38 如何增加监控

更新时间：2020-08-28 09:43:01

> 天才就是百分之二的灵感，百分之九十八的汗水。——爱迪生

前言

你好，我是彤哥。

通过前面章节的不断调优，我们的实战项目不管是扩展性、安全性，还是性能方面，都有了质的提升。

不过，要说现在可以上线了，总感觉还缺点什么，现在应用对我们还像一个黑盒子一样，也不知道内存使用情况、连接数情况等，也不知道有没有达到系统瓶颈。

所以，我们还需要给应用添加上监控，做到对服务运行的各种指标心里有数，才能更可控。

好了，让我们一起进入今天的学习吧。

什么是监控？

所谓监控，是指收集系统的一系列数据（指标），并以某种形式展示出来，从而达到监测系统运行状况的目的。

有了监控，我们可以提前发现系统运行的问题，比如内存持续增高，并针对性地进行修改，以防止生产出现事故。

即使生产出现事故，我们也可以利用监控数据协助排查问题，达到快速定位快速解决生产问题的目的。

那么，以上一节为例，我们如何监控 Netty 对内存的使用情况呢？

如何监控内存？

在 Java 应用程序中，内存分为堆内存和直接内存，堆内存使用 jmap、jconsole 等工具都可以很方便地查看，但是，直接内存目前也没有特别好的方法去查看，然而，Netty 使用的又是直接内存，所以，我们需要把直接内存监控起来，时刻注意 Netty 使用直接内存的情况。

那么，该如何监控 Netty 对直接内存的使用情况呢？

其实，在之前的源码分析中，我们都多多少少见过 metric 相关的字眼，它们就是用来做监控的，比如，在创建非池化的直接内存的 ByteBuf 时，实际创建的是它的增强类，里面就有一些监控的东西：

```
private static final class InstrumentedUnpooledUnsafeDirectByteBuf extends UnpooledUnsafeDirectByteBuf {
    InstrumentedUnpooledUnsafeDirectByteBuf(
            UnpooledByteBufAllocator alloc, int initialCapacity, int maxCapacity) {
        super(alloc, initialCapacity, maxCapacity);
    }

    @Override
    protected ByteBuffer allocateDirect(int initialCapacity) {
        ByteBuffer buffer = super.allocateDirect(initialCapacity);
        // 增加直接内存的使用
        ((UnpooledByteBufAllocator) alloc()).incrementDirect(buffer.capacity());
        return buffer;
    }

    @Override
    protected void freeDirect(ByteBuffer buffer) {
        int capacity = buffer.capacity();
        super.freeDirect(buffer);
        // 减少直接内存的使用
        ((UnpooledByteBufAllocator) alloc()).decrementDirect(capacity);
    }
}
public final class UnpooledByteBufAllocator extends AbstractByteBufAllocator implements ByteBufAllocatorMetricProvider {
    private final UnpooledByteBufAllocatorMetric metric = new UnpooledByteBufAllocatorMetric();

    void incrementDirect(int amount) {
        // 监控直接内存的使用
        metric.directCounter.add(amount);
    }
}
private static final class UnpooledByteBufAllocatorMetric implements ByteBufAllocatorMetric {
    // 统计直接内存
    final LongCounter directCounter = PlatformDependent.newLongCounter();
    // 统计堆内存
    final LongCounter heapCounter = PlatformDependent.newLongCounter();
}
```


可以看到，连堆内存的使用情况也有监控到，不过，默认地，Netty 使用的是池化的直接内存的 ByteBuf，这里只能做到非池化的监控，所以，这里的直接内存的使用情况对我们的帮助并不大。

那么，还有没有其它地方有类似的监控指标呢？

答案还是在源码里，不管是池化的还是非池化的，最终都会调用到 PlatformDependent 的 allocateDirectNoCleaner() 方法分配内存，关于 NoCleaner 我们之前分析过了，它是 Netty 可以实现自己控制内存的关键，这里就不再赘述了：

```
public static ByteBuffer allocateDirectNoCleaner(int capacity) {
    assert USE_DIRECT_BUFFER_NO_CLEANER;

    incrementMemoryCounter(capacity);
    try {
        return PlatformDependent0.allocateDirectNoCleaner(capacity);
    } catch (Throwable e) {
        decrementMemoryCounter(capacity);
        throwException(e);
        return null;
    }
}
```

这里有个 incrementMemoryCounter(capacity) 方法，跟踪进去：

```
private static void incrementMemoryCounter(int capacity) {
    if (DIRECT_MEMORY_COUNTER != null) {
        long newUsedMemory = DIRECT_MEMORY_COUNTER.addAndGet(capacity);
        if (newUsedMemory > DIRECT_MEMORY_LIMIT) {
            DIRECT_MEMORY_COUNTER.addAndGet(-capacity);
            throw new OutOfDirectMemoryError("failed to allocate " + capacity
                    + " byte(s) of direct memory (used: " + (newUsedMemory - capacity)
                    + ", max: " + DIRECT_MEMORY_LIMIT + ')');
        }
    }
}
```

看见没，分配了多少内存，DIRECT_MEMORY_COUNTER 的值就增加多少，这不就是妥妥地监控吗？

所以，我们只需要拿到这个值，就可以监控直接内存的使用情况了，该如何拿到这值呢？

查看该值，发现它是一个私有变量，不过好在，Netty 提供了一个公共静态方法可以获取到它的值，除了它的值，我们还可以获取最大可使用的直接内存的值：

```
public final class PlatformDependent {
    private static final AtomicLong DIRECT_MEMORY_COUNTER;
    // 已使用的直接内存
    public static long usedDirectMemory() {
        return DIRECT_MEMORY_COUNTER != null ? DIRECT_MEMORY_COUNTER.get() : -1;
    }
    // 最大可使用的直接内存
    public static long maxDirectMemory() {
        return DIRECT_MEMORY_LIMIT;
    }
}
```

好了，现在知道我们要监控的数据在哪了，下面就是该如何实现监控了？

其实，实现监控并没有想像得那么难，简单一点，我们只需要定时的打印上面这两个值即可，稍微复杂一点，公司如果有监控平台，也可以把这两个值发送过去，使用图表更直观地监控起来。

我们这里就简单点，定时地打印到控制台。

在业界，也有一些框架可以帮我们实现监控，比如 dropwizard，本节，我们就使用它来将监控指标打印到控制台。

首先，是在 pom.xml 中添加依赖：

```
<dependency>
    <groupId>io.dropwizard.metrics</groupId>
    <artifactId>metrics-core</artifactId>
    <version>4.1.9</version>
</dependency>
```

然后编写一个监控的工具类：

```
public class MetricsUtils {

    public static void start() {
        // 注册
        MetricRegistry metricRegistry = new MetricRegistry();
        metricRegistry.register("usedDirectMemory"
                , (Gauge<Long>) () -> PlatformDependent.usedDirectMemory());
        metricRegistry.register("maxDirectMemory"
                , (Gauge<Long>) () -> PlatformDependent.maxDirectMemory());
        // 打印到控制台
        ConsoleReporter consoleReporter = ConsoleReporter.forRegistry(metricRegistry).build();
        consoleReporter.start(5, TimeUnit.SECONDS);
    }
}
```

在服务端的 main () 方法中调用这个静态的 start () 方法就可以了，让我们来看看效果：

```
20-6-27 16:37:17 ============================================================

-- Gauges --------------------------------------------------------------
maxDirectMemory
             value = 1883242496
usedDirectMemory
             value = 0
```


因为我现在只是启动了程序，还没有客户端连接进来，所以，usedDirectMemory 还是 0，连一个客户端进来试试：

```
20-6-27 16:38:32 ============================================================

-- Gauges --------------------------------------------------------------
maxDirectMemory
             value = 1883242496
usedDirectMemory
             value = 16777223
```

usedDirectMemory 有值了，是 16M，因为默认情况下 Netty 使用的是池化的直接内存的 ByteBuf，创建第一个 ByteBuf 时需要向直接内存申请一个 PoolChunk 的大小，也就是 16M，跟内存池的知识联系起来了。

使用 Netty 的时候直接内存是一个非常重要的指标，除了这个指标，还有哪些指标应该监控起来呢？

还要监控哪些指标？

我认为一些常用的指标都可以监控起来，有些是排查问题的时候需要，有的是预警需要的，还有些可能是老板比较关心的，总结起来大概有：

1. 线程池的使用情况，监控线程数
2. 待处理任务的情况，监控每个线程的待处理任务数
3. 发送缓冲区的情况，监控服务器是否健康，是否有消息积累

4. 连接数的情况，统计日活、峰值，灰度发布的时候需要
5. 内存使用情况

等等，你还能想到哪些指标应该监控起来呢？

后记

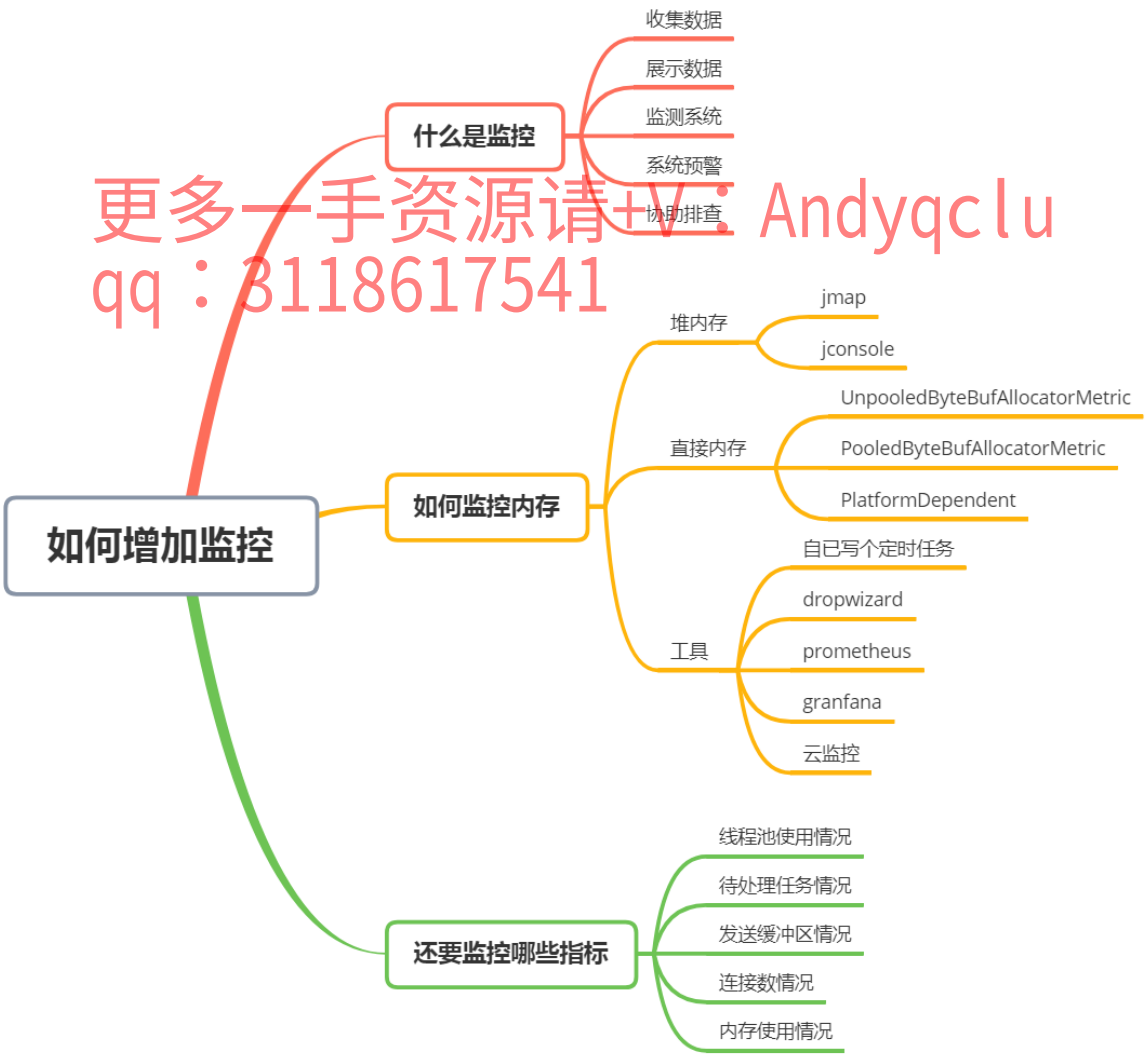
本节，我们一起给实战项目加上了监控，有了监控，我们才能做到心里有数，能够帮助我们提前规避掉问题，甚至出现问题了，我们也可以做到不慌，协助我们快速定位问题、快速解决问题。

到这里，整个实战项目的调优就讲解完毕了，使用了这些调优手段，终于可以安心的部署到生产环境了。

但是，到了生产就一定不会出现问题吗？真的出现问题怎么办？

下一节，我将介绍一下如何快速排查生产问题，敬请期待。

思维导图



}