

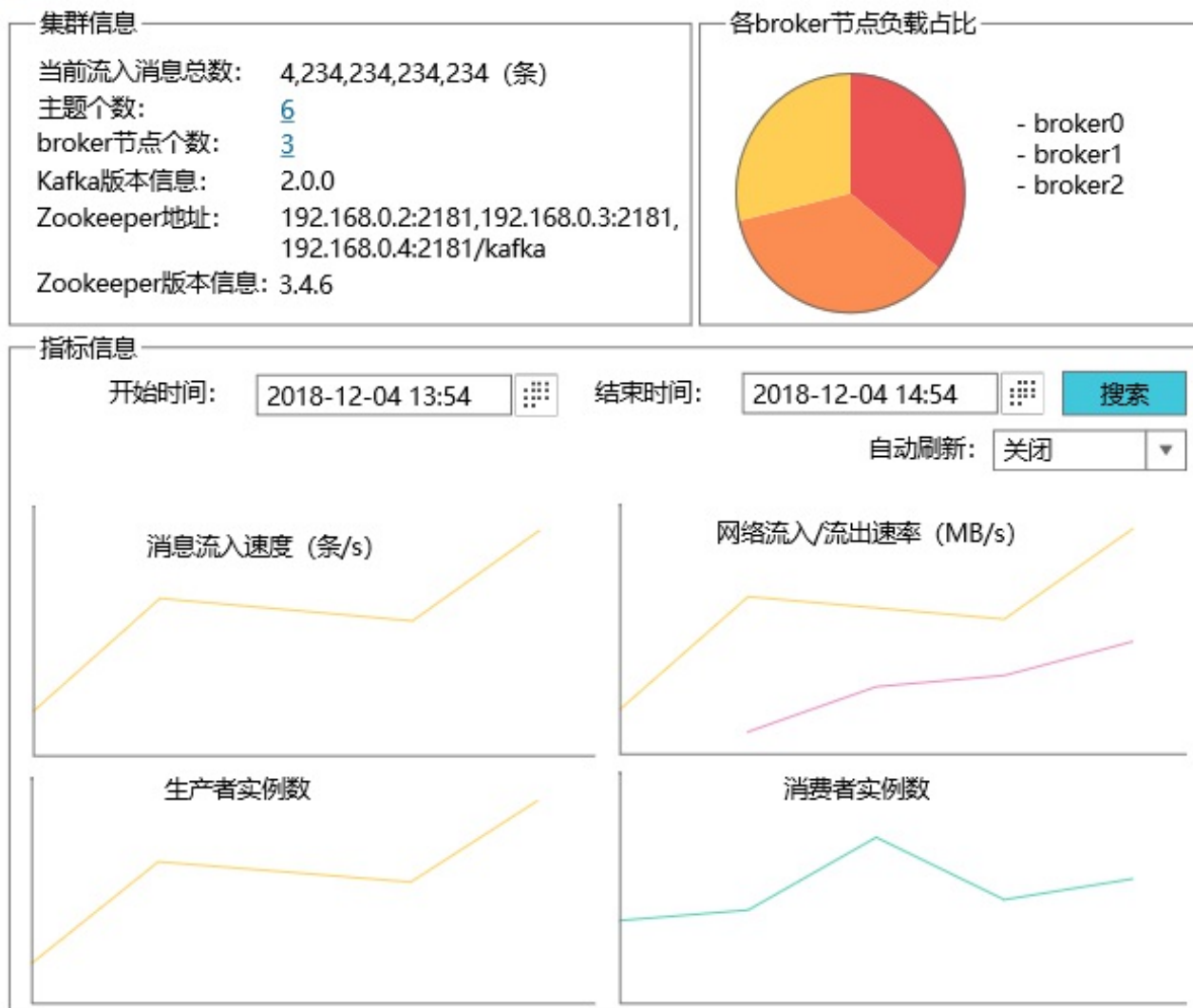
Kafka监控

任何应用功能再强大、性能再优越，如果没有与之匹配的监控，那么一切都是虚无缥缈的。监控不仅可以为应用提供运行时的数据作为依据参考，还可以迅速定位问题，提供预防及告警等功能，很大程度上增强了整体服务的鲁棒性。

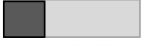
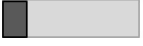
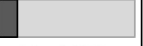
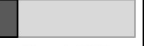
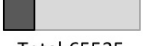



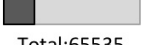



目前的 Kafka 监控产品有很多，比如 Kafka Manager、Kafka Eagle、Kafka Monitor、KafkaOffsetMonitor、Kafka Web Console、Burrow 等，它们都有各自的优缺点。以 Kafka Manager 为例，它提供的监控功能也是相对比较完善的，在实际应用中具有很高的使用价值。但有一个遗憾就是其难以和公司内部系统平台关联，对于业务资源的使用情况、相应的预防及告警的联动无法顺利贯通。在人力、物力等条件允许的情况下，自定义一套监控系统非常有必要。

接下来几个章节的内容并不是讲述如何使用现存的一些 Kafka 监控产品，而是讲述如何自己实现一套 Kafka 的监控产品。从监控维度来看，Kafka 可以分为集群信息、broker 信息、主题信息和消费组信息四个方面。有些情况下，也可以将 ZooKeeper 的监控信息概括进来，毕竟 ZooKeeper 也是 Kafka 整体架构的一部分，不过这里并不打算讨论 ZooKeeper 的更多监控细节，本章只以 Kafka 本身为主进行探讨。以集群信息为例，它需要展示整个集群的整体面貌，其中可以囊括一些 broker 概要信息、主题概要信息和消费组概要信息等内容，下面3张图就展示的就一份关于集群层面信息的监控设计文稿。

集群 > cluster.demo



第一张图展示的是集群的一些总览信息，包括基本的主题个数、broker 节点个数、Kafka 版本、ZooKeeper 地址及版本等。图中右上角是集群中各个 broker 节点负载的占比，如果负载均衡严重失调，则会对集群整体性能及使用上造成很大的困扰。图中下半部分是一些历史曲线信息，比如整个集群的消息流入/流出速度（条/s），我们可以通过这些历史曲线来了解整个集群的运行状况。

| broker列表概要信息 | | | | | | |
|--------------|---|---|---|---|--|---|
| 0 | Host: 192.168.0.2 Port: 9092 JMX Port: 9999 Controller | FD | CPU % | Memory | DISK | MsgIn: 2536条/s Partitions: 100 Leaders: 33 Under-replicated: 0 |
| | |  |  |  |  | |
| | | Total:65535 | | Total:8GB | Total:1TB | |
| 1 | Host: 192.168.0.3 Port: 9092 JMX Port: 9999 | IORead:10.2MB/s BytesIn:10.2MB/s | IOWrite:18.2MB/s BytesOut:18.2MB/s | | | MsgIn: 2336条/s Partitions: 100 Leaders: 34 Under-replicated: 0 |
| | | FD | CPU % | Memory | DISK | |
| | |  |  |  |  | |
| 2 | Host: 192.168.0.4 Port: 9092 JMX Port: 9999 | IORead:10.2MB/s BytesIn:10.2MB/s | IOWrite:18.2MB/s BytesOut:18.2MB/s | | | MsgIn: 2636条/s Partitions: 100 Leaders: 34 Under-replicated: 0 |
| | | FD | CPU % | Memory | DISK | |
| | |  |  |  |  | |

第二张图展示的是集群中各个 broker 的必要信息，这样可以在全局上了解各个节点的运行状态，这个图的设计灵感来源于 RabbitMQ 的监控插件 rabbitmq_management，这种形式的信息概览设计得非常精巧，比如图中的 Controller 标记，代表集群中的唯一一个控制器所处的节点位置，这样一目了然。图中的每一项都可以链接到具体的 broker 信息的页面，这样可以更详细地了解每一个 broker（比如其中可以包含一些历史曲线等）。

监控一般要配套告警模块，否则只能人为地进行监控，有了告警模块可以对某些重要的监控项设定告警阈值，以便能够及时地通知相关的人员处理故障，或者也可以触发自动运维的动作，这一切都是为了更好地利用 Kafka 为应用服务，如下图所示。

| 告警设置 | | | | | | | |
|------|------|----|----|------|------|---|-------|
| 新增 | | | | | | | |
| 告警项 | 告警类型 | 条件 | 数值 | 告警级别 | 重复次数 | 域 | 操作 |
| | | | | | | | 编辑 删除 |
| | | | | | | | |
| | | | | | | | |

[操作历史>>](#)

这里只是给出一个设计的思路，以全局的视角展示一些监控信息项，至于全局的布局把握，在实现时还是要根据实际的情况来做具体的分析。至于相关的实现细节，我们不妨继续接着来看下面几个章节的内容。

监控数据的来源

要实现一个自定义的 Kafka 监控系统，首先得知道从哪里获取监控指标。Kafka 自身提供的监控指标（包括 broker 和主题的指标，而集群层面的指标可以通过各个 broker 的指标值累加来获得）都可以通过 JMX（Java Managent Extension, Java 管理扩展）来获取，在使用 JMX 之前需要确保 Kafka 开启了 JMX 的功能（默认关闭）。Kafka 在启动时需要通过配置 JMX_PORT 来设置 JMX 的端口号并以此来开启 JMX 的功能，示例如下：

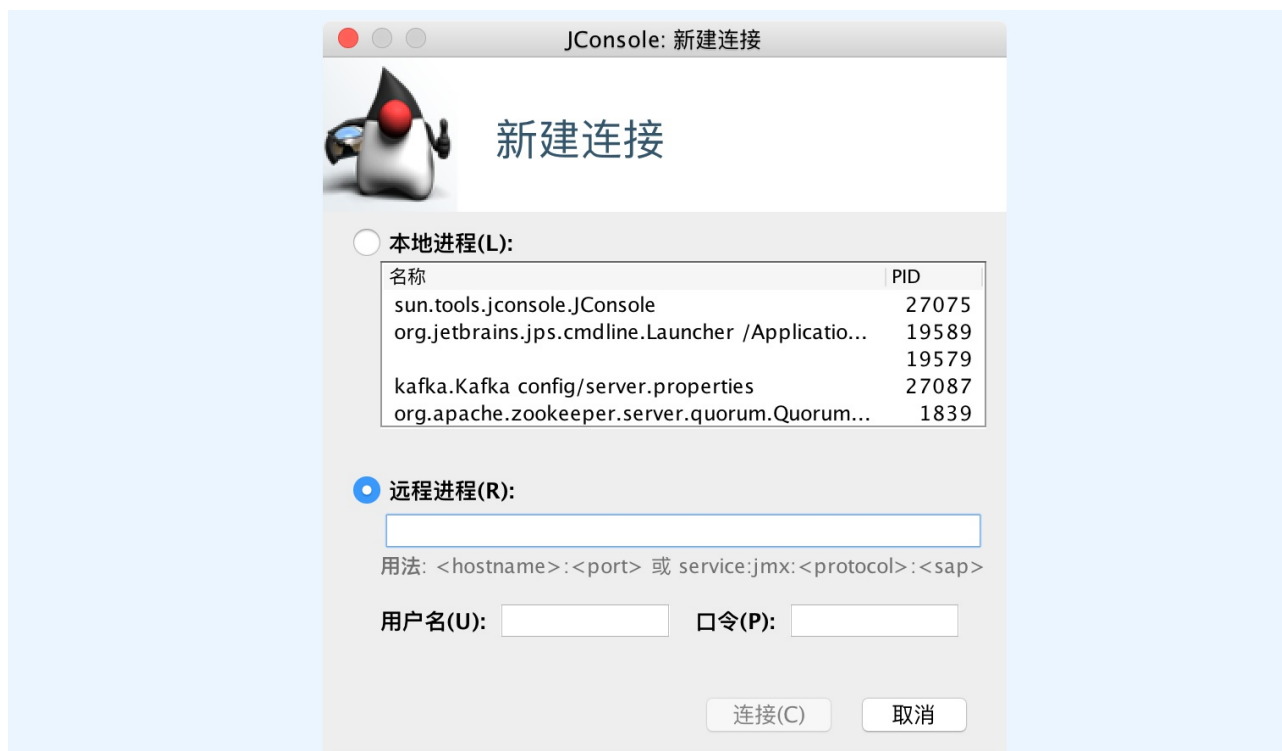
```
JMX_PORT=9999 nohup bin/kafka-server-start.sh
config/server.properties &
```

开启 JMX 之后会在 ZooKeeper 的 /brokers/ids/<brokerId> 节点中有对应的呈现（jmx_port 字段对应的值），示例如下：

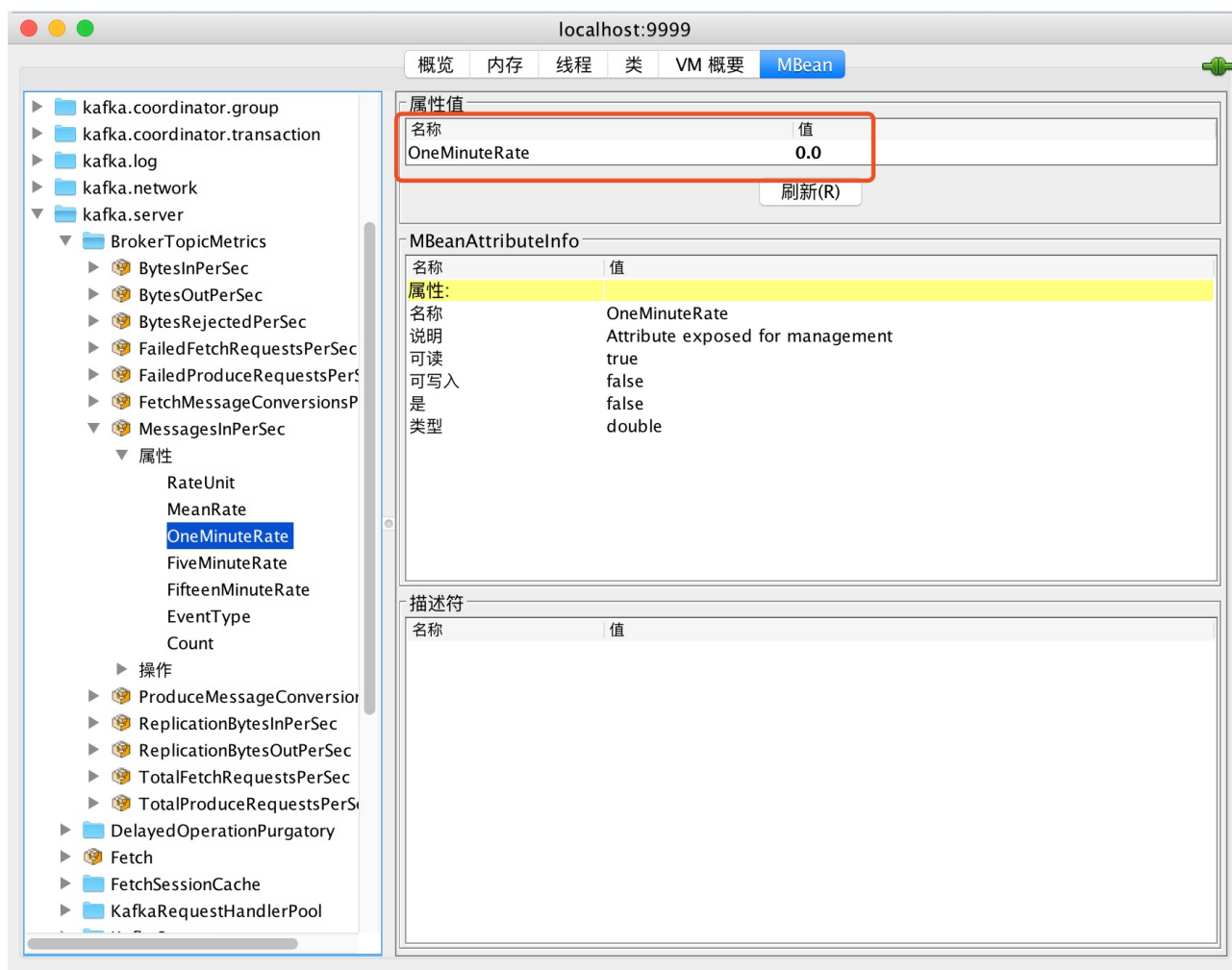
```
{"listener_security_protocol_map":
{"PLAINTEXT": "PLAINTEXT"}, "endpoints":
["PLAINTEXT://localhost:9092"], "jmx_port": 9999, "h
ost": "localhost", "timestamp": "1540025558270", "por
t": 9092, "version": 4}
```

开启 JMX 功能之后，最简单的获取监控指标的方式莫过于直接使用 Java 自带的工具 JConsole 了（仅对 Java 用户而言，如果读者不喜欢这个工具，可以试一下 Kafka 自带的 kafka.tools.JmxTool），上面我们设置了 JMX 的端口号为 9999（IP 地址为 localhost），那

么可以直接在 JConsole 中输入
service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi 或
localhost:9999 来连接 Kafka，如下图所示。



读者是否还记得在上一节中第2张图的右侧有一个 MsgIn 的指标，它表示当前 broker 中消息流入的速度，单位是条/s (messages/s)，而下图中的 kafka.server-BrokerTopicMetrics-MessagesInPerSec-OneMinuteRate 对应的就是这个指标在一分钟内的监控数值。



注意在 OneMinuteRate 同一级中还有 Count、FiveMinuteRate、FifteenMinuteRate、MeanRate、RateUnit 属性，它们与 OneMinuteRate 一起所对应的具体含义如下表所示。

| 属 性 名 称 | 属 性 含 义 |
|-------------------|---|
| Count | 消息流入的总数 |
| FiveMinuteRate | 5分钟内流入的平均速度 |
| FifteenMinuteRate | 15分钟内流入的平均速度 |
| EventType | 事件类型，对 MsgIn 而言固定为“messages”，表示消息个数；对于一些其他类型的指标，这时间类型的值会有所不同，比如对与 MessagesInPerSec 同一级别的 BytesInPerSec 而言，这个属性值为“bytes”，表示字节数 |
| OneMinuteRate | 1分钟内流入的平均速度 |

| | |
|----------|---|
| MeanRate | 平均速度 |
| RateUnit | 时间单位，值固定为 SECONDS，即“秒”，它和 EventType 组成这个指标的单位，即 messages/s，也就是条/s |

OneMinuteRate

OneMinuteRate 不是我们通常思维逻辑上的“一分钟内的平均速度”，而是一个受历史时刻影响的拟合值。如果通过程序计算某一个分钟内的平均速度值，那么有可能你会发现所得到的计算值与 OneMinuteRate 的值相差很大。

Kafka 是基于 Yammer Metrics 进行指标统计的，Yammer Metrics 是由 Yammer 提供的一个 Java 库，用于检测 JVM 上相关服务运行的状态，它对 OneMinuteRate 的定义如下所示。

Returns the one-minute exponentially-weighted moving average rate at which events have occurred since the meter was created.

由定义可知，OneMinuteRate 是一种指数加权移动平均值（学术上简称为 [EWMA \(https://en.wikipedia.org/wiki/EWMA_chart\)](https://en.wikipedia.org/wiki/EWMA_chart)）。在 Yammer Metrics 中关于 OneMinuteRate 的实现细节如下。首先定义 alpha 的值：

```
private static final double M1_ALPHA = 1.0D -  
Math.exp(-0.08333333333333333D);
```

这个 alpha 的值大概为 0.07995558537067671。
OneMinuteRate 的计算代码如下：

```

long count = this.uncounted.getAndSet(0L);
double instantRate = (double)count /
this.interval;
if(this.initialized) {
    this.rate += this.alpha * (instantRate -
this.rate);
} else {
    this.rate = instantRate;
    this.initialized = true;
}

```

可以简化为：

$$X[n] = X[n-1] + \alpha \cdot (X[\text{interval}] - X[n-1]) = \alpha \cdot X[\text{interval}] + (1-\alpha) \cdot X[n-1];$$

其中 $X[\text{interval}]$ 指的是在 $T[n-1]$ 至 $T[n]$ 时间内的真实测算值，或者可以认为是真实值。上面的公式可以换算为：

$$X[\text{当前预估值}] = \alpha \times X[\text{当前真实值}] + (1-\alpha) \times X[\text{上一时刻的预估值}]$$

鉴于 Yammer Metrics 中的 OneMinuteRate 的 α 值为 0.08 左右，所以这个 OneMinuteRate 的值特别“倚重”历史值。

还有两个类似的值 FiveMinuteRate 和 FifteenMinuteRate，它们的计算过程与 OneMinuteRate 一样，只是 α 的值不一样；

```

//0.01652854617838251
private static final double M5_ALPHA = 1.0D -
Math.exp(-0.016666666666666666D);
//0.005540151995103271
private static final double M15_ALPHA = 1.0D -
Math.exp(-0.005555555555555555D);

```


实际情况下，在发送速度起伏较大的时候，OneMinuteRate 的值与对应的一分钟内的真实值相差很大。如果发送速度趋于平缓并持续一段时间，那么 OneMinuteRate 的值才与真实值相匹配。读者在使用这个属性时需要熟记它背后代表的具体含义，避免在实际应用中产生偏差。

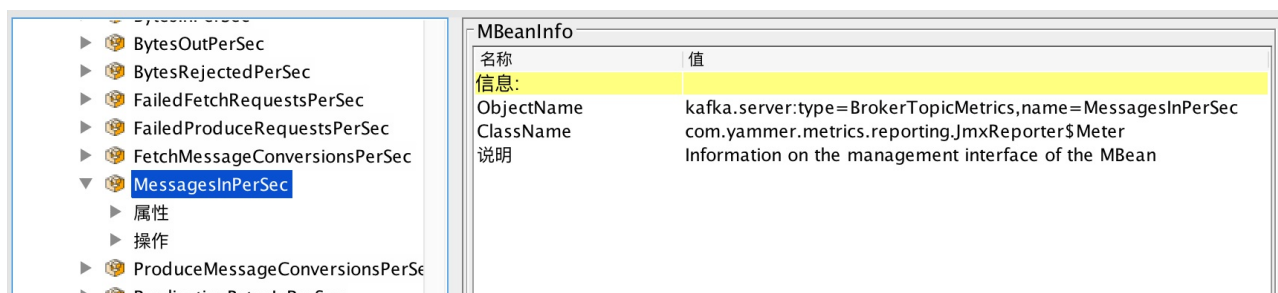
获取监控指标

前面我们了解了如何使用工具来连接 Kafka 并获取相关的监控指标，不过我们并不能指望在监控系统中嵌入这些工具来获取监控指标。Java 自身就包含了 JMX 的连接器，通过它就可以让我们能够用编程的手段来使监控系统很容易地获取相应的监控指标值。

在通过 JMX 获取某个具体的监控指标值之前需要指定对应的 JMX 指标（MBean）名称，同样以前面的 MsgIn 为例，它对应的 MBean 名称为：

```
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec
```

Kafka 自身提供的指标有很多（其余的会在后面的篇幅中详细说明），虽然它们的 MBean 名称一般都有规律可循，但是要记住这些内容也并非易事，在实际使用它们时可以通过 JConsole 工具来辅助获取。如下图所示，我们可以很容易找到 MessagesInPerSec（MsgIn），但要拼写成功整个 MBean 名称的话还需要费点精力，不如直接复制右侧 ObjectName 所对应的值。



注意，在“古老”的 Kafka 0.8.2.0 之前，MBean 名称的组织形式会有所不同，不过同样通过 JConsole 工具辅助来获取具体的 MBean 名称。代码清单31-1中给出了一个详细的示例来演示如何通过编程的手段获取 MsgIn 指标所对应的值。

```
//代码清单31-1 使用JMX来获取监控指标
import javax.management.*;
import javax.management.remote.JMXConnector;
import
javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import java.io.IOException;

public class JmxConnectionDemo {
    private MBeanServerConnection conn;
    private String jmxURL;
    private String ipAndPort;

    public JmxConnectionDemo(String ipAndPort) {
        this.ipAndPort = ipAndPort;
    }
    //初始化JMX连接
    public boolean init(){
        jmxURL = "service:jmx:rmi:///jndi/rmi://"
+ ipAndPort + "/jmxrmi";
        try {
            JMXServiceURL serviceURL = new
JMXServiceURL(jmxURL);
            JMXConnector connector =
JMXConnectorFactory
                .connect(serviceURL, null);
            conn =
connector.getMBeanServerConnection();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        if (conn == null) {
            return false;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return true;
}

//获取MsgIn指标的值
public double getMsgInPerSec() {
    //拼接MsgIn对应的MBean的名称
    String objectName =
        "kafka.server:type=BrokerTopicMetrics," +
            "name=MessagesInPerSec";
    Object val = getAttribute(objectName,
        "OneMinuteRate");
    if (val != null) {
        return (double) (Double) val;
    }
    return 0.0;
}

//根据MBean名称和属性来获取具体的值
private Object getAttribute(String objName,
    String objAttr) {
    ObjectName objectName;
    try {
        objectName = new ObjectName(objName);
        return conn.getAttribute(objectName,
            objAttr);
    } catch (MalformedObjectNameException |
        IOException |
            ReflectionException |
                InstanceNotFoundException |

```

```
        AttributeNotFoundException |
MBeanException e) {
    e.printStackTrace();
}
return null;
}
}
```

可以通过运行下面的程序来获取具体的指标值：

```
public static void main(String[] args) {
    JmxConnectionDemo jmxConnectionDemo =
        new
JmxConnectionDemo("localhost:9999");
    jmxConnectionDemo.init();

System.out.println(jmxConnectionDemo.getMsgInPerSec());
}
```

这段代码可以在 Github 上获取，请点击[这里](https://github.com/hiddenzzh/kafka_book_demo/blob/master)
(https://github.com/hiddenzzh/kafka_book_demo/blob/master)

上面的示例是获取了某个 broker 当前一分钟内消息流入的速度（messages/s），如果要计算整个集群的 MsgIn，那么只需将旗下的各个 broker 的 MsgIn 值累计即可。

通过 JMX 可以获取 Kafka 自身提供的运行状态指标，不过一些配置类信息（如各个 broker 的 IP 地址、端口号、JMX 端口号、AR 信息和 ISR 信息等）一般无法通过 JMX 来获取。对于 broker 的 IP 地址之类的信息，我们可以通过手动配置的方式来将其加入监控系统，但对于 AR 和 ISR 信息之类的信息，我们难以通过手动的方式来解决，这里可以借助 Kafka 连接的 ZooKeeper 来实现这些信息的获取，比如前面提及的 /brokers/ids/<brokerId> 节点。

除此之外，对于 Kafka 的一些硬件指标，比如 iowait、ioutil 等可以通过第三方工具如 Falcon、Zabbix 来获取。