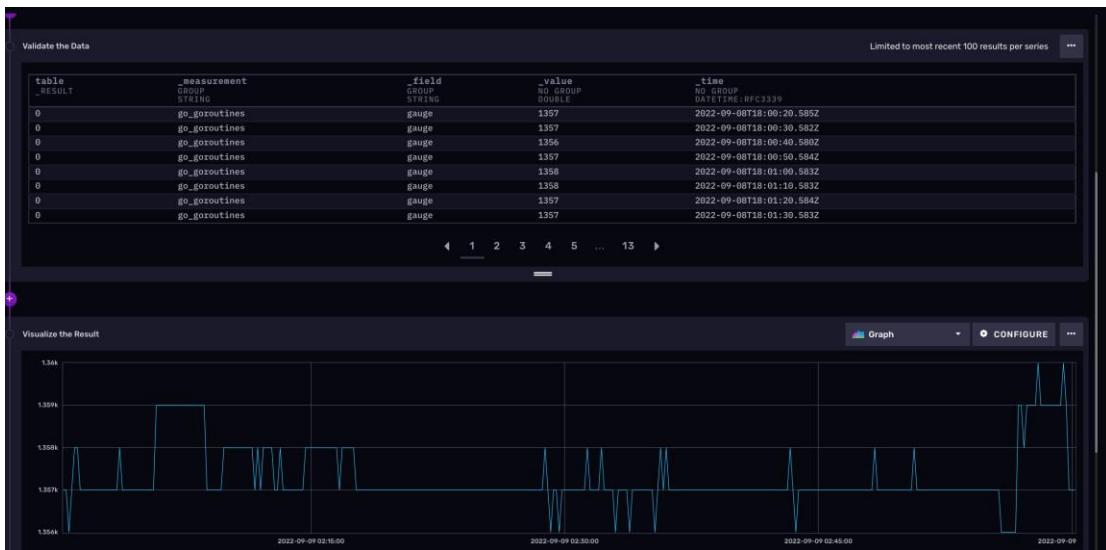
The screenshot shows the 'Build a Query' interface. The 'FROM' section has 'test' and 'test_init' selected. The 'FILTER' section shows '_measurement' set to 'go_goroutines' with a count of 1. The 'SAMPLE' section lists various metrics like 'go_gc_duration_sec...', 'go_info', etc.

3.2.5.2 提交查询

点击 RUN 按钮。



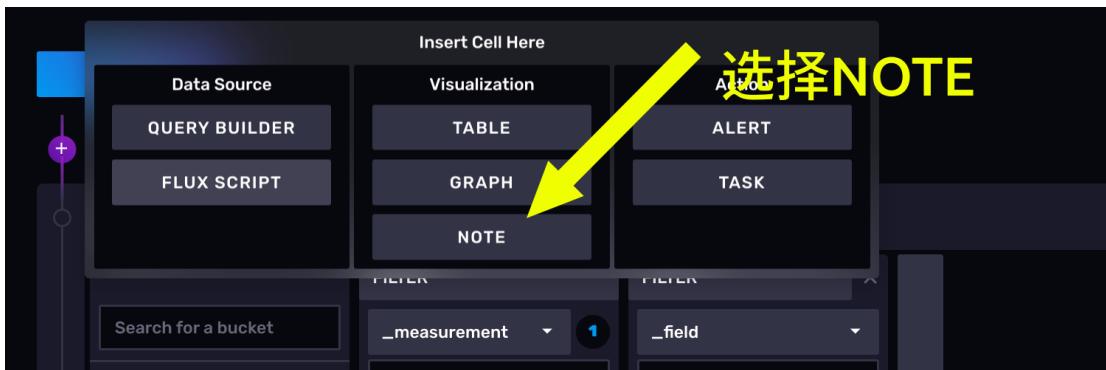
可以看到下面的原始数据和折线图都出现了



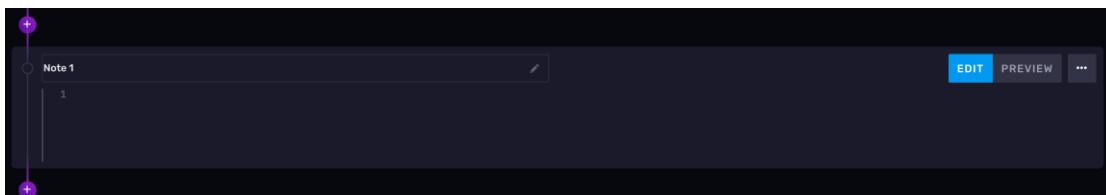
3.2.5.3 添加说明 cell

notebook 允许用户在工作流中加入说明性的 cell。我们选择在最前面加一个说明性 cell。

首先，点击左侧的紫色十号。



点击 NOTE 按钮。可以看到，我们已经创建了一个说明 cell。这里面还支持
MarkDown 语法，



现在，我们随便写点东西

```
1 # 你好
2 go_goroutines是轻量级线程数
```

点击右上右上角的 PREVIEW 按钮，markdown 就会被渲染展示。



第4章 FLUX 语法

4.1 认识 FLUX 语言

Flux 是一种函数式的数据脚本语言，它旨在将查询、处理、分析和操作数据统一为一种语法。

想要从概念上理解 FLUX，你可以想想水处理的过程。我们从源头把水抽取出来，然后按照我们的用水需求，在管道上进行一系列的处理修改（去除沉积物，净化）等，最终以消耗品的方式输送到我们的目的地（饮水机、灌溉等）。



注意：InfluxData 公司对 FLUX 语言构想并不是仅仅让它作为 InfluxDB 的特定查询语言，而是希望它像 SQL 一样，成为一种标准。按照这个计划，FLUX 语言应该具备处理来自不同数据源的数据的能力。

4.2 最简示例

与处理水一样，使用 FLUX 语言进行查询时会执行以下操作。

- (1) 从数据源中查询指定数量的数据
- (2) 根据时间或字段筛选数据

(3) 将数据进行处理或者聚合以得到预期结果

(4) 返回最终的结果

下面 3 个示例的处理逻辑都是一样的，只不过数据源有所不同，

这 3 个示例只是让大家看一下语法，不需要运行。

示例 1：从 InfluxDB 查询数据并聚合

```
from(bucket: "example-bucket")
|> range(start: -1d)
|> filter(fn: (r) => r._measurement == "example-measurement")
|> mean()
|> yield(name: "_results")
```

示例 2：从 CSV 文件查询数据并聚合

```
import "csv"

csv.from(file: "path/to/example/data.csv")
|> range(start: -1d)
|> filter(fn: (r) => r._measurement == "example-measurement")
|> mean()
|> yield(name: "_results")
```

示例 3：从 PostgreSQL 数据库查询数据并聚合

```
import "sql"

sql.from(
  driverName: "postgres",
  dataSourceName: "postgresql://user:password@localhost",
  query: "SELECT * FROM TestTable",
)
|> filter(fn: (r) => r.UserID == "123ABC456DEF")
|> mean(column: "purchase_total")
|> yield(name: "_results")
```

上面 3 个示例用的函数都是一模一样的，下面来讲解示例中出现的代码：

- `from()` 函数可以指定数据源。
- `|>` 管道转发符，将一个函数的输出转发给下一个函数。
- `range()`, `filter()` 两个函数在根据列的值对数据进行过滤
- `mean()` 函数在计算所剩数据的平均值。
- `yield()` 将最终的计算结果返回给用户。

4.3 铭记 FLUX 是一门查询语言

虽然，FLUX 语言的自我定位一个脚本语言，但是我们必须注意它也是一个查询语言的事实。因此，一个 FLUX 脚本想要成功执行，它就必须返回一个表流。就像是 SQL 语言想要正确执行，它就必须返回一张表。

表流是 FLUX 里提出一种数据结构，在后面的课程里我们会表流的概念进行深度的讲
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

解。另外需要注意，我们后面的代码，如果只返回一个单值，比如单个整数或者字符串这种，那就必须把这个值转换成表流才能运行。这个时候必须使用 `array.from` 函数。

示例如下：

```
from "array
x = 1
array.from(rows: [{"value":x}])
```

`array.from` 函数的作用就是把 `x` 这个单值，包装在了一个表流里面返回了。

4.4 注意 InfluxDB 支持的 FLUX 语言版本

需要注意，因为 InfluxDB 是一个用 Go 语言编写的数据库，它的整个项目成果就是一个单独的可执行二进制文件，所以 FLUX 语言其实也会被编译到同一个文件里。这意味着 InfluxDB 和 FLUX 会有版本绑定的关系。

这里，我放了一个链接 <https://docs.influxdata.com/flux/v0.x/influxdb-versions/>，它是官方 FLUX 文档的一部分，这里明确记录了 InfluxDB 版本的 FLUX 语言版本的对应关系。

InfluxDB Open Source (OSS)

InfluxDB OSS version	Flux version
InfluxDB nightly	0.185.0
InfluxDB 2.4	0.179.0
InfluxDB 2.3	0.171.0
InfluxDB 2.2	0.162.0
InfluxDB 2.1	0.139.0
InfluxDB 2.0	0.131.0
InfluxDB 1.8	0.65.1
InfluxDB 1.7	0.50.2

4.5 FLUX 的基本语法

4.5.1 注释

在 FLUX 脚本中，没有多行注释一说，用户只能写单行注释。如果一行以两个斜杠开

头，那么这一行中的所有内容会被视为注释。

示例：

```
// 这是一行注释。
```

4.5.2 变量与复制

使用赋值运算符 (=) 将表达式的结果赋值变量，最终你可以使用变量名来返回变量的值。

示例：

```
s = "foo" // string
i = 1 // integer
f = 2.0 // float (floating point number)

s // Returns foo
i // Returns 1
f // Returns 2.0
```

4.5.3 基本表达式

FLUX 支持基本的表达式，比如：

- + 数字相加或字符串拼接
- - 数字减法
- * 数字相乘
- / 数字除法
- % 取模

示例：

```
1 + 1
// Returns 2

10 * 3
// Returns 30

(12.0 + 18.0) / (2.0 ^ 2.0) + (240.0 % 55.0)
// Returns 27.5

"John " + "Doe " + "is here!"
// Returns John Doe is here!
```

4.5.4 谓词表达式

4.5.4.1 比较运算符

谓词表达式使用比较运算符和逻辑运算符来实现，谓词表达式的最后的返回结果只能为 true 或 false

示例：

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
"John" == "John"  
// Returns true  
  
41 < 30  
// Returns false  
  
"John" == "John" and 41 < 30  
// Returns false  
  
"John" == "John" or 41 < 30  
// Returns true
```

另外

- =~可以判断一个字符串时候能被正则表达式匹配上。
- !~是=~的反操作，判断一个字符串是不是不能被某个正则表达式匹配。

例如：

```
"abcdefg" =~ "abc|bcd"  
// Returns true  
"abcdefg" !~ "abc|bcd"  
// Returns false
```

4.5.4.2 逻辑运算符

在 FLUX 语言中，表示与逻辑需要使用关键字 and，表示或逻辑需要使用关键字 or。

示例：

```
a = true  
b = false  
x = a and b  
// Returns false  
  
y = a or b  
// Returns true
```

最后，not 可以用来进行逻辑取反。

示例：

```
a = true  
b = not a  
// Returns false
```

4.5.5 控制语句

所谓控制语句是指一个编程语言中用来空值代码执行顺序的语法。

比如：

- if else
- for while 循环
- try catch 异常捕获

不过，在 InfluxDB 中，这些语法统统没有。唯一一个和 if else 比较像的是 FLUX 语言

中的条件子句，它和 python 中的条件子句功能一样且语法相似，和 java 语言相比的话它有些像三元表达式。

示例如下：

```
x = 0
y = if x == 0 then "hello" else "world"
```

此处，if then else 被我们成为条件子句，你需要先指定一个条件，然后当条件为 true 的时候，条件子句会返回 then 后面的内容，也就是"hello"。如果是 flase，那么就会返回 else 后面的内容，也就是"world"。

第5章 FLUX 中的数据类型

5.1 10 个基本数据类型

5.1.1 Boolean (布尔型)

5.1.1.1 将数据类型转换为 boolean

使用 bool()函数可以将下述的 4 个基本数据类型转换为 boolean:

string (字符串): 字符串必须是 "true" 或 "false"

float (浮点数): 值必须是 0.0 (false) 或 1.0 (true)

int (整数): 值必须是 0 (false) 或 1 (true)

uint (无符号整数): 值必须是 0 (false) 或 1 (true)

示例：

```
bool(v: "true")
// Returns true

bool(v: 0.0)
// Returns false

bool(v: 0)
// Returns false

bool(v: uint(v: 1))
// Returns true
```

5.1.2 bytes (字节)

注意是 bytes (复数) 不是 byte， bytes 类型表示一个由字节组成的序列。

5.1.2.1 定义 bytes

FLUX 没有提供关于 bytes 的语法。可以使用 bytes 函数将字符串转为 bytes。

```
bytes(v:"hello")
// Returns [104 101 108 108 111]
```

注意：只有字符串类型可以转换为 bytes。

5.1.2.2 将表示十六进制的字符串转为 bytes

(1) 引入 "contrib/bonitoo-io/hex" 包

(2) 使用 hex.bytes() 将表示十六进制的字符串转为 bytes

```
import "contrib/bonitoo-io/hex"

hex.bytes(v: "FF5733")
// Returns [255 87 51] (bytes)
```

5.1.2.3 使用 display() 函数获取 bytes 的字符串形式

使用 display() 返回字节的字符串表示形式。bytes 的字符串表示是 0x 开头的十六进制表示。

示例：

```
import "sampledata"

sampledata.string()
    |> map(fn: (r) => ({r with _value: display(v: bytes(v: r._value))}))
```

result			
table	_time	_value	tag*
0	2021-01-01T00:00:00Z	0x736d706c5f673971637a73	t1
0	2021-01-01T00:00:10Z	0x736d706c5f306d6776396e	t1
0	2021-01-01T00:00:20Z	0x736d706c5f706877363634	t1
0	2021-01-01T00:00:30Z	0x736d706c5f6775767a7934	t1

5.1.3 Duration 持续时间

持续时间提供了纳秒级精度的时间长度。

5.1.3.1 持续时间的语法

- ns： 纳秒
- us： 微秒
- ms： 毫秒
- s : 秒
- m : 分钟
- h : 小时
- d : 天
- w : 周

- mo: 日历月
- y : 日历年

示例:

```
1ns // 1 纳秒  
1us // 1 微妙  
1ms // 1 毫秒  
1s // 1 秒  
1m // 1 分钟  
1h // 1 小时  
1d // 1 天  
1w // 1 星期  
1mo // 1 日历月  
1y // 1 日历年  
  
3d12h4m25s // 3 天 12 小时 4 分钟又 25 秒
```

注意！持续时间的声明不要包含先导 0

比如:

```
01m // 解析为整数 0 和 1 分钟的持续时间  
02h05m // 解析为整数 0、2 小时的持续时间，整数 0 和 5 分钟的持续时间。而不是 2  
小时又 5 分钟
```

5.1.3.2 将其他数据类型解释为持续时间

使用 duration()函数可以将以下基本数据类型转换为持续时间

字符串: 将表示持续时间字符串的函数转换为持续时间。

int: 将整数先解释为纳秒再转换为持续时间

unit: 将整数先解释为纳秒再转换为持续时间。

```
duration(v: "1h30m")  
// Returns 1h30m  
  
duration(v: 1000000)  
// Returns 1ms  
  
duration(v: uint(v: 3000000000))  
// Returns 3s
```

注意！你可以在 FLUX 语言中使用 duration 类型的变量与时间做运算，但是你不能在 table 中创建 duration 类型的列。

5.1.3.3 duration 的算术运算

要对 duration 进行加法、减法、乘法或除法操作，需要按下面的步骤来。

- (1) 使用 int()或 unit()将持续时间转换为 int 数值
- (2) 使用算术运算符进行运算

(3) 把运算的结果再转换回 Duration 类型

示例：

```
duration(v: int(v: 6h4m) + int(v: 22h32s))
// 返回 1d4h4m32s

duration(v: int(v: 22h32s) - int(v: 6h4m))
// 返回 15h56m32s

duration(v: int(v: 32m10s) * 10)
// 返回 5h21m40s

duration(v: int(v: 24h) / 2)
// 返回 12h
```

注意！声明持续时间的时候不要包含前导 0，前面的零会被 FLUX 识别为整数

5.1.3.4 时间和持续时间相加运算

(1) 导入 date 包

(2) 使用 date.add() 函数将持续时间和时间相加

示例：

```
import "date"

date.add(d: 1w, to: 2021-01-01T00:00:00Z)
// 2021-01-01 加上一周
// Returns 2021-01-08T00:00:00.000000000Z
```

5.1.3.5 时间和持续时间相减运算

(1) 导入 date 包

(2) 使用 date.add() 函数从时间中减去持续时间

示例：

```
import "date"

date.sub(d: 1w, from: 2021-01-01T00:00:00Z)
// 2021-01-01 减去一周
// Returns 2020-12-25T00:00:00.000000000Z
```

5.1.4 Regular expression 正则表达式

5.1.4.1 定义一个正则表达式

FLUX 语言是 GO 语言实现的，因此使用 GO 的正则表达式语法。正则表达式需要声明在正斜杠之间 / /

5.1.4.2 使用正则表达式进行逻辑判断

使用正则表达式进行逻辑判断，需要使用 =~ 和 != 操作符。=~ 的意思是左值（字符串）

能够被右值匹配，!<~表示左值（字符串）不能被右值匹配。

```
"abc" =~ /\w/
// Returns true

"z09se89" =~ /^[a-z0-9]{7}$/
// Returns true

"foo" !~ /^f/
// Returns false

"FOO" =~ /(?!i)foo/
// Returns true
```

5.1.4.3 将字符串转为正则表达式

(1) 引入 regexp 包

(2) 使用 regexp.compile() 函数可以将字符串转为正则表达式

```
import "regexp"

regexp.compile(v: "^- [a-z0-9]{7}")
// Returns ^- [a-z0-9]{7} (regexp type)
```

5.1.4.4 将匹配的子字符串全部替换

(1) 引入 regexp 包

(2) 使用 regexp.replaceAllString() 函数，并提供下列参数：

- r: 正则表达式
- v: 要搜索的字符串
- t: 一旦匹配，就替换为该字符串

示例：

```
import "regexp"

regexp.replaceAllString(r: /a(x*)b/, v: "-ab-axxb-", t: "T")
// Returns "-T-T-"
```

5.1.4.5 得到字符串中第一个匹配成功的结果

(1) 导入 regexp 包

(2) 使用 regexp.findString() 来返回正则表达式匹配中的第一个字符串，需要传递以下参数：

- r: 正则表达式
- v: 要进行匹配的字符串

示例：

```
import "regexp"

regexp.findString(r:"abc|bcd", v:"xxabcwwwed")
```

```
// Returns "abc"
```

5.1.5 String 字符串

5.1.5.1 定义一个字符串

字符串类型表示一个字符序列。字符串是不可改变的，一旦创建就无法修改。

字符串是一个由双引号括起来的字符序列，在 FLUX 中，还支持你用\x 作为前缀的十六进制编码来声明字符串。

示例：

```
"abc"  
"string with double \" quote"  
"string with backslash \\"  
"日本語"  
"\xe6\x97\xaa\xe6\x9c\xac\xe8\xaa\x9e"
```

5.1.5.2 将其他基本数据类型转换为字符串

使用 string()函数可以将下述基本类型转换为字符串：

- boolean 布尔值
- bytes 字节序列
- duration 持续时间
- float 浮点数
- uint 无符号整数
- time 时间

```
string(v: 42)  
// 返回 "42"
```

5.1.5.3 将正则表达式转换为字符串

因为正则表达式也是一个基本数据类型，所以正则表达式也可以转换为字符串，但是需要借助额外的包。

- (1) 引入 regexp 包
- (2) 使用 regexp.compile()将

5.1.6 Time 时间点

5.1.6.1 定义一个时间点

一个 time 类型的变量其实是一个纳秒精度的时间点。

示例：时间点必须使用 RFC3339 的时间格式进行声明

```
YYYY-MM-DD  
YYYY-MM-DDT00:00:00Z
```

```
YYYY-MM-DDT00:00:00.000Z
```

5.1.6.2 date 包

date 包里的函数主要是用来从 Time 类型的值里提取年月日秒等信息的。

比如 date.hour:

```
import "date"
x = 2020-01-01T19:22:31Z
date.hour(t:x)
//Returns 19
```

5.1.7 Float 浮点数

5.1.7.1 定义一个浮点数

FLUX 中的浮点数是 64 位的浮点数。

一个浮点数包含整数位，小数点，和小数位。

示例：

```
0.0
123.4
-123.456
```

5.1.7.2 科学计数法

FLUX 没有直接提供科学计数法语法，但是你可以使用字符串换写出一个科学计数法表示的浮点数，再使用 float() 函数将该字符串转换为浮点数。

示例：

```
1.23456e+78
// Error: error @1:8-1:9: undefined identifier e

float(v: "1.23456e+78")
// Returns 1.23456e+78 (float)
```

5.1.7.3 无限

FLUX 也没有提供关于无限的语法，定义无限要使用字符串与 float() 函数结合的方式。

示例：

```
+Inf
// Error: error @1:2-1:5: undefined identifier Inf

float(v: "+Inf")
// Returns +Inf (float)
```

5.1.7.4 Not a Number 非数字

FLUX 语言不支持直接从语法上声明 NAN，但是你可以使用字符串与 float() 函数的方法声明一个 NaN 的 float 类型变量。

示例：

```
NaN
```

```
// Error: error @1:2-1:5: undefined identifier NaN  
float(v: "NaN")  
// Returns NaN (float)
```

5.1.7.5 将其他基本类型转换为 float

使用 float 函数可以将基本数据类型转换为 float 类型的值。

string: 必须得是一个符合数字格式的字符串或者科学计数法。

bool: true 转换为 1.0, false 转换为 0.0

int (整数)

uint (无符号整数)

示例:

```
float(v: "1.23")  
// 1.23  
  
float(v: true)  
// Returns 1.0  
  
float(v: 123)  
// Returns 123.0
```

5.1.7.6 对浮点数进行逻辑判断

使用 FLUX 表达式来比较浮点数。逻辑表达式两侧必须是同一种类型。

示例:

```
12345600.0 == float(v: "1.23456e+07")  
// Returns true  
  
1.2 > -2.1  
// Returns true
```

5.1.8 Integer 整数

5.1.8.1 定义一个整数

一个 integer 的变量是一个 64 位有符号的整数。

类型名称: int

最小值: -9223372036854775808

最大值: 9223372036854775807

一个整数的声明就是普通的整数写法，前面可以加 - 表示负数。-0 和 0 是等效的。

示例:

```
0  
2  
1254  
-1254
```

5.1.8.2 将数据类型转换为整数

使用 `int()` 函数可以将下述的基本类型转换为整数：

string: 字符串必须符合整数格式，由数字[0-9]组成

bool: true 返回 1, 0 返回 false

duration: 返回持续时间的纳秒数

time: 返回时间点对应的 Unix 时间戳纳秒数

float: 返回小数点前的整数部分，也就是截断

unit: 返回等效于无符号整数的整数，如果超出范围，就会发生整数环绕

```
int(v: "123")
// 123

int(v: true)
// Returns 1

int(v: 1d3h24m)
// Returns 986400000000000

int(v: 2021-01-01T00:00:00Z)
// Returns 16094592000000000000

int(v: 12.54)
// Returns 12
```

你可以在将浮点数转换为整数之前进行舍入操作。

当你将浮点数转换为整数时，会进行截断操作。如果你想进行四舍五入，可以使用 `math` 包中的 `round()` 函数。

5.1.8.3 将表示十六进制数字的字符串转换为整数

将表示十六进制数字的字符串转换为整数，需要。

(1) 引入 contrib/bonito-jo/hex 包

(2) 使用 `hex.int()` 函数将表示十六进制数字的字符串转换为整数

```
import "contrib/bonitoo-io/hex"  
  
hex.int(v: "e240")  
// Returns 123456
```

5.1.9 UIntegers 无符号整数

FLUX 语言里不能直接声明无符号整数，但这却是一个 InfluxDB 中具备的类型。在 FLUX 语言中，我们需要使用 `uint` 函数来讲字符串、整数或者其他数据类型转换成无符号整数。

示例：

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

```
uint(v: "123")
// 123

uint(v: true)
// Returns 1

uint(v: 1d3h24m)
// Returns 9864000000000000

uint(v: 2021-01-01T00:00:00Z)
// Returns 16094592000000000000

uint(v: 12.54)
// Returns 12

uint(v: -54321)
// Returns 18446744073709497295
```

5.1.10 Null 空值

5.1.10.1 定义一个 Null 值

FLUX 语言并不能在语法上直接支持声明一个 Null，但是我们可以通过 `debug.null` 这个函数来声明一个指定类型的空值。

示例：

```
import "internal/debug"

// Return a null string
debug.null(type: "string")

// Return a null integer
debug.null(type: "int")

// Return a null boolean
debug.null(type: "bool")
```

5.1.10.2 定义一个 null

截至目前，还无法在 FLUX 语言中手动地声明一个 NULL 值。

注意！空字符串不是 null 值

5.1.10.3 判断值是否为 null

你可以使用 `exists`（存在）这个关键字来判断目标值是不是非空，如果是空值我们会得到一个 `false`，如果不是空值我们会得到一个 `true`。

示例：

```
import "array"
import "internal/debug"

x = debug.null(type: "string")
y = exists x
// Returns false
```

5.1.11 正则表达式类型

正则表达式在 FLUX 中作为一种数据类型，而且在语法上提供直接的支持，可以在谓词表达式中使用正则表达式。

示例：

```
regex = /^foo/  
  
"foo" =~ regex  
// Returns true  
  
"bar" =~ regex  
// Returns false
```

5.1.12 display 函数

使用 display() 函数可以将任何类型的值输出为相应的字符串类型。

示例：

```
x = bytes(v: "foo")  
  
display(v: x)  
// Returns "0x666f6f"
```

5.2 FLUX 类型不代表 InfluxDB 类型

需要注意，FLUX 语言里有些基本数据类型比如持续时间(Duration)和正则表达式是不能放在表流里面充当字段类型的。简单来说，Duration 类型和正则表达式类型都是 FLUX 语言特有的。有些类型是为了让 FLUX 在编写代码时更加方便，让它能够拥有更多的特性，但这并不代表这些类型能够存储到 InfluxDB 中。

5.3 4 个复合类型

5.3.1 Record (记录)

5.3.1.1 定义一个 Record

一个记录是一堆键值对的集合，其中键必须是字符串，值可以是任意类型，在键上没有空白字符的前提下，键上的双引号可以省略。

在语法上，record 需要使用 {} 声明，键值对之间使用英文逗号 (,) 分开。另外，一个 Record 的内容可以为空，也就是里面没有键值对。

示例：0

```
{foo: "bar", baz: 123.4, quz: -2}  
  
{"Company Name": "ACME", "Street Address": "123 Main St.", id:  
1123445}
```

5.3.1.2 从 record 中取值

1) 点表示法 取值

如果 key 中没有空白字符，那么你可以使用 .key 的方式从 record 中取值。

示例：

```
c = {name: "John Doe", address: "123 Main St.", id: 1123445}

c.name
// Returns John Doe

c.id
// Returns 1123445
```

2) 中括号方式取值

可以使用[" "]的方式取值，当 key 中有空白字符的时候，也只能用这种方式来取值。

```
c = {"Company Name": "ACME", "Street Address": "123 Main St.", id: 1123445}

c["Company Name"]
// Returns ACME

c["id"]
// Returns 1123445
```

5.3.1.3 嵌套与链式取值

Record 类型可以进行嵌套引用。

从嵌套的 Record 中引用值的时候可以采用链式调用的方式。链式调用时，点表示法和中括号还可以混用。

```
customer =
{
    name: "John Doe",
    address: {
        street: "123 Main St.",
        city: "Pleasantville",
        state: "New York"
    }
}

customer.address.street
// Returns 123 Main St.

customer["address"]["city"]
// Returns Pleasantville

customer["address"].state
// Returns New York
```

5.3.1.4 record 的 key 是静态的

record 类型变量中的 key 是静态的，一旦声明，其中的 key 就被定死了。一旦你访问这

一个 record 中一个没有的 key，就会直接抛出异常。正常的话应该返回 null。

```
o = {foo: "bar", baz: 123.4}

o.key
// Error: type error: record is missing label haha
// 错误：类型错误：record 找不到 haha 这个标签
```

5.3.1.5 操作 records

1) 拓展一个 record

使用 with 操作符可以拓展一个 record，当原始的 record 中有这个 key 时，原先 record 的值会被覆盖；如果原先的 record 中没有制定的 key，那么会将旧 record 中的所有元素和 with 中指定的元素复制到一个新的 record 中。

示例：覆盖原先的值，并添加一个 key 为 pet，value 为"Spot"的元素。

```
c = {name: "John Doe", id: 1123445}

{c with name: "Xiao Ming", pet: "Spot"}
// Returns {id: 1123445, name: Xiao Ming, pet: Spot}
```

```
c = {name: "John Doe", id: 1123445}
a = {c with name: "Xiao Ming", pet: "Spot"}
```



5.3.1.6 列出一个 record 中所有的 keys

(1) 导入 experimental (实验的) 包。

(2) 使用 experimental.objectKeys(o:c)方法来拿到一个 record 的所有 key。

示例：

```
import "experimental"

c = {name: "John Doe", id: 1123445}

experimental.objectKeys(o: c)
// Returns [name, id]
```

5.3.1.7 比较两个 record 是否相等

可以使用双等号`=`来判断两个 record 是否相等。如果两个 record 的每个 key，每个

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

key 对应的 value 和类型都相同，那么两个 record 就相等。

示例：

```
{id: 1, msg: "hello"} == {id: 1, msg: "goodbye"}  
// Returns false  
  
{foo: 12300.0, bar: 34500.0} == {bar: float(v: "3.45e+04"), foo:  
float(v: "1.23e+04")}  
// Returns true
```

5.3.1.8 将 record 转为字符串

使用 display() 函数可以将 record 转为字符串。

示例：

```
x = {a: 1, b: 2, c: 3}  
  
display(v: x)  
  
// Returns "{a: 1, b: 2, c: 3}"
```

5.3.1.9 嵌套 Record 的意义

注意，嵌套的 Record 无法放到 FLUX 语言返回的表流中，这个时候会发生类型错误，它会说 Record 类型不能充当某一列的类型。那 FLUX 为什么还支持对 Record 进行嵌套使用呢？

其实这是为了一些网络通讯的功能来服务，在 FLUX 语言中我们有一个 http 库。借助这个函数库，我们可以向外发送 http post 请求，而这种时候我们就有可能要发送嵌套的 json。细心的同学可能发现，我们的 record 在语法层面上和 json 语法是统一的，而且 FLUX 语言提供了一个 json 函数库，借助这个库中的 encode 函数，我们可以轻易地将一个 record 转为 json 字符串然后发送出去。

5.3.2 Array (数组)

5.3.2.1 定义一个 Array

数据是一个由相同类型的值构成的有序序列。

在语法上，数组是用方括号[]起来的一堆同类型元素，元素之间用英文逗号(,)分隔，并且类型必须相同。

示例：

```
["1st", "2nd", "3rd"]  
  
[1.23, 4.56, 7.89]  
  
[10, 25, -15]
```

5.3.2.2 从 Array 中取值

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

可以使用中括号 [] 加索引的方式从数组中取值，数组索引从 0 开始。

示例：

```
arr = ["one", "two", "three"]  
  
arr[0]  
// Returns one  
  
arr[2]  
// Returns two
```

5.3.2.3 遍历一个数组

5.3.2.4 检查一个数组中是否包含某元素

示例

使用 contains()函数可以检查一个数组中是否包含某个元素。

```
names = ["John", "Jane", "Joe", "Sam"]  
  
contains(value: "Joe", set: names)  
// Returns true
```

5.3.3 Dictionary (字典)

5.3.3.1 定义一个字典

字典和记录很像，但是 key-value 上的要求有所不同。

一个字典是一堆键值对的集合，其中所有键的类型必须相同，且所有值的类型必须相同。

在语法上， dictionary 需要使用方括号[]声明，键的后面跟冒号（:）键值对之间需要使用英文逗号（，）分隔。

示例：

```
[0: "Sun", 1: "Mon", 2: "Tue"]  
  
["red": "#FF0000", "green": "#00FF00", "blue": "#0000FF"]  
  
[1.0: {stable: 12, latest: 12}, 1.1: {stable: 3, latest: 15}]
```

5.3.3.2 引用字典中的值

(1) 导入 dict 包

(2) 使用 dict.get()并提供下述参数：

- a) **dict:** 要取值的字典
- b) **key:** 要用到的 key
- c) **default:** 默认值，如果对应的 key 不存在就返回该值

示例：

```
import "dict"

positions =
[
    "Manager": "Jane Doe",
    "Asst. Manager": "Jack Smith",
    "Clerk": "John Doe",
]

dict.get(dict: positions, key: "Manager", default: "Unknown
position")
// Returns Jane Doe

dict.get(dict: positions, key: "Teller", default: "Unknown
position")
// Returns Unknown position
```

5.3.3.3 从列表创建字典

- (1) 导入 dict 包
- (2) 使用 dict.fromList()函数从一个由 records 组成的数组中创建字典。其中，数组中的每个 record 必须是{key:xxx,value:xxx}形式

示例：

```
import "dict"

list = [{key: "k1", value: "v1"}, {key: "k2", value: "v2"}]

dict.fromList(pairs: list)
// Returns [k1: v1, k2: v2]
```

5.3.3.4 向字典中插入键值对

- (1) 导入 dict 包
- (2) 使用 dict.insert()函数添加一个新的键值对，如果 key 早就存在，那么就会覆盖这个 key 对应的 value。

示例：

```
import "dict"

exampleDict = ["k1": "v1", "k2": "v2"]

dict.insert(dict: exampleDict, key: "k3", value: "v3")
// Returns [k1: v1, k2: v2, k3: v3]
```

5.3.3.5 从字典中移除键值对

- (1) 引入 dict 包
- (2) 使用 dict.remove 方法从字典中删除一个键值对

示例：

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
import "dict"

exampleDict = ["k1": "v1", "k2": "v2"]

dict.remove(dict: exampleDict, key: "k2")
// Returns [k1: v1]
```

5.3.4 function (函数)

5.3.4.1 声明一个函数

一个函数是使用一组参数来执行操作的代码块。函数可以是命名的，也可以是匿名的。在小括号()中声明参数，并使用箭头=>将参数传递到代码块中。

示例：

```
square = (n) => n * n

square(n:3)
// Returns 9
```

FLUX 不支持位置参数。调用函数时，必须显示指定参数名称。

5.3.4.2 为函数提供默认值

我们可以为某些函数指定默认值，如果为函数指定了默认值，也就意味着在调用这个函数时，有默认值的函数时非必须的。

示例：

```
chengfa = (a,b=100) => a * b

chengfa(a:3)
// Returns 300
```

5.4 函数包

Flux 的标准库使用包组织起来的。包将 Flux 的众多函数分门别类，默认情况下加载 universe 包，这个包中的函数可以不用 import 直接使用。其他包的函数需要在你的 Flux 脚本中先用 import 语法导入一下。

示例：

```
import "array"
import "math"
import "influxdata/influxdb/sample"
```

但是，截至目前，虽然你可以自定义函数，但是你无法自定义包。如果希望将自己的自定义函数封装在一个包里以供复用，那就必须从源码上进行修改。

第6章 如何使用 FLUX 语言的文档

6.1 如何查看函数文档

这是 FLUX 语言的文档 <https://docs.influxdata.com/flux/v0.x/> , 通常来说我们使用 FLUX 的文档主要是用它来查看一些函数怎么用, 如图所示:

Flux documentation
这些很少用

Flux is an open source functional data scripting language designed for querying, analyzing, and acting on data. Flux supports multiple data source types, including:

- Time series databases (such as InfluxDB)
- Relational SQL databases (such as MySQL and PostgreSQL)
- CSV

Flux unifies code for querying, processing, writing, and acting on data into a single syntax. The language is designed to be usable, readable, flexible, composable, testable, contributable, and shareable.

[Get started with Flux](#)

这些更重要

点击 Standard libaray, 就可以看到 FLUX 的所有函数包了。效果如下图所示:

universe package >

The `universe` package provides options and primitive functions that are loaded into the Flux runtime by default and do not require an import statement.

array package >

The `array` package provides functions for manipulating array and building tables from arrays.

bitwise package >

The `bitwise` package provides functions for performing bitwise operations on integers.

contrib package >

The `contrib` package contains packages and functions contributed and maintained by members of the Flux and InfluxDB communities.

csv package >

The `csv` package provides tools for working with data in annotated CSV format.

点击一个包的左侧的+按钮, 就可以看到这个包里的所有函数, 任意点击其中一个, 就可以看到这个函数的详细说明, 包括会返回什么, 调用的时候需要传递什么参数等等。

requests.do() function

The returned response contains the following properties:

- statusCode: HTTP status code returned from the request.
- body: Contents of the request. A maximum size of 100MB will be read from the response body.
- headers: Headers present on the response.
- duration: Duration of request.

Function type signature

```
(
  method: string,
  url: string,
  ?body: bytes,
  ?config: {A with timeout: duration, insecureSkipVerify: bool},
  ?headers: [string:string],
  ?params: [string:[string]],
) => {statusCode: int, headers: [string:string], duration: duration, body: bytes}
```

For more information, see [Function type signatures](#).

Parameters

再往下拉，你还可以看到每个函数都有很详细的使用示例。代码基本上是可以拿来改就用的。

Make a GET request

```
import "http/requests"

response = requests.do(url: "http://example.com", method: "GET")

requests.peek(response: response)
```

Make a GET request that needs authorization

```
import "http/requests"
import "influxdata/influxdb/secrets"

token = secrets.get(key: "TOKEN")

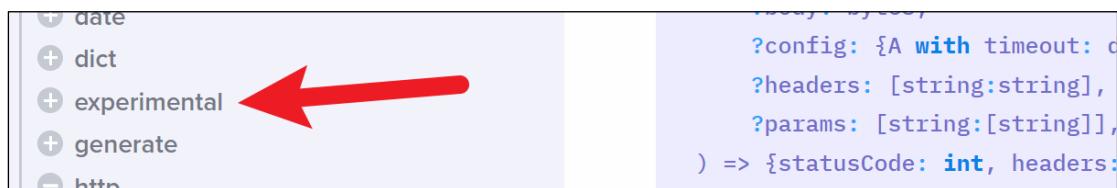
response =
  requests.do(
    method: "GET",
    url: "http://example.com",
    headers: ["Authorization": "token ${token}"],
  )

requests.peek(response: response)
```

6.2 避免使用实验中的函数

另外，需要额外注意有一个函数库的名字叫 `experimental`，这个单词是实验的意思，也就是在未来的 FLUX 版本中，这个函数有可能会变，参数名可能也不是很确定，甚至这个

函数可能会在未来的某个版本被放弃。



如果你有升级的打算，那么 experimental 里面的函数应该敬而远之，否则在未来的某个时间，很有可能会导致重复开发。

6.3 查看函数可以在哪些版本中使用

另外需要注意，每个函数的文档标题正下方都会标记这个函数是从哪个 FLUX 版本开始加入的。比如从下图我们就可以知道 request.do() 函数是从 0.173 之后才能用的。

The screenshot shows the 'requests.do()' function documentation. At the top, it says 'request.do() function'. Below that, it indicates the function was added in 'Flux 0.173.0+'. There is also a link to 'View InfluxDB support'.

下面这张图告诉我们 array.concat() 函数从 0.173 版本之后就不能再用了。

The screenshot shows the 'array.concat()' function documentation. At the top, it says 'array.concat() function'. Below that, it indicates the function was available from 'Flux 0.155.0 – 0.173.0' and links to 'View InfluxDB support'. A red box highlights a note: 'array.concat() is experimental and subject to change at any time.'

第7章 FLUX 查询 InfluxDB

7.1 前言

本章内容强烈建议跟随视频课学习

7.2 FLUX 查询 InfluxDB 的语法

使用 FLUX 语言查询 InfluxDB，必须以 from -> range 打头。

例如：

```
from(bucket: "test_init")
|> range(start: -1h)
```

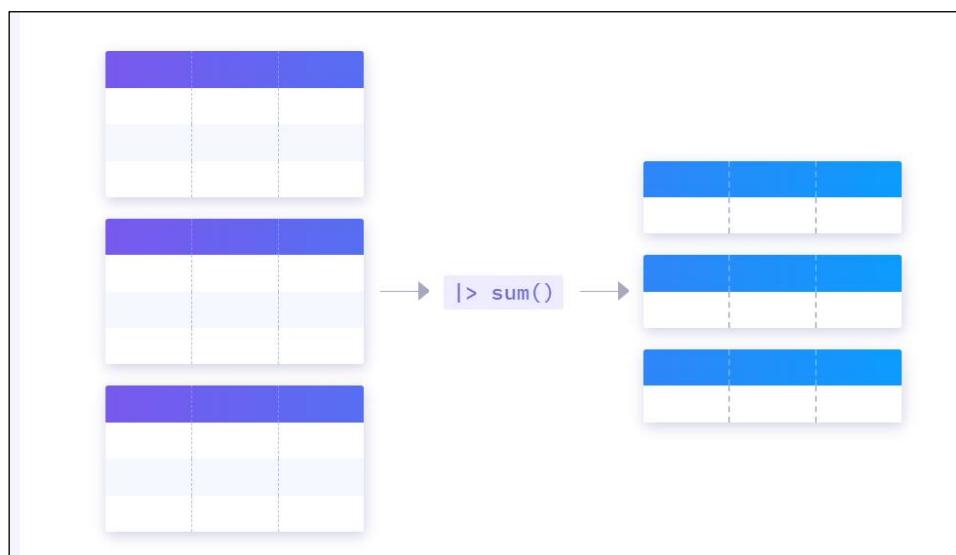
range 必须紧跟在 from 后面，不这么写的话会直接报错。

7.3 表、表流以及序列

我们知道 InfluxDB 是使用序列的方式去管理数据的。而 FLUX 语言又企图兼容一些关系型数据库的查询，而关系型数据库里的数据结构就是一个有行有列的 table。因此对于 FLUX 语言来说，就需要将序列和表统一成一个东西。

所以 FLUX 引入了表流的概念。

简单来说，FLUX 可以一次性查出多个序列，这个时候一个序列对应一张表，而表流其实就是多张表的集合。同时表流和表的关系其实是全表和子表的关系，子表是全表按照 _field, tag_set 和 measurement 进行 group by 之后的结果。在这种情况下，如果调用聚合函数，其实只会在子表中进行聚合。



最后，如果一张表对应的是一个序列了，那么一张表里的一行其实就对应着序列中的一个数据点了。

7.4 filter 维度过滤

使用 filter 函数可以对数据按照 measurement、标签集和字段级进行过滤，后面的课程会给大家讲解 filter 的性能问题。

7.5 类型转换函数与下划线字段

Flux 语言中有很多不用指定字段名称的管道函数，比如 `toInt()`。其实 `toInt()`这个函数默认要求你的字段中必须要有 `_value` 字段，没有 `_value` 字段的话也会直接报错。

其实在我们查询出来的数据中，以下划线开头的字段其实代表了一种约定，就是 FLUX 中有很多函数想要正常运行时要依赖于这些下划线打头的字段的。

所以原则上来说，程序员应该遵守这些约定，不要擅自更改下划线开头的字段。

7.6 map 函数

map 函数的作用是遍历表流中的每一条数据。

示例：

```
import "array"

array.from(rows: [{"name": "tony"}, {"name": "jack"}])
|> map(fn: (r) => {
    return if r["name"] == "tony" then {"_name": "tony 不是
jack"} else {"_name": "jack 不是 tony"}
})
```

这里需要注意，map 函数需要我们传递一个参数 fn，它要求传递一个单个参数输入，且输出必须是 record 的函数，其中输入数据的类型会是 record。

7.7 自定义管道函数

此处，我们定义一个管道函数，它可以将表流中的_value 字段的值乘上 x 倍。请同学们在接下来的示例中注意声明管道函数时所用的语法。

```
big100 = (table=<-, x) => {
    return table
        |> map(fn: (r) => ({r with "_value": r["_value"]*x}))
```

接下来我们调用刚才声明的函数，最终整个脚本如下：

```
big100 = (table=<-, x) => {
    return table
        |> map(fn: (r) => ({r with "_value": r["_value"]*x}))}

from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> big100(x:100)
```

可以自行运行查看函数效果。

这里需要强调的是，管道函数的第一个参数必须写成 table=<-，它表示通过管道符输入进来的表流数据，需要注意，table 并不一定写成 table 但是=<-的格式绝对不能变。

7.8 在文档中区分管道函数和普通函数

再次来到函数文档。

count() function

Flux 0.7.0+ View InfluxDB support

`count()` returns the number of records in each input table.

The function counts both null and non-null records.

Empty tables

`count()` returns `0` for empty tables. To keep empty tables in your data, set the following parameters for the following functions:

Function	Parameter
<code>filter()</code>	<code>onEmpty: "keep"</code>
<code>window()</code>	<code>createEmpty: true</code>
<code>aggregateWindow()</code>	<code>createEmpty: true</code>

1.这个叫做函数签名

2.如果第一个参数是<-tables的格式那么说明这是个管道函数

当我们看到一个函数文档，它会有一个区域叫做 Function type Signature（函数签名），它表示着函数接收哪些参数以及会返回什么。最前面的小括号里的内容就是参数列表，如果参数列表的第一个参数是`<-tables: stream[A]`，那就表示它是一个可以接收表流输入的管道函数。

反之，如果没有`<-tables: stream[A]`，那么它就是一个普通函数。

7.9 window 和 aggregateWindow 函数

window 函数和 aggregateWindow 函数其实代表着 InfluxDB 中的两种开窗方式，两者不同的地方在于，window 函数会将整个表流重新分组。window 开窗后，是按照序列+窗口的方式对整个表流进行分组。但是 aggregateWindow 函数会保留原来的分组方式，这样一来，使用 aggregateWindow 函数进行开窗后的表流，仍然是按照序列的方式来分组的。

7.10 yield 和 join

当 flux 脚本中出现未被赋值给某个变量的表流时，InfluxDB 执行 FLUX 脚本时会自动帮他们在管道的最后面加上`|> yield(name: "_result")`函数，yield 函数其实是指定了我们当前这个表流是整个 FLUX 脚本最后返回的结果，而且这个结果的名字叫`"_result"`。当 FLUX 脚本中出现多个为赋值给变量的表流时，给多个表流自动补上`|>yield(name:"_result")`就会出问题了，这是因为当有多个表流后面都有`|>yield`时，其实相当于一个 FLUX 脚本会返回多个结果。但是此处要求名称是不能重复的，所以当有多个未赋值的表流时，就必须显示指定`yield(name:"xxx")`，而且名称千万不可重复。

但是，在一个 FLUX 脚本里同时返回多个结果集并不是推荐的操作，这通常会让程序的逻辑变的很奇怪，我们之所以能在同一个 FLUX 脚本里面写多次 from 函数，其实是为了方
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

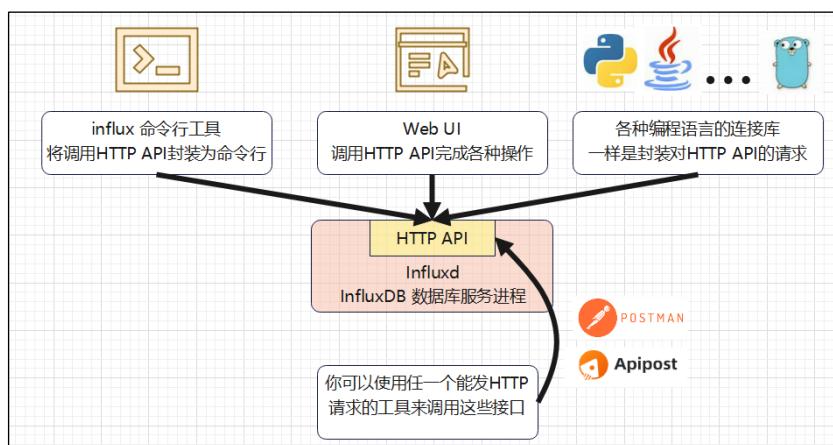
便我们进行 join 的。

再但是，老师并不建议在 FLUX 脚本中使用 join 操作，这必须要谈到 FLUX 脚本的常见使用场景，就是每隔一段时间进行一次查询。如果这个时候，我用一个 from 从 InfluxDB 中查询数据，其中有 code=01 等机器编号信息。然后我再用一个 from 去查询 mysql，得到一张机器的属性表。接下来对两张表进行 join，这在逻辑上很合理，但最大的问题就是 FLUX 脚本无法实现数据的缓存。如果我这个 FLUX 脚本是每 15 秒执行一次，那就会导致我们需要每 15 秒要去 mysql 上全表扫描一遍机器信息表，效率十分低下。

个人建议仅使用 FLUX 进行简单的查询，然后在应用层的程序里进行 join 操作。因此，本课程并不讲解 FLUX 语言的 join 操作。

第8章 前言：如何与 InfluxDB 交互

InfluxDB 启动后，会向外提供一套 HTTP API。外部程序可以也仅能通过 HTTP API 与 InfluxDB 进行通信。我们后面要讲到的 influx 命令行、Web UI 和各编程语言的客户端库，其内部都是封装的对 HTTP API 的调用。



所以各种客户端同 InfluxDB 交互时，都离不开 API TOKEN。因为 HTTP 是一种支持官方且简单的协议，这也方便了用户进行二次开发。

第9章 InfluxDB HTTP API

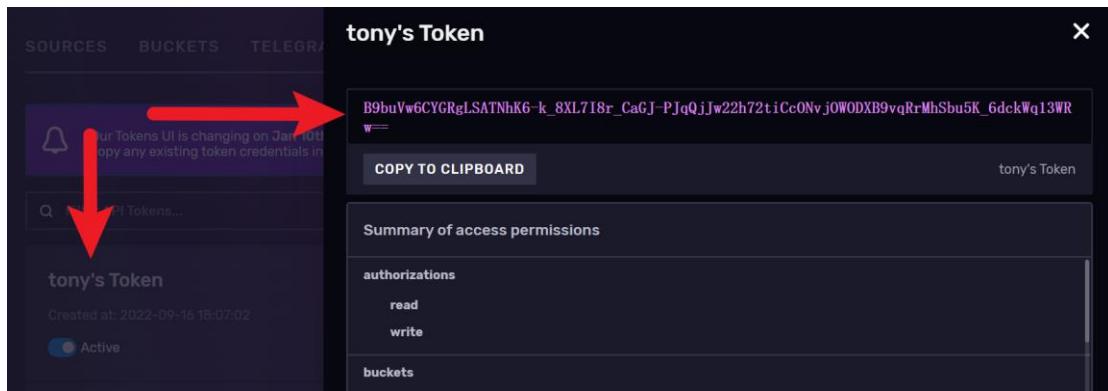
InfluxDB 提供了丰富的 API 和客户端库，可以随时和你的应用程序集成。你也可以随时使用 curl 和 Apipost、Postman 这类程序来测试 API 接口。

本课程会先带大家看一些最常用的 API，然后再告诉大家如何使用 API 文档。但本课程不会对 InfluxDB 的全部 API 进行讲解。

9.1 准备 token

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

在你想尝试使用 HTTP API 与 InfluxDB 进行交互时，首先应该用账号和密码登到 Web UI 上选择或创建一个对应的 API TOKEN。课程中，我们使用 tony's Token，这是一个具有全部权限的 API Token，实际开发时应谨慎使用，防止 Token 被劫持出现安全问题。



在后面的操作中，你每次发出 HTTP 请求时都需要在请求头上携带 token。

9.2 准备接口测试工具

在 shell 中你可以使用 curl 测试接口，不过带图形界面的程序终归是更易用一些。

本课程选用 ApiPost 这一专门的接口测试软件进行演示。ApiPost 是一款国产软件，对标的是 google 的 postman，截至视频课录制时，ApiPost 的最新版本是 6，易用性比上一个版本有大幅提升，用起来很顺手。

9.2.1 安装 ApiPost

同学们可以直接访问 ApiPost 的官网下载对应系统的安装包 <https://www.apipost.cn/>



后面请同学们自行完成软件的安装。

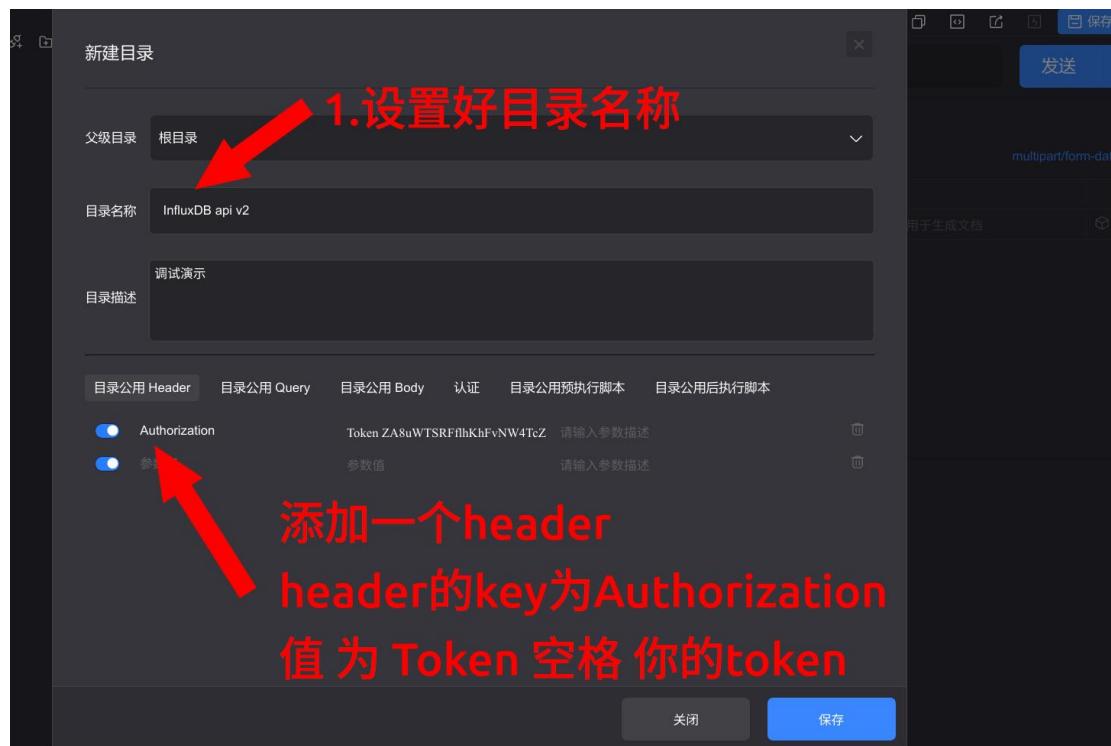
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

9.2.2 准备调试环境

(1) 在左侧的目录栏上有一个文件夹  按钮，点一下，创建一个新的目录。



(2) 给目录命名，同时为这个目录添加一个公用 header。这样这个目录下的所有接口都会自动带上这个 header，不需要我们再一个个地手动设置了。我们之前提到过，要想使用 InfluxDB 的 API，请求头上必须要加上 token。所以，我们就把 token 设为公用 header。



9.3 接口的授权

9.3.1 Token 授权方式

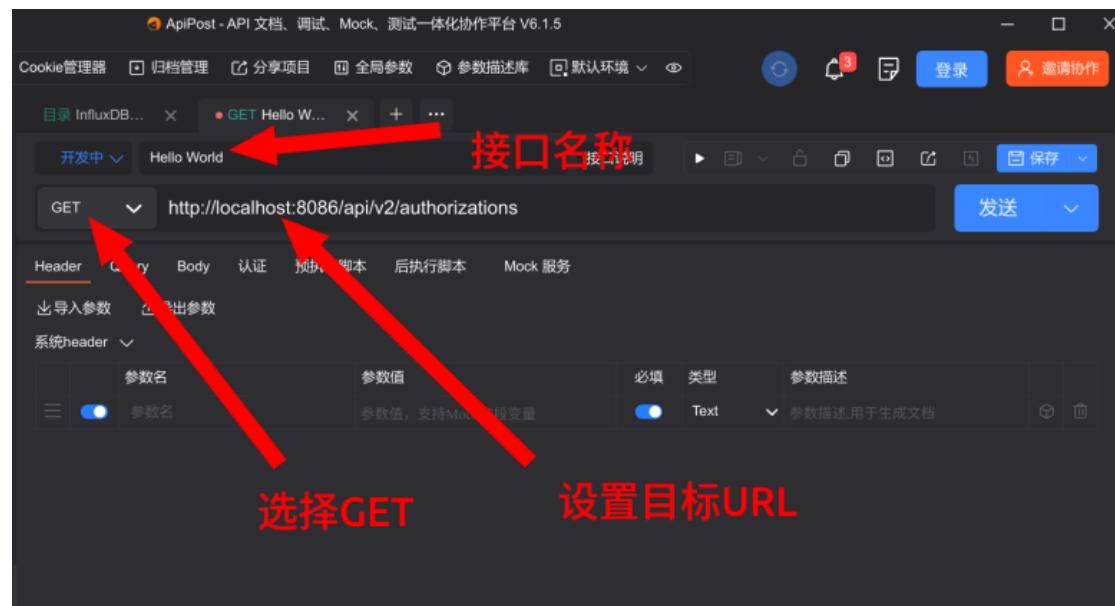
9.3.1.1 成功什么样

现在我们先来看一下授权是否是成功的。

(1) 首先，点击左侧的目录名称，右键会弹出一个菜单栏。点击新建接口



(2) 首先你可以自定义一个接口的名称，然后在接下来的 URL 栏里，填写 <http://localhost:8086/api/v2/authorizations> 点击发送。



(3) 接下来我们可以看到页面的下方弹出了返回的数据。这个接口返回的数据我们在 InfluxDB 上目前所有的 Token 信息，包括他们拥有什么权限。

```

1  {
2    "authorizations": "/api/v2/authorizations",
3    "backup": "/api/v2/backup",
4    "buckets": "/api/v2/buckets",
5    "checks": "/api/v2/checks",
6    "dashboards": "/api/v2/dashboards",
7    "delete": "/api/v2/delete",
8    "external": {
9      "statusFeed": "https://www.influxdata.com/feed/json"
10     },
11    "flags": "/api/v2/flags",
12    "labels": "/api/v2/labels",
13    "me": "/api/v2/me",
14    "notificationEndpoints": "/api/v2/notificationEndpoints",
15    "notificationRules": "/api/v2/notificationRules",
16    "orgs": "/api/v2/orgs",
17    "plugins": "/api/v2/telegraf/plugins",
18    "query": {
19      "analyze": "/api/v2/query/analyze"
20    }
21  }

```

成功看到数据，说明我们的 Token 是有效的。s

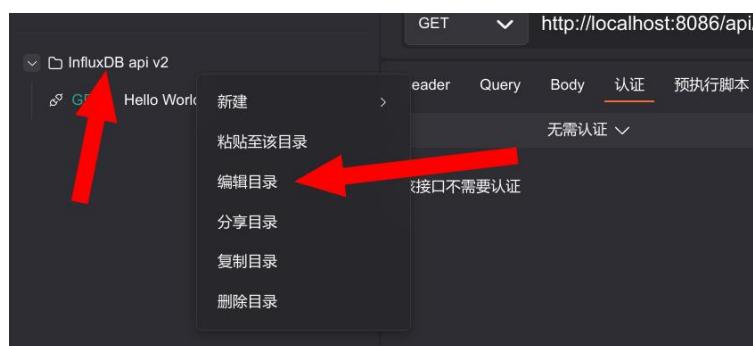
(4) 最后记得点击保存，或者使用 **Ctrl+S** 快捷键。这样，我们目录下面才会真正留下一个接口。方便你日后访问。



9.3.1.2 失败什么样

我们也可以看一下授权失败是什么效果。

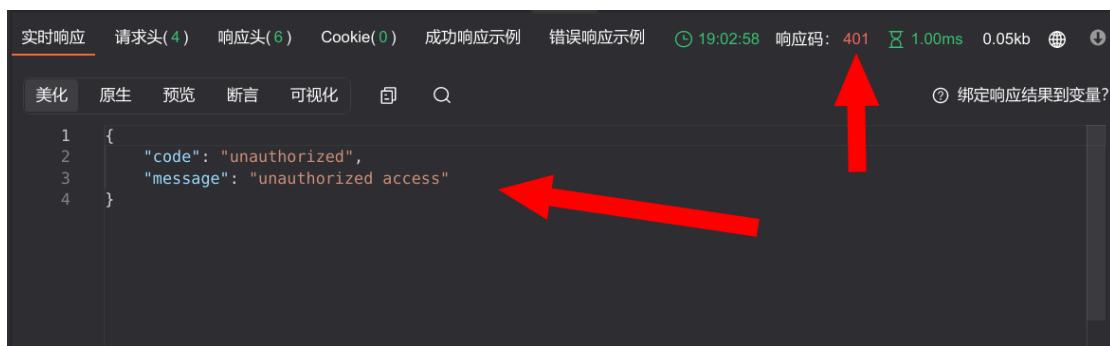
(1) 在目录上点击右键，再点击编辑目录。



(2) 将 Authorization 请求头关掉。点击右下角的保存



(3) 现在回到我们的接口调试页面上，再次点击发送。



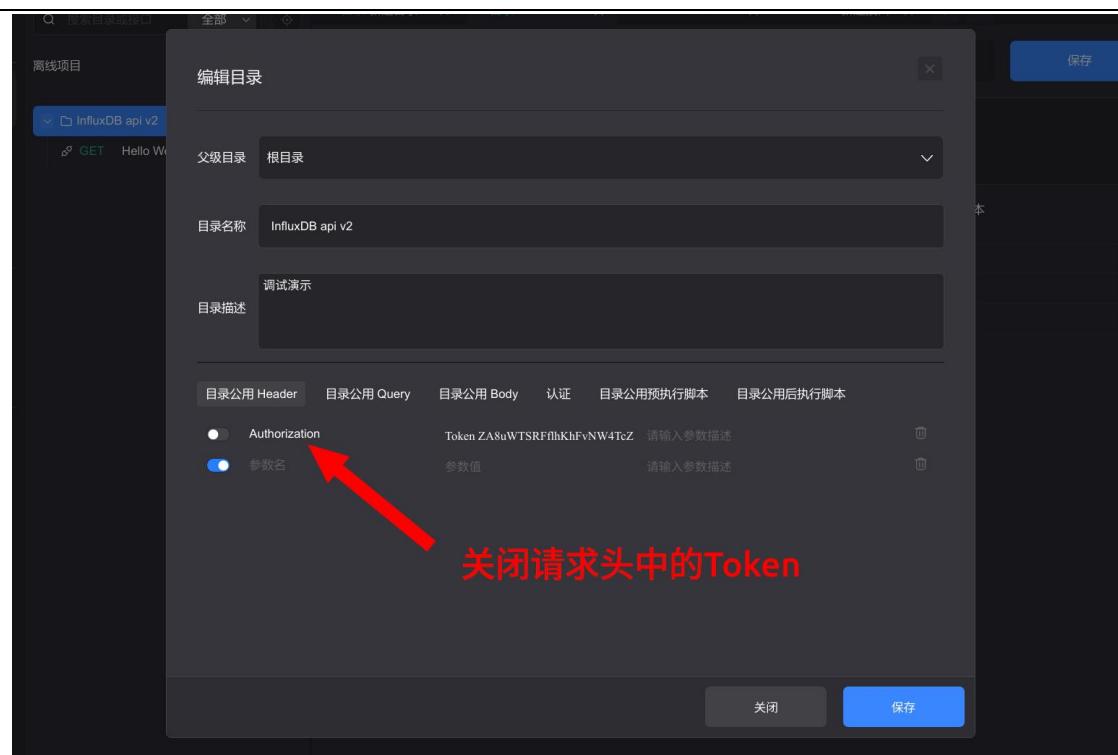
(4) 可以看到状态码为 401，而且数据的 json 告诉我们没有授权。

(5) 记得回去将目录的公共请求头放开，继续进行后面的操作。

9.3.2 登录授权方式

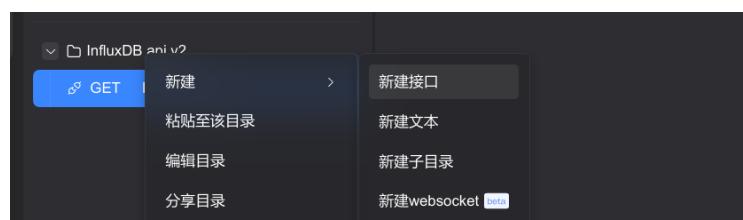
登录授权其实是留给 Web UI 用的，但是你也可以尝试用这种方式获取授权。InfluxDB 服务端会判断你的 cookie 是否合法、以及是否过期。符合要求的话就能调用接口实现一系列操作。

进行接下来的操作前，记得关闭目录下的公用请求头。



9.3.2.1 创建登录会话

(1) 在 InfluxDB api v2 目录下创建一个新的接口



(2) 给接口自定义一个名称



(3) 请求的类型选择 POST

(4) 填写目标的 URL

(5) 配置登录信息

- 在请求连接的下方点击一下“认证”按钮
- 在认证方式上选择 Basic auth 认证
- 在下方的输入框上输入 InfluxDB 的用户名和密码，课程中是 tony, 11111111。

(6) 点击发送。

9.3.2.2 登录原理

1) 什么是 Basic auth 认证

你常见的认证方式可能是将用户名和密码放到 post 请求的请求体中，再发送给服务端进行认证。不过我们刚才并没有在请求体里放用户名和密码，而是配置了一个叫 Basic auth 认证的东西。这个功能叫做 http 基本认证，是 http 协议的一部分。

基本认证的默认实现是：

- 把用户名和密码用英文冒号拼起来，也就成了 tony:11111111
- 再将拼起来的字符串用 Base64 算法编码。（tony:11111111 的 Base64 编码为 dG9ueToxMTEzMTEzMQ==）
- 给编码结果 dG9ueToxMTEzMTEzMQ==，添加一个前缀 Basic 所以最后的结果就是。

```
Basic dG9ueToxMTEzMTEzMQ==
```

(4) 把这个字符串放到一个 key 为 authorization 的请求头中，发送给服务端。

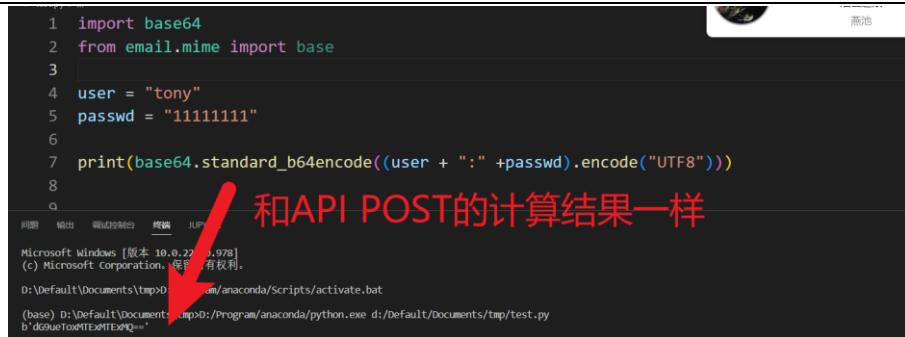
2) 查看请求头

所以，你可以在页面下方查看这次请求的请求头，如图所示，它就是我们基本认证的结晶。

The screenshot shows the Network tab of a browser's developer tools. A red arrow points from the text above to the 'Authorization' header in the list. The header value is 'Basic dG9ueToxMTEzMTEzMQ=='.

Header	Value
user-agent	ApiPOST Runtime +https://www.apipost.cn
accept	*
accept-encoding	gzip, deflate, br
connection	keep-alive
Authorization	Basic dG9ueToxMTEzMTEzMQ==
Content-Type	multipart/form-data; boundary=-----651521810871710688103718
Content-Length	-

在众多的编程语言中，base64 算法都会作为标准库的一部分存在。你可以在 python 中验证 tony:11111111 的 base64 编码结果。



```

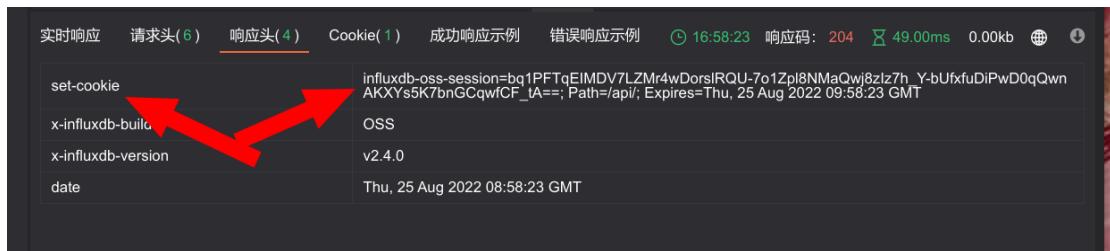
1 import base64
2 from email.mime import base
3
4 user = "tony"
5 passwd = "11111111"
6
7 print(base64.standard_b64encode((user + ":" + passwd).encode("UTF8")))
8
9

```

和 API POST 的计算结果一样

3) 查看响应头

我们还可以看一下这次请求的响应头，你可以看到响应头上有一个 key 为 set-cookie 的键值对。set-cookie 键其实会向浏览器，或者编程语言中的 Session 对象添加一个全局 cookie。



以后每次的请求就会自动携带这个 cookie，以后 InfluxDB 的接口服务就会依据这个 cookie 来判断请求方是否有权限进行相关操作。Apipost 也有记录 cookie 的功能，你可以在 Apipost 的 Cookie 管理器中，查看已经设置的全局 cookie。

4) 查看 Cookie 管理器

(1) Apipost 的 Cookie 管理器在页面的最上方。



(2) 弹出的窗口就是 Cookie 管理器。下面首先会列出你的域名或者 host，老师这里是 localhost，点一下，可以看到它下面的全部 cookie。



(3) 再点一下 influxdb-oss-seesion，就可以看到这个 cookie 的内容可，可以看到它跟刚才响应头的 set-cookie 内容一模一样。

9.3.2.3 验证授权效果

接下来，我们会到之前的列出所有 token 的接口里去，在目录共享请求头关闭的前提下，调用 api。



(1) 直接点击发送按钮

1.点击发送

2.可以看到数据出现了

```

1 {
2   "links": {
3     "self": "/api/v2/authorizations"
4   },
5   "authorizations": [
6     {
7       "id": "09dc7d49cb636000",
8       "token": "Bffs82TC-hRG58aamxHxDLPGxFAHChhb3YmxrV9UCokSmG-aHDQ0sQuNci6JywWG4W350o95Mr_oFqCraWFbA==",
9       "status": "active",
10      "description": "WRITE example01 bucket / READ Name this Configuration telegraf config",
11      "orgID": "84f8210c906f0282",
12      "org": "atguigu_com",
13      "userID": "09dc6fd81692f000",
14      "user": "atguigu",
15      "permissions": [
16        {
17          "action": "write",
18          "resource": {
19            "type": "buckets",
20            "id": "c42df67f179485c4",
21            "orgID": "84f8210c906f0282"
22          }
23        }
24      ]
25    }
26  }
27}
  
```

(2) 响应码为 200，且成功出现了数据，说明我们现在是有权限的，可以点击下面的请求头按钮，看一下这次请的请求头。

点一下，查看这次请求的请求头

这就是之前influxdb给我们设置的cookie

accept	/*
accept-encoding	gzip, deflate, br
user-agent	Apipost Runtime +https://www.apipost.cn
connection	keep-alive
cookie	influxdb-oss-session=B6RXORo9J5ZjkH78WziDnq_qtxctBZgn4kREbQZEYYVSybD6Zw9XkXRklr-pWvVnqlm_Mxh3C8mHBkMZh23dfA==

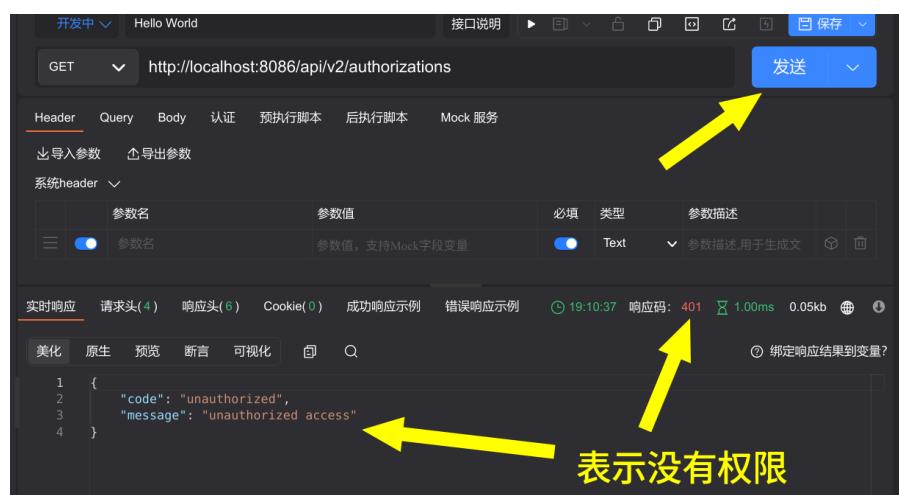
9.3.2.4 关闭全局 cookie 再查看效果

接下来我们关闭全局 cookie 再查看效果。

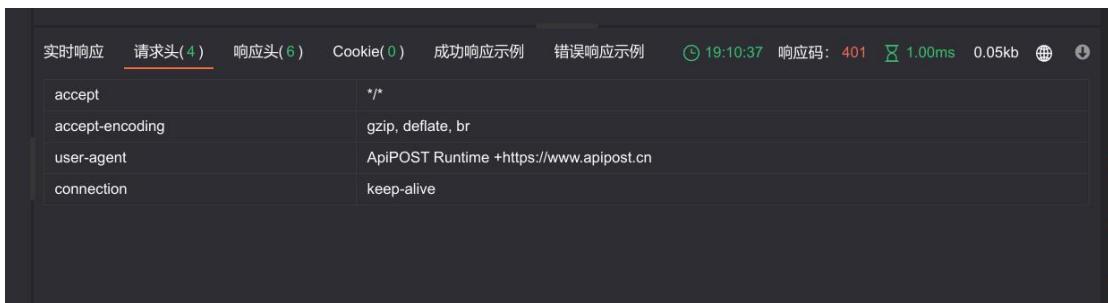
(1) 在 Apipost 中打开 Cookie 管理器，按图中操作，关闭 Cookie。



(2) 再次向 /api/v2/authorizations 发送请求。可以看到我们这次没权限了



(3) 再次查看请求头。这次我们失去了 cookie。



9.3.2.5 小结

这一节，我们学会了用登录的方式获取授权。但是，大家还要注意两点

- http 基本认证默认实现的安全问题

我们前面讲过，http 基本认证其实就是把 tony:11111111 的 Base64 编码放在的请求头上，但是 Base64 只是一种数据的编码方式，它不是加密算法也不是信息摘要算法。这也就是说，一旦我的请求被拦截，那对方就能看到我的用户名和密码，对我实施中间人攻击。

如图所示，Base64 编码的字符串也可以被解码为明文。



```

1 import base64
2 data = "YXRndWlndToxMTEzMTEzMQ=="
3 result = base64.standard_b64decode(data)
4 s = str(result,encoding="UTF-8")
5 print(s)

```

被解码为明文，用户名和密码泄露

问题 调试控制台 终端 JUPYTER

四 8月 - 19:38 ~/桌面/data

- @atguigu /bin/python3 /home/dengziqi/桌面/data/data_gen.py
atguigu:11111111

所以，从安全角度考虑，不应当在开发时将 Web UI 暴露在公网上。而且集成应用时，授权也千万不可以使用登录方式，应该全部使用 token。

- Cookie 有过期时间

当你和别的应用进行集成时，也不应该使用登录的方式，登录授予的 cookie 是有过期时间的，大概半小时，cookie 就会过期。用户必须重新登录拿到新的 cookie 才能和 InfluxDB 继续交互。

授权的内容就讲到这了。

同学们进行后面操作的时候，记得恢复目录的公用请求头，并关闭 Cookie。这样，我们就还是使用 token 授权的方式完成后面的一系列操作。

9.4 接口安全：配置 HTTPS

HTTP 是一种纯文本通信协议。早期很多互联网协议都是使用明文的方式来传输数据的。这样，最大的问题就是如果我们的网络请求被劫持，那么劫持的一方可以看到我们请求中的所有数据（包括 Token），这样就算是使用 Token 进行授权比 user:password 安全一些，但泄漏 Token 也会带来很多麻烦事。所以，InfluxDB 官方强烈建议我们开启 HTTPS。

9.4.1 使用 openssl 生成证书

下面是官方给出的命令模板。

```
sudo openssl req -x509 -nodes -newkey rsa:2048 \
-keyout /etc/ssl/influxdb-selfsigned.key \
-out /etc/ssl/influxdb-selfsigned.crt \
-days <NUMBER_OF_DAYS>
```

自己跑的时候可以参考做一下调整

命令解释：

- req -x509，指定生成自签名证书的格式。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

● -newkey rsa:2048, 生成证书请求或者自签名整数时自动生成密钥，然后生成的密钥名称由 keyout 指定。rsa:2048 意思是产生 rsa 密钥，位数是 2048。

- -keyout, 指定生成的密钥名称。
- -out, 证书的保存路径
- -days, 证书的有效期限，单位是 day (天), 默认是 365 天。

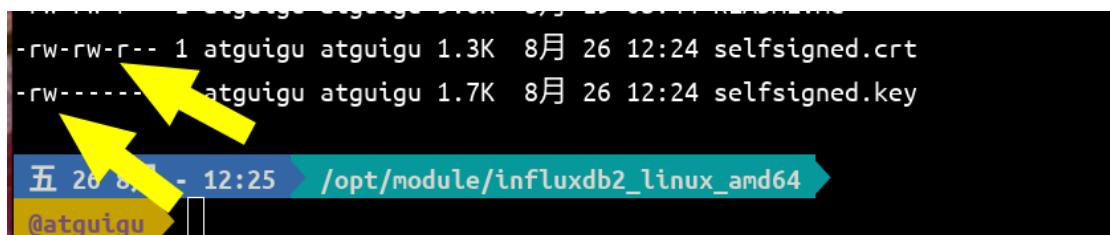
现在，我们执行下面的命令：

```
openssl req -x509 -nodes -newkey rsa:2048 \
-keyout /opt/module/influxdb2_linux_amd64/selfsigned.key \
-out /opt/module/influxdb2_linux_amd64/selfsigned.crt \
-days 60
```

执行这个命令后，会让你输入更多信息。你可以直接全部敲回车，将这些字段留空。
不影响生成我们有效的证书文件。

执行完这个命令后，/opt/module/influxdb2_linux_amd/ 目录下会产生两个文件，一个是 selfsigned.crt (证书文件) 另一个 is selfsigned.key (密钥文件)。而且他们的有效期是 60 天至此，你的密钥文件就成功生成了！

9.4.2 确保启动 influxd 的用户对密钥整数文件有读取权限



```
-rw-rw-r-- 1 atguigu atguigu 1.3K 8月 26 12:24 selfsigned.crt
-rw----- 1 atguigu atguigu 1.7K 8月 26 12:24 selfsigned.key
五 26 8月 - 12:25 > /opt/module/influxdb2_linux_amd64 >
@atguigu
```

9.4.3 启动 influxd 服务时指定证书和密钥路径

使用 influxd 命令启动 InfluxDB 服务时，记得指定一下整数的密钥的路径。

```
./influxd \
--tls-cert="/opt/module/influxdb2_linux_amd64/selfsigned.crt" \
--tls-key="/opt/module/influxdb2_linux_amd64/selfsigned.key"
```

9.4.4 验证 HTTPS 协议是否生效

回到我们的 ApiPost6，再次向 http://localhost:8086/api/v2/authorizations 发送 GET 请求。

The screenshot shows a REST client interface with a GET request to `http://localhost:8086/api/v2/authorizations`. The response status is 400, and the error message is "Client sent an HTTP request to an HTTPS server." A yellow arrow points from the text "状态码为400，我们不能使用http的方式访问了，必须改成https" to the error message.

状态码为400，我们不能使用http的方式访问了，必须改成https

可以看到，我们使用 http 的协议头再进行访问，响应的状态码为 400，并提示我们向 HTTPS 服务器发送了一个 HTTP 请求。现在我们将 URL 前面的 http 改成 https 再试一下。

The screenshot shows a REST client interface with a GET request to `https://localhost:8086/api/v2/authorizations`. The response status is 200, and the JSON data is displayed. A yellow arrow points from the text "将http换成https" to the URL. Another yellow arrow points from the text "状态码为200，拿到了正常的数据" to the response body.

将http换成https

状态码为200，拿到了正常的数据

```

1 {
2   "links": {
3     "self": "/api/v2/authorizations"
4   },
5   "authorizations": [
6     {
7       "id": "09dc7d49cb636000",
8       "token":
9         "Bffs82TC-hRG58aamxHxDLPGxFAHCHbhb3YmxrV9UCokSmG-aHDQOsQuNci6JywWG4W35Qo9SMr_oFqCraWFbA==",
10      "status": "active",
11      "description": "WRITE example01 bucket / READ Name this Configuration telegraf config",
12      "orgID": "84f8210c906f0282",
13    }
14  ]
15}
  
```

如果你也能达到这个效果，说明 influxd 的 ssl/tls 认证已经开启，服务端和客户端传递的将会是加密数据而非明文数据。

9.4.5 记得更改已存在的 telegraf 配置和 Scrappers

我们之前在 WebUI 中配置过 Telegraf 配置和指标的抓取任务，当时我们配的是 http 协议的 URL，现在也需要全部换成 https。

The screenshot shows the 'Load Data' interface with the 'SCRAPERS' tab selected. There is a search bar and a sorting dropdown. Two entries are listed:

- example03_scraping**: Bucket: example03, URL: http://localhost:8086/metrics
- new target**: Bucket: test_init, URL: http://localhost:8086/metrics

Yellow annotations highlight the 'SCRAPERS' tab and the 'example03_scraping' entry.

9.5 其他生产安全考虑

HTTPS 是 InfluxDB 开发时，最基础也是最该考虑的安全措施，除此之外 InfluxDB 在设计时还为用户考虑了其他的安全措施，这里给大家简单地介绍一下，不再进行操作演示。

9.5.1 IP 白名单

可以参考: <https://docs.influxdata.com/influxdb/v2.4/security/enable-hardening/>

这个 IP 白名单并不是限制谁可以访问我的。

而是限制，我 InfluxDB 的查询可以访问谁的。因为 FLUX 语言具有发送网络请求的能力，你可以使用 InfluxDB 的相关配置限定，FLUX 脚本可以向哪些地址发送请求。

9.5.2 机密管理

这一块的内容可以参考: <https://docs.influxdata.com/influxdb/v2.4/security/secrets/>

假如，我们的自己的应用程序和 InfluxDB 集成，而用到的一段 FLUX 脚本刚好需要使用某个第三方服务的用户名和密码（比如查询 mysql）。

比如：

```
import "influxdata/influxdb/secrets"
import "sql"

sql.from(
    driverName: "postgres",
    dataSourceName: "postgresql://tony:11111111@localhost",
    query:"SELECT * FROM example-table",
)
```

应用和 InfluxDB 服务之间走的也是 HTTP 通信，那么写在脚本中的用户名和密码是有可能泄漏的。这个时候，你可以把用户名和密码用键值的方式放到 InfluxDB 管理起来，

然后，你就可以在脚本里用 key 的方式在 InfluxDB 里获取 tony 的用户名和密码了。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

```
import "influxdata/influxdb/secrets"
import "sql"

username = secrets.get(key: "POSTGRES_USERNAME")
password = secrets.get(key: "POSTGRES_PASSWORD")

sql.from(
    driverName: "postgres",
    dataSourceName: "mysql://${username}:${password}@localhost",
    query:"SELECT * FROM example-table",
)
```

这样，我们 Mysql 的用户名和密码，就没有在网络上泄漏的风险了。

9.5.3 token 管理

可以参考：<https://docs.influxdata.com/influxdb/v2.4/security/tokens/>

我们之前讲过在 Web 上去创建 token。这里再给大家补充一下 Token 的类型。

- **操作者 Token**。操作者令牌有跨越组织的管理权限，它对 InfluxDB OSS 2.x 上的所有组织和资源有完全的读写访问权限。某些操作必须需要操作员权限（比如 查看服务器配置）。操作者 Token 是在 InfluxDB 初始化设置的过程中创建的。要想再创建一个操作者 Token，就必须使用先有的操作者 Token。

由于操作者 Token 对 InfluxDB 中所有的组织具有完全的读写访问权限。因此 InfluxDB 建议为每个组织创建一个全权限 Token，并用这些 Token 来管理 InfluxDB。这有助于防止组织间不小心误操作对方资源。

- **全权限 Token**。对单个组织中所有资源的完全读取和写入访问权限
- **读/写 Token**。对组织中特定的存储桶进行读取和写入。

9.5.4 禁用部分开发功能

可以参考：<https://docs.influxdata.com/influxdb/v2.4/security/disable-devel/>

InfluxDB 的 API 中，有一部分是为了方便外部系统去监控和观测 InfluxDB 的状态和性能的。如果你觉得这部分可能影响安全，那么你可以随时把它们禁了。

比如：

- /metrics，上文给大家演示过，这里面有各种监控 InfluxDB 运行的指标
- Web UI，用户的图形界面交互。
- /debug/pprof，这个接口里面是 Go 语言程序的运行时指标，比如堆内存用了多少，有多少线程数等等。

9.6 如何使用 API 文档

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

9.6.1 查看 API 文档

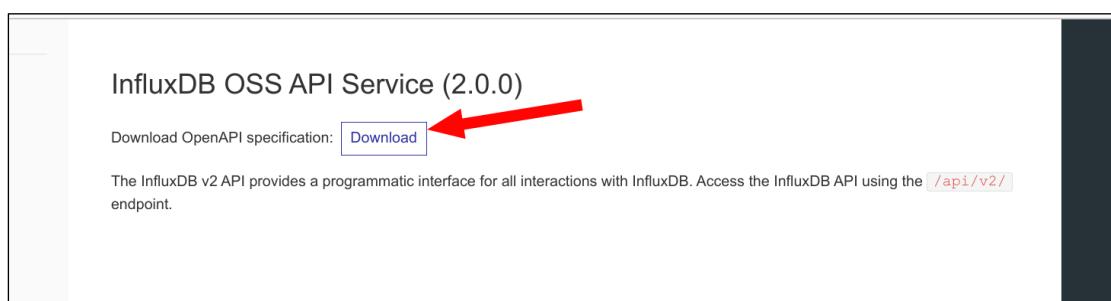
- 可以直接在浏览器上访问 `http(s)://localhost:8086/docs`，这样可以直接看到对应当前 InfluxDB 版本对应的 API 文档。

- 另外也可以在 InfluxDB 官网上查看在线文档。

<https://docs.influxdata.com/influxdb/v2.4/api/>

9.6.2 测试工具与 OpenAPI

如果你访问的是本地部署的 InfluxDB，那么访问 <http://localhost:8086> 还能下载相应的 OpenAPI 文档。



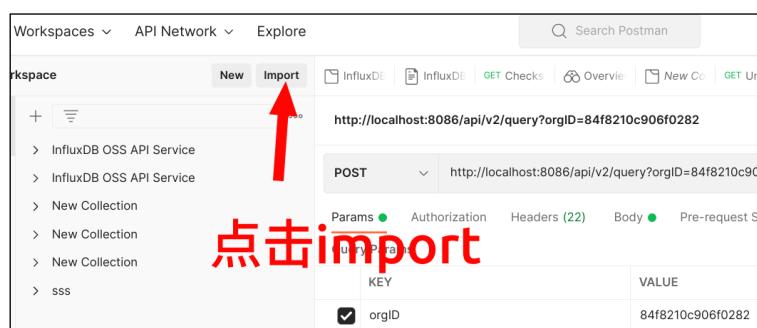
访问 <http://localhost:8086/doc> 页面的顶部有一个 Download 按钮，点一下。浏览器里会说下载了一个 json 文件。



可以打开看一下，这其实是一个符合 OpenApi3.0 格式的 API 文档定义文件。现在的 ApiPost 和 Postman 对这一格式都能自动生成接口测试。此处我们拿 postman 作为演示。

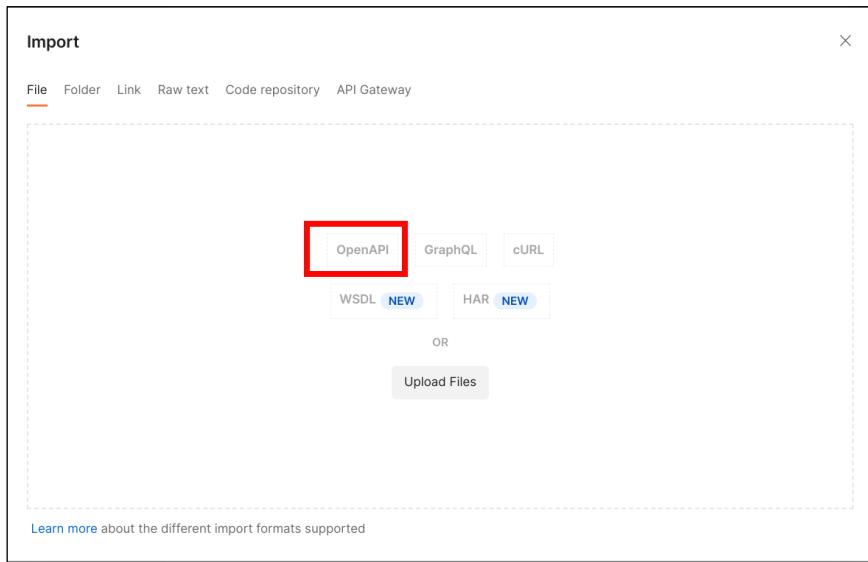
9.6.3 示例：Postman 快速生成测试项目

9.6.3.1 使用 postman 导入 openapi

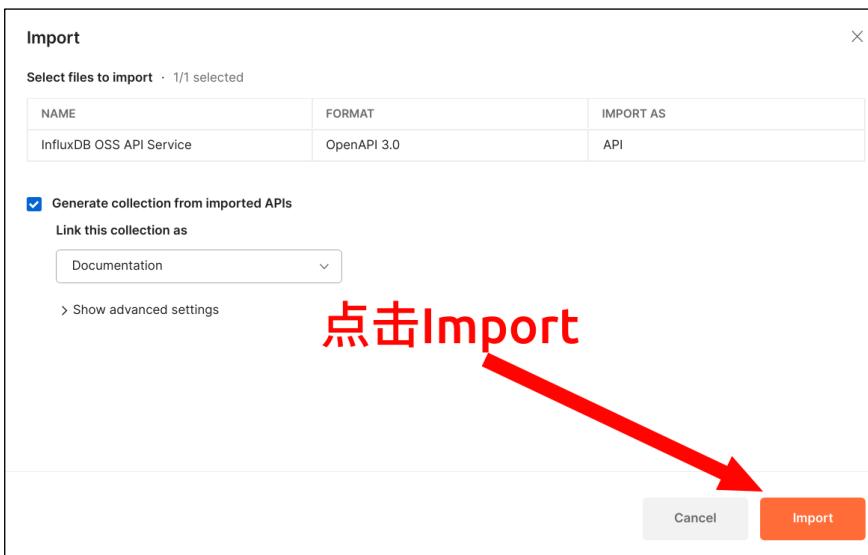


在 postman 的左侧目录上方，有一个 import 按钮，点一下，会弹出一个对话框。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

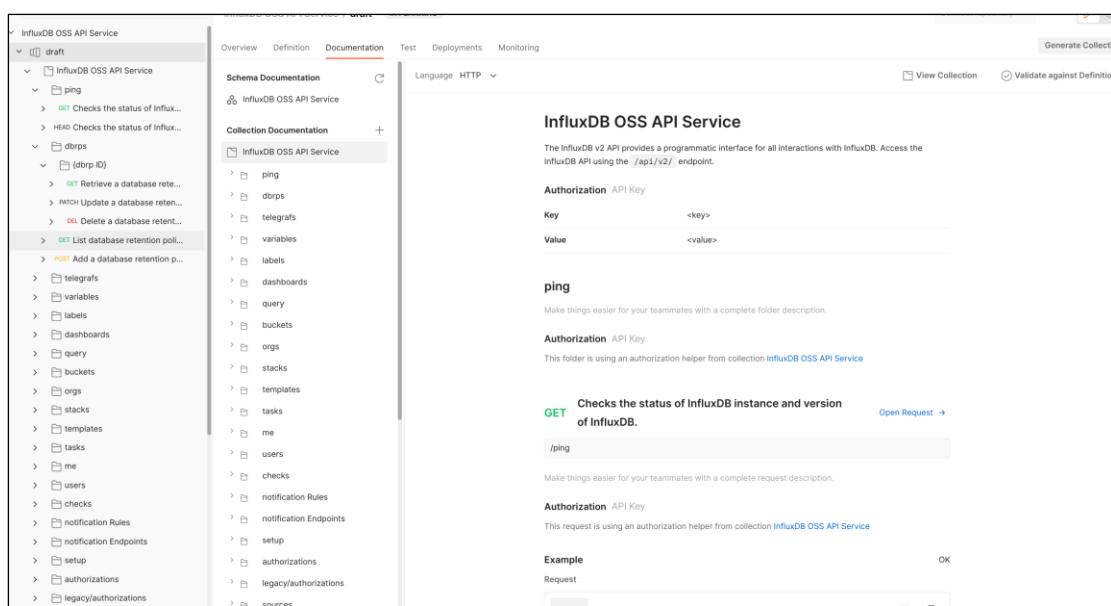


可以看到，这里说支持上传 OpenApi 格式的文件。点击 Upload Files 按钮，选择刚才下载的 swagger(2).json。最后点击右下角的 Import 按钮。



9.6.3.2 查看导入效果

可以看到，InfluxDB 中的全部 API 已经导入到 postman 中了。但是这里没有文档中的说明性文字了。找回他们的方法是在 postman 的左边找到 draft，点击一下，再点击右方的 Documentation。如下图所示：



现在，你就可以看到既能阅读，又能立刻进行测试的 API 文档了。

第10章 使用 influx 命令行工具

从 InfluxDB 2.1 版本之后，influx 命令行和 InfluxDB 数据库服务程序是分开打包的，所以安装 InfluxDB 并不会附带 influx 命令行工具。用户必须单独安装 influx 命令行工具。

influx 命令行工具包含很多管理 influxDB 的命令。包括存储桶、组织、用户、任务等。

从 2.1 版本之后，安装 InfluxDB 不会附带 influx 命令行工具，现在 influx 工具和 InfluxDB 在源码上也已经分开维护了，下载时需要注意对上版本。

10.1 安装 influx 命令行工具

这次，我们另辟蹊径，不看着官方文档安装了，改从 [github](#) 上下载安装。

10.1.1 如何去找开源项目的发行版

10.1.1.1 什么是发行版

这一部分的内容可以详细参考 Gitee 官方文档 <https://gitee.com/help/articles/4328#article-header0>

所谓发行，就是这个开源项目进行到一定程度，各种特性和功能已经趋于完善和稳定，到了可以出一个阶段性版本的时候了。

通常来说，github 或者 gitee 上放的是一个项目的源码，但是源码需要经过编译之后才能运行的，那么当作者觉得自己的项目，目前开发进度差不多，应该没什么坑的时候，他就可以自己创建一个发行版。这个时候，作者需要自己上传一些附件，比如 v1.0.0 的编译

后程序，v1.0.0 的文档和源码等。

规范的发行信息里面应该还有比如 changelog（修改记录）这些信息，告诉用户，这个版本相比上个版本，增加了哪些新的功能，又修复了哪些 bug。

10.1.1.2 如何去找一个项目的发行版

首先，你可以去访问官网，通常来说一个开源项目通常应该有它自己的官网，在它的官网上，应该可以找到它的历史版本。但是，有些官网就是新版发布了之后就下架旧版的下载资源，比如 InfluxDB 就是这么干的。

另外，通常开源项目都会在 github 或者 gitee 上去维护一个版本的时间线。

打开你关注的开源项目首页，如图所示是 InfluxDB 的项目首页。

这就是发行版的信息

点一下蓝色的
releases 可以看到
历史上所有的发行版

点击右下角的 Release。

v2.4.0 [Latest]

In addition to the list of changes below, please also see the [official release notes](#) for other important information about this release.

v2.4.0 [2022-08-18]

Bug Fixes

- 1. [21885a7](#): Log the log level at startup
- 2. [76cfdb](#): Emit zipfile for windows
- 3. [69a95dc](#): Update the condition when resetting cursor
- 4. [4789d54](#): Improve error messages opening index partitions
- 5. [00edb77](#): Create TSI MANIFEST files atomically

可以看到这个框架从盘古开天辟地至今的所有发行版。通常，在一个版本记录的最下方，会有这个版本对应的已编译好的可执行程序和源码。你可以把它下载下来使用。

OSS BINARY FILES	SHA256
influxdb2-2.4.0-darwin-amd64.tar.gz	156c52b6999134284abae7b110d25416740414c56702c29fd8efe00b6be98682
influxdb2-2.4.0-linux-arm64.tar.gz	4fde9265af5ab9a2b8fb10bb5df42bd809fd24af352cd7eca0ee9ecf4fdb3b9
influxdb2-2.4.0-windows-amd64.zip	07769bd83e5f211490e1d3dac3fc0cb674a25edc860137f5df444856308fa4e
influxdb2-2.4.0-linux-amd64.tar.gz	51ddd49a482490752647a4f134c3c838adab64903f2a6ed9c90daff8418b7c58

OSS UBUNTU & DEBIAN PACKAGE FILES	SHA256
influxdb2-2.4.0-arm64.deb	f0579ede760b2b65b327b95a6fb4bcfeab4cb06fa26aa961fb9810c3dafdf620
influxdb2-2.4.0-amd64.deb	7de3ccf9672d259a82e79d629397913512045a4dba1e2cb1ef98e8887d2da599

OSS REDHAT & CENTOS PACKAGE FILES	SHA256
influxdb2-2.4.0.x86_64.rpm	02b3e1843fe232c2944e23322be228530bb752d50bdfc22abd5b2201ad12b6fc
influxdb2-2.4.0.aarch64.rpm	25b30d342a0ae9e275d25a23070115363ddd6e1412934c28aa9d2085e26833da

▼ Assets 2

Source code (zip)	8 days ago
Source code (tar.gz)	8 days ago

10.1.2 去找 influx 命令行工具的开源项目

大多数时候，你会在 `github` 上通过搜索项目名称的方式来从查找你关注的项目。但是如果项目本身的热度不高，那它可能不会出现在搜索结果的第一页里。最后你要向后翻好久才能找到你的项目。

当前 InfluxDB 的热度还算行，但是它周围对应的工具热度就不一定高了。这个时候，你可以将目光聚焦于单个公司下的所有项目。

点击查看所有项目

找到 influx-cli 项目，打开之。<https://github.com/influxdata/influx-cli>

About

- CLI for managing resources in InfluxDB v2
- Readme
- MIT license
- 36 stars
- 37 watching
- 10 forks

Releases 6

v2.4.0 (Latest) 7 days ago + 5 releases

10.1.3 下载安装发行版

点击 Releases 链接，看到最新的版本。

v2.4.0 (Latest)

jeffreysmith2nd released this 7 days ago v2.4.0 5c7c34f

In addition to the list of changes below, please also see the [official release notes](#) for other important information about this release.

v2.4.0 [2022-08-18]

往下看页面，找到 linux-amd64.tar.gz

11. 051abaa: Clarity difference in virtual vs physical dbrops when listing

OSS BINARY FILES	SHA256
influxdb2-client-2.4.0-darwin-amd64.tar.gz	bf36c892cd85f4d16d3b94f204cd90df5feb52ab6fa0dd3b7c831720c58ed87f
influxdb2-client-2.4.0-linux-arm64.tar.gz	ccfb6695f82fd14a891fda47fbdb02b2bc8cd67a50976817b1344e6f5eff843
influxdb2-client-2.4.0-windows-amd64.zip	f5575b2bc18641568868e699f2fb8efcf6070d34f8dd68d84709a228e8c0d581
influxdb2-client-2.4.0-linux-amd64.tar.gz	3c22de8823d9816eecf47c2f9620883f53cb975db6aab06e796d715c3
OSS UBUNTU & DEBIAN PACKAGE FILES	SHA256
influxdb2-client-2.4.0-arm64.deb	b5844d371d2659e535d7020ed175ef227de32b6e8be0a1cb7070c1db66390a94
influxdb2-client-2.4.0-amd64.deb	f28b86f21bf1deaf359eaa90104bd45c35d8be9f5c4ebe0f7c41a7cbd2b0872f
OSS REDHAT & CENTOS PACKAGE FILES	SHA256
influxdb2-client-2.4.0.x86_64.rpm	125fc3f00bed1177fd3a34ec32e6b5c3d826d9129f78ab61e8c5eb9fee478743
influxdb2-client-2.4.0.aarch64.rpm	18ae12522d9d7f355c934d84c7630954281b400362198b3164aa1d08140dea60

▼ Assets 2

- [Source code \(zip\)](#) 8 days ago
- [Source code \(tar.gz\)](#) 8 days ago

下载到 /opt/software/

```
五 26 8月 - 18:54 ➔ /opt/software
@atguigu ➔ ll
总用量 133M
-rw-rw-r-- 1 atguigu atguigu 88M 8月 19 03:49 influxdb2-2.4.0-linux-amd64.tar.gz
-rw-rw-r-- 1 atguigu atguigu 5.8M 8月 19 03:29 influxdb2-client-2.4.0-linux-amd64.tar.gz
-rw-rw-r-- 1 atguigu atguigu 39M 8月 17 04:25 telegraf-1.23.4_linux_amd64.tar.gz
```

解压到 /opt/module/

```
tar -zxvf influxdb2-client-2.4.0-linux-amd64.tar.gz -C
/opt/module/
```

10.2 配置 influx-cli

10.2.1 创建配置

influx 命令行工具是你每执行一次操作时，调用一次命令。并不是开启一个持续性的会话。而 influx 其实底层还是封装的对 InfluxDB 的服务进程的 http 请求。也就是它还是要配置 Token 什么的来获取授权。

所以，为了避免以后每次请求的时候都在命令行里面写一遍 token。我们应该先去搞个配置文件。

使用下面的命令可以创 influx 命令行的配置。

```
./influx config create --config-name influx.conf \
--host-url http://localhost:8086 \
--org atguigu \
--token
ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHFW1YIVlcj9Am5iJ4ueHawWh49_jszoKybEym
HqgR5mAWg4XMv4tb9TP3w== \
--active
```

- 这个命令其实会在`~/.influxdbv2`目录下创建一个`configs`文件，这个文件中，就是我们命令行中写的各项配置。如图所示：

```
田 vim configs 111x24
["influx.conf"]
  url = "http://localhost:8086"
  token = "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w=="
  org = "atguigu"
  active = true
#
```

10.2.2 更改配置

如果你中途配置错误了，再使用上文的命令，它会说这个配置已经存在。

```
五 26 8月 - 20:01  /opt/module/influxdb2-client-2.4.0-linux-amd64
@atguigu ./influx config create --config-name influx.conf \
--host-url http://localhost:8086 \
--org atguigu \
--token ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w== \
--active

Error: failed to create config "influx.conf": config "influx.conf" already exists
```

也就是说，在`/home/dengziqi/.influxdbv2/configs`文件中，`["name"]`配置快不能重复必须全局唯一。

这个时候如果你想调整配置，应该把`create`换成`update`。

也就是

```
./influx config update --config-name influx.conf xxxxxxxxx
```

10.2.3 在多份配置之间切换

我们现在用下面的命令再创建一个配置，直接复制`influx.conf`中的内容，把名字修改成`influx2.conf`

```
./influx config create --config-name influx2.conf \
--host-url http://localhost:8086 \
--org atguigu \
--token
ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEym
HqgR5mAwg4XMv4tb9TP3w== \
--active
```

命令成功执行后，再次打开`~/.influxdbv2/configs`文件。

```

["influx.conf"] ← 第一次创建的配置
  url = "http://localhost:8086"
  token = "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAkg4XMv4tb9TP3w=="
  org = "atguigu"
  previous = true ← previous=true表示这是之前的

["influx2.conf"] ← 第二次创建的配置
  url = "http://localhost:8086"
  token = "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAkg4XMv4tb9TP3w=="
  org = "atguigu"
  active = true ← active=true表示目前正在使用
#

```

可以看到 configs 中的文件内容变了，多了一个名为["influx2.conf"]的配置块，而且，旧的["influx.conf"]从 active="true" 变成了 previous="true"，同时 ["influx2.conf"] 中有一个 active="true" 的键值对。说明，如果现在使用 influx-cli 执行操作，那会直接使用 influx2.conf 配置块中的内容。

你还可以使用下面的命令切换当前正在使用的配置。

```
influx config influx.conf
```

Active	Name	URL	Org
*	influx.conf	http://localhost:8086	atguigu

再次查看 ~/.influxdbv2/configs 文件

```
vim ~/.influxdbv2/configs
```

```

["influx.conf"] ← influx.conf又变成active = true状态
  url = "http://localhost:8086"
  token = "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAkg4XMv4tb9TP3w=="
  org = "atguigu"
  active = true

["influx2.conf"]
  url = "http://localhost:8086"
  token = "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHF1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAkg4XMv4tb9TP3w=="
  org = "atguigu"
  previous = true
#
# [eu-central]

```

10.2.4 删除一个配置

influx2.conf 现在对我们来说是多余的了，现在，我们将它删除掉。

使用下面的命令删除 influx2.conf。

```
./influx config remove influx2.conf
```

```
五 26 8月 - 21:32 > /opt/module/influxdb2-client-2.4.0-linux-amd64 >
@atguigu > ./influx config remove influx2.conf
Active  Name          URL           Org      Deleted
influx2.conf  http://localhost:8086  atguigu true
```

执行后，再次查看~/.influxdbv2/config 文件

```
Phi vim ~/.influxdbv2/configs 107x24
["influx.conf"]
url = "http://localhost:8086"
token = "ZA8uWTSRFlhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w=="
org = "atguigu"
active = true
#
# [eu-central]
# url = "https://eu-central-1-1.aws.cloud2.influxdata.com"
# token = "XXX"
# org = ""
#
# [us-central]
```

可以看到，["influx2.conf"]消失了。而且，我们的 influx.conf 自动变成了 active=true。

10.3 influx-cli 命令罗列

我们已经知道 influx-cli 背后封装的是对 InfluxDB HTTP API 的请求。那么 influx-cli 有多少功能基本上就取决于它封装了多少命令，本课程不会介绍 influx-cli 的全部功能。通过下表，同学们可以一探 influx-cli 的功能。

详情可以参考：<https://docs.influxdata.com/influxdb/v2.4/reference/cli/influx/>

命令	直译	解释
apply	应用	应用一个 InfluxDB 模板
auth	认证	管理 API Token 的相关
backup	备份	备份数据（只支持 InfluxDB OSS）
bucket	桶	管理存储桶的命令
bucket-schema	桶模式	管理存储桶模式的命令（）
completion	完成	生成完成的脚本
config	配置	管理配置文件
dashboards	仪表盘	列出所有的仪表盘
delete	删除	在 InfluxDB 中删除数据点。
export	导出	将 InfluxDB 中的资源导出为模板
help	帮助	查看任何命令的帮助手册
org	组织	组织管理的命令

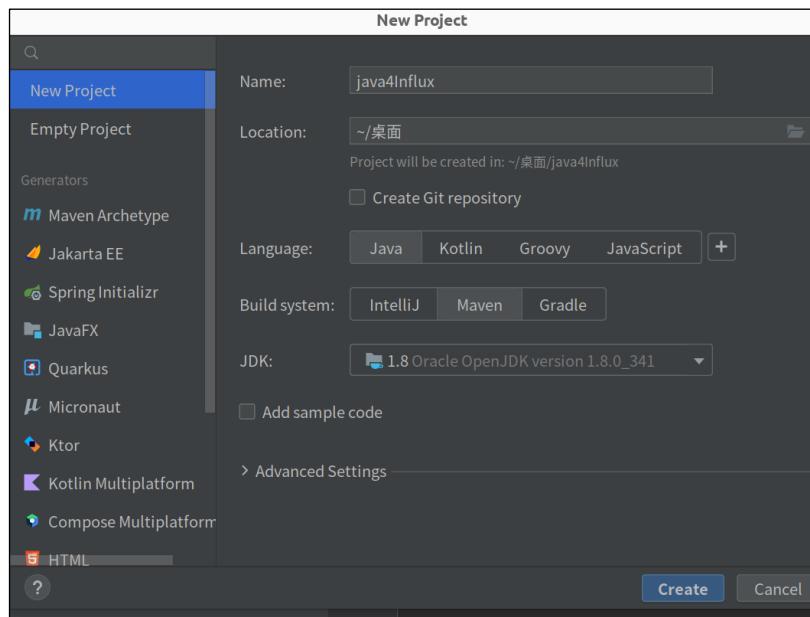
ping	乒乓	访问 InfluxDB 的/health api 作为健康状态检查
query	查询	执行一个 FLUX 脚本的查询
restore	恢复	从备份出来的数据恢复（只有 InfluxOSS 支持）
scripts	脚本	管理 InfluxDB 上的交互本(只有 InfluxDB Cloud 支持)
secret	机密	管理机密
setup	设置	命令行版本的初始化操作，首次安装 InfluxDB 时的设置用户名、密码、组织、存储桶等等。
stacks	堆栈	你可以将堆栈理解为一个 InfluxDB 模板的实例。你可以使用 stacks 管理这些实例。
task	任务	任务管理命令
telegrafs	telegrafs	管理 telegraf 配置的命令
template	模板	总结和验证 InfluxDB 模板
user	用户	管理用户相关的命令
v1	版本 1	使用 InfluxDB V1 兼容的 API
version	版本	打印 influx-cli 的版本
write	写	向 InfluxDB 写数据

第11章 JAVA 操作 InfluxDB

InfluxDB 客户端可以参考：<https://github.com/influxdata/influxdb-client-java>

11.1 创建一个 maven 项目

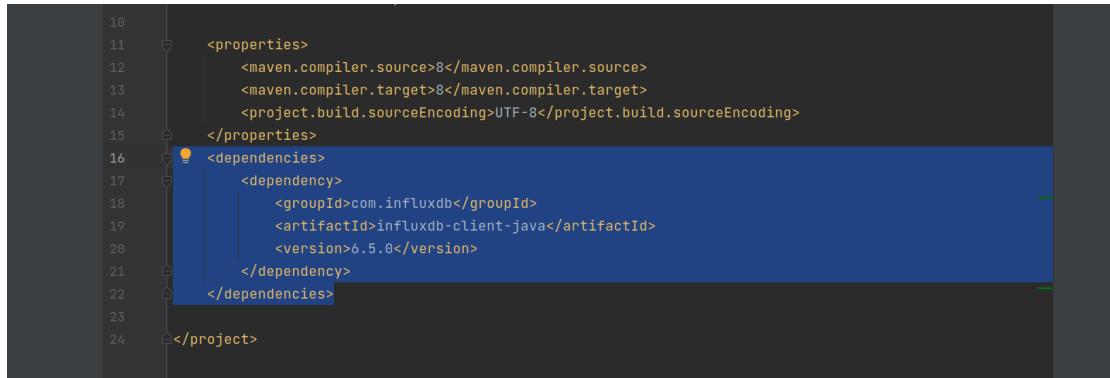
这里我创建了一个名为 java4influx 的 maven 项目



更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

11.2 导入 maven 依赖

在 pom.xml 里加入如下依赖。



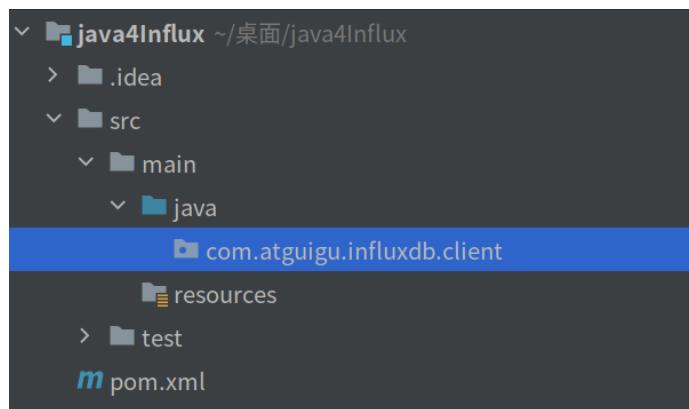
```
10<properties>
11    <maven.compiler.source>8</maven.compiler.source>
12    <maven.compiler.target>8</maven.compiler.target>
13    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14</properties>
15<dependencies>
16    <dependency>
17        <groupId>com.influxdb</groupId>
18        <artifactId>influxdb-client-java</artifactId>
19        <version>6.5.0</version>
20    </dependency>
21</dependencies>
22</project>
```

```
<dependencies>
<dependency>
    <groupId>com.influxdb</groupId>
    <artifactId>influxdb-client-java</artifactId>
    <version>6.5.0</version>
</dependency>
</dependencies>
```

刷新一下 maven，下载依赖。

11.3 创建一个 package

在 src/main/java 下创建一个 package。这里名为 com.atguigu.influxdb.client。



最后，项目结构如上图所示。

11.4 示例：查看 InfluxDB 健康状态

11.4.1 创建 ExampleHealthy 类

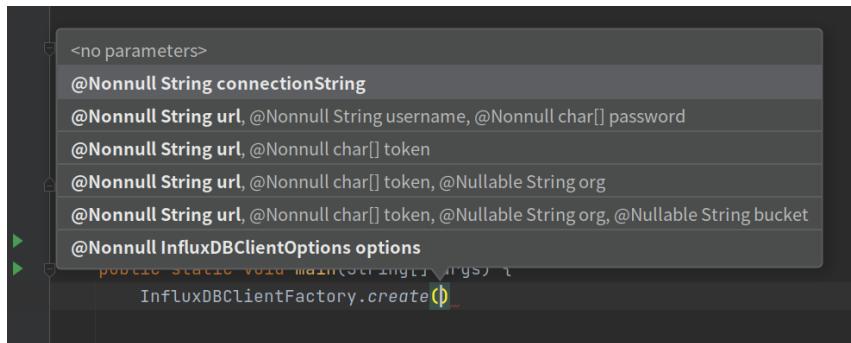
如图所示



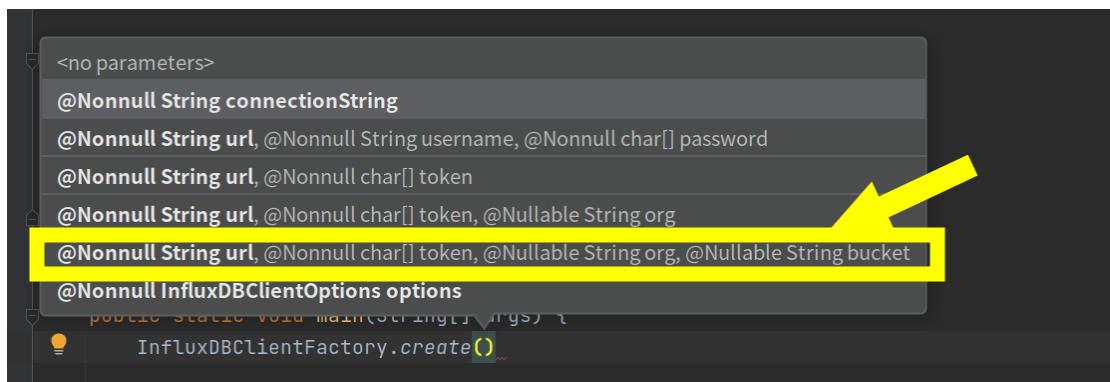
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

11.4.2 创建客户端对象

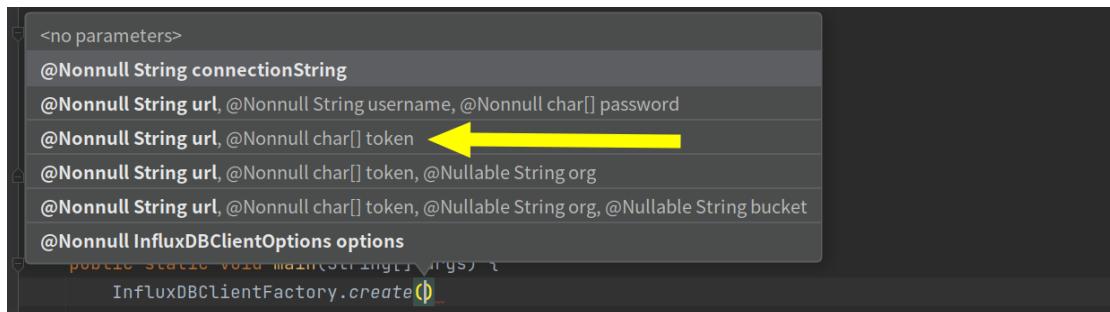
influxdb-client-java 内部其实封装的是各种 HTTP 请求，所以 token, org 什么 HTTP API 上需要的东西，在创建客户端的时候都需要考虑。



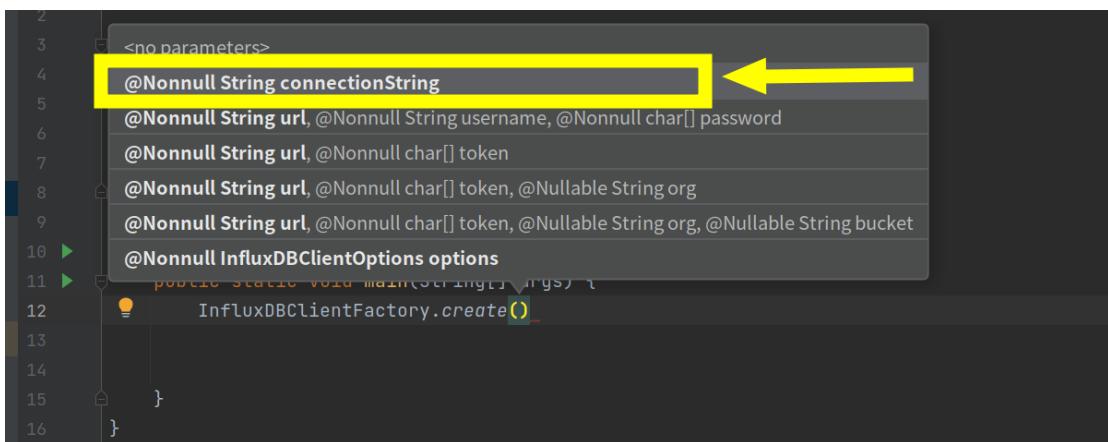
从上图可以看到 `InfluxDBClientFactory.create` 方法其实有多种重载。这是因为不同的接口它需要的权限和操作的范围不同。比如一个读写权限的 token，它只能对某个存储桶进行操作，那么建立连接时就应该指定 `bucket`，也就是使用下图的重载。



但如果你用的是操作员 token，希望完成一些创建组织，删除用户的操作，那么就不应该在创建连接时指定存储桶。此时，应该使用下图所示的重载。



不过、检查 InfluxDB 的健康状态不需要任何权限和 token。此时，我们只需指定一个 URL，那就可以使用下图所示的重载了。

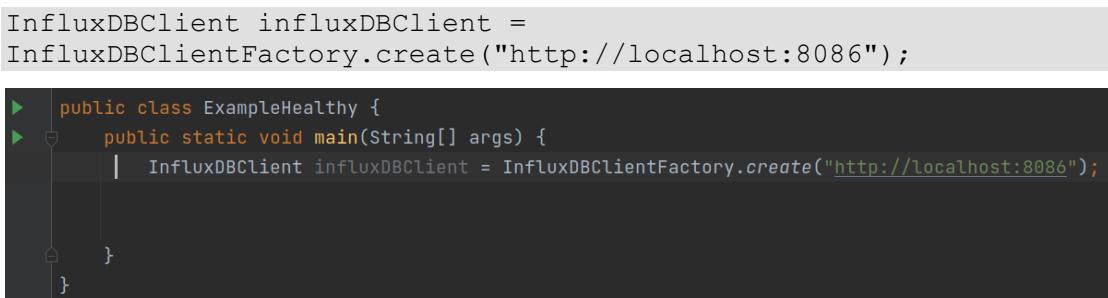


```

2
3     <no parameters>
4     @Nonnull String connectionString
5     @Nonnull String url, @Nonnull String username, @Nonnull char[] password
6     @Nonnull String url, @Nonnull char[] token
7     @Nonnull String url, @Nonnull char[] token, @Nullable String org
8     @Nonnull String url, @Nonnull char[] token, @Nullable String org, @Nullable String bucket
9
10    @Nonnull InfluxDBClientOptions options
11
12    public static void main(String[] args) {
13        InfluxDBClientFactory.create()
14    }
15
16 }

```

老师这里是在 Ubuntu 上演示，目标 URL 是 <http://localhost:8086>。所以最终代码如下。



```

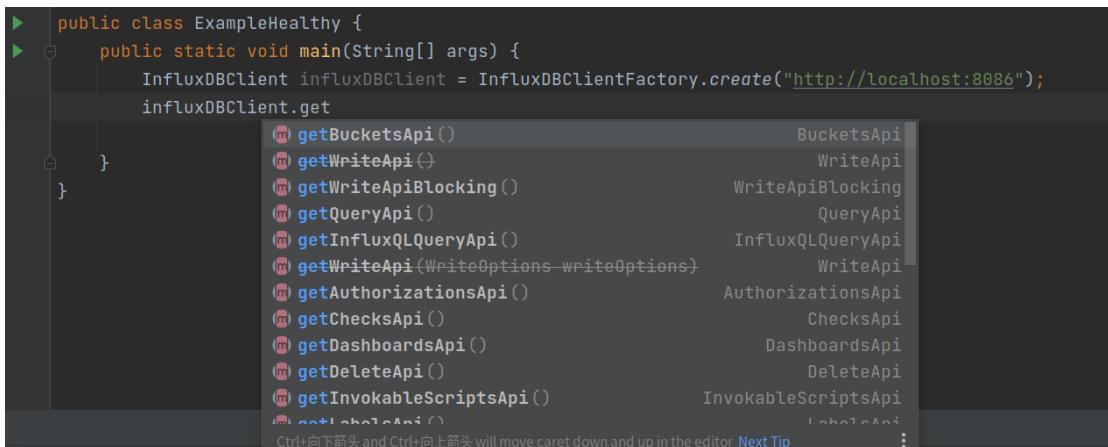
public class ExampleHealthy {
    public static void main(String[] args) {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create("http://localhost:8086");
    }
}

```

InfluxDBClient 对象就是我们的客户端对象。

InfluxDBClient 可以返回各种 API 对象。

如下图所示。这体现了 java 对 InfluxDB HTTP API 的封装。



```

public class ExampleHealthy {
    public static void main(String[] args) {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create("http://localhost:8086");
        influxDBClient.get()
    }
}

```

getBucketsApi()	BucketsApi
getWriteApi()	WriteApi
getWriteApiBlocking()	WriteApiBlocking
getQueryApi()	QueryApi
getInfluxQLQueryApi()	InfluxQLQueryApi
getWriteApi(WriteOptions writeOptions)	WriteApi
getAuthorizationsApi()	AuthorizationsApi
getChecksApi()	ChecksApi
getDashboardsApi()	DashboardsApi
getDeleteApi()	DeleteApi
getInvokableScriptsApi()	InvokableScriptsApi
getLabelsApi()	LabelsApi

11.4.3 调用 API

一些简单的 api，也可以通过 InfluxDBClient 对象直接调用。比如我们的检查 InfluxDB 健康状态，就可以直接调用 InfluxDBClient 对象的 ping 方法。如下所示。

```
System.out.println(influxDBClient.ping());
```

```
▶ public class ExampleHealthy {  
▶   public static void main(String[] args) {  
▶     InfluxDBClient influxDBClient = InfluxDBClientFactory.create("http://localhost:8086");  
▶     System.out.println(influxDBClient.ping());|  
▶   }  
▶ }
```

11.4.4 运行

ping 方法会返回一个布尔值。如果 InfluxDB 可以 ping 通，那么就会得到 true，否则返回 false，并记录一条失败日志。



```
n: └─ ExampleHealthy ×  
    /home/dengziqi/dev_lang/jdk1.8.0_341/bin/java ...  
    true  
  
    Process finished with exit code 0
```

11.4.5 补充

在之前的版本，有一个测试 InfluxDB 是否健康的 API 叫做 health，不过现在这个接口已经被标记为废弃。health 方法返回一个 HealthCheck 对象，相对而言，对这个对象的处理比直接处理布尔值要麻烦很多。在以后的版本，提倡用 ping 方法检查健康状态。

```
▶ public class ExampleHealthy {  
▶   public static void main(String[] args) {  
▶     InfluxDBClient influxDBClient = InfluxDBClientFactory.create("http://localhost:8086");  
▶     System.out.println(influxDBClient.health());|  
▶   }  
▶ }
```

11.5 示例：查询 InfluxDB 中的数据

11.5.1 创建一个 JAVA 类

在 com.atguigu.influxdb.client 下创建一个新的 java 类，ExampleQuery。



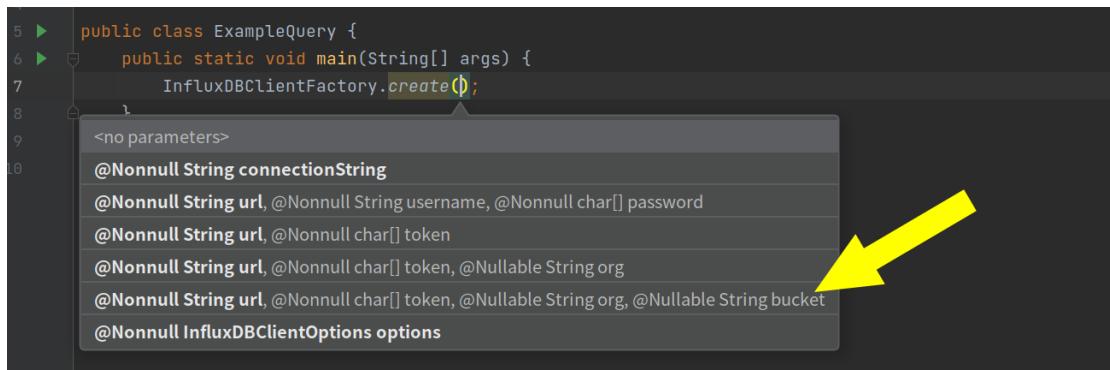
11.5.2 加入一个 main 方法

稍后 main 方法里面会写我们的查询逻辑。

```
1 package com.atguigu.influxdb.client;
2
3 public class ExampleQuery {
4     public static void main(String[] args) {
5         }
6     }
7 }
8 }
```

11.5.3 创建 InfluxDB 客户端对象

这次我们要操作 InfluxDB 中具体存储桶的数据，建立连接时，推荐选择图中的重载方法。



这个方法需要 4 个参数。

- url, InfluxDB 服务的 URL, 在老师这里就是 `http://localhost:8086`。
- token, 授权的 token, 而且类型还必须得是 `char[]`。
- org, 指定要访问的组织。
- bucket, 要访问的存储桶。

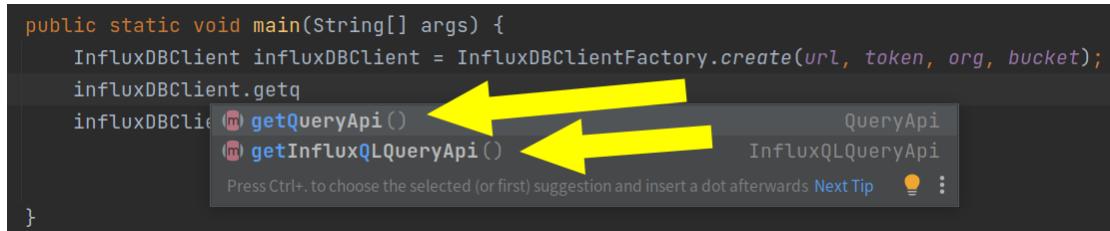
现在，我们在 `ExampleQuery` 下声明 4 个静态变量。

如下图所示：

```
5 public class ExampleQuery {
6
7     private static String org = "atguigu_com";
8
9     private static String bucket = "test_init";
10
11    private static String url = "http://localhost:8086";
12
13    private static char[] token =
14        "ZABWUTSRfIhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHogR5mAWg4XMv4tb9TP3w==".toCharArray();
15
16    public static void main(String[] args) throws InterruptedException {
17        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
18    }
19 }
```

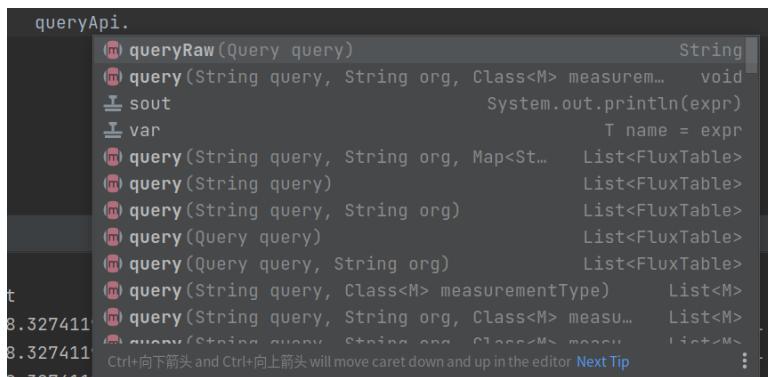
11.5.4 获取查询 API 对象

使用 InfluxDBClient 对象点一下，可以看到 InfluxDBClient 其实提供了两种 API。这也是为了兼容性来考虑的。InfluxQLQueryAPI 是 InfluxDB2.x 做的向前兼容。这里我们选择第一个方法，也就是 getQueryApi，这意味着我们使用 v2 api 进行查询。



11.5.5 了解查询 API

概括性地说，QueryApi 对象下有两个方法 query 和 queryRaw。



两个方法都需要传入一个 FLUX 脚本作为查询语句。但主要的不同点在于返回的结果上。

- queryRaw 方法返回 API 中的 CSV 格式数据（String 类型）。
- query 方法视图将查询后的结果封装为各种对象（可以自己指定也可以使用 influxdb-client-java 提供的 FluxTable）。

两个方法各有很多不同的实现，其中一大部分是用来制定连接参数的，比如你创建连接对象的时候没有制定 org 和 bucket，那么可以延迟到调用具体 api 的时候再指定。

11.5.6 query

现在，我们先用 query 方法去查询一下 InfluxDB 中的数据，现在我们要查询 test_init 存储桶最近 2 分钟的数据。

代码如下：

```
List<FluxTable> query = queryApi.query("from(bucket:\\"test_init\\")\n|> range(start:-2m)");
```

```
public static void main(String[] args) throws InterruptedException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    QueryApi queryApi = influxDBClient.getQueryApi();
    List<FluxTable> query = queryApi.query("from(bucket:\\"test_init\\") |> range(start:-2m)");
}
```

我们的查询结果 List<FluxTable>其实对应了 FLUX 查询语言中的表流概念。

我们可以打印一下 query 变量。

```
ExampleQuery ×
/home/dengziqi/dev_lang/jdk1.8.0_341/bin/java ...
[FluxTable[cOLUMNS=8, RECORDS=12], FluxTable[cOLUMNS=8, RECORDS=12], FluxTable[cOLUMNS=8, RECORDS=12]]
```

现在，我们只取第一个 FluxTable，看看里面有什么。

之前我们讲 FLUX 的时候，有讲过表流和 groupKey 之间的关系。

```
public static void main(String[] args) throws InterruptedException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    QueryApi queryApi = influxDBClient.getQueryApi();
    List<FluxTable> query = queryApi.query("from(bucket:\\"test_init\\") |> range(start:-2m)");
    FluxTable fluxTable = query.get(0);
    fluxTable.get|
        ⌂ getRecords() List<FluxRecord>
        ⌂ getColumn() List<FluxColumn>
        ⌂ getClass() Class<? extends FluxTable>
        ⌂ getGroupKey() List<FluxColumn>
    Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip ⋮
```

现在可以打印一下 groupKey，看看里面有什么东西。

代码如下：

```
public static void main(String[] args) throws InterruptedException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    QueryApi queryApi = influxDBClient.getQueryApi();
    List<FluxTable> query = queryApi.query("from(bucket:\\"test_init\\") |> range(start:-2m)");
    FluxTable fluxTable = query.get(0);
    List<FluxColumn> groupKey = fluxTable.getGroupKey();
    for (FluxColumn fluxColumn : groupKey) {
        System.out.println(fluxColumn);
    }
}
```

输出结果如下：

```
/home/dengziqi/dev_lang/jdk1.8.0_341/bin/java ...
FluxColumn[index=2, label='_start', dataType='dateTime:RFC3339', group=true, defaultValue='']
FluxColumn[index=3, label='_stop', dataType='dateTime:RFC3339', group=true, defaultValue='']
FluxColumn[index=6, label='_field', dataType='string', group=true, defaultValue='']
FluxColumn[index=7, label='_measurement', dataType='string', group=true, defaultValue='']
```

这其实可以说明我们的整个表流是以_start, _stop, _field, _measurement 4 列为 groupKey 分组的结果。

现在，我们可以尝试打印一下数据。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

代码如下：

```
public static void main(String[] args) throws InterruptedException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    QueryApi queryApi = influxDBClient.getQueryApi();
    List<FluxTable> query = queryApi.query("from(bucket:\"test_init\") |> range(start:-2m)");
    for (FluxTable fluxTable : query) {
        List<FluxRecord> records = fluxTable.getRecords();
        for (FluxRecord record : records) {
            System.out.println(record.getValues());
        }
    }
}
```

结果如下：

```
ExampleQuery / 
result=+-----+
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5633747782, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5631079852, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5631984182, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5628998012, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5620949532, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.562092542, _value=0.0, _field:10000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5616342512, _value=0.0, _field:100000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5626604322, _value=0.0, _field:100000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5616740102, _value=0.0, _field:100000, _measurement=storage_writer |
result=| _start:2022-08-31T18:53:46.5429686672, _stop:2022-08-31T18:59:46.5429686672, _time=2022-08-31T18:59:40.5609208992, _value=0.0, _field:100000, _measurement=storage_writer |
result=+-----+
```

剩下的功能大家可以自己探索。

11.5.7 queryRaw

老师在这里创建了一个新类，叫 ExampleQueryRaw。

代码复制的都是 ExampleQuery 的。

唯一不同的地方就是把 queryApi.query 改为了 queryApi.queryRaw。

同时，query 变量的类型也从 List<FluxTable> 变成了 String

```
▶ public class ExampleQueryRaw {
    1 usage
    private static String org = "atguigu_com";

    1 usage
    private static String bucket = "test_init";

    1 usage
    private static String url = "http://localhost:8086";

    1 usage
    private static char[] token =
        "ZA8uWTSRffLKhFvNW4TczWvvd2NHFw1YIVLc]9Am5iJ4ueHawWh49...jszoKybEymHqgR5mAwg4XMv4tb9TP3w==".toCharArray();

    ▶   public static void main(String[] args) throws InterruptedException {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
        QueryApi queryApi = influxDBClient.getQueryApi();
        String query = queryApi.queryRaw("from(bucket:\"test_init\") |> range(start:-2m)");
        ⚡
    }
}
```



现在，我们将查询的结果打印一下。

```
/home/dengziqi/dev_lang/jdk1.8_341/bin/java ...
,result_table,_start,_stop,_time,_value,_field,_measurement
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:00.561977614Z,698,counter,boltdb_reads_total
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:10.562503598Z,692,counter,boltdb_reads_total
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:20.562366677Z,694,counter,boltdb_reads_total
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:30.560729138Z,696,counter,boltdb_reads_total
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:40.56103196Z,702,counter,boltdb_reads_total
,_result,0,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:50.562284648Z,704,counter,boltdb_reads_total
,_result,1,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:00.561977614Z,27,counter,boltdb_writes_total
,_result,1,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:10.562503598Z,27,counter,boltdb_writes_total
,_result,1,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:20.562366677Z,27,counter,boltdb_writes_total
,result_1,2022-08-31T19:04:52.816677498Z,2022-08-31T19:05:52.816677498Z,2022-08-31T19:05:30.560729138Z,27,counter,boltdb_writes_total
```

可以看到，我们打印出了 CSV 格式的数据，这是因为 InfluxDB HTTP API 本来在请求体中放的就是 CSV 格式的数据。所以 QueryRaw 方法其实就是返回原始的 CSV。

11.6 同步写和异步写的区别

同步写，就是当我调用写入方法时，立刻向 InfluxDB 发起一个请求，将数据传送过去，而且当前线程会一直**阻塞**等待写入操作完成。

异步写，其实是我调用写入方法的时候，先不执行写入这个操作，而是将数据放入一个缓冲区。当缓冲区满了，我再真正地将数据发送给 InfluxDB，这样相当于实现了一个攒批的效果。

在后面的示例中，我们会向建议先在 InfluxDB 中创建一个名为 example_java 的存储桶，再学习后面的写入示例。

11.7 示例：同步写入 InfluxDB

11.7.1 创建一个类

这次我们创建一个名为 ExampleWriteSync 的类，org,bucket,url,token 什么的还是复制之前例子中的，但是此处我们把 bucket 改为 example_java。

总体代码如下：

```
public class ExampleWriteSync {

    private static String org = "atguigu";
    private static String bucket = "example_java";
    private static String url = "http://localhost:8086";
    private static char[] token =
        "ZA8uWTSRFFlhKhFvNW4TcZwwvd2NHFW1YIVlcj9Am5iJ4ueHawWh49_jszoKybEy
mHqgR5mAyg4XMv4tb9TP3w==".toCharArray();

    public static void main(String[] args) {
        InfluxDBClient influxDBClient =
        InfluxDBClientFactory.create(url, token, org, bucket);

    }
}
```

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```

public class ExampleWriteSync {

    1 usage
    private static String org = "atguigu_com";

    1 usage
    private static String bucket = "example_java";

    1 usage
    private static String url = "http://localhost:8086";

    1 usage
    private static char[] token =
        "ZA8uWTSRflhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w==".toCharArray();

    public static void main(String[] args) throws URISyntaxException, IOException {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    }
}

```

11.7.2 获取 API 对象

我们可以看到，InfluxDBClient 上有多种方法获取写操作 API 对象。其中 WriteApiBlocking 是同步写入，WriteApi 是异步写入。

```

public static void main(String[] args) {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    influxDBClient.wri
}
    ⏎ getWriteApiBlocking() WriteApiBlocking
    ⏎ makeWriteApi() WriteApi
    ⏎ getWriteApi() WriteApi
    ⏎ makeWriteApi(WriteOptions writeOptions) WriteApi
    ⏎ getWriteApi(WriteOptions writeOptions) WriteApi

```

Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

我们现在先使用 getWriteApiBlocking 方法获取同步写入的 API。

11.7.3 有哪些写入方法

```

private static String url = "http://localhost:8086";
    1 usage
private static String token =
    "ZA8uWTSRflhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w==";
    1 usage
private static char[] token =
        "ZA8uWTSRflhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAwg4XMv4tb9TP3w==".toCharArray();
    1 usage
public static void main(String[] args) throws URISyntaxException, IOException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    influxDBClient.wri
}
    ⏎ getWriteApiBlocking() WriteApiBlocking
    ⏎ makeWriteApi() WriteApi
    ⏎ getWriteApi() WriteApi
    ⏎ makeWriteApi(WriteOptions writeOptions) WriteApi
    ⏎ getWriteApi(WriteOptions writeOptions) WriteApi

```

Press Enter to insert, Tab to replace Next Tip

简单归纳，写入 API 给用户提供了 3 类方法写入数据。

- writeMeasurement， 用户可以写入自己的 POJO 类
- writePoint， influxdb-client-java 提供了一个 Point 类， 用户可以将一条条数据封装为一个个 Point，写入 InfluxDB。

- writeRecord，用户可以用符合 InfluxDB 行协议的字符串向 InfluxDB 写入数据。

另外，这三类方法都有与之对应的带 s 后缀的版本，表示可以一次写入多条。

11.7.4 通过 Point 对象写入 InfluxDB

11.7.4.1 构建 Point 对象

使用下面的代码创建一个 point 对象。

```
Point point = Point.measurement("temperature")
    .addTag("location", "west")
    .addField("value", 55D)
    .time(Instant.now(), WritePrecision.MS);
```

这是典型的构造器设计模式，measurement 是一个静态方法，它会帮我们 new 一个 Point。addTag 和 addField 不再解释。最终的 time，我们通过第二个参数指定写入时间戳的精度。这里是将写入的时间精度确定为了毫秒，如果你传入了一个纳秒时间戳，但精度指明了毫秒，那超出毫秒的部分会被直接截断。

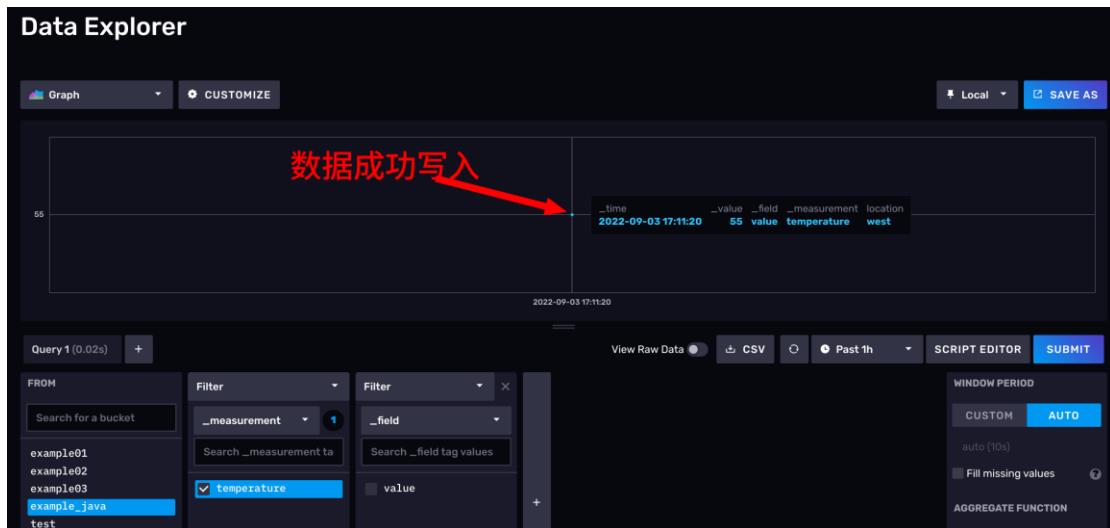
11.7.4.2 将 point 写出

使用下面的代码，直接将 point 写到 InfluxDB 中。记得在此之前创建 example_java 存储桶。

```
writeApiBlocking.writePoint(point);
```

11.7.4.3 验证写入结果

执行程序后，在 InfluxDB DataExplorer 查看 example_java 里面有没有新的数据，并将它展示出来。



11.7.5 通过行协议写入 InfluxDB

11.7.5.1 注释上一个示例的代码

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

现在，我们将前面通过 Point 写数据的代码注释掉。

```

8 public static void main(String[] args) throws URISyntaxException, IOException {
9     InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
10    WriteApiBlocking writeApiBlocking = influxDBClient.getWriteApiBlocking();
11
12    //    Point point = Point.measurement("temperature")
13    //        .addTag("location", "west")
14    //        .addField("value", 55D)
15    //        .time(Instant.now(), WritePrecision.MS);
16
17    //    writeApiBlocking.writePoint(point);
18
19 }

```

11.7.5.2 编写代码

在 main 方法中追下下面的代码

```
writeApiBlocking.writeRecord(WritePrecision.NS, "temperature,location=west value=60.0");
```

此处我们在行协议中省略时间戳，让 InfluxDB 自动帮我们把时间补上。

11.7.5.3 验证写入结果

运行代码后，一样还是去 InfluxDB 上查看数据。



如图所示，第二条数据已经成功进入 InfluxDB 了。

11.7.6 通过 POJO 类写入 InfluxDB

11.7.6.1 注释上一个示例的代码

同样，我们还是先注释掉上一次用 InfluxDB 行协议写入数据的代码。

```

    public static void main(String[] args) throws URISyntaxException, IOException {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
        WriteApiBlocking writeApiBlocking = influxDBClient.getWriteApiBlocking();

        // Point point = Point.measurement("temperature")
        //     .addTag("location", "west")
        //     .addField("value", 55D)
        //     .time(Instant.now(), WritePrecision.MS);

        // writeApiBlocking.writePoint(point);

        // writeApiBlocking.writeRecord(WritePrecision.NS, "temperature,location=west value=60.0");
    }

```

11.7.6.2 添加一个静态内部类

```

private static class Temperature {
}

```

11.7.6.3 @Measurement 注解

给静态内部类加一个注解。

```

@Measurement(name = "temperature")
private static class Temperature {
}

```

@Measuremet 注解必须加到类上，表示这个类对应 InfluxDB 中的哪个测量名称。

11.7.6.4 添加成员变量

```

@Measurement(name = "temperature")
private static class Temperature {

    String location;

    Double value;

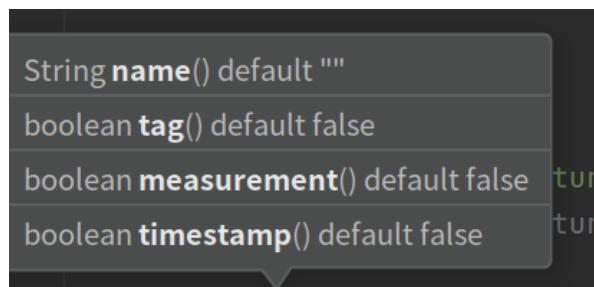
    Instant time;
}

```

11.7.6.5 @Column 注解

@Column 注解只能用在成员变量上。

@Column 有 4 种实现，如下图所示。



你可以将一个成员变量指定为 tag、measurement、timestamp 还是 field。

最终代码如下：

```

@Measurement(name = "temperature")
private static class Temperature {
}

```

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

```
@Column(tag = true)
String location;

@Column
Double value;

@Column(timestamp = true)
Instant time;
}
```

11.7.6.6 创建一个 Temperature 对象并给其属性赋值

代码如下

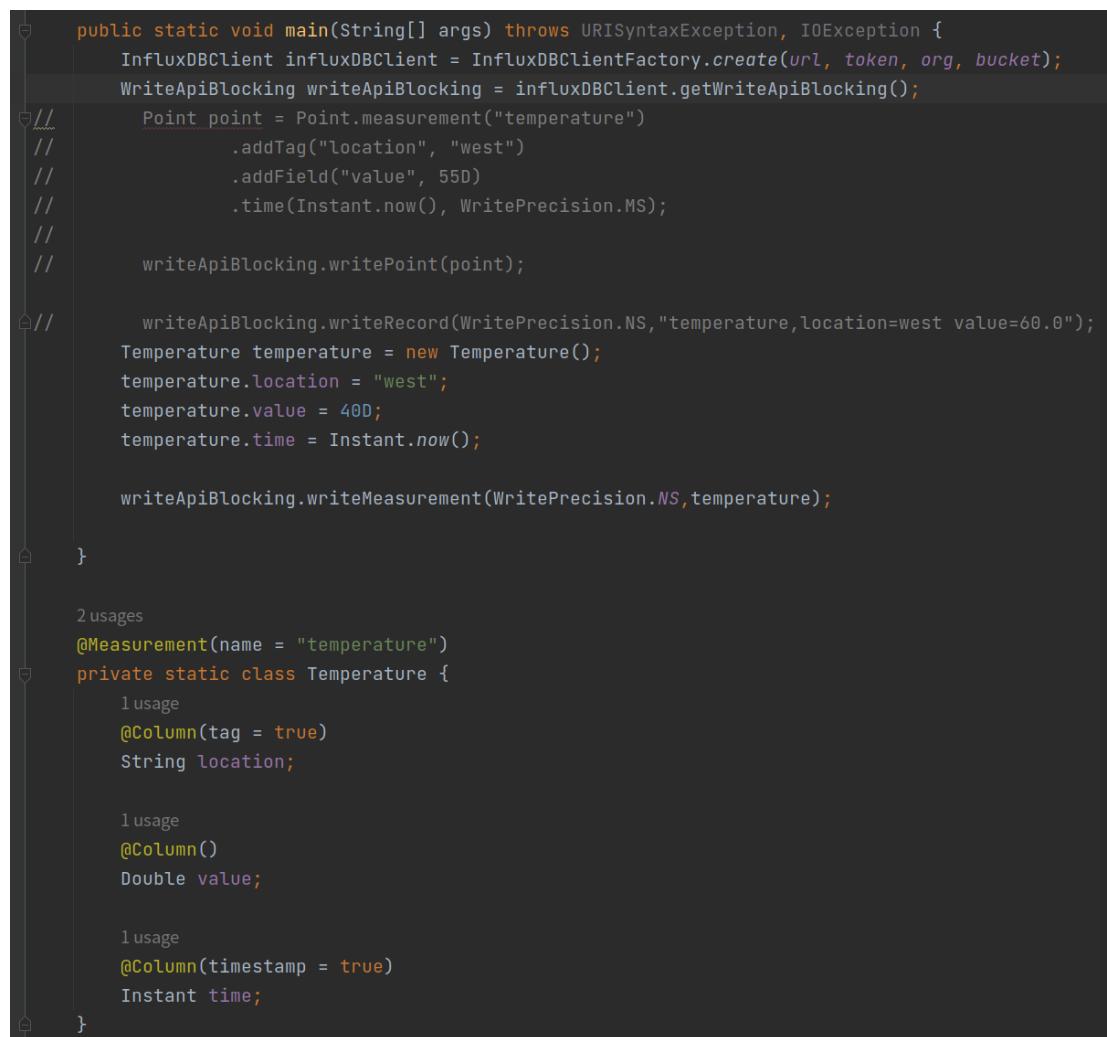
```
Temperature temperature = new Temperature();
temperature.location = "west";
temperature.value = 40D;
temperature.time = Instant.now();
```

11.7.6.7 写到 InfluxDB

现在，我们将这个 POJO 类的对象写到 InfluxDB

```
writeApiBlocking.writeMeasurement(WritePrecision.NS,temperature);
```

最终的代码如下图所示：



```
public static void main(String[] args) throws URISyntaxException, IOException {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    WriteApiBlocking writeApiBlocking = influxDBClient.getWriteApiBlocking();
    Point point = Point.measurement("temperature")
        .addTag("location", "west")
        .addField("value", 55D)
        .time(Instant.now(), WritePrecision.MS);
    writeApiBlocking.writePoint(point);

    writeApiBlocking.writeRecord(WritePrecision.NS, "temperature,location=west value=60.0");
    Temperature temperature = new Temperature();
    temperature.location = "west";
    temperature.value = 40D;
    temperature.time = Instant.now();

    writeApiBlocking.writeMeasurement(WritePrecision.NS,temperature);
}

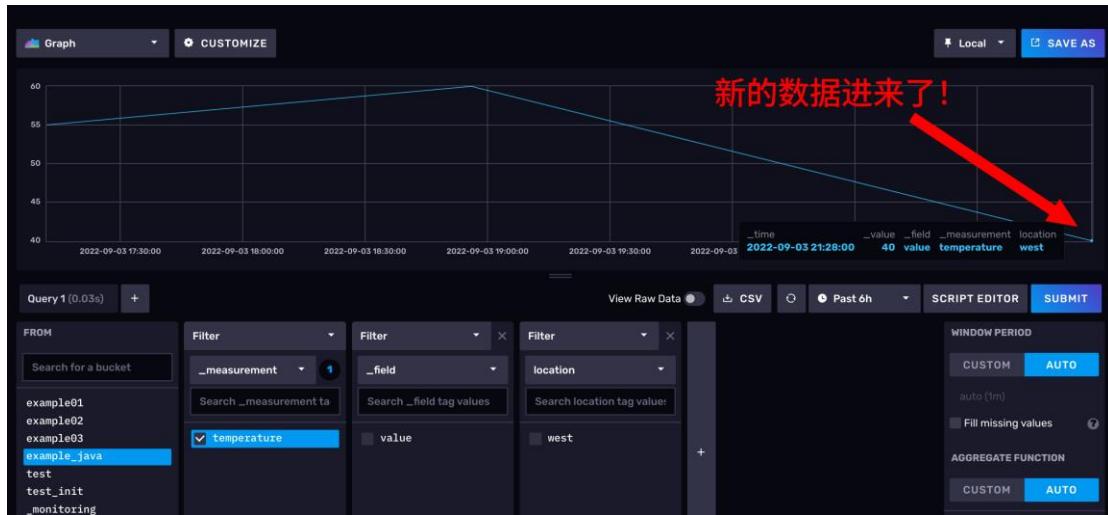
2 usages
@Measurement(name = "temperature")
private static class Temperature {
    1 usage
    @Column(tag = true)
    String location;

    1 usage
    @Column()
    Double value;

    1 usage
    @Column(timestamp = true)
    Instant time;
}
```

11.7.6.8 验证写入效果

运行 main 方法，去 DataExplorer 上查看输出效果。



如图所示，数据已经成功进入 InfluxDB。

11.8 示例：异步写入 InfluxDB

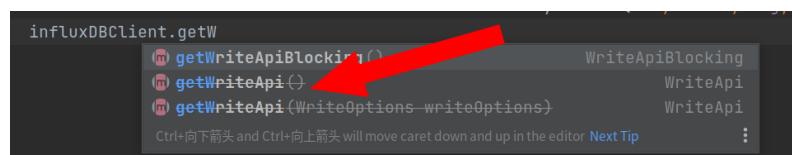
11.8.1 创建一个类

创建一个名为 ExampleWriteAsync 的类，一样还是复用我们之前的 org、bucket、url 和 token。并创建 InfluxDBClient 对象，基础代码如下图所示。

```
public class ExampleWriteAsync {
    private static String org = "atguigu_com";
    private static String bucket = "example_java";
    private static String url = "http://localhost:8086";
    private static char[] token =
        "ZA8uWTSRffLhKhFvNW4TcZwwvd2NHFV1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAWg4XMv4tb9TP3w==".toCharArray();
    public static void main(String[] args) {
        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    }
}
```

11.8.2 获取 API 对象

可以看到 getWriteApi 方法已经被标记为弃用了，



更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

现在鼓励使用的是 makeWriteApi 方法。实际上，在目前版本，getWriteApi 的内部实现已经是直接调用 makeWriteApi 了。

```
public static void main(String[] args) {
    InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
    influxDBClient.make|_
}
```

Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

11.8.3 编写写入代码

和之前的同步写入一样， writeApi 对象也有
writeRecord、writePoint、wirteMeasurement 多种写入方法，这里不再赘述。

这里，我们只用最简单的 writeRecord 方法插入一条数据，把代码跑通。

```
writeApi.writeRecord(WritePrecision.NS, "temperature,location=north value=60.0");
```

11.8.4 验证写入结果（写入失败）

此时，我们运行的代码如下图所示。

```
public class ExampleWriteAsync {

    usage
    private static String org = "atguigu_com";

    usage
    private static String bucket = "example_java";

    usage
    private static String url = "http://localhost:8086";

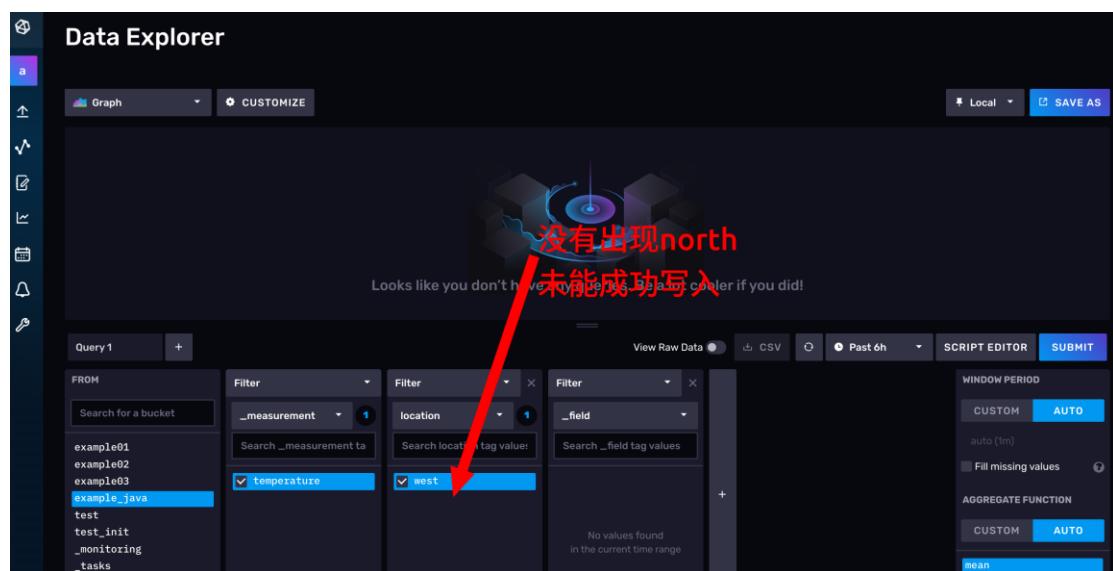
    usage
    private static char[] token =
        "ZA8uWTSRfflhKhFvNw4TcZwwvd2NHFW1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAeWg4XMv4tb9TP3w==".toCharArray();

    public static void main(String[] args) {

        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
        WriteApi writeApi = influxDBClient.makeWriteApi();

        writeApi.writeRecord(WritePrecision.NS, record: "temperature,location=north value=60.0");
    }
}
```

在 Web UI 上打开 Data Explorer 查看写入结果。



可以看到，这里没有出现我们刚才的数据，这表示我们刚才的写入失败了。但是，我们的 java 程序没有报错。

这是因为 WriteApi 会使用一个守护线程，帮我们管理缓冲区，它会在缓冲区满或者距离上次写出数据过 1 秒时将数据写出去。我们刚才就放了一条数据，缓冲区没满、write 方法调用完程序就立刻退出了，所以下台线程压根就没有做写的操作。

11.8.5 修改代码

现在，有两种方式让守护线程执行写的操作。

- 手动触发缓冲区刷写

```
writeApi.flush();
```

- 关闭 InfluxDBClient

```
influxDBClient.close();
```

这里，我们先用第一种。

修改后的代码如下

```

public class ExampleWriteAsync {

    1 usage
    private static String org = "atguigu_com";

    1 usage
    private static String bucket = "example_java";

    1 usage
    private static String url = "http://localhost:8086";

    1 usage
    private static char[] token =
        "ZABuWTSRfflhKhFvNW4TcZwwvd2NHFw1YIVlcj9Am5iJ4ueHawWh49_jszoKybEymHqgR5mAWg4XMv4tb9TP3W==".toCharArray();

    public static void main(String[] args) {

        InfluxDBClient influxDBClient = InfluxDBClientFactory.create(url, token, org, bucket);
        WriteApi writeApi = influxDBClient.makeWriteApi();

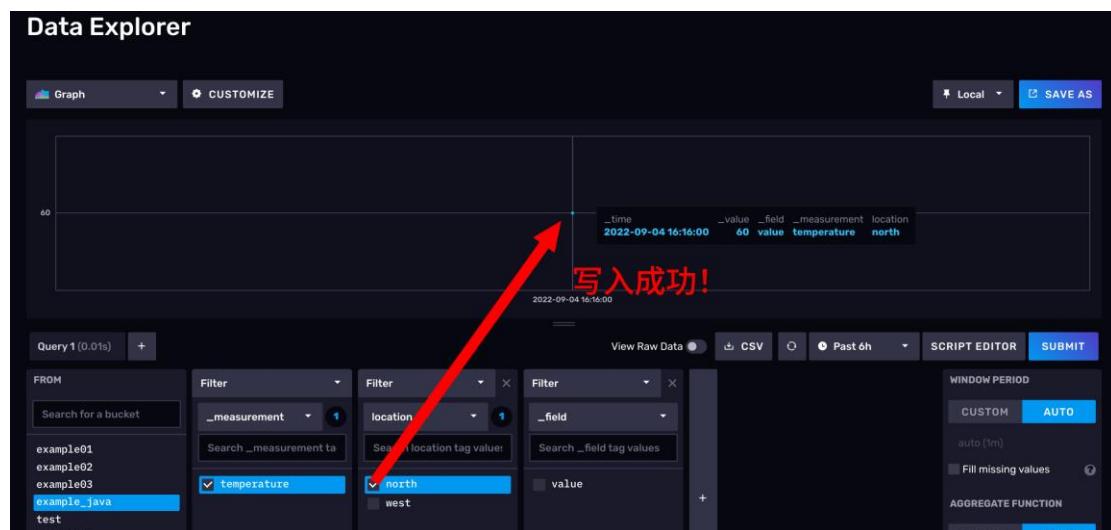
        writeApi.writeRecord(WritePrecision.NS, record: "temperature,location=north value=60.0");
        influxDBClient.close();
    }
}

```

运行，之后去 Data Explorer 上查看结果。

11.8.6 验证写入结果（写入成功）

如果能看到 location Tag 上出现的 north 标签，而且能够查出来一条数据，那么写入操作就成功了！



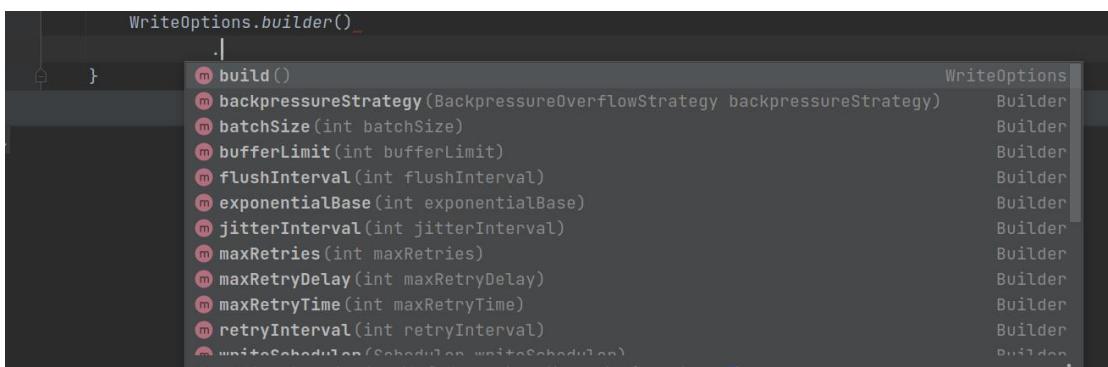
11.8.7 小结：异步写入工作逻辑

- writeApi 里有一个缓冲区，这个缓冲区的大小默认是 10000 条数据。
- 虽然有缓冲区但是 writeApi 写出数据并不是一次把整个缓冲区都写出去，而是按照批次（默认是 1000 条）的单位来写。
- 当产生被压或者写入失败时，守护线程会自动重试写入数据。

11.8.8 异步写入的配置

异步攒批的操作的守护线程隐式进行的，好在它的行为我们可以进行具体的配置。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)



influxdb-client-java 为我们提供了一个 WriteOption 对象，调用 makeWriteApi 时可以传入这个对象，通过上图的提示我们可以看到，缓冲区的大小，批的大小，刷写的间隔我们都是可以进行明确指定的。

11.8.9 默认配置

```
31 usages
@ThreadSafe
public final class WriteOptions implements WriteApi.RetryOptions {

    1 usage
    public static final int DEFAULT_BATCH_SIZE = 1000;
    1 usage
    public static final int DEFAULT_FLUSH_INTERVAL = 1000;
    1 usage
    public static final int DEFAULT_JITTER_INTERVAL = 0;
    1 usage
    public static final int DEFAULT_RETRY_INTERVAL = 5000;
    1 usage
    public static final int DEFAULT_MAX_RETRIES = 5;
    1 usage
    public static final int DEFAULT_MAX_RETRY_DELAY = 125_000;
    1 usage
    public static final int DEFAULT_MAX_RETRY_TIME = 180_000;
    1 usage
    public static final int DEFAULT_EXPONENTIAL_BASE = 2;
    1 usage
    public static final int DEFAULT_BUFFER_LIMIT = 10000;
```

11.9 兼容 V1 Api

这里只做简单的介绍。

使用 InfluxDBClientFactory 创建 Client 对象时，调用 createV1 方法。

```
ame, @Nullable char[] password, @Nonnull String database
InfluxDBClientFactory.createV1();
```

这个时候你获取的就是兼容 V1 API 的 Client 对象。

第12章 使用 InfluxDB 模板

12.1 什么是 InfluxDB 模板

InfluxDB 模板是一份 yaml 风格的配置文件。它包含了一套完整的仪表盘、Telegraf 配置和报警配置。InfluxDB 模板力图保证开箱即用，把 yaml 文件下载下来，往 InfluxDB 里一导，从数据采集一直到数据监控报警就全部为你创建好。

InfluxDB 官方在 [github](#) 上收录了一批模板。开发前可以在这里逛一逛，看有没有可以直接拿来用的。

<https://github.com/influxdata/community-templates>

12.2 示例：使用模板快速部署

在这节示例中，我们会使用社区模板快速创建一套 docker 的监控模板。要完成这个示例，你需要提前掌握 docker 的相关知识。

12.2.1 找到 Docker 模板文档

访问上一节的 <https://github.com/influxdata/community-templates> 找到 Docker 模板的目录，点进去。

可以看到，有一节的标题是 Quick install 里面有详细的配置说明。

Quick install

InfluxDB UI

In the InfluxDB UI, go to Settings->Templates and enter this URL: <https://raw.githubusercontent.com/influxdata/community-templates/master/docker/docker.yml>

Influx CLI

If you have your InfluxDB credentials [configured in the CLI](#), you can install this template with:

```
influx apply -f https://raw.githubusercontent.com/influxdata/community-templates/master/docker/docker.yml
```

Included resources

- 1 Bucket: `docker`, 7d retention
- Labels: Telegraf Plugin Labels
- 1 Telegraf Configuration
- 1 Dashboard: `Docker`
- 1 Variable: `bucket`
- 4 Alerts: Container cpu, mem, disk, non-zero exit
- 1 `Notification` Endpoint: Http Post
- 1 `Notification Rules`: Crit Alert

12.2.2 安装模板

使用 influx-cli 安装模板。

```
influx apply -f  
https://raw.githubusercontent.com/influxdata/community-  
templates/master/docker/docker.yml
```

命令执行后，会弹出下面的消息，询问你是否使用上面的资源。

```
+-----+  
| + | dashboards | Docker | 0000000000000000 | elegant-yonath-d4d015 | inputs.system | 09edf888e5a92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | unruffled-benz-d4d007 | inputs.cpu | 09edf888e7a92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | dreamy-heyrovsky-d4d011 | inputs.disk | 09edf888e8292000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | goofy-yallow-d4d005 | inputs.diskio | 09edf888e7292000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | inspiring-goodall-d4d00b | inputs.docker | 09edf888e6292000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | agreeing-goldberg-d4d013 | inputs.kernel | 09edf888e7e92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | vigilant-chatelet-d4d00f | inputs.mem | 09edf888e6a92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | adoring-pasteur-d4d00d | inputs.net | 09edf888e6692000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | gallant-sanderson-d4d003 | inputs.processes | 09edf888e6e92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | elastic-morse-d4d001 | inputs.swap | 09edf888e8692000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | elegant-yonath-d4d015 | inputs.system | 09edf888e5a92000 |  
+-----+  
| + | telegraf | Docker Monitoring | 0000000000000000 | nice-kilby-d4d009 | outputs.influxdb_v2 | 09edf888e7692000 |  
+-----+  
| TOTAL | 13 |  
+-----+  
? Confirm application of the above resources (y/N) y
```

这里所指的资源，涉及要在你的 InfluxDB 中创建什么名称的 Bucket，创建什么定时任务和报警任务，创建什么仪表盘等等。如果你是在一个正在生产、且存在相似业务的 InfluxDB 上，那这个列表还是要好好看一看的，避免出现存储桶重名之类的现象。

确定没有问题之后，敲 y 回车。

LABEL ASSOCIATIONS				
RESOURCE TYPE	RESOURCE NAME	RESOURCE ID	LABEL NAME	LABEL ID
dashboards	Docker	09ee20c80f3b5000	inputs.docker	09edf888e6292000
dashboards	Docker	09ee20c80f3b5000	inputs.system	09edf888e5a92000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.cpu	09edf888e7a92000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.disk	09edf888e8292000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.diskio	09edf888e7292000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.docker	09edf888e6292000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.kernel	09edf888e7e92000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.mem	09edf888e6a92000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.net	09edf888e6692000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.processes	09edf888e6e92000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.swap	09edf888e8692000
telegrafs	Docker Monitoring	09ee20c80fab6000	inputs.system	09edf888e5a92000
telegrafs	Docker Monitoring	09ee20c80fab6000	outputs.influxdb_v2	09edf888e7692000
		TOTAL	13	

Stack ID: 09ee20c80d692000

如果接下来展示的内容以 Stack ID: xxxx 结尾，那说明安装成功！

这里的 Stack（栈）概念，其实就是模板的实例。

12.2.3 查看安装结果

现在，我们打开 InfluxDB 的 Web UI，看一下我们模板的导入效果。

下图是模板为我们创建的存储桶，名为 docker。

12.2.3.1 存储桶

有一个名为 docker 的存储桶

The screenshot shows the InfluxDB Web UI with the 'BUCKETS' tab selected. At the top, there are tabs for SOURCES, BUCKETS, TELEGRAF, SCRAPERS, and API TOKENS. Below the tabs is a search bar with the placeholder 'Filter buckets...' and a dropdown menu set to 'Sort by Name (A → Z)'. A red arrow points to the 'docker' bucket, which is listed with the following details: Retention: 7 days, ID: e219a4dab9429856. At the bottom of the screen, there are buttons for '+ ADD DATA' and 'SETTINGS'.

12.2.3.2 telegraf 配置

一个名为 Docker Monitor 的 Telegraf 配置文件，这个配置文件可能需要根据你 Docker 的配置进行一些修改

The screenshot shows the 'Load Data' section of the Grafana interface. The 'TELEGRAF' tab is selected. A red arrow points to the 'Docker Monitoring' section, which is highlighted in yellow. The text '模板为我们创建的 Telegraf 配置' (Template created for us) is overlaid on the right side of the screenshot.

12.2.3.3 仪表盘

模板还帮我们创建了一个名为 Docker 仪表盘，只不过我们现在的 Bucket 里面还没有数据，所以这列的图表都还没有显示出来。

The screenshot shows the 'Dashboards' section of the Grafana interface. A red arrow points to the 'Docker' dashboard, which is highlighted in yellow. The text '模板创建的仪表盘' (Dashboard created by template) is overlaid on the right side of the screenshot. Below the dashboard list, a red arrow points to the 'Docker' dashboard itself, with the text '这些仪表盘是空的，是因为我们的Telegraf还没跑起来 存储桶中还没数据' (These dashboards are empty because our Telegraf hasn't run yet and there's no data in the bucket) overlaid.

12.2.3.4 报警规则

模板还帮我们设置了 4 个报警规则，根据题目的描述，分别是

- 容器 CPU 使用率持续 15 分钟超过 80%
- 容器硬盘使用率超过 80%
- 容器的内存使用率持续 15 分钟超过 80%

- 容器没有以 0 状态（正常结束）退出。

The screenshot shows the Grafana 'Alerts' interface with four active checks listed:

- Container CPU Usage Pct**: Container cpu usage is above 80% for 15 minutes. Last completed at 2022-09-04T15:45:00Z, Last updated 1 hour ago, ID: 09ee20c80ef98000, Task ID: 09ee20c810d0f000.
- Container Disk Usage**: Container disk usage is above 80%. Last completed at 2022-09-04T15:45:00Z, Last updated 1 hour ago, ID: 09ee20c80ef98002, Task ID: 09ee20c81450f000.
- Container Mem Usage Pct**: Container mem usage is above 80% for 15 minutes. Last completed at 2022-09-04T15:45:00Z, Last updated 1 hour ago, ID: 09ee20c80ef98001, Task ID: 09ee20c81210f000.
- Container Non Zero Exit Code**: Container exits with a non zero exit status. Last completed at 2022-09-04T15:50:00Z, Last updated 1 hour ago, ID: 09ee20c80f398000, Task ID: 09ee20c81310f000.

12.2.4 运行 Telegraf 采集数据

现在我们要使用 Telegraf 跑 docker 模板里的配置文件。但之前我们在~/bin 目录下写过一个 host_tel.sh 的启停脚本，那个文件会保证全局只有一个 telegraf。所以，为了避免混乱，我们现在要先把之前的 telegraf 停掉。

```
host_tel.sh stop
```

接下来，我们编写新的脚本。

还是在~/bin 目录下，创建 docker_tel.sh 文件。键入以下内容。

```
#!/bin/bash

export INFLUX_TOKEN=h106QMEj47juNUco-6T-
op1Tzz0IeMh5MhBIDT8vUdv1R3BVeAzMvWGq2DtmJ1cyuPwvPmHTLbZLTbnKxz3UK
A==

export INFLUX_HOST=http://localhost:8086/
export INFLUX_ORG=atguigu

/opt/module/telegraf-1.23.4/usr/bin/telegraf --config
http://localhost:8086/api/v2/telegrafs/09edf888eeeb6000
```

按照 docker 模板的要求，在运行 telegraf 之前，我们需要声明 INFLUX_TOKEN、INFLUX_HOST 和 INFLUX_ORG 三个变量。

然后，我们修改一下 docker_tel.sh 的执行权限。

```
chmod 755 ./docker_tel.sh
```

最终，启动 docker_tel.sh

```
./docker_tel.sh
```

```

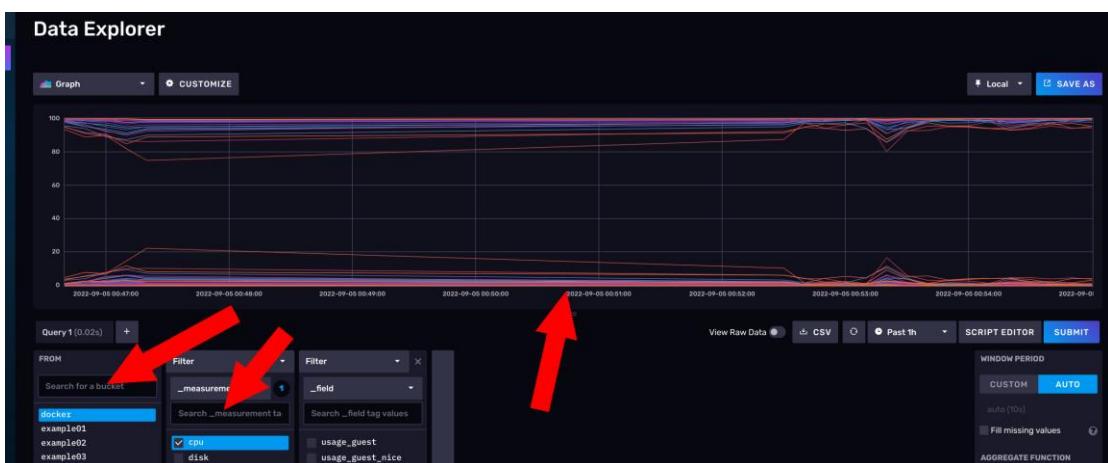
Mon 5 Sep - 00:51 ~
@atguigu docker_tel.sh
2022-09-04T16:52:08Z W! DeprecationWarning: Option "container_names" of plugin "inputs.docker" deprecated since version 1.4.0 and will be removed in 2.0.0: use 'container_name_include' instead
2022-09-04T16:52:08Z W! DeprecationWarning: Option "perdevice" of plugin "inputs.docker" deprecated since version 1.18.0 and will be removed in 2.0.0: use 'perdevice_include' instead
2022-09-04T16:52:08Z W! DeprecationWarning: Option "total" of plugin "inputs.docker" deprecated since version 1.18.0 and will be removed in 2.0.0: use 'total_include' instead
2022-09-04T16:52:08Z I! Starting Telegraf 1.23.4
2022-09-04T16:52:08Z I! Loaded inputs: cpu disk diskio docker kernel mem net processes swap system
2022-09-04T16:52:08Z I! Loaded aggregators:
2022-09-04T16:52:08Z I! Loaded processors:
2022-09-04T16:52:08Z I! Loaded outputs: influxdb_v2
2022-09-04T16:52:08Z I! Tags enabled: host=dengziqi-WUJIE-16
2022-09-04T16:52:08Z W! Deprecated inputs: 0 and 3 options
2022-09-04T16:52:08Z I! [agent] Config: Interval:10s, Quiet:false, Hostname:"dengziqi-WUJIE-16", Flush Interval:10s

```

如上图所示，telegraf 成功启动。

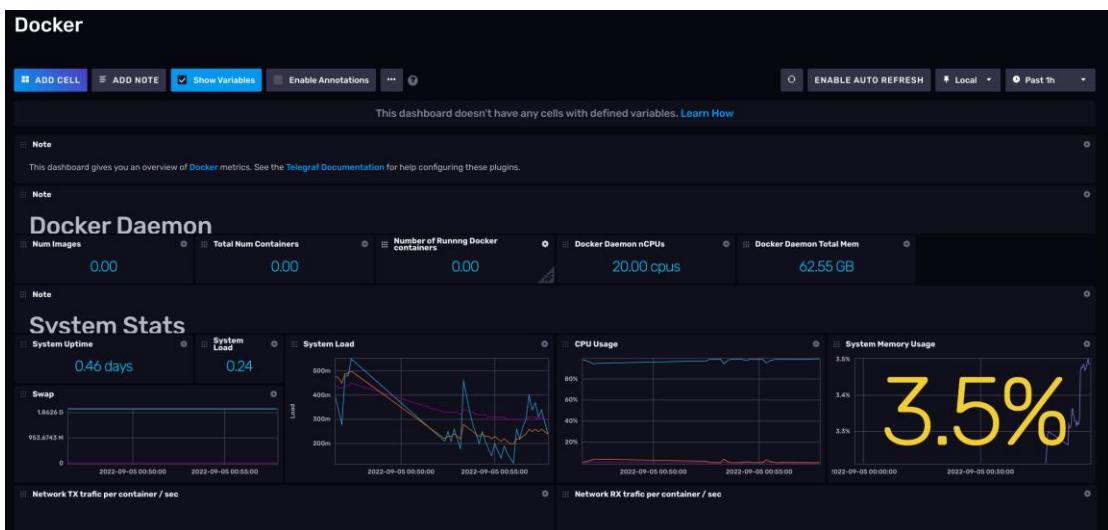
12.2.5 查看模板效果

首先，可以在 DataExplorer 中查看 docker 存储桶中有没有数据。



如图所示，数据已经成功进入 InfluxDB。

接下来，我们可以看一下仪表盘的状态，如下图，仪表盘也是成功展示数据的。



更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

12.2.6 运行一个 docker 容器

使用下面的命令，运行一个 docker 的入门容器。

```
docker run -dp 80:80 docker/getting-started
```

如果你的主机上没有 docker/getting-started 镜像，那么 docker 回去 dockerhub 上拉取镜像，因为这个镜像在国外，速度可能会很慢，如果拉取失败，请自行百度替换源的方法。

另外，容器运行后，还需要一段时间让 telegraf 采集数据。

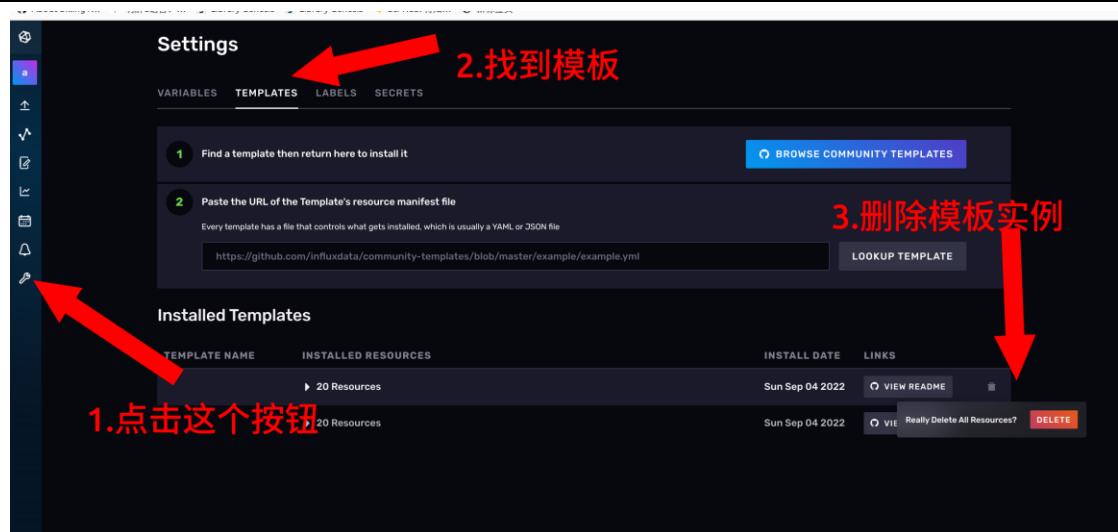
12.2.7 再次查看仪表盘

如下图所示，我们的镜像和容器数都从 0 变成了 1。而且系统的内存使用率变高了。



12.2.8 删除 stack (模板实例)

- 在 Web UI 上，点击左侧工具栏的 按钮，再点击上方的 TEMPLATES。可以看到已安装的模板列表，每个末班列表的右边都有一个删除按钮，通过这种方式可以快捷删除。



删除后，stack 所涉的所有资源会全部消失。

- 你也可以通过 influx-cli 来删除

```
influx stacks remove -o atguigu --stack-id=09ee20c80d692000
```

influx-cli 的功能很全，你也可以用它来给 stack 重命名，或者查看一个组织下的所有 stack 等。

12.3 InfluxDB 模板的不足

12.3.1 FLUX 兼容性

我们之前看到，很多 InfluxDB 模板里面都会内嵌 FLUX 语言脚本。但是不同的 InfluxDB 里面编译进去的 FLUX 语言版本是不一样的。最重要的是，FLUX 语言目前还处在较快的变动期，标准库还未确定。尤其是后面版本的 FLUX 可能会废弃之前版本里的函数和 API。这就导致 FLUX 语言向前兼容性不佳。

下图是 InfluxDB 版本和 FLUX 语言版本的对应关系。

InfluxDB OSS version	Flux version
InfluxDB nightly	0.181.0
InfluxDB 2.4	0.179.0
InfluxDB 2.3	0.171.0
InfluxDB 2.2	0.162.0
InfluxDB 2.1	0.139.0
InfluxDB 2.0	0.131.0
InfluxDB 1.8	0.65.1
InfluxDB 1.7	0.50.2

比如，InfluxDB 2.4 的 FLUX 版本是 0.179，Influx2.0 就是 0.131.0。4 次迭代就能让 FLUX 相差 40 多个版本。最典型的比如 sample-data 这个模板。

The screenshot shows a GitHub repository page for 'influxdata/community-templates'. The repository is public and has 44 watchers, 140 forks, and 228 stars. The 'Code' tab is selected. A red arrow points to the dropdown menu where 'master' is currently selected, with 'sample-data' listed as an alternative. Below the dropdown, there's a list of files and their commit history. At the bottom, there's a note: 'Sample Data Template'.

在 InfluxDB 2.3 之后就不能用了。如果你加载这份模板，会提示配置文件的第 11 行有问题。

```
✗ 五 2 9月 - 09:26 ➤ /opt/module/influxdb2-client-2.4.0-linux-amd64
@atguigu ➤ ./influx apply -u http://raw.githubusercontent.com/influxdata/community-templates/master/sample-data/sample-data.yml

Error: failed to read template(s) from "http://raw.githubusercontent.com/influxdata/community-templates/master/sample-data/sample-data.yml": yaml: line 11: mapping values are not allowed in this context
```

这其实就是因为这份模板中的，`csv.from(url:xxx)`已被废弃。

```

32 lines (28 sloc) | 951 Bytes
1 apiVersion: influxdata.com/v2alpha1
2 kind: Task
3 metadata:
4   name: fetch-sample-data
5 spec:
6   every: 5m
7   name: Fetch Sample Data
8   query: |
9     import "experimental"
10
11   csv.from(url: "https://influx-testdata.s3.amazonaws.com/bitcoin-historical-annotated.csv")
12   |> to(bucket: "sample_data")
13

```

新版本的FLUX，`csv.from`不允许设置url

再加上官方维护的这套模板仓库、出于“年久失修”的状态，缺乏维护。所以、你可能还需要自己手动修改一下模板。

12.3.2 生态不如 Grafana

Grafana 是一个专门做监控仪表盘的框架、支持设置监控任务而且支持以多种数据库作为数据源。社区活跃度高于 InfluxDB，所以 Grafana 框架下有更加丰富且好用的模板。Grafana 还支持将 InfluxDB 作为数据源，这样在框架选型上可以使用 InfluxDB+Grafana 的方案。这样，InfluxDB 就只负责读写，Grafana 负责数据展示和报警。

下图是 Grafana 社区提供的模板。可以看到过滤后，支持以 InfluxDB 作为数据源的仪表盘就有 1117 个。

The screenshot shows a search results page for 'Search dashboards'. A red arrow points to the 'Data Source' dropdown menu, which is set to 'InfluxDB'. Another red arrow points to the search results, specifically highlighting three dashboards:

- Telegraf: system dashboard**: 5/5 ratings, 30.1K downloads, InfluxDB
- Apache JMeter Dashboard using Core InfluxdbBackendListenerClient**: 3.29/5.7 ratings, 19.1K downloads, InfluxDB
- Mware vSphere - Overview**: 3.97/5.29 ratings, 16.8K downloads, InfluxDB

A red box highlights the 'InfluxDB' selection in the dropdown menu.

反观 InfluxDB 收录的模板、活跃度和模板更新上都赶不上 Grafana。

The screenshot shows a GitHub repository page for 'influxdata/community-templates'. The 'Issues' tab is selected, indicated by a red arrow and the label 'Issues少'. On the right side, there is a 'Releases' section with a red arrow pointing to it, containing the text 'FLUX语言一周一更新但是模板没有跟着变'.

第13章 定时任务

13.1 什么是定时任务

InfluxDB 任务是一个定时执行的 FLUX 脚本，它先查询数据，然后以某种方式进行修改或聚合，然后将数据写回 InfluxDB 或执行其他操作。

13.2 示例：将数据转成 json 发给别的应用

13.2.1 创建任务的途径

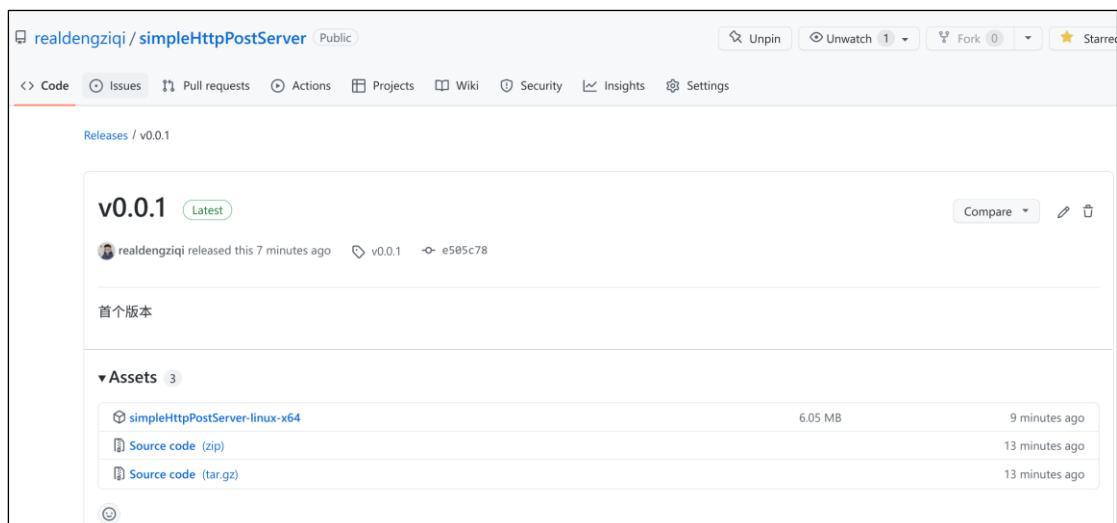
有很多方式可以帮你创建任务，比如 DataExplorer、Notebook、HTTP API 和 influx-cli。不过，此处我们为了还原定时任务的本来面貌，我们会使用 influx-cli 去创建定时任务。

13.2.2 本任务的需求

我们的定时任务要实现下面的需求。

- (1) 每 30s 调度一次
- (2) 查询最近 30s 的第一条数据
- (3) 将数据转为 json
- (4) 将 json 通过 HTTP 发送 SimpleHttpPostServer。

SimpleHttpPostServer 是老师自己用 go 语言写的一个最简单的 HTTP POST 服务，它的功能就是接收一个 POST 请求，然后把请求体的内容转为字符串打印出来。你可以在本次课程的资料（在尚硅谷微信公众号回复“大数据”获取）中获取 simpleHttpServer 的源码和编译后程序。或者访问 github 地址 <https://github.com/realdengziqi/simpleHttpPostServer>，下载源码自行编译，或者在发行记录中下载我已编译好的 linux-x64 可执行程序。



13.2.3 启动 simpleHttpPostServer

下载 simpleHttpPostServer 后，cd 到其所在目录，执行 simpleHttpPostServer 程序。效果如下图所示，终端会被阻塞。

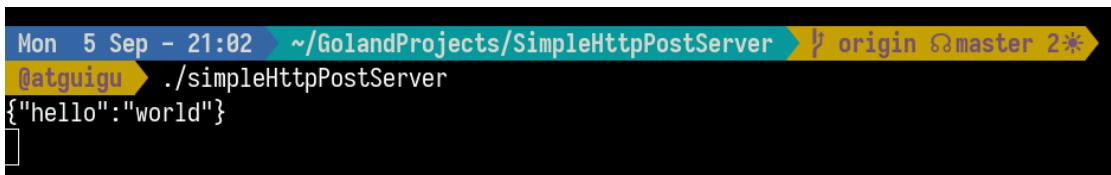


13.2.4 测试 simpleHttpPostServer 是否正常工作

我们可以使用 curl 来验证 simpleHttpPostServer 是否正常工作。使用下面的命令，向 simpleHttpPostServer 发送一个 POST 请求。

```
curl -POST http://localhost:8080/ -d '{"hello":"world"}'
```

之后，查看 simpleHttpPostServer 所占的终端，如果终端出现了一条新的数据，那么说明 simpleHttpPostServer 工作正常。



13.2.5 在 DataExplorer 中编写 FLUX 脚本

我们首先在 DataExplorer 中把查询到转为 json 的这段逻辑写好。

我们要查询的是 test_init 存储桶下的 go_goroutines 测量。这个测量反应的是我们当前 InfluxDB 程序中的 goroutines（轻量级线程）数量。

打开 DataExplrer，编写如下代码

```
import "json"
```

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

```
import "http"

from(bucket: "test_init")
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go goroutines")
|> first(column: "_value")
|> map(
  fn: (r) => {
    status_code = http.post(url:"http://localhost:8080/", data:
json.encode(v:r))
    return {r with status_code:status_code}
  }
)

import "json"
import "http"

from(bucket: "test_init")
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go goroutines")
|> first(column: "_value")
|> map(
  fn: (r) => {
    status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
    return {r with status_code:status_code}
  }
)
```

代码解释：

- from -> range -> filter， 指定了数据源并取出了我们想要的序列， 其中 range 的 start 参数我们写死-30s。
- first 函数， Flux 查询 InfluxDB 返回的数据默认是按照时间从先到后排序的， first 函数配合前面的查询相当于只取了最近 30s 的第一条数据。
- map 函数， 我们在 map 函数里完成数据的发送。这里需要注意， 因为 map 函数要求必须返回 record， 而且输出的 record 不能和输入的 record 一模一样。所以， 结尾 return 的时候， 我们使用了 with 语法， 给 map 输出的 record 增加了一个字段。也就是我们发出 http 请求响应的状态码。在 map 中的匿名函数， 我们使用了 http.post 向 <http://localhost:8080/> 发送了一条 json 格式的数据。

13.2.6 运行代码并观察效果

现在， 我们点击 SUBMIT 按钮， 执行这个 FLUX 查询脚本， 观察 DataExplorer 返回的数据和 simpleHttpPostServer 的输出。

13.2.6.1 观察 DataExplorer

点击 SUBMIT 后， 点一下 view Raw Data 按钮， 我们关注 table 格式的数据。可以看到， 现在数据中多了 status_code 一列， 而且它的值是 200。而且， 因为 first()函数的作用， 这

次查询，我们只产生了一行数据。

The screenshot shows the InfluxDB Data Explorer. At the top, there are tabs for 'Graph' and 'CUSTOMIZE', and buttons for 'Local' and 'SAVE AS'. Below the tabs is a table with one row of data:

_RESULT	_measurement	_field	_value	_start	_stop	_time	status_code
0	go_goroutines	gauge	1269	2022-09-05T14:45:53.430Z	2022-09-05T14:46:23.430Z	2022-09-05T14:46:02.036Z	200

Below the table is a 'Query 1 (0.02s)' section containing the Flux query:

```

1 import "json"
2 import "http"
3
4 from(bucket: "test_init")
5 |> range(start:-30s)
6 |> filter(fn: (r) => r["_measurement"] == "go_goroutines")
7 |> first(column: "_value")
8 |> map(
9   fn: (r) => {
10     status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
11     return {r with status_code:status_code}
12   }
13 )

```

13.2.6.2 观察 SimpleHttpPostServer

可以看到，我们的 httpServer 顺利收到了 JSON。

```

$ Mon 5 Sep - 22:46 ~/GoLandProjects/SimpleHttpPostServer ✘ origin Qmaster 2* 
@atguigu ./simpleHttpPostServer
{"_field": "gauge", "_measurement": "go_goroutines", "_start": "2022-09-05T14:57:34.571187023Z", "_stop": "2022-09-05T14:58:04.571187023Z", "_time": "2022-09-05T14:57:42.036393554Z", "_value": 1269}

```

截至目前，说明我们的 FLUX 脚本可以实现需求，现在的问题就是如何将这份脚本设为定时任务。

13.2.7 配置定时调度

现在，在我们的查询逻辑前面插入一行。

```
option task = { name: "example_task", every: 30s, offset:0m }
```

表示，我们做了一个设定，指定了一个名为 example_task 的任务。这个任务每隔 30 秒执行一次。offset 这里暂时设为 0m，后面我们会专门讲这里的 offset 有什么意义。

```

import "json"
import "http"

option task = { name: "example_task", every: 30s, offset:0m }

from(bucket: "test_init")
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> first(column: "_value")
|> map(
  fn: (r) => {
    status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
    return {r with status_code:status_code}
  }
)

```

13.2.8 使用 influx-cli 创建任务

虽然我们在 DataExplorer 里写了一个 option，但是具体到 option 会不会生效，必须要看 FLUX 脚本再跟 InfluxDB 的那个 HTTP API 进行交互。所以这里点击 SUBMIT 按钮只会再执行一次查询，option task 并不会生效。这里，我们会先用 influx-cli 创建一遍任务。

先把 Flux 脚本复制出来，在 /opt/modules/examples 里创建一个文件，就叫 example_task.flux 吧。将之前写的脚本粘贴进去。

```
import "json"
import "http"

option task = { name: "example_task", every: 30s, offset:0m }

from(bucket: "test_init")
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> first(column: "_value")
|> map(
  fn: (r) => {
    status_code = http.post(url:"http://localhost:8080/", data:
  json.encode(v:r))
    return {r with status_code:status_code}
  }
)
```

使用下面的命令，创建 FLUX 任务。

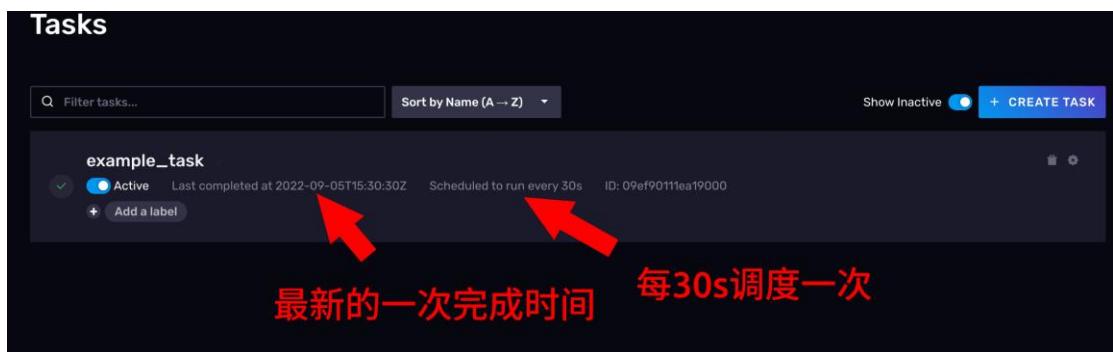
```
./influx task create --org atguigu -f
/opt/module/examples/example_task.flux
```

可以看到，我们的任务已经成功创建了。

```
Mon 5 Sep - 23:26  /opt/module/influxdb2-client-2.4.0-linux-amd64
@atguigu > ./influx task create --org atguigu_com -f /opt/module/examples/example_task.flux
D          Name      Organization ID      Organization   Status Every Cron Script
D
9ef90111ea19000     example_task     84f8210c906f0282     atguigu_com    active 30s
```

13.2.9 在 Web UI 上查看定时任务

点击左侧工具栏的  按钮，查看任务列表。可以看到，任务已经成功创建。



The screenshot shows the 'Tasks' page in the InfluxDB Web UI. At the top, there is a search bar labeled 'Filter tasks...', a sorting dropdown 'Sort by Name (A → Z)', and a 'CREATE TASK' button. Below the header, a table lists the task details:

	Name	Organization ID	Organization	Status	Every	Cron	Script
9ef90111ea19000	example_task	84f8210c906f0282	atguigu_com	active	30s		

Two red arrows point to specific details in the table row for 'example_task': one points to the 'Last completed at 2022-09-05T15:30:30Z' timestamp, and another points to the 'Scheduled to run every 30s' cron expression.

最新的一次完成时间 **每30s调度一次**

点一下任务的名称，可以进去看任务执行详情。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

Tasks > example_task Runs

可以快速修改任务

STATUS	SCHEDULE	STARTED	DURATION	VIEW LOGS
success	2022-09-05 23:32:30	2022-09-05 23:32:30	0.017 seconds	VIEW LOGS
success	2022-09-05 23:32:00	2022-09-05 23:32:00	0.016 seconds	VIEW LOGS
success	2022-09-05 23:31:30	2022-09-05 23:31:30	0.016 seconds	VIEW LOGS
success	2022-09-05 23:31:00	2022-09-05 23:31:00	0.015 seconds	VIEW LOGS

详情页面上，可以看到任务的调度时间、开始时间、任务耗时等信息。在最上方点击 EDIT TASK 按钮，可以看到当时的任务定义。在这里，你也可以直接修改任务的定义。

Edit Task

```

1 import "json"
2 import "http"
3
4 option task = { name: "example_task", every: 30s, offset:0m }
5
6 from(bucket: "test_init")
7 |> range(start:-30s)
8 |> filter(fn: (r) => r["_measurement"] == "go_goroutines")
9 |> first(column: "_value")
10 |> map(
11   fn: (r) => {
12     status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
13     return {r with status_code:status_code}
14   }
15 )

```

13.2.10 在接收端查看定时任务效果

数据的接收端就是 simpleHttpPostServer。可以看到，我们的接收端目前就是每隔 30s 收到一条 json 数据。

```

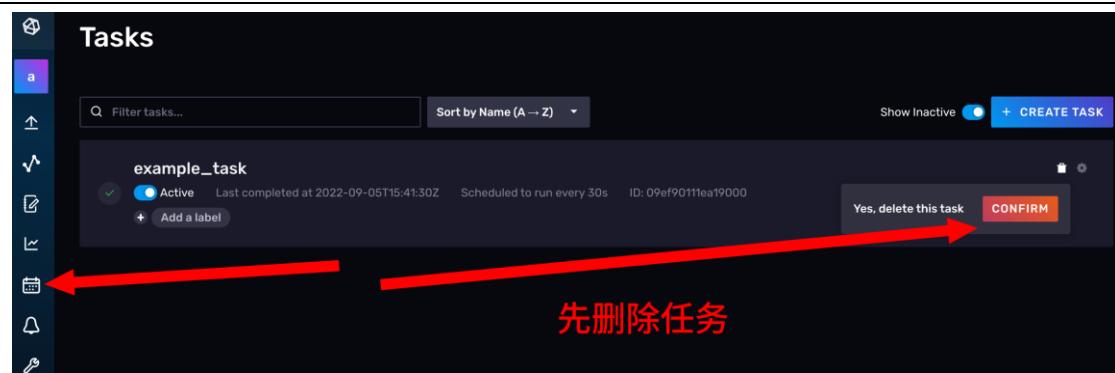
{"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:33:00Z", "stop": "2022-09-05T15:33:30Z", "time": "2022-09-05T15:33:02.048878156Z", "value": 1287}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:33:30Z", "stop": "2022-09-05T15:34:00Z", "time": "2022-09-05T15:33:32.037652276Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:34:00Z", "stop": "2022-09-05T15:34:30Z", "time": "2022-09-05T15:34:02.03751324Z", "value": 1285}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:34:30Z", "stop": "2022-09-05T15:35:00Z", "time": "2022-09-05T15:34:32.038802296Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:35:00Z", "stop": "2022-09-05T15:35:30Z", "time": "2022-09-05T15:35:02.036739218Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:35:30Z", "stop": "2022-09-05T15:36:00Z", "time": "2022-09-05T15:35:32.037819599Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:36:00Z", "stop": "2022-09-05T15:36:30Z", "time": "2022-09-05T15:36:02.037629885Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:36:30Z", "stop": "2022-09-05T15:37:00Z", "time": "2022-09-05T15:36:32.039837336Z", "value": 1286}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:37:00Z", "stop": "2022-09-05T15:37:30Z", "time": "2022-09-05T15:37:02.034649887Z", "value": 1284}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:37:30Z", "stop": "2022-09-05T15:38:00Z", "time": "2022-09-05T15:37:32.038825861Z", "value": 1284}, {"_field": "gauge", "_measurement": "go_goroutines", "start": "2022-09-05T15:38:00Z", "stop": "2022-09-05T15:38:30Z", "time": "2022-09-05T15:38:02.035728752Z", "value": 1284}

```

完成！

13.2.11 使用 DataExplorer 创建任务

这一次，我们用 DataExplorer 来创建任务。现在，我们先将已经存在的任务删除。



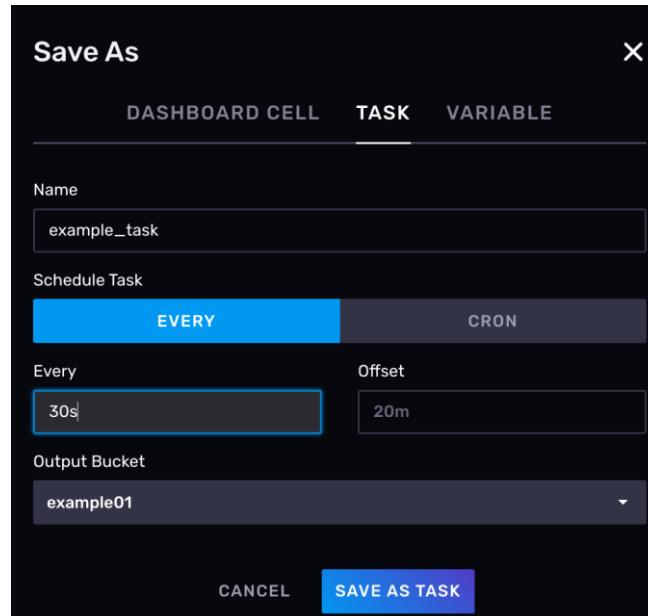
打开 DataExplorer，编辑 FLUX 脚本，将我们之前写的查询脚本粘进去。注意，要删除 option 一行。如下图所示：

```

1 import "json"
2 import "http"
3
4
5 from(bucket: "test_init")
6 |> range(start:-30s)
7 |> filter(fn: (r) => r["_measurement"] == "go goroutines")
8 |> first(column: "_value")
9 |> map(
10   fn: (r) => {
11     status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
12     return {r with status_code:status_code}
13   }
14 )

```

完成上面的操作后，点击 DataExplorer 页面右上方的 SAVE AS 按钮，在弹出的对话框中选择 TASK 选项卡。



- Name，填写为 example_task
- Every，填写 30s

- Offset 可以空着，这样默认就是 0。此处显示的 20m 是前端渲染效果，和具体的任务执行无关。
- 此处的 OutputBucket 的填写需要注意，本来我们自己编写的脚本并没有指定要把数据回写到 InfluxDB，但是如果从 Web UI 创建定时任务的话，Output Bucket 不能不设，这样会造成回写操作。这也是为什么我们之前不用 Web UI 创建任务的原因。

配置好后，点击 SAVE AS TASK。

13.2.12 再次查看任务详情（注意 Web UI 的小动作）

现在，我们再次回到任务列表。可以看到，任务已经成功创建，并且已经正常运行。

STATUS	SCHEDULE	STARTED	DURATION	
success	2022-09-06 00:02:30	2022-09-06 00:02:30	0.018 seconds	<button>VIEW LOGS</button>
success	2022-09-06 00:02:00	2022-09-06 00:02:00	0.021 seconds	<button>VIEW LOGS</button>
success	2022-09-06 00:01:30	2022-09-06 00:01:30	0.016 seconds	<button>VIEW LOGS</button>

点击 EDIT TASK 按钮，查看我们的 FLUX 脚本。你会惊奇的发现，我们的 FLUX 代码居然被修改了，原本不回写数据库的操作被强制加上了一个 to 函数。另外，可以看到我们代码的前面也被加上了一个 option task 代码，这说明 Web UI 的页面点按操作只不过是帮我们完成了手敲代码的几个步骤。

```

import "json"
import "http"
option v = i
bucket: "example01"
}

option task = {
  name: "example_task",
  every: 30s,
}

from(bucket: "test_init")
|> range(start:-30s)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> first(column: "_value")
|> map(
  fn: (r) => {
    status_code = http.post(url:"http://localhost:8080/", data: json.encode(v:r))
    return r with status_code:status_code
  }
)
|> to(bucket: "example01", org: "atguigu_com")

```

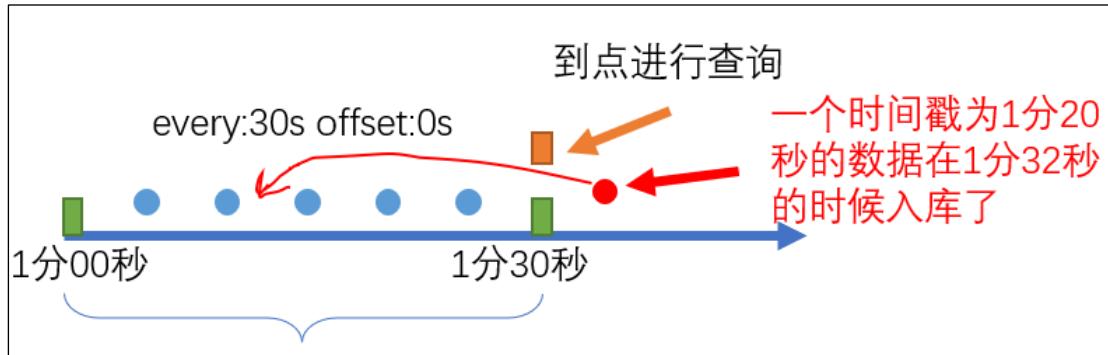
总的来说，看开发者能否接收代码被隐式修改。如果这种行为无法接受，那么强烈建议用 influx-cli 的方式去创建任务。

13.3 数据迟到问题

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

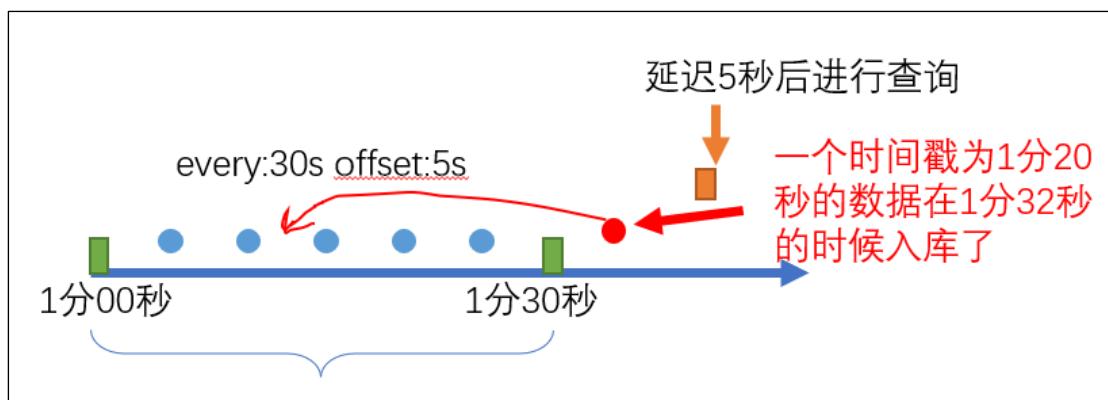
option 中的 offset 是专门用来帮我们处理迟到问题的。

首先，我们来关注一个迟到的场景，如下图所示。我们的定时任务每次查询最近 30 秒的数据。同时，调度的间隔设为每 30 秒执行一次。



这个时候，由于网络的延迟，本该 1 分 20 秒入库的数据，1 分 32 秒的时候才来。但在 1 分 30 秒的时候，我们的查询已经执行完了。这个时候我们错过了 1 分 20 秒的数据。

这个时候，如果我们将 offset 设为 5 秒。如下图所示：



定时任务的执行时间向后延迟了 5 秒，但是查询的还是原来范围的数据。这个时候定时任务执行的时间是 1 分 35 秒。原先的迟到数据就能被我们查询到了。

13.4 cron 表达式

其实 InfluxDB 的定时任务还支持 cron 表达式。option 的写法如下：

```
option task = {
    // ...
    cron: "0 * * * *",
}
```

本教程就不做详细演示了。

13.4.1 参考资料

crontab 是 linux 上一个可以设置定时执行命令的工具。cron 表达式就是最早在就是在 crontab 中使用的一种表示时间间隔的表达式。相关资料可以参考菜鸟教程开源的 cron 教程。

<https://www.runoob.com/linux/linux-comm-crontab.html>

13.4.2 cron 辅助开发工具

如果是对 cron 非常熟悉，一口气能直接写对那当然是很好的。如果开发时一次写不准可以百度一些在线的 cron 生成工具作为辅助。

不过，这里我更推荐 gitee 上的开源 cron 生成器项目。比如：

<https://gitee.com/toktok/easy-cron> 这一个基于 node.js 的 cron 生成工具。你可以把代码拉下来自己部署，也可以使用它的在线 demo。

<http://www.easysb.cn/open/easy-cron/index.html>

这个工具可以同时支持 5、6、7 字段的 cron 表达式。

The screenshot shows the 'Cron表达式测试页面' (Cron Expression Test Page) from www.easysb.cn. The interface includes a search bar for '请输入cron表达式(http://www.easysb.cn)' and a configuration button. The main area has tabs for '秒' (Seconds), '分' (Minutes), '时' (Hours), '日' (Days), '月' (Months), and '周' (Weeks). The '秒' tab is selected, showing options like '每秒' (Every second), '区间' (Range) from 0 to 59 seconds, '循环' (Loop) every 1 second, and '指定' (Specify) with a list of numbers from 0 to 59. To the right, a sidebar displays the generated cron expression: '4 1/2 5 7-8 1 ? *'. It also shows the breakdown of the expression: '秒 4', '分 1/2', '时 5', '日 7-8', '月 1', '周 ?'. Below this, there are sections for '执行时间' (Execution Time) set to '2015-09-02 12:23:10' and '执行预览' (Execution Preview) showing dates from 2016-01-07 to 2017-01-08 at 05:01:04.

13.5 补充：InfluxDB 抓取任务的本质

之前我们在 Web UI 里设置的抓取任务，其背后其实就是定时执行的 FLUX 脚本。只不过 InfluxDB 在 API 上将他们分开了。

```
import "experimental/prometheus"
prometheus.scrape(url: "http://localhost:8086/metrics")
```

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

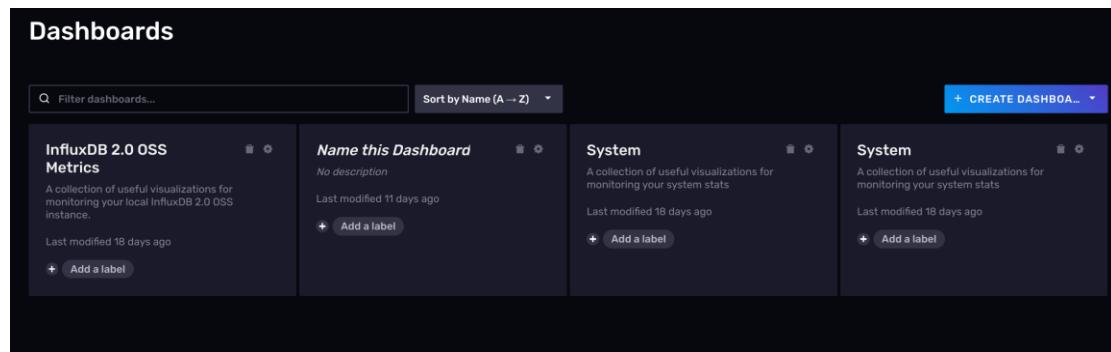
在当前的 FLUX 版本中，experimental/prometheus 库为我们提供了采集 prometheus 格式数据的能力。详细可以参考 <https://docs.influxdata.com/flux/v0.x/prometheus/scrape-prometheus/>

第14章 InfluxDB 仪表盘

14.1 什么是 InfluxDB 仪表盘

前面已经随着课程内容给大家介绍过 InfluxDB 的仪表盘功能了。

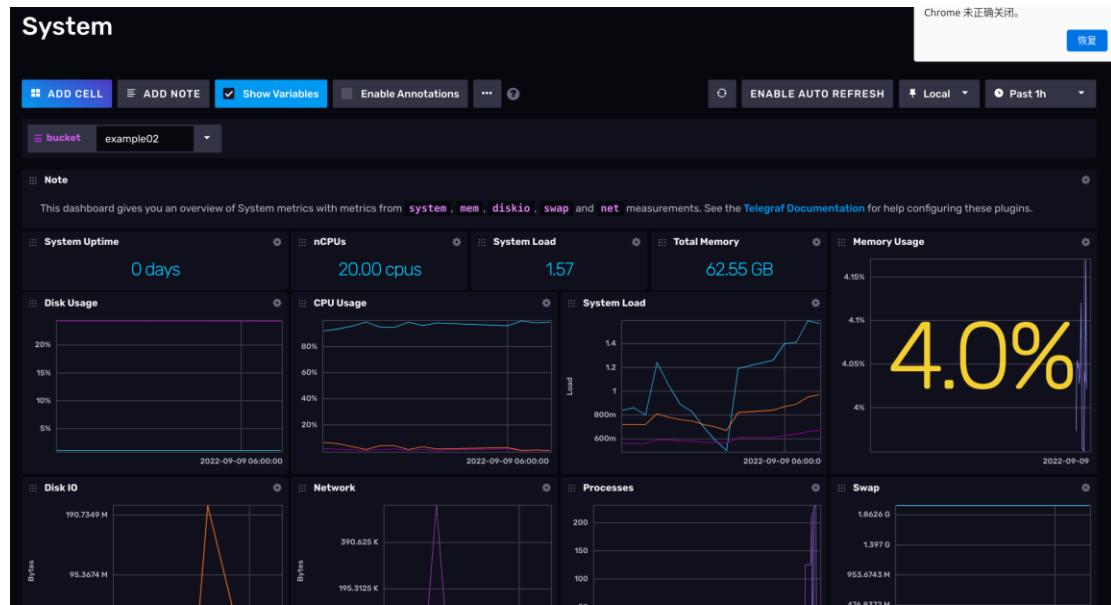
点击左侧的  按钮，可以进入 InfluxDB 的仪表盘管理页面。可以看到仪表盘的管理页面，如下图所示：



The screenshot shows the 'Dashboards' section of the InfluxDB UI. At the top, there's a search bar labeled 'Filter dashboards...' and a dropdown menu 'Sort by Name (A → Z)'. On the right, a blue button '+ CREATE DASHBOARD...' is visible. Below the header, there are four dashboard cards:

- InfluxDB 2.0 OSS Metrics**: A collection of useful visualizations for monitoring your local InfluxDB 2.0 OSS instance. Last modified 18 days ago. Buttons: '+ Add a label'.
- Name this Dashboard**: No description. Last modified 11 days ago. Buttons: '+ Add a label'.
- System**: A collection of useful visualizations for monitoring your system stats. Last modified 18 days ago. Buttons: '+ Add a label'.
- System**: A collection of useful visualizations for monitoring your system stats. Last modified 18 days ago. Buttons: '+ Add a label'.

我这里打开一个 System 仪表盘，注意，这个仪表盘中的内容依赖我们之前做的示例 2。



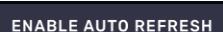
这是一个监控主机硬件与网络资源的仪表盘。仪表盘中的每个 Cell 其实都是一个 FLUX 查询语句，通过执行 FLUX 获取数据结果，再使用 UI 将它展示为各类图表。在你打开仪表盘的一瞬间，InfluxDB 就会执行这些查询。

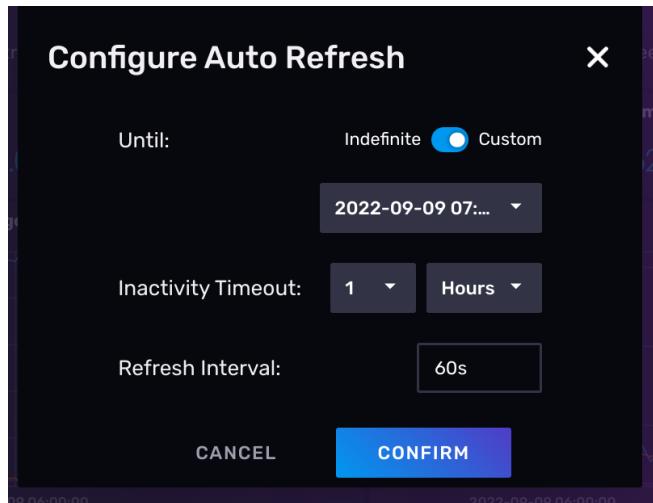
14.2 仪表盘控件

14.2.1 手动刷新

右上方的  按钮，点击一次可以重新执行一轮仪表盘中的查询。因为通常的 FLUX 脚本都是查询距当前一段时间的数据，所以刷新的功能还是比较必要的。

14.2.2 开启自动刷新

右上方的  按钮，可以开启仪表盘的自动刷新

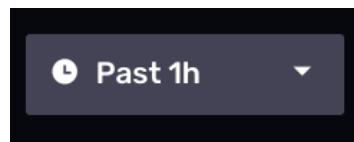


14.2.3 切换显示时区

"Local" 按钮，可以选择将当前的日期时间显示为当前时区还是 UTC。

14.2.4 设定查询范围

指定查询过去多长一段时间的数据。



14.2.5 添加一个 Cell

Cell 就是仪表盘中多个的图形的一个图形。添加图形对应的是左上角 ADD CELL 按钮。



14.2.6 添加一个 Note

一个 Note 也是仪表盘中的一个模块，支持 Markdown 语法。对应左上角 ADD NOTE

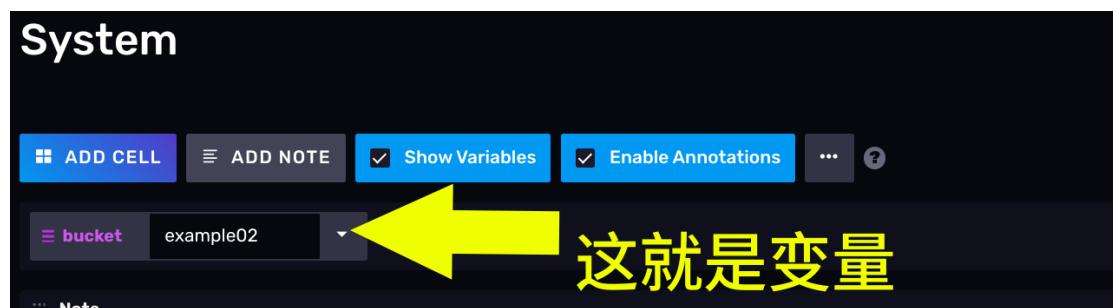
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

按钮。



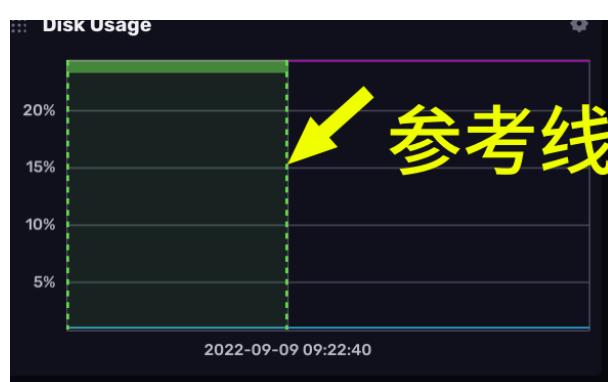
14.2.7 显示变量

如果仪表盘中包含涉及到变量的查询，那么在仪表盘的顶部会出现一个下拉菜单，通过下拉菜单这一指定变量的值，从而操作仪表盘展示响应数据。对应左上角的 Show Variables。



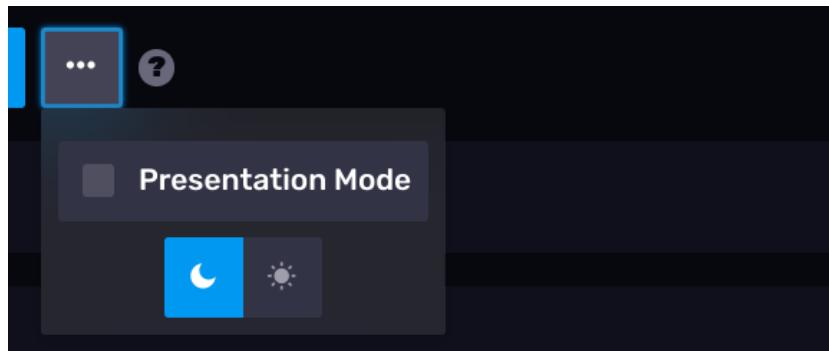
14.2.8 开启注解

你可以按住 shift 和鼠标左键，在仪表盘的图示上添加参考线。打开个关闭注解会影响参考线的可见性。



14.2.9 全屏和黑夜模式

此功能在左上角的 ... 按钮，如图所示：



14.3 示例：制作可交互的动态仪表盘

本示例要对 CPU 使用情况的相关指标制作仪表盘，这依赖于示例 2。请在完成示例 2 的基础上完成改示例。

14.3.1 需求

用户希望我们的仪表盘上能加入一个下拉菜单以选择查看哪个 CPU 的使用情况。要监控的指标是 `useage_user`，仪表盘上要显示每 1 分钟，CPU 使用率的最大值、最小值和中位数。

14.3.2 创建变量

这里先不解释为什么创建变量。

鼠标悬停在左侧的 按钮，在弹出栏上选择 Variables。如图所示：



点击右上角 CREATE VARIABLES 按钮，选择 New Variable，会弹出一个创建变量的对话框。在右上角的 Type 为 Query 的前提下，在脚本编辑区键入以下内容：

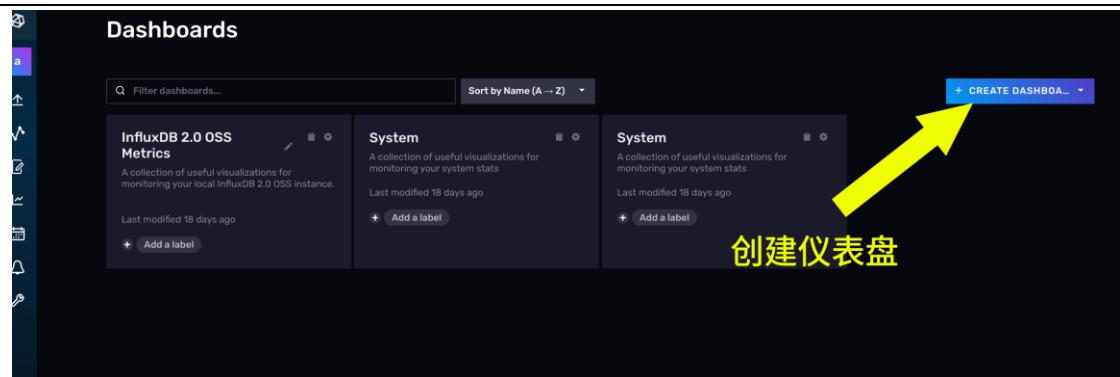
```
import "influxdata/influxdb/schema"  
  
schema.tagValues(bucket: "example02", tag:"cpu")
```

解释：

这个脚本可以查询出 example02 存储桶中的 cpu 标签有哪些标签值。

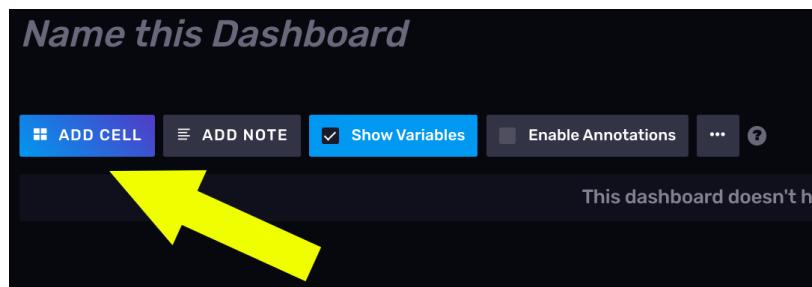
左上角需要给变量指定一个名称，这里老师输入的是 CPU。

14.3.3 创建新的仪表盘



回到仪表盘管理页面，点击 CREATE DASHBOARD 按钮，创建一个新的仪表盘。点击左上角的 ADD CELL 按钮。

14.3.4 创建新的 cell



可以看到，又出现了我们熟悉的 DataExplorer。进入后直接切换到 SCRIPT EDITOR。
键入以下内容。

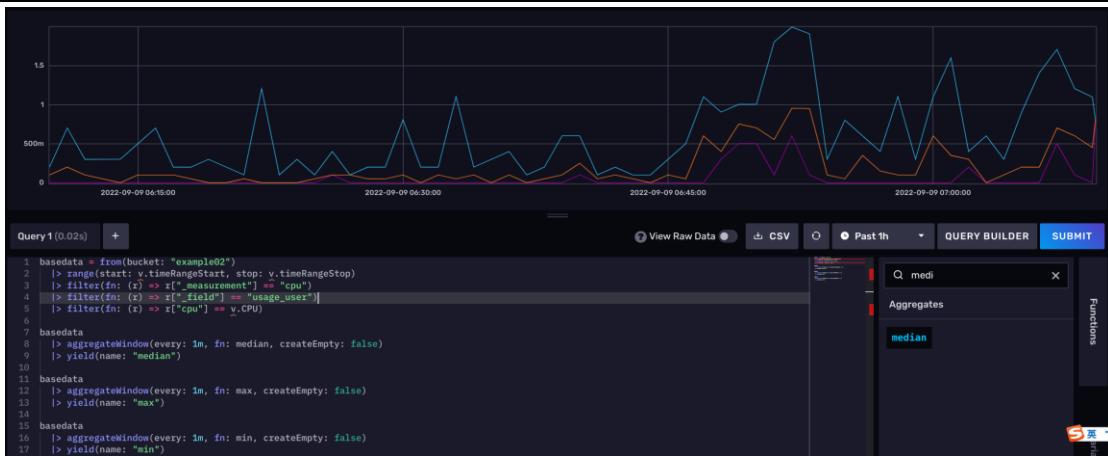
```
basedata = from(bucket: "example02")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "cpu")
|> filter(fn: (r) => r["_field"] == "usage_user")
|> filter(fn: (r) => r["cpu"] == v.CPU)

basedata
|> aggregateWindow(every: 1m, fn: median, createEmpty: false)
|> yield(name: "median")

basedata
|> aggregateWindow(every: 1m, fn: max, createEmpty: false)
|> yield(name: "max")

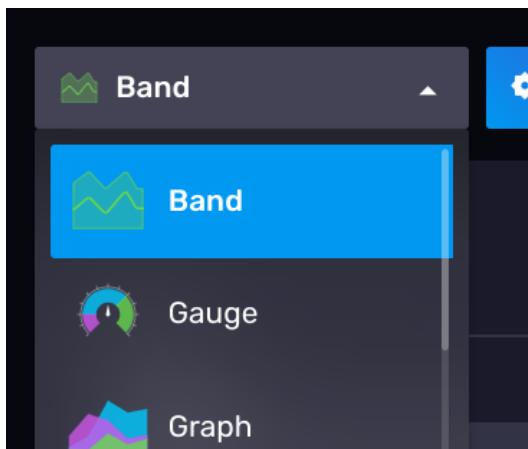
basedata
|> aggregateWindow(every: 1m, fn: min, createEmpty: false)
|> yield(name: "min")
```

点击 SUBMIT 查看效果。



14.3.5 优化展示效果

默认的可视化类型为 Graph，我们现在将它切换为 Band，表示带有边界的折线图。



切换图形后，点击 CUSTOMIZE，进行自定义设置。

有一栏是 Aggregate Functions，在这里分别指定 Upper Column Name 为 max。Main Column Name 为 median，Lower Column Name 为 min。



这就是有边界的折线图的效果。

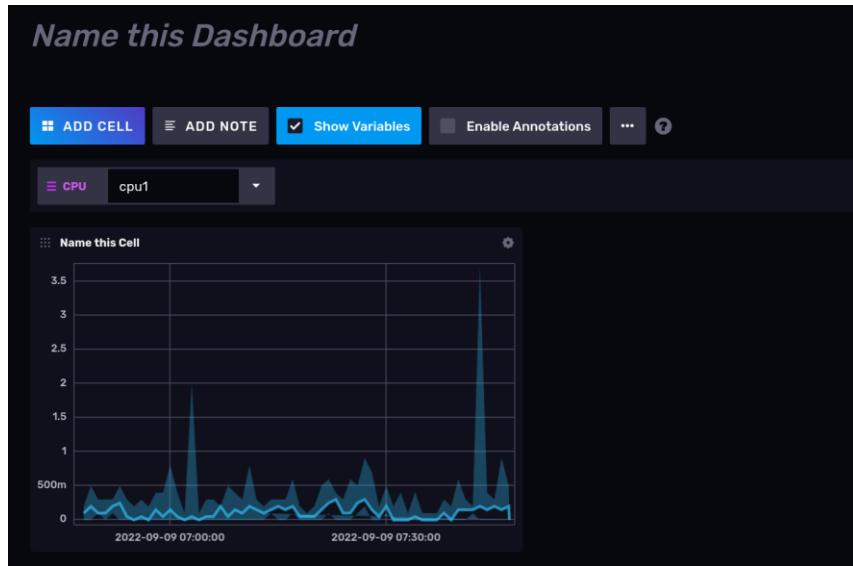
最后，点击右上角的对号保存。

14.3.6 查看效果

可以看到，在仪表盘的顶部出现了一个名为 CPU 的下拉菜单，通过这个下拉菜单，我

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

们可以控制整个仪表盘，但前提是 cell 对应的 FLUX 查询语句引用了我们设置的变量 v.CPU。



使用下拉菜单选择不同的 CPU，可以显示对应的数据。



完成！

14.4 示例：更加灵活的变量与仪表盘

14.4.1 需求

在上一个示例中，我们可以通名为 CPU 的变量对仪表盘中展示的序列进行动态的调整。

但是上一个示例中的仪表盘还有一个缺陷。如图所示，我们每次只能展示一个序列，但是如果我们要对比两个 CPU 的性能差别呢？这个时候上一个示例做出的仪表盘就不够用了。

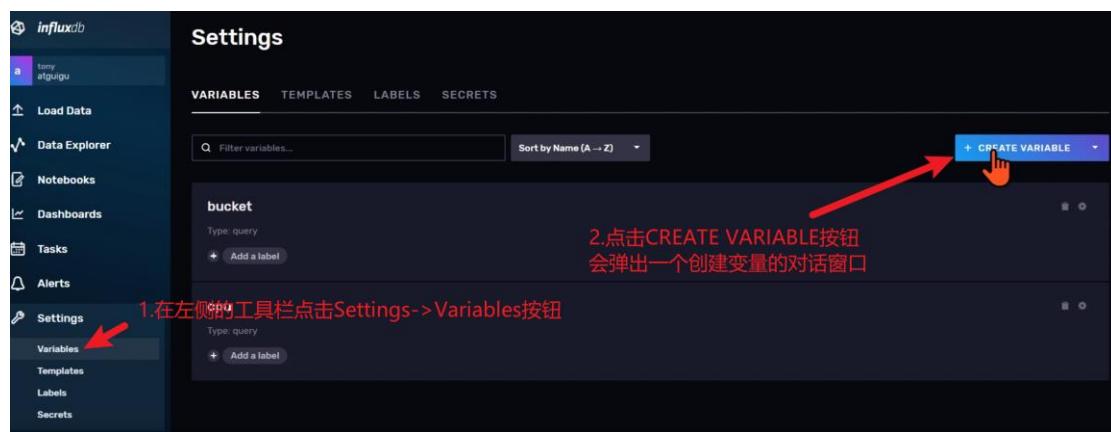
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)



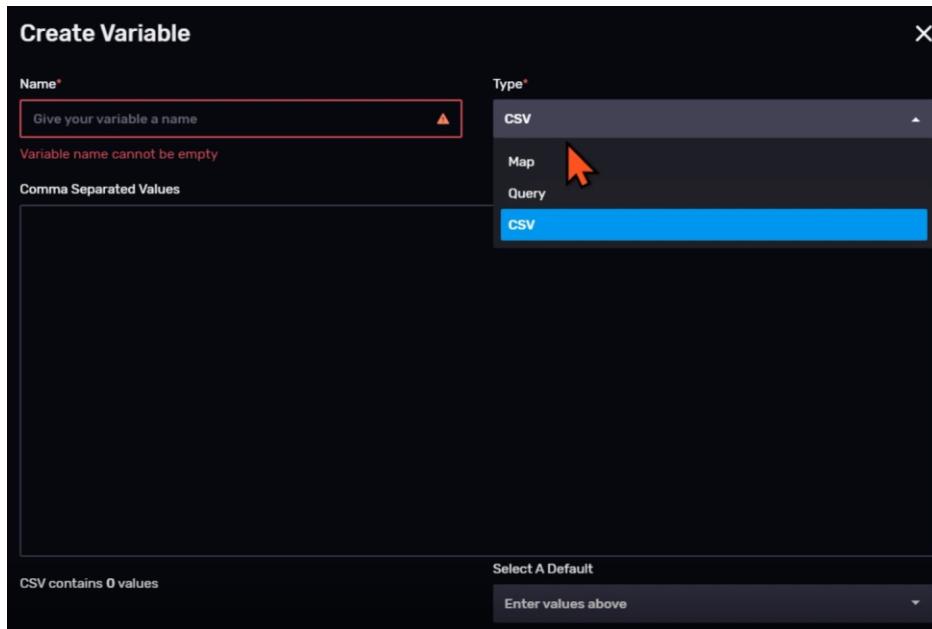
现在，我们希望仪表盘中能够同时显示两个 CPU 到的工作状况，方便我们在视觉上进行对比。

14.4.2 创建变量

(1) 在左侧的工具栏点击 Settings->Variables 按钮。进入到变量的配置页面

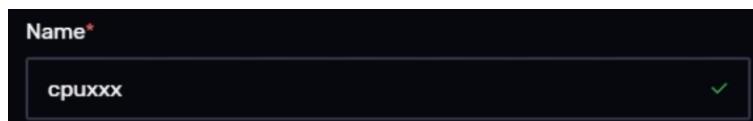


(2) 点击页面右上角的 CREATE VARIABLE 按钮。Web UI 上会弹出一个创建变量的对话窗。



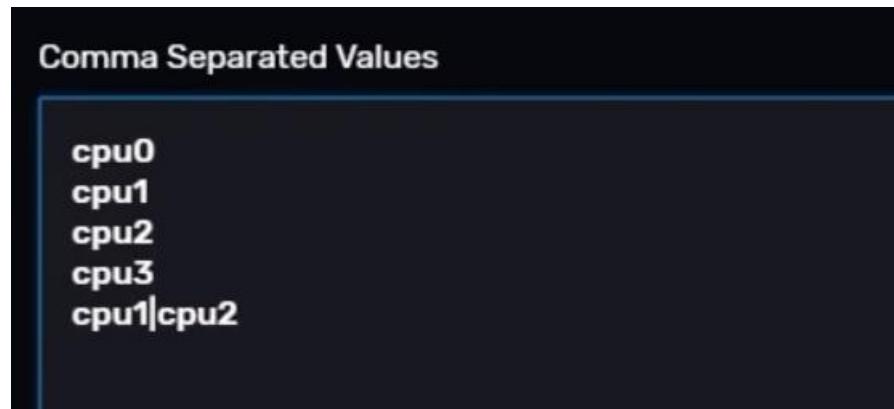
(3) 在右上角的 Type 下拉菜单中选择 CSV。(上一个示例中我们创建的是 Query 类型, Query 类型的变量可以根据数据的状况进行动态的变化。但是另外的 Map 类型和 CSV 类型不行, 它们是静态的, 如果想让其中的值发生改变, 除非再次通过 API 或者 Web UI 对其值进行手动的调整)

(4) 在左上角给变量起好名字, 在演示中, 我们将变量名设为 cpuxxx。

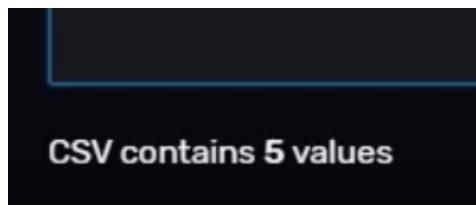


(5) 中间的主要区域是用来设置变量的值的, 这里可以使用 CSV 格式, 但却没有必要非要按照行列的方式来组织这些值。这里的 CSV 格式其实只是要求你用,(英文逗号)来分隔值。其实这个地方也能用换行的方式来分隔值, 如图所示, 老师用的是换行分隔的方式。

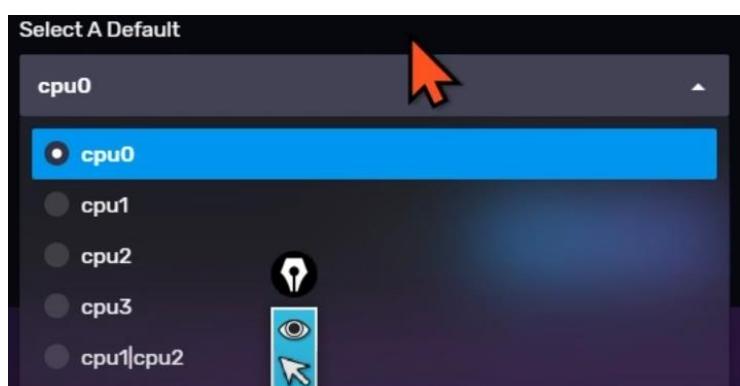
此处, 我们将值设为 cpu0、cpu1、cpu2、cpu3 和 cpu1|cpu2。**注意! 此处的 cpu1|cpu2 是正则表达式写法, 表示 cpu1 或者 cpu2。**



(6) 左下角会实时显示你当前给变量 cpuxxx 设了几种取值。



- 在右下角有一个 Select A Default 下拉菜单，它可以给我们的变量设置一个默认值。此处可以将默认值设为 cpu0。至此，我们的 cpuxxx 就创建好了。



14.4.3 修改 FLUX 脚本（添加正则过滤）

首先，如图所示，先点击齿轮按钮，再在弹出的菜单中点击 Configure 按钮，就可以修改当前的 Cell 了。



现在，我们需要对上一个示例中的查询脚本作一些修改。在 filter 中去添加一个蒸锅 cpuxxx 的取值进行正则过滤的方法。

下面是我们的最终脚本，此处我们就不展示数据的最大值和最小值了。红色的部分是我们需要额外注意的。

```
basedata = from(bucket: "example02")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "cpu")
|> filter(fn: (r) => r["_field"] == "usage_user")
|> filter(fn: (r) =>
  regexp.matchRegexp(r:regexp.compile(v:v.cpuxxx),v:r["cpu"]))

```

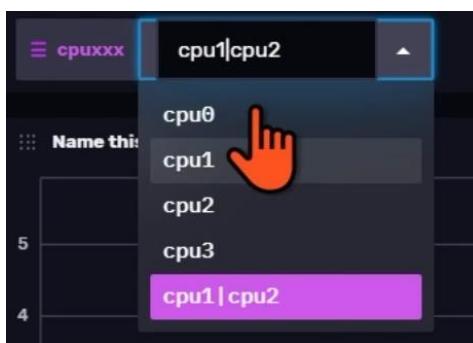
代码解释：

- `regexp.compile(v:v.cpuxxx)`: 需要注意，我们在 InfluxDB 中设置的变量的类型始终都是字符串类型，所以要进行正则匹配的话必须先把字符串转成正则表达式。`regexp` 包下的 `compile` 函数就是专门用来将字符串转为正则表达式的。
- `regexp.matchRegexpString`: 用来判断字符串能否与正则表达式匹配。如果可以匹配上，那么该函数就会返回 `true`，如果匹配不上，那么就会返回 `false`。
- 这样的话，当我们将变量 `cpuxxx` 的值置为 `cpu1|cpu2` 时，就可以同时展示出我们想要的两个序列了。

最终，点击右上角的 \checkmark 按钮保存修改后的 cell。

14.4.4 查看最终效果

回到仪表盘后，可以看到，最上方的变量下拉菜单已经从 `cpu` 变成了 `cpuxxx`，这说明仪表盘会自动判断内部的 cell 用到了哪些变量并做出相应的调整。如下图所示，这就是修改后的效果。



此时，选择 `cpu1|cpu2`，就可以看到之前的 cell 里面会出现两条序列了。



完成！

第15章 InfluxDB 服务进程参数（influxd 命令的用法）

15.1 influxd 命令罗列

我们的 InfluxDB 下载好后，解压目录下的 influxd 就是我们 InfluxDB 服务进程的启动命令。本课程不会介绍 influxd 的全部命令，通过下面的命令列表，大家可以窥探 InfluxDB 的一些可配置的能力。

详情可以参考：<https://docs.influxdata.com/influxdb/v2.4/reference/cli/influx/>

命令	直译	解释
downgrade	降级	将元数据格式降级以匹配旧的发行版
help	帮助	打印 influxd 命令的帮助信息
inspect	检查	检查磁盘上数据库的数据
print-config	打印配置	(此命令 2.4 已被废弃) 打印完整的 influxd 在当前环境的配置信息
recovery	恢复	恢复对 InfluxDB 的操作权限，管理 token、组织和用户
run	运行	运行 influxd 服务 (默认)
upgrade	升级	将 InfluxDB 从 1.x 升级到 InfluxDB2.4
version	版本	打印 InfluxDB 的当前版本

不一定必须通过 influxd 命令来查看 InfluxDB 的当前配置。你还可以使用 influx-cli 的命令：

```
influx server-config
```

15.2 influxd 的两个重要命令

在生产条件下最有可能用到的两个命令就是 inspect 和 recovery。下面，我们对这两个命令做一下详细的介绍。

15.2.1 inspect 命令

你可以使用下面的命令来查看 inspect 这个子命令的帮助信息。

```
./influxd inspect -h
Available Commands:
  build-tsi          Rebuilds the TSI index and (where necessary) the Series File.
  delete-tsm         Deletes a measurement from a raw tsm file.
  dump-tsi           Dumps low-level details about tsi1 files.
  dump-tsm           Dumps low-level details about tsm1 files
  dump-wal           Dumps TSM data from WAL files
  export-index        Exports TSI index data
  export-lp           Export TSM data as line protocol
  report-tsi          Reports the cardinality of TSI files
  report-tsm          Run TSM report
  verify-seriesfile   Verifies the integrity of series files.
  verify-tombstone    Verify the integrity of tombstone files
  verify-tsm           Verifies the integrity of TSM files
  verify-wal           Check for WAL corruption
```

你会发现 inspect 这个子命令下还有很多子命令。

这里出现的 tsi、tsm、wal 都跟 InfluxDB 底层的存储引擎相关，本课程并不涉及这一部分的内容。这里可以稍微点一下，你可以使用下面的命令查看 InfluxDB 中数据存储的大概情况。

```
./influxd inspect report-tsm
```

执行结果如下图所示。

```
[atguigu@host1 influxdb2_linux_amd64]$ ./influxd inspect report-tsm
DB          RP      Shard  File          Series  New (est) Min Time          Max Time          Load Time
67c102a3ce1dbf3c autogen 1  000000012-000000002.tsm 1911  1911  2022-09-16T12:55:49.336368751Z 2022-09-18T23:59:54.534475311Z 467.712µs
9e9cce19f22ae746 autogen 3  000000001-000000001.tsm 1     1     2022-09-17T06:47:14Z 2022-09-17T06:55:15Z 44.768µs
67c102a3ce1dbf3c autogen 73  000000043-000000003.tsm 5936  3489  2022-09-19T00:00:04.532267286Z 2022-09-25T12:38:06.576864759Z 2.053834ms
9e9cce19f22ae746 autogen 74  000000001-000000001.tsm 1     1     2022-09-24T18:50:36.576166376Z 2022-09-24T19:36:06.575722553Z 123.197µs
8a56382f3e2bc62a autogen 165 000000005-000000002.tsm 4     4     2022-09-23T05:46:22.996Z 2022-09-23T15:11:17.291Z 118.001µs
55098ee282b4b2ca autogen 174 000000001-000000001.tsm 1     1     2022-09-23T13:39:05.929112818Z 2022-09-23T13:39:05.929112818Z 115.116µs
2a16bf3d23fb7c7d1 autogen 196 000000006-000000002.tsm 138   138   2022-09-24T09:45:00.010057758Z 2022-09-24T21:26:35.003999875Z 95.51µs
b6a24a92ec038131 autogen 204 000000001-000000001.tsm 289   289   2022-09-25T11:17:10Z 2022-09-25T11:59:55Z 135.488µs
b6a24a92ec038131 autogen 205 000000001-000000001.tsm 289   0     2022-09-25T12:00:00Z 2022-09-25T12:59:55Z 67.569µs
b6a24a92ec038131 autogen 206 000000001-000000001.tsm 289   0     2022-09-25T13:00:00Z 2022-09-25T13:59:55Z 286.496µs
b6a24a92ec038131 autogen 207 000000001-000000001.tsm 289   0     2022-09-25T14:00:00Z 2022-09-25T14:59:55Z 55.586µs
b6a24a92ec038131 autogen 208 000000001-000000001.tsm 289   0     2022-09-25T15:00:00Z 2022-09-25T15:59:55Z 53.308µs

Summary: Files: 12
Time Range: 2022-09-16T12:55:49.336368751Z - 2022-09-25T15:59:55Z
Duration: 219h4m5.663631249s

Statistics
  Series:
    - 2a16bf3d23fb7c7d1 (est): 130 (2%)
    - b6a24a92ec038131 (est): 289 (4%)
    - 67c102a3ce1dbf3c (est): 5408 (92%)
    - 9e9cce19f22ae746 (est): 2 (0%)
    - 8a56382f3e2bc62a (est): 4 (0%)
    - 55098ee282b4b2ca (est): 1 (0%)
  Total (est): 5826
Completed in 23.183631ms
[atguigu@host1 influxdb2_linux_amd64]$
```

展示出来的信息中包含了 InfluxDB 的数据存储情况，比如当前整个 InfluxDB 有多少序列，每个存储桶中又有多少序列等等。

另外，还有一个比较重要的 export-tsm 命令，它可以将某个存储桶中的数据全部导出为 InfluxDB 行协议。后面我们会在一个示例中详细演示它的使用。

15.2.2 recovery 命令

recovery 是恢复的意思。

可以先用下面的命令查看 recovery 这一子命令的帮助信息。

```
./influxd recovery -h
```

如图所示，influxd recovery 命令的作用主要是用来修复或者重新生成对 InfluxDB 进行操作所需的 operator（操作者）权限的。

```
[atguigu@host1 influxdb2_linux_amd64]$ ./influxd recovery -h
Commands used to recover / regenerate operator access to the DB

Usage:
  influxd recovery [flags]
  influxd recovery [command]

Available Commands:
  auth      On-disk authorization management commands, for recovery
  org       On-disk organization management commands, for recovery
  user      On-disk user management commands, for recovery

Flags:
  -h, --help   help for recovery
```

recovery 下面还有 3 个子命令，分别是 auth、org 和 user。它们分别与 token、组织和用户有关。

下面主要是讲解 auth 子命令的用法，使用下面的命令可以进一步查看 auth 子命令的帮助信息。

```
./influxd recovery auth -h
```

返回的结果如下图所示：

```
[atguigu@host1 influxdb2_linux_amd64]$ ./influxd recovery auth -h
On-disk authorization management commands, for recovery

Usage:
  influxd recovery auth [flags]
  influxd recovery auth [command]

Available Commands:
  create-operator Create new operator token for a user
  list           List authorizations

Flags:
  -h, --help   help for auth
```

可以看到它有两个子命令。

- **create-operator:** 为一个用户创建一个新的操作者 token。
- **list:** 列出当前数据库中的全部 token。

使用下面的命令就可以为 tony 用户再次创建一个 operator-token 了。

```
./influxd recovery auth create-operator --username tony --org
atguigu
```

命令执行后，终端会显示如下图所示的内容，可以看到这里创建了一个名为 tony's Recovery Token 的操作者 token。

```
0a149c67972e3000      tony      09fd705e55488000      tony's Recovery Token      2iVsstZTAh
o-rI8re_Mw02F0AFicngJnC3Ms53y57LtZMD0y0Ydkz5aFv2J-u-BpZANU8Ld7MG27a1wBP6KUQ==  [read:authorizations write:authorizations read:buckets write:buckets
e:buckets read:dashboards write:dashboards read:orgs write:orgs read:sources write:sources read:tasks write:tasks read:telegraf
s read:views write:views read:documents write:documents read:notificationRules write:notificationRules read:notificationEndpoints write:notificationEndpoints
read:checks write:checks read:dbrp write:dbrp read:notebooks write:notebooks read:annotations write:annotations read:remotes
write:remotes read:replications write:replications]
```

15.3 influxd 常用配置项

influxd 的可用配置项超多，本课程不会全部讲解。

详细可以参考：<https://docs.influxdata.com/influxdb/v2.4/reference/config-options/#assets-path>

以下是一些常用的参数

- bolt-path: BoltDB 文件的路径。
- engine-path: InfluxDB 文件的路径
- sqlit-path: sqlite 的路径，InfluxDB 里面还用到了 sqlite，它里面会存放一些关于任务执行的元数据，
- flux-log-enabled: 是否开启日志，默认是 false。
- log-level: 日志级别，支持 debug、info、error 等。默认是 info。

15.4 如何对 influxd 进行配置

有 3 种方式可以对 influxd 的配置。这里以 http-bind-address 进行操作，为大家演示。

15.4.1 命令行参数

进行如下操作前，记得关闭当前正在运行的 influxd。你可以使用下面的命令来杀死当然的 influxd 进程。否则，原先的 influxd 进程会锁住 BoltDB 数据库，别的进程不能访问。当然你也可以修改 BoltDB 路径，但是那样太过麻烦。

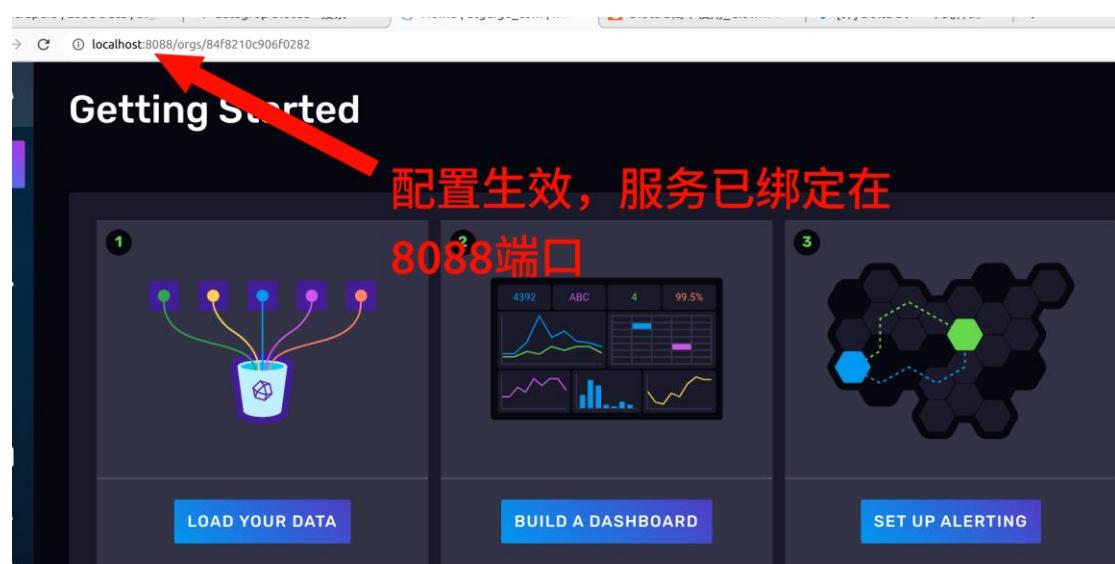
```
ps -ef | grep influxd | grep -v grep | awk '{print $2}' | xargs kill
```

用户 influxd 命令启动 InfluxDB 时，通过命令行参数来传递一个配置项。

比如：

```
./influxd --http-bind-address=:8088
```

可以尝试访问 8088 端口，看服务有没有挂到端口上



15.4.2 环境变量

同样，还是先杀死之前的 influxd 进程。

运行下面的命令。

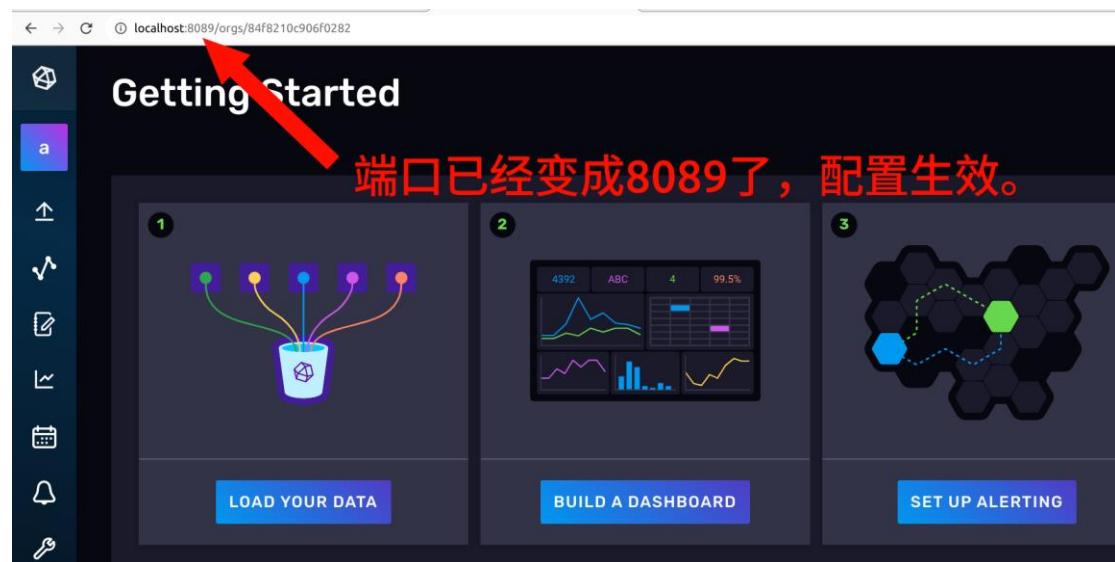
```
ps -ef | grep influxd | grep -v grep | awk '{print $2}' | xargs kill
```

用户可以声明一个环境变量，对 influxd 进行配置

比如：

```
export INFLUXD_HTTP_BIND_ADDRESS=:8089
```

现在，我们启动一下 influxd 看下效果。



最后，因为我们用的是 export 命令，临时搞了一个环境变量，如果你觉得当前 shell 会话不重要，可以关闭当前 shell 会话。否则，你可以使用 unset 命令来销毁这个环境变量。

```
unset INFLUXD_HTTP_BIND_ADDRESS
```

15.4.3 配置文件

你还在 influxd 所在的目录下放一个 config 文件，它可以是 config.json, config.toml, config.yaml。这 3 种格式 influxd 都能识别，不过文件中的内容一定要合法。influxd 启动时会自动检测这个文件。

在 InfluxDB 的安装目录下创建一个 config.json 文件。

```
vim /opt/module/influxdb2_linux_amd64/config.json
```

编辑如下内容。

```
{  
    "http-bind-address": ":9090"  
}
```

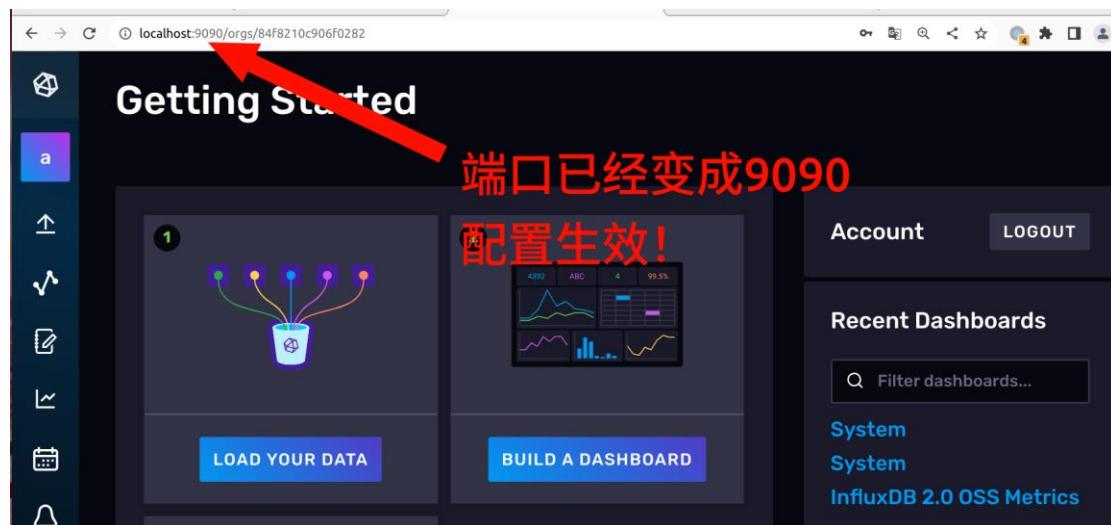
启动之前记得停掉之前的 InfluxDB 进程。

```
ps -ef | grep influxd | grep -v grep | awk '{print $2}' | xargs kill
```

现在再启动一下，看看效果。

```
./influxd
```

可以看到端口已经变成 9090。配置同样是生效的。



15.4.4 小结

最后，如果要做配置的修改，建议一定要参考 InfluxDB 的官方文档，这一部分写的非常清楚，而且官网已经给出了进行配置的各种模板。用好官方文档，可以大大提高开发效率。

The screenshot shows the InfluxDB configuration interface for the `http-idle-timeout` parameter. Red annotations explain the following:

- A red arrow points from the text "配置参数" (Configuration Parameters) to the parameter name `http-idle-timeout`.
- A red arrow points from the text "参数解释" (Parameter Description) to the description text below the parameter.
- A red arrow points from the text "默认值" (Default Value) to the default value `3m0s`.
- A red arrow points from the text "命令行参数该怎么传" (How to pass command-line parameters) to the `influxd flag` section.
- A red arrow points from the text "环境变量怎么设" (How to set environment variables) to the `Environment variable` section.
- A red arrow points from the text "配置文件怎么写" (How to write configuration files) to the `Configuration file` section.
- A red box highlights the `YAML`, `TOML`, and `JSON` options under the configuration file section, with a red arrow pointing to the text "三种配置文件写法, 可以切换" (Three ways to write configuration files, can be switched).

第16章 时序数据库是怎么存储用户名和密码的

InfluxDB 内部自带了一个用 Go 语言写的 BlotDB，BlotDB 是一个键值数据库，它的功能比较有限，基本上就是专注于存值、读值。同时，因为功能有限，它也可以做的很小很轻量。

InfluxDB 就是把用户名、密码、token 什么的信息存在这样的键值数据库里的。默认情况下，BlotDB 的数据会存储在一个单独的文件中，这个文件会在`~/.influxdbv2/` 路径下，名称为 `influxd.bolt`。

这个文件的路径可以在 `influxd` 通过 `bolt-path` 配置项来进行修改。

第17章 从 InfluxDB OSS 迁移数据

17.1 将 InfluxDB 中的数据导出

导出 InfluxDB 数据必须使用 influxd 命令（注意，不是 influx 命令）。

在 InfluxDB2.x 中，数据导出是以存储桶为单位的。

下面是示例命令：

```
influxd inspect export-lp \
--bucket-id 12ab34cd56ef \
--engine-path ~/.influxdbv2/engine \
--output-path path/to/export.lp
--start 2022-01-01T00:00:00Z \
--end 2022-01-31T23:59:59Z \
--compress
```

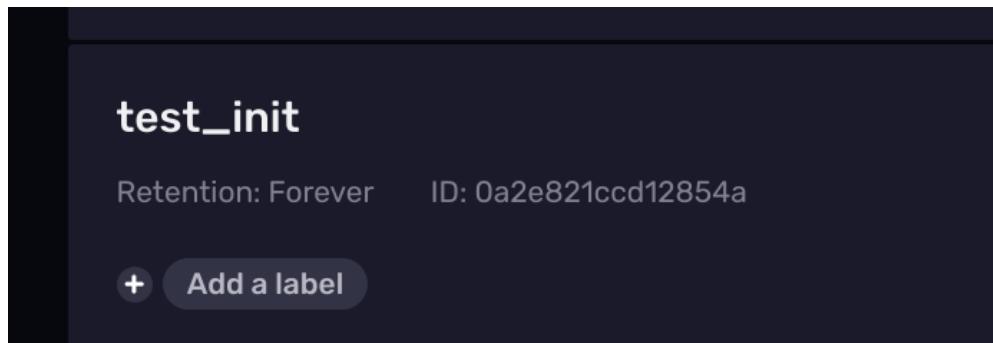
参数讲解：

- influxd inspect, influxd 是可以操作 InfluxDB 服务进程的命令行工具，inspect 是 influxd 命令的子命令，使用 inspect 可以
- export-lp, 是 export xxx to line protocol 的缩写，表示将数据导出为行协议。它是 inspect 的子命令。
- bucket-id, **inspect 的必须参数**。存储桶的 id
- engine-path, **inspect 的必须参数**，不过有默认值~/.influxdbv2/engine。所以如果你的数据目录是~/.influxdbv2/engine 那么不指定这个参数也行。
- output-path, **inspect 的必须参数**，指定输出文件的位置。
- start, 非必须，导出数据的开始时间
- end, 非必须，导出数据的结束时间。
- compress, 建议启用，如果启用了，那么 influxd 会使用 gzip 的方式压缩输出的数据。

17.2 示例：将 InfluxDB 中的数据导出

这次，我们尝试导出 test_init 的数据导出，截至目前，这个 bucket 里面的数据应该是当前最多的。

(1) 首先，你可以使用 influx-cl 也可以使用 Web UI 来查看我们想要导出的 bucket 对应的 ID。这里，课程选择使用 Web UI，可以看到 test_init 存储桶的 ID 为 0a2e821cc12854a。



(2) 于是，我们运行下面的命令，尝试把数据导出。

```
./influxd inspect export-lp \
--bucket-id 0a2e821ccd12854a \
--output-path ./oh.lp
```

这条命令会把 test_init 存储桶里的数据以 InfluxDB 行协议的格式导出到当前目录下的 oh.lp 文件中。

正常情况下，程序会输出一系列读写信息。

```
二 30 8月 - 18:57 /opt/module/influxdb2_linux_amd64
[atguigu] ./influxd inspect export-lp --bucket-id 0a2e821ccd12854a \
--output-path ./oh.lp
[{"level": "info", "ts": "1661857044.301934", "caller": "export_lp/export_lp.go:219", "msg": "exporting TSM files", "tsm_dir": "/home/dengziqi/.influxdbv2/engine/data/0a2e821ccd12854a", "file_count": 3}
[{"level": "info", "ts": "1661857047.3002422", "caller": "export_lp/export_lp.go:315", "msg": "exporting WAL files", "wal_dir": "/home/dengziqi/.influxdbv2/engine/wal/0a2e821ccd12854a", "file_count": 1}
[{"level": "info", "ts": "1661857047.9432976", "caller": "export_lp/export_lp.go:204", "msg": "export complete"}]
```

(3) 使用下面的命令查看当前路径下的文件及其大小。

```
ls -lh
```

ls 的 h 参数，可以将文件的字节数打印为更容易阅读的 MB、GB 单位。

```
二 30 8月 - 18:57 /opt/module/influxdb2_linux_amd64
[atguigu] ls -lh
总用量 1.6G
-rwxr-xr-x 1 dengziqi dengziqi 144M 8月 19 03:44 influxd
-rw-rw-r-- 1 dengziqi dengziqi 1.1K 8月 19 03:44 LICENSE
-rw-rw-r-- 1 dengziqi dengziqi 1.5G 8月 30 18:57 oh.lp
-rw-rw-r-- 1 dengziqi dengziqi 9.6K 8月 19 03:44 README.md
-rw-rw-r-- 1 dengziqi dengziqi 1.3K 8月 26 12:24 selfsigned.crt
-rw----- 1 dengziqi dengziqi 1.7K 8月 26 12:24 selfsigned.key
```

可以看到，我们导出的数据文件 oh.lp 有 1.5G 大小。

(4) 现在，我们使用 tail 命令来查看一下文件的内容。

```
tail -15 ./oh.lp
```

命令输出的是文件的最后 15 行内容，可以看到里面全是 InfluxDB 行协议的数据。

```
[-- 4月 08 - 19:02 /opt/module/influxdb_2.1/linux_and64
root@host: ~]# tail -15 /joh.log

service_bucket_new_duration.methods/find_buckets 0.01=0 1661857045934934887
query_influxdb_read_request_duration_seconds,op=readAggregates,org_id=g4f8210c90f6f0282 sum=0.00164263099999999999 1661857045934934887
http_api_request_duration_seconds.handlersplatform,method=GET,path=/file_name.svg,response_code=200,status=2XX,user_agent=Chrome 0.01=0 1661857045934934887
storage_shard_series.buckets@010e9f39551fc7f,engines1n,ld23,paths/home/dengziqi/.influxdbv2/engine/data@010e9f39551fc7f/autogen/23,walPaths/home/dengziqi/.influxdbv2/engine/wal@010e9f39551fc7f/autogen/23,gauge=1 1661857045934934887
34887
http_api_request_duration_seconds.handlersplatform,method=GET,path=/file_name.svg,response_code=200,status=2XX,user_agent=Chrome sum=0.023169561 1661857045934934887
task_scheduler_execute_dels.sum#0 1661857045934934887
service_password_new_duration.methods/compare_password@0.13=0 1661857045934934887
http_api_request_duration_seconds.handlersplatform,method=GET,path=/v2/buckets,response_code=200,status=2XX,user_agent=Chrome count=9 1661857045934934887
storage_shard_write_dropped_sum,bucket@0a2e82cc1d2854a,engines1n,ld75,paths/home/dengziqi/.influxdbv2/engine/data@0a2e82cc1d2854a/autogen/75,walPaths/home/dengziqi/.influxdbv2/engine/wal@0a2e82cc1d2854a/autogen/75,counter=0 1
storage_shard_write_dropped_sum,bucket@0a2e82cc1d2854a,engines1n,ld75,paths/home/dengziqi/.influxdbv2/engine/data@0a2e82cc1d2854a/autogen/75,walPaths/home/dengziqi/.influxdbv2/engine/wal@0a2e82cc1d2854a/autogen/75,counter=0 1
storage_wal_user_writes.bucket@010e9f39551fc7f,engines1n,ld23,paths/home/dengziqi/.influxdbv2/engine/data@010e9f39551fc7f/autogen/23,walPaths/home/dengziqi/.influxdbv2/engine/wal@010e9f39551fc7f/autogen/23,counter=0 1661857045934934887
34887
http_api_request_duration_seconds.handlersplatform,method=POST,path=/api/v2/signin,response_code=204,status=2XX,user_agents=Chrome 0.01=0 1661857045934934887
service_org_new_duration.method=find_org_by_id_to_0.025=0 1661857045934934887
service_org_duration.method=find_labels_for_resource@0.025=0 1661857045934934887
service_user_new_duration.method=find_user_by_id@0.025=0 1661857045934934887
service_user_new_duration.method=find_user_by_id@0.025=0 1661857045934934887
go_memonstat_msapan_inuse_bytes.gauge1@0.02126e-06 1661857045934934887
```

不过，我们要注意到 InfluxDB 行协议的一个特点，其实对于整个文件来说，多条数据的 measurement 其实是重复的，tagset 的重复率也不低，filed 的变化也不会很大。这种高度重复的数据其实是非常适合压缩算法的。

17.3 示例：导出数据时压缩

(1) 现在，我们重新运行数据导出的命令，这次在命令的最后加上--compress 参数。

```
./influxd inspect export-lp \
--bucket-id 0a2e821ccd12854a \
--output-path ./oh.lp
--compress
```

不必担心目录下已经存在 oh.lp 文件，程序会直接将其覆盖的。

(2) 使用 ls 命令再次查看文件大小。

ls -lh

```
二 30 8月 - 19:10 > /opt/module/influxdb2_linux_amd64 >
@atguigu > ls -lh
总用量 234M
-rwxr-xr-x 1 dengziqi dengziqi 144M 8月 19 03:44 influxd
-rw-rw-r-- 1 dengziqi dengziqi 1.1K 8月 19 03:44 LICENSE
-rw-rw-r-- 1 dengziqi dengziqi 91M 8月 30 19:10 oh.lp
-rw-rw-r-- 1 dengziqi dengziqi 9.6K 8月 19 03:44 README.md
-rw-rw-r-- 1 dengziqi dengziqi 1.3K 8月 26 12:24 selfsigned.crt
-rw----- 1 dengziqi dengziqi 1.7K 8月 26 12:24 selfsigned.key
```

可以看到文件从之前的 1.5G 变成了现在的 91M，压缩率非常高。

第18章 FLUX 查询优化

18.1 使用谓词下推的查询

谓词下推常见于 SQL 查询中，一个 SQL 中的谓词，通常指的是 where 条件。

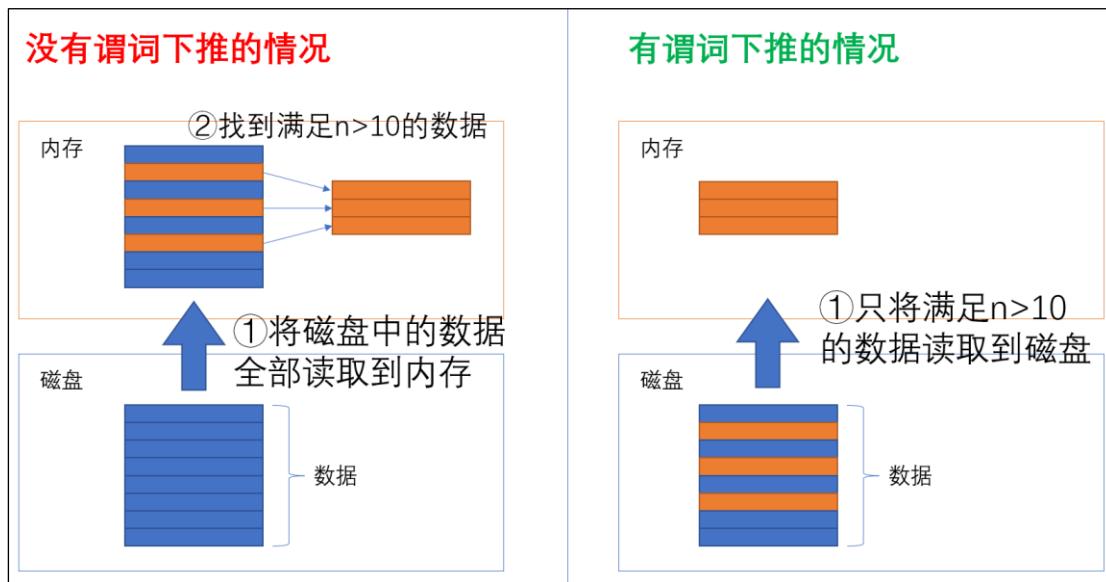
我们看一个最简单的 SQL 语句。它从一个名为 A 的表中查询数据，并按照 $n > 10$ 的条件对数据进行过滤。

```
select *  
from A
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
where n > 10
```

你可以想象一下这条 SQL 语句在计算机中的执行流程。大体上有下面两种方式。



- 一种是将磁盘里的数据全部读到内存中，再在内存中进行过滤。这种方式我们通常说它没有内存下推
- 另一种是在查询时，就只从磁盘取自己需要的数据到内存中，再进行下一步的操作。通常，我们说这种方式实现了谓词下推。

虽然说 FLUX 语言表面上是一个脚本语言，但在查询这件事上，它并不是老老实实一行行执行的，而是有了优化器的参与。FLUX 语言在执行时，会尽可能实现谓词下推的优化，什么样的查询可以实现谓词下推，可以参考官网文档的优化查询一节
<https://docs.influxdata.com/influxdb/v2.4/query-data/optimize-queries/>

Pushdown functions and function combinations

Most pushdowns are supported when querying an InfluxDB 2.4 or later version. In the following table, a handful of pushdowns are not supported in InfluxDB Cloud.

Functions	InfluxDB 2.4	InfluxDB Cloud
count()	✓	✓
duplicate()	✓	✓
filter() *	✓	✓
fill()	✓	✓
first()	✓	✓
last()	✓	✓
max()	✓	✓
mean()	✓	✓
min()	✓	✓

另外，后面我们会告诉大家如何去查看一个查询的执行计划。

18.2 避免将窗口宽度设得过小

窗口（基于时间间隔对数据进行分组）通常用于聚合和降采样数据。将窗口设长一点可以提高性能。窗口过窄会导致需要更多的算力来评估每条数据应该分配到哪个窗口，合理的窗口宽度应该根据查询的总时间宽度来决定。

18.3 避免使用“沉重”的功能

下面的这些函数对于 FLUX 来说会比较重，这些函数会使用更多的内存和 CPU，使用这些函数时要想要是否必要。

- map()
- reduce()
- join()
- union()
- pivot()

不过官方又说，InfluxData 一直在优化 FLUX 的性能，所以当前的列表不一定是将来更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

的情况

18.4 尽可能使用 set () 而不是 map ()

如果你要给数据查一个静态常量，那么 set 比 map 要有很大的性能优势。map 是我们上一小节说的沉重操作。在后面的示例，我们会比较两种操作的差距。

18.5 平衡数据的时间范围和数据精度

想要保证查询的性能良好，应该平衡好查询的时间范围和数据精度。如果，有一个 measurement 的数据每秒入库一条，你一次请求 6 个月的数据，那么一个序列就能包含 1550 万点数据。如果序列数再多一些，那么数据很可能会变成数十亿点。Flux 必须将这些数据拉到内存再返回给用户。所以一方面做好谓词下推尽量减少对内存的使用。另外，如果必须要查询很长时间范围的数据，那应该创建一个定时任务来对数据进行降采样，然后将查询目标从原始数据改为降采样数据。

18.6 使用 FLUX 性能分析工具查看查询性能

执行 FLUX 查询时，你可以导入一个名为 profiler 的包，然后添加一个 option 选项以查看当前 FLUX 语句的执行计划。比如：

```
option profiler.enabledProfilers = ["query", "operator"]
```

这里的 query 和 operator 是查询计划的两个选项，query 表示你要查看整个执行脚本的执行情况，operator 表示你要查看一个 FLUX 查询各个算子的执行情况。

18.6.1 query (查询)

query 提供有关整个 Flux 脚本执行的统计信息。启用后，结果将多出一个表，其中包括以下信息：

- TotalDuration: 查询总持续时间（以纳秒为单位）
- CompileDuration: 编译查询脚本所花费的时间（以纳秒为单位）
- QueueDuration: 排队所花费的时间（以纳秒为单位）
- RequeueDuration: 重新排队花费的时间（以纳秒为单位）
- PlanDuration: 计划查询所花费的时间（以纳秒为单位）
- ExecuteDuration: 执行查询所花费的时间（以纳秒为单位）
- Concurrency: 并发，分配给处理查询的 goroutines。
- MaxAllocated: 查询分配的最大字节数（内存）
- TotalAllocated: 查询时分配的总字节数（包括释放然后再次使用的内存）

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- RuntimeErrors: 查询执行期间返回的错误消息
- flux/query-plan: flux 查询计划
- influxdb/scanned-values: 数据库扫描磁盘的数据条数
- influxdb/scanned-bytes: 数据库扫描磁盘的字节数

18.6.2 operator (算子)

有关一个查询脚本中每个操作的统计信息。在存储层中执行的操作将作为单个操作返回。启用此配置后，返回的结果将多出一个表，并包含以下内容

- Type: 操作类型
- Label: 标签
- Count: 执行这个操作的总次数
- MinDuration: 操作被执行多次中，最快的一次花费的时间（以纳秒为单位）
- MaxDuration: 操作被执行多次中，最慢的一次花费的时间（以纳秒为单位）
- DurationSum: 当前操作完成的总持续时间（以纳秒为单位）。
- MeanDuration: 操作被执行多次的平均持续时间（以纳秒为单位）。

18.7 示例：使用 profile 优化查询

18.7.1 编写查询

首先，打开 DataExplorer。写下如下代码。

```
from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go goroutines")
|> map(fn: (r) => ({r with hello:"world"}))
```

这段代码，会从 test_init 存储桶查询一个名为 go_goroutines 的 measurement，这个测量下没有 tag，所以我们只有一个序列。

map 函数帮我们在 filter 后的数据上加了一个常量列，列名是 hello，值是字符串 world。

18.7.2 执行查询

现在，SUBMIT 一下上面的代码，然后点击 View Raw Data。数据应该如下所示。

我们的常量列

```

1 from(bucket: "test_init")
2 |> range(start: -1h)
3 |> filter(fn: (r) => r["_measurement"] == "go_goroutines")
4 |> map(fn: (r) => ({r with hello:"world"}))

```

可以看到有一个常量列。

18.7.3 修改代码查看性能指标和执行计划

现在，我们对代码做出修，以方便我们观察执行计划。

```

import "profiler"

option profiler.enabledProfilers = ["query", "operator"]

from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> map(fn: (r) => ({r with hello:"world"}))

```

代码解释：

- option profiler.enabledProfilers，这其实是个开关选项，当后面的列表出现“query”时，就显示整个查询的性能和执行计划，当然出现“operator”时，会具体显示每个算子的性能指标。

- import "profiler"，引包，enabledProfilers 是 profiler 中的开关，不引没法用。

现在，再次点击 SUBMIT，同样还是观察原始数据。一上来展示的还是我们查询出来的数据，需要切换到页尾才能看到性能指标和执行计划。

Query 1 (0.02s)

```

1 import "profiler"
2
3 option profiler.enabledProfilers = ["query", "operator"]
4
5 from(bucket: "test_init")
6 |> range(start: -1h)
7 |> filter(fn: (r) => r["_measurement"] == "go goroutines")
8 |> map(fn: (r) => ({r with hello:"world"}))

```

table	_measurement	_field	_value	_start	_stop	_time	hello
_RESULT	GROUP STRING	GROUP STRING	NO GROUP DOUBLE	GROUP DATETIME:RFC3339	GROUP DATETIME:RFC3339	NO GROUP DATETIME:RFC3339	NO GROUP STRING
0	go goroutines	gauge	1301	2022-09-05T17:32:44.126Z	2022-09-05T18:32:44.126Z	2022-09-05T17:32:52.038Z	world
0	go goroutines	gauge	1301	2022-09-05T17:32:44.126Z	2022-09-05T18:32:44.126Z	2022-09-05T17:33:02.037Z	world

1 2 3 4 5 ... 182

View Raw Data CSV Past 1h QUERY BUILDER SUBMIT

Filter Functions... Transformations

aggregate.rate chandeMomentumOsci...

现在，我们看到两张表，有一个_measurement 为 profiler/query，这是我们整个查询的性能指标和执行计划。还有一个_measurement 为 profiler/operator 的，这是我们每个算子的性能指标。里面包括某个算子运行了多长时间等信息。

table	_measurement	CompileDuzation	Concurrency	ExecuteDuration	flux/query-plan
_PROFILER	GROUP STRING	NO GROUP LONG	NO GROUP LONG	NO GROUP LONG	NO GROUP STRING
0	profiler/query	182810	0	6644771	digraph { "merged_ReadRange4_filter2" "map3" "merged_ReadRange4_filter2" -> "map3" }

table	_measurement	Count	DurationSum	Label	MaxDuration	MeanDuration	MinDuration	Type
_PROFILER	GROUP STRING	NO GROUP LONG	NO GROUP LONG	NO GROUP STRING	NO GROUP LONG	NO GROUP DOUBLE	NO GROUP LONG	NO GROUP STRING
1	profiler/operator	1	3425879	merged_ReadRange4_filter2	3425879	3425879	3425879	*influxdb.readFilterSource
1	profiler/operator	3	3273240	map3	3229716	1091080	1331	*universe.mapTransformation

18.7.4 如何判断谓词下推

按照官方文档的说法，如果实现了谓词下推，那么多个 operator 会合并成一个。我们可以看到现在操作列表里，有一个叫做 merged_ReadRange4_filter2 的算子操作，后面紧跟的是我们的 map 操作。这说明 from -> range -> filter 被合并了。它们是一步操作。

table	_measurement	Count	DurationSum	Label	MaxDuration	MeanDuration	MinDuration	Type
_PROFILER	GROUP STRING	NO GROUP LONG	NO GROUP LONG	NO GROUP STRING	NO GROUP LONG	NO GROUP DOUBLE	NO GROUP LONG	NO GROUP STRING
1	profiler/operator	1	3551998	merged_ReadRange4_filter2	3551998	3551998	3551998	*influxdb.readFilterSource
1	profiler/operator	3	3387871	map3	3337987	1129290.333333333	1238	*universe.mapTransformation

1 69 70 71 72 73

View Raw Data CSV Past 1h QUERY BUILDER SUBMIT

Filter Functions... Transformations

aggregate.rate

18.7.5 查看查询性能

查询性能有很多指标，但是我们现在只关注两个指标，一个是 MaxAllocated。它表示的是我们为了完成查询，总共使用过的内存（包含释放后又申请的内存）。

es MaxAllocated	
NO GROUP	LONG
52736	

现在这个指标的数值是 52736，也就是说为了完成这个查询，我们前后用了大概 50kb 的内存。

另一个是 TotalDuration，表示执行这个操作的总时间，现在是 17365972 纳秒

TotalDuration	
NO GROUP	LONG
17365972	

18.7.6 在 map 后增加 AggregateWindow

现在，我们在 map 后面加上一个 AggregateWindow 函数。

整体代码如下：

```
import "profiler"

option profiler.enabledProfilers = ["query", "operator"]

from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> map(fn: (r) => ({r with hello:"world"}))
|> aggregateWindow(column: "_value", every: 1h, fn:mean)
```

红色的部分是我们新增的代码。

18.7.7 查看查询性能

首先关注我们的 operator 表，可以看到，操作数从之前的 2 个变成了 3 个。map 的后面，多了一个聚合窗口操作。

table	_measurement	Count	DurationSum	Label	MaxDuration	MeanDuration
_PROFILER	GROUP	NO GROUP	NO GROUP	NO GROUP	NO GROUP	NO GROUP
1	profiler/operator	1	3979552	merged_ReadRange10_filter2	3979552	3979552
1	profiler/operator	3	3708401	map3	3656369	1236133.6666666667
1	profiler/operator	3	47395	aggregateWindow9	45632	15798.333333333334

查询的 MaxAllocated 依然是 52736，没有变。

MaxAllocated
NO GROUP
LONG
52736

因为之前需要返回几百条数据，现在开窗聚合，只需要返回两条数据，所以查询的持续时间有所缩短。

TotalDuration
NO GROUP
LONG
7702588

18.7.8 将 AggregateWindow 移至 map 前

现在，我们将 AggregateWindow 移到 map 之前 filter 之后。修改后的代码整体如下：

```
import "profiler"

option profiler.enabledProfilers = ["query", "operator"]

from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> aggregateWindow(column: "_value", every: 1h, fn:mean)
|> map(fn: (r) => ({r with hello:"world"}))
```

注意红色是修改的地方。

18.7.9 查看查询性能

首先还是关注 operator 表，这张表里之前是 3 个操作，现在变成了两个。map 之前，有一个 ReadWindowAggregateByTime 操作。也就是说，我们的 aggregateWindow 操作实现了谓词下推。

table	_measurement	Count	DurationSum	Label	MaxDuration	MeanDuration
_PROFILER	GROUP STRING	NO GROUP LONG	NO GROUP LONG	NO GROUP STRING	NO GROUP LONG	NO GROUP DOUBLE
1	profiler/operator	1	605598	ReadWindowAggregateByTime11	605598	605598
1	profiler/operator	3	264038	map8	224315	88012.6666

```

Query 1 (0.01s) + ? View Raw Data ⚡

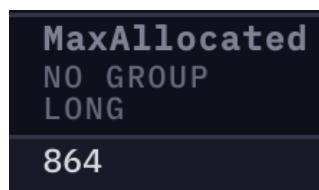
4
5 from(bucket: "test_init")
6   . |> range(start: -1h)
7   . |> filter(fn: (x) => x["_measurement"] == "go goroutines")
8   . |> aggregateWindow(column: "_value", every: 1h, fn: mean)
9   . |> map(fn: (x) => ({x with hello:"world"}))

```

当读磁盘的操作完成后，内存中只会存在聚合后的两条数据。

现在我们关注查询性能指标。

可以看到 MaxAllocated 变成了 864，之前这一指标的数值还是 52736。之前要消耗 50KB，现在却 1KB 都不到。



查询的持续时间也有进一步缩短。



18.7.10 将 map 改为 set

最后值得说一下，我们的 map 操作数据的原理是对数据集中的数据一行一行处理。此处，我们用它实现了添加常量的功能。其实还有一个同样能完成此类任务的算子叫 set，它操作数据的逻辑是直接操作整个数据集。

数据量越大，这两个算子的性能差距就越明显。

此处，我们将 aggregateWindow 算子去掉，并将 map 改成 set。与第一次查看性能时的代码做比较，当时我们还没有做聚合操作。

改完的代码整体如下：

```

import "profiler"

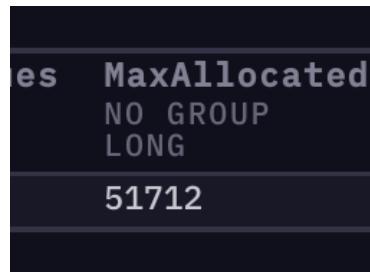
option profiler.enabledProfilers = ["query", "operator"]

```

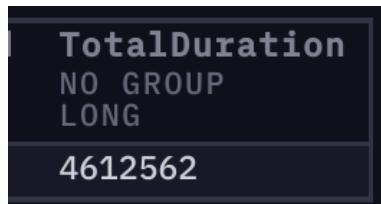
```
from(bucket: "test_init")
|> range(start: -1h)
|> filter(fn: (r) => r["_measurement"] == "go_goroutines")
|> set(key: "hello", value: "world")
```

18.7.11 查看查询性能

运行之后，查看查询性能。可以看到用 Set 版的 MaxAllocated 是 51712，这跟 map 版的 52736 几乎没啥区别。



但是，我们看一下 TotalDuration 这个指标 4612562。之前的 map 版在这个指标上可是 17365972。



这说明 set 操作要比 map 要快。

第19章 使用 InfluxDB 搭建报警系统

19.1 什么是监控

监控其实每隔一段时间对数据计算一下。比如，我有一个一氧化碳浓度传感器，每 1 分钟我就算一下这 1 分钟内室内一氧化碳浓度的平均值。将这个结果跟一个写死的标准值做比较，如果超过了就报警。这就是监控的基本逻辑。

所以，InfluxDB 中的监控其实也是一个 FLUX 脚本写的定时任务。只不过，不管是在 HTTP API 还是在 Web UI 上，InfluxDB 都把它和定时任务分离区别对待了。

19.2 认识检查、报警终端和报警规则

在 Web UI 的左侧工具栏中，点击 Alerts 按钮，会打开一个报警的配置页面。上方的选项栏中显示着 CHECKS（检查）、NOTIFICATION ENDPOINTS（报警终端）和 NOTIFICATION RULES（报警规则）分别对应着 InfluxDB 进行报警所需要的 3 个组件。

1. 点击左侧的Alerts按钮
可以进入报警的配置页面

2. 上方的三个按钮分别对应着检查、报警终端和报警规则的配置

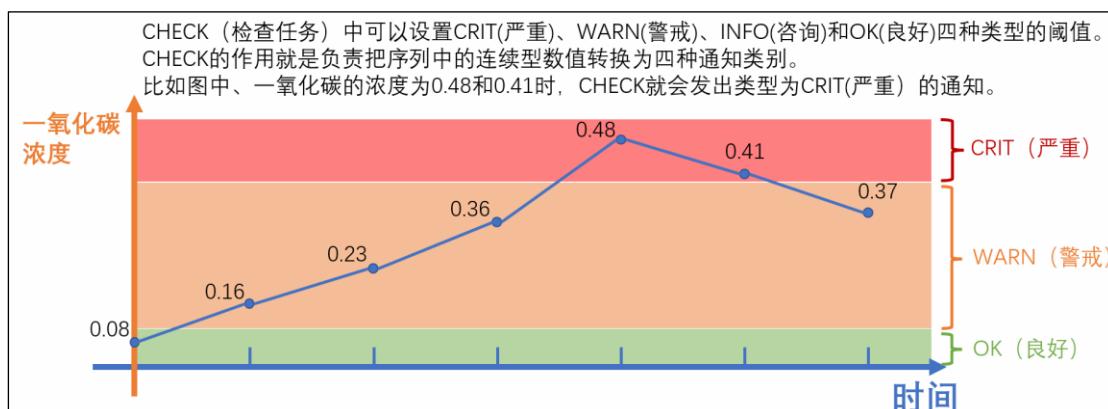
Get started monitoring by creating a check

When a value crosses a specific threshold:
+ THRESHOLD CHECK

If a service stops sending metrics:
+ DEADMAN CHECK

三个组件的功能分别如下：

- CHECKS（检查）：它其实也是一种定时任务，我们可以称之为检查任务。检查任务会从目标存储桶中读取部分数据然后进行阈值检查，并最终出 4 类信号。CRIT（严重）、WARN（警戒）、INFO（信息）和 OK（良好）。



- NOTIFICATION ENDPOINTS（报警终端）：是一个向指定地址发送报警信号的组件。
- NOTIFICATION RULES（报警规则）：它可以指定哪些 Check 出问题了发送微信报警，哪些 Check 出问题了可以发邮件通知。它相当于 Check 与报警终端之间的路由。

19.3 示例：模拟对一氧化碳浓度的报警

19.3.1 需求

假设我们现在有一个可以采集一氧化碳浓度的传感器，这个传感器通过物联网网络每隔一段时间就向我们部署在服务器上的 InfluxDB 插入一条数据，格式如下。

```
co,code=01 value=0.001 1664851126000
```

现在，我们希望使用 InfluxDB 能够完成下述的报警功能。

- 当 CO 浓度大于 0.04 的时候发出 CRIT（严重）级别的通知信号。
- 当 CO 浓度介于 0.04 和 0.01 之间的时候发出 WARN（警戒）级别的通知信号。

- 当 CO 浓度低于 0.01 的时候发出 OK (良好) 级别的通知信号。

最终，当 CO 浓度超标时，我们希望相关的工作人员能够收到一通电话，以便对事故做出快速响应。

19.3.2 辅助工具

为了方便验证报警终端的效果，老师写了一个很简单的仅支持 POST 请求的 HTTP 服务。在我们的课程资料里面有相关的源码和编译好后的可以在 linux_x86_64 平台上执行的程序。

直接使用下面的命令即可开启一个监听本地 8080 端口的 POST HTTP 服务。

```
./simpleHttpPostServer-linux-x64
```

这个命令执行后会阻塞终端，当它接收到 POST 请求后，会自动将请求体中的内容打印到终端。如果 0.0.0.0 不是你想绑定的 host 或者 8080 端口已经被占用。你可以使用下面的两个参数来修改。

```
-h 指定绑定的 host  
-p 指定绑定的 port
```

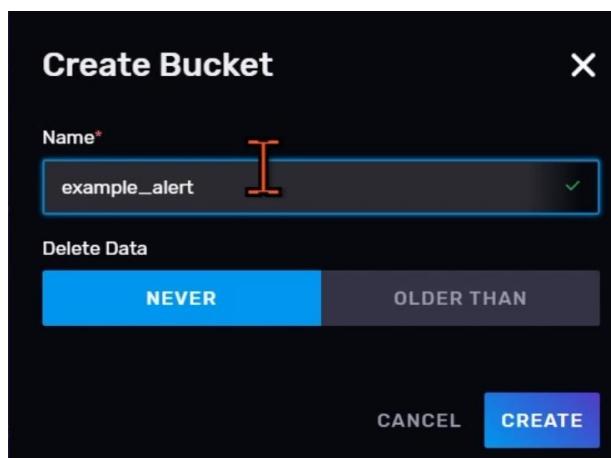
例如：

```
./simpleHttpPostServer-linux-x64 -h localhost -p 8080
```

具体可以参考项目地址：<https://github.com/realdengziji/simpleHttpPostServer>

19.3.3 创建一个新的存储桶

为了避免我们将本示例的数据同之前的示例搞混，此处我们先创建一个新的名为 example_alert 的存储桶，如下图所示：



19.3.4 准备数据模板

在本示例中，我们会自己手动一条条地向 InfluxDB 中插入数据，所以可以打开一个文本编辑器（本教程使用 vs code）先编写一个 InfluxDB 行协议的数据模板，后面可以直接

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

复制数据，稍微改一下数值然后接着插入。

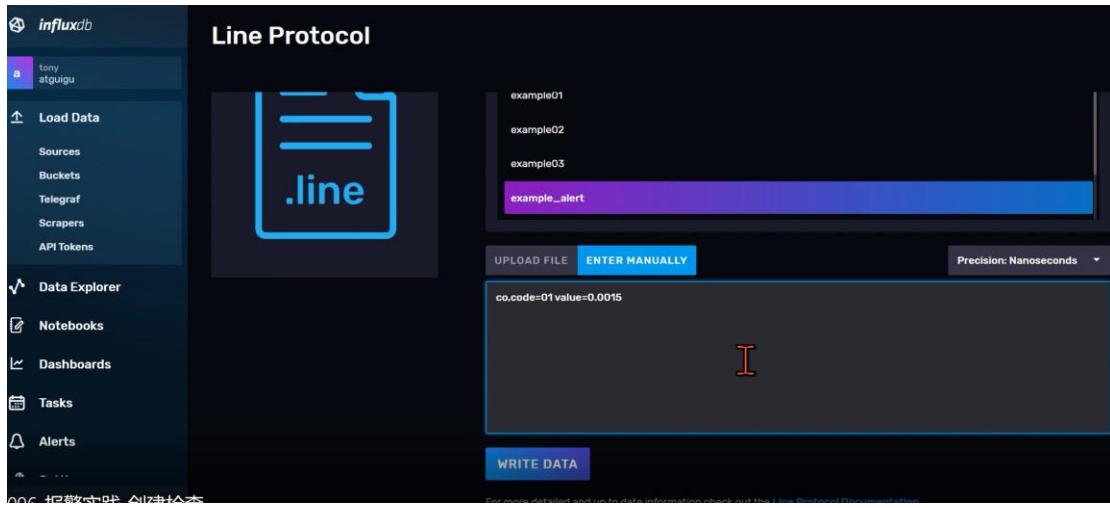
数据模板如下：

```
co,code=01 value=0.001
```

19.3.5 事先插入一两条数据

这一步操作是为了后面进行创建检查的操作时，查询构造器中有东西可选。所以为了顺利创建检查，这一步的操作不可以省略。

此处，我们在 Web UI 导入行协议数据的窗口上，分两次各导入一条数据。如图所示：



数据如下：

第一次

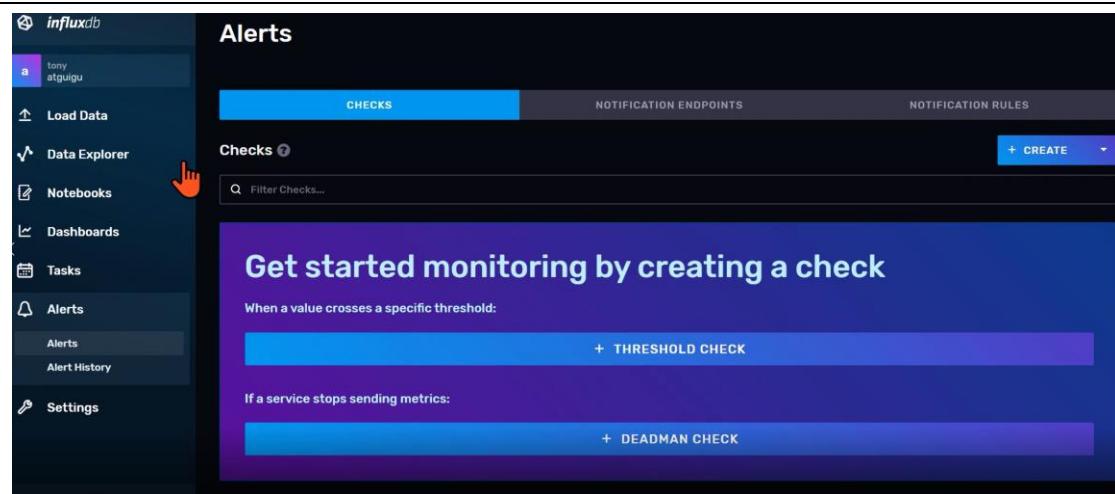
```
co,code=01 value=0.0015
```

第二次

```
co,code=01 value=0.0025
```

19.3.6 创建检查（CHECK）

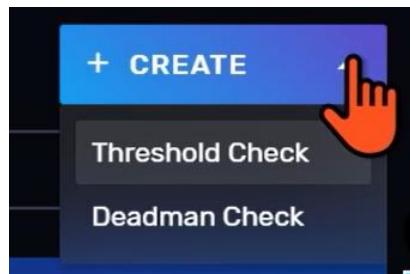
(1) 在左侧的工具栏中点击 Alerts 按钮。默认情况下会进入 CHECKS 的页面，如图所示：



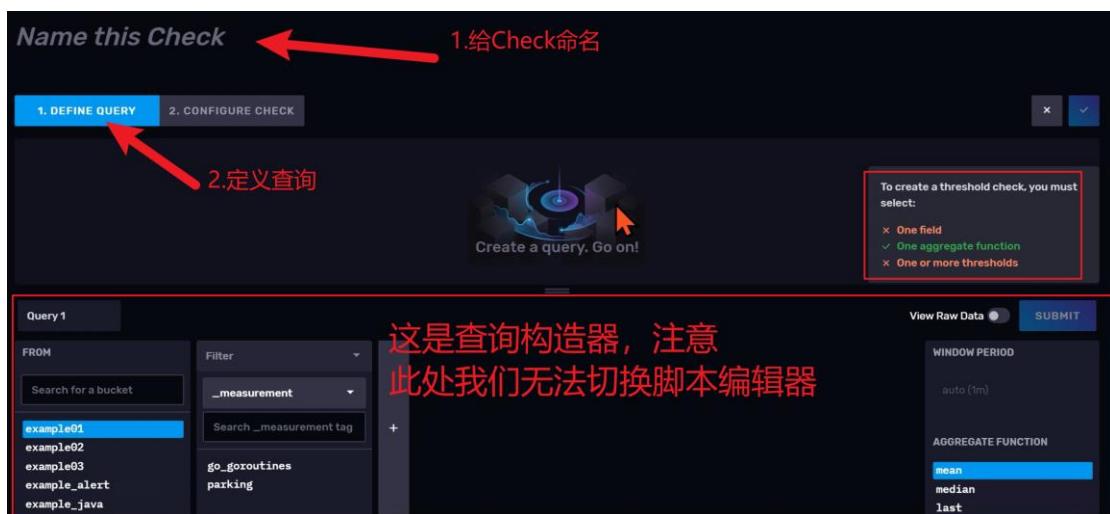
(2) 将鼠标悬停在右上角 CREATE 按钮，会弹出一个下拉菜单，其中包括两个按钮：

- THRESHOLD CHECK（阈值检查）：这类检查任务主要是去判断数据有没有超出某种阈值限定。
- Deadman Check（死人检查）：这类检查任务是去判断某个序列下多长时间没有写入新的数据了。你也可以设定一个值，比如一旦超过 30s 某个序列还没有数据入库，就发出一个警戒信号。

此处，我们选择 Threshold Check，创建一个阈值检查。



(3) 后面会弹出一个对话窗口，其布局和 Data Explorer 非常像，但是功能上会有一些差异。



- 最上方有一个 Name this Check，点击一下可以给当前创建的 Check 命名。
- 左上角有一个选项卡，默认是选中了 DEFINE QUERY（定义查询），也就是上图所示的页面效果。
- 页面的下方是一个查询构造器，需要注意，此处我们无法切换到脚本编辑器，也就是在此处，我们只能使用查询构造器来实现查询。
- 最右边还有一个清单。上面说到为了创建一个阈值检查，你必须选择：
 - 一个字段
 - 一个聚合函数（也就是开窗后的聚合函数）
 - 一个或更多的值域。

(4) 现在，我们需要构造查询。如下图所示。

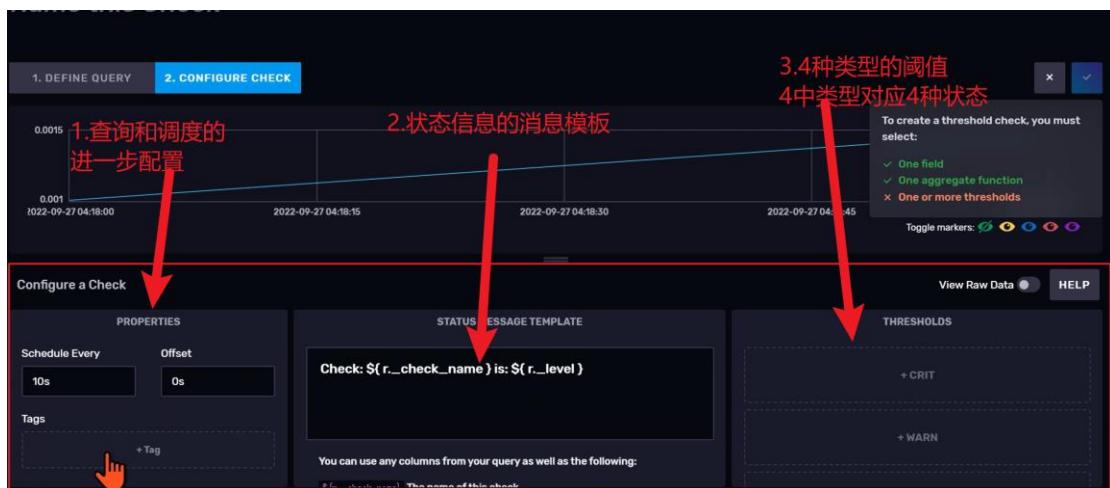


- a) 在存储桶处选择 example_alert
- b) _measurement 处选择 co

- c) 注意，虽然我们当前 co 这个 measurement 下只有一个序列，但是还是必须将 _field=value 添加到过滤条件中，否则右上方检查项 One Field 不会通过。
- d) 最后将聚合逻辑从默认的 mean 改为 max。
- e) 点击 submit，可以预览数据的查询效果。

(5) 点击左上方的 CONFIGURE CHECK 按钮。这会让我们进入一个新的页面，在这个页面中，我们可以对阈值进行配置。

首先要注意，只有页面的下半部分发生了改变。



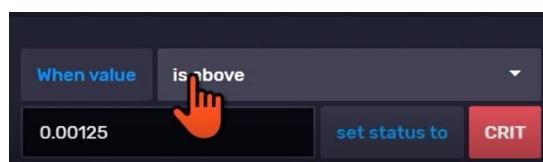
● 最左侧的卡片对应的是查询和调度的进一步配置。这里我们将 Schedule Every 设为 15s 这样，每隔 15 秒就会调用一次检查。

● 中间的 STATUS MESSAGE TEMPLATE 是状态消息模板。这里支持使用 Shell 风格的取值语法。\${ }。这里 r 的含义在后面会进行详细的讲解。此处保持默认的模板不做任何修改。

● 最右侧的 THRESHOLDS 对应的是值域的设定。此处包含 4 种类型的值域，对应着一个检查能够发出的 4 中状态信息。它们分别是：

- CRIT (critical 的前 4 个字母) 表示严重紧急。
- WARN (warning) 表示警告、警戒
- Info (Information) 表示普通信息，提醒
- ok 表示状态良好

此时，点击右下角的 CRIT 按钮，会弹出一个小的设置窗口，如下图所示

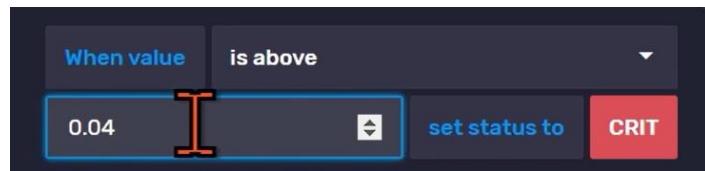


更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网

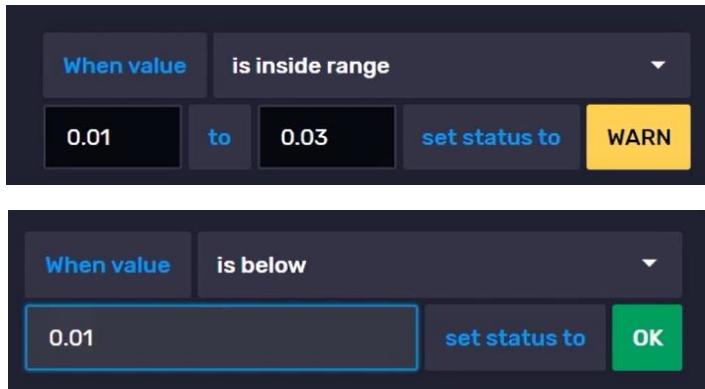
这里的意思就是，当值大于多少的时候将检查的状态设置为 CRIT。此处 When value 右边的 is above 就是大于的意思，可以看到这还是一个下拉菜单。我们可以点击一下。会发现它还有更多可选的选项，包括 is below（小于）、is inside range（在什么范围之内）等。



0.00125 是 Web UI 根据我们当前的查询结果自动帮我们填充的。这里，根据我们的需求，co 的浓度值大于 0.04 的时候才将状态置为 CRIT。效果如下。



同理，设置 Warn 和 ok，Info 就不设置了。结果如下。



最后点击右上角的对号，保存 Check。

现在，我们回到了最初的 CHECKS 页面，可以看到，下方的列表里就有一个我们刚才配置的 Check。

The screenshot shows the InfluxDB interface with the 'Alerts' section selected. A specific alert named 'CO_Alert' is highlighted. A red arrow points to the 'Task ID: 0a0adfc76252b000' field. A red annotation text states: '注意这个Task ID 说明它本质上也是一个定时任务' (Note that this Task ID indicates it is本质上 also a scheduled task).

19.3.7 测试 Check

现在，可以回到上传数据的页面，尝试插入两条数据测试一下检查的运行效果。

The screenshot shows the InfluxDB interface with the 'Line Protocol' section selected. A red arrow points to the 'ENTER MANUALLY' button at the bottom of the data entry form. The form contains the data point 'co,code=01 value=0.025'.

插入的数据如下：

```
co,code=01 value=0.025
```

这时一氧化碳的浓度为 0.025，介于 0.01 到 0.03 之间，这个时候我们刚才创建的 CHECK 应该发出 WARN 级别的信号。

现在，我们可以在左侧的工具栏里，点击 Alert History。

1.点击Alert History

可以看到，现在我们的状态记录里面就躺着一条级别为 WARN 的通知。这里，右面的 MESSAGE 显示 Check:CO_Alert is : warn 就是我们的消息模板生成的消息。

19.3.8 修改消息模板

当前，我们的消息模板提示的消息还不够精确，我们希望进行报警的时候能够把当前的一氧化碳浓度的值也输出出来。

这个时候可以看一下官方文档关于模板的说法，可以发现，官方文档指出，我们是可以通过 `r.fieldName` 的方式去访问到数据具体值的。

可以通过字段名称来提取值

```

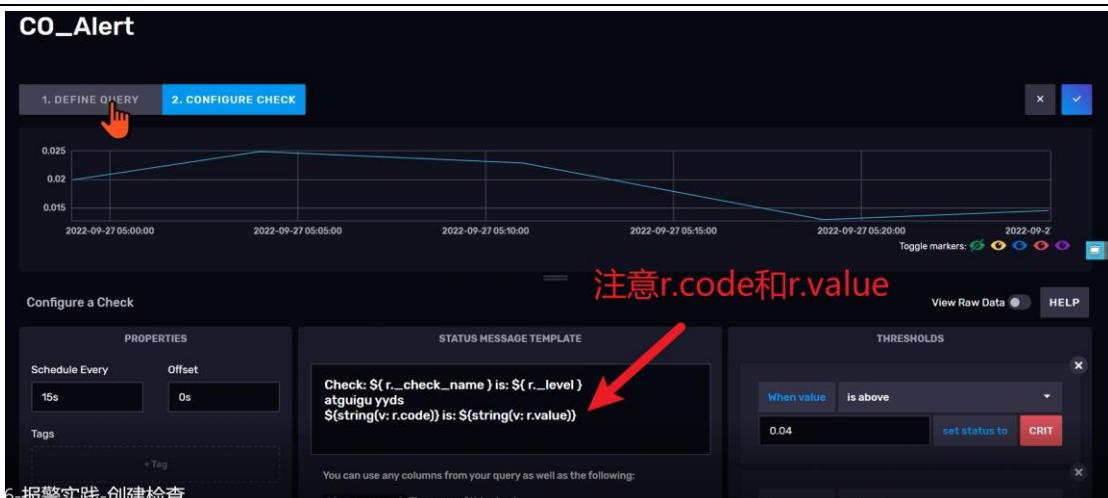
r.columnName .

Use data from the following columns:
• columns included in the query output
• custom tags added to the query output
• _check_id
• _check_name
• _level
• _source_measurement
• _type

Example status message template
From ${r._check_name}:
${r._field} is ${r._level}.
Its value is ${string(v: r.field_name)}.

```

这样的话我们就可以重新修改消息模板，最终的消息模板如下图所示：



注意，模板中的 r.code 和 r.value。通过这个操作，我们可以直接提取数据中的设备编号和当前的一氧化碳浓度值。

19.3.9 验证消息模板

接下来，我们再次插入一条数据。

```
co,code=01 value=0.0146
```

0.0146 在 0.01 和 0.03 之间，我们之前创建的 CHECK 应该再发出一个 WARN 级别的信号。

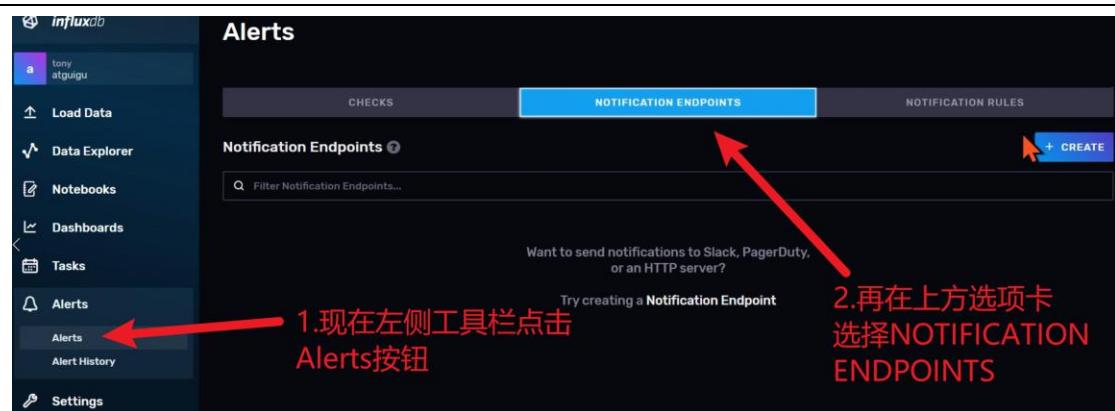
现在，还是在左侧的工具栏点击 Alter History，查看检查汇报的状态记录。我们发现新的状态记录里面的 MESSAGE 有所改变，这次在信息中我们可以看到设备编号和当时的一氧化碳浓度值。

TIME	LEVEL	CHECK	MESSAGE
2022-09-27 05:24:45	warn	CO_Alert	Check: CO_Alert is: warn atguigu yyds 01 is: 0.0146

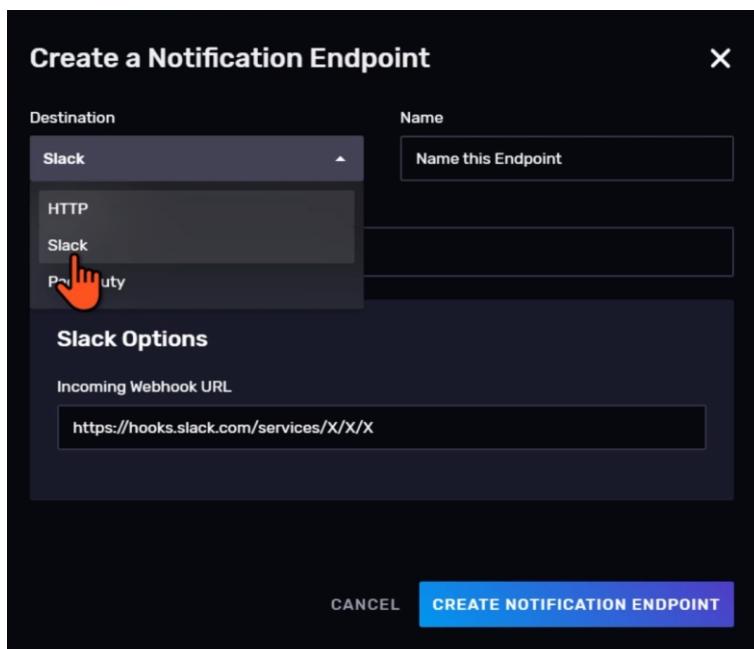
19.3.10 创建报警终端（NOTIFICATION ENDPOINT）

仅有状态记录还不够，我们还需要将信息发送到外部系统，比如给开发人员发送邮件，或者拨打电话。那么这个负责向外面发送消息的组件就是报警终端。

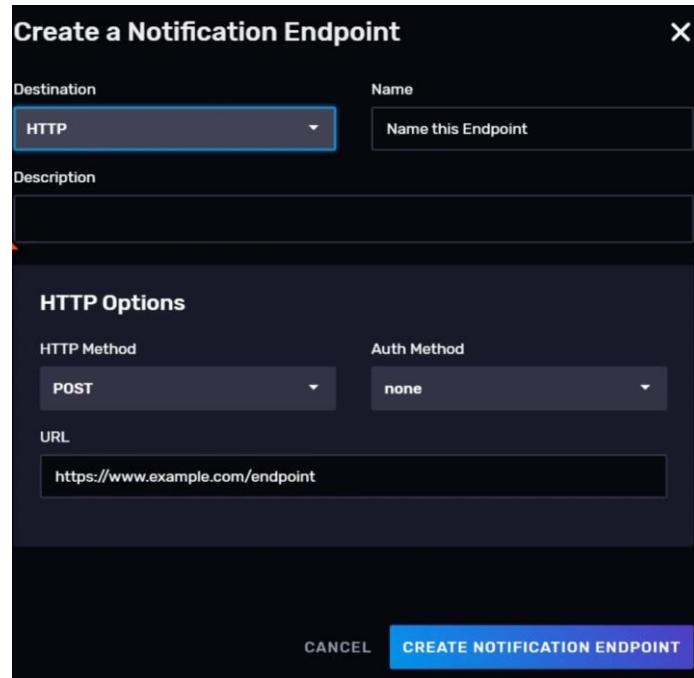
(1) 首先点击左侧工作栏中的 Alerts 按钮，进入页面后，在上方的工具栏选择 NOTIFICATION ENDPOINTS 选项卡。



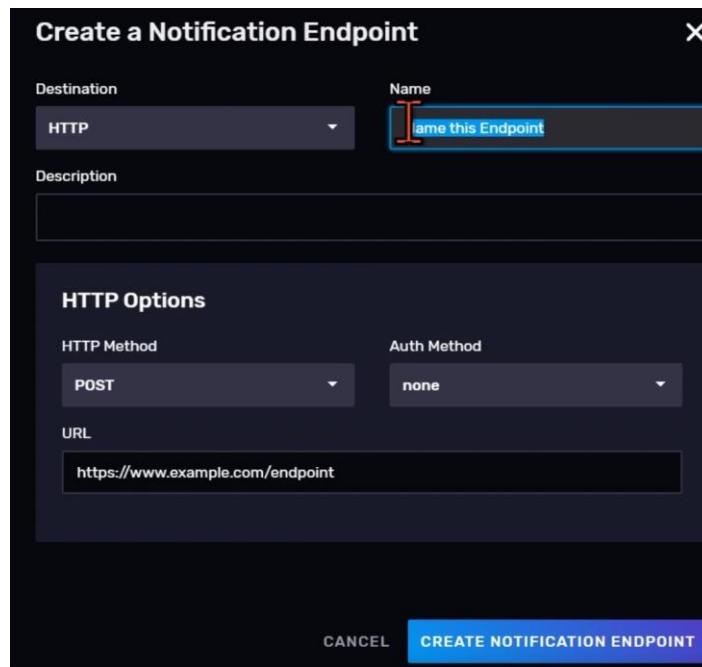
(2) 点击右上角的 CREATE 按钮，会弹出一个如下图所示的对话窗。



左上角的 Destination 有个下拉菜单，这个其实是报警终端的类型，可以看到此处为我们提供了 3 种终端，HTTP、Slack 和 Pagerduty。Slack 和 Pagerduty 是海外开发团队常用的通讯软件，此处我们选择 HTTP。



(3) 选择 HTTP 后，可以看到窗口中的配置项会发生变化。所谓 HTTP 终端，其实就是向一个目标地址发送 POST 请求。



(4) 我们暂时不向睿象云对接，而是想办法先去观察一下 HTTP 终端发出数据的数据结构。此处，在资料包中找到我们的辅助工具 SimpleHttpPosyServer。

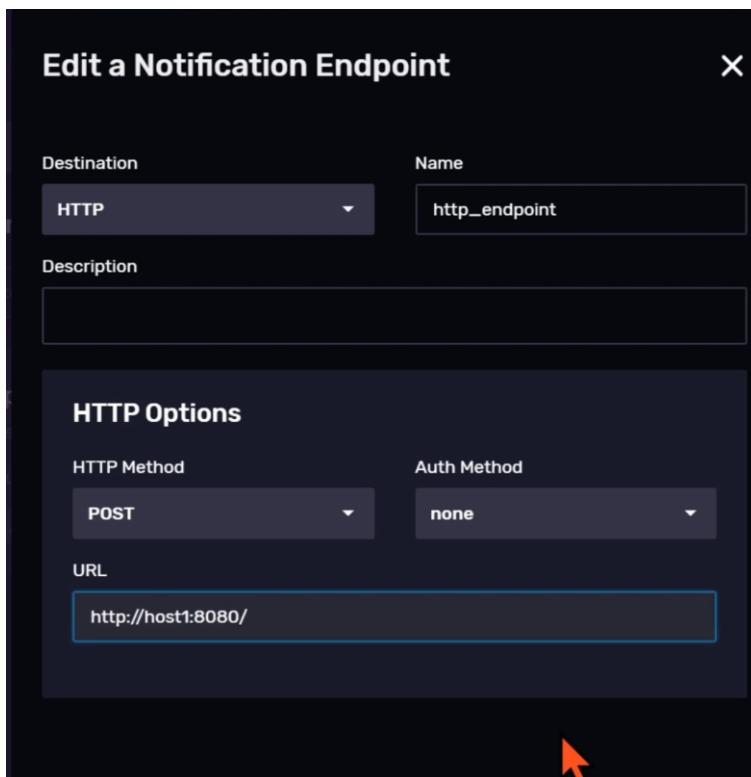
名称	修改日期	类型	大小
向日葵内网穿透工具rpm包	2022/10/4 9:28	文件夹	
influxdb2-client-2.4.0-linux-amd64.tar.gz	2022/10/4 9:27	GZ 压缩文件	5,910 KB
simpleHttpPostServer-linux-x64	2022/10/4 9:27	文件	6,192 KB
node_exporter-1.4.0-rc.0.linux-amd64.tar.gz	2022/10/4 9:27	GZ 压缩文件	9,508 KB
telegraf-1.23.4_linux_amd64.tar.gz	2022/10/4 9:27	GZ 压缩文件	39,878 KB
influxdb2-2.4.0-linux-amd64.tar.gz	2022/10/4 9:27	GZ 压缩文件	89,941 KB

将它拷贝的 Linux 虚拟机后，执行下述命令。

```
./simpleHttpPostServer-linux-x64
```

执行后，程序会监听 0.0.0.0:8080 地址。当它收到 POST 请求时，会在终端打印收到的数据。

(5) 现在，我们可以将 InfluxDB 中的 HTTP 终端地址设为 <http://host1:8080>。如下图所示：



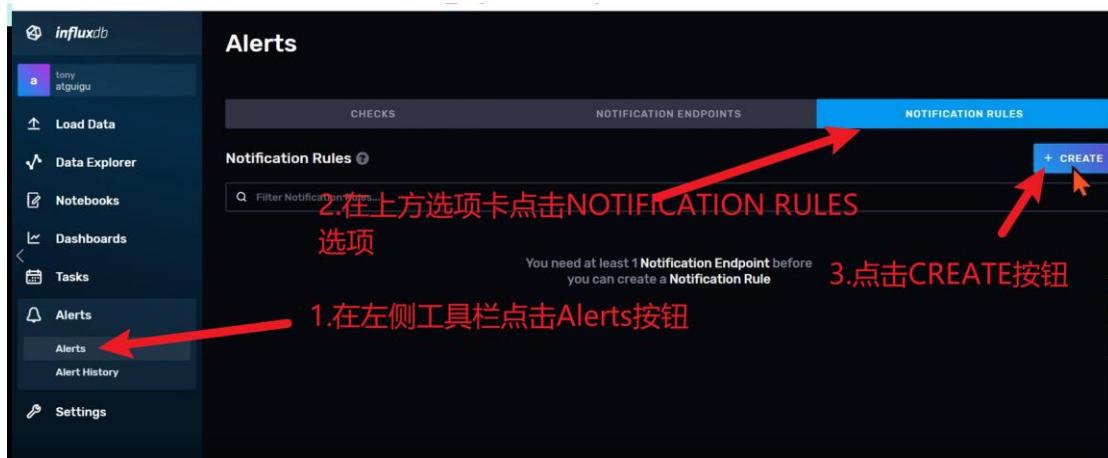
(6) 最后点击右下角的 CREATE NOTIFICATION ENDPOINT，创建终端。

19.3.11 创建报警规则（NOTIFICATION RULES）

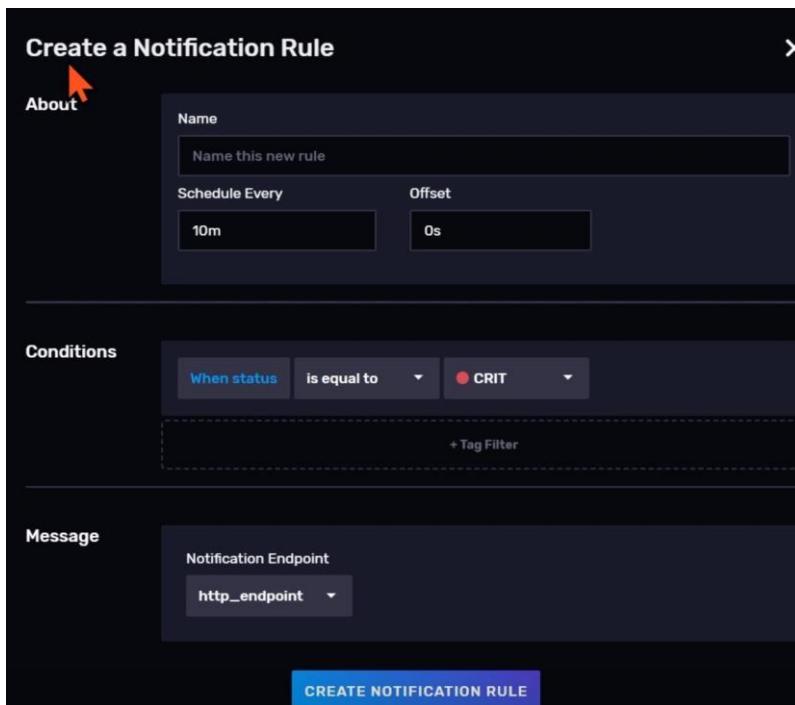
报警规则起到报警信息和终端之间的路由作用。报警规则可以指定哪些 Check 的何种级别的信息发送给哪个终端。

注意！创建报警规则的前提是已经创建了至少一个报警终端，否则 Web UI 上的创建报警规则按钮会变成灰色，也就是无法创建报警规则。

(1) 首先在左侧工具栏点击 Alerts 按钮，然后在上方的选项卡点击 NOTIFICATION。接着点击 CREATE 按钮。



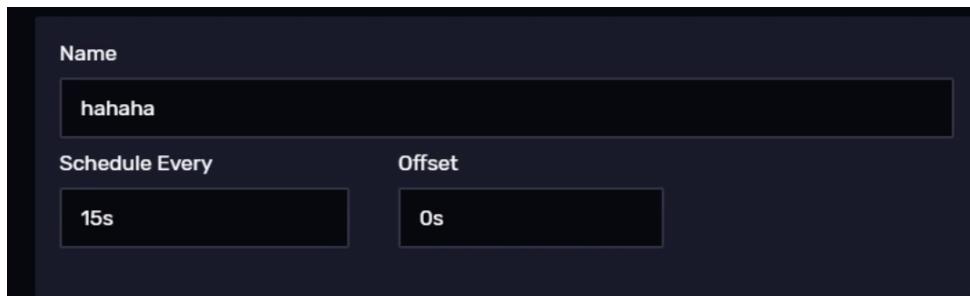
(2) 现在可以看到一个设置报警规则的弹窗。如下图所示。



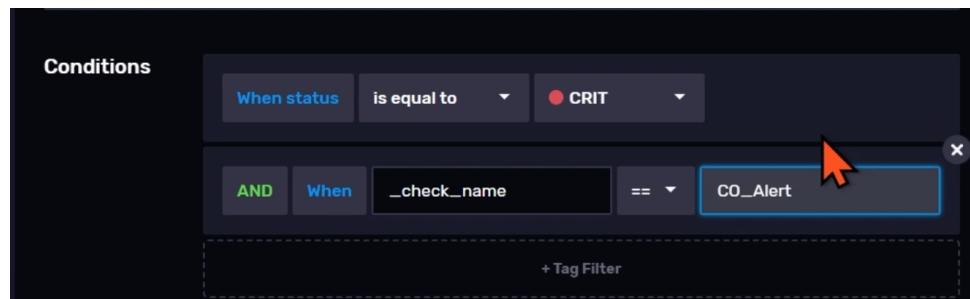
最上方可以设置调度时间，看上去就像是一个定时任务。中间 Conditions 的意思是条件。比如现在默认的条件就是当 InfluxDB 中有 CHECK 的状态为 CRIT 时，就使用 http_endpoint 发送报警信息。需要注意中间的 Conditions 还有一个按钮叫做 tag Filter，也就是按照标签过滤。

(3) 首先，为了能够更快的看到报警效果，我们将调度的时间设为 15 秒。名字可以

自己随便起。效果如下图所示。



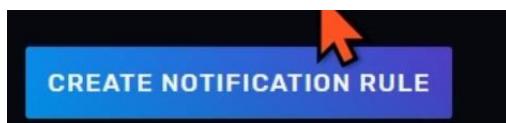
(4) 在中间的 Conditions 区域，点击一下 Tag Filter 按钮，然后添加一个标签过滤条件为 `_check_name == CO_Alert`。其中 CO_Alert 是我们之前创建的检查的名称。后面我们会讲到检查和通知规则的工作原理。此处先这样设置，设置后的效果如下图。



(5) 窗口最下方的 Message 区域，因为我们目前只有一个名为 http_endpoint 的终端，所以这里 UI 自动帮我们选择的 http_endpoint，保持现状就好。



(6) 最后点击最下方的 CREATE NOTIFICATION RULE 按钮，创建规则。



19.3.12 测试报警信号发送效果

(1) 现在，我们要测试一下在 InfluxDB 里已经搭建好的报警链路。只需插入一条模拟的一氧化碳浓度的数据，让它的值大于 0.04 即可。

插入的数据如下：

```
co,code=01 value=0.05
```

如图所示：

The screenshot shows the InfluxDB Line Protocol interface. On the left is a large blue icon of a document with '.line' written on it. To its right is a sidebar titled 'Bucket' with four items: 'example01', 'example02', 'example03', and 'example_alert' (which is highlighted with a purple background). Below the sidebar are two buttons: 'UPLOAD FILE' and 'ENTER MANUALLY'. The main area has a dropdown menu 'Precision: Nanoseconds'. At the bottom, there is a text input field containing the line protocol data: 'co.code=01 value=0.05'.

(2) 接下来，在左侧工具栏点击 Alert History，来到报警历史页面，等待大概 15 秒。

The screenshot shows the 'Check Statuses' page. At the top, there are tabs for 'STATUSES' (which is selected) and 'NOTIFICATIONS'. Below the tabs is a search bar and a refresh button. The main area is a table with columns: TIME, LEVEL, CHECK, and MESSAGE. The first row, which has a timestamp of '2022-09-27 09:15:15', a level of 'crit', and a check of 'CO_Alert', is highlighted with a red box and a red arrow pointing to it. The message for this row is 'Check: CO_Alert is: crit atguigu yyds 01 is: 0.05'. Other rows show 'warn' levels and different checks like 'CO_Alert' and 'hahaha'.

正常情况下，在检查状态历史记录里应该会出现一个级别为 crit 的状态信息。

(3) 点击上方的 NOTIFICATIONS 按钮，这时应该出现一个通知记录，这个列表里面是 InfluxDB 向外发送通知的记录。可以看到这条记录的最右边有一个绿色的对号，这说明我们的消息已经成功通过 http_endpoint 发送出去了。

The screenshot shows the 'Check Statuses' page with the 'NOTIFICATIONS' tab selected. The table has columns: TIME, LEVEL, CHECK, NOTIFICATION RULE, NOTIFICATION ENDPOINT, and SENT. The first row has a timestamp of '2022-09-27 09:15:30', a level of 'crit', a check of 'CO_Alert', a notification rule of 'hahaha', a notification endpoint of 'http_endpoint', and a 'SENT' column with a green checkmark. Red arrows point from the 'NOTIFICATIONS' tab to the table and from the 'SENT' column to the checkmark.

(4) 回到之前开启 simpleHttpPostServer 的终端，看一下里面的内容。

```
[atguigu@host1 module]$ ./simpleHttpPostServer-linux-x64
{"check_id":"0a0adfc75eddc009","check_name":"CO_Alert","level":"crit","measurement":"notifications","message":"Check: CO_Alert is: crit\natguigu yyds 01 is: 0.05","notification_endpoint_id":"0a0b1b37bae0a009","notification_endpoint_name":"http_endpoint","notification_rule_id":"0a0b1fe01cfb39009","notification_rule_name":"hahaha","source_measurement":"co","source_timestamp":1664244990000000000,"start":"2022-09-27T02:16:15Z","status_timestamp":1664244990000000000,"stop":"2022-09-27T02:16:45Z","time":"2022-09-27T02:16:45Z","type":"threshold","version":1,"code":"01","value":0.05}
```

如图所示，我们成功收到一个 POST 请求，并将它请求体中的数据打印在了控制台。

将这条 json 数据格式化后的效果如下：

```
✓ {  
    "_check_id": "0a0adfc75eddc000",  
    "_check_name": "CO_Alert",  
    "_level": "crit",  
    "_measurement": "notifications",  
    "_message": "Check: CO_Alert is: crit\nnatguigu yyds\\n01 is: 0.06",  
    "_notification_endpoint_id": "0a0b1b37bae0a000",  
    "_notification_endpoint_name": "http_endpoint",  
    "_notification_rule_id": "0a0b1fe01cfb3000",  
    "_notification_rule_name": "hahaha",  
    "_source_measurement": "co",  
    "_source_timestamp": 16642449900000000000,  
    "_start": "2022-09-27T02:16:15Z",  
    "_status_timestamp": 16642449900000000000,  
    "_stop": "2022-09-27T02:16:45Z",  
    "_time": "2022-09-27T02:16:45Z",  
    "_type": "threshold",  
    "_version": 1,  
    "code": "01",  
    "value": 0.06  
}
```

可以看到，其中包含了数据的时间，报警的消息，报警的级别，事发时的一氧化碳浓度值等等。如果能够在终端看到最终的 json，说明 InfluxDB 的报警配置已经完成并且能够正常工作。

19.3.13 检查和报警规则的工作原理

之前我们设置报警规则的时候发现，配置报警规则是需要设置调度的时间间隔的，感觉上有些奇怪，为什么一个规则还需要每隔一段时间去执行一次呢？这要先从检查的工作原理开始讲起。

InfluxDB 安装好后，会有一个由 InfluxDB 自动创建的名为 monitoring 的存储桶。我们可以用 DataExplorer 去查询一下其中的内容。

查询结果如下：

```

table _RESULT _measurement _field _value _check_id _check_name _level _source_id
0 statuses _message Check: CO_Alert is: warn atguigu yyds 01 is: 0.024 0a0adfc75eddc000 CO_Alert warn co
table _RESULT _measurement _field _value _check_id _check_name _level _source_measurement _start
1 statuses _source_timestamp 1664239290000000000 0a0adfc75eddc000 CO_Alert warn co 2022-09-15T10:00:00Z
2 statuses value 0.024 0a0adfc75eddc000 CO_Alert warn co 2022-09-15T10:00:00Z

```

Query 1 (0.01s) + View Raw Data CSV Past 1h QUERY BUILDER SUBMIT

```

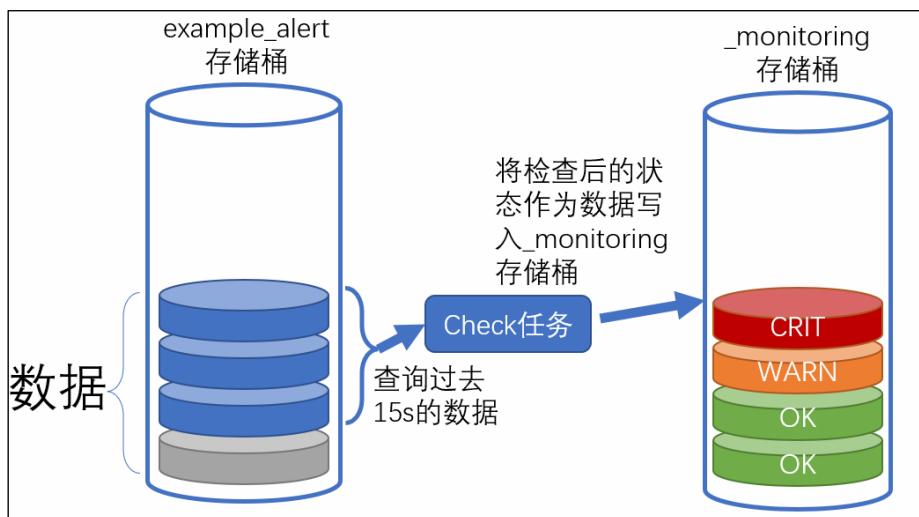
1 from(bucket: "_monitoring")
2 |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
3 |> filter(fn: (x) => x["_measurement"] == "statuses")
4

```

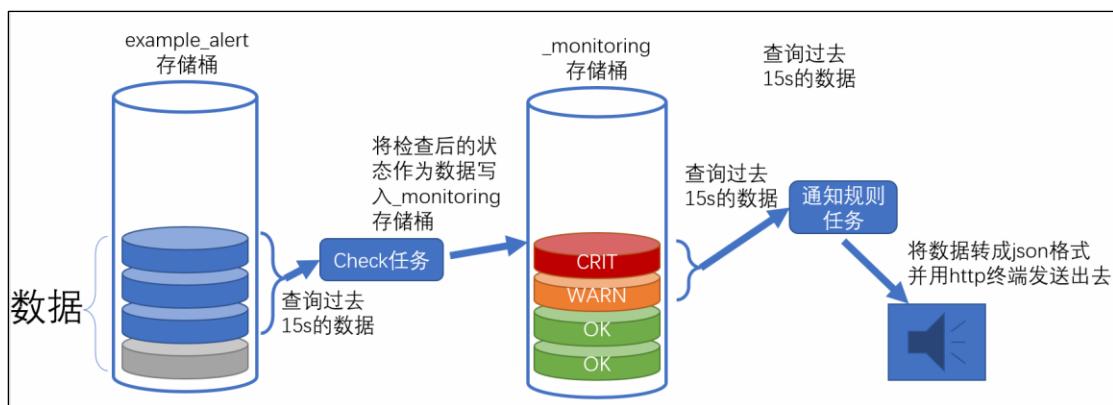
Filter Functions... Transformations

可以看到，查询的结果中，包含了 Check 任务生成的报警信息。比如有一个字段名为 _check_name，值为 CO_Alert，这就是我们之前设置的检查名称。

也就是我们定时执行的 Check，其实是定时从 example_alert 里面查询数据出来，然后对其进行阈值检查，最终，将检查后的状态信息写入到 monitoring 存储桶中。效果如下图所示：



其实后面的通知策略也是一个定时任务，它从 monitoring 查询最近一段时间的数据，并根据你设置的条件对数据进行过滤，最终如果有符合要求的数据，就使用我们的 http_endpoint 将数据转成 json 格式发送出去。最终整个流程如下图所示。



这也就是 Check、Notification rule 和 Notification endpoint 在一起工作的原理。

19.4 示例：集成睿象云（报警系统的 Saas 方案）

19.4.1 什么是睿象云

睿象云是一个告警平台，它提供五花八门的报警方式。你可以充值，选择电话报警，按照说明进行一下配置，之后你会得到一个 API 接口。以后，当你的系统需要报警时，只需在代码里想这个 API 发送一个 http 请求，睿象云就会按照你配置的电话号码，拨打电话，语音提醒程序员该加班了。

19.4.2 注册睿象云

官网地址：<https://www.aiops.com/>

注册过程（略）

19.4.3 创建自己的报警 API

19.4.4 在睿象云上创建报警 API

(1) 首先进入睿象云的首页，点击左侧的智能告警平台按钮，进入智能告警平台的工作页面。



(2) 如下图所示，点击上方选项卡的集成按钮，进入集成的配置页面



(3) 这个时候，左侧有一个监控工具列表，可以看到睿象云可以和很多监控工具进行集成，但是这个列表里面并没有 InfluxDB，这个时候还有一种万能的集成方案，REST API。REST API 会向外提供一个 URL，只要你的监控工具能够以 API 要求的数据格式向睿象云发送 POST 请求，那就就可以同睿象云集成。



(4) 这时，我们会进入一个配置页面。首先需要设置一个应用名称。再点击下方的蓝色按钮，保存并获取应用 key。

< 新建应用 / api

取消

应用名称 **TICK_TEST**

自定义一个应用名称

自动关闭时间 30 分钟 针对未关闭告警生效 (针对历史数据生效)

针对未认领的告警，以监控工具最后一次发出告警事件为准，如果设定的时间内未进行手动关闭告警，系统将自动为您关闭告警；自定义区间范围：0-1440分钟，0表示永不关闭，1440分钟即为24小时。jira类型的集成，此处的时间只能为0。

开启自动去重

规则说明：同一个应用且EVENT_ID相同且级别相同，同一个应用且告警名称相同级别相同的告警数据会被实时压缩成一条数据。最新的数据会覆盖历史数据，记录最新发生时间和服务频次。

认领后未处理提醒时间设置 3 小时

当告警被认领后，在一定的时间窗口内未被关闭，默认设置为3小时，系统会推送提醒处理通知的短信给团队的admin和认领人本人（或是可以指定通知人，以用户选择为准）。0表示不开启提醒功能。

主告警自动生成jira任务

默认关闭，如需开启，请先前往配置-Jira配置信息管理。开启后每产生一个主告警就会对应生成一个Jira问题。

此应用长时间未产生数据监控配置

默认关闭，开启后需要配置监控时长，间隔时间内没有告警接入，将邮件提醒选择的用户【标准版及专业版授权的用户】。时间范围：1~24小时，建议设置5小时以上更为合适。

未命中分派策略通知方式配置

默认关闭，开启后可选择通知对象【标准版及专业版授权的用户】及对应通知的方式。当该应用产生的告警没有命中任何分派策略时（分派策略已经配置为前提，如果用户没有配置任何分派策略将不会分派给全部成员），按照此配置通知相关人员。
微信绑定微信公众账号 (前往绑定) APP离线下载才可成功通知(右侧已勾选)

点击保存并获取应用key

保存并获取应用key

(5) 这个时候页面上会出现一行红字，这个就是 Appkey。注意不要把这个 key 泄漏出去哦。

应用名称 TICK_TEST

AppKey 8f [REDACTED] 3d

自动关闭时间 ① 30 分钟 针对未关闭告警生效 (针对历史数据生效)

开启自动去重 ①

认领后未处理提醒时间设置 ① 3 小时

主告警自动生成Jira任务 ①

此应用长时间未产生数据监控配置 ①

未命中分派策略通知方式配置 ①

AppKey这个很重要，睿象云依据该Key来区别
你是平台的哪个用户。关联信息 [查看](#)

key不要泄漏

至此，我们的报警 API 就算是配置完了。

19.4.5 创建分派策略

现在外部可以通过接口向睿象云发送报警信息了，但是睿象云平台如何将报警信息发
送给具体的个人呢。

将报警信息转发给具体的人，这个过程叫做分派。

(1) 回到报警平台的主页，点击上方的配置按钮，然后再点击下方的右边的新建分派按钮。



(2) 此时会进入一个新的配置页面，可以按照下图的说明进行操作。注意，如果此处设置分派人的时候啥都不显示，说明你的账户目前还没有绑定邮箱，这个时候请自行绑定邮箱，再进行后面的操作。



配置好后，点击保存。

现在，我们的 TICK_TEST API 一旦接收到报警信息，就会通知到具体的人了。

19.4.6 InfluxDB 尝试与睿象云进行对接

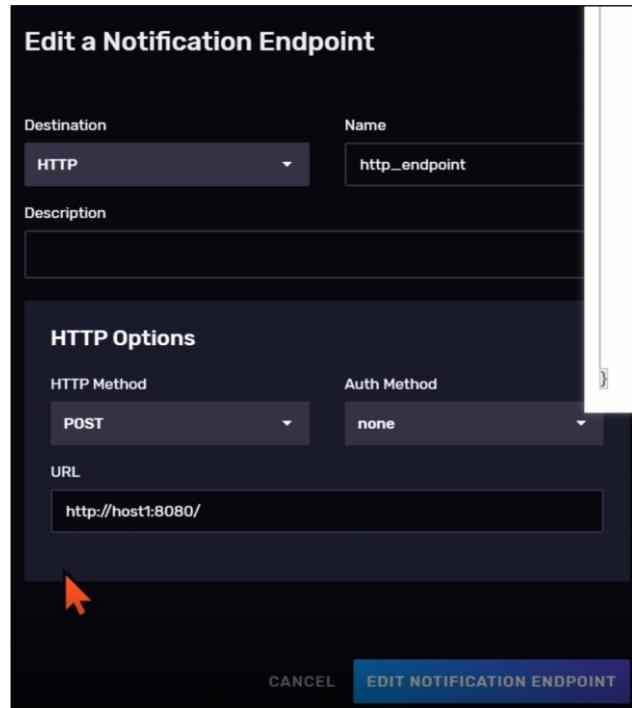
现在，我们可以尝试与睿象云进行对接了，现在看来，只要让 InfluxDB 发出的通知数据符合睿象云 API 的要求就好了。我们可以回看一下刚才在睿象云创建的 API 的说明文档（在集成->右侧应用列表->找到你自己创建的 REST API 应用->点击编辑->页面下方可见）

如下图所示，这里说明了我们应该发送什么格式的数据。

接口		
接口	http://api.aiops.com/alert/api/event	操作
参数	备注	操作
调用方式	POST	
参数格式(body)	{ "app": " 8feac05a2e4a46c495769f198a198a3d ", "eventId": "12345", "eventType": "trigger", "alarmName": "FAILURE for production/HTTP o n machine 192.168.0.253", "entityName": "host-192.168.0.253", "entityId": "host-192.168.0.253", "priority": 1, "alarmContent": { "ping time": "2 500ms", "load avg": 0.75 }, "details": { "details": "haha" }, "contexts": [{ "type": "link", "text": "generatorURL", "href": "http://www.baidu.com" }, { "type": "link", "href": "http://www.sina.com", "text": "CPU Alerting" }, { "type": "image", "src": "http://www.baidu.com/a.png" }] }	
参数格式(URL参数方 式)	?app=8feac05a2e4a46c495769f198a198a3d&eventId=xxx&eventType=trigger&alarmName=xxx&priority=2	
参数列表		
参数	备注	操作
app	必须	需要告警集成的应用KEY
eventType	必须	触发告警trigger, 解决告警resolve
eventId	可选	外部事件id, 告警关闭时用到
alarmName	可选	告警标题, alarmName与eventId不能同时为空
alarmContent	必须	告警内容详情
entityName	可选	告警对象名
entityId	可选	告警对象id
priority	可选	提醒 1, 警告 2, 严重 3, 通知 4, 致命 5
host	可选	主机
service	可选	服务

19.4.7 报警终端的不足

现在我们回到 Web UI 的 Alerts 页面，并点击到 http_endpoint 的编辑页面。会发现，没有地方可以修改发送数据的格式。不错，InfluxDB 的报警终端就是没法去设置发送的数据格式的。所以到这一步，我们前功尽弃了。但是老师还有一个方案，下一节我们会直接接触检查与报警的底层。

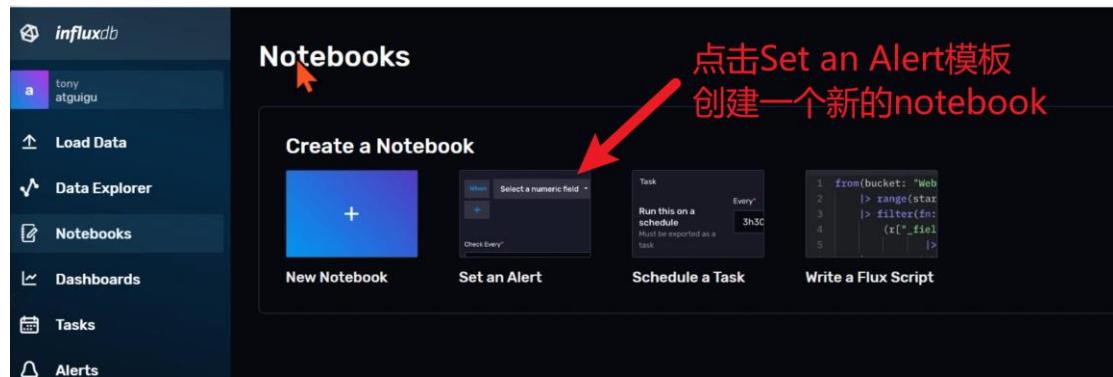


19.5 示例：Notebook 与报警的底层

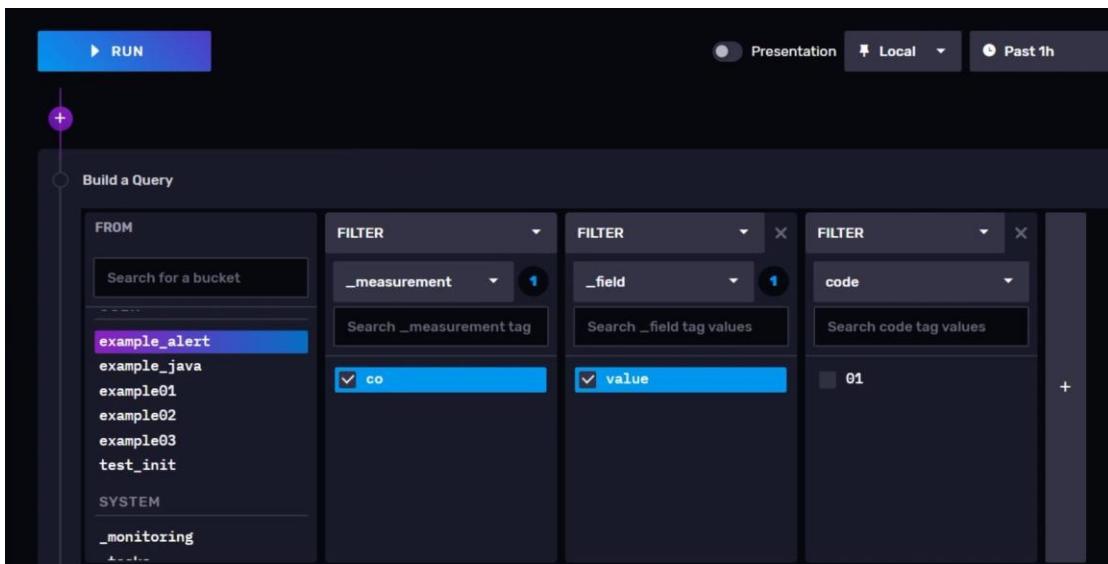
在之前，我们说过使用 Notebook 也可以创建报警任务，但是之后就再也没有接触过 Notebook。这一节，我们直接借助 Notebook，走进报警的底层。

19.5.1 使用 Notebook 创建报警任务

(1) 首先，在左侧点击 Notebooks 按钮，来到 Notebooks 的配置页面。然后，点击 Set an Alert 模板，创建一个新的 notebook。



(2) 进入 Notebook 后，会发现第一个 Cell 是一个查询构造器。此处，我们将存储桶设为 example_alert，_measurement 设为 co，_field 设为 value。效果如下图所示：



Build a Query

FROM

Search for a bucket

example_alert

example_java

example01

example02

example03

test_init

SYSTEM

_monitoring

FILTER

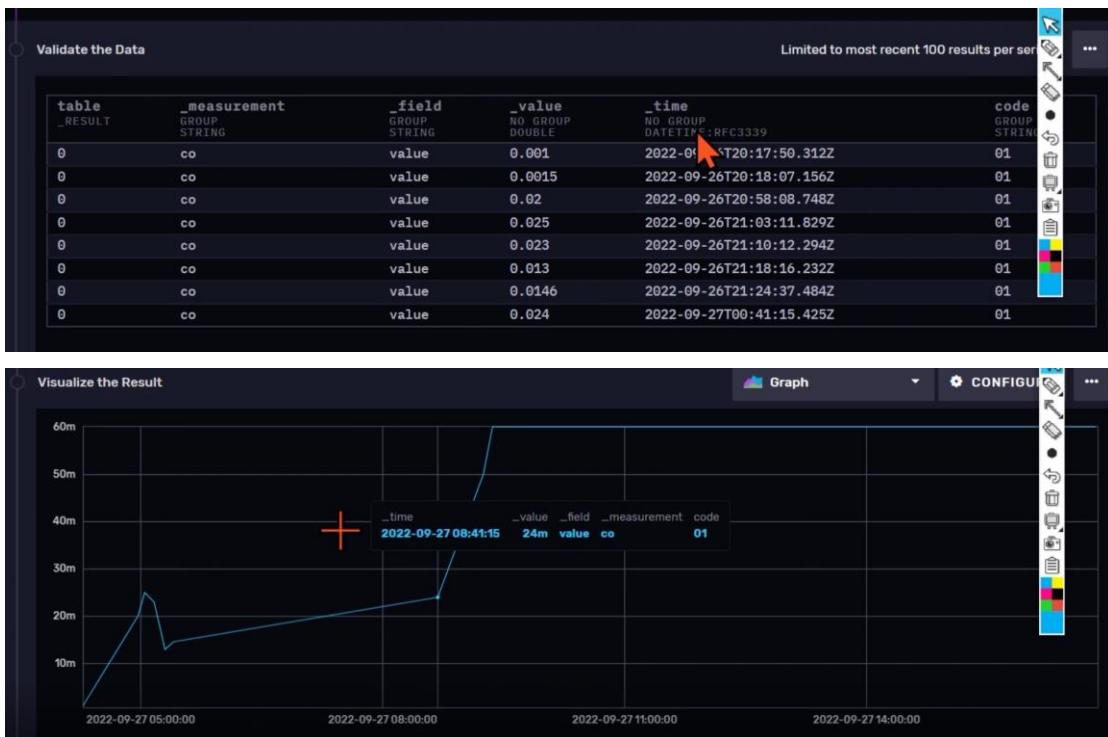
_measurement: co

_field: value

code: 01

RUN

(3) 点击上方的 RUN 按钮，查看执行效果。



Validate the Data

Limited to most recent 100 results per series

table	_measurement	_field	_value	_time	code
RESULT	GROUP STRING	GROUP STRING	NO GROUP DOUBLE	NO GROUP DATETIME:RFC3339	GROUP STRING
0	co	value	0.001	2022-09-27T20:17:50.312Z	01
0	co	value	0.0015	2022-09-26T20:18:07.156Z	01
0	co	value	0.02	2022-09-26T20:58:08.748Z	01
0	co	value	0.025	2022-09-26T21:03:11.829Z	01
0	co	value	0.023	2022-09-26T21:10:12.294Z	01
0	co	value	0.013	2022-09-26T21:18:16.232Z	01
0	co	value	0.0146	2022-09-26T21:24:37.484Z	01
0	co	value	0.024	2022-09-27T00:41:15.425Z	01

Visualize the Result

Graph

CONFIGURE

60m

50m

40m

30m

20m

10m

0m

2022-09-27 05:00:00

2022-09-27 08:00:00

2022-09-27 11:00:00

2022-09-27 14:00:00

time: 2022-09-27 08:41:15

_value: 24m

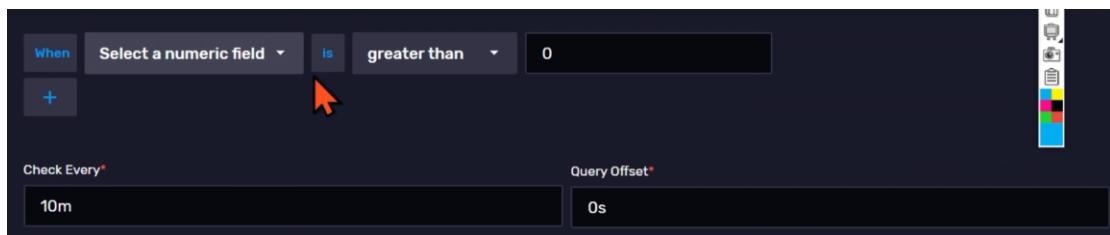
_field: value

_measurement: co

code: 01

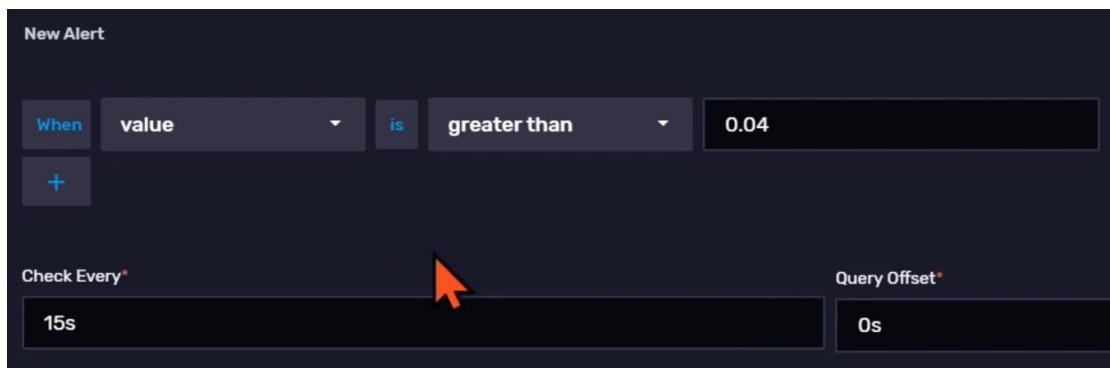
可以看到下方的两个 cell，一个 cell 是将查询出来的数据原样展示了出来，另一个是将数据绘制成了折线图。

(4) 最后，下面还有一个 cell，左上角有这个 cell 的名称，New Alert。也就是说明这个 cell 是用来配置报警的。

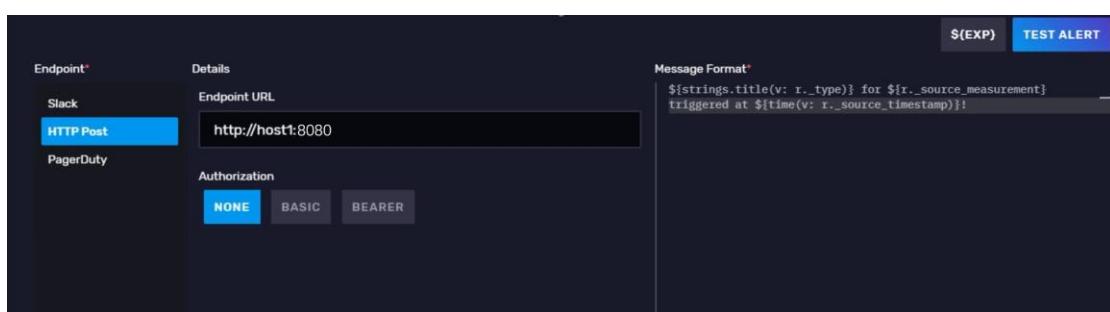


上方有两块，一个是用来设置报警条件的，另一个是用来设置调度间隔的。此处，我们还是按照需求，将报警阈值设为 0.04。细心的同学可能发现，我们此处的报警阈值只能设置一种，少了 crit、warn、info 和 ok，这个问题我们后面会提到。

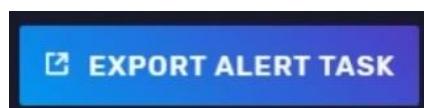
设置好后的效果如下图所示。



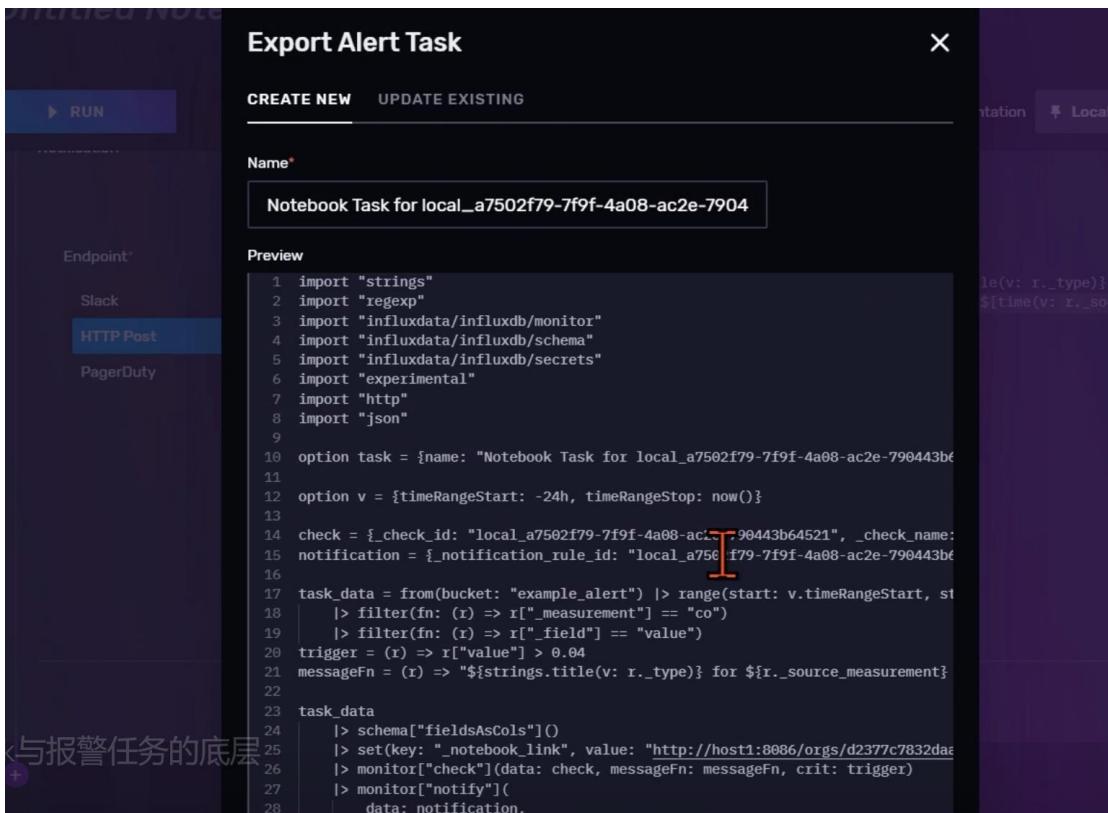
(5) 再看到这个 cell 的底部，这是报警终端的配置。此处，我们还是选择 http 终端。并将目标 URL 设置为 <http://host1:8080>。



(6) 上面的操作都完成后，点击右下方的 EXPORT ALERT TASK 按钮。



我们会惊喜地发现 Notebook 直接为我们生成了一个很长的 FLUX 脚本。如下图所示。现在，建议大家先将脚本复制出来，粘贴到 Data Explorer 里。稍后，我们自己研读这段脚本。



19.5.2 脚本解读

脚本如下：

```

import "strings"
import "regexp"
import "influxdata/influxdb/monitor"
import "influxdata/influxdb/schema"
import "influxdata/influxdb/secrets"
import "experimental"
import "http"
import "json"

option task = {name: "Notebook Task for local_8dc08939-f5df-447e-8e53-41532537902f", every: 10m, offset: 0s}

option v = {timeRangeStart: -24h, timeRangeStop: now()}

check = {_check_id: "local_8dc08939-f5df-447e-8e53-41532537902f", _check_name: "Notebook Generated Check", _type: "custom", tags: {}}
notification = {_notification_rule_id: "local_8dc08939-f5df-447e-8e53-41532537902f", _notification_rule_name: "Notebook Generated Rule", _notification_endpoint_id: "local_8dc08939-f5df-447e-8e53-41532537902f", _notification_endpoint_name: "Notebook Generated Endpoint"}

task_data = from(bucket: "example_alert") |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "co")
|> filter(fn: (r) => r["_field"] == "value")

```

```

trigger = (r) => r["undefined"] > 0
messageFn = (r) => "${strings.title(v: r._type)} for
${r._source_measurement} triggered at ${time(v:
r._source_timestamp)}!"
```

```

task_data
    |> schema["fieldsAsCols"]()
    |> set(key: "_notebook_link", value:
"http://host1:8086/orgs/d2377c7832daa87c/notebooks/0a0bc4b03a6ba0
00")
    |> monitor["check"] (data: check, messageFn: messageFn, crit:
trigger)
    |> monitor["notify"] (
        data: notification,
        endpoint: http["endpoint"] (url: "http://host1:8080") (
            mapFn: (r) => {
                body = {r with _version: 1}

                return {headers: {"Content-Type":
"application/json"}, data: json["encode"] (v: body)}
            },
        ),
    )
)

```

接下来，我们就按照从前到后的顺序为大家讲解这段代码。

19.5.2.1 导包

最上方的 import 代码我们直接跳过，不讲解了。

19.5.2.2 option task

```
option task = {name: "Notebook Task for local_8dc08939-f5df-447e-8e53-41532537902f", every: 15s, of}
```

option task 其实就是对定时任务的设置，这一行代码其实也表名了，notebook 帮我们生成的报警脚本本质上是一个 InfluxDB 的定时任务。

19.5.2.3 option v

第一行代码 option v，声明了一个 record 类型的变量，里面有两个键值对，其实分别表名了查询时间范围的开端和末端。这里显示-24h，其实是因为我们直接在 notebook 里面操作的时候，右上角的时间范围设置了-24h。后面我们会把它改成-15s。

```

option v = {timeRangeStart: -24h, timeRangeStop: now()}
check = {_check_id: "local_a7502f79-713a-4a08-ac2e-790443b64521", _check_name: "Notebook Generated Check",
notification = {_notification_rule_id: "local_a7502f79-7f9f-4a08-ac2e-790443b64521", _notification_rule_name:
"Generated Check"}
```

```

task_data = from(bucket: "example_alert") |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "co")
|> filter(fn: (r) => r["_field"] == "value")
```

19.5.2.4 check 和 notification 两个变量

这两行代码分别声明了一个 record，它其实是用来给后面的 monitor 函数做参数用的，

因为 monitoring 存储桶中需要 _check_id 和 _check_name 和 _type 等字段，所以 notebook 自动生成的时候自动帮我们安排好了。

```
check = {_check_id: "local_a7502f79-7f9f-4a08-ac2e-790443b64521", _check_name: "Notebook Generated Check", _type: "monitoring", _version: 1}
notification = {_notification_rule_id: "local_a7502f79-7f9f-4a08-ac2e-790443b64521", _notification_rule_name: "Notebook Generated Check", _version: 1}
```

19.5.2.5 查询数据

```
task_data = from(bucket: "example_alert") |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "co")
|> filter(fn: (r) => r["_field"] == "value")
```

上图中的代码完成了对 example_alert 存储桶的查询。并且查询的表流被赋值给了一个名为 task_data 的变量。

19.5.2.6 声明阈值函数

```
trigger = (r) => r["value"] > 0.04
```

此处，有一个名为 trigger 的函数，可见它的主体逻辑就是一个谓词表达式。在这里声明一个函数其实是因为后面有个 monitor 函数需要传入一个谓词函数。另外，可以看到这个函数的逻辑就是用来判断一氧化碳浓度是否超过 0.04 的。

19.5.2.7 消息模板

```
messageFn = (r) => "${strings.title(v: r._type)} for ${r. source_measurement} triggered at ${time(v: r. source_time)}
```

这也是一个函数，不过它直接返回一个字符串，这里面的内容其实是消息模板。

19.5.2.8 报警逻辑

接下来这一大段都是报警的逻辑。

```
task_data
|> schema["fieldsAsCols"]()
|> set(key: "_notebook_link", value: "http://host1:8086/orgs/d2377c7832daa87c/notebooks/0a0b8ac396eba000")
|> monitor["check"](data: check, messageFn: messageFn, crit: trigger, )
|> monitor["notify"](
  data: notification,
  endpoint: http["endpoint"](~: "http://host1:8086")(
    mapFn: (r) => {
      body = {r with _version: 1}
      return {headers: {"Content-Type": "application/json"}, data: json["encode"](v: body)}
    },
  ),
)
```

(1) 首先 schema["fieldAsCols"] 函数起到了转换数据结构的作用。效果如下图所示：



(2) set 函数为表流添加了一个常量字段

(3) monitor["check"]函数起到了检查状态的作用

```
|> monitor["check"](data: check, messageFn: messageFn, crit: trigger,)
```

需要注意，data 参数传进来的 check 变量其实就是 _check_id, _check_name 这些变量。messageFn 是消息模板，crit 在我们之前的 CHECK 里是一个报警级别，但是这里变成了函数的形参，传进来的函数值是 trigger 是我们之前提到过得谓词函数。

```
trigger = (r) => r["value"] > 0.04
```

另外注意，虽然 notebook 帮我们生成的脚本里面值传递了 crit 参数，但是 monitor["check"]函数其实还有其他可传的参数。如下图所示：

```
|> monitor["check"](data: check, messageFn: messageFn, crit: trigger,)
```

```
|> monitor["notify"](
    data: notification,
    endpoint: http["endpoint"](url: "http://host1:8086")(
        mapFn: (r) => {
            body = {r with _version: 1}
    )
}
```

可以看到，还有 info、ok、warn 参数。所以其实我们还是可以手动修改脚本把这些值域范围给补上的。

(4) monitor["notify"]函数是用来向外发送数据的，可以看到里面声明了一个 http 终端。最后，十分需要注意的是有一个名为 body 的局部变量。

```
body = {r with _version: 1}
```

这个其实是我们发送 POST 请求的请求体。r 是我们表流里面的数据，所以说，对接不上睿象云的症结就在这里。

我们只需要将 body 修改成符合睿象云 api 要求的格式就可以了。

(5) 为什么我们不直接用 if else 的逻辑来完成大于小于检查然后直接向外发送请求，而是用两个专门的 monitor 函数来完成功能呢？主要是因为 monitor 函数会在我们的 Alter history 里留下痕迹。也就是 monitor["check"] 和 monitor["notification"] 函数会向 _monitoring 存储桶里写检查和通知记录，这是非常重要的。

19.5.3 修改脚本以集成睿象云

最后，我们把 body 这个局部变量做一下修改，让它符合睿象云 API 要求的格式就好了。

```

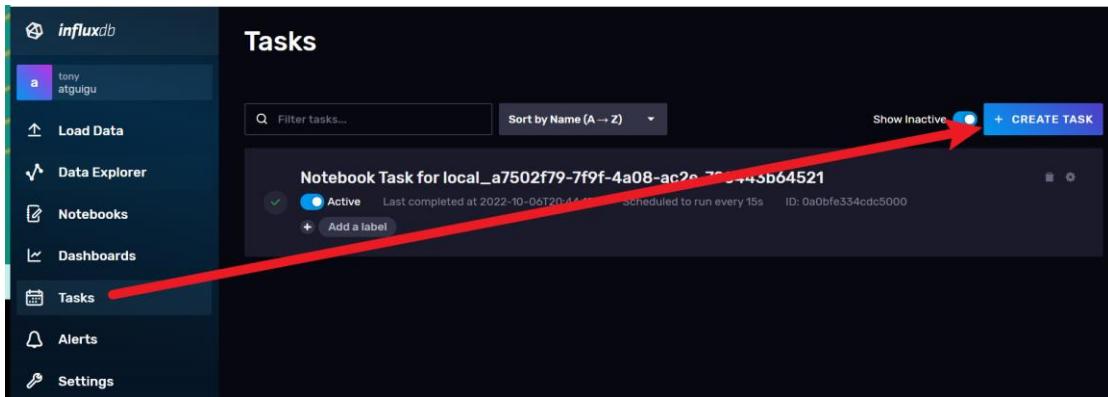
endpoint: http["endpoint"](url: "http://host1:8080")(
  mapFn: (r) => {
    body = {
      "app": "5ea82fc1c8e949d484e9c9e332d30cb0",
      "eventId": string(v: now()),
      "eventType": "trigger",
      "alarmName": "一氧化碳浓度大于0.04",
      "alarmContent": r["code"] + "设备的检测值是" + str["value"]
    }
    return {headers: {"Content-Type": "application/json"}, data: json["encode"](v: body)}
  }
)

```

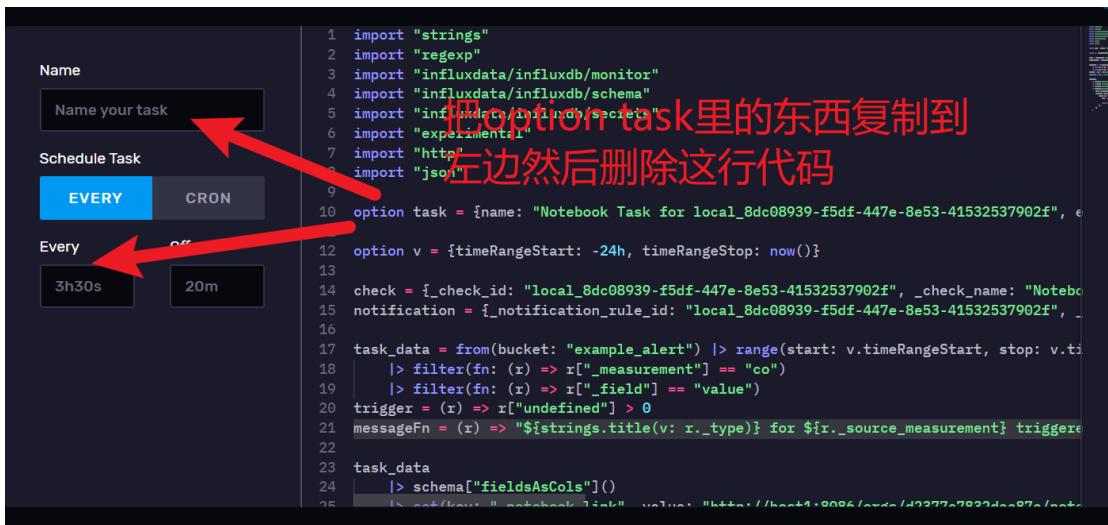
修改好的代码如上图所示。

19.5.4 创建定时任务

(1) 点击左侧 Tasks 按钮，在点击右上方的 CREATE TASK 按钮，



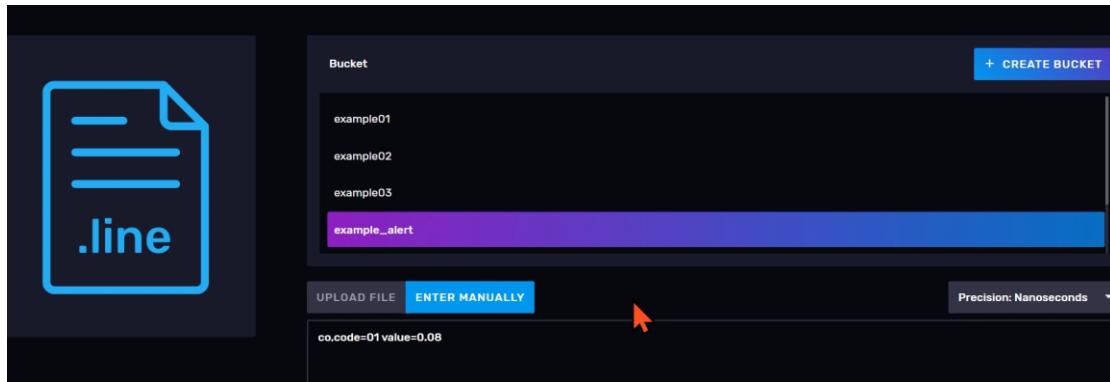
(2) 将方才我们修改好的脚本粘贴到编辑区域，将 option task 一行代码中的信息写到左侧的设置表单中，并删除原先的 option task 代码。



(3) 最后点击右上角的 Save 按钮，创建这个定时任务

19.5.5 测试报警效果

现在，我们上传一条 value 大于 0.04 的数据，测试一下对接的效果。



数据如下

```
co,code=01 value=0.08
```

等待一段时间，可以看到，我们收到了一通电话，这个电话里面就向我们说明了一氧化碳浓度的值超过 0.04 了。



19.6 示例：改进报警系统

19.6.1 当前的报警架构

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

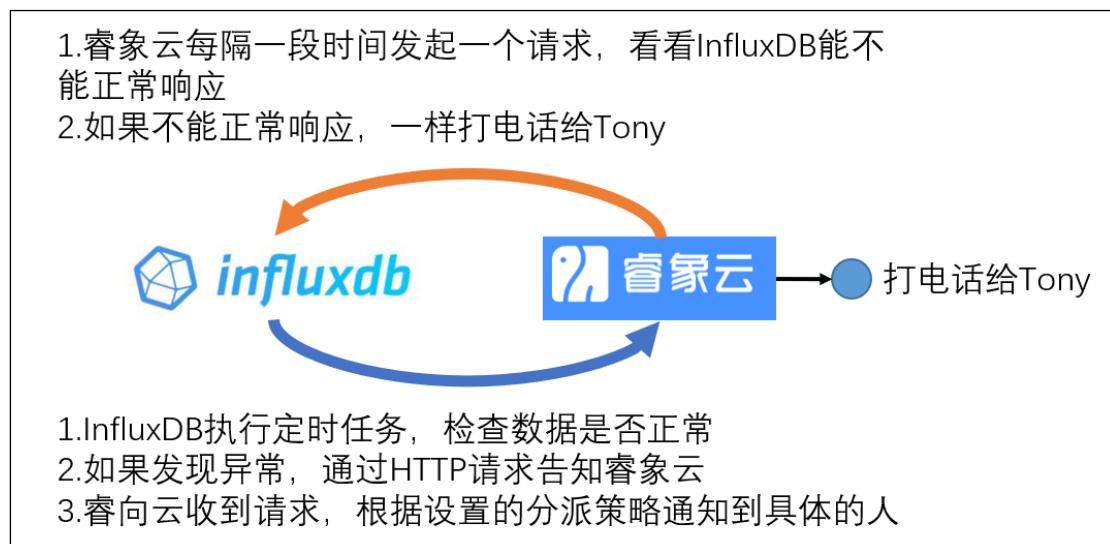
你可以将睿象云视作一个高可用的报警服务，也就是不论如何睿象云 24 小时都能无故障访问。那么结合睿象云，我们的 InfluxDB 可以设置一个检查数据合理性的定时任务，每隔一段时间就把最近的数据抽出来算一下。如果数据不合适，就发送一个报警信号给睿象云，然后由睿象云向我们具体的技术人员发起通知。



19.6.2 更值得信任的架构

上一节的架构有一个问题，如果一晚上过去了，我的 InfluxDB 出现异常宕机了。InfluxDB 宕机了自然不会向睿象云发送报警信息，所以一夜过去了，你睡了一个好觉，但那真是一个平安夜吗？

所以，如果睿象云能够知道 InfluxDB 还有没有活着就好了。最好是睿象云能够每隔一段时间检查一下我的 InfluxDB 还在不在、能不能用。这种行为，我们称为业务可用性检查。



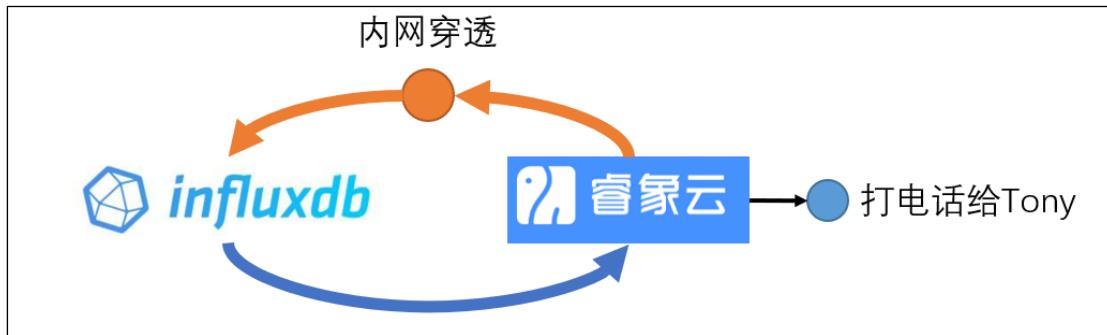
在这张图中，橘黄色的箭头就是睿象云对 InfluxDB 的检查。

19.6.3 接下来示例中的架构

使用睿象云进行报警，其实是向睿象云购买软件服务，这套软件在睿象云的服务器上，而不是自己企业的服务器上。这种方式我们称为 SaaS，软件即服务。这种情况下，想让睿象云能够反过来访问我们的 InfluxDB 服务，那就得让 InfluxDB 服务暴露在公网。这个时

候 InfluxDB 要么本身就在公网，要么利用内网穿透。因为老师这里的演示环境是内网，所以需要搭建内网传统。

最终整体的架构如下。



这样，不管内网穿透崩了，还是 InfluxDB 崩了都会触发报警。

19.6.4 搭建内网穿透

本教程采用花生壳提供的内网穿透工具来实现内网穿透。一个新的花生壳账号，有免费的内网穿透额度，而且免费提供的 1M 大小的带宽

19.6.4.1 安装花生壳内网穿透客户端

访问官网下载页面：<https://hsk.oray.com/download>



注意选择和自己的系统匹配的安装包，我们演示的是使用的 CentOS，所以此处我选择 CentOS Linux(x86_64)



使用下面的命令安装 deb 包。

```
sudo rpm -ivh ./phddns_5.2.0_amd64.rpm
```

安装完成后，会自动开启一个名为 Phtunnel 的服务，并且你会拥有一个可以控制这项服务的命令行工具，叫做 phddns。所有相关的信息，都显示在安装好后的打印出来的提示信息里了。

```
+-----Phtunnel start install-----+
Updating / installing...
 1:phddns-5.2.0-1                               ##### [100%]
Binary file /proc/1/cmdline matches
Created symlink from /etc/systemd/system/multi-user.target.wants/phtunnel.service
Created symlink from /etc/systemd/system/multi-user.target.wants/phddns_minih

+-----Phtunnel Service Install Success-----+
+-----Oray Phtunnel Linux 5.2.0-----+
| SN: oraydc5a3ffe5d3d  Default password: admin |
| Usage: phddns(start|status|stop|restart|reset|version) |
| Remote Management Address http://b.oray.com |
+-----+
```

19.6.4.2 激活 SN

正常情况下，安装好后，phddns 会自动运行。可以使用 phddns status 命令查看程序的运行状态。

```
phddns status
```

如果显示 ONLINE，那就是正常运行。

注意这里的 SN 码，使我们的设备标识码。

另外，这里显示我们有一个远程的管理地址，是 <http://b.oray.com>。

在浏览器里访问这个地址。会进入一个登录页面，如下图所示：



现在，切换到 SN 登录，可以看到这里需要输入我们的设备 SN 码。刚才安装的时候也提示过我们，初始密码是 admin。现在输入 SN 码和密码，点击登录。



这里需要注册一个贝瑞账号，进行激活，这些操作请同学们自行完成。



19.6.4.3 配置内网穿透

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网

(1) 激活成功后，看到管理页面，先点击左侧工具栏的内网穿透按钮，进入内网穿透的管理面板，点击新增映射。



(2) 按照图中顺序先后操作，注意内网主机是指你刚才安装内网穿透所在的主机。试用版最高只能有 1Mbps 带宽，换算上行和下行速度应当在 128kb/s



点击确定后，会回到内网穿透的管理页面。

(3) 如果能看到下图所示的卡片，那就说明内网穿透已经配置成功。



以后我们访问 <https://1674b87n99.oicp.vip> 就相当是在访问本地的 127.0.0.1:8086 了

19.6.5 配置业务可用性检测

19.6.5.1 创建监控任务

(1) 首先回到睿象云的主页，在左侧点击图中标出的业务可用性监测平台。



(2) 来到监控任务的主页后，点击图中标出的绿色按钮（创建监控）



(3) 首先完成监控设置，此处我们要把监控地址设成刚才我们配置过内网穿透的地址。地址使用 /health，对这个地址发起 Get 请求，正常情况下会返回一个 json 格式的数据，它会告诉我们 InfluxDB 目前是否健康。另外，如果请求成功的话状态码应该为 200。最后，

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

对这个接口进行 get 请求，并不需要 token 加持。

监控类型设置

监控类型: API监控

监控名称: InfluxDB健康状况

监控地址: https://r603j59799.zicp.fun/health

GET

模板选择: 可选择同类监控作为模板

(4) 响应部分设置如下图所示，解释一下，这里的意思就是如果接口 2 秒内完成响应那说明速度比较满意，如果在 2~5 秒之间说明比较慢，如果大于 5 秒说明非常缓慢。

响应部分设置

2. 2秒内响应说明服务正常

基础配置 结果验证

● 正常响应时限: 2000 毫秒 注: apdex满意样本时间

● 缓慢响应时限: 5000 毫秒 注: apdex容忍样本时间

1. 点击基础配置

3. 响应时间大于5秒说明服务响应缓慢

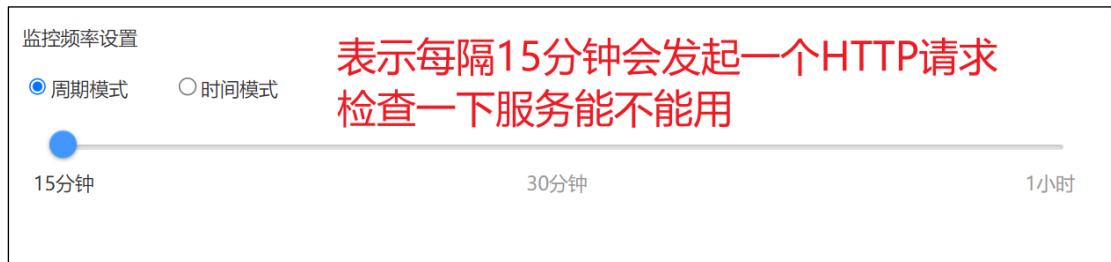
(5) 点击右上角的结果验证，设置响应码为 200，也就是说响应码为 200 是我们期望的正常状态。

响应部分设置

基础配置 结果验证

1 响应码: 200

(6) 监控频率设置使用 15 分钟，其实免费版最快只能每隔 15 分钟访问一次，充值后可以获得更高的访问频率



(7) 运营商与监控区域，是指你要使用哪个省份、哪个运营商的网络对你的接口发起访问，因为有的时候一个接口，可能移动的网络可以访问，联通的网就访问不通。最后我们只选择一台主机。如下图所示。



(8) 上述操作都完成后，点击右上角的保存按钮。



19.6.5.2 配置报警规则

(1) 回到监控列表后，可以看到页面上已经有一个监控项了。

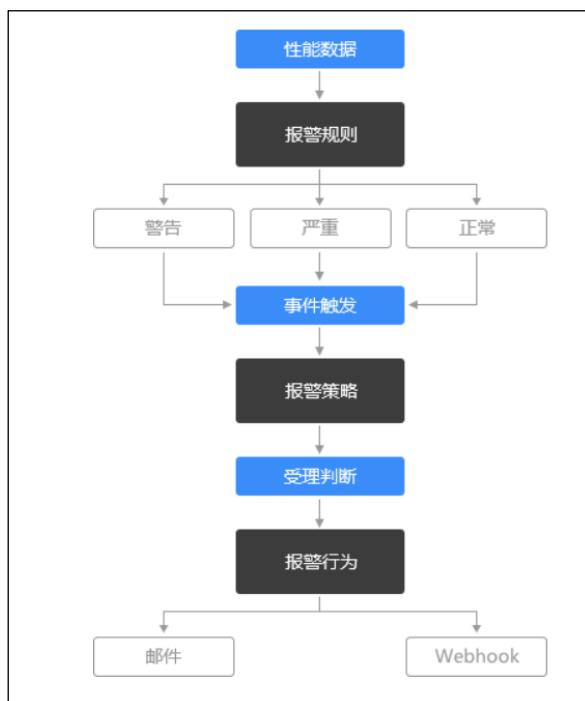
搜索	运行中	已关闭	低可用	批量开启	批量关闭	创建监控	设置告警						
编号	状态	名称	类型	监控内容	平均响应时间	可用性	监控点	告警	开关	编辑	查看	报告	删除
1	□	InfluxDB健康状况	API 监控	https://r603j59799.zicp.fun/health	78.00 ms	100.00%	1	0	□	编辑	查看	报告	删除

(2) 现在点击左侧的告警按钮，我们来配置告警的通道。

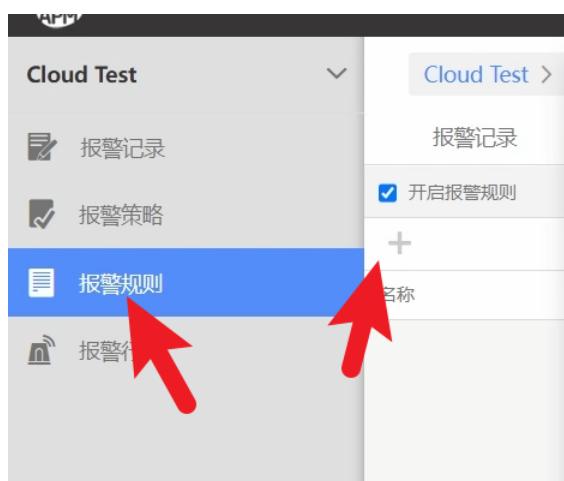
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：尚硅谷官网



(3) 可以看到，我们面对着 3 个概念：报警规则、报警策略、报警行为。和我们的 InfluxDB 一样，这里的报警规则对应着检查任务，它将数据翻译为警告、严重和正常三种信号。报警行为相当于报警终端，你可以选择发送邮件还是拨打电话。报警策略就是用来连接报警规则和报警行为的，它相当于 InfluxDB 中的报警规则。



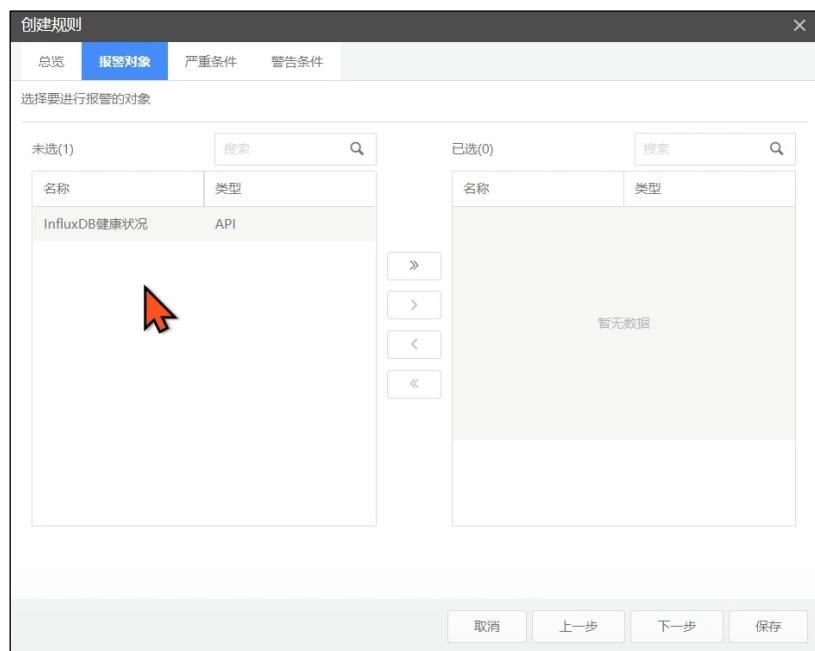
(4) 首先我们来配置报警规则首先，点击左侧的报警规则按钮，然后点击+号



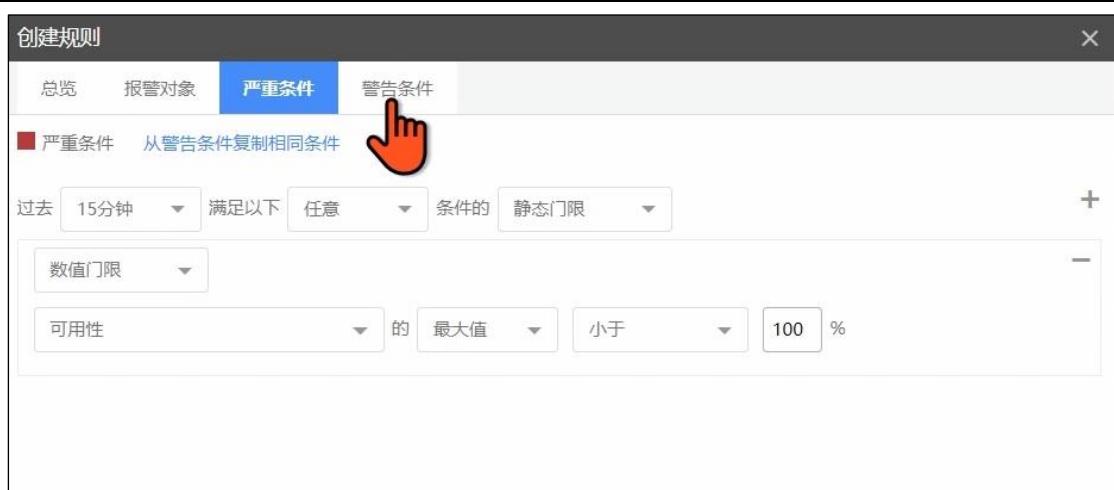
(5) 给规则命名、并在规则类型上选择 API 监控。



(6) 点击上方选项卡的报警对象，可以看到这里左边是已经创建的 API 监控类型的监控任务，选中左边的 InfluxDB 健康状况，再点击中间的>> 按钮，将它加到已选列表里来，再点击下一步。

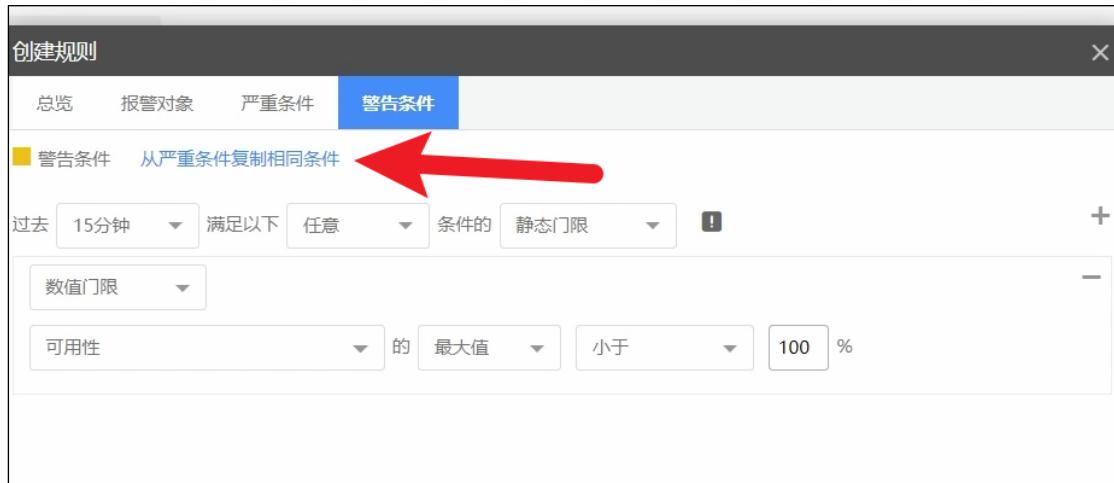


(7) 可以看到，这里要设置严重条件，这相当于我们在 InfluxDB 里面设置 CRIT 的阈值，此处我们设置为，如果过去 15 分钟的可用率不是 100%，那么就认为已经发生严重错误。



如图所示，再次点击右下角的下一步。

(8) 这一步叫做警告条件，相当于 InfluxDB 中的 warn。此处我们直接点击蓝色的按钮，从严重条件复制相同条件，然后再右下角点击保存。

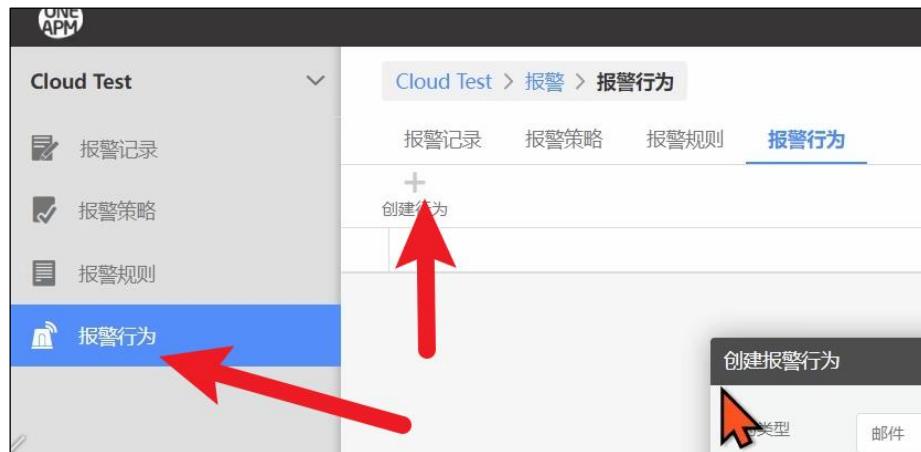


(9) 最后，一切正常的话，我们的报警规则列表中就会多出一条，如下图所示。

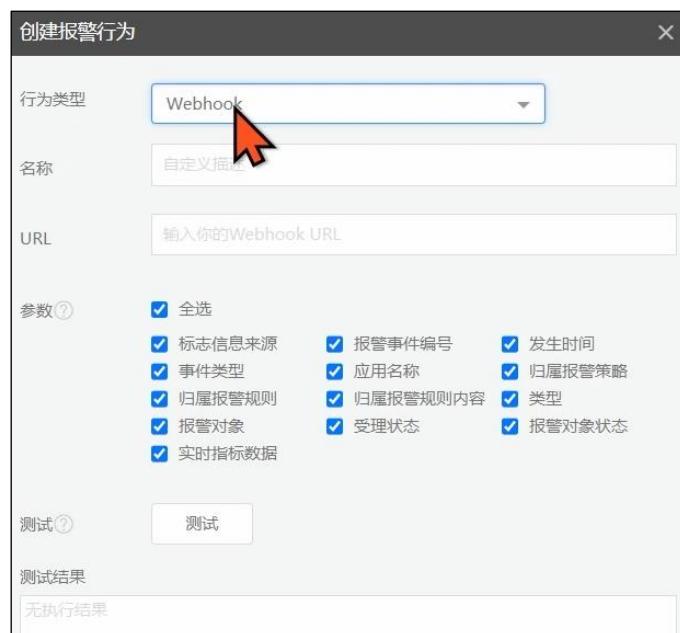


19.6.5.3 配置报警行为

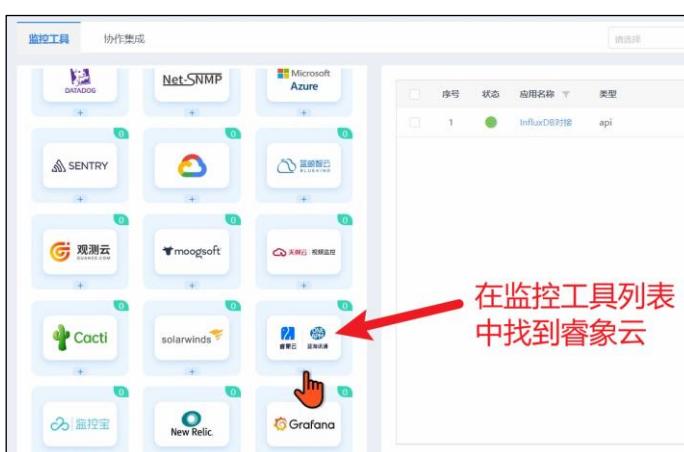
(1) 点击左侧的报警行为按钮，然后点击+号，创建一个新的报警行为。



(2) 在弹出的窗口上先选好行为类型，此处我们选择 Webhook。可以看到这里需要一个 URL，它也相当于我们要向外发送一个请求。所以这个 URL 指定谁，其实还是应该对接到睿象云来做处理。



(3) 回到智能告警平台的集成页面，在集成工具中找到睿象云。点一下创建。



(4) 先命好名，然后直接点击下方的保存并获取应用 key。



(5) 可以看到配置说明上的 url 自动补全了一个随机字符串，这个就是 key。现在将它复制一下。



(6) 回到创建报警行为的窗口中，填写 URL，再点击测试，如果测试结果显示为 connect success，那就说明我们的配置是正确的。点击右下角的保存。



19.6.5.4 修改分派策略

现在我们的告警平台中，已经创建了两个应用了。但是直接我们之前创建过一个分派策略，它会将 REST API 收到的报警通知全部转发给某个用户。但是我们刚才创建的 Webhook 还尚未包含在这个分派策略中。

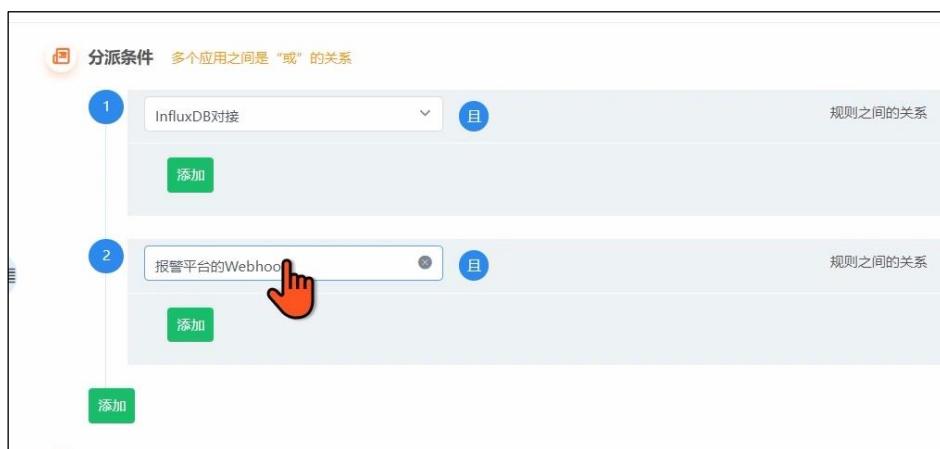
所以此处，找到我们之前设置的分派策略，点击右边的编辑按钮。



进入编辑页面后，点击图中指出的添加按钮。



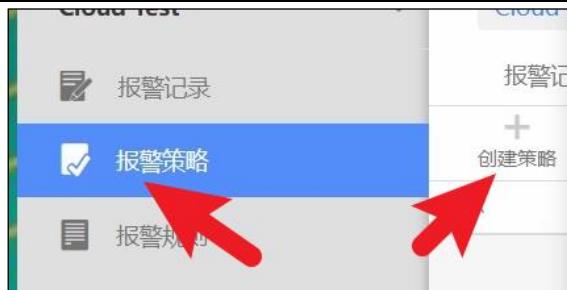
可以看到，此处我们就会展示出来两个应用，也就是这两个接口收到的报警通知都会转发给我们指定的用户。



最后点击保存，分派策略就修改成功了。

19.6.5.5 创建报警策略

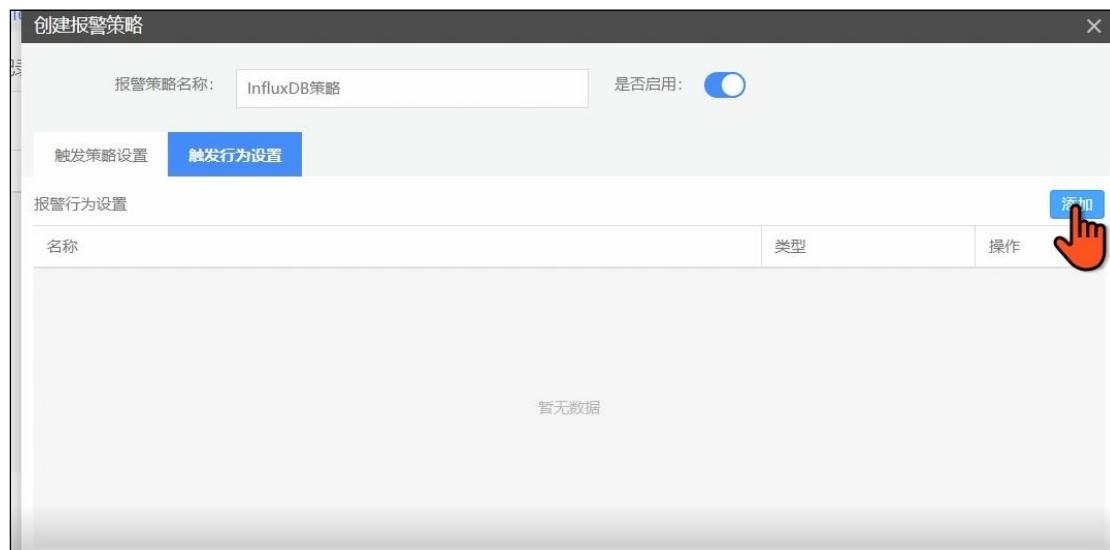
(1) 回到之前的监控平台，告警管理页面。首先点击左侧的报警策略按钮，再点击+号，创建一个报警策略。



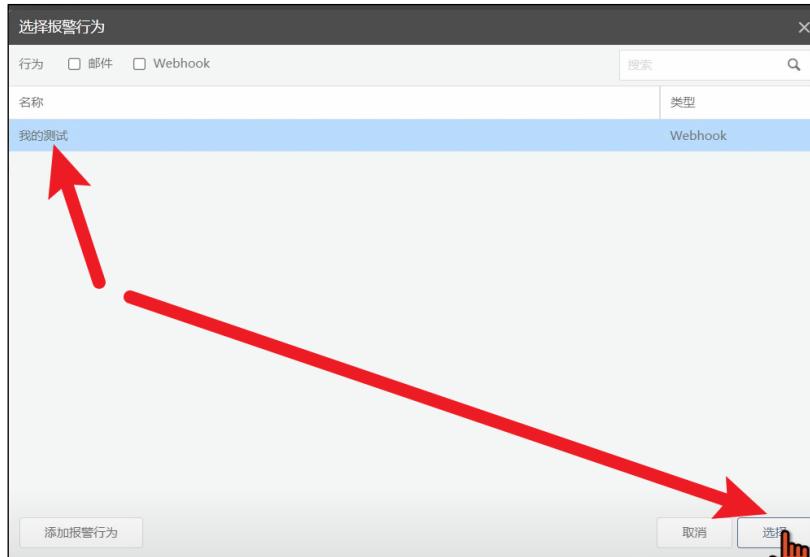
(2) 触发策略的配置如下图所示。



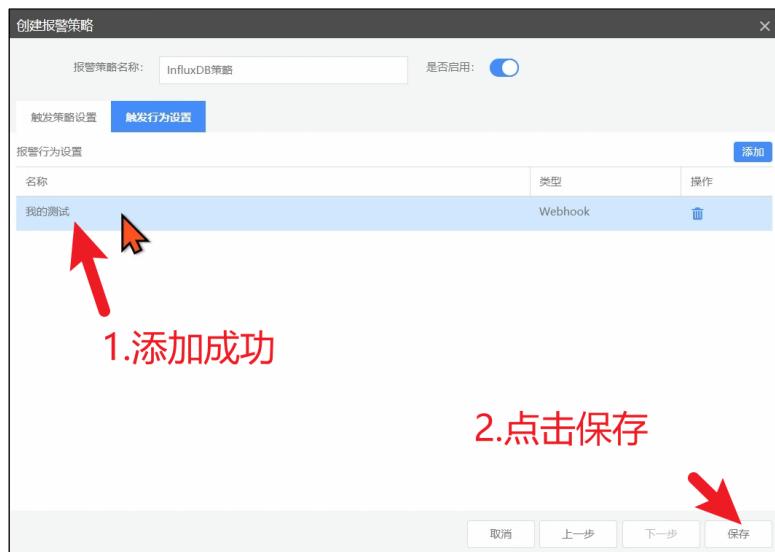
(3) 点击上方的触发行为设置按钮，然后点击右侧的添加按钮。



(4) 可以看到有一个可以进行选择的选项卡，这是我们之前创建的报警行为，鼠标点一下将它选中，然后点击右下角的选择按钮。



(5) 如果报警行为成功添加，就点击右下角的保存按钮。至此我们的报警策略就搭建成功了。



19.6.6 小结

经过上述操作，我们的报警架构就已经完成升级了，感兴趣的同学可以自行尝试一下把 InfluxDB 挂掉会发生怎样的效果。

第20章 附录 1：拓展的带注释的 CSV

20.1 什么是拓展的带注释的 CSV

拓展的带注释的 CSV 在 CSV 的基础之上提供了额外的注释和选项，用于制定应该如何将 CSV 数据转换为行协议并写入 InfluxDB。

20.2 FLUX 为什么想给 CSV 加注释

下面是一套典型的 CSV 文件。

```
m,host,used_percent,time  
mem,host1,64.23,2020-01-01T00:00:00Z  
mem,host2,72.01,2020-01-01T00:00:00Z  
mem,host1,62.61,2020-01-01T00:00:10Z  
mem,host2,72.98,2020-01-01T00:00:10Z  
mem,host1,63.40,2020-01-01T00:00:20Z  
mem,host2,73.77,2020-01-01T00:00:20Z
```

对编程语言和数据库来说，数据的类型非常重要。但是常见的 CSV 并没有告诉你，我哪一列需要用 Long 哪一列需要用 String。所以在大多数的编程语言里，通常编程语言的 CSV 包内会有下面的两种基本功能。

- (1) 自动推断 CSV 文件中可能的数据类型并将其解析。
- (2) 允许用户向解析方法传递参数，显示声明 CSV 文件中的哪些列需要解释为 int，哪些列需要解释为 string。

不过 FLUX 的思路是额外创建一个数据格式。

在传统的 CSV 文件的最前面，加上两三行，显示说明这些字段都是什么数据类型。

这就是 FLUX 的带注释的 CSV 数据格式的基本思想。

除此之外，拓展的带注释的 CSV 还支持一些 InfluxDB 的其他特性。

20.1 带注释 CSV 的作用

当你查询 InfluxDB 时，返回的数据格式其实就是带注释的 CSV。不过通常情况下，各种编程语言的客户端库和可视化工具会帮我们处理这种数据。所以带注释的 CSV 对用户来说属于一个偏底层的数据格式。

20.2 4 种拓展注释

20.2.1 数据类型

如果要将某列数据类型对应到 InfluxDB 中的类型，需要在 CSV 数据的最前面加上 #datatype 注释。

比如：

```
#datatype string double long
```

表示第一列是字符串，第二列是双精度浮点数，第三列是 long 整数。

下面这些是拓展的带注释的 CSV 支持的所有数据类型

数据类型	映射到 InfluxDB 中的数据类型
------	---------------------

measurement	说明这一列是 InfluxDB 中的 measurement
tag	说明这一列是 InfluxDB 中的 tag (标签)
dateTime	说明这一列是 InfluxDB 中的 timestamp (时间戳)
field	说明这一列是 InfluxDB 中的 field (字段)
ignored	不映射 InfluxDB 中的数据类型，这一列会被忽略。
string	说明这一列是 InfluxDB 中 string 类型的 field
double	说明这一列是 InfluxDB 中的 float 类型的 field
long	说明这一列是 InfluxDB 中的 integer 类型的 field
unsignedLong	说明这一列是 InfluxDB 中的 unsigned integer field
boolean	说明这一列是 InfluxDB 中的 boolean field

其中。

- datetime 后面可以加上一个日期格式告诉程序该如何解析日期时间。

比如

```
#datatype dateTime:RFC3339
#datatype dateTime:RFC3339Nano
#datatype dateTime:number
#datatype dateTime:2006-01-02
```

- double, long, unsignedLong 可以用小数点后面跟上一个英文逗号表示数据中有千位分隔符。这个时候类型注释信息需要用双引号引起来

比如

```
#datatype "long:., "
```

- boolean 布尔值可以指明哪些值是 True, 哪些值是 False

比如

```
#datatype "boolean:y,Y,1:n,N,0"
```

就表示, 如果这一列的值是 y 或 Y 又或者 1, 那么它就对应 boolean 中的 True。如果这一列的是 n 或 N 或 0, 那么就对应 boolean 中的 False。同样, 这个语法中出现了逗号, 写的时候应该把它用双引号括起来。

20.2.1.1 严格模式

所有的类型后面都可以加上一个:strict 关键字, 表示如果数据不符合 long 格式, 比如 1.2 那么整行数据的导入直接失败, 如果不加的话, 程序一般会把 1.2 截断以适应 long。

20.2.2 常量

使用#constant 注释, 相当于为 csv 中的每行数据加上一个固定的数据。比如, 我 csv 里的数据全部都是要去同一个名为 m 的 measurement 的。那么是其实是没有必要在每一行

都加上一个值为 m 的列。我可以在 csv 的头部加上这么一行。

```
#constant measurement,m
```

那么所有的数据的 measurement 就都为 m。

如果你要加上多个常量，那么需要让每个#constant 注释独占一行。

比如

```
#constant measurement,m  
#constant tag,dataSource, csv
```

它的语法其实是

```
#constant <datatype>,<column-label>,<column-value>
```

但是 measurement 比较特殊。

20.2.3 时区

因为 CSV 里面的数据是日期时间类型，InfluxDB 中的数据必须时间戳。这个时候是有必要指定时区的。

你可以在 CSV 数据的头部加一行

```
#timezone ±HMM
```

指定当前数据相对于 UTC 时区的偏移量。

20.2.4 字段拼接

使用#concat 可以使用已有列拼接出来一个新列。它引用列名的语法跟 bash shell 是一个风格。

语法：

```
#concat, string, fullName, ${firstName} ${lastName}
```

这个意思就是说，给 csv 添加一个新的 string 类型的列，名为 fullName。它由 firstName 的值+空格+lastName 的值组成。

通常来说，这个语法拼日期更好用一些。

例如：

```
#concat,dateTime:2006-01-02,${Year}-${Month}-${Day}
```

```
Year,Month,Day,Hour,Minute,Second,Tag,Value  
2020,05,22,00,00,00,test,0  
2020,05,22,00,05,00,test,1  
2020,05,22,00,10,00,test,2
```

20.3 自定义分隔符

可以在 CSV 的头部加一行 sep=;

表示把 CSV 的分隔符从默认的 , 切换到 ;

```
sep=;
```

更多 Java –大数据 –前端 –python 人工智能资料下载，可百度访问：尚硅谷官网

20.4 简短注释

拓展的注释还有更简短的写法

那就是把类型信息什么的加载每个列列名的后面。

比如：

```
m|measurement,location|tag|Hong Kong,temp|double,pm|long|0,time|dateTime:RFC3339
weather, San Francisco, 51.9, 38, 2020-01-01T00:00:00Z
weather, New York, 18.2, , 2020-01-01T00:00:00Z
weather,, 53.6, 171, 2020-01-01T00:00:00Z
```

意思就是

- m 列是 measurement，并且没有默认值
- localtion 列是一个默认值为 Hong Kong 的 tag
- temp 列的类型是 double
- pm 列的类型是 long 而且默认值是 0
- time 列是一个时间戳，而且数据格式是 RFC3339 格式的，没有默认值。

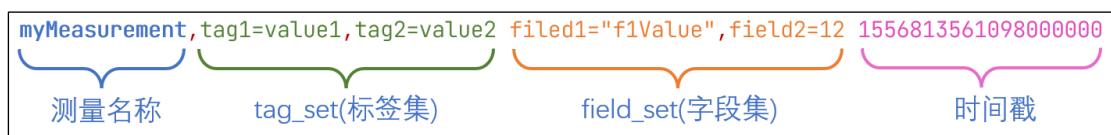
第21章 附录 2： InfluxDB 行协议

21.1 认识 InfluxDB 行协议

InfluxDB 行协议是 InfluxDB 数据库独创的一种数据格式，它由纯文本构成，只要数据符合这种格式，就能使用 InfluxDB 的 HTTP API 将数据写入数据库。

与 CSV 相似，在 InfluxDB 行协议中，一条数据和另一条数据之间使用换行符分隔，所以一行就是一条数据。另外，在时序数据库领域，一行数据由下面 4 种元素构成。

- (1) measurement (测量名称)
- (2) Tag Set (标签集)
- (3) Field Set (字段集)
- (4) Timestamp (时间戳)



下面我们详细介绍一下各个元素的作用。

21.2 measurement (测量名称)

必需

测量的名称。

你可以将它当作普通关系型数据的 table，虽然实际上不是这么回事。

在 InfluxDB 行协议中，测量名称不可省略。

大小写敏感

不可以用下划线_打头

21.3 Tag Set (标签集)

标签应该用在一些值的范围有限（可枚举）的，不太会变动的属性上。比如传感器的类型和 id 等等。在 InfluxDB 中一个 Tag 相当于一个索引。给数据点加上 Tag 有利于将来对数据进行检索。但是如果索引太多了，就会减慢数据的插入速度。

可选

键值关系使用=表示

多个键值对之间使用英文逗号 , 分隔

标签的键和值都区分大小写

标签的键不能以下划线_开头

键的数据类型：字符串

值的数据类型：字符串

21.4 Field Set (字段集)

必需

一个数据点上所有的字段键值对，键是字段名，值是数据点的值。

一个数据点至少要有一个字段。

字段集的键是大小写敏感的。

字段

键的数据类型：字符串

值的数据类型：浮点数 | 整数 | 无符号整数 | 字符串 | 布尔值

21.5 Timestamp (时间戳)

可选

数据点的 Unix 时间戳，每个数据点都可以制定自己的时间戳。

如果时间戳没有指定。那么 InfluxDB 就使用当前系统的时间戳。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

数据类型: Unix timestamp

如果你的数据里的时间戳不是以纳秒为单位的, 那么需要在数据写入时指定时间戳的精度。

21.6 空格

行协议中的空格决定了 InfluxDB 如何解释数据点, 第一个未转义的空格将测量值&Tag Set (标签集) 与 Field Set (字段集) 分开。第二个未转义空格将 Field Set (字段级) 和时间戳分开。



21.7 协议中的数据类型及其格式

21.7.1 Float (浮点数)

IEEE-754 标准的 64 位浮点数。这是默认的数据类型。

示例: 字段级值类型为浮点数的行协议

```
myMeasurement fieldKey=1.0
myMeasurement fieldKey=1
myMeasurement fieldKey=-1.234456e+78
```

21.7.2 Integer (整数)

有符号 64 位整数。需要在数字的尾部加上一个小写数字 i。

整数最小值	整数最大值
-9223372036854775808i	9223372036854775807i

示例: 字段值类型为有整数的

21.7.3 UInteger (无符号整数)

无符号 64 位整数。需要在数字的尾部加上一个小写数字 u。

无符号整数最小值	无符号整数最大值
0u	18446744073709551615u

示例: 字段值类型为无符号整数的航协议

```
myMeasurement fieldKey=1u
myMeasurement fieldKey=12485903u
```

21.7.4 String (字符串)

普通文本字符串，长度不能超过 64KB

示例：

```
# String measurement name, field key, and field value
myMeasurement fieldKey="this is a string"
```

21.7.5 Boolean (布尔值)

true 或者 false。

示例：

布尔值	支持的语法
True	t, T, true, True, TRUE
False	f, F, false, False, FALSE

示例：

```
myMeasurement fieldKey=true
myMeasurement fieldKey=false
myMeasurement fieldKey=t
myMeasurement fieldKey=f
myMeasurement fieldKey=TRUE
myMeasurement fieldKey=FALSE
```

不要对布尔值使用引号，否则会被解释为字符串

21.7.6 Unix Timestamp (Unix 时间戳)

如果你写时间戳，

```
myMeasurementName fieldKey="fieldValue" 1556813561098000000
```

21.8 注释

以井号 # 开头的一行会被当做注释。

示例：

```
# 这是一行数据
myMeasurement fieldKey="string value" 1556813561098000000
```

第22章 附录 3：Prometheus 数据格式

22.1 认识 Prometheus 数据格式

Prometheus 也是一种时序数据库，不过它通常被用在运维场景下。Prometheus 是开放原子基金会的第二个毕业项目，这个基金会的第一个毕业项目就是大名鼎鼎的 k8s。

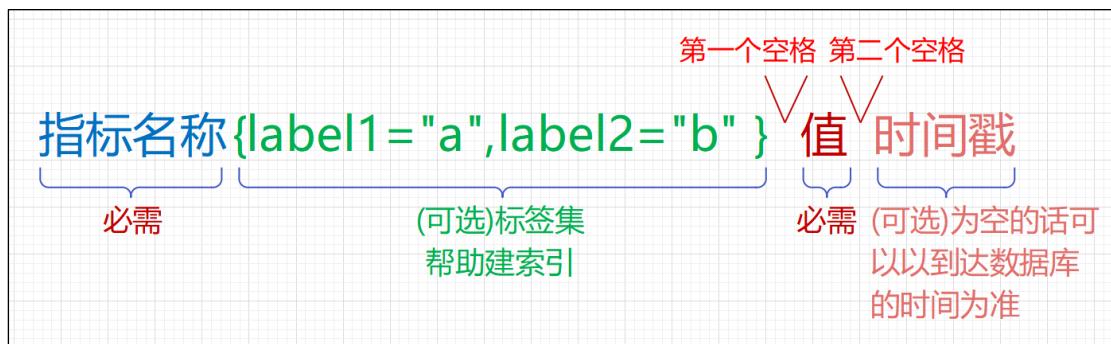
同 InfluxDB 一样，Prometheus 也有自己的数据格式，只要数据符合这种格式就能被 Prometheus 识别并写入数据库。而且 Prometheus 数据格式也是纯文本的。近期 Prometheus

技术热度高涨，有一个名为 OpenMetrics 的数据协议越来越流行，它致力于让全球的指标监控有一样的数据格式，而这个数据协议就是根据 Prometheus 数据格式改的，两者 100% 兼容，足以见其影响力。

22.2 Prometheus 数据格式的构成

Prometheus 数据格式主要包含四个元素

- (1) 指标名称（必需）
- (2) 标签集（可选）：标签集是一组键值对，键是标签的名称，值是具体的标签内容，而且值必须得是字符串。指标名称和标签共同组成索引。
- (3) 值（必须）：必须满足浮点数格式
- (4) 时间戳（可选）：Unix 毫秒级时间戳。



- (1) 第 1 个空格，将 指标名称&标签集 与 指标值 分隔开
- (2) 第 2 个空格，将 指标值 与 Unix 时间戳 分隔开

第23章 附录 4：时序数据库中的数据模型

想要正确使用时序数据库，就必须理解时序数据库管理数据的逻辑。

这里，我们会和普通的 SQL（关系型）数据库做一下比较。

23.1 普通关系型数据库中的表

下面这张表示 SQL（关系型）数据库中一个简单的示例。表中有创建索引和未创建索引的列。

- park_id、planet、time 是创建了索引的列。
- _foodships 是未创建索引的列。

park_id	planet	time	#_foodships
1	Earth	14291856000000000000	0
1	Earth	14291856010000000000	3

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

	1	Earth		14291856020000000000		15	
	1	Earth		14291856030000000000		15	
	2	Saturn		14291856000000000000		5	
	2	Saturn		14291856010000000000		9	
	2	Saturn		14291856020000000000		10	
	2	Saturn		14291856030000000000		14	
	3	Jupiter		14291856000000000000		20	
	3	Jupiter		14291856010000000000		21	
	3	Jupiter		14291856020000000000		21	
	3	Jupiter		14291856030000000000		20	
	4	Saturn		14291856000000000000		5	
	4	Saturn		14291856010000000000		5	
	4	Saturn		14291856020000000000		6	
	4	Saturn		14291856030000000000		5	

23.2 InfluxDB 中的数据表示

上一节的数据，如果换到 InfluxDB 中，会换一种形式进行表示。

```

name: foodships
tags: park_id=1, planet=Earth
time                      #_foodships
-----
2015-04-16T12:00:00Z      0
2015-04-16T12:00:01Z      3
2015-04-16T12:00:02Z      15
2015-04-16T12:00:03Z      15

name: foodships
tags: park_id=2, planet=Saturn
time                      #_foodships
-----
2015-04-16T12:00:00Z      5
2015-04-16T12:00:01Z      9
2015-04-16T12:00:02Z      10
2015-04-16T12:00:03Z      14

name: foodships
tags: park_id=3, planet=Jupiter
time                     #_foodships
-----
2015-04-16T12:00:00Z     20
2015-04-16T12:00:01Z     21
2015-04-16T12:00:02Z     21
2015-04-16T12:00:03Z     20

name: foodships
tags: park_id=4, planet=Saturn
time                     #_foodships
-----
2015-04-16T12:00:00Z     5
2015-04-16T12:00:01Z     5
2015-04-16T12:00:02Z     6
2015-04-16T12:00:03Z     5

```

你可以这样理解。

- InfluxDB 中的 measurement (foodships) 相当于 SQL (关系型) 数据库中的表

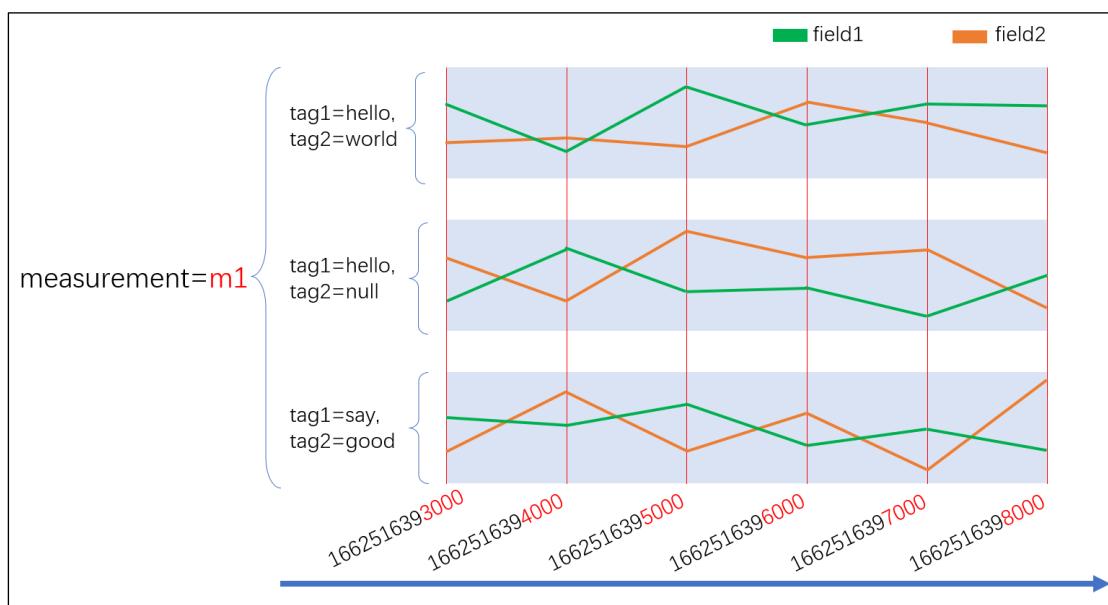
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- InfluxDB 中的 tags (park_id 和 planet) 相当于 SQL (关系型) 数据库中的索引列
- InfluxDB 中的 fields (在这里是#_foodships) 相当于 SQL (关系型) 数据库中的未建索引的列。
- InfluxDB 中的数据点 (比如, `2015-04-16T12:00:00Z 5←`) 相当于 SQL (关系型) 数据库中的一行。

23.3 理解序列的概念至关重要

简单来说, InfluxDB 这类数据库是用序列的方式来管理数据的。在 InfluxDB 中, 唯一的 measurement, tag_set 和 field (一个字段) 组合是一个 series (序列)。比如下图中有 6 条连续的线, 这里面每个条线就是一个序列。每一个序列的数据在内存和磁盘上紧密存放, 这样当你要查询这一序列的数据时, InfluxDB 可以很快定位到这一序列中的好多条数据。

你也可以将 measurement, tag, field 视为索引, 而且它们本身就是索引。



以序列的方式管理数据是时序数据库和传统关系型数据库最不同的地方。传统的关系型数据库通常是以 record (记录或者行) 的方式管理数据, 这个时候, 关系型数据库可以让你快速地通过索引定位到一条数据。

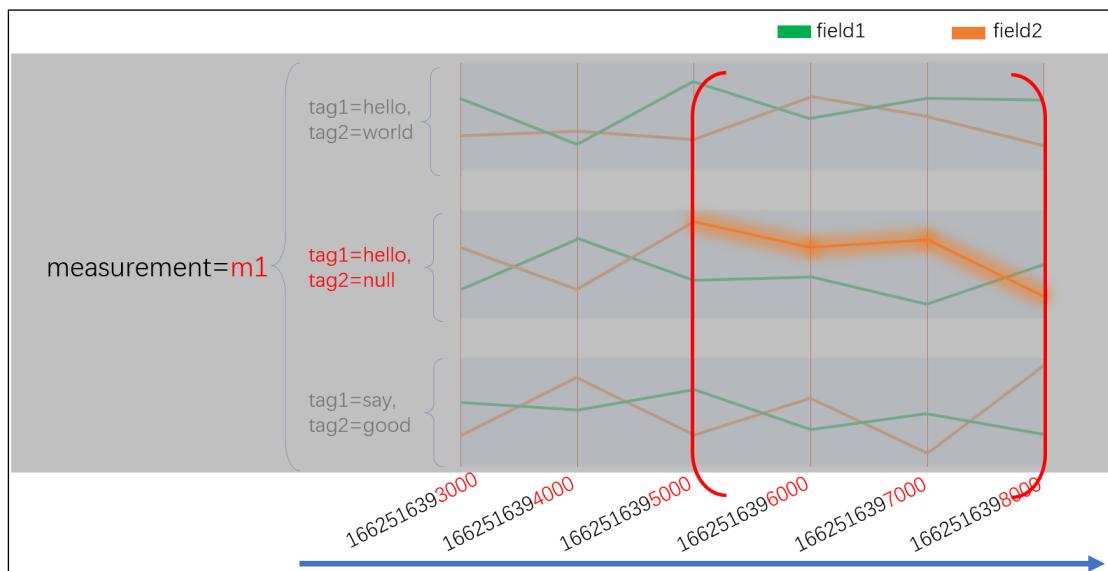
measurement	tag1	tag2	field1	field2	time
m1	hello	world	9	46	1662516393000
m1	hello	world	30	29	1662516394000
m1	hello	world	39	31	1662516395000
m1	hello	world	14	33	1662516396000
m1	hello	world	1	24	1662516397000
m1	hello	world	24	9	1662516398000
m1	hello	null	42	49	1662516393000
m1	hello	null	33	30	1662516394000
m1	hello	null	30	22	1662516395000
m1	hello	null	46	47	1662516396000
m1	hello	null	1	14	1662516397000
m1	hello	null	24	14	1662516398000
m1	say	good	28	41	1662516393000
m1	say	good	36	13	1662516394000
m1	say	good	11	2	1662516395000
m1	say	good	41	12	1662516396000
m1	say	good	28	10	1662516397000
m1	say	good	43	33	1662516398000

但是在时序场景下，我们通常需要查找某个设备最近一段时间的数据。这个时候对于传统关系型数据库来说，很可能需要多次寻址来找到多个 record 才能完成查询。而时序库是把索引打到一批次的数据上，所以在这种场景下的读写，时序库性能是远强于 B+树数据库的。

23.4 双索引设计与高效查询思路

我们之前说到你可以将 measurement、tag_set 和 field 视为索引，还没有提到最重要的时间。其实，在 InfluxDB 中时间也是索引，数据在入库时，会按时间戳进行排序。这样，我们在进行查询时，一般遵循下面的思路。

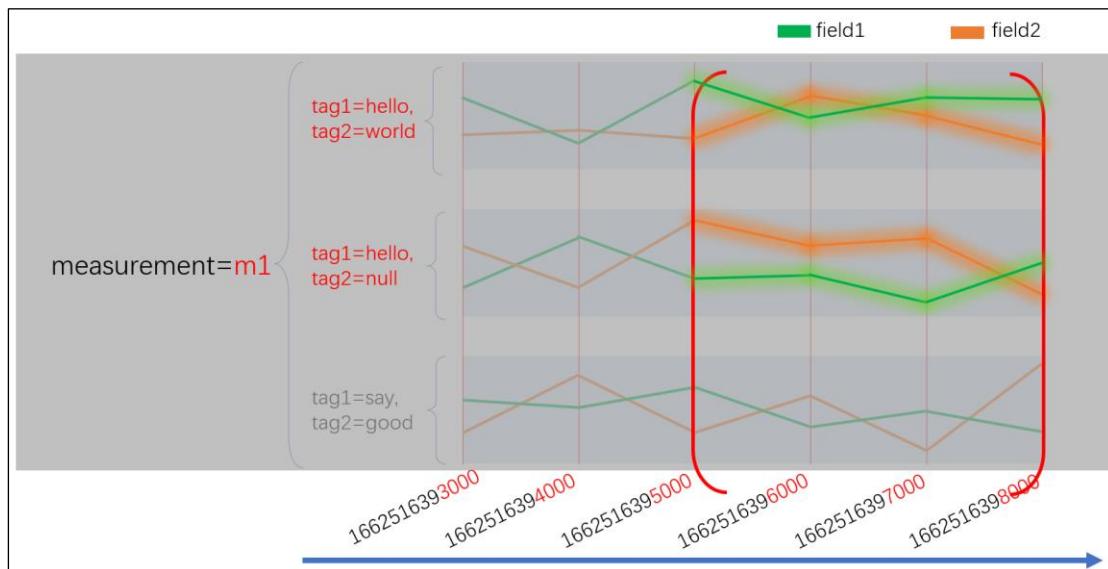
- (1) 先指定要从哪个存储桶查询数据
- (2) 指定数据的时间范围
- (3) 指定 measurement、tag_set、和 field 说明我要查询哪个序列。



23.5 我一次只能查询一个序列吗

一次只能查询一个序列，这显然是不合理的。

假如，我现在只指定要查询 measurement 为 m1 和 tag1 为 hello 的数据，那么就会命中图中 4 条序列。所以实际上，measurement, tag, field 都是倒排索引。



23.6 时间线膨胀（高基数问题）

时间线膨胀是所有时序数据库都绕不过的问题。简单地来解释时间线膨胀，就是我们的时序数据库中序列太多了。

当序列过多时，时序数据库的写入和读取性能通常都会有明显的下降。所以，当你去网上看一些时序数据库的压测文章时，需要注意文章有没有将序列数考虑进去。

第24章 附录 5：时间的标准与格式

本附录是专门向大家讲解时间的相关知识的。

24.1 GMT（格林威治标准时间）

格林威治（又译格林尼治）它是一个位处英国伦敦的小镇。

17 世纪，英国航海事业发展迅速，当时海上航行亟需精确的精度指示，于是英国皇家在格林威治这个地方设立了一个天文台负责测量正确经度的工作。

后来 1884 年，在美国华盛顿召开的国际经度会以决定以经过格林尼治天文台（旧址）的经线为本初子午线（0 度经线）。同时这次会以也将全球划分为了 24 个时区。0 度经线所在的时区为 0 时区。

现在，有时候你要买一个机械表，如果说支持 GMT，意思就是支持显示格林威治标
更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

准时间。



24.2 UT (世界时)

1928年，国际天文联合会提出了UT的概念，UT主要用来衡量一天究竟有多长。一旦一天的长度可以确定，那么将这个长度除以24就能确定一小时的长度。以此类推、分钟、秒的长度我们就都能确定了。

UT也是以格林威治时间作为标准的，它规定格林威的子夜为0点。

在当时，衡量一天长度的方法就是通过天文观测，看地球多久转一圈。但一来天文观测存在误差。二来，地球的自转越来越慢。计时方法亟需革新。

24.3 计时技术与国际原子时

人类历史上出现的计时手段大体上能分为三类

- 一是试图通过某种匀速的运动来表示时间、比如沙漏、水钟、香钟（烧香）。这种方式的缺陷很大，是一种很粗略的时间衡量方法
- 二是通过天文观测，通过日月或其他星辰的参考确定时间。现在我们已经知道，星系的运动也不是匀速的过程。
- 三是通过固定频率的震动，最早是伽利略通过教堂的吊灯发现了摆的等时性，也就是摆角较小时，吊灯摆动一次的时间是相同的。距今三四百年前的摆钟，基本上都是利用这一原理实现的。

现在，人类已知的最精确的计时技术是原子钟，它以原子共振频率标准来计算和保持时间的准确。它的精度可以达到持续运行上亿年而误差不超过1秒。

基于这种技术，后来国际计量协会结合了全球400多个原子钟，规定1秒为铯-133原

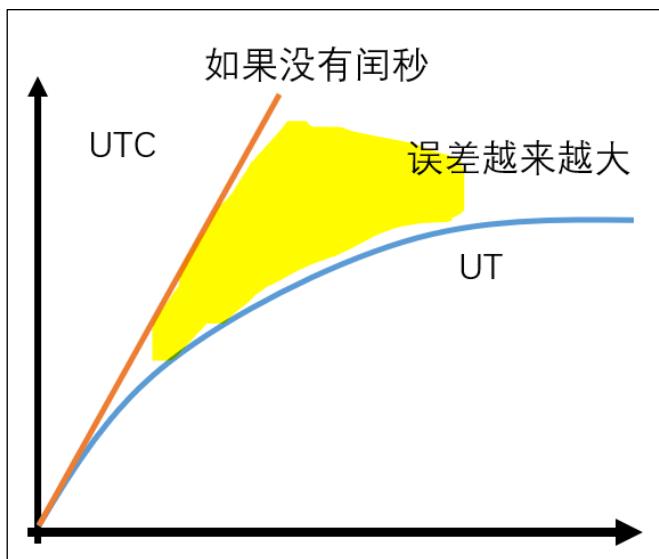
子基态两个超精细能级间跃迁辐射震荡 9,192,631,770 周所持续的时间。这个定义就叫国际原子时（International Atomic Time， TAI）。这样，我们钟表里指针应该转多快也有了一个统一的标准。

国际原子时的秒长以格林威治时间 1958 年 1 月 1 日 0 时的秒长为基准。也就是规定，在这一瞬间，国际原子时的秒长和世界时的秒长是一样的。

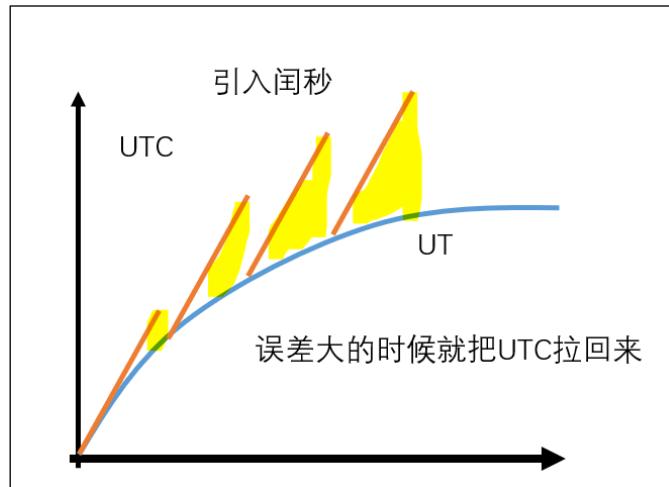
24.4 UTC（世界协调时）

UTC，universal Time Coordinated。世界协调时，世界统一时间、国际协调时。它以国际原子时的秒长为基准。但是我们知道，UT 基于天文观测，地球越来越慢那 UT 的秒长应该越来越长。如果不进行干预那么 UTC 和 UT 之间就会有越来越大的误差，

如下图所示：



如果这种状况持续下去，在好多好多好多年后，人类可能就是 UTC 时间凌晨 3 点起床挤地铁上班了。因此，让 UTC 符合人类生活习惯，就必须控制 UTC 和 UT 的误差大小，于是 UTC 引入了闰秒。所谓闰秒，也就是让在某个时间点上，人为规定这一分钟比普通的分钟多一秒，它有 61 秒。这个时候 1 分 59 秒过了应该接着是 2 分 0 秒，但是在遇到闰秒时会遇到 1 分 60 秒。



看似好像也能接受。但是何时加入闰秒是不可预测的。它是由国际地球自转服务（IERS）每隔一段时间依据实际情况决定的。对计算机的程序而言，闰秒机制具有明显的破坏性，相关国际标准机构一直在讨论是否继续这种做法。

24.5 小结：UTC/GMT

- GMT 是最早的国际时间标准，后来是 UTC。
- 因为 UTC 要逼近 UT，而 UT 又以 GMT 为标准。十分严格地说，UTC 和 GMT 不是一个东西。但宽松地说，你可以把 UTC 等同于 GMT，而且有些网站和应用程序就是这么干的。
- 因为 UTC 标准已经使用多年。所以现在如果再看到 GMT 这个词，它指的通常不是国际时间，而是格林威治所在的时区，也就是 0 时区。同时，通常行政区有很多适应自己所在地的时区缩写，遗憾的是，这种写法经常会撞车。

比如，CCT，它可以表示美国中部时间（Central Standard Time），澳大利亚中部时间（Central Standard Time），中国标准时间（China Standard Time）和古巴标准时间（Cuba Standard Time）

所以、如果我写 CCT 2022-08-03 11:56 就很容易误解了。这个时候我们非常需要一种没有歧义的日期时间写法。

24.6 时区与 UTC 偏移量

现行的时区表示更多是使用 UTC+偏移量的方式来表示的。比如北京是在东 8 区，时间比 UTC 要早 8 小时，那么在表示北京时区的方式就是 UTC+08:00。虽然地理界定上只有东西十二区，但是什么地方采用什么方式表达时间实际取决于当地的行政命令。因此 UTC+12:00 并不是偏移量的上限。打开你电脑上的日期时间设置，你会发现有的国家采

用的是 UTC+14:00。还有的国家偏移量并不完全是小时的整数倍，比如 UTC+12:45。

时间和语言 > 日期和时间



同时，也有很多应用会使用 GMT+0800 的方式表示，效果是一样的。

24.7 日期时间的表示格式

2022 年 9 月 3 日该怎么表示？是 2022/09/03 还是 2022-09-03 还是 Sep 03 2022？这又是一个标准问题，当前的情况是，各个国家有符合本地习惯的日期时间格式标准，同时国际上也有诸多日期时间格式标准，比如 ISO 8601 和 RFC3339 等。

各种格式都有软件采用，所以编程语言中的日期标准库，一般都会准备 `dateformat` 工具，自己编码日期时间的格式。

24.8 ISO 8601

国际标准 ISO 8601，是国际标准化组织的日期和时间的表示方法和我们之前提过的 UTC 不同，UTC 是一种时间标准，而 ISO 8601 是一种标准的时间格式，大多数的编程语言都支持。

使用 ISO 8601 格式可以明确表示下面的时间。

- 公历日期
- 24 小时制的时间
- UTC 时区偏移量
- 时间间隔
- 以及上面几种元素的组合。

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

ISO 8601 的表示非常灵活，这里不会将其完全列出，我们直说最常见的日期时间格式。

比如，下面就是一个符合 ISO 8601 的日期时间表示。

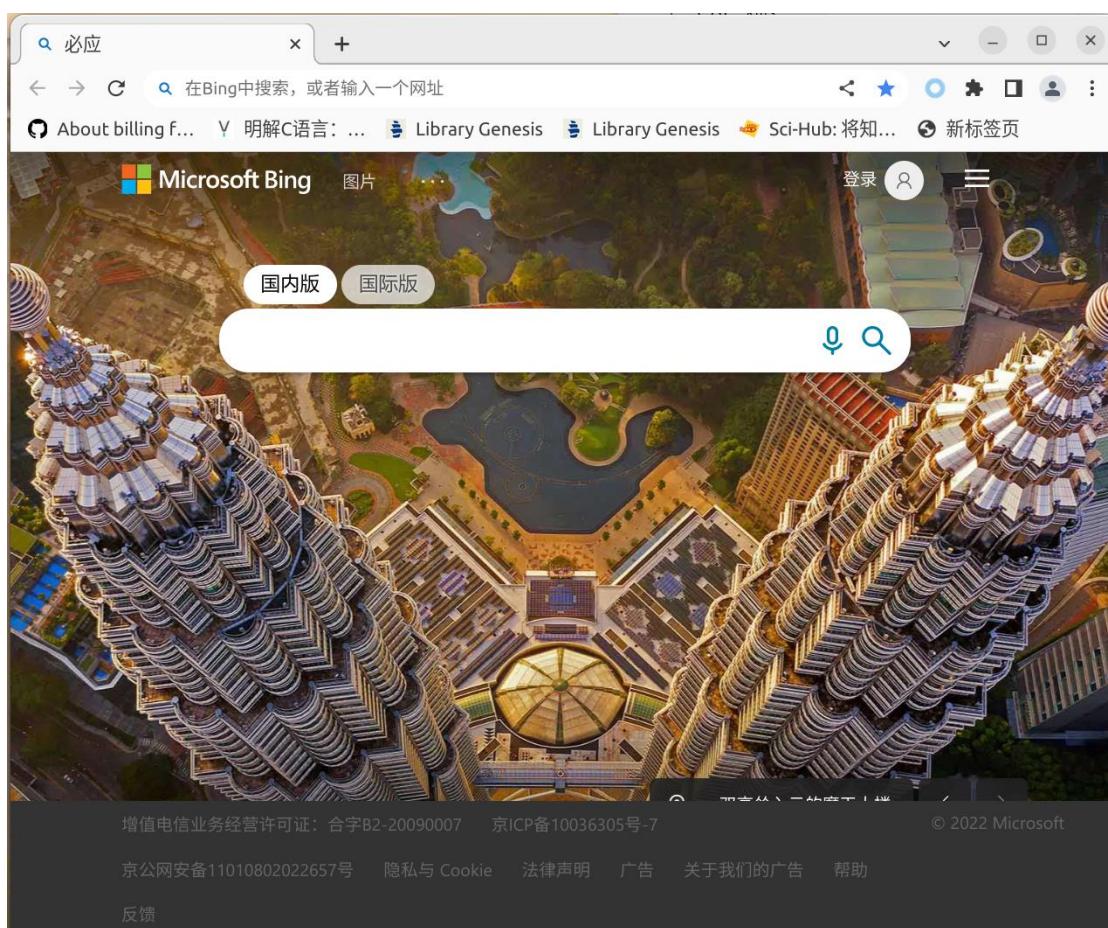
2022-09-03T14:13:00Z，这个时间戳中间的 T 用来分隔 日期 和 时间，最后字母 Z 表示 0 时区，也就是 UTC 或 GMT 时间。

24.9 示例：在编程语言中获取 UTC 时间和 ISO 格式

本例会在浏览器中用 JavaScript 操作日期对象。

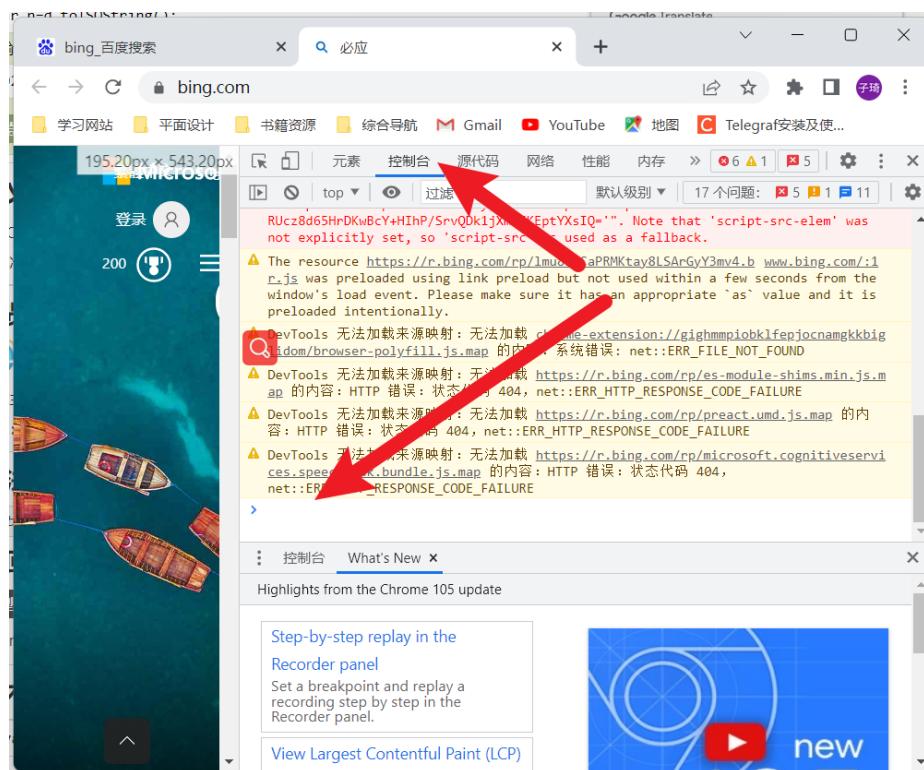
24.9.1 打开浏览器

打开浏览器，随便进入一个页面，空白标签页更好。老师这里演示时使用的 chrome 浏览器，打开的是 bing 搜索首页。



24.9.2 按 F12 打开开发者工具

按 F12 打开开发者工具，在开发者工具的上方找到 Console 按钮，点一下。如果设置了中文页面，那么应该是控制台。效果如下。



进入控制台后，有一些报错信息和警告，这些都是 bing 页面的问题，可以忽视。下方有一个蓝色的箭头，在这里，我们可以直接编写 JavaScript 代码并且立刻运行。

24.9.3 编写代码

(1) 创建一个日期对象

```
x = new Date()
```

代码敲完后，回车执行

```
> x = new Date()
< Sat Sep 03 2022 14:44:29 GMT+0800 (中国标准时间)
```

可以看到控制台有一个返回的结果，表示当前的时间是 2022 年 9 月 3 日星期六 14 点 44 分 29 秒，时区的偏移量是从格林威治时区向东偏移 8 小时。

(2) 使用下面的代码获取 ISO 标准的日期时间字符串。

```
x.toISOString()
```

```
> x.toISOString()
< '2022-09-03T06:44:29.689Z'
```

可以看到我们之前是 2022 14:44:29 现在对到 UTC 时间 0 时区，成 06:44:29 了。完成！

24.10 RFC 3339

更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

RFC 是 request for comment 的简写。它其实是一系列加了编号的文件。这一系列文件收集了有关互联网的文章，包括 UNIX 和某些互联网社区上的软件文件。RFC 还收录了很多与互联网标准相关的文章，包含各种网络协议，以及我们今天要谈及的时间格式标准问题。

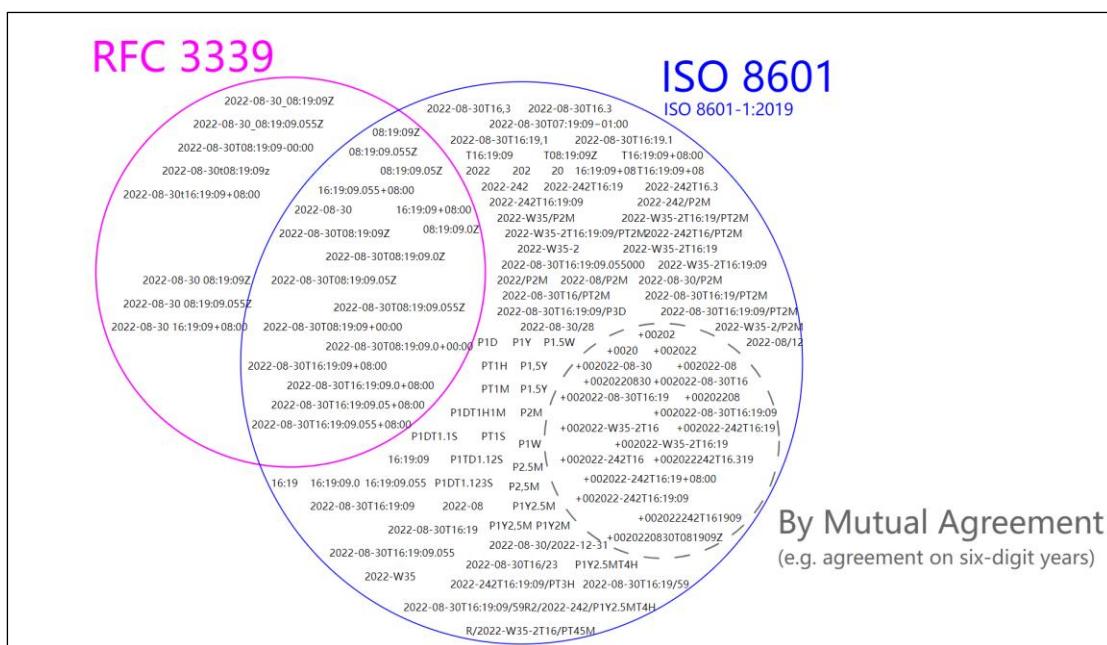
RFC3339 这篇文章的原文标题叫作，Date and Time on the Internet: Timestamps（互联网上的日期和时间：时间戳），发表的时间是 2002 年 7 月。感兴趣的同学可以参考原文：
<https://www.rfc-editor.org/rfc/rfc3339>

简单来说 RFC3339 对日期时间的定义更加简洁，去掉了 ISO 8601 中一些小众的表达方式，也更具有可读性。

24.11 RFC3339 和 ISO8601 之间的关系

可以参考：<https://ijmacd.github.io/rfc3339-iso8601/>

这个网站可以实时展示当前时间的 RFC3339 表示和 ISO 8601 表示。下面这张图是它的截图。可以看到 RFC 3339 的表述有一部分和 ISO 8601 是相同的。



仔细去看你会发现，RFC3339 格式的日期时间表述，基本上都能第一时间反应出来它表述的是什么时间。而 ISO 8601 中就会有像 2022-242T16:55:17/PT3H 这种看上去奇奇怪怪的表述。

24.12 补充：Unix 时间戳与闰秒

24.12.1 什么是 Unix 时间戳

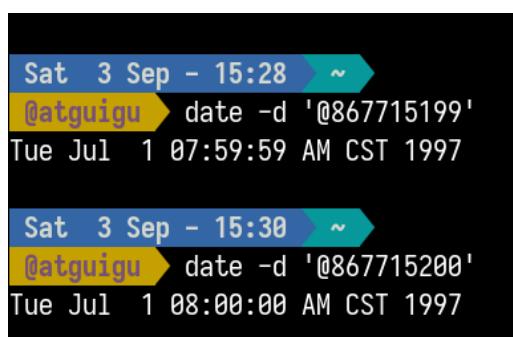
更多 Java – 大数据 – 前端 – python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

Unix 时间戳是一种将时间跟踪为运行总秒数的方法，这个技术从 1970 年 1 月 1 日的 UTC 开始。因此，Unix 时间戳只表示从特定时间点到现在的秒数。而且，需要注意的是，无论你身处何，这个总秒数的值在技术上都不会发生改变。所以这对计算机系统，客户端和服务端的通信和日期跟踪十分有用。

24.12.2 Unix 时间戳是怎么处理闰秒的

关于闰秒问题，我们之前说过，什么时候出现闰秒是不确定的。那么在 Unix 时间戳里，是怎么处理闰秒的呢？答案是减慢时钟。

比如 1997 年 6 月 30 日 23:59:59 到 1997 年 7 月 1 日 00:00:00 应该发生一次闰秒。



```
Sat 3 Sep - 15:28 ~
@atguigu date -d '@867715199'
Tue Jul 1 07:59:59 AM CST 1997

Sat 3 Sep - 15:30 ~
@atguigu date -d '@867715200'
Tue Jul 1 08:00:00 AM CST 1997
```

那么 867715200 这个时间戳应该对应 1997 年 6 月 30 日的 23:59:60。但是 Linux 好像压根不知道这件事。这是因为 Unix 时间戳标准里，把一天定死为 86400 秒了。所以类 Unix 的处理方案是，当闰秒发生时由 ntp 服务把时钟慢下来，当时间戳为 867715199 的时候，让它在这个值上多停留 1 秒然后再进入 867715200。