sentinel源码分析

1. sentinel源码工程搭建

1.1 环境准备

在nacos的官网介绍中，sentinel源码运行，需要的java运行环境有：

- JDK 1.8+
- Maven 3.2+

1.2 源码构建

1.2.1 源码下载

从github上，下载sentinel的源码到本地；

```
https://github.com/alibaba/Sentinel/releases
```

1.6.3
d0220

v1.6.3

sczyh30 released this on 30 Jul 2019 · 254 commits to master since this release

This version provides entire support for managing API gateway flow rules and customized API groups in Sentinel dashboard.

Features / Enhancements

- Add support for managing gateway flow rules and customized API group in Sentinel dashboard (#869)
- Add support for excluding some URLs in Web Servlet CommonFilter (#914)
- Add `Ordered` interface support for Spring Cloud Gateway filter (#937)
- Support displaying SystemRule of CPU usage strategy in the dashboard (#927)
- Use the unified context name in Web Servlet filter (#944)

Bug fixes

- Fix the empty value matching problem in `GatewayParamParser` (#937)

Thanks for the contributors: @cdfive, @Crazy10552, @jasonjoo2010, @linlinisme, @lym-ifae

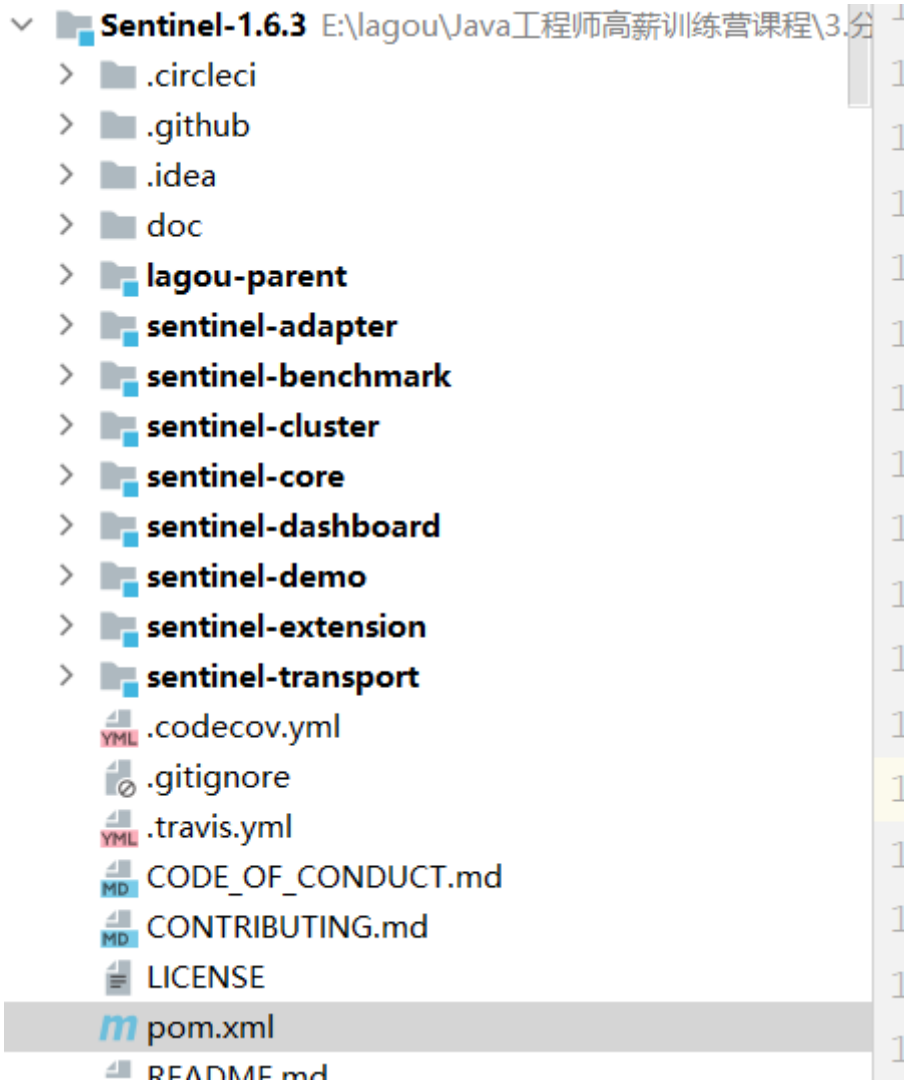
Assets 3

sentinel-dashboard-1.6.3.jar	20.1 MB
Source code (zip)	
Source code (tar.gz)	

1.2.2 导入idea工程

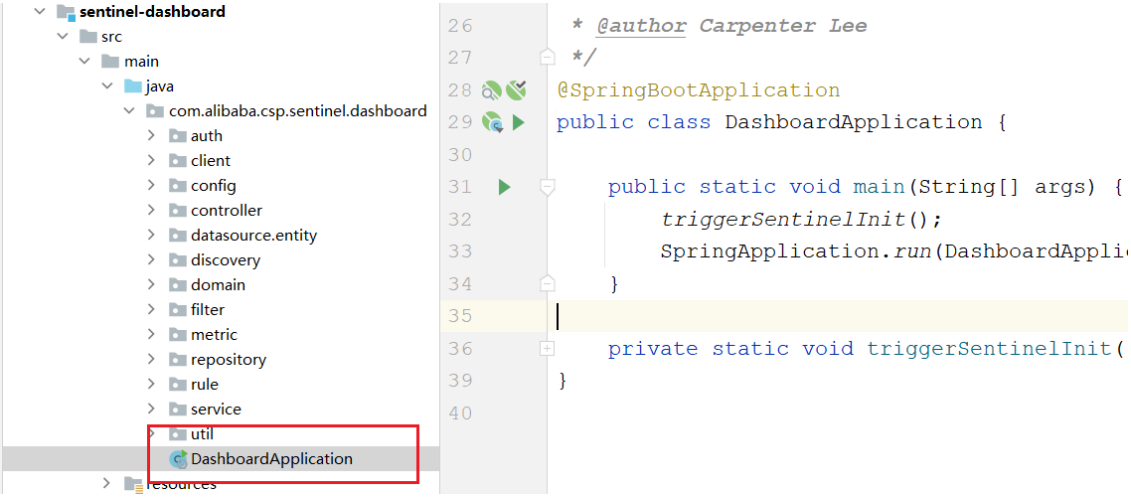
1. 将lagou-parent工程放入sentinel源码中导入idea

2. idea工程目录



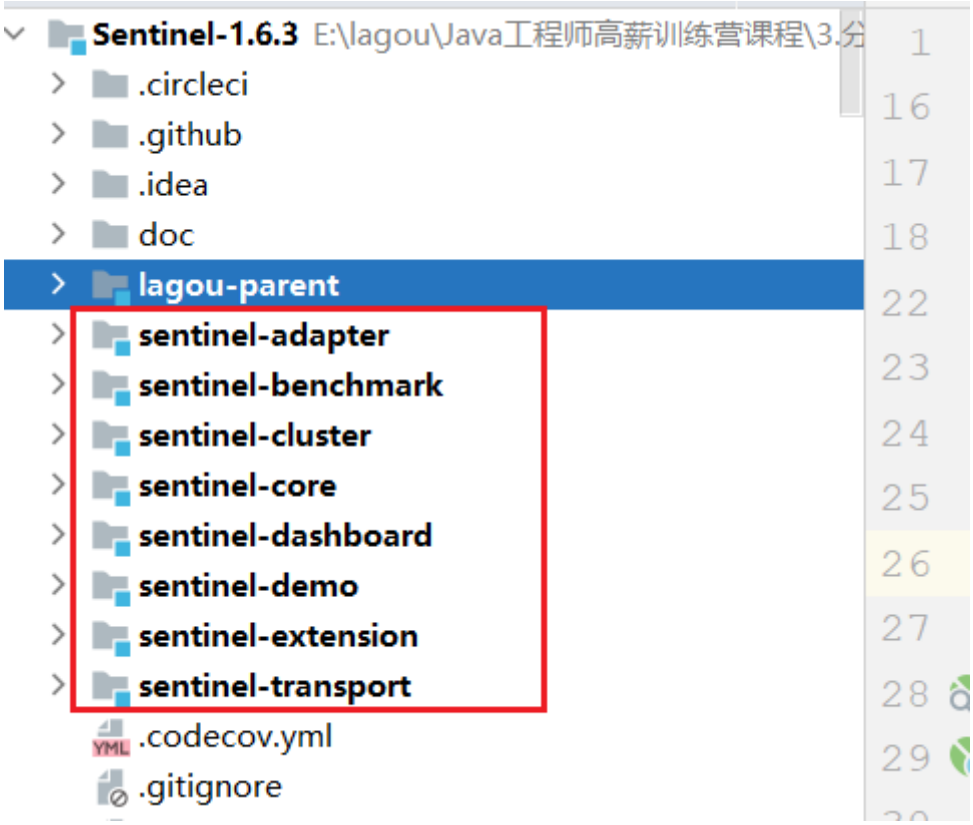
3. 工程启动

进入到sentinel-dashboard模块下，启动该模块下的com.alibaba.csp.sentinel.dashboard.DashboardApplication类。



1.3 sentinel项目结构

先来看下整个sentinel项目结构

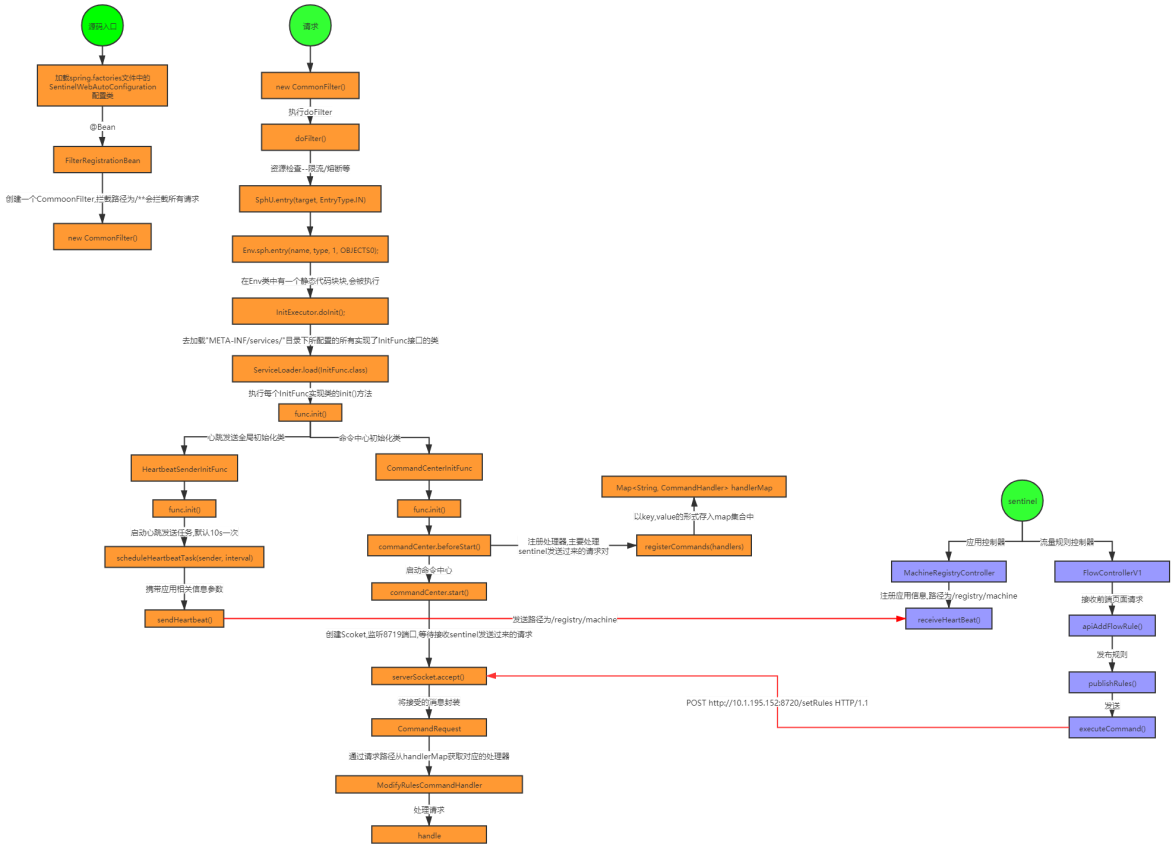


- **sentinel-core 核心模块，限流、降级、系统保护等都在这里实现**
- **sentinel-dashboard 控制台模块，可以对连接上的sentinel客户端实现可视化的管理**
- **sentinel-transport 传输模块，提供了基本的监控服务端和客户端的API接口，以及一些基于不同库的实现**
- **sentinel-extension 扩展模块，主要对DataSource进行了部分扩展实现**
- **sentinel-adapter 适配器模块，主要实现了对一些常见框架的适配**
- **sentinel-demo 样例模块，可参考怎么使用sentinel进行限流、降级等**
- **sentinel-benchmark 基准测试模块，对核心代码的精确性提供基准测试**

2. sentinel源码

2.1 客户端服务注册

2.1.1 客户端服务注册流程分析



2.1.2 主要源码跟踪

1. 导入sentinel的起步依赖后,会加载spring-cloud-alibaba-sentinel-2.1.0.RELEASE.jar下面的spring.factories文件,在文件中加载SentinelWebAutoConfiguration类,针对sentinel的自动配置类

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.alibaba.cloud.sentinel.SentinelWebAutoConfiguration,\
com.alibaba.cloud.sentinel.SentinelWebFluxAutoConfiguration,\
com.alibaba.cloud.sentinel.endpoint.SentinelEndpointAutoConfiguration,\
com.alibaba.cloud.sentinel.custom.SentinelAutoConfiguration,\
com.alibaba.cloud.sentinel.feign.SentinelFeignAutoConfiguration

org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker=\
com.alibaba.cloud.sentinel.custom.SentinelCircuitBreakerConfiguration
```

2. SentinelWebAutoConfiguration类中声明了一个FilterRegistrationBean,在这个方法中会创建sentinel核心的一个过滤器CommonFilter

```
@Bean
@ConditionalOnProperty(name = "spring.cloud.sentinel.filter.enabled",
matchIfMissing = true)
public FilterRegistrationBean sentinelFilter() {
    FilterRegistrationBean<Filter> registration = new
FilterRegistrationBean<>();
    // 获取sentinel的过滤器配置信息
    SentinelProperties.Filter filterConfig = properties.getFilter();

    if (filterConfig.getUrlPatterns() == null
```

```
        || filterConfig.getUrlPatterns().isEmpty()) {
        List<String> defaultPatterns = new ArrayList<>();
        //设置过滤器拦截路径为/*,拦截所有请求
        defaultPatterns.add("/*");
        filterConfig.setUrlPatterns(defaultPatterns);
    }

    registration.addUrlPatterns(filterConfig.getUrlPatterns().toArray(new
String[0]));
    //创建CommonFilter过滤器
    Filter filter = new CommonFilter();
    registration.setFilter(filter);
    registration.setOrder(filterConfig.getOrder());
    registration.addInitParameter("HTTP_METHOD_SPECIFY",
          String.valueOf(properties.getHttpMethodSpecify()));
    log.info(
          "[Sentinel Starter] register Sentinel CommonFilter with
urlPatterns: {}.",
          filterConfig.getUrlPatterns());
    return registration;

}
```

3. CommonFilter过滤器中doFilter方法会拦截所有请求

```
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
        throws IOException, ServletException {
    HttpServletRequest sRequest = (HttpServletRequest) request;
    Entry urlEntry = null;
    Entry httpMethodUrlEntry = null;

    try {
        //获取请求路径
        String target = FilterUtil.filterTarget(sRequest);
        UrlCleaner urlCleaner = WebCallbackManager.getUrlCleaner();
        if (urlCleaner != null) {
            target = urlCleaner.clean(target);
        }

        if (!StringUtil.isEmpty(target)) {
            // Parse the request origin using registered origin parser.
            String origin = parseOrigin(sRequest);
            ContextUtil.enter(WebServletConfig.WEB_SERVLET_CONTEXT_NAME,
origin);
            //资源检查--限流/熔断等.
            urlEntry = SphU.entry(target, EntryType.IN);
            // Add method specification if necessary
            if (httpMethodSpecify) {
                httpMethodUrlEntry =
SphU.entry(sRequest.getMethod().toUpperCase() + COLON + target,
                        EntryType.IN);
            }
        }
        chain.doFilter(request, response);
    } catch (BlockException e) {
        HttpServletResponse sResponse = (HttpServletResponse) response;
```

```
        // Return the block page, or redirect to another URL.
        WebCallbackManager.getUrlBlockHandler().blocked(sRequest, sResponse,
e);
    } catch (IOException | ServletException | RuntimeException e2) {
        Tracer.traceEntry(e2, urlEntry);
        Tracer.traceEntry(e2, httpMethodUrlEntry);
        throw e2;
    } finally {
        if (httpMethodUrlEntry != null) {
            httpMethodUrlEntry.exit();
        }
        if (urlEntry != null) {
            urlEntry.exit();
        }
        ContextUtil.exit();
    }
}
```

4. SphU.entry(target, EntryType.IN)方法进行资源初始化,限流 熔断等操作

```
public static Entry entry(String name, EntryType type) throws BlockException
{
    // Env类中有静态方法会被调用
    return Env.sph.entry(name, type, 1, OBJECTS0);
}
```

5. Env类

```
public class Env {

    public static final Sph sph = new CtSph();

    static {
        // 执行初始化
        InitExecutor.doInit();
    }

}
```

6. doInit方法

```
public static void doInit() {
    //判断是否是第一次初始化,不是则直接返回
    if (!initialized.compareAndSet(false, true)) {
        return;
    }
    try {
        //此处去加载"META-INF/services/"目录下所配置的所有实现了InitFunc接口的类
        ServiceLoader<InitFunc> loader = ServiceLoader.load(InitFunc.class);
        List<OrderWrapper> initList = new ArrayList<OrderWrapper>();
        for (InitFunc initFunc : loader) {
            RecordLog.info("[InitExecutor] Found init func: " +
initFunc.getClass().getCanonicalName());
            //将加载完的所有实现类排序
            insertSorted(initList, initFunc);
        }
```

```
        for (OrderWrapper w : initList) {
            //执行每个InitFunc实现类的init()方法,init()方法又会去加载其它所需资源
            w.func.init();
            RecordLog.info(String.format("[InitExecutor] Executing %s with
order %d",
                w.func.getClass().getCanonicalName(), w.order));
        }
    } catch (Exception ex) {
        RecordLog.warn("[InitExecutor] WARN: Initialization failed", ex);
        ex.printStackTrace();
    } catch (Error error) {
        RecordLog.warn("[InitExecutor] ERROR: Initialization failed with
fatal error", error);
        error.printStackTrace();
    }
}
```

7. w.func.init()方法会执行每个InitFunc实现类的init()方法,其中有一个实现类HeartbeatSenderInitFunc完成客户端服务心跳发送

```
/**
 * 心跳信号发送器的全局初始化类
 *
 * @author Eric Zhao
 */
@InitOrder(-1)
public class HeartbeatSenderInitFunc implements InitFunc {

    private ScheduledExecutorService pool = null;

    private void initSchedulerIfNeeded() {
        if (pool == null) {
            pool = new ScheduledThreadPoolExecutor(2,
                new NamedThreadFactory("sentinel-heartbeat-send-task",
true),
                new DiscardOldestPolicy());
        }
    }

    @Override
    public void init() {
        HeartbeatSender sender =
HeartbeatSenderProvider.getHeartbeatSender();
        if (sender == null) {
            RecordLog.warn("[HeartbeatSenderInitFunc] WARN: No
HeartbeatSender loaded");
            return;
        }

        initSchedulerIfNeeded();
        //设置心跳任务发送间隔时间 默认10s发送一次
        long interval = retrieveInterval(sender);
        setIntervalIfNotExists(interval);
        //启动心跳任务
        scheduleHeartbeatTask(sender, interval);
    }
```

```
    private boolean isValidHeartbeatInterval(Long interval) {
        return interval != null && interval > 0;
    }

    private void setIntervalIfNotExists(long interval) {
        SentinelConfig.setConfig(TransportConfig.HEARTBEAT_INTERVAL_MS,
String.valueOf(interval));
    }

    long retrieveInterval(/*@NonNull*/ HeartbeatSender sender) {
        Long intervalInConfig = TransportConfig.getHeartbeatIntervalMs();
        if (isValidHeartbeatInterval(intervalInConfig)) {
            RecordLog.info("[HeartbeatSenderInitFunc] Using heartbeat
interval "
                + "in Sentinel config property: " + intervalInConfig);
            return intervalInConfig;
        } else {
            long senderInterval = sender.intervalMs();
            RecordLog.info("[HeartbeatSenderInit] Heartbeat interval not
configured in "
                + "config property or invalid, using sender default: " +
senderInterval);
            return senderInterval;
        }
    }

    private void scheduleHeartbeatTask(/*@NonNull*/ final HeartbeatSender
sender, /*@Valid*/ long interval) {
        pool.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                try {
                    //发送心跳
                    sender.sendHeartbeat();
                } catch (Throwable e) {
                    RecordLog.warn("[HeartbeatSender] Send heartbeat error",
e);
                }
            }
        }, 5000, interval, TimeUnit.MILLISECONDS);
        RecordLog.info("[HeartbeatSenderInit] HeartbeatSender started: "
            + sender.getClass().getCanonicalName());
    }
}
```

8. sender.sendHeartbeat();方法

```
public boolean sendHeartbeat() throws Exception {
    if (TransportConfig.getRuntimePort() <= 0) {
        RecordLog.info("[SimpleHttpHeartbeatSender] Runtime port not
initialized, won't send heartbeat");
        return false;
    }
    // 获取Socket连接地址
    InetSocketAddress addr = getAvailableAddress();
    if (addr == null) {
```

```
        return false;
    }
    // 封装SimpleHttpRequest对象，发送路径为/registry/machine
    SimpleHttpRequest request = new SimpleHttpRequest(addr, HEARTBEAT_PATH);
    // 设置请求参数
    request.setParams(heartBeat.generateCurrentMessage());
    try {
        // 发送
        SimpleHttpResponse response = httpClient.post(request);
        if (response.getStatusCode() == OK_STATUS) {
            return true;
        }
    } catch (Exception e) {
        RecordLog.warn("[SimpleHttpHeartbeatSender] Failed to send heartbeat
to " + addr + " : ", e);
    }
    return false;
}
```

9. 通过发送/registry/machine最终会到达sentinel服务的MachineRegistryController的receiveHeartBeat方法

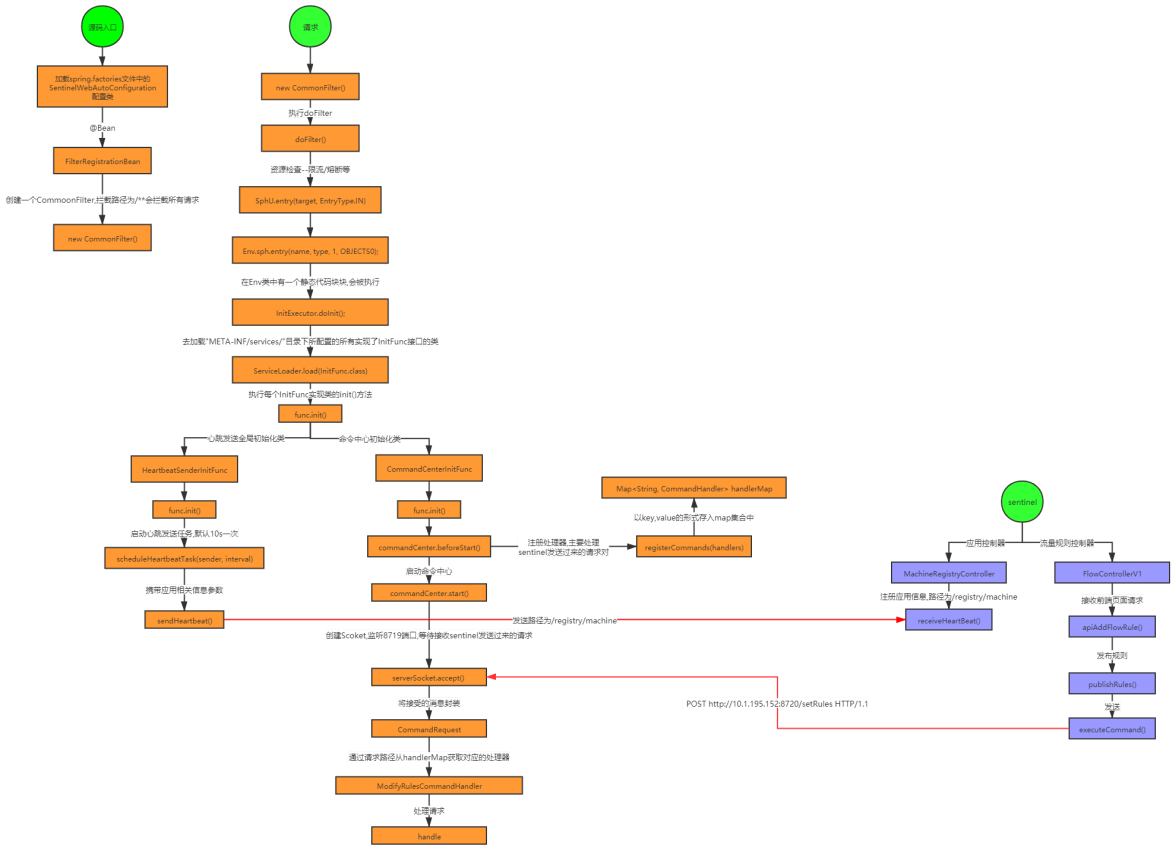
```
/**
 * 注册应用处理器
 */

@Controller
@RequestMapping(value = "/registry", produces =
MediaType.APPLICATION_JSON_VALUE)
public class MachineRegistryController {

    private final Logger logger =
LoggerFactory.getLogger(MachineRegistryController.class);

    @Autowired
    private AppManagement appManagement;

    /**
     * 注册应用信息
     */
    @ResponseBody
    @RequestMapping("/machine")
    public Result<?> receiveHeartBeat(String app, @RequestParam(value =
"app_type", required = false, defaultValue = "0") Integer appType, Long
version, String v, String hostname, String ip, Integer port) {
        if (app == null) {
            app = MachineDiscovery.UNKNOWN_APP_NAME;
        }
        if (ip == null) {
            return Result.ofFail(-1, "ip can't be null");
        }
        if (port == null) {
            return Result.ofFail(-1, "port can't be null");
        }
        if (port == -1) {
            logger.info("Receive heartbeat from " + ip + " but port not set
yet");
```

```
            return Result.ofFail(-1, "your port not set yet");
        }
        String sentinelVersion = StringUtil.isEmpty(v) ? "unknown" : v;
        version = version == null ? System.currentTimeMillis() : version;
        try {
            MachineInfo machineInfo = new MachineInfo();
            machineInfo.setApp(app);
            machineInfo.setAppType(appType);
            machineInfo.setHostname(hostname);
            machineInfo.setIp(ip);
            machineInfo.setPort(port);
            machineInfo.setHeartbeatVersion(version);
            machineInfo.setLastHeartbeat(System.currentTimeMillis());
            machineInfo.setVersion(sentinelVersion);
            // 将接受到的应用信息添加到应用程序管理appManagement
            appManagement.addMachine(machineInfo);
            return Result.ofSuccessMsg("success");
        } catch (Exception e) {
            logger.error("Receive heartbeat error", e);
            return Result.ofFail(-1, e.getMessage());
        }
    }
}
```

2.2 客户端请求处理

在学习sentinel中,我们知道一些数据存储,限流规则等都是在客户端存储的 ,那么客户端是怎么处理sentinel发送过来的请求呢?

2.2.1 客户端请求处理流程分析



2.2.2 主要源码跟踪

1. w.func.init()方法会执行每个InitFunc实现类的init()方法,其中有一个实现类CommandCenterInitFunc完成sentinel服务端发送过来的请求相关操作

```
/**
 * 命令中心初始化类
 * @author Eric Zhao
 */
@InitOrder(-1)
public class CommandCenterInitFunc implements InitFunc {

    @Override
    public void init() throws Exception {
        CommandCenter commandCenter =
CommandCenterProvider.getCommandCenter();

        if (commandCenter == null) {
            RecordLog.warn("[CommandCenterInitFunc] Cannot resolve
CommandCenter");
            return;
        }
        // 注册处理器
        commandCenter.beforeStart();
        // 启动命令中心
        commandCenter.start();
        RecordLog.info("[CommandCenterInit] Starting command center: "
                + commandCenter.getClass().getCanonicalName());
    }
}
```

2. commandCenter.beforeStart()注册处理器,会将所有的处理器进行注册,以key-value的形式存入handlerMap中

```
private static final Map<String, CommandHandler> handlerMap = new
ConcurrentHashMap<String, CommandHandler>();

public void beforeStart() throws Exception {
        // 注册处理器
        Map<String, CommandHandler> handlers =
CommandHandlerProvider.getInstance().namedHandlers();
        registerCommands(handlers);
    }
    public static void registerCommands(Map<String, CommandHandler>
handlerMap) {
        if (handlerMap != null) {
            for (Entry<String, CommandHandler> e : handlerMap.entrySet()) {
                registerCommand(e.getKey(), e.getValue());
            }
        }
    }
@SuppressWarnings("rawtypes")
public static void registerCommand(String commandName, CommandHandler
handler) {
    if (StringUtil.isEmpty(commandName)) {
        return;
```

```
    }

    if (handlerMap.containsKey(commandName)) {
        CommandCenterLog.warn("Register failed (duplicate command): " +
commandName);
        return;
    }

    handlerMap.put(commandName, handler);
}
```

3. commandCenter.start();启动命令中心

```
@Override
public void start() throws Exception {
    int nThreads = Runtime.getRuntime().availableProcessors();
    this.bizExecutor = new ThreadPoolExecutor(nThreads, nThreads, 0L,
TimeUnit.MILLISECONDS,
        new ArrayBlockingQueue<Runnable>(10),
        new NamedThreadFactory("sentinel-command-center-service-executor"),
        new RejectedExecutionHandler() {
            @Override
            public void rejectedExecution(Runnable r, ThreadPoolExecutor
executor) {
                CommandCenterLog.info("EventTask rejected");
                throw new RejectedExecutionException();
            }
        });

    Runnable serverInitTask = new Runnable() {
        int port;

        {
            try {
                //从配置文件中获取端口,如果没有配置设置默认端口8719
                port = Integer.parseInt(TransportConfig.getPort());
            } catch (Exception e) {
                port = DEFAULT_PORT;
            }
        }

        @Override
        public void run() {
            boolean success = false;
            // 获取可用的端口用以创建一个ServerSocket
            ServerSocket serverSocket = getServerSocketFromBasePort(port);

            if (serverSocket != null) {
                CommandCenterLog.info("[CommandCenter] Begin listening at
port " + serverSocket.getLocalPort());
                socketReference = serverSocket;
                // 在主线程中启动ServerThread用以接收socket请求
                executor.submit(new ServerThread(serverSocket));
                success = true;
                port = serverSocket.getLocalPort();
            } else {
```

```
                CommandCenterLog.info("[CommandCenter] chooses port fail,
http command center will not work");
            }

            if (!success) {
                port = PORT_UNINITIALIZED;
            }

            TransportConfig.setRuntimePort(port);
            executor.shutdown();
        }

    };

    new Thread(serverInitTask).start();
}


private static ServerSocket getServerSocketFromBasePort(int basePort) {
    int tryCount = 0;
    while (true) {
        try {
            //如果发现端口占用情况,则尝试3次,每次端口号加1
            ServerSocket server = new ServerSocket(basePort + tryCount / 3,
100);
            server.setReuseAddress(true);
            return server;
        } catch (IOException e) {
            tryCount++;
            try {
                TimeUnit.MILLISECONDS.sleep(30);
            } catch (InterruptedException e1) {
                break;
            }
        }
    }
    return null;
}


public static Set<String> getCommands() {
    return handlerMap.keySet();
}

class ServerThread extends Thread {

    private ServerSocket serverSocket;

    ServerThread(ServerSocket s) {
        this.serverSocket = s;
        setName("sentinel-courier-server-accept-thread");
    }

    @Override
    public void run() {
        while (true) {
            Socket socket = null;
            try {
```

```
                //Socket监听
                socket = this.serverSocket.accept();
                setSocketSoTimeout(socket);
                // 将接收到的socket封装到HttpEventTask中由业务线程去处理
                HttpEventTask eventTask = new HttpEventTask(socket);
                bizExecutor.submit(eventTask);
            } catch (Exception e) {
                CommandCenterLog.info("Server error", e);
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (Exception e1) {
                        CommandCenterLog.info("Error when closing an opened
socket", e1);
                    }
                }
                try {
                    // In case of infinite log.
                    Thread.sleep(10);
                } catch (InterruptedException e1) {
                    // Indicates the task should stop.
                    break;
                }
            }
        }
    }
}
```

4. HttpEventTask类处理sentinel发送过来的请求信息

```
@Override
public void run() {
    if (socket == null) {
        return;
    }

    BufferedReader in = null;
    PrintWriter printWriter = null;
    try {
        long start = System.currentTimeMillis();
        in = new BufferedReader(new
InputStreamReader(socket.getInputStream(), SentinelConfig.charset()));
        OutputStream outputStream = socket.getOutputStream();

        printWriter = new PrintWriter(
            new OutputStreamWriter(outputStream,
Charset.forName(SentinelConfig.charset())));
        //读取消息内容
        String line = in.readLine();
        CommandCenterLog.info("[SimpleHttpCommandCenter] socket income: " +
line
            + "," + socket.getInetAddress());
        //封装CommandRequest对象
        CommandRequest request = parseRequest(line);

        if (line.length() > 4 && StringUtil.equalsIgnoreCase("POST",
line.substring(0, 4))) {
```

```
            // Deal with post method
            // Now simple-http only support form-encoded post request.
            String bodyLine = null;
            boolean bodyNext = false;
            boolean supported = false;
            int maxLength = 8192;
            while (true) {
                // Body processing
                if (bodyNext) {
                    if (!supported) {
                        break;
                    }
                    char[] bodyBytes = new char[maxLength];
                    int read = in.read(bodyBytes);
                    String postData = new String(bodyBytes, 0, read);
                    parseParams(postData, request);
                    break;
                }

                bodyLine = in.readLine();
                if (bodyLine == null) {
                    break;
                }
                // Body seperator
                if (StringUtil.isEmpty(bodyLine)) {
                    bodyNext = true;
                    continue;
                }
                // Header processing
                int index = bodyLine.indexOf(":");
                if (index < 1) {
                    continue;
                }
                String headerName = bodyLine.substring(0, index);
                String header = bodyLine.substring(index + 1).trim();
                if (StringUtil.equalsIgnoreCase("content-type", headerName))
{
                    if (StringUtil.equals("application/x-www-form-
urlencoded", header)) {
                        supported = true;
                    } else {
                        // not support request
                        break;
                    }
                } else if (StringUtil.equalsIgnoreCase("content-length",
headerName)) {
                    try {
                        int len = new Integer(header);
                        if (len > 0) {
                            maxLength = len;
                        }
                    } catch (Exception e) {
                    }
                }
            }
        }

        // 验证目标命令是否合法
```

```
        String commandName = HttpCommandUtils.getTarget(request);
        if (StringUtil.isBlank(commandName)) {
            badRequest(printWriter, "Invalid command");
            return;
        }

        // 找到匹配的命令处理程序。
        CommandHandler<?> commandHandler =
SimpleHttpCommandCenter.getHandler(commandName);
        if (commandHandler != null) {
            //执行处理方法
            CommandResponse<?> response = commandHandler.handle(request);
            handleResponse(response, printWriter, outputStream);
        } else {
            // No matching command handler.
            badRequest(printWriter, "Unknown command `" + commandName +
'`');
        }
        printWriter.flush();

        long cost = System.currentTimeMillis() - start;
        CommandCenterLog.info("[SimpleHttpCommandCenter] Deal a socket task:
" + line
            + ", address: " + socket.getInetAddress() + ", time cost: " +
cost + " ms");
    } catch (Throwable e) {
        CommandCenterLog.warn("[SimpleHttpCommandCenter] CommandCenter
error", e);
        try {
            if (printWriter != null) {
                String errorMessage = SERVER_ERROR_MESSAGE;
                if (!writtenHead) {
                    internalError(printWriter, errorMessage);
                } else {
                    printWriter.println(errorMessage);
                }
                printWriter.flush();
            }
        } catch (Exception e1) {
            CommandCenterLog.warn("[SimpleHttpCommandCenter] Close server
socket failed", e);
        }
    } finally {
        closeResource(in);
        closeResource(printWriter);
        closeResource(socket);
    }
}
```

5. commandHandler.handle(request)处理请求,例如sentinel发送过来的是/setRules,则调用ModifyRulesCommandHandler

```
@CommandMapping(name = "setRules", desc = "modify the rules, accept param:
type={ruleType}&data={ruleJson}")
public class ModifyRulesCommandHandler implements CommandHandler<String> {

    @Override
```

```java
    public CommandResponse<String> handle(CommandRequest request) {
        //获取规则类型
        String type = request.getParam("type");
        //获取参数数据
        String data = request.getParam("data");
        if (StringUtil.isNotEmpty(data)) {
            try {
                data = URLDecoder.decode(data, "utf-8");
            } catch (Exception e) {
                RecordLog.info("Decode rule data error", e);
                return CommandResponse.ofFailure(e, "decode rule data
error");
            }
        }

        RecordLog.info(String.format("Receiving rule change (type: %s): %s",
type, data));

        String result = "success";

        if (FLOW_RULE_TYPE.equalsIgnoreCase(type)) {//限流
            List<FlowRule> flowRules = JSONArray.parseArray(data,
FlowRule.class);
            FlowRuleManager.loadRules(flowRules);
            if (!writeToDataSource(getFlowDataSource(), flowRules)) {
                result = WRITE_DS_FAILURE_MSG;
            }
            return CommandResponse.ofSuccess(result);
        } else if (AUTHORITY_RULE_TYPE.equalsIgnoreCase(type)) {//授权
            List<AuthorityRule> rules = JSONArray.parseArray(data,
AuthorityRule.class);
            AuthorityRuleManager.loadRules(rules);
            if (!writeToDataSource(getAuthorityDataSource(), rules)) {
                result = WRITE_DS_FAILURE_MSG;
            }
            return CommandResponse.ofSuccess(result);
        } else if (DEGRADE_RULE_TYPE.equalsIgnoreCase(type)) {//熔断
            List<DegradeRule> rules = JSONArray.parseArray(data,
DegradeRule.class);
            DegradeRuleManager.loadRules(rules);
            if (!writeToDataSource(getDegradeDataSource(), rules)) {
                result = WRITE_DS_FAILURE_MSG;
            }
            return CommandResponse.ofSuccess(result);
        } else if (SYSTEM_RULE_TYPE.equalsIgnoreCase(type)) {//系统规则
            List<SystemRule> rules = JSONArray.parseArray(data,
SystemRule.class);
            SystemRuleManager.loadRules(rules);
            if (!writeToDataSource(getSystemSource(), rules)) {
                result = WRITE_DS_FAILURE_MSG;
            }
            return CommandResponse.ofSuccess(result);
        }
        return CommandResponse.ofFailure(new
IllegalArgumentException("invalid type"));
    }
```

```
}
```

6. 假如type是限流则调用FlowRuleManager.loadRules(flowRules)去加载限流规则

```
public boolean updateValue(T newValue) {
    if (isEqual(value, newValue)) {
        return false;
    }
    RecordLog.info("[DynamicSentinelProperty] Config will be updated to: " +
newValue);

    value = newValue;
    for (PropertyListener<T> listener : listeners) {
        listener.configUpdate(newValue);
    }
    return true;
}
```

7. FlowPropertyListener类的configUpdate方法

```
public void configUpdate(List<FlowRule> value) {
    //构建限流规则集合
    Map<String, List<FlowRule>> rules =
FlowRuleUtil.buildFlowRuleMap(value);
    if (rules != null) {
        flowRules.clear();
        //将限流规则集合放入flowRules中以key-value的形式存入
        flowRules.putAll(rules);
    }
    RecordLog.info("[FlowRuleManager] Flow rules received: " + flowRules);
}
```

8. buildFlowRuleMap方法

```
public static <K> Map<K, List<FlowRule>> buildFlowRuleMap(List<FlowRule>
list, Function<FlowRule, K> groupFunction,

 Predicate<FlowRule> filter, boolean shouldSort) {
    Map<K, List<FlowRule>> newRuleMap = new ConcurrentHashMap<>();
    if (list == null || list.isEmpty()) {
        return newRuleMap;
    }
    Map<K, Set<FlowRule>> tmpMap = new ConcurrentHashMap<>();
    //遍历限流规则
    for (FlowRule rule : list) {
        if (!isValidRule(rule)) {
            RecordLog.warn("[FlowRuleManager] Ignoring invalid flow rule
when loading new flow rules: " + rule);
            continue;
        }
        if (filter != null && !filter.test(rule)) {
            continue;
        }
        if (StringUtil.isBlank(rule.getLimitApp())) {
            rule.setLimitApp(RuleConstant.LIMIT_APP_DEFAULT);
        }
```

```
        //根据流量规则生成不同的控制器
        TrafficShapingController rater = generateRater(rule);
        rule.setRater(rater);

        K key = groupFunction.apply(rule);
        if (key == null) {
            continue;
        }
        Set<FlowRule> flowRules = tmpMap.get(key);

        if (flowRules == null) {
            // Use hash set here to remove duplicate rules.
            flowRules = new HashSet<>();
            tmpMap.put(key, flowRules);
        }

        flowRules.add(rule);
    }
    Comparator<FlowRule> comparator = new FlowRuleComparator();
    for (Entry<K, Set<FlowRule>> entries : tmpMap.entrySet()) {
        List<FlowRule> rules = new ArrayList<>(entries.getValue());
        if (shouldSort) {
            // Sort the rules.
            Collections.sort(rules, comparator);
        }
        newRuleMap.put(entries.getKey(), rules);
    }

    return newRuleMap;
}
```

9. generateRater(rule)方法

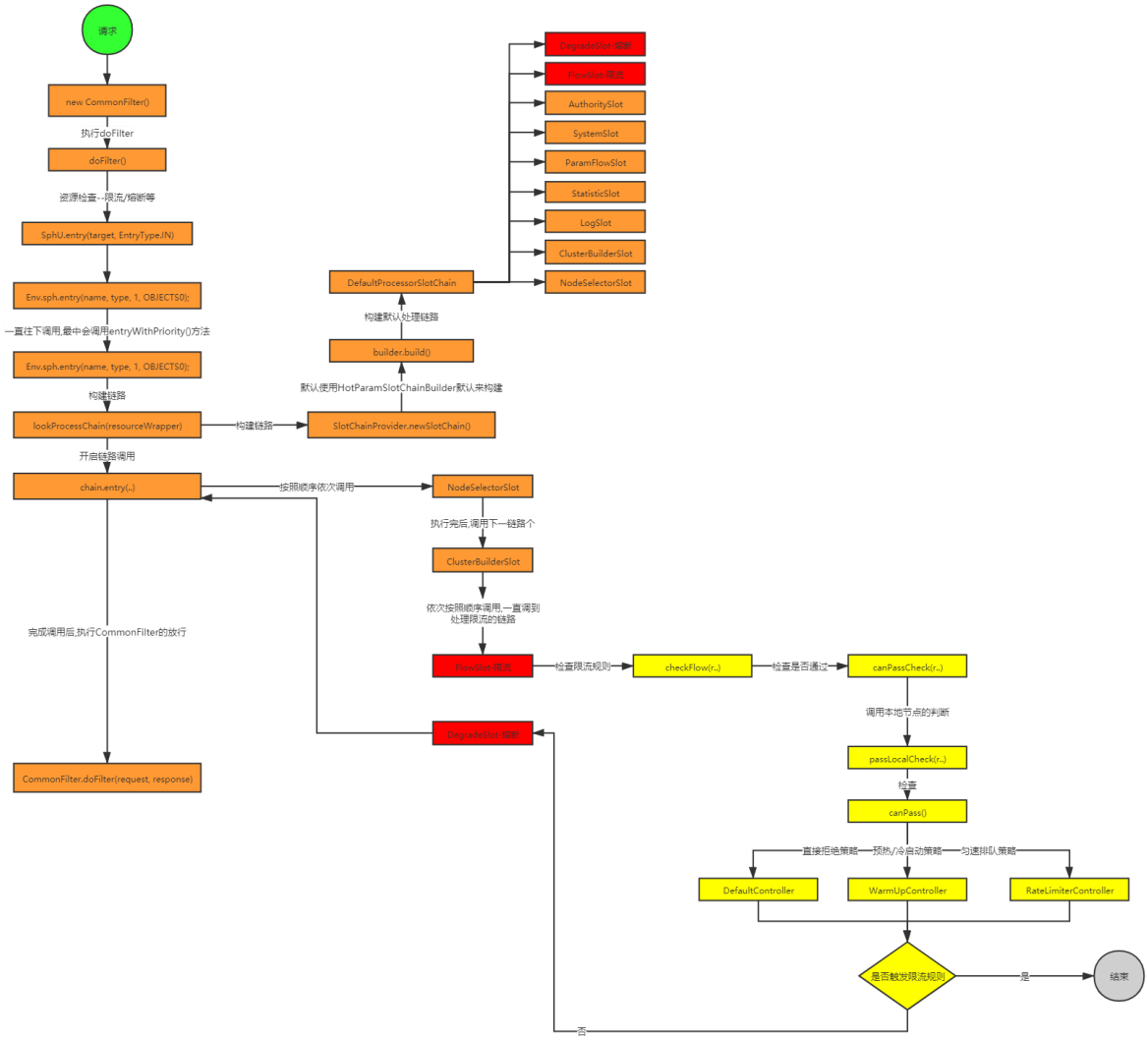
```
private static TrafficShapingController generateRater(/*@Valid*/ FlowRule
rule) {
    //如果限流类型是QPS,则根据不同的流控规则生成不同的处理器,这个地方使用的是策略模式
    if (rule.getGrade() == RuleConstant.FLOW_GRADE_QPS) {
        switch (rule.getControlBehavior()) {
            case RuleConstant.CONTROL_BEHAVIOR_WARM_UP:
                return new WarmUpController(rule.getCount(),
rule.getWarmUpPeriodSec(),
                    ColdFactorProperty.coldFactor);//流控规则为预热策略
            case RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER:
                return new
RateLimiterController(rule.getMaxQueueingTimeMs(), rule.getCount());//流控规则
为匀速排队策略
            case RuleConstant.CONTROL_BEHAVIOR_WARM_UP_RATE_LIMITER:
                return new WarmUpRateLimiterController(rule.getCount(),
rule.getWarmUpPeriodSec(),
                    rule.getMaxQueueingTimeMs(),
ColdFactorProperty.coldFactor);//流控规则为预热+匀速排队策略
            case RuleConstant.CONTROL_BEHAVIOR_DEFAULT:
            default:
                // Default mode or unknown mode: default traffic shaping
controller (fast-reject).
        }
    }
```

```
    //默认是直接拒接策略
    return new DefaultController(rule.getCount(), rule.getGrade());
}
```

2.3 sentinel限流

在刚才源码中我们知道客户端是如何处理sentinel发送过来的限流,那么我们看看客户端是如何完成限流操作的

2.3.1 限流流程分析



2.3.2 主要源码跟踪

1. SphU.entry(target, EntryType.IN)代码完成限流/熔断等操作

```
public static Entry entry(String name, EntryType type) throws BlockException
{
    // Env类中有静态方法会被调用
    return Env.sph.entry(name, type, 1, OBJECTS0);
}
```

2. 经过调用最终会调用CtSph的entryWithPriority方法

```
private Entry entryWithPriority(ResourceWrapper resourceWrapper, int count,
boolean prioritized, Object... args)
    throws BlockException {
```

```
    Context context = ContextUtil.getContext();
    if (context instanceof NullContext) {
        // The {@link NullContext} indicates that the amount of context has
exceeded the threshold,
        // so here init the entry only. No rule checking will be done.
        return new CtEntry(resourceWrapper, null, context);
    }

    if (context == null) {
        // Using default context.
        context = MyContextUtil.myEnter(Constants.CONTEXT_DEFAULT_NAME, "",
resourceWrapper.getType());
    }

    // Global switch is close, no rule checking will do.
    if (!Constants.ON) {
        return new CtEntry(resourceWrapper, null, context);
    }
    //核心方法--构建链路
    ProcessorSlot<Object> chain = lookProcessChain(resourceWrapper);

    /*
     * Means amount of resources (slot chain) exceeds {@link
Constants.MAX_SLOT_CHAIN_SIZE},
     * so no rule checking will be done.
     */
    if (chain == null) {
        return new CtEntry(resourceWrapper, null, context);
    }

    Entry e = new CtEntry(resourceWrapper, chain, context);
    try {
        // 开始进行链路调用
        chain.entry(context, resourceWrapper, null, count, prioritized,
args);
    } catch (BlockException e1) {
        e.exit(count, args);
        throw e1;
    } catch (Throwable e1) {
        // This should not happen, unless there are errors existing in
Sentinel internal.
        RecordLog.info("Sentinel unexpected exception", e1);
    }
    return e;
}
```

3. lookProcessChain方法构建链路

```
ProcessorSlot<Object> lookProcessChain(ResourceWrapper resourceWrapper) {
    ProcessorSlotChain chain = chainMap.get(resourceWrapper);
    if (chain == null) {
        synchronized (LOCK) {
            chain = chainMap.get(resourceWrapper);
            if (chain == null) {
                // Entry size limit.
                if (chainMap.size() >= Constants.MAX_SLOT_CHAIN_SIZE) {
                    return null;
```

```
                }
                // 构建链路
                chain = SlotChainProvider.newSlotChain();
                Map<ResourceWrapper, ProcessorSlotChain> newMap = new
HashMap<ResourceWrapper, ProcessorSlotChain>(
                    chainMap.size() + 1);
                newMap.putAll(chainMap);
                newMap.put(resourceWrapper, chain);
                chainMap = newMap;
            }
        }
    }
    return chain;
}
```

SlotChainProvider.newSlotChain()

```
public static ProcessorSlotChain newSlotChain() {
    if (builder != null) {
        return builder.build();
    }
    //解析链路构造器-默认会使用HotParamSlotChainBuilder热点参数链路构造器
    resolveSlotChainBuilder();

    if (builder == null) {
        RecordLog.warn("[SlotChainProvider] Wrong state when resolving slot
chain builder, using default");
        builder = new DefaultSlotChainBuilder();
    }
    //构建
    return builder.build();
}
```

builder.build()

```
public ProcessorSlotChain build() {
    ProcessorSlotChain chain = new DefaultProcessorSlotChain();
    //负责收集资源的路径，并将这些资源的调用路径，以树状结构存储起来，用于根据调用路径来限
流降级
    chain.addLast(new NodeSelectorSlot());
    //用于构建资源的 ClusterNode 以及调用来源节点。ClusterNode 保持某个资源运行统计信
息（响应时间、QPS、block 数目、线程数、异常数等）以及调用来源统计信息列表
    chain.addLast(new ClusterBuilderSlot());
    //该类对链路的传递不做处理，只有在抛出BlockException的时候，向上层层传递的过程中，会
通过该类来输入一些日志信息
    chain.addLast(new LogSlot());
    //用于记录、统计不同纬度的运行指标监控信息
    chain.addLast(new StatisticSlot());
    //用于频繁（“热点”）参数进行流量控制。
    chain.addLast(new ParamFlowSlot());
    //根据配置的黑白名单和调用来源信息，来做黑白名单控制
    chain.addLast(new SystemSlot());
    //会根据对于当前系统的整体情况，对入口资源的调用进行动态调配。其原理是让入口的流量和当
前系统的预计容量达到一个动态平衡。
    chain.addLast(new AuthoritySlot());
    //主要完成限流
```

```
    chain.addLast(new FlowSlot());
    //熔断  主要针对资源的平均响应时间（RT）以及异常比率，来决定资源是否在接下来的时间被自动熔断掉。
    chain.addLast(new DegradeSlot());

    return chain;
}
```

4. chain.entry方法开启链路调用,会对链路中每个进行逐一调用,一直到到FlowSlot

```
public void entry(Context context, ResourceWrapper resourceWrapper, DefaultNode node, int count,
                  boolean prioritized, Object... args) throws Throwable {
    // 检查限流规则
    checkFlow(resourceWrapper, context, node, count, prioritized);
    // 调用下一个
    fireEntry(context, resourceWrapper, node, count, prioritized, args);
}
```

5. checkFlow限流规则检查

```
public void checkFlow(Function<String, Collection<FlowRule>> ruleProvider, ResourceWrapper resource,
                      Context context, DefaultNode node, int count, boolean prioritized) throws BlockException {
    if (ruleProvider == null || resource == null) {
        return;
    }
    // 根据资源名称找到对应的限流规则
    Collection<FlowRule> rules = ruleProvider.apply(resource.getName());
    if (rules != null) {
        for (FlowRule rule : rules) {
            // 遍历规则，依次判断是否通过
            if (!canPassCheck(rule, context, node, count, prioritized)) {
                throw new FlowException(rule.getLimitApp(), rule);
            }
        }
    }
}
```

ruleProvider.apply(resource.getName());

```
private final Function<String, Collection<FlowRule>> ruleProvider = new Function<String, Collection<FlowRule>>() {
    @Override
    public Collection<FlowRule> apply(String resource) {
        // 查找限流规则
        Map<String, List<FlowRule>> flowRules = FlowRuleManager.getFlowRuleMap();
        return flowRules.get(resource);
    }
};
```

6. canPassCheck()判断规则是否通过

```
public boolean canPassCheck(/*@NonNull*/ FlowRule rule, Context context,
DefaultNode node, int acquireCount,
                                                    boolean prioritized) {
    String limitApp = rule.getLimitApp();
    if (limitApp == null) {
        return true;
    }
    //判断是否是集群
    if (rule.isClusterMode()) {
        return passClusterCheck(rule, context, node, acquireCount,
prioritized);
    }
    //不是集群则调用本地检查
    return passLocalCheck(rule, context, node, acquireCount, prioritized);
}
```

```
private static boolean passLocalCheck(FlowRule rule, Context context,
DefaultNode node, int acquireCount,
                                          boolean prioritized) {
    Node selectedNode = selectNodeByRequesterAndStrategy(rule, context,
node);
    if (selectedNode == null) {
        return true;
    }
    //获取规则处理器进行检查
    return rule.getRater().canPass(selectedNode, acquireCount, prioritized);
}
```

7. DefaultController默认拒接策略

```
/**
 * 默认限流控制器（立即拒绝策略）。
 *
 * @author jialiang.linjl
 * @author Eric Zhao
 */
public class DefaultController implements TrafficShapingController {

    private static final int DEFAULT_AVG_USED_TOKENS = 0;

    private double count;
    private int grade;

    public DefaultController(double count, int grade) {
        this.count = count;
        this.grade = grade;
    }

    @Override
    public boolean canPass(Node node, int acquireCount) {
        return canPass(node, acquireCount, false);
    }

    @Override
    public boolean canPass(Node node, int acquireCount, boolean prioritized)
{
```

```
        // 当前已经统计的数
        int curCount = avgUsedTokens(node);
        //如果已经统计的数+请求计数 > 限流数量,则返回false,代表限流
        if (curCount + acquireCount > count) {
            if (prioritized && grade == RuleConstant.FLOW_GRADE_QPS) {
                long currentTime;
                long waitInMs;
                currentTime = TimeUtil.currentTimeMillis();
                waitInMs = node.tryOccupyNext(currentTime, acquireCount,
count);
                if (waitInMs < OccupyTimeoutProperty.getOccupyTimeout()) {
                    node.addWaitingRequest(currentTime + waitInMs,
acquireCount);
                    node.addOccupiedPass(acquireCount);
                    sleep(waitInMs);

                    // PriorityWaitException indicates that the request will
pass after waiting for {@link @waitInMs}.
                    throw new PriorityWaitException(waitInMs);
                }
            }
            return false;
        }
        return true;
    }

    private int avgUsedTokens(Node node) {
        if (node == null) {
            return DEFAULT_AVG_USED_TOKENS;
        }
        // 如果当前是线程数限流，则返回node.curThreadNum()当前线程数
        // 如果是QPS限流，则返回node.passQps()当前已经通过的qps数据
        return grade == RuleConstant.FLOW_GRADE_THREAD ? node.curThreadNum()
: (int)(node.passQps());
    }


}
```