多线程设计模式

十四、Single Thread Execution 设计模式

14.1、机场过安检

Single Thread Execution 模式是指在同一时刻只能有一个线程去访问共享资源，就像独木桥一样每次只允许一人通行，简单来说， Single Thread Execution 就是采用排他式的操作保证在同一时刻只能有一个线程访问共享资源。

相信大家都有乘坐飞机的经历，在进入登机口之前必须经过安全检查，安检口类似于独木桥，每次只能通过一个人，工作人员除了检查你的登机牌以外，还要联网检查身份证信息以及是否携带危险物品，如下图所示。

14.1.1、非线程安全

先模拟一个非线程安全的安检口类，旅客（线程）分别手持登机牌和身份证接受工作人员的检查，示例代码如下所示。

```
package com.bjsxt.chapter14;
```

```
public class FlightSecurity {

    private int count = 0;
    private String boardingPass = "null";// 登机牌
    private String idCard = "null";// 身份证

    public void pass(String boardingPass, String idCard) {
        this.boardingPass = boardingPass;
        this.idCard = idCard;
        this.count++;
        check();
    }

    private void check() {
        // 简单的业务，当登机牌和身份证首位不相同时则表示检查不通过
        if (boardingPass.charAt(0) != idCard.charAt(0)) {
            throw new RuntimeException("-----Exception-----" + toString());
        }
    }

    @Override
    public String toString() {
        return "FlightSecurity{" +
                "count=" + count +
                ", boardingPass='" + boardingPass + '\'' +
                ", idCard='" + idCard + '\'' +
                '}';
    }

}
```

FlightSecurity 比较简单，提供了一个 pass 方法，将旅客的登机牌和身份证传递给 pass 方法，在 pass 方法中调用 check 方法对旅客进行检查，检查的逻辑也足够的简单，只需要检测登机牌和身份证首位是否相等（当然这样在现实中非常不合理，但是为了使测试简单我们约定这么做），我们看以下代码所示的测试。

```
package com.bjsxt.chapter14;

public class FlightSecurityTest {

    static class Passengers extends Thread {
        // 机场安检类
        private final FlightSecurity flightSecurity;
        // 旅客身份证
        private final String idCard;
        // 旅客登机牌
        private final String boardingPass;

        public Passengers(FlightSecurity flightSecurity, String idCard, String
boardingPass) {
            this.flightSecurity = flightSecurity;
            this.idCard = idCard;
            this.boardingPass = boardingPass;
        }

        @Override
        public void run() {
            while (true) {
                // 旅客不断地过安检
                flightSecurity.pass(boardingPass, idCard);
            }
```

```
        }
    }

    public static void main(String[] args) {
        // 定义三个旅客，身份证和登机牌首位均相同
        final FlightSecurity flightsecurity = new FlightSecurity();
        new Passengers(flightsecurity, "A123456", "AF123456").start();
        new Passengers(flightsecurity, "B123456", "BF123456").start();
        new Passengers(flightsecurity, "C123456", "CF123456").start();
    }

}
```

看起来每一个客户都是合法的，因为每一个客户的身份证和登机牌首字母都一样，运行上面的程序却出现了错误，而且错误的情况还不太一样，运行多次，发现了两种类型的错误信息，程序输出如下：

```
java.lang.RuntimeException: -----Exception-----FlightSecurity{count=218,
boardingPass='AF123456', idCard='B123456'}
java.lang.RuntimeException: -----Exception-----FlightSecurity{count=676,
boardingPass='BF123456', idCard='B123456'}
```

首字母相同检查不能通过和首字母不相同检查不能通过，为什么会出现这样的情况呢？首字母相同却不能通过？更加奇怪的是传入的参数明明全都是首字母相同的，为什么会出现首字母不相同的错误呢。

14.1.2、问题分析

首字母相同却未通过检查

1)线程 A 调用 pass 方法，传人”A123456”“AF123456”并且对 idcard 赋值成功，由于 CPU 调度器时间片的轮转，CPU 的执行权归 B 线程所有。

2) 线程 B 调用 pass 方法，传入”B123456”“BF123456”并且对 idcard 赋值成功，覆盖 A 线程赋值的 idCard。

3)线程 A 重新获得 CPU 的执行权，将 boardingPass 赋于 AF123456,因此 check 无法通过。

4)在输出 toString 之前，B 线程成功将 boardingPass 覆盖为 BF123456。

为何出现首字母不相同的情况

1)线程 A 调用 pass 方法，传入”A123456”“AF123456”并且对 id Card 赋值成功，由于 CPU 调度器时间片的轮转，CPU 的执行权归 B 线程所有。

2)线程 B 调用 pass 方法，传入”B123456”“BF123456”并且对 id Card 赋值成功，覆盖 A 线程赋值的 idCard。

3)线程 A 重新获得 CPU 的执行权，将 boardingPass 赋于 AF123456,因此 check 无法通过。

4)线程 A 检查不通过，输出 idcard=”A123456”和 boardingPass=”BF123456”。

14.1.3、线程安全

上面出现的问题说到底就是数据同步的问题，虽然线程传递给 pass 方法的两个参数能够百分之百地保证首字母相同，可是在为 FlightSecurity 中的属性赋值的时候会出现多个线程交错的情况，结合我们之前所讲内容可知，需要对共享资源增加同步保护，改进代码如下。

```
public synchronized void pass(String boardingPass, String idCard) {
    this.boardingPass = boardingPass;
    this.idCard = idCard;
    this.count++;
    check();
}
```

修改后的 pass 方法，无论运行多久都不会再出现检查出错的情况了，为什么只在 pas 方法增加 synchronized 关键字， check 以及 toString 方法都有对共享资源的访问，难道它们不加同步就不会引起错误么？由于 check 方法是在 pass 方法中执行的，pass 方法加同步已经保证了 single thread execution，因此 check 方法不需要增加同步，toString 方法原因与此相同。

何时适合使用 single thread execution 模式呢？答案如下。

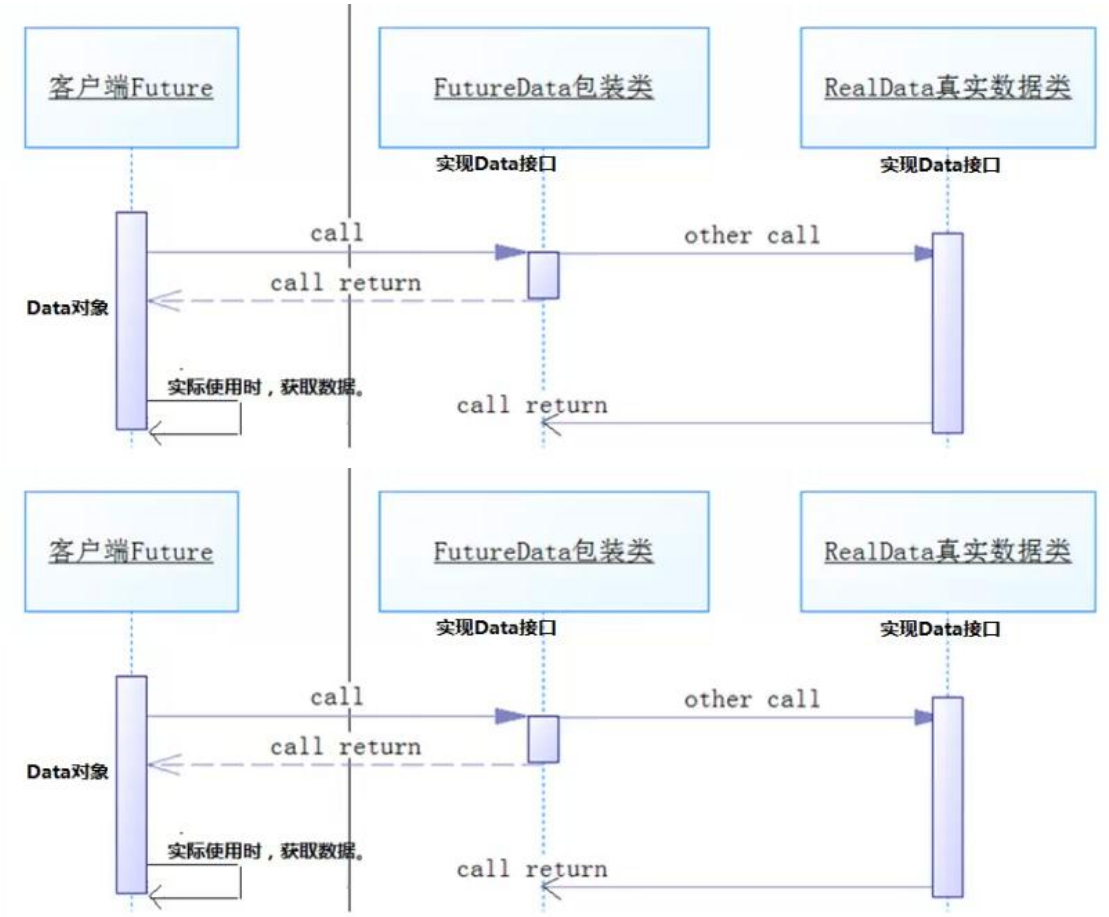
A．多线程访问资源的时候，被 synchronized 同步的方法总是排他性的。

B．多个线程对某个类的状态发生改变的时候 ,比如 Flightsecurity 的登机牌以及身份证。

在 Java 中经常会听到线程安全的类和线程非安全的类，所谓线程安全的类是指多个线程在对某个类的实例同时进行操作时，不会引起数据不一致的问题，反之则是线程非安全的类，在线程安全的类中经常会看到 synchronized 关键字的身影。

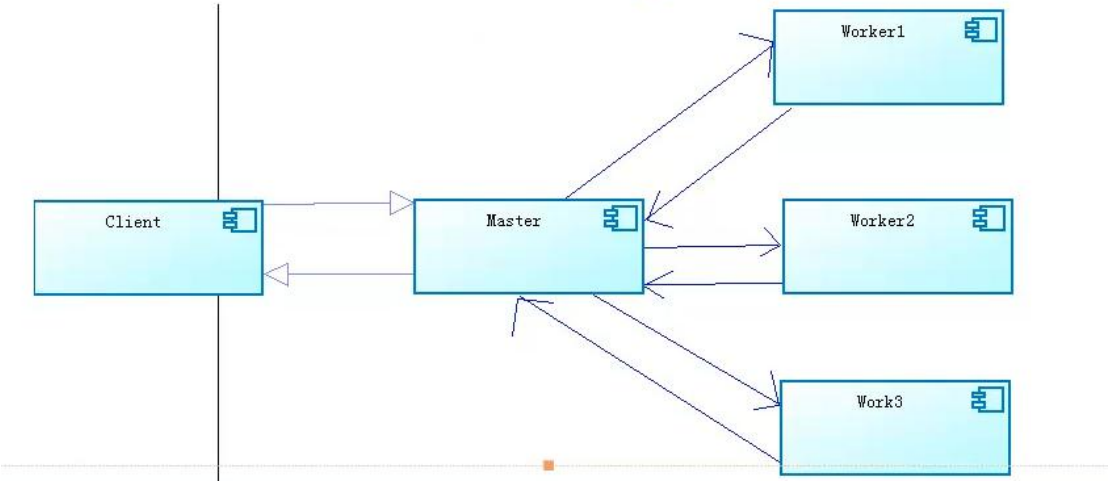
十五、Future 设计模式

Future 模式有点类似于商品订单。比如在网购时，当看重某一件商品事，就可以提交订单，当订单处理完成后，在家里等待商品送货上门即可。或者说更形象的我们发送 Ajax 请求的时候，页面是异步的进行后台处理，用户无须一直等待请求的结果，可以继续浏览或操作其他内容。

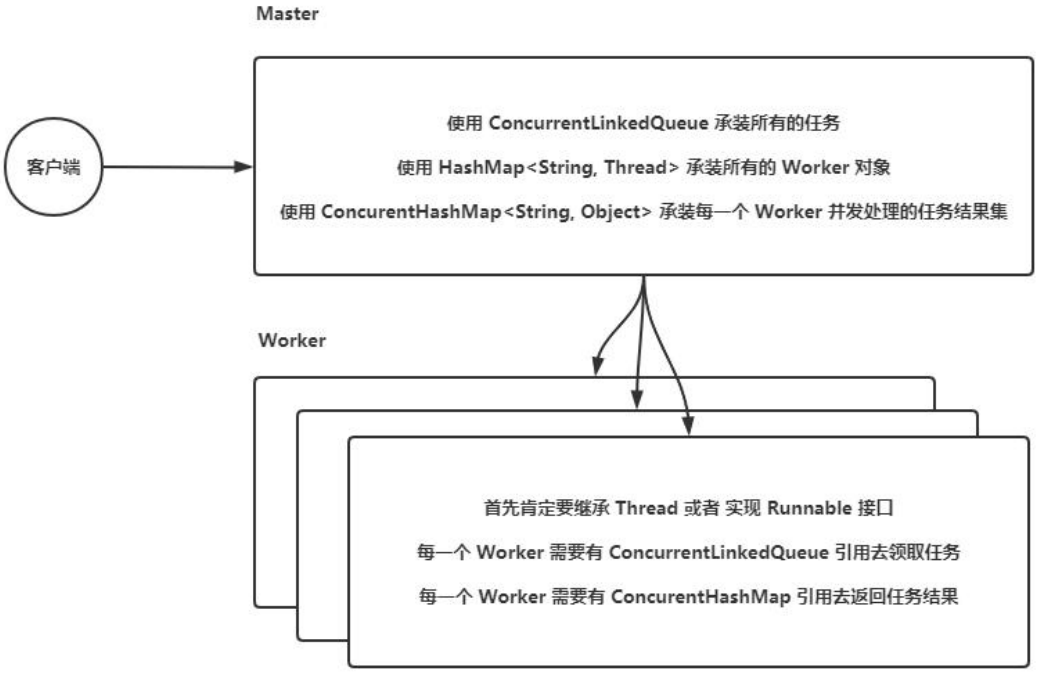


十六、Master-Worker 设计模式

Master- Worker 模式是常用的并行计算模式。它的核心思想是系统由两类进程协作工作： Master 进程和 Worker 进程。 Master 负责接收和分配任务，Worker 负责处理子任务。当各个 Worker-子进程处理完成后，会将结果返回给 Master，由 Master 做归纳和总结。其好处是能将一个大任务分解成若干个小任务，并行执行，从而提高系统的吞吐量。

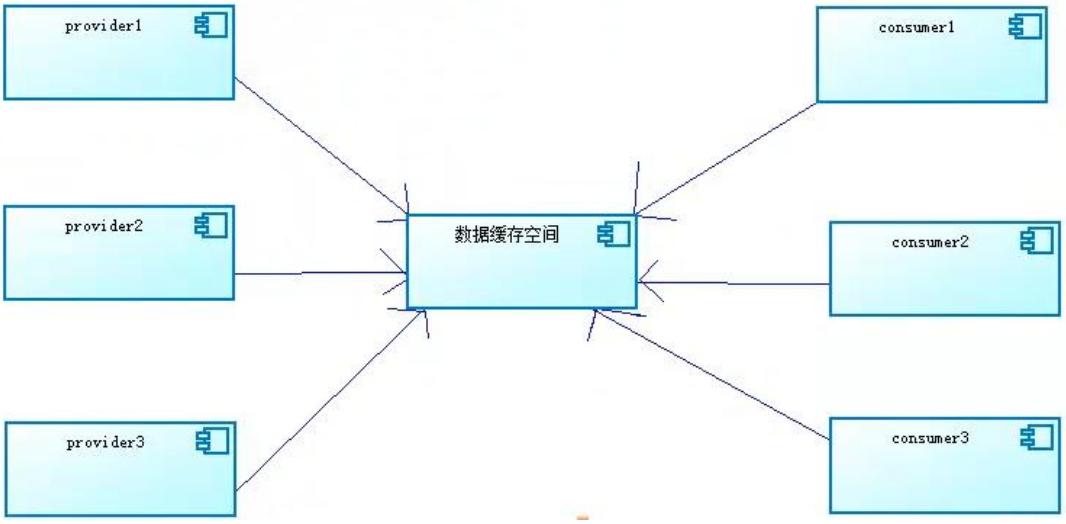


具体代码实现逻辑图如下：



十七、生产者消费者设计模式

生产者和消费者也是一个非常经典的多线程模式，我们在实际开发中应用非常广泛的思想理念。在生产消费模式中：通常由两类线程，即若干个生产者的线程和若干个消费者的线程。生产者线程负责提交用户请求，消费者线程则负责具体处理生产者提交的任务，在生产者和消费者之间通过共享内存缓存区进行通信。



具体代码逻辑实现思路：

十八、Immutable 不可变对象设计模式

不可变对象一定是线程安全的。

18.1、关于时间日期 API 线程不安全的问题

想必大家对 SimpleDateFormat 并不陌生。SimpleDateFormat 是 Java 中一个非常常用的类，该类用来对日期字符串进行解析和格式化输出，但如果使用不小心会导致非常微妙和难以调试的问题，因为 DateFormat 和 SimpleDateFormat 类不都是线程安全的，在多线程环境下调用 format() 和 parse() 方法应该使用同步代码来避免问题。关于时间日期 API 的线程不安全问题直到 JDK8 出现以后才得到解决。

关于线程不安全的代码示例如下：

```
package com.bjsxt.chapter18.demo01;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.*;

public class SimpleDateFormatThreadUnsafe {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 初始化时间日期API
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        // 创建任务线程，执行任务将字符串转成指定格式日期
        Callable<Date> task = () -> sdf.parse("20200808");
        // 创建线程池，数量为 10
        ExecutorService pool = Executors.newFixedThreadPool(10);
        // 构建结果集
        List<Future<Date>> results = new ArrayList<>();
        // 开始执行任务线程，将结果添加至结果集
        for (int i = 0; i < 10; i++) {
            results.add(pool.submit(task));
        }
        // 打印结果集中的内容
        // 在任务线程执行过程中并且访问结果集内容就会报错
        for (Future<Date> future : results) {
            System.out.println(future.get());
        }
        // 关闭线程池
        pool.shutdown();
    }
```

```
}
```

运行结果如下：

```
Tue Dec 04 00:00:00 CST 87
Wed Feb 04 00:00:00 CST 201
Sat Feb 04 00:00:00 CST 8
Sun Feb 04 00:00:00 CST 20
Sun Feb 04 00:00:00 CST 20
Sat Aug 08 00:00:00 CST 2020
Sat Feb 04 00:00:00 CST 8
Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.NumberFormatException: For input string: "E.2"
    at java.base/java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.base/java.util.concurrent.FutureTask.get(FutureTask.java:191)
    at com.bjsxt.chapter18.SimpleDateFormatDemo.main(SimpleDateFormatDemo.java:27)
```

我们先自己来解决一下这个问题，线程不安全，我给它放到 ThreadLocal 中是否可行呢？

```
package com.bjsxt.chapter18.demo01;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * 将每次需要格式转换的参数都放入 ThreadLocal 中进行
 */
public class DateFormatThreadLocal {

    private static final ThreadLocal<DateFormat> df =
            ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd"));

    public static Date convert(String source) throws ParseException {
        return df.get().parse(source);
    }

}
```

然后格式化日期代码如下：

```
package com.bjsxt.chapter18.demo01;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.*;

public class SimpleDateFormatThreadSafe {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // 初始化时间日期API
        SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
        // 创建任务线程，执行任务将字符串转成指定格式日期
        //Callable<Date> task = () -> sdf.parse("20191020");
        // 使用 ThreadLocal 处理非线程安全
        Callable<Date> task = () -> DateFormatThreadLocal.convert("20191020");
        // 创建线程池，数量为10
        ExecutorService pool = Executors.newFixedThreadPool(10);
        // 构建结果集
        List<Future<Date>> results = new ArrayList<>();
        // 开始执行任务线程，将结果添加至结果集
        for (int i = 0; i < 10; i++) {
            results.add(pool.submit(task));
        }
```

```
        // 打印结果集中的内容
        // 在任务线程执行过程中并且访问结果集内容就会报错
        for (Future<Date> future : results) {
            System.out.println(future.get());
        }
        // 关闭线程池
        pool.shutdown();
    }

}
```

上面的程序不管运行多少次都不会再出现线程不安全的问题。

18.2、定义不可变对象的策略

如何定义不可变对象呢？官方文档描述如下：

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields *final* and *private*.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as *final*. A more sophisticated approach is to make the constructor *private* and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    1. Don't provide methods that modify the mutable objects.
    2. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

参考官网文档后设计一个不可变对象，如下：

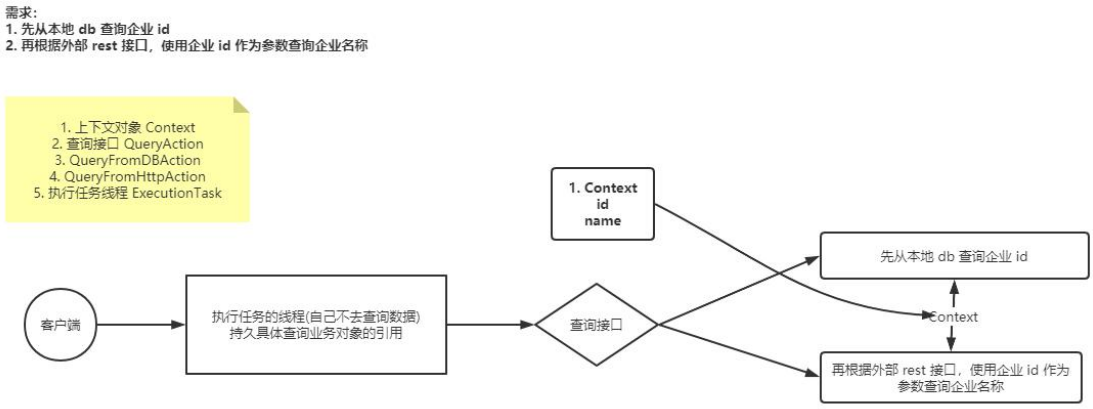
```
package com.bjsxt.chapter18.demo02;

public final class Person {

    private final String name;
    private final String address;

    public Person(final String name, final String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
```

```
        return address;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }

}
```

十九、多线程上下文设计模式

关于上下文（Context),我们在开发的过程中经常会遇到，比如开发 Struts2 的 ActionContext、 Spring 中的 ApplicationContext，上下文是贯穿整个系统或阶段生命周期的对象，其中包含了系统全局的一些信息，比如登录之后的用户信息、账号信息，以及在程序每一个阶段运行时的数据。

具体的代码业务逻辑图：



二十、Balking 设计模式

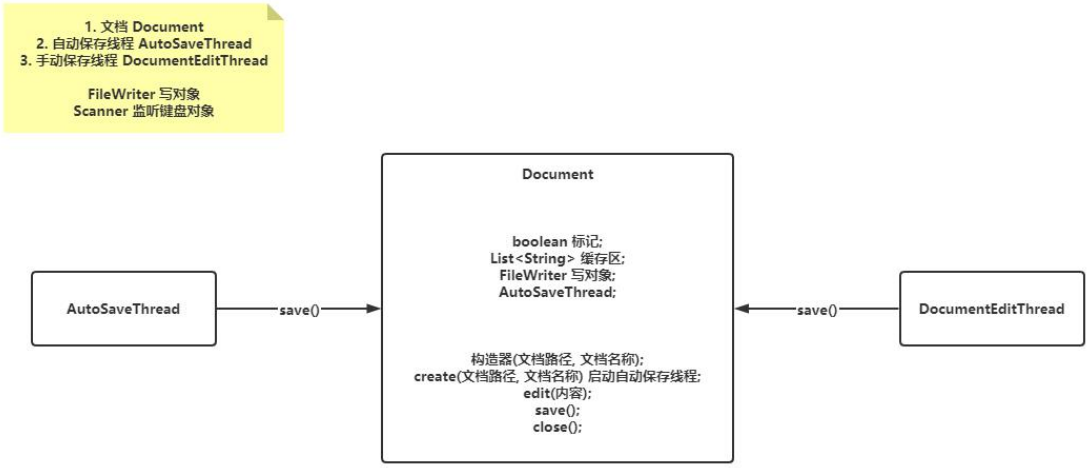
20.1、什么是 Balking 设计

多个线程监控某个共享变量，A 线程监控到共享变量发生变化后即将触发某个动作，但是此时发现有另外一个线程 B 已经针对该变量的变化开始了行动，因此 A 便放弃了准备开始的工作，我们把这样的线程间交互称为 Balking（犹豫）设计模式。其实这样的场景在生活中很常见，比如你去饭店吃饭，吃到途中想要再点一个小菜，于是你举起手示意服务员，其中一个服务员看到了你举手正准备走过来的时候，发现距离你比较近的服务员已经准备要受理你的请求于是中途放弃了。

再比如，我们在用 word 编写文档的时候，每次的文字编辑都代表着文档的状态发生了改变，除了我们可以使用 ctrl+s 快捷键手动保存以外，word 软件本身也会定期触发自动保存，如果 word 自动保存文档的线程在准备执行保存动作的时候，恰巧我们进行了主动保存，那么自动保存文档的线程将会放弃此次的保存动作。

看了以上两个例子的说明 ,想必大家已经清楚了 Balking 设计模式要解决的问题了吧，简短截说就是某个线程因为发现其他线程正在进行相同的工作而放弃即将开始的任务，在本章中，我们将通过模拟 word 文档自动保存与手动保存的功能讲解 Balking 模式的设计与应用。

20.2、Balking 模式之文档编辑



Document

在代码中，设计了 Document 类代表文档本身，在 Document 中有两个主要法 save 和 edit 分别用于保存文档和编辑文档。

```
package com.bjsxt.chapter20;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

// 代表正在编辑的文档
public class Document {

    // 如果文档发生改变，changed 会被设置为 true
    private boolean changed;
    // 一次需要保存的内容
    private List<String> content = new ArrayList<>();
    // 写对象
    private final FileWriter fileWriter;
```

```
    // 自动保存文档的对象
    private static AutoSaveThread autoSaveThread;

    // 构造函数需要传入文档保存的路径和文档名称
    public Document(String documentPath, String documentName) throws IOException {
        this.fileWriter = new FileWriter(new File(documentPath, documentName));
    }

    // 静态方法，主要用于创建文档，顺便启动自动保存文档的线程
    public static Document create(String documentPath, String documentName) throws
IOException {
        Document document = new Document(documentPath, documentName);
        autoSaveThread = new AutoSaveThread(document);
        autoSaveThread.start();
        return document;
    }

    // 文档的编辑，其实就是往 content 队列中提交字符串
    public void edit(String content) {
        synchronized (this) {
            this.content.add(content);
            // 文档改变 changed 会变为 true
            this.changed = true;
        }
    }

    // 文档关闭的时候首先中断自动保存线程，然后关闭 writer 释放资源
    public void close() throws IOException {
        autoSaveThread.interrupt();
        fileWriter.close();
    }

    // save 方法用于为外部显示进行文档保存
    public void save() throws IOException {
        synchronized (this) {
            // 如果文档已经保存了，则直接返回
            if (!changed)
                return;
            for (String cache : content) {
                this.fileWriter.write(cache);
                this.fileWriter.write("\r\n");
            }
            this.fileWriter.flush();
            System.out.println(Thread.currentThread().getName() + " 保存成功，保存内容
为：" + this.content);
            // 将 changed 修改为 false，表明此刻再没有新的内容编辑
            this.changed = false;
            // 清空缓存数据
            this.content.clear();
        }
    }

}
```

在上述代码中：

edit 方法和 save 方法进行方法同步，其目的在于防止当文档在保存的过程中如果遇到新的内容被编辑时引起的共享资源冲突问题。

changed 在默认情况下为 false，当有新的内容被编辑的时候将被修改为 true。

在进行文档保存的时候，首先查看 changed 是否为 true，如果文档发生过编辑则在文

档中保存新的内容，否则就会放弃此次保存动作，changed 是 balking pattern 关注的状态，当 changed 为 tue 的时候就像远处的服务员看到客户的请求被另外一个服务员接管了一样，于是放弃了任务的执行。

在创建 Document 的时候，顺便还会启动自动保存文档的线程，该线程的主要目的在于在固定时间里执行一次文档保存动作。

AutoSaveThread

与平日里编写 word 文档一样，word 会定期自动保存我们编辑的文档，如果在电脑出现故障重启之时，没有来得及对文档保存，也不至于损失太多劳动成果，它甚至能够百分之百的恢复， AutoSaveThread 类扮演的角色便在于此。

```
package com.bjsxt.chapter20;

import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class AutoSaveThread extends Thread {

    private final Document document;

    public AutoSaveThread(Document document) {
        super("AutoSaveThread");
        this.document = document;
    }

    @Override
    public void run() {
        while (true) {
            try {
                // 每隔 1 秒自动保存一次文档
                document.save();
                TimeUnit.SECONDS.sleep(1);
            } catch (IOException | InterruptedException ioException) {
                ioException.printStackTrace();
                break;
            }
        }
    }
}
```

AutoSaveThread 比较简单，其主要的工作就是每隔一秒的时间调用一次 Document 的 save 方法。

DocumentEditThread

AutoSaveThread 线程用于文档自动保存，那么 DocumentEditThread 线程则类似于主动编辑文档的作者，在 DocumentEditThread 中除了对文档进行修改编辑之外，还会同时按下 Ctrl+S 组合键（调用 save 方法）主动保存。

```
package com.bjsxt.chapter20;

import java.io.IOException;
```

```
import java.util.Scanner;

// 该线程代表的是主动进行文档编辑的线程，为了增加交互性，我们使用 Scanner
public class DocumentEditThread extends Thread {

    private final String documentPath;
    private final String documentName;
    private final static Scanner SCANNER = new Scanner(System.in);

    public DocumentEditThread(String documentPath, String documentName) {
        super("DocumentEditThread");
        this.documentPath = documentPath;
        this.documentName = documentName;
    }

    @Override
    public void run() {
        int times = 0;
        try {
            Document document = Document.create(documentPath, documentName);
            while (true) {
                // 获取用户的键盘输入
                String text = SCANNER.next();
                if ("exit".equals(text)) {
                    document.close();
                    break;
                }
                // 将内容编辑到 document 中
                document.edit(text);
                if (times == 5) {
                    // 用户再输入了 5 次以后进行文档保存
                    document.save();
                    times = 0;
                }
                times++;
            }
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }
    }

}
```

DocumentEditThread 类代表了主动编辑文档的线程，在该线程中，我们使用 Scanner 交互的方式，每一次对文档的修改都不可能直接保存（Ctl+S），因此在程序中约定了五次以后主动执行保存动作，当输入 exit 时，表示要退出此次文档编辑，测试代码如下：

```
package com.bjsxt.chapter20;

public class BalkingTest {

    public static void main(String[] args) {
        new DocumentEditThread("D:\\", "balking.txt").start();
    }

}
```

二十一、Guarded Suspension 设计模式

21.1、什么是 Guarded Suspension 设计模式

Suspension 是“挂起”、“暂停”的意思，而 Guarded 则是“担保”的意思，连在一起就是确保挂起。当线程在访问某个对象时，发现条件不满足，就暂时挂起等待条件满足时再次访问。

Guarded Suspension 设计模式是很多设计模式的基础，比如生产者消费者模式，同样在 Java 并发包中的 BlockingQueue 中也大量使用到了 Guarded Suspension 设计模式。

21.2、Guarded Suspension 的示例

```
package com.bjsxt.chapter21;

import java.util.LinkedList;

public class GuardedSuspensionQueue {

    // 定义存放 Integer 类型的 queue
    private final LinkedList<Integer> QUEUE = new LinkedList<>();

    // 定义 queue 的最大容量
    private final int LIMIT;

    public GuardedSuspensionQueue(int limit) {
        LIMIT = limit;
    }

    // 往队列中插入数据，如果 queue 中的元素超过了最大容量，则会陷入阻塞
    public void offer(Integer data) throws InterruptedException {
        synchronized (this) {
            // 判断 queue 的当前元素是否超过了 LIMIT
            while (QUEUE.size() >= LIMIT) {
                System.out.println("----------队列已满----------");
                this.wait();// 挂起当前线程
            }
            // 插入元素并且唤醒 take 线程
            QUEUE.addLast(data);
            System.out.println("插入成功，插入的元素为：" + data);
            this.notifyAll();
        }
    }

    // 从队列中获取元素，如果队列此时为空，则会使当前线程阻塞
    public Integer take() throws InterruptedException {
        synchronized (this) {
            // 如果队列为空
```

```
            while (QUEUE.isEmpty()) {
                System.out.println("----------队列已空----------");
                this.wait();// 挂起当前线程
            }
            // 通过 offer 线程可以继续插入数据
            this.notifyAll();
            return QUEUE.removeFirst();
        }
    }

}
```

在 GuardedSuspensionQueue 中，我们需要保证线程安全的是 queue，分别在 take 和 offer 方法中对应的临界值是 queue 为空和 queue 的数量>=100,当 queue 中的数据已经满时，如果有线程调用 offer 方法则会被挂起(Suspension),同样，当 queue 没有数据的时候，调用 take 方法也会被挂起。

Guarded Suspension 模式是一个非常基础的设计模式，它主要关注的是当某个条件（临界值）不满足时将操作的线程正确地挂起，以防止出现数据不一致或者操作超过临界值的控制范围。

二十二、Latch 设计模式

Latch(阀门)设计模式也叫做 Count Down 设计模式。当若干个线程并发执行完某个特定的任务，然后等到所有的子任务都执行结束之后再统一汇总。

CountDownLatch 能够使一个线程在等待另外一些线程完成各自工作之后，再继续执行。使用一个计数器进行实现。计数器初始值为线程的数量。当每一个线程完成自己任务后，计数器的值就会减一。当计数器的值为 0 时，表示所有的线程都已经完成了任务，然后在 CountDownLatch 上等待的线程就可以恢复执行任务。示例代码如下：

```
package com.bjsxt.chapter22;

/**
 * 自定义 CountDownLatch
 */
public class CustomCountDownLatch {

    // 阀门值
    private final int latchNum;

    // 计数器
    private int count;

    public CustomCountDownLatch(int latch_num) {
        this.latchNum = latch_num;
        this.count = latch_num;
    }

    public void countDown() {
        synchronized (this) {
            this.count--;
            this.notifyAll();
        }
    }
```

```
    public void await() {
        synchronized (this) {
            if (count != 0) {
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

}
```

其实 JDK 的 JUC 包下已经有对应的类存在了，叫做 CountDownLatch，具体使用如下：

```
package com.bjsxt.chapter22;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

public class Main {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("第一阶段开始工作。");

        int count = 5;

        // 自定义 CountDownLatch
        // CustomCountDownLatch latch = new CustomCountDownLatch(count);

        // JDK CountDownLatch
        CountDownLatch latch = new CountDownLatch(count);

        for (int i = 0; i < count; i++) {
            new Thread(() -> {
                try {
                    System.out.println(Thread.currentThread().getName());
                    TimeUnit.SECONDS.sleep(2);
                    // 每一个线程完成自己任务后，计数器减一
                    latch.countDown();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        latch.await();

        System.out.println("阶段一全部完成，第二阶段开始工作。");
    }

}
```

二十三、Two Phase Termination 设计模式

23.1、什么是 Two Phase Termination 模式

当一个线程正常结束，或者因被打断而结束，或者因出现异常而结束时，我们需要考虑如何同时释放线程中资源，比如文件句柄、 Socket 套接字句柄、数据库连接等比较稀缺的资源。 Two Phase Termination 设计模式可以帮助我们实现，如图所示。

如图所示，我们使用“作业中”表示线程的执行状态，当希望结束这个线程时，发出线程结束请求，接下来线程不会立即结束，而是会执行相应的资源释放动作直到真正的结束，在终止处理状态时，线程虽然还在运行，但是进行的是终止处理工作，因此终止处理又称为线程结束的第二个阶段，而受理终止要求则被称为线程结束的第一个阶段。

在进行线程两阶段终结的时候需要考虑如下几个问题。

- 第二阶段的终止保证安全性，比如涉及对共享资源的操作。
- 要百分之百地确保线程结束，假设在第二个阶段出现了死循环、阻塞等异常导致无法结束
- 对资源的释放时间要控制在一个可控的范围之内。

23.2、Two Phase Termination 的示例

```
package com.bjsxt.chapter23;

import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

public class CounterIncrement extends Thread {

    private volatile boolean terminated = false;
    private int counter = 0;

    @Override
    public void run() {
        try {
            while (!terminated) {
                System.out.println(Thread.currentThread().getName() + " " + counter++);

TimeUnit.MILLISECONDS.sleep(ThreadLocalRandom.current().nextInt(1000));
            }
        } catch (InterruptedException e) {
            // e.printStackTrace();
        } finally {
            this.clean();
        }
    }

    private void clean() {
        System.out.println("二阶段终止操作被执行。" + counter);
    }

    public void close() {
        this.terminated = true;
        this.interrupt();
    }

}
```

23.3、使用 Two Phase Termination 完成通信资源关闭

服务端代码

```
package com.bjsxt.chapter23.demo02;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 服务端
```

```
 */
public class AppServer extends Thread {

    // 端口
    private int port;
    // 静态端口
    private final static int DEFAULT_PORT = 12722;
    // 是否启动服务标记
    private volatile boolean isRunning = true;
    // 服务端
    private ServerSocket server;
    // 客户端集合
    private List<ClientHandler> clientHandlers = new ArrayList<>();
    // 线程池
    private final ExecutorService threadPool = Executors.newFixedThreadPool(10);

    public AppServer() {
        this(DEFAULT_PORT);
    }

    public AppServer(int port) {
        this.port = port;
    }

    @Override
    public void run() {
        System.out.println("服务端已启动，启动端口为：" + this.port);
        try {
            server = new ServerSocket(port);
            while (isRunning) {
                Socket client = this.server.accept();
                ClientHandler clientHandler = new ClientHandler(client);
                threadPool.submit(clientHandler);
                this.clientHandlers.add(clientHandler);
                System.out.println(client.getLocalAddress() + " 客户端已成功接入服务端。
");
            }
        } catch (IOException ioException) {
            // ioException.printStackTrace();
            this.isRunning = false;
        } finally {
            // 二阶段处理
            this.dispose();
        }
    }

    // 二阶段处理的业务
    private void dispose() {
        // 关闭所有的客户端
        clientHandlers.stream().forEach(ClientHandler::stop);
        // 关闭线程池
        this.threadPool.shutdown();
        System.out.println("服务端二阶段终止操作被执行。");
    }

    // 对外显示声明的关闭方法
    public void shutdown() throws IOException {
        // 如果已经停止则 return
        if (!isRunning)
            return;

        this.isRunning = false;
```

```
        this.interrupt();
        this.server.close();
        System.out.println("服务端已关闭。");
    }

}
```

客户端代码

```
package com.bjsxt.chapter23.demo02;

import java.io.*;
import java.net.Socket;

/**
 * 客户端
 */
public class ClientHandler implements Runnable {

    private final Socket socket;
    // 是否运行标记
    private volatile boolean isRunning = true;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try (InputStream is = socket.getInputStream();
             OutputStream os = socket.getOutputStream();
             BufferedReader br = new BufferedReader(new InputStreamReader(is));
             PrintWriter pw = new PrintWriter(os)) {
            while (isRunning) {
                String message = br.readLine();
                if (null == message)
                    break;
                System.out.println("收到客户端的消息为：" + message);
                pw.write("ehco message: " + message + "\r\n");
                pw.flush();
            }
        } catch (IOException ioException) {
            // ioException.printStackTrace();
            this.isRunning = false;
        } finally {
            // 二阶段处理
            this.stop();
            System.out.println("客户端二阶段终止操作被执行。");
        }
    }

    public void stop() {
        // 如果已经停止则 return
        if (!isRunning)
            return;
        // 说明正在运行，调用 stop 后修改为 false
        this.isRunning = false;
        try {
            this.socket.close();
        } catch (IOException ioException) {
            ioException.printStackTrace();
```

```
        }
        System.out.println("客户端已退出。");
    }

}
```

测试类

```
package com.bjsxt.chapter23.demo02;

import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class AppServerClient {

    public static void main(String[] args) throws InterruptedException, IOException {
        AppServer appServer = new AppServer(13345);
        appServer.start();

        TimeUnit.SECONDS.sleep(20);
        appServer.shutdown();
    }

}
```