Portex

OpenBytes

Dec 01, 2022

CONTENTS

This documentation defines Portex. It is a table-structed data definition language designed for both structed-data and unstructed-data.

CHAPTER

ONE

DATA STORAGE

Portex is designed for a table-structed data storage system, which has the following features:

- Strongly typed: The type of the data in each column MUST be the same
- Support storing the binary file
- Support storing nested table

CHAPTER

TWO

PORTEX

Portex is a language for describing the data structure of the objects stored in the table. It defines the name and the type of each table column. It also tells the data user how to access the data in the table.

Portex is defined with JSON, this doc uses `yaml` to represent JSON objects for better legibility.

2.1 Basic Syntax

Here is the basic syntax of Portex:

```yaml
---
type: <type>
<type-param 1>: <value 1>
<type-param 2>: <value 2>
...: ...
...: ...
```

Portex provides the basic key `type`. Its value means the type of data and presented by a JSON string. The bulitin supported types can be found in *Primitive Types* and *Complex Types*.

The most important feature of Portex is that the type is configurable, different types has different parameters.

For example, the *enum* type has parameters `values` to indicate the possible values of the enum.

So an enum of `"dog"` and `"cat"` can be defined:

```yaml
---
type: enum
values: ["dog", "cat"]
```

Besides the builtin types, the customized types can also be configurable, check *Template Type* for more details.

2.2 Nullable Type

Portex provides a common parameter `nullable` for all types to indicate whether the value can be null.

name	type	required	default	description
`nullable`	JSON boolean	False	`False`	Default to `False`, which means all types are not nullable by default. Setting to `True` allows the stored value to be null.

Examples:

Nullable 32-bits signed integer:

```
---
type: int32
nullable: true
```

2.3 Primitive Types

Portex provides a set of primitive types:

2.3.1 boolean

The `boolean` type represents a binary value, only two values are supported: `true` and `false`

2.3.2 binary

The `binary` type represents a sequence of 8-bit unsigned binary.

2.3.3 string

The `string` type represents a sequence of UTF-8 encoded characters.

2.3.4 numeric types

There are four numeric types in Portex, they share the same parameters.

- `int32`: 32-bit signed integer.
- `int64`: 64-bit signed integer.
- `float32`: single precision (32-bit) IEEE 754 floating-point number.
- `float64`: double precision (64-bit) IEEE 754 floating-point number

Examples:

1. 32-bits signed integer:

```
---
type: int32
```

2. single precision floating-point number:

```
---
type: float32
```

2.4 Complex Types

2.4.1 enum

The `enum` type represents a value which is restricted in a fixed set of values.

The parameter `values` is provided for `enum` to indicate the set of values. It is a JSON array with at least one element, where each element is unique.

name	type	required	description
`values`	JSON array	True	Contains at least one element, and each element is unique.

Examples:

1. enum to represent colors

```
---
type: enum
values: [red, yellow, blue]
```

2. enum to represent animals

```
---
type: enum
values: [dog, cat, bird]
```

2.4.2 record

The `record` type is where a user can define complex data structures by grouping related variables together in the same place. It is similar to `struct` in C++ or the `Series` in pandas. It is preferred to use `record` to hold the grouped data in each row, and a column, in Portex, is a series of records of the same type.

The parameter `fields` is used in the `record` to define the member vairables. Each field should have a `name` and a `type`. The `fields` is defined in a one dimentional array manner, so it can easily be expanded into a multi-column row.

name	type	required	description
`fields`	JSON array	True	It is a one dimentional array. Each element in the array represents a member variable of the record. The member variables are ordered.
`fields.<index>`	JSON object	True	One element in the array, which represents a member variable of the record.
`fields.<index>.name`	JSON string	True	The name of the member variable.
`fields.<index>.type`	JSON string	True	The type of the member variable. It does not have to be a primitive type. It could be any type defined in the context.
`fields.<index>.<type-param>`	-	False	Type related parameters.

Examples:

1. a 2D point which uses `x` and `y` to represent its coordinates:

```yaml
---
type: record
fields:
  - name: x
    type: int32

  - name: y
    type: int32
```

In a tabular view:

x	y
<x coordinate>	<y coordinate>

2. a student record which contains the basic information of a student:

```yaml
---
type: record
fields:
  - name: name
    type: string
```

(continues on next page)

(continued from previous page)

```
  - name: gender
    type: enum
    values: [male, female, other]

  - name: age
    type: int32

  - name: student number
    type: string
```

In a tabular view:

name	gender	age	student number
<student name>	<student gender>	<student age>	<student number>

3. a 2D line which is represented by two 2D point coordinates:

```
---
type: record
fields:
  - name: point1
    type: record
    fields:
      - name: x
        type: int32

      - name: y
        type: int32

  - name: point2
    type: record
    fields:
      - name: x
        type: int32

      - name: y
        type: int32
```

In a tabular view:

point1		point2	
x	y	x	y
<x coordinate>	<y coordinate>	<x coordinate>	<y coordinate>

This example shows the record can be nested, it can be used to support the multi-indexing feature in a columnar store.

2.4.3 array

The `array` type represents a sequence of elements which have the same type.

Type `array` has two parameters `items` and `length`:

- `items` is used to indicate the type of the items in the array.
- `length` is used to indicate the length of the array.

name	type	required	default	description
`items`	JSON object	True	-	-
`items.type`	JSON string	True	-	Represent the type of the items in the array.
`items.<type-param>`	-	False	-	Represent the type parameter of the items in the array.
`length`	JSON integer	False	null	Represent the length of the array, used to define an array with a fixed length.

Examples:

1. an int32 array with unlimited length:

```
---
type: array
items:
  type: int32
```

2. an int32 array with fixed length:

```
---
type: array
items:
  type: int32
length: 2
```

3. a polygon represented by its vertex coordinates:

```
---
type: array
items:
  type: record
  fields:
    - name: x
      type: int32
```

(continues on next page)

(continued from previous page)

```
  - name: y
    type: int32
```

when the item type is `record`, the behavior of an `array` will change to a table:

Note: `array` + `record` = table

A `record` can be understood as a row in the table, then `array` put many rows together to get a table.

So the polygon array can be visually represented in table structure:

x	y
<x coordinate>	<y coordinate>
<x coordinate>	<y coordinate>
<x coordinate>	<y coordinate>
<x coordinate>	<y coordinate>
<x coordinate>	<y coordinate>
...	...

2.5 Temporal Types

Portex provides a set of temporal types:

2.5.1 date

The `date` type represents a date in a calendar without timezone or time of day.

The storage type of `date` is `int32`. It represents the days since UNIX epoch `1970-01-01`.

Examples:

A date object:

```
---
type: date
```

2.5.2 time

The `time` type represents a time of day, independent of any particular calendar, timezone or date.

The parameter `unit` is provided for `time` to indicate time resolution.

name	type	required	description
`unit`	JSON string	True	The time resolution, support `s`, `ms`, `us` and `ms`: - `s` for second - `ms` for millisecond - `us` for microsecond - `ns` for nanosecond

The `s` and `ms` time will be stored as `int32` and `us` and `ns` time will be stored as `int64`. And it represents an offset from `00:00:00` with the giving unit.

Examples:

A time with millisecond resolution:

```
---
type: time
unit: ms
```

2.5.3 timestamp

The `timestamp` type represents a time of day with date.

Type `timestamp` has two parameters `unit` and `tz`:

- `unit` is used to indicate time resolution.
- `tz` is used to indicate the timezone info.

name	type	required	description
`unit`	JSON string	True	The time resolution, support `s`, `ms`, `us` and `ms`: - `s` for second - `ms` for millisecond - `us` for microsecond - `ns` for nanosecond
`tz`	JSON string	False	The timezone info, default to naive timestamp. Supported timezone list: TZ_LIST

The storage type of `timestamp` is `int64`. It represents an offset from `1970-01-01T00:00:00` with the giving unit.

Examples:

1. A naive timestamp with millisecond resolution:

```
---
type: timestamp
unit: ms
```

2. A aware timestamp with microsecond resolution and timezone info is Asia/Shanghai:

```
---
type: timestamp
unit: us
tz: Asia/Shanghai
```

2.5.4 timedelta

The `timedelta` type represents a time duratiion, the difference between two dates or times.

The parameter `unit` is provided for `timedelta` to indicate time resolution.

name	type	required	description
`unit`	JSON string	True	The time resolution, support `s`, `ms`, `us` and `ms`: - `s` for second - `ms` for millisecond - `us` for microsecond - `ns` for nanosecond

The storage type of `timedelta` is `int64`. It represents an time offset with the giving unit.

Examples:

A timedelta with millisecond resolution:

```
---
type: timedelta
unit: ms
```

2.6 Type Import

Portex supports Type Import, which means the schema structure can be defined and shared in the community.

A **package** is used to distribute a group of pre-defined types. And these types can be imported from the package.

Tip: Just like a programming language, Portex also uses packages for distributing pre-defined types. Take python as an example. Python package is used to distribute a set of functions which can be reused.

The git repository is used as a carrier for a schema package. A schema package is distributed, developed, and imported through a public git repository.

OpenBytes defines a set of standard formats for open datasets. These formats are put on a Github repo and distributed as a schema package whose url is https://github.com/Project-OpenBytes/portex-standard.

2.6.1 How to build a schema package?

1. Create a remote git repo;
2. Commit a file named `ROOT.yaml` to indicate the root path of the schema;
3. Commit the schema structure files which need to be reused into the git repo.

2.6.2 How to import types from a package?

1. Use *Parameters* `imports` to indicate what types needs to be imported and which package these types come from;
2. Put the schema structure name or alias which needs to be referenced in the `type` field.

Parameters

The parameter `imports` is provided for type importing, and it should be put on the top level of the schema definition file.

name	type	required	description
`imports`	JSON array	False	A JSON object which indicates what types needs to be imported and which package these types come from.
`imports.<index>`	JSON object	True	Each item in the `imports` array indicates a group of imported types which come from a same package.
`imports.<index>.repo`	JSON string	True	The url and the revision of the schema package, which follows the following format: "<url>@<rev>".
`imports.<index>.types`	JSON array	True	A JSON array to indicate the types needs to be imported from the package to this file.
`imports.<index>.types.<index>`	JSON object	True	Each item in the `imports.<index>.types` array indicates one imported type.
`imports.<index>.types.<index>.name`	JSON string	True	The name of the imported type which follows the *Dot Syntax*
`imports.<index>.types.<index>.alias`	JSON string	False	The alias of the imported type. If this field is given, it will replace the `imports.types.<index>.name` as the unique identifier of the imported type. This field is useful for solving the type name conflicts in different packages.

Dot Syntax

The **doc syntax** is used for referencing pre-defined type.

Dot syntax is:

1. Based on the file path of the schema structure file;
2. Use dot `.` to replace the file separator (`/` for Linux and \ for Windows);
3. Remove the file extension.

For example, there is a schema repo with the following file structure:

```
.
├── geometry
│   ├── Vector2D.yaml
│   └── Vector3D.yaml
└── ROOT.yaml    # the ROOT.yaml file is used to indicate the root of the schema package.
```

The schema file `geometry/Vector2D.yaml` needs to be written as `geometry.Vector2D` for referencing.

Example

For example, two pre-defined types `Vector2D` and `Vector3D` need to be imported from a Github repo, whose url is https://github.com/Project-OpenBytes/portex-standard and the tag is `v1.0.0`.

The repo file structure is:

```
.
├── geometry
│   ├── Vector2D.yaml
│   └── Vector3D.yaml
└── ROOT.yaml    # the ROOT.yaml file is used to indicate the root of the schema package.
```

Here is how the `Vector2D` and `Vector3D` are imported:

```
---
imports:
  - repo: https://github.com/Project-OpenBytes/portex-standard@v1.0.0
                                          # Use "<url>@<rev>" format to # point out␣
↪where the
                                          # source code comes from.
    types:
      - name: geometry.Vector2D           # Use "dot syntax" to point out the type␣
↪defined in
                                          # "geometry/Vector2D.yaml" that needs to be␣
↪imported
                                          # to this file.
      - name: geometry.Vector3D
        alias: Vector3D                   # Use "alias" field to rename the imported␣
↪type.
                                          # "alias" will replace the origin name as␣
↪the unique
                                          # identifier. Which means "geometry.Vector3D
↪" will
```

(continues on next page)

(continued from previous page)

```
                                        # be treated as illegal name. Only "Vector3D
↪" can be
                                        # used for referencing the imported type.

type: record
fields:
  - name: point2d
    type: geometry.Vector2D       # Use the "name" defined in the "imports" field to␣
↪reuse
                                  # the pre-defined type.
  - name: point3d
    type: Vector3D                # Use the "alias" defined in the "imports" field to␣
↪reuse
                                  # the pre-defined type.
```

2.7 Template Type

One of the most important features in Portex is configurable type, different types provide different parameters to adjust their behaviors.

Such as `enum` type provide `values`, `record` type provides `fields` etc.

2.7.1 Parameters

Portex provides `template` type to define customized configurable types.

Two parameters are provided in `template` type:

- `parameters` is used to indicate the parameters.
- `declaration` is used to indicate how the parameters take effect.

name	type	required	description
`parameters`	JSON array	False	Indicate all the parameters for this template.
`parameters.<index>`	JSON object	True	Each element in `parameters` defines a parameter.
`parameters.<index>.name`	JSON string	True	The name of the parameter.
`parameters.<index>.default`	-	False	The default value of the parameter. The parameter is optional if the default value is set. The parameter is required if the default value is not set.
`parameters.<index>.options`	JSON array	False	An array to list all possible values. Parameter value not listed in the array will not be accepted.
`declaration`	JSON object	True	The declaration of template, use `$<name>` to indicate how different parameters take effect in the template.
`declaration.type`	JSON string	True	The type of the template.
`declaration.<type-param>`	-	True	The parameters of the actual type.

Examples:

1. A 2D point type:

```yaml
# geometry/Point.yaml
---
type: template
declaration:
  type: record
  fields:
    - name: x
      type: int32

    - name: y
      type: int32
```

after definition, this `Point` type can be referenced:

```yaml
---
type: record
fields:
  - name: point1
    type: geometry.Point

  - name: point2
    type: geometry.Point
```

it can be visually represented in table structure:

point1		point2	
x	y	x	y
<int32 value>	<int32 value>	<int32 value>	<int32 value>

2. A 2D point type with configurable label:

```yaml
# geometry/LabeledPoint.yaml
---
type: template
parameters:
  - name: labels                # "labels" is a required parameter

declaration:
  type: record
  fields:
    - name: x
      type: int32

    - name: y
      type: int32

    - name: label
      type: enum
      values: $labels             # the values of enums depend on the input "labels"
```

after definition, this `LabeledPoint` type can be referenced:

```yaml
---
type: record
fields:
  - name: labeled_point
    type: geometry.LabeledPoint
    values: ["visble", "occluded"]
```

it can be visually represented in table structure:

labeled_point		
x	y	label
<int32 value>	<int32 value>	<"visble" or "occluded">

Error: Setting the type name as a parameter, as shown in the following example, is not allowed in Portex.

```yaml
# geometry/Point.yaml
---
type: template
parameters:
  - name: coords
    default: int32          # $coords represent the name of the type

declaration:
  type: record
  fields:
    - name: x
      type: $coords         # The type name should be put after keyword "type:"
                            # set the type name as parameter is not allowed in Portex

    - name: y
      type: $coords
```

Note: Check the *object unpack* syntax for creating a template type with configurable internal types.

2.7.2 Parameter "exist_if"

Portex provides a special parameter `exist_if` to control whether a field in `record` exists.

When `declaration.type` is `record`, the parameter `declaration.fields.<index>.exist_if` can be used to control whether the field exists.

name	required	default	description
`declaration.fields.<index>.exist_if`	False	True	The field exists if the value of `exist_if` is not `null`, otherwise it does not.

Examples:

a Point type with or without a enum label:

```yaml
# geometry/Point.yaml
---
type: template
parameters:
  - name: labels
    default: null

declaration:
  type: record
  fields:
    - name: x
      type: int32

    - name: y
      type: int32

    - name: label
      exist_if: $labels              # When "labels" is not "null", the "label
↪" field exists,
      type: enum
      values: $labels
```

after definition, this `Point` type can be referenced with a parameter `labels`:

```yaml
---
type: record
fields:
  - name: point
    type: geometry.Point

  - name: labeled_point
    type: geometry.Point
    labels: ["visble", "occluded"]
```

it can be visually represented in table structure:

point		labeled_point		
x	y	x	y	label
<int32 value>	<int32 value>	<int32 value>	<int32 value>	<"visble" or "occluded">

2.7.3 Unpack Syntax

Portex provides unpack syntax for JSON object and JSON array in template type.

Object unpack

Portex use + symbol for object unpack, it is used to unpack the JSON object parameter and merge it into another JSON object.

This syntax is used to create the template type whose internal type is configurable. Just like the builtin *array* type, the type of the array elements can be configured by its `items` parameter

---

Note: Portex object unpack is similar with YAML merge key.

Examples:

1. A 2D point type with configurable coordinate type:

```
# geometry/Point.yaml
---
type: template
parameters:
  - name: coords
    default:                    # "coords" is not a required parameter
      type: int32               # the default value of "coords" is '{"type": "int32"}
↪'

declaration:
  type: record
  fields:
    - name: x
      +: $coords               # use object unpack symbol "+" to unpack $coords
                               # which makes the coordinate type configurable
                               # $coords should be a JSON object

    - name: y
      +: $coords
```

after definition, this `Point` type can be referenced with a parameter `coords`:

```
---
type: record
fields:
  - name: point1
    type: geometry.Point
    coords:
      type: float32         # set the coordinate type to "float32"

  - name: point2
    type: geometry.Point    # use the default type "int32"
```

it can be visually represented in table structure:

point1		point2	
x	y	x	y
<float32 value>	<float32 value>	<int32 value>	<int32 value>

Array unpack

Portex also use + symbol for array unpack. The syntax +$<name> is used to unpack the JSON array parameter and merge it into another JSON array.

This syntax can be used to extend the record fields.

Examples:

1. A 2D point type with extensible fields:

```yaml
# geometry/Point.yaml
---
type: template
parameters:
  - name: extra
    default: []        # the default value is an empty array, which means add no␣
↪fields

declaration:
  type: record
  fields:
    - name: x
      type: int32

    - name: y
      type: int32

    - +$extra          # use "+$<name>" syntax to unpack the parameter "extra"
                       # which makes the record fields extensible
                       # $extra should be a JSON array
```

after definition, this `Point` type can be referenced with a parameter `extra`:

```yaml
---
type: record
fields:
  - name: point1
    type: geometry.Point
    extra:
      - name: label         # set "label" as a extra field
        type: enum
        values: ["visble", "occluded"]

  - name: point2
    type: geometry.Point    # the default behavior is no extra field
```

it can be visually represented in table structure:

point1			point2	
x	y	label	x	y
<int32 value>	<int32 value>	<”visble” or “occluded”>	<int32 value>	<int32 value>