ThreadLocal 的使用场景在 Java 的多线程编程中，为保证多个线程对共享变量的安全访问，通常会使用 synchronized 来保证同一时 刻只有一个线程对共享变量进行操作。 但在有些情况下，synchronized 不能保证多线程对共享变量的正确读写。例如类有一个类变量，该类变量会被多个类方法读写，当多线程操作该类的实 例对象时，如果线程对类变量有读取、写入操作就会发生类变量读写错误，即便是在类方法前加上 synchronized 也无效，因为同一个线程在两次调用方 法之间时锁是被释放的，这时其它线程可以访问对象的类方法，读取或修改类变量。 这种情况下可以将类变量放到 ThreadLocal 类型的对象中，使变量在每个线程中都有独立拷贝，不会出现一个线程读取变量时而被另一个线程修改的现 象。

下面举例说明：

```
public class QuerySvc {
 private String sql;

 private    static   ThreadLocal<string></string>  sqlHolder  =  new ThreadLocal<string></string>();

 public QuerySvc() {

 }

 public  void execute() {
  System.out.println("Thread " + Thread.currentThread().getId() +" Sql is " + sql);
  System.out.println("Thread  "  +  Thread.currentThread().getId()  +" Thread Local variable Sql is " + sqlHolder.get());
 }

 public  String getSql() {
```

```
  return sql;
 }

 public  void setSql(String sql) {
  this.sql = sql;
  sqlHolder.set(sql);

 }

}
```

为了说明多线程访问对于类变量和 ThreadLocal 变量的影响，QuerySvc 中分别设置了类变量 sql 和 ThreadLocal 变量，使用时先创建 QuerySvc 的一个实例对象，然后产生多个线程，分别设置不同的 sql 实例对象，然后再调用 execute 方法，读取 sql 的值，看是否是 set 方法中写入的值。 这种场景类似 web 应用中多个请求线程携带不同查询条件对一个 servlet 实例的访问，然后 servlet 调用业务对象，并传入不同查询条件，最后要保 证每个请求得到的结果是对应的查询条件的结果。

使用 QuerySvc 的工作线程如下：

```
public class Work extends Thread {

 private QuerySvc querySvc;
 private String sql;

 public Work(QuerySvc querySvc,String sql) {
  this.querySvc = querySvc;
  this.sql = sql;

 }


 public void run() {
  querySvc.setSql(sql);
  querySvc.execute();
 }
```

```
}
```

运 行线程代码如下：

```
QuerySvc qs = new QuerySvc();
   for (int k=0; k<10; k++){
    String sql = "Select * from table where id =" + k;
    new Work(qs,sql).start();
}
```

先创建一个 QuerySvc 实例对象，然后创建若干线程来调用 QuerySvc 的 set 和 execute 方法，每个 线程传入的 sql 都不一样，从运行结果可以看出 sql 变量中值不能保证在 execute 中值和 set 设置的值一样，在 web 应用中就表现为一个用户查询的结果不是自己的查询条件返回的结果，而是另一个用户查询条件的结果；而 ThreadLocal 中的值总是和 set 中设 置的值一样，这样通过使用 ThreadLocal 获得了线程安全性。

如果一个对象要被多个线程访问，而该对象存在类变量被不同类方法读写，为获得线程安全，可以用 ThreadLocal 来替代类变量。