Thread-Specific Storage

Bogdan Dumitriu

Department of Computer Science
University of Utrecht

October 26, 2005

Outline

1. Thread-Safe Storage in Java
   - Sample Application
   - Discussing ThreadLocal
   - Using ThreadLocal
2. Discussion
   - Participants
   - When to use?
   - Some examples
3. Summary
   - Benefits/Drawbacks
   - Questions

Imagine that...

Context:

- multi-threaded application using version control
- originally, anonymous access was allowed
- then, the admin changed his/her mind
- code has to be changed to support authentication

Class used to access the repository

class RepositoryOps

```
public void checkOut() {
  System.out.println("checking out...");
}

public void update() {
  System.out.println("updating...");
}

public void commit() {
  System.out.println("committing...");
}

// ...
```

Application uses multiple MyThread threads

class MyThread extends Thread

```java
private RepositoryOps repoOps;

public MyThread(RepositoryOps repoOps) {
  this.repoOps = repoOps;
}

public void run() {
  String threadId = "Thread " + getId();
  System.out.println(threadId + " checking out...");
  repoOps.checkOut();
  System.out.println(threadId + " working...");
  System.out.println(threadId + " updating...");
  repoOps.update();
  System.out.println(threadId + " committing...");
```

The application

Main class

```
public static void main(String args[]) {
  RepositoryOps repoOps = new RepositoryOps();

  new MyThread(repoOps).start();
  new MyThread(repoOps).start();
  new MyThread(repoOps).start();
}
```

Adding authentication (1)

- Now, let's add authentication
- First step: create the User object

Adding authentication (1)

- Now, let's add authentication
- First step: create the User object

Data about a user

User class

```
static public final User ANONYMOUS_USER =
  new User("anonymous");

private String name;

public User(String name) {
  this.name = name;
}

public void setName(String name) {
  this.name = name;
}

public String getName() {
```

User rights

User class

```
public boolean canCommit() {
  if (name.equals(ANONYMOUS_USER_NAME)) {
    return false;
  }
  else {
    return true;
  }
}

public boolean canUpdate() {
  return canCommit();
}

public boolean canCheckOut() {
```

Basic solution

- Immediate solution: add user field to MyThread

class MyThread extends Thread

```
private User user;

public void setUser(User user) {
  this.user = user;
}
```

- But what if MyThread cannot be changed for whatever reason?

Basic solution

- Immediate solution: add user field to MyThread

class MyThread extends Thread

```
private User user;

public void setUser(User user) {
  this.user = user;
}
```

- But what if MyThread cannot be changed for whatever reason?

Basic solution

- Immediate solution: add user field to MyThread

class MyThread extends Thread

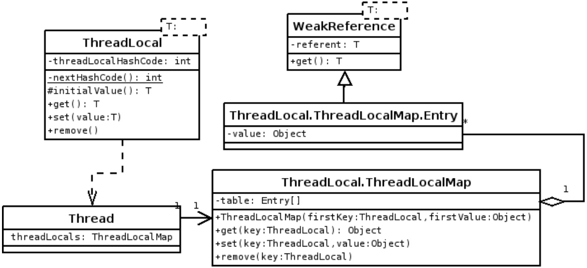
```
private User user;

public void setUser(User user) {
  this.user = user;
}
```

- But what if MyThread cannot be changed for whatever reason?

Adding authentication (2)

- Second step: intordcing the `ThreadLocal<T>` class



ThreadLocal
T:
-threadLocalHashCode: int
-nextHashCode(): int
#initialValue(): T
+get(): T
+set(value:T)
+remove()

WeakReference
T:
-referent: T
+get(): T

ThreadLocal.ThreadLocalMap.Entry
-value: Object

Thread
threadLocals: ThreadLocalMap

ThreadLocal.ThreadLocalMap
-table: Entry[]
+ThreadLocalMap(firstKey:ThreadLocal,firstValue:Object)
+get(key:ThreadLocal): Object
+set(key:ThreadLocal,value:Object)
+remove(key:ThreadLocal)

Getting thread local values

ThreadLocal class

```
public T get() {
  Thread t = Thread.currentThread();
  ThreadLocalMap map = t.threadLocals;
  if (map != null)
    return (T) map.get(this);

  T value = initialValue();
  t.threadLocals =
    new ThreadLocalMap(this, firstValue);
  return value;
}
```

Setting thread local values

ThreadLocal class

```
public void set(T value) {
  Thread t = Thread.currentThread();
  ThreadLocalMap map = t.threadLocals;
  if (map != null)
    map.set(this, value);
  else
    t.threadLocals =
      new ThreadLocalMap(this, firstValue);
}
```

Adding authentication (3)

- Third step: change RepositoryOps to use authentication

Adding thread local variable to RepositoryOps

class RepositoryOps

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>();
```

class RepositoryOps

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>() {
    @Override
    protected User initialValue() {
      return User.ANONYMOUS_USER;
    }
  };
```

Adding thread local variable to RepositoryOps

class RepositoryOps

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>();
```

class RepositoryOps

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>() {
    @Override
    protected User initialValue() {
      return User.ANONYMOUS_USER;
    }
  };
```

Adding thread local variable to RepositoryOps

class RepositoryOps

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>() {
    @Override
    protected User initialValue() {
      int threadId = Thread.currentThread();
      // lookup User based on threadId -> user
      return user;
    }
  };
```

Adding security to RepositoryOps

class RepositoryOps

```
public void checkOut() {
  if (user.get().canCheckOut()) {
    System.out.println("checking out...");
  }
}
public void update() {
  if (user.get().canUpdate()) {
    System.out.println("updating...");
  }
}
public void commit() {
  if (user.get().canCommit()) {
    System.out.println("committing...");
  }
```

Adding authentication (4)

- However, it is unpleasant to use `user.get()` all the time
- Fourth step: introduce a proxy...
- ... and refactor RepositoryOps to use it

Adding authentication (4)

- However, it is unpleasant to use `user.get()` all the time
- Fourth step: introduce a proxy...
- ... and refactor RepositoryOps to use it

Use a proxy to access the thread local object

class UserProxy

```
private static ThreadLocal<User> user =
  new ThreadLocal<User>() {
    // ...
  };

public boolean canCommit() {
  return user.get().canCommit();
}

public boolean canUpdate() {
  return user.get().canUpdate();
}

public boolean canCheckOut() {
```

Refactoring RepositoryOps

class RepositoryOps

```
private UserProxy userProxy = new UserProxy();

public void checkOut() {
  if (userProxy.canCheckOut()) {
    System.out.println("checking out...");
  }
}

public void update() {
  if (userProxy.canUpdate()) {
    System.out.println("updating...");
  }
}
```

Summing up

So what have we learned about thread specific storage?

- similar to defining a field in each thread class, but actually you don’t have to define it
- single logical variable that has independent values in each separate thread
- in Java, all you have to use is the ThreadLocal class

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Participants

What about all the participants in the book?

- Thread Specific Object: User
- Key: threadLocalHashCode in ThreadLocal
- Key Factory: ThreadLocal.nextHashCode()
- Thread Specific Object Set: threadLocals in Thread
- Thread Specific Object Proxy: UserProxy
- Application Thread: MyThread

Issues with ThreadLocal

Why use ThreadLocal when...

- we can just add a field to our thread class
- we don't need to understand how ThreadLocal works
- we don't need to write proxies/more code
- we don't need to worry that the objects are garbage collected too late

Tom Hawtin

In general, if you have control of Thread construction, adding a field will be faster and possibly less buggy than using ThreadLocal.

Issues with ThreadLocal

Why use ThreadLocal when...

- we can just add a field to our thread class
- we don't need to understand how ThreadLocal works
- we don't need to write proxies/more code
- we don't need to worry that the objects are garbage collected too late

Tom Hawtin

In general, if you have control of Thread construction, adding a field will be faster and possibly less buggy than using ThreadLocal.

Still...

However:

- sometimes we can't add fields to our thread class
- using ThreadLocal makes it easier to associate a thread with its per-thread data
    - e.g. for passing per-thread context information

Still...

However:

- sometimes we can't add fields to our thread class
- using ThreadLocal makes it easier to associate a thread with its per-thread data
    - e.g. for passing per-thread context information

Nice uses

In *Exploiting ThreadLocal to enhance scalability*, Brian Goetz explains possible uses.

Per-thread Singleton:

- process-wide Singletons vs. thread-wide ones
- e.g. a JDBC Connection is not thread safe
- a Connection pool is the customary solution
- alternative: use ThreadLocal in the Singleton

Per-thread Singleton

class ConnectionDispenser

```
private ThreadLocal<Connection> conn =
  new ThreadLocal<Connection>() {
    @Override
    protected Connection initialValue() {
      return DriverManager.getConnection(<url>);
    }
  };

public static Connection getConnection() {
  return (Connection) conn.get();
}
```

Nice uses

Debugging multi-threaded applications:

- more generally, applications which collect per-thread info
- debugging a multi-threaded app is cumbersome
- instead of println's, use a per-thread logger
- use DebugLogger.put() during run
- retrieve saved info with DebugLogger.get() at the end

Per-thread Singleton

class DebugLogger

```
private ThreadLocal<List> list =
  new ThreadLocal<List>() {
    @Override
    public List initialValue() {
      return new ArrayList();
    }
  };

public static void put(String text) {
  list.get().add(text);
}

public String[] get() {
  return list.get().toArray(new String[0]);
}
```

Nice uses

Servlet-based applications:

- use ThreadLocal variables to store per-request info
- only use when unit of work is one request!
- otherwise, context will get mixed up

Nice uses

Based on another article by Brian Goetz, *Can ThreadLocal solve the double-checked locking problem?*

Remember the DCL (double-checked locking) problem?

class DoubleCheck

```
private static Resource resource = null;

public static Resource getResource() {
  if (resource == null) {
    synchronized {
      if (resource == null)
        resource = new Resource();
    }
  }
  return resource;
```

Double-Checked Locking

- idea is to replace the check that the resource is null
- instead, check if thread has executed the synchronized block
- use ThreadLocal to do this
- can you suggest how?

Nice uses

class ThreadLocalDCL

```
private static ThreadLocal<Boolean> initHolder
  = new ThreadLocal<Boolean>();
private static Resource resource = null;

public Resource getResource() {
  if (initHolder.get() == null) {
    synchronized {
      if (resource == null)
        resource = new Resource();
      initHolder.set(true);
    }
  }
  return resource;
}
```

ThreadLocal performance is an issue!

- DCL was designed to help with performance
- does ThreadLocal beat synchronized lazy initialization?
- in JDK 1.2 ThreadLocal uses synchronized WeakHashMap with Thread as key
- in JDK 1.3 Thread has threadLocals field, but Thread.currentThread() is bottleneck
- ThreadLocal and Thread.currentThread() rewritten in JDK 1.4 → finally faster

ThreadLocal performance is an issue!

- DCL was designed to help with performance
- does ThreadLocal beat synchronized lazy initialization?
- in JDK 1.2 ThreadLocal uses synchronized WeakHashMap with Thread as key
- in JDK 1.3 Thread has threadLocals field, but Thread.currentThread() is bottleneck
- ThreadLocal and Thread.currentThread() rewritten in JDK 1.4 → finally faster

ThreadLocal performance is an issue!

- DCL was designed to help with performance
- does ThreadLocal beat synchronized lazy initialization?
- in JDK 1.2 ThreadLocal uses synchronized WeakHashMap with Thread as key
- in JDK 1.3 Thread has threadLocals field, but Thread.currentThread() is bottleneck
- ThreadLocal and Thread.currentThread() rewritten in JDK 1.4 → finally faster

ThreadLocal performance is an issue!

- DCL was designed to help with performance
- does ThreadLocal beat synchronized lazy initialization?
- in JDK 1.2 ThreadLocal uses synchronized WeakHashMap with Thread as key
- in JDK 1.3 Thread has threadLocals field, but Thread.currentThread() is bottleneck
- ThreadLocal and Thread.currentThread() rewritten in JDK 1.4 → finally faster

Advantages

Thread specific storage is good when you want to:

- avoid synchronization → scalability
- make thread unsafe objects easily usable by threads
- add context to your threads but:
  - you can't/won't change your thread classes
  - you don't want to pass objects around (per-thread Singleton)
- document a variable as being thread safe

Liabilities

You should think twice before using thread specific storage when:

- your application will run in a web application server
- more generally, when using thread pools
- memory is more important than synch time

Questions

???

Question 1

Assume SUN wants to add syntax support for thread specific storage.

Proposal

```
threadlocal <type> <variable-name>
```

Explain how this would be handled “behind the scenes”. Discuss (some of) the following issues:

- what would such a declaration be internally translated to?
- what is the semantics of assigning to such a variable?
- what is the semantics of reading such a variable?
- garbage collection of such a variable?
- whatever else you think is relevant...

Question 2

If you don't like question 1, you can choose this one instead. You are not required (or expected) to answer both questions.

Identify and explain one or two significant differences between the pthread model (discussed in the book) and the ThreadLocal model (discussed during the lecture).

For each identified difference, discuss why one version or the other is preferable. Or if you think both have advantages/disadvantages, discuss those.