Javaの理論と実践: 良いハウスキーピング習慣を身につける

不必要なリソースが残っていませんか

Brian Goetz
Principal Consultant
Quiotix

2006年 3月 21日

ガーベジ・コレクションは、Java™プラットフォームの機能として誰にも好まれています。ガーベジ・コレクションによって開発が単純になり、あらゆるカテゴリーの潜在的コード・エラーを無くすことができます。しかし、確かにガーベジ・コレクションを利用すればリソース管理を無視できる場合が多いのですが、時には自分で掃除（ハウスキーピング）をする必要があります。今回の『Javaの理論と実践』では、Brian Goetzが、ガーベジ・コレクションの限界と、自分で掃除すべき場合について解説します。

このシリーズの他の記事を見る

私達は子供の頃、遊び終わったおもちゃは片づけるように両親に言われたものです。よく考えてみると、そうした小言の理由は、単純に部屋をきれいにしておきたいという抽象的な願望以上に、家の中の床面積は限られており、おもちゃだらけになると他の目的に使えない（例えば歩き回る、など）という現実的な制限からだったのだと思います。

充分な面積があれば、片づける意欲は下がります。そして、面積に反比例して、きれいにしようという意欲は減退するものです。Arlo Guthrieの有名なバラード、『Alice's Restaurant Massacre』の一節では、正にこれが歌われています。

> ・・・あんな大きな部屋があるし、長椅子は全部外に出してしまったし、この先ずっと、自分たちのガラクタは捨てずに済む・・・

良きにつけ悪しきにつけ、ガーベジ・コレクションのおかげで、私達は片づけることを怠けがちです。

明示的にリソースを解放する

Javaプログラムで使われるリソースの圧倒的大部分はオブジェクトです。そしてガーベジ・コレクションは、オブジェクトを片づける上では適切な仕事をします。そこで、皆さんは好きなだけStringsを使います。やがてガーベジ・コレクターは、いつそれらが使い終わったのかを判別し、皆さんが何ら手助けしなくても、使われたメモリーは再利用されます。

その一方、メモリーではないリソース、例えばファイル・ハンドルやソケット・ハンドルなどは、close() やdestroy()、shutdown()、release() といった名前を持つメソッドを使って、プログラムによって明示的に解放する必要があります。一部のクラス、例えばプラットフォーム・クラス・ライブラリーの中のファイル・ハンドル・ストリーム実装などは、「セーフティー・ネット」としてファイナライザーを提供しています。もしプログラムがリソース解放を忘れた場合にも、プログラムがリソース解放を終了したとガーベジ・コレクター判断すると、ファイナライザーが解放作業を行ってくれます。しかし、たとえファイル・ハンドルがファイナライザーを提供しており、皆さんが忘れた場合にもクリーンアップを行ってくれるにしても、ファイル・ハンドルを使い終わったら明示的に閉じた方が、やはり好ましいのです。明示的に閉じることによって、他の手段を待つよりも早く閉じることができ、リソース不足が発生しにくくなります。

一部のリソースでは、ファイナライザーが解放するまで待つことが選択肢となり得ないことがあります。仮想リソース (例えばロック取得やセマフォー許可など) の場合、LockやSemaphoreは、「時既に遅し」となるまでガーベジ・コレクションされません。またデータベース接続などのようなリソースでは、ファイナライザーを待っていると、まず確実にリソースが足りなくなります。多くのデータベース・サーバーでは、(ライセンス容量に基づいて) 一定数の接続しか受け付けません。もしサーバー・アプリケーションが各リクエストに対して新しいデータベース接続を開いてしまい、使い終わってもそのまま放置してしまうと、既に必要なくなった接続をファイナライザーが閉じるよりもずっと前に、そのデータベースは容量に達してしまう可能性が高くなります。

メソッドに直結されたリソース

大部分のリソースは、アプリケーションのライフタイム中ずっと保持されているわけではありません。むしろ、あるアクティビティーのライフタイム期間中に保持されるのです。アプリケーションがファイル・ハンドルを開き、ある文書を処理するための読み込みを行う場合、通常はファイルからの読み取りが終われば、アプリケーションにはファイル・ハンドルが必要なくなります。

最も単純な場合では、リソースは取得され、使用され、そして、できれば同じメソッド・コールの中で解放されます。例えばリスト1のloadPropertiesBadly() メソッドのような場合です。

リスト1.単一メソッドの中で不適切にリソースを取得し、使用し、解放する・・・こんなことをしてはいけません

```
    public static Properties loadPropertiesBadly(String fileName)
            throws IOException {
        FileInputStream stream = new FileInputStream(fileName);
        Properties props = new Properties();
        props.load(stream);
        stream.close();
        return props;
    }
```

残念ながら、この例はリソース・リークを起こす可能性があります。すべてがうまく行けば、メソッドが戻る前にstreamは閉じられます。しかし、もしprops.load() メソッドがIOExceptionを投げたとすると、(ガーベジ・コレクターがファイナライザーを実行するまで) streamは閉じられません。これを解決するためには、try...finally機構を使って、何が起きても確実にstreamが閉じられるようにします (リスト2)。

リスト2.単一メソッドの中で適切にリソースを取得し、使用し、解放する

```
public static Properties loadProperties(String fileName)
        throws IOException {
    FileInputStream stream = new FileInputStream(fileName);
    try {
        Properties props = new Properties();
        props.load(stream);
        return props;
    }
    finally {
        stream.close();
    }
}
```

リソース取得（ファイルのオープン）が、tryブロックの外にあることに注意してください。もしtryブロックの中に置かれたとすると、たとえリソース取得が例外を投げたとしてもfinallyブロックは実行されます。この手法は不適切なばかりではなく（取得していないリソースを解放することはできません）、finallyブロックの中のコードは恐らく独自の例外（NullPointerExceptionなど）を投げてしまいます。finallyブロックから投げられる例外は、そのブロックの終了を引き起こした例外を置き換えます。つまり元々の例外は失われ、デバッグ作業には役に立たなくなります。

必ずしも見た目ほど容易ではない

あるメソッドの中で取得されたリソースを、finallyを使って解放することは信頼性が高いのですが、複数のリソースが関係してくると面倒なことになります。例えば、JDBCのConnectionを使ってクエリーを実行し、ResultSetを繰り返すメソッドを考えてみてください。このメソッドは、Connectionを取得し、それを使ってStatementを作成し、Statementを実行するとResultSetが得られます。しかし中間JDBCオブジェクト（StatementとResultSet）は独自のclose() メソッドを持っており、これらを使い終わった後には解放しなければなりません。しかし、「当然」クリーンアップを行うと思える方法（リスト3）では、うまく行かないのです。

リスト3.複数リソースを解放しようとして失敗する例・・・こんなことをしてはいけません

```
public void enumerateFoo() throws SQLException {
    Statement statement = null;
    ResultSet resultSet = null;
    Connection connection = getConnection();
    try {
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM Foo");
        // Use resultSet
    }
    finally {
        if (resultSet != null)
            resultSet.close();
        if (statement != null)
            statement.close();
        connection.close();
    }

}
```

この「ソリューション」がうまく行かない理由は、ResultSetとStatementのclose() メソッド自体がSQLExceptionを投げられるためです。そのために、finallyブロックの中にある、後の方のclose() ステートメントが実行されないのです。これに対処するための選択肢は幾つかありますが、どれも

面倒です。つまり、各close() をtry..catchブロックでラップする、try...finallyブロックをリスト4のようにネストする、あるいは、リソース取得と解放を管理するための一種のミニ・フレームワークを書く、などです。

リスト4.複数リソースを解放するための、信頼性の高い（ただし面倒な）方法

```
public void enumerateBar() throws SQLException {
    Statement statement = null;
    ResultSet resultSet = null;
    Connection connection = getConnection();
    try {
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM Bar");
        // Use resultSet
    }
    finally {
        try {
            if (resultSet != null)
                resultSet.close();
        }
        finally {
            try {
                if (statement != null)
                    statement.close();
            }
            finally {
                connection.close();
            }
        }
    }
}

private Connection getConnection() {
    return null;
}
```

ほとんどが例外を投げる

私達は誰も、重量級のオブジェクト（データベース接続など）を解放するためにはfinallyを使うべきだということを知っています。しかし、finallyを使ってストリームを閉じる、などとなると、それほど細心の注意は払いません（何と言っても、ファイナライザーが面倒を見てくれるのです）。また、リソースを使うコードがチェックあり例外（checked exception）を投げない場合にも、finallyを使うことを忘れがちです。リスト5は、バウンド・コレクション（bounded collection）に対するadd() メソッドの実装を示しています。これはSemaphoreを使ってバウンドを強制し、スペースが利用可能となるまでクライアントが効率良く待てるようにしています。

リスト5.脆弱なバウンド・コレクション実装・・・こんなことをしてはいけません

```
public class LeakyBoundedSet<T> {
    private final Set<T> set = ...
    private final Semaphore sem;

    public LeakyBoundedSet(int bound) {
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = set.add(o);
        if (!wasAdded)
            sem.release();
        return wasAdded;
    }
}
```

LeakyBoundedSetはまず、許可が利用可能（コレクションの中にスペースがあることを意味します）となるまで待ちます。次にコレクションに要素を追加します。もし、その要素が既にコレクションの中にあるために追加処理が失敗すると、（実際には予約したスペースを使わなかったので）LeakyBoundedSetは許可を解放します。

LeakyBoundedSetの問題点は、必ずしもすぐには明確になりません。しかし、もしSet.add() が例外を投げたとしたらどうでしょう。このシナリオは、追加される要素や既にSetの中にある要素に対するSet実装に欠陥がある場合、あるいはequals() 実装やhashCode() 実装（あるいは、SortedSetの場合はcompareTo() 実装）などに欠陥がある場合に起こります。当然ながら、この問題に対するソリューションは、finallyを使ってセマフォー許可を解放することです。これは非常に簡単なのですが、実は非常に忘れられがちなのです。こうした間違いがテスト中に発見されることは稀なため、時限爆弾を抱えたようなことになります。リスト6は、もっと信頼性の高いBoundedSet実装です。

リスト6.Semaphoreを使って確実にSetをバウンドする

```
  public class BoundedSet<T> {
    private final Set<T> set = ...
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        }
        finally {
            if (!wasAdded)
                sem.release();
        }
    }
}
```

FindBugs（参考文献）のようなコード監査ツールを利用すると、不適切なリソース解放インスタンス（メソッドの中でストリームを開きながら、閉じるのを忘れている、など）を検出することができます。

様々なライフサイクルを持つリソース

様々なライフサイクルを持つリソースに関しては、Cを使っていた頃に逆戻りします。つまり、手動でリソースのライフサイクルを管理するのです。一部のサーバー・アプリケーション（マルチプレーヤー・ゲーム・サーバーなど）では、セッション期間中はクライアントがサーバーに永続的なネットワーク接続を行うため、ユーザー単位で取得されたリソース（ソケット接続を含む）は、ユーザーがログアウトしたら必ず解放する必要があります。こうした場合には、きちんと整理しておけば良いのです。つまり、ユーザー単位リソースへの参照がActiveUserオブジェクトの中のみに保持されていれば、（明示的に、あるいはガーベジ・コレクションによって）ActiveUserが解放された時にリソースを解放することができます。

様々なライフサイクルを持つリソースは、ほとんど必ず、どこかにあるグローバル・コレクションの中に保存されます（あるいはグローバル・コレクションからアクセス可能です）。従ってリソース・リークを防ぐためには、いつリソースが不要となったのか、このグローバル・コレクションから、いつ削除すべきかを判断することが非常に重要です。（前回の記事、「弱参照でメモリー・リークを塞ぐ」では、そのための手法を幾つか紹介しています。）この時点では、リソースは間もなく解放されることが分かっているため、リソースに関連付けられた非メモリー・リソースも、その時に同時に解放することができます。

リソースの所有権

タイミング良くリソースを解放するための手法の鍵は、所有権に関して厳密な階層構造を維持することです。所有権には、リソースを解放する責任を伴います。アプリケーションがスレッド・プールを作成し、そのスレッド・プールがスレッドを作成した場合、スレッドはプログラムの終了前に解放が必要な（終了を許されている）リソースです。しかしアプリケーションには、このスレッドの所有権はなく、スレッド・プールが所有権を持っています。従って、このスレッドの解放については、スレッド・プールが責任を持つ必要があります。当然ですが、アプリケーションがスレッド・プール自体を解放するまで、スレッド・プールはスレッドを解放することはできません。

所有権の階層構造、つまり各リソースが自分の取得するリソースを所有し、またその解放に責任を持つという構造を維持することによって、制御不可能な混乱を防ぐことができます。こうしたルールを適用する場合には、ガーベジ・コレクションのみでは解放できない各リソースは、何らかのライフサイクル・サポート（close() メソッドなど）を提供する必要があります。（こうしたリソースの中には、ガーベジ・コレクションのみでは解放できないリソースを直接的あるいは間接的に所有するリソースも含まれます。）

ファイナライザー

プラットフォーム・ライブラリーが、開いたファイル・ハンドラーをクリーンアップするためのファイナライザーを提供しているのであれば、明示的に閉じるのを忘れる危険性が大幅に減少するはずです。では、なぜもっと頻繁にファイナライザーが使われないのでしょう。幾つかの理由

がありますが、一番の理由は、ファイナライザーを正しく書くことが非常に面倒なためです（そして非常に容易に、誤ったものを書きやすいのです）。正しくコード化することが難しい上に、ファイナライズのタイミングは決定論的ではなく、またファイナライザーが正しく実行できるという保証すらないのです。またファイナライズによって、インスタンス化や、ファイナライズ可能オブジェクトのガーベジ・コレクションにオーバーヘッドが加わります。ファイナライザーをリソース解放のための主要手段にすべきではないのです。

まとめ

ガーベジ・コレクションは、非常に多くのクリーンアップをしてくれます。しかし、一部のリソース（ファイル・ハンドルやソケット・ハンドル、スレッド、データベース接続、セマフォー許可など）では、相変わらず明示的な解放が必要です。リソースのライフタイムが特定なコール・フレームのライフタイムに直結している場合には、finallyブロックを使ってリソースを解放できることが多いのですが、もっとライフタイムの長いリソースを確実に解放するためには、何らかの戦略が必要です。明示的な解放が必要なオブジェクトを直接的、間接的に所有するオブジェクトに対しては、ライフサイクル・メソッド（例えばclose() やrelease()、destroy() など）を提供して、信頼性の高いクリーンアップを行う必要があります。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。