数据库类

1、mysql隔离级别：

SQL 标准定义了四个隔离级别：

- **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读**。
- **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生**。
- **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生**。
- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读**。

隔离级别	脏读	不可重复读	幻读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

2、数据库索引原则：

应该建立索引的字段：

1.主键和外键上必须有索引

2.经常与其它表连接的连接字段

3.经常进行范围查询的列

4.经常排序查询的列

5.where子句中的字段

6.group by、order by、distnict字段后面的字段

不应该建立索引的字段：

1.查询很少的列

2.区分度不高（大量重复值）的列

3.text, image和bit数据类型的列。这些列的数据量要么相当大，要么取值很少

4.经常被修改的列

3、索引失效的情况：

1.如果条件中有or，即使其中有条件带索引也不会使用

2.对于多列索引（复合索引），条件中没有索引的第一个字段，则不会使用索引

3.like查询是以%开头

4、分组排序

每个类别最贵的两种商品

```
mysql> select a.*
    -> from mygoods a
    -> where (select count(*)
    -> from mygoods
    -> where cat_id = a.cat_id and price > a.price  ) <2
    -> order by a.cat_id,a.price desc;
```

Springboot类

1、springboot启动原理

2、单例模式与springboot

Spring 中 bean 的默认作用域就是 singleton(单例)的。 除了 singleton 作用域，Spring 中 bean 还有下面几种作用域：

- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

3、bean实例化

写了也看不懂

4、线程池主要的参数及整个流程

`ThreadPoolExecutor` 3 个最重要的参数：

- **`corePoolSize`** ：核心线程数线程数定义了最小可以同时运行的线程数量。
- **`maximumPoolSize`** ：当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- **`workQueue`**：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

`ThreadPoolExecutor`其他常见参数：

1. **`keepAliveTime`**:当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime`才会被回收销毁；
2. **`unit`**：`keepAliveTime` 参数的时间单位。
3. **`threadFactory`** :executor 创建新线程的时候会用到。
4. **`handler`** :饱和策略。关于饱和策略下面单独介绍一下。

5、beanFactory和ApplicationContext

两者对比：

- `BeanFactory` ： 延迟注入(使用到某个 bean 的时候才会注入),相比于`ApplicationContext` 来说会占用更少的内存，程序启动速度更快。
- `ApplicationContext` ： 容器启动的时候，不管你用没用到，一次性创建所有 bean 。`BeanFactory` 仅提供了最基本的依赖注入支持，`ApplicationContext` 扩展了 `BeanFactory` ,除了有`BeanFactory`的功能还有额外更多功能，所以一般开发人员使用`ApplicationContext`会更多。

6、线程数量设定依据

1、CPU密集型
CPU密集即需要大量运算，且没有阻塞，CPU一直全速运行

参考公式：CPU核数 + 1

比如8核CPU 我们可以构建9个线程的线程池

2、IO密集型
算法一： CPU核数*2
算法二：
IO密集即该任务需要大量的IO，大量的阻塞
IO密集型，大部分线程都会被阻塞，故需要多配置线程数
参考公式： CPU核心数/（1-阻塞系数）
阻塞系数： 0.8-0.9

比如 8核CPU： 8/（1-0.9） = 80 个线程数

7、threadlocal

8、数据结构

double与bigdecimal?

BigDecimal是Java中用来表示任意精确浮点数运算的类，在BigDecimal中，使用unscaledValue × 10-scale来表示一个浮点数。其中，unscaledValue是一个BigInteger，scale是一个int。从这个表示方法来看，BigDecimal只能标识有限小数，不过可以表示的数据范围远远大于double，在实际应用中基本足够了。

arrays.toList为什么不能move?

Arrays.asList() 方法返回的并不是 java.util.ArrayList ， 而是 java.util.Arrays 的一个内部类,这个内部类并没有实现集合的修改方法或者说并没有重写这些方法。

方法返回的是Arrays的一个内部类ArrayList

==和equals的区别

浮点数之间的等值判断，基本数据类型不能用==来比较，包装数据类型不能用 equals 来判断。要用compareTo

`==` 对于基本类型和引用类型的作用效果是不同的：

- 对于基本数据类型来说， `==` 比较的是值。
- 对于引用数据类型来说， `==` 比较的是对象的内存地址。

> 因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。

`equals()` 作用不能用于判断基本数据类型的变量，只能用来判断两个对象是否相等。`equals()`方法存在于`Object`类中，而`Object`类是所有类的直接或间接父类。

bigInt与Long和BigInteger

如果不是无符号类型，BIGINT(20)的取值范围为-9223372036854775808~9223372036854775807。与Java.lang.Long的取值范围完全一致，mybatis会将其映射为Long
而BIGINT(20) UNSIGNED的取值范围是0 ~ 18446744073709551615，其中一半的数据超出了Long的取值范围，Mybatis将其映射为BigInteger

KAFKA(中间件)

1、KAFKA如何保证事务一致性；

2、CDC?

6、设置线程数量的根据

redis类

1、常见数据结构

Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。Memcached 只支持最简单的 k/v 数据类型。

数据结构类

1、LinkedList与ArrayList的区别，空间复杂度和时间复杂度

链表（LinkedList）

虽然是一种线性表，但是并不会按线性的顺序存储数据，使用的不是连续的内存空间来存储数据。

链表的插入和删除操作的复杂度为 O(1)，只需要知道目标位置元素的上一个元素即可。但是，在查找一个节点或者访问特定位置的节点的时候复杂度为 O(n)。

数组（Array）

是一种很常见的数据结构。它由相同类型的元素（element）组成，并且是使用一块连续的内存来存储。

我们直接可以利用元素的索引（index）可以计算出该元素对应的存储地址。

数组的特点是：**提供随机访问** 并且容量有限。

```
假如数组的长度为 n。
访问：O（1）//访问特定位置的元素
插入：O（n ）//最坏的情况发生在插入发生在数组的首部并需要移动所有元素时
删除：O（n）//最坏的情况发生在删除数组的开头发生并需要移动第一元素后面所有的元素时
```

数组 vs 链表

- 数组支持随机访问，而链表不支持。
- 数组使用的是连续内存空间对 CPU 的缓存机制友好，链表则相反。
- 数组的大小固定，而链表则天然支持动态扩容。如果声明的数组过小，需要另外申请一个更大的内存空间存放数组元素，然后将原数组拷贝进去，这个操作是比较耗时的！

##

设计模式

1、常用的设计模式

工厂设计模式

Spring使用工厂模式可以通过 `BeanFactory` 或 `ApplicationContext` 创建 bean 对象。

两者对比：

- `BeanFactory` ：延迟注入(使用到某个 bean 的时候才会注入),相比于`ApplicationContext` 来说会占用更少的内存，程序启动速度更快。
- `ApplicationContext` ：容器启动的时候，不管你用没用到，一次性创建所有 bean 。`BeanFactory` 仅提供了最基本的依赖注入支持，`ApplicationContext` 扩展了 `BeanFactory` ,除了有`BeanFactory`的功能还有额外更多功能，所以一般开发人员使用`ApplicationContext`会更多。

ApplicationContext的三个实现类：

1. `ClassPathXmlApplication`：把上下文文件当成类路径资源。
2. `FileSystemXmlApplication`：从文件系统中的 XML 文件载入上下文定义信息。
3. `XmlWebApplicationContext`：从Web系统中的XML文件载入上下文定义信息。

Example:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class App {
  public static void main(String[] args) {
    ApplicationContext context = new FileSystemXmlApplicationContext(
        "C:/work/IOC Containers/springframework.applicationcontext/src/main/resources/bean-factory-config.xml");

    HelloApplicationContext obj = (HelloApplicationContext) context.getBean("helloApplicationContext");
    obj.getMsg();
  }
}
```

单例设计模式

在我们的系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、对话框、注册表、日志对象、充当打印机、显卡等设备驱动程序的对象。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能会导致一些问题的产生，比如：程序的行为异常、资源使用过量、或者不一致性的结果。

使用单例模式的好处:

- 对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统开销；
- 由于 new 操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻 GC 压力，缩短 GC 停顿时间。

Spring 中 bean 的默认作用域就是 singleton(单例)的。 除了 singleton 作用域，Spring 中 bean 还有下面几种作用域：

- prototype : 每次请求都会创建一个新的 bean 实例。
- request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。
- session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Spring 实现单例的方式：

- xml : `<bean id="userService" class="top.snailclimb.UserService" scope="singleton"/>`
- 注解：`@Scope(value = "singleton")`

Spring 通过 `ConcurrentHashMap` 实现单例注册表的特殊方式实现单例模式。Spring 实现单例的核心代码如下

```
// 通过 ConcurrentHashMap（线程安全） 实现单例注册表
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(64);

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
        Assert.notNull(beanName, "'beanName' must not be null");
        synchronized (this.singletonObjects) {
            // 检查缓存中是否存在实例
            Object singletonObject = this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                //...省略了很多代码
                try {
                    singletonObject = singletonFactory.getObject();
                }
                //...省略了很多代码
                // 如果实例对象在不存在，我们注册到单例注册表中。
                addSingleton(beanName, singletonObject);
            }
            return (singletonObject != NULL_OBJECT ? singletonObject : null);
        }
    }
    //将对象添加到单例注册表
    protected void addSingleton(String beanName, Object singletonObject) {
            synchronized (this.singletonObjects) {
                this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL_OBJECT));

            }
        }
}
```

KAFKA

1、如何实现的exactly once

Kafka 如何保证消息不重复消费

kafka出现消息重复消费的原因：

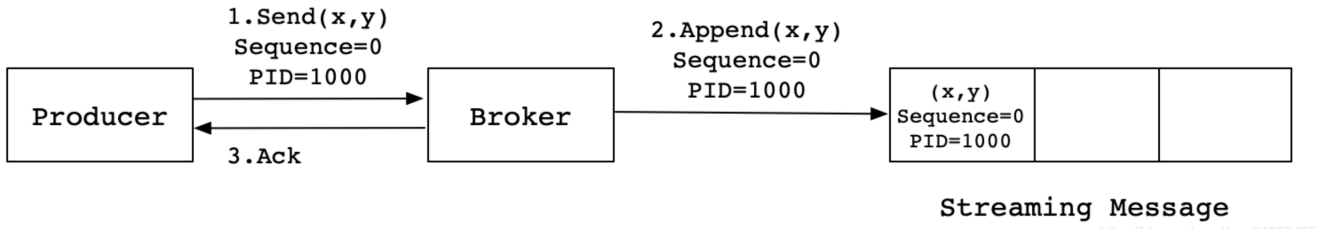
- 服务端侧已经消费的数据没有成功提交 offset（根本原因）。
- Kafka 侧 由于服务端处理业务时间长或者网络链接等等原因让 Kafka 认为服务假死，触发了分区 rebalance。

解决方案：

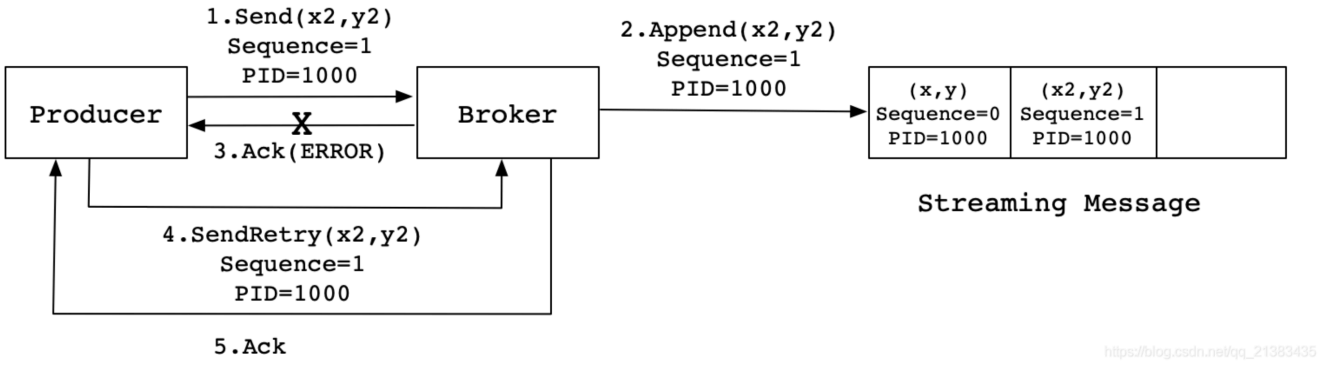
- 消费消息服务做幂等校验，比如 Redis 的set、MySQL 的主键等天然的幂等功能。这种方法最有效。

2.3.2 幂等性引入之后解决了什么问题?

面对这样的问题，Kafka引入了幂等性。那么幂等性是如何解决这类重复发送消息的问题的呢？下面我们可以先来看看流程图：



同样，这是一种理想状态下的发送流程。实际情况下，会有很多不确定的因素，比如Broker在发送Ack信号给Producer时出现网络异常，导致发送失败。异常情况如下图所示：



当Producer发送消息(x2,y2)给Broker时，Broker接收到消息并将其追加到消息流中。此时，Broker返回Ack信号给Producer时，发生异常导致Producer接收Ack信号失败。对于Producer来说，会触发重试机制，将消息(x2,y2)再次发送，但是，由于引入了幂等性，在每条消息中附带了PID（ProducerID）和SequenceNumber。相同的PID和SequenceNumber发送给Broker，而之前Broker缓存过之前发送的相同的消息，那么在消息流中的消息就只有一条(x2,y2)，不会出现重复发送的情况。

- 将 `enable.auto.commit` 参数设置为 false，关闭自动提交，开发者在代码中手动提交 offset。那么这里会有个问题：**什么时候提交offset合适?**
    - 处理完消息再提交：依旧有消息重复消费的风险，和自动提交一样
    - 拉取到消息即提交：会有消息丢失的风险。允许消息延时的场景，一般会采用这种方式。然后，通过定时任务在业务不繁忙（比如凌晨）的时候做数据兜底。

2、consumer group

PYTHON

1、dataframe

Left Join

```
personDataFrame.join(orderDataFrame, personDataFrame("id_person") ===
orderDataFrame("id_person"), "left")
```

分组算平均值

```
df['B'].groupby(by=df['A']).mean()
```

分组排序

rank方法

```
data['new_rank'] = data.groupby('house_code')['q_score_new'].rank(ascending=False,
method='dense')
```

sort_values方法

```
data.sort_values(['q_score_new'],
ascending=False).groupby(['house_code']).cumcount() + 1
```