# 大型电商分布式系统实践 第10周

DATAGURU专业数据分析社区

# 上节内容回顾

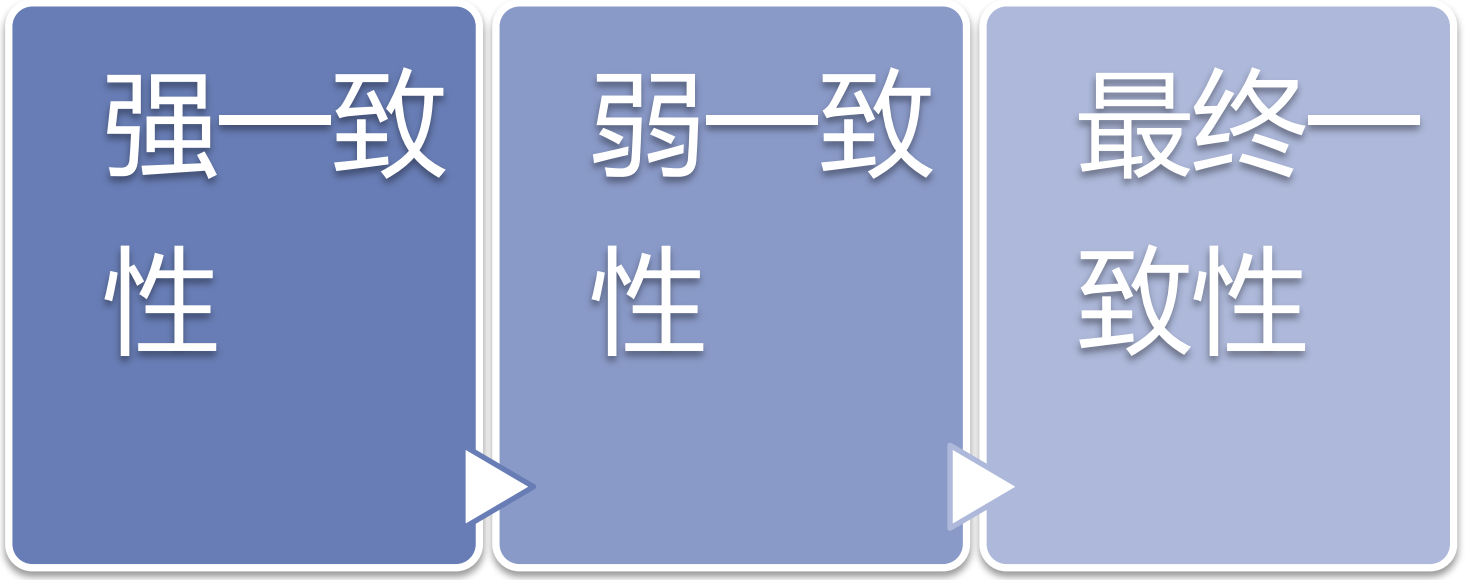
# 高并发系统与普通系统的区别

高并发系统设计与普通系统设计的区别在于，高并发系统既要保障系统的可用性和可扩展性，又要兼顾数据的一致性，还要处理多线程下线程安全的问题。任何细微问题，都有可能在高并发环境下被无限的放大，直至系统宕机。



# 数据一致性

分布式系统常常通过数据的复制来提高系统的可靠性和容错性，并且将数据的副本存放到不同的机器上，由于存在多个副本，使得维护副本一致性的代价很高。所以，许多分布式系统都采用弱一致性或者是最终一致性，来提高系统的性能和吞吐能力，这样，不同的一致性模型也相继被提出：



# 原子操作

原子操作指的是不可分割的操作，它要么执行成功，要么执行失败，不会产生中间状态，在多线程程序中，原子操作是一个非常重要的概念，它常常用来实现一些数据同步机制，具体的例子如java的原子变量、数据库的事务等等，原子操作也是一些常见的多线程程序bug的源头，并发相关的问题对于测试来说，并不是每次都能够重现，因此处理起来十分棘手。

# java的原子操作——一个count统计的例子

# java的原子操作——一个count统计的例子

当java代码最终被编译成字节码时，run()方法会被编译成这么几条指令：

```
public void run();
  Code:
   0:   aload_0
   1:   getfield        #17;
   4:   dup
   5:   getfield        #26; //获取count.count的值,并将其压入栈顶
   8:   iconst_1                //将int型1压入栈顶
   9:   iadd                    //将栈顶两int型数值相加,并将结果压入栈顶
   10:  putfield        #26; //将栈顶的结果赋值给count.count
   13:  aload_0
   14:  getfield        #19;
   17:  invokevirtual   #31;
   20:  return
}
```

# java的原子操作—原子变量实现count统计

# java的原子操作—原子变量实现count统计

native方法compareAndSwapInt在linux下的jdk的实现如下：

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj,
jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

注：该段代码来自openjdk6的源码，代码的路径为hotspot/src/share/vm/prims/unsafe.cpp

# java的原子操作—原子变量实现count统计

Unsafe_CompareAndSwapInt最终通过Atomic::cmpxchg(x, addr, e)来实现原子操作，而Atomic::cmpxchg在x86处理器架构下的linux下的jdk实现如下：

```
inline jint Atomic::cmpxchg(jint exchange_value, volatile jint* dest, jint compare_value){
  int mp = os::is_MP();
  __asm__ volatile(LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                   : "=a" (exchange_value)
                   : "r" (exchange_value),"a" (compare_value),"r" (dest),"r" (mp)
                   : "cc", "memory");
  return exchange_value;
}
```

注:该段代码来自openjdk6的源码，代码的路径为hotspot/src/os_cpu/linux_x86/vm/atomic_linux_x86.inline.hpp

# 数据库事务

数据库事务具有ACID属性，即原子性(Atomic)、一致性(Consistency)，隔离性(Isolation)，持久性(Durability)，为针对数据库的一系列操作提供了一种从失败状态恢复到正常状态的方法，使数据库在异常状态下也能够保持数据的一致性，并且，面对并发访问时，数据库能够提供一种隔离方法，避免彼此间的操作互相干扰。

# 数据库事务

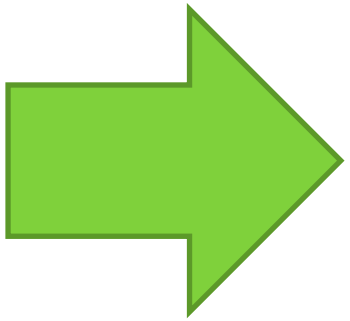
通过java进行数据库事务操作的代码：

```
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/hhuser",
"root", "123456");
conn.setAutoCommit(false);
try{
        Statement stmt = conn.createStatement();
        int insertResult = stmt.executeUpdate
        ("insert into hhuser set userid=125,nick = 'chenkangxian'");
        int updateResult = stmt.executeUpdate
        ("update hhuser set nick='chenkangxian@abc.com' where userid = 125");
        if(insertResult > 0 && updateResult > 0){
                conn.commit();
        }else{
                conn.rollback();
        }
}catch(Exception e){
        conn.rollback();
}
```

# 多线程同步--synchronized

还是前面Count计数的例子，通过在java中使用synchronized关键字和锁，实现线程间的同步：

```
public void run() {
        synchronized(count){
                count.count++;
        }
        ……
}
```



```
……
6:   monitorenter
7:   aload_0
8:   getfield
11:  dup
12:  getfield
15:  iconst_1
16:  iadd
17:  putfield
20:  aload_1
21:  monitorexit
……
```

# 多线程同步--ReentrantLock

在Count对象中加入ReentrantLock的实例：

```
private final ReentrantLock lock = new ReentrantLock();
```

然后在count.count++之前加锁，并且，++操作完成之后，释放锁给其他线程：

```
count.lock.lock();
count.count++;
count.lock.unlock();
```

# 典型案例—秒杀抽奖系统设计

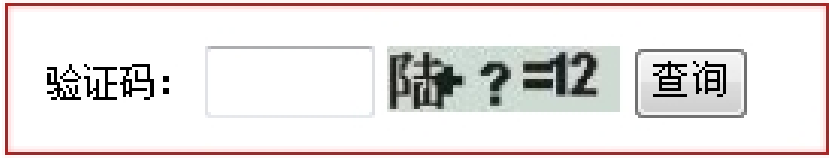
# 秒杀抽奖系统设计—防机器提交

简单验证码:


稍微复杂的验证码:

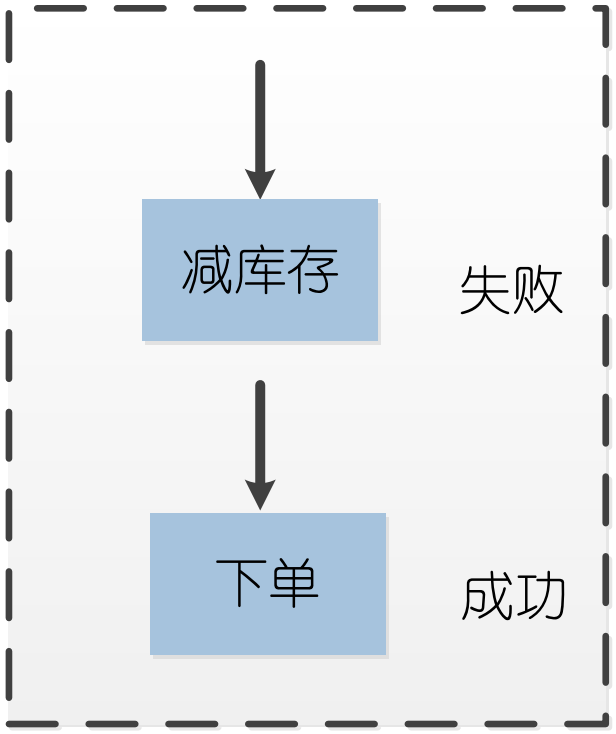

混淆过的验证码:


提问验证码:

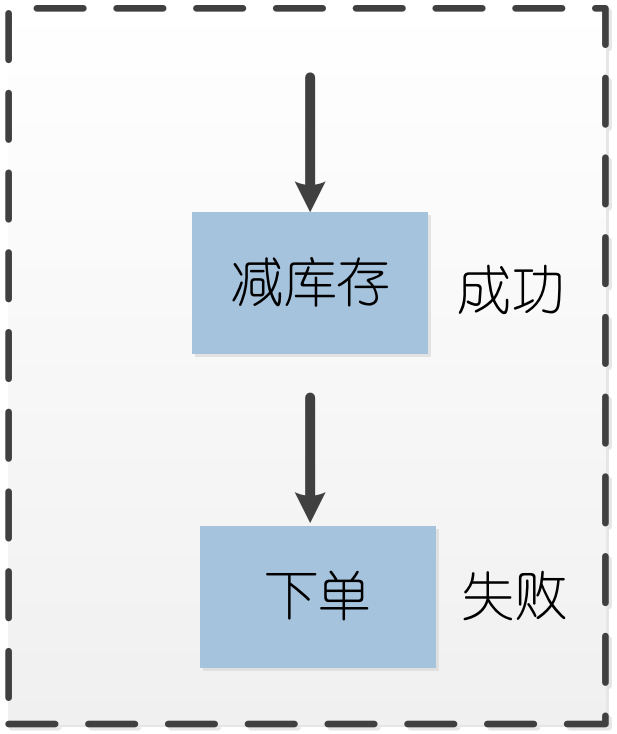



中文验证码:

# 秒杀抽奖系统设计—并发减库存




超卖




少卖

# 秒杀抽奖系统设计—库存拆分

| id | 名称 | 价格 | 库存 |
| --- | --- | --- | --- |
| 10000 | XX手机 | ¥100.00 | 10000 |

| id | 名称 | 价格 | 库存 |
| --- | --- | --- | --- |
| 10001 | XX手机 | ¥100.00 | 2000 |
| 10002 | XX手机 | ¥100.00 | 2000 |
| 10003 | XX手机 | ¥100.00 | 2000 |
| 10004 | XX手机 | ¥100.00 | 2000 |

一拆五

select sum(库存) from 商品表 ⟶ 查询库存

update 商品表 set 库存=库存-1 where id=1000x and sum(库存)>0 ⟶ 减库存

# Thanks

## FAQ时间