Ajax と Java EE との統合を容易にする

非同期通信モデルと同期通信モデルを適切に組み合わせる

Patrick Gan
Senior IT Specialist, Application Innovation Services Center of Excellence
IBM

2006年 7月 25日

Ajax が一般的になるにつれ、このホットな話題についての俗説を払拭し、またAjax を使う上での問題に取り組む必要性が高まってきました。この記事では、シニアIT スペシャリストである Patrick Gan が、Java™ EE Web アプリケーションにAjax 技術を導入する場合、開発ライフサイクル全体において、どのような影響が考えられるかを検証します。Ajaxによる非同期通信ベースのパターンを採用する際に起こり得る問題を意識しておくことは、効果的なAjax 統合を行う上で大いに役立つはずです。

Ajax (Asynchronous JavaScript + XML) は、ごく最近の用語です (新しい衣装をまとった古い技術だと言う人も中にはいますが)。そしてJava EE コミュニティーをはじめ、様々な Web 開発コミュニティーで大きな話題を呼んでいます。Ajax技術は、過度に Web ページが更新されるのを避けることでアプリケーションの使いやすさを改善しています。またAjax の手法ではクライアントサイドとサーバーサイドの利点を組み合わせており、両者のコードを活用して、ほとんどシームレスな UI を Web ユーザーに提示することができます。Ajax は、Web の復興活動 (いわゆるWeb 2.0) を実現するための中心の 1 つとして宣伝されています。

皆さんは勤勉な Java EE 開発者として、Ajax に関する無数のハウツー記事を読み、Ajaxによって自分のアプリケーションが改善されるのではないかと興奮しているのではないでしょうか。しかし、Ajaxの非同期通信ベースのパターンを、どのようにして Java EE アプリケーションに採り入れるのでしょう。この記事では、その答えを見つけるためのヒントとして、Ajaxの導入が Java EE アプリケーションの設計や開発、パフォーマンス、テストなどに与える影響を検証します。この記事の目的は、Ajaxを使わないように推奨することでもなく、皆さんが突き当たる問題が Ajax 技術に固有の問題であると示すことでもありません。むしろ、Ajaxを効果的に、悪影響がないように使うために、問題に対して備えを固め、問題の影響を最小限にとどめるためにどうすべきかを示すことが目的です。

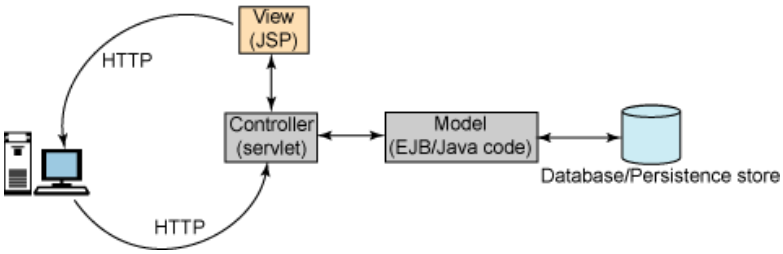
設計上の欠点に対処する

Java コミュニティーはこれまで長い間、Web 関連のアプリケーション開発に対して、適切なデザイン・パターンを適用しようと努力してきました。そうしたパターンとして、最も広く使われて

いるものの 1 つに MVC (Model-View-Controller)パターンがあります。Apache Struts などいくつかのオープンソース・フレームワークは、このデザイン・パターンやデザイン・アーキテクチャーに基づいています(参考文献を参照)。MVC には、コンサーンの分離や冗長コードの減少など、いくつかの利点があります。

コンサーンが分離されているため、アプリケーション開発プロジェクトに携わる各開発者は、事前協議されたインターフェースをアプリケーション・アーキテクチャー全体に渡って使うことによって、特定のロールに集中することができます。例えばモデル・レイヤーの開発者は、基礎となるデータ・パーシスタンス技術とインターフェースを行う技術、例えばJDBC や EJB (Enterprise JavaBeans) コンポーネント、Java クラスなどに集中することができます。ビュー・レイヤーの開発者は、JSP(JavaServer Pages) 技術やタグ・ライブラリーなどのプレゼンテーション関連技術に集中することができます。コントローラー・レイヤーは、コンサーンを明確に分離したまま受信したリクエストをバックエンドのパーシスタンス・コールにルーティングすることによって、モデルとビューを分離、仲介します。図1 は、MVC のアーキテクチャーを説明したものです。

図1. MVC アーキテクチャー



Java EE Web アプリケーションに Ajax を導入するということは、コンサーンの分離(そして開発者のロールの分離) を意味します。場合によると、Ajax によって大量のJavaScript コードがビュー・レイヤー (JSP) ページに戻される可能性があります。表1 は、Ajax を使わない場合の MVC ビュー・レイヤーで、どのくらいコードが必要になるかを示したものです。コントローラー・レイヤーはサーブレットで実装され、ビュー・レイヤーはJSP で実装されているとしています。(同期リクエストと非同期リクエストの違いについては、次のセクション、開発のジレンマに対処する、で説明します。)

表1. Ajax を使わない場合の MVC: 典型的なビュー・レイヤー・シーケンスに関連するコード量

シーケンス	説明	コードが必要か
同期リクエストを呼び出す前	フォーム送信の準備にスクリプトレット・コードが必要	不要
同期リクエストの呼び出し	フォーム送信は、ボタン、またはリンクを呼び出すことによって呼び出される。DOM要素の値は (GET あるいは POST によって) 自動的に HttpRequest に設定される。	不要。必要なものは、ページ送信を呼び出すための方法のみ。
同期リクエストへのレスポンスの処理	サーバーサイド・コードが実行を終了すると、通常は、オブジェクトが (HttpRequestによって、あるいは HttpSession に保存されて) JSP に送り返される。この時点で、(HttpRequestあるいは HttpSession を介して、またスクリプトレットあるいは何らかのタグ・ライブラリーを通して)JSP 上でこのオブジェクトにアクセスでき、オブジェクト内容を表示するためのスクリプトは最小限で済む。	最小限のスクリプトレットが必要。

表 1 を、表 2 と比べてみてください。表 2 は、Ajax を使った場合の MVC レイヤーを説明しています。この場合も、コントローラー・レイヤーはサーブレットで実装され、ビュー・レイヤーはJSP技術で実装されるものと想定しています。

表2. Ajax を使った場合の MVC: 典型的なビュー・レイヤー・シーケンスに関連するコード量

シーケンス	説明	コードが必要か
非同期リクエストを呼び出す前	Ajax コールに必要な DOM 要素の値を取得するために JavaScript コードが必要	必要
非同期リクエストの呼び出し	XMLHTTPRequest を作成し、DOM 要素の (それまでに集められた) 値を関連付け、そして (XMLHTTPRequest.send()) を送信するために JavaScript コードが必要	必要
非同期リクエストへのレスポンスの処理	サーバーサイド・コードが実行を終了すると、 (XML レスポンス・ストリームから)結果を取り出し、適切な DOM 要素に値を伝搬させるために、JavaScript コードが必要になる。	必要

Ajax を使用した場合には、ビュー・レイヤーのスクリプト・コードの量が増えることが分かります。これには、次のように3 つの大きな欠点があります。

- JSP は大量の JavaScript コードを必要とする
- この設計では、ロール・コンサーンの分離が破られる
- この設計では、モノリシック JSP が再導入されてしまう (Model 1 の手法。つまり大量の HTML や CSS コード、画像、スクリプト・コードなど)。これは非常に読みにくく、また維持管理が難しいアンチパターンです(参考文献を参照)。

こうした設計上の欠点を回避し、あるいは少なくとも軽減するための選択肢としては、下記をあげることができます。

- **再利用を考えて設計する**: 残念なことに、一般的には、Ajax をサポートするために専用のスクリプト・コードが必要になることは避けられません。そのため、スクリプト・コードは最大限再利用できるように計画し、設計する必要があります。
- **クライアントサイド MVC の手法を使う**: Dave Crane らによる Ajax in Action (参考文献を参照) に詳述されているように、クライアントサイド MVC の手法を使うことです。ただし、この手法ではコンサーンの分離は促進されますが、複雑さが増します。そのため、十分注意して使う必要があります。
- **Ajax フレームワークを使う**: いくつかのオープンソースの Ajax フレームワーク、例えば DWR (Direct WebRemoting:参考文献を参照) などを利用すると、最小限のコーディングで Ajax パターンを Java EEアプリケーションにうまく統合することができます。
- **設計の有効性を再検討する**: 基本的に、Ajax によって Web アプリケーションにはデスクトップ・アプリケーションの属性が与えられます。もし、対象とするWeb アプリケーションで、クライアントサイドの対話動作の大部分が Ajax を活用できるのであれば、そのアプリケーションはデスクトップ・アプリケーションとして設計した方が適切かもしれません。

開発のジレンマに対処する

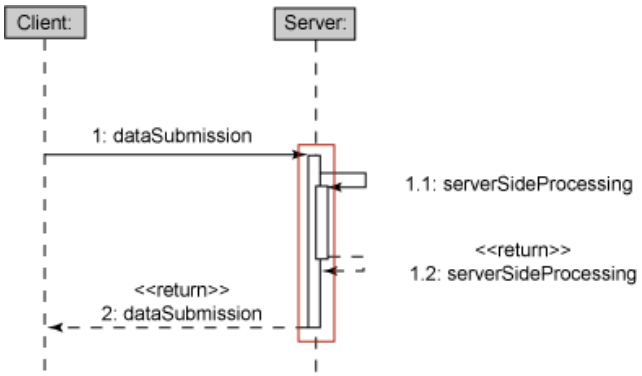
Java Web 開発に Ajax を使用する場合には、同期通信モデルと非同期通信モデルの違いを完全に理解することが重要です(参考文献を参照) 。非同期通信モデルがサポートされていないと、クライ

アントサイドの開発や、Webフレームワークとの統合、タグ・ライブラリーの使用、IDE の使い方、スレッドの振る舞いなどに影響を与える可能性があります。

同期型のリクエスト/レスポンスという通信モデルでは、(Web サーバーやアプリケーション・サーバー、Webアプリケーションなどではなく、) 常にブラウザーが (Web ユーザーを介して)リクエストを起動します。逆に Web サーバーやアプリケーション・サーバー、Webアプリケーションなどは、受信したリクエストに応答します。同期型リクエスト/レスポンスの対が処理されている間は、ユーザーはブラウザーを使い続けることができません。

図 2 は、通常の Web アプリケーションの同期通信モデルを説明したシーケンス図です。サーバーのライフラインで、クライアントからのデータ送信とサーバーサイド処理が緊密に結合されていることに注意してください。

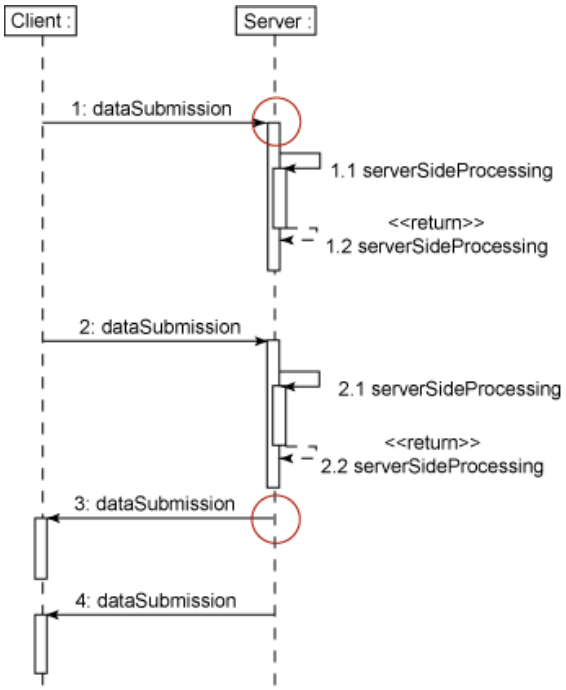
図2. 同期通信のシーケンス



非同期型のリクエスト / レスポンスという通信モデルでは、ブラウザーから Webサーバーやアプリケーション・サーバー、Web アプリケーションなどへの (Webユーザーを介しての) 通信 (あるいはその逆の通信) は、分離されています。非同期型リクエスト/ レスポンスの対が処理されている間、カレントの非同期リクエストが処理される一方で、Webユーザーはブラウザーを使い続けることができます。非同期リクエストの処理が終了すると、非同期レスポンスが、(Webサーバーやアプリケーション・サーバー、Web アプリケーションなどから) クライアント・ページに戻されます。通常、このプロセス中には、呼び出しがWeb ユーザーに影響を与えることはありません。ユーザーはレスポンスを待つ必要がないのです。

図 3 のシーケンス図は、非同期通信モデルを説明したものです。 サーバーサイド処理での最初の dataSubmissionと、最初に返される dataSubmission の両方に赤い丸がついていることに注意してください。両者のシーケンスは分離されています。またこの図は、このモードで複数の送信(スレッド) が発生しそうである、という重要な事実をハイライトしています (このセクションの後の方、スレッドの問題で詳細を説明します) 。

図3. 非同期通信のシーケンス



クライアントサイドでの影響

Web アプリケーションに Ajax を導入する際、開発チームはいくつかの危険性に注意する必要があります。危険性の主なものは、生成されるHTML ページと、そのページがブラウザーとどのように対話動作するかに関するものです。これらの問題は、ChrisLaffra による 2 回シリーズの記事、「Considering Ajax (参考文献を参照)」で詳しく説明されています。念頭に置くべき重要な点としては、下記が挙げられます。

- **スクリプトが利用できない可能性がある**: 多くのユーザーのブラウザーでは、様々な理由から JavaScript が利用できないようになっています。
- **多種類のブラウザーをサポートしようとするとコード要求が増加する**: 複数のブラウザーや複数バージョンのブラウザーをサポートするアプリケーションでは、スクリプト・コードが増える可能性があります。これは、それぞれのブラウザーがDOM 要素を解釈する方法 (そして、こうした要素に操作を行う JavaScript コード)に微妙な違いがあるためです。
- **JavaScript はセキュアではない**: 大部分のブラウザーでは、HTML ページに関連付けられた JavaScript ソースコードは、ソースを見るオプションを選択すれば見ることができます。Ajax パターンを使う場合には、スクリプト・コードで実装しているロジックの機密性に問題がないことを確認する必要があります。

Web フレームワークとの統合

自分が選択した、好みの Java EE Web フレームワークと Ajax 開発を統合しようと思うのは自然なことです。しかし、一部のJava EE Web フレームワークは (今のところ) 、そのままの状態で非同期通信モデルをサポートするようになっていません。これが何を意味するかを知るためには、サーブレットが同期通信と非同期通信をどのように扱うかを理解する必要があります。図4 は、同期リクエストを処理する場合の、通常のサーブレット・シーケンスを示しています。

図4. 同期リクエストを処理する場合のサーブレット・シーケンス

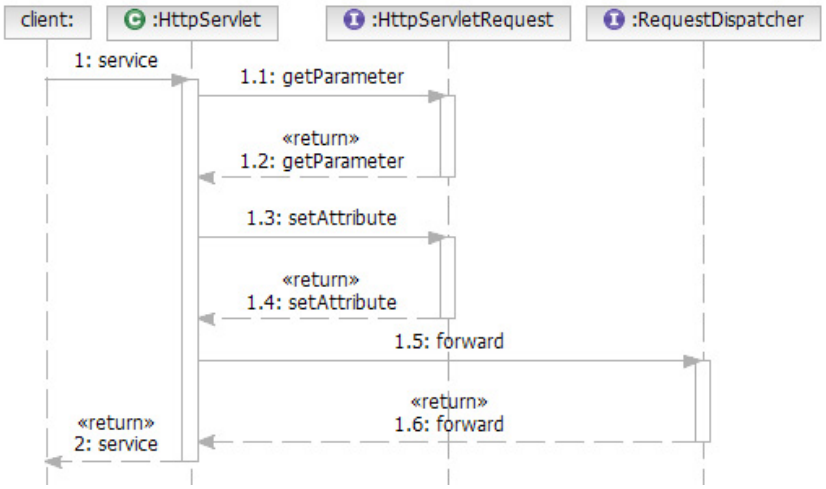


図 4 は、Java EE Web 開発者にとっておなじみのはずです。ブラウザーからのリクエストは、まずコントローラー・サーブレットのservice() によって処理されます。このサーブレットは、自分が必要とする値を、HttpRequestから (パラメーターとして、あるいは属性として) 取得します。コントローラー・プロセスが終了すると、その結果はHttpRequest (あるいは HttpSession) の中に戻され、RequestDispatcher が、そのページにコントロールを転送します(あるいは含めます) 。

図 5 は、非同期リクエストを処理する場合のサーブレット・シーケンスを示しています。

図5. 非同期リクエストを処理する場合のサーブレット・シーケンス

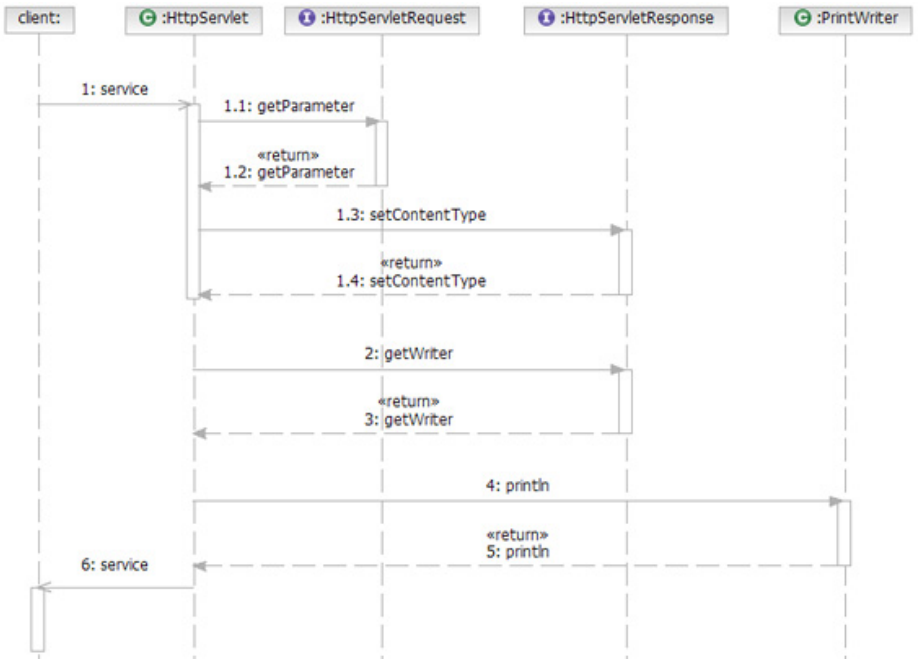


図 5 のシーケンスは、同期シーケンスとは少し異なります。ブラウザーからのリクエストは、まずコントローラー・サーブレットのservice() によって処理されます。このサーブレットは、自分が必要とする値を、HttpRequestから (パラメーターとして、あるいは属性として) 取得します。コントローラー・プロセスが終了したら、HttpServletResponse のコンテンツ・タイプを XML に設定

する必要があります。また、コントローラー・ロジックの結果はPrintWriter を使って書き出されます。この時点では、RequestDispatcher はバイパスされて使われません。

大部分の Java EE Web フレームワークがサポートしていないのは、正にこの (非同期)シーケンスなのです。そのため、Ajax との統合が困難なのです。非同期通信をサポートしないポートレットや JSF (JavaServer Faces) フレームワークも、同じ問題に直面します。

この状況を克服するための選択肢としては、下記のようなものがあります。

- **Web フレームワークと共存する**: 自分が選択したフレームワークが組み込みで Ajax をサポートするようになるのを待ったり、そのフレームワークで強制的にAjax をサポートしたりする代わりに、すべての非同期リクエストを処理するために別のサーブレットを提供します。DWRは、この手法を採用しています。この手法の欠点は、Ajax リクエストが、そのフレームワークの機能を容易に利用できないことです。
- **Web フレームワークに統合する**: 無料で入手できるエクステンションを使ったり、カスタムのエクステンションを書いたりすることで、自分の選択したWeb フレームワークとの統合方法を工夫することができます。
- **Ajax をサポートするフレームワークに移行する**: 新しいフレームワークでは、非同期通信モデルをサポートするようになってきています。その1 つがApache Shale です (参考文献を参照)。

タグ・ライブラリーを使う

一般的に、Java Web アプリケーションの開発ではタグ・ライブラリー (taglibs)を多用します。多くの Java EE Web フレームワークと同様、一部の taglibs は今のところ、そのままでは非同期通信モデルをサポートしていません。そのためXMLHttpRequest を使って送信されたデータを HttpServletRequest に変換する(あるいはその逆の) 方法がありません。つまり、非同期通信をサポートしないtaglibs は、Ajax の XMLHttpRequest コールの呼び出し中は使い物にならないのです。そこで選択肢としては次のようになります。

- **非同期モデルをサポートしない taglibs を使うことをあきらめる**: 現在 taglibs で生成しているコードを、HTML/JavaScript コードに移行します。(もし Web アプリケーションが taglibs に大きく依存している場合には、この手法によってビュー・レイヤー・ページのサイズが増加することになります。)
- **問題を回避する**: この問題に対して既に回避策を持っている Ajax フレームワークを使います。その一例がDWR です (ExecutionContext.forwardToString() を見てください) 。この場合は、それまで使っていたtaglibs を使い続けることができます。
- **Ajax をサポートする taglibs を使う**: 非同期モデルをサポートする taglibs、例えば AjaxTags (Ajax JSP Tag Library) などを使います (参考文献を参照)。

IDE を使った開発とデバッグ

JavaScript ソリューションの開発者を補助するための JavaScript デバッグ・ツールは数多くあります。しかし通常のJava 開発環境では、 XMLHTTPRequest の値や、Ajax に関連する様々な特質を調べることができません。

1 つの解決策としては、ATF (AJAX Toolkit Framework) を活用することです (参考文献を参照)。ATF は Eclipse プラグインであり、編集時構文チェックや、埋め込みのMozilla Web ブラウザー、埋

め込みの DOM ブラウザー、埋め込みの JavaScriptデバッガーなど、高度な JavaScript 編集機能を持っています。また ATF にはPersonality Builder 機能が含まれており、任意の Ajax ランタイム・フレームワーク用にIDE 機能を構築する上で役立ちます。また構築された環境機能は、ATF が既にサポートしている一連のランタイム環境に追加されます。

スレッドの問題

通常の同期 Web アプリケーションでは、一部の領域で、ボタンやリンクのクリックのために多少長めの処理時間が必要です。短気な、あるいは経験の浅いWeb ユーザーは、処理を早めようとボタンやリンクを何度もクリックし、複数のフォーム送信を呼び出しがちです。また場合によると、ユーザーは(デスクトップ・アプリケーションの場合のように) ダブルクリックが必要と思ってしまうこともあります。Webアプリケーションによっては、フォーム送信を繰り返し行っても実害のないこともありますが、副作用によって深刻なスレッド問題やレース条件(複数のスレッドが、ある 1 つのコード・ブロックを実行しようとして競合する)を引き起こす場合もあります。例えば、銀行アプリケーションで Transfer Funds(送金) ボタンを繰り返しクリックすると、意図せずに送金操作を繰り返してしまう可能性があります。

同期通信モデルと非同期通信モデルの両方をサポートする Web アプリケーションも、その機能を適切に分析し、準備しておかないと、同じ状況に陥ります。両方の通信モデルをサポートするアプリケーションは、対象のページに対して様々な方式のサーバーサイド呼び出し(つまり、同期のみ、非同期のみ、そして同期非同期の混合) を行ってしまうかもしれません。繰り返しクリックのシナリオにも見られるように、非同期コールの処理は多少遅いかもしれません。また、もしアプリケーションが防止策を講じていない場合には、(ページが更新されず、そのページに対してさらにアクティビティーが行われることが防止できないため)非同期スレッドの処理中にユーザーが同期コールを呼び出してしまうかも知れません。その結果、2つのスレッドは同時に処理されることになります。こうした状況は、Web ページ上の同じボタンや同じリンクから発生したものではありませんが、(繰り返しクリックの問題と同じように)サーバーサイド・コードにスレッドの問題を引き起こします。

例えば図 6 に示すように、銀行アプリケーションの送金用ページの例を見てください。

図6. 送金の例

この例では、赤で示す Transfer Funds は Ajax コールを呼び出し、(黄色の)Logout リンクは同期コールを呼び出すものとします。短気な、あるいは経験の浅いユーザーが赤のボタンと黄色のリ

ンクを連続してクリックすると(そして両方のリンクがコード中で共通パスを持っているとすると) 、やがてレース・コンディションが発生します。

一般的に言って、この状況が発生するのを防ぐには、2 つの方法があります。その1 つはクライアントサイドでの解決策です。リンクあるいはボタンが呼び出されたら、その後は現在のスレッドが実行を完了するまでページ送信が行われないように、JavaScriptを使って禁止するのです。2 番目の解決策は、レース・コンディションが起きないようサーバーサイド・コードの同期動作に頼ることによって、複数のスレッド送信を許す方法です。この問題を解決するために同期動作を導入する場合には、JavaEE Web コンポーネント (サーブレットやポートレット、JSFなど) がマルチスレッドであることを忘れないでください。コード(特に、リクエスト / レスポンスのライフサイクル処理に関連するコード) の大きな部分を同期化する場合には、十分注意してください。同期動作を誤って使うと、実質的にアプリケーションはシングル・スレッドのアプリケーションとなり、従ってスループットが低下します。

パフォーマンス上の落とし穴に対処する

Ajax を使った場合には、Java EE Web ベースのアプリケーションのパフォーマンスが影響を受ける可能性もあります。各リクエストに対してスレッドの追加が許されることによって、2つのリソースが影響を受ける可能性があります。

第 1 に、サーブレット・コンテナー内のスレッド・プールが影響を受ける可能性があります。スレッド・プールは、ある 1 つの Web コンテナー内で同時に実行することが許されるスレッドの最大数を規定します。1 つのクライアント・リクエストに対して、1 つのスレッドが要求されます。しかし、1 つのクライアント・リクエストは必ずしも 1 つのユーザー・リクエストと同じではありません。ブラウザーが、1 つのユーザー・リクエストに対していくつかのクライアント・リクエストを要求するかもしれません。例えば、フォームを送信する 1 人のユーザーに対して、いくつかのクライアント・リクエスト (つまり、フォームの値の送信、GIF ファイルの取得、JavaScript ファイルの取得、CSS ファイルの取得) が要求されるかもしれません。同期リクエストと非同期リクエストを同時に送信することが許されている場合には、この状況は、(Ajax リクエストに対して) ユーザー・リクエストごとに少なくとも、もう 1 本のスレッドを追加して使用することを意味します。ユーザー・リクエストごとにもう 1 本スレッドを追加するという可能性は、たいしたことではないように思えるかもしれませんが、アプリケーションの負荷が重い場合には (ユーザー・リクエスト毎に追加される各スレッドと平均ユーザー数とを掛け合わせて考えると) 、その影響は明らかです。当然ですが、この状況はサーブレット・コンテナーのパフォーマンスに影響を与える可能性があります。

もう 1 つ、影響を受ける可能性のあるリソースは、データベース・コネクション・プールです。通常のJava EE Web アプリケーションでは、ユーザー・リクエストに対して、シャロー・リクエスト(shallow request) とディープ・リクエスト (deep request) という、2 種類のシーケンスが使えるようになっています。シャロー・リクエストはWeb ページからのリクエストであり、サーバーサイド・コードを実行しますが、リクエストを完了するためにパーシスタンス・ストア(データベースなど) にアクセスすることはありません。ディープ・リクエストもWeb ページからのリクエストであり、サーバーサイド・コードを実行しますが、リクエストを完了するためにパーシスタンス・ストアへのアクセスを実際に行います。

(データベース接続が必要だとすると) ディープ・リクエスト・シーケンスでは、追加スレッドを許すことによって、データベース・コネクション・プールの次のような側面が影響を受ける可能性があります。

- 接続を待つ平均スレッド数
- ミリ秒単位での平均接続待ち時間
- 接続が使用される平均時間

その結果、コネクション・プールの平均サイズを大きくしたり、接続の数を増やしたりする必要が出てくるかも知れません。

テストに取り組む

Java 開発者は、Java SE や Java EE コードの周囲にユニットテスト・ハーネスを用意することを次第に強く意識するようになっています。Ajaxの導入によってブラウザー内の JavaScript の量が増えるにつれ、しっかりとしたJavaScript ユニットテスト・フレームワークも求められるようになっています。そうしたものとして現在入手できるのは、JsUnitや Selenium、HttpUnit などです (参考文献を参照) 。

こうしたフレームワークは、Web ページ上の DOM 要素を操作する JavaScript関数に対して、ユニットテスト開発用のハーネスを提供します。また、ユニットテストを、テスト・スイートとしてグループ化することもできます。例えばSelenium のブラウザー互換性テスト機能を使うと、様々なブラウザーやオペレーティング・システムでのJavaScript 関数をテストすることができます。Selenium は JavaScript と Iframesを使って、テスト自動化エンジンをブラウザーに埋め込みます。この手法は、JavaScriptが使えるブラウザーであればどれでも動作するはずであり、複数のブラウザーや複数のブラウザー・バージョンをサポートするアプリケーションのテストに特に便利なはずです。Seleniumも JsUnit も、連続統合 (continuous integration) をサポートしています。つまりJavaScript のユニットテストとテスト・スイートを、自動ビルド・プロセスの中に統合することができます。

まとめ

Java EE アプリケーションに Ajax を導入することには、(他の技術やパターンを導入する場合と同様)、利点と欠点があります。この記事では、JavaEE Web アプリケーションに Ajax を統合する場合の全体像を示しました。Ajaxの非同期通信モデルは、伝統的な Java EE Web アプリケーションがサポート対象としてきた同期モデルとは大きく異なっています。予期しない問題に突き当たるのを避けるために、どのような潜在的問題があるのか、Ajaxに飛び込む前に十分に検討することが必要です。

また、Java EE フレームワークでの Ajax サポートやユーティリティーは改善が続けられています。今後は、最初からAjax をサポートしているフレームワークを活用することによって、統合に伴う複雑さを軽減することを考えるべきでしょう。そうした候補として、JSFベースの Apache Shale とサーブレット・ベースの DWR は、今後注目すべき主な2 つ、と言うことができます。

著者について

Patrick Gan

Patrick Gan は、IBM Global Services のシニア情報技術スペシャリストです。彼の専門領域は、Java EE やオブジェクト指向アプリケーション開発、オープンソース・フレームワークの使用などです。主に顧客との協力業務を行っており、顧客による Java EE アプリケーション開発の、設計や開発フェーズに協力しています。また、IBM 独自の Java EE ベース・フレームワークの設計や開発、維持管理などにも携わっています。IBM に入社して 6 年になり、コンピューター・サイエンスの学位を持っています。