

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）

2020-01-31 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 12:29 大小 10.01M



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，在设计上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。

 复制代码

```
1 public class Aggregator {
2     public static RequestStat aggregate(List<RequestInfo> requestInfos, long dur:
3         double maxRespTime = Double.MIN_VALUE;
4         double minRespTime = Double.MAX_VALUE;
```

```

5     double avgRespTime = -1;
6     double p999RespTime = -1;
7     double p99RespTime = -1;
8     double sumRespTime = 0;
9     long count = 0;
10    for (RequestInfo requestInfo : requestInfos) {
11        ++count;
12        double respTime = requestInfo.getResponseTime();
13        if (maxRespTime < respTime) {
14            maxRespTime = respTime;
15        }
16        if (minRespTime > respTime) {
17            minRespTime = respTime;
18        }
19        sumRespTime += respTime;
20    }
21    if (count != 0) {
22        avgRespTime = sumRespTime / count;
23    }
24    long tps = (long)(count / durationInMillis * 1000);
25    Collections.sort(requestInfos, new Comparator<RequestInfo>() {
26        @Override
27        public int compare(RequestInfo o1, RequestInfo o2) {
28            double diff = o1.getResponseTime() - o2.getResponseTime();
29            if (diff < 0.0) {
30                return -1;
31            } else if (diff > 0.0) {
32                return 1;
33            } else {
34                return 0;
35            }
36        }
37    });
38
39    if (count != 0) {
40        int idx999 = (int)(count * 0.999);
41        int idx99 = (int)(count * 0.99);
42        p999RespTime = requestInfos.get(idx999).getResponseTime();
43        p99RespTime = requestInfos.get(idx99).getResponseTime();
44    }
45    RequestStat requestStat = new RequestStat();
46    requestStat.setMaxResponseTime(maxRespTime);
47    requestStat.setMinResponseTime(minRespTime);
48    requestStat.setAvgResponseTime(avgRespTime);
49    requestStat.setP999ResponseTime(p999RespTime);
50    requestStat.setP99ResponseTime(p99RespTime);
51    requestStat.setCount(count);
52    requestStat.setTps(tps);
53    return requestStat;
54 }
55 }
56

```

```
57 public class RequestStat {
58     private double maxResponseTime;
59     private double minResponseTime;
60     private double avgResponseTime;
61     private double p999ResponseTime;
62     private double p99ResponseTime;
63     private long count;
64     private long tps;
65     //...省略getter/setter方法...
66 }
```

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。

 复制代码

```
1 public class ConsoleReporter {
2     private MetricsStorage metricsStorage;
3     private ScheduledExecutorService executor;
4
5     public ConsoleReporter(MetricsStorage metricsStorage) {
6         this.metricsStorage = metricsStorage;
7         this.executor = Executors.newSingleThreadScheduledExecutor();
8     }
9
10    public void startRepeatedReport(long periodInSeconds, long durationInSeconds) {
11        executor.scheduleAtFixedRate(new Runnable() {
12            @Override
13            public void run() {
14                long durationInMillis = durationInSeconds * 1000;
15                long endTimeInMillis = System.currentTimeMillis();
16                long startTimeInMillis = endTimeInMillis - durationInMillis;
17                Map<String, List<RequestInfo>> requestInfos =
18                    metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
19                Map<String, RequestStat> stats = new HashMap<>();
```



```

20         for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet()) {
21             String apiName = entry.getKey();
22             List<RequestInfo> requestInfosPerApi = entry.getValue();
23             RequestStat requestStat = Aggregator.aggregate(requestInfosPerApi, di);
24             stats.put(apiName, requestStat);
25         }
26         System.out.println("Time Span: [" + startTimeInMillis + ", " + endTimeInMillis + "]");
27         Gson gson = new Gson();
28         System.out.println(gson.toJson(stats));
29     }
30 }, 0, periodInSeconds, TimeUnit.SECONDS);
31 }
32
33 }
34
35 public class EmailReporter {
36     private static final Long DAY_HOURS_IN_SECONDS = 86400L;
37
38     private MetricsStorage metricsStorage;
39     private EmailSender emailSender;
40     private List<String> toAddresses = new ArrayList<>();
41
42     public EmailReporter(MetricsStorage metricsStorage) {
43         this(metricsStorage, new EmailSender(/*省略参数*/));
44     }
45
46     public EmailReporter(MetricsStorage metricsStorage, EmailSender emailSender) {
47         this.metricsStorage = metricsStorage;
48         this.emailSender = emailSender;
49     }
50
51     public void addToAddress(String address) {
52         toAddresses.add(address);
53     }
54
55     public void startDailyReport() {
56         Calendar calendar = Calendar.getInstance();
57         calendar.add(Calendar.DATE, 1);
58         calendar.set(Calendar.HOUR_OF_DAY, 0);
59         calendar.set(Calendar.MINUTE, 0);
60         calendar.set(Calendar.SECOND, 0);
61         calendar.set(Calendar.MILLISECOND, 0);
62         Date firstTime = calendar.getTime();
63         Timer timer = new Timer();
64         timer.schedule(new TimerTask() {
65             @Override
66             public void run() {
67                 long durationInMillis = DAY_HOURS_IN_SECONDS * 1000;
68                 long endTimeInMillis = System.currentTimeMillis();
69                 long startTimeInMillis = endTimeInMillis - durationInMillis;
70                 Map<String, List<RequestInfo>> requestInfos =
71                     metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);

```

```

72         Map<String, RequestStat> stats = new HashMap<>();
73         for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet())
74             String apiName = entry.getKey();
75             List<RequestInfo> requestInfosPerApi = entry.getValue();
76             RequestStat requestStat = Aggregator.aggregate(requestInfosPerApi, di);
77             stats.put(apiName, requestStat);
78         }
79         // TODO: 格式化为html格式, 并且发送邮件
80     }
81 }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
82 }
83
84 }

```

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。

之前的划分方法是把所有的逻辑都放到 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：

 复制代码

```
1 public class Aggregator {
2     public Map<String, RequestStat> aggregate(
3         Map<String, List<RequestInfo>> requestInfos, long durationInMillis) {
4         Map<String, RequestStat> requestStats = new HashMap<>();
5         for (Map.Entry<String, List<RequestInfo>> entry : requestInfos.entrySet())
6             String apiName = entry.getKey();
7             List<RequestInfo> requestInfosPerApi = entry.getValue();
8             RequestStat requestStat = doAggregate(requestInfosPerApi, durationInMillis);
9             requestStats.put(apiName, requestStat);
10    }
11    return requestStats;
12 }
13
14 private RequestStat doAggregate(List<RequestInfo> requestInfos, long durationInMillis) {
15     List<Double> respTimes = new ArrayList<>();
16     for (RequestInfo requestInfo : requestInfos) {
17         double respTime = requestInfo.getResponseTime();
18         respTimes.add(respTime);
19     }
20
21     RequestStat requestStat = new RequestStat();
22     requestStat.setMaxResponseTime(max(respTimes));
23     requestStat.setMinResponseTime(min(respTimes));
24     requestStat.setAvgResponseTime(avg(respTimes));
25     requestStat.setP999ResponseTime(percentile999(respTimes));
26     requestStat.setP99ResponseTime(percentile99(respTimes));
27     requestStat.setCount(respTimes.size());
28     requestStat.setTps((long) tps(respTimes.size(), durationInMillis/1000));
29     return requestStat;
30 }
31
32 // 以下的函数的代码实现均省略...
33 private double max(List<Double> dataset) {}
34 private double min(List<Double> dataset) {}
35 private double avg(List<Double> dataset) {}
36 private double tps(int count, double duration) {}
37 private double percentile999(List<Double> dataset) {}
38 private double percentile99(List<Double> dataset) {}
39 private double percentile(List<Double> dataset, double ratio) {}
40 }
```

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：

 复制代码

```
1 public interface StatViewer {
2     void output(Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMillis);
3 }
4
5 public class ConsoleViewer implements StatViewer {
6     public void output(
7         Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMillis) {
8         System.out.println("Time Span: [" + startTimeInMillis + ", " + endTimeInMillis + "]");
9         Gson gson = new Gson();
10        System.out.println(gson.toJson(requestStats));
11    }
12 }
13
14 public class EmailViewer implements StatViewer {
15     private EmailSender emailSender;
16     private List<String> toAddresses = new ArrayList<>();
17
18     public EmailViewer() {
19         this.emailSender = new EmailSender(/*省略参数*/);
20     }
21
22     public EmailViewer(EmailSender emailSender) {
23         this.emailSender = emailSender;
24     }
25
26     public void addToAddress(String address) {
27         toAddresses.add(address);
28     }
29
30     public void output(
31         Map<String, RequestStat> requestStats, long startTimeInMillis, long endTimeInMillis) {
32         // format the requestStats to HTML style.
33         // send it to email toAddresses.
34     }
35 }
```

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、Aggregator、StatViewer）来完成整个工作流程。重构之后的代码如下所示：


```
1 public class ConsoleReporter {
2     private MetricsStorage metricsStorage;
3     private Aggregator aggregator;
4     private StatViewer viewer;
5     private ScheduledExecutorService executor;
6
7     public ConsoleReporter(MetricsStorage metricsStorage, Aggregator aggregator,
8         this.metricsStorage = metricsStorage;
9         this.aggregator = aggregator;
10        this.viewer = viewer;
11        this.executor = Executors.newSingleThreadScheduledExecutor();
12    }
13
14    public void startRepeatedReport(long periodInSeconds, long durationInSeconds) {
15        executor.scheduleAtFixedRate(new Runnable() {
16            @Override
17            public void run() {
18                long durationInMillis = durationInSeconds * 1000;
19                long endTimeInMillis = System.currentTimeMillis();
20                long startTimeInMillis = endTimeInMillis - durationInMillis;
21                Map<String, List<RequestInfo>> requestInfos =
22                    metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMil
23                Map<String, RequestStat> requestStats = aggregator.aggregate(requestIn
24                viewer.output(requestStats, startTimeInMillis, endTimeInMillis);
25            }
26        }, 0L, periodInSeconds, TimeUnit.SECONDS);
27    }
28
29 }
30
31 public class EmailReporter {
32     private static final Long DAY_HOURS_IN_SECONDS = 86400L;
33
34     private MetricsStorage metricsStorage;
35     private Aggregator aggregator;
36     private StatViewer viewer;
37
38     public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, S
39         this.metricsStorage = metricsStorage;
40         this.aggregator = aggregator;
41         this.viewer = viewer;
42     }
43
44     public void startDailyReport() {
45         Calendar calendar = Calendar.getInstance();
46         calendar.add(Calendar.DATE, 1);
47         calendar.set(Calendar.HOUR_OF_DAY, 0);
48         calendar.set(Calendar.MINUTE, 0);
49         calendar.set(Calendar.SECOND, 0);
50         calendar.set(Calendar.MILLISECOND, 0);
51         Date firstTime = calendar.getTime();
```

```

52     Timer timer = new Timer();
53     timer.schedule(new TimerTask() {
54         @Override
55         public void run() {
56             long durationInMillis = DAY_HOURS_IN_SECONDS * 1000;
57             long endTimeInMillis = System.currentTimeMillis();
58             long startTimeInMillis = endTimeInMillis - durationInMillis;
59             Map<String, List<RequestInfo>> requestInfos =
60                 metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
61             Map<String, RequestStat> stats = aggregator.aggregate(requestInfos, durationInMillis);
62             viewer.output(stats, startTimeInMillis, endTimeInMillis);
63         }
64     }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
65 }
66 }

```

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：

 复制代码

```

1  public class PerfCounterTest {
2      public static void main(String[] args) {
3          MetricsStorage storage = new RedisMetricsStorage();
4          Aggregator aggregator = new Aggregator();
5
6          // 定时触发统计并将结果显示到终端
7          ConsoleViewer consoleViewer = new ConsoleViewer();
8          ConsoleReporter consoleReporter = new ConsoleReporter(storage, aggregator, consoleViewer);
9          consoleReporter.startRepeatedReport(60, 60);
10
11         // 定时触发统计并将结果输出到邮件
12         EmailViewer emailViewer = new EmailViewer();
13         emailViewer.addToAddress("wangzheng@xzg.com");
14         EmailReporter emailReporter = new EmailReporter(storage, aggregator, emailViewer);
15         emailReporter.startDailyReport();
16
17         // 收集接口访问数据
18         MetricsCollector collector = new MetricsCollector(storage);
19         collector.recordRequest(new RequestInfo("register", 123, 10234));

```

```
20 collector.recordRequest(new RequestInfo("register", 223, 11234));
21 collector.recordRequest(new RequestInfo("register", 323, 12334));
22 collector.recordRequest(new RequestInfo("login", 23, 12434));
23 collector.recordRequest(new RequestInfo("login", 1223, 14234));
24
25 try {
26     Thread.sleep(100000);
27 } catch (InterruptedException e) {
28     e.printStackTrace();
29 }
30 }
31 }
```

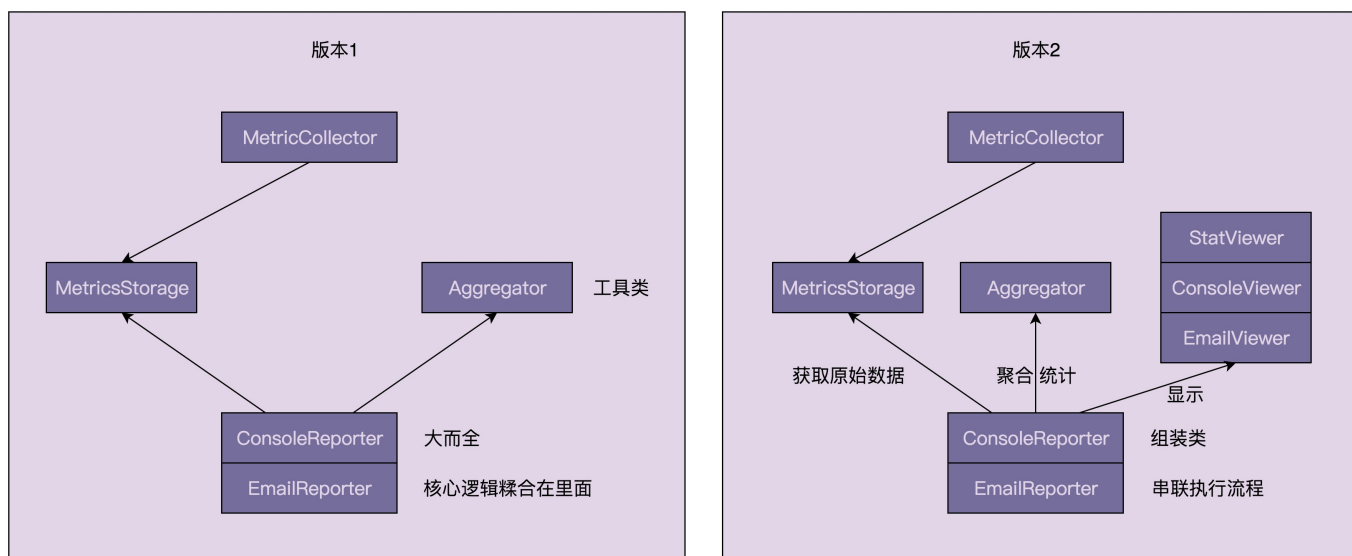
Review 版本 2 的设计与实现

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。

点击参加小程序学习打卡 

8个月，攻克设计模式



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 38 | 总结回顾面向对象、设计原则、编程规范、重构技巧等知识点

下一篇 加餐二 | 设计模式、重构、编程规范等相关书籍推荐

精选留言 (10)

 写留言



小晏子

2020-01-31

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...

展开 ∨



2



平风造雨

2020-02-01

1. Reporter中线程调用的run方法可以单独提取一个方法不依赖额外的线程去调用，方便单元测试。

2. 另外Reporter中的线程模型是否可以单独提取出一个类，该类负责按需创建线程，并且调用实际的埋点统计方法。
3. 可以借助框架层面依赖注入的方式，更为简单的构造Reporter类。

展开 ∨



1



辣么大

2020-02-01

问题1， reporter可测性差的问题，可以mock storage，将request信息到map中。

```
// mock
```

```
MetricsStorage storage = new MockRedisMetricsStorage();
```

问题2， reporter的创建过程可以使用简单工厂方法。Aggregator完全没有必要暴露出...

展开 ∨



1



javaadu

2020-01-31

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

展开 ∨



1



javaadu

2020-01-31

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； ConsoleReporter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。

展开 ∨



1



Jxin

2020-01-31

1.将定时和输出报表这两件事分离。单独的定时线程，在关键的时间点都触发一个事件。输出报表的两个类去监听自己关心的时间job的事件（生产消费模式）。如此一来，定时触发好不好使不再是我api使用方考虑的事。我只需要测试对应输出报表的业务是否正常。然后就控制台和邮件这两个报表类，其生成报表的逻辑是一样的，仅仅是展示的“方式”不一样。所以让我选，我会合并这两个类，生成报表的逻辑为私有方法，然后单独写一个...

展开 ∨



守拙

2020-01-31

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？

...

展开 ▾



liu_liu

2020-01-31

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

仔细学习分析一下重构后带来的好处，解决了哪些问题

展开 ▾



ちよくん

2020-01-31

打卡

展开 ▾

