# 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**。
3. 常见架构风格对比这种题，给的对比语言都不一样，还是要从原理上去想，不会写的，可以从**“怎么工作的”、“加入新的功能怎么办”、“运行时能不能直接调用、数据需不需要传输”**三个方面考虑。
4. 不要害怕，对比着写就好。

## 质量属性效应树

## 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。

# 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

### 1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

### 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

### 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。

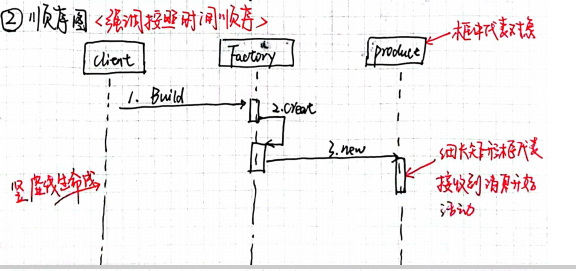
（经过一段时间引发某个时间--->参与者为时间）

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）。

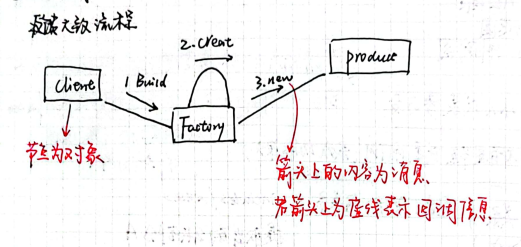
### UML图样例（可能会填图）

#### 顺序图



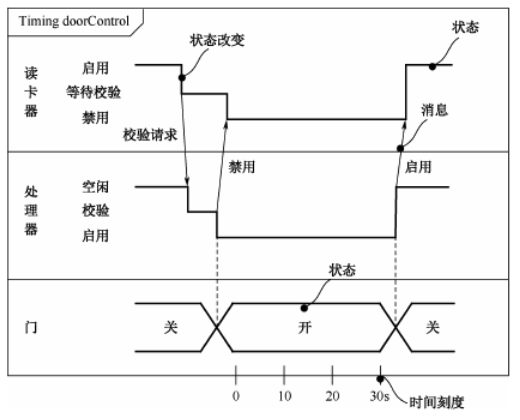
强调按照时间顺序

#### 通信图



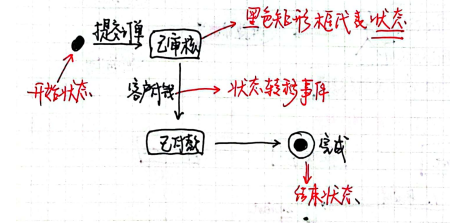
#### 定时图

强调实际时间



#### 状态图

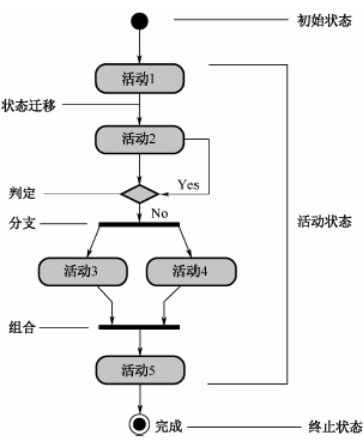
用于描述状态转换和复杂对象



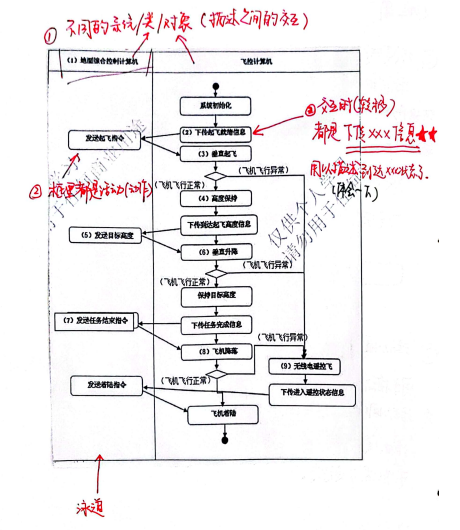
#### 活动图

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 | xxxxx | 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，管理数据 |
| 应用扩展 | 独立于系统应用，数据库接口标准化，易于在应用间共享 | 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 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### 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。

### 分布式缓存

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。

### 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

2、哈希分片

3、一致性哈希分片

### Redis常见问题

#### 缓存雪崩

#### 缓存穿透

#### 缓存预热

#### 缓存更新

#### 缓存降级

### 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（point-in-time snapshot）。

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 |
| 恢复性能 | 恢复大数据集的时候会更快 | 数据恢复比较慢 |
| 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 | 安全性更高 |
| 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缓存 VS Redis缓存

Memcache:

1、数据类型：key-value

2、不支持持久化

3、分布式存储方式为客户端哈希分片、一致性哈希

4、支持多线程

5、支持内存管理，有私有内存池

6、不支持事务

Redis：

1、数据类型有key-value，list、set、hash等多种数据结构

2、支持持久化

3、多种分布式存储方式，主从、哨兵、集群

4、不支持多线程

5、无内存管理

6、有限支持事务

### 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

## 层次型软件架构

### 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。

### MVP

### MVVM

### 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.程序代码重用性强，即便更改数据库，只需修改配置文件，不必重写代码程序

2.业务逻辑代码可读性强，不会有大量SQL

3.持久化技术可自动优化，减少对数据库访问，提高效率

4.通过OR映射，向业务逻辑提供面向对象的访问

5.简化开发工作，关注于业务逻辑开发

#### 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

#### 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

#### 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

## Rest架构风格

### 定义

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

## RIA富互联网架构风格

## 微服务和SOA

### 微服务

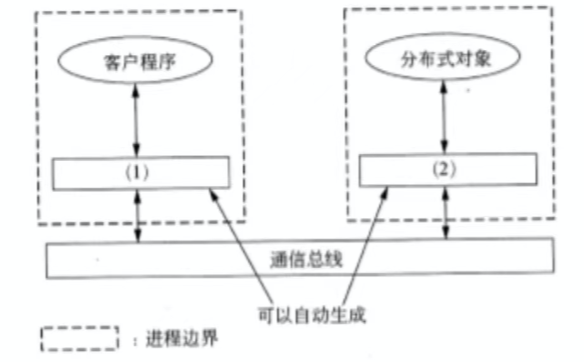
### 微服务架构模式方案

### 微服务设计约束

### 微服务和SOA区别

### SOA

### 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

### 负载均衡算法

## Session共享

### 有状态和无状态

### Session共享

## CDN

## Web架构进化图[关注图，有可能填图]

## 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

## 云原生

## 边缘计算

## 物联网

# 安全（见单独的印象笔记//暂时未整理过来）