厦門大學

软 件 学 院

《中间件技术》大作业实验报告

赖博阳 24320152202757　　阮逸松 24320152202802

徐畅 24320152202835　　林欢 24320152202773

危乃鑫 24320152202823　　刘爱琪 24320152202778

张舒婷 24320152202857　　肖昶辉 24320152202834

2018 年 6 月 4 日

目录

一、项目介绍

1.1、项目概要

翻转课堂是一种新的学习模式，是对传统模式的一种颠覆，在诞生以后，就表现出强大的优势。翻转课堂目前变得越来越普遍。然而，翻转课堂的管理工作琐碎又耗时，如课堂点名、主题发布、学生分组、学生每次讨论的展示成绩汇总和计算、老师对每次课后作业的分数统计等事务，如果完全采用人工方式处理，那么不仅浪费老师和学生的时间和精力，在工作处理中也存在效率低，失误难以避免等问题。针对这些问题，我们需要解放人力，减轻老师和助教工作压力，进行课堂管理系统的开发，以期促进课堂管理的高效、便利进行。

然而，在实际应用中，该系统对高并发的支持度有限，难以实现普及到全校，甚至全国的推广。因此我们决定利用中间件技术将该系统改为分布式运行，以提高事务处理能力和响应速度，更具推广和使用价值。

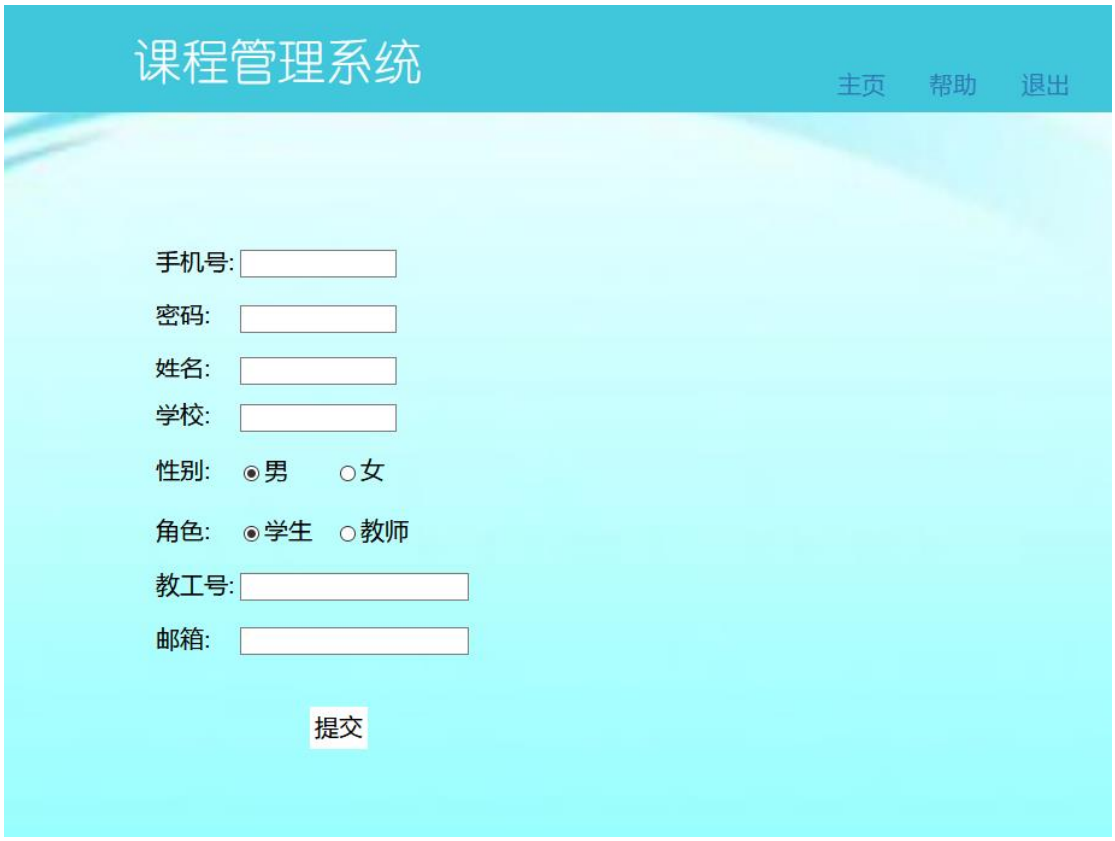
1.2、系统功能

本系统需具备的主要功能如下：

1) 课程、班级、讨论课的创建管理及选课退课
2) 设定讨论话题及分组
3) 课堂点名
4) 组队课堂展示的选题、打分
5) 课后报告的提交、打分

1.3 原有系统界面展示

1.3.1Web 端：



1.3.2 小程序端：

1.4、系统不足

根据现有的选课系统的使用经验，选课系统面对压力主要是开放选课、课堂签到等短时高并发的访问。在这种场景下，选课系统常常会出现无响应、报错甚至崩溃的情况。我们需要针对这个场景针对性的优化我们的系统。

经过分析，我们需要衡量的指标有：

每秒查询量（QPS）：服务器响应错误率为 0 的情况下每秒可承受的 RESTFul API 请求数的最大值。

最大并发数（Max Concurrency）：服务器瞬时承载 API 请求的最大值。反映系统最大可以负载的用户量。以响应错误率达到 1%时的并发数为测量指标。

最坏响应时间（Worst Case Latency）：服务器在高负载下可能出现的响应时间的最大值。反映用户感受到的延迟。以达到 QPS 时的 P99 延迟为测量指标。

我们将模拟本系统中以下几个常见的使用场景所需要的 API 请求序列，对几种系统架构进行测试并比较它们的性能。

1. 用户登陆
2. 课程查询课程选课
3. 学生签到
4. 课程选课
5. 新建课程

1.5、改进方向

原有的课程管理系统无法满足高并发量情况下的业务需求，我们考虑采用分布式框架对原有的系统进行改进，将课堂管理系统的功能模块进行分布式处理，在不改变现有的数据库结构和业务逻辑的基础上，使用分布式框架动态调度服务。在系统运行的时候，将会根据实际情况动态考虑在哪台服务器上开启什么模块，以达到最佳的系统运行效果。

我们的目标是，使用分布式框架之后，在 500 用户并发量的情况下，测试的平均响应时间小于 15 秒； 100 用户并发量的情况下，事务吞吐量在 100QPS 的水平。

目前考虑使用 Orleans 框架对现有项目进行改造，实现分布式操作。

二、方法介绍

2.1、异步调用

异步调用允许进程不用阻塞当前线程来等待处理完成，而可以进行后续操作，直到其他线程处理完成，再回调通知此线程。

异步对可能会被屏蔽的活动（如 Web 访问）至关重要。对 Web 资源的访问有时很慢或会延迟。如果此类活动在同步过程中被屏蔽，整个应用必须等待。在异步过程中，应用程序可继续执行不依赖 Web 资源的其他工作，直至潜在阻止任务完成。

异步调用的优点如下：

1. 避免阻塞、减少需要的进程数
2. 异步调用缺点：
3. 有额外的 CPU 开销、对耗时相对较短的调用反而响应更慢

2.2、Orleans 框架

Orleans 是在.NET 平台上的一种构建分布式、高规模（伸缩）的应用程序，它就是为了分布式、并发而生，因此可以解决课堂管理系统中大并发、高用户量的问题。

其中，核心角色为 Grains、Silos、Client。Grains 可以理解为一个服务，是主要的业务逻辑实现与抽象；Silos 是一个 server，用于存储 Grains，并等待调用；Client 则为具体的客户端。

在本项目中，Orleans 框架如下，Silo 包为 gRPC 的服务器，service 被分配到不同的 RPC 服务器上，实现分布式集群。相关代码如下：

```
private static void Main()
{
    _silo = new SiloHostBuilder()
        .UseAdoNetClustering(options =>
        {
            options.Invariant = "MySql.Data.MySqlClient";
            options.ConnectionString = _Configuration.GetConnectionString("MYSQL57");
        })
        .Configure<ClusterOptions>(options =>
        {
            options.ClusterId = "xmu.crms.silo";
            options.ServiceId = "xmu.crms.silo.services";
        })
        .ConfigureEndpoints(GetLocalIpAddress(), 11111, 30000)
        .ConfigureServices(svc =>
        {
            svc.AddDbContextPool<CrmsContext>(options =>
                options.UseMySQL(_Configuration.GetConnectionString("MYSQL57"))
            );
            svc
                .AddViceVersaClassDao()
                .AddViceVersaClassService()
                .AddViceVersaCourseDao()
                .AddViceVersaCourseService()
                .AddViceVersaGradeDao()
                .AddViceVersaGradeService()
                .AddHighGradeSchoolService()
                .AddHighGradeSeminarService()
                .AddInsomniaFixedGroupService()
                .AddInsomniaLoginService()
                .AddInsomniaSeminarGroupService()
                .AddInsomniaTopicService()
                .AddInsomniaUserService();
        })
        .ConfigureApplicationParts(parts =>
            parts.AddApplicationPart(typeof(HighGradeExtensions).Assembly).WithReferences())
        .ConfigureApplicationParts(parts =>
            parts.AddApplicationPart(typeof(ViceVersaExtensions).Assembly).WithReferences())
        .ConfigureApplicationParts(parts =>
            parts.AddApplicationPart(typeof(InsomniaExtensions).Assembly).WithReferences())
        .ConfigureLogging(builder => builder.SetMinimumLevel(LogLevel.Warning).AddConsole())
        .Build();

     Task.Run(StartSilo);


    AssemblyLoadContext.Default.Unloading += context =>
    {
        Task.Run(StopSilo);
        _SiloStopped.WaitOne();
    };

    _SiloStopped.WaitOne();
}

private static async Task StartSilo()
{
    await _silo.StartAsync();
    Console.WriteLine("Silo started");
}

private static async Task StopSilo()
{
    await _silo.StopAsync();
    Console.WriteLine("Silo stopped");
    _SiloStopped.Set();
}
```

在 Insomnia 包中，ClassController 可以作为 Client，调用对应的 RPC 服务，相关代码如下：

```
public ClassController(CrmsContext db, IClusterClient client)
{
    _db = db;
    _courseService = client.GetGrain<ICourseService>(0);
    _classService = client.GetGrain<IClassService>(0);
    _userService = client.GetGrain<IUserService>(0);
    _fixGroupService = client.GetGrain<IFixGroupService>(0);
    _seminarService = client.GetGrain<ISeminarService>(0);
}
```

通过 Orleans 框架，成功将 service 分到多个服务器上进行分布式集群计算，提高了运行性能。

采用分布式的优点如下：

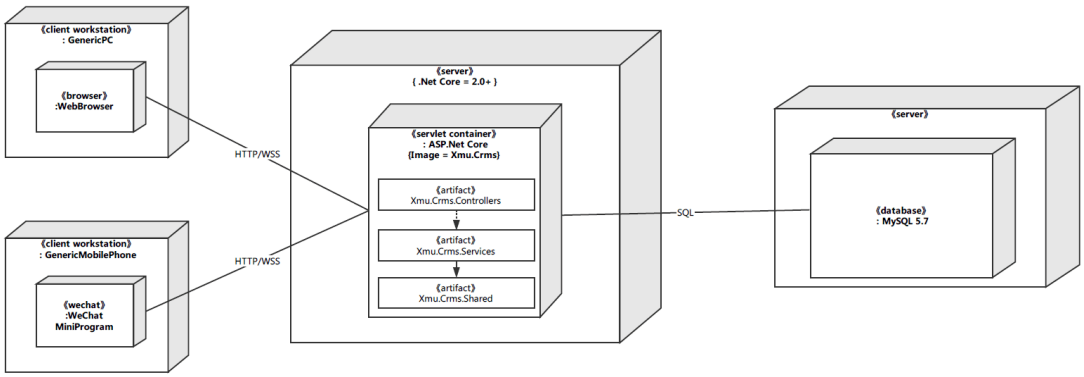
1. 可靠性、高容错性：一台服务器的系统崩溃不会影响到其他的服务器。
2. 可扩展性：在分布式计算系统可以根据需要增加更多的机器。
3. 灵活性：可以很容易的安装、实施和调试新的服务。
4. 计算速度快：分布式计算机系统可以有多台计算机的计算能力，使得比其它系统有更快的处理速度。
5. 开放性：由于是开放的系统，本地和远程都可以访问到该服务。
6. 高性能：相较于集中式计算机网络集群可以提供更高的性能，及更好的性价比。

采用分布式的缺点如下：

1. 网络基础设施成本高：网络基础设置问题，包括传输、高负载、信息丢失问题。
2. 有额外的序列化、反序列化开销

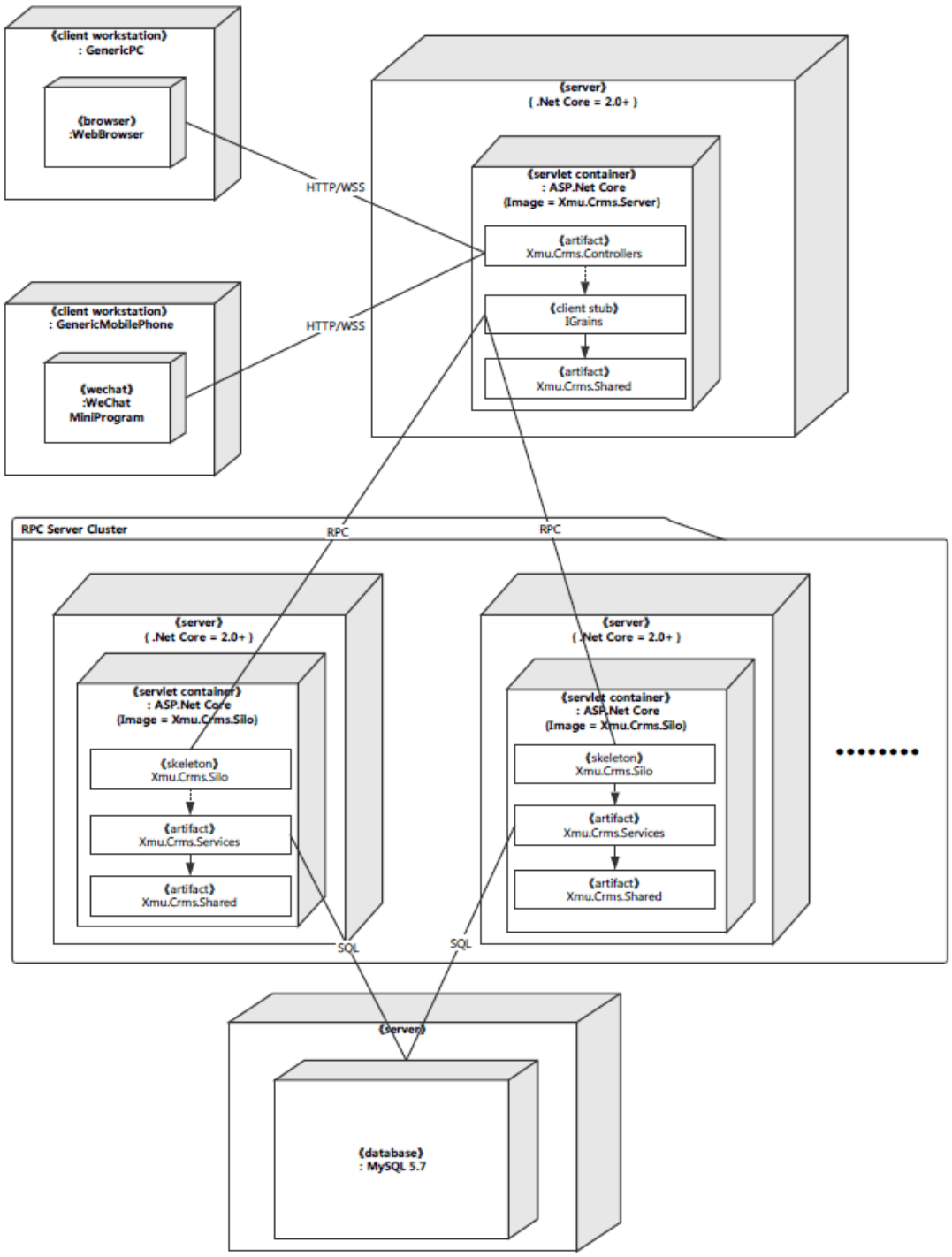
三、总体设计

原系统部署图：



如图，原始方案和异步调用方案中，controller 和 service 位于相同服务器上。主服务器通过 http 协议和 wss 协议与客户端交互，通过 SQL 连接与数据库服务器交互，进行数据库操作。

采用 Orleans 框架后的部署图：



采用 Orleans 框架进行分布式部署后，service 被分配到 RPC 服务器上。主

服务器中的 controller 调用 Client Stub，由 Client Stub 在分布式集群中找到对应的 RPC 服务，再通过 RPC 服务调用 services，与数据库服务器进行相应操作。该部署方案实现了 service 和 controller 的分离，并将不同 service 分布到了不同的服务器上。

四、性能改进

表 1 测试数据为系统优化前（origin ）、异步化（async）、分布式（orleans）分别在 25、50、100、200、400、500 用户数并发的情况下的事务平均延迟（毫秒）、吞吐量（完成事务/秒）数据。测试事务包含了用户登陆（简单数据库读/CPU 密集的哈希计算）、课程查询（复杂数据库读/多表连接）、课程添加（复杂数据库写）、学生签到（简单数据库写）。测试中，每个事务需要连续完成上述操作的 API 请求，每个用户重复执行 10 次事务；虚拟用户的 think time 和 ramp-up time 均为 0，以最大化对系统的压力。作为对比，普通用户手动操作完成一次该事务需要约一分钟。

测试环境：

PC x2:

CPU: E3-1240v5

RAM: 32G

OS: Windows 10 v1709 (FCU)

.Net SDK: 2.1.300-rc1-008673

DB: MySQL 8.0

用户数	系统状态	延迟	吞吐量
25	async	579	39.74563
	origin	523	44.61099
	orleans	555	38.7657
50	async	413	109.36133
	origin	847	55.4078

	orleans	397	101.41988
100	async	2170	44.28698
	origin	3205	29.63051
	orleans	638	120.59817
200	async	7582	25.62427
	origin	11618	15.46599
	orleans	2210	66.39225
400	async	17966	21.56543
	origin	17462	21.95582
	orleans	9263	34.65634
500	async	25910	17.90837
	origin	26534	18.05347
	orleans	14615	28.19554

表 1 测试数据

我们对系统不同状态下的事务平均延迟、吞吐量数据进行可视化处理，得到图 1 及图 2。

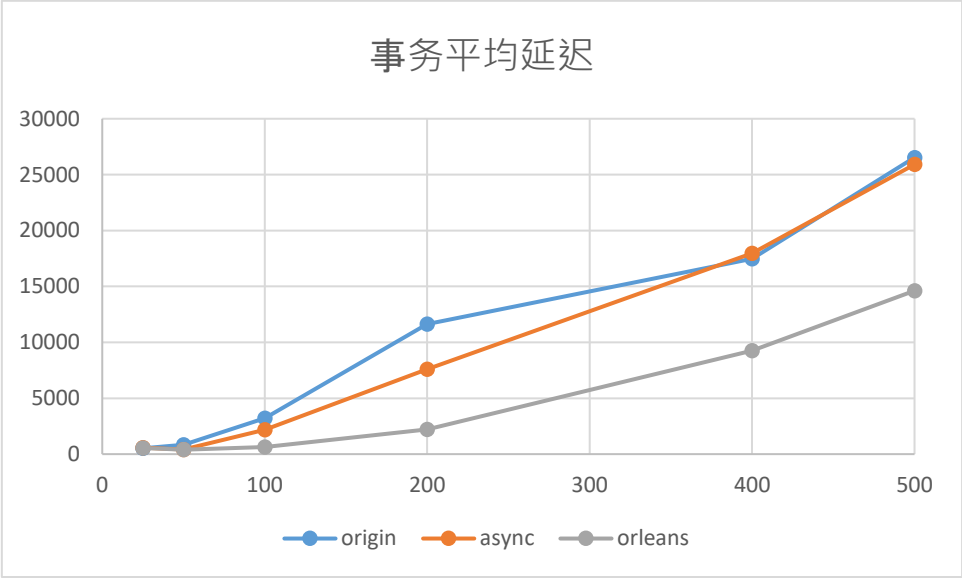


图 1 事务平均延迟

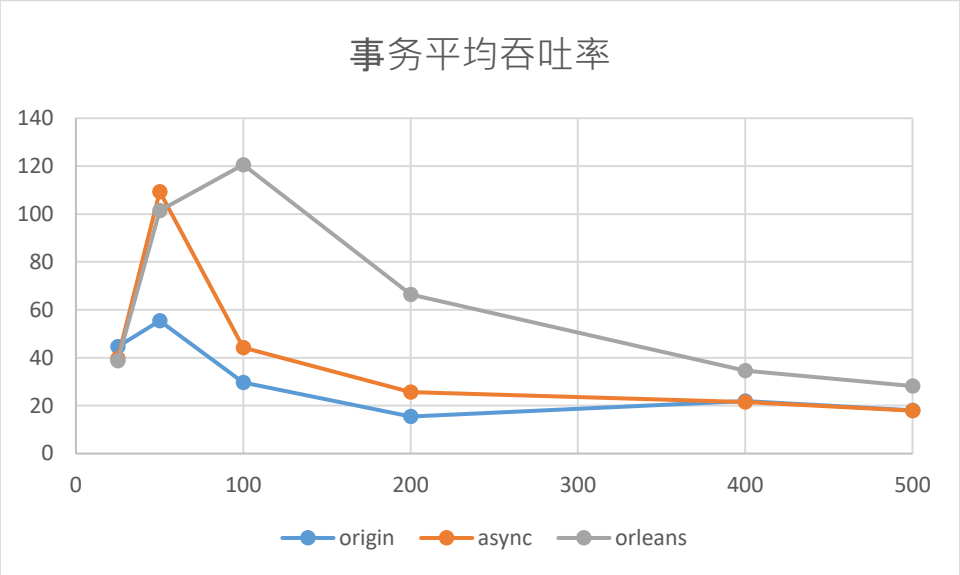


图 2 事务平均吞吐率

由图 1 事务平均延迟我们可以看出在事务平均延迟方面，总体上分布式<异步化<优化前；由图 2 事务平均吞吐量我们可以得到：在事务平均吞吐量方面，总体上优化前<异步化<分布式。由此，我们可以得到结论：分布式系统的性能优于异步化系统性能，同时异步化系统性能优于优化前系统性能。采用 Orleans 框架进行分布式部署的系统在事务平均延迟方面平均减少 50%的时间，同时吞吐量平均提升 1.36 倍。

直观感受上，在 500 虚拟用户的并发情况下，事务的总延迟为十五秒，针对单个 API 请求的延迟不超过 3 秒。且在实际情况下，用户的 think time 和 ramp-up time 均远大于 0，意味着实际服务 500 用户时，系统表现将优于现有测试结果。

未来我们将进行更多的测试寻找性能瓶颈，如网络 IO、数据库、磁盘 IO、CPU 等。我们还将会从服务器负载均衡、数据库查询优化、分布式架构设计等角度扩展优化思路。

五、成员分工（排序）

赖博阳：系统架构选择、架构设计、系统异步化实现

徐畅：架构设计、系统异步化实现、系统分布式实现

危乃鑫：系统异步化实现、异步化测试

张舒婷：异步化测试、分布式测试

阮逸松：基准测试、系统文档、展示

林欢：数据分析、报告、PPT

刘爱琪：数据分析、报告、PPT

肖昶辉：基准测试、系统文档

六、总结

总而言之，在项目过程中，我们采用 Orleans 框架进行分布式部署后，系统性能得到了很大程度上的提升，在事务平均延迟方面平均减少 50%的时间，同时吞吐量平均提升 1.36 倍。期待未来可以通过改进系统使得系统整体性能得到进一步的提升。