一、消息队列

面试题目

1. 为什么使用消息队列进行应用解耦、异步处理、流量削峰及信息通信?
2. 如何保证消息消费的幂等性?
3. 如何保证生产者发送消息的可靠性?
4. 如何保证消息的顺序性?
5. 怎么解决消息积压和消息过期问题?
6. 你所熟知的消息队列产品有哪些，简述优缺点?
7. 消息队列的一般存储方式有哪些?
8. 如何自己设计消息队列

01-消息队列是什么?

消息队列，分布式系统中重要的组件，主要解决应用解耦，异步消息，流量削峰等问题。可实现高性能，高可用，可伸缩的最终一致性架构，是大型分布式系统不可或缺的基本组件【中间件】。

> 注意：消息队列容易与Java内部的MessageQueue搞混，我们一般所谓的消息队列，多指消息中间件，分布式消息队列。

目前主流的信息队列：

- Kafka
- RabbitMQ
- RocketMQ，老版本叫MetaQ
- ActiveMQ，日落西山
- Ones【阿里】
- 自研的MQ
- ...

消息队列的组成：

- Producer 消息的生产者
- Consumer 消息的消费者

- Message Broker：主要职责存储消息，转发消息。转发消息又分：**Pull类型**，**Push类型**
    - Pull：是指 Consumer 主动从 Message Broker 获取消息
    - Push：是指 Message Broker 主动将 Consumer 感兴趣的消息推送给 Consumer

消息队列的五大场景：

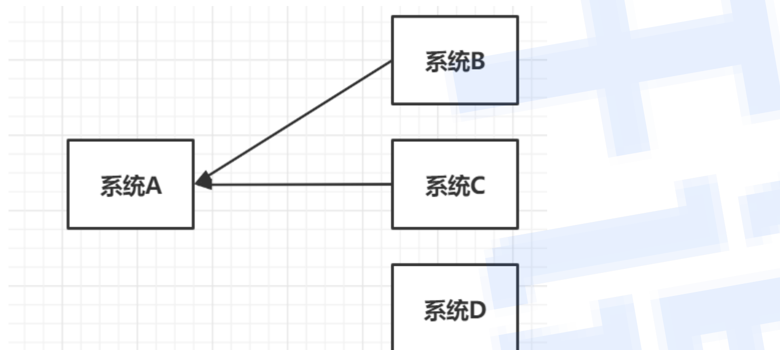
- 应用解耦
- 异步处理
- 削峰填谷
- 日志处理
- ~~消息通讯【很少，IM】~~

消息队列优缺点：

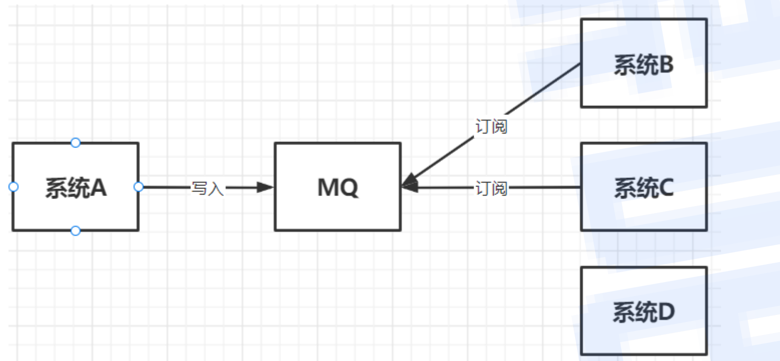
- 优点：应用解耦、异步提速、削峰填谷
- 缺点：
    - 系统可用性降低：一荣俱荣，一损俱损，外部依赖越多，服务越是容易挂掉。
    - 系统复杂度提升：1.消息怎么保证不重复消费。2.消息怎么保证不丢失。3.需要怎么保证顺序性
    - 会出现一致性【没有共识】问题：弱一致、强可用。引入MQ需要保证数据的最终一致性，且系统能够容忍短暂的不一致性

02-应用解耦、异步处理、流量削峰、信息通信及日志处理

应用解耦



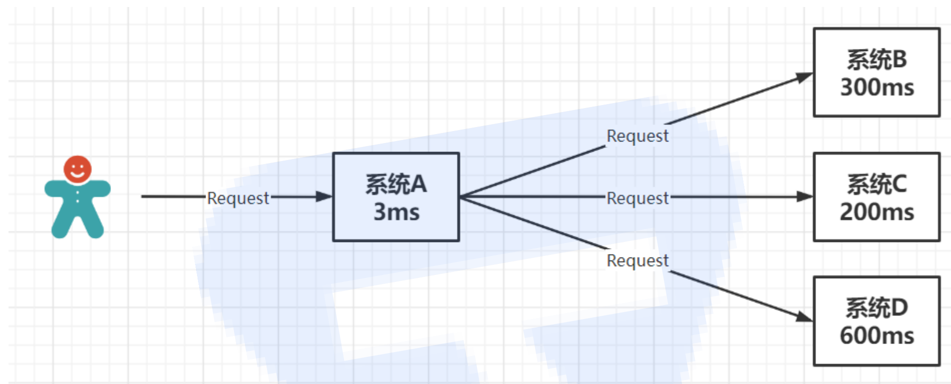
缺点是耦合性太强：每个系统都要接入A，每次加入新系统是不是都需要修改代码？



应用解耦：需要被接入的系统A只需要将消息写入MQ即可，每个需要对接系统A的系统只需要订阅消息队列的消息，系统A的代码完全不需要修改。

举例：用户支付订单完成后，系统需要给用户发红包、增加积分、发货、物流等等行为，就可以通过这样的方式进行解耦。

异步处理

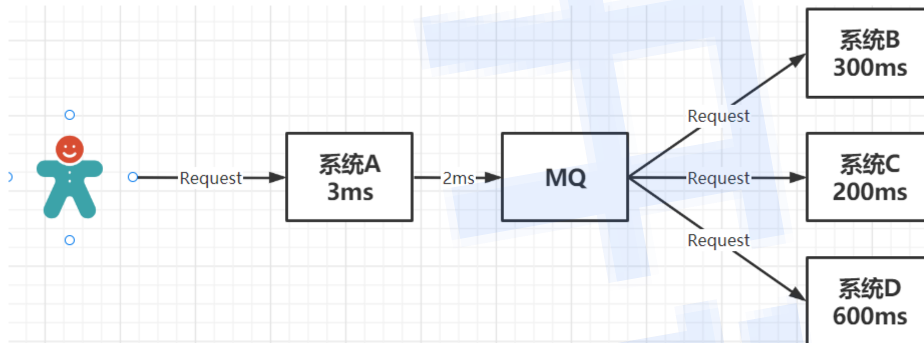


一次请求耗时：3 + 300 + 200 + 600 = 1103ms

> 请求执行耗时较长有什么弊病：系统百分之80%的性能问题，都是慢查询导致
>
> - 系统请求耗时较长 --- 吞吐量大幅降低 --- 系统资源占用陡升

特点：串行、逐个、同步

1. 系统A需要等待，BCD逐个执行完毕
2. 如果任意系统出错，那么真个流程就报错
3. 如果任意系统超时，那么整个流程就超时

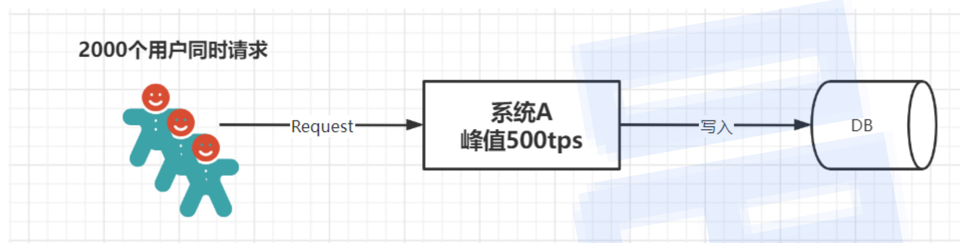


一次请求耗时：3 + 2 = 5ms

特点：异步、并行

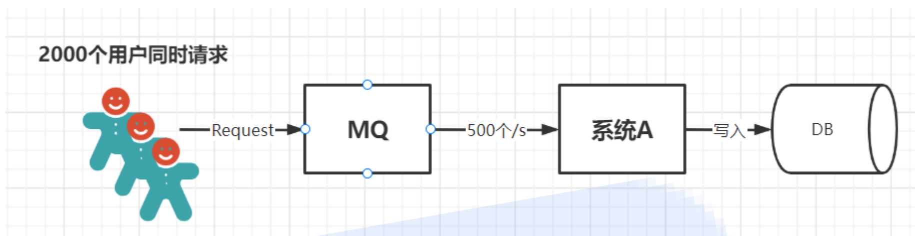
- 前提：系统A返回的结果，并不依赖BCD处理的结果
- MQ发送消息失败咋办？

流量削峰



系统A压测得出接口峰值处理能力500tps。

- 大多数系统，一定会有访问量的波峰和波谷。比如：12306、美团外卖...
- 如果在访问量大的情况下，所有请求都打到数据库上，那么再强悍的数据库也很难承受。



将多余请求积压在MQ中，系统A慢慢处理。把超过500峰值的流量削掉，填在空闲的其他低估时期，所以才叫削峰填谷。

消息队列会不会被打挂掉?

信息通信

这个很好理解：消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。

举例:

- IM 聊天
- 点对点消息队列。有基于消息队列的 RPC 框架实现，例如rabbitmq-jsonrpc ，用的人较少。
- 面向物联网的 MQTT 。阿里在开源的 RocketMQ 基础上，增加了 MQTT 协议的支持，可见 消息队列 for IoT 。
- ....

日志处理

日志处理，是指将消息队列用在日志处理中，比如 Kafka 的应用，解决大量**日志传输**的问题。

- 日志采集client，负责日志数据采集，定时批量写入 Kafka 队列
- Kafka，负责日志数据的接收，存储和转发
- 日志处理Server：订阅并消费 Kafka 队列中的日志数据，将日志信息进行分析，可视化展示等...

最常见日志解决方案：ELKK

- Elasticsearch ： 实时日志分析服务的核心技术，一个 schemaless ，实时的数据存储服务，通过 index 组织数据，兼具强大的搜索和统计功能。
- Logstash ： 对接 Kafka 写入的日志，做日志解析，统一成 JSON 输出给 Elasticsearch 中。
- Kibana ： 基于 Elasticsearch 的数据可视化组件，超强的数据可视化能力是众多公司选择 ELK stack 的重要原因。
- Kafka ： 接收用户日志的消息队列。

03-消息的幂等性

什么是幂等性?

> 幂等【idempotence】是一个数学与计算机学的概念，常见于抽象代数中。**在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。**

> ps.幂等函数，或幂等方法是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。
>
> 大白话：幂等性就是一个数据或者一个请求，给你重复来了多次，你得确保对应的数据是不会改变的，不能出错。

消息队列的三种消费方式：性能层层递减，可靠性逐步提升【1 -> 2 -> 3】

1. At Most once：消息至多被消费一次，消息可能会丢失，但绝不重传。
   - 特点：吞吐量达，实现简单。消息可能会丢失
2. **At lest once：消息至少被消费一次，消息可以重传，但绝不丢失。**
   - 特点：消息不会丢失，消息会出现多次投递【重复消息】
3. Excatly once：每一条消息只被传递一次
   - 特点：消息不会丢失，消息不会重复。实现起来比较复杂，相比较下来性能不佳

为什么会出现重复消息呢?

- 正常情况下，出现重复消息的概率其实很小【程序员的第六感】
- 对于 Producer 来说：可能因为网络问题，Producer 重试多次发送消息，实际第一次就发送成功，那么就会产生多条相同的消息。
- 对于 Consumer 来说：可能因为 Broker 的消息进度丢失，导致消息重复投递给 Consumer 。Consumer 消费成功，但是因为 JVM 异常崩溃，导致消息的消费进度未及时同步给 Consumer 。对于大多数消息队列，考虑到性能，消费进度是异步定时同步给 Broker

> 举例： kafka有一个叫做offset的概念，就是每个消息写进去，都有一个offset代表他的序号，然后consumer消费了数据之后，每隔一段时间，会把自己消费过的消息的offset提交一下，代表我已经消费过了，下次就算重启，kafka就会让消费者从上次消费到的offset来继续消费。但是万事总有例外，如果consumer消费了数据，还没来得及发送自己已经消费的消息的offset就挂了，那么重启之后就会收到重复的数据。

如何保证消息的幂等性?

消费者实现幂等性，有两种方式：

- 方式一：框架层统一封装
- 方式二：业务层自己实现

方式一：框架层统一封装：

- 消息排重唯一标识，由Producer基于消息生成唯一标识
- 消息排重标识的存储器：关系型数据库(排重表)，KV数据库
- 排重逻辑：
  - 业务逻辑执行成功：插入排重记录，并且需要让插入记录和业务逻辑在同一事务中。
  - 业务逻辑执行失败：不能插入排重记录

方式二：业务层自己实现【建议】：

方式很多，这个和 HTTP 请求实现幂等是一样的逻辑：

- 乐观锁
  - 先查询数据库，判断数据是否已经被更新过。如果是，则直接返回消费完成，否则执行消费。
  - 更新数据库时，带上数据的状态。如果更新失败，则直接返回消费完成，否则执行消费。
- 分布式锁，Zookeeper 或者 Redis 实现分布式锁

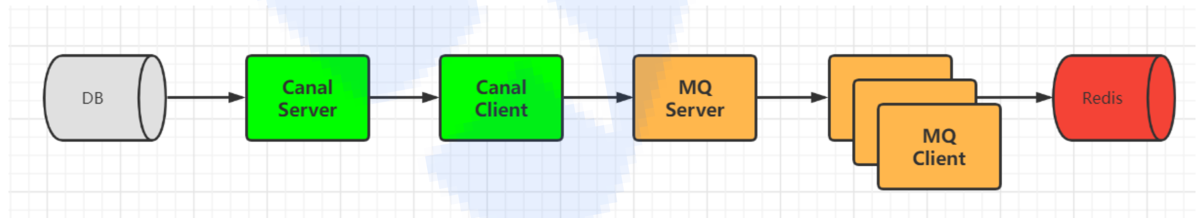
04-消息的顺序性

消息队列中的若干消息如果是对同一个数据进行操作，这些操作具有前后的关系，必须要按前后的顺序执行，否则就会造成数据异常。

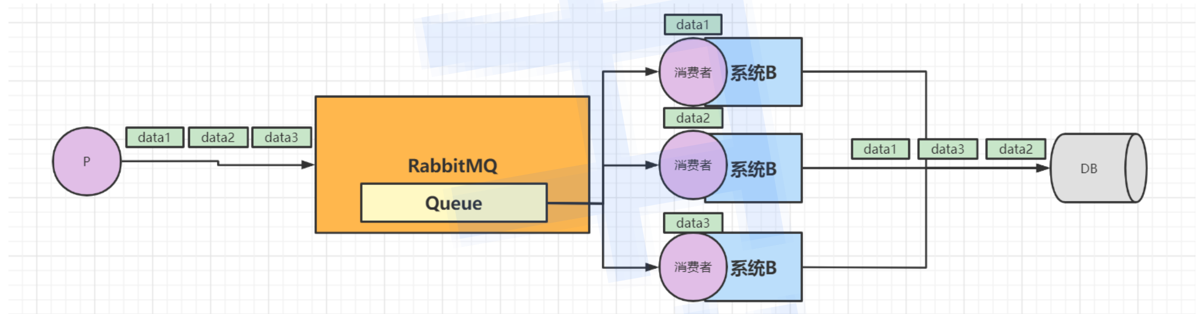
MQ顺序错乱的场景：

- 比如通过mysql binlog进行数据同步，由于对数据库的数据操作是具有顺序性的，如果操作顺序搞反，就会造成不可估量的错误。
- 比如数据库对一条数据依次进行了 插入->更新->删除操作，这个顺序必须是这样，如果在同步过程中，消息的顺序变成了 删除->插入->更新，那么原本应该被删除的数据，就没有被删除，造成数据的不一致问题。

数据同步：



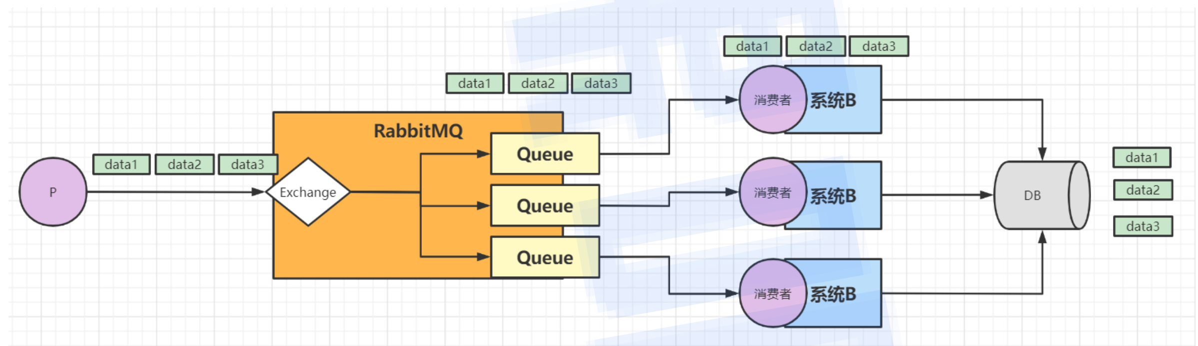
详细剖析：一个 queue，多个 consumer。比如，生产者向 RabbitMQ 里发送了三条数据，顺序依次是 data1/data2/data3，压入的是 RabbitMQ 的一个内存队列。有三个消费者分别从 MQ 中消费这三条数据中的一条，结果消费者 2 先执行完操作，把 data2 存入数据库，然后是 data1/data3。这不明显乱了。



解决方案：

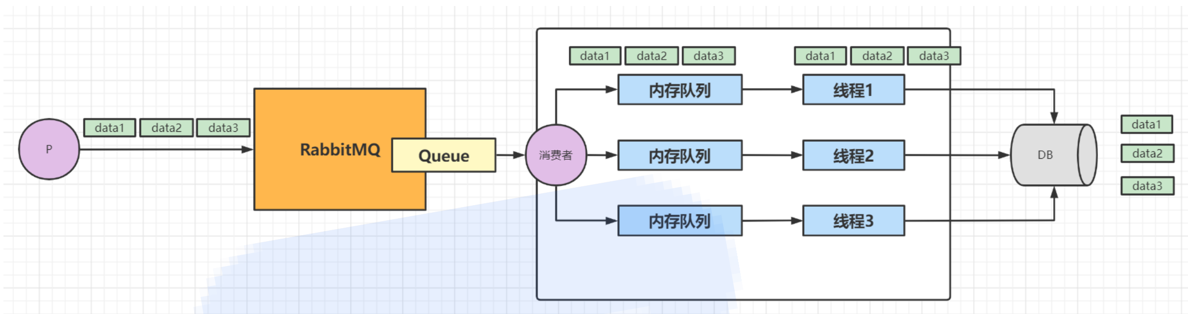
方法一：拆分多个queue，每个queue一个consumer，这样会造成吞吐量下降。【**路由键**】

- 回到原始状态，一个生产者，一个消费者，一个MQ

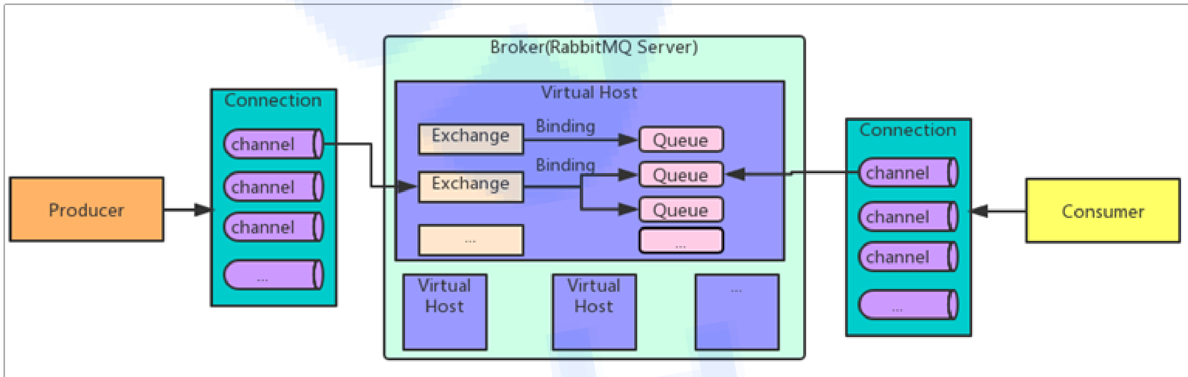


方法二：一个queue对应一个consumer，然后这个consumer内部用内存队列排队，然后分发给底层不同的worker处理

- 生产者发送三个消息到一个queue中，三个消息具有先后顺序
- 消费者不在去直接消费消息，而是将消息保存到内存队列中【根据关键值进行hash操作，将关键值相同的数据发送到相同的队列里面】
- 消费线程直接去内存队列获取消息消费，这样就可以保证消息的顺序



05-消息的可靠性



如果保证消息的可靠性？需要解决如下问题

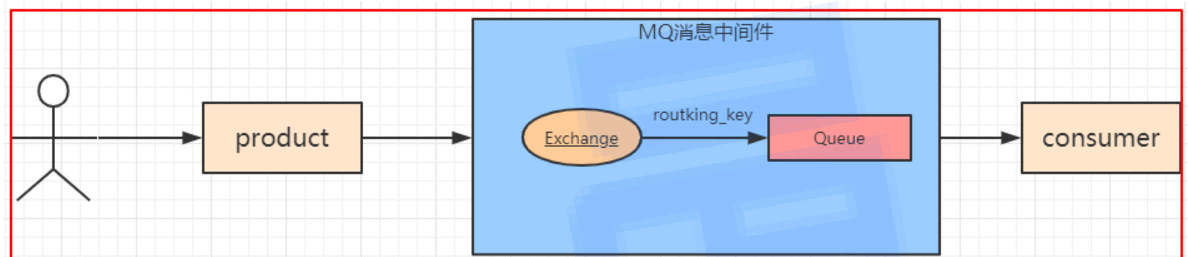
- **问题1：生产者能百分之百将消息发送给消息队列！**
  - 两种意外情况：
    - 第一，消费者发送消息给MQ失败，消息丢失；
    - 第二，交换机路由到队列失败，路由键写错；
- **问题2：消费者能百分百接收到请求，且业务执行过程中还不能出错！**

1. 生产者确认

在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式。

- confirm 确认模式
- return 退回模式

rabbitmq 整个消息投递的路径为：



- 消息从生产者(producer)发送消息到交换机(exchange)，不论是否成功，都会执行一个确认回调方法confirmCallback 。
- 消息从交换机(exchange)到消息队列(queue)投递失败则会执行一个返回回调方法 returnCallback 。

我们将利用这两个 callback 控制消息的可靠性投递

1.1 confirm 确认模式

目标：演示消息确认模式效果

生产者发布消息确认模式特点，不论消息是否进入交换机均执行回调方法

实现步骤：

1. 在配置文件中，开启生产者发布消息确认模式
2. 编写生产者确认回调方法
3. 在RabbitTemplate中，设置消息发布确认回调方法
4. 请求测试：
   - 测试成功回调：
   - 测试失败回调：

实现过程：

1. 在配置文件中，开启生产者发布消息确认模式

```
# 开启生产者确认模式：(confirm),投递到交换机，不论失败或者成功都回调
spring.rabbitmq.publisher-confirms=true
```

2. 编写生产者确认回调方法

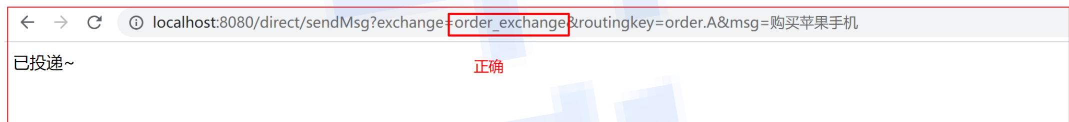
```
//发送消息回调确认类，实现回调接口ConfirmCallback,重写其中confirm()方法
@Component
public class MessageConfirmCallback implements RabbitTemplate.ConfirmCallback {
    /**
     * 投递到交换机，不论投递成功还是失败都回调次方法
     * @param correlationData 投递相关数据
     * @param ack 是否投递到交换机
     * @param cause 投递失败原因
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
        if (ack){
            System.out.println("消息进入交换机成功{}");
        } else {
            System.out.println("消息进入交换机失败{} ， 失败原因：" + cause);
        }
    }
}
```

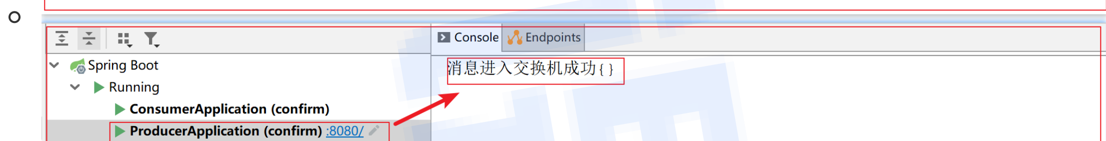
3. 在RabbitTemplate中，设置消息发布确认回调方法

```
@Component
public class MessageConfirmCallback implements RabbitTemplate.ConfirmCallback{

    @Autowired
    private RabbitTemplate rabbitTemplate;
    /**
```

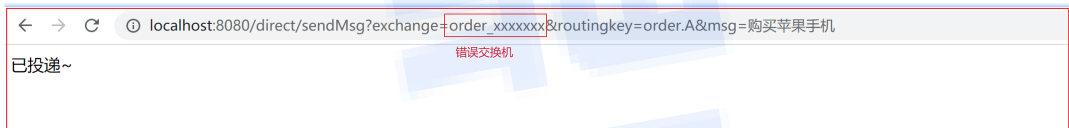
```java
     * 创建RabbitTemplate对象之后执行当前方法，为模板对象设置回调确认方法
     * 设置消息确认回调方法
     * 设置消息回退回调方法
     */
    @PostConstruct
    public void initRabbitTemplate(){
        //设置消息确认回调方法
        rabbitTemplate.setConfirmCallback(this::confirm);
    }
    /**
     * 投递到交换机，不论投递成功还是失败都回调次方法
     * @param correlationData 投递相关数据
     * @param ack 是否投递到交换机
     * @param cause 投递失败原因
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
        if (ack){
            System.out.println("消息进入交换机成功{}");
        } else {
            System.out.println("消息进入交换机失败{} ， 失败原因：" + cause);
        }
    }
}
```

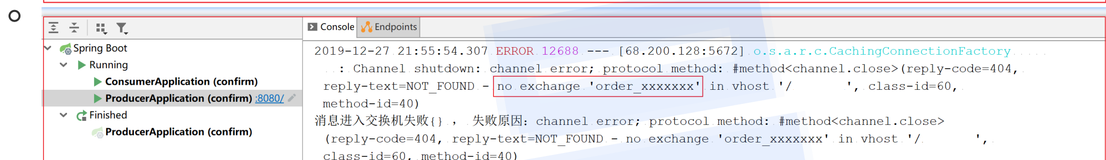
4. 请求测试：
    - 测试成功回调：`http://localhost:8080/direct/sendMsg?exchange=order_exchange&routingkey=order.A&msg=购买苹果手机`



    - 

    - 测试失败回调：`http://localhost:8080/direct/sendMsg?exchange=order_xxxxxxx&routingkey=order.A&msg=购买苹果手机`



    - 

1.2 return 退回模式

目标：演示消息回退模式效果

消息回退模式特点：**消息进入交换机**，路由到队列过程中出现异常则执行回调方法

实现步骤：

1. 在配置文件中，开启生产者发布消息回退模式
2. 在MessageConfirmCallback类中，实现接口RabbitTemplate.ReturnCallback

3. 并重写RabbitTemplate.ReturnCallback接口中returnedMessage()方法
4. 在RabbitTemplate中，设置消息发布回退回调方法
5. 请求测试：
    - 测试成功回调：
    - 测试失败回调：

实现过程：

1. 在配置文件中，开启生产者发布消息回退模式

```
# 开启生产者回退模式:(returns)，交换机将消息路由到队列，出现异常则回调
spring.rabbitmq.publisher-returns=true
```

2. 在MessageConfirmCallback类中，实现接口RabbitTemplate.ReturnCallback

```
@Component
public class RabbitConfirm implements RabbitTemplate.ConfirmCallback
    ,RabbitTemplate.ReturnCallback {
    //..省略
}
```

3. 并重写RabbitTemplate.ReturnCallback接口中returnedMessage()方法

```
/**
     * 当消息投递到交换机，交换机路由到消息队列中出现异常，执行returnedMessaged方法
     * @param message 投递消息内容
     * @param replyCode 返回错误状态码
     * @param replyText 返回错误内容
     * @param exchange 交换机名称
     * @param routingKey 路由键
     */
    @Override
    public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {
        System.out.println("交换机路由至消息队列出错：>>>>>>>");
        System.out.println("交换机："+exchange);
        System.out.println("路由键："+routingKey);
        System.out.println("错误状态码："+replyCode);
        System.out.println("错误原因："+replyText);
        System.out.println("发送消息内容："+message.toString());
        System.out.println("<<<<<<<<");
    }
```

4. 在RabbitTemplate中，设置消息发布回退回调方法

```
@PostConstruct
public void initRabbitTemplate(){
    //设置消息确认回调方法
    rabbitTemplate.setConfirmCallback(this::confirm);
    //设置消息回退回调方法
    rabbitTemplate.setReturnCallback(this::returnedMessage);
}
```

5. 请求测试失败执行returnedMessage方法：`http://localhost:8080/direct/sendMsg?exchange=order_exchange&routingkey=xxxxx&msg=购买苹果手机`

localhost:8080/direct/sendMsg?exchange=order_exchange&routingkey=xxxxx&msg=购买苹果手机

已投递~

路由键错误

```
交换机路由至消息队列出错：>>>>>>>
交换机：order_exchange
路由键：xxxxx
错误状态码：312
错误原因：NO_ROUTE
发送消息内容：(Body:'购买苹果手机' MessageProperties [headers={}, contentType=text/plain,
 contentEncoding=UTF-8, contentLength=0, receivedDeliveryMode=PERSISTENT, priority=0,
 deliveryTag=0])
<<<<<<<<
消息进入交换机成功{}
```

1.3 小结

确认模式：

- 设置publisher-confirms="true" 开启 确认模式。
- 实现RabbitTemplate.ConfirmCallback接口，重写confirm方法
- 特点：不论消息是否成功投递至交换机，都回调confirm方法，只有在发送失败时需要写业务代码进行处理。

退回模式

- 设置publisher-returns="true" 开启 退回模式。
- 实现RabbitTemplate.ReturnCallback接口，重写returnedMessage方法
- 特点：消息进入交换机后，只有当从exchange路由到queue失败，才去回调returnedMessage方法；

2. 消费者确认(ACK)

ack指 **Acknowledge**，拥有确认的含义，是消费端收到消息的一种确认机制；

消息确认的三种类型：

- 自动确认：acknowledge="**none**"
- 手动确认：acknowledge="**manual**"
- 根据异常情况确认：acknowledge="**auto**"，（这种方式使用麻烦，不作讲解）

其中自动确认是指，当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息缓存中移除。但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。

如果设置了手动确认方式，则需要在业务处理成功后，调用`channel.basicAck()`，手动签收，如果出现异常，则调用`channel.basicNack()`方法，让其自动重新发送消息。

MQ消息中间件
product
Exchange
routking_key
Queue
consumer

自己实现消费者签收代码：自定义监听器涉及三个对象：三个对象必须注入Spring容器

1. 自定义监听器对象
2. 自定义监听器的适配器Adaptor对象

3. 监听器的容器对象

可以使用RabbitTemplate中提供的签收方式：

2.1 代码实现

目标：演示消费者手动确认效果

自定义消费者接收消息监听器，监听收到消息的内容，手动进行签收；当业务系统抛出异常则拒绝签收，重回队列

实现步骤：

1. 搭建新的案例工程consumer-received-ack，用于演示ack消费者签收
2. 在消费者工程中，创建自定义监听器类CustomAckConsumerListener，实现ChannelAwareMessageListener接口
3. 编写监听器配置类ListenerConfiguration，配置自定义监听器绑定消息队列`order.A`
    - 注入消息队列监听器适配器对象到ioc容器
    - 注入消息队列监听器容器对象到ioc容器：
        - 配置连接工厂
        - 配置自定义监听器适配器对象
        - 配置消息队列
        - 开启手动签收
4. 启动消费者服务，观察控制台，消费者监听器是否与RabbitMQ建立Connection
5. 测试发送消息手动签收
6. 模拟业务逻辑出现异常情况
7. 测试异常情况，演示拒绝签收消息，消息重回队列

实现过程：

1. 搭建新的案例工程consumer-received-ack，搭建过程类似于生产者确认
    -

2. 在消费者工程中，创建自定义监听器类CustomAckConsumerListener，实现ChannelAwareMessageListener接口

```
/**
 * 自定义监听器，监听到消息之后，立即执行onMessage方法
 */
@Component
public class CustomAckConsumerListener implements
ChannelAwareMessageListener {
```

```java
    /**
     * 监听到消息之后执行的方法
     * @param message 消息内容
     * @param channel 消息所在频道
     */
    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        //获取消息内容
        byte[] messageBody = message.getBody();
        String msg = new String(messageBody, "utf-8");
        System.out.println("接收到消息，执行具体业务逻辑{} 消息内容："+msg);
        //获取投递标签
        MessageProperties messageProperties = message.getMessageProperties();
        long deliveryTag = messageProperties.getDeliveryTag();
        /**
         * 签收消息，前提条件，必须在监听器的配置中，开启手动签收模式
         * 参数1：消息投递标签
         * 参数2：是否批量签收：true一次性签收所有，false，只签收当前消息
         */
        channel.basicAck(deliveryTag,false);
        System.out.println("手动签收完成：{}");
    }
}
```

3. 编写监听器配置类ListenerConfiguration，配置自定义监听器绑定消息队列`order.A`

   - 注入消息队列监听器适配器对象到ioc容器
   - 注入消息队列监听器容器对象到ioc容器：
     - 配置连接工厂
     - 配置自定义监听器
     - 配置消息队列
     - 开启手动签收
   - ```java
     /**
      * 消费者监听器配置，将监听器绑定到消息队列上
      */
     @Configuration
     public class ListenerConfiguration {

         /**
          * 注入消息监听器适配器
          * @param customAckConsumerListener 自定义监听器对象
          */
         @Bean
         public MessageListenerAdapter messageListenerAdapter(CustomAckConsumerListener customAckConsumerListener){
             //创建自定义监听器适配器对象
             return new MessageListenerAdapter(customAckConsumerListener);
         }

         /**
          * 注入消息监听器容器
          * @param connectionFactory 连接工厂
     ```
     ```

```java
     * @param messageListenerAdapter 自定义的消息监听器适配器
     */
    @Bean
    public SimpleMessageListenerContainer simpleMessageListenerContainer(
            ConnectionFactory connectionFactory,
            MessageListenerAdapter messageListenerAdapter){

        //简单的消息监听器容器对象
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        //绑定消息队列
        container.setQueueNames("order.A");
        //设置连接工厂对象
        container.setConnectionFactory(connectionFactory);
        //设置消息监听器适配器
        container.setMessageListener(messageListenerAdapter);
        //设置手动确认消息：NONE(不确认消息)，MANUAL(手动确认消息)，AUTO(自动确认消息)
        container.setAcknowledgeMode(AcknowledgeMode.MANUAL);
        return container;
    }
}
```

4. 启动消费者控制，观察控制台，消费者监听器是否与RabbitMQ建立Connection

Overview | Connections | Channels | Exchanges | Queues | Admin

Cluster rabbit@32743bcfe0da
User heima Log out

Connections

All connections (1)

Pagination

Page 1 of 1 - Filter: Regex ?

Displaying 1 item , page size up to: 100

Overview				Details			Network	
Virtual host	Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
/itheima	192.168.200.1:50456 rabbitConnectionFactory#29526c05:0	heima	running	○	AMQP 0-9-1	1	2iB/s	0iB/s

5. 测试发送消息手动签收，请求地址http://localhost:8080/direct/sendMsg?exchange=order_exchange&routingkey=order.A&msg=购买苹果手机

- Spring Boot
  - Running
    - AckConsumerApplication
    - AckProducerApplication :8080/
  - Finished
    - ConsumerApplication (confirm)
    - ProducerApplication (confirm)
      - ProducerApplication (confirm)
      - ProducerApplication (confirm)

Console | Endpoints

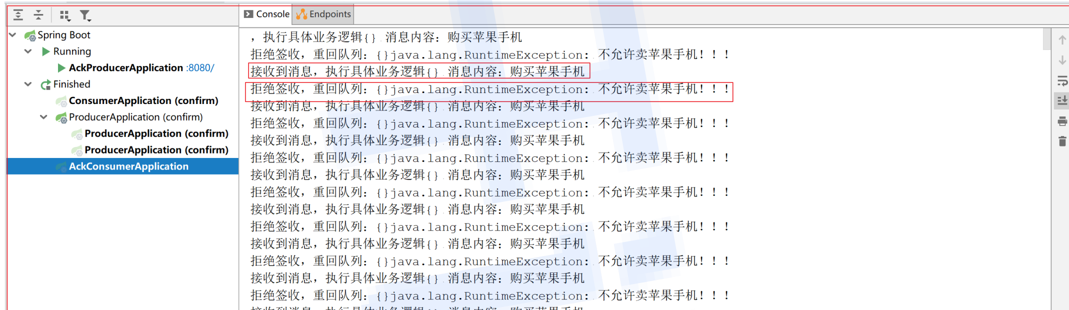
接收到消息，执行具体业务逻辑{} 消息内容：购买苹果手机
手动签收完成：{}

6. 模拟业务逻辑出现异常情况，修改自定义监听器

```java
@Override
public void onMessage(Message message, Channel channel) throws Exception {
    //获取消息内容
    byte[] messageBody = message.getBody();
    String msg = new String(messageBody, "utf-8");
    System.out.println("接收到消息，执行具体业务逻辑{} 消息内容："+msg);
    //获取投递标签
    MessageProperties messageProperties = message.getMessageProperties();
    long deliveryTag = messageProperties.getDeliveryTag();
    try {
        if (msg.contains("苹果")){
            throw new RuntimeException("不允许卖苹果手机！！！");
```

```java
            }
            /**
             * 手动签收消息
             * 参数1：消息投递标签
             * 参数2：是否批量签收：true一次性签收所有，false，只签收当前消息
             */
            channel.basicAck(deliveryTag,false);
            System.out.println("手动签收完成：{}");

        } catch (Exception ex){
            /**
             * 手动拒绝签收
             * 参数1：当前消息的投递标签
             * 参数2：是否批量签收：true一次性签收所有，false，只签收当前消息
             * 参数3：是否重回队列，true为重回队列，false为不重回
             */
            channel.basicNack(deliveryTag,false,true);
            System.out.println("拒绝签收，重回队列：{}"+ex);
        }
    }
```

7. 测试异常情况，演示拒绝签收消息，消息重回队列
    - 请求地址包含苹果，抛出异常：http://localhost:8080/direct/sendMsg?exchange=order_exchange&routingkey=order.A&msg=购买苹果手机
    - 控制台打印结果



2.2 小结

- 如果想手动签收消息，那么需要自定义实现消息接收监听器，实现ChannelAwareMessageListener接口
- 设置AcknowledgeMode模式
    - none：自动
    - auto：异常模式
    - manual：手动
- 调用channel.basicAck方法签收消息
- 调用channel.basicNAck方法拒签消息

06-消息的积压与过期问题

消息积压

场景：线上故障，几千万条数据在MQ里积压了7-8个小时。

这时，怎么处理呢？修复consumer端的BUG，然后等待8个小时消费完毕吗？肯定不行！一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条。如果你积压了上千万数据，即使消费者恢复，也需要2小时以上才能恢复。

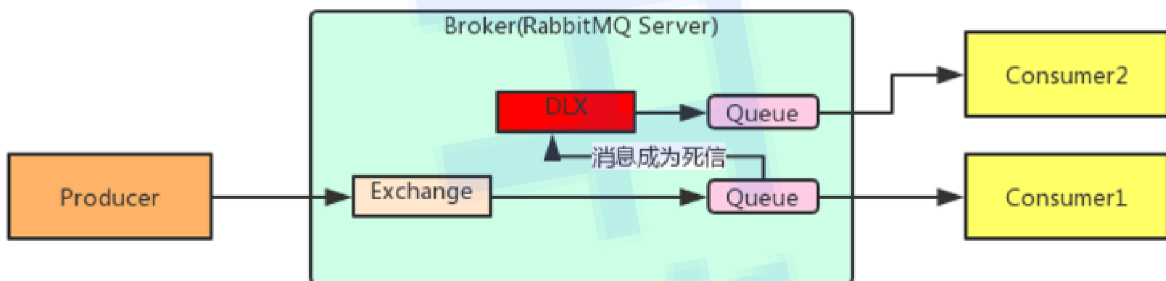
解决方案：临时扩容，加快消费速度

1. 先修复consumer的问题，确保其恢复消费速度，然后将现有consumer都停掉。
2. 临时建立好原先10倍或者20倍的queue数量
3. 然后写一个临时分发消息的consumer程序，这个程序部署上去消费积压的消息，消费之后不做耗时处理，直接均匀轮询写入临时建好分10数量的queue里面。
4. 紧接着征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的消息。
5. 这种做法相当于临时将queue资源和consumer资源扩大10倍，以正常速度的10倍来消费消息。
6. 等快速消费完了之后，恢复原来的部署架构，重新用原来的consumer机器来消费消息。

消息过期

消息过期、丢失，发送失败如何处理？任由消息消失？

死信队列：当消息成为Dead message后，可以被重新发送到另一个交换机，这个交换机就是Dead Letter Exchange（死信交换机 简写：DLX）。

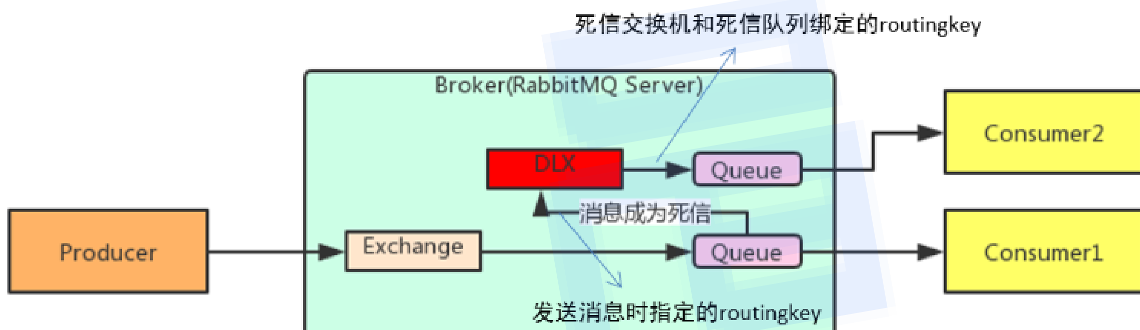


消息成为死信的三种情况：

1. 队列消息长度到达限制；
2. 消费者拒接消息(basicNack)，并且不把消息重新放回源队列，requeue=false；
3. 源队列存在消息过期设置，消息到达超时时间未被消费；

设置死信队列绑定死信交换机：

给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key



死信队列案例

目标：演示消息队列中消息超时失效

实现步骤：

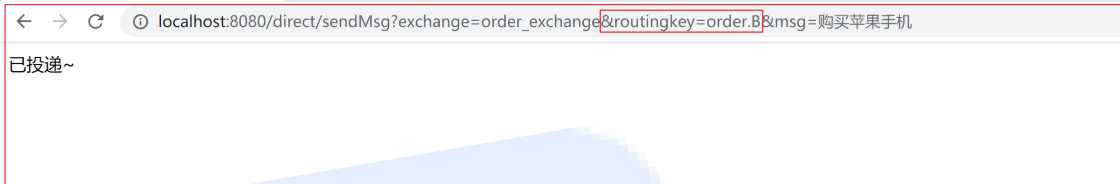
1. 在RabbitMQ管理控制台中，创建死信队列 `deadQueue`
2. 在RabbitMQ管理控制台中，创建死信交换机 `deadExchange`
3. 死信队列绑定死信交换机，路由键为 `order.dead`
4. 消息队列order.B绑定死信交换机
5. 向消息队列 `order.B` 中发送消息【消息队列order.B中的消息失效时间为5秒】
6. 在RabbitMQ管理控制台中，将消息队列 `order.B` 绑定到交换机 `order_exchange` 上
7. 等待5秒，消息队列order.B中的消息进入死信队列

实现过程：

1. 在RabbitMQ管理控制台中，创建死信队列 `deadQueue`
2. 在RabbitMQ管理控制台中，创建死信交换机 `deadExchange`
3. 死信队列绑定死信交换机，路由键为 `order.dead`
4. 删除 `order.B` 消息队列，重建之后绑定死信交换机

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/itheima	deadQueue	classic	D Args	idle	0	0	0			
/itheima	order.A	classic	D Args	idle	0	0	0	0.00/s	0.00/s	0.00/s
/itheima	order.B	classic	D TTL DLX DLK Args	idle	0	0	0			

5. 向消息队列`order.B`中发送消息【消息队列order.B中的消息失效时间为5秒】



6. 等待5秒，消息队列order.B中的消息进入死信队列

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/itheima	deadQueue	classic	D Args	idle	2	0	2			
/itheima	order.A	classic	D Args	idle	0	0	0	0.00/s	0.00/s	0.00/s
/itheima	order.B	classic	D TTL DLX DLK Args	idle	0	0	0	0.00/s		

小结

1. 死信交换机和死信队列和普通的没有区别
2. 当消息成为死信后，如果该队列绑定了死信交换机，则消息会被死信交换机重新路由到死信队列
3. 消息成为死信的三种情况：
   - 队列消息长度到达限制；
   - 消费者拒接消费消息，并且不重回队列；
   - 原队列存在消息过期设置，消息到达超时时间未被消费；

07-常见消息队列产品优缺点

这四者，对比如下表格如下：

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
公司/社区	Apache	Rabbit	阿里	Apache
开发语言	Java	Erlang	Java	Java&Scala
协议支持	OpenWire、STOMP、REST,XMPP,AMQP	AMQP、XMPP、SMTP、STOMP	自定义	自定义协议，社区封装了http协议支持
单机吞吐量	万级（最差）	万级（其次）	十万级（最好）	十万级（次之）
Topic数量&吞吐量			**topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic**	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
消息延迟	毫秒级	**微秒级，这是 RabbitMQ 的一大特点，延迟最低**	毫秒级	毫秒级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	可靠性高	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

ActiveMQ

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ ，但是现在确实大家用的不多了(特别是互联网公司)，没经过大规模吞吐量场景的验证(**性能较差**)，社区也不是很活跃(主要精力在研发 ActiveMQ Apollo)，所以大家还是算了，不推荐用。

RabbitMQ【建议】

后来大家开始用 RabbitMQ，但是确实 Erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，社区活跃度也高。Spring Cloud 在消息队列的支持上，对 RabbitMQ 是比较不错的，所以在选型上又更加被推崇。

RocketMQ

不过现在确实越来越多的公司，会去用 RocketMQ，确实很不错（阿里出品）。目前已经加入 Apache，所以社区层面有相应的保证，并且是使用 Java 语言进行实现，对于 Java 工程师更容易去深入研究和掌控它。目前，也是比较推荐去选择的。并且，如果使用阿里云，可以直接使用其云服务。

Kafka

总结

- **中小型公司**:
    - 技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择
    - 中小型公司使用 RocketMQ 也是没什么问题的选择，特别是以 Java 为主语言的公司。
- 大型公司:
    - 基础架构研发实力较强，用 RocketMQ 是很好的选择。
- 如果是大数据领域的实时计算、日志采集等场景:
    - 用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。
    - 目前国内也是有非常多的公司，将 Kafka 应用在业务系统中，例如唯品会、陆金所、美团等等。

08-消息队列的一般存储方式

当前业界几款主流的MQ消息队列采用的存储方式主要有以下三种方式。

1. 文件系统

目前业界较为常用的几款产品（RocketMQ / Kafka / RabbitMQ）均采用的是消息刷盘至所部署虚拟机/物理机的文件系统来做持久化

- 刷盘一般可以分为异步刷盘、同步刷盘两种模式

消息刷盘为消息存储提供了一种高效率、高可靠性和高性能的数据持久化方式。除非部署 MQ 机器本身或是本地磁盘挂了，否则一般是不会出现无法持久化的故障问题。

2. 分布式KV存储【自定义】

这类 MQ 一般会采用诸如 LevelDB 、RocksDB 和 Redis 来作为消息持久化的方式。

由于分布式缓存的读写能力要优于 DB ，所以在对消息的读写能力要求都不是比较高的情况下，采用这种方式倒也不失为一种可以替代的设计方案。

消息存储于分布式 KV 需要解决的问题在于如何保证 MQ 整体的可靠性。

3. 关系型数据库 DB

Apache下开源的另外一款MQ—ActiveMQ（默认采用的KahaDB做消息存储）可选用 JDBC 的方式来做消息持久化，通过简单的 XML 配置信息即可实现JDBC消息存储。

由于，普通关系型数据库（如 MySQL ）在单表数据量达到千万级别的情况下，其 IO 读写性能往往会出现瓶颈。因此，如果要选型或者自研一款性能强劲、吞吐量大、消息堆积能力突出的 MQ 消息队列，那么并不推荐采用关系型数据库作为消息持久化的方案。在可靠性方面，该种方案非常依赖 DB ，如果一旦 DB 出现故障，则 MQ 的消息就无法落盘存储会导致线上故障。

小结

- 存储效率：**文件系统 > 分布式 KV 存储 > 关系型数据库 DB**
- 直接操作文件系统肯定是最快和最高效的，而关系型数据库 TPS 一般相比于分布式 KV 系统会更低一些
  - 简略地说，关系型数据库本身也是一个需要读写文件 Server ， 这时 MQ 作为 Client与其建立连接并发送待持久化的消息数据，同时又需要依赖 DB 的事务等，这一系列操作都比较消耗性能
- MQ的存储方式如何取舍：
  - 如果追求高效的IO读写，那么选择操作文件系统会更加合适一些
  - 如果从易于实现和快速集成来看，**文件系统 < 分布式 KV 存储 < 关系型数据库 DB**，但是性能会下降很多。
  - 从消息中间件的本身定义来考虑，应该尽量减少对于外部第三方中间件的依赖。

思考题：如何自己设计一个消息队列

二、存储

今日主要的内容介绍

送分题：选择，填空，大题【输出解题思想】

1、行存储与列存储

1. 行存储和列存储有什么优势和劣势？
2. Mysql是行存储还是列存储？
3. Mongodb是行存储还是列存储？
4. Hbase是行存储还是列存储？
5. Google怎么存海量爬取的结果？

2、事务

1. 什么是事务？解决什么问题？
2. ACID是什么，分别解决什么问题？
3. 给我讲讲事务隔离级别？
4. 给我讲讲MVCC？
5. 读锁和写锁的作用是什么？
6. select for update用的是读锁还是写锁？

3、索引

1. 索引是什么?
2. Mysql的索引是正排索引还是倒排索引
3. 说几个Mysql中使用索引不当的例子
4. Mysql为什么不用红黑树索引?
5. 索引需要占用多少空间?
6. B树和哈希表索引的优势和劣势?

4、亿行数据

1. 表太大的一张Disk存不下怎么办?
2. 一些旧数据（比如半年前的支付记录）想备份后从表中移走，怎么办?
3. 数据量太大了查询速度下降怎么办?
4. 表太大导致I/O瓶颈（读取速度慢），怎么办?
5. 网络带宽不够，该怎么办?

100000w行数据怎么存储?

10TB数据怎么存储?

01-行式存储与列式存储

1.1 OLTP与OLAP

当今的数据处理大致可分为两大类

1. 联机事务处理 OLTP (on-line **transaction** processing)
    - OLTP 是传统关系型数据库的主要应用，用来执行一些基本的、日常的**事务**处理， 比如数据库记录的增、删、改、查等等
    - 不适合海量数据处理
    - ACID
    - 串行化【单线程】： 事务之间相互影响，要么锁住，(写锁)
2. 联机分析处理 OLAP (On-Line **Analytical** Processing)
    - OLAP 是分布式数据库的主要应用，它对实时性要求不高，但处理的数据量大，通常应用于复杂的动态报表系统上。

1.2 什么是行式存储？什么又是列式存储?

行式存储法(Row-based)：数据一行行存储

- 传统的关系型数据库，如 **Oracle**、**DB2**、**MySQL**、**SQL SERVER** 等采用行式存储法(Row-based)
- 在基于行式存储的数据库中， 数据是按照行数据为基础逻辑存储单元进行存储的， 一行中的数据在存储介质中以连续存储形式存在。

列式存储(Column-based)：数据一列列存储

- 列式是相对于行式存储来说的，新兴的 **Hbase**、HP Vertica、EMC Greenplum 等分布式数据库均采用列式存储。
- 在基于列式存储的数据库中， 数据是按照列为基础逻辑存储单元进行存储的，一列中的数据在存储介质中以连续存储形式存在。

- ElasticSearch、MongoDB、Solr、Splunk...
- Redis(K/V)、

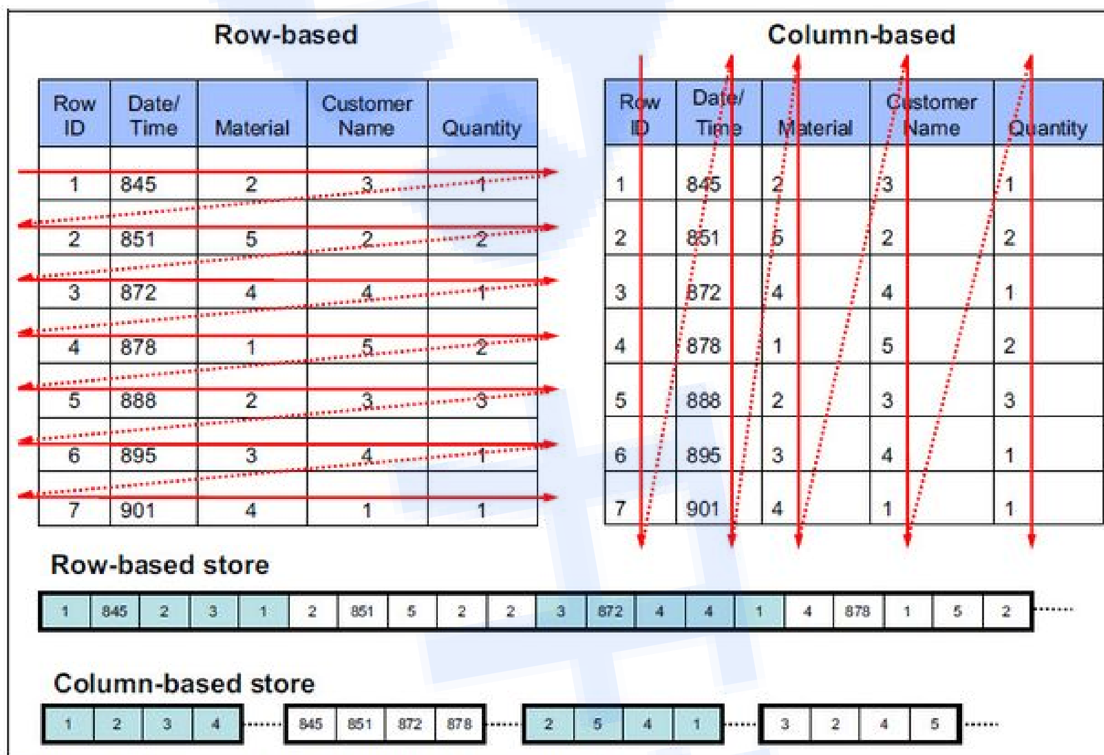


Figure 1-4 Row-based and column-based storage models

> ps：MongoDB，属于文档存储，存储一个JSON(BSON)文档、或者一个文本（二进制）文档
>
> - 类似：elasticserach，solr，oss......
> - 也被称为NoSql

1.3 行式存储 适用场景：

- 适合随机的增删改查操作;
- 需要在行中选取所有属性的查询操作;
- 需要频繁插入或更新的操作，其操作与索引和行的大小更为相关。

行式存储实操中，我们会发现行式数据库在读取数据时存在一个固有的“缺陷”。这个"缺陷"是什么？不擅长在海量数据中找数据。

1.4 列式存储 适用场景：

- **低延迟：**查询过程中，可针对各列的运算并发执行，最后在内存中聚合完整记录集，最大可能降低查询响应时间;
- **查找数据高效：**无需维护索引，查询过程中能够尽量减少无关IO，避免全表扫描
- **节省空间：**因为各列独立存储，且数据类型已知，可以针对该列的数据类型、数据量大小等因素动态选择压缩算法，以提高物理存储利用率；如果某一行的某一列没有数据，那在列存储时，就可以

不存储该列的值，这将比行式存储更节省空间。

当然，跟行数据库一样，列式存储也有不太适用的场景，这个场景是什么？

- 数据需要频繁更新的交易场景，表中列属性较少的小量数据库场景，不适合做含有删除和更新的实时操作

1.5 行存储 vs 列存储【读、写、删】

1.5.1 读取数据分析

假设：

- **从2^30行中取出1列数据 100001 ~ 200000行的name, status列**

行存储读取成本分析：

- 场景：读取10W行数据的2列
  - 先取行，再取列：从连续的空间取出10w行，从10w行结果集合中取出name和status列
  - **1行平均10KB：**10W行=10W * 10kb = ≈ 1G数据
  - Buffer如何是4M，将进行256次读取
- 10W过滤操作

列存储读取成本分析：

- 场景：读取10W行数据的2列
  - 先取列，再取行：从2列中取出10w条
  - 1列的一个条目平均20Byte，10W行的两列=20 byte * 10W ≈ 2M
  - Buffer如何是4M，将进行1次读取
- 不需要更多的过滤操作

1.5.2 更新一行多个值分析

- 行存储：一行数据
- 列存储：多列数据（次数多）

1.5.3 添加一行数据分析

- 行存储：一行数据
- 列存储：多列数据（次数多）

1.5.4 事务操作分析

- 行存储：锁一行数据
- 列存储：锁对应的列（范围大）

02-什么是事务?

事务概述

- 事务指的是逻辑上的一组操作，组成这组操作的各个单元要么全都成功，要么全都失败。
- 事务作用：保证在一个事务中多次SQL操作要么全都成功，要么全都失败。

事务基本特性

（ACID，是针对单个事务的一个完美状态）

- **原子性（Atomicity）**：原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
- **一致性（Consistency）**：事务前后数据的完整性必须保持一致。**保证一致性的工具：锁**

- **隔离性（Isolation）**：事务的隔离性是指多个用户**并发**访问数据库时，一个用户的事务不能被其它用户的事务所干扰，多个并发事务之间数据要相互隔离。**隔离性由隔离级别保障！**
  - 正常情况下数据库是做不到完完全全的隔离，可以增强隔离级别，但是效率会非常低。
  - read uncommitted --> read committed --> repeatable read --> serializable 【MySQL默认隔离级别RR】
    - 事务并发问题：**脏读、不可重复读**，幻读
    - 丢失更新的问题！
  - **MVCC：multiple version concurrency control(多版本并发控制）为什么需要MVCC?**
  - 能不能在隔离性和效果之间找一个平衡点呢?
- **持久性（Durability）**：持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。
  - **能不能做到100%?**

事务并发问题【事务隔离不足导致】

如果不考虑隔离性，事务存在3中并发访问问题。

1. 脏读：一个事务读到了另一个事务**未提交**的数据
2. 不可重复读：一个事务读到了另一个事务**已经提交**(update)的数据。引发另一个事务，在事务中的多次查询结果不一致。
3. 虚读 /幻读：一个事务读到了另一个事务已经**插入(insert)**的数据。导致另一个事务，在事务中多次查询的结果不一致。

事务隔离级别

数据库规范规定了4种隔离级别，分别用于描述两个事务并发的所有情况。

- **read uncommitted** 读未提交，一个事务读到另一个事务没有提交的数据。
  - 存在：3个问题（脏读、不可重复读、幻读）。
  - 解决：0个问题
- **read committed** 读已提交，一个事务读到另一个事务已经提交的数据。
  - 存在：2个问题（不可重复读、幻读）。
  - 解决：1个问题（脏读）
- **repeatable read**:可重复读，在一个事务中读到的数据始终保持一致，无论另一个事务是否提交。
  - 存在：1个问题（幻读）。
  - 解决：2个问题（脏读、不可重复读）
- **serializable 串行化**，同时只能执行一个事务，相当于事务中的单线程。
  - 存在：0个问题。
  - 解决：3个问题（脏读、不可重复读、幻读）

安全和性能对比

- 安全性：`serializable > repeatable read > read committed > read uncommitted`
- 性能：`serializable < repeatable read < read committed < read uncommitted`

常见数据库的默认隔离级别：

- MySql：`repeatable read`
- Oracle：`read committed`

事务案例：

- 设置数据库的隔离级别
  - `set session transactionisolation level` 级别字符串
  - 级别字符串：`read uncommitted`、`read committed`、`repeatable read`、`serializable`
  - 例如：`set session transaction isolation level read uncommitted;`
- 读未提交：read uncommitted

时间	事务A	事务B
T0	设置A事务隔离级别【读未提交】	设置B事务隔离级别【读未提交】
T1	开始A事务	开始B事务
T2	查询	
T3		更新数据
T4	再次查询：查询到B未提交数据【脏读】	
T5		回滚
T6	再次查询：B未提交数据消失	
T7	A事务提交【必须提交否则会对下次测试产生影响】	B事务提交

  - 注意：完成之后必须结束事务，commit，

- 读已提交：read committed

时间	事务A	事务B
T0	设置A事务隔离级别【读已提交】	设置B事务隔离级别【读已提交】
T1	开始A事务	开始B事务
T2	查询	
T3		更新数据
T4	再次查询：数据没变，解决脏读问题	
T5		B事务提交
T6	再次查询：数据改变，存在不可重复读问题	
T7	A事务提交	

- 可重复读：repeatable read

时间	事务A	事务B
T0	设置A事务隔离级别【可重复读】	设置B事务隔离级别【可重复读】
T1	开始A事务	开始B事务
T2	查询	
T3		更新数据
T4	再次查询：数据没变，解决脏读问题	
T5		B事务提交
T6	再次查询：数据没变，解决不可重复问题	
T7	A事务提交	

- 串行化：serializable
时间	事务A	事务B
T0	设置A事务隔离级别【串行化】	设置B事务隔离级别【串行化】
T1	开始A事务	开始B事务
T2	查询	
T3		更新数据--提示等待，如果A没有进一步操作B将等待超时
T4	A事务提交或者回滚	
T5		B等待结束，执行操作
T6		B事务提交

03-保证一致性的工具：锁

- 锁的范围：行锁(row)/表锁(table)/全局锁(db)
  - 全局锁：锁的是整个database。由MySQL的SQL layer层实现的
  - 表级锁：锁的是某个table。由MySQL的SQL layer层实现的
  - 行级锁：锁的是某行数据，也可能锁定行之间的间隙。由某些存储引擎实现，比如InnoDB。
- 按照锁的功能来说分为：**共享锁**和**排他锁**。
- 共享锁Shared Locks(**读锁**)：允许其他读(select for share)，不允许写(update或select for update)
  - 兼容性：加了共享锁的记录，允许其他事务再加共享锁，不允许其他事务再加排他锁
  - 加锁方式：select...lock in share mode
    - 触发表级共享锁：select * from table for ...
    - 触发行级共享锁：select * from table where id=1 for updata ...

- 排他锁Exclusive Locks(**写锁**)：不允许其他读(select for share)和写(select for update)
    - 兼容性：加了排他锁的记录，不允许其他事务再加共享或者排他锁
    - 加锁方式：select…for update

04-什么是MVCC?

MVCC（多版本的并发控制，英文全称：Multi Version Concurrency Control）机制主要用来解决事务中的**丢失更新问题**。MVCC是用于数据库提供并发访问控制的并发控制技术。与MVCC相对的是基于锁的并发控制，`Lock-Based Concurrency Control`（LBCC）。

MVCC最大的好处，相信也是耳熟能详：**读不加锁，读写不冲突。**

在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。

多版本并发控制：仅仅是一种技术概念，并没有统一的实现标准， 其核心理念就是数据快照，不同的事务访问不同版本的数据快照，从而实现不同的事务隔离级别。虽然字面上是说具有多个版本的数据快照，但这并不意味着数据库必须拷贝数据，保存多份数据文件，这样会浪费大量的存储空间。InnoDB通过事务的undo日志巧妙地实现了多版本的数据快照。

- MVCC 在mysql 中的实现依赖的是 **undo log** 与 **read view** 。

InnoDB的MVCC是通过在每行记录后面保存两个隐藏的列来实现的。这两个列，一个保存了行的事务ID，一个保存了行的回滚指针。每开始一个新的事务，都会自动递增产生一个新的事务id。事务开始时刻的会把事务id放到当前事务影响的行事务id中，当查询时需要用当前事务id和每行记录的事务id进行比较。

注意：MVCC只在**REPEATABLE READ**和**READ COMMITIED**两个隔离级别下工作

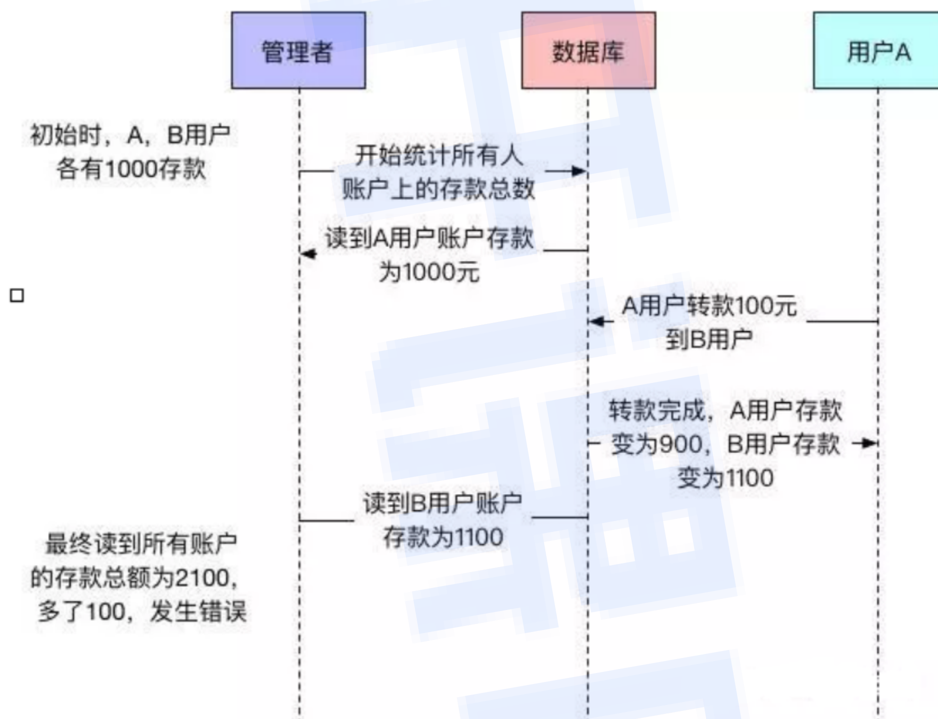
什么是丢失更新？举两个栗子

案例1：两个事务针对同一数据都发生修改操作时，会存在丢失更新问题。

时间	取款事务A	取款事务B
T0	开始事务	
T1		开始事务
T2	查询账户余额1000元	
T3		查询账户余额1000元
T4		汇入100元，把余额改为1100元
T5		提交事务
T6	取出100元，把余额改为900元	
T7	撤销事务	
T8	余额恢复为1000元【丢失更新】	

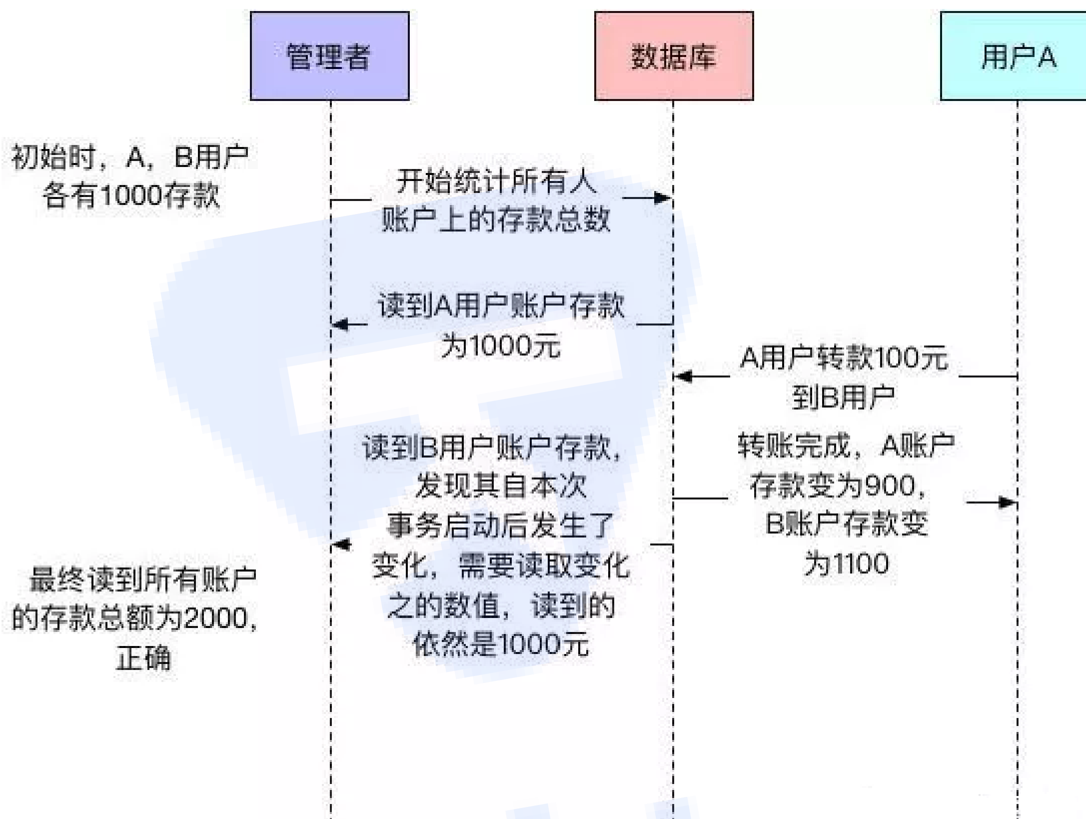
案例2：管理者要查询所有用户的存款总额，假设除了用户A和用户B之外，其他用户的存款总额都为0，A、B用户各有存款1000，所以所有用户的存款总额为2000。但是在查询过程中，用户A会向用户B进行转账操作。转账操作和查询总额操作的时序图如下图所示。

- **转账和查询的时序图：**



使用MVCC机制

查询总额事务先读取了用户A的账户存款，然后转账事务会修改用户A和用户B账户存款，查询总额事务读取用户B存款时不会读取转账事务修改后的数据，而是读取本事务开始时的数据副本【在REPEATABLE READ隔离等级下】



MVCC使得数据库读不会对数据加锁，普通的SELECT请求不会加锁，提高了数据库的并发处理能力。借助MVCC，数据库可以实现READ COMMITTED，REPEATABLE READ等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I特性（隔离性)。

MVCC下的读操作

在MVCC并发控制中，读操作可以分成两类：**快照读 (snapshot read)与当前读 (current read)。**

> **快照读**：读取的是记录的可见版本 (有可能是历史版本)，不用加锁。
>
> **当前读**：读取的是记录的最新版本，并且当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？

- **快照读**：简单的select操作，属于快照读，不加锁。(当然，也有例外，下面会分析） 不加读锁 读历史版本
- **当前读**：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。 加行写锁 读当前版本

```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert into table values (…);
update table set ? where ?;
delete from table where ?;
-- 所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加共享锁外，其他的操作，都加的是排它锁。
```

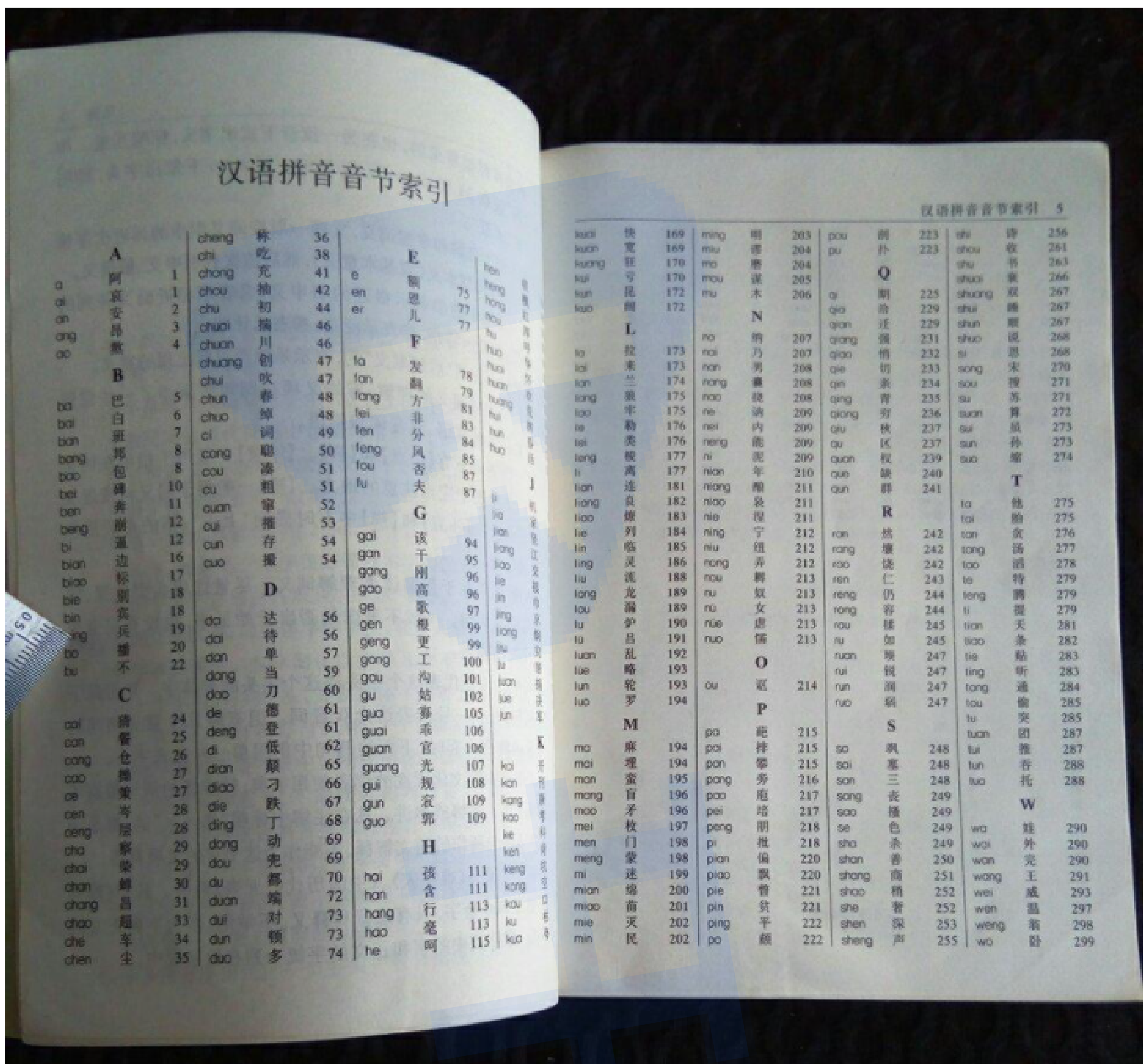
小结

- MVCC是指执行普通的`SEELCT`操作时访问记录的**版本链**的过程
- 目标是使不同事务的`读-写`、`写-读`操作并发执行，从而提升系统性能。
- **前提：`READ COMMITTD`、`REPEATABLE READ`这两种隔离级别的事务**

05-什么是索引?

索引是什么

- **索引是高效获取数据的数据结构。**
- 作用：加速查询
- 一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往是存储在磁盘上的文件中的。
- 我们通常所说的索引，包括聚集索引、覆盖索引、组合索引、前缀索引、唯一索引等，没有特别说明，默认都是使用B+树结构组织的索引。



优势：

- 检索效率：可以提高数据检索的效率，降低数据库的IO成本，类似于书的目录。
- 快速排序：通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗。
  - 被索引的列会自动进行排序，包括【单列索引】和【组合索引】，只是组合索引的排序要复杂一些。
  - 如果按照索引列的顺序进行排序，对应order by语句来说，效率就会提高很多。

劣势：

- 索引会占据磁盘空间
- 索引虽然会提高查询效率，但是会降低更新表的效率。比如每次对表进行增删改操作，MySQL不仅要保存数据，还有保存或者更新对应的索引文件。

正排索引与倒排索引

正排和倒排值得是什么呀？**索引的功能**

举例：一张user数据表【id、name、status】

- **正排索引：查询张三这个名字在哪个文档**
  - 文档 -> 词
  - 索引格式：<文档, List.of(...词)>
- **倒排索引：查询文档里有哪些词汇**
  - 词 -> 文档编号 -> 文档doc

- 索引格式：<词, 文档编号>

请问mysql的索引是什么？正排还是倒排？

> 根据id查行，全表扫描
>
> 加了索引 --> B+树的索引

06-MySQL的索引应该用什么？

6.1 MySQL对索引的要求

索引的数据结构，至少需要支持两种最常用的查询需求：

1. 等值查询：根据某个值查找数据，比如：`select * from t_user where age=76;`
2. 范围查询：根据某个范围区间查找数据，比如：`select * from t_user where age>=76 and age<=86;`
3. 排序
4. 分组
5. ..

同时需要考虑时间和空间因素：**性价比高**

- 在执行时间方面，我们希望通过索引，查询数据的时间尽可能小；
- 在存储空间方面，我们希望索引不要消耗太多的内存空间和磁盘空间。

6.2 MySQL不用索引行不行？

- 行不行？完全可以
- 时间复杂度O(n)
- 插入和删除使得存储数据产生碎片

> 用不用的选择权在谁手里？

6.3 Hash表

Hash表，Java中的HashMap，TreeMap就是Hash表结构，以键值对的方式存储数据。我们使用Hash表存储表数据Key可以存储索引列，Value可以存储行记录或者行磁盘地址。Hash表在等值查询时效率很高，时间复杂度为O(1)；

- 但是不支持范围快速查找，范围查找时还是只能通过扫描全表方式。
- 数据结构比较稀疏，不适合做聚合，不适合做范围等查找。

使用场景：

- 对查询并发要求很高，**K/V内存数据库，缓存**

6.4 二叉查找树

- 二叉树特点：每个节点最多有2个分叉，左子树和右子树数据顺序左小右大。
- 二叉树的检索复杂度和树高相关：**理想状态**下效率可以达到O(logn)
- **那是不是任何列使用二叉树效率都会提升呢？答案是否定的。**

极端情况下，二叉查找树会构建成为单向链表=查找全表扫描。

对磁盘不友好【一旦变成了全表扫描，磁盘io将是极其沉重】

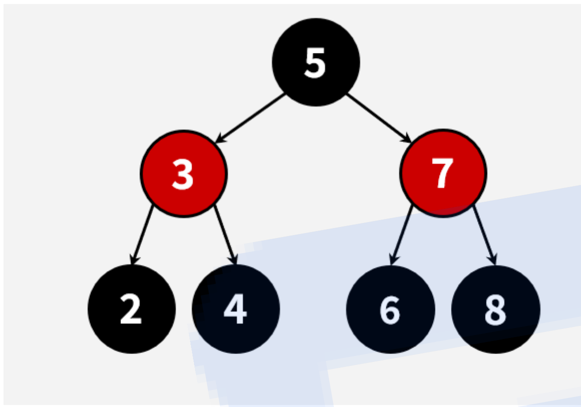
6.5 红黑树

> 红黑树是一个近似平衡二叉树
>
> 在建立mysql的索引的时候，要谨慎

平衡二叉树是采用二分法思维，平衡二叉查找树除了具备二叉树的特点，最主要的特征是树的左右两个子树的层级**最多相差1**。在插入删除数据时通过左旋/右旋操作保持二叉树的平衡，不会出现左子树很高、右子树很矮的情况。

使用平衡二叉查找树查询的性能接近于二分查找法，时间复杂度是 O(log2n)。

> unique key 为什么不用红黑树，反正只存一个主键？



平衡二叉树存在的问题

1. 时间复杂度和树高相关：树有多高就需要检索多少次，每个节点的读取，都对应一次磁盘 IO 操作【瓶颈】。
    - **磁盘每次寻道时间为10ms**，在表数据量大时，对响应时间要求高的场景下，查询性能就会出现瓶颈。
    - 举例：1百万的数据量，log2n约等于20次磁盘IO，时间20*10=0.2s
2. 平衡二叉树不支持范围查询快速查找，范围查询时需要从根节点多次遍历，查询效率极差。
3. 数据量大的情况下，索引存储空间占用巨大

- 10亿行数据，时间复杂度O(logn)，最多不超过30次查到数据
- 最简单索引构成：<ID，行号，指针>
- 假如key为bigint=8字节，每个节点有两个指针，每个指针为4个字节，一个节点占用的空间16个字节（8+4*2=16）。
- 索引大小：10亿x16(bigint)=15GB

如何减少IO操作次数呢?

如何才能降低存储空间呢?

6.6 B树：改进二叉树，为多叉树

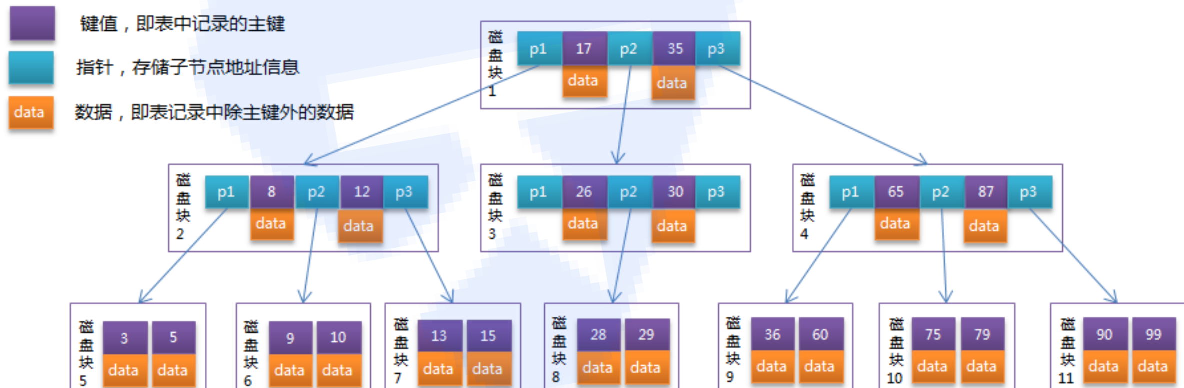
多想要减少耗时的IO操作，就要尽量降低树的高度。每个节点存储多个元素，在每个节点尽可能多的存储数据。每个节点可以存储1000个索引（16k/16=1000），这样就将二叉树改造成了多叉树，通过增加树的叉树，将树从高瘦变为矮胖。

举例：构建1百万条数据，树的高度只需要2层就可以（1000*1000=1百万），也就是说只需要2次磁盘IO就可以查询到数据。磁盘IO次数变少了，查询数据的效率也就提高了。

主要特点：

1. B树的节点中存储着多个元素，每个内节点有多个分叉。
2. 节点中的元素包含键值和数据，节点中的键值从大到小排列。也就是说，在所有的节点都储存数据。
3. 父节点当中的元素不会出现在子节点中。
4. 所有的叶子结点都位于同一层，叶节点具有相同的深度，叶节点之间没有指针连接。

以下面的B树为例，我们的键值为表主键，具备唯一性。



B树如何查询数据？：假如我们查询值等于15的数据。查询路径磁盘块1->磁盘块2->磁盘块7。

优点：

- 磁盘IO次数会大大减少。
- 比较是在内存中进行的，比较的耗时可以忽略不计。
- B树的高度一般2至3层就能满足大部分的应用场景，所以使用B树构建索引可以很好的提升查询的效率。

缺点：

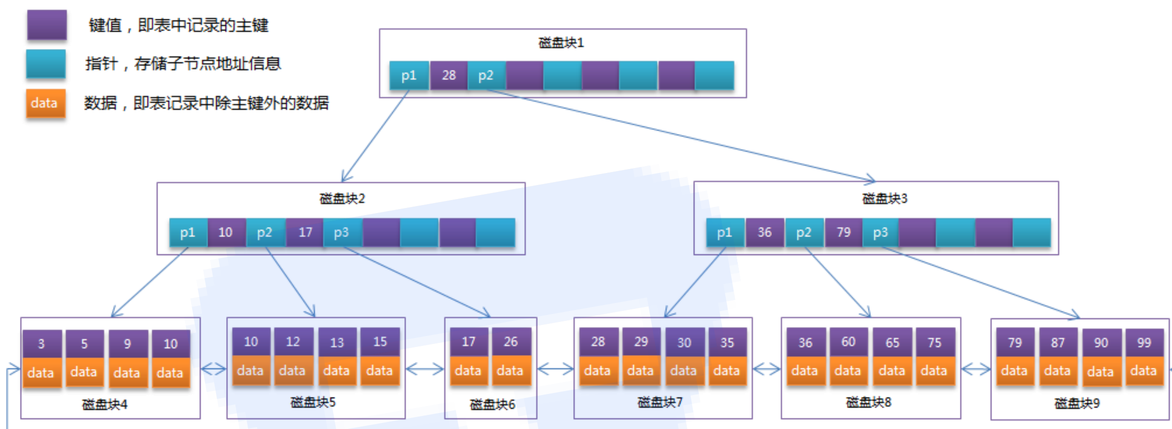
- **B树不支持范围查询的快速查找**：如果我们想要查找15和26之间的数据，查找到15之后，需要回到根节点重新遍历查找，需要从根节点进行多次遍历，查询效率有待提高。
- **空间占用较大**：如果data存储的是行记录，行的大小随着列数的增多，所占空间会变大。一个页中可存储的数据量就会变少，树相应就会变高，磁盘IO次数就会变大。

6.7 B+树：改进B树，非叶子节点不存存储数据

在B树基础上，MySQL在B树的基础上继续改造，使用B+树构建索引。B+树和B树最主要的区别在于**非叶子节点是否存储数据**的问题

- B树：非叶子节点和叶子节点都会存储数据。
- B+树：只有叶子节点才会存储数据，非叶子节点至存储键值。叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表。

B+树的最底层叶子节点包含所有索引项。具备中路返回特性



等值查询：假如我们查询值等于15的数据。查询路径磁盘块1->磁盘块2->磁盘块5。

范围查询：假如我们想要查找15和26之间的数据。

- 查找路径是磁盘块1->磁盘块2->磁盘块5。
- 首先查找值等于15的数据，将值等于15的数据缓存到结果集【三次磁盘IO】。
- 查找到15之后，底层的叶子节点是一个有序列表，我们从磁盘块5，键值15开始向后遍历筛选所有符合筛选条件的数据。
- 第四次磁盘IO：根据磁盘5后继指针到磁盘中寻址定位到磁盘块6，将磁盘6加载到内存中，在内存中从头遍历比较，15<17<26，15<26<=26，将data缓存到结果集。

优点：

- 继承了B树的优点【多叉树的优点】
- 保证等值和范围查询的快速查找
- MySQL的索引就采用了B+树的数据结构。

07-索引使用不当的情况有哪些?

1. 主键索引：使用UUID等无序主键
2. 大字段建立索引
3. 使用区分度不大的列作为索引【gender：man，woman】
4. Order By 和Group By后的字段没有建立索引
5. 查询条件中使用函数
6. 频繁建立索引
7. 频繁增删改查的字段建立索引
8. 查询SQL索引失效【or，like..】

08-单节点MySQL的瓶颈在哪?

> 你是个天才，你浑身是铁，碾的了多少钉子

MySQL单机的存储能力、连接数是有限的，它自身就很容易会成为系统的瓶颈。

当单表数据量在百万以里时，我们还可以通过添加从库、优化索引提升性能。

数据量朝着千万以上趋势增长，再怎么优化数据库，很多操作性能仍下降严重。为了减少数据库的负担，提升数据库响应速度，缩短查询时间，这时候就需要进行`分库分表`。

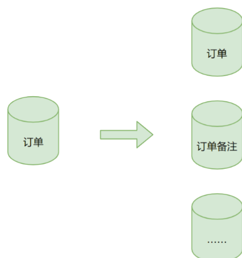
8.1 什么是分库分表?

分库分表就是要将大量数据分散到多个数据库中，使每个数据库中数据量小响应速度快，以此来提升数据库整体性能。核心理念就是对数据进行切分（`Sharding`），以及切分后如何对数据的快速定位与整合。

针对数据切分类型，大致可以分为：垂直（纵向）切分和水平（横向）切分两种。

8.2 垂直拆分：根据业务逻辑拆分

- 垂直分库：安装业务垂直拆分数据库
- 垂直分表：



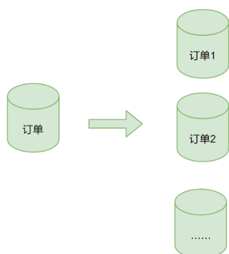
优点：

- 业务间解耦，不同业务的数据进行独立的维护、监控、扩展
- 在高并发场景下，一定程度上缓解了数据库的压力

缺点：

- 提升了开发的复杂度，由于业务的隔离性，很多表无法直接访问，必须通过接口方式聚合数据，
- 分布式事务管理难度增加
- 数据库还是存在单表数据量过大的问题，并未根本上解决，需要配合水平切分

8.3 水平拆分：根据sharding规则拆分



库内分表：库内分表虽然将表拆分，但子表都还是在同一个数据库实例中，只是解决了单一表数据量过大的问题，并没有将拆分后的表分布到不同机器的库上，还在竞争同一个物理机的CPU、内存、网络IO。

分库分表：分库分表则是将切分出来的子表，分散到不同的数据库中，从而使得单个表的数据量变小，达到分布式的效果。

优点：

- 解决高并发时单库数据量过大的问题，提升系统稳定性和负载能力
- 业务系统改造的工作量不是很大

缺点：

- 跨分片的事务一致性难以保证
- 跨库的join关联查询性能较差
- 扩容的难度和维护量较大

8.4 有哪些sharding规则

hash取模

- 优点：
  - 数据分片相对比较均匀，不易出现某个库并发访问的问题
  - 这样同一个用户的数据都会存在同一个库里，用 `userId` 作为条件查询就很好定位了
- 缺点：
  - 但这种算法存在一些问题，当某一台机器宕机，本应该落在该数据库的请求就无法得到正确的处理，这时宕掉的实例会被踢出集群，此时算法变成hash(userId) mod N-1，用户信息可能就不再在同一个库中。
  - 扩展性较差

取值范围，举例：根据年代

- 优点：
  - 单表数据量是可控的
  - 水平扩展简单只需增加节点即可，无需对其他分片的数据进行迁移
  - 能快速定位要查询的数据在哪个库
- 缺点：
  - 由于连续分片可能存在数据热点，如果按时间字段分片，有些分片存储最近时间段内的数据，可能会被频繁的读写，而有些分片存储的历史数据，则很少被查询

三、缓存

1. 什么是缓存穿透？
2. LRU缓存怎么设计？
3. 为什么会有MRU置换？
4. 布隆过滤器在什么场景下使用？
5. Redis的5种基本数据类型

01-什么是缓存?

缓存：加速数据访问的存储，降低延迟(latency)，提升吞吐量(Throughput)的利器。

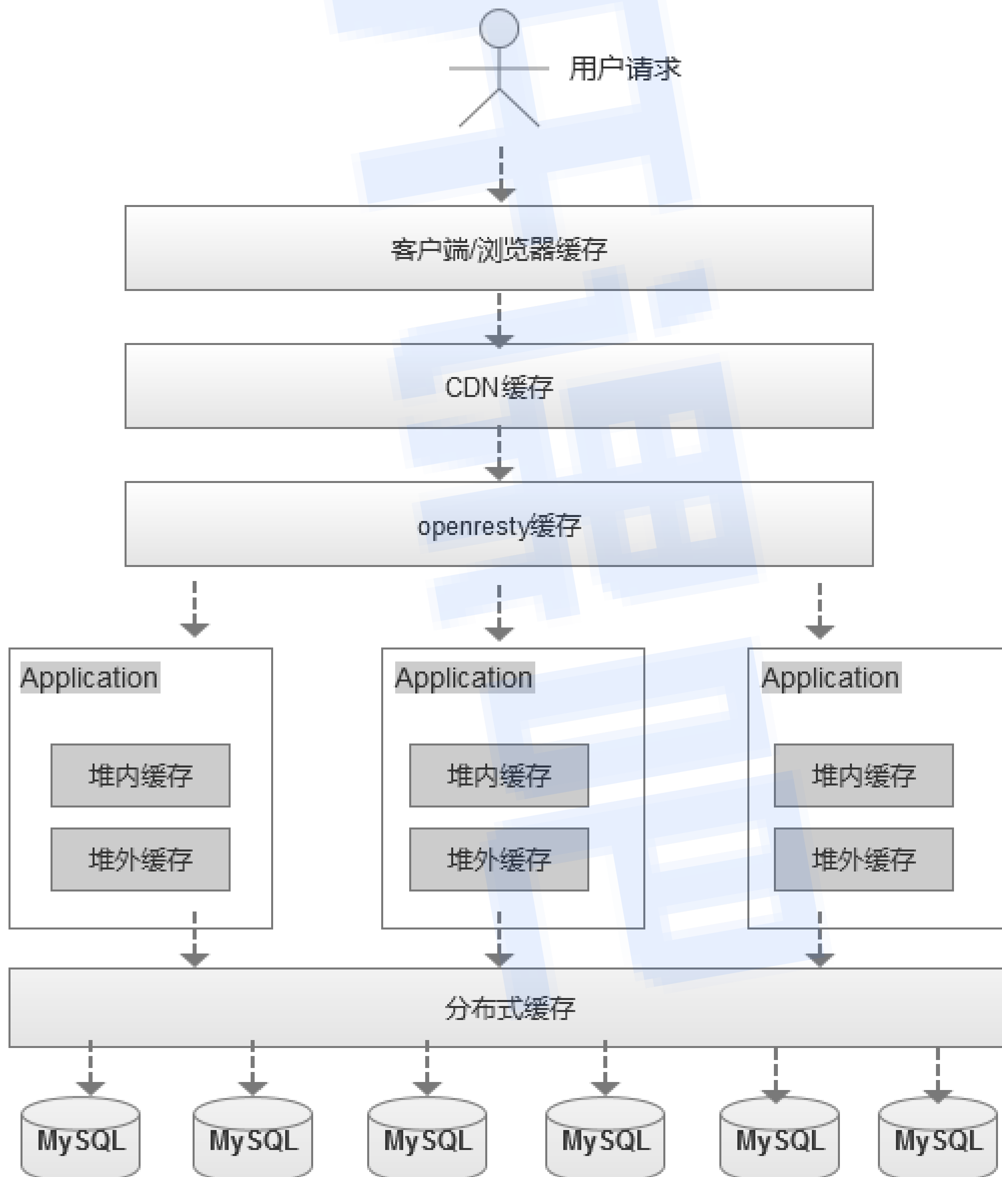
1.1 缓存分类

1. 查库
2. ConcurrentHashMap
3. LRU
4. Guava Cache
5. 分布式缓存(redis，MemCache)
6. 多级缓存

1.2 在什么地方加缓存?

缓存对于每个开发者来说是相当熟悉了，为了提高程序的性能我们会去加缓存，但是在什么地方加缓存，如何加缓存呢?

举个栗子：假设一个网站，需要提高性能，缓存可以放在浏览器，可以放在反向代理服务器，还可以放在应用程序进程内，同时可以放在分布式缓存系统中。



从用户请求数据到数据返回，数据经过了浏览器，CDN，代理服务器，应用服务器，以及数据库各个环节。每个环节都可以运用缓存技术。从浏览器/客户端开始请求数据，通过 HTTP 配合 CDN 获取数据的变更情况，到达代理服务器（Nginx）可以通过反向代理获取静态资源。再往下来到应用服务器可以通过进程内（堆内）缓存，分布式缓存等递进的方式获取数据。如果以上所有缓存都没有命中数据，才会回源到数据库。

缓存的顺序：用户请求 → HTTP 缓存 → CDN 缓存 → 代理服务器缓存 → 进程内缓存 → 分布式缓存 → 数据库。

- **距离用户越近**，缓存能够发挥的效果也就越好。

1.3 缓存读写流程

缓存读取：没有读取到数据就读取缓存后的存储

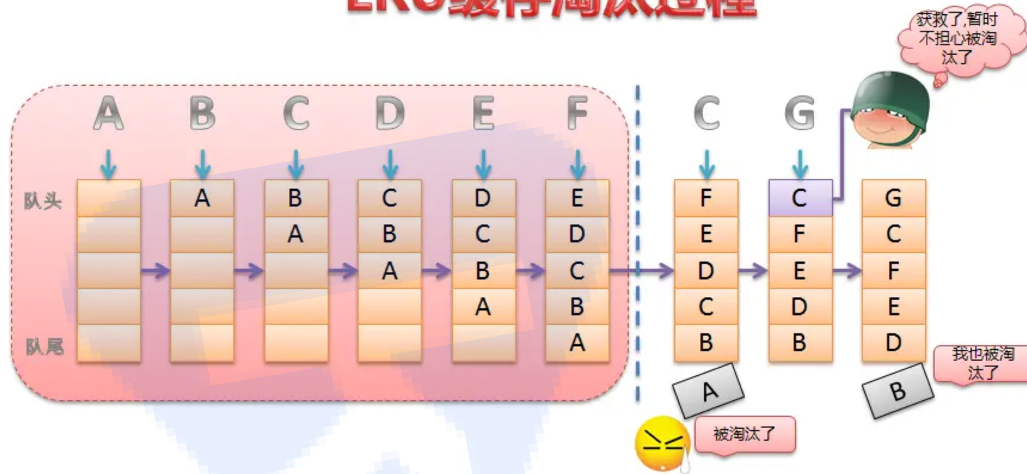
- 关注穿透路径：一级-->二级 --> 三级 --> 内存 --> 磁盘
- 穿透率：
  - 命中缓存(hit)
  - 穿透缓存(Penetration)
  - 穿透率：没有命中/总访问次数
  - 命中率：1-穿透率

缓存写入：什么时候写入?

- write Through：穿透写入
- Write Back：回写
  - 预先写入：预热
  - 延迟写入：定时、事件触发

02-缓存置换策略

- 空间有限：保留热数据，移除冷数据【什么样的数据是冷数据？】**在不同的业务场景，定义也不一样!**
- FIFO:先进先出，先进入缓存的会先被淘汰。
  - 最简单的策略，但是会导致我们命中率很低。
  - 试想一下我们如果有个访问频率很高的数据是所有数据第一个访问的，而那些不是很高的是后面再访问的，那这样就会把我们的首个数据，但是频率很高的给挤出。
- 最近最久未使用(Least Recently Used)简称LRU：
  - 在这种算法中避免了上面的问题，每次访问数据都会将其放在我们的队头，如果需要淘汰数据，就只需要淘汰队尾即可。
  - 潜在问题：淘汰热点数据，如果有个数据在1个小时的前59分钟访问了1万次(热点数据)，再后一分钟没有访问这个数据，但是有其他的数据访问，就导致了我们这个热点数据被淘汰。



- 最近最少使用(Least Frequently Used) 简称LFU：
  - 利用额外的空间记录每个数据的使用频率，然后选出频率最低进行淘汰。
  - 这样就避免了LRU不能处理时间段的问题。
- 最近最常使用算法（MRU）：这个缓存算法最先移除最近最常使用的条目。一个MRU算法擅长处理一个条目越久，越容易被访问的情况。
- 自适应缓存替换算法(ARC)：在IBM Almaden研究中心开发，这个缓存算法同时跟踪记录LFU和LRU，以及驱逐缓存条目，来获得可用缓存的最佳使用。
- ...

03-LRU(Least Frequently Used)

1. **FIFO:先进先出**
2. **LRU:最近最少使用算法。**
3. **LFU:最近最少频率使用。**

上面列举了三种常见淘汰策略，对于这三种，实现成本是一个比一个高，同样的命中率也是一个比一个好。而我们一般来说选择的方案居中即可，即实现成本不是太高，而命中率也还行的LRU。

如何实现一个LRU呢?

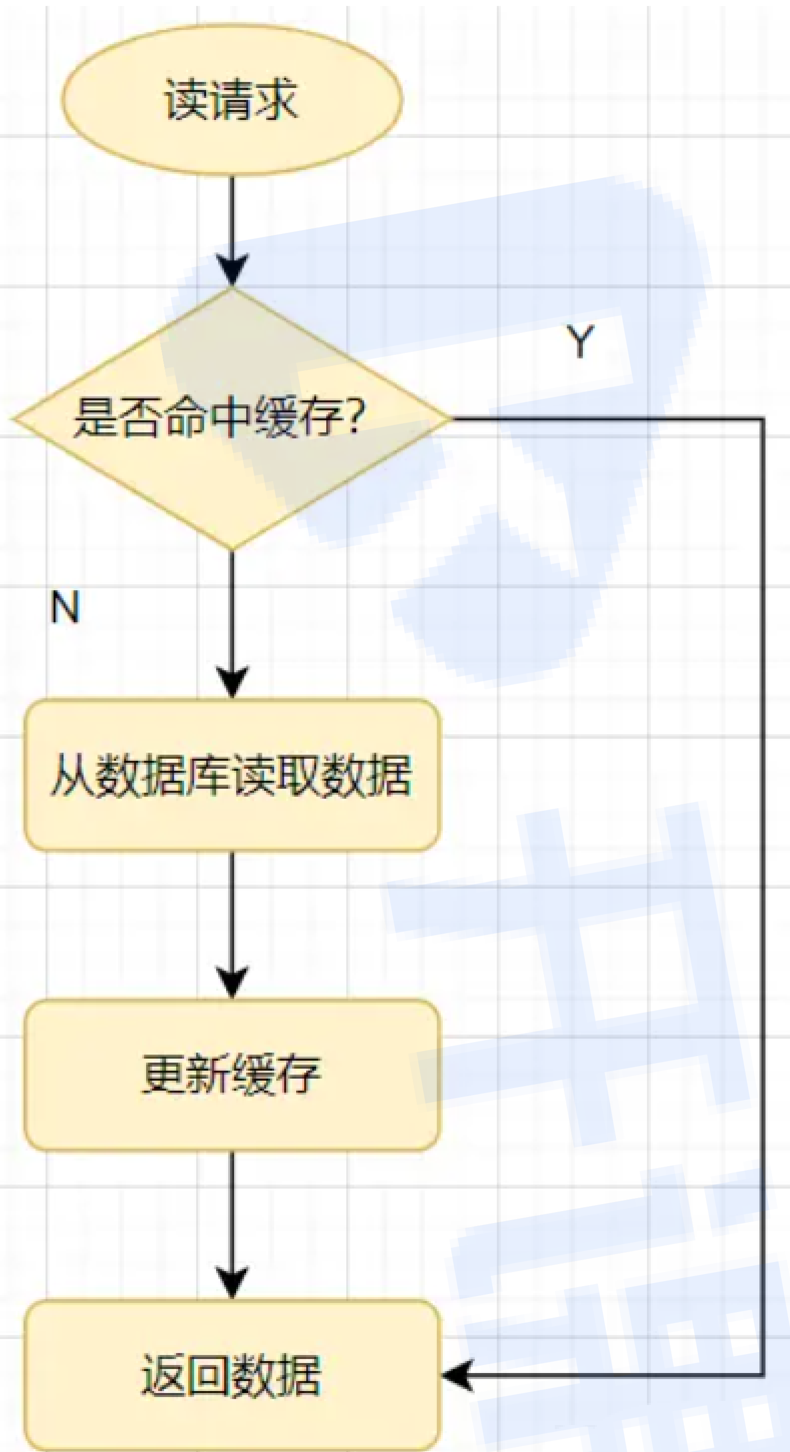
- 我们可以通过继承LinkedHashMap，重写removeEldestEntry方法，即可完成一个简单的LRU。
- 在LinkedHashMap中维护了一个entry(用来放key和value的对象)链表。
- 在每一次get或者put的时候都会把插入的新entry，或查询到的老entry放在我们链表末尾。
- 可以注意到我们在构造方法中，大小特意设置到max*1.4，在下面的removeEldestEntry方法中只需要size>max就淘汰，这样我们这个map永远也走不到扩容的逻辑了，通过重写LinkedHashMap，几个简单的方法我们实现了我们的LRUMap。

```
1  class LRUMap extends LinkedHashMap {
2
3          private final int max;
4          private Object lock;
5
6          public LRUMap(int max, Object lock) {
7              //重点1：无需扩容
8              super((int) (max * 1.4f), 0.75f, true);
```

```java
            this.max = max;
            this.lock = lock;
        }
        /**
         * 重点2：重写LinkedHashMap的removeEldestEntry方法即可，在Put的时候判断，如果为true，就会删除最老的
         */
        @Override
        protected boolean removeEldestEntry(Map.Entry eldest) {
            return size() > max;
        }
        public Object getValue(Object key) {
            synchronized (lock) {
                return get(key);
            }
        }
        public void putValue(Object key, Object value) {
            synchronized (lock) {
                put(key, value);
            }
        }
        public boolean removeValue(Object key) {
            synchronized (lock) {
                return remove(key) != null;
            }
        }
        public boolean removeAll(){
            clear();
            return true;
        }
    }

```

04-什么是缓存穿透?

一个常见的缓存使用方式：读请求来了，先查下缓存，缓存有值命中，就直接返回；缓存没命中，就去查数据库，然后把数据库的值更新到缓存，再返回。



缓存穿透：指查询一个一定不存在的数据，由于缓存是不命中的，需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，进而给数据库带来压力。

4.1 大量数据判断是否存在问题

看起来通过一个中间层就可以解决问题！是不是用**HashMap**就好了呢?

一般场景足以解决问题，而且还简单，HashMap时间复杂度可以达到O(1)。但是因为HashMap数据是在内存里面的，如果大量的数据远超出了服务器的内存，就无法使用HashMap。

这个时候就轮到**布隆过滤器**来做这个事情了。

05-布隆过滤器使用场景

布隆过滤器主要是在redis中间的比较多，因此像这种数据结构类的主要是考原理以及使用场景。

5.1 使用场景

1. 钓鱼网站
2. 垃圾邮件检测
3. **黄色网站筛查**【24小时定时生成，1个月】
4. 解决缓存穿透：通常使用布隆过滤器去解决redis中的缓存穿透问题

先举一个例子，在我们身边充斥着各种各样的XXPorn网站，为了不毒害我们祖国的花朵，于是国家网警就开始对这些网站进行割除过滤，问题来了，这些网站的地址其实是不停的更换的，这些垃圾网站和正常网站加起来，全世界据统计也有几十亿个。因此就会带来如下的问题：

- **（1）网站数量太多，存储起来比较麻烦。一个地址最起码有32个字节，一亿个地址就需要1.6G的内存。**
- **（2）一个一个比较太费时间了。**

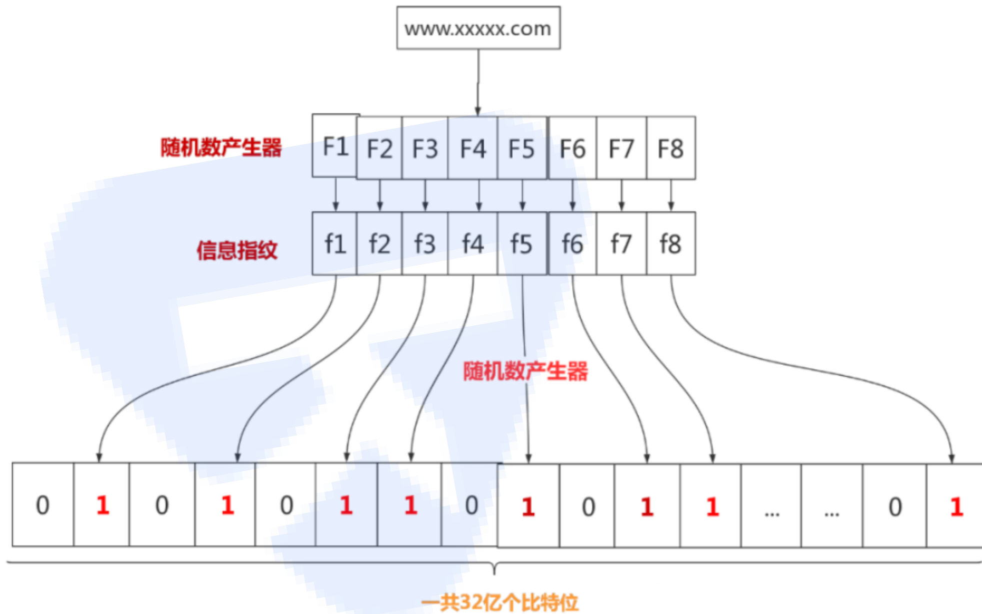
因此布隆过滤器的高效强大就显现出来了，他是如何做到高效的呢？

本质上其实就是一个**HASH映射器。**他的底层其实是一个超大的二进制向量和一系列随机映射函数。现在我们按照之前的那个例子，我们存储1亿个垃圾网站地址。

- 第一步：建立一个32亿二进制（比特）【204mb】，也就是4亿字节的向量。全部置0

- 第二步：网警用八个不同的随机数产生器（F1,F2, ...,F8）产生八个信息指纹（f1, f2, ..., f8）。
- 第三步：用一个随机数产生器 G 把这八个信息指纹映射到 1 到32亿中的八个自然数 g1, g2, ...,g8。

- 第四步：把这八个位置的二进制全部设置为一。



OK，有一天网警查到了一个可疑的网站，想判断一下是否是XX网站，于是就开始检查了。通过同样的方法将XX网站通过哈希映射到32亿个比特位数组上的8个点。如果8个点的其中有一个点不为1，则可以判断该元素一定不存在集合中。

> **注意：如果两个XX网站通过上面的步骤映射到了相同的8个点上，或者是有一部分点是重合的，这时候该怎么办？于是就出现了误报，也就是说A网站在12345678个点上全部置1，B网站通过同样的方式在23456789上全部置1，这时候B网站来了是不能确定是否包含的。这个逻辑相信各位都理解。这个是最基础的面试问题。**

5.2 布隆过滤器是个啥?

布隆过滤器其实就是加快判定一个元素是否在集合中出现的方法。

> **考点：布隆过滤器只能判定一个元素不在集合里面，不能判断存在！什么意思呢，就是说一个苹果不在篮子里，这个我可以通过布隆过滤器知道，但是一定在篮子里嘛？这个通过布隆过滤器我是不能判定的。**

1、怎么理解布隆过滤器?

布隆过滤器是一种占用空间很小的数据结构，它由一个很长的二进制向量和一组Hash映射函数组成，它用于检索一个元素是否在一个集合中，**空间效率和查询时间都比一般的算法要好的多**，缺点是有一定的**误识别率**和删除困难。

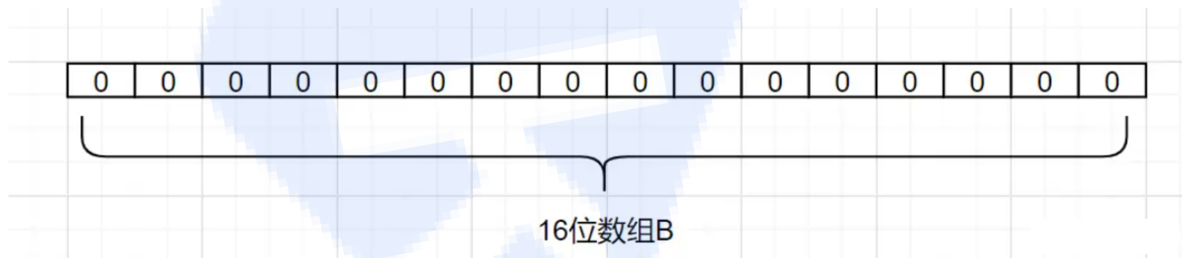
- 增加和查询元素的时间复杂度为:O(K), (K为哈希函数的个数，一般比较小)，与数据量大小无关 。

2、布隆过滤器原理

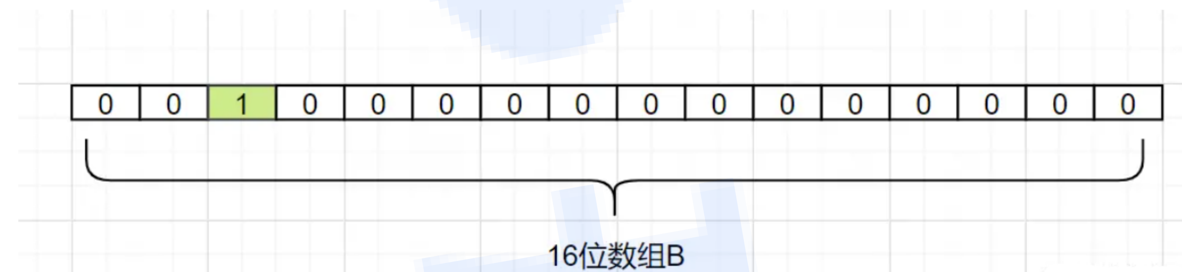
> 二进制向量【数组、集合】【BitSet类】100万位
>
> 映射目标数据add【多次hash映射到二进制向量的集合中】
>
> 判断是否重复方法【多次hash的结果判断在Set集合中是否存在，如何又一次不存在！也就意味着不存在】

假设我们有个集合A，A中有n个元素。利用**k个哈希散列**函数，将A中的每个元素**映射**到一个长度为a位的数组B中的不同位置上，这些位置上的二进制数均设置为1。如果待检查的元素，经过这k个哈希散列函数的映射后，发现其k个位置上的二进制数**全部为1**，这个元素很可能属于集合A，反之，**一定不属于集合A**。

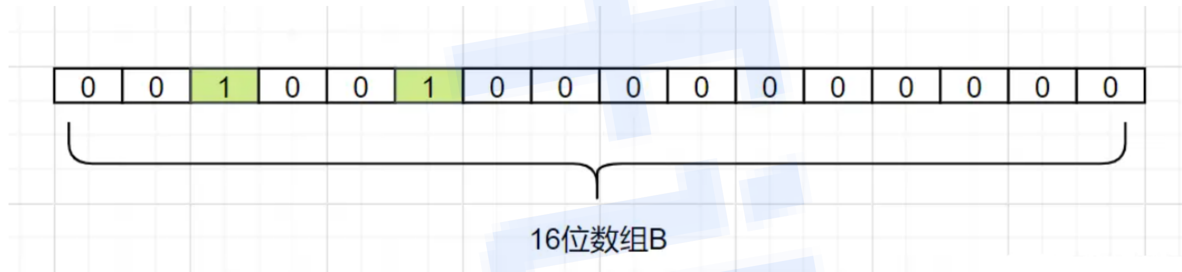
来看个简单例子吧，假设集合A有3个元素，分别为{**d1,d2,d3**}。有1个哈希函数，为**Hash1**。现在将A的每个元素映射到长度为16位数组B。



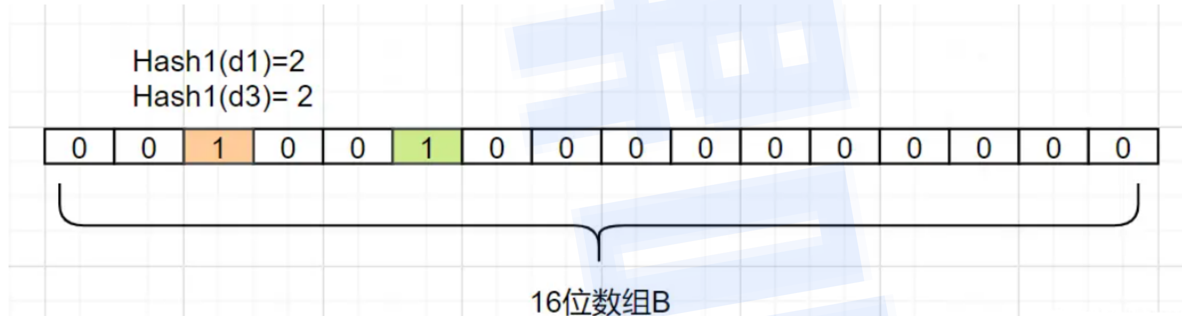
我们现在把d1映射过来，假设Hash1（d1）= 2，我们就把数组B中，下标为2的格子改成1，如下：



我们现在把**d2**也映射过来，假设Hash1（d2）= 5，我们把数组B中，下标为5的格子也改成1，如下：



接着我们把**d3**也映射过来，假设Hash1（d3）也等于 2，它也是把下标为2的格子标1：



因此，我们要确认一个元素dn是否在集合A里，我们只要算出Hash1（dn）得到的索引下标，只要是0，那就表示这个元素不在集合A，如果索引下标是1呢？那该元素可能是A中的某一个元素。因为你看，d1和d3得到的下标值，都可能是1，还可能是其他别的数映射的，布隆过滤器是存在这个缺点的：会存在hash碰撞导致的假阳性，判断存在误差。

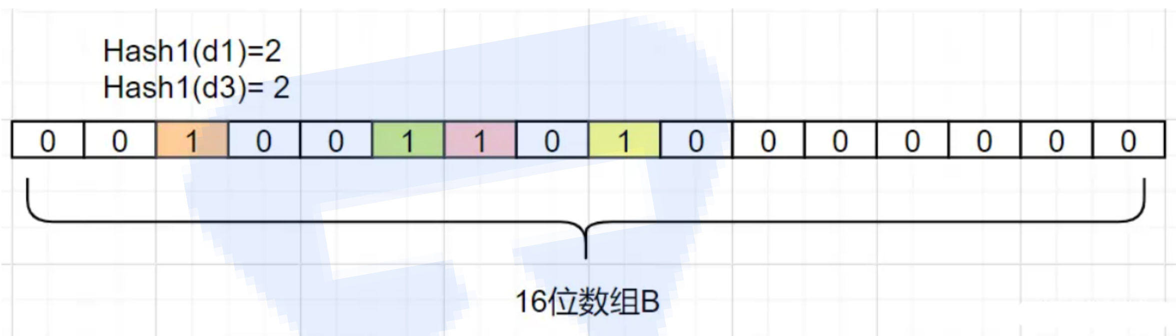
如何减少这种误差(假阳性)呢?

用多少存储空间来进行判断重复合适呢?

- 搞多几个哈希函数映射，降低哈希碰撞的概率

- 同时增加B数组的bit长度，可以增大hash函数生成的数据的范围，也可以降低哈希碰撞的概率

我们又增加一个Hash2**哈希映射**函数，假设Hash2（d1）=6,Hash2（d3）=8,它俩不就不冲突了嘛，如下：



即使**存在误差**，我们可以发现，布隆过滤器并**没有存放完整的数据**，它只是运用一系列哈希映射函数计算出位置，然后填充二进制向量。如果**数量很大的话**，布隆过滤器通过**极少的错误率，换取了存储空间的极大节省**，还是挺划算的。

开源实现：

- **Google的Guava类库**
- Twitter的 Algebird 类库

3、关于误报率

通过上面的解释相信都大概了解的差不多了，其实就是hash函数映射，由于有hash冲突产生了误报率，误报率也就是判断失败的情况。

既然是由于hash冲突，那我把布隆过滤器的二进制向量调到很大，这样不就解决了嘛，但是由于数据量比较大，因此现在就要考虑一下误报率和存储效率之间选择一个折中值了。有一个计算公式如下：

假设位数组的长度为m，哈希函数的个数为k。检测某一元素是否在该集合中的误报率是：

$$(1-[1-\frac{1}{m}]^{kn})^k \approx (1-e^{-\frac{kn}{m}})^k$$

如何使得**误报率**最小，数学问题，求导就可以了。

5.3 代码实现布隆过滤器

上面只是给出了其原理，下面我们代码实现一下。

```
1  /**
2   * 自定义布隆过滤器
3   */
4  public class MyBloomFilter {
5
6      // 2 << 25 约等于100万个比特位
7      private static final int DEFAULT_SIZE = 2 << 25;
8      private static final int[] seeds = new int[]{3, 5, 7, 11, 13, 19, 23,
   37};
9      //这么大存储在BitSet
10     private BitSet bits = new BitSet(DEFAULT_SIZE);
11     private SimpleHash[] func = new SimpleHash[seeds.length];
12
13     public static void main(String[] args) {
```

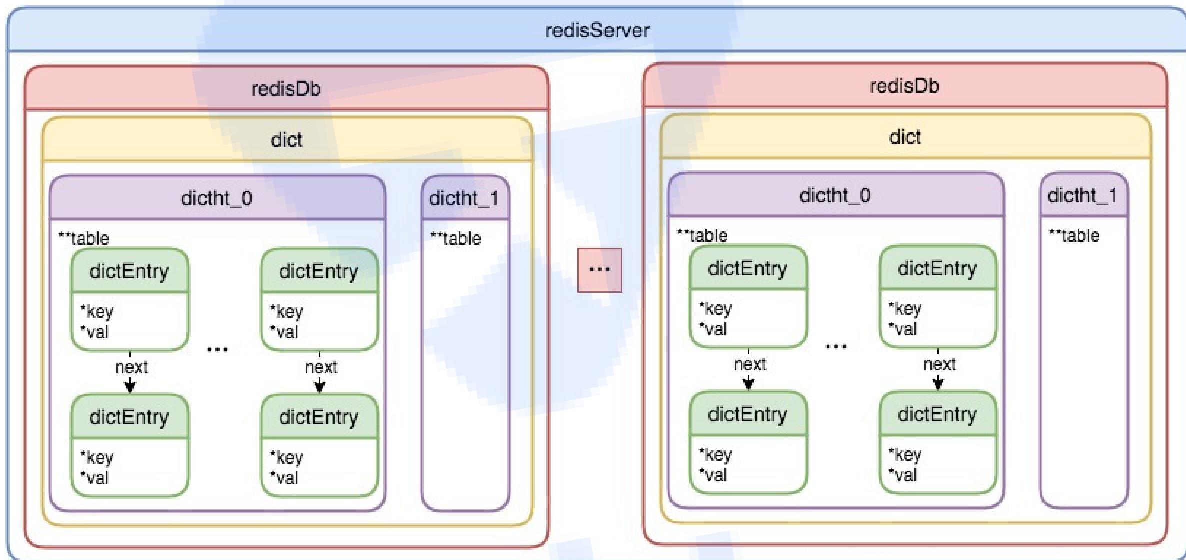
```java
        //可疑网站
        String value = "www.kkb.com";
        MyBloomFilter filter = new MyBloomFilter();
        //加入之前判断一下
        System.out.println(filter.contains(value));
        filter.add(value);
        //加入之后判断一下
        System.out.println(filter.contains(value));
    }

    //构造函数
    public MyBloomFilter() {
        for (int i = 0; i < seeds.length; i++) {
            func[i] = new SimpleHash(DEFAULT_SIZE, seeds[i]);
        }
    }

    //添加网站
    public void add(String value) {
        for (SimpleHash f : func) {
            bits.set(f.hash(value), true);
        }
    }

    //判断可疑网站是否存在
    public boolean contains(String value) {
        if (value == null) {
            return false;
        }
        boolean ret = true;
        for (SimpleHash f : func) {
            //核心就是通过“与”的操作
            ret = ret && bits.get(f.hash(value));
        }
        return ret;
    }
}
```

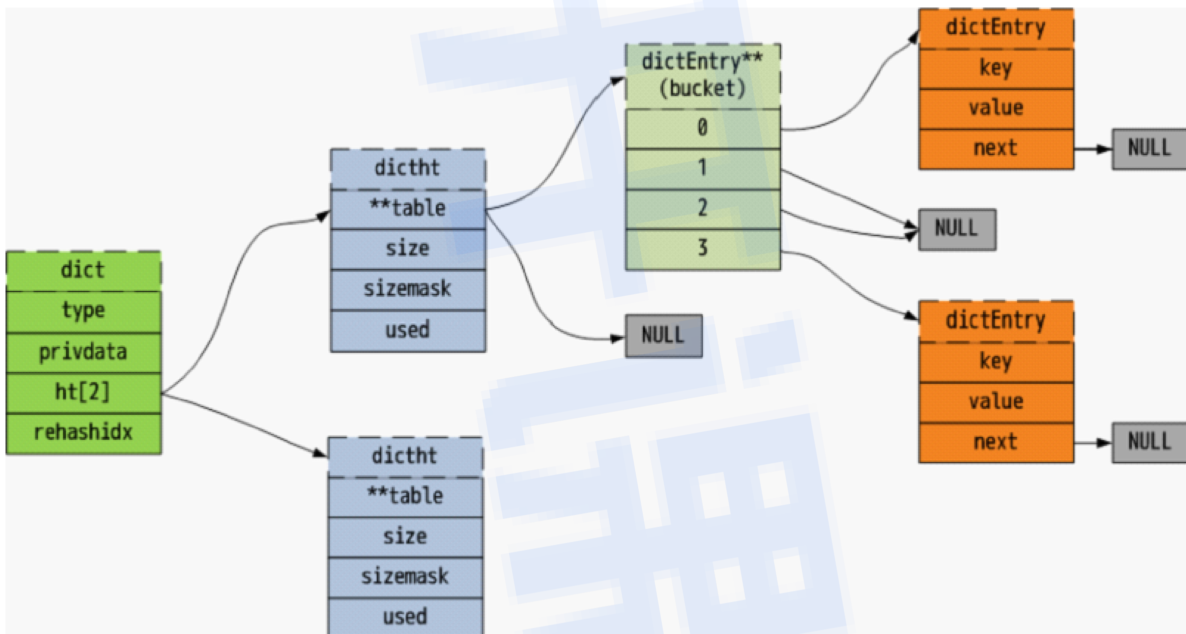
还有一个SimpleHash，我们看一下

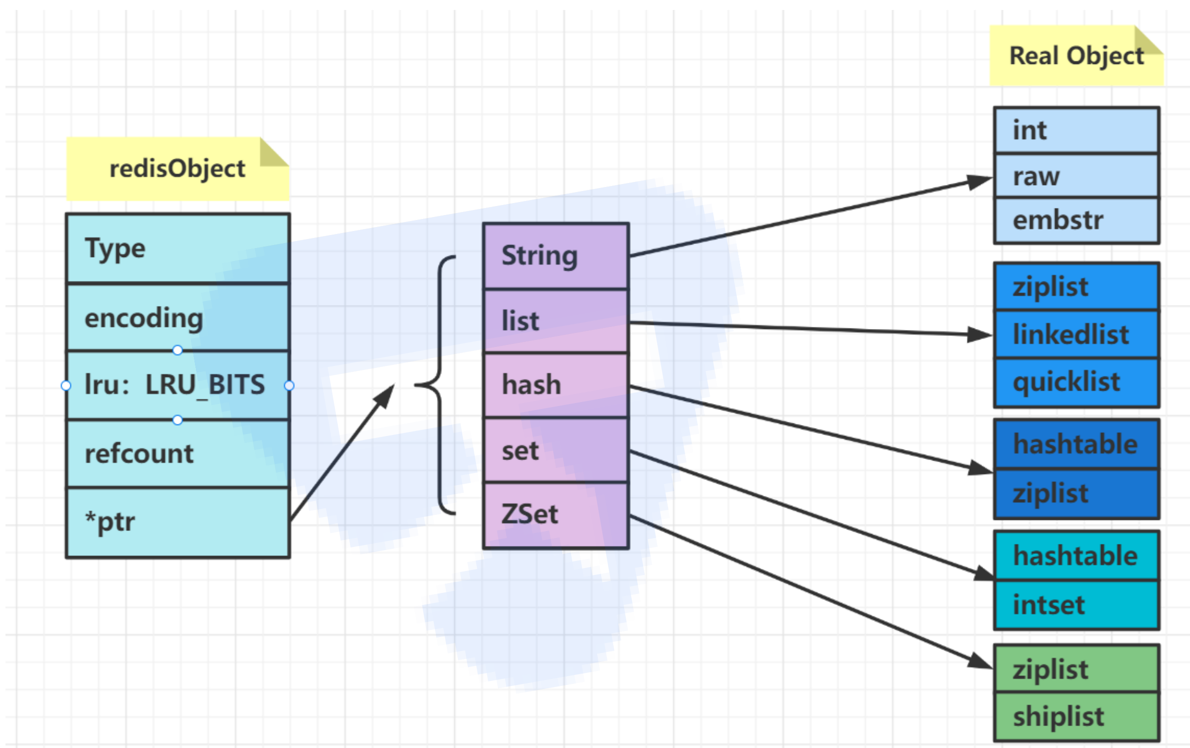
```java
/**
 * 简单的哈希函数实现
 */
public class SimpleHash {
    private int cap;
    private int seed;

    public SimpleHash(int cap, int seed) {
        this.cap = cap;
        this.seed = seed;
    }

    public int hash(String value) {
        int result = 0;
        int len = value.length();
        for (int i = 0; i < len; i++) {
            result = seed * result + value.charAt(i);
```

```
18            }
19            return (cap - 1) & result;
20        }
21    }
```

06-Redis的5种基本数据类型的底层结构







Redis支持5种结构，而每种结构都有至少两种具体实现；

这样做的好处在于：解耦合("面向接口编程")。当需要增加或改变内部实现时，使用者不受影响；根据不同的应用场景自动切换内部编码，提高效率，榨取最后一滴性能。

6.1 String类型

- int：8个字节，字符串是整型时，这个值使用long整型表示。
- 在redis3.2版本之前：
  - embstr：字符串<=39
  - raw：字符串>39
- 在redis3.2版本之后：
  - embstr：字符串<=44
  - raw：字符串>44
- raw和embstr的主要区别embstr只分配一次内存空间，而raw需要分配两次空间
  - embstr优点：与raw相比，embstr创建时少分配一次空间，删除时少释放一次空间，以及对象的所有数据连在一起，寻找方便。
  - embstr缺点：如果字符串的长度增加需要重新分配内存时，整个redisObject和sds都需要重新分配空间
- 字符串长度不能超过512MB

6.2 List类型：

- 存储多个**有序**字符串
- 一个列表可以存储2^64-1个元素。
- 支持双端插入和弹出，可以获取指定位置元素，可以作为数组、队列和栈使用
- 在redis3.0版本之前：ziplist压缩列表【数组，查询快，增删慢】、linkedlist双向链表【双向链表，查询慢，增删快】
- 在redis3.0版本之后：
  - ziplist、linkedlist
  - quicklist快速列表：是ziplist和linkedlist的结合

6.3 Hash类型

- 对于Hash底层实现，我们做不了主，是由Redis自动适应！
- ziplist：hash中元素数量少于512个 且 键值对字符串长度小于64字节
  - 与哈希表相比，压缩列表用于元素个数少、元素长度小的场景
  - 优势：集中存储，节省空间
  - 劣势：元素的操作复杂度也由O(1)变为了O(n)。
    - 但由于哈希中元素数量较少，因此操作的时间并没有明显劣势
- HashTable：hash中元素数量大于512个 或 键值对字符串长度大于64字节

6.4 Set类型：

- 存储多个**无序**不重复字符串
- 一个集合中最多可以存储2^64-1个元素；
- 除了支持常规的增删改查，Redis还支持多个集合取交集、并集、差集。不支持索引操作
- intset：
  - 集合所有元素都是整数类型
- hashtable：
  - 集合元素中有任意一个字符串类型
  - 集合元素数量大于512时

6.5 ZSet类型：

- 有序集合zset与set集合一样，元素都不能重复。但与Set不同的是，有序集合中的元素是有顺序的。与列表使用索引下标作为排序依据不同，有序集合为每个元素设置一个分数（score）作为排序依据。
- ziplist：集合元素数量小于128，且所有元素的长度小于64字节
- skiplist：集合元素数量大于128，或元素长度大于64字节。
  - 跳表的时间复杂度是O(logN)
  - 支持顺序操作