前言

优化接口性能对每个后端开发同学来说见惯不惯了，也是一项必备的技能，因为我们平时开发中都会对外提供接口，性能差的话，功能多少会有影响。另外接口性能算是一个跟开发语言无关的公共问题，该问题既简单又复杂。

一般导致接口性能问题的原因不尽相同，项目功能不同的接口，导致接口出现性能问题的原因可能也不一样，要根据场景来分享，即具体情况具体分析。

但是本文主要总结了一些优化接口性能的办法，一般可以从以下几点进行优化：

- 加索引
- 代码重构
- 增加缓存
- 引入一些中间件，比如 mq
- 分库分表
- 拆分服务
- 等等

1. 索引

根据经验，相信大家第一个想到接口性能优化方法就是：**优化索引**。

是的，优化索引的成本是最小的。你可以通过查看 sql 的执行计划来判断 sql 是否走索引等。

又或者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 语句实际耗时（一般通过慢查询日志可以知道）。

优化索引会从以下几个 Question 入手：

- 使用的 sql 语句是否加索引了？
- 加索引后，实际是否生效了？
- mysql 会不会使用到加的索引？

1.1 没加索引的情况

在平时项目中比较常见的问题：就是在 sql 语句中 `where` 条件的关键字段，或者 `order by` 后面的排序字段，漏加索引。

当然项目初期体量比较小，表中的数据量小，加不加索引 sql 查询性能差别不大，没啥影响。

随后，如果业务发展起来了，表中数据量也越来越多，此时就不得不加索引了。

查看某张表索引情况的命令：

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

查看整张表的建表语句，里面也会显示索引情况：

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

另外，添加索引的命令：

```
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);
```

需要注意的地方：**mysql 没有命令可以直接修改索引，想要修改索引，只能先删除索引，再重新添加新的。**

删除索引可以使用以下两个命令：

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

1.2 索引没生效的情况

通过上面的一些命令，我们已经可以知道怎么查看索引、创建索引以及删除索引。

在已经能够确认索引有的情况下，接下来需要关注它是否生效了？

首先我们可以使用 mysql 的 `explain` 命令来查看 sql 的执行计划，它会显示索引的使用情况。

For example：

```
explain select * from `t_order` where Fdeal_id=1001;
```

结果如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_order	NULL	ref	idx_deal_id	idx_deal_id	9	const	1	100.00	NULL

通过 `ref`、`key`、`key_len` 这几列可以知道索引使用情况，执行计划包含列的含义如下图所示：

`explain` 执行计划中包含关键的信息如下：

- select_type: 查询类型
- table: 表名或者别名
- partitions: 匹配的分区
- type: 访问类型
- possible_keys: 可能用到的索引
- key: 实际用到的索引
- key_len: 索引长度
- ref: 与索引比较的列
- rows: 估算的行数
- filtered: 按表条件筛选的行百分比

下面列举了常见索引失效的原因：

- 不满足最左前缀原则
- 使用了 select *

- 使用索引列时进行计算
- 范围索引没有放后面
- 字符类型没有加引号
- 索引列上使用了函数
- like 查询左侧有%
- 等等

以上原因都没有命中，则需要继续排查是什么原因。

1.3 索引没选对

不知你有没有遇到过这种情况：相同的 sql，只是参数不同而已。有时候是走索引 A，有的时候却走索引 B。

此时可以理解为 mysql 索引没有选对。

当然可以使用 `force index` 来强制查询 sql 走指定的索引。

其实 mysql 没选对索引，大概率是它的优化器所选择的结果，后面会有专门的文章介绍，这里就不展开说明了。

2. 优化 sql 技巧

当使用了索引后，接口性能也没见效。

这种情况就要试着优化一下 sql 语句，不只优化索引就完事了。

下面列举了 sql 优化的一些技巧：

- 尽量避免使用 select *
- 必要的话进行批量操作
- 用 union all 代替 union
- 尽量分页查使用 limit
- 使用高效的分页
- join 的表不宜过多
- 索引的数量不能太多
- 要对索引优化
- 选择合理的字段类型

这些技巧在这里我就不深入了，后面也会单独写一篇文章来详细介绍，在这里我就不深入了。

3. 使用远程调用（rpc）

在大多数时候，项目中往往需要在某个接口中，调用其它服务的接口。

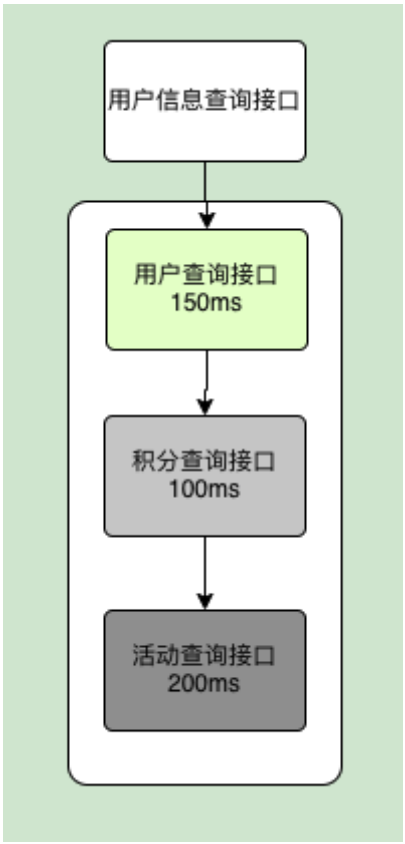
比如商城的业务场景：

下单时需要调用**用户信息接口**，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，另外也需要**调用商品信息接口**，在用户信息查询接口中需要返回：商品主图链接、价格、活动等信息。而积分在**积分服务**中，活动在**活动服务**中。

因此，为了汇总这些数据统一返回，需要另外提供一个对外接口的服务。

于是，**用户信息查询接口**就需要调用用户查询接口、积分查询接口和活动接口，然后汇总数据统一返回。

调用过程如下图所示：



可以知道远程调用接口总耗时为：450ms = 150ms + 100ms + 200ms.

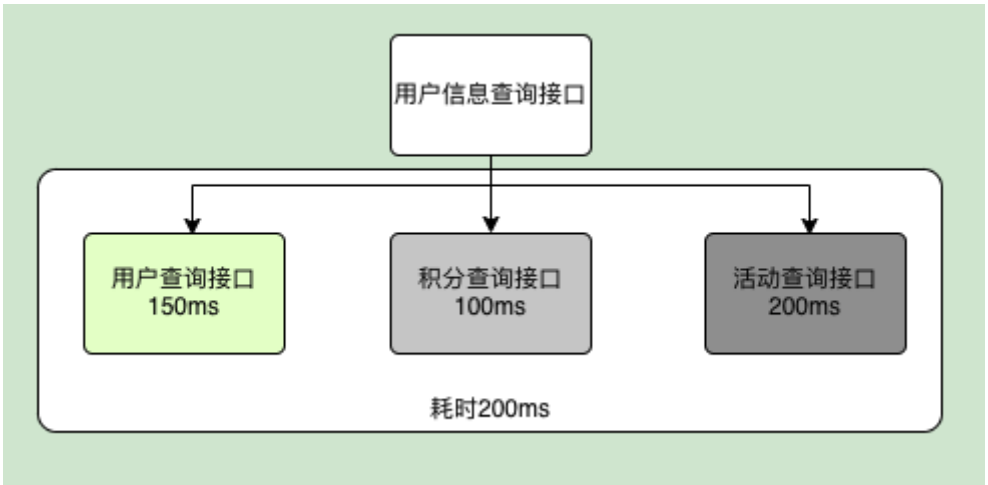
很明显这种串行远程调用接口性能是很差的，效率也非常低，远程调用接口的总耗时为调用各个远程接口耗时之和。

那么如何优化远程调用接口的性能呢？继续往下看。

3.1 改成并行调用

既然串行远程调用多个接口性能很低，那么我们可以改成并行调用哈。

并行调用如下图所示：



可以看到，改成并行后，远程调用接口总耗时为 200ms = 200ms（即对应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。

如果你是使用 `Golang` 语言开发的话，可以通过并发 `goroutine` 实现该功能。这里简单举个例子：

```
func TestGoroutine() {
 fmt.Println("start test goroutine...")

 wg := &sync.WaitGroup{}
 wg.Add(2)

 // 开启 goroutine one 模拟调用用户信息
 go func(uid string) {
  fmt.Println("goroutine one, todo: get userInfo, uid: ", uid)
  wg.Done()
 }("1001")

 // 开启 goroutine two
 go func(activeId string) {
  fmt.Println("goroutine two, todo: get activeInfo, activeId: ", activeId)
  wg.Done()
 }("1002")

 wg.Wait()
 fmt.Println("end test goroutine...")
}

```

> 温馨提醒一下，Go 语言的并发 goroutine 在前面的文章有介绍，这里不再赘述，详见：Golang无限开启Goroutine？该如何限定Goroutine数量？。

3.2 使用缓存

我们可以考虑把数据冗余一下，把用户信息、积分和活动信息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。

接下来可以通过用户 id，直接从 redis 中查询出来，这大大提高了效率。

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缓存方案。

但需要注意的是，如果使用了缓存方案，就要另外考虑数据一致性的问题，详见：**数据库和缓存的一致性问题，看这一篇就够了。**

用户信息、积分和活动信息更新的话，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. 重复调用接口

在同一个接口中，重复调用在我们平时开发的代码中可以说随处可见，但是如果没有控制好的话，会大大影响接口的性能。

4.1 循环去查数据库

大多数时候，我们需要从指定的数据库集合中，查询出需要用到的数据。

当有多个用户 id 传多来时，如果每个用户 id 都需要查一遍的话，那么就需要循环多次去查询数据库了。我们都知道，每查询一次数据库，就会进行一次远程调用。这是非常耗时的操作。

那么，我们可以提供一个根据用户id 集合批量查询用户信息数据的接口，只需远程调用一次即可，就能查询出所需要的数据了。

> 这里温馨提示下：`id` 集合的大小需要做限制以及做入参校验，否则也会影响查询性能，最好一次不要请求太多的数据。可以根据业务实际情况而定。

4.2 避免出现死循环

有些时候，写代码一不留神，循环语句就出现死循环了。

出现这种情况往往就是 `condition` 条件没处理好，导致没有退出循环，从而导致死循环。

出现死循环，大概率是代码的 bug 导致的，不过这种情况很容易被测出来。

但是，可能还有一种比较隐秘的死循环代码，当用正常数据时，测不出问题，一旦出现有异常数据，才会复现死循环的问题。

4.3 避免无限递归

一些导致无限递归的场景以及影响接口性能程度这里就不啰嗦了，总之，在写递归代码时，建议设定一个递归的深度(假设限定为 5)，然后在递归方法中做一定判断，如果深度大于 5 时，则自动返回，这样就可以避免无限递归了。

5. 考虑使用异步处理

很多时候，在进行接口性能优化时，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。

这样实现的接口表面上看起来没啥问题，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 使用线程池

使用线程池改造将**发站内通知**和**用户操作日志**功能提交到了两个单独的线程池中。
这样在接口中重点关注的是业务逻辑操作，把其他的逻辑交给线程异步执行了，改造后，不仅让接口性能瞬间提升了，还大大提高了效率。

> 温馨提示：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。

那么可以使用下面的 mq 来解决此问题。

5.2 使用 mq 异步处理

异步使用 mq 改造。
将**发站内通知**和**用户操作日志**功能，以 mq 消息的方式发送到 mq 服务器，然后由 mq 消费者进行消费消息，消费者才是真正的执行这两个功能。
同样改造后，接口性能也是提升了的，因为发送 mq 消息吞吐量高，速度快，我们只需关注业务操作的代码即可。

6. 注意锁的粒度

通常，在一些业务场景中：

- 为了防止多个线程并发修改某个共享数据，造成数据的异常以及避免出现资源竞争的问题
- 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 我们会考虑加锁，但如果加锁方式不对的话，会导致锁的粒度太粗，同样也会非常影响接口的性能。

6.1 sync.Mutex

在 Golang 中提供了 `sync.Mutex` 关键字给代码加锁。

> sync.Mutex是一个互斥锁，可以由不同的goroutine加锁和解锁。

`sync.Mutex` 是 Golang 标准库提供的一个互斥锁，当一个goroutine获得互斥锁权限后，其他请求锁的 goroutine 会阻塞在 Lock() 方法的调用上，直到调用 Unlock() 方法被释放。
未加锁的例子：
10 个并发的 goroutine 打印同一个数字 100，为避免重复打印，实现 `printOnce(num int)` 函数，使用集合 set 记录已打印过的数字。若数字已经打印过，则不再打印。

```go
package test

import (
    "fmt"
    "testing"
```

```
    "time"
)

var set = make(map[int]bool, 0)

func printOnce(index int, num int) {
    if _, ok := set[num]; !ok {
        fmt.Println(index, num)
    }
    set[num] = true
}

func TestPrint(t *testing.T) {
    for i := 0; i < 10; i++ {
        go printOnce(i, 100)
    }
    time.Sleep(time.Second)
}

```

输出结果：

```
$ go test -v mutex_test.go
=== RUN   TestPrint
9 100
3 100
--- PASS: TestPrint (1.00s)
PASS
ok      command-line-arguments  1.304s

```

程序多次运行后会发现打印次数多次，因为对同一个数据结构set的访问发生了冲突。

并发访问中比如多个goroutine并发更新同一个资源，比如计时器、账户余额、秒杀系统、向同一个缓存中并发写入数据等等。如果没有互斥控制，很容易会出现异常，比如计时器计数不准确、用户账户可能出现透支、秒杀系统出现超卖、缓存出现数据缓存等等，后果会比较严重。

使用互斥锁的 `Lock()` 和 `Unlock()` 方法解决以上冲突问题：

```
package test

import (
    "fmt"
```

```
    "sync"
    "testing"
    "time"
)

var m sync.Mutex
var set = make(map[int]bool, 0)

func printOnce(index int, num int) {
    m.Lock()
    defer m.Unlock()

    if _, ok := set[num]; !ok {
        fmt.Println(index, num)
    }
    set[num] = true
}

func TestPrint(t *testing.T) {
    for i := 0; i < 10; i++ {
        go printOnce(i, 100)
    }
    time.Sleep(time.Second)
}

```

相同的数字只会比打印一次，当一个 goroutine 调用了 Lock() 方法时，其他 goroutine 被阻塞了，直到 Unlock() 调用将锁释放。因此被包裹部分的代码就能避免冲突，实现互斥。

6.2 redis 分布式锁

redis 分布式锁在前面的文章也讲过，详见：**真正的 Redis 分布式锁，就该是这样实现**.

6.3 数据库级别的锁

使用 mysql 数据库中锁主要有三种级别：

- 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。

如果并发度越高，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。**

7. 考虑是否要分库分表

有些时候，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。

此时就需要考虑做分库分表了，详见：**数据库之分库分表的一些总结**。

8. 其它辅助优化接口功能

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。

8.1 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关注三个参数：

- 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。

8.2 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。 如果你想了解更多功能，可以访问 Prometheus 的官网：**https://prometheus.io/**

8.3 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`。