# Redis应用场景

## 缓存

提到redis，我们第一想到的应用场景肯定是缓存，因为redis是基于内存的<K,V>数据库，具有很强大的读写性能。当我们出现读多写少的热点数据时，缓存会起到非常非常重要的作用。如何保证缓存的一致性是很重要的。

热点数据（经常会被查询，但是不经常被修改或者删除的数据），首选是使用redis缓存，毕竟强大的QPS和极强的稳定性不是所有类似工具都有的，而且相比于memcached还提供了丰富的数据类型可以使用，另外，内存中的数据也提供了AOF和RDB等持久化机制可以选择，要冷、热的还是忽冷忽热的都可选。

结合具体应用需要注意一下：很多人用spring的AOP来构建redis缓存的自动生产和清除，过程可能如下：

* Select 数据库前查询redis，有的话使用redis数据，放弃select 数据库，没有的话，select 数据库，然后将数据插入redis
* update或者delete数据库钱，查询redis是否存在该数据，存在的话先删除redis中数据，然后再update或者delete数据库中的数据

上面这种操作，如果并发量很小的情况下基本没问题，但是高并发的情况请注意下面场景：

为了update先删掉了redis中的该数据，这时候另一个线程执行查询，发现redis中没有，瞬间执行了查询SQL，并且插入到redis中一条数据，回到刚才那个update语句，这个悲催的线程压根不知道刚才那个该死的select线程犯了一个弥天大错！于是这个redis中的错误数据就永远的存在了下去，直到下一个update或者delete。

## 队列--分布式消息队列

任务队列：顾名思义，就是“传递消息的队列”。与任务队列进行交互的实体有两类，一类是**生产者（producer）**，另一类则是**消费者（consumer）**。生产者将需要处理的任务放入任务队列中，而消费者则不断地从任务独立中读入任务信息并执行。

**任务队列的好处：**

1. **松耦合**。生产者和消费者只需按照约定的任务描述格式，进行编写代码。
2. **易于扩展**。多消费者模式下，消费者可以分布在多个不同的服务器中，由此降低单台服务器的负载。

由于Redis有提供了队列（列表）的数据类型，我们可以把他用作分布式队列。实现任务队列，只需让生产者将任务使用LPUSH加入到某个键中，然后另一个消费者不断地使用RPOP命令从该键中取出任务即可。

**将Redis用作日志收集器：**实际上还是一个队列，多个端点将日志信息写入Redis，然后一个worker统一将所有日志写到磁盘。同时，由于Redis有订阅发布机制，所以它可以实现一些消息队列比如：Rocket MQ，Active MQ的一些功能。

## 单线程机制的应用

由于Redis是单线程机制，所以不会出现并发问题，所以可以将其作为：计数器，产生唯一ID，做秒杀系统。

**计数器：**诸如统计点击数等应用。由于单线程，可以避免并发问题，保证不会出错，而且100%毫秒级性能！爽。

**产生唯一ID**：在分布式系统中，如何产生唯一的商品id，订单id等等是很重要的，所以我们可以通过Redis的单线程计数器来产生唯一的ID，并且能保证性能。

**做秒杀系统：**在秒杀系统中，如何对抗高并发与解决超卖问题是两个关键问题。我们可以先将商品放入Redis中，在Redis中单线程的用DECR命令判断是否超卖了，从而将很少的流量放入到数据库端，达到限流与超卖问题。

## 分布式锁

分布式锁在很多应用场景下是非常有效的手段，比如当运行在多个机器上的不同进程需要访问同一个竞争资源的时候，那么就会涉及到进程对资源的加锁和释放，这样才能保证数据的安全访问。分布式锁实现的方案有很多，比如基于ZooKeeper实现、或者基于Mysql实现等等，今天我们来一起看看如何基于Redis实现分布式锁服务。

对于分布式锁的目标，我们必须首先明确三点：

1. 任何一个时间点必须只能够有一个客户端拥有锁。
2. 不能够有死锁，也就是最终客户端都能够获得锁，尽管可能会经历失败。
3. 错误容忍性要好，只要有大部分的Redis实例存活，客户端就应该能够获得锁。

理解了上面我们列出的三个点，我们来分析一下一般的基于Redis实现的分布式锁：

使用Redis实现锁最简单的办法是**创建一个key，利用key的唯一性（如果有一个客户端创建成功了这个key，那么其他客户端就无法获得）**，且这个key通常有有限的存活时间，这一点可以利用Redis的过期时间特性，所以锁最终会被释放掉，当客户端需要释放资源的时候，客户端delete这个key即可。

So far so good!但是有个单点问题，假如Redis master挂掉怎么办，因此我们需要加个slave，当master挂掉的时候可以切换到slave。这又带来了新的问题，由于Redis的复制是异步的，**因此我们不能保证同时只有一个客户端获得锁（这就是为什么zk集群可以稳定的，因为zk自身保证一致性）**。

这个模型有很显然的竞态：

1. Client在Master上面获得了锁。
2. master在数据同步到slave之前挂掉了。
3. slave升级成为master。
4. Client B申请了同样的资源的锁，成功了！

在特定条件下这种情况是会发生的，当出现多个客户端同时获得锁的时候，我们就认为可以这种锁方案是不可靠的。