その他の組み込み型・関数

これまでの説明の中で、いくつかのデータの種類（ここでは誤解をおそれず、大雑把にクラスとしておきます）について説明してきました。

- プリミティブ型
  - `boolean`
  - `number`
  - `string`と正規表現
  - `undefined`と`null`
- 複合型
  - `Array`
  - `Object`
  - `Map`と`WeakMap`
  - `Set`と`WeakSet`
- 関数

TypeScriptとJavaScriptはブラウザの中の言語として作られたため、数多くのブラウザ環境専用のクラスを備えていますが、それ以外にも言語の組み込みのクラスがいくつかありますので、本章ではそれらについて説明していきます。なお、`Error`クラスについては例外処理で触れます。

本章で扱う組み込み型や関数は次の通りです

- `Date`: 日付を扱うクラス
- `RegExp`: 正規表現
- `JSON`: JSON形式の文字列をパースしたり、JavaScriptのオブジェクトをJSON形式にシリアライズするクラス。サーバー通信時のボディでよく利用される。
- `URL`と`URLSearchParams`: URLをパースしたり、URLを組み立てるクラス。サーバー通信時にURLやクエリーを取り出すのに利用される。
- 時間のための関数

`Date`

日付と時間を扱うのが`Date`型です。これは最初期から実装されている型で、TypeScriptやJavaScriptで日付を扱う場合、まず出てくるのがこれです。ただし、即席で作られたこともあって評判がよくなく、大切な機能がいくつか抜け落ちていたり、文字列のパースが柔軟性がなかったりするため、いくつもの追加のモジュールなどが作られています。

以前はmoment.jsが長い間広く使われてきましたが、現在は積極的な開発を中止する声明を出しています。そのmoment.jsの声明の中で推奨されているのが次のライブラリです。

- Luxon (型定義は別に `npm install @types/luxon` が必要)
- Day.js (TypeScript型定義同梱)
- date-funs (TypeScript型定義同梱)
- js-Joda (TypeScript型定義同梱)

より便利なフォーマットやパース、日時演算などはこちらのパッケージを利用すると簡単に行えます。

JavaScript本体においても、`Date`を置き換える`Temporal`が提案されています。

- https://github.com/tc39/proposal-temporal

将来的に、ここの説明は`Temporal`ベースで書き換えられると思いますが、ひとまずここでは`Date`のよくある使い方について紹介していきます。

TypeScriptの`Date`型は数字に毛の生えたようなものですので、それを前提にみていくと良いと思います。

現在時刻の取得・エポック時刻

`new Date()`で簡単に作成できます。コンピュータに保存されているタイムゾーンの情報も含んだ、`Date`のインスタンスが作成できます。

```
// 現在時刻でDateのインスタンス作成
const now = new Date();
// newを付けないと文字列として帰ってくる
const nowStr = Date();
// 'Sun Sep 06 2020 22:36:08 GMT+0900 (Japan Standard Time)'
```

コンピュータの世界ではUNIX時刻、あるいはUNIX秒、エポック（Epoch）秒、エポック時刻と呼ばれるものがよく使われます。これは1970年1月1日（UTC基準）からの経過時間で時間を表すものです。JavaScriptの中ではミリ秒単位であって秒ではないため、本章ではエポック時刻という名前で統一します。

```
// ミリ秒単位のエポック時刻取得
const now = Date.now();
```

`console.time()`と`console.timeEnd()`でも時間計測ができますが、何かしらの処理の間の時間を撮りたい場合には、`Date.now()`を複数回呼び出すことで、ミリ秒単位で時間が計測できます。ブラウザでは`performance.now()`という高精度タイマーがありましたが、セキュリティの懸念もあって現在は精度が落とされていますので、`Date.now()`とあまり差はないでしょう。

```
const start = Date.now();
//  :
// 時間のかかる処理
//  :
const duration = Date.now() - start;
// 経過時間（ミリ秒）の取得
```

このエポック時刻から `Date` のインスタンスにする場合は `new Date()` の引数にミリ秒単位の時間を入れます。逆に、 `Date()` のインスタンスからエポック時刻を取得するには `valueOf()` メソッドを使います。

```
// 現在の時刻から100秒（10万ミリ秒）前の時刻の取得
const hundredSecAgo = new Date(Date.now() - 100 * 1000);

// エポック時刻取得
const epoch = hundredSecAgo.valueOf();
```

さまざまな時間の情報がありますが、TypeScriptではどれを基準に扱うべきでしょうか？ブラウザはユーザーインタフェースであるため、ユーザーの利用環境のタイムゾーン情報を持っています。しかし、多くのユーザーの情報を同時に扱うサーバーではタイムゾーン情報も含めて扱うのは手間隙がかかります。データベースエンジンによってはタイムゾーン込みの時刻も扱いやすいものもあったりはしますが、シンプルに扱うためには以下の指針で大部分のシステムはまかなえるでしょう。

- クライアントで時刻を取得してサーバーに送信するときは、 `Date().now` などでエポック時刻にしてからサーバーに送信する（タイムゾーン情報なし）
- サーバーでは常にエポック時刻で扱う（ただし、言語によっては秒単位だったり、ミリ秒単位だったり、マイクロ秒単位だったり違いはあるため、そこはルールを決めておきましょう）
- サーバーからフロントに送った段階で `new Date()` などを使って、ローカル時刻化する

特定の日時の `Date` インスタンスの作成

特定の日時インスタンスを作成するには、 `Date()` コンストラクタの引数に数値を設定して作成します。月の数値が1少なく評価される（1月は0）な点に注意が必要です。この日時は現在のタイムゾーンで評価されます。

```
// 2020年9月21日21時10分5秒
// 日本で実行すると日本時間21時（UTCでは9時間前の12時）に
const d = new Date(2020, 8, 21, 21, 10, 5)
```

UTCの時刻から生成したい場合には、`Date.UTC()`関数を使います。これはエポック秒を返すのでこれを`new Date()`に渡すことで、UTC指定の時刻のインスタンスが作成できます。

```
// UTCの2020年9月21日11時10分5秒
// 日本で実行すると日本時間20時（日本時間はUTCは9時間進んでいるように見える）に
const d = new Date(Date.UTC(2020, 8, 21, 11, 10, 5))
```

日付のフォーマット出力

RFC 3393形式にするには、`toISOString()`メソッドを使います。`toString()`だとECMAScriptの仕様書で定められたロケール情報も含む文字列で出力を行ます。後者の場合、ミリ秒単位のデータは丸められてしまいます。

```
const now = new Date()
now.toISOString()
// '2020-09-21T12:38:15.655Z'

now.toString()
// 'Mon Sep 21 2020 21:38:15 GMT+0900 (Japan Standard Time)'
```

短い形式やオリジナルの形式にするには自分でコードを書く必要があります。短く日時を表現しようとする場合のコードは次のようになります。月のみカレンダーの表記と異なって、0が1月になる点に注意してください。

```
const str = `${
    now.getFullYear()
}/${
    String(now.getMonth() + 1).padStart(2, '0')
}/${
    String(now.getDate()).padStart(2, '0')
} ${
    String(now.getHours()).padStart(2, '0')
}:${
    String(now.getMinutes()).padStart(2, '0')
}:${
    String(now.getSeconds()).padStart(2, '0')
}`;
// "2020/09/06 13:55:43"
```

`padStart()`と、テンプレート文字列のおかげで、以前よりははるかに書きやすくなりましたが、Day.jsなどの提供するフォーマット関数を使った方が短く可読性も高くなるでしょう。

日付データの交換

クライアントとサーバーの間ではJSONなどを通じてデータをやりとりします。JSONでデータを転送するときは数値か文字列で表現する必要があります。日付データは既に説明した通りに、ミリ秒か秒のどちらかの数値で交換するのがもっともコード的には少ない配慮で実現できます。しかし、一方で出力された数字の列を見ても、即座に現在の日時を暗算できる人はいないでしょう。可読性という点では文字列を使いたいこともあるでしょう。本節ではデータをやりとりする相手ごとのデータ変換の仕方について詳しく説明していきます。

TypeScript（含むJavaScript同士）

TypeScript（含むJavaScript）であれば、`toISOString()`でも、`toString()`でも、どちらの方式で出力した文字列であってもパースできます。`new Date()`に渡すと`Date`のインスタンスが、`Date.parse()`に渡すと、エポック時刻が帰ってきます。

```
const fromToISOString = new Date(`2020-09-21T12:38:15.655Z`)
// 2020-09-21T12:38:15.655Z

const fromToString = new Date(`Mon Sep 21 2020 21:38:15 GMT+0900 (Japan Standard Time)`)
// 2020-09-21T12:38:15.000Z

const fromToISOStringEpoch = Date.parse(`2020-09-21T12:38:15.655Z`)
// 1600691895655

const fromToStringEpoch = Date.parse(`Mon Sep 21 2020 21:38:15 GMT+0900 (Japan Standard Time)`)
// 1600691895000
```

Goとの交換の場合

Goでは日付のフォーマットが何種類か選べますが、このうち、タイムゾーンの時差が数値で入っているフォーマットはTypeScriptでパース可能です。ナノ秒の情報がミリ秒に丸められてしまいますが、一番精度良く伝達できるのは`time.RFC3339Nano`の出力です。

- `time.RubyDate`
- `time.RFC822Z`
- `time.RFC1123Z`
- `time.RFC3339`
- `time.RFC3339Nano`

GoでRFC3339Nanoで出力

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    fmt.Println(now.Format(time.RFC3339Nano))
    // 2020-09-21T21:35:45.057076+09:00
}
```

TypeScriptでパース

```ts
const receivedFromGo = new Date(`2020-09-21T21:35:45.057076+09:00`)
// 2020-09-21T12:35:45.057Z
```

他の言語に出力する場合、`toISOString()`が無難でしょう。Goは`time.RFC3339`か、`time.RFC3339Nano`を使ってパースできます。結果はどちらも同じです。

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t, err := time.Parse(time.RFC3339, "2020-09-21T12:38:15.655Z")
    fmt.Println(t)
    // 2020-09-21 12:38:15.655 +0000 UTC
}
```

Pythonとの交換の場合

Pythonで出力する場合は`datetime.datetime.isoformat()`メソッドを使うと良いでしょう。このメソッドはタイムゾーン情報を取り払い、現在のタイムゾーンの表記そのものをフォーマットして出力します。TypeScriptの`new Date()`はUTCであることを前提としてパースするため、出力時はUTCとして出すように心がける必要があります。このUTCで出力された文字列はTypeScriptでパースできます。

```
from datetime import datetime, timezone

# localの場合はastimezone()を呼んでUTCに
localtime = datetime.now()
utctime = localtime.astimezone(timezone.utc)
utctime.isoformat()
# 2020-09-21T13:42:58.279772+00:00

# あるいは、最初からUTCで扱う
utctime = datetime.utcnow()
utctime.isoformat()
# 2020-09-21T13:42:58.279772+00:00
```

パースはやっかいです。Stack Overflowでスレッドが立つぐらいのネタです[1]。Python 3.7からは`fromisoformat()`というクラスメソッドが増えましたが、以前からの`datetime.strptime()`にフォーマット指定を与えた方が高速とのことです。

```
from datetime import datetime

s = '2020-09-21T12:38:15.655Z'

datetime.fromisoformat(s.replace('Z', '+00:00'))
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)

datetime.strptime(s, '%Y-%m-%dT%H:%M:%S.%f%z')
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)
```

いっそのこと、エポック時刻で扱う方法の方がシンプルでしょう。TypeScriptはミリ秒単位で、Pythonは秒単位なので、1000で割ってから渡す必要があるのと、UTCの数値であることを明示する必要があります。

```
from datetime import datetime
datetime.fromtimestamp(1600691895655 / 1000.0, timezone.utc)
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)
```

1 https://stackoverflow.com/questions/127803/how-do-i-parse-an-iso-8601-formatted-date

Javaとの交換の場合(8以降)

Java8から標準になったクラス群[2]を使ってTypeScriptとの交換を行ってみます。ここで紹介するコードはJava8以降で動作するはずです。このサンプルはJava 11で検証しています。

`Date.toISOString()`と同等の出力は`DateTimeFormatter`で作成できます。UTCのゾーンになるようにインスタンスを作成してから`format()`メソッドを使って変換します。

```
import java.time.Instant;
import java.time.ZonedDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;

class ParseTest {
    public static void main(String[ ] args) {
        var RFC3339_FORMAT = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'");

        var l = ZonedDateTime.now();
        var isoString = l.withZoneSameInstant(ZoneOffset.UTC).format(RFC3339_FORMAT);
        System.out.println(isoString);
        // "2020-09-23T13:55:53.780Z"
     }
}
```

この文字列はTypeScriptでパースできます。

TypeScriptで生成した文字列のパースには前述の `DateTimeFormatter` も使えますが、それ以外には `java.time.Instant` が使えます。これはある時点での時刻を表すクラスです。TypeScriptの `Date.toISOString()` の出力する文字列をパースできます。実際に日時の操作を行う `LocalDateTime` や `ZonedDateTime` へも、ここから変換できます。次のサンプルは `Instant` でパースし、 `ZonedDateTime` に変換しています。

```
import java.time.Instant;
import java.time.ZonedDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;

class ParseTest {
    public static void main(String[ ] args) {
        var i = Instant.parse("2020-09-23T14:06:11.027Z");
        var l = ZonedDateTime.ofInstant(i, ZoneId.systemDefault());

        var f = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        System.out.println(l.format(f));
    }
}
```

2 Java8の日時APIはとりあえずこれだけ覚えとけ

1時間後、1日後、1ヶ月後、1年後の日時の取得

バッチの次回実行時間の計算や、以前のバッチの間に含まれる情報のフィルタリングのために、1時間後や1日後、1ヶ月後、あるいは1時間前や1日前といった日時の演算が必要になることがあります。

現在時刻の`Date`インスタンスを作成し、現在の情報を元に、期待する値のインクリメントを行うだけです。1時間後を計算するときに、12月31日の23時30分だから年月日をインクリメントしなければならない、今年の2月は閏年だから1年後の計算の日数が変わる、みたいなことは考慮する必要はなく、そのような演算はJavaScriptが行います。

set系メソッドは値を変更してしまいますし、メソッドが返すのは`Date`インスタンスではなく、エポック秒なので、必要に応じて新しいインスタンスを作りましょう。

```
const now = new Date();

// 1時間後
const oneHourLater = new Date();
oneHourLater.setHours(now.getHours() + 1);

// 1日後
const oneDayLater = new Date();
oneDayLater.setDate(now.getDate() + 1);

// 1ヶ月後
const oneMonthLater = new Date();
oneMonthLater.setMonth(now.getMonth() + 1);

// 1年後
const oneYearLater = new Date();
oneYearLater.setFullYear(now.getFullYear() + 1);
```

同一日時かどうかの比較

2つの日時が同じ日かどうか確認したいことがあります。たとえば、チケットの日時が今日かどうか、といった比較です。JavaScriptが提供しているのは、日時がセットになった`Date`型のみです。他の言語だと、日付だけを扱うクラス、時間だけを扱うクラス、日時を扱うクラスを別に用意しているものもありますが、JavaScriptは1つだけです。日付だけを扱いたいケースでも`Date`型を使う必要があります。

`Date`型で日付だけを扱う方法で簡単なのは時間部分をすべて0に正規化してしまう方法です。`setHours()`に0を4つ設定すると、時、分、秒、ミリ秒のすべてがゼロになります。また、この関数を実行するとエポック時刻が帰ってくるので、これを比較するのがもっとも簡単でしょう。ただし、このメソッドはその日付を変更してしまうため、変更したくない場合は新しいインスタンスを作ってからこのメソッドを呼ぶと良いでしょう。

この0時は現在のタイムゾーンでの日時になります。もし、UTCで計算したい場合は、`setUTCHours()`を使います。

```
// 今日の0時0分0秒のエポック時刻
const today = (new Date()).setHours(0, 0, 0, 0)

// 比較したい日時
const someDate: Date

// 同じ日ならtrue
const isSameDay = (new Date(someDate)).setHours(0, 0, 0, 0) === today;
```

日時ではなく時間だけを扱う

前節では日付だけを扱いましたが、時間だけを扱いたいケースもあるでしょう。こちらも、年月日を同一の日付、例えば、2000年1月1日などの特定の日付に固定してしまえば扱えます。例えば、サーバーではUTCの0時0分からの経過秒で扱うユースケースを考えてみます。例えば、毎日の会議が日本時間で10時であったとします。UTCで朝1時、日本時間の10時を表現するには、3600になりますね。UTCの2000年1月1日0時のエポック秒は946,684,800になりますので、これに足して`Date`インスタンスを作れば良いことがわかります。

このクラスはローカルタイムゾーンの時間を計算してくれるので、`setHours()`で時間を取得すると、すでにローカルタイムゾーンの時間になります。

```
// 基準日で時間だけオフセットしたDateインスタンス
const meetingTime = new Date((946684800 + 3600) * 1000); // JSの時間はミリ秒なので1000倍する
console.log(`${meetingTime.getHours()}:${meetingTime.getMinutes()}`);
// 10:00
```

今日のミーティングの日時情報を得るにはは、今日の日付を設定すればOKです。

```
// 今日のミーティング時間を計算するには、今日の年月日を設定する
const now = new Date();
const todayMeeting = new Date(meetingTime);
todayMeeting.setFullYear(now.getFullYear(), now.getMonth(), now.getDate());
```

Date型のタイムゾーンの制約

`Date`は「タイムゾーンが扱える」ということは何度か説明していますが、任意の日付で自由にタイムゾーンが扱えるわけではありません。標準の国際化の機能を使えば、ニューヨークの今の時間は？というのを数値で取得、みたいなことを計算する能力はあるのですが、それが使いやすいインタフェースを提供していません。一旦文字列にしてからパースするしかありません。

```
const now = new Date();
console.log(now.toLocaleString('en-US', { timeZone: 'America/Los_Angeles' }));
// '12/8/2020, 2:19:55 AM'
```

コンピュータでタイムゾーンで扱う名前はIANAのデータベースが一次情報になります。

- https://www.iana.org/time-zones

ただし、これは.tar.gzなどでの提供のみなので、一覧をみたい場合はWikipediaなどの方が見やすいでしょう。

- https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

`Date`のみで扱うのは、UTCと現在時刻ぐらいにしておいた方が無難です。最初に紹介した各種ライブラリなどは、タイムゾーンを扱う機能も持っているので、そちらを利用する方が良いでしょう。ただし、標準の国際化機能を利用したもの以外、タイムゾーンはかなり大きなデータファイルを必要とするので、ビルドしたJavaScriptのファイルサイズは大きくなる可能性があります。

RegExp

正規表現のためのクラスです。TypeScriptでは組み込みで正規表現があります。正規表現は`string`と一緒に使うクラスで、ちょっと賢い検索を実現します。

正規表現はいくつかの要素を組み合わせた文字列のパターン（ルール）を組み合わせて作ります。このルールを活用することで、「電話番号にマッチする」といった単なる比較以上の比較が実現できます。かなり複雑なパターンも記述できますが、TypeScriptにおいてはあまり活躍の場がありません。ユーザー入力のチェックや設定ファイルぐらいでしょう。

正規表現の評価は、入力の文字列とパターンで行われます。文字列を探索し、パターンにマッチしているかどうかをみていきます。たいてい、複数の文字数の文字列を評価することになります。途中でマッチしない文字が出てきたらそこで評価終了となり、マッチしなかった、という結果が返ります。

正規表現はリテラルで記述できます。

```
const input = "03-1234-5678";

if input.match(/\d{2,3}-\d{3,4}-\d{4}/) {
    console.log("電話番号です");
}
```

使い方

正規表現は特定の目的を持って実行されます。検索であったり、ルールに従って分割だったり

- `文字列.match(正規表現)`
- `文字列.matchAll(正規表現)`
- `文字列.search(正規表現)`
- `文字列.replace(正規表現, 置き換える文字列)`
- `文字列.split(正規表現)`

正規表現側にも

- `正規表現.exec(文字列)`
- `正規表現.test(文字列)`

パターンのルール

正規表現の構成要素は主にこの3つです。

- 文字種
- 繰り返し
- グループ化（キャプチャ）

文字種は、2つの書き方があります。これは一文字を表します。文字クラスが柔軟な文字のルールが記述できます。このルールに従ったいずれかの文字が来たらマッチします。

- 普通の文字列（`a`など）
- 文字クラス（`[abc]`、`[a-zA-Z0-9]`、`[^x]`、`\s`、`.`など）

`[abc]`は、このカッコの中のどれかの文字にマッチします。これだけあればどんなルールも記述できますが、アルファベット全部とかになると正規表現が随分と長くなってしまいます。短く書くためのルールがいくつかあります。`[a-z]`は、aからzの小文字のアルファベットすべてにマッチします。上記のサンプルの`a-zA-Z0-9`は、すべての大文字小文字数字にマッチします。`^`をつけると、「これら以外の文字」と言うルールになります。また、`\s`（改行やタブなどのスペースにマッチ）、`\w`（英数字にマッチ）、`\d`（数字にマッチ）といったさらなる短縮系もあります。それぞれ、大文字にすると否定（`\S`は改行タブスペース以外にマッチ）といったルールになります。また、`.`はすべての文字にマッチします。

文字ではないですが、先頭の`^`は文字列の先頭、最後の`$`は末尾となります。`/abc/`は`----abc-----`にもマッチしますが、余計な文字が前後に付かないことを保証するには`/^abc$/`とします。

文字のルールが記述できたので、それを並べれば文章にマッチします。例えば、`/abc/`は、`"abc"`の文字列にマッチします。これを柔軟にしていくメタ文字が何種類か提供されています。こちらも、文字クラスと同様に、冗長な書き方と、短縮系が提供されています。

- 前のルールをn回繰り返す（`{n}`）
- 前のルールをn回以上繰り返す（`{n,}`）
- 前のルールをn〜m回繰り返す（`{n,m}`）
- 前のルールがあってもなくても良い（`?`）
- 前のルールを0回以上繰り返す（`*`）
- 前のルールを1回以上繰り返す（`+`）

TypeScriptの環境設定ではファイルの拡張子のルールを設定することがよくありますが、`.ts`でも`.tsx`でもマッチする書き方は`.tsx?`です。最後の一文字を無視しても良い、というルールになります。

日本の郵便番号は数字3桁+数字4桁です。これをルールにすると次のようになります。

```
const postalCode = "141-0032";
if postalCode.match(/\d{3}-\d{4}/ {
    console.log("郵便番号です");
}
```

これはもちろん、次のように書いてもマッチしますが、なるべく抜け漏れのない短い、意図が伝わり安いパターン記述が良いでしょう。

- `/[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/`（上記と等価だが、長すぎる）
- `/\d+-\d+/`（ただし、これでは文字数が間違っても通ってしまうのでNG）
- `/\w{3}-\w{4}/`（ただし、これでは数字以外に文字列もマッチしてしまう）
- `/.+/`（すべての入力にマッチしてしまってテストにならない）

最後にグルーピングとキャプチャです。上記のルールだけでもマッチしているかどうかの条件にはだいたい使えます。しかし、もっと柔軟にしたいことがあります。例えば、固定電話の番号は同じ10桁でも、都道府県によってルールがいくつかあります。

- 2桁-4桁-4桁
- 3桁-3桁-4桁
- 4桁-2桁-4桁
- 5桁-1桁-4桁

正規表現ではこれまでのルールで作ったパターンを`( )`でグループ化し、グループ単位で繰り返し（ルールは前述のものと同様）や、どちらかにマッチすればOK（`|`）　といったことが可能です。

上記の電話番号はそれぞれ、次のようにルール化できます。ついでに、先頭がゼロというのもパターンのルールに入れます。

- `/^0\d-\d{4}-\d{4}$/`
- `/^0\d{2}-\d{3}-\d{4}$/`
- `/^0\d{3}-\d{2}-\d{4}$/`
- `/^0\d{4}-\d-\d{4}$/`

これらをグループ化し、OR化してあげれば、すべての電話番号にマッチするパターンが作れます。一見複雑に見えますが、グループとORを分解すると解読しやすくなります。

```
``/(^0\d-\d{4}-\d{4}$)|(^0\d{2}-\d{3}-\d{4}$)|(^0\d{3}-\d{2}-\d{4}$)|(^0\d{4}-\d-\d{4}$)/``
```

2桁の一番上のものは、2桁目は3か4か5かなので、正確には`/0[3-5]/`とできると思います。正規表現だけでどこまでビジネスロジックを入れ込むかは設計次第です。正規表現だとマッチしたかどうかの0/1でしか判断できません。2桁だが09みたいな変則番号が来たときにエラーを通知する場合は、2桁で広くマッチさせておいて、次に桁をみるエラーチェックを行うなど、2段構成にする必要があるでしょう。

キャプチャ

グループ化にはもう一つおまけがあります。`match()`の場合、どのカッコごとにマッチした結果を保存して返してくれます。

```
> const tel = /(^0\d-\d{4}-\d{4}$)|(^0\d{2}-\d{3}-\d{4}$)|(^0\d{3}-\d{2}-\d{4}$)|(^0\d{4}-\d-\d{4}$)/
> "042-234-1234".match(tel)
[
  '042-234-1234',
  undefined,
  '042-234-1234',
  undefined,
  undefined,
  index: 0,
  input: '042-234-1234',
  groups: undefined
]
```

レスポンスは一部フィールドを持った配列です。もしどれにもマッチしなければ`null`が返ります。

- 0番目: 全体のマッチした結果
- n番目: n番目のグループのマッチ結果（マッチしなければ `undefined`）
- `index`: 入力文字列の何番目の文字列からマッチしたか（マッチするまでに何文字読み飛ばしたか）
- `input`: 正規表現と比較した入力値
- `groups`: 名前付きグループの場合、名前ごとのマッチ結果

グループはネストさせることもできますが、正規表現のパターンのスタートの位置で機械的にインデックスが割り振られます。グループは番号でアクセスもできますが、ECMAScript 2018以降であれば名前をつけることもできます。電話番号は先頭から、それぞれ次のような分類になっています。

- 市外局番: Area Code
- 市内局番: Message Area
- 加入者番号: Subscriber Number

`(パターン)`を`(?<名前>パターン)`と書き換えると名前が付きます。

```
``/^(?<AC>0\d{1})-(?<MA>\d{4})-(?<SA>\d{4})$/``
```

これで、名前でマッチした結果にアクセスできます。

```
> const match = "01-2345-6789".match(/^(?<AC>0\d{1})-(?<MA>\d{4})-(?<SA>\d{4})$/)
> const { AC, MA, SA } = match.groups
> AC
'01'
> MA
'2345'
> SA
'6789'
```

JSON

JavaScriptで外部のAPIのやりとりなどで一番使うのはこのJSONでしょう。サーバーから帰ってくるJSON形式の文字列を、プログラム中で扱いやすいJavaScriptやTypeScriptのプリミティブなデータ型などに変換します。

基本は`parse()`と`stringify()`を呼ぶだけですので使い方に迷うことはないでしょう。

JSONのパース

```
// aはany型
const a = JSON.parse(‘{"name": "John Cleese"}`);

// asで型情報を付与できる。
const p = JSON.parse(‘{"name": "Terry Gilliam"}`) as Person;
```

`parse()` は `any` 型になります。 `as` で何かしらの型にキャストする必要がありますが、このパースは当然実行時に行われます。 `as` はコンパイル時の型チェックのためのものなので、実際に実行時にどのような型が来るかは推測でしかありません。このJSONのパース周りは、コンパイルは通ったのに、想定した型と違う情報がきたためにエラーになる、ということが一番起きやすいポイントです。 `as` は一見その型と同等であると保証しているように見せてしまいますが、要素の有無のチェックなどは動的な言語を扱っている意識を忘れないで行いましょう。

型定義をする必要がないかと言えば、明らかなスペルチェックは見つけられますし、補完もされるので、可能なら定義しておく方が良いでしょう。

`SyntaxError` 例外

この関数は、JSONの文法違反があると `SyntaxError` 例外を投げます。TypeScriptで例外を扱うことは稀ですが、キャッチしないと何か操作したのに処理が行われていないように見えるなどの不具合になります。大々的にエラーダイアログが出たりはしないので気付きにくかったりしますが、コンソールには出力されていたりします。パースするときには `try` でくくって、エラーが出たときにはダミーの値を入れるなり、自分でエラーダイアログを表示するなり、対処しましょう。

```
let person: Person
try {
    person = JSON.parse(input);
} catch (e: unknown) {
    // fallback
    person = { name: "Eric Idle" };
}
```

この関数は `fetch()` の中でも使われています。例外の発生についてもまったく同じです。

```
const res = await fetch("/api/person");
// 本当はres.okで通信が成功したかチェックが必要！
person = { name: "Michael Palin" };
```

HTTPのAPIの場合、エラーがあると、 `Forbidden` などのステータスコードの文字列がレスポンスとして帰ってくることがあり、これをJSONにパースしようとすると次のような例外が発生します。

```
Uncaught SyntaxError: Unexpected token F in JSON at position 0
```

ほとんどの場合はステータスコードのチェックで防げますが、try文を使うとさらに安心です。

JSONとコメント

JSONの厳格な文法（json.org）にはコメントはありません。しかし、入れたくなることがあります。

TypeScriptの設定ファイルの`tsconfig.json`や、VSCodeの設定ファイルなどはコメントが入れられます。前者はTypeScriptパーサーを流用したパーサーを使っていますので、TypeScriptと同じコメントが利用できます。後者はJSON with Comments(.jsonc)というモードを持っています。標準の`JSON.parse()`は仕様に従っているのでコメントがあるとエラーになります。次のようなライブラリを使う必要があるでしょう。

- strip-json-comments

設定ファイルとして使って読み手が自分自身だけならいいのですが、サーバーなどとのデータ交換用にJSONを使う場合、読み手がコメントを無視して読んでくれないかぎりはコメントを使うべきではありません。その他の方法としては、JSON Schemaの仕様にある`$comment`キーを使うという妥協案もあります。

```
{
   "$comment": "コメントです",
   "location": "鳥貴族"
}
```

文字列化

文字列化はJavaScriptのオブジェクト、配列、数値、文字列、boolean型などの値を渡すと、それを安全に伝送できる文字列にしてくれます。

JSON形式に文字列化

```
// bは文字列
const b = JSON.stringify({person: "Graham Chapman")
// '{"person":"Graham Chapman"}'
```

`stringify()`には2つ追加の引数があります。1つは置換関数、もう1つはインデントです。

このうち置換関数は`function(key: any, value: any): any`な関数で、キーと値を見て、実際に出力する値を決めますが、あまり使い勝手の良いものではありません。階層があったり、配列で同型のオブジェクトがあったりして、仮に同名のキーがあっても、渡される情報だけではどちらの値か区別できなかったりします。`bigint`などの変換できない型の場合はreplacerが実行される前にエラーになってしまうため、この関数で出力できるように文字列にするといった使い方もできません。事前に出力可能なオブジェクト・配列・プリミティブだけのきれいな情報に変換しておくべきです。そのため、忘れてしまっても構いません。

インデント

デフォルトではインデントがなく、文字数最小で出力されます。インデントに数値、あるいは`"  "`といった文字列を渡すことでインデントが行われて見やすくなります。ただし、Node.jsやブラウザの`console.log()`の場合はインデントを設定しなくても見やすく表示してくれるため、整形してファイル出力したい場合以外は使う必要はないと思います。

インデント

```
const montyPython = {
    members: [
        "John Cleese",
        "Terry Gilliam",
        "Eric Idle",
        "Michael Palin",
        "Graham Chapman",
        "Terry Jones"
    ],
};

console.log(JSON.stringify(montyPython));
// {"members":["John Cleese","Terry Gilliam","Eric Idle","Michael Palin","Graham Chapman","Terry
Jones"]}

console.log(JSON.stringify(montyPython, null, 2));
// {
//   "members": [
//     "John Cleese",
//     "Terry Gilliam",
//     "Eric Idle",
//     "Michael Palin",
//     "Graham Chapman",
//     "Terry Jones"
//   ]
// }
```

JSONとデータロス

JSONは言語をまたいで使われるデータのシリアライズのための仕組みです。元はJavaScriptのオブジェクト表現をフォーマットにしたものではありますが（JSONは作成されたのではなく、発見されたと言われています）、いくつかTypeScriptと違うところもあります。

JSONは単純な木構造であり、TypeScriptのメモリ上の表現のすべてを表現できるわけではありません。例えば、親が子を、子が親を参照しているような循環構造の場合、うまく文字列化できず、エラーになります。事前に変換する関数を使って、子から親方向の参照を切った新しいオブジェクト階層を作るなどして単方向の参照になるようにします。

循環参照があるとTypeError

```
const person = {name: "Terry Jones"};
const group = {name: "Pythons", member: [person]};
person.group = group; // お互いに参照しあっているため、循環参照になる

JSON.stringify(group)
// Uncaught TypeError: Converting circular structure to JSON
```

また、JSONが扱えるデータ型はそれほど多くありません。ネイティブで扱える型は以下の6つです。他のものはうまく文字列にならなかったり、なったとしても再度パースしたときにもとの型が復元できないことがあります。

- オブジェクト
- 配列
- 文字列
- 数値
- boolean型
- null

例えば `undefined` の場合はそのキーがなかったことになります。クラスの場合はメンバーフィールドのみのオブジェクトになります。一応データとしては復元できますが、戻すときには単なるオブジェクトで、クラスのインスタンスではなくなります。各クラスに、オブジェクトからインスタンスを復元するファクトリーメソッドを用意してあげる必要があるでしょう。日付は文字列になります。これも、事前に `valueOf()` で数値化しても良いでしょう。 `Map()` などは値が完全に失われたインスタンスになるため、注意が必要です。

クラス、日付

```
> class C { constructor() { this.a = 1; this.b = "hello"; } }
> const i = new C();
> JSON.stringify(i);
// '{"a":1,"b":"hello"}'

> JSON.stringify(new Date())
// '"2020-09-15T14:41:37.173Z"'

> const m = new Map([[1, 2], [2, 4], [3, 8]])
// Map(3) { 1 => 2, 2 => 4, 3 => 8 }
> JSON.stringify(m);
// '{}'
```

`URL` と `URLSearchParams`

正規表現とJSONを紹介しました。正規表現はユーザーの入力のチェックにたまに使います。JSONはサーバーとのデータのやりとりのフォーマットとしてよく使います。もう一つ、サーバーとのやり取りで利用するのが `URL` や `URLSearchParams` です。

サーバーにリクエストを送るときにデータを詰め込む箱としては次の3つがあります。

- パス
- メソッド
- ボディ

このうち、主にパスに使うのが本節で紹介する `URL` と `URLSearchParams` です。TypeScriptの元になっているECMAScriptには含まれないものですが、ブラウザには備わっていますし、Node.jsにも追加されました。Node.jsはもともと別のURL解析関数を持っていましたが、そちらは非推奨になり、現在はこちらのブラウザ互換のクラスが推奨になっています。

`URL`

使い方は簡単で、コンストラクタにパスを入れると、パスのそれぞれの構成要素（プロトコルやホスト名、パス）などに分解します。

```
> u = new URL("https://developer.mozilla.org/en-US/docs/Web/API/URL#Methods")
URL {
    href: 'https://developer.mozilla.org/en-US/docs/Web/API/URL#Methods',
    origin: 'https://developer.mozilla.org',
    protocol: 'https:',
    username: '',
    password: '',
    host: 'developer.mozilla.org',
    hostname: 'developer.mozilla.org',
    port: '',
    pathname: '/en-US/docs/Web/API/URL',
    search: '',
    searchParams: URLSearchParams {},
    hash: '#Methods'
 }
```

一部を書き換えてtoString()を呼ぶことでURLが作成できます。

```
> u.pathname = "/en-US/docs/Web/API/URL/createObjectURL"
> u.toString()
'https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL#Methods'
```

この程度であればテンプレート文字列を使うなどしても簡単ですが、この手の文字列を組み立てるクラスは積極的に使うべきです。SQLインジェクションなどのセキュリティホールは自分で命令を組み立てたときに、エスケープし忘れて想定しない命令が差し込まれてしまうことで発生します。URLも、何かを呼び出すときの鍵ですので、想定外のURLができてしまうのは問題があります。

URLにはパスの最後にクエリーパラメータが付きますが、そこをsよりするのが`URLSearchParams`です。上記の例ではクエリーがなかったので空となっていますが、URLにクエリーがあれば、ここの部分にパラメータが保存されます。

caniuse.comで検索したURLを渡すと次のようになります。

```
> c = new URL("https://caniuse.com/?search=url%20search%20params")
URL {
    href: 'https://caniuse.com/?search=url%20search%20params',
    origin: 'https://caniuse.com',
    searchParams: URLSearchParams { 'search' => 'url search params' },
}
```

URLSearchParams

`URLSearchParams`はURLの一部としても利用されますが、クエリー部分だけを渡すことで、`URLSearchParams`を単独で利用できます。

```
> q = new URLSearchParams("?search=url%20search%20params")
URLSearchParams { 'search' => 'url search params' }
```

このクラスは、`Map`に似ていますが、同一のキーに複数の値を持たせることができます。

メソッド	説明
`get(key: string): string`	値を取得。複数ある場合は先頭のみ。
`getAll(key: string): string[]`	値を配列で取得。
`append(key: string, value: string)`	値を設定（複数共存可能）
`set(key: value, value: string)`	値を設定（上書きして1つだけ残す）
`delete(key: value)`	値を削除
`has(key: value): boolean`	キーが存在するか確認
`toString(): string`	文字列を生成

イテレータプロトコルをサポートしており、キーだけ(`keys()`)、値だけ(`values()`)、キーと値のペア(本体を渡す)の3通りのfor ofループが可能です。

`URLSearchParams`は基本的にURLのクエリーパラメータ部分の組み立てに使いますが、HTMLのフォームで、デフォルトの`enctype`（`application/x-www-form-urlencoded`）とも互換性があります。

HTMLのフォームを使って送信する変わりに`fetch()`を使ってTypeScriptでリクエストを送るときの組み立てにも使えますし、Node.jsなどでサーバーを実装した場合に、ポストされたボディをパースするのにも利用できます。こちらもURLと同様に、セキュリティホールを作らないためにも、なるべく利用するようにしましょう。

なお、この`URLSearchParams`がキーと値をそれぞれエスケープするのと同じロジックは`encodeURIComponent()`と`decodeURIComponent()`という関数でも利用できます。

厳密にはURLのクエリーとフォームは少し異なり、フォームとクエリーの場合はスペースはプラス（`+`）で、パスの一部は`%20`だったりしますが、どちらも入れ替えてデコードすると正しく戻るため、そこまで厳密にみなくても大丈夫です。`URLSearchParams`はプラスにしますが、`encodeURIComponent()`は`%20`にします。

時間のための関数

`setTimeout()`

`setInterval()`

`requestAnimationFrame()`