REST ユーザ、NetOps、DevOps のための JSON 解析レシピ

豊沢 聡◉著

CUTT
カットシステム

■**サンプルファイルのダウンロードについて**

本書掲載のサンプルファイルは、一部を除いてインターネット上のダウンロードサービスからダウンロードすることができます。詳しい手順については、本書の巻末にある袋とじの内容をご覧ください。

なお、ダウンロードサービスのご利用にはユーザー登録と袋とじ内に記されている番号が必要です。そのため、本書を中古書店から購入されたり、他者から貸与、譲渡された場合にはサービスをご利用いただけないことがあります。あらかじめご承知おきください。

・本書の内容についてのご意見、ご質問は、お名前、ご連絡先を明記のうえ、小社出版部宛文書（郵送またはE-mail）でお送りください。
・電話によるお問い合わせはお受けできません。
・本書の解説範囲を越える内容のご質問や、本書の内容と無関係なご質問にはお答えできません。
・匿名のフリーメールアドレスからのお問い合わせには返信しかねます。

はじめに

jq は、JSON からほしいデータだけをさくっと抽出するコマンドライン指向のツールです。REST API の応答をそのままパイプ経由で処理できるため、DevOps や NetOps に好んで利用されています。

たとえば、Github から octocat のリポジトリのリストを得るなら、次のようにするでしょう。

```
$ curl -s https://api.github.com/users/octocat/repos | jq -r '.[].name'
boysenberry-repo-1
git-consortium
hello-worId
Hello-World
⋮
```

jq はしかし、値を抜き出すだけのツールではありません。他のテキスト処理ツールと比べても遜色のない機能が備わっているので、日付文字列を変換する、２つの配列から共通要素を探すなどの操作もできます。while や if などの制御構造や変数定義の機能もあるので、プログラミングらしいこともできます。関数を作成して呼び出すといった、シェルスクリプト的な使い方も可能です。

とは言え、jq は根源的に、フィルタの処理結果を次のフィルタに入力することで段階的に入力を変形していくパイプライン処理メカニズムです。一般のスクリプト言語にはない制約やスタイルがあり、awk などと比べるとわかりにくいところがあります。

本書は、値の抽出にとどまらないこれら jq の高度な使い方を紹介します。複雑な JSON テキストの処理を jq 単体で書けるようになるのが目標です。不要な行の削除や出力の調整は grep などのおなじみのツールに任せてパイプで組み合わせることもできますが、1 つの処理系だけで書ければ、メンテナンスや共有が楽になります。

本書では、各種の問題に対するレシピを jq の関数の形で提示します。読者はそれら関数をそのまま利用してもよいですが、本書の主たる目的は、類似の問題を自力で解決できるよう、jq のコーディングパターンを理解するところにあります。もちろん、すべてのユースケースを紹介できるわけはありませんし、掲載した解法よりも美しい方法もあるでしょうが、主要なコーディング上の疑問には答えられるのではと思っています。

本書は、ある程度まで jq を使い込んだ読者をターゲットとしています。ベーシックな用法は、導入篇の『jq ハンドブック』（カットシステム、2021 年）、あるいは次に示すオフィシャルな jq マニュアルを参照してください。

```
https://stedolan.github.io/jq/manual/v1.6/
```

jq のコマンドオプション、特殊記号、演算子、関数、定数は、筆者の次の Github ページにまとめてあります。リファレンスとしてご参照ください。

```
https://github.com/stoyosawa/jqDoc-public
```

JSON を日常的に利用する NetOps や DevOps 諸氏のお役に立てれば幸いです。

2023 年 3 月
豊沢 聡

本書の利用法

本書の各節では、設定した問題、その解法である jq の関数、その用法と説明を示します。たとえば、「オブジェクトのプロパティ名と値を入れ替えたい」という問題に対しては、次の関数を提供します［レシピ 003］。

```
def swap_property:
    with_entries({"key": (.value | tostring), "value": .key});
```

関数は章単位にまとめたファイルに収容したので、include（あるいは import）ディレクティブからファイルを読み込めば、呼び出せます。次の例では、ファイル object.jq に収容した上記の swap_property 関数を呼び出しています。

```
$ echo '{"キー":"値"}' | jq 'include "object"; swap_property'
{
  "値": "キー"
}
```

自作関数もビルトイン同様に利用できるので、ビルトインのフィルタと組み合わせられます。たとえば、上記のオブジェクトからプロパティ値を抜き出すには、次のようにします。

```
$ echo '{"キー":"値"}' | jq 'include "object"; swap_property | ."値"'
"キー"
```

jq 関数はポピュラーな話題ではないので、付録 A で作成方法と注意事項を示しました。慣れない方は先にそちらをお読みください。

関数呼び出しが望まれないときは、def 以下のブロックをコピーして使えます。関数と言っても、中身はコマンドラインで記述するフィルタと変わりありません（単にまとめているだけなのはシェルスクリプトの関数と同じです）。ただし、末尾のセミコロン ; は関数定義の末尾を示すデリミタなので、直接利用のときは外します。

```
#                                                    ここにあったセミコロンは抜く ↓
$ echo '{"キー":"値"}' | jq 'with_entries({"key":(.value|tostring), "value":.key})'
{
  "値": "キー"
}
```

複数行の関数も、単一引用符でくくられていれば、インデントや改行込みのままコピーペーストで使えます。次に示すのはNの階乗（N!）を計算する手すさび関数ですが［レシピF02］、これもdef以下の行をフィルタを記述する箇所にそのまま張り込むだけです。

```
def factorial:                                          # 関数定義
    tonumber |
    reduce range(1; .+1) as $n (
        1;
        . * $n
    );
```

```
$ echo 10 | jq '    tonumber |                          # そのままコピペ
>     reduce range(1; .+1) as $n (
>         1;
>         . * $n
>     )'                                                # 閉じ単一引用符を忘れずに
3628800                                                 # 10!
```

本書の構成

本書は、JSONデータ型単位に章立てしています。ただし、nullや真偽値に関する問題は数が少ないので、独立した章にはしていません。たとえば、「空の値ならすべてnullと判断する」という問題は、「空の値」に0も含んでいるので、数値を扱う第1章で取り上げています。また、配列についてはジャンルに分けて4つの章で扱います。

関数は、それぞれの章につき1つのファイルにまとめています。たとえば、第1章の関数はファイルnumber.jqに定義しています。

レシピ番号は章タイトルに「近い」アルファベット1文字を先頭にして順に付けています（先頭文字に重なりが多いので苦し紛れのものもいくつかあります）。第1章の10番目のレシピはN10です。

各章の概要を次表に示します。

章番号	内容	関数ファイル	レシピ番号
第 1 章	数値関連	number.jq	N01 ～ N20
第 2 章	文字列関連	string.jq	S01 ～ S25
第 3 章	配列関連	array.jq	A01 ～ A19
第 4 章	集合関連（配列）	set.jq	M01 ～ M10
第 5 章	統計関連（配列）	stat.jq	E01 ～ E08
第 6 章	オブジェクト関連	object.jq	O01 ～ O13
第 7 章	SQL 関連（配列）	sql.jq	Q01 ～ Q19
第 8 章	アルゴリズム関連（手すさび）	fun.jq	F01 ～ F08
第 9 章	その他	others.jq	T01 ～ T06
付録 A	関数定義の説明	sample.jq	番号なし
付録 B	ビルトイン関数の用法	-	-
付録 C	用語集	-	-
付録 D	記号一覧	-	-
付録 E	本書の関数の一覧	-	-

第 4 章、第 5 章、第 7 章は、第 3 章と同じく配列が対象です。ただ、第 4 章は配列を集合、第 5 章は配列を実験データなどの数値の羅列、第 7 章はオブジェクトの配列をデータベースの「テーブル」としてそれぞれ見立て、その手法を用いたレシピを掲載しています。たとえば、第 4 章では 2 つの配列の積集合（intersection）を、第 5 章では配列の値の標準偏差（stdev）を、第 7 章ではテーブルに対する SELECT や UPDATE を取り上げます。

第 8 章は手すさびの章です。ここにはフィボナッチ数列や素数列の計算など、jq でやろうとは思わないアルゴリズムなレシピを収録しています。遊びではありますが、根源的にパイプライン処理マシンである jq で複雑な問題に取り組むときのコーディングパターンを感じ取れると思います。再帰呼び出しも紹介します。

jq の高度な関数（map や with_entries）の用法は付録 B で補足説明しました。jq あるいは JSON 固有の用語（「エントリー化」や「パス配列」）は付録 C に、jq で多用される特殊記号（[] や .）は付録 D にまとめたので、参考にしてください。

関数ファイルは出版社のダウンロードサービスからダウンロードできます。

注意事項

以下、本書で使用するサンプルファイルや表記などで注意すべき点を説明します。

jq のバージョン

本書で使用し、動作を確認した jq は現在最新のバージョン 1.6（2018 年 11 月 1 日リリース）です。jq のバージョンは --version コマンドオプションから確認できます。

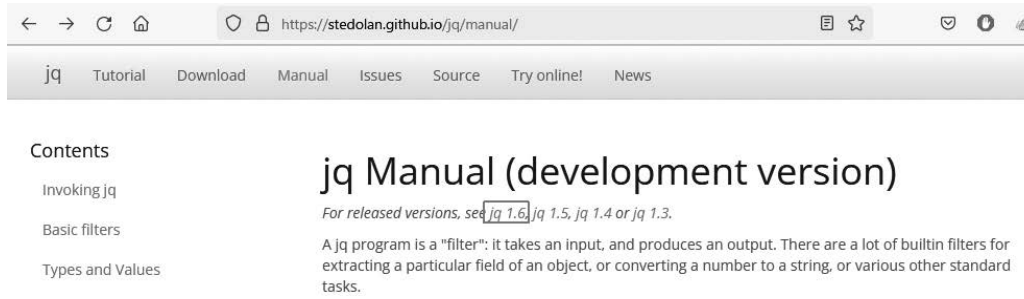
```
$ jq --version
jq-1.6
```

ディストリビューションによっては、正規表記ライブラリ抜きでコンパイルされたものもあります。抜きでビルドされたかは、次のように正規表現フィルタを試すことで確認できます（入力文字列が正の整数なら true を返す）。

```
$ jq -n '"17" | test("^\\d+$")'
jq: error (at <stdin>:0): jq was compiled without ONIGURUMA regex library.
  match/test/sub and related functions are not available
```

とくに理由がなければ、1.6 の正規表現付きのものをインストールしてください。

jq マニュアルはバージョン毎に分かれています。トップページの［Manual］からアクセスすると次図のように開発版の「jq Manual (develpoment version)」が表示されます。一部 1.6 にはない機能も記述されているので、1.6 のものを参照してください。



対象環境

JSON および jq はプラットフォーム独立なので、実行環境は問いません。しかし引用符やエスケープ文字の扱いに癖のあるコマンドライン環境もあります。そのため、本書では Unix（あるいは Windows Subsystem for Linux）のみを使用しています。たとえば、フィルタをくくる引用符には単一引用符 ' を使っています。

Windows で利用できないわけではありませんが、くくり文字に二重引用符 " を使う（単一引用符は使えません）、引用内部の二重引用符（JSON 文字列のくくり）を適切に扱うなど、コマンドプロンプトでの使用で配慮が必要です。

サンプルの入力

用例の JSON テキストは、短いものなら echo コマンドから、あるいは -n（ロングフォーマットは --null-input）を用いてフィルタ内部の即値から入力します。次の例は、3.1415 を入力し、その正弦関数値を求めています。

```
$ echo 3.1415 | jq 'sin'                                    # echoから入力
9.265358966049026e-05

$ jq -n '3.1415 | sin'                                      # -nから即値で入力
9.265358966049026e-05
```

長いものは事前に掲載し、実行時では入力部分を次のように ... で省いて示します。

```
$ ... | jq 'include "stats"; stdev'
```

これら長いサンプルは、ファイル function.json にオブジェクトプロパティとして収容してあります。これも出版社のダウンロードサービスからダウンロードできます。中身を一部示します。

```
$ cat function.json
{
⋮
  "sake_table": [
    ["獺祭", "東洋美人", "五橋", "富久寿", "五橋"],
    ["磯自慢", "志田泉", "喜久酔", "花の舞"],
    ["十四代", "上喜元", "出羽桜", "初孫"]
  ],
```

```
 ⋮
  "stats": [44, 40, 42, 44, 44, 42, 40, 44, 45, 44],
 ⋮
}
```

利用にあたっては、jq .stats function.json のように個々のプロパティを選択的に抽出します。次に、プロパティ .sake1 の値（配列）を用いるときの例を示します。

```
$ jq '.sake1' function.json
[
  "獺祭",
  "東洋美人",
  "五橋",
  "富久寿",
  "五橋"
]
```

構造上の制限から function.json に収容できなかったサンプルファイル（1 行 1 行が独立した JSON テキストなファイルなど）もありますが、これらもレシピでその都度説明を加えています。

もっとも、ファイルの中身そのものに意味があるわけではないので、似たような構造なら好みの JSON テキストでテストしても問題はありません。

実行例

基本は全出力の掲載ですが、さしさわりのない長いものはドット（⋮や ...）で省略しています。

jq のデフォルト出力フォーマットそのままでは読みにくいものは、-c コマンドプロンプト（ロングフォーマットは --compact-output）あるいは -r（--raw-output）を介しています。それでもまだ読みにくいものは、手で整形しています。実行例をそのまま実行してもまったく同じ出力にならないものもある点、ご注意ください。

コメント

本書に記載したサンプルおよび実行例の JSON テキストには、ハッシュ # を先付けした筆者の注釈が加えられているものもあります。あくまで参考のためです。JSON テキストにはコメントは記述できないので、そのままコピーすると、# 部分がエラーになります。

jq フィルタはコメントを受け付けます。# 以降が行末まで無視されるので、行単位で書かれた関数コードでは有効ですが、コマンドラインから 1 行で記述するフィルタでは実用上利用できません。

目次

第1章

数値関数

本章では、数値を入力として受ける関数のレシピを紹介します。
本章の関数はファイル number.jq にまとめてあります。実行例では次のように呼び出します。

```
$ jq -n 'include "number"; m_pi'                    # πを出力する関数m_pi
3.141592653589793
```

N01 πを得る

目的

高精度なπ（円周率）がほしい（たいていの言語にはこの定数があります）。

関数

```
def m_pi:
    3.141592653589793;
    # 1 | atan * 4;
```

用例

次の例では、πの値を出力しています。

```
$ jq -n 'include "number"; m_pi'
3.141592653589793
```

次の例では、60 度の sin 値を求めています。ビルトイン関数 sin はラジアンしか受け付けません。

```
$ jq -n 'include "number"; 60 * m_pi / 180 | sin'
0.8660254037844386
```

説明

直値だけのフィルタは、入力をすべて無視してその値を出力します。

借用元の C 言語の M_PI（math.h ヘッダ）より桁数が短いのは、それが jq の倍精度浮動小数点数の精度範囲だからです。ヘッダから直接コピーしたものと、jq による処理後を比較します。jq は精度範囲外の数値が入力されると自動的にその範囲内に丸めます。

```
#       Cの定数。下5桁分長い
$ echo 3.14159265358979323846 | tee /dev/tty | jq '.'
3.14159265358979323846                                  # もとのM_PI
3.141592653589793                                       # jqの精度範囲
```

直値が美しくないと思うのなら、コメントアウトしている行を代わりに使用します。$\tan^{-1}(1)$ は π /4 を返します。あるいは［レシピ F07］を参照してください。

sin などラジアンを使用する三角関数を角度（°）から使えるようにするのが主たる用途です。後述の［レシピ N03］や［レシピ N05］内からも呼び出しています。

N02 e を得る

目的

高精度な e（ネイピア数）がほしい（たいていの言語にはこの定数がある）。

関数

```
def m_e:
    2.718281828459045;
    # 1 | exp;
```

用例

次の例では、ネイピア数の値を出力しています。

```
$ jq -n 'include "number"; m_e'
2.718281828459045
```

次の例では、e の 2 乗をビルトイン関数の pow から計算しています。

```
$ jq -n 'include "number"; pow(m_e; 2)'
7.3890560989306495
```

説明

あらかじめ用意した定数を返すだけの関数です。値は C 言語の M_E（math.h ヘッダ）から借用しましたが、［レシピ N01］のπ同様、jq の浮動小数点数の精度範囲内で丸めています。

直値が嫌いなら、コメントアウトしている行を代わりに使用します。exp から e^1 を計算すれば得られます。

イチから計算したい好事家には、級数から e を算出する関数を［レシピ F08］で紹介します。

N03 ラジアンを角度に変換する

目的

ラジアンを角度に変換したい（Python の math.degrees と類似の機能）。

関数

```
def to_degree:
    . * 180 / m_pi;
```

用例

次の例では、三角形の底辺と高さの比を配列で示した [2, 1] から、その三角形の角度を求めています。ビルトイン関数 atan の出力はラジアンなので、この関数から角度に変換します。

```
$ echo '[2, 1]' | jq 'include "number"; .[1] / .[0] | atan | to_degree'
26.56505117707799
```

説明

入力値 . に 180 を掛けて π で割るだけです。π は［レシピ N01］で作成した m_pi 関数を利用しています。同じ number.jq に属する関数なので、そのまま呼び出せます。

N04 角度をラジアンに変換する

目的

角度をラジアンに変換したい（Python の math.radians と類似の機能）。

関数

```
def to_radian:
    . * m_pi / 180;
```

用例

次の例では、角度単位で sin(30) を計算しています。

```
$ echo 30 | jq 'include "number"; to_radian | sin'
0.49999999999999994
```

説明

ここでも［レシピ N01］の m_pi 関数を利用しています。中身は［レシピ N04］の逆です。

N05 絶対値を計算する

目的

絶対値を計算したい。

関数

```
def abs:
    fabs;
```

用例

次の例では、-2.3 の絶対値を求めています。

```
$ jq -n 'include "number"; -2.3 | abs'
2.3
```

説明

ビルトイン関数の fabs（floating point abs）がすでにありますが、C の関数に慣れていないと、しばしば f を付け忘れます。そこで、異なる名前で再定義します。

関数定義そのものにおもしろみはありませんが、これは、関数が定義したブロックを関数名と単純に置き換えるだけのマクロでしかないことを示しています。用例の -2.3 | abs は、置き換えられて -2.3 | fabs になります。この構造は、関数ブロックがどんなに長く複雑になっても変わりません。

N06 整数のビット長を返す

目的

整数のビット長を知りたい（Python の int.bit_length と類似の機能）。

関数

```
def bit_length:
    if . == 0 then
        0
    else
```

```
        fabs |
        log2 | floor + 1
    end;
```

用例

次の例では、235_{10}（2 進数にして $1110\ 1011_2$）のビット長を計算しています。

```
$ jq -n 'include "number"; 235 | bit_length'
8
```

関数は負の値も正として扱うので、–235 にしても結果は同じです。

```
$ jq -n 'include "number"; -235 | bit_length'
8
```

次の例では、0 から 15 までの数値をそれぞれのビット長を配列で出力します。連続した数値の生成には range を使います（付録 B.1）。配列要素を順次処理した結果を再び配列として返すには、map です（付録 B.4）。

```
$ jq -cn 'include "number"; [ range(0; 16) ] | map(bit_length)'
[0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4]
```

説明

整数 N のビット長は $\log_2(N)$ から計算できます。ビルトイン関数は log2 です。

ただし、log2 は N = 0 のとき、エラーこそ報告しませんが負の無限大を出力します。値が無限大かは isinfinite から確認できます。

```
$ jq -n '0 | log2'                                  # log2(0)
-1.7976931348623157e+308

$ jq -n '0 | log2 | isinfinite'                     # log2(0)は無限
true
```

数値 0 のビット数は 0 と定義されているので、N = 0 だけは if-then-else-end で例外処理をします（付録 B.2）。

fabs が用いられていることからわかるように、負の値は正として計算しています。

log2 の結果は浮動小数点数になるので、切り捨ての floor から整数に直します。切り上げの ceil を使わず、切り捨ててから 1 を加えているのは、2 の N 乗には N+1 ビットが必要だからです（たとえば、$2^2 = 4_{10} = 100_2$ なので、3 ビットです）。次の例では 2 の 1 乗から 5 乗までの値の $\log_2$+1 を計算しています。

```
$ jq -cn '[2, 4, 8, 16, 32] | map(log2 + 1)'
[2,3,4,5,6]
```

N07 10 進数を N 進数に変換する

目的

10 進数を任意の N 進数に変換したい。

関数

```
def base_n_array($base):
    fabs | floor |
    [., . % $base] |
    [
        while (
            .[0] != 0;
            [.[0] / $base | floor, . % $base]
        )
    ] |
    map(.[1]) |
    reverse;
```

用例

次の例では、10 進数で 7585_{10} を 7 進数に変換します。

```
$ echo 7585 | jq -c 'include "number"; base_n_array(7)'
[3,1,0,5,4]
```

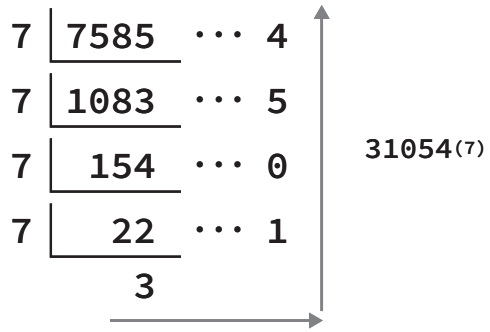
出力は配列です。あとから 2 進数化や 16 進数化でこの関数を用いるので、（関数からの）使い

勝手を考慮して数値配列のまま出力するように設計しています。文字列が必要なら、tostring と join を通します（ただし N ≦ 10 のときのみ）。

```
$ echo 7585 | jq -r 'include "number"; base_n_array(7) | map(tostring) | join("")'
31054
```

説明

10 進数を N 進数に変換するには、商が 0 になるまで N で順次割っていき、その剰余を並べます（高校数学で出てくる方法です）。次の図は 7585_{10} を 7 進数にするために順次 7 で割っていき、その余りの 45013 をひっくり返し、31054_7 を得る様子を示しています。



関数は数値を入力として受け、パラメータ引数に指定した値（$base）を底としてこの計算をします。

最初に、fabs と floor で正の整数化することで、入力をクリーンにしています（文字列や配列が入力されたときは、手当てをしていないのでエラーが上がります）。

図の割り算には、割られる数（前の商）と剰余の 2 つの値を収容するコンテナが必要です。ここではこれを配列で表現しています。初期状態では第 0 要素が入力値（割られる数）、第 1 要素が $base で割ったときの剰余です。上図では [7585, 4] です。

あとは割り算を続けていくわけですが、これには while でループを組みます（付録 B.3）。第 1 引数にはループを続行する条件式を記述します。割られる数 .[0] が 0 以外ならループ続行なので、ここは .[0] != 0 です。第 2 引数では配列を更新します。ここでポイントは、まず割られる数を先に割り（.[0] / $base |）、この出力の整数値（. | floor、すなわち floor）を第 0 要素に、その値の剰余（. % $base）を第 1 要素にしているところです。

while はこの [割られる数 , 剰余] 配列（.）を毎回出力します。while の出力は、全体を [] で

くくることで配列化します（付録 B.8）。つまり、[while ...] が生成するのは配列の配列です。コードの最後の 2 行を除けば、ループ時の状況を確認できます。次は、図と同じ 7585 と底 7 を指定したときの出力です。

```
[
  [7585, 4],
  [1083, 5],
  [154, 0],
  [22, 1],
  [3, 3]
]
```

図と同じパターンが得られました。

あとはこの配列の配列から、内側の配列の第 1 要素（.[1]）を抜き出すだけです。これには、配列の全要素を順に操作して置き換える map を使います（付録 B.4）。（フィルタ版の）高階関数なので、引数にはどんなフィルタでも書き込めます。

配列の並びは図の上からなので、下から読むために reverse で反転します。

N08 10 進数を 2 進数に変換する

目的

10 進数を文字列表記の 2 進数に変換したい（Python の bin と類似の機能）。なお、JSON には数値を 2 進数のまま表記する方法がないので、Python 風に先頭に 0b を加えた文字列として出力します。

関数

```
def bin:
    base_n_array(2) |
    map(tostring) |
    "0b" + join("");
```

用例

実行例を示します。先ほど同様、入力値は 7585 とします。

```
$ echo 7585 | jq -r 'include "number"; bin'
0b1110110100001
```

説明

［レシピ N07］の汎用 N 進数関数を早速用いています。あとは、得られた配列の要素を数値から文字列に変換し、join で連結します。このとき、先頭に 0b も加えます。

N09 10 進数を 16 進数に変換する

目的

10 進数を文字列表記の 16 進数に変換したい（Python の hex と類似の機能）。16 進数もやはり表記は文字列にし、先頭に 0x を加えます。

関数

```
def hex:
    ["0", "1", "2", "3", "4", "5", "6", "7",
     "8", "9", "A", "B", "C", "D", "E", "F"] as $hexadecimal |
    base_n_array(16) |
    map($hexadecimal[.]) |
    "0x" + join("");
```

用例

実行例を示します。先ほど同様、入力値は 7585 とします。

```
$ echo 7585 | jq -r 'include "number"; hex'
0x1DA1
```

説明

16 進数はアルファベットの A から F を使うので、たとえば 10 = "A" などの対応付けが必要です。これを定義しているのが最初の行の $hexadecimal 変数です（付録 B.5）。配列なので、$hexadecimal[n] の要領で 10 進数 n を指定すれば、その 16 進数文字列が得られます。

ASCII コードから計算しないのはプログラマの名折れだと思う方は、次のレシピを参照してください。比べるとわかりますが、たかだか 16 文字なので、こちらの方が絶対楽で高速です。