サーブレットにおける static

シングルトン・パタンで説明するように、static キーワードはデータベース接続プールのような共有資源では不可欠と言える。しかしながら static キーワードを付したできたインスタンスを、はたして JVM は唯一のものとして扱ってくれるのだろうか？実際はそんなに簡単なものではなく、**static なクラスやインスタンスの範囲はあるクラスローダのなかでのみ有効**なのである。このことは特に複数のクラスローダが存在する Tomcat のようなサーブレット・コンテナや EJB サーバでは要注意である。この問題に関しては、例えば Ted Neward 氏が書いた Javageeks.com のホワイト・ペーパなど幾つかの資料が詳しく取り扱っている。ここではそのポイントのみを実験を含めて説明したい。

1. static で指定されたクラス変数は本当に唯一なのか？

Neward が示している事例をあげよう。まず次のような Dummy というクラスを考えよう。これには staticCount というクラス変数が存在する。従って複数回コンストラクタが呼ばれれば、この変数は呼ばれるたびにインクリメントされる。

```
/**
 * これまでに作られたインスタンスの数をプリントするだけのシンプルなクラス
 */
public class Dummy
{
public static int staticCount = 0;
public Dummy()
{
staticCount++;
System.out.println("Instance #" + staticCount + " constructed.");
}
public int getStaticCount()
{
return staticCount;
}
}
```

例えば次のようなコードを考えよう:

```
import java.io.*;
import java.net.*;
/**
 * シンプルなテスト--3 つの Dummy オブジェクトとをインスタンス化し staticCount を観察する
 * 予想されるようにスタティックなフィールドがインクリメントされる
 */
```

```
public class StaticTest
{
public static void main (String args[]) throws Exception
{
new Dummy();
new Dummy();
new Dummy();
}
}
```

このコードを実行すると、コンソールには予想どおり次のような出力が表示される:

```
C:\Projects\Papers\JavaStatics\src>java StaticTest
Instance #1 constructed.
Instance #2 constructed.
Instance #3 constructed.
C:\Projects\Papers\JavaStatics\src>
```

しかしこの Dummy クラスを次のようなコードでアクセスしたらどのような結果になるだろうか？

```
import java.io.*;
import java.net.*;
/**
 * 3 つの分離した ClassLoader に同じクラス（statics つき）をロードする
 * このクラスが報告する staticCount がどうなるかを調べる
 */
public class StaticTestClassLoader
{
public static void main (String args[]) throws Exception
{
URL[] url = { new File("subdir").toURL() };
URLClassLoader cl1 = new URLClassLoader(url);
URLClassLoader cl2 = new URLClassLoader(url);
URLClassLoader cl3 = new URLClassLoader(url);
cl1.loadClass("Dummy").newInstance();
cl2.loadClass("Dummy").newInstance();
cl3.loadClass("Dummy").newInstance();
}
}
```

このコードの実行結果は次のようなものになる:

```
C:\Projects\Papers\JavaStatics\src>md subdir
C:\Projects\Papers\JavaStatics\src>move Dummy.class subdir
C:\Projects\Papers\JavaStatics\src>java StaticTestClassLoader
Instance #1 constructed.
Instance #1 constructed.
Instance #1 constructed.
C:\Projects\Papers\JavaStatics\src>
```

つまりクラスローダが異なると static で指定した要素はもはや唯一なもので無くなってしまう。このようになってしまっている理由は、Sun の技術者たちによれば、別のクラスローダを使うことで同じ JVM に同じクラスの異なったバージョンが存在できるようにするためだという。スタティックなインスタントとそれをロードした ClassLoader を結びつけることで、「稼動させたままの」コードの更新が出来る。すなわち上の例で cl1.loadClass("Dummy").newInstance()とその後の cl2 が呼ばれるまでの

間でディスク上の Dummy クラスに変更が生じたとすると、古いバージョンが cl1 のかたちで JVM に共存していても cl2 は Dummy の新しいバージョンを取得できる。

上の例では 3 つの URLClassLoader のインスタンスが各々 Dummy というクラス・ファイルを探すが、そのインスタンスはまずそれを親の ClassLoader である AppClassLoader に委譲する。AppClassLoader は CLASSPATH からコードをロードする役割を持っているが、そのクラス更にその上の ClassLoader のトリーをたどっても見つからなかったのでそのクラスは見つからないと報告してくる。その後この新しいインスタンスは自分自身をチェックする。そのサブディレクトリに Dummy.class があることを見つけらら、これをロードし staticCount というインスタンスを作る。問題はその後で作られる cl2 や cl3 といったクラスローダも同じような動作をするがいずれも同じレベルのクラスローダをチェックしないで、親しか調べないことである。その結果各々のクラスローダが新しい staticCount を生成してしまうのである。

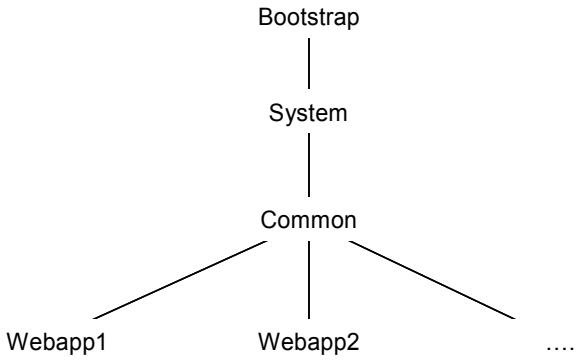
2. サーブレット・コンテナの自動再ロードとクラスローダ

上記のような事態はサーブレットの自動再ロード機能(auto reloading)をもつサーブレット・コンテナで発生する。自動再ロードはまさしくあるアプリケーションを動作させたままでそれに使われているクラスがアップデートできる機能である。ディスク上のサーブレットのクラスが変更されると、たとえこのサーブレットの古いバージョンが他のスレッドで使用中であっても、サーブレット・コンテナは自動的にこのサーブレットを再ロードする。そのためには新しいサーブレットは別のクラスローダでロードされねばならない。

従って**自動再ロードをオンにしたサーブレット・コンテナでは、スタティックなインスタンスを使うときは十分な配慮が必要になる**。

3. サーブレット・コンテナのクラスローダ

例えば Tomcat や J2EE のフレームワークでは幾つかのクラスローダが階層化されて使われている。これらはやはり Java 2 の委譲パタン(delegation pattern)が採用されている。以下は Tomcat 6 の階層である。Tomcat 5 からはかなり簡素化されている（Server と Shared という階層がない）ので、注意が必要である。



Tomcat 6 のクラスローダたち

Bootstrap	JVM に組み込まれている。ベーシックな Java クラス・ライブラリのクラスをロードする。このローダは boot クラス・パスにあるクラスのみをロードする。これらのクラスは安全なのでロードのプロセスは簡素化され高速である。Tomcat では加えてシステム拡張ディレクトリ ($JAVA_HOME/jre/lib/ext)にある Jar ファイルのすべてのクラスを対象とする。
System	通常は一般の CLASSPATH の環境変数の内容で初期化される。しかしながら Tomcat では代わりに以下のリポジトリから System クラスローダを構築する: • $CATALINA_HOME/bin/bootstrap.jar - Tomcat 6 サーバを初期化するための main()メソッドとそれに属するクラスローダ実装クラスを含む • $CATALINA_HOME/bin/tomcat-juli.jar – Jakarta commons logging API と名前が変えられたパッケージと java.util.logging LogManager
Common	このクラスローダは Tomcat の内部クラスと総てのウェブ・アプリケーションから見える付加的クラスが含まれる。通常はアプリケーションのクラスはここに置くべきではないと解説に書かれている。このクラスローダからは$CATALINA_HOME/lib にある総てのパックされていないクラスとリソース、及び Jar ファイルの総てのクラスとリソースが可視である。デフォルトとしては以下のものがある: annotations-api.jar – JEE のアノテーションのクラス catalina.jar – Tomcat 6 の Catalina サーブレット・コンテナ部分の実装 catalina-ant.jar - Tomcat Catalina Ant タスク catalina-ha.jar – 利用性の高いパッケージ catalina-tribes.jar – グループ通信パッケージ el-api.jar - EL 2.1 API. jasper.jar - Jasper 2 コンパイラとランタイム jasper-el.jar - Jasper 2 EL 実装 jasper-jdt.jar - Eclipse JDT 3.2 Java コンパイラ jsp-api.jar - JSP 2.1 API. servlet-api.jar - Servlet 2.5 API. tomcat-coyote.jar - Tomcat コネクタとユーティリティのクラス **tomcat-dbcp.jar - Commons DBCP に基づいて名前が変えられたデータベース接続プール** tomcat-i18n-**.jar – 他の言語の為のリソース・バンドルを含むオプションの Jar たち。デフォルトのバンドルはまたここの Jar に含められるので、メッセージの国際化が不要な場合はこれらは安全に外すことが出来る。
WebappX	各アプリケーションが自分のプライベートなクラスの為に持つローダと考えればよい。このクラスローダは単一の Tomcat 6 インスタンスに配備された各ウェブ・アプリケーション毎につくられる。WEB-INF/classes のディレクトリにある総てのパックされていないクラスとリソース、加えてウェブ・アプリケーションのアーカイブの為の/WEB-INF/lib にある Jar ファイルにある総てのクラスとリソースが含まれているウェブ・アプリケーションから可視になるが、それ以外には可視にはならない。このクラスローダは Servlet 仕様書の第 2.3 版の 9.7.2 説のウェブ・アプリケーション・クラスローダに従い、デフォルトの Java 2 の委譲モデルとは異なっている。このウェブ・アプリケーションの WebappX のクラスローダからのあるクラスのロード要求が処理されるとき、このクラスローダはまずローカルなレポジトリを調べる。例外があって、JRE ベース・クラスの部分であるクラスはオーバライドできない。いくつかのクラス(JDK 1.4 以降の XML パーサ・コンポーネントなど)については、JDK 1.4 の Endorsed 機能が使える。最後に、Servlet API のクラスを含む Jar ファイルすべてをこのクラスローダは無視する。Tomcat 6 の他のクラスローダはすべて通常の委譲パターンに従う。

注意しなければいけないのは、親のローダがロードしたインスタンスからは子供のローダがロードしたインスタンスを直接的にアクセスできないことである。その逆は可能である。

もう一つ注意しなければならないのは、**Common は tomcat-dbcp.jar というデータベースの接続プールのクラスが可視になっており、これがこのメモのポイントになっている部分である。**これは後ほどまとめのところで説明したい。

WebappX のローダは通常の委譲モデルと違い、要求されたクラス・ファイルがまず自分のローカルなレポジトリを探し、無ければ上位のクラスローダにこれを委譲する。これはサーブレット仕様書（例えば 2.3 版の 9.7.2）に次のように書いてあるからである：
「アプリケーションのクラスローダは、WAR のなかにパッケージされたクラスとリソースのほうを、コンテナにわたるライブラリの Jar たちよりも先にロードするよう実装されることが勧告される。」

そのほうが確かに一般的には全体としてロードの速度が上がることが期待される。

4. Tomcat 6 の自動再ロード

自動再ロードで問題になるのは、**自動再ロードが出来るコンテナでは、WebappX の階層のローダが、あるアプリケーションのなかで新しくなる**ことである。昔の O'Reilly の解説（http://www.unix.com.ua/orelly/java-ent/servlet/ch03_02.htm）では次のように書いてあった：
「サーバがサーブレットに要求を送るときに、そのサーバは先ずそのサーブレットのクラス・ファイルが変更されていないか調べる。もし変更されていれば、このサーバは古いバージョンのロードに使われたクラスローダを放棄し、新しいバージョンをロードするためにクラスローダの新しいインスタンスを作る。古いサーブレットのバージョンはメモリ上に無期限に駐在できる（そして他のクラスは古いサーブレットのインスタンスへの参照を維持するので、思いがけない副作用を引き起こす）が、古いバージョンは新たな要求を処理する為にはもはや使われない。」

筆者の以前の経験では Tomcat は新しい要求が来るたびではなくて、自分が管理するクラス・パスのクラスに変更がでたかを定期的に検知していた。

最近のサーブレット仕様書（2.3 版）は次のように書いている：
「SRV.3.7 再ロードの考察：コンテナの提供者は開発し易いようにクラスの再ロードの機能を実装することを要求されていないが、そのような実装は総てのサーブレット及びそれらが使うクラスの総てが単一のクラスローダのクラス検出範囲内に無ければならない。この要求は該アプリケーションがデベロッパが期待したとおり動作するのを保証するために必要である。開発の手助けとして、セッション・バインディング・リスナへの通知の完全な意味がコンテナによってサポートされ、クラスロードによるセッションの終了を監視するのに使われるようにするべきである。これまでの世代のコンテナは、サーブレットをロードするのに新しいクラスローダを生成していたが、該サーブレット・コンテキストで使われる他のサーブレットまたはクラスをロードするために使われるクラスローダとは異なっ

ていた。このことがあるサーブレット・コンテキスト内のオブジェクト参照が予期せぬクラスまたはオブジェクトを指し、予期せぬ動作を引き起こす可能性がある。この要求は新しいクラスローダの要求発生で引き起こされる問題を防止するために必要である。」

「SRV.9.8 ウェブ・アプリケーションの置き換え：サーバは該コンテナを再開始しないであるアプリケーションを新しいバージョンに置き換えることが出来ねばならない。あるアプリケーションが置き換えられたときに、該コンテナはそのアプリケーションのセッションのデータが保持されるしっかりした方法を提供しなければならない。」

なお Tomcat の Q&A では以下の事柄が書かれている：
「クラスローダ（及びそれがロードした Class オブジェクト）はリサイクルされない。これらは JVM がつくった恒久ヒープにストアされ、アプリケーションを再配備するときには新しいクラスローダがつくられ、これらの総てのクラスのもうひとつのコピーをロードする。このことは最終的に OufOfMemoryErrors を起こし得る。」

5. Tomcat 6 での実験

さて仕様書のこの要求が Tomcat 6 でどのように実装されているのだろうか？実際に調べてみよう。
Tomcat 6 で再ロードを可能にするためには、CATALINA_HOME/conf/context.xml のなかの
<Context>
を
<Context reloadable="true">
に変更する。この状態で Tomcat が稼動中に既にインスタンス化されているクラス・ファイルを更新すると、DOS 画面には次のような表示がされるはずである。

```
2008/09/08 12:57:37 org.apache.catalina.core.StandardContext reload
情報: このコンテキストの再ロードを開始しました
```

次のような StaticTest というサーブレットをコンパイルして CATALINA_HOME\ROOT\WEB-INF\classes のディレクトリにおいて実験して見よう。

```
/**
 * Dummy をアクセスしてインスタンス数を返すサーブレット
 */
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class StaticTest extends HttpServlet
{
      public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
      {
```

```
      res.setContentType("text/html; charset=Shift_JIS");
      PrintWriter out = res.getWriter();
      out.println("<HTML>");
      out.println("<HEAD><TITLE>Static Test</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<BIG>v1..Instant count : " + new Dummy().getStaticCount()
+ "  Thread : " + Thread.currentThread().toString() + " Time : "+ new
Date().toString() + "<BR></BIG>");
// 30秒間のスリープ中に再ロード実験をするためのコード
      try {
            Thread.sleep(30000);
      } catch (InterruptedException e) {      }
      out.println("<BIG>v1..Instant count : " + new Dummy().getStaticCount()
+ "  Thread : " + Thread.currentThread().toString() + " Time : "+ new
Date().toString()  + "</BIG>");
      out.println("</BODY></HTML>");
      }
}
```

このサーブレットは http://localhost:8080/servlet/StaticTest とアクセスを繰り返すたびに生成した Dummy のインスタンスの数を表示する。30 秒間の再ロード実験部分を外すと 1 つずつ Dummy のインスタンスが増え、そうしないと以下のように 30 秒の間隔を置いてひとつづつ計 2 つインスタンスが増える:

```
v1..Instant count : 1 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:52:35 JST 2008
v1..Instant count : 2 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:53:05 JST 2008
```

この 30 秒間の間にこのサーブレットのクラス・ファイルを v1 から v2 に更新するとどうなるだろうか。

```
ブラウザ上の表示:
v1..Instant count : 3 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:55:51 JST 2008
v1..Instant count : 4 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:56:21 JST 2008
Tomcat の表示:
2008/09/19 10:56:04 org.apache.catalina.core.StandardContext reload
情報: このコンテキストの再ロードを開始しました
```

このことは、**たとえ途中（この場合は 10 時 56 分 04 秒）でクラス・ファイルが更新されても、スレッドが使用中のときはこのサーブレットおよび Dummy のオブジェクトが維持される**ことを意味している。その後 http://localhost:8080/servlet/StaticTest とブラウザでアクセスすると、以下のようにインスタントは新しく 1 からインクリメントしてしまう。

```
v2..Instant count : 1 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:58:49 JST 2008
v2..Instant count : 2 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 10:59:19 JST 2008
再度アクセス:
v2..Instant count : 3 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:02:15 JST 2008
v2..Instant count : 4 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:02:45 JST 2008
```

以上のことから**自動再ロードにより Dummy のオブジェクトが新たに作られ、その Static な staticCount もリセットされることがわかる。つまり WebappX の新しいクラスローダが Dummy をロードした為、2 つ目の staticCount というスタティックな変数が生成されたことを意味する。**

次に Dummy.class ファイルを Common の階層のクラスローダにロードさせる為に、CATALINA_HOME\ROOT\WEB-INF\classes から$CATALINA_HOME/lib に移動させてみよう:

まず http://localhost:8080/servlet/StaticTest とブラウザでアクセスすると 30 秒後に以下の応答が返ってくる:
v1..Instant count : 1 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:29:04 JST 2008
v1..Instant count : 2 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:29:34 JST 2008

次にブラウザの更新をクリックして、応答が返る前に StaticTest.Class を v2 のバージョンに変更すると次のように、変更されないままの状態で v1 サーブレットからの応答が返ってくる:
v1..Instant count : 3 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:30:44 JST 2008
v1..Instant count : 4 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:31:14 JST 2008

次のメッセージは Tomcat のコンソールに表示されたもので、11 時 30 分 52 秒に再ロードが開始されたことを意味する:
2008/09/19 11:30:52 org.apache.catalina.core.StandardContext reload
情報: このコンテキストの再ロードを開始しました

その後ブラウザの更新をクリックすると 30 秒後に次のような新しい v2 のサーブレットからの応答が返ってくる:
v2..Instant count : 5 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:32:18 JST 2008
v2..Instant count : 6 Thread : Thread[http-8080-1,5,main] Time : Fri Sep 19 11:32:48 JST 2008

この応答では Instant count は前の状態から継続されていて、リセットされていないことがわかる。

このように、**スタティックなクラス・ファイルは$CATALINA_HOME/lib に置かないと、自動再ロードで更新されてしまう**ことが理解されよう。

6. Tomcat 6 のサービス・マネージャの再配備

Tomcat のサービス・マネージャはアプリケーションの配備や削除をコンテナを止めることなく行うことが出来る。しかしながら、この機能はアプリケーション単位であり、そのアプリケーションに含まれるスタティックな要素に対しては特に留意していない。従って自動再ロードを使う場合は注意が必要である。つまりスタティックなクラス・ファイルは WAR に含めてはいけない。

7. まとめ

- static な要素はあくまでもこれをインスタンス化したクラスローダの範囲内でのみ唯一である。
- **自動再ロードを使う Tomcat の環境では、シングルトンのようなスタティックな要素を含む再ロードで更新されてはいけないクラス・ファイルは、通常のウェブ・アプリケーションのクラスローダの範囲に置いてはいけない**。一瞬でもその範囲にコピーすると別のインスタンスが生成されるので、Eclipse のような開発環境を使う場合は細心の注意を要する。
- そのようなファイルは$CATALINA_HOME/lib に置くべきである。
- サーブレット・コンテナのクラスローダのところで説明したように、**Common というクラス・ローダは tomcat-dbcp.jar が可視になっているのは以上のような理由による**。
- 自動再ロードは取扱いに注意を要するので、なるべく使わないほうが好ましい。

参考:シングルトン(Singleton)

デザイン・パタンのなかでも良く利用されるのがシングルトン・パタンである。これはいつでも唯一のあるシングルトンのインスタンスしか存在しないようにするもので、一般的には複数のインスタンスやスレッドが共有する資源(リソース)として利用される。データベースのコネクション・プールがその典型的な使い方である。

構造的には、デフォルトのコンストラクタをプライベートにし、他のクラスが直接的にインスタンスか出来ないようにする。このオブジェクトを返すスタティックなメソッドが用意される。このメソッドはクラス・レベルのメソッドで、オブジェクトを生成することなくアクセスできるようになる。

標準的なコードは次のようになる:

```
class SingletonClass {

      private static SingletonClass singletonObject;
      /** プライベートなコンストラクタにして他のクラスがインスタンス化するのを防止*/
      private SingletonClass() {
            //     Optional Code
      }
      /** スタティックなアクセスのメソッドで、このオブジェクトをかえす*/
      /** 同期化することで複数のスレッドのアクセスを防止*/
      public static synchronized SingletonClass getSingletonObject() {
            if (singletonObject == null) {
                  singletonObject = new SingletonClass();
            }
            return singletonObject;
      }
      /** このコードはクローンを使って複数のインスタンスが作られるのを防止する*/
      public Object clone() throws CloneNotSupportedException {
            throw new CloneNotSupportedException();
      }
}
```

シングルトンを使う各クラスは以下のようにこれを生成する:

```
            SingletonClass obj = SingletonClass.getSingletonObject();
```

実装によってはオブジェクトがガベージ・コレクションの対象になって、せっかくのデータが消滅す

る可能性がある。従って何時においてもアプリケーションが走っている間は必ずこのオブジェクトが参照されているようにしなければならない。

注:この内容はオンラインのチュートリアルである Java Beginner.com の Java Singleton Design Pattern という記事などを参考にしている。

参考：SimpleDateFormat や Formatter のスレッド対策

スレッド安全でないクラスの対策にも static が使われることが多いので、注意したい。

java.text.SimpleDateFormat というクラスは非常に有用な API で良く使われているが、**スレッド安全でない例外的な API のひとつであるため、良く問題を起こすことでも有名**である。サーブレット・コンテナ上で使用するときも、きちんとスレッド安全な使い方をしないといけない。スレッド安全でない他のクラスには：

数字用の NumberFormat、DecimalFormat、および DecimalFormatSymbols

日付や時間用の DateFormat、SimpleDateFormat、および DateFormatSymbols

テキスト・メッセージ用の MessageFormat、および ChoiceFormat

がある。

更に同じ問題を持っているのが 1.5 版から導入された Formatter（フォーマッタ）というクラスと Formattable というインターフェイスである。Formatter は C を知っているに人にはわかりやすい printf に似た書式付の文字列出力用のインタープリタである。書式を指定して、ある値（数値、日付、時刻など）を文字列に変換する。

しかしながら同じく 1.5 版で導入された **PrintStream.format/printf（及びこれを使っている System.out.printf や System.err.printf など）及び String.format は、Formatter を暗黙的に実装しているにもかかわらず、スレッド安全である**ので、極力これらを使用することをお勧めする。

1. 同期化による対策

ドイツのソフトウエアのコンサルタントの Steve McLeod による Java Tip #8: Thread-safe Alternatives to Java's SimpleDateFormat class という記事（http://www.solidsimplesafe.com/view/13）では、幾つかの解決スタイルが示されている。その中には安全でないもの、org.apache.commons.lang.time.FastDataFormat というスレッド安全だが機能が制限されているクラスを使うもの、及び同じくスレッド安全な org.joda.time.format.DateTimeFormat というサード・パーティのクラスを使うものなども紹介されているが、ネーティブな API を使ったものとしては次の 2 つがある。

最初の方法は df というスタティックでプライベートなインスタンスを用意し、それに対し同期化されたスタティックなメソッドを用意するものである。ユーザはこの format というメソッドをアクセスする。Format 以外のメソッドを使う時はそれに応じた同期化されたメソッドを幾つか用意すれば良い。同期化されたメソッド間では排他制御がされる:

```
//safe - uses synchronization
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

private static DateFormat df=SimpleDateFormat.getInstance();

public synchronized static String format(Date date){
      return df.format(date);
}
```

この例で使われている getInstance()というメソッドはデフォルトの日付と時間共に SHORT のスタイルを使うフォーマッタのインスタンスを取得するものである。そうでなくてパタンとロケールを指定したいときは new キーワードを使って SimpleDateFormat のコンストラクタを呼べばよい。

紹介されているもうひとつの方法はこのクラスのメソッドを使うときにその都度新しいインスタンスを作る方法であるが、これはその分処理時間を要するのであまりお薦めできない:

```
// safe - always creates new instances of SimpleDateFormat
import java.util.Date;
import java.text.SimpleDateFormat;

public static String format(Date date){
    return SimpleDateFormat.getInstance().format(date);
}
```

Steve McLeod の方法以外に、同期ブロックを使う方法がある:

```
//safe - uses synchronized block
import java.text.SimpleDateFormat;

public static SimpleDateFormat df=new SimpleDateFormat(); //designate
pattern and locale here if necessary

// ……..

synchronized (df){
```

```
      // use df only within this synchronized block
}
```

これは df というオブジェクトを使うコードを必ず同期ブロックに含めるものである。同期ブロックは何個存在してもかまわない。df というオブジェクトのロックはひとつしか存在せず、ブロック間で排他制御がされる。処理能力を下げない為に、同期ブロックはなるべく小さくし、かつブロック数を多くしないことが好ましい。この方法はうっかり df を使うコードをブロック外に置いてしまう可能性があるので、注意が必要である。コンパイラはそれを間違いだと指摘してくれない。

そのようなミスが心配な人は、多少処理能力が落ちても df オブジェクトをブロック内で毎回生成することをお薦めする:

```
//safe - always creates new instances of SimpleDateFormat within a
synchronized block
import java.text.SimpleDateFormat;


// ……..

synchronized (this){
      SimpleDateFormat df=new SimpleDateFormat(); //designate pattern and
locale here if necessary
      // use df only within this synchronized block
}
```

この場合は同期ブロックはロックを取得するオブジェクトは特にないので this としておく。

2.　スレッド毎に DateFormat のオブジェクトを持つ方法

もうひとつの対策は、同期化でスレッドを待たせるのではなくて、ThreadLocal を利用してスレッド毎にスレッド安全でないクラスのコピーをもつ方法である。これは The Java Specialists' Newsletter の 172 号に Dr. Heinz M. Kabutz が寄稿したものが参考になる。

これはかなり高度なコードである。SoftReference を含まない ThreadLocal<DateFormat>のオブジェクト tl で先ず考えてみると、スレッドは DateConverter をインスタンス化することで DateFormat 型のデータを持つ tl を持ち、testConvert というメソッドを呼ぶと、getDateFormat でその自分の

DateFormat オブジェクトを呼び出し、それの parse メソッドで入力文字列を解析し、format メソッドで目的の文字列に変換する。そのスレッドが存在している限り、たとえそのスレッドが再度 DateFormat を使うことが無いにしても、この tl オブジェクトは存在し続ける。SoftReference を使用したのは、これを配慮したためである。SoftReference はメモリー要求に応じてガベージ・コレクタの判断でクリアされるソフト参照オブジェクトである。もしガベージ・コレクタによってメモリ上から削除されていたときには、getDateFormat メソッドは新しく SoftReference を作成している。

```
import java.lang.ref.SoftReference;
import java.text.*;
import java.util.Date;

public class DateConverter {
  private static final ThreadLocal<SoftReference<DateFormat>> tl
      = new ThreadLocal<SoftReference<DateFormat>>();

  private static DateFormat getDateFormat() {
    SoftReference<DateFormat> ref = tl.get();
    if (ref != null) {
      DateFormat result = ref.get();
      if (result != null) {
        return result;
      }
    }
    DateFormat result = new SimpleDateFormat("yyyy/MM/dd");
    ref = new SoftReference<DateFormat>(result);
    tl.set(ref);
    return result;
  }

  public void testConvert(String date) {
    try {
      DateFormat formatter = getDateFormat();
      Date d = formatter.parse(date);
      String newDate = formatter.format(d);
      if (!date.equals(newDate)) {
        System.out.println(date + " converted to " + newDate);
      }
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

testConver というメソッドの代わりにプログラマたちは必要なコードを自分のアプリケーションにあわせて書けばよい。これにより format あるいは parse のメソッドでのスレッド競合問題を気にしなくて済む。