```
                "-p", "C:\\Users\\***\\AppData\\Roaming\\Mozilla\\Firefox\\" +
                      "Profiles\\***"
            ],
            "nightly": [
                "-b", "C:\\Program Files\\Mozilla Firefox Nightly\\" +
                      "firefox.exe"
            ]
        }
    }
```

すると、"default":で始まる部分で指定したオプションが既定となります。また、cfx run -g nightlyと入力することで、"nightly":で始まる部分で指定したオプションが有効となります。

引き続き、[Hack #18]では、Add-on SDKを使って実際に拡張機能を開発する手順を解説します。

HACK #18 拡張機能の開発（基礎編）

Add-on SDKを用いてSDKベース拡張機能を開発しながら、基本的な開発手法を解説します。

Translator JP拡張機能

本稿で実際に開発する拡張機能「Translator JP」は、**図18-1**のようにWebページで英語の文字列を選択して右クリックメニューから[Translate into Japanese]をクリックすると、**図18-2**のように日本語に翻訳して置き換える拡張機能です。

図18-1 Translator JP拡張機能の動作例❶

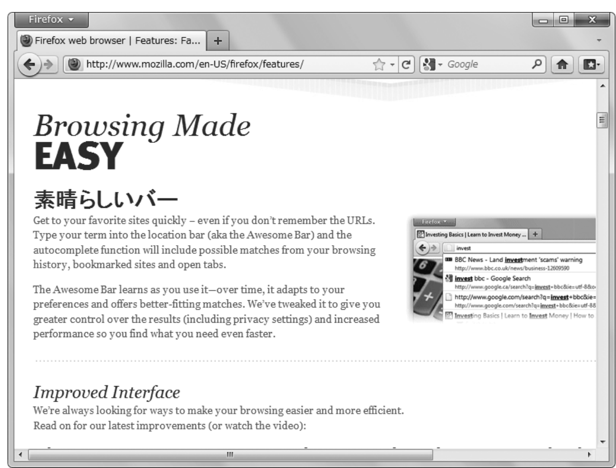

図18-2　Translator JP拡張機能の動作例❷

最小構成のパッケージ

個々の拡張機能を構成するソースファイルを格納したフォルダ全体をパッケージと呼びます。はじめに、**図18-3**に示すとおりにパッケージのルートフォルダ「translator-jp」とその中に各種ソースファイルおよびフォルダを新規作成します。

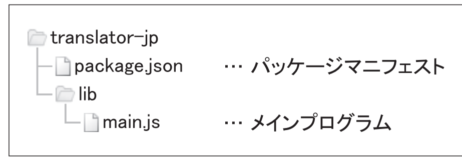

図18-3　パッケージのフォルダ構成

なお、Translator JP拡張機能の完全なソースコードは下記URLから入手可能です。

http://firefoxhacks.org/source.html

パッケージマニフェスト

`package.json`ファイルはパッケージマニフェストと呼ばれる特殊なファイルです。パッケージマニフェストはCommonJSのPackagesの仕様をベースとしており、そのパッケージの名前や作者といったメタ情報を**例18-1**のようにJSON形式で記述します。なお、cfxツールではFirefoxのJavaScriptエンジンと異なり、{}内にコメントを含め

ると文法エラーとなるため、各行の//以下のコメントは記述しないでください。

例18-1　package.json

```
{
    "name": "translator-jp",         // ❶
    "fullName": "Translator JP",     // ❷
    "description": "Translates selection into Japanese.",    // ❸
    "author": "Gomita",    // ❹
    "version": "0.1"       // ❺
}
```

❶ 通常はルートフォルダ名と同一の値を指定します。半角英数字のみで、空白などの混在は不可です。

❷ アドオンマネージャ上で表示される拡張機能の名前です。こちらは空白などの混在が可能です。

❸ 拡張機能の説明です。一文程度で、簡単な説明を記述してください。

❹ 拡張機能の作者です。適宜変更してください。

❺ 拡張機能のバージョンです。パッケージマニフェストでtrailing commaは許可されないため、行末にカンマを入れないように注意してください。

上記以外にも色々なプロパティを指定することができます。詳しくは、Package Specification（https://addons.mozilla.org/en-US/developers/docs/sdk/1.0/dev-guide/addon-development/package-spec.html）を参照してください。

メインプログラム

`lib`フォルダ内の`main.js`ファイルはメインプログラムと呼ばれ、その拡張機能がインストールされた直後に一度だけ実行されるファイルです。今回は試しに**例18-2**のような内容を記述します。

例18-2　main.js

```
exports.main = function(){
    console.log("Translator JP");
};
```

`exports.main = ...`という書き方に注目してください。メインプログラムはそれ自体がCommonJS形式のモジュール（http://www.commonjs.org/specs/modules/1.0/）となっており、拡張機能インストール直後にメインプログラムの中の`main`メソッドがエントリポイントとなって実行開始されます。

console.logという関数はAdd-on SDKが定義するグローバルオブジェクトの1つで、コマンドプロンプトまたはエラーコンソールへデバッグ用の文字列を出力します。Add-on SDKのプログラム内で使用可能なグローバルオブジェクトには**表18-1**にあげた3種類があります。一般的なWebページ上で動作するJavaScriptとは異なり、windowやdocumentといったグローバルオブジェクトは存在しないことに注意してください。

表18-1　グローバルオブジェクトの分類

種類	概要
JavaScript Globals	JavaScript 1.8.1の仕様で定められたグローバルオブジェクト。String、Array、Date、JSONなど
CommonJS Globals	CommonJS Module 1.0の仕様で定められたグローバルオブジェクト。require、exports、defineの3つ
SDK Globals	Add-on SDKが定義するグローバルオブジェクト。console.logなど

動作確認

ここまでで一度Translator JP拡張機能の動作確認をします。[Hack #17]で解説したようにcfx runコマンドでTranslator JP拡張機能を実行してください。初回実行時はAdd-on SDKが個々の拡張機能を識別するためのIDであるプログラムIDが自動的に付与され、パッケージマニフェストへ"id"プロパティが追加されます。

再度cfx runコマンドを実行し、Firefox起動後にコマンドプロンプトへinfo: Translator JPと出力されることを確認してください。なお、現段階ではTranslator JP拡張機能はアドオンマネージャ上で「Test App 0.1」と表示されます。パッケージマニフェストへ記述した内容は後述のインストーラ作成の段階で有効となることに注意してください。

使用するモジュールの一覧

ここからはAdd-on SDKが提供するaddon-kitライブラリの各種モジュールを駆使しながら実際にTranslator JPの機能を実装します。はじめに、どの機能をどのモジュールで実装するかを整理して**表18-2**に示します。

表18-2　Translator JPで使用するモジュールの一覧

機能	使用するモジュール
右クリックメニューへの項目追加	context-menuモジュール
Webサービスを使った翻訳	requestモジュール
選択範囲の文字列を置換	selectionモジュール

右クリックメニューへの項目追加

Add-on SDKでWebページ上での右クリックメニューへ項目を追加するには、context-menuモジュールを使います。メインプログラムを**例18-3**のように変更します。

例18-3　main.js

```
const contextMenu = require("context-menu");    // ❶

exports.main = function() {
    contextMenu.Item({    // ❷
        label: "Translate into Japanese",            // ❸
        context: contextMenu.SelectionContext(),    // ❹
    });
};
```

❶ CommonJS Globalsの1つであるrequire関数を用い、context-menuモジュールをインポートします。

❷ context-menuモジュールのItemコンストラクタで右クリックメニュー項目を生成、追加します[†]。

❸ Itemコンストラクタの引数オブジェクトのlabelプロパティへメニュー項目のラベル「Translate into Japanese」を指定します。

❹ Itemコンストラクタの引数オブジェクトのcontextプロパティへ、選択範囲上でのみ項目を表示するオプションを指定します。

動作確認

ソースコード変更後に再度動作確認する際は、Firefoxを終了して再度cfx runコマンドでFirefoxを再起動してください。適当なWebページを開き、文字列を選択した状態で右クリックメニューを開くと、[Translate into Japanese]が追加されていることを確認してください。

アドオンマネージャ上で「Test App 0.1」を一度無効化して再度有効化しても、変更が反映されないことに注意してください。

† 過去のバージョンのJetpack SDKではaddメソッドを使ってメニュー項目を明示的に追加する手順が必要でしたが、Add-on SDKでは単にコンストラクタでメニュー項目を生成するだけで追加されます。

Webページの選択範囲の取得

次に、先ほど追加した右クリックメニューの項目をクリックした際に、Webページの選択範囲の文字列を取得する処理を実装します。Add-on SDKでは将来的なプロセス分離に備えてe10sモデル†を導入しており、拡張機能のメインプログラム側がWebページのDOMへ直接アクセスして選択範囲を取得することができません。拡張機能のメインプログラムからWebページのDOMへアクセスするには、コンテントスクリプトと呼ばれる仕組みを使う必要があります。

コンテントスクリプトは、親分である拡張機能のメインプログラムから生み出された子分のような存在で、メインプログラムとは異なるJavaScriptコンテキストで実行され、なおかつWebページのDOMへのアクセスが可能です。コンテントスクリプトはメインプログラム側のイベント発生を検知し、必要に応じてWebページのDOMへアクセスした後、処理結果をWeb Worker API風のメッセージングモデルでメインプログラムへ送信します（**図18-4**）。

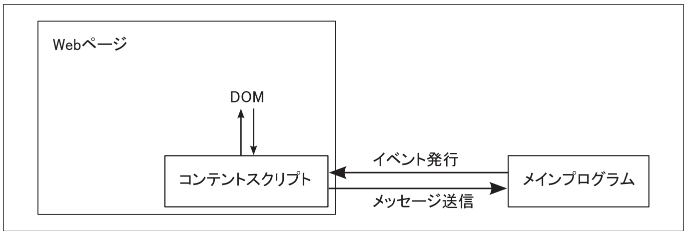

図18-4　メインプログラムとコンテントスクリプトの関係

今回のTranslator JP拡張機能では、具体的に以下のような流れでWebページの選択範囲の取得処理を行います。

1. メインプログラムは、ブラウザにWebページが読み込まれるたびにコンテントスクリプトをロードする
2. 右クリックメニューの項目が選択されると、メインプログラムはコンテントスクリプトへ「click」イベントを送信する
3. コンテントスクリプトは「click」イベントの発生を検知し、WebページのDOMから選択範囲の文字列を取得し、その内容をメッセージとしてメインプログラムへ送信する
4. メインプログラムはコンテントスクリプトからのメッセージを受信し、実際の翻訳処理を実行する

† Electrolysis（e10s）の詳細については**[Hack #34]**を参照してください。

メインプログラムを**例18-4**のように変更します。

例18-4　main.js

```
const contextMenu = require("context-menu");

exports.main = function() {
    contextMenu.Item({
        label: "Translate into Japanese",
        context: contextMenu.SelectionContext(),
        // ❺
        contentScript: "self.on('click', function() {" +
                       "    var sel = window.getSelection().toString();" +
                       "    self.postMessage(sel);" +
                       "});",
        // ❻
        onMessage: function(sel) {
            console.log("selection: " + sel);
            // [ToDo] 翻訳処理を実装
        }
    });
};
```

❺ Itemコンストラクタの引数オブジェクトへcontentScriptプロパティを追加し、Webページへコンテントスクリプトをロードします。コンテントスクリプトでは、後述するEventEmitterフレームワークのAPIの1つであるself.onによってclickイベントを監視し、clickイベント発生時にWebページのDOM（window.getSelection()メソッド）から選択範囲の文字列を取得します。さらに、取得した結果をself.postMessageによってメインプログラムへ送信します。なお、self.postMessageは異なるプロセス間でのデータ送受信を前提としており、送信可能なデータは文字列やJSON化可能なオブジェクトに限られます。

❻ メインプログラムでは、コンテントスクリプトからのメッセージ受信時のコールバック処理として、引数として渡された選択範囲の文字列を取得し、翻訳処理を実行します。現時点では、ひとまずconsole.logでコマンドプロンプトへ出力します。

動作確認

右クリックメニューから［Translate into Japanese］をクリック後、Webページの選択範囲の文字列がコマンドプロンプトへ出力されることを確認してください。

EventEmitterフレームワーク

Webアプリの開発において、ある要素上で発生するイベントに対してリスナを追加する際には、DOMのAPIであるaddEventListenerメソッドを使うのが一般的です。これに対して、Add-on SDKではon関数でイベントリスナを追加するAPIが随所に見られます。このようなイベント駆動型のプログラミングモデルを、EventEmitterフレームワークと呼びます。

EventEmitterフレームワークのもう1つの特徴として、DOMのAPIではイベントリスナへの引数としてeventオブジェクトが渡されるのに対し、EventEmitterフレームワークではイベントリスナへの引数としてeventオブジェクト以外の任意のオブジェクトを複数渡すことが可能です（**表18-3**）。

表18-3　DOMのAPIとEventEmitterフレームワークの比較

処理	DOMのAPI	EventEmitterフレームワーク
イベントリスナの追加	*element*.addEventListener(*type*, *listener*, *useCapture*)	*item*.on(*type*, *listener*)
イベントリスナの削除	*element*.removeEventListener(*type*, *listener*, *useCapture*)	*item*.removeListener(*type*, *listener*)
イベントリスナへ渡される引数	eventオブジェクト	モジュールのAPIによって様々
イベントの発行	document.createEventでeventオブジェクトを生成し、event.initEventで発行	EventEmitterのプライベートメソッド_emit(*type*, リスナへ渡す引数)で発行

Webサービスを使った翻訳

次に、先ほど未実装だった翻訳処理を実装します。翻訳にはGoogle Translate API（http://code.google.com/intl/ja/apis/language/translate/v1/using_rest_translate.html）というWebサービスを用います。Google Translate APIはREST型のシンプルなAPIで、レスポンスはJSON形式です。

Google Translate APIの入出力例

- **リクエスト**

```
http://ajax.googleapis.com/ajax/services/language/trans
late?v=1.0&q=hello&langpair=|ja
```

- **レスポンス**

```
{"responseData": {"translatedText":"こんにちは","dete
ctedSourceLanguage":"en"}, "responseDetails": null,
```

```
    "responseStatus": 200}
```

Add-on SDKでWebサービスへリクエストを送信するには、requestモジュールを使用します。メインプログラムを**例18-5**のように変更します。

例18-5　main.js

```
const contextMenu = require("context-menu");
const request = require("request");    // ❼

exports.main = function() {
    contextMenu.Item({
        …
        onMessage: function(sel) {
            var req = request.Request({    // ❽
                url: "http://ajax.googleapis.com/ajax/services/" +
                     language/translate",    // ❾
                content: { v: "1.0", q: sel, langpair: "|ja" },    // ❿
                onComplete: function(response) {    // ⓫
                    console.log(response.json.toSource());
                    console.log(response.json.responseData.
                                translatedText);
                    // [ToDo] Webページの選択範囲を置換する処理
                }
            });
            req.get();    // ⓬
        }
    });
};
```

❼ requestモジュールをインポートします。

❽ requestモジュールのRequestコンストラクタで、Webサーバと通信するためのrequestオブジェクトを生成します。

❾ Requestコンストラクタの引数オブジェクトには、urlプロパティで送信先URLを指定します。

❿ 送信するクエリをcontentプロパティで指定します。今回はパラメータqへ翻訳前の選択範囲の文字列をセットします。

⓫ Webサービスからのレスポンス受信時のコールバック処理をonCompleteプロパティで指定します。onCompleteの引数にはレスポンス内容を表すresponseオブジェクトが渡されます。responseオブジェクトは、そのtextプロパティからレスポンス内容の文字列を取得できるほか、JSON形式文字列であればjsonプロパティからJavaScriptオブジェクトへパースした結果を取得することも可能です。ここではひとまずレスポンス内容をJSONとしてパースした結果全体と、その中の翻訳結果をconsole.logで出力します。

⓬ requestオブジェクトのgetメソッドを実行することで、実際にリクエストを送信します。

選択範囲の文字列を置換

Add-on SDKでWebページの選択範囲を取得／置換するには、selectionモジュールを使用します。以前実装した右クリックメニュー項目クリック時の選択範囲取得の処理では、コンテントスクリプトを使って実装する必要がありましたが、今回はメインプログラムから直接選択範囲を置き換えることができます（例18-6）。

例18-6　main.js

```
const contextMenu = require("context-menu");
const request = require("request");
const selection = require("selection");    // ⓭
               …
      onComplete: function(response) {
          var translated = response.json.responseData.translatedText;
          selection.text = translated;    // ⓮
      }
      …
```

⓭ selectionモジュールをインポートします。

⓮ Webページ中の選択範囲を、Webサービスからのレスポンスから取得した結果に置き換えます。

動作確認

Webページ上で適当な英語の文字列を選択し、右クリックメニューから［Translate into Japanese］をクリックすると、しばらくして選択範囲の文字列が日本語へ翻訳された文字列に置き換わることを確認してください。

HACK #19 addon-kitライブラリ

Add-on SDKの標準ライブラリの1つであるaddon-kitライブラリについて、各モジュールの使い方を解説します。

[Hack #16]で解説したように、Add-on SDKには標準ライブラリとしてaddon-kitとapi-utilsの2つのライブラリが含まれます。このうちaddon-kitライブラリは、SDKベース拡張機能を開発するにあたり基本的な機能を実装するための高レベルなモジュール群です。addon-kitライブラリに含まれるすべてのモジュールをタイプ別に分類すると、表19-1、表19-2、表19-3、表19-4のようになります。