ミングなどでComponents.manager.removeBootstrappedManifestLocation()を使ってアンロードする必要があります。

```
function shutdown(aData, aReason) {
  const IOService = Cc["@mozilla.org/network/io-service;1"]
            .getService(Ci.nsIIOService);
  var manifest = "jar:" + IOService.newFileURI(aData.installPath).spec
                        + "!/chrome.manifest";
  Components.manager.removeBootstrappedManifestLocation(manifest);
}
```

これらの読み込み操作については、Firefoxのアドオンマネージャ側で自動的に行うようにするというバグがすでに登録されており†、Firefox 8正式版または今後のメジャーリリースでは、特に明示的な読み込み操作を行わずともchrome.manifestの内容が読み込まれるようになる可能性があります。

ただ、その場合でも自動的に読み込まれるのはファイル名がchrome.manifestである物だけに限定されると思われますので、それ以外のファイル名でパッケージに含められたマニフェストファイルについては、chrome.manifestからmanifestディレクトリで間接的に読み込むか、もしくは上記の方法で動的に読み込みを行う必要があります。

HACK #34

e10sにおけるプロセス間通信の基本

Firefoxのプロセス間通信機能の基本的な仕組みを解説します。

Electrolysis（e10s）は、ブラウザのUIを実行するプロセスとコンテンツ領域を描画するプロセスを分離する技術です。応答性の向上やマルチコアプロセッサの有効活用、コンテンツ領域のクラッシュからの復帰を容易にするなどの恩恵を得ることを目的としていて、Firefox Mobileで先行的に有効化されており、将来的にはデスクトップ版のFirefoxでも、今後のメジャーアップデートで有効化される予定となっています。

この変更は過去に行われてきたAPIの変更の中でも特に大きな物で、拡張機能に対しても大きな影響を与えます。Add-on SDKはこの変更を見越して開発が行われているため、SDKが提供するAPIに基づいて新規に開発される拡張機能については、特に問題はありません。しかしながら、SDKを使わずに開発されてきた従来からある拡張機能の場合には、e10sに対応するための抜本的な設計変更が必要となる場合があります。例えば以下のようなことはe10sの導入によってできなくなるため、これらの機能に依存している拡張機能はそのままでは動作しなくなります。

† 参照：https://bugzilla.mozilla.org/show_bug.cgi?id=675371

- XULの<browser/>要素からdocShellプロパティ（nsIDocShellインタフェース）にアクセスできなくなる。それに伴い、docShellからたどるすべての機能もアクセスできなくなる（canGoBack、canGoForward、webNavigation、contentDocument、contentWindowなど）
- コンテンツのDOMにcontent.<プロパティ名>でアクセスできなくなる
- コンテンツ領域の中で発生したloadやclickといったDOMのイベントをキャプチャリングフェーズで捕捉できなくなる

では、Chrome領域での操作をコンテンツ領域に反映させたり、コンテンツ領域の中で発生したイベントをChrome領域に反映させたりするにはどうすればよいのでしょうか？ その鍵を握るのが「コンテントスクリプト」と「メッセージマネージャ」です。

コンテントスクリプトとメッセージマネージャ

e10sが有効化された環境では、<browser remote="true"/>のようにremote属性の値がtrueに設定されたインラインフレームの内容が、別プロセスで処理されるようになります。Firefoxのメインプロセスから分離されたコンテンツ領域のプロセスは、内部的には「最小の機能だけを持ったFirefox」のように動作します。

このコンテンツ領域側のプロセスでは、今までのFirefoxと同様に、「フレームの外側の名前空間で動作するChrome権限を持った制御用のスクリプト」と「フレームの内側の名前空間で動作するChrome権限を持たないスクリプト（Webページ内のスクリプト）」の2種類のコードが実行されることになります。またコンテンツ領域側のプロセスの中では、e10sが有効化される前のFirefoxの場合と同じように、Chrome権限を持つスクリプトは自在にWebページの内容にアクセスすることができますし、XPConnectを使ってXPCOMコンポーネントの機能を呼び出すこともできます。この「Chrome権限を持った制御用のスクリプト」は「コンテントスクリプト」と呼ばれます。

メッセージマネージャは、Chrome領域のプロセスのスクリプト（Chromeスクリプト）とコンテントスクリプトの双方に対して以下の機能を提供します。

- Chromeスクリプトから任意の内容のスクリプトをコンテンツ領域のプロセスに読み込ませ、コンテントスクリプトの名前空間で実行させる機能
- Chrome領域とコンテンツ領域のプロセス間での、文字列ベースでのメッセージ送受信機能

Chrome領域のプロセスとコンテンツ領域のプロセスの間での通信は、JSON形式でデータを添付したメッセージによって行われます。そのため、やりとりできる情報はJSON形式で表現可能な内容のみに限られ、DOMのノードやイベントなどのネイティ

ブオブジェクトはプロセス間では一切受け渡せません。メッセージを受け取った側のプロセスに属するメッセージマネージャは、あらかじめ登録されていたリスナにそのメッセージを渡します。e10sではこのような仕組みにより、文字列ベースでのプロセス間通信が実現されます（**図34-1**）。

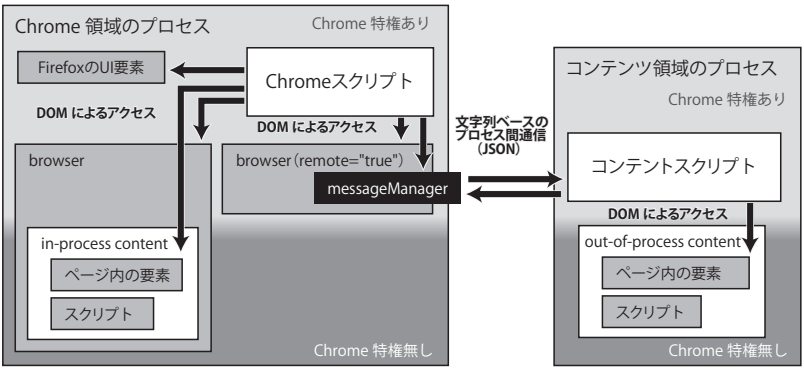

図34-1　Chromeスクリプトとコンテントスクリプト

Firefox 4以後のバージョンは本格的なe10sの導入に先駆けてe10s互換のAPIが利用可能になっており、このAPIを利用するように拡張機能を設計しておくことによって、e10sが完全に有効化された後でも変わらず拡張機能が動作し続けるようにすることができます。具体的にどのようなAPIがあるのかについては、後ほどで詳しく解説します。

JSON形式とは？

JSON（JavaScript Object Notation）は、構造化された情報を扱うための軽量なデータ形式です。JavaScriptのオブジェクトリテラルの記法に基づいており、文字列、数値、真偽値、配列、ハッシュ、nullの組み合わせですべての情報を表現します。例えば以下のような書き方をします。

```
{ "array-value"   : [0, 1, 2],
  "string-value"  : "string",
  "boolean-value" : true,
  "null-value"    : null }
```

JSONはXMLやYAMLほど複雑でなく、単純な文字列としてシリアライズしてメッセージとしてやりとりできる上に、解釈時にはeval()で評価するだけでJavaScriptのハッシュとして構造化された情報を復元できるということで、様々な場面で活用されるようになりました。ただし、現在ではeval()をそのまま使

用するのはセキュリティ面で危険なため、JSONのパースを行うためのライブラリを使うことが推奨されており、Firefoxなどの多くのモダンブラウザではブラウザ自身によって提供されているネイティブのJSON用ユーティリティも利用できます。具体的には、`JSON.stringify()`にJavaScriptのオブジェクトを渡せばその内容がJSON形式の文字列にシリアライズされた結果が返され、`JSON.parse()`にJSON形式の文字列を渡せばその内容が評価されJavaScriptのオブジェクトとして復元された結果が返されます。

in-process contentとout-of-process content

e10s導入後も、必ずすべてのインラインフレームが別プロセス化されるというわけではありません。サイドバーの内容などは今までと同様にChrome領域のプロセスで処理されます。このようにChrome領域のプロセス内で処理されるインラインフレームを「in-process content」、本稿で紹介したように別プロセスで処理されるインラインフレームを「out-of-process content」と呼びます。デスクトップ版のFirefox 4などe10sが本格導入されていない環境では、タブはすべてin-process contentということになります。

両者の違いを意識しないで済むようにするため、in-process contentとout-of-process contentにはどちらも共通のメッセージマネージャのAPIでアクセスすることができます。

コンテンツ領域からChrome領域への通信の例

実際に動作するコードの例を示しながら、メッセージマネージャによるプロセス間通信の具体的な手順を簡単に説明しましょう。以下は、リンク先がPDFファイルであった場合に確認のダイアログを開いて、リンクを開く操作をユーザーが任意にキャンセルできるようにする、という拡張機能のためのコードの例です。

```
// コンテントスクリプトの内容（コンテンツ領域側のプロセス内で実行される）
var script = "data:application/javascript,"+encodeURIComponent(<![CDATA[
      addEventListener("click", function(aEvent) { ❷
        ❹
        var Ci = Components.interfaces;
        var href = aEvent.originalTarget.ownerDocument.evaluate(
                     "ancestor-or-self::*[@href]/@href",
                     aEvent.originalTarget, null,
                     Ci.nsIDOMXPathResult.STRING_TYPE, null
```

```
                ).stringValue || "";
          var results = sendSyncMessage("ContentClick", { link : href }); ❺
        if (results.some(function(aResult) { ❿
              return aResult && aResult.canceled;
            })) {
          aEvent.stopPropagation();
          aEvent.preventDefault();
        }
      }, true);
    ]]>.toString());

// コンテンツ領域から送られたメッセージを受け取るメッセージリスナ
var listener = function(aMessage) {
      ❻
      let isPDF = aMessage.json.link &&
                  aMessage.json.link.match(/¥.pdf$/i); ❼
      let canceled = isPDF && !confirm("PDFを開きますか？"); ❽
      return { canceled : !!canceled }; ❾
    };

var messageManager = gBrowser.selectedBrowser.messageManager;
messageManager.loadFrameScript(script, true); ❶
messageManager.addMessageListener("ContentClick", listener); ❸
```

実際に行われる処理の順番を追いかけてみると、❶まず、任意のコンテントスクリプトを現在のタブのコンテンツ領域のプロセスに読み込ませます。読み込まれたコンテントスクリプトは、❷キャプチャリングフェーズのイベントリスナを登録してユーザーのクリック操作を待ち受けます。また、それと並行して❸Chromeスクリプトが現在のタブのコンテンツ領域にメッセージリスナを登録します。その後、ユーザーがコンテンツ領域をクリックすると、❹コンテントスクリプトの中で登録したイベントリスナがイベントを捕捉して、❺収集した情報を含むメッセージを送信します。❻Chromeスクリプト側のメッセージリスナはこのメッセージを受け取って、❼リンク先がPDFであるかどうかの判別を行い、❽PDFであれば確認のダイアログを表示して、❾結果を含むハッシュを返します。この時点でコンテントスクリプト側に制御が戻り、❿コンテントスクリプトは返ってきた結果に応じてDOMのイベントの伝搬を阻止します（図34-2）。

このとき、❶の読み込み処理および、❺から❻の間と❾から❿の間のメッセージのやりとりがプロセス間の通信となります。プロセス間でやりとりできる情報はJSON形式で表現できるデータに限られるため、リンクのDOMノードではなくリンク先のURI文字列だけを渡していることに注意してください。

図34-2 確認のダイアログが表示されたところ

Chrome領域からコンテンツ領域への通信の例

先ほどの例ではコンテンツ領域での操作をトリガーとして処理の一部をChrome領域に委譲しましたが、逆に、Chrome領域での操作をトリガーとしてコンテンツ領域の側に何らかの処理を行わせることもできます。以下は、「上の階層に移動」ボタンがクリックされたときに、現在コンテンツ領域に表示されているページの1階層上のページに移動する（例えばhttp://www.example.com/about/を閲覧しているときであればhttp://www.example.com/に移動する）、という拡張機能のためのコードの例です。

```
// コンテントスクリプトの内容
var script = "data:application/javascript,"+encodeURIComponent(<![CDATA[
      // Chrome領域から送られたメッセージを受け取るメッセージリスナ
      var listener = function(aMessage) {
            ❼
              let uri = content.location.href.replace(/[^¥/]+¥/?$/, ""); ❽
            if (/^¥w+:(¥/¥/)?[^¥/]+/.test(uri)) {
              if (aMessage.json.newTab) ❾
                content.open(uri); ❿
              else
                content.location.href = uri; ⓫
            }
          };
      addMessageListener("ContentUp", listener); ❷
    ]]>.toString());

// その拡張機能自身が追加した「上の階層に移動」ボタンをクリックしたときに実行される処理
function contentUp(aEvent) {
  ❹
  var newTab = aEvent && ((aEvent.button == 0 && aEvent.ctrlKey) ||
                          aEvent.button == 1); ❺
  gBrowser.selectedBrowser.messageManager
```

```
    .sendAsyncMessage("ContentUp", { newTab : newTab }); ❻
}

gBrowser.selectedBrowser.messageManager
  .loadFrameScript(script, true); ❶
document.getElementById("toolbar-goUp")
  .addEventListener("click", contentUp, false); ❸
```

実際に行われる処理の順番を追いかけてみると、この場合も❶まずはコンテントスクリプトを現在のタブのコンテンツ領域のプロセスに読み込ませます。読み込まれたコンテントスクリプトは❷メッセージリスナを登録して、メッセージが受信されるのを待ち受けます。また、それと並行して❸「上の階層に移動」ボタン（ここではidがtoolbar-goUpである<toolbarbutton/>を想定）にイベントリスナを登録します。ユーザーが「上の階層に移動」ボタンをクリックすると❹ボタンに登録されたイベントリスナが呼ばれ、❺タブを開く操作（中クリックまたはCtrl-クリック）かどうかを判断し、❻収集した情報を含むメッセージを送信します。❼コンテントスクリプト側のメッセージリスナはこのメッセージを受け取って、❽1階層上のURIを求め、❾タブで開くかどうかのフラグを見て❿新しいタブを開くか⓫現在のタブで読み込みを行います。

このとき、❶の読み込み処理および、❻から❼の間のメッセージのやりとりがプロセス間の通信となります。この場合も、プロセス間でやりとりできる情報はJSON形式で表現できるデータに限られるため、DOMのイベントオブジェクトそのものではなく、判断結果の真偽値だけを渡していることに注意してください。

コンテンツ領域からChrome領域への通信の場合とは異なり、Chrome領域からコンテンツ領域への通信は必ず一方通行になります。Chromeスクリプトはコンテントスクリプトのメッセージリスナが返す値を受け取ることはできません。コンテントスクリプトから何らかの情報を返したい場合は、別途コンテントスクリプト側からメッセージを送信して、Chromeスクリプトでそれを受け取る必要があります。

また、コンテントスクリプトからの通信が同期的に処理される（Chromeスクリプト側のメッセージリスナが値を返すまでの間は、コンテントスクリプト側の処理が一旦停止される）のに対して、Chromeスクリプトから送ったメッセージはコンテントスクリプトに常に非同期に処理されます。Chromeスクリプトがメッセージを送っても、その場ですぐにコンテントスクリプトがメッセージを受け取るとは限りません。コンテントスクリプト側の処理が完了してからChromeスクリプト側で次の処理を行いたいという場合には、XMLHttpRequestで読み込みの完了を待ってから次の処理を行うのと同じように、コンテントスクリプトからのメッセージを待ち受けて非同期処理の完了を検知する必要があります。

Firefox Mobileのe10s

デスクトップ版に先駆けてFirefox Mobileでe10sが有効化されたのにはいくつか理由があります。

まず、モバイル環境では1つ1つのプロセスあたりが利用できるCPU資源やメモリ空間などのリソースに厳しい制限があります。このような環境では、シングルプロセスの実装ではすぐに制限の上限までリソースを使い尽くしてしまうため、タブブラウズ機能などは使い物にならなくなってしまい、酷い場合にはOSによってプロセスが強制終了されてしまうことすらあります。e10sを有効化するとプロセスごとに消費するリソースの量を分散できるため、モバイル環境ではむしろ一刻も早くe10sを有効化しなければならなかったとさえ言えるでしょう。

また、Firefox Mobileはまだ正式リリースが一度も行われていない状態だったため、拡張機能の数もデスクトップ版のFirefoxほど多くは出揃っていませんでした。ドラスティックな設計の変更を行うなら今のうちにやっておいたほうが影響が少なくて済む、というのもe10s採用の理由の1つです。

このような理由からe10sが全面的に採用されているFirefox Mobileは、将来のデスクトップ版Firefoxにおけるe10sの利用のモデルケースとして見ることができます。Firefox Mobileのソースコード自体が、e10sの枠組みの中でどのようにして目的を達成すればよいのかの豊富なサンプルとなりますので、行き詰まったときにはMXRなどでFirefox Mobileでの解法を調べてみるとよいでしょう。

Firefox Mobileのオンラインソースコード検索（MXR）
http://mxr.mozilla.org/mobile-browser/

HACK #35 メッセージマネージャのAPI詳説

Chromeウィンドウからコンテンツ領域にアクセスするときに使用するメッセージマネージャの詳細を解説します。

[Hack #34]ではメッセージマネージャの具体的な利用例を先に示しました。本稿では、先ほどの例でも利用したe10sのAPIについて詳しく解説します。

Chromeスクリプト側のメッセージマネージャは、個々の`<browser/>`要素の`messageManager`プロパティとしてアクセスできます†。

† よって、`<tabbrowser/>`要素の場合は各タブごとに1つずつ個別のメッセージマネージャが存在することになります。