Redis缓存穿透，缓存击穿，缓存雪崩

题目难度：★★★★

知识点标签：Redis缓存穿透，缓存击穿，缓存雪崩

学习时长：20分钟

题目描述

描述一下你对Redis缓存穿透，缓存击穿，缓存雪崩的理解，以及简单描述一下每种情况的解决方案

解题思路

> 面试官问题可以从几个方面来回答：三种情况发生的概念描述与原因，解决方案

缓存穿透，缓存击穿，缓存雪崩

- 缓存穿透：key对应的数据在数据源并不存在，每次针对此key的请求从缓存获取不到，请求都会到数据源，从而可能压垮数据源。比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。
- 缓存击穿：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，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。
- 缓存雪崩：当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

缓存穿透，缓存击穿，缓存雪崩解决方案

1.缓存穿透解决方案

一个一定不存在缓存及查询不到的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，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。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。**另外也有一个**更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

```
// 伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
```

```
    String cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    }

    cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        //数据库查询不到，为空
        cacheValue = GetProductListFromDB();
        if (cacheValue == null) {
          //如果发现为空，设置个默认值，也缓存起来
          cacheValue = string.Empty;
        }
        CacheHelper.Add(cacheKey, cacheValue, cacheTime);
        return cacheValue;
    }
}
```

2.缓存击穿解决方案

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：缓存被“击穿”的问题。

使用互斥锁(mutex key)

业界比较常用的做法，是使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。

SETNX，是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现锁的效果。

```
// 伪代码
public String get(key) {
    String value = redis.get(key);
    if (value == null) { //代表缓存值过期
      //设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db
      if (redis.setnx(key_mutex, 1, 3 * 60) == 1) {  //代表设置成功
        value = db.get(key);
        redis.set(key, value, expire_secs);
        redis.del(key_mutex);
      } else {  //这个时候代表同时候的其他线程已经load db并回设到缓存了，这时候重试获取缓存值即可
        sleep(50);
        get(key);  //重试
      }
    } else {
```

```
        return value;
    }
}
```

3.缓存雪崩解决方案

缓存失效时的雪崩效应对底层系统的冲击非常可怕！大多数系统设计者考虑用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开，比如我们可以在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。

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

```
// 加锁排队伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    String lockKey = cacheKey;

    String cacheValue = CacheHelper.get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        synchronized(lockKey) {
            cacheValue = CacheHelper.get(cacheKey);
            if (cacheValue != null) {
                return cacheValue;
            } else {
              //这里一般是sql查询数据
                cacheValue = GetProductListFromDB();
                CacheHelper.Add(cacheKey, cacheValue, cacheTime);
            }
        }
        return cacheValue;
    }
}
```

总结

针对业务系统，永远都是具体情况具体分析，没有最好，只有最合适。

于缓存其它问题，缓存满了和数据丢失等问题，大伙可自行学习。最后也提一下三个词LRU、RDB、AOF，通常我们采用LRU策略处理溢出，Redis的RDB和AOF持久化策略来保证一定情况下的数据安全。