不可思议！平均执行耗时仅1.5ms的接口在超时时间100ms下成功率竟然还不到5个9！！

转转技术团队 2025-01-15 437 阅读6分钟

智能总结 复制 重新生成

文章讲述了一个平均执行耗时仅 1.5ms 的接口在超时时间 100ms 下成功率不到 5 个 9 的问题。通过分析验证，发现有请求处理时间超 100ms，原因可能是 GC 等。给出解决方案，如参照调用方 TP9999 设超时时间，框架可采用弹性超时方案，适用于偶发性超时场景。

关联问题: 怎样优化接口耗时 框架优化效果如何 弹性超时怎么设置

基于该文章内容继续向AI提问

1 背景

一个春暖花开的午后，客服技术部佩姐（P）找过来向我们反馈一个问题，如下是我们的对话：

```css
P：云杰，我们最近在治理服务质量，有个接口的成功率达不到公司标准5个9。

我：赞，你们也开始质量治理了，详细说说。

P：我们sccis有个重要的lookupWarehouseIdRandom接口，先查询缓存，未命中的再从数据库查并回写到缓存，平均执行耗时只有1

我：不至于吧！？平均执行耗时1.5ms，在调用方超时时间配100ms（60多倍！）的情况下竟然还有这么多超时？

P：真的！！不信你看看！！！

我：看看就看看！
```

如下开始本篇的研究之旅。

2 验证与分析

2.1 准备工作

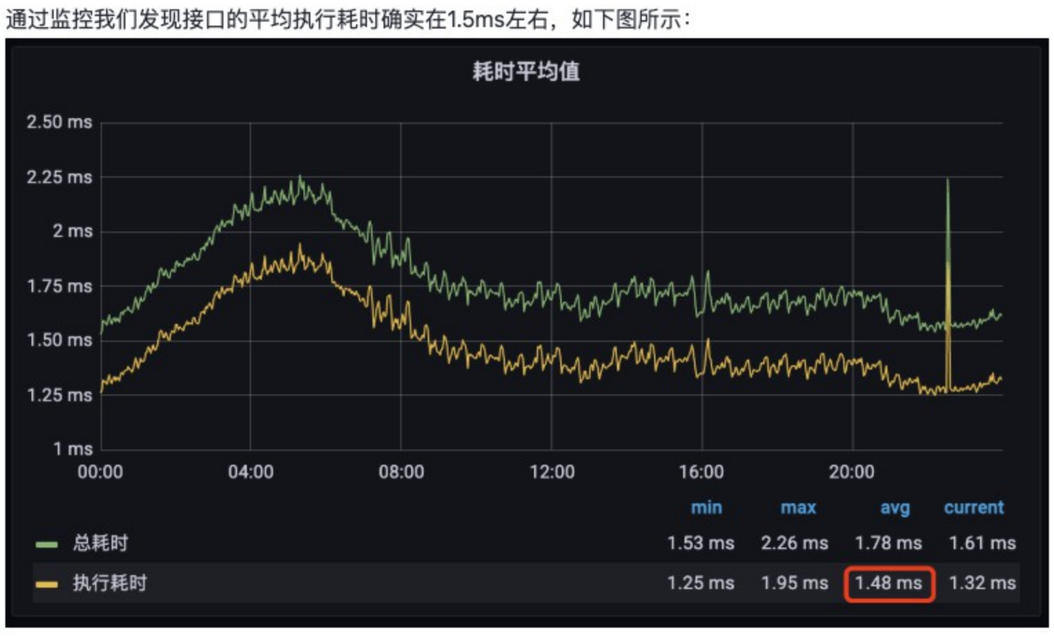
在开始验证之前，先简要介绍下转转RPC框架SCF的调用过程，如下图所示：

1. **序列化**：SCF接收到调用方的请求，做负载均衡、序列化等；
2. **发送**：SCF将序列化后的二进制流通过网络发送给服务方结点；
3. **反序列化**：服务方结点接收到数据后，将数据交给SCF，做反序列化、排队等；
4. **执行**：SCF将请求交由服务方的实现方法进行处理；
5. **序列化**：SCF将服务方的处理结果序列化为二进制数据流；
6. **发回**：将数据发回给调用方；
7. **反序列化**：调用方SCF收到请求后，将二进制数据反序列化为对象交由调用方代码，使得调用方看起来跟本地方法调用一样。

如上是一次完整的RPC调用链路。

2.2 验证

通过监控我们发现接口的平均执行耗时确实在1.5ms左右，如下图所示：



但调用方scoms在超时时间为100ms的情况下确实仍然有很多请求超时：

太让人震惊了！！！

2.3 问题分析

通过如上的RPC调用过程链路示意，我们可以看出任意一个子过程都可能会发生抖动，造成超时。但我们可以从整体上把链路分为框架和业务两个部分（分界点如图所示）：

- **框架**：指底层的网络和SCF耗时，属于客观原因，包括图中的1、2、3、5、6、7；
- **业务**：单纯指业务服务的执行4，属于主观原因。

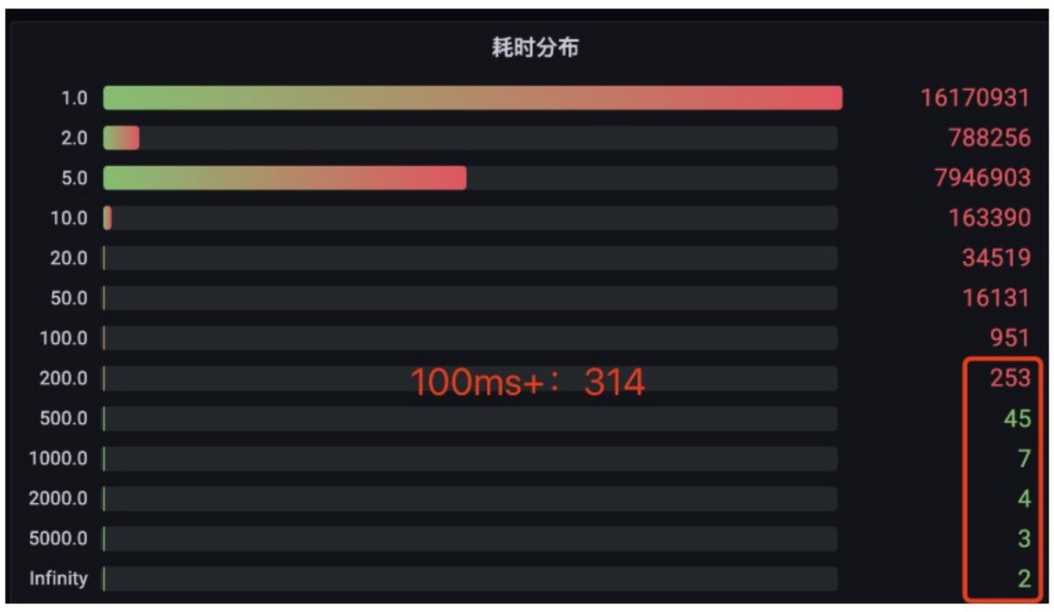
转转技术团队 LV.6
公众号：转转技术
作者榜No.14 优秀作者
466 文章 1.6m 阅读 17k 粉丝
关注 私信

目录 收起

因为框架耗时复杂多变，不好统计，我们可以统计业务的执行耗时分布，以此来判断问题出在框架上还是出在业务上。

- 如果业务的执行耗时分布都非常低，那就说明超时花在了框架上；
- 如果业务的执行耗时分布都有很多高耗时的，那就说明超时花在了业务逻辑上。

正好服务方的接口有耗时分布监控，通过监控我们发现绝大部分情况都在5ms内处理完成，但仍有314个请求处理时间直接超过了100ms！！！



这个发现也让我们大吃一惊：**平均执行耗时1.5ms的接口，竟然还会有这么多请求执行耗时越过100ms！！**
那么这些时间都花在哪里了呢？

2.4 排查

目前的监控都是接口的整体执行耗时，我们需要深入接口内部看看时间都花在哪里了。我们对接口分为如下几个部分，并分段监控起来。

```
@Override
public Result<Map<Long, Long>> lookupWarehouseIdRandom(List<Long> skuIds) {
    long t1 = System.currentTimeMillis();
    try {
        log.info("act=lookupWarehouseIdRandom skuIds.size={}", skuIds.size());
        List<Long> realSkuIdList = skuIds.stream().filter(Objects::nonNull).distinct().collect(Collectors.toList());
        if (CollectionUtils.isEmpty(realSkuIdList)) {
            long t2 = System.currentTimeMillis();
            TIME_COST.labels( ...labelValues: "emptyIds").observe( amt: t2 - t1);
            return Result.success(Collections.emptyMap());
        }
```
emptyIds

```
        List<String> skuIdRedisKeyList = realSkuIdList.stream().map(RedisKeyConstant::getSkuKey).collect(Collectors.toList());

        long t3 = System.currentTimeMillis();
        Map<String, String> skuId2CacheStringMap = jodisClient.pipeGet(skuIdRedisKeyList);
        long t4 = System.currentTimeMillis();
        TIME_COST.labels( ...labelValues: "jodis").observe( amt: t4 - t3);
```
jodis：先查缓存

```
        // 需要去请求db的数据
        List<Long> needSearchDbSkuIdList = Lists.newArrayList();

        // 可从缓存中获取到的数据
        List<SkuInventoryInfoCache> skuInventoryInfoCaches = realSkuIdList.stream().map(skuId -> {
            try {
                String val = skuId2CacheStringMap.get(RedisKeyConstant.getSkuKey(skuId));
                if (StringUtils.isNotBlank(val)) {
                    return JSONObject.parseObject(val, SkuInventoryInfoCache.class);
                }
                needSearchDbSkuIdList.add(skuId);
            } catch (Exception e) {
                log.error("act=lookupWarehouseIdRandom.parseObject e={}", E2s.exception2String(e));
                needSearchDbSkuIdList.add(skuId);
            }
            return null;
        }).filter(Objects::nonNull).collect(Collectors.toList());
        long t5 = System.currentTimeMillis();
        TIME_COST.labels( ...labelValues: "calculateDBIds").observe( amt: t5 - t4);
```
calculateDBIds：计算缓存未命中的ids

```
        if (CollectionUtils.isNotEmpty(needSearchDbSkuIdList)) {
            List<Inventory> inventories = inventoryService.selectBySkuIds(realSkuIdList);
            long t6 = System.currentTimeMillis();
            TIME_COST.labels( ...labelValues: "selectDB").observe( amt: t6 - t5);
```
selectDB：从数据库查询缓存未命中的ids

```
            Map<Long, List<Inventory>> skuId2InventoryListMap = inventories.stream().collect(Collectors.groupingBy(Inventory::getSkuId));

            List<SkuInventoryInfoCache> intoCacheList = needSearchDbSkuIdList.stream().map(skuId -> {
                SkuInventoryInfoCache skuInventoryInfoCache = new SkuInventoryInfoCache().setSkuId(skuId);
                List<Inventory> inventoryGroupList = skuId2InventoryListMap.get(skuId);
                if (CollectionUtils.isEmpty(inventoryGroupList)) {
                    return skuInventoryInfoCache;
                }
                skuInventoryInfoCache.setWarehouseInventoryInfoList(inventoryGroupList.stream().map(this::skuInventoryInfoFor).collect(Collectors.toList()));
                return skuInventoryInfoCache;
            }).collect(Collectors.toList());

            Map<String, String> intoCacheMapData = intoCacheList.stream()
                    .collect(Collectors.toMap(skuInventoryInfoCache -> RedisKeyConstant.getSkuKey(skuInventoryInfoCache.getSkuId()),
                            JSON::toJSONString, (i1, i2) -> i1));
            long t7 = System.currentTimeMillis();
            TIME_COST.labels( ...labelValues: "mapIntoCache").observe( amt: t7 - t6);
```
mapIntoCache：DB数据结构映射到cache数据结构

```
            jodisClient.pipeSetex(intoCacheMapData, seconds: 10 * 60);
            long t8 = System.currentTimeMillis();
            TIME_COST.labels( ...labelValues: "setIntoCache").observe( amt: t8 - t7);
```
setInfoCache：放入缓存

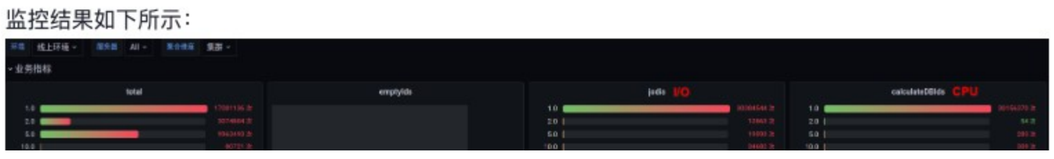
```
            skuInventoryInfoCaches.addAll(intoCacheList);
        }
        long t9 = System.currentTimeMillis();

        // 封装结果
        Map<Long, Optional<Long>> skuId2WarehouseIdOptionalMap = skuInventoryInfoCaches.stream()
                .filter(cache -> CollectionUtils.isNotEmpty(cache.getWarehouseInventoryInfoList()))
                .collect(Collectors.toMap(SkuInventoryInfoCache::getSkuId, cache -> {
                    List<SkuWarehouseInventoryInfo> warehouseInventoryInfoList = cache.getWarehouseInventoryInfoList();
                    return warehouseInventoryInfoList.stream() Stream<SkuWarehouseInventoryInfo>
                            .filter(dto -> dto.getQuantity()
                                    - dto.getLockQuantity()
                                    - dto.getAllocatedQuantity()
                                    - dto.getFrozenQuantity() > 0)
                            .map(SkuWarehouseInventoryInfo::getWarehouseId) Stream<Long>
                            .findAny();
                }, (c1, c2) -> c1));
```
toResult：提取结果id

```
        // 过滤v为null的值，减少网络传输数据
        Map<Long, Long> resultMap = Maps.newHashMap();
        skuId2WarehouseIdOptionalMap.forEach((skuId, warehouseIdOptional) -> {
            warehouseIdOptional.ifPresent(warehouseId -> resultMap.put(skuId, warehouseId));
        });
        long t10 = System.currentTimeMillis();
        TIME_COST.labels( ...labelValues: "toResult").observe( amt: t10 - t9);

        return Result.success(resultMap);
    } finally {
        long t11 = System.currentTimeMillis();
        TIME_COST.labels( ...labelValues: "total").observe( amt: t11 - t1);
        ID_SIZE.labels( ...labelValues: "all").observe(skuIds.size());
        if (t11 - t1 >=90){
            ID_SIZE.labels( ...labelValues: "gt90").observe(skuIds.size());
        }
    }
}
```

监控结果如下所示：

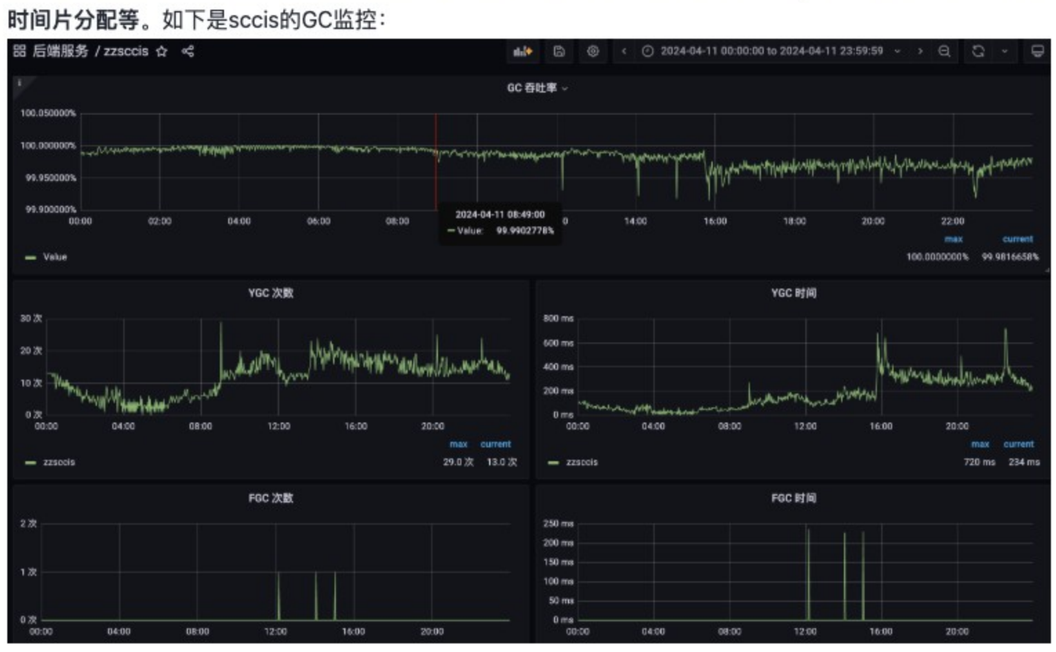


从结果可以看到：

- I/O操作容易抖动，出现较多次100ms+；
- 最简单的CPU操作虽然没有那么多100ms+，但也有不少20ms左右的情况（而且都是从1ms突变到20ms，而不是渐变）。

2.5 原因

原来我们是被1.5ms给平均了！什么原因会导致这种长尾效应呢？情况可能有很多，GC（极度怀疑）、CPU时间片分配等。如下是sccis的GC监控：



为此，我们也对比了转转商品服务zzproduct的getProductById()接口，发现也有同样的情况：

3 解决方案

至此，**我们看到业务接口平均执行耗时虽然仅有1.5ms，但仍会出现不少超过100ms的长尾效应，当然框架也会出现。其原因有多种，GC（极有可能）、CPU时间片分配、网络抖动等等。**

而这，也确实刷新了我们所有人的认知。

反过来想，如果业务接口要达到公司要求的5个9要求，该怎么办呢？其实很简单，我们可以参照调用方的TP9999来设置超时时间。如下图，scoms调用该接口的TP99999是123ms，而业务把超时时间配置成了100ms，那肯定达不到5个9的标准了。要么把超时时间改为123ms（简单直接），要么优化业务逻辑（目测很难，因为平均执行耗时只有1.5ms）或JVM调优（很有希望）。

3.1 框架优化-弹性超时

基于本文分析，RPC框架也可以针对这种长尾效应做一定优化：**不改变超时时间100ms配置情况下，允许一段时间（可配）一些量（可配）的请求在200ms（可配）时间内返回，既提高了服务质量，又不太影响用户体验**，我们称之为弹性超时方案。

3.1.1 效果

如下图所示，我们在服务管理平台支持按服务&函数设置弹性超时，这里我们将上文zzscoms调zzsccis的 `IInventoryWrapCacheFacade.lookupWarehouseIdRandom(List)` 函数配置成每40秒允许15个请求的超时时间延长至1300毫秒。

通过配置弹性超时，我们看到这种偶发性的超时基本被容忍消灭掉了，如下图所示：

3.1.2 适用场景

弹性虽好，可不要贪杯！它更多适用于一些偶发性超时场景，比如网络抖动、GC、CPU抖动、冷启动等，如果是大面积的超时还是需要深入分析治理。

4 总结

本文深入分析了平均耗时仅有1.5ms的接口也会出现大量100ms+的前因后果，并在框架层面给出了弹性超时的解决方案。这也刷新了我们的认知，由于GC、CPU时间片等原因，一些看起来很简单的操作（如i++）也会出现偶发性长耗时。

> 关于作者

杜云杰，高级架构师，转转架构部负责人，转转技术委员会执行主席，腾讯云TVP。负责服务治理、MQ、云平台、APM、分布式调用链路追踪、监控系统、配置中心、分布式任务调度平台、分布式ID生成器、分布式锁等基础组件。微信号：`waterystone`，欢迎建设性交流。

道阻且长，拥抱变化；而困而知，且勉且行。

标签： RPC

评论 1

登录 / 注册 即可发布评论！

最热 | 最新

你一户口本都是憨熊 抓娃遂夏普攻城湿 @外包公司
alibaba fastjson处理大对象的时候耗时比较严重
8天前 1 评论 ···

为你推荐

Wireshark TS | 循序渐进看系统访问偶发失败
7ACE | 1年前 | 233 点赞 评论 Wiresh... TCP/IP 网络协议

【从0-1 千万级直播项目实战】线上发布时RPC调用经常404/503，被运营骂惨了
消灭知识盲区 | 1年前 | 2.8k 8 评论 后端 架构 Java

一次Elastic APM导致的线上性能问题
某某祺 | 2年前 | 1.2k 8 评论 后端

什么？监控打点也能导致RPC调用超时？
Crady | 11月前 | 354 3 评论 后端

记一次服务假死的问题排查
burg_xun | 2年前 | 2.1k 10 4 Java 后端

Troubleshooting系列-一次基于okhttp外部接口调用超时问题分析
技术驿站 | 10月前 | 1.5k 5 评论 后端 面试 Java

记一次Go net库DNS问题排查
保护我方李元芳 | 3年前 | 2.3k 2 1 Go

ACK Net Exporter 与 sysAK 出击：一次深水区的网络疑难问题排查经历
阿里云云原生 | 1年前 | 1.6k 2 评论 云原生 容器

java浅拷贝BeanUtils.copyProperties引发的RPC异常
京东云开发者 | 4月前 | 130 点赞 2 后端

java浅拷贝BeanUtils.copyProperties引发的RPC异常
京东云开发者 | 6月前 | 213 5 评论 后端

Spring-Security oauth2 设置永久token踩的坑
董_董 | 3年前 | 3.0k 35 3 后端 Spring

记一次 .NET 某供应链WEB网站 CPU 爆高事故分析
一线码农聊技术 | 2年前 | 738 点赞 评论 后端

Logback日志滚动陷阱
spilledyear | 4年前 | 2.1k 5 2 Java

年轻代频繁ParNew GC，导致http服务rt飙高
烂猪皮 | 4年前 | 1.6k 6 3 Java

java浅拷贝BeanUtils.copyProperties引发的RPC异常 | 京东物流技术团队
京东云开发者 | 1年前 | 1.1k 7 2 Java 后端