ThreadLocal

1 概念

- 从Java官方文档中的描述：ThreadLocal类是用来提供线程内部的局部变量。这种变量在多线程环境下访问（get和set方法）时，可以保证各个线程的变量相对独立于其他线程内的变量。ThreadLocal实例通常来说，都是private static类型的，用来关联线程和线程的上下文
- 作用：ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。
- 简单来说
    - 在多线程并发的场景下，我们可以通过ThreadLocal在同一个线程，不同组件中传递公共变量，因为每个线程的变量都是独立的，所以不会相互的影响。

2 常用方法

- 创建ThreadLocal对象：ThreadLocal()
- 设置当前线程绑定的局部变量：public void set(T value)
- 获取当前线程绑定的局部变量：public T get()
- 移除当前线程绑定的局部变量：public void remove()

3 简单使用

示例

实体类对象Bean

```
public class Bean1 {
    private String context;

    public String getContext() {
        return context;
    }

    public void setContext(String context) {
        this.context = context;
    }
}
```

测试

```
public static void main(String[] args) {
    Bean1 demo1 = new Bean1();
    for (int i = 0; i < 5; i++) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                demo1.setContext(Thread.currentThread().getName() + "的数据");
                System.out.println("-------------------------");
                System.out.println(Thread.currentThread().getName()
                                   + "------>" + demo1.getContext());
            }
        });
        thread.setName("线程" + i);
        thread.start();
    }
}
```

结果

```
-------------------------
-------------------------
-------------------------
线程0------>线程2的数据
-------------------------
线程2------>线程2的数据
线程1------>线程4的数据
线程4------>线程4的数据
-------------------------
线程3------>线程3的数据
```

原因：线程不隔离

解决

- 方式一：加个同步代码块

```
public static void main(String[] args) {
    Bean1 demo1 = new Bean1();
    for (int i = 0; i < 5; i++) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (Demo1.class){
                    demo1.setContext(Thread.currentThread().getName() + "的数据");
                    System.out.println("-------------------------");
                    System.out.println(Thread.currentThread().getName()
                                       + "------>" + demo1.getContext());
                }
            }
        });
        thread.setName("线程" + i);
        thread.start();
    }
}
```

结果

```
-------------------------
线程0------>线程0的数据
-------------------------
线程3------>线程3的数据
-------------------------
线程2------>线程2的数据
-------------------------
线程1------>线程1的数据
-------------------------
线程4------>线程4的数据
```

- 方式二：ThreadLocal

```
public class Bean2 {

    ThreadLocal<String> threadLocal = new ThreadLocal();

    private String context;

    public String getContext() {
        //return context;
        return threadLocal.get();
    }

    public void setContext(String context) {
//        this.context = context;
        threadLocal.set(context);
    }
}
```

结果

```
-------------------------
线程0------>线程0的数据
-------------------------
-------------------------
线程3------>线程3的数据
-------------------------
线程4------>线程4的数据
-------------------------
线程2------>线程2的数据
线程1------>线程1的数据
```

4 ThreadLocal 和synchronized关键字

- 虽然ThreadLocal模式和synchronized关键字都用于处理多线程并发访问变量的问题，不过两者之间处理的角度和思路不同

区别

synchronized

- 原理：同步机制采用“以时间换空间”的方式，只提供了一份变量，让不同的线程排队访问
- 重点：多个线程之间访问资源同步

ThreadLocal

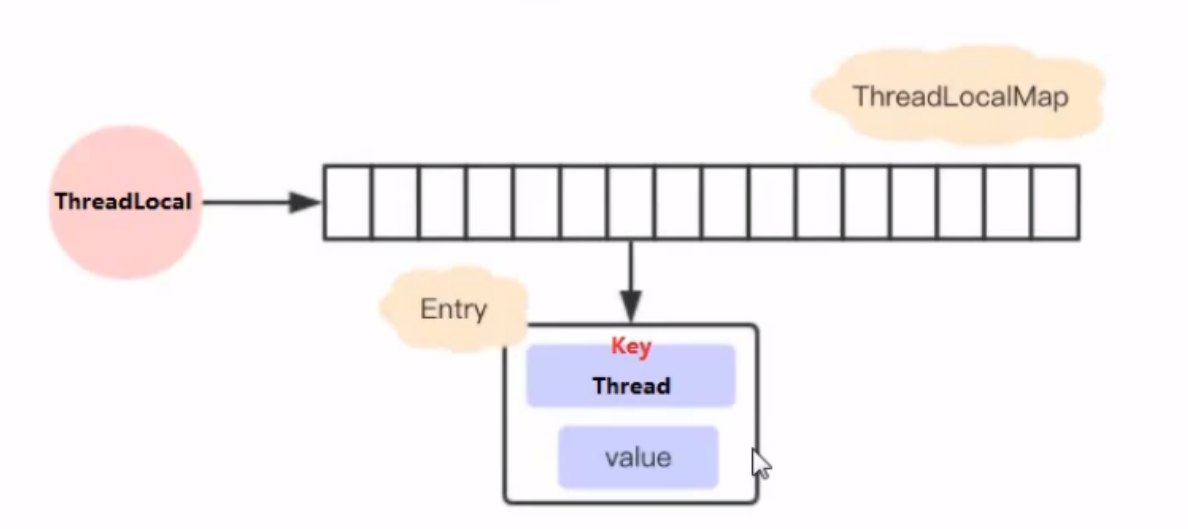
- 原理：采用"以空间换时间"的方式，为每一个线程都提供了一份变量的副本，从而实现同时访问而不会互相干扰
- 重点：多线程中让每个线程之间的数据相互隔离

总结：上一个实例中，synchronized和ThreadLocal都可以解决问题，但是ThreadLocal会更加的合适，可以使程序拥有更高的并发性

5 内部结构和底层实现原理

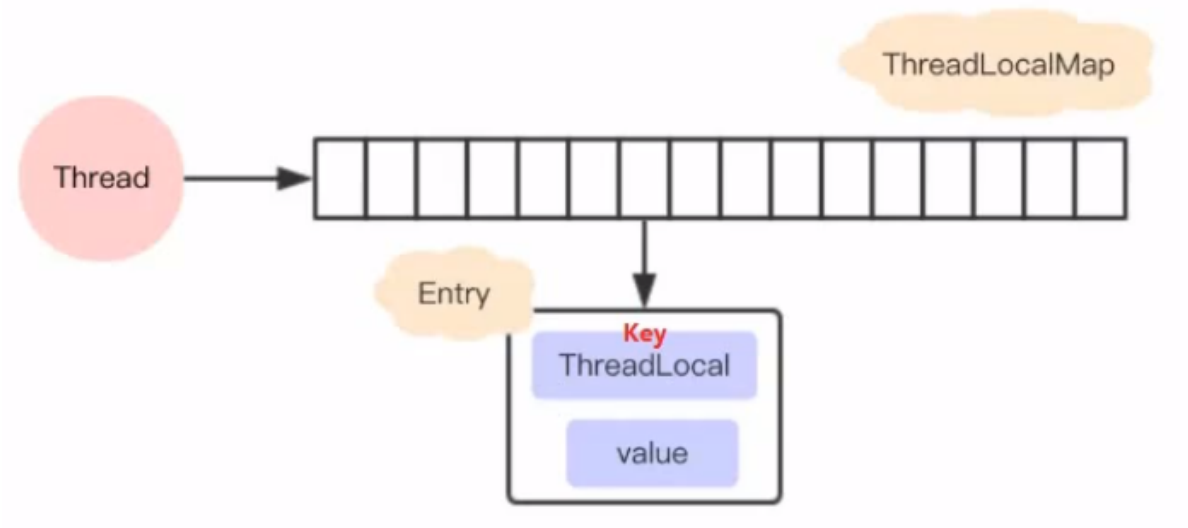
（1）内部结构

JDK早期ThreadLocal设计



- 每个ThreadLocal都会创建一个Map，然后线程作为Map的key，存储的局部变量作为Map的value。达到隔离效果

JDK8中ThreadLocal的设计



- 每个Thread维护一个ThreadLocalMap，key为ThreadLocal实例本身，value是要存储的局部变量
- 具体过程

- 每个Thread线程内部都有一个Map（ThreadLocalMap）
- key为ThreadLocal实例本身，value是要存储的局部变量副本
- Thread内部的Map由ThreadLocal维护，由ThreadLocal负责向map获取和设置线程的变量值
- 对于不同的线程，每次获取Value，别的线程也获取不到当前的线程的Value，所以就形成了副本的隔离

（2）底层实现原理

- set()

```
public void set(T value) {
    //获取当前线程
    Thread t = Thread.currentThread();
    //获取当前线程的ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //若不为空，将ThreadLocal和新的value放进去
        map.set(this, value);
    else
        //调用createMap()
        createMap(t, value);
}

void createMap(Thread t, T firstValue) {
    //创建新的ThreadLocalMap,将ThreadLocal和新的value放进去
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

- get()

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

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;
    }

    protected T initialValue() {
        return null;
    }

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

- remove()

```
    public void remove() {
        ThreadLocalMap m = getMap(Thread.currentThread());
        if (m != null)
            //调用map.remove()方法，以当前的ThreadLocal为key删除对应的实体entry
            m.remove(this);
    }
```

- ThreadLocalMap

之后再补。。。。

6 内存泄漏

（1）概念

- Memory overflow：内存溢出，没有足够的内存提供申请者使用
- Memory leak：内存泄漏，程序中已动态分配的堆内存由于某种原因，程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果，内存泄漏的堆积会导致内存溢出。

（2）强引用和弱引用

Java中的引用有4种类型：强、软、弱、虚。内存泄漏主要涉及强引用和弱引用

- 强引用：最常见的普通对象的引用，只要还有强引用指向一个对象，垃圾回收器就不会回收这种对象。

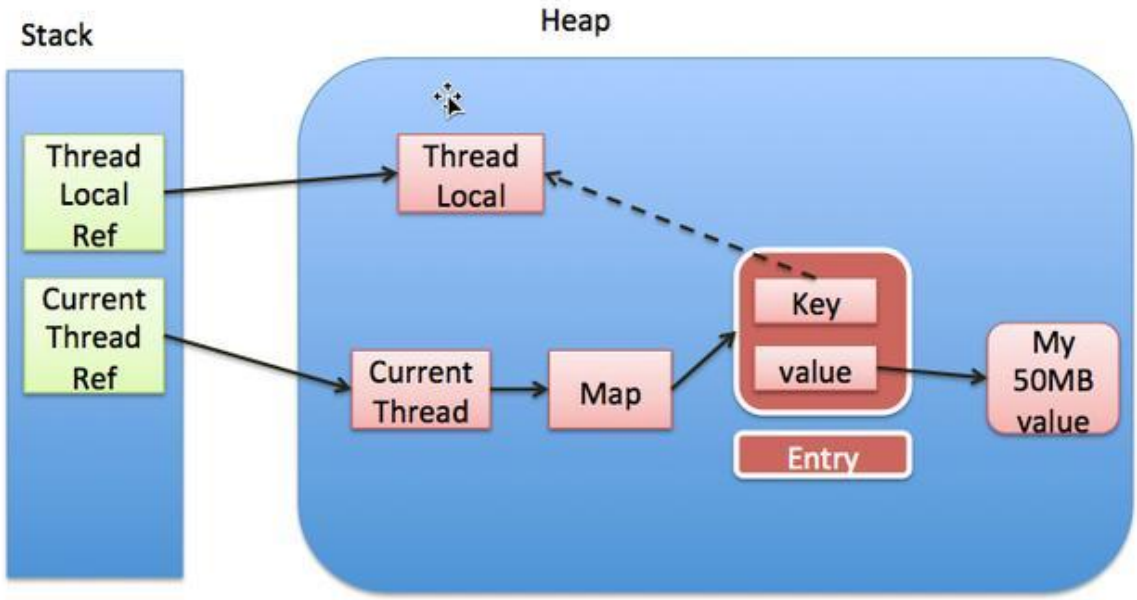
```
Object object =new Object();
String str ="hello";
```

- 弱引用：弱引用用来描述**非必需**对象,垃圾回收器一旦发现只具有弱引用的对象，不管当前内存空间是否足够，都会回收它的内存。

```
public class test {
    public static void main(String[] args) {
        WeakReference<People>reference=new WeakReference<People>(new
People("laowang",25));
        System.out.println(reference.get());
        System.gc();//通知GVM回收资源
        System.out.println(reference.get());
    }
}
class People{
    public String name;
    public int age;
    public People(String name,int age) {
        this.name=name;
        this.age=age;
    }
    @Override
    public String toString() {
        return "[name:"+name+",age:"+age+"]";
    }
}

输出结果：
    [name:laowang,age:25]
    null
```

（3）内存泄漏



- Thread中有一个map，就是ThreadLocalMap
- ThreadLocalMap的key是ThreadLocal，值是我们自己设定的
- 当ThreadLocal使用完之后，ThreadLocalRef和ThreadLocal之间的**强引用**就会断掉，ThreadLocal就会被回收
- ThreadLocal被回收后，map中的entry的key就为空，value还存在，这就是内存泄漏

出现内存泄漏的原因

- 没有手动删除这个 Entry
- CurrentThread 依然运行

相对第一种方式，第二种方式显然更不好控制，特别是使用线程池的时候，线程结束是不会销毁的.

那么为什么 key 要用弱引用呢

- 事实上，在 ThreadLocalMap 中的`set/getEntry`方法中，会对 key 为 null （也即是 ThreadLocal 为 null ）进行判断，如果为 null 的话，那么是会又如 value 置为 null 的.
- 这就意味着使用完 ThreadLocal , CurrentThread 依然运行的前提下．就算忘记调用 remove 方法，弱引用比强引用可以多一层保障：弱引用的 ThreadLocal 会被回收．对应value在下一次 ThreadLocalMap 调用`set/get/remove`中的任一方法的时候会被清除，从而避免内存泄漏.

7 其他使用场景

还可以使用在JDBC的事务当中。

- 我们都知道，事务具有原子性，是不可分割的
- 所以需要保证我们每一次对数据库进行操作，要保证都是同一个连接对象
- 再操作数据库前，关闭JDBC对数据库的自动提交，在操作结束之后，进行手动提交，若出现异常，则进行回滚

- 所以我们可以使用ThreadLocal对数据库的连接对象进行存储，key为当前线程，value为connection，通过threadlocal.get()方法，就可以保证在同一个线程中，得到的connection都是一致的
- 常规的解决方案：传参（将service层的connection传递到dao层）；加锁
    - 弊端：降低程序性能（加锁）；提高了代码的耦合度（传参）