# 问题引入

## 代码冗长

* 遍历逻辑代码冗长
* 排序逻辑代码冗长
* 集合筛选转换代码冗长
* 空值判断代码冗长
* 线程创建代码冗长

## 安全问题

* 集合层级结构扁平化的安全问题
* 集合并发操作的安全问题

## 性能问题

* 过滤求值性能问题
* 滥用Stream的forEach的性能问题
* 乱用并行流的性能问题

# 优化方案设计

## 代码冗长优化方案

* 合理设计，尽量复用代码
* 代码重构，使代码更简洁
* 使用最新的语法和工具类进行开发
* 利用lambda表达式简化代码

## 安全问题优化方案

* 只读共享：只读对象可以共享给多个线程访问，但是任何线程都不能改它。
* 加锁：被保护对象只能通过持有特定的锁来访问。
* 利用lambda表达式来解决安全问题。

## 性能问题优化方案

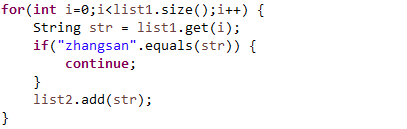
* 利用多线程发挥出多处理器的强大能力
* 在不需要对集合数据做操作的时候尽量使用增强型的for循环来遍历集合
* 利用Stream流的懒特性和短路操作
* 区分出业务属于计算密集型还是IO密集型业务，合理使用parallelStream

# 代码分析与方案选择

## 冗长问题分析和优化方案选择

### 遍历逻辑优化

1. 代码：

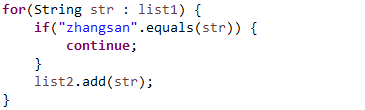


1. 分析：

以上代码利用for循环对list1做遍历，排除掉“zhangsan”，将list1中的数据存到list2中，看起来已经很简单了，但还是有优化的余地，优化后可以让我们少写很多代码。

1. 优化方案：

* A：采用增强型的for循环



* B：采用Stream + lambda 表达式



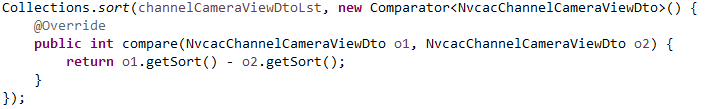
1. 选择：

* 方案分析：
* 方案A是java8之前推荐使用的循环方式，比用下标的循环更简洁；
* 方案B是java8之后在增强型for循环外增加的一种方法，最大的优点就是简洁、优雅；
* 方案选择：

通过以上代码的分析和比较，方案A比原始方案更简单，方案B则在方案A的基础之上继续优化，极大的提高了编程效率和程序的可读性，只需要1行代码就能完成7行代码的逻辑，所以推荐大家选择方案B。

### 排序逻辑优化

1. 代码：



1. 分析：

以上代码利用Collections工具类的排序功能，sort方法参数中传入了一个排序的匿名内部类对象，对通道关联的监控点集合进行排序。为了完成排序功能，写了6行代码，我们完全可以用更简洁的代码来完成以上的功能。

1. 优化方案：

* A：Collections.sort + lambda 表达式



* B：采用list.sort + lambda 表达式



* C：采用list.sort + lambda 表达式 + 方法引用

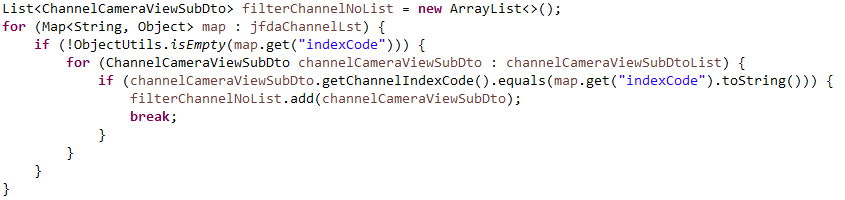


1. 选择：

通过以上代码的分析和比较，三种方案都比原来要简洁很多，方案A和方案B原理一样，Collections.sort内部就是利用list的sort方法来排序，所以方案B要略优于A，而方案C利用了方法引用，在方案B的基础之上只需要一行就能解决，所以我们会优先选择方案C来做集合的排序。

### 集合筛选转换优化

1. 代码：



1. 分析：

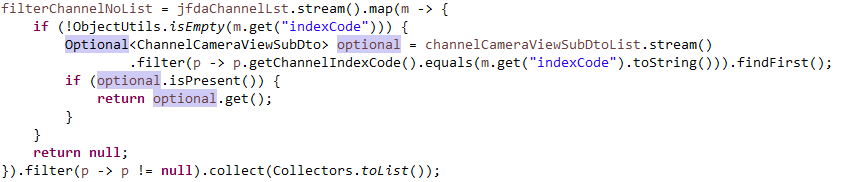
以上代码通过两层循环遍历和比较，将两个集合中indexCode相等的对象放到一个新的list中，功能上没问题，但是代码比较多，不够简洁，同时性能也不是最好的。

1. 优化方案：

* A：先对集合进行map筛选 + 再两个集合取交集



* B：对集合做 map + filter +collect



1. 选择：

* 方案分析：
* 方案A先将jfdaChannelLst集合中的通道编码转换成indexCodeList（字符串集合）中，再利用字符串集合过滤channelCameraViewSubDtoList；
* 方案B比两层for循环稍微好点，先通过map函数对集合进行转换，再通过filter过滤无用数据，最后通过collect收集元素放到一个新的集合中；
* 对以上两种方案和原始代码做几次简单的性能测试，结果如下：







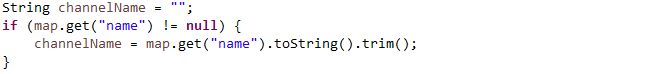
消耗时间从小到大排序为：方案A < 方案B < 两层for循环。

* 方案选择：

通过以上代码的分析和比较，从代码的简洁和性能上，我们肯定要选方案A。

### 空值判断优化

1. 代码：



1. 分析：

以上代码是通过指定的key值从map中获取value对象，并判断其值是否为空，不为空则去除字符串的空格。上面的代码很简单，但是这段代码却有很大的改造空间，改造完后可以让其更加精炼。

1. 优化方案：

* A：通过三目运算符简化代码



* B：采用Optional.ofNullable+ lambda 表达式

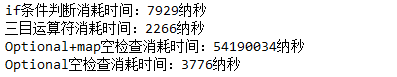


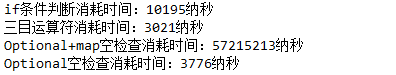
* C：采用Optional.ofNullable

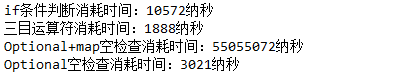


1. 选择：

* 方案分析：
* 对以上三种方案和原始代码做几次简单的性能测试，结果如下：







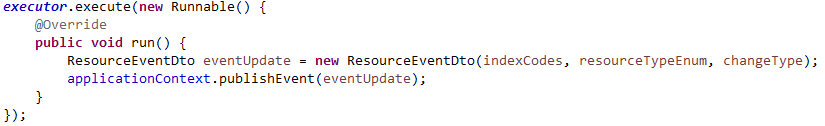
通过三次测试，消耗时间从小到大排序为：方案A<方案C<if条件判断<方案B。

* 方案选择：

通过以上代码的分析和比较，3个方案都能一句话完成空值判断，但是三者实现的方式存在很大的差异，我个人倾向方案C，方案C的扩展性更好，但是方案A比用lambda表达式更简洁、性能更好。所以从代码的简洁和性能上，我们选择方案A。

### 线程创建优化

1. 代码：

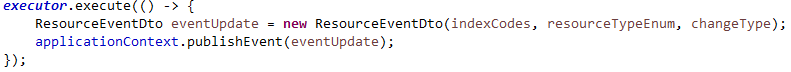


1. 分析：

以上代码是在事务提交之后通过线程池工具类创建一个线程发布变更通知。虽然上面的代码没有功能上的问题，但是看起来就是让人不太满意。

1. 优化方案：

* A：采用高阶函数替换Runnable接口



1. 选择：

通过以上代码的分析和比较，方案A用4行代码就替换原来7行代码完成的功能，我们没有理由拒绝方案A来优化我们的代码。

## 安全问题分析和优化方案选择

### 集合层级结构扁平化的安全优化

1. 代码：



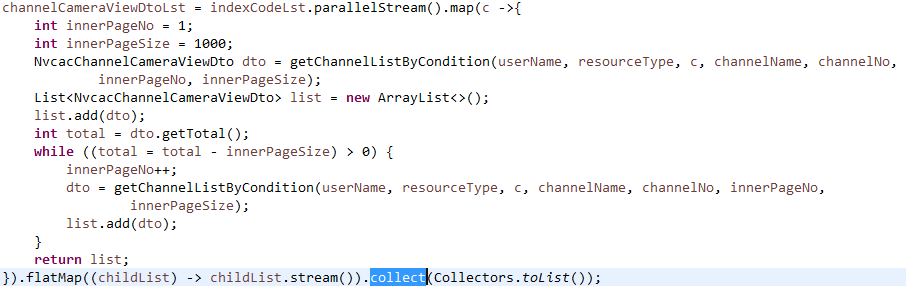
1. 分析：

由于isupm返回的数据不能满足界面的要求，我们需要将属于用户的通道信息全部查询出来，再通过界面传入的条件过滤返回的资源信息，最后重新分页返回给前端。以上代码片段是获取用户有权限通道的部分代码，根据用户名和区域编码并发查询用户的通道信息，最后将信息放到一个Vector集合中。

这段代码的性能还行，但是令人不满意的地方有两处，一处是必须要用线程安装的集合去承载数据，原因是利用并发流访问接口，不使用Vector，会造成集合数据的丢失；另一处则是集合最后还要做一次排序来固定数据展示的顺序，防止界面上展示的数据的顺序产生波动。

1. 优化方案：

* A：采用flatMap + collect，把子Stream中的元素压缩到父集合中

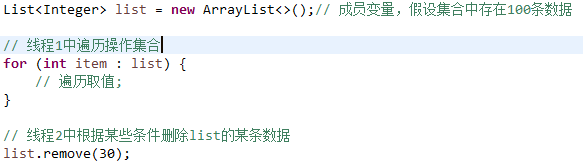


1. 选择：

通过以上代码的分析和比较，方案A明显优于原始的代码，不用创建一个安全集合去承载数据，排序问题也通过collect方法解决掉了，同时性能上也略优于原始的方案，所以从代码简洁性和性能上我们要选择方案A来优化代码。

### 集合并发操作的安全优化

1. 代码：

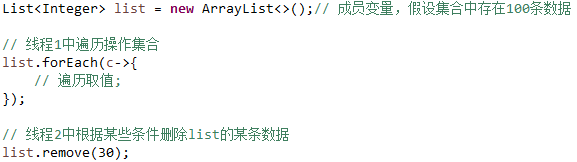


1. 分析：

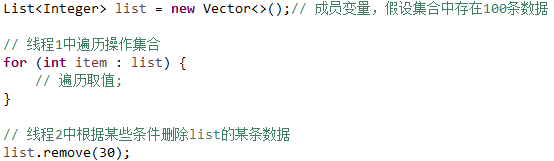
以上代码模拟两个线程操作成员变量list，成员变量的作用域在整个对象内部都是可见的，导致以上代码会有一定几率报并发修改异常，代码本身很简洁，无需再精简，但是怎么样写代码让逻辑正确安全，需要我们思考一下。

1. 优化方案：

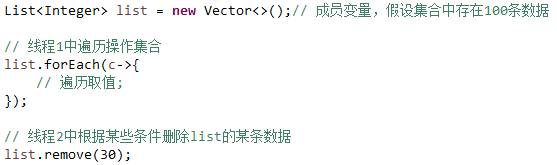
* A：采用forEach来迭代



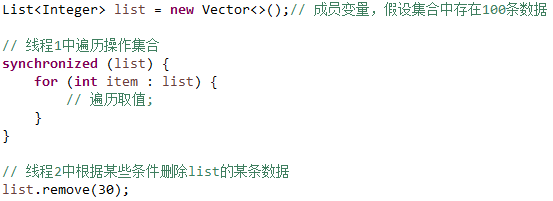
* B：采用线程安全的集合Vector来定义list



* C：采用线程安全的集合Vector来定义list + forEach来迭代



* D：采用线程安全的集合Vector来定义list + synchronized (list)来迭代



1. 选择：

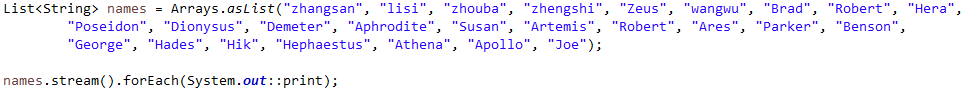
* 方案分析：
* 方案A采用forEach来遍历和增强型for循环没有区别，同样会报并发异常；
* 方案B采用Vector来定义集合，结果和ArrayList一样，依然会报并发异常，原因是for循环和锁没关系，remove操作和for循环这两个操作不是互斥的；
* 方案C采用Vector来定义list，再利用Vector自带的forEach来迭代，我们去看源码会发现forEach也是加锁的，这样要么先删除，要么先遍历，不会出现遍历的时候修改数据，自然就不会抛异常了；
* 方案D采用Vector来定义list，再在for循环前面加上锁，结果和方案C一样，都能达到安全访问。
* 方案选择：

方案A和方案B不用考虑，方案C从代码的可读性和封装性上要优于方案D，所以我们选择方案C来安全访问成员集合变量。

## 性能问题分析和优化方案选择

### ForEach的性能优化

1. 代码：



1. 分析：

以上代码通过流的forEach方法遍历names集合，这里我没有用项目中的代码来举例，但是这样的写法在项目中到处都是，这是一个经典的滥用forEach的例子，这是因为大家还没真正理解Stream的作用，只是为了用forEach而用forEach。

1. 优化方案：

* A：采用list自带的iterator方法来迭代



* B：采用list自带的forEach方法来迭代



* C：采用带下标的for循环来迭代

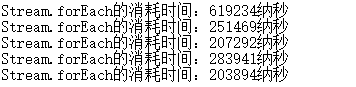


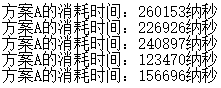
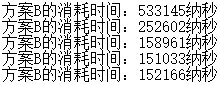
* D：采用增强型的for循环来迭代

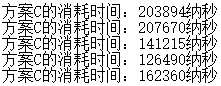
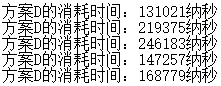


1. 选择：

* 方案分析：
* 原始代码和方案B都是利用Stream的forEach方法，方法内第一次使用stream的forEach方法速度很慢，为了让它们能更贴近真实场景，需要对Stream先预热执行；
* 对以上四种方案和原始代码做5次简单的性能测试，结果如下：



通过五次测试，消耗时间从小到大排序为：方案C <方案D<方案A<方案B< Stream.forEach

* 方案选择：

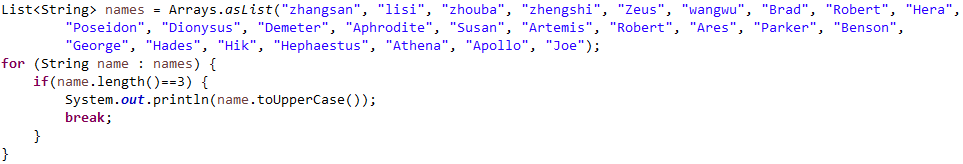
通过对方案的性能测试能发现对于集合的迭代，命令式的循环迭代编码性能要好于forEach操作。方案A/C/D的性能都差不多，但是方案D的写法最简单，所以针对集合的遍历操作，推荐大家使用增强型的for循环，如果你非要用forEach，建议优先选择用集合自带的forEach。

看起来这和4.1.1遍历逻辑优化，推荐大家使用lambda表达式来简化遍历逻辑矛盾，我想说的是，在性能和简洁性，是有优先级的，这和CAP原则一样，你想要一致性、可用性和分区容错性3个特性都被兼顾，这个想法是不可能实现的，我们只能根据实际的场景兼顾两个特性，弥补另外一个特性。因为我们的行业特性，不像互联网业务对于性能的要求那么高，所以损失一点点的性能是完全可以接受的，所以在编码的时候应该优先考虑代码逻辑的简洁性，但是并不是说要滥用lambda表达式，比如简单的for循环，完全没必要都改成forEach来遍历。

要知道我们使用流是因为我们需要对集合进行过滤、聚合或者数据转换，如果只是单纯的遍历，尽量使用增强型的for循环，但也不排除后续java会对forEach的性能进行优化。

### 数组过滤性能优化

1. 代码：



1. 分析：

以上代码通过增强型for循环从集合中找出第一个长度为3的名字，并将名字转换成大写。

1. 优化方案：

* A：采用Steam + filter + map + findFirst 来寻值



* B：采用Steam + filter + findFirst 来寻值

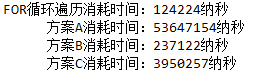
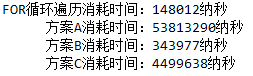


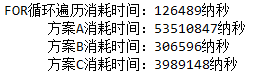
* C：采用parallelStream + filter + findFirst 来寻值



1. 选择：

* 方案分析：
* 方案A经过22次比较后找到“Hik”这个长度为3的name，再通过map函数内部toUpperCase转成大写，最后通过findFirst返回名称；
* 方案B经过22次比较后找到“Hik”这个长度为3的name，再通过findFirst返回名称，最后通过toUpperCase将返回名称转换大写；
* 方案C在方案B的基础上换成并行流取值；
* 对以上三种方案和原始代码做几次简单的性能测试，结果如下：



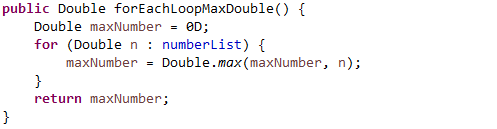
通过三次测试，消耗时间从小到大排序为：for循环< 方案B < 方案C < 方案A

* 方案选择：

通过方案分析和性能测试能发现对于计算密集型的业务，命令式的循环迭代编码性能要好于lambda表达式，lambda表达式越多性能越差。在数据量不多的情况下使用parallelStream的性能反而比不上串行的stream，这是因为准备线程池和其它相关资源也是需要时间的。如果从代码的简洁和优雅性来讲我们应该选择方案B，如果追求性能上的卓越的话，我们应该选择增强型的for循环来查询，个人推荐使用方案B来实现。

### 计算密集型业务的性能优化

1. 代码：



1. 分析：

以上代码是通过增强型的for循环从100W、1000W、3000W和5000W的数组中找出最大的值，最大值为数组的最后一个数字，保证循环一直执行下去，代码很简单。

1. 优化方案：

* A：采用Stream + lambda表达式



* B：采用parallelStream + lambda表达式



1. 选择：

* 方案分析：
* 方案A利用串行流，reduce获取数组中最大的值；
* 方案B利用并行流，reduce获取数组中最大的值；
* 此案例由于数据量比较大，所以没有做Stream的预热，在1000W数据量以下预热后的Stream的性能要比实验结果要好一点，但是随着数据量的增多差距就不明显了；
* 对以上两种方案和原始代码做了以下性能测试，结果如下：
* 100W



* 1000W



* 3000W



* 5000W



通过上面的性能测试，当数组为100W的时候，消耗时间从小到大排序为：for循环< 方案A < 方案B；当数组为1000W的时候，消耗时间从小到大排序为：for循环< 方案A < 方案B；当数组为3000W的时候，消耗时间从小到大排序为：for循环< 方案B < 方案A；当数组为5000W的时候，for循环和方案B时间已经很接近了，而方案A的性能就比较差了，这样可以推断出当1亿条数据的时候方案B采用并行流消耗的时间是最小的。

* 方案选择：

通过方案分析和性能测试能发现对于计算密集型的业务，传统的命令式代码性能是最好的，只有计算量非常大的时候我们才可以采用方案B来计算。当然从代码的优雅和简洁性来考虑的话，也可以采用方案A，这个需要我们对业务场景有一定的了解，选择最优的方案。

### IO密集型业务的性能优化

1. 代码：



1. 分析：

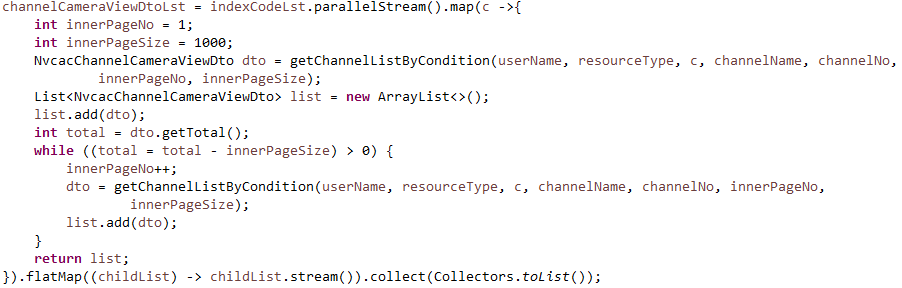
以上代码片段是获取用户有权限通道的部分代码，根据用户名和区域编码并发查询用户的通道信息getChannelListByCondition是调用isupm提供的restful接口，接口是/api/privilegeService/v1/resources/query，涉及到大量的网络传输的业务就是IO密集型业务，上面这段代码的性能在区域增多的情况下，性能会越来越差。

1. 优化方案：

* A：采用线程池，并发执行



* B：采用parallelStream + lambda表达式



1. 选择：

* 方案分析：
* 方案A利用Executors.newFixedThreadPool开线程并发执行，在java8之前，大家一般都是采用这种方式来开线程并发执行任务，性能肯定是提升了，但是代码量也多了起来；
* 方案B利用parallelStream 并发执行任务，从代码量来看，代码更精简和优雅了；
* 对以上两种方案和原始代码做几次简单的性能测试，结果如下：

通过四次测试，消耗时间从小到大排序为：方案B < 方案A < for循环。

* 方案选择：

通过方案分析和性能测试能发现对于IO密集型的业务，串行操作的性能随着IO操作的增加，性能就越来越慢，因为大部分时间都阻塞在IO等待上，这时利用多线程就可以很好的解决单线程的劣势，所以单线程的方案肯定不行。在没有parallelStream之前，我们只能选择方案A，但是有了parallelStream之后，我们会发现并发操作更简单了，所以我们会选择方案B来实现。

# 总结

引用Biggie Smalls的一句话：“**代码越多，问题也就越多**”，任何逻辑如果能用简洁的语句描述，为何还要长篇大论呢。代码越简洁，越能被发现问题，对于软件项目的发展和维护就越有利。java代码主要是被解释执行的，代码越多，被解释的次数也越多，同理性能也会变差，利用java8的新特性能减少我们写的代码量，但是它编译之后真正执行的命令行数却未必比传统的行数少，这是我们需要考虑和抉择的，需要我们在性能和简洁性上做抉择。

Stream的forEach方法，它的遍历性能比传统的命令式代码要略差，但这个缺点并不能说明它就是不好的，我们需要理解的是为什么要用Stream，用它能帮我们解决什么问题，往往合理的利用java8的Stream+Lambda表达式，减少执行的步骤，可以使代码更精简和优化，代码的性能也会略优于传统的命令式编码，这个通过之前案例的性能分析已经说明了。

本文用了很大的篇幅来描述java8的新特性给我们带来的好处，但是最重要的还是体现在合理两个字上，任何事物都不可能没有缺点，要避免过犹不及，如何规避这些缺点需要我们在开发的时候多思考和比较，不要盲目追求新技术，也不要惧怕新技术。