# Chapter1

### 1. 如何将设备组装起来形成完整的计算机硬件系统

(1) 基本的**硬件模块**: 控制器, 运算器, 内存储器, 外存储器, 输入设备, 输出设备,... (2) 硬件模块之间的**连接关系**: 总线

### 2. 架构的共性

(1) 一组基本的构成要素——构件 (2) 这些要素之间的连接关系——连接件 (3) 这些要素连接之后形成的拓扑结构——物理分布 (4) 作用于这些要素或连接关系上的限制条件——约束 (5) 质量——性能

### 3. 软件架构的构建和连接件

**$$** **构件**: 各种基本的软件构造模块 (函数, 对象, 模式等);**$$** **连接件**: 将它们组合起来形成完整的软件系统;

### 4. 软件架构的定义

(1) (**1994**) SA={components, connectors, constrains} (2) (**2000**) Architecture={component, connector, environment, principle}. 架构是以构件, 构件之间的关系, 构件与环境之间的关系以及指导系统设计与演化的原理为内容的某一系统的基本组织结构 (3) (**1992**) SA={elements, form, rational} 软件架构是由一组具有一定形式的元素构成: 这组元素分成 3 类: 处理元素, 数据元素和连接元素; 处理元素负责对数据进行加工, 数据元素是被加工的信息, 连接元素把架构的不同部分组合连接起来。软件架构形式是由专有属性和关系组成。专有属性用于限制软件架构元素的选择。关系用于限制软件架构元素组合的拓扑结构。在多个架构方案中选择合适的架构方案往往基于一组准则 (4) (**2010**) SA={elements, relations, properties} (5) (**归纳**) 提供了一个结构, 行为和属性的高级抽象从一个较高的层次来考虑组成系统的构件, 构件之间的连接, 以及由构件与构件交互形成的拓扑结构这些要素应该满足一定的限制, 遵循一定的设计规则, 能够在一定的环境下进行演化反映系统开发中具有重要影响的设计决策, 便于各种人员的交流, 反映多种关注, 据此开发的系统能完成系统既定的功能和性能需求。架构 = 构件 + 连接件 + 拓扑结构 + 约束 + 质量

### 5. 程序规模和软件架构的关系

随着软件系统规模越来越大, 越来越复杂用户需求 (功能性) 越来越复杂, 变化越来越频繁; 用户软件质量 (非功能性) 的要求也越来越高; 如何将成百上千个功能组合起来, 同时满足用户质量需求, 变得越来越困难。此时, 整个系统的全局结构和设计显得越来越重要。很多质量需求主要体现在架构中而不是功能模块内部的实现中。**结论**: 对于大规模的复杂软件系统来说, 对系统全局结构的设计比起对算法的选择和数据结构的设计明显重要得多。

### 6. 软件架构的关注点

(1) 如何将复杂的软件系统划分为模块 (2) 如何规范模块的构成 (3) 如何将这些模块组织为完整的系统 (4) 以及保证系统的质量要求

### 7. 软件架构的目标

**$$** **主要目标**: 建立一个一致的系统及其视图集, 并表达为最终用户和软件设计者需要的结构形式, 支持用户和设计者之间的交流与理解。**$$** **外向目标**: 建立满足最终用户要求的系统需求;**$$** **内向目标**: 建立满足系统设计者需要以及易于系统实现, 维护和扩展的系统构件构成。

### 8. 软件架构的作用

(1) 交流的手段: 在软件设计者, 最终用户之间方便的交流; (2) 可传递的, 可复用的模型: 可重复利用的, 可转移的系统抽象 (用到其它的项目) (提高大规模重复利用率) (3) 关键决策的体现 (折中) (性能与安全性) (可维护性与可靠性) (当前开发费用和未来开发代价)

### 9. 软件架构的重要意义

(1) SA 是软件开发过程初期的产品, 在开发的早期阶段就考虑系统的正确设计与方案选择, 为以后开发, 测试, 维护各个阶段提供了保证; (2) 与其他后期的设计活动相比, SA 设计的成本和代价要低得多; (3) 正确有效的 SA 设计会给软件开发带来极大的便利; (4) 在大型软件系统中, 质量属性更多的是由系统结构和功能划分来实现的, 而不再仅仅依靠所选择的算法或数据结构。

### 10. 软件架构在软件生命周期中的作用

(1) **项目规划**: 考虑项目的规模, 复杂度, 可行性等; (2) **需求分析**: 利用 SA, 支持用户, 项目负责人, 系统架构师, 程序员, 测试人员之间进行交流和协商; 从不同视角审查备选的 SA, 对得出的意见进行综合, 找出合理的平衡方案; 从用户角度考虑未来的需求会发生什么变化, 并使 SA 能够提前支持这些变化; (3) **软件设计**: 参考经典 SA 风格, 设计系统架构模型, 推敲其存在的缺陷和替代方案并进行评估; 进而逐步细化 SA, 并对定型后的 SA 作文档化工作; (4) **软件实现**: 各开发团队按照 SA 规定的 “构件及其之间的相互关系” 进行开发, 保证最终得到的系统与最初的 SA 一致; (5) **测试与评审**: 根据 SA 的约束条件, 对软件的质量属性进行测试; (6) **维护与升级**: 把 SA 文档作为维护和升级的重要依据。

### 11. 软件架构和其他软件设计活动的比较

(1) **起点模糊**。在用户需求尚没有明确定义, 用户对其所期望的系统功能, 行为和性能尚未确定的情况下就要进行 SA 设计; ——故而需要交流与反复; (2) **高抽象层次**。SA 分析处理的是高层次的系统构件或子系统之间的关系, 而非变量, 函数等低层次的概念; (3) **分布决策**。来自用户, 架构师, 测试人员等多方面; (4) **贯穿全局**。SA 设计活动在项目开始时进行, 但其作用则贯穿整个项目周期, 越往后越显出重要性。

### 12. 什么是架构师的关注点以及软件质量定义

**软件质量**: 软件与明确的和隐含的需求相一致的程度

### 13. 质量和质量的关系

如果提高一个质量, 经常会影响（提高或降低）其它质量

### 14. 软件质量因素及对策

(1) **性能**: 软件的 “时间**$$**空间” 效率; 集群, 缓存, 异步 (2) **安全性**: 在对合法用户提供服务的同时, 阻止未授权用户的使用企图; 防火墙, 入侵检测, 加密, 单一入口 (3) **易用性**: 用户使用软件的容易程度, 用户容易使用和学习; 统一风格界面, 定制 (4) **重用性**: 减少代码编写, 提高团队开发效率, 降低维护代价; 构件, 框架, 软件产品线 (5) **健壮性**/**可用性**: 在异常情况下, 软件能够正常运行的能力（容错, 恢复）集群, 灾备, 故障转移 (6) **可修改性**: 软件适应 “变化” 的能力, 系统很容易被修改从而适应新的需求或采用新的算法, 数据结构的能力; 稳定的功能分解, 设计模式, 分层, 解耦 (7) **可测试性**: 软件是否易于被测试。一致的错误处理方式 (8) **集成性**: 让分别开发的组件在一起正确工作的能力; 最小化接口复杂度, 统一命名规范, 遵循标准 (9) **移植性**: 是软件不经修改或稍加修改就可以运行于不同软硬件环境 (CPU, OS 和编译器) 的能力; 开发语言选择, 分层, 遵循标准 (10) **兼容性**: 不同产品相互交换信息的能力; API 设计, 遵循标准 (11) **经济性**: 开发成本, 开发时间和对市场的适应能力。重用性, 可修改性 (12) **正确性**: 软件按照需求正确执行任务的能力; (13) **完备性**: 软件能够支持用户所需求的全部功能的能力;

### 16. 架构和质量的联系

架构的选择极大地影响部分软件质量, 但不是全部。架构只为获得某个质量创造条件, 但并不能保证肯定获得

### 17. 软件架构技术发展与演化

**第一种**: (1) 系统 = 算法 + 数据结构 (1960’s ) (2) 系统 = 子程序 + 子程序 (1970’s ) (3) 系统 = 对象 + 对象 (1980’s ) (4) 系统 = 构件 + 连接件 (1990’s ) (5) 系统 = 服务 + 服务总线 (2000’s) (6) 系统 = 服务集群 + 中间件 (now) **第二种**: (1) 系统规模与复杂度: 简单到复杂 (2) 系统开放度: 封闭到开放 (3) 模块粒度: 细到粗 (4) 关注层面: 模块到连接件 **第三种**: (1) 单一服务器 (2) 应用服务和数据服务分离 (3) 缓存的使用 **第四种**: (1) 集群与并发 (2) 读写分离 (3) CDN 与反向代理 **第五种**: (1) 分布式数据库和文件系统 (2) nosql (3) 垂直划分业务服务 (4) 分布式服务

### 18. 软件中间件的定义

**中间件** 是一组程序, 应用于分布式系统各应用之中, 为系统屏蔽底层通讯和提供公共服务, 并保障系统的高可靠性, 高可用性, 高灵活性。

### 19. 中间件的作用

(1) 分布式应用借助中间件在不同技术之间共享资源。 (2) 中间件位于客户机/服务器的操作系统之上, 管理计算机资源和网络通讯。 (3) 中间件是连接两个独立应用程序或独立系统的软件, 即使它们具有不同的接口。 (4) 通过中间件, 应用程序可以工作于多平台或 OS 环境中, 中间件是一个通道, 保障应用信息的可靠传递。 (5) 中间件是一个桥梁, 达成商务构件的互联互通。 (6) 中间件是一个框架, 促成企业应用的完整集成。 (7) 中间件是一个平台, 实现分布式应用的顺利部署。

### 20. 哪些软件是中间件

**$$** 是的: (1) .NET (2) J2EE (3) RPC (4) SQL\*NET (5) HTTP (6) CORBA**$$** 不是的: (1) Mail System (2) Lotus Notes (3) WeChat

### 21. 如何鉴定中间件

(1) 中间件是软件。 (2) 中间件在应用软件和系统软件之间。 (3) 中间件是一类软件而不是一种。 (4) 中间件不是开发工具, 通常包含开发, 管理和部署。

### 22. 软件中间件的作用

(1) **屏蔽异构性**: 异构性表现在计算机的软硬件差异, 包括硬件 (CPU 和指令集, 硬件结构, 驱动程序等), 操作系统 (不同 OS 的 API 和开发环境), 数据库 (不同的存储和访问格式) 等。长期以来, 高级语言依赖于特定的编译器和 OS API 来编程, 而他们是不兼容的。 (2) **实现互操作**: 因为异构性, 产生的结果是软件依赖于计算环境, 使得各种不同软件之间在不同平台之间不能移植, 或者移植困难。而且, 因为网络协议和通信机制不同, 这些系统不能有效相互集成。 (3) **共性凝练和复用**: 软件应用领域越来越多, 相同领域的应用系统之间许多基础功能和结构是有相似性的。通过中间件提供简单, 一致, 集成的开发和运行环境, 简化分布式系统的设计, 编程和管理。

### 23. 软件中间件的意义

(1) 缩短开发周期 (2) 节约应用程序开发成本 (3) 降低运行成本 (4) 降低故障率 (5) 改善决策 (6) 应用系统群集/集成 (7) 减少软件维护 (8) 提高质量 (9) 改进技术 (10) 提高产品吸引力

### 24. 软件中间件的分类

(1) **应用服务类中间件**: 为应用系统提供一个综合的计算环境和支撑平台, 包括对象请求代理（ORB）中间件, 事务监控交易中间件, Java 应用服务器中间件等。 (2) **应用集成类中间件**: 提供各种不同网络应用系统之间的消息通信, 服务集成和数据集成功能, 包括常见的消息中间件, 企业集成 EAI, 企业服务总线以及相配套的适配器。 (3) **业务架构类中间件**: 除了可以将底层共性技术的特征抽象到中间件, 还可以将业务共性抽象至中间件, 形成应用模式, 如业务流程, 业务模型, 业务规则, 交互应用等。 (4) 数据访问, 消息, 分布式对象, 交易, 分布式构件（应用服务器, DCOM）, 工作流, 应用集成, 门户, 其他

### 25. 软件架构设计的模型

(1) **用例视图**: 描述系统的典型场景与功能, 主要图形包括 use case diagram 等。 (2) **逻辑视图**: 描述系统的抽象概念与功能 (类, 对象, 接口, 模式等), 主要图形包括 class diagrams, Communication diagrams 和 sequence diagrams 等; (3) **开发视图**: 描述系统中的子系统, 模块, 文件, 资源及其之间的关系, 主要图形包括 component diagrams, package diagrams 等; (4) **进程视图**: 描述系统的进程及其之间的通信协作关系, 主要图形包括 activity diagram, sequence diagram 等; (5) **物理视图**: 描述系统如何被安装, 部署与配置在分布式的物理环境下, 主要图形包括 deployment diagram 等。

### 26. Kruchten4+1视图模型的地位

(1) 4+1 视图模型是一种已经被标准化 (UML, RUP) 的方法, 用来描述和文档化软件系统的体系结构。 (2) 每个视图从不同的角度刻画了系统的结构属性, 将它们综合在一起就形成了系统的整体架构。逻辑视图和开发视图描述系统的静态结构, 而进程视图和物理视图描述系统的动态结构。对于不同类型的软件系统来说, 侧重的角度也有所不同。

# Chapter2-1

### 1. 软件设计模式的定义

(1) **最佳的实践**: (1) 面向对象的软件开发人员的试验和错误的经验总结; (2) 软件开发过程中一般问题的解决方案. (2) **代码编制工程化**（软件工程的基石）: (1) 重用代码; (2) 让代码更容易被他人理解; (3) 保证代码可靠性. (3) 每种软件设计模式都描述了一个在日常生产、生活中不断重复发生的问题，以及该问题的核心解决方案，都有相应的原理.

### 2. 软件设计模式的用途

(1) **开发人员的共同平台**: 软件设计模式提供标准的术语系统，且具体到特定情景. (2) **最佳的实践**: 软件设计模式提供了软件开发过程中面临的一般问题的最佳解决方案。有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计.

### 3. 软件设计模式的分类

(1) 创建型模式 (2) 结构型模式 (3) 行为型模式 (4) Java Web应用中还有: J2EE 设计模式

### 4. 创建型模式的定义，特点及包含

**$$** **定义**: 一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象.**$$** **特点**: 程序在判断针对某个给定实例需要创建哪些对象时更加灵活.**$$** **包含**: 工厂模式, 抽象工厂模式, 单例模式, 建造者模式, 原型模式

### 5. 结构型模式的特点及包含

**$$** **特点**: (1) 关注类和对象的组合. (2) 继承的概念被用来组合接口和定义组合对象获得新功能的方式.**$$** **包含**: 适配器模式, 桥接模式, 过滤器模式, 组合模式, 装饰器模式, 外观模式, 享元模式, 代理模式

### 6. 行为型模式的特点和包含

**$$** **特点**: 关注对象之间的通信.**$$** **包含**: 责任链模式, 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, 空对象模式, 策略模式, 模板模式, 访问者模式

### 7. J2EE模式的特点和包含

**$$** **特点**: 关注表示层.**$$** **包含**: MVC 模式, 业务代表模式, 组合实体模式, 数据访问对象模式, 前端控制器模式, 拦截过滤器模式, 服务定位器模式, 传输对象模式

### 8. 设计模式原则包含哪些原则

(1) **开闭原则**: 对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，需要使用接口和抽象类. (2) **里氏代换原则**: 任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范. (3) **依赖倒转原则**: 这个原则是开闭原则的基础。具体内容: 针对接口编程，依赖于抽象而不依赖于具体. (4) **接口隔离原则**: 使用多个隔离的接口，比使用单个接口要好。客户端不应该依赖它不需要的接口。降低类之间的耦合度。建立单一接口，尽量细化接口，接口中的方法尽量少。注意适度原则，一定要适度，过大的话会增加耦合性，而过小的话会增加复杂性和开发成本. (5) **迪米特法则又称最少知道原则**: 一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。降低系统的耦合度，使类与类之间保持松耦合状态. (6) **合成复用原则**: 复用类我们可以通过“继承”和“合成”两种方式来实现。尽量使用合成/聚合的方式，而不是使用继承。继承的优点: 容易实现并且容易修改和扩展继承来的内容。继承的缺点: 它最大的缺点就是增加了类之间的依赖，继承是属于“白箱”复用，父类对子类来说是透明的，这破坏了类的封装性。合成复用存在的缺点就是在系统中会存在较多的对象需要管理.

### 9. 简单依赖和聚合以及组合的关系

**$$** **相同点**: (1) 简单依赖和聚合都有接收 A 类型的方法. (2) 聚合和组合都有 A 类型字段.**$$** **不同点**: (1) 简单依赖没有 A 类型字段，接收 A 类型仅在方法中使用. (2) 组合没有接收 A 类型方法，自己 new 一个 A 类型实例使用.

# Chapter2-2

### 1. 创建型模式的要点

(1) 不允许显示创建

### 2. 单例模式的定义

单例模式提供了一种创建对象的最佳方式。单例模式涉及到一个单一的类：该类负责创建自己的对象，并确保只有单个对象被创建。该类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象.

### 3. 单例模式的注意点

(1) 单例类只能有一个实例. (2) 单例类必须自己创建自己的唯一实例. (3) 单例类必须给所有其他对象提供这一实例.

### 4. 单例模式的特点

(1)**模式意图**: 保证一个类仅有一个实例，并提供一个访问它的全局访问点. (2)**解决问题**: 一个全局使用的类频繁地创建与销毁. (3)**使用时机**: 当你想控制实例数目，节省系统资源的时候. (4)**解决方案**: 判断系统是否已经有这个单例，如果有则返回，如果没有则创建. (5)**关键代码**: 构造函数是私有的.

### 5. 单例模式的关键点

(1) 别人无法创建自己可以创建. (2) 别人可以获取.

### 6. 单例实现的方式

(1)**懒汉式**，线程不安全（最基础的实现方式）:(1)Lazy初始化;(2)无多线程安全，在多线程不能正常工作;(3)没有加锁synchronized，严格意义上它并不算单例模式;(4)

public class SingletonLazyUnsafe{private static SingletonLazyUnsafe instance;private SingletonLazyUnsafe(){}public static SingletonLazyUnsafe getInstance(){if(instance==null){instance=new SingletonLazyUnsafe();}return instance;}}

(2)**懒汉式**，线程安全:(1)Lazy初始化：第一次调用才初始化，避免内存浪费.; (2)多线程安全，能够在多线程中很好的工作，但是，效率很低，99%情况下一般不需要同步.; (3)缺点：必须加锁synchronized才能保证单例，但加锁会影响效率.; (4)

public class SingletonLazySafe{private static SingletonLazySafe instance;private SingletonLazySafe(){}public static synchronized SingletonLazySafe getInstance(){if(instance==null){instance=new SingletonLazySafe();}return instance;}}

(3)**饿汉式**:(1)无Lazy初始化，比较常用，类加载时就初始化，浪费内存，容易产生垃圾对象.; (2)多线程安全：是，没有加锁，执行效率会提高.; (3)基于classloader机制避免了多线程的同步问题，不过，instance在类装载时就实例化.; (4)

public class SingletonEager{private static final SingletonEager instance=new SingletonEager();private SingletonEager(){}public static SingletonEager getInstance(){return instance;}}

(4)**双检锁**/**双重校验锁**（DCL，即double**$$**checkedlocking）:(1)JDK版本：JDK1.5起; (2)Lazy初始化; (3)多线程安全：采用双锁机制，安全且在多线程情况下能保持高性能.; (4)

public class SingletonDCL{private static volatile SingletonDCL instance;private SingletonDCL(){}public static SingletonDCL getInstance(){if(instance==null){synchronized(SingletonDCL.class){if(instance==null){instance=new SingletonDCL();}}}return instance;}}

(5)**登记式**/**静态内部类**:(1)Lazy初始化：利用了classloader机制来保证初始化instance时只有一个线程，而Singleton类被装载了，instance不一定被初始化.; (2)SingletonHolder类没有被主动使用，只有通过显式调用getInstance方法时，才会显式装载SingletonHolder类，从而实例化instance。实现Singleton类延迟加载.; (3)多线程安全：达到双检锁的功效.; (4)

public class SingletonHolder{private SingletonHolder(){}private static class Holder{private static final SingletonHolder INSTANCE=new SingletonHolder();}public static SingletonHolder getInstance(){return Holder.INSTANCE;}}

### 7. 单例模式的优缺点

**$$** **优点**:(1)在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例.; (2)避免对资源的多重占用（比如写文件操作）。**$$** **缺点**:没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化.

### 8. 单例模式的使用场景及注意事项

(1)要求生产唯一序列号. (2)WEB中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来. (3)创建的一个对象需要消耗的资源过多，比如I/O与数据库的连接.; 注意事项: getInstance()方法中需要使用同步锁synchronized(Singleton.class)防止多线程同时进入造成instance被多次实例化.

### 9. 工厂模式的定义

工厂模式提供了一种创建对象的最佳方式。工厂模式在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象.

class ShapeFactory{public static Shape getShape(String shapeType){if("CIRCLE".equalsIgnoreCase(shapeType)){return new Circle();}else if("RECTANGLE".equalsIgnoreCase(shapeType)){return new Rectangle();}return null;}}public class SimpleFactoryDemo{public static void main(String[] args){Shape shape1=ShapeFactory.getShape("CIRCLE");shape1.draw();Shape shape2=ShapeFactory.getShape("RECTANGLE");shape2.draw();}}

### 10. 工厂模式的特点

(1)**模式意图**: 定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行.; (2)**解决问题**: 主要解决接口选择的问题.; (3)**使用时机**: 明确地计划不同条件下创建不同实例时。解决方案：让其子类实现工厂接口，返回的也是一个抽象的产品.; (4)**关键代码**: 创建过程在其子类执行.

### 11. 工厂模式的优缺点

**$$** **优点**:(1)一个调用者想创建一个对象，只要知道其名称就可以了.; (2)扩展性高，如果想增加一个产品，只要扩展一个工厂类即可.; (3)屏蔽产品的具体实现，调用者只关心产品的接口.**$$** **缺点**:(1)每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加.; (2)在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖.

### 12. 工厂模式的使用场景及注意事项

**$$** **使用场景**:(1)日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方.; (2)数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时.; (3)设计一个连接服务器的框架，需要三个协议，"POP3""IMAP""HTTP"，可以把这三个作为产品类，共同实现一个接口.**$$** **注意事项**:(1)作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式.; (2)需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度.

### 13. 抽象工厂模式的定义

抽象工厂模式是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂，提供了一种创建对象的最佳方式。在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象.

### 14. 抽象工厂模式的特点

(1)**模式意图**: 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类.; (2)**解决问题**: 主要解决接口选择的问题.; (3)**使用时机**: 系统的产品有多于一个的产品族，而系统只消费其中某一族的产品.; (4)**解决方案**: 在一个产品族里面，定义多个产品.; (5)**关键代码**: 在一个工厂里聚合多个同类产品.

### 15. 抽象工厂的优缺点

**$$** **优点**: 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象.**$$** **缺点**: 产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的Creator里加代码，又要在具体的里面加代码.

### 16. 抽象工厂的使用场景和注意事项

**$$** **使用场景**:(1)QQ换皮肤，一整套一起换.; (2)生成不同操作系统的程序.**$$** **注意事项**:产品族难扩展，产品等级易扩展.

### 17. 适配器模式的定义

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。适配器模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能.

### 18. 适配器模式的特点

(1)**模式意图**: 将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作.; (2)**解决问题**: 解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的.; (3)**使用时机**:(1)系统需要使用现有的类，而此类的接口不符合系统的需要.; (2)想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口.; (3)通过接口转换，将一个类插入另一个类系中.; (4)**解决方案**: 继承或依赖（推荐）.; (5)**关键代码**: 适配器继承或依赖已有的对象，实现想要的目标接口.

### 19. 适配器模式的实现

(1)**目标**（Target）接口: 当前系统业务所期待的接口，它可以是抽象类或接口.; (2)**适配者**（Adaptee）类: 它是被访问和适配的现存组件库中的组件接口.; (3)**适配器**（Adapter）类: 它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者.

interface Target{void request();}class Adaptee{public void specificRequest(){System.out.println("Called specificRequest from Adaptee");}}class Adapter extends Adaptee implements Target{@Override public void request(){specificRequest();}}public class ClassAdapterDemo{public static void main(String[] args){Target target=new Adapter();target.request();}}  
interface Target{void request();}class Adaptee{public void specificRequest(){System.out.println("Called specificRequest from Adaptee");}}class Adapter implements Target{private Adaptee adaptee;public Adapter(Adaptee adaptee){this.adaptee=adaptee;}@Override public void request(){adaptee.specificRequest();}}public class ObjectAdapterDemo{public static void main(String[] args){Adaptee adaptee=new Adaptee();Target target=new Adapter(adaptee);target.request();}}

### 20. 双向适配器模式的定义

(1) 同时实现两个接口（**目标接口**和**适配者接口**). (2) 内部持有适配者或目标接口的对象（对象适配器的方式）. (3) 转换逻辑可以在两种方向上进行.

### 21. 适配器模式的优缺点

**$$** **优点**:(1)可以让任何两个没有关联的类一起运行.; (2)提高了类的复用.; (3)增加了类的透明度.; (4)灵活性好.**$$** **缺点**:过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构.

### 22. 适配器模式的使用场景和注意事项

**$$** **使用场景**:有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式.**$$** **注意事项**:适配器不是在详细设计时添加的，而是解决正在服役的项目的问题.

### 23. 桥接模式的定义

**桥接模式**（Bridge Pattern）通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。桥接模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响.

### 24. 桥接模式的特点

(1)**模式意图**: 将抽象部分与实现部分分离，使它们都可以独立的变化.; (2)**解决问题**: 在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活.; (3)**使用时机**: 实现系统可能有多个角度分类，每一种角度都可能变化.; (4)**解决方案**: 把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合.; (5)**关键代码**: 抽象类依赖实现类.

### 25. 桥接模式的实现

(1)**抽象化**（Abstraction）角色: 定义抽象类，并包含一个对实现化对象的引用.; (2)**扩展抽象化**（Refined Abstraction）角色: 是抽象化角色的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法.; (3)**实现化**（Implementor）角色: 定义实现化角色的接口，供扩展抽象化角色调用.; (4)**具体实现化**（Concrete Implementor）角色: 给出实现化角色接口的具体实现.

interface Implementor{void operationImpl();}class ConcreteImplementorA implements Implementor{@Override public void operationImpl(){System.out.println("ConcreteImplementorA operation");}}abstract class Abstraction{protected Implementor implementor;public Abstraction(Implementor implementor){this.implementor=implementor;}public abstract void operation();}class RefinedAbstraction extends Abstraction{public RefinedAbstraction(Implementor implementor){super(implementor);}public void operation(){System.out.print("RefinedAbstraction: ");implementor.operationImpl();}}public class BridgePatternDemo{public static void main(String[] args){Implementor implementorA=new ConcreteImplementorA();Abstraction abstractionA=new RefinedAbstraction(implementorA);abstractionA.operation();}}

### 26. 桥接模式的优缺点

**$$** **优点**:(1)抽象和实现的分离.; (2)优秀的扩展能力.; (3)实现细节对客户透明.**$$** **缺点**:桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程.

### 27. 桥接模式的使用场景和注意事项

**$$** **使用场景**:(1)如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系.; (2)对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用.; (3)一个类存在两个独立变化的维度，且这两个维度都需要进行扩展.**$$** **注意事项**:对于两个独立变化的维度，使用桥接模式再适合不过了.

### 28. 桥接模式和适配器结合使用

当桥接模式的实现化角色的接口与现有类的接口不一致时，可以在二者中间定义一个适配器将二者连接起来.

### 29. 代理模式/委托模式的定义

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介.

### 30. 代理模式的特点

(1)**模式意图**: 为其他对象提供一种代理以控制对这个对象的访问.; (2)**解决问题**: 在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层.; (3)**使用时机**: 想在访问一个类时做一些控制.; (4)**解决方案**: 增加中间层.; (5)**关键代码**: 实现与被代理类组合.

### 31. 静态代理的内容

interface Subject{void request();}class RealSubject implements Subject{public void request(){System.out.println("RealSubject: Handling request.");}}class Proxy implements Subject{private RealSubject realSubject;public void request(){System.out.println("Proxy: Performing pre\*\*$$\*\*processing.");if(realSubject==null){realSubject=new RealSubject();}realSubject.request();System.out.println("Proxy: Performing post\*\*$$\*\*processing.");}}public class ProxyPatternDemo{public static void main(String[] args){Subject proxy=new Proxy();proxy.request();}}

### 32. 静态代理的优缺点

**$$** **优点**:(1)可以做到在不修改目标对象的功能前提下，对目标功能扩展.**$$** **缺点**:因为代理对象需要与目标对象实现一样的接口，所以会有很多代理类,类太多。同时，一旦接口增加方法，目标对象与代理对象都要维护.

### 33. 动态代理的内容

JDK自带动态代理。其通过自己实现InvocationHandler来实现动态代理，真正的代理对象由JDK再运行时为我们动态的来创建.

### 34. 动态代理实现

class DynamicProxyHandler implements InvocationHandler{private Object target;public DynamicProxyHandler(Object target){this.target=target;}public Object invoke(Object proxy, Method method, Object[] args)throws Throwable{System.out.println("Proxy: Performing pre\*\*$$\*\*processing.");Object result=method.invoke(target,args);System.out.println("Proxy: Performing post\*\*$$\*\*processing.");return result;}}public class JDKDynamicProxyDemo{public static void main(String[] args){RealSubject realSubject=new RealSubject();Subject proxy=(Subject)Proxy.newProxyInstance(RealSubject.class.getClassLoader(),new Class[]{Subject.class},new DynamicProxyHandler(realSubject));proxy.request();}}

### 35. 动态代理的总结

虽然相对于静态代理，动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。但是JDK自带动态代理只能支持实现了Interface的类。是实现Spring AOP的基础.

### 36. 动态代理的优缺点

**$$** **优点**:(1)代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用.; (2)代理对象可以扩展目标对象的功能；例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类.; (3)代理模式能将客户端与目标对象分离，在一定程度上降低系统的耦合度.**$$** **缺点**:(1)在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢.; (2)增加了系统的复杂度.

### 37. 代理模式按职责划分的使用场景

(1)远程代理. (2)虚拟代理. (3)Copy**$$**on**$$**Write代理. (4)保护代理. (5)Cache代理. (6)防火墙代理. (7)同步化代理. (8)智能引用代理.

### 38. 中介者模式的定义

中介者模式定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。中介者模式是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护.

### 39. 中介者模式的特点

(1)**模式意图**: 用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互.; (2)**解决问题**: 对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理.; (3)**使用时机**: 多个类相互耦合，形成了网状结构.; (4)**解决方案**: 将网状结构分离为星型结构.; (5)**关键代码**: 对象Colleague之间的通信封装到一个类中单独处理.

### 40. 中介者模式的实现

(1)**抽象中介者**（Mediator）角色: 它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法.; (2)**具体中介者**（ConcreteMediator）角色: 实现中介者接口，定义一个List来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色.; (3)**抽象同事类**（Colleague）角色: 定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能.; (4)**具体同事类**（Concrete Colleague）角色: 是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互.

interface Mediator{void sendMessage(String message,Colleague colleague);void registerColleague(Colleague colleague);}class ConcreteMediator implements Mediator{private List<Colleague> colleagues=new ArrayList<>();public void registerColleague(Colleague colleague){colleagues.add(colleague);}public void sendMessage(String message,Colleague sender){for(Colleague colleague:colleagues){if(colleague!=sender){colleague.receiveMessage(message);}}}}abstract class Colleague{protected Mediator mediator;public Colleague(Mediator mediator){this.mediator=mediator;}public abstract void receiveMessage(String message);public abstract void sendMessage(String message);}class ColleagueA extends Colleague{public ColleagueA(Mediator mediator){super(mediator);}public void receiveMessage(String message){System.out.println("Colleague received: "+message);}public void sendMessage(String message){System.out.println("Colleague sends: "+message);mediator.sendMessage(message,this);}}public class MediatorPatternDemo{public static void main(String[] args){Mediator mediator=new ConcreteMediator();Colleague colleagueA=new ColleagueA(mediator);Colleague colleagueB=new ColleagueA(mediator);mediator.registerColleague(colleagueA);mediator.registerColleague(colleagueB);colleagueA.sendMessage("Hello from A!");colleagueB.sendMessage("Hello from B!");}}

### 41. 中介者模式和单例模式结合

不定义中介者接口，把具体中介者对象实现成为单例。同事对象不持有中介者，而是在需要的时候直接获取中介者对象并调用.

### 42. 中介者模式的优缺点

**$$** **优点**:(1)降低了类的复杂度，将一对多转化成了一对一.; (2)各个类之间的解耦.; (3)符合迪米特原则.**$$** **缺点**:中介者会庞大，变得复杂难以维护.

### 43. 中介者模式的使用场景和注意事项

**$$** **使用场景**:(1)系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象.; (2)想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类.**$$** **注意事项**:不应当在职责混乱的时候使用.

### 44. 观察者模式的定义

又被称为发布**$$**订阅/模型**$$**视图模式，属于行为型设计模式的一种。是一个在项目中经常使用的模式。指多个对象间存在一对多的依赖关系，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被自动更新.

### 45. 观察者模式的特点

(1)**模式意图**: 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被自动更新.; (2)**解决问题**: 一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作.; (3)**使用时机**: 一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知.; (4)**解决方案**: 使用面向对象技术，可以将这种依赖关系弱化.; (5)**关键代码**: 在抽象类里有一个ArrayList存放观察者们.

### 46. 观察者模式的实现

(1)**抽象目标**（Subject）角色: 提供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.; (2)**具体目标**（Concrete Subject）角色: 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.; (3)**抽象观察者**（Observer）角色: 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.; (4)**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.

interface Subject{void registerObserver(Observer observer);void removeObserver(Observer observer);void notifyObservers();}class ConcreteSubject implements Subject{private List<Observer> observers=new ArrayList<>();private String state;public void registerObserver(Observer observer){observers.add(observer);}public void removeObserver(Observer observer){observers.remove(observer);}public void notifyObservers(){for(Observer observer:observers){observer.update(state);}}public void setState(String state){this.state=state;notifyObservers();}}interface Observer{void update(String state);}class ConcreteObserver implements Observer{private String name;public ConcreteObserver(String name){this.name=name;}public void update(String state){System.out.println(name+" received update: "+state);}}public class ObserverPatternDemo{public static void main(String[] args){ConcreteSubject subject=new ConcreteSubject();Observer observer1=new ConcreteObserver("Observer 1");Observer observer2=new ConcreteObserver("Observer 2");subject.registerObserver(observer1);subject.registerObserver(observer2);subject.setState("State Changed to A");subject.setState("State Changed to B");}}

### 47. 观察者模式的注意事项

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则.

### 48. 观察者模式的扩展

在Java中，通过java.util.Observable类和java.util.Observer接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例.

(1)**Observable**类: 用Vector向量存所有要通知的观察者对象，包含3个方法.; (2)**Observer**接口: 监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用void update(Observable o,Object arg)方法，进行相应的工作.

public class ConcreteSubject extends Observable{private String state;public void setState(String state){this.state=state;setChanged();notifyObservers(state);}public String getState(){return state;}}public class ConcreteObserver implements Observer{private String name;public ConcreteObserver(String name){this.name=name;}public void update(Observable o,Object arg){System.out.println(name+" received update: "+arg);}}public class ObserverPatternWithJavaAPI{public static void main(String[] args){ConcreteSubject subject=new ConcreteSubject();ConcreteObserver observer1=new ConcreteObserver("Observer 1");ConcreteObserver observer2=new ConcreteObserver("Observer 2");subject.addObserver(observer1);subject.addObserver(observer2);subject.setState("State Changed to A");subject.setState("State Changed to B");}}

### 49. 观察者模式的优缺点

**$$** **优点**:(1)观察者和被观察者是抽象耦合的.; (2)建立一套触发机制.**$$** **缺点**:(1)如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间.; (2)如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃.; (3)观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化.

### 50. 观察者模式的使用场景和注意事项

**$$** **使用场景**:(1)一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用.; (2)一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变。可以降低对象之间的耦合度.; (3)一个对象必须通知其他对象，而并不知道这些对象是谁。需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制.**$$** **注意事项**:(1)JAVA中已经有了对观察者模式的支持类.; (2)避免循环引用.; (3)如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式.

### 51. 访问者模式的定义

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式.

### 52. 访问者模式的特点

(1)**模式意图**: 主要将数据结构与数据操作分离.; (2)**解决问题**: 稳定的数据结构和易变的操作耦合问题.; (3)**使用时机**: 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中.; (4)**解决方案**: 在被访问的类里面加一个对外提供接待访问者的接口.; (5)**关键代码**: 在数据基础类里面有一个方法接受访问者，将自身引用传入访问者.

### 53. 访问者模式的实现

(1)**抽象访问者**（Visitor）角色: 定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作visit(),其参数类型标识了被访问的具体元素.; (2)**具体访问者**（ConcreteVisitor）角色: 实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么.; (3)**抽象元素**（Element）角色: 声明一个包含接受操作accept()的接口，被接受的访问者对象作为accept()方法的参数.; (4)**具体元素**（ConcreteElement）角色: 实现抽象元素角色提供的accept()操作，其方法体通常都是visitor.visit(this)，另外具体元素中可能还包含本身业务逻辑的相关操作.; (5)**对象结构**（Object Structure）角色: 是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由List、Set、Map等聚合类实现.

interface Visitor{void visit(ElementA elementA);void visit(ElementB elementB);}class ConcreteVisitor implements Visitor{@Override public void visit(ElementA elementA){System.out.println("Processing ElementA");}@Override public void visit(ElementB elementB){System.out.println("Processing ElementB");}}interface Element{void accept(Visitor visitor);}class ElementA implements Element{public void accept(Visitor visitor){visitor.visit(this);}}class ElementB implements Element{public void accept(Visitor visitor){visitor.visit(this);}}class ObjectStructure{private Element[] elements;public ObjectStructure(){elements=new Element[]{new ElementA(),new ElementB()};}public void accept(Visitor visitor){for(Element element:elements){element.accept(visitor);}}}public class VisitorPatternDemo{public static void main(String[] args){ObjectStructure objectStructure=new ObjectStructure();Visitor visitor=new ConcreteVisitor();objectStructure.accept(visitor);}}

### 54. 访问者模式的优缺点

**$$** **优点**:(1)符合单一职责原则.; (2)优秀的扩展性.; (3)灵活性.**$$** **缺点**:(1)具体元素对访问者公布细节，违反了迪米特原则.; (2)具体元素变更比较困难.; (3)违反了依赖倒置原则，依赖了具体类，没有依赖抽象.

### 55. 访问者模式的使用场景和注意事项

**$$** **使用场景**:(1)对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作.; (2)需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类.**$$** **注意事项**:访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器.

# Chapter3-1

### 1. 计算复杂性的表现

(1)高性能 (2)高扩展 (3)高可用

### 2. 传统垂直结构改造的核心

传统垂直架构改造的核心就是要对应用进行服务化，服务化改造使用到的核心技术就是分布式服务框架.

### 3. 从集中到分布式依旧存在的问题

(1)系统开发维护成本高，部署效率低，应用数量膨胀，数据库连接数持续变高. (2)代码复用难，导致开发、测试、维护等工作烦、难、杂. (3)难以适应敏捷持续交付的挑战.

### 4. 大规模系统架构设计的一般原则

大规模系统架构的设计一般原则就是尽可能地拆分，以达到更好的独立扩展与伸缩、更灵活的部署、更好的隔离和容错、更高的开发效率.

### 5. 业务纵向拆分方法

按业务进行梳理，根据业务的特性把应用拆开，不同的业务模块独立部署.

### 6. 业务横向拆分方法

将核心的、公共的业务拆分出来，通过分布式服务框架对业务进行服务化，消费者通过标准的契约来消费这些服务。服务提供者独立打包、部署和演进，与消费者解藕.

### 7. 服务治理的目标

有效管控服务，提升服务运行质量，防止业务服务代码架构腐化.

### 8. 服务治理解决的主要问题

(1)**生命周期的管理**：服务上线随意，线上服务鱼龙混杂，上线容易下线难。需要规范化上线审批、测试发布流程、下线通知等. (2)**服务容量规划**：需要能够采集服务调用性能、时延、成功率、系统资源占用等综合性能指标，分析并识别服务容量瓶颈，合理分配服务容量（计算、数据、网络等资源）. (3)**运行时治理**：流量陡增时的非核心业务SLA降级；缓存失效，数据I/O开销陡增导致业务失败，需要在线调大服务调用超时时间；非核心服务故障时，业务应放行并执行本地降级逻辑. (4)服务安全：身份验证与权限验证.

### 9. 非功能度量指标/性能指标包括什么

(1)CPU**速度**(MIPS Million Instructions Per Second) (2)**网络带宽**(Mbps Megabits per second) (3)**吞吐量**(MIPS、TFLOPS、TPS、QPS ):指系统在单位时间内处理请求的数量. (4)**RT响应时间**(s):指系统对请求作出响应的时间. (5)**网络延时**(s) (6)**并发用户数**:指系统可以同时承载的正常使用系统功能的用户的数量. (7)**TPS** (Transactions Per Second):客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数，包括了：1)用户请求服务器 2)服务器自己的内部处理 3)服务器返回给用户这三个过程，每秒能够完成N个这三个过程，TPS也就是N. (8)**QPS** (Queries Per Second):每秒能处理查询数目。是一台服务器每秒能够相应的查询次数，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。QPS基本类似于TPS，但是不同的是，对于一个页面的一次访问，形成一个TPS；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“QPS”之中.

### 10. 扩展性Scalability定义及分类

(1)**定义**:适应不断增长的处理任务的能力. (2)**分类**: (a)垂直扩展：提高硬件配置. (b)水平扩展：增加新的计算机分布式计算.

### 11. 高可用性的定义

系统在几乎任何时刻都可被正常，访问通常量化为一年中正常运行的时间比.

### 12. 系统可用性如何计算

(1)**系统可用性**=MTTF/(MTTF+MTTR). (2)**平均故障间隔时间**MTBF=MTTF+MTTR. (3)**平均故障时间**MTTF. (4)**平均修复时间**MTTR.

# Chapter3-3

### 1. 为什么要设计高性能集群/负载均衡

单服务器有性能天花板，无论如何优化，无论采用多好的硬件，总会有一个上限，当无法满足业务需求时，就需要设计高性能集群来提升系统整体的处理性能.

### 2. 高性能集群的复杂性

高性能集群的复杂性主要体现在需要增加一个任务分配器（负载均衡器）以及为任务选择一个合适的任务分配算法.

### 3. 常见的负载均衡器

(1) DNS负载均衡. (2) 硬件负载均衡. (3) 软件负载均衡.

### 4. DNS负载均衡定义

DNS是最简单也是最常见的负载均衡方式，一般用来实现地理级别的均衡.

### 5. DNS负载均衡的优缺点

**$$** **优点**: (1) 简单、成本低：交给DNS服务器处理，无须自己开发、维护负载均衡设备.; (2) 就近访问，提升访问速度：DNS解析时可以根据请求来源IP，解析成距离用户最近的服务器地址，可以加快访问速度，改善性能.**$$** **缺点**: (1) **更新不及时**：DNS缓存的时间比较长，修改DNS配置后，还是有很多用户会继续访问修改前的IP，这样的访问会失败，达不到负载均衡的目的，并且也影响用户正常使用业务.; (2) **扩展性差**：DNS负载均衡的控制权在域名商那里，无法根据业务特点针对其做更多的定制化功能和扩展特性.; (3) **分配策略比较简单**：DNS负载均衡支持的算法少；不能区分服务器的差异（不能根据系统与服务的状态来判断负载）；也无法感知后端服务器的状态.

### 6. 硬件负载均衡的定义

硬件负载均衡通过单独的硬件设备来实现负载均衡功能.

### 7. 硬件负载均衡的优缺点

**$$** **优点**: (1) **功能强大**：全面支持各层级的负载均衡，支持全面的负载均衡算法，支持全局负载均衡.; (2) **性能强大**.; (3) **稳定性高**：商用硬件负载均衡，经过了良好的严格测试，经过大规模使用，稳定性高.; (4) **支持安全防护**：硬件均衡设备除具备负载均衡功能外，还具备防火墙、防DDoS攻击等安全功能.**$$** **缺点**: (1) **价格昂贵**.; (2) **扩展能力差**：硬件设备，可以根据业务进行配置，但无法进行扩展和定制.

### 8. 软件负载均衡的定义

软件负载均衡通过负载均衡软件来实现负载均衡功能，常见的有Nginx和LVS，Nginx是7层负载均衡，LVS是Linux内核的4层负载均衡，4层和7层的区别就在于协议和灵活性，Nginx支持HTTP、E**$$**mail协议，而LVS是4层负载均衡，和协议无关，几乎所有应用都可以做，例如，聊天、数据库等.

### 9. 软件和硬件方法的区别

软件和硬件负载均衡方法的最主要区别就在于性能。硬件负载均衡性能远远高于软件负载均衡性能。Nginx性能是万级，LVS的性能是十万级，而F5性能是百万级.

### 10. 软件负载均衡的实现

(1) 使用开源的系统进行负载均衡.; (2) 如果业务比较特殊，也可能基于开源系统进行定制（例如，Nginx插件），甚至进行自研.

### 11. 软件负载均衡的优缺点

**$$** **优点**: (1) **简单**：无论是部署还是维护都比较简单.; (2) **便宜**：只要买个Linux服务器，装上软件即可.; (3) **灵活**：4层和7层负载均衡可以根据业务进行选择；也可以根据业务进行比较方便的扩展，例如，可以通过Nginx的插件来实现业务的定制化功能.**$$** **与硬件负载均衡相比的缺点**: (1) 性能一般.; (2) 功能没有硬件负载均衡那么强大.; (3) 一般不具备防火墙和防DDoS攻击等安全功能.

### 12. 负载均衡的典型架构

(1) **地理级别负载均衡**：当用户访问时，DNS会根据用户的地理位置来决定返回哪个机房的IP. (2) **集群级别负载均衡**：F5收到请求后，进行集群级别的负载均衡. (3) **机器级别的负载均衡**：Nginx收到用户请求后，将用户请求发送给集群里面的某台服务器.

### 13. 负载均衡算法分类

根据算法期望达到的目的，大体上可以分为下面几类:

(1) **任务数平分类**：负载均衡系统将收到的任务平均分配给服务器进行处理，这里的“平均”可以是绝对数量的平均，也可以是比例或者权重上的平均.; (2) **负载均衡类**：负载均衡系统根据服务器的负载来进行分配，用连接数、I/O使用率、网卡吞吐量等来衡量系统的压力.; (3) **性能最优类**：负载均衡系统根据服务器的响应时间来进行任务分配，优先将新任务分配给响应最快的服务器.; (4) **Hash类**：负载均衡系统根据任务中的某些关键信息进行Hash运算，将相同Hash值的请求分配到同一台服务器上。常见的有源地址Hash、目标地址Hash、session id hash、用户id hash等.

### 14. 轮询的定义

负载均衡系统收到请求后，按照顺序轮流分配到服务器上.

### 15. 轮询的特点及优缺点

(1) 轮询是最简单的一个策略，无须关注服务器本身的状态，例如：某服务器有bug或某服务器能力较强都无视.; (2) “简单”是轮询算法的优点，也是它的缺点.

### 16. 加权轮询的定义

负载均衡系统根据服务器权重进行任务分配，这里的权重一般是根据硬件配置进行静态配置的.

### 17. 加权轮询的特点及优缺点

加权轮询是轮询的一种特殊形式，其主要目的就是为了解决不同服务器处理能力有差异的问题，解决了轮询算法中无法根据服务器的配置差异进行任务分配的问题，但同样存在无法根据服务器的状态差异进行任务分配的问题.

### 18. 负载最低优先的定义

负载均衡系统将任务分配给当前负载最低的服务器。LVS，可以以“连接数”来判断服务器的状态，服务器连接数越大，表明服务器压力越大。Nginx，可以以“HTTP请求数”来判断服务器状态（Nginx内置的负载均衡算法不支持这种方式，需要进行扩展）

### 19. 负载均衡衡量标准分类

(1) 如果是 **CPU密集型**，可以以“CPU负载”来衡量系统压力.; (2) 如果是 **I/O密集型**，可以以“I/O负载”来衡量系统压力.

### 20. 负载最低优先的特点及好处坏处

(1) 解决了轮询算法中无法感知服务器状态的问题，由此带来的代价是复杂度要增加很多.; (2) CPU负载最低优先的算法要求以某种方式收集每个服务器的CPU负载.; (3) 不同业务最优的时间间隔是不一样的，时间间隔太短容易造成频繁波动，时间间隔太长又可能造成峰值来临时响应缓慢.; (4) 负载最低优先算法基本上能够比较完美地解决轮询算法的缺点，可以感知服务器当前的运行状态，其代价是复杂度大幅上升.

负载最低优先算法，看起来很美好，但实际应用场景反而不如轮询（包括加权轮询）.

### 21. 性能最优算法的定义及和负载区别

性能最优优先类算法则是站在客户端的角度来进行分配的，优先将任务分配给处理速度最快的服务器，通过这种方式达到最快响应客户端的目的，负载最低优先类算法是站在服务器的角度来进行分配的.

### 22. 性能最优算法的特点及好处坏处

(1) 性能最优优先类算法本质上也是感知了服务器的状态只是通过响应时间这个外部标准来衡量服务器状态而已.; (2) 复杂度很高，主要体现在：负载均衡系统需要收集和分析每个服务器每个任务的响应时间，在大量任务处理的场景下，这种收集和统计本身也会消耗较多的性能.; (3) 工业上使用采样，并调优采样率.

### 23. Hash类算法的定义

负载均衡系统根据任务中的某些关键信息进行Hash运算，将相同Hash值的请求分配到同一台服务器上，这样做的目的主要是为了满足特定的业务需求.

### 24. Hash算法的分类

(1) **源地址Hash**：将来源于同一个源IP地址的任务分配给同一个服务器进行处理，适合于存在事务、会话的业务.; (2) **ID Hash**：将某个ID标识的业务分配到同一个服务器中进行处理，这里的ID一般是临时性数据的ID（如session id）.

# Chapter3-4

### 1. 成功的分布式系统依赖于什么

隐藏或简化消息传递的通信模型.

### 2. 进程通信的挑战

对于分布在网络中的多进程，底层的网络是不可靠的，比用共享内存更难，对于分布式系统进程通信是心脏.

### 3. 如何进行进程通信

没有基于共享内存的原语，基于低层级的消息传递由底层网络支撑.

### 4. RPC的特点

(1)**隐藏了大多数复杂的信息传递.** (2)**理想的客户端/服务器应用程序.**

### 5. MON的特点

(1)**高级消息排队模型，类似于电子邮件.** (2)**通信并不遵循相当严格的客户机/服务器交互模式.**

### 6. 如何对通信模型分类

(1)**寻址类型**: direct（sym/asym）& indirect. (2)**阻塞类型**: synchronous & asynchronous. (3)**缓存类型**: not-buffered & buffered. (4)**内容类型**: event & command & data & stream. (5)**确认类型**: unconfirmed & confirmed & 3-way-handshake. (6)**接收节点数**: Point-to-Point & Multicast & Anycast & Geocast & Broadcast. (7)**通信方向**: 单向 & 双向（半双工/全双工）. (8)**发起方**: Client-Initiated（pull）& Server-Initiate（Push）. (9)**消息存储**: (a)持续通信：由通信中间件存储的消息，只要它需要传递它。收发双方无需同步，接收器将在下次运行时得到消息.; (b)瞬态通信：收发双方均运行时，才会存储消息中间件传输中断或接收者无法传递消息，则丢弃该消息.传输级通信服务.

### 7. 面向消息中间件（MOM）的特点

(1)**基于消息的通信.** (2)**消息存储在消息队列里.** (3)**消息服务器解耦了客户端和服务端.** (4)**关于消息内容的各种假设.**

### 8. MOM的形式

**消息排队**和**发布-订阅**.

### 9. 面向消息的持久消息通信（MOPC）的特点

(1)**面向消息的中间件服务的重要类别.** (2)**持久异步通信.** (3)**为消息提供中间形态的存储容量.** (4)**不需要发送方或接收方在消息传输期间保持活跃状态.** (5)**支持允许消息传输的时间开销.**

### 10. MOM的属性

(1)**异步交互.** (2)**可靠服务交付.** (3)**通过中间消息服务器处理消息.** (4)**支持数据库集成.**

### 11. 消息队列模型的特点

(1)**应用程序通过在特定的队列中插入消息来进行通信（basic idea）.** (2)**在最终被传递到目的地之前，通过一系列通信服务器转发的消息.** (3)**接收双方无需同步.** (4)**发送者可以保证其消息最终将被插入到收件人的队列中，但不能保证何时插入.**

### 12. 异步（持久）通信的好处和坏处

**好处**: (1)高并发. (2)速度快. (3)可以补偿发送方和接收方之间的速度偏差（但不补偿速度差异）. (4)发送方接收方不需要同时运行. **坏处**: (1)同步更复杂. (2)发送方不知道是否已经送达. (3)存在缓冲区溢出风险.

### 13. 消息队列的特点

(1)**作为发送和接收之间的命名消息目的地.** (2)**允许进程独立执行和失败.** (3)**可以掩盖进程失败和通信失败.**

### 14. 消息队列优先级分类

(1)最高优先级优先. (2)加权公平调度.

### 15. 消息中间件（MOM）的附加功能

(1)**支持多种消息传递模型.** (2)**队列管理.** (3)**连接管理.** (4)**服务质量（QoS）.** (5)**数据转换.**

### 16. 队列管理器的作用

(1)**创建/删除队列.** (2)**允许启动和停止队列.** (3)**更改现有队列的属性.** (4)**允许监视性能、故障和恢复.**

### 17. 附加队列管理器的功能

(1)**消息代理.** (2)**重新格式化数据.** (3)**全局翻译.** (4)**了解源代码.** (5)**和目标的结构/格式.**

### 18. 队列消息传递模式

(1)**一对一（Message Passing）.** (2)**一对多.** (3)**多对一.** (4)\*\*多对多（发布订阅模式）.

# Chapter3-5

### 1. 如何得知WebService中API和服务器的信息

(1) 确定IP (2) 获取API名称 (3) 知悉输入输出参数

### 2. 分布式服务框架的原理和Service层介绍

**$$** **原理** **1. Service层**：(1) Customer Service Interface（Spring Bean） (2) Dynamic Proxy（Java/cglib/Javas Sit）（在这里注册方法） **2. FilterChain层**：LoadBalance，Timeout/resend **3. RPC层**（框架提供）：(1) Binary Protocol，Text Protocol，WebService，Restful，Hessian (2) NIO Framework，Http Based Protocol (3) Transport**$$** **Service**介绍：主要包括Java动态代理，消费者使用，主要用于将服务提供者的接口封装成远程服务调用；Java反射，服务提供者使用，根据消费者请求消息中的接口名、方法名、参数列表反射调用服务提供者的接口本地实现类。再向上就是业务的服务接口定义和实现类，对于使用Spring配置化开发的就是Spring Bean，具体服务逻辑内容由业务部门来实现，平台部分负责将业务接口发布成远程服务。

### 3. 分布式服务框架主要功能

(1) **服务注册中心**：负责服务的发布和通知，通常支持对等集群部署，某一个服务注册中心宕机并不会导致整个服务注册中心集群不可用。即便整个服务注册中心全部宕机，也只影响新服务的注册和发布，不影响已经发布的服务的访问。HSF使用的是基于数据库的ConfigServer，Dubbo默认使用Zookeeper，SpringCloud使用eureka。 (2) **服务治理中心**：通常包含服务治理接口和服务质量Portal，架构师、测试人员和系统运维人员通过服务治理Portal对服务的运行状态、历史数据、健康度和调用关系等进行可视化的分析和维护，目标就是要持续化服务，防止服务架构腐化，保证服务高质量运行。

### 4. 分布式服务框架的功能特性

(1) **服务订阅分布**： **$$** **配置化发布和引用服务**：支持通过XML配置的方法发布和导入服务，降低对业务代码的侵入。 **$$** **服务自动发现机制**：支持服务实时自动发现，由注册中心推送服务地址，消费者不需要配置服务提供者地址，服务地址透明化。 **$$** **服务在线注册和去注册**：支持运行注册新服务，也支持运行态取消某个服务的注册。

(2) **服务路由**： **$$** **路由策略**：默认提供随机路由、轮循、基于权重的路由策略。避免每个框架使用者都重复开发。 **$$** **粘滞连接**：总是向同一个提供方发起请求，除非此提供方宕机，再切换到另一台。 **$$** **路由定制**：支持用户自定义路由策略，扩展平台的功能。

(3) **集群容错**： **$$** **Failover**：失败自动切换，当出现失败，重试其他服务器，通常用于读操作；也可用于幂等性写操作。 **$$** **Failback**：失败自动恢复，后台记录失败请求，定时重发，通常用于消息通知操作。 **$$** **Failfast**：快速失败，只发起一次调用，失败立即报错，通常用于非幂等性的写操作。

(4) **服务调用**： **$$** **同步调用**：消费者发起服务调用之后，同步阻塞等待服务端响应。 **$$** **异步调用**：消费者发起服务调用之后，不阻塞立即返回，由服务端返回应答后异步通知消费者。 **$$** **并行调用**：消费者同时对多个服务提供者批量发起服务调用请求，批量发起请求后，集中等待应用。

(5) **多协议**： **$$** **私有协议**：支持二进制等私有协议，支持私有协议定制和扩展。 **$$** **公有协议**：提供Web Service等公有协议，用于外部服务对接。

(6) **序列化方式**： **$$** **二进制类序列化**：支持Thrift、Protocol buffer等二进制协议，提升序列化性能。 **$$** **文本类序列化**：支持JSON、XML等文本类型的序列化方式，提升通用性和可读性。

(7) **统一配置**： **$$** **本地静态配置**：安装部署修改一次，运行态不修改的配置，可以存放到本地配置文件中。 **$$** **基于配置中心的动态配置**：运行态需要调整的参数，统一放到配置中心（服务注册中心），修改之后统一下发，实时生效。

### 5. 幂等性的概念，重要性和场景

**$$** **概念**：用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。 **$$** **重要性**：由于服务无状态的本质，对于业务的敏感性是很弱的。如果不支持幂等性的话，就会导致服务重复操作，对于业务数据进行违背业务逻辑的重复性操作。 **$$** **场景**： (1) 网络波动：因网络波动，可能会引起重复请求 (2) 分布式消息消费：任务发布后，使用分布式消息服务来进行消费 (3) 用户重复操作：用户在使用产品时，可能无意地触发多笔交易，甚至没有响应而有意触发多笔交易 (4) 未关闭的重试机制：因开发、测试、或运维人员未检测出错误的情况下开启的重试机制（如Nginx重试、RPC通信重试或业务层重试等）

### 6. 数据库的幂等性及解决方案

**$$** **幂等性的影响往往作用在数据上，而不同数据库操作对于幂等性的反应也不一样**： (1) **新增类请求**：不具备幂等性 (2) **查询类动作**：重复查询不会产生或变更新的数据，查询具有天然幂等性 (3) **更新类请求**： **$$** 基于主键的计算式Update，不具备幂等性 **$$** 基于主键的非计算式Update，具备幂等性 **$$** 基于条件查询的更新，不一定具备幂等性 (4) **删除类请求**： **$$** 基于主键的Delete具备幂等性 **$$** 一般业务层面都是逻辑删除（即update操作），而基于主键的逻辑删除操作也是具有幂等性的**$$** **解决办法**： (1) **数据库加锁法**：让关键资源的操作串行起来，但是这会引入其他的问题，包括效率问题，死锁问题等。 (2) **全局唯一ID法**：根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。该方案缺点实现起来困难，同时与服务的业务解绑有一定的冲突。 (3) **去重表法**：在本身具有唯一标识的业务场景下是非常好的方法，利用唯一的标识号，如订单号等，判断操作是否被重复执行。 (4) **多版本控制法**：为每一次操作添加一次版本号，以示区别。缺点在于版本号的管理，以及通常适用于更新操作，并且往往需要配合日志来完成数据最终一致性。 (5) **状态机控制法**：这种方法适合在有状态机流转的情况下，比如订单的创建和付款，订单的付款肯定是在之前，这时我们可以通过在设计状态字段时，使用int类型，并且通过值类型的大小来做幂等，比如订单的创建为0，付款成功为100，付款失败99。

### 7. 微服务的定义及需要做什么

**$$** **定义**： (1) 微服务是一种架构设计模式。在微服务架构中，业务逻辑被拆分成一系列小而松散耦合的分布式组件，共同构成了较大的应用。每个组件都被称为微服务。 (2) 每个微服务都在整体架构中执行着单独的任务，或负责单独的功能。 (3) 每个微服务可能会被一个或多个其他微服务调用，以执行较大应用需要完成的具体任务。 (4) 系统为任务执行——比如搜索或显示图片任务，或者其他可能需要多次执行的任务提供了统一的解决处理方式，并限制应用内不同地方生成或维护相同功能的多个版本。 (5) 微服务是围绕业务功能构建的，可以通过全自动部署机制进行独立部署。这些服务的集中化管理已经是最少的，它们可以用不同的编程语言编写，并使用不同的数据存储技术。**$$** **做什么**： (1) 负责单个功能 (2) 单独部署 (3) 包含一个或多个进程 (4) 拥有自己的数据存储 (5) 一支小团队就能维护几个微服务 (6) 可替换的

### 8. 微服务与SOA架构的区别

SOA通常由大块业务逻辑组成，耦合松散，适用于任何类型公司架构，注重中央管理，目标是确保应用能互斥操作；而微服务由单独任务或小块业务逻辑组成，总是松耦合，适用于小型、专注于功能交叉的公司架构，注重分散管理，目标是执行新功能并快速扩展开发团队。

### 9. web传参方式分类

(1) **URL参数传递**：通过URL的查询字符串（即问号后面的参数）传递参数。 (2) **URL路径传递**：通过URL的路径部分传递参数。

### 10. SOA服务架构的服务集成（和微服务对比）

SOA体系下，服务之间通过企业服务总线（Enterprise Service Bus）通信，许多业务逻辑在中间层（消息的路由、转换和组织）。微服务架构倾向于降低中心消息总线（类似于ESB）的依赖，将业务逻辑分布在每个具体的服务终端。大部分微服务基于HTTP、JSON这样的标准协议，集成不同标准和格式变的不再重要。另外一个选择是采用轻量级的消息总线或者网关，有路由功能，没有复杂的业务逻辑。

### 11. 微服务架构的服务集成及评价

(1) **点对点**：直接调用服务，每个微服务都开放REST API，并且调用其它微服务的接口。在比较简单的微服务应用场景下，这种方式还可行，随着应用复杂度的提升，会变得越来越不可维护，这是尽量不采用点对点的集成方式。 (2) **API-网关方式**：其核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GW注册和管理服务。目前，API网关方式应该是微服务架构中应用最广泛的设计模式。 (3) **消息代理**：微服务也可以集成在异步的场景下，通过队列和订阅主题，实现消息的发布和订阅。一个微服务可以是消息的发布者，把消息通过异步的方式发送到队列或者订阅主题下。作为消费者的微服务可以从队列或者主题共获取消息。通过消息中间件把服务之间的直接调用解耦。通常异步的生产者/消费者模式，通过AMQP、MQTT等异步消息规范。 (4) **客户端发现**：使用客户端发现模式时，客户端决定相应服务实例的网络位置，并且对请求实现负载均衡。客户端查询服务注册表，后者是一个可用服务实例的数据库；然后使用负载均衡算法从中选择一个实例，并发出请求。客户端从服务注册服务中查询，其中是所有可用服务实例的库。客户端使用负载均衡算法从多个服务实例中选择出一个，然后发出请求。服务实例的网络位置在启动时被记录到服务注册表，等实例终止时被删除。服务实例的注册信息通常使用心跳机制来定期刷新。客户端发现模式优缺点兼有。这一模式相对直接，除了服务注册外，其它部分无需变动。此外，由于客户端知晓可用的服务实例，能针对特定应用实现智能负载均衡，比如使用哈希一致性。这种模式的一大缺点就是客户端与服务注册绑定，要针对服务端用到的每个编程语言和框架，实现客户端的服务发现逻辑。 (5) **服务端发现模式**：客户端通过负载均衡器向某个服务提出请求，负载均衡器查询服务注册表，并将请求转发到可用的服务实例。如同客户端发现，服务实例在服务注册表中注册或注销。服务端发现模式兼具优缺点。它最大的优点是客户端无需关注发现的细节，只需要简单地向负载均衡器发送请求，这减少了编程语言框架需要完成的发现逻辑。并且，有些部署环境免费提供这一功能。这种模式也有缺点。除非负载均衡器由部署环境提供，否则会成为一个需要配置和管理的高可用系统组件。

### 12. 微服务架构的服务注册

(1) 服务注册表是服务发现的核心部分，是包含服务实例的网络地址的数据库。 (2) 服务注册表需要高可用而且随时更新。客户端能够缓存从服务注册表中获取的网络地址，然而，这些信息最终会过时，客户端也就无法发现服务实例。因此，服务注册表会包含若干服务端，使用复制协议保持一致性。 (3) 服务实例必须在注册表中注册和注销。注册和注销有两种不同的方法：服务实例自己注册，管理服务实例注册的其它系统组件。

### 13. 微服务架构服务注册分类

(1) **自注册方式**： **$$** 当使用自注册模式时，服务实例负责在服务注册表中注册和注销。另外，如果需要的话，一个服务实例也要发送心跳来保证注册信息不会过时。 **$$** **优点**：相对简单，无需其它系统组件。 **$$** **缺点**：把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。(2) **第三方注册模式**： **$$** 服务实例则不需要向服务注册表注册；相反，被称为服务注册器的另一个系统模块会处理。服务注册器会通过查询部署环境或订阅事件的方式来跟踪运行实例的更改。一旦侦测到有新的可用服务实例，会向注册表注册此服务。服务管理器也负责注销终止的服务实例。 **$$** **优点**：服务与服务注册表解耦合，无需为每个编程语言和框架实现服务注册逻辑。 **$$** **缺点**：除非该服务内置于部署环境，否则需要配置和管理一个高可用的系统组件。

### 14. 微服务架构的数据去中心化的表现

(1) 单体架构中，不同功能的服务模块都把数据存储在某个中心数据库中。 (2) 微服务方式，多个服务之间的设计相互独立，数据也应该相互独立（比如，某个微服务的数据库结构定义方式改变，可能会中断其它服务）。因此，每个微服务都应该有自己的数据库。

### 15. 微服务架构的数据去中心化核心要点

(1) 每个微服务有自己私有的数据库持久化业务数据 (2) 每个微服务只能访问自己的数据库，而不能访问其它服务的数据库 (3) 某些业务场景下，需要在一个事务中更新多个数据库。这种情况也不能直接访问其它微服务的数据库，而是通过对于微服务进行操作。 (4) 数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可以采用不同的数据库技术（SQL、NoSQL等）。在复杂的业务场景下，如果包含多个微服务，通常在客户端或者中间层（网关）处理

### 16. Docker引擎的特点

(1) Docker使用C/S架构，Client通过接口与Server进程通信实现容器的构建，运行和发布。Client和Server可以运行在同一台集群，也可以通过跨主机实现远程通信。 (2) Docker 镜像（Image）：一个只读的模板。镜像可以用来创建Docker容器，一个镜像可以创建很多容器。Docker提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。镜像（Image）就是一堆只读层（read-only layer）的统一视角。这些只读层，它们重叠在一起。除了最下面一层，其它层都会有一个指针指向下一层。这些层是Docker内部的实现细节，并且能够在docker宿主机的文件系统上访问到。统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角，这样在用户的角度看来，只存在一个文件系统。 (3) 仓库（Repository）：集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。仓库分为公开仓库（Public）和私有仓库（Private）两种形式。最大的公开仓库是Docker Hub，存放了数量庞大的镜像供用户下载。国内的公开仓库包括时速云、网易云等，可以提供大陆用户更稳定快速的访问。当然，用户也可以在本地网络内创建一个私有仓库。当用户创建了自己的镜像之后就可以使用push命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上pull下来就可以了。Docker仓库的概念跟Git类似，注册服务器可以理解为GitHub这样的托管服务。 (4) 容器（container）：Docker利用容器（Container）来运行应用。容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的Linux环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。一个运行态容器被定义为一个可读写的统一文件系统加上隔离的进程空间和包含其中的进程。文件系统隔离技术使得Docker成为了一个非常有潜力的虚拟化技术。一个容器中的进程可能会对文件进行修改、删除、创建，这些改变都将作用于可读写层。 (5) **仓库（Repository）**：集中存放镜像文件的场所。 (6) **Docker CLI**：实现容器和镜像的管理，为用户提供统一的操作界面。

### 17. Docker和VM的比较

(1) 虚拟机的Guest OS即为虚拟机安装的操作系统，它是一个完整操作系统内核；虚拟机的Hypervisor层可以简单理解为一个硬件虚拟化平台，它在Host OS是以内核态的驱动存在的。它们被Docker Engine层所替代。 (2) Docker有着比虚拟机更少的抽象层。由于Docker不需要Hypervisor实现硬件资源虚拟化，运行在Docker容器上的程序直接使用的都是实际物理机的硬件资源。因此在CPU、内存利用率上Docker将会在效率上有优势，具体的效率对比在下几个小节里给出。

### 18. Docker的优势和劣势

**$$** **优势**： (1) **更快速的交付和部署**：Docker在整个开发周期都可以完美的辅助你实现快速交付。Docker允许开发者在装有应用和服务本地容器做开发。可以直接集成到可持续开发流程中。 (2) **高效的部署和扩容**：Docker容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。这种兼容性可以让用户把一个应用程序从一个平台直接迁移到另外一个。Docker的兼容性和轻量特性可以很轻松的实现负载的动态管理。你可以快速扩容或方便的下线你的应用和服务，这种速度趋近实时。 (3) **更高的资源利用率**：Docker对系统资源的利用率很高，一台主机上可以同时运行数千个Docker容器。容器除了运行其中应用外，基本不消耗额外的系统资源，使得应用的性能很高，同时系统的开销尽量小。传统虚拟机方式运行10个不同的应用就要起10个虚拟机，而Docker只需要启动10个隔离的应用即可。 (4) **更简单的管理**：使用Docker，只需要小小的修改，就可以替代以往大量的更新工作。所有的修改都以增量的方式被分发和更新，从而实现自动化并且高效的管理。**$$** **劣势**： (1) **资源隔离方面不如虚拟机**：Docker是利用cgroup实现资源限制的，只能限制资源消耗的最大值，而不能隔绝其他程序占用自己的资源。 (2) **安全性问题**：Docker目前并不能分辨具体执行指令的用户，只要一个用户拥有执行Docker的权限，那么他就可以对Docker的容器进行所有操作，不管该容器是否是由该用户创建。 (3) **快速更新带来的兼容性问题**：Docker目前还在版本的快速更新中，细节功能调整比较大。一些核心模块依赖于高版本内核，存在版本兼容问题。

### 19. Docker的架构

(1) **Docker引擎**：Docker使用C/S架构，Client通过接口与Server进程通信实现容器的构建，运行和发布。Client和Server可以运行在同一台集群，也可以通过跨主机实现远程通信。 (2) **Docker 镜像（Image）**：一个只读的模板。镜像可以用来创建Docker容器，一个镜像可以创建很多容器。Docker提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。镜像（Image）就是一堆只读层（read-only layer）的统一视角。这些只读层，它们重叠在一起。除了最下面一层，其它层都会有一个指针指向下一层。这些层是Docker内部的实现细节，并且能够在docker宿主机的文件系统上访问到。统一文件系统（Union File System）技术能够将不同的层整合成一个文件系统，为这些层提供了一个统一的视角，这样在用户的角度看来，只存在一个文件系统。 (3) **仓库（Repository）**：集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。仓库分为公开仓库（Public）和私有仓库（Private）两种形式。最大的公开仓库是Docker Hub，存放了数量庞大的镜像供用户下载。国内的公开仓库包括时速云、网易云等，可以提供大陆用户更稳定快速的访问。当然，用户也可以在本地网络内创建一个私有仓库。当用户创建了自己的镜像之后就可以使用push命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上pull下来就可以了。Docker仓库的概念跟Git类似，注册服务器可以理解为GitHub这样的托管服务。 (4) **容器（container）**：Docker利用容器（Container）来运行应用。容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。可以把容器看做是一个简易版的Linux环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。容器的定义和镜像几乎一模一样，也是一堆层的统一视角，唯一区别在于容器的最上面那一层是可读可写的。一个运行态容器被定义为一个可读写的统一文件系统加上隔离的进程空间和包含其中的进程。文件系统隔离技术使得Docker成为了一个非常有潜力的虚拟化技术。一个容器中的进程可能会对文件进行修改、删除、创建，这些改变都将作用于可读写层。 (5) **仓库（Repository）**：集中存放镜像文件的场所。 (6) **Docker CLI**：实现容器和镜像的管理，为用户提供统一的操作界面。

### 20. Docker调度工具

因为容器没有操作系统或者hypervisor，容器没有独立运作的能力，所以，它们需要有自己的调度管理工具。它的主要任务就是负责在最合适的主机上启动容器，并且将它们关联起来。它必须能够通过自动的故障转移（fail-overs）来处理错误，并且当一个实例不足以处理计算数据时，它能够扩展容器来解决问题。

### 21. Swarm是什么

Swarm是Docker官方的集群管理和编排工具。一个机器运行了一个Swarm的镜像（就像运行其他Docker镜像一样），它负责调度容器，在图片上鲸鱼代表这个机器。Swarm使用了和Docker标准API一致的API，这意味着在Swarm上运行一个容器和在单一主机上运行容器使用相同的命令。尽管有新的flags可用，但是开发者在使用Swarm的同时并不需要改变它的工作流程。Swarm由多个代理（agent）组成，把这些代理称之为节点（node）。这些节点就是主机，这些主机在启动Docker daemon的时候就会打开相应的端口，以此支持Docker远程API。这些机器会根据Swarm调度器分配给它们的任务，拉取和运行不同的镜像。当启动Docker daemon时，每一个节点都能够被贴上一些标签（label），这些标签以键值对的形式存在，通过标签就能够给予每个节点对应的细节信息。当运行一个新的容器时，这些标签就能够被用来过滤集群。  
Swarm的调度策略： **$$** **random策略**：随机选择节点。一般用于开发测试阶段。 **$$** **spread策略**：默认策略，Swarm优先选择占用资源（如CPU、内存等）最少的节点，能保证集群中所有节点资源的均匀使用。 **$$** **binpack策略**：与spread相反，它的目的是尽可能地填满一个节点，以保证更多空余的节点。

### 22. Mesos组成

(1) **ZooKeeper**：帮助Marathon查找Mesos master的地址，同时它具有多个实例可用，以此应付故障的发生。 (2) **Marathon**：负责启动，监控，扩展容器。 (3) **Mesos master**：给节点分配任务，同时如果某一个节点有空闲的CPU/RAM，它就会通知Marathon。 (4) **Mesos slave**：运行容器，并且报告当前可用的资源。

### 23. K8S组成

(1) **Kubernetes**：Docker容器的编排系统，它使用label和pod的概念来将容器换分为逻辑单元。Pods是同地协作（co-located）容器的集合，这些容器被共同部署和调度，形成了一个服务，这是Kubernetes和其他两个框架的主要区别。相比于基于相似度的容器调度方式（就像Swarm和Mesos），这个方法简化了对集群的管理。 (2) **Kubernetes调度器**：任务就是寻找那些PodSpec.NodeName为空的pods，然后通过对它们赋值来调度对应集群中的容器。相比于Swarm和Mesos，Kubernetes允许开发者通过定义PodSpec.NodeName来绕过调度器。调度器使用谓词（predicates）和优先级（priorities）来决定一个pod应该运行在哪一个节点上。通过使用一个新的调度策略配置可以覆盖掉这些参数的默认值。 **$$** **谓词**：是强制性的规则，它能够用来调度集群上一个新的pod。如果没有任何机器满足该谓词，则该pod会处于挂起状态，直到有机器能够满足条件。 **$$** **Predicate**：节点的需求。 **$$** **PodFitPorts**：没有任何端口冲突。 **$$** **PodFitsResource**：有足够的资源运行。 **$$** **PodNoDiskConflict**：有足够的空间来满足pod和链接的数据卷。 **$$** **MatchNodeSelector**：能够匹配pod中的选择器查找参数。 **$$** **HostName**：能够匹配pod中的host参数。

# Chapter4-1

### 1. 数据的定义

(1)**一般而言**，数据是指对客观事件进行记录并可以鉴别的符号，是对客观事物的性质、状态以及相互关系等进行记载的物理符号或这些物理符号的组合。它是可识别的、抽象的符号. (2)**表示形态上**，数据可以是狭义上的数字，可以是具有一定意义的文字、字母、数字符号的组合、图形、图像、视频、音频等，也可以是客观事物的属性、数量、位置及其相互关系的抽象表示. (3)**在计算机科学**中，数据是指所有能输入到计算机并被计算机程序处理的符号的介质的总称，是用于输入电子计算机进行处理，具有一定意义的数字、字母、符号和模拟量等的通称.

### 2. 数据与信息的联系和区别

(1)数据是信息的表达、载体，信息是数据的内涵，是形与质的关系. (2)数据本身没有意义，数据只有对实体行为产生影响时才成为信息.

### 3. 数据与语义的关系

(1)数据的表现形式还不能完全表达其内容，需要经过解释，数据和关于**数据的解释**是不可分的. (2)**数据的解释**是指对数据含义的说明，数据的含义称为数据的语义，数据与其**语义**是不可分的.

### 4. 数据的软件价值

数据+语义+逻辑=业务. 代码+业务=软件应用系统.

### 5. 数据带来的变化

(1)**数据**：海量离散. (2)**语义**：异构. (3)**逻辑**：复杂多用户多片段. (4)**业务**：单机到互用性到跨界，自定义用户数据，loTdata. (5)**软件应用系统**：面向海量数据的高并发复杂分布式异构系统.

### 6. 数据架构的变化

(1)单机MySQL. (2)Memcached+Mysql+垂直分离. (3)Mysql主从读写分离. (4)分库分表+水平拆分+Mysql集群. (5)Nosql.

### 7. 单机Mysql的问题

(1)数据量过大，**单机存储**不下. (2)数据量过大，单机内存**存不下数据索引**,索引是加速数据库访问效率的一种机制，存储于内存，数据量越大，索引越大，内存消耗越多，每当加入数据，都需要维持索引，不做优化，数据访问将非常缓慢. (3)访问量过大，在**读写混合的情况**下，一个实例不能承受.

### 8. 缓存原则

频繁被访问的数据可以被放置于缓存当中，以供频繁访问.

### 9. 垂直分离

按业务划分将不同的数据**放在不同的数据库**里.

### 10. Mysql主从复制和读写分离做法

(1)**主从复制**：容灾备份，缓存备份，保证数据的完整性. (2)**读写分离**：增删改是写，查为读。“读”的职能允许从被查询的数据库服务器读数据。“写”的职能允许向写数据的数据库服务器写数据。分工明确，结合缓存能实现性能的一大提升.

### 11. 为什么使用分库分表+水平拆分+Mysql集群

(1)数据库中的数据量不一定是可控的，在未进行分库分表的情况下，随着时间和业务的发展，库中的**表会越来越多**，表中的数据量也会越来越大，相应地，数据操作，增删改查的**开销也会越来越大**.(2)另外，由于无法进行分布式部署，而一台**服务器的资源**（CPU、磁盘、内存、IO等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈.

### 12. 云时代对DB的新需求

(1)**高性能** - 对数据库高并发读写的需求。数据库并发负载非常高，往往要达到每秒上万次读写请求，硬盘IO无法承受. (2)**海量存储** - 对海量数据的高效率存储和访问的需求。对海量数据进行SQL查询，效率极其低下不可忍受. (3)**高伸缩性与高可用性** - 对数据库的高可扩展性和高可用性的需求。无法通过简单添加更多的硬件和服务节点来扩展性能和负载能力，需要停机维护和数据迁移.

### 13. 数据层非功能需求与架构技术

(1)**存储高性能**。读写分离、数据缓存、分库分表、NoSQL. (2)**存储高可用**。主从、CAP理论. (3)**存储高扩展**。分库分表、NoSQL.

### 14. 读写分离基本原理

将数据库读写操作**分散到不同的节点**上. (1)数据库服务器搭建主从集群，一主一从、一主多从都可以. (2)数据库主机负责读写操作，从机只负责读操作. (3)数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据. (4)业务服务器将写操作发给数据库主机，将读操作发给数据库从机.

### 15. 主丛复制延迟问题的定义

如果业务服务器将数据写入到数据库主服务器后立刻进行读取，此时读操作访问的是从机，主机还没有将数据复制过来，到从机读取数据是读不到最新数据的，业务上就可能出现问题.

### 16. 应对复制延迟的方案

(1)写操作后的读操作指定发给数据库主服务器. (2)读从机失败后再读一次主机. (3)关键业务读写操作全部指向主机，非关键业务采用读写分离.

### 17. 分配机制的分类

程序代码封装或中间件封装.

### 18. 主备复制实现

(1)主机存储数据，通过复制通道将数据复制到备机. (2)正常情况下，客户端无论读写操作，都发送给主机，备机不对外提供任何读写服务. (3)主机故障情况下，客户端不会自动将请求发给备机，此时整个系统处于不可用状态，不能读写数据，但数据并没有全部丢失，因为备机上有数据. (4)如果主机能够恢复（人工或自动），客户端继续访问主机，主机继续将数据复制给备机. (5)如果主机不能恢复，则需要人工升级备机为主机，增加新备机，切换访问链路. (6)主机不能恢复的情况下，成功写入主机但还没有复制到备机的数据会丢失，需要人工进行排查和恢复，也许有的数据就永远丢失了，业务上需要考虑如何应对此类风险. (7)如果主从间数据复制延迟，由于备机并不对外提供读写操作，因此对业务没有影响，但如果延迟较多，恰好此时主机又宕机了，则可能丢失较多数据，因此对于复制延迟也不能掉以轻心。一般的做法是做复制延迟的监控措施，当延迟数据量较大时及时预警，由人工干预处理.

### 19. 主从复制优缺点分析

**优点**：（1)对于客户端来说，不需要感知备机的存在，即使灾难恢复后，原来的备机被人工修改为主机后，对于客户端来说，只是认为主机的地址换了，无需知道是原来的备机升级为主机了. (2)对于主机和备机来说，双方只需要进行数据复制即可，无须进行状态判断和主备倒换等复杂操作. **缺点**：（1)备机仅是备份，不提供读写操作，硬件浪费. (2)故障后需人工干预，无法自动恢复.

### 20. 主备复制和主从复制设计的关键

(1)主备间状态判断：状态传递方式. 状态检测原因. (2)倒换决策：倒换时机. 倒换策略. 自动程度. (3)数据冲突.

### 21. 互连式架构定义

主备机直接建立状态传递的渠道. (1)可以是网络连接（如，各开一个端口），也可以是非网络连接（用串口线连接）. (2)可以是主机发送状态给备机，也可以是备机拉取主机的状态. (3)可以和数据复制通道共用，也可以独立一条通道. (4)状态传递通道可以是一条，也可以是多条，还可以是不同类型的通道混合（如，网络+串口）.

### 22. 如何充分利用主备自动倒换方案

(1)主备机共享一个对于客户端来说唯一的地址（如，虚拟IP）. (2)客户端记录主备机各自的IP，备机具有拒绝服务的能力.

### 23. 互连式架构缺点

(1)状态传递通道本身故障了，则备机会主动升级为主机. (2)虽然可以通过多通道来降低通道故障的机率，但是通道越多，后续的状态决策越复杂，特别是容易收到多种矛盾的信息.

### 24. 中介式架构定义

在主备机之间引入第三方中介，主备机之间不直接连接，而都去连接中介，并且通过中介来传递状态信息. **优点**：(1)**连接管理**更简单：主备机无须再建立和管理多种类型的状态传递连接通道，只要连接到中介即可，实际上降低了主备机的连接复杂度. (2)**状态决策**更简单：无须考虑多种类型的连接通道获取状态信息如何决策的问题. **缺点**：结构复杂了.

### 25. 中介式架构决策步骤

(1)无论主机还是备机，初始状态都是备机，并且只要与中介断开连接，就将自己降级为备机，因此可能出现双备机的情况. (2)主机与中介断连后，中介能够立刻告知备机，备机将被升级为主机. (3)如果是网络中断导致主机与中介断连，主机自己会降级为备机，网络恢复后，旧的主机以新的备机身份向中介上报自己的状态. (4)如果是掉电重启或者进程重启，旧的主机初始状态为备机，与中介恢复连接后，发现已经有主机了，保持自己备机状态不变. (5)主备机与中介连接都正常的情况下，按照实际的状态决定是否进行倒换，如，响应超时.

### 26. 模拟式架构定义

主备机之间并不传递任何状态数据，而是备机模拟成为一个客户端，向主机发起模拟的读写操作，根据读写操作的响应情况来判断主机的状态.

### 27. 模拟式架构与互连对比

**模拟式**相比**互连式**的优缺点: (1)实现更加简单，省去了状态传递通道的建立和管理工作.; (2)模拟式读写操作获取状态信息只有响应信息（如，HTTP404，超时，响应时间超过3s等），没有互连式那么多样，基于有限的状态来做状态决策，可能出现偏差.

### 28. 主主复制定义及做法

两台机器都是主机，互相将数据复制给对方，客户端可以任意挑选其中一台进行读写操作. (1)两台主机都存储数据，通过复制通道将数据复制到另一台主机. (2)正常情况下，客户端可以将读写操作发送给任意一台主机. (3)一台主机故障情况下，客户端只需要将读写操作发送给主机B即可，反之亦然. (4)如果故障主机能够恢复，则客户端继续访问两台主机，两台主机继续相互复制对方数据. (5)如果故障主机不能恢复，则需要人工操作，增加一台新的机器为主机. (6)原有故障主机不能恢复的情况下，成功写入原有故障主机但没有复制到正常主机的数据会丢失. (7)如果两台主机间复制延迟，则可能出现客户端刚写入的数据，在另一台主机上读取不到.

### 29. 主主复制的优缺点

**优点**：(1)两台主机，无倒换概念. (2)客户端无须区分主备机身份. (3)必须保证数据能够双向复制，然而很多数据无法双向复制. **缺点**：数据双向复制复杂，容易产生数据冲突.

### 30. 数据集中集群的定义

数据集中集群称为一主多备/从。数据都只能往主机写，而读操作可以参考主备，主从的架构进行灵活变化.

### 31. 数据集中集群的特点

**复杂度高**：（1)多备即多通道，增加了主机的复制压力，同时增加了对正常读写的压力（实践中，需要考虑降低该压力）; (2)多通道，情况不一，容易导致数据不一致，需要在备机之间进行数据一致性检查和修正; (3)多备对单主状态的检测结果不一致，容易出现不同的判断和决策; (4)单主多备，当主机宕机，如何重新选主，需要算法.

### 32. 数据分散集群的定义

数据分散集群指多个服务器组成一个集群，每台服务器都会负责存储一部分数据，同时，为了提升硬件利用率，每台服务器又会备份一部分数据.

### 33. 数据分散集群的特点

数据分散集群的复杂度在于如何将数据分配到不同的服务器上，需要思考的方面有： (1)**均衡性**：保证数据分区基本均衡. (2)**容错性**：部分服务器故障后，这些服务器上的数据分区需要分配给其他服务器. (3)**可伸缩性**：当集群容量不够，扩充新的服务器后，算法能够自动将数据分区迁移到新服务器，并保证扩容后所有服务器的均衡性.

### 34. 数据分散集群中的特别角色

数据分散集群中，必须要有一个角色来负责执行数据分配算法，可以是独立服务器，如HDFS架构. 也可以是集群选举出的服务器，也称之为“主机”，但职责完全不同，如Elasticsearch.

### 35. 数据集中集群VS数据分散集群

(1)**写数据角色**。数据集中集群架构中，客户端只能将数据写到主机. 数据分散集群架构中，客户端可以向任意服务器中读写数据. (2)**应用场景**。数据集中集群适合数据量不大，集群机器数量不多的场景，如ZooKeeper集群，一般个位数机器. 数据分散集群，由于其良好的可伸缩性，适合业务数据量巨大、集群机器数量庞大的业务场景，如Hadoop集群，可达上千台服务器.

# Chapter4-2

### 1. 数据库和数据库实例区别与联系

**区别**： (1) 数据库是文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合； (2) 数据库实例是程序，是位于用户和操作系统之间的一层数据管理软件，用户对数据库中的数据做任何的操作，包括数据定义、数据查询、数据维护、数据库运行控制等等都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库进行交互。**联系**：数据库是由文件组成（一般来说都是二进制文件）的，一般不可能直接对这些二进制文件进行操作，以实现数据库的SELECT、UPDATE、INSERT和DELETE操作，需要数据库实例来完成对数据库的操作。

### 2. 分库分表的定义

分库分表的本质是数据拆分，是对数据进行分而治之的通用概念。为了分散数据库的压力，采用分库分表将一个表结构分为多个表，或者将一个表的数据分片后放入多个表，这些表可以放在同一个库里，也可以放到不同的库里，甚至可以放在不同的数据库实例上。

### 3. 数据拆分分类

(1) 垂直拆分：根据业务的维度，将原本的一个库（表）拆分为多个库（表），每个库（表）与原有的结构不同。 (2) 水平拆分：根据分片（sharding）算法，将一个库（表）拆分为多个库（表），每个库（表）依旧保留原有的结构。

### 4. 分库分表的发展阶段

(1) 单库单表。 (2) 单库多表:单表内数据越来越多，查询性能下降，有锁表的可能，并阻塞所有其他的操作 (3) 多库多表:单台数据库的存储空间不够用，增加和减少索引消耗时间过长。对数据库进行水平切分，将切分的数据库和表水平地分散到不同的数据库实例上。

### 5. 分库分表的操作时机

(1) 如果在数据库中表的数量达到了一定量级，则需要进行分表，分解单表的大数据量对索引查询带来的压力，并方便对索引和表结构的变更 (2) 如果数据库的吞吐量达到了瓶颈，就需要增加数据库实例，利用多个数据库实例来分解大量的数据库请求带来的系统压力 (3) 如果希望在扩容时对应用层的配置改变最少，就需要在每个数据库实例中预留足够的数据库数量

### 6. 客户端分片的定义

客户端分片就是使用分库分表的数据库的应用层直接操作分片逻辑，分片规则需要在同一个应用的多个节点间进行同步，每个应用层都嵌入一个操作切片的逻辑实现，一般通过依赖Jar包来实现。

### 7. 客户端分片实现方式

应用层/定制JDBC/ORM框架

### 8. 应用层实现方式

(1) 直接在应用层读取分片规则，然后解析分片规则，据此实现切分的路由逻辑，从应用层直接决定每次操作应该使用哪个数据库实例、库、表等。 (2) 需要侵入业务，但实现简单，适合快速上线，切分逻辑由开发者自行定义，容易调试维护。但要求开发者既要实现业务逻辑，还需要实现框架需求。 (3) 该实现方式会让数据库保持的连接比较多，对整体应用服务器池的维护将造成压力。

### 9. 定制JDBC实现方式

(1) 可让开发者集中精力实现业务逻辑，无须关心分库分表的实现。 (2) 通过定制JDBC协议来实现，也就是针对业务逻辑层提供与JDBC一致的接口，分库分表在JDBC的内部实现。 (3) 开发者需要理解JDBC协议，流行的框架有Sharding JDBC

### 10. 通过定制ORM实现

分片规则实现到ORM框架（介于上述两者间）中或者通过ORM框架支持的扩展机制来完成分库分表的逻辑。

### 11. 代理分片定义和好处坏处

代理分片就是在应用层和数据库层之间增加一个代理层，把分片的路由规则配置在代理层，代理层对外提供与JDBC兼容的接口给应用层。 **好处**：应用层的开发人员不用关心分片规则，只需关心业务逻辑的实现，待业务逻辑实现之后，在代理层配置路由规则即可。 **坏处**： (1) 代理层的引入增加了一层网络传输，对性能会造成影响。 (2) 需要维护代理层，增加了人员和硬件的成本。

### 12. 垂直切分定义和分类

垂直切分是指按照业务将表进行分类或分拆，将其分布到不同数据库上。 **分类**：按业务进行分库和按业务进行分表。

### 13. 分开的不同方法

(1) 不同业务模块的数据可以分散到不同数据库服务器。 (2) 也可以冷热分离，根据数据的活跃度将数据进行拆分。冷数据：变化更新频率低，查询次数多的数据。热数据：变化更新频率高，活跃的数据。 (3) 也可以人为将一个表中的内容划分为多个表，例如将查询较多，变化不多的字段拆分成一张表放在查询性能高的服务器，而将频繁更新的字段拆分并部署到更新性能高的服务器。

### 14. 垂直切分优缺点

**优点**：(1) 拆分后业务清晰，拆分规则明确 (2) 系统之间进行整合或扩展很容易 (3) 按照成本、应用的等级或类型等将表放到不同的机器上，便于管理 (4) 便于实现动静分离、冷热分离的数据库表的设计模式 (5) 数据维护简单**缺点**：(1) 部分业务表无法关联（Join），只能通过接口方式解决，提高了系统的复杂度 (2) 受每种业务的不同限制，存在单库性能瓶颈，不易进行数据扩展和提升性能 (3) 事务处理复杂

### 15. 水平切分定义

水平切分不是将表进行分类，而是将其按照某个字段的某种规则分散到多个库中，在每个表中包含一部分数据，所有表加起来是全量数据。简言之，将数据按一定规律，按行切分，并分配到不同的库表里，表结构完全一样。

### 16. 水平切分优缺点

**优点**：(1) 单库单表的数据保持在一定的量级，有助于性能的提高 (2) 切分的表的结构相同，应用层改造较少，只需要增加路由规则即可 (3) 提高了系统的稳定性和负载能力**缺点**：(1) 切分后，数据是分散的，很难利用数据库的Join操作，跨库Join性能差 (2) 拆分规则难以抽象 (3) 分片事务的一致性难以解决 (4) 数据扩容的难度和维护量极大

### 17. 垂直/水平切分的共同点和偏向

**共同点**(1) 存在分布式事务的问题 (2) 存在跨节点Join的问题 (3) 存在跨节点合并排序、分页的问题 (4) 存在多数据源管理的问题**偏向**(1) 垂直切分更偏向于业务拆分的过程 (2) 水平切分更偏向于技术性能指标

### 18. 水平切分的路由过程

(1) 分库分表后，数据将分布到不同的分片表中，通过分库分表规则查找到对应的表和库的过程叫做路由。 (2) 我们在设计表时需要确定对表按照什么样的规则进行分库分表。

### 19. 水平切分的分片维度

(1) 按哈希切片。对数据的某个字段求哈希，再除以分片总数后取模，取模后相同的数据为一个分片，这样将数据分成多个分片。**好处**：数据切片比较均匀，对数据压力分散的效果较好。**缺点**：数据分散后，对于查询需求需要进行聚合处理 (2) 按照时间切片。按照时间的范围将数据分布到不同的分片。

### 20. 分库分表引起的问题

(1) 扩容与迁移问题。 (2) 查询问题 (3) 分布式事务问题。多库多表分布式所引发的一致性问题。 (4) 同组数据跨库问题。要尽量把同一组数据放到同一台数据库服务器上，不但在某些场景下可以利用本地事务的强一致性，还可以使这组数据实现自治。

### 21. 扩容与迁移

(1) 按照新旧分片规则，对新旧数据库进行双写 (2) 将双写前按照旧分片规则写入的历史数据，根据新分片规则迁移写入新的数据库 (3) 将按照旧的分片规则查询改为按照新的分片规则查询 (4) 将双写数据库逻辑从代码中下线，只按照新的分片规则写入数据 (5) 删除按照旧分片规则写入的历史数据

### 22. 扩容与迁移的问题

(一)**数据一致性问题**。（1）由于数据量大，通常会造成不一致问题，因此，通常先清理旧数据，洗完后再迁移到新规则的新数据库下，再做全量对比。还需要对比评估迁移过程中是否有数据更新，如果有需要迭代清洗，直至一致。（2）如果数据量巨大，无法全量对比，需要抽样对比，抽样特征需要根据业务特点进行选取。（3）线上记录迁移过程中的更新操作日志，迁移后根据更新日志与历史数据共同决定数据的最新状态，以达到迁移数据的最终一致性。（二）**动静数据分离问题**。对于一些动静敏感的数据，如交易数据，最好将动静数据分离。选取时间点对静历史数据进行迁移。

### 23. 查询问题定义和解决方案

在分库分表以后，如果查询的标准是分片的主键，则可以通过分片规则再次路由并查询，但是对于其他主键的查询、范围查询、关联查询、查询结果排序等，并不是按照分库分表维度来查询的。**查询问题的解决方案**：(1) 在多个分片表查询后合并数据集（效率很低） (2) 按查询需求定义多分片维度，形成多张分片表（空间换时间） (3) 通过搜索引擎解决，如果有实时要求，还需要实时搜索。（难度大）

# Chapter4-3

### 1. 缓存的概念，作用和原理

**$$** **概念** ① 用于存储数据的硬件或软件组件，以使得后续更快访问响应的数据。 ② 缓存中的数据可能是提前计算好的结果、数据的副本等.**$$** **作用** ① 主要解决高并发，热点数据访问的性能问题。 ② 提供高性能的数据快速访问。**$$** **原理** ① 将数据写入到读取速度更快的存储 ② 将数据缓存到离应用最近的位置 ③ 将数据缓存到离用户最近的位置

### 2. 数据缓存的存储介质分类

**$$** **硬件介质上来看**：内存和硬盘**$$** **技术上**：内存，硬盘文件和数据库 ① 内存：缓存于内存中是最快的选择，无需额外的I/O开销，但是内存的缺点是没有持久化到物理磁盘，一旦应用异常break down而重新启动，数据很难或者无法复原。 ② 硬盘：一般来说，很多缓存框架会结合使用内存和硬盘，在内存分配空间满了或是在异常的情况下，可以被动或主动的将内存空间数据持久化到硬盘中，达到释放空间或备份数据的目的。 ③ 数据库：非传统数据库，而是key-value存储结构的特殊数据库（如BerkeleyDB和Redis），响应速度和吞吐量都远高于关系型数据库等。

### 3. 数据缓存的基本类型

(1) 本地缓存 (2) 分布式缓存 (3) 反向代理缓存 (4) CDN缓存

### 4. 本地缓存的定义，缺点，应用场景以及缓存介质和实现方法

**$$** **定义**：本地缓存指的是在应用中的缓存组件，其最大的优点是应用和cache是在同一个进程内部，请求缓存非常快速，没有过多的网络开销等，在单应用不需要集群支持或者集群情况下各节点无需互相通知的场景下使用本地缓存较合适；**$$** **缺点**：因为缓存跟应用程序耦合，多个应用程序无法直接的共享缓存，各应用或集群的各节点都需要维护自己的单独缓存，对内存是是一种浪费。**$$** **应用场景**：缓存字典等常用数据。**$$** **缓存介质**：① 硬盘缓存：将数据缓存到硬盘，减少网络传输的消耗 ② 内存缓存：直接将数据存储到本级内存中，通过程序直接维护缓存对象。**$$** **实现方法**：应用编码；中间件，如Ehcache、 Guava Cache等

### 5. 分布式缓存的定义，应用场景和常用中间件

**$$** **定义**：分布式缓存指的是与应用分离的缓存组件或服务，其最大的优点是自身就是一个独立的应用，与本地应用隔离，多个应用可直接的共享缓存。**$$** **应用场景**： ① 缓存经过复杂运算得出的数据 ② 缓存存储系统中频繁访问的热点数据，减轻存储系统压力**$$** **常用中间件**：Memcached，Redis

### 6. 反向代理缓存的定义，应用场景和开源实现

**$$** **定义**：反向代理位于应用服务器机房，处理所有对WEB服务器的请求。如果用户请求的页面在代理服务器上有缓冲的话，代理服务器直接将缓冲内容发送给用户。如果没有缓冲则先向WEB服务器发出请求，取回数据，本地缓存后再发送给用户。通过降低向WEB服务器的请求数，从而降低了WEB服务器的负载。**$$** **应用场景**：一般只缓存体积较小的静态文件资源，如css、js、图片**$$** **开源实现**：Varnish，Nginx和Squid

### 7. CDN缓存的定义，目标和做法

**$$** **定义**：CDN (Content Delivery Network)：内容分发网络。通过在现有互联网中增加一层新的网络架构（CDNS），将网站内容发布到最接近用户的网络“边缘”，使用户可以就近取得所需的内容**$$** **目标**：解决由于网络带宽小、用户访问量大、网点分布不均等原因所造成的用户访问网站响应速度慢的问题。**$$** **做法**：将一个服务器的内容平均分布到多个服务器上；智能识别服务器，让用户获取离用户最近的服务器，提高访问速度。 ① 分布式存储 ② 负载均衡 ③ 网络请求的重定向 ④ 内容管理

### 8. 数据缓存的两个术语：命中列和最大元素/最大空间

**$$** **命中率**① 命中率=返回正确结果数/请求缓存次数 ② 命中率问题是缓存中的一个非常重要的问题，它是衡量缓存有效性的重要指标。命中率越高，表明缓存的使用率越高。 ③ 缓存的管理者希望缓存命中率接近 100%。而实际得到的命中率则与缓存的大小、缓存用户兴趣点的相似性、缓存数据的变化或个性化频率，以及如何配置缓存有关。 ④ 命中率很难预测，但对现在中等规模的Web缓存来说，40%的命中率是很合理的。**$$** **最大元素**① 缓存中可以存放的最大元素的数量，一旦缓存中元素数量超过这个值（或者缓存数据所占空间超过其最大支持空间），那么将会触发缓存启动清空策略 ② 根据不同的场景合理的设置最大元素值往往可以一定程度上提高缓存的命中率，从而更有效的使用缓存。

### 9. HTTP再验证的定义，时机和实现

**$$** **定义**：HTTP 再验证 (revalidation)：原始服务器的内容可能会发生变化，缓存要不时对其进行检测，看看它们保存的副本是否仍是服务器上最新的副本，进行“新鲜度检测”。**$$** **时机**：缓存可以在任意时刻，以任意的频率对副本进行再验证。但由于缓存中通常会包含数百万的文档，而且网络带宽是很珍贵的，所以大部分缓存只有在客户端发起请求，并且副本旧得足以需要检测的时候，才会对副本进行再验证。**$$** **实现**：① 为了有效地进行再验证，HTTP 定义了一些特殊的请求，不用从服务器上获取整个对象，就可以快速检测出内容是否是最新的。 ② HTTP 为我们提供了几个用来对已缓存对象进行再验证的工具，但最常用的是 If-Modified-Since 首部。将这个首部添加到 GET 请求中去，就可以告诉服务器，只有在缓存了对象的副本之后，又对其进行了修改的情况下，才发送此对象。

### 10. 服务器收到GET-if-modified-sience的3种情况

(1) **再验证命中**(revalidate hit)或缓慢命中(slow hit)：如果服务器对象未被修改，服务器会向客户端发送一个小的 HTTP 304 Not Modified 响应。只要缓存知道副本仍然有效，就会再次将副本标识为暂时新鲜的，并将副本提供给客户端。 (2) **再验证未命中**：如果服务器对象与已缓存副本不同，服务器向客户端发送一条普通的、带有完整内容的 HTTP 200 OK 响应。 (3) **对象被删除**：如果服务器对象已经被删除了，服务器就回送一个 404 NotFound 响应，缓存也会将其副本删除。

### 11. 缓存清空策略的定义和分类

**$$** **定义**：缓存清空策略：在缓存的存储空间有限制，当缓存空间被用满时，既要保证稳定服务，又要有效提升命中率。常见的一般策略有：**$$** **分类**：① **FIFO**(first in first out)：先进先出策略，最先进入缓存的数据在缓存空间不够的情况下（超出最大元素限制）会被优先被清除掉，以腾出新的空间接受新的数据。策略算法主要比较缓存元素的创建时间。在数据实效性要求场景下可选择该类策略，优先保障最新数据可用。 ② **LFU**(less frequently used)：最少使用策略，无论是否过期，根据元素的被使用次数判断，清除使用次数较少的元素释放空间。策略算法主要比较元素的hitCount（命中次数）。在保证高频数据有效性场景下，可选择这类策略。 ③ **LRU**(least recently used)：最近最少使用策略，无论是否过期，根据元素最后一次被使用的时间戳，清除最远使用时间戳的元素释放空间。策略算法主要比较元素最近一次被get使用时间。在热点数据场景下较适用，优先保证热点数据的有效性。 ④ 根据过期时间判断，清理过期时间最长的元素 ⑤ 根据过期时间判断，清理最近要过期的元素 ⑥ 随机清理 ⑦ 根据关键字（或元素内容）长短清理等

### 12. 数据缓存的更新策略分类

(1) **Cache aside**① 失效：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。 ② 命中：应用程序从cache中取数据，取到后返回。 ③ 更新：先把数据存到数据库中，成功后，再让缓存失效。(2) **Read/Write Through Pattern**① Read Through 是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或LRU换出），Cache Aside是由调用方负责把数据加载入缓存，而Read Through则用缓存服务自己来加载，从而对应用方是透明的。 ② Write Through 在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由Cache自己更新数据库（这是一个同步操作）。(3) **Write Behind Caching Pattern**① 俗称write back，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的I/O操作飞快无比，因为异步（比如消息队列），write back还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。 ② 但是这个设计的最大致命问题在于数据的非强一致性，极可能造成数据的丢失。假如使用redis作为缓存数据库，最致命的问题在于redis并不能保证绝对不丢失数据，也就是redis的持久化能力（两种持久化都无法保证数据绝对丢失）不足，redis一旦挂了，可能造成数据丢失且无法恢复。

### 13. 本地缓存的实现方法分类

(1) **编程直接实现**① 成员变量或局部变量实现。以局部变量map结构缓存部分业务数据，减少频繁的重复数据库I/O操作。缺点仅限于类的自身作用域内，类间无法共享缓存。 ② 静态变量实现(2) 中间件**Ehcache**① Ehcache是现在最流行的纯Java开源缓存框架，配置简单、结构清晰、功能强大，是一个轻量级的缓存实现，Hibernate里面集成了相关缓存功能。 ② Ehcache的核心定义主要包括：1. cache manager：缓存管理器，允许多实例2. cache：缓存管理器内可以放置若干cache，存放数据的实质，所有cache都实现了 Ehcache接口，这是一个真正使用的缓存实例；通过缓存管理器的模式，可以在单个应用中轻松隔离多个缓存实例，独立服务于不同业务场景需求，缓存数据物理隔离，同时需要时又可共享使用。

### 14. 分布式缓存的迁移

(1) 平滑迁移 (2) 一致性哈希 (3) 停机迁移

### 15. 一致性哈希的做法

① 一致性哈希将整个哈希值空间组织成一个虚拟的圆环（哈希环）。 ② 将各个服务器使用Hash进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置 ③ 将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走” ，第一台遇到的服务器就是其应该定位到的服务器

### 16. 平滑迁移的做法

#### 1. **双写**

**$$** **目标**：同时向旧缓存和新缓存写数据。**$$** **实现**：在迁移过程中，依旧使用旧的路由规则（如关键字哈希除以2取余），同时应用新的规则（关键字哈希除以4取余）将数据写入到新的缓存集群。此时，前两个新的分片可以重用旧缓存分片的数据，而只新增两个分片处理新的数据。

#### 2. **迁移历史数据**

**$$** **目标**：将旧缓存中的历史数据迁移到新的缓存集群。**$$** **实现**：将历史数据按新的规则写入新集群。这一过程中可能需要开发迁移工具，并验证数据迁移的成功。**$$** **优化**：在某些情况下，如果缓存数据并非强依赖或具有时效性，可以不迁移旧数据，依靠新数据逐步替代。

#### 3. **切读**

**$$** **目标**：将所有读操作转移到新的缓存集群。**$$** **实现**：这一步骤通过切换读取逻辑，将应用中的缓存读取操作改为从新的集群获取。通常无需修改应用代码，只需切换读取开关。

#### 4. **下线双写**

**$$** **目标**：停用向旧缓存集群写数据的逻辑。**$$** **实现**：当确认新缓存集群的读写正常并且数据一致性没有问题时，下线双写操作，并清理旧的缓存分片中的冗余数据。

### 17. 一致性哈希的容错性和可扩展性

① 现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。 ② 如果在系统中增加一台服务器NodeX，此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

### 18. 一致性哈希的缺点和解决办法

**$$** **缺点**：一致性哈希算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜问题。此时必然造成大量数据集中到一个节点上。**$$** **解决办法**：为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。

### 19. 停机迁移的做法及评价

**$$** **做法**① 停机应用，先将应用停止服务 ② 迁移历史数据，按照新的规则把历史数据迁移到新的缓存数据集群中 ③ 更改应用的数据源配置，指向新的缓存集群 ④ 重新启动应用**$$** **评价**：该方式简单，高效，能够有效避免数据的不一致，但需要由业务方评估影响，一般在晚上访问量较小，或者非核心服务的场景下比较适用。

### 20. 缓存导致的问题

(1) 数据一致性 (2) 缓存穿透 (3) 缓存雪崩 (4) 缓存高可用 (5) 缓存热点

### 21. 数据一致性问题的原因和解决办法

**$$** **原因**：因为缓存属于持久化数据的一个副本，所以不可避免的会出现数据不一致问题，如脏读或读不到数据的情况。**$$** **解决办法**① 先写缓存，再写数据库：【描述】缓存写成功，但写数据库失败或响应延迟，则下次读取（并发读）缓存时，就出现脏读；【解决】这个写缓存的方式，本身就是错误的，需要改为先写持久化介质，再写缓存的方式② 先写数据库，再写缓存：【描述】写数据库成功，但写缓存失败，则下次读取（并发读）缓存时，则读不到数据；【解决1】根据写入缓存的响应来进行判断，如果缓存写入失败，则回滚数据库操作。该方法增加了程序的复杂度；【解决2】缓存使用时，假如读缓存失败，先读数据库，再回写缓存③ 缓存异步刷新：【描述】指数据库操作和写缓存不在一个操作步骤中，比如在分布式场景下，无法做到同时写缓存或需要异步刷新；【解决】根据日志中用户刷新数据的时间间隔，以及针对数据可能产生不一致的时间，进行同步操作

### 22. 缓存穿透的原因和解决办法

**$$** **原因**：缓存穿透指的是使用不存在的key进行大量的高并发查询，这导致缓存无法命中，每次请求都要穿透到后端数据库系统进行查询，使得数据库压力过大，甚至导致数据库服务崩溃。**$$** **解决办法**① 通常将空值缓存起来，再次接收到同样的查询请求时，若命中缓存并值为空，就会直接返回，不会透传到数据库，避免缓存穿透② 对恶意的查询攻击，可以对查询条件设置规则，不符合条件产生规则的直接拒绝

### 23. 缓存并发的原因和解决办法

**$$** **原因**：缓存并发的问题通常发生在高并发的场景下，当一个缓存key过期时，因为访问这个缓存key的请求量较大，多个请求同时发现缓存过期，因此多个请求会同时访问数据库来查询最新数据，并且回写缓存，这样会造成应用和数据库的负载增加，性能降低，由于并发较高，甚至会导致数据库崩溃**$$** **解决办法**① **分布式锁**：使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。该方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。 ② **本地锁**：与分布式锁类似，通过本地锁的方式来限制只有一个线程去数据库中查询数据，而其他线程只需等待，等前面的线程查询到数据后再访问缓存。但是，这种方法只能限制一个服务节点只有一个线程取数据库中查询，如果一个服务有多个节点，则会有多个数据库查询操作，也就是说在节点数量较多的情况下并没有完全解决缓存并发的问题。③ **软过期**：（1）软过期指对缓存中的数据设置失效时间，就是不使用缓存服务提供的过期时间，而是业务层在数据中存储过期时间信息，由业务程序判断是否过期并更新，在发现了数据即将过期时，将缓存的时效延长，程序可以派遣一个线程去数据库中获取最新的数据，其他线程会先继续使用旧数据并等待，直至派遣线程获取最新数据后再更新缓存。（2）也可以通过异步更新服务来更新设置软过期的缓存，这样应用层就不用关心缓存并发的问题。

### 24. 缓存雪崩的原因和解决办法

**$$** **原因**：缓存雪崩指缓存服务器重启或者大量缓存集中在某一个时间段内失效，业务系统需要重新生成缓存，给后端数据库造成瞬时负载升高的压力，甚至导致数据库崩溃。**$$** **解决办法**① 更新锁机制：对缓存更新操作进行加锁保护，保证只有一个线程能进行缓存更新② 失效时间分片机制：对不同的数据使用不同的失效时间，甚至对相同的数据、不同的请求使用不同的失效时间③ 后台更新机制：由后台线程来更新缓存，并不是业务线程来更新缓存。（1）后台线程除了定时更新缓存，还要频繁去读取缓存（2）业务线程发现缓存失效后，通过消息队列发送一条消息通知后台线程更新缓存（适合业务刚上线时缓存预热）④ 缓存集群：可以做缓存的主从与缓存水平分片

### 25. 缓存热点的原因和解决办法

**$$** **原因**：一些特别热点的数据，高并发访问同一份缓存数据，导致缓存服务器压力过大。**$$** **解决办法**：复制多份缓存副本，把请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压力

### **Chapter 1 - 系统体系结构的内涵及要点**

#### **Slide 1-1**

1.1 系统体系结构概述

o软件架构 1-5

#### **Slide 1-2**

1.1 系统体系结构概述

o软件架构 6-17

1.2 系统体系结构与中间件

o中间件 18-26

### **Chapter 2 - 软件设计模式基础**

#### **Slide 2-1**

2.1 软件设计模式概述

o设计模式简介 1-2

o设计模式类型 2-7

o设计模式原则 8-9

#### **Slide 2-2**

2.2 常用软件设计模式

单例模式 1-8工厂模式 9-16适配器模式 17-22桥接模式 23-28代理模式 29-37中介者模式 38-43观察者模式 44-50访问者模式 51-55

### **Chapter 3 - 计算层的软架构技术**

#### **Slide 3-1**

3.1 计算层的软架构技术挑战

o计算模式的变量 1-8

o非功能度量指标 9-12

#### **Slide 3-3**

3.3 负载均衡

o负载均衡概论 1-3

oDNS负载均衡 4-5

o硬件负载均衡 6-7

o软件负载均衡 8-11

o负载均衡算法概论 12-13

o(加权)轮询 14-17

o负载均衡最低 18-20

o性能最优 21-22

o Hash 23-24

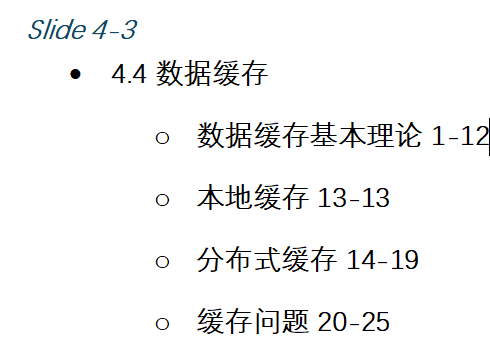
#### **Slide 3-4**

3.4 消息队列

o进程通信模型 1-6

oMOM面向消息中间件&消息队列 7-18

#### **Slide 3-5**

3.5 分布式服务框架

o分布式服务框架 1-6

o微服务 7-15

oDocker 16-23

### **Chapter 4 - 数据层的软架构技术**

#### **Slide 4-1**

4.1 数据驱动的软件架构演化

o数据与软件 1-4

o数据带来的架构变化 5-13

4.2 数据读写与主从分离

o读写分离 14-14

o主备与主从复制 15-21

o主备倒换与主从倒换 22-30

o主主复制 31-32

o数据集群 33-38

#### **Slide 4-2**

4.3 数据分库分表

o分库分表概念 1-5

o分库分表解决方案

客户端分片 6-10

代理分片 11-11

o分库分表架构设计

垂直切分 12-14

水平切分 15-19

分库分表问题 20-23

o分库分表中间件