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）

在第25节、第26节中，我们讲了如何对一个性能计数器框架进行分析、设计与实现，并且实践了之前学过的一些设计原则和设计思想。当时我们提到，小步快跑、逐步迭代是一种非常实用的开发模式。所以，针对这个框架的开发，我们分多个版本来逐步完善。

在第25、26节课中，我们实现了框架的第一个版本，它只包含最基本的一些功能，在设计与实现上还有很多不足。所以，接下来，我会针对这些不足，继续迭代开发两个版本：版本2和版本3，分别对应第39节和第40节的内容。

在版本2中，我们会利用之前学过的重构方法，对版本1的设计与实现进行重构，解决版本1存在的设计问题，让它满足之前学过的设计原则、思想、编程规范。在版本3中，我们再对版本2进行迭代，并且完善框架的功能和非功能需求，让其满足第25节课中罗列的所有需求。

话不多说，让我们正式开始版本2的设计与实现吧！

回顾版本1的设计与实现

首先，让我们一块回顾一下版本1的设计与实现。当然，如果时间充足，你最好能再重新看一下第25、26节的内容。在版本1中，整个框架的代码被划分为下面这几个类。

- MetricsCollector：负责打点采集原始数据，包括记录每次接口请求的响应时间和请求时间戳，并调用MetricsStorage提供的接口来存储这些原始数据。
- MetricsStorage和RedisMetricsStorage：负责原始数据的存储和读取。
- Aggregator：是一个工具类，负责各种统计数据的计算，比如响应时间的最大值、最小值、平均值、百分位值、接口访问次数、tps。
- ConsoleReporter和EmailReporter：相当于一个上帝类（God Class），定时根据给定的时间区间，从数据库中取出数据，借助Aggregator类完成统计工作，并将统计结果输出到相应的终端，比如命令行、邮件。

MetricCollector、MetricsStorage、RedisMetricsStorage的设计与实现比较简单，不是版本2重构的重点。今天，我们重点来看一下Aggregator和ConsoleReporter、EmailReporter这几个类。

我们先来看一下Aggregator类存在的问题。

Aggregator类里面只有一个静态函数，有50行左右的代码量，负责各种统计数据的计算。当要添加新的统计功能的时候，我们需要修改aggregate()函数代码。一旦越来越多的统计功能添加进来之后，这个函数的代码量会持续增加，可读性、可维护性就变差了。因此，我们需要在版本2中对其进行重构。

```
public class Aggregator {
  public static RequestStat aggregate(List<RequestInfo> requestInfos, long durationInMillis) {
    double maxRespTime = Double.MIN_VALUE;
    double minRespTime = Double.MAX_VALUE;
    double avgRespTime = -1;
    double p999RespTime = -1;
    double p99RespTime = -1;
    double sumRespTime = 0;
```

www.ixuexi.cf 分享站 用户标记 zyceson

```
    long count = 0;
    for (RequestInfo requestInfo : requestInfos) {
      ++count;
      double respTime = requestInfo.getResponseTime();
      if (maxRespTime < respTime) {
        maxRespTime = respTime;
      }
      if (minRespTime > respTime) {
        minRespTime = respTime;
      }
      sumRespTime += respTime;
    }
    if (count != 0) {
      avgRespTime = sumRespTime / count;
    }
    long tps = (long)(count / durationInMillis * 1000);
    Collections.sort(requestInfos, new Comparator<RequestInfo>() {
      @Override
      public int compare(RequestInfo o1, RequestInfo o2) {
        double diff = o1.getResponseTime() - o2.getResponseTime();
        if (diff < 0.0) {
          return -1;
        } else if (diff > 0.0) {
          return 1;
        } else {
          return 0;
        }
      }
    });

    if (count != 0) {
      int idx999 = (int)(count * 0.999);
      int idx99 = (int)(count * 0.99);
      p999RespTime = requestInfos.get(idx999).getResponseTime();
      p99RespTime = requestInfos.get(idx99).getResponseTime();
    }
    RequestStat requestStat = new RequestStat();
    requestStat.setMaxResponseTime(maxRespTime);
    requestStat.setMinResponseTime(minRespTime);
    requestStat.setAvgResponseTime(avgRespTime);
    requestStat.setP999ResponseTime(p999RespTime);
    requestStat.setP99ResponseTime(p99RespTime);
    requestStat.setCount(count);
    requestStat.setTps(tps);
    return requestStat;
  }
}

public class RequestStat {
  private double maxResponseTime;
  private double minResponseTime;
  private double avgResponseTime;
  private double p999ResponseTime;
  private double p99ResponseTime;
  private long count;
  private long tps;
  //...省略getter/setter方法...
}
```

我们再来看一下ConsoleReporter和EmailReporter这两个类存在的问题。

ConsoleReporter和EmailReporter两个类中存在代码重复问题。在这两个类中，从数据库中取数据、做统计的逻辑都是相同的，可以抽取出来复用，否则就违反了DRY原则。

整个类负责的事情比较多，不相干的逻辑糅合在里面，职责不够单一。特别是显示部分的代码可能会比较复杂（比如Email的显示方式），最好能将这部分显示逻辑剥离出来，设计成一个独立的类。

除此之外，因为代码中涉及线程操作，并且调用了Aggregator的静态函数，所以代码的可测试性也有待提高。

```
public class ConsoleReporter {
  private MetricsStorage metricsStorage;
  private ScheduledExecutorService executor;

  public ConsoleReporter(MetricsStorage metricsStorage) {
    this.metricsStorage = metricsStorage;
    this.executor = Executors.newSingleThreadScheduledExecutor();
  }

  public void startRepeatedReport(long periodInSeconds, long durationInSeconds) {
    executor.scheduleAtFixedRate(new Runnable() {
      @Override
      public void run() {
        long durationInMillis = durationInSeconds * 1000;
        long endTimeInMillis = System.currentTimeMillis();
        long startTimeInMillis = endTimeInMillis - durationInMillis;
        Map<String, List<RequestInfo>> requestInfos =
                metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
        Map<String, RequestStat> stats = new HashMap<>();
        for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet()) {
          String apiName = entry.getKey();
          List<RequestInfo> requestInfosPerApi = entry.getValue();
          RequestStat requestStat = Aggregator.aggregate(requestInfosPerApi, durationInMillis);
          stats.put(apiName, requestStat);
        }
        System.out.println("Time Span: [" + startTimeInMillis + ", " + endTimeInMillis + "]");
        Gson gson = new Gson();
        System.out.println(gson.toJson(stats));
      }
    }, 0, periodInSeconds, TimeUnit.SECONDS);
  }

}

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

  private MetricsStorage metricsStorage;
  private EmailSender emailSender;
  private List<String> toAddresses = new ArrayList<>();

  public EmailReporter(MetricsStorage metricsStorage) {
    this(metricsStorage, new EmailSender(/*省略参数*/));
  }

  public EmailReporter(MetricsStorage metricsStorage, EmailSender emailSender) {
    this.metricsStorage = metricsStorage;
    this.emailSender = emailSender;
  }
```

www.ixuexi.cf 分享站 用户标记 zyceson

```
  public void addToAddress(String address) {
    toAddresses.add(address);
  }

  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;
        Map<String, List<RequestInfo>> requestInfos =
                metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
        Map<String, RequestStat> stats = new HashMap<>();
        for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet()) {
          String apiName = entry.getKey();
          List<RequestInfo> requestInfosPerApi = entry.getValue();
          RequestStat requestStat = Aggregator.aggregate(requestInfosPerApi, durationInMillis);
          stats.put(apiName, requestStat);
        }
        // TODO: 格式化为html格式，并且发送邮件
      }
    }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
  }

}
```

针对版本1的问题进行重构

Aggregator类和ConsoleReporter、EmailReporter类主要负责统计显示的工作。在第26节中，我们提到，如果我们把统计显示所要完成的功能逻辑细分一下，主要包含下面4点：

1. 根据给定的时间区间，从数据库中拉取数据；
2. 根据原始数据，计算得到统计数据；
3. 将统计数据显示到终端（命令行或邮件）；
4. 定时触发以上三个过程的执行。

之前的划分方法是将所有的逻辑都放到ConsoleReporter和EmailReporter这两个上帝类中，而Aggregator只是一个包含静态方法的工具类。这样的划分方法存在前面提到的一些问题，我们需要对其进行重新划分。

面向对象设计中的最后一步是组装类并提供执行入口，所以，组装前三部分逻辑的上帝类是必须要有的。我们可以将上帝类做的很轻量级，把核心逻辑都剥离出去，形成独立的类，上帝类只负责组装类和串联执行流程。这样做的好处是，代码结构更加清晰，底层核心逻辑更容易被复用。按照这个设计思路，具体的重构工作包含以下4个方面。

- 第1个逻辑：根据给定时间区间，从数据库中拉取数据。这部分逻辑已经被封装在MetricsStorage类中了，所以这部分不需要处理。

- 第2个逻辑：根据原始数据，计算得到统计数据。我们可以将这部分逻辑移动到Aggregator类中。这样Aggregator类就不仅仅是只包含统计方法的工具类了。按照这个思路，重构之后的代码如下所示：

```
public class Aggregator {
  public Map<String, RequestStat> aggregate(
          Map<String, List<RequestInfo>> requestInfos, long durationInMillis) {
    Map<String, RequestStat> requestStats = new HashMap<>();
    for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet()) {
      String apiName = entry.getKey();
      List<RequestInfo> requestInfosPerApi = entry.getValue();
      RequestStat requestStat = doAggregate(requestInfosPerApi, durationInMillis);
      requestStats.put(apiName, requestStat);
    }
    return requestStats;
  }

  private RequestStat doAggregate(List<RequestInfo> requestInfos, long durationInMillis) {
    List<Double> respTimes = new ArrayList<>();
    for (RequestInfo requestInfo : requestInfos) {
      double respTime = requestInfo.getResponseTime();
      respTimes.add(respTime);
    }

    RequestStat requestStat = new RequestStat();
    requestStat.setMaxResponseTime(max(respTimes));
    requestStat.setMinResponseTime(min(respTimes));
    requestStat.setAvgResponseTime(avg(respTimes));
    requestStat.setP999ResponseTime(percentile999(respTimes));
    requestStat.setP99ResponseTime(percentile99(respTimes));
    requestStat.setCount(respTimes.size());
    requestStat.setTps((long) tps(respTimes.size(), durationInMillis/1000));
    return requestStat;
  }

  // 以下的函数的代码实现均省略...
  private double max(List<Double> dataset) {}
  private double min(List<Double> dataset) {}
  private double avg(List<Double> dataset) {}
  private double tps(int count, double duration) {}
  private double percentile999(List<Double> dataset) {}
  private double percentile99(List<Double> dataset) {}
  private double percentile(List<Double> dataset, double ratio) {}
}
```

- 第3个逻辑：将统计数据显示到终端。我们将这部分逻辑剥离出来，设计成两个类：ConsoleViewer类和EmailViewer类，分别负责将统计结果显示到命令行和邮件中。具体的代码实现如下所示：

```
public interface StatViewer {
  void output(Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMills);
}

public class ConsoleViewer implements StatViewer {
  public void output(
          Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMills) {
    System.out.println("Time Span: [" + startTimeInMillis + ", " + endTimeInMills + "]");
    Gson gson = new Gson();
    System.out.println(gson.toJson(requestStats));
```

www.ixuexi.cf 分享站 用户标记 zyceson

```
  }
}

public class EmailViewer implements StatViewer {
  private EmailSender emailSender;
  private List<String> toAddresses = new ArrayList<>();

  public EmailViewer() {
    this.emailSender = new EmailSender(/*省略参数*/);
  }

  public EmailViewer(EmailSender emailSender) {
    this.emailSender = emailSender;
  }

  public void addToAddress(String address) {
    toAddresses.add(address);
  }

  public void output(
          Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMills) {
    // format the requestStats to HTML style.
    // send it to email toAddresses.
  }
}
```

- 第4个逻辑：组装类并定时触发执行统计显示。在将核心逻辑剥离出来之后，这个类的代码变得更加简洁、清晰，只负责组装各个类（MetricsStorage、Aggegrator、StatViewer）来完成整个工作流程。重构之后的代码如下所示：

```
public class ConsoleReporter {
  private MetricsStorage metricsStorage;
  private Aggregator aggregator;
  private StatViewer viewer;
  private ScheduledExecutorService executor;

  public ConsoleReporter(MetricsStorage metricsStorage, Aggregator aggregator, StatViewer viewer) {
    this.metricsStorage = metricsStorage;
    this.aggregator = aggregator;
    this.viewer = viewer;
    this.executor = Executors.newSingleThreadScheduledExecutor();
  }

  public void startRepeatedReport(long periodInSeconds, long durationInSeconds) {
    executor.scheduleAtFixedRate(new Runnable() {
      @Override
      public void run() {
        long durationInMillis = durationInSeconds * 1000;
        long endTimeInMillis = System.currentTimeMillis();
        long startTimeInMillis = endTimeInMillis - durationInMillis;
        Map<String, List<RequestInfo>> requestInfos =
                metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
        Map<String, RequestStat> requestStats = aggregator.aggregate(requestInfos, durationInMillis);
        viewer.output(requestStats, startTimeInMillis, endTimeInMillis);
      }
    }, 0L, periodInSeconds, TimeUnit.SECONDS);
  }

}
```

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

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

  public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, StatViewer viewer) {
    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;
        Map<String, List<RequestInfo>> requestInfos =
                metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
        Map<String, RequestStat> stats = aggregator.aggregate(requestInfos, durationInMillis);
        viewer.output(stats, startTimeInMillis, endTimeInMillis);
      }
    }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
  }
}
```

经过上面的重构之后，我们现在再来看一下，现在框架该如何来使用。

我们需要在应用启动的时候，创建好ConsoleReporter对象，并且调用它的startRepeatedReport()函数，来启动定时统计并输出数据到终端。同理，我们还需要创建好EmailReporter对象，并且调用它的startDailyReport()函数，来启动每日统计并输出数据到制定邮件地址。我们通过MetricsCollector类来收集接口的访问情况，这部分收集代码会跟业务逻辑代码耦合在一起，或者统一放到类似Spring AOP的切面中完成。具体的使用代码示例如下：

```
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, consoleViewer);
    consoleReporter.startRepeatedReport(60, 60);

    // 定时触发统计并将结果输出到邮件
```

www.ixuexi.cf 分享站 用户标记 zyceson

```
    EmailViewer emailViewer = new EmailViewer();
    emailViewer.addToAddress("wangzheng@xzg.com");
    EmailReporter emailReporter = new EmailReporter(storage, aggregator, emailViewer);
    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();
    }
  }
}
```

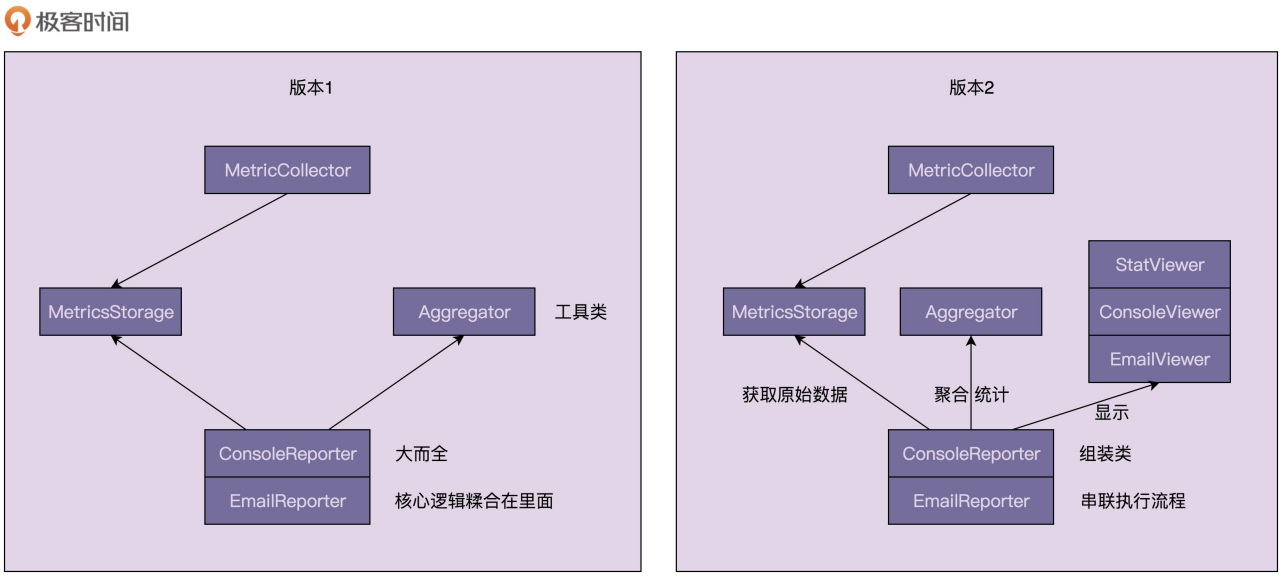
Review版本2的设计与实现

现在，我们Review一下，针对版本1重构之后，版本2的设计与实现。

重构之后，MetricsStorage负责存储，Aggregator负责统计，StatViewer（ConsoleViewer、EmailViewer）负责显示，三个类各司其职。ConsoleReporter和EmailReporter负责组装这三个类，将获取原始数据、聚合统计、显示统计结果到终端这三个阶段的工作串联起来，定时触发执行。

除此之外，MetricsStorage、Aggregator、StatViewer三个类的设计也符合迪米特法则。它们只与跟自己有直接相关的数据进行交互。MetricsStorage输出的是RequestInfo相关数据。Aggregator类输入的是RequestInfo数据，输出的是RequestStat数据。StatViewer输入的是RequestStat数据。

针对版本1和版本2，我画了一张它们的类之间依赖关系的对比图，如下所示。从图中，我们可以看出，重构之后的代码结构更加清晰、有条理。这也印证了之前提到的：面向对象设计和实现要做的事情，就是把合适的代码放到合适的类中。



刚刚我们分析了代码的整体结构和依赖关系，我们现在再来具体看每个类的设计。

Aggregator类从一个只包含一个静态函数的工具类，变成了一个普通的聚合统计类。现在，我们可以通过依赖注入的方式，将其组装进ConsoleReporter和EmailReporter类中，这样就更加容易编写单元测试。

Aggregator类在重构前，所有的逻辑都集中在aggregate()函数内，代码行数较多，代码的可读性和可维护性较差。在重构之后，我们将每个统计逻辑拆分成独立的函数，aggregate()函数变得比较单薄，可读性提高了。尽管我们要添加新的统计功能，还是要修改aggregate()函数，但现在的aggregate()函数代码行数很少，结构非常清晰，修改起来更加容易，可维护性提高。

目前来看，Aggregator的设计还算合理。但是，如果随着更多的统计功能的加入，Aggregator类的代码会越来越多。这个时候，我们可以将统计函数剥离出来，设计成独立的类，以解决Aggregator类的无限膨胀问题。不过，暂时来说没有必要这么做，毕竟将每个统计函数独立成类，会增加类的个数，也会影响到代码的可读性和可维护性。

ConsoleReporter和EmailReporter经过重构之后，代码的重复问题变小了，但仍然没有完全解决。尽管这两个类不再调用Aggregator的静态方法，但因为涉及多线程和时间相关的计算，代码的测试性仍然不够好。这两个问题我们留在下一节课中解决，你也可以留言说说的你解决方案。

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。

面向对象设计中的最后一步是组装类并提供执行入口，也就是上帝类要做的事情。这个上帝类是没办法去掉的，但我们可以将上帝类做得很轻量级，把核心逻辑都剥离出去，下沉形成独立的类。上帝类只负责组装类和串联执行流程。这样做的好处是，代码结构更加清晰，底层核心逻辑更容易被复用。

面向对象设计和实现要做的事情，就是把合适的代码放到合适的类中。当我们要实现某个功能的时候，不管如何设计，所需要编写的代码量基本上是一样的，唯一的区别就是如何将这些代码划分到不同的类中。不同的人有不同的划分方法，对应得到的代码结构（比如类与类之间交互等）也不尽相同。

好的设计一定是结构清晰、有条理、逻辑性强，看起来一目了然，读完之后常常有一种原来如此的感觉。差的设计往往逻辑、代码乱塞一通，没有什么设计思路可言，看起来莫名其妙，读完之后一头雾水。

课堂讨论

1. 今天我们提到，重构之后的ConsoleReporter和EmailReporter仍然存在代码重复和可测试性差的问题，你可以思考一下，应该如何解决呢？
2. 从上面的使用示例中，我们可以看出，框架易用性有待提高：ConsoleReporter和EmailReporter的创建过程比较复杂，使用者需要正确地组装各种类才行。对于框架的易用性，你有没有什么办法改善一下呢？

欢迎在留言区写下你的思考和想法，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。

精选留言：

- 小晏子 2020-01-31 10:01:20

课后思考：
1. 将两个reporter中的run里的逻辑单独提取出来做成一个公共函数void doReport(duration, endTime, startTime)，这个函数易于单独测试，两个reporter类中调用doReport，因为两个reporter类中并无特殊的逻辑处理，只使用了jdk本身提供的功能，我们可以相信jdk本身的正确性，所以这块就可以不写单元测试了，这就简化了测试也解决了重复代码的问题。
[1赞]

- 守拙 2020-01-31 15:36:37
课堂讨论

1. 今天我们提到，重构之后的 ConsoleReporter 和 EmailReporter 仍然存在代码重复和可测试性差的问题，你可以思考一下，应该如何解决呢？

ConsoleReporter和EmailReporter的代码重复集中在viewer#output()部分.可以抽象一个AbsReporter,将重复代码放在基类中,并让ConsoleReporter和EmailReporter继承自AbsReporter. 这里基类与衍生类完全符合is-a关系, 但并未使用多态性.

2. 从上面的使用示例中，我们可以看出，框架易用性有待提高：ConsoleReporter 和 EmailReporter 的创建过程比较复杂，使用者需要正确地组装各种类才行。对于框架的易用性，你有没有什么办法改善一下呢？

可以使用builder模式改造, 提供更友好的依赖注入方式. 除此以外, 还应编写良好的注释, 帮助客户端程序员正确的使用框架.

示例:

```
ConsoleReporter instance = ConsoleReporter.Builder()

.setMetricsStorate(storage)

.setAggregator(aggregator)

.setStatViewer(viewer)

.setExecutor(executor)

.build();
```

- javaadu 2020-01-31 12:04:33
2. 如果使用Spring Boot之类的框架，就可以利用框架做自动注入；如果没有，则可以用工厂方法设计模式来拼比掉复杂的对象创建过程

- javaadu 2020-01-31 12:03:37
1. 看了下，ConoleReporter和EmailReporter的核心区别在于使用的显示器不同，另外就是调度的频次不同，第二个不同是可以通用化的，可以提取出一个抽象的调度器（把查询数据、调用聚合统计对象的代码都放进去），支持每秒、分、时、天调度；ConsoleReportor和EmailReporter都使用这个调度器，自己只维护对应的显示器对象的引用就可以了。

- liu_liu 2020-01-31 10:13:07
1. 可定义父类，重复代码抽取为函数进行复用

2. 用工厂方法，屏蔽创建过程

- 高源 2020-01-31 07:18:39
  仔细学习分析一下重构后带来的好处，解决了哪些问题

- ちよくん 2020-01-31 01:23:10
  打卡