## 02 架构面试： 给你一个业务场景，给我画一下架构图?

讲义地址： https://mp.weixin.qq.com/s/yZ0l6KVomDh3A5K-fm8wag

**架构图1：业务场景视图/业务架构图**

用例分析图

**架构图2：系统技术架构图/宏观技术架构图**

**架构图3：系统外部依赖架构(System Context Diagram)**

**架构图4：子系统依赖分析(Container Diagram)/系统内部依赖架构**

**架构图5：组件架构图(Component Diagram)**

**架构图6：模块架构图**

**架构图7：逻辑架构视图/ 核心功能流程与数据流程**

uml 三剑客

**架构图8：部署架构分析**

## 03 架构面试： 你们系统qps多少？怎么部署的?

讲义地址：https://mp.weixin.qq.com/s/k1j5xy3dRTaz65aH1NdNXQ

- 一、什么是QPS?
- 二、什么是TPS?
- 三、QPS和TPS的关系
- 四、什么是RT，响应时间
- 五、什么是并发数?
- 六、吐吞量
- 七、什么是PV、UV、DAU、MAU
  - PV
  - UV
  - DAU
  - MAU
- 八、最佳线程数量
- 九、统吞吐量评估
- 十、每天有几千万请求，你的系统如何部署?

## 04 架构面试：百亿级存储，怎么设计？只是分库分表？(字节面试真题)

讲义地址：https://mp.weixin.qq.com/s/xGYM0pXAHfaLMpTxBJvLBg

**百亿级数据存储架构，只有分库分表吗?**

- 从0到1, 百亿级数据存储架构，怎么设计？

**- 百亿级数据存储架构，怎么维护多副本之间的数据一致?**

- 方案一：同步双写

**- 方案2：异步双写**

- 方案2.1 使用内存队列（如阻塞队列）异步

- 方案2.2 使用消息队列（如阻塞队列）异步

**- 方案三：定期同步**

- 方案四：数据订阅

**- 方案五：冗余表的同步双写**

-冗余表的同步双写实现方案

-冗余表的异步双写实现方案：

**- 方案六：ETL数据同步**

从0到1, 百亿级数据存储架构，怎么设计？

# 05: 如何设计一个高可用的软件架构？请说明设计思路

讲义链接 https://mp.weixin.qq.com/s/tNOhR16aC-cr5EqKiwbAOQ

## 1. 第一个要点：总体战略思想

1）question： 纵观全局，高可用的目标是什么?

2）question： 有高可用，就有不可用，不可用有哪些分类呢?

3）question： 架构总体考虑，有什么策略?

4）question： 架构如何分层的？每层叫什么?

## 2. 第二个要点：细分来看，接入层，应用层高可用方案

1）question：接入层的策略是什么？有哪些高可用方案?

2）question：应用层/服务层HA架构的主要原则包括哪些？有哪些主要策略?

## 3. 第三个要点：服务层高可用方案

1）question：如何划分微服务？如何进行解耦?

2）question：服务层冗余设计原则是什么?

3）question：各级服务的部署原则是什么?

4）question：各级服务HA上线发布原则?

5）question：服务的三可原则落地?

## 4. 第四个要点：数据层和基础设施层高可用方案

1）question：数据层高可用方案包含哪些?

2）question：这些方案如何实现的?

3）question：基础设施层包含哪些?

4）question：综合措施大的方面有哪些? 每个措施包含哪些具体方案

## 5. 问题总结

总览全局-架构分层-分而治之-具体方案

# 06 得物面试：10wqps高并发，如何防止重复下单?

讲义地址： https://mp.weixin.qq.com/s/QZ4z_I5Gl6FNz6jcyGkknA

## 第一个要点： 电商订单支付核心流程分析

（1）问题1： 订单支付的核心业务流程和交互流程包括哪些?

（2）问题2： 订单支付状态的变化有哪些?

## 第二个要点： 重复下单的定义、危害以及应对策略

（1）问题1： 什么是重复下单?

（2）问题2： 重复下单会带来哪些问题?

（3）问题3： 什么场景下会发生重复下单?

## 第三个要点： 重复下单问题及幂等性问题

（1）问题1： 什么是幂等性问题?

（2）问题2： 如何解决接口幂等性问题?

### 第四个要点： 如何解决重复下单问题

（1）方案一：提交订单按钮置灰

（2）方案二：请求唯一ID+数据库唯一索引约束

（3）方案三：reids分布式锁+请求唯一ID

（4）方案四：reids分布式锁+token

（5）方案五：技术+产品+运营支持

### 第五个要点： 实操 - reids分布式锁+token 解决重复下单的问题

（1）step1：使用AOP进行 BizToken 的无入侵生成

（2）step2：编写服务验证逻辑，通过 aop 代理方式实现

（2）step3：使用redission分布式锁保证幂等

### 第六个要点：防止重复下单总结

## 07 网易面试：JDK1.8为什么将HashMap 头插法 改 尾插法？

讲义地址： https://mp.weixin.qq.com/s/BG3Z5Y-NC8JBttGSAa8YBw

第一个要点： hash表的结构和问题

（1）问题1： 什么是哈希表

（3）问题2： 什么是hash冲突（/hash碰撞）

（3）问题3： 采用链地址法，解决hash碰撞，会出现什么样的极端情况?

第二个要点： JDK1.7 中HashMap头插法的核心流程， 优点和缺点

（1）问题1： 头插法的核心流程?

（2）问题2： 头插法的优点?

（3）问题3： 头插法的缺点?

（4）问题4： JDK1.7头插法导致的在扩容场景导致恶性死循环的根因？如何破解?

第三个要点：JDK1.8中HashMap的底层数据结构

（1）问题1： JDK1.8的HashMap是怎么扩容和迁移的?

（2）问题2： 什么是尾插法的优点?

（3）问题3： 什么是尾插法的缺点?

第四个要点：头插法和尾插法的性能对比

（1）问题1： 头插法理论O(1)，为什么 JDK1.8反而更快

第五个要点：尾插法和头插法总结

## 08:网易一面：如何设计线程池？请手写一个简单线程池?

### 说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如极兔、有赞、希音、百度、网易的面试资格，遇到一几个很重要的面试题：

> 如何设计线程池?

与之类似的、其他小伙伴遇到过的问题还有：

> 请手写一个简单线程池?

这个问题，很多小伙伴在社群里边反馈，都遇到过


**线程池的知识，既是面试的核心知识，又是开发的核心知识。**

所以，这里尼恩给大家做一下系统化、体系化的线程池梳理，使得大家可以充分展示一下大家雄厚的“技术肌肉”，**让面试官爱到“不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的《尼恩Java面试宝典》V62版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从公众号【技术自由圈】获取。

## 为什么要使用线程池?

多线程编程是在开发过程中非常基础且非常重要的一个环节，基本上任何一家软件公司或者项目中都会使用多线程。主要有三个原因：

1. 降低资源的消耗。降低线程创建和销毁的资源消耗。
2. 提高响应速度：线程的创建时间为T1，执行时间T2，销毁时间T3，免去T1和T3的时间
3. 提高线程的可管理性

总之：线程池是一种常用的并发编程工具，它可以帮助我们更好地管理和复用线程资源，提高程序的性能和稳定性。

线程池也是 3高架构的基础技术。

## JAVA中的线程池组件

在Java中，我们可以使用java.util.concurrent包中提供的ThreadPoolExecutor类来创建和使用线程池。

ThreadPoolExecutor 是非常高频，非常常用的组件。

对于 ThreadPoolExecutor 的底层原理和源码，大家要做到非常深入的掌握。大家一定要深入看ThreadPoolExecutor线程池源码，了解其执行过程。

另外，看懂线程池执行流程和源码设计有助于提升我们多线程编程技术和解决工作中遇到的问题。

## 手写线程池的重要性

很多小伙伴给尼恩反馈， 说线程池的**源码太难，看不懂。**

怎么办呢?

大家可以先易后难。

可以手撸一个简单版的线程池加强一下对执行流程的理解。然后再深入源码去历险记。

或者说，如果我们想要更好地理解线程池的工作原理，那么自己动手实现一个简单的线程池是一个很好的选择。

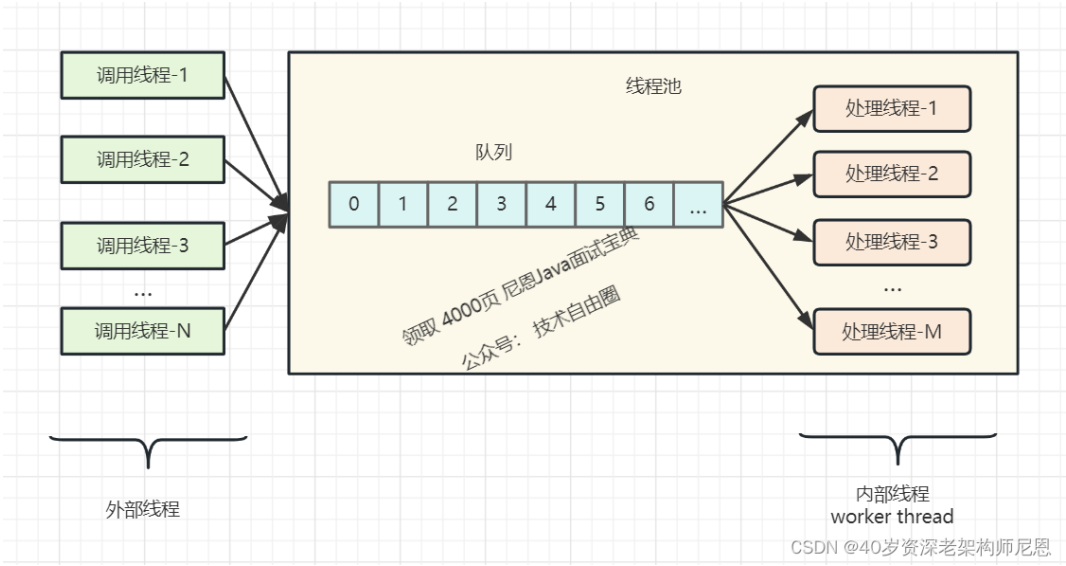
接下来，我将带领大家一步一步地实现一个简单的线程池。

我们将从最基本的功能开始，逐步添加更多的功能和优化，最终实现一个完整的线程池。

## 线程池的实现原理

线程池是一个典型的生产者-消费者模型。下图所示为线程池的实现原理：

- 调用方不断向线程池中提交任务；（生产者）
- 线程池中有一组线程，不断地从队列中取任务，（消费者）
- 线程池管理一个任务队列，对 异步任务进行缓冲 （缓冲区）




**要实现一个线程池，有几个问题需要考虑：**

1. 队列设置多长？

   如果是无界的，调用方不断往队列中方任务，可能导致内存耗尽。如果是有界的，当队列满了之后，调用方如何处理？
2. 线程池中的线程个数是固定的，还是动态变化的？
3. 每次提交新任务，是放入队列？还是开新线程
4. 当没有任务的时候，线程是睡眠一小段时间？还是进入阻塞？如果进入阻塞，如何唤醒？

**针对问题4，有3种做法：**

1. 不使用阻塞队列，只使用一般的线程安全的队列，也无阻塞/唤醒机制。

   当队列为空时，线程池中的线程只能睡眠一会儿，然后醒来去看队列中有没有新任务到来，如此不断轮询。
2. 不使用阻塞队列，但在队列外部，线程池内部实现了阻塞/唤醒机制
3. 使用阻塞队列

很显然，做法3最完善，既避免了线程池内部自己实现阻塞/唤醒机制的麻烦，也避免了做法1的睡眠/轮询带来的资源消耗和延迟。

现在来带大家手写一个简单的线程池，让大家更加理解线程池的工作原理

## 手写一个简单线程池

### 第一步：定义线程池接口

首先，我们需要定义一个线程池接口，用来表示线程池应该具备哪些功能。

一个简单的线程池应该至少具备以下几个功能：

- 添加任务并执行
- 关闭线程池
- 强制关闭线程池

因此，我们可以定义一个ThreadPool接口，它包含三个方法：execute、shutdown和shutdownNow。

```
import java.util.List;

// 线程池接口
public interface ThreadPool {
```

```
    // 提交任务到线程池
    void execute(Runnable task);

    // 优雅关闭
    void shutdown();

    //立即关闭
    List<Runnable> shutdownNow();
}
```

其中：

- execute方法用来添加任务并执行；
- shutdown方法用来关闭线程池，它会等待已经提交到线程池中的任务都执行完毕后再关闭；
- shutdownNow方法用来强制关闭线程池，它会立即停止所有正在执行或等待执行的任务，并返回未执行的任务列表。

### 第二步：实现线程的池化管理

接下来，我们需要实现一个简单的线程池类，它实现了ThreadPool接口，并提供了基本的功能。

为了简化代码，先实现一个v1版本，这是一个基础版本，一个简单的实现示例。

在v1版本中，我们先不考虑拒绝策略、自动调节线程资源等高级功能。

下面是一个简单的实现示例：

**首先定义一个工作线程类：**

```
// 定义一个工作线程类
public class WorkerThread extends Thread {
    // 用于从任务队列中取出并执行任务
    private BlockingQueue<Runnable> taskQueue;

    // 构造方法，传入任务队列
    public WorkerThread(BlockingQueue<Runnable> taskQueue) {
        this.taskQueue = taskQueue;
    }

    // 重写run方法
    @Override
    public void run() {
        // 循环执行，直到线程被中断
        while (!Thread.currentThread().isInterrupted() && !taskQueue.isEmpty()) {
            try {
                // 从任务队列中取出一个任务，如果队列为空，则阻塞等待
                Runnable task = taskQueue.take();
                // 执行任务
                task.run();
            } catch (Exception e) {
                e.printStackTrace();
                // 如果线程被中断，则退出循环
                break;
            }
        }
    }
}
```

然后，基于一个线程池接口，实现一个简单的线程池，

```
// 简单的线程池实现
public class SimpleThreadPool implements ThreadPool {
    // 线程池初始化时的线程数量
    private int initialSize;
    // 任务队列
    private BlockingQueue<Runnable> taskQueue;
    // 用于存放和管理工作线程的集合
    private List<WorkerThread> threads;
    // 是否已经被shutdown标志
    private volatile boolean isShutdown = false;

    public SimpleThreadPool(int initialSize) {
        this.initialSize = initialSize;
        taskQueue = new LinkedBlockingQueue<>();
        threads = new ArrayList<>(initialSize);
        // 初始化方法，创建一定数量的工作线程，并启动它们
        for (int i = 0; i < initialSize; i++) {
            WorkerThread workerThread = new WorkerThread(taskQueue);
            workerThread.start();
            threads.add(workerThread);
        }
    }

    // 实现execute方法，用于将任务加入到任务队列，并通知工作线程来执行
    @Override
    public void execute(Runnable task) {
        if (isShutdown) {
            throw new IllegalStateException("ThreadPool is shutdown");
```

```
        }
        taskQueue.offer(task);
    }

    // 关闭线程池，等待所有线程执行完毕
    @Override
    public void shutdown() {
        // 修改状态
        isShutdown = true;
        for (WorkerThread thread : threads) {
            // 中断线程
            thread.interrupt();
        }
    }

    @Override
    public List<Runnable> shutdownNow() {
        // 修改状态
        isShutdown = true;
        // 清空队列
        List<Runnable> remainingTasks = new ArrayList<>();
        taskQueue.drainTo(remainingTasks);

        // 中断所有线程
        for (WorkerThread thread : threads) {
            thread.interrupt();
        }
        // 返回未执行任务集合
        return remainingTasks;
    }
}
```

这个版本的线程池实现了基本的添加任务并执行、关闭线程池和强制关闭线程池等功能。

它在构造方法中接收一个初始化线程池大小参数，用于初始化任务队列和工作线程集合，并创建一定数量的工作线程。

### 第三步：自定义线程池的基本参数

在上一步中，我们实现了一个简单的线程池，它具备了基本的功能。

但是，它存在一个问题：任务队列没有指定容量大小，是个无界队列，其次只指定了初始的线程池大小，应该要提供根据不同的应用场景来调整线程池的大小参数，以提高性能和资源利用率。

因此线程池实现类需要实现自定义初始大小、最大大小以及核心大小的功能。

- 初始大小是指线程池初始化时创建的工作线程数量
- 最大大小是指线程池能够容纳的最多的工作线程数量
- 核心大小是指线程池在没有任务时保持存活的工作线程数量。

这三个参数需要在基本的线程池实现类中定义为成员变量，并在构造方法中传入并赋值。

同时，还需要在execute方法中根据这三个参数来动态地调整工作线程的数量，例如：

- 当活跃的工作线程数量小于核心大小时，尝试创建并启动一个新的工作线程来执行任务；
- 当活跃的工作线程数量大于等于核心大小时，将任务加入到任务队列，等待空闲的工作线程来执行；
- 当任务队列已满时，尝试创建并启动一个新的工作线程来执行任务，
- 当活跃的工作线程数量达到最大大小时，无法再创建新的工作线程。

我们还需要在构造方法里提供一个参数**queueSize**，用于限制队列大小。

下面我们就对类进行改造：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class SimpleThreadPool implements ThreadPool {
    // 线程池初始化时的线程数量
    private int initialSize;
    // 线程池最大线程数量
    private int maxSize;
    // 线程池核心线程数量
    private int coreSize;
    // 队列大小
    private int queueSize;
    // 任务队列
    private BlockingQueue<Runnable> taskQueue;
    // 用于存放和管理工作线程的集合
    private List<WorkerThread> threads;
    // 是否已经被shutdown标志
    private volatile boolean isShutdown = false;

    public SimpleThreadPool(int initialSize, int maxSize, int coreSiz, int queueSizee) {
        // 初始化参数
        this.initialSize = initialSize;
```

```
        this.maxSize = maxSize;
        this.coreSize = coreSize;
        taskQueue = new LinkedBlockingQueue<>(queueSize);
        threads = new ArrayList<>(initialSize);

        // 初始化方法，创建一定数量的工作线程，并启动它们
        for (int i = 0; i < initialSize; i++) {
            WorkerThread workerThread = new WorkerThread();
            workerThread.start(taskQueue);
            threads.add(workerThread);
        }
    }

    @Override
    public void execute(Runnable task) {
        if (isShutdown) {
            throw new IllegalStateException("ThreadPool is shutdown");
        }
        // 当线程数量小于核心线程数时，创建新的线程
        if (threads.size() < coreSize) {
            addWorkerThread(task);
        } else if (!taskQueue.offer(task)) {
            // 当队列已满时，且线程数量小于最大线程数量时，创建新的线程
            if (threads.size() < maxSize) {
                addWorkerThread(task);
            } else {
                throw new IllegalStateException("执行任务失败");
            }
        }
    }


    // 创建新的线程，并执行任务
    private void addWorkerThread(Runnable task) {
        WorkerThread workerThread = new WorkerThread();
        workerThread.start(taskQueue);
        threads.add(workerThread);
        // 任务放入队列
        taskQueue.offer(task);
    }

    ////省略其它代码

}
```

这一步，我们在 **SimpleThreadPool**里新增了**initialSize**，**maxSize**， **coreSize** 三个变量，在构造方法里传入对应三个参数，同时在**execute**方法里，当有任务进入时，先判断当前线程池数量是否满足不同条件，进而执行不同的处理逻辑。

### 第四步：设计饱和拒绝策略

这个功能是为了处理当任务队列已满且无法再创建新的工作线程时，也是就线程池的工作量饱和时，如何处理被拒绝的任务。

不同的场景可能需要不同的拒绝策略，例如

- 直接抛出异常
- 忽略任务
- 阻塞当前线程
- 等等

为了让用户可以自定义拒绝策略，需要

1. 定义一个拒绝策略接口，声明一个方法，用于处理被拒绝的任务。
2. 然后需要在基本的线程池实现类中定义一个拒绝策略成员变量，并在构造方法中传入并赋值。
3. 最后，在execute方法中，在无法创建新的工作线程时，调用拒绝策略来处理该任务。

下面是一个简单的实现示例，

我们首先定义了一个**RejectedExecutionHandler**接口，用来表示拒绝策略。用户可以根据需要实现这个接口，并在构造线程池时传入自己的拒绝策略。

```
public interface RejectedExecutionHandler {
    // 参数：r 代表被拒绝的任务，executor 代表线程池对象
    void rejectedExecution(Runnable r, ThreadPool executor);
}
```

我们再实现一个直接抛出异常的拒绝策略

```
// 直接抛出异常的拒绝策略
public class AbortPolicy implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPool executor) {
        throw new RuntimeException("The thread pool is full and the task queue is full.");
    }
}
```

我们也可以实现一个丢弃策略：

```
// 忽略任务的拒绝策略
public class DiscardPolicy implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPool executor) {
        // do nothing
        System.out.println("Task rejected: " + r);
    }
}
```

接下来，我们再优化**SimpleThreadPool**类，

```
public class SimpleThreadPool implements ThreadPool {
    // 线程池初始化时的线程数量
    private int initialSize;
    // 线程池最大线程数量
    private int maxSize;
    // 线程池核心线程数量
    private int coreSize;
    // 队列大小
    private int queueSize;
    // 任务队列
    private BlockingQueue<Runnable> taskQueue;
    // 用于存放和管理工作线程的集合
    private List<WorkerThread> threads;
    // 是否已经被shutdown标志
    private volatile boolean isShutdown = false;
    // 默认的拒绝策略
    private final static RejectedExecutionHandler DEFAULT_REJECT_HANDLER = new AbortPolicy();
    // 拒绝策略成员变量
    private final RejectedExecutionHandler rejectHandler;


    public SimpleThreadPool(int initialSize, int maxSize, int coreSize, int queueSize) {
        this(initialSize, maxSize, coreSize, queueSize, DEFAULT_REJECT_HANDLER);
    }

    public SimpleThreadPool(int initialSize, int maxSize, int coreSize , int queueSize,
RejectedExecutionHandler rejectHandler) {
        System.out.printf("初始化线程池: initialSize: %d, maxSize: %d, coreSize: %d%n", initialSize, maxSize,
coreSize);
        // 初始化参数
        this.initialSize = initialSize;
        this.maxSize = maxSize;
        this.coreSize = coreSize;
        taskQueue = new LinkedBlockingQueue<>(queueSize);
        threads = new ArrayList<>(initialSize);
        this.rejectHandler = rejectHandler;

        // 初始化方法，创建一定数量的工作线程，并启动它们
        for (int i = 0; i < initialSize; i++) {
            WorkerThread workerThread = new WorkerThread(taskQueue);
            workerThread.start();
            threads.add(workerThread);
        }
    }

    @Override
    public void execute(Runnable task) {
        System.out.printf("添加任务: %s%n", task.toString());
        if (isShutdown) {
            throw new IllegalStateException("ThreadPool is shutdown");
        }
        // 当线程数量小于核心线程数时，创建新的线程
        if (threads.size() < coreSize) {
            addWorkerThread(task);
            System.out.printf("创建新的线程: thread count: %d, number of queues: %d%n", threads.size(),
taskQueue.size());
        } else if (!taskQueue.offer(task)) {
            // 当队列已满时，且线程数量小于最大线程数量时，创建新的线程
            if (threads.size() < maxSize) {
                addWorkerThread(task);
                System.out.printf("创建新的线程: thread count: %d, number of queues: %d%n", threads.size(),
taskQueue.size());
            } else {
                //使用拒绝策略
                rejectHandler.rejectedExecution(task, this);
            }
        }
    }
    // 省略其它代码
}
```

这个版本的线程池在构造方法中新增了一个handler参数，用来表示拒绝策略。当任务队列已满时，它会调用handler的rejectedExecution方法来处理被拒绝的任务。

### 第五步：性能优化之：自动调节线程资源

到目前为止，我们已经实现了一个简单但功能完备的线程池。

**但是，它还有很多可以优化和扩展的地方。**

例如，可以添加自动调节线程资源的功能，为啥需要 自动调节 线程资源呢?

**因为线程资源，是非常昂贵的。**

自动调节 线程资源功能是为了让线程池可以根据任务的变化，动态地增加或减少工作线程的数量，以提高性能和资源利用率。

为了实现这个功能，需要在基本的线程池实现类中定义一个空闲时长成员变量，并在构造方法中传入并赋值。

空闲时长是指当工作线程没有任务执行时，可以保持存活的时间。如果超过这个时间还没有新的任务，那么工作线程就会自动退出。

同时，还需要在工作线程类中定义一个空闲开始时间成员变量，并在run方法中更新它。

空闲开始时间是指当工作线程从任务队列中取出一个任务后，上一次取出任务的时间。

如果当前时间减去空闲开始时间大于空闲时长，那么工作线程就会自动退出。

ok，那么我们继续改进线程池，

```
public class SimpleThreadPool implements ThreadPool {
    // 省略其它代码
    // 线程空闲时长
    private long keepAliveTime;

    public SimpleThreadPool(int initialSize, int maxSize, int coreSize, int queueSize, long keepAliveTime) {
        this(initialSize, maxSize, coreSize, queueSize, keepAliveTime, DEFAULT_REJECT_HANDLER);
    }

    public SimpleThreadPool(int initialSize, int maxSize, int coreSize, int queueSize, long keepAliveTime,
RejectedExecutionHandler rejectHandler) {
        System.out.printf("初始化线程池: initialSize: %d, maxSize: %d, coreSize: %d%n", initialSize, maxSize,
coreSize);
        // 初始化参数
        this.initialSize = initialSize;
        this.maxSize = maxSize;
        this.coreSize = coreSize;
        taskQueue = new LinkedBlockingQueue<>(queueSize);
        threads = new ArrayList<>(initialSize);
        this.rejectHandler = rejectHandler;
        this.keepAliveTime = keepAliveTime;
        // 初始化方法，创建一定数量的工作线程，并启动它们
        for (int i = 0; i < initialSize; i++) {
            // 传入相关参数到工作线程
            WorkerThread workerThread = new WorkerThread(keepAliveTime, taskQueue, threads);
            workerThread.start();
            threads.add(workerThread);
        }
    }

    @Override
    public void execute(Runnable task) {
        if (isShutdown) {
            throw new IllegalStateException("ThreadPool is shutdown");
        }
        RunnableWrapper wrapper = (RunnableWrapper) task;
        System.out.printf("put task: %s %n" , wrapper.getTaskId());

        // 当线程数量小于核心线程数时，创建新的线程
        if (threads.size() < coreSize) {
            addWorkerThread(task);
            System.out.printf("小于核心线程数，创建新的线程: thread count: %d, queue remainingCapacity : %d%n",
threads.size(), taskQueue.remainingCapacity());
        } else if (!taskQueue.offer(task)) {
            // 当队列已满时，且线程数量小于最大线程数量时，创建新的线程
            if (threads.size() < maxSize) {
                addWorkerThread(task);
                System.out.printf("队列已满，创建新的线程: thread count: %d, queue remainingCapacity : %d%n",
threads.size(), taskQueue.remainingCapacity());
            } else {
                rejectHandler.rejectedExecution(task, this);
            }
        } else {
            System.out.printf("任务放入队列: thread count: %d, queue remainingCapacity : %d%n", threads.size(),
taskQueue.remainingCapacity());
        }
    }


    // 省略其它代码
}
```

然后改造工作线程**WorkerThread**：

```
// 定义一个工作线程类
public class WorkerThread extends Thread {
    private List<WorkerThread> threads;
    // 空闲时长
    private long keepAliveTime;
    // 用于从任务队列中取出并执行任务
    private BlockingQueue<Runnable> taskQueue;
    // 构造方法，传入任务队列
    public WorkerThread(long keepAliveTime, BlockingQueue<Runnable> taskQueue, List<WorkerThread> threads) {
        this.keepAliveTime = keepAliveTime;
        this.taskQueue = taskQueue;
        this.threads = threads;
    }

    // 重写run方法
    @Override
    public void run() {
        long lastActiveTime = System.currentTimeMillis();
        // 循环执行，直到线程被中断
        Runnable task;
        while (!Thread.currentThread().isInterrupted() && !taskQueue.isEmpty()) {
            try {
                // 从任务队列中取出一个任务，如果队列为空，则阻塞等待
                task = taskQueue.poll(keepAliveTime, TimeUnit.MILLISECONDS);
                if (task != null) {
                    task.run();
                    System.out.printf("WorkerThread %d, current task: %s%n", Thread.currentThread().getId(),
task.toString());

                    lastActiveTime = System.currentTimeMillis();
                } else if (System.currentTimeMillis() - lastActiveTime >= keepAliveTime) {
                    // 从线程池中移除
                    threads.remove(this);
                    System.out.printf("WorkerThread %d exit %n", Thread.currentThread().getId());
                    break;
                }
            } catch (Exception e) {
                // 从线程池中移除
                threads.remove(this);
                e.printStackTrace();
                // 如果线程被中断，则退出循环
                break;
            }
        }
    }
}
```

在**WorkerThread**类**run**方法里，采用`taskQueue.poll`方法指定等待时长，这里是**线程退出的关键。**

如果超时未获取到任务，则表明当前线程长时间未处理任务，可以正常退出，并从线程池里移除该线程。

## 手写一个简单线程池的完整代码

下面，我们给出完整的所有代码：

拒绝策略相关类：

```
// 拒绝策略接口
public interface RejectedExecutionHandler {
    // 参数：r 代表被拒绝的任务，executor 代表线程池对象
    void rejectedExecution(Runnable r, ThreadPool executor);
}

// 忽略任务的拒绝策略
public class DiscardPolicy implements RejectedExecutionHandler {
    public void rejectedExecution(Runnable r, ThreadPool executor) {
        // do nothing
        RunnableWrapper wrapper = (RunnableWrapper) r;
        System.out.println("Task rejected: " + wrapper.getTaskId());
    }
}
```

为了通过输出日志，清晰的展现线程池中任务的运行流程，新增了**RunnableWrapper**用于记录taskId，方便日志监控。

```
public class RunnableWrapper implements Runnable{
    private final Integer taskId;

    public RunnableWrapper(Integer taskId) {
```

```
        this.taskId = taskId;
    }

    public Integer getTaskId() {
        return this.taskId;
    }

    @Override
    public void run() {
        System.out.println("Task " + taskId + " is running.");
        try {
            Thread.sleep(100);
        } catch (Exception e) {
            e.printStackTrace();
            // ignore
        }
        System.out.println("Task " + taskId + " is completed.");
    }
}
```

线程池接口

```
// 线程池接口
public interface ThreadPool {

    // 提交任务到线程池
    void execute(Runnable task);

    // 优雅关闭
    void shutdown();

    //立即关闭
    List<Runnable> shutdownNow();
}
```

工作线程类：

```
// 定义一个工作线程类
public class WorkerThread extends Thread {
    private List<WorkerThread> threads;
    // 空闲时长
    private long keepAliveTime;
    // 用于从任务队列中取出并执行任务
    private BlockingQueue<Runnable> taskQueue;

    // 构造方法，传入任务队列
    public WorkerThread(long keepAliveTime, BlockingQueue<Runnable> taskQueue, List<WorkerThread> threads) {
        this.keepAliveTime = keepAliveTime;
        this.taskQueue = taskQueue;
        this.threads = threads;
    }

    // 重写run方法
    @Override
    public void run() {
        long lastActiveTime = System.currentTimeMillis();
        // 循环执行，直到线程被中断
        Runnable task;
        while (!Thread.currentThread().isInterrupted() || !taskQueue.isEmpty()) {
            try {
                // 从任务队列中取出一个任务，如果队列为空，则阻塞等待
                task = taskQueue.poll(keepAliveTime, TimeUnit.MILLISECONDS);
                RunnableWrapper wrapper = (RunnableWrapper) task;
                if (task != null) {
                    System.out.printf("WorkerThread %s, poll current task: %s%n",
Thread.currentThread().getName(), wrapper.getTaskId());
                    task.run();
                    lastActiveTime = System.currentTimeMillis();
                } else if (System.currentTimeMillis() - lastActiveTime >= keepAliveTime) {
                    // 从线程池中移除
                    threads.remove(this);
                    System.out.printf("WorkerThread %s exit %n", Thread.currentThread().getName());
                    break;
                }
            } catch (Exception e) {
                // 从线程池中移除
                System.out.printf("WorkerThread %s occur exception%n", Thread.currentThread().getName());
                threads.remove(this);
                e.printStackTrace();
                // 如果线程被中断，则退出循环
                break;
            }
        }
    }
```

```
}
```

简单线程池实现类

```
public class SimpleThreadPool implements ThreadPool {
    // 线程池初始化时的线程数量
    private int initialSize;
    // 线程池最大线程数量
    private int maxSize;
    // 线程池核心线程数量
    private int coreSize;
    // 任务队列
    private BlockingQueue<Runnable> taskQueue;
    // 用于存放和管理工作线程的集合
    private List<WorkerThread> threads;
    // 是否已经被shutdown标志
    private volatile boolean isShutdown = false;
    // 默认的拒绝策略
    private final static RejectedExecutionHandler DEFAULT_REJECT_HANDLER = new AbortPolicy();
    // 拒绝策略成员变量
    private final RejectedExecutionHandler rejectHandler;
    // 线程空闲时长
    private long keepAliveTime;

    public SimpleThreadPool(int initialSize, int maxSize, int coreSize, int queueSize, long keepAliveTime) {
        this(initialSize, maxSize, coreSize, queueSize, keepAliveTime, DEFAULT_REJECT_HANDLER);
    }

    public SimpleThreadPool(int initialSize, int maxSize, int coreSize, int queueSize, long keepAliveTime,
RejectedExecutionHandler rejectHandler) {
        System.out.printf("初始化线程池: initialSize: %d, maxSize: %d, coreSize: %d%n", initialSize, maxSize,
coreSize);
        // 初始化参数
        this.initialSize = initialSize;
        this.maxSize = maxSize;
        this.coreSize = coreSize;
        taskQueue = new LinkedBlockingQueue<>(queueSize);
        threads = new ArrayList<>(initialSize);
        this.rejectHandler = rejectHandler;
        this.keepAliveTime = keepAliveTime;
        // 初始化方法，创建一定数量的工作线程，并启动它们
        for (int i = 0; i < initialSize; i++) {
            WorkerThread workerThread = new WorkerThread(keepAliveTime, taskQueue, threads);
            workerThread.start();
            threads.add(workerThread);
        }
    }

    @Override
    public void execute(Runnable task) {
        if (isShutdown) {
            throw new IllegalStateException("ThreadPool is shutdown");
        }
        RunnableWrapper wrapper = (RunnableWrapper) task;
        System.out.printf("put task: %s %n" , wrapper.getTaskId());

        // 当线程数量小于核心线程数时，创建新的线程
        if (threads.size() < coreSize) {
            addWorkerThread(task);
            System.out.printf("小于核心线程数，创建新的线程: thread count: %d, queue remainingCapacity : %d%n",
threads.size(), taskQueue.remainingCapacity());
        } else if (!taskQueue.offer(task)) {
            // 当队列已满时，且线程数量小于最大线程数量时，创建新的线程
            if (threads.size() < maxSize) {
                addWorkerThread(task);
                System.out.printf("队列已满，创建新的线程: thread count: %d, queue remainingCapacity : %d%n",
threads.size(), taskQueue.remainingCapacity());
            } else {
                rejectHandler.rejectedExecution(task, this);
            }
        } else {
            System.out.printf("任务放入队列: thread count: %d, queue remainingCapacity : %d%n", threads.size(),
taskQueue.remainingCapacity());
        }
    }


    // 创建新的线程，并执行任务
    private void addWorkerThread(Runnable task) {
        WorkerThread workerThread = new WorkerThread(keepAliveTime, taskQueue, threads);
        workerThread.start();
        threads.add(workerThread);
        // 任务放入队列
        try {
            taskQueue.put(task);
        } catch (InterruptedException e) {
```

```
            throw new RuntimeException(e);
        }
    }


    // 关闭线程池，等待所有线程执行完毕
    @Override
    public void shutdown() {
        System.out.printf("shutdown thread, count: %d, queue remainingCapacity : %d%n", threads.size(),
taskQueue.remainingCapacity());
        // 修改状态
        isShutdown = true;
        for (WorkerThread thread : threads) {
            // 中断线程
            thread.interrupt();
        }
    }

    @Override
    public List<Runnable> shutdownNow() {
        System.out.printf("shutdown thread now, count: %d, queue remainingCapacity : %d%n", threads.size(),
taskQueue.remainingCapacity());

        // 修改状态
        isShutdown = true;
        // 清空队列
        List<Runnable> remainingTasks = new ArrayList<>();
        taskQueue.drainTo(remainingTasks);

        // 中断所有线程
        for (WorkerThread thread : threads) {
            thread.interrupt();
        }
        // 返回未执行任务集合
        return remainingTasks;
    }
}
```

### 第六步：验证线程池

接下来，我们编写一个测试用例来验证我们的线程池是否能正常运行。

最后，我们编写了一个测试用例**SimpleThreadPoolTest**，

```
// 定义一个测试用例类
public class SimpleThreadPoolTest {
    public static void main(String[] args) throws InterruptedException {
        SimpleThreadPool threadPool = new SimpleThreadPool(1, 4, 2, 3, 2000, new DiscardPolicy());
        for (int i = 0; i < 10; i++) {
            threadPool.execute(new RunnableWrapper(i));
        }
        Thread.sleep(10_000);
        threadPool.shutdown();
    }
}
```

这个测试用例创建了一个拥有1个初始线程、最多4个线程、核心线程数为2、队列长度为3，空闲线程保留时间为2000毫秒的线程池。

它使用了一个简单的拒绝策略，当任务被拒绝时，它会打印一条消息。

然后，测试用例向线程池中提交了10个简单的任务，每个任务都会打印一条消息，然后睡眠100毫秒，再打印一条消息。

最后，测试用例调用了shutdown方法来关闭线程池。

当我们运行这个测试用例时，会看到类似下面的输出：

```
初始化线程池: initialSize: 1, maxSize: 4, coreSize: 2
put task: 0
小于核心线程数，创建新的线程: thread count: 2, queue remainingCapacity : 2
put task: 1
WorkerThread Thread-1, poll current task: 0
WorkerThread Thread-0, poll current task: 1
Task 1 is running.
任务放入队列: thread count: 2, queue remainingCapacity : 2
put task: 2
Task 0 is running.
任务放入队列: thread count: 2, queue remainingCapacity : 2
put task: 3
```

```
任务放入队列: thread count: 2, queue remainingCapacity : 1
put task: 4
任务放入队列: thread count: 2, queue remainingCapacity : 0
put task: 5
WorkerThread Thread-2, poll current task: 2
Task 2 is running.
队列已满, 创建新的线程: thread count: 3, queue remainingCapacity : 0
put task: 6
WorkerThread Thread-3, poll current task: 3
Task 3 is running.
队列已满, 创建新的线程: thread count: 4, queue remainingCapacity : 0
put task: 7
Task rejected: 7
put task: 8
Task rejected: 8
put task: 9
Task rejected: 9
Task 2 is completed.
Task 1 is completed.
WorkerThread Thread-2, poll current task: 4
Task 4 is running.
Task 0 is completed.
Task 3 is completed.
WorkerThread Thread-1, poll current task: 6
WorkerThread Thread-0, poll current task: 5
Task 6 is running.
Task 5 is running.
Task 5 is completed.
Task 6 is completed.
Task 4 is completed.
WorkerThread Thread-3 exit
WorkerThread Thread-1 exit
WorkerThread Thread-0 exit
WorkerThread Thread-2 exit
shutdown thread, count: 0, queue remainingCapacity : 3
```

从输出中可以看出，线程池中最多有4个线程在同时运行。当有空闲线程时，新提交的任务会被立即执行；否则，新提交的任务会被添加到任务队列中等待执行。

ok，到了这里，大家能够帮助大家更好地理解上文中实现的简单线程池。

接下来，大家可以去进一步技术深入，去研究线程池的源码了。

### 实操总结

通过实操，我们一步一步地实现了一个简单的线程池。

我们从最基本的功能开始，逐步添加了拒绝策略、自动调节线程资源等高级功能，并对线程池进行了优化。

通过这个过程，我们可以更好地理解线程池的工作原理和实现细节。

当然，这只是一个简单的示例，实际应用中的线程池可能会更加复杂和强大。与Java标准库中提供的线程池相比，仍然存在一些不足之处。例如：

- 没有提供足够的构造参数和方法，让用户可以更好地控制和监控线程池的行为。
- 没有提供足够多的拒绝策略，让用户可以根据不同的场景选择不同的拒绝策略。
- 没有提供定时任务和周期性任务的执行功能。
- 没有提供足够完善的错误处理机制。

Java标准库中提供的线程池实现（如ThreadPoolExecutor类），则在这些方面都做得更好。

它提供了丰富的构造参数和方法，让用户可以更好地控制和监控线程池的行为；

它提供了多种拒绝策略，让用户可以根据不同的场景选择不同的拒绝策略；

它还提供了ScheduledThreadPoolExecutor类，用来执行定时任务和周期性任务；

它还提供了完善的错误处理机制，可以帮助用户更好地处理异常情况。

如果你想要深入了解Java标准库中提供的线程池实现，可以参考以下清华大学出版社，所出版的尼恩 Java高并发三部曲之二 《Java高并发核心编程 卷2 加强版》。

## 09 字节面试：如何避免TheadLocal导致的内存泄漏问题?

讲义地址： https://mp.weixin.qq.com/s/Mc06De42MvV-JXsJWbZ3mw

### 第一个要点： 什么是ThreadLocal?

（1）问题1： 什么是ThreadLocal?

ThreadLocal的基本使用，

源代码在：Java高并发核心编程（卷2）源码
https://gitee.com/crazymaker/Java-high-concurrency-core-Programming-Volume-2-source-code.git

（2）问题2： ThreadLocal的作用和优劣势有哪些?

（3）问题3： ThreadLocal的使用场景?

### 第二个要点： ThreadLocal实现原理

（1）问题1： ThreadLocal的内部结构演进

（2）问题2： ThreadLocalMap对象和Entry是什么

（3）问题3： ThreadLocal的结构模型?

### 第三个要点： 什么是内存泄漏，ThreadLocal是怎么造成内存泄露的?

（1）问题1： 什么是内存泄漏?

（2）问题2： ThreadLocal是怎么造成内存泄露的?

（3）问题3： 什么是key的泄漏?

（4）问题4： 什么是value的泄漏?

### 第四个要点： 如何解决内存泄漏?

（1）问题1： key的泄漏如何解决?

（2）问题2： value的泄漏如何解决?

（3）问题3： JDK中ThreadLocal清理策略有哪些?

（4）问题4： 业务主动清理：手动清除解决内存泄漏。

### 第五个要点： 使用ThreadLocal有哪些性能问题及优化措施?

（1）问题1：ThreadLocal有哪些性能问题?

（2）问题2： 编程规范：推荐使用 static final 修饰ThreadLocal对象。

（3）问题3： 编程规范JDK三个内存泄露策略彻底失效，为什么？ 如何解决?

### 第六个要点： ThreadLocal总结

## 10 得物面试：消息0丢失，Kafka如何实现?

讲义地址： https://mp.weixin.qq.com/s/nk4_mlOlvyQDA5qlkd-aLg

### 第一个要点： kafka的消息的发送总流程

要实现0丢失，或者不丢失， 那么整个消息传递的链路，都不能有闪失

（1）问题1： kafka的消息的发送总流程，包括几个环节，大致是什么?

### 第二个要点： 第一阶段：生产阶段如何实现0丢失方式

（1）问题1： 生产阶段，为何会造成消息丢失?

（2）问题2： Producer（生产者）保证消息不丢失的方法是什么?

### 第三个要点： 第二阶段：Broker端保证消息不丢失的方法

（1）问题1： 存储阶段，为何会造成消息丢失?

（2）问题2： Broker端 保证消息不丢失的方法是什么?

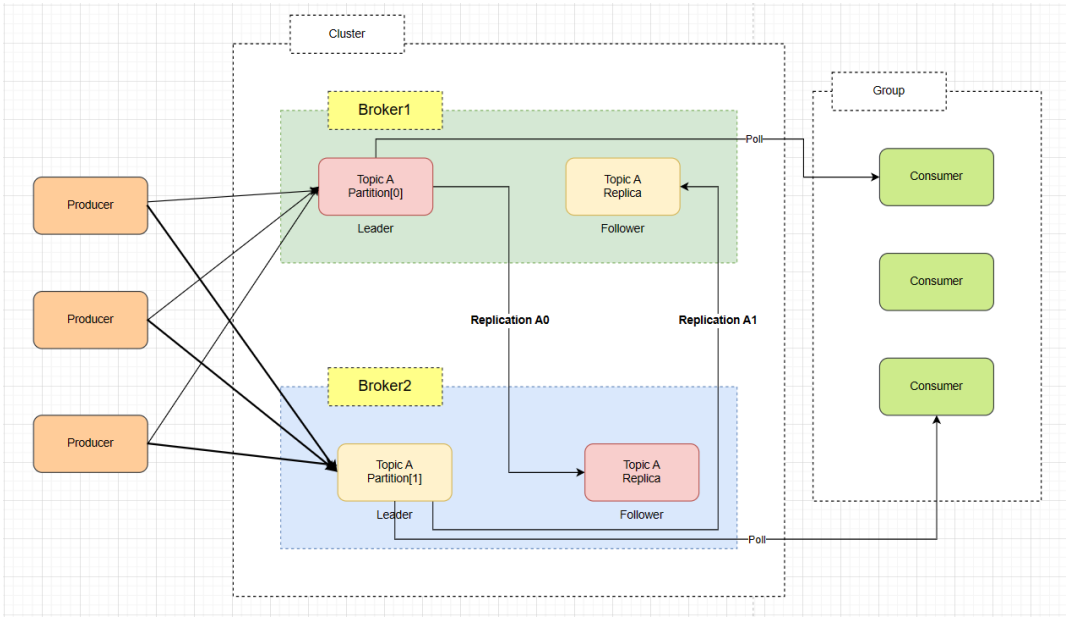
### 第三个要点： 第三阶段：消费端保证消息不丢失的方法

（1）问题1： 消费阶段，为何会造成消息丢失?

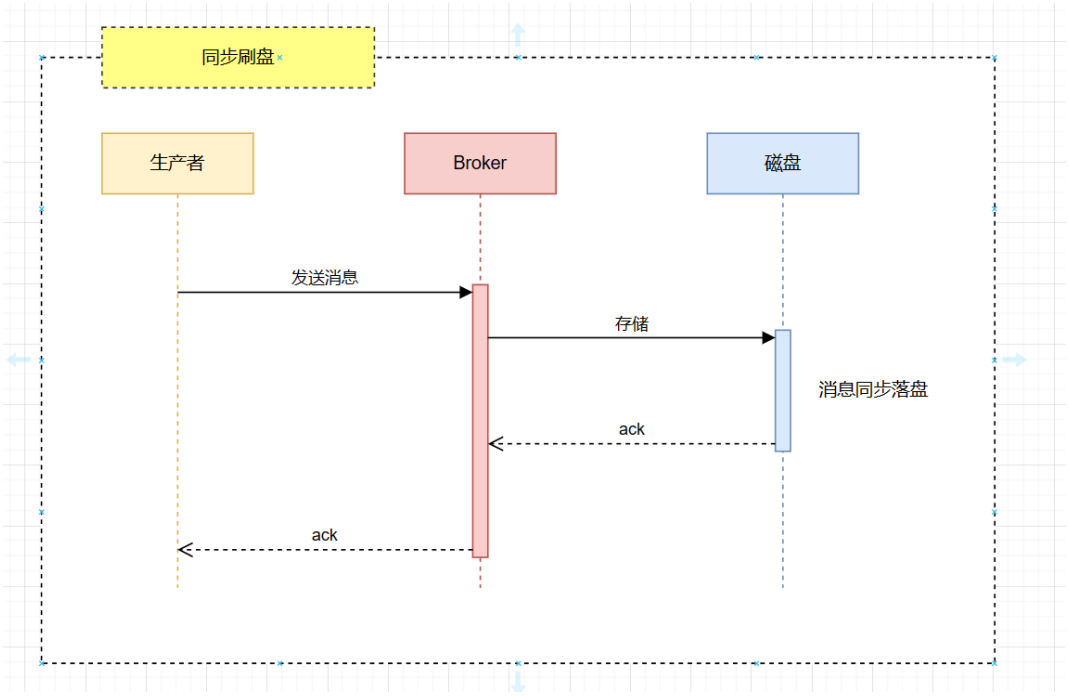
（2）问题2： 消费端 保证消息不丢失的方法是什么?

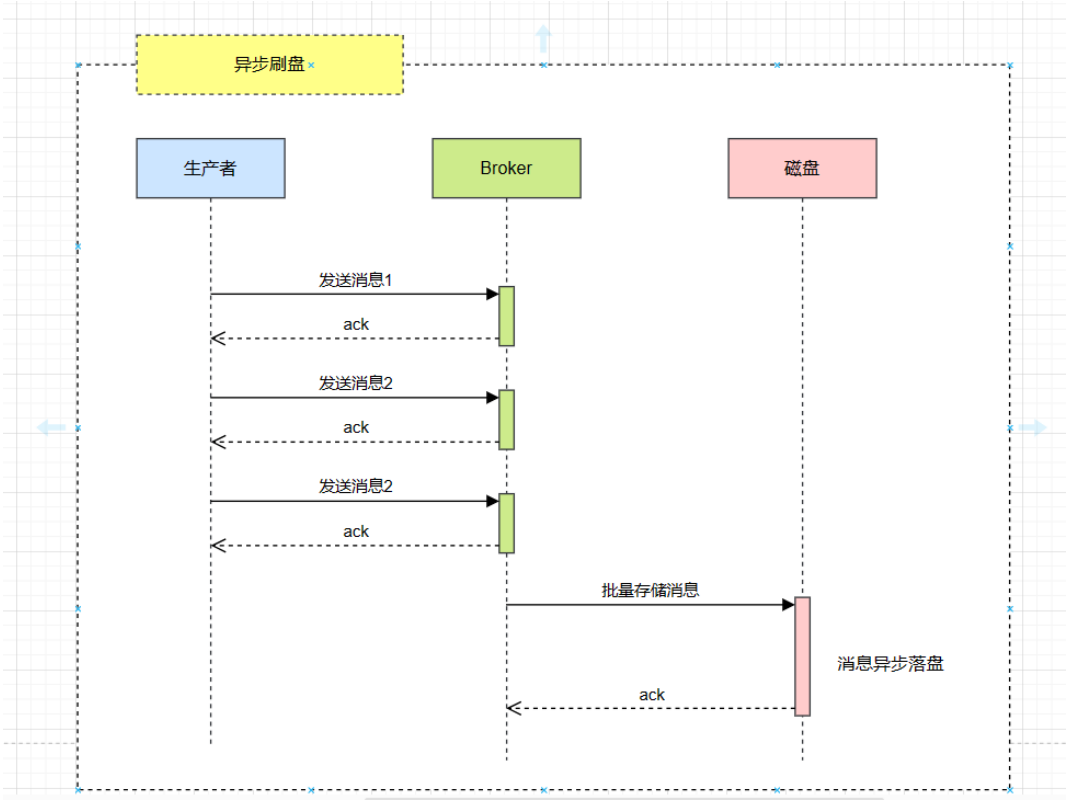
### 第四个要点： 总结一下，卡夫卡消息不丢失的总体 方法

视频中的卡夫卡架构图




kafka-同步刷盘与异步刷盘






## 11 面试题目：Rocketmq消息0丢失，如何实现?

讲义地址：https://mp.weixin.qq.com/s/j2dWEk4JVPToZBQ4v7orTg

**第1个要点： RocketMQ的消息的发送总流程**

要实现0丢失，或者不丢失， 那么整个消息传递的链路，都不能有闪失

- 问题1： RocketMQ的消息的发送总流程，包括几个环节，大致是什么?

**第2个要点： 第一阶段：生产阶段如何实现0丢失方式**

- 问题1： 生产阶段，为何会造成消息丢失?
- 问题2： Producer（生产者）保证消息不丢失的方法是什么?

**第3个要点： 第二阶段：Broker端保证消息不丢失的方法**

- 问题1：存储阶段，为何会造成消息丢失?
- 问题2：Broker端保证消息不丢失的方法是什么?

**第4个要点： 第三阶段：消费端保证消息不丢失的方法**

- 问题1： 消费阶段，为何会造成消息丢失?
- 问题2： 消费端 保证消息不丢失的方法是什么?
- 问题3：消费端采用异步消费模式，如何保证消息不丢失?

**第5个要点： 总结一下，RocketMQ消息不丢失的总体 方法**

## 12 面试官：听说你sql写的挺溜的，你说一说查询sql的执行过程

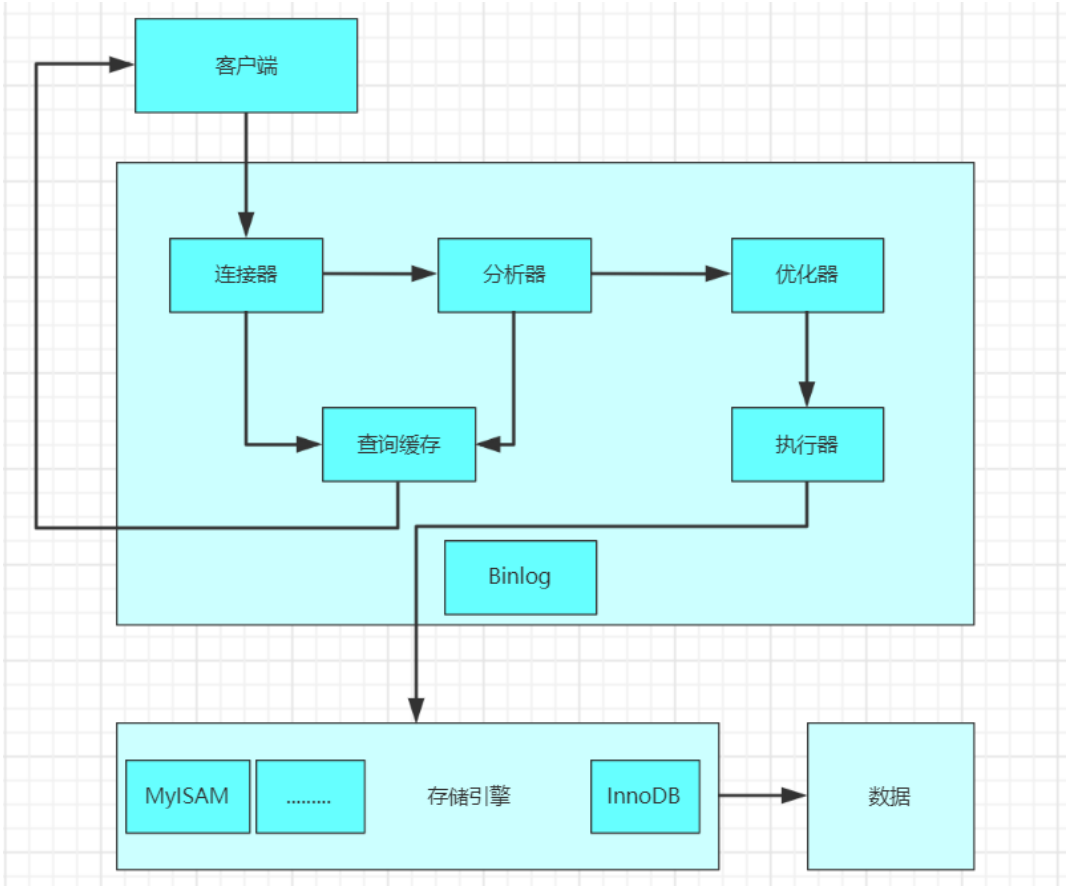
当希望Mysql能够高效的执行的时候，最好的办法就是清楚的了解Mysql是如何执行查询的，只有更加全面的了解SQL执行的每一个过程，才能更好的进行SQl的优化。

当执行一条查询的SQl的时候大概发生了一下的步骤：

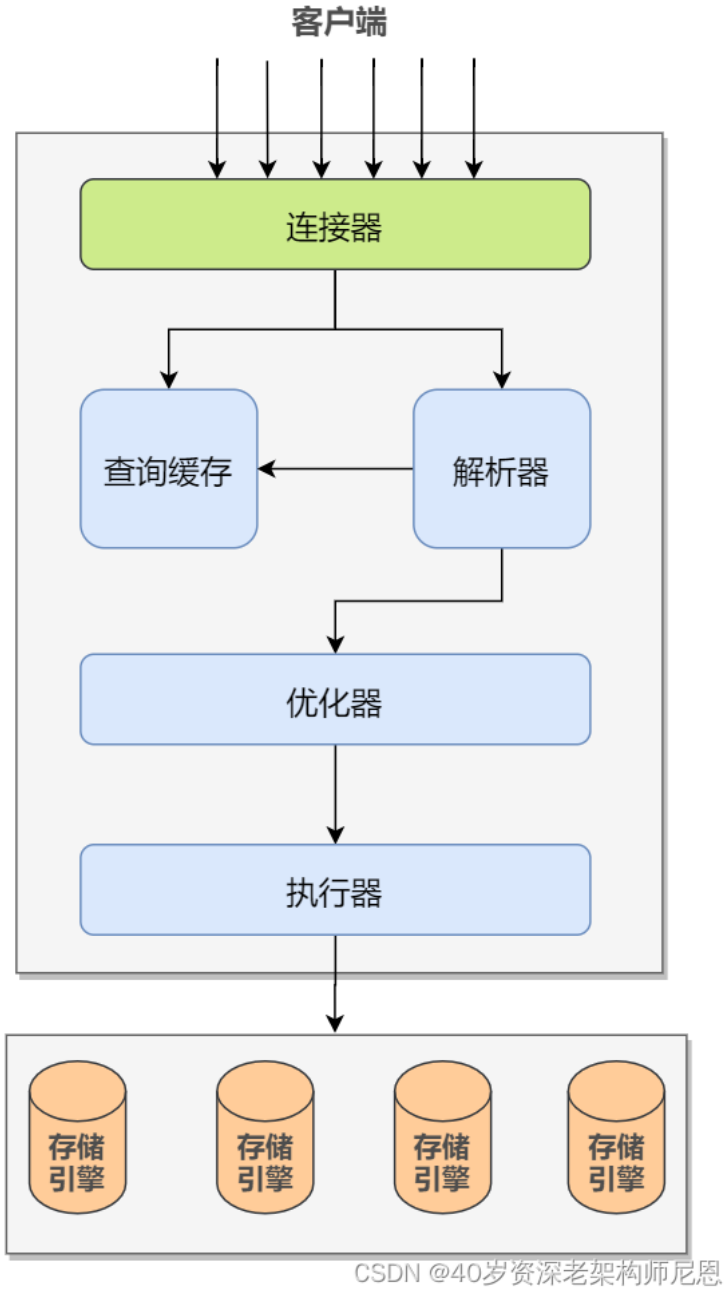
1. 客户端发送查询语句给服务器。
2. 服务器首先检查缓存中是否存在该查询，若存在，返回缓存中存在的结果。若是不存在就进行下一步。
3. 服务器进行SQl的解析、语法检测和预处理，再由优化器生成对应的执行计划。
4. Mysql的执行器根据优化器生成的执行计划执行，调用存储引擎的接口进行查询。
5. 服务器将查询的结果返回客户端。

### Mysql的执行的流程

Mysql的执行的流程图如下图所示：




再来一个图





这里以一个实例进行说明Mysql的的执行过程，新建一个User表，如下：

```
// 新建一个表
DROP TABLE IF EXISTS User;
CREATE TABLE `User` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(10) DEFAULT NULL,
  `age` int DEFAULT 0,
  `address` varchar(255) DEFAULT NULL,
  `phone` varchar(255) DEFAULT NULL,
  `dept` int,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT CHARSET=utf8;

// 并初始化数据，如下
INSERT INTO User(name,age,address,phone,dept)VALUES('张三',24,'北京','13265543552',2);
INSERT INTO User(name,age,address,phone,dept)VALUES('张三三',20,'北京','13265543557',2);
INSERT INTO User(name,age,address,phone,dept)VALUES('李四',23,'上海','13265543553',2);
INSERT INTO User(name,age,address,phone,dept)VALUES('李四四',21,'上海','13265543556',2);
INSERT INTO User(name,age,address,phone,dept)VALUES('王五',27,'广州','13265543558',3);
INSERT INTO User(name,age,address,phone,dept)VALUES('王五五',26,'广州','13265543559',3);
INSERT INTO User(name,age,address,phone,dept)VALUES('赵六',25,'深圳','13265543550',3);
INSERT INTO User(name,age,address,phone,dept)VALUES('赵六六',28,'广州','13265543561',3);
INSERT INTO User(name,age,address,phone,dept)VALUES('七七',29,'广州','13265543562',4);
INSERT INTO User(name,age,address,phone,dept)VALUES('八八',23,'广州','13265543563',4);
INSERT INTO User(name,age,address,phone,dept)VALUES('九九',24,'广州','13265543564',4);
```

现在针对这个表发出一条SQl查询：

查询每个部门中25岁以下的员工个数大于3的员工个数和部门编号，并按照人工个数降序排序和部门编号升序排序的前两个部门。

```
SELECT dept,COUNT(phone) AS num FROM User WHERE age< 25 GROUP BY dept HAVING num >= 3 ORDER BY num DESC,dept
ASC LIMIT 0,2;
```

## 执行连接器

开始执行这条sql时，会检查该语句是否有权限，若是没有权限就直接返回错误信息，有权限会进行下一步，校验权限的这一步是在图一的连接器进行的，对连接用户权限的校验。

## 执行检索内存

相连建立之后，履行查询语句的时候，会先行检索内存，Mysql会先行冗余这个sql与否履行过，以此`Key-Value`的形式平缓适用内存中，Key是`检索预定`，Value是`结果集`。

假如内存key遭击中，便会间接回到给客户端，假如没命中，便会履行后续的操作，完工之后亦会将结果内存上去，当下一次进行查询的时候也是如此的循环操作。

## 执行分析器

分析器主要有两步：（1）词法分析（2）语法分析

> 词法分析主要执行`提炼关键性字`，比如select，`提交检索的表`，`提交字段名`，`提交检索条件`。
>
> 语法分析主要执行辨别你`输出的sql与否准确`，是否`合乎mysql的语法`。

当Mysql没有命中内存的时候，接着执行的是 FROM student 负责把数据库的表文件加载到内存中去，`WHERE age< 60`，会把所示表中的数据进行过滤，取出符合条件的记录行，生成一张临时表，如下图所示。

| id | name | age | address | phone | dept |
|---|---|---|---|---|---|
| 40 | 张三 | 24 | 北京 | 13265543552 | 2 |
| 41 | 张三三 | 20 | 北京 | 13265543557 | 2 |
| 42 | 李四 | 23 | 上海 | 13265543553 | 2 |
| 43 | 李四四 | 21 | 上海 | 13265543556 | 2 |
| 49 | 八八 | 23 | 广州 | 13265543563 | 4 |
| 50 | 九九 | 24 | 广州 | 13265543564 | 4 |

`GROUP BY dept` 会把上图的临时表分成若干临时表，切分的过程如下图所示：

| id | name | age | address | phone | dept |
|---|---|---|---|---|---|
| 40 | 张三 | 24 | 北京 | 13265543552 | 2 |
| 41 | 张三三 | 20 | 北京 | 13265543557 | 2 |
| 42 | 李四 | 23 | 上海 | 13265543553 | 2 |
| 43 | 李四四 | 21 | 上海 | 13265543556 | 2 |

| id | name | age | address | phone | dept |
|---|---|---|---|---|---|
| 49 | 八八 | 23 | 广州 | 13265543563 | 4 |
| 50 | 九九 | 24 | 广州 | 13265543564 | 4 |

查询的结果只有部门2和部门3才有符合条件的值，生成如上两图的临时表。接着执行`SELECT后面的字段`，SELECT后面可以是`表字段`也可以是`聚合函数`。

这里SELECT的情况与是否存在`GROUP BY`有关，若是不存在Mysql直接按照上图内存中整列读取。若是存在分别SELECT临时表的数据。

最后生成的临时表如下图所示：

| dept | num |
|---|---|
| 2 | 4 |
| 4 | 2 |

紧接着执行`HAVING num>2`过滤员工数小于等于2的部门，对于`WHERE`和`HAVING`都是进行过滤，那么这两者有什么不同呢?

第一点是WHERE后面只能对表字段进行过滤，不能使用聚合函数，而HAVING可以过滤表字段也可以使用聚合函数进行过滤。

第二点是WHERE是对执行from USer操作后，加载表数据到内存后，WHERE是对`原生表的字段`进行过滤，而HAVING是对`SELECT后的字段进行过滤`，也就是WHERE`不能使用别名进行过滤`。

因为执行WHERE的时候，还没有SELECT，还没有给字段赋予别名。接着生成的临时表如下图所示：

```
+------+------+
| dept | num |
+------+------+
|    2 |    4 |
+------+------+
```

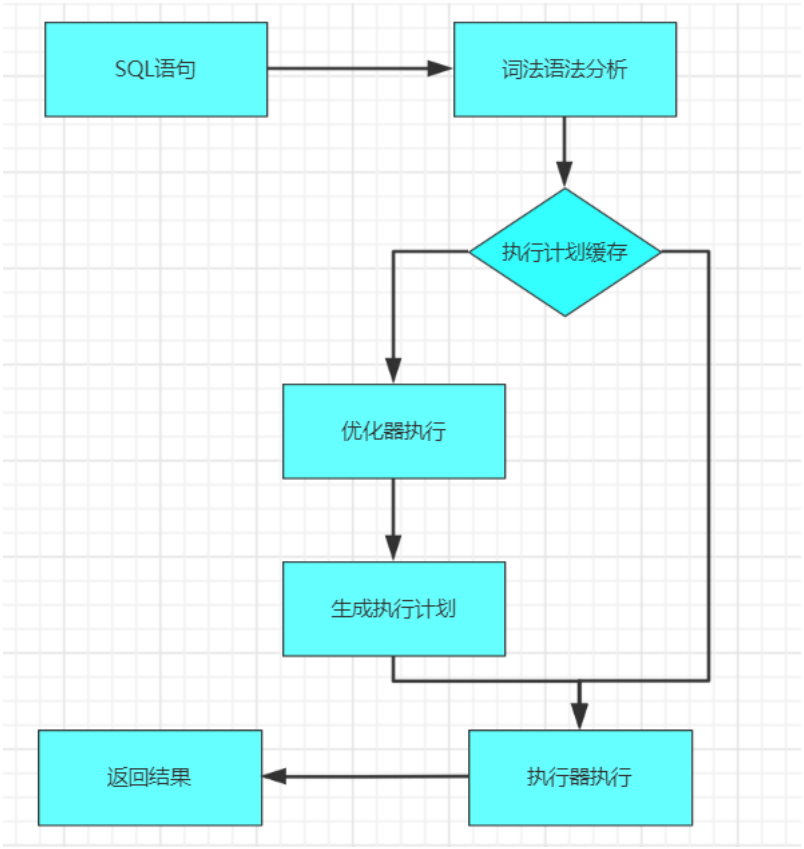
最后在执行 `ORDER BY后面的排序` 以及 `limit0,2` 取得前两个数据，因为这里数据比较少，没有体现出来。最后生成得结果也是如上图所示。接着判断这个sql语句 `是否有语法错误`，`关键性词与否准确` 等等。

## 执行优化器

查询优化器会将解析树转化成执行计划。一条查询可以有多种执行方法，最后都是返回相同结果。优化器的作用就是找到这其中 `最好的执行计划`。

生成执行计划的过程会消耗较多的时间，特别是存在许多可选的执行计划时。如果在一条SQL语句执行的过程中将该语句对应的最终执行计划进行缓存。

当 `相似的语句` 再次被输入服务器时，就可以直接 `使用已缓存的执行计划`，从而跳过SQL语句生成执行计划的整个过程，进而可以提高语句的执行速度。




MySQL使用基于成本的查询优化器。它会尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最少的一个。

## 执行执行器

由优化器生成得执行计划，交由执行器进行执行，执行器调用存储引擎得接口，存储引擎获取数据并返回，结束整个查询得过程。

这里之讲解了select的过程，对于update这些修改数据或者删除数据的操作，会涉及到事务，会使用两个日志模块，redo log和binlog日志。

以前的Mysql的默认存储引擎MyISAM引擎是没redo log的，而现在的默认存储引擎InnoDB引擎便是透过redo 复杂度来拥护事务的，保证事务能够准确的回滚或者提交，保证事务的ACID。

# 13 题: 美团面试：索引的设计规范，你知道那些?

本题的答案地址（自由自由圈）

https://mp.weixin.qq.com/s/86pc-l6SrryEkafeeu3WbA

也可以扫码打开

## 基础知识

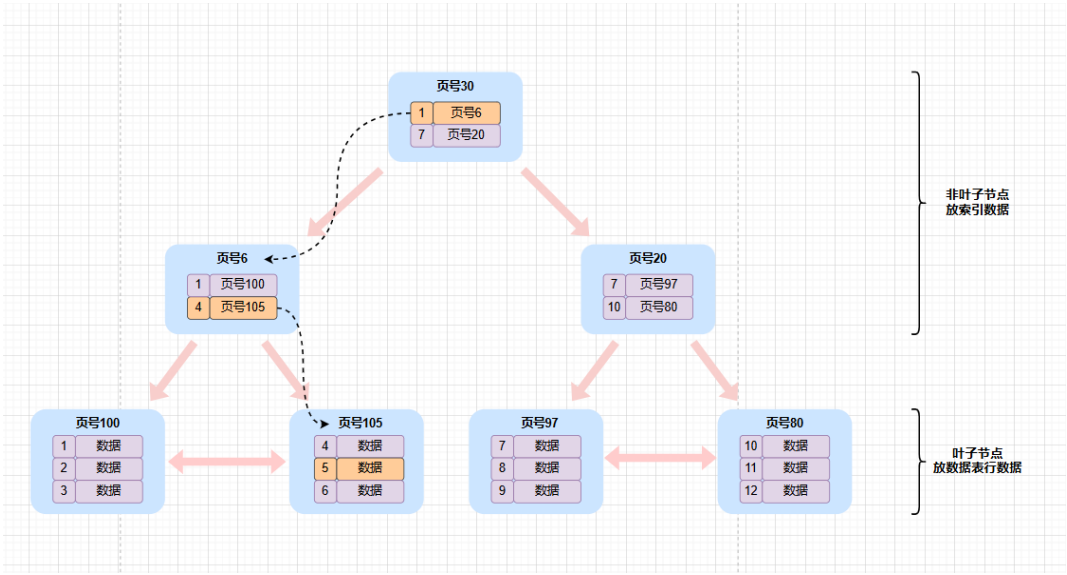
## 一、聚集索引

实际的数据页只能按照一颗B+树进行排序，因此每张表只能拥有一个聚集索引。

在多数情况下，查询优化器倾向于采用聚集索引。因为聚集索引能够在B+树索引的叶子节点上直接找到数据。

此外，由于定义了数据的逻辑顺序，聚集索引能够特别快地访问针对范围值的查询。

查询优化器能够快速发现某一段范围的数据页需要扫描。




## 二、二级（辅助）索引 / 非聚集索引

对于辅助索引，叶子节点并不包含行记录的全部数据。

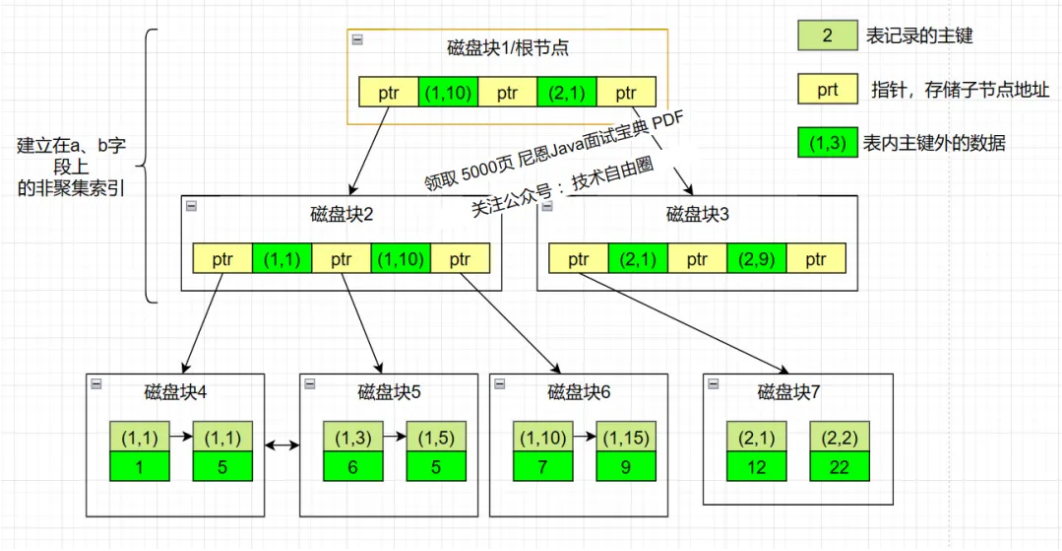
叶子节点除了包含键值以外，每个叶子节点中的索引行中还包含了一个书签（bookmark）。该书签用来告诉InnoDB存储引擎哪里可以找到与索引相对应的行数据。

由于InnoDB存储引擎表时索引组织表，因此InnoDB存储引擎的辅助索引的书签就是相应行数据的聚集索引键key。

辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。

当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引来找到一个完整的行记录。

如下图：




**特点：**

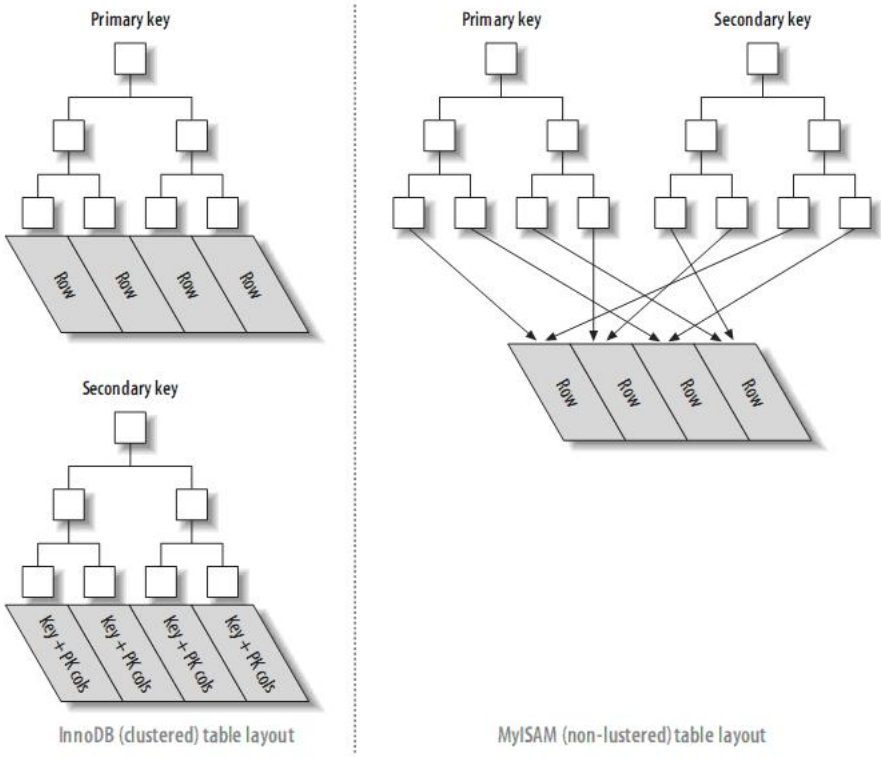
1）**手动创建，可以有多个**

2）**内节点包含索引列、主键列、页号（page_no）**

3）**叶子节点只包含索引列以及记录主键的值**

4）**每层节点都是按照索引列的值从小到大排序（索引列值相同时按照主键排序）**

## 聚簇索引和非聚簇索引表的对比：



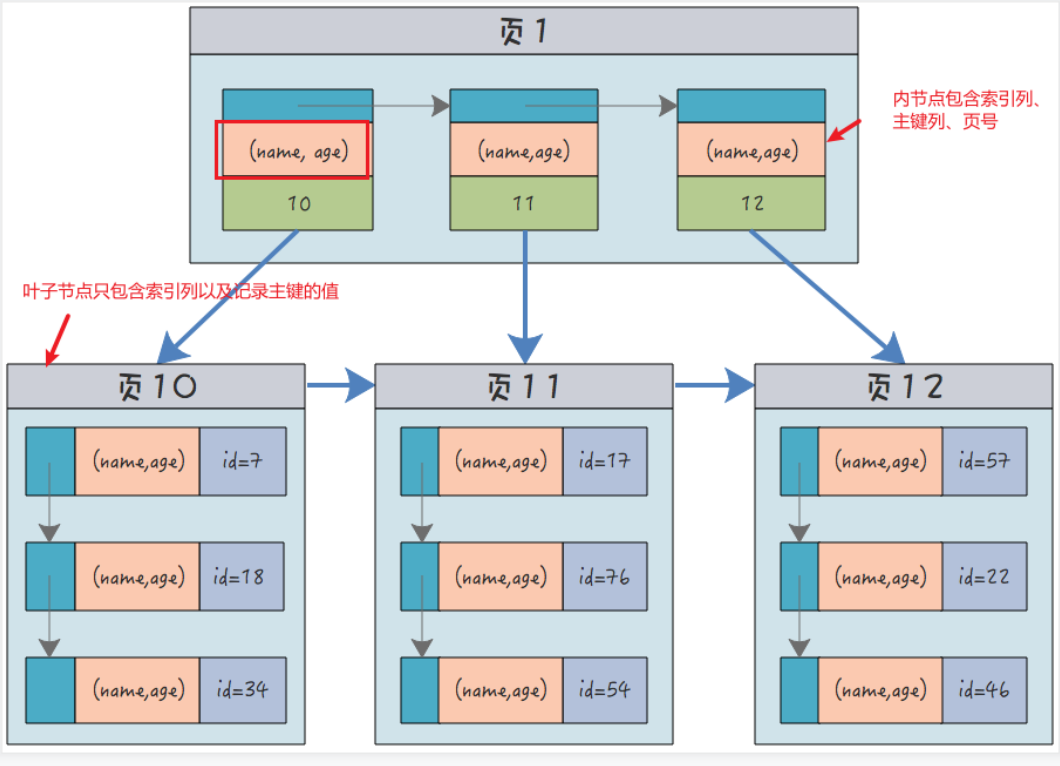

## 三、联合索引

联合索引是一种特殊的二级索引。

联合索引指的是同时对多列创建的索引，创建联合索引后，叶子节点会同时包含每个索引列的值，并且同时根据多列排序，这个排序和我们所理解的字典序类似。

每个叶子节点同时保存了所有的索引列，除此之外，还是只包含了主键id。

如下图：




**特点：**

**手动创建，可以有多个**

**内节点包含索引列、主键列、页号**

3）**叶子节点只包含索引列以及记录主键的值**

每层节点先按照索引中的**第1列**排序。第1列值相等时，按第2列排序。第2列值相等时，按第3列排序 依次类推，所有列都相等时按照主键排序。

## 五）什么是最左前缀原则？什么是最左匹配原则

1）顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。

2）最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如
a = 1 and b = 2 and c > 3 and d = 4
如果建立(a,b,c,d)顺序的索引，d是用不到索引的，

如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

3）=和in可以乱序，比如a = 1 and b = 2 and c = 3
建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

## 四、覆盖索引-covering index

即从辅助索引中就可以得到查询的记录，而不需要查询聚集索引中的记录。

使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以**减少大量的IO操作。**

**只需要在一棵索引树上**就能获取SQL所需的所有列数据，**无需回表**，速度更快。

覆盖索引是一种非常强大的工具，能大大提高查询性能，只需要读取索引而不用读取数据有**以下一些优点**：

1）索引项通常比记录要小，所以MySQL访问更少的数据

2）索引都按值的大小顺序存储，相对于随机访问记录，需要更少的I/O

3）覆盖索引对于InnoDB表尤其有用，因为InnoDB使用聚集索引组织数据，如果二级索引中包含查询所需的数据，就不再需要在聚集索引中查找了。

## 14：字节面试：MySQL死锁 是什么，如何解决？

讲义地址： https://mp.weixin.qq.com/s/eRYvWoUzbrcX0RsYNaobWg

### 第一个要点：Mysql死锁及锁的分类

1）question：什么是Mysql死锁?

2）question：Mysql有哪些锁?
1.从操作粒度分类

- 表级锁
- 行级锁
- 页级锁

2.从操作类型分类

- 读锁
- 写锁
- 意向锁

3.从操作性能分类

- 乐观锁
- 悲观锁

### 第二个要点：InnoDB存储引擎三种行锁模式

- 1）记录锁 Record Lock
  RC、RR隔离级别支持
- 2）间隙锁 Gap Lock
  RR隔离级别支持
- 3）临键锁 Next-key Lock
  RR隔离界别支持

### 第三个要点：事务的隔离级别和锁的关系

1）事务的隔离级别有哪些?
引发的问题（脏读/幻读/不可重复读）
2）事务隔离级别和锁的关系

### 第四个要点：死锁产生的原因和解决方案

1）如何查看Innodb锁行锁使用情况?

2）表级锁死锁是如何产生及解决的?

3）行级锁死锁是如何产生及解决的?

- 1.未走索引，升级为表锁
- 2.事务互相持有对方的锁
- 3.事务只有一个sql发生死锁

### 第五个要点：三个死锁案例分析

- 案例一：拆借款
- 案例二：有则更新，无则插入
- 案例三：死锁日志分析

### 第六个要点：实际开发中如何预防和处理死锁?

- 1.死锁产生的前提有哪些?
- 2.如何预防死锁呢?
- 3.遇到死锁该如何处理?

## 15：贝壳面试：MySQL联合索引，最左匹配原则是什么?

讲义地址： https://mp.weixin.qq.com/s/aq2-gR0Wj5L9wPTClH3-xw

第一个要点： MySQL调优的一个重点工作，建立联合索引实现索引覆盖

第二个要点： MySQL索引机制回顾

（1）问题1： 什么是索引?

（2）问题2： 如何创建、查询、删除索引?

第三个要点：MySQL联合索引

（1）问题1： 什么是联合索引?

（2）问题2： 联合索引存储结构?

（3）问题3： 最左前缀匹配原则有哪些?

（4）问题4： 为什么要遵循最左前缀匹配?

（5）问题5： 一定要遵循最左前缀匹配吗?

第四个要点： 联合索引注意事项?

## 16 MYSQL专题： 字节面试：MySQL什么时候 锁表？如何防止锁表?

讲义地址： https://mp.weixin.qq.com/s/972ifwCLVycmcosQJ3oRjA

**- Mysql调优的重点目标： 避免**锁表****

- 1.锁表带来的性能问题

- 2.锁表引发死锁

- 3.锁表降低系统可用性

- 4.锁表引发数据一致性问题

**- 回顾一下mysql锁的分类**

- 1.表级锁：

- 2.行级锁：

- 3.页级锁：

**- InnoDB存储引擎中的 表锁和行锁**

**- InnoDB的表级锁**

- 表级锁之一：表锁

- 表级锁之二：元数据锁

- 表级锁之三：意向锁

**- InnoDB的行级锁**

- InnoDB存储引擎三种行级锁

- 行级锁之一：记录锁(Record Locks)

- 行级锁之二：间隙锁(Gap Locks)

- 行级锁之三：临键锁(Next-Key Locks)

**- InnoDB如何**加锁？ **

- 第1个维度：InnoDB 如何加意向锁?

- 第2个维度：InnoDB如何加表级锁?:

- 第3个维度：InnoDB 如何加行级锁?:

**- 回到面试题：什么情况下，MySQL会锁定整个表?**

- 1.对表进行结构性修改:

- 2.手动锁定表:

- 3.MyISAM写操作:

- 4.两个或多个事务在同时修改一个表时:

- 5.索引操作:

- 6.并发操作:

- 7索引不可用时加锁操作:

- 8.索引选择不恰当:

- 9.更新和删除场景，where没命中索引（最常见场景）：

- 10.查询场景，索引失效的情况下，行锁升表锁

**-10.1.使用函数或操作符**

**-10.2.模糊查询**

**-10.3.不符合最左前缀原则**

**-10.4.数据类型不一致**

**-10.5.使用 OR 条件**

**-10.6.隐式类型转换**

**-10.7.范围条件**

**-10.8. NULL判断**

**-10.9.更新频繁的列**

**-10.10.查询优化器选择**

- MySQL会锁表的场景总结

**- 回到面试题第二问：如何减少或避免锁表**

- 1.使用合适的存储引擎
- 2.优化查询和索引
- 3.分解大事务
- 4.锁策略和隔离级别
- 5.分区和分表
- 6.避免长时间的锁定操作
- 7.监控和分析
- 8.读写分离
- 9.使用合适的锁
- 10.业务层优化
- 技术自由圈几个 和MYSQL 有关的核心面试真题

## 17：阿里面试：MVCC是如何实现的？请简述其工作原理。

讲义地址： https://mp.weixin.qq.com/s/rMTfN-Fan2cSSJZQgSsaOg

### 第一个要点：MVCC基础

（1）问题1：MVCC定义：多版本并发控制，允许多个事务同时读取同一数据行的不同版本。

（2）问题2：根本目标：解决读写冲突，提升并发性能。

（3）问题3：事务并发场景：包括读-读、读-写、写-写等。

（4）问题4：COW无锁架构：通过保存数据行的多个版本来实现无锁读。

### 第二个要点：MVCC与锁、事务隔离的关系

（1）问题1：MVCC与锁关系：MVCC通过多版本实现无锁读，但写操作仍需加锁。

（2）问题2：事务定义：一系列数据库操作的逻辑单位。

（3）问题3：ACID特性：原子性、一致性、隔离性、持久性。

（4）问题4：隔离级别：读未提交、读已提交、可重复读、串行化。

（5）问题5：三角关系：并发性、数据一致性、隔离级别之间存在权衡。

### 第三个要点：MVCC实现机制

（1）问题1：隐藏字段：如row_id、deleted_bit、trx_id、roll_ptr。

（2）问题2：Undo-log：记录数据修改前的状态，用于回滚和MVCC。

（3）问题3：Read-View：确定事务可见的数据版本。

（4）问题4：快照读与当前读区别。

### 第四个要点：ReadView详解

（1）问题1：定义：事务执行时的一致性视图。

（2）问题2：核心属性：包括创建ReadView时活跃的事务列表。

（3）问题1：读取规则：根据ReadView判断数据版本是否可见。

（4）问题4：生成规则：在事务开始时创建，基于当前活跃事务列表。

第五个要点：总结

## 18 Redis专题： 字节面试： 什么是跳表？请手写一个跳表。

字节面试： 请手写一个跳表。

讲义地址： https://mp.weixin.qq.com/s/WnG6DiiUptImg6NixCZQuw

源码仓库： Java高并发核心编程 卷2 一书的 源码仓库

### 第一个要点： 如何理解跳表（skiplist）

（1）问题1： 什么是跳表以及它的底层数据结构

（2）问题2： 如何提高有序单链表的查询效率?

（3）问题3： 跳表的特点

（4）问题4： 跳表的随机层数在插入性能上的优化

（5）问题5： 用跳表查询到底有多快?

### 第二个要点： 为什么Redis一定要用跳表实现ZSet

（1）问题1： Redis中ZSet底层数据结构

勘误：listpack紧凑列表，在Redis7.0中新增

（2）问题2： redis的zpilist和skiplist之间何时进行转换?

（3）问题3： Redis对跳表的实现及改进与优化

（4）问题4： Redis为什么使用跳表而不使用平衡树实现ZSet?

### 第三个要点： 跳表与平衡树、哈希表、B+树之间的比较

（1）问题1： 跳表与平衡树、哈希表的比较

（2）问题2： 跳表与B+树的比较

### 第四个要点： 基于Java实现一个跳表

（1）step1： 先定义好跳表的节点（Node）

（2）step2： 定义好 生产 随机层数的方法

（3）step3： 定义好 数据 插入 的方法

（4）step4： 定义好 数据 删除 的方法

（5）step5： 定义好 数据 查找 的方法

### 第五个要点： 跳表的总结

## 19 得物面试：为啥Redis用哈希槽，不用一致性哈希?

讲义地址：https://mp.weixin.qq.com/s/Q68UN34-BqxyQFtkJL98lg

### 1.第一个要点：数据分片

1）question: 哈希槽和一致性哈希是什么?

2）question：什么是数据分片？有什么用?

3）question：这道面试题从何开始作答?

### 2.第二个要点：经典哈希算法

1）question: 经典哈希是如何进行数据分片的?

2）question：经典哈希取模分片有什么问题?

3）question：经典哈希的问题有什么应对之策吗?

### 3.第三个要点：一致性Hash算法

1）question: 一致性hash是如何进行数据分片的？两个核心阶段

2）question: 槽位是如何映射到redis节点的呢?
3）question: 一致性hash的原理是什么?
4）question：一致性Hash的经典场景有什么?

### 4.第四个要点：一致性Hash优缺点

1）question：经典Hash和一致性Hash对比起来怎么样?

2）question：一致性Hash有什么问题吗？存在数据倾斜

3）question：一致性hash的数据倾斜有什么应对之策吗？虚拟节点

4）question：为什么使用Hash槽而不选择一致性Hash呢？

### 5.第五个要点：Redis集群的核心特点：去中心化

1）question：分布式集群设计需要考虑哪几个方面？

2）question：元数据存储的架构模式有哪些呢？

### 6.第六个要点：Redis怎么保持的元数据一致性

1）question：redis 如何进行数据分片的？

2）question：redis 如何管理元数据的一致性？

3）question：为什么Redis Cluster哈希槽数量是16384 (16K)？

### 7.问题总结：为什么Redis是使用哈希槽而不是一致性哈希呢?

## 20 DDD面试题：什么是领域驱动设计？什么是贫血模型？什么是充血模型？（大厂面烂了）

讲义地址： https://mp.weixin.qq.com/s/OT6LuwX1XyQ0n8P_FMzS-g

也可以参考《尼恩面试宝典 42专题 DDD 专题》

## 21 DDD面试题：外部接口调用，DDD应该放在哪一层？

讲义地址： https://mp.weixin.qq.com/s/604S55gbm7GJDKxTKpbJgA

也可以参考《尼恩面试宝典 42专题 DDD 专题》

## 22 DDD面试题：DDD聚合和表的对应关系是什么？（来自蚂蚁面试）

讲义地址： https://mp.weixin.qq.com/s/n_0NaMLTALYYrm8HIzDT5Q

也可以参考《尼恩面试宝典 42专题 DDD 专题》