Java の理論と実践: ステートフルな Web アプリケーションはすべて壊れているのか

HttpSession とその仲間たちは見た目よりも厄介なものです

Brian Goetz
Senior Staff Engineer
Sun Microsystems

2008年 9月 23日

サーブレット・フレームワークが提供するセッション状態管理メカニズム、`HttpSession` はステートフルなアプリケーションを簡単に作成できるようにしますが、使い方を間違えやすいのも事実です。可変データ (JavaBeans クラスなど) に `HttpSession` を使用する Web アプリケーションのなかには、十分な調整をせずにこのメカニズムを使用しているために、並行性にさまざまな悪影響を及ぼす可能性を招いているものが数多くあります。

このシリーズの他の記事を見る

Java™ エコシステムには多数の Web フレームワークがありますが、そのすべてが、直接的あるいは間接的に、サーブレットのインフラストラクチャーをベースとしています。サーブレット API は `HttpSession` と `ServletContext` メカニズムによる状態管理をはじめ、アプリケーションがユーザー・リクエスト間で状態を維持できるようにする数多くの便利な機能を提供します。しかし、Web アプリケーションでの共有状態には微妙な (そしてそのほとんどは慣例による) 適用ルールがあり、多くのアプリケーションは知らず知らずのうちにこれらのルールに抵触しています。そのため、数多くのステートフルな Web アプリケーションには捕らえにくい深刻な欠点が伴う結果となっています。

スコープ・コンテナー

サーブレット仕様の `ServletContext`、`HttpSession`、`HttpRequest` オブジェクトはいずれもスコープ・コンテナーと呼ばれます。このそれぞれのオブジェクトには、アプリケーションに代わってデータを保存する `getAttribute()` と `setAttribute()` メソッドがあります。3 つのオブジェクトの違いは、スコープ・コンテナーの有効期間です。データが維持されるのは、`HttpRequest` の場合はリクエストの有効期間のみ、`HttpSession` の場合はユーザーとアプリケーションとの間でセッションが維持される期間、そして `ServletContext` の場合はアプリケーションの有効期間中ずっとデータが維持されます。

HTTP プロトコルはステートレスであるため、ステートフルな Web アプリケーションの構築ではスコープ・コンテナーが大いに役立ちます。サーブレット・コンテナーがアプリケーションの

状態、そしてデータのライフサイクルを管理する役目を受け持つからです。仕様ではこの話題についてはほとんど触れていないものの、セッション・スコープおよびアプリケーション・スコープが設定されたコンテナーはある程度、スレッド・セーフにしなければなりません。それは、`getAttribute()` メソッドと `setAttribute()` メソッドは常に、異なるスレッドによって呼び出される可能性があるためです (仕様では、この 2 つのメソッド実装をスレッド・セーフにすることが必須だとは直接規定していませんが、提供するサービスの性質からして、これらのメソッドは実質的にスレッド・セーフである必要があります)。

スコープ・コンテナーが Web アプリケーションにもたらす可能性のある大きな利点はもう 1 つあります。それは、スコープ・コンテナーではアプリケーション状態の複製とフェイルオーバーをアプリケーションに対してトラスペアレントに管理することができるということです。

セッション

セッションとは、特定のユーザーと Web アプリケーションの間で交わされる一連のリクエストとレスポンスです。ユーザーは自分たちの認証クレデンシャル、ショッピング・カートの内容、そして以前のリクエスト時に Web フォームに入力した情報を Web サイトが記憶していると期待します。しかし、コアとなる HTTP プロトコルはステートレスなので、リクエストに関するすべての情報はリクエスト自体に保管しなければなりません。そのため、リクエスト・レスポンスのサイクル単位ではなく、それよりも長い期間、ユーザーとの対話動作を有効にするにはセッション状態をどこかに保持しておく必要があります。サーブレット・フレームワークは、それぞれのリクエストをセッションに関連付けることが可能で、そのセッションに関連する (キー、値) データ項目の値のストアとして機能する `HttpSession` インターフェースを提供します。リスト 1 に、`HttpSession` にショッピング・カートのデータを保管する典型的なサーブレット・コードの抜粋を記載します。

リスト 1. HttpSession を使用したショッピング・カート情報の保管

```
HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart)session.getAttribute("shoppingCart");
if (cart == null) {
    cart = new ShoppingCart(...);
    session.setAttribute("shoppingCart");
}
doSomethingWith(cart);
```

リスト 1 のサーブレットの使い方は典型的なもので、アプリケーションはオブジェクトがすでにセッションに配置されているかどうかを調べ、配置されていない場合にはセッションのその後のリクエストで使用できるようにオブジェクトを作成します。サーブレットをベースにビルドされた Web フレームワーク (JSP、JSF、SpringMVC など) は詳細を見せませんが、基本的にはセッション・スコープとしてタグが付けられたデータに対して、これと同様の操作を行います。残念ながら、リスト 1 の使用方法も同じく誤っている可能性があります。

スレッド化についての考慮事項

HTTP リクエストがサーブレット・コンテナーに到達すると、`HttpRequest` および `HttpResponse` オブジェクトが作成されてサーブレットの `service()` メソッドに渡されます。これは、サーブレット・コンテナーが管理するスレッドのコンテキストで行われます。レスポンスを作成する役割を担うサーブレットは、そのスレッドの制御をレスポンスが完了するまで維持し、レスポンスが完了した時点でスレッドを使用可能なワーカー・スレッドのプールに戻します。サーブレッ

ト・コンテナーは、スレッドとセッションの間のアフィニティーについては維持しません。そのため、特定のセッションで次に入ってくるリクエストは、おそらく現行のリクエストとは異なるスレッドによって対応されることになります。実際、同じ 1 つのセッションで複数の同時リクエストが入ってくる可能性もあります。このような事態は、フレームまたは AJAX 技術を使用して、ユーザーがページと対話している間にサーバーからデータを取得してくる Web アプリケーションで起こり得ます。このような場合、複数の異なるスレッドを並行して実行している 1 人のユーザーから複数の同時リクエストが入ってくることもあります。

このようなスレッド化に関する考慮事項は、大抵の場合、Web アプリケーション開発者には関係してきません。HTTP が持つステートレスな特質は、レスポンスが、リクエストに保管されたデータ (他の同時リクエストとは共有されないデータ)、そしてすでに並行制御を行っているリポジトリー (データベースなど) に保管されたデータだけの関数になるように仕向けるからです。しかし Web アプリケーションがデータを `HttpSession` や `ServletContext` といった共有コンテナーに保管するようになれば、それはもう並行アプリケーションです。そのため、開発者はアプリケーション内でのスレッド・セーフについて考慮しなければならなくなりました。

スレッド・セーフという用語は通常コードについて説明する際に使用されていますが、実際にはデータに関する用語です。具体的に言うと、スレッド・セーフとは複数のスレッドがアクセスする可変データへのアクセスを適切に調整することです。サーブレット・アプリケーションは多くの場合、スレッド・セーフです。これは、アプリケーション間で共有する可変データはないため、さらに同期化を行う必要はないという事実のおかげですが、Web アプリケーションに共有状態が導入される可能性は山ほどあります。例えば `HttpSession` や `ServletContext` などのスコープ・コンテナーだけではありません。`HttpServlet` オブジェクトの静的フィールドやインスタンス・フィールドも然りです。Web アプリケーションにリクエスト間でデータを共有する必要が出てきた場合には、アプリケーション開発者はその共有データがどこに存在するかに注意を払い、スレッド化の危険を避けるために共通データにアクセスするときにはスレッド間で十分に調整が行われることを確実にしなければなりません。

Web アプリケーションにとってのスレッド化のリスク

Web アプリケーションが、例えばショッピング・カートなどの可変セッション・データを `HttpSession` に保管すると、2 つのリクエストが同時にショッピング・カートにアクセスしようとする可能性が生まれます。この場合、以下をはじめとする複数の障害モードが考えられます。

- アトミック性の障害。あるスレッドが複数のデータ項目を更新しているときに、データ項目の整合が取れていない状態のうちに、別のスレッドがこのデータを読み取るという障害です。
- 読み取りスレッドと書き込みスレッドの間での可視性の障害。あるスレッドがカートを変更しているにも関わらず、別のスレッドには、古くなったカートの内容または矛盾した状態のカートの内容が可視になるという障害です。

アトミック性の障害

リスト 2 に記載するのは、ゲームのアプリケーションでハイスコアを設定および取得するメソッドの (壊れた) 実装です。この実装でハイスコアを表すために使用している `PlayerScore` オブジェクトは、`name` と `score` というプロパティーを持つ通常の JavaBean クラスで、アプリケーション・スコープが設定された `ServletContext` に保管されます (ここでは、アプリケーションの起動時に

は初期状態でのハイスコアが `highScore` 属性として `ServletContext` にインストールされているため、`getAttribute()` 呼び出しは失敗しないと推測されています)。

リスト 2. 壊れたスキームによるスコープ・コンテナーでの関連項目の保管

```
public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    PlayerScore result = new PlayerScore();
    result.setName(hs.getName());
    result.setScore(hs.getScore());
    return result;
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.getScore() > hs.getScore()) {
        hs.setName(newScore.getName());
        hs.setScore(newScore.getScore());
    }
}
```

リスト 2 のコードにはいくつも壊れている箇所があります。ここで取っている手法は、ハイスコア・プレイヤーの名前とスコアの可変ホルダーを `ServletContext` に保管し、新たにハイスコアが出た場合には、その名前とスコアの両方を更新するというものです。

現在のハイスコアである 1000 点を出したプレイヤーは Bob で、彼のスコアは Joe が出したスコアである 1100 点に抜かされたとします。Joe のスコアが書き込まれているときに、別のプレイヤーがハイスコアを要求すると、`getHighScore()` メソッドはサーブレット・コンテキストから `PlayerScore` オブジェクトを取得し、そこから名前とスコアを取得することになります。しかしタイミングが悪ければ、このメソッドが Bob の名前と Joe のスコアを取得して、Bob が 1100 点を出したと示す可能性があります。これは、事実とは異なります (このような失敗は無料のゲーム・サイトでは許されるかもしれませんが、「スコア」を「銀行預金残高」に置き換えてみてください。かなりの問題になるはずです)。互いにアトミックとなるはずの 2 つの操作 (名前とスコアのペアを取得する操作と、名前とスコアのペアを更新する操作) が実際には互いに対してアトミックに実行されず、矛盾した状態の共有データを一方のスレッドが参照できてしまったという点で、これはアトミック性の障害です。

さらに、スコア更新のロジックは「check-then-act (チェック後に行動)」というパターンに従うため、2 つのスレッドがハイスコアを更新しようと「競合」状態になり、予測できない結果となる場合もあります。例えば現在のハイスコアが 1000 で、2 人のプレイヤーがそれぞれのハイスコアである 1100、1200 を同時に登録したとします。たまたまタイミングが悪いと、どちらも「新しいスコアは現在のハイスコアよりも高得点か？」という検査にパスし、ハイスコアを更新する処理ブロックが実行されます。この場合もタイミングによっては、矛盾した結果 (名前は一方のプレイヤーのもので、ハイスコアはもう一方のプレイヤーのもの) になったり、まったく誤った結果 (スコアが 1100 のプレイヤーが 1200 を記録したプレイヤーの名前とスコアを上書きする) になったりすることが考えられます。

可視性の障害

アトミック性の障害より捉えにくいのは、可視性の障害です。同期化が行われないと、同じ変数に対して書き込みを行うスレッドと読み取りを行うスレッドがある場合、読み取りを行うほうのスレッドが古いデータ、つまりすでに無効なデータを参照してしまう可能性があります。その上、変数 `y` のほうが変数 `x` より前に書き込まれたにも関わらず、読み取るほうのスレッドに可視になるのは、変数 `x` の最新データと変数 `y` の古いデータとなる可能性さえあります。可視性の障害は予測不可能で、頻繁には起きないため捉えにくく、デバッグが困難な稀にしか発生しない断続的障害を引き起こすこともあります。可視性の障害をもたらすのは、データ競合による、共有変数にアクセスする際の適切な同期化の失敗です。データ競合のあるプログラムは、その振る舞いを確実に予測できないという点で壊れていると言えます。

JMM (Java Memory Model) では、変数を読み取るスレッドに別のスレッドでの書き込み結果を間違いなく可視にするための条件を定義しています (JMM についての詳細な説明はこの記事では行わないので、「参考文献」を参照してください)。JMM が定義しているのは、プログラムの操作に対する順序付けで、happens-before (事前発生) と呼ばれる関係を持つ順序付けです。スレッド全体での happens-before の順序付けは、共通するロックでの同期化、または共通の volatile 変数へのアクセスによってのみ作成されます。happens-before の順序付けが作成されない場合、Java プラットフォームでは大幅な遅延が生じるか、あるいはあるスレッドで行われた変数への書き込み操作が、別のスレッドでの当該変数の読み取り操作の際に可視になるように、順序が変更されることになります。

リスト 2 のコードには、アトミック性の障害だけでなく、可視性の障害もあります。それに該当するのは、`updateHighScore()` メソッドが `ServletContext` から `HighScore` オブジェクトを取得してから、`HighScore` オブジェクトの状態を変更するという部分です。この目的としては、これらの変更を `getHighScore()` を呼び出す他のスレッドに可視にすることですが、`updateHighScore()` の name および score プロパティーに対する書き込み操作が行われてから、`getHighScore()` を呼び出す他のスレッドでこれらのプロパティーが読み取られるまでの間で happens-before の順序付けが作成されていません。そのため、読み取り側のスレッドに正しい値が可視になるかどうかは運に任せるしかないのです。

考えられるソリューション

サーブレット仕様ではサーブレット・コンテナーが保証しなければならない happens-before に関して十分に記述していませんが、結論として至るのは、共有スコープ・コンテナー (`HttpSession` または `ServletContext`) に属性を配置するのは、別のスレッドがその属性を取得する前でなければならないということです (この結論の根拠については、『Java Concurrency in Practice』 4.5.1 を参照してください。この仕様が言っているのは結局のところ、「要求スレッドを実行する複数のサーブレットがアクティブに単一のセッション・オブジェクトに同時アクセスする可能性があるため、開発者がセッション・リソースへのアクセスを適切に同期化しなければならない」ということに尽きます)。

書き込み後に設定する set-after-write 手法

「ベスト・プラクティス」としてよく言われているのは、スコープ・セッション・コンテナーに保管された可変データを更新するときには、データを変更してからもう一度 `setAttribute()` を呼び出すという手法です。この手法を使って書き直した `updateHighScore()` の一例をリスト 3 に記

載します (この手法を使用する動機の 1 つは、コンテナーに値が変更されたことを示唆し、それによってコンテナーが分散 Web アプリケーションのインスタンス全体でセッションまたはアプリケーションの状態を再同期化できるようにすることです)。

リスト 3. set-after-write 手法を使用して、サーブレット・コンテナーに値の更新を示唆する例

```
public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.getScore() > hs.getScore()) {
        hs.setName(newScore.getName());
        hs.setScore(newScore.getScore());
        ctx.setAttribute("highScore", hs);
    }
}
```

この手法はクラスター化されたアプリケーションのなかで効率的にセッションとアプリケーションの状態を複製するという問題には役立ちますが、残念ながら、この例での基本的なスレッド・セーフの問題を修正するには不十分です。可視性の問題 (別のプレイヤーには `updateHighScore()` で更新された値が可視にならない可能性があるという問題) は緩和するものの、アトミック性に関して考えられる複数の問題を対処するまでには至りません。

同期化ピギーバック

happens-before の順序付けは推移的であり、その範囲は `updateHighScore()` での `setAttribute()` 呼び出しと `getHighScore()` での `getAttribute()` 呼び出しの間に限られているため、set-after-write 手法によって可視性の問題は解消されます。`HighScore` の状態が更新された後に `setAttribute()` が呼び出され、その後に `getAttribute()` によって `HighScore` の状態が返され、さらにその後に `getHighScore()` の呼び出し側が `HighScore` の状態を使用します。この推移性により、`getHighScore()` の呼び出し側に可視になる値は、最後の `setAttribute()` 呼び出しが行われた時点より古い値にはならないと結論を下すことができます。この手法は同期化ピギーバック (piggybacking on synchronization) と呼ばれますが、それは `getHighScore()` メソッドと `updateHighScore()` メソッドがそれらが同期していることを `getAttribute()` および `setAttribute()` で利用して最小限の可視性を保証することができるからです。ただし、この手法でもまだ、この記事の例には対処しきれません。set-after-write 手法は状態の複製には役立つかもしれませんが、スレッド・セーフを実現するには力不足です。

不変性への依存

スレッド・セーフなアプリケーションを作成するのに役立つ手法は、できるだけ多くの不変データを使用するようにすることです。リスト 4 のハイスコアの例は、`HighScore` の不変の実装を使用して書き直しています。この実装では、存在しないプレイヤーとスコアのペアを呼び出し側が参照できることがあるというアトミック性の障害も、`getHighScore()` の呼び出し側が `updateHighScore()` の呼び出しによって書き込まれた最新の値を参照できないという可視性の障害もありません。

リスト 4. 不変の HighScore オブジェクトの使用によるアトミック性および可視性の問題の大部分の解消

```
Public class HighScore {
```

```
    public final String name;
    public final int score;

    public HighScore(String name, int score) {
        this.name = name;
        this.score = score;
    }
}

public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    return (PlayerScore) ctx.getAttribute("highScore");
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    if (newScore.score > hs.score)
        ctx.setAttribute("highScore", newScore);
}
```

リスト 4 のコードには障害の可能性の低いモードが多く含まれています。まず、`setAttribute()` と `getAttribute()` での同期化ピギーバックが可視性を保証します。そして単一の不変データ項目だけが保管されているという事実が、`getHighScore()` の呼び出し側に対して名前とスコアのペアの矛盾した更新が示されるというアトミック性の障害の可能性を排除します。

不変のオブジェクトをスコープ・コンテナーに配置することで、アトミック性と可視性の障害はほとんど回避されます。さらに、実質上不変のオブジェクトについてもスコープ・コンテナーに配置するのが安全です。実質上不変のオブジェクトとは、理論的には可変であっても、公開されてからは実際に変更されることが決してないオブジェクトのことです。例えば、いったん `HttpSession` に配置された JavaBean オブジェクトのセッターは決して呼び出されることはなく、この場合 JavaBean オブジェクトは実質上不変のオブジェクトということになります。

`HttpSession` に置かれたデータにアクセスするのは、そのセッションでのリクエストだけではありません。コンテナーがある種の状態複製を行っている場合には、コンテナー自体がアクセスする場合もあります。

> `HttpSession` または `ServletContext` に配置するデータはすべて、スレッド・セーフであるか、または実質的に不変でなければなりません。

アトミックな状態遷移の実現

リスト 4 のコードには 1 つだけ問題が残っています。それは、`updateHighScore()` での check-then-act パターンにより、ハイスコアを更新しようとする 2 つのスレッド間での競合が発生する可能性が依然としてあることです。そのため、タイミングが悪ければ更新が失われてしまう場合もあります。2 つのスレッドが「新しいスコアは現在のハイスコアよりも高得点か？」というチェックに同時にパスすると、両方のスレッドが `setAttribute()` を呼び出します。タイミングによっては、2 つのうちの高いほうのスコアがハイスコアとして記録されるという保証はありません。この最後の問題に対処するには、アトミックにスコア参照を更新すると同時に、それぞれが干渉しないようにする手段が必要です。それにはいくつかの手法を使用できます。

リスト 5 では `updateHighScore()` に同期化の操作を追加し、更新プロセスに伴う check-then-act パターンが別の更新操作と同時には実行できないようにします。このような条件付き変更ロジッ

クが `updateHighScore()` で使用するロックと同じロックを取得する場合には、この手法で十分です。

リスト 5. 同期化を使用した最後のアトミック性の問題の解消

```
public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    PlayerScore hs = (PlayerScore) ctx.getAttribute("highScore");
    synchronized (lock) {
        if (newScore.score > hs.score)
            ctx.setAttribute("highScore", newScore);
    }
}
```

リスト 5 の手法でも上手くいきますが、それよりもさらに優れた手法があります。それは、`java.util.concurrent` パッケージの `AtomicReference` クラスを使用することです。このクラスは、`compareAndSet()` 呼び出しによってアトミックな条件付き更新操作を行うように設計されています。リスト 6 に、`AtomicReference` を使用してこの例の最後のアトミック性を修復する方法を示します。このコードのほうがリスト 5 のコードよりも優れている理由は、ハイスコアの更新方法についての前提に反する可能性が少ないためです。

リスト 6. AtomicReference を使用した最後のアトミック性の問題の解消

```
public PlayerScore getHighScore() {
    ServletContext ctx = getServletConfig().getServletContext();
    AtomicReference<PlayerScore> holder
        = (AtomicReference<PlayerScore>) ctx.getAttribute("highScore");
    return holder.get();
}

public void updateHighScore(PlayerScore newScore) {
    ServletContext ctx = getServletConfig().getServletContext();
    AtomicReference<PlayerScore> holder
        = (AtomicReference<PlayerScore>) ctx.getAttribute("highScore");
    while (true) {
        HighScore old = holder.get();
        if (old.score >= newScore.score)
            break;
        else if (holder.compareAndSet(old, newScore))
            break;
    }
}
```

スコープ・コンテナーに配置された可変オブジェクトの場合、その状態遷移は同期化、または `java.util.concurrent` のアトミックな可変クラスによってアトミックに行われる必要があります。

HttpSession に対するアクセスのシリアライズ

これまでに説明した例では、アプリケーション全体の `ServletContext` でデータにアクセスする際に伴うさまざまな危険を回避しようとしてきました。`ServletContext` にはどのリクエストからでもアクセスできるため、`ServletContext` にアクセスするときには慎重な調整が必要なことは明らかです。その一方、大抵のステートフルな Web アプリケーションは、セッション・スコープ・コンテナー、`HttpSession` に大きく依存しています。複数の同時リクエストが同じセッションでどのように発生するかは明らかではないかもしれませんが、結局のところ、セッションは特定の

ユーザーとブラウザー・セッションに結び付けられているため、ユーザーが複数のページを同時に要求しそうにはありません。しかし Ajax アプリケーションのようにリクエストをプログラムによって生成するアプリケーションでは、セッションでのリクエストが重なる可能性があります。

残念ながら、単一のセッションで複数のリクエストが重なる可能性はあります。そこで、セッションでのリクエストを簡単にシリアライズできるとしたら、これまでに説明した `HttpSession` の共有オブジェクトにアクセスする際の危険はほぼすべて問題にならなくなります。シリアライズによってアトミック性の障害がなくなり、`HttpSession` に潜在する同期化ピギーバックが可視性の障害をなくすからです。そして特定のセッションに結び付けられたリクエストをシリアライズしても、セッションでリクエストが完全に重なることはめったになく、ましてやセッションでのリクエストが数多く重なることは極めて稀であるため、スループットに大きな影響を与える可能性は極めて低いはずです。

残念ながら、サーブレット仕様では「同じセッションでのリクエストはシリアライズしなければならない」と規定していませんが、SpringMVC フレームワークではこのようなシリアライズを行うための方法を提供しています。その方法は、他のフレームワークにも簡単に実装できるものです。SpringMVC コントローラーの基本クラスである `AbstractController` にはブール変数、`synchronizeOnSession` が用意されています。このブール変数をセットすると、ロックが使用され、セッションで同時に実行できるリクエストが確実に 1 つだけとなります。

> オブジェクトを EDT (Event Dispatch Thread) に制限すると Swing アプリケーションでの同期化要件が減るのと同じように、`HttpSession` でリクエストをシリアライズすることで、並行性の多くの危険は消え去ります。

まとめ

ステートフルな Web アプリケーションの多くには、並行性に関する顕著な脆弱性があります。この脆弱性は、`HttpSession` や `ServletContext` などのスコープ・コンテナーに保管された可変データにアクセスするときに、十分な調整を行っていないことに起因しています。`getAttribute()` メソッドと `setAttribute()` メソッドに備わっている同期化で十分だと思い込みがちですが、それで十分なのは、属性が不変または実質上不変であったり、スレッド・セーフであったりする場合、あるいはコンテナーにアクセスする可能性のあるリクエストがシリアライズされる場合など、特定の状況下においてのみです。一般に、スコープ・コンテナーに配置するあらゆるものは、実質上不変であるか、スレッド・セーフでなければなりません。サーブレット仕様で規定されたスコープ・コンテナーのメカニズムは、自らを同期化しない可変オブジェクトを管理するようには意図されていないからです。最もよくないことは、通常のJavaBeans クラスを `HttpSession` に保管することです。この手法でも上手くいくと保証されるのは、JavaBean がセッションに保管された後には変更されることがない場合に限られます。

著者について

Brian Goetz

Brian Goetz はこれまで 20 年間、プロのソフトウェア開発者として活躍してきました。現在は Sun Microsystems のシニア・スタッフ・エンジニアで、複数の JCP Expert Group の一員でもあります。2006年5月、Addison-Wesley から彼の著書『Java Concurrency In Practice』が出版されています。人気の業界紙に掲載された Brian のこれまでの記事、そして今後の記事を参照してください。