1. 穿透、击穿、雪崩
    1. 穿透：合法性校验、布隆过滤器
    2. 击穿：合理过期时间（如秒杀活动结束后再过期）
    3. 雪崩：均匀过期时间、加锁
2. **数据库**和Redis一致性

    旁路缓存（更新数据库后删除缓存）、加分布式锁

    > Redis过期策略：定时、懒惰、定期

3. Redis淘汰策略
    1. 不淘汰
    2. 淘汰
        1. 移除最快过期
        2. 随机删除「有过期时间」的
        3. 随机删除
        4. LRU、LFU
            1. 「有过期时间」的
            2. 全体中选
4. [todo]分布式锁
    1. synchronized和Lock那些锁都是基于jvm虚拟机的，在分布式环境下不起作用
    2. 分布式锁可以用redis的setNx命令或者Redisson来实现

        主要是redis有个setNx命令，这个命令是往redis中插入一条数据，**如果key不存在，插入该数据并返回1，视为加锁成功**；如果key已经存在，不插入数据并返回0，加锁失败。setNx命令能保证原子性主要是因为redis是单线程的
        （它快的原因是完全基于内存，采用**IO多路复用**和**部分多线程**加快性能瓶颈：**网络延迟**）

        > I/O多路复用就是用一个线程同时监听多个socket。

5. Redis中的`setNx`如何保证原子性：redis是单线程的
6. Redis主从同步：从库定期从主库中同步数据，保持主库和从库的数据一致。**读请求都交给从库处理，写请求交给主库处理**，达成读写分离，提高并发能力。

    从库向主库发出同步请求，并给出自己的数据集id和offset偏移量，主库根据id判断是不是第一次同步。通过RDB和repl-baklog
7. Redis集群保证高并发高可用
    1. 主从模式
    2. 哨兵模式：在主从模式的基础上添加了哨兵结点
    3. 分片集群：多组主从结点，并移除了哨兵。主结点们P2P的形式形成网络，请求发给任意一个redis结点，会自动将**请求路由到正确的结点上**处理。可以**解决海量数据存储和写请求高并发问题**

> 分片集群有一个插槽的概念，插槽上共有16384个槽位，每个主节点分到一部分的槽位。分片集群在处理数据的时候，会通过将key做哈希算法并和槽位数量取模，把key映射到其中一个槽位上，由分到这个槽位的主节点来处理

8. Redis脑裂

   主节点和集群断开连接，导致集群重新选举出了一个主节点，导致集群中出现两个主节点。如果此时**原主节点**依旧在接收客户端的写请求，在恢复后变成从节点导致这段时间的数据丢失。

9. [todo]select、poll、epoll命令的区别
   - 都是I/O多路复用模型
   - 前两种当用户得到socket就绪的报告的时候，必须轮询所有的socket才能得知是哪一个socket就绪并处理。epoll的实现中，在报告socket就绪的时候会同时把就绪的socket写入用户空间，用户可以直接使用

10. [todo]Redis网络模型

    I/O多路复用+事件分派+三个处理器

    > I/O多路复用监听socket请求，并将事件派发到对应的处理器处理.
    >
    > 命令请求处理器在接收到请求后，会先把请求转换为redis命令，再放入队列中等待执行，执行完毕后，将结果放入缓冲区
    >
    > 命令回复处理器会读取缓冲区中的数据进行回复

11. mysql定位慢查询

    > MySQL内置了一个慢查询日志的功能，默认是关闭的，可以手动开启。他会把执行时间超过预设值的sql记录到慢查询日志中

12. 执行sql查询语句过程：连接器、缓存查询、分析器、优化器、执行器
    1. 连接器建立TCP链接，验证身份，读取权限
    2. 查询语句才有此过程（mysql8.0不复存在）
    3. 分析器检查语法错误
    4. 优化器决定执行计划（是否索引、哪个索引）
    5. 执行器执行

13. update和select
    - Update会涉及三大日志：undo-log、redo-log、bin-log

14. 通过`explain`命令查询这条sql语句的执行计划
15. 聚簇索引、非聚簇索引

    > 聚簇索引的B+树的**叶结点上保存了完整的整行数据**，一张表只能有一个聚簇索引，如果有主键的话，**默认会用主键**来做聚簇索引
    >
    > 非聚簇索引就是我们为字段**手动建立的索引**

16. 字符串当主键
    1. 占据空间：索引树上除了索引字段也必须存储主键
    2. 每次插入都可能挪动其他节点在B+树中的位置（因为不是有序插入）

3. 字串占用空间大，导致B+树节点更大读取更慢（还可能使得每个节点存储的索引数变少）
4. 字符串比较更复杂，降低查询效率

- 用自增整数当主键

17. 什么是回表

    回表查询就是在使用索引的时候，我们**先要去普通索引上查找一遍**，然后又要**回到聚簇索引中查找完整数据的现象**

18. 什么是覆盖索引

    覆盖索引就是在查询的时候可以直接从索引里获得**所有**需要的数据，不需要进行回表。比如查的就是id，就不需要回表了。

19. [todo]mysql超大分页怎么处理：覆盖索引+子查询

    比如使用limit时需要排序：在子查询中只查找id，然后通过覆盖索引的方式让这条子查询只走普通索引树。就不会读取完整的整行数据，能大幅度减少读取和排序数据的时间。然后子查询结果的id值，去聚簇索引里读取id对应的完整行数据就行了

20. 创建索引的原则
    1. 区分度大的字段
    2. 数据量大切经常查询、排序、分组的表里的字段
    3. 控制索引数量
    4. 尽量使用联合索引
21. 最左前缀法则

    使用联合索引时，查询要从索引的**最左前列开始**，并且**不跳过**索引中的列，否则会让后面字段的索引失效

22. 索引失效发生的情况
    1. 用了联合索引但违背最左原则
    2. 用了联合索引，范围查询「联合索引中右侧的索引」是不生效的

       > 比如：联合索引 (col1, col2)，而查询条件为 col2 > 10

    3. 对索引使用函数、计算、类型转化：比如`id+1=10`

       > **MySQL 在遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较。**
       >
       > ```
       > select * from t_user where phone = 1300000001;  # 索引失效，phone背转换为数字了（类型转化）
       > select * from t_user where id = "1";  # 索引不失效
       > ```

    4. 对索引使用左或者左右模糊匹配。
    5. where子句中的OR左右有非索引字段
23. 事务的特性

    一组操作必须一起成功或者一起失败。ACID：原子、一致、隔离、持久

24. 事务隔离级别：读**未提交**（脏读），读**已提交**，**可重复读**，**串行化**

25. mysql三大日志
    1. undo：修改前把旧数据存入。也用于MVCC
    2. redo：修改数据时先修改内存，并写入redolog等待系统刷盘（prepare）提交（commit）
    3. bin：**记录**了所有数据库**表结构变更和表数据修改**：追加写，STATEMENT、ROW、MIX
    4. > Redis日志： RDB、AOF、混合
26. 事务隔离性如何保证
    1. MVCC：multiple version concurrent controll 多版本并发控制
        1. 读已提交：每次查询时ReaView
        2. 可重复读：事务开启时ReadView

        写操作：先复制数据，修改副本。事务提交才会更新到数据库中被其他可见
    2. 锁：写写冲突
27. Mysql主从同步：
    1. Mysql在进行每一次修改数据库中内容的操作的时候，会把**执行的SQL写入到bin-log**中
    2. 从库会专门有个线程来**读取主库的bin-log**日志，并将内容写入到从库的中继日志**relay-log**里面
    3. 然后从库读取relay-log并**执行相应的操作**，达成数据同步
28. 分库分表
    1. **水平**分库、水平分表：**数据拆分**。表结构一样，数据拆到一样的若干表中
    2. **垂直**分库、垂直分表：把拥挤在一起的业务/字段分开（建立关系表）
29. **单例Bean**是无状态的，**线程安全**
30. Spring Bean的生命周期：实例化 -> 属性赋值 -> 初始化 -> 销毁
31. Bean的作用域：
    1. `singleton` : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
    2. `prototype` : 每次请求都会创建一个新的 bean 实例。
    3. `request` :每个 HTTP 请求处理过程中都会创建一个新的 Bean 实例
    4. `session` : 每个用户会话（Session）中都会创建一个新的 Bean 实例

        > 用户会话是指用户与 Web 应用之间的一段交互期间，通常从用户登录到退出登录为一次会话
    5. `global-session`： 全局 session 作用域，仅仅在基于 Portlet 的 web 应用中才有意义，Spring5 已经没有了。Portlet 是能够生成语义代码（例如：HTML）片段的小型 Java Web 插件。它们基于 portlet 容器，可以像 servlet 一样处理 HTTP 请求。但是，与 servlet 不同，每个 portlet 都有不同的会话。
32. 什么是AOP

    面向切面编程，可以把一部分共通的逻辑抽出来单独编写，并**通过切点定位到需要被增强的类和方法**，然后通过环绕通知，可以在**不侵入代码的情况下增强类或者方法**的功能。比如**写日志**就是很经典的使用场景
33. Spring中的事务

    在spring中使用事务有两种方法，一种是编程式事务，一种是注解式事务。

    编程式事务需要拿到tranactionalTemplate，在代码里显式地开启或者回滚事务。

注解式事务通过@transactional注解就可以使用事务。注解式事务的**底层使用的就是动态代理和AOP实现**的

34. Spring事务失效的场景
    1. 自己在方法中捕获了异常并处理，**不向外抛出异常而是自己处理**，会导致事务失效。
    2. 方法中**抛出的异常不是运行时异常**
    3. 方法**不是public**的
35. [todo]Spring循环引用
    1. 三级缓存：
        1. **一级缓存（singletonObjects）**：存放**最终形态**的 Bean（已经实例化、属性填充、初始化）
        2. **二级缓存（earlySingletonObjects）**：存放**过渡** Bean（半成品，尚未属性填充）
        3. **三级缓存（singletonFactories）**：存放`ObjectFactory`，`ObjectFactory`的`getObject()`方法（最终调用的是`getEarlyBeanReference()`方法）可以**生成原始 Bean 对象或者代理对象**（如果 Bean 被 AOP 切面代理）。**三级缓存只会对单例 Bean 生效。**

        > 流程**举例**：A包括B，B包括A。创建A，缺B。去创建B，但A也没好。去**三级缓存**中调用`getObject()`方法获取**A的前期暴露对象**（由`getEarlyBeanReference()`生成，然后把**前期暴露对象放入二级缓存**，然后B借用它来创建）

    2. 懒加载：

        没有被标记为懒加载，**那么它会在 Spring IoC 容器**启动的过程中被创建和初始化**。

        被标记为懒加载，第一次被请求时才创建
36. [todo]构造方法循环依赖
37. [todo]SpringMVC执行流程
38. [todo]SpringBoot自动配置原理
39. Spring常见注解：参见Spring八股
40. [todo]Mybatis执行流程
    1. 读取配置文件：mybatis-config.xml为MyBatis的全局配置文件。这个核心配置文件最终会被封装成一个Configuration对象
    2. 加载映射文件：
    3. 构造会话工厂获取`SqlSessionFactory`：`SqlSessionFactory builder = new SqlSessionFactoryBuilder().build(inputStream);`
    4. 创建会话对象`SqlSession`：
    5. Executor执行器：MyBatis的核心，负责**SQL语句的生成和查询缓存的维护**，它将根据SqlSession传递的参数动态地生成需要执行的SQL语句，同时负责查询缓存的维护
    6. MappedStatement对象：
    7. 输入参数映射：
    8. 封装结果集：
41. Mybatis可以执行批量插入，然后返回数据库主键列表。（JDBC都可以）
42. Mybatis仅支持association（一对一）关联对象和 collection（一对多）关联集合对象的**延迟加载**：`lazyLoadingEnabled=true|false。`

> 底层原理：使用 `CGLIB` 创建目标对象的代理对象，当**调用目标方法时，进入拦截器方法**，比如调用 `a.getB().getName()`，拦截器 `invoke()` 方法发现 `a.getB()` 是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 `a.getB().getName()` 方法的调用
>
> 即：Mybatis会返回一个**代理对象而不是实际的数据对象**，该**代理对象会拦截所有访问操作**，并判断当前访问的属性是否已经被加载。如果该属性未被加载，则通过SQL语句查询出该属性并进行加载；如果该属性已被加载，则直接返回该属性的值，不再进行加载操作

43. MyBatis 的 xml 映射文件中，不同的 xml映射文件，id 是否可以重复

    配置了`namespace`的不同XML文件中，id可以重复。但如果没有namespace则不能重复。

    > 原因就是 namespace+id 是作为 `Map<String, MappedStatement>` 的 key 使用

44. Mybatis如何执行批处理

    使用 `BatchExecutor` 完成批处理

```java
SqlSession sqlSession = sqlSessionFactory.openSession(ExecutorType.BATCH);//设置成BATCH模式
try {
    // 开启批处理模式
    sqlSession.getConnection().setAutoCommit(false);
    // 执行批量操作
    for (YourEntity entity : entities) {
        sqlSession.insert("com.example.mapper.YourMapper.insert", entity);
    }
    // 提交批处理操作
    sqlSession.commit();
} finally {
    sqlSession.close();
}
```

45. Mybatis有哪些Executor执行器

    - `SimpleExecutor`： **每执行一次** update 或 select，**就开启一个 Statement 对象，用完立刻关闭** Statement 对象。
    - `ReuseExecutor`： 执行 update 或 select，以 sql 作为 key 查找 **Statement 对象，存在就使用，不存在就创建**，用完后，不关闭 Statement 对象，而是放置于 Map<String, Statement>内，供下一次使用。简言之，就是重复使用 Statement 对象。
    - `BatchExecutor`：执行 **update**（没有 select，JDBC **批处理不支持 select**），将所有 sql 都添加到批处理中（addBatch()），**等待统一执行**（executeBatch()），它缓存了多个 Statement 对象，每个 Statement 对象都是 addBatch()完毕后，等待逐一执行 executeBatch()批处理。与 JDBC 批处理相同。

    作用范围：`Executor` 的这些特点，都严格限制在 SqlSession 生命周期范围内。

46. Mybatis可否映射Enum类型

    MyBatis 可以映射任何对象到表的一列上。映射方式为自定义一个 `TypeHandler`，实现 `TypeHandler` 的 `setParameter()` 和 `getResult()` 接口方法

47. MyBatis 的 xml 映射文件和 MyBatis 内部数据结构之间的映射关系

    `<parameterMap>` 标签会被解析为 `ParameterMap` 对象，其每个子元素会被解析为 `ParameterMapping`对象。

    `<resultMap>` 标签会被解析为 `ResultMap` 对象，其每个子元素会被解析为 `ResultMapping` 对象。

    `<select>、<insert>、<update>、<delete>` 被解析为 `MappedStatement` 对象，标签内的 sql 会被解析为 `BoundSql`对象

48. Mybatis一二级缓存

    1. 一级缓存：多次查询条件完全相同的SQL

        - **一级缓存无过期时间，只有生命周期**。当**会话结束**时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉。spring整合之后，如果没有事务，一级缓存是没有意义的

    `session`：一个MyBatis会话中执行的所有语句，都会共享这一个缓存

    `statement`：缓存只对当前执行的这一个`Statement`有效

    > 一级缓存时执行**commit，close，增删改**等操作，就会**清空**当前的一级缓存。为了避免脏读

    2. 二级缓存：不建议开启

        二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。关闭sqlsession后(close)，才会把该sqlsession一级缓存中的数据添加到namespace的二级缓存中。

        二级缓存是建立在同一个namespace下的，如果对表的操作查询可能有多个namespace，那么得到的数据就是错误的。

        > 在查询订单详情时我们需要把订单信息也查询出来，那么这个订单详情的信息被二级缓存在orderDetailMapper的namespace中，这个时候有人要修改订单的基本信息，那就是在orderMapper的namespace下修改，他是不会影响到orderDetailMapper的缓存的，那么你再次查找订单详情时，拿到的是缓存的数据，这个数据其实已经是过时的.
        >
        > 1）对该表的操作与查询都在同一个namespace下，其他的namespace如果有操作，就会发生数据的脏读。
        >
        > 2）对关联表的查询，关联的所有表的操作都必须在同一个namespace。

        - **二级缓存有过期时间，但没有后台线程进行检测**

        > 当对SqlSession执行更新操作（**update、delete、insert）后commit**时，不仅清空其自身的一级缓存（执行更新操作的效果），也**清空**二级缓存

49. [todo] [SpringCloud问题63-79]SpringCloud的组件有哪些

    网关、注册中心、负载均衡、服务过程调用、服务保护、链路追踪[todo]

50. [RabbitMQ/kafka问题80-92]

51. ArrayList：默认长度10，**真正放入元素才会分配容量**，扩容到**1.5倍**（右移实现除2）。实现了 **`List`**，**`Cloneable`**，**`Serializable`**以及**`RandomAccess`** 接口。只能存储对象，允许使用范型，创建时不用指定大小

    ConyOnWriteArrayList：线程安全且**只有写写才会互斥**。get可能读到旧值。**不会留冗余空间**

LinkedList：实现了`List`,`Cloneable`,`Serializable`以及**`Deque`(双端队列)**

52.
    - 数组=>list，可以用arrays.**asList**或者**stream**流。但注意使用asList后，List和数组指向**同一地址**，修改List会影响数组
    - List=>数组，直接调用list的**toArray()**方法，它互不影响
53. HashMap

    初始化16，超过阈值0.75回扩容变2倍（ArrayList是1.5）。线程不安全（但注意hashTable线程安全）

    可以存储key/value为null。但只能有一个key为null。（hashTable不能）

    是hashSet的底层。

    Hash表+红黑树：**数组的长度小于 64，那么会选择先进行数组扩容。当链表长度大于阈值（默认为 8）时，将链表（尾插法）转化为红黑树**。

    LinkedHashMap：在 `HashMap` 基础上维护一条双向链表，且被访问的数据会被移到末尾（可用作LRU）

    > Java1.7中使用头插法，多线程情况下可能导致形成环形链表A->B->A

54. ConcurrentHashMap
    - jdk1.7中，它的实现是通过segment数组+数组+链表实现的
    - 在jdk1.8中，采用了和hashmap一样的node数组+链表/红黑树的设计。**加锁只在node结点上加锁，一次锁住一个链表或者红黑树，锁的粒度更细**，并发度更高
55. 进程和线程：线程共享进程的地址空间、JVM中线程共享**堆**（包含**字符串常量池**）。独占虚拟机栈、本地方法栈、PC
56. 创建线程的方式：严格来说**只有`start()`**一种方式
    1. `runnable`和`callable`区别：分别只有`run()`和`call()`方法
        1. `Runnable` 不返回结果，也不能抛出检查异常，适用于简单任务。常与`Thread`类或`ExecutorService`的`execute()`方法一起使用。
        2. `Callable` 返回结果，并且可以抛出检查异常，适用于需要返回结果和处理异常的复杂任务。常与`ExecutorService`的`submit()`方法一起使用，该方法**返回一个`Future`对象**，可以用于获取任务的结果。
    2. `run`和`start`区别：
        1. **`run()` 方法**：如果直接调用 `run()` 方法，它只是一个普通的 Java 方法调用，不会启动新线程，代码在当前线程中执行。

           每个继承自 `Thread` 类的线程或者实现了 `Runnable` 接口的线程必须重写`run()`
        2. **`start()` 方法**：调用 `start()` 方法会启动一个新线程，在新线程中执行 `run()` 方法的代码，具有并发执行的效果
57. 操作系统中**进程**有哪些状态：

    创建态，就绪态，执行态，阻塞态，终止态。

    线程生命周期：没有就绪态而变为RUNNABLE
    - **NEW**: 初始状态，线程被创建出来但没有被调用 `start()` 。
    - **RUNNABLE**: 运行状态（但对应**OS**分类中的**就绪或阻塞态**）线程被调用了 `start()`等待运行的状态。
    - **BLOCKED**：阻塞状态（不同于OS中的阻塞），需要等待**锁**释放。

- **WAITING**：等待状态，表示该线程需要等待**其他线程**做出一些特定动作（通知或中断）。
  - **TIME**_WAITING：**超时等待状态**，可以在指定的时间后自行返回而不是像 WAITING 那样一直等待。
- **TERMINATED**：终止状态，表示该线程已经运行完毕。

58. 如何保证线程按顺执行：
    1. 使用`join()`方法：thread1.join()即表示T1执行完后再向后
    2. 使用 `CountDownLatch`：实际上是用**锁**控制。资源数设置为1，每个线程执行完释放。
    3. 使用 `ExecutorService`和 `Future`：设置线程池，用`Future.get()`等待future执行完成
59. `Notify()`和`notifyAll()`：`notify()`是从wait队列里**唤醒一个**，`notifyAll()`会唤醒**全部**
60. `sleep()`是Thread类的，**synchronized**方法中使用**不会释放锁**

    `wait()`是**Object中定义**的，synchronized方法中使用后**会释放锁**
61. 如何停止一个线程：
    1. 使用`thread.interrupt()`，线程将抛出 `InterruptedException`
    2. 如果使用 `ExecutorService` 来管理线程，可以使用 `shutdown()` 和 `shutdownNow()` 方法来停止线程池中的线程
       - `shutdown()` 方法会平滑地关闭线程池，等待所有任务完成。
       - `shutdownNow()` 方法会试图停止所有正在执行的任务，并返回等待执行的任务列表。
    3. 标志位方法：难懂

       > `volatile` 关键字修饰的 `running` 标志位来控制线程的运行状态。当主线程调用 `stop()` 方法时，`running` 被设置为 `false`，线程会退出循环并停止运行
62. 线程池：
    - 线程池的核心参数：核心线程数、最大线程数、阻塞队列、临时线程的存活时间（以及单位）、线程工厂、饱和策略、、
    - 流程：核心线程数->最大线程数->（临时线程）->阻塞队列->饱和（执行饱和策略如丢弃最早/此任务）
    - 常见阻塞队列：
      - ArrayblokingQueue：有界、数组、FIFO
      - LinkedblokingQueue：无界、链表、FIFO
      - SynchronizedQueue：每次插入都需要等取出操作后才能运行
      - DelayedworkQueue：优先级队列、执行时间最靠前的
    - 核心线程数设置：N表示CPU核心数
      - CPU密集：N+1
      - IO密集：2N+1
    - 线程池种类：
      - 固定线程数：核心=最大
      - 单线程线程池
      - 缓存线程池：核心=0，最大=∞。使用SynchronizedQueue队列，不存储元素

        - 延迟线程池：使用DelayEdWorkQueue，支持定时任务或者周期执行。
    - 不建议使用Executors创建线程池（他会使用大量LinkedBlockingQueue无界队列）。建议使用ThreadPoolExecutor
63. CountDownLatch：允许N个线程阻塞在一个地方，所有线程执行完后通过。
64. Semaphore关键字：
65. **Synchronized关键字**：原子、**可见**、有序
    - **只能**实现非公平锁，构造方法不能使用 synchronized 关键字修饰（构造方法本身是线程安全的）

    底层原理：

    > 基于jvm，每个java对象都有一个内置的moniter，这个moniter是通过c++实现的。它内部维护了一个owner指向持有它的线程，一个entryList用于存放获取锁阻塞的线程，一个waitinglist用于存放调用了wait方法的线程，还有一个count计数用于支持锁重入。当一个进程要进行加锁操作的时候，它先检查该对象的moniter的owner是否为null，是的话，获取该锁并把owner改成自己，不是的话放入进入entryList，当锁的拥有者释放锁后，owner重新指向空，其他线程得以开始争夺锁
    >
    > 其实wait/notify等方法也依赖于monitor对象。由monitor实现的锁属于重量级锁

    Synchronized和Lock：

    - Synchronized关键字，Lock是接口
    - Synchronized是**隐式加锁**，而Lock是显式的
    - Synchronized可以作用于方法上，lock只能作用于方法块
    - Synchronized底层是通过monitor实现的，lock是通过AQS实现的
66. **Volatile关键字：**
    1. 保证该变量对所有线程可见，要求访问此变量时到主存读取最新值（而不是线程空间的副本）
    2. 阻止指令重排序，任何读（写）操作不能重排序到此变量的读（写）操作之前（后）。
67. AQS：线程之间的同步机制

    维护一个State变量以及抽象队列，线程访问时**读取state值**。如果为0则使用**CAS**（确保多个线程同时抢state资源的原子性）更新为1（视为获取了锁）。**如果不为0则把线程封装成虚拟队列的节点，放入队列排队**。

    可以公平也可以非公平：公平性体现在队列中线程顺序获取锁，但如果新来的线程可以和队列头的线程一起竞争则非公平。
68. [todo]ReenTrantLock：类似synchronized，默认使用非公平锁。JDK层面实现。
69. 锁的等级、锁的升级过程

    锁的等级分别为**偏向锁**、**自旋锁**（也叫轻量级锁）、**重量级锁**（monitor实现）
    - monitor是依赖于操作系统底层的mutex lock互斥锁实现的，因此每次阻塞或者唤醒线程都**涉及用户态和内核态的切换**

    > 当只有一个线程在获取锁的时候，此时锁会**从偏向锁开始**，进程只在第一次获取锁的时候进行**CAS操作，然后在**锁**的对象**头**中**设置自己的线程ID**，当要重复获取这个锁的时候，只需要检查线程ID是不是自己就行，不需要进行CAS。

> 此时如果**又来一个线程**，那么在线程获取锁失败后就会将当前锁升级为**自旋锁**，锁的持有权依旧属于原来的线程，只是**新来的线程会不断通过CAS试图获取锁，失败了就会原地自旋。当自旋的次数到达阈值或者出现了两个以上的线程争夺锁**的时候，锁就**升级为重量级锁**。

70. 谈谈CAS

    CompareAndSwap，是一种乐观锁。预期值，原值，和新值。

    > 线程先读取原值并将这个值赋给预期值，然后对原值进行自己的操作得到新值，要新值同步到主存时，线程先读取主存中现在的数据并和预期值对比，如果一致，就把新值更新到主存中。如果不一致，就发生自旋。（**新值同步到主存时看看主存值是不是原值**）

71. 乐观锁和悲观锁
    1. 乐观锁：**不加锁**，在**写入前确认**该值是不是已经被其他线程修改过。*一般通过版本号或者时间戳之类的方式来确认数据是否被其他线程修改*
    2. 悲观锁：**加锁阻塞**
72. [todo]谈谈JMM
73. 谈谈JVM
    - 线程共享：堆、方法区
        - 堆：**对象、数组**。新生代、老年代
        - 方法区：**类信息、静态变量/方法、运行时常量池**（常量池是类中的，被加载后进入运行时常量池）
    - 线程私有：虚拟机栈、本地方法栈、程序计数器
        - 栈：局部变量、中间结果、方法帧。栈内存过大会导致最大线程数变小。
            - 递归次数过多会导致栈溢出
            - 局部变量只要不逃逸出方法外部，就线程安全。
    - 直接内存由操作系统管理（而不是JVM），也是共享的。用于作为NIO缓冲区。
    - **GC只在堆**上进行
74. ThreadLocal
    1. 底层原理：每个线程都有一个自己的ThreadLocalMap。Key是ThreadLocal的弱引用，Value保存的值。以它自己为Key去查询当前线程在ThreadLocalMap中的值
    2. 内存泄漏：Key是弱引用，但Value是强引用不会被回收。当ThreadLocal对象被GC回收。ThreadLocalMap会出现Key为null的值。
75. 一些并发基础知识
    1. 互斥、请求和保持、循环等待、不可剥夺
    2. 并发问题的主要原因是破坏了：原子性、顺序性、内存可见性
76. 类加载器：是一个对象
    1. 把`*.class`加载到JVM中。
    2. 加载过程：**加载**、**链接**（验证、准备、解析）、**初始化**、使用、销毁
        - 启动类加载器**BootStrap**ClassLoader：最顶层

        - 扩展类加载器**Extension**ClassLoader：加载常用的jar包和类
        - 应用程序类加载器**App**ClassLoader：面向用户
            - 自定义加载器**Custom**ClassLoader
    3. 双亲委派模型：优先把任务交给自己的父类加载器来执行，直到最顶层（以保证一个类只被加载一次）。打破需要使用自定义类加载器`loadClass`

77. GC回收
    - 对象没有被引用就是垃圾
        - 引用计数器（但循环引用会出问题）
        - 可达性分析，通过GCRoo他引用链查找（GCRoot无法达到就是垃圾）
    - 回收算法：
        - **标记-清除**：大量碎片
        - **标记-复制**：常用于新生代（存活对象较少）内存一分为二
        - **标记-整理**：常用于老年代
        - **分代收集**：新生代被继续分为伊甸区（E）和幸存区（S）。新对象在E区，经历一次GC后进入S区。经历GC15次后进入老年代。
    - **Minor**GC对**新生代**
    - **Mixed**GC**新生代和部分老年代**
    - FullGC整个堆内存
78. JVM有哪些垃圾回收器
    1. 串行垃圾回收器：标记-复制、标记-整理
    2. 并行垃圾回收器：Parallel Scavenge（新生代）+ Parallel Old（老年代）**JDK8**默认收集器。关注吞吐量
    3. CMS：关注用户线程暂停时间。真正的并发收集器
        1. 初始标记：暂停所有其他线程，标记与Root相连对象
        2. 并发标记：同时开启GC可达对象标记（根据第一步）和用户线程，记录引用更新
        3. 重新标记：暂停用户线程，修正并发标记期间的更新
        4. 并发清除：堆未标记区域清楚
    4. [todo]G1：**JDK9以后**的默认收集器，整体上“标记-整理”，局部上“标记-复制”。分为多个区域，根据预期停顿时间选择优先级最大的一块区域回收
    5. ZGC：标记-复制，不受堆内存大小限制
79. Java中的几种引用
    1. **强引用**：`new`就是强引用，不会被回收
    2. **弱引用**：只要检测到了就回收
    3. **软引用**：内存不足会被回收
    4. **虚引用**：配合引用队列，通知对象已被回收。
80. [todo]JVM调优

81. 设计模式
    1. 简单工厂模式
    2. 工厂方法模式
    3. 抽象工厂模式
    4. 单例模式
82. [todo]日志
83. 408特辑
    1. OSI七层：物理、数据链路、网络、传输、会话、表示、应用
    2. TCP：SYN、ACK、seq、ack

       三次握手才可以阻止重复历史连接的初始化

       四次挥手主要是FIN表示发送方不再对另一方发送数据，不代表另一方也不发送数据了。故而需要双发都发送FIN（也需要两次ACK）
    3. TCP和UDP区别

TCP	**UDP**
面向连接、可靠、字节流	无连接、不可靠、性能高
长文切片	整个包发送

    4. 网络拥塞控制算法：慢开始、拥塞避免、快重传、快恢复
84. HTTP短/长链接
    1. HTTP端口：80；HTTPS端口：443；HTTP2.0基于HTTPS
        1. HTTP2.0会将头部进行压缩，如果在一次请求中多个请求使用相同或者相似的首部，那么，协议会消除重复的部分。HTTP2.0在服务端和客户端上都维护了一张头部信息表，所有字段都会存入该表并有自己的索引号，只要发送索引号就可以了
        2. HTTP2.0使用二进制发送数据（更快）、引入Stream流的概念
        3. HTTP2.0支持服务器推送
        4. HTTP3.0使用UDP+QUIC协议来保证可靠性
    2. HTTPS在TCP三次握手后还要经过 SSL/TLS握手（四次通信）
        1. 客户端发送ClientHello请求：包含TLS版本，随机数，客户端支持算法
        2. 服务器收到后响应：包含TLS版本，随机数，服务端支持算法，以及**服务器数字证书**
        3. 客户端用CA公钥验证数字证书真实性后取出公钥加密报文（请求中含有随机数）并发送，结束通知并把所有数据作为个人摘要发送给服务器校验
        4. 服务器用私钥解密报文，用三个随机数计算密钥，并用密钥加密应答请求。
85. java方法的调用过程

对于x.f(a)方法的这样一个方法，先查找x的类型，然后到x的方法表里面找名字为F的所有方法，看有没有参数列表符合的方法，当找到这个方法，且**方法是private、static或者 final的，那么直接调用这个方法**，这就是静态绑定。如果这个方法不是这几个字段修饰的，因为java是有多态的，所以还要查找x的实际类型，然后重复上述过程，如果找到了，就直接调用实际类型里的方法，如果找不到，查找实际类型父类的方法表。

86. 进程通信方式：管道、消息队列、信号量Semaphore、共享内存、信号Signal、socket、文件
87. JWT和Cookie区别
    1. Cookie明文传输，不会对数据做验证，默认不跨域，每次HTTP都会自动带上cookie
    2. Jwt一般存放在客户端本地内存或者cookie中，会加上数据签名保证数据不被篡改，需要手动添加在请求头中
88. get和post
    1. post报文的请求参数在body中，相对安全。
    2. get的参数只能是ASCII码，post可以是任意格式数据
    3. post报文在发送的时候一般是两个数据包（消息头+body）body部分一般在得到服务器100continue后再发。
89. Java接口和抽象类

	抽象类	**接口**
	用来继承	用来实现
其中可否有非抽象类	可以	必须全是抽象类
其中的变量	任意类型	必须是编译时常量
其中静态代码块	可以有	不能有静态代码块
构造方法	可以有	不能有
是否必须实现方法	差异部分交给子类	必须重写接口的所有方法

90. Spring注解
    1. @Autowired默认按类型注入；@Resource默认按名注入；@Qualifier 类型->名
91. 红黑树
    1. 只能红黑
    2. 根一定黑
    3. 红红不连
    4. 根到叶路径上黑数相同
92. Java包装类（Integer）：适应范型要求，表示null值（基本数据类型无法表示null值）
93. 泛型允许在定义类、接口、方法的时候**使用类型参数**，提高代码的重用性和可读性。

    在**编译时**，编译器会做类型擦除，把**泛型变成Object**。泛型在编译的时候不进行类型检查，只在运行时进行类型检查