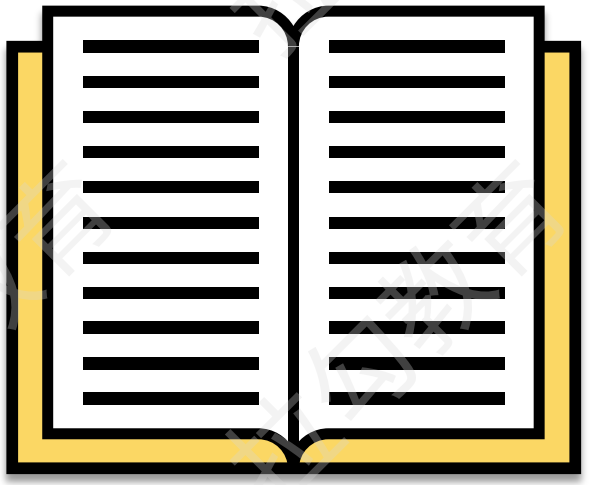
《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

第30讲：server-alarm 插件核心剖析
如何避免收到告警信息

前言

1. 指定告警的规则（Rule）
2. 接收监控数据
3. 告警检查，将预先定义好的告警规则阈值与接收到的监控数据进行比较
4. 如果监控数据符合告警规则，就会触发告警，那么将会通过指定途径将告警信息发送用户
5. 在收到告警消息之后，运维人员或是开发人员就会开始关注并处理相应的问题



AlarmCore


```
rules:
  service_resp_time_rule: #告警规则的名称，必须以"_rule"结尾
    #指标名称，该指标必须是 long、double或是 int类型，这里是服务响应时间
    metrics-name: service_resp_time
    include-names: #该规则适用的 serviceName，默认匹配全部服务
      - demo-webapp
      - demo-provider
    #告警阈值，即服务响应时长超过1s则匹配成功
    threshold: 1000
    op: ">" # 比较方式
    #告警检查的时间窗口
    period: 5
    #两层含义，在下面说明该告警规则的完整含义时由相应体现
    count: 3
    #当告警触发之后，后续连续2次告警检查即使成功，也不会再发送告警消息，也就是进入了沉默期，默认与period的配置值相同
    silence-period: 2
    message: {name} response time is longger than 1s # 告警信息
```

AlarmCore

该告警规则检查的是 demo-webapp 和 demo-provider 两个服务（include-names）的 service_resp_time 监控指标（metrics-name），5 分钟（period）为一个时间窗口，在一个时间窗口内，demo-webapp 响应时长有 3 次（count）超过（op）了 1s（threshold）即为符合告警条件，连续 3 个（count）时间窗口符合告警条件才真正会触发告警，发送相应的告警信息（message）为了防止连续告警消息造成骚扰，在触发告警之后的 2 个时间窗口 （silence-period）内，无论是否再次触发告警，不再发送任何告警消息

AlarmCore


```
Rules rules = ...//读取并解析 alarm-settings.yml 配置文件
//创建 NotifyHandler对象并初始化
NotifyHandler notifyHandler = new NotifyHandler(rules);
notifyHandler.init(new AlarmStandardPersistence());
```

AlarmCore

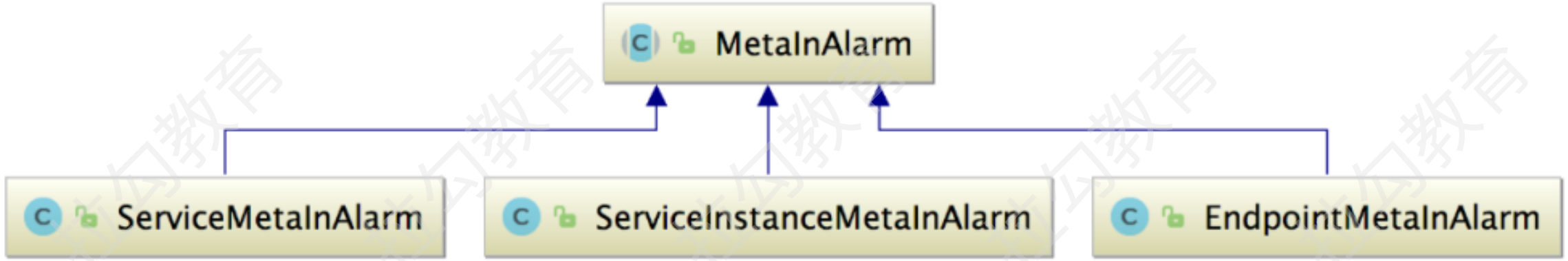

```
private LocalDateTime lastExecuteTime;

public void start(List<AlarmCallback> allCallbacks) {
    lastExecuteTime = LocalDateTime.now();
    Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
        List<AlarmMessage> alarmMessageList = new ArrayList<>(30);      LocalDateTime
checkTime = LocalDateTime.now();
        //计算当前时间与上次告警检查时间的差值
        int minutes = Minutes.minutesBetween(lastExecuteTime, checkTime).getMinutes();
        boolean[] hasExecute = new boolean[]{false};
        runningContext.values().forEach(ruleList -> ruleList.forEach(runningRule -> {
            if (minutes > 0) { //差值一分钟以上，才会进行告警检查
                runningRule.moveTo(checkTime); //这里的告警检查操作都是在 RunningRule中完成的
                if (checkTime.getSecondOfMinute() > 15) {
                    hasExecute[0] = true;
                    alarmMessageList.addAll(runningRule.check());
                }
            }
```

AlarmCore


```
            if (minutes > 0) {//差值一分钟以上，才会进行告警检查
                runningRule.moveTo(checkTime);//这里的告警检查操作都是在 RunningRule中完成的
                if (checkTime.getSecondOfMinute() > 15) {
                    hasExecute[0] = true;
                    alarmMessageList.addAll(runningRule.check());
                }
            }
        });
        if (hasExecute[0]) {//更新最近一次检查时间，注意，这里会保证lastExecuteTime始终为整分钟，例如：17:30:00或17:31:00
            lastExecuteTime = checkTime.minusSeconds(checkTime.getSecondOfMinute());
        }
        if (alarmMessageList.size() > 0) { //将告警信息通过AlarmCallback指定的方式发送出去，AlarmCallback的内容后面会展开分析
            allCallbacks.forEach(callback -> callback.doAlarm(alarmMessageList));
        }
    }, 10, 10, TimeUnit.SECONDS);
```

RunningRule



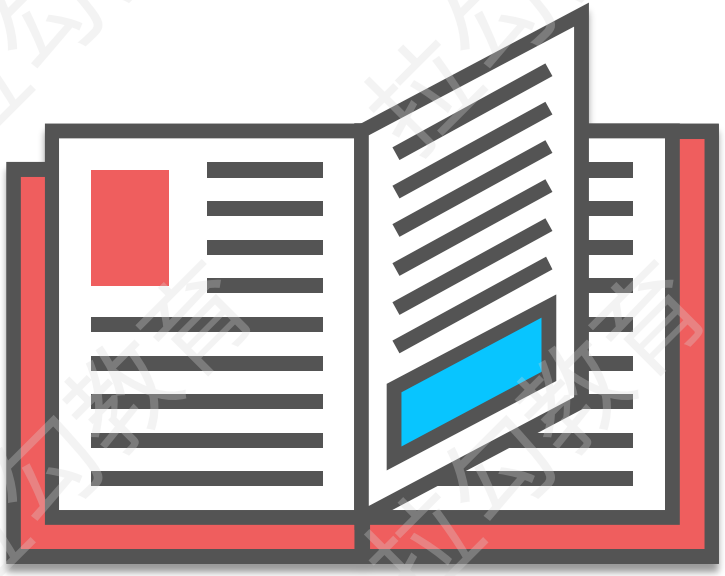
RunningRule


```
private int period;  //告警检查的时间窗口
//长度固定为period的List，其中每个元素对应一分钟的监控值
private LinkedList<Metrics> values;
private LocalDateTime endTime; //最后一次进行告警检查的时间
private int counter; //当前达到告警阈值的次数
private int silenceCountdown; //当前剩余的沉默周期数
```

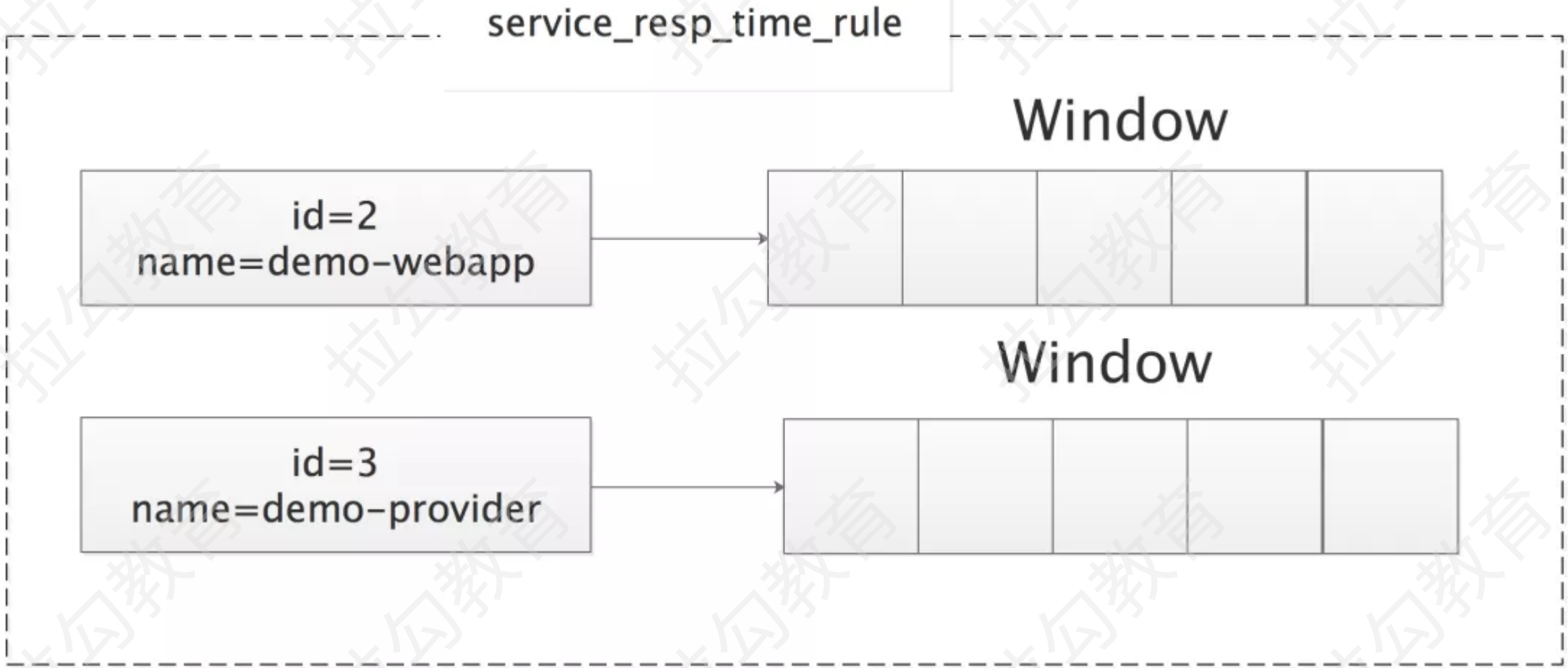
RunningRule

Window 中有三个核心方法：

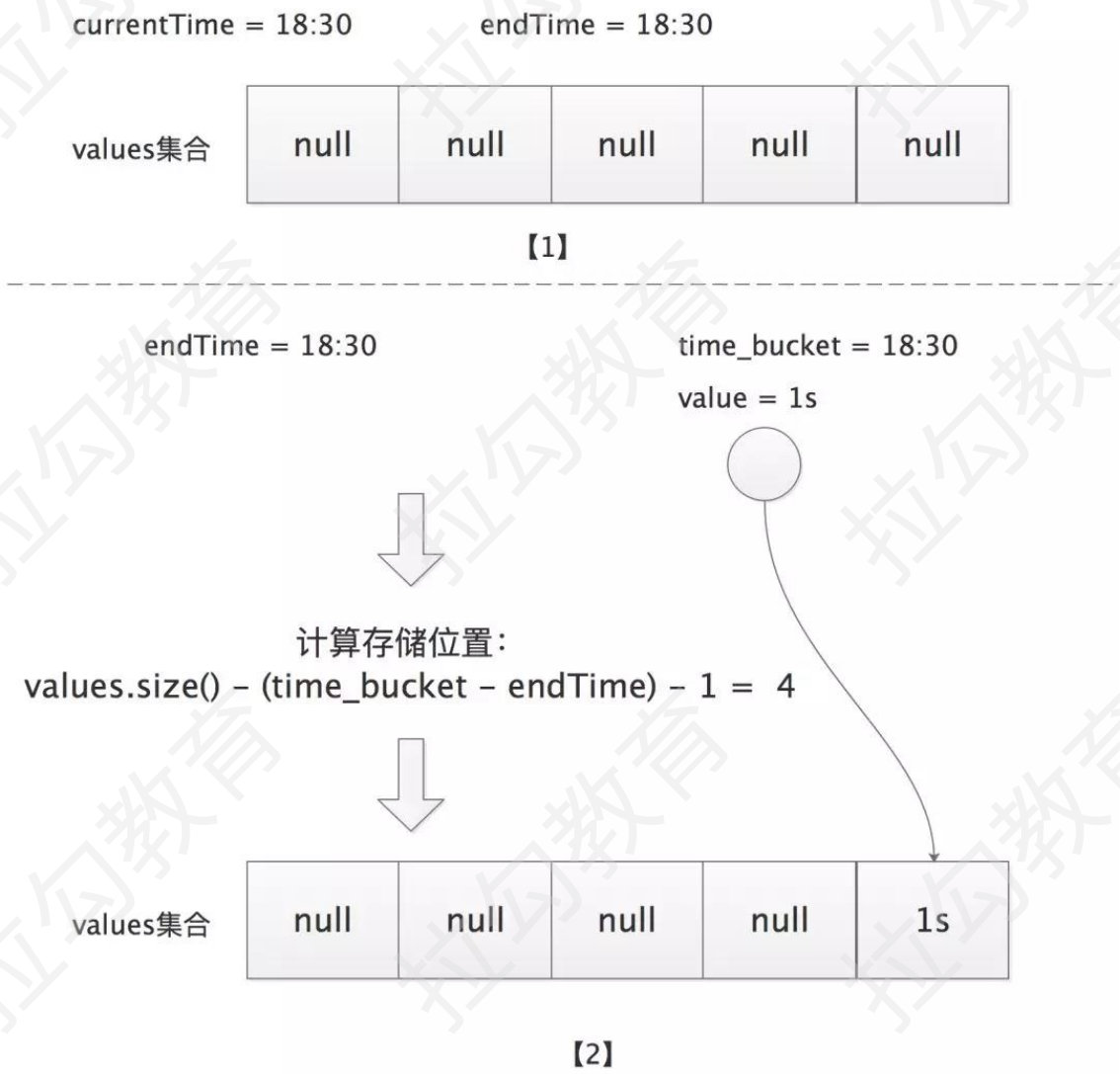
- **moveTo(LocalDateTime) 方法**：根据传入时间与 endTime 的差值更新 values 集合
- **add(Metrics) 方法**：将指定监控数据更新到 values 集合中相应的位置
- **checkAlarm() 方法**：根据 values 集合中记录的监控数据进行告警检查，并返回相应的告警消息



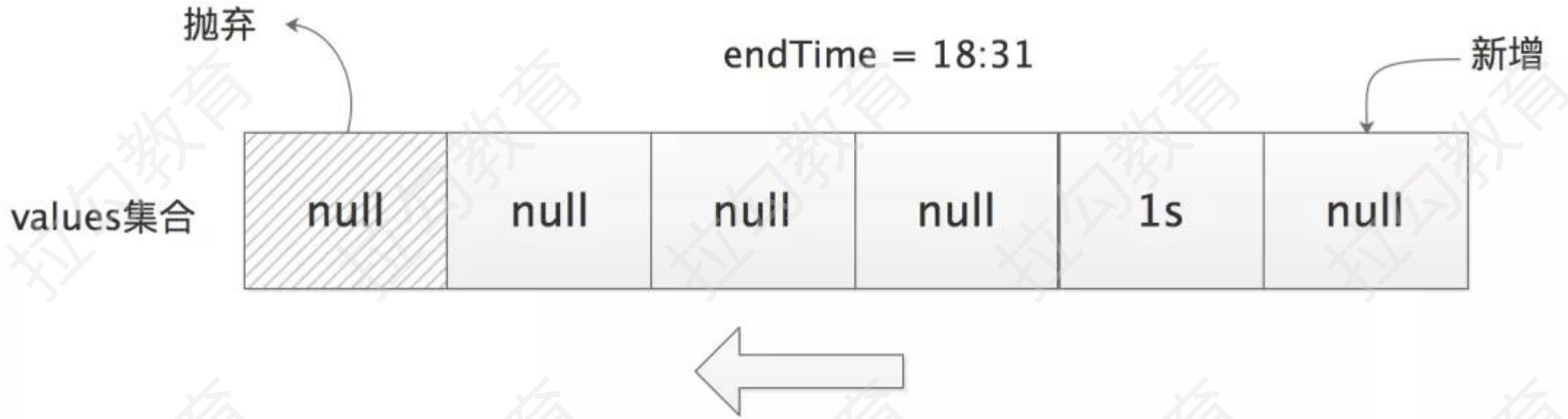
RunningRule



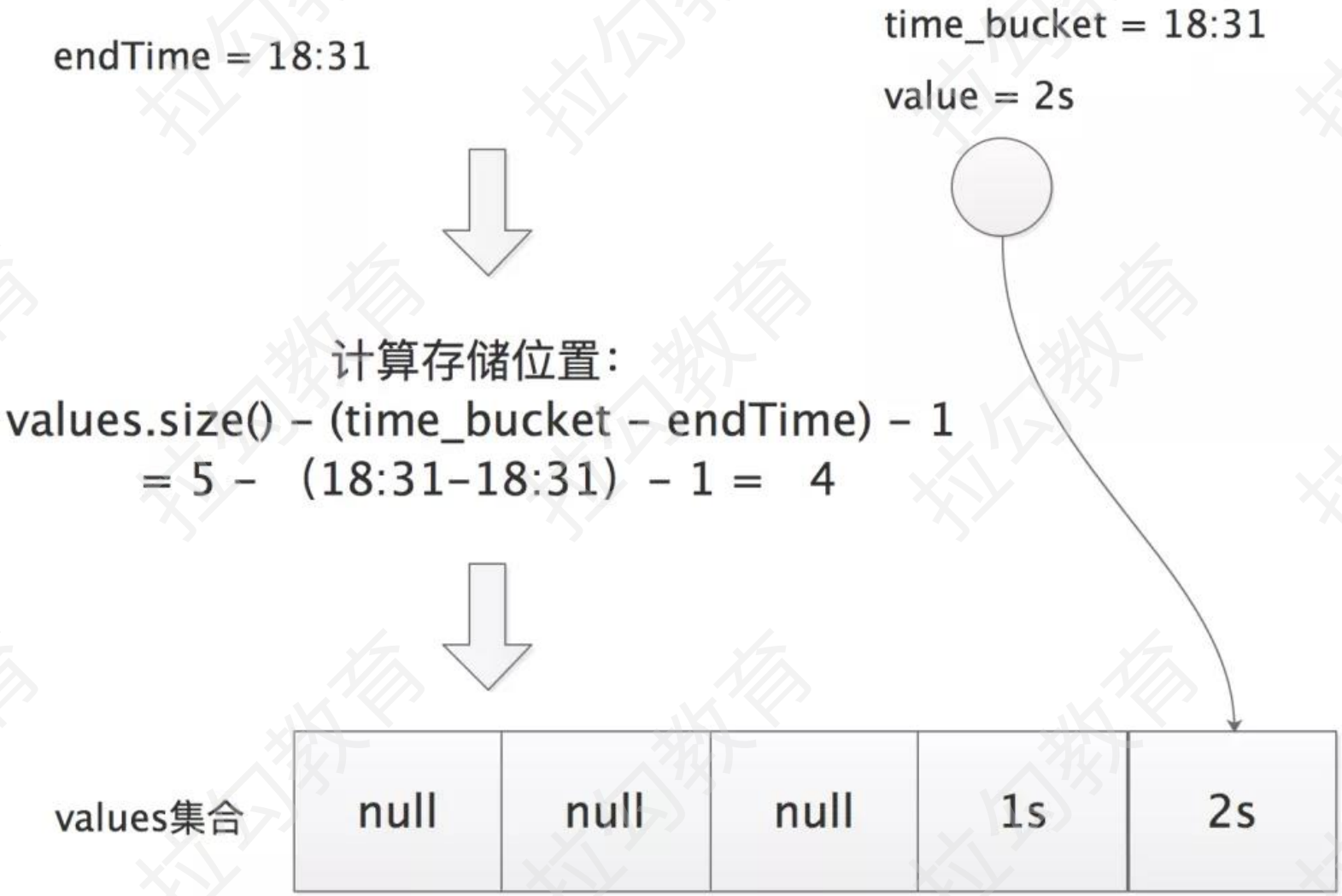
RunningRule



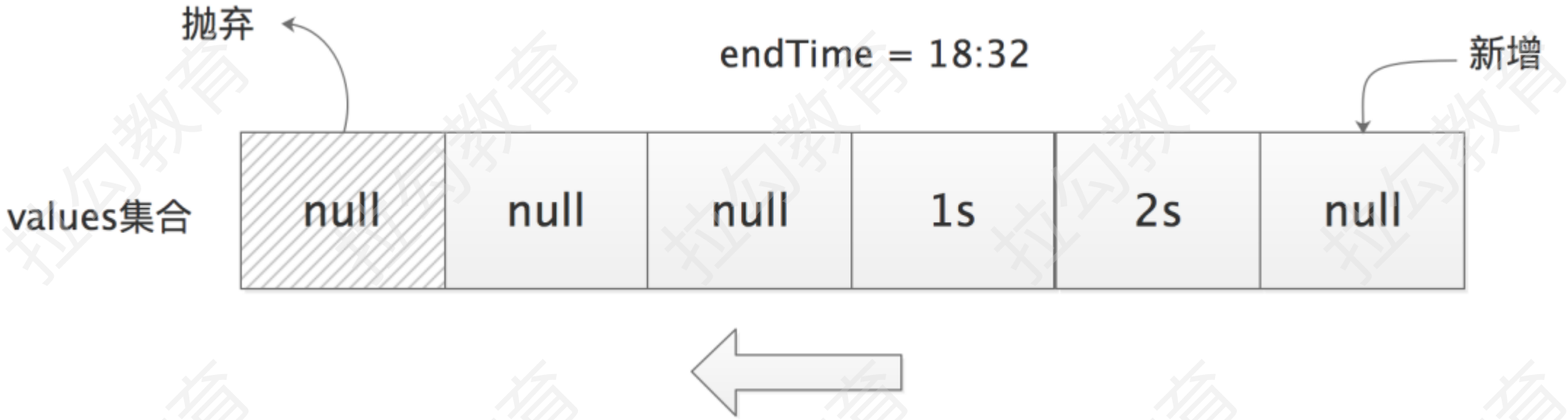
RunningRule



RunningRule



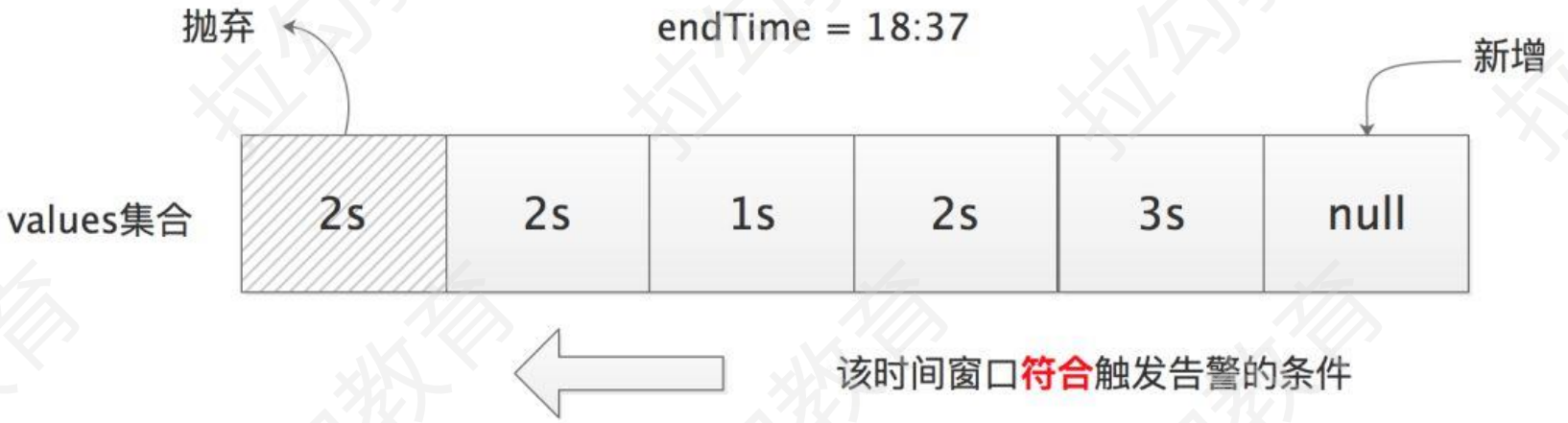
RunningRule

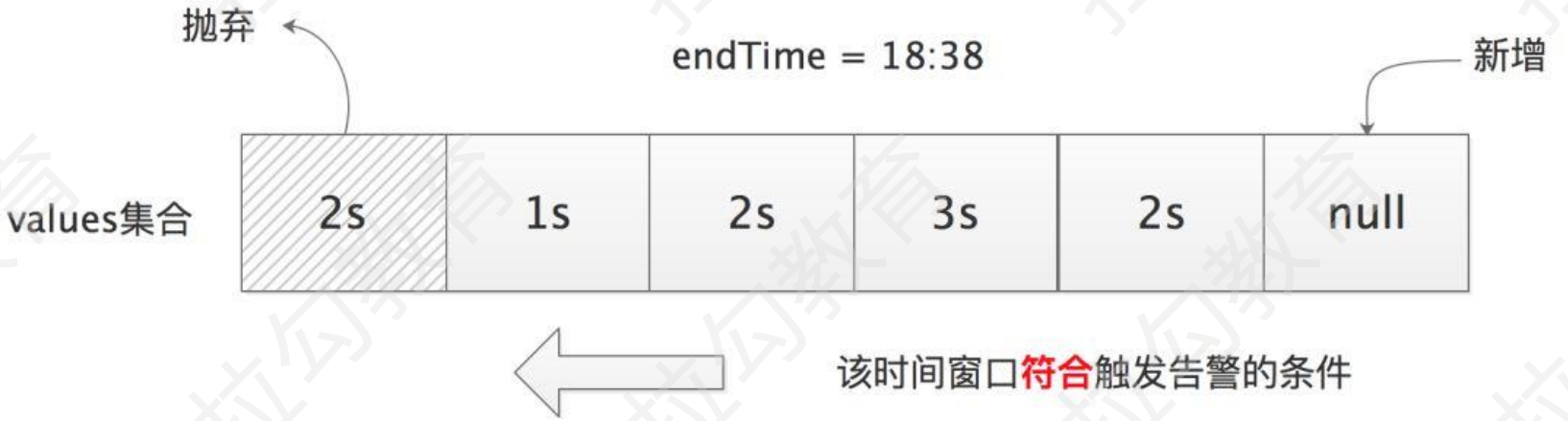


RunningRule

RunningRule

RunningRule





NotifyHandler

NotifyHandler

AnnotationScan

是 OAP 中的注解扫描器，它上面可以注册多个 AnnotationListener 监听器

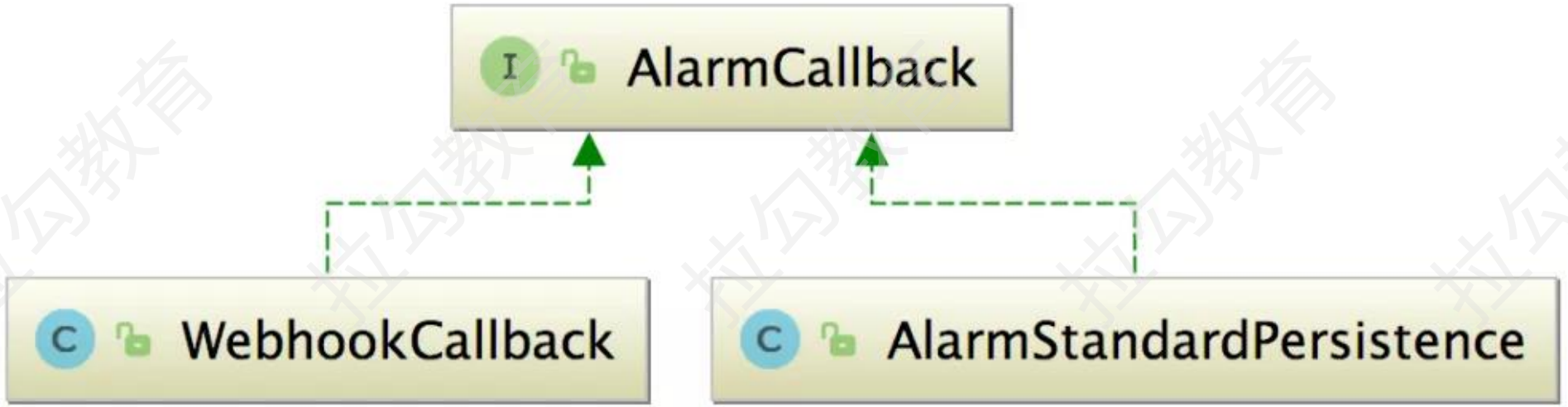
扫描 classpath 时，会根据类上的注解

从注册的 AnnotationListener 集合中找到匹配的 AnnotationListener 进行处理

NotifyHandler


```
public void notify(Metrics metrics) {
    ... ... //省略 Metrics 分类检查以及 MetaInAlarm 的创建过程
    //根据 metricsName 查找相应的告警规则
    List<RunningRule> runningRules =
core.findRunningRule(meta.getMetricsName());
    if (runningRules == null) {
        return;
    }
    //由 RunningRule 处理后续告警流程
    runningRules.forEach(rule -> rule.in(metaInAlarm, metrics));
}
```

发送告警消息


AlarmCallback
WebhookCallback
AlarmStandardPersistence

发送告警消息

Webhook

是常见的事件监听方式之一，它允许第三方应用监听系统的某些特定事件

例如这里的发送告警消息，当告警被触发之后

WebhookCallback 会通过 HTTP POST 方式将告警消息发送到第三方应用指定的 URL 地址

第三方应用通过监听该地址获取告警消息并展示给用户

例如在 Gitlab 中也提供了 Webhook 的功能

用户可以使用 Webhook 监听项目代码的 Push 事件，触发 Jenkins 的自动打包和部署

发送告警消息


```
rules:
 # 前文介绍的告警配置（略）
 webhooks: # 可以配置多个 URL
  - http://127.0.0.1/notify/
  - http://127.0.0.1/go-wechat/
```

发送告警消息


```
// 创建 HttpClient
CloseableHttpClient httpClient = HttpClients.custom().build();
//remoteEndpoints集合就是 alarm-settings.yml 文件中配置的 URL
remoteEndpoints.forEach(url -> {
    HttpPost post = new HttpPost(url); // 创建 HttpPost请求
    //配置请求的超时信息，ConnectionTimeOut、RequestTimeOut以及SocketTimeOut都是1s
    post.setConfig(requestConfig);
    post.setHeader("Accept", "application/json");
    post.setHeader("Content-type", "application/json");
    //生成JSON格式的告警信息
    StringEntity entity = new StringEntity(gson.toJson(alarmMessage));
    post.setEntity(entity);
    //发送请求
    CloseableHttpResponse httpResponse = httpClient.execute(post);
    //检查 Http 响应码
    StatusLine statusLine = httpResponse.getStatusLine();
    if (statusLine != null && statusLine.getStatusCode() != 200) {
        logger.error("send alarm to " + url + " failure. Response code: " + statusLine.getStatusCode());
    }
});
```

AlarmStandardPersistence

告警消息除了会通过 WebhookCallback 发送出去之外

还会通过 AlarmStandardPersistence 进行持久化

在收到 AlarmMessage 之后，AlarmStandardPersistence 会将其转换成 AlarmRecord

并交给 RecordStreamProcessor 进行持久化

larmRecord 的核心字段与 AlarmMessage 的字段基本一致

Next：第31讲《OAL 语言，原来定义创造一门新语言如此轻松》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U