对象的组合

- 设计线程安全的类
  - 基本要素
    - 找出构成对象状态的所有变量
    - 找出约束状态变量的不变性条件
    - 建立对象状态的并发访问管理策略
  - 分析对象的状态
    - 从对象的域开始
    - 如果对象中所有的域都是基本类型的变量,那么这些域就是对象的全部状态
  - 收集同步需求
    - 如果不了解对象的不变性条件与后验条件,那么就不能确保线程安全性.要满足在状态变量的有效值或状态转换上的各种约束条件,就需要借助于原子性与封装性
  - 依赖状态的操作
    - 先验条件

      针对方法（method），它规定了在调用该方法之前必须为真的条件。
    - 后验条件

      针对方法，它规定了方法顺利执行完毕之后必须为真的条件。
    - 如果在某个操作中包含有基于状态的先验条件,那么这个操作就称为依赖状态的操作
    - 并发程序中要一直等先验条件为真,然后再执行操作
  - 状态的所有权
    - 对象封装它拥有的状态,对它封装的状态拥有所有权
    - 如果发布了某个可变对象的引用,那么就不再拥有独占的控制权,最多是“共享控制权”
    - 容器类通常表现出一种“所有权分离”的形式
      - 容器类拥有其自身的状态
      - 客户代码拥有容器中各个对象的状态
- 实例封闭
  - 将数据封装在对象内部,可以将数据的访问限制在对象的方法上,从而更容易确保线程在

访问数据时总能持有正确的锁

    - 封闭机制更容易构造线程安全的类,因为当封闭类的状态时,在分析类的线程安全性时就无须检查整个程序

    - Java监视器模式

        - 把所有的可变状态都封装起来

        - 由对象自己的内置锁来保护

- 线程安全性的委托

- 独立的状态变量

- 当委托失效时

    - 如果一个类是由多个独立且线程安全的状态变量组成,并且在所有的操作中都不包含无效状态转换,那么可以将线程安全性委托给底层的状态变量

- 发布底层的状态变量

    - 如果一个状态变量是线程安全的,并且没有任何不变性条件来约束它的值,在变量的操作上也不存在任何不允许的状态转换,那么就可以安全的发布这个变量

    - 私有构造函数捕获模式

```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) {
        this(a[0], a[1]);
    }

    public SafePoint(SafePoint p) {
        this(p.get());
    }

    public SafePoint(int x, int y) {
        this.set(x, y);
    }

    public synchronized int[] get() {
        return new int[]{x, y};
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

    - 不调用this(p.x,p.y)

- 在现有的线程安全类中添加功能

    - 客户端加锁机制

        - 对于使用某个对象X的客户端代码,使用X本身用于保护其状态的锁来保护这段客户

代码

- 要使用客户端加锁,必须知道对象X使用的是哪一个锁
- 比较脆弱,会破坏同步策略的封装性

- 组合
  - demo

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
   private final List<T> list;

   /**
    * PRE: list argument is thread-safe.
    */
   public ImprovedList(List<T> list) { this.list = list; }

   public synchronized boolean putIfAbsent(T x) {
      boolean contains = list.contains(x);
      if (contains)
         list.add(x);
      return !contains;
   }

   // Plain vanilla delegation for List methods.
   // Mutative methods must be synchronized to ensure atomicity of putIfAbsent.

   public int size() {
      return list.size();
   }

   ...

   public synchronized boolean remove(Object o) {
      return list.remove(o);
   }

}
```

  - 比客户端加锁有性能损失,但是更健壮

- 将同步策略文档化
  - 在文档中说明客户代码需要了解的线程安全性保证,以及代码维护人员需要了解的同步策略
  - 猜测线程安全性的方法:从实现者的角度去解释规范,而不是从使用者的角度去解释

New 2 New 3 New 4