JDBCモニタによるアプリケーション透過的ポリシエンフォースメント

吉澤武朗† 渡邊裕治† 沼尾雅之†

JDBC インタフェースに準拠したポリシベースアクセスモニタの構成法を提案する．JDBC は，Java でのリレーショナルデータベースへのアクセスインタフェースを規定しているが，ポリシ評価に必要なコンテクスト情報を受け渡す手段は規定されていない．そこで，アクセスモニタ機能を包含するドライバを JDBC の実装として提供し，コンテクスト情報をアプリケーション透過的に受け渡す仕組みを用意することで，アプリケーションロジックと，モニタにおけるポリシエンフォースメントのロジックを完全に分離し，データベースアクセスに対するアプリケーション透過的なポリシエンフォースメントを可能にする．本手法を用いれば，既存のアプリケーションに対して，そのコードに変更を加えることなくポリシエンフォースメント機能を付加することが可能となる．また，新規にアプリケーションを開発する場合においても，プログラムの複雑化を避けることができる．

JDBC Monitor for Application Transparent Policy Enforcement

Takeo Yoshizawa,† Yuji Watanabe† and Masayuki Numao†

This paper describes a construction method of policy based access monitor that is fully compliant with JDBC interface. JDBC provides the Java interfaces for accessing relational database. However, JDBC does not provide the way to transfer context information such as user who is accessing the data, or purpose of access, which are necessary for policy evaluation. We provide JDBC driver containing access monitor function as an implementation of JDBC interface, and transfer context information to the driver in an application transparent manner. In this way, it enables the separation of application logic and policy enforcement logic. By using this method, existing application that is not compliant with policy can be made policy compliant without changing its code. In the case of new application development, it reduces the complexity of the program to be compliant with policy.

1. はじめに

近年，個人情報保護法や Sarbanes-Oxley 法（SOX 法）など，企業における情報管理の徹底を義務付ける数多くの法律が施行されている．たとえば，多くの企業が収集している顧客情報などは，個人情報としてその利用が利用目的や顧客の同意などによって制御される必要がある．また，個人情報に限らず，特定の情報に対してはアクセスポリシ上アクセスが許可される利用者によるアクセスであっても，後の監査のためにそれを記録に残しておく必要がある．このように，現在企業は情報管理に際して多くのセキュリティ要件を満たす必要に迫られている．一方，企業に蓄積されているそれら管理すべき重要な情報の量は膨大であり，これらの適切な管理はもはや IT システムによるサポートなしには成立しえないのが現状である．にもかかわらず，これら重要情報を扱う既存の IT システムを構成するアプリケーションには，セキュリティ要件を満たしていないものも多い．したがって，そのようなアプリケーションは今後必要な要件に対応させていく必要があるが，通常これは既存のアプリケーションの変更を要する．しかしながら，既存のアプリケーションの変更は，コスト，運用面で困難な場合が多い．また，新規にアプリケーションを作成する場合にも，多くのセキュリティ要件を直接アプリケーションに組み込んでしまえば，アプリケーションロジックとセキュリティのためのロジックが混在し，アプリケーションが複雑化してしまう．一般にアプリケーションはその後の運用などにあわせて改訂などが行われるため，保守作業が必要である．さらに，アプリケーションの完成後にセキュリティ要件が変化する可能性もある．複雑化したアプリケーションの保守コストは膨大となり，場合によっては作り直さざるをえない場合もある．

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

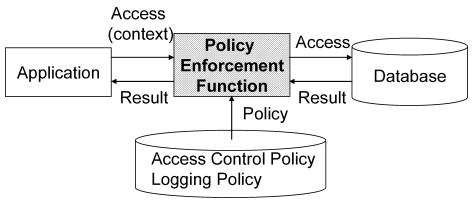

図 1 一般的なポリシエンフォースメントの構成（ISO/IEC 10181-3）
Fig. 1 Typical policy enforcement architecture (ISO/IEC 10181-3).

これらのことを考えると，一般には，ISO 認可モデル ISO/IEC 10181-3[1] にも定められているように，あらかじめセキュリティ要件を書き下したポリシを用意し，アプリケーションとデータの間にアクセス要求に対してポリシに従った処理を実行するためのポリシエンフォースメント機能を設置する**図 1** のような構成がとられる．一方，通常アクセス制御やログの記録などといったデータアクセスに対するポリシエンフォースメントには，「誰が，いつ，何に，どのような方法で，何の目的で」アクセスしているのかなどといったアクセス要求に対するコンテクスト情報が必要で，これらコンテクスト情報をポリシに対して評価し，その判定結果に基づいて定められた処理が実行される．通常，コンテクスト情報の多くはアプリケーション側に存在しており，ISO 認可モデルの構成でポリシエンフォースメントを実現するためには，アプリケーション側でコンテクスト情報を収集し，これをポリシエンフォースメント機能に渡す必要がある．通常，これらの処理は明示的にアプリケーションが実行する必要があるため，そのためのロジックがアプリケーションロジックに入り込む形になってしまう．すると，既存のアプリケーションに対するポリシエンフォースメント機能の導入には，アプリケーションの変更が必要となり，新規のアプリケーション開発であっても，アプリケーションコードが複雑化してしまうことになる．

ところで，現在の企業システムを構成するアプリケーションの多くが Java 言語によって記述されている．また，それらアプリケーションが利用するデータは通常データベースに格納されている．そこで本論文では Java 言語で記述されたデータベースアクセスを行うアプリケーションに対して，JDBC API のフック，コンテクストハンドラ，フィルタリングプロファイルという 3 つの仕組みによってアプリケーションロジックと，ポリシエンフォースメントロジックを分離する方法を提案することで，上記課題を克服するアプリケーション透過的ポリシエンフォースメントを実現することを目的とする[2]．

以降では，まず 2 章で本論文に関連する技術や研究を紹介し，それらでは実現できなかったデータベースアクセスに対するアプリケーション透過的なポリシエンフォースメントを実現する方法について 3 章で紹介する．続いて，その提案手法を実装し，実際のアプリケーションに適用した結果や，そのアプリケーションのパフォーマンスへの影響などについて 4 章で述べ，最後に 5 章で結論を述べる．

2. 関 連 技 術

アプリケーション透過的なポリシエンフォースメントを実現する技術は，各種実行環境に対してそれぞれ提案されている．たとえば青柳らは，OS のファイルシステムに対してアプリケーション透過的にアクセス制御を実行するための技術を提案している[3]．ここでは，OS のシステム API をフックし，その位置でアクセス制御に関する機能を実行することで，アプリケーションからその処理を隠蔽している．これは，本論文で提案している JDBC API のフックと同じであるが，OS レベルのプリミティブな処理に対するモニタであるため，アプリケーションレベルで意味付けされたコンテクスト情報を見つけにくくなっている．本論文では，アプリケーションと同じレベルの API である JDBC API をモニタすることによって，アプリケーションレベルのコンテクスト情報が得やすくなっている．

一方，データベースアクセスに対してアプリケーション透過的なポリシエンフォースメントを実現する例としては，JDBC API と（ほぼ）同じ API を持つ Java クラスを提供し，その中でアクセス制御機能を実行することにより，アプリケーションコードとアクセス制御ロジックの分離を図るものもある[4]．しかしながら，既存のアプリケーションに対しては通常の JDBC API の呼び出し部分のコードを，モニタクラスの呼び出しに変更するようなアプリケーションコードの書き換えが必要となる．また，コンテクスト情報の受け渡しに独自 API を設けており，それらは明示的にアプリケーション内で呼び出される必要があるため，ポリシエンフォースメントロジックの一部がアプリケーションに混在することになる．

また，データベース内においてポリシエンフォースメント機能を実現するもの[5] も，アプリケーションとは独立にポリシエンフォースメントの実行が可能である．たとえば，通常データベース自体にもアクセス制御やログの記録といった機能は備わっており，アプリケーションからデータベースにコンテクスト情報を渡

す処理以外はデータベース内に閉じた形で実行される．

しかしながら，実際の運用においては，データベース側でのポリシエンフォースメントはアプリケーションからのデータベースアクセスに対して十分に機能しない場合がある．たとえば前述のように，多くのコンテクスト情報は通常アプリケーション側に存在するが，アプリケーションからデータベースへ任意のコンテクスト情報を渡す一般的な仕組みは存在しない．したがって，データベース側で十分なポリシエンフォースメントが実行できない場合がある．また，アプリケーションのパフォーマンスに対する要求と，データベース側でのポリシエンフォースメントの仕組みの相性に起因する問題もある．多くのアプリケーションではデータベースアクセスに対するパフォーマンス向上のためにコネクションプーリングと呼ばれる技術が利用される．コネクションプーリングでは，重い処理であるデータベース接続を各アクセスごとに確立することを避けるために，一度使用された接続を保持し，再利用する．通常データベース接続には，データベースユーザが関連付けられ，この情報をもとにデータベース側でポリシエンフォースメントを行うが，異なるユーザに関連付けられた接続は再利用できないため，多くの異なるユーザがアプリケーションを利用するような場合には，ユーザごとにコネクションを生成する必要があり，パフォーマンスの低下が問題となる．この問題を避けるため，多くのアプリケーションでは，データベース接続は単一ユーザで実行してしまうような構成がとられる．すると，データベース側では実際にアクセスしてきているユーザが判別できず，ポリシエンフォースメントを実行できなくなるといった問題が生じる．

このように，既存のデータベースアクセスを対象とした技術においては，アプリケーション側に存在するコンテクスト情報をいかにアプリケーション透過的に収集し，ポリシエンフォースメント機能に伝達するかが課題である．

3. 提案技術

アプリケーション側にあるコンテクスト情報を用いてポリシエンフォースメントを実行するためには，少なくともアプリケーション側においてコンテクスト情報を収集し，ポリシエンフォースメント機能を呼び出す（収集されたコンテクスト情報の受け渡しを含む）必要がある．したがって，アプリケーション透過的なポリシエンフォースメントを実現するためには，少なくとも以下の条件 1，2 を満たす必要がある．また，ポリシエンフォースメントとしてアクセス制御を実行する場合には，条件 3 についても考慮する必要がある．

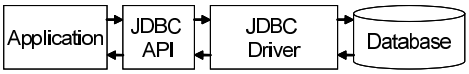

図 2 JDBC API
Fig. 2 JDBC API.

条件 1：ポリシエンフォースメント機能の呼び出しをアプリケーションが意識する必要がないこと
条件 2：コンテクスト情報の収集をアプリケーションが意識する必要がないこと
条件 3：アクセス制御の結果アプリケーションに渡されるデータがアプリケーションの動作に異常をきたさないこと

ここでは，具体的に上記の条件に対応して，以下の 3 つの仕組みを組み合わせることで，アプリケーション透過的なポリシエンフォースメントを実現する．

1： JDBC API のフックによるデータアクセスへの透過的介入
2： コンテクストハンドラによるアプリケーションロジック外でのコンテクスト情報の収集と引渡し
3： フィルタリングプロファイルによるアクセス制御結果の調整

以下では，それらの詳細について述べる．

3.1 JDBC API のフック

ここでは，関連研究にもあるように，既存の API の中に処理を隠蔽することによってアプリケーションからのデータアクセスに対する透過的介入を実現する．Java プログラムからのデータベースアクセスに対しては，JDBC API[6] が定義されており，本論文ではこの API を用いる．そこでまずこの JDBC API について説明し，その後にこのインタフェースを利用した具体的な構成について述べる．

3.1.1 JDBC API

JDBC API は，Java アプリケーションからデータベースにアクセスする際の標準インタフェースを規定している．アプリケーションはそのインタフェースのみを利用することで，種々あるデータベースの実装とは独立した形でデータベースアクセスを行うことができる．実装に依存した実際のデータベースアクセスは，この JDBC API を実装した JDBC ドライバによって行われる．この様子を**図 2** に示す．通常この JDBC ドライバは各データベースベンダなどから提供される．

この API を利用した一般的なデータベースアクセスの例を**表 1** に示す☆．まず，DataSource オブジェ

☆ 実際のコードに必要な例外処理や，リソースの開放処理は省略している．

表 1 サンプルアプリケーションコード
Table 1 Sample application code.

```
1  // データベースに接続する
2  DataSource ds
3  = (DataSource) ctx.
4  lookup("java:comp/env/jdbc/ds");
5  Connection conn = ds.getConnection();
6  Statement stmt = conn.createStatement();
7
8  // SQL クエリを送り，結果を得る
9  String sql
10 = "SELECT EMPNO, NAME, EDLEVEL,"+
11 " SALARY, JOB FROM EMPLOYEE";
12 ResultSet rs = stmt.executeQuery(sql);
13
14 // 結果を表示
15 while (rs.next()) {
16    System.out.print(rs.getString("EMPNO"));
17    System.out.print(rs.getString("NAME"));
18    // ... 中略...
19    System.out.println(rs.getString("JOB"));
20 }
```

クトを取得することで，接続対象となるデータベースを選択する（表 1 の 2〜4 行目）．次に，選択されたデータベースに対して接続を確立するための Connection オブジェクトを得（表 1 の 5 行目），その接続を介してデータベースに対してクエリを発行するための Statement オブジェクトを得る（表 1 の 6 行目）．その後，得られた Statement オブジェクトを用いてデータベースにクエリを発行し（表 1 の 9〜12 行目），その結果を ResultSet オブジェクトの形で得る（表 1 の 12 行目）．最後に得られた ResultSet オブジェクトから結果データを取り出すことでデータを得る（表 1 の 15〜20 行目）．

3.1.2 JDBC モニタの構成

前述のように，JDBC の実際の動作はその実装である JDBC ドライバに依存する．一方で，アプリケーションコードに記述されているのは JDBC のインタフェース（API）だけである．したがって**図 3** に示すように，本来の JDBC ドライバをラップするようなドライバをモニタとして提供し，ここにポリシエンフォースメント機能を導入することで，ポリシエンフォースメント機能の呼び出しをアプリケーションが意識する必要がなくなる．

一方で，前述のように，ポリシエンフォースメントを実行するためにはコンテクスト情報が必要となる．しかし，JDBC API はデータへのアクセスのためのインタフェースは規定しているが，任意のコンテクスト情報の受渡しをサポートしていないなど，場合によっては必要なコンテクスト情報をアプリケーションからモニタに渡すことができない．

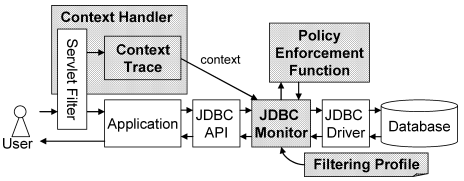

図 3 JDBC モニタのアーキテクチャ
Fig. 3 JDBC monitor architecture.

そこで，図 3 の左上に示すように，これらのコンテクスト情報の受渡しを担う別の機能を配置する．ここではこれを，コンテクストハンドラと呼ぶ．以下では，このコンテクストハンドラが，どのようにアプリケーション透過的にコンテクスト情報の収集および，それらのモニタへの受渡しを行うかについて記す．

3.2 コンテクストハンドラ

3.2.1 コンテクストトレース

前述の構成においては，ポリシエンフォースメントがモニタにおいて実行されるため，すべてのコンテクスト情報をモニタに渡す必要がある．しかしながら，必要なコンテクスト情報が取得可能なアプリケーションコード中の位置から複数のメソッドが入れ子状に呼び出され，その中で JDBC API が呼び出されているケースなどでは，コンテクスト情報をモニタに渡すことが困難となる．仮に，コンテクスト情報が取得可能な位置から，すべてのメソッド呼び出しの引数としてコンテクスト情報を引き渡して JDBC API 呼び出しの位置までコンテクスト情報を渡していくようなことをすれば，コンテクスト情報収集のためのロジックがアプリケーションロジックに混在することとなり，コードを複雑化させてしまう．また，適用対象が既存アプリケーションであった場合，多くのコードの変更が必要となる．

さらに前述のように，JDBC API は任意の一般的なコンテクスト情報を通すことができるようなインタフェースは提供しておらず，たとえば P3P [7] で記述されたプライバシポリシに見られるデータの利用目的などのコンテクスト情報を JDBC API を通して受け渡すことはできない．また前述のように，データベース接続でよく用いられるコネクションプーリングとの相性から，適切なコンテクスト情報が得られない場合もある．

これらの問題を解決するために，動作する任意のアプリケーションからアクセス可能な位置に，コンテクスト情報を格納するための機能である，コンテクストハンドラを設ける．アプリケーションは，コンテク

スト情報を，それが取得可能な位置からコンテクストハンドラに格納する．このようにしておいて，アプリケーションが JDBC API を呼び出し，モニタのコードが実行される際に，モニタが格納されたコンテクスト情報をコンテクストハンドラから取得することで，JDBC API やその他のメソッドコールチェーンを飛び越えてコンテクスト情報を取得することができる．しかしながら，コンテクストハンドラには複数のトランザクションに対するコンテクスト情報が格納されうるため，モニタがコンテクストハンドラからコンテクスト情報を取得する際には，今実行されているデータアクセスに対して適切なコンテクスト情報を選択する必要がある．そのために，ここではコンテクストハンドラにコンテクスト情報が格納される際に，アプリケーションからそのトランザクションに関連する識別子を取得し，コンテクスト情報と関連付けておく．その後，モニタのコードが実行される段階で，同様にその識別子をアプリケーションから取得し，それをキーにコンテクストハンドラからコンテクスト情報を取得することで，適切なコンテクスト情報を得る．

本論文では具体的に，**表 2** に示すように，各実行環境に合わせてスレッド ID やスレッド ID とセッション ID の組合せを識別子として用いる．

まず，シングルスレッドで実行されるような一般的なローカル Java アプリケーションの場合には，スレッド ID を識別子として用いる．アプリケーションがコンテクストハンドラにコンテクスト情報を渡す際に，コンテクストハンドラが背後で動作しているスレッドを自動的に取得し，その ID をコンテクスト情報と関連付けて格納しておく．その後，モニタにおいて実行中のスレッド ID をキーにコンテクストハンドラからコンテクスト情報を取得することで，同一スレッドに関連付けられたコンテクスト情報を適切に取得する☆．ただし，この場合，適用対象のアプリケーションが表 2 のような適用条件を満たしている必要がある．一方 Web アプリケーションの場合，複数のスレッドにまたがってトランザクションが実行され，この適用条件を満たさないケースがある．そこでこの場合はセッション ID を併用する☆☆．具体的には，アプリケーションにおいてセッション ID およびコンテクスト情報が得られた時点で，そのアプリケーションの実行スレッドの ID とセッション ID の対応と，そのセッション ID とコンテクスト情報の対応をコンテクストハンドラに記録しておく．ただし，スレッドプールの利用を考慮し，すでに同じスレッド ID と他のセッション ID とのマッピングが存在する場合にはそれを消去する．その後，モニタにおいては実行中のスレッド ID をキーに先ほどの対応付けからセッション ID を取得し，得られたセッション ID に関連付いたコンテクスト情報を取得することで，適切なコンテクスト情報を取得することができる．また，終了したセッションの ID が再利用されることもあるため，終了したセッションに対するマッピングは消去しておく．ただし，この場合においても表 2 のような適用条件を満たしている必要がある．したがって，ここで提案した方法だけでは，単一ユーザの処理がアプリケーションの途中で複数のスレッドに分散し，その先でデータベースアクセスが行われるようなマルチスレッド環境や，リモート EJB の中でデータベースアクセスが行われるアプリケーションのように，コンテクスト情報が取得可能な実行環境と，データベースアクセスが行われる（JDBC API が呼び出される）実行環境が異なるような環境など，いくつかの環境には対応できない．しかしながら，ここで用いた識別子は一般的な環境において汎用的に利用可能な識別子のみであり，一般的な構成のアプリケーションの多くは表 2 の条件を満たしていることから，この方法は多くのアプリケーションに適用可能であるといえる．

表 2 コンテクストトレースのサポート環境
Table 2 Applicable scope of context trace.

主な適用対象	利用識別子	適用条件
ローカル Java アプリケーション（主にシングルスレッドで動作するもの）	スレッド ID	コンテクスト情報が対応付けられるスレッドと，アプリケーションの中でデータベースアクセスを実行する際のスレッドが，同一トランザクション内で等しいことが保証されている．
Web アプリケーション（J2EE 環境上で動作し，リモート EJB 利用なし）	スレッド ID ＋ セッション ID	1 つのトランザクションが 1 つのセッション ID で一意に特定可能で，セッション ID に対応付けられる 1 つ以上のスレッドと，アプリケーションがデータベースアクセスを実行する際のスレッドが，同一トランザクション内で等しいことが保証されている．
その他	N/A	N/A

3.2.2 透過的コンテクスト情報の取得

これまで述べた方法により，コンテクスト情報をモニタ部で利用するためのロジックの大半はアプリケーションから分離することができるが，実際には一部，つまりアプリケーションからコンテクストハンドラにコンテクスト情報を渡す部分がアプリケーションロジックに残ることになる．通常この部分の分離は困難

☆ スレッドプールを用いている場合でも同一スレッドは同時に 1 つしか動作しないため，同一スレッド ID に関連付けられるコンテクスト情報を上書きすることで対応が可能．

☆☆ ここではセッションの安全性はサーバやアプリケーションによって確保されていることを前提としている．

であるが，アプリケーションが J2EE コンテナ[8] 上で動作しているような場合には，この部分もアプリケーションから隠蔽することができる．J2EE とは，Java の機能セットの 1 つで，サーバ上で動作するアプリケーションに必要な機能を含んでいる．この J2EE を利用した Web アプリケーションなどは，特定のフレームワークにのっとった形で実装されており，すべてのアプリケーションが J2EE コンテナの上で動作する．J2EE コンテナ中では，サーブレットフィルタと呼ばれる機能を用いることができ，これを用いれば（サーブレットや JSP の形で実装される）アプリケーションの特定の処理の前に，アプリケーションコードから見えない位置であらかじめ指定した処理を実行することができる．そこで，サーブレットフィルタ中にコンテクスト情報を取得するコードを埋め込むことで，アプリケーション透過的にコンテクスト情報をコンテクストハンドラに渡すことができる．これによって，ユーザインタフェースからアプリケーションロジックに渡るようなコンテクスト情報（ユーザ ID やアクセス目的など）を，アプリケーション呼び出しの直前で取得することができる．

3.3 フィルタリングプロファイル

上記の方法により，アプリケーションからのデータベースへのアクセスに対して，モニタにおいてアプリケーション透過的にポリシ評価を実行することができるようになった．一方，この評価結果に基づいてアクセス制御を実行する場合，アクセスが拒否されたデータに対しては，そのデータがそのままアプリケーションに渡されないように，モニタにおいて返されるデータを書き換える必要がある．しかしながら，たとえば，データベースから返されるデータが null であってはならないなど，アプリケーションがデータベースアクセスの結果として得られるデータに何らかの前提を置いているような場合，モニタにおいてアクセスが拒否されたデータを，その前提から外れるようなデータに書き換えてしまうとその後のアプリケーションの動作に異常をきたす場合があり，アプリケーション透過的なポリシエンフォースメントが実現できない．そこで本論文では，各データ型ごとにアクセスが拒否された際に返す結果を設定できるようにすることで，データに置かれた前提を満たす形でアクセス制御を実行できるようにする．ここではこのようにアクセスが拒否された際のモニタの振舞いの設定をフィルタリングプロファイルと呼ぶ．

しかしながら，データベースから返されるデータをその後の何らかの計算に用い，その計算が正しいデータに基づいて行われていることを前提とするアプリケーションなど，データに対する前提が固定的でないアプリケーションに対しては，この方法では対応しきれない．この場合，本論文の手法では完全にアプリケーション透過的にアクセス制御を実現することはできないが，アクセス拒否が発生した際に明示的に例外（Exception）を発生させるように，フィルタリングプロファイルにおいて設定できるようにすることで，アクセス拒否の結果をアプリケーションに通知する汎用的な方法を提供し，ポリシエンフォースメントロジックとアプリケーションロジックの混在を極小化する．

3.4 JDBC モニタの適用可能範囲

上記の提案技術によって完全にアプリケーション透過的にポリシエンフォースメントを実行するためには，アプリケーションが表 2 の条件を満たし，J2EE コンテナ上で動作し，アプリケーションがデータベースから取得するデータに対してあらかじめ記述しきれないような前提を持たないことが必要となる．しかしながら，これらの前提条件はデータベースの情報を単純に参照・表示するような Web アプリケーションの多くが満たすものであり，そのような Web アプリケーションは企業システムにおいて広く用いられている．したがって，本論文の技術によってアプリケーション透過的にポリシエンフォースメントが実現可能となるアプリケーションは多数存在すると考えられる．

4. 実システムでの運用

4.1 プロトタイプ構成

本論文で提案する JDBC モニタのプロトタイプを作成した．このプロトタイプは，JDBC 経由のデータベースアクセスを汎用的にフックできるように，モニタインタフェースを提供する．アプリケーション開発者は，このインタフェースを実装することで任意のポリシエンフォースメント機能を実現することができる．現在のプロトタイプでは，アクセス制御機能とログ取得機能を実装している．これを**図 4** に示す．この実装では，前述の提案手法を用いてユーザ ID，SQL をコンテクスト情報としてモニタに渡し，これに応じてモニタインタフェースを実装している各機能が実行される．ログ取得機能では，アクセスのあった時間，ユーザ ID，実行した SQL を記録する．アクセス制御機能では，以下の 2 つのステップによりアクセス制御を実行する．まず，与えられた SQL を，SQL Analyzer によってアクセスの種類やアクセス対象カラムといった，より詳細なコンテクスト情報に分解し[9]，これらとユーザ ID を事前に定められたポリシに対して評価

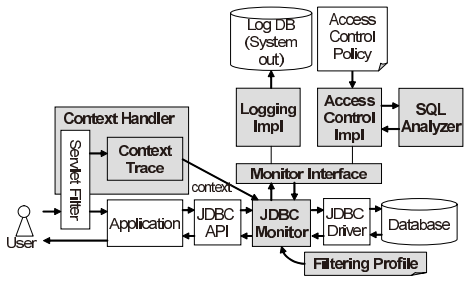

図 4 プロトタイプの構成
Fig. 4 Prototype system.

表 3 既存アプリケーションの出力
Table 3 Output of the existing application.

ID	NAME	EDLEVEL	SALARY	JOB
000001	EVA	16	36,170.00	MANAGER
000002	HEATHER	18	28,480.00	ANALYST
000003	IRVING	16	32,250.00	MANAGER
000004	JAMES	16	20,450.00	DESIGNER

する．その後，その結果（およびフィルタリングプロファイルの設定内容）に応じて，SQL（SELECT 文）の実行結果である ResultSet オブジェクトがアプリケーションに返す結果を書き換える．

4.2 システムの実行例

ここで，このプロトタイプシステムを使って，ポリシエンフォースメント機能を持たず，単に与えられた SQL に対してデータを返すだけのアプリケーションが，本論文で提案する JDBC モニタを用いて実際にどのようにポリシエンフォースメント機能を獲得するかについて説明する．ここで用いるサンプルの既存アプリケーションのコードは，表 1 に示したものとする．このアプリケーションは，データベース中の EMPLOYEE テーブルから従業員の社員番号（EMPNO），名前（NAME），教育レベル（EDLEVEL），給与（SALARY），役職（JOB）を取得し，これを表示する．

このアプリケーションの出力は，**表 3** のようになる．特にアクセス制御はなされず，与えられたクエリから得られるすべてのデータを返すことになる．また，この実行結果に対するログの記録もない．

このアプリケーションに対して，本論文で提案した方法を適用する．モニタは，EDLEVEL と SALARY へのアクセスを監視しており，データ利用者として一般従業員とマネージャを設定する．ポリシとしては，「ID，NAME，JOB は誰でも読み取ることができる」，「一般従業員とマネージャは，マネージャの SALARY を読み取ることができる」，「マネージャは，すべての従業員の EDLEVEL を読み取ることができる」という 3 つのステートメントからなるものを設定し，明示的に許可されないデータにはアクセスできないものとする☆．このポリシのもと，一般従業員がデータにアクセスしたときのアプリケーションの出力結果は，**表 4** のようになる．EDLEVEL カラムと，SALARY カラムのうち，同レコードの JOB カラムの値が MANAGER でないものについてアクセスが拒否されていることが分かる．ここでは，フィルタリングプロファイルによって数値に対するアクセスが拒否された場合 0 を返すように設定しているが，任意の値に設定が可能である．一方，マネージャがアクセスしたときの結果は，**表 5** のようになる．一般従業員の場合と異なり，EDLEVEL が見えている．また，追加したログ機能により，アクセスのあった時間，アクセスユーザ名，実行した SQL などが記録される．

表 4 一般従業員のアクセス結果
Table 4 Output of the access by an ordinary employee.

ID	NAME	EDLEVEL	SALARY	JOB
000001	EVA	0	36,170.00	MANAGER
000002	HEATHER	0	0	ANALYST
000003	IRVING	0	32,250.00	MANAGER
000004	JAMES	0	0	DESIGNER

表 5 マネージャのアクセス結果
Table 5 Output of the access by a manager.

ID	NAME	EDLEVEL	SALARY	JOB
000001	EVA	16	36,170.00	MANAGER
000002	HEATHER	18	0	ANALYST
000003	IRVING	16	32,250.00	MANAGER
000004	JAMES	16	0	DESIGNER

ここで，既存のアプリケーションのコードはまったく変更していない．既存のシステムとの相違は，使用している JDBC ドライバが今回提案するモニタに設定されている点と，Web アプリケーションに対してサーブレットフィルタを追加し，その部分でコンテクスト情報を収集しているという点のみである．

4.3 パフォーマンス評価

本論文の方式では，アプリケーションコード上ではポリシエンフォースメント機能を意識する必要はないが，実際のアプリケーションの処理にポリシエンフォースメントのための処理が追加される形になる．したがって，ポリシエンフォースメント機能の実行によって処理の負荷が増加することになる．そこで，JDBC モニタを利用した場合の処理負荷がどの程度になるのかを見るために，前述のプロトタイプを用いて以下のよう

☆ 実際には機械判別可能なポリシ言語を用いて記述する．

なパフォーマンスの測定を行った．

ここでは，図 4 の構成に対して，データベースとして DB2 V8.2 FP12，アプリケーションサーバとして WebSphere Application Server Enterprise Edition v5.1（テスト環境）を用い（図 4 のデータベースとユーザアクセス以外はすべてこのアプリケーションサーバ上で動作する），これらを 1 台の PC（CPU: Pentium 4 3 GHz，Memory: 1 GB）上で動作させる．一方ユーザのアクセスは別の PC（CPU: Pentium M 1.8 GHz，Memory: 1 GB）から行う．これらのシステム構成の上で，以下の 4 つの構成で同一のアプリケーションを実行し，それぞれの構成ごとにアクセス結果レコード数に対するレスポンスタイムの応答を見る．

構成 1：JDBC ドライバとして通常のもの（DB2 用の JDBC Type2 ドライバ）を用いる構成（図 2）

構成 2：JDBC ドライバとして構成 1 のドライバをラップした JDBC モニタを利用し，ポリシエンフォースメント機能は実行しない構成（図 4 において，Logging Impl，Access Control Impl の動作をなくしたもの）

構成 3：構成 2 に，ユーザ名と実行した SQL をログとして標準出力に出力するログ機能を追加した構成（図 4 において Access Control Impl の動作をなくしたもの）

構成 4：構成 3 に加え，ユーザ名に応じてセルレベルのアクセス制御☆を実行する構成（図 4）

ここでのアプリケーションは，単純なサーブレットと JSP を用いた Web アプリケーションで，ユーザ名と SQL を入力として受け付け，その SQL の実行結果をブラウザに表示するというものである．このユーザ名と SQL の入力を送信してから，その結果が返るまでの時間を一般的なパフォーマンス測定ツールを用いて計測した．この結果を**図 5** に示す．

この結果から，まず，構成 1 と構成 2 のレスポンスタイムの差が非常に小さく，JDBC モニタによってアプリケーションからのデータベースアクセスをフックする処理は非常に軽いことが分かる．アクセスレコード数が増加するにつれてレスポンスタイムの差がわずかにではあるが増加するのは，アクセスレコード数が増加した分データをアプリケーションが取り出すための命令（ResultSet インタフェースのメソッドの実行）数が増加し，結果としてフックする処理が多く実行されるからであると考えられる．しかしながら，一般的な Web アプリケーションなどにおいては，それほど大量のデータを一度に取り出しそれぞれ表示するような処理はあまり多くない．したがってこのレスポンスタイムの差が問題になることはほぼないと考えてよい．また，構成 3 においても，ほとんどレスポンスタイムの悪化は見られない．これはここで追加したログ機能が非常に単純かつ実行時間の短いものであるためであると考えられる．

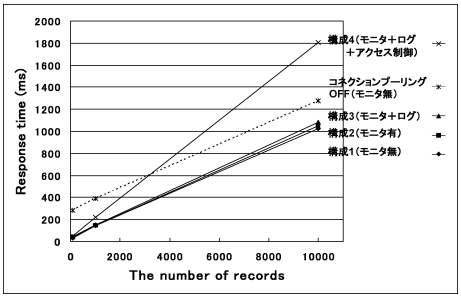

図 5 各構成ごとのアクセス結果レコード数に対するレスポンスタイムの応答

Fig. 5 Response time response to the number of records for each construction.

一方，構成 4 においてアクセス制御機能を付加すると，レスポンスタイムが約 1.8 倍と非常に処理が重くなる．ここでのアクセス制御の実装方式では，アプリケーションからの SQL クエリの実行後，アプリケーションからの ResultSet のアクセスに対してポリシを評価し，各セルごとにアクセスの可否を判定している．したがって，アプリケーションから ResultSet オブジェクトに対するアクセス数が増加するほど処理の負荷が増すことになる．通常，得られたデータを単に表示するようなアプリケーションにおいては，一度にそれほど多くのレコードを表示することは少ないが，アクセスされるレコード数が多い場合には問題となる．このレスポンスタイムの悪化を防ぐためには，アクセス制御ロジックの実装方法をチューニングする必要がある．たとえば，アプリケーションからの SQL クエリを，データベースに送る前にその実行結果がポリシの評価結果に合致するように書き換えておく方法[5] や，セルレベルのアクセス制御に必要な条件判定部を，同じく SQL の書き換えによって効率化する方法[10] などの適用が考えられる．ただし，本論文ではこれらポリシエンフォースメント機能の実装に関しては規定しない．

また 2 章で，データベース側でポリシエンフォースメントを実行する方法では，ポリシエンフォースメント機能がコネクションプーリングと共存できないケースがあることを述べた．前述の実験環境においては，

☆ 前節のアクセス制御実行例のように，テーブルの行と列の交点（セル）のレベルでアクセスを制御すること．

コネクションプーリングが機能していない場合はコネクションプーリングが機能している場合よりも 1 回のデータベース接続で約 250 ms のレスポンスタイムの遅延が見られた．したがって，仮にデータベース側でポリシエンフォースメントを実行することを前提とし，そのプラットフォームとしてのパフォーマンスを計測すれば，図 5 の点線のような結果となり，かなり大きなレコード数に対しても本論文のポリシエンフォースメントのプラットフォームとしての性能を示す構成 2 よりも実行時間が余分に必要となることになる．

5. おわりに

本論文では，JDBC API のフック，コンテクストハンドラ，フィルタリングプロファイルといった主に 3 つの技術を提案し，データベースアクセスに対するアプリケーション透過的なポリシエンフォースメントが，ある条件下で実現可能であることを示した．また，実験によりそれらのパフォーマンスに対する影響も分析し，一般的な Web アプリケーションなどへの適用可能性を確認した．今後，適用可能範囲の拡大や，パフォーマンスの向上を図り，より実用的な技術としていくことを展望としたい．

参考文献

1) ISO/IEC 10181-3:1996: Information technology — Open Systems Interconnection — Security frameworks for open systems: Access control framework, International Organization for Standardization and International Electrotechnical Commission (1996).
2) 吉澤武朗，渡邊裕治，百合山まどか，沼尾雅之：JDBC インタフェースに準拠したポリシーベースアクセスモニタの構成法，CSS2004, pp.595–600, 情報処理学会 (2004).
3) 青柳慶光，鮫島吉喜：機密ファイル持出し防止システムの検討，CSS2002, pp.59–64, 情報処理学会 (2002).
4) 相馬仁志，市川範子，近藤誠一，松田昇平，松岡恭正：メタデータを用いたデータアクセス制御ミドルウェアの開発，日本データベース学会 Letters, Vol.2, No.3, pp.29–32 (Dec. 2003).
5) Agrawal, R., Kiernan, J., Srikant, R. and Xu, Y.: Hippocratic Databases, *28th International Conference on Very Large Data Bases* (2002).
6) JDBC Technology. http://java.sun.com/javase/technologies/database/index.jsp
7) Cranor, L., Langheinrich, M., Marchiori, M. and Reagle, J.: The platform for privacy preferences 1.0 (P3P1.0) specification. W3C Recommendation (2002).
8) Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/javaee/index.jsp
9) 沼尾雅之，渡邊裕治，百合山まどか：顧客データアクセスにおけるプライバシーポリシーの展開，SCIS2004, pp.137–142, 電子情報通信学会 (2004).
10) 吉澤武朗，渡邊裕治，沼尾雅之：モニタによるデータベースセキュリティのためのクエリ書き換え手法，SCIS2006, 3C3-2, 電子情報通信学会 (2006).

(平成 18 年 11 月 27 日受付)
(平成 19 年 6 月 5 日採録)

吉澤 武朗

平成 15 年東京大学大学院工学系研究科精密機械工学専攻修士課程修了．同年日本アイ・ビー・エム株式会社入社．東京基礎研究所副主任研究員．プライバシ保護，トレーサビリティ，ビジネスプロセスモデリングに関する研究開発に従事．

渡邊 裕治（正会員）

平成 13 年東京大学大学院工学系研究科電子情報工学専攻博士課程修了．同年日本アイ・ビー・エム株式会社入社．東京基礎研究所主任研究員．平成 16 年度情報処理学会論文賞受賞．トラステッドコンピューティング，ネットワークセキュリティ，プライバシ保護，トレーサビリティに関する研究開発に従事．博士（工学）．

沼尾 雅之（正会員）

昭和 58 年東京大学大学院工学系研究科電子工学専攻修士課程修了．同年日本アイ・ビー・エム株式会社入社．同社東京基礎研究所にてロジックプログラミング，エキスパートシステム，データマイニング，電子商取り引きシステム等の研究を経て，現在，ビジネストランスフォーメーション技術担当．ネットワークセキュリティ，プライバシ保護，トレーサビリティおよびシステムモデリングやビジネスプロセスモデリングに関する研究開発に従事．博士（情報理工学）．