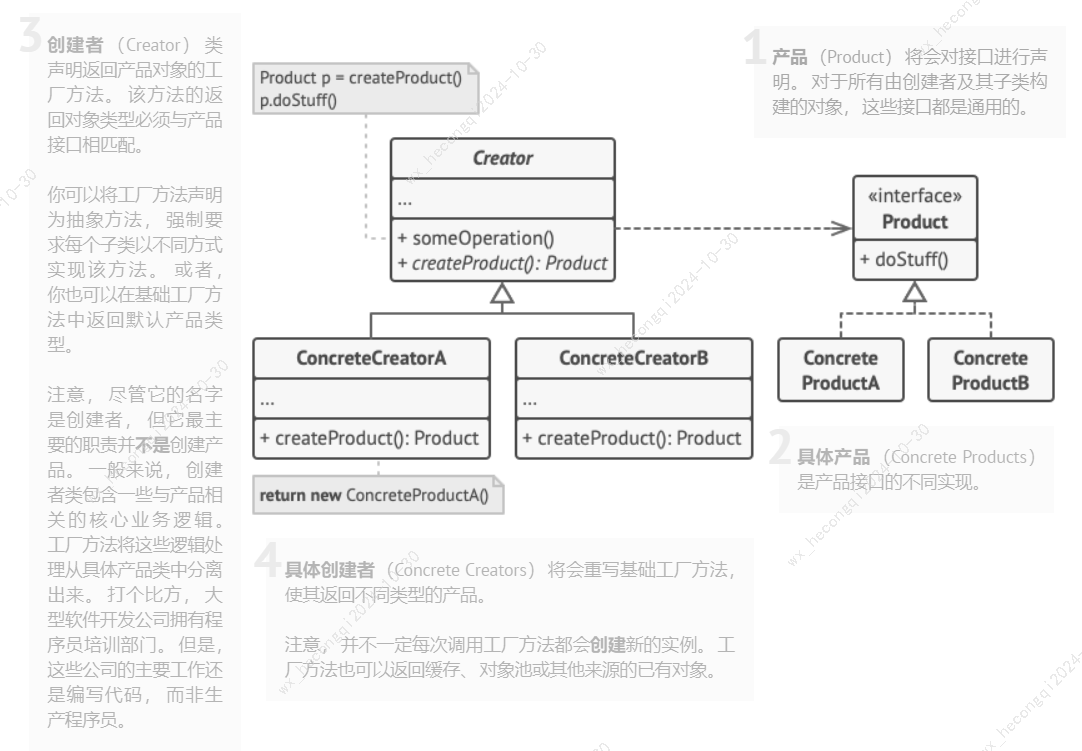
## 编程语言

### 设计模式

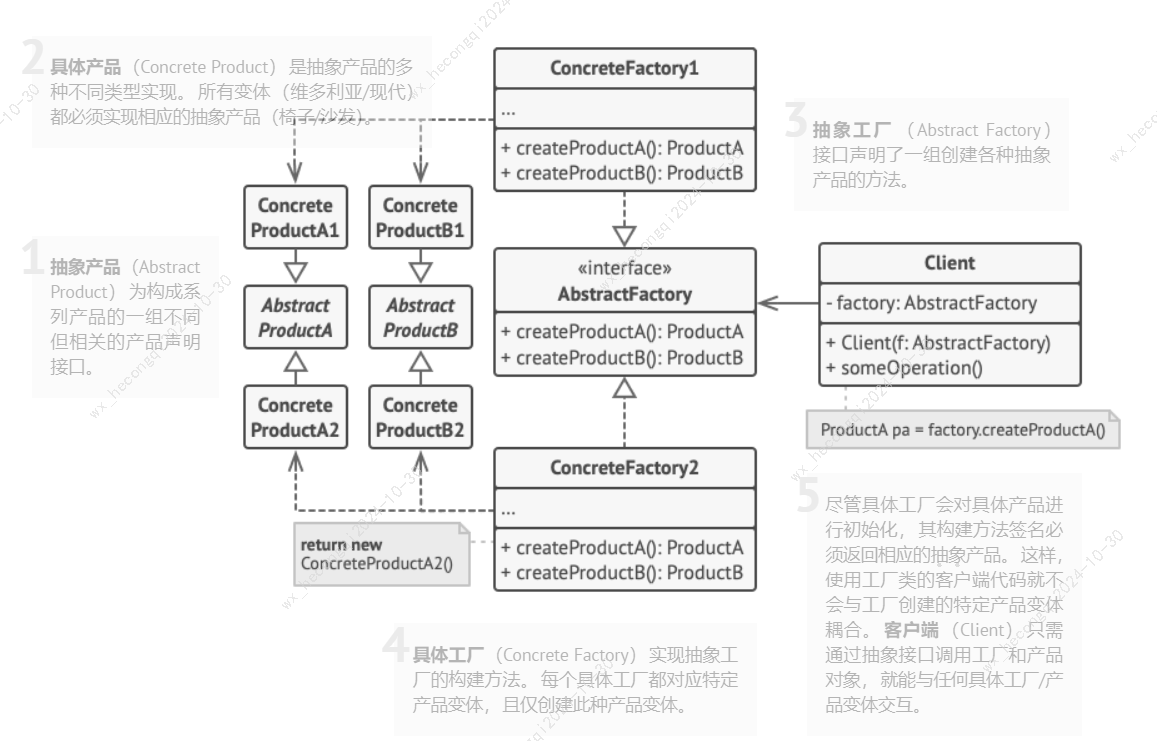
设计模式，往小了看，可以设计一个功能类，往大了说，可以帮你设计系统结构

#### 创建型

* 单例模式
* 原型模式：复制/克隆已有对象
* 生成器模式（Builder）：分步骤创建复杂对象
* 工厂方法模式：父类提供创建对象的方法，子类提供具体实现。

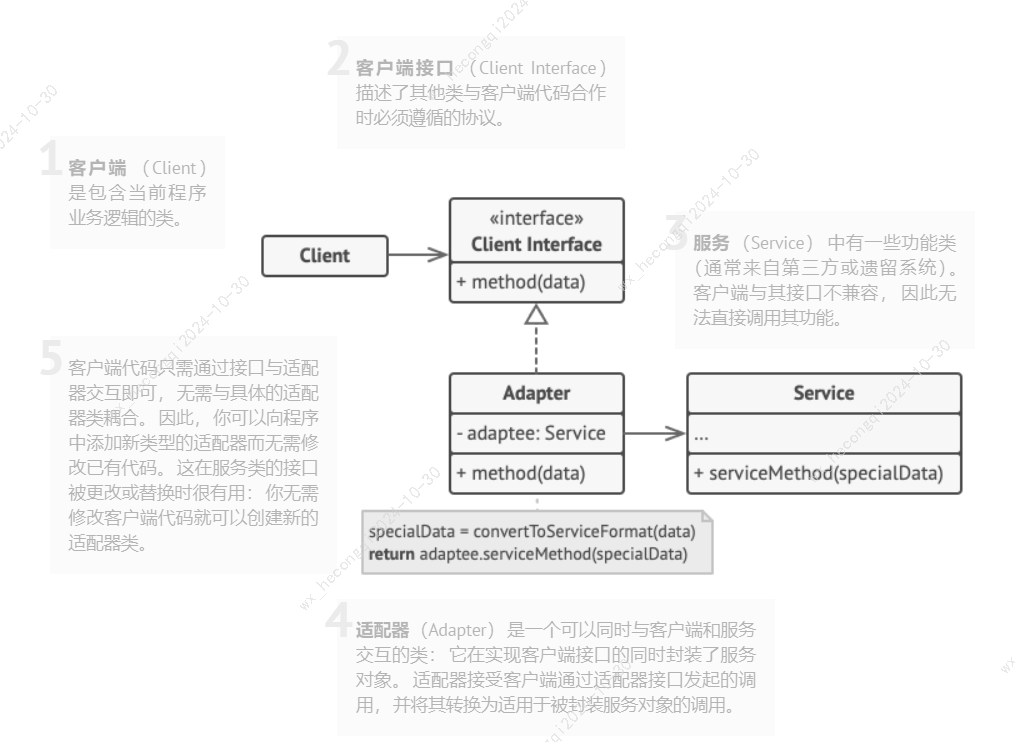


* 抽象工厂模式：与工厂方法相比，抽象工厂会创建多个产品，具体工厂代表具体风格，或者叫做一个系列

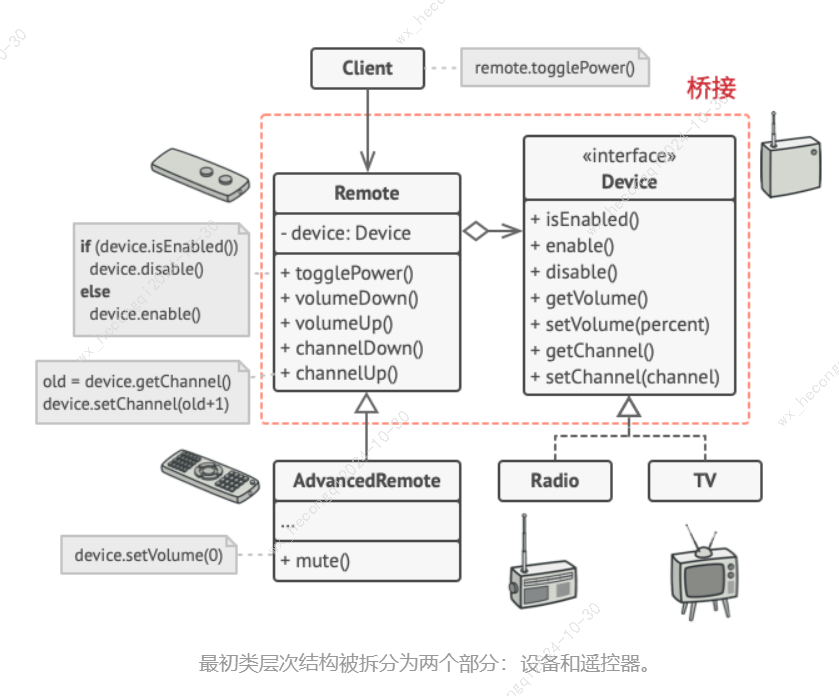


#### 结构型

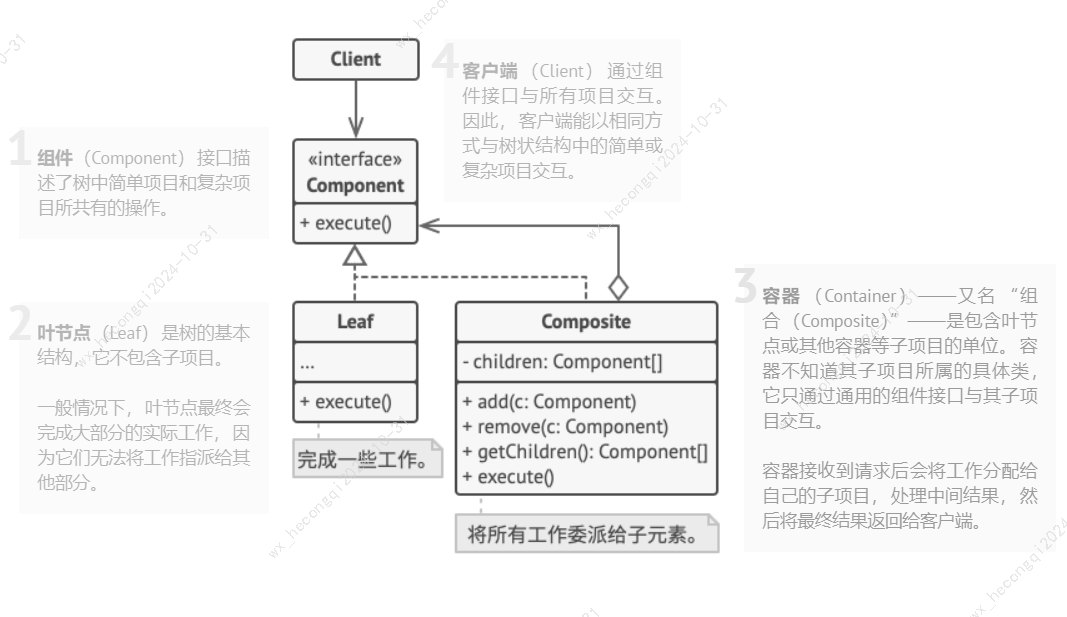
* 适配器模式：使接口不兼容的对象相互合作，Client只能调Client Interface接口的实现类，但又需要Service的能力，Adapter完美实现适配



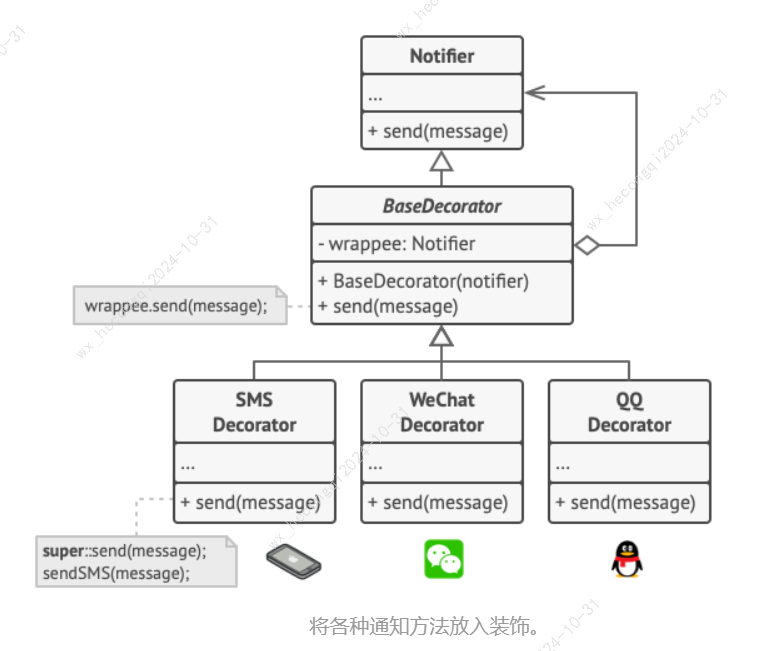
* 桥接模式：将继承改为组合，抽取某个维度成为独立的类层次。考虑如下场景，可以给每个设备都设计一个遥控机，也可以只设计一个遥控器，在遥控器里引入设备对象。



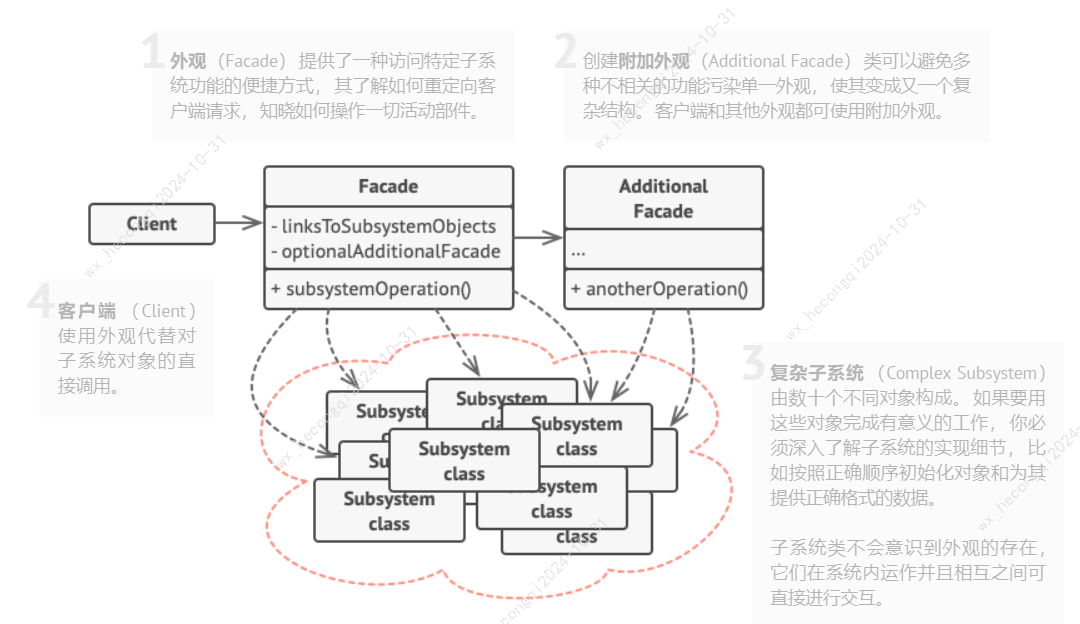
* 组合模式：所有元素共用一个接口，适合树状结构对象



* 装饰模式：不更改对象，同时增加行为



* 外观模式：封装复杂功能，提供简单接口调用



* 享元模式

共享数据，节约内存

Tree.java: 包含每棵树的独特状态

public class Tree {

private int x;

private int y;

private TreeType type;

public Tree(int x, int y, TreeType type) {

this.x = x;

this.y = y;

this.type = type;

}

public void draw(Graphics g) {

type.draw(g, x, y);

}

}

TreeType.java: 包含多棵树共享的状态

public class TreeType {

private String name;

private Color color;

private String otherTreeData;

public TreeType(String name, Color color, String otherTreeData) {

this.name = name;

this.color = color;

this.otherTreeData = otherTreeData;

}

public void draw(Graphics g, int x, int y) {

g.setColor(Color.BLACK);

g.fillRect(x - 1, y, 3, 5);

g.setColor(color);

g.fillOval(x - 5, y - 10, 10, 10);

}

}

TreeFactory.java: 封装创建享元的复杂机制

public class TreeFactory {

static Map<String, TreeType> treeTypes = new HashMap<>();

public static TreeType getTreeType(String name, Color color, String otherTreeData) {

TreeType result = treeTypes.get(name);

if (result == null) {

result = new TreeType(name, color, otherTreeData);

treeTypes.put(name, result);

}

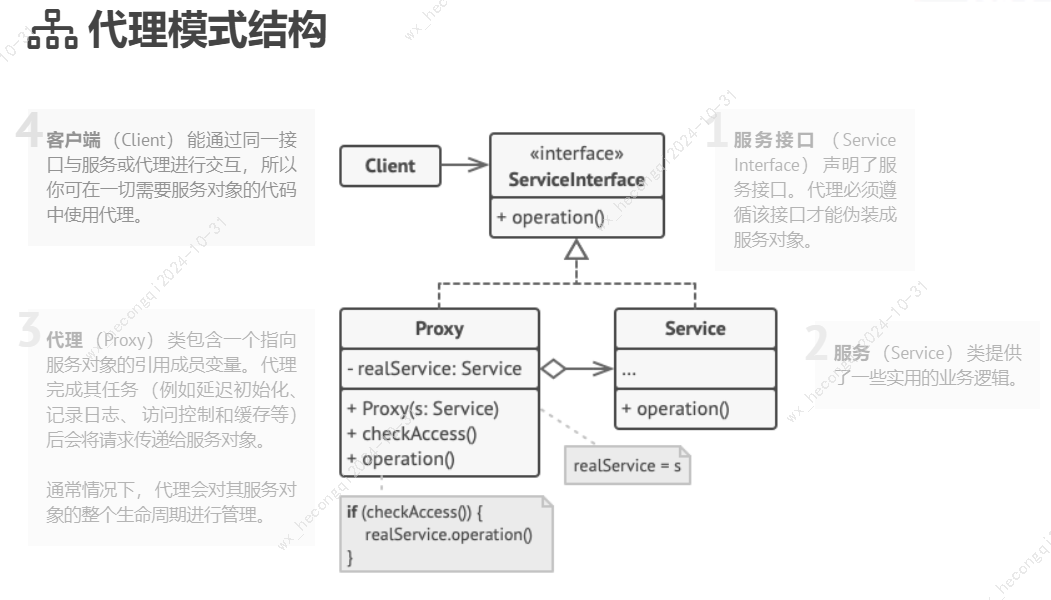
return result;

}

}

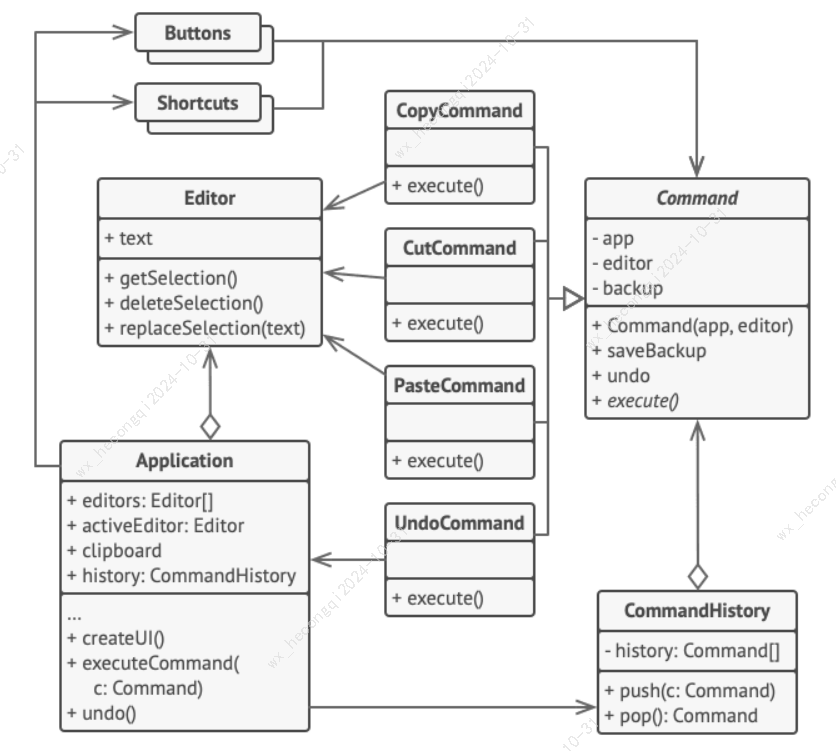
* 代理模式

代理与适配器模式不同，下图中Proxy与Service都是ServiceInterface的子类，而适配器是为了解决接口的不相容问题。

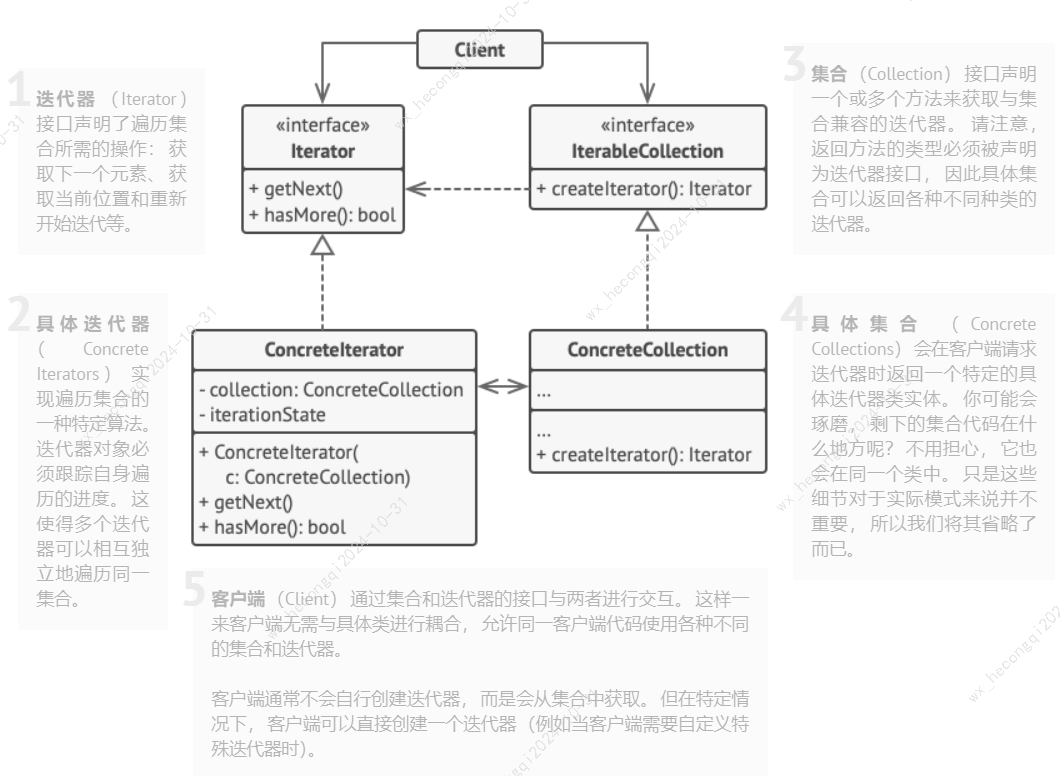


#### 行为型

* 责任链模式：请求通过责任链依次处理
* 命令模式：将请求转换为对象， 可放入队列，实现延迟执行。

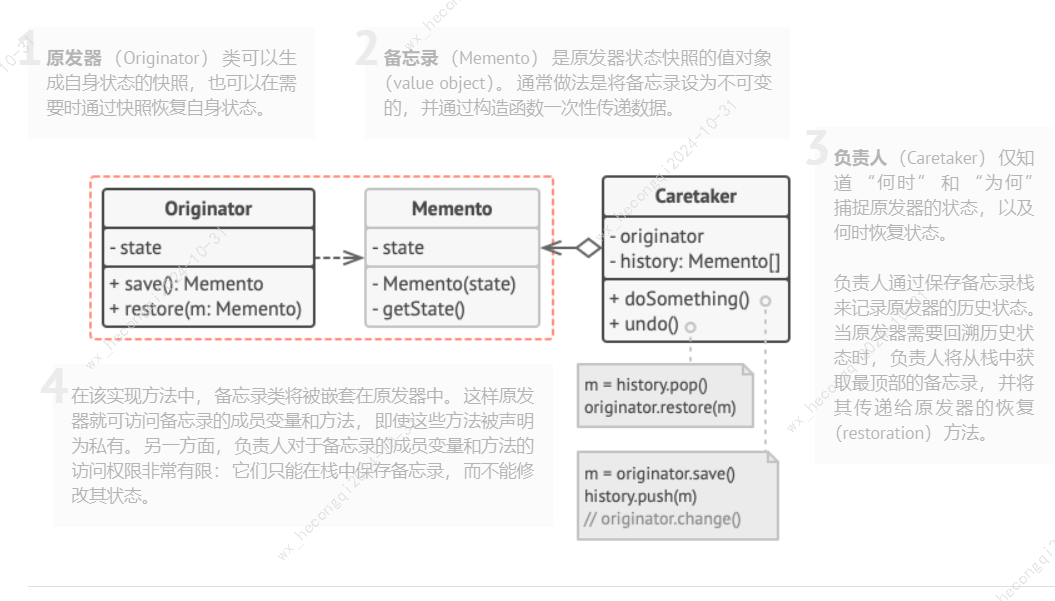


* 迭代器模式：不暴露底层数据（列表、树、图），却能遍历所有元素，比如树可以深度优先遍历或者广度优先遍历

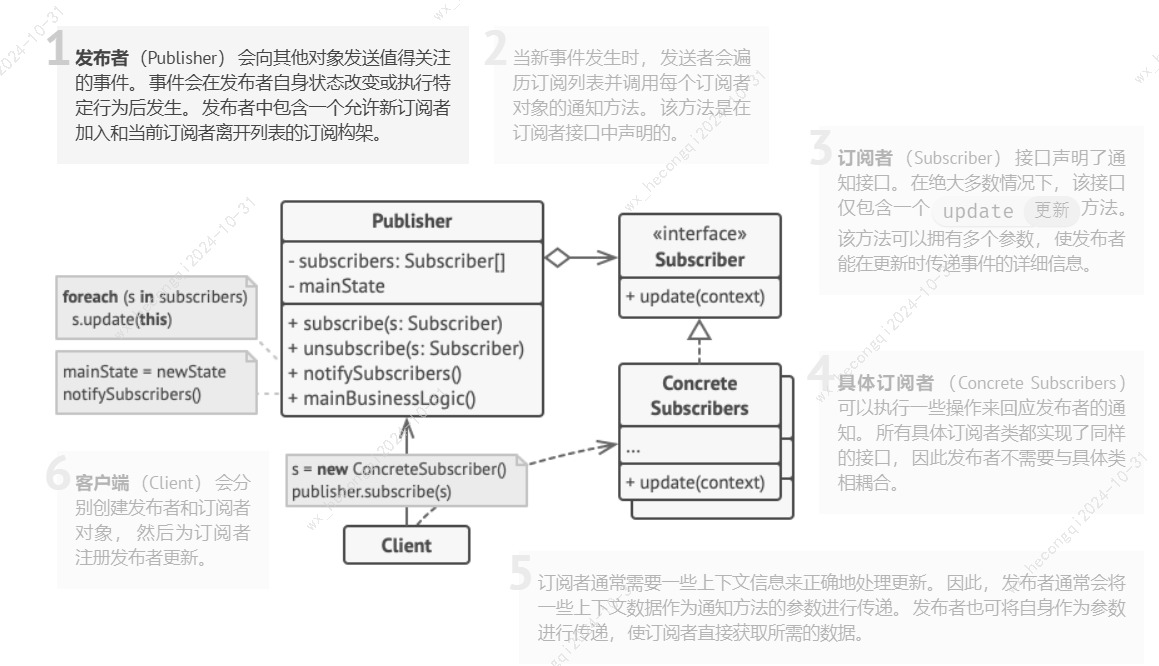


* 备忘录模式

备忘录即备忘的目录，备忘即快照，目录即列表



* 观察者模式



* 状态模式

将状态变化逻辑抽取到状态类，简化代码。

* 策略模式

定义一系列算法， 并将每种算法分别放入独立的类中， 以使算法的对象能够相互替换

* 模版方法模式

它在超类中定义了一个算法的框架， 允许子类在不修改结构的情况下重写算法的特定步骤。

* 访问者模式

将算法与其所作用的对象隔离开。

### Java

#### 线程池参数

核心线程数、最大线程数、最大存活时间、时间单位、线程工厂、阻塞队列、拒绝策略。

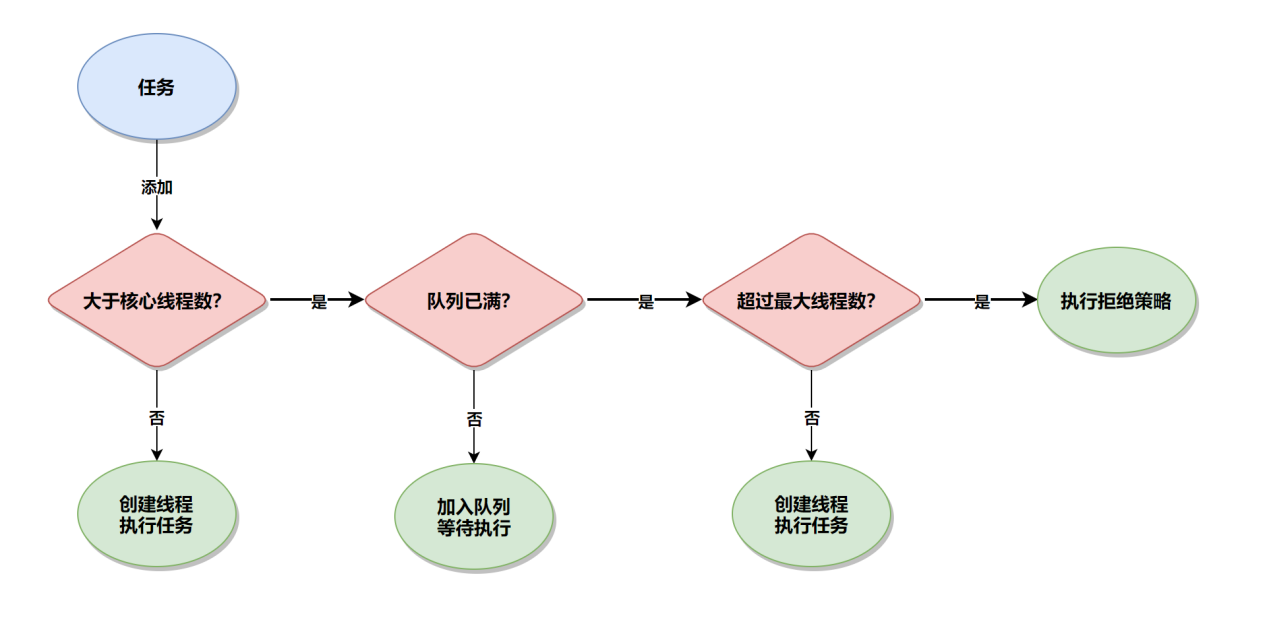
任务处理流程：

1 等待提交任务

1. 2.1 当前线程数小于核心线程数，即使有空闲的线程，也会创建线程处理任务，break(结束第2步)
2. 2.2 当前线程数大于核心线程数，则先放入阻塞队列
3. 2.3 如果阻塞队列未满，且运行线程数小于最大线程数，则等待核心线程空闲再执行，所以无界队列线程数不会超过核心线程数
4. 2.4 如果阻塞队列满了，且运行线程数小于最大线程数，则创建非核心线程运行任务
5. 2.5 如果阻塞队列满了，且运行线程数大于最大线程数，则执行拒绝策略

3 当一个线程完成任务后，它会从队列取下一个任务执行

4 当一个线程无事可做超过最大存活时间时，如果当前线程数超过核心线程数，会销毁线程



#### 并发场景

并行可以更好的利用多核的能力，并发可以让一个CPU核心应对成千上万个IO任务。

http请求线程池、tcp请求线程池、定时任务工作线程池

#### 锁

**1 内置锁（synchronized）**，每个对象都有一个内置的监视器锁，当一个线程进入同步块或者方法时，它会获得对象的锁，其他线程必须等待，直到该线程释放锁后，才能获取锁。

包括对象级锁、类级锁

内置锁的特点：具备重入锁特性，即一个线程已经获得某个锁，它可以再次获得相同的锁

**2** **重入锁（ReentrantLock）**

* 可重入
* 支持尝试获取锁 tryLock，如果锁不可用不阻塞线程
* 支持中断 tryInterruptibly() ,线程在等待锁时可以响应中断
* 支持超时获取锁，线程可以设置最长等待时间

重入锁相比内置锁提高了灵活性，还支持非阻塞、设置等待时间功能

**3 读写锁（ReadWriteLock）**

允许多个线程同时获得读锁，只允许一个线程获得写锁

读写锁相比重入锁，增加了读写的区分能力，适合读多写少的场景

**4 StampedLock**

不仅支持读写锁，还支持乐观读，先不加锁，事后校验。在读写锁的基础上，进一步提高了性能

#### 线程通信

**1 wait 和 notify**，调用对象的wait方法会释放持有的对象锁并阻塞等待，调用对象的notify方法会通知对象的阻塞线程，但被通知的线程不能立即执行，必须等当前线程释放锁后，被通知的线程才能尝试重新获取锁并继续执行。

**2 join方法**，允许一个线程等待另一个线程的完成。

**3 BlockingQueue**，线程安全的队列，队列为空，消费者阻塞，队列满了，直接失败返回。

**4 CountDownLatch**，主线程等待多个子线程完成，多个线程同时开始执行

### JVM

#### 内存结构

永久代/元空间：存储类结构

堆：存储对象实例

栈：线程私有，存放局部变量表、程序计数器

### Spring基础

#### 容器初始化过程

1 创建容器，比如 AnnotationConfigApplicationContext

2 读取注解，扫描类路径

3 Bean 定义的解析，并加载到注册表

4 Bean 的实例化

5 依赖注入，通过构造器注入或者属性注入，将依赖对象注入到 Bean 实例

6 初始化回调

7 完成初始化，开始提供服务

#### Bean生命周期

1 通过反射机制实例化 Bean 对象

2 设置属性/依赖注入

3 Aware 接口回调，可以感知到容器，将容器信息传递给 Bean

4 BeanPostProcessor，如果容器有的话，调用它的 before 方法

5 初始化，如调用 init method 方法

6 BeanPostProcessor，如果容器有的话，调用它的 after 方法

7 Bean 完成初始化，可以使用

8 销毁及回调

9 销毁完成

Q1：BeanFactoryAware、ApplicationContextAware 回调时，我想通过容器对象访问其他Bean，但其他Bean可能还没有创建，怎么办？

A1：没关系，通过BeanFactory.getBean获取Bean的时候，如果Bean还没创建好，Spring会自动创建。

Q2：Spring如何解决Bean的循环依赖？

A2：Spring允许循环依赖，如A依赖B，B依赖A。Spring在初始化A的时候，因为A依赖B，就去初始化B，B又依赖A，Spring会创建半成品的A，B依赖注入半成品A，B创建完成，A依赖注入B，结束。

#### AOP

基本概念：

1. 切面（Aspect）：封装横切关注点，包括通知和切点。在实际应用中，切面是一个类，包含通知逻辑。
2. 连接点（Join Point）：程序执行过程中的一个点，AOP可以在这些点上插入代码。
3. 切点（Pointcut）：切点指那些你想增强的地方，通常定义为一组连接点。
4. 通知（Advice）：通知是AOP中的增强操作，有前置、后置、环绕、返回、异常通知。其中环绕通知可以中断目标方法的执行
5. 目标对象：被切面增强的对象，SpringAOP通过动态代理将增强逻辑添加到目标对象的方法中。
6. 代理：增强后的对象，SpringAOP通过动态代理生成目标对象的代理类，通过代理执行增强后的逻辑

### SpringBoot

#### 优点

1. 简化配置，减少甚至提供默认配置
2. 生态完整，基于Spring框架，集成了SpringMVC，还内置日志、消息、数据源组件
3. 内嵌服务器，实现轻量级部署
4. 与SpringCloud集成，便于构建微服务

#### 应用启动过程

1 SpringBoot应用由main方法启动，调SpringApplication.run引导应用的启动过程

2 加载环境变量，读取配置文件、命令行参数

3 设置应用监听器

4 创建ApplicationContext，扫描注解类，注册成Bean；扫描自动配置类，通过配置条件自动注入Bean

5 启动内嵌的Web容器，将Controller注册到Web容器

6 调用实现了ApplicationRunner接口的Bean

7 应用启动完成

Q：简单介绍下SpringBoot的自动配置/自动装配

A：

-@SpringBootApplication注解复合了@EnableAutoConfiguration注解，隐式启动了自动配置注解。

-模块包的META-INF目录下有spring.factories文件列出了所有自动配置类。

-自动配置类通过条件注解自动注入Bean，如@ConditionOnClass，@ConditionOnMissingBean等。

-条件注解就是满足条件后才会自动注入Bean

# 分布式系统

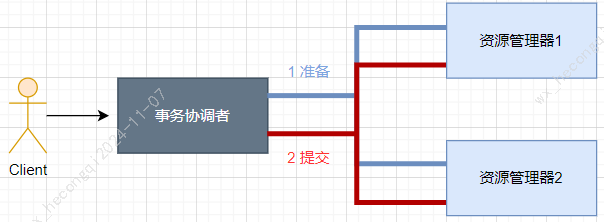
#### 分布式事务

数据库做了分库，或者业务系统做了服务拆分以后，跨库或者跨服务的调用如果需要保证一致性，就需要分布式事务了。

##### 两阶段提交

包含两个角色：分布式事务协调者（coordinator）和N个参与者（participant）

第一阶段：准备，第二阶段：提交



在准备阶段，会开启事务，并写入Redo和Undo日志，但不提交事务。

问题：

性能问题：准备阶段需要全部参与者执行完，才可进入提交阶段，期间一直占用资源。

事务协调者单点故障问题。

##### 三阶段提交

在两阶段的准备阶段之前增加询问阶段，全部参与者反馈OK后，再进入准备阶段

##### TCC

Try-confirm-cancel，类似两阶段提交

##### SEATA

SEATA是阿里开源的分布式事务解决方案，包括AT、TCC、XA模式等

###### AT

Auto Transcation。它是非侵入的，适合支持本地事务的关系数据库，数据源代理类（DataSourceProxy）会完成分布式事务的大部分工作。整体也分两阶段，可实现写隔离、读隔离。

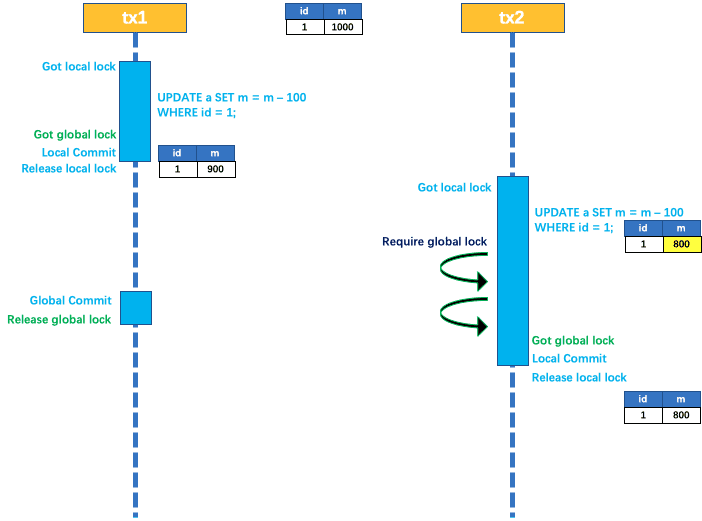
一阶段：提交事务、释放资源锁

二阶段：通过一阶段Undo日志回滚

如下图全局事务 tx1：

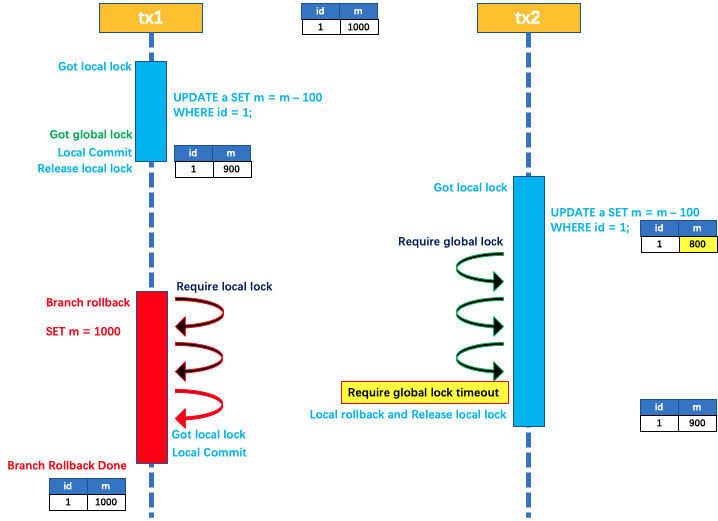
一阶段：先开始本地事务、获取全局锁、提交本地事务、释放本地锁。

二阶段：全局事务提交、释放全局锁



下图全局事务tx1中，本地事务回滚需要获取本地锁。

下图全局事务tx2中，在本地事务开始后、提交前，获取全局事务失败，会触发本地事务回滚。



###### TCC

Try-Confirm-Cancel。它是侵入式的，不依赖底层数据库资源的，需要自定义实现prepare、commit、rollback的模式，比较复杂。

TCC 的 Try 操作作为一阶段，负责资源的检查和预留；Confirm 操作作为二阶段提交操作，执行真正的业务；Cancel 是二阶段回滚操作，执行预留资源的取消，使资源回到初始状态。

###### XA

它是X/Open组织定义分布式事务处理标准。它要求数据资源提供对规范和协议的支持，和AT模式一样，它是非侵入式的，且主流数据库广泛支持。

缺点：

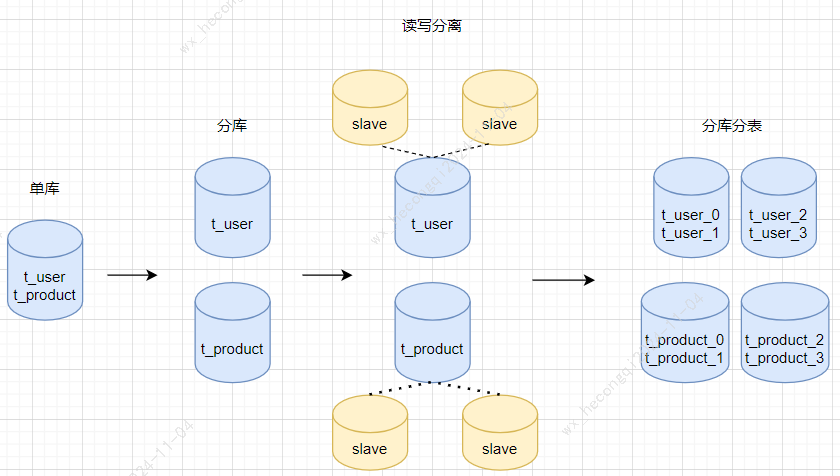
XA prepare 后，分支事务进入阻塞阶段，收到 XA commit 或 XA rollback 前必须阻塞等待。事务资源长时间得不到释放，锁定周期长，而且在应用层上面无法干预，性能差。

# 数据

## 数据库

### 挑战

关系数据库的技术变迁



#### 全局ID问题

分库分表以后，为了避免ID重复，不能使用自增的主键。

雪花ID算法：

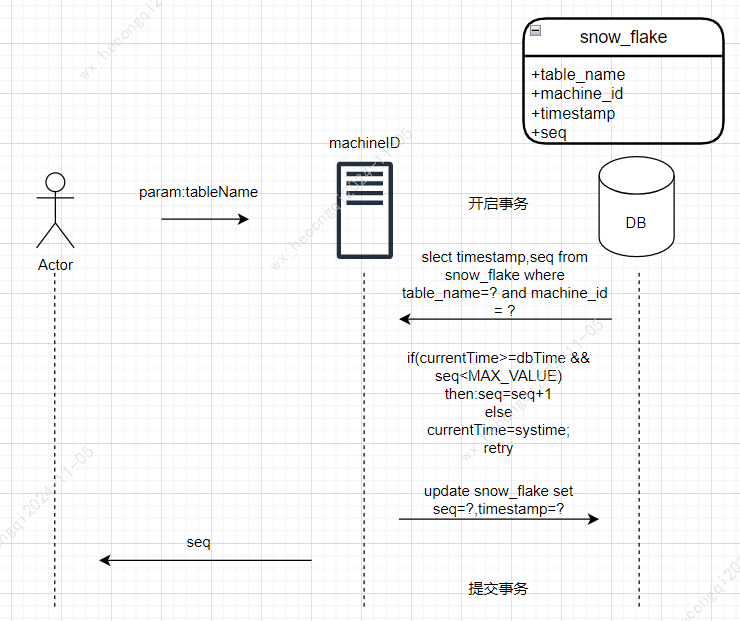
41位的时间戳 + 10位机器ID + 12位序列号。

41位时间戳可以用69年，机器ID支持1024个，同一毫秒最多4096个ID（位数可自定义）

机器ID代表需求方的ID，即需要新增数据的后端微服务机器/容器

将数据表名、最新时间戳、机器ID、序列号都入库保存。通过微服务提供ID生成器。

注意：ID生成器需要处理时间回拨问题



#### 主从延迟

数据库做了读写分离以后，新增/修改主库数据，还没来得及同步到从库，此时查询数据，可能出现主从不一致。

解决办法：

1. 强制走主库
2. Sleep方案：每次查从库之前先sleep
3. 等主库位点
4. 等GTID

详细解释一下等主库位点：

当一个事务执行完成后，我们马上发起一个查询请求，可以通过下面的步骤实现：

1. 当一个事务执行完成后，马上执行show master status 获取主库的File和Position



1. 选择一个从库执行查询
2. 在从库上执行 select master\_pos\_wait(File,Position,1)
3. 如果返回的值>=0，则在这个从库上执行
4. 否则回主库查询

#### 查询所有分区

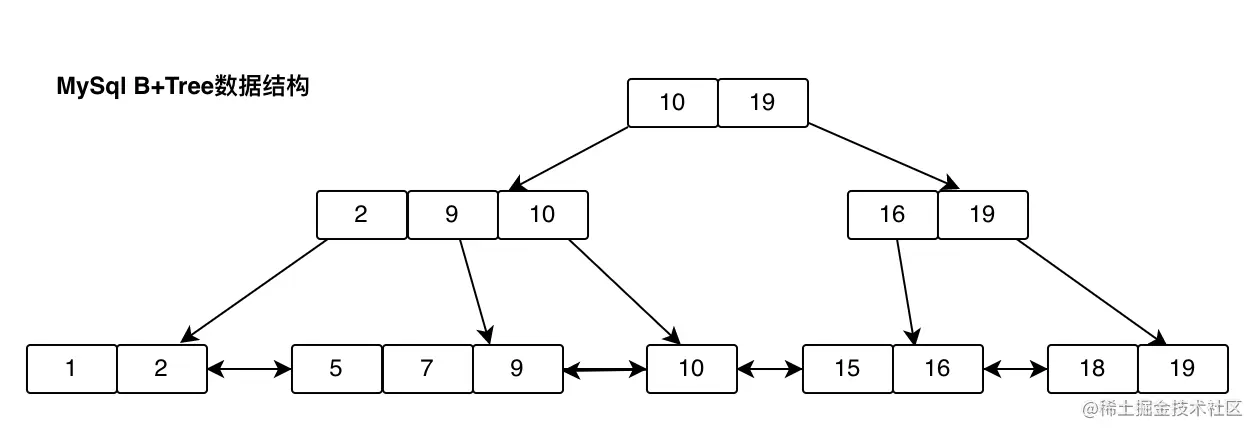
如果查询时没有带入分区键，可能需要查询所有分区。

比如用户表可以按user\_id分区。订单表呢？用户和商户都要经常查询订单表，可以将订单表的数据按用户分区存一份，按商户分区存一份。新增订单时，发送消息给MQ给用户服务和订单服务，分别按用户维度和商户维度创建订单数据，用户维度数据按user\_id分区，上午维度按merchant\_id分区。

这样用户和商户都有分区键可用，而且也分担了查询压力。

### MySQL

#### 底层数据结构



MySQL中，B+树是InnoDB存储引擎使用的数据结构，用于组织和管理索引。

1 叶子节点组成双向列表，用于解决范围查询和顺序访问，叶子节点既存储键值，也存储数据（聚簇索引）

2 非叶子节点只存索引，指向子节点的最大/最小值

3 树是平衡的

#### 索引类型

索引分为聚簇索引和二级索引，它们都使用B+树存储

1. 聚簇索引是主索引，在InonoDB中，每张表的数据都按主键顺序存储，即表的数据是按照B+树结构组织的。
2. 二级索引是除了主键以外的其他索引，用来加速非主键列的查询。二级索引的B+树的叶子节点存储的是列的值和对应的主键值。当使用二级索引查询时，先通过二级索引找到主键值，再通过主键去聚簇索引的B+树查找实际的数据行

#### 索引失效

1 使用 or 查询且不是每个条件都有索引

2 使用负向条件 not in、!=，索引只支持精确匹配或者范围查找

3 like 查询以 % 打头，%部分为未知，无法匹配索引

4 对索引列进行函数操作，因为索引是基于原始值

5 多列索引时，不支持最左前缀原则

#### 性能优化

**查询优化**

1 复杂子查询优化，比如exists语句，主查询的每一行都要做一次子查询，可以改为关联查询

SELECT name FROM users u WHERE EXISTS (SELECT 1 FROM orders o WHERE o.user\_id = u.id);

2 limit 分页，页数越大越慢如何优化？limit查询只返回ID，再关联业务表查询，前端限制只能上一页/下一页，不给用户机会查看比如第1万页

3 避免主查询 select \*，只返回需要的数据，减少传输量，优化查询性能

4 使用索引，减少数据扫描量

**表结构优化**

1 选择合适的数字类型、合适长度的字符串类型

2 大表可用分区分割表数据

3 避免外键约束，因为会增加锁定和写入成本

4 分库分表，减少单库单表的压力

**缓存优化**

频繁读取的变化少的数据，可以考虑使用外部缓存系统

**事务和锁优化**

长事务会阻塞其他查询，尽量缩短事务执行时间

**读写分离**

通过主从复制实现读写分离，减少主库的查询压力

**硬件优化**

1 虚拟机换成物理机

2 增加内存

3 硬盘换成SSD

**定期维护**

1 定期对数据归档，提升查询性能

2 定期分析慢SQL

3 定期监控磁盘、CPU的资源使用

#### 分片

ShardingSphere

#### 分布式事务

##### 最终一致性

1. 主节点完成数据写入后，发送事件到MQ通知从节点，从节点收到通知完成数据写入并通知主节点。
2. 从节点写入失败，可以进行重试
3. 从节点多次重试依然失败，可以回滚，并发送通知给主节点
4. 主节点回滚

#### 锁

* 悲观锁：适合高冲突场景
* 乐观锁：即MVCC，支持更高的并发，事务开始时创建新版本，不锁定数据，事务提交时校验版本，版本没有修改则提交成功，否则提交失败

### Redis

#### 分布式锁

用于解决分布式环境下多个进程或多个服务对共享资源的并发操作，避免冲突

**Redis分布式锁的实现原理**

1 获取锁

SET key value NX PX expiration\_time

key是锁的唯一标识。

value是锁的值，用来确保不会被错误释放。

NX确保key仅在不存在时才会被设置

PX expiration\_time设置锁的过期时间，防止进程异常退出，锁永远不被释放

2 释放锁

if redis.get(key) == value:

redis.del(key)

通过校验value的方式，可以确保锁不会被其他客户端错误释放

**存在的问题**

锁的失效问题，任务的执行时间超过了锁的过期时间，可以通过锁续期来延长锁的有效时间

**分布式锁的使用场景**

分布式任务调度

**解决方案**

Redisson

#### 集群

键->哈希槽->节点

哈希槽只分配给主节点，主节点出现故障以后，从节点会代替失效的主节点，集群内更新哈希槽与节点的映射信息

#### 一致性问题

容忍短暂的不一致，提供最终一致性，一致性要求特别高可以用分布式事务

1 查询和更新流程：应用程序先从缓存读数据，如果缓存未命中，则从数据库加载，并写入缓存。更新时，先更新数据库，再清除缓存。

2 设置过期时间：缓存数据设置过期时间

3 分布式场景：使用分布式锁保证一次只有一个线程更新缓存和数据库

#### 缓存穿透击穿雪崩

1. 穿透是请求的数据在缓存和数据库都不存在
2. 击穿是热点数据缓存过期，导致大量请求访问数据库，可使用分布式锁解决，可以采用异步方式提高性能
3. 雪崩指在某时间点，大量缓存同时失效，导致大量请求访问数据库，可设置不同过期时间解决

### ES

#### 倒排索引

正排索引是文档和词项的映射关系，倒排索引是词项到文档的映射关系。

在正排索引中，查找一个词项时，需要遍历所有的文档。

倒排索引可以快速找到某个词项出现在哪些文档中

### 分布式数据库

* OceanBase
* Doris
* TIDB

### MongoDB

以JSON格式存储数据，支持嵌套结构，代替关系数据库的关联操作

## 大数据

Hadoop

Spark