彻底理解ThreadLocal - 相见欢

知其然

synchronized这类线程同步的机制可以解决多线程并发问题，在这种解决方案下，多个线程访问到的，都是同一份变量的内容。为了防止在多线程访问的过程中，可能会出现的并发错误。不得不对多个线程的访问进行同步，这样也就意味着，多个线程必须先后对变量的值进行访问或者修改，这是一种以延长访问时间来换取线程安全性的策略。

而ThreadLocal类为每一个线程都维护了自己独有的变量拷贝。每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了，那就没有任何必要对这些线程进行同步，它们也能最大限度的由CPU调度，并发执行。并且由于每个线程在访问该变量时，读取和修改的，都是自己独有的那一份变量拷贝，变量被彻底封闭在每个访问的线程中，并发错误出现的可能也完全消除了。对比前一种方案，这是一种以空间来换取线程安全性的策略。

来看一个运用ThreadLocal来实现数据库连接Connection对象线程隔离的例子。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {

        private static ThreadLocal<Connection> connectionHolder = new Threa
                @Override
                protected Connection initialValue() {
                        Connection conn = null;
                        try {
                                conn = DriverManager.getConnection(
                                                "jdbc:mysql://localhost:330
                                                "password");
                        } catch (SQLException e) {
                                e.printStackTrace();
                        }
                        return conn;
                }
        };
```

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

        public static void setConnection(Connection conn) {
                connectionHolder.set(conn);
        }
}
```

通过调用ConnectionManager.getConnection()方法，每个线程获取到的，都是和当前线程绑定的那个Connection对象，第一次获取时，是通过initialValue()方法的返回值来设置值的。通过ConnectionManager.setConnection(Connection conn)方法设置的Connection对象，也只会和当前线程绑定。这样就实现了Connection对象在多个线程中的完全隔离。在Spring容器中管理多线程环境下的Connection对象时，采用的思路和以上代码非常相似。

知其所以然

那么到底ThreadLocal类是如何实现这种“为每个线程提供不同的变量拷贝”的呢？先来看一下ThreadLocal的set()方法的源码是如何实现的：

```
/**
 * Sets the current thread's copy of this thread-local variable
 * to the specified value.  Most subclasses will have no need to
 * override this method, relying solely on the {@link #initialValue}
 * method to set the values of thread-locals.
 *
 * @param value the value to be stored in the current thread's copy of
 *        this thread-local.
 */
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

没有什么魔法，在这个方法内部我们看到，首先通过getMap(Thread t)方法获取一个和当前线程相关的ThreadLocalMap，然后将变量的值设置到这个

ThreadLocalMap对象中，当然如果获取到的ThreadLocalMap对象为空，就通过createMap方法创建。

线程隔离的秘密，就在于ThreadLocalMap这个类。ThreadLocalMap是ThreadLocal类的一个静态内部类，它实现了键值对的设置和获取（对比Map对象来理解），每个线程中都有一个独立的ThreadLocalMap副本，它所存储的值，只能被当前线程读取和修改。ThreadLocal类通过操作每一个线程特有的ThreadLocalMap副本，从而实现了变量访问在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。还有一点就是，ThreadLocalMap存储的键值对中的键是this对象指向的ThreadLocal对象，而值就是你所设置的对象了。

为了加深理解，我们接着看上面代码中出现的getMap和createMap方法的实现：

```
ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
```

```
void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }
```

代码已经说的非常直白，就是获取和设置Thread内的一个叫threadLocals的变量，而这个变量的类型就是ThreadLocalMap，这样进一步验证了上文中的观点：每个线程都有自己独立的ThreadLocalMap对象。打开java.lang.Thread类的源代码，我们能得到更直观的证明：

```
/* ThreadLocal values pertaining to this thread. This map is maintained
     * by the ThreadLocal class. */
    ThreadLocal.ThreadLocalMap threadLocals = null;
```

那么接下来再看一下ThreadLocal类中的get()方法，代码是这么说的：

```
/**
     * Returns the value in the current thread's copy of this
     * thread-local variable.  If the variable has no value for the
     * current thread, it is first initialized to the value returned
```

```
     * by an invocation of the {@link #initialValue} method.
     *
     * @return the current thread's value of this thread-local
     */
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null)
                return (T)e.value;
        }
        return setInitialValue();
    }

    /**
     * Variant of set() to establish initialValue. Used instead
     * of set() in case user has overridden the set() method.
     *
     * @return the initial value
     */
    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }
```

这两个方法的代码告诉我们，在获取和当前线程绑定的值时，ThreadLocalMap对象是以this指向的ThreadLocal对象为键进行查找的，这当然和前面set()方法的代码是相呼应的。

进一步地，我们可以创建不同的ThreadLocal实例来实现多个变量在不同线程间的访问隔离，为什么可以这么做？因为不同的ThreadLocal对象作为不同键，当然也可以在线程的ThreadLocalMap对象中设置不同的值了。通过ThreadLocal对象，在多线程中共享一个值和多个值的区别，就像你在一个HashMap对象中存储一个键值对和多个键值对一样，仅此而已。

设置到这些线程中的隔离变量，会不会导致内存泄漏呢？ThreadLocalMap对象保存在Thread对象中，当某个线程终止后，存储在其中的线程隔离的变

量，也将作为Thread实例的垃圾被回收掉，所以完全不用担心内存泄漏的问题。在多个线程中隔离的变量，光荣的生，合理的死，真是圆满，不是么？

最后再提一句，ThreadLocal变量的这种隔离策略，也不是任何情况下都能使用的。如果多个线程并发访问的对象实例只允许，也只能创建那么一个，那就没有别的办法了，老老实实的使用同步机制来访问吧。