# 第一题 质量属性及架构评估

## 常规五大架构风格

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 |  | 风格1/方案1 | 风格2/方案2 |
| 1 |  | 面向服务架构 | 面向资源Rest架构 |
| 数据获取方式 | 将现有的多个系统和异构的数据源包装为服务，采用Web服务暴露数据接口，客户端需要通过服务调用获取数据，这种方法工作量大，复杂度较高。 | 绕开了复杂的功能封装，只需要明确数据的位置与标识，通过特定的网络协议直接使用标识定位并获取数据，工作量小，实现简单。 |
| 数据交互方式 | 采用远程过程调用和异步XML消息等模式实现数据交互，这种方式适合于系统之间功能调用时进行的少量数据传输，而在进行单纯的数据访问时效率不高，稳定性也较差。 | 以数据资源为核心，在对数据资源进行标识的基础上，通过标识符直接对数据资源进行访问与交互，实现简单且效率较高。 |
| 数据访问的上下文无关性 | 数据访问是与上下文有关的，具体表现在每次客户端进行数据请求都需要附加唯一的请求标识，并且服务端需要区分不同的客户端请求，效率较低。 | 数据访问是与上下文无关的，客户端通过全局唯一的统一资源标识符(URI)请求对应的数据资源，服务端不需要区分不同的客户端请求。 |
| 2 |  | 管道-过滤器 | 数据存储为中心 |
| 用户交互 | 管道-过滤器架构风格则**对用户的交互式数据处理支持有限**。 | 以数据存储为中心的架构风格能够很好地支持**交互式数据处理**。 |
| 扩展性 | 管道-过滤器架构风格同样以数据格式解耦数据处理过程之间的依赖关系，但其**在数据处理逻辑关系的灵活定义方面较差** | 以数据存储为中心的架构风格，以数据格式解耦各种功能之间的依赖关系，并可以灵活定义功能之间的逻辑顺序 |
| 数据管理 | 管道-过滤器架构风格通常只能支持有限度的数据格式，并且在数据格式转换方面的灵活性较差。 | 以数据存储为中心的架构将数据存储在统一的中心存储器中，中心存储器能够表示多种数据格式，并能够为数据格式转换提供各种支持。 |
| 3 |  | 管道-过滤器 | 仓库风格 |
| 灵活性(修改xxx) | 数据驱动机制，处理流程需要事先确定，**交互性差，对用户的交互式数据处理支持有限** | 模型驱动机制，以数据解耦功能关系，无需事先定义，交互性好 |
| 可扩展（增加xxxx） | 数据格式解耦数据处理过程之间的依赖关系，但数据和处理紧密耦合，加入新的处理时需要和数据进行适配，**在数据处理逻辑关系的灵活定义方面较差** | 数据和处理解耦，可动态添加和删除组件 |
| 性能 | 优点：   1. 支持开发调用，性能提高   缺点   1. 数据格式需要转换，性能降低 | 优点：   1. 不需要数据转换，性能提高   缺点：   1. 组件之间相互独立，不存在相互调用，性能降低 |
| 其他 | 1. 顺序结构 2. 使用数据流 3. 数据驱动流程 4. 接口进行适配 | 1. 星型结构 2. 使用文件或者模型 3. 业务功能驱动流程 4. 模型进行适配 |
| 4 |  | 管道-过滤器 | 主程序-子程序 |
| 灵活性(修改xxx) | 数据驱动机制，处理流程需要事先确定，**交互性差，对用户的交互式数据处理支持有限** | 子程序作为构件，由主程序调用，需要提前定义好处理顺序 |
| 扩展性(增加xxx) | 数据格式解耦数据处理过程之间的依赖关系，但数据和处理紧密耦合，加入新的处理时需要和数据进行适配，**在数据处理逻辑关系的灵活定义方面较差** | 加入新的算法时，需要重写主程序，重启系统 |
| 性能 | 数据在构件间传输，性能较差 | 共享存储区交换数据，性能高 |
| 5 |  | 面向对象 | 规则/解释器 |
| 灵活性(修改xxx) | 把xx封装成对象，在系统加载时加载，不可动态改变 | 把xxx描述可动态改变的规则数据 |
| 扩展性(增加xxx) | 加入新的xx，需要定义新的对象，并重启系统 | 加入新的xx，只需要定义新的规则，扩展解释规则 |
| 性能 | xx在系统内编码运行，直接运行，性能高 | 需要对规则实时解释，性能差 |



1. 对于拓展性，如果要重写代码的，要标明需**重启系统**。
2. 对于性能，对于能直接调用的，需标明**系统内编码，直接运行**。

## 架构风险点/权衡点/敏感点定义

架构分析风险点是架构设计中，潜在的、可能出现问题的决策带来的隐患。

架构分析敏感点是为实现某个质量属性，一个或多个构件中具有的特性。

架构分析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。

## 出题方式

### 架构对比

1、一般出现在架构分析，当中的某个小节

2、出题方式一般为“为了实现某个系统，提出了两个方案，请从xxx、xxx等方面对比，并说明选择某个方案的原因”

3、或者是没有给出对比方面，直接“说明选择某个方案的原因”

【解题思路】

【1】 给了对比方面

如果题目中说明了对比方面，那么选择这个方案的原因就是这几个方面，该方案优于别的方案，考虑这三个方面即可。

1、对比这几个方面，各个方案是如何实现的，从xx方面看，方案1采用的是xxxx，这种方案的优缺点是xxxx;而方案2采用xxxx优点是xxxxx。注意的是题目中对方案的实现描述并不会很清晰，答案仅仅用题目中的材料是不够的：

1、要真实理解这个架构的实现逻辑，可以画图理解，不能单从题目材料出发，每个交互都要带什么参数要考虑好才能对比优缺点

2、需要根据题目中的关键字积累该技术的优缺点，要积累。

2、通过以上对比，可以看到该方案架构清晰/效率较高/实现简单，更加满足系统需求。

【2】 没有给出对比方面

如果题目中没有给出对比方面，那么题目中描述系统会有非功能性要求，比如可修改性、性能、安全性等，从非功能性需求角度出发。

1、分析系统需求“xxxxx”（引用题目材料），该系统在xxxx方面需求较高。

2、方案1采用了xxxxx(方案的实现方法，同样的不能光依靠材料，需要积累)可以实现xxxx，但xxxxx；方案2采用了xxxxx，实现xxx，但xxxx

3、通过以上对比，方案1更能满足系统需求。

### 质量属性效应树

知道怎么填空就好了，简单的

# 第二题 软件建模

结构化系统建模 —> 三大模型：ER模型、数据流图、状态转换图

面向对象系统建模—> UML图

数据库建模—> ER图，转换到关系模型

## 情况1 结构化建模-DFD

### 信息工程方法中的“实体”和面向对象方法中的“类”之间不同

实体用于数据建模，而类用于面向对象建模。

实体只有属性，而类有属性和操作。

### 数据流图和流程图的区别

数据流图作为一种图形化工具，用来说明业务处理过程、系统边界内所包含的功能和系统的数据流。

流程图以图形化的方式展示应用程序，从输入到输出的逻辑过程，描述处理过程的控制流。

两者的区别有：

1.数据流图的处理结果可以并行，流程图在某个时间点只处于一个处理过程。

2.数据流图展示系统的数据流，流程图展示系统的控制流。

3.数据流程展示全局的处理过程，遵循不一样的计时标准，流程图处理遵循一致的计时标准。

4.数据流图适用于系统分析中的逻辑建模阶段，流程图适用于系统设计中的物理建模阶段

### DFD中的元素和定义

1.数据流，数据在系统内传播的路径，因此由一定成分固定的数据组成。

2.外部实体，代表系统之外的实体，可以是人、物或者其他软件系统。

3.加工，对数据进行处理的单元，接收一定的数据输入、对其进行处理，并产生输出。

4.数据存储，表示信息的静态存储，可以是文件、文件的一部分或者数据库元素。

### 补全DFD中的空

## 情况2 信息工程建模-数据库-ER

### 请说明关系型数据库开发中，逻辑数据模型设计过程包含哪些任务

逻辑结构设计阶段的主要任务是确定数据模型、将ER图转换成指定的数据模型、确定完整性约束、确定用户视图。

### 数据库结构模式

图示

描述已自动生成

自底向上：

1、内模式：数据**物理**结构和存储方式的描述，数据库内部的表示方式，只有一个内模式。内部视图：可以有多个内部视图。物理级数据库：描述数据的实际存储组织。

2、概念模式：**全体数据**的**逻辑**结构和特征描述，只有一个概念模式。DBA视图：所有用户的公共数据视图。概念级数据库：管理员可看到使用的数据库。

3、外模式：数据库用户能看到使用的**局部数据**的**逻辑**结构和特征描述，一个数据库有多个外模式，一个应用程序只有一个外模式。用户视图：可相互重叠，用户的所有操作针对用户视图。用户级数据库：外部记录组成。

### 分布式数据库结构

图示

低可信度描述已自动生成

分布式数据结构是在集中式数据结构的基础层加了一部分，其中局部内模式、局部概念模式就是对应局部数据库的内模式、外模式。

1、分布模式（分配模式）：描述数据在物理上各节点的分布形式，是物理分配视图。

2、分片模式：描述数据在逻辑上的分片形式，是全局数据逻辑划分的视图，每一个逻辑划分就是一个分片。

3、全局概念模式：分布式数据库的整体抽象，描述全局数据的逻辑结构。

4、全局外模式：全局概念模式的子集，是分布式数据库的最高抽象。

### 数据库设计步骤

1、需求分析：确定需求、确定设计目标、分析和收集数据、整理文档。

2、概念设计：将需求分析抽象成局部E-R模型，再将局部E-R模型集成为全局E-R模型

3、逻辑设计：确定数据模型、将ER图转换成指定的数据模型、确定完整性约束、确定用户视图。

4、物理设计：对于给定的数据模型，选取一个最适合应用环境的物理结构

### 什么是超类实体

超类实体由多个实体中所共有的属性组成。

如，收件人、寄件人可以组成用户实体，实体属性是两者共有属性。

### 数据库字段属性

1、派生属性可以由其他属性计算获得，用于存储计算结果值。

2、简单属性，原子性、不可再分的属性。

3、复合属性可以细分为更小的部分，即划分为一组更细粒度的属性。适用于某些用户访问全部属性，某些用户只想访问一部分属性的情况。

4、单值属性，对于一个特定的实体只有一个单独的值。

5、多值属性，一个属性对应一组值。（例如学生表单中的亲属属性，可能1个也可能2个）

### 完整性约束

1、实体完整性：确定基本关系的主属性不能取空值。

2、参照完整性：关系R1的外码和关系R2的主码对应，R1外码值只能为R2主码值或空。

3、用户自定义完整性：用户针对具体应用定义的完整性约束，可以用触发器实现。

## 情况3 面向对象建模-UML

### 面向对象方法中的“用例”可分为什么层次

用例按照层次，可以分为“Essential Use Cases”抽象用例、“Real Use Cases”基础用例。

抽象用例描述用例的本质属性，与如何实现这个用例无关，独立于实现该用例的软硬件技术。用于分析阶段。

基础用例描述用例的实现方式，表达设计和实现用例时采用的方法技术，用于设计阶段。

### UML图定义

UML是Unified Model Language的缩写，中文是统一建模语言，是由一整套图表组成的标准化建模语言，由构造块、公共机制、架构三部分组成，其中构造快由建模元素、关系、图组成。

动态图（7种）。动态图描述系统的动态模型和组成对象间的交互关系。

1、用例图。由参与者、用例、边界及它们之间的关系构成描述系统功能的视图，用于对系统的功能行为建模。

2、顺序图。序列图显示具体用例的详细流程，显示了流程中中不同对象之间的调用关系。

3、通信图。描述了接发消息的对象的组织关系，强调对象之间的合作关系。

4、时序图。时序图用来显示随时间变化，一个或多个元素的值或状态的更改。

5、状态图。**用于描述一个对象在其生存周期间的动态行为，表现一个对象所经历的状态序列、引起状态转移的事件，以及状态转移伴随的动作。**

6、活动图。**用于描述系统的工作流程和并发行为，活动图中一个活动结束后立刻进入下一个活动**。

特殊的，泳道活动图：将一个活动图分组，每个组表示一个特定的类或对象，它们负责完成组内的活动。每个活动都明确属于一个泳道，不可跨越泳道，而转移可以跨越泳道。

7、交互概览图。一个交互概览图是活动图的一种形式，它的节点代表交互图。

静态图（7种），静态图显示了系统的静态结构，特别是存在事物的种类的内部结构、相互之间的联系。

1、类图。展示模型的静态结构，描述类、接口、协作以及他们之间关系。

2、对象图。对象图表示在某一时刻一组对象以及它们之间关系。

3、包图。包图由包和包之间的关系构成，它是维护和控制系统总体结构的重要建模工具。

4、构件图。描绘了系统中的构件接口，以及它们之间的关系。

5、部署图。部署图表示该软件系统如何部署到硬件环境中。

6、制品图。展示了一组制品以及其间依赖关系。利用制品图可以对系统的静态实现视图建模。

7、组合结构图。描述了一个"组合结构"的内部结构，以及他们之间的关系。

|  |  |  |  |
| --- | --- | --- | --- |
| 包含/使用关系<include>  /<use> | 提取出的公共部分A，功能B需要用到A | 箭头指向公共部分A |  |
| 扩展关系 <extend> | 主用例A，辅用例B  实现A的基本功能，有可能会拓展使用B | 箭头指向主用例A |  |
| 泛化关系 | 强调父子关系 | 箭头指向父类，箭头是空心 |  |

### 状态图和活动图的区别

1、状态图侧重描述行为的结果；活动图侧重描述行为的动作。

2、状图图不可描述并发行为；而活动图可以描述并发行为。

### uml图中类图与用例图内的关系

用例图关系有三种：

1、包含(使用)关系：include，用例可以简单地包含其他用例具有的行为。

2、扩展关系：extend，指在一定条件下，把新的行为加入到已有的用例中，其中获得的新用例称为扩展用例。

3、泛化关系：强调父子关系

类图关系有六种：

1、依赖关系。如果类A用到类B，但是和B的关系不是太明显的时候，就可以把这种关系看作是依赖关系。

2、关联关系。指类和类之间的连接，它使一个类知道另一个类的属性和方法。

3、聚合关系。聚合关系体现的是整体与部分的拥有关系。整体与部分之间是可分离的，它们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享，

4、组合关系。组合关系体现整体与部分间的包含关系，整体与部分是不可分的，部分也不能给其它整体共享，作为整体的对象负责部分的对象的生命周期。

5、泛化关系。泛化关系指类与类之间的继承关系。

6、实现关系。实现关系是指接口及其实现类之间的关系

|  |  |  |  |
| --- | --- | --- | --- |
| 依赖关系 | 修改x的定义会引起另一个y的修改，常见于局部变量，方法参数等，如y中一个变量x | 虚线、实心箭头、指向局部变量 |  |
| 泛化关系 | 父子关系 | 实线、空心箭头、指向父类 |  |
| 关联关系 | 最弱、通用的语义联系 | 一条实线 |  |
| 聚合关系 | 部分与整体，且生命周期不同，可见于构造器中的成员变量。如汽车（引擎、轮胎） | 实线、空心菱形箭头、指向整体 |  |
| 组合关系 | 部分与整体，紧密联系、生命周期相同。如公司（部门） | 实线、黑色实心菱形箭头，指向整体 |  |
| 实现关系 | 接口及实现类 | 虚线、空心箭头、指向接口 |  |

### 用例图的组件和模型构建步骤

1、参与者（用小人表示），表示系统之外，需要使用系统或与系统交互的事物，包括人/组织、设备、外部系统。

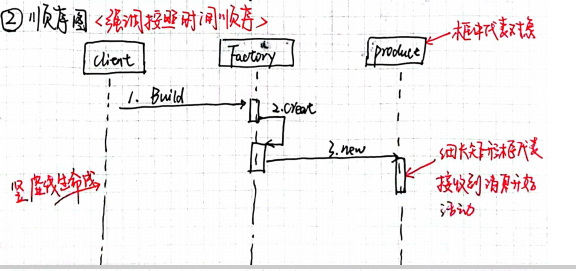
（经过一段时间引发某个时间--->参与者为时间）

2、用例（用椭圆表示），代表系统的功能单元。

用例模型描述的是外部参与者所理解的系统功能。用例图模型构建步骤：识别参与者、合并需求获取用例、细化用例描述、调整用例模型（可选）。

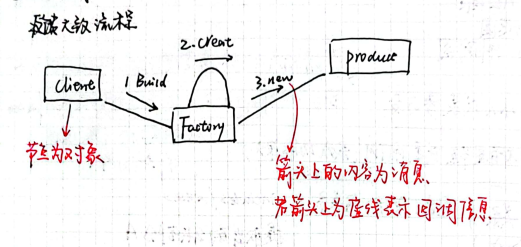
### UML图样例（可能会填图）

#### 顺序图



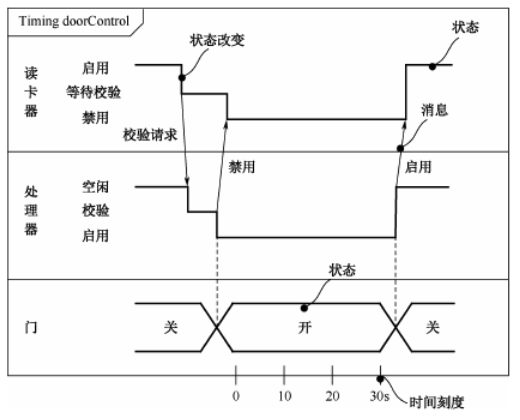
强调按照时间顺序

#### 通信图



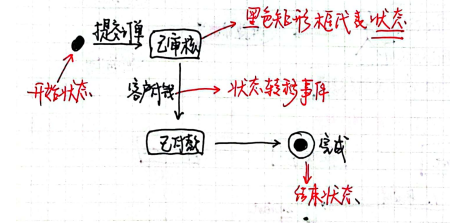
#### 定时图

强调实际时间



#### 状态图

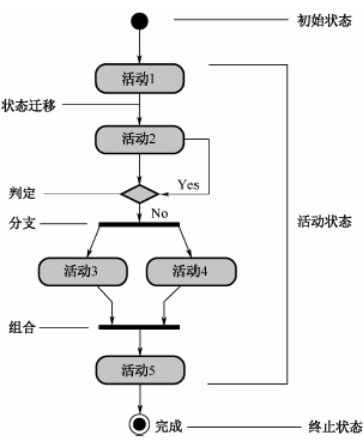
用于描述状态转换和复杂对象



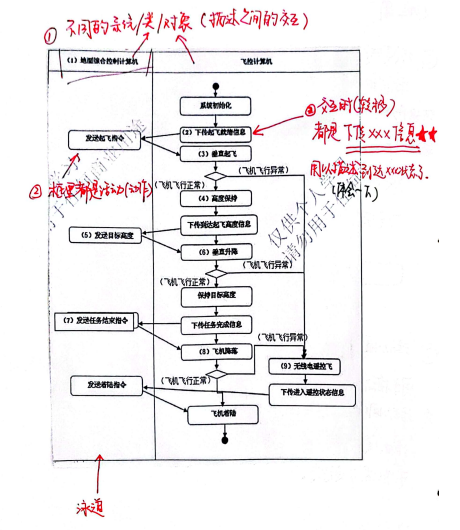
#### 活动图

大致流程（内部工作过程）+并发行为

描述操作（类方法）的行为，也可描述用例和对象内部的工作过程，同时还可描述并发行为



#### 泳道活动图



### 4+1视图

“4+1”视图模型从5个不同的视角来描述软件架构，每个视图只关心系统的一个侧面，5个视图结合在一起才能反映系统的软件结构的全部内容。

这5个不同的视角包括逻辑视图、开发视图、进程视图、物理视图和场景视图：

1、逻辑视图。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。使用UML中的类图描述逻辑视图。（系统有哪些功能，拆分成哪些类）

2.、实现视图。开发视图也称为模块视图、开发视图，侧重于软件模块的组织和管理，考虑软件内部的需求。使用UML中的实现图来来建模。

3、进程视图。进程视图侧重于系统的运行特性以及逻辑视图中的功能抽象如何适应进程结构等，关注一些非功能性需求。

4.、物理视图。物理视图考虑如何把软件映射到硬件上，解决系统拓扑结构、系统安装和通信等问题。使用UML中的部署图来建模。

5、场景。场景可以看作是系统中重要系统活动的抽象，它使4个视图有机联系起来。场景使用UML中的用例图建模。

|  |  |  |  |
| --- | --- | --- | --- |
| 视图 | 特性 | 关注人员 | 关注点 |
| 逻辑视图 | 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。 | 最终用户 | 系统功能 |
| 开发视图 | **开发视图也称为模块视图，在 UML 中被称为实现视图**，它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。 | 程序员 | 系统的配置、装配等问题 |
| 进程视图 | 进程视图侧重于系统的运行特性，主要关注一些非功能性需求，例如，系统的性能和可用性等。进程视图强调并发性、分布性、系统集成性和容错能力。 | 系统集成人员 | 系统的性能、可伸缩性、吞吐率等问题 |
| 物理视图 | **物理视图在 UML 中被称为部署视图**，它主要考虑如何把软件映射到硬件上，它通常要考虑到解决系统拓扑结构、系统安装和通信等问题。 | 系统工程师 | 系统的发布、安装、拓扑结构等问题 |
| 场景 | 场景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。**场景视图对应 UML 中的用例视图**。 | 分析人员和测试人员 | 系统的行为 |

# 第三题 数据库与缓存

## 数据库

### 对比

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 |  | 数据库1 | 数据库2 |
| 1 |  | 关系型数据库 | NoSQL |
| 性能 | 关系数据库应付上万次SQL查询还勉强可以，但是应付上万次SQL写数据请求，硬盘I／O就已经无法承受了。特别是涉及到多表连接操作，会导致响应变慢 | NoSQL数据库支持高并发数据访问，性能较高 |
| 可用性 | **关系数据库采用中央数据存储，容易成为系统的性能瓶颈，单点故障**很容易导致系统崩溃，负载过高往往导致系统出现宕机现象 | NoSQL数据库基于分布式数据存储，不存在单点故障和性能瓶颈，系统可用性高 |
| 扩展性 | **关系数据库多采用中央数据存储，使得数据容量受限于前期设计的上限，很难实现数据容量的横向扩展** | NoSQL数据库能够支持海量数据的存储，且易于横向扩展 |
| 数据格式 | **关系数据库中采用元组方式组织数据，难以使用新型数据格式，难以维护** | NoSQL数据库的数据存储结构松散，能够灵活支持多种类型的数据格式 |
| 2 |  | 关系型数据库 | 文件系统 |
| 设计难度 | **数据结构需要符合关系模式，设计难度大** | 针对特定应用系统，难度较小 |
| 数据冗余 | 遵守数据库范式，冗余少 | 可能在多个文件中复制相同的数据属性，冗余大 |
| 数据架构 | 以数据库为中心，组织管理数据 | 以应用为中心，组织管理数据 |
| 应用扩展 | 数据独立于系统应用，数据库接口标准化，易于在应用间共享数据 | 符合特定应用系统要求，文件数据很难在不同应用系统中共享 |
| 3 |  | 关系型数据库 | 内存型数据库 |
| 主要数据类型 | 关系模式 | 键值对模式 |
| 读写性能 | 外存读写，性能相对较低 | 内存直接读写 |
| 存储容量 | 基于磁盘存储，存储容量大 | 基于内存存储，存储容量受限 |
| 可靠性 | 内建恢复机制，可靠性较高 | 恢复机制复杂，可靠性较低 |

### 数据库事务的实现方式

1、原子性。数据库中实现原子性和持久性最简单的策略是“影子拷贝”，该策略假定某个时刻只有一个活动的事务，首先对数据库做副本，所有的写操作都在数据库副本上执行，而保持原始数据库不变，若任一时刻操作终止，只需要删除副本即可，原始数据不受影响。

2、一致性。常见完整性约束检查保持一致性。

3、隔离性。两段锁协议是实现隔离性的常见方案。通过定义锁的增长、收缩两个阶段约束事务的加锁、解锁过程，保证事务串行化执行，但由于事务不能一次性获得所有的锁，有可能死锁。

4、持久性。所有的数据库修改操作均在日志中记录，事务的写操作等到事务提交后才会执行。日中中记录事务开始、事务结束时间、以及事务修改的新值。

### 为什么要用Memcached缓存代替数据库查询缓存

1.缓存架构，数据库查询缓存通常每个数据库只有一个实例，因此存储内容受数据库服务器内存限制，可缓存数据有限。而Memcached可采用高速分布式缓存服务器结构，不受数据库服务器约束，可拓展性更好。

2.缓存有效性：数据库查询缓存只要发生写操作就会失效，即使更新的是数据库中的其他行。而Memcached可通过键值对将数据散列缓存，有效降低缓存的更新频率，从而提高缓存有效性。

3.缓存数据类型：数据库查询缓存只能缓存数据库行，而Memcached理论上可以缓存任何内容，可将分散在数据库的关系或列表组合后缓存，提高缓存针对性和效率。

### 集中式数据库 VS 分布式数据库，两种如何实现拓展

集中式数据库架构是一个由处理器、与它相关的数据存储设备以及其他外围设备组成，它被物理地定义到单个位置，系统提供数据处理能力，用户可以在同样的站点上操作，也可以在地理位置隔开的其他站点上通过远程终端操作。

分布式数据架构可以使用多个计算机系统的多个局部数据库系统组成，数据可以在多个不同的局部数据库中进行传送，由不同的数据库管理系统软件进行管理，运行在不同的计算机上，支持多种不同的操作系统，这些机器位于不同的地理位置并通过多种通信网络连接在一起，一个应用程序可以操作位于不同地理位置机器上的数据。

集中式数据库通过向上扩展提升，具体实现方式包括硬件扩容（增加CPU数量、内存容量、磁盘数量）和硬件升级（更换高端主机或高速磁盘）。

分布式数据架构通过向外扩展提升，具体的实现方式包括数据复制、数据垂直切分或水平切分、缓存和全文搜索。

### “局部数据库+缓存”的读写分离 + 分布式 + 避免单点故障的建设方案

局部数据库+缓存的读写分离为：局部数据库写，缓存读

分布式：多个数据库系统

避免单点故障：热备份数据库系统

故，以上的建设方案为：

由多个局部数据库系统、多个热备份数据库系统、多个数据缓存组成。

局部数据库负责数据的写入，多个热备份数据库系统用以解决单点故障的问题，多个数据缓存负责为应用提供所读取的数据。

读取操作：应用层访问缓存，若命中则返回，否则从局部数据库中读取并将数据加载至缓存后返回。

添加操作：采用延迟加载策略，将数据直接写入局部数据库。

修改操作：更改局部数据库中的数据，将缓存中的数据标记为无效。

删除操作：删除局部数据库中的数据，将缓存中的数据标记为无效。

### 数据库规范化设计定义、反规范化设计定义、优点、缺点、实现技术

规范化设计：通过对数据库表的设计，降低数据库冗余程度，是数据库系统设计的重要技术。

反规范化设计：规范化设计后，数据库设计者希望牺牲部分规范化提高性能，这种从规范化设计回退的方法称为反规范化技术。

优点：

1.降低连接操作的需求

2.降低外键和索引的数目

3.提高查询效率

4.有可能减少表的数目

缺点：

1.数据重复存储，浪费磁盘空间

2.可能出现数据完整性问题

3.为保障数据一致性，增加了数据维护的复杂度

4.降低了修改速度

实现技术：

1.增加冗余列。在多个表中保留相同的列，通过增加冗余，减少或避免查询时的连接操作。

2.增加派生列。在表中增加可有本表或其他表中数据计算生成的列，减少查询的连接操作，避免计算或使用集合函数。

3.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，则把两个表重新组成一个表，减少连接提高性能。

4.水平切分表。根据一列或多列数据的值，把数据放在多个独立的表中，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在多个介质上时使用。

5.垂直切分表。将主键和部分列放在一个表中，主键和其他列放在另外一个表中，在查询时减少IO次数。

## Redis

### Redis分布式存储方案

Redis分布式存储常见方案：

1、主从模式：一主多从，故障时手动切换

2、哨兵模式：带哨兵的一主多从，主节点故障时自动选择新的˙主节点

3、集群模式：分节点对等集群，分slots，不同的slot信息存储到不同的节点。

### Redis集群切片的常见方式

Redis集群切片常见方式：

1、客户端切片，客户端通过key的哈希值对应到不同的服务器。

2、中间件切片，在应用软件和Redis之间，由中间件实现服务到后台Redis节点的路由分派。

3、客户端服务端协作分片。客户端采用一致性哈希，服务端提供错误节点的重定向服务，对应到不同的服务器。

### Redis分片方案

1、范围分片：根据数据范围来分片

2、哈希分片：通过对key做哈希运算来分片

3、一致性哈希分片：哈希分片的改进，解决重新分配节点的无法命中问题，利于扩展节点

### Redis常见问题

#### 缓存雪崩

【大部分缓存失效导致数据库崩溃】

解决方案

1、使用锁或队列:保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。

2、为key设置不同的缓存失效时间:在固定的一个缓存时间的基础上+随机一个时间作为缓存失效时间。

3、二级缓存:设置一个有时间限制的缓存+一个无时间限制的缓存。避免大规模访问数据库。

#### 缓存穿透

【无效数据，导致查询数据】

解决方案

1、如果查询结果为空，直接设置一个默认值存放到缓存，这样第二次到缓冲中获取就有值了。设置一个不超过5分钟的过期时间，以便能正常更新缓存。

2、设置布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

【布隆过滤器】

布隆过滤器用于快速识别1个元素不在一个集合中。通过一个长二进制向量和一系列随机映射函数来记录与识别某个数据是否在一个集合中。

优点：

1、占用内存小

2、查询效率高

3、不需要存储元素本身，在某些对保密要求比较严格的场合有很大优势

缺点：

1、有一定的误判率，即存在假阳性，不能准确判断元素是否在集合中。

2、一般情况下不能从布隆过滤器中删除元素

3、不能获取元素本身

#### 缓存预热

【系统上线后，将需要缓存的数据缓存到缓存中】

解决方案：

1、直接写个缓存刷新页面，上线时手工操作。

2、数据量不大时，可以在项目启动的时候自动进行加载。

3、定时刷新缓存。

#### 缓存更新

除Redis系统自带的缓存失效策略，常见采用以下两种:

1、定时清理过期的缓存。

2、当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

#### 缓存降级

降级的目的是保证核心服务可用，即使是有损的，而且有些服务是无法降级的(如电商的购物流程等)；

在进行降级之前要对系统进行梳理，从而梳理出哪些必须保护，哪些可降级。

### Redis的存储类型

1、String

2、List

3、HashMap（用于结构化存储）

4、Set

5、Sorted Set

### Redis过期删除

1、定时删除：在设置某个key 的过期时间同时，我们创建一个定时器，让定时器在该过期时间到来时，立即执行对其进行删除的操作。

优点：定时删除对内存是最友好的，能够保存内存的key一旦过期就能立即从内存中删除。

缺点：对CPU最不友好，在过期键比较多的时候，删除过期键会占用一部分 CPU 时间，对服务器的响应时间和吞吐量造成影响。

2、惰性删除：设置该key 过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key。

优点：对 CPU友好，我们只会在使用该键时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查。

缺点：对内存不友好，如果一个键已经过期，但是一直没有使用，那么该键就会一直存在内存中，如果数据库中有很多这种使用不到的过期键，这些键便永远不会被删除，内存永远不会释放。从而造成内存泄漏。

3、定期删除：每隔一段时间，我们就对一些key进行检查，删除里面过期的key。

优点：可以通过限制删除操作执行的时长和频率来减少删除操作对 CPU 的影响。另外定期删除，也能有效释放过期键占用的内存。

缺点：难以确定删除操作执行的时长和频率。另外最重要的是，在获取某个键时，如果某个键的过期时间已经到了，但是还没执行定期删除，那么就会返回这个键的值，这是业务不能忍受的错误。

4、定期删除和惰性删除结合的方式，这种方式的失效场景是Redis中key的过期时间都设置的相同，Redis同时失效，导致所有请求转发到数据库，数据库压力增大。

### Redis内存淘汰机制

1、noeviction：当内存使用超过配置的时候会返回错误，不会驱逐任何键，默认机制。

2、volatile-ttl：从配置了过期时间的键中驱逐马上就要过期的键，ttl值越大，越快过期。

3、allkeys-lru：加入键的时候，如果过限，首先通过LRU算法驱逐最久没有使用的键

4、volatile-lru：加入键的时候如果过限，首先从设置了过期时间的键集合中驱逐最久没有使用的键

5、allkeys-random：加入键的时候如果过限，从所有key随机删除

6、volatile-random：加入键的时候如果过限，从过期键的集合中随机驱逐

7、volatile-lfu：从所有配置了过期时间的键中驱逐使用频率最少的键

8、allkeys-lfu：从所有键中驱逐使用频率最少的键

总结下来就是：

1、不淘汰

2、ttl

3、随机

4、lru (least Recently Used)最少访问

5、lfu (Least Frequently Used)）最不经常访问

### Redis持久化

1、RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照。

2、AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。

|  |  |  |
| --- | --- | --- |
|  | RDB | AOF |
| 备份量 | 重量级的全量备份，保存整个数据库 | 增量级增量备份，一次只保存一个修改命令 |
| 保存间隔时间 | 保存时间间隔长 | 保存时间间隔短，默认1s |
| 恢复性能 | 恢复大数据集的时候会更快 | 数据恢复比较慢 |
| 阻塞情况 | Save会阻塞,bgsave或者自动不会阻塞 | 不会阻塞 |
| 文件大小 | 压缩过的非常紧凑的文件 | 对于相同的数据集，AOF文件的体积要大于RDB文件 |
| 数据安全 | 数据安全性不如AOF，容易丢数据 | 安全性更高，根据策略决定 |
| 保存过程 | 保存整个数据集的过程是比繁重的，比较耗CPU、耗时 | AOF文件是一个只进行追加的日志文件，保存过程轻量 |
| 数据丢失 | 如果服务器宕机，那么就可能丢失几分钟的数据 | 数据更完整，，秒级数据丢失 |

## Memcached

### Memcached VS Redis

Memcache是一个高性能的分布式的内存对象缓存系统，用于动态Web应用以减轻数据库负载，Memcache通过在内存里维护一个统一的巨大的hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。

Redis是一个开源语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的APl。

|  |  |  |
| --- | --- | --- |
|  | Memcache | Redis |
| 数据类型 | 简单key/value结构 | 数据类型有key-value、list、set、hash等多种数据结构 |
| 持久性 | 不支持 | 支持 |
| 分布式存储 | 客户端哈希分片/一致性哈希 | 多种分布式存储方式，主从、哨兵、集群 |
| 多线程支持 | 支持 | Redis5.0之前不支持；  Redis5.0之后支持 |
| 内存管理 | 私有内存池/内存池 | 无内存管理 |
| 事务支持 | 不支持 | 有限支持 |
| 数据容灾 | 不支持，不能做数据恢复 | 支持，可在灾难发生时，恢复数据 |

### Memcached缓存可靠性和一致性问题

Memcache缓存可靠性和一致性问题：

1、MemCache没有持久化功能，掉电后数据全部丢失，且无法恢复，存在可靠性问题。

2、MemCache不支持事务，操作过程中存在数据不一致问题。

## 其他

### 分布式缓存定义

分布式缓存指的是在高并发环境下，为减轻数据库压力和提高系统响应时间，在数据库系统和应用系统之间增加的独立缓存系统。

### 解决缓存雪崩

1、缓存失效后，通知加排它锁或者队列的方式控制数据库写缓存的线程数量，缓存更新串行化

2、给不同key设置随机或者不同的失效时间，使得失效时间尽量均匀

3、设置两级或多级缓存，避免访问数据库

### 解决黑客伪造大量无效key，导致查询数据库的问题

无效解决方案为无效的key置空值的问题：不在系统的key值是无限的，如果均设置为空，会造成内存资源的极大浪费，导致性能急剧下降。

正确解决方案：查询缓存之前，首先对key值进行过滤，只允许系统中存在的key值进行后续操作。

### 下面数据库缓存读写方式为什么没有解决数据不一致的问题

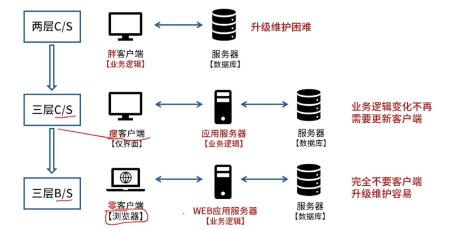
“应用程序读取数据时，首先读缓存，如果缓存不存在再读取数据库；写数据时，先写缓存再写数据库或先写数据库再写缓存”

问题在于：在写数据时，有可能存在缓存写成功、数据库写失败或者数据库写成功、缓存写失败的情况，从而导致数据不一致。也有可能在多个请求发生时，产生读写冲突的并发行为。

# 第四题 Web架构综合考察

## 层次型软件架构

### 层次C/S B/S架构



### MVC 架构风格

#### MVC

MVC架构风格按照业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集在一个部件里，在改进和个性化定制界面时，不需要重新编写业务逻辑。

MVC构架把整个软件系统分为模型、视图和控制器三个部分。

1、模型负责维护持久性的业务数据，实现业务处理功能，并将业务数据的变化及时通知给视图。

2、视图负责呈现模型中包含的业务数据，响应模型变化，更新呈现形式，并向控制器传递用户的界面动作。

3、控制器负责将用户的界面动作映射为模型的业务处理功能，并调用模型，根据返回结果选择新的视图。

MVC模式允许多种界面的拓展，视图的变更与增加，与模型无关。

MVC模式易于维护，控制器和视图随着模型的扩展而扩展，只需要保持公共接口，旧版本可继续使用。

MVC模式可支持强大功能的界面。

在J2EE体系中：

* 1. View：JSP
  2. Controller：Servlet
  3. Mode：Entity Bean、Session Bean(EJB)

#### MVP

MVP是MVC的变种。

在MVC中，某些场景下，View层直接采用观察者/发布订阅模式监听Model层的变化，这样View层和Model层相互持有、相互操作，导致紧密耦合，在可维护性上有待提升。由此，MVP模式应运而生。为了解决这个问题，MVP模式将View层和Model层解耦，之间的交互只能通过Presenter层，实际上，MVP 模式的目的就是将View层和Model层完全解耦，使得对View层的修改不会影响到Model层，而对Model层的数据改动也不会影响到View层。

MVP 实现了 V与M的解耦，其优点包括:

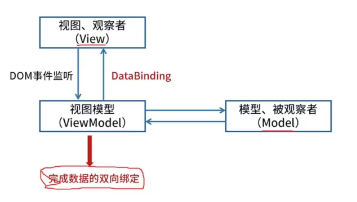
* 1. 模型与视图完全分离，可以修改视图而不影响模型
  2. 可以更高效地使用模型，因为所有交互都发生在一个地方【Presenter】内部
  3. 可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑
  4. 如果把逻辑放在Presenter中，就可以脱离用户接口来测试这些逻辑(单元测试)

#### MVVM

View ：html，CSS，模板文件

ViewModel : Js , runtime, compler

Model： Java／C＃业务逻辑处理



### 数据访问/持久层

#### 定义

数据持久层根据分层思想，通过建立逻辑操作接口，采取一定关系/对象映射策略，隐藏数据库访问代码细节，向业务人员提供透明化的数据操作机制。

#### 带来的好处

1.简化数据层访问，隐藏数据库细节，即便更改数据库，只需修改配置文件，不必重写代码程序

2.通过OR映射，向业务逻辑提供面向对象的访问

3.分离业务逻辑和数据访问，简化开发工作，关注于业务逻辑开发。

4.业务逻辑代码可读性强，不会有大量SQL

5.持久化技术可自动优化，减少对数据库访问，提高效率。

#### 在持久层设计阶段需要考虑的问题

1.网络流量问题；

2.返回结果集的问题；

3.查询或锁定超时的问题；

4.应用程序开发工具的问题；

5.使用游标的问题；

6.应用层设计的问题等

#### Hibernate VS Mybatis

|  |  |  |
| --- | --- | --- |
|  | Hibernate | Mybatis |
| 简单对比 | 强大、复杂、间接、**SQL无关** | 小巧、简单、直接、**SQL相关** |
| 模型 | ORM | SQL Mapping |
| 可移植性 | 不关心具体的数据库，可移植性强 | 根据数据库SQL编写，可移植性差 |
| 简单多表关联 | 不支持 | 支持 |
| 复杂对象 | 支持对象的继承和聚合等立体化关系 | 不支持 |

#### ORM和SQLMapping定义及优缺点

ORM对象关系映射，在对象模型和关系型数据库之间建立起对应关系，并且提供一种机制，可通过 JavaBean 对象操作数据库表中的数据。使得程序员在开发过程中仅仅面对一个对象的概念，降低了对程序员数据库知识的要求，简化了数据库相关的开发工作。

ORM优点：

1、使用ORM可以大大降低学习和开发成本。

2、程序员不用再写SQL来进行数据库操作。

3、减少程序的代码量。

4、降低由于SQL代码质量差而带来的影响。

ORM缺点(不写SQL导致性能低，不能处理复杂语句)

1、不太容易处理复杂查询语句。

2、性能较直接用SQL差。

SqlMapping，致力于POJO和SQL语句之间的映射，将接口和 Java对象映射成数据库中的记录。

优点：

1、入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

2、可以进行更为细致的SQL优化，可以减少查询字段。

缺点

1、框架还是比较简陋，功能尚有缺失，不支持数据复杂、数据对象聚合、继承等关系

2、仍需要使用SQL，不利于移植数据库。

#### 数据在线访问的优缺点

优点：

1、性能比直接SQL好

2、可以处理复杂查询语句

缺点：

1、要求程序员懂SQL语句

2、修改与维护相对困难

## Rest架构风格

### 定义

Rest是一种通常使用HTTP和xml进行web通信的技术，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确定，客户端应用程序通过URI获取资源的表现，并通过获取资源的动作使其状态改变。

Rest风格的特点是客户端/服务器、无状态、缓存、统一接口、分层系统、按需代码。

优点：

1.实现技术成熟、简单

2.基于URI和超链接，不需要集中式的服务信息仓库

3.支持缓存、无状态、可支持大量客户端

4.轻量级Web框架

5.测试相对简单，用浏览器即可

缺点：

1.Rest要求输入参数以URI传递，对参数容量有一定限制

2.在URI中表达复杂类型的参数比较困难，且目前不存在公认编组/解组方法

3.Rest提倡的风格和实际实现仍有差距，如高层Rest用GET、POST、DELETE、PUT四种方法，而实际应用通常只采用GET和POST。

Rest将**资源、资源的表现、获取资源的动作**三者分离。

### rest的五个原则

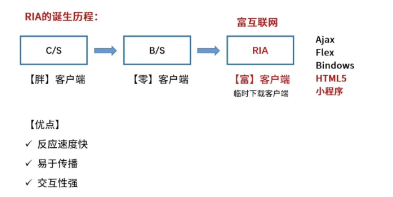
1. 网络上所有的事物都抽象为资源。
2. 每个资源对应一个唯一的资源标识。
3. 通过通用的连接件接口对资源进行操作。
4. 对资源的各种操作不会改变资源标识。
5. 所有的操作都是无状态的。

## RIA富互联网架构风格

RIA 结合了CS架构反应速度快、交互性强的优点，以及BS架构传播范围广及容易传播的特点。数据能够被缓存在客户端，从而可以实现一个比基于HTML的响应速度更快目数据往返于服务器的次数更少的用户界面。

技术：JAVA，AJAX，H5，Flex，Bindows，Laszlo

特点：打开比较慢，因为需要加载，之后的运行表现力强，效率高。大部分网页游戏采用。



## 微服务和SOA

#### 微服务

微服务是一种架构风格，在单个应用中包含众多松散耦合、课独立部署的小型服务或组件，这些服务通常拥有自己的技术栈，包括数据库和数据管理模型。

微服务的优点：

1、复杂应用解耦：化整为零，易于团队开发。

2、独立：独立开发、独立测试及独立部署、独立运行。

3、技术选型灵活：支持异构，每个服务使用不同的技术栈。

4、容错：故障被隔离在单个服务中，通过重试等机制可实现应用层容错

5、松耦合易扩展：可根据需求独立扩展

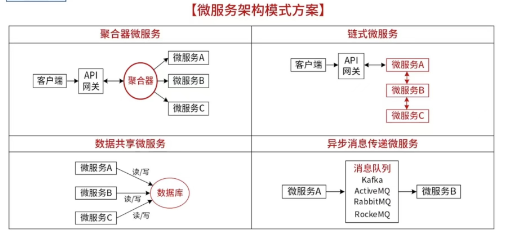
面临的问题有：

1、分布式环境下的数据一致性

2、测试复杂，服务间具有依赖测试

3、运维复杂

#### 微服务架构模式方案



#### 微服务设计约束

* 1. 微服务个体约束。每个微服务都是独立的，修改一个微服务不能影响另一个微服务】
  2. 微服务与微服务之间的横向关系。通过第三方服务注册中心来满足服务的可发现性
  3. 微服务与数据层之间的纵向约束。数据是微服务的“私产”，访问时需要通过微服务
  4. 全局视角下的微服务分布式约束。高效运维整个系统

### 微服务和SOA区别

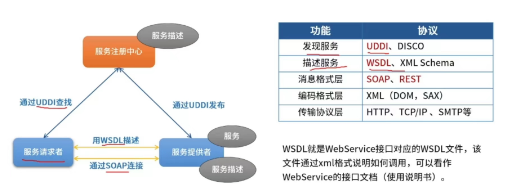
* 1. 微服务比sOA更精细，可以独立进程方式存在。
  2. 微服务接口更通用化，如用HTTPRESTful，各种终端都可调用，语言无关，平台无关。
  3. 微服务更倾向于分布式部署，互联网场景更适合。

|  |  |
| --- | --- |
| 微服务 | SOA |
| 能拆分的就拆分 | 是整体的，服务能放一起的都放一起 |
| 纵向业务划分 | 水平分多层 |
| 由单一组织负责 | 按层级划分不同部门的组织负责 |
| 细粒度 | 粗粒度 |
| 两句话可以解释明白 | 几百字只相当于SOA的目录 |
| 独立的子公词 | 类似大公司里面划分了一些业务单元（BU） |
| 组件小 | 存在较复杂的组件 |
| 业务逻辑存在于每一个服务中 | 业务逻辑横跨多个业务领域 |
| 使用轻量级的通信方式，如HTTP | 企业服务产总线（ESB）充当了服务之间通信的角色 |

|  |  |
| --- | --- |
| 微服务架构实现 | SOA实现 |
| 团队级，自底向上开展实施 | 企业级，自顶向下开展实施 |
| 一个系统被折分成多个服务，粒度细 | 服务由多个子系统组成，粒度大 |
| 无集中式总线，松散的服务架构 | 企业服务总线，集中式的服务架构 |
| 集成方式简单 （HTTP／REST／JSON） | 集成方式复杂（ESB／WS／SOAP） |
| 服务能独立部署 | 单块架构系统，相互依赖，部署复杂 |

### SOA

#### SOA



SOA是一个组件模型，将应用程序不同功能单元封装为服务，通过定义好良好的接口和契约联系起来，接口采用中立的方式定义，不涉及底层编程接口和通信模型，使得系统中的服务可以以一种统一、通用的方式交互。

面向服务架构的常用实现方式包括webservice、ESB。

webservice包含服务提供者、服务请求者和服务注册中心

1、服务提供者通过发布的Web服务描述语言（WSDL），定义了服务的接口、入参和出参等信息，描述对外提供的服务功能；

服务提供者主要完成服务的设计、描述、定义和发布工作。在服务设计阶段，如何高效地对服务进行设计和定义是一个难题。为解决该问题，制定出基本的设计原则：需要对业务流程分析梳理，并综合考虑粗粒度、松耦合、自包含等特点，同时为了避免服务间通信量过大、交互频繁，需尽量减少了服务的数量。

2、服务请求者通过简单对象访问协议(SOAP)访问服务提供者，SOAP主要定义了服务之间如何交互以及消息格式；

服务的请求者即是服务的消费者，通过webservice技术调用服务。服务请求者通过服务注册中心可获取对应的服务接口、参数和返回值，之后利用webservice技术传递服务所需要的参数，进而调用该服务相关的运算、处理和分析。

3、服务注册中心一般采用UDDI实现，UDDI用于描述、发现与集成服务，服务提供者向UDDI发布服务，服务调用者查询UDDI上已发布的服务。

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服务描述，服务请求者在此查找需要的服务。虽然在某些情况下服务注册中心是可选角色，但注册中心的存在可使服务提供者和服务消费者进一步解耦。

ESB是一个实现系统间集成和互联互通的重要技术架构，基于内容的路由与过滤，具备复杂数据的传输能力，可提供一系列标准接口

WSDL就是WebService接口对应的WSDL文件，该文件通过xml格式说明如何调用，可以看作WebService的接口文档。

SOA包含的服务有：

1. 交互服务。交互服务实现人与服务之间的交互功能。人可以是服务的消费者,也可以是服务的提供者。人员需要请求服务时，向连通服务发送消息请求，由连通服务查找服务，并将请求消息传递给服务提供者。

2.连接服务。连接服务又称连通服务，在完成服务的接入、服务间的通信和交互基础上,还提供安全性、可靠性、高性能的服务能力保障。

3. 业务服务。业务服务指为新建服务提供的特定运行支持环境。

4. 业务流程服务。流程服务是业务流程的运行环境，提供流程驱动、服务调用、事务管理等功能。

5. 信息服务。信息服务特指为上层应用系统、同层的其他服务等提供数据访问及资源访问服务。其目标是使应用系统能够统一、透明、高效地访问和操纵位于网络环境中的各种分布、异构的数据资源。

6. 协作服务。协作服务既可以实现组织之间的交互通信,也可以实现组织内部之间的交互通信。

顺口“交连业业，信息协作“

#### ESB

ESB是一个实现系统间集成和互联互通的重要技术架构，基于内容的路由与过滤，具备复杂数据的传输能力，可提供一系列标准接口。

主要功能有：

1.应用程序透明性

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.安全支持

7.监控管理

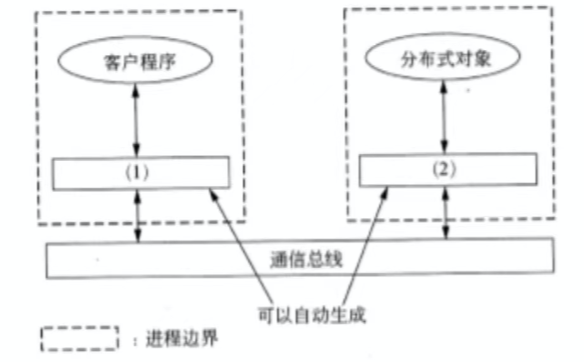
优势：

1.在部署方面。能够实现灵活部署。

2.在集成系统耦合度方面。待集成系统只需和总线联系，彼此之间不需要互相通信，降低系统耦合度。

3.在可扩展性方面。在加入新的待集成系统时，只需要采用插件的方式实现传输协议和数据格式的适配，系统可扩展性强。

#### 分布式对象通过ESB调用



（1）客户端桩 （2）服务端框架

一次远程调用的过程如下：

①客户程序将调用请求发送给客户端桩，对于客户程序来说，桩就是服务程序在客户端的代理。

②客户端桩负责将远程调用请求进行编组并发送给通信总线。

③调用请求经通信总线传送到服务端框架。

④服务端框架将调用请求解组并分派给真正的远程对象实现（服务程序）。

⑤服务程序完成客户端的调用请求，将结果返回给服务端框架。

⑥服务端框架将调用结果编组并发送给通信总线。

⑦调用结果经通信总线传送到客户端桩。

⑧客户端桩将调用结果解组并返回给客户程序，客户程序得到调用结果。

## 负载均衡

### 负载均衡分类

【一】应用层负载均衡

1、http重定向。HTTP重定向就是应用层的请求转发。用户的请求其实已经到了HTTP重定向负载均衡服务器，服务器根据算法要求用户重定向，用户收到重定向请求后，再次请求真正的集群。

特点:实现简单，但性能较差。

2、反向代理服务器。在用户的请求到达反向代理服务器时，由反向代理服务器根据算法转发到具体的服务器。常用的apache、nginx都可以充当反向代理服务器。

特点:部署简单，但代理服务器可能成为性能的瓶颈。

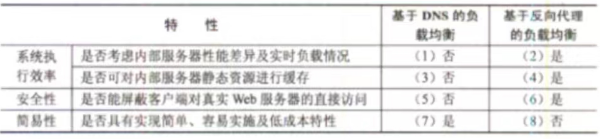
【二】传输层负载均衡

1、DNS域名解析负载均衡。DNS域名解析负载均衡就是在用户请求DNS服务器，获取域名对应的IP地址时，DNS服务器直接给出负载均衡后的服务器IP。

特点:效率比HTTP重定向高，减少维护负载均衡服务器成本。但一个应用服务器故障，不能及时通知DNS，而且DNS负载均衡的控制权在域名服务商那里，网站无法做更多的改善和更强大的管理。

2、基于NAT的负载均衡。基于NAT的负载均衡将一个外部IP地址映射为多个IP地址，对每次连接请求动态地转换为一个内部节点的地址。

特点:技术较为成熟，一般在网关位置，可以通过硬件实现。.像四层交换机一般就采用了这种技术。



### 负载均衡算法

静态算法(不考虑动态负载)

(1)轮转算法:轮流将服务请求(任务)调度给不同的节点(即:服务器)。

(2)加权轮转算法:考虑不同节点处理能力的差异。

(3)源地址哈希散列算法:根据请求的源IP地址，作为散列键从静态分配的散列表找出对应的节点。

(4)目标地址哈希散列算法:根据请求目标IP做散列找出对应节点。

(5)随机算法:随机分配，简单，但不可控。

动态算法(考虑动态负载)

(1)最小连接数算法:新请求分配给当前活动请求数量最少的节点，每个节点处理能力相同的情况下。

(2)加权最小连接数算法:考虑节点处理能力不同，按最小连接数分配。

(3)加权百分比算法:考虑了节点的利用率、硬盘速率、进程个数等，使用利用率来表现剩余处理能力。

硬件负载均衡:F5

软件负载均衡:LVS、Nginx、HAproxy

## Session共享

### 有状态和无状态

无状态服务(statelesS service)对单次请求的处理，不依赖其他请求，也就是说，处理一次请求所需的全部信息，要么都包含在这个请求里，要么可以从外部获取到(比如说数据库)，服务器本身不存储任何信息。

有状态服务(stateful service)则相反，它会在自身保存一些数据，先后的请求是有关联的。

判断以下构件是有状态服务还是无状态服务：

(1)Identification Bean(身份认证构件) 有

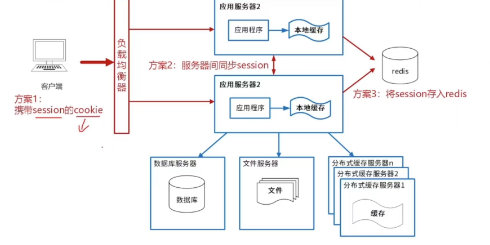
(2)ResPublish Bean(资源发布构件) 无

(3)ResRetrieval Bean(资源检索构件)无

(4)OnlineEdit Bean (在线编辑构件）有

(5)Statistics Bean(统计分析构件) 无

### Session共享



1、客户端携带session的cookie

2、服务器间同步session

3、将session存入redis

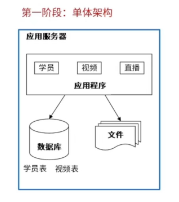
4、使用jwt等无状态登录

## CDN

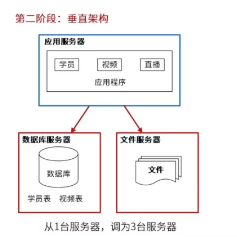
CDN的全称是Content Delivery Network,即内容分发网络。其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输得更快、更稳定。

## Web架构进化图[关注图，有可能填图]

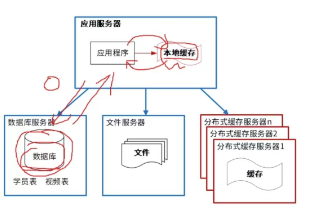
* 1. 单体结构



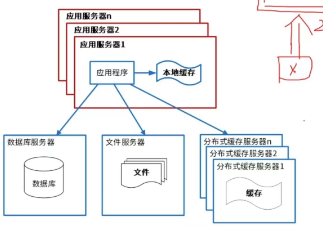
* 1. 垂直结构



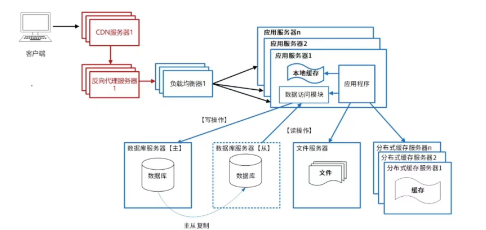
* 1. 增加缓存改善性能



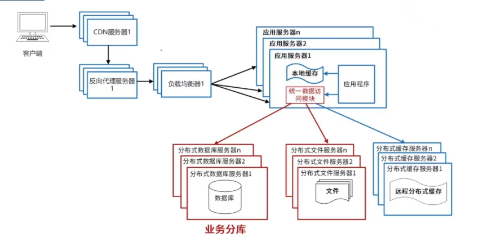
* 1. 使用服务集群改善并发能力



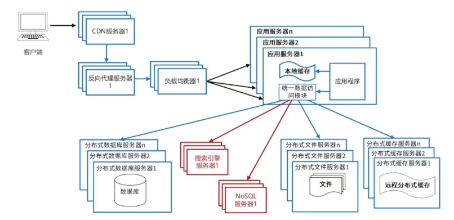
* 1. 使用反向代理和CDN加速响应



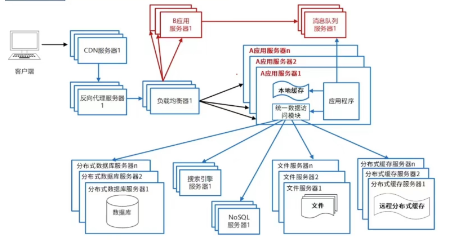
* 1. 分布式文件系统和分布式数据库系统



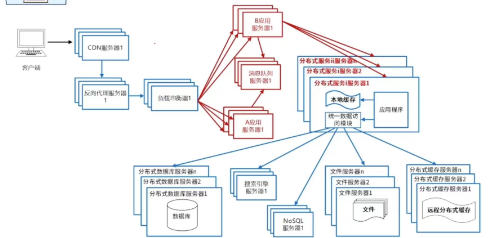
* 1. Nosql和搜索引擎



* 1. 业务拆分



* 1. 分布式服务



## 问答

### 第1问 非功能性需求

1.操作性需求。操作性需求指定了系统完成任务所需的操作环境及其可能的改变。

分类时看：**技术环境、系统集成、可移植性、可维护性**。

2.性能需求。性能需求的核心是性能问题,如响应时间、容量和可靠性。

分类时看：**速度需求、容量需求、可用性与可靠性需求**。有时候这部分和操作性有点分不开，可以用排除。

3.安全需求。安全性是防止信息系统崩溃和数据丢失的能力。

分类时看访问控制、加密与验证、病毒控制、数据完整等等，这个一般OK、

4.文化与政治需求。文化与政治需求是指针对使用系统的不同国家所特有的需求。

分类时看**多语种需求、用户定制需求**、未申明的术语、法律需求。注意用户定制需求是文化政治需求！

### 第2问 填图

暂无，看缘分，遇到拿不住的宁可选重复，保住一空。

### 第3问 随机

随机问题，看‘全.word’那个文档整理。

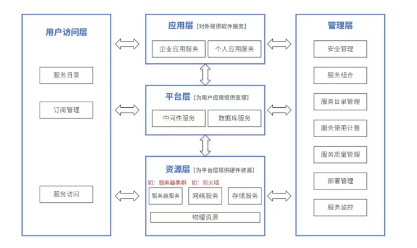
# 新技术/其他

## 云计算

【云计算】是集合了大量计算设备和资源，对用户屏蔽底层差异的分布式处理架构，其用户与提供实际服务的计算资源是相分离的。

【云计算架构】

* 1. 管理层提供对所有层次会计算服务的管理功能。
  2. 用户访问层：方便用户使用云计算服务所需的各种支撑服务，针对每个层次的云计算服务都需要提供相应的访问接口。
  3. 应用层:提供软件服务，如:财务管理，客户关系管理，商业智能。
  4. 平台层：为用户提供对资源层服务的封装，使用户可以构建自己的应用。
  5. 资源层：提供虚拟化的资源，从而隐藏物理资源的复杂性。如:服务器，存储。



【云计算优点】超大规模、虚拟化、高可靠性、高可伸缩性、按需服务、成本低(前期投入低、综合使用成本也低)

【按服务类型分类】

* 1. Saas软件即服务：基于多租户技术实现，直接提供应用程序
  2. Paas平台即服务：虚拟中间件服务器、运行环境和操作系统
  3. laaS基础设施即服务：包括服务器、存储和网络等服务

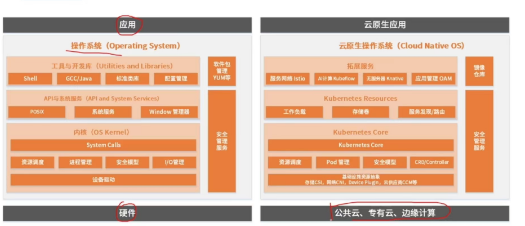
【按部署方式分类】

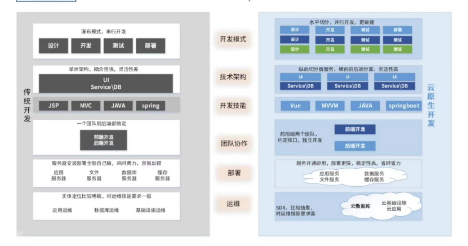
* 1. 公有云:面向互联网用户需求，通过开放网络提供云计算服务
  2. 私有云:面向企业内部提供云计算服务
  3. 混合云:兼顾以上两种情况的云计算服务

## 云原生

【云原生】是基于分布部署和统一运管的分布式云，以容器、微服务、DevOps等技术为基础建立的一套云技术产品体系。

【云原生架构模式】





1、服务化架构模式:典型代表【微服务】，服务拆分使维护压力大增。

2、Mesh化架构模式:把中间件框架(RPC、缓存、异步消息)从业务进程中分离，由Mesh进程完成。

3、Serverless模式:非常适合于事件驱动的数据计算任务。

4、存储计算分离模式:各类暂态数据(如session)用云服务保存。

5、分布式事务模式:解决微服务模式中多数据源事务问题。

6、可观测架构:包括乱Logging、Tracing、Metrics三个方面。

7、事件驱动架构:本质上是一种应用/组件间的集成架构模式。

【云原生架构设计原则】

1. 服务化原则:使用微服务
2. 弹性原则:可根据业务变化自动伸缩
3. 可观测原则:通过日志、链路跟踪和度量
4. 韧性原则:面对异常的抵御能力
5. 所有过程自动化原则:自动化交付工具
6. 零信任原则:默认不信任网络内部和外部的任何人/设备/系统
7. 架构持续演进原则:业务高速迭代情况下的架构与业务平衡

【云原生架构反模式】

1、庞大的单体应用。需要多人开发的业务模块，考虑通过服务化进行拆分，并让组织与架构匹配。

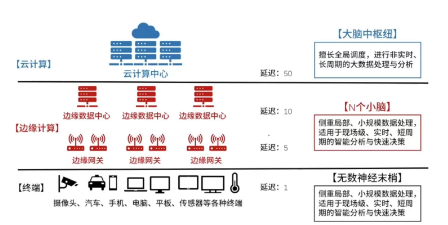
2、单体应用“硬拆”为微服务(服务拆分要适度) 。小规模软件的服务拆分(为拆而拆)、数据依赖(服务间数据依赖)、性能降低

3、缺乏自动化能力的微服务。手动维护大量微服务是不现实的。

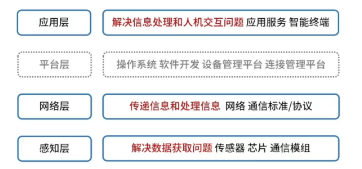
## 边缘计算

是指在靠近物或数据源头的一侧，采用网络、计算、存储、应用核心能力为一体的开放平台，就近提供最近端服务。

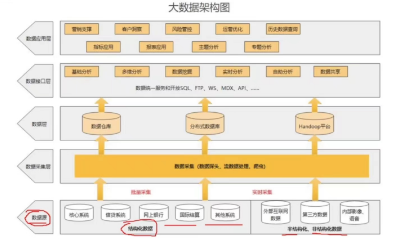
【边缘计算的本质】计算处理职能的本地化。



## 物联网



## 大数据



# 安全

## 五大服务

GB/ T 9387.2—1995中定义了5大类安全服务,分别是认证服务、访问控制服务、数据机密性服务、数据完整性服务和抗抵赖性服务。

### 认证服务

认证服务的基本目的,是为了防止其他实体占用和独立操作被认证实体的身份。认证服务提供了实体声称其身份的保证,只有在主题和验证者的关系背景下,认证才有意义。认证服务的主要实现方式包括以下5种:

1. 已有的信息,如认证口令。
2. 拥有的信息,如 IC 卡、令牌等。
3. 不可改变的特性,如指纹、虹膜等生物特征
4. 相信可靠的第三方建立的认证。
5. 环境,如主机地址等。

关于PKI公钥认证的思考：

认证是第一道屏障，他的目的是判断用户身份是否真实。认证之后，才会根据用户身份对资源访问做权限控制（访问控制，RBAC之类的）。这是两个部分，我们常讨论的PKI其实都是认证服务，只不过在某些特殊情况下，可以实现特殊业务的权限控制。

PKI公钥认证就是第四种方式“相信第三方的认证”，因此不像口令认证一样有显式的调用登录模块，PKI公钥没有显式去登录认证接口，而是直接开始交互，遵循CA的规范，把认证交给第三方，**通过“公钥加密认证、私钥解密签名”的方式**信赖PKI平台。

特殊情况下，PKI可基于用户私钥对私有数据进行加密保护，实现用户私人数据的权限访问限制。

### 访问控制服务

访问控制服务决定开放环境中允许使用哪些资源、在什么地方适合组织为授权访问的过程。在访问控制实例中,访问可以是对一个系统或对系统内部进行。常见的访问控制服务的实现方式包括以下三种方式:

1. 自主访问控制( DAC )。自主访问控制是一种接入控制服务,通过执行基于系统实体身份及其到系统资源的接入授权。包括在文件、文件夹和共享资源中设置许可。用户有权对自身所创建的文件、数据表等访问对象进行访问,并可将其访问权授予其他用户或收回其访问权限。允许访问对象制定针对该对象访问的控制策略,通常可通过切问控制列表来限定针对客体可执行的操作。
2. 强制访问控制( MAC )。强制访问控制是系统强制主体服从访问控制策略,龙由系统对用户所创建的对象,按照规定的规则控制用户权限及操作对象的访问。其主英特征是对所有主体及其所控制的进程、文件、段、设备等客体实施强制访问控制。
3. 基于角色访问控制( RBAC )。基于角色访问控制主要通过对角色的访问进行控制，使权限与角色相关联，用户通过成为适当角色的成员而得到其角色的权限。用户可依其责任和资格分派相应的角色，角色可依新需求和系统合并赋予新权限，而权限也可根据需要从某角色中收回。

### 数据机密性服务

数据机密性服务的目的是确保信息仅仅是对被授权者可用,它良的保护可以通过确保数据被限制于授权者获得,或通过特定方式表示数据来获得。官息的机密性主要通过以下两种方式实现：

1. 通过禁止访问提供机密性,即可以通过访问控制,以及通过物理媒体保护和路由选择控制保证机密性。
2. 通过加密提供机密性,即防止数据泄漏在传输或存储中。加密机制包括基于对称的加密机制和基于非对称的加密机制。
3. 除此以外,还可以通过数据填充、通过虚假事件(例如隐藏在不可信链路上交换的信息流总量)、通过保护 PDU 头和通过时间可变域提供机密性。

### 数据完整性服务

数据完整性服务的目的是通过阻止威胁或探测威胁,保护可能遭到不同方式危害的数据完整性和数据相关属性完整性,即保证数据不以未经授权方式进行改变或损毁。数据完整性的常见实现方式包括:

1. 阻止对数据传输媒介访问的机制。包括物理隔离、不受干扰的信道,路由控制,访问控制等。
2. 探测对数据非授权修改的机制。包括密封、数字签名、数据重复、与密码变换相结合的数字指纹和消息序列号等。

### 抗抵赖服务

抗抵赖服务是提供有关特定事件或行为的证据,包括证据的生成、验证和记录,以及在解决纠纷时随即进行的证据恢复和再次验证。抗抵赖性服务的实现方式主要包括数字签名、用户认证、操作日志等技术。

## 安全威胁

1. 物理安全威胁：指对系统所用设备的威胁，如自然灾害、电源故障、数据库故障、设备被盗等造成数据丢失或信息泄漏。
2. 通信链路安全威胁：指在传输路线上安装窃听器装置或对通信链路进行干扰。
3. 网络安全威胁：指由于因特网的开放性、国际性、无安全管理性，对内部网络形成的严重安全威胁。
4. 操作系统安全威胁：指操作系统本身的后门或安全陷阱，如木马等。
5. 应用系统安全威胁：指网络服务或用户业务系统安全的威胁，包括应用系统自身漏洞，也受到木马的威胁。
6. 管理安全威胁：指人员管理和各自安全管理制度。

## 信息摘要

散列函数是一种公开的数学函数。散列函数运算的输入信息也可叫作报文。散列函数运算后所得到的结果叫作散列码或者叫作消息摘要。

散列函数可以用来检测报文的可靠性。接收者对收到的报文用与发送者相同的散列函数进行运算，**如果得到与发送者相同的散列码，则可以认为报文没有被篡改**，否则，报文就是不可信的。散列函数具有如下一些特点:

1、不同内容的报文具有不同的散列码，而一旦原始报文有任何改变，哪怕改变一位 信息，则通过散列函数计算后得到的散列码也将完全不同。这样，这个散列码就好比是这个 报文所特有的“指纹”。

2、散列函数是单向的，即求解某一个报文的散列码非常容易，但是根据散列码来倒 推原始报文是非常困难的。

3、对于任何一个报文，无法预知它的散列码。

4、散列码具有固定的长度，不管原始报文的长度如何，通过散列函数运算后的散列 码都具有一样的长度。

常用的信息摘要算法：MD5、SHA等：MD5：市场上广泛使用；SHA：散列值为128位和160位，密钥长度较长，安全性高于MD5

**消息摘要的目的是为了防止发送的消息被篡改。 对摘要进行加密的目的是防止抵赖。**

## 数字签名

对于计算机系统中传送、存储的重要文件、数据、信息等，一般需要有某种方式来确认其真实性，即接收者能够确认自己得到的信息确实是由该信息所声称的发送者发出的，而不是由非法入侵者伪造、冒充发出的，并且还要能够保证信息在传送、存储中没有被恶意篡改， 这样这份信息才能真实地反映发送方的意图。另外，对于发送方来说，如果发出一份信息， 还必须有一定的措施阻止其否认自己发出信息的行为，即不可否认性，由此引出数字签名技术。

数字签名主要由两个算法组成:**签名算法和验证算法**。

1、通过使用签名算法签名一个消息，所得到的签名能够通过一个验证算法来验证签名的真实性和有效性。 所以数字签名技术的大致过程就是:信息的发送方对信息利用**自己的私钥**进行签名，接着发送方把这个签名和信息一起发送给接收方。

2、接收方收到信息后利用发送方的公钥来对其中的数字签名进行验证，确认其合法性。

## 数字信封

数字信封的过程：

1、发送方将原文用对称密钥加密，将对称密钥以非对称加密的方式发送

2、接收方接收到数字信封，用自己的私钥解密，取出对称密钥解密。

## 加密

对称加密，加密速度快，但密钥强度不高、密钥分发困难，适用于大内容传输：DES、3DES、AES、IDEA、RC-5等

**大数据下，流加密效率更高，性能更好**。

非对称加密 A公钥加密只能用A私钥解密，A私钥加密也只有A公钥解密 加密速度慢，不适用大内容传输：RSA、ECC、Elgamal等

## 数据库安全

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。

# 例题

## 架构评估

【1】面向服务vs面向资源

案例题某公司拟开发一个市场策略跟踪与分析系统，根据互联网上用户对公司产品信息的访问情况和产品实际销售情况来追踪各种市场策略的效果。其中互联网上用户对公司产品信息的访问情况需要借助两种不同的第三方Web分析软件进行数据采集与统计，并生成不同格式的数据报表；公司产品的实际销售情况则需要通过各个分公司的产品销售电子表格或数据库进行采集与汇总。得到相关数据后，还要对数据进行分析与统计，并通过浏览器以在线的方式向市场策略制定者展示最终的市场策略效果。

在对市场策略跟踪与分析系统的架构进行设计时，公司的架构师王工提出采用面向服务的系统架构，首先将各种待集成的第三方软件和异构数据源统一进行包装，然后将数据访问功能以标准Web服务接口的形式对外暴露，从而支持系统进行数据的分析与处理，前端则采用CSS等技术实现浏览器数据的渲染与展示。架构师李工则认为该系统的核心在于数据的定位、汇聚与转换，更适合采用面向资源的架构，即首先为每种数据元素确定地址，然后将各种数据格式统一转换为JSON格式，通过对JSON数据的组合支持数据的分析与处理任务，处理结果经过渲染后在浏览器的环境中进行展示。在架构评估会议上，专家对这两种方案进行综合评价，最终采用了李工的方案。

【问题1】请根据题干描述，对市场策略跟踪与分析系统的数据源特征与数据操作方式进行分析，完成表4-1中的(1)~(3),并用200字以内的文字说明李工方案的优点。



(1)数据报表非实时电子表格与数据库2)非实时(3)只读产品

通过对系统的数据源特征和数据操作方式进行分析可以看出，待集成的数据均为持久型数据(文件或数据库)，系统对数据的访问均为只读非实时性的。针对上述应用特征，李工提出的面向资源的架构方式以对数据资源的只读访问为核心，通过数据唯一标识直接对各种数据进行访问与获取，系统架构清晰、实现简单、效率较高。

【问题2】请从数据获取方式、数据交互方式和数据访问的上下文无关性三个方面对王工和李工的方案进行比较，并用500字以内的文字说明为什么没有采用王工的方案。

1、从数据获取方式看，王工的方案需要将现有的多个系统和异构的数据源包装为服务，采用Web服务暴露数据接口，客户端需要通过服务调用获取数据，这种方法工作量大，复杂度较高。李工的方案则绕开了复杂的功能封装，只需要明确数据的位置与标识，通过特定的网络协议直接使用标识定位并获取数据，与王工的方案相比工作量小，实现简单。

2、从数据交互方式看，王工的方案采用远程过程调用和异步XML消息等模式实现数据交互，这种方式适合于系统之间功能调用时进行的少量数据传输，而在进行单纯的数据访问时效率不高，稳定性也较差。李工的方案则以数据资源为核心，在对数据资源进行标识的基础上，通过标识符直接对数据资源进行访问与交互，实现简单且效率较高。

3、从数据访问的上下文无关性看，王工的方案中数据访问是与上下文有关的，具体表现在每次客户端进行数据请求都需要附加唯一的请求标识，并且服务端需要区分不同的客户端请求，效率较低。李工的方案中数据访问是与上下文无关的，客户端通过全局唯一的统一资源标识符(URI)请求对应的数据资源，服务端不需要区分不同的客户端请求。

【2】管道-过滤器vs数据存储为中心

某软件公司为其新推出的字处理软件设计了一种脚本语言，专门用于开发该字处理软件的附加功能插件。为了提高该语言的编程效率，公司组织软件工具开发部门为脚本语言研制一套集成开发环境。软件工具开发部门根据字处理软件的特点，对集成开发环境进行了需求分析，总结出以下3项核心需求：

（1）集成开发环境需要提供对脚本语言的编辑、语法检查、解释、执行和调试等功能的支持，并要实现各种功能的灵活组合、配置与替换。

（2）集成开发环境需要提供一组可视化的编程界面，用户通过对界面元素拖曳和代码填充的方式就可以完成功能插件核心业务流程的编写与组织。

（3）在代码调试功能方面，集成开发环境需要实现在脚本语言编辑界面中的代码自动定位功能。具体来说，在调试过程中，编辑界面需要响应调试断点命中事件，并自动跳转到当前断点处所对应的代码。

针对上述需求，软件工具开发部门对集成开发环境的架构进行分析与设计，王工认为该集成开发环境应该采用管道-过滤器的架构风格实现，李工则认为该集成开发环境应该采用以数据存储为中心的架构风格来实现。公司组织专家对王工和李工的方案进行了评审，最终采用了李工的方案。

【问题1】

请用200字以内的文字解释什么是软件架构风格，并从集成开发环境与用户的交互方式、集成开发环境的扩展性、集成开发环境的数据管理三个方面说明为什么最终采用了李工的设计方案。

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。

1、从集成开发环境与用户的交互方式看，用户通常采用交互式的方式对脚本语言进行编辑、解释执行与调试。在这种情况下，采用以数据存储为中心的架构风格能够很好地支持交互式数据处理，而管道-过滤器架构风格则对用户的交互式数据处理支持有限。

2、从集成开发环境的扩展性来看，系统核心需求要求实现各种编辑、语法检查、解释执行等多种功能的灵活组织、配置与替换。在这种情况下，采用以数据存储为中心的架构风格，以数据格式解耦各种功能之间的依赖关系，并可以灵活定义功能之间的逻辑顺序。管道-过滤器架构风格同样以数据格式解耦数据处理过程之间的依赖关系，但其在数据处理逻辑关系的灵活定义方面较差。

3、从集成开发环境的数据管理来看，集成开发环境需要支持脚本语言、语法树（用于检查语法错误）、可视化模型、调试信息等多种数据类型，并需要支持数据格式的转换。以数据存储为中心的架构将数据存储在统一的中心存储器中，中心存储器能够表示多种数据格式，并能够为数据格式转换提供各种支持。管道-过滤器架构风格通常只能支持有限度的数据格式，并且在数据格式转换方面的灵活性较差。

【问题2】

在对软件系统架构进行设计时，要对架构需求进行分析，针对特定需求选择最为合适的架构风格，因此实际的软件系统通常会混合多种软件架构风格。请对核心需求进行分析，说明为了满足需求（2）和（3），分别应采用何种架构风格，并概要说明采用相应架构风格后的架构设计过程。

为了满足需求2，应该采用解释器架构风格。具体来说，需要：①为可视化编程元素及其拖拽关系定义某种语言，并描述其语法与语义；②编写解释器对该语言进行解释；③生成对应的脚本语言程序。

为了满足需求3，应该采用隐式调用架构风格。具体来说，首先需要定义“断点在调试过程中命中”这一事件，并实现当断点命中后的屏幕定位函数。集成开发环境维护一个事件注册表结构，将该事件与屏幕定位函数关联起来形成注册表中的一个记录项。在调试过程中，集成开发环境负责监听各种事件，当“断点在调试过程中命中”这一事件发生时，集成开发环境查找事件注册表，找到并调用屏幕定位函数，从而实现脚本语言编辑界面与调试代码的自动定位。

【3】MVC架构风格

某商业银行欲开发一套个人银行系统，为用户提供常见的金融服务，包括转账、查询、存款变更和个人信息管理等功能。该软件除了业务需求外，还有一些特殊的表现层需求：

（1）根据用户级别的不同，界面和可用功能是不同的；

（2）支持Web、Windows、手机App等多种不同类型的界面；

（3）考虑到将来功能的扩展，需要系统支持界面的定制以及动态生成等功能，以降低系统维护和新功能发布的成本。

经过对需求的讨论，该银行初步决定采用MVC模式设计该个人银行系统的表现层，采用XML作为GUI的描述语言，并应用XML的界面管理技术来实现灵活的界面配置、界面动态生成和界面定制。

【问题1】采用MVC模式对界面设计的作用。

MVC模式作用：

1．允许多种界面的扩展，视图的变更与增加，与模型无关；

2．易于维护，控制器和视图随着模型的扩展而扩展，只要保持公共接口，控制器和视图的旧版本可以继续使用；

3．可支持功能强大的用户界面。

【问题2】请从设计模式的角度，简要说明设计方案采用XML作为GUI描述语言的机制。

从设计模式的角度来说，整个XML**表现层解析的机制是一种策略模式**。

在调用显示GUI时，不是直接调用特定的表现技术的API，而是装载GUI对应的XML配置文件，然后根据特定的表现技术的解析器解析XML，得到GUI视图实例对象。这样，对于GUI开发人员来说，GUI视图只需要维护一套XML文件即可。

## 数据库与缓存

【1】关系型数据库和NoSQL

某软件公司欲开发一个基于Web 2.0的大型社交网络系统。就该系统的数据架构而言，李工决定采用公司熟悉的数据架构，使用通用的商用关系型数据库，系统内部数据 采用中央集中方式存储。该系统投入使用后，初期用户数量少，系统运行平稳。6个月后，用户数出现了爆炸式增长，系统暴露出诸多问题，集中表现在：

（1）用户执行读写操作时，响应时间均变得很慢；

（2）随着系统功能的扩充，原有数据格式发生变化，又出现新的数据格式，维护困难；

（3）数据容量很快超过系统原有的设计上限，数据库扩容困难；

（4）软件系统不断出现宕机，整个系统可用性较差。

经过多次会议讨论，公司的王工建议采用NoSQL数据库来替代关系数据库，以解决上述问题。但李工指出NoSQL数据库出现时间不长，在使用上可能存在风险。公司技术人员对NoSQL数据库产品进行了认真测试，最终决定采用NoSQL数据库来替代现有的数据库系统。

【问题1】

分别解释产生问题（1）～（4）的原因。

1．用户响应时间慢。大型社交网络系统要根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用动态页面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。**关系数据库应付上万次SQL查询还勉强可以，但是应付上万次SQL写数据请求，硬盘I／O就已经无法承受了。特别是涉及到多表连接操作，会导致响应变慢**。

2．数据格式变化。大型社交网络系统随着用户的使用，会不断地增加新的功能，导致原有数据格式发生变化，甚至出现新的数据格式。但**关系数据库中采用元组方式组织数据，难以使用新型数据格式，难以维护**。

3．数据容量超过设计上限。对于大型社交网络系统，往往会在很短时间内产生海量数据。**关系数据库多采用中央数据存储，使得数据容量受限于前期设计的上限，很难实现数据容量的横向扩展**。

4．系统可用性差：**关系数据库采用中央数据存储，容易成为系统的性能瓶颈，单点故障**很容易导致系统崩溃，负载过高往往导致系统出现宕机现象。

【问题2】

请针对问题（1）～（4），分别指出NoSQL数据库的哪些特点促使公司最终采用了NoSQL数据库。

针对问题1，NoSQL数据库支持高并发数据访问，性能较高。

针对问题2，NoSQL数据库的数据存储结构松散，能够灵活支持多种类型的数据格式。

针对问题3，NoSQL数据库能够支持海量数据的存储，且易于横向扩展。

针对问题4，NoSQL数据库基于分布式数据存储，不存在单点故障和性能瓶颈，系统可用性高。

【问题3】

请指出该系统采用NoSQL数据库时可能存在的问题。

该系统采用NoSQL数据库时可能存在的问题有：

1．NoSQL数据库的现有产品不够成熟，大多数产品处于初创期。

2．NoSQL数据库并未形成一定的标准，产品种类繁多，缺乏官方支持。

3．NoSQL数据库不提供对SQL的支持，学习和应用迁移成本较高。

4．NoSQL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。

## 安全

【1】不可抵赖

要求邮件以加密方式传输，邮件最大附件内容为500MB，发送者不可抵赖，若邮件被第三方截获，第三方无法篡改。设计邮件加密系统。

邮件以加密方式传输，且附件内容500MB ------> 大数据量的加密方式，正文要用对称加密方式、密钥以非对称加密方式（数字信封技术）

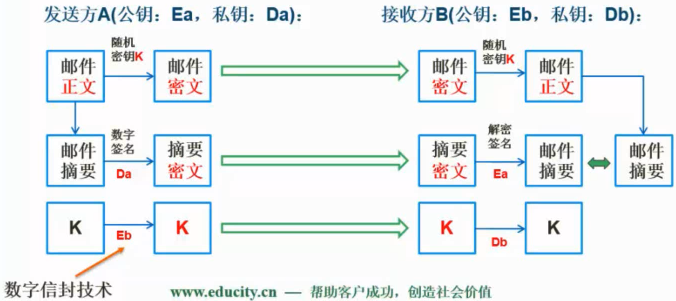
发送者不可抵赖 -------> 数字签名（常用摘要+加密作为一种签名算法）

无法篡改 -------> 信息摘要技术

**答案：**

**1、邮件正文通过信息摘要技术得到邮件摘要，发送方通过自己的私钥加密邮件摘要，并将摘要密文发送；**

**2、接收方解密后将邮件摘要和邮件正文进行验证，验证是否报文篡改 (数字签名 + 信息摘要技术混合)**



【2】公钥认证和数据库安全

某软件公司拟开发一套信息安全支撑平台，为客户的局域网业务环境提供信息安全保护。该支撑平台的主要需求如下：

（1）为局域网业务环境提供用户身份鉴别与资源访问授权功能；

（2）为局域网环境中交换的网络数据提供加密保护；

（3）为服务器和终端机存储的敏感持久数据提供加密保护；

（4）保护的主要实体对象包括局域网内交换的网络数据包、文件服务器中的敏感数据文件、数据库服务器中的敏感关系数据和终端机用户存储的敏感数据文件；

（5）服务器中存储的敏感数据按安全管理员配置的权限访问；

（6）业务系统生成的单个敏感数据文件可能会达到数百兆的规模；

（7）终端机用户存储的敏感数据为用户私有；

（8）局域网业务环境的总用户数在100人以内。

【问题1】

在确定该支撑平台所采用的用户身份鉴别机制时，王工提出采用基于口令的简单认证机制，而李工则提出采用基于公钥体系的认证机制。项目组经过讨论，确定采用基于公钥体系的机制，请结合上述需求具体分析采用李工方案的原因。

1．基于口令的认证方式实现简单，但由于口令复杂度及管理方面的原因，易受到认证攻击；而在基于公钥体系的认证方式中，由于其密钥机制的复杂性，同时在认证过程中私钥不在网络上传输，因此可以有效防止认证攻击，与基于口令的认证方式相比更为安全。

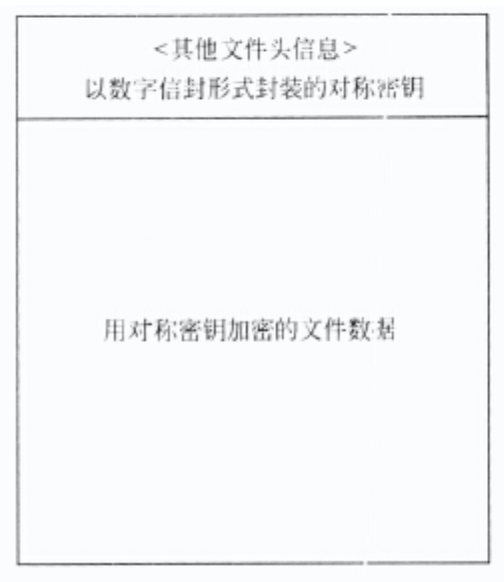
2．按照需求描述，**在完成用户身份鉴别后，需依据用户身份进一步对业务数据进行安全保护，且受保护数据中包含用户私有的终端机数据文件**，在基于口令的认证方式中，用户口令为用户和认证服务器共享，没有用户独有的直接秘密信息，而在基于公钥的认证方式中，可基于用户私钥对私有数据进行加密保护，实现更加简便。

3．基于公钥体系的认证方式协议和计算更加复杂，因此其计算复杂度要高？基于口令的认证方式，但业务环境的总用户数在100人以内，用户规模不大，运行环境又为局域网环境，因此基于公钥体系的认证方式可满足平台效率要求。

【问题2】

针对需求（7），项目组经过讨论，确定了基于数字信封的加密方式，其加密后的文件结构如图5-1所示。

请结合需求说明对文件数据进行加密时，应采用对称加密的块加密方式还是流加密方式，为什么？并对该机制中的数据加密与解密过程进行描述。



应采用流加密方式。因为需求中提及“单个敏感数据文件可能会达到数百兆的规模”，文件数据量较大，使用流加密方式可以获得更高的加解密效率。

数据加密与解密过程如下：

其加密过程为：首先生成一个对称密钥，使用用户公钥加密这个对称密钥后存储在文件头，然后用生成的对称密钥加密文件数据存储；

其解密过程为：用户首先使用自己的私钥解密被加密的对称密钥，再用该对称密钥解密出数据原文。

【问题3】

对数据库服务器中的敏感关系数据进行加密保护时，客户业务系统中的敏感关系数据主要是特定数据库表中的敏感字段值，客户要求对不同程度的敏感字段采用不同强度的密钥进行防护，且加密方式应尽可能减少安全管理与应用程序的负担。目前数据库管理系统提供的基本数据加密方式主要包括加解密API和透明加密两种，请用300字以内的文字对这两种方式进行解释，并结合需求说明应采用哪种加密方式。

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。