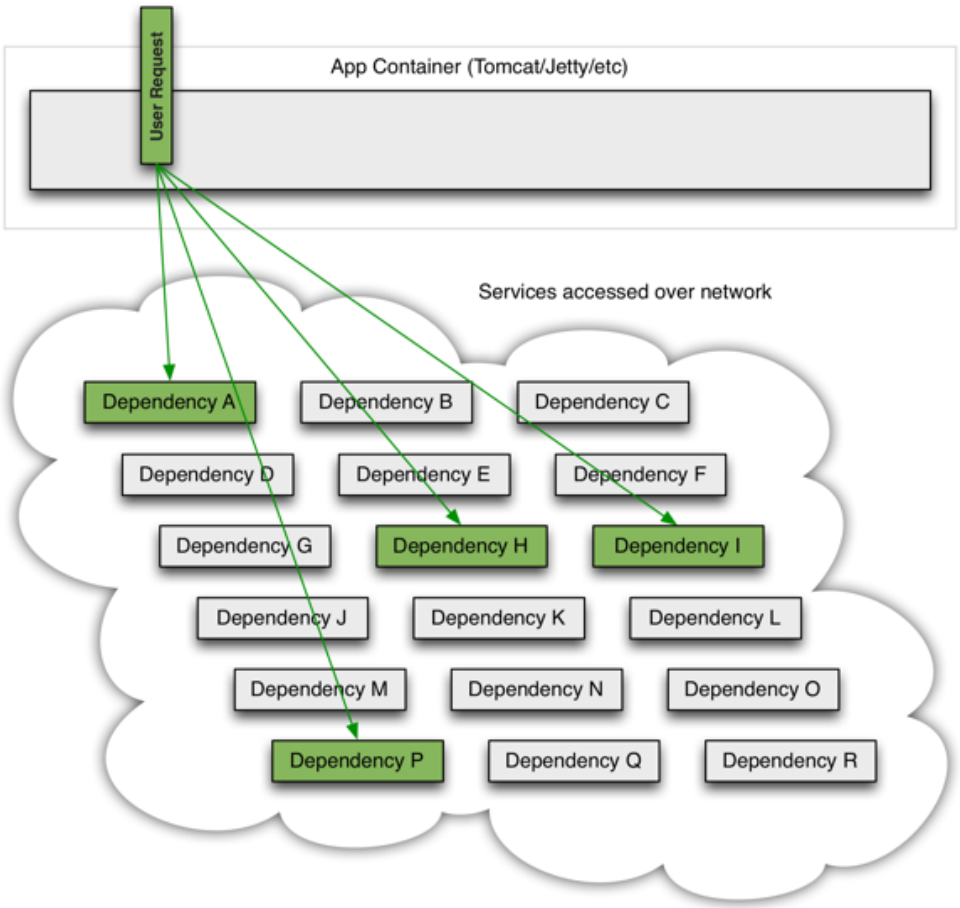
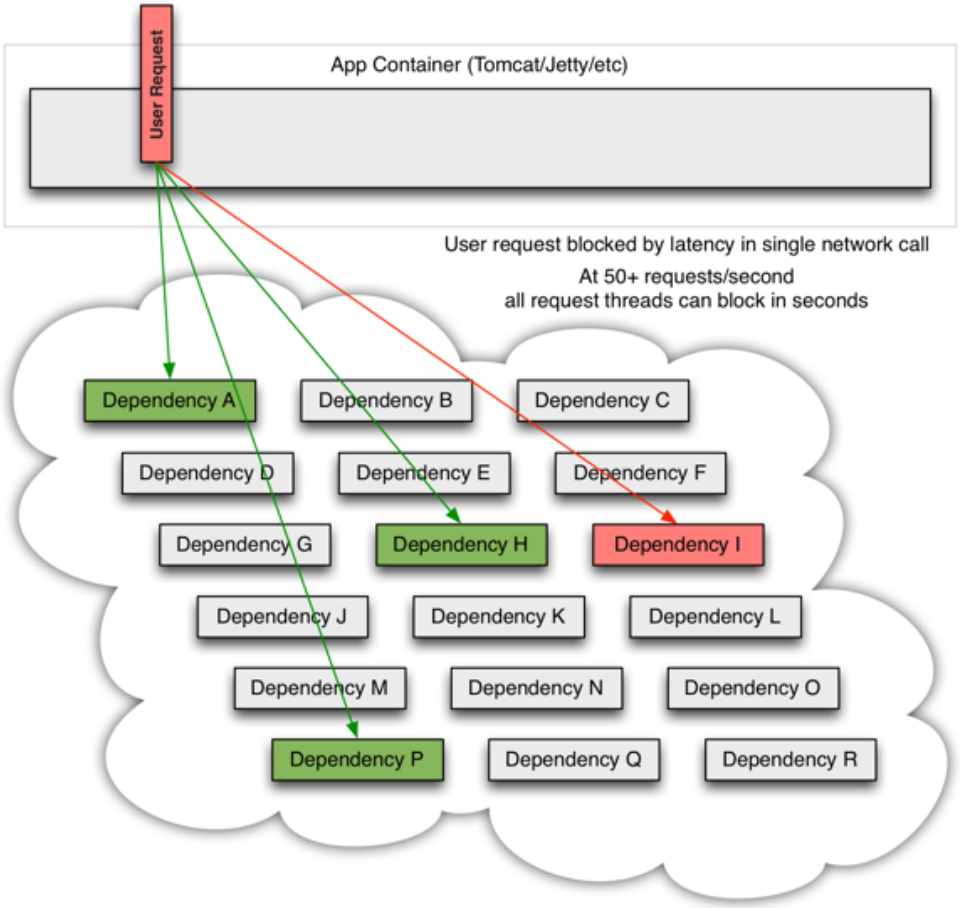
一:为什么需要Hystrix?

在大中型分布式系统中，通常系统很多依赖(HTTP,hession,Netty,Dubbo等)，如下图:

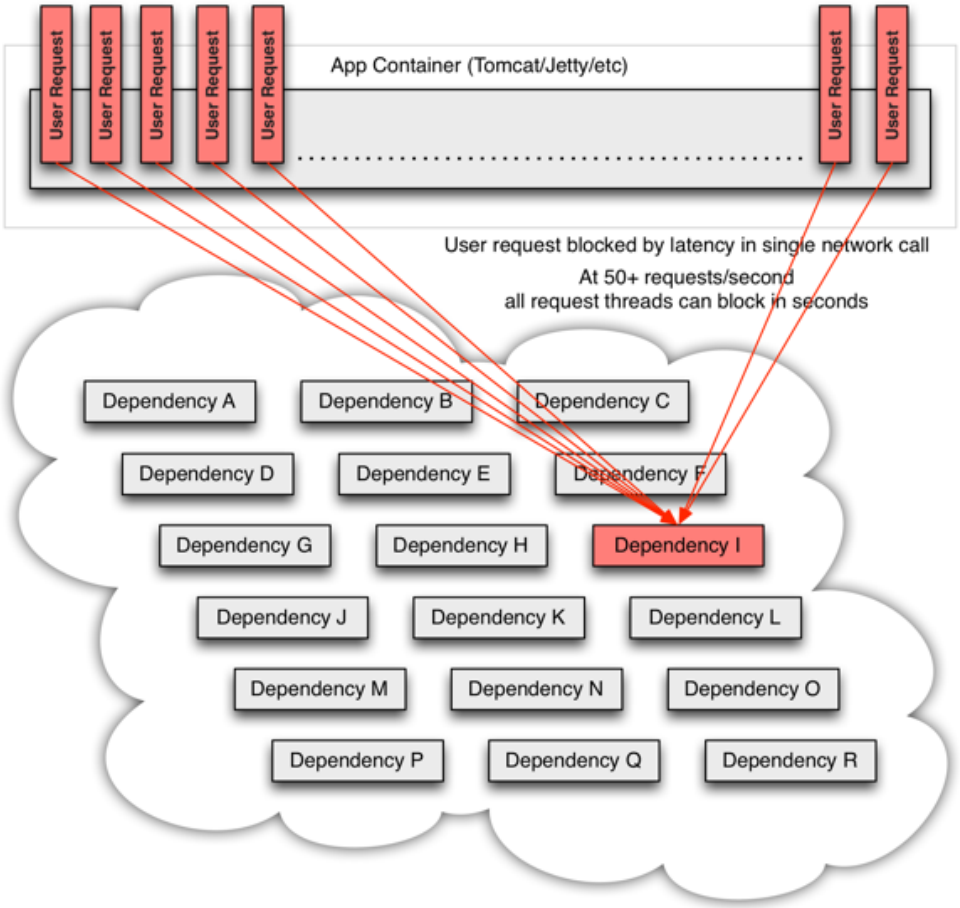


在高并发访问下,这些依赖的稳定性与否对系统的影响非常大,但是依赖有很多不可控问题:如网络连接缓慢，资源繁忙，暂时不可用，服务脱机等.

如下图：QPS为50的依赖 I 出现不可用，但是其他依赖仍然可用.



当依赖I 阻塞时,大多数服务器的线程池就出现阻塞(BLOCK),影响整个线上服务的稳定性.如下图:



在复杂的分布式架构的应用程序有很多的依赖，都会不可避免地在某些时候失败。高并发的依赖失败时如果没有隔离措施，当前应用服务就有被拖垮的风险。

Java代码

1. 例如:一个依赖30个SOA服务的系统,每个服务99.99%可用。
2. 99.99%的30次方 ≈ 99.7%
3. 0.3% 意味着一亿次请求 会有 3,000,00次失败
4. 换算成时间大约每月有2个小时服务不稳定.
5. 随着服务依赖数量的变多，服务不稳定的概率会成指数性提高.

解决问题方案:对依赖做隔离,Hystrix就是处理依赖隔离的框架,同时也是可以帮我们做依赖服务的治理和监控.

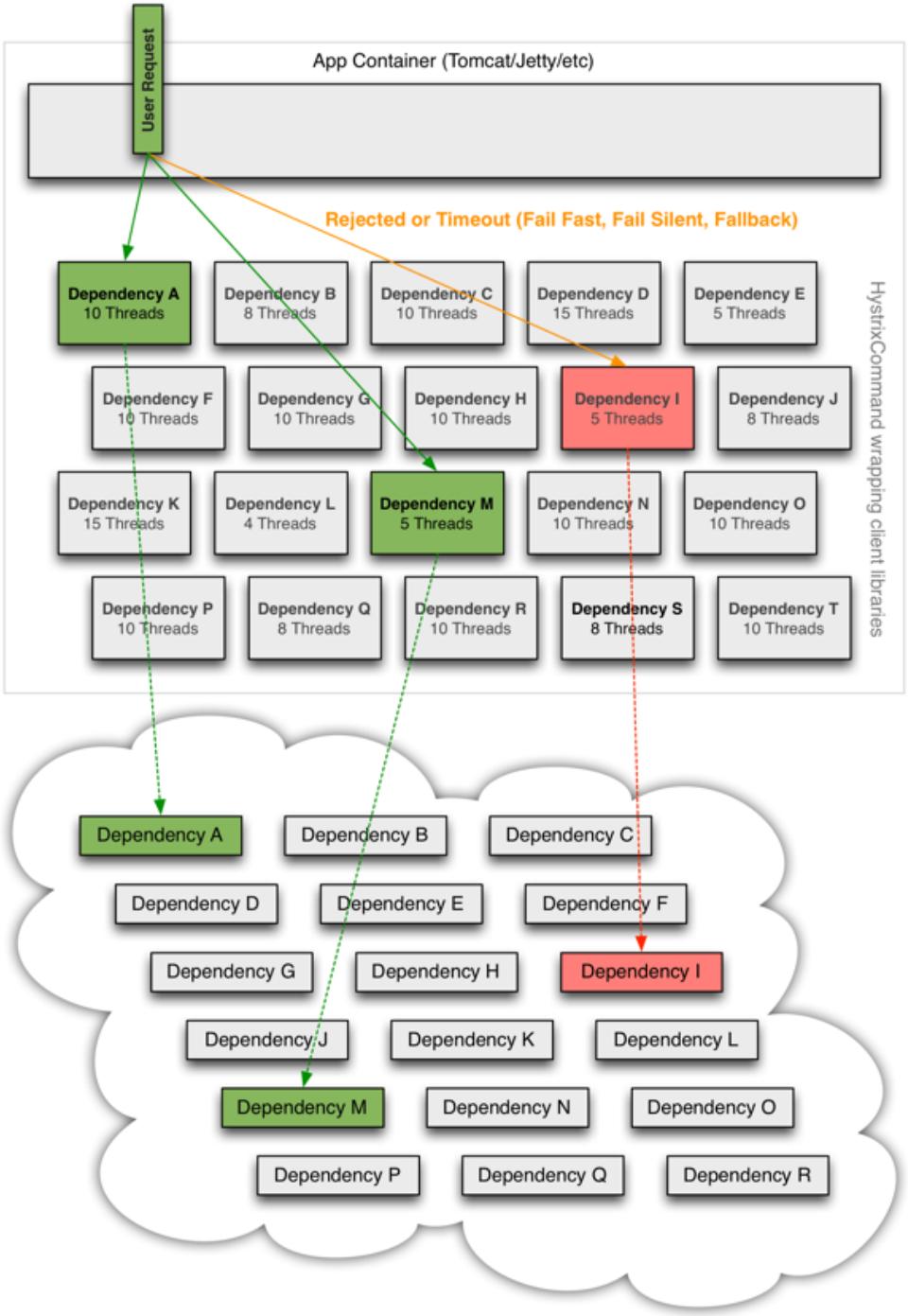
Netflix 公司开发并成功使用Hystrix,使用规模如下:

Java代码

1. The Netflix API processes 10+ billion HystrixCommand executions per day using thread isolation.
2. Each API instance has 40+ thread-pools with 5-20 threads in each (most are set to 10).

二:Hystrix如何解决依赖隔离

1:Hystrix使用命令模式HystrixCommand(Command)包装依赖调用逻辑，每个命令在单独线程中/信号授权下执行。
2:可配置依赖调用超时时间,超时时间一般设为比99.5%平均时间略高即可.当调用超时时，直接返回或执行fallback逻辑。
3:为每个依赖提供一个小的线程池（或信号），如果线程池已满调用将被立即拒绝，默认不采用排队.加速失败判定时间。
4:依赖调用结果分:成功，失败（抛出异常），超时，线程拒绝，短路。 请求失败(异常，拒绝，超时，短路)时执行fallback(降级)逻辑。
5:提供熔断器组件,可以自动运行或手动调用,停止当前依赖一段时间(10秒)，熔断器默认错误率阈值为50%,超过将自动运行。
6:提供近实时依赖的统计和监控
Hystrix依赖的隔离架构,如下图:



三:如何使用Hystrix

1:使用maven引入Hystrix依赖

Html代码

```
<!-- 依赖版本 -->
<hystrix.version>1.3.16</hystrix.version>
<hystrix-metrics-event-stream.version>1.1.2</hystrix-metrics-event-stream.version>

<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>${hystrix.version}</version>
 </dependency>
    <dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-metrics-event-stream</artifactId>
    <version>${hystrix-metrics-event-stream.version}</version>
 </dependency>
<!-- 仓库地址 -->
<repository>
    <id>nexus</id>
    <name>local private nexus</name>
    <url>http://maven.oschina.net/content/groups/public/</url>
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</repository>
```

2:使用命令模式封装依赖逻辑

Java代码

```
public class HelloWorldCommand extends HystrixCommand<String> {
    private final String name;
    public HelloWorldCommand(String name) {
        //最少配置:指定命令组名(CommandGroup)
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }
    @Override
    protected String run() {
        // 依赖逻辑封装在run()方法中
        return "Hello " + name +" thread:" + Thread.currentThread().getName();
    }
```

```
13.     //调用实例
14.     public static void main(String[] args) throws Exception{
15.         //每个Command对象只能调用一次,不可以重复调用,
16.         //重复调用对应异常信息:This instance can only be executed once. Please instantiate a new instance.
17.         HelloWorldCommand helloWorldCommand = new HelloWorldCommand("Synchronous-hystrix");
18.         //使用execute()同步调用代码,效果等同于:helloWorldCommand.queue().get();
19.         String result = helloWorldCommand.execute();
20.         System.out.println("result=" + result);
21.
22.         helloWorldCommand = new HelloWorldCommand("Asynchronous-hystrix");
23.         //异步调用,可自由控制获取结果时机,
24.         Future<String> future = helloWorldCommand.queue();
25.         //get操作不能超过command定义的超时时间,默认:1秒
26.         result = future.get(100, TimeUnit.MILLISECONDS);
27.         System.out.println("result=" + result);
28.         System.out.println("mainThread=" + Thread.currentThread().getName());
29.     }
30.
31. }
32.     //运行结果: run()方法在不同的线程下执行
33.     // result=Hello Synchronous-hystrix thread:hystrix-HelloWorldGroup-1
34.     // result=Hello Asynchronous-hystrix thread:hystrix-HelloWorldGroup-2
35.     // mainThread=main
```

note:异步调用使用 command.queue()get(timeout, TimeUnit.MILLISECONDS);同步调用使用command.execute() 等同于 command.queue().get();

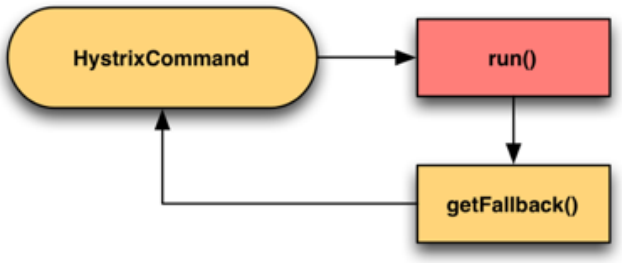
3:注册异步事件回调执行

Java代码

```
1. //注册观察者事件拦截
2. Observable<String> fs = new HelloWorldCommand("World").observe();
3. //注册结果回调事件
4. fs.subscribe(new Action1<String>() {
5.     @Override
6.     public void call(String result) {
7.         //执行结果处理,result 为HelloWorldCommand返回的结果
8.         //用户对结果做二次处理.
```

```
    }
});
//注册完整执行生命周期事件
fs.subscribe(new Observer<String>() {
        @Override
        public void onCompleted() {
            // onNext/onError完成之后最后回调
            System.out.println("execute onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            // 当产生异常时回调
            System.out.println("onError " + e.getMessage());
            e.printStackTrace();
        }
        @Override
        public void onNext(String v) {
            // 获取结果后回调
            System.out.println("onNext: " + v);
        }
    });
/* 运行结果
call execute result=Hello observe-hystrix thread:hystrix-HelloWorldGroup-3
onNext: Hello observe-hystrix thread:hystrix-HelloWorldGroup-3
execute onCompleted
*/
```

4:使用Fallback() 提供降级策略



Java代码

```
//重载HystrixCommand 的getFallback方法实现逻辑
public class HelloWorldCommand extends HystrixCommand<String> {
    private final String name;
    public HelloWorldCommand(String name) {
```

```
5.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Hello
WorldGroup"))
6.              /* 配置依赖超时时间,500毫秒*/
7.
.andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withEx
ecutionIsolationThreadTimeoutInMilliseconds(500)));
8.        this.name = name;
9.    }
10.    @Override
11.    protected String getFallback() {
12.        return "exeucute Falled";
13.    }
14.    @Override
15.    protected String run() throws Exception {
16.        //sleep 1 秒,调用会超时
17.        TimeUnit.MILLISECONDS.sleep(1000);
18.        return "Hello " + name +" thread:" +
Thread.currentThread().getName();
19.    }
20.    public static void main(String[] args) throws Exception{
21.        HelloWorldCommand command = new HelloWorldCommand("test-
Fallback");
22.        String result = command.execute();
23.    }
24. }
25. /* 运行结果:getFallback() 调用运行
26. getFallback executed
27. */
```

NOTE: 除了HystrixBadRequestException异常之外，所有从run()方法抛出的异常都算作失败，并触发降级getFallback()和断路器逻辑。

HystrixBadRequestException用在非法参数或非系统故障异常等不应触发回退逻辑的场景。

5:依赖命名:CommandKey

Java代码

```
1. public HelloWorldCommand(String name) {
2.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Exam
pleGroup"))
3.              /* HystrixCommandKey工厂定义依赖名称 */
```

```
4.
.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld")));
5.         this.name = name;
6.     }
```

NOTE: 每个CommandKey代表一个依赖抽象,相同的依赖要使用相同的CommandKey名称。依赖隔离的根本就是对相同CommandKey的依赖做隔离.

6:依赖分组:CommandGroup

命令分组用于对依赖操作分组,便于统计,汇总等.

Java代码

```
1. //使用HystrixCommandGroupKey工厂定义
2. public HelloWorldCommand(String name) {
3.
Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))
4. }
```

NOTE: CommandGroup是每个命令最少配置的必选参数，在不指定ThreadPoolKey的情况下，字面值用于对不同依赖的线程池/信号区分.

7:线程池/信号:ThreadPoolKey

Java代码

```
1. public HelloWorldCommand(String name) {
2.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExamplеGroup"))
3.
.andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))
4.                 /* 使用HystrixThreadPoolKey工厂定义线程池名称*/
5.
.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));

6.         this.name = name;
7.     }
```

NOTE: 当对同一业务依赖做隔离时使用CommandGroup做区分,但是对同一依赖的不同远程调用如(一个是redis 一个是http),可以使用HystrixThreadPoolKey做隔离区分.

最然在业务上都是相同的组，但是需要在资源上做隔离时，可以使用HystrixThreadPoolKey区分.

8:请求缓存 Request-Cache

Java代码

```
1. public class RequestCacheCommand extends
HystrixCommand<String> {
2.     private final int id;
3.     public RequestCacheCommand( int id) {
4.
super(HystrixCommandGroupKey.Factory.asKey("RequestCacheCommand"))
;
5.         this.id = id;
6.     }
7.     @Override
8.     protected String run() throws Exception {
9.         System.out.println(Thread.currentThread().getName() + " execute
id=" + id);
10.         return "executed=" + id;
11.     }
12.     //重写getCacheKey方法,实现区分不同请求的逻辑
13.     @Override
14.     protected String getCacheKey() {
15.         return String.valueOf(id);
16.     }
17.
18.     public static void main(String[] args){
19.         HystrixRequestContext context =
HystrixRequestContext.initializeContext();
20.         try {
21.             RequestCacheCommand command2a = new
RequestCacheCommand(2);
22.             RequestCacheCommand command2b = new
RequestCacheCommand(2);
23.             Assert.assertTrue(command2a.execute());
24.             //isResponseFromCache判定是否是在缓存中获取结果
25.             Assert.assertFalse(command2a.isResponseFromCache());
26.             Assert.assertTrue(command2b.execute());
27.             Assert.assertTrue(command2b.isResponseFromCache());
28.         } finally {
29.             context.shutdown();
30.         }
31.         context = HystrixRequestContext.initializeContext();
32.         try {
33.             RequestCacheCommand command3b = new
RequestCacheCommand(2);
34.             Assert.assertTrue(command3b.execute());
35.             Assert.assertFalse(command3b.isResponseFromCache());
36.         } finally {
```

```
            context.shutdown();
        }
    }
}
```

NOTE:请求缓存可以让(CommandKey/CommandGroup)相同的情况下,直接共享结果，降低依赖调用次数，在高并发和CacheKey碰撞率高场景下可以提升性能.

Servlet容器中，可以直接实用Filter机制Hystrix请求上下文

Java代码

```
1. public class HystrixRequestContextServletFilter implements Filter {
2.     public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
3.     throws IOException, ServletException {
4.         HystrixRequestContext context =
HystrixRequestContext.initializeContext();
5.         try {
6.             chain.doFilter(request, response);
7.         } finally {
8.             context.shutdown();
9.         }
10.     }
11. }
12. <filter>
13.         <display-name>HystrixRequestContextServletFilter</display-name>
14.         <filter-name>HystrixRequestContextServletFilter</filter-name>
15.         <filter-
class>com.netflix.hystrix.contrib.requestservlet.HystrixRequestContextServl
etFilter</filter-class>
16.     </filter>
17.     <filter-mapping>
18.         <filter-name>HystrixRequestContextServletFilter</filter-name>
19.         <url-pattern>/*</url-pattern>
20.     </filter-mapping>
```

9:信号量隔离:SEMAPHORE

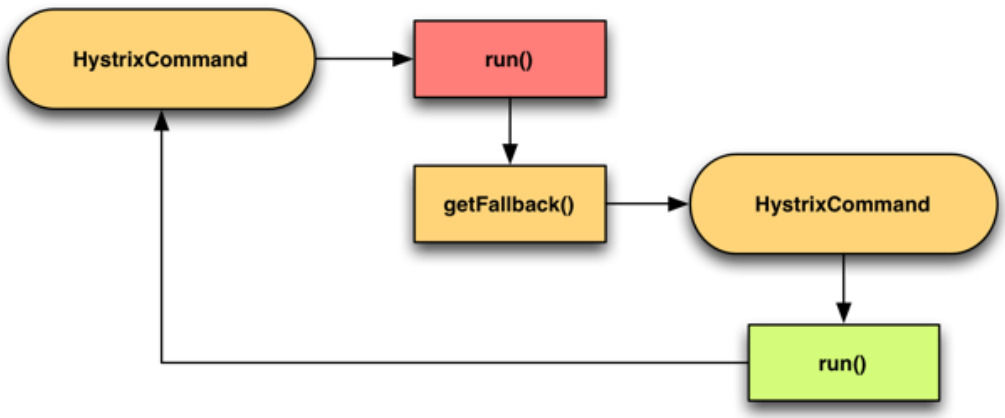
隔离本地代码或可快速返回远程调用(如memcached,redis)可以直接使用信号量隔离,降低线程隔离开销.

Java代码

```
1. public class HelloWorldCommand extends HystrixCommand<String> {
2.     private final String name;
3.     public HelloWorldCommand(String name) {
4.
```

```
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Hello
WorldGroup"))
5.             /* 配置信号量隔离方式,默认采用线程池隔离 */
6.
.andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withEx
ecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrate
gy.SEMAPHORE)));
7.         this.name = name;
8.     }
9.     @Override
10.     protected String run() throws Exception {
11.         return "HystrixThread:" + Thread.currentThread().getName();
12.     }
13.     public static void main(String[] args) throws Exception{
14.         HelloWorldCommand command = new
HelloWorldCommand("semaphore");
15.         String result = command.execute();
16.         System.out.println(result);
17.         System.out.println("MainThread:" +
Thread.currentThread().getName());
18.     }
19. }
20. /** 运行结果
21.  HystrixThread:main
22.  MainThread:main
23. */
```

10:fallback降级逻辑命令嵌套



适用场景:用于fallback逻辑涉及网络访问的情况,如缓存访问。

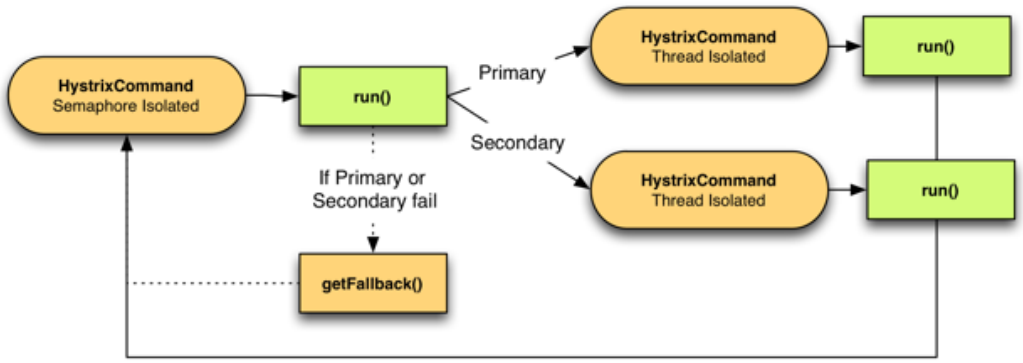
Java代码

```
1. public class CommandWithFallbackViaNetwork extends
HystrixCommand<String> {
2.     private final int id;
3.
4.     protected CommandWithFallbackViaNetwork(int id) {
5.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Remo
teServiceX"))
6.
.andCommandKey(HystrixCommandKey.Factory.asKey("GetValueCommand")
));
7.         this.id = id;
8.     }
9.
10.     @Override
11.     protected String run() {
12.         // RemoteService.getValue(id);
13.         throw new RuntimeException("force failure for example");
14.     }
15.
16.     @Override
17.     protected String getFallback() {
18.         return new FallbackViaNetwork(id).execute();
19.     }
20.
21.     private static class FallbackViaNetwork extends
HystrixCommand<String> {
22.         private final int id;
23.         public FallbackViaNetwork(int id) {
24.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Remo
teServiceX"))
25.
.andCommandKey(HystrixCommandKey.Factory.asKey("GetValueFallbackCom
mand"))
26.                 // 使用不同的线程池做隔离，防止上层线程池跑满，影响
降级逻辑.
27.
.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("RemoteServiceXFal
lback")));
28.             this.id = id;
29.         }
30.         @Override
31.         protected String run() {
```

```
32.             MemCacheClient.getValue(id);
33.         }
34.
35.         @Override
36.         protected String getFallback() {
37.             return null;
38.         }
39.     }
40. }
```

NOTE:依赖调用和降级调用使用不同的线程池做隔离，防止上层线程池跑满，影响二级降级逻辑调用.

11:显示调用fallback逻辑,用于特殊业务处理



Java代码

```
1. public class CommandFacadeWithPrimarySecondary extends
HystrixCommand<String> {
2.     private final static DynamicBooleanProperty usePrimary =
DynamicPropertyFactory.getInstance().getBooleanProperty("primarySeconda
ry.usePrimary", true);
3.     private final int id;
4.     public CommandFacadeWithPrimarySecondary(int id) {
5.         super(Setter
6.
.withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
7.
.andCommandKey(HystrixCommandKey.Factory.asKey("PrimarySecondaryCo
mmand"))
8.                 .andCommandPropertiesDefaults(
9.                         HystrixCommandProperties.Setter()
10.
.withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)));
11.         this.id = id;
```

```
12.     }
13.     @Override
14.     protected String run() {
15.         if (usePrimary.get()) {
16.             return new PrimaryCommand(id).execute();
17.         } else {
18.             return new SecondaryCommand(id).execute();
19.         }
20.     }
21.     @Override
22.     protected String getFallback() {
23.         return "static-fallback-" + id;
24.     }
25.     @Override
26.     protected String getCacheKey() {
27.         return String.valueOf(id);
28.     }
29.     private static class PrimaryCommand extends
HystrixCommand<String> {
30.         private final int id;
31.         private PrimaryCommand(int id) {
32.             super(Setter
33.
.withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
34.
.andCommandKey(HystrixCommandKey.Factory.asKey("PrimaryCommand"))
35.
.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("PrimaryCommand"
))
36.                     .andCommandPropertiesDefaults(
37.                             // we default to a 600ms timeout for primary
38.
HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(600
)));
39.             this.id = id;
40.         }
41.         @Override
42.         protected String run() {
43.             // perform expensive 'primary' service call
44.             return "responseFromPrimary-" + id;
45.         }
46.     }
47.     private static class SecondaryCommand extends
HystrixCommand<String> {
48.         private final int id;
```

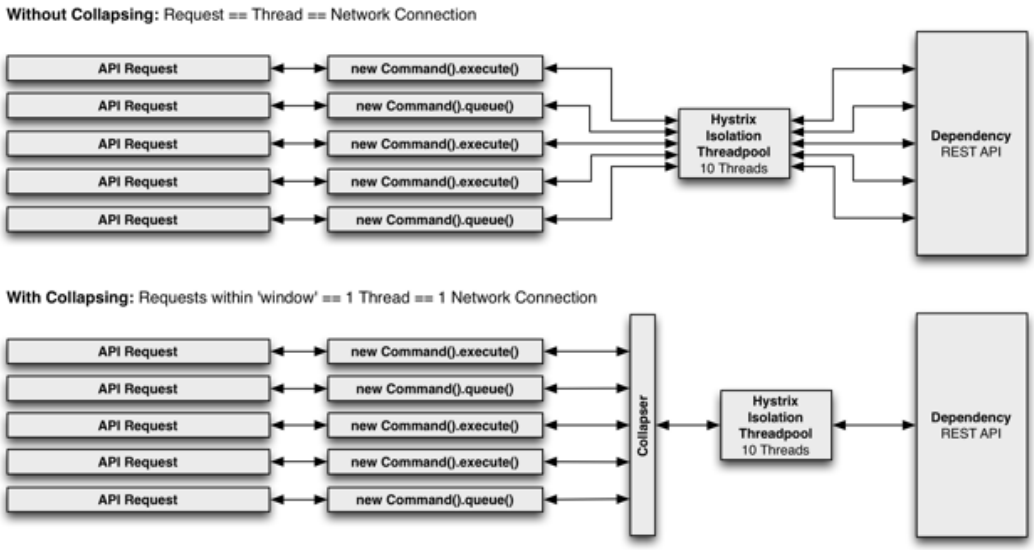
```
49.         private SecondaryCommand(int id) {
50.             super(Setter
51.                 
.withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
52.                 
.andCommandKey(HystrixCommandKey.Factory.asKey("SecondaryCommand"
))
53.                 
.andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("SecondaryComman
d"))
54.                     .andCommandPropertiesDefaults(
55.                             // we default to a 100ms timeout for secondary
56.                             
HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(100
)));
57.             this.id = id;
58.         }
59.         @Override
60.         protected String run() {
61.             // perform fast 'secondary' service call
62.             return "responseFromSecondary-" + id;
63.         }
64.     }
65.     public static class UnitTest {
66.         @Test
67.         public void testPrimary() {
68.             HystrixRequestContext context = 
HystrixRequestContext.initializeContext();
69.             try {
70.                 
ConfigurationManager.getConfigInstance().setProperty("primarySecondary.us
ePrimary", true);
71.                 assertEquals("responseFromPrimary-20", new 
CommandFacadeWithPrimarySecondary(20).execute());
72.             } finally {
73.                 context.shutdown();
74.                 ConfigurationManager.getConfigInstance().clear();
75.             }
76.         }
77.         @Test
78.         public void testSecondary() {
79.             HystrixRequestContext context = 
HystrixRequestContext.initializeContext();
80.             try {
81.                 
```

```
ConfigurationManager.getConfigInstance().setProperty("primarySecondary.us
ePrimary", false);
82.                 assertEquals("responseFromSecondary-20", new
CommandFacadeWithPrimarySecondary(20).execute());
83.             } finally {
84.                 context.shutdown();
85.                 ConfigurationManager.getConfigInstance().clear();
86.             }
87.         }
88.     }
89. }
```

NOTE:显示调用降级适用于特殊需求的场景,fallback用于业务处理，fallback不再承担降级职责，建议慎重使用，会造成监控统计换乱等问题.

12:命令调用合并:HystrixCollapser

命令调用合并允许多个请求合并到一个线程/信号下批量执行。
执行流程图如下:



Java代码

```
1. public class CommandCollapserGetValueForKey extends
HystrixCollapser<List<String>, String, Integer> {
2.     private final Integer key;
3.     public CommandCollapserGetValueForKey(Integer key) {
4.         this.key = key;
5.     }
6.     @Override
7.     public Integer getRequestArgument() {
8.         return key;
9.     }
```

```
10.     @Override
11.     protected HystrixCommand<List<String>> createCommand(final
Collection<CollapsedRequest<String, Integer>> requests) {
12.         //创建返回command对象
13.         return new BatchCommand(requests);
14.     }
15.     @Override
16.     protected void mapResponseToRequests(List<String>
batchResponse, Collection<CollapsedRequest<String, Integer>> requests) {
17.         int count = 0;
18.         for (CollapsedRequest<String, Integer> request : requests) {
19.             //手动匹配请求和响应
20.             request.setResponse(batchResponse.get(count++));
21.         }
22.     }
23.     private static final class BatchCommand extends
HystrixCommand<List<String>> {
24.         private final Collection<CollapsedRequest<String, Integer>>
requests;
25.         private BatchCommand(Collection<CollapsedRequest<String,
Integer>> requests) {
26.
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("Exam
pleGroup"))
27.
.andCommandKey(HystrixCommandKey.Factory.asKey("GetValueForKey")));
28.             this.requests = requests;
29.         }
30.         @Override
31.         protected List<String> run() {
32.             ArrayList<String> response = new ArrayList<String>();
33.             for (CollapsedRequest<String, Integer> request : requests) {
34.                 response.add("ValueForKey: " + request.getArgument());
35.             }
36.             return response;
37.         }
38.     }
39.     public static class UnitTest {
40.         HystrixRequestContext context =
HystrixRequestContext.initializeContext();
41.         try {
42.             Future<String> f1 = new
CommandCollapserGetValueForKey(1).queue();
43.             Future<String> f2 = new
CommandCollapserGetValueForKey(2).queue();
```

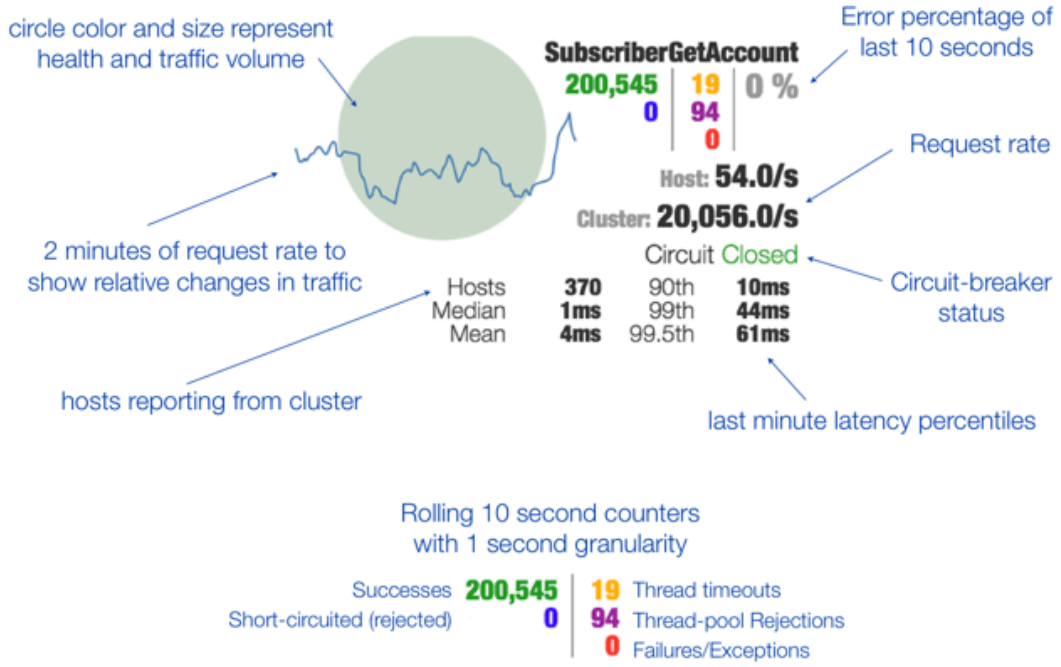
```
44.             Future<String> f3 = new
CommandCollapserGetValueForKey(3).queue();
45.             Future<String> f4 = new
CommandCollapserGetValueForKey(4).queue();
46.             assertEquals("ValueForKey: 1", f1.get());
47.             assertEquals("ValueForKey: 2", f2.get());
48.             assertEquals("ValueForKey: 3", f3.get());
49.             assertEquals("ValueForKey: 4", f4.get());
50.             assertEquals(1,
HystrixRequestLog.getCurrentRequest().getExecutedCommands().size());
51.             HystrixCommand<?> command =
HystrixRequestLog.getCurrentRequest().getExecutedCommands().toArray(n
ew HystrixCommand<?>[1])[0];
52.             assertEquals("GetValueForKey",
command.getCommandKey().name());
53.
assertTrue(command.getExecutionEvents().contains(HystrixEventType.COLL
APSED));
54.
assertTrue(command.getExecutionEvents().contains(HystrixEventType.SUCC
ESS));
55.         } finally {
56.          context.shutdown();
57.         }
58.     }
59. }
```

NOTE:使用场景:HystrixCollapser用于对多个相同业务的请求合并到一个线程甚至可以合并到一个连接中执行，降低线程交互次和IO数,但必须保证他们属于同一依赖.

四:监控平台搭建Hystrix-dashboard

1:监控dashboard介绍

dashboard面板可以对依赖关键指标提供实时监控,如下图:



2:实例暴露command统计数据

Hystrix使用Servlet对当前JVM下所有command调用情况作数据流输出
配置如下:

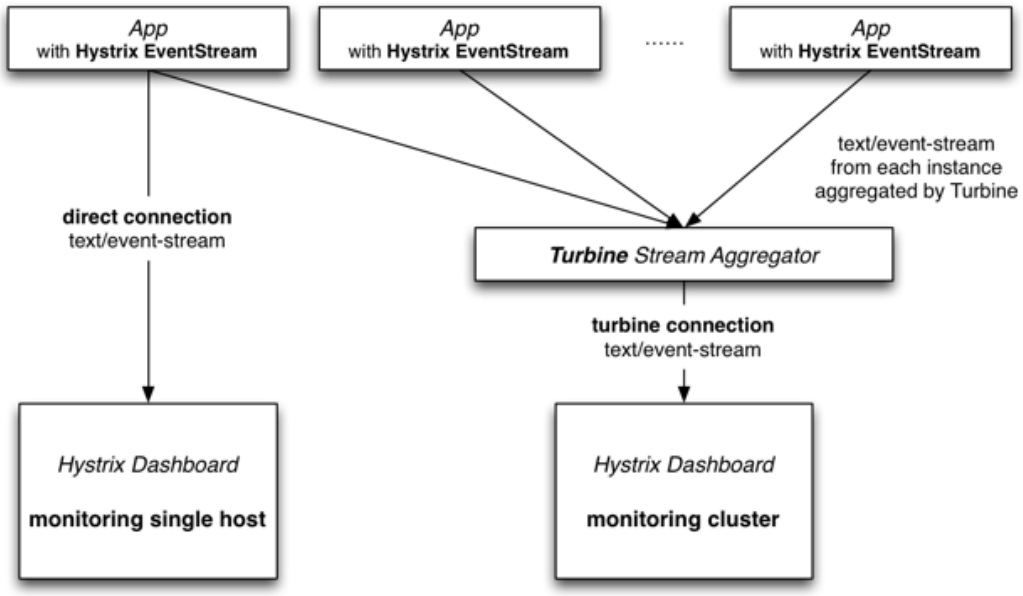
Xml代码

```
<servlet>
    <display-name>HystrixMetricsStreamServlet</display-name>
    <servlet-name>HystrixMetricsStreamServlet</servlet-name>
    <servlet-class>com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HystrixMetricsStreamServlet</servlet-name>
    <url-pattern>/hystrix.stream</url-pattern>
</servlet-mapping>
<!--
    对应URL格式：http://hostname:port/application/hystrix.stream
-->
```

3:集群模式监控统计搭建

1)使用Turbine组件做集群数据汇总

结构图如下;



2)内嵌jetty提供Servlet容器,暴露HystrixMetrics

Java代码

```
1. public class JettyServer {
2.     private final Logger logger =
LoggerFactory.getLogger(this.getClass());
3.     private int port;
4.     private ExecutorService executorService =
Executors.newFixedThreadPool(1);
5.     private Server server = null;
6.     public void init() {
7.         try {
8.             executorService.execute(new Runnable() {
9.                 @Override
10.                 public void run() {
11.                     try {
12.                         //绑定8080端口,加载HystrixMetricsStreamServlet并映射
url
13.                         server = new Server(8080);
14.                         WebAppContext context = new WebAppContext();
15.                         context.setContextPath("/");
16.                         context.addServlet(HystrixMetricsStreamServlet.class,
"/hystrix.stream");
17.                         context.setResourceBase(".");
18.                         server.setHandler(context);
19.                         server.start();
20.                         server.join();
```

```
                } catch (Exception e) {
                    logger.error(e.getMessage(), e);
                }
            }
        });
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
  }
  public void destory() {
    if (server != null) {
        try {
            server.stop();
            server.destroy();
            logger.warn("jettyServer stop and destroy!");
        } catch (Exception e) {
            logger.error(e.getMessage(), e);
        }
    }
  }
}
```

3)Turbine搭建和配置

a:配置Turbine Servlet收集器

Java代码

```
<servlet>
  <description></description>
  <display-name>TurbineStreamServlet</display-name>
  <servlet-name>TurbineStreamServlet</servlet-name>
  <servlet-class>com.netflix.turbine.streaming.servlet.TurbineStreamServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>TurbineStreamServlet</servlet-name>
  <url-pattern>/turbine.stream</url-pattern>
 </servlet-mapping>
```

b:编写config.properties配置集群实例

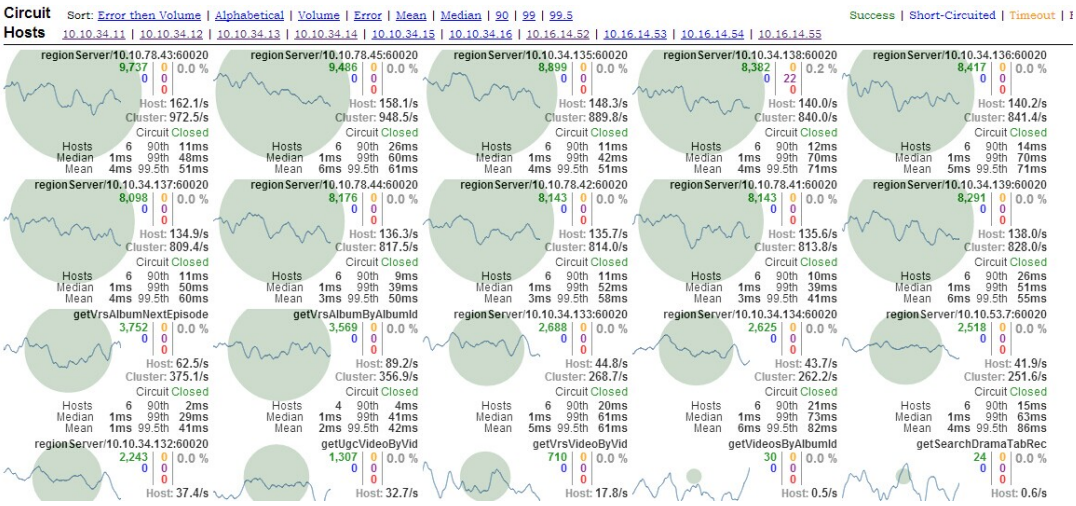
Java代码

```
#配置两个集群:mobil-online,ugc-online
turbine.aggregator.clusterConfig=mobil-online,ugc-online
```

```
3. #配置mobil-online集群实例
4. turbine.ConfigPropertyBasedDiscovery.mobil-
online.instances=10.10.*.*,10.10.*.*,10.10.*.*,10.10.*.*,10.10.*.*,10.10.*.
*,10.16.*.*,10.16.*.*,10.16.*.*,10.16.*.*
5. #配置mobil-online数据流servlet
6. turbine.instanceUrlSuffix.mobil-online=:8080/hystrix.stream
7. #配置ugc-online集群实例
8. turbine.ConfigPropertyBasedDiscovery.ugc-
online.instances=10.10.*.*,10.10.*.*,10.10.*.*,10.10.*.*#配置ugc-online数
据流servlet
9. turbine.instanceUrlSuffix.ugc-online=:8080/hystrix.stream
```

c:使用Dashboard配置连接Turbine

如下图：



五:Hystrix配置与分析

1:Hystrix 配置

1):Command 配置

Command配置源码在HystrixCommandProperties,构造Command时通过Setter进行配置

具体配置解释和默认值如下

Java代码

```
1. //使用命令调用隔离方式,默认:采用线程隔
离,ExecutionIsolationStrategy.THREAD
2. private final HystrixProperty<ExecutionIsolationStrategy>
executionIsolationStrategy;
3. //使用线程隔离时，调用超时时间，默认:1秒
4. private final HystrixProperty<Integer>
```

```
executionIsolationThreadTimeoutInMilliseconds;
5. //线程池的key,用于决定命令在哪个线程池执行
6. private final HystrixProperty<String>
executionIsolationThreadPoolKeyOverride;
7. //使用信号量隔离时，命令调用最大的并发数,默认:10
8. private final HystrixProperty<Integer>
executionIsolationSemaphoreMaxConcurrentRequests;
9. //使用信号量隔离时，命令fallback(降级)调用最大的并发数,默认:10
10. private final HystrixProperty<Integer>
fallbackIsolationSemaphoreMaxConcurrentRequests;
11. //是否开启fallback降级策略 默认:true
12. private final HystrixProperty<Boolean> fallbackEnabled;
13. // 使用线程隔离时，是否对命令执行超时的线程调用中断
（Thread.interrupt()）操作.默认:true
14. private final HystrixProperty<Boolean>
executionIsolationThreadInterruptOnTimeout;
15. // 统计滚动的时间窗口,默认:5000毫秒
circuitBreakerSleepWindowInMilliseconds
16. private final HystrixProperty<Integer>
metricsRollingStatisticalWindowInMilliseconds;
17. // 统计窗口的Buckets的数量,默认:10个,每秒一个Buckets统计
18. private final HystrixProperty<Integer>
metricsRollingStatisticalWindowBuckets; // number of buckets in the
statisticalWindow
19. //是否开启监控统计功能,默认:true
20. private final HystrixProperty<Boolean>
metricsRollingPercentileEnabled;
21. // 是否开启请求日志,默认:true
22. private final HystrixProperty<Boolean> requestLogEnabled;
23. //是否开启请求缓存,默认:true
24. private final HystrixProperty<Boolean> requestCacheEnabled; //
Whether request caching is enabled.
```

2):熔断器（Circuit Breaker）配置

Circuit Breaker配置源码在HystrixCommandProperties,构造Command时通过Setter进行配置,每种依赖使用一个Circuit Breaker

Java代码

```
1. // 熔断器在整个统计时间内是否开启的阀值，默认20秒。也就是10秒钟内
至少请求20次，熔断器才发挥起作用
2. private final HystrixProperty<Integer>
circuitBreakerRequestVolumeThreshold;
3. //熔断器默认工作时间,默认:5秒.熔断器中断请求5秒后会进入半打开状态,
放部分流量过去重试
```

```
4. private final HystrixProperty<Integer>
circuitBreakerSleepWindowInMilliseconds;
5. //是否启用熔断器,默认true. 启动
6. private final HystrixProperty<Boolean> circuitBreakerEnabled;
7. //默认:50%。当出错率超过50%后熔断器启动.
8. private final HystrixProperty<Integer>
circuitBreakerErrorThresholdPercentage;
9. //是否强制开启熔断器阻断所有请求,默认:false,不开启
10. private final HystrixProperty<Boolean> circuitBreakerForceOpen;
11. //是否允许熔断器忽略错误,默认false, 不开启
12. private final HystrixProperty<Boolean> circuitBreakerForceClosed;
```

3):命令合并(Collapser)配置

Command配置源码在HystrixCollapserProperties,构造Collapser时通过Setter进行配置

Java代码

```
1. //请求合并是允许的最大请求数,默认: Integer.MAX_VALUE
2. private final HystrixProperty<Integer> maxRequestsInBatch;
3. //批处理过程中每个命令延迟的时间,默认:10毫秒
4. private final HystrixProperty<Integer> timerDelayInMilliseconds;
5. //批处理过程中是否开启请求缓存,默认:开启
6. private final HystrixProperty<Boolean> requestCacheEnabled;
```

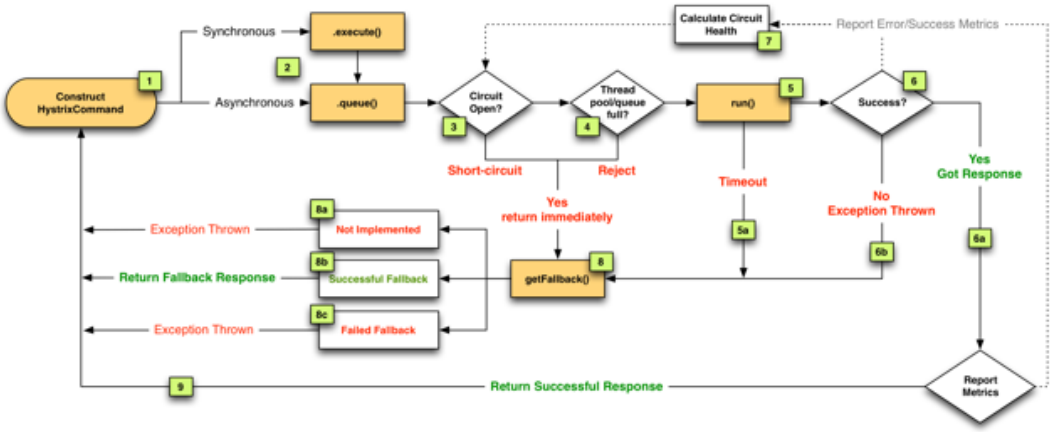
4):线程池(ThreadPool)配置

Java代码

```
1. /**
2. 配置线程池大小,默认值10个.
3. 建议值:请求高峰时99.5%的平均响应时间 + 向上预留一些即可
4. */
5. HystrixThreadPoolProperties.Setter().withCoreSize(int value)
6. /**
7. 配置线程值等待队列长度,默认值:-1
8. 建议值:-1表示不等待直接拒绝,测试表明线程池使用直接决绝策略+ 合适大小的非回缩线程池效率最高.所以不建议修改此值。
9. 当使用非回缩线程池时,
queueSizeRejectionThreshold,keepAliveTimeMinutes 参数无效
10. */
11. HystrixThreadPoolProperties.Setter().withMaxQueueSize(int value)
```

2:Hystrix关键组件分析

1):Hystrix流程结构解析



流程说明:

1:每次调用创建一个新的HystrixCommand,把依赖调用封装在run()方法中.

2:执行execute()/queue做同步或异步调用.

3:判断熔断器(circuit-breaker)是否打开,如果打开跳到步骤8,进行降级策略,如果关闭进入步骤.

4:判断线程池/队列/信号量是否跑满，如果跑满进入降级步骤8,否则继续后续步骤.

5:调用HystrixCommand的run方法.运行依赖逻辑

5a:依赖逻辑调用超时,进入步骤8.

6:判断逻辑是否调用成功

6a:返回成功调用结果

6b:调用出错，进入步骤8.

7:计算熔断器状态,所有的运行状态(成功, 失败, 拒绝,超时)上报给熔断器，用于统计从而判断熔断器状态.

8:getFallback()降级逻辑.

以下四种情况将触发getFallback调用:

(1):run()方法抛出非HystrixBadRequestException异常。

(2):run()方法调用超时

(3):熔断器开启拦截调用

(4):线程池/队列/信号量是否跑满

8a:没有实现getFallback的Command将直接抛出异常

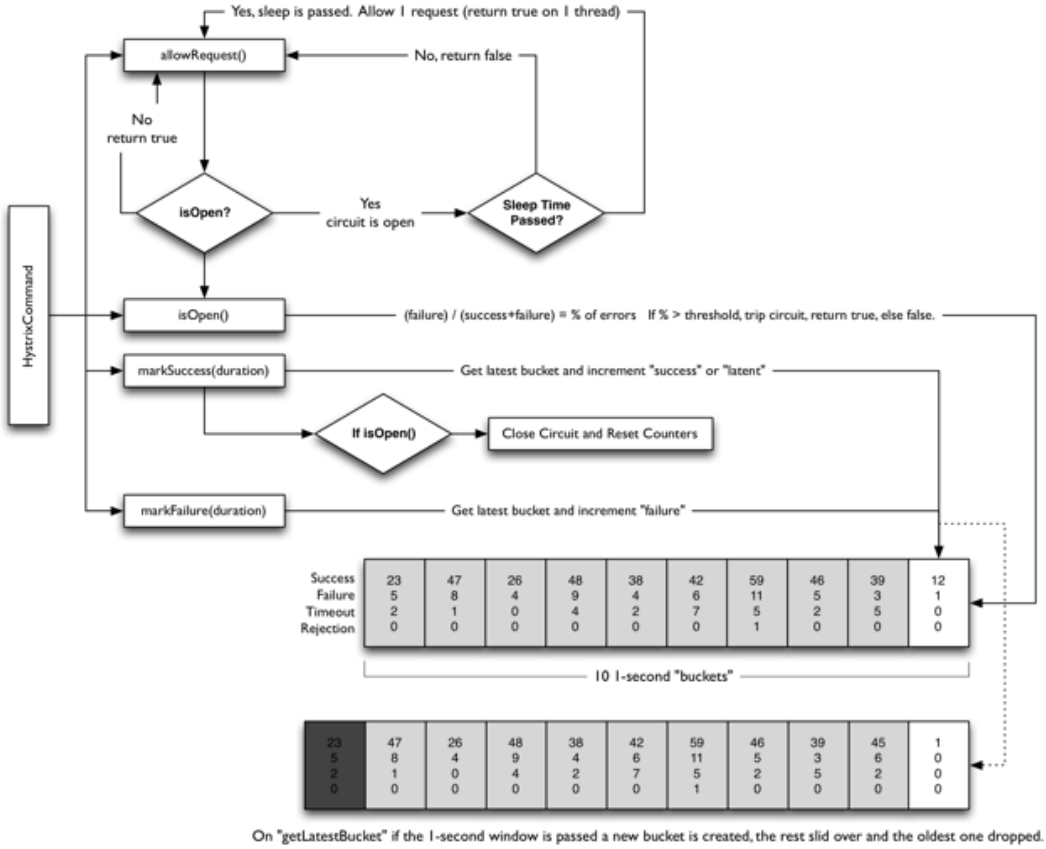
8b:fallback降级逻辑调用成功直接返回

8c:降级逻辑调用失败抛出异常

9:返回执行成功结果

2):熔断器:Circuit Breaker

Circuit Breaker 流程架构和统计



每个熔断器默认维护10个bucket,每秒一个bucket,每个blucket记录成功,失败,超时,拒绝的状态,

默认错误超过50%且10秒内超过20个请求进行中断拦截.

3)隔离(Isolation)分析

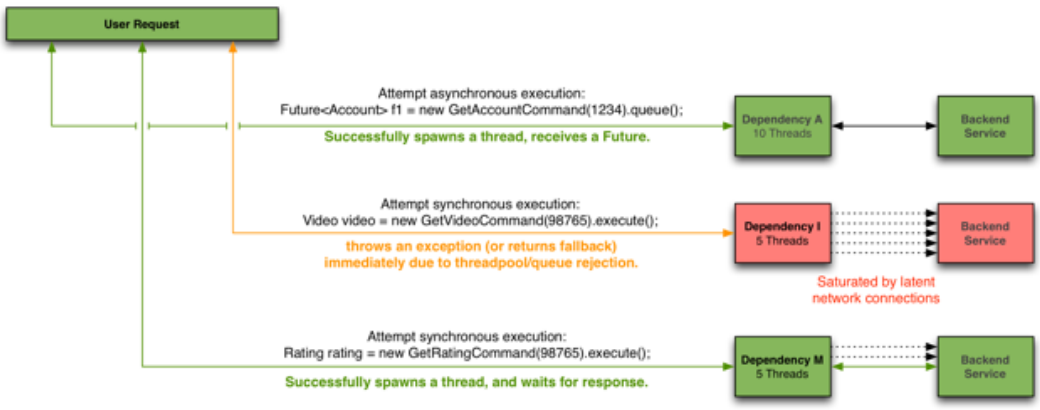
Hystrix隔离方式采用线程/信号的方式,通过隔离限制依赖的并发量和阻塞扩散.

(1):线程隔离

把执行依赖代码的线程与请求线程(如:jetty线程)分离，请求线程可以自由控制离开的时间(异步过程)。

通过线程池大小可以控制并发量，当线程池饱和时可以提前拒绝服务,防止依赖问题扩散。

线上建议线程池不要设置过大，否则大量堵塞线程有可能会拖慢服务器。



(2):线程隔离的优缺点

线程隔离的优点:

[1]:使用线程可以完全隔离第三方代码,请求线程可以快速放回。

[2]:当一个失败的依赖再次变成可用时，线程池将清理，并立即恢复可用，而不是一个长时间的恢复。

[3]:可以完全模拟异步调用，方便异步编程。

线程隔离的缺点:

[1]:线程池的主要缺点是它增加了cpu，因为每个命令的执行涉及到排队(默认使用SynchronousQueue避免排队)，调度和上下文切换。

[2]:对使用ThreadLocal等依赖线程状态的代码增加复杂性，需要手动传递和清理线程状态。

NOTE: Netflix公司内部认为线程隔离开销足够小，不会造成重大的成本或性能的影响。

Netflix 内部API 每天100亿的HystrixCommand依赖请求使用线程隔，每个应用大约40多个线程池，每个线程池大约5-20个线程。

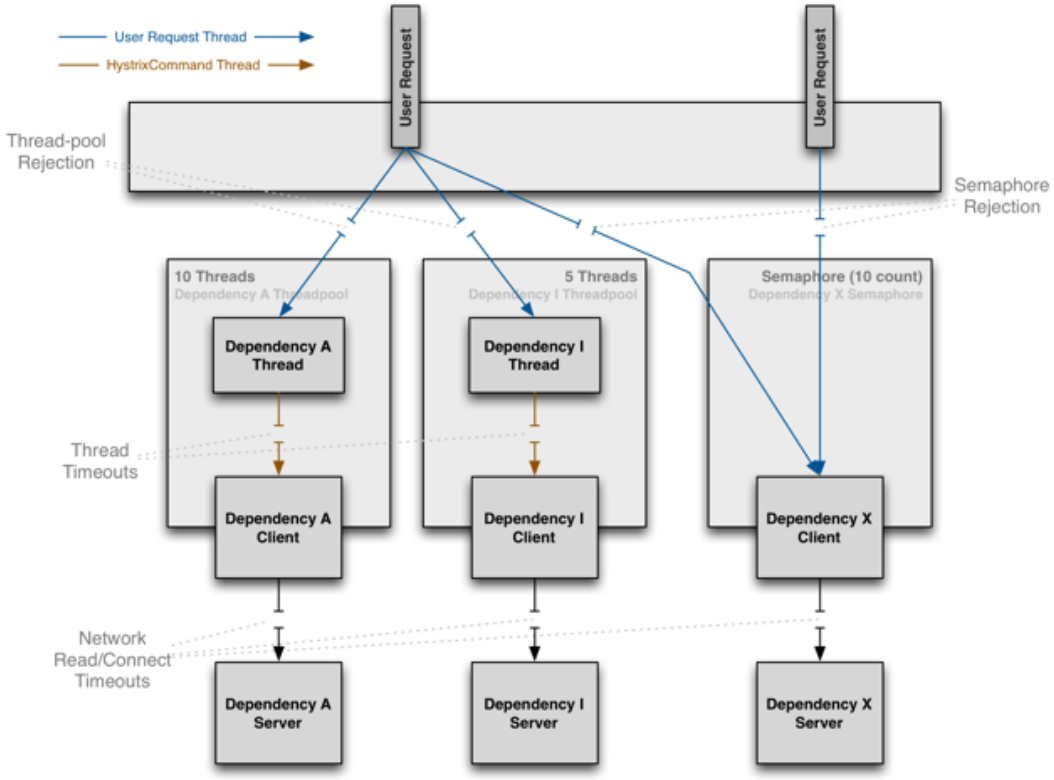
(3):信号隔离

信号隔离也可以用于限制并发访问，防止阻塞扩散, 与线程隔离最大不同在于执行依赖代码的线程依然是请求线程（该线程需要通过信号申请），

如果客户端是可信的且可以快速返回，可以使用信号隔离替换线程隔离,降低开销.

信号量的大小可以动态调整, 线程池大小不可以.

线程隔离与信号隔离区别如下图:



解析图片出自官网wiki , 更多内容请见官网: https://github.com/Netflix/Hystrix

- 查看图片附件

顶

踩

分享到:

nginx+lua+kafka实现日志统一收集汇总 | JVM日志和参数的理解