后端性能优化

李华卿

- 寻找瓶颈
- 解决瓶颈
- 预防瓶颈

寻找瓶颈

- 1.推测：经验判断 & review代码

- 2.验证：查询日志（CAT、MTrace、XmdLog）、

工具检查（Jprofile、Jmap、Jstack、Linux命令）

解决性能瓶颈

性能是什么?

一个程序对内存和时间的需求称为程序性能。 ——时间、空间复杂度

性能是在规定技术指标及相关约束条件下产品功能特质的必然表现或反映。 ——响应时间、吞吐量、资源使用率等

性能是系统运行的资源开销。

资源有哪些?

计算资源

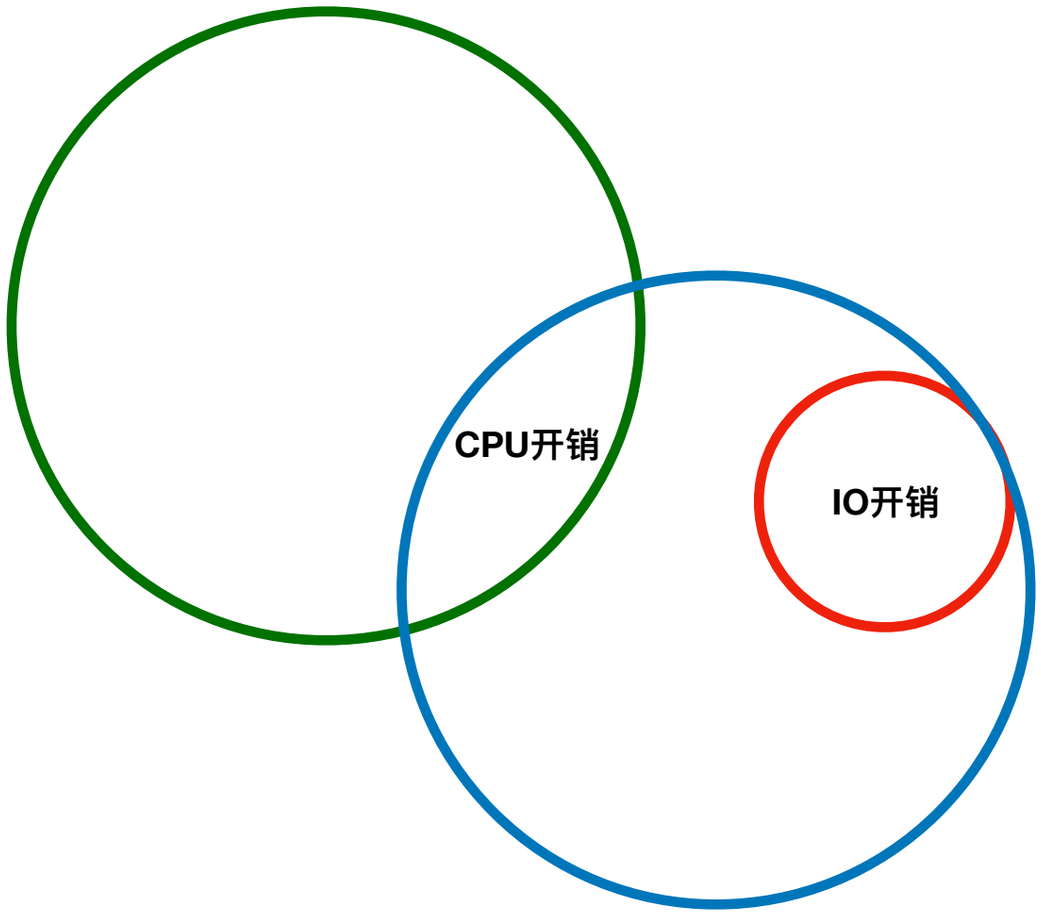
空间资源

CPU分片

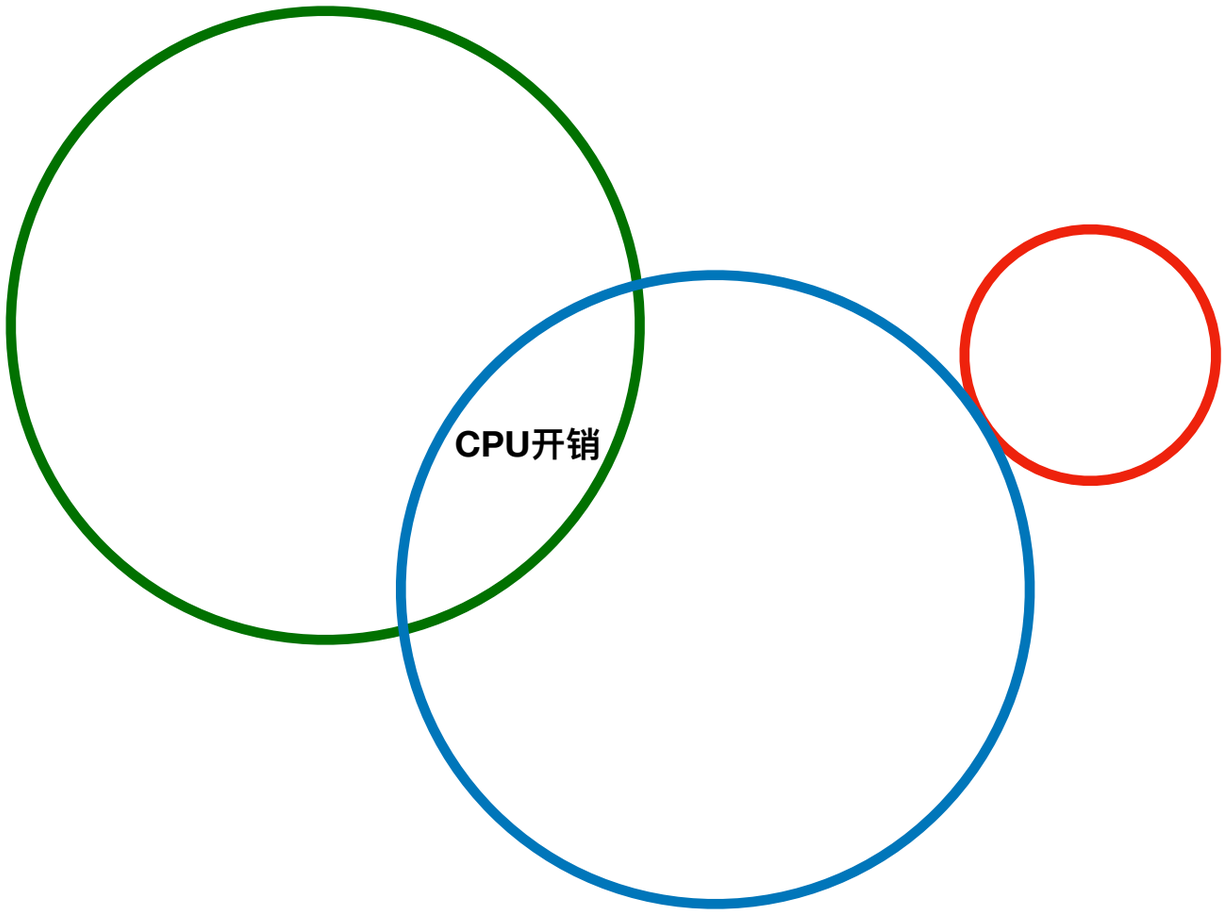
IO

时间资源

一个透传的API接口资源使用模型



增加缓存后的资源使用模型



资源利用率存在短板效应，任意资源利用率过高都会引起问题：

计算资源瓶颈：Cpu idle过低
空间资源瓶颈：OOM、空间不足
时间资源瓶颈：积压、cpu wait

因此性能优化本质是平衡各项资源利用率

解决时间资源瓶颈：空间换时间

适用情景：网络、磁盘IO瓶颈，吞吐并发有压力

对策：缓存、冗余

缓存：应用层、避免IO、非持久化
冗余：数据层、减少IO、持久化

缓存

解决方案1：本地内存缓存 Guava Cache、ConcurrentHashMap

优势：直接内存速度最快、注解式使用方便、LRU自带内存管理

劣势：无法解决分布式一致性问题、缺少管理工具、无法持久化

```
private LoadingCache<Integer, Integer> mtDealIdToDpGroupIdCache = CacheBuilder.newBuilder()
        .expireAfterWrite(1, TimeUnit.DAYS).build(new CacheLoader<Integer, Integer>() {
            @Override
            public Integer load(Integer mtDealId) throws Exception {
                return getDpGroupId(mtDealId);
            }
        });
```

解决方案2：缓存服务 Squirrel

优势：解决分布式一致性问题、管理监控方便、能够持久化、空间大、高可用

劣势：需要微量网络IO

```
@Cacheable(value = "templateCache", key = "'template_id_'+#templateId")
public DispatchTemplate getById(Integer templateId){
    return dispatchTemplateMapper.selectByPrimaryKey(templateId);
}
```

冗余

数据库层面：字段冗余、索引

优势：减少join、查询次数

劣势：违背第三范式，牵一发动全身

系统层面：数据冗余

优势：IO可控

劣势：一致性问题

e.g.美团订单中心 VS 点评订单中心

解决时间资源瓶颈：计算换时间

适用情景：CPU使用率低，而任务较重或吞吐大

对策：并行、并发、异步

并行：多线程执行同一级别的任务

并发：多线程执行同样的任务

异步：非主流程异步化

并行　　适用情景：一个任务完成需要多个子任务，每个子任务互不依赖

解决方案1：Future

```
public static class Task implements Callable<String> {
    @Override
    public String call() throws Exception {
        String tid = String.valueOf(Thread.currentThread().getId());
        System.out.printf("Thread#%s : in call\n", tid);
        return tid;
    }
}

public static void main(String[] args) throws InterruptedException, ExecutionException {
    List<Future<String>> results = new ArrayList<~>();
    ExecutorService es = Executors.newCachedThreadPool();
    for(int i=0; i<100;i++)
        results.add(es.submit(new Task()));

    for(Future<String> res : results)
        System.out.println(res.get());
}
```

```
Thread#60 : in call
Thread#59 : in call
Thread#56 : in call
Thread#57 : in call
Thread#55 : in call
Thread#54 : in call
Thread#53 : in call
Thread#52 : in call
```

```
Thread#16 : in call
Thread#15 : in call
Thread#14 : in call
Thread#11 : in call
Thread#13 : in call
11
12
13
14
15
16
17
18
19
20
```

劣势：无法用Spring管理

并行

解决方案2：Spring @Async

```
private TaskRequirementValidateParam build(DispatchTask dispatchTask, DispatchTemplate dispatc
    Set<TaskValidateParamTypeEnum> paramTypeSet = buildParamTypeSet(dispatchTemplate);
    TaskRequirementValidateParam param = new TaskRequirementValidateParam();
    Future<PoiVendor> poiVendorFuture = null;
    Future<PoiKdb> poiKdbFuture = null;
    Future<OrderDetailDTO> orderFuture = null;
    if(paramTypeSet.contains(TaskValidateParamTypeEnum.POI_VENDOR)){
        poiVendorFuture = taskRequirementParamBuilder.getPoiVendor(dispatchTask.getPoiId());
    }
    if(paramTypeSet.contains(TaskValidateParamTypeEnum.POI_KDB)){
        poiKdbFuture = taskRequirementParamBuilder.getPoiKdb(dispatchTask.getPoiId());
    }
    if(paramTypeSet.contains(TaskValidateParamTypeEnum.ORDER_INFO)){
        orderFuture = taskRequirementParamBuilder.getOrder(dispatchTask.getOrderId());
    }
    if(poiVendorFuture != null){
            param.setPoiVendor(poiVendorFuture.get());
    }

    if(poiKdbFuture != null){
            param.setPoiKdb(poiKdbFuture.get());
    }
    if(orderFuture != null){
            param.setOrderParam(buildOrderParam(orderFuture.get()));
    }
    return param;
}
```

并行

解决方案2：Spring @Async

```
@Async
public Future<PoiVendor> getPoiVendor(Integer poiId){
    try {
        Thread.sleep(7000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    List<PoiVendor> poiVendors = poiVendorService.getActive
    PoiVendor poiVendor = CollectionUtils.isEmpty(poiVendor
    LOGGER.info("get poivendor ready!");
    return new AsyncResult<>(poiVendor);
}

@Async
public Future<PoiKdb> getPoiKdb(Integer poiId){
    try {
        Thread.sleep(5000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    PoiKdb poiKdb = poiKdbService.getActivePoiKdb(poiId);
    LOGGER.info("get poiKdb ready!");
    return new AsyncResult<>(poiKdb);
}
```

```
(TaskRequirementParamBuilder.java:70) get order ready!
(TaskRequirementParamBuilder.java:58) get poiKdb ready!
(TaskRequirementParamBuilder.java:46) get poivendor ready!
```

需注意异常处理及事务

并发　　适用情景：任务耗时方差较大或对响应时间要求严格

e.g. Pigeon Forking策略：

Pigeon提供了Forking的策略，支持同时向N个服务器发送请求、使用最先返回的响应

e.g. Squirrel请求策略：

超时默认50ms，先等10ms，没有响应再发送请求

e.g. 快速失败策略：

Timeout设置为TP99时间

异步　　适用情景：非主流程耗时较多

解决方案1：Thread(Pool)　　脱离Spring管理

解决方案2：Guava EventBus　　适用于class内

```java
/**
 * STEP1:将calls放入加锁线程池中
 * @param calls
 */
public void putInQueue(List<Call> calls) { lockExecutor.post(calls); }

/**
 * STEP2:加锁并将call放入threadpool中等待执行
 * @param calls
 */
@Subscribe
@AllowConcurrentEvents
@Transactional
public void lockCalls(List<Call> calls){
    try{
        calls.stream().forEach(call -> {
            if(lockCall(call)){
                logger.info("lock call success call is {}",call.getId());
                CatTool.eventSuccessed(CatEventEnum.CALL_LOCKED);
                callExecutor.post(call);
            }else{
                CatTool.eventFailed(CatEventEnum.CALL_LOCKED,String.valueOf
                logger.info("lock call failed call is {}",call.getId());
            }
        });
    }catch(Exception e){
        logger.error("fail lock call error : {}" ,e);
    }

}
```

异步

解决方案3：Spring Async　　适用于class外

```java
@Service
@EnableAsync
public class NewTaskAsyncListener implements ApplicationListener<NewTaskAsyncEvent> {
    private static final Logger logger = LoggerFactory.getLogger(NewTaskAsyncListener.class);

    @Resource
    private ExecuteTaskAction executeTaskAction;

    @Override
    @Async
    public void onApplicationEvent(NewTaskAsyncEvent event) {
        DispatchTask dispatchTask = (DispatchTask) event.getSource();
        try{
            executeTaskAction.doAction(dispatchTask);
        }catch(Exception e){
            logger.error("execute task occur error  task id is {}", dispatchTask.getId(),e);
        }

    }
}
```

解决计算资源瓶颈：时间换计算

适用情景：任务时效性低，实时计算资源紧张、开销大

对策：批处理、限流

批处理：为开销大的任务指定批处理策略

限流：限制吞吐、QoS

批处理　　前提：任务批处理边界成本低

e.g. 订单用餐提醒

由job驱动统一发送一段时间内的提醒，而非每个任务定时执行

e.g. Redis持久化策略、Log4j上报策略

达到一定数量或超过一段时间后，统一进行处理

限流 - 限制调用量

解决方案：原子自增计数AtomicInteger实现

```
private static final int MAX_COUNT = 20;

private AtomicInteger filterCount = new AtomicInteger();

public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain) throws IOException, ServletException {
    if(filterCount.intValue() > MAX_COUNT) {
        //请求个数太多，跳转到排队页面
        request.getRequestDispatcher("index.jsp").forward(request, response);
    } else {
        //请求个数加1
        filterCount.incrementAndGet();
        chain.doFilter(request, response);
    }
}
```

限流 - 限制并发量

解决方案1：令牌策略（Token Bucket）

e.g. Redis incr/descr 实现令牌分发/回收

解决方案2：pool、client

e.g. TheadPool、c3p0、HttpClient

```
public static HttpClient create() {
    if (defaultClient == null) {
        defaultClient = HttpClientBuilder.create()
                .setUserAgent("resv-cc HTTP Client")
                .setRetryHandler(new StandardHttpRequestRetryHandler(3, true))
                .setServiceUnavailableRetryStrategy(new DefaultServiceUnavailc
                .setMaxConnPerRoute(256)
                .setMaxConnTotal(1024)
                .build();
    }
    return defaultClient;
}
```

解决方案3：MQ限流

解决计算资源瓶颈：空间换计算

适用情景：计算开销大、输入不变，需要重复计算

对策：中间态数据

e.g. 星空统计：第一次查询计算出统计数据存储，后续查询直接展示

e.g. CC监控：将明细信息变成必要的统计信息方便规则过滤

解决计算资源瓶颈：计算换空间

适用情景：高IO、空间受限

对策：压缩

压缩

存储压缩(gz、jpeg、bitmap)

传输压缩　　序列化压缩(hession)
　　传输协议压缩(CompactRequest)

解决资源瓶颈：综合法 - 扩充资源

适用情景：单机资源达到瓶颈、无法协调或协调成本过高

对策：横向扩展、分而治之

横向扩展

数据库层面：读写分离（Zebra、Atlas）

应用层面：负载均衡（Pigeon、OCTO策略、MGW）

分而治之

分布式计算：MapReduce（Hadoop、Spark）

分布式任务：Quartz、Crane

分布式存储：分库分表（zebra、Atlas）

解决资源瓶颈：非技术手段

适用情景：复杂流程、依赖项多

对策：流程重塑、功能解耦、依赖反转、打散收敛

预防瓶颈

- 1.监控：Falcon、CAT

- 2.压测：PTest、Jmeter、Quake(PTest2.0)