二

32 应对容器时代面临的挑战：长风破浪会有时、直挂云帆济沧海

当今的时代，容器的使用越来越普及，Cgroups、Docker、Kubernetes 等项目和技术越来越成熟，成为很多大规模集群的基石。

容器是一种沙盒技术，可以对资源进行调度分配和限制配额、对不同应用进行环境隔离。

容器时代不仅给我们带来的机遇，也带来了很多挑战。跨得过去就是机会，跳不过去就是坑。

在容器环境下，要直接进行调试并不容易，我们更多地是进行应用性能指标的采集和监控，并构建预警机制。而这需要架构师、开发、测试、运维人员的协作。

但监控领域的工具又多又杂，而且在持续发展和不断迭代。最早期的监控，只在系统发布时检查服务器相关的参数，并将这些参数用作系统运行状况的指标。监控服务器的健康状况，与用户体验之间紧密相关，悲剧在于监控的不完善，导致发生的问题比实际检测到的要多很多。

随着时间推移，日志管理、预警、遥测以及系统报告领域持续发力。其中有很多有效的措施，诸如安全事件、有效警报、记录资源使用量等等。但前提是我们需要有一个清晰的策略和对应工具，进行用户访问链路跟踪，比如 Zabbix、Nagios 以及 Prometheus 等工具在生产环境中被广泛使用。

性能问题的关键是人，也就是我们的用户。但已有的这些工具并没有实现真正的用户体验监控。仅仅使用这些软件也不能缓解性能问题，我们还需要采取各种措施，在勇敢和专注下不懈地努力。

一方面，Web 系统的问题诊断和性能调优，是一件意义重大的事情。需要严格把控，也需要付出很多精力。

当然，成功实施这些工作对企业的回报也是巨大的！

另一方面，拿 Java 领域事实上的标准 Spring 来说，SpringBoot 提供了一款应用指标收集器——Micrometer，官方文档连接：https://micrometer.io/docs。

- 支持直接将数据上报给 Elasticsearch、Datadog、InfluxData 等各种流行的监控系统。
- 自动采集最大延迟、平均延迟、95% 线、吞吐量、内存使用量等指标。

此外，在小规模集群中，我们还可以使用 Pinpoint、Skywalking 等开源 APM 工具。

容器环境的资源隔离性

容器毕竟是一种轻量级的实现方式，所以其封闭性不如虚拟机技术。

举个例子：

> 物理机/宿主机有 96 个 CPU 内核、256GB 物理内存，容器限制的资源是 4 核 8G，那么容器内部的 JVM 进程看到的内核数和内存数是多少呢？
>
> 目前来说，JVM 看到的内核数是 96，内存值是 256G。

这会造成一些问题，基于 CPU 内核数 availableProcessors 的各种算法都会受到影响，比如默认 GC 线程数：假如啥都不配置，JVM 看见 96 个内核，设置 GC 并行线程数为 `96*5/8~=60`，但容器限制了只能使用 4 个内核资源，于是 60 个并行 GC 线程来争抢 4 个机器内核，造成严重的 GC 性能问题。

同样的道理，很多线程池的实现，根据内核数量来设置并发线程数，也会造成剧烈的资源争抢。如果容器不限制资源的使用也会造成一些困扰，比如下面介绍的坏邻居效应。基于物理内存 totalPhysicalMemorySize 和空闲内存 freePhysicalMemorySize 等配置信息的算法也会产生一些奇怪的 Bug。

最新版的 JDK 加入了一些修正手段。

JDK 对容器的支持和限制

新版 JDK 支持 Docker 容器的 CPU 和内存限制：

> https://blogs.oracle.com/java-platform-group/java-se-support-for-docker-cpu-and-memory-limits

可以增加 JVM 启动参数来读取 Cgroups 对 CPU 的限制：

> https://www.oracle.com/technetwork/java/javase/8u191-relnotes-5032181.html#JDK-8146115

Hotspot 是一个规范的开源项目，关于 JDK 的新特性，可以阅读官方的邮件订阅，例如：

> https://mail.openjdk.java.net/pipermail/jdk8u-dev/

其他版本的 JDK 特性，也可以按照类似的命名规范，从官网的 Mailing Lists 中找到：

> https://mail.openjdk.java.net/mailman/listinfo

关于这个问题的排查和分析，请参考前面的章节[《JVM 问题排查分析调优经验》]。

坏邻居效应

有共享资源的地方，就会有资源争用。在计算机领域，共享的资源主要包括：

- 网络
- 磁盘
- CPU
- 内存

在多租户的公有云环境中，会存在一种严重的问题，称为“坏邻居效应”（noisy neighbor phenomenon）。当一个或多个客户过度使用了某种公共资源时，就会明显损害到其他客户的系统性能。（就像是小区宽带一样）

吵闹的坏邻居（noisy neighbor），用于描述云计算领域中，用来描述抢占共有带宽，磁盘 I/O、CPU 以及其他资源的行为。

坏邻居效应，对同一环境下的其他虚拟机/应用的性能会造成影响或抖动。一般来说，会对其他用户的性能和体验造成恶劣的影响。

云，是一种多租户环境，同一台物理机，会共享给多个客户来运行程序/存储数据。

坏邻居效应产生的原因，是某个虚拟机/应用霸占了大部分资源，进而影响到其他客户的性能。

带宽不足是造成网络性能问题的主要原因。在网络中传输数据严重依赖带宽的大小，如果某个应用或实例占用太多的网络资源，很可能对其他用户造成延迟/缓慢。坏邻居会影响虚拟机、数据库、网络、存储以及其他云服务。

有一种避免坏邻居效应的方法，是使用裸机云（bare-metal cloud）。裸机云在硬件上直接

运行一个应用，相当于创建了一个单租户环境，所以能消除坏邻居。虽然单租户环境避免了坏邻居效应，但并没有解决根本问题。超卖（over-commitment）或者共享给太多的租户，都会限制整个云环境的性能。

另一种避免坏邻居效应的方法，是通过在物理机之间进行动态迁移，以保障每个客户获得必要的资源。此外，还可以通过 存储服务质量保障（QoS，quality of service）控制每个虚拟机的 IOPS，来限制坏邻居效应。通过 IOPS 来限制每个虚拟机使用的资源量，就不会造成某个客户的虚机/应用/实例去挤占其他客户的资源/性能。

有兴趣的同学可以查看：

> [谈谈公有云的坏邻居效应](https://github.com/cncounter/translation/blob/master/tiemao_2016/45_noisy_neighbors/noisy_neighbor_cloud _performance.md)

GC 日志监听

从 JDK 7 开始，每一款垃圾收集器都提供了通知机制，在程序中监听 GarbageCollectorMXBean，即可在垃圾收集完成后收到 GC 事件的详细信息。目前的监听机制只能得到 GC 完成之后的 Pause 数据，其它环节的 GC 情况无法观察到。

一个简单的监听程序实现如下：

```
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.sun.management.GarbageCollectionNotificationInfo;
import com.sun.management.GcInfo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.management.ListenerNotFoundException;
import javax.management.Notification;
import javax.management.NotificationEmitter;
import javax.management.NotificationListener;
import javax.management.openmbean.CompositeData;
import java.lang.management.*;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicLong;

/**
 * GC 日志监听并输出到 Log
 * JVM 启动参数示例:
 * -Xmx4g -Xms4g -XX:+UseG1GC -XX:MaxGCPauseMillis=50
```

```
 * -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps
 */
@Configuration
public class BindGCNotifyConfig {

    public BindGCNotifyConfig() {
    }

    //
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    private final AtomicBoolean inited = new AtomicBoolean(Boolean.FALSE);
    private final List<Runnable> notifyCleanTasks = new CopyOnWriteArrayList<Runnab
    private final AtomicLong maxPauseMillis = new AtomicLong(0L);
    private final AtomicLong maxOldSize = new AtomicLong(getOldGen().getUsage().get
    private final AtomicLong youngGenSizeAfter = new AtomicLong(0L);

    @PostConstruct
    public void init() {
        try {
            doInit();
        } catch (Throwable e) {
            logger.warn("[GC 日志监听-初始化]失败! ", e);
        }
    }

    @PreDestroy
    public void close() {
        for (Runnable task : notifyCleanTasks) {
            task.run();
        }
        notifyCleanTasks.clear();
    }

    private void doInit() {
        //
        if (!inited.compareAndSet(Boolean.FALSE, Boolean.TRUE)) {
            return;
        }
        logger.info("[GC 日志监听-初始化]maxOldSize=" + mb(maxOldSize.longValue()));

        // 每个 mbean 都注册监听
        for (GarbageCollectorMXBean mbean : ManagementFactory.getGarbageCollectorMX
            if (!(mbean instanceof NotificationEmitter)) {
                continue;
            }
            final NotificationEmitter notificationEmitter = (NotificationEmitter) m
            // 添加监听
            final NotificationListener notificationListener = getNewListener(mbean)
            notificationEmitter.addNotificationListener(notificationListener, null,

            logger.info("[GC 日志监听-初始化]MemoryPoolNames=" + JSON.toJSONString(m
            // 加入清理队列
            notifyCleanTasks.add(new Runnable() {
                @Override
                public void run() {
                    try {
```

```
                    // 清理掉绑定的 listener
                    notificationEmitter.removeNotificationListener(notification
                } catch (ListenerNotFoundException e) {
                    logger.error("[GC 日志监听-清理]清理绑定的 listener 失败", e)
                }
            }
        });
    }
}

private NotificationListener getNewListener(final GarbageCollectorMXBean mbean)
    //
    final NotificationListener listener = new NotificationListener() {
        @Override
        public void handleNotification(Notification notification, Object ref) {
            // 只处理 GC 事件
            if (!notification.getType().equals(GarbageCollectionNotificationInf
                return;
            }
            CompositeData cd = (CompositeData) notification.getUserData();
            GarbageCollectionNotificationInfo notificationInfo = GarbageCollect
            //
            JSONObject gcDetail = new JSONObject();

            String gcName = notificationInfo.getGcName();
            String gcAction = notificationInfo.getGcAction();
            String gcCause = notificationInfo.getGcCause();
            GcInfo gcInfo = notificationInfo.getGcInfo();
            // duration 是指 Pause 阶段的总停顿时间，并发阶段没有 pause 不会通知。
            long duration = gcInfo.getDuration();
            if (maxPauseMillis.longValue() < duration) {
                maxPauseMillis.set(duration);
            }
            long gcId = gcInfo.getId();
            //
            String type = "jvm.gc.pause";
            //
            if (isConcurrentPhase(gcCause)) {
                type = "jvm.gc.concurrent.phase.time";
            }
            //
            gcDetail.put("gcName", gcName);
            gcDetail.put("gcAction", gcAction);
            gcDetail.put("gcCause", gcCause);
            gcDetail.put("gcId", gcId);
            gcDetail.put("duration", duration);
            gcDetail.put("maxPauseMillis", maxPauseMillis);
            gcDetail.put("type", type);
            gcDetail.put("collectionCount", mbean.getCollectionCount());
            gcDetail.put("collectionTime", mbean.getCollectionTime());

            // 存活数据量
            AtomicLong liveDataSize = new AtomicLong(0L);
            // 提升数据量
            AtomicLong promotedBytes = new AtomicLong(0L);
```

```
                // Update promotion and allocation counters
                final Map<String, MemoryUsage> before = gcInfo.getMemoryUsageBefore
                final Map<String, MemoryUsage> after = gcInfo.getMemoryUsageAfterGc
                //
                Set<String> keySet = new HashSet<String>();
                keySet.addAll(before.keySet());
                keySet.addAll(after.keySet());
                //
                final Map<String, String> afterUsage = new HashMap<String, String>(
                //
                for (String key : keySet) {
                    final long usedBefore = before.get(key).getUsed();
                    final long usedAfter = after.get(key).getUsed();
                    long delta = usedAfter - usedBefore;
                    // 判断是 yong 还是 old，算法不同
                    if (isYoungGenPool(key)) {
                        delta = usedBefore - youngGenSizeAfter.get();
                        youngGenSizeAfter.set(usedAfter);
                    } else if (isOldGenPool(key)) {
                        if (delta > 0L) {
                            // 提升到老年代的量
                            promotedBytes.addAndGet(delta);
                            gcDetail.put("promotedBytes", mb(promotedBytes));
                        }
                        if (delta < 0L || GcGenerationAge.OLD.contains(gcName)) {
                            liveDataSize.set(usedAfter);
                            gcDetail.put("liveDataSize", mb(liveDataSize));
                            final long oldMaxAfter = after.get(key).getMax();
                            if (maxOldSize.longValue() != oldMaxAfter) {
                                maxOldSize.set(oldMaxAfter);
                                // 扩容；老年代的 max 有变更
                                gcDetail.put("maxOldSize", mb(maxOldSize));
                            }
                        }
                    } else if (delta > 0L) {
                        //
                    } else if (delta < 0L) {
                        // 判断 G1
                    }
                    afterUsage.put(key, mb(usedAfter));
                }
                //
                gcDetail.put("afterUsage", afterUsage);
                //

                logger.info("[GC 日志监听-GC 事件]gcId={}; duration:{}; gcDetail: {}
            }
        };

        return listener;
    }

    private static String mb(Number num) {
        long mbValue = num.longValue() / (1024 * 1024);
        if (mbValue < 1) {
            return "" + mbValue;
```

```
        }
        return mbValue + "MB";
    }

    private static MemoryPoolMXBean getOldGen() {
        List<MemoryPoolMXBean> list = ManagementFactory
                .getPlatformMXBeans(MemoryPoolMXBean.class);
        //
        for (MemoryPoolMXBean memoryPoolMXBean : list) {
            // 非堆的部分-不是老年代
            if (!isHeap(memoryPoolMXBean)) {
                continue;
            }
            if (!isOldGenPool(memoryPoolMXBean.getName())) {
                continue;
            }
            return (memoryPoolMXBean);
        }
        return null;
    }

    private static boolean isConcurrentPhase(String cause) {
        return "No GC".equals(cause);
    }

    private static boolean isYoungGenPool(String name) {
        return name.endsWith("Eden Space");
    }

    private static boolean isOldGenPool(String name) {
        return name.endsWith("Old Gen") || name.endsWith("Tenured Gen");
    }

    private static boolean isHeap(MemoryPoolMXBean memoryPoolBean) {
        return MemoryType.HEAP.equals(memoryPoolBean.getType());
    }

    private enum GcGenerationAge {
        OLD,
        YOUNG,
        UNKNOWN;

        private static Map<String, GcGenerationAge> knownCollectors = new HashMap<S
            put("ConcurrentMarkSweep", OLD);
            put("Copy", YOUNG);
            put("G1 Old Generation", OLD);
            put("G1 Young Generation", YOUNG);
            put("MarkSweepCompact", OLD);
            put("PS MarkSweep", OLD);
            put("PS Scavenge", YOUNG);
            put("ParNew", YOUNG);
        }};

        static GcGenerationAge fromName(String name) {
            return knownCollectors.getOrDefault(name, UNKNOWN);
        }
```

```
        public boolean contains(String name) {
            return this == fromName(name);
        }
    }

}
```

不只是 GC 事件，内存相关的信息都可以通过 JMX 来实现监听。很多 APM 也是通过类似的手段来实现数据上报。

APM 工具与监控系统

在线可视化监控是如今生产环境必备的一个功能。业务出错和性能问题随时都可能会发生，而且现在很多系统不再有固定的业务窗口期，所以必须做到 7x24 小时的实时监控。

目前业界有很多监控工具，各有优缺点，需要根据需要进行抉择。

一般来说，系统监控可以分为三个部分：

- 系统性能监控，包括 CPU、内存、磁盘 IO、网络等硬件资源和系统负载的监控信息。
- 业务日志监控，场景的是 ELK 技术栈、并使用 Logback+Kafka 等技术来采集日志。
- APM 性能指标监控，比如 QPS、TPS、响应时间等等，例如 MicroMeter、Pinpoint 等。

系统监控的模块也是两大块：

- 指标采集部分
- 数据可视化系统

如今监控工具是生产环境的重要组成部分。测量结果的可视化、错误追踪、性能监控和应用分析是对应用的运行状况进行深入观测的基本手段。

认识到这一需求非常容易，但要选择哪一款监控工具或者哪一组监控工具却异常困难。

下面介绍几款监测工具，这些工具包括混合开源和 SaaS 模式，每个都有其优缺点，可以说没有完美的工具，只有合适的工具。

指标采集客户端

- Micrometer：作为指标采集的基础类库，基于客户端机器来进行，用户无需关注具体的

JVM 版本和厂商。以相同的方式来配置，可以对接到不同的可视化监控系统服务。主要用于监控、告警，以及对当前的系统环境变化做出响应。Micrometer 还会注册 JMX 相关的 MBeans，非常简单和方便地在本地通过 JMX 来查看相关指标。如果是生产环境中使用，则一般是将监控指标导出到其他监控系统中保存起来。

- 云服务监控系统：云服务监控系统厂商一般都会提供配套的指标采集客户端，并对外开放各种 API 接口和数据标准，允许客户使用自己的指标采集系统。
- 开源监控系统：各种开源监控系统也会提供对应的指标采集客户端。

云服务监控系统

SaaS 服务的监控系统一般提供存储、查询、可视化等功能的一体化云服务。大多包含免费试用和收费服务两种模式。如果企业和机构的条件允许，付费使用云服务一般是最好的选择，毕竟“免费的才是最贵的”。

下面我们一起来看看有哪些云服务：

- AppOptics，支持 APM 和系统监控的 SaaS 服务，支持各种仪表板和时间轴等监控界面，提供 API 和客户端。
- Datadog，支持 APM 和系统监控的 SaaS 服务，内置各种仪表板，支持告警。支持 API 和客户端，以及客户端代理。
- Dynatrace，支持 APM 和系统监控的 SaaS 服务，内置各种仪表板，集成了监控和分析平台。
- Humio，支持 APM、日志和系统监控的 SaaS 服务。
- Instana，支持自动 APM、系统监控的 SaaS 服务。
- New Relic，这是一款具有完整 UI 的可视化 SaaS 产品，支持 NRQL 查询语言，New Relic Insights 基于推模型来运行。
- SignalFx，在推送模型上运行的 SaaS 服务，具有完整 UI。支持实时的系统性能、微服务，以及 APM 监控系统，支持多样化的预警“检测器”。
- Stackdriver，是 Google Cloud 的嵌入式监测套件，用于监控云基础架构、软件和应用的性能，排查其中的问题并加以改善。这个监测套件属于 SaaS 服务，支持内置仪表板和告警功能。
- Wavefront，是基于 SaaS 的指标监视和分析平台，支持可视化查询，以及预警监控等功能，包括系统性能、网络、自定义指标、业务 KPI 等等。
- 听云，是国内最大的应用性能管理（APM）解决方案提供商。可以实现应用性能全方位可视化，从 PC 端、浏览器端、移动客户端到服务端，监控定位崩溃、卡顿、交互过慢、第三方 API 调用失败、数据库性能下降、CDN 质量差等多维复杂的性能问题。

- OneAPM，OneAPM（蓝海讯通）提供端到端 APM 应用性能管理软件及应用性能监控软件解决方案。
- Plumbr，监测可用性和性能问题，使用跟踪技术，能迅速定位错误相关的位置信息，发现、验证和修复各种故障和性能问题。
- Takipi，现在改名叫做 OverOps，系统故障实时监测系统。能快速定位问题发生的时间、位置和原因。

其中做得比较好的有国外的 Datadog，国内的听云。

开源监控系统

- Pinpoint，受 Dapper 启发，使用 Java/PHP 来实现的大型分布式系统 APM 工具。Pinpoint 提供了一套解决方案，可通过跟踪分布式应用程序之间的事务来快速定位调用链路。
- Atlas，是 Netflix 旗下的一款开源的，基于内存的时序数据库，内置图形界面，支持高级数学运算和自定义查询语言。
- ELK 技术栈，一般用于日志监控，Elasticsearch 是搜索引擎，支持各种数据和指标存储，日志监控一般通过 Logstash 执行分析，Kibana 负责人机交互和可视化。
- Influx，InfluxDB 是由 InfluxData 开发的一款开源时序型数据库。它由 Go 写成，着力于高性能地查询与存储时序数据。InfluxDB 被广泛应用于存储系统的监控数据、IoT 行业的实时数据等场景，通过类似 SQL 的查询语言来完成数据分析。InfluxData 工具套件可用于实时流处理，支持抽样采集指标、自动过期、删除不需要的数据，以及备份和还原等功能。
- Ganglia，用于高性能计算系统、群集和网络的可伸缩的分布式监控工具。起源于加州大学伯克利分校，是一款历史悠久的多层级指标监控系统，在 Linux 系统中广受欢迎。
- Graphite，当前非常流行的多层级次指标监控系统，使用固定数量的底层数据库，其设计和目的与 RRD 相似。由 Orbitz 在 2006 年创建，并于 2008 年开源。
- KairosDB，是建立在 Apache Cassandra 基础上的时序数据库。可以通过 Grafana 来绘制精美漂亮的监控图表。
- Prometheus，具有简单的内置 UI，支持自定义查询语言和数学运算的、开源的内存时序数据库。Prometheus 设计为基于拉模型来运行，根据服务发现，定期从应用程序实例中收集指标。
- StatsD，开源的、简单但很强大的统计信息聚合服务器。

其中 Pinpoint 和 Prometheus 比较受欢迎。

参考链接

- 利用 JMX 的 Notifications 监听 GC
- 推荐 7 个超棒的监控工具

上一页