共享模型之不可变

1. 不可变类的使用

1.引入

下面的代码在运行时，由于 SimpleDateFormat 不是线程安全的，会抛出异常

```
Slf4j
public class SimpleDateFormatTest {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd");

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                try {
                    log.debug("{}", sdf.parse("1951-04-21"));
                } catch (Exception e) {
                    log.error("{}", e);
                }
            }).start();
        }
    }
}
```

解决方法，在临界区sdf对象上加锁，但这样对性能会有一定的影响

```
@Slf4j
public class SimpleDateFormatTest {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd");

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                synchronized (sdf) {
                    try {
                        log.debug("{}", sdf.parse("1951-04-21"));
                    } catch (Exception e) {
                        log.error("{}", e);
                    }
                }
            }).start();
        }
    }
}
```

还可以使用JD8中的 不可变日期格式化类

如果一个对象在不能够修改其内部状态（属性），那么它就是线程安全的，因为不存在并发修改啊！这样的对象在Java 中有很多，例如在 Java 8 后，提供了一个新的日期格式化类：

```
@Slf4j
public class DateTimeFormatterTest {
    public static void main(String[] args) {
        DateTimeFormatter dtf =
DateTimeFormatter.ofPattern("yyyy-MM-dd");

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                try {
                    log.debug("{}", dtf.parse("1951-04-21"));
                } catch (Exception e) {
                    log.error("{}", e);
                }
            }).start();
        }
    }
```

```
}
```

2. 不可变类的设计

2.1 final的使用

另一个大家熟悉的String类也是不可变类

```
public final class String
    implements java.io.Serializable, Comparable<String>,
CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    .....
}
```

发现该类、类中所有属性都是 final 的，其中成员变量因为没有setter方法，故也是不可变的，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

2.2 保护性拷贝

使用字符串时，也有一些跟修改相关的方法啊，比如substring()、replace()等，那么下面就看一看这些方法是 如何实现的，就以substring 为例：

```
public String substring(int beginIndex, int endIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    if (endIndex > value.length) {
        throw new StringIndexOutOfBoundsException(endIndex);
    }
    int subLen = endIndex - beginIndex;
    if (subLen < 0) {
        throw new StringIndexOutOfBoundsException(subLen);
    }
    // 上面是一些校验，下面才是真正的创建新的String对象
```

```
    return ((beginIndex == 0) && (endIndex == value.length)) ?
this
            : new String(value, beginIndex, subLen);
}
```

发现其方法最后是调用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）

```
public String(char value[], int offset, int count) {
    if (offset < 0) {
        throw new StringIndexOutOfBoundsException(offset);
    }
    if (count <= 0) {
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        if (offset <= value.length) {
            this.value = "".value;
            return;
        }
    }
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset +
count);
    }
    // 上面是一些安全性的校验，下面是给String对象的value赋值，新创建了一个
数组来保存String对象的值
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

3. 无状态类的设计-享元模式

3.1 适用条件

当需要重用数量有限的同一类对象时

3.2 享元模式的体现

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

```
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

> 注意：
>
> - Byte, Short, Long 缓存的范围都是-128-127
> - Character 缓存的范围是 0-127
> - Boolean 缓存了 TRUE和 FALSE
> - 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

3.3 final原理

设置 final 变量的原理

理解了 volatile 原理 (读写屏障)，再对比 final 的实现就比较简单了

```
public class TestFinal {
 final int a = 20;
 }
```

字节码

```
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>":()V
4: aload_0
5: bipush 20
7: putfield #2 // Field a:I
 <-- 写屏障
10: retu
```

- 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。**