1. Background

Currently, the implementation of monitoring is mostly based on Prometheus instrumentation. The existing simple metrics are sufficient for monitoring, but it is not easy to implement monitoring for part of other functions through this simple instrumentation.

2. Current Monitoring Overview

a. Direct Instrumentation

The direct instrumentation method involves explicitly calling the instrumentation function before and after certain code executions, such as using the MetricsUtil.counterInc method for cumulative instrumentation, or using the Metrics.counterInc method for metric accumulation instrumentation, for example:

```java
 public void applyBlock(BlockCapsule blockCapsule) {
  WitnessCapsule wc;
  long blockNum = blockCapsule.getNum();
  long blockTime = blockCapsule.getTimeStamp();
  byte[] blockWitness =
blockCapsule.getWitnessAddress().toByteArray();
  wc = consensusDelegate.getWitness(blockWitness);
  wc.setTotalProduced(wc.getTotalProduced() + 1);
  Metrics.counterInc(MetricKeys.Counter.MINER, 1,
StringUtil.encode58Check(blockWitness),
    MetricLabels.Counter.MINE_SUCCESS);
  wc.setLatestBlockNum(blockNum);
  wc.setLatestSlotNum(dposSlot.getAbSlot(blockTime));
  consensusDelegate.saveWitness(wc);
```

b. Aspect Instrumentation

Some methods require instrumentation and statistical information before and after their execution. In order to avoid adding instrumentation code repeatedly at the beginning and end of these methods, aspects are defined. These methods are configured as join points using

wildcard matching, and the instrumentation function is executed before and after these methods using AOP.

```java
 @Around("execution(public * org.tron.core.Wallet.*(..))")
 public Object walletAroundAdvice(ProceedingJoinPoint pjp) throws
Throwable {
  Object result;
  Histogram.Timer requestTimer = Metrics.histogramStartTimer(
    MetricKeys.Histogram.INTERNAL_SERVICE_LATENCY,
    pjp.getSignature().getDeclaringType().getSimpleName(),
    pjp.getSignature().getName());
  try {
   result = pjp.proceed();
  } catch (Throwable throwable) {
   Metrics.counterInc(
     MetricKeys.Counter.INTERNAL_SERVICE_FAIL, 1,
     pjp.getSignature().getDeclaringType().getSimpleName(),
     pjp.getSignature().getName());
   throw throwable;
  } finally {
   Metrics.histogramObserve(requestTimer);
  }
  return result;
 }
```

c. Exporter Approach

A custom exporter is defined by extending the exporter functionality template provided by Prometheus. This actually defines a way to provide data. Obtaining the specific data can be implemented by extending the corresponding methods. Prometheus retrieves data provided by these exporters through polling.

```java

 @Override
 public List<MetricFamilySamples> collect(Predicate<String>
nameFilter) {
```

```
  List<MetricFamilySamples> mfs = new
ArrayList<MetricFamilySamples>();
  addOperatingSystemMetrics(mfs, nameFilter == null ? ALLOW_ALL :
nameFilter);
  return mfs;
 }
```

d.Logger

Logs are collected and analyzed by defining an appender and utilizing the extension functionality provided by the logger logging framework.

Java

```
public class InstrumentedAppender extends
UnsynchronizedAppenderBase<ILoggingEvent> {
 public static final String COUNTER_NAME = "tron:error_info";

 private static final Counter defaultCounter =
Counter.build().name(COUNTER_NAME)
   .help("tron log statements at error type levels")
   .labelNames("topic", "type")
   .register();

 @Override
 protected void append(ILoggingEvent event) {
  if (Metrics.enabled() && event.getLevel().toInt() ==
Level.ERROR_INT) {
   String type = MetricLabels.UNDEFINED;
   if (event.getThrowableProxy() != null) {
    type = event.getThrowableProxy().getClassName();
    type = type.substring(type.lastIndexOf(".") + 1);
   }
   defaultCounter.labels(event.getLoggerName(), type).inc();
  }
 }
}
```

3. Implementation of New Monitoring Features

Currently, new monitoring features can be implemented using existing instrumentation methods and custom extensions, as follows:

Direct instrumentation is the most commonly used method for metric statistics in the current application. Most metric statistics can be achieved using direct instrumentation. For instrumentation of multiple stores of the same type, the instrumentation function can be implemented in the upper abstract class (such as TronStoreWithRevoking, and TronDatabase).

Note:

Branch GitHub - tronprotocol/java-tron at feature/metrics

Please be aware that only instrumentation requirements can be added to this branch

and any processing logic should not be changed or affected.

1. Submit the instrumentation code to this branch.
2. Login to the node.
    a. cd /data/FullNode
    b. sh build.sh
    c. sh start.sh
3. After the restart is completed, verify by running:
    a. curl localhost:9527/metrics | grep [new instrumentation keyword]
4. Open Grafana and create a new instrumentation panel.