1 缓存的概念

1.1 外存

外储存器是指除计算机内存及 CPU 缓存以外的储存器，此类储存器一般断电后仍然能保存数据。

常见的外存储器有硬盘、软盘、光盘、U 盘等，一般的软件都是安装在外存中（windows 系统指的是 CDEF 盘, Linux 系统指的是挂载点）。

1.2 内存

内存是计算机中重要的部件之一，它是与 CPU 进行沟通的桥梁。计算机中所有程序的运行都是在内存中进行的，因此内存的性能对计算机的影响非常大。

内存(Memory)也被称为内存储器，其作用是用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。只要计算机在运行中，CPU 就会把需要运算的数据调到内存中进行运算，当运算完成后 CPU 再将结果传送出来，内存的运行也决定了计算机的稳定运行，此类储存器一般断电后数据就会被清空。

1.3 缓存

缓存就是把一些外存上的数据保存到内存上而已，怎么保存到内存上呢，我们运行的所有程序，里面的变量值都是放在内存上的，所以说如果要想使一个值放到内存上，实质就是在获得这个变量之后，用一个生存期较长的变量存放你想存放的值，在 java 中一些缓存一般都是通过 map 集合来做的。

广义的缓存是把一些慢存（较慢的外存）上的数据保存到快存（较快的存储）上，

简单讲就是，如果某些资源或者数据会被频繁的使用，而这些资源或数据存储在系统外部，比如数据库、硬盘文件等，那么每次操作这些数据的时候都从数据库或者硬盘上去获取，速度会很慢，会造成性能问题（系统停工待料）。于是我们把这些数据冗余一份到快存里面，每次操作的时候，先到快存里面找，看有没有这些数据，如果有，那么就直接使用，如果没有那么就获取它，并复制一份到快存中，下一次访问的时候就可以直接从快存中获取。从而节省大量的时间，可以看出，缓存是一种典型的空间换时间的方案。

生活中这样的例子处处可见，举例：

- **举例 1** --CPU--L1/L2--内存--磁盘

CPU 需要数据时先从 L1/L2 中读取，如果没有到内存中找，如果还没有会到磁盘上找。

- **举例 2** --Maven

还有如用过 Maven 的朋友都应该知道，我们找依赖的时候，先从本机仓库找，再从本地服务器仓库找，最后到远程仓库服务器找。

- **举例 3** --京东仓储

还有如京东的物流为什么那么快？他们在各个地都有分仓库，如果该仓库有货物那么送货的速度是非常快的。

- **举例 4** --数据库中的索引，也是以空间换取时间，缓存也是以空间换取时间。

1.4 重要的指标

1.4.1 缓存命中率

--表明缓存是否运行良好的

即【从缓存中读取数据的次数】与【总读取次数】的比率，命中率越高越好：

命中率 = 从缓存中读取次数 / (总读取次数[从缓存中读取次数 + 从慢速设备上读取的次数])

Miss 率 = 没有从缓存中读取的次数 / (总读取次数[从缓存中读取次数 + 从慢速设备上读取的次数])

这是一个非常重要的监控指标，如果做缓存一定要健康这个指标来看缓存是否工作良好。

1.4.2 移除策略

--不同的移除策略实际上看的是不同的指标

即如果缓存满了，从缓存中移除数据的策略；常见的有 LFU、LRU、FIFO：

FIFO（First In First Out）：先进先出算法，即先放入缓存的先被移除；

LRU（Least Recently Used）：最久未使用算法，使用时间距离现在最久的那个被移除；

LFU（Least Frequently Used）：最近最少使用算法，一定时间段内使用次数（频率）最少的那个被移除。

TTL（Time To Live ）

存活期，即从缓存中创建时间点开始直到它到期的一个时间段（不管在这个时间段内有没有访问都将过期）

TTI（Time To Idle）

空闲期，即一个数据多久没被访问将从缓存中移除的时间。

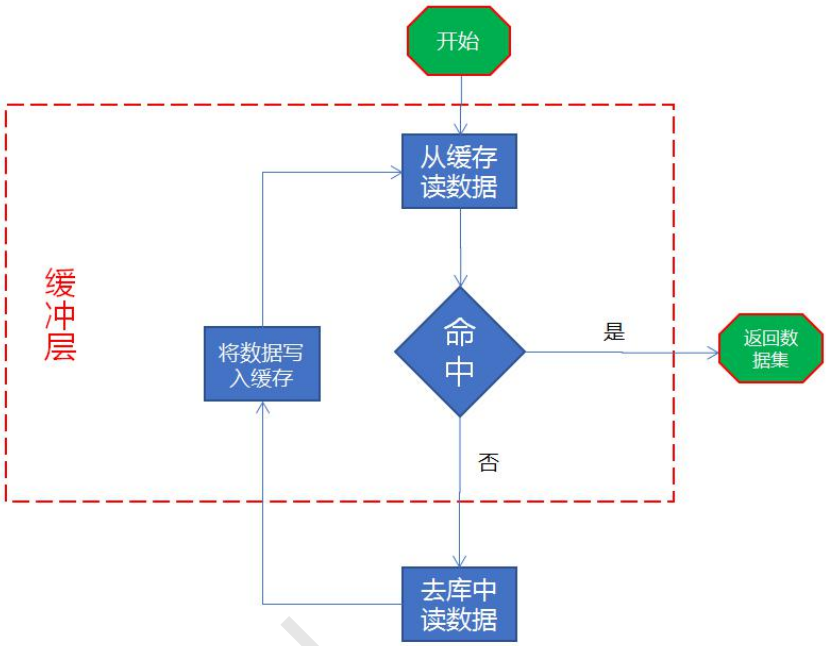
实际设计缓存时以上重要指标都应该考虑进去，当然根据实际需求可能有的指标并不会采用进设计去

2 缓存在 java 中的实现

在 Java 中，我们一般对调用方法进行缓存控制，比如我调用"findUserById(Long id)"，那么我应该在调用这个方法之前先从缓存中查找有没有，如果没有再掉该方法如从数据库加载用户，然后添加到缓存中，下次调用时将会从缓存中获取到数据。Java 中广泛使用的分布式缓存 Redis

2.1 缓存逻辑流程

流程图如下：



2.2 逻辑流程代码：

```
@Override
public Provinces detail(String provinceid) {
    Provinces provinces = null;

    //在redis查询
    provinces = (Provinces)redisTemplate.opsForValue().get(provinceid);
    if (null != provinces){
        //滑动过期
        redisTemplate.expire(provinceid,20000, TimeUnit.MILLISECONDS);
        System.out.println("缓存中得到数据");
        return provinces;
    }

    provinces = super.detail(provinceid);
    if (null != provinces){
        redisTemplate.opsForValue().set(provinceid,provinces);//set缓存
        redisTemplate.expire(provinceid,20000, TimeUnit.MILLISECONDS);
    }

    return provinces;
}
```

3 基于注解的 Cache

Spring 3.1 起，提供了基于注解的对 Cache 的支持。使用 Spring Cache 的好处：

- 基于注解，代码清爽简洁；
- 基于注解也可以实现复杂的逻辑；

- 可以对缓存进行回滚；

Spring Cache 并非具体的缓存技术，而是基于各种缓存产品（如 Guava、EhCache、Redis 等）共性进行的一层封装，结合 SpringBoot 的开箱即用的特性用起来会非常方便，因为 Spring Cache 通过注解隔离了具体的缓存产品，让用户更加专注于应用层面。具体的底层缓存技术究竟采用了 Guava、EhCache 还是 Redis，只需要简单的配置就可以实现方便的切换。

3.1 设计理念

正如 Spring 框架的其它服务一样，Spring cache 首先是提供了一层抽象，核心抽象主要体现在两个接口上

org.springframework.cache.Cache：代表缓存本身

org.springframework.cache.CacheManager：代表对缓存的处理和管理

抽象的意义在于屏蔽实现细节的差异和提供扩展性，Cache 的抽象解耦了缓存的使用和缓存的后端存储，方便后续更换存储。

3.2 使用 Spring Cache

分三步：

- 声明缓存
- 开启 Spring 的 cache 功能
- 配置后端的存储

3.2.1 声明缓存

@Cacheable("books")

public Book findBook(ISBN isbn) {...}

用法很简单，在方法上添加@cacheable 等注解，表示缓存该方法的结果。

当方法有被调用时 ,先检查 cache 中有没有针对该方法相同参数的调用发生过 ,如果有，从 cache 中查询并返回结果。如果没有 ,则执行具体的方法逻辑 ,并把结果缓存到 cache 中。当然这一系列逻辑对于调用者来说都是透明的。

其它的缓存操作的注解包含如下（详细说明可参见官方文档）:

@Cacheable triggers cache population

@CacheEvict triggers cache eviction

@CachePut updates the cache without interfering with the method execution

@Caching regroups multiple cache operations to be applied on a method

@CacheConfig shares some common cache-related settings at class-level

3.2.2 开启 Spring Cache 的支持

<cache:annotation-driven />

或者使用注解@EnableCaching 的方式

3.2.3 配置缓存后端存储

SpringCache 包本身提供了一个 manager 实现，用法如下：

```
@Bean
public CacheManager cacheManager() {
    //jdk 里，内存管理器
    SimpleCacheManager cacheManager = new SimpleCacheManager();
    cacheManager.setCaches(Collections.singletonList(new ConcurrentMapCache("province
    return cacheManager;
}
```

更常用的，RedisCacheManager（来自于 Spring Data Redis 项目），用法如下：

```
@Bean
public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {
    return RedisCacheManager
            .builder(connectionFactory)
            .cacheDefaults(
                    RedisCacheConfiguration.defaultCacheConfig()
                    .entryTtl(Duration.ofSeconds(20))) //缓存时间绝对过期时间 20s
            .transactionAware()
            .build();
}
```

3.2.4 缓存 key 的生成

我们都知道缓存的存储方式一般是 key value 的方式，那么在 Spring cache 里，key 是如何被设置的呢，在这里要引入 KeyGenerator，它负责 key 的生成策略，只需要按自己的业务，为 params 设计一套生成 key 的规则即可（简单地连接各个参数值）：

```
//key 的生成，springcache 的内容，跟具体实现缓存器无关
@Bean
public KeyGenerator keyGenerator() {
    return new KeyGenerator() {
        @Override
        public Object generate(Object target, Method method, Object... params) {

            StringBuilder sb = new StringBuilder();
            sb.append(target.getClass().getSimpleName());
            sb.append(method.getName());
            for (Object obj : params) {
                sb.append(obj.toString());
            }
            return sb.toString();
        }
    };
}
```

3.3 注解风格说明

3.3.1 @CachePut 注解说明

```
public @interface CachePut {

    String[] value();                    //缓存的名字，可以把数据写到多个缓存

    String key() default "";             //缓存 key，如果不指定将使用默认的 KeyGenerator 生成，后边介绍

    String condition() default ""; //满足缓存条件的数据才会放入缓存，condition 在调用方法之前和之后都会判断

    String unless() default "";      //用于否决缓存更新的，不像 condition，该表达
```

只在方法执行之后判断，此时可以拿到返回值 result 进行判断了

}

@CachePut 注解使用

应用到写数据的方法上，如新增/修改方法，调用方法时会自动把相应的数据放入缓存。

比如下面调用该方法时，会把 user.id 作为 key，返回值作为 value 放入缓存

3.3.2 @CacheEvict 注解说明

```
public @interface CacheEvict {

    String[] value();                          //请参考@CachePut

    String key() default "";                   //请参考@CachePut

    String condition() default "";             //请参考@CachePut

    boolean allEntries() default false;        //是否移除所有数据
```

boolean beforeInvocation() default false;//是调用方法之前移除/还是调用之后移除

}

@CacheEvict 注解使用

应用到移除数据的方法上，如删除方法，调用方法时会从缓存中移除相应的数据

3.3.3 @Cacheable 注解说明

public @interface Cacheable {

String[] value(); //请参考@CachePut

String key() default ""; //请参考@CachePut

String condition() default "";//请参考@CachePut

String unless() default ""; //请参考@CachePut

}

@Cacheable 注解使用

应用到读取数据的方法上，即可缓存的方法，如查找方法：先从缓存中读取，如果没有再调用方法获取数据，然后把数据添加到缓存中

4 缓存带来的一致性问题

读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：数据库和缓存更新，就容易出现缓存(Redis)和数据库（MySQL）间的数据一致性问题。

讨论一致性问题之前，先来看一个更新的操作顺序问题：

先删除缓存，再更新数据库

问题：同时有一个请求 A 进行更新操作，一个请求 B 进行查询操作。可能出现:

（1）请求 A 进行写操作（key = 1 value = 2），先删除缓存 key = 1 value = 1

（2）请求 B 查询发现缓存不存在

（3）请求 B 去数据库查询得到旧值 key = 1 value = 1

（4）请求 B 将旧值写入缓存 key = 1 value = 1

（5）请求 A 将新值写入数据库 key = 1 value = 2

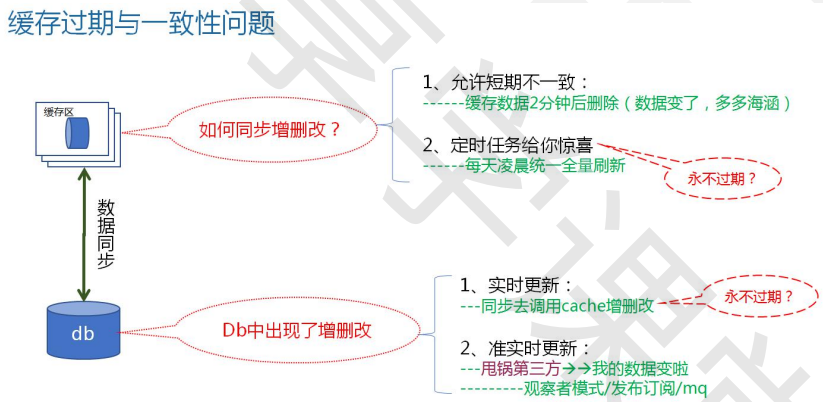
缓存中数据永远都是脏数据

我们比较推荐操作顺序：

先删除缓存，再更新数据库，再删缓存（双删，第二次删可异步延时）

```
public void write(String key,Object data){
 redis.delKey(key);
 db.updateData(data);
 Thread.sleep(500);
 redis.delKey(key);
 }
```

接下来，看一看缓存同步的一些方案，见下图：



4.1 数据实时同步更新

更新数据库同时更新缓存，使用缓存工具类和或编码实现。

优点：数据实时同步更新，保持强一致性

缺点：代码耦合，对业务代码有侵入性

4.2 数据准实时更新

准一致性，更新数据库后，异步更新缓存，使用观察者模式/发布订阅/MQ 实现；

优点：数据同步有较短延迟 ，与业务解耦

缺点：实现复杂，架构较重

4.3 缓存失效机制

弱一致性，基于缓存本身的失效机制

优点：实现简单，无须引入额外逻辑

缺点：有一定延迟，存在缓存击穿/雪崩问题

4.4 定时任务更新

最终一致性，采用任务调度框架，按照一定频率更新

优点：不影响正常业务

缺点：不保证一致性，依赖定时任务

5 缓存穿透、缓存击穿、缓存雪崩及解决方案

5.1 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

5.2 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：

此时需要对数据库的查询操作，加锁 ---- lock （因考虑到是对同一个参数数值上一把锁，此处 synchronized 机制无法使用）

加锁的标准流程代码如下（一样解决击穿的问题）：

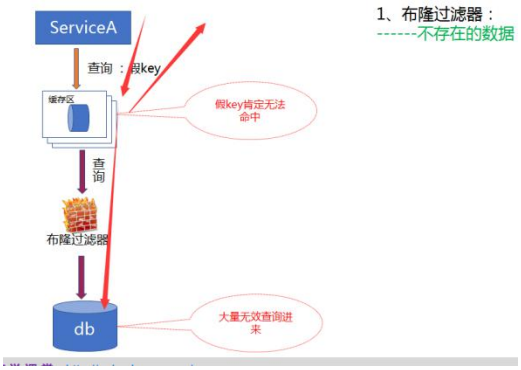
```
//2.加锁排队，阻塞式锁
doLock(provinceid);
try{//第二个线程进来了
    // 一次只有一个线程
     //双重校验，不加也没关系，无非是多刷几次库
    valueWrapper = cm.getCache(CACHE_NAME).get(provinceid);//第二个线程，能从缓存里拿到值？
    if (valueWrapper != null) {
        logger.info("缓存中得到数据");
        return (Provinces) (valueWrapper.get());//第二个线程，这里返回
    }

    Provinces provinces = super.detail(provinceid);
    // 3.从数据库查询的结果不为空，则把数据放入缓存中，方便下次查询
    if (null != provinces){
        cm.getCache(CACHE_NAME).put(provinceid, provinces);
    }
    return provinces;
}catch(Exception e){
    return null;
}finally{
    //4.解锁
    releaseLock(provinceid);
}
```

5.3 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。

解决方案：使用布隆过滤器



（1）布隆过滤器的使用方法，类似 java 的 SET 集合，只不过它能以更小的内存，存储更大的数据

类似以下伪代码

```
SET set = new HashSET();   //创建布隆过滤器

初始化加载业务数据

set.add(id)

查询

query ( id ) {

      set.contain ( id ) ==true ---》  去查数据库

}
```

（2）对应的生产代码，使用如下：

```
@PostConstruct //对象创建后，自动调用本方法          此标签指定，对象创建后，自动拉起运行
public void init(){//在bean初始化完成后，实例化bloomFilter,并加载数据
    List<Provinces> provinces = this.list();          查询业务数据

    //当成一个SET----- 占内存，比hashset占得小很多          指定过滤器容量
    bf = BloomFilter.create(Funnels.stringFunnel(Charsets.UTF_8), provinces.size());// 32个
    for (Provinces p : provinces) {
        bf.put(p.getProvinceid());          业务id放入过滤器
    }
}

if(!bf.mightContain(provinceid)){          类似set判断是否包含id
    System.out.println("非法访问--------"+System.currentTimeMillis());
    return null;
}
```