

《面向对象设计与构造》

Lec07-并发场景的分析与设计

OO2021课程组

北京航空航天大学计算机学院

提纲

- 并发场景分析
- 综合性设计
- 线程管理问题
- SOLID设计原则
- 作业解析

软件开发阶段划分

- 软件开发经历的核心阶段
 - 分析阶段：理解用户需求，识别关键业务场景和数据
 - 设计阶段：建立可扩展的架构，确定核心类及其关系，设计核心算法
 - 实现阶段：使用具体语言实现程序，尽可能使用可重用的库资源，代码风格、确保简洁和易理解至关重要
 - 测试阶段：尽可能使用自动化手段来开展测试，代码变动就回归测试，确保软件质量的稳定性
 - 单元测试：以类为单位开展测试，覆盖所有的分支和语句
 - 集成测试：针对类之间的交互协同开展测试
 - 黑盒测试：基于需求来开展测试，关注功能、鲁棒性和性能等

面向对象不只是编程实现

- 面向对象首先源自于程序设计
- 逐渐人们发现它其实是一套方法学，思维方法
 - 对象化是人们认识和理解世界的基本思维
 - UML是这套方法学的集大成者，可用于各个领域的问题分析和设计
- 上升到方法学层次来讨论面向对象不是故弄玄虚
 - 帮助我们建立系统性的分析设计方法

面向对象思维

- 面向对象思维(object-oriented thinking)是以对象为视角的思维
 - 有哪些对象?
 - 对象做什么?
 - 对象之间有什么连接/关系?
- 在不同阶段的面向对象思维
 - 分析阶段：理解和识别需求中的“对象”
 - 设计阶段：构造“对象”来实现需求
 - 实现阶段：使用程序语言来实现“对象”
 - 测试阶段：逐个检查“对象”的功能和性能，然后对“对象集成”进行测试

面向对象之“分析思维”

- 对象化思维来理解软件需求
 - 识别类-----数据、控制、设备...
 - 识别类的职责-----管理数据、实施控制/控制策略、输入输出处理
 - 基于类来分析软件功能-----多个类协同的方式来阐述功能场景
- 需求一般都是从用户视角来规定软件的功能和性能
 - 输入、输出、流程
 - 平台
 - 数据
- 课程作业要求在层次上相当于软件需求
 - 实际上比一般的软件需求要细致，介于软件需求和概要设计之间

面向对象之 “分析思维”

- 例子

- 新闻类、博客类网站每天会以不确定的频率发布新的消息
- 用户难以知道什么时候有信息更新
- 不能要求用户使用浏览器刷新网站页面来获得信息更新
- 开发一个网站内容更新订阅系统，功能要求如下：
 - 能够根据用户需要订阅一个网站的内容更新
 - 能够自动获得网站内容的更新，包括主题、日期和信息片段(snippet)
 - 能够自适应网站内容更新的频度
 - 能够对更新的主题、日期、信息片段进行有效管理
 - 提供对更新的全文搜索
 - 能够把来自不同网站的相同更新进行合并

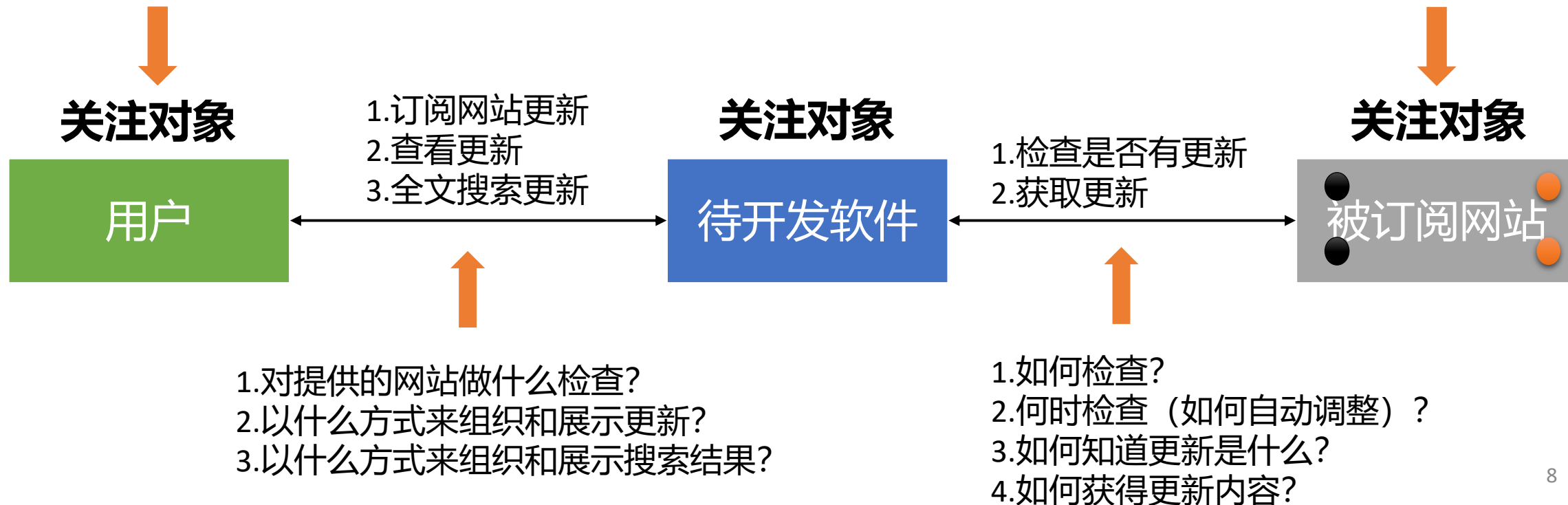


面向对象之“分析思维”

- 1.交互关系分析：待开发的**软件与用户**、**外部环境**有哪些交互？

- 1.是否区分该对象实例？
- 2.该对象有哪些感兴趣的数据？

- 1.是否区分该对象实例？
- 2.该对象有哪些感兴趣的数据？

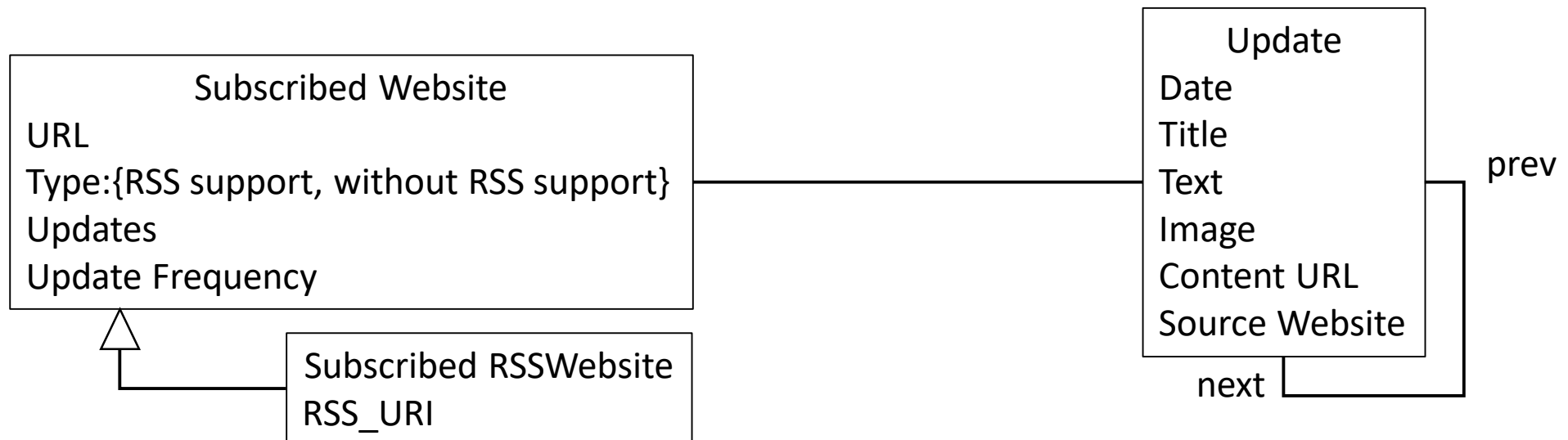


面向对象之“分析思维”

- 通过这种对象分析，确定了待开发软件的边界、对象之间的交互关系、对象交互的特征
 - 用户交互的**数据特征**：提供待订阅网站、提供更新搜索词
 - 用户交互的**并发特征**：是否有多用户的并发交互？并发的用户交互有哪些共享数据？
 - 用户交互的**时间特征**：不同类别用户的交互频度？交互持续时间？
 - 被订阅网站交互的**数据特征**：网页内容(html)、更新列表文件(xml)、网页层次结构(html)
 - 被订阅网站交互的**并发特征**：是否与多个网站并发交互？并发交互有哪些共享数据？
 - 被订阅网站交互的**时间特征**：不同网站的更新频度、通信速度
- 在分析过程中理解和细化了软件需求，并识别出潜在的问题
 - 如何找到网页中有更新？
 - 有更新列表文件的情况：按照更新列表来识别和提取更新
 - 无更新列表文件的情况：按照给定的页面来识别和提取更新

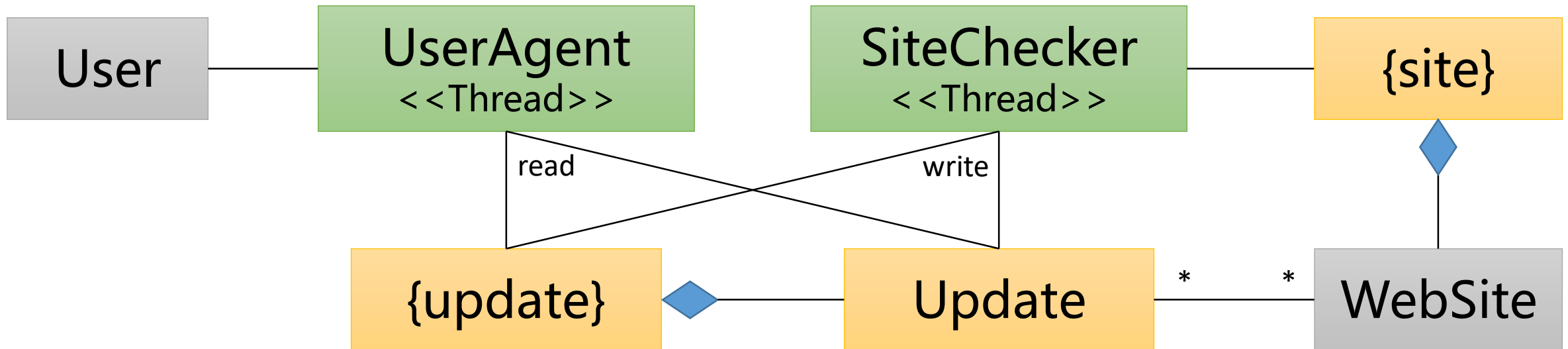
面向对象之“分析思维”

- 2.进入“待开发系统”对象内部进行分析
 - 运行平台：单机/Web服务器？ ←与外部对象的**交互并发特征**
 - 管理哪些数据：被订阅网站、网站内容更新
 - 对数据的管理手段：维护订阅网站列表、管理更新、管理订阅网站与更新之间的关系、维护订阅网站的更新检查频度、检查不同订阅网站的更新是否相同、检查订阅网站是否有更新、提取订阅网站的更新



面向对象之“分析思维”

- 2.进入“待开发系统”对象内部进行分析
 - 识别并发主体：并发技术来处理外部的并发交互
 - 并发特征带来的数据管理针对性（共享数据）

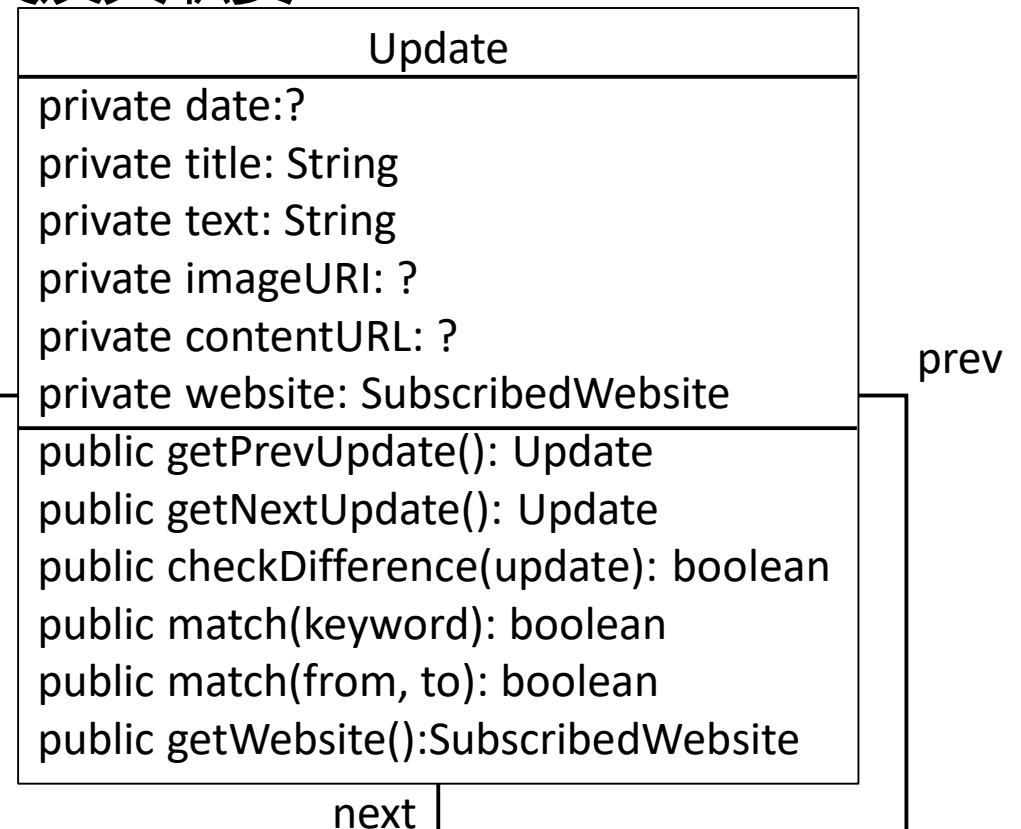
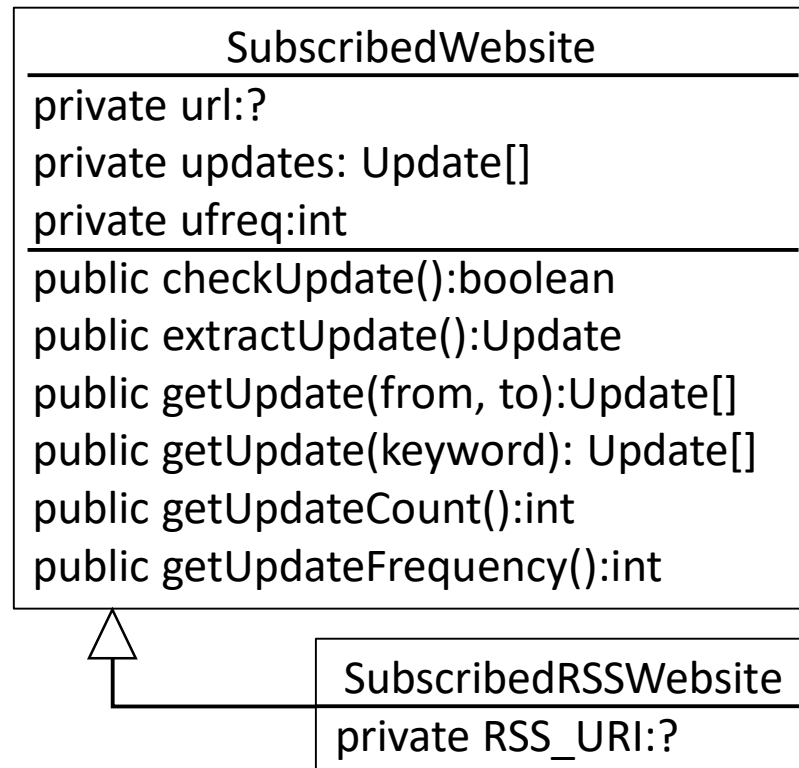


面向对象之“分析思维”

- 能够根据用户需要**订阅一个网站内容更新**
- 能够**自适应**网站内容更新的频率
- 能够**获得网站内容的更新**
- 能够**管理更新**的主题、日期、信息片段(snippet)
- 提供对更新的**全文搜索**
- 能够把来自不同网站的相同更新**进行合并**

检查是不是所有的需求都能够得到满足(如何满足是设计问题)

进行分析
类及其职责



思考题

- 云端文档协同编辑现在是很多办公系统的标配功能
- 请按照前面所介绍的策略来分析云端文档协同编辑涉及的交互关系、交互的数据特征、并发特征和时间特征



系统性的设计考虑

- **系统性**：全面、有条理的梳理主要的相关因素，从而形成完整和闭合的逻辑体
 - 设计需要关心的核心要素：核心功能、核心性能、扩展性
- 我们关注
 - 并发结构的识别和设计
 - 数据管理的分类设计
 - 数据中内在结构的表达
 - 类方法职责的简化
 - 类之间的协同
 - 空间与时间的平衡

并发结构的识别与设计

多线程与多种线程

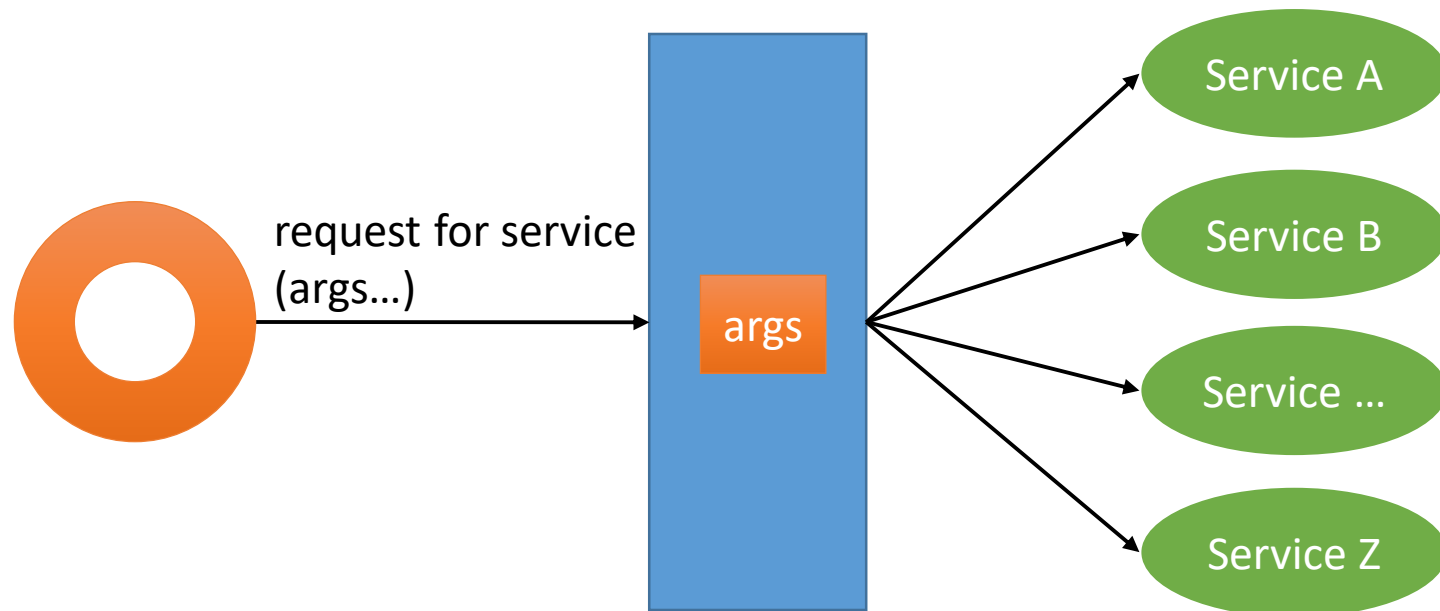
- 识别并发行为

- 待实现软件与外部多个对象进行相对独立的并发交互
 - 按照交互对象提供的**交互机制**进行分类
 - 通过HTTP机制来获取信息
 - 通过Streaming机制来获取信息
 - 按照交互对象提供的**交互内容**进行分类
 - 有RSS支持的网站和无RSS支持的网页
 - 对应两类不同的线程设计：RSSWebSiteChecker、WebPageChecker
- 待实现软件要处理的数据有显著的重复模式，且处理相对独立
 - Web系统的日志处理：日志记录着用户访问Web系统的行为，具有典型的模式；处理目标是提取用户行为、出现的问题、系统响应时间等，相对独立
 - 扫描一个规模较大的字符串数组看是否出现某个关键词：重复模式是不断在一个局部的数组中查找关键词的出现情况

并发结构的识别与设计

- 识别并发行为

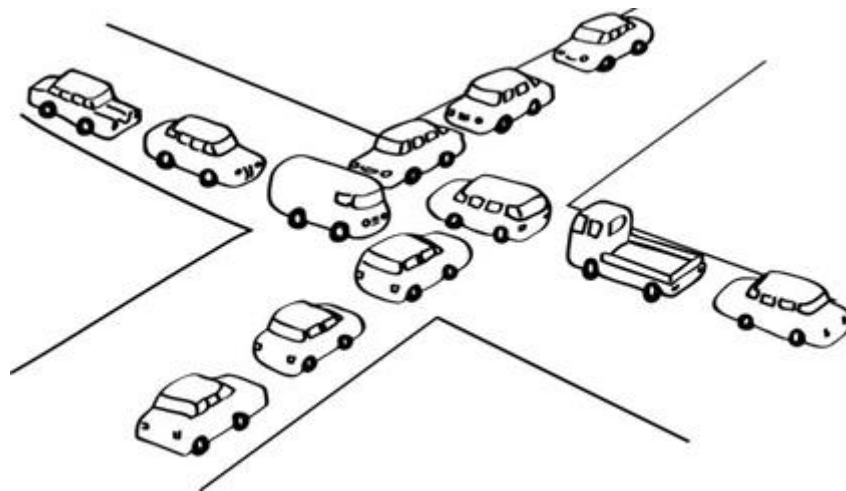
- 设想一个软件持续监听某网络端口以获得一种特定请求 “request for service”，根据服务参数的不同，该软件需要进行不同的处理。



需要几类线程来支撑这个软件？

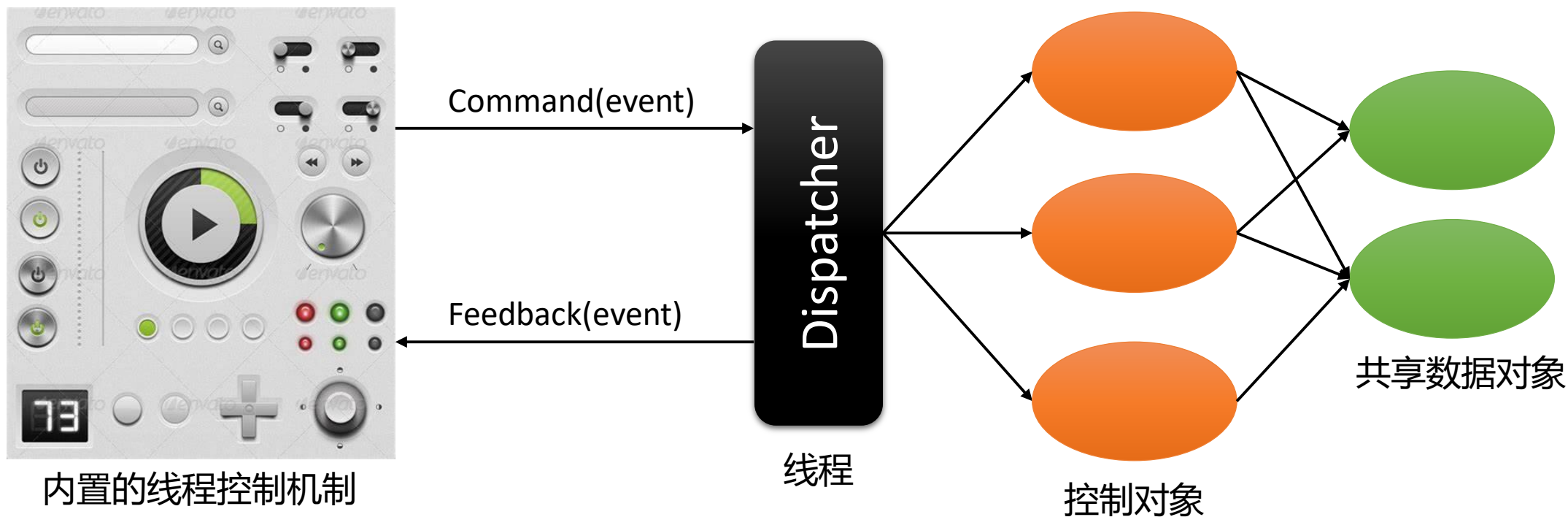
并发场景的识别与设计

- 三类并发行为场景
 - 人机交互的并发
 - 多client请求处理的并发
 - workflow处理的并发



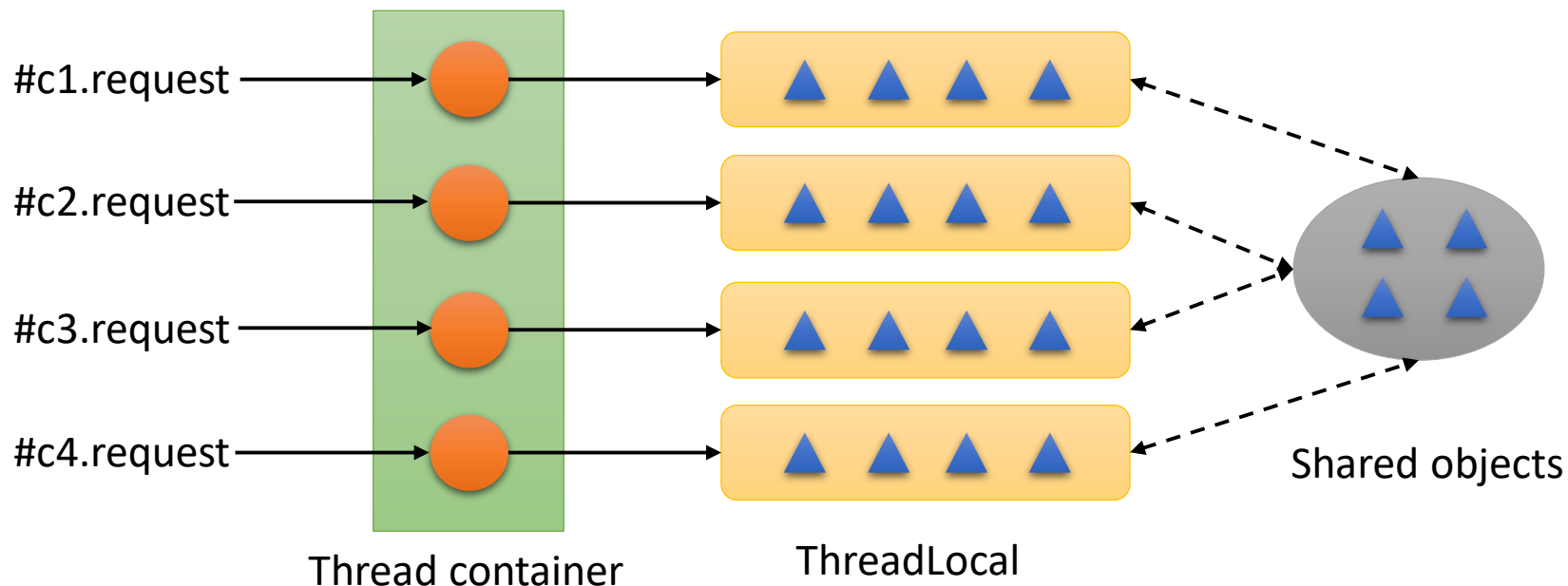
人机交互中的并发识别与设计

- Event dispatch
 - 确保界面响应的即时性，随时可以发出command，且在感兴趣的数据对象状态发生变化时即时在GUI上观察到相应的feedback



多Client请求处理的并发识别与设计

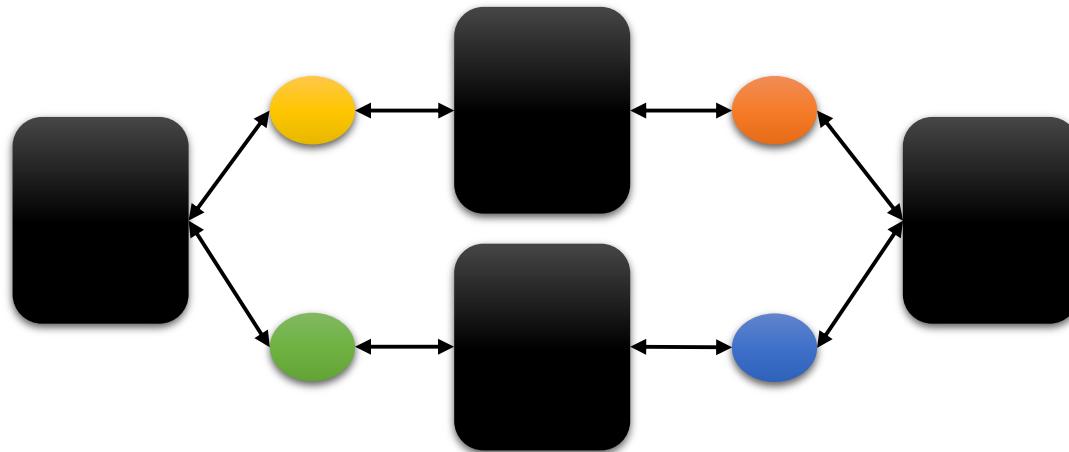
- 按照client来隔离请求的处理
 - 每个client请求的发出时机不同
 - 每个client请求涉及的数据可能具有privacy要求
 - **隔离式处理**可提高系统的吞吐率、可靠性和安全性(security)



ThreadLocal是JDK提供的一个类，用于隔离线程对数据的访问，即只能被特定线程对象访问。

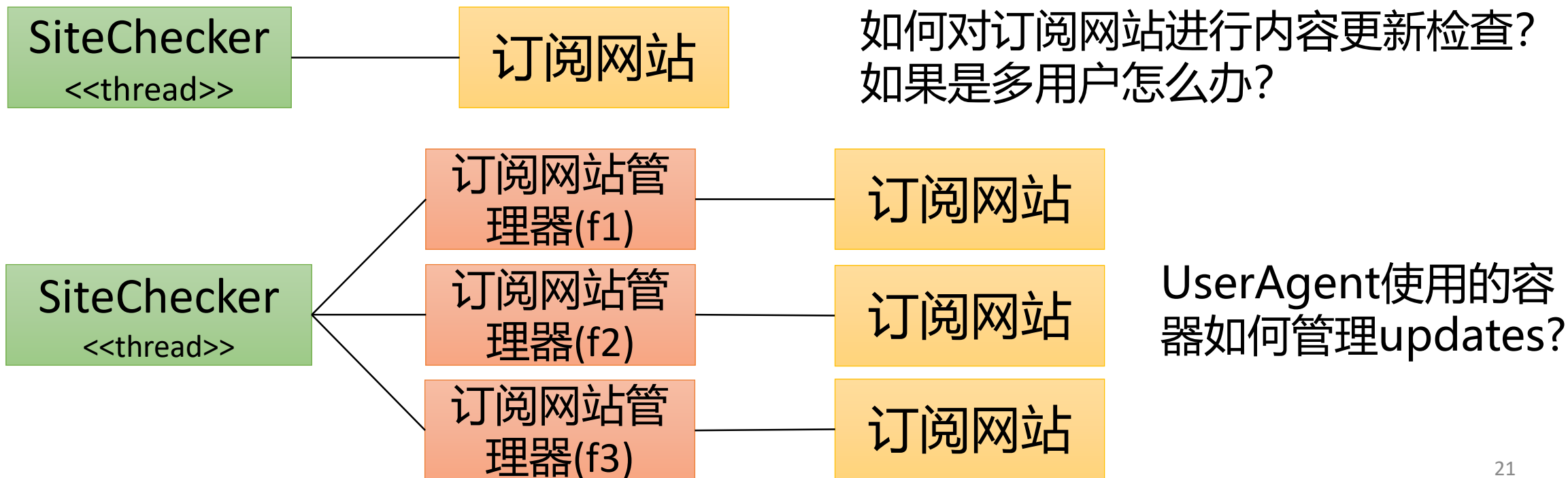
workflow处理的并发识别与设计

- workflow Work flow
 - 多个任务之间相对独立，但在特定数据上形成依赖关系
 - 整体具有Pipeline特征
- 隔离生产者和消费者
 - 通过共享数据访问控制来隔离依赖关系
 - 复杂的并发处理场景，容易出现安全问题和轮询问题



数据管理的分类设计

- 数据管理是一个基础功能，频繁访问，需要考虑性能因素
 - 并不是简单做一个容器就能满足功能和性能要求的问题
 - 网站内容更新订阅系统中，如何管理用户订阅的网站？



数据管理的分类设计

- 数据管理需要考虑数据的空间关系特征
 - 出租车呼叫系统如何管理出租车和乘客?
 - 出租车：位置、信用、公司
 - 乘客：VIP、普通会员和普通乘客
- 从问题域出发，对被管理对象的特征和行为进行分类
- 按照类别进行管理，构造相应的管理容器和访问机制
 - 按照位置区域来管理出租车和乘客请求
 - 状态维护简单，消息发布便捷

数据内在结构的表达

- 问题域中很多数据具有内在的结构
 - 程序员为了简化，往往使用字符串来表示这些数据，使得数据中的逻辑结构在代码处理中变成隐含知识
 - 电话号码
 - URL地址
 - 结构化 ~ 非结构化
- 如果软件对相应数据的处理不涉及其内在逻辑结构
 - 电话号码只是用来记录和显示
- 如果软件对相应数据的处理涉及其内在逻辑结构
 - 需根据电话号码来识别归属地，从而与地理位置相关信息进行关联

类方法职责的简化

- 类中每个方法都要完成一个完整的功能，赋予一定的职责
 - 尽量保持每个方法只做一件事情，方法间往往会有调用
- 不能混淆功能的执行时机和功能的实现场所
 - SubscribedWebsite中谁负责更新网站检查频率？
 - 方案A：单独增加一个方法updateFrequency(int freq)
 - 方案B：在checkUpdate中来更新频率
 - 方案C：在extractUpdate中来更新频率

SubscribedWebsite
private url:? private updates: Update[] private ufreq:int
public checkUpdate():boolean public extractUpdate():Update public getUpdate(from, to):Update[] public getUpdate(keyword): Update[] public getUpdateCount():int public getUpdateFrequency():int

类方法职责的简化

- 建议采用方案B：在checkUpdate中来更新频率
 - 首先SubscribedWebsite的管理已经按照其更新频率进行了分类
 - checkUpdate的执行具有简单的**周期性**
 - 如果连续**三次**发现未更新，则调整更新频率，同时把相应对象调整到别的管理类别中
 - 即简化了方法职责，又保证了效率，同时减少了反复检查导致的CPU浪费
 - 如果连续三次发现都已经更新了呢？
- 从方法职责单一性角度，把频率计算和更新单独设计成一个私有方法，被checkUpdate调用
 - checkUpdate在siteChecker线程中被调用

Gmail大致就是按照这个思路来动态调整对POP3邮箱新邮件的检查

SubscribedWebsite
private url:? private updates: Update[] private lastNhits: int private ufreq:int
public checkUpdate():boolean public extractUpdate():Update public getUpdate(from, to):Update[] public getUpdate(keyword): Update[] public getUpdateCount():int public getUpdateFrequency():int private calcFrequency():int

类之间的协同

- 设计开始时往往能够给出一个清晰的架构
- 设计进展过程中会导致整体性的丧失，甚至没注意到这个丧失
 - 随着设计推进，会不断增加新的类、方法和属性
 - 做事情多的类倾向于做更多的事情
 - 管理数据多的类倾向于管理更多的数据
- 需要时刻注意从协同角度检查和梳理设计
 - 维护简洁的设计架构，使用UML类图来观察整体态势
 - 注意检查类之间的协同关系，避免出现God类和Idiot类
 - 持续重构类之间的协同
 - 两个类之间的协同：主从响应
 - 多个类之间的协同：集中调度式协同、层次分派

空间与时间的平衡

- 空间与时间矛盾是设计中的经典问题
 - 需要综合考虑进行平衡
- 当系统功能逐渐增多之后，需要重构优化
 - 以网站更新订阅系统为例
 - 网站不仅内容会发生变化，结构也可能会发生变化
 - 非RSS网站的内容变化建立在网页快照的对比基础之上
 - 这会带来什么问题？
 - 通过缓存一些历史状态，可以加速处理，提高程序执行效率
 - 几乎所有的服务器设计都会使用cache机制

SubscribedRSSWebsite

private RSS_URI:?

private cachedSitePage: String

SubscribedWebsite

private url:?

private updates: Update[]

private lastNhits: int

private ufreq:int

private cachedSiteMap: Map

public checkUpdate():boolean

public extractUpdate():Update

public getUpdate(from, to):Update[]

public getUpdate(keyword):Update[]

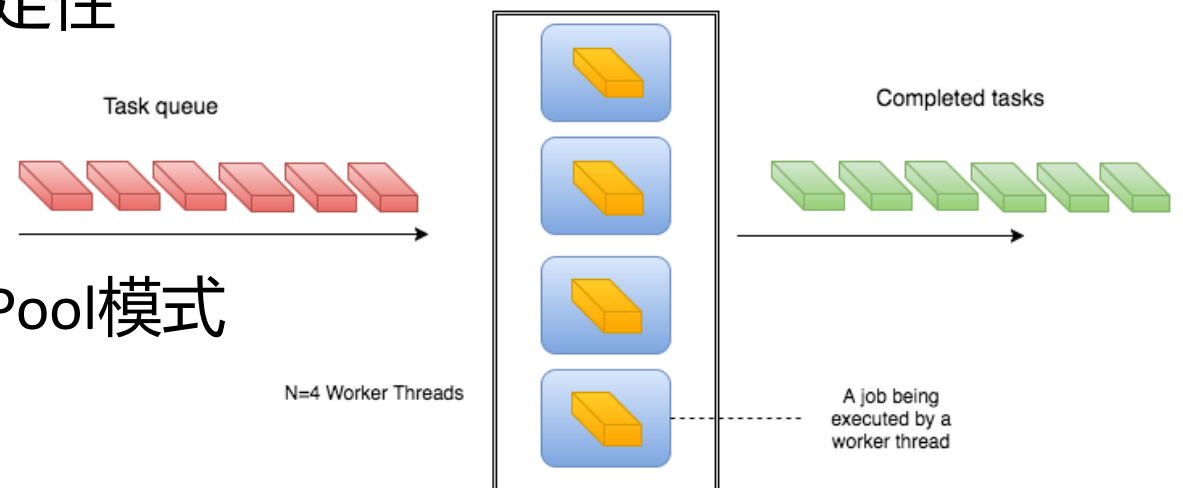
public getUpdateCount():int

public getUpdateFrequency():int

private calcFrequency():int

线程管理问题

- 对于并发任务/请求数量较少的情况，一般是任务到达时创建线程，处理完成则线程执行结束
- 对于高并发应用而言，往往会有很多任务并发到达，需要集中创建数量较多的线程，资源消耗较大
 - 线程对象重用
 - 线程数量进行控制，确保系统的稳定性
 - 线程池(ThreadPool) 模式
 - 每个线程都是一个产线工人
 - 线程池把到达任务分配给空闲工人
 - Worker Thread模式也是一种ThreadPool模式



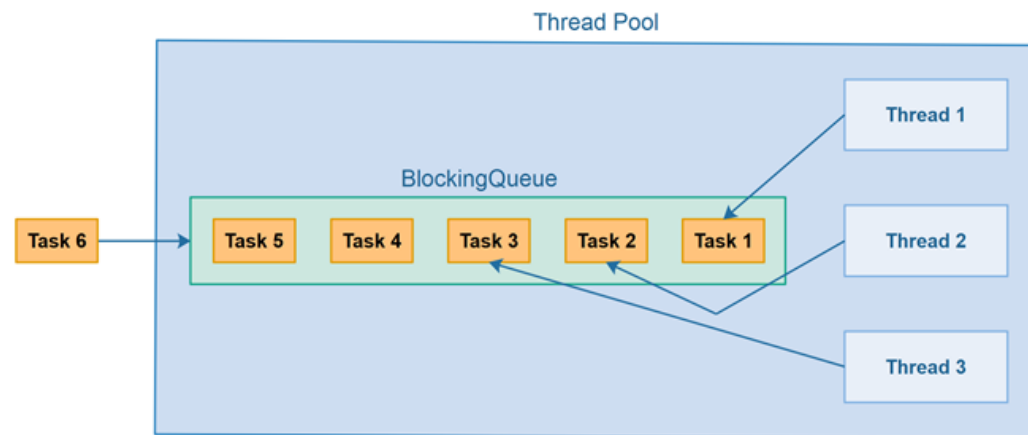
ThreadPool模式

- ThreadPool内部管理两大核心对象
 - `LinkedBlockingQueue`: 用于管理到达的任务Task
 - `Thread`对象: 用于消耗到达的任务
 - 在queue的访问上进行同步控制
- 任务是什么?
 - 实现**`Runnable`**接口的对象
 - 封装好要执行的动作和用到的对象
 - 任务间尽可能不要有共享对象
- ThreadPool提供的接口
 - `execute(task)`: 把task加入队列

如果希望能够获得任务处理的结果, 怎么办?

设置共享对象

实现**`Callable`**接口



ThreadPool模式

- java.util.concurrent.Executors是个线程池Factory
 - ThreadPoolExecutor类
 - FixedThreadPool:线程数量固定
 - CachedThreadPool:线程数量不固定
 - ScheduledThreadPoolExecutor类
 - ScheduledThreadPool:可周期循环执行的线程池
- ThreadPoolExecutor类功能丰富
 - execute(Runnable command)
 - remove(Runnable task)
 - shutdown()
- 可以灵活扩展ThreadPoolExecutor
 - 可暂停的线程池

```
public class PausableThreadPoolExecutor extends ThreadPoolExecutor {
    private boolean isPaused;
    private ReentrantLock pauseLock = new ReentrantLock();
    private Condition unpaused = pauseLock.newCondition();
    public PausableThreadPoolExecutor(...) { super(...); }
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        pauseLock.lock();
        try {
            while (isPaused) unpaused.await();
        } catch (InterruptedException ie) {
            t.interrupt();
        } finally {
            pauseLock.unlock();
        }
    }
    public void pause() {
        pauseLock.lock();
        try {
            isPaused = true;
        } finally {
            pauseLock.unlock();
        }
    }
    public void resume() {
        pauseLock.lock();
        try {
            isPaused = false;
            unpaused.signalAll();
        } finally {
            pauseLock.unlock();
        }
    }
}
```

再说Lock与Condition

- Lock提供了灵活使用机制，对应于synchronized block的monitor
 - 可以对读写进行优化
- Condition提供了线程间的交互机制
 - await(), signal(), signalAll()
- Lock提供了获得线程同步条件的方法：newCondition
 - Condition c = lock.newCondition()

```
final private Lock lock
    = new ReentrantLock();
final private Condition producedMsg
    = lock.newCondition();
final private Condition consumedMsg
    = lock.newCondition();
```

```
public void consumeMessage() {
    lock.lock();
    try {
        while (!newmsg) producedMsg.await();
        newmsg = false;
        //consume the message
        consumedMsg.signal();
    } catch (InterruptedException ie) {}
    finally { lock.unlock(); }
}
```

```
public void publishMessage() {
    lock.lock();
    try {
        while (newmsg) consumedMsg.await();
        newmsg = true;
        //publish the message
        producedMsg.signal();
    } catch (InterruptedException ie) {}
    finally { lock.unlock(); }
}
```

思考题

- 在ThreadPool模式中，任务之间一般相互独立
 - 一般根据用户发起的请求来创建任务
- 实际的业务系统中，业务请求的处理往往涉及到访问历史数据
 - 产生了共享问题
- 请思考如何在任务间引入共享访问及控制
 - 注意Task实现的是Runnable接口

设计原则是对架构的整体要求

设计模式：用以解决特定（具有普遍性）问题的一种方案，如对象构造工厂、职责代理等

体系结构：软件模块被组织/集成为系统的结构，如MVC， Pipeline

设计原则：关于架构的整体要求和约束，通过满足设计原则来获得好的设计质量

- 经典的5个设计原则(SOLID)
 - SRP、OCP、LSP、ISP、DIP
- 其他常用的设计原则

SOLID之SRP

- Single Responsibility Principle
 - 每个类或方法都只有一个明确的职责
- 类所管理的数据首先应“聚焦”
 - 类职责：使用(多个)方法，从多个方面来综合维护对象所管理的数据
 - 方法职责：从某个特定方面来维护对象的状态（更新、查询）

```
public class Elevator{  
    //fields such as floor, status, ...  
    public void move(int dest_floor){  
        //让电梯运动到目标楼层  
    }  
    public void Scan4TakingRequest (Queue q)  
    {  
        //扫描请求队列来寻找可以捎带的请求  
    }  
}
```

类/方法职责多，就意味着逻辑难以封闭，容易受到外部因素变化而变化，导致类/方法不稳定。

SOLID之OCP

Open/closed principle. In object-oriented programming, the **open/closed principle** states "software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

Open/closed principle - Wikipedia
https://en.wikipedia.org/wiki/Open/closed_principle

Bertrand Meyer



- Open Close Principle
 - 无需修改已有实现(close), 而是通过扩展来增加新功能(open)
- 当扩展电梯系统支持多部电梯的调度时
 - 改写Scheduler: 既处理单个电梯, 也处理电梯间的调度
 - 扩展Scheduler: 原来的scheduler适用于单部电梯 (维护电梯局部队列), 扩展出更上层的scheduler, 维护全局队列
- 继承是达成OCP的重要手段
 - 重用获得父类的职责和能力, close
 - 添加新的数据和方法, open
 - 重写父类的方法, open

需注意: 保持好子类 and 父类之间的交互关系

SOLID之LSP

- Liskov Substitution Principle
 - 任何父类出现的地方都可以使用子类来代替，并不会导致使用相应类的程序出现错误。
 - `BaseClass b = new BaseClass(...)` → `BaseClass b = new DerivedClass(...)`
 - 子类虽然继承了父类的属性和方法，但往往会增加一些属性和方法，可能会破坏父类的相关约束
 - 例：Queue和SortedQueue类
 - Queue类提供了一个`getLastInElement()`方法，即返回最近一次入队列的元素，其实是返回队列尾部的元素
 - SortedQueue类则对队列中的元素进行排序，每次有新元素加入队列时，按照元素之间的大小关系插入到特定的位置
 - 此时调用SortedQueue的`getLastInElement`会怎么样？如何解决这个问题？

SOLID之ISP

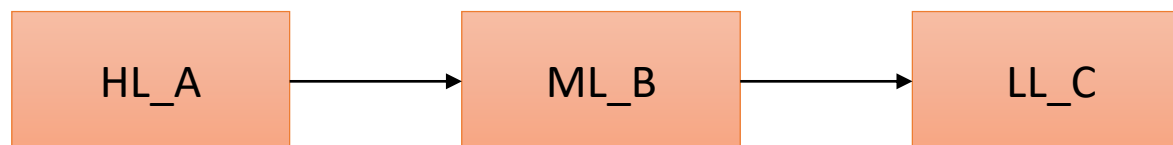
- 通过接口来建立行为抽象层次具有更好的灵活性
 - 当实现接口类时，必须要实现其中定义的所有操作，否则不能创建对象
- Interface Segregation Principle
 - 一个接口只封装一组高度内聚的操作
 - 避免封装多种可能/可选的方案
- 例：Payment是一个接口类，用来规范电子商务中的付款方式
 - 信用卡付款、储蓄卡付款、支付宝付款是三种并列的付款方式
 - 商户和用户可根据情况进行选择
- 假设Payment同时把这三类付款方式都纳入作为操作接口
 - 你在一个平台开店铺(应用)，必须要使用平台提供的Payment接口
 - 每个商品都要实现这三个接口
 - 有什么问题？如何解决？

SOLID之DIP

DIP: Dependency Inversion Principle 依赖倒置原则

```
public class CustomerManager
{
    FileLogger f; //thread safe logger
    CustomerBase customers; //thread safe container
    public void Insert(Customer c)
    {
        try{
            customers.add(c);
            f.logRecord(c.toString());
        }catch (Exception e){ f.LogError(e); }
    }
    ...
}

public class FileLogger
{
    ..file handler..
    public synchronized void LogError(Exception e){//Log Error in a physical file }
    public synchronized void LogRecord(String str){//Log Record in a physical file }
}
```



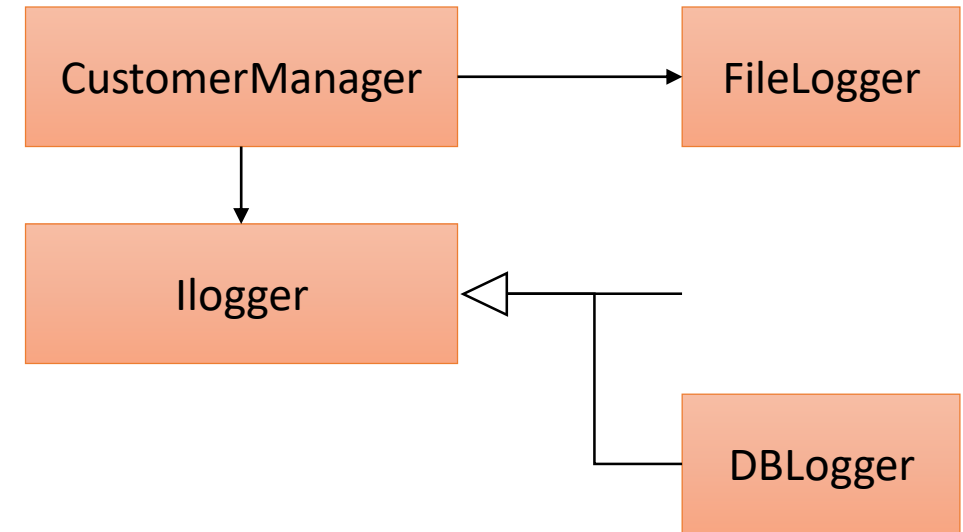
CustomerManager依赖于Filelogger,
即把信息记录到具体存储文件中



在数据库存储环境中想要重用
CustomerManager类，怎么办？

SOLID之DIP

- A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
- B. Abstractions should not depend on details. Details should depend on abstractions.



```
public class CustomerManager
{
    ...field attributes here...
    private Ilogger mylogger;
    public CustomerManager(Ilogger logger){mylogger = logger;}
    public void Insert(Customer c)
    {
        try{
            //Insert Logic
        }catch (Exception e){
            mylogger.LogError(e);
        }
    }
}
```

```
interface Ilogger
{
    public synchronized void LogError(Exception e);
}
public class FileLogger implements Ilogger
{
    public synchronized void LogError(Exception e)
    { //Log Error in a physical file }
}
public class DBLogger implements Ilogger
{
    public synchronized void LogError(Exception e)
    { //Log Error in a DB }
}
```

作业

- 本次作业仍然关注多部电梯的调度控制
 - 引入电梯类型 (A/B/C) , 运行时间参数、载客人数和可停靠楼层有差异
 - 继续上次作业, 能够动态加入电梯
 - 带来了换乘问题
- 电梯类是否需要扩展?
 - 也许参数化更合理?
- 调度器类是否需要扩展?
 - 两层调度策略不变
 - 不可直达请求的拆分: 静态还是动态?
- 性能考察乘客请求的等待时间
 - 调度时是否可以进行估计? 并据此来分配请求/或者竞争请求