我用这11招，让接口性能提升了100倍

前言

> 苏三的免费刷题网站：[www.susan.net.cn](http://cxyroad.com/”http://www.susan.net.cn”) 里面：面试八股文、BAT面试真题、工作内推、工作经验分享、技术专栏等等什么都有，欢迎收藏和转发。

接口性能优化对于从事后端开发的同学来说，肯定再熟悉不过了，因为它是一个跟开发语言无关的公共问题。

该问题说简单也简单，说复杂也复杂。

有时候，只需加个索引就能解决问题。

有时候，需要做代码重构。

有时候，需要增加缓存。

有时候，需要引入一些中间件，比如mq。

有时候，需要需要分库分表。

有时候，需要拆分服务。

等等。。。

导致接口性能问题的原因千奇百怪，不同的项目不同的接口，原因可能也不一样。

本文我总结了一些行之有效的，优化接口性能的办法，给有需要的朋友一个参考。

1.索引

接口性能优化大家第一个想到的可能是：`优化索引`。

没错，优化索引的成本是最小的。

你通过查看线上日志或者监控报告，查到某个接口用到的某条sql语句耗时比较长。

这时你可能会有下面这些疑问：

1. 该sql语句加索引了没?
2. 加的索引生效了没?
3. mysql选错索引了没?

1.1 没加索引

sql语句中`where`条件的关键字段，或者`order by`后面的排序字段，忘了加索引，这个问题在项目中很常见。

项目刚开始的时候，由于表中的数据量小，加不加索引sql查询性能差别不大。

后来，随着业务的发展，表中数据量越来越多，就不得不加索引了。

可以通过命令：

```
show index from `order`;
```

能单独查看某张表的索引情况。

也可以通过命令：

```
show create table `order`;
```

查看整张表的建表语句，里面同样会显示索引情况。

通过`ALTER TABLE`命令可以添加索引：

```
ALTER TABLE `order` ADD INDEX idx_name (name);
```

也可以通过`CREATE INDEX`命令添加索引：

```
CREATE INDEX idx_name ON `order` (name);
```

不过这里有一个需要注意的地方是：想通过命令修改索引，是不行的。

目前在mysql中如果想要修改索引，只能先删除索引，再重新添加新的。

删除索引可以用`DROP INDEX`命令：

```
ALTER TABLE `order` DROP INDEX idx_name;
```

用`DROP INDEX`命令也行：

```
DROP INDEX idx_name ON `order`;
```

1.2 索引没生效

通过上面的命令我们已经能够确认索引是有的，但它生效了没？此时你内心或许会冒出这样一个疑问。

那么，如何查看索引有没有生效呢?

答：可以使用`explain`命令，查看mysql的执行计划，它会显示索引的使用情况。

例如：

```
explain select * from `order` where code='002';
```

结果：(https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/42e055f082b740c195f53031b639b2d9~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=kKRZu4D2YVl4CMz3czF%2BHARejb8%3D)通过这几列可以判断索引使用情况，执行计划包含列的含义如下图所示：(https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/81a4c9efe48749bfab9a83496bd0b7cb~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=Dzh49%2FHjgiwNCljSxurMj2Y0JQo%3D)如果你想进一步了解explain的详细用法，可以看看我的另一篇文章《[explain | 索引优化的这把绝世好剑，你真的会用吗？](http://cxyroad.com/"https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490262&idx=1&sn=a67f610afa984ecca130a54a3be453ab&chksm=c0ebc23ef79c4b2869dea998e413c5cbea6aeeea01ee74efc7c1a5fc228baa7beca215adf3ea&token=751314179&lang=zh_CN&scene=21#wechat_redirect")》

说实话，sql语句没有走索引，排除没有建索引之外，最大的可能性是索引失效了。

下面说说索引失效的常见原因：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/ceddeae15780475595840964471d890d3~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=793MfISUdSs1WjXn6x%2Fm8GCV%2FZ8%3D)如果不是上面的这些原因，则需要再进一步排查一下其他原因。

1.3 选错索引

此外，你有没有遇到过这样一种情况：明明是同一条sql，只有入参不同而已。有的时候走的索引a，有的时候却走的索引b?

没错，有时候mysql会选错索引。

必要时可以使用`force index`来强制查询sql走某个索引。

至于为什么mysql会选错索引，后面有专门的文章介绍的，这里先留点悬念。

2. sql优化

如果优化了索引之后，也没啥效果。

接下来试着优化一下sql语句，因为它的改造成本相对于java代码来说也要小得多。

下面给大家列举了sql优化的15个小技巧：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/68c0156d8fd54e2eba70b70a42e51054~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=wfWyeG9Kpg6rZSxYi9OjBgxaOck%3D)由于这些技巧在我之前的文章中已经详细介绍过了，在这里我就不深入了。

更详细的内容，可以看我的另一篇文章《[聊聊sql优化的15个小技巧](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490473&idx=1&sn=ba07d64aa8c89054bd263efd660d3a3f&chksm=c0ebc341f79c4a579a6f41465df0459a696ab8ef0499a5f6fd76269af1064fde0286afdb027d&token=2041133408&lang=zh_CN&scene=21#wechat_redirect”)》，相信看完你会有很多收获。

3. 远程调用

很多时候，我们需要在某个接口中，调用其他服务的接口。

比如有这样的业务场景：

在用户信息查询接口中需要返回：用户名称、性别、等级、头像、积分、成长值等信息。

而用户名称、性别、等级、头像在用户服务中，积分在积分服务中，成长值在成长值服务中。为了汇总这些数据统一返回，需要另外提供一个对外接口服务。

于是，用户信息查询接口需要调用用户查询接口、积分查询接口 和 成长值查询接口，然后汇总数据统一返回。

调用过程如下图所示：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/b270632983b446358d7df23b42c748e9~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=g2cAqAlz8y2W5gl4%2BHD9ThlBqzA%3D)调用远程接口总耗时530ms = 200ms + 150ms + 180ms

显然这种串行调用远程接口性能是非常不好的，调用远程接口总的耗时为所有的远程接口耗时之和。

那么如何优化远程接口性能呢?

3.1 并行调用

上面说到，既然串行调用多个远程接口性能很差，为什么不改成并行呢?

如下图所示：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/456ceb1f1aad46479b1b556b4d443dd9~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=%2FVlF9JeY4ajiluG741bD8foHFoQ%3D)调用远程接口总耗时200ms = 200ms（即耗时最长的那次远程接口调用）

在java8之前可以通过实现`Callable`接口，获取线程返回结果。

java8以后通过`CompleteFuture`类实现该功能。我们这里以CompleteFuture为例：

```
public UserInfo getUserInfo(Long id) throws InterruptedException, ExecutionException {
    final UserInfo userInfo = new UserInfo();
    CompletableFuture userFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteUserAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);

    CompletableFuture bonusFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteBonusAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);

    CompletableFuture growthFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteGrowthAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);
    CompletableFuture.allOf(userFuture, bonusFuture, growthFuture).join();

    userFuture.get();
    bonusFuture.get();
    growthFuture.get();
```

```
    return userInfo;
}
```

> 温馨提醒一下，这两种方式别忘了使用线程池。示例中我用到了executor，表示自定义的线程池，为了防止高并发场景下，出现线程过多的问题。

3.2 数据异构

上面说到的用户信息查询接口需要调用用户查询接口、积分查询接口 和 成长值查询接口，然后汇总数据统一返回。

那么，我们能不能把数据冗余一下，把用户信息、积分和成长值的数据统一存储到一个地方，比如：redis，存的数据结构就是用户信息查询接口所需要的内容。然后通过用户id，直接从redis中查询数据出来，不就OK了？

如果在高并发的场景下，为了提升接口性能，远程接口调用大概率会被去掉，而改成保存冗余数据的数据异构方案。

但需要注意的是，如果使用了数据异构方案，就可能会出现数据一致性问题。

用户信息、积分和成长值有更新的话，大部分情况下，会先更新到数据库，然后同步到redis。但这种跨库的操作，可能会导致两边数据不一致的情况产生。

4. 重复调用

`重复调用`在我们的日常工作代码中可以说随处可见，但如果没有控制好，会非常影响接口的性能。

不信，我们一起看看。

4.1 循环查数据库

有时候，我们需要从指定的用户集合中，查询出有哪些是在数据库中已经存在的。

实现代码可以这样写：

```
public List<User> queryUser(List<User> searchList) {
    if (CollectionUtils.isEmpty(searchList)) {
        return Collections.emptyList();
    }

    List<User> result = Lists.newArrayList();
    searchList.forEach(user -
> result.add(userMapper.getUserById(user.getId())));
    return result;
}
```

这里如果有50个用户，则需要循环50次，去查询数据库。我们都知道，每查询一次数据库，就是一次远程调用。

如果查询50次数据库，就有50次远程调用，这是非常耗时的操作。

那么，我们如何优化呢?

具体代码如下：

```
public List<User> queryUser(List<User> searchList) {
    if (CollectionUtils.isEmpty(searchList)) {
        return Collections.emptyList();
    }
    List<Long> ids = searchList.stream().map(User::getId).collect(Collector
s.toList());
```

```
    return userMapper.getUserByIds(ids);
}
```

提供一个根据用户id集合批量查询用户的接口，只远程调用一次，就能查询出所有的数据。

> 这里有个需要注意的地方是：id集合的大小要做限制，最好一次不要请求太多的数据。要根据实际情况而定，建议控制每次请求的记录条数在500以内。

4.2 死循环

有些小伙伴看到这个标题，可能会感到有点意外，死循环也算？

代码中不是应该避免死循环吗？为啥还是会产生死循环？

有时候死循环是我们自己写的，例如下面这段代码：

```
while(true) {
    if(condition) {
        break;
    }
    System.out.println(”do samething”);
}
```

这里使用了while(true)的循环调用，这种写法在`CAS自旋锁`中使用比较多。

当满足condition等于true的时候，则自动退出该循环。

如果condition条件非常复杂，一旦出现判断不正确，或者少写了一些逻辑判断，就可能在某些场景下出现死循环的问题。

出现死循环，大概率是开发人员人为的bug导致的，不过这种情况很容易被测出

来。

> 还有一种隐藏的比较深的死循环，是由于代码写的不太严谨导致的。如果用正常数据，可能测不出问题，但一旦出现异常数据，就会立即出现死循环。

4.3 无限递归

如果想要打印某个分类的所有父分类，可以用类似这样的递归方法实现：

```
public void printCategory(Category category) {
  if(category == null
      || category.getParentId() == null) {
     return;
  }
  System.out.println("父分类名称："+ category.getName());
  Category parent = categoryMapper.getCategoryById(category.getParentId());
  printCategory(parent);
}
```

正常情况下，这段代码是没有问题的。

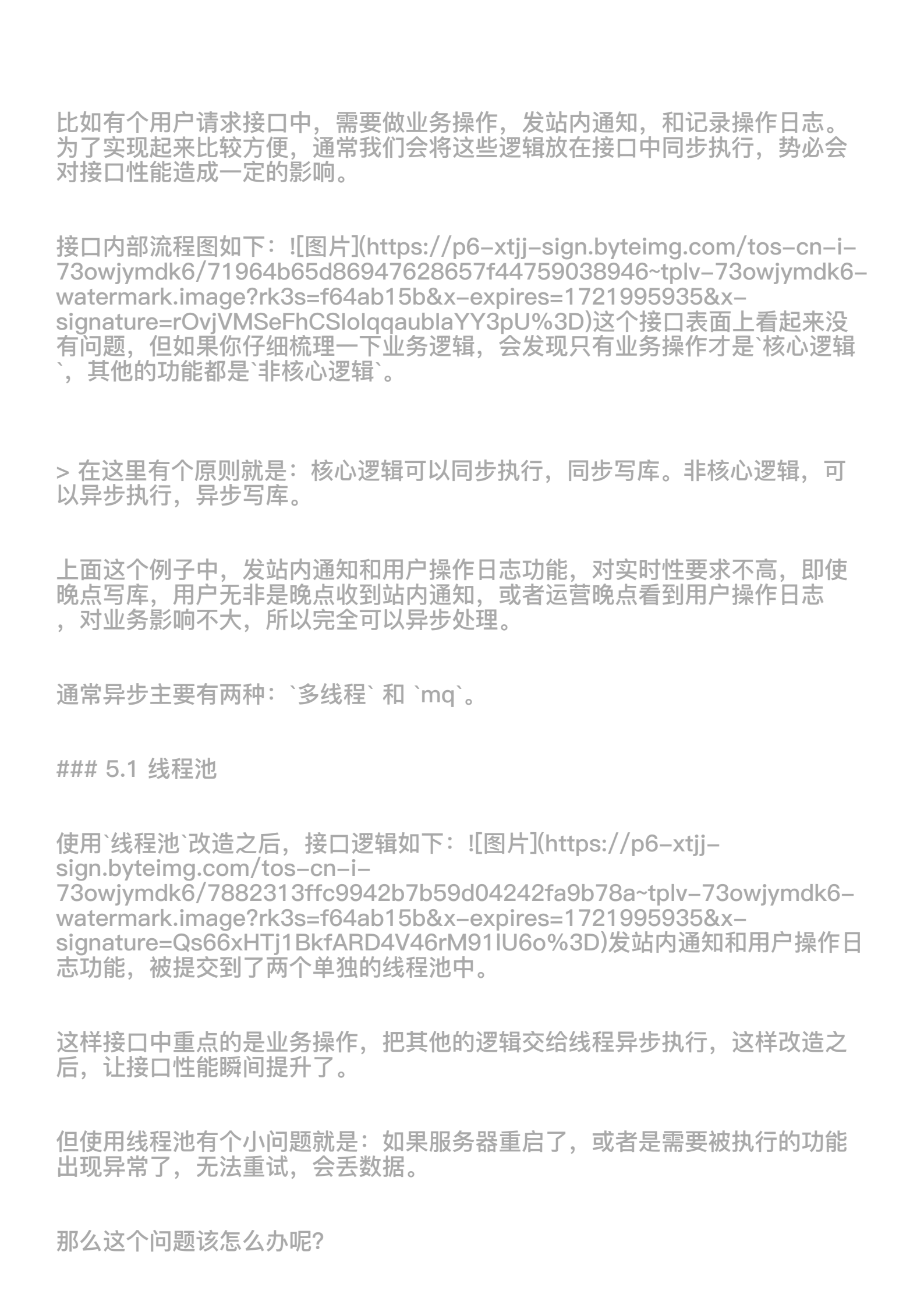
但如果某次有人误操作，把某个分类的parentId指向了它自己，这样就会出现无限递归的情况。导致接口一直不能返回数据，最终会发生堆栈溢出。

> 建议写递归方法时，设定一个递归的深度，比如：分类最大等级有4级，则深度可以设置为4。然后在递归方法中做判断，如果深度大于4时，则自动返回，这样就能避免无限循环的情况。

5. 异步处理

有时候，我们接口性能优化，需要重新梳理一下业务逻辑，看看是否有设计上不太合理的地方。

比如有个用户请求接口中，需要做业务操作，发站内通知，和记录操作日志。为了实现起来比较方便，通常我们会将这些逻辑放在接口中同步执行，势必会对接口性能造成一定的影响。

接口内部流程图如下：  (https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/71964b65d86947628657f44759038946~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=rOvjVMSeFhCSIolqqaublaYY3pU%3D)这个接口表面上看起来没有问题，但如果你仔细梳理一下业务逻辑，会发现只有业务操作才是`核心逻辑`，其他的功能都是`非核心逻辑`。

> 在这里有个原则就是：核心逻辑可以同步执行，同步写库。非核心逻辑，可以异步执行，异步写库。

上面这个例子中，发站内通知和用户操作日志功能，对实时性要求不高，即使晚点写库，用户无非是晚点收到站内通知，或者运营晚点看到用户操作日志，对业务影响不大，所以完全可以异步处理。

通常异步主要有两种：`多线程` 和 `mq`。

5.1 线程池

使用`线程池`改造之后，接口逻辑如下：(https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/7882313ffc9942b7b59d04242fa9b78a~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=Qs66xHTj1BkfARD4V46rM91lU6o%3D)发站内通知和用户操作日志功能，被提交到了两个单独的线程池中。

这样接口中重点的是业务操作，把其他的逻辑交给线程异步执行，这样改造之后，让接口性能瞬间提升了。

但使用线程池有个小问题就是：如果服务器重启了，或者是需要被执行的功能出现异常了，无法重试，会丢数据。

那么这个问题该怎么办呢?

5.2 mq

使用`mq`改造之后，接口逻辑如下：(https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/01322e927f4b452fbcef472561236293~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=arFYZ%2BHz1lb2XYUeUWqp3hd7qWU%3D)对于发站内通知和用户操作日志功能，在接口中并没真正实现，它只发送了mq消息到mq服务器。然后由mq消费者消费消息时，才真正的执行这两个功能。

这样改造之后，接口性能同样提升了，因为发送mq消息速度是很快的，我们只需业务操作的代码即可。

6. 避免大事务

很多小伙伴在使用spring框架开发项目时，为了方便，喜欢使用`@Transactional`注解提供事务功能。

没错，使用@Transactional注解这种声明式事务的方式提供事务功能，确实能少写很多代码，提升开发效率。

但也容易造成大事务，引发其他的问题。

下面用一张图看看大事务引发的问题。(https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/18db3c8c8a27416ebb4c1f2eec2b8b19~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=fyGYS5kuOHrUsPlkrcTrWWeNFwE%3D)从图中能够看出，大事务问题可能会造成接口超时，对接口的性能有直接的影响。

我们该如何优化大事务呢?

1. 少用@Transactional注解
2. 将查询(select)方法放到事务外
3. 事务中避免远程调用
4. 事务中避免一次性处理太多数据

5. 有些功能可以非事务执行
6. 有些功能可以异步处理

关于大事务问题我的另一篇文章《[让人头痛的大事务问题到底要如何解决？](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490259&idx=1&sn=1dd11c5f49103ca303a61fc82ce406e0&chksm=c0ebc23bf79c4b2db58b28ef752560bd91a1932ceb6713c9b19b821db0f29e1c58275d334076&token=2041133408&lang=zh_CN&scene=21#wechat_redirect”)》，它里面做了非常详细的介绍，如果大家感兴趣可以看看。

7. 锁粒度

在某些业务场景中，为了防止多个线程并发修改某个共享数据，造成数据异常。

为了解决并发场景下，多个线程同时修改数据，造成数据不一致的情况。通常情况下，我们会：`加锁`。

但如果锁加得不好，导致锁的粒度太粗，也会非常影响接口性能。

7.1 synchronized

在java中提供了`synchronized`关键字给我们的代码加锁。

通常有两种写法：`在方法上加锁` 和 `在代码块上加锁`。

先看看如何在方法上加锁：

```
public synchronized doSave(String fileUrl) {
    mkdir();
    uploadFile(fileUrl);
    sendMessage(fileUrl);
}
```

```

这里加锁的目的是为了防止并发的情况下，创建了相同的目录，第二次会创建失败，影响业务功能。

但这种直接在方法上加锁，锁的粒度有点粗。因为doSave方法中的上传文件和发消息方法，是不需要加锁的。只有创建目录方法，才需要加锁。

我们都知道文件上传操作是非常耗时的，如果将整个方法加锁，那么需要等到整个方法执行完之后才能释放锁。显然，这会导致该方法的性能很差，变得得不偿失。

这时，我们可以改成在代码块上加锁了，具体代码如下：

```
public void doSave(String path,String fileUrl) {
    synchronized(this) {
      if(!exists(path)) {
          mkdir(path);
       }
    }
    uploadFile(fileUrl);
    sendMessage(fileUrl);
}
```

这样改造之后，锁的粒度一下子变小了，只有并发创建目录功能才加了锁。而创建目录是一个非常快的操作，即使加锁对接口的性能影响也不大。

最重要的是，其他的上传文件和发送消息功能，任然可以并发执行。

当然，这种做在单机版的服务中，是没有问题的。但现在部署的生产环境，为了保证服务的稳定性，一般情况下，同一个服务会被部署在多个节点中。如果哪天挂了一个节点，其他的节点服务任然可用。

多节点部署避免了因为某个节点挂了，导致服务不可用的情况。同时也能分摊整个系统的流量，避免系统压力过大。
```

同时它也带来了新的问题：synchronized只能保证一个节点加锁是有效的，但如果有多个节点如何加锁呢?

答：这就需要使用：`分布式锁`了。目前主流的分布式锁包括：redis分布式锁、zookeeper分布式锁 和 数据库分布式锁。

由于zookeeper分布式锁的性能不太好，真实业务场景用的不多，这里先不讲。

下面聊一下redis分布式锁。

7.2 redis分布式锁

在分布式系统中，由于redis分布式锁相对于更简单和高效，成为了分布式锁的首先，被我们用到了很多实际业务场景当中。

使用redis分布式锁的伪代码如下：

```
public void doSave(String path,String fileUrl) {
  try {
    String result = jedis.set(lockKey, requestId, "NX", "PX", expireTime);
    if ("OK".equals(result)) {
      if(!exists(path)) {
        mkdir(path);
        uploadFile(fileUrl);
        sendMessage(fileUrl);
      }
      return true;
    }
  } finally{
      unlock(lockKey,requestId);
  }
  return false;
}
```

跟之前使用`synchronized`关键字加锁时一样，这里锁的范围也太大了，换句话说就是锁的粒度太粗，这样会导致整个方法的执行效率很低。

其实只有创建目录的时候，才需要加分布式锁，其余代码根本不用加锁。

于是，我们需要优化一下代码：

```
public void doSave(String path,String fileUrl) {
   if(this.tryLock()) {
      mkdir(path);
   }
   uploadFile(fileUrl);
   sendMessage(fileUrl);
}

private boolean tryLock() {
    try {
    String result = jedis.set(lockKey, requestId, ”NX”, ”PX”, expireTime);
    if (”OK”.equals(result)) {
      return true;
    }
  } finally{
      unlock(lockKey,requestId);
  }
  return false;
}
```

上面代码将加锁的范围缩小了，只有创建目录时才加了锁。这样看似简单的优化之后，接口性能能提升很多。说不定，会有意外的惊喜喔。哈哈哈。

redis分布式锁虽说好用，但它在使用时，有很多注意的细节，隐藏了很多坑，如果稍不注意很容易踩中。详细内容可以看看我的另一篇文章《[聊聊redis分布式锁的8大坑](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490430&idx=1&sn=a1f42f9a981a8f161941a6472f317b10&chksm=c0ebc396f79c4a801a330917ca700e7d7a6af3a3c2c5a4e11a05770da925de8aa9ed3c277737&token=2041133408&lang=zh_CN&scene=21#wechat_redirect”)》

7.3 数据库分布式锁

mysql数据库中主要有三种锁：

* 表锁：加锁快，不会出现死锁。但锁定粒度大，发生锁冲突的概率最高，并发度最低。
* 行锁：加锁慢，会出现死锁。但锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
* 间隙锁：开销和加锁时间界于表锁和行锁之间。它会出现死锁，锁定粒度界于表锁和行锁之间，并发度一般。

并发度越高，意味着接口性能越好。

所以数据库锁的优化方向是：

优先使用`行锁`，其次使用`间隙锁`，再其次使用`表锁`。

8.分页处理

有时候我会调用某个接口批量查询数据，比如：通过用户id批量查询出用户信息，然后给这些用户送积分。

但如果你一次性查询的用户数量太多了，比如一次查询2000个用户的数据。参数中传入了2000个用户的id，远程调用接口，会发现该用户查询接口经常超时。

调用代码如下：

```
List<User> users = remoteCallUser(ids);
```

众所周知，调用接口从数据库获取数据，是需要经过网络传输的。如果数据量太大，无论是获取数据的速度，还是网络传输受限于带宽，都会导致耗时时间比较长。

那么，这种情况要如何优化呢?

答：`分页处理`。

将一次获取所有的数据的请求，改成分多次获取，每次只获取一部分用户的数据，最后进行合并和汇总。

其实，处理这个问题，要分为两种场景：`同步调用` 和 `异步调用`。

8.1 同步调用

如果在`job`中需要获取2000个用户的信息，它要求只要能正确获取到数据就好，对获取数据的总耗时要求不太高。

但对每一次远程接口调用的耗时有要求，不能大于500ms，不然会有邮件预警。

这时，我们可以同步分页调用批量查询用户信息接口。

具体示例代码如下：

```
List<List<Long>> allIds = Lists.partition(ids,200);
```

```
for(List<Long> batchIds:allIds) {
    List<User> users = remoteCallUser(batchIds);
}
```

代码中我用的`google`的`guava`工具中的`Lists.partition`方法，用它来做分页简直太好用了，不然要巴拉巴拉写一大堆分页的代码。

8.2 异步调用

如果是在`某个接口`中需要获取2000个用户的信息，它考虑的就需要更多一些。

除了需要考虑远程调用接口的耗时之外，还需要考虑该接口本身的总耗时，也不能超时500ms。

这时候用上面的同步分页请求远程接口，肯定是行不通的。

那么，只能使用`异步调用`了。

代码如下：

```
List<List<Long>> allIds = Lists.partition(ids,200);

final List<User> result = Lists.newArrayList();
allIds.stream().forEach((batchIds) -> {
    CompletableFuture.supplyAsync(() -> {
        result.addAll(remoteCallUser(batchIds));
        return Boolean.TRUE;
    }, executor);
})
```

使用CompletableFuture类，多个线程异步调用远程接口，最后汇总结果统一返回。

9.加缓存

解决接口性能问题，`加缓存`是一个非常高效的方法。

但不能为了缓存而缓存，还是要看具体的业务场景。毕竟加了缓存，会导致接口的复杂度增加，它会带来数据不一致问题。

在有些并发量比较低的场景中，比如用户下单，可以不用加缓存。

还有些场景，比如在商城首页显示商品分类的地方，假设这里的分类是调用接口获取到的数据，但页面暂时没有做静态化。

如果查询分类树的接口没有使用缓存，而直接从数据库查询数据，性能会非常差。

那么如何使用缓存呢?

9.1 redis缓存

通常情况下，我们使用最多的缓存可能是：`redis`和`memcached`。

但对于java应用来说，绝大多数都是使用的redis，所以接下来我们以redis为例。

由于在关系型数据库，比如：mysql中，菜单是有上下级关系的。某个四级分类是某个三级分类的子分类，这个三级分类，又是某个二级分类的子分类，而这个二级分类，又是某个一级分类的子分类。

这种存储结构决定了，想一次性查出这个分类树，并非是一件非常容易的事情。这就需要使用程序递归查询了，如果分类多的话，这个递归是比较耗时的。

所以，如果每次都直接从数据库中查询分类树的数据，是一个非常耗时的操作。

这时我们可以使用缓存，大部分情况，接口都直接从缓存中获取数据。操作redis可以使用成熟的框架，比如：jedis和redisson等。

用jedis伪代码如下：

```
String json = jedis.get(key);
if(StringUtils.isNotEmpty(json)) {
   CategoryTree categoryTree = JsonUtil.toObject(json);
   return categoryTree;
}
return queryCategoryTreeFromDb();
```

先从redis中根据某个key查询是否有菜单数据，如果有则转换成对象，直接返回。如果redis中没有查到菜单数据，则再从数据库中查询菜单数据，有则返回。

此外，我们还需要有个job每隔一段时间，从数据库中查询菜单数据，更新到redis当中，这样以后每次都能直接从redis中获取菜单的数据，而无需访问数据库了。![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/60b64d5d275b4c8489e755c59a76af49~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=eAwemjXcKErOa4Sr9XHH8QBiFR8%3D)这样改造之后，能快速的提升性能。

但这样做性能提升不是最佳的，还有其他的方案，我们一起看看下面的内容。

9.2 二级缓存

上面的方案是基于redis缓存的，虽说redis访问速度很快。但毕竟是一个远程调用，而且菜单树的数据很多，在络传输的过程中，是有些耗时的。

有没有办法，不经过请求远程，就能直接获取到数据呢?

答：使用`二级缓存`，即基于内存的缓存。

除了自己手写的内存缓存之后，目前使用比较多的内存缓存框架有：guava、Ehcache、caffine等。

我们在这里以`caffeine`为例，它是spring官方推荐的。

第一步，引入caffeine的相关jar包

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
    <version>2.6.0</version>
</dependency>
```

第二步，配置CacheManager，开启EnableCaching

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager(){
        CaffeineCacheManager cacheManager = new CaffeineCacheMana
```

```
ger();
        //Caffeine配置
        Caffeine<Object, Object> caffeine = Caffeine.newBuilder()
                //最后一次写入后经过固定时间过期
                .expireAfterWrite(10, TimeUnit.SECONDS)
                //缓存的最大条数
                .maximumSize(1000);
        cacheManager.setCaffeine(caffeine);
        return cacheManager;
    }
}
```

第三步，使用Cacheable注解获取数据

```
@Service
public class CategoryService {

   @Cacheable(value = ”category”, key = ”#categoryKey”)
   public CategoryModel getCategory(String categoryKey) {
      String json = jedis.get(categoryKey);
      if(StringUtils.isNotEmpty(json)) {
         CategoryTree categoryTree = JsonUtil.toObject(json);
         return categoryTree;
      }
      return queryCategoryTreeFromDb();
   }
}
```

调用categoryService.getCategory()方法时，先从caffine缓存中获取数据，如果能够获取到数据，则直接返回该数据，不进入方法体。

如果不能获取到数据，则再从redis中查一次数据。如果查询到了，则返回数据，并且放入caffine中。

如果还是没有查到数据，则直接从数据库中获取到数据，然后放到caffine缓存中。

具体流程图如下：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-

73owjymdk6/77a5d36cf9e54741bf0bd4319b0acc8a~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=o0soP8k6BYlwN2zdLka9bn3lvAY%3D)该方案的性能更好，但有个缺点就是，如果数据更新了，不能及时刷新缓存。此外，如果有多台服务器节点，可能存在各个节点上数据不一样的情况。

由此可见，二级缓存给我们带来性能提升的同时，也带来了数据不一致的问题。使用二级缓存一定要结合实际的业务场景，并非所有的业务场景都适用。

但上面我列举的分类场景，是适合使用二级缓存的。因为它属于用户不敏感数据，即使出现了稍微有点数据不一致也没有关系，用户有可能都没有察觉出来。

10. 分库分表

有时候，接口性能受限的不是别的，而是数据库。

当系统发展到一定的阶段，用户并发量大，会有大量的数据库请求，需要占用大量的数据库连接，同时会带来磁盘IO的性能瓶颈问题。

此外，随着用户数量越来越多，产生的数据也越来越多，一张表有可能存不下。由于数据量太大，sql语句查询数据时，即使走了索引也会非常耗时。

这时该怎么办呢?

答：需要做`分库分表`。

如下图所示：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/021a9fbb2fc645d79f3c4f4a53362515~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=aaSX7EB6ZKxuVpS2zBwHNws3o40%3D)图中将用户库拆分成了三个库，每个库都包含了四张用户表。

如果有用户请求过来的时候，先根据用户id路由到其中一个用户库，然后再定位到某张表。

路由的算法挺多的：

* `根据id取模`，比如：id=7，有4张表，则7%4=3，模为3，路由到用户表3。
* `给id指定一个区间范围`，比如：id的值是0-10万，则数据存在用户表0，id的值是10-20万，则数据存在用户表1。
* `一致性hash算法`

分库分表主要有两个方向：`垂直`和`水平`。

说实话垂直方向（即业务方向）更简单。

在水平方向（即数据方向）上，分库和分表的作用，其实是有区别的，不能混为一谈。

* `分库`：是为了解决数据库连接资源不足问题，和磁盘IO的性能瓶颈问题。
* `分表`：是为了解决单表数据量太大，sql语句查询数据时，即使走了索引也非常耗时问题。此外还可以解决消耗cpu资源问题。
* `分库分表`：可以解决 数据库连接资源不足、磁盘IO的性能瓶颈、检索数据耗时 和 消耗cpu资源等问题。

如果在有些业务场景中，用户并发量很大，但是需要保存的数据量很少，这时可以只分库，不分表。

如果在有些业务场景中，用户并发量不大，但是需要保存的数量很多，这时可以只分表，不分库。

如果在有些业务场景中，用户并发量大，并且需要保存的数量也很多时，可以分库分表。

关于分库分表更详细的内容，可以看看我另一篇文章，里面讲的更深入《[阿里二面：为什么分库分表？](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490459&idx=1&sn=1e4296228c00aa4203aab481575ac916&chksm=c0ebc373f79c4a658de7ce7f0d8cf30b1f45adb346c2386321779e7cf85a757a12337d3ae233&token=2041133408&lang=zh_CN&scene=21#wechat_redirect”)》

11. 辅助功能

优化接口性能问题，除了上面提到的这些常用方法之外，还需要配合使用一些辅助功能，因为它们真的可以帮我们提升查找问题的效率。

11.1 开启慢查询日志

通常情况下，为了定位sql的性能瓶颈，我们需要开启mysql的慢查询日志。把超过指定时间的sql语句，单独记录下来，方面以后分析和定位问题。

开启慢查询日志需要重点三个参数：

* `slow_query_log` 慢查询开关
* `slow_query_log_file` 慢查询日志存放的路径
* `long_query_time` 超过多少秒才会记录日志

通过mysql的`set`命令可以设置：

```
set global slow_query_log='ON';
set global slow_query_log_file='/usr/local/mysql/data/slow.log';
set global long_query_time=2;
```

设置完之后，如果某条sql的执行时间超过了2秒，会被自动记录到slow.log文件中。

当然也可以直接修改配置文件`my.cnf`

```
[mysqld]
slow_query_log = ON
slow_query_log_file = /usr/local/mysql/data/slow.log
long_query_time = 2
```

但这种方式需要重启mysql服务。

很多公司每天早上都会发一封慢查询日志的邮件，开发人员根据这些信息优化sql。

11.2 加监控

为了出现sql问题时，能够让我们及时发现，我们需要对系统做`监控`。

目前业界使用比较多的开源监控系统是：`Prometheus`。

它提供了 `监控` 和 `预警` 的功能。

架构图如下：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/f6c2609a0eaa4820a2e03dc34bfb1a4f~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=ZXBycKYfWKQBJ%2FeHk677zGVj63k%3D)

我们可以用它监控如下信息：

* 接口响应时间
* 调用第三方服务耗时
* 慢查询sql耗时
* cpu使用情况
* 内存使用情况
* 磁盘使用情况
* 数据库使用情况

等等。。。

它的界面大概长这样子：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/d3fa4117ff204c2f8f14c51ba74af627~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=R3Qexoxm2yrAy70ja6lZOzcW%2FvY%3D)可以看到mysql当前qps，活跃线程数，连接数，缓存池的大小等信息。

如果发现数据量连接池占用太多，对接口的性能肯定会有影响。

这时可能是代码中开启了连接忘了关，或者并发量太大了导致的，需要做进一步排查和系统优化。

截图中只是它一小部分功能，如果你想了解更多功能，可以访问Prometheus的官网：[prometheus.io/](http://cxyroad.com/ "https://prometheus.io/")

11.3 链路跟踪

有时候某个接口涉及的逻辑很多，比如：查数据库、查redis、远程调用接口，发mq消息，执行业务代码等等。

该接口一次请求的链路很长，如果逐一排查，需要花费大量的时间，这时候，我们已经没法用传统的办法定位问题了。

有没有办法解决这问题呢?

用分布式链路跟踪系统：`skywalking`。

架构图如下：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/47ba794de8f242acbdbf141f8a77bf12~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=QdaxkcjRaliD1VxyOlGMad%2FquwE%3D)通过skywalking定位性能问题：![图片](https://p6-xtjj-sign.byteimg.com/tos-cn-i-73owjymdk6/4f3b64799460481b8195da886259c46b~tplv-73owjymdk6-watermark.image?rk3s=f64ab15b&x-expires=1721995935&x-signature=uC1aSBvQCXK9NRA6%2BTQHbdM7cDg%3D)在skywalking中可以通过`traceId`（全局唯一的id），串联一个接口请求的完整链路。可以看到整个接口的耗时，调用的远程服务的耗时，访问数据库或者redis的耗时等等，功能非常强大。

之前没有这个功能的时候，为了定位线上接口性能问题，我们还需要在代码中加日志，手动打印出链路中各个环节的耗时情况，然后再逐一排查。

如果你用过skywalking排查接口性能问题，不自觉的会爱上它的。如果你想了解更多功能，可以访问skywalking的官网：[skywalking.apache.org/](http://cxyroad.com/

”https://skywalking.apache.org/”)

最后说一句(求，别白嫖我)

如果这篇文章对您有所帮助，或者有所启发的话，帮忙扫描下发二维码一下，您的支持是我坚持写作最大的动力。

求一键三连：点赞、转发、在看。

公众号：【苏三说技术】，在公众号中回复：面试、代码神器、开发手册、时间管理有超赞的粉丝福利，另外回复：加群，可以跟很多BAT大厂的前辈交流和学习。

原文链接: https://juejin.cn/post/7393298241762279433