

26讲实战二（下）：如何实现一个支持各种统计规则的性能计数器



在上一节课中，我们对计数器框架做了需求分析和粗略的模块划分。今天这节课，我们利用面向对象设计、实现方法，并结合之前学过的设计思想、设计原则来看一下，如何编写灵活、可扩展的、高质量的代码实现。

话不多说，现在就让我们正式开始今天的学习吧！

小步快跑、逐步迭代

在上一节课中，我们将整个框架分为数据采集、存储、聚合统计、显示这四个模块。除此之外，关于统计触发方式（主动推送、被动触发统计）、统计时间区间（统计哪一个时间段内的数据）、统计时间间隔（对于主动推送方法，多久统计推送一次）我们也做了简单的设计。这里我就不重新描述了，你可以打开上一节课回顾一下。

虽然上一节课的最小原型为我们奠定了迭代开发的基础，但离我们最终期望的框架的样子还有很大的距离。我自己在写这篇文章的时候，试图去实现上面罗列的所有功能需求，希望写出一个完美的框架，发现这是件挺烧脑的事情，在写代码的过程中，一直有种“脑子不够使”的感觉。我这个有十多年工作经验的人尚且如此，对于没有太多经验的开发者来说，想一下子把所有需求都实现出来，更是一件非常有挑战的事情。一旦无法顺利完成，你可能就会有很强的挫败感，就会陷入自我否定的情绪中。

不过，即便你有能力将所有需求都实现，可能也要花费很大的设计精力和开发时间，迟迟没有产出，你的leader会因此产生很强的不可控感。对于现在的互联网项目来说，小步快跑、逐步迭代是一种更好的开发模式。所以，我们应该分多个版本逐步完善这个框架。第一个版本可以先实现一些基本功能，对于更高级、更复杂的功能，以及非功能性需求不做过高的要求，在后续的v2.0、v3.0……版本中继续迭代优化。

针对这个框架的开发，我们在v1.0版本中，暂时只实现下面这些功能。剩下的功能留在v2.0、v3.0版本，也就是我们后面的第39节和第40节课中再来讲解。

- 数据采集：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。
- 存储：负责将采集的原始数据保存下来，以便之后做聚合统计。数据的存储方式有很多种，我们暂时只支持Redis这一种存储方式，并且，采集与存储两个过程同步执行。

- 聚合统计：负责将原始数据聚合为统计数据，包括响应时间的最大值、最小值、平均值、99.9百分位值、99百分位值，以及接口请求的次數和tps。
- 显示：负责将统计数据以某种格式显示到终端，暂时只支持主动推送给命令行和邮件。命令行间隔n秒统计显示上m秒的数据（比如，间隔60s统计上60s的数据）。邮件每日统计上日的数据。

现在这个版本的需求比之前的要更加具体、简单了，实现起来也更加容易一些。实际上，学会结合具体的需求，做合理的预判、假设、取舍，规划版本的迭代设计开发，也是一个资深工程师必须要具备的能力。

面向对象设计与实现

在[第13节](#)和[第14节](#)课中，我们把面向对象设计与实现分开来讲解，界限划分比较明显。在实际的软件开发中，这两个过程往往是交叉进行的。一般是先有一个粗糙的设计，然后着手实现，实现的过程发现问题，再回过头来补充修改设计。所以，对于这个框架的开发来说，我们把设计和实现放到一块来讲解。

回顾上一节课中的最小原型的实现，所有的代码都耦合在一个类中，这显然是不合理的。接下来，我们就按照之前讲的面向对象设计的几个步骤，来重新划分、设计类。

1.划分职责进而识别出有哪些类

根据需求描述，我们先大致识别出下面几个接口或类。这一步不难，完全就是翻译需求。

- MetricsCollector类负责提供API，来采集接口请求的原始数据。我们可以为MetricsCollector抽象出一个接口，但这并不是必须的，因为暂时我们只能想到一个MetricsCollector的实现方式。
- MetricsStorage接口负责原始数据存储，RedisMetricsStorage类实现MetricsStorage接口。这样做是为了今后灵活地扩展新的存储方法，比如用HBase来存储。
- Aggregator类负责根据原始数据计算统计数据。
- ConsoleReporter类、EmailReporter类分别负责以一定频率统计并发送统计数据到命令行和邮件。至于ConsoleReporter和EmailReporter是否可以抽象出可复用的抽象类，或者抽象出一个公共的接口，我们暂时还不能确定。

2.定义类及类与类之间的关系

接下来就是定义类及属性和方法，定义类与类之间的关系。这两步没法分得很开，所以，我们今天将它们合在一起讲解。

大致地识别出几个核心的类之后，我的习惯性做法是，先在IDE中创建好这几个类，然后开始试着定义它们的属性和方法。在设计类、类与类之间交互的时候，我会不断地用之前学过的设计原则和思想来审视设计是否合理，比如，是否满足单一职责原则、开闭原则、依赖注入、KISS原则、DRY原则、迪米特法则，是否符合基于接口而非实现编程思想，代码是否高内聚、低耦合，是否可以抽象出可复用代码等等。

MetricsCollector类的定义非常简单，具体代码如下所示。对比上一节课中最小原型的代码，MetricsCollector通过引入RequestInfo类来封装原始数据信息，用一个采集函数代替了之前的两个函数。

```
public class MetricsCollector {  
    private MetricsStorage metricsStorage; //基于接口而非实现编程  
  
    //依赖注入  
    public MetricsCollector(MetricsStorage metricsStorage) {  
        this.metricsStorage = metricsStorage;  
    }  
  
    //用一个函数代替了最小原型中的两个函数  
    public void recordRequest(RequestInfo requestInfo) {  
        if (requestInfo == null || StringUtils.isBlank(requestInfo.getApiName())) {  
            return;  
        }  
        metricsStorage.saveRequestInfo(requestInfo);  
    }  
}  
  
public class RequestInfo {  
    private String apiName;  
    private double responseTime;  
    private long timestamp;  
    //...省略constructor/getter/setter方法...  
}
```

MetricsStorage类和RedisMetricsStorage类的属性和方法也比较明确。具体的代码实现如下所示。注意，一次性取太长时间区间的数据，可能会导致拉取太多的数据到内存中，有可能会撑爆内存。对于Java来说，就有可能会触发OOM（Out Of Memory）。而且，即便不出现OOM，内存还够用，但也会因为内存吃紧，导致频繁的Full GC，进而导致系统接口请求处理变慢，甚至超时。这个问题解决起来并不难，先留给你自己思考一下。我会在第40节课中解答。

```

public interface MetricsStorage {
    void saveRequestInfo(RequestInfo requestInfo);

    List<RequestInfo> getRequestInfos(String apiName, long startTimeInMillis, long endTimeInMillis);

    Map<String, List<RequestInfo>> getRequestInfos(long startTimeInMillis, long endTimeInMillis);
}

public class RedisMetricsStorage implements MetricsStorage {
    //...省略属性和构造函数等...
    @Override
    public void saveRequestInfo(RequestInfo requestInfo) {
        //...
    }

    @Override
    public List<RequestInfo> getRequestInfos(String apiName, long startTimestamp, long endTimestamp) {
        //...
    }

    @Override
    public Map<String, List<RequestInfo>> getRequestInfos(long startTimestamp, long endTimestamp) {
        //...
    }
}

```

MetricsCollector类和MetricsStorage类的设计思路比较简单，不同的人给出的设计结果应该大差不差。但是，统计和显示这两个功能就不一样了，可以有多种设计思路。实际上，如果我们把统计显示所要完成的功能逻辑细分一下的话，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3个过程的执行。

实际上，如果用一句话总结一下的话，**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**。所以，我们现在要做的工作就是，把以上的4个功能逻辑划分到几个类中。划分的方法有很多种，比如，我们可以把前两个逻辑放到一个类中，第3个逻辑放到另外一个类中，第4个逻辑作为上帝类（God Class）组合前面两个类来触发前3个逻辑的执行。当然，我们也可以把第2个逻辑单独放到一个类中，第1、3、4都放到另外一个类中。

至于到底选择哪种排列组合方式，判定的标准是，让代码尽量地满足低耦合、高内聚、单一职责、对扩展开放对修改关闭等之前讲到的各种设计原则和思想，尽量地让设计满足代码易复用、易读、易扩展、易维护。

我们暂时选择把第1、3、4逻辑放到ConsoleReporter或EmailReporter类中，把第2个逻辑放到Aggregator类中。其

中，Aggregator类负责的逻辑比较简单，我们把它设计成只包含静态方法的工具类。具体的代码实现如下所示：

```
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;  
        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;  
                }  
            }  
        });  
        int idx999 = (int)(count * 0.999);  
        int idx99 = (int)(count * 0.99);  
        if (count != 0) {  
            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...
}

```

ConsoleReporter类相当于一个上帝类，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命令行。具体的代码实现如下所示：

```

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

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

    // 第4个代码逻辑：定时触发第1、2、3代码逻辑的执行；
    public void startRepeatedReport(long periodInSeconds, long durationInSeconds) {
        executor.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                // 第1个代码逻辑：根据给定的时间区间，从数据库中拉取数据；
                long durationInMillis = durationInSeconds * 1000;
            }
        }, 0, periodInSeconds, TimeUnit.SECONDS);
    }
}

```

```

        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();
            // 第2个代码逻辑：根据原始数据，计算得到统计数据；
            RequestStat requestStat = Aggregator.aggregate(requestInfosPerApi, durationInMillis);
            stats.put(apiName, requestStat);
        }
        // 第3个代码逻辑：将统计数据显示到终端（命令行或邮件）；
        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;
    }

    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);
}
}

```

3.将类组装起来并提供执行入口

因为这个框架稍微有些特殊, 有两个执行入口: 一个是MetricsCollector类, 提供了一组API来采集原始数据; 另一个是ConsoleReporter类和EmailReporter类, 用来触发统计显示。框架具体的使用方式如下所示:


```

public class Demo {
    public static void main(String[] args) {
        MetricsStorage storage = new RedisMetricsStorage();
        ConsoleReporter consoleReporter = new ConsoleReporter(storage);
        consoleReporter.startRepeatedReport(60, 60);

        EmailReporter emailReporter = new EmailReporter(storage);
        emailReporter.addToAddress("wangzheng@xzg.com");
        emailReporter.startDailyReport();

        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设计与实现

我们前面讲到了SOLID、KISS、DRY、YAGNI、LOD等设计原则，基于接口而非实现编程、多用组合少用继承、高内聚低耦合等设计思想。我们现在就来看下，上面的代码实现是否符合这些设计原则和思想。

- MetricsCollector

MetricsCollector负责采集和存储数据，职责相对来说还算比较单一。它基于接口而非实现编程，通过依赖注入的方式来传递MetricsStorage对象，可以在不需要修改代码的情况下，灵活地替换不同的存储方式，满足开闭原则。

- MetricsStorage、RedisMetricsStorage

MetricsStorage和RedisMetricsStorage的设计比较简单。当我们需要实现新的存储方式的时候，只需要实现MetricsStorage接口即可。因为所有用到MetricsStorage和RedisMetricsStorage的地方，都是基于相同的接口函数来编程的，所以，除了在组装类的地方有所改动（从RedisMetricsStorage改为新的存储实现类），其他接口函数调用的地方都不需要改动，满足开闭原则。

- Aggregator

Aggregator类是一个工具类，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。当需要扩展新的统计功能的时候，需要修改aggregate()函数代码，并且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。所以，从刚刚的分析来看，这个类的设计可能存在职责不够单一、不易扩展等问题，需要在之后的版本中，对其结构做优化。

- ConsoleReporter、EmailReporter

ConsoleReporter和EmailReporter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。而且整个类负责的事情比较多，职责不是太单一。特别是显示部分的代码，可能会比较复杂（比如Email的展示方式），最好是将显示部分的代码逻辑拆分成独立的类。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不好。

今天我们给出的代码实现还是有诸多问题的，在后面的章节（第39、40讲）中，我们会慢慢优化，给你展示整个设计演进的过程，这比直接给你最终的最优方案要有意义得多！实际上，优秀的代码都是重构出来的，复杂的代码都是慢慢堆砌出来的。所以，当你看到那些优秀而复杂的开源代码或者项目代码的时候，也不必自惭形秽，觉得自己写不出来。毕竟罗马不是一天建成的，这些优秀的代码也是靠几年的时间慢慢迭代优化出来的。

重点回顾

好了，今天的内容到此就讲完了。我们一块总结回顾一下，你需要掌握的重点内容。

写代码的过程本就是一个修修改改、不停调整的过程，肯定不是一气呵成的。你看到的那些大牛开源项目的设计和实现，也都是在不停优化、修改过程中产生的。比如，我们熟悉的Unix系统，第一版很简单、粗糙，代码不到1万行。所以，迭代思维很重要，不要刚开始就追求完美。

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。至于到底选择哪种划分方法，判定的标准是让代码尽量地满足低耦合、高内聚、单一职责、对扩展开放对修改关闭等之前讲的各种设计原则和思想，尽量地做到代码可复用、易读、易扩展、易维护。

课堂讨论

今天课堂讨论题有下面两道。

1. 对于今天的设计与代码实现，你有没有发现哪些不合理的地方？有哪些可以继续优化的地方呢？或者留言说说你的设计方案。
2. 说一个你觉得不错的开源框架或者项目，聊聊你为什么觉得它不错？

欢迎在留言区写下你的答案，和同学们一起交流和分享。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



geek

新年快乐 一起学习 一起提高 2020

2020-01-01 00:31



辣么大

想了三点，希望和小伙伴们讨论一下：

- 1、RequestInfo save 一次写入一条。是否需要考虑通过设置参数，例如一次写入1000或10000条？好处不用频繁的与数据库建立连接。
- 2、聚合统计Aggregator是否可以考虑不写代码实现统计的逻辑，而是使用一条SQL查询实现同样的功能？
- 3、EmailReporter startDailyReport 没指定明确的统计起止时间。设置统计指定区间的request info，例如08:00~次日08:00，然后发邮件。

2020-01-01 06:23



Jxin

沙发！

1. 栏主新年快乐。零点发帖，啧啧啧。
2. 给出github地址吧，我们来提pr，一个学习用demo大家合力下就当练手，没必要自己死磕全实现哈。
3. 关于邮件和控制台两个接入层。实现代码重了。可以把定时统计下沉到下一层来实现，然后两个接入层共用这个实现。然后收集的统计数据类型应该可以提供差异化配置的api。在消费统计数据的消息时，做差异化分发，实现各接入层仅看到自己想看的数据。

4. spring1.x~3.x，兼容老版本做得挺好。springboot在自动装配的实现上下足了功夫（插件化，易插拔）。netty的实现也挺挺讲究，还能顺带学网络相关知识。以上其实都运用一系列设计原则。在没看栏主专栏前，我是啃这些学的场景。

2020-01-01 00:48



卫江

上面的代码设计与实现，我认为有两个重点是需要改进的：

1. 不同的统计规则，通过抽象统计规则抽象类，每一个具体的统计（最大时间，平均时间）单独实现，同时在 Aggregator 内中通过 List等容器保存所有的统计规则实现类，提供注册函数来动态添加新的统计规则，使得Aggregator否则开闭原则，各个统计规则也符合单一责任原则。
2. 显示方式很明显是一个变化点，需要抽象封装，抽象出 显示接口，在汇报类中通过依赖注入的方式来使用具体的显示类，这样一来，reporter类更加责任单一，我们也可以通过扩展新的显示类来扩展功能，符合开闭原则，每一个显示实现类更加否则单一责任。

2020-01-02 11:08



堵车

要写出优美的代码，首先要有一颗对丑陋代码厌恶的心

2020-01-02 10:44



Eden Ma

2020新年快乐 早上醒来第一件事就是听卖 者和看争哥的更新

2020-01-01 12:05



Murre

https://github.com/murrelsCoding/learning_geek/tree/master/src/main/java/design_pattern/demo2/performance_monitoring

敲了一下，主要是实现了redis存储部分逻辑，redis命令不是很熟，可能有更好的方案

2020-01-02 18:14



哈喽沃德

什么时候开始讲设计模式呢

2020-01-02 08:43



啦啦啦

新年快乐

2020-01-01 11:01



AaronYu

把老师的代码做了一个整理，试着运行了一下。

小伙伴们感兴趣的可以看一下：<https://github.com/Aaronyu29/DesignPattern/tree/master/src/u026>

2020-01-02 11:58



何沛

Aggregator考虑到后期新增新的维度统计，可以考虑使用责任链模式。

ConsoleReporter、EmailReporter 出现了代码复用，可以用模板设计模式。

2020-01-02 09:33



Young!

我觉得在使用方面需要优化，1,建议可以将使用哪个数据库存储方式，时间范围，使用邮箱还是命令行作为输出做成类似 spring 的可配置项，2,减少启动代码，最好使用一行或者注解就可以起到拦截请求并统计输出的作用。

2020-01-01 23:15



Frank

打卡，今天又进步一点点，利用元旦的时间，将上一篇和这一篇的内容过了一遍，参照文章的思路使用代码简单实现了一遍，加深了理解。

2020-01-01 21:36



Jeff.Smile

争哥这套课程确实呕心沥血，哈哈

2020-01-01 18:37



wenxueliu



timestamp

赞，记录思考过程才是最真实的案例

2020-01-01 15:23



Monday

RequestInfo.timestamp属性是接口响应的开始时间戳吗？如果是的话，说明我被Demo中的10234,11234这类数据给误导了

2020-01-01 12:05



Geek_3b1096

喜欢一小步一小步改进过程

2020-01-01 11:54



东方奇骥

因为我们项目统计数据较多，一般会写es，也会利用es的聚合功能。

2020-01-01 10:20



DFighting

我觉得代码里的问题主要有两处：

- 1、统计类应该抽象成一个接口，相关统计函数的实现可以做依赖注入，也可以不做
- 2、数据的采集和存储不应该放在一起，因为这样势必会影响业务代码的响应时间，虽然存储类抽象成为了一个接口，并通过依赖注入的方式便于扩展，但从采集数据和存储应该是不同的层次的设计。前者很难做到不侵入业务代码（兼顾性能的前提下），而后者很难不做到和存储解耦，这两个放在一起，太不合适了

2020-02-21 11:47



不记年

ConsoleReporter EmailReporter 这两个类的类名和职责不统一。从字面意思更像是负责显示的类。可以抽象出一个Reporter类。该类提供了基本的代码框架，通过builder模式将负责存储，统计，展示的类注入进来。也可以抽象出一个AbstractReporter类，提供一个负责展示是的抽象方法。ConsoleReporter EmailReporter 集成这个抽象类并实现各自的展示方法。我个人更倾向于第一种。

统计方面可以采用sql语句的形式来暴露给用户使用。因为我觉得产品最终的形式是要有一个web界面供用户配置各种规则的。采用sql的方式可以减少使用者的心智负担，也可以大大缩短开发人员的开发时间

2020-02-21 09:34