

Qiita



ログイン

新規登録

トレンド

質問

公式イベント

公式コラム

募集

Organization

幅広い業界



この記事は最終更新日から1年以上が経過しています。



@minoritea

TOML仕様の和訳

TOML

最終更新日 2020年12月03日 投稿日 2015年01月04日

TOML v1.0.0-rc.2 の和訳です。

原文はこちら。

<https://github.com/toml-lang/toml.io/blob/main/specs/en/v1.0.0-rc.2.md>

また、この翻訳の本体はこちらです。

<https://github.com/toml-lang/toml.io/blob/main/specs/ja/v1.0.0-rc.2.md>

- v1.0.0-rc.2の翻訳者はToruNiinaさんです。



TOML v1.0.0-rc.2



271



211



By Tom Preston Werner, Pradyn Gedam, et al.

目的

TOMLは明瞭なセマンティクスを持ち、可読性の高い、ミニマルな設定ファイルフォーマットとなることを目的として作られています。

TOMLは曖昧さなしに連想配列に変換できるよう設計されていて、様々な言語上でそれらのデータ構造に展開することができます。

目次

- 仕様
- コメント
- キーと値の組
- キー
- 文字列
- 整数
- 浮動小数点数
- ブーリアン
- オフセット付き日時
- ローカルの日時
- ローカルの日付
- ローカルの時刻
- 配列
- テーブル
- インライン・テーブル
- テーブルの配列
- ファイル名（拡張子）
- MIME Type



- TOMLはケース・センシティブです。大文字と小文字は区別されます。
- TOMLファイルはユニコード(UTF-8)でエンコードされている必要があります
- 空白はタブ(0x09)もしくはスペース(0x20)のことです
- 改行はLF(0x0A)もしくはCRLF(0x0D 0x0A)です。

コメント

ハッシュ記号（`#`）に続けて改行までをコメントとします。

ただし、文字列の内部はコメントになりません。

```
# この行は全てコメントです。  
key = "value" # 行末までコメントです。  
another = "# This is not a comment"
```

タブ以外の制御文字（U+0000からU+0008, U+000A から U+001F, それと U+007F）はコメント内では使用できません。

キーと値の組

TOML文書の最も基本的な構造はキーと値の組です。

キーは等号（`=`）の左に、値は右に記述します。

キーと値の周りにある空白は無視されます。

キー、等号、値は同じ行に入れる必要があります（いくつかのタイプの値は複数行で記述される場合があります）。

```
key = "value"
```

値は以下のいずれかのデータ型でなければなりません。

- 文字列



271

211

- ブーリアン
- オフセット付き日時
- ローカルの日時
- ローカルの日付
- ローカルの時刻
- 配列
- インライン・テーブル

未定義の値は不正となります。

```
key = # 不正
```

キーと値の組のあとには、必ず改行かファイルの終端（EOF）が来なければなりません（ただし、**インライン・テーブル**の場合は例外です）。

```
first = "Tom" last = "Preston-Werner" # INVALID
```

キー

キーはベア・キー、クォート付きキー、ドット付きキーのいずれかです。

ベア・キーはASCII英数字とアンダースコア、ダッシュのみで構成されます（`A-Za-z0-9_-`）。

ASCII数字のみでも構いませんが、文字列として解釈されることに注意してください（例: `1234`）。

```
key = "value"
bare_key = "value"
bare-key = "value"
1234 = "value"
      ^^^^^^^^^^^^^
```

クォート付きキーは後述の基本文字列もしくはリテラル文字列と同じルールに従いま



271

211

これによってより広い文字セットをキーに使うことができます。

ベスト・プラクティスとして、絶対に必要なケース以外はベア・キーを使うことをおすすめします。

```
"127.0.0.1" = "value"
"character encoding" = "value"
"𐀀𐀁𐀂" = "value"
'key2' = "value"
'quoted "value"' = "value"
```

ベア・キーでは空白は許容されませんが、クオート付きキーでは空文字列をキーに使うことができます（非推奨ではありますが）。

```
= "no key name" # 不正です
"" = "blank" # 可能ですがお奨めしません
'' = 'blank' # 可能ですがお奨めしません
```

ドット付きキーはドット（`.`）で接続されたベア・キーもしくはクオート付きキーの連なりです。

近い属性同士をまとめるために用いることができます。

```
name = "Orange"
physical.color = "orange"
physical.shape = "round"
site."google.com" = true
```

JSONで記述すると、上記のデータは以下のような構造になります。

```
{
  "name": "Orange",
  "physical": {
    "color": "orange",
    "shape": "round"
  },
  "site": {
    "google.com": true
  }
}
```



271

211

```
}  
}
```

ドットで分けられたそれぞれの部分の周りに空白をいれても無視されます。しかしながら、ベスト・プラクティスは空白をいれないことです。

同じキーを複数回定義することはできません。

```
# やらないでくださいね  
name = "Tom"  
name = "Pradyun"
```

このとき、ベア・キーとクオーテッド・キーは等価であることに気をつけてください。

```
# これらのキーは同一なので、不正です  
spelling = "favorite"  
"spelling" = "favourite"
```

キーは直接値が定義されていない限り、そのキー以下に内容を追加することが出来ます。

```
# これによって"fruit"に対応する値はテーブルになります。  
fruit.apple.smooth = true  
  
# なので、このように"fruit"テーブルに追加できます。  
fruit.orange = 2
```

```
# 以下は間違った例です。  
  
# ここで、fruit.appleに対応する値は整数になっています。  
fruit.apple = 1  
  
# しかしここでは、fruit.appleをテーブルとして使おうとしています。  
# 整数をテーブルにすることはできません。  
fruit.apple.smooth = true
```



ドット付きキーを整列させずに使うことは非推奨です。

可能ですが、非推奨です

```
apple.type = "fruit"
orange.type = "fruit"

apple.skin = "thin"
orange.skin = "thick"

apple.color = "red"
orange.color = "orange"
```

こちらの方が推奨されます

```
apple.type = "fruit"
apple.skin = "thin"
apple.color = "red"

orange.type = "fruit"
orange.skin = "thick"
orange.color = "orange"
```

ベア・キーはASCII整数のみで構成されていてもよいので、小数点数のように見えるドット付きキーを書くことも可能です。

ただし、それをするに足る理由がないのなら（多分ないと思います）、避けたほうがよいでしょう。

```
3.14159 = "pi"
      ^^
```

JSONで記述すると、上記のデータは以下のような構造になります。

```
{ "3": { "14159": "pi" } }
```



271

211

文字列を記述するには、4種類の方法があります。基本文字列、複数行基本文字列、リテラル文字列、複数行リテラル文字列です。

どれもUTF-8として有効なユニコード文字のみで構成されている必要があります。

基本文字列はクォーテーションマーク(`"`)で囲まれた文字列です。

クォーテーションマーク、バックスラッシュ、タブ以外の制御文字 (U+0000からU+0008 と U+000AからU+001F、U+007F) はエスケープする必要があります。その他のユニコード文字は全て文字列内で使えます。

```
str = "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation\tSF."
```

利便性のために、いくつかの文字についてはエスケープシーケンスの短縮形が用意されています。

<code>\b</code>	- backspace	(U+0008)
<code>\t</code>	- tab	(U+0009)
<code>\n</code>	- linefeed	(U+000A)
<code>\f</code>	- form feed	(U+000C)
<code>\r</code>	- carriage return	(U+000D)
<code>\"</code>	- quote	(U+0022)
<code>\\</code>	- backslash	(U+005C)
<code>\uXXXX</code>	- unicode	(U+XXXX)
<code>\UXXXXXXXX</code>	- unicode	(U+XXXXXXXX)

全てのユニコード文字は `\uXXXX` もしくは `\UXXXXXXXX` の形式にエスケープできます。エスケープコードは正しいユニコード・スカラー値である必要があります。

上記以外のエスケープシーケンスは将来のために予約されています。もし使ってしまった場合はTOMLはエラーを出すでしょう。

ときにはあなたは文書の一節を書いたり (例: 翻訳文書)、とても長い文を改行したくなることがあるでしょう。TOMLはそのようなときの手間を省くことができます。

複数行基本文字列はクォーテーションマーク3つずつで囲まれた文字列で、その中で改行ができます。

最初の3つのクォーテーションマークの直後に来た改行は取り除かれます。その他の空




```
str1 = """
Roses are red
Violets are blue"""
```

TOMLのパーサは改行をそのプラットフォームに応じて自由に正規化できます。

```
# Unixでは上記のTOMLファイルは以下と同じパース結果になります:
str2 = "Roses are red\nViolets are blue"

# Windowsでは上記のTOMLファイルは以下と同じパース結果になります:
str3 = "Roses are red\r\nViolets are blue"
```

長い文字列を不要な行頭の空白なしに書きたい場合は、`\`を行末に書きます。
行末に`\`があった場合、全ての空白（もしくは改行）は、空白でない文字が現れるか、文字列が終わるまで取り除かれます。

このとき`\`も同時に取り除かれます。

全てのエスケープシーケンスは基本文字列と同様に複数行文字列でも使えます。

```
# 以下のそれぞれの文字列は同じとなります:
str1 = "The quick brown fox jumps over the lazy dog."

str2 = """
The quick brown \

fox jumps over \
the lazy dog."""

str3 = """\
The quick brown \
fox jumps over \
the lazy dog.\
"""
```

基本的に全てのユニコード文字が使えますが、タブ、改行、キャレッジリターン以外の制御文字(U+0000からU+0008、U+000B、U+000C、U+000EからU+001F、U+007F)とバックスラッシュはエスケープされる必要があります。



1つ、あるいは2つの連続したクォーテーションマークは複数行基本文字列のなかで自由に使うことができます。文字列の最初と最後にも書くことができます。

```
str4 = """Here are two quotation marks: """. Simple enough."""
# str5 = """Here are three quotation marks: """"."" # 不正です。
str5 = """Here are three quotation marks: ""\". """
str6 = """Here are fifteen quotation marks: ""\"""\"""\"""\"""\". """

# 内容は "This," she said, "is just a pointless statement." です。
str7 = """\"This,\" she said, \"is just a pointless statement.\"\"\"
```

あなたがWindowsパスや正規表現を書くことが多い場合、バックスラッシュをエスケープしなければならないと、面倒で間違いやすくなるでしょう。そのような場合のため、TOMLはエスケープなしのリテラル形式の文字列をサポートしています。

リテラル文字列はシングル・クォート(')で囲まれた文字列です。基本文字列のように一行に書きます:

```
# そのままの文字列を得ることができます。
winpath = 'C:\Users\nodejs\templates'
winpath2 = '\\ServerX\admin$\system32\'
quoted = 'Tom "Dubs" Preston-Werner'
regex = '<i>\c*\s*>'
```

エスケープが無いため、シングル・クォートはリテラル文字列中では書けません。そのような場合のため、TOMLはリテラル文字列の複数行版をサポートしています。

複数行リテラル文字列はシングル・クォート3つずつで囲まれた文字列で、その中で改行ができます。

リテラル文字列と同様にエスケープはありません。文字列の頭の改行は取り除かれます。他の文字列の中身は全て変更なしに読み込まれます。

```
regex2 = '''I [dw]on't need \d{2} apples'''
lines = '''
文字列開始直後の改行は
```



保持されます。

...

1つか2つのシングル・クォートは複数行リテラル文字列の中で使うことができます。ですが、3つ以上の連続するシングル・クォートは使うことができません。

```
quot15 = '''Here are fifteen quotation marks: '''

# apos15 = '''Here are fifteen apostrophes: ''' # 不正
apos15 = "Here are fifteen apostrophes: '"

# 内容は 'That,' she said, 'is still pointless.' となります。
str = '''That,' she said, 'is still pointless.''''
```

タブ以外の制御文字をリテラル文字列の中で使うことはできません。

よって、バイナリデータを扱う際には、Base64か、他の適切なASCIIもしくはUTF-8のエンコーディング形式でエンコードすることを推奨します。そして、それらのエンコーディングは（TOMLを読み込む）アプリケーション側で取り扱う必要があります。

整数

整数は、整数全体を表します。

正の数を表すときはプラス符号 `+` を前につけても、つけなくても構いません。負の数の場合はマイナス符号 `-` を前につけます。

```
int1 = +99
int2 = 42
int3 = 0
int4 = -17
```

大きな数字を読みやすく表記するために、アンダースコアを使うことができます。それぞれのアンダースコアは最低1つの数字で挟まれている必要があります。

```
int5 = 1_000
```



271

211

```
int7 = 53_49_221 # インド式命数法
int8 = 1_2_3_4_5 # 可能ですがお奨めしません
```

(10進数のゼロ以外の) 整数の表記をゼロから始めることはできません。

`-0` と `+0` は許容され、プレフィックスなしのゼロと同じ値となります。

非負整数は2進数、8進数、16進数の形式で表すこともできます。

これらの表記では、`+` をつけることはできませんが、ゼロを頭にして続けることができます。

16進数を表す場合大文字小文字は問いません。

アンダースコアは数字の間に用いることはできますが、プレフィックスと数字の間に入れることはできません。

```
# 16進数表記はプレフィックス`0x`をつけます
hex1 = 0xDEADBEEF
hex2 = 0xdeadbeef
hex3 = 0xdead_beef

# 8進数表記はプレフィックス`0o`をつけます
oct1 = 0o01234567
oct2 = 0o755 # Unixのファイルパーミッションを表すのに便利です

# 2進数表記はプレフィックス`0b`をつけます
bin1 = 0b11010110
```

任意の符号付き64ビット整数 (−9,223,372,036,854,775,808 ~

9,223,372,036,854,775,807) は許容され、可逆的な形で読み込まれる必要があります。

もし可逆的な形で読み込めない整数が現れた場合、パーサはエラーを出力する必要があります。

浮動小数点数

浮動小数点数はIEEE754 64ビット倍精度で実装されます。

浮動小数点数は整数部 (整数型と同じルールで記述) と、それに続く小数部もしくは



271

211

小数部と指数部の両方を用いることもできますが、その場合は小数部を指数部より前に置く必要があります。

```
# 小数表記
flt1 = +1.0
flt2 = 3.1415
flt3 = -0.01

# 指数表記
flt4 = 5e+22
flt5 = 1e06
flt6 = -2E-2

# 複合
flt7 = 6.626e-34
```

小数部は小数点の後に数字をならべて表記します。

指数部は **E**（大文字小文字は問わない）の後に整数（整数型と同じルールで記述）で表記します。

小数点を使う場合、少なくとも1つの数字に挟まれていなければなりません。

```
# 以下は不正です
invalid_float_1 = .7
invalid_float_2 = 7.
invalid_float_3 = 3.e+20
```

整数と同様、可読性のためにアンダースコアを使うこともできます。

それぞれのアンダースコアは最低1つの数字に挟まれていなければなりません。

```
flt8 = 9_224_617.445_991_228_313
```

`-0.0` と `+0.0` は許容され、IEEE754に従って実値に展開されます。

以下の特別な浮動小数点数も記述することができます。



```
# 無限大
sf1 = inf # 正の無限大
sf2 = +inf # 正の無限大
sf3 = -inf # 負の無限大

# 非数
sf4 = nan # sNaNかqNaNかは実装依存です
sf5 = +nan # `nan`と同じ
sf6 = -nan # 許容されますが、展開は実装依存です
```

ブーリアン

ブーリアン値はただのトークンです（皆がいつも使っているやつです）。小文字のみとします。

```
bool1 = true
bool2 = false
```

オフセット付き日時

あいまいさなしに特定の瞬間を表現するためにRFC 3339表記のオフセット付き日時を使います。

```
odt1 = 1979-05-27T07:32:00Z
odt2 = 1979-05-27T00:32:00-07:00
odt3 = 1979-05-27T00:32:00.999999-07:00
```

可読性のために、`T`の代わりに日付と時刻の間に空白を置くこともできます（RFC 3339 section 5.6 で許可されています）。

```
odt4 = 1979-05-27 07:32:00Z
```



271

211

小数秒部分の精度は実装依存ですが、最低ミリ秒以上精度が要求されます。もし実装がサポートしているより高い精度の値が与えられた場合、実装を超える精度については丸めではなく切り捨てられる必要があります。

ローカルの日時

RFC 3339のオフセットを省略した場合、その値はオフセットやタイムゾーンなしの日時として扱われます。そしてその値は追加の情報なしには特定の瞬間を表すデータに変換されることはありません。仮にそのようなデータに変換する必要がある場合、その変換は実装依存となります。

```
ldt1 = 1979-05-27T07:32:00
ldt2 = 1979-05-27T00:32:00.999999
```

小数秒部分の精度は実装依存ですが、最低ミリ秒以上精度が要求されます。もし実装がサポートしているより高い精度の値が与えられた場合、実装を超える精度については丸めではなく切り捨てられる必要があります。

ローカルの日付

RFC 3339日時の日付部分のみを記述した場合、それはオフセットやタイムゾーンなしの、その日付そのものを表すデータとなります。

```
ld1 = 1979-05-27
```

ローカルの時刻

RFC 3339日時の時刻部分のみを記述した場合、それはオフセットやタイムゾーンなしの、その時刻そのものを表すデータとなります。



271

211

```
lt1 = 07:32:00
lt2 = 00:32:00.999999
```

小数秒部分の精度は実装依存ですが、最低ミリセカンド以上の精度が要求されます。もし実装がサポートしているより高い精度の値が与えられた場合、実装を超える精度については丸めではなく切り捨てられる必要があります。

配列

配列は角括弧で囲まれた値の集まりです。空白は無視されます。各要素はカンマで区切られます。

キーと値の組で使うことが許されている全ての型を、配列でも使うことができます。また、異なる型の値を混ぜて使うことも可能です。

```
integers = [ 1, 2, 3 ]
colors = [ "red", "yellow", "green" ]
nested_array_of_int = [ [ 1, 2 ], [3, 4, 5] ]
nested_mixed_array = [ [ 1, 2 ], ["a", "b", "c"] ]
string_array = [ "all", 'strings', ""are the same"", ''type'' ]

# 異なる型を混ぜて使うこともできます。
numbers = [ 0.1, 0.2, 0.5, 1, 2, 5 ]
contributors = [
    "Foo Bar <foo@example.com>",
    { name = "Baz Qux", email = "bazqux@example.com", url = "https://example.com/bazqux" },
]
```

配列は複数行に書くこともできます。最後の要素の後ろにカンマを置くこともできます。

改行やコメントは値や閉じ括弧の前に好きなだけ置くことができます。

```
integers2 = [
    1, 2, 3
]
```



271

211


```
1,  
2, # this is ok  
]
```

テーブル

テーブル（ハッシュテーブルや連想配列のことです）はキーと値の組からなる集まりです。

テーブルは角括弧で囲まれたテーブル名が書かれた行から始まります。

配列では角括弧の中には値しか入らないので、配列とテーブルの宣言は区別することができます。

```
[table]
```

この行のあと、次のテーブルが始まるか、ファイルが終わるまでに出てきたキーと値はこのテーブルに属します。

テーブル内のキーと値の各ペアの順番は保証されません。

```
[table-1]  
key1 = "some string"  
key2 = 123  
  
[table-2]  
key1 = "another string"  
key2 = 456
```

テーブルの名前付けのルールはキーと同じとなります（[キーの定義](#)を参照してください）。

```
[dog."tater.man"]  
type.name = "pug"
```

これは以下のJSONと同じ構造を表します:



271

211

```
{ "dog": { "tater.man": { "type": { "name": "pug" } } } }
```

テーブル名の周りの空白は無視されます。しかし余分な空白は使わないことお奨めします。

```
[a.b.c]          # ベスト・プラクティスです
[ d.e.f ]        # [d.e.f]と同じ
[ g . h . i ]    # [g.h.i]と同じ
[ j . "λ" . 'l' ] # [j."λ". 'l']と同じ
```

もし、上位のテーブルそのものを記述する必要がないのであれば、省略することもできます。

```
# [x] 省略可
# [x.y] これも可
# [x.y.z] これも省略可
[x.y.z.w] # ここから始めて構いません

[x] # そのあとで上位テーブルを定義することもできます。
```

空のテーブルは単純にキーと値のペアを書かかないことで作れます。

キーと同様にテーブルも同じ名前で複数回定義することはできません。
もしそうした場合不正となります。

```
# やらないでくださいね
```

```
[a]
b = 1

[a]
c = 2
```

```
# やっちゃダメですってば
```



271

211

```
[a.b]
c = 2
```

テーブルの定義は整列させることが推奨されます。

```
# 正しいですが、非推奨です
[fruit.apple]
[animal]
[fruit.orange]
```

```
# こちらが推奨されます
[fruit.apple]
[fruit.orange]
[animal]
```

ドット付きキーを用いた場合、その左にある全てのキーはテーブルになります。テーブルは複数回定義することができないので、ドット付きキーで作られたテーブルを `[table]` の形で再度定義することはできません。同様に、ドット付きキーを用いて `[table]` の形で定義されたテーブルを再度定義することもできません。

しかし、`[table]` の形でドット付きキーで作られたテーブルの下にテーブルを追加することはできます。

```
[fruit]
apple.color = "red"
apple.taste.sweet = true

# [fruit.apple]           # できません
# [fruit.apple.taste]     # できません

[fruit.apple.texture] # サブテーブルを追加することはできます
smooth = true
```



インライン・テーブル

インライン・テーブルはテーブルを表現するためのよりコンパクトな構文です。他の表現だと冗長になるようなグループになったデータを表すのに向いています。インライン・テーブルは波括弧（`{`と`}`）で囲まれている必要があります。波括弧の中では、キーと値のペアをカンマ区切りでゼロ個以上置くことができます。値にはインライン・テーブル自体を含む全ての型を使うことができます。

インライン・テーブルは一行で表現されるべきです。インライン・テーブルでは、配列と違って、最後の要素のあとにコンマを足すことはできません。波括弧の内側では改行は認められません。ただし、それぞれの値の中で改行することは、それぞれの値の型で認められている範囲でできます。ですが、できるとしてもインライン・テーブルを複数行に分割することはなるべく避けるべきです。もしあなたがその必要があるように感じたのならば、普通のテーブルを用いるべきです。

```
name = { first = "Tom", last = "Preston-Werner" }
point = { x = 1, y = 2 }
animal = { type.name = "pug" }
```

上記のインライン・テーブルは下記の普通のテーブルと同一の定義です。

```
[name]
first = "Tom"
last = "Preston-Werner"

[point]
x = 1
y = 2

[animal]
type.name = "pug"
```

インライン・テーブルの中のキーやサブテーブルはそれ自体で完結しています。新しいキーやサブテーブルを追加することはできません。



```
# type.edible = false # 不正です。
```

同様に、インライン・テーブルを使って既に定義されたテーブルにキーやサブテーブルを追加することもできません。

```
[product]
type.name = "Nail"
# type = { edible = false } # 不正です。
```

テーブルの配列

最後に説明する型はテーブルの配列です。テーブル名を角括弧で二重に囲むことで表されます。

テーブル名のあと、次のテーブルが始まるか、ファイルが終わるまでに出てきたキーと値はそのテーブルに属します。

二重角括弧で囲まれた同じ名前を持つテーブルは、一つの配列の要素となります。テーブルは表記順に配列に挿入されます。

配列内のキーと値のペアを持たないテーブルは空のテーブルとして扱われます。

```
[[products]]
name = "Hammer"
sku = 738594937

[[products]]

[[products]]
name = "Nail"
sku = 284758393

color = "gray"
```

これは以下のJSONと同じ構造を表します:

```
{
```



271

211

```
{ },  
{ "name": "Nail", "sku": 284758393, "color": "gray" }  
]  
}
```

ネストしたテーブルの配列を作ることができます。その場合は、サブテーブルにも二重角括弧表記を使ってください。

それぞれのテーブルは、その上の最も近い場所に定義されている上位テーブルの要素となる配列に属します。

```
[[fruit]]  
  name = "apple"  
  
  [fruit.physical] # テーブル  
    color = "red"  
    shape = "round"  
  
  [[fruit.variety]] # ネストされたテーブルの配列  
    name = "red delicious"  
  
  [[fruit.variety]]  
    name = "granny smith"  
  
[[fruit]]  
  name = "banana"  
  
  [[fruit.variety]]  
    name = "plantain"
```

上記のTOMLは下記のJSONに置き換えることができます。

```
{  
  "fruit": [  
    {  
      "name": "apple",  
      "physical": {  
        "color": "red",  
        "shape": "round"  
      },  
    },  
  ],  
}
```



```

    { "name": "granny smith" }
  ]
},
{
  "name": "banana",
  "variety": [
    { "name": "plantain" }
  ]
}
]
}

```

(テーブルの配列を含む) テーブルが属する上位テーブルがテーブルの配列である場合、上位テーブルはサブテーブルよりも先に定義されなければなりません。サブテーブルが先に定義された場合、パース時にエラーになります。

```

# このTOMLファイルは不正です
[fruit.physical] # ここで、`fruit`は通常のテーブルであると推論されます
  color = "red"
  shape = "round"

[[fruit]] # ここで`fruit`がテーブルの配列であることが判明し、パーサーはエラーを出力しま
  name = "apple"

```

すでに値として定義された配列に対して、テーブルの配列を追加しようとする行為はパース時にエラーとなります。仮にその配列が空であってもです。

```

# このTOMLファイルは不正です
fruit = []

[[fruit]] # ここでエラーになります

```

既にテーブルの配列として定義されたテーブルの後に、同じ名前を持つ通常のテーブルを定義しようとした場合、パース時にエラーとなります。同様に、通常のテーブルとして定義されたテーブルのあとに、同じ名前を持つテーブルの配列を定義した場合も、エラーとなります。



271

211

```
# 不正なTOMLドキュメント

[[fruit]]
  name = "apple"

  [[fruit.variety]]
    name = "red delicious"

# 不正: このテーブルは先に定義された [[fruits.variety]] と矛盾する
[fruit.variety]
  name = "granny smith"

[fruit.physical]
  color = "red"
  shape = "round"

# 不正: このテーブルは先に定義された [fruits.physical] と矛盾する
[[fruit.physical]]
  color = "green"
```

テーブルの配列を作るために、インライン・テーブルを使うこともできます。

```
points = [ { x = 1, y = 2, z = 3 },
            { x = 7, y = 8, z = 9 },
            { x = 2, y = 4, z = 8 } ]
```

ファイル名（拡張子）

TOMLのファイルには `.toml` という拡張子を付ける必要があります。

MIME Type

インターネット経由でTOMLファイルを転送する場合、ふさわしいMIMEタイプは `application/toml` です。



271

211

新規登録して、もっと便利にQiitaを使ってみよう

1. あなたにマッチした記事をお届けします
2. 便利な情報をあとで効率的に読み返せます

[ログインすると使える機能について](#)

新規登録

ログイン



271

211



HTTP2のヘッダ圧縮技術

by 0xffffff7



PostgreSQLのあまり知られていない型3種

by choplin



README データベース設計

by yoshinoritera55



文字列のルックアップに mysql の ENUM 型を使う

by ArimaRyunosuke



Wranglerに選べるオファー実施中

PR Jeep Japan



在庫管理にはバーコード・QRコードが便利！活用事例をご紹介

PR ブラザー販売

コメント

この記事にコメントはありません。

あなたもコメントしてみませんか :)

新規登録

すでにアカウントを持っている方は[ログイン](#)



271

211

How developers code is here.

© 2011-2023 Qiita Inc.





ガイドとヘルプ

- About
- 利用規約
- プライバシーポリシー
- ガイドライン
- デザインガイドライン
- ご意見
- ヘルプ
- 広告掲載

コンテンツ

- リリースノート
- 公式イベント
- 公式コラム
- 募集
- アドベントカレンダー
- Qiita 表彰プログラム
- API

SNS

-  Qiita（キータ）公式
-  Qiita マイルストーン
-  Qiita 人気の投稿
-  Qiita（キータ）公式

Qiita 関連サービス

- Qiita Team
- Qiita Jobs
- Qiita Zine
- Qiita 公式ショップ

運営

- 運営会社
- 採用情報
- Qiita Blog



271

211