

开源博客

[+ 写博客](#)

大家都在搜...



京东云开发者的个人空间 / 案例分享 / 正文

万字好文：大报文问题实战 | 京东云技术团队

原创 京东云开发者 ✨ 案例分享 07/07 09:41 阅读数 5.5K

 本文被收录于专区
开发技能
[进入专区参与更多专题讨论 >](#)

关于作者



京东云开发者 ✨

[关注](#)[私信](#)[提问](#)

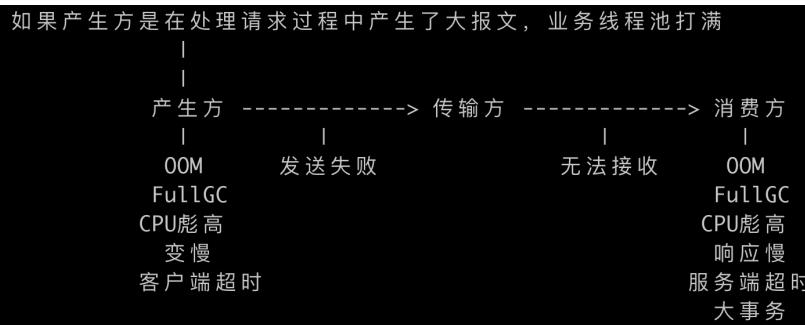
文章	经验值	粉丝	关注
1.2K	3.7W	959	5

导读

大报文问题，在京东物流内较少出现，但每次出现往往是大事故，甚至导致上下游多个系统故障。大报文的背后，是不同商家业务体量不同，特别是 B 端业务的采购及销售出库单，一些头部商家对京东系统支持业务复杂度及容量能力的要求越来越高。因此我们有必要把这个问题重视起来，从组织上根本上解决。

1 认识大报文问题

大报文问题，是指不同的系统通过网络进行数据交互时 payload size 过大导致的系统可用性下降问题。



对于大报文的产生方，过大的报文在序列化时消耗更多内存和 CPU，在传输时 (JSF/MQ) 可能超过中间件的大限制导致传输失败；对于大报文的消费方，过大的报文在反序列化时会产生大对象，消耗更多的内存和 CPU，容易触发 FullGC 甚至 OOM，而在处理过程中要遍历的内容更多，造成响应变慢，如果涉及数据库操作容易产生大事务、慢 SQL，这些容易触发超时，如果客户端有重试机制，会进一步加重大报文消费方负载，严重时导致服务集群整体不可用。

此外，由于大报文与小报文是在一个接口上完成的，使用相同的 UMP key，它会导致监控失真，报警阈值无效。如果日志记录了原始报文，也可能磁盘打满和响应变慢。

在京东物流技术体系内，具体表现为：

大报文场景	后果
MQ 的 producer 发送了大的 Message	由于 JMQ 对消息大小的限制，导致 producer 发送失败：消息未送达
MQ consumer 反序列化 Message 并处理计算时产生大对象，频繁 FullGC，CPU 使用率飙升	
JSF Consumer 调用 API 时传入大入参值	由于 JSF Server 对 payload 大小限制，导致服务端将报文抛弃：无法送达
JSF Provider 响应变慢，产生大对象，频繁 FullGC，CPU	

作者的专辑

全部

- 源码分析 (3)
- AI大语言模型 (6)
- 测试 (13)
- 性能优化 (7)

作者的其它热门文章

- 亿级流量背后战场，京东11.11大促全方...
- 数千个数据库、遍布全国的物理机，京...
- 错过就要再等一年，如何在大促中获得...
- 京东智联云与CDA携手 共同打造电商领...
- 我在北京做研发 | 【混合多云第一课】...

热门资讯

- 1 [基于 NT 架构的全新 QQ Windows 版正式发布](#)
- 2 [爱奇艺客户端“白嫖”电视机，后台满速上传](#)
- 3 [美国将限制中国使用亚马逊、微软等提供训练 AI 模型的云服务](#)
- 4 [2023 年收入最高的技术岗位](#)
- 5 [资金严重短缺，又一流行开源项目宣布停止功能开发](#)
- 6 [LeaferJS 发布：开源、性能强悍的 2D 图形库](#)
- 7 [Threads 注册量已破三千万，后端基于 CPython 深度“魔改”](#)
- 8 [Visual Studio Code 1.80 发布，支持终端图片功能](#)
- 9 [deepin 采用 Asahi Linux 适配 Apple M1](#)
- 10 [7 月数据库排行：Oracle 大涨，再度拉开比分](#)



推荐关注

JSF Provider 返回值包含大对象

由于 JSF Consumer 对 payload 大小限制，导致 consumer 无法获取响应

JSF Consumer 产生大对象，频繁 FullGC，CPU 使用率飙升，甚至 OOM

◆ JMQ/JSF 对 payload 大小的限制都属于防御性保护措施，目前的值是科学的，它们都已经足够大了。在紧急止血情况下可以调整配置参数来暂时提高 payload 大小限制，但长期看它会加重系统的风险，应该从设计入手避免超过 payload 大小限制。

1.1 背景知识

1.1.1 JMQ 限制

根据 JMQ 的官方文档，单条消息大小：JMQ4 不要超过 4M，JMQ2 不要超过 2M。

具体原理是发送消息时在生产端做主动校验，如果消息大小超过阈值则抛出异常（代码实现与官方文档不一致）：

```
class ClusterManager {
    protected volatile int maxSize = 4194304; // 4MB
}

class MessageProducer implement Producer { // Producer接口的具体实现类
    ClusterManager clusterManager;

    // producer.send时做校验
    int checkMessages(List<Message> messages) {
        int size = 0;
        for (Message message : messages) {
            size += message.getSize() // 压缩后的大小
        }
        if (size > this.clusterManager.getMaxSize()) {
            throw new IllegalArgumentException("the total bytes of message body must be less than " + this.
        }
    }
}
```

◆ 经与 JMQ 团队确认，JMQ 消息大小的限制，以代码实现为准（官方文档不准确）：



1.1.2 JSF 限制

根据 JSF 官方文档，JSF 可以在 server 和 consumer 端分别设置 payload size，默认都是 8MB。

数据包大小限制

默认数据包大小为 8M，即服务的请求和返回序列化后默认不超过 8M，要不然数据接收方会丢弃此数据。
数据较大，建议开启调用压缩，如果压缩后数据还大，可以自行定义数据包配置。

如果请求数据包较小，则服务调用者进行配置。

注意：由于 JSF 是长连接使用的，请把特别配置的 consumer 放到其它 consumer 之前加载，防止配置覆盖。
<jaf:consumer payload="16777216" /> <!-- 默认是8388608=8*1024*1024 -->

调用压缩

在 Consumer 故送请求和 provider 返回响应的时候，都可以开启调用压缩。

目前支持多种方法：snappy, lzma,

1. 如果 consumer 配置了压缩，且请求的数据大于 2048B，那么请求数据将被压缩后再发给 provider。

<jaf:consumer compress="snappy" />

服务端针对压缩过的请求，返回响应的时候也会判断是否启动压缩。（但是：虽然客户端配了压缩，如果请求小于 2048B，还是需要服务端配置压缩才触发）

2. 如果 provider 配置了压缩，那么不管请求时是不是带压缩标识，返回响应的时候都会被压缩。如果数据大小超过阈值，则将响应数据压缩后重发给 consumer。

<jaf:provider compress="snappy" />

最佳实践：请求数据大的客户端配置，响应数据大的服务端配置。

◆ 需要注意，触发 provider 报文长度限制时，JSF consumer (老版本) 并不会立即失败，而是依靠客户端超时

（[六十分钟 / 或当且仅当 ICF 的缺陷](#)）。具体原因：JSF 依靠底层 netty 来实现报文长度限制，当 provider 从请求报文头



纯洁的微笑

文章 57 访问 23.5W



longforus

文章 5 访问 1.4K



志成就

文章 246 访问 40.9W



SOFASStack

文章 726 访问 40.5W



MyCAT

文章 2 访问 1.5K

热门软件

- GraphicsFuzz - 图形着色器测试框架
- uni-app - 基于 Vue.js 的跨平台框架
- SOAR - SQL 智能优化与改写工具
- SOFAJRaft - 基于 RAFT 一致性算法的 Ja...
- delicate - Rust 编写的分布式任务调度平...
- beeshell - React Native 应用基础组件库
- Firecracker - 结合硬件虚拟化安全性与容...
- OpenHMD - 用于 VR 开发的开源项目
- pdf.tocgen - PDF 目录生成命令行工具
- Scorecards - 开源项目安全性评分应用
- pacific - 方舟编译器的 Runtime 参考实现
- dnSpy - .NET 调试器和汇编编辑器
- Eunomia-bpf - WASM 模块或 JSON 中...
- DeFiBus - 分布式金融级消息总线
- RedFish - 跨平台 Redis GUI 客户端
- Auto-GPT - GPT-4 自动化工作项目
- MISP - 开源威胁情报和共享平台
- TensorSpace - 神经网络 3D 可视化框架
- MoliCode - 多语言代码生成器
- doodooke - 多多客是一款支持微信，百...



TooLongFrameException，而该异常的处理依赖 netty 的 ChannelHandler.caught 方法，SSP 里没有对 TooLongFrameException 做处理（吃掉异常），provider 端不给 consumer 任何响应（请求被扔进黑洞），因此造成 consumer 一直等待响应直到超时，而这可能把 consumer 端的业务线程池拖死。

```
class LengthFieldBasedFrameDecoder { // 基于netty io.netty.handler.codec.LengthFieldBasedFrameDecoder的改动
    protected Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {
        // 从JMS协议的报文头里获取本次请求的payload size, 此时还没有读取8MB的body
        long frameLength = getUnadjustedFrameLength(in, actualLengthFieldOffset, lengthFieldLength, byteOrder);
        if (frameLength > maxFrameLength) { // maxFrameLength即8MB限制
            throw new TooLongFrameException();
        }
    }
}

class ServerChannelHandler implements ChannelHandler {
    public void exceptionCaught(ChannelHandlerContext ctx, final Throwable cause) {
        if (cause instanceof IOException) {
            // ...
        } else if (cause instanceof RpcException) {
            // 这里可以看到遇到这种异常, JMS是如何给consumer端响应的
            ResponseMessage responseMessage = new ResponseMessage(); // 给consumer的响应
            responseMessage.getMsgHeader().setMsgType(Constants.RESPONSE_MSG);
            String causeMsg = cause.getMessage();
            String channelInfo = BaseServerHandler.getKey(ctx.channel());
            String causeMsg2 = "Remote Error Channel:" + channelInfo + " cause: " + causeMsg;
            ((RpcException) cause).setErrorMsg(causeMsg2);
            responseMessage.setException(cause); // 异常传递给consumer
            // socket.write回consumer
            ChannelFuture channelFuture = ctx.writeAndFlush(responseMessage);
        } else {
            // TooLongFrameException会走到这里, 它的继承关系如下:
            // TooLongFrameException -> DecoderException -> CodecException -> RuntimeException
            // 异常被吃掉了, 不给consumer响应
            logger.warn("catch " + cause.getClass().getName() + " at {} : {}", NetUtils.channelToString(channel.remoteAddress(), channel.localAddress()), cause.getMessage());
        }
    }
}
```

◆ 经与 JSF 团队确认，consumer 端或 provider 端发出的消息过大（超过 payload）时 consumer 端得不到正确的异常响应只提示请求超时的问题，已经在 1.7.5 版本修复：需要 provider 端升级。升级后，如果 consumer 端发送的消息过大，provider 会立即响应 RpcException。

~~jsef 1.7.5 (2022.01.26)~~

bugfix:
1. Q E部分注册中心在网络丢包的情况下，客户端寻找到正常的注册中心后不能去正确的自动恢复
2. r 正在注册中心推送大量callback事件，客户调用callback逻辑的线程池被打满的场景下，注册中心不能收到推送消息失败通知的问题
3. 修复consumer端或provider端发出的消息过大（超过playload） consumer端得不到正确的异常响应，只报云请求超时的问题

此外，在 JSF 旧版本下，consumer 使用了默认的 5 秒超时，但 consumer 抛出超时异常总用时是 48 秒，这是为什么？

这是因为 consumer 配置的 timeout 不包括序列化时间，这 48 秒是把 8MB 的报文序列化的耗时

```
class JSFClientTransport {  
    // consumer同步调用provider  
    ResponseMessage send(BaseMessage msg, int timeout) {  
        MsgFuture<ResponseMessage> future = doSendAsyn(msg, timeout);  
        return future.get(timeout, TimeUnit.MILLISECONDS);  
    }  
  
    MsgFuture doSendAsyn(final BaseMessage msg, int timeout) {  
        final MsgFuture resultFuture = new MsgFuture(getChannel(), msg.getHeader(), timeout);  
        Protocol protocol = ProtocolFactory.getProtocol(msg.getProtocolType(), msg.getHeader().getCodec)  
        byteBuf = protocol.encode(request, byteBuf); // 发送报文前的序列化  
    }  
}
```

```

        channel.writeAndFlush(request, channel.voidPromise()); // socket.write, 异步IO
        resultFuture.setSentTime(JSFContext.systemClock.now());
    }
}

class MsgFuture implements java.util.concurrent.Future {
    final long genTime = JSFContext.systemClock.now(); // new的时候就赋值了
    volatile long sentTime;

    // 抛出超时异常逻辑
    ClientTimeoutException clientTimeoutException() {
        Date now = new Date();
        String errorMsg = "[JSF-22110]Waiting provider return response timeout . Start time: " + DateUtils.
            + ", End time: " + DateUtils.dateToMillisStr(now)
            + ", Client elapsed: " + (sentTime - genTime) // 它包括: 序列化时间, 由于异步IO因此不包括socket
            + "ms, Server elapsed: " + (now.getTime() - sentTime);
        return new ClientTimeoutException(errorMsg);
    }
}

```

1.1.3 物流网关限制

物流网关在 nginx 层通过 client_max_body_size 做了 5MB 限制。这意味着，JSF 限制了 8MB，但通过物流网关对外开放成 HTTP JSON API 时，调用者实际的限制是 5MB。

1.1.4 MySQL 限制

max_allowed_packet, net_buffer_length 等参数在底层控制 TCP 层的报文长度，京东物流体系内该值足够大，研发不必关注。

研发需要关注的是字段长度的定义，主要是 varchar 的长度。MySQL 通过 sql_mode 参数控制字段超过长度后的行为是字段截断还是中断事务。对于京东物流业务执行链路比较长的场景来讲，同一个字段可能多处保存，例如订单行里的 skuName，就会在 OFC/WMS 等系统保存，sku_name varchar 长度的不一致，特殊场景下可能造成上下游交互出现问题。

1.1.5 其他限制

DUCC value 的长度默认限制为 4W 字符。

UMP Key 的限制 128。

JMQ 的 businessId 长度限制 100，Producer 在发送是默认超时 2 秒，Producer 发送失败默认重试 2 次。

JMQ 消费者抛出异常会导致重试(进入 retry-db)，首次重试 10 分钟，如果重试还不成功会越来越慢推送直至过期。过期时间：JMQ2 为 3 天，JMQ4 为 30 天。

JSF 如果不配置 consumer timeout，则使用默认值：5 秒。

Zookeeper ZNode 限制长度 1MB。虽然可以通过 jute.maxbuffer 这个 Java 系统属性修改，但强烈不建议。

原则上，所有依赖的中间件都要确认其限制约束，提升健壮性，避免边界条件被触发而产生出乎意料的错误。

1.2 产生原因

1.2.1 集合类字段无约束

导致京东物流线上事故的大报文问题中，绝大部分都属于该类问题。而这又可以细分为两种场景：

```

interface JsfAPI {
    // 场景1：批量接口，对批量的大小无限制
    void foo(List<Request> requests);
}

```

`class Request {`

}

当数据量增大时，报文也会增大，造成几 MB 到几十 MB 的报文传输，系统为了处理这样大数据量的报文，必然会产生大对象，并且这种对象会一直处于内存中，在数据保存处理时，会造成内存不能释放，可能触发频繁 FullGC，CPU 使用率飙升。同时，处理集合数据，往往会有数据遍历过程，如果无并发则时间复杂度是 $O(N)$ ，大的数据集必然带来更慢的响应速度，而 consumer 端不会根据 payload 大小动态设置超时时间，它可能导致 consumer 端超时，超时可能带来多次重试，进而加重服务端压力。

例如：无印良品订单 sku 品类过多，比如一个出库单包含 2 万个 sku 的极端情况。

例如：WMS 出库发货后向 ECLP 回传信息，之前都是通过一个 JMQ Topic: eclp_delivery 进行回传，一份消息包含了（订单主档，箱明细，包裹明细）3 部分信息。后来在石化场景下，一个订单的包裹明细数量非常多，导致 ECLP 处理报文时 CPU 飙升，同时 MQ Listener 与对外服务共享 CPU，导致接单功能可用率降低。后来，从源头入手把一个订单按照明细进行分页式拆分（之前是整单回传，之后是按明细分页回传），同时把 eclp_delivery 这一个 topic 拆分成 3 个 topic:（订单，箱明细，包裹明细），解决了大报文问题。

1.2.2 大字段无约束

它指的是某一个字段（不是集合大小），由于没加长度限制，在特定场景下传入了远超预期大小的数据而造成的故障。

ECLP 的商品主数据有个下发商品的接口，有个字段 skuName，接口没有对该字段长度进行约束。系统一直平稳运行，直到有个商家下发了某一个商品，它的 skuName 达到了 10KB（事后发现，商家是把该商品详情页的整个 HTML 通过 skuName 传过来了），插入数据库时超过了字段长度限制 varchar(200)，导致插入失败，但由于没有考虑到这种场景，返回了误导的错误提示。展开来看，如果 ECLP 为 skuName 定义了 MySQL Text 类型字段，还会有更严重问题：ECLP 接收下商品，下发给 WMS，但 WMS 里的 skuName 是 varchar(200)，这个问题就只能人工处理了，甚至与商家沟通。

WMS6.0 为了考虑多场景全满足，在出库单预留了扩展字段，在接单时技术 BP 自行决定写入哪个扩展字段。京喜 BP 下发出库单时在订单明细维度传入了 handOverSlip（交接单，其实是团单信息，里面有多层明细嵌套），该字段其实是一个大 JSON，单个长度 10KB 上下，接单环节没问题。但组建集合单会把多个出库单组建一个集合单，共产生 3000 多个明细，仅 handOverSlip 就占 30MB，造成组建集合单后下发（JSF 调用）拣货时遇到了 JSF 8MB 限制问题，下发失败，单据卡在那里，现场生产无法继续。

WMS6.0 的用户中心系统，为其他系统提供了发送咚咚通知的服务，具体实现是调用集团的咚咚发送接口：xxx 生产系统 -> 用户中心 -> 咚咚系统。链路上每一个环节都未对通知内容 content 字段长度做限制。一次 xxx 生产系统调用用户中心传入了超 8MB 的 content 字段，触发了咚咚系统的 JSF 底层的报文限制，最终在用户中心产生了 ClientTimeoutException，它导致用户中心的 JSF 业务线程池打满；而由于用户中心为所有业务生产系统服务，现场操作会依赖它，进而导致生产卡顿，现场多环节无法正常生产。

Amazon FBA 的 SP-API (Sell Partner API)，对可能出现风险的字段都做了长度限制，例如：

```
String displayableOrderComment; // maxLength: 1000
String sellerSku; // maxLength: 50
String giftMessage; // maxLength: 512
String displayableComment; // maxLength: 250
```

1.2.3 查询接口返回大量数据

ECLP 主数据有个接口：导出所有 warehouse list，调用方很多，访问频率不高，每次响应长度 3MB。该接口在线上出现过多次事故（2019 年）。这个接口显然是不该存在的，但把它下线需要推动所有的调用方改动，这个周期很长阻力也很大。

最开始，直接查数据库，出现事故后加入 JimDB，再次出现事故后配置了 JimDB 的 local cache，后又加入 JSF 限流等措施。



1.2.4 导出问题

这个问题与【1.2.3 查询接口返回大量数据】看上去类似，但有很大不同：一个同步调用，返回的数据量相对少，另一个异步执行，返回数据量巨大。

WMS6.0 的报表都有导出的需求，例如导出最近 3 个月的明细数据。贴近商家的 OFC (如 ECLP)，也有类似需求，商家要求导出明细数据。系统执行过程大致是：根据用户指定的条件异步执行 SQL，把数据库返回的数据集写入 Excel，并存放到 blob storage (指定 TTL)，用户在规定时间 (TTL) 内根据 storage key 去 blob storage 下载，完成整个导出过程。

这里的关键问题是查询数据库，而数据库作为共享资源往往是整个系统的瓶颈（增加副本数量意味着成本上升），它变慢会拖垮整个系统。如何查询数据库，有 8 个可选项：

编号	方案	cons	pros	MySQL压力
1	PageHelper: 当前方案	深分页/慢查询工单/框架支持	开发简单	100
2	scroll pagination	scrollId不好找/必须order by/filesort/额外写SQL	如果写好无慢查询工单	50
3	covering index pagination/deferred join	索引不容易设计/研发侵入强/JOIN带SQL难写/降低回表次数为pageSize次	比PageHelper快	60
4	Client side cursor: enableStreamingResults	慢查询工单	响应快	40
5	MySQL side cursor: useCursorFetch, setFetchSize	cursor物化到一张临时表, IO/CPU飙升		80
6	Presto union all MySQL(T) & Hive[T+1]	加密字段Hive取不到/Presto团队禁止取数>1万行/学习PrestoSQL	W6无需开发	10
7	Java union all MySQL(T) & Hive[T+1]; merge sort	加密字段Hive取不到/sqlParse(order by, table name)/慢		10
8	split DateRange into DateChunks; merge sort	SqlParser(order by)	性能好/可控性高	20

导出问题的本质，是大范围 table scan，很难设计精细的复合索引。WMS6.0 最初使用的是方案 1，它会产生深分页 limit offset 问题：越往后的页面越慢，对数据库的压力越大。举例：要导出 100 万行记录，每页 1 万，那么到 50 万记录时，每次分页查询相当于数据库要扫描 50 万 + 行记录后抛弃绝大部分并返回 1 万行，这还要继续执行 50 次，此外分页组件还要额外执行 count 语句以计算总行数。

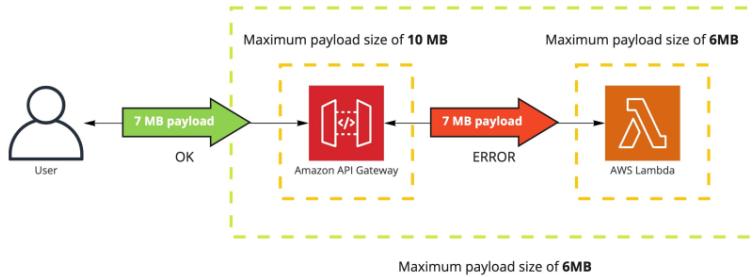
如果每页是 1 千呢？因此，数据库的压力被放大了，可以简单理解为“全表扫描”了【 $50 + 100 \text{ (count 计算)} = 150$ 】次，远不如不分页（不分页还要解决 OOM 问题）。目前，WMS6.0 改用了方案 8，根本上解决了数据库慢查询问题。思路是不再盲目静态分页，而是根据时间条件切分成多个 SQL，分别查询，保证每个 SQL 返回数据量不大从而避免慢 SQL。例如，某个仓要导出最近 3 个月的出库单数据，那么把这 1 个 date range 拆分 (explode) 成 N 个 date range，分别执行：

```
condition = DateRange(from = "2022-01-01 00:00:00", to = "2022-04-01 00:00:00") // 用户指定的时间范围: 3个月
// sql = select * from ob_shipment_order where xxx and update_time between condition.from and condition.to
List<DateRange> chunks = explode(condition)
for (DateRange chunk : chunks) {
    // 该chunk的时间范围已经变成了1天，甚至是1小时，具体值是根据SQL执行计划估算得来的：数据量越大则拆分越细
    sql = select * from ob_shipment_order where xxx and update_time between chunk.from and chunk.to
    mysql.query(sql)
}
```

1.2.5 payload 约束不一致产生的问题

链路上经过不同的系统，不同系统对 payload size 的约束不同，也可能产生问题，因为决定是否可以正常处理的是最小的那个，但链路长时相关方可能不知道，在异步场景下这个问题尤为明显。

例如，aws 的 API Gateway 与 Lambda 对 payload size 有不同的约束，最终用户必须知道限制最严格的那个环节。



用方却浑然不觉。

如果通过物流网关对外开放，网关 nginx 限制是 5MB，而 JSF 是 8MB，设计上没问题（fail fast），但可能造成服务方承诺与调用者感知端到端的不一致。

JSF 对 provider (jsf:server) 和 consumer 可以分别设置不同的报文大小限制，理论上也可能出现问题，但在京东物流尚未出现，可不必关注。

1.2.6 其他非入口场景

它发生在系统执行过程内部。典型场景是 DAO 层查询数据库返回大结果集，Redis 大 key 问题等。这要根据具体中间件机制来识别，例如，MyBatis 支持插件来识别 DAO 查询出大结果集：

```
public class ListResultInterceptor implements org.apache.ibatis.plugin.Interceptor {
    private static final int RESULTSET_SIZE_THRESHOLD = 10000;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        Object result = invocation.proceed();
        if (result != null && result instanceof List) {
            int resultSetSize = ((List) result).size();
            if (resultSetSize > RESULTSET_SIZE_THRESHOLD) {
                // 报警
            }
        }
        return result;
    }
}
```

2 设计原则

2.1 主动显式强约束

即，主动防御式自我保护，而不是依靠使用者的“自觉”：外部用户不可信赖。

对于 JSF，可以通过 JSR303 向 API Consumer 显式传递约束，并且该约束可以通过框架对业务代码无侵入地自动执行。对于 MQ，由于生产者与消费者解耦，无法直接传递约束，只能靠主动监控、人工协调。

它的前提条件，是研发有能力去主动识别出大报文风险。

2.2 Fail Fast

如果有前端，那么前端加约束，避免大报文传递给后端。

对于后端，链式的上下游关系中，上游要把好关。

这个原则并不是说下游不用关心大报文问题，恰恰相反，链路的每个环节都要关心，但 Fail Fast 可以降低整体的不必要的损耗成本，也可以缓解某个环节保护机制缺失带来的介入和修数成本。

2.3 上下游对齐隐式约束

同一个业务字段在上下游传递时，字段长度约束要一致，否则可能会出现上游成功落库下游无法落库的情况。

2.4 大报文产生方负责拆分

解决大报文的根本思路是拆分报文：大 -> 小。

对应 MQ 来讲，应该是 Producer 负责拆分大报文为小报文。



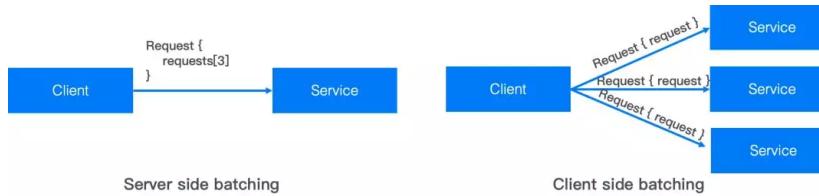
典型场景：拣货下架调用库存预占接口，一次性传入 1 万个 sku

- provider 产生的大报文：应该变成分页返回结果

典型场景：一次性返回所有 warehouse 列表

API的设计者可能会希望实现一个服务端的批量更新能力，但是我们建议要尽量避免这样做。除非对于客户来说提供原子化+事务性的批量很有意义（all-or-nothing），否则实现服务端的批量更新有诸多的弊端，而客户端批量更新则有优势：

- 服务端批量更新带来了API语义和实现上的复杂度，例如当部分更新成功时的语义、状态表达等。
- 即使我们希望支持批量事物，也要考虑到是否不同的后端实现都能支持事务性。
- 批量更新往往给服务端性能带来很大挑战，也容易被客户端滥用接口。
- 在客户端实现批量，可以更好的将负载由不同的服务端来承担（见图）。
- 客户端批量可以更灵活的由客户端决定失败重试策略。



需要注意的是，拆分报文，会增加生产方和消费方的复杂度，尤其是消费方：幂等，集齐，(并发和异步调用时产生的)乱序，业务的原子性保证等。例如，一个出库单明细行过多时，整单预占库存(大报文) -> 按订单明细分页预占(小报文)。

拣货下架按明细维度分页调用库存预占接口场景下，如果订单不允许缺量：整单预占时，该订单预占库存的原子性(要么全成功预占，要么一个sku都不预占)是由库存系统(provider)保证的；而在按订单明细维度分页预占时，原子性需要在拣货系统(consumer)保证，即如果后面页码的预占失败则需要把前面页码的预占释放。这增加 consumer 端复杂度，但为了系统的性能和可用性，这是值得的。当然，也有另外一个可选方案，仍旧让库存保证原子性，但库存接口需要增加类似(currentPage, totalPages)的参数，那样就是库存更复杂了。无论如何，都增加了整体复杂度。

3 具体办法

3.1 报文分页

适用场景：MQ，以及 JSF 返回大报文响应。

为了保持报文的完整性，也便于消费方实现幂等、集齐等逻辑，需要在报文里额外增加分页信息：

currentPage/totalPages。

```

class Payload {
    List<Item> items;
    int currentPage, totalPages;
}

void sendPayload(Payload payload) {
    int currentPage = 1;
    int totalPages = payload.getItems().size() / batchSize;
    Lists.partition(payload.getItems(), batchSize).forEach(subItems -> {
        Payload subPayload = new Payload(subItems)
        subPayload.setPageInfo(currentPage, totalPages)
        producer.send(subPayload)
        currentPage++;
    });
}

```

在极端复杂场景下，也可以考虑分拆 topic，但不推荐，因为它可能额外引入乱序问题。



3.2 报文转存

适用场景：MQ/JSF。

这种方案，也被称为 Claim Check Pattern。

把大的明细 List，按照固定 batch size 转存到 JFS/OSS/JimKV/S3 等外部 blob storage，在报文里存放指针 (blob 地址) 列表。

```
class BigPayload {
    List<Item> items;
}

class SmallPayload {
    List<String> itemBlobKeys;
}

void sendPayload(BigPayload bigPayload) {
    SmallPayload smallPayload = new SmallPayload();
    Lists.partition(bigPayload.getItems(), batchSize).forEach(subItems -> {
        List<String> itemBlobKeys = blogStore.putObjects(subItems)
        smallPayload.addItemBlobKeys(itemBlobKeys);
    });
    producer.send(JSON.encode(smallPayload));
}
```

目前上游系统（ecip、序列号、OMC 等）、DTC、下游系统（各版本 WMS）的信息传递使用了该办法，共用一个 JFS 集群。

◆ Side effects: 1) 引入额外依赖，而且消费方被迫引入依赖 2) 需要 Blob 存储的 TTL 机制或定期清理，否则加大存储成本 3) 为消费方带来了不确定性，从 blob 拿回的数据可能超大，在反序列化和处理过程中有 OOM/FullGC 等风险（虽然一些 json 库提供了底层的基于词法 token 的 Streaming Parsing API，但如果要读取全部内容仍然耗费大量内存）

3.3 报文截断

适用场景：大字段。

在确定用户体验可以接受的情况下，上层进行字段内容截断 (truncate)。及早截断，不要依赖下层数据库的截断机制。

3.4 分页调用

适用场景：JSF。

两种场景：一种是批量接口，即入参是集合，另一种是入参对象里有集合字段。

```
class FooRequest {
    @javax.validation.constraints.Size(min = 1, max = 200)
    private List<Bar> barItems;
}

interface JsfAPI {
    // 场景1：批量接口
    void foo(@javax.validation.constraints.Size(min = 1, max = 200) List<FooRequest> requests)

    // 场景2：请求对象里有集合字段
    void bar(FooRequest request);
}
```

对于 JSF Consumer，可以通过 JSF 异步调用，它相当于 redis pipeline 模式，也可以通过客户端线程池并发调



⚠️ async异步调用时，retries设置无效，不重试

```

int maxJsfRetries = 3; // JSF async下的自动重试只能应用层自己做了
int retried = 0;
do {
    List<ResponseFuture<Result<ObLocatingResultDto>>> futures = new LinkedList();
    Lists.partition(volist, batchSize).forEach(subVoList -> {
        ObLocatingOrderDto dto = mapper.INSTANCE.toDTO(subVoList);
        locatingAppService.outboundOrderLocate(dto); // async JSF call
        ResponseFuture<Result<ObLocatingResultDto>> future = RpcContext.getContext().getFuture();
        futures.add(future);
    });
}

for (ResponseFuture<Result<ObLocatingResultDto>> future : futures) {
    try {
        Result<ObLocatingResultDto> result = future.get();
    } catch (RpcException jsfException) {
        retried++;
    } catch (Throwable e) {
        // 额外的业务逻辑：与JSF并发同步调用相同的处理逻辑
    }
}
} while (retried <= maxJsfRetries);

```

👉 JSF 异步调用时，jsf:consumer 配置的 retries 无效，这是因为异步发送后如果出现网络超时，只能由业务代码通过 future.get() 才能拿到结果，JSF 底层没有机会进行自动重试。而同步调用时，JSF 底层可以判断出超时，它有机会根据配置进行自动重试。更多细节可以查看 JSF 的 FailoverClient.doSendMsg 方法。

3.5 MQ 替代 JSF

适用场景：单向通知类请求，相当于 AsyncAPI。

大的报文往往意味着更长的处理时长，JSF 同步调用下 consumer 必须同步等待 provider 端的返回，这会同时占用 consumer 和 provider 双方的线程池资源，极端情况下可能导致双方线程池用尽。JSF 下可能耗尽线程池，进而拖死被强依赖的上游，产生雪崩效应；而 MQ 下，只会消费积压。

异步交互，使得上游对下游响应时间的依赖转换为吞吐率的依赖。MQ 实现了消费者和生产者在时间和空间上的解耦，消息的消费者可以承受更大范围的处理速度范围。

3.6 总结

具体办法	适用的大报文场景
报文分页	JMQ，以及JSF返回大报文响应
报文转存	MQ/JSF
报文截断	大字段，且业务可接受
分页调用	JSF
MQ替代JSF	单向通知类请求

4 最佳实践

4.1 单个接口与批量接口分离

根据 sku 编号查询商品资料，往往伴随着多个 sku 一起查询的需求，如何设计接口？

有的这样：



```

    Result<List<SkuInfo>> getSkusBySkuId(String skuId);
    Result<List<SkuInfo>> listSkuInfo(List<String> skus);
}

```

由于批量接口在技术上已经满足了单个查询的功能，有的团队干脆去掉了单个查询接口，造成使用者查询单个sku时：

```
Result<SkuInfo> result = jsfAPI.listSkuInfo(Lists.newArrayList("EMG1800752592"));
```

应该这样：

```

interface JsfAPI {
    Result<SkuInfo> getSkuInfo(String sku);
}

interface JsfBulkAPI {
    Result<List<SkuInfo>> listSkuInfo(List<String> skus);
}

```

4.2 线程池隔离

JsfAPI 与 JsfBulkAPI 把批量与单一接口进行分离后，可以分配到不同的线程池，尽可能互不干扰，这同理于 Bulkhead Pattern。

单一接口	批量接口
处理关键业务，SLA 要求更高	风险高，性能差

JSF 可以通过 jsf:server 定义线程池，并为 jsf:provider 分配不同的 server。

4.3 大报文与小报文分离

如果大报文实在无法拆分（例如，上游团队不配合），为了降低极端请求对绝大部分正常请求的影响，可以采用大小报文分离的办法。

对于 JMQ，为了防止某一个大报文的消费长耗时或异常导致小报文的消费积压，可以把大报文转发到“慢队列”进行消费。

此外，也要考虑如何缓解 UMP 监控失真问题。

4.4 JMQ 设置合理的批量大小

开启并行消费 开启 关闭

应答超时时间(ms)	<input type="text" value="0"/>
批量大小(条)	<input type="text" value="1"/> ✓
延迟消费(ms)	<input type="text" value="0"/>

该值决定了 MessageListener.onMessage 入参 messages 的 size。

```

interface MessageListener {
    void onMessage(List<Message> messages) throws Exception;
}

```

JMQ Consumer 的 ACK 是以批为单位的，例如设置为 10，则 10 条消息里任意一条产生异常都会导致 10 条全失败。在建立场景下，如果发现问题，可以把该值调整为 1，避免大小报文相互影响。

信息数超过它，则 IO 阻塞以防止拉取新消息)。同时它也有两大负面效应：1) ACK 以批为单位，一个错误导致整批错误，整批重试 2) 消息大小限制取决于整批所有消息大小，可能触发大报文问题。

如果消费策略里配置了批量大小=10，如果每条消息1MB，那么消费不了任何一条消息？还是压根在消费时不限制？

 消息平台服务

消费没限制大小

对于京东物流绝大部分业务系统来讲，这点提升与繁重的业务处理来比不值一提，例如：I/O 节省了 5ms，但单个消息处理需要 200ms (因为要通过接口查询，处理，然后写库)，反倒是 side effect 成为主要矛盾。因此，绝大部分场景下该值应该设置为 1。如果业务逻辑类似于集齐：把 N 个消息拿下来，本地缓冲暂不处理，等满足条件了再 merge 并一次性处理，那么可以调整批量大小为非 1。

JMQ Producer 提供了批量发送方法：

```
interface Producer {
    void send(List<Message> messages) throws JMQException;
}
```

我们的业务代码也在使用，例如：

```
/**
 * 发送分播结果消息
 */
public void send(List<CheckResultDto> checkResultDtos) {
    List<Message> messageList = Lists.newArrayList();
    for (CheckResultDto checkResultDto : checkResultDtos) {
        String messageText = JmqMessage.createReportBody(checkResultDto.getUuid(), Lists.newArrayList(check
            messageList.add(JmqMessage.create(topic, messageText, checkResultDto.getUuid(), checkResultDto.getW
        }
        producer.send(messageList);
    }
}
```

这里要注意，分批发送时，1) 发送的超时 (默认 2s) 作用于整批消息，而不是单个消息 2) 消息大小限制 (4MB) 作用于整批消息之和，因此批包含的消息越多越可能失败。

4.5 避免大日志

尤其是 AOP/Interceptor/Filter 等统一处理的代码，因为对报文的打印往往需要先 json 序列化。

```
if (logger.isInfoEnabled()) {
    log.info(JsonUtil.toJson(request); // CPU intensive and disk I/O intensive(虽然日志是顺序写)
}
```

如果确实要记录，也可以考虑采样率方式记录大报文日志。

4.6 显式约束由严开始

开放 API 由于消费方多而且不确定性高，客观上造成了“只有一次做对的机会”。

List size limit, property max length limit 等，要在开放 API 的第一时间公布出去。如果开始不约束，后期加约束可能遭遇大的阻力和沟通成本。此外，遵循从严开始的规律，为自己争取主动：你把限制放开，没人找你岔，反之则阻力大。例如：order.items max size limit 由 100 变成 200，你可以放心地做；但由 200 变成 100，你要征得现有使用者的全部确认。

例如，Amazon FBA 的 SP-API 对集合的条数限制绝大部分是 50。

5 治理机制



无论采用哪种大报文问题解决办法，识别出大报文场景是前提。

技术上，可以通过 JSF Filter 分析报文长度，把尚未触发 8MB 但有潜在风险的自动识别出来。但 JMQ 无相关机制，业务系统要自行实现相关拦截机制。

5.1.1 JSF 自动识别

provider 端自动识别即可。

```
@Slf4j
public final class PayloadSizeFilter extends AbstractFilter {
    private static final int PAYLOAD_SIZE_THRESHOLD = 4 << 20; // 4MB = 8MB(JSF限制) * 50%
    private static final int BATCH_SIZE_THRESHOLD = 1000;

    @Override
    public ResponseMessage invoke(RequestMessage requestMessage) {
        if (!RpcContext.getContext().isProviderSide()) {
            // 只在provider端检查大报文：它才是我们要保护的对象
            return getNext().invoke(requestMessage);
        }

        // 自动识别潜在的大报文场景：针对报文大小
        Integer payloadSize = requestMessage.getMsgHeader().getLength();
        if (payloadSize != null && payloadSize > PAYLOAD_SIZE_THRESHOLD) {
            // 这里使用最简单的日志把潜在大报文暴露出来，各团队可以做更细化的机制
            // 由于logbook限制只有error level日志才能配置"关键字报警"，这里使用log.error
            // 如果不想自动报警，只是人工巡检，可以log.warn
            String methodName = requestMessage.getMethodName();
            String className = requestMessage.getClassName();
            log.error("Suspected BIG payload: {}.{}, {}>{}", className, methodName, payloadSize, PAYLOAD_SI
        }

        // 自动识别潜在的大报文场景：报文字节小，但仍会导致处理慢，例如 List<String> orderNos，如果发来1万个单号？
        // 这里只能识别出入参是List的场景，对于字段类型是List的场景无效
        Invocation invocation = requestMessage.getInvocationBody();
        Class[] argClasses = invocation.getArgClasses();
        Object[] args = invocation.getArgs();
        for (int i = 0; i < argClasses.length; i++) {
            Class argClass = argClasses[i];
            if (Collection.class.isAssignableFrom(argClass)) {
                // 入参类型是Collection
                Collection collection = (Collection) args[i];
                if (collection.size() > BATCH_SIZE_THRESHOLD) {
                    log.error("Too BIG Collection argument: {}>{}", collection.size(), BATCH_SIZE_THRESHOLD
                }
            }
        }

        return getNext().invoke(requestMessage);
    }
}
```

5.1.2 JMQ 自动识别

在 consumer 端加自动识别，如果发现，协同 producer 方确认风险判断是否需要改造。

```
public interface BigPayloadTrait extends MessageListener {
    int THRESHOLD_BIG_PAYLOAD = 2 << 20; // 2MB = 4MB(JMQ限制) * 50%

    default boolean suspectedBigPayload(List<Message> messages) {
        for (Message message : messages) {
            if (message.getSize() > THRESHOLD_BIG_PAYLOAD) {
                return true;
            }
        }

        return false;
    }
}
```



人工识别会有遗漏场景，关注监控全局指标，尤其是分析一些跳点，可能补充发现大报文场景。

5.3 设计应急预案

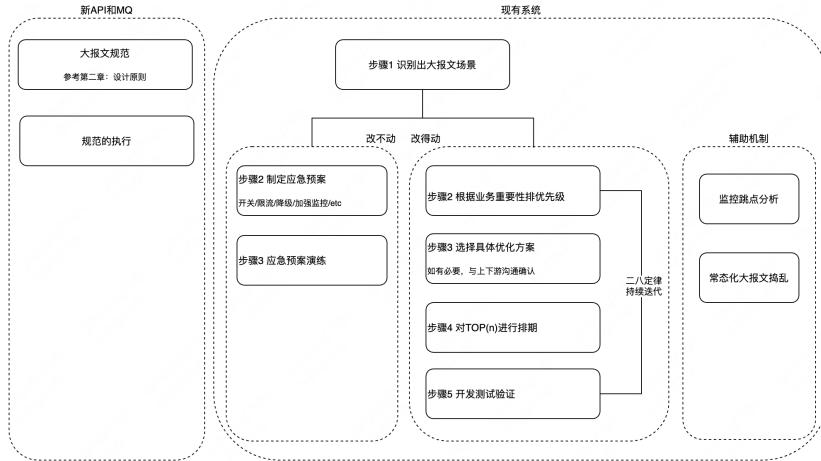
有些大报文问题，可能暂时无法通过技术手段解决，例如，已经有商家接入的对外接口，开放时没有对 List size 限制，加限制后需要商家配合修改做客户端分页，而商家不配合。这时候，可以采用大促期降级，限流，加开关，加强监控，设计应急预案，为此接口提供独立的线程池来隔离正常请求等手段解决。

5.4 常态化的大报文捣乱演练

以第三方视角帮助识别出尚未识别的大报文场景，不要自己给自己捣乱。

5.5 团队执行

推进大报文治理工作时，为了便于项目追踪管理，可以采用如下流程。



5.5.1 新的 API 和 MQ

这里也包括现有 API/MQ 上加字段场景。

设计和评审时，检查：

- 字段长度，在上下游上长度对齐
- JSF 接口对 List 等集合类型加 @Size 显式约束和校验，对 List 性批量接口入参也加 @Size
- MQ Producer 确保不发出大报文

5.5.2 现有系统治理

为所有 JSF 和 MQ 加入大报文预先监控机制 (具体可参考【5.1 识别大报文场景】)，根据是否改得动做相应的治 理动作。

作者：京东物流 高鹏

来源：京东云开发者社区 自猿其说 Tech





点击引领话题

其他人还在看

更多精彩内容

- Meta 积极推动 Python 项目采纳 PEP 703 提案，让全局解...
🔥 周热点 | 爱奇艺客户端“白嫖”电视机；Twitter劲敌注册...
字节跳动开源西瓜播放器 (HTML5)：内置解析器、能节省...
Bytebase 2.4 发布，支持 Snowflake SQL 审核
DHorse v1.2.1 发布，基于 k8s 的发布平台
Winamp 播放器 App 开启内测，提供 Android & iOS 版本
Apache EventMesh v1.9.0 发布，已支持轻量化事件连接...
文件系统考古：1974-Unix V7 File System
HarfBuzz 8.0 发布，引入 WebAssembly“WASM”整形器
DjangoAdmin 敏捷开发框架 FastAPI+AntdVue 版本 v1.0...
字节跳动开源KubeAdmiral：基于 K8s 的新一代多集群编...
TiDB v7.1 resource control 资源管控特性体验贴
与 NGINX 团队直接交流 | 微服务之月火热报名中
IDE 新 UI 的这 5 个设计，用了就离不开！
MySQL Router高可用搭建
使用 InstructPix2Pix 对 Stable Diffusion 进行指令微调
以科技创新驱动高质量发展，天翼云操作系统获国资委权威...
中国开源发展所面临的几个问题和挑战
深度Q网络：DQN项目实战CartPole-v0
实例讲解看nsenter带你“上帝视角”看网络
ALC Xi'an 正式成立，首场线下 Meetup 来袭！
Hugging Face 创始人采访实录：创业历程、AI 发展趋势洞察
GreatSQL社区月报 | 2023.05
我用低代码结合ChatGPT开发，每天多出1小时摸鱼
带你彻底掌握Bean的生命周期
创新涌动于先 | 2023 PingCAP 用户峰会等你来！
ARHUD驾车导航技术概览
基础大模型能像人类一样标注数据吗？
用 ChatGPT 实现综艺节目中的“你说我猜”游戏 | 征稿活动V6
Dgraph v23.0.1 发布，具有图形后端的原生 GraphQL 数...
CXYGZL - 工作流 V2.0.3 版本发布，支持业务系统对接
国内主流应用商店及分发平台基本完成 APP 签名和验签
smart-doc 2.7.2 发布，Java 零注解 API 文档生成工具
TIOBE 7 月榜单：C++ 即将超越 C，JavaScript 进入 Top6
GIMP 2.99.16 发布，开源图形编辑器
dbeaver 23.1.2 发布，可视化数据库管理平台
Linux 6.5-rc1 发布，初步支持 USB4 v2、CacheStat 和 A...
RuoYi-Vue-Plus 发布 4.8.0 新增 sms4j 短信融合
比 rz / sz 还好用的文件传输工具发布 v1.1.3
SeaTunnel毕业！首个国人主导的数据集成项目成为Apache...
介绍 9 个研发质量度量指标
2022 开源社年度报告：打开新世界
“让用户成为自己的导演”——专访Tiledmedia大中华区总经...
从 GaussDB(DWS)的技术演进，看数据仓库的积淀与新生
平凯星辰重磅支持 2023 开放原子全球开源峰会，开源数据...
OpenAI 领导层建议成立人工智能国际监管组织
最强AIGC实战应用速成指南来了！14天掌握核心技术
安全日报（2023.06.27）
天翼云边缘安全加速平台亮相2023亚太内容分发大会暨CD...
图书搜索领域重大突破！用 Apache SeaTunnel、Milvus ...
海睿思分享 | 低代码开发直面行业变革：革新，创新？
Serverless函数计算介绍
Hugging News #0602: Transformers Agents 介绍、大语...
DolphinDB x 恒泰证券 | 一体化投交平台：打破编程桎梏，...
对话钉钉音视频专家冯津伟：大模型不是万能的
AI低代码编程崛起，23年还要不要学软件？
【专题速递】虚实共生，视界无限

热门评论



五十风 2023-07-10 14:25

我c，每次手机投屏到电视时都有个爱奇艺，一直没太在意。原来除了自启投屏功能，还在跑流量。回去就卸载了



snake 2023-07-10 12:43

很印度



Raphael_goh 2023-07-10 11:33

其实GPT最擅长不是找答案，而是通过GPT启发思路。一些复杂逻辑，通过GPT简化思路，然后再配合自己的编码进行纠正，可以极大提升开发效率。但直接通过GPT找到所谓的“正确”答案，被怎么坑的都不知道。就和不要迷信权威一样，见仁见智吧。



Shine_Sun 2023-06-29 23:14

这就好比在火车上问人家买到票没有一样，你去谷歌问牛人这个语言好学吗？这不是搞笑吗？去大街上问估计百分之百说难学。



哎呦-又忘了 2023-07-10 15:57

只能评论出这种了，其他不让评 0.0



AK 2023-07-02 14:21

Loading [a11y]/accessibility-menu.js



**我的ID是jmjoy** 2023-07-10 14:37

还是阿三会吹，国内目前只敢吹5.5G。

**roomsss** 2023-07-04 21:11

早就该限制了。从贸易z开始的时候。打断他们产业链，现在开始人家都不知道存储了多少了。

**陈心巧** 2023-06-29 09:29

cuda有吗，TensorFlow/pytorch有吗，chatgpt有吗？好像都没，那么硬件软件产品都没有，是靠什么进入第一梯队的呢

**LookEyes** 2023-07-04 15:08

感谢小米提供的资金和技术。

**霹雳** 2023-07-10 10:49

我都怀疑红帽让竞品公司给渗透了，或者选了个三哥当领导。

**沐火浴风** 2023-07-10 13:38

Twitter不是被马斯克开源了吗？开源了还担心竞争？

**ardorleo** 2023-07-10 11:22

ubuntu 在偷着乐~~~ 😊

**Yanlongli** 2023-07-10 07:47

这个不只是占你流量、耗你电费，持续工作会影响硬件寿命

**中医药人工智能研究** 2023-07-05 14:38

三哥刚抢了小米50亿，有钱了，说话口气就是不一样哈！

**陨落人间** 2023-07-10 14:38

如果在海外，被发现估计被罚得裤穿窿。。

**红薯** 2023-06-29 19:27

牛 x

**世说z** 2023-06-28 11:51

社区最大的贡献者本来就是红帽，到底是社区在被放血还是红帽在被放血？红帽这套操作无非是让那些半点不贡献只想搞白嫖的人没那么舒服罢了，开源如果真的想继续活下去，就得明白谁才是最大的贡献者。

**开源中国阅卷组组长** 2023-07-09 21:24

用户自己跑，宽带警告你，企业跑就没事？

**Francesca** 2023-06-29 11:06

你说进就进了？国足是否是世界第一梯队的？按投入的资金来看确实是第一梯队的，花了那么多钱，踢得怎么样呢？

**osc_58622652** 2023-07-04 22:33

中国也开始限制用于制造芯片的稀土了 乌鸦哥说得好 难办就不要办了

**aabbcccli** 2023-07-10 11:21

就两个字：“牛逼”！

**-乐天-** 2023-06-30 12:19

红薯早已经实现财务自由了！

**钛元素** 2023-07-10 10:12

关键是在俺们这里，维权实在太困难了

**LookEyes** 2023-07-03 20:14

小问题？法盲吧？违反了《隐私保护法》和别人的肖像权了吧？平台居然还有法盲...

**osc_58622652** 2023-07-04 22:34

【坏消息】2019年全球原生镓产量约370.6t，有超过96%的产量来自中国，为356.6t。【好消息】中国对这玩意出口管制后，还有一国的原生镓产量相对较大。【坏消息】这个国家是俄罗斯。【好消息】除俄罗斯外，还有一国原生镓产量相对较大。这个国家是乌克兰，年产4t。【坏消息】乌克兰产这个的公司是俄罗斯铝业分公司，在乌东。

【好消息】日韩也稳定年产3t。【坏消息】全球各国镓生产原料主要来自中国铝土矿。【好消息】世界其它地区也有铝土矿。【坏消息】其它地区铝土矿中镓的含量和可利用度远低于中国铝土矿



**王政** 2023-06-28 10:30

通篇过了一遍，核心思想大概可以概括为：re-package is cheap, show community your code. 从这个角度我完全赞同——但是这个问题的根本障碍在于社区对所谓的OSS许可证的原教旨主义坚持，对于伤害社区的行为过于包容。也就是说，由于社区并没有足够强力的组织，对社区做贡献的人一直在被放血。

**放学路上** 2023-06-27 19:31

uniapp还是帮助了很多中小厂吧

**逍遥huang** 2023-07-10 11:56

开源中国有毛病吧，发啥都相似度高，沙雕玩意

**局** 2023-07-03 16:56

爬虫写得好，牢饭吃得早。

**元久** 2023-07-10 14:17

既然电视上这样，手机和电脑也偷流量吗？

**飞越围墙** 2023-06-29 18:14

关于第1点,有没有可能在Google码代码的本就是大佬们,所以这么快

**单一结构** 2023-07-10 10:59

运行配置、编译需要联网（外网下载几个g的包，即可src需要github的git，编译时候需要git验证）。安装时候需要‘管理员权限’，额外提一下，没有‘VulkanSDK’的用户需要安装‘VulkanSDK’，虽然 readme 没提到

**talent-tan** 2023-06-30 13:52

我一直以为开源中国最大的股东是红薯，没想到竟然是百度 😊，恭喜红薯实现财务自由，我实现红薯自由

**中殖** 2023-07-10 15:23

应该让搞AI的管互联网，这样大家才会互联互通。而不能让国内搞分连网的那帮人管AI，那样只会没投胎呢就要求别人搞证！

**LonnyWong** 2023-07-10 10:55

tsz -B 设置了多大的缓冲区？等 tsz 升级到 1.1.4 应该会修复（代码改了，还没发布）。

**开源海哥** 2023-07-10 10:48

和同一个数据库查询的性能完全一样，不会存在任何性能损耗。

**刘少** 2023-06-29 08:50

完了，这一次，A800也马上被禁了

**奉天承运皇帝** 2023-06-28 22:42

AI吹牛第一梯队

**h4cd** 2023-07-03 16:53

这人姓马，是叫马克扎克伯格是吧？

**8ug_icu** 2023-07-10 13:58

Twitter 没开源吧

**FlameMida** 2023-06-28 23:44

尬吹第一梯队？

**undefined** 2023-07-10 14:51

按照这个榜单趋势Java都快要被下边一位的C#赶上了

**小xu中年** 2023-07-10 13:59

优秀

**崔红保** 2023-06-29 14:08

这篇文章的核心关键点：无论是否基于 uni-app 开发，只要 App 的源码中包含安装 APK 的代码，即使采用原生开发，都会被 Google Play 下架。

**ymgydxd** 2023-06-29 20:26

受访者都是CS牛人，学习速度是真快



个出来，弄得好像是真的一样。回答一个问题，给了个错误答案，我告诉它答案是错误的，它道歉了一句后又把刚才的答案发了一遍，如此反复到接受它的错误答案为止

 或者 **或许是阿猜** 2023-07-10 10:42

谁看用户协议呢

 **AmbitiousL** 2023-07-10 10:25

这种跨数据源的关联查询，性能怎么样

 **Francesca** 2023-07-09 21:58

P2P上传应该是要通知用户或者给出明显的关闭按钮，这种偷偷上传的就是流氓软件罢了

 **韦小仇** 2023-06-29 14:01

我宁愿相信中国国足已经进入全球第一梯队

 **哎呦-又忘了** 2023-07-10 15:56

中国工程院院士、鹏城实验室主任高文在WAIC2023的昇腾人工智能产业高峰论坛上表示，鹏城实验室已启动脑海大模型计划，目标打造国内首个完全自主创新、开源开放的自然语言预训练大模型底座，参数级别达到2000亿，性能对标ChatGPT。

 **kiduc** 2023-07-10 10:42

不要用EFI

 **烈冰** 2023-06-30 11:54

百度居然是原股东，真是想不到

 **ppp5p** 2023-07-10 10:19

mac我也用edge

 **单一结构** 2023-07-10 13:39

java掉的这么快的吗。是没了甲骨文？红帽掐断了免费用户？还是VMware被卖了的原因？

 **逍遥huang** 2023-07-10 11:56

。。。

 **wryyyyyy** 2023-07-07 09:57

讨厌各种符号，Rust符号太多了，看着就心烦

 **code4all** 2023-07-10 15:13

mark

OSCHINA 社区

关于我们
联系我们
加入我们
合作伙伴
Open API

攻略

项目运营

Awesome 软件 (持续更新中)

在线工具

Gitee.com
企业研发管理
CopyCat-代码克隆检测
实用在线工具
[国家反诈中心APP下载](#)

QQ群



530688128

公众号



视频号



OSCHINA 小程序



取人人网技术文章，根据你的阅读喜好进行个性推荐

Loading [a11y]/accessibility-menu.js

