《面向对象设计与构造》
Lec07-并发场景的分析与设计

OO2022课程组

北京航空航天大学计算机学院

提纲

- 面向对象分析
- 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，确保代码的简洁和易理解至关重要
  - 测试阶段：尽可能开展自动化测试，代码变动触发回归测试，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
- 也是一套适用于整个系统开发周期的方法学
  - 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
  - 不同阶段的关注点有差异
- 由编程实现技巧上升到方法学
- 需要注意：学术界对面向对象方法学仍然存在争议
  - 理性对待

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”

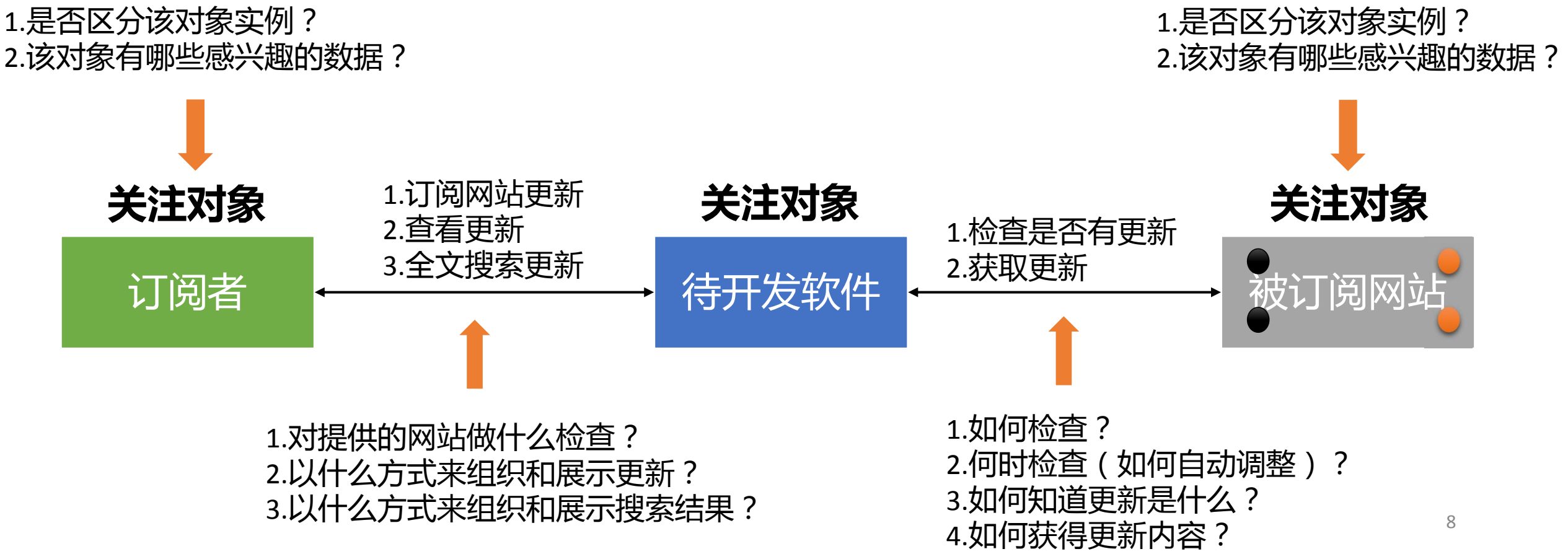
- 对象化思维来理解软件需求
  - 识别类-----数据项、控制操作、设备类别…
  - 识别类的职责-----管理数据、控制策略、输入输出
  - 基于类来分析软件功能-----功能场景表达为多个类的协同流程
- 需求一般都是从**用户视角**来规定软件的功能和性能
  - 输入、输出、流程
  - 平台
  - 数据
- 作业指导书在层次上相当于软件需求
  - 实际上比一般的软件需求要细致，规定了诸多具体约束条件

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、日期和信息摘要
    - *能够自适应网站内容更新的频度*
    - 能够对更新的主题、日期、信息摘要进行有效管理
    - 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？

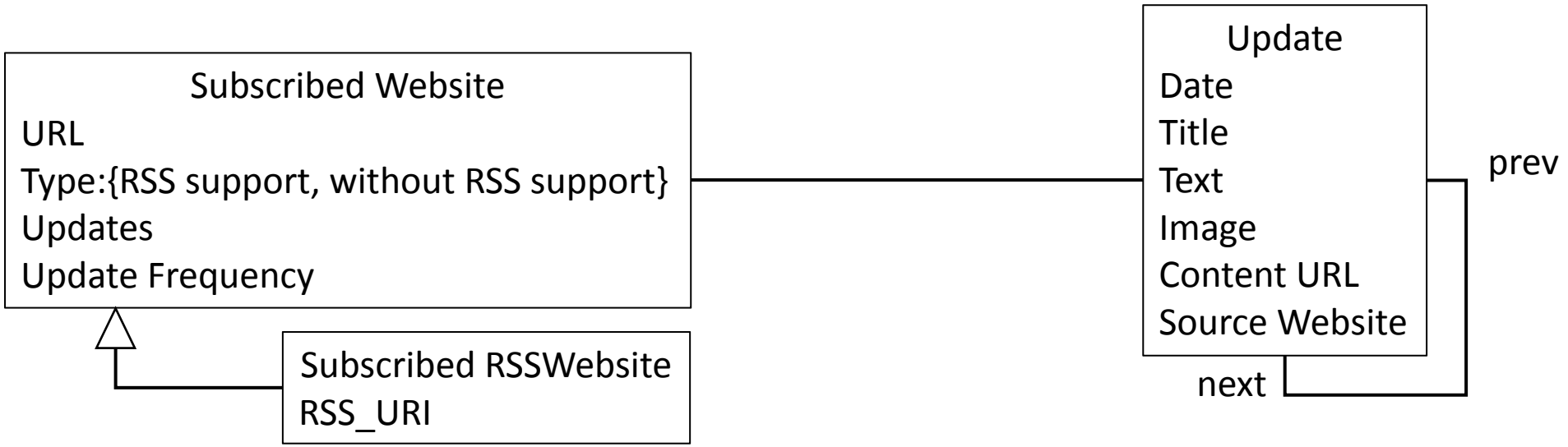


面向对象之“分析思维”

- 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
  - 订阅者交互的**数据特征**：提供待订阅网站、提供更新搜索词
  - 订阅者交互的**并发特征**：是否有多用户并发交互？并发的用户交互有哪些共享数据？
  - 订阅者交互的**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

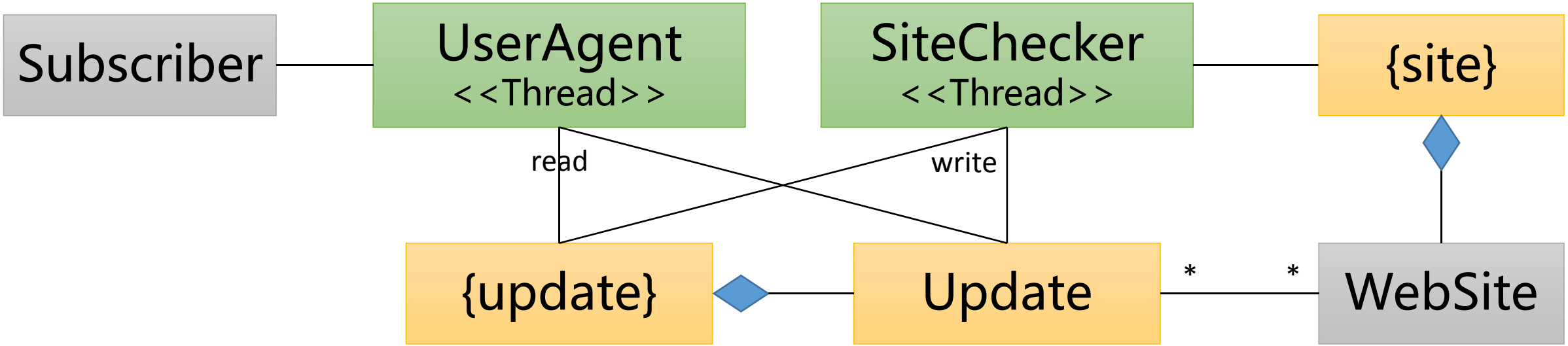
面向对象之“分析思维”

- 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）

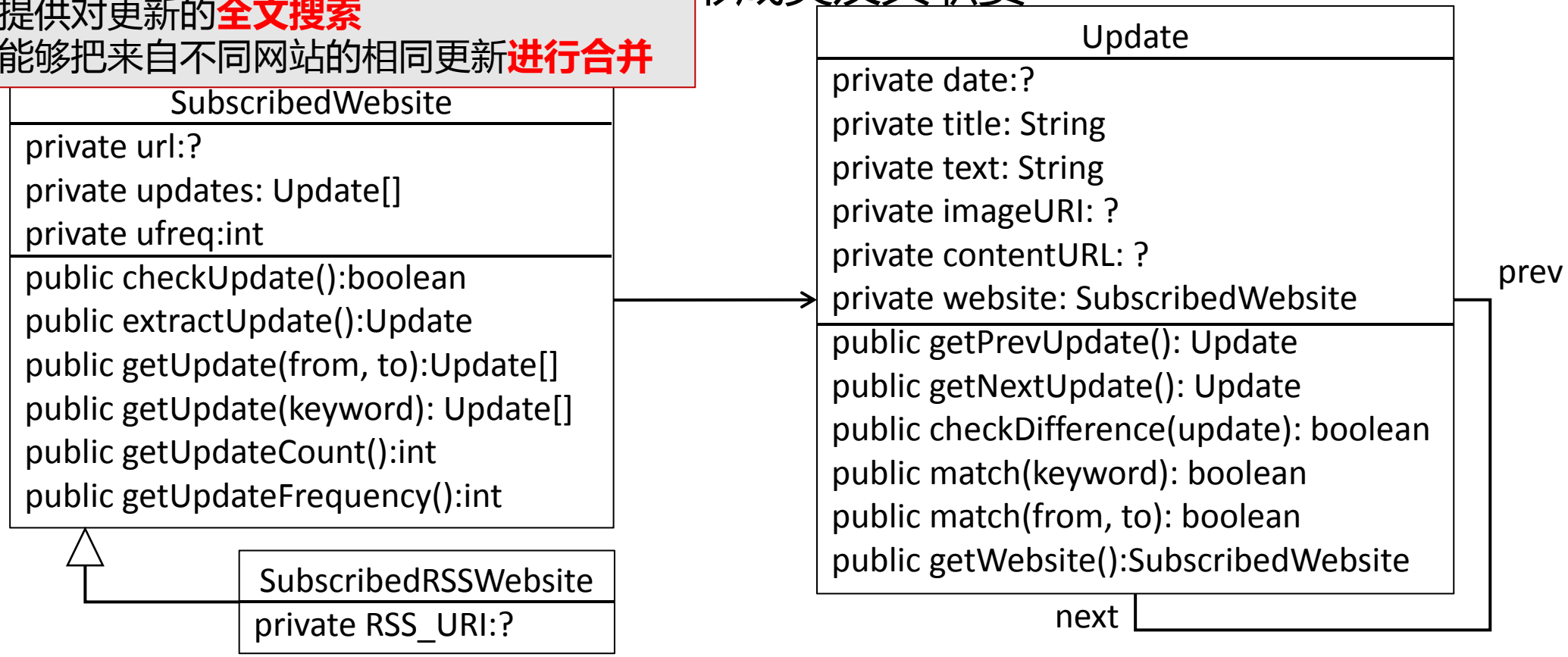


面向对象之“分析思维”

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)

- 能够根据用户需要**订阅一个网站内容更新**
  能够**自适应**网站内容更新的频度
  能够**获得网站内容的更新**
  能够**管理更新**的主题、日期、信息摘要
  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**

部进行分析
形成类及其职责



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

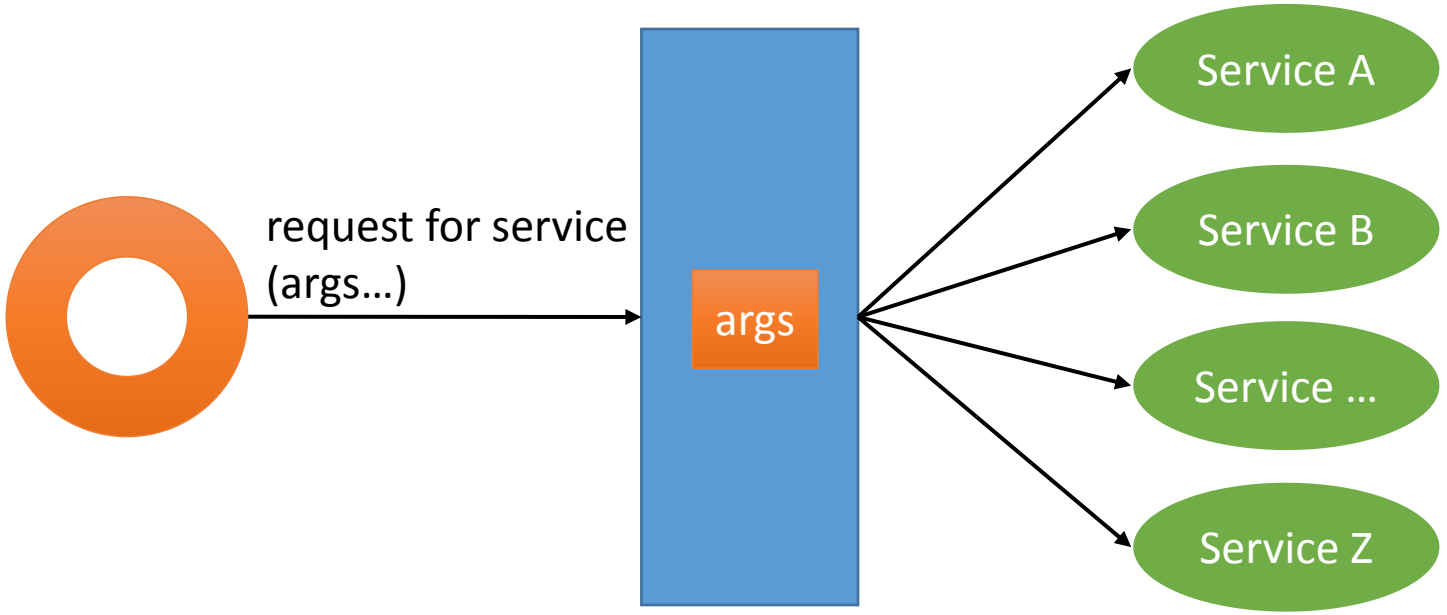
- **系统性**：全面梳理主要因素，形成完整和闭合的逻辑体
  - 核心功能、核心性能、扩展性
- 我们关注
  - 并发结构的识别和设计
  - 数据与行为中的内在结构关系识别与表达
  - 数据管理的分类设计
  - 类方法职责的简化
  - 类之间的协同
  - 空间与时间的平衡

并发结构的识别与设计

- 识别并发行为
  - 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
      - 如HTTP机制、Streaming机制
    - 按照交互对象提供的**交互内容**进行分类
      - 如RSS支持网站和无RSS支持网页，**RSSWebSiteChecker**、**WebPageChecker**
  - 要处理的数据有显著的重复模式
    - Web系统的日志处理：日志记录着用户访问Web系统的行为，具有层次结构；处理目标是提取用户行为、出现的问题、系统响应时间等
      - 多个专门提取特定信息的线程，分别提取信息
    - 扫描一个规模较大的字符串数组看是否出现某个关键词
      - 字符串切段，分别交给若干线程来并发扫描

并发结构的识别与设计

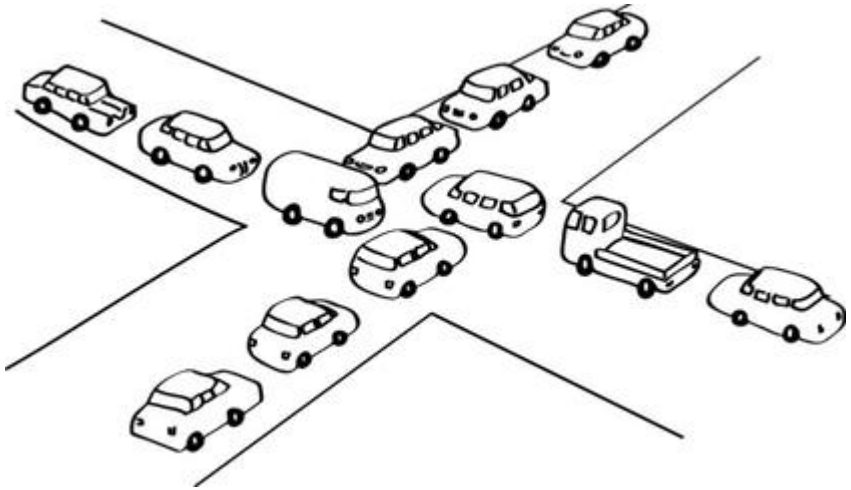
- 识别并发行为
  - 设想一个软件持续监听某网络端口以获得一种特定请求“request for service”，根据服务参数的不同，该软件需要进行不同的处理。



需要设计几类线程来构造这个软件？

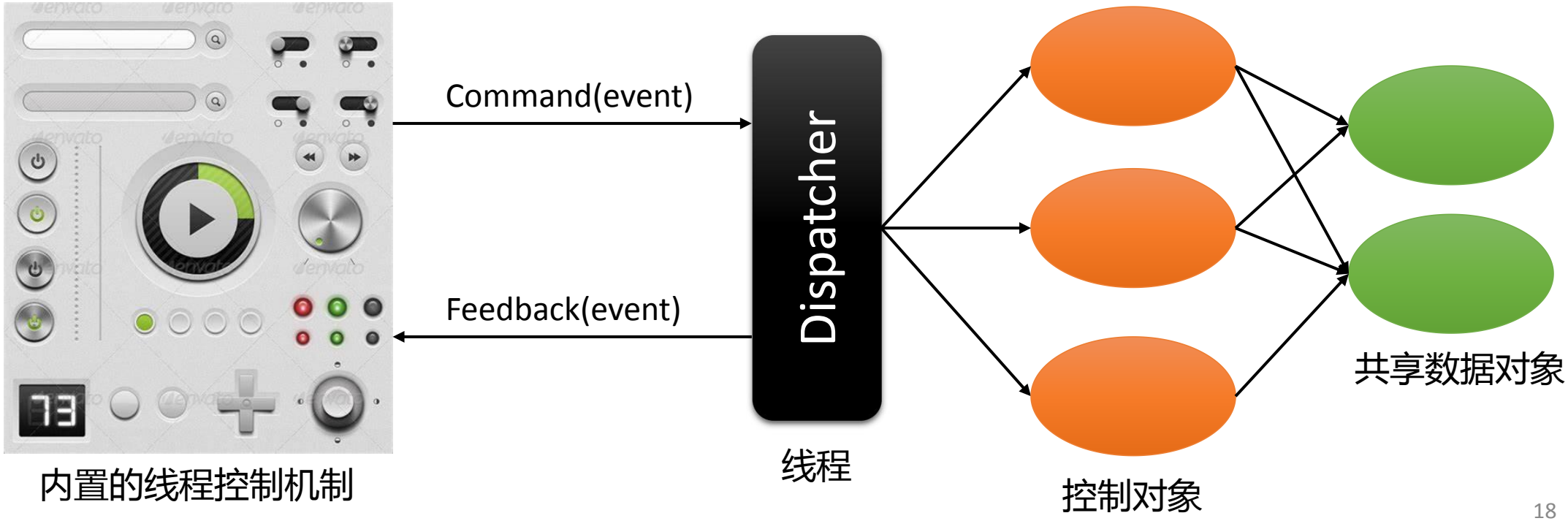
并发场景的识别与设计

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



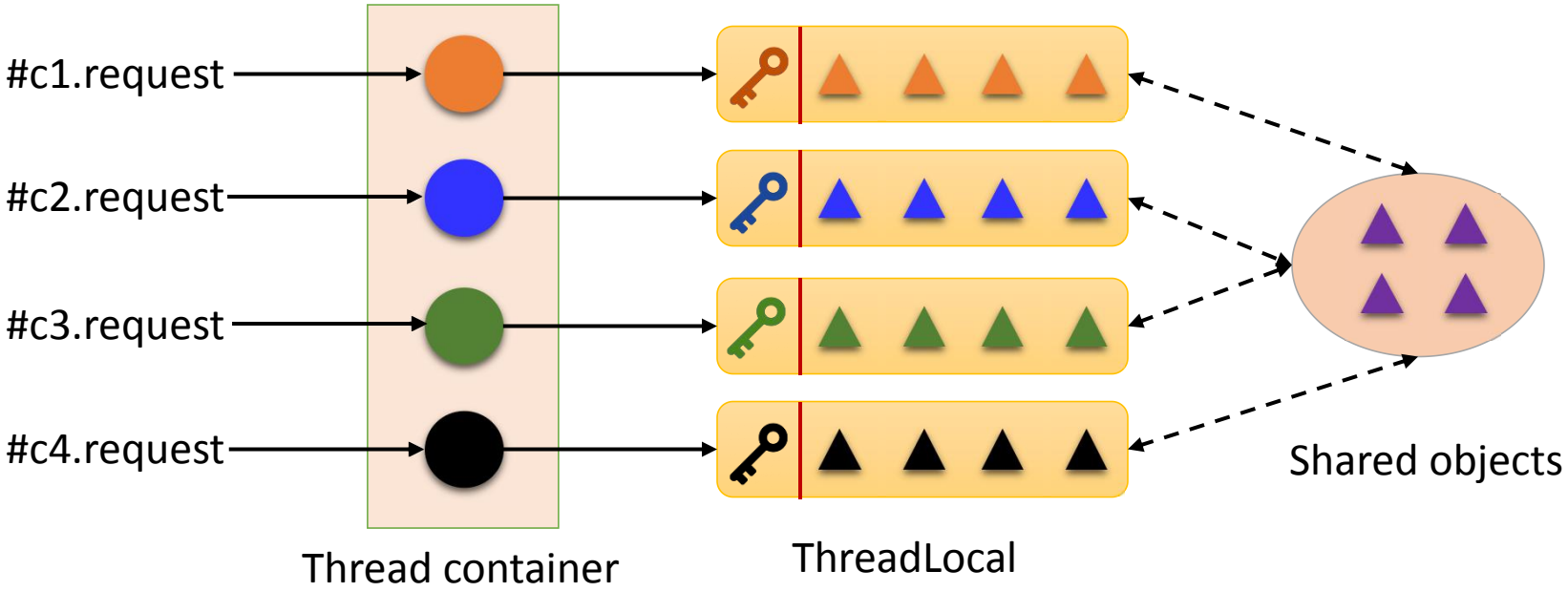
人机交互中的并发识别与设计

- 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

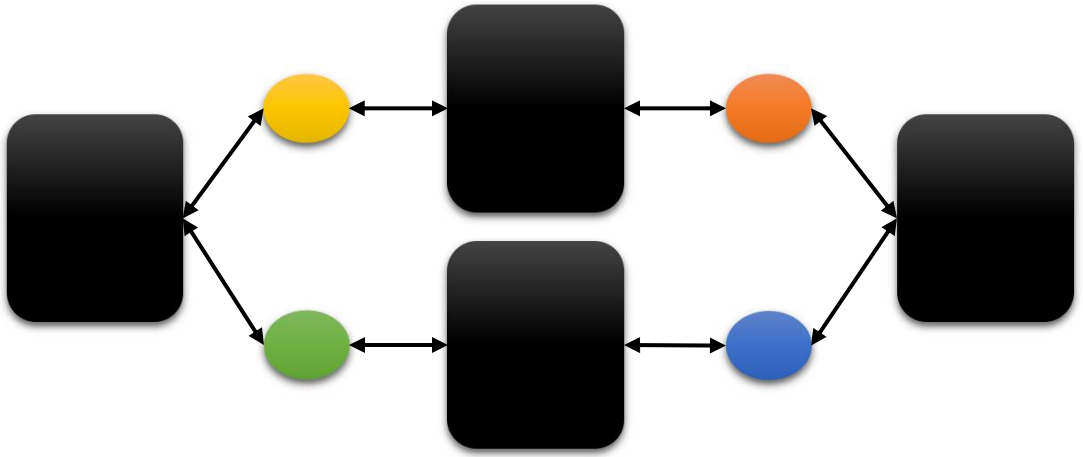
- 按照client来隔离请求的处理
  - 每个client请求的发出时机不同
  - 每个client请求涉及的数据可能具有privacy要求
  - **隔离式处理**可提高系统的吞吐率、可靠性和安全性(security)



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。

工作流处理的并发识别与设计

- 工作流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



流水线结构的动态可配置性

- 实验中给出的流水线架构有很多应用场景
- 可配置是其优点
  - Request静态给出了其处理工序要求
  - Controller据此来动态建立worker之间的**流水协作**关系
  - 一个worker处理完请求之后，通过controller“转交给”下一个worker
- Request的处理工序要求甚至可以动态变化
  - 场景：让一部有紧急任务的车辆尽可能快的从A地点行驶到B地点
    - 要求对整个道路交通的干扰最小化
  - 解决方案：根据交通流和该车辆的道路选择动态调度交通信号灯
    - 路线上的信号灯系统（线程）之间形成**流水协作**关系

使用Lock时的线程交互控制

- Lock提供了灵活使用机制
  - 读写线程间仍然有**步调协调**的问题，wait, notify机制不可用
- 与Lock配套的Condition机制
  - 确保condition与lock的语义匹配
  - 读写操作时注意condition控制的一致性
  - await(), signal(), signalAll()

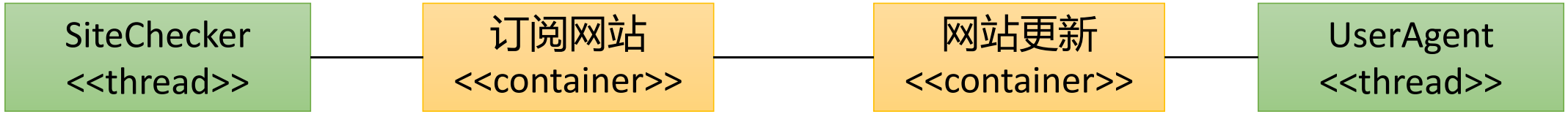
```
final private Lock lock
                = new ReentrantLock();
final private Condition producedMsg
                = lock.newCondition();
final private Condition
consumedMsg
                = lock.newCondition();
```

```
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(); }
}
```

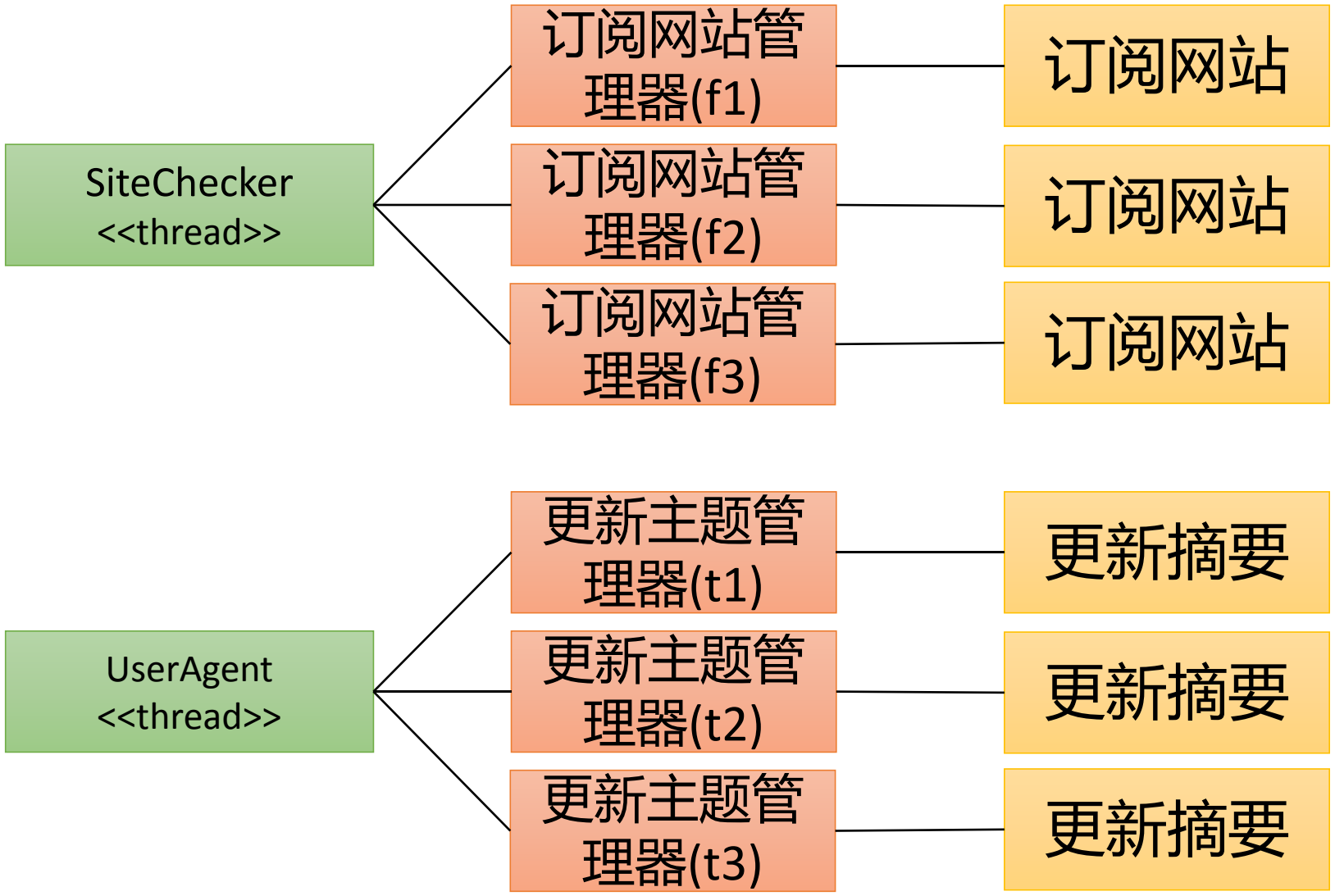
数据管理的分类设计

- 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与更新？



如何**及时**对订阅网站进行内容更新检查？
如何让用户**关心**的更新得到更高关注度？

数据管理的分类设计



按照实际**更新频率**来**分类**管理订阅网站

按照用户**兴趣主题**来**分类**管理更新摘要

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

空间与时间的平衡

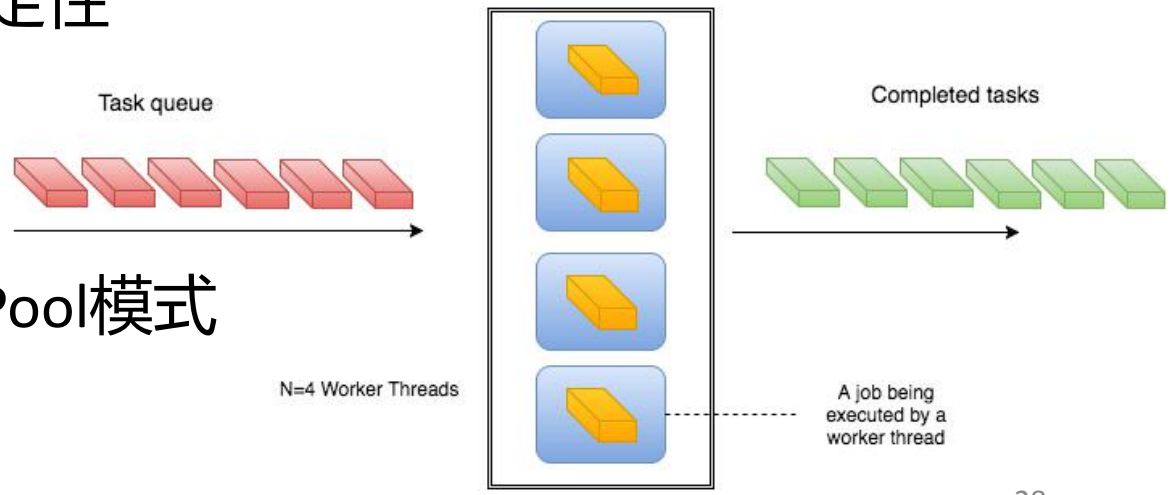
- 空间与时间矛盾是设计中的经典问题
  - 平衡考虑
- 当系统功能逐渐增多之后，需要重构优化
  - 网站内容会发生变化，结构也会发生变化
  - 非RSS网站的内容更新检测手段是网页快照对比
    - 这会带来什么问题？
- 缓存历史状态进行加速处理，提高程序执行效率
  - 几乎所有的服务器设计都会使用cache机制

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

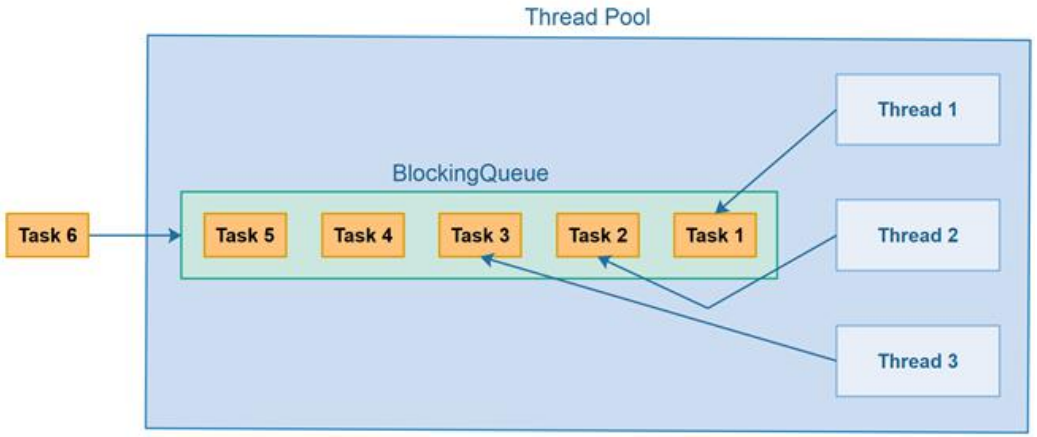
线程管理问题

- 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



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();
         }
       }
}
```

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

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 Meye

- 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：保持好子类和父类之间的交互关系

SOLID之LSP

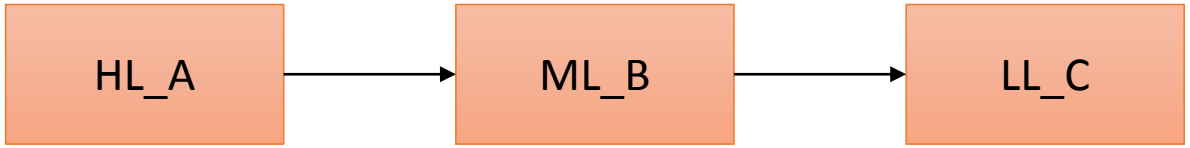
- Liskov Substitution Principle
  - 任何父类出现的地方都可以使用子类来代替，并不会导致使用相应类的程序出现错误。
    - BaseClass b = new BaseClass(…) → BaseClass b = new DerivedClass(…)
  - 子类虽然继承了父类的属性和方法，但往往会增加一些属性和方法，可能会破坏父类的相关约束
  - 例：Queue和SortedQueue类
    - Queue类提供了一个getLastInElement()方法，即返回最近一次入队列的元素，其实现是返回队列尾部的元素
    - SortedQueue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



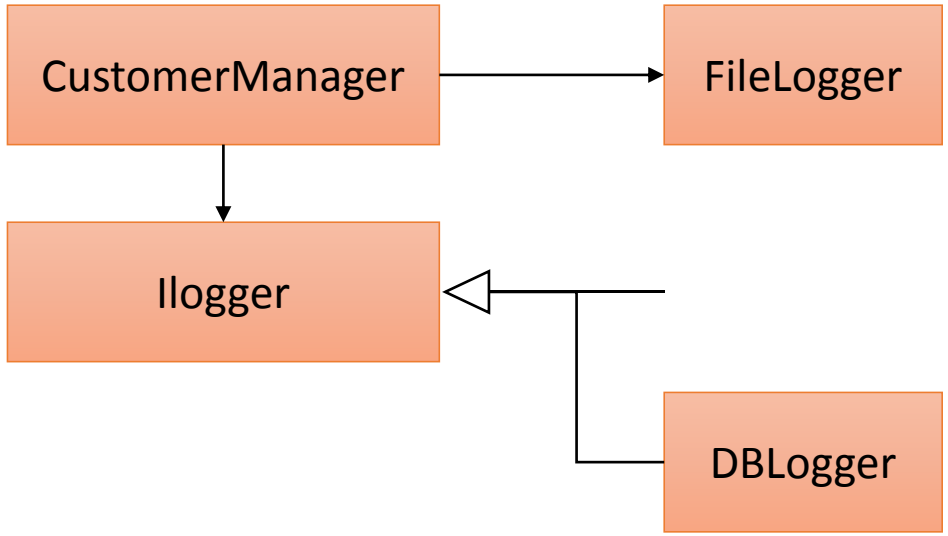

```
public class CustomerManager
{    FileLogger f;  //thread safe logger
     CustomerBase customers; //thread safe container
     public void Insert(Customer c)
     {
        try{
            customers.add(c);
            f.logRecord(c.toString());
        }catch (Exception e){ f.LogError(e); }
     }
     …
}
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
  - 横向电梯的楼座停靠可配置
  - 电梯容量和运行时间代价可配置
- 引出了换乘问题
  - 必须要换乘：路径可达性问题
  - 可能要换乘：**路径代价估计**问题（运行时间、停靠时间、**换乘等待时间**）
- 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？