Gmall

缓存&锁&分布式锁

# 一、缓存专题

## 1、查询缓存

**为了系统性能的提升**，我们一般都会将数据放入缓存中，加速访问。而db承担**数据落盘**工作。

高并发系统，缓存是必备。

### 1）、整合Redis作为缓存（使用jedis操作）

|  |
| --- |
| spring.redis.host=192.168.159.133  spring.redis.port=6379  spring.redis.timeout=10000  spring.redis.pool.max-active=100 |
| pom  <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>3.0.1</version>  </dependency> |
| 配置类  /\*\*  \* JedisConnectionFactory factory 这个参数是从容器中自动获取的  \* @param factory  \* @return 将我们做好的 JedisPool 放在容器中  \*/  @Bean  public JedisPool jedisPoolConfig(JedisConnectionFactory factory) {  //1、连接工厂中所有信息都有。  JedisPoolConfig config = factory.getPoolConfig();  JedisPool jedisPool = new JedisPool(config, factory.getHostName(), factory.getPort(), factory.getTimeout());  return jedisPool;  } |

## 2、缓存问题

### 1）、缓存穿透

* **缓存穿透**是指**查询一个一定不存在的数据**，由于缓存是不命中，将去查询数据库，但是数据库也无此记录，并且处于容错考虑，我们没有将这次查询的null写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。**在流量大时**，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。
* 解决

空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

### 2）、缓存雪崩

* 缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。
* 解决：

原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

### 3）、缓存击穿

* 对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个key在大量请求同时进来前正好失效，那么所有对这个key的数据查询都落到db，我们称为缓存击穿。和缓存雪崩的区别：
  + - 击穿是一个热点key失效
    - 雪崩是很多key集体失效
* 缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。
* 解决：

分布式锁

## 3、分布式锁



核心代码

|  |
| --- |
| 加锁  String token = UUID.randomUUID().toString();  String lock = jedis.set(key, token, "NX", "EX",20); |
| 解锁  String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";  jedis.eval(script, Collections.singletonList(key), Collections.singletonList(token)); |
| 进入redis测试  docker exec –it redis redis-cli  docker exec –it redis redis-benchmark |

## 4、缓存的使用模式

缓存使用模式分为两大类：Cache-Aside和Cache As SoR

专业名词：

* SoR（system-of-record）：记录系统，或者可以叫数据源，实际存储原始数据的系统
* Cache：缓存，是SoR的快照数据，Cache的访问速度比SoR要快，放入Cache的目的是提升系统速度，减少回源到SoR的次数
* 回源：回到数据源头检索数据，Cache没有命中需要回到SoR读取数据，这叫做回源

### 1）、Cache-Aside

Cache-Aside：即业务代码围绕Cache编写，由业务代码维护缓存；

如：

|  |
| --- |
| 读场景 |
| data = cache.getIfPresent(key);  if(data == null){  data = loadFromSoR(key);  cache.put(key,data)  }  先从缓存中获取，没有查询SoR，再放入缓存 |

|  |
| --- |
| 写场景（主动更新和删除时被动读取） |
| **双写模式**  //1、先将数据写入SoR  writeToSoR(k,v);  //2、执行成功后立即同步写入缓存  cache.put(k,v) |
| 或者  **失效模式**  //1、先将数据写入SoR  writeToSoR(k,v);  //2、失效缓存，下次读取时从缓存中加载（进入读场景），  cache.invalidate(k); |

|  |
| --- |
| 并发更新问题与解决（多个缓存实例，同时更新自己里面的同样数据） |
| 1、考虑使用中间件如Canal订阅binlog，进行增量更新分布式缓存，不会存在缓存数据不一致问题。但是会有延迟，我们可以调整合理的过期时间来容忍这个延迟  2、读服务，可以考虑一致性哈希，相同的操作负载均衡到同一个实例，从而减少并发几率。 |

### 2）、Cache-As-SoR

即把Cache当做SoR，所有操作都是对Cache进行，然后Cache委托SoR进行数据的真实读写。即业务代码只看到Cache的操作，看不到关于SoR的代码；三种实现模式：read-through、

write-through、write-behind

* read-through

业务代码首先调用Cache，Cache不命中，由Cache回源到SoR（类似回调的方式或者Cache配置的方式），而不是业务代码。【Guava、Ehcache均支持这种模式】

优点：业务代码整洁

* write-through

称为穿透写模式/直写模式。业务代码调用Cache写数据，然后由Cache负责写缓存和写SoR，而不是业务代码。

* write-behind

也叫write-back；回写模式。不同于write-through的是，write-through是同步写，而write-back是异步写，异步成功后可以实现批量写、合并写、延时写等

### 3）、Copy-Pattern

应用级缓存：（数据一定不能是脏数据）【Map、Ehcache、Guava】

Mybatis：二级缓存Map  
 if(map.get(1)){

return obj;

}else{ queryFromDb }

obj = cache;

缓存读取来的数据是否是改写安全；

obj = cache;

obj2 = new Obj();

BeanUtils.copyProperties(obj,obj2)

obj2.compute();

//写时复制

obj = ObjHaha();

obj2 = new Obj();

BeanUtils.copyProperties(obj,obj2)

map.put(k, obj2);

obj.compute();

分布式缓存：

缓存使用两种复制模式，注意。Copy-On-Read（读时复制）、Copy-On-Write（写时复制）。

有些进程内缓存很多是基于引用的，所以拿到缓存中的数据如果进行修改，可能发生不可预测的问题。

读时复制：读取到的缓存的值，复制内容封装一个新的对象。

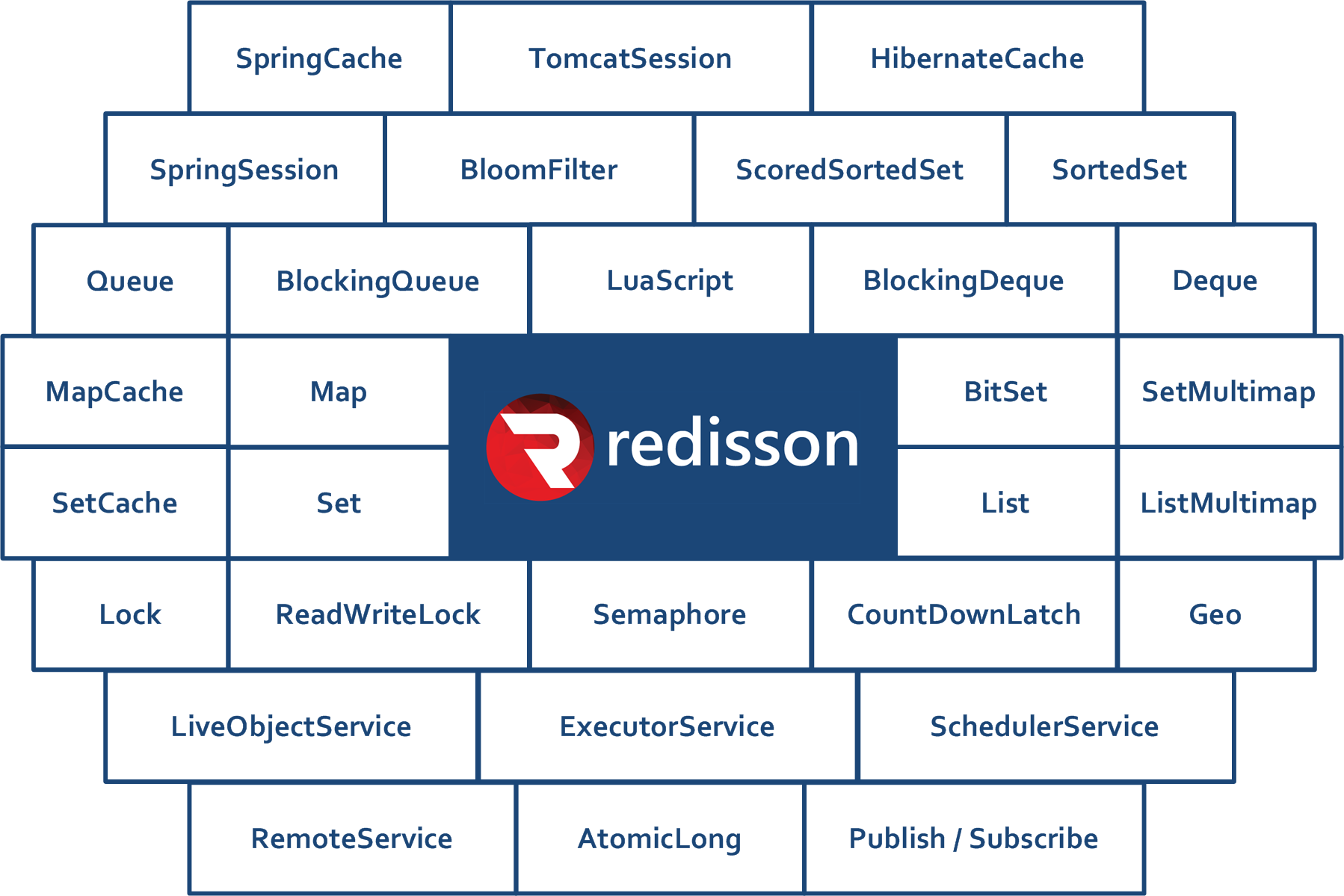
写时复制：给缓存中写的值，复制一个新的对象写入

## 5、redisson

<https://github.com/redisson/redisson/wiki/1.-Overview>

### 1）、简介

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务。其中包括(BitSet, Set, Multimap, SortedSet, Map, List, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, AtomicLong, CountDownLatch, Publish / Subscribe, Bloom filter, Remote service, Spring cache, Executor service, Live Object service, Scheduler service) Redisson提供了使用Redis的最简单和最便捷的方法。Redisson的宗旨是促进使用者对Redis的关注分离（Separation of Concern），从而让使用者能够将精力更集中地放在处理业务逻辑上。



### 2）、整合

|  |
| --- |
| 引入pom  *<!-- https://mvnrepository.com/artifact/org.redisson/redisson -->* <**dependency**>  <**groupId**>org.redisson</**groupId**>  <**artifactId**>redisson</**artifactId**>  <**version**>3.10.5</**version**> </**dependency**> |
| 配置  **spring.redis.host**=**192.168.159.130 spring.redis.port**=**6379** |
| 配置类  @Configuration **public class** GmallRedissonConfig {   @Value(**"${spring.redis.host}"**)  **private** String **host**;   @Value(**"${spring.redis.port}"**)  **private** String **port**;   @Bean  **public** RedissonClient redissonClient(){  Config config = **new** Config();  config.useSingleServer().setAddress(**"redis://"**+**host**+**":"**+**port**);  RedissonClient redisson = Redisson.*create*(config);  **return** redisson;  } } |

### 3）、锁机制

lock：可重入锁（最常用的锁？和synchronized有何不同？）

ReadWriteLock：读写锁（读共享，写互斥，写时不读）

Semaphore：信号量（停车场、acquire(1)占领一个车位,release()释放一个车位）

CountDownLatch：闭锁（班长最后一个关灯走人。。。）

#### 1、可重入锁（Reentrant Lock）

基于Redis的Redisson分布式可重入锁RLock Java对象实现了java.util.concurrent.locks.Lock接口。同时还提供了异步（Async）、反射式（Reactive）和RxJava2标准的接口。

|  |
| --- |
| RLock lock = redisson.getLock("anyLock");  // 最常见的使用方法，如果获取不到锁会一直自旋下去等待获取到锁  lock.lock(); |
| 另外Redisson还通过加锁的方法提供了leaseTime的参数来指定加锁的时间。超过这个时间后锁便自动解开了。  // 加锁以后10秒钟自动解锁  // 可以无需调用unlock方法手动解锁  lock.lock(10, TimeUnit.SECONDS);  // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁；如果没有等待时间，锁不住直接返回false  boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);  if (res) {  try {  ...  } finally {  lock.unlock();  }  } |
|  |

#### 2、公平锁（Fair Lock）

它保证了当多个Redisson客户端线程同时请求加锁时，优先分配给先发出请求的线程。所有请求线程会在一个队列中排队，当某个线程出现宕机时，Redisson会等待5秒后继续下一个线程，也就是说如果前面有5个线程都处于等待状态，那么后面的线程会等待至少25秒。

|  |
| --- |
| RLock fairLock = redisson.getFairLock("anyLock");  // 最常见的使用方法  fairLock.lock(); |
| // 10秒钟以后自动解锁  // 无需调用unlock方法手动解锁  fairLock.lock(10, TimeUnit.SECONDS);  // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁  boolean res = fairLock.tryLock(100, 10, TimeUnit.SECONDS);  ...  fairLock.unlock(); |

#### 3、联锁（MultiLock）

|  |
| --- |
| 基于Redis的Redisson分布式联锁[RedissonMultiLock](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/RedissonMultiLock.html)对象可以将多个RLock对象关联为一个联锁，每个RLock对象实例可以来自于不同的Redisson实例。 |
| RLock lock1 = redissonInstance1.getLock("lock1");  RLock lock2 = redissonInstance2.getLock("lock2");  RLock lock3 = redissonInstance3.getLock("lock3");  RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);  // 同时加锁：lock1 lock2 lock3  // 所有的锁都上锁成功才算成功。  lock.lock();  ...  lock.unlock(); |

#### 4、红锁（RedLock）

|  |
| --- |
| 基于Redis的Redisson红锁RedissonRedLock对象实现了[Redlock](http://redis.cn/topics/distlock.html)介绍的加锁算法。该对象也可以用来将多个RLock对象关联为一个红锁，每个RLock对象实例可以来自于不同的Redisson实例。 |
| RLock lock1 = redissonInstance1.getLock("lock1");  RLock lock2 = redissonInstance2.getLock("lock2");  RLock lock3 = redissonInstance3.getLock("lock3");  RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);  // 同时加锁：lock1 lock2 lock3  // 红锁在大部分节点上加锁成功就算成功。  lock.lock();  ...  lock.unlock(); |
| RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);  // 给lock1，lock2，lock3加锁，如果没有手动解开的话，10秒钟后将会自动解开  lock.lock(10, TimeUnit.SECONDS);  // 为加锁等待100秒时间，并在加锁成功10秒钟后自动解开  boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);  ...  lock.unlock(); |

#### 5、读写锁（ReadWriteLock）

|  |
| --- |
| 基于Redis的Redisson分布式可重入读写锁[RReadWriteLock](http://static.javadoc.io/org.redisson/redisson/3.4.3/org/redisson/api/RReadWriteLock.html) Java对象实现了java.util.concurrent.locks.ReadWriteLock接口。其中读锁和写锁都继承了[RLock](https://github.com/redisson/redisson/wiki/8.-%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%92%8C%E5%90%8C%E6%AD%A5%E5%99%A8#81-%E5%8F%AF%E9%87%8D%E5%85%A5%E9%94%81reentrant-lock)接口。  分布式可重入读写锁允许同时有多个读锁和一个写锁处于加锁状态。 |
| RReadWriteLock rwlock = redisson.getReadWriteLock("anyRWLock");  // 最常见的使用方法  rwlock.readLock().lock();  // 或  rwlock.writeLock().lock(); |
| // 10秒钟以后自动解锁  // 无需调用unlock方法手动解锁  rwlock.readLock().lock(10, TimeUnit.SECONDS);  // 或  rwlock.writeLock().lock(10, TimeUnit.SECONDS);  // 尝试加锁，最多等待100秒，上锁以后10秒自动解锁  boolean res = rwlock.readLock().tryLock(100, 10, TimeUnit.SECONDS);  // 或  boolean res = rwlock.writeLock().tryLock(100, 10, TimeUnit.SECONDS);  ...  lock.unlock(); |

#### 6、信号量（Semaphore）

|  |
| --- |
| 基于Redis的Redisson的分布式信号量（[Semaphore](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RSemaphore.html)）Java对象RSemaphore采用了与java.util.concurrent.Semaphore相似的接口和用法。同时还提供了[异步（Async）](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RSemaphoreAsync.html)、[反射式（Reactive）](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RSemaphoreReactive.html)和[RxJava2标准](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RSemaphoreRx.html)的接口。 |
| RSemaphore semaphore = redisson.getSemaphore("semaphore");  semaphore.acquire();  //或  semaphore.acquireAsync();  semaphore.acquire(23);  semaphore.tryAcquire();  //或  semaphore.tryAcquireAsync();  semaphore.tryAcquire(23, TimeUnit.SECONDS);  //或  semaphore.tryAcquireAsync(23, TimeUnit.SECONDS);  semaphore.release(10);  semaphore.release();  //或  semaphore.releaseAsync(); |

#### 7、可过期性信号量（PermitExpirableSemaphore）

基于Redis的Redisson可过期性信号量（[PermitExpirableSemaphore](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RPermitExpirableSemaphore.html)）是在RSemaphore对象的基础上，为每个信号增加了一个过期时间。每个信号可以通过独立的ID来辨识，释放时只能通过提交这个ID才能释放。它提供了[异步（Async）](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RPermitExpirableSemaphoreAsync.html)、[反射式（Reactive）](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RPermitExpirableSemaphoreReactive.html)和[RxJava2标准](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RPermitExpirableSemaphoreRx.html)的接口。

|  |
| --- |
| RPermitExpirableSemaphore semaphore = redisson.getPermitExpirableSemaphore("mySemaphore");  String permitId = semaphore.acquire();  // 获取一个信号，有效期只有2秒钟。  String permitId = semaphore.acquire(2, TimeUnit.SECONDS);  // ...  semaphore.release(permitId); |

#### 8、闭锁（CountDownLatch）

|  |
| --- |
| 基于Redisson的Redisson分布式闭锁（[CountDownLatch](http://static.javadoc.io/org.redisson/redisson/3.10.0/org/redisson/api/RCountDownLatch.html)）Java对象RCountDownLatch采用了与java.util.concurrent.CountDownLatch相似的接口和用法。 |
| RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");  latch.trySetCount(1);  latch.await();  // 在其他线程或其他JVM里  RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");  latch.countDown(); |

### 4）、分布式集合

<https://github.com/redisson/redisson/wiki/7.-distributed-collections>

# 二、高并发压力测试

Jmeter：

Apache Ab：

应用级锁与分布式锁到底能解决什么问题。测试就知道了

|  |
| --- |
| netstat -aon|findstr "443"  tasklist |findstr “进程id”  taskkill /f /t /im “进程id”（用管理员） |
| nginx配置负载均衡  upstream tomcat {  server 192.168.159.1:8080 weight=20;  server 192.168.159.1:8081 weight=20;  server 192.168.159.1:8082 weight=20;  server 192.168.159.1:8083 weight=20;  }  location / {  proxy\_pass http://tomcat  } |
| C:\Users\lfy\AppData\Roaming\feiq\RichOle\4058099743.bmp |

# 三、高并发优化