链滴

Redis 分布式锁

作者：dql

原文链接：https://ld246.com/article/1683959644430

来源网站：链滴

<h5 id="Watch-Dog-机制如果线程阻塞会一直持有锁吗">Watch Dog 机制如果线程阻塞会一直持锁吗</h5>
2023/5/9 21:06:15
在 Redis 中，Watch Dog 机制是一种乐观锁并发控制机制。它可以用于保证多个客户端并发修同一个数据时的数据一致性。
当一个客户端启用了 Watch 命令监视某个 Key 之后，如果这个 Key 被其他客户端修改了，那该客户端试图修改该 Key 时会失败，从而避免了数据不一致的情况。
在 Watch Dog 机制中，如果某个线程阻塞（比如在操作数据库或网络请求时），并不会一直持锁。因为 Watch 命令执行后，Redis 服务器会为客户端创建一个 Watch 命令所监视的事务状态，在客户端提交事务或者中途出现错误时都会清除该状态。
如果 Watch 命令所监视的 Key 被其它客户端修改，则在执行 EXEC 命令之前， Redis 服务器会测到事务状态已经变化，并向客户端返回一个空回复以及一个事务执行失败的通知。
因此，即使某个客户端被阻塞了，只要在事务提交之前没有其他客户端修改了 Watch 命令所监的 Key，就能正常提交事务。如果其他客户端修改了该 Key，则事务提交时会失败，这时候需要客户重新启动事务来处理异常情况。
<h5 id="分布式情况下为什么使用redision分布式锁-而不使用-synchronized关键字">分布式情况为什么使用 redision 分布式锁，而不使用 @synchronized 关键字</h5>
2023/5/9 08:19:17
在分布式系统中，多个节点同时访问共享资源时，使用传统的 Java 语言提供的 synchronized 键字是无法实现分布式锁的。因为 synchronized 关键字只能用于单机环境下的同步，无法跨越多台务器。而 Redisson 则提供了一种分布式锁的解决方案，它基于 Redis 实现，可以实现分布式环境下锁服务。
Redisson 分布式锁的优势在于：

可以避免死锁问题。Redisson 分布式锁采用了类似心跳检测的机制，当获取锁的客户端由于某原因宕机或者网络异常断开连接时，Redisson 会自动释放该客户端持有的锁，避免造成死锁的情况。/li>
支持公平锁和非公平锁。在高并发场景下，非公平锁可能会导致某些线程长时间无法获取到锁，而影响系统性能。而 Redisson 支持公平锁和非公平锁两种锁策略，可以根据实际业务需求选择合适锁类型。
支持可重入锁。在复杂应用场景下，可能需要对同一个资源进行多次加锁和解锁操作，此时就需支持可重入锁。Redisson 分布式锁支持可重入锁，保证了同一个线程可以多次获取同一个锁而不会成死锁问题。

综上所述，Redisson 分布式锁具有更高的灵活性、可靠性和性能，是一种更加适合分布式环境的锁服务。新建文本文档.txt
<h5 id="分布式锁都有那些实现">分布式锁都有那些实现</h5>
2023/5/9 08:31:08
分布式锁是一种用于控制分布式系统中多个节点对共享资源进行访问的机制，目前实现分布式锁方式有以下几种：

基于 Redis 实现的分布式锁：通过在 Redis 中存储一个标识符来实现锁的获取和释放，并且支设置锁的超时时间和可重入等特性。
基于 Zookeeper 实现的分布式锁：通过创建一个临时顺序节点来实现锁的获取和释放，并且支设置锁的超时时间和可重入等特性。
基于数据库实现的分布式锁：通过在数据库中存储一个标识符或者记录来实现锁的获取和释放，且需要保证数据库的高可用性和性能。
基于文件系统实现的分布式锁：通过创建一个文件或者目录来实现锁的获取和释放，并且需要保文件系统的高可用性和性能。
基于内存网格实现的分布式锁：通过在内存网格中存储一个标识符或者记录来实现锁的获取和释，并且需要保证内存网格的高可用性和性能。

以上列举的分布式锁实现方式仅为常见的几种，实际上还有很多其他的实现方式。每种实现方式有其优缺点，需要根据实际场景和需求来选择合适的方式。
<h5 id="redissen实现分布式锁">redissen 实现分布式锁</h5>
2023/5/7 23:49:33
Redission 是 Redis 的一个 Java 客户端，它提供了一系列分布式锁的实现。
使用 Redission 实现分布式锁的步骤如下：

创建 Redission 客户端。可以通过以下方式之一创建客户端：

使用默认配置创建客户端：
<pre class="code-block-wrapper"> <pan>复制代码 <code class="hljs code-block-body ">
</code></pre>
<code class="hljs code-block-body ">
</code>
<code class="hljs code-block-body ">RedisClient redisClient = RedisClient.creat("");
RedissonClient redissonClient = Redisson.create(redisClient);
</code>
使用自定义配置创建客户端：
<pre class="code-block-wrapper"> <pan>复制代码 <code class="hljs code-block-body ">
</code></pre>
<code class="hljs code-block-body ">
</code>
<code class="hljs code-block-body ">Config config = new Config();
config.useSingleServer().setAddress("");
RedissonClient redissonClient = Redisson.create(config);
</code>

获取分布式锁。使用 <code>getLock()</code>方法获取分布式锁对象，并调用 <code>lock()</code>方法获取锁。
<pre class="code-block-wrapper"> span>复制代码 <code class="hljs code-block-body ">
</code></pre>
<code class="hljs code-block-body ">
</code>
<code class="hljs code-block-body ">
</code> <code class="hljs code-block-body ">RLock lock = redissonClient.getLock("myock");
lock.lock();
// 执行临界区代码
lock.unlock();
</code>

释放分布式锁。在临界区代码执行完毕后，调用 <code>unlock()</code>方法释放锁。
<pre class="code-block-wrapper"> span>复制代码 <code class="hljs code-block-body ">
</code></pre>
<code class="hljs code-block-body ">
</code>
<code class="hljs code-block-body ">
</code> <code class="hljs code-block-body ">RLock lock = redissonClient.getLock("myock");
lock.lock();
try {
// 执行临界区代码
} finally {

lock.unlock();
}
</code>

需要注意的是，使用 Redission 实现分布式锁时，不需要手动设置锁的过期时间，因为 Redissio 会自动设置锁的过期时间，并在锁过期时自动释放锁。
<h4 id="高并发怎么保证线程安全-分布式怎么保证-">高并发怎么保证线程安全?分布式怎么保证?</4>
2023/5/11 00:10:32
在高并发场景下，保证线程安全是非常重要的。线程安全是指多个线程访问共享资源时，不会出数据错误或不一致的问题。为了保证线程安全，可以采用以下几种方式：

使用同步机制

使用同步机制可以保证在同一时间只有一个线程能够访问共享资源，从而避免了竞争条件和数据突等问题。常见的同步机制包括 synchronized 关键字、ReentrantLock 类等。
<ol start="2">
使用原子类

原子类是 JDK 提供的一组线程安全的基本数据类型，可以保证对变量的操作是原子性的，即不被其他线程干扰。常见的原子类包括 AtomicInteger、AtomicLong、AtomicBoolean 等。
<ol start="3">
使用线程安全的容器

Java 中提供了很多线程安全的容器，例如 ConcurrentHashMap、CopyOnWriteArrayList 等这些容器都是针对高并发场景进行优化的，并且内部实现了各种同步措施，可以保证多线程下的数据全。
在分布式环境下，保证数据安全更加复杂，需要考虑到多个节点之间的通信和数据同步。为了保分布式系统的数据安全，可以采用以下几种方式：

使用分布式锁

分布式锁可以在分布式环境下实现数据的互斥访问，避免多个节点同时对同一个数据进行修改操。常见的分布式锁包括 ZooKeeper、Redis 等。
<ol start="2">
使用分布式事务

分布式事务可以在分布式环境下保证数据一致性，避免由于多个节点之间的数据交互出现数据不致的问题。常见的分布式事务解决方案包括 XA 协议、TCC 补偿机制等。
<ol start="3">
使用 CAP 定理

在分布式环境下，根据 CAP 定理的原则，无法同时满足一致性、可用性和分区容错性三个特性因此，在设计分布式系统时，需要根据实际需求进行权衡，并选择适合的方案。
总之，在高并发和分布式环境下，保证数据安全是非常重要的。通过使用同步机制、原子类、线安全的容器、分布式锁、分布式事务等方式，可以有效地保证数据的正确性和一致性