プリミティブ型

プログラムの解説にはよく、リテラルという言葉がでてきます。 リテラルというのは、専用の文法を持ち、ソースコード中に直接記述できるデータのことです。TypeScriptには何種類かあります。

- `boolean` 型
- `number` 型
- `string` 型
- 配列
- オブジェクト
- 関数
- `undefined`
- `null`

このうち、それ以上分解できないシンプルなデータを「プリミティブ型」と呼びます。 ここでは、よく出てくるプリミティブ型を紹介します。

配列とオブジェクトは次章の複合型で、関数は関数で紹介します。

`boolean` リテラル

`boolean` 型は `true` / `false` の2つの真偽値を取るデータ型です。 `if` 文、 `while` ループなどの制御構文や、三項演算子などを使ってプログラムの挙動をコントロールするために大切な型です。

```
// 値を表示
console.log(true);
console.log(false);

// 変数に代入。変数の型名はboolean
const flag: boolean = true;

// 他のデータ型への変換
console.log(flag.toString()); // 'true' / 'false' になる
console.log(String(flag));    // こちらでも変換可能
console.log(Number(flag));    // 1, 0になる

// 他のデータ型をtrue/falseに変換
const notEmpty = Boolean("test string"); // 変換ルールは後述
const flag = flagStr === 'true';         // 'true'の文字をtrueにするなら
const str = "not empty string";          // true/false反転するが演算子一つで変換可能
const isEmpty = !str;                    // 反転すると!Boolean()と同じ
const notEmpty = !!str;                  // もう1つ使うと反転せずにboolean型に
```

TypeScriptでは、数字のゼロ（負も含む）、空文字列、`null`、`undefined`、`NaN`を変換すると`false`、それ以外を変換すると`true`になります[1]。

1 この真偽値への変換ルールは言語によって異なります。例えば、Pythonでは空の配列や辞書も偽（`False`）になります。Rubyの場合は数字のゼロや空文字列も真（`true`）になります。

ド・モルガンの法則

`if`文の条件式が複雑なときに、それを簡単にするのにごくたまに役立つのがド・モルガンの法則です。 次のような法則でNOTとANDとORの組みを変換できます。

ド・モルガンの法則

```
!(P || Q) == !P && !Q
!(P && Q) == !P || !Q
```

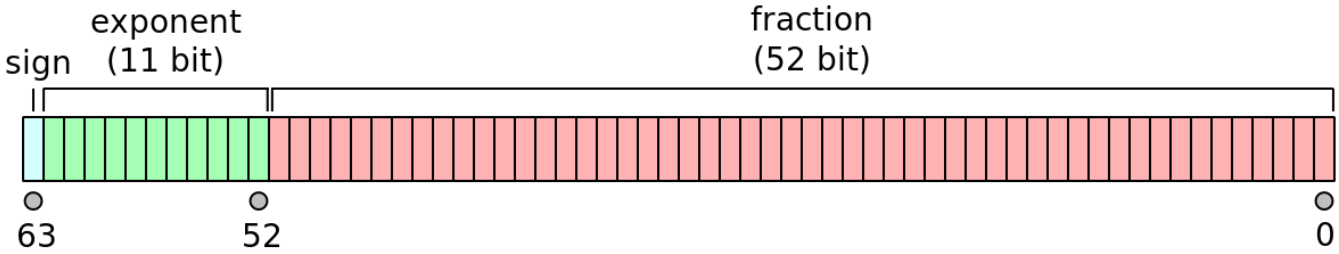
特に、右辺から左辺への変換がコードの可読性を高めることが多いと思います。 NOTの集合同士の演算というのは普段の生活ではあまり出てきません。 集合のAND/ORを考えてから逆転させる方が簡単にイメージできる思いますので、条件を書く時に、想定が漏れてロジックが正しく動作しない、ということが減るでしょう。 また、より構成要素が多い論理式のときに、式を整理するのにも使えます。

数値型

TypeScriptには組み込みで2種類の数値型があります。ほとんどの場合は`number`だけで済むでしょう。

`number`

TypeScript（というか、その下で動作しているJavaScript）は64ビットの浮動小数点数で扱います。これはどのCPUを使っても基本的に同じ精度を持ちます[2]。整数をロスなく格納できるのは53ビット(-1)までなので、±約9007兆までの整数を扱えます。それ以上の数値を入れると、末尾が誤差としてカットされたりして、整数を期待して扱うと問題が生じる可能性があります。正確な上限と下限は`Number.MAX_SAFE_INTEGER`、`Number.MIN_SAFE_INTEGER`という定数で見ることができます。また、`Number.isSafeInteger(数値)`という関数で、その範囲内に収まっているかどうかを確認できます。



The memory format of an IEEE 754 double floating point value. by Codekaizen (CC 4.0 BY-SA)

2 IEEE 754という規格で決まっています。

```
// 値を表示
console.log(10.5);
console.log(128);
console.log(0b11);  // 0bから始まると2進数
console.log(0o777); // 0oから始まると8進数
console.log(0xf7);  // 0xから始まると16進数

// 変数に代入。変数の型名はnumber
const year: number = 2019;

// 他のデータ型への変換
console.log(year.toString());  // '2019'になる
console.log(year.toString(2)); // toStringの引数で2進数-36進数にできる
console.log(Boolean(year));    // 0以外はtrue

// 他のデータ型を数値に変換
console.log(parseInt("010"));     // →10 文字列はparseIntで10進数/16進数変換
console.log(parseInt("010", 8));  // →8 2つめの数値で何進数として処理するか決められる
console.log(Number(true));        // boolean型はNumber関数で0/1になる
```

変換の処理は、方法によって結果が変わります。10進数を期待するものはradix無しの `parseInt()` で使っておけば間違いありません。

文字列から数値の変換

	`Number(文字列)`	`parseInt(文字列)`	`parseInt(文字列, radix)`	リテラル
"10"	10	10	radixによって変化	10
"010"	10	10	radixによって変化	8 (8進数)
"0x10"	16	16	radixが16以外は0	16 (16進数)
"0o10"	8	0	0	8 (8進数)
"0b10"	2	0	0	2 (2進数)

なお、リテラルの8進数ですが、ESLintの推奨設定を行うと `no-octal` というオプションが有効になります。このフラグが有効だと、8進数を使用すると警告になります。

注釈

IE8以前及びその時代のブラウザは、 `parseInt()` に0が先頭の文字列を渡すと8進数になっているため、かならずradixを省略せずに10を設定しろ、というのが以前言われていました。その世代のブラウザは現在市場に出回っていないため、10は省略しても問題ありません。

また、8進数リテラルは以前のJavaScriptは `0777` のように、ゼロ始まりのものも使えましたが、現在はこちらの記法はES5以上で非推奨となっており、TypeScriptではエラーになります。

数値型の使い分け

TypeScriptには組み込みで2種類の数値型があります。 2つの型を混ぜて計算することはできません。

- `number` : 64ビットの浮動小数点数
- `bigint` : 桁数制限のない整数（ `10n` のように、後ろにnをつける）

`number` 型は多くのケースではベストな選択になります。特に浮動小数点数を使うのであればこちらしかありません。 それ以外に、±2^53-1までであれば整数として表現されるので情報が減ったりはしません。また、これらの範囲では一番高速に演算できます。

`bigint` 型は整数しか表現できませんが、桁数の制限はありません。ただし、現時点では `"target": "esnext"` と設定しないと使えません。 使える場面はかなり限られるでしょう。本ドキュメントでは詳しく扱いません。

`number` は2進数で表した数値表現なので、 0.2 + 0.1などのようなきれいに2進数で表現できない数値は誤差が出てしまいます。金額計算など、多少遅くても正確な計算が必要な場合は decimal.js [3] などの外部ライブラリを使います。

numberの誤差

```
const a = 0.1;
const b = 0.2;
console.log(a + b);
// 0.30000000000000004
```

3 https://www.npmjs.com/package/decimal.js

演算子

`+`、`-`、`*`、`/`、`%`（剰余）のよくある数値計算用の演算子が使えます。これ以外に、`**`というベキ乗の演算子がES2016で追加されています[4]。

また、`number`は整数としても扱えますのでビット演算も可能です。ビット演算は2進数として表現した表を使って、計算するイメージを持ってもらえると良いでしょう。コンピュータの内部はビット単位での処理になるため、高速なロジックの実装で使われることがよくあります[5]。

ただし、ビット演算時には精度は32ビット整数にまで丸められてから行われるため、その点は要注意です。

AND	`a & b`	**2つの数値の対応するビットがともに1の場合に1を返します。**
OR	`a | b`	2つの数値の対応するビットのどちらかが1の場合に1を返します。
XOR	`a ^ b`	2つの数値の対応するビットのどちらか一方のみが1の場合に1を返します。
NOT	`~a`	ビットを反転させます
LSHIFT	`a << b`	aのビットを、b(32未満の整数)分だけ左にずらし、右から0をつめます。
RSHIFT	`a >> b`	aのビットを、b(32未満の整数)分だけ右にずらし、左から0をつめます。符
0埋めRSHIFT	`a >>> b`	aのビットを、b(32未満の整数)分だけ右にずらし、左から0をつめます。

なお、トリッキーな方法としては、次のビット演算を利用して、小数値を整数にする方法があります。なぜ整数になるかはぜひ考えて見てください[6]。これらの方法、とくに後者の方は小数値を整数にする最速の方法として知られているため、ちょっと込み入った計算ロジックのコードを読むと出てくるかもしれません。

- `~~`を先頭につける
- `|0`を末尾につける

4 以前は`Math.pow(x, y)`という関数を使っていました。

5 ビット演算を多用する用途としては、遺伝子情報を高速に計算するのに使うFM-Indexといったアルゴリズムの裏で使われる簡潔データ構造と呼ばれるデータ構造があります。

6 ただし、ビット演算なので、本来扱えるよりもかなり小さい数字でしか正常に動作しません。

特殊な数値

数値計算の途中で、正常な数値として扱えない数値が出てくることがあります。業務システムでハンドリングすることはあまりないと思います。もし意図せず登場することがあればロジックの不具合の可能性が高いでしょう。

- 無限大: `Infinity`
- 数字ではない: `NaN` (Not a Number)

`Math` オブジェクト

TypeScriptで数値計算を行う場合、 `Math` オブジェクトの関数や定数を使います。

数値の最大値、最小値

関数	説明
`Math.max(x, y, …)`	複数の値の中で最大の値を返す。配列内の数値の最大値を取得したい場合は `Math.ma`
`Math.min(x, y, …)`	複数の値の中で最小の値を返す。配列内の数値の最小値を取得したい場合は `Math.mi`

乱数生成の関数もここに含まれます。0から1未満の数値を返します。例えば0-9の整数が必要な場合は、10倍して `Math.floor()` などを使うと良いでしょう。

乱数

関数	説明
`Math.random()`	0以上1未満の疑似乱数を返す。暗号的乱数が必要な場合は `crypto.randomBytes()` を代わり

整数に変換する関数はたくさんあります。一見、似たような関数が複数あります。例えば、 `Math.floor()` と `Math.trunc()` は似ていますが、負の値を入れた時に、前者は数値が低くなる方向に（-1.5なら-2）丸めますが、後者は0に近く方向に丸めるといった違いがあります。

整数変換

関数	説明
`Math.abs(x)`	xの絶対値を返す。
`Math.ceil(x)`	x以上の最小の整数を返す。
`Math.floor(x)`	x以下の最大の整数を返す。
`Math.fround(x)`	xに近似の単精度浮動小数点数を返す。ES2015以上。
`Math.round(x)`	xを四捨五入して、近似の整数を返す。
`Math.sign(x)`	xが正なら1、負なら-1、0なら0を返す。ES2015以上。
`Math.trunc(x)`	xの小数点以下を切り捨てた値を返す。ES2015以上。

整数演算の補助関数もいくつかあります。ビット演算と一緒に使うことが多いと思われます。

32ビット整数

関数	説明
`Math.clz32(x)`	xを2進数32ビット整数値で表した数の先頭の0の個数を返す。ES2015以上。
`Math.imul(x, y)`	32ビット同士の整数の乗算の結果を返す。超えた範囲は切り捨てられる。主にビット

平方根などに関する関数もあります。

ルート

関数・定数	説明
`Math.SQRT1_2`	1/2の平方根の定数。
`Math.SQRT2`	2の平方根の定数。
`Math.cbrt(x)`	xの立方根を返す。ES2015以上。
`Math.hypot(x, y, …)`	引数の数値の二乗和の平方根を返す。ES2015以上。
`Math.sqrt(x)`	xの平方根を返す。

対数関係の関数です。

対数

関数・定数	説明
`Math.E`	自然対数の底（ネイピア数）を表す定数。
`Math.LN10`	10の自然対数を表す定数。
`Math.LN2`	2の自然対数を表す定数。
`Math.LOG10E`	10を底としたeの対数を表す定数。
`Math.LOG2E`	2を底としたeの対数を表す定数。
`Math.exp(x)`	`Math.E ** x` を返す。
`Math.expm1(x)`	`exp(x)` から1を引いた値を返す。ES2015以上。
`Math.log(x)`	xの自然対数を返す。

関数・定数	説明
`Math.log1p(x)`	1 + x の自然対数を返す。ES2015以上。
`Math.log10(x)`	xの10を底とした対数を返す。ES2015以上。
`Math.log2(x)`	xの2を底とした対数を返す。ES2015以上。

最後は円周率や三角関数です。引数や返り値で角度を取るものはすべてラジアンですので、度（°）で数値を持っている場合は `* Math.PI / 180` でラジアンに変換してください。

`string` リテラル

`string` リテラルは文字列を表現します。シングルクオート、ダブルクオートでくくると表現できます。 シングルクオートとダブルクオートは、途中で改行が挟まると「末尾がない」とエラーになってしまいますが、 バッククオートでくくると、改行が中にあっても大丈夫なので、複数行あるテキストをそのまま表現できます。

```
// 値を表示
// シングルクオート、ダブルクオート、バッククオートでくくる
console.log('hello world');

// 変数に代入。変数の型名はstring
const name: string = "future";

// 複数行
// シングルクオート、ダブルクオートだとエラーになる
// error TS1002: Unterminated string literal.
const address = '東京都
品川区';
// バッククオートならOK。ソースコード上の改行はデータ上も改行となる
const address = `東京都
品川区`;

// 他のデータ型への変換
console.log(parseInt('0100', 10)); // 100になる
console.log(Boolean(name));        // 空文字列はfalse、それ以外はtrueになる

// 他のデータ型をstringに変換
const year = 2019;
console.log((2019).toString(2));     // numberはtoStringの引数で2進数-36進数にできる
console.log((true).toString());      // boolean型を'true'/'false'の文字列に変換
console.log(Stirng(false));          // こちらでも可
```

JavaScriptはUTF-16という文字コードを採用しています。Javaと同じです。絵文字など、一部の文字列は1文字分のデータでは再現できずに、2文字使って1文字を表現することがあります。これをサロゲートペアと呼びます。範囲アクセスなどで文字列の一部を扱おうとすると、絵文字の一部だけを拾ってしまう可能性がある点には要注意です。何かしらの文字列のロジックのテストをする場合には、絵文字も入れるようにすると良いでしょう。

文字列のメソッド

文字列には、その内部で持っている文字列を加工したり、一部を取り出したりするメソッドがいくつもあります。 かなり古いJavaScriptの紹介だと、HTMLタグをつけるためのメソッドが紹介されていたりもします、TypeScriptの型定義ファイルにも未だに存在はしますが、それらのメソッドは言語標準ではないためここでは説明しません。

文字コードの正規化

ユニコードには、同じ意味だけどコードポイントが異なり、字形が似ているけど少し異なる文字というものが存在します。たとえば、全角のアルファベットのＡと、半角アルファベットのAがこれにあたります。 それらを統一してきれいにするのが正規化です。文字列の `normalize("NFKC")` というメソッド呼び出しをすると、これがすべてきれいになります。

```
> "ＡＢＣｱｲｳｴｵ㍼".normalize("NFKC")
'ABCアイウエオ平成'
```

正規化を行わないと、例えば、「６月６日議事録.md」という全角数字のファイル名を検索しようとして、「6月6日議事録」という検索ワードで検索しようとしたときにひっかからない、ということがおきます。検索対象と検索ワードの両方を正規化しておけば、このような表記のブレがなくなるため、ひっかかりそうでひっかからない、ということが減らせます。

正規形は次の4通りあります。Kがついているものがこのきれいにする方です。また、Dというのは、濁音の記号とベースの文字を分割するときの方法、Cは結合するときの方法になります。macOSの文字コードがNFKDなので、たまにmacOSのChromeでGoogle Spreadsheetを使うと、コピペだかなんだかのタイミングでこのカタカナの濁音が2文字に分割された文字列が挿入されることがあります。 `NFKC` をつかっておけば問題はないでしょう。

- `NFC`
- `NFD`
- `NFKC`
- `NFKD`

正規化をこのルールに従って行うと、ユーザーに「全角数字で入力する」ことを強いるような、かっこ悪いUIをなくすことができます。ユーザーの入力はすべてバリデーションの手前で正規化すると良いでしょう。

ただし、長音、ハイフンとマイナス、漢数字の１、横罫線など、字形が似ているものの意味としても違うものはこの正規化でも歯が立たないので、別途対処が必要です。

文字列の結合

従来のJavaScriptは他の言語でいうprintf系の関数がなく、文字列を`+`で結合したり、配列に入れて`.join()`で結合したりしましたが、いまどきは文字列テンプレートリテラルがありますので、ちょっとした結合は簡単に扱えます。 printfのような数値の変換などのフォーマットはなく、あくまでも文字列結合をスマートにやるためのものですが、複数行のテキストを表現できますし、プレースホルダ内には自由に式が書けます。 もちろん、数が決まらない配列などは従来どおり`.join()`を使います。

```
// 古いコード
console.log("[Debug]: " + variable);

// 新しいコード
console.log(`[Debug]: ${variable}`);
```

このバッククオートを使う場合は、関数を前置することで、文字列を加工することができます。国際化でメッセージを置き換える場面などで利用されます。

文字列の事前処理

テンプレートリテラルに関数を指定すると（タグ付きテンプレートリテラル）、文字列を加工する処理を挟めます。よく使われるケースは翻訳などでしょう。 テンプレートリテラルの前に置かれた関数は、最初に文字列の配列がきて、その後はプレースホルダの数だけ引数が付く構造になっています。 文字列の配列は、プレースホルダに挟まれた部分のテキストになります。 自作する機会は多くないかもしれませんが、コード理解のために覚えておいて損はないでしょう。

タグ付きテンプレートリテラル

```
function i18n(texts, ...placeholders) {
  // texts = ['小動物は', 'が好きです']
  // placeholders = ['小旅行']
  return // 翻訳処理
}

const hobby = "小旅行"
console.log(i18n`小動物は${hobby}が好きです`);
```

`undefined` と `null`

JavaScript/TypeScriptでは、`undefined`と`null`があります。他の言語では`null`（もしくは`None`、`nil`と呼ぶことも）だけの場合がほとんどですが、JavaScript/TypeScriptでは2種類登場します。

このうち、`undefined` は未定義やまだ値が代入されていない変数を参照したり、オブジェクトの未定義の属性に触ると帰ってくる値です。 TypeScriptはクラスなどで型定義を行い、コーディングがしやすくなるとよく宣伝されますが、「`undefined` に遭遇するとわかっているコードを事前にチェックしてくれる」ということがその本質だと思われます。

```
let favoriteGame: string; // まだ代入してないのでundefined;
console.log(favoriteGame);
```

このコードは、`tsconfig.json` で `strict: true` （もしくは `strictNullChecks: true` ）の場合にはコンパイルエラーになります。

JavaScriptはメソッドや関数呼び出し時に数が合わなくてもエラーにはならず、指定されなかった引数には `undefined` が入っていました。 TypeScriptでは数が合わないとエラーになりますが、 `?` を変数名の最後に付与すると、省略可能になります。

```
function print(name: string, age?: number) {
    console.log(`name: ${name}, age: ${age || 'empty'}`);
}
```

意図せずうっかりな「未定義」が `undefined` であれば、意図をもって「これは無効な値だ」と設定するのが `null` です。 ただし、Javaと違って、気軽に `null` を入れることはできません。 変数の章でも紹介した、「AもしくはBのデータが格納できる」という合併型（Union Type）の型宣言ができるので、これをつかって `null` を代入します。 TypeScriptでは「これは無効な値をとる可能性がありますよ」というのは意識して許可してあげなければなりません。

```
// stringかnullを入れられるという宣言をしてnullを入れる
let favaoriteGame: string | null = null;
```

`undefined` と `null` は別のものなので、コンパイラオプションで `compilerOptions.strict: true` もしくは、 `compilerOptions.strictNullChecks: true` の場合は、 `null` 型の変数に `undefined` を入れようとしたり、その逆をするとエラーになります。 これらのオプションを両方とも `false` にすれば、エラーにはならなくなりますが、副作用が大きいため、これらのオプションは有効にして、普段から正しくコードを書く方が健全です。

nullとundefinedは別物

```
const a: string | null = undefined;
// error TS2322: Type 'undefined' is not assignable to type 'string | null'.

const b: string | undefined = null;
// error TS2322: Type 'null' is not assignable to type 'string | undefined'.
```

まとめ

TypeScript（とJavaScript）で登場する、プリミティブ型を紹介してきました。 これらはプログラムを構成する上でのネジやクギとなるデータです。