弹力设计篇之“重试设计”

2018-03-13 陈皓

弹力设计篇之“重试设计”
朗读人：柴巍　05’51’’ | 2.68M

关于重试，这个模式应该是一个很普遍的设计模式了。当我们把单体应用服务化，尤其是微服务化掉，本来在一个进程内的函数调用就成了远程调用，这样就会涉及到网络上的问题。

网络上有很多的各式各样的组件，如：DNS 服务，网卡、交换机、路由器、负载均衡等设备，这些设备都不一定是稳定的，在数据传输的整个过程中，只要一个环节出了问题，那么都会导致问题。

重试的场景

所以，我们需要一个重试的机制。但是，我们需要明白的是，**＂重试＂的语义是我们认为这个故障是暂时的，而不是永久的，所以，我们会去重试。**

所以，设计重试这个事时，我们需要定义出什么情况下需要重试，例如，调用超时、被调用端返回了某种可以重试的错误（如繁忙中、流控中、维护中、资源不足等）。

而对于一些别的错误，则最好不要重试，比如：业务级的错误（如没有权限、或是非法数据等错误），技术上的错误（如：HTTP 的 503 等，这种原因可能是触发了代码的 bug，重试下去没有意义）。

重试的策略

关于重试的设计，一般来说，都需要有个重试的最大值，经过一段时间不断的重试后，就没有必要再重试了，应该报故障了。在重试过程中，每一次重试时不成功时都应该休息一会儿再重试，这样可以避免因为重试过快而导致网络上的负担更重。

在重试的设计中，我们一般都会引入，Exponential Backoff 的策略，也就是所谓的 " 指数级退避 "。在这种情况下，每一次重试所需要的休息时间都会翻倍增加。这种机制主要是用来让被调用方能够有更多的时间来从容处理我们的请求。这其实和 TCP 的拥塞控制有点像。

如果我们写成代码应该是下面这个样子。

首先，我们定义一个调用返回的枚举类型，其中包括了 5 种返回错误——成功 SUCCESS、维护中 NOT_READY、流控中 TOO_BUSY、没有资源 NO_RESOURCE、系统错误 SERVER_ERROR。

```
public enum Results {
    SUCCESS,
    NOT_READY,
    TOO_BUSY,
    NO_RESOURCE,
    SERVER_ERROR
}
```

接下来，我们定义一个 Exponential Backoff 的函数，其返回 2 的指数。这样，每多一次重试就需要多等一段时间。如：第一次等 200ms，第二次要 400ms，第三次要等 800ms……

```
public static long getWaitTimeExp(int retryCount) {
    long waitTime = ((long) Math.pow(2, retryCount) );
    return waitTime;
}
```

下面是真正的重试逻辑。我们可以看到，在成功的情况下，以及不属于我们定义的错误下，我们是不需要重试的，而两次重试间需要等的时间是以指数上升的。

```
public static void doOperationAndWaitForResult() {

    // Do some asynchronous operation.
long token = asyncOperation();

    int retries = 0;
    boolean retry = false;

    do {
        // Get the result of the asynchronous operation.
        Results result = getAsyncOperationResult(token);

        if (Results.SUCCESS == result) {
            retry = false;
        } else if ( (Results.NOT_READY == result) ||
                    (Results.TOO_BUSY == result) ||
                    (Results.NO_RESOURCE == result) ||
                    (Results.SERVER_ERROR == result) ) {
            retry = true;
        } else {
            retry = false;
        }
        if (retry) {
            long waitTime = Math.min(getWaitTimeExp(retries), MAX_WAIT_INTERVAL);
            // Wait for the next Retry.
            Thread.sleep(waitTime);
        }
    } while (retry && (retries++ < MAX_RETRIES));
}
```

上面的代码是非常基本的重试代码，没有什么新鲜的，我们来看看 Spring 中所支持的一些重试策略。

Spring 的重试策略

Spring Retry 是专门的一个项目：https://github.com/spring-projects/spring-retry ，其中把 Spring 封装成了一个组件，是以 AOP 的方式通过 Annotation 的方式使用。例如如下的使用方式。

```
@Service
public interface MyService {
    @Retryable(
      value = { SQLException.class },
      maxAttempts = 2,
      backoff = @Backoff(delay = 5000))
    void retryService(String sql) throws SQLException;
    ...
}
```

配置 @Retryable 注解，只对 SQLException 的异常进行重试，重试两次，每次延时 5000ms。相关的细节可以看相应的文档。我在这里，只想让你看一下 Spring 有哪些重试的策略。

- NeverRetryPolicy：只允许调用 RetryCallback 一次，不允许重试。
- AlwaysRetryPolicy：允许无限重试，直到成功，此方式逻辑不当会导致死循环。
- SimpleRetryPolicy：固定次数重试策略，默认重试最大次数为 3 次，RetryTemplate 默认使用的策略。
- TimeoutRetryPolicy：超时时间重试策略，默认超时时间为 1 秒，在指定的超时时间内允许重试。
- CircuitBreakerRetryPolicy：有熔断功能的重试策略，需设置 3 个参数 openTimeout、resetTimeout 和 delegate；关于熔断，会在后面描述。
- CompositeRetryPolicy：组合重试策略。有两种组合方式，乐观组合重试策略是指只要有一个策略允许重试即可以，悲观组合重试策略是指只要有一个策略不允许重试即不可以。但不管哪种组合方式，组合中的每一个策略都会执行。

关于 Backoff 的策略如下。

- NoBackOffPolicy：无退避算法策略，即当重试时是立即重试；

- FixedBackOffPolicy：固定时间的退避策略，需设置参数 sleeper 和 backOffPeriod，sleeper 指定等待策略，默认是 Thread.sleep，即线程休眠，backOffPeriod 指定休眠时间，默认 1 秒。

- UniformRandomBackOffPolicy：随机时间退避策略，需设置 sleeper、minBackOffPeriod 和 maxBackOffPeriod。该策略在 [minBackOffPeriod, maxBackOffPeriod] 之间取一个随机休眠时间，minBackOffPeriod 默认为 500 毫秒，maxBackOffPeriod 默认为 1500 毫秒。

- ExponentialBackOffPolicy：指数退避策略，需设置参数 sleeper、initialInterval、maxInterval 和 multiplier。initialInterval 指定初始休眠时间，默认为 100 毫秒。maxInterval 指定最大休眠时间，默认为 30 秒。multiplier 指定乘数，即下一次休眠时间为当前休眠时间 *multiplier。

- ExponentialRandomBackOffPolicy：随机指数退避策略，引入随机乘数，之前说过固定乘数可能会引起很多服务同时重试导致 DDos，使用随机休眠时间来避免这种情况。

重试设计的重点

重试的设计重点主要如下：

- 要确定什么样的错误下需要重试；

- 重试的时间和重试的次数。这种在不同的情况下要有不同的考量。有时候，而对一些不是很重要的问题时，我们应该更快失败而不是重试一段时间若干次。比如一个前端的交互需要用到后端的服务。这种情况下，在面对错误的时候，应该快速度失败报错（比如：网络错误请重试）。而面对其它的一些错误，比如流控，那么应该使用指数退避的方式，以避免造成更多的流量。

- 如果超过重试次数，或是一段时间，那么重试就没有意义了。这个时候，说明这个错误不是一个短暂的错误，那么我们对于新来的请求，就没有必要再进行重试了，这个时候对新的请求直接返回错误就好了。但是，这样一来，如果后端恢复了，我们怎么知道呢，此时需要使用我们的熔断设计了。这个在后面会说。

- 重试还需要考虑被调用方是否有幂等的设计。如果没有，那么重试是不安全的，可能会导致一个相同的操作被执行多次。

- 重试的代码比较简单也比较通用，完全可以不用侵入到业务代码中。这里有两个模式。一个是代码级的，像 Java 那样可以使用 Annotation 的方式（在 Spring 中你可以用到这样的注释），如果没有注释也可以包装在底层库或是 SDK 库中不需要让上层业务感知到。另外一种是走 Service Mesh 的方式（关于 Service Mesh 的方式，会在后面的文章中介绍）。

- 对于有事务相关的操作。我们可能会希望能重试成功，而不至于走业务补偿那样的复杂的回退流程。对此，我们可能需要一个比较长的时间来做重试，但是我们需要保存住请求的上下文，这可能对程序的运行有比较大的开销，因此，有一些设计会先把这样的上下文暂存在本机或是数据库中，然后腾出资源来去做别的事，过一会再回来把之前的请求从存储中捞出来重试。

小结

好了，我们来总结一下今天分享的主要内容。首先，我讲了重试的场景，比如流控，但并不是所有的失败场景都适合重试。接着我讲了重试的策略，包括简单的指数退避策略，和 Spring 实现的多种策略。

这些策略可以用 Java 的 Annotation 来实现，或者用 Server Mesh 的方式，从而不必写在业务逻辑里。最后，我总结了重试设计的重点。下篇文章中，我们讲述熔断设计。希望对你有帮助。

也欢迎你分享一下你实现过哪些场景下的重试？所采用的策略是什么？实现的过程中遇到过哪些坑？

文末给出了《分布式系统设计模式》系列文章的目录，希望你能在这个列表里找到自己感兴趣的内容。

- 弹力设计篇
  - 认识故障和弹力设计
  - 隔离设计 Bulkheads
  - 异步通讯设计 Asynchronous
  - 幂等性设计 Idempotency
  - 服务的状态 State
  - 补偿事务 Compensating Transaction
  - 重试设计 Retry
  - 熔断设计 Circuit Breaker
  - 限流设计 Throttle
  - 降级设计 degradation
  - 弹力设计总结
- 管理设计篇
  - 分布式锁 Distributed Lock
  - 配置中心 Configuration Management
  - 边车模式 Sidecar
  - 服务网格 Service Mesh

- 网关模式 Gateway
- 部署升级策略

- 性能设计篇
  - 缓存 Cache
  - 异步处理 Asysnchronous
  - 数据库扩展
  - 秒杀 Flash Sales
  - 边缘计算 Edge Computing

精选留言

道道 1

之前做的重试策略是：异常发生的时候，数据库记录当前上下文，依据重试次数来确定重试时间，推送给延迟消息队列控制重试

2018-03-14

shufang 1

spring真的是只有想不到没有做不到～

2018-03-13

小沫 1

你好，对于重试是否可以不让当前线程休眠呢。如果当前线程休眠 此时这个线程的利用率就不高，我觉得应该放到线程池里面是否好一些呢？

2018-03-13

诤 0

server error不是不应该重试，属于服务端内部错误，不是暂时性的

2018-04-03

秋天 0

好文

2018-03-19

NonStatic 0

用过.net core上的Polly：http://www.thepollyproject.org/ 推荐给用C#的兄弟姐妹们。

2018-03-18

幻想 0

又一篇好文。感恩。。。

2018-03-13

Kennedy 0

503是服务过载，短暂不可用，可以重试吧？陈老师

2018-03-13