- 什么缓存：缓存即为将硬盘中的常用数据加载到内存中以提供程序读取。
- 为什么使用缓存：缓存是用来提高程序运行速度（直接从内存中读数据，比较快）、减轻数据库压力（对数据库请求减少，压力降低）。但是使用缓存存在缓存穿透、缓存击穿、缓存雪崩的问题。
- 何为缓存穿透、缓存击穿、缓存雪崩：
    - 缓存穿透：缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。
    - 缓存击穿：缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。
    - 缓存雪崩：缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。
    - 解决方案1：通过加锁减轻数据库的瞬时压力。但是当我们通过加一把锁来解决的话会降低应用的性能，因此我们可以将锁的粒度细化。锁的粒度划分条件可以根据系统实际来，例如给不同地域发送过来的请求加不同的锁。对于缓存，我们可以模仿mybatis的BlockingCache的做法，对不同的key加不同的锁，即对于某个key同一时间只有一个请求能访问。
    - 缓存击穿和缓存雪崩解决方案：
        - 控制瞬时请求量（即通过锁控制同一时间只能有一个请求访问key对应的value）

- 从缓存本身解决（即对每个缓存设置不同的过期时间，避免它们同时失效）
  - 缓存穿透解决方案：前段校验

在做缓存时为什么要使用缓存淘汰策略：因为一般缓存都是通过Map等容器存储，而java中的容器放满以后会自动扩容，所以我们要设定一个策略使得容器中的缓存数量有限（即只对热点数据进行缓存），当缓存数量达到一定的数量以后就要根据不同的策略去淘汰其中的旧数据，这样缓存数量才能保持在一个定值。如果我们不加限制的话缓存的数量越来越多那就没什么意义了，查询不仅会变慢而且还可能会撑爆内存。

JDK基本数据包装类是如何做缓存的？通过static代码块初始化缓存，我们在代码开发如果不使用spring等框架的时候也可以参考这种做法：

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty( s: "java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

静态常量存放在方法区中

static代码块在类加载的时候会被执行，利用这个机制实现缓存

构造方法私有，意味着只能在Integer中实例化它

缓存：

- **网站性能优化第一定律：优先考虑使用缓存优化性能**
- **Mark老师的推论：缓存离用户越近越好**

缓存的基本原理和本质：缓存是将数据存在访问速度较高的介质中。可以减少数据访问的时间，同时避免重复计算。

合理使用缓冲的准则：

- **频繁修改的数据，尽量不要缓存，读写比2:1以上才有缓存的价值。**
- **缓存一定是热点数据。**
- **应用需要容忍一定时间的数据不一致。**
- **缓存可用性问题，一般通过热备或者集群来解决。**
- **缓存预热，新启动的缓存系统没有任何数据，可以考虑将一些热点数据提前加载到缓存系统。**
- **解决缓存击穿：1、布隆过滤器（能判断出一定不存在，但不能确定一定存在）。2、把不存在的数据也缓存起来，比如有请求总是访问key = 23的数据，但是这个key = 23的数据在系统中不存在，当访问次数超过N次后可以考虑在缓存中构建一个(key=23 value = null)的数据。**

分布式缓存与一致性哈希，以集群的方式提供缓存服务，有两种实现：

- **余数hash算法：**
  - **需要更新同步的分布式缓存，所有的服务器保存相同的缓存数据，带来的问题就是，缓存的数据量受限制，其次，数据要在所有的机器上同步，代价很大。**
  - **每台机器只缓存一部分数据，然后通过一定的算法选择缓存服务器。常见的余数hash算法存在当有服务器上下线的时候，大量缓存数据重建的问题。所以提出了一致性哈希算法。**
- **一致性哈希：**
  - **首先求出服务器（节点）的哈希值，并将其配置到0～2^32的圆（continuum）上。**

- 然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
- 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过2^32仍然找不到服务器，就会保存到第一台服务器上。
- 一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。
- 一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题，此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，**一致性哈希算法引入了虚拟节点机制**，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器ip或主机名的后面增加编号来实现。例如，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

同步和异步，阻塞和非阻塞：

- 同步和异步关注的是结果消息的通信机制：
  - 同步：同步的意思就是调用方需要主动等待结果的返回
  - 异步：异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。**常见的异步手段有多线程、消息队列。**
- 阻塞和非阻塞主要关注的是等待结果返回调用方的状态：
  - 阻塞：是指结果返回之前，当前线程被挂起，不做任何事。
  - 非阻塞：是指结果在返回之前，线程可以做一些其他事，不会被挂起。

- 同步阻塞：同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就在店里面一直等，期间不做任何事(包括看手机)，等着商家进货，直到有货为止，这个效率很低。jdk里的BIO就属于 同步阻塞。
- 同步非阻塞：同步非阻塞在编程中可以抽象为一个轮询模式，你去了商店之后，发现衣服卖完了，这个时候不需要傻傻的等着，你可以去其他地方比如奶茶店，买杯水，但是你还是需要时不时的去商店问老板新衣服到了吗。jdk里的NIO就属于 同步非阻塞。
- 异步阻塞：异步阻塞这个编程里面用的较少，有点类似你写了个线程池,submit然后马上future.get()，这样线程其实还是挂起的。有点像你去商店买衣服，这个时候发现衣服没有了，这个时候你就给老板留给电话，说衣服到了就给我打电话，然后你就守着这个电话，一直等着他响什么事也不做。这样感觉的确有点傻，所以这个模式用得比较少。
- 异步非阻塞：好比你去商店买衣服，衣服没了，你只需要给老板说这是我的电话，衣服到了就打。然后你就随心所欲的去玩，也不用操心衣服什么时候到，衣服一到，电话一响就可以去买衣服了。jdk里的AIO就属于异步

分布式锁：

- 基于redis和zookeeper实现分布式锁的细节以及注意事项
  (https://mp.weixin.qq.com/s/VfhvEcxi-9T8rU8bgvUdjA)

服务雪崩、降级、熔断（https://www.cnblogs.com/rjzheng/p/10340176.html）

- 服务雪崩：一个服务失败，导致整条调用链路的服务都失败的情形，我们称之为服务雪崩。
- 服务熔断：当下游的服务因为某种原因突然变得不可用或响应过慢，上游服务为了保证自己整体服务的可用性，不再继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。
- 服务降级（两种场景）：
  - 当下游的服务因为某种原因响应过慢，下游服务主动停掉一些不太重要的业务，释放出服务器资源，增加响应速度。当业务流程没法简化，也没有次要功能可关的时候，只能降低一致性了，即将核心业务流程的同步改异步，将强一致性改最终一致性！

- 当下游的服务因为某种原因**不可用**，**上游主动调用本地的一些降级逻辑**，避免卡顿，迅速返回给用户。

- 理解熔断与降级：服务降级有很多种降级方式，如**开关降级、限流降级、熔断降级**，服务熔断属于降级方式的一种。从实现上来说，熔断和降级必定是一起出现。因为当发生下游服务不可用的情况，这个时候为了对最终用户负责，就需要进入上游的降级逻辑了。因此，将熔断降级视为降级方式的一种，也是可以说的通的。