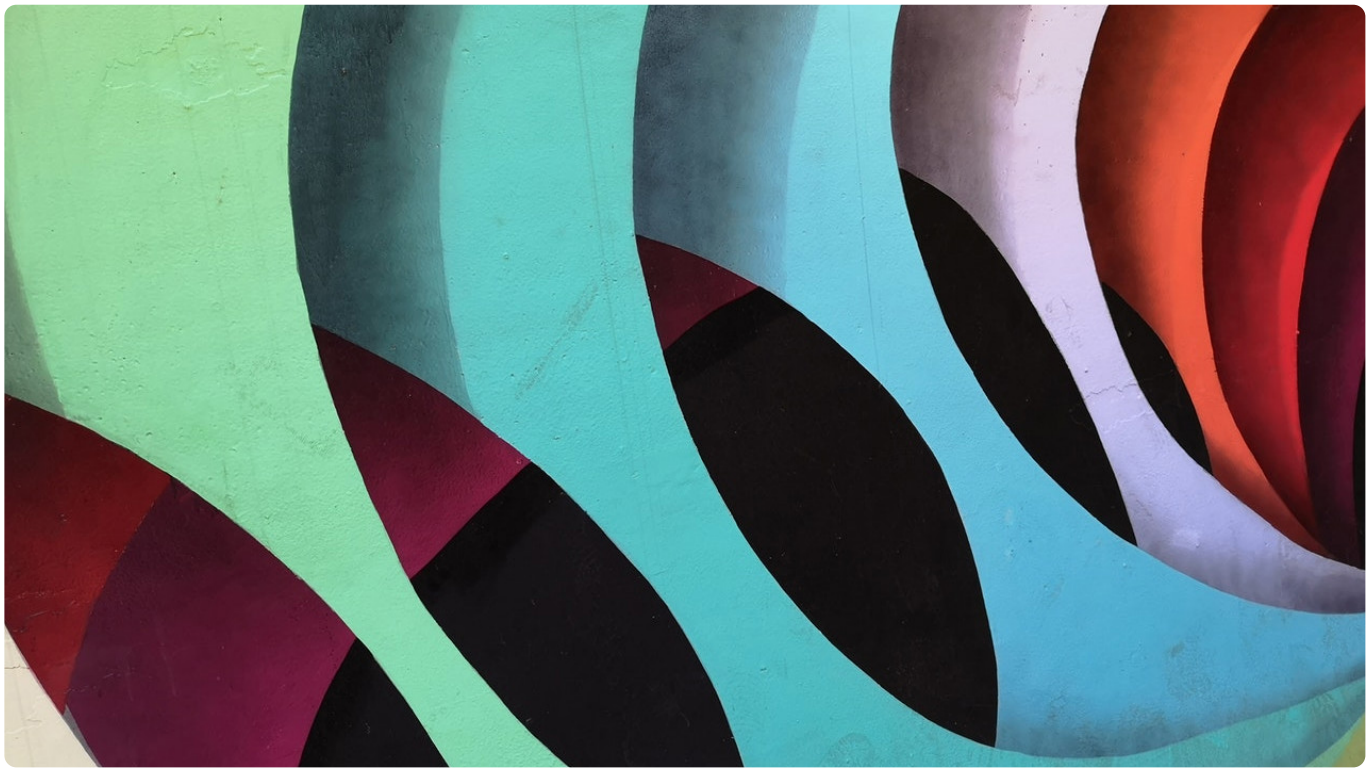
40 | 运用学过的设计原则和思想完善之前讲的性能计数器项目（下）

2020-02-03 王争



讲述：冯永吉

时长 15:17　大小 12.26M

上一节课中，我们针对版本 1 存在的问题（特别是 Aggregator 类、ConsoleReporter 和 EmailReporter 类）进行了重构优化。经过重构之后，代码结构更加清晰、合理、有逻辑性。不过，在细节方面还是存在一些问题，比如 ConsoleReporter、EmailReporter 类仍然存在代码重复、可测试性差的问题。今天，我们就在版本 3 中持续重构这部分代码。

除此之外，在版本 3 中，我们还会继续完善框架的功能和非功能需求。比如，让原始数据的采集和存储异步执行，解决聚合统计在数据量大的情况下会导致内存吃紧问题，以及提高框架的易用性等，让它成为一个能用且好用的框架。

话不多说，让我们正式开始版本 3 的设计与实现吧！

代码重构优化

我们知道，继承能解决代码重复的问题。我们可以将 ConsoleReporter 和 EmailReporter 中的相同代码逻辑，提取到父类 ScheduledReporter 中，以解决代码重复问题。按照这个思路，重构之后的代码如下所示：

复制代码

```
public abstract class ScheduledReporter {
  protected MetricsStorage metricsStorage;
  protected Aggregator aggregator;
  protected StatViewer viewer;

  public ScheduledReporter(MetricsStorage metricsStorage, Aggregator aggregato
    this.metricsStorage = metricsStorage;
    this.aggregator = aggregator;
    this.viewer = viewer;
  }

  protected void doStatAndReport(long startTimeInMillis, long endTimeInMillis)
    long durationInMillis = endTimeInMillis -  startTimeInMillis;
    Map<String, List<RequestInfo>> requestInfos =
            metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis)
    Map<String, RequestStat> requestStats = aggregator.aggregate(requestInfos,
    viewer.output(requestStats, startTimeInMillis, endTimeInMillis);
  }

}
```

ConsoleReporter 和 EmailReporter 代码重复的问题解决了，那我们再来看一下代码的可测试性问题。因为 ConsoleReporter 和 EmailReporter 的代码比较相似，且 EmailReporter 的代码更复杂些，所以，关于如何重构来提高其可测试性，我们拿 EmailReporter 来举例说明。将重复代码提取到父类 ScheduledReporter 之后，EmailReporter 代码如下所示：

复制代码

```
public class EmailReporter extends ScheduledReporter {
  private static final Long DAY_HOURS_IN_SECONDS = 86400L;

  private MetricsStorage metricsStorage;
  private Aggregator aggregator;
  private StatViewer viewer;

  public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, S
    this.metricsStorage = metricsStorage;
```

```
    this.aggregator = aggregator;
    this.viewer = viewer;
  }

  public void startDailyReport() {
    Calendar calendar = Calendar.getInstance();
    calendar.add(Calendar.DATE, 1);
    calendar.set(Calendar.HOUR_OF_DAY, 0);
    calendar.set(Calendar.MINUTE, 0);
    calendar.set(Calendar.SECOND, 0);
    calendar.set(Calendar.MILLISECOND, 0);
    Date firstTime = calendar.getTime();

    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      @Override
      public void run() {
        long durationInMillis = DAY_HOURS_IN_SECONDS * 1000;
        long endTimeInMillis = System.currentTimeMillis();
        long startTimeInMillis = endTimeInMillis - durationInMillis;
        doStatAndReport(startTimeInMillis, endTimeInMillis);
      }
    }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
  }
}
```

前面提到，之所以 EmailReporter 可测试性不好，一方面是因为用到了线程（定时器也相当于多线程），另一方面是因为涉及时间的计算逻辑。

实际上，在经过上一步的重构之后，EmailReporter 中的 startDailyReport() 函数的核心逻辑已经被抽离出去了，较复杂的、容易出 bug 的就只剩下计算 firstTime 的那部分代码了。我们可以将这部分代码继续抽离出来，封装成一个函数，然后，单独针对这个函数写单元测试。重构之后的代码如下所示：

复制代码

```
public class EmailReporter extends ScheduledReporter {
  // 省略其他代码...
  public void startDailyReport() {
    Date firstTime = trimTimeFieldsToZeroOfNextDay();
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      @Override
      public void run() {
        // 省略其他代码...
      }
```

```
    }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
  }

  // 设置成protected而非private是为了方便写单元测试
  @VisibleForTesting
  protected Date trimTimeFieldsToZeroOfNextDay() {
    Calendar calendar = Calendar.getInstance(); // 这里可以获取当前时间
    calendar.add(Calendar.DATE, 1);
    calendar.set(Calendar.HOUR_OF_DAY, 0);
    calendar.set(Calendar.MINUTE, 0);
    calendar.set(Calendar.SECOND, 0);
    calendar.set(Calendar.MILLISECOND, 0);
    return calendar.getTime();
  }
}
```

简单的代码抽离成 trimTimeFieldsToZeroOfNextDay() 函数之后，虽然代码更加清晰了，一眼就能从名字上知道这段代码的意图（获取当前时间的下一天的 0 点时间），但我们发现这个函数的可测试性仍然不好，因为它强依赖当前的系统时间。实际上，这个问题挺普遍的。一般的解决方法是，将强依赖的部分通过参数传递进来，这有点类似我们之前讲的依赖注入。按照这个思路，我们再对 trimTimeFieldsToZeroOfNextDay() 函数进行重构。重构之后的代码如下所示：

复制代码

```
public class EmailReporter extends ScheduledReporter {
  // 省略其他代码...
  public void startDailyReport() {
    // new Date()可以获取当前时间
    Date firstTime = trimTimeFieldsToZeroOfNextDay(new Date());
    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      @Override
      public void run() {
        // 省略其他代码...
      }
    }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
  }

  protected Date trimTimeFieldsToZeroOfNextDay(Date date) {
    Calendar calendar = Calendar.getInstance(); // 这里可以获取当前时间
    calendar.setTime(date); // 重新设置时间
    calendar.add(Calendar.DATE, 1);
    calendar.set(Calendar.HOUR_OF_DAY, 0);
    calendar.set(Calendar.MINUTE, 0);
    calendar.set(Calendar.SECOND, 0);
    calendar.set(Calendar.MILLISECOND, 0);
```

```
23      return calendar.getTime();
24    }
25  }
```

经过这次重构之后，trimTimeFieldsToZeroOfNextDay() 函数不再强依赖当前的系统时间，所以非常容易对其编写单元测试。你可以把它作为练习，写一下这个函数的单元测试。

不过，EmailReporter 类中 startDailyReport() 还是涉及多线程，针对这个函数该如何写单元测试呢？我的看法是，这个函数不需要写单元测试。为什么这么说呢？我们可以回到写单元测试的初衷来分析这个问题。单元测试是为了提高代码质量，减少 bug。如果代码足够简单，简单到 bug 无处隐藏，那我们就没必要为了写单元测试而写单元测试，或者为了追求单元测试覆盖率而写单元测试。经过多次代码重构之后，startDailyReport() 函数里面已经没有多少代码逻辑了，所以，完全没必要对它写单元测试了。

功能需求完善

经过了多个版本的迭代、重构，我们现在来重新 Review 一下，目前的设计与实现是否已经完全满足第 25 讲中最初的功能需求了。

最初的功能需求描述是下面这个样子的，我们来重新看一下。

> 我们希望设计开发一个小的框架，能够获取接口调用的各种统计信息，比如响应时间的最大值（max）、最小值（min）、平均值（avg）、百分位值（percentile），接口调用次数（count）、频率（tps） 等，并且支持将统计结果以各种显示格式（比如：JSON 格式、网页格式、自定义显示格式等）输出到各种终端（Console 命令行、HTTP 网页、Email、日志文件、自定义输出终端等），以方便查看。

经过整理拆解之后的需求列表如下所示：

> 接口统计信息：包括接口响应时间的统计信息，以及接口调用次数的统计信息等。
> 统计信息的类型：max、min、avg、percentile、count、tps 等。
> 统计信息显示格式：JSON、HTML、自定义显示格式。
> 统计信息显示终端：Console、Email、HTTP 网页、日志、自定义显示终端。

经过挖掘，我们还得到一些隐藏的需求，如下所示：

> 统计触发方式：包括主动和被动两种。主动表示以一定的频率定时统计数据，并主动推送到显示终端，比如邮件推送。被动表示用户触发统计，比如用户在网页中选择要统计的时间区间，触发统计，并将结果显示给用户。

> 统计时间区间：框架需要支持自定义统计时间区间，比如统计最近 10 分钟的某接口的 tps、访问次数，或者统计 12 月 11 日 00 点到 12 月 12 日 00 点之间某接口响应时间的最大值、最小值、平均值等。

> 统计时间间隔：对于主动触发统计，我们还要支持指定统计时间间隔，也就是多久触发一次统计显示。比如，每间隔 10s 统计一次接口信息并显示到命令行中，每间隔 24 小时发送一封统计信息邮件。

版本 3 已经实现了大部分的功能，还有以下几个小的功能点没有实现。你可以将这些还没有实现的功能，自己实现一下，继续迭代出框架的第 4 个版本。

> 被动触发统计的方式，也就是需求中提到的通过网页展示统计信息。实际上，这部分代码的实现也并不难。我们可以复用框架现在的代码，编写一些展示页面和提供获取统计数据的接口即可。
>
> 对于自定义显示终端，比如显示数据到自己开发的监控平台，这就有点类似通过网页来显示数据，不过更加简单些，只需要提供一些获取统计数据的接口，监控平台通过这些接口拉取数据来显示即可。
>
> 自定义显示格式。在框架现在的代码实现中，显示格式和显示终端（比如 Console、Email）是紧密耦合在一起的，比如，Console 只能通过 JSON 格式来显示统计数据，Email 只能通过某种固定的 HTML 格式显示数据，这样的设计还不够灵活。我们可以将显示格式设计成独立的类，将显示终端和显示格式的代码分离，让显示终端支持配置不同的显示格式。具体的代码实现留给你自己思考，我这里就不多说了。

非功能需求完善

Review 完了功能需求的完善程度，现在，我们再来看，版本 3 的非功能性需求的完善程度。在第 25 讲中，我们提到，针对这个框架的开发，我们需要考虑的非功能性需求包括：易用性、性能、扩展性、容错性、通用性。我们现在就依次来看一下这几个方面。

1. 易用性

所谓的易用性，顾名思义，就是框架是否好用。框架的使用者将框架集成到自己的系统中时，主要用到 MetricsCollector 和 EmailReporter、ConsoleReporter 这几个类。通过 MetricsCollector 类来采集数据，通过 EmailReporter、ConsoleReporter 类来触发主动统计数据、显示统计结果。示例代码如下所示：

复制代码

```
public class PerfCounterTest {
  public static void main(String[] args) {
    MetricsStorage storage = new RedisMetricsStorage();
    Aggregator aggregator = new Aggregator();

    // 定时触发统计并将结果显示到终端
    ConsoleViewer consoleViewer = new ConsoleViewer();
    ConsoleReporter consoleReporter = new ConsoleReporter(storage, aggregator,
    consoleReporter.startRepeatedReport(60, 60);

    // 定时触发统计并将结果输出到邮件
    EmailViewer emailViewer = new EmailViewer();
    emailViewer.addToAddress("wangzheng@xzg.com");
    EmailReporter emailReporter = new EmailReporter(storage, aggregator, email'
    emailReporter.startDailyReport();

    // 收集接口访问数据
    MetricsCollector collector = new MetricsCollector(storage);
    collector.recordRequest(new RequestInfo("register", 123, 10234));
    collector.recordRequest(new RequestInfo("register", 223, 11234));
    collector.recordRequest(new RequestInfo("register", 323, 12334));
    collector.recordRequest(new RequestInfo("login", 23, 12434));
    collector.recordRequest(new RequestInfo("login", 1223, 14234));

    try {
      Thread.sleep(100000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

从上面的使用示例中，我们可以看出，框架用起来还是稍微有些复杂的，需要组装各种类，比如需要创建 MetricsStorage 对象、Aggregator 对象、ConsoleViewer 对象，然后注入到 ConsoleReporter 中，才能使用 ConsoleReporter。除此之外，还有可能存在误用的情况，比如把 EmailViewer 传递进了 ConsoleReporter 中。总体上来讲，框架的使用方式暴露了太多细节给用户，过于灵活也带来了易用性的降低。

为了让框架用起来更加简单（能将组装的细节封装在框架中，不暴露给框架使用者），又不失灵活性（可以自由组装不同的 MetricsStorage 实现类、StatViewer 实现类到 ConsoleReporter 或 EmailReporter），也不降低代码的可测试性（通过依赖注入来组装类，方便在单元测试中 mock），我们可以额外地提供一些封装了默认依赖的构造函数，让使用者自主选择使用哪种构造函数来构造对象。这段话理解起来有点复杂，我把按照这个思路重构之后的代码放到了下面，你可以结合着一块看一下。

复制代码

```
public class MetricsCollector {
  private MetricsStorage metricsStorage;

  // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
  public MetricsCollectorB() {
    this(new RedisMetricsStorage());
  }

  // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
  public MetricsCollectorB(MetricsStorage metricsStorage) {
    this.metricsStorage = metricsStorage;
  }
  // 省略其他代码...
}

public class ConsoleReporter extends ScheduledReporter {
  private ScheduledExecutorService executor;

  // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
  public ConsoleReporter() {
    this(new RedisMetricsStorage(), new Aggregator(), new ConsoleViewer());
  }

  // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
  public ConsoleReporter(MetricsStorage metricsStorage, Aggregator aggregator,
    super(metricsStorage, aggregator, viewer);
    this.executor = Executors.newSingleThreadScheduledExecutor();
  }
  // 省略其他代码...
}

public class EmailReporter extends ScheduledReporter {
  private static final Long DAY_HOURS_IN_SECONDS = 86400L;

  // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
  public EmailReporter(List<String> emailToAddresses) {
    this(new RedisMetricsStorage(), new Aggregator(), new EmailViewer(emailToA
  }

  // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
```

```
  public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, S
    super(metricsStorage, aggregator, viewer);
  }
  // 省略其他代码...
}
```

现在，我们再来看下框架如何来使用。具体使用示例如下所示。看起来是不是简单多了呢？

复制代码

```
public class PerfCounterTest {
  public static void main(String[] args) {
    ConsoleReporter consoleReporter = new ConsoleReporter();
    consoleReporter.startRepeatedReport(60, 60);

    List<String> emailToAddresses = new ArrayList<>();
    emailToAddresses.add("wangzheng@xzg.com");
    EmailReporter emailReporter = new EmailReporter(emailToAddresses);
    emailReporter.startDailyReport();

    MetricsCollector collector = new MetricsCollector();
    collector.recordRequest(new RequestInfo("register", 123, 10234));
    collector.recordRequest(new RequestInfo("register", 223, 11234));
    collector.recordRequest(new RequestInfo("register", 323, 12334));
    collector.recordRequest(new RequestInfo("login", 23, 12434));
    collector.recordRequest(new RequestInfo("login", 1223, 14234));

    try {
      Thread.sleep(100000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

如果你足够细心，可能已经发现，RedisMeticsStorage 和 EmailViewer 还需要另外一些配置信息才能构建成功，比如 Redis 的地址，Email 邮箱的 POP3 服务器地址、发送地址。这些配置并没有在刚刚代码中体现到，那我们该如何获取呢？

我们可以将这些配置信息放到配置文件中，在框架启动的时候，读取配置文件中的配置信息到一个 Configuration 单例类。RedisMetricsStorage 类和 EmailViewer 类都可以从这个 Configuration 类中获取需要的配置信息来构建自己。

2. 性能

对于需要集成到业务系统的框架来说，我们不希望框架本身代码的执行效率，对业务系统有太多性能上的影响。对于性能计数器这个框架来说，一方面，我们希望它是低延迟的，也就是说，统计代码不影响或很少影响接口本身的响应时间；另一方面，我们希望框架本身对内存的消耗不能太大。

对于性能这一点，落实到具体的代码层面，需要解决两个问题，也是我们之前提到过的，一个是采集和存储要异步来执行，因为存储基于外部存储（比如 Redis），会比较慢，异步存储可以降低对接口响应时间的影响。另一个是当需要聚合统计的数据量比较大的时候，一次性加载太多的数据到内存，有可能会导致内存吃紧，甚至内存溢出，这样整个系统都会瘫痪掉。

针对第一个问题，我们通过在 MetricsCollector 中引入 Google Guava EventBus 来解决。实际上，我们可以把 EventBus 看作一个“生产者 - 消费者”模型或者“发布 - 订阅”模型，采集的数据先放入内存共享队列中，另一个线程读取共享队列中的数据，写入到外部存储（比如 Redis）中。具体的代码实现如下所示：

复制代码

```
public class MetricsCollector {
  private static final int DEFAULT_STORAGE_THREAD_POOL_SIZE = 20;

  private MetricsStorage metricsStorage;
  private EventBus eventBus;

  public MetricsCollector(MetricsStorage metricsStorage) {
    this(metricsStorage, DEFAULT_STORAGE_THREAD_POOL_SIZE);
  }

  public MetricsCollector(MetricsStorage metricsStorage, int threadNumToSaveDa
    this.metricsStorage = metricsStorage;
    this.eventBus = new AsyncEventBus(Executors.newFixedThreadPool(threadNumTo
    this.eventBus.register(new EventListener());
  }

  public void recordRequest(RequestInfo requestInfo) {
    if (requestInfo == null || StringUtils.isBlank(requestInfo.getApiName()))
      return;
    }
    eventBus.post(requestInfo);
  }

  public class EventListener {
```

```
    @Subscribe
    public void saveRequestInfo(RequestInfo requestInfo) {
      metricsStorage.saveRequestInfo(requestInfo);
    }
  }
}
```

针对第二个问题，解决的思路比较简单，但代码实现稍微有点复杂。当统计的时间间隔较大的时候，需要统计的数据量就会比较大。我们可以将其划分为一些小的时间区间（比如 10 分钟作为一个统计单元），针对每个小的时间区间分别进行统计，然后将统计得到的结果再进行聚合，得到最终整个时间区间的统计结果。不过，这个思路只适合响应时间的 max、min、avg，及其接口请求 count、tps 的统计，对于响应时间的 percentile 的统计并不适用。

对于 percentile 的统计要稍微复杂一些，具体的解决思路是这样子的：我们分批从 Redis 中读取数据，然后存储到文件中，再根据响应时间从小到大利用外部排序算法来进行排序（具体的实现方式可以看一下《数据结构与算法之美》专栏）。排序完成之后，再从文件中读取第 count*percentile（count 表示总的数据个数，percentile 就是百分比，99 百分位就是 0.99）个数据，就是对应的 percentile 响应时间。

这里我只给出了除了 percentile 之外的统计信息的计算代码，如下所示。对于 percentile 的计算，因为代码量比较大，留给你自己实现。

复制代码

```
public class ScheduleReporter {
  private static final long MAX_STAT_DURATION_IN_MILLIS = 10 * 60 * 1000; // 1(

  protected MetricsStorage metricsStorage;
  protected Aggregator aggregator;
  protected StatViewer viewer;

  public ScheduleReporter(MetricsStorage metricsStorage, Aggregator aggregator
    this.metricsStorage = metricsStorage;
    this.aggregator = aggregator;
    this.viewer = viewer;
  }

  protected void doStatAndReport(long startTimeInMillis, long endTimeInMillis)
    Map<String, RequestStat> stats = doStat(startTimeInMillis, endTimeInMillis
    viewer.output(stats, startTimeInMillis, endTimeInMillis);
  }
```

```
18  private Map<String, RequestStat> doStat(long startTimeInMillis, long endTime
19    Map<String, List<RequestStat>> segmentStats = new HashMap<>();
20    long segmentStartTimeMillis = startTimeInMillis;
21    while (segmentStartTimeMillis < endTimeInMillis) {
22      long segmentEndTimeMillis = segmentStartTimeMillis + MAX_STAT_DURATION_I
23      if (segmentEndTimeMillis > endTimeInMillis) {
24        segmentEndTimeMillis = endTimeInMillis;
25      }
26      Map<String, List<RequestInfo>> requestInfos =
27              metricsStorage.getRequestInfos(segmentStartTimeMillis, segmentEn
28      if (requestInfos == null || requestInfos.isEmpty()) {
29        continue;
30      }
31      Map<String, RequestStat> segmentStat = aggregator.aggregate(
32              requestInfos, segmentEndTimeMillis - segmentStartTimeMillis);
33      addStat(segmentStats, segmentStat);
34      segmentStartTimeMillis += MAX_STAT_DURATION_IN_MILLIS;
35    }
36
37    long durationInMillis = endTimeInMillis - startTimeInMillis;
38    Map<String, RequestStat> aggregatedStats = aggregateStats(segmentStats, du
39    return aggregatedStats;
40  }
41
42  private void addStat(Map<String, List<RequestStat>> segmentStats,
43                       Map<String, RequestStat> segmentStat) {
44    for (Map.Entry<String, RequestStat> entry : segmentStat.entrySet()) {
45      String apiName = entry.getKey();
46      RequestStat stat = entry.getValue();
47      List<RequestStat> statList = segmentStats.putIfAbsent(apiName, new Array
48      statList.add(stat);
49    }
50  }
51
52  private Map<String, RequestStat> aggregateStats(Map<String, List<RequestStat
53                                                  long durationInMillis) {
54    Map<String, RequestStat> aggregatedStats = new HashMap<>();
55    for (Map.Entry<String, List<RequestStat>> entry : segmentStats.entrySet())
56      String apiName = entry.getKey();
57      List<RequestStat> apiStats = entry.getValue();
58      double maxRespTime = Double.MIN_VALUE;
59      double minRespTime = Double.MAX_VALUE;
60      long count = 0;
61      double sumRespTime = 0;
62      for (RequestStat stat : apiStats) {
63        if (stat.getMaxResponseTime() > maxRespTime) maxRespTime = stat.getMax
64        if (stat.getMinResponseTime() < minRespTime) minRespTime = stat.getMin
65        count += stat.getCount();
66        sumRespTime += (stat.getCount() * stat.getAvgResponseTime());
67      }
68      RequestStat aggregatedStat = new RequestStat();
69      aggregatedStat.setMaxResponseTime(maxRespTime);
```

```
70          aggregatedStat.setMinResponseTime(minRespTime);
71          aggregatedStat.setAvgResponseTime(sumRespTime / count);
72          aggregatedStat.setCount(count);
73          aggregatedStat.setTps(count / durationInMillis * 1000);
74          aggregatedStats.put(apiName, aggregatedStat);
75        }
76        return aggregatedStats;
77      }
78  }
79
```

3. 扩展性

前面我们提到，框架的扩展性有别于代码的扩展性，是从使用者的角度来讲的，特指使用者可以在不修改框架源码，甚至不拿到框架源码的情况下，为框架扩展新的功能。

在刚刚讲到框架的易用性的时候，我们给出了框架如何使用的代码示例。从示例中，我们可以发现，框架在兼顾易用性的同时，也可以灵活地替换各种类对象，比如 MetricsStorage、StatViewer。举个例子来说，如果我们要让框架基于 HBase 来存储原始数据而非 Redis，那我们只需要设计一个实现 MetricsStorage 接口的 HBaseMetricsStorage 类，传递给 MetricsCollector 和 ConsoleReporter、EmailReporter 类即可。

4. 容错性

容错性这一点也非常重要。对于这个框架来说，不能因为框架本身的异常导致接口请求出错。所以，对框架可能存在的各种异常情况，我们都要考虑全面。

在现在的框架设计与实现中，采集和存储是异步执行，即便 Redis 挂掉或者写入超时，也不会影响到接口的正常响应。除此之外，Redis 异常，可能会影响到数据统计显示（也就是 ConsoleReporter、EmailReporter 负责的工作），但并不会影响到接口的正常响应。

5. 通用性

为了提高框架的复用性，能够灵活应用到各种场景中，框架在设计的时候，要尽可能通用。我们要多去思考一下，除了接口统计这样一个需求，这个框架还可以适用到其他哪些场景中。比如是否还可以处理其他事件的统计信息，比如 SQL 请求时间的统计、业务统计（比如支付成功率）等。关于这一点，我们在现在的版本 3 中暂时没有考虑到，你可以自己思考一下。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

还记得吗？在第 25、26 讲中，我们提到，针对性能计数器这个框架的开发，要想一下子实现我们罗列的所有功能，对任何人来说都是比较有挑战的。而经过这几个版本的迭代之后，我们不知不觉地就完成了几乎所有的需求，包括功能性和非功能性的需求。

在第 25 讲中，我们实现了一个最小原型，虽然非常简陋，所有的代码都塞在一个类中，但它帮我们梳理清楚了需求。在第 26 讲中，我们实现了框架的第 1 个版本，这个版本只包含最基本的功能，并且初步利用面向对象的设计方法，把不同功能的代码划分到了不同的类中。

在第 39 讲中，我们实现了框架的第 2 个版本，这个版本对第 1 个版本的代码结构进行了比较大的调整，让整体代码结构更加合理、清晰、有逻辑性。

在第 40 讲中，我们实现了框架的第 3 个版本，对第 2 个版本遗留的细节问题进行了重构，并且重点解决了框架的易用性和性能问题。

从上面的迭代过程，我们可以发现，大部分情况下，我们都是针对问题解决问题，每个版本都聚焦一小部分问题，所以整个过程也没有感觉到有太大难度。尽管我们迭代了 3 个版本，但目前的设计和实现还有很多值得进一步优化和完善的地方，但限于专栏的篇幅，继续优化的工作留给你自己来完成。

最后，我希望你不仅仅关注这个框架本身的设计和实现，更重要的是学会这个逐步优化的方法，以及其中涉及的一些编程技巧、设计思路，能够举一反三地用在其他项目中。

课堂讨论

最后，还是给你留一道课堂讨论题。

正常情况下，ConsoleReporter 的 startRepeatedReport() 函数只会被调用一次。但是，如果被多次调用，那就会存在问题。具体会有什么问题呢？又该如何解决呢？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

上一篇　39 | 运用学过的设计原则和思想完善之前讲的性能计数器项目（上）

下一篇　41 | 单例模式（上）：为什么说支持懒加载的双重检测不比饿汉式更优？

精选留言 (10)

写留言

辣么大
2020-02-03

思考题：startRepeatedReport()多次调用，会启动多个线程，每个线程都会执行统计和输出工作。
想了一种简单的实现方式，将runnable做为成员变量，第一次调用startRepeatedReport()时初始化，若多次调用，判空，返回。
public void startRepeatedReport(long periodInSeconds, long durationInSeconds...

展开

4

Jxin
2020-02-03

先回答问题：
1.会导致多余线程做多余的统计和展示。因为每次调用都会起一个异步线程输出统计数据到控制台。这样既会带来额外的性能开销，又会导致统计信息不易阅读。

2.在ConsoleReporter内部维护一个可视字段 started。然后在方法执行时，优先判断该...

展开

4

3

undefined
2020-02-03

深入浅出，过瘾。

展开

2

小晏子
2020-02-03

课后思考：如果 startRepeatedReport()被多次调用，那么会生成多个线程以fixed rate去请求然后输出结果到console上，一方面导致输出结果混乱，另一方面增加了系统的负担。要解决该问题有个办法是再重构一下代码，示意如下（未测试），
```

private Future<?> future; ...

展开

2
```

高源
2020-02-03

老师39,40课完整源代码可以提供下吗，我准备好好研究学习下

2

守拙
2020-02-04

课堂讨论:

正常情况下，ConsoleReporter 的 startRepeatedReport() 函数只会被调用一次。但是...

展开

1

Andy
2020-02-04

老师能提供课程代码吗?

展开

1

javaadu
2020-02-03

课堂讨论，使用一个标记flag作为该函数被调用国的标记，并给这个函数加锁，解决并发问题

1

平风造雨
2020-02-03

调用多次可以通过多线程共享的状态变量来解决，CAS或者加锁进行状态的变更。

1

Jeff.Smile
2020-02-03

沙发，打卡！一路跟进！

展开

1