Redis

1.redis 简介

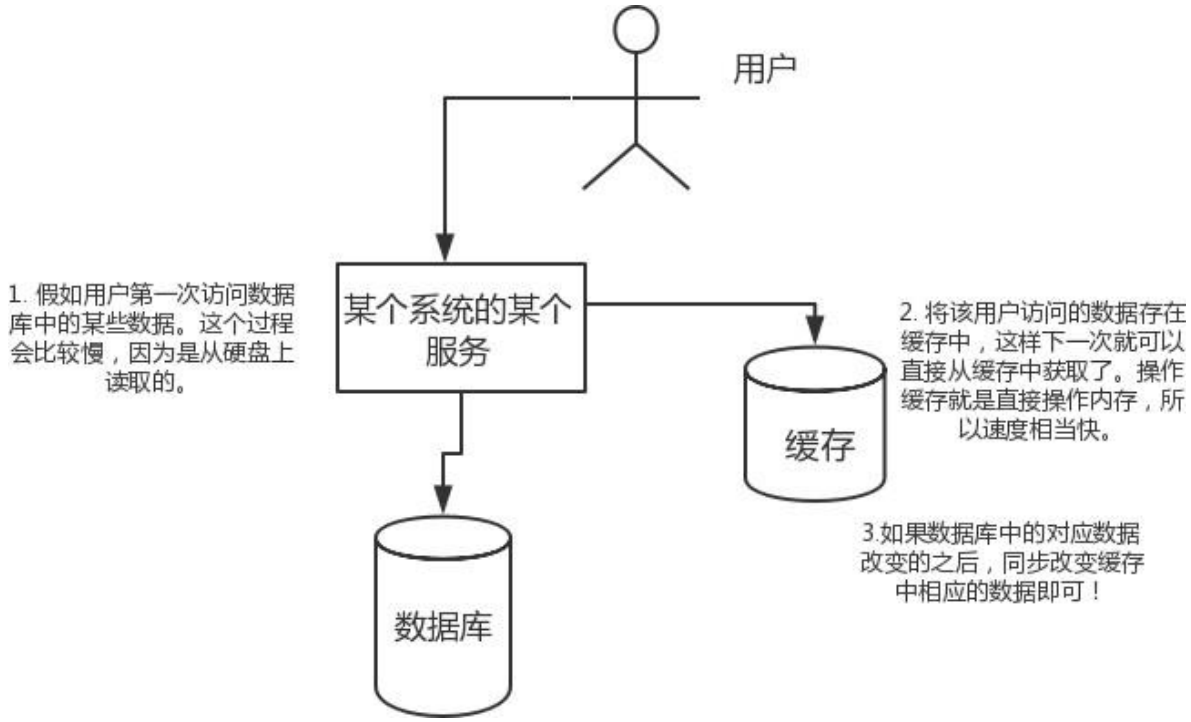
简单来说 redis 就是一个数据库，不过与传统数据库不同的是 redis 的数据是存在内存中的，所以存写速度非常快，因此 redis 被广泛应用于缓存方向。另外，redis 也经常用来做分布式锁。redis 提供了多种数据类型来支持不同的业务场景。除此之外，redis 支持事务 、持久化、LUA脚本、LRU驱动事件、多种集群方案。

2.为什么要用 redis /为什么要用缓存

主要从“高性能”和“高并发”这两点来看待这个问题。

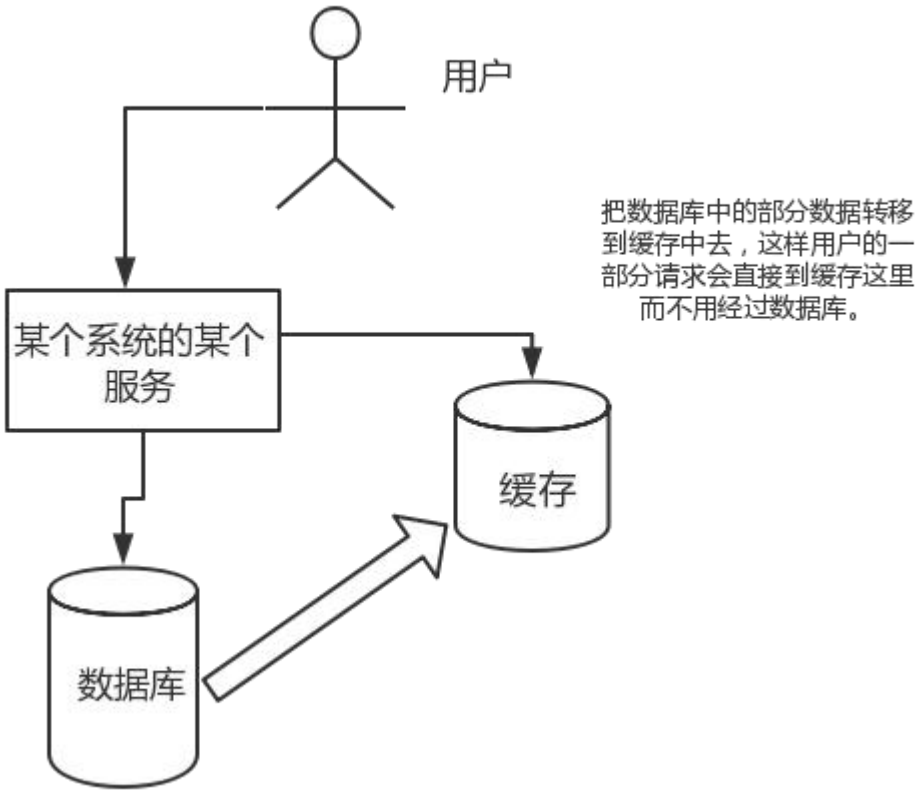
高性能

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数 缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当 快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！



高并发

直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。



3.为什么要用 redis 而不用 map/guava 做缓存?

> 下面的内容来自 segmentfault 一位网友的提问，地址：https://segmentfault.com/q/1010000009106416

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached服务的高可用，整个程序架构上较为复杂。

4.redis 和 memcached 的区别

对于 redis 和 memcached 我总结了下面四点。现在公司一般都是用 redis 来实现缓存，而且 redis 自身也越来越强大了！

1. **redis**支持更丰富的数据类型（支持更复杂的应用场景）：Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。memcache支持简单的数据类型，String。
2. **Redis**支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用,而**Memecache**把数据全部存在内存之中。
3. 集群模式：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 redis 目前是原生支持 cluster 模式的.

4. **Memcached**是多线程，非阻塞**IO**复用的网络模型；**Redis**使用单线程的多路 **IO** 复用模型。

> 来自网络上的一张图，这里分享给大家！

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

5.redis 常见数据结构以及使用场景分析

1. String

常用命令: set,get,decr,incr,mget 等。

String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。 常规key-value缓存应用； 常规计数：微博数，粉丝数等。

2.Hash

常用命令： hget,hset,hgetall 等。

Hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以Hash数据结构来存储用户信息，商品信息等等。比如下面我就用 hash 类型存放了我本人的一些信息：

```
key=JavaUser293847
value={
  “id”: 1,
  “name”:
  “SnailClimb”, “age”:
  22,
  “location”: “Wuhan, Hubei”
}
```

3.List

> 常用命令: lpush,rpush,lpop,rpop,lrange等

list 就是链表，Redis list 的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表， 消息列表等功能都可以用Redis的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。另

外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

4.Set

> 常用命令： sadd,spop,smembers,sunion 等

set 对外提供的功能与list类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在 一个set集合内的重要接口，这个也是list所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常 方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：

```
sinterstore  key1  key2          将交集存在key1内
```

5.Sorted Set

> 常用命令： zadd,zrange,zrem,zcard等

和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

举例： 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 SortedSet 结构进行存储。

6.redis 设置过期时间

Redis中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库， 这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如果假设你设置了一批 key 只能存活1个小时 ,那么接下来1小时后 ,redis是怎么对这批key进行删除的？ 定

期删除+惰性删除。

通过名字大概就能猜出这两个删除方式的意思了。

- 定期删除：redis默认是每隔 100ms 就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。注意这里是随机抽取的。为什么要随机呢？你想一想假如 redis 存了几十万个 key ，每隔100ms就遍历所有的设置过期时间的 key 的话，就会给 CPU 带来很大的负载！
- 惰性删除 :定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删除。假如你的过期key，靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那个 key，才会被redis给删除掉。这就是所谓的惰性删除，也是够懒的哈！

但是仅仅通过设置过期时间还是有问题的。我们想一下：如果定期删除漏掉了很多过期 key，然后你也没及时去查， 也就没走惰性删除 ,此时会怎么样？如果大量过期key堆积在内存里 ,导致redis内存块耗尽了。怎么解决这个问题 呢？

redis 内存淘汰机制。

7.redis 内存淘汰机制（MySQL里有2000w数据，Redis中只存

20w的数据，如何保证Redis中的数据都是热点数据？）

redis 配置文件 redis.conf 中有相关注释，我这里就不贴了，大家可以自行查阅或者通过这个网址查看：
http://download.redis.io/redis-stable/redis.conf

redis 提供 **6**种数据淘汰策略：

5. **volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
6. **volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
7. **volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
8. **allkeys-lru**：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）.
9. **allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰
10. **no-eviction**：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

备注： 关于 **redis** 设置过期时间以及内存淘汰机制，我这里只是简单的总结一下，后面会专门写一篇文章来总结！

8.redis 持久化机制（怎么保证 redis 挂掉之后再重启数据可以进行恢

复）

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后回复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重一点就是，Redis支持持久化，而且支持两种不同的持久化操作。**Redis**的一种持久化方式叫快照（**snapshotting**，**RDB**），另一种方式是只追加文件（**append-only file,AOF**）.这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自己的持久化方法。

快照（**snapshotting**）持久化（**RDB**）

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份 ,可以将快照复制到其他服务器从而创建具有相同数据的服务器副本(Redis主从结构 ,主要用来提高Redis性 能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save    900                     #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命
1
令创建快照。

save 300 10                     #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命
令创建快照。

save 60 10000                   #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命
令创建快照。
```

AOF（append-only file）持久化

与快照持久化相比，AOF持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数开启：

```
appendonly yes
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
appendfsync always #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec #每秒钟同步一次，显示地将多个写命令同步到硬盘
appendfsync no #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec选项 ，让Redis每秒同步一次AOF文件，Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操 作的时候，Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项`aof-use-rdb-preamble`开启）。

如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点，快速加载同时避免丢失过多的数据。当然缺点也是有的， AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

补充内容：**AOF** 重写

AOF重写可以产生一个新的AOF文件，这个新的AOF文件和原有的AOF文件所保存的数据库状态一样，但体积更小。

AOF重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有AOF文件进行任何读 入、分析或者写入操作。

在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建新AOF文件期间，记录服务器执行的所有写命令。当子进程完成创建新AOF文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新AOF文件的末尾，使得新旧两个AOF文件所保存的数据库状态一致。最后，服务器用新的AOF文件替换旧的AOF文件，以此来完成AOF文件重写操作

9.redis 事务

Redis 通过 MULTI、EXEC、WATCH 等命令来实现事务(transaction)功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令 请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

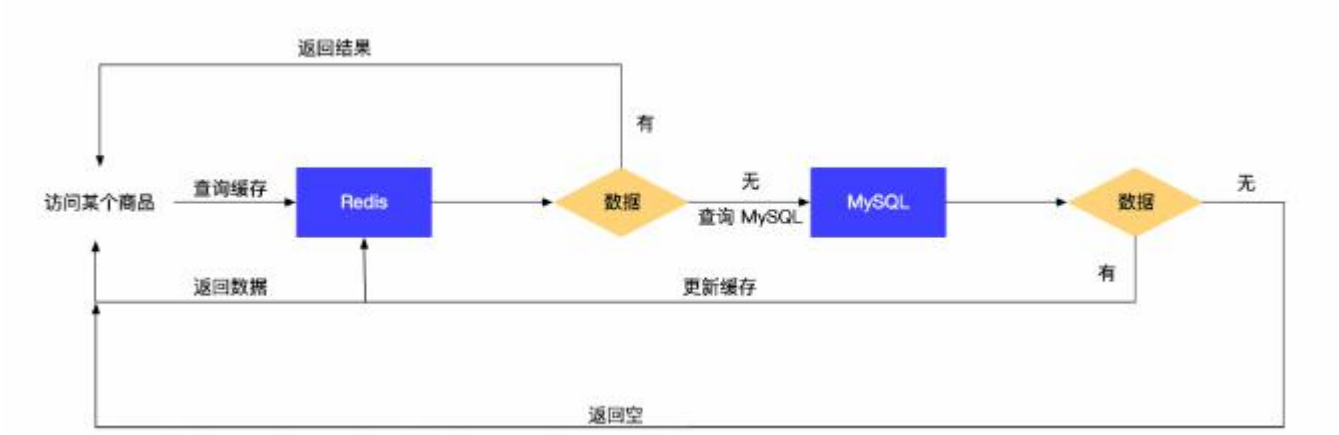
在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务总是具有原子性（Atomicity)、一致性(Consistency)和隔离性（Isolation），并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性（Durability）。

10.Redis 常见异常及解决方案

缓存使用过程当中，我们经常遇到的一些问题总结有四点：

10.1 缓存穿透

一般访问缓存的流程，如果缓存中存在查询的商品数据，那么直接返回。 如果缓存中不存在商品数据， 就要访问数据库。



由于不恰当的业务功能实现，或者外部恶意攻击不断地请求某些不存在的数据内存，由于缓存中没有保存该数据，导致所有的请求都会落到数据库上，对数据库可能带来一定的压力，甚至崩溃。

解决方案：

针对缓存穿透的情况， 简单的对策就是将不存在的数据访问结果， 也存储到缓存中，避免缓存访问的穿透。最终不存在商品数据的访问结果也缓存下来。有效的避免缓存穿透的风险。

10.2 缓存雪崩

当缓存重启或者大量的缓存在某一时间段失效，这样就导致大批流量直接访问数据库，对 DB 造成压力， 从而引起 DB 故障，系统崩溃。

举例来说， 我们在准备一项抢购的促销运营活动，活动期间将带来大量的商品信息、库存等相关信息的查询。 为了避免商品数据库的压力，将商品数据放入缓存中存储。 不巧的是，抢购活动期间，大量的热门商品缓存同时失效过期了，导致很大的查询流量落到了数据库之上。对于数据库来说造成很大的压力。

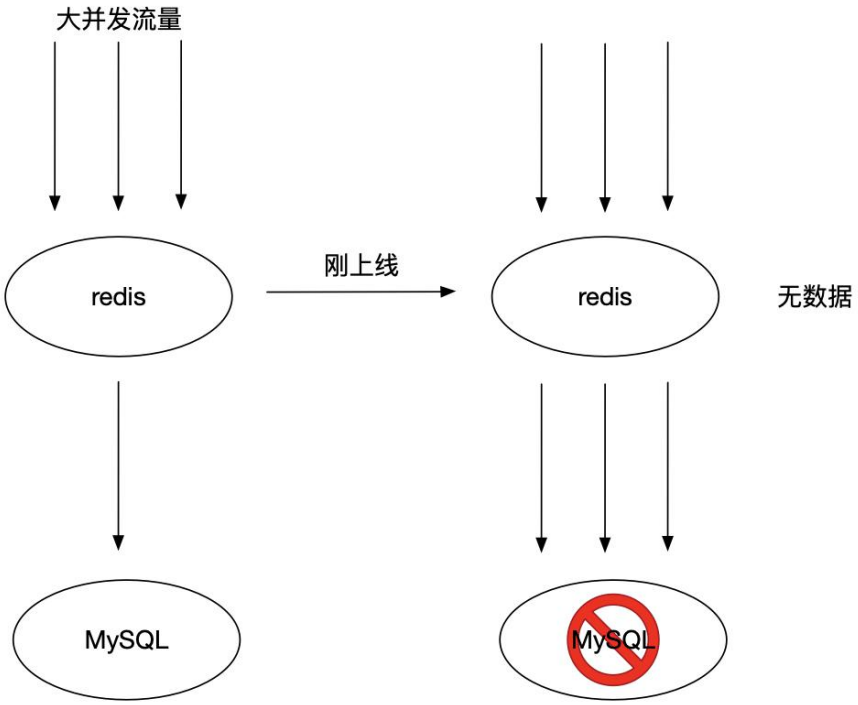
解决方案：

1. 将商品根据品类热度分类， 购买比较多的类目商品缓存周期长一些， 购买相对冷门的类目商品，缓存周期短一些；
2. 在设置商品具体的缓存生效时间的时候， 加上一个随机的区间因子， 比如说 5~10 分钟之间来随意选择失效时间；
3. 提前预估 DB 能力， 如果缓存挂掉，数据库仍可以在一定程度上抗住流量的压力

这三个策略能够有效的避免短时间内，大批量的缓存失效的问题。

10.3 缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题。用户直接查询事先被预热的缓存数据。如图所示：



如果不进行预热， 那么 Redis 初识状态数据为空，系统上线初期，对于高并发的流量，都会访问到数据库中， 对数据库造成流量的压力。

解决方案：

1. 数据量不大的时候，工程启动的时候进行加载缓存动作；
2. 数据量大的时候，设置一个定时任务脚本，进行缓存的刷新；
3. 数据量太大的时候，优先保证热点数据进行提前加载到缓存。

10.4 缓存降级

降级的情况，就是缓存失效或者缓存服务挂掉的情况下，我们也不去访问数据库。我们直接访问内存部分数据缓存或者直接返回默认数据。

举例来说：

> 对于应用的首页，一般是访问量非常大的地方，首页里面往往包含了部分推荐商品的展示信息。这些推荐商品都会放到缓存中进行存储，同时我们为了避免缓存的异常情况，对热点商品数据也存储到了内存中。同时内存中还保留了一些默认的商品信息。如下图所示：

降级一般是有损的操作，所以尽量减少降级对于业务的影响程度。

11.分布式环境下常见的应用场景

11.1 分布式锁

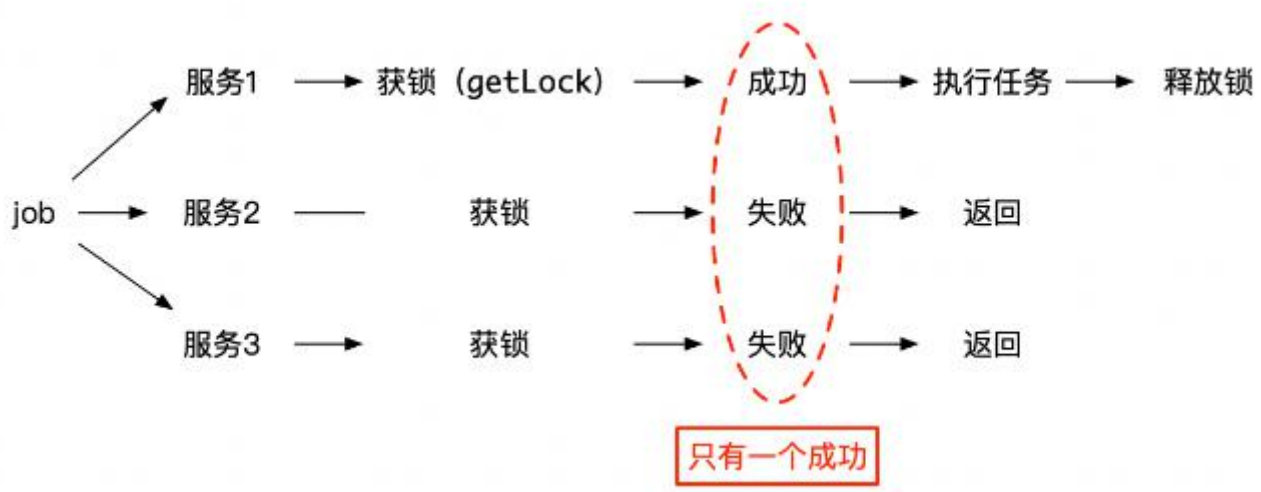
当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的操作或者访问。 与之对应有线程锁，进程锁。

分布式锁可以避免不同进程重复相同的工作，减少资源浪费。 同时分布式锁可以避免破坏数据正确性的发生， 例如多个进程对同一个订单操作，可能导致订单状态错误覆盖。应用场景如下。

11.1.1 定时任务重复执行

随着业务的发展，业务系统势必发展为集群分布式模式。如果我们需要一个定时任务来进行订单状态的统计。比如每 15 分钟统计一下所有未支付的订单数量。那么我们启动定时任务的时候，肯定不能同一时刻多个业务后台服务都去执行定时任务， 这样就会带来重复计算以及业务逻辑混乱的问题。

这时候，就需要使用分布式锁，进行资源的锁定。那么在执行定时任务的函数中，首先进行分布式锁的获取，如果可以获取的到，那么这台机器就执行正常的业务数据统计逻辑计算。如果获取不到则证明目前已有其他的服务进程执行这个定时任务，就不用自己操作执行了，只需要返回就行了。如下图所示：



11.1.2 避免用户重复下单

分布式实现方式有很多种：

1. 数据库乐观锁方式
2. 基于 Redis 的分布式锁
3. 基于 ZK 的分布式锁

咱们这篇文章主要是讲 Redis，那么我们重点介绍基于 Redis 如何实现分布式锁。

分布式锁实现要保证几个基本点。

1. 互斥性：任意时刻，只有一个资源能够获取到锁。
2. 容灾性：能够在未成功释放锁的的情况下，一定时限内能够恢复锁的正常功能。
3. 统一性：加锁和解锁保证同一资源来进行操作。

加锁代码演示：

```
public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String traceId, int expireTime) {
        SetParams setParams = new SetParams();
        setParams.ex(expireTime);
        setParams.nx();
        String result = jedis.set(lockKey, traceId, setParams);
        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;
```

```
    }
```

解锁代码演示：

```
    public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String traceId) {


            String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

            Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(traceId));

            if (RELEASE_SUCCESS.equals(result)) {

                return true;

            }

            return false;

        }
```

11.2 分布式自增 ID

应用场景

随着用户以及交易量的增加， 我们可能会针对用户数据，商品数据，以及订单数据进行分库分表的操作。这时候由于进行了分库分表的行为，所以 MySQL 自增 ID 的形式来唯一表示一行数据的方案不可行了。 因此需要一个分布式 ID 生成器，来提供唯一 ID 的信息。

实现方式

通常对于分布式自增 ID 的实现方式有下面几种：

1. 利用数据库自增 ID 的属性
2. 通过 UUID 来实现唯一 ID 生成
3. Twitter 的 SnowFlake 算法
4. 利用 Redis 生成唯一 ID

在这里我们自然是说 Redis 来实现唯一 ID 的形式了。使用 Redis 的 INCR 命令来实现唯一 ID。

Redis 是单进程单线程架构 ,不会因为多个取号方的 INCR 命令导致取号重复。因此 ,基于 Redis 的 INCR 命令实现序列号的生成基本能满足全局唯一与单调递增的特性。

代码相对简单， 不做详细的展示了。

12.Redis 集群模式

作为缓存数据库，肯定要考虑缓存服务稳定性相关的保障机制。

持久化机制就是一种保障方式。持久化机制保证了 Redis 服务器重启的情况下也不会损失（或少量损失）数据，因为持久化会把内存中数据保存到硬盘上，重启会从硬盘上加载数据。

随着 Redis 使用场景越来越多，技术发展越来越完善，在 Redis 整体服务上的容错、扩容、稳定各个方面都需要不断优化。 因此在 Redis 的集群模式上也有不同的搭建方式来应对各种需求。

总结来说，Redis 集群模式有三种：

- 主从模式
- 哨兵模式
- Cluster 集群模式

12.1 主从模式

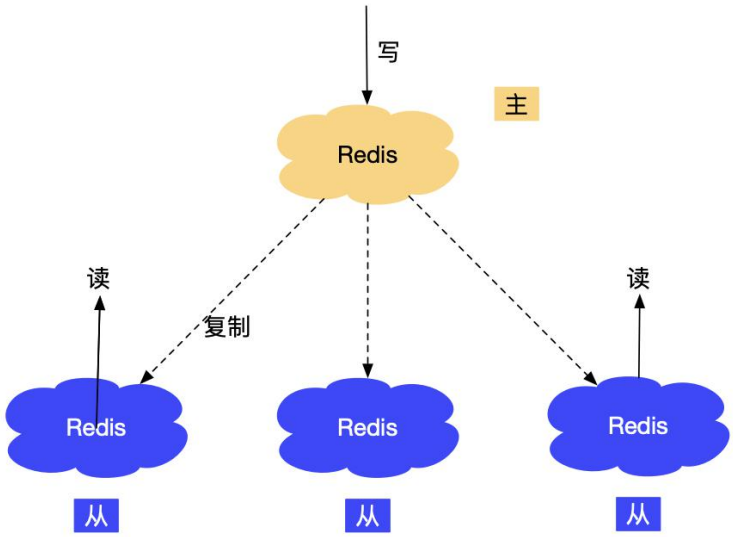
为了 Redis 服务避免单点故障，通常的做法是将 Redis 数据复制多个副本以部署在不同的服务器上。这样即使有一台服务器出现故障，其他服务器依然可以继续提供服务。为此，Redis 提供了复制（ replication ）功能，可以实现当一台数据库中的数据更新后，自动将更新的数据同步到其他数据库上。

Redis 服务器分为两类：一类是主数据库（Master），另一类是从数据库（Slave）。

主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。

从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

如图所示：



优点

1. 一个主，可以有多个从，并以非阻塞的方式完成数据同步；
2. 从服务器提供读服务，分散主服务的压力，实现读写分离；
3. 从服务器之前可以彼此连接和同步请求，减少主服务同步压力。

缺点

1. 不具备容错和恢复功能，主服务存在单点风险；
2. Redis 的主从复制采用全量复制，需要服务器有足够的空余内存；
3. 主从模式较难支持在线扩容。

12.2 哨兵模式

Redis 提供的 sentinel（哨兵）机制，通过 sentinel 模式启动redis后，自动监控 Master/Slave 的运行状态，基本原理是：**心跳机制 + 投票裁决**。

简单来说，哨兵的作用就是监控 Redis 系统的运行状况。它的功能包括以下两个：

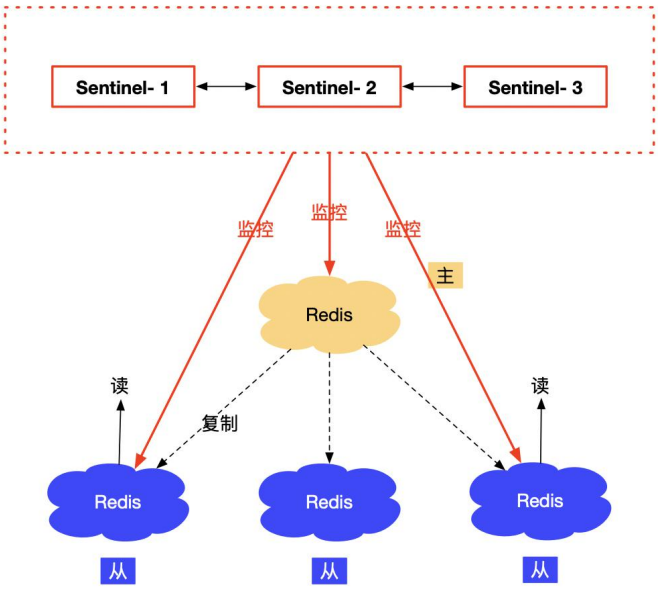
1. 监控主数据库和从数据库是否正常运行；
2. 主数据库出现故障时自动将从数据库转换为主数据库。

哨兵模式主要有下面几个内容：

- **监控（ Monitoring ）**：Sentinel 会定期检查主从服务器是否处于正常工作状态。
- **提醒（ Notification ）**：当被监控的某个 Redis 服务器出现异常时，Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

- **自动故障迁移（Automatic failover）**：当一个主服务器不能正常工作时，Sentinel 会开始一次自动故障迁移操作，它会将失效主服务器的其中一个从服务器升级为新的主服务器，并让失效主服务器的其他从服务器改为复制新的主服务器；当客户端试图连接失效的主服务器时，集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

Redis Sentinel 是一个分布式系统，你可以在一个架构中运行多个 Sentinel 进程(progress)。



优点

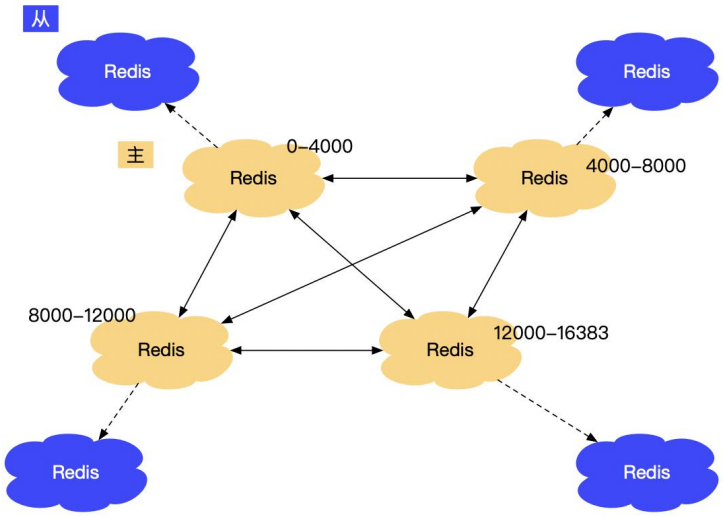
1. 哨兵模式主从可以切换，具备基本的故障转移能力；
2. 哨兵模式具备主从模式的所有优点。

缺点

1. 哨兵模式也很难支持在线扩容操作；
2. 集群的配置信息管理比较复杂。

12.3 Cluster 集群模式

Redis Cluster 是一种服务器 Sharding 技术，3.0 版本开始正式提供。采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。如图所示：



Cluster 集群结构特点：

1. Redis Cluster 所有的物理节点都映射到 [0-16383] slot 上(不一定均匀分布) , Cluster 负责维护节点、桶、值之间的关系；
2. 在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384 的值，从之前划分的 16384 个桶中选择一个；
3. 所有的 Redis 节点彼此互联 (PING-PONG 机制) ，内部使用二进制协议优化传输效率；
4. 超过半数的节点检测到某个节点失效时则判定该节点失效；
5. 使用端与 Redis 节点链接,不需要中间 proxy 层，直接可以操作，使用端不需要连接集群所有节点，连接集群中任何一个可用节点即可。

优点

1. 无中心架构，节点间数据共享，可动态调整数据分布；
2. 节点可动态添加删除，扩展性比较灵活；
3. 部分节点异常，不影响整体集群的可用性。

缺点

1. 集群实现比较复杂；
2. 批量操作指令 (mget、mset 等) 支持有限；
3. 事务操作支持有限。

13.如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁(zookeeper 和 redis 都可以实现分布式锁)。(如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能)

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所

以首推Zookeeper。参考：

- https://www.jianshu.com/p/8bddd381de06

14.如何保证缓存与数据库双写时的数据一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如　何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的　　情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致　的情况串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

参考

https://zhuanlan.zhihu.com/p/63428500