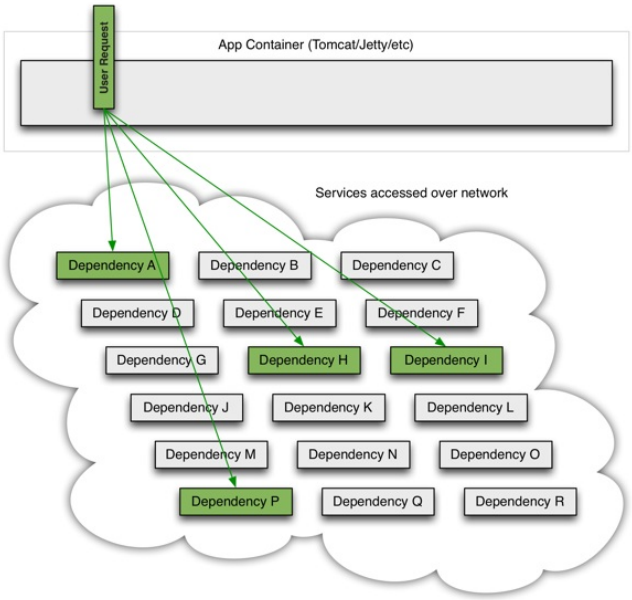
转载请注明出处哈:http://hot66hot.iteye.com/blog/2155036

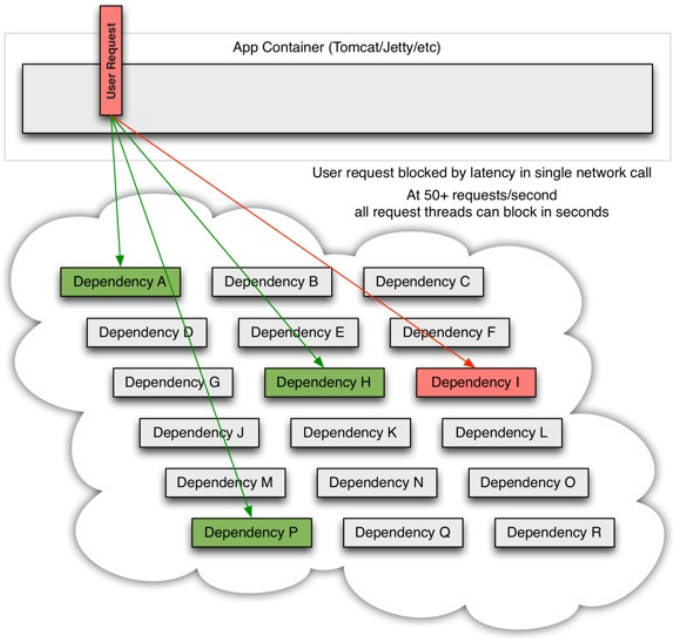
一:为什么需要Hystrix?

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。

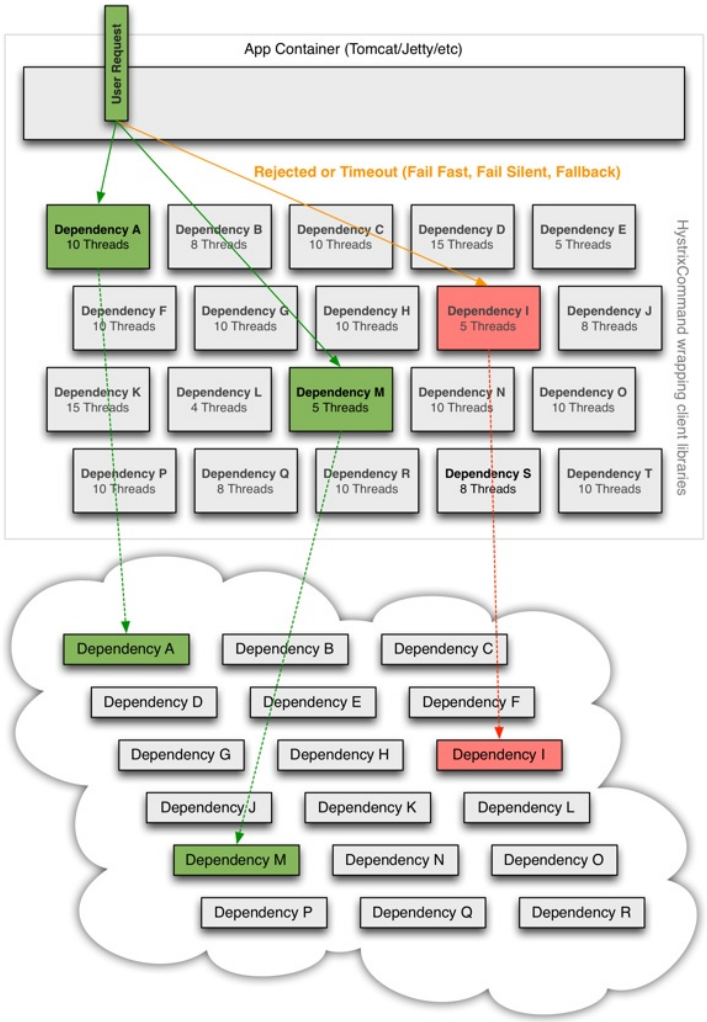
3:为每个依赖提供一个小的线程池（或信号），如果线程池已满调用将被立即拒绝，默认不采用排队.加速失败判定时间。

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

```
<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>
<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

```
publicclass HelloWorldCommand extends HystrixCommand<String> {
    privatefinal String name;
    public HelloWorldCommand(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }
    @Override
    protected String run() {
        return"Hello " + name +" thread:" + Thread.currentThread().getName();
    }
    publicstaticvoid main(String[] args) throws Exception{
        HelloWorldCommand helloWorldCommand = new HelloWorldCommand("Synchronous-hystrix");
        String result = helloWorldCommand.execute();
        System.out.println("result=" + result);
        helloWorldCommand = new HelloWorldCommand("Asynchronous-hystrix");
        Future<String> future = helloWorldCommand.queue();
        result = future.get(100, TimeUnit.MILLISECONDS);
        System.out.println("result=" + result);
        System.out.println("mainThread=" + Thread.currentThread().getName());
    }
}
```

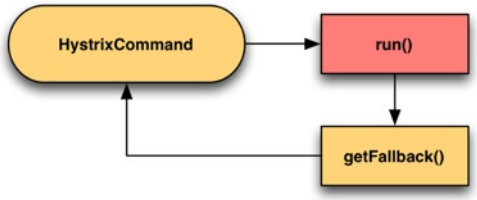
note:异步调用使用 command.queue()get(timeout, TimeUnit.MILLISECONDS);同步调用使用command.execute() 等同于 command.queue().get();

3:注册异步事件回调执行

Java代码

```
Observable<String> fs = new HelloWorldCommand("World").observe();
fs.subscribe(new Action1<String>() {
    @Override
    publicvoid call(String result) {
    }
});
fs.subscribe(new Observer<String>() {
            @Override
            publicvoid onCompleted() {
                System.out.println("execute onCompleted");
            }
            @Override
            publicvoid onError(Throwable e) {
                System.out.println("onError " + e.getMessage());
                e.printStackTrace();
            }
            @Override
            publicvoid onNext(String v) {
                System.out.println("onNext: " + v);
            }
        });
```

4:使用Fallback() 提供降级策略



Java代码

```
1. publicclass HelloWorldCommand extends HystrixCommand<String> {
2.     privatefinal String name;
3.     public HelloWorldCommand(String name) {
4.         super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))
5.                 .andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withExecutionIsolationThreadTimeoutInMilliseconds(500)));
6.         this.name = name;
7.     }
8.     @Override
9.     protected String getFallback() {
10.         return"exeucute Falled";
11.     }
12.     @Override
13.     protected String run() throws Exception {
14.         TimeUnit.MILLISECONDS.sleep(1000);
15.         return"Hello " + name +" thread:" + Thread.currentThread().getName();
16.     }
17.     publicstaticvoid main(String[] args) throws Exception{
18.         HelloWorldCommand command = new HelloWorldCommand("test-Fallback");
19.         String result = command.execute();
20.     }
21. }
```

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

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. public HelloWorldCommand(String name) {
2.     Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))
3. }
```

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("ExampleGroup"))
3.                 .andCommandKey(HystrixCommandKey.Factory.asKey("HelloWorld"))
4.                 .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("HelloWorldPool")));
5.         this.name = name;
6.     }
```

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

```
publicclass RequestCacheCommand extends HystrixCommand<String> {
    privatefinalint id;
    public RequestCacheCommand( int id) {
        super(HystrixCommandGroupKey.Factory.asKey("RequestCacheCommand"));
        this.id = id;
    }
    @Override
    protected String run() throws Exception {
        System.out.println(Thread.currentThread().getName() + " execute id=" + id);
        return"executed=" + id;
    }
    @Override
    protected String getCacheKey() {
        return String.valueOf(id);
    }
    publicstaticvoid main(String[] args){
        HystrixRequestContext context = HystrixRequestContext.initializeContext();
        try {
            RequestCacheCommand command2a = new RequestCacheCommand(2);
            RequestCacheCommand command2b = new RequestCacheCommand(2);
            Assert.assertTrue(command2a.execute());
            Assert.assertFalse(command2a.isResponseFromCache());
            Assert.assertTrue(command2b.execute());
            Assert.assertTrue(command2b.isResponseFromCache());
        } finally {
            context.shutdown();
        }
        context = HystrixRequestContext.initializeContext();
        try {
            RequestCacheCommand command3b = new RequestCacheCommand(2);
            Assert.assertTrue(command3b.execute());
            Assert.assertFalse(command3b.isResponseFromCache());
        } 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

```
publicclass HystrixRequestContextServletFilter implements Filter {
    publicvoid doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
     throws IOException, ServletException {
        HystrixRequestContext context = HystrixRequestContext.initializeContext();
        try {
            chain.doFilter(request, response);
        } finally {
            context.shutdown();
        }
    }
}
<filter>
    <display-name>HystrixRequestContextServletFilter</display-name>
    <filter-name>HystrixRequestContextServletFilter</filter-name>
    <filter-class>com.netflix.hystrix.contrib.requestservlet.HystrixRequestContextServletFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>HystrixRequestContextServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

9:信号量隔离:SEMAPHORE

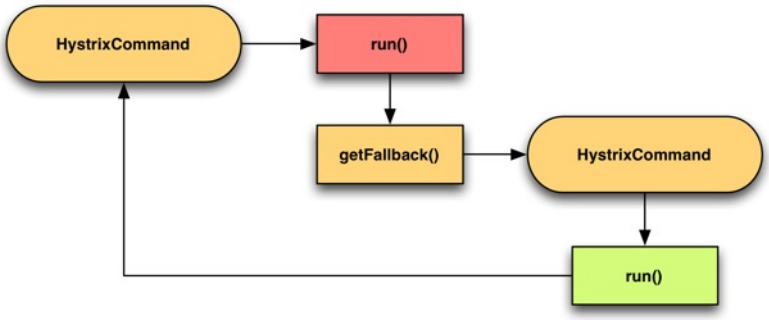
隔离本地代码或可快速返回远程调用(如memcached,redis)可以直接使用信号量隔离,降低线程隔离开销.

Java代码

```
publicclass HelloWorldCommand extends HystrixCommand<String> {
    privatefinal String name;
    public HelloWorldCommand(String name) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("HelloWorldGroup"))
                .andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolati
        this.name = name;
```

```
    }
    @Override
    protected String run() throws Exception {
        return"HystrixThread:" + Thread.currentThread().getName();
    }
    publicstaticvoid main(String[] args) throws Exception{
        HelloWorldCommand command = new HelloWorldCommand("semaphore");
        String result = command.execute();
        System.out.println(result);
        System.out.println("MainThread:" + Thread.currentThread().getName());
    }
}
```

10:fallback降级逻辑命令嵌套



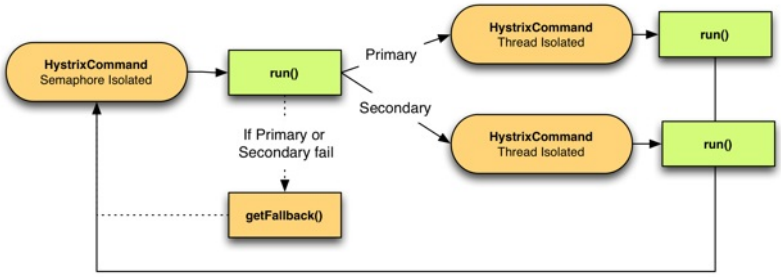
适用场景:用于fallback逻辑涉及网络访问的情况,如缓存访问。

Java代码

```
publicclass CommandWithFallbackViaNetwork extends HystrixCommand<String> {
    privatefinalint id;
    protected CommandWithFallbackViaNetwork(int id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceX"))
                .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueCommand")));
        this.id = id;
    }
    @Override
    protected String run() {
        thrownew RuntimeException("force failure for example");
    }
    @Override
    protected String getFallback() {
        returnnew FallbackViaNetwork(id).execute();
    }
    privatestaticclass FallbackViaNetwork extends HystrixCommand<String> {
        privatefinalint id;
        public FallbackViaNetwork(int id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceX"))
                    .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueFallbackCommand"))
                    .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("RemoteServiceXFallback")));
            this.id = id;
        }
        @Override
        protected String run() {
            MemCacheClient.getValue(id);
        }
        @Override
        protected String getFallback() {
            returnnull;
        }
    }
}
```

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. publicclass CommandFacadeWithPrimarySecondary extends HystrixCommand<String> {
2.     privatefinalstatic DynamicBooleanProperty usePrimary = DynamicPropertyFactory.getInstance().getBooleanProperty("primarySecondary.usePrimary", true
3.     privatefinalint id;
4.     public CommandFacadeWithPrimarySecondary(int id) {
5.         super(Setter
6.                 .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
7.                 .andCommandKey(HystrixCommandKey.Factory.asKey("PrimarySecondaryCommand"))
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.             returnnew PrimaryCommand(id).execute();
17.         } else {
18.             returnnew 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.     privatestaticclass PrimaryCommand extends HystrixCommand<String> {
30.         privatefinalint 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.                             HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(600)));
38.             this.id = id;
39.         }
40.         @Override
41.         protected String run() {
42.             return"responseFromPrimary-" + id;
43.         }
44.     }
45.     privatestaticclass SecondaryCommand extends HystrixCommand<String> {
46.         privatefinalint id;
47.         private SecondaryCommand(int id) {
48.             super(Setter
49.                     .withGroupKey(HystrixCommandGroupKey.Factory.asKey("SystemX"))
50.                     .andCommandKey(HystrixCommandKey.Factory.asKey("SecondaryCommand"))
51.                     .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("SecondaryCommand"))
52.                     .andCommandPropertiesDefaults(
53.                             HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(100)));
54.             this.id = id;
55.         }
56.         @Override
57.         protected String run() {
58.             return"responseFromSecondary-" + id;
59.         }
60.     }
61.     publicstaticclass UnitTest {
62.         @Test
```

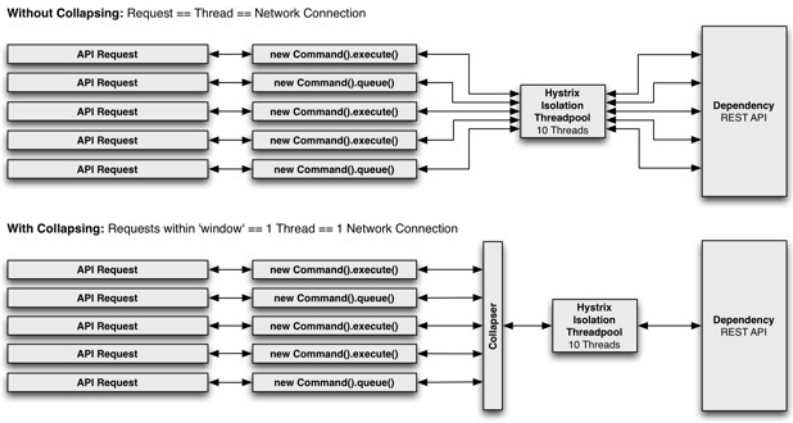
```
63.         publicvoid testPrimary() {
64.             HystrixRequestContext context = HystrixRequestContext.initializeContext();
65.             try {
66.                 ConfigurationManager.getConfigInstance().setProperty("primarySecondary.usePrimary", true);
67.                 assertEquals("responseFromPrimary-20", new CommandFacadeWithPrimarySecondary(20).execute());
68.             } finally {
69.                 context.shutdown();
70.                 ConfigurationManager.getConfigInstance().clear();
71.             }
72.         }
73.         @Test
74.         publicvoid testSecondary() {
75.             HystrixRequestContext context = HystrixRequestContext.initializeContext();
76.             try {
77.                 ConfigurationManager.getConfigInstance().setProperty("primarySecondary.usePrimary", false);
78.                 assertEquals("responseFromSecondary-20", new CommandFacadeWithPrimarySecondary(20).execute());
79.             } finally {
80.                 context.shutdown();
81.                 ConfigurationManager.getConfigInstance().clear();
82.             }
83.         }
84.     }
85. }
```

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.  publicclass CommandCollapserGetValueForKey extends HystrixCollapser<List<String>, String, Integer> {
2.      privatefinal 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.         returnnew BatchCommand(requests);
13.     }
14.     @Override
15.     protectedvoid mapResponseToRequests(List<String> batchResponse, Collection<CollapsedRequest<String, Integer>> requests) {
16.         int count = 0;
17.         for (CollapsedRequest<String, Integer> request : requests) {
18.             request.setResponse(batchResponse.get(count++));
19.         }
20.     }
21.     privatestaticfinalclass BatchCommand extends HystrixCommand<List<String>> {
22.         privatefinal Collection<CollapsedRequest<String, Integer>> requests;
23.         private BatchCommand(Collection<CollapsedRequest<String, Integer>> requests) {
24.                 super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"))
25.                     .andCommandKey(HystrixCommandKey.Factory.asKey("GetValueForKey")));
26.             this.requests = requests;
```

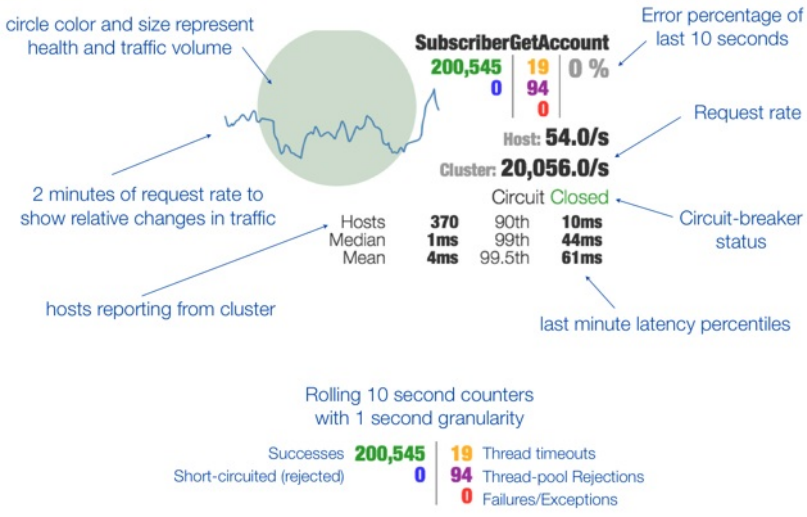
```
        }
        @Override
        protected List<String> run() {
            ArrayList<String> response = new ArrayList<String>();
            for (CollapsedRequest<String, Integer> request : requests) {
                response.add("ValueForKey: " + request.getArgument());
            }
            return response;
        }
    }
    publicstaticclass UnitTest {
        HystrixRequestContext context = HystrixRequestContext.initializeContext();
        try {
            Future<String> f1 = new CommandCollapserGetValueForKey(1).queue();
            Future<String> f2 = new CommandCollapserGetValueForKey(2).queue();
            Future<String> f3 = new CommandCollapserGetValueForKey(3).queue();
            Future<String> f4 = new CommandCollapserGetValueForKey(4).queue();
            assertEquals("ValueForKey: 1", f1.get());
            assertEquals("ValueForKey: 2", f2.get());
            assertEquals("ValueForKey: 3", f3.get());
            assertEquals("ValueForKey: 4", f4.get());
            assertEquals(1, HystrixRequestLog.getCurrentRequest().getExecutedCommands().size());
            HystrixCommand<?> command = HystrixRequestLog.getCurrentRequest().getExecutedCommands().toArray(new HystrixCommand<?>[1])
    [0];
            assertEquals("GetValueForKey", command.getCommandKey().name());
            assertTrue(command.getExecutionEvents().contains(HystrixEventType.COLLAPSED));
            assertTrue(command.getExecutionEvents().contains(HystrixEventType.SUCCESS));
        } finally {
         context.shutdown();
        }
    }
}
```

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

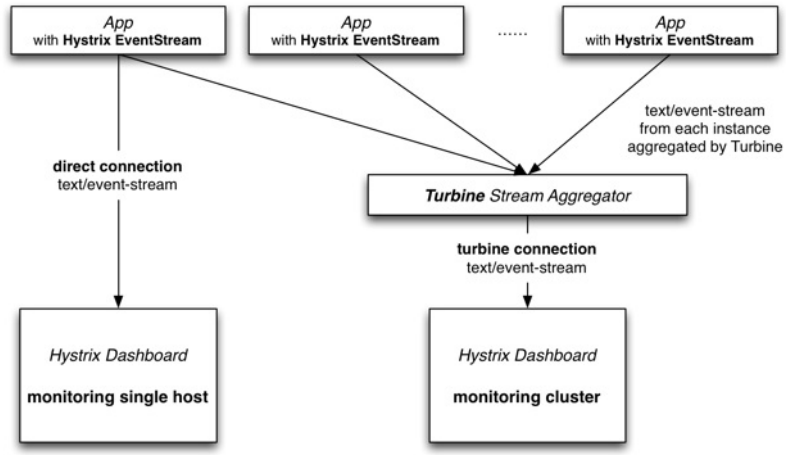
```
<servlet>
    <display-name>HystrixMetricsStreamServlet</display-name>
    <servlet-name>HystrixMetricsStreamServlet</servlet-name>
    <servlet-class>com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet</servlet-class>
```

```
</servlet>
<servlet-mapping>
    <servlet-name>HystrixMetricsStreamServlet</servlet-name>
    <url-pattern>/hystrix.stream</url-pattern>
</servlet-mapping>
```

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

```
publicclass JettyServer {
    privatefinal Logger logger = LoggerFactory.getLogger(this.getClass());
    privateint port;
    private ExecutorService executorService = Executors.newFixedThreadPool(1);
    private Server server = null;
    publicvoid init() {
        try {
            executorService.execute(new Runnable() {
                @Override
                publicvoid run() {
                    try {
                        server = new Server(8080);
                        WebAppContext context = new WebAppContext();
                        context.setContextPath("/");
                        context.addServlet(HystrixMetricsStreamServlet.class, "/hystrix.stream");
                        context.setResourceBase(".");
                        server.setHandler(context);
                        server.start();
                        server.join();
                    } catch (Exception e) {
                        logger.error(e.getMessage(), e);
                    }
                }
            });
        } catch (Exception e) {
            logger.error(e.getMessage(), e);
        }
    }
    publicvoid 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>
```

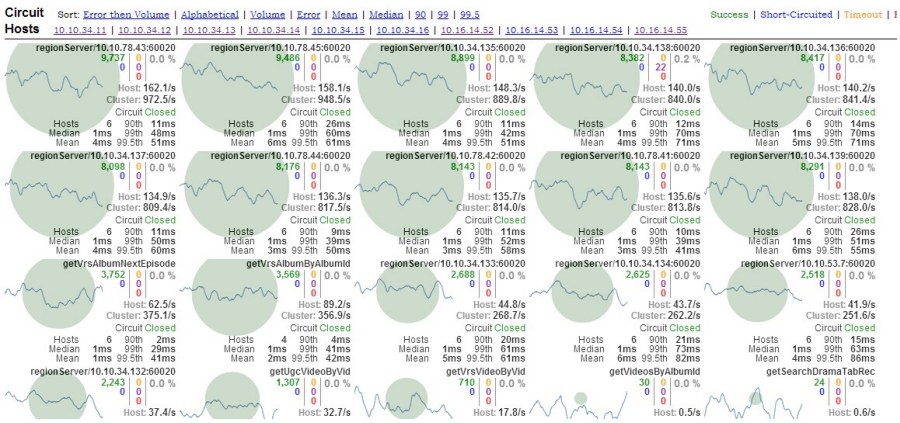
b:编写config.properties配置集群实例

Java代码

```
#配置两个集群:mobil-online,ugc-online
turbine.aggregator.clusterConfig=mobil-online,ugc-online
#配置mobil-online集群实例
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.*.*
#配置mobil-online数据流servlet
turbine.instanceUrlSuffix.mobil-online=:8080/hystrix.stream
#配置ugc-online集群实例
turbine.ConfigPropertyBasedDiscovery.ugc-online.instances=10.10.*.*,10.10.*.*,10.10.*.*,10.10.*.*#配置ugc-online数据流servlet
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

```
privatefinal HystrixProperty<ExecutionIsolationStrategy> executionIsolationStrategy;
privatefinal HystrixProperty<Integer> executionIsolationThreadTimeoutInMilliseconds;
privatefinal HystrixProperty<String> executionIsolationThreadPoolKeyOverride;
privatefinal HystrixProperty<Integer> executionIsolationSemaphoreMaxConcurrentRequests;
privatefinal HystrixProperty<Integer> fallbackIsolationSemaphoreMaxConcurrentRequests;
privatefinal HystrixProperty<Boolean> fallbackEnabled;
privatefinal HystrixProperty<Boolean> executionIsolationThreadInterruptOnTimeout;
privatefinal HystrixProperty<Integer> metricsRollingStatisticalWindowInMilliseconds;
privatefinal HystrixProperty<Integer> metricsRollingStatisticalWindowBuckets;
privatefinal HystrixProperty<Boolean> metricsRollingPercentileEnabled;
privatefinal HystrixProperty<Boolean> requestLogEnabled;
privatefinal HystrixProperty<Boolean> requestCacheEnabled;
```

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

```
privatefinal HystrixProperty<Integer> circuitBreakerRequestVolumeThreshold;
```

2. privatefinal HystrixProperty<Integer> circuitBreakerSleepWindowInMilliseconds;
3. privatefinal HystrixProperty<Boolean> circuitBreakerEnabled;
4. privatefinal HystrixProperty<Integer> circuitBreakerErrorThresholdPercentage;
5. privatefinal HystrixProperty<Boolean> circuitBreakerForceOpen;
6. privatefinal 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. privatefinal HystrixProperty<Integer> maxRequestsInBatch;
2. privatefinal HystrixProperty<Integer> timerDelayInMilliseconds;
3. privatefinal HystrixProperty<Boolean> requestCacheEnabled;

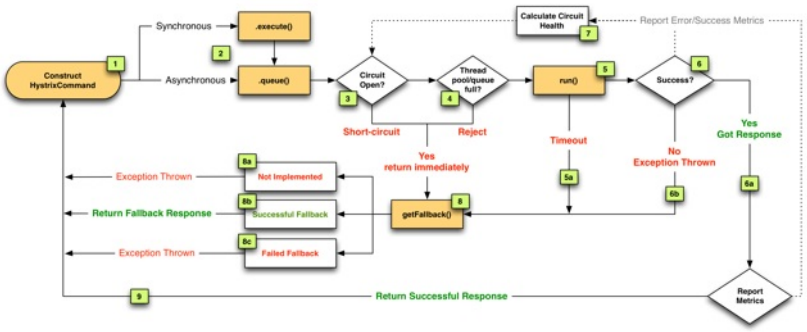
4):线程池(ThreadPool)配置

Java代码

1. HystrixThreadPoolProperties.Setter().withCoreSize(int value)
2. 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

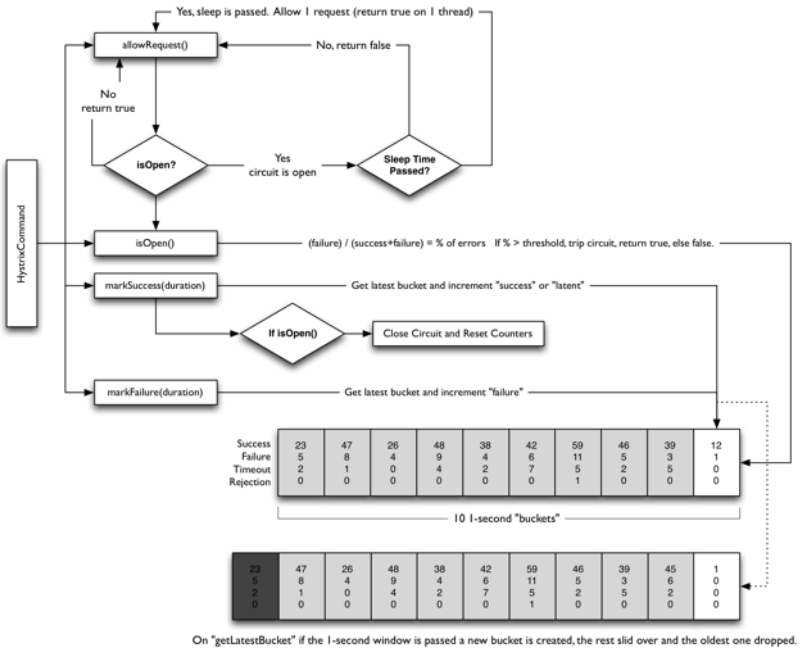
8b:fallback降级逻辑调用成功直接返回

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

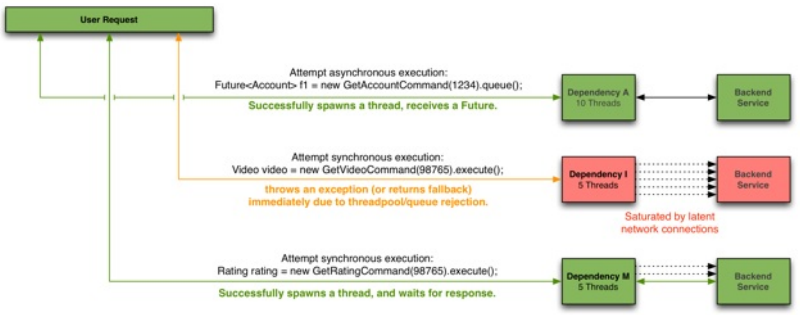
Hystrix隔离方式采用线程/信号的方式,通过隔离限制依赖的并发量和阻塞扩散.

(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。

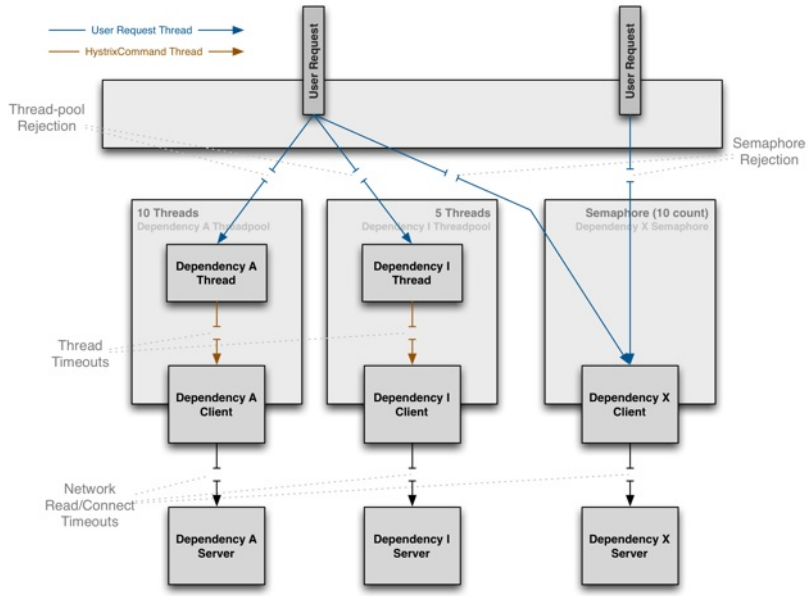
Netflix 内部API 每天100亿的HystrixCommand依赖请求使用线程隔，每个应用大约40多个线程池，每个线程池大约5-20个线程。

(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.

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