日期转换问题

可变类如果不加线程安全保护，会有线程安全问题。

SimpleDateFormat 可变类会造成线程安全问题。

DateTimeFormatter 不会造成线程安全问题。

**package** com.concurrent.p8;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.text.ParseException;  
**import** java.text.SimpleDateFormat;  
**import** java.time.format.DateTimeFormatter;  
**import** java.time.temporal.TemporalAccessor;  
  
@Slf4j(topic = **"c.Test\_SimpleDateFormat"**)  
**public class** Test\_SimpleDateFormat {  
  
 */\*\*  
 \* SimpleDateFormat可变类会造成线程安全问题  
 \*/* @Test  
 **public void** test1() **throws** InterruptedException {  
 *//定义共享可变类对象* SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  
 *//10个线程访问共享变量* **for** (**int** i = 0; i < 10; i++) {  
 *//出异常 java.lang.NumberFormatException: multiple points* **new** Thread(() -> {  
 **try** {  
 ***log***.debug(sdf.parse(**"2021-09-20"**) + **""**);  
 } **catch** (ParseException e) {  
 e.printStackTrace();  
 }  
 }).start();  
 }  
 Thread.*sleep*(1000);  
 }  
  
 */\*\*  
 \* synchronized上锁，可以解决问题，但是效率低  
 \*/* @Test  
 **public void** test2() **throws** InterruptedException {  
 *//定义共享可变类对象* SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  
 *//10个线程访问共享变量* **for** (**int** i = 0; i < 10; i++) {  
 *//出异常* **new** Thread(() -> {  
 *//添加synchronized后可以正常运行* **synchronized** (sdf) {  
 **try** {  
 ***log***.debug(sdf.parse(**"2021-09-20"**) + **""**);  
 } **catch** (ParseException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
 }  
 Thread.*sleep*(1000);  
 }  
  
 */\*\*  
 \* 使用DateTimeFormatter不可变类  
 \*/* @Test  
 **public void** test3() **throws** InterruptedException {  
 DateTimeFormatter dtf = DateTimeFormatter.*ofPattern*(**"yyyy-MM-dd"**);  
 *//10个线程访问共享变量* **for** (**int** i = 0; i < 10; i++) {  
 **new** Thread(() -> {  
 TemporalAccessor parse = dtf.parse(**"2021-09-20"**);  
 ***log***.debug(parse + **""**);  
 }).start();  
 }  
 Thread.*sleep*(1000);  
 }  
  
}

不可变设计

String是不可变的，



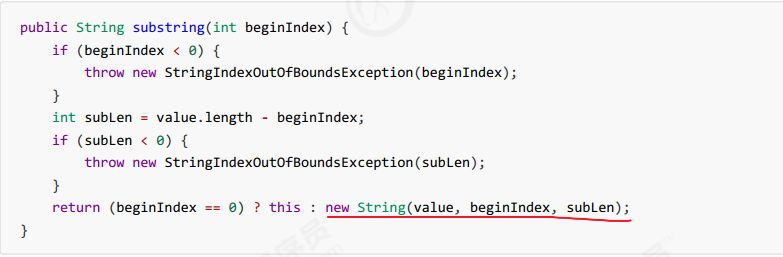
final的使用

发现该类、类中所有属性都是 final 的

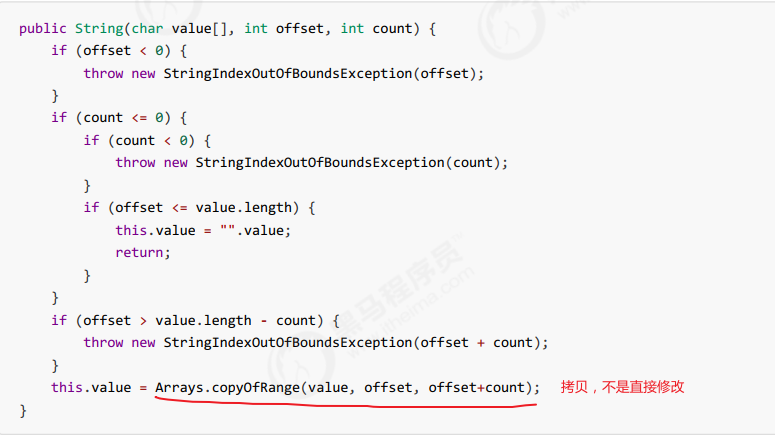
* 属性用 final 修饰保证了该属性是只读的，不能修改
* 类用 final 修饰保证了该类中的方法不能被覆盖，防止子类无意间破坏不可变性

保护性拷贝

substring方法使用了保护性拷贝



发现其内部是调用 String 的构造方法创建了一个新字符串，再进入这个构造看看，是否对 final char[] value 做出了修改，



结果发现也没有，构造新字符串对象时，会生成新的 char[] value，对内容进行复制 。这种通过创建副本对象来避免共享的手段称之为【保护性拷贝（defensive copy）】

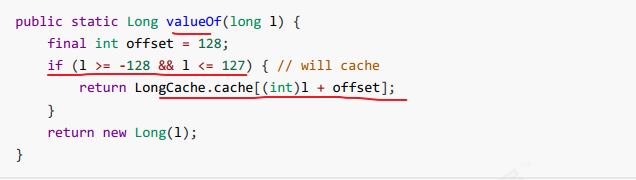
享元模式 Flyweight pattern

重用对象，减少对内存的使用。

1、体现

包装类：

在JDK中 Boolean，Byte，Short，Integer，Long，Character 等包装类提供了 valueOf 方法，例如 Long 的valueOf 会缓存 -128~127 之间的 Long 对象，在这个范围之间会重用对象，大于这个范围，才会新建 Long 对象。



注意：

Byte, Short, Long 缓存的范围都是 -128~127

Character 缓存的范围是 0~127

Integer的默认范围是 -128~127

最小值不能变

但最大值可以通过调整虚拟机参数 `

-Djava.lang.Integer.IntegerCache.high` 来改变

Boolean 缓存了 TRUE 和 FALSE

String串池：

BigDecimal BigInteger：

虽然本身是安全的，但是这些安全方法组合在一起就可能是不安全的。

2、享元模式应用-数据库连接池

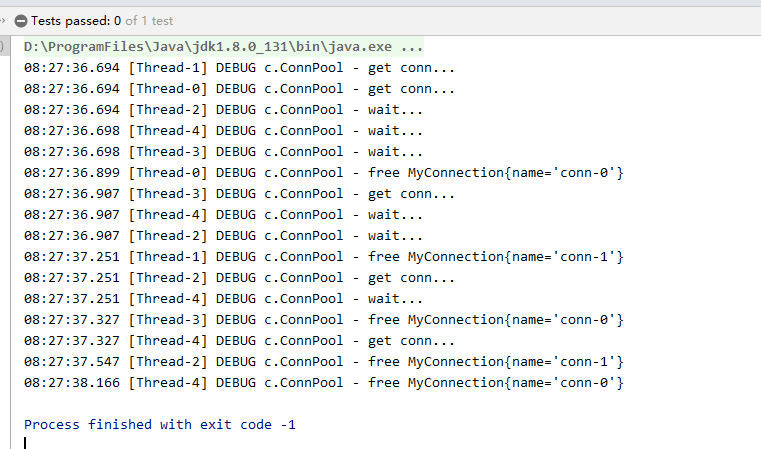
例如：一个线上商城应用，QPS 达到数千，如果每次都重新创建和关闭数据库连接，性能会受到极大影响。 这时预先创建好一批连接，放入连接池。一次请求到达后，从连接池获取连接，使用完毕后再还回连接池，这样既节约了连接的创建和关闭时间，也实现了连接的重用，不至于让庞大的连接数压垮数据库。

**package** com.concurrent.p8;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.sql.\*;  
**import** java.util.Map;  
**import** java.util.Properties;  
**import** java.util.Random;  
**import** java.util.concurrent.Executor;  
**import** java.util.concurrent.atomic.AtomicIntegerArray;  
  
@Slf4j(topic = **"c.Test\_DBConnPool"**)  
**public class** Test\_DBConnPool {  
  
 @Test  
 **public void** test\_ConnPool() {  
 *//创建连接池对象* ConnPool pool = **new** ConnPool(2);  
 **for** (**int** i = 0; i < 5; i++) {  
 **new** Thread(() -> {  
 Connection connection = pool.borrow();  
 *//模拟等待时间* **try** {  
 Thread.*sleep*(**new** Random().nextInt(1000));  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//归还连接* pool.free(connection);  
 }).start();  
 }  
 **while** (**true**) ;  
 }  
 */\*\*  
 \* 08:27:36.694 [Thread-1] DEBUG c.ConnPool - get conn...  
 \* 08:27:36.694 [Thread-0] DEBUG c.ConnPool - get conn...  
 \* 08:27:36.694 [Thread-2] DEBUG c.ConnPool - wait...  
 \* 08:27:36.698 [Thread-4] DEBUG c.ConnPool - wait...  
 \* 08:27:36.698 [Thread-3] DEBUG c.ConnPool - wait...  
 \* 08:27:36.899 [Thread-0] DEBUG c.ConnPool - free MyConnection{name='conn-0'}  
 \* 08:27:36.907 [Thread-3] DEBUG c.ConnPool - get conn...  
 \* 08:27:36.907 [Thread-4] DEBUG c.ConnPool - wait...  
 \* 08:27:36.907 [Thread-2] DEBUG c.ConnPool - wait...  
 \* 08:27:37.251 [Thread-1] DEBUG c.ConnPool - free MyConnection{name='conn-1'}  
 \* 08:27:37.251 [Thread-2] DEBUG c.ConnPool - get conn...  
 \* 08:27:37.251 [Thread-4] DEBUG c.ConnPool - wait...  
 \* 08:27:37.327 [Thread-3] DEBUG c.ConnPool - free MyConnection{name='conn-0'}  
 \* 08:27:37.327 [Thread-4] DEBUG c.ConnPool - get conn...  
 \* 08:27:37.547 [Thread-2] DEBUG c.ConnPool - free MyConnection{name='conn-1'}  
 \* 08:27:38.166 [Thread-4] DEBUG c.ConnPool - free MyConnection{name='conn-0'}  
 \*/*}  
  
*/\*\*  
 \* 自定义数据库连接池  
 \*/*@Slf4j(topic = **"c.ConnPool"**)  
**class** ConnPool {  
 *//1:数据库连接池大小* **private int poolSize**;  
 *//2:数据库连接池的状态* **private** AtomicIntegerArray **status**;  
 *//3:数据库连接池对象* **private** Connection[] **conn**;  
  
 *//4:构造方法* **public** ConnPool(**int** poolSize) {  
 **this**.**poolSize** = poolSize;  
 **this**.**status** = **new** AtomicIntegerArray(poolSize);  
 **this**.**conn** = **new** Connection[poolSize];  
 **for** (**int** i = 0; i < poolSize; i++) {  
 **this**.**conn**[i] = **new** MyConnection(**"conn-"** + i );  
  
 }  
 }  
  
 *//5:借连接* **public** Connection borrow() {  
 **while** (**true**) {  
 **for** (**int** i = 0; i < **poolSize**; i++) {  
 **if** (**status**.get(i) == 0) {  
 *//cas变更状态* **if** (**status**.compareAndSet(i, 0, 1)) {  
 ***log***.debug(**"get conn..."**);  
 **return conn**[i];  
 }  
 }  
 }  
 *//如果遍历1次后没有空闲的连接，则进入等待* **synchronized** (**this**) {  
 **try** {  
 ***log***.debug(**"wait..."**);  
 **this**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 *//6:还连接* **public void** free(Connection conn) {  
 *//归还连接时要检查该连接是否属于连接池* **for** (**int** i = 0; i < **poolSize**; i++) {  
 **if** (**this**.**conn**[i] == conn) {  
 **this**.**status**.set(i, 0);  
 *//唤醒等待线程* **synchronized** (**this**) {  
 ***log***.debug(**"free {}"**, conn);  
 **this**.notifyAll();  
 }  
 **break**;  
 }  
 }  
  
 }  
}

**class** MyConnection **implements** Connection {  
  
 **private** String **name**;  
  
 **public** MyConnection(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }

.......

运行结果：



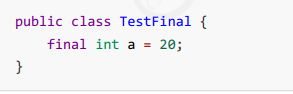
以上实现没有考虑：

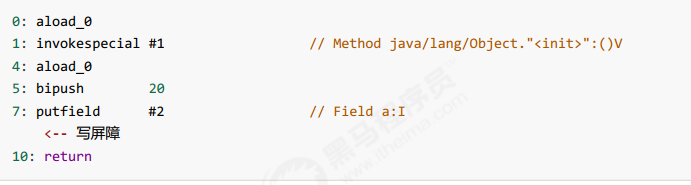
* 连接的动态增长与收缩
* 连接保活（可用性检测）
* 等待超时处理
* 分布式 hash

对于关系型数据库，有比较成熟的连接池实现，例如c3p0, druid等 对于更通用的对象池，可以考虑使用apache commons pool，例如redis连接池可以参考jedis中关于连接池的实现。

final原理

1、设置final变量的原理





发现 final 变量的赋值也会通过 putfield 指令来完成，同样在这条指令之后也会加入写屏障，保证在其它线程读到

它的值时不会出现为 0 的情况。

2、获取final变量的原理