Table of Contents

[1.软件架构 3](#_Toc25814)

[1.1.架构风格（6个） 3](#_Toc22041)

[1.1.1. 数据流风格 3](#_Toc10945)

[1.1.2. 调用/返回风格 4](#_Toc28257)

[1.1.3. 独立构建风格 5](#_Toc28131)

[1.1.4. 虚拟机风格 6](#_Toc8873)

[1.1.5. 仓库风格 6](#_Toc29480)

[1.2.典型层次结构 7](#_Toc23335)

[1.2.1.MVC 7](#_Toc25826)

[1.2.2.MVP 8](#_Toc22524)

[1.2. 面向服务的架构 8](#_Toc3450)

[1.2.0.什么是SOA 8](#_Toc16782)

[1.2.1. SOA关键技术 9](#_Toc25817)

[1.2.1.5.BPEL 11](#_Toc17618)

[1.2.2. SOA实现 11](#_Toc28169)

[1.2.3.微服务 13](#_Toc25325)

[1.2.3. 对比 13](#_Toc3797)

[2.1.软件质量属性 14](#_Toc5841)

[2.2.3个点 15](#_Toc6516)

[2. 数据库相关 16](#_Toc18619)

[2.1. NOSQL 16](#_Toc16513)

[2.2. Memcached 16](#_Toc1737)

[Redis 和 Memecached 区别 17](#_Toc12245)

[2.3. 数据库读写分离 17](#_Toc29429)

[2.3.1.数据库+缓存 17](#_Toc6391)

[2.3.2.主从复制 18](#_Toc6716)

[2.4.数据库拆分 18](#_Toc14396)

[2.4.1.垂直拆分 18](#_Toc17856)

[2.4.2.水平拆分 19](#_Toc16342)

[2.5反规范化 20](#_Toc7710)

[3. 系统安全 21](#_Toc16477)

[3.1.非对称加密 21](#_Toc31312)

[3.2.对称加密 21](#_Toc8853)

[3.3.数据库加解密 22](#_Toc16089)

[4. 设计模式 22](#_Toc9349)

[4.1.设计模式类型 22](#_Toc5896)

[4.2.适配器 23](#_Toc7191)

[4.3.策略 23](#_Toc24980)

[5. 系统需求分析 24](#_Toc5075)

[5.1. DFD 24](#_Toc13027)

[5.2.UML 25](#_Toc14975)

[5.2.1.用例图-用例之间的关系 25](#_Toc8885)

[5.2.2.类之间的关系 26](#_Toc9380)

[6.系统的可靠性分析与设计 27](#_Toc2066)

[6.1.可靠性 27](#_Toc5957)

[6.2.N版本程序设计 27](#_Toc7588)

[6.3.动态冗余 28](#_Toc3407)

[6.4.恢复块方法 28](#_Toc6631)

[7.项目管理 29](#_Toc30090)

[7.1.项目计划 29](#_Toc25042)

[7.2.最短/最长工期 30](#_Toc21779)

[7.3.涉及到规划需要列表 30](#_Toc29169)

# 1.软件架构

## 1.1.架构风格（6个）

软件架构风格是指描述特定软件系统组织方式的惯用模式。组织方式描述了系统的组成构件和这些构件的组织方式，惯用模式则反映众多系统共有的结构和语义。

### 数据流风格

数据流风格的软件架构是一种最常见，结构最为简单的软件架构。这样的架构下，所有的数据按照流的形式在执行过程中前进，不存在结构的反复和重构。在流动过程中，数据经过序列间的数据处理组件进行处理，然后将处理结果向后传送，最后进行输出。

#### 1.1.1.1批处理序列

批处理风格的每一步处理都是独立的，并且每一步是顺序执行的。只有当前一步处理完， 后一步处理才能开始。数据传送在步与步之间作为一个整体。

#### 1.1.1.2.管道—过滤器

每个构件都有一组输入和输出，构件读输入的数据流经过内部处理,然后产生输出数据流,这个过程通常是通过对输入数据流的变换或计算来完成的。这里的构件称为过滤器,连接件就是数据传输的管道，将—个过滤器的输出传到另一个过滤器的输入。

### 调用/返回风格

调用返回风格顾名思义，就是指在系统中采用了调用与返回机制。利用调用-返回实际上是一种分治策略，其主要思想是将一个复杂的大系统分解为一些子系统，以便降低复杂度，并且增加可修改性。程序从其执行起点开始执行该构件的代码，程序执行结束，将控制返回给程序调用构件。

#### 1.1.2.1主程序/子程序

主程序/子程序风格是结构化开发时期的经典架构风格。这种风格一般采用单线程控制，把问题划分为若干处理步骤，构件即为主程序和子程序。子程序通常可合成为模块。过程调用作为交互机制，即充当连接件。调用关系具有层次性，其语义逻辑表现为子程序的正确性，取决于它调用的子程序的正确性。

#### 1.1.2.2.面向对象

抽象数据类型概念对软件系统有着重要作用。这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在对象中。这种风格的构件是对象。对象是一种被称作管理者的构件，因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

#### 1.1.2.3.层次结构

构件组织成一个层次结构，连接件通过决定层间如何交互的协议 来定义。每层为上一层提供服务，使用下一层的服务，只能见到与自己邻接的层。通过层次结构，可以将大的问题分解为若干个渐进的小问题逐步解决，可以隐藏问题的复杂度。修改某一层，最多影响其相邻的两层。

### 独立构建风格

独立构件风格主要强调系统中的每个构件都是相对独立的个体，它们之间不直接通信，以降低耦合度，提升灵活性。独立构件风格主要包括：进程通讯和事件系统子风格。

#### 1.1.3.1.进程通信

构件是独立的过程，连接件是消息传递，构件通常是命名过程，消息传递的方式可以是点对点、异步或同步方式，以及远程过程调用等。

#### 1.1.3.2.事件驱动系统/隐式调用

构件不直接调用一个过程，而是触发或广播一个或多个事件。构件中的过程在一个或多个事件中注册,当某个事件被触发时,系统自动调用这个事件注册的所有过程。一个事件的触发就导致了另一个模块中的过程调用。这种风格中的构件是匿名的过程,它们之间交互的连接件往往是以过程之间的隐式调用来实现的。主要优点是为软件复用提供了强大的支持,为构件的维护和演化带来了方便;其缺点是构件放弃对系统计算的控制。

### 虚拟机风格

虚拟机风格的基本思想是人为构建一个运行环境，在这个环境之上，可以解析与运行自定义的一些语言，这样来增加架构的灵活性，虚拟机风格主要包括解释器和规则为中心两种架构风格。

#### 1.1.4.1.解释器

解释器通常包括一个完成解释工作的解释引擎，一个包含将被解释的代码的存储区，一个记录解释引擎当前工作状态的数据结构，以及一个记录源代码被解释执行的进度的数据结构，具有解释器风格的软件中含有一个虚拟机，可以仿真硬件的执行过程和一些关键应用。其缺点是执行效率比较低。

#### 1.1.4.2.基于规则的系统

基于规则的系统包括规则集、规则解释器、规则/数据选择器和工作内存，一般用在人工智能领域和决策支持系统中。

### 仓库风格

在仓库（repository）风格中，有两种不同的构件：中央数据结构说明当前状态，独立构件在中央数据存储上执行，仓库与外构件间的相互作用在系统中会有大的变化。

#### 1.1.5.1.数据库系统

构件主要有两大类，一类是中央共享数据源，保存当前系统的数据状态；另一类是多个独立处理单元，处理单元对数据元素进行操作。

#### 1.1.5.2.黑板系统

包括知识源、黑板和控制三部分。知识源包括若干独立计算的不同单元,提供解决向题的知识。知识源响应黑板的变化,也只修改黑板；黑板是一个全局数据库,包含问题域解空间的全部状态,是知识源相互作用的唯一媒介;知识源响应是通过黑板状态的变化来控制。

#### 1.1.5.3.超文本系统

构件以网状链接方式相互连接，用户可以在构件之间进行按照人 类的联想思维方式任意跳转到相关构件，超文本是一种非线性的网状信息组织方法，它以结点为基本单位，链作为结点之间的联想式关联。

## 1.2.典型层次结构

### 1.2.1.MVC

Model 是对应用状态和业务功能的封装，我们可以将它理解为同时包含数据和行为的领域模型。Model 接受 Controller 的请求并完成相应的业务处理，在状态改变的时候向 View 发出相应的通知。

View 实现可视化界面的呈现并捕捉最终用户的交互操作（例如鼠标和键盘的操作）。

View 捕获到用户交互操作后会直接转发给 Controller，后者完成相应的 UI 逻辑。 如果需要涉及业务功能的调用，Controller 会直接调用 Model。在完成 UI 处理后，Controller会根据需要控制原 View 或者创建新的 View 对用户交互操作予以响应。

### 1.2.2.MVP

MVP 的全称为 Model-View-Presenter，Model 提供数据，View 负责显示，Controller/Presenter 负责逻辑的处理。MVP 是从经典的模式 MVC 演变而来。MVP 与 MVC 也有一些显著的区别，MVC 模式中元素之间“混乱”的交互主要体现在允许 View 和 Model 直接进行“交流”，这在 MVP 模式中是不允许的。在 MVP 中 View 并不直接使用Model，它们之间的通信是通过 Presenter来进行的，所有的交互都发生在 Presenter 内部，而在 MVC 中 View 会直接从 Model 中读取数据而不是通过 Controller。

## 面向服务的架构

### 1.2.0.什么是SOA

面向服务的架构（Service-Oriented Architecture,SOA）是一种组件模型，把应用程序中的不同功能单元（即服务）通过这些服务之间定义良好的接口和契约联系起来，使得这些系统中的服务能够以-种统一和通用的方式进行交互。从应用角度看，SOA 是一种应用框架，它关注企业日常的业务应用，将其划分为单独的业务功能和流程，并抽象为服务，用户和系统开发人员可以构建、部署和整合这些服务，无需依赖特定的应用程序及应用平台，从而提高企业业务流程的灵活性。SOA有助于实现更多的信息资产重用、更轻松地管理和更快地应用开发与部署。

### SOA关键技术

#### 1.2.1.1.UDDI

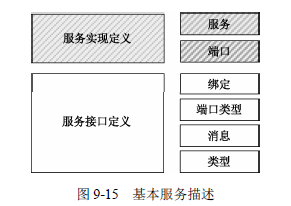
UDDI（Universal DescriptionDiscovery and Integration，统一描述、发现和集成）提供了一种服务发布、查找和定位的方法，是服务的信息注册规范，以便被需要该服务的用户发现和使用它。UDDI 规范描述了服务的概念，同时也定义了一种编程接口。通过 UDDI 提供的标准接口，企业可以发布自己的服务供其他企业查询和调用，也可以查询特定服务的描述信息，并动态绑定到该服务上。

#### 1.2.1.2.WSDL

WSDL（Web ServiceDescription Language，Web 服务描述语言）用于描述 Web 服务的接口和操作功能。WSDL 描述的重点是服务，它包含服务实现定义和服务接口定义。

服务接口定义就是一种抽象的、可重用的定义，行业标准组织可以使用这种抽象的定义来规定一些标准的服务类型，服务实现者可以根据这些标准定义来实现具体的服务。

服务实现定义描述了给定服务提供者如何实现特定的服务接口。服务实现定义中包含服务和端口描述。一个服务往往会包含多个服务访问入口，而每个访问入口都会使用一个端口元素来描述，端口描述的是一个服务访问入口的部署细节，例如，通过哪个地址来访问，应当使用怎样的消息调用模式来访问等。



#### 1.2.1.3.SOAP

SOAP（Simple ObjectAccess Protocol，简单对象访问协议）为建立 Web 服务和服务请求之间的通信提供支持。SOAP 用 XML 来格式化消息，用 HTTP 来承载消息。

#### 1.2.1.4.REST

REST 从资源的角度来定义整个网络系统结构，分布在各处的资源由统一资源标识符（URI）确定，客户端应用程序通过 URI 获取资源的表现，并通过获得资源表现使得其状态发生改变。REST 提出了如下一些设计概念和准则：

（1）网络上的所有事物都被抽象为资源。

（2）每个资源对应一个唯一的资源标识。

（3）通过通用的连接件接口对资源进行操作。

（4）对资源的各种操作不会改变资源标识。

（5）所有的操作都是无状态的

### 1.2.1.5.BPEL

BPEL（Business Process Execution Language For Web Services）面向 Web 服务的业务流程执行语言。BPEL 可将多个 Web 服务组合到一个新的复合服务（称作业务流程）中。

### SOA实现

SOA 只是一种概念和思想，需要借助于具体的技术和方法来实现它。从本质上来看， SOA 是用本地计算模型来实现一个分布式的计算应用，也有人称这种方法为“本地化设计，分布式工作”模型。CORBA、DCOM 和 EJB 等都属于这种解决方式，也就是说，SOA 最终可以基于这些标准来实现。

#### WebService

在 Web Service（Web 服务）的解决方案中，一共有三种工作角色，其中服务提供者和服务请求者是必需的，服务注册中心是一个可选的角色。它们之间的交互和操作构成了 SOA 的一种实现架构。

（1）服务提供者。服务提供者是服务的所有者，该角色负责定义并实现服务，使用 WSDL 对服务进行详细、准确、规范地描述，并将该描述发布到服务注册中心，供服务请求者查找并绑定使用。

（2）服务请求者。服务请求者是服务的使用者，虽然服务面向的是程序，但程序的最终使用者仍然是用户。从架构的角度看，服务请求者是查找、绑定并调用服务，或与服务进行交互的应用程序。服务请求者角色可以由浏览器来担当，由人或程序（例如，另外一个服

务）来控制。

（3）服务注册中心。服务注册中心是连接服务提供者和服务请求者的纽带，服务提供者在此发布他们的服务描述，而服务请求者在服务注册中心查找他们需要的服务。不过，在某些情况下，服务注册中心是整个模型中的可选角色。例如，如果使用静态绑定的服务，服 务提供者则可以把描述直接发送给服务请求者。

#### 企业服务总线

ESB 是传统中间件技术与 XML、Web 服务等技术结合的产物，主要支持异构系统集成。在一个复杂的企业计算环境中，如果服务提供者和服务请求者之间采用直接的端到端的交互，那么随着企业信息系统的增加和复杂度的提高，系统之间的关联会逐渐变得非常复杂，这将带来昂贵的系统维护费用。ESB 消除了服务请求者与服务提供者之间的直接连接，使得服务请求者与服务提供者之间进一步解耦。

ESB的主要功能：

（1）服务位置透明性；

（2）传输协议转换；

（3）消息格式转换；

（4）消息路由；

（5）消息增强；

（6）安全性；

（7）监控与管理。

### 1.2.3.微服务

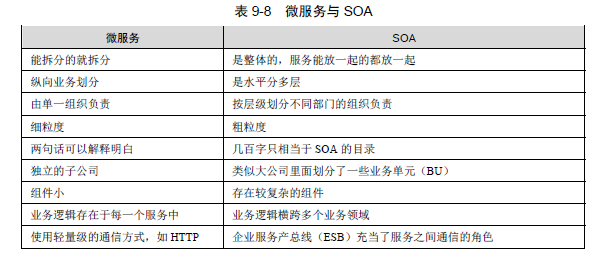
微服务顾名思义，就是很小的服务，所以它属于面向服务架构的一种。通俗一点来说，微服务类似于古代著名的发明，活字印刷术，每个服务都是一个组件，通过编排组合的方式来使用，从而真正做到了独立、解耦、组件化、易维护、可复用、可替换、高可用、最终达

到提高交付质量、缩短交付周期的效果。

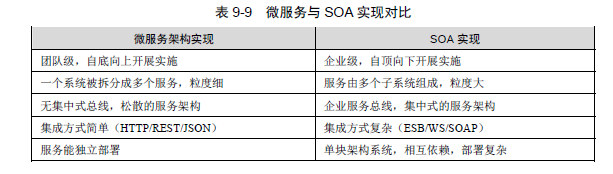
从专业的角度来看，微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

### 对比

微服务可以讲是 SOA 的一种，但仔细分析与推敲，我们又能发现他们的一些差异。这种差异表现在多个方面。



实现差异



## 2.1.软件质量属性

**性能**（performance）是指系统的响应能力，即要经过多长时间才能对某个事件做出响应，或者在某段时间内系统所能处理的事件的个数。代表参数：响应时间、吞吐量 设计策略：优先级队列、资源调度。

**可靠性**（reliability）是软件系统在应用或系统错误面前，在意外或错误使用的情况下维持系统的功能特性。基本能力代表参数：MTTF、MTBF 设计策略：冗余、心跳线。

**可用性**（availability）是系统能够正常运行的时间比例。经常用两次故障直接的时间长度或在出现故障时系统能够恢复正常的速度来表示。 代表参数：故障间隔时间 设计策略：冗余、心跳线。

**安全性**（security）是指系统在向合法用户提供服务的同时能够阻止非授权用户使用的企图或拒绝服务的能力。安全性又可划分为机密性、完整性、不可否认性及可控性等特性。设计策略：追踪审计、信息隐藏

**可修改性**（modifiability）是指能够快速地以较高的性能价格比对系统进行变更的能力。通常以某些具体的变更为基准，通过考察这些变更的代价衡量可修改性。

**功能性**（functionality）是系统所能完成所期望的工作的能力。一项任务的完成需要系统中许多 或大多数构件的相互协作。

**可变性**（changeability）是指系统结构经扩充或变更而成为新体系结构的能力。这种新体系结构 应该符合预先定义的规则，在某些具体方面不同于原有的体系结构。当要将某个体系结构作为一系列产品（例如，软件产品线）的基础时，可变性是很重要的。

**互操作性**（interoperation）作为系统组成部分的软件不是独立存在的，经常与其他系统的自身环境相互作用。为了支持互操作性（interoperation），软件体系结构必须为外部可视的功能特性和数据结构提供精心设计的软件入口。程序和用其他编程语言编写的软件系统的交互作用就是互操作性的问题，这种互操作性也影响应用的软件体系结构。

## 2.2.3个点

风险点

系统架构风险是指架构设计中潜在的、存在问题的架构决策所带来的隐患。

敏感点

敏感点是指为了实现某种特定的质量属性，一个或多个构件所具有的特性。

权衡点

权衡点是影响多个质量属性的特性，是多个质量属性的敏感点。

# 数据库相关

## NOSQL

|  |  |
| --- | --- |
| 优势 | |
| 性能 | 支持高并发数据访问，性能较高。 |
| 性能 | 数据存储结构松散，能够灵活支持多种类型的数据格式。 |
| 性能 | 数据库能够支持海量数据的存储，且易于横向扩展。 |
| 可用性 | 数据库基于分布式数据存储，不存在单点故障和性能瓶颈，可用性高。 |

|  |  |
| --- | --- |
| 劣势 | |
| 其他 | 数据库的现有产品不够成熟，大多数产品处于初创期 |
| 其他 | 并未形成一定的标准，产品种类繁多，缺乏官方支持。 |
| 其他 | 数据库不提供对 SQL 的支持，学习和应用迁移成本较高。 |
| 其他 | 数据库支持的特性不够丰富，现有产品提供的功能比较有限。 |

## Memcached

Memcached是一套分布式的高速缓存系统，与Redis相似。一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。为了提高性能，Memcached中保存的数据都存储在Memcached内置的内存存储空间中。由于数据仅存在于内存中，因此重启Memcached、重启操作系统会导致全部数据消失。另外，内容容量达到指定值之后，就基于LRU(Least Recently Used)算法自动删除不使用的缓存。Memcached本身是为缓存而设计的服务器，因此并没有过多考虑数据的永久性问题。

### Redis 和 Memecached 区别

|  |  |  |
| --- | --- | --- |
|  | Memecached | Redis |
| 数据类型 | key/value | key/value，list,map,set,hash,sorted |
| 持久性 | 不支持 | 支持 |
| 分布式存储 | 不支持 | 多种方式，主从、Sentinel、Cluster 等 |
| 多线程 | 支持 | 不支持 |
| 内存管理 | 有 | 无 |
| 事务支持 | 不支持 | 有限支持 |

## 数据库读写分离

### 2.3.1.数据库+缓存

读写分离架构应用非常广泛，很多网站采用数据库+缓存的方式来实现。通过缓存层来承载大量的读访问，如广泛采用的 Memcached，其自身往往不具备持久层存储的功能，通常和数据库一起组成分布式的数据架构，由数据库负责数据持久化存储和写入功能，缓存负责承载大量的并发访问，从而提高了系统的数据处理效率。要避免数据访问的单点故障，通常采用主数据库热备份的方式来实现。所以，要实现题目要求的分布式数据架构，需要多个局部数据库系统、多个热备份数据库系统和多个数据缓存组成。

读写分离结构中，应用读取数据时访问缓存，如果没有命中所需数据，则从主数据库中读取数据并写入缓存；对于新增、修改和删除操作，需要采用延迟加载的策略，新增时只修改主数据库，修改和删除时除了修改主数据库中的内容，还需要将缓存中的数据标记为失效。

### 2.3.2.主从复制

主从复制机制使得同样的数据，存在多个副本，这样让用户查询数据时，可以选择该数据最近的副本进行访问，提高效率，降低资源使用时的冲突，避免单点故障。主从之间通过交换 REDO 日志的形式来同步数据。

## 2.4.数据库拆分

当我们使用读写分离、缓存后，数据库的压力还是很大的时候，这就需要使用到数据库拆分了。数据库拆分简单来说，就是指通过某种特定的条件，按照某个维度，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面以达到分散单库（主机）负载的效果。切分模式： 垂直（纵向）拆分、水平拆分。

### 2.4.1.垂直拆分

一个数据库由很多表的构成，每个表对应着不同的业务，垂直切分是指按照业务将表进行分类，分布到不同的数据库上面，这样也就将数据或者说压力分担到不同的库上面。

### 2.4.2.水平拆分

垂直拆分后遇到单机瓶颈，可以使用水平拆分。相对于垂直拆分的区别是：垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。

相对于垂直拆分，水平拆分不是将表的数据做分类，而是按照某个字段的某种规则来分散到多个库之中，每个表中包含一部分数据。简单来说，我们可以将数据的水平切分理解为是按照数据行的切分，就是将表中的某些行切分到一个数据库，而另外的某些行又切分到其他的数据库中，主要有分表，分库两种模式。

|  |  |
| --- | --- |
| 优点 | |
| 性能 | 不存在单库大数据，高并发的性能瓶颈 |
| 可修改性 | 对应用透明，应用端改造较少 |
| 可用性 | 提高了系统的稳定性跟负载能力 |
| 其他 | 按照合理拆分规则拆分，join操作基本避免跨库 |

|  |  |
| --- | --- |
| 缺点 | |
| 可操作性 | 拆分规则难以抽象 |
| 一致性 | 分片事务一致性难以解决 |
| 可维护性 | 数据多次扩展难度跟维护量极大 |
| 性能 | 跨库join性能较差 |

## 2.5反规范化

规范化设计后，数据库设计者希望牺牲部分规范化来提高性能，这种从规范化设计的回退方法称为反规范化技术。

采用反规范化技术的益处：降低连接操作的需求、降低外码和索引的数目，还可能减少表的数目，能够提高查询效率。可能带来的问题：数据的重复存储，浪费了磁盘空间；可能出现数据的完整性问题，为了保障数据的一致性，增加了数据维护的复杂性，会降低修改速度。

|  |  |
| --- | --- |
| 缺点 | |
| 增加冗余列 | 在多个表中保留相同的列，通过增加数据冗余减少或避免查询时的连接操作。 |
| 增加派生列 | 在表中增加可以由本表或其它表中数据计算生成的列，减少查询时的连接操作并避免计算或使用集合函数。 |
| 重新组表 | 如果许多用户需要查看两个表连接出来的结果数据，则把这两个表重新组成一个表 来减少连接而提高性能。 |
| 水平分割表 | 根据一列或多列数据的值，把数据放到多个独立的表中，主要用于表数据规模很大、表中数据相对独立或数据需要存放到多个介质上时使用。 |
| 垂直分割表 | 对表进行分割，将主键与部分列放到一个表中，主键与其它列放到另一个表中， 在查询时减少 I/O 次数。 |

# 系统安全

## 3.1.非对称加密

非对称加密技术是指加密与解密使用不同的密钥进行。这种密钥是成对产生的，一对密钥中公钥加密，私钥解密；私钥加密，则公钥解密。使用该技术进行保密传输时，使用接收方的公钥加密，然后进行传输，当接收方收到信息后，使用自己的私钥解密。而当使用该技术进行完整性保障时，也需要用到信息摘要技术（防篡改），先将需要发送的信息产生摘要，摘要使用发送方的私钥加密（防抵赖），接收方收到以后，如果能以发送方的公钥解密出相应摘要，则说明信息完整。

## 3.2.对称加密

对称加密技术，顾名思义，就是指的加密与解密密钥是一样的。这种机制之下，发送者与接收者拥有同样的密钥，需要保密传输数据时，由发送方加密信息内容，将其发给接收方，接收方用相同的密钥解密即可。在利用到对称加密技术做完整性保障时，可以将需要发送的信息产生摘要，将摘要使用对称密钥加密，然后将信息与加密后的摘要一起发送给接收方，当接收方收到消息后，将摘要的加密包解开，并对收到的信息重新产生摘要，将两个摘要进行对比，便可知信息在传递过程中完整性是否遭到破坏。

## 3.3.数据库加解密

目前数据库管理系统提供的基本数据加密支持主要有以下两种：

（1）加解密 API：数据库管理系统提供可在 SQL 语句中调用的加解密 API，应用可以利用这些 API构建自己的基础架构，对数据进行加密保护。

（2）透明加密：安全管理员为数据库敏感字段选择加密方式及密钥强度，应用访问受保护数据时只需使用口令打开或关闭密钥表，对数据的加密和解密由数据库管理系统自动完成。

加解密 API 方式的灵活性强，但构建和管理复杂；而透明加密方式管理简单，应用程序负担轻，但灵活性较差。用户要求尽可能减少安全管理与应用程序的负担，因此应选择透明加密方式。

# 设计模式

## 4.1.设计模式类型

创建型模式主要用于创建对象，为设计类实例化新对象提供指南（抽构工原单）。结构型模式主要用于对类如何设计以形成更大的结构提供指南（适桥组装外享代）。行为型模式主要用类之间交互以及分配责任的方式提供指南。

## 4.2.适配器

在实现工具之间数据格式的灵活转换时，通常采用适配器设计模式。即应首先定义一个统一的数据转换接口类，然后针对不同的数据格式转换需求定义对应的实际转换类，实际转换类需要继承数据转换接口类，并实现接口转换类定义的接口。

## 4.3.策略

在具有公共接口的独立类中定义每个计算。可以利用该模式创建各种促销类，它们从同一个超类继承。每个类都有相同名称的标准接口方法，用于根据订单编号计算将要折扣的金额总数。计算每个促销的内部代码对促销类来说完全不同。

**适配器与策略的区别**

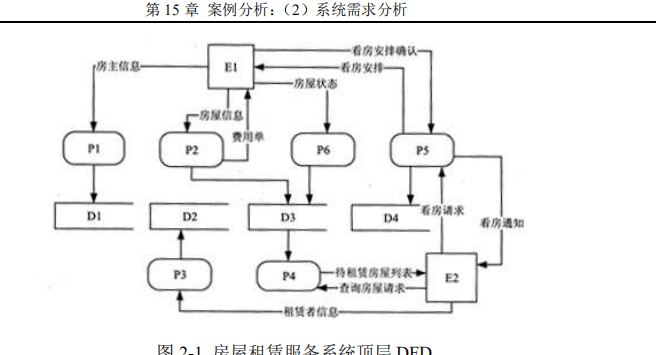
策略模式：方法的形参为接口对象，实参为接口的实现类。适配器模式：在适配器中定义适配者来辅助实现接口。

策略模式所有的策略都需要暴露出去，由客户端决定使用哪一个策略。而适配器模式是定义好接口的实现方式以及内部需要引用的类，客户端直接调用适配器的方法。

# 系统需求分析

## DFD

图例。数据流：箭头。外部实体：矩形。加工：圆角矩形。数据存储：右侧开放矩形。



**数据流**：数据流是数据在系统内传播的路径，因此由一组成分固定的数据组成。

**外部实体**：代表系统之外的实体，可以是人、物或其他软件系统。

**加工（处理）**：加工是对数据进行处理的单元，它接收一定的数据输入，对其进行处理，并产生输出。

**数据存储**：表示信息的静态存储，可以是文件或数据库的元素。

**其他：**加工要有输入流也要有输出。存储到实体需要加工。实体到存储也要加工。数据存储也可以有输出。

## 5.2.UML

### 5.2.1.用例图-用例之间的关系

**包含**。**速记:做A要先做B，那么A和B就是包含关系。**当可以从两个或两个以上的用例中提取公共行为时，应该使用包含的关系来表示它们。其中这个提取出来的公共用例成为抽象用例，而把原始用例成为基本用例或基础用例。其中“<<include>>”是包含关系的构造型，箭头指向抽象用例。例如，在机房收费系统中“注册学生信息”和“充值”两个用例都需要操作员或者管理员登陆，为此，可以定义一个抽象用例“用户登陆”。用例“注册学生信息”和“充值”与用例“用户登陆”之间的关系就是包含关系。

**扩展**。**速记:做A不一定做B，或者做B不一定做A，那么A和B就是扩展关系。**如果一个用例明显地混合了两种或者两种以上的不同场景，即根据情况可能发生多种分支，则可以将这个用例分为一个基本用例和一个或多个扩展用例，这样可能会使描述更加清晰。扩展用例为基用例添加新的行为。扩展用例可以访问基用例的属性，因此他能根据基用例中扩展点的当前状态来决定是否执行自己。而扩展用例对基用例不可见。如机房收费系统中“维护学生信息”操作时如果发现信息有误或者更新则需要使用“修改学生信息”用例完成更新，所以用例“查询上机记录”和“导出EXCEL”之间的关系就是扩展关系。“<<extend>>”是扩展关系的构造型，箭头指向基本用例。

**泛化**。**速记:A和B名称差不多大概率是泛化。**当多个用例共同拥有一种类似的结构和行为时，可以将他们的共性抽象成为父用例，其他的用例作为泛化关系的子用例。在用例的泛化关系中，子用例是父用例的一种特殊形式，它继承了父用例的所有结构、行为、关系。其中三角箭头指向父用例。假如在机房收费系统的注册可以通过本地注册和网上注册。

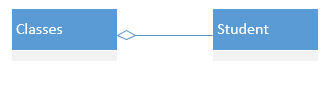
### 5.2.2.类之间的关系

类之间的关系包括：关联、聚合、组合、依赖、泛化 。

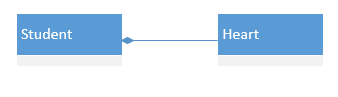
**关联**：关联关系使用实线加箭头表示，类之间的关系比依赖要强。学生与老师是关联的，学生可以不用电脑，但是学生不能没有老师。

IMG_256

**聚合**：聚合关系使用实线加空心菱形表示。聚合用来表示集体与个体之间的关联关系。例如班级与学生之间存在聚合关系。



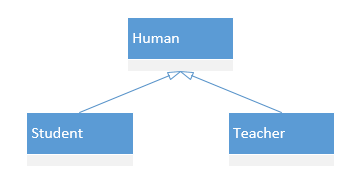
**组合**：复合关系使用实线加实心菱形表示。组合又叫复合，用来表示个体与组成部分之间的关联关系。例如学生与心脏之间存在复合关系。



**依赖**：依赖关系使用虚线加箭头表示。学生在学习生活中经常使用电脑，于是对电脑产生了依赖。依赖关系是五种关系中耦合最小的一种关系。类A要完成某个功能引用了类B，则类A依赖类B。

IMG_256

泛化：泛化是学术名称，通俗来讲，泛化指的是类与类之间的继承关系和类与接口之间的实现关系。



# 6.系统的可靠性分析与设计

## 6.1.可靠性

可靠性（Reliability）是指产品在规定的条件下和规定的时间内完成规定功能的能力。四个特性：成熟性，容错性，易恢复性，可靠性的依从性。

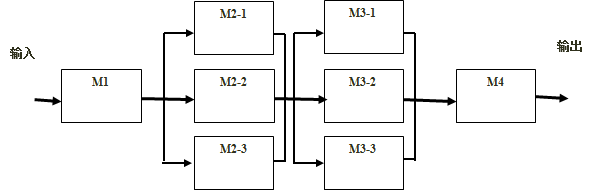
## 6.2.N版本程序设计

N 版本程序设计是一种静态的故障屏蔽技术，其设计思想是用 N 个具有相同功能的程序同时执行一项计算，结果通过多数表决来选择。其中 N 个版本的程序必须由不同的人独立设计，使用不同的方法、设计语言、开发环境和工具来实现，目的是减少 N 个版本的程序在表决点上相关错误的概率。

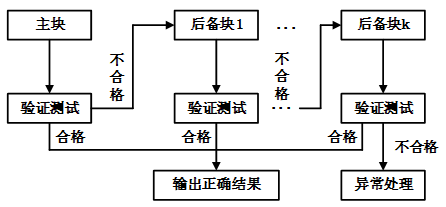


## 6.3.动态冗余

动态冗余又称为主动冗余，它是通过故障检测、故障定位及故障恢复等手段达到容错的目的。其主要方式是多重模块待机储备，当系统检测到某工作模块出现错误时，就用一个备用的模块来替代它并重新运行。各备用模块在其待机时，可与主模块一样工作，也可以不工作。前者叫热备份系统（双重系统），后者叫冷备份系统（双工系统、双份系统）。



## 6.4.恢复块方法



# 7.项目管理

## 7.1.项目计划

（1）项目背景

（2）项目经理、项目经理的主管领导、客户方联系人、客户方的主管领导，项目领导小组(项目管理 团队)和项目实施小组人员

（3）项目的总体技术解决方案

（4）所选择的项目管理过程及执行水平

（5）对这些过程的工具、技术和输入输出的描述

（6）选择的项目的生命周期和相关的项目阶段

（7）项目最终目标和阶段性目标

（8）进度计划

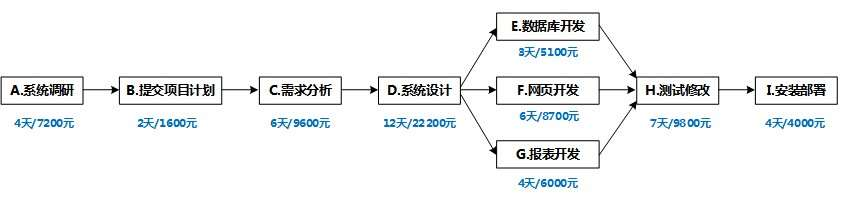
（9）项目预算

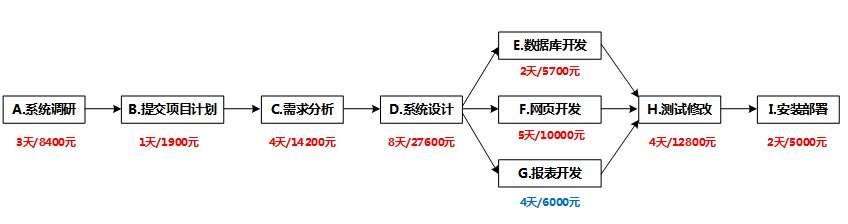
（10）变更流程和变更控制委员会

（11）对于内容、范围和时间的关键管理评审，以便于确定悬留问题和未决决策

## 7.2.最短/最长工期

画图。正常工作成本 = 7200 元+1600 元+9600 元+22200 元+5100 元+8700 元+6000 元+9800 元+4000 元=74200 元。正常工作工期 = 4+2+6+12+6+7+4=41 天。





最短工期成本=8400 元+1900 元+14200 元+27600 元+5700 元+10000 元+6000 元+12800 元+5000 元=91600 元。最短工期=3+1+4+8+5+4+2=27 天。

## 7.3.涉及到规划需要列表

若考察如何分配加班天数且要费用最低，需要把每个节点压缩天数和费用表格列出。然后按照压缩的费用排序，凑出天数。

