

# Gulimall

## 缓存与分布式锁



# 一、缓存

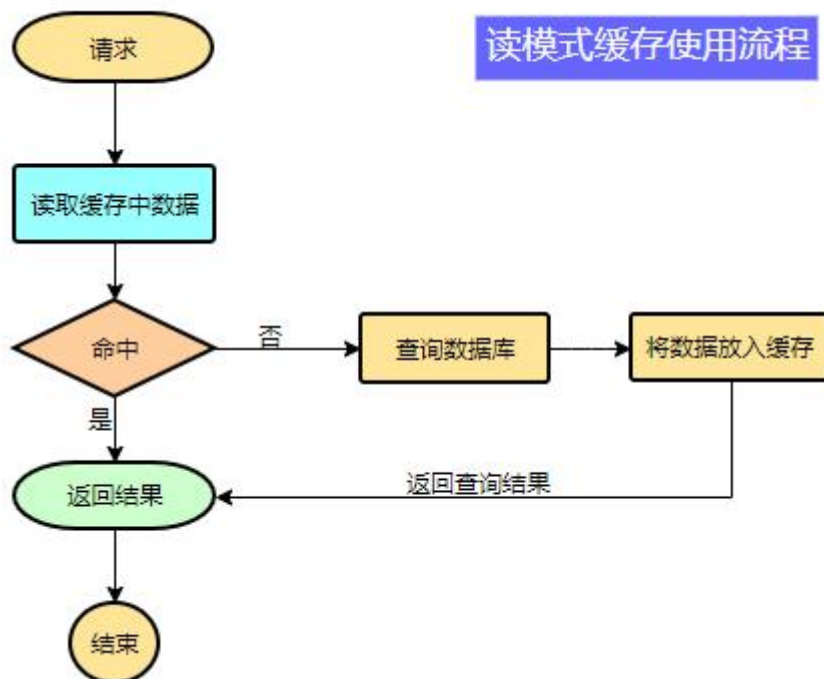
## 1、缓存使用

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

**哪些数据适合放入缓存？**

- 即时性、数据一致性要求不高的
- 访问量大且更新频率不高的数据（读多，写少）

举例：电商类应用，商品分类，商品列表等适合缓存并加一个失效时间(根据数据更新频率来定)，后台如果发布一个商品，买家需要 5 分钟才能看到新的商品一般还是可以接受的。



```

data = cache.load(id);//从缓存加载数据
if(data == null){
    data = db.load(id);//从数据库加载数据
    cache.put(id,data);//保存到 cache 中
}
return data;
    
```

**注意：**在开发中，凡是放入缓存中的数据我们都应该指定过期时间，使其可以在系统即使没有主动更新数据也能自动触发数据加载进缓存的流程。避免业务崩溃导致的数据永久不一致问题。

## 2、整合 redis 作为缓存

### 1、引入 redis-starter

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

### 2、配置 redis

```
spring:
  redis:
    host: 192.168.56.10
    port: 6379
```

### 3、使用 RedisTemplate 操作 redis

```
@Autowired
StringRedisTemplate stringRedisTemplate;
@Test
public void testStringRedisTemplate(){
    ValueOperations<String, String> ops = stringRedisTemplate.opsForValue();
    ops.set("hello", "world_" + UUID.randomUUID().toString());
    String hello = ops.get("hello");
    System.out.println(hello);
}
```

### 4、切换使用 jedis

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

## 二、缓存失效问题

先来解决大并发读情况下的缓存失效问题：

### 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，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：  
缓存空结果、并且设置短的过期时间。

### 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。

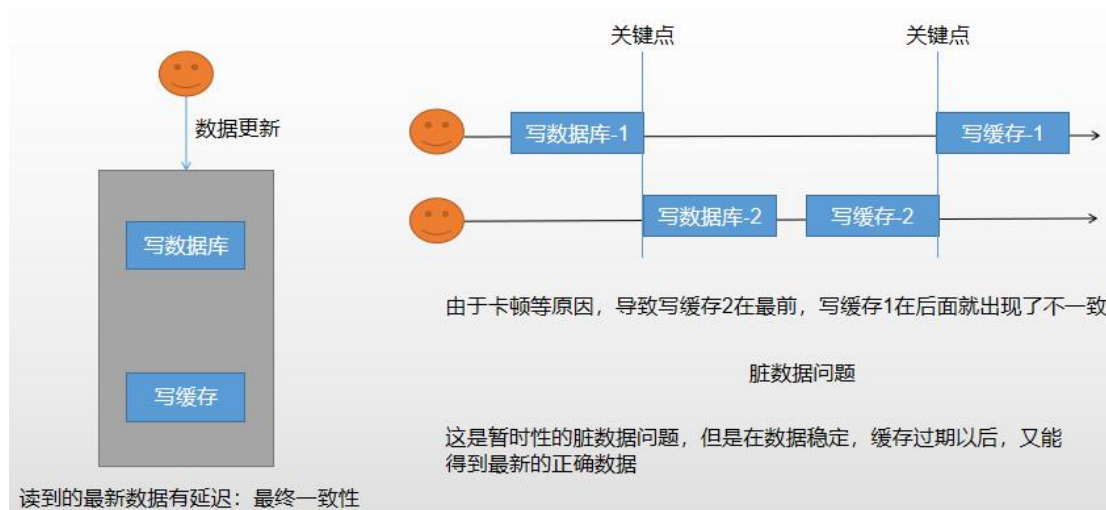
### 3、缓存击穿

- 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。
- 解决：  
加锁

## 三、缓存数据一致性

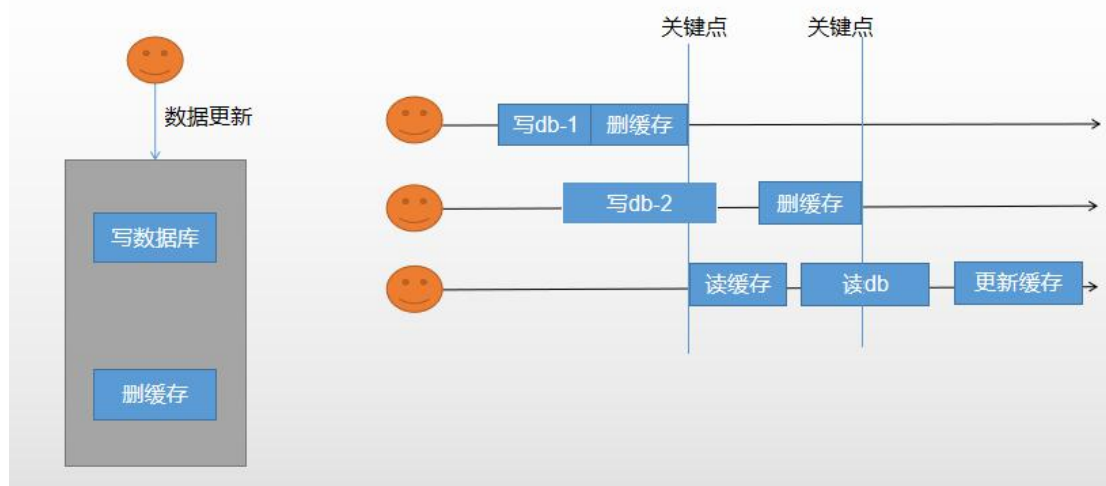
### 1、保证一致性模式

#### 1、双写模式



#### 2、失效模式

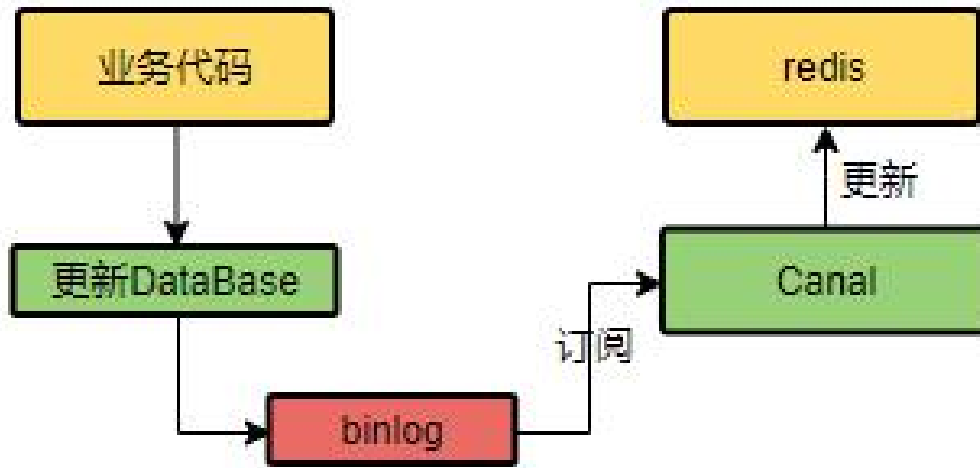
##### 缓存一致性-失效模式



### 3、改进方法 1-分布式读写锁

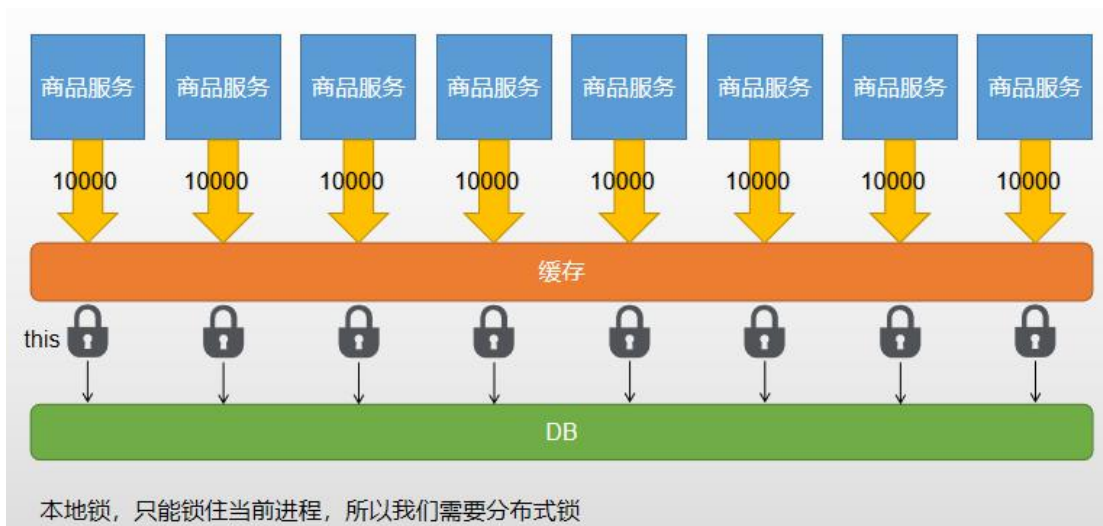
分布式读写锁。读数据等待写数据整个操作完成

#### 4、改进方法 2-使用 canal

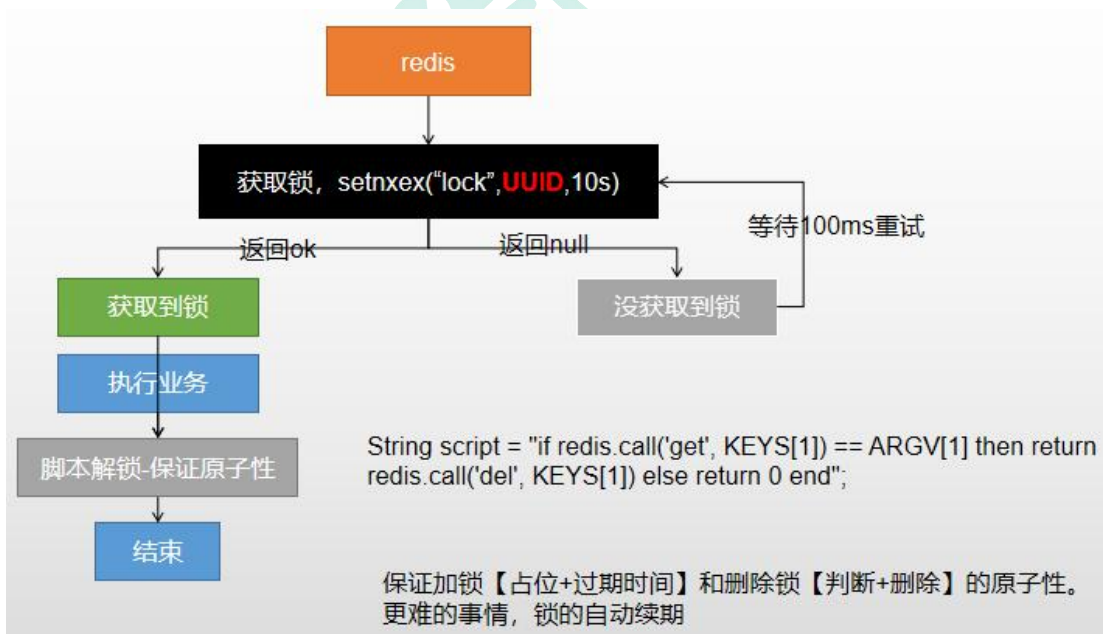


## 四、分布式锁

### 1、分布式锁与本地锁



### 2、分布式锁实现



使用 RedisTemplate 操作分布式锁

```
public Map<String, List<Catalog2Vo>> getCatalogJsonFromDbWithRedisLock() {
```

```
//1、占分布式锁。去redis占坑
```

```
String uuid = UUID.randomUUID().toString();
Boolean lock =
redisTemplate.opsForValue().setIfAbsent("lock",uuid,300,TimeUnit.SECONDS);
if(lock){
    System.out.println("获取分布式锁成功...");
    //加锁成功... 执行业务
    //2、设置过期时间，必须和加锁是同步的，原子的
    //redisTemplate.expire("lock",30,TimeUnit.SECONDS);
    Map<String, List<Catalog2Vo>> dataFromDb;
    try{
        dataFromDb = getDataFromDb();
    }finally {
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then
return redis.call('del', KEYS[1]) else return 0 end";
        //删除锁
        Long lock1 = redisTemplate.execute(new
DefaultRedisScript<Long>(script, Long.class)
        , Arrays.asList("lock"), uuid);
    }

    //获取值对比+对比成功删除=原子操作 Lua 脚本解锁
    // String lockValue = redisTemplate.opsForValue().get("lock");
    // if(uuid.equals(lockValue)){
    //     //删除我自己的锁
    //     redisTemplate.delete("lock");//删除锁
    // }
    return dataFromDb;
}else {
    //加锁失败...重试。synchronized ()
    //休眠 100ms 重试
    System.out.println("获取分布式锁失败...等待重试");
    try{
        Thread.sleep(200);
    }catch (Exception e){

    }

    return getCatalogJsonFromDbWithRedisLock();//自旋的方式
}

}
```



## 3、Redisson 完成分布式锁

### 1、简介

**Redisson** 是架设在 **Redis** 基础上的一个 Java 驻内存数据网格（In-Memory Data Grid）。充分的利用了 Redis 键值数据库提供的一系列优势，**基于 Java 实用工具包中常用接口**，为用户提供了一系列具有分布式特性的常用工具类。**使得原本作为协调单机多线程并发程序的工具包获得了协调分布式多机多线程并发系统的能力**，大大降低了设计和研发大规模分布式系统的难度。同时结合各富特色的分布式服务，更进一步简化了分布式环境中程序相互之间的协作。

官方文档：<https://github.com/redisson/redisson/wiki/%E7%9B%AE%E5%BD%95>

### 2、配置

```
// 默认连接地址 127.0.0.1:6379
RedissonClient redisson = Redisson.create();

Config config = new Config();
//redis://127.0.0.1:7181
//可以用"rediss://"来启用 SSL 连接
config.useSingleServer().setAddress("redis://192.168.56.10:6379");
RedissonClient redisson = Redisson.create(config);
```

### 3、使用分布式锁

```
RLock lock = redisson.getLock("anyLock");// 最常见的使用方法
lock.lock();
// 加锁以后 10 秒钟自动解锁// 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 boolean res = lock.tryLock(100,
10, TimeUnit.SECONDS);
if (res) {
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

## 4、使用其他

参照官方文档进行测试

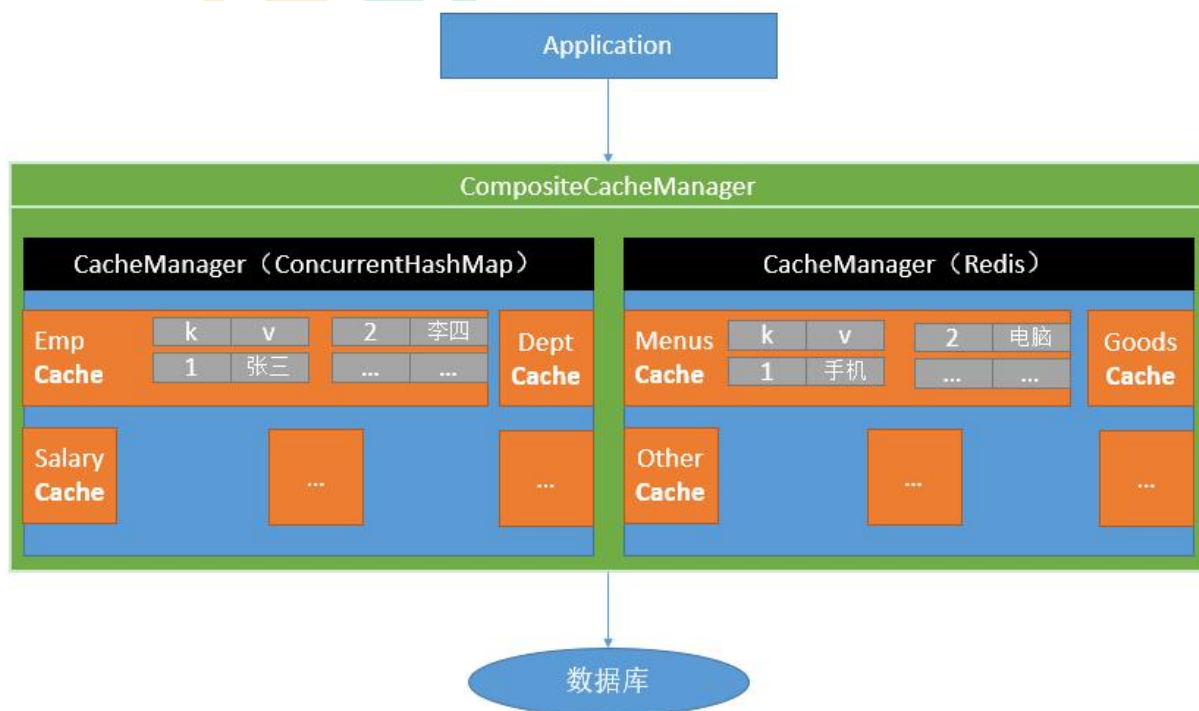


## 五、Spring Cache

### 1、简介

- Spring 从 3.1 开始定义了 `org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口来统一不同的缓存技术；并支持使用 JCache（JSR-107）注解简化我们开发；
- Cache 接口为缓存的组件规范定义，包含缓存的各种操作集合；Cache 接口下 Spring 提供了各种 xxxCache 的实现；如 RedisCache，EhCacheCache，ConcurrentMapCache 等；
- 每次调用需要缓存功能的方法时，Spring 会检查指定参数的指定的目标方法是否已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。
- 使用 Spring 缓存抽象时我们需要关注以下两点；
  - 1、确定方法需要被缓存以及他们的缓存策略
  - 2、从缓存中读取之前缓存存储的数据

### 2、基础概念



### 3、注解

Cache	缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存（Cache）组件
@Cacheable	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略

@Cacheable/@CachePut/@CacheEvict 主要的参数

value	缓存的名称，在 spring 配置文件中定义，必须指定至少一个	例如： @Cacheable(value="mycache") 或者 @Cacheable(value={"cache1","cache2"})
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @Cacheable(value="testcache",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存，在调用方法之前之后都能判断	例如： @Cacheable(value="testcache",condition="#userName.length()>2")
allEntries (@CacheEvict)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： @CacheEvict(value="testcache",allEntries=true)
beforeInvocation (@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： @CacheEvict(value="testcache", beforeInvocation=true)
unless (@CachePut) (@Cacheable)	用于否决缓存的，不像 condition，该表达式只在方法执行之后判断，此时可以拿到返回值 result 进行判断。条件为 true 不会缓存，false 才缓存	例如： @Cacheable(value="testcache",unless="#result == null")

## 4、表达式语法

### *Cache SpEL available metadata*

名字	位置	描述	示例
methodName	root object	当前被调用的方法名	#root.methodName
method	root object	当前被调用的方法	#root.method.name
target	root object	当前被调用的目标对象	#root.target
targetClass	root object	当前被调用的目标对象类	#root.targetClass
args	root object	当前被调用的方法的参数列表	#root.args[0]
caches	root object	当前方法调用使用的缓存列表（如@Cacheable(value={"cache1", "cache2"})），则有两个cache	#root.caches[0].name
argument name	evaluation context	方法参数的名字。可以直接 #参数名，也可以使用 #p0或#a0 的形式，0代表参数的索引；	#iban、#a0、#p0
result	evaluation context	方法执行后的返回值（仅当方法执行之后的判断有效，如 'unless'，'cache put'的表达式 'cache evict'的表达式 beforeInvocation=false）	#result

## 5、缓存穿透问题解决

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-cache</artifactId>

</dependency>

允许 null 值缓存