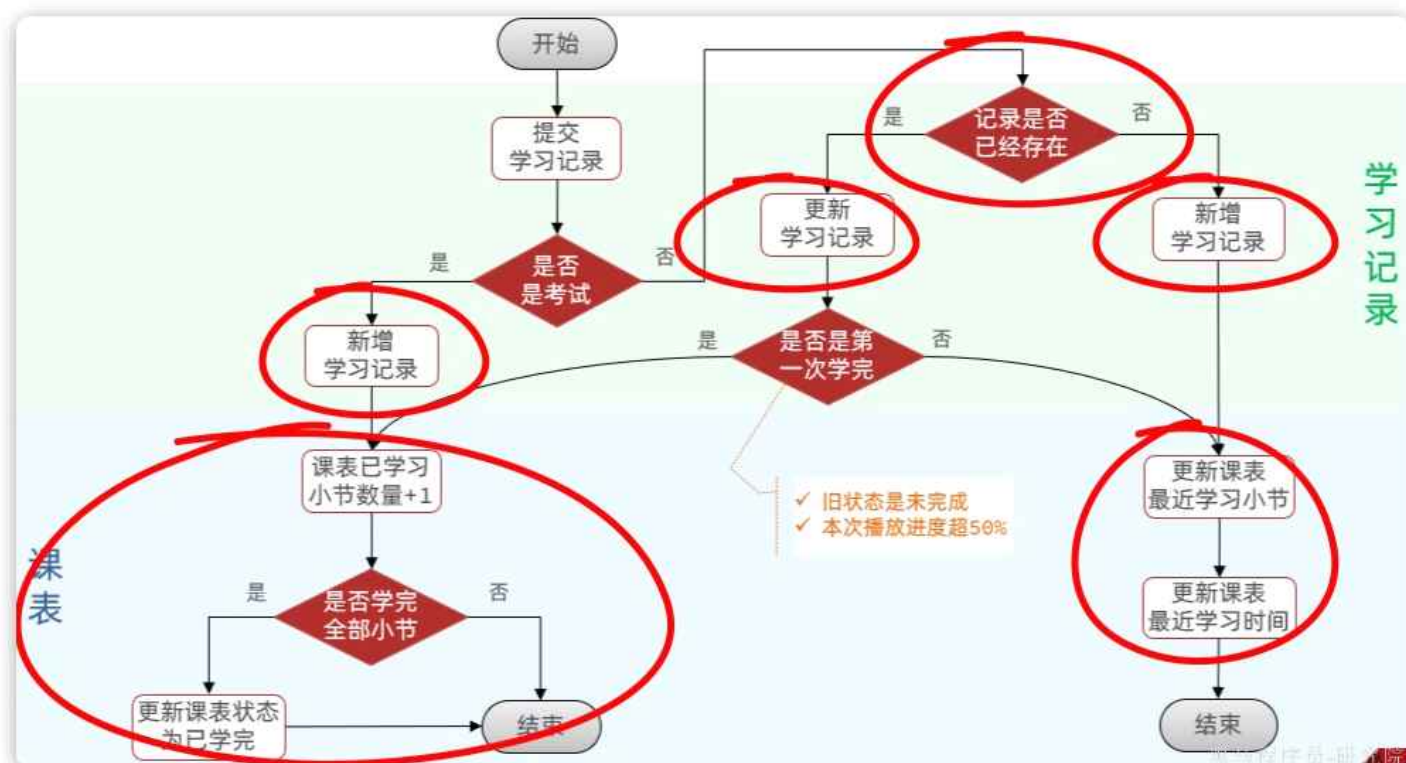


day04-高并发优化

昨天我们实现了学习计划和学习进度的统计功能。特别是学习进度部分，为了更精确的记录用户上一次播放的进度，我们采用的方案是：前端每隔15秒就发起一次请求，将播放记录写入数据库。

但问题是，提交播放记录的业务太复杂了，其中涉及到大量的数据库操作：



在并发较高的情况下，会给数据库带来非常大的压力。该怎么解决呢？

今天我们就来分析一下，当碰到高并发的数据库写业务时，该如何优化。通过今天的学习，大家可以掌握下面的技能：

- 理解高并发优化的常见方案
- 掌握Redis合并写请求的方案
- 掌握DelayQueue的使用

特别是其中的高并发优化方案，在很多的业务场景下都可以用到。

1.高并发优化方案

解决高并发问题从宏观角度来说有3个方向：

提高单机并发

尽可能减小业务接口的RT (ResponseTime) ,
提升单机性能和并发能力



水平扩展

将热点服务水平扩展，做好负载均衡，提高整个集群的并发能力



服务保护

做好服务熔断、降级保护措施，提高服务的高可用性



黑马程序员-研究院

其中，水平扩展和服务保护侧重的是运维层面的处理。而提高单机并发能力侧重的则是业务层面的处理，也就是我们程序员在开发时可以做到的。

因此，我们本章重点讨论如何通过编码来提供业务的单机并发能力。

1.1.单机并发能力

在机器性能一定的情况下，提高单机并发能力就是要尽可能缩短业务的响应时间

(ResponseTime)，而对响应时间影响最大的往往是对数据库的操作。而从数据库角度来说，我们的业务无非就是**读或写**两种类型。

对于读多写少的业务，其优化手段大家都比较熟悉了，主要包括两方面：

- 优化代码和SQL
- 添加缓存

对于写多读少的业务，大家可能较少碰到，优化的手段可能也不太熟悉，这也是我们要讲解的重点。

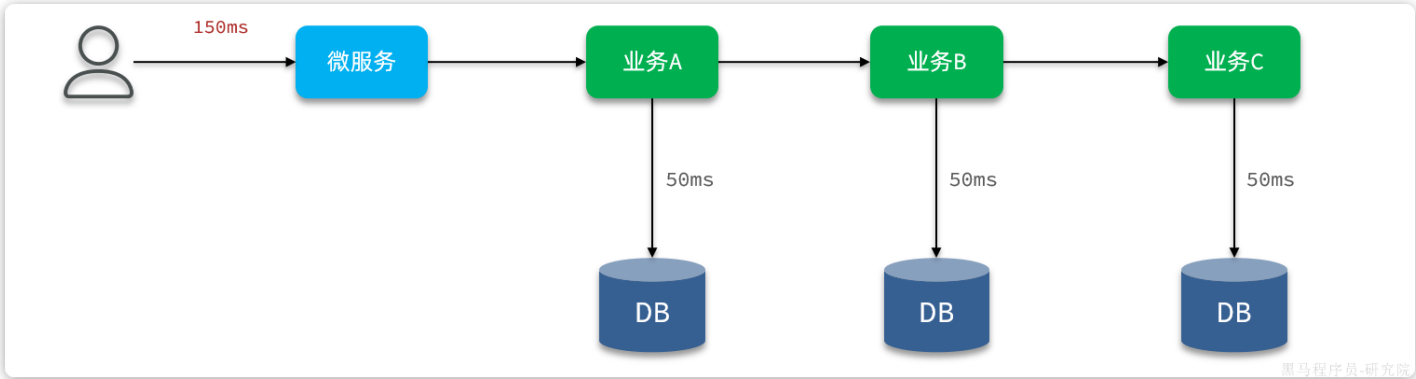
对于高并发写的优化方案有：

- 优化代码及SQL
- 变同步写为异步写
- 合并写请求

代码和SQL优化与读优化类似，我们就不再赘述了，接下来我们着重分析一下变同步为异步、合并写请求两种优化方案。

1.2.变同步为异步

假如一个业务比较复杂，需要有多次数据库的写业务，如图所示：

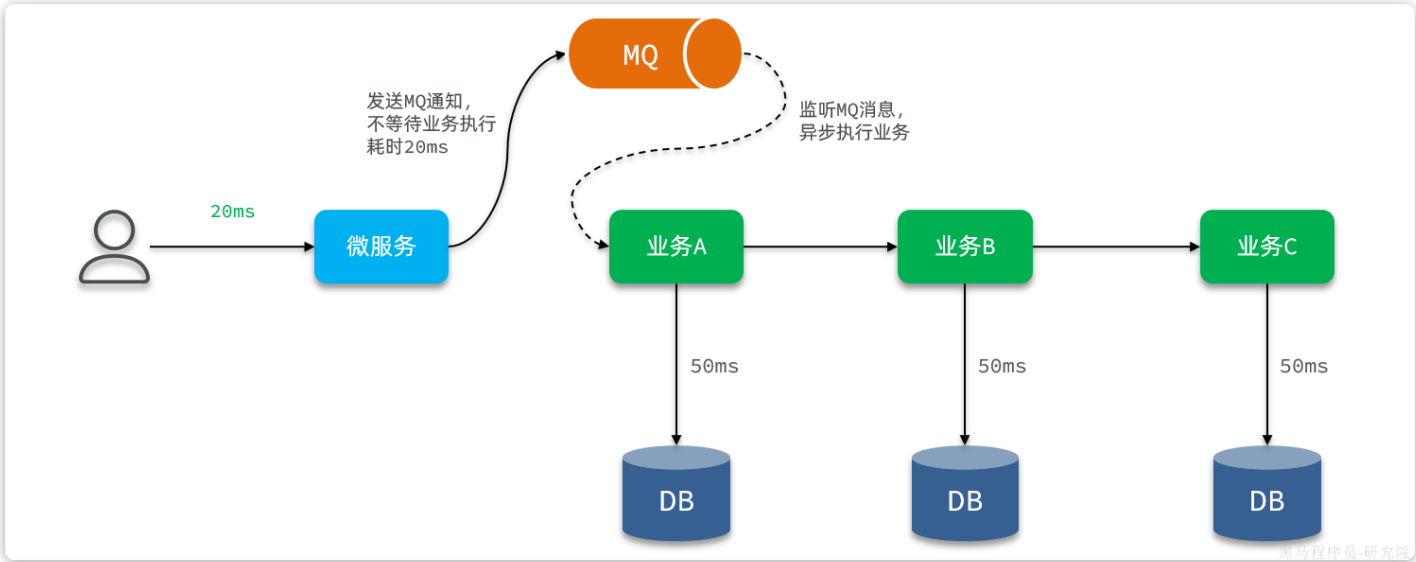


由于各个业务之间是同步串行执行，因此整个业务的响应时间就是每一次数据库写业务的响应时间之和，并发能力肯定不会太好。

优化的思路很简单，我们之前讲解MQ的时候就说过，利用MQ可以把同步业务变成异步，从而提高效率。

- 当我们接收到用户请求后，可以先不处理业务，而是发送MQ消息并返回给用户结果。
- 而后通过消息监听器监听MQ消息，处理后续业务。

如图：



这样一来，用户请求处理和后续数据库写就从同步变为异步，用户无需等待后续的数据库写操作，响应时间自然会大大缩短。并发能力自然大大提高。



优点：

- 无需等待复杂业务处理，大大减少响应时间
- 利用MQ暂存消息，起到流量削峰整形作用
- 降低写数据库频率，减轻数据库并发压力

缺点：

- 依赖于MQ的可靠性
- 降低了些频率，但是没有减少数据库写次数

应用场景：

- 比较适合应用于业务复杂，业务链较长，有多次数据库写操作的业务。

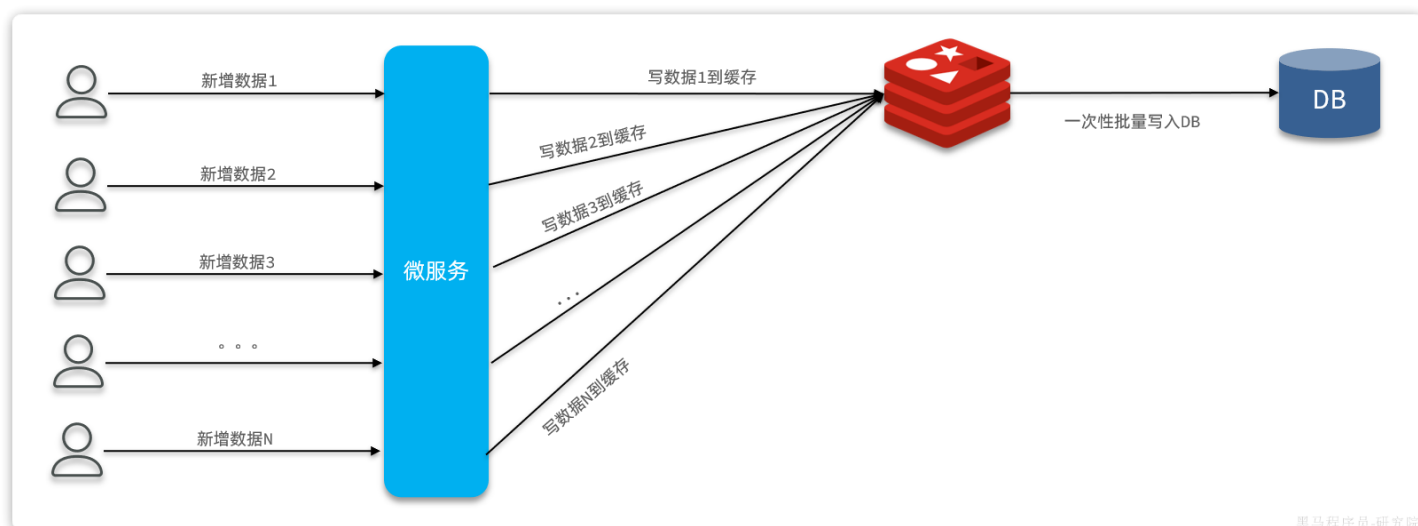
1.3.合并写请求

合并写请求方案其实是参考高并发读的优化思路：当读数据库并发较高时，我们可以把数据缓存到Redis，这样就无需访问数据库，大大减少数据库压力，减少响应时间。

既然读数据可以建立缓存，那么写数据可不可以也缓存到Redis呢？

答案是肯定的，合并写请求就是指当写数据库并发较高时，不再直接写到数据库。而是先将数据缓存到Redis，然后定期将缓存中的数据批量写入数据库。

如图：



由于Redis是内存操作，写的效率也非常高，这样每次请求的处理速度大大提高，响应时间大大缩短，并发能力肯定有很大的提升。

而且由于数据都缓存到Redis了，积累一些数据后再批量写入数据库，这样数据库的写频率、写次数都大大减少，对数据库压力小了非常多！



优点：

- 写缓存速度快，响应时间大大减少
- 降低数据库的写频率和写次数，大大减轻数据库压力

缺点：

- 实现相对复杂
- 依赖Redis可靠性
- 不支持事务和复杂业务

场景：

- 写频率较高、写业务相对简单的场景

2.播放进度记录方案改进

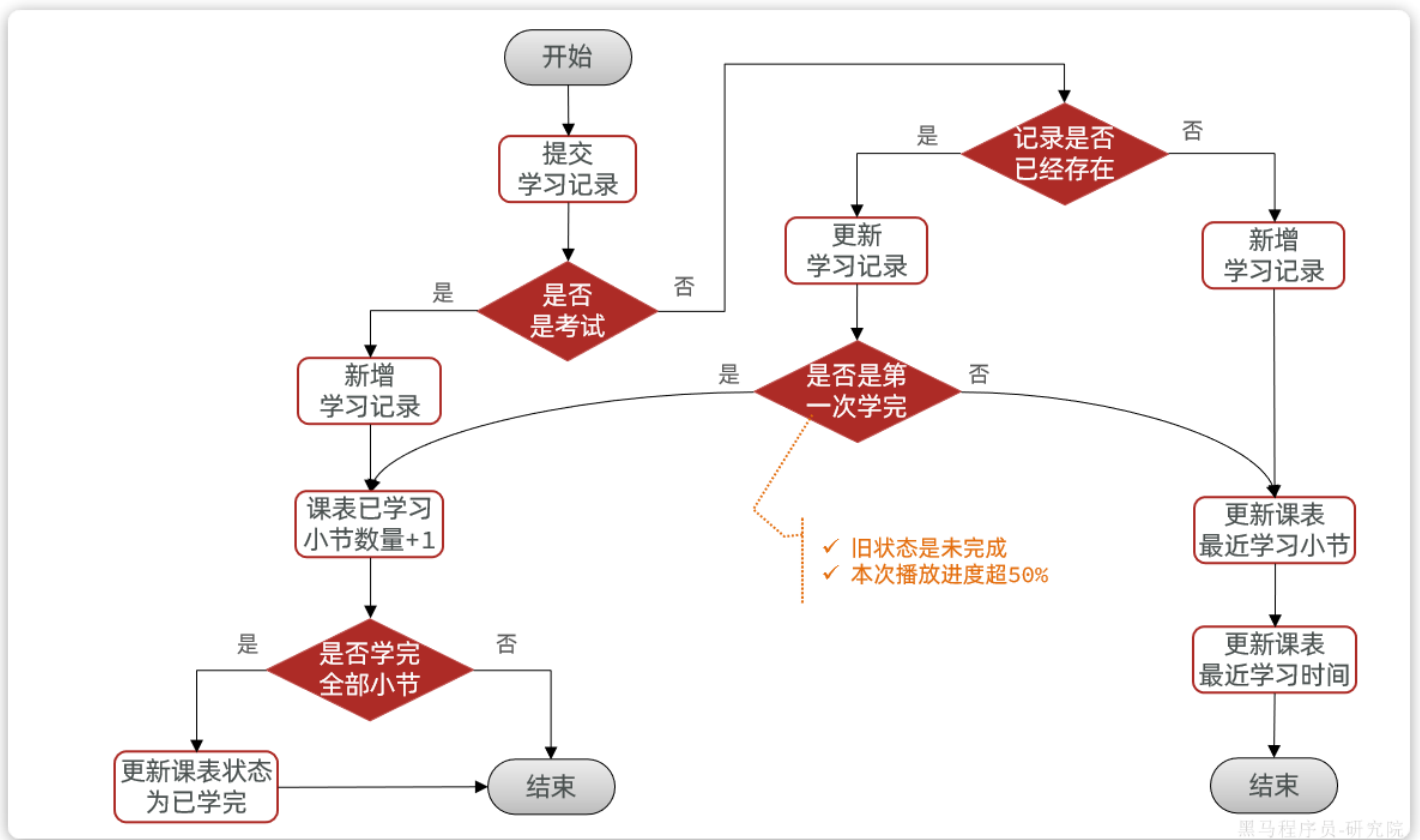
播放进度统计包含大量的数据库读、写操作。不过保存播放记录还是以写数据库为主。因此优化的方向还是以高并发写优化为主。

大家思考一下，针对播放进度记录业务来说，应该采用哪种优化方案呢？

- 变同步为异步？
- 合并写？

2.1.优化方案选择

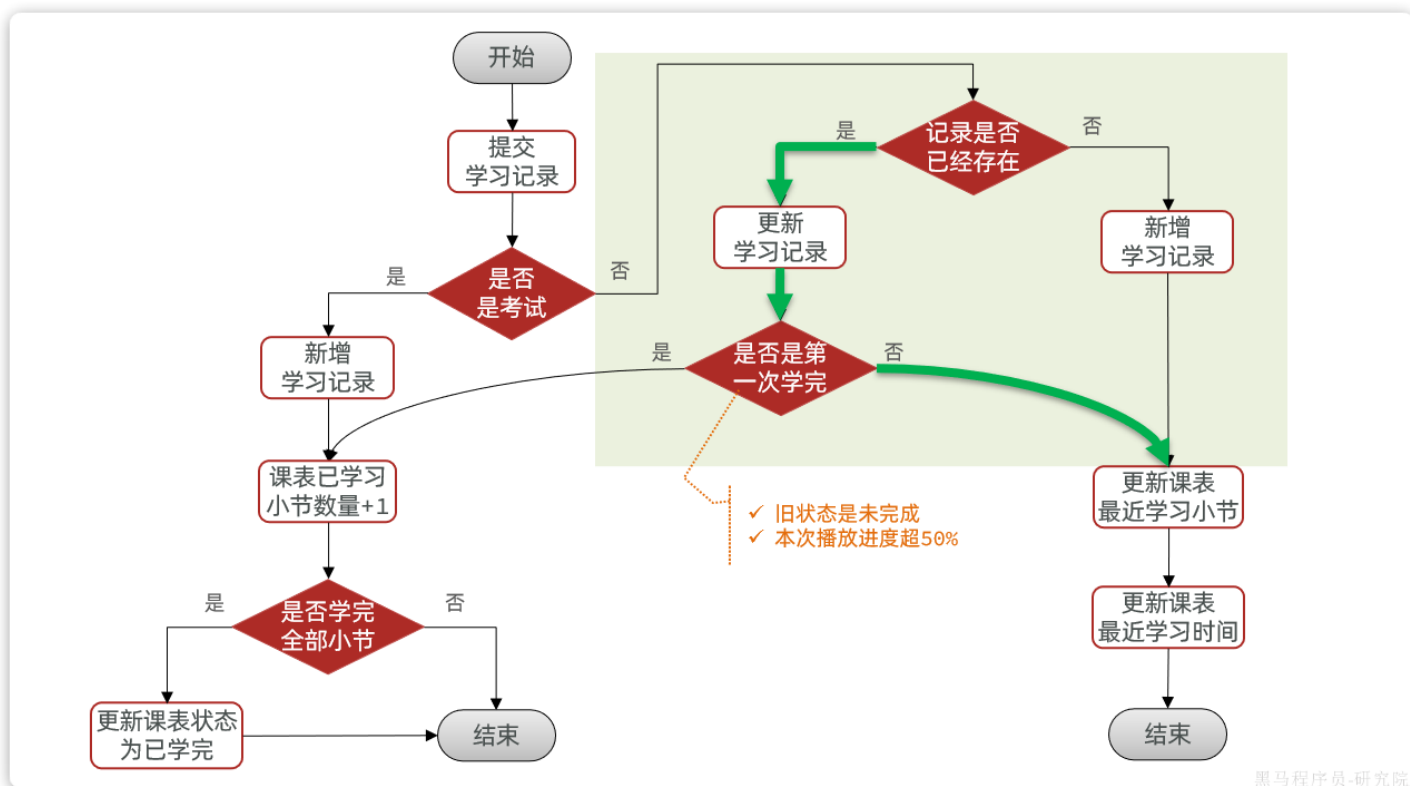
虽然播放进度记录业务较为复杂，但是我们认真思考一下整个业务分支：



黑马程序员-研究院

- 考试：每章只能考一次，还不能重复考试。因此属于低频行为，可以忽略
- 视频进度：前端每隔15秒就提交一次请求。在一个视频播放的过程中，可能有数十次请求，但完播（进度超50%）的请求只会有一次。因此多数情况下都是更新一下播放进度即可。

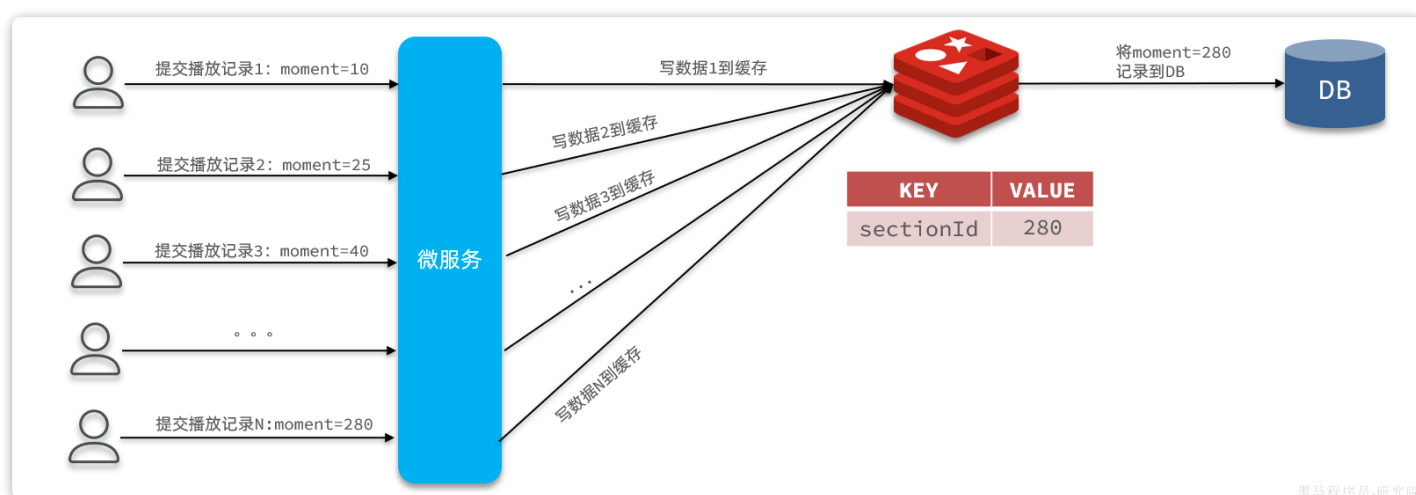
也就是说，95%的请求都是在更新 `learning_record` 表中的 `moment` 字段，以及 `learning_lesson` 表中的正在学习的小节id和时间。



黑马程序员-研究院

而播放进度信息，不管更新多少次，下一次续播肯定是从最后的一次播放进度开始续播。也就是说我们只需要记住最后一次即可。因此可以采用合并写方案来降低数据库写的次数和频率，而异步写做不到。

综上，提交播放进度业务虽然看起来复杂，但大多数请求的处理很简单，就是更新播放进度。并且播放进度数据是可以合并的（覆盖之前旧数据）。我们建议采用合并写请求方案：

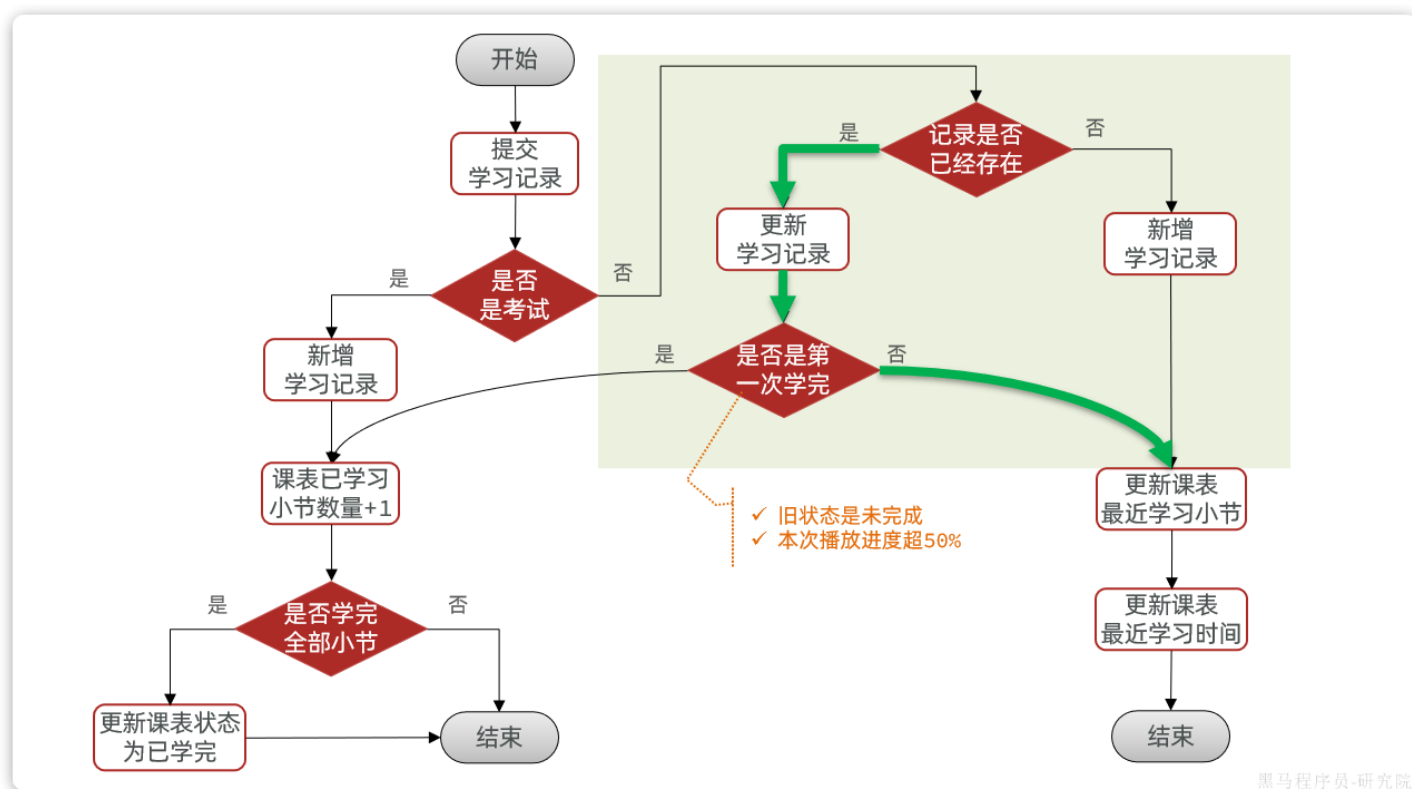


黑马程序员-研究院

2.2.Redis数据结构设计

我们先讨论下Redis缓存中需要记录哪些数据。

我们的优化方案要处理的不是所有的提交学习记录请求。仅仅是视频播放时的高频更新播放进度的请求，对应的业务分支如图：



这条业务支线的流程如下：

- 查询播放记录，判断是否存在
 - 如果不存在，新增一条记录
 - 如果存在，则更新学习记录
- 判断当前进度是否是第一次学完
 - 播放进度要超过50%
 - 原本的记录状态是未学完
- 更新课表中最近学习小节id、学习时间

这里有多次数据库操作，例如：

- 查询播放记录：需要知道播放记录是否存在、播放记录当前的完成状态
- 更新播放记录：更新播放进度
- 更新最近学习小节id、时间

一方面我们要缓存写数据，减少写数据库频率；另一方面我们要缓存播放记录，减少查询数据库。因此，缓存中至少要包含3个字段：

- 记录id: id, 用于根据id更新数据库
- 播放进度: moment, 用于缓存播放进度
- 播放状态 (是否学完) : finished, 用于判断是否是第一次学完

既然一个小节要保存多个字段，是不是可以考虑使用Hash结构来保存这些数据，如图：

KEY	FIELD	VALUE
sectionId	moment	126
	finished	false
	id	110

不过，这样设计有一个问题。课程有很多，每个课程的小节也非常多。每个小节都是一个独立的KEY，需要创建的KEY也会非常多，浪费大量内存。

而且，用户学习视频的过程中，可能会在多个视频之间来回跳转，这就会导致频繁的创作缓存、缓存过期，影响到最终的业务性能。该如何解决呢？

既然一个课程包含多个小节，我们完全可以把一个课程的多个小节作为一个KEY来缓存，如图：

KEY	HashKey	HashValue
lessonId	sectionId:1	<pre>1 { 2 "id": 1, 3 "moment": 242, 4 "finished": true 5 }</pre>
	sectionId:2	<pre>1 { 2 "id": 2, 3 "moment": 20, 4 "finished": false 5 }</pre>
	sectionId:3	

```

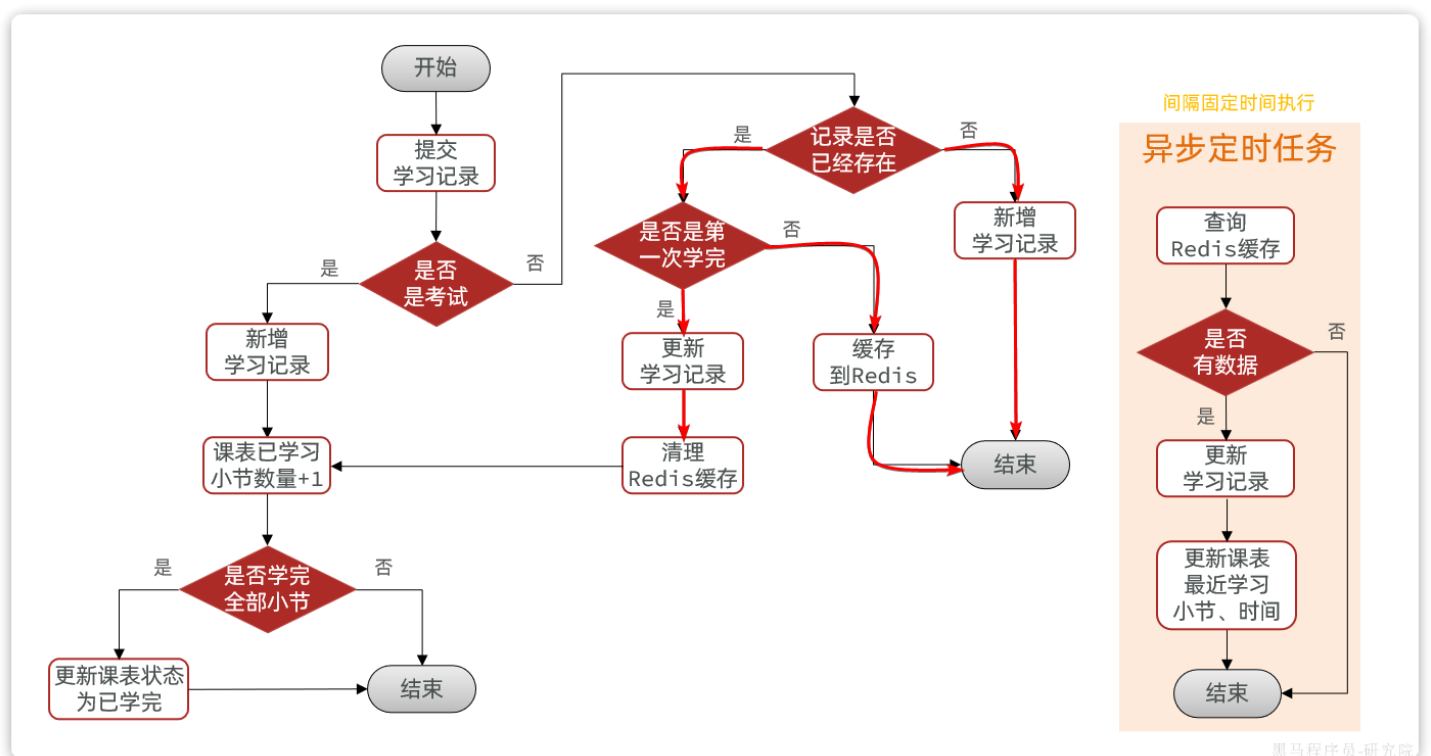
1 {
2   "id": 3,
3   "moment": 121,
4   "finished": false
5 }

```

这样做有两个好处：

- 可以大大减少需要创建的KEY的数量，减少内存占用。
- 一个课程创建一个缓存，当用户在多个视频间跳转时，整个缓存的有效期都会被延续，不会频繁的创建和销毁缓存数据

添加缓存以后，学习记录提交的业务流程就需要发生一些变化了，如图：



变化最大的有两点：

- 提交播放进度后，如果是更新播放进度则不写数据库，而是写缓存
- 需要一个定时任务，定期将缓存数据写入数据库

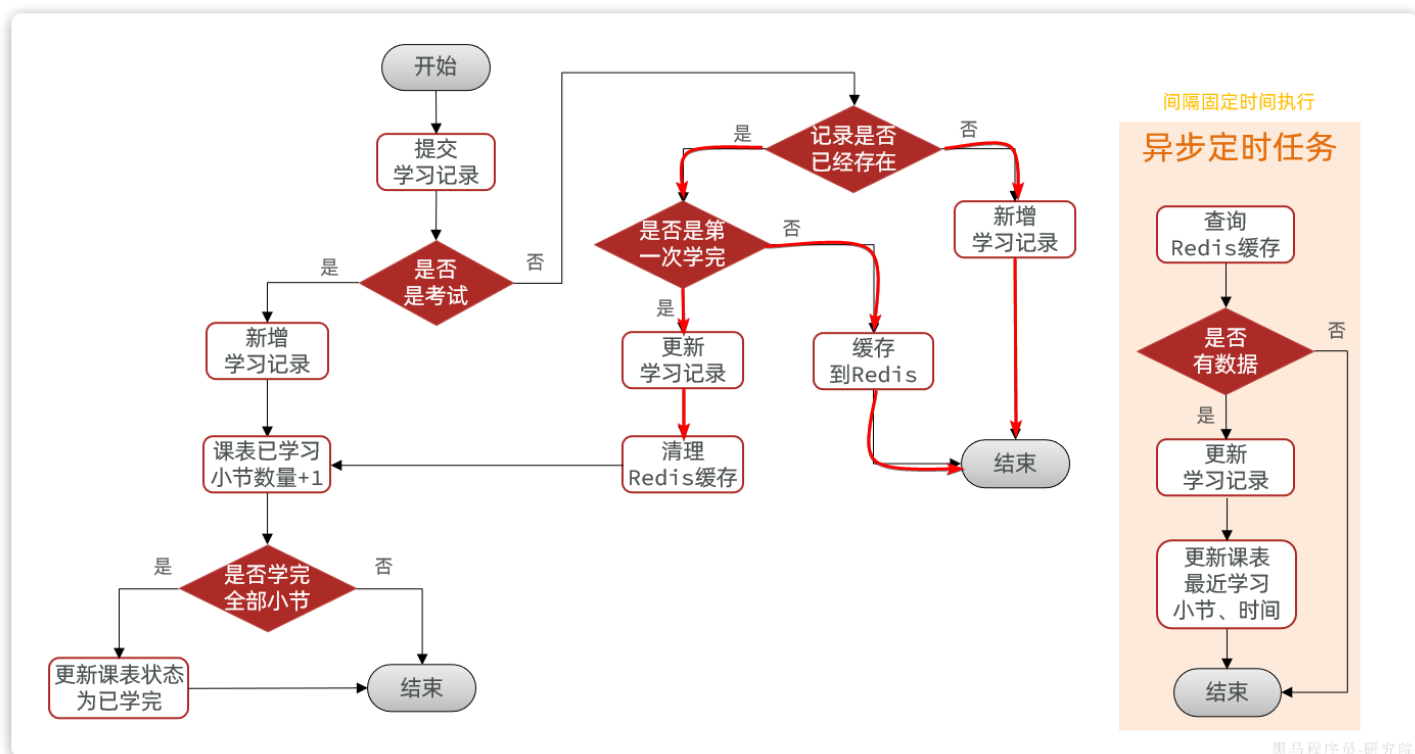
变化后的业务具体流程为：

- 1.提交学习记录
- 2.判断是否是考试

- 是：新增学习记录，并标记有小节被学完。走步骤8
- 否：走视频流程，步骤3
- 3.查询播放记录缓存，如果缓存不存在则查询数据库并建立缓存
- 4.判断记录是否存在
 - 4.1.否：新增一条学习记录
 - 4.2.是：走更新学习记录流程，步骤5
- 5.判断是否是第一次学完（进度超50%，旧的状态是未学完）
 - 5.1.否：仅仅是要更新播放进度，因此直接写入Redis并结束
 - 5.2.是：代表小节学完，走步骤6
- 6.更新学习记录状态为已学完
- 7.清理Redis缓存：因为学习状态变为已学完，与缓存不一致，因此这里清理掉缓存，这样下次查询时自然会更新缓存，保证数据一致。
- 8.更新课表中已学习小节的数量+1
- 9.判断课程的小节是否全部学完
 - 是：更新课表状态为已学完
 - 否：结束

2.3.持久化思路

对于合并写请求方案，一定有一个步骤就是持久化缓存数据到数据库。一般采用的是定时任务持久化：



但是定时任务的持久化方式在播放进度记录业务中存在一些问题，主要就是时效性问题。我们的产品要求视频续播的时间误差不能超过30秒。

- 假如定时任务间隔较短，例如20秒一次，对数据库的更新频率太高，压力太大
- 假如定时任务间隔较长，例如2分钟一次，更新频率较低，续播误差可能超过2分钟，不满足需求



注意：

如果产品对于时间误差要求不高，定时任务处理是最简单，最可靠的一种方案，推荐大家使用。

那么问题来了，有什么办法能够在不增加数据库压力的情况下，保证时间误差较低吗？

假如一个视频时长为20分钟，我们从头播放至15分钟关闭，每隔15秒提交一次播放进度，大概需要提交60次请求。

但是下一次我们再次打开该视频续播的时候，肯定是从最后一次提交的播放进度来续播。也就是说续播进度之前的N次播放进度都是没有意义的，都会被覆盖。

既然如此，我们完全没有必要定期把这些播放进度写到数据库，只需要将用户最后一次提交的播放进度写入数据库即可。

但问题来了，我们怎么知道哪一次提交是最后一次提交呢？

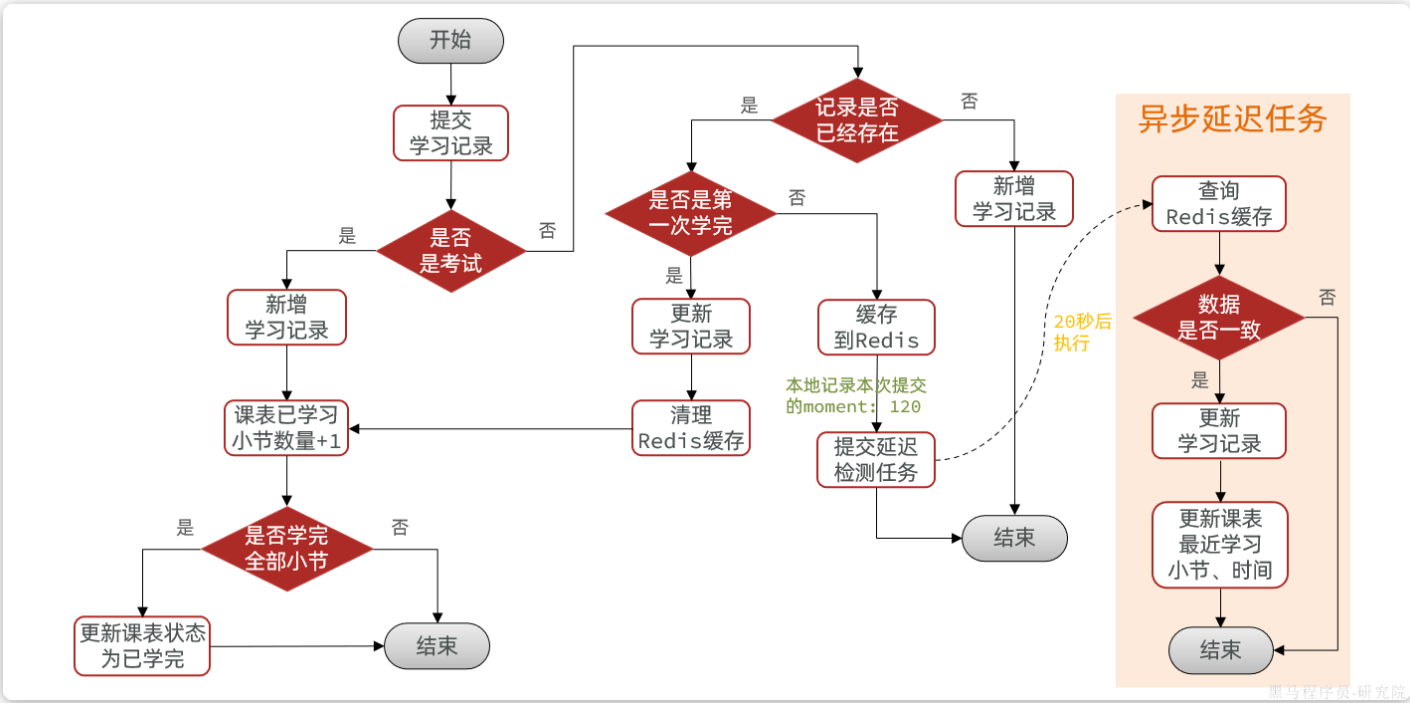
💡 只要用户一直在提交记录，Redis中的播放进度就会一直变化。如果Redis中的播放进度不变，肯定是停止了播放，是最后一次提交。

因此，我们只要能判断Redis中的播放进度是否变化即可。怎么判断呢？

每当前端提交播放记录时，我们可以设置一个延迟任务并保存这次提交的进度。等待20秒后（因为前端每15秒提交一次，20秒就是等待下一次提交），检查Redis中的缓存的进度与任务中的进度是否一致。

- 不一致：说明持续在提交，无需处理
- 一致：说明是最后一次提交，更新学习记录、更新课表最近学习小节和时间到数据库中

流程如下：



3.延迟任务

为了确定用户提交的播放记录是否变化，我们需要将播放记录保存为一个延迟任务，等待超过一个提交周期（20s）后检查播放进度。

那么延迟任务该如何实现呢？

3.1.延迟任务方案

延迟任务的实现方案有很多，常见的有四类：

	DelayQueue	Redisson	MQ	时间轮
原理	JDK自带延迟队列，基于阻塞队列实现。	基于Redis数据结构模拟JDK的DelayQueue实现	利用MQ的特性。例如RabbitMQ的死信队列	时间轮算法
优点	<ul style="list-style-type: none">不依赖第三方服务	<ul style="list-style-type: none">分布式系统下可用不占用JVM内存	<ul style="list-style-type: none">分布式系统下可以不占用JVM内存	<ul style="list-style-type: none">不依赖第三方服务性能优异
缺点	<ul style="list-style-type: none">占用JVM内存只能单机使用	<ul style="list-style-type: none">依赖第三方服务	<ul style="list-style-type: none">依赖第三方服务	<ul style="list-style-type: none">只能单机使用

以上四种方案都可以解决问题，不过本例中我们会使用DelayQueue方案。因为这种方案使用成本最低，而且不依赖任何第三方服务，减少了网络交互。

但缺点也很明显，就是需要占用JVM内存，在数据量非常大的情况下可能会有问题。但考虑到任务存储时间比较短（只有20秒），因此也可以接收。

如果你们的数据量非常大，DelayQueue不能满足业务需求，大家也可以替换为其它延迟队列方式，例如Redisson、MQ等

3.2.DelayQueue的原理

首先来看一下DelayQueue的源码：

```
1 public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
2     implements BlockingQueue<E> {
3
4     private final transient ReentrantLock lock = new ReentrantLock();
5     private final PriorityQueue<E> q = new PriorityQueue<E>();
6
```

```
7 // ... 略
8 }
```

可以看到DelayQueue实现了BlockingQueue接口，是一个阻塞队列。队列就是容器，用来存储东西的。DelayQueue叫做延迟队列，其中存储的就是**延迟执行的任务**。

我们可以看到DelayQueue的泛型定义：

```
1 DelayQueue<E extends Delayed>
```

这说明存入 DelayQueue 内部的元素必须是 Delayed 类型，这其实就是一个延迟任务的规范接口。来看一下：

```
1 public interface Delayed extends Comparable<Delayed> {
2
3     /**
4      * Returns the remaining delay associated with this object, in the
5      * given time unit.
6      *
7      * @param unit the time unit
8      * @return the remaining delay; zero or negative values indicate
9      *         that the delay has already elapsed
10     */
11     long getDelay(TimeUnit unit);
12 }
```

从源码中可以看出，Delayed类型必须具备两个方法：

- `getDelay()`：获取延迟任务的剩余延迟时间
- `compareTo(T t)`：比较两个延迟任务的延迟时间，判断执行顺序

可见，Delayed类型的延迟任务具备两个功能：获取剩余延迟时间、比较执行顺序。当然，我们可以对Delayed做实现和功能扩展，比如添加延迟任务的数据。

将来每一次提交播放记录，就可以将播放记录保存在这样的一个 Delayed 类型的延迟任务里并设定20秒的延迟时间。然后交给 DelayQueue 队列。 DelayQueue 会调用 compareTo 方法，根据剩

余延迟时间对任务排序。剩余延迟时间越短的越靠近队首，这样就会被优先执行。

3.3.DelayQueue的用法

首先定义一个Delayed类型的延迟任务类，要能保持任务数据。

```
1 package com.tianji.learning.utils;
2
3 import lombok.Data;
4
5 import java.time.Duration;
6 import java.util.concurrent.Delayed;
7 import java.util.concurrent.TimeUnit;
8
9 @Data
10 public class DelayTask<D> implements Delayed {
11     private D data;
12     private long deadlineNanos;
13
14     public DelayTask(D data, Duration delayTime) {
15         this.data = data;
16         this.deadlineNanos = System.nanoTime() + delayTime.toNanos();
17     }
18
19     @Override
20     public long getDelay(TimeUnit unit) {
21         return unit.convert(Math.max(0, deadlineNanos - System.nanoTime()),
22             TimeUnit.NANOSECONDS);
23     }
24
25     @Override
26     public int compareTo(Delayed o) {
27         long l = getDelay(TimeUnit.NANOSECONDS) -
28             o.getDelay(TimeUnit.NANOSECONDS);
29         if(l > 0){
30             return 1;
31         }else if(l < 0){
32             return -1;
33         }else {
34             return 0;
35         }
36     }
37 }
```


接下来就可以创建延迟任务，交给延迟队列保存：

```
1 package com.tianji.learning.utils;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.junit.jupiter.api.Test;
5
6 import java.time.Duration;
7 import java.util.concurrent.DelayQueue;
8
9 @Slf4j
10 class DelayTaskTest {
11     @Test
12     void testDelayQueue() throws InterruptedException {
13         // 1. 初始化延迟队列
14         DelayQueue<DelayTask<String>> queue = new DelayQueue<>();
15         // 2. 向队列中添加延迟执行的任务
16         log.info("开始初始化延迟任务。。。");
17         queue.add(new DelayTask<>("延迟任务3", Duration.ofSeconds(3)));
18         queue.add(new DelayTask<>("延迟任务1", Duration.ofSeconds(1)));
19         queue.add(new DelayTask<>("延迟任务2", Duration.ofSeconds(2)));
20         // TODO 3. 尝试执行任务
21
22     }
23 }
```

最后，补上执行任务的代码：

```
1 package com.tianji.learning.utils;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.junit.jupiter.api.Test;
5
6 import java.time.Duration;
7 import java.util.concurrent.DelayQueue;
8
9 @Slf4j
10 class DelayTaskTest {
11     @Test
12     void testDelayQueue() throws InterruptedException {
13         // 1. 初始化延迟队列
14         DelayQueue<DelayTask<String>> queue = new DelayQueue<>();
15         // 2. 向队列中添加延迟执行的任务
```

```
16     log.info("开始初始化延迟任务。。。");
17     queue.add(new DelayTask<>("延迟任务3", Duration.ofSeconds(3)));
18     queue.add(new DelayTask<>("延迟任务1", Duration.ofSeconds(1)));
19     queue.add(new DelayTask<>("延迟任务2", Duration.ofSeconds(2)));
20     // 3. 尝试执行任务
21     while (true) {
22         DelayTask<String> task = queue.take();
23         log.info("开始执行延迟任务: {}", task.getData());
24     }
25 }
26 }
```



注意：

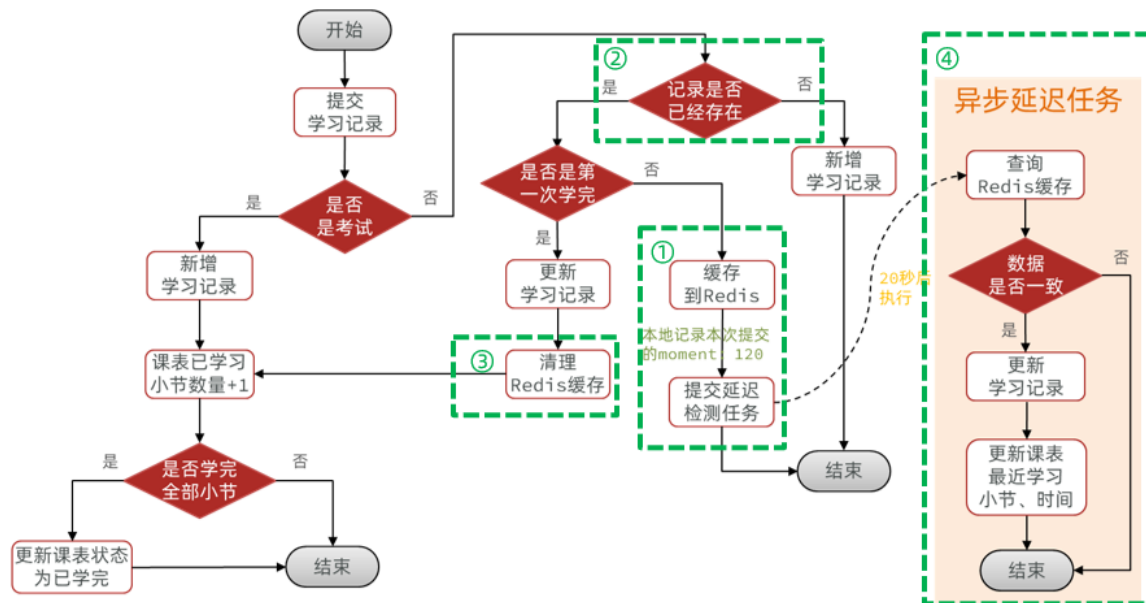
这里我们是直接同一个线程来执行任务了。当没有任务的时候线程会被阻塞。而在实际开发中，我们会准备线程池，开启多个线程来执行队列中的任务。

4. 代码改造

接下来，我们就可以按照之前分析的方案来改造代码了。

4.1. 定义延迟任务工具类

首先，我们要定义一个工具类，帮助我们改造整个业务。在提交学习记录业务中，需要用到异步任务和缓存的地方有以下几处：



黑马程序员-研究院

因此，我们的工具类就应该具备上述4个方法：

- ① 添加播放记录到Redis，并添加一个延迟检测任务到DelayQueue
- ② 查询Redis缓存中的指定小节的播放记录
- ③ 删除Redis缓存中的指定小节的播放记录
- ④ 异步执行DelayQueue中的延迟检测任务，检测播放进度是否变化，如果无变化则写入数据库

工具类代码如下：

```

1 package com.tianji.learning.utils;
2
3 import com.tianji.common.utils.JsonUtils;
4 import com.tianji.common.utils.StringUtils;
5 import com.tianji.learning.domain.po.LearningLesson;
6 import com.tianji.learning.domain.po.LearningRecord;
7 import com.tianji.learning.mapper.LearningRecordMapper;
8 import com.tianji.learning.service.ILearningLessonService;
9 import lombok.Data;
10 import lombok.NoArgsConstructor;
11 import lombok.RequiredArgsConstructor;
12 import lombok.extern.slf4j.Slf4j;
13 import org.springframework.data.redis.core.StringRedisTemplate;
14 import org.springframework.stereotype.Component;
15
16 import javax.annotation.PostConstruct;
17 import javax.annotation.PreDestroy;
18 import java.time.Duration;
19 import java.time.LocalDateTime;
  
```

```

20 import java.util.Objects;
21 import java.util.concurrent.CompletableFuture;
22 import java.util.concurrent.DelayQueue;
23
24 @Slf4j
25 @Component
26 @RequiredArgsConstructor
27 public class LearningRecordDelayTaskHandler {
28
29     private final StringRedisTemplate redisTemplate;
30     private final LearningRecordMapper recordMapper;
31     private final ILearningLessonService lessonService;
32     private final DelayQueue<DelayTask<RecordTaskData>> queue = new
DelayQueue<>();
33     private final static String RECORD_KEY_TEMPLATE = "learning:record:{";
34     private static volatile boolean begin = true;
35
36     @PostConstruct
37     public void init(){
38         CompletableFuture.runAsync(this::handleDelayTask);
39     }
40     @PreDestroy
41     public void destroy(){
42         begin = false;
43         log.debug("延迟任务停止执行! ");
44     }
45
46     public void handleDelayTask(){
47         while (begin) {
48             try {
49                 // 1. 获取到期的延迟任务
50                 DelayTask<RecordTaskData> task = queue.take();
51                 RecordTaskData data = task.getData();
52                 // 2. 查询Redis缓存
53                 LearningRecord record = readRecordCache(data.getLessonId(),
data.getSectionId());
54                 if (record == null) {
55                     continue;
56                 }
57                 // 3. 比较数据, moment值
58                 if(!Objects.equals(data.getMoment(), record.getMoment())) {
59                     // 不一致, 说明用户还在持续提交播放进度, 放弃旧数据
60                     continue;
61                 }
62
63                 // 4. 一致, 持久化播放进度数据到数据库
64                 // 4.1. 更新学习记录的moment

```

```

65         record.setFinished(null);
66         recordMapper.updateById(record);
67         // 4.2.更新课表最近学习信息
68         LearningLesson lesson = new LearningLesson();
69         lesson.setId(data.getLessonId());
70         lesson.setLatestSectionId(data.getSectionId());
71         lesson.setLatestLearnTime(LocalDateTime.now());
72         lessonService.updateById(lesson);
73     } catch (Exception e) {
74         log.error("处理延迟任务发生异常", e);
75     }
76 }
77 }
78
79 public void addLearningRecordTask(LearningRecord record){
80     // 1.添加数据到Redis缓存
81     writeRecordCache(record);
82     // 2.提交延迟任务到延迟队列 DelayQueue
83     queue.add(new DelayTask<>(new RecordTaskData(record),
Duration.ofSeconds(20)));
84 }
85
86 public void writeRecordCache(LearningRecord record) {
87     log.debug("更新学习记录的缓存数据");
88     try {
89         // 1.数据转换
90         String json = JsonUtils.toJsonStr(new RecordCacheData(record));
91         // 2.写入Redis
92         String key = StringUtils.format(RECORD_KEY_TEMPLATE,
record.getLessonId());
93         redisTemplate.opsForHash().put(key,
record.getSectionId().toString(), json);
94         // 3.添加缓存过期时间
95         redisTemplate.expire(key, Duration.ofMinutes(1));
96     } catch (Exception e) {
97         log.error("更新学习记录缓存异常", e);
98     }
99 }
100
101 public LearningRecord readRecordCache(Long lessonId, Long sectionId){
102     try {
103         // 1.读取Redis数据
104         String key = StringUtils.format(RECORD_KEY_TEMPLATE, lessonId);
105         Object cacheData = redisTemplate.opsForHash().get(key,
sectionId.toString());
106         if (cacheData == null) {
107             return null;

```

```

108         }
109         // 2. 数据检查和转换
110         return JsonUtils.toBean(cacheData.toString(),
LearningRecord.class);
111     } catch (Exception e) {
112         log.error("缓存读取异常", e);
113         return null;
114     }
115 }
116
117 public void cleanRecordCache(Long lessonId, Long sectionId){
118     // 删除数据
119     String key = StringUtils.format(RECORD_KEY_TEMPLATE, lessonId);
120     redisTemplate.opsForHash().delete(key, sectionId.toString());
121 }
122
123 @Data
124 @NoArgsConstructor
125 private static class RecordCacheData{
126     private Long id;
127     private Integer moment;
128     private Boolean finished;
129
130     public RecordCacheData(LearningRecord record) {
131         this.id = record.getId();
132         this.moment = record.getMoment();
133         this.finished = record.getFinished();
134     }
135 }
136 @Data
137 @NoArgsConstructor
138 private static class RecordTaskData{
139     private Long lessonId;
140     private Long sectionId;
141     private Integer moment;
142
143     public RecordTaskData(LearningRecord record) {
144         this.lessonId = record.getLessonId();
145         this.sectionId = record.getSectionId();
146         this.moment = record.getMoment();
147     }
148 }
149 }

```

4.2.改造提交学习记录功能

接下来，改造提交学习记录的功能：

```
1 package com.tianji.learning.service.impl;
2
3 import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
4 import com.tianji.api.client.course.CourseClient;
5 import com.tianji.api.dto.course.CourseFullInfoDTO;
6 import com.tianji.api.dto.leanring.LearningLessonDTO;
7 import com.tianji.api.dto.leanring.LearningRecordDTO;
8 import com.tianji.common.exceptions.BizIllegalException;
9 import com.tianji.common.exceptions.DbException;
10 import com.tianji.common.utils.BeanUtils;
11 import com.tianji.common.utils.UserContext;
12 import com.tianji.learning.domain.dto.LearningRecordFormDTO;
13 import com.tianji.learning.domain.po.LearningLesson;
14 import com.tianji.learning.domain.po.LearningRecord;
15 import com.tianji.learning.enums.LessonStatus;
16 import com.tianji.learning.enums.SectionType;
17 import com.tianji.learning.mapper.LearningRecordMapper;
18 import com.tianji.learning.service.ILearningLessonService;
19 import com.tianji.learning.service.ILearningRecordService;
20 import com.tianji.learning.utils.LearningRecordDelayTaskHandler;
21 import lombok.RequiredArgsConstructor;
22 import org.springframework.stereotype.Service;
23 import org.springframework.transaction.annotation.Transactional;
24
25 import java.time.LocalDateTime;
26 import java.util.List;
27
28 /**
29  * <p>
30  * 学习记录表 服务实现类
31  * </p>
32  *
33  * @author 虎哥
34  * @since 2022-12-10
35  */
36 @Service
37 @RequiredArgsConstructor
38 public class LearningRecordServiceImpl extends
    ServiceImpl<LearningRecordMapper, LearningRecord> implements
    ILearningRecordService {
39
40     private final ILearningLessonService lessonService;
```

```
41
42     private final CourseClient courseClient;
43
44     private final LearningRecordDelayTaskHandler taskHandler;
45
46     @Override
47     public LearningLessonDTO queryLearningRecordByCourse(Long courseId) {
48         // 1.获取登录用户
49         Long userId = UserContext.getUser();
50         // 2.查询课表
51         LearningLesson lesson = lessonService.queryByUserAndCourseId(userId,
courseId);
52         // 3.查询学习记录
53         // select * from xx where lesson_id = #{lessonId}
54         List<LearningRecord> records =
lambdaQuery().eq(LearningRecord::getLessonId, lesson.getId()).list();
55         // 4.封装结果
56         LearningLessonDTO dto = new LearningLessonDTO();
57         dto.setId(lesson.getId());
58         dto.setLatestSectionId(lesson.getLatestSectionId());
59         dto.setRecords(BeanUtils.copyList(records, LearningRecordDTO.class));
60         return dto;
61     }
62
63     @Override
64     @Transactional
65     public void addLearningRecord(LearningRecordFormDTO recordDTO) {
66         // 1.获取登录用户
67         Long userId = UserContext.getUser();
68         // 2.处理学习记录
69         boolean finished = false;
70         if (recordDTO.getSectionType() == SectionType.VIDEO) {
71             // 2.1.处理视频
72             finished = handleVideoRecord(userId, recordDTO);
73         } else {
74             // 2.2.处理考试
75             finished = handleExamRecord(userId, recordDTO);
76         }
77         if (!finished) {
78             // 没有新学完的小节，无需更新课表中的学习进度
79             return;
80         }
81         // 3.处理课表数据
82         handleLearningLessonsChanges(recordDTO);
83     }
84
```



```

85     private void handleLearningLessonsChanges(LearningRecordFormDTO recordDTO)
86     {
87         // 1.查询课表
88         LearningLesson lesson = lessonService.getById(recordDTO.getLessonId());
89         if (lesson == null) {
90             throw new BizIllegalException("课程不存在，无法更新数据！");
91         }
92         // 2.判断是否有新的完成小节
93         boolean allLearned = false;
94         // 3.如果有新完成的小节，则需要查询课程数据
95         CourseFullInfoDTO cInfo =
96         courseClient.getCourseInfoById(lesson.getCourseId(), false, false);
97         if (cInfo == null) {
98             throw new BizIllegalException("课程不存在，无法更新数据！");
99         }
100        // 4.比较课程是否全部学完：已学习小节 >= 课程总小节
101        allLearned = lesson.getLearnedSections() + 1 >= cInfo.getSectionNum();
102        // 5.更新课表
103        lessonService.lambdaUpdate()
104            .set(lesson.getLearnedSections() == 0,
105                LearningLesson::getStatus, LessonStatus.LEARNING.getValue())
106            .set(allLearned, LearningLesson::getStatus,
107                LessonStatus.FINISHED.getValue())
108            .set(allLearned, LearningLesson::getFinishTime,
109                LocalDateTime.now())
110            .setSql("learned_sections = learned_sections + 1")
111            .eq(LearningLesson::getId, lesson.getId())
112            .update();
113    }
114
115    private boolean handleVideoRecord(Long userId, LearningRecordFormDTO
116    recordDTO) {
117        // 1.查询旧的学习记录
118        LearningRecord old = queryOldRecord(recordDTO.getLessonId(),
119        recordDTO.getSectionId());
120        // 2.判断是否存在
121        if (old == null) {
122            // 3.不存在，则新增
123            // 3.1.转换PO
124            LearningRecord record = BeanUtils.copyBean(recordDTO,
125                LearningRecord.class);
126            // 3.2.填充数据
127            record.setUserId(userId);
128            // 3.3.写入数据库
129            boolean success = save(record);
130        }
131    }

```

```
124         if (!success) {
125             throw new DbException("新增学习记录失败! ");
126         }
127         return false;
128     }
129     // 4.存在, 则更新
130     // 4.1.判断是否是第一次完成
131     boolean finished = !old.getFinished() && recordDTO.getMoment() * 2 >=
recordDTO.getDuration();
132     if (!finished) {
133         LearningRecord record = new LearningRecord();
134         record.setLessonId(recordDTO.getLessonId());
135         record.setSectionId(recordDTO.getSectionId());
136         record.setMoment(recordDTO.getMoment());
137         record.setId(old.getId());
138         record.setFinished(old.getFinished());
139         taskHandler.addLearningRecordTask(record);
140         return false;
141     }
142     // 4.2.更新数据
143     boolean success = lambdaUpdate()
144         .set(LearningRecord::getMoment, recordDTO.getMoment())
145         .set(LearningRecord::getFinished, true)
146         .set(LearningRecord::getFinishTime, recordDTO.getCommitTime())
147         .eq(LearningRecord::getId, old.getId())
148         .update();
149     if (!success) {
150         throw new DbException("更新学习记录失败! ");
151     }
152     // 4.3.清理缓存
153     taskHandler.cleanRecordCache(recordDTO.getLessonId(),
recordDTO.getSectionId());
154     return true;
155 }
156
157 private LearningRecord queryOldRecord(Long lessonId, Long sectionId) {
158     // 1.查询缓存
159     LearningRecord record = taskHandler.readRecordCache(lessonId,
sectionId);
160     // 2.如果命中, 直接返回
161     if (record != null) {
162         return record;
163     }
164     // 3.未命中, 查询数据库
165     record = lambdaQuery()
166         .eq(LearningRecord::getLessonId, lessonId)
167         .eq(LearningRecord::getSectionId, sectionId)
```

```
168         .one();
169         // 4. 写入缓存
170         taskHandler.writeRecordCache(record);
171         return record;
172     }
173
174     private boolean handleExamRecord(Long userId, LearningRecordFormDTO
recordDTO) {
175         // 1. 转换DTO为PO
176         LearningRecord record = BeanUtils.copyBean(recordDTO,
LearningRecord.class);
177         // 2. 填充数据
178         record.setUserId(userId);
179         record.setFinished(true);
180         record.setFinishTime(recordDTO.getCommitTime());
181         // 3. 写入数据库
182         boolean success = save(record);
183         if (!success) {
184             throw new DbException("新增考试记录失败!");
185         }
186         return true;
187     }
188 }
```

5.练习

5.1.线程池的使用

目前我们的延迟任务执行还是单线程模式，大家将其改造为线程池模式，核心线程数与CPU核数一致即可。

5.2.定时任务方案

课堂中我们讲解了基于延迟任务的持久化方案，但定时任务方案也是非常常用的一种。大家可以尝试利用定时任务的方式来解决数据持久化问题。

5.3.预习


参考产品原型中与课程互动问答有关的功能：

<https://lanhuapp.com/web/#/item/project/product?tid=b688242e-152e-4c39-8737-575cdc992579&pid=4c3fbd53-c67d-4...>

蓝湖

正在加载 正在加载


还有：

 <https://lanhuapp.com/web/#/item/project/product?tid=b688242e-152e-4c39-8737-575cdc992579&pid=4c3fbd53-c67...>

蓝湖


正在加载 正在加载

以及后台管理页面：

 <https://lanhuapp.com/web/#/item/project/product?tid=b688242e-152e-4c39-8737-575cdc992579&pid=ab7fe2ae-92...>

蓝湖

正在加载 正在加载

 <https://lanhuapp.com/web/#/item/project/product?tid=b688242e-152e-4c39-8737-575cdc992579&pid=ab7fe2ae-92...>

蓝湖


正在加载 正在加载

思考一下两个问题：

- 互动问答相关接口可能有哪些？
- 互动问答的数据库表该如何设计？

6.面试

面试官：你在开发中参与了哪些功能开发让你觉得比较有挑战性？

 答：我参与了整个学习中心的功能开发，其中有很多的学习辅助功能都很有特色。比如视频播放的进度记录。我们网站的课程是以录播视频为主，为了提高用户的学习体验，需要实现视频续播功能。这个功能本身并不复杂，只不过我们产品提出的要求比较高：

- 首先续播时间误差要控制在30秒以内。
- 而且要做到用户突然断开，甚至切换设备后，都可以继续上一次播放

要达成这个目的，使用传统的手段显然是不行的。

首先，要做到切换设备后还能续播，用户的播放进度必须保存在服务端，而不是客户端。

其次，用户突然断开或者切换设备，续播的时间误差不能超过30秒，那播放进度的记录频率就需要比较高。我们会在前端每隔15秒就发起一次心跳请求，提交最新的播放进度，记录到服务端。这样用户下一次续播时直接读取服务端的播放进度，就可以将时间误差控制在15秒左右。

面试官：那播放进度在服务端保存在哪里呢？是数据库吗？如果是数据库，如何解决高频写入给数据库带来巨大压力？



答：

提交播放记录最终肯定是要保存到数据库中的。因为我们不仅要去做视频续播，还有用户学习计划、学习进度统计等功能，都需要用到用户的播放记录数据。

但确实如你所说，前端每隔15秒一次请求，如果在用户量较大时，直接全部写入数据库，对数据库压力会比较大。因此我们采用了合并写请求的方案，当用户提交播放进度时会先缓存在Redis中，后续再将数据保存到数据库即可。

由于播放进度会不断覆盖，只保留最后一次即可。这样就可以大大减少对于数据库的访问次数和访问频率了。