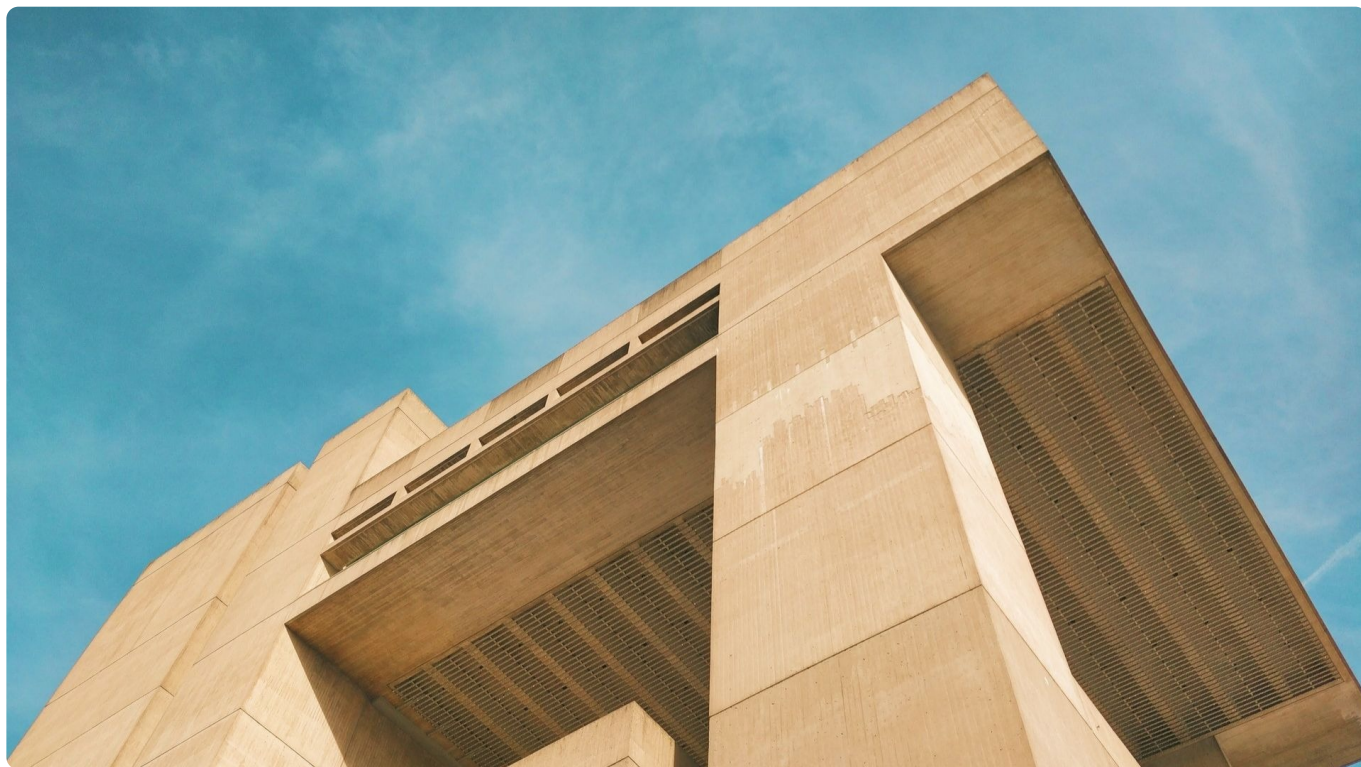


## 92 | 项目实战一：设计实现一个支持各种算法的限流框架（实现）

2020-06-03 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 12:43 大小 11.65M



上一节课，我们介绍了如何通过合理的设计，来实现功能性需求的同时，满足易用、易扩展、灵活、低延迟、高容错等非功能性需求。在设计的过程中，我们也借鉴了之前讲过的一些开源项目的设计思想。比如，我们借鉴了 Spring 的低侵入松耦合、约定优于配置等设计思想，还借鉴了 MyBatis 通过 MyBatis-Spring 类库将框架的易用性做到极致等设计思路。

今天，我们讲解这样一个问题，针对限流框架的开发，如何做高质量的代码实现。说的具体点就是，如何利用之前讲过的设计思想、原则、模式、编码规范、重构技巧等，写出易扩展、易维护、灵活、简洁、可复用、易测试的代码。



话不多少，让我们正式开始今天的学习吧！

## V1 版本功能需求

我们前面提到，优秀的代码是重构出来的，复杂的代码是慢慢堆砌出来的。小步快跑、逐步迭代是我比较推崇的开发模式。所以，针对限流框架，我们也不用想一下子就做得大而全。况且，在专栏有限的篇幅内，我们也不可能将一个大而全的代码阐述清楚。所以，我们可以先实现一个包含核心功能、基本功能的 V1 版本。

针对上两节课中给出的需求和设计，我们重新梳理一下，看看有哪些功能要放到 V1 版本中实现。

在 V1 版本中，对于接口类型，我们只支持 HTTP 接口（也就 URL）的限流，暂时不支持 RPC 等其他类型的接口限流。对于限流规则，我们只支持本地文件配置，配置文件格式只支持 YAML。对于限流算法，我们只支持固定时间窗口算法。对于限流模式，我们只支持单机限流。

尽管功能“裁剪”之后，V1 版本实现起来简单多了，但在编程开发的同时，我们还是要考虑代码的扩展性，预留好扩展点。这样，在接下来的新版本开发中，我们才能够轻松地扩展新的限流算法、限流模式、限流规则格式和数据源。

## 最小原型代码

上节课我们讲到，项目实战中的实现等于面向对象设计加实现。而面向对象设计与实现一般可以分为四个步骤：划分职责识别类、定义属性和方法、定义类之间的交互关系、组装类并提供执行入口。在 [第 14 讲](#) 中，我还带你用这个方法，设计和实现了一个接口鉴权框架。如果你印象不深刻了，可以回过头去再看下。

不过，我们前面也讲到，在平时的工作中，大部分程序员都是边写代码边做设计，边思考边重构，并不会严格地按照步骤，先做完类的设计再去写代码。而且，如果想一下子就把类设计得很好、很合理，也是比较难的。过度追求完美主义，只会导致迟迟下不了手，连第一行代码也敲不出来。所以，我的习惯是，先完全不考虑设计和代码质量，先把功能完成，先把基本的流程走通，哪怕所有的代码都写在一个类中也无所谓。然后，我们再针对这个 MVP 代码（最小原型代码）做优化重构，比如，将代码中比较独立的代码块抽离出来，定义成独立的类或函数。

我们按照先写 MVP 代码的思路，把代码实现出来。它的目录结构如下所示。代码非常简单，只包含 5 个类，接下来，我们针对每个类——讲解一下。

 复制代码

```
1 com.xzg.ratelimiter
2   --RateLimiter
3 com.xzg.ratelimiter.rule
4   --ApiLimit
5   --RuleConfig
6   --RateLimitRule
7 com.xzg.ratelimiter.alg
8   --RateLimitAlg
```

我们先来看下 **RateLimiter** 类。代码如下所示：

 复制代码

```
1 public class RateLimiter {
2     private static final Logger log = LoggerFactory.getLogger(RateLimiter.class)
3     // 为每个api在内存中存储限流计数器
4     private ConcurrentHashMap<String, RateLimitAlg> counters = new Concurr
5     private RateLimitRule rule;
6
7     public RateLimiter() {
8         // 将限流规则配置文件ratelimiter-rule.yaml中的内容读取到RuleConfig中
9         InputStream in = null;
10        RuleConfig ruleConfig = null;
11        try {
12            in = this.getClass().getResourceAsStream("/ratelimiter-rule.yaml");
13            if (in != null) {
14                Yaml yaml = new Yaml();
15                ruleConfig = yaml.loadAs(in, RuleConfig.class);
16            }
17        } finally {
18            if (in != null) {
19                try {
20                    in.close();
21                } catch (IOException e) {
22                    log.error("close file error:{}", e);
23                }
24            }
25        }
26
27        // 将限流规则构建成支持快速查找的数据结构RateLimitRule
28        this.rule = new RateLimitRule(ruleConfig);
29    }
30}
```

```

31 public boolean limit(String appId, String url) throws InternalErrorException
32     ApiLimit apiLimit = rule.getLimit(appId, url);
33     if (apiLimit == null) {
34         return true;
35     }
36
37     // 获取api对应内存中的限流计数器 (rateLimitCounter)
38     String counterKey = appId + ":" + apiLimit.getApi();
39     RateLimitAlg rateLimitCounter = counters.get(counterKey);
40     if (rateLimitCounter == null) {
41         RateLimitAlg newRateLimitCounter = new RateLimitAlg(apiLimit.getLimit())
42         rateLimitCounter = counters.putIfAbsent(counterKey, newRateLimitCounter)
43         if (rateLimitCounter == null) {
44             rateLimitCounter = newRateLimitCounter;
45         }
46     }
47
48     // 判断是否限流
49     return rateLimitCounter.tryAcquire();
50 }
51

```


RateLimiter 类用来串联整个限流流程。它先读取限流规则配置文件，映射为内存中的 Java 对象（RuleConfig），然后再将这个中间结构构建成一个支持快速查询的数据结构（RateLimitRule）。除此之外，这个类还提供供用户直接使用的最顶层接口（limit() 接口）。

**我们再来看下 RuleConfig 和 ApiLimit 两个类。**代码如下所示：

```

1 public class RuleConfig {
2     private List<UniformRuleConfig> configs;
3
4     public List<AppRuleConfig> getConfigs() {
5         return configs;
6     }
7
8     public void setConfigs(List<AppRuleConfig> configs) {
9         this.configs = configs;
10    }
11
12    public static class AppRuleConfig {
13        private String appId;
14        private List<ApiLimit> limits;
15
16        public AppRuleConfig() {}
17

```

 复制代码

```

18     public AppRuleConfig(String appId, List<ApiLimit> limits) {
19         this.appId = appId;
20         this.limits = limits;
21     }
22     //...省略getter、setter方法...
23 }
24 }
25
26 public class ApiLimit {
27     private static final int DEFAULT_TIME_UNIT = 1; // 1 second
28     private String api;
29     private int limit;
30     private int unit = DEFAULT_TIME_UNIT;
31
32     public ApiLimit() {}
33
34     public ApiLimit(String api, int limit) {
35         this(api, limit, DEFAULT_TIME_UNIT);
36     }
37
38     public ApiLimit(String api, int limit, int unit) {
39         this.api = api;
40         this.limit = limit;
41         this.unit = unit;
42     }
43     // ...省略getter、setter方法...
44 }

```

从代码中，我们可以看出来，RuleConfig 类嵌套了另外两个类 AppRuleConfig 和 ApiLimit。这三个类跟配置文件的三层嵌套结构完全对应。我把对应关系标注在了下面的示例中，你可以对照着代码看下。

 复制代码

```


1  configs:                <!--对应RuleConfig-->
2  - appId: app-1          <!--对应AppRuleConfig-->
3    limits:
4      - api: /v1/user <!--对应ApiLimit-->
5        limit: 100
6        unit: 60
7      - api: /v1/order
8        limit: 50
9  - appId: app-2
10    limits:
11      - api: /v1/user
12        limit: 50
13      - api: /v1/order
14        limit: 50

```





这个类是限流算法实现类。它实现了最简单的固定时间窗口限流算法。每个接口都要在内存中对应一个 RateLimitAlg 对象，记录在当前时间窗口内已经被访问的次数。RateLimitAlg 类的代码如下所示。对于代码的算法逻辑，你可以看下上节课中对固定时间窗口限流算法的讲解。

 复制代码

```
1 public class RateLimitAlg {
2     /* timeout for {@code Lock.tryLock() }. */
3     private static final long TRY_LOCK_TIMEOUT = 200L; // 200ms.
4     private Stopwatch stopwatch;
5     private AtomicInteger currentCount = new AtomicInteger(0);
6     private final int limit;
7     private Lock lock = new ReentrantLock();
8
9     public RateLimitAlg(int limit) {
10         this(limit, Stopwatch.createStarted());
11     }
12
13     @VisibleForTesting
14     protected RateLimitAlg(int limit, Stopwatch stopwatch) {
15         this.limit = limit;
16         this.stopwatch = stopwatch;
17     }
18
19     public boolean tryAcquire() throws InternalErrorException {
20         int updatedCount = currentCount.incrementAndGet();
21         if (updatedCount <= limit) {
22             return true;
23         }
24
25         try {
26             if (lock.tryLock(TRY_LOCK_TIMEOUT, TimeUnit.MILLISECONDS)) {
27                 try {
28                     if (stopwatch.elapsed(TimeUnit.MILLISECONDS) > TimeUnit.SECONDS.toMi
29                         currentCount.set(0);
30                         stopwatch.reset();
31                     }
32                     updatedCount = currentCount.incrementAndGet();
33                     return updatedCount <= limit;
34                 } finally {
35                     lock.unlock();
36                 }
37             } else {
38                 throw new InternalErrorException("tryAcquire() wait lock too long:" +
39             }
40         } catch (InterruptedException e) {
41             throw new InternalErrorException("tryAcquire() is interrupted by lock-ti
42         }
43     }
```

## Review 最小原型代码

刚刚给出的 MVP 代码，虽然总共也就 200 多行，但已经实现了 V1 版本中规划的功能。不过，从代码质量的角度来看，它还有很多值得优化的地方。现在，我们现在站在一个 Code Reviewer 的角度，来分析一下这段代码的设计和实现。

结合 SOLID、DRY、KISS、LOD、基于接口而非实现编程、高内聚松耦合等经典的设计思想和原则，以及编码规范，我们从代码质量评判标准的角度重点剖析一下，这段代码在可读性、扩展性等方面的表现。其他方面的表现，比如复用性、可测试性等，这些你可以比葫芦画瓢，自己来进行分析。

### 首先，我们来看下代码的可读性。

影响代码可读性的因素有很多。我们重点关注目录设计（package 包）是否合理、模块划分是否清晰、代码结构是否高内聚低耦合，以及是否符合统一的编码规范这几点。

因为涉及的代码不多，目录结构前面也给出了，总体来说比较简单，所以目录设计、包的划分没有问题。

按照上节课中的模块划分，RuleConfig、ApiLimit、RateLimitRule 属于“限流规则”模块，负责限流规则的构建和查询。RateLimitAlg 属于“限流算法”模块，提供了基于内存的单机固定时间窗口限流算法。RateLimiter 类属于“集成使用”模块，作为最顶层类，组装其他类，提供执行入口（也就是调用入口）。不过，RateLimiter 类作为执行入口，我们希望它只负责组装工作，而不应该包含具体的业务逻辑，所以，RateLimiter 类中，从配置文件中读取限流规则这块逻辑，应该拆分出来设计成独立的类。

如果我们把类与类之间的依赖关系图画出来，你会发现，它们之间的依赖关系很简单，每个类的职责也比较单一，所以类的设计满足单一职责原则、LOD 迪米特法则、高内聚松耦合的要求。

从编码规范上来讲，没有超级大的类、函数、代码块。类、函数、变量的命名基本能达意，也符合最小惊奇原则。虽然，有些命名不能一眼就看出是干啥的，有些命名采用了缩写，比



如 RateLimitAlg，但是我们起码能猜个八九不离十，结合注释（限于篇幅注释都没有写，并不代表不需要写），很容易理解和记忆。

总结一下，在最小原型代码中，目录设计、代码结构、模块划分、类的设计还算合理清晰，基本符合编码规范，代码的可读性不错！

## 其次，我们再来看下代码的扩展性。

实际上，这段代码最大的问题就是它的扩展性，也是我们最关注的，毕竟后续还有更多版本的迭代开发。编写可扩展代码，关键是要建立扩展意识。这就像下象棋，我们要多往前想几步，为以后做准备。在写代码的时候，我们要时刻思考，这段代码如果要扩展新的功能，那是否可以在尽量少改动代码的情况下完成，还是需要要大动干戈，推倒重写。

具体到 MVP 代码，不易扩展的最大原因是，没有遵循基于接口而非实现的编程思想，没有接口抽象意识。比如，RateLimitAlg 类只是实现了固定时间窗口限流算法，也没有提炼出更加抽象的算法接口。如果我们要替换其他限流算法，就要改动比较多的代码。其他类的设计也有同样的问题，比如 RateLimitRule。

除此之外，在 RateLimiter 类中，配置文件的名称、路径，是硬编码在代码中的。尽管我们说约定优于配置，但也要兼顾灵活性，能够让用户在需要的时候，自定义配置文件名称、路径。而且，配置文件的格式只支持 Yaml，之后扩展其他格式，需要对这部分代码做很大的改动。

## 重构最小原型代码

根据刚刚对 MVP 代码的剖析，我们发现，它的可读性没有太大问题，问题主要在于可扩展性。主要的修改点有两个，一个是将 RateLimiter 中的规则配置文件的读取解析逻辑拆出来，设计成独立的类，另一个是参照基于接口而非实现编程思想，对于 RateLimitRule、RateLimitAlg 类提炼抽象接口。

按照这个修改思路，我们对代码进行重构。重构之后的目录结构如下所示。我对每个类都稍微做了说明，你可以对比着重构前的目录结构来看。

```
1 // 重构前：
```


 复制代码

```

2  com.xzg.ratelimiter
3      --RateLimiter
4  com.xzg.ratelimiter.rule
5      --ApiLimit
6      --RuleConfig
7      --RateLimitRule
8  com.xzg.ratelimiter.alg
9      --RateLimitAlg
10
11 // 重构后:
12 com.xzg.ratelimiter
13     --RateLimiter(有所修改)
14 com.xzg.ratelimiter.rule
15     --ApiLimit(不变)
16     --RuleConfig(不变)
17     --RateLimitRule(抽象接口)
18     --TrieRateLimitRule(实现类, 就是重构前的RateLimitRule)
19 com.xzg.ratelimiter.rule.parser
20     --RuleConfigParser(抽象接口)
21     --YamlRuleConfigParser(Yaml格式配置文件解析类)
22     --JsonRuleConfigParser(Json格式配置文件解析类)
23 com.xzg.ratelimiter.rule.datasource
24     --RuleConfigSource(抽象接口)
25     --FileRuleConfigSource(基于本地文件的配置类)
26 com.xzg.ratelimiter.alg
27     --RateLimitAlg(抽象接口)
28

```

其中, RateLimiter 类重构之后的代码如下所示。代码的改动集中在构造函数中, 通过调用 RuleConfigSource 来实现了限流规则配置文件的加载。


 复制代码

```

1  public class RateLimiter {
2      private static final Logger log = LoggerFactory.getLogger(RateLimiter.class)
3      // 为每个api在内存中存储限流计数器
4      private ConcurrentHashMap<String, RateLimitAlg> counters = new ConcurrencySafeHashMap<>();
5      private RateLimitRule rule;
6
7      public RateLimiter() {
8          //改动主要在这里: 调用RuleConfigSource类来实现配置加载
9          RuleConfigSource configSource = new FileRuleConfigSource();
10         RuleConfig ruleConfig = configSource.load();
11         this.rule = new TrieRateLimitRule(ruleConfig);
12     }
13
14     public boolean limit(String appId, String url) throws InternalErrorException {
15         //...代码不变...
16     }
17 }

```

我们再来看下，从 RateLimiter 中拆分出来的限流规则加载的逻辑，现在是如何设计的。这部分涉及的类主要是下面几个。我把关键代码也贴在了下面。其中，各个 Parser 和 RuleConfigSource 类的设计有点类似策略模式，如果要添加新的格式的解析，只需要实现对应的 Parser 类，并且添加到 FileRuleConfig 类的 PARSER\_MAP 中就可以了。

 复制代码

```
1  com.xzg.ratelimiter.rule.parser
2      --RuleConfigParser(抽象接口)
3      --YamlRuleConfigParser(Yaml格式配置文件解析类)
4      --JsonRuleConfigParser(Json格式配置文件解析类)
5  com.xzg.ratelimiter.rule.datasource
6      --RuleConfigSource(抽象接口)
7      --FileRuleConfigSource(基于本地文件的配置类)
8
9  public interface RuleConfigParser {
10      RuleConfig parse(String configText);
11      RuleConfig parse(InputStream in);
12  }
13
14  public interface RuleConfigSource {
15      RuleConfig load();
16  }
17
18  public class FileRuleConfigSource implements RuleConfigSource {
19      private static final Logger log = LoggerFactory.getLogger(FileRuleConfigSource.class);
20
21      public static final String API_LIMIT_CONFIG_NAME = "ratelimiter-rule";
22      public static final String YAML_EXTENSION = "yaml";
23      public static final String YML_EXTENSION = "yml";
24      public static final String JSON_EXTENSION = "json";
25
26      private static final String[] SUPPORT_EXTENSIONS =
27          new String[] {YAML_EXTENSION, YML_EXTENSION, JSON_EXTENSION};
28      private static final Map<String, RuleConfigParser> PARSER_MAP = new HashMap<>();
29
30      static {
31          PARSER_MAP.put(YAML_EXTENSION, new YamlRuleConfigParser());
32          PARSER_MAP.put(YML_EXTENSION, new YamlRuleConfigParser());
33          PARSER_MAP.put(JSON_EXTENSION, new JsonRuleConfigParser());
34      }
35
36      @Override
37      public RuleConfig load() {
38          for (String extension : SUPPORT_EXTENSIONS) {
39              InputStream in = null;
40              try {
```

```
41         in = this.getClass().getResourceAsStream("/") + getFileNameByExt(extension);
42         if (in != null) {
43             RuleConfigParser parser = PARSER_MAP.get(extension);
44             return parser.parse(in);
45         }
46     } finally {
47         if (in != null) {
48             try {
49                 in.close();
50             } catch (IOException e) {
51                 log.error("close file error:{}", e);
52             }
53         }
54     }
55 }
56 return null;
57 }
58
59 private String getFileNameByExt(String extension) {
60     return API_LIMIT_CONFIG_NAME + "." + extension;
61 }
62
```

## 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，你需要重点掌握的内容。

优秀的代码是重构出来的，复杂的代码是慢慢堆砌出来的。小步快跑、逐步迭代是我比较推崇的开发模式。追求完美主义会让我们迟迟无法下手。所以，为了克服这个问题，一方面，我们可以规划多个小版本来开发，不断迭代优化；另一方面，在编程实现的过程中，我们可以先实现 MVP 代码，以此来优化重构。

如何对 MVP 代码优化重构呢？我们站在 Code Reviewer 的角度，结合 SOLID、DRY、KISS、LOD、基于接口而非实现编程、高内聚松耦合等经典的设计思想和原则，以及编码规范，从代码质量评判标准的角度，来剖析代码在可读性、扩展性、可维护性、灵活、简洁、复用性、可测试性等方面的表现，并且针对性地去优化不足。

## 课堂讨论

1. 针对 MVP 代码，如果让你做 code review，你还能发现哪些问题？如果让你做重构，你还会做哪些修改和优化？

2. 如何重构代码，支持自定义限流规则配置文件名和路径？如果你熟悉 Java，你可以去了解一下 Spring 的设计思路，看看如何借鉴到限流框架中来解决这个问题？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

## 更多课程推荐

# MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇  
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 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 91 | 项目实战一：设计实现一个支持各种算法的限流框架（设计）

下一篇 93 | 项目实战二：设计实现一个通用的接口幂等框架（分析）

## 精选留言 (11)

写留言



jaryoung

2020-06-03

课后习题二：

如何重构代码，支持自定义限流规则配置文件名和路径？

```
public static final String DEFAULT_API_LIMIT_CONFIG_NAME = "ratelimiter-rule";  
private final String customApiLimitConfigPath;...
```

展开 ▾



6



**HuaMax**

2020-06-03

stopwatch.reset()之后要调用stopwatch.start()重新开始，或者stopwatch.stop().start()  
(), 亲入坑。。。



4



**高源**

2020-06-03

老师今天讲的骨架，有代码吗，我想结合你讲的自己再多考虑和分析，学习其中的方法解决的问题



3



**Jxin**

2020-06-03

- 1.随手写都如此牛逼。。。
- 2.还是有个git代码仓好点，这样手机看难受。
- 3.为什么要懒加载，直接在初始化时，将算法规则与算法实例绑定，将api与限流算法实例绑定。对于这个限流框架的应用场景不是更合适吗。如此便可以把懒加载的代码抽离，使业务聚焦业务而不用关心实例创建。...

展开 ▾



2



**leezer**

2020-06-03

RatelimitAlg在重构后应该是可支持多种算法形式，那么在limit调用时应该不是直接new出来，可以通过策略形式进行配置，而算法的选取应该包含默认和指定，也可以配置到文件规则里面。



2



**Jie**

2020-06-05

<https://github.com/wangzheng0822/ratelimiter4j>

老师忘记在专栏里面放自己项目的地址了么，翻看隔壁算法之美发现的

展开 ▾



1







**Geek\_54edc1**

2020-06-04

1、RateLimiter类中，构建api对应在内存中的限流计数器（RateLimitAlg）这个逻辑可以独立出来，初始化的过程中，就将api和相应RateLimitAlg实现类的对应关系建立好； 2、可以使用DI框架，FileRuleConfigSource构建时，从bean配置文件读取构造参数，如果没有提供构造参数就用默认值

展开 ∨



1



**Heaven**

2020-06-03

1.可以将配置类和实际的拦截器接口实现类进行相分离,然后在实现类里面去执行查找接口拦截规则并执行对应接口的Alg,对于Alg实现类,抽取出接口,方便自定义算法,并且在内部实现诸如漏桶算法的实现,利用用户配置和策略模式来进行实现

2.对于这个问题,可以参考Spring给出的Resource接口,并给出了基于不同的读取方式的实现类,而且为了简化开发,给出ResourceLoader,并且还有着DefaultResourceLoader,可以根...

展开 ∨



1



**Liam**

2020-06-03

Ralimiter#tryAcquire 方法，前三行，先更新count是否有问题，当前时间窗口可能会累积上一个时间窗口的计数，导致统计不准确

展开 ∨



1



**傲慢与偏执，**

2020-06-03

学习学习

展开 ∨



1



**马以**

2020-06-03

哈哈，新鲜出炉

展开 ∨



1