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）

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，写出易读、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，我们针对每个类一一讲解一下。

```
com.xzg.ratelimiter
```

```
  --RateLimiter
com.xzg.ratelimiter.rule
  --ApiLimit
  --RuleConfig
  --RateLimitRule
com.xzg.ratelimiter.alg
  --RateLimitAlg
```

我们先来看下RateLimiter类。代码如下所示：

```
public class RateLimiter {
  private static final Logger log = LoggerFactory.getLogger(RateLimiter.class);
  // 为每个api在内存中存储限流计数器
  private ConcurrentHashMap<String, RateLimitAlg> counters = new ConcurrentHashMap<>();
  private RateLimitRule rule;

  public RateLimiter() {
    // 将限流规则配置文件ratelimiter-rule.yaml中的内容读取到RuleConfig中
    InputStream in = null;
    RuleConfig ruleConfig = null;
    try {
      in = this.getClass().getResourceAsStream("/ratelimiter-rule.yaml");
      if (in != null) {
        Yaml yaml = new Yaml();
        ruleConfig = yaml.loadAs(in, RuleConfig.class);
      }
    } finally {
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          log.error("close file error:{}", e);
        }
      }
    }

    // 将限流规则构建成支持快速查找的数据结构RateLimitRule
    this.rule = new RateLimitRule(ruleConfig);
  }

  public boolean limit(String appId, String url) throws InternalErrorException {
    ApiLimit apiLimit = rule.getLimit(appId, url);
    if (apiLimit == null) {
      return true;
    }

    // 获取api对应在内存中的限流计数器（rateLimitCounter）
    String counterKey = appId + ":" + apiLimit.getApi();
    RateLimitAlg rateLimitCounter = counters.get(counterKey);
    if (rateLimitCounter == null) {
      RateLimitAlg newRateLimitCounter = new RateLimitAlg(apiLimit.getLimit());
      rateLimitCounter = counters.putIfAbsent(counterKey, newRateLimitCounter);
      if (rateLimitCounter == null) {
        rateLimitCounter = newRateLimitCounter;
      }
    }

    // 判断是否限流
    return rateLimitCounter.tryAcquire();
  }
```

```
}
```

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：

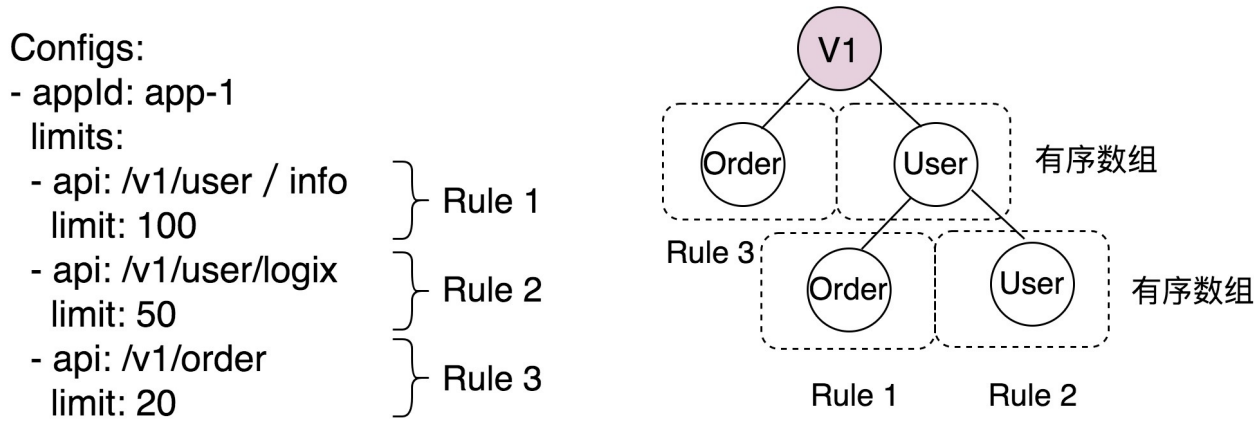
```
public class RuleConfig {
  private List<UniformRuleConfig> configs;

  public List<AppRuleConfig> getConfigs() {
    return configs;
  }

  public void setConfigs(List<AppRuleConfig> configs) {
    this.configs = configs;
  }

  public static class AppRuleConfig {
    private String appId;
    private List<ApiLimit> limits;

    public AppRuleConfig() {}

    public AppRuleConfig(String appId, List<ApiLimit> limits) {
      this.appId = appId;
      this.limits = limits;
    }
    //...省略getter、setter方法...
  }
}

public class ApiLimit {
  private static final int DEFAULT_TIME_UNIT = 1; // 1 second
  private String api;
  private int limit;
  private int unit = DEFAULT_TIME_UNIT;

  public ApiLimit() {}

  public ApiLimit(String api, int limit) {
    this(api, limit, DEFAULT_TIME_UNIT);
  }

  public ApiLimit(String api, int limit, int unit) {
    this.api = api;
    this.limit = limit;
    this.unit = unit;
  }
  // ...省略getter、setter方法...
}
```

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。

```
configs:          <!--对应RuleConfig-->
- appId: app-1    <!--对应AppRuleConfig-->
  limits:
  - api: /v1/user <!--对应ApiLimit-->
    limit: 100
    unit: 60
  - api: /v1/order
    limit: 50
- appId: app-2
  limits:
  - api: /v1/user
    limit: 50
  - api: /v1/order
    limit: 50
```

我们再来看下RateLimitRule这个类。

你可能会好奇，有了RuleConfig来存储限流规则，为什么还要RateLimitRule类呢？这是因为，限流过程中会频繁地查询接口对应的限流规则，为了尽可能地提高查询速度，我们需要将限流规则组织成一种支持按照URL快速查询的数据结构。考虑到URL的重复度比较高，且需要按照前缀来匹配，我们这里选择使用Trie树这种数据结构。我举了个例子解释一下，如下图所示。左边的限流规则对应到Trie树，就是图中右边的样子。



RateLimitRule的实现代码比较多，我就不在这里贴出来了，我只给出它的定义，如下所示。如果你感兴趣的话，可以自己实现一下，也可以参看我的另一个专栏《数据结构与算法之美》的第55讲。在那节课中，我们对各种接口匹配算法有非常详细的讲解。

```
public class RateLimitRule {
  public RateLimitRule(RuleConfig ruleConfig) {
    //...
  }

  public ApiLimit getLimit(String appId, String api) {
    //...
  }
}
```

最后，我们看下RateLimitAlg这个类。

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。

```
public class RateLimitAlg {
  /* timeout for {@code Lock.tryLock() }. */
  private static final long TRY_LOCK_TIMEOUT = 200L;  // 200ms.
  private Stopwatch stopwatch;
  private AtomicInteger currentCount = new AtomicInteger(0);
  private final int limit;
  private Lock lock = new ReentrantLock();

  public RateLimitAlg(int limit) {
    this(limit, Stopwatch.createStarted());
  }

  @VisibleForTesting
  protected RateLimitAlg(int limit, Stopwatch stopwatch) {
    this.limit = limit;
    this.stopwatch = stopwatch;
  }

  public boolean tryAcquire() throws InternalErrorException {
    int updatedCount = currentCount.incrementAndGet();
    if (updatedCount <= limit) {
      return true;
    }

    try {
      if (lock.tryLock(TRY_LOCK_TIMEOUT, TimeUnit.MILLISECONDS)) {
        try {
          if (stopwatch.elapsed(TimeUnit.MILLISECONDS) > TimeUnit.SECONDS.toMillis(1)) {
            currentCount.set(0);
            stopwatch.reset();
          }
          updatedCount = currentCount.incrementAndGet();
          return updatedCount <= limit;
        } finally {
          lock.unlock();
        }
      } else {
        throw new InternalErrorException("tryAcquire() wait lock too long:" + TRY_LOCK_TIMEOUT + "ms");
      }
    } catch (InterruptedException e) {
      throw new InternalErrorException("tryAcquire() is interrupted by lock-time-out.", e);
    }
  }
}
```

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。

```
// 重构前：
com.xzg.ratelimiter
  --RateLimiter
com.xzg.ratelimiter.rule
  --ApiLimit
  --RuleConfig
  --RateLimitRule
com.xzg.ratelimiter.alg
  --RateLimitAlg

// 重构后：
com.xzg.ratelimiter
  --RateLimiter(有所修改)
com.xzg.ratelimiter.rule
  --ApiLimit(不变)
  --RuleConfig(不变)
  --RateLimitRule(抽象接口)
  --TrieRateLimitRule(实现类，就是重构前的RateLimitRule)
com.xzg.ratelimiter.rule.parser
  --RuleConfigParser(抽象接口)
  --YamlRuleConfigParser(Yaml格式配置文件解析类)
  --JsonRuleConfigParser(Json格式配置文件解析类)
com.xzg.ratelimiter.rule.datasource
  --RuleConfigSource(抽象接口)
  --FileRuleConfigSource(基于本地文件的配置类)
com.xzg.ratelimiter.alg
  --RateLimitAlg(抽象接口)
  --FixedTimeWinRateLimitAlg(实现类，就是重构前的RateLimitAlg)
```

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。

```
public class RateLimiter {
  private static final Logger log = LoggerFactory.getLogger(RateLimiter.class);
  // 为每个api在内存中存储限流计数器
  private ConcurrentHashMap<String, RateLimitAlg> counters = new ConcurrentHashMap<>();
  private RateLimitRule rule;

  public RateLimiter() {
    //改动主要在这里：调用RuleConfigSource类来实现配置加载
    RuleConfigSource configSource = new FileRuleConfigSource();
    RuleConfig ruleConfig = configSource.load();
    this.rule = new TrieRateLimitRule(ruleConfig);
  }

  public boolean limit(String appId, String url) throws InternalErrorException, InvalidUrlException {
    //...代码不变...
  }
```

```
}
```

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。

```
com.xzg.ratelimiter.rule.parser
  --RuleConfigParser(抽象接口)
  --YamlRuleConfigParser(Yaml格式配置文件解析类)
  --JsonRuleConfigParser(Json格式配置文件解析类)
com.xzg.ratelimiter.rule.datasource
  --RuleConfigSource(抽象接口)
  --FileRuleConfigSource(基于本地文件的配置类)

public interface RuleConfigParser {
  RuleConfig parse(String configText);
  RuleConfig parse(InputStream in);
}

public interface RuleConfigSource {
  RuleConfig load();
}

public class FileRuleConfigSource implements RuleConfigSource {
  private static final Logger log = LoggerFactory.getLogger(FileRuleConfigSource.class);

  public static final String API_LIMIT_CONFIG_NAME = "ratelimiter-rule";
  public static final String YAML_EXTENSION = "yaml";
  public static final String YML_EXTENSION = "yml";
  public static final String JSON_EXTENSION = "json";

  private static final String[] SUPPORT_EXTENSIONS =
      new String[] {YAML_EXTENSION, YML_EXTENSION, JSON_EXTENSION};
  private static final Map<String, RuleConfigParser> PARSER_MAP = new HashMap<>();

  static {
    PARSER_MAP.put(YAML_EXTENSION, new YamlRuleConfigParser());
    PARSER_MAP.put(YML_EXTENSION, new YamlRuleConfigParser());
    PARSER_MAP.put(JSON_EXTENSION, new JsonRuleConfigParser());
  }

  @Override
  public RuleConfig load() {
    for (String extension : SUPPORT_EXTENSIONS) {
      InputStream in = null;
      try {
        in = this.getClass().getResourceAsStream("/" + getFileNameByExt(extension));
        if (in != null) {
          RuleConfigParser parser = PARSER_MAP.get(extension);
          return parser.parse(in);
        }
      } finally {
        if (in != null) {
          try {
            in.close();
          } catch (IOException e) {
            log.error("close file error:{}", e);
          }
```

```
        }
      }
    }
    return null;
  }

  private String getFileNameByExt(String extension) {
    return API_LIMIT_CONFIG_NAME + "." + extension;
  }
}
```

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。

精选留言：

- jaryoung 2020-06-03 11:49:36
  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;
  public FileRuleConfigSource(String configLocation) {
  this.customApiLimitConfigPath = configLocation;
  }

  private String getFileNameByExt(String extension) {
  return StringUtils.isEmpty(customApiLimitConfigPath) ?
  DEFAULT_API_LIMIT_CONFIG_NAME + "." + extension
  : customApiLimitConfigPath;
  }

Spring boot 如何实现配置文件约定和扫描？可以去看看ConfigFileApplicationListener 这个类，如何跑起来，请去debug，不懂怎么debug，请新建窗口输入 google.com [3赞]

- 高源 2020-06-03 06:17:06
  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 [2赞]

- Geek_54edc1 2020-06-04 17:43:28
  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

- Heaven 2020-06-03 19:31:49
  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,可以根据传入前缀,来创建不同的Resource,对于字符串查找树这个实现,我是真的没想到,不过可以在这个基础上,借鉴HashMap的实现,在api接口足够少的时候,使用简单的map保存,多了再转为树
  再往深了说,BeanFactory需要传入资源生成对应的实体Bean,而为了简化开发,一般是使用ApplicationContext来初始化Bean,需要传入一个资源给ApplicationContext,并在里面动态解析生成Bean对象,这样的流程,值得我们的框架借鉴点有很多

- HuaMax 2020-06-03 18:25:08
  stopwatch.reset()之后要调用stopwatch.start()重新开始，或者stopwatch.stop().start()，亲入坑。。。

- Jxin 2020-06-03 11:58:27
  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。
  4.还得考虑动态限流配置调整的功能。

- Liam 2020-06-03 09:06:24
  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

- leezer 2020-06-03 08:50:56
  RatelimitAlg在重构后应该是可支持多种算法形式，那么在limit调用的时候应该不是直接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。

- 傲慢与偏执， 2020-06-03 08:28:41
  学习学习

- 马以 2020-06-03 00:14:37
  哈哈，新鲜出炉