我们上一讲讲解的那个图片，顺着那个图片的流程，来一个一个的讲解hystrix的核心技术

1、创建command，2种command类型

2、执行command，4种执行方式

3、查找是否开启了request cache，是否有请求缓存，如果有缓存，直接取用缓存，返回结果

首先，有一个概念，叫做reqeust context，请求上下文，一般来说，在一个web应用中，hystrix

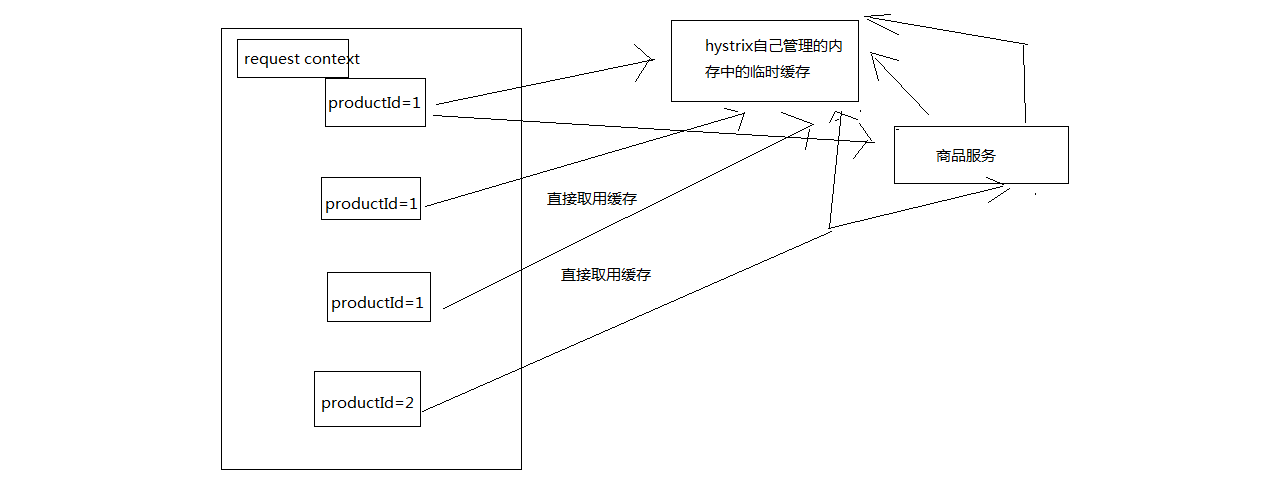
我们会在一个filter里面，对每一个请求都施加一个请求上下文，就是说，tomcat容器内，每一次请求，就是一次请求上下文

然后在这次请求上下文中，我们会去执行N多代码，调用N多依赖服务，有的依赖服务可能还会调用好几次

在一次请求上下文中，如果有多个command，参数都是一样的，调用的接口也是一样的，其实结果可以认为也是一样的

那么这个时候，我们就可以让第一次command执行，返回的结果，被缓存在内存中，然后这个请求上下文中，后续的其他对这个依赖的调用全部从内存中取用缓存结果就可以了

不用在一次请求上下文中反复多次的执行一样的command，提升整个请求的性能



Request cache原理

HystrixCommand和HystrixObservableCommand都可以指定一个缓存key，然后hystrix会自动进行缓存，接着在同一个request context内，再次访问的时候，就会直接取用缓存

用请求缓存，可以避免重复执行网络请求

多次调用一个command，那么只会执行一次，后面都是直接取缓存

对于请求缓存（request caching），请求合并（request collapsing），请求日志（request log），等等技术，都必须自己管理HystrixReuqestContext的声明周期

在一个请求执行之前，都必须先初始化一个request context

HystrixRequestContext context = HystrixRequestContext.initializeContext();

然后在请求结束之后，需要关闭request context

context.shutdown();

一般来说，在java web来的应用中，都是通过filter过滤器来实现的

public class HystrixRequestContextServletFilter implements Filter {

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HystrixRequestContext context = HystrixRequestContext.initializeContext();

try {

chain.doFilter(request, response);

} finally {

context.shutdown();

}

}

}

@Bean

public FilterRegistrationBean indexFilterRegistration() {

FilterRegistrationBean registration = new FilterRegistrationBean(new IndexFilter());

registration.addUrlPatterns("/");

return registration;

}

结合咱们的业务背景，我们做了一个批量查询商品数据的接口，在这个里面，我们其实通过HystrixObservableCommand一次性批量查询多个商品id的数据

但是这里有个问题，如果说nginx在本地缓存失效了，重新获取一批缓存，传递过来的productId都没有进行去重，1,1,2,2,5,6,7

那么可能说，商品id出现了重复，如果按照我们之前的业务逻辑，可能就会重复对productId=1的商品查询两次，productId=2的商品查询两次

我们对批量查询商品数据的接口，可以用request cache做一个优化，就是说一次请求，就是一次request context，对相同的商品查询只能执行一次，其余的都走request cache

public class CommandUsingRequestCache extends HystrixCommand<Boolean> {

private final int value;

protected CommandUsingRequestCache(int value) {

super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));

this.value = value;

}

@Override

protected Boolean run() {

return value == 0 || value % 2 == 0;

}

@Override

protected String getCacheKey() {

return String.valueOf(value);

}

}

@Test

public void testWithCacheHits() {

HystrixRequestContext context = HystrixRequestContext.initializeContext();

try {

CommandUsingRequestCache command2a = new CommandUsingRequestCache(2);

CommandUsingRequestCache command2b = new CommandUsingRequestCache(2);

assertTrue(command2a.execute());

// this is the first time we've executed this command with

// the value of "2" so it should not be from cache

assertFalse(command2a.isResponseFromCache());

assertTrue(command2b.execute());

// this is the second time we've executed this command with

// the same value so it should return from cache

assertTrue(command2b.isResponseFromCache());

} finally {

context.shutdown();

}

// start a new request context

context = HystrixRequestContext.initializeContext();

try {

CommandUsingRequestCache command3b = new CommandUsingRequestCache(2);

assertTrue(command3b.execute());

// this is a new request context so this

// should not come from cache

assertFalse(command3b.isResponseFromCache());

} finally {

context.shutdown();

}

}

缓存的手动清理

public static class GetterCommand extends HystrixCommand<String> {

private static final HystrixCommandKey GETTER\_KEY = HystrixCommandKey.Factory.asKey("GetterCommand");

private final int id;

public GetterCommand(int id) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GetSetGet"))

.andCommandKey(GETTER\_KEY));

this.id = id;

}

@Override

protected String run() {

return prefixStoredOnRemoteDataStore + id;

}

@Override

protected String getCacheKey() {

return String.valueOf(id);

}

/\*\*

\* Allow the cache to be flushed for this object.

\*

\* @param id

\* argument that would normally be passed to the command

\*/

public static void flushCache(int id) {

HystrixRequestCache.getInstance(GETTER\_KEY,

HystrixConcurrencyStrategyDefault.getInstance()).clear(String.valueOf(id));

}

}

public static class SetterCommand extends HystrixCommand<Void> {

private final int id;

private final String prefix;

public SetterCommand(int id, String prefix) {

super(HystrixCommandGroupKey.Factory.asKey("GetSetGet"));

this.id = id;

this.prefix = prefix;

}

@Override

protected Void run() {

// persist the value against the datastore

prefixStoredOnRemoteDataStore = prefix;

// flush the cache

GetterCommand.flushCache(id);

// no return value

return null;

}

}

《亿级流量电商详情页系统的大型高并发与高可用缓存架构实战》

1、亿级流量的电商网站的商品详情页系统架构

2、大型的企业级缓存架构，支撑高并发与高可用

3、几十万QPS的高并发+99.99%高可用+1T以上的海量数据+绝对数据安全的redis集群架构

4、高并发场景下的数据库+缓存双写一致性保障方案

5、大缓存的维度化拆分方案

6、基于双层nginx部署架构的缓存命中率提升方案

7、基于kafka+spring boot+ehcache+redis+nginx+lua的多级缓存架构

8、基于zookeeper的缓存并发更新安全保障方案

9、基于storm+zookeeper的大规模缓存预热解决方案

10、基于storm+zookeeper+nginx+lua的热点缓存自动降级与恢复解决方案

11、基于hystrix的高可用缓存服务架构

12、hystrix的进阶高可用架构方案、架构性能优化以及监控运维

13、基于hystrix的大规模缓存雪崩解决方案

14、高并发场景下的缓存穿透解决方案

15、高并发场景下的缓存失效解决方案