

## 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）

王争 · 设计模式之美



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

```
1 public abstract class ScheduledReporter {
2     protected MetricsStorage metricsStorage;
3     protected Aggregator aggregator;
4     protected StatViewer viewer;
5
6     public ScheduledReporter(MetricsStorage metricsStorage, Aggregator aggregator,
7         this.metricsStorage = metricsStorage;
8         this.aggregator = aggregator;
9         this.viewer = viewer;
10 }
11
12 protected void doStatAndReport(long startTimeInMillis, long endTimeInMillis) {
13     long durationInMillis = endTimeInMillis - startTimeInMillis;
14     Map<String, List<RequestInfo>> requestInfos =
15         metricsStorage.getRequestInfos(startTimeInMillis, endTimeInMillis);
16     Map<String, RequestStat> requestStats = aggregator.aggregate(requestInfos, du
17     viewer.output(requestStats, startTimeInMillis, endTimeInMillis);
18 }
19
20 }
```

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

```
1 public class EmailReporter extends ScheduledReporter {
2     private static final Long DAY_HOURS_IN_SECONDS = 86400L;
3
4     private MetricsStorage metricsStorage;
5     private Aggregator aggregator;
6     private StatViewer viewer;
7
8     public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, Stat
9         this.metricsStorage = metricsStorage;
10        this.aggregator = aggregator;
11        this.viewer = viewer;
12 }
```


```

13
14 public void startDailyReport() {
15     Calendar calendar = Calendar.getInstance();
16     calendar.add(Calendar.DATE, 1);
17     calendar.set(Calendar.HOUR_OF_DAY, 0);
18     calendar.set(Calendar.MINUTE, 0);
19     calendar.set(Calendar.SECOND, 0);
20     calendar.set(Calendar.MILLISECOND, 0);
21     Date firstTime = calendar.getTime();
22
23     Timer timer = new Timer();
24     timer.schedule(new TimerTask() {
25         @Override
26         public void run() {
27             long durationInMillis = DAY_HOURS_IN_SECONDS * 1000;
28             long endTimeInMillis = System.currentTimeMillis();
29             long startTimeInMillis = endTimeInMillis - durationInMillis;
30             doStatAndReport(startTimeInMillis, endTimeInMillis);
31         }
32     }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
33 }
34 }

```

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

```

1 public class EmailReporter extends ScheduledReporter {
2     // 省略其他代码...
3     public void startDailyReport() {
4         Date firstTime = trimTimeFieldsToZeroOfNextDay();
5         Timer timer = new Timer();
6         timer.schedule(new TimerTask() {
7             @Override
8             public void run() {
9                 // 省略其他代码...
10            }


```

```

11     }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
12 }
13
14 // 设置成protected而非private是为了方便写单元测试
15 @VisibleForTesting
16 protected Date trimTimeFieldsToZeroOfNextDay() {
17     Calendar calendar = Calendar.getInstance(); // 这里可以获取当前时间
18     calendar.add(Calendar.DATE, 1);
19     calendar.set(Calendar.HOUR_OF_DAY, 0);
20     calendar.set(Calendar.MINUTE, 0);
21     calendar.set(Calendar.SECOND, 0);
22     calendar.set(Calendar.MILLISECOND, 0);
23     return calendar.getTime();
24 }
25 }

```

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

```

1 public class EmailReporter extends ScheduledReporter {
2     // 省略其他代码...
3     public void startDailyReport() {
4         // new Date()可以获取当前时间
5         Date firstTime = trimTimeFieldsToZeroOfNextDay(new Date());
6         Timer timer = new Timer();
7         timer.schedule(new TimerTask() {
8             @Override
9             public void run() {
10                 // 省略其他代码...
11             }
12         }, firstTime, DAY_HOURS_IN_SECONDS * 1000);
13     }
14
15     protected Date trimTimeFieldsToZeroOfNextDay(Date date) {
16         Calendar calendar = Calendar.getInstance(); // 这里可以获取当前时间
17         calendar.setTime(date); // 重新设置时间
18         calendar.add(Calendar.DATE, 1);
19         calendar.set(Calendar.HOUR_OF_DAY, 0);

```

```
20     calendar.set(Calendar.MINUTE, 0);
21     calendar.set(Calendar.SECOND, 0);
22     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，那我们就不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() 函数里面已经没有什么代码逻辑了，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


```
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, co
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, emailVie
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 }
```

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

```
1 public class MetricsCollector {
2     private MetricsStorage metricsStorage;
3
4     // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
5     public MetricsCollectorB() {
6         this(new RedisMetricsStorage());
7     }
8
9     // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
10    public MetricsCollectorB(MetricsStorage metricsStorage) {
11        this.metricsStorage = metricsStorage;
12    }
13    // 省略其他代码...
14 }
15
16 public class ConsoleReporter extends ScheduledReporter {
17     private ScheduledExecutorService executor;
18
19     // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
20     public ConsoleReporter() {
21         this(new RedisMetricsStorage(), new Aggregator(), new ConsoleViewer());
22     }
23 }
```




```

24 // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
25 public ConsoleReporter(MetricsStorage metricsStorage, Aggregator aggregator, St
26     super(metricsStorage, aggregator, viewer);
27     this.executor = Executors.newSingleThreadScheduledExecutor();
28 }
29 // 省略其他代码...
30 }
31
32 public class EmailReporter extends ScheduledReporter {
33     private static final Long DAY_HOURS_IN_SECONDS = 86400L;
34
35     // 兼顾代码的易用性，新增一个封装了默认依赖的构造函数
36     public EmailReporter(List<String> emailToAddresses) {
37         this(new RedisMetricsStorage(), new Aggregator(), new EmailViewer(emailToAddr
38     }
39
40     // 兼顾灵活性和代码的可测试性，这个构造函数继续保留
41     public EmailReporter(MetricsStorage metricsStorage, Aggregator aggregator, Stat
42         super(metricsStorage, aggregator, viewer);
43     }
44     // 省略其他代码...
45 }

```

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

```

1 public class PerfCounterTest {
2     public static void main(String[] args) {
3         ConsoleReporter consoleReporter = new ConsoleReporter();
4         consoleReporter.startRepeatedReport(60, 60);
5
6         List<String> emailToAddresses = new ArrayList<>();
7         emailToAddresses.add("wangzheng@xzg.com");
8         EmailReporter emailReporter = new EmailReporter(emailToAddresses);
9         emailReporter.startDailyReport();
10
11         MetricsCollector collector = new MetricsCollector();
12         collector.recordRequest(new RequestInfo("register", 123, 10234));
13         collector.recordRequest(new RequestInfo("register", 223, 11234));
14         collector.recordRequest(new RequestInfo("register", 323, 12334));
15         collector.recordRequest(new RequestInfo("login", 23, 12434));
16         collector.recordRequest(new RequestInfo("login", 1223, 14234));
17
18         try {
19             Thread.sleep(100000);

```

```
20     } catch (InterruptedException e) {
21         e.printStackTrace();
22     }
23 }
24 }
```

如果你足够细心，可能已经发现，RedisMetricsStorage 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，有可能会導致内存吃紧，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

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

```

3     private MetricsStorage metricsStorage;
4     private EventBus eventBus;
5
6     public MetricsCollector(MetricsStorage metricsStorage) {
7         this(metricsStorage, DEFAULT_STORAGE_THREAD_POOL_SIZE);
8     }
9
10    public MetricsCollector(MetricsStorage metricsStorage, int threadNumToSaveData)
11        this.metricsStorage = metricsStorage;
12        this.eventBus = new AsyncEventBus(Executors.newFixedThreadPool(threadNumToSav
13        this.eventBus.register(new EventListener());
14    }
15
16    public void recordRequest(RequestInfo requestInfo) {
17        if (requestInfo == null || StringUtils.isBlank(requestInfo.getApiName())) {
18            return;
19        }
20        eventBus.post(requestInfo);
21    }
22
23    public class EventListener {
24        @Subscribe
25        public void saveRequestInfo(RequestInfo requestInfo) {
26            metricsStorage.saveRequestInfo(requestInfo);
27        }
28    }
29 }
30

```

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  $\text{count} \times \text{percentile}$ （count 表示总的个数，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

```
1 public class ScheduleReporter {
2     private static final long MAX_STAT_DURATION_IN_MILLIS = 10 * 60 * 1000; // 10mi
3
4     protected MetricsStorage metricsStorage;
5     protected Aggregator aggregator;
6     protected StatViewer viewer;
7
8     public ScheduleReporter(MetricsStorage metricsStorage, Aggregator aggregator, S
9         this.metricsStorage = metricsStorage;
10        this.aggregator = aggregator;
11        this.viewer = viewer;
12    }
13
14    protected void doStatAndReport(long startTimeInMillis, long endTimeInMillis) {
15        Map<String, RequestStat> stats = doStat(startTimeInMillis, endTimeInMillis);
16        viewer.output(stats, startTimeInMillis, endTimeInMillis);
17    }
18
19    private Map<String, RequestStat> doStat(long startTimeInMillis, long endTimeInM
20        Map<String, List<RequestStat>> segmentStats = new HashMap<>();
21        long segmentStartTimeMillis = startTimeInMillis;
22        while (segmentStartTimeMillis < endTimeInMillis) {
23            long segmentEndTimeMillis = segmentStartTimeMillis + MAX_STAT_DURATION_IN_M
24            if (segmentEndTimeMillis > endTimeInMillis) {
25                segmentEndTimeMillis = endTimeInMillis;
26            }
27            Map<String, List<RequestInfo>> requestInfos =
28                metricsStorage.getRequestInfos(segmentStartTimeMillis, segmentEndTi
29            if (requestInfos == null || requestInfos.isEmpty()) {
30                continue;
31            }
32            Map<String, RequestStat> segmentStat = aggregator.aggregate(
33                requestInfos, segmentEndTimeMillis - segmentStartTimeMillis);
34            addStat(segmentStats, segmentStat);
35            segmentStartTimeMillis += MAX_STAT_DURATION_IN_MILLIS;
36        }
37
38        long durationInMillis = endTimeInMillis - startTimeInMillis;
39        Map<String, RequestStat> aggregatedStats = aggregateStats(segmentStats, durat
40        return aggregatedStats;
41    }
42
43    private void addStat(Map<String, List<RequestStat>> segmentStats,
```

```

44         Map<String, RequestStat> segmentStat) {
45     for (Map.Entry<String, RequestStat> entry : segmentStat.entrySet()) {
46         String apiName = entry.getKey();
47         RequestStat stat = entry.getValue();
48         List<RequestStat> statList = segmentStats.putIfAbsent(apiName, new ArrayList<>());
49         statList.add(stat);
50     }
51 }
52
53 private Map<String, RequestStat> aggregateStats(Map<String, List<RequestStat>>
54                                             long durationInMillis) {
55     Map<String, RequestStat> aggregatedStats = new HashMap<>();
56     for (Map.Entry<String, List<RequestStat>> entry : segmentStats.entrySet()) {
57         String apiName = entry.getKey();
58         List<RequestStat> apiStats = entry.getValue();
59         double maxRespTime = Double.MIN_VALUE;
60         double minRespTime = Double.MAX_VALUE;
61         long count = 0;
62         double sumRespTime = 0;
63         for (RequestStat stat : apiStats) {
64             if (stat.getMaxResponseTime() > maxRespTime) maxRespTime = stat.getMaxResponseTime();
65             if (stat.getMinResponseTime() < minRespTime) minRespTime = stat.getMinResponseTime();
66             count += stat.getCount();
67             sumRespTime += (stat.getCount() * stat.getAvgResponseTime());
68         }
69         RequestStat aggregatedStat = new RequestStat();
70         aggregatedStat.setMaxResponseTime(maxRespTime);
71         aggregatedStat.setMinResponseTime(minRespTime);
72         aggregatedStat.setAvgResponseTime(sumRespTime / count);
73         aggregatedStat.setCount(count);
74         aggregatedStat.setTps(count / durationInMillis * 1000);
75         aggregatedStats.put(apiName, aggregatedStat);
76     }
77     return aggregatedStats;
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。





王争

前 Google 工程师, 《数据结构与算法之美》专栏作者

“设计原则与思想”模块内容已结束, 邀请你填写调查问卷。让我了解你的想法, 更好地改进课程内容!

立即填写



## AI智能总结

本文深入探讨了如何通过设计原则和思想来完善性能计数器项目。首先, 通过代码重构优化, 解决了 ConsoleReporter 和 EmailReporter 中的代码重复问题, 并提高了可测试性。重点讨论了提高 EmailReporter 的可测试性, 通过抽离代码和参数传递来解决时间计算逻辑的问题。文章还详细介绍了功能需求的完善, 包括接口统计信息、统计信息显示格式和终端、统计触发方式、统计时间区间和间隔等。最后, 提出了版本3中还未实现的功能点, 包括被动触发统计的方式、自定义显示终端和显示格式的设计。整体来说, 本文通过具体的代码示例和思路分析, 展示了如何通过设计原则和思想来不断完善性能计数器项目, 为读者提供了一种深入理解和应用设计原则的思路。文章还涉及了性能、扩展性、容错性和通用性等方面的讨论, 为读者提供了全面的技术视角。

© 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。

## 全部留言 (46)

最新 精选



李小四

2020-02-22

设计模式\_40:

# 作业

导致多个线程的重复统计。

办法：加入进程内的全局变量(注意多线程同步问题)。

# 感想

我个人是Android工程师，客户端的开发默认就要思考一个问题：方法重复调用时(如多次点击某个按钮等)逻辑是否还正常。

25, 26, 39, 40这四节课边听边读反复了好几遍了，因为目的是掌握该掌握的东西，而不是简单地打个卡，所以整个春节期间就卡在这四节课的重复中了，不停地循环听。。。

也有好处：我反倒对这几节内容非常熟悉了，有两点感受较深：

1> 方法论：分清楚 \*功能性需求\* 与 \*非功能型需求\*

之前是想到什么注意什么，往往做不到穷举。

2> 一步一步地重构，其实解决的是自信问题：

做事要先解决思想问题，也就是心理问题：

– 没有人能够一步到位地完美解决问题，优秀的代码是演进的，也就是说，代码结构不完美的状态是跳不过去的。

– 我们始终聚焦在解决问题上，代码有问题非常正常。

– 我们要带着成就感不断重构代码，而不是带着对自己否定的愧疚感，这非常重要。

– 成就感让你追求卓越，愧疚感只是让你不想犯错(而这是做不到的)。

共 2 条评论>

👍 87



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) {
    if (runnable != null) {
        System.out.println("duplicate calls!");
        return;
    }
    runnable = () -> {
        long durationInMillis = durationInSeconds * 1000;
        long endTimeInMillis = System.currentTimeMillis();
```

```
    long startTimestamp = endTimestamp - durationInMilliseconds;
    doReport(startTimestamp, endTimestamp);
};
executor.scheduleAtFixedRate(runnable, 0, periodInSeconds, TimeUnit.SECONDS);
}
```

代码放在了: <https://github.com/gdhuocoder/Algorithms4/tree/master/designpattern/u40>

共 5 条评论 >

👍 19



Andy

2020-02-04

老师能提供课程代码吗?



👍 16



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。



👍 14



6点无痛早起学习的和...

2020-03-04

这两篇实战对自己的感觉就是越来越懵, 可能还需要多读读, 也不知道是不是跟自己的项目做的很少有原因, 前面的理论都还能消化

共 1 条评论 >

👍 7



undefined

2020-02-03

深入浅出, 过瘾。



👍 7



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, 优先判断该字段

是否已经变为true。如果是则不再往下执行。也算是保证该函数的幂等性。

个人疑问：

1.怎么做到这样分步展示重构过程的？我现在写，基本一边写就一边重构，停手也就差不多到合适的质量了。刻意要展示重构手法，展示的知识点会有很多疏漏，并无法做到这样一步一步的展示（下意识一步到位，并不知道怎么退到不好的代码结构）。

2.能理解栏主尽量不依赖任何框架的初衷。但对于java，spring其实才是标准，感觉是不是基于spring框架来写demo还好点？我现在比较喜欢让代码依赖spring框架来实现，感觉这样会显得优雅一些。栏主怎么看？

共 6 条评论 >

👍 7



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

共 3 条评论 >

👍 7



Q罗

2020-03-18

终于进去到期待已久的设计模式了



👍 4



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



👍 4