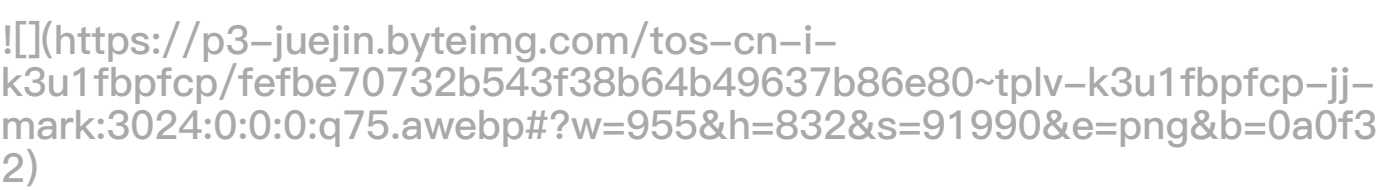
Redis缓存雪崩、击穿、穿透、预热

> Redis缓存雪崩、缓存击穿、缓存穿透、缓存预热.

背景

在实际工程中，Redis缓存问题常伴随高并发场景出现。例如，`电商大促、活动报名、突发新闻`时，由于缓存失效导致大量请求访问数据库，导致`雪崩`、`击穿`、`穿透`等问题。因此，新系统上线前需`预热`缓存，以应对高并发，减轻数据库压力。本章主要围绕这几个核心问题，针对产生场景、分析原因、并给出相应的解决方案。

文章导读



缓存雪崩

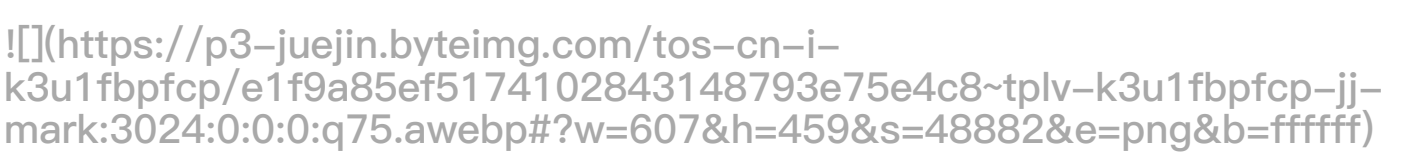
什么是缓存雪崩?

指在一个高并发的系统中，由于大量的缓存数据在同一时间段内过期或失效，导致大量请求无法从缓存中获取数据，因此大量并发请求可能导致数据库服务器过载，甚至宕机，从而引发整个系统崩溃。

产生原因

1、硬件故障。 比如：机房着火、停电等导致`redis挂机`。

2、软件设计。redis中有`大量key同时过期或者大面积失效`。导致缓存中查不到对应key。此时访问压力给到数据库。如图示：



实际应用场景中，如：

* 金融系统股票价格、汇率等数据实时变动；
* 热点新闻缓存过期时，大量用户访问新闻详情页的请求；
* 电商平台在大促期间，如“618”、“双十一”等。

如果这些数据在缓存中的过期时间设置得过于集中，或者缓存服务器突然宕机，那么在数据过期或缓存失效的瞬间，大量请求会直接访问数据库，导致数据库负载剧增，可能引发`雪崩`。

程序示例

这里给出伪代码：

```
// 缓存类
class SimpleCache {
    private Map<String, Object> cache;
    private long expirationTime;

    public SimpleCache(long expirationTime) {
        this.cache = new HashMap<>();
        this.expirationTime = expirationTime;
    }

    public Object get(String key) {
        // 假设所有缓存项都已过期
        return null; // 返回null表示缓存失效
    }

    // 省略添加数据到缓存
}
```

```
// 模拟数据源访问
class DataSource {
    public Object fetchData(String key) {
        // 模拟数据源访问的延迟
        try {
            Thread.sleep(1000); // 假设每次访问需要1秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Data for " + key;
    }
}

public class CacheAvalancheSimulation {
    public static void main(String[] args) {
        // 设置缓存过期时间为一个非常短的时间，以模拟缓存雪崩
        SimpleCache cache = new SimpleCache(1); // 假设过期时间为1毫秒
        DataSource dataSource = new DataSource();

        // 模拟大量请求
        for (int i = 0; i < 1000000; i++) {
            String key = "data" + i;
            Object data = cache.get(key);
            if (data == null) {
                // 缓存失效，从数据库获取数据
                data = dataSource.fetchData(key);
            }
            // 获取到的数据或处理数据
        }
    }
}

```

流程说明：

1. **初始化缓存**：

* 假设缓存中存储了一些数据。
* 设置缓存中所有项的过期时间为相同的、非常短的时间点。
2. **模拟大量请求**：

* 当缓存项过期后，程序开始接收大量请求。
* 每个请求都尝试从缓存中获取数据。

3. **缓存失效与数据库访问**：

* 由于所有缓存项都已过期，所有请求都无法从缓存中获取数据。
* 因此，这些请求都会转而访问数据库来获取数据。
4. **数据源压力增大**：

* 短时间内的大量请求导致数据源承受巨大的访问压力。
* 如果数据源无法及时响应所有请求，就会出现性能下降或崩溃的情况。

解决方案

针对雪崩问题的解决方案，一般可以从以下几个角度考虑

> 方案一、设置redis的某些key永不过期

简单地使用 `SET` 命令来添加这个key，不需要调用任何设置过期时间的命令：

```
SET product:500100 ”{”name”:”ProductName”, ”price”:99.99, ”status”:”上架中”}”
```

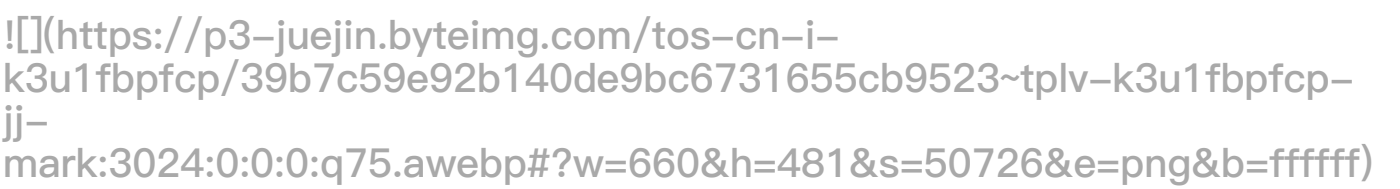
在java中，可以使用Jedis或者RedisTemplate客户端

```
redisTemplate.opsForValue().set(productCode, productData);
```

值得注意的是，可能因为Redis的内存限制或其他原因（如使用了Redis的LRU淘汰策略）而被删除。因此，在设计系统时，除了考虑key的过期时间外，还需要考虑Redis的内存管理和淘汰策略。

> 方案二、搭建高可用redis集群+持久化

如图，一般线上环境都会有redis集群保证高可用。



对应集群模式：

* 主从模式（Master-Slave）
* 哨兵模式 （Sentinel）
* 分片模式 （Cluster）

关于三种集群模式的优缺点对比，配置这里不再赘述。图示只说明redis集群保障高可用的能力。

> 方案三、服务降级、熔断、限流

在Spring Cloud中使用Hystrix和Sentinel进行熔断降级和限流，这里给出简单的程序示例。

* **服务熔断降级 - Hystrix**

配置pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

在启动类上添加`@EnableHystrix`注解来启用Hystrix。

```
@SpringBootApplication
@EnableHystrix
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

```

在服务调用方法上添加`@HystrixCommand`注解，并配置相应的降级逻辑。

```
@Service
public class MyService {

    @HystrixCommand(fallbackMethod = "fallbackMyMethod")
    public String busiMethod() {
        // 调用远程服务
    }

    public String fallbackMyMethod() {
        // 降级逻辑
        return "Fallback response";
    }
}

```

对于Hystrix的YML配置，你可以在`application.yml`或`application.properties`中设置一些全局参数，例如超时时间、请求缓存等。

```
hystrix:
  command:
    default:
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 5000 # 设置命令执行的超时时间
      fallback:
```

```
        enabled: true # 启用降级逻辑
```

* **限流 – Sentinel**

在`pom.xml`中添加了Sentinel的依赖。

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

在启动类上添加`@EnableDiscoveryClient`和`@EnableSentinel`注解。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableSentinel
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Sentinel的限流规则通常不是在YML文件中配置的，而是通过Sentinel Dashboard进行动态配置。这里设置一些基础配置，例如数据源等。

在`application.yml`中，你可以配置Sentinel的数据源类型以及Dashboard的地址。

```
spring:
  cloud:
    sentinel:
```

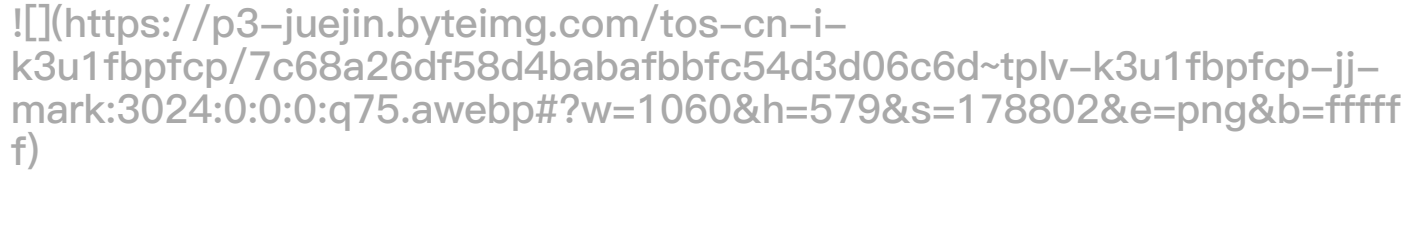
```
      transport:
        dashboard: 127.0.0.1:8080 # Sentinel Dashboard的地址
      datasource:
        flow:
          default:
            type: file
            file:
              file-name: classpath:flow-rules.json # 规则文件路径
            rule-type: flow
```

需要创建一个规则文件，例如`flow-rules.json`，并在其中定义限流规则。

```
[    {        "resource": "some-resource",        "count": 10,        "grade": 1,        "limitApp": "default",        "strategy": 0    }]
```

这里我们针对`some-resource`的QPS限流规则，允许每秒最多10个请求。Sentinel的主要优势在于其动态规则管理能力，因此通常建议通过Sentinel Dashboard进行实时配置和监控。

当然，对于限流，也可以在网关层面使用Spring Cloud Gateway实现。如下图：



Spring Cloud Gateway是分布式系统中流量的入口。它需要对大量的请求进行管理和控制，以防止服务调用失败（如超时、异常）导致的请求堆积在网关上。通过在网关层面进行限流，可以快速失败并返回给客户端，从而保护后端服务的稳定性。

> 方案四、设计多级缓存架构

多级缓存主要从：`redis缓存`、`本地缓存`、`Nginx缓存`考虑。可以参考前边的文章 《[多级缓存设计和实战应用](http://cxyroad.com/”https://mp.weixin.qq.com/s/nJzDpQcgnKaGMet3NvVWAQ”)》，这里详细介绍了多级缓存方案。

缓存穿透

什么是缓存穿透?

一句话，就是查询一条根本没有的记录。它`既不存在于redis中，也不存在数据库中`。一般我们的查询顺序是先查redis、再查数据库。每次请求最终都会访问到数据库。造成数据库访问压力。这种现象叫`缓存穿透`。

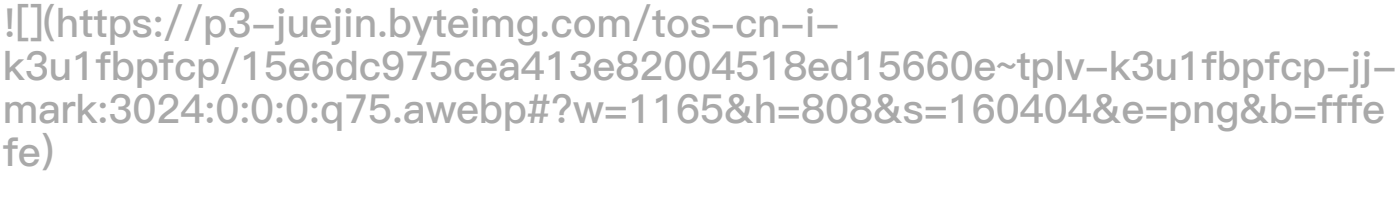
产生原因

* 查询记录在redis和数据库均不存在
* 恶意攻击

解决方案

> 方案一、使用布隆过滤器

使用布隆过滤器解决缓存穿透问题具体流程如下：



流程说明：

* Web端发起请求到App服务应用
* 经过布隆过滤器判断key是否存在
* 若不存在，直接返回null
* 若key存在或者被误判(下文详述)，查询redis

* 若redis中有结果，直接返回
* 未查到，则查询数据库返回结果
* 数据库查到结果，返回结果。并将对应key回写redis
* 未查到，则返回null

基于上述流程，给出程序伪代码：

```
public class RequestHandler {
    private BloomFilter<String> bloomFilter;
    private Jedis jedis;
    private DatabaseService databaseService; // 假设这是处理数据库查询的服务

    // 初始化布隆过滤器，这里仅作示意，实际使用需配置正确的funnel和预估元素数量
    public RequestHandler(Funnel<String> funnel, long expectedInsertions) {
        bloomFilter = BloomFilter.create(funnel, expectedInsertions, 0.01); // 假设误报率为0.01
        jedis = new Jedis("localhost"); // 假设Redis服务在本地
        databaseService = new DatabaseService(); // 初始化数据库服务
    }

    public Object handleRequest(String key) {
        // 检查布隆过滤器
        if (!bloomFilter.mightContain(key)) {
            // key不存在，直接返回null
            return null;
        }

        // 检查Redis
        String redisResult = jedis.get(key);
        if (redisResult != null) {
            // Redis中有结果，直接返回
            return redisResult;
        }

        // 查询数据库
        Optional<Object> databaseResult = databaseService.queryDatabase(key);
        if (databaseResult.isPresent()) {
            // 数据库查到结果，返回结果，并将对应key回写Redis
            Object result = databaseResult.get();
            jedis.set(key, result.toString()); // 假设结果可以转换为字符串并存
```

储在Redis中
            return result;
        } else {
            // 数据库未查到，返回null
            return null;
        }
    }

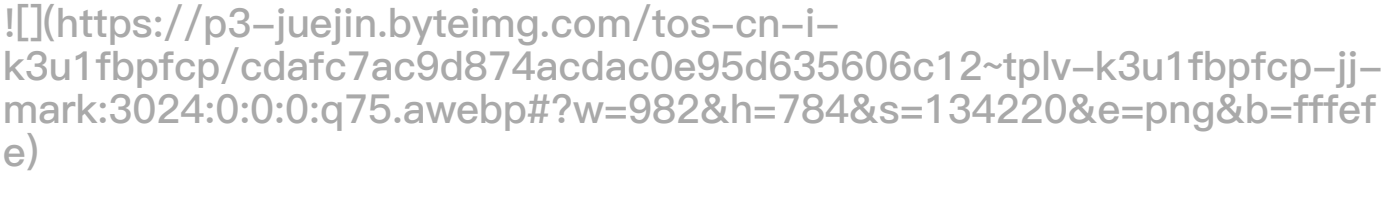
    // 假设的数据库服务类，用于查询数据库
    private static class DatabaseService {
        public Optional<Object> queryDatabase(String key) {
            // 这里应实现具体的数据库查询逻辑
            // 返回Optional封装的结果，若找到结果则返回
Optional.of(result)，否则返回Optional.empty()
            return Optional.empty(); // 示意性代码，实际应替换为数据库查询
逻辑
        }
    }
}

```

关于布隆过滤器使用和原理，可参考先前的文章《[布隆过滤器原理
](http://cxyroad.com/
”https://mp.weixin.qq.com/s/bpkxNH548NnrsdfYhKKgcg”)》。

> 方案一、缓存空对象或者默认值

针对要查询的数据，需求层面沟通，可以在Redis里存一个缺省值(比如，零、负数、defaultNull等)。




流程描述：

* 先去redis查键user:xxxxxx没有，再去mysql查没有获得 ，这就发生了一次穿透现象；
* 第一次来查询user:xxxxxx，redis和mysql都没有，返回null给调用者；
* 将 user:xxxxxx,defaultNull 回写redis；
```

* 第二次查user:xxxxxx，此时redis就有值了。

可以直接从Redis中读取default缺省值返回给业务应用程序，避免了把大量请求发送给mysql处理，打爆mysql。

但是，此方法只能解决key相同的情况，但是无法应对黑客恶意攻击。

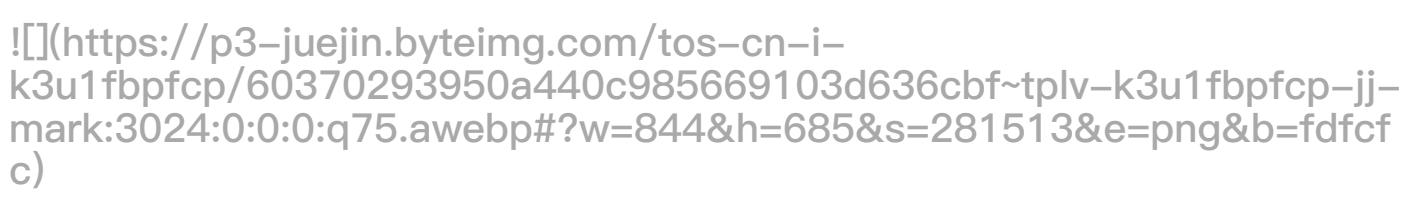
缓存击穿

什么是缓存击穿?

一句话，就是`热点key突然失效`了，大量请求暴击数据库。关键词：大量请求、同一个热点key、正好失效。

发生场景

* 某个key以到过期时间，但是还是被访问到
* key被删除，但是被访问到
* 如，某电商网站的今日特卖



解决方案

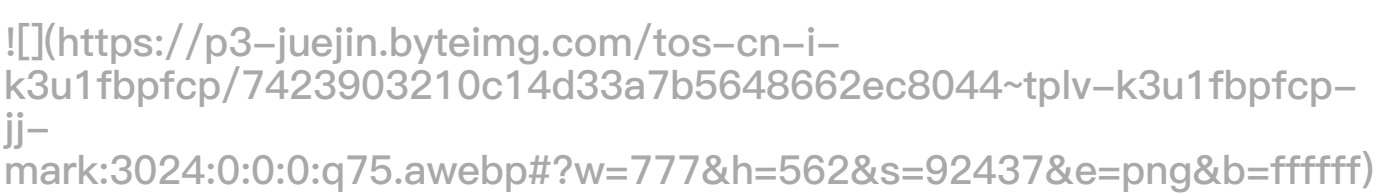
> 方案一、差异失效时间

差异失效时间，对于访问频繁的热点key，不设置过期时间。和上边类似，这里不再赘述。

> 方案二、采用互斥锁，双检加锁策略更新

双检锁通常用于解决懒加载中的并发问题，即只有当数据在缓存中不存在时，才进行数据库查询，并且确保`只有一个线程`进行数据库查询。

基本流程如下：



1. 当大量请求进来，多个线程同时去查询数据库获取同一条数据；
2. 在第一个查询数据的请求上使用一个`互斥锁`来锁定；
3. 其他的线程走到这一步拿不到锁排队等着；
4. 第一个线程查询到了数据，然后做缓存；
5. 后面的线程进来发现已经有缓存了，就直接走缓存。

对应程序伪代码示例：

```
public String get(String key) {
    String value = redis.get(key); // 查询缓存
    if (value != null) {
        // 缓存存在，直接返回
        return value;
    } else {
        // 缓存不存在，对方法加锁（同步块）
        synchronized (TaskFuture.class) {
            // 再次查询Redis,防止在同步块外等待的线程已经加载了数据
            value = redis.get(key);
            if (value != null) {
                // 再次查到数据，直接返回
                return value;
            } else {
                // 二次查询Redis也不存在，查询数据库
                value = dao.get(key);
                // 将数据存入Redis缓存
                redis.setnx(key, value, time);
                // 返回从数据库查询到的值
                return value;
            }
```

```
        }
    }
}
```

注意: 在真实的生产环境中，考虑到缓存一致性和数据一致性的问题，有些情况下可能还需要使用更复杂的并发控制机制。

缓存预热

缓存预热是一种`缓存优化`技术，其核心思想是在系统上线或服务重启之前，`提前`将相关的缓存数据加载到缓存系统中。这样做的目的是为了避免在实际请求到达时进行缓存项的加载，从而减少了响应时间，提升了系统的性能。

以下是一些常见的缓存预热实现方案：

> 方案一、批量预加载数据

在系统启动或服务重启时，可以批量查询数据库并将结果存入缓存中。这种方式适用于`数据量相对固定且比较小`的情况。比如：常用的字典配置信息。

```
public void preloadCache() {
    List<String> keys = ... // 获取需要预热的缓存键列表
    for (String key : keys) {
        Object value = dao.get(key); // 从数据库中获取数据
        if (value != null) {
            redis.set(key, serialize(value), cacheExpiration); // 将数据存入缓存
        }
    }
}
```

> 方案二、异步预热

如果预热的`数据量很大`，或者预热过程比较耗时，可以考虑使用`异步任务`来执行预热操作。这样可以避免阻塞系统启动，并且可以利用系统的空闲时间来预热缓存。

```
@PostConstruct
public void startPreloadCacheAsync() {
    CompletableFuture.runAsync(() -> preloadCache());
}
```

> 方案三、定时任务预热

对于某些`周期性`变化的数据，可以用`定时任务`来定期预热缓存。例如，对于每天更新的数据，可以在每天的某个固定时间执行预热操作。

```
@Scheduled(fixedRate = 24 * 60 * 60 * 1000) // 每天执行一次
public void schedulePreloadCache() {
    preloadCache();
}
```

> 方案四、使用SpringBoot `InitializingBean`接口实现

下面是一个使用`InitializingBean`接口实现缓存预热的示例：

```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;

@Component
public class CachePreloader implements InitializingBean {

    @Override
```

```
    public void afterPropertiesSet() throws Exception {
        // 获取需要预热的缓存键列表
        List<String> keysToPreload = ...;
        for (String key : keysToPreload) {
            // 从数据源获取数据
            Object value = dbService.get(key);
            if (value != null) {
                // 将数据放入缓存
                cacheService.put(key, value);
            }
        }
    }
}
```

总结
--

本文主要介绍了redis缓存使用中常见的问题：缓存雪崩、缓存击穿、缓存穿透、缓存预热等。详情归纳如下：

缓存问题	产生原因	解决方案
缓存雪崩	大量缓存失效，导致数据库过载	1. 分散缓存失效时间 2. 多级缓存 3. 缓存高可用 4. 服务降级限流
缓存穿透	查询不存在的数据，导致数据库过载	1. 布隆过滤器 2. 空值缓存
缓存击穿	热点数据失效，导致数据库过载	1. 热点数据永不过期 2. 使用互斥锁

往期推荐

[Redis多级缓存设计实战应用](http://cxyroad.com/”https://mp.weixin.qq.com/s/nJzDpQcgnKaGMet3NvVWAQ”)

[Redis分布式锁使用及常见问题](http://cxyroad.com/”https://mp.weixin.qq.com/s/p8mQ7hg-QKyWoG74ogrzvA”)

[Redis调优BigKey如何处理？](http://cxyroad.com/

”https://mp.weixin.qq.com/s/BZaTk6ZpsL127ERgVXZKPw”)

[Redis布隆过滤器原理](http://cxyroad.com/
”https://mp.weixin.qq.com/s/bpkxNH548NnrsdfYhKKgcg”)

[Redis常见经典面试题](http://cxyroad.com/
”https://mp.weixin.qq.com/s/sSg2X1oCrYSYkYhSaGZkdQ”)

结尾
--

共享即共赢。如有帮助，欢迎`转发`、`点赞`和`在看`。公众号【码易有道】，一起做长期且正确的事情！！！

原文链接: https://juejin.cn/post/7352079398072074303