SA

第一章

1. 软件危机的表现： 软件成本日益增长。开发进度难以控制。软件质量差。软件维护困难

2. 软件危机的原因： 用户需求不明确。缺乏正确的理论指导。软件规模越来越大。软件复杂度越来越高

3. 软件工程的三要素： 方法，工具，过程

4. 构件概念： 构件是指语义完整、语法正确和有可重用价值的单位软件，是软件重用过程中可以明确辨识的系统；结构上，它是语义描述、通讯接口和实现代码的复合体

5. 常见的构件模型： OMG的CORBA。Sun的EJB。Microsoft的DCOM（分布式构件对象模型）

6. 构件分类方法： 关键字分类法。刻面分类法。超文本组织方法。

7. 软件体系结构的定义： 软件体系结构为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素的描述、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。（软件体系结构不仅指定了系统的组织结构和拓扑结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理）

8. 软件体系结构的意义： 1体系结构是风险承担者进行交流的手段 2 体系结构是早期设计决策的体现 3软件体系结构是可传递和可重用的模型

第二章：

1.软件体系结构模型及功能：

> 结构模型：以体系结构的构件、连接件和其他概念来刻画结构，并力图通过结构来反映系统的重要语义内容，包括系统的配置、约束、隐含的假设条件、风格、性质等。
>
> 框架模型：侧重于整体的结构。主要以一些特殊的问题为目标建立只针对和适应该问题的结构。
>
> 动态模型：是对结构或框架模型的补充，研究系统的“大颗粒”的行为性质。例如，描述系统的重新配置或演化。动态可以指系统总体结构的配置、建立或拆除通信通道或计算的过程。
>
> 过程模型：过程模型研究构造系统的步骤和过程。

功能模型：认为体系结构是由一组功能构件按层次组成，下层向上层提供服务。可以看作是一种特殊的框架模型。

2.“4+1”视图模型及功能作用：

逻辑视图：主要支持系统的功能需求，即系统提供给最终用户的服务。在逻辑视图中，系统分解成一系列的功能抽象，这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析，而且可用作标识在整个系统的各个不同部分的通用机制和设计元素。在面向对象技术中，通过抽象、封装和继承，可以用对象模型来代表逻辑视图，用类图来描述逻辑视图。

开发视图：也称模块视图，主要侧重于软件模块的组织和管理。开发视图要考虑软件内部的需求，如软件开发的容易性、软件的重用和软件的通用性，要充分考虑由于具体开发工具的不同而带来的局限性。开发视图通过系统输入输出关系的模型图和子系统图来描述。

进程视图：侧重于系统的运行特性，主要关注一些非功能性的需求。进程视图强调并发性、分布性、系统集成性和容错能力，以及从逻辑视图中的主要抽象如何适合进程结构。它也定义逻辑视图中的各个类的操作具体是在哪一个线程中被执行的。进程视图可以描述成多层抽象，每个级别分别关注不同的方面。在最高层抽象中，进程结构可以看作是构成一个执行单元的一组任务。它可看成一系列独立的，通过逻辑网络相互通信的程序。它们是分布的，通过总线或局域网、广域网等硬件资源连接起来。

物理视图：主要考虑如何把软件映射到硬件上，它通常要考虑到系统性能、规模、可靠性等。解决系统拓扑结构、系统安装、通讯等问题。当软件运行于不同的节点上时，各视图中的构件都直接或间接地对应于系统的不同节点上。因此，从软件到节点的映射要有较高的灵活性，当环境改变时，对系统其他视图的影响最小。

场景视图：可以看作是那些重要系统活动的抽象，它使四个视图有机联系起来，从某种意义上说场景是最重要的需求抽象。在开发体系结构时，它可以帮助设计者找到体系结构的构件和它们之间的作用关系。同时，也可以用场景来分析一个特定的视图，或描述不同视图构件间是如何相互作用的。场景可以用文本表示，也可以用图形表示。

3.核心模型的基本元素：构件、连接件、配置、端口、角色。

4.生命周期模型阶段：

软件体系结构的非形式化描述阶段；规范描述和分析阶段；求精及其验证；实施；演化和扩展；提供评价和度量；终结；

第三章

9.软件体系结构风格是什么？ 是描述某一特定应用领域中系统组织方式的惯用模式

10.经典的软件体系结构风格有哪些?
1 数据流风格：批处理序列；管道/过滤器
2 调用/返回风格：主程序/子程序；面向对象风格；层次结构
3 独立构件风格：进程通讯；事件系统
4 虚拟机风格：解释器；基于规则的系统
5 仓库风格：数据库系统；超文本系统；黑板系统

11.C/S（客户-服务器）风格定义： C/S软件体系结构是基于资源不对等，且为实现共享而提出来的。它定义了工作站如何与服务器相连，以实现数据和应用分布到多个处理机上。有三个主要组成部分：数据库服务器、客户应用程序和网络。

C/S风格优点:

1. 具有强大的数据操作和事务处理能力，模型思想简单，易于人们理解和接受
2. 系统的客户应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以适合各构件的要求，这对于硬件和软件的变化显示出极大的适应性和灵活性，而且易于对系统进行扩充和缩小
3. 在C/S体系结构中，客户应用程序的开发集中于数据的显示和分析，而数据库服务器的开发则集中于数据的管理，不必在每一个新的应用程序中都要对一个DBMS进行编码。将大的应用处理任务分布到许多通过网络连接的低成本计算机上，以节约大量费用

缺点：开发成本较高。客户端程序设计复杂。信息内容和形式单一。用户界面风格不一，使用繁杂，不利于推广使用。软件移植困难。软件维护和升级困难。新技术不能轻易应用。

12.三层C/S风格： 三层：表示层，功能层，数据层。

优点：1 允许合理地划分三层结构的功能，使之在逻辑上保持相对独立性，能提高系统和软件的可维护性和可扩展性 2 允许更灵活有效地选用相应的平台和硬件系统，使之在处理负荷能力上与处理特性上分别适应于结构清晰的三层；并且这些平台和各个组成部分可以具有良好的可升级性和开放性 3 应用的各层可以并行开发，可以选择各自最适合的开发语言 4 利用功能层有效地隔离开表示层与数据层，未授权的用户难以绕过功能层而利用数据库工具或黑客手段去非法地访问数据层，为严格的安全管理奠定了坚实的基础

缺点：1 三层C/S结构各层间的通信效率若不高，即使分配给各层的硬件能力很强，其作为整体来说也达不到所要求的性能 2 设计时必须慎重考虑三层间的通信方法、通信频度及数据量。这和提高各层的独立性一样是三层C/S结构的关键问题

13．B/S风格（浏览/服务器）： 就是三层C/S应用结构的一种实现方式，其具体结构为：浏览器/Web服务器/数据库服务器

优点：1 基于B/S体系结构的软件，系统安装、修改和维护全在服务器端解决。用户在使用系统时，仅仅需要一个浏览器就可运行全部的模块，真正达到了“零客户端”的功能，很容易在运行时自动升级 2 还提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的开放性基础

缺点：1 B/S体系结构缺乏对动态页面的支持能力，没有集成有效的数据库处理功能 2系统扩展能力差，安全性难以控制 3 在数据查询等响应速度上，要远远地低于C/S体系结构 4 数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理(OLTP)应用

14.HMB风格（基于层次消息总线的体系结构风格）： HMB风格基于层次消息总线/支持构件的分布和并发，构件之间通过消息总线进行通信。 消息总线是系统的连接件，负责消息的分派、传递和过滤以及处理结果的返回。各个构件挂接在消息总线上，向总线登记感兴趣的消息类型。构件根据需要发出消息,由消息总线负责把该消息分派到系统中所有对此消息感兴趣的构件，消息是构件之间通信的唯-方式。构件接收到消息后,根据自身状态对消息进行响应,并通过总线返回处理结果。由于构件通过总线进行连接,并不要求各个构件具有相同的地址空间或局限在一台机器上。

15.HMB风格的消息总线的功能有哪些? 1 消息登记 2 消息分派和传递 3 消息过滤

第五章

1. **UML*中的关系：*

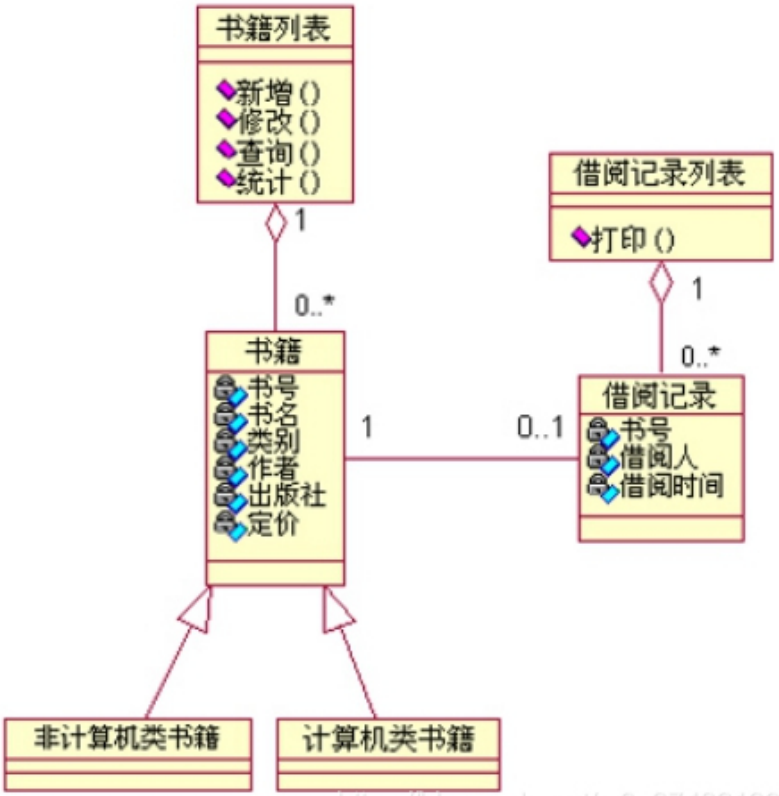
依赖，关联，泛化，实现。

2. 类之间的关系：

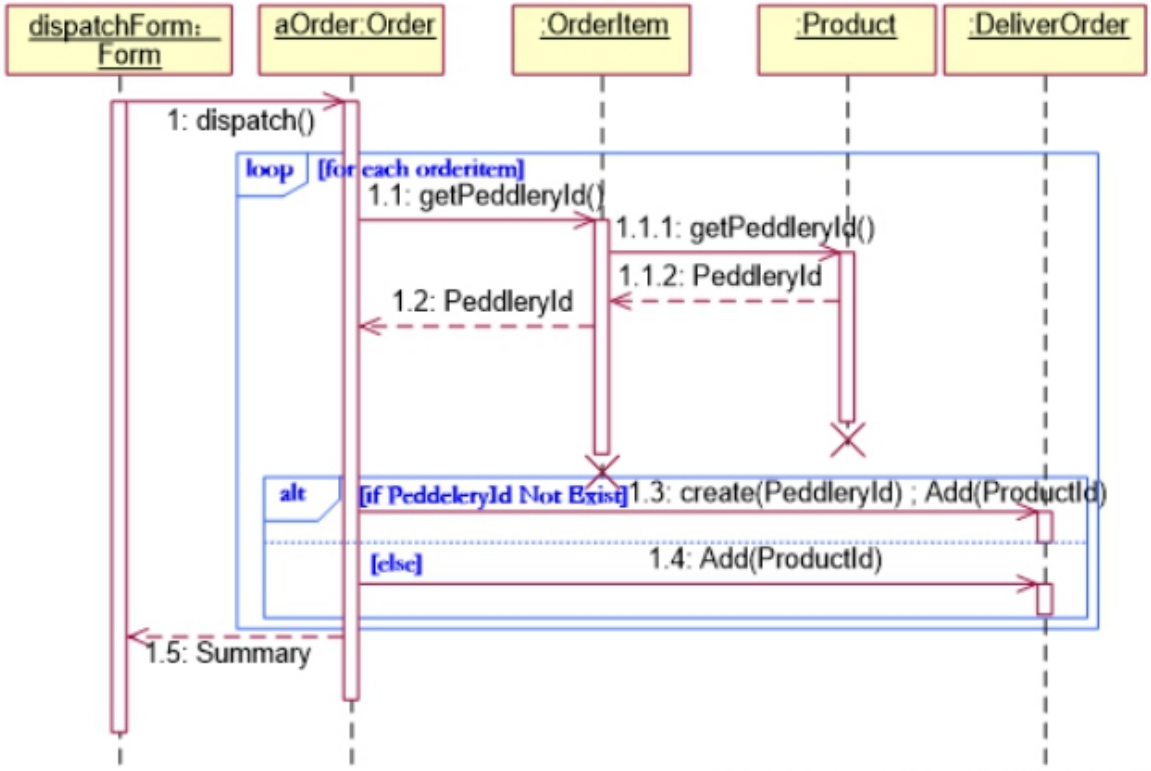
关联，依赖，泛化，聚合，组合，实现，流关系。

3．UML画法

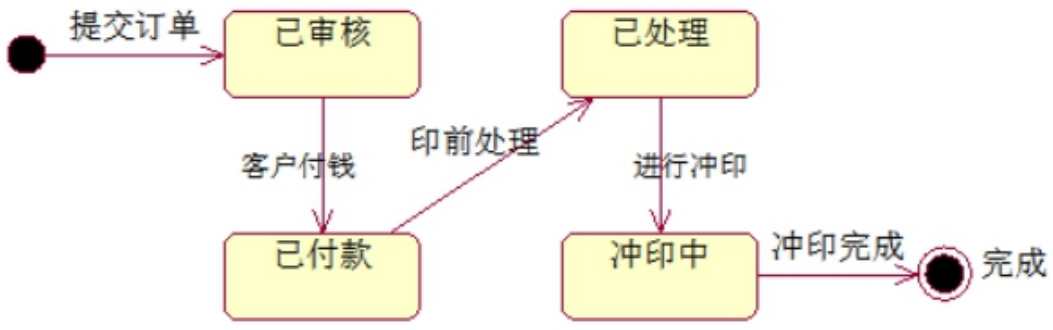
类图



顺序图



状态图



第六章:

XML概念: XML是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言，用于定义其他与特定领域有关的，语义的，结构化的标记语言的句法语言。

第八章:

1. *SOA定义: *

SOA是一种应用程序体系结构，在这种体系结构中，所有功能都定义为独立的服务，这些服务带有定义明确的可调用接口，能够以定义好的顺序调用这些服务来形成业务流程。

2. *SOA特征: *

松散耦合、粗粒度服务、标准化接口

3. *SOA设计原则: *

明确定义的接口；自包含和模块化；粗粒度；松耦合；互操作性、兼容和策略声明

4. *SOA关键技术: *

服务栈	主要技术
发现服务层	UDDI、DISCO
描述服务层	WSDL、XML Schema
消息格式层	

SOAP、REST

编码格式层

XML

传输协议层

HTTP、TCP/IP、SMTP等

每一层的作用:

发现服务层：用来帮助客户端应用程序解析远程服务的位置，通过UDDI来实现。

描述服务层：为客户端应用程序提供正确的与远程服务交互的描述信息，通过WSDL来实现

消息格式层：用来保证客户端应用程序和服务器端在格式设置上保持一致，通过SOAP实现

编码层：主要为客户端和服务器之间提供一个标准的，独立于平台的数据交换编码格式。通过XML实现。

传输层：为客户端和服务器之间提供两者交互的网络通信协议，通过HTTP和SMTP实现

5.SOAP应用: HTTP、RPC

6.SOAP 包含几部分? （1）SOAP封装。定义一个整体框架，用来表示消息中包含什么内容，谁来处理这些内容，以及这些内容是可选的或是必需的 （2）****SOAP编码规则****。定义了一种序列化的机制，用于交换系统所定义的数据类型的实例 （3）****SOAP RPC表示。****定义一个用来表示远程过程调用和应答的协议 （****4）SOAP绑定****。定义一个使用底层传输协议来完成在节点间交换SOAP信封的约定

第十二章

1. 设计模式: （下边两种表述都可以）

设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

描述了一个出现在特定设计语境中的特殊的再现设计问题，并为它的解决方案提供了一个经过充分验证的通用图示。

2. 设计模式的组成:

模式名称、问题、解决方案、后果 。

3. 设计模式的分类:

创建型模式：工厂方法模式，抽象工厂模式，原型模式，单例模式，构建器模式；

结构型模式：适配器模式、桥接模式、组合模式、装饰模式、外观模式，享元模式，代理模式；

行为型模式：职责链模式，命令模式，解释器模式，迭代器模式，中介者模式，备忘录模式，观察者模式，状态模式，策略模式，模板方法模式，访问者模式

4. MVC与SpringMVC:

（1）MVC是什么?

MVC是模型(model)- 视图(view)- 控制器(controller)的缩写，是一种软件设计典范。它是用一种业务逻辑、数据与界面显示分离的方法来组织代码，将众多的业务逻辑聚集到一个部件里面，在需要改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑，达到减少编码的时间。

（2）MVC构成：

Model：封装领域数据及逻辑

View：查询领域数据并展现给用户

Conctroller：截获用户请求并改变领域数据

（2）SpringMVC分层：

实体类也是POJO类，也就是MVC的数据模型，实体类仅有属性以及获取和设置属性的get和set方法，没有事务处理方法。

DAO负责数据访问

Service业务层主要用来处理业务逻辑

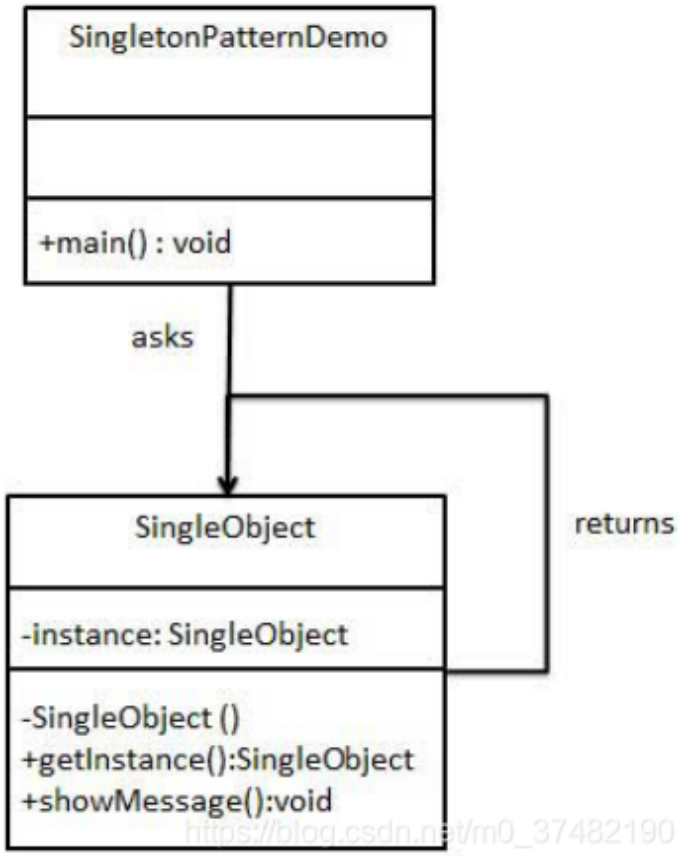
Controller层负责具体的业务模块流程的控制，在此层里面要调用Serice层的接口来控制业务流程。

（4）IOC“控制反转”：借助于“第三方”实现具有依赖关系的对象之间的解耦。一种将控制权转移的设计模式，由传统的程序控制转移到容器控制。

（5）DI依赖注入：就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。常用的有两种方式：构造方法注入和setter方法注入。

5. 单例模式:

保证一个类只有一个实例，并提供一个访问它的全局访问点



SingleObject 类提供了一个静态方法，供外界获取它的静态实例。**SingletonPatternDemo**，我们的演示类使用 **SingleObject** 类来获取**SingleObject** 对象。

public class SingleObject { //创建 SingleObject 的一个对象 private static SingleObject instance = new SingleObject(); //让构造函数为 private，这样该类就不会被实例化 private SingleObject(){} //获取唯一可用的对象 public static SingleObject getInstance(){ return instance; } public void showMessage(){ System.out.println(“Hello World!”); } }

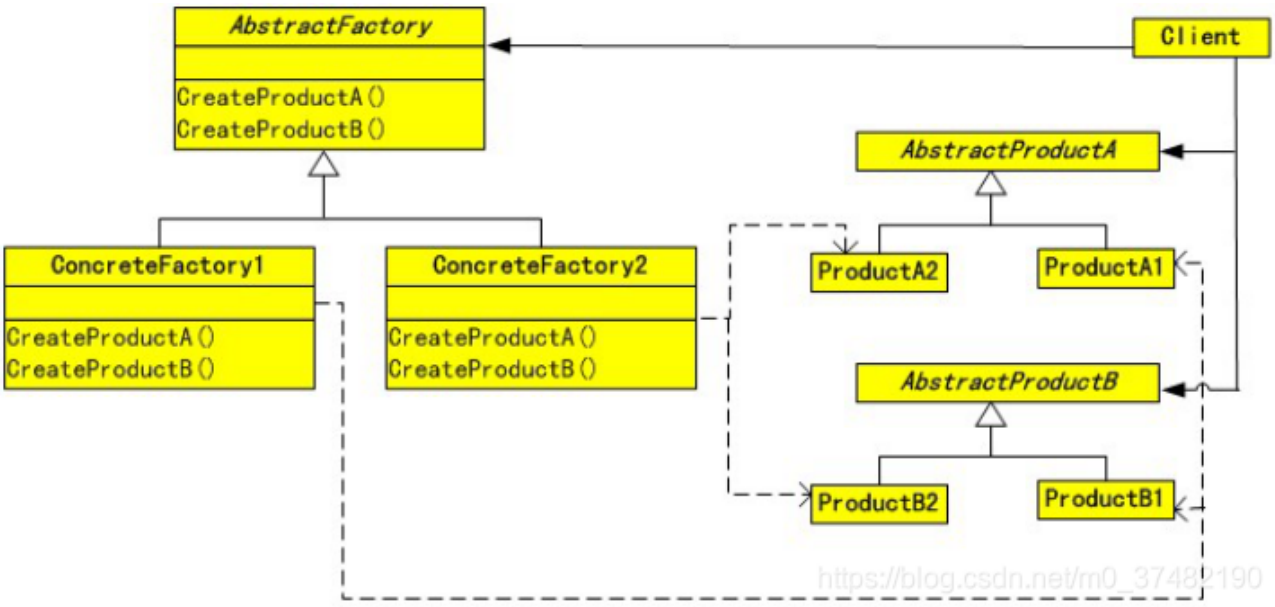
public class SingletonPatternDemo { public static void main(String[] args) { //获取唯一可用的对象 SingleObject object = SingleObject.getInstance(); //显示消息 object.showMessage(); } }

```
public class SingleObject {
//创建 SingleObject 的一个对象
    private static SingleObject instance = new SingleObject();
//让构造函数为 private，这样该类就不会被实例化
    private SingleObject(){}
//获取唯一可用的对象
    public static SingleObject getInstance(){
        return instance;
    }
    public void showMessage(){
    System.out.println("Hello World!");
    }
}
```

```
public class SingletonPatternDemo {
    public static void main(String[] args) {
//获取唯一可用的对象
    SingleObject object = SingleObject.getInstance();
//显示消息
    object.showMessage();
    }
}
```

6. 抽象工厂模式：

提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类



AbstractFactory：抽象工厂，声明抽象产品的方法

ConcreteFactory：具体工厂，执行生成抽象产品的方法，生成一个具体的产品

AbstractProduct：抽象产品，为一种产品声明接口

Product：具体产品，定义具体工厂生成的具体产品的对象，实现产品接口

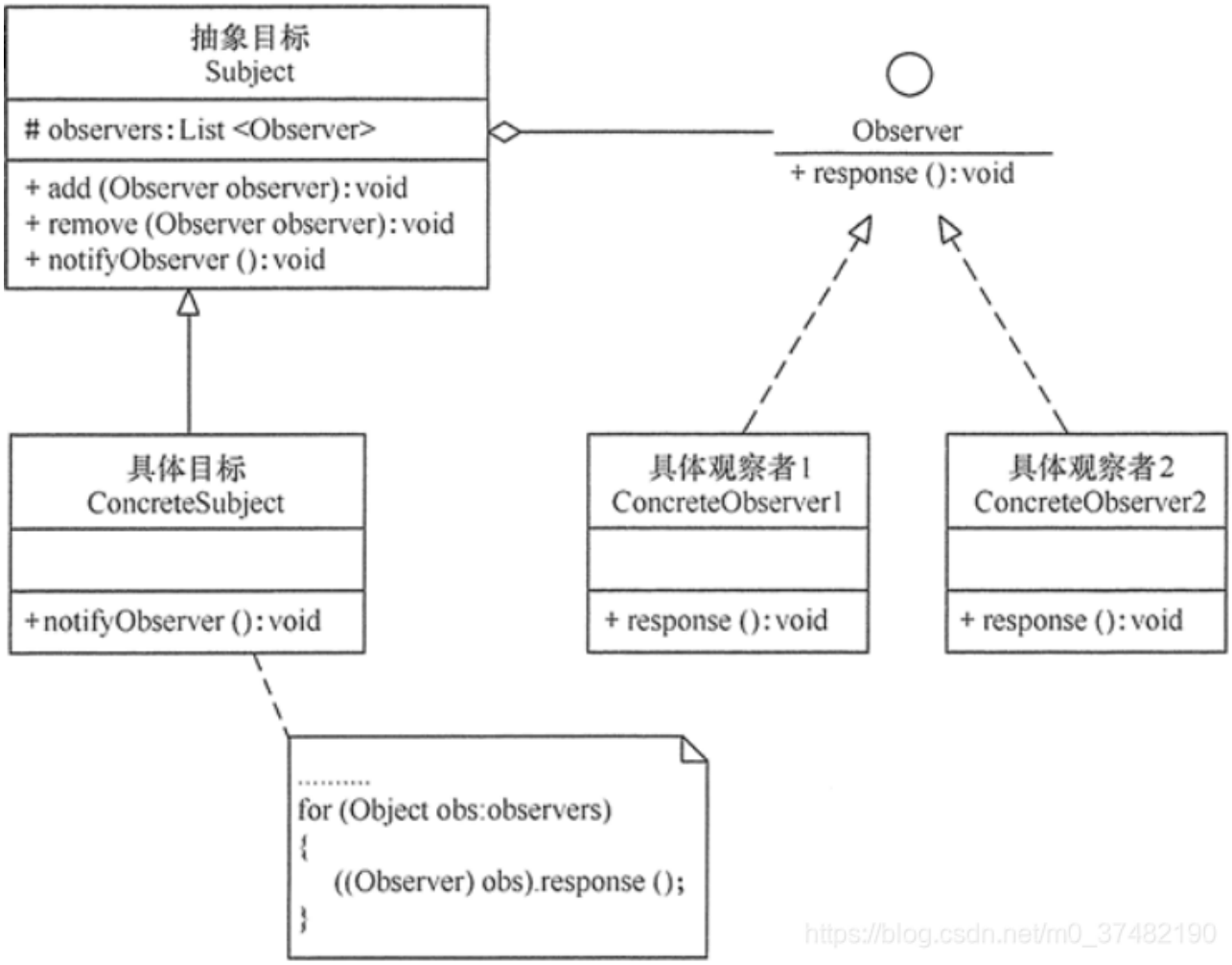
Client：客户，应用程序使用抽象产品和抽象工厂生成对象

```
/* 抽象工厂的抽象类，Java定义为接口*/
public interface AbstractFactory {
    //创建产品的方法
    public AbstractProductA  CreateProductA();
    public AbstractProductB  CreateProductB();
}
/*抽象产品接口*/
public interface AbstractProductA {
}
public interface AbstractProductB {
}
```

```
/*实际的产品*/
public class ProductA1 implements AbstractProductA {
    //实际的产品A1
}
public class ProductA2 implements AbstractProductA {
    //实际的产品A2
}
public class ProductB1 implements AbstractProductB {
    //实际的产品B1
}
public class ProductB2 implements AbstractProductB {
    //实际的产品B2
}
/*在以下程序中，CFactory是ConcreteFactory的简写*/
/*在第1个车间中，创建实际产品A1*/
public class CFactory1 implements AbstractFactory {
    public AbstractProductA  CreateProductA() {
        return new ProductA1();
    }
}
/*在第1个车间中，创建实际产品B1*/
public class CFactory1 implements AbstractFactory {
    public AbstractProductB  CreateProductB() {
        return new ProductB1();
    }
}

/*在第2个车间中，创建实际产品A2*/
public class CFactory2 implements AbstractFactory {
    public AbstractProductA  CreateProductA() {
        return new ProductA2();
    }
}
/*在第2个车间中，创建实际产品B2*/
public class CFactory2 implements AbstractFactory {
    public AbstractProductB  CreateProductB() {
        return new ProductB2();
    }
}
```

7. 观察者模式：

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新



抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

具体主题（Concrete Subject）角色：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

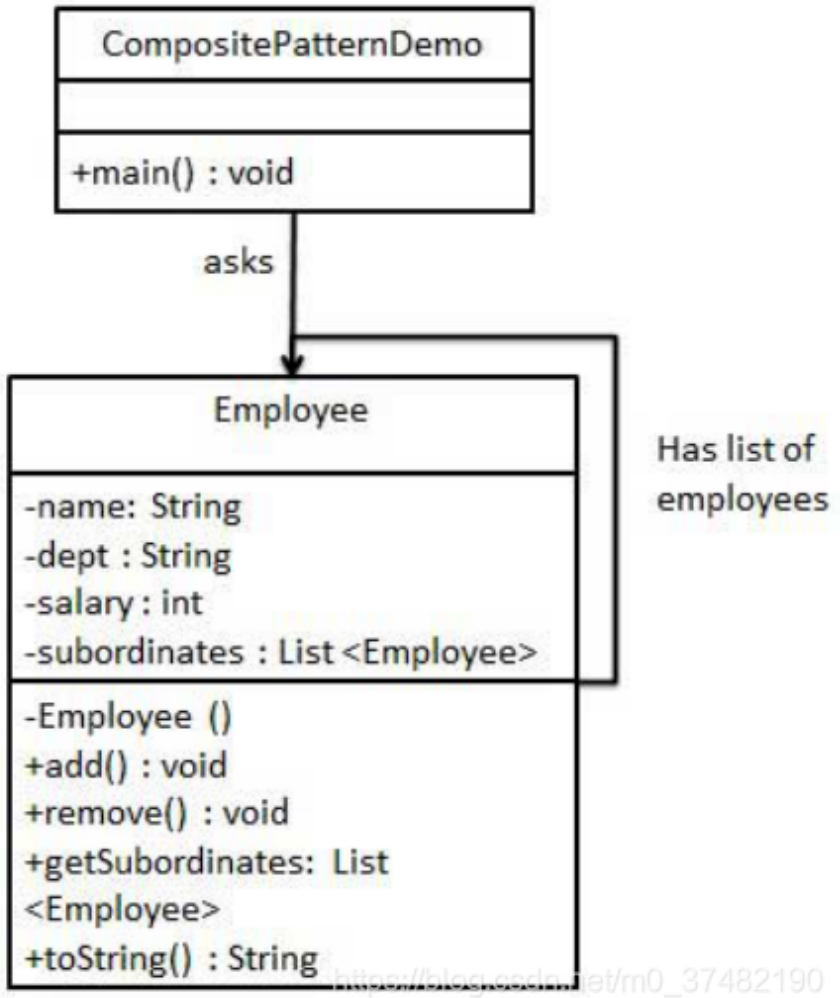
```
import java.util.*;
public class ObserverPattern
{
    public static void main(String[] args)
    {
        Subject subject=new ConcreteSubject();
        Observer obs1=new ConcreteObserver1();
        Observer obs2=new ConcreteObserver2();
        subject.add(obs1);
        subject.add(obs2);
        subject.notifyObserver();
```

```
    }
}
//抽象目标
abstract class Subject
{
    protected List<Observer> observers=new ArrayList<Observer>();
    //增加观察者方法
    public void add(Observer observer)
    {
        observers.add(observer);
    }
    //删除观察者方法
    public void remove(Observer observer)
    {
        observers.remove(observer);
    }
    public abstract void notifyObserver(); //通知观察者方法
}
//具体目标
class ConcreteSubject extends Subject
{
    public void notifyObserver()
    {
        System.out.println("具体目标发生改变...");
        System.out.println("--------------");

        for(Object obs:observers)
        {
            ((Observer)obs).response();
        }

    }
}
//抽象观察者
interface Observer
{
    void response(); //反应
}
//具体观察者1
class ConcreteObserver1 implements Observer
{
    public void response()
    {
        System.out.println("具体观察者1作出反应! ");
    }
}
//具体观察者1
class ConcreteObserver2 implements Observer
```

```
{
    public void response()
    {
        System.out.println("具体观察者2作出反应！");
    }
}
```

8. 组合模式：

将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性



我们有一个类 **Employee**，该类被当作组合模型类。**CompositePatternDemo**，我们的演示类使用 **Employee** 类来添加部门层次结构，并打印所有员工。

```
import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;
```

```
   //构造函数
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : "+ name
      +", dept : "+ dept + ", salary :"
      + salary+" ]");
   }
}

import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   //构造函数
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }
```

```
    public void remove(Employee e) {
       subordinates.remove(e);
    }

    public List<Employee> getSubordinates(){
      return subordinates;
    }

    public String toString(){
       return ("Employee :[ Name : "+ name
       +", dept : "+ dept + ", salary :"
       + salary+" ]");
    }
}
```

9. 中间件概念:

中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于操作系统之上，管理计算资源和网络通信，实现应用之间的互操作。

10. 中间件功能:

(1) 负责客户机与服务器之间的连接和通信，以及客户机与应用层之间的高效率通信机制。

(2) 提供应用层不同服务之间的互操作机制，以及应用层与数据库之间的连接和控制机制。

(3) 提供一个多层体系结构的应用开发和运行的平台，以及一个应用开发框架，支持模块化的应用开发。

(4) 屏蔽硬件、操作系统、网络和数据库的差异。

(5) 提供应用的负载均衡和高可用性、安全机制与管理功能，以及交易管理机制，保证交易的一致性。

(6) 提供一组通用的服务去执行不同的功能，避免重复的工作和使应用之间可以协作。

11. 消息中间件定义和使用场景:

（1）定义：消息中间件属于分布式系统中一个子系统，关注于数据的发送和接收，利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。

（2）使用场景:

四个典型场景：典型的异步处理、应用解耦、流量削锋、消息通讯四个场景。

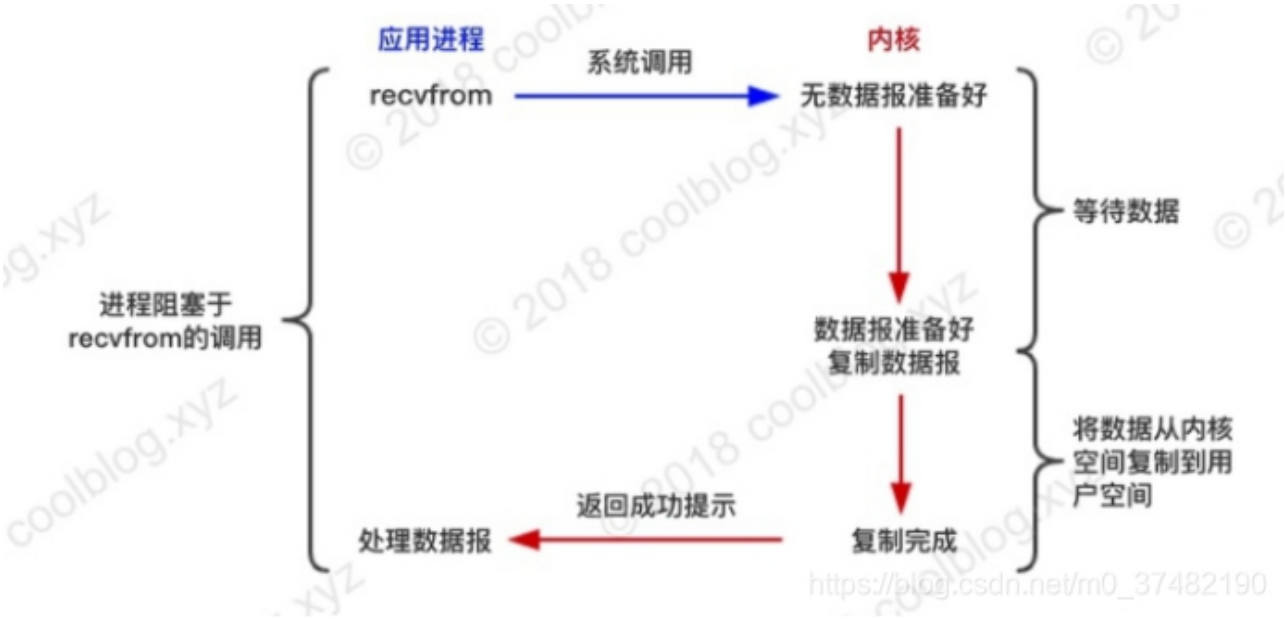
举例：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。（流量削锋）

11.常用消息中间件：

ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ。

12.I/O模型：

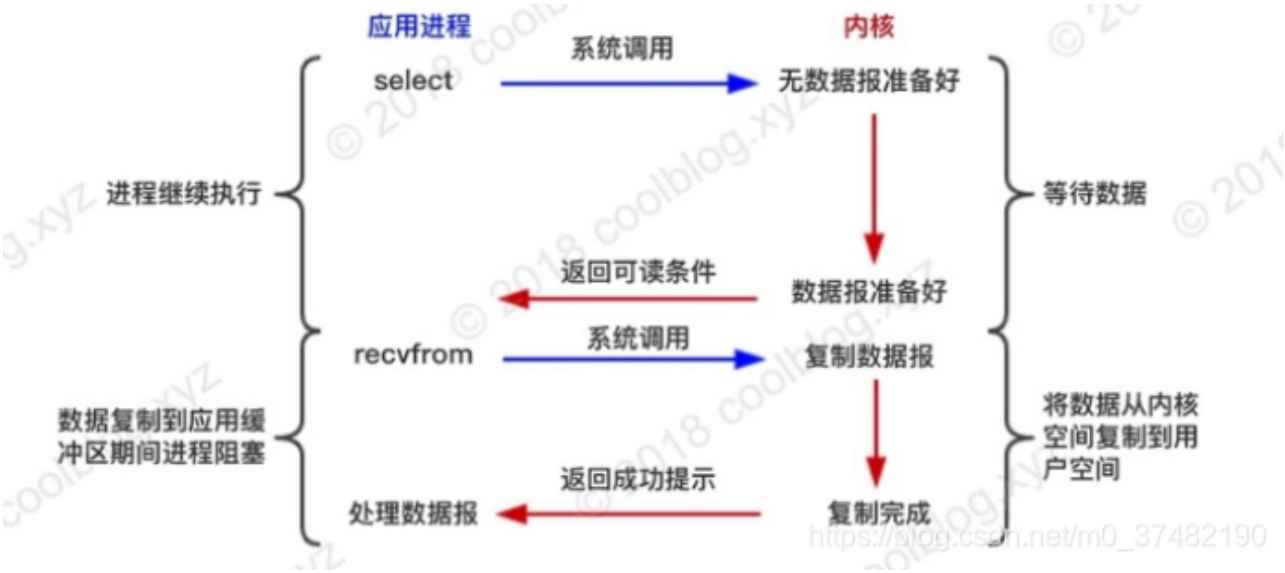
（1）阻塞I/O模型：进程或线程等待某个条件，如果条件不满足，则一直等下去。条件满足，则进行下一步操作。



（2）非阻塞I/O模型：应用进程与内核交互，目的未达到时，不再一味的等着，而是直接返回。然后通过轮询的方式，不停的去问内核数据准备好没。

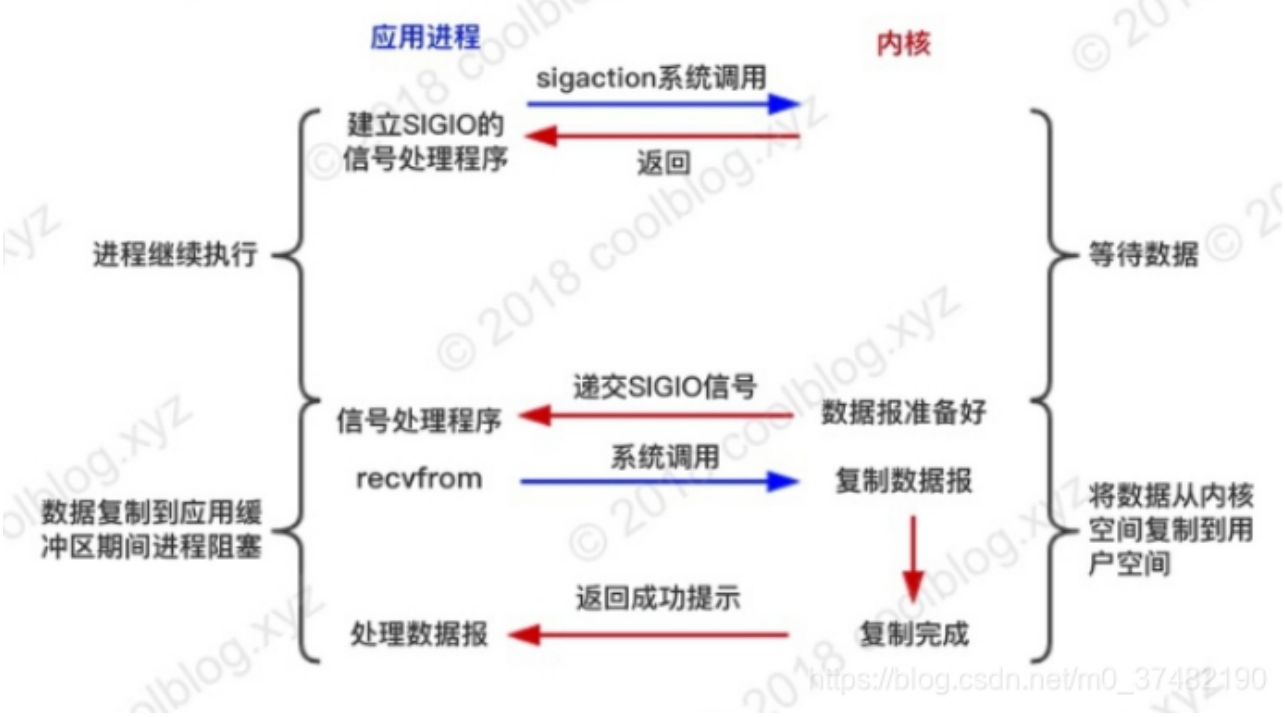
（3）I/O 复用模型

一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。



（4）信号驱动式 I/O 模型

首先开启Socket信号驱动I/O功能，并通过系统调用sigaction执行一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据准备就绪时，就为进程生成一个SIGIO信号，通过信号会掉通知应用程序调用recvfrom来读取数据，并通知主循环函数来处理数据。



（5）异步 I/O 模型

告知内核启动某个操作，并让内核在整个操作完成后（包括将数据从内核复制到用户自己的缓冲区）通知开发者。

第十一章

1. 软件质量属性:

性能、可靠性（容错，健壮性），可用性，安全性，可修改性（可维护性，可扩展性，结构重组，可移植性），功能性，可变性，继承性，互操作性。

2. 评估的主要方式:

基于调查问卷或检查表的评估方式，基于场景的评估方式，基于度量的评估方式

3. *ATAM评估步骤: *

描述ATAM方法；描述业务动机；描述架构；确定架构方法；生成质量属性效用树；分析架构方法；讨论场最和对场景分级.；分析架构方法；描述评估结果。

4. *SAAM评估步骤: *

形成场景、描述体系结构、对场景进行分类和确定优先级、对间接场景进行单个评估、评估场景的相互作用、形成总体评价

第十三章

1.软件产品线定义：产品线是一个产品集合，这些产品共享一个公共的、可管理的特征集，这个特征集能满足选定的市场或任务领域的特定需求。这些系统遵循一个预描述的方式，在公共的核心资源基础上开发的

检查表的评估方式，基于场景的评估方式，基于度量的评估方式

3. *ATAM评估步骤: *

描述ATAM方法；描述业务动机；描述架构；确定架构方法；生成质量属性效用树；分析架构方法；讨论场最和对场景分级.；分析架构方法；描述评估结果。

4. *$SAAM$评估步骤：*

形成场景、描述体系结构、对场景进行分类和确定优先级、对间接场景进行单个评估、评估场景的相互作用、形成总体评价

第十三章

1.软件产品线定义：产品线是一个产品集合，这些产品共享一个公共的、可管理的特征集，这个特征集能满足选定的市场或任务领域的特定需求。这些系统遵循一个预描述的方式，在公共的核心资源基础上开发的## 目标

0.1 原理