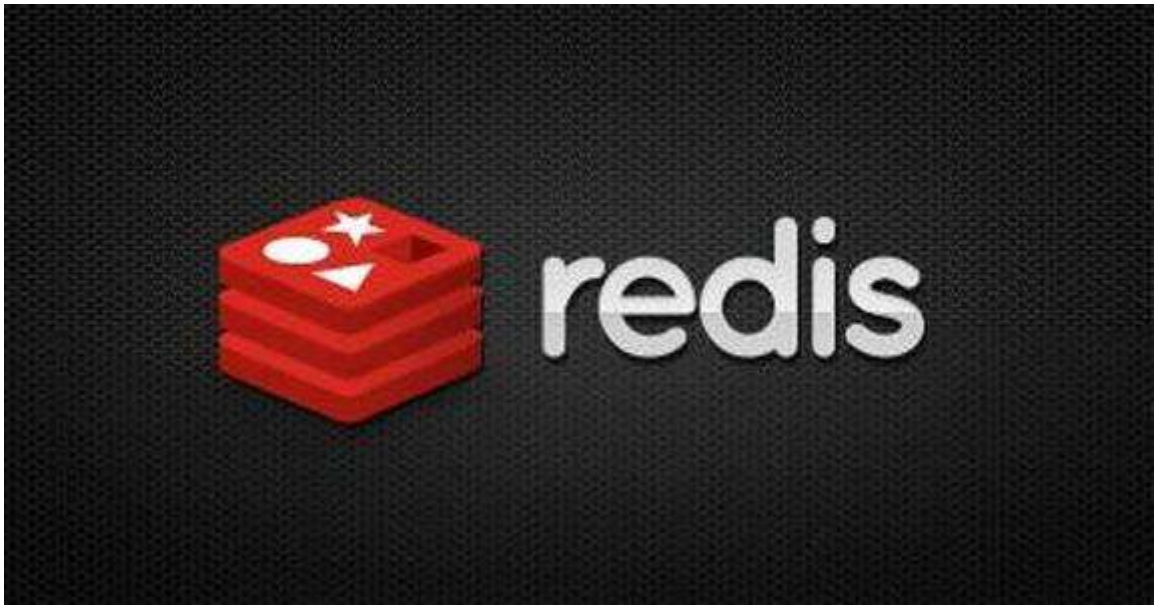


Redis数据库

灵魂拷问：不是学了MySQL吗，存数据也能存了啊，又学一个数据库干嘛？

在前面我们学习了MySQL数据库，它是一种传统的关系型数据库，我们可以使用MySQL来更好地管理和组织我们的数据，虽然在小型Web应用下，只需要一个MySQL+Mybatis自带的缓存系统就可以胜任大部分的数据存储工作。但是MySQL的缺点也很明显，它的数据始终是存储在硬盘上的，对于我们的用户信息这种不需要经常发生修改的内容，使用MySQL存储确实可以，但是如果是快速更新或是频繁使用的数据，比如微博热搜、双十一秒杀，这些数据不仅要求服务器需要提供更高的响应速度，而且还需要面对短时间内上百万甚至上千万次访问，而MySQL的磁盘IO读写性能完全不能满足上面的需求，能够满足上述需求的只有内存，因为速度远高于磁盘IO。

因此，我们需要寻找一种更好的解决方案，来存储上述这类特殊数据，弥补MySQL的不足，以应对大数据时代的重重考验。

NoSQL概论

NoSQL全称是Not Only SQL（不仅仅是SQL）它是一种非关系型数据库，相比传统SQL关系型数据库，它：

- 不保证关系数据的ACID特性
- 并不遵循SQL标准
- 消除数据之间关联性

乍一看，这玩意不比MySQL垃圾？我们再来看看它的优势：

- 远超传统关系型数据库的性能
- 非常易于扩展
- 数据模型更加灵活
- 高可用

这样，NoSQL的优势一下就出来了，这不就是我们正要寻找的高并发海量数据的解决方案吗！

NoSQL数据库分为以下几种：

- **键值存储数据库：**所有的数据都是以键值方式存储的，类似于我们之前学过的HashMap，使用起来非常简单方便，性能也非常高。

- **列存储数据库**：这部分数据库通常是用来应对分布式存储的海量数据。键仍然存在，但是它们的特点是指向了多个列。
- **文档型数据库**：它是以一种特定的文档格式存储数据，比如JSON格式，在处理网页等复杂数据时，文档型数据库比传统键值数据库的查询效率更高。
- **图形数据库**：利用类似于图的数据结构存储数据，结合图相关算法实现高速访问。

其中我们要学习的Redis数据库，就是一个开源的**键值存储数据库**，所有的数据全部存放在内存中，它的性能大大高于磁盘IO，并且它也可以支持数据持久化，他还支持横向扩展、主从复制等。

实际生产中，我们一般会配合使用Redis和MySQL以发挥它们各自的优势，取长补短。

Redis安装和部署

我们这里还是使用Windows安装Redis服务器，但是官方指定是安装到Linux服务器上，我们后面学习了Linux之后，再来安装到Linux服务器上。由于官方并没有提供Windows版本的安装包，我们需要另外寻找：

- 官网地址：https://redis.io
- GitHub Windows版本维护地址：https://github.com/tporadowski/redis/releases

---

基本操作

在我们之前使用MySQL时，我们需要先在数据库中创建一张表，并定义好表的每个字段内容，最后再通过`insert`语句向表中添加数据，而Redis并不具有MySQL那样的严格的表结构，Redis是一个键值数据库，因此，可以像Map一样的操作方式，通过键值对向Redis数据库中添加数据（操作起来类似于向一个HashMap中存放数据）

在Redis下，数据库是由一个整数索引标识，而不是由一个数据库名称。 默认情况下，我们连接Redis数据库之后，会使用0号数据库，我们可以通过Redis配置文件中的参数来修改数据库总数，默认为16个。

我们可以通过`select`语句进行切换：

```
select 序号;
```

数据操作

我们来看看，如何向Redis数据库中添加数据：

```
set <key> <value>
-- 一次性多个
mset [<key> <value>]...
```

所有存入的数据默认会以**字符串**的形式保存，键值具有一定的命名规范，以方便我们可以快速定位我们的数据属于哪一个部分，比如用户的数据：

```
-- 使用冒号来进行板块分割，比如下面表示用户xxx的信息中的name属性，值为lbw
set user:info:用户ID:name lbw
```

我们可以通过键值获取存入的值：

```
get <key>
```

你以为Redis就仅仅只是存取个数据吗？它还支持数据的过期时间设定：

```
set <key> <value> EX 秒
set <key> <value> PX 毫秒
```

当数据到达指定时间时，会被自动删除。我们也可以单独为其他的键值对设置过期时间：

```
expire <key> 秒
```

通过下面的命令来查询某个键值对的过期时间还剩多少：

```
ttl <key>
-- 毫秒显示
pttl <key>
-- 转换为永久
persist <key>
```

那么当我们想直接删除这个数据时呢？直接使用：

```
del <key>...
```

删除命令可以同时拼接多个键值一起删除。

当我们想要查看数据库中所有的键值时：

```
keys *
```

也可以查询某个键是否存在：

```
exists <key>...
```

还可以随机拿一个键：

```
randomkey
```

我们可以将一个数据库中的内容移动到另一个数据库中：

```
move <key> 数据库序号
```

修改一个键为另一个键：

```
rename <key> <新的名称>
-- 下面这个会检查新的名称是否已经存在
renamex <key> <新的名称>
```

如果存放的数据是一个数字，我们还可以对其进行自增自减操作：

```
-- 等价于a = a + 1
incr <key>
-- 等价于a = a + b
incrby <key> b
-- 等价于a = a - 1
decr <key>
```

最后就是查看值的数据类型：

```
type <key>
```

Redis数据库也支持多种数据类型，但是它更偏向于我们在Java中认识的那些数据类型。

数据类型介绍

一个键值对除了存储一个String类型的值以外，还支持多种常用的数据类型。

Hash

这种类型本质上就是一个HashMap，也就是嵌套了一个HashMap罢了，在Java中就像这样：

```
#Redis默认存String类似于这样：
Map<String, String> hash = new HashMap<>();
#Redis存Hash类型的数据类似于这样：
Map<String, Map<String, String>> hash = new HashMap<>();
```

它比较适合存储类这样的数据，由于值本身又是一个Map，因此我们可以在此Map中放入类的各种属性和值，以实现一个Hash数据类型存储一个类的数据。

我们可以像这样来添加一个Hash类型的数据：

```
hset <key> [<字段> <值>]...
```

我们可以直接获取：

```
hget <key> <字段>
-- 如果想要一次性获取所有的字段和值
hgetall <key>
```

同样的，我们也可以判断某个字段是否存在：

```
hexists <key> <字段>
```

删除Hash中的某个字段：

```
hdel <key>
```

我们发现，在操作一个Hash时，实际上就是我们普通操作命令前面添加一个`h`，这样就能以同样的方式去操作Hash里面存放的键值对了，这里就不一一列出所有的操作了。我们来看看几个比较特殊的。

我们现在想要知道Hash中一共存了多少个键值对：

```
hlen <key>
```

我们也可以一次性获取所有字段的值：

```
hvals <key>
```

唯一需要注意的是，Hash中只能存放字符串值，不允许出现嵌套的的情况。

List

我们接着来看List类型，实际上这个猜都知道，它就是一个列表，而列表中存放一系列的字符串，它支持随机访问，支持双端操作，就像我们使用Java中的LinkedList一样。

我们可以直接向一个已存在或是不存在的List中添加数据，如果不存在，会自动创建：

```
-- 向列表头部添加元素
lpush <key> <element>...
-- 向列表尾部添加元素
rpush <key> <element>...
-- 在指定元素前面/后面插入元素
linsert <key> before/after <指定元素> <element>
```

同样的，获取元素也非常简单：

```
-- 根据下标获取元素
lindex <key> <下标>
-- 获取并移除头部元素
lpop <key>
-- 获取并移除尾部元素
rpop <key>
-- 获取指定范围内的
lrange <key> start stop
```

注意下标可以使用负数来表示从后到前数的数字（Python：搁这儿抄呢是吧）：

```
-- 获取列表a中的全部元素
lrange a 0 -1
```

没想到吧，push和pop还能连着用呢：

```
-- 从前一个数组的最后取一个数出来放到另一个数组的头部，并返回元素
rpoplpush 当前数组 目标数组
```

它还支持阻塞操作，类似于生产者和消费者，比如我们想要等待列表中有了数据后再进行pop操作：

```
-- 如果列表中没有元素，那么就等待，如果指定时间（秒）内被添加了数据，那么就执行pop操作，如果超时就作废，支持同时等待多个列表，只要其中一个列表有元素了，那么就能执行
blpop <key>... timeout
```

Set和SortedSet

Set集合其实就像Java中的HashSet一样（我们在JavaSE中已经讲解过了，HashSet本质上就是利用了一个HashMap，但是Value都是固定对象，仅仅是Key不同）它不允许出现重复元素，不支持随机访问，但是能够利用Hash表提供极高的查找效率。

向Set中添加一个或多个值：

```
sadd <key> <value>...
```

查看Set集合中有多少个值：

```
scard <key>
```

判断集合中是否包含：

```
-- 是否包含指定值
sismember <key> <value>
-- 列出所有值
smembers <key>
```

集合之间的运算：

```
-- 集合之间的差集
sdiff <key1> <key2>
-- 集合之间的交集
sinter <key1> <key2>
-- 求并集
sunion <key1> <key2>
-- 将集合之间的差集存到目标集合中
sdiffstore 目标 <key1> <key2>
-- 同上
sinterstore 目标 <key1> <key2>
-- 同上
sunionstore 目标 <key1> <key2>
```

移动指定值到另一个集合中：

```
smove <key> 目标 value
```

移除操作：

```
-- 随机移除一个幸运儿
spop <key>
-- 移除指定
srem <key> <value>...
```

那么如果我们要求Set集合中的数据按照我们指定的顺序进行排列怎么办呢？这时就可以使用SortedSet，它支持我们为每个值设定一个分数，分数的大小决定了值的位置，所以它是有序的。

我们可以添加一个带分数的值：

```
zadd <key> [<value> <score>]...
```

同样的：

```
-- 查询有多少个值
zcard <key>
-- 移除
zrem <key> <value>...
-- 获取区间内的所有
zrange <key> start stop
```

由于所有的值都有一个分数，我们也可以根据分数段来获取：

```
-- 通过分数段查看
zrangebyscore <key> start stop [withscores] [limit]
-- 统计分数段内的数量
zcount <key>  start stop
-- 根据分数获取指定值的排名
zrank <key> <value>
```

https://www.jianshu.com/p/32b9fe8c20e1

有关Bitmap、HyperLogLog和Geospatial等数据类型，这里暂时不做介绍，感兴趣可以自行了解。

持久化

我们知道，Redis数据库中的数据都是存放在内存中，虽然很高效，但是这样存在一个非常严重的问题，如果突然停电，那我们的数据不就全部丢失了吗？它不像硬盘上的数据，断电依然能够保存。

这个时候我们就需要持久化，我们需要将我们的数据备份到硬盘上，防止断电或是机器故障导致的数据丢失。

持久化的实现方式有两种方案：一种是直接保存当前**已经存储的数据**，相当于复制内存中的数据到硬盘上，需要恢复数据时直接读取即可；还有一种就是保存我们存放数据的**所有过程**，需要恢复数据时，只需要将整个过程完整地重演一遍就能保证与之前数据库中的内容一致。

RDB

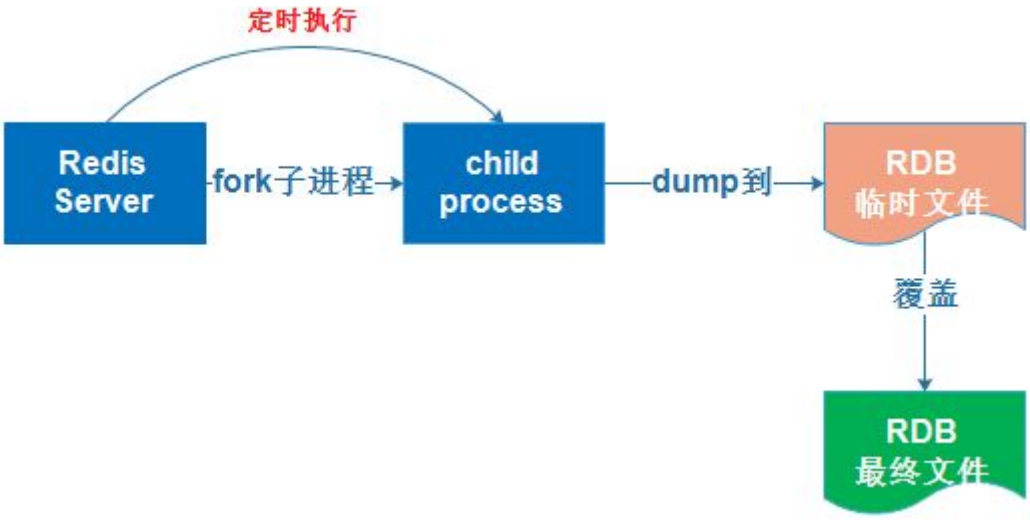
RDB就是我们所说的第一种解决方案，那么如何将数据保存到本地呢？我们可以使用命令：

```
save
-- 注意上面这个命令是直接保存，会占用一定的时间，也可以单独开一个子进程后台执行保存
bgsave
```

执行后，会在服务端目录下生成一个dump.rdb文件，而这个文件中就保存了内存中存放的数据，当服务器重启后，会自动加载里面的内容到对应数据库中。保存后我们可以关闭服务器：

```
shutdown
```

重启后可以看到数据依然存在。



虽然这种方式非常方便，但是由于会完整复制所有的数据，如果数据库中的数据量比较大，那么复制一次可能就需要花费大量的时间，所以我们可以每隔一段时间自动进行保存；还有就是，如果我们基本上都是在进行读操作，而没有进行写操作，实际上只需要偶尔保存一次即可，因为数据几乎没有怎么变化，可能两次保存的都是一样的数据。

我们可以在配置文件中设置自动保存，并设定在一段时间内写入多少数据时，执行一次保存操作：

```
save 300 10 # 300秒（5分钟）内有10个写入
save 60 10000 # 60秒（1分钟）内有10000个写入
```

配置的save使用的都是bgsave后台执行。

AOF

虽然RDB能够很好地解决数据持久化问题，但是它的缺点也很明显：每次都需要去完整地保存整个数据库中的数据，同时后台保存过程中也会产生额外的内存开销，最严重的是它并不是实时保存的，如果在自动保存触发之前服务器崩溃，那么依然会导致少量数据的丢失。

而AOF就是另一种方式，它会以日志的形式将我们每次执行的命令都进行保存，服务器重启时会将所有命令依次执行，通过这种重演的方式将数据恢复，这样就能很好解决实时性存储问题。

但是，我们多久写一次日志呢？我们可以自己配置保存策略，有三种策略：

- always：每次执行写操作都会保存一次
- everysec：每秒保存一次（默认配置），这样就算丢失数据也只会丢一秒以内的数据
- no：看系统心情保存

可以在配置文件中配置：

```
# 注意得改成也是
appendonly yes

# appendfsync always
appendfsync everysec
# appendfsync no
```

重启服务器后，可以看到服务器目录下多了一个`appendonly.aof`文件，存储的就是我们执行的命令。

AOF的缺点也很明显，每次服务器启动都需要进行过程重演，相比RDB更加耗费时间，并且随着我们的操作变多，不断累计，可能到最后我们的aof文件会变得无比巨大，我们需要一个改进方案来优化这些问题。

Redis有一个AOF重写机制进行优化，比如我们执行了这样的语句：

```
lpush test 666
lpush test 777
lpush test 888
```

实际上用一条语句也可以实现：

```
lpush test 666 777 888
```

正是如此，只要我们能够保证最终的重演结果和原有语句的结果一致，无论语句如何修改都可以，所以我们可以通过这种方式将多条语句进行压缩。

我们可以输入命令来手动执行重写操作：

```
bgrewriteaof
```

或是在配置文件中配置自动重写：

```
# 百分比计算，这里不多介绍
auto-aof-rewrite-percentage 100
# 当达到这个大小时，触发自动重写
auto-aof-rewrite-min-size 64mb
```

至此，我们就完成了两种持久化方案的介绍，最后我们再来进行一下总结：

- AOF：
  - 优点：存储速度快、消耗资源少、支持实时存储
  - 缺点：加载速度慢、数据体积大
- RDB：
  - 优点：加载速度快、数据体积小
  - 缺点：存储速度慢大量消耗资源、会发生数据丢失

事务和锁机制

和MySQL一样，在Redis中也有事务机制，当我们需要保证多条命令一次性完整执行而中途不受到其他命令干扰时，就可以使用事务机制。

我们可以使用命令来直接开启事务：

```
multi
```

当我们输入完所有要执行的命令时，可以使用命令来立即执行事务：

```
exec
```

我们也可以中途取消事务：

```
discard
```

实际上整个事务是创建了一个命令队列，它不像MySQL那种在事务中也能单独得到结果，而是我们提前将所有的命令装在队列中，但是并不会执行，而是等我们提交事务的时候再统一执行。

锁

又提到锁了，实际上这个概念对我们来说已经不算是陌生了。实际上在Redis中也会出现多个命令同时竞争同一个数据的情况，比如现在有两条命令同时执行，他们都要去修改a的值，那么这个时候就只能动用锁机制来保证同一时间只能有一个命令操作。

虽然Redis中也有锁机制，但是它是一种乐观锁，不同于MySQL，我们在MySQL中认识的锁是悲观锁，那么什么是乐观锁什么是悲观锁呢？

- 悲观锁：时刻认为别人会来抢占资源，禁止一切外来访问，直到释放锁，具有强烈的排他性质。
- 乐观锁：并不认为会有人来抢占资源，所以会直接对数据进行操作，在操作时再去验证是否有其他人抢占资源。

Redis中可以使用watch来监视一个目标，如果执行事务之前被监视目标发生了修改，则取消本次事务：

```
watch
```

我们可以开两个客户端进行测试。

取消监视可以使用：

```
unwatch
```

至此，Redis的基础内容就讲解完毕了，在之后的SpringCloud阶段，我们还会去讲解集群相关的知识，包括主从复制、哨兵模式等。

使用Java与Redis交互

既然了解了如何通过命令窗口操作Redis数据库，那么我们如何使用Java来操作呢？

这里我们需要使用到Jedis框架，它能够实现Java与Redis数据库的交互，依赖：

```
<dependencies>
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>4.0.0</version>
    </dependency>
</dependencies>
```

基本操作

我们来看看如何连接Redis数据库，非常简单，只需要创建一个对象即可：

```
public static void main(String[] args) {
    //创建Jedis对象
    Jedis jedis = new Jedis("localhost", 6379);

    //使用之后关闭连接
    jedis.close();
}
```

通过Jedis对象，我们就可以直接调用命令的同名方法来执行Redis命令了，比如：

```
public static void main(String[] args) {
    //直接使用try-with-resouse，省去close
    try(Jedis jedis = new Jedis("192.168.10.3", 6379)){
        jedis.set("test", "lbwnb");   //等同于 set test lbwnb 命令
        System.out.println(jedis.get("test"));  //等同于 get test 命令
    }
}
```

Hash类型的数据也是这样：

```
public static void main(String[] args) {
    try(Jedis jedis = new Jedis("192.168.10.3", 6379)){
        jedis.hset("hhh", "name", "sxc");   //等同于 hset hhh name sxc
        jedis.hset("hhh", "sex", "19");    //等同于 hset hhh age 19
        jedis.hgetAll("hhh").forEach((k, v) -> System.out.println(k+": "+v));
    }
}
```

我们接着来看看列表操作：

```
public static void main(String[] args) {
    try(Jedis jedis = new Jedis("192.168.10.3", 6379)){
        jedis.lpush("mylist", "111", "222", "333");  //等同于 lpush mylist 111 222
333 命令
        jedis.lrange("mylist", 0, -1)
                .forEach(System.out::println);    //等同于 lrange mylist 0 -1
    }
}
```

实际上我们只需要按照对应的操作去调用同名方法即可，所有的类型封装Jedis已经帮助我们完成了。

SpringBoot整合Redis

我们接着来看如何在SpringBoot项目中整合Redis操作框架，只需要一个starter即可，但是它底层没有用Jedis，而是Lettuce：

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

starter提供的默认配置会去连接本地的Redis服务器，并使用0号数据库，当然你也可以手动进行修改：

```yaml
spring:
  redis:
    #Redis服务器地址
    host: 192.168.10.3
    #端口
    port: 6379
    #使用几号数据库
    database: 0
```

starter已经给我们提供了两个默认的模板类：

```
@Configuration(
    proxyBeanMethods = false
)
@ConditionalOnClass({RedisOperations.class})
@EnableConfigurationProperties({RedisProperties.class})
@Import({LettuceConnectionConfiguration.class,
JedisConnectionConfiguration.class})
public class RedisAutoConfiguration {
    public RedisAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean(
        name = {"redisTemplate"}
    )
    @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        RedisTemplate<Object, Object> template = new RedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }

    @Bean
    @ConditionalOnMissingBean
    @ConditionalOnSingleCandidate(RedisConnectionFactory.class)
    public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        return new StringRedisTemplate(redisConnectionFactory);
    }
}
```

那么如何去使用这两个模板类呢？我们可以直接注入`StringRedisTemplate`来使用模板：

```
@SpringBootTest
```

```
class SpringBootTestApplicationTests {

    @Autowired
    StringRedisTemplate template;

    @Test
    void contextLoads() {
        ValueOperations<String, String> operations = template.opsForValue();
        operations.set("c", "xxxxx");   //设置值
        System.out.println(operations.get("c"));   //获取值

        template.delete("c");    //删除键
        System.out.println(template.hasKey("c"));   //判断是否包含键
    }

}
```

实际上所有的值的操作都被封装到了`ValueOperations`对象中，而普通的键操作直接通过模板对象就可以使用了，大致使用方式其实和Jedis一致。

我们接着来看看事务操作，由于Spring没有专门的Redis事务管理器，所以只能借用JDBC提供的，只不过无所谓，正常情况下反正我们也要用到这玩意：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

```
@Service
public class RedisService {

    @Resource
    StringRedisTemplate template;

    @PostConstruct
    public void init(){
        template.setEnableTransactionSupport(true);   //需要开启事务
    }

    @Transactional    //需要添加此注解
    public void test(){
        template.multi();
        template.opsForValue().set("d", "xxxxx");
        template.exec();
    }
}
```

我们还可以为RedisTemplate对象配置一个Serializer来实现对象的JSON存储：

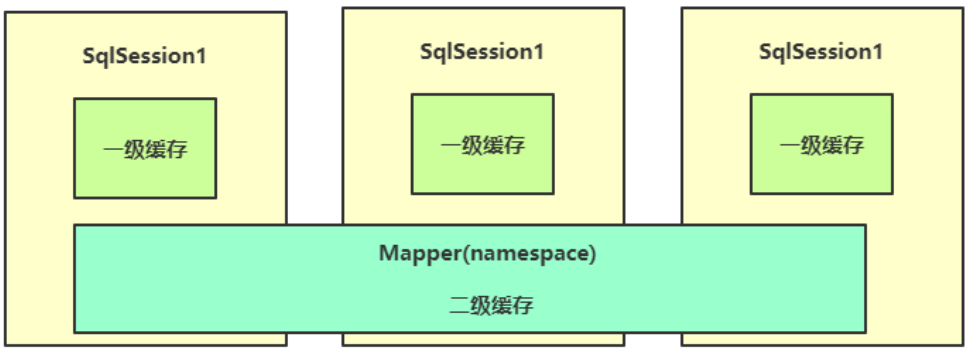
```
@Test
void contextLoad2() {
    //注意Student需要实现序列化接口才能存入Redis
    template.opsForValue().set("student", new Student());
    System.out.println(template.opsForValue().get("student"));
}
```

使用Redis做缓存

我们可以轻松地使用Redis来实现一些框架的缓存和其他存储。

Mybatis二级缓存

还记得我们在学习Mybatis讲解的缓存机制吗，我们当时介绍了二级缓存，它是Mapper级别的缓存，能够作用与所有会话。但是当时我们提出了一个问题，由于Mybatis的默认二级缓存只能是单机的，如果存在多台服务器访问同一个数据库，实际上二级缓存只会在各自的服务器上生效，但是我们希望的是多台服务器都能使用同一个二级缓存，这样就不会造成过多的资源浪费。



我们可以将Redis作为Mybatis的二级缓存，这样就能实现多台服务器使用同一个二级缓存，因为它们只需要连接同一个Redis服务器即可，所有的缓存数据全部存储在Redis服务器上。我们需要手动实现Mybatis提供的Cache接口，这里我们简单编写一下：

```
//实现Mybatis的Cache接口
public class RedisMybatisCache implements Cache {

    private final String id;
    private static RedisTemplate<Object, Object> template;

    //注意构造方法必须带一个String类型的参数接收id
    public RedisMybatisCache(String id){
        this.id = id;
    }

    //初始化时通过配置类将RedisTemplate给过来
    public static void setTemplate(RedisTemplate<Object, Object> template) {
        RedisMybatisCache.template = template;
    }

    @Override
    public String getId() {
```

```
        return id;
    }

    @Override
    public void putObject(Object o, Object o1) {
        //这里直接向Redis数据库中丢数据即可，o就是Key，o1就是Value，60秒为过期时间
        template.opsForValue().set(o, o1, 60, TimeUnit.SECONDS);
    }

    @Override
    public Object getObject(Object o) {
        //这里根据Key直接从Redis数据库中获取值即可
        return template.opsForValue().get(o);
    }

    @Override
    public Object removeObject(Object o) {
        //根据Key删除
        return template.delete(o);
    }

    @Override
    public void clear() {
        //由于template中没封装清除操作，只能通过connection来执行
                template.execute((RedisCallback<Void>) connection -> {
            //通过connection对象执行清空操作
            connection.flushDb();
            return null;
        });
    }

    @Override
    public int getSize() {
        //这里也是使用connection对象来获取当前的Key数量
        return template.execute(RedisServerCommands::dbSize).intValue();
    }
}
```

缓存类编写完成后，我们接着来编写配置类：

```
@Configuration
public class MainConfiguration {
    @Resource
    RedisTemplate<Object, Object> template;

    @PostConstruct
    public void init(){
        //把RedisTemplate给到RedisMybatisCache
        RedisMybatisCache.setTemplate(template);
    }
}
```

最后我们在Mapper上启用此缓存即可：

```
//只需要修改缓存实现类implementation为我们的RedisMybatisCache即可
@CacheNamespace(implementation = RedisMybatisCache.class)
@Mapper
public interface MainMapper {

    @Select("select name from student where sid = 1")
    String getSid();
}
```

最后我们提供一个测试用例来查看当前的二级缓存是否生效：

```
@SpringBootTest
class SpringBootTestApplicationTests {


    @Resource
    MainMapper mapper;

    @Test
    void contextLoads() {
        System.out.println(mapper.getSid());
        System.out.println(mapper.getSid());
        System.out.println(mapper.getSid());
    }

}
```

手动使用客户端查看Redis数据库，可以看到已经有一条Mybatis生成的缓存数据了。

Token持久化存储

我们之前使用SpringSecurity时，remember-me的Token是支持持久化存储的，而我们当时是存储在数据库中，那么Token信息能否存储在缓存中呢，当然也是可以的，我们可以手动实现一个：

```
//实现PersistentTokenRepository接口
@Component
public class RedisTokenRepository implements PersistentTokenRepository {
    //Key名称前缀，用于区分
    private final static String REMEMBER_ME_KEY = "spring:security:rememberMe:";
    @Resource
    RedisTemplate<Object, Object> template;

    @Override
    public void createNewToken(PersistentRememberMeToken token) {
        //这里要放两个，一个存seriesId->Token，一个存username->seriesId，因为删除时是通过username删除

 template.opsForValue().set(REMEMBER_ME_KEY+"username:"+token.getUsername(), token.getSeries());
        template.expire(REMEMBER_ME_KEY+"username:"+token.getUsername(), 1, TimeUnit.DAYS);
        this.setToken(token);
    }
```

```
    //先获取，然后修改创建一个新的，再放入
    @Override
    public void updateToken(String series, String tokenValue, Date lastUsed) {
        PersistentRememberMeToken token = this.getToken(series);
        if(token != null)
           this.setToken(new PersistentRememberMeToken(token.getUsername(),
series, tokenValue, lastUsed));
    }

    @Override
    public PersistentRememberMeToken getTokenForSeries(String seriesId) {
        return this.getToken(seriesId);
    }

    //通过username找seriesId直接删除这两个
    @Override
    public void removeUserTokens(String username) {
        String series = (String)
template.opsForValue().get(REMEMBER_ME_KEY+"username:"+username);
        template.delete(REMEMBER_ME_KEY+series);
        template.delete(REMEMBER_ME_KEY+"username:"+username);
    }


    //由于PersistentRememberMeToken没实现序列化接口，这里只能用Hash来存储了，所以单独编写
一个set和get操作
    private PersistentRememberMeToken getToken(String series){
        Map<Object, Object> map =
template.opsForHash().entries(REMEMBER_ME_KEY+series);
        if(map.isEmpty()) return null;
        return new PersistentRememberMeToken(
                (String) map.get("username"),
                (String) map.get("series"),
                (String) map.get("tokenValue"),
                new Date(Long.parseLong((String) map.get("date"))));
    }

    private void setToken(PersistentRememberMeToken token){
        Map<String, String> map = new HashMap<>();
        map.put("username", token.getUsername());
        map.put("series", token.getSeries());
        map.put("tokenValue", token.getTokenValue());
        map.put("date", ""+token.getDate().getTime());
        template.opsForHash().putAll(REMEMBER_ME_KEY+token.getSeries(), map);
        template.expire(REMEMBER_ME_KEY+token.getSeries(), 1, TimeUnit.DAYS);
    }
}
```

接着把验证Service实现了：

```
@Service
public class AuthService implements UserDetailsService {

    @Resource
    UserMapper mapper;
```

```
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Account account = mapper.getAccountByUsername(username);
        if(account == null) throw new UsernameNotFoundException("");
        return User
                .withUsername(username)
                .password(account.getPassword())
                .roles(account.getRole())
                .build();
    }
}
```

Mapper也安排上：

```
@Data
public class Account implements Serializable {
    int id;
    String username;
    String password;
    String role;
}
```

```
@CacheNamespace(implementation = MybatisRedisCache.class)
@Mapper
public interface UserMapper {

    @Select("select * from users where username = #{username}")
    Account getAccountByUsername(String username);
}
```

最后配置文件配一波：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .rememberMe()
            .tokenRepository(repository);
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth
            .userDetailsService(service)
            .passwordEncoder(new BCryptPasswordEncoder());
}
```

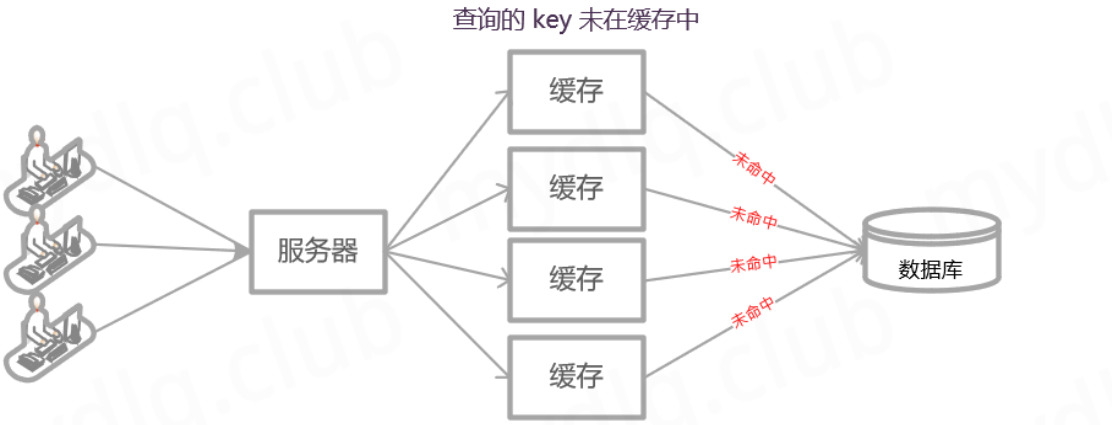
OK，启动服务器验证一下吧。

三大缓存问题

注意：这部分内容作为选学内容。

虽然我们可以利用缓存来大幅度提升我们程序的数据获取效率，但是使用缓存也存在着一些潜在的问题。

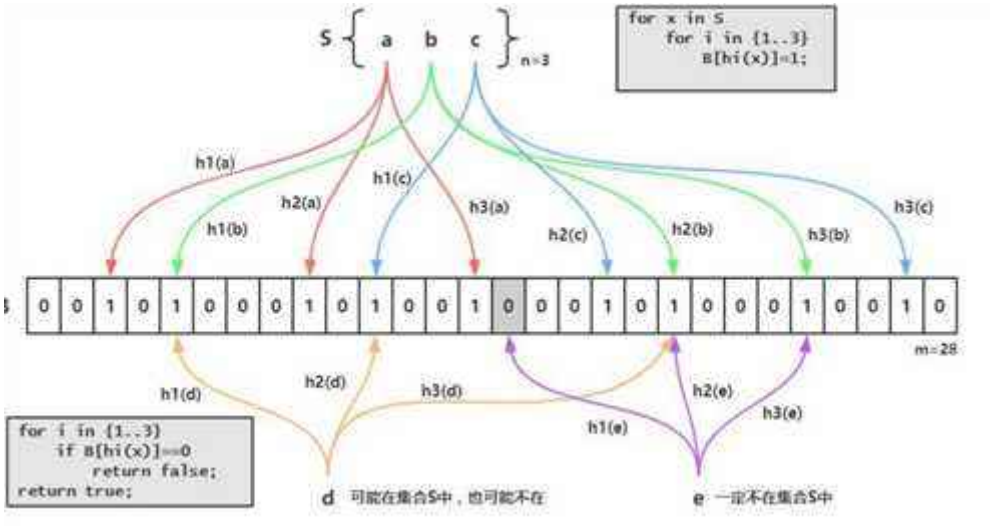
缓存穿透



当我们去查询一个一定不存在的数据，比如Mybatis在缓存是未命中的情况下需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

这显然是很浪费资源的，我们希望的是，如果这个数据不存在，为什么缓存这一层不直接返回空呢，这时就不必再去查数据库了，但是也有一个问题，缓存不去查数据库怎么知道数据库里面到底有没有这个数据呢?

这时我们就可以使用布隆过滤器来进行判断。什么是布隆过滤器？（当然不是打辅助的那个布隆，只不过也挺像，辅助布隆也是挡子弹的）

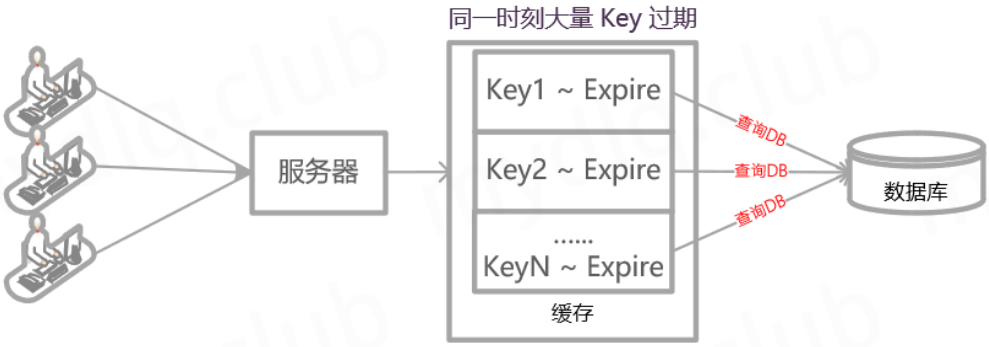


使用布隆过滤器，能够告诉你某样东西一定不存在或是某样东西可能存在。

布隆过滤器本质是一个存放二进制位的bit数组，如果我们要添加一个值到布隆过滤器中，我们需要使用N个不同的哈希函数来生成N个哈希值，并对每个生成的哈希值指向的bit位置1，如上图所示，一共添加了三个值abc。

接着我们给一个d，那么这时就可以进行判断，如果说d计算的N个哈希值的位置上都是1，那么就说明d可能存在；这时候又来了个e，计算后我们发现有一个位置上的值是0，这时就可以直接断定e一定不存在。

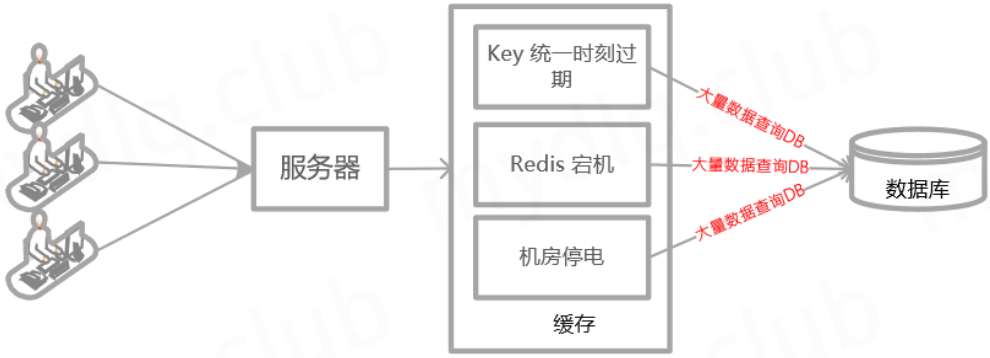
缓存击穿



某个 Key 属于热点数据，访问非常频繁，同一时间很多人都在访问，在这个Key失效的瞬间，大量的请求到来，这时发现缓存中没有数据，就全都直接请求数据库，相当于击穿了缓存屏障，直接攻击整个系统核心。

这种情况下，最好的解决办法就是不让Key那么快过期，如果一个Key处于高频访问，那么可以适当地延长过期时间。

缓存雪崩



当你的Redis服务器炸了或是大量的Key在同一时间过期，这时相当于缓存直接GG了，那么如果这时又有很多的请求来访问不同的数据，同一时间内缓存服务器就得向数据库大量发起请求来重新建立缓存，很容易把数据库也搞GG。

解决这种问题最好的办法就是设置高可用，也就是搭建Redis集群，当然也可以采取一些服务熔断降级机制，这些内容我们会在SpringCloud阶段再进行探讨。