Laurence的技术博客

Spring基于ThreadLocal的“资源-事务”线程绑定设计的缘起

标签： spring hibernate session 数据库 sharding service

2012-07-25 14:17 23179人阅读 评论(12) 收藏 举报

题目起的有些拗口了，简单说，这篇文章想要解释Spring为什么会选择使用ThreadLocal将资源和事务绑定到线程上，这背后有着什么样的起因和设计动机，通过分析帮助大家更清晰地认识Spring的线程绑定机制。本文原文链接：

http://blog.csdn.net/bluishglc/article/details/7784502 转载请注明出处！

“原始”的数据访问写法

访问任何带有事务特性的资源系统，像数据库，都有着相同的特点：首先你需要获得一个访问资源的“管道”，对于数据库来说，这个所谓的“管道”是JDBC里的Connection,是Hibernate里的Session.然后你会通过“管道”下达一系列的读写指令，比如数据库的SQL，最后你会断开这个“管道”，释放对这个资源的连接。在Spring里，用访问资源的“管道”来指代资源，因此JDBC的Connection和Hibernate的Session都被称之为“资源”(Resource)(本文会交替使用这两种称呼)。另一方面，资源与事务又有着紧密的关系，事务的开启与提交都是在某个“Resource”上进行的。以Hibernate为例，一种“原始”的数据访问程序往往会写成这样：

```java
Session session = sessionFactory.openSession();//获取“资源”
Transaction tx = null;
try {
    tx = session.beginTransaction(); //开始事务
    ....
    DomainObject domainObject = session.load(...); //数据访问操作
    ....
```

```
    domainObject.processSomeBusinessLogic();//业务逻辑计算
    ....
    session.save(domainObject); //另一个数据访问操作
    ....
    session.save(anotherDomainObject); //又一个数据访问操作
    ....
    session.commit(); //提交事务
}
catch (RuntimeException e) {
    tx.rollback();
    throw e;
}
finally {
    session.close(); //释放资源
}
```

上述代码的思路很直白：首先获得数据库“资源”，然后在该资源上开始一个事务，经过一系列夹杂着业务计算和数据访问的操作之后，提交事务,释放资源。

分层带来的困扰

相信很多人一下就能看出上面代码的问题：业务逻辑与数据访问掺杂在了一起，犯了分层的“忌讳”。一个良好的分层系统往往是这样实现上述代码的：使用Service实现业务逻辑，使用DAO向Service提供数据访问支持。

某个Service的实现类：

```
[java]

public class MyServiceImpl implements MyService {

    public void processBusiness(){

        //在这里获得资源并开启事务么？NO！会引入数据访问的API，“污染"Service，破坏了分层!
        //Session session = sessionFactory.openSession();
        //session.beginTransaction();
        ....
        DomainObject domainObject = myDao.getDomainObject(...); //数据访问操作
        ....
        domainObject.processSomeBusinessLogic();//业务逻辑计算
        ....
        myDao.save(domainObject); //另一个数据访问操作
        ....
        myDao.save(anotherDomainObject); //又一个数据访问操作
        ....
    }
    ....
}
```

某个DAO的Hibernate实现类：

```java
public class MyDaoHibernateImpl implements MyDao {

    public void save(DomainObject domainObject){
        //在这里获得资源并开启事务么？NO！你怎么确定这个方法一定是一个独立的事务
        //而不会是某个事务的一部分呢？比如我们上面的Service。
        //Session session = sessionFactory.openSession();
        //session.beginTransaction();
        ....
        session.save(domainObject);
    }
    ....
}
```

矛盾的焦点

从“分层”的角度看，上述方案算是“完美”了，但却回避了一个现实的技术问题：如何安置“获取资源”(也就是session)和“开启事务”的代码呢？像代码中注释的那样，好像放在哪里都有问题，看上去像是一个“不可调和”的矛盾。如果要解决这个“不可调和”的矛盾，在技术上需要解决两个难题：

1. 如何“透明”地进行事务定界(Transaction Demarcation)?
2. 如何构建一个“上下文”，在事务开始与事务提交时，以及在事务过程中所有数据访问方法都能“隐式”地得到“同一个资源”(数据库连接/Hibernate Session)。所谓“隐式”是指不能把同一个资源实例用参数的方式传给数据访问方法，否则必然会出现数据访问层的上层代码受到数据访问专有API污染的问题(即破获了分层)，而使用全局变量显然是不行的，因为全局变量是唯一的，没有哪个应用能容忍只使用一个数据库连接，对于一个用户请求一个线程的多线程Web应用环境更是如此。

Spring的解决之道

Spring使用基于AOP的声明式事务定界解决了第一个问题，而使用基于ThreadLocal的资源与事务线程绑定成功地解决了第二个问题。(关于spring的具体实现，可以参考我的另一篇文章：Spring源码解析(一) Spring事务控制之Hibernate ，第一个问题所涉及源码主要是：org.springframework.aop.framework.JdkDynamicAopProxy 和 org.springframework.transaction.interceptor.TransactionInterceptor

第二个问题所涉及源码主要是：

org.springframework.transaction.support.AbstractPlatformTransactionManager 和

org.springframework.transaction.support.TransactionSynchronizationManager)

本文我们重点关注Spring是如何解决第二个问题的，对于这个问题有两点需要特别地解释：

1. “上下文”：Spring使用的是“线程上下文”，也就是TreadLocal,原因非常简单，做为一种线程作用域变量，它能很好地被“隐式”获取，即在当前线程下可以直接得到该变量(避免了参数传递)，同时又不会像全局变量那样作用域过大且全局只有一个实例。实际上，从更大的背景上来看，大多数的spring应用为B/S架构的web应用，受servlet线程模型的影响，此类web应用都是一个用户请求到达开启一个新的线程进行处理，在此背景下，spring这种以线程作为上下文绑定资源和事务的处理方式无疑是非常合适的。
2. “资源与事务的生命周期”：如果只从“线程绑定”的字面上理解，很容易让人误解为绑定到线程上的资源和事务的生命周期与线程是等长的，这是错误的。实际上，资源和事务的生命周期与线程生命周期没有必然联系，只是当资源和事务存在时，它们会以TreadLocal的形式绑定到线程上而已。而资源的生命周期与事务的生命周期才是等长的，我们把资源-事务这种生命周期关系称为：Connection-Per-Transaction 或是 Session-Per-Transaction。

Hibernate自己动手丰衣足食

作为一小段插曲，我们聊聊Hibernate。大概是为满足对Session-Per-Transaction的普遍需求，Hibernate也实现了自己的Session-Per-Transaction模型，就是大家所熟知的SessionFactory.getCurrentSession(),该方法返回绑定在当前线程上session实例，若当前线程没有session实例，创建一个新的实例以ThreadLocal的形式绑定到当前线程上，同时，该方法生成的session其实是一个session代理，这个代理会对内部的实际session附加如下动作：

1. 对session的数据操作方法进行拦截，确认在执行操作前已经调用过begainTranscation()开启了一个事务，否则会抛出异常。这一点确保了对session的使用必须总是从创建一个事务开始的。
2. 当事务在commit或rollback后session会自动关闭。这一点确保了事务提交后session也将不可用。

正是这两点确保了Session与Transaction保持了一致的生命周期。

一切是这样进行的

结合上述场景和Spring的解决方案，一个使用了Spring声明性事务，实现了良好分层的程序，它的资源和事务在Spring的控制下是这样工作的：

1. 若当前线程执行到了一个需要进行事务控制的方法(如某个service的方法)，通过AOP拦截，spring会在方法执行前申请一个数据库连接或者一个hibernate session.
2. 成功获得资源后，开启一个事务。
3. 将资源也就是数据库连接或是hibernate session的实例存放于当前线程的ThreadLocal里(也就是进行所谓的线程绑定)
4. 在方法执行过程中，任何需要获得数据库连接或是hibernate session进行数据访问的地方都会从当前线程的ThreadLocal里取出同一个数据库连接或是hibernate session的实例进行操作(这个动作由Spring提供的各种Template类实现)。
5. 方法执行结束，同样通过AOP拦截，spring取出绑定到当前线程上的事务(对于hibernate来说就是取出绑定在当前线程上一个SessionHolder实例，它保存着当前的session与transaction实例)，执行提交。
6. 事务提交之后，释放资源，清空当前线程上绑定的所有对象！
7. 如果该线程之后有新的事务发起，一切会重新开始，Spring会使用新的数据库连接或是hibernate session实例，开始新的事务，两个事务之间没有任何关系。

一个小小的总结

1. Connection-Per-Transaction/Session-Per-Transaction几乎总是你需要的。
2. 在分层架构中，有些变量或对象确实需要跨越分层工作（比如本文示例中的Connection/Session/Transaction),你可能需一种“上下文”(或者说是一种跨层的作用域)来存放这种变量或是对象，从而避免以“参数”的形式在层间传递它，线程局部变量ThreadLocal可能正是你需要的.

近期其他博文：

Spring源码解析(一) Spring事务控制之Hibernate

数据库分库分表(sharding)系列(三) 关于使用框架还是自主开发以及sharding实现层面的考量

数据库分库分表(sharding)系列(二) 全局主键生成策略

数据库分库分表(sharding)系列(一) 拆分实施策略和示例演示

- 上一篇 Spring源码解析(一) Spring事务控制之Hibernate
- 下一篇 数据库分库分表(sharding)系列(四) 多数据源的事务处理

顶 28　踩 1