# 前言：

首先恭喜你能获得此书. 它
覆盖的公司有：阿里巴巴、京东、百度、腾讯、美团、今日头条等一线大厂历年面试题
内容涵盖：Java、MyBatis、ZooKeeper、Dubbo、Elasticsearch、Memcached、 Redis、MySQL、Spring、Spring Boot、Spring Cloud、RabbitMQ、Kafka、等技术栈.

## 目录

# 源码框架专题

## 源码分析专题–MyBatis面试题

### 1、什么是Mybatis？

1、Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身， 不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能， 灵活度高。

2、MyBatis 可以使用 XML 或注解来配置和映射原生信息， 将 POJO 映射成数据库中的记录， 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

3、通过 xml 文件或注解的方式将要执行的各种 statement 配置起来， 并通过java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。（ 从执行 sql 到返回 result 的过程）。

## 2、Mybaits 的优点：

1、基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签， 支持编写动态 SQL 语句， 并可重用。

2、与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；

3、很好的与各种数据库兼容（ 因为 MyBatis 使用 JDBC 来连接数据库，所以只要JDBC 支持的数据库 MyBatis 都支持）。

4、能够与 Spring 很好的集成；

5、提供映射标签， 支持对象与数据库的 ORM 字段关系映射； 提供对象关系映射标签， 支持对象关系组件维护。

## 3、MyBatis 框架的缺点：

1、SQL 语句的编写工作量较大， 尤其当字段多、关联表多时， 对开发人员编写SQL 语句的功底有一定要求。

2、SQL 语句依赖于数据库， 导致数据库移植性差， 不能随意更换数据库。

## 4、MyBatis 框架适用场合：

1、MyBatis 专注于 SQL 本身， 是一个足够灵活的 DAO 层解决方案。

2、对性能的要求很高，或者需求变化较多的项目，如互联网项目， MyBatis 将是不错的选择。

## 5、MyBatis 与Hibernate 有哪些不同？

1、Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句。

2、Mybatis 直接编写原生态 sql， 可以严格控制 sql 执行性能， 灵活度高， 非常适合对关系数据模型要求不高的软件开发， 因为这类软件需求变化频繁， 一但需求变化要求迅速输出成果。但是灵活的前提是 mybatis 无法做到数据库无关性， 如果需要实现支持多种数据库的软件，则需要自定义多套 sql 映射文件，工作量大。

3、Hibernate 对象/关系映射能力强， 数据库无关性好， 对于关系模型要求高的软件，如果用 hibernate 开发可以节省很多代码， 提高效率。

## 6、#{}和${}的区别是什么？

#{}是预编译处理， ${}是字符串替换。

Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的set 方法来赋值；

Mybatis 在处理${}时， 就是把${}替换成变量的值。使用#{}

可以有效的防止 SQL 注入， 提高系统安全性。

## 7、当实体类中的属性名和表中的字段名不一样 ，怎么办 ？

第 1 种： 通过在查询的 sql 语句中定义字段名的别名， 让字段名的别名和实体类的属性名一致。

```
<select id=”selectorder” parametertype=”int” resultetype=”
me.gacl.domain.order”>
        select order_id id, order_no orderno ,order_price price form orders
where order_id=#{id};
</select>
```

第 2 种： 通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder"parameterType="int"
resultMap="orderresultmap">
select * from orders where order_id=#{id}
</select>

<resultMap type=”me.gacl.domain.order” id=”orderresultmap”>
    <!–用 id 属性来映射主键字段–>
    <id property=”id” column=”order_id”>
    <!–用 result 属性来映射非主键字段，property 为实体类属性名，column 为
数据表中的属性–>
    <result property = “orderno” column =”order_no”/>
    <result property=”price” column=”order_price” />
</reslutMap>
```

## 8、 模糊查询like 语句该怎么写?

第 1 种： 在 Java 代码中添加 sql 通配符。

```
string wildcardname = “%smi%”;
list<name> names = mapper.selectlike(wildcardname);
```

```
<select id=”selectlike”>
select * from foo where bar like #{value}
</select>
```

第 2 种： 在 sql 语句中拼接通配符， 会引起 sql 注入

```
string wildcardname = “smi”;
list<name> names = mapper.selectlike(wildcardname);

<select id=”selectlike”>
select * from foo where bar like "%"#{value}"%"
</select>
```

## 9、通常一个Xml 映射文件，都会写一个Dao 接口与之对应，请问，这个Dao 接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载吗？

Dao 接口即 Mapper 接口。接口的全限名，就是映射文件中的 namespace 的值； 接口的方法名， 就是映射文件中 Mapper 的 Statement 的 id 值； 接口方法内的参数， 就是传递给 sql 的参数。

Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值， 可唯一定位一个 MapperStatement。在 Mybatis 中， 每一个<select>、<insert>、<update>、<delete>标签， 都会被解析为一个 MapperStatement 对象。

举例： com.mybatis3.mappers.StudentDao.findStudentById， 可以唯一找到 namespace 为 com.mybatis3.mappers.StudentDao 下面 id 为findStudentById 的 MapperStatement 。

Mapper 接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。Mapper 接口的工作原理是 JDK 动态代理， Mybatis 运行时会使用 JDK 动态代理为 Mapper 接口生成代理对象 proxy， 代理对象会拦截接口方法， 转而执行 MapperStatement 所代表的 sql， 然后将 sql 执行结果返回。

## 10、Mybatis 是如何进行分页的？分页插件的原理是什么？

Mybatis 使用 RowBounds 对象进行分页， 它是针对 ResultSet 结果集执行的内存分页，而非物理分页。可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能， 也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用 Mybatis 提供的插件接口， 实现自定义插件， 在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

## 11、Mybatis 是如何将sql 执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用<resultMap>标签， 逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用 sql 列的别名功能， 将列的别名书写为对象属性名。

有了列名与属性名的映射关系后， Mybatis 通过反射创建对象， 同时使用反射给对象的属性逐一赋值并返回， 那些找不到映射关系的属性， 是无法完成赋值的。

## 12、如何执行批量插入?

首先,创建一个简单的 insert 语句:

```
<insert id=”insertname”>
insert into names (name) values (#{value})
</insert>
```

然后在 java 代码中像下面这样执行批处理插入:

```
list<string>names=newarraylist();
names.add(“fred”);
names.add(“barney”);
names.add(“betty”);
names.add(“wilma”);
// 注意这里 executortype.batch
sqlsession sqlsession =
sqlsessionfactory.opensession(executortype.batch); try {
    namemapper mapper = sqlsession.getmapper(namemapper.class); for
    (string name: names) {
        mapper.insertname(name);
    }
    sqlsession.commit();
} catch (Exception e)
    { e.printStackTrace();
    sqlSession.rollback();
```

```
        throw e;
}
finally {
        sqlsession.close();
}
```

## 13、如何获取自动生成的(主)键值?

insert 方法总是返回一个 int 值， 这个值代表的是插入的行数。

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。

示例：

```
<insert id=“insertname“ usegeneratedkeys=“true“ keyproperty=“id“>
        insert into names (name) values (#{name})
</insert>
name name = new name();
name.setname(“fred“);

int rows = mapper.insertname(name);
// 完成后,id 已经被设置到对象中
system.out.println(“rows inserted = “ + rows);
system.out.println(“generated key value = “ + name.getid());
```

## 14、在 mapper 中如何传递多个参数?

1、第一种：

DAO 层的函数

```
public UserselectUser(String name,String area);
```

对应的 xml,#{0}代表接收的是 dao 层中的第一个参数，#{1}代表 dao 层中第二参数，更多参数一致往后加即可。

```
                    fromuser_user_t      whereuser_name = #{0}
anduser_area=#{1}
</select>
```

2、第二种： 使用 @param 注解:

```
public interface usermapper {
    user selectuser(@param(“username”) string
username,@param(“hashedpassword”) string hashedpassword);
}
```

然后,就可以在 xml 像下面这样使用(推荐封装为一个 map,作为单个参数传递给mapper):

```
<select id=”selectuser” resulttype=”user”> select
            id, username, hashedpassword from
            some_table
            where username = #{username}
            and hashedpassword = #{hashedpassword}
</select>
```

3、第三种： 多个参数封装成 map

```
try {
    //映射文件的命名空间.SQL 片段的 ID，就可以调用对应的映射文件中的
SQL
    //由于我们的参数超过了两个，而方法中只有一个 Object 参数收集，因此
我们使用 Map 集合来装载我们的参数
    Map < String, Object > map = new HashMap();
    map.put("start", start);
    map.put("end", end);
    return sqlSession.selectList("StudentID.pagination", map);
} catch (Exception e)
    { e.printStackTrace();
    sqlSession.rollback();
    throw e;
} finally {
    MybatisUtil.closeSqlSession();
}
```

## 15、Mybatis 动态sql 有什么用？执行原理？有哪些动态sql？

Mybatis 动态 sql 可以在 Xml 映射文件内，以标签的形式编写动态 sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接 sql 的功能。

Mybatis 提供了 9 种动态 sql 标签：trim | where | set | foreach | if | choose | when | otherwise | bind 。

## 16、Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签？

答： <resultMap>、<parameterMap>、<sql>、<include>、

<selectKey>， 加上动态 sql 的 9 个标签， 其中<sql>为 sql 片段标签， 通过 <include>标签引入 sql 片段，<selectKey>为不支持自增的主键生成策略标签。

17、Mybatis 的 Xml 映射文件中， 不同的 Xml 映射文件， id 是否可以重复？

不同的 Xml 映射文件， 如果配置了 namespace， 那么 id 可以重复； 如果没有配置 namespace， 那么 id 不能重复；

原因就是 namespace+id 是作为 Map<String, MapperStatement>的 key 使用的， 如果没有 namespace， 就剩下 id， 那么， id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

## 18、为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

Hibernate 属于全自动 ORM 映射工具， 使用 Hibernate 查询关联对象或者关联集合对象时， 可以根据对象关系模型直接获取， 所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

## 19、 一对一、一对多的关联查询 ？

```
<mapper namespace="com.lcb.mapping.userMapper">
    <!--association      一对一关联查询 -->
```

```
    <select id="getClass" parameterType="int" resultMap="ClassesResultMap">
        select * from class c,teacher t where c.teacher_id=t.t_id and c.c_id=#{id}
    </select>

    <resultMap type="com.lcb.user.Classes" id="ClassesResultMap">
        <!-- 实体类的字段名和数据表的字段名映射 -->
        <id property="id" column="c_id"/>
        <result property="name" column="c_name"/>
        <association property="teacher" javaType="com.lcb.user.Teacher">
            <id property="id" column="t_id"/>
            <result property="name" column="t_name"/>
        </association>
    </resultMap>


    <!--collection      一对多关联查询 -->
    <select id="getClass2"parameterType="int"
resultMap="ClassesResultMap2">
        select * from class c,teacher t,student s where c.teacher_id=t.t_id and
c.c_id=s.class_id and c.c_id=#{id}
    </select>

    <resultMap type="com.lcb.user.Classes" id="ClassesResultMap2">
        <id property="id" column="c_id"/>
        <result property="name" column="c_name"/>
        <association property="teacher" javaType="com.lcb.user.Teacher">
            <id property="id" column="t_id"/>
```

```
            <result property="name" column="t_name"/>
        </association>

        <collection property="student" ofType="com.lcb.user.Student">
            <id property="id" column="s_id"/>
            <result property="name" column="s_name"/>
        </collection>
    </resultMap>
</mapper>
```

## 20、MyBatis 实现一对一有几种方式?具体怎么操作的？

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次, 通过在 resultMap 里面配置 association 节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的 外键 id，去再另外一个表里面查询数据,也是通过 association 配置，但另外一个表的查询通过 select 属性配置。

## 21、MyBatis 实现一对多有几种方式,怎么操作的？

有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面的 collection 节点配置一对多的类就可以完成； 嵌套查询是先查一个表,根据这个表里面的 结果的外键 id,去再另外一个表里面查询数据,也是通过配置 collection,但另外一个表的查询通过 select 节点配置。

## 22、Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

答： Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一， collection 指的就是一对多查询。在 Mybatis 配置文件中， 可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

它的原理是， 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。这就是延迟加载的基本原理。

当然了， 不光是 Mybatis， 几乎所有的包括 Hibernate， 支持延迟加载的原理都是一样的。

## 23、Mybatis 的一级、二级缓存:

1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存， 其存储作用域为 Session， 当 Session flush 或 close 之后， 该 Session 中的所有 Cache 就将清空， 默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储， 不同在于其存储作用域为 Mapper(Namespace)， 并且可自定义存储源， 如 Ehcache。默认不打开二级缓存， 要开启二级缓存， 使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置

<cache/> ；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear 。

## 24、什么是 MyBatis 的接口绑定？有哪些实现方式？

接口绑定，就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定, 我们直接调用接口方法就可以,这样比起原来了 SqlSession 提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定，就是在接口的方法上面加上@Select、@Update 等注解，里面包含 Sql 语句来绑定；另外一种就是通过 xml 里面写 SQL 来绑定, 在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名。当 Sql 语句比较简单时候,用注解绑定, 当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多。

## 25、使用 MyBatis 的mapper 接口调用时有哪些要求？

1、Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同；
2、Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同；
3、Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的resultType 的类型相同；
4、Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

## 26、Mapper 编写有哪几种方式？

第一种： 接口实现类继承 SqlSessionDaoSupport： 使用此种方法需要编写mapper 接口， mapper 接口实现类、mapper.xml 文件。

1、在 sqlMapConfig.xml 中配置 mapper.xml 的位置

```
<mappers>
    <mapper resource="mapper.xml 文件的地址" />
    <mapper resource="mapper.xml 文件的地址" />
</mappers>
```

1、定义 mapper 接口
3、实现类集成 SqlSessionDaoSupport
mapper 方法中可以 this.getSqlSession()进行数据增删改查。4、spring 配置

```
<bean id=" " class="mapper 接口的实现">
    <property name="sqlSessionFactory"
ref="sqlSessionFactory"></property>
</bean>
```

第二种： 使用 org.mybatis.spring.mapper.MapperFactoryBean：

1、在 sqlMapConfig.xml 中配置 mapper.xml 的位置， 如果 mapper.xml 和mappre 接口的名称相同且在同一个目录， 这里可以不用配置

```
<mappers>
    <mapper resource="mapper.xml 文件的地址" />
    <mapper resource="mapper.xml 文件的地址" />
</mappers>
```

2、定义 mapper 接口：

1、mapper.xml 中的 namespace 为 mapper 接口的地址
2、mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致
3、Spring 中定义

```
<bean id="" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface"          value="mapper 接口地址" />
    <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

第三种： 使用 mapper 扫描器： 1、

mapper.xml 文件编写：

mapper.xml 中的 namespace 为 mapper 接口的地址；
mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致； 如果将 mapper.xml 和 mapper 接口的名称保持一致则不用在 sqlMapConfig.xml 中进行配置。

2、定义 mapper 接口：

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致， 且放在同一个目录3、配置 mapper 扫描器：

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="mapper 接口包地址
"></property>
    <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactory"/>
</bean>
```

4、使用扫描器后从 spring 容器中获取 mapper 的实现对象。

## 27、简述 Mybatis 的插件运行原理，以及如何编写一个插件。

答： Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件， Mybatis 使用 JDK 的动态代理， 为需要拦截的接口生成代理对象以实现接口方法拦截功能， 每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke() 方法， 当然， 只会拦截那些你指定需要拦截的方法。

编写插件： 实现 Mybatis 的 Interceptor 接口并复写 intercept()方法， 然后在给插件编写注解， 指定要拦截哪一个接口的哪些方法即可， 记住， 别忘了在配置文件中配置你编写的插件。

# 源码分析专题-Spring面试题

## 1、什么是spring?

Spring 是个 java 企业级应用的开源开发框架。Spring 主要用来开发 Java 应用， 但是有些扩展是针对构建 J2EE 平台的 web 应用。Spring 框架目标是简化 Java 企业级应用开发， 并通过 POJO 为基础的编程模型促进良好的编程习惯。

## 2、使用Spring 框架的好处是什么？

- 轻量：Spring 是轻量的，基本的版本大约 2MB。
- 控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- 面向切面的编程(AOP)：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- 容器：Spring 包含并管理应用中对象的生命周期和配置。
- MVC 框架：Spring 的WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
- 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- 异常处理：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO 抛出的）转化为一致的 unchecked 异常。

## 3、Spring 由哪些模块组成?

以下是 Spring 框架的基本模块：

- Core module
- Bean module
- Context module
- Expression Language module
- JDBC module
- ORM module
- OXM module
- Java Messaging Service(JMS) module
- Transaction module
- Web module
- Web-Servlet module
- Web-Struts module
- Web-Portlet module

## 4、核心容器（应用上下文) 模块。

这是基本的 Spring 模块， 提供 spring 框架的基础功能， BeanFactory 是 任何以 spring 为基础的应用的核心。Spring 框架建立在此模块之上，它使 Spring 成为一个容器。

## 5、BeanFactory – BeanFactory 实现举例。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从正真的应用代码中分离。

最常用的 BeanFactory 实现是 XmlBeanFactory 类。

## 6、XMLBeanFactory

最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory， 它根据 XML 文件中的定义加载 beans。该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

## 7、解释AOP 模块

AOP 模块用于发给我们的 Spring 应用做面向切面的开发， 很多支持由 AOP 联盟提供，这样就确保了 Spring 和其他 AOP 框架的共通性。这个模块将元数据编程引入 Spring。

## 8、解释JDBC 抽象和DAO 模块。

通过使用 JDBC 抽象和 DAO 模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题， 它在各种不同的数据库的错误信息之上， 提供了一个统一的异常访问层。它还利用 Spring 的 AOP 模块给 Spring 应用中的对象提供事务管理服务。

## 9、解释对象/关系映射集成模块。

Spring 通过提供 ORM 模块，支持我们在直接 JDBC 之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的 ORM 框架，如 Hiberate,JDO 和 iBATIS SQL Maps。Spring 的事务管理同样支持以上所有 ORM 框架及 JDBC。

## 10、解释 WEB 模块。

Spring 的 WEB 模块是构建在 application context 模块基础之上， 提供一个适合 web 应用的上下文。这个模块也包括支持多种面向 web 的任务，如透明地处理多个文件上传请求和程序级请求参数的绑定到你的业务对象。它也有对 Jakarta Struts 的支持。

## 12、Spring 配置文件

Spring 配置文件是个 XML 文件，这个文件包含了类信息，描述了如何配置它们， 以及如何相互调用。

## 13、什么是 Spring IOC 容器？

Spring IOC 负责创建对象，管理对象（ 通过依赖注入（ DI），装配对象，配置对象， 并且管理这些对象的整个生命周期。

## 14、IOC 的优点是什么？

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

## 15、ApplicationContext 通常的实现是什么?

- FileSystemXmlApplicationContext ： 此容器从一个 XML 文件中加载 beans 的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

- ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载 beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。
- WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的所有 bean。

## 16、Bean 工厂和 Application contexts 有什么区别？

Application contexts 提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的 bean 发布事件。另外，在容器或容器内的对象上执行的那些不得不由 bean 工厂以程序化方式处理的操作，可以在 Application contexts 中以声明的方式处理。Application contexts 实现了 MessageSource 接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。

## 17、一个 Spring 的应用看起来象什么？

- 一个定义了一些功能的接口。
- 这实现包括属性，它的 Setter ， getter 方法和函数等。
- Spring AOP。
- Spring 的 XML 配置文件。
- 使用以上功能的客户端程序。

## 依赖注入

## 18、什么是 Spring 的依赖注入？

依赖注入，是 IOC 的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC 容器）负责把他们组装起来。

## 19、有哪些不同类型的 IOC（依赖注入）方式？

- 构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
- Setter 方法注入：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂 方法实例化 bean 之后，调用该 bean 的 setter 方法，即实现了基于 setter 的依赖注入。

## 20、哪种依赖注入方式你建议使用，构造器注入，还是 Setter 方法注入？

你两种依赖方式都可以使用，构造器注入和 Setter 方法注入。最好的解决方案是用构造器参数实现强制依赖，setter 方法实现可选依赖。

## Spring Beans

## 21.什么是 Spring beans?

Spring beans 是那些形成 Spring 应用的主干的 java 对象。它们被 Spring IOC 容器初始化， 装配， 和管理。这些 beans 通过容器中配置的元数据创建。比如， 以 XML 文件中 的形式定义。

Spring 框架定义的 beans 都是单件 beans。在 bean tag 中有个属性“singleton”，如果它被赋为 TRUE，bean 就是单件，否则就是一个 prototype bean。默认是 TRUE， 所以所有在 Spring 框架中的 beans 缺省都是单件。

## 22、一个 Spring Bean 定义 包含什么？

一个 Spring Bean 的定义包含容器必知的所有配置元数据， 包括如何创建一个bean， 它的生命周期详情及它的依赖。

## 23、如何给 Spring 容器提供配置元数据?

这里有三种重要的方法给 Spring 容器提供配置元数据。XML 配

置文件。

基于注解的配置。基

于 java 的配置。

## 24、你怎样定义类的作用域?

当定义一个 在 Spring 里， 我们还能给这个 bean 声明一个作用域。它可以通过bean 定义中的 scope 属性来定义。如，当 Spring 要在需要的时候每次生产一个新的 bean 实例，bean 的 scope 属性被指定为 prototype。另一方面，一个 bean

每次使用的时候必须返回同一个实例， 这个 bean 的 scope 属性 必须设为 singleton。

## 25、解释 Spring 支持的几种bean 的作用域。

Spring 框架支持以下五种 bean 的作用域：

- singleton : bean 在每个 Spring ioc 容器中只有一个实例。
- prototype：一个 bean 的定义可以有多个实例。
- request：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。
- session：在一个 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。
- global-session：在一个全局的 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

缺省的 Spring bean 的作用域是 Singleton.

## 26、Spring 框架中的单例 bean 是线程安全的吗?

不， Spring 框架中的单例 bean 不是线程安全的。

## 27、解释 Spring 框架中bean 的生命周期。

- Spring 容器 从 XML 文件中读取 bean 的定义，并实例化 bean。
- Spring 根据 bean 的定义填充所有的属性。

- 如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到setBeanName 方法。
- 如果 Bean 实现了 BeanFactoryAware 接口， Spring 传递beanfactory 给 setBeanFactory 方 法 。
- 如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在postProcesserBeforeInitialization()方法内调用它们。
- 如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法， 如果 bean 声明了初始化方法，调用此初始化方法。
- 如果有 BeanPostProcessors 和 bean 关联，这些 bean 的postProcessAfterInitialization() 方法将被调用。
- 如果 bean 实现了 DisposableBean，它将调用 destroy()方法。

## 28、哪些是重要的 bean 生命周期方法？你能重载它们吗？

有两个重要的 bean 生命周期方法， 第一个是 setup ， 它是在容器加载 bean 的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（ init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（ @PostConstruct 和@PreDestroy）。

## 29、什么是 Spring 的内部bean？

当一个 bean 仅被用作另一个 bean 的属性时， 它能被声明为一个内部 bean， 为了定义 inner bean， 在 Spring 的 基于 XML 的 配置元数据中， 可以在 或 元素内使用 元素， 内部 bean 通常是匿名的， 它们的 Scope 一般是 prototype。

## 30、在 Spring 中如何注入一个java 集合？

Spring 提供以下几种集合的配置元素：

- 类型用于注入一列值，允许有相同的值。
- 类型用于注入一组值，不允许有相同的值。
- 类型用于注入一组键值对，键和值都可以为任意类型。
- 类型用于注入一组键值对，键和值都只能为 String 类型。

## 31、什么是 bean 装配?

装配，或 bean 装配是指在 Spring 容器中把 bean 组装到一起，前提是容器需要知道 bean 的依赖关系， 如何通过依赖注入来把它们装配到一起。

## 32、什么是 bean 的自动装配？

Spring 容器能够自动装配相互合作的 bean， 这意味着容器不需要和配置， 能通过 Bean 工厂自动处理 bean 之间的协作。

## 33、解释不同方式的自动装配 。

有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

- no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。

- byName：通过参数名 自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
- byType:：通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byType，之后容器试图匹配、装配和该 bean 的属性具有相同类型的 bean。如果有多个 bean 符合条件，则抛出错误。
- constructor：这个方式类似于 byType， 但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
- autodetect：首先尝试使用 constructor 来自动装配，如果无法工作， 则使用 byType 方式。

## 34.自动装配有哪些局限性？

自动装配的局限性是：

- 重写：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。
- 基本数据类型：你不能自动装配简单的属性，如基本数据类型，String 字符串，和类。
- 模糊特性：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

## 35、你可以在 Spring 中注入一个null 和一个空字符串吗？

可以。

## Spring 注解

## 36、什么是基于 Java 的Spring 注解配置? 给一些注解的例子.

基于 Java 的配置， 允许你在少量的 Java 注解的帮助下， 进行你的大部分 Spring 配置而非通过 XML 文件。

以@Configuration 注解为例， 它用来标记类可以当做一个 bean 的定义， 被 Spring IOC 容器使用。另一个例子是@Bean 注解， 它表示此方法将要返回一个对象， 作为一个 bean 注册进 Spring 应用上下文。

## 37、什么是基于注解的容器配置?

相对于 XML 文件，注解型的配置依赖于通过字节码元数据装配组件，而非尖括号的声明。

开发者通过在相应的类， 方法或属性上使用注解的方式， 直接组件类中进行配置， 而不是使用 xml 表述 bean 的装配关系。

## 38、怎样开启注解装配？

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在 Spring 配置文件中配置 context:annotation-config/元素。

## 39、@Required 注解

这个注解表明 bean 的属性必须在配置的时候设置，通过一个 bean 定义的显式的属性值或通过自动装配， 若 @Required 注解的 bean 属性未被设置， 容器将抛出 BeanInitializationException。

## 40、@Autowired 注解

@Autowired 注解提供了更细粒度的控制， 包括在何处以及如何完成自动装配。它的用法和@Required 一样，修饰 setter 方法、构造器、属性或者具有任意名称和/或多个参数的 PN 方法。

## 41、@Qualifier 注解

当有多个相同类型的 bean 却只有一个需要自动装配时， 将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆， 指定需要装配的确切的 bean。

# Spring 数据访问

## 42.在 Spring 框架中如何更有效地使用JDBC?

使用 SpringJDBC 框架， 资源管理和错误处理的代价都会被减轻。所以开发者只需写 statements 和 queries 从数据存取数据，JDBC 也可以在 Spring 框架提供的模板类的帮助下更有效地被使用， 这个模板叫 JdbcTemplate （ 例子见这里here）

## 43、JdbcTemplate

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句， 提供自定义的数据错误处理。

## 44、Spring 对 DAO 的支持

Spring 对数据访问对象（DAO）的支持旨在简化它和数据访问技术如 JDBC，Hibernate or JDO 结合使用。这使我们可以方便切换持久层。编码时也不用担心会捕获每种技术特有的异常。

## 45、使用 Spring 通过什么方式访问Hibernate?

在 Spring 中有两种方式访问 Hibernate：

- 控制反转 Hibernate Template 和 Callback。
- 继承 HibernateDAOSupport 提供一个 AOP 拦截器。

## 46、Spring 支持的 ORM

Spring 支持以下 ORM：

- Hibernate
- iBatis
- JPA (Java Persistence API)
- TopLink
- JDO (Java Data Objects)
- OJB

## 47.如何通过HibernateDaoSupport 将Spring 和Hibernate 结合起来？

用 Spring 的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

- 配置 the Hibernate SessionFactory。
- 继承 HibernateDaoSupport 实现一个 DAO。
- 在 AOP 支持的事务中装配。

## 48、Spring 支持的事务管理类型

Spring 支持两种类型的事务管理：

- 编程式事务管理：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
- 声明式事务管理：这意味着你可以将业务代码和事务管理分离，你只需用注解和 XML 配置来管理事务。

## 49、Spring 框架的事务管理有哪些优点？

- 它为不同的事务 API 如 JTA，JDBC，Hibernate，JPA 和 JDO，提供一个不变的编程模式。
- 它为编程式事务管理提供了一套简单的 API 而不是一些复杂的事务 API 如

- 它支持声明式事务管理。
- 它和 Spring 各种数据访问抽象层很好得集成。

## 50、你更倾向用那种事务管理类型？

大多数 Spring 框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

## Spring 面向切面编程（AOP） 51、

## 解释 AOP

面向切面的编程，或 AOP，是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。

## 52、Aspect 切面

AOP 核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组 API 提供横切功能。比如，一个日志模块可以被称作日志的 AOP 切面。根据需求的不同，一个应用程序可以有若干切面。在 Spring AOP 中，切面通过带有@Aspect 注解的类实现。

## 52、在 Spring AOP 中，关注点和横切关注的区别是什么？

关注点是应用中一个模块的行为， 一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点， 此关注点是整个应用都会使用的功能， 并影响整个应用， 比如日志， 安全和数据传输， 几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

## 54、连接点

连接点代表一个应用程序的某个位置， 在这个位置我们可以插入一个 AOP 切面， 它实际上是个应用程序执行 Spring AOP 的位置。

## 55、通知

通知是个在方法执行前或执行后要做的动作， 实际上是程序执行时要通过SpringAOP 框架触发的代码段。

Spring 切面可以应用五种类型的通知：

- before：前置通知，在一个方法执行前被调用。
- after: 在方法执行之后调用的通知，无论方法执行是否成功。
- after-returning: 仅当方法成功完成后执行的通知。
- after-throwing: 在方法抛出异常退出时执行的通知。
- around: 在方法执行之前和之后调用的通知。

## 56、切点

切入点是一个或一组连接点， 通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

## 57、什么是引入?

引入允许我们在已存在的类中增加新的方法和属性。

## 58、什么是目标对象?

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（ advised） 对象。

## 59、什么是代理?

代理是通知目标对象后创建的对象。从客户端的角度看， 代理对象和目标对象是一样的。

## 60、有几种不同类型的自动代理？

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

## 61、什么是织入。什么是织入应用的不同点？

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。 织入可以在

编译时， 加载时， 或运行时完成。

## 62、解释基于 XML Schema 方式的切面实现。

在这种情况下， 切面由常规类以及基于 XML 的配置实现。

## 63、解释基于注解的切面实现

在这种情况下(基于@AspectJ 的实现)，涉及到的切面声明的风格与带有 java5 标注的普通 java 类一致。

## Spring 的MVC

## 64、什么是 Spring 的MVC 框架？

Spring 配备构建 Web 应用的全功能 MVC 框架。Spring 可以很便捷地和其他MVC 框架集成，如 Struts，Spring 的 MVC 框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

## 65、DispatcherServlet

Spring 的 MVC 框架是围绕 DispatcherServlet 来设计的，它用来处理所有的 HTTP 请求和响应。

## 66、WebApplicationContext

WebApplicationContext 继承了 ApplicationContext 并增加了一些 WEB 应用必备的特有功能， 它不同于一般的 ApplicationContext ， 因为它能处理主题， 并找到被关联的 servlet。

## 67、什么是 Spring MVC 框架的控制器？

控制器提供一个访问应用程序的行为， 此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring 用一个非常抽象的方式实现了一个控制层， 允许用户创建多种用途的控制器。

## 68、@Controller 注解

该注解表明该类扮演控制器的角色， Spring 不需要你继承任何其他控制器基类或引用 Servlet API。

## 69、@RequestMapping 注解

该注解是用来映射一个 URL 到一个类或一个特定的方处理法上。

# 分布式专题

# 分布式专题-ZooKeeper 面试题

## 1. ZooKeeper 面试题？

ZooKeeper 是一个开放源码的分布式协调服务， 它是集群的管理者， 监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终， 将简单易用的接口和性能高效、功能稳定的系统提供给用户。

分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

Zookeeper 保证了如下分布式一致性特性：

1、顺序一致性
2、原子性
3、单一视图
4、可靠性
5、实时性（ 最终一致性）

客户端的读请求可以被集群中的任意一台机器处理， 如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此， 随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。

有序性是 zookeeper 中非常重要的一个特性， 所有的更新都是全局有序的， 每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（ Zookeeper Transaction Id）。而读请求只会相对于更新有序， 也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。

## 2. ZooKeeper 提供了什么？

1、文件系统
2、通知机制

## 3. Zookeeper 文件系统

Zookeeper 提供一个多层级的节点命名空间（ 节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据， 而文件系统中只有文件节点可以存放数据而目录节点不行。

Zookeeper 为了保证高吞吐和低延迟， 在内存中维护了这个树状的目录结构， 这种特性使得 Zookeeper 不能用于存放大量的数据， 每个节点的存放数据上限为1M。

## 4. ZAB 协议？

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

## 5. 四种类型的数据节点 Znode

1、PERSISTENT-持久节点
除非手动删除，否则节点一直存在于 Zookeeper 上

2、EPHEMERAL-临时节点
临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

3、PERSISTENT_SEQUENTIAL-持久顺序节点
基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

4、EPHEMERAL_SEQUENTIAL-临时顺序节点
基本特性同临时节点， 增加了顺序属性， 节点名后边会追加一个由父节点维护的自增整型数字。

# 6. Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听， 当服务端的一些指定事件触发了这个 Watcher， 服务端会向指定客户端发送一个事件通知来实现分布式的通知功能， 然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制：

1、客户端注册 watcher
2、服务端处理 watcher
3、客户端回调 watcher

Watcher 特性总结：

1、一次性
无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力， 不然对于更新非常频繁的节点， 服务端会不断的向客户端发送事件通知， 无论对于网络还是服务端的压力都非常大。

2、客户端串行执行
客户端 Watcher 回调的过程是一个串行同步的过程。3、轻

量

3.1 、Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
3.2 、客户端向服务端注册 Watcher 的时候， 并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

4、 watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee， 即客户端监听事件后， 才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性， 而无法保证强一致性。

5、注册 watcher getData、exists、getChildren 6、触发

watcher create、delete、setData

7、当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候， 是无法接收到 watch 的。而当 client 重新连接时， 如果需要的话， 所有先前注册过的 watch， 都会被重新注册。通常这是完全透明的。只有在一个特殊情况下， watch 可能会丢失： 对于一个未创建的 znode 的 exist watch， 如果在客户端断开连接期间被创建了， 并且随后在客户端连接上之前又删除了， 这种情况下， 这个 watch 事件可能会被丢失。

# 7. 客户端注册Watcher 实现

1、调用 getData()/getChildren()/exist()三个 API， 传入 Watcher 对象2、标记请求 request， 封装 Watcher 到 WatchRegistration
3、封装成 Packet 对象， 发服务端发送 request
4、收到服务端响应后， 将 Watcher 注册到 ZKWatcherManager 中进行管理5、请求返回， 完成注册。

# 8. 服务端处理Watcher 实现

1、服务端接收 Watcher 并存储
接收到客户端请求， 处理请求判断是否需要注册 Watcher， 需要的话将数据节点的节点路径和 ServerCnxn（ ServerCnxn 代表一个客户端和服务端的连接， 实现了 Watcher 的 process 接口， 此时可以看成一个 Watcher 对象） 存储在

WatcherManager 的 WatchTable 和 watch2Paths 中去。

2、Watcher 触发
以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

2.1 封装 WatchedEvent
将通知状态（ SyncConnected）、事件类型（ NodeDataChanged） 以及节点路径封装成一个 WatchedEvent 对象

2.2 查询 Watcher
从 WatchTable 中根据节点路径查找 Watcher

2.3 没找到； 说明没有客户端在该数据节点上注册过 Watcher

2.4 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的， 触发一次就失效了）

3、调用 process 方法来触发 Watcher
这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。

# 9. 客户端回调Watcher

客户端 SendThread 线程接收事件通知， 交由 EventThread 线程回调 Watcher。客户端的 Watcher 机制同样是一次性的， 一旦被触发后， 该 Watcher 就失效了。

# 10. ACL 权限控制机制

UGO（ User/Group/Others）

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

ACL（ Access Control List） 访问控制列表包括三

个方面：

权限模式（ Scheme）

1、IP： 从 IP 地址粒度进行权限控制
2、Digest： 最常用， 用类似于 username:password 的权限标识来进行权限配置， 便于区分不同应用来进行权限控制
3、World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识“world:anyone”
4、Super： 超级用户

授权对象

授权对象指的是权限赋予的用户或一个指定实体， 例如 IP 地址或是机器灯。

权 限 Permission

1、CREATE： 数据节点创建权限， 允许授权对象在该 Znode 下创建子节点

2、DELETE： 子节点删除权限， 允许授权对象删除该数据节点的子节点
3、READ： 数据节点的读取权限， 允许授权对象访问该数据节点并读取其数据内容或子节点列表等

4、WRITE： 数据节点更新权限， 允许授权对象对该数据节点进行更新操作
5、ADMIN： 数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作

# 11. Chroot 特性

3.2.0 版本后，添加了 Chroot 特性，该特性允许每个客户端为自己设置一个命名空间。如果一个客户端设置了 Chroot， 那么该客户端对服务器的任何操作， 都将会被限制在其自己的命名空间下。

通过设置 Chroot， 能够将一个客户端应用于 Zookeeper 服务端的一颗子树相对应，在那些多个应用公用一个 Zookeeper 进群的场景下，对实现不同应用间的相互隔离非常有帮助。

# 12. 会话管理

分桶策略：将类似的会话放在同一区块中进行管理，以便于 Zookeeper 对会话进行不同区块的隔离处理以及同一区块的统一处理。

分配原则：每个会话的“下次超时时间点”（ExpirationTime）

计算公式：

```
ExpirationTime_=currentTime+sessionTimeout
```

ExpirationTime = (ExpirationTime_ / ExpirationInrerval + 1) * ExpirationInterval , ExpirationInterval 是指 Zookeeper 会话超时检查时间间隔，默认 tickTime

## 13. 服务器角色

Leader

1、事务请求的唯一调度和处理者， 保证集群事务处理的顺序性
2、集群内部各服务的调度者

Follower

1、处理客户端的非事务请求， 转发事务请求给 Leader 服务器2、参
与事务请求 Proposal 的投票
3、参与 Leader 选举投票

Observer

1、3.0 版本以后引入的一个服务器角色， 在不影响集群事务处理能力的基础上提升集群的非事务处理能力
2、处理客户端的非事务请求， 转发事务请求给 Leader 服务器
3、不参与任何形式的投票

## 14. Zookeeper 下 Server 工作状态

服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。

1、LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader， 因此需要进入 Leader 选举状态。

2、FOLLOWING： 跟随者状态。表明当前服务器角色是 Follower。

3、LEADING： 领导者状态。表明当前服务器角色是 Leader。

4、OBSERVING： 观察者状态。表明当前服务器角色是 Observer。

# 15. 数据同步

整个集群完成 Leader 选举之后， Learner（ Follower 和 Observer 的统称） 回向Leader 服务器进行注册。当 Learner 服务器想 Leader 服务器完成注册后， 进入数据同步环节。

数据同步流程：（ 均以消息传递的方式进行）

Learner 向 Learder 注 册

数据同步

同步确认

Zookeeper 的数据同步通常分为四类：

1、直接差异化同步（ DIFF 同步）
2、先回滚再差异化同步（ TRUNC+DIFF 同步）
3、仅回滚同步（ TRUNC 同步）
4、全量同步（ SNAP 同步）

在进行数据同步前， Leader 服务器会完成数据同步初始化：

peerLastZxid：

- 从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid（该 Learner 服务器最后处理的 ZXID）

minCommittedLog：

- Leader 服务器 Proposal 缓存队列 committedLog 中最小 ZXID

maxCommittedLog：

- Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID

直接差异化同步（ DIFF 同步）

- 场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog 之间

先回滚再差异化同步（ TRUNC+DIFF 同步）

- 场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚--回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID

仅回滚同步（ TRUNC 同步）

- 场景：peerLastZxid 大于 maxCommittedLog

全量同步（SNAP 同步）

- 场景一：peerLastZxid 小于 minCommittedLog
- 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid

## 16. zookeeper 是如何保证事务的顺序一致性的？

zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期; 纪元; 世; 新时代）用来标识 leader 周期，如果有新的 leader 产生出来，epoch会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

## 17. 分布式集群中为什么会有Master？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行leader 选举。

## 18. zk 节点宕机如何处理？

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时， 其他节点会继续提供服务。
如果是一个 Follower 宕机， 还有 2 台服务器提供访问， 因为 Zookeeper 上的数据是有多个副本的， 数据并不会丢失；
如果是一个 Leader 宕机， Zookeeper 会选举出新的 Leader。
ZK 集群的机制是只要超过半数的节点正常， 集群就能正常提供服务。只有在 ZK 节点挂得太多， 只剩一半或不到一半节点能工作， 集群才失效。

所以
3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)
2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

## 19. zookeeper 负载均衡和nginx 负载均衡区别

zk 的负载均衡是可以调控， nginx 只是能调权重， 其他需要可控的都需要自己写插件；
但是 nginx 的吞吐量比 zk 大很多， 应该说按业务选择用哪种方式。

## 20. Zookeeper 有哪几种几种部署模式？

部署模式： 单机模式、伪集群模式、集群模式。

## 21. 集群最少要几台机器，集群规则是怎样的?

集群规则为 2N+1 台， N>0， 即 3 台。

## 22. 集群支持动态添加机器吗？

其实就是水平扩容了， Zookeeper 在这方面不太好。两种方式：

全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。

逐个重启： 在过半存活即可用的原则下， 一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。

3.5 版本开始支持动态扩容。

## 23. Zookeeper 对节点的watch 监听通知是永久的吗？为什么不是永久的?

不是。官方声明： 一个 Watch 事件是一个一次性的触发器， 当被设置了 Watch 的数据发生了改变的时候， 则服务器将这个改变发送给设置了 Watch 的客户端， 以便通知它们。

为什么不是永久的， 举个例子， 如果服务端变动频繁， 而监听的客户端很多情况下，每次变动都要通知到所有的客户端， 给网络和服务器造成很大压力。
一般是客户端执行 getData(“/节点 A”,true)， 如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件， 但是在之后节点 A 又发生了变更， 而客户端又没有设置 watch 事件， 就不再给客户端发送。
在实际应用中， 很多情况下， 我们的客户端不需要知道服务端的每一次变动， 我只要最新的数据即可。

## 24. Zookeeper 的java 客户端都有哪些？

java 客户端： zk 自带的 zkclient 及 Apache 开源的 Curator。

## 25. chubby 是什么，和 zookeeper 比你怎么看？

chubby 是 google 的， 完全实现 paxos 算法， 不开源。zookeeper 是 chubby 的开源实现， 使用 zab 协议， paxos 算法的变种。

## 26. 说几个zookeeper 常用的命令。

常用命令： ls get set create delete 等。

## 27. ZAB 和 Paxos 算法的联系与区别？

相同点：
1、两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行
2、Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交
3、ZAB 协议中， 每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot

不同点：

ZAB 用来构建高可用的分布式数据主备系统（ Zookeeper）， Paxos 是用来构建分布式一致性状态机系统。

## 28. Zookeeper 的典型应用场景

Zookeeper 是一个典型的发布/订阅模式的分布式数据管理与协调框架，开发人员可以使用它来进行分布式数据的发布和订阅。

通过对 Zookeeper 中丰富的数据节点进行交叉使用， 配合 Watcher 事件通知机制， 可以非常方便的构建一系列分布式应用中年都会涉及的核心功能， 如：

1、数据发布/订阅
2、 负载均衡
3、 命名服务
4、分布式协调/通知
5、集群管理
6、 Master 选举
7、分布式锁
8、分布式队列

1. 数据发布/订阅

介绍

数据发布/订阅系统， 即所谓的配置中心， 顾名思义就是发布者发布数据供订阅者进行数据订阅。

目的

动态获取数据（ 配置信息）
实现数据（ 配置信息） 的集中式管理和数据的动态更新

设计模式

Push 模 式
Pull 模 式

数据（ 配置信息） 特性

1、数据量通常比较小
2、数据内容在运行时会发生动态更新
3、集群中各机器共享， 配置一致

如： 机器列表信息、运行时开关配置、数据库配置信息等

基于 Zookeeper 的实现方式

- 数据存储：将数据（配置信息）存储到 Zookeeper 上的一个数据节点
- 数据获取：应用在启动初始化节点从 Zookeeper 数据节点读取数据，并在该节点上注册一个数据变更 Watcher
- 数据变更：当变更数据时，更新 Zookeeper 对应节点数据，Zookeeper 会将数据变更通知发到各客户端，客户端接到通知后重新读取变更后的数据即可。

2. 负载均衡zk

的命名服务

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，这个路径就可以作为一个名字， 指向集群中的集群， 提供的服务的地址， 或者一个远程的对象等等。

分布式通知和协调

对于系统调度来说： 操作人员发送通知实际是通过控制台改变某个节点的状态， 然后 zk 将这些变化发送给注册了这个节点的 watcher 的所有客户端。

对于执行情况汇报： 每个工作进程都在某个目录下创建一个临时节点。并携带工作的进度数据， 这样汇总的进程可以监控目录子节点的变化获得工作进度的实时的全局情况。

zk 的命名服务（ 文件系统）

命名服务是指通过指定的名字来获取资源或者服务的地址，利用 zk 创建一个全局的路径，即是唯一的路径， 这个路径就可以作为一个名字， 指向集群中的集群， 提供的服务的地址， 或者一个远程的对象等等。

zk 的配置管理（ 文件系统、通知机制）

程序分布式的部署在不同的机器上，将程序的配置信息放在 zk 的 znode 下，当有配置发生改变时，也就是 znode 发生变化时，可以通过改变 zk 中某个目录节点的内容， 利用 watcher 通知给各个客户端， 从而更改配置。

Zookeeper 集群管理（ 文件系统、通知机制）

所谓集群管理无在乎两点： 是否有机器退出和加入、选举 master。
对于第一点， 所有机器约定在父目录下创建临时目录节点， 然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 zookeeper 的连接断开，其所创建的临时目录节点被删除， 所有其他机器都收到通知： 某个兄弟目录被删除， 于是， 所有人都知道：它上船了。
新机器加入也是类似，所有机器收到通知：新兄弟目录加入，highcount 又有了， 对于第二点， 我们稍微改变一下， 所有机器创建临时顺序编号目录节点， 每次选取编号最小的机器作为 master 就好。

Zookeeper 分布式锁（ 文件系统、通知机制）

有了 zookeeper 的一致性文件系统， 锁的问题变得容易。锁服务可以分为两类， 一个是保持独占， 另一个是控制时序。

对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。
对于第二类， /distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点， 和选 master 一样， 编号最小的获得锁， 用完删除， 依次方便。

Zookeeper 队列管理（ 文件系统、通知机制）

两种类型的队列：

1、同步队列， 当一个队列的成员都聚齐时， 这个队列才可用， 否则一直等待所有成员到达。
2、队列按照 FIFO 方式进行入队和出队操作。

第一类， 在约定目录下创建临时目录节点， 监听节点数目是否是我们要求的数目。

第二类， 和分布式锁服务中的控制时序场景基本原理一致， 入列有编号， 出列按编号。在特定的目录下创建 PERSISTENT_SEQUENTIAL 节点， 创建成功时

Watcher 通知等待的队列， 队列删除序列号最小的节点用以消费。此场景下

Zookeeper 的 znode 用于消息存储，znode 存储的数据就是消息队列中的消息内容，SEQUENTIAL 序列号就是消息的编号， 按序取出即可。由于创建的节点是持久化的， 所以不必担心队列消息的丢失问题。

# 分布式专题-Dubbo　面试题

## 1、为什么要用Dubbo？

随着服务化的进一步发展， 服务越来越多， 服务之间的调用和依赖关系也越来越复杂，诞生了面向服务的架构体系(SOA)，

也因此衍生出了一系列相应的技术， 如对服务提供、服务调用、连接处理、通信协议、序列化方式、服务发现、服务路由、日志输出等行为进行封装的服务框架。

就这样为分布式系统的服务治理框架就出现了， Dubbo 也就这样产生了。

## 2、Dubbo 的整体架构设计有哪些分层?

接口服务层（ Service）：该层与业务逻辑相关，根据 provider 和 consumer 的业务设计对应的接口和实现

配置层（ Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心

服务代理层（ Proxy）：服务接口透明代理，生成服务的客户端 Stub 和服务端的 Skeleton， 以 ServiceProxy 为中心， 扩展接口为 ProxyFactory

服务注册层（ Registry）： 封装服务地址的注册和发现， 以服务 URL 为中心， 扩展接口为 RegistryFactory、Registry、RegistryService

路由层（ Cluster）： 封装多个提供者的路由和负载均衡， 并桥接注册中心， 以Invoker 为中心， 扩展接口为 Cluster、Directory、Router 和 LoadBlancce

监控层（ Monitor）： RPC 调用次数和调用时间监控， 以 Statistics 为中心， 扩展接口为 MonitorFactory、Monitor 和 MonitorService

远程调用层（ Protocal）：封装 RPC 调用，以 Invocation 和 Result 为中心， 扩展接口为 Protocal、Invoker 和 Exporter

信息交换层（ Exchange）： 封装请求响应模式， 同步转异步。以 Request 和Response 为中心， 扩展接口为 Exchanger、ExchangeChannel、 ExchangeClient 和 ExchangeServer

网络传输层（ Transport）：抽象 mina 和 netty 为统一接口，以 Message 为中心， 扩展接口为 Channel、Transporter、Client、Server 和 Codec

数据序列化层（ Serialize）： 可复用的一些工具， 扩展接口为 Serialization、 ObjectInput、ObjectOutput 和 ThreadPool

## 3、默认使用的是什么通信框架，还有别的选择吗?

默认也推荐使用 netty 框架， 还有 mina。

## 4、服务调用是阻塞的吗？

默认是阻塞的， 可以异步调用， 没有返回值的可以这么做。

Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务， 相对多线程开销较小， 异步调用会返回一个 Future 对象。

## 5、一般使用什么注册中心？还有别的选择吗？

推荐使用 Zookeeper 作为注册中心， 还有 Redis、Multicast、Simple 注册中心， 但不推荐。

## 6、默认使用什么序列化框架，你知道的还有哪些？

推荐使用 Hessian 序列化， 还有 Duddo、FastJson、Java 自带序列化。

## 7、服务提供者能实现失效踢出是什么原理？

服务失效踢出基于 zookeeper 的临时节点原理。

## 8、服务上线怎么不影响旧版本？

采用多版本开发， 不影响旧版本。

## 9、如何解决服务调用链过长的问题？

可以结合 zipkin 实现分布式服务追踪。

## 10、说说核心的配置有哪些？

| 配置 | 配置说明 |
| --- | --- |
| dubbo:service | 服务配置 |
| dubbo:reference | 引用配置 |
| dubbo:protocol | 协议配置 |
| dubbo:applicatio n | 应用配置 |

| | |
|---|---|
| dubbo:module | 模块配置 |
| dubbo:registry | 注册中心配置 |
| dubbo:monitor | 监控中心配置 |
| dubbo:provider | 提供方配置 |
| dubbo:consumer | 消费方配置 |
| dubbo:method | 方法配置 |
| dubbo:argument | 参数配置 |

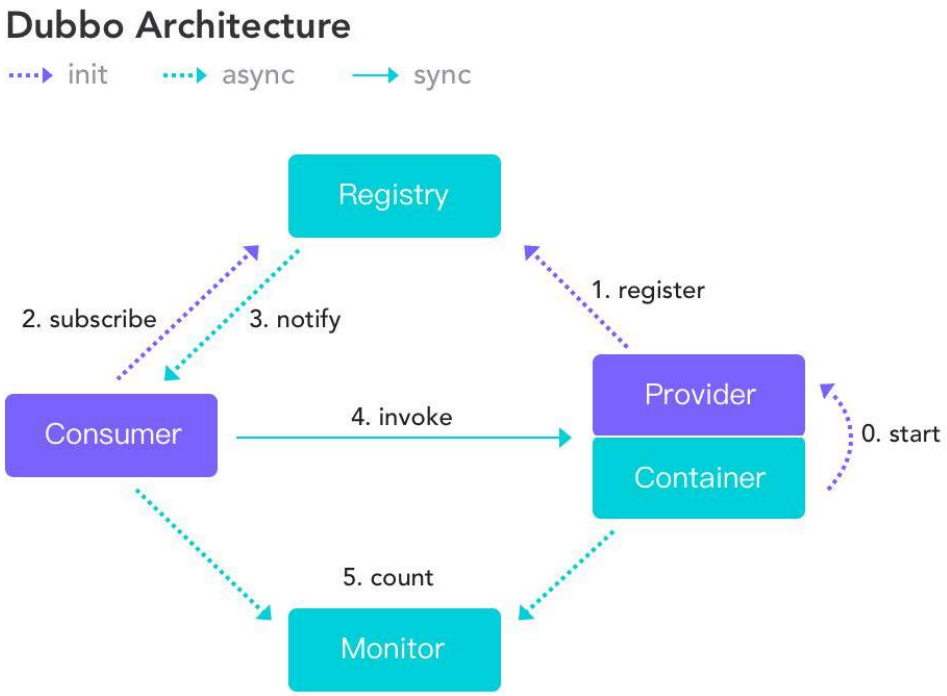
## 11、Dubbo 推荐用什么协议？

- dubbo://（推荐）
- rmi://
- hessian://
- http://
- webservice://
- thrift://
- memcached://
- redis://
- rest://

## 12、同一个服务多个注册的情况下可以直连某一个服务吗？

可以点对点直连， 修改配置即可， 也可以通过 telnet 直接某个服务。

## 13、画一画服务注册与发现的流程图？




## 14、Dubbo 集群容错有几种方案？

| 集群容错方案 | 说明 |
| --- | --- |
| Failover Cluster | 失败自动切换，自动重试其它服务器（默认） |
| Failfast Cluster | 快速失败，立即报错，只发起一次调用 |
| Failsafe Cluster | 失败安全，出现异常时，直接忽略 |
| Failback Cluster | 失败自动恢复，记录失败请求，定时重发 |
| Forking Cluster | 并行调用多个服务器，只要一个成功即返回 |

| | |
|---|---|
| Broadcast Cluster | 广播逐个调用所有提供者，任意一个报错则报错 |

## 15、Dubbo 服务降级，失败重试怎么做？

可以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是“接口名称+Mock”后缀。然后在 Mock 类里实现自己的降级逻辑

## 16、Dubbo 使用过程中都遇到了些什么问题？

在注册中心找不到对应的服务,检查 service 实现类是否添加了@service 注解无法连接到注册中心,检查配置文件中的对应的测试 ip 是否正确

## 17、Dubbo Monitor 实现原理？

Consumer 端在发起调用之前会先走 filter 链； provider 端在接收到请求时也是先走 filter 链， 然后才进行真正的业务逻辑处理。

默认情况下， 在 consumer 和 provider 的 filter 链中都会有 Monitorfilter。1、

MonitorFilter 向 DubboMonitor 发送数据

2、DubboMonitor 将数据进行聚合后（ 默认聚合 1min 中的统计数据） 暂存到 ConcurrentMap<Statistics, AtomicReference> statisticsMap， 然后使用一个含有 3 个线程（ 线程名字： DubboMonitorSendTimer）的线程池每隔 1min 钟， 调用 SimpleMonitorService 遍历发送 statisticsMap 中的统计数据，每发送完毕一个， 就重置当前的 Statistics 的 AtomicReference
3、SimpleMonitorService 将这些聚合数据塞入 BlockingQueue queue 中（ 队列大写为 100000）

4、SimpleMonitorService 使用一个后台线程（线程名为：DubboMonitorAsyncWriteLogThread）将 queue 中的数据写入文件（该线程以死循环的形式来写）

5、SimpleMonitorService 还会使用一个含有 1 个线程（线程名字：DubboMonitorTimer）的线程池每隔 5min 钟，将文件中的统计数据画成图表

## 18、Dubbo 用到哪些设计模式？

Dubbo 框架在初始化和通信过程中使用了多种设计模式，可灵活控制类加载、权限控制等功能。

工厂模式
Provider 在 export 服务时，会调用 ServiceConfig 的 export 方法。ServiceConfig 中有个字段：

```
private static final Protocol protocol =
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtensi on();
```

Dubbo 里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了 JDK SPI 的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在 classpath 下增加个文件就可以了，代码零侵入。另外，像上面的 Adaptive 实现，可以做到调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的实现类。

装饰器模式
Dubbo 在启动和调用阶段都大量使用了装饰器模式。以 Provider 提供的调用链为例，具体的调用链代码是在 ProtocolFilterWrapper 的 buildInvokerChain 完成的，具体是将注解中含有 group=provider 的 Filter 实现，按照 order 排序，最后的调用顺序是：

```
EchoFilter->ClassLoaderFilter->GenericFilter->ContextFilter->
ExecuteLimitFilter -> TraceFilter -> TimeoutFilter -> MonitorFilter ->
ExceptionFilter
```

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter 的作用是判断是否是回声测试请求， 是的话直接返回内容， 这是一种责任链的体现。而像ClassLoaderFilter 则只是在主功能上添加了功能，更改当前线程的 ClassLoader， 这是典型的装饰器模式。

观察者模式
Dubbo 的 Provider 启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服务，订阅时，采用了观察者模式，开启一个 listener。注册中心会每 5 秒定时检查是否有服务更新， 如果有更新， 向该服务的提供者发送一个 notify 消息，provider 接受到 notify 消息后， 即运行 NotifyListener 的 notify 方法， 执行监听器方法。

动态代理模式
Dubbo 扩展 JDK SPI 的类 ExtensionLoader 的 Adaptive 实现是典型的动态代理实现。Dubbo 需要灵活地控制实现类， 即在调用阶段动态地根据参数决定调用哪个实现类， 所以采用先生成代理类的方法， 能够做到灵活的调用。生成代理类的代码是 ExtensionLoader 的 createAdaptiveExtensionClassCode 方法。代理类的主要逻辑是， 获取 URL 参数中指定参数的值作为获取实现类的 key。

## 19、Dubbo 配置文件是如何加载到Spring 中的？

Spring 容器在启动的时候，会读取到 Spring 默认的一些 schema 以及 Dubbo 自定义的 schema， 每个 schema 都会对应一个自己的 NamespaceHandler， NamespaceHandler 里面通过 BeanDefinitionParser 来解析配置信息并转化为需要加载的 bean 对象！

## 20、Dubbo SPI 和 Java SPI 区别？

JDK SPI
JDK 标准的 SPI 会一次性加载所有的扩展实现，如果有的扩展吃实话很耗时，但也没用上，很浪费资源。

所以只希望加载某个的实现， 就不现实了

DUBBO SPI
1， 对 Dubbo 进行扩展， 不需要改动 Dubbo 的源码
2， 延迟加载， 可以一次只加载自己想要加载的扩展实现。
3， 增加了对扩展点 IOC 和 AOP 的支持， 一个扩展点可以直接 setter 注入其它扩展点。
3， Dubbo 的扩展机制能很好的支持第三方 IoC 容器， 默认支持 Spring Bean。

## 21、Dubbo 支持分布式事务吗？

目前暂时不支持， 可与通过 tcc-transaction 框架实现

介绍： tcc-transaction 是开源的 TCC 补偿性分布式事务框架

Git 地址： https://github.com/changmingxie/tcc-transaction

TCC-Transaction 通过 Dubbo 隐式传参的功能， 避免自己对业务代码的入侵。

## 22、Dubbo 可以对结果进行缓存吗？

为了提高数据访问的速度。Dubbo 提供了声明式缓存， 以减少用户加缓存的工作量

<dubbo:reference cache="true" />

其实比普通的配置文件就多了一个标签 cache="true"

## 23、服务上线怎么兼容旧版本？

可以用版本号（ version） 过渡， 多个不同版本的服务注册到注册中心， 版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。

## 24、Dubbo 必须依赖的包有哪些？

Dubbo 必须依赖 JDK， 其他为可选。

## 25、Dubbo telnet 命令能做什么？

dubbo 服务发布之后， 我们可以利用 telnet 命令进行调试、管理。
Dubbo2.0.5 以上版本服务提供端口支持 telnet 命令

连接服务
telnet localhost 20880 //键入回车进入 Dubbo 命令模式。

查看服务列表

```
dubbo>ls
com.test.TestService
```

```
dubbo>ls com.test.TestService
create
delete
query
```

- ls (list services and methods)
- ls : 显示服务列表。
- ls -l : 显示服务详细信息列表。
- ls XxxService：显示服务的方法列表。
- ls -l XxxService：显示服务的方法详细信息列表。

## 26、Dubbo 支持服务降级吗？

以通过 dubbo:reference 中设置 mock="return null"。mock 的值也可以修改为 true，然后再跟接口同一个路径下实现一个 Mock 类，命名规则是“接口名称+Mock”后缀。然后在 Mock 类里实现自己的降级逻辑

## 27、Dubbo 如何优雅停机？

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

## 28、Dubbo 和 Dubbox 之间的区别？

Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加了服务可 Restful 调用，更新了开源组件等。

## 29、Dubbo 和 Spring Cloud 的区别？

根据微服务架构在各方面的要素， 看看 Spring Cloud 和 Dubbo 都提供了哪些支持。

| | Dubbo | Spring Cloud |
|---|---|---|
| 服务注册中心 | Zookeeper | Spring Cloud Netflix Eureka |
| 服务调用方式 | RPC | REST API |
| 服务网关 | 无 | Spring Cloud Netflix Zuul |
| 断路器 | 不完善 | Spring Cloud Netflix Hystrix |
| 分布式配置 | 无 | Spring Cloud Config |
| 服务跟踪 | 无 | Spring Cloud Sleuth |
| 消息总线 | 无 | Spring Cloud Bus |
| 数据流 | 无 | Spring Cloud Stream |
| 批量任务 | 无 | Spring Cloud Task |
| …… | …… | …… |

使用 Dubbo 构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了， 总是让人不怎么放心， 但是如果你是一名高手， 那这些都不是问题； 而 Spring Cloud 就像品牌机， 在Spring Source 的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性， 但是如果要在使用非原装组件外的东西， 就需要对其基础有足够的了解。

## 30、你还了解别的分布式框架吗？

别的还有 spring 的 spring cloud， facebook 的 thrift， twitter 的 finagle 等

# 分布式专题-Elasticsearch 面试题

## 1、elasticsearch 了解多少，说说你们公司es 的集群架构，索引数据大小，分片有多少，以及一些调优手段 。

面试官： 想了解应聘者之前公司接触的 ES 使用场景、规模， 有没有做过比较大规模的索引设计、规划、调优。
解答：
如实结合自己的实践场景回答即可。
比如： ES 集群架构 13 个节点， 索引根据通道不同共 20+索引， 根据日期， 每日递增 20+， 索引： 10 分片， 每日递增 1 亿+数据，
每个通道每天索引大小控制： 150GB 之内。仅索

引层面调优手段：

### 1.1 、设计阶段调优

1、根据业务增量需求， 采取基于日期模板创建索引， 通过 roll over API 滚动索引；

2、使用别名进行索引管理；

3、每天凌晨定时对索引做 force_merge 操作， 以释放空间；

4、采取冷热分离机制，热数据存储到 SSD，提高检索效率；冷数据定期进行 shrink 操作，以缩减存储；

5、采取 curator 进行索引的生命周期管理；

6、仅针对需要分词的字段， 合理的设置分词器；

7、Mapping 阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。… …..

1.2 、写入调优

1、写入前副本数设置为 0；

2、写入前关闭 refresh_interval 设置为-1， 禁用刷新机制； 3、写

入过程中： 采取 bulk 批量写入；

4、写入后恢复副本数和刷新间隔；

5、尽量使用自动生成的 id。

1.3 、查询调优

1、禁用 wildcard；

2、禁用批量 terms（ 成百上千的场景）；

3、充分利用倒排索引机制， 能 keyword 类型尽量 keyword； 4、

数据量大时候， 可以先基于时间敲定索引再检索；

5、设置合理的路由机制。

1.4 、其他调优

部署调优， 业务调优等。

上面的提及一部分， 面试者就基本对你之前的实践或者运维经验有所评估了。

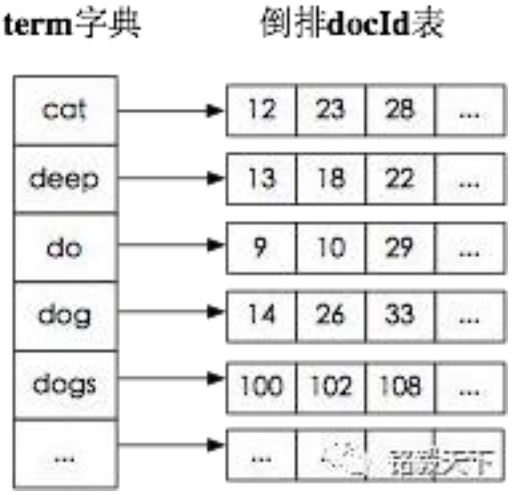
## 2、elasticsearch 的倒排索引是什么

面试官： 想了解你对基础概念的认知。

解答： 通俗解释一下就可以。

传统的我们的检索是通过文章， 逐个遍历找到对应关键词的位置。
而倒排索引，是通过分词策略，形成了词和文章的映射关系表，这种词典+映射表即为倒排索引。

有了倒排索引， 就能实现 o（ 1） 时间复杂度的效率检索文章了， 极大的提高了检索效率。



学术的解答方式：

倒排索引，相反于一篇文章包含了哪些词，它从词出发，记载了这个词在哪些文档中出现过，由两部分组成——词典和倒排表。

加分项：倒排索引的底层实现是基于：FST（Finite State Transducer）数据结构。
lucene 从 4+版本后开始大量使用的数据结构是 FST。FST 有两个优点：

1、空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间；

2、查询速度快。O(len(str))的查询时间复杂度。

# 3、elasticsearch 索引数据多了怎么办，如何调优，部署

面试官：想了解大数据量的运维能力。
解答：索引数据的规划，应在前期做好规划，正所谓“设计先行，编码在后”，这样才能有效的避免突如其来的数据激增导致集群处理能力不足引发的线上客户检索或者其他业务受到影响。
如何调优，正如问题 1 所说，这里细化一下：

## 3.1 动态索引层面

基于模板+时间+rollover api 滚动创建索引，举例：设计阶段定义：blog 索引的模板格式为：blog_index_时间戳的形式，每天递增数据。

这样做的好处：不至于数据量激增导致单个索引数据量非常大，接近于上线 2 的32 次幂-1，索引存储达到了 TB+甚至更大。

一旦单个索引很大，存储等各种风险也随之而来，所以要提前考虑+及早避免。

### 3.2 存储层面

冷热数据分离存储，热数据（比如最近 3 天或者一周的数据），其余为冷数据。对于冷数据不会再写入新数据，可以考虑定期 force_merge 加 shrink 压缩操作，节省存储空间和检索效率。

### 3.3 部署层面

一旦之前没有规划，这里就属于应急策略。
结合 ES 自身的支持动态扩展的特点，动态新增机器的方式可以缓解集群压力，注意：如果之前主节点等规划合理，不需要重启集群也能完成动态新增的。

## 4、elasticsearch 是如何实现master 选举的

面试官：想了解 ES 集群的底层原理，不再只关注业务层面了。

解答：
前置前提：

1、只有候选主节点（master：true）的节点才能成为主节点。

2、最小主节点数（min_master_nodes）的目的是防止脑裂。这个我看

了各种网上分析的版本和源码分析的书籍，云里雾里。
核对了一下代码，核心入口为 findMaster，选择主节点成功返回对应 Master，否则返回 null。选举流程大致描述如下：

第一步：确认候选主节点数达标，elasticsearch.yml 设置的值 discovery.zen.minimum_master_nodes；

第二步： 比较： 先判定是否具备 master 资格， 具备候选主节点资格的优先返回； 若两节点都为候选主节点，则 id 小的值会主节点。注意这里的 id 为 string 类型。

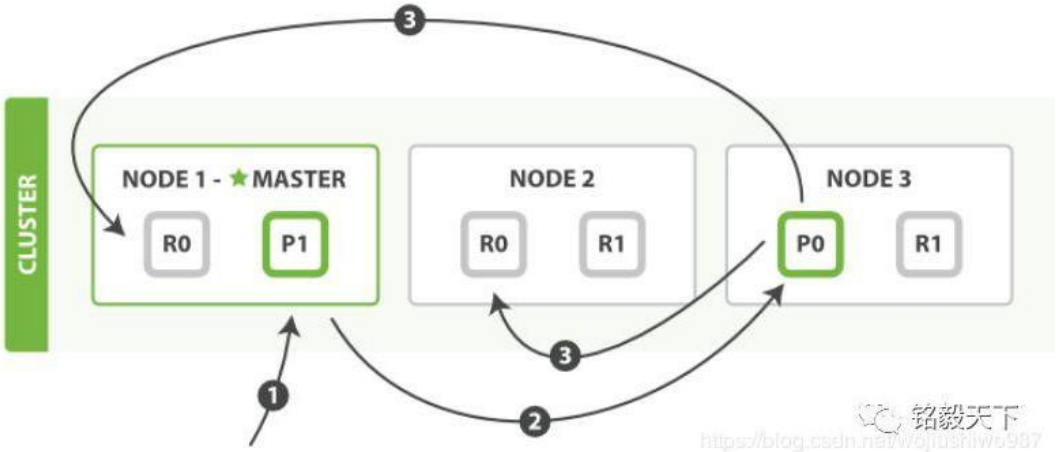
题外话： 获取节点 id 的方法。

```
1GET /_cat/nodes?v&h=ip,port,heapPercent,heapMax,id,name 2ip
                port heapPercent heapMax id          name
```

## 5、详细描述一下Elasticsearch 索引文档的过程

面试官： 想了解 ES 的底层原理， 不再只关注业务层面了。

解答：
这里的索引文档应该理解为文档写入 ES， 创建索引的过程。
文档写入包含：单文档写入和批量 bulk 写入，这里只解释一下：单文档写入流程。记住官方文档中的这个图。




第一步：客户写集群某节点写入数据，发送请求。（ 如果没有指定路由/协调节点， 请求的节点扮演路由节点的角色。）

第二步： 节点 1 接受到请求后， 使用文档_id 来确定文档属于分片 0。请求会被转到另外的节点， 假定节点 3。因此分片 0 的主分片分配到节点 3 上。

第三步： 节点 3 在主分片上执行写操作， 如果成功， 则将请求并行转发到节点 1 和节点 2 的副本分片上， 等待结果返回。所有的副本分片都报告成功， 节点 3 将向协调节点（ 节点 1） 报告成功， 节点 1 向请求客户端报告写入成功。

如果面试官再问： 第二步中的文档获取分片的过程？
回答：借助路由算法获取，路由算法就是根据路由和文档 id 计算目标的分片 id 的过程。

```
1shard = hash(_routing) %  (num_of_primary_shards)
```

## 6、详细描述一下Elasticsearch 搜索的过程？

面试官： 想了解 ES 搜索的底层原理， 不再只关注业务层面了。

解答：
搜索拆解为“query then fetch” 两个阶段。
query 阶段的目的： 定位到位置， 但不取。步骤
拆解如下：

1、假设一个索引数据有 5 主+1 副本 共 10 分片， 一次请求会命中（ 主或者副本分片中） 的一个。

2、每个分片在本地进行查询， 结果返回到本地有序的优先队列中。

3、第 2） 步骤的结果发送到协调节点， 协调节点产生一个全局的排序列表。

fetch 阶段的目的： 取数据。
路由节点获取所有文档， 返回给客户端。

## 7、Elasticsearch 在部署时，对Linux 的设置有哪些优化方法

面试官： 想了解对 ES 集群的运维能力。

解答：

1、关闭缓存 swap;

2、堆内存设置为： Min（ 节点内存/2, 32GB）;

3、设置最大文件句柄数；

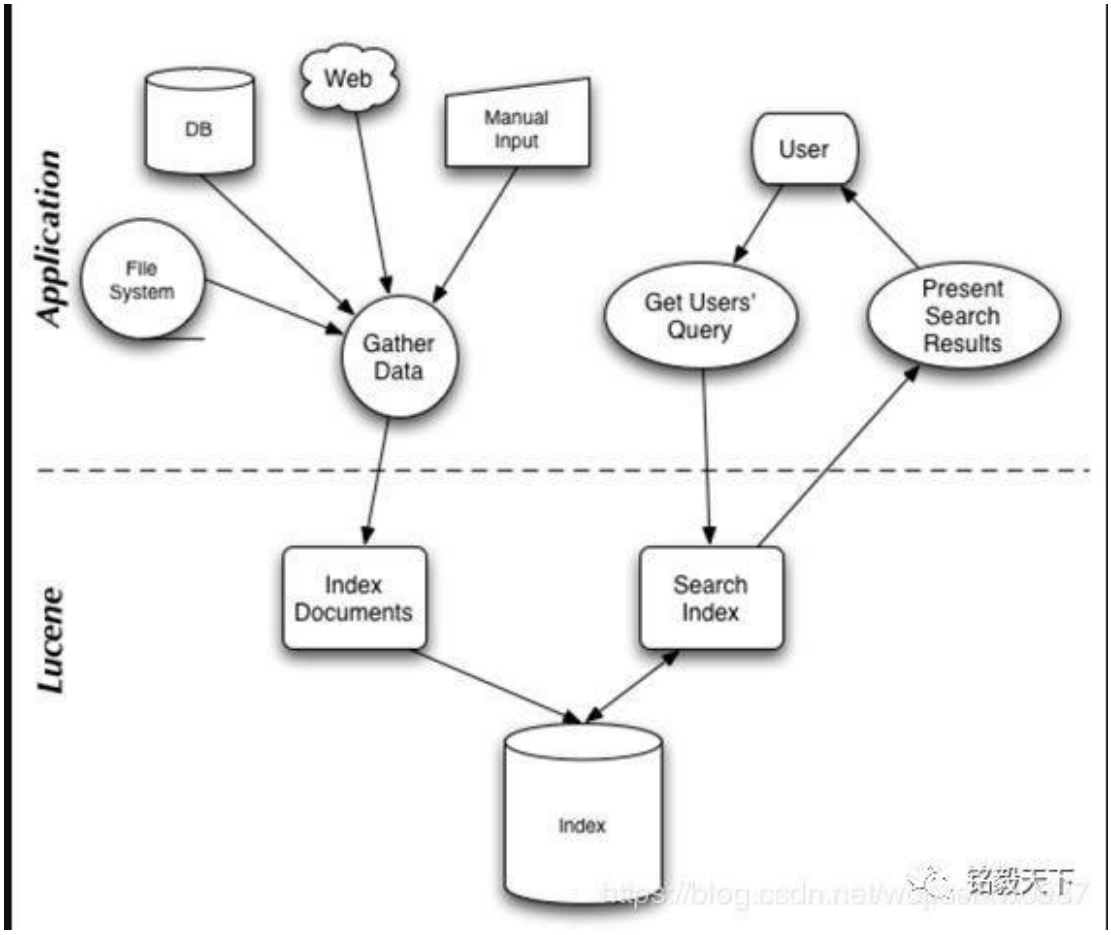
4、线程池+队列大小根据业务需要做调整；

5、磁盘存储 raid 方式—— 存储有条件使用 RAID10，增加单节点性能以及避免单节点存储故障。

## 8、lucence 内部结构是什么？

面试官： 想了解你的知识面的广度和深度。

解答：




Lucene 是有索引和搜索的两个过程， 包含索引创建，索引， 搜索三个要点。可以基于这个脉络展开一些。

最近面试一些公司，被问到的关于 Elasticsearch 和搜索引擎相关的问题，以及自己总结的回答。

## 9、Elasticsearch 是如何实现Master 选举的？

1、Elasticsearch 的选主是 ZenDiscovery 模块负责的， 主要包含 Ping（ 节点之间通过这个 RPC 来发现彼此） 和 Unicast（ 单播模块包含一个主机列表以控制哪些节点需要 ping 通） 这两部分；

2、对所有可以成为 master 的节点（ node.master: true） 根据 nodeId 字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（ 第 0 位） 节点， 暂且认为它是 master 节点。

3、如果对某个节点的投票数达到一定的值（ 可以成为 master 节点数 n/2+1） 并且该节点自己也选举自己， 那这个节点就是 master。否则重新选举一直到满足上述条件。

4、补充： master 节点的职责主要包括集群、节点和索引的管理， 不负责文档级别的管理； data 节点可以关闭 http 功能*。

## 10、Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个master，另外 10 个选了另一个master，怎么办？

1、当集群 master 候选数量不小于 3 个时， 可以通过设置最少投票通过数量（ discovery.zen.minimum_master_nodes） 超过所有候选节点一半以上来解决脑裂问题；
2、当候选数量为两个时， 只能修改为唯一的一个 master 候选， 其他作为 data 节点，避免脑裂问题。

## 11、客户端在和集群连接时，如何选择特定的节点执行请求的？

1、TransportClient 利用 transport 模块远程连接一个 elasticsearch 集群。它并不加入到集群中，只是简单的获得一个或者多个初始化的 transport 地址，并以 轮询 的方式与这些地址进行通信。

## 12、详细描述一下 Elasticsearch 索引文档的过程。

协调节点默认使用文档 ID 参与计算（ 也支持通过 routing）， 以便为路由提供合适的分片。

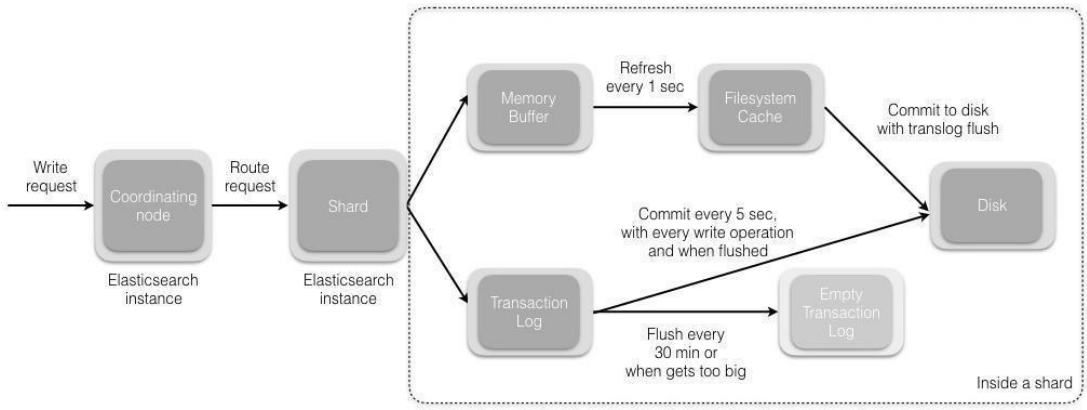
```
shard = hash(document_id) %  (num_of_primary_shards)
```

1、当分片所在的节点接收到来自协调节点的请求后， 会将请求写入到 Memory Buffer，然后定时（ 默认是每隔 1 秒）写入到 Filesystem Cache，这个从 Momery Buffer 到 Filesystem Cache 的过程就叫做 refresh；

2、当然在某些情况下，存在 Momery Buffer 和 Filesystem Cache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystem cache 中的数据写入到磁盘中时， 才会清除掉， 这个过程叫做 flush；

3、在 flush 过程中， 内存中的缓冲将被清除， 内容被写入一个新段， 段的 fsync 将创建一个新的提交点， 并将内容刷新到磁盘， 旧的 translog 将被删除并开始一个新的 translog。

4、flush 触发的时机是定时触发（ 默认 30 分钟） 或者 translog 变得太大（ 默认为 512M） 时；



补充： 关于 Lucene 的 Segement：

1、Lucene 索引是由多个段组成， 段本身是一个功能齐全的倒排索引。

2、段是不可变的， 允许 Lucene 将新的文档增量地添加到索引中， 而不用从头重建索引。

3、对于每一个搜索请求而言， 索引中的所有段都会被搜索， 并且每个段会消耗CPU 的时钟周、文件句柄和内存。这意味着段的数量越多， 搜索性能会越低。

4、为了解决这个问题，Elasticsearch 会合并小段到一个较大的段，提交新的合并段到磁盘，并删除那些旧的小段。

## 13、详细描述一下 Elasticsearch 更新和删除文档的过程。

1、删除和更新也都是写操作， 但是 Elasticsearch 中的文档是不可变的， 因此不能被删除或者改动以展示其变更；

2、磁盘上的每个段都有一个相应的.del 文件。当删除请求发送后，文档并没有真的被删除，而是在.del 文件中被标记为删除。该文档依然能匹配查询， 但是会在结果中被过滤掉。当段合并时， 在.del 文件中被标记为删除的文档将不会被写入新段。

3、在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新时， 旧版本的文档在.del 文件中被标记为删除， 新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询， 但是会在结果中被过滤掉。

## 14、详细描述一下 Elasticsearch 搜索的过程。

1、搜索被执行成一个两阶段过程， 我们称之为 Query Then Fetch；

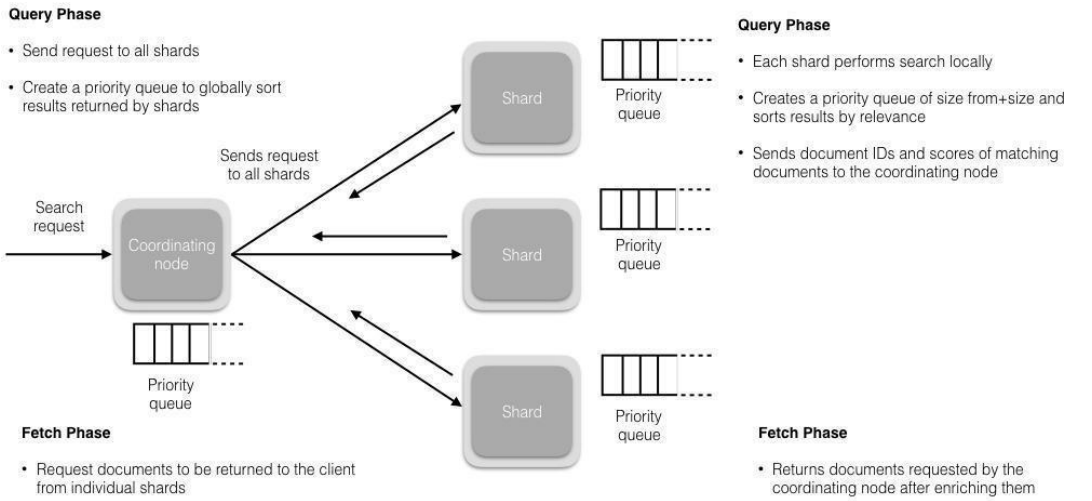
2、在初始查询阶段时，查询会广播到索引中每一个分片拷贝（ 主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。

PS：在搜索的时候是会查询 Filesystem Cache 的，但是有部分数据还在 Memory Buffer， 所以搜索是近实时的。

3、每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

4、接下来就是 取回阶段， 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并 丰富 文档， 如果有需要的话， 接着返回文档给协调节点。一旦所有的文档都被取回了， 协调节点返回结果给客户端。

5、补充： Query Then Fetch 的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFS Query Then Fetch 增加了一个预查询的处理，询问 Term 和 Document frequency，这个评分更准确， 但是性能会变差。*



## 15、在 Elasticsearch 中，是怎么根据一个词找到对应的倒排索引的？

SEE：

- Lucene 的索引文件格式(1)
- Lucene 的索引文件格式(2)

## 16、Elasticsearch 在部署时，对Linux 的设置有哪些优化方法？

1、64 GB 内存的机器是非常理想的， 但是 32 GB 和 16 GB 机器也是很常见的。少于 8 GB 会适得其反。

2、如果你要在更快的 CPUs 和更多的核心之间选择， 选择更多的核心更好。多个内核提供的额外并发远胜过稍微快一点点的时钟频率。

3、如果你负担得起 SSD， 它将远远超出任何旋转介质。 基于 SSD 的节点， 查询和索引性能都有提升。如果你负担得起， SSD 是一个好的选择。

4、即使数据中心们近在咫尺， 也要避免集群跨越多个数据中心。绝对要避免集群跨越大的地理距离。

5、请确保运行你应用程序的 JVM 和服务器的 JVM 是完全一样的。 在 Elasticsearch 的几个地方， 使用 Java 的本地序列化。

6、通过设置 gateway.recover_after_nodes、gateway.expected_nodes、gateway.recover_after_time 可以在集群重启的时候避免过多的分片交换， 这可能会让数据恢复从数个小时缩短为几秒钟。

7、Elasticsearch 默认被配置为使用单播发现， 以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。最好使用单播代替组播。

8、不要随意修改垃圾回收器（ CMS） 和各个线程池的大小。

9、把你的内存的（ 少于） 一半给 Lucene（ 但不要超过 32 GB！）， 通过 ES_HEAP_SIZE 环境变量设置。

10、内存交换到磁盘对服务器性能来说是致命的。如果内存交换到磁盘上， 一个 100 微秒的操作可能变成 10 毫秒。 再想想那么多 10 微秒的操作时延累加起来。 不难看出 swapping 对于性能是多么可怕。

11、Lucene 使用了大量的文件。同时， Elasticsearch 在节点和 HTTP 客户端之间进行通信也使用了大量的套接字。 所有这一切都需要足够的文件描述符。你应该增加你的文件描述符， 设置一个很大的值， 如 64,000。

补充： 索引阶段性能提升方法

1、使用批量请求并调整其大小： 每次批量数据 5– 15 MB 大是个不错的起始点。

2、存储： 使用 SSD

3、段和合并：Elasticsearch 默认值是 20 MB/s，对机械磁盘应该是个不错的设置。如果你用的是 SSD，可以考虑提高到 100– 200 MB/s。如果你在做批量导入， 完全不在意搜索， 你可以彻底关掉合并限流。另外还可以增加

index.translog.flush_threshold_size 设置， 从默认的 512 MB 到更大一些的值， 比如 1 GB，这可以在一次清空触发的时候在事务日志里积累出更大的段。

4、如果你的搜索结果不需要近实时的准确度， 考虑把每个索引的 index.refresh_interval 改到 30s。

5、如果你在做大批量导入，考虑通过设置 index.number_of_replicas: 0 关闭副本。

## 17、对于 GC 方面，在使用Elasticsearch 时要注意什么？

1、SEE： https://elasticsearch.cn/article/32

2、倒排词典的索引需要常驻内存， 无法 GC， 需要监控 data node 上 segment memory 增长趋势。

3、各类缓存， field cache, filter cache, indexing cache, bulk queue 等等， 要设置合理的大小，并且要应该根据最坏的情况来看 heap 是否够用，也就是各类缓存全部占满的时候，还有 heap 空间可以分配给其他任务吗？ 避免采用 clear cache 等“自欺欺人”的方式来释放内存。

4、避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用scan & scroll api 来实现。

5、cluster stats 驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过 tribe node 连接。

6、想知道 heap 够不够， 必须结合实际应用场景， 并对集群的 heap 使用情况做持续的监控。

## 18、Elasticsearch 对于大数据量（上亿量级）的聚合如何实现？

Elasticsearch 提供的首个近似聚合是 cardinality 度量。它提供一个字段的基数，即该字段的 distinct 或者 unique 值的数目。它是基于 HLL 算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 ＝ 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。

## 19、在并发情况下，Elasticsearch 如果保证读写一致？

1、可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；

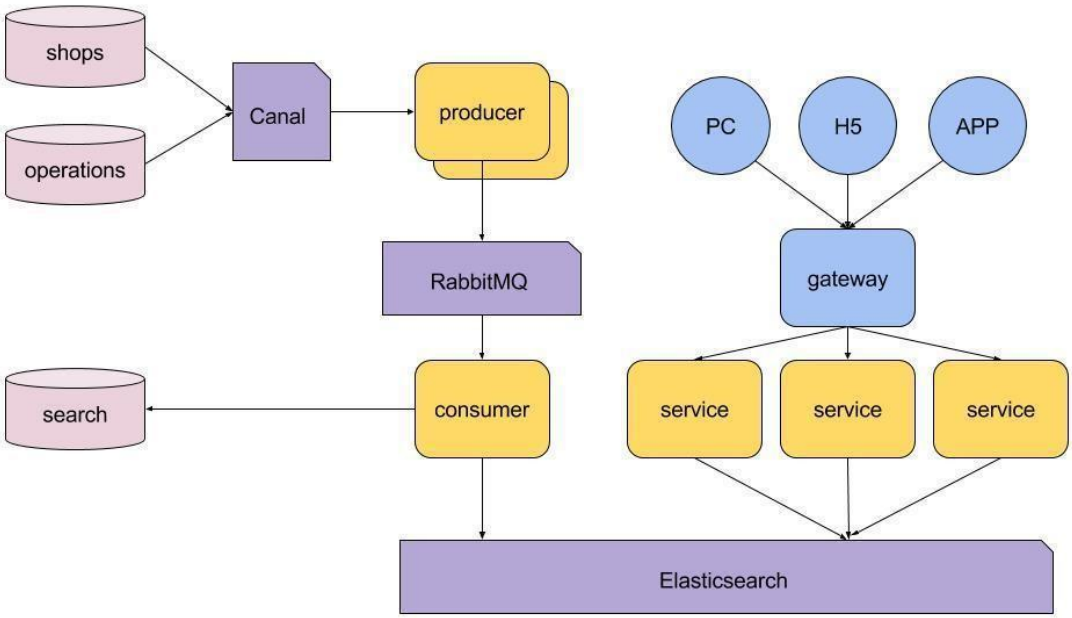
2、另外对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。

3、对于读操作，可以设置 replication 为 sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置 replication 为 async 时，也可以通过设置搜索请求参数 _preference 为 primary 来查询主分片，确保文档是最新版本。

## 20、如何监控 Elasticsearch 集群状态？

Marvel 让你可以很简单的通过 Kibana 监控 Elasticsearch。你可以实时查看你的集群健康状态和性能，也可以分析过去的集群、索引和节点指标。

## 21、介绍下你们电商搜索的整体技术架构。




## 22、介绍一下你们的个性化搜索方案？

SEE 基于 word2vec 和 Elasticsearch 实现个性化搜索

## 23、是否了解字典树？

常用字典数据结构如下所示：

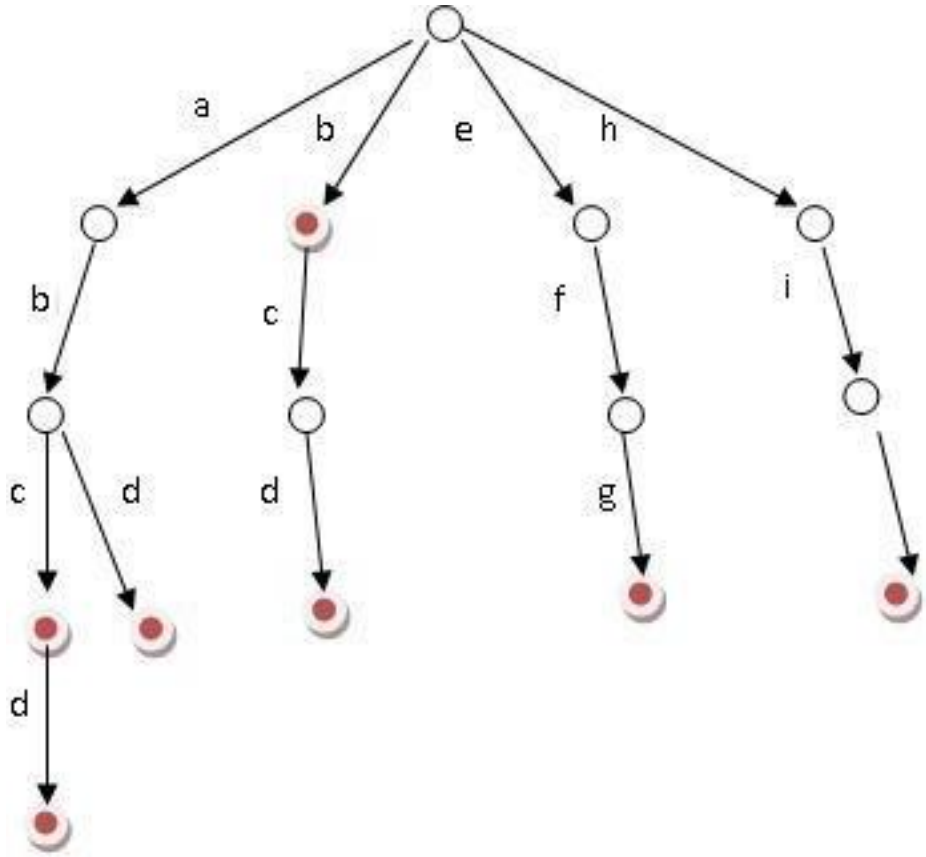
| 数据结构 | 优缺点 |
| --- | --- |
| 排序列表Array/List | 使用二分法查找，不平衡 |
| HashMap/TreeMap | 性能高，内存消耗大，几乎是原始数据的三倍 |
| Skip List | 跳跃表，可快速查找词语，在lucene、redis、Hbase等均有实现。相对于TreeMap等结构，特别适合高并发场景（Skip List介绍） |
| Trie | 适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存（数据结构之trie树） |
| Double Array Trie | 适合做中文词典，内存占用小，很多分词工具均采用此种算法（深入双数组Trie） |
| Ternary Search Tree | 三叉树，每一个node有3个节点，兼具省空间和查询快的优点（Ternary Search Tree） |
| Finite State Transducers (FST) | 一种有限状态转移机，Lucene 4有开源实现，并大量使用 |

Trie 的核心思想是空间换时间， 利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。它有 3 个基本性质：

1、根节点不包含字符， 除根节点外每一个节点都只包含一个字符。

2、从根节点到某一节点， 路径上经过的字符连接起来， 为该节点对应的字符串。

3、每个节点的所有子节点包含的字符都不相同。



1、可以看到， trie 树每一层的节点数是 26^i 级别的。所以为了节省空间， 我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数× 单词长度。

2、实现： 对每个结点开一个字母集大小的数组， 每个结点挂一个链表， 使用左儿子右兄弟表示法记录这棵树；

3、对于中文的字典树，每个节点的子节点用一个哈希表存储，这样就不用浪费太 大的空间，而且查询速度上可以保留哈希的复杂度 O(1)。

## 24、拼写纠错是如何实现的？

1、拼写纠错是基于编辑距离来实现；编辑距离是一种标准的方法，它用来表示经 过插入、删除和替换操作从一个字符串转换到另外一个字符串的最小操作步数；

2、编辑距离的计算过程： 比如要计算 batyu 和 beauty 的编辑距离， 先创建一个 7× 8 的表（ batyu 长度为 5， coffee 长度为 6，各加 2），接着，在如下位置填入黑色数字。其他格的计算过程是取以下三个值的最小值：

如果最上方的字符等于最左方的字符， 则为左上方的数字。否则为左上方的数字 +1。（ 对于 3,3 来说为 0）
左方数字+1（ 对于 3,3 格来说为 2） 上
方数字+1（ 对于 3,3 格来说为 2）

最终取右下角的值即为编辑距离的值 3。

| | | b | e | a | u | t | y |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| b | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| a | 2 | 1 | 1 | 1 | 2 | 3 | 4 |
| t | 3 | 2 | 2 | 2 | 2 | 2 | 3 |
| y | 4 | 3 | 3 | 3 | 3 | 3 | 2 |
| u | 5 | 4 | 4 | 4 | 3 | 4 | 3 |

对于拼写纠错， 我们考虑构造一个度量空间（ Metric Space）， 该空间内任何关系满足以下三条基本条件：

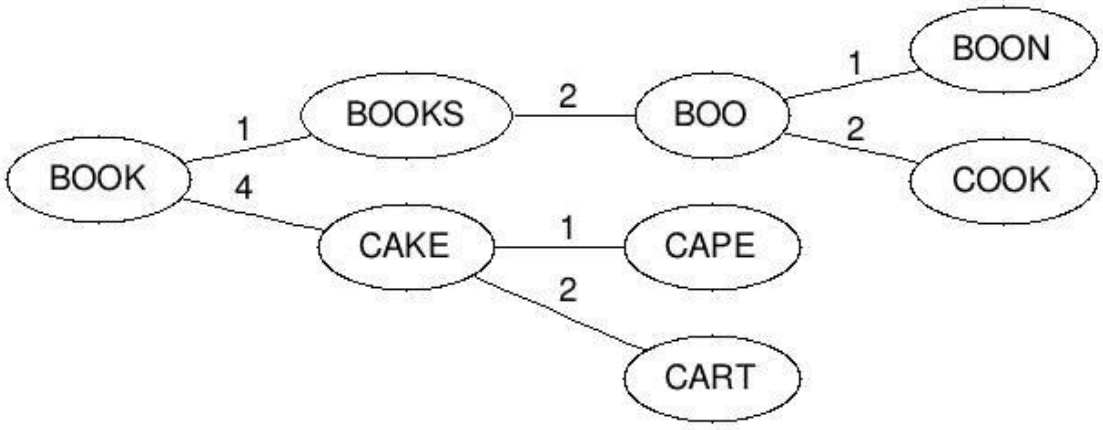
d(x,y) = 0 -- 假如 x 与 y 的距离为 0， 则 x=y
d(x,y) = d(y,x) -- x 到 y 的距离等同于 y 到 x 的距离
d(x,y) + d(y,z) >= d(x,z) -- 三角不等式

1、根据三角不等式，则满足与 query 距离在 n 范围内的另一个字符转 B，其与 A 的距离最大为 d+n， 最小为 d-n。

2、BK 树的构造就过程如下： 每个节点有任意个子节点， 每条边有个值表示编辑距离。所有子节点到父节点的边上标注 n 表示编辑距离恰好为 n。比如，我们有棵

树父节点是“book”和两个子节点“cake”和“books”，“book”到“books”的边标号 1，“book”到“cake”的边上标号 4。从字典里构造好树后，无论何时你想插入新单词时，计算该单词与根节点的编辑距离，并且查找数值为 d(neweord, root)的边。递归得与各子节点进行比较，直到没有子节点，你就可以创建新的子节点并将新单词保存在那。比如，插入“boo”到刚才上述例子的树中，我们先检查根节点，查找 d(“book”, “boo”) = 1 的边，然后检查标号为 1 的边的子节点，得到单词“books”。我们再计算距离 d(“books”, “boo”)=2，则将新单词插在“books”之后，边标号为 2。

3、查询相似词如下：计算单词与根节点的编辑距离 d，然后递归查找每个子节点标号为 d-n 到 d+n（包含）的边。假如被检查的节点与搜索单词的距离 d 小于 n，则返回该节点并继续查询。比如输入 cape 且最大容忍距离为 1，则先计算和根的编辑距离 d(“book”, “cape”)=4，然后接着找和根节点之间编辑距离为 3 到 5 的，这个就找到了 cake 这个节点，计算 d(“cake”, “cape”)=1，满足条件
所以返回 cake，然后再找和 cake 节点编辑距离是 0 到 2 的，分别找到 cape 和cart 节点，这样就得到 cape 这个满足条件的结果。



3、由于 Memcache 没有持久化机制， 因此宕机所有缓存数据失效。Redis 配置为持久化，宕机重启后， 将自动加载宕机时刻的数据到缓存系统中。具有更好的灾备机制。

4、Memcache 可以使用 Magent 在客户端进行一致性 hash 做分布式。Redis 支持在服务器端做分布式（ PS:Twemproxy/Codis/Redis-cluster 多种分布式实现方式）

5、Memcached 的简单限制就是键（ key） 和Value 的限制。最大键长为 250 个字符。可以接受的储存数据不能超过 1MB（ 可修改配置文件变大）， 因为这是典型 slab 的最大值， 不适合虚拟机使用。而 Redis 的 Key 长度支持到 512k。

6、Redis 使用的是单线程模型， 保证了数据按顺序提交。Memcache 需要使用cas 保证数据一致性。CAS（Check and Set） 是一个确保并发一致性的机制， 属于“ 乐观锁” 范畴； 原理很简单： 拿版本号， 操作， 对比版本号， 如果一致就操作， 不一致就放弃任何操作

cpu 利用。由于 Redis 只使用单核， 而 Memcached 可以使用多核， 所以平均每一个核上 Redis 在存储小数据时比 Memcached 性能更 高。而在 100k 以上的数据中，Memcached 性能要高于 Redis 。

7、memcache 内存管理： 使用 Slab Allocation。原理相当简单， 预先分配一系列大小固定的组，然后根据数据大小选择最合适的块存储。避免了内存碎片。（ 缺点： 不能变长， 浪费了一定空间） memcached 默认情况下下一个 slab 的最大值为前一个的 1.25 倍。

8、redis 内存管理： Redis 通过定义一个数组来记录所有的内存分配情况， Redis 采用的是包装的 malloc/free， 相较于 Memcached 的内存 管理方法来说， 要简单很多。由于 malloc 首先以链表的方式搜索已管理的内存中可用的空间分配，导致内存碎片比较多

# 分布式专题-Redis面试题

## 1、什么是Redis?

Redis 是完全开源免费的， 遵守 BSD 协议， 是一个高性能的 key-value 数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
Redis 不仅仅支持简单的 key-value 类型的数据， 同时还提供 list， set， zset， hash 等数据结构的存储。
Redis 支持数据的备份， 即 master-slave 模式的数据备份。

Redis 优 势

性能极高 – Redis 能读的速度是 110000 次/s,写的速度是 81000 次/s 。
丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及Ordered Sets 数据类型操作。
原子 – Redis 的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过 MULTI 和 EXEC 指令包起来。
丰富的特性 – Redis 还支持 publish/subscribe, 通知, key 过期等等特性。

Redis 与其他 key-value 存储有什么不同？

Redis 有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis 的数据类型都是基于基本数据结构的同时对程序员透明， 无需进行额外的抽象。

Redis 运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存， 因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样 Redis 可以做很多内部复杂性很强的事情。同时， 在磁盘格式方面他们是紧凑的以追加的方式产生的， 因为他们并不需要进行随机访问。

## 2、Redis 的数据类型？

答： Redis 支持五种数据类型： string（ 字符串），hash（ 哈希）， list（ 列表）， set（ 集合） 及 zsetsorted set： 有序集合)。

我们实际项目中比较常用的是 string，hash 如果你是 Redis 中高级用户，还需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub。

如果你说还玩过 Redis Module，像 BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

## 3、使用Redis 有哪些好处？

1、速度快， 因为数据存在内存中， 类似于 HashMap， HashMap 的优势就是查找和操作的时间复杂度都是 O1)

2、支持丰富数据类型， 支持 string， list， set， Zset， hash 等

3、支持事务， 操作都是原子性， 所谓的原子性就是对数据的更改要么全部执行， 要么全部不执行

4、丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

## 4、Redis 相比Memcached 有哪些优势？

1、Memcached 所有的值均是简单的字符串， redis 作为其替代者， 支持更为丰富的数据类
2、Redis 的速度比 Memcached 快很3、
Redis 可以持久化其数据

## 5、Memcache 与Redis 的区别都有哪些？

1、存储方式 Memecache 把数据全部存在内存之中， 断电后会挂掉， 数据不能超过内存大小。 Redis 有部份存在硬盘上， 这样能保证数据的持久性。

2、数据支持类型 Memcache 对数据类型支持相对简单。 Redis 有复杂的数据类型。

3、使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。 Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话， 会浪费一定的时间去移动和请求。

## 6、Redis 是单进程单线程的？

答： Redis 是单进程单线程的， redis 利用队列技术将并发访问变为串行访问， 消除了传统数据库串行控制的开销。

## 7、一个字符串类型的值能存储最大容量是多少？

答： 512M

## 8、Redis 的持久化机制是什么？各自的优缺点？

Redis 提供两种持久化机制 RDB 和 AOF 机制:

1、RDBRedis DataBase)持久化方式： 是指用数据集快照的方式半持久化模式) 记录 redis 数据库的所有键值对,在某个时间点将数据写入一个临时文件， 持久化结束后， 用这个临时文件替换上次持久化的文件， 达到数据恢复。

优点：

1、只有一个文件 dump.rdb， 方便持久化。

2、容灾性好， 一个文件可以保存到安全的磁盘。

3、性能最大化， fork 子进程来完成写操作， 让主进程继续处理命令， 所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能) 4.相对于数据集大时， 比 AOF 的启动效率更高。

缺点：

1、数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障， 会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

2、AOFAppend-only file)持久化方式： 是指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 aof 文件。

优点：

1、数据安全， aof 持久化可以配置 appendfsync 属性， 有 always， 每进行一次命令操作就记录到 aof 文件中一次。

2、通过 append 模式写文件， 即使中途服务器宕机， 可以通过 redis-check-aof 工具解决数据一致性问题。

3、AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（ 文件过大时会对命令进行合并重写）， 可以删除其中的某些命令（ 比如误操作的 flushall）)

缺点：

1、AOF 文件比 RDB 文件大， 且恢复速度慢。
2、数据集大的时候， 比 rdb 启动效率低。

## 9、Redis 常见性能问题和解决方案：

1、Master 最好不要写内存快照，如果Master 写内存快照，save 命令调度 rdbSave函数， 会阻塞主线程的工作， 当快照比较大时对性能影响是非常大的， 会间断性暂停服务

2、如果数据比较重要， 某个 Slave 开启 AOF 备份数据， 策略设置为每秒同步一

3、为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网

4、尽量避免在压力很大的主库上增加从

5、主从复制不要用图状结构， 用单向链表结构更为稳定， 即：Master <- Slave1 <- Slave2 <- Slave3… 这样的结构方便解决单点故障问题，实现 Slave 对 Master 的替换。如果 Master 挂了， 可以立刻启用 Slave1 做 Master， 其他不变。

## 10、redis 过期键的删除策略？

1、定时删除:在设置键的过期时间的同时，创建一个定时器 timer). 让定时器在键的过期时间来临时， 立即执行对键的删除操作。

2、惰性删除:放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是 否过期，如果过期的话， 就删除该键;如果没有过期， 就返回该键。

3、定期删除:每隔一段时间程序就对数据库进行一次检查，删除里面的过期键。至 于要删除多少过期键， 以及要检查多少个数据库， 则由算法决定。

## 11、Redis 的回收策略（淘汰策略）?

volatile-lru：从已设置过期时间的数据集（ server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl： 从已设置过期时间的数据集（ server.db[i].expires） 中挑选将要过期的数据淘汰

volatile-random： 从已设置过期时间的数据集（ server.db[i].expires） 中任意选择数据淘汰

allkeys-lru： 从数据集（ server.db[i].dict） 中挑选最近最少使用的数据淘汰

allkeys-random： 从数据集（ server.db[i].dict） 中任意选择数据淘汰

no-enviction（ 驱逐） ： 禁止驱逐数据

注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据， 后面的 lru、ttl 以及 random 是三种不同的淘汰策略， 再加上一种 no-enviction 永不回收的策略。

使用策略规则：

1、如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率 低， 则使用 allkeys-lru

2、如果数据呈现平等分布， 也就是所有的数据访问频率都相同， 则使用 allkeys-random

## 12、为什么 edis 需要把所有数据放到内存中？

答：Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中， 磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天， redis 将会越来越受欢迎。如果设置了最大使用的内存， 则数据已有记录数达到内存限值后不能继续插入新值。

## 13、Redis 的同步机制了解么？

答：Redis 可以使用主从同步，从从同步。第一次同步时，主节点做一次 bgsave， 并同时将后续修改操作记录到内存 buffer， 待完成后将 rdb 文件全量同步到复制节点， 复制节点接受完成后将 rdb 镜像加载到内存。加载完成后， 再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

## 14、Pipeline 有什么好处，为什么要用pipeline？

答：可以将多次 IO 往返的时间缩减为一次，前提是 pipeline 执行的指令之间没有因果相关性。使用 redis-benchmark 进行压测的时候可以发现影响 redis 的 QPS 峰值的一个重要因素是 pipeline 批次指令的数目。

## 15、是否使用过 Redis 集群，集群的原理是什么？

1)、Redis Sentinal 着眼于高可用， 在 master 宕机时会自动将 slave 提升为 master， 继续提供服务。

2)、Redis Cluster 着眼于扩展性， 在单个 redis 内存不足时， 使用 Cluster 进行分片存储。

## 16、Redis 集群方案什么情况下会导致整个集群不可用？

答： 有 A， B， C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了， 那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

## 17、Redis 支持的Java 客户端都有哪些？官方推荐用哪个？

答： Redisson、Jedis、lettuce 等等， 官方推荐使用 Redisson。

## 18、Jedis 与 Redisson 对比有什么优缺点？

答： Jedis 是 Redis 的 Java 实现的客户端， 其 API 提供了比较全面的 Redis 命令的支持；Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作， 不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

## 19、Redis 如何设置密码及验证密码？

设置密码： config set requirepass 123456 授权

密码： auth 123456

## 20、说说 Redis 哈希槽的概念？

答： Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念， Redis 集群有16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽， 集群的每个节点负责一部分 hash 槽。

## 21、Redis 集群的主从复制模型是怎样的？

答： 为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用， 所以集群使用了主从复制模型,每个节点都会有 N-1 个复制品.

## 22、Redis 集群会有写操作丢失吗？为什么？

答：Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

## 23、Redis 集群之间是如何复制的？

答： 异步复制

## 24、Redis 集群最大节点个数是多少？

答： 16384 个。

## 25、Redis 集群如何选择数据库？

答： Redis 集群目前无法做数据库选择， 默认在 0 数据库。

## 26、怎么测试 Redis 的连通性？

答： 使用 ping 命令。

## 27、怎么理解 Redis 事务？

答：

1） 事务是一个单独的隔离操作： 事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中， 不会被其他客户端发送来的命令请求所打断。

2） 事务是一个原子操作： 事务中的命令要么全部被执行， 要么全部都不执行。

## 28、Redis 事务相关的命令有哪几个？

答： MULTI、EXEC、DISCARD、WATCH

## 29、Redis key 的过期时间和永久有效分别怎么设置？

答： EXPIRE 和 PERSIST 命令。

## 30、Redis 如何做内存优化？

答： 尽可能使用散列表（ hashes）， 散列表（ 是说散列表里面存储的数少） 使用的内存非常小， 所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的 web 系统中有一个用户对象， 不要为这个用户的名称， 姓氏， 邮箱， 密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面.

## 31、Redis 回收进程如何工作的？

答： 一个客户端运行了新的命令， 添加了新的数据。Redi 检查内存使用情况， 如果大于 maxmemory 的限制, 则根据设定好的策略进行回收。一个新的命令被执行， 等等。所以我们不断地穿越内存限制的边界， 通过不断达到边界然后不断地回收回到边界以下。如果一个命令的结果导致大量内存被使用（ 例如很大的集合的交集保存到一个新的键）， 不用多久内存限制就会被这个内存使用量超越。

## 32、都有哪些办法可以降低 Redis 的内存使用情况呢？

答：如果你使用的是 32 位的 Redis 实例，可以好好利用 Hash,list,sorted set,set 等集合类型数据， 因为通常情况下很多小的 Key-Value 可以用更紧凑的方式存放到一起。

## 33、Redis 的内存用完了会发生什么？

答：如果达到设置的上限，Redis 的写命令会返回错误信息（ 但是读命令还可以正常返回。） 或者你可以将 Redis 当缓存来使用配置淘汰机制， 当 Redis 达到内存上限时会冲刷掉旧的内容。

## 34、一个 Redis 实例最多能存放多少的 keys？List、Set、Sorted Set 他们最多能存放多少元素？

答：理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。任何 list、set、和 sorted set 都可以放 232 个元素。换句话说， Redis 的存储极限是系统中的可用内存值。

## 35、MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证redis 中的数据都是热点数据？

答： Redis 内存数据集大小上升到一定大小的时候， 就会施行数据淘汰策略。相关知

识： Redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（ server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl： 从已设置过期时间的数据集（ server.db[i].expires） 中挑选将要过期的数据淘汰

volatile-random： 从已设置过期时间的数据集（ server.db[i].expires） 中任意选择数据淘汰

allkeys-lru： 从数据集（ server.db[i].dict） 中挑选最近最少使用的数据淘汰

allkeys-random： 从数据集（ server.db[i].dict） 中任意选择数据淘汰

no-enviction（ 驱逐） ： 禁止驱逐数据

## 36、Redis 最适合的场景？

1、会话缓存（ Session Cache）

最常用的一种使用 Redis 的情景是会话缓存（session cache）。用 Redis 缓存会话比其他存储（ 如 Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时， 如果用户的购物车信息全部丢失， 大部分人都会不高兴的，现在， 他们还会这样吗？ 幸运的是， 随着 Redis 这些年的改进， 很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。

2、全页缓存（ FPC）

除基本的会话 token 之外， Redis 还提供很简便的 FPC 平台。回到一致性问题， 即使重启了 Redis 实例， 因为有磁盘的持久化， 用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。 再次以 Magento 为例， Magento 提供一个插件来使用 Redis 作为全页缓存后端。 此外， 对 WordPress 的用户来说， Pantheon 有一个非常好的插件 wp-redis， 这个插件能帮助你以最快速度加载你曾浏览过的页面。

3、队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。 如果你快速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源项目， 这些项目的目的就是利用 Redis 创建非常好的后端工具， 以满足各种队列需求。例如， Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

4， 排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set） 和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以， 我们要从排序集合中获取到排名最靠前的 10 个用户–我们称之为“user_scores”，我们只需要像下面一样执行即可： 当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数， 你需要这样执行： ZRANGE user_scores 0 10 WITHSCORES Agora Games 就是一个很好的例子， 用 Ruby 实现的， 它的排行榜就是使用 Redis 来存储数据的， 你可以在这里看到。

5、发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用， 还可作为基于发布/订阅的脚本触发器， 甚至用 Redis 的发布/订阅功能来建立聊天系统！

## 37、假如 Redis 里面有 1 亿个key，其中有 10w 个key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

答： 使用 keys 指令可以扫出指定模式的 key 列表。

对方接着追问： 如果这个 redis 正在给线上的业务提供服务， 那使用 keys 指令会有什么问题？

这个时候你要回答 redis 关键的一个特性：redis 的单线程的。keys 指令会导致线程阻塞一段时间， 线上服务会停顿， 直到指令执行完毕， 服务才能恢复。这个时候可以使用 scan 指令， scan 指令可以无阻塞的提取出指定模式的 key 列表， 但是会有一定的重复概率，在客户端做一次去重就可以了， 但是整体所花费的时间会比直接用 keys 指令长。

## 38、如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

答：如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，redis 可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值， 使得过期时间分散一些。

## 39、使用过 Redis 做异步队列么，你是怎么用的？

答：一般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候， 要适当 sleep 一会再重试。

如果对方追问可不可以不用 sleep 呢？

list 还有个指令叫 blpop，在没有消息的时候，它会阻塞住直到消息到来。如果对方追问能不能生产一次消费多次呢？ 使用 pub/sub 主题订阅者模式， 可以实现1:N 的消息队列。

如果对方追问 pub/sub 有什么缺点？

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 RabbitMQ 等。

如果对方追问 redis 如何实现延时队列？

我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话， 怎么问的这么详细。但是你很克制，然后神态自若的回答道：使用 sortedset，拿时间戳作为score，消息内容作为 key 调用 zadd 来生产消息，消费者用 zrangebyscore 指令获取 N 秒之前的数据轮询进行处理。到这里， 面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指， 在椅子背后。

## 40、使用过 Redis 分布式锁么，它是什么回事？

先拿 setnx 来争抢锁， 抢到之后， 再用 expire 给锁加一个过期时间防止锁忘记了释放。

这时候对方会告诉你说你回答得不错， 然后接着问如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了， 那会怎么样？

这时候你要给予惊讶的反馈：唉， 是喔， 这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋， 故作思考片刻， 好像接下来的结果是你主动思考出来的， 然后回答： 我记得 set 指令有非常复杂的参数， 这个应该是可以同时把 setnx 和expire 合成一条指令来用的！对方这时会显露笑容， 心里开始默念： 摁， 这小子还不错。

## 41、如何实现集群中的 session 共享存储？

Session 是运行在一台服务器上的，所有的访问都会到达我们的唯一服务器上，这样我们可以根据客户端传来的 sessionID，来获取 session，或在对应 Session 不存在的情况下（session 生命周期到了/用户第一次登录），创建一个新的 Session； 但是，如果我们在集群环境下， 假设我们有两台服务器 A，B， 用户的请求会由Nginx 服务器进行转发（别的方案也是同理）， 用户登录时， Nginx 将请求转发至服务器 A 上，A 创建了新的 session，并将 SessionID 返回给客户端，用户在浏览其他页面时， 客户端验证登录状态， Nginx 将请求转发至服务器 B， 由于 B 上并没有对应客户端发来 sessionId 的 session，所以会重新创建一个新的 session，并且再将这个新的 sessionID 返回给客户端， 这样， 我们可以想象一下， 用户每一次操作都有 1/2 的概率进行再次的登录， 这样不仅对用户体验特别差， 还会让服务器上的 session 激增， 加大服务器的运行压力。

为了解决集群环境下的 seesion 共享问题， 共有 4 种解决方案：

1. 粘性 session

粘性 session 是指 Ngnix 每次都将同一用户的所有请求转发至同一台服务器上， 即将用户与服务器绑定。

2. 服务器 session 复制

即每次 session 发生变化时， 创建或者修改， 就广播给所有集群中的服务器， 使所有的服务器上的 session 相同。

3. session 共享

缓存 session， 使用 redis， memcached。4.session

持久化

将 session 存储至数据库中， 像操作数据一样才做 session。

## 42、memcached 与redis 的区别？

1、Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset， hash 等数据结构的存储。而 memcache 只支持简单数据类型，需要客户端自己处理复杂对象

2、Redis 支持数据的持久化， 可以将内存中的数据保持在磁盘中， 重启的时候可以再次加载进行使用（ PS： 持久化在 rdb、aof）。

# 分布式专题-RabbitMQ 面试题

## 1、什么是rabbitmq

采用 AMQP 高级消息队列协议的一种消息队列技术,最大的特点就是消费并不需要确保提供方存在,实现了服务之间的高度解耦

## 2、为什么要使用rabbitmq

1、在分布式系统下具备异步,削峰,负载均衡等一系列高级功能;
2、拥有持久化的机制， 进程消息， 队列中的信息也可以保存下来。
3、实现消费者和生产者之间的解耦。
4、对于高并发场景下，利用消息队列可以使得同步访问变为串行访问达到一定量 的限流，利于数据库的操作。
5.可以使用消息队列达到异步下单的效果， 排队中， 后台进行逻辑下单。

## 3、使用rabbitmq 的场景

1、服务间异步通信
2、顺序消费
3、定时任务
4、请求削峰

## 4、如何确保消息正确地发送至RabbitMQ？ 如何确保消息接收方消费了消息？

发送方确认模式

将信道设置成 confirm 模式（ 发送方确认模式）， 则所有在信道上发布的消息都会被指派一个唯一的 ID。
一旦消息被投递到目的队列后， 或者消息被写入磁盘后（ 可持久化的消息）， 信道会发送一个确认给生产者（ 包含消息唯一 ID）。
如果 RabbitMQ 发生内部错误从而导致消息丢失， 会发送一条 nack（ not acknowledged， 未确认） 消息。

发送方确认模式是异步的， 生产者应用程序在等待确认的同时， 可以继续发送消息。当确认消息到达生产者应用程序， 生产者应用程序的回调方法就会被触发来处理确认消息。

接收方确认机制

接收方消息确认机制

消费者接收每一条消息后都必须进行确认（ 消息接收和消息确认是两个不同操作）。
只有消费者确认了消息， RabbitMQ 才能安全地把消息从队列中删除。
这里并没有用到超时机制， RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说， 只要连接不中断， RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性；

下面罗列几种特殊情况

如果消费者接收到消息， 在确认之前断开了连接或取消订阅， RabbitMQ 会认为消息没有被分发， 然后重新分发给下一个订阅的消费者。（ 可能存在消息重复消费的隐患， 需要去重）
如果消费者接收到消息却没有确认消息， 连接也未断开， 则 RabbitMQ 认为该消费者繁忙， 将不会给该消费者分发更多的消息。

## 5.如何避免消息重复投递或重复消费？

在消息生产时， MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id， 作为去重的依据（ 消息投递失败并重传）， 避免重复的消息进入队列；

在消息消费时，要求消息体中必须要有一个 bizId（ 对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等） 作为去重的依据， 避免同一条消息被重复消费。

## 6、消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大， 且并发数受系统资源限制， 会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接， 且每条 TCP 连接上的信道数量没有限制。

## 7、消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（ round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（ 前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能

## 8、消息怎么路由？

消息提供方->路由->一至多个队列
消息发布到交换器时， 消息将拥有一个路由键（ routing key）， 在消息创建时设定。通过队列路由键， 可以把队列绑定到交换器上。
消息到达交换器后， RabbitMQ 会将消息的路由键与队列的路由键进行匹配（ 针对不同的交换器有不同的路由规则）；

常用的交换器主要分为一下三种

fanout： 如果交换器收到消息， 将会广播到所有绑定的队列上direct：
如果路由键完全匹配， 消息就被投递到相应的队列
topic： 可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时， 可以使用通配符

## 9、如何确保消息不丢失？

消息持久化， 当然前提是队列必须持久化
RabbitMQ 确保持久性消息能从服务器重启中恢复的方式是， 将它们写入磁盘上的一个持久化日志文件， 当发布一条持久性消息到持久交换器上时， Rabbit 会在消息提交到日志文件后才发送响应。
一旦消费者从持久队列中消费了一条持久化消息， RabbitMQ 会在持久化日志中把这条消息标记为等待垃圾收集。如果持久化消息在被消费之前 RabbitMQ 重启， 那么 Rabbit 会自动重建交换器和队列（ 以及绑定），并重新发布持久化日志文件中的消息到合适的队列。

## 10、使用 RabbitMQ 有什么好处？

1、服务间高度解耦
2、异步通信性能高
3、流量削峰

## 11、RabbitMQ 的集群

镜像集群模式

你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，然后每次你写消息到 queue 的时候，都会自动把消息到多个实例的 queue 里进行消息同步。

好处在于， 你任何一个机器宕机了， 没事儿， 别的机器都可以用。坏处在于， 第一，这个性能开销也太大了吧， 消息同步所有机器， 导致网络带宽压力和消耗很

重！ 第二， 这么玩儿， 就没有扩展性可言了， 如果某个 queue 负载很重， 你加机器，新增的机器也包含了这个 queue 的所有数据，并没有办法线性扩展你的 queue

## 12、mq 的缺点

系统可用性降低

系统引入的外部依赖越多，越容易挂掉，本来你就是 A 系统调用 BCD 三个系统的接口就好了， 人 ABCD 四个系统好好的， 没啥问题， 你偏加个 MQ 进来， 万一MQ 挂了咋整？ MQ 挂了， 整套系统崩溃了， 你不就完了么。

系统复杂性提高
硬生生加个 MQ 进来，你怎么保证消息没有重复消费？ 怎么处理消息丢失的情况？ 怎么保证消息传递的顺序性？ 头大头大， 问题一大堆， 痛苦不已

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？ 你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构， 你引入它有很多好处， 但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉， 最好之后， 你会发现， 妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的

# 分布式专题-kafka 面试题

## 1、如何获取topic 主题的列表

bin/kafka-topics.sh --list --zookeeper localhost:2181

## 2、生产者和消费者的命令行是什么？

生产者在主题上发布消息：
bin/kafka-console-producer.sh --broker-list 192.168.43.49:9092 --topic Hello-Kafka
注意这里的 IP 是 server.properties 中的 listeners 的配置。接下来每个新行就是输入一条新消息。

消费者接受消息：
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic Hello-Kafka --from-beginning

## 3、consumer 是推还是拉？

Kafka 最初考虑的问题是， customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计： producer 将消息推送到 broker， consumer 从
broker 拉取消息。

一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式， 将消息推送到下游的 consumer。这样做有好处也有坏处： 由 broker 决定消息推送的速率， 对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息， 但不幸的是， push 模式下， 当 broker 推送的速率远大于 consumer 消费的速率时， consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。

Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率， 将可能导致一次只推送较少的消息而造成浪费。Pull 模式下， consumer 就可以根据自己的消费能力去决定这些策略。

Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。

# 4、讲讲kafka 维护消费状态跟踪的方法

大部分消息系统在 broker 端的维护消息被消费的记录： 一个消息被分发到 consumer 后 broker 就马上进行标记或者等待 customer 的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。

但是这样会不会有什么问题呢？ 如果一条消息发送出去之后就立即被标记为消费过的， 一旦 consumer 处理消息时失败了（ 比如程序崩溃） 消息就丢失了。为了解决这个问题， 很多消息系统提供了另外一个个功能： 当消息被发送出去之后仅仅被标记为已发送状态， 当接到 consumer 已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，首先如果 consumer 处理消息成功了但是向 broker 发送响应时失败了，这条消息将被消费两次。第二个问题时， broker 必须维护每条消息的状态， 并且每次都要先锁住消息然后更改状态然后释放锁。这样麻烦又来了， 且不说要维护大量的状态数据， 比如如果消息发送出去但没有收到消费成功的通知， 这条消息将一直处于被锁定的状态， Kafka 采用了不同的策略。Topic 被分成了若干分区，每个分区在同一时间只被一个 consumer 消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态就很容易了，仅仅需要一个整数而已。这样消费状态的跟踪就很简单了。

这带来了另外一个好处： consumer 可以把 offset 调成一个较老的值， 去重新消费老的消息。这对传统的消息系统来说看起来有些不可思议， 但确实是非常有用的， 谁规定了一条消息只能被消费一次呢？

## 5、讲一下主从同步**

https://blog.csdn.net/honglei915/article/details/37565289

## 6、为什么需要消息系统，mysql 不能满足需求吗？

1. 解耦：

允许你独立的扩展或修改两边的处理过程， 只要确保它们遵守同样的接口约束。

2. 冗余：

消息队列把数据进行持久化直到它们已经被完全处理， 通过这一方式规避了数据丢失风险。许多消息队列所采用的”插入-获取-删除”范式中，在把一个消息从队 列中删除之前， 需要你的处理系统明确的指出该消息已经被处理完毕， 从而确保你的数据被安全的保存直到你使用完毕。

3. 扩展性：

因为消息队列解耦了你的处理过程， 所以增大消息入队和处理的频率是很容易的， 只要另外增加处理过程即可。

4. 灵活性 & 峰值处理能力：

在访问量剧增的情况下， 应用仍然需要继续发挥作用， 但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪 费。使用消息队列能够使关键组件顶住突发的访问压力， 而不会因为突发的超负荷的请求而完全崩溃。

5. 可恢复性：

系统的一部分组件失效时， 不会影响到整个系统。消息队列降低了进程间的耦合度， 所以即使一个处理消息的进程挂掉， 加入队列中的消息仍然可以在系统恢复后被处理。

6. 顺序保证：

在大多使用场景下， 数据处理的顺序都很重要。大部分消息队列本来就是排序的， 并且能保证数据会按照特定的顺序来处理。（ Kafka 保证一个 Partition 内的消息的有序性）

7. 缓冲：

有助于控制和优化数据流经过系统的速度， 解决生产消息和消费消息的处理速度不一致的情况。

8. 异步通信：

很多时候， 用户不想也不需要立即处理消息。消息队列提供了异步处理机制， 允许用户把一个消息放入队列， 但并不立即处理它。想向队列中放入多少消息就放多少， 然后在需要的时候再去处理它们。

## 7、Zookeeper 对于Kafka 的作用是什么？

Zookeeper 是一个开放源码的、高性能的协调服务，它用于 Kafka 的分布式应用。

Zookeeper 主要用于在集群中不同节点之间进行通信

在 Kafka 中， 它被用于提交偏移量， 因此如果节点在任何情况下都失败了， 它都可以从之前提交的偏移量中获取

除此之外，它还执行其他活动，如: leader 检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

## 8、数据传输的事务定义有哪三种？

和 MQTT 的事务定义一样都是 3 种。

⑴ ） 最多一次: 消息不会被重复发送， 最多被传输一次， 但也有可能一次不传输

⑵ ） 最少一次: 消息不会被漏发送， 最少被传输一次， 但也有可能被重复传输.

⑶ ） 精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次， 这是大家所期望的

## 9、Kafka 判断一个节点是否还活着有那两个条件？

⑴ ）节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接
⑵ ） 如果节点是个 follower,他必须能及时的同步 leader 的写操作， 延时不能太久

## 10、Kafka 与传统 MQ 消息系统之间有三个关键区别

(1).Kafka 持久化日志， 这些日志可以被重复读取和无限期保留
(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性
(3).Kafka 支持实时的流式处理

## 11、讲一讲 kafka 的 ack 的三种机制

request.required.acks 有三个值 0 1 -1(all)

0:生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候就会丢数据。

1：服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他不确保是否复制完成新 leader 也会导致数据丢失。

-1(all)： 服务端会等所有的 follower 的副本受到数据后才会受到 leader 发出的ack， 这样数据不会丢失

12、消费者如何不自动提交偏移量， 由应用提交？
将 auto.commit.offset 设为 false， 然后在处理一批消息后 commitSync() 或者异步提交 commitAsync()
即：

```
ConsumerRecords<>records=consumer.poll(); for
(ConsumerRecord<> record :records){
。 。 。
tyr{
```

```
consumer.commitSync()
}
。。。
}
```

## 13、消费者故障，出现活锁问题如何解决？

出现“活锁”的情况，是它持续的发送心跳，但是没有处理。为了预防消费者在这种情况下一直持有分区，我们使用 max.poll.interval.ms 活跃检测机制。在此基础上，如果你调用的 poll 的频率大于最大间隔，则客户端将主动地离开组，以便其他消费者接管该分区。发生这种情况时，你会看到 offset 提交失败（调用 commitSync（）引发的 CommitFailedException）。这是一种安全机制，保障只有活动成员能够提交 offset。所以要留在组中，你必须持续调用 poll。

消费者提供两个配置设置来控制 poll 循环：

max.poll.interval.ms：增大 poll 的间隔，可以为消费者提供更多的时间去处理返回的消息（调用 poll(long)返回的消息，通常返回的消息都是一批）。缺点是此值越大将会延迟组重新平衡。

max.poll.records：此设置限制每次调用 poll 返回的消息数，这样可以更容易的预测每次 poll 间隔要处理的最大值。通过调整此值，可以减少 poll 间隔，减少重新平衡分组的

对于消息处理时间不可预测地的情况，这些选项是不够的。处理这种情况的推荐方法是将消息处理移到另一个线程中，让消费者继续调用 poll。但是必须注意确保已提交的 offset 不超过实际位置。另外，你必须禁用自动提交，并只有在线程完成处理后才为记录手动提交偏移量（取决于你）。还要注意，你需要 pause 暂停分区，不会从 poll 接收到新消息，让线程处理完之前返回的消息（如果你的处理能力比拉取消息的慢，那创建新线程将导致你机器内存溢出）。

## 14、如何控制消费的位置

kafka 使用 seek(TopicPartition, long)指定新的消费位置。用于查找服务器保留的最早和最新的 offset 的特殊的方法也可用（seekToBeginning(Collection) 和 seekToEnd(Collection)）

## 15、kafka 分布式（不是单机）的情况下，如何保证消息的顺序消费?

Kafka 分布式的单位是 partition，同一个 partition 用一个 write ahead log 组织，所以可以保证 FIFO 的顺序。不同 partition 之间不能保证顺序。但是绝大多数用户都可以通过 message key 来定义， 因为同一个 key 的 message 可以保证只发送到同一个 partition。

Kafka 中发送 1 条消息的时候， 可以指定(topic, partition, key) 3 个参数。
partiton 和 key 是可选的。如果你指定了 partition，那就是所有消息发往同 1 个 partition，就是有序的。并且在消费端，Kafka 保证，1 个 partition 只能被 1 个 consumer 消费。或者你指定 key（比如 order id），具有同 1 个 key 的所有消息， 会发往同 1 个 partition。

## 16、kafka 的高可用机制是什么？

这个问题比较系统，回答出 kafka 的系统特点，leader 和 follower 的关系，消息读写的顺序即可。

https://www.cnblogs.com/qingyunzong/p/9004703.html

https://www.tuicool.com/articles/BNRza2E

https://yq.aliyun.com/articles/64703

## 17、kafka 如何减少数据丢失

https://www.cnblogs.com/huxi2b/p/6056364.html

## 18、kafka 如何不消费重复数据？比如扣款，我们不能重复的扣。

其实还是得结合业务来思考， 我这里给几个思路：

比如你拿个数据要写库， 你先根据主键查一下， 如果这数据都有了， 你就别插入了， update 一下好吧。

比如你是写 Redis， 那没问题了， 反正每次都是 set， 天然幂等性。

比如你不是上面两个场景， 那做的稍微复杂一点， 你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id， 类似订单 id 之类的东西， 然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？ 如果没有消费过， 你就处理， 然后这个 id 写 Redis。如果消费过了， 那你就别处理了， 保证别重复处理相同的消息即可。

比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束 了，重复数据插入只会报错， 不会导致数据库中出现脏数据。

# Java核心专题-并发编程（一）

## 1、在java 中守护线程和本地线程区别？

java 中的线程分为两种： 守护线程（ Daemon） 和用户线程（ User）。

任何线程都可以设置为守护线程和用户线程，通过方法 Thread.setDaemon(bool on)；true 则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon() 必须在 Thread.start() 之前调用， 否则运行时会抛出异常。

两者的区别：
唯一的区别是判断虚拟机(JVM)何时离开，Daemon 是为其他线程提供服务，如果全部的 User Thread 已经撤离， Daemon 没有可服务的线程， JVM 撤离。也可以理解为守护线程是 JVM 自动创建的线程（但不一定）， 用户线程是程序创建的线程； 比如 JVM 的垃圾回收线程是一个守护线程， 当所有线程已经撤离， 不再产生垃圾， 守护线程自然就没事可干了， 当垃圾回收线程是 Java 虚拟机上仅剩的线程时， Java 虚拟机会自动离开。

扩展： Thread Dump 打印出来的线程信息， 含有 daemon 字样的线程即为守护进程，可能会有：服务守护进程、编译守护进程、windows 下的监听 Ctrl+break 的守护进程、Finalizer 守护进程、引用处理守护进程、GC 守护进程。

## 2、线程与进程的区别？

进程是操作系统分配资源的最小单元， 线程是操作系统调度的最小单元。一个程序

至少有一个进程,一个进程至少有一个线程。

## 3、什么是多线程中的上下文切换？

多线程会共同使用一组计算机上的 CPU，而线程数大于给程序分配的 CPU 数量时，为了让各个线程都有执行的机会，就需要轮转使用 CPU。不同的线程切换使用 CPU 发生的切换数据等就是上下文切换。

## 4、死锁与活锁的区别，死锁与饥饿的区别？

死锁： 是指两个或两个以上的进程（ 或线程） 在执行过程中， 因争夺资源而造成的一种互相等待的现象， 若无外力作用， 它们都将无法推进下去。

产生死锁的必要条件：
1、互斥条件： 所谓互斥就是进程在某一时间内独占资源。
2、请求与保持条件： 一个进程因请求资源而阻塞时， 对已获得的资源保持不放。
3、不剥夺条件:进程已获得资源， 在末使用完之前， 不能强行剥夺。
4、循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

活锁： 任务或者执行者没有被阻塞， 由于某些条件没有满足， 导致一直重复尝试， 失败，尝试， 失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待； 活锁有可能自行解开， 死锁则不能。

饥饿： 一个或者多个线程因为种种原因无法获得所需要的资源， 导致一直无法执行的状态。

Java 中导致饥饿的原因：
1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。
2、线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前 持续地对该同步块进行访问。
3、线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)， 因为其他线程总是被持续地获得唤醒。

## 5、Java 中用到的线程调度算法是什么？

采用时间片轮转的方式。可以设置线程的优先级， 会映射到下层的系统上面的优先级上，如非特别需要， 尽量不要用， 防止线程饥饿。

## 6、什么是线程组，为什么在Java 中不推荐使用？

ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象， 也可以有线程组， 组中还可以有线程， 这样的组织结构有点类似于树的形式。

为什么不推荐使用？ 因为使用有很多的安全隐患吧， 没有具体追究， 如果需要使用，推荐使用线程池。

## 7、为什么使用Executor 框架？

每次执行任务创建线程 new Thread()比较消耗性能， 创建一个线程是比较耗时、耗资源的。
调用 new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制的创建， 线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪， 还有线程之间的频繁交替也会消耗很多系统资源。
接使用 new Thread() 启动的线程不利于扩展， 比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

## 8、在Java 中Executor 和Executors 的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池， 来满足业务的需求。
Executor 接口对象能执行我们的线程任务。
ExecutorService 接口继承了 Executor 接口并进行了扩展， 提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。
使用 ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成， 并可以使用 get()方法获取计算的结果。

## 9、如何在Windows 和Linux 上查找哪个线程使用的 CPU 时间最长？

参考：
http://daiguahub.com/2016/07/31/使用 jstack 找出消耗 CPU 最多的线程代码/

## 10、什么是原子操作？在 Java Concurrency API 中有哪些原子类(atomic classes)？

原子操作（atomic operation） 意为”不可被中断的一个或一系列操作”。处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。 在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。 CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS的原子操作。

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境 下避免数据不一致必须的手段。
int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值， 这就会引发错误。
为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5， java.util.concurrent.atomic 包提供了 int 和long 类型的原子包装类， 它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下， 当有多个线程同时执行这些类的实例包含的方法时， 具有排他性， 即当某个线程进入方法，执行其中的指令时， 不会被其他线程打断， 而别的线程就像自旋锁一样， 一直等到该方法执行完成， 才由 JVM 从等待队列中选择一个另一个线程进入， 这只是一种逻辑上的理解。

原子类： AtomicBoolean， AtomicInteger， AtomicLong， AtomicReference 原子数组：AtomicIntegerArray， AtomicLongArray， AtomicReferenceArray 原子属性更新器：AtomicLongFieldUpdater， AtomicIntegerFieldUpdater， AtomicReferenceFieldUpdater 解决 ABA 问题的原子类： AtomicMarkableReference（ 通过引入一个 boolean 来反映中间有没有变过）， AtomicStampedReference（ 通过引入一个 int 来累加来反映中间有没有变过）

## 11、Java Concurrency API 中的 Lock 接口(Lock interface) 是什么？对比同步它有什么优势？

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。
他们允许更灵活的结构， 可以具有完全不同的性质， 并且可以支持多个相关类的条件对象。

它的优势有：

可以使锁更公平
可以使线程在等待锁的时候响应中断
可以让线程尝试获取锁， 并在无法获取锁的时候立即返回或者等待一段时间可以在不同的范围， 以不同的顺序获取和释放锁

整体上来说 Lock 是 synchronized 的扩展版， Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁， 当然， 在大部分情况下， 非公平锁是高效的选择。

## 12、什么是 Executors 框架？

Executor 框架是一个根据一组执行策略调用， 调度， 执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的 解决方案， 因为可以限制线程的数量并且可以回收再利用这些线程。利用 Executors 框架可以非常方便的创建一个线程池。

## 13、什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（ BlockingQueue） 是一个支持两个附加操作的队列。

这两个附加的操作是： 在队列为空时， 获取元素的线程会等待队列变为非空。当队列满时， 存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景， 生产者是往队列里添加元素的线程， 消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器， 而消费者也只从容器里拿元素。

JDK7 提供了 7 个阻塞队列。分别是：

ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。
LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。
PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。
DelayQueue： 一个使用优先级队列实现的无界阻塞队列。
SynchronousQueue： 一个不存储元素的阻塞队列。
LinkedTransferQueue： 一个由链表结构组成的无界阻塞队列。
LinkedBlockingDeque： 一个由链表结构组成的双向阻塞队列。

Java 5 之前实现同步存取时， 可以使用普通的一个集合， 然后在使用线程的协作和线程同步可以实现生产者， 消费者模式， 主要的技术就是用好，
wait ,notify,notifyAll,sychronized 这些关键字。而在 java 5 之后， 可以使用阻塞队列来实现， 此方式大大简少了代码量， 使得多线程编程更加容易， 安全方面也有保障。
BlockingQueue 接口是 Queue 的子接口， 它的主要用途并不是作为容器， 而是作为线程同步的的工具， 因此他具有一个很明显的特性， 当生产者线程试图向
BlockingQueue 放入元素时， 如果队列已满， 则线程被阻塞， 当消费者线程试图从中取出一个元素时， 如果队列为空， 则该线程会被阻塞， 正是因为它所具有这个特性， 所以在程序中多个线程交替向 BlockingQueue 中放入元素， 取出元素， 它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列， 然后解析线程不断从队列取数据解析。

## 14、什么是 Callable 和Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果， 并且无法抛出返回结果的异常， 而 Callable 功能更强大一些， 被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。
可以认为是带有回调的 Runnable。

Future 接口表示异步任务，是还没有完成的任务给出的未来结果。所以说 Callable 用于产生结果， Future 用于获取结果。

## 15、什么是 FutureTask?使用 ExecutorService 启动任务。

在 Java 并发程序中 FutureTask 表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是调用了 Runnable 接口所以它可以提交给 Executor 来执行。

## 16、什么是并发容器的实现？

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法， 它们将会串行执行。比如 Vector， Hashtable， 以及 Collections.synchronizedSet， synchronizedList 等方法返回的容器。
可以通过查看 Vector， Hashtable 等这些同步容器的实现代码， 可以看到这些容器实现线程安全的方式就是将它们的状态封装起来， 并在需要同步的方法上加上关键字 synchronized。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性， 例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制， 可以称为分段锁， 在这种锁机制下， 允许任意数量的读线程并发地访问 map， 并且执行读操作的线程和写操作的线程也可以并发的访问 map， 同时允许一定数量的写操作线程并发地修改 map， 所以它可以在并发环境下实现更高的吞吐量。

## 17、多线程同步和互斥有几种实现方法，都是什么？

线程同步是指线程之间所具有的一种制约关系， 一个线程的执行依赖另一个线程的消息， 当它没有得到另一个线程的消息时应等待， 直到消息到达时才被唤醒。线程互斥是指对于共享的进程系统资源， 在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时， 任何时刻最多只允许一个线程去使用， 其它要使用该资源的线程必须等待， 直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类： 用户模式和内核模式。顾名思义， 内核模式就是指利用系统内核对象的单一性来进行同步， 使用时需要切换内核态与用户态， 而用户模式就是不需要切换到内核态， 只在用户态完成操作。
用户模式下的方法有： 原子操作（ 例如一个单一的全局变量）， 临界区。内核模式下的方法有： 事件， 信号量， 互斥量。

## 18、什么是竞争条件？你怎样发现和解决竞争？

当多个进程都企图对共享数据进行某种处理， 而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件（ race condition）。

## 19、你将如何使用 thread dump？你将如何分析 Thread dump？

新建状态（ New）

用 new 语句创建的线程处于新建状态，此时它和其他 Java 对象一样，仅仅在堆区中被分配了内存。

就绪状态（ Runnable）

当一个线程对象创建后，其他线程调用它的 start()方法，该线程就进入就绪状态， Java 虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中， 等待获得 CPU 的使用权。

运行状态（ Running）

处于这个状态的线程占用 CPU， 执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

阻塞状态（ Blocked）

阻塞状态是指线程因为某些原因放弃 CPU， 暂时停止运行。当线程处于阻塞状态时， Java 虚拟机不会给线程分配 CPU。直到线程重新进入就绪状态， 它才有机会转到运行状态。

阻塞状态可分为以下 3 种：

位于对象等待池中的阻塞状态（Blocked in object’s wait pool）：

当线程处于运行状态时， 如果执行了某个对象的 wait()方法， Java 虚拟机就会把线程放到这个对象的等待池中， 这涉及到“线程通信”的内容。

位于对象锁池中的阻塞状态（Blocked in object’s lock pool）：

当线程处于运行状态时， 试图获得某个对象的同步锁时， 如果该对象的同步锁已经被其他线程占用， Java 虚拟机就会把这个线程放到这个对象的锁池中， 这涉及到“线程同步”的内容。

其他阻塞状态（ Otherwise Blocked）：

当前线程执行了 sleep()方法，或者调用了其他线程的 join()方法，或者发出了 I/O 请求时，就会进入这个状态。

死亡状态（ Dead）
当线程退出 run()方法时， 就进入死亡状态， 该线程结束生命周期。
我们运行之前的那个死锁代码 SimpleDeadLock.java， 然后尝试输出信息(

```
/* 时间，jvm 信息 */
2017-11-01 17:36:28
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.144-b01 mixed mode):

/* 线程名称：DestroyJavaVM 编
号：#13
优先级：5
系统优先级：0
jvm 内部线程 id：0x0000000001c88800
对应系统线程 id（NativeThread ID）：0x1c18
线程状态： waiting on condition [0x0000000000000000] （等待某个条件） 线
程详细状态：java.lang.Thread.State: RUNNABLE 及之后所有 */
"DestroyJavaVM" #13 prio=5 os_prio=0 tid=0x0000000001c88800 nid=0x1c18
waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"Thread-1" #12 prio=5 os_prio=0 tid=0x0000000018d49000
nid=0x17b8 waiting for monitor entry [0x0000000019d7f000]
/* 线程状态：阻塞（在对象同步上）
     代码位置：at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56) 等待
     锁：0x00000000d629b4d8
     已经获得锁：0x00000000d629b4e8*/
    java.lang.Thread.State: BLOCKED (on object monitor) at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
```

```
        - waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
        - locked <0x00000000d629b4e8> (a java.lang.Object)

"Thread-0" #11 prio=5 os_prio=0 tid=0x0000000018d44000 nid=0x1ebc waiting for
monitor entry [0x000000001907f000]
    java.lang.Thread.State: BLOCKED (on object monitor) at
        com.leo.interview.SimpleDeadLock$A.run(SimpleDeadLock.java:34)
    -       waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
        - locked <0x00000000d629b4d8> (a java.lang.Object)

"Service Thread" #10 daemon prio=9 os_prio=0 tid=0x0000000018ca5000
nid=0x1264 runnable [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #9 daemon prio=9 os_prio=2 tid=0x0000000018c46000
nid=0xb8c waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os_prio=2 tid=0x0000000018be4800
nid=0x1db4 waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os_prio=2 tid=0x0000000018be3800
nid=0x810 waiting on condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE
```

```
"Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0 tid=0x0000000018bcc800
nid=0x1c24 runnable [0x00000000193ce000]

    java.lang.Thread.State: RUNNABLE
      at java.net.SocketInputStream.socketRead0(Native Method) at
java.net.SocketInputStream.socketRead(SocketInputStream.java:116)       at
      java.net.SocketInputStream.read(SocketInputStream.java:171)       at
      java.net.SocketInputStream.read(SocketInputStream.java:141)       at
      sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)        at
      sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)         at
      sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
      - locked <0x00000000d632b928> (a java.io.InputStreamReader)  at
      java.io.InputStreamReader.read(InputStreamReader.java:184)        at
      java.io.BufferedReader.fill(BufferedReader.java:161)

      at java.io.BufferedReader.readLine(BufferedReader.java:324)
      - locked <0x00000000d632b928> (a java.io.InputStreamReader)  at
      java.io.BufferedReader.readLine(BufferedReader.java:389)
      at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:6
4)

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x0000000017781800
nid=0x524 runnable [0x0000000000000000]

    java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x000000001778f800
nid=0x1b08 waiting on condition [0x0000000000000000]

    java.lang.Thread.State: RUNNABLE
```

```
"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001776a800 nid=0xdac in
Object.wait() [0x0000000018b6f000]
    java.lang.Thread.State: WAITING (on object monitor) at
     java.lang.Object.wait(Native Method)
     - waiting on <0x00000000d6108ec8> (a java.lang.ref.ReferenceQueue$Lock)
     at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
     - locked <0x00000000d6108ec8> (a java.lang.ref.ReferenceQueue$Lock)
     at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164) at
     java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x0000000017723800
nid=0x1670 in Object.wait() [0x00000000189ef000]
    java.lang.Thread.State: WAITING (on object monitor) at
     java.lang.Object.wait(Native Method)
     - waiting on <0x00000000d6106b68> (a java.lang.ref.Reference$Lock)
     at java.lang.Object.wait(Object.java:502)
     at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
     - locked <0x00000000d6106b68>  (a  java.lang.ref.Reference$Lock) at
     java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"VM Thread" os_prio=2 tid=0x000000001771b800 nid=0x604 runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x0000000001c9d800 nid=0x9f0
runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x0000000001c9f000 nid=0x154c
runnable
```

```
"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x0000000001ca0800 nid=0xcd0
runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x0000000001ca2000 nid=0x1e58
runnable

"VM Periodic Task Thread" os_prio=2 tid=0x0000000018c5a000
nid=0x1b58 waiting on condition

JNI global references: 33


/* 此处可以看待死锁的相关信息！ */
Found one Java-level  deadlock:
=============================

"Thread-1":
   waiting to lock monitor 0x0000000017729fc8 (object 0x00000000d629b4d8, a
java.lang.Object),
   which is held by "Thread-0"
"Thread-0":
   waiting to lock monitor 0x0000000017727738 (object 0x00000000d629b4e8,
a java.lang.Object),
   which is held by "Thread-1"


Java stack information for the threads listed above:
===================================================
=====
"Thread-1": at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
```

```
        - waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
        - locked <0x00000000d629b4e8> (a java.lang.Object)
"Thread-0":
        at com.leo.interview.SimpleDeadLock$A.run(SimpleDeadLock.java:34)
        - waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
        - locked <0x00000000d629b4d8> (a  java.lang.Object)

Found 1 deadlock.

/* 内存使用状况，详情得看 JVM 方面的书 */
Heap
 PSYoungGen              total 37888K, used 4590K [0x00000000d6100000,
0x00000000d8b00000, 0x0000000100000000)
   eden space 32768K, 14% used
[0x00000000d6100000,0x00000000d657b968,0x00000000d8100000)
   from space 5120K, 0% used
[0x00000000d8600000,0x00000000d8600000,0x00000000d8b00000)
   to      space 5120K, 0% used
[0x00000000d8100000,0x00000000d8100000,0x00000000d8600000)
 ParOldGen               total 86016K, used 0K [0x0000000082200000,
0x0000000087600000, 0x00000000d6100000)
   object space 86016K, 0% used
[0x0000000082200000,0x0000000082200000,0x0000000087600000)
 Metaspace               used 3474K, capacity 4500K, committed 4864K,
reserved 1056768K
   class space         used 382K, capacity 388K, committed 512K, reserved
1048576K
```

## 20、为什么我们调用 start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

当你调用 start()方法时你将创建新的线程， 并且执行在 run()方法里的代码。
但是如果你直接调用 run()方法，它不会创建新的线程也不会执行调用线程的代码， 只会把 run 方法当作普通方法去执行。

## 21、Java 中你怎样唤醒一个阻塞的线程？

在 Java 发展史上曾经使用 suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题， 比较典型的还是死锁问题。
解决方案可以使用以对象为目标的阻塞， 即利用 Object 类的 wait()和 notify()方法实现线程阻塞。
首先，wait、notify 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程， 但它需要重新获取改对象的锁， 直到获取成功才能往下执行； 其次， wait、notify 方法必须在 synchronized 块或方法中被调用， 并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个， 如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

## 22、在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别？

CyclicBarrier 可以重复使用， 而 CountdownLatch 不能重复使用。

Java 的 concurrent 包里面的 CountDownLatch 其实可以把它看作一个计数器， 只不过这个计数器的操作是原子操作， 同时只能有一个线程去操作这个计数器， 也就是同时只能有一个线程去减这个计数器里面的值。
你可以向 CountDownLatch 对象设置一个初始的数字作为计数值，任何调用这个对象上的 await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为 0 为止。
所以在当前计数到达零之前， await 方法会一直受阻塞。之后， 会释放所有等待的线程， await 的所有后续调用都将立即返回。这种现象只出现一次— — 计数无法被重置。如果需要重置计数， 请考虑使用 CyclicBarrier。
CountDownLatch 的一个非常典型的应用场景是： 有一个任务想要往下执行， 但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个 CountDownLatch 对象的 await()方法， 其他的任务执行完自己的任务后调用同一个 CountDownLatch 对象上的 countDown()方法， 这个调用 await()方法的任务将一直阻塞等待， 直到这个 CountDownLatch 对象的计数值减到 0 为止。

CyclicBarrier 一个同步辅助类， 它允许一组线程互相等待， 直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待， 此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用， 所以称它为循环 的 barrier。

## 23、什么是不可变对象，它对写并发应用有什么帮助？

不可变对象(Immutable Objects)即对象一旦被创建它的状态（ 对象的数据，也即对象属性值） 就不能改变， 反之即为可变对象(Mutable Objects)。
不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类， 如 String、基本类型的包装类、BigInteger 和 BigDecimal 等。
不可变对象天生是线程安全的。它们的常量（ 域） 是在构造函数中创建的。既然它们的状态无法修改， 这些常量永远不会变。

不可变对象永远是线程安全的。
只有满足如下状态， 一个对象才是不可变的； 它
的状态不能在创建后再被修改；

所有域都是 final 类型； 并且，
它被正确创建（ 创建期间没有发生 this 引用的逸出）。

## 24、什么是多线程中的上下文切换？

在上下文切换过程中， CPU 会停止处理当前运行的程序， 并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看， 上下文切换有点像我们同时阅读几本书， 在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中， 上下文切换过程中的“ 页码” 信息是保存在进程控制块（ PCB） 中的。PCB 还经常被称作“ 切换桢” （ switchframe）。“ 页码” 信息会一直保存到 CPU 的内存中， 直到他们被再次使用。
上下文切换是存储和恢复 CPU 状态的过程， 它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

## 25、Java 中用到的线程调度算法是什么？

计算机通常只有一个 CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU 的使用权才能执行指令.所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得 CPU 的使用权,分别执行各自的任务.在运行池中,会有多个处于就绪状态的线程在等待 CPU,JAVA 虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配 CPU 的使用权.

有两种调度模型： 分时调度模型和抢占式调度模型。
分时调度模型是指让所有的线程轮流获得 cpu 的使用权,并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

java 虚拟机采用抢占式调度模型， 是指优先让可运行池中优先级高的线程占用CPU， 如果可运行池中的线程优先级相同， 那么就随机选择一个线程， 使其占用CPU。处于运行状态的线程会一直运行， 直至它不得不放弃 CPU。

## 26、什么是线程组，为什么在 Java 中不推荐使用？

线程组和线程池是两个不同的概念， 他们的作用完全不同， 前者是为了方便线程的管理，后者是为了管理线程的生命周期， 复用线程， 减少创建销毁线程的开销。

## 27、为什么使用 Executor 框架比使用应用创建和管理线程好？

为什么要使用 Executor 线程池框架

1、每次执行任务创建线程 new Thread()比较消耗性能， 创建一个线程是比较耗时、耗资源的。

2、调用 new Thread()创建的线程缺乏管理， 被称为野线程， 而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪， 还有线程之间的频繁交替也会消耗很多系统资源。

3、直接使用 new Thread() 启动的线程不利于扩展， 比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

使用 Executor 线程池框架的优点

1、能复用已存在并空闲的线程从而减少线程对象的创建从而减少了消亡线程的开 销。

2、可有效控制最大并发线程数， 提高系统资源使用率， 同时避免过多资源竞争。

3、框架中已经有定时、定期、单线程、并发数控制等功能。

综上所述使用线程池框架 Executor 能更好的管理线程、提供系统资源使用率。

## 28、java 中有几种方法可以实现一个线程？

继 承 Thread 类
实现 Runnable 接口
实现 Callable 接口， 需要实现的是 call() 方法

## 29、如何停止一个正在运行的线程？

使用共享变量的方式
在这种方式中， 之所以引入共享变量， 是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号， 通知中断线程的执行。

使用 interrupt 方法终止线程

如果一个线程由于等待某些事件的发生而被阻塞， 又该怎样停止该线程呢？ 这种情况经常会发生， 比如当一个线程由于需要等候键盘输入而被阻塞， 或者调用Thread.join()方法，或者 Thread.sleep()方法， 在网络中调用 ServerSocket.accept()方法，或者调用了 DatagramSocket.receive()方法时，都有可能导致线程阻塞， 使线程处于处于不可运行状态时， 即使主程序中将该线程的共享变量设置为 true， 但该线程此时根本无法检查循环标志， 当然也就无法立即中断。这里我们给出的建议是，不要使用 stop()方法，而是使用 Thread 提供的interrupt()方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常， 从而使线程提前结束阻塞状态， 退出堵塞代码。

## 30、notify()和 notifyAll()有什么区别？

当一个线程进入 wait 之后，就必须等其他线程 notify/notifyall,使用 notifyall,可以唤醒所有处于 wait 状态的线程，使其重新进入锁的争夺队列中，而 notify 只能唤醒一个。

如果没把握， 建议 notifyAll， 防止 notigy 因为信号丢失而造成程序异常。

## 31、什么是 Daemon 线程？它有什么意义？

所谓后台(daemon)线程， 是指在程序运行的时候在后台提供一种通用服务的线程， 并且这个线程并不属于程序中不可或缺的部分。因此， 当所有的非后台线程结束时， 程序也就终止了， 同时会杀死进程中的所有后台线程。反过来说，

只要有任何非后台线程还在运行， 程序就不会终止。必须在线程启动之前调用 setDaemon()方法， 才能把它设置为后台线程。注意： 后台进程在不执行 finally 子句的情况下就会终止其 run()方法。

比如： JVM 的垃圾回收线程就是 Daemon 线程， Finalizer 也是守护线程。

## 32、java 如何实现多线程之间的通讯和协作？

中断 和 共享变量

## 33、什么是可重入锁（ReentrantLock）？

举例来说明锁的可重入性

```
public class
    UnReentrant{ Lock lock =
    new Lock(); public void
    outer(){
```

```
            lock.lock();
            inner();
            lock.unlock();
        }
        public void
            inner(){ lock.lock()
            ;
            //do something
            lock.unlock();
        }
}
```

outer 中调用了 inner，outer 先锁住了 lock，这样 inner 就不能再获取 lock。其实调用 outer 的线程已经获取了 lock 锁， 但是不能在 inner 中重复利用已经获取的锁资源，这种锁即称之为 不可重入可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

synchronized、ReentrantLock 都是可重入的锁， 可重入锁相对来说简化了并发编程的开发。

## 34、当一个线程进入某个对象的一个 synchronized 的实例方法后，其它线程是否可进入此对象的其它方法？

如果其他方法没有 synchronized 的话， 其他线程是可以进入的。

所以要开放一个线程安全的对象时， 得保证每个方法都是线程安全的。

## 35、乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

悲观锁： 总是假设最坏的情况， 每次去拿数据的时候都认为别人会修改， 所以每次在拿数据的时候都会上锁， 这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制， 比如行锁， 表锁等， 读锁， 写锁等， 都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁： 顾名思义， 就是很乐观， 每次去拿数据的时候都认为别人不会修改， 所 以不会上锁， 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据， 可以使用版本号等机制。乐观锁适用于多读的应用类型， 这样可以提高吞吐量， 像数据库提供的类似于 write_condition 机制， 其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实 现方式 CAS 实现的。

乐观锁的实现方式：
1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标 识， 不一致时可以采取丢弃和再次尝试的策略。
2、java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

CAS 缺点：

1、ABA 问题：
比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A， 这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、循环时间长开销大：
对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：
当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

## 36、SynchronizedMap 和 ConcurrentHashMap 有什么区别？

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。
ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get,put,remove 等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。
另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变。

## 37、CopyOnWriteArrayList 可以用于什么应用场景？

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在
CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时， 读取操作可以安全地执行。

1、由于写操作的时候， 需要拷贝数组， 会消耗内存， 如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc；
2、不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的,虽然 CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求；

CopyOnWriteArrayList 透露的思想
1、读写分离， 读和写分开
2、最终一致性
3、使用另外开辟空间的思路， 来解决并发冲突

## 38、什么叫线程安全？servlet 是线程安全吗?

线程安全是编程中的术语， 指某个函数、函数库在多线程环境中被调用时， 能够正确地处理多个线程之间的共享变量， 使程序功能正确完成。

Servlet 不是线程安全的，servlet 是单实例多线程的，当多个线程同时访问同一个方法， 是不能保证共享变量的线程安全性的。
Struts2 的 action 是多实例多线程的， 是线程安全的， 每个请求过来都会 new 一个新的 action 分配给这个请求， 请求完成后销毁。
SpringMVC 的 Controller 是线程安全的吗？ 不是的，和 Servlet 类似的处理流程。

Struts2 好处是不用考虑线程安全问题； Servlet 和 SpringMVC 需要考虑线程安全问题，但是性能可以提升不用处理太多的 gc，可以使用 ThreadLocal 来处理多线程的问题。

## 39、volatile 有什么用？能否用一句话说明下volatile 的应用场景？

volatile 保证内存可见性和禁止指令重排。

volatile 用于多线程环境下的单次操作(单次读或者单次写)。

## 40、为什么代码会重排序？

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

在单线程环境下不能改变程序运行的结果；
存在数据依赖关系的不允许重排序
需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

## 41、在 java 中wait 和sleep 方法的不同？

最大的不同是在等待时 wait 会释放锁，而 sleep 一直持有锁。Wait 通常被用于线程间交互，sleep 通常被用于暂停执行。

直接了解的深入一点吧：

在 Java 中线程的状态一共被分成 6 种：

初始态： NEW

创建一个 Thread 对象， 但还未调用 start()启动线程时， 线程处于初始态。运行态：

RUNNABLE

在 Java 中， 运行态包括就绪态 和 运行态。
就绪态 该状态下的线程已经获得执行所需的所有资源， 只要 CPU 分配执行权就能运行。
所有就绪态的线程存放在就绪队列中。
运行态 获得 CPU 执行权， 正在执行的线程。由于一个 CPU 同一时刻只能执行一条线程，
因此每个 CPU 每个时刻只有一条运行态的线程。

阻塞态

当一条正在执行的线程请求某一资源失败时， 就会进入阻塞态。而在 Java 中， 阻塞态专指请求锁失败时进入的状态。由一个阻塞队列存放所有阻塞态的线程。处于阻塞态的线程会不断请求资源， 一旦请求成功， 就会进入就绪队列， 等待执行。PS： 锁、IO、Socket 等都资源。

等待态

当前线程中调用 wait、join、park 函数时， 当前线程就会进入等待态。也有一个等待队列存放所有等待态的线程。线程处于等待态表示它需要等待其他线程的指示才能继续运行。进入等待态的线程会释放 CPU 执行权， 并释放资源（ 如： 锁）

超时等待态

当运行中的线程调用 sleep(time)、wait、join、parkNanos、parkUntil 时， 就会进入该状态； 它和等待态一样， 并不是因为请求不到资源， 而是主动进入， 并且进入后需要其他线程唤醒；进入该状态后释放 CPU 执行权 和 占有的资源。与等待态的区别： 到了超时时间后自动进入阻塞队列， 开始竞争锁。

终止态

线程执行结束后的状态。

注意：

wait()方法会释放 CPU 执行权 和 占有的锁。
sleep(long)方法仅释放 CPU 使用权，锁仍然占用；线程被放入超时等待队列，与yield 相比，它会使线程较长时间得不到运行。
yield()方法仅释放 CPU 执行权， 锁仍然占用， 线程会被放入就绪队列， 会在短时间内再次执行。

wait 和 notify 必须配套使用， 即必须使用同一把锁调用；
wait 和 notify 必须放在一个同步块中调用 wait 和 notify 的对象必须是他们所处同步块的锁对象。

## 42、用 Java 实现阻塞队列

参考 java 中的阻塞队列的内容吧， 直接实现有点烦：
http://www.infoq.com/cn/articles/java-blocking-queue

## 43、一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用 Thread.getUncaughtExceptionHandler() 来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给

handler 的 uncaughtException()方法进行处理。

## 44、如何在两个线程间共享数据？

在两个线程间共享变量即可实现共享。
一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

## 45、Java 中 notify 和 notifyAll 有什么区别？

notify() 方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而 notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

## 46、为什么 wait, notify 和 notifyAll 这些方法不在 thread 类里面？

一个很明显的原因是 JAVA 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。由于 wait，notify 和 notifyAll 都是锁级别的操作，所以把他们定义在 Object 类中因为锁属于对象。

## 47、什么是 ThreadLocal 变量？

ThreadLocal 是 Java 里一种特殊的变量。每个线程都有一个 ThreadLocal 就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用 ThreadLocal 让 SimpleDateFormat 变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己

独有的变量拷贝， 将大大提高效率。首先， 通过复用减少了代价高昂的对象的创建个数。其次， 你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

## 48、Java 中 interrupted 和 isInterrupted 方法的区别？

interrupt

interrupt 方法用于中断线程。调用该方法的线程的状态为将被置为"中断"状态。注意： 线程中断仅仅是置线程的中断状态位， 不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（ 也就是线程中断后会抛出 interruptedException 的方法） 就是在监视线程的中断状态， 一旦线程的中断状态被置为“中断状态”， 就会抛出中断异常。

interrupted

查询当前线程的中断状态， 并且清除原状态。如果一个线程被中断了， 第一次调用 interrupted 则返回 true， 第二次和后面的就返回 false 了。

isInterrupted

仅仅是查询当前线程的中断状态

## 49、为什么 wait 和notify 方法要在同步块中调用？

Java API 强制要求这样做， 如果你不这么做， 你的代码会抛出 IllegalMonitorStateException 异常。还有一个原因是为了避免 wait 和 notify 之间产生竞态条件。

## 50、为什么你应该在循环中检查等待条件?

处于等待状态的线程可能会收到错误警报和伪唤醒， 如果不在循环中检查等待条件， 程序就会在没有满足结束条件的情况下退出。

## 51、Java 中的同步集合与并发集合有什么区别？

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合， 不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用， 阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

## 52、什么是线程池？ 为什么要使用它？

创建线程要花费昂贵的资源和时间， 如果任务来了才创建线程那么响应时间会变长， 而且一个进程能创建的线程数有限。为了避免这些问题， 在程序启动的时候就创建若干线程来响应处理， 它们被称为线程池， 里面的线程叫工作线程。从JDK1.5 开始， Java API 提供了 Executor 框架让你可以创建不同的线程池。

## 53、怎么检测一个线程是否拥有锁？

在 java.lang.Thread 中有一个方法叫 holdsLock()， 它返回 true 如果当且仅当当前线程拥有某个具体对象的锁。

## 54、你如何在 Java 中获取线程堆栈？

kill -3 [java pid]
不会在当前终端输出， 它会输出到代码执行的或指定的地方去。比如， kill -3 tomcat pid, 输出堆栈到 log 目录下。
Jstack [java pid]
这个比较简单， 在当前终端显示， 也可以重定向到指定文件中。
-JvisualVM： Thread Dump
不做说明， 打开 JvisualVM 后， 都是界面操作， 过程还是很简单的。55、
JVM 中哪个参数是用来控制线程的栈堆栈小的?
-Xss 每个线程的栈大小

## 56、Thread 类中的yield 方法有什么作用？

使当前线程从执行状态（ 运行状态） 变为可执行态（ 就绪状态）。

当前线程到了就绪状态， 那么接下来哪个线程会从就绪状态变成执行状态呢？ 可能是当前线程， 也可能是其他线程， 看系统的分配了。

## 57、Java 中 ConcurrentHashMap 的并发度是什么？

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的， 它是 ConcurrentHashMap 类构造函数的一个可选参数， 默认值为 16， 这样在多线程情况下就能避免争用。

在 JDK8 后， 它摒弃了 Segment（ 锁段） 的概念， 而是启用了一种全新的方式实现,利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

## 58、Java 中 Semaphore 是什么？

Java 中的 Semaphore 是一种新的同步类， 它是一个计数信号。从概念上讲， 从概念上讲，信号量维护了一个许可集合。如有必要， 在许可可用前会阻塞每一个acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数， 并采取相应的行动。信号量常常用于多线程的代码中， 比如数据库连接池。

## 59、Java 线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是 void，它定义在Executor 接口中。

而 submit()方法可以返回持有计算结果的 Future 对象， 它定义在 ExecutorService 接口中， 它扩展了 Executor 接口， 其它线程池类像 ThreadPoolExecutor 和 ScheduledThreadPoolExecutor 都有这些方法。

## 60、什么是阻塞式方法？

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情， ServerSocket 的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起， 直到得到结果之后才会返回。此外， 还有异步和非阻塞式方法在任务完成前就返回。

## 61、Java 中的 ReadWriteLock 是什么？

读写锁是用来提升并发程序性能的锁分离技术的成果。

## 62、volatile 变量和 atomic 变量有什么不同？

Volatile 变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用 volatile 修饰 count 变量那么 count++ 操作就不是原子性的。

而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 getAndIncrement()方法会原子性的进行增量操作把当前值加一， 其它数据类型和引用变量也可以进行相似操作。

## 63、可以直接调用 Thread 类的run ()方法么？

当然可以。但是如果我们调用了 Thread 的 run()方法， 它的行为就会和普通的方法一样，会在当前线程中执行。为了在新的线程中执行我们的代码， 必须使用 Thread.start()方法。

## 64、如何让正在运行的线程暂停一段时间？

我们可以使用 Thread 类的 Sleep()方法让线程暂停一段时间。需要注意的是， 这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为 Runnable， 并且根据线程调度，它将得到执行。

## 65、你对线程优先级的理解是什么？

每一个线程都是有优先级的， 一般来说， 高优先级的线程在运行时会具有优先权， 但这依赖于线程调度的实现， 这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级， 但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从 1-10)， 1 代表最低优先级， 10 代表最高优先级。

java 的线程优先级调度会委托给操作系统去处理， 所以与具体的操作系统优先级有关，
如非特别需要， 一般无需设置线程优先级。

## 66、什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing )？

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它， 它的执行便依赖于线程调度器的实现。
同上一个问题， 线程调度并不受到 Java 虚拟机控制， 所以由应用程序来控制它是更好的选择（ 也就是说不要让你的程序依赖于线程的优先级）。

时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。

## 67、你如何确保 main()方法所在的线程是Java 程序最后结束的线程？

我们可以使用 Thread 类的 join()方法来确保所有程序创建的线程在 main()方法退出前结束。

## 68、线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object 类中 wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。

## 69、为什么线程通信的方法 wait(), notify()和notifyAll()被定义在 Object 类里？

Java 的每个对象中都有一个锁(monitor，也可以成为监视器) 并且 wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在 Java 的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是 Object 类的一部分，这样 Java 的每一个类都有用于线程间通信的基本方法。

## 70、为什么 wait(), notify()和notifyAll ()必须在同步方法或者同步块中被调用？

当一个线程需要调用对象的 wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的 notify() 方法。同样的，当一个线程需要调用对象的 notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

## 71、为什么 Thread 类的sleep()和 yield ()方法是静态的？

Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作， 并避免程序员错误的认为可以在其他非运行线程调用这些方法。

## 72、如何确保线程安全？

在 Java 中可以有很多方法来保证线程安全—— 同步， 使用原子类(atomic concurrent classes)， 实现并发锁， 使用 volatile 关键字， 使用不变类和线程安全类。

## 73、同步方法和同步块，哪个是更好的选择？

同步块是更好的选择， 因为它不会锁住整个对象（ 当然你也可以让它锁住整个对象）。同步方法会锁住整个对象， 哪怕这个类中有多个不相关联的同步块， 这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则， 只在需要锁住的代码块锁住相应的对象， 这样从侧面来说也可以避免死锁。

## 74、如何创建守护线程？

使用 Thread 类的 setDaemon(true)方法可以将线程设置为守护线程， 需要注意的是， 需要在调用 start()方法前调用这个方法， 否则会抛出

IllegalThreadStateException 异常。

## 75、什么是 Java Timer 类？如何创建一个有特定时间间隔的任务？

java.util.Timer 是一个工具类， 可以用于安排一个线程在未来的某个特定时间执行。Timer 类可以用安排一次性任务或者周期任务。
java.util.TimerTask 是一个实现了 Runnable 接口的抽象类， 我们需要去继承这个类来创建我们自己的定时任务并使用 Timer 去安排它的执行。

# Java核心专题-Java并发编程（二）

## 1、并发编程三要素？

1、原子性
原子性指的是一个或者多个操作， 要么全部执行并且在执行的过程中不被其他操作打断， 要么就全部都不执行。

2、可见性
可见性指多个线程操作一个共享变量时， 其中一个线程对变量进行修改后， 其他线程可以立即看到修改的结果。

3、有序性
有序性， 即程序的执行顺序按照代码的先后顺序来执行。

## 2、实现可见性的方法有哪些？

synchronized 或者 Lock：保证同一个时刻只有一个线程获取锁执行代码，锁释放之前把最新的值刷新到主内存， 实现可见性。

## 3、多线程的价值？

1、发挥多核 CPU 的优势
多线程，可以真正发挥出多核 CPU 的优势来，达到充分利用 CPU 的目的，采用多线程的方式去同时完成几件事情而不互相干扰。

2、防止阻塞
从程序运行效率的角度来看， 单核 CPU 不但不会发挥出多线程的优势， 反而会因为在单核 CPU 上运行多线程导致线程上下文的切换， 而降低程序整体的效率。但是单核 CPU 我们还是要应用多线程，就是为了防止阻塞。试想，如果单核 CPU 使用单线程， 那么只要这个线程阻塞了， 比方说远程读取某个数据吧， 对端迟迟未返回又没有设置超时时间， 那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题， 多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞， 也不会影响其它任务的执行。

3、便于建模
这是另外一个没有这么明显的优点了。假设有一个大的任务 A，单线程编程，那么就要考虑很多， 建立整个程序模型比较麻烦。但是如果把这个大的任务 A 分解成几个小任务，任务 B、任务 C、任务 D，分别建立程序模型，并通过多线程分别运行这几个任务， 那就简单很多了。

## 4、创建线程的有哪些方式？

1、继承 Thread 类创建线程类
2、通过 Runnable 接口创建线程类
3、通过 Callable 和 Future 创建线程4、通
过线程池创建

## 5、创建线程的三种方式的对比？

1、采用实现 Runnable、Callable 接口的方式创建多线程。优势是：
线程类只是实现了 Runnable 接口或 Callable 接口， 还可以继承其他类。
在这种方式下， 多个线程可以共享同一个 target 对象， 所以非常适合多个相同线程来处理同一份资源的情况， 从而可以将 CPU、代码和数据分开， 形成清晰的模型， 较好地体现了面向对象的思想。

劣势是：
编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread()方法。
2、使用继承 Thread 类的方式创建多线程优势
是：
编写简单， 如果需要访问当前线程， 则无需使用 Thread.currentThread()方法， 直接使用 this 即可获得当前线程。

劣势是：
线程类已经继承了 Thread 类， 所以不能再继承其他父类。
3、Runnable 和 Callable 的区别
1、Callable 规定（ 重写）的方法是 call()，Runnable 规定（ 重写）的方法是 run()。2、Callable 的任务执行后可返回值， 而 Runnable 的任务是不能返回值的。
3、Call 方法可以抛出异常， run 方法不可以。
4、运行 Callable 任务可以拿到一个 Future 对象， 表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过 Future 对象可以了解任务执行情况， 可取消任务的执行， 还可获取执行结果。

# 6、线程的状态流转图

线程的生命周期及五种基本状态：
![img_2.png][img_2.png]

# 7、Java 线程具有五中基本状态

1、新建状态（ New）：当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread()；
2、就绪状态（ Runnable）： 当调用线程对象的 start()方法（ t.start();）， 线程即进入就绪状态。处于就绪状态的线程， 只是说明此线程已经做好了准备， 随时等待 CPU 调度执行， 并不是说执行了 t.start()此线程立即就会执行；

3、运行状态（ Running）： 当 CPU 开始调度处于就绪状态的线程时， 此时线程才得以真正执行， 即进入到运行状态。注： 就 绪状态是进入到运行状态的唯一入口， 也就是说， 线程要想进入运行状态执行， 首先必须处于就绪状态中；

4、阻塞状态（ Blocked）：处于运行状态中的线程由于某种原因，暂时放弃对 CPU 的使用权， 停止执行， 此时进入阻塞状态， 直到其进入到就绪状态， 才 有机会再次被 CPU 调用以进入到运行状态。

根据阻塞产生的原因不同， 阻塞状态又可以分为三种：
1、等待阻塞：运行状态中的线程执行 wait()方法，使本线程进入到等待阻塞状态；
2、同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)， 它会进入同步阻塞状态；
3、其他阻塞：通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时， 线程重新转入就绪状态。

5、死亡状态（ Dead）：线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

## 8、什么是线程池？有哪几种创建方式？

线程池就是提前创建若干个线程， 如果有任务需要处理， 线程池里的线程就会处理任务，处理完之后线程并不会被销毁， 而是等待下一个任务。由于创建和销毁线程都是消耗系统资源的， 所以当你想要频繁的创建和销毁线程的时候就可以考虑使用线程池来提升系统的性能。

java 提供了一个 java.util.concurrent.Executor 接口的实现用于创建线程池。

## 9、四种线程池的创建：

1、newCachedThreadPool 创建一个可缓存线程池
2、newFixedThreadPool 创建一个定长线程池， 可控制线程最大并发数。
3、newScheduledThreadPool 创建一个定长线程池， 支持定时及周期性任务执行。
4、newSingleThreadExecutor 创建一个单线程化的线程池， 它只会用唯一的工作线程来执行任务。

## 10、线程池的优点？

1、重用存在的线程， 减少对象创建销毁的开销。
2、可有效的控制最大并发线程数， 提高系统资源的使用率，同时避免过多资源竞争， 避免堵塞。
3、提供定时执行、定期执行、单线程、并发数控制等功能。

## 11、常用的并发工具类有哪些？

1、CountDownLatch
2、CyclicBarrier
3、Semaphore
4、Exchanger

## 12、CyclicBarrier 和 CountDownLatch 的区别

1、CountDownLatch 简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用 countDown()方法发出通知后， 当前线程才可以继续执行。

2、cyclicBarrier 是所有线程都进行等待， 直到所有线程都准备好进入 await()方法之后，所有线程同时开始执行！
3、CountDownLatch 的计数器只能使用一次。而 CyclicBarrier 的计数器可以使用 reset() 方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景， 比如如果计算发生错误， 可以重置计数器， 并让线程们重新执行一次。
4、CyclicBarrier 还提供其他有用的方法， 比如 getNumberWaiting 方法可以获得 CyclicBarrier 阻塞的线程数量。isBroken 方法用来知道阻塞的线程是否被中断。如果被中断返回 true， 否则返回 false。

## 13、synchronized 的作用？

在 Java 中， synchronized 关键字是用来控制线程同步的， 就是在多线程的环境下， 控制 synchronized 代码段不被多个线程同时执行。
synchronized 既可以加在一段代码上， 也可以加在方法上。

## 14、volatile 关键字的作用

对于可见性， Java 提供了 volatile 关键字来保证可见性。
当一个共享变量被 volatile 修饰时， 它会保证修改的值会立即被更新到主存， 当有其他线程需要读取时， 它会去内存中读取新值。
从实践角度而言， volatile 的一个重要作用就是和 CAS 结合， 保证了原子性， 详细的可以参见 java.util.concurrent.atomic 包下的类， 比如 AtomicInteger。

## 15、什么是 CAS

CAS 是 compare and swap 的缩写， 即我们所说的比较交换。
cas 是一种基于锁的操作， 而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住， 等一个之前获得锁的线程释放锁之后， 下一个线程才可以访

问。而乐观锁采取了一种宽泛的态度， 通过某种方式不加锁来处理资源， 比如通过给记录加 version 来获取数据， 性能较悲观锁有很大的提高。
CAS 操作包含三个操作数 —— 内存位置（ V）、预期原值（ A） 和新值(B)。如果内存地址里面的值和 A 的值是一样的， 那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了， 那么 a 线程需要自旋， 到下次循环才有可能机会执行。
java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的( AtomicInteger,AtomicBoolean,AtomicLong)。

# 16、CAS 的问题

1、CAS 容易造成 ABA 问题
一个线程 a 将数值改成了 b， 接着又改成了 a， 此时 CAS 认为是没有变化， 其实是已经变化过了， 而这个问题的解决方案可以使用版本号标识， 每操作一次

version 加 1。在 java5 中，已经提供了 AtomicStampedReference 来解决问题。
2、不能保证代码块的原子性
CAS 机制所保证的知识一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新， 就不得不使用 synchronized 了。3、CAS 造成 CPU 利用率增加
之前说过了 CAS 里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu 资源会一直被占用。

# 17、什么是 Future？

在并发编程中， 我们经常用到非阻塞的模型， 在之前的多线程的三种实现中， 不管是继承 thread 类还是实现 runnable 接口，都无法保证获取到之前的执行结果。通过实现 Callback 接口， 并用 Future 可以来接收多线程的执行结果。
Future 表示一个可能还没有完成的异步任务的结果， 针对这个结果可以添加Callback 以便在任务执行成功或失败后作出相应的操作。

## 18、什么是 AQS

AQS 是 AbustactQueuedSynchronizer 的简称， 它是一个 Java 提高的底层同步工具类，用一个 int 类型的变量表示同步状态， 并提供了一系列的 CAS 操作来管理这个同步状态。AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器， 比如我们提到的 ReentrantLock， Semaphore， 其他的诸如 ReentrantReadWriteLock， SynchronousQueue， FutureTask 等等皆是基于AQS 的。

## 19、AQS 支持两种同步方式：

1、独占式
2、共享式
这样方便使用者实现不同类型的同步组件， 独占式如 ReentrantLock， 共享式如 Semaphore，CountDownLatch，组合式的如 ReentrantReadWriteLock。总之， AQS 为使用提供了底层支撑， 如何组装实现， 使用者可以自由发挥。

## 20、ReadWriteLock 是什么

首先明确一下，不是说 ReentrantLock 不好，只是 ReentrantLock 某些时候有局限。如果使用 ReentrantLock， 可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致， 但这样， 如果线程 C 在读数据、线程 D 也在读数据， 读数据是不会改变数据的， 没有必要加锁， 但是还是加锁了， 降低了程序的性能。因为这个，才诞生了读写锁 ReadWriteLock。ReadWriteLock 是一个读写锁接口，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写

的分离， 读锁是共享的， 写锁是独占的， 读和读之间不会互斥， 读和写、写和读、写和写之间才会互斥， 提升了读写的性能。

## 21、FutureTask 是什么

这个其实前面有提到过，FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类， 可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然， 由于 FutureTask 也是
Runnable 接口的实现类， 所以 FutureTask 也可以放入线程池中。

## 22、synchronized 和ReentrantLock 的区别

synchronized 是和 if、else、for、while 一样的关键字， ReentrantLock 是类， 这是二者的本质区别。既然 ReentrantLock 是类， 那么它就提供了比
synchronized 更多更灵活的特性， 可以被继承、可以有方法、可以有各种各样的类变量，
ReentrantLock 比 synchronized 的扩展性体现在几点上：
1、ReentrantLock 可以对获取锁的等待时间进行设置， 这样就避免了死锁2、
ReentrantLock 可以获取各种锁的信息
3、ReentrantLock 可以灵活地实现多路通知
另外，二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的park 方法加锁， synchronized 操作的应该是对象头中 mark word， 这点我不能确定。

## 23、什么是乐观锁和悲观锁

1、乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态， 乐观锁认为竞争不总是会发生， 因此它不需要持有锁， 将比较-替换这两个动作作

为一个原子操作尝试去修改内存中的变量， 如果失败则表示发生冲突， 那么就应该有相应的重试逻辑。

2、悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态， 悲观锁认为竞争总是会发生， 因此每次对某资源进行操作时， 都会持有一个独占的锁， 就像 synchronized， 不管三七二十一， 直接上了锁就操作资源了。

## 24、线程 B 怎么知道线程A 修改了变量

1、volatile 修饰变量
2、synchronized 修饰修改变量的方法3、
wait/notify
4、while 轮询

## 25、synchronized、volatile、CAS 比较

1、synchronized 是悲观锁， 属于抢占式， 会引起其他线程阻塞。2、
volatile 提供多线程共享变量可见性和禁止指令重排序优化。

3、CAS 是基于冲突检测的乐观锁（ 非阻塞）

## 26、sleep 方法和 wait 方法有什么区别?

这个问题常问，sleep 方法和 wait 方法都可以用来放弃 CPU 一定的时间，不同点在于如果线程持有某个对象的监视器， sleep 方法不会放弃这个对象的监视器， wait 方法会放弃这个对象的监视器

## 27、ThreadLocal 是什么？有什么用？

ThreadLocal 是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射， 各个线程之间的变量互不干扰， 在高并发场景下， 可以实现无状态的调用， 特别适用于各个线程依赖不通的变量值完成操作的场景。简单说 ThreadLocal 就是一种以空间换时间的做法， 在每个 Thread 里面维护了一个以开地址法实现的 ThreadLocal.ThreadLocalMap， 把数据进行隔离， 数据不共享， 自然就没有线程安全方面的问题了。

## 28、为什么 wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是 JDK 强制的，wait()方法和 notify()/notifyAll()方法在调用前都必须先获得对象的锁

## 29、多线程同步有哪几种方法？

Synchronized 关键字， Lock 锁实现， 分布式锁等。

## 30、线程的调度策略

线程调度器选择优先级最高的线程运行， 但是， 如果发生以下情况， 就会终止线程的运行：
1、线程体中调用了 yield 方法让出了对 cpu 的占用权利
2、线程体中调用了 sleep 方法使线程进入睡眠状态
3、线程由于 IO 操作受到阻塞
4、另外一个更高优先级线程出现
5） 在支持时间片的系统中， 该线程的时间片用完

## 31、ConcurrentHashMap 的并发度是什么

ConcurrentHashMap 的并发度就是 segment 的大小， 默认为 16， 这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap， 这也是 ConcurrentHashMap 对 Hashtable 的最大优势， 任何情况下， Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

## 32、Linux 环境下如何查找哪个线程使用CPU 最长

1、获取项目的 pid， jps 或者 ps -ef | grep java， 这个前面有讲过2、top -H -p pid， 顺序不能改变

## 33、Java 死锁以及如何避免？

Java 中的死锁是一种编程情况，其中两个或多个线程被永久阻塞，Java 死锁情况出现至少两个线程和两个或更多资源。

Java 发生死锁的根本原因是： 在申请锁时发生了交叉闭环申请。

## 34、死锁的原因

1、是多个线程涉及到多个锁，这些锁存在着交叉，所以可能会导致了一个锁依赖 的闭环。

例如： 线程在获得了锁 A 并且没有释放的情况下去申请锁 B， 这时， 另一个线程已经获得了锁 B，在释放锁 B 之前又要先获得锁 A，因此闭环发生，陷入死锁循环。
2、默认的锁申请操作是阻塞的。

所以要避免死锁， 就要在一遇到多个对象锁交叉的情况， 就要仔细审查这几个对象的类中的所有方法， 是否存在着导致锁依赖的环路的可能性。总之是尽量避免在一个同步方法中调用其它对象的延时方法和同步方法。

## 35、怎么唤醒一个阻塞的线程

如果线程是因为调用了 wait()、sleep()或者 join()方法而导致的阻塞，可以中断线程， 并且通过抛出 InterruptedException 来唤醒它； 如果线程遇到了 IO 阻塞， 无能为力，因为 IO 是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

## 36、不可变对象对多线程有什么帮助

前面有提到过的一个问题， 不可变对象保证了对象的内存可见性， 对不可变对象的读取不需要进行额外的同步手段， 提升了代码执行效率。

## 37、什么是多线程的上下文切换

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 CPU 执行权的线程的过程。

## 38、如果你提交任务时，线程池队列已满，这时会发生什么

这里区分一下：
1、如果使用的是无界队列 LinkedBlockingQueue， 也就是无界队列的话， 没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列， 可以无限存放任务

2、如果使用的是有界队列比如 ArrayBlockingQueue， 任务首先会被添加到 ArrayBlockingQueue 中， ArrayBlockingQueue 满了， 会根据 maximumPoolSize 的值增加线程数量， 如果增加了线程数量还是处理不过来， ArrayBlockingQueue 继续满， 那么则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务， 默认是 AbortPolicy

## 39、Java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后， 操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

## 40、什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它， 它的执行便依赖于线程调度器的实现。时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。线程调度并不受到 Java 虚拟机控制， 所以由应用程序来控制它是更好的选择（ 也就是说不要让你的程序依赖于线程的优先级）。

## 41、什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作， 因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线

程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁， 再阻塞， 这样可能是一种更好的策略。

## 42、Java Concurrency API 中的Lock 接口(Lock interface) 是什么？对比同步它有什么优势？

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构， 可以具有完全不同的性质， 并且可以支持多个相关类的条件对象。
它的优势有：
1、可以使锁更公平
2、可以使线程在等待锁的时候响应中断
3、可以让线程尝试获取锁， 并在无法获取锁的时候立即返回或者等待一段时间
4、可以在不同的范围， 以不同的顺序获取和释放锁

## 43、单例模式的线程安全性

老生常谈的问题了， 首先要说的是单例模式的线程安全意味着： 某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法， 我总结一下：
1、饿汉式单例模式的写法： 线程安全
2、懒汉式单例模式的写法： 非线程安全
3、双检锁单例模式的写法： 线程安全

## 44、Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个

线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1， 相当于变成了一个 synchronized 了。

## 45、Executors 类是什么？

Executors 为 Executor， ExecutorService， ScheduledExecutorService， ThreadFactory 和 Callable 类提供了一些工具方法。
Executors 可以用于方便的创建线程池

## 46、线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住： 线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的， 而 run 方法里面的代码才是被线程自身所调用的。
如果说上面的说法让你感到困惑， 那么我举个例子， 假设 Thread2 中 new 了 Thread1， main 函数中 new 了 Thread2， 那么：
1、Thread2 的构造方法、静态块是 main 线程调用的， Thread2 的 run()方法是 Thread2 自己调用的
2、Thread1 的构造方法、静态块是 Thread2 调用的， Thread1 的 run()方法是 Thread1 自己调用的

## 47、同步方法和同步块，哪个是更好的选择?

同步块， 这意味着同步块之外的代码是异步执行的， 这比同步整个方法更提升代码的效率。请知道一条原则： 同步的范围越小越好。

## 48、Java 线程数过多会造成什么异常？

1、线程的生命周期开销非常高

2、消耗过多的 CPU 资源
如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争 CPU 资源时还将产生其他性能的开销。

3、降低稳定性
JVM 在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同，并且承受着多个因素制约，包括 JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出OutOfMemoryError 异常。

# 性能调优专题-MySQL面试题

## 1、MySQL 中有哪几种锁？

1、表级锁： 开销小， 加锁快； 不会出现死锁； 锁定粒度大， 发生锁冲突的概率最高，并发度最低。

2、行级锁： 开销大， 加锁慢； 会出现死锁； 锁定粒度最小， 发生锁冲突的概率最低，并发度也最高。

3、页面锁： 开销和加锁时间界于表锁和行锁之间； 会出现死锁； 锁定粒度界于表锁和行锁之间， 并发度一般。

## 2、MySQL 中有哪些不同的表格？

共有 5 种类型的表格：

1、MyISAM

2、 Heap

3、 Merge

4、INNODB

5、ISAM

## 3、简述在MySQL 数据库中 MyISAM 和InnoDB 的区别

MyISAM：

不支持事务， 但是每次查询都是原子的； 支持

表级锁， 即每次操作是对整个表加锁； 存储表

的总行数；

一个 MYISAM 表有三个文件： 索引文件、表结构文件、数据文件；

采用菲聚集索引， 索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。

InnoDb：

支持 ACID 的事务， 支持事务的四种隔离级别； 支持

行级锁及外键约束： 因此可以支持写并发； 不存储

总行数：

一个 InnoDb 引擎存储在一个文件空间（ 共享表空间， 表大小不受操作系统控制，一个表可能分布在多个文件里）， 也有可能为多个（ 设置为独立表空， 表大小受操作系统文件大小限制， 一般为 2G）， 受操作系统文件大小的限制；

主键索引采用聚集索引（ 索引的数据域存储数据文件本身）， 辅索引的数据域存储主键的值； 因此从辅索引查找数据， 需要先通过辅索引找到主键值， 再访问辅索引； 最好使用自增主键， 防止插入数据时， 为维持 B+树结构， 文件的大调整。

## 4、MySQL 中InnoDB 支持的四种事务隔离级别名称，以及逐级之间的区别？

SQL 标准定义的四个隔离级别为：

1、read uncommited ： 读到未提交数据2、

read committed： 脏读， 不可重复读3、

repeatable read： 可重读

4、serializable ： 串行事物

## 5、CHAR 和VARCHAR 的区别？

1、CHAR 和 VARCHAR 类型在存储和检索方面有所不同

2、CHAR 列长度固定为创建表时声明的长度， 长度值范围是 1 到 255 当 CHAR 值被存储时，它们被用空格填充到特定长度， 检索 CHAR 值时需删除尾随空格。

## 6、主键和候选键有什么区别？

表格的每一行都由主键唯一标识,一个表只有一个主键。

主键也是候选键。按照惯例， 候选键可以被指定为主键， 并且可以用于任何外键引用。

## 7、myisamchk 是用来做什么的？

它用来压缩 MyISAM 表， 这减少了磁盘或内存使用。

MyISAM Static 和 MyISAM Dynamic 有什么区别？

在 MyISAM Static 上的所有字段有固定宽度。动态 MyISAM 表将具有像 TEXT， BLOB 等字段， 以适应不同长度的数据类型。

MyISAM Static 在受损情况下更容易恢复。

## 8、如果一个表有一列定义为TIMESTAMP，将发生什么？

每当行被更改时， 时间戳字段将获取当前时间戳。

列设置为 AUTO INCREMENT 时， 如果在表中达到最大值， 会发生什么情况？

它会停止递增， 任何进一步的插入都将产生错误， 因为密钥已被使用。

怎样才能找出最后一次插入时分配了哪个自动增量？

LAST_INSERT_ID 将返回由 Auto_increment 分配的最后一个值， 并且不需要指定表名称。

## 9、你怎么看到为表格定义的所有索引？

索引是通过以下方式为表格定义的：

SHOW INDEX FROM <tablename>;

## 10、LIKE 声明中的%和_是什么意思？

% 对应于 0 个或更多字符，_只是 LIKE 语句中的一个字符。

如何在 Unix 和 MySQL 时间戳之间进行转换？

UNIX_TIMESTAMP 是从 MySQL 时间戳转换为 Unix 时间戳的命令
FROM_UNIXTIME 是从 Unix 时间戳转换为 MySQL 时间戳的命令

## 11、列对比运算符是什么？

在 SELECT 语句的列比较中使用=，<>，<=，<，> =，>，<<，>>，<=>，AND， OR 或 LIKE 运算符。

## 12、BLOB 和TEXT 有什么区别？

BLOB 是一个二进制对象， 可以容纳可变数量的数据。TEXT 是一个不区分大小写的 BLOB。

BLOB 和 TEXT 类型之间的唯一区别在于对 BLOB 值进行排序和比较时区分大小写， 对 TEXT 值不区分大小写。

## 13、MySQL_fetch_array 和MySQL_fetch_object 的区别是什么？

以下是 MySQL_fetch_array 和 MySQL_fetch_object 的区别：

MySQL_fetch_array（） – 将结果行作为关联数组或来自数据库的常规数组返回。

MySQL_fetch_object – 从数据库返回结果行作为对象。

## 14、MyISAM 表格将在哪里存储，并且还提供其存储格式？

每个 MyISAM 表格以三种格式存储在磁盘上：

·“.frm”文件存储表定义

·数据文件具有“.MYD”（MYData）扩展名

索引文件具有“.MYI”（MYIndex）扩展名

## 15、MySQL 如何优化DISTINCT？

DISTINCT 在所有列上转换为 GROUP BY， 并与 ORDER BY 子句结合使用。SELECT

DISTINCT t1.a FROM t1,t2 where t1.a=t2.a;

## 16、如何显示前 50 行？

在 MySQL 中， 使用以下代码查询显示前 50 行：

SELECT*FROM

LIMIT 0,50;

## 17、可以使用多少列创建索引？

任何标准表最多可以创建 16 个索引列。

## 18、NOW（）和 CURRENT_DATE（）有什么区别？

NOW（） 命令用于显示当前年份， 月份， 日期， 小时， 分钟和秒。

CURRENT_DATE（） 仅显示当前年份， 月份和日期。

## 19、什么是非标准字符串类型？

1、TINYTEXT

2、TEXT

3、MEDIUMTEXT

4、LONGTEXT

## 20、什么是通用 SQL 函数？

1、CONCAT(A, B) – 连接两个字符串值以创建单个字符串输出。通常用于将两个或多个字段合并为一个字段。

2、FORMAT(X, D)- 格式化数字 X 到 D 有效数字。

3、CURRDATE(), CURRTIME()- 返回当前日期或时间。

4、NOW（） – 将当前日期和时间作为一个值返回。

5、MONTH（）， DAY（）， YEAR（）， WEEK（）， WEEKDAY（） – 从日期值中提取给定数据。

6、HOUR（）， MINUTE（）， SECOND（） – 从时间值中提取给定数据。

7、DATEDIFF（ A， B） – 确定两个日期之间的差异， 通常用于计算年龄

8、SUBTIMES（ A， B） – 确定两次之间的差异。

9、FROMDAYS（ INT） – 将整数天数转换为日期值。

## 21、MySQL 支持事务吗？

在缺省模式下，MySQL 是 autocommit 模式的，所有的数据库更新操作都会即时提交， 所以在缺省情况下， MySQL 是不支持事务的。

但是如果你的 MySQL 表类型是使用 InnoDB Tables 或 BDB tables 的话， 你的MySQL 就可以使用事务处理,使用 SET
AUTOCOMMIT=0 就可以使 MySQL 允许在非 autocommit 模式， 在非
autocommit 模式下，你必须使用 COMMIT 来提交你的更改，或者用 ROLLBACK 来回滚你的更改。

## 22、MySQL 里记录货币用什么字段类型好

NUMERIC 和 DECIMAL 类型被 MySQL 实现为同样的类型， 这在 SQL92 标准允许。他们被用于保存值， 该值的准确精度是极其重要的值， 例如与金钱有关的数据。当声明一个类是这些类型之一时， 精度和规模的能被(并且通常是)指定。

例如：

salary DECIMAL(9,2)

在这个例子中， 9(precision)代表将被用于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。

因此， 在这种情况下， 能被存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

## 23、MySQL 有关权限的表都有哪几个？

MySQL 服务器通过权限表来控制用户对数据库的访问， 权限表存放在 MySQL 数据库里，由 MySQL_install_db 脚本初始化。这些权限表分别 user，db，table_priv，columns_priv 和 host 。

## 24、列的字符串类型可以是什么？

字符串类型是：

1、SET

2、BLOB

3、ENUM

4、CHAR

5、TEXT

## 25、MySQL 数据库作发布系统的存储，一天五万条以上的增量，预计运维三年,怎么优化？

1、设计良好的数据库结构，允许部分数据冗余，尽量避免 join 查询，提高效率。

2、选择合适的表字段数据类型和存储引擎，适当的添加索引。

3、MySQL 库主从读写分离。

4、找规律分表，减少单表中的数据量提高查询速度。5、

添加缓存机制，比如 memcached，apc 等。

6、不经常改动的页面，生成静态页面。

7、书写高效率的 SQL。比如 SELECT * FROM TABEL 改为 SELECT field_1, field_2, field_3 FROM TABLE.

## 26、锁的优化策略

1、读写分离

2、分段加锁

3、减少锁持有的时间

4. 多个线程尽量以相同的顺序去获取资源

不能将锁的粒度过于细化， 不然可能会出现线程的加锁和释放次数过多， 反而效率不如一次加一把大锁。

## 27、索引的底层实现原理和优化

B+树， 经过优化的 B+树

主要是在所有的叶子结点中增加了指向下一个叶子节点的指针， 因此 InnoDB 建议为大部分表使用默认自增的主键作为主索引。

## 28、什么情况下设置了索引但无法使用

1、以“%”开头的 LIKE 语句，模糊匹配

2、OR 语句前后没有同时使用索引

3、数据类型出现隐式转化（ 如 varchar 不加单引号的话可能会自动转换为 int 型）

## 29、实践中如何优化 MySQL

最好是按照以下顺序优化： 1、

SQL 语句及索引的优化

2、数据库表结构的优化

3、系统配置的优化

4、硬件的优化

详细可以查看 阿里 P8 架构师谈：MySQL 慢查询优化、索引优化、以及表等优化结

## 30、优化数据库的方法

1、选取最适用的字段属性，尽可能减少定义字段宽度，尽量把字段设置 NOTNULL， 例如“省份”、“性别”最好适用 ENUM

2、使用连接(JOIN)来代替子查询

3、适用联合(UNION)来代替手动创建的临时表
4、事务处理

5、锁定表、优化事务处理

6、适用外键， 优化锁定表

7、建立索引

8、优化查询语句

## 31、简单描述 MySQL 中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）

索引是一种特殊的文件(InnoDB 数据表上的索引是表空间的一个组成部分)， 它们包含着对数据表里所有记录的引用指针。

普通索引(由关键字 KEY 或 INDEX 定义的索引)的唯一任务是加快对数据的访问速度。

普通索引允许被索引的数据列包含重复的值。如果能确定某个数据列将只包含彼此各不相同的值， 在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。也就是说， 唯一索引可以保证数据记录的唯一性。

主键， 是一种特殊的唯一索引， 在一张表中只能定义一个主键索引， 主键用于唯一标识一条记录， 使用关键字 PRIMARY KEY 来创建。

索引可以覆盖多个数据列，如像 INDEX(columnA, columnB)索引，这就是联合索引。

索引可以极大的提高数据的查询速度， 但是会降低插入、删除、更新表的速度， 因为在执行这些写操作时， 还要操作索引文件。

## 32、数据库中的事务是什么?

事务（ transaction） 是作为一个单元的一组有序的数据库操作。如果组中的所有操作都成功， 则认为事务成功， 即使只有一个操作失败， 事务也不成功。如果所

有操作完成， 事务则提交， 其修改将作用于所有其他数据库进程。如果一个操作失败，则事务将回滚， 该事务所有操作的影响都将取消。

事务特性：

1、原子性： 即不可分割性， 事务要么全部被执行， 要么就全部不被执行。

2、一致性或可串性。事务的执行使得数据库从一种正确状态转换成另一种正确状 态

3、隔离性。在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何 其他事务，

4、持久性。事务正确提交后， 其结果将永久保存在数据库中， 即使在事务提交后有了其他故障， 事务的处理结果也会得到保存。

或者这样理解：

事务就是被绑定在一起作为一个逻辑工作单元的 SQL 语句分组， 如果任何一个语句操作失败那么整个操作就被失败， 以后操作就会回滚到操作前状态， 或者是上有个节点。为了确保要么执行， 要么不执行， 就可以使用事务。要将有组语句作为事务考虑， 就需要通过 ACID 测试， 即原子性， 一致性， 隔离性和持久性。

## 33、SQL 注入漏洞产生的原因？如何防止？

SQL 注入产生的原因： 程序开发过程中不注意规范书写 sql 语句和对特殊字符进行过滤，导致客户端可以通过全局变量 POST 和 GET 提交一些 sql 语句正常执行。

防止 SQL 注入的方式：
开启配置文件中的 magic_quotes_gpc 和 magic_quotes_runtime 设置

执行 sql 语句时使用 addslashes 进行 sql 语句转换Sql 语句

书写尽量不要省略双引号和单引号。

过滤掉 sql 语句中的一些关键词： update、insert、delete、select、 * 。

提高数据库表和字段的命名技巧， 对一些重要的字段根据程序的特点命名， 取不易被猜到的。

## 34、为表中得字段选择合适得数据类型

字段类型优先级: 整形>date,time>enum,char>varchar>blob,text
优先考虑数字类型， 其次是日期或者二进制类型， 最后是字符串类型， 同级别得数据类型， 应该优先选择占用空间小的数据类型

## 35、存储时期

Datatime:以 YYYY-MM-DD HH:MM:SS 格式存储时期时间， 精确到秒， 占用 8 个字节得存储空间， datatime 类型与时区无关
Timestamp:以时间戳格式存储，占用 4 个字节，范围小 1970-1-1 到 2038-1-19， 显示依赖于所指定得时区， 默认在第一个列行的数据修改时可以自动得修改
timestamp 列得值
Date:（ 生日） 占用得字节数比使用字符串.datatime.int 储存要少， 使用 date 只需要 3 个字节， 存储日期月份， 还可以利用日期时间函数进行日期间得计算
Time:存储时间部分得数据
注意:不要使用字符串类型来存储日期时间数据（ 通常比字符串占用得储存空间小， 在进行查找过滤可以利用日期得函数）
使用 int 存储日期时间不如使用 timestamp 类型

## 36、对于关系型数据库而言，索引是相当重要的概念，请回答有关索引的几个问题：

1、索引的目的是什么？

快速访问数据表中的特定信息， 提高检索速度
创建唯一性索引， 保证数据库表中每一行数据的唯一性。加速
表和表之间的连接

使用分组和排序子句进行数据检索时， 可以显著减少查询中分组和排序的时间2、索引对

数据库系统的负面影响是什么？

负面影响：

创建索引和维护索引需要耗费时间， 这个时间随着数据量的增加而增加； 索引需要占用物理空间， 不光是表需要占用数据空间， 每个索引也需要占用物理空间； 当对表进行增、删、改、的时候索引也要动态维护， 这样就降低了数据的维护速度。

3、为数据表建立索引的原则有哪些？

在最频繁使用的、用以缩小查询范围的字段上建立索引。 在

频繁使用的、需要排序的字段上建立索引

4、什么情况下不宜建立索引？

对于查询中很少涉及的列或者重复值比较多的列， 不宜建立索引。对于一

些特殊的数据类型， 不宜建立索引， 比如文本字段（ text） 等

## 37、解释 MySQL 外连接、内连接与自连接的区别

先说什么是交叉连接: 交叉连接又叫笛卡尔积，它是指不使用任何条件，直接将一个表的所有记录和另一个表中的所有记录一一匹配。

内连接 则是只有条件的交叉连接，根据某个条件筛选出符合条件的记录，不符合条件的记录不会出现在结果集中， 即内连接只连接匹配的行。
外连接 其结果集中不仅包含符合连接条件的行，而且还会包括左表、右表或两个表中的所有数据行， 这三种情况依次称之为左外连接， 右外连接， 和全外连接。

左外连接， 也称左连接， 左表为主表， 左表中的所有记录都会出现在结果集中， 对于那些在右表中并没有匹配的记录， 仍然要显示， 右边对应的那些字段值以NULL 来填充。右外连接，也称右连接，右表为主表，右表中的所有记录都会出现在结果集中。左连接和右连接可以互换， MySQL 目前还不支持全外连接。

## 38、Myql 中的事务回滚机制概述

事务是用户定义的一个数据库操作序列， 这些操作要么全做要么全不做， 是一个不可分割的工作单位， 事务回滚是指将该事务已经完成的对数据库的更新操作撤销。

要同时修改数据库中两个不同表时， 如果它们不是一个事务的话， 当第一个表修改完，可能第二个表修改过程中出现了异常而没能修改， 此时就只有第二个表依旧是未修改之前的状态， 而第一个表已经被修改完毕。而当你把它们设定为一个

事务的时候， 当第一个表修改完， 第二表修改出现异常而没能修改， 第一个表和第二个表都要回到未修改的状态， 这就是所谓的事务回滚

## 39、SQL 语言包括哪几部分？每部分都有哪些操作关键字？

SQL 语言包括数据定义(DDL)、数据操纵(DML),数据控制(DCL)和数据查询（ DQL） 四个部分。

数据定义： Create Table,Alter Table,Drop Table, Craete/Drop Index 等数据操纵：

Select ,insert,update,delete,

数据控制： grant,revoke 数

据查询： select

## 40、完整性约束包括哪些？

数据完整性(Data Integrity)是指数据的精确(Accuracy)和可靠性(Reliability)。

分为以下四类：

1、实体完整性： 规定表的每一行在表中是惟一的实体。

2、域完整性：是指表中的列必须满足某种特定的数据类型约束，其中约束又包括 取值范围、精度等规定。

3、参照完整性： 是指两个表的主关键字和外关键字的数据应一致，保证了表之间的数据的一致性， 防止了数据丢失或无意义的数据在数据库中扩散。

4、用户定义的完整性：不同的关系数据库系统根据其应用环境的不同，往往还需 要一些特殊的约束条件。用户定义的完整性即是针对某个特定关系数据库的约束条件， 它反映某一具体应用必须满足的语义要求。

与表有关的约束：包括列约束(NOT NULL（ 非空约束）)和表约束(PRIMARY KEY、foreign key、check、UNIQUE) 。

## 41、什么是锁？

答： 数据库是一个多用户使用的共享资源。当多个用户并发地存取数据时， 在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可 能会读取和存储不正确的数据， 破坏数据库的一致性。

加锁是实现数据库并发控制的一个非常重要的技术。当事务在对某个数据对象进 行操作前， 先向系统发出请求， 对其加锁。加锁后事务就对该数据对象有了一定的控制， 在该事务释放锁之前， 其他的事务不能对此数据对象进行更新操作。

基本锁类型： 锁包括行级锁和表级锁

## 42、什么叫视图？游标是什么？

答： 视图是一种虚拟的表， 具有和物理表相同的功能。可以对视图进行增， 改， 查，操作， 视图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易， 相比多表查询。

游标： 是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标， 但是需要逐条处理数据的时候， 游标显得十分重要。

## 43、什么是存储过程？用什么来调用？

答： 存储过程是一个预编译的 SQL 语句， 优点是允许模块化的设计， 就是说只需创建一次， 以后在该程序中就可以调用多次。如果某次操作需要执行多次 SQL， 使用存储过程比单纯 SQL 语句执行要快。可以用一个命令对象来调用存储过程。

## 44、如何通俗地理解三个范式？

答：第一范式：1NF 是对属性的原子性约束，要求属性具有原子性，不可再分解； 第二范

式：2NF 是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性；

第三范式： 3NF 是对字段冗余性的约束， 即任何字段不能由其他字段派生出来， 它要求字段没有冗余。。

范式化设计优缺点:

优点:

可以尽量得减少数据冗余， 使得更新快， 体积小

缺点:对于查询需要多个表进行关联， 减少写得效率增加读得效率， 更难进行索引优化

反范式化:

优点:可以减少表得关联， 可以更好得进行索引优化

缺点:数据冗余以及数据异常， 数据得修改需要更多的成本

## 45、什么是基本表？什么是视图？

答：基本表是本身独立存在的表，在 SQL 中一个关系就对应一个表。 视图是从一个或几个基本表导出的表。视图本身不独立存储在数据库中， 是一个虚表

## 46、试述视图的优点？

答：(1) 视图能够简化用户的操作 (2) 视图使用户能以多种角度看待同一数据；(3) 视图为数据库提供了一定程度的逻辑独立性； (4) 视图能够对机密数据提供安全保护。

## 47、 NULL 是什么意思

答： NULL 这个值表示 UNKNOWN(未知):它不表示“”(空字符串)。对 NULL 这个值的任何比较都会生产一个 NULL 值。您不能把任何值与一个 NULL 值进行比较， 并在逻辑上希望获得一个答案。

使用 IS NULL 来进行 NULL 判断

## 48、主键、外键和索引的区别？

主键、外键和索引的区别

定义：

主键– 唯一标识一条记录， 不能有重复的， 不允许为空

外键– 表的外键是另一表的主键, 外键可以有重复的, 可以是空值索引–

该字段没有重复值， 但可以有一个空值

作用：

主键– 用来保证数据完整性

外键– 用来和其他表建立联系用的索

引– 是提高查询排序的速度

个数：

主键– 主键只能有一个

外键– 一个表可以有多个外键

索引– 一个表可以有多个唯一索引

## 49、你可以用什么来确保表格里的字段只接受特定范围里的值?

答： Check 限制， 它在数据库表格里被定义， 用来限制输入该列的值。

触发器也可以被用来限制数据库表格里的字段能够接受的值， 但是这种办法要求触发器在表格里被定义， 这可能会在某些情况下影响到性能。

## 50、说说对 SQL 语句优化有哪些方法？（选择几条）

1、Where 子句中：where 表之间的连接必须写在其他 Where 条件之前，那些可以过滤掉最大数量记录的条件必须写在 Where 子句的末尾.HAVING 最后。

2、用 EXISTS 替代 IN、用 NOT EXISTS 替代 NOT IN。

3、 避免在索引列上使用计算

4、避免在索引列上使用 IS NULL 和 IS NOT NULL

5、对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

6、应尽量避免在 where 子句中对字段进行 null 值判断， 否则将导致引擎放弃使用索引而进行全表扫描

7、应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描

# 性能调优专题-JVM 面试题

## 1、你能保证 GC 执行吗？

不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC 的执行。

## 2、怎么获取 Java 程序使用的内存？堆使用的百分比？

可以通过 java.lang.Runtime 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。Runtime.freeMemory() 方法返回剩余空间的字节数， Runtime.totalMemory() 方法总内存的字节数，Runtime.maxMemory() 返回最大内存的字节数。

## 3、Java 中堆和栈有什么区别？

JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小， 也不会在多个线程之间共享， 而堆被整个 JVM 的所有线程共享。

## 4、JVM 选项 -XX:+UseCompressedOops 有什么作用？ 为什么要使用？

当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位， 因此堆内存会突然增加， 差不多要翻倍。这也会对 CPU 缓存（ 容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大小， 通过压缩 OOP 可以节省一定的内存。通过 -XX:+UseCompressedOops 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP 。

## 5、64 位 JVM 中，int 的长度是多数？

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位和 64 位的 Java 虚拟机中，int 类型的长度是相同的。

## 6、Serial 与 Parallel GC 之间的不同之处？

Serial 与 Parallel 在 GC 执行的时候都会引起 stop-the-world。它们之间主要不同 serial 收集器是默认的复制收集器，执行 GC 的时候只有一个线程，而
parallel 收集器使用多个 GC 线程来执行。

## 7、32 位和 64 位的 JVM，int 类型变量的长度是多数？

32 位和 64 位的 JVM 中，int 类型变量的长度是相同的，都是 32 位或者 4 个字节。

## 8、Java 中 WeakReference 与 SoftReference 的区别？

虽然 WeakReference 与 SoftReference 都有利于提高 GC 和 内存的效率，但是 WeakReference，一旦失去最后一个强引用，就会被 GC 回收，而软引用虽然不能阻止被回收，但是可以延迟到 JVM 内存不足的时候。

## 9、WeakHashMap 是怎么工作的？

WeakHashMap 的工作与正常的 HashMap 类似， 但是使用弱引用作为 key， 意思就是当 key 对象没有任何引用时， key/value 将会被回收。

## 10、怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位？

你可以检查某些系统属性如 sun.arch.data.model 或 os.arch 来获取该信息。

## 11、32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数？

理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5 GB，Solaris 大约3GB。64 位 JVM 允许指定最大的堆内存， 理论上可以达到 2^64， 这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul， 堆内存到 1000G 都是可能的。

## 12、JRE、JDK、JVM 及 JIT 之间有什么不同？

JRE 代表 Java 运行时（ Java run-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（ Java development kit），是 Java 程序的开发工具，如 Java 编译器， 它也包含 JRE。JVM 代表 Java 虚拟机（ Java virtual machine）， 它的责任是运行 Java 应用。JIT 代表即时编译（ Just In Time compilation）， 当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码， 如， 主要的热点代码会被准换为本地代码， 这样有利大幅度提高 Java 应用的性能。

## 13、GC 是什么？为什么要有GC？

答：

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃， Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的， Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理， 因为垃圾收集器会自动进行管理。要请求垃圾收集， 可以调用下面的方法之一： System.gc() 或
Runtime.getRuntime().gc()， 但 JVM 可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的 Java 进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java 平台对堆内存回收和再利用的基本算法被称为标记和清除，但是 Java 对其进行了改进，采用“分代式垃圾收集”。这种方法会跟 Java 对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的 JVM 参数：

- -Xms / -Xmx — 堆的初始大小 / 堆的最大大小
- -Xmn — 堆中年轻代的大小
- -XX:-DisableExplicitGC — 让 System.gc()不产生任何作用
- -XX:+PrintGCDetails — 打印 GC 的细节

- -XX:+PrintGCDateStamps — 打印 GC 操作的时间戳
- -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
- -XX:NewRatio — 可以设置老生代和新生代的比例
- -XX:PrintTenuringDistribution — 设置每次新生代 GC 后输出幸存者乐园中对象年龄的分布
- -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值
- -XX:TargetSurvivorRatio：设置幸存区的目标使用率

# 微服务专题

## 微服务专题-微服务面试题

1、您对微服务有何了解？

微服务， 又称微服务 架构， 是一种架构风格， 它将应用程序构建为以业务领域为模型的小型自治服务集合 。

通俗地说， 你必须看到蜜蜂如何通过对齐六角形蜡细胞来构建它们的蜂窝状物。他们最初从使用各种材料的小部分开始， 并继续从中构建一个大型蜂箱。这些细胞形成图案， 产生坚固的结构， 将蜂窝的特定部分固定在一起。这里， 每个细胞独立于另一个细胞， 但它也与其他细胞相关。这意味着对一个细胞的损害不会损害其他细胞， 因此， 蜜蜂可以在不影响完整蜂箱的情况下重建这些细胞。

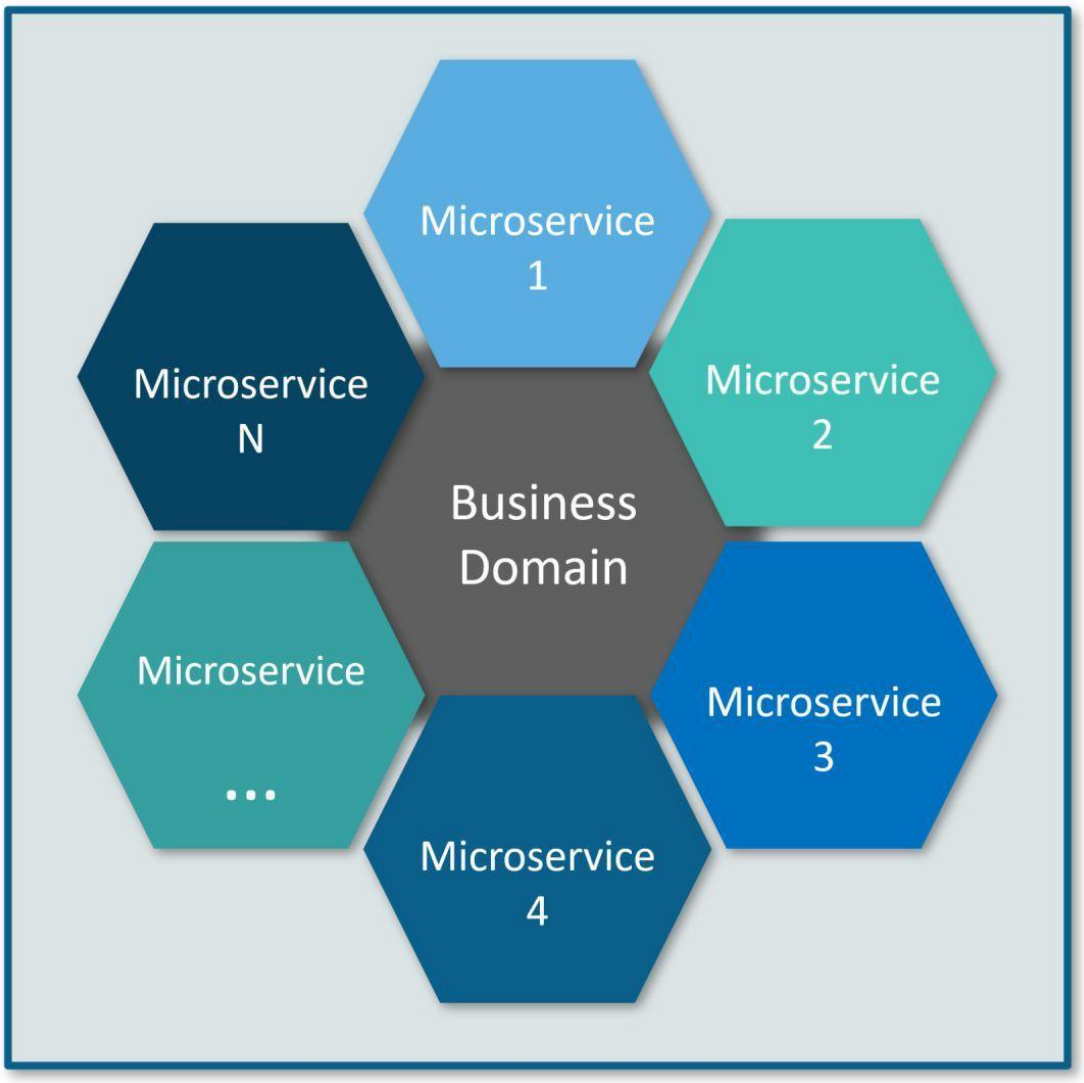


图 1： 微服务的蜂窝表示 – 微服务访谈问题

请参考上图。这里， 每个六边形形状代表单独的服务组件。与蜜蜂的工作类似， 每个敏捷团队都使用可用的框架和所选的技术堆栈构建单独的服务组件。就像在蜂箱中一样， 每个服务组件形成一个强大的微服务架构， 以提供更好的可扩展性。此外， 敏捷团队可以单独处理每个服务组件的问题， 而对整个应用程序没有影响或影响最小。

## 2、微服务架构有哪些优势？

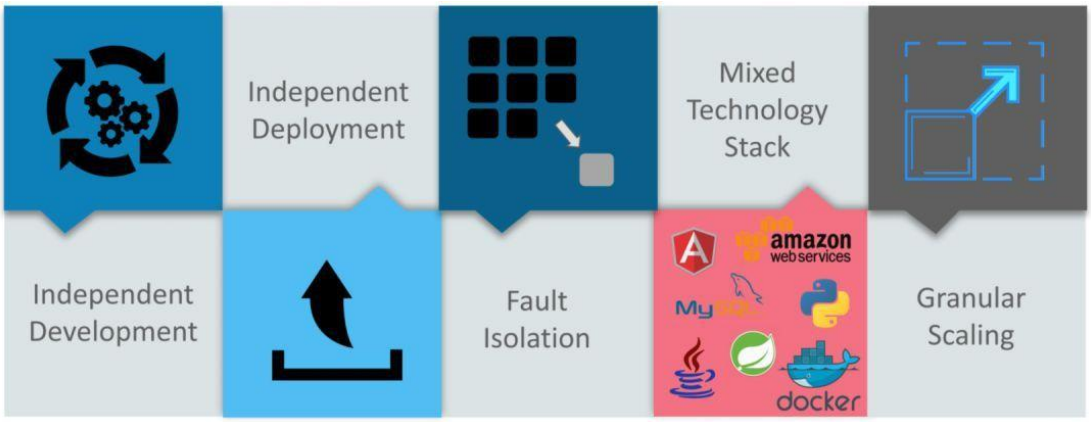



图 2： 微服务的 优点 – 微服务访谈问题

- 独立开发 – 所有微服务都可以根据各自的功能轻松开发
- 独立部署 – 基于其服务，可以在任何应用程序中单独部署它们
- 故障隔离 – 即使应用程序的一项服务不起作用，系统仍可继续运行
- 混合技术堆栈 – 可以使用不同的语言和技术来构建同一应用程序的不同服务
- 粒度缩放 – 单个组件可根据需要进行缩放，无需将所有组件缩放在一起

## 3。微服务有哪些特点？


图 3： 微服务的 特点 – 微服务访谈问题

- 解耦 – 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展
- 组件化 – 微服务被视为可以轻松更换和升级的独立组件
- 业务能力 – 微服务非常简单，专注于单一功能
- 自治 – 开发人员和团队可以彼此独立工作，从而提高速度
- 持续交付 – 通过软件创建，测试和批准的系统自动化，允许频繁发布软件
- 责任 – 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品
- 分散治理 – 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题
- 敏捷 – 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

## 4、设计微服务的最佳实践是什么？

以下是设计微服务的最佳实践：

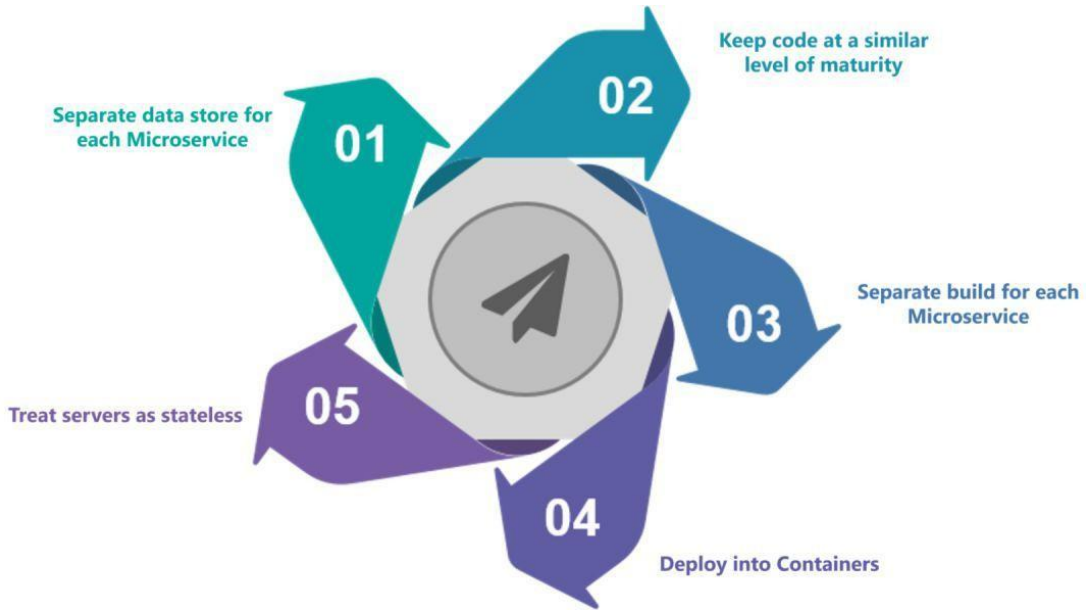



图 4： 设计微服务的最佳实践 – 微服务访谈问题

## 5、微服务架构如何运作？

微服务架构具有以下组件：

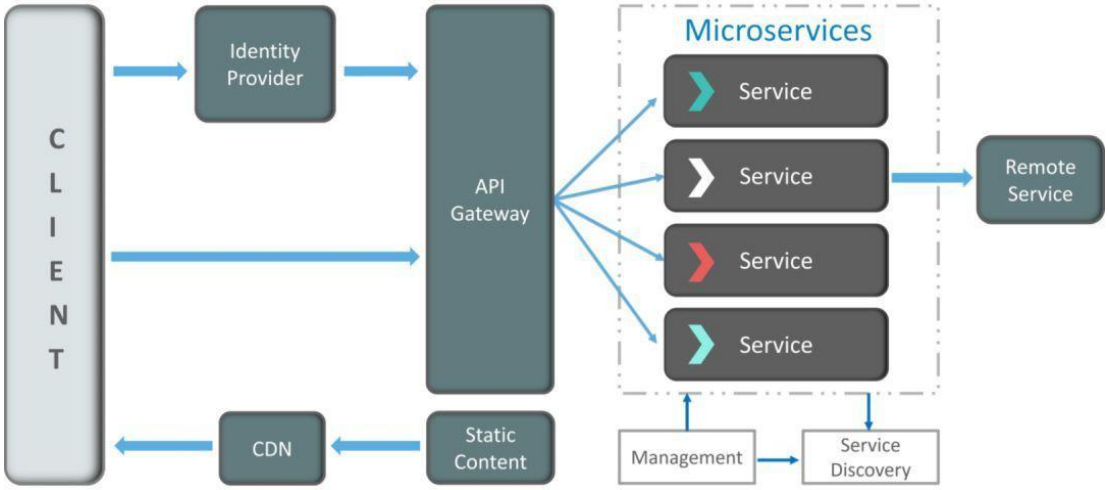



图 5： 微服务 架构 – 微服务面试问题

- 客户端 – 来自不同设备的不同用户发送请求。
- 身份提供商 – 验证用户或客户身份并颁发安全令牌。
- API 网关 – 处理客户端请求。
- 静态内容 – 容纳系统的所有内容。
- 管理 – 在节点上平衡服务并识别故障。
- 服务发现 – 查找微服务之间通信路径的指南。
- 内容交付网络 – 代理服务器及其数据中心的分布式网络。
- 远程服务 – 启用驻留在 IT 设备网络上的远程访问信息。

## 6、微服务架构的优缺点是什么？

| 微服务架构的优点 | 微服务架构的缺点 |
| --- | --- |
| 自由使用不同的技术 | 增加故障排除挑战 |
| 每个微服务都侧重于单一功能 | 由于远程呼叫而增加延迟 |
| 支持单个可部署单元 | 增加了配置和其他操作的工作量 |
| 允许经常发布软件 | 难以保持交易安全 |
| 确保每项服务的安全性 | 艰难地跨越各种边界跟踪数据 |
| 多个服务是并行开发和部署的 | 难以在服务之间进行编码 |

## 7、单片，SOA 和微服务架构有什么区别？

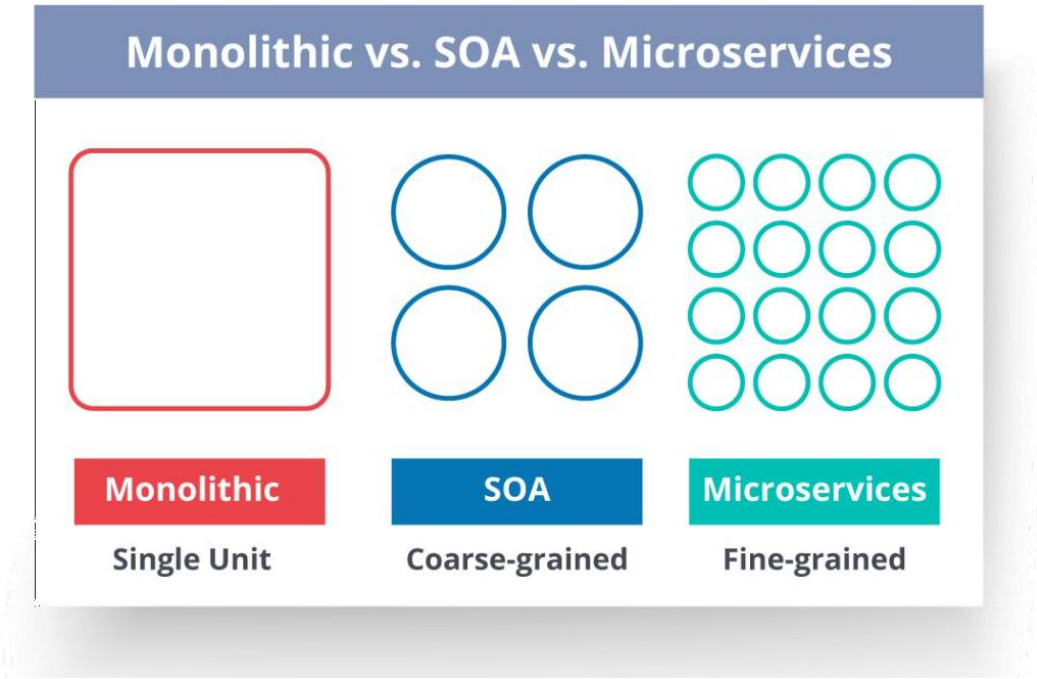



图 6： 单片 SOA 和微服务之间的比较 – 微服务访谈问题

- 单片架构类似于大容器，其中应用程序的所有软件组件组装在一起并紧密封装。

- 一个面向服务的架构是一种相互通信服务的集合。通信可以涉及简单的数据传递，也可以涉及两个或多个协调某些活动的服务。
- 微服务架构是一种架构风格，它将应用程序构建为以业务域为模型的小型自治服务集合。

## 8、在使用微服务架构时，您面临哪些挑战？

开发一些较小的微服务听起来很容易， 但开发它们时经常遇到的挑战如下。

- 自动化组件：难以自动化，因为有许多较小的组件。因此，对于每个组件，我们必须遵循 Build，Deploy 和 Monitor 的各个阶段。
- 易感性：将大量组件维护在一起变得难以部署，维护，监控和识别问题。它需要在所有组件周围具有很好的感知能力。
- 配置管理：有时在各种环境中维护组件的配置变得困难。
- 调试：很难找到错误的每一项服务。维护集中式日志记录和仪表板以调试问题至关重要。

## 9、SOA 和微服务架构之间的主要区别是什么？

SOA 和微服务之间的主要区别如下：

| SOA | 微服务 |
| --- | --- |
| 遵循" 尽可能多的共享 "架构方法 | 遵循" 尽可能少分享 "的架构方法 |
| 重要性在于 业务功能 重用 | 重要性在于" 有界背景 " 的概念 |
| 他们有 共同的 治理 和标准 | 他们专注于 人们的 合作 和其他选择的自由 |
| 使用 企业服务总线（ESB） 进行通信 | 简单的消息系统 |
| 它们支持 多种消息协议 | 他们使用 轻量级协议 ，如 HTTP / REST 等。 |
| 多线程， 有更多的开销来处理I / O. | 单线程 通常使用Event Loop功能进行非锁定I / O处理 |
| 最大化应用程序服务可重用性 | 专注于 解耦 |
| 传统的关系数据库 更常用 | 现代 关系数据库 更常用 |
| 系统的变化需要修改整体 | 系统的变化是创造一种新的服务 |
| DevOps / Continuous Delivery正在变得流行，但还不是主流 | 专注于DevOps /持续交付 |

## 10、微服务有什么特点？

您可以列出微服务的特征， 如下所示：

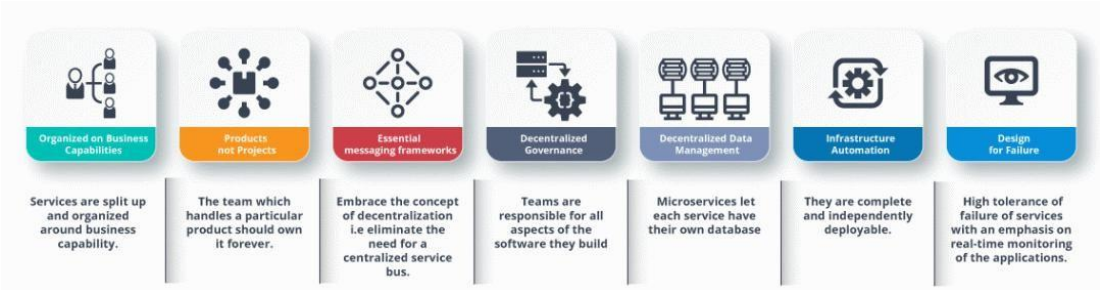



图 7： 微服务的特征 – 微服务访谈问题

## 11、什么是领域驱动设计？

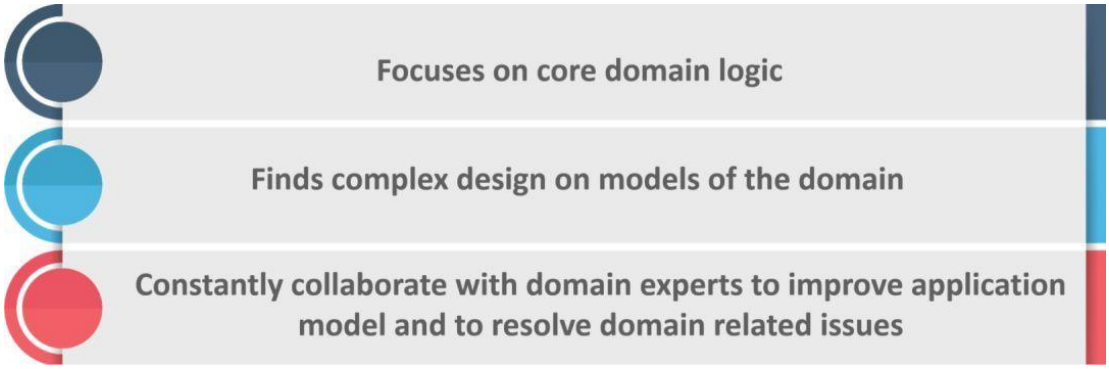



图 8： DDD 原理 – 微服务面试问题

## 12、为什么需要域驱动设计（DDD）？

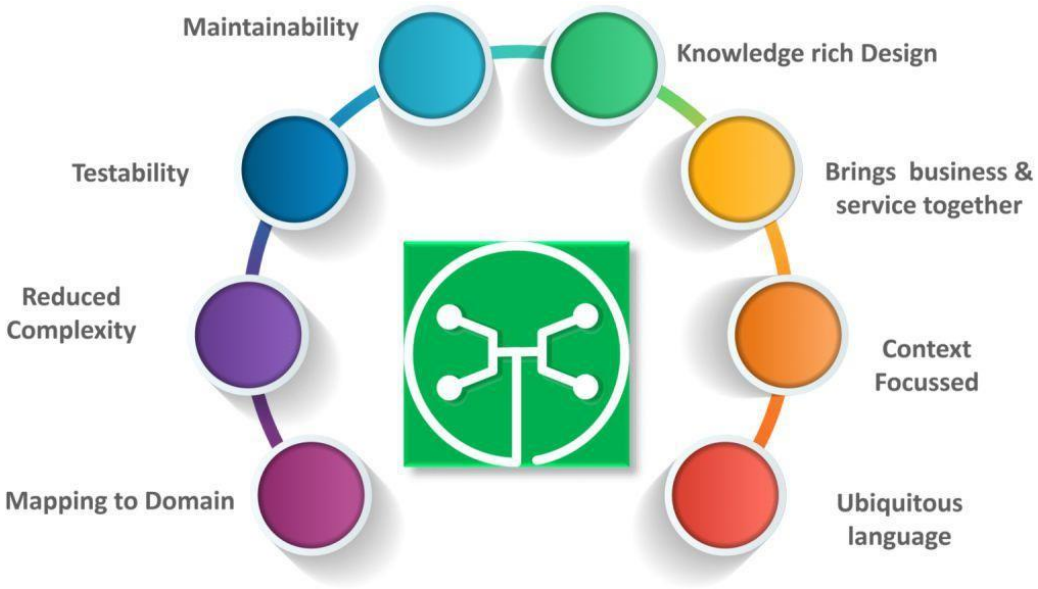



图 9： 我们需要 DDD 的因素 – 微服务面试问题

## 13、什么是无所不在的语言？

如果您必须定义泛在语言（ UL），那么它是特定域的开发人员和用户使用的通用语言，通过该语言可以轻松解释域。

无处不在的语言必须非常清晰，以便它将所有团队成员放在同一页面上，并以机器可以理解的方式进行翻译。

## 14、什么是凝聚力？

模块内部元素所属的程度被认为是凝聚力。

## 15、什么是耦合？

组件之间依赖关系强度的度量被认为是耦合。一个好的设计总是被认为具有高内 聚力和低耦合性。

## 16、什么是 REST / RESTful 以及它的用途是什么？

Representational State Transfer（ REST） / RESTful Web 服务是一种帮助计算机系统通过 Internet 进行通信的架构风格。这使得微服务更容易理解和实现。

微服务可以使用或不使用 RESTful API 实现，但使用 RESTful API 构建松散耦合的微服务总是更容易。

## 17、你对 Spring Boot 有什么了解？

事实上，随着新功能的增加，弹簧变得越来越复杂。如果必须启动新的 spring 项目，则必须添加构建路径或添加 maven 依赖项，配置应用程序服务器，添加 spring 配置。所以一切都必须从头开始。

Spring Boot 是解决这个问题的方法。使用 spring boot 可以避免所有样板代码和配置。因此，基本上认为自己就好像你正在烘烤蛋糕一样，春天就像制作蛋糕所需的成分一样，弹簧靴就是你手中的完整蛋糕。


图 10： Spring Boot 的因素 – 微服务面试问题

## 18、什么是 Spring 引导的执行器？

Spring Boot 执行程序提供了 restful Web 服务，以访问生产环境中运行应用程序的当前状态。在执行器的帮助下，您可以检查各种指标并监控您的应用程序。

## 19、什么是 Spring Cloud？

根据 Spring Cloud 的官方网站， Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理， 服务发现， 断路器， 智能路由， 领导选举， 分布式会话， 集群状态）。

## 20、Spring Cloud 解决了哪些问题？

在使用 Spring Boot 开发分布式微服务时，我们面临的问题很少由 Spring Cloud 解决。

- 与分布式系统相关的复杂性 – 包括网络问题，延迟开销，带宽问题，安全问题。
- 处理服务发现的能力 – 服务发现允许集群中的进程和服务找到彼此并进行通信。
- 解决冗余问题 – 冗余问题经常发生在分布式系统中。
- 负载平衡 – 改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。
- 减少性能问题 – 减少因各种操作开销导致的性能问题。

## 21、在 Spring MVC 应用程序中使用 WebMvcTest 注释有什么用处？

```
@WebMvcTest(value = ToTestController.class, secure = false):
```

在测试目标只关注 Spring MVC 组件的情况下，WebMvcTest 注释用于单元测试 Spring MVC 应用程序。在上面显示的快照中，我们只想启动 ToTestController。执行此单元测试时， 不会启动所有其他控制器和映射。

## 22。你能否给出关于休息和微服务的要点？

虽然您可以通过多种方式实现微服务， 但 REST over HTTP 是实现微服务的一种方式。REST 还可用于其他应用程序，如 Web 应用程序，API 设计和 MVC 应用程序， 以提供业务数据。

微服务是一种体系结构， 其中系统的所有组件都被放入单独的组件中， 这些组件可以单独构建， 部署和扩展。微服务的某些原则和最佳实践有助于构建弹性应用程序。

简而言之， 您可以说 REST 是构建微服务的媒介。

## 23、什么是不同类型的微服务测试？

在使用微服务时， 由于有多个微服务协同工作， 测试变得非常复杂。因此， 测试分为不同的级别。

- 在底层，我们有面向技术的测试，如单元测试和性能测试。这些是完全自动化的。
- 在中间层面，我们进行了诸如压力测试和可用性测试之类的探索性测试。
- 在顶层， 我们的 验收测试数量很少。这些验收测试有助于利益相关者理解和验证软件功能。

## 24、您对 Distributed Transaction 有何了解？

分布式事务是指单个事件导致两个或多个不能以原子方式提交的单独数据源的突 变的任何情况。在微服务的世界中， 它变得更加复杂， 因为每个服务都是一个工作单元， 并且大多数时候多个服务必须协同工作才能使业务成功。

## 25、什么是 Idempotence 以及它在哪里使用？

幂等性是能够以这样的方式做两次事情的特性， 即最终结果将保持不变， 即好像它只做了一次。

用法：在远程服务或数据源中使用 Idempotence，这样当它多次接收指令时，它只处理指令一次。

## 26、什么是有界上下文？

有界上下文是域驱动设计的核心模式。DDD 战略设计部门的重点是处理大型模型和团队。DDD 通过将大型模型划分为不同的有界上下文并明确其相互关系来处理大型模型。

## 27、什么是双因素身份验证？

双因素身份验证为帐户登录过程启用第二级身份验证。


图 11： 双因素认证的表示 – 微服务访谈问题

因此， 假设用户必须只输入用户名和密码， 那么这被认为是单因素身份验证。

## 28、双因素身份验证的凭据类型有哪些？

这三种凭证是：

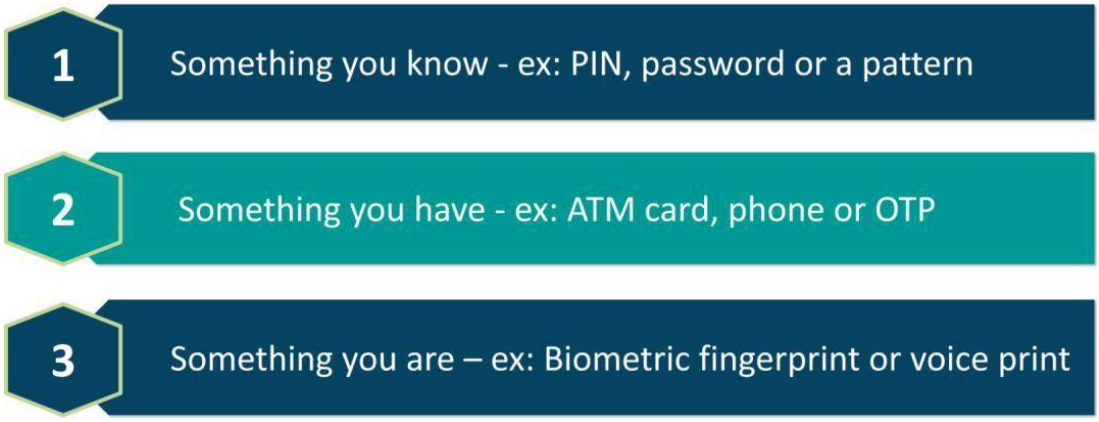


图 12： 双因素认证的证书类型 – 微服务面试问题

## 29、什么是客户证书？

客户端系统用于向远程服务器发出经过身份验证的请求的一种数字证书称为客户 端证书。客户端证书在许多相互认证设计中起着非常重要的作用， 为请求者的身份提供了强有力的保证。

## 30、PACT 在微服务架构中的用途是什么？

PACT 是一个开源工具， 允许测试服务提供者和消费者之间的交互， 与合同隔离， 从而提高微服务集成的可靠性。

微服务中的用法

- 用于在微服务中实现消费者驱动的合同。
- 测试微服务的消费者和提供者之间的消费者驱动的合同。

查看即将到来的批次

## 31、什么是 OAuth？

OAuth 代表开放授权协议。这允许通过在 HTTP 服务上启用客户端应用程序（ 例如第三方提供商 Facebook， GitHub 等） 来访问资源所有者的资源。因此， 您可以在不使用其凭据的情况下与另一个站点共享存储在一个站点上的资源。

## 32、康威定律是什么？

“任何设计系统的组织（广泛定义）都将产生一种设计，其结构是组织通信结构的副本。” – Mel Conway

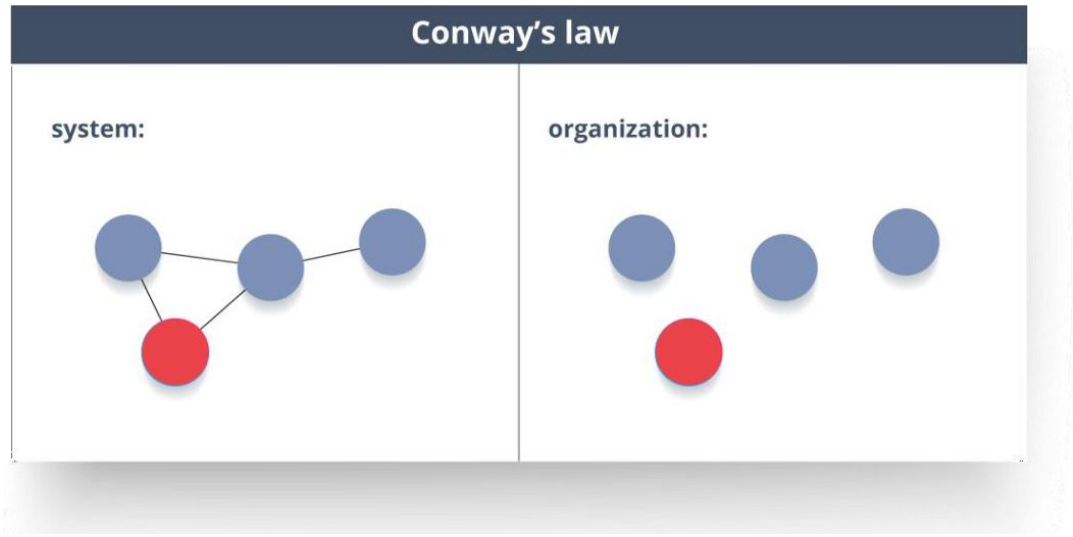



图 13： Conway 定律的表示 – 微服务访谈问题

该法律基本上试图传达这样一个事实： 为了使软件模块起作用， 整个团队应该进行良好的沟通。因此， 系统的结构反映了产生它的组织的社会边界。

## 33、合同测试你懂什么？

根据 Martin Flower 的说法， 合同测试是在外部服务边界进行的测试， 用于验证其是否符合消费服务预期的合同。

此外， 合同测试不会深入测试服务的行为。更确切地说， 它测试该服务调用的输入& 输出包含所需的属性和所述响应延迟， 吞吐量是允许的限度内。

## 34、什么是端到端微服务测试？

端到端测试验证了工作流中的每个流程都正常运行。这可确保系统作为一个整体 协同工作并满足所有要求。

通俗地说， 你可以说端到端测试是一种测试， 在特定时期后测试所有东西。

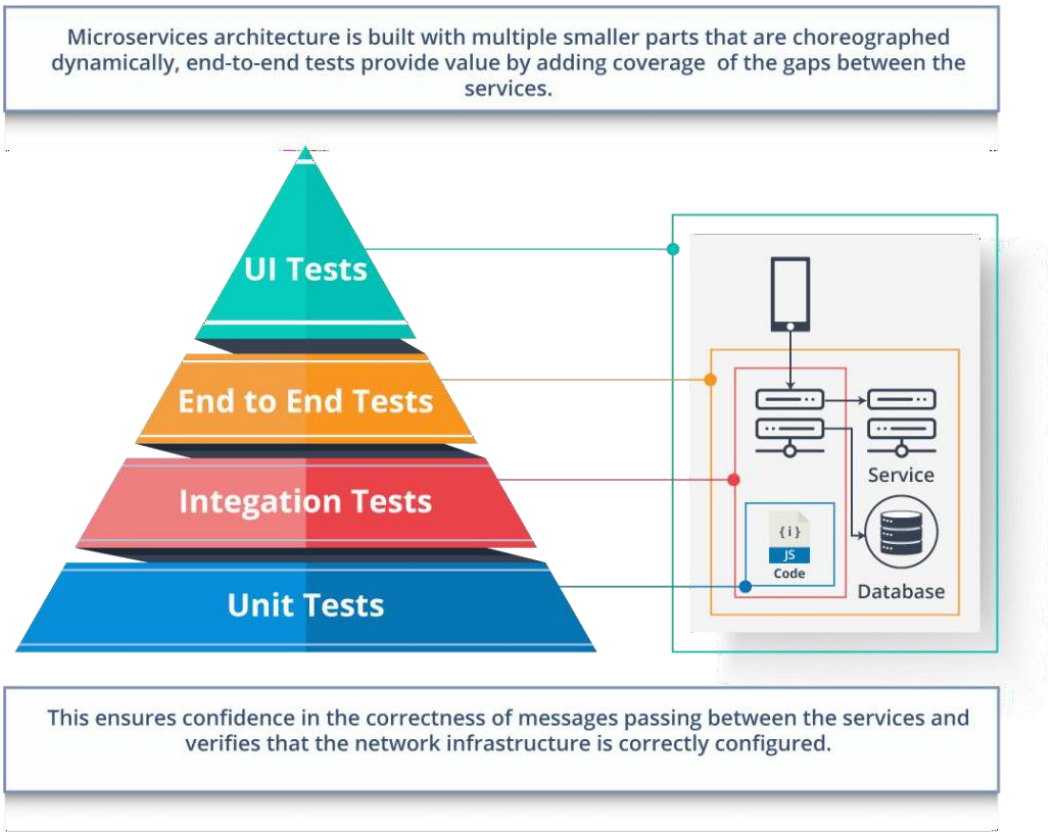



图 14： 测试层次 – 微服务面试问题

## 35、Container 在微服务中的用途是什么？

容器是管理基于微服务的应用程序以便单独开发和部署它们的好方法。您可以将微服务封装在容器映像及其依赖项中， 然后可以使用它来滚动按需实例的微服务， 而无需任何额外的工作。

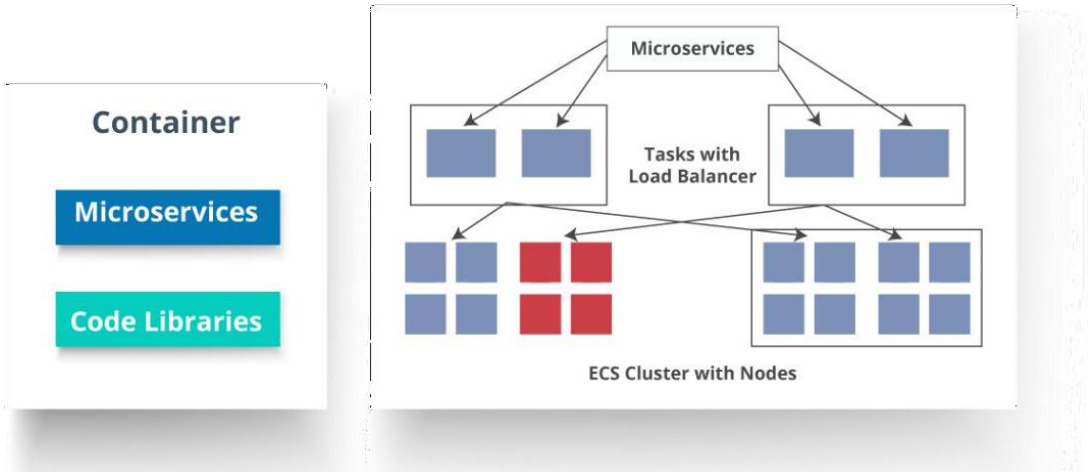



图 15： 容器的表示及其在微服务中的使用方式 – 微服务访谈问题

## 36、什么是微服务架构中的 DRY？

DRY 代表不要重复自己。它基本上促进了重用代码的概念。这导致开发和共享库， 这反过来导致紧密耦合。

## 37、什么是消费者驱动的合同（CDC）？

这基本上是用于开发微服务的模式， 以便它们可以被外部系统使用。当我们处理微服务时，有一个特定的提供者构建它， 并且有一个或多个使用微服务的消费者。

通常，提供程序在 XML 文档中指定接口。但在消费者驱动的合同中， 每个服务消费者都传达了提供商期望的接口。

## 38、Web，RESTful API 在微服务中的作用是什么？

微服务架构基于一个概念， 其中所有服务应该能够彼此交互以构建业务功能。因此， 要实现这一点， 每个微服务必须具有接口。这使得 Web API 成为微服务的一个非常重要的推动者。RESTful API 基于 Web 的开放网络原则， 为构建微服务架构的各个组件之间的接口提供了最合理的模型。

## 39、您对微服务架构中的语义监控有何了解？

语义监控， 也称为 综合监控， 将自动化测试与监控应用程序相结合， 以检测业务失败因素。

## 40、我们如何进行跨功能测试？

跨功能测试是对非功能性需求的验证， 即那些无法像普通功能那样实现的需求。

## 41、我们如何在测试中消除非决定论？

非确定性测试（NDT） 基本上是不可靠的测试。所以， 有时可能会发生它们通过， 显然有时它们也可能会失败。当它们失败时， 它们会重新运行通过。

从测试中删除非确定性的一些方法如下：

1、 隔离

2、 异步

3、 远程服务

4、 隔离

5、 时间
6、 资源泄漏

## 42、Mock 或Stub 有什么区别？

存根

- 一个有助于运行测试的虚拟对象。
- 在某些可以硬编码的条件下提供固定行为。
- 永远不会测试存根的任何其他行为。

例如，对于空堆栈，您可以创建一个只为 empty（）方法返回 true 的存根。因此， 这并不关心堆栈中是否存在元素。

嘲笑

- 一个虚拟对象，其中最初设置了某些属性。
- 此对象的行为取决于 set 属性。
- 也可以测试对象的行为。

例如，对于 Customer 对象，您可以通过设置名称和年龄来模拟它。您可以将 age 设置为 12，然后测试 isAdult（）方法，该方法将在年龄大于 18 时返回 true。因此， 您的 Mock Customer 对象适用于指定的条件。

## 43、您对 Mike Cohn 的测试金字塔了解多少？

Mike Cohn 提供了一个名为 Test Pyramid 的模型。这描述了软件开发所需的自动化测试类型。

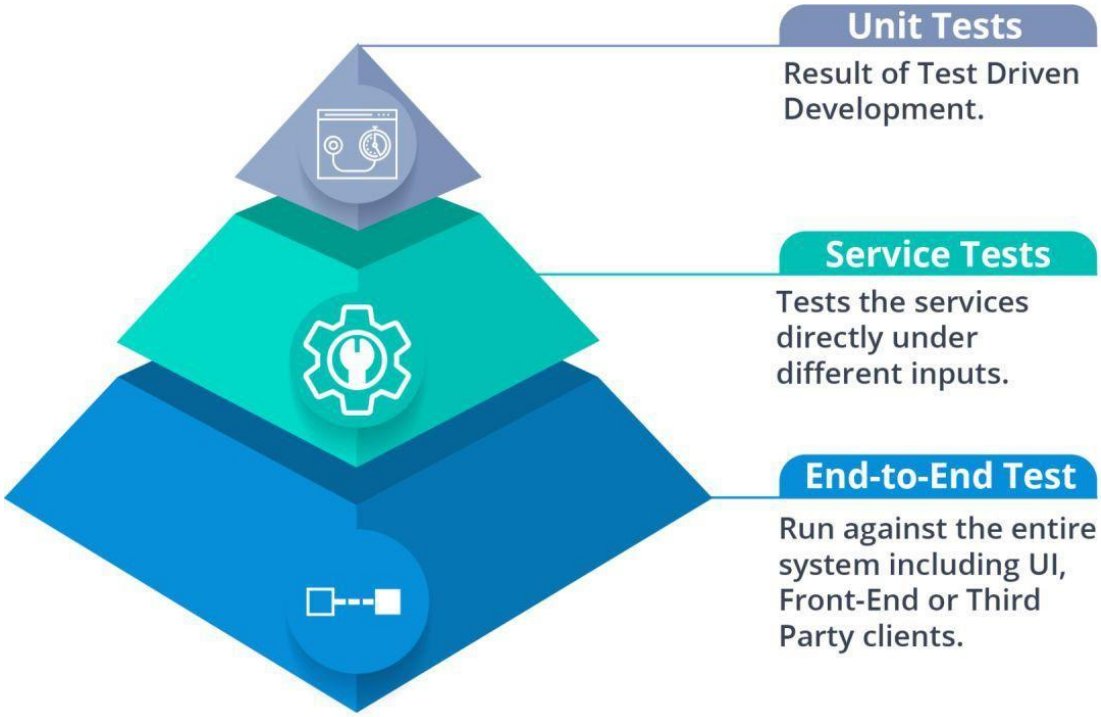



图 16： Mike Cohn 的测试金字塔 – 微服务面试问题

根据金字塔， 第一层的测试数量应该最高。在服务层， 测试次数应小于单元测试级别，但应大于端到端级别。

## 44、Docker 的目的是什么？

Docker 提供了一个可用于托管任何应用程序的容器环境。在此，软件应用程序和支持它的依赖项紧密打包在一起。

因此， 这个打包的产品被称为 Container， 因为它是由 Docker 完成的， 所以它被称为 Docker 容器！

## 45、什么是金丝雀释放？

Canary Releasing 是一种降低在生产中引入新软件版本的风险的技术。这是通过将变更缓慢地推广到一小部分用户， 然后将其发布到整个基础架构， 即将其提供给每个人来完成的。

## 46、什么是持续集成（CI）？

持续集成（ CI） 是每次团队成员提交版本控制更改时自动构建和测试代码的过程。这鼓励开发人员通过在每个小任务完成后将更改合并到共享版本控制存储库来共享代码和单元测试。

## 47、什么是持续监测？

持续监控深入监控覆盖范围， 从浏览器内前端性能指标， 到应用程序性能， 再到主机虚拟化基础架构指标。

## 48、架构师在微服务架构中的角色是什么？

微服务架构中的架构师扮演以下角色：

- 决定整个软件系统的布局。
- 帮助确定组件的分区。因此，他们确保组件相互粘合，但不紧密耦合。
- 与开发人员共同编写代码，了解日常生活中面临的挑战。
- 为开发微服务的团队提供某些工具和技术的建议。

- 提供技术治理，以便技术开发团队遵循微服务原则。

## 49、我们可以用微服务创建状态机吗？

我们知道拥有自己的数据库的每个微服务都是一个可独立部署的程序单元， 这反过来又让我们可以创建一个状态机。因此， 我们可以为特定的微服务指定不同的状态和事件。

例如， 我们可以定义 Order 微服务。订单可以具有不同的状态。Order 状态的转换可以是 Order 微服务中的独立事件。

## 50、什么是微服务中的反应性扩展？

Reactive Extensions 也称为 Rx。这是一种设计方法， 我们通过调用多个服务来收集结果， 然后编译组合响应。这些调用可以是同步或异步， 阻塞或非阻塞。Rx 是分布式系统中非常流行的工具， 与传统流程相反。

希望这些微服务面试问题可以帮助您进行微服务架构师访谈。

翻译来源：
https://www.edureka.co/blog/interview-questions/microservices-intervie w-questions/

# 微服务专题-Spring Boot面试题

## 1、什么是Spring Boot？

多年来， 随着新功能的增加， spring 变得越来越复杂。只需访问 https://spring.io/projects 页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目， 我们必须添加构建路径或添加 Maven 依赖关系， 配置应用程序服务器， 添加 spring 配置。因此， 开始一个新的 spring 项目需要很多努力， 因为我们现在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。Spring Boot 已经建立在现有 spring 框架之上。使用 spring 启动， 我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot 可以帮助我们以最少的工作量， 更加健壮地使用现有的 Spring 功能。

## 2、Spring Boot 有哪些优点？

Spring Boot 的优点有：

1、减少开发， 测试时间和努力。

2、使用 JavaConfig 有助于避免使用 XML。

3、避免大量的 Maven 导入和各种版本冲突。

4、提供意见发展方法。

5、通过提供默认值快速开始开发。

6、没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat， Glassfish 或其他任何东西。

7、需要更少的配置 因为没有 web.xml 文件。只需添加用@ Configuration 注释的类，然后添加用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将@Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。

8、基于环境的配置 使用这些属性，您可以将您正在使用的环境传递到应用程序：-Dspring.profiles.active = {enviornment}。在加载主应用程序属性文件后， Spring 将在（ application{environment} .properties）中加载后续的应用程序属性文件。

## 3、什么是JavaConfig？

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

1、面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

2、减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将

JavaConfig 与 XML 混合匹配是理想的。

3、类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

## 4、如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat 将重新启动。Spring Boot 有一个开发工具（ DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
```

## 5、Spring Boot 中的监视器是什么？

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的REST 端点来检查状态。

## 6、如何在Spring Boot 中禁用 Actuator 端点安全性？

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用

来禁用安全性。只有在执行机构端点在防火墙后访问时， 才建议禁用安全性。

## 7、如何在自定义端口上运行Spring Boot 应用程序？

为了在自定义端口上运行 Spring Boot 应用程序， 您可以在
application.properties 中指定端口。

```
server.port=8090
```

## 8、什么是 YAML？

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化， 而且更少混淆。可以看出 YAML 具有分层配置数据。

## 9、如何实现Spring Boot 应用程序的安全性？

为了实现 Spring Boot 的安全性， 我们使用 spring-boot-starter-security 依赖项， 并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展 WebSecurityConfigurerAdapter 并覆盖其方法。

## 10、如何集成 Spring Boot 和 ActiveMQ？

对于集成 Spring Boot 和 ActiveMQ， 我们使用

依赖关系。 它只需要很少的配置， 并且不需要样板代码。

## 11、如何使用 Spring Boot 实现分页和排序？

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的

传递给存储库方法。

## 12、什么是 Swagger？你用Spring Boot 实现了它吗？

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

## 13、什么是 Spring Profiles？

Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION 中， 某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

## 14、什么是 Spring Batch？

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪， 事务管理， 作业处理统计信息， 作业重新启动， 跳过和资源管理。它还提供了更先进的技术服务和功能， 通过优化和分区技术， 可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

## 15、什么是 FreeMarker 模板？

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码， 而设计人员可以处理 html 页面设计。最后使用

freemarker 可以将这些结合起来， 给出最终的输出页面。

## 16、如何使用 Spring Boot 实现异常处理？

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类， 来处理控制器类抛出的所有异常。

## 17、您使用了哪些 starter maven 依赖项？

使用了下面的一些依赖项

```
spring-boot-starter-activemq
spring-boot-starter-security
```

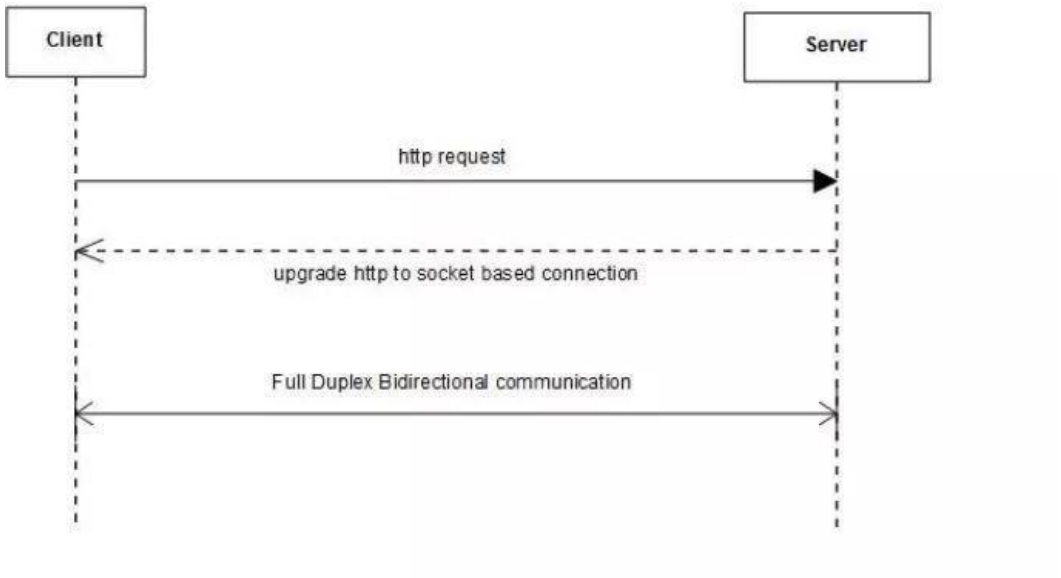
这有助于增加更少的依赖关系， 并减少版本的冲突。

## 18、什么是 CSRF 攻击？

CSRF 代表跨站请求伪造。这是一种攻击， 迫使最终用户在当前通过身份验证的Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求， 而不是数据窃取， 因为攻击者无法查看对伪造请求的响应。

## 19、什么是 WebSockets？

WebSocket 是一种计算机通信协议， 通过单个 TCP 连接提供全双工通信信道。




1、WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。2、

WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

3、单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

4、Light -与 http 相比， WebSocket 消息数据交换要轻得多。

# 20、什么是 AOP？

在软件开发过程中， 跨越应用程序多个点的功能称为交叉问题。这些交叉问题与应用程序的主要业务逻辑不同。因此， 将这些横切关注与业务逻辑分开是面向方面编程（ AOP） 的地方。

## 21、什么是 Apache Kafka？

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的， 容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

## 22、我们如何监视所有 Spring Boot 微服务？

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（ 如它们是否已启动） 以及它们的组件（ 如数据库等） 是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是， 我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务， 管理员将不得不击中所有 50 个应用程序的执行终端。

为了帮助我们处理这种情况， 我们将使用位于

的开源项目。 它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

# 微服务专题-Spring Cloud 面试题

## 1、什么是Spring Cloud？

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序， 提供与外部系统的集成。Spring cloud Task， 一个生命周期短暂的微服务框架， 用于快速构建执行有限数据处理的应用程序。

## 2、使用Spring Cloud 有什么优势？

使用 Spring Boot 开发分布式微服务时， 我们面临以下问题

1、与分布式系统相关的复杂性-这种开销包括网络问题， 延迟开销， 带宽问题， 安全问题。

2、服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录， 在该目录中注册服务， 然后能够查找并连接到该目录中的服务。

3、冗余-分布式系统中的冗余问题。

4、负载平衡 --负载平衡改善跨多个计算资源的工作负荷， 诸如计算机， 计算机集群， 网络链路， 中央处理单元， 或磁盘驱动器的分布。

5、性能-问题 由于各种运营开销导致的性能问题。

6、部署复杂性-Devops 技能的要求。

## 3、服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时， 我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署， 添加和修改这些属性变得更加复杂。有些服务可能会下降， 而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找， 因此无需处理服务地点的任何更改和处理。

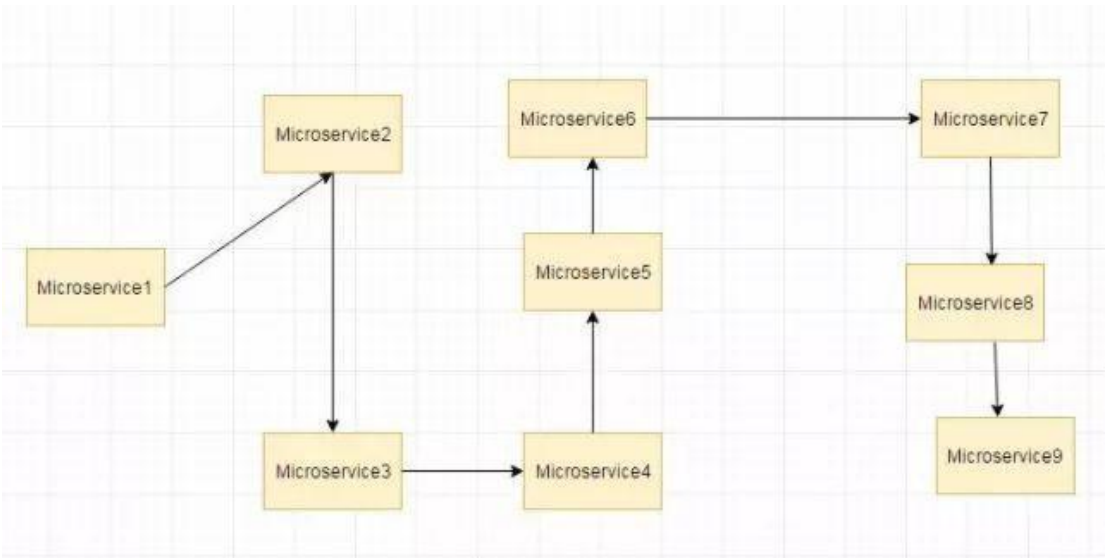
## 4、负载平衡的意义什么？

在计算中， 负载平衡可以改善跨计算机， 计算机集群， 网络链接， 中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用， 最大化吞吐量， 最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉 及专用软件或硬件， 例如多层交换机或域名系统服务器进程。

## 5、什么是Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统， 服务和第三方库的访问点， 当出现故障是不可避免的故障时， 停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统， 涉及到许多微服务。这些微服务彼此协作。思考以下微服务




假设如果上图中的微服务 9 失败了， 那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

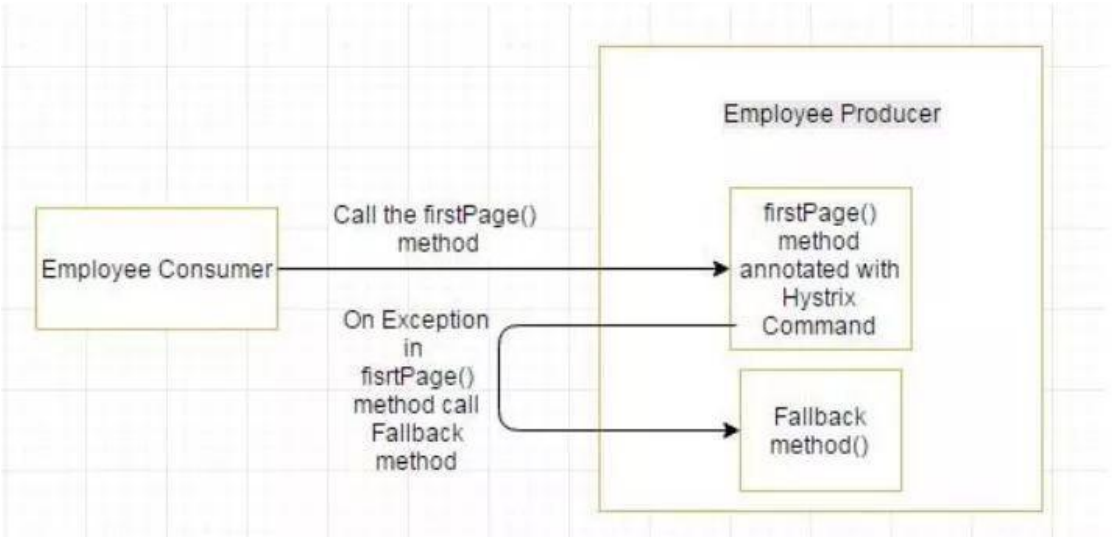
随着微服务数量的增加， 这个问题变得更加复杂。微服务的数量可以高达 1000. 这是 hystrix 出现的地方 我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

简化图如下所示

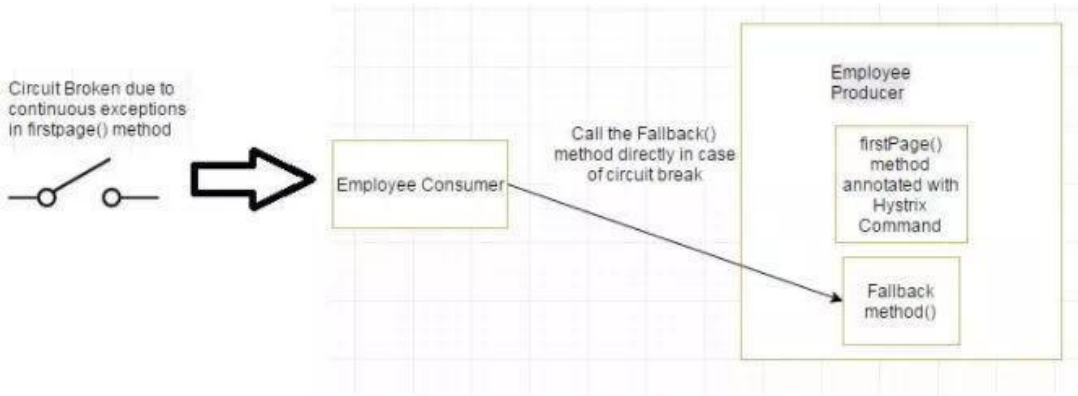

现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常， 则回退方法将返回一些值。

## 6、什么是Hystrix 断路器？我们需要它吗？

由于某些原因， employee-consumer 公开服务会引发异常。在这种情况下使用Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。




如果 firstPage method() 中的异常继续发生， 则 Hystrix 电路将中断， 并且员工使用者将一起跳过 firtsPage 方法， 并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间， 并导致异常恢复。可能发生的情况是， 在负载较小的情况下， 导致异常的问题有更好的恢复机会 。




## 7、什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤 1、

使用功能区进行负载平衡。

2、获取服务实例， 然后获取基本 URL。

3、利用 REST 模板来使用服务。 前面的代码如下

```
@Controller
public class ConsumerControllerClient {

@Autowired
private LoadBalancerClient loadBalancer;

public void getEmployee() throws RestClientException, IOException  {

      ServiceInstance
serviceInstance=loadBalancer.choose("employee-producer");

      System.out.println(serviceInstance.getUri()); String

      baseUrl=serviceInstance.getUri().toString();

      baseUrl=baseUrl+"/employee";

      RestTemplate restTemplate = new RestTemplate();
      ResponseEntity<String> response=null;
      try{ response=restTemplate.exchange(baseUrl,
                  HttpMethod.GET, getHeaders(),String.class);
```

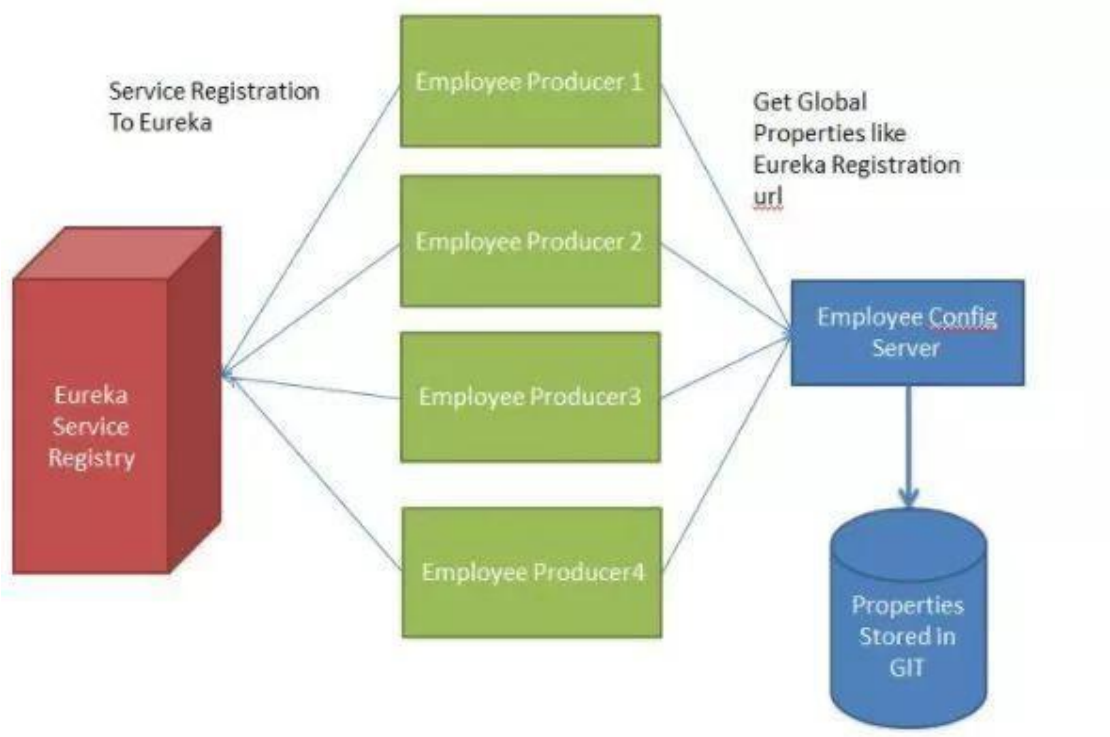
```
}catch (Exception ex)
{
      System.out.println(ex);
}
System.out.println(response.getBody());
```

之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

## 8、什么是Spring Cloud Bus？我们需要它吗？

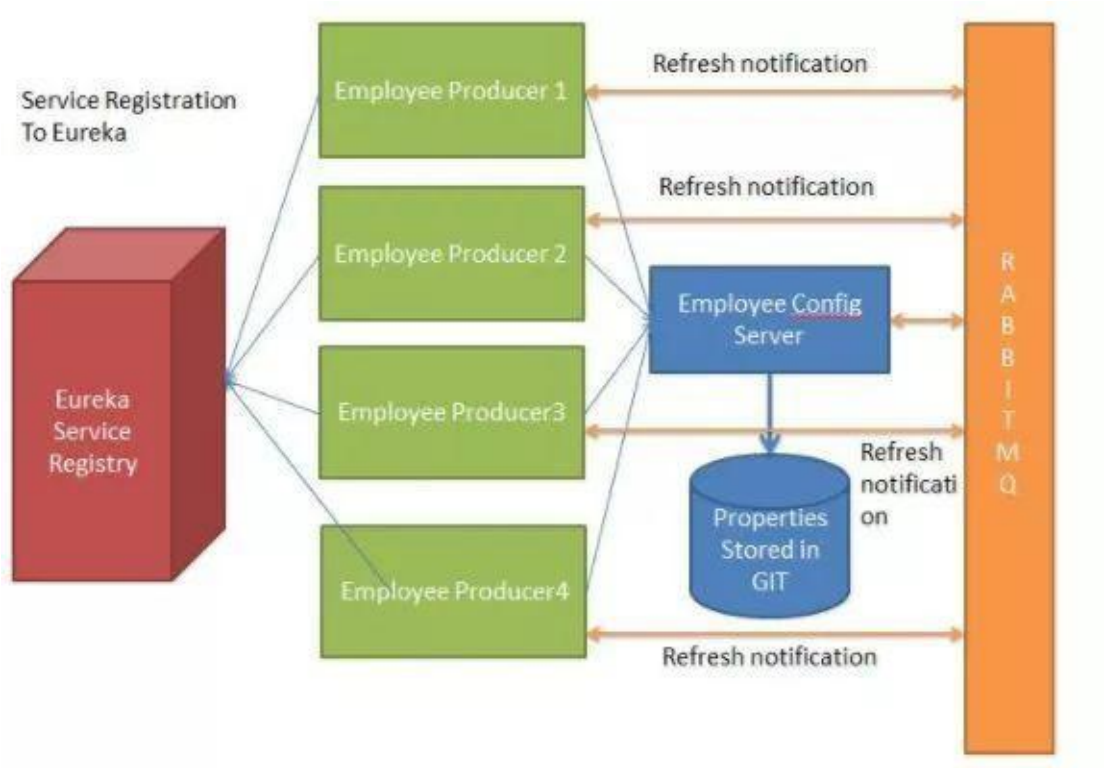
考虑以下情况： 我们有多个应用程序使用 Spring Cloud Config 读取属性， 而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。




如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器， 会发生什么情况。在这种情况下， 我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 http：// localhost： 8080 / refresh。同样对于 Employee Producer2 http：// localhost： 8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。




Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此， 在上面的示例中， 如果我们刷新 Employee Producer1， 则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行， 这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例， 此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的 刷新。