Spring

1. BeanFactory 和 ApplicationContext

•BeanFactory 采用的是延迟加载的形式来注入 Bean,即只有在使用某个 bean 的时候 ,才对该 Bean 进行加载实例化.好处是节约内存，缺点是速度比较慢.

- ApplicationContext 则相反的，它是**在 Ioc 容易启动时就一次性创建所有的 Bean**,这样的好处是可以马上发现 Spring 配置文件中的错误；坏处就是浪费内存。

application context 扩展了 BeanFactory，还提供了其他的功能。

- 国际化的功能
- 消息发送、响应机制(继承至 MessageSource)
- 统一加载资源的功能(继承至 ResourceLoader)
- 强大的事件机制(继承至 ApplicationEventPublisher)
- 对 Web 应用的支持()

2. spring 注解的实现原理

3. factoryBean 是什么？

BeanFactory 是 Spring 容器中的一个基本类也是很重要的一个类，在 BeanFactory 中可以**创建和管理 Spring 容器中的 Bean**，它对于 Bean 的创建有一个统一的流程。

FactoryBean 是一个工厂 Bean，可以生成某一个类型 Bean 实例，它最大的一个作用是：可以让我们**自定义 Bean 的创建过程**。

```
public interface FactoryBean<T> {
    //返回的对象实例
```

```
    T getObject() throws Exception;
    //Bean 的类型
    Class<?> getObjectType();
    //true 是单例，false 是非单例  在 Spring5.0 中此方法利用了 JDK1.8 的新特性变成了 default 方法，返回 true
    boolean isSingleton();
}
```

4. spring 依赖注入的实现原理

Spring 大量引入了 **Java 的 Reflection 机制** ,通过动态调用的方式避免硬编码方式的约束，并在此基础上建立了其核心组件 BeanFactory，以此作为其依赖注入机制的实现基础。

我们分三步走：

第一步解析需要注入的 beans.xml 生成 map

第二步根据第一步生成的 map 集合，生成 object 集合：

第三步就是扫描所有的 object 对象，将对象中注解信息解析出来匹配对应的注入对象，然后通过反射设置对象，完成了对象的注入。

下面就是一为 bean 对象注入相应属性值的关键代码，其中分了两种情况注入，一种是通过 ref 属性注入的 ,还有一种是经过 value 属性注入的简单属性(为了注入简单属性 ,在读取 xml 配置的时候，也保存属性为 value 的值，因此 propertyDefinition 中增加了 value 属性，用来存储对应的值)：

上面代码中最外层的 for 循环是依次将属性的值注入进从 xml 中读取并实例化的 bean 对象中。下面来分析循环中的一个：首先从集合 beanDefines 拿出一个 BeanDefinition 对象，接着从存储实例化的 bean 对象的 Map 对象 sigletons 中取得 bean 的引用。因为存储 bean 的 key 是相应 beanDefinition 中的 id 属性，所以可以很简单的通过这个 id 的值找到对应的实例化的 bean 对象(此时还为注入任何属性)。程序严谨一点就先判断一下 bean 是否为空，

因为师为 bean 注入属性，当然先保证它自身的存在。有了 bean 的实例对象后，就可以跟据 bean 的字节码码对象生成该 bean 的属性描述数组 ps，这是利用工具类 Introspector 办到的。然后开始遍历从配置文件读取的 bean 的定义对象 beanDefinition，从中获取某一个属性的描述对象。接下来的关键就是再次通过一个 for 循环遍历 ps 的属性描述数组，找到与前面从 beanDefinition 中获取的属性描述对象名字相同的属性的名字，找到后，通过该 ps 对象中的属性描述对象利用反射获取 setter 方法(Method setter = properdesc.getWriteMethod())。简单点说，就是以 properdesc 的 name 为依据，去 beanDefinition 里的属性描述对象里找着相同名字的属性定义对象。如果找到就可以开始执行注入功能了 ,在这里就可以判断该属性定义对象的 ref 属性有值还是 value 对象的值存在，如果是 ref，就可以从 spring 中 bean 的 Map 容器找 ref 属性值的 bean 对象，如果是 value 属性的定义，就调用 ConvertUtils 工具类中的 convert 方法，将 value 值转换成 bean 的属性的类型。最后就是设置 setter 方法的可访问性(准确的说就是为了能够访问私有的方法)，然后利用反射在相应的 bean 上调用该方法。

5. spring 的事务

事务是一个不可分割操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。

Spring 的事务管理提供了两种方式：编程式事务管理和声明式事务管理。

1.Spring 事务管理的 API

1、 PlatformTransactionManager 接口

平台事务管理器接口

Spring 事务策略是通过 PlatformTransactionManager 接口体现的 ,该接口是 Spring 事务策略的**核心**。该接口的源代码如下：

```
public interface PlatformTransactionManager {
    //平台无关的获得事务的方法
    TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException;

    //平台无关的事务提交方法
    void commit(TransactionStatus status) throws TransactionException;

    //平台无关的事务回滚方法
    void rollback(TransactionStatus status) throws TransactionException;
}
```

在 PlatformTransactionManager 接口内，包含一个 getTransaction（TransactionDefinition definition)方法 ,该方法根据一个 TransactionDefinition 参数 ,返回一个 TransactionStatus 对象。TransactionStatus 对象表示一个事务，该事务可能是一个新的事务，也可能是一个已经存在的事务对象，这由 TransactionDefinition 所定义的事务规则所决定。

2.TransactionDefinition 接口

TransactionDefinition 接口包含与事务属性相关的方法，如下所示：

```
public interface TransactionDefinition{

    int getIsolationLevel();

    int getPropagationBehavior();

    int getTimeout();

    boolean isReadOnly();

}
```

TransactionDefinition 接口定义的事务规则包括 :事务隔离级别、事务传播行为、事务超时、

事务的只读属性和事务的回滚规则。

（1）事务的隔离级别

所谓事务的隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- TransactionDefinition.ISOLATION_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，该级别就是 TransactionDefinition.ISOLATION_READ_COMMITTED；
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据，该级别不能防止脏读和不可重复读，因此很少使用该隔离级别；
- TransactionDefinition.ISOLATION_READ_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。
- TransactionDefinition.ISOLATION_REPEATABLE_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。
- TransactionDefinition.ISOLATION_SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是，这将严重影响程序的性能，通常情况下也不会用到该级别。

（2）事务的传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。TransactionDefinition 接口定义了如下几个表示传播行为的常量：

- TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。
- TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

- TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- TransactionDefinition.PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

这里需要指出的是，以 PROPAGATION_NESTED 启动的事务内嵌于外部事务中（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。另外，外部事务的回滚也会导致嵌套子事务的回滚。

(3).事务超时

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

(4). 事务的只读属性

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。所谓事务性资源就是指那些被事务管理的资源，比如数据源、 JMS 资源，以及自定义的事务性资源等等。如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。在 TransactionDefinition 接口中，以 boolean 类型来表示该事务是否只读。

(5). 事务的回滚规则

通常情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常），则默认将回滚事务。如果没有抛出任何异常，或者抛出了已检查异常，则仍然提交事务。这通常也是大多数开发者希望的处理方式，也是 EJB 中的默认处理方式。但是，我们可以根据需要人为控制事务在抛出某些未检查异常时任然提交事务，或者在抛出某些已检查异常时回滚事务。

3、TransactionStatus 接口

PlatformTransactionManager.getTransaction(…) 方法返回一个 TransactionStatus 对象，该对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务）。TransactionStatus 接口提供了一个简单的控制事务执行和查询事务状态的方法。该接口的源代码如下：

```
public  interface TransactionStatus{

   boolean isNewTransaction();

   void setRollbackOnly();

   boolean isRollbackOnly();

}
```

Spring 的编程式事务

在 Spring 出现以前，编程式事务管理对基于 POJO 的应用来说是唯一选择。用过 Hibernate 的人都知道，我们需要在代码中显式调用 beginTransaction()、commit()、rollback() 等事务管理相关的方法，这就是编程式事务管理。通过 Spring 提供的事务管理 API，我们

可以在代码中灵活控制事务的执行。在底层，Spring 仍然将事务操作委托给底层的持久化框架来执行。

1、基于底层 API 的编程式事务管理

下面给出一个基于底层 API 的编程式事务管理的示例，基于 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus 三个核心接口，我们完全可以通过编程的方式来进行事务管理。

```
public class BankServiceImpl implements BankService {
    private BankDao bankDao;
    private TransactionDefinition txDefinition;
    private PlatformTransactionManager txManager;
    public boolean transfer(Long fromId， Long toId， double amount) {
        // 获取一个事务
        TransactionStatus txStatus = txManager.getTransaction(txDefinition);
        boolean result = false;
        try {
            result = bankDao.transfer(fromId， toId， amount);
            txManager.commit(txStatus);     // 事务提交
        } catch (Exception e) {
            result = false;
            txManager.rollback(txStatus);       // 事务回滚
            System.out.println("Transfer Error!");
        }
        return result;
    }
}
```

如上所示，我们在 BankServiceImpl 类中增加了两个属性：一个是 TransactionDefinition 类型的属性，它用于定义事务的规则；另一个是 PlatformTransactionManager 类型的属性，

用于执行事务管理操作。如果一个业务方法需要添加事务，我们首先需要在方法开始执行前调用 PlatformTransactionManager.getTransaction(…) 方法启动一个事务；创建并启动了事务之后，便可以开始编写业务逻辑代码，然后在适当的地方执行事务的提交或者回滚。

2、基于 TransactionTemplate 的编程式事务管理

当然，除了可以使用基于底层 API 的编程式事务外，还可以使用基于 TransactionTemplate 的编程式事务管理。通过上面的示例可以发现，上述事务管理的代码散落在业务逻辑代码中，破坏了原有代码的条理性，并且每一个业务方法都包含了类似的启动事务、提交/回滚事务的样板代码。Spring 也意识到了这些，并提供了简化的方法，这就是 Spring 在数据访问层非常常见的 **模板回调模式**。

```
public class BankServiceImpl implements BankService {
    private BankDao bankDao;
    private TransactionTemplate transactionTemplate;
    public boolean transfer(final Long fromId , final Long toId , final double amount) {
        return (Boolean) transactionTemplate.execute(new TransactionCallback(){
            public Object doInTransaction(TransactionStatus status) {
                Object result;
                try {
                    result = bankDao.transfer(fromId , toId , amount);
                } catch (Exception e) {
                    status.setRollbackOnly();
                    result = false;
                    System.out.println("Transfer Error!");
                }
```

```
                return result;
            }
        });
    }
}
```

相应的配置文件如下所示：

```
<bean id="bankService" class="footmark.spring.core.tx.programmatic.template.BankServiceImpl">

    <property name="bankDao" ref="bankDao"/>

    <property name="transactionTemplate" ref="transactionTemplate"/>

</bean>
```

TransactionTemplate 的 execute() 方法有一个 TransactionCallback 类型的参数，该接口中定义了一个 doInTransaction() 方法，通常我们以匿名内部类的方式实现 TransactionCallback 接口，并在其 doInTransaction() 方法中书写业务逻辑代码。这里可以使用默认的事务提交和回滚规则，这样在业务代码中就不需要显式调用任何事务管理的 API。doInTransaction() 方法有一个 TransactionStatus 类型的参数，我们可以在方法的任何位置调用该参数的 setRollbackOnly() 方法将事务标识为回滚的，以执行事务回滚。

Spring 的声明式事务管理

Spring 的声明式事务管理是建立在 Spring AOP 机制之上的，其本质是对目标方法前后进行拦截，并在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码

中掺杂事务管理的代码，只需在配置文件中作相关的事务规则声明（或通过等价的基于标注的方式），便可以将事务规则应用到业务逻辑中。总的来说，声明式事务得益于 Spring IoC 容器 和 Spring AOP 机制的支持 :IoC 容器为声明式事务管理提供了基础设施 ,使得 Bean 对于 Spring 框架而言是可管理的；而由于事务管理本身就是一个典型的横切逻辑（正是 AOP 的用武之地），因此 Spring AOP 机制是声明式事务管理的直接实现者。

Spring 2.x 引入了 <tx> 命名空间，结合使用 <aop> 命名空间，带给开发人员配置声明式事务的全新体验，配置变得更加简单和灵活。总的来说，开发者只需基于<tx>和<aop>命名空间在 XML 中进行简答配置便可实现声明式事务管理。下面基于<tx>使用 Hibernate 事务管理的配置文件：

```
<!-- 配置 DataSourece -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/ssh</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
```

```
    <property name="password">
        <value>root</value>
    </property>
</bean>

<!-- 配置 sessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <!-- 数据源的设置 -->
    <property name="dataSource" ref="dataSource" />
    <!-- 用于持久化的实体类类列表 -->
    <property name="annotatedClasses">
        <list>
            <value>cn.edu.tju.rico.model.entity.User</value>
            <value>cn.edu.tju.rico.model.entity.Log</value>
        </list>
    </property>
    <!-- Hibernate 的配置 -->
    <property name="hibernateProperties">
        <props>
            <!-- 方言设置    -->
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
```

```
                <!-- 显示 sql -->
                <prop key="hibernate.show_sql">true</prop>
                <!-- 格式化 sql -->
                <prop key="hibernate.format_sql">true</prop>
                <!-- 自动创建/更新数据表 -->
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
</bean>

<!-- 配置 TransactionManager -->
<bean id="txManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<!-- 配置事务增强处理的切入点，以保证其被恰当的织入 -->
<aop:config>
    <!-- 切点 -->
    <aop:pointcut expression="execution(* cn.edu.tju.rico.service.impl.*.*(..))"
        id="bussinessService" />
    <!-- 声明式事务的切入 -->
```

```
    <aop:advisor advice-ref="txAdvice" pointcut-ref="bussinessService" />
</aop:config>

<!-- 由 txAdvice 切面定义事务增强处理 -->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <!-- get 打头的方法为只读方法,因此将 read-only 设为 true -->
        <tx:method name="get*" read-only="true" />
        <!-- 其他方法为读写方法,因此将 read-only 设为 false -->
        <tx:method name="*" read-only="false" propagation="REQUIRED"
                isolation="DEFAULT" />
    </tx:attributes>
</tx:advice>
```

事实上，**Spring 配置文件中关于事务的配置总是由三个部分组成，即：DataSource、TransactionManager 和代理机制三部分，无论哪种配置方式，一般变化的只是代理机制这部分。**其中，DataSource、TransactionManager 这两部分只是会根据数据访问方式有所变化，比如使用 hibernate 进行数据访问时，DataSource 实际为 SessionFactory，TransactionManager 的实现为 HibernateTransactionManager。

2、基于 @Transactional 的声明式事务管理

除了基于命名空间的事务配置方式，Spring 还引入了基于 Annotation 的方式，具体主要涉及@Transactional 标注。@Transactional 可以作用于接口、接口方法、类以及类方法上：当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性；当作用于方

法上时，该标注来覆盖类级别的定义。如下所示：

```
@Transactional(propagation = Propagation.REQUIRED)
public boolean transfer(Long fromId , Long toId , double amount) {
    return bankDao.transfer(fromId , toId , amount);
}
```

Spring 使用 BeanPostProcessor 来处理 Bean 中的标注，因此我们需要在配置文件中作如下声明来激活该后处理 Bean，如下所示：

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

与前面相似，transaction-manager、datasource 和 sessionFactory 的配置不变，只需将基于<tx>和<aop>命名空间的配置更换为上述配置即可。

3、Spring 声明式事务的本质

就 Spring 声明式事务而言，无论其基于 <tx> 命名空间的实现还是基于 @Transactional 的实现，其本质都是 Spring AOP 机制的应用：即通过以@Transactional 的方式或者 XML 配置文件的方式向业务组件中的目标业务方法插入事务增强处理并生成相应的代理对象供应用程序(客户端)使用从而达到无污染地添加事务的目的。

6. Spring 数据库连接池的配置

```
<!-- 配置C3P0连接池================================ -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql:///spring4_day03"/>
    <property name="user" value="root"/>
    <property name="password" value="abc"/>
</bean>

<!-- 配置Spring的JDBC的模板========================== -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

7. mybatis 中的#和$的区别

#{}：占位符号，好处防止 sql 注入

${}：sql 拼接符号

动态 SQL 是 mybatis 的强大特性之一，也是它优于其他 ORM 框架的一个重要原因。mybatis 在对 sql 语句进行预编译之前，会对 sql 进行动态解析，解析为一个 BoundSql 对象，也是在此处对动态 SQL 进行处理的。在动态 SQL 解析阶段， #{ } 和 ${ } 会有不同的表现。

1.例如，Mapper.xml 中如下的 sql 语句：

select * from user where name = #{name};

动态解析为：

select * from user where name = ?;

一个 #{ } 被解析为一个参数占位符 ？ 。

而${ } 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换。

2.例如，Mapper.xml 中如下的 sql：

select * from user where name = ${name};

当我们传递的参数为 "Jack" 时，上述 sql 的解析为：

select * from user where name = "Jack";

预编译之前的 SQL 语句已经不包含变量了，完全已经是常量数据了。 综上所得， ${ } 变量的替换阶段是在动态 SQL 解析阶段，而 #{ }变量的替换是在 DBMS 中。

用法：

1、能使用 #{ } 的地方就用 #{ }

首先这是为了性能考虑的，相同的预编译 sql 可以重复利用。其次，${ } 在预编译之前已

经被变量替换了，这会存在 sql 注入问题。例如，如下的 sql：

select * from ${tableName} where name = #{name} ；

假如，我们的参数 tableName 为 user; delete user; --，那么 SQL 动态解析阶段之后，预编译之前的 sql 将变为：

select * from user; delete user; -- where name = ?;

-- 之后的语句将作为注释，不起作用，因此本来的一条查询语句偷偷的包含了一个删除表数据的 SQL。

2. 表名作为变量时，必须使用 ${ }

这是因为，表名是字符串，使用 sql 占位符替换字符串时会带上单引号 ''，这会导致 sql 语法错误，例如：

select * from #{tableName} where name = #{name};

预编译之后的 sql 变为：

select * from ? where name = ?;

假设我们传入的参数为 tableName = "user"，name = "Jack"，那么在占位符进行变量替换后，sql 语句变为：

select * from 'user' where name='Jack';

上述 sql 语句是存在语法错误的，表名不能加单引号 ''（注意，反引号 ``是可以的）。

sql 的预编译

1. 定义：

sql 预编译指的是数据库驱动在发送 sql 语句和参数给 DBMS 之前对 sql 语句进行编译，这样 DBMS 执行 sql 时，就不需要重新编译

2. 为什么需要预编译

JDBC 中使用对象 PreparedStatement 来抽象预编译语句，使用预编译。预编译阶段可以优化 sql 的执行。预编译之后的 sql 多数情况下可以直接执行，DBMS 不需要再次编译，越复杂的 sql，编译的复杂度将越大，预编译阶段可以合并多次操作为一个操作。预编译语句对象可以重复利用。把一个 sql 预编译后产生的 PreparedStatement 对象缓存下来，下次对于同一个 sql ,可以直接使用这个缓存的 PreparedState 对象。mybatis 默认情况下，将对所有的 sql 进行预编译。

多线程与并发

8. ReentrantLock 的底层实现

内部结构：

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

默认构造器初始化为 NonfairSync 对象，即非公平锁，而带参数的构造器可以指定使用公平锁和非公平锁。

Sync 扩展了 AbstractQueuedSynchronizer。

NonfairSync

1. **lock()**

   lock()源码如下

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);
}
```

首先用一个 CAS 操作，判断 state 是否是 0（表示当前锁未被占用），如果是 0 则把它置为 1，并且设置当前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS 操作只能保证一个线程操作成功，剩下的只能乖乖的去排队啦。

“非公平”即体现在这里，如果占用锁的线程刚释放锁，state 置为 0，而排队等待锁的线程还未唤醒时，新来的线程就直接抢占了该锁，那么就“插队”了。

若当前有三个线程去竞争锁，假设线程 A 的 CAS 操作成功了，拿到了锁开开心心的返回了，那么线程 B 和 C 则设置 state 失败，走到了 else 里面。我们往下看 acquire。

acquire(arg)

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

1. 第一步。尝试去获取锁。如果尝试获取锁成功，方法直接返回。

tryAcquire(arg)

```
复制代码
final boolean nonfairTryAcquire(int acquires) {
    //获取当前线程
    final Thread current = Thread.currentThread();
    //获取state变量值
    int c = getState();
    if (c == 0) { //没有线程占用锁
        if (compareAndSetState(0, acquires)) {
            //占用锁成功,设置独占线程为当前线程
            setExclusiveOwnerThread(current);
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) { //当前线程已经占用该锁
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 更新state值为新的重入次数
        setState(nextc);
        return true;
    }
    //获取锁失败
    return false;
}
```

非公平锁 tryAcquire 的流程是：检查 state 字段，若为 0，表示锁未被占用，那么尝试占用，若不为 0，检查当前锁是否被自己占用，若被自己占用，则更新 state 字段，表示重入锁的次数。如果以上两点都没有成功，则获取锁失败，返回 false。

2. 第二步，入队。由于上文中提到线程 A 已经占用了锁，所以 B 和 C 执行 tryAcquire 失败，并且入等待队列。如果线程 A 拿着锁死死不放，那么 B 和 C 就会被挂起。

先看addWaiter(Node.EXCLUSIVE)

复制代码

```
/**
 * 将新节点和当前线程关联并且入队列
 * @param mode 独占/共享
 * @return 新节点
 */
private Node addWaiter(Node mode) {
    //初始化节点,设置关联线程和模式(独占 or 共享)
    Node node = new Node(Thread.currentThread(), mode);
    // 获取尾节点引用
    Node pred = tail;
    // 尾节点不为空,说明队列已经初始化过
    if (pred != null) {
        node.prev = pred;
        // 设置新节点为尾节点
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    // 尾节点为空,说明队列还未初始化,需要初始化head节点并入队新节点
    enq(node);
    return node;
}
```

B、C 线程同时尝试入队列，由于队列尚未初始化，tail==null，故至少会有一个线程会走到 enq(node)。我们假设同时走到了 enq(node)里。

```
/**
 * 初始化队列并且入队新节点
 */
private Node enq(final Node node) {
    //开始自旋
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            // 如果tail为空,则新建一个head节点,并且tail指向head
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            // tail不为空,将新节点入队
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

这里体现了经典的自旋+CAS组合来实现非阻塞的原子操作。由于compareAndSetHead的实现使用了 unsafe 类提供的 CAS 操作，所以只有一个线程会创建 head 节点成功。假设线程 B 成功，之后 B、C 开始第二轮循环，此时 tail 已经不为空，两个线程都走到 else 里面。假设 B 线程 compareAndSetTail 成功，那么 B 就可以返回了，C 由于入队失败还需要第三轮循环。最终所有线程都可以成功入队。

3. 第三步，挂起。B 和 C 相继执行 acquireQueued(final Node node, int arg)。这个方法让已经入队的线程尝试获取锁，若失败则会被挂起。

```
/**
 * 已经入队的线程尝试获取锁
 */
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true; //标记是否成功获取锁
    try {
        boolean interrupted = false; //标记线程是否被中断过
        for (;;) {
            final Node p = node.predecessor(); //获取前驱节点
            //如果前驱是head,即该结点已成老二，那么便有资格去尝试获取锁
            if (p == head && tryAcquire(arg)) {
                setHead(node); // 获取成功,将当前节点设置为head节点
                p.next = null; // 原head节点出队,在某个时间点被GC回收
                failed = false; //获取成功
                return interrupted; //返回是否被中断过
            }
            // 判断获取失败后是否可以挂起,若可以则挂起
            if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                // 线程若被中断,设置interrupted为true
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

code 里的注释已经很清晰的说明了 acquireQueued 的执行流程。假设 B 和 C 在竞争锁的过程中 A 一直持有锁，那么它们的 tryAcquire 操作都会失败，因此会走到第 2 个 if 语句中。我们再看下 shouldParkAfterFailedAcquire 和 parkAndCheckInterrupt 都做了哪些事吧。

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    //前驱节点的状态
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        // 前驱节点状态为signal,返回true
        return true;
    // 前驱节点状态为CANCELLED
    if (ws > 0) {
        // 从队尾向前寻找第一个状态不为CANCELLED的节点
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 将前驱节点的状态设置为SIGNAL
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

/**
 * 挂起当前线程,返回线程中断状态并重置
 */
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

线程入队后能够挂起的前提是，它的前驱节点的状态为 SIGNAL，它的含义是“Hi，前面的兄弟，如果你获取锁并且出队后，记得把我唤醒！”。所以 shouldParkAfterFailedAcquire 会先判断当前节点的前驱是否状态符合要求，若符合则返回 true，然后调用 parkAndCheckInterrupt，将自己挂起。如果不符合，再看前驱节点是否>0(CANCELLED)，若是那么向前遍历直到找到第一个符合要求的前驱，若不是则将前驱节点的状态设置为 SIGNAL。

整个流程中，如果前驱结点的状态不是 SIGNAL，那么自己就不能安心挂起，需要去找个安心的挂起点，同时可以再尝试下看有没有机会去尝试竞争锁。

2.unlock()

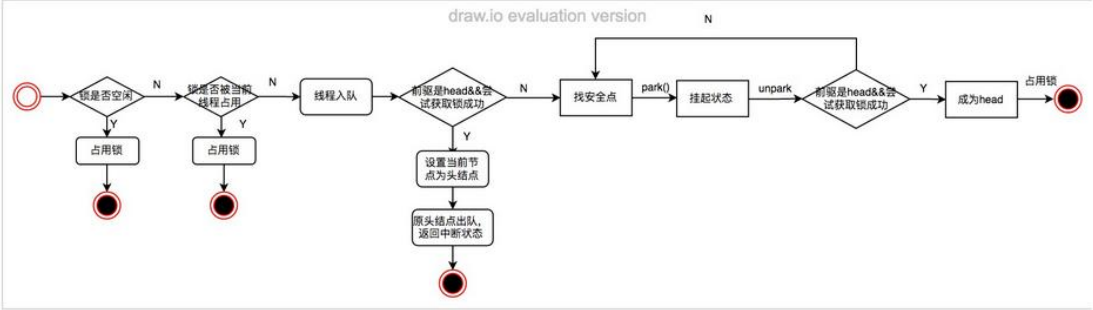
```
public void unlock() {
    sync.release(1);
}

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

流程大致为先尝试释放锁，若释放成功，那么查看头结点的状态是否为 SIGNAL，如果是则唤醒头结点的下个节点关联的线程，如果释放失败那么返回 false 表示解锁失败。这里我们也发现了，每次都只唤起头结点的下一个节点关联的线程。

```
protected final boolean tryRelease(int releases) {
    // 计算释放后state值
    int c = getState() - releases;
    // 如果不是当前线程占用锁,那么抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        // 锁被重入次数为0,表示释放成功
        free = true;
        // 清空独占线程
        setExclusiveOwnerThread(null);
    }
    // 更新state值
    setState(c);
    return free;
}
```

这里入参为 1。tryRelease 的过程为：当前释放锁的线程若不持有锁，则抛出异常。若持有锁，计算释放后的 state 值是否为 0，若为 0 表示锁已经被成功释放，并且则清空独占线程，最后更新 state 值，返回 free。



fairSync

公平锁和非公平锁不同之处在于，公平锁在获取锁的时候，不会先去检查 state 状态，而是直接执行 aqcuire(1)。

公平锁的大致逻辑与非公平锁是一致的，不同的地方在于有了!hasQueuedPredecessors()这个判断逻辑，即便 state 为 0，也不能贸然直接去获取，要先去看有没有还在排队的线程，若没有，才能尝试去获取，做后面的处理。反之，返回 false，获取失败。

hasQueuedPredecessors()

```
public final boolean hasQueuedPredecessors() {
    Node t = tail; // 尾结点
    Node h = head;//头结点
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());//判断是否有排在自己之前的线程
}
```

1. **h !=t && (s = h.next) == null**，这个逻辑成立的一种可能是 head 指向头结点，tail 此时还

为 null。考虑这种情况：当其他某个线程去获取锁失败，需构造一个结点加入同步队列中（假设此时同步队列为空），在添加的时候，需要先创建一个无意义傀儡头结点（在 AQS 的 enq 方法中，这是个自旋 CAS 操作），有可能在将 head 指向此傀儡结点完毕之后，还未将 tail 指向此结点。很明显，此线程时间上优于当前线程，所以，返回 true，表示有等待中的线程且比自己来的还早。

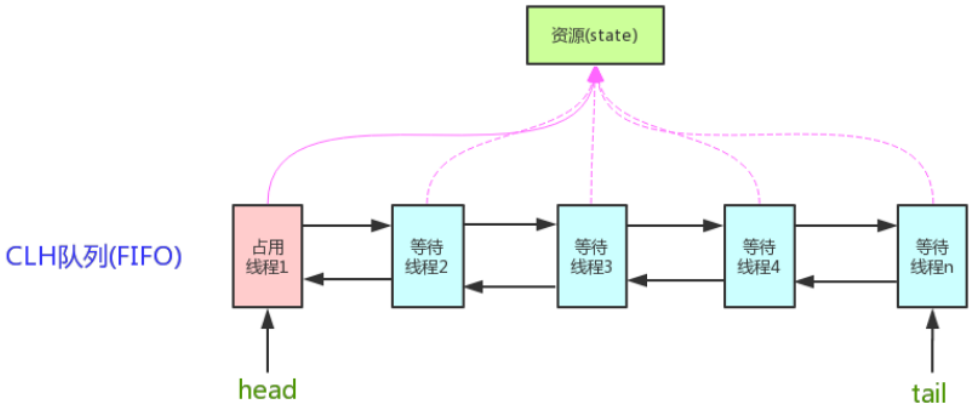
2.**h != t && (s = h.next) != null && s.thread != Thread.currentThread()**。同步队列中已经有若干排队线程且当前线程不是队列的老二结点，此种情况会返回 true。假如没有 s.thread !=Thread.currentThread()这个判断的话，会怎么样呢？若当前线程已经在同步队列中是老二结点（头结点此时是个无意义的傀儡结点),此时持有锁的线程释放了资源，唤醒老二结点线程，老二结点线程重新 tryAcquire（此逻辑在 AQS 中的 acquireQueued 方法中），又会调用到 hasQueuedPredecessors，不加 s.thread !=Thread.currentThread()这个判断的话，返回值就为 true，导致 tryAcquire 失败。

9. AQS

谈到并发，不得不谈 ReentrantLock；而谈到 ReentrantLock，不得不谈 AbstractQueuedSynchronizer（AQS）!

类如其名，抽象的队列式的同步器，AQS 定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的 ReentrantLock/Semaphore/CountDownLatch...。

框架：



它维护了一个 volatile int state（代表共享资源）和一个 FIFO 线程等待队列。

AQS 定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如 ReentrantLock）和 Share（共享，多个线程可同时执行，如 Semaphore/CountDownLatch）。

自定义同步器实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到 condition 才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回 true，失败则返回 false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回 true，失败则返回 false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回 true，否则返回 false。

以 ReentrantLock 为例，state 初始化为 0，表示未锁定状态。A 线程 lock()时，会调用 tryAcquire()独占该锁并将 state+1。此后，其他线程再 tryAcquire()时就会失败，直到 A 线程 unlock()到 state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是可以重复获取此锁的（state 会累加），这就是可重入的概念。但要注意，获取

多少次就要释放多么次，这样才能保证 state 是能回到零态的。

再以 CountDownLatch 以例，任务分为 N 个子线程去执行，state 也初始化为 N（注意 N 要与线程个数一致）。这 N 个子线程是并行执行的，每个子线程执行完后 countDown()一次，state 会 CAS 减 1。等到所有子线程都执行完后(即 state=0)，会 unpark()主调用线程，然后主调用线程就会从 await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared 中的一种即可。但 AQS 也支持自定义同步器同时实现独占和共享两种方式，如 ReentrantReadWriteLock。

源码详解

1. acquire(int)

```
1 public final void acquire(int arg) {
2     if (!tryAcquire(arg) &&
3         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4         selfInterrupt();
5 }
```

函数流程如下：

1. **tryAcquire()尝试直接去获取资源，如果成功则直接返回；**
2. **addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；**
3. **acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回 true，否则返回 false。**
4. **如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断 selfInterrupt()，将中断补上。**

tryAcquire(arg)

```
final boolean nonfairTryAcquire(int acquires) {
    //获取当前线程
    final Thread current = Thread.currentThread();
    //获取state变量值
    int c = getState();
    if (c == 0) { //没有线程占用锁
        if (compareAndSetState(0, acquires)) {
            //占用锁成功,设置独占线程为当前线程
            setExclusiveOwnerThread(current);
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) { //当前线程已经占用该锁
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 更新state值为新的重入次数
        setState(nextc);
        return true;
    }
    //获取锁失败
    return false;
}
```

addWaiter(Node)

此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点。

```
private Node addWaiter(Node mode) {
    //以给定模式构造结点。mode有两种：EXCLUSIVE（独占）和SHARED（共享）
    Node node = new Node(Thread.currentThread(), mode);

    //尝试快速方式直接放到队尾。
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }

    //上一步失败则通过enq入队。
    enq(node);
    return node;
}
```

变量 waitStatus 则表示当前被封装成 Node 结点的等待状态 ,共有 4 种取值 CANCELLED、

SIGNAL、CONDITION、PROPAGATE。

- CANCELLED：值为 1，在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消该 Node 的结点，其结点的 waitStatus 为 CANCELLED，即结束状态，进入该状态后的结点将不会再变化。
- SIGNAL：值为-1，被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。说白了，就是处于唤醒状态，只要前继结点释放锁，就会通知标识为 SIGNAL 状态的后继结点的线程执行。
- CONDITION：值为-2，与 Condition 相关，该标识的结点处于等待队列中，结点的线程等待在 Condition 上，当其他线程调用了 Condition 的 signal()方法后，CONDITION 状态的结点将从等待队列转移到同步队列中，等待获取同步锁。
- PROPAGATE：值为-3，与共享模式相关，在共享模式中，该状态标识结点的线程处于可运行状态。
- 0 状态：值为 0，代表初始化状态。

AQS 在判断状态时，通过用 waitStatus>0 表示取消状态，而 waitStatus<0 表示有效状态。

acquireQueued(Node, int)

流程：

1. 结点进入队尾后，检查状态，找到安全休息点；
2. 调用 park()进入 waiting 状态，等待 unpark()或 interrupt()唤醒自己；
3. 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head 指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程 1。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;//标记是否成功拿到资源
    try {
        boolean interrupted = false;//标记等待过程中是否被中断过

        //又是一个"自旋"！
        for (;;) {
            final Node p = node.predecessor();//拿到前驱
            //如果前驱是head，即该结点已成老二，那么便有资格去尝试获取资源（可能是老大释放完资源唤醒自己的，当然也可能被interrupt了）。
            if (p == head && tryAcquire(arg)) {
                setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。
                p.next = null; // setHead中node.prev已置为null，此处再将head.next置为null，就是为了方便GC回收以前的head结点。
                failed = false;
                return interrupted;//返回等待过程中是否被中断过
            }

            //如果自己可以休息了，就进入waiting状态，直到被unpark()
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;//如果等待过程中被中断过，哪怕只有那么一次，就将interrupted标记为true
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

shouldParkAfterFailedAcquire(Node, Node)

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)
{
    int ws = pred.waitStatus;//拿到前驱的状态
    if (ws == Node.SIGNAL)
        //如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了
        return true;
    if (ws > 0) {
        /*
         * 如果前驱放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。
         * 注意：那些放弃的结点，由于被自己"加塞"到它们前边，它们相当于形成一个无引用链，稍后就会被保安大叔赶走了(GC 回收)！
         */
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        //如果前驱正常，那就把前驱的状态设置成 SIGNAL，告诉它拿完号后通知自己一下。有可能失败，人家说不定刚刚释放完呢！
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

整个流程中，如果前驱结点的状态不是 SIGNAL，那么自己就不能安心去休息，需要去找个

安心的休息点，同时可以再尝试下看有没有机会轮到自己拿号。

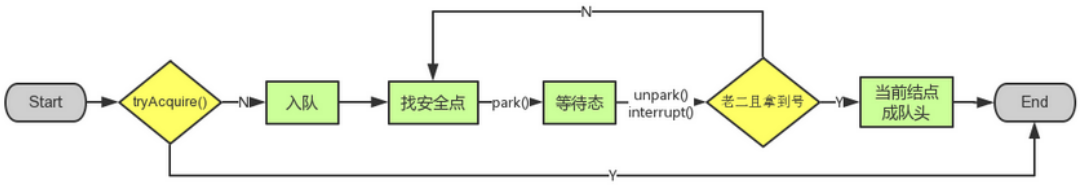
parkAndCheckInterrupt()

```
1 private final boolean parkAndCheckInterrupt() {
2     LockSupport.park(this);//调用park()使线程进入waiting状态
3     return Thread.interrupted();//如果被唤醒，查看自己是不是被中断的。
4 }
```

park()会让当前线程进入 waiting 状态。在此状态下，有两种途径可以唤醒该线程：1）被 unpark()；2）被 interrupt()。

总结下 acquire(int)的流程吧：

1. 调用自定义同步器的 tryAcquire()尝试直接去获取资源，如果成功则直接返回；
2. 没成功，则 addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被 unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回 true，否则返回 false。
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断 selfInterrupt()，将中断补上。



2. release(int)

```
1 public final boolean release(int arg) {
2     if (tryRelease(arg)) {
3         Node h = head;//找到头结点
4         if (h != null && h.waitStatus != 0)
5             unparkSuccessor(h);//唤醒等待队列里的下一个线程
6         return true;
7     }
8     return false;
9 }
```

它是根据 tryRelease()的返回值来判断该线程是否已经完成释放掉资源了。

tryRelease(int)

```
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
```

unparkSuccessor(Node)

此方法用于唤醒等待队列中下一个线程。下面是源码：

```
private void unparkSuccessor(Node node) {
    //这里，node一般为当前线程所在的结点。
    int ws = node.waitStatus;
    if (ws < 0)//置零当前线程所在的结点状态，允许失败。
        compareAndSetWaitStatus(node, ws, 0);

    Node s = node.next;//找到下一个需要唤醒的结点s
    if (s == null || s.waitStatus > 0) {//如果为空或已取消
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)//从这里可以看出，<=0的结点，都是还有效的结点。
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread);//唤醒
}
```

1.acquireShared(int)

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

- tryAcquireShared()尝试获取资源，成功则直接返回；
- 失败则通过 doAcquireShared()进入等待队列，直到获取到资源为止才返回。

doAcquireShared(int)

此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。

```
1 private void doAcquireShared(int arg) {
2     final Node node = addWaiter(Node.SHARED);//加入队列尾部
3     boolean failed = true;//是否成功标志
4     try {
5         boolean interrupted = false;//等待过程中是否被中断过的标志
6         for (;;) {
7             final Node p = node.predecessor();//前驱
8             if (p == head) {//如果到head的下一个，因为head是拿到资源的线程，此时node被唤醒，很可能是head用完资源来唤醒自己的
9                 int r = tryAcquireShared(arg);//尝试获取资源
10                if (r >= 0) {//成功
11                    setHeadAndPropagate(node, r);//将head指向自己，还有剩余资源可以再唤醒之后的线程
12                    p.next = null; // help GC
13                    if (interrupted)//如果等待过程中被打断过，此时将中断补上。
14                        selfInterrupt();
15                    failed = false;
16                    return;
17                }
18            }
19
20            //判断状态，寻找安全点，进入waiting状态，等着被unpark()或interrupt()
21            if (shouldParkAfterFailedAcquire(p, node) &&
22                parkAndCheckInterrupt())
23                interrupted = true;
24        }
25    } finally {
26        if (failed)
27            cancelAcquire(node);
28    }
29 }
```

只有线程是 head.next 时（“老二”），才会去尝试获取资源，有剩余的话还会唤醒之后的队友。

setHeadAndPropagate(Node, int)

```
1 private void setHeadAndPropagate(Node node, int propagate) {
2     Node h = head;
3     setHead(node);//head指向自己
4      //如果还有剩余量，继续唤醒下一个邻居线程
5     if (propagate > 0 || h == null || h.waitStatus < 0) {
6         Node s = node.next;
7         if (s == null || s.isShared())
8             doReleaseShared();
9     }
10 }
```

此方法在 setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合（比如还有剩余资源），还会去唤醒后继结点，毕竟是共享模式！

2.releaseShared()

此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {//尝试释放资源
        doReleaseShared();//唤醒后继结点
        return true;
    }
    return false;
}
```

独占模式下的 tryRelease()在完全释放掉资源（state=0）后，才会返回 true 去唤醒其他线程，这主要是基于独占下可重入的考量；而共享模式下的 releaseShared()则没有这种要求，共享模式实质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等待结点。

doReleaseShared()

```
private void doReleaseShared() {
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;
                unparkSuccessor(h);//唤醒后继
            }
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;
        }
        if (h == head)// head发生变化
            break;
    }
}
```

10. start 和 run 方法源码

start()：它的作用是启动一个新线程，新线程会执行相应的 run()方法。start()不能被重复调用。

run() ：run()就和普通的成员方法一样，可以被重复调用。单独调用 run()的话，会在当前线程中执行 run()，而并不会启动新线程！

如果直接调用线程类的 run()方法，这就是普通的函数调用。此时，程序中仍只有主线程一个线程，而采用 start()来调用，则不止一个线程，不仅有主线程，只有我们自己要执行的线程。即，start()方法能够异步的调用 run()方法，但是直接调用 run()方法却是同步的，无法达到多线层的目的。

Thread.java 中 start()方法的源码如下：

```
public synchronized void start() {

    // A zero status value corresponds to state "NEW".如果线程不是"NEW"，
```

则抛出异常！

```
    if (threadStatus != 0)
        throw new IllegalThreadStateException();


    // 将线程添加到ThreadGroup中,底层实现：Thread threads[]数组;
    group.add(this);

    boolean started = false;
    try {
        // 通过start0()启动线程

        start0();//private native void start0();查看open jdk

        // 设置started标记
        started = true;
    } finally {
        try {
            if (!started) {

                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
        }
    }
}
```

ThreadGroup中的add():

```
 void add(Thread t) {
        synchronized (this) {
//判断线程是否销毁
            if (destroyed) {
                throw new IllegalThreadStateException();
            }
//如果线程组为空，则新建一个大小为4的线程组
            if (threads == null) {
                threads = new Thread[4];
            } else if (nthreads == threads.length) {

                //nthreads我的理解是线程组中的线程数，当nthreads ==
```

threads.length时，就表示已经有8个线程了，再有线程加进来时threadgroup就不够放

了，就得扩容，2倍扩容类似于arraylist的扩容方式。

```
                threads = Arrays.copyOf(threads, nthreads * 2);
            }
            threads[nthreads] = t;

            // This is done last so it doesn't matter in case the

            // thread is killed不太理解，线程被杀死的操作是什么？本来被放到线
```

程组的线程被杀死后，nthreads还要++？？

```
            nthreads++;

        }
    }
```

计算机网络

11. https 中的加密算法

对称加密算法：DES、AES、IDEA、RC4、RC5、RC6 等。

非对称加密算法：RSA、DSA（数字签名用）等。

hash 算法：MD5

1.DES 加密算法

DES 加密算法是一种分组密码，以 64 位为分组对数据加密，它的密钥长度是 56 位，加密解密用同一算法。DES 加密算法是对密钥进行保密，而公开算法，包括加密和解密算法。这样，只有掌握了和发送方相同密钥的人才能解读由 DES 加密算法加密的密文数据。因此，破译 DES 加密算法实际上就是搜索密钥的编码。对于 56 位长度的密钥来说，如果用穷举法来进行搜索的话，其运算次数为 256。

随着计算机系统能力的不断发展，DES 的安全性比它刚出现时会弱得多，然而从非关键性质的实际出发，仍可以认为它是足够的。不过，DES 现在仅用于旧系统的鉴定，而更多地

选择新的加密标准。

2.AES 加密算法

AES 加密算法是密码学中的高级加密标准，该加密算法采用对称分组密码体制，密钥长度的最少支持为 128、192、256，分组长度 128 位，算法应易于各种硬件和软件实现。这种加密算法是美国联邦政府采用的区块加密标准，这个标准用来替代原先的 DES，已经被多方分析且广为全世界所使用。

AES 加密算法被设计为支持 128 / 192 / 256 位 (/32=nb)数据块大小 (即分组长度) ; 支持 128 / 192 / 256 位(/32=nk)密码长度，,在 10 进制里 ,对应 34×1038、62×1057、1.1×1077 个密钥。

3.RSA 加密算法

RSA 加密算法是目前最有影响力的公钥加密算法，并且被普遍认为是目前最优秀的公钥方案之一。RSA 是第一个能同时用于加密和数宇签名的算法，它能够抵抗到目前为止已知的所有密码攻击，已被 ISO 推荐为公钥数据加密标准。RSA 加密算法基于一个十分简单的数论事实 : 将两个大素数相乘十分容易，但那时想要，但那时想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥。

4.MD5 加密算法

MD5 为计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护。对 MD5 加密算法简要的叙述可以为：MD5 以 512 位分组来处理输入的信息，且每一分组又被划分为 16 个 32 位子分组，经过了一系列的处理后，算法的输出由四个 32 位分组组成，将这四个 32 位分组级联后将生成一个 128 位散列值。

MD5 被广泛用于各种软件的密码认证和钥匙识别上。MD5 用的是哈希函数，它的典型应用是对一段信息产生信息摘要，以防止被篡改。MD5 的典型应用是对一段 Message 产生

fingerprin 指纹，以防止被“篡改”。如果再有一个第三方的认证机构，用 MD5 还可以防止文件作者的“抵赖”，这就是所谓的数字签名应用。MD5 还广泛用于操作系统的登陆认证上，如 UNIX、各类 BSD 系统登录密码、数字签名等诸多方。

12. TCP 序列号回绕与解决

如果序列号每次交换都会自增 1，而连接又长时间存在且高速交换报文时，总有一个时刻序列号的长度会超过 32 位（2^32 - 1），此时序列号就会重置为 0，此时的序列号就会从 0 重新开始自增 1。这就是 TCP 序列号回绕问题。

问题发生了就要解决，那么如果解决这个问题呢？这就要用到 TCP 头部中的额外选项——时间戳选项了。当使用时间戳选项时，发送方将一个 32 位的数值填充到时间戳数值字段（称作 TSV 或 TSval）作为时间戳选项的第一个部分；而接收方则将收到的时间戳数值原封不动的填充至第二部分的时间戳回显重试字段（称作 TSER 或 TSecr），如果这个报文属于客户端 SYN 握手包，那么由于不知道服务端时间，第二部分的时间戳回显重试字段将被置为 0。由于包含了时间戳选项，TCP 头部的长度将会增长 10 字节（8 个字节用于保存 2 个时间戳数值，而另两个字节用于指明选项的数值与长度）。

[RFC1323]推荐发送方如果启用时间戳选项，那么每秒钟至少将时间戳选项加 1。

如果真的出现这样序列号相同的报文段，那么直接丢弃掉与最近接收到的报文段时间戳相差最远的那个报文段即可。

13. TCP 最大报文长度

TCP 封装在 IP 内，IP 数据报最大长度 65535 ，头部最小 20，TCP 头部长度最小 20，所以最大封装数据长度为 65535-20-20=65495

java 基础

14. String 类为什么是 final 的？

主要是为了“效率” 和 “安全性” 的缘故。若 String 允许被继承, 由于它的高度被使用率, 可能会降低程序的性能，所以 String 被定义成 final。

15. String 的底层实现

1.实现接口

public final class String

implements java.io.Serializable, Comparable<String>, CharSequence {

}

- java.io.Serializable

  这个序列化接口没有任何方法和域，仅用于标识序列化的语意。

- Comparable<String>

  这个接口只有一个 compareTo(T 0)接口，用于对两个实例化对象比较大小。

- CharSequence

  这个接口是一个只读的字符序列。包括 length(), charAt(int index), subSequence(int start, int end)这几个 API 接口，值得一提的是，StringBuffer 和 StringBuild 也是实现了改接口。

2.主要变量

/** The value is used for character storage. */

```
private final char value[];

/** Cache the hash code for the string */

private int hash; // Default to 0

public static final Comparator<String> CASE_INSENSITIVE_ORDER

    = new CaseInsensitiveComparator();
```

可以看到 ,value[]是存储 String 的内容的 ,即当使用 String str = "abc";的时候 ,本质上 ,"abc" 是存储在一个 char 类型的数组中的。

而 hash 是 String 实例化的 hashcode 的一个缓存。因为 String 经常被用于比较，比如在 HashMap 中。如果每次进行比较都重新计算 hashcode 的值的话，那无疑是比较麻烦的，而保存一个 hashcode 的缓存无疑能优化这样的操作。

最后，这个 CASE_INSENSITIVE_ORDER 在下面内部类中会说到，其根本就是持有一个静态内部类，用于忽略大小写得比较两个字符串。

3.内部类

再 String 只有一个内部类，那就是

```
private static class CaseInsensitiveComparator
            implements Comparator<String>, java.io.Serializable {
        // use serialVersionUID from JDK 1.2.2 for interoperability
        private static final long serialVersionUID = 8575799808933029326L;

        public int compare(String s1, String s2) {
            int n1 = s1.length();
            int n2 = s2.length();
            int min = Math.min(n1, n2);
            for (int i = 0; i < min; i++) {
                char c1 = s1.charAt(i);
                char c2 = s2.charAt(i);
                if (c1 != c2) {
                    c1 = Character.toUpperCase(c1);
                    c2 = Character.toUpperCase(c2);
```

```
                if (c1 != c2) {
                    c1 = Character.toLowerCase(c1);
                    c2 = Character.toLowerCase(c2);
                    if (c1 != c2) {
                        // No overflow because of numeric promotion
                        return c1 - c2;
                    }
                }
            }
        }
        return n1 - n2;
    }

    /** Replaces the de-serialized object. */
    private Object readResolve() { return CASE_INSENSITIVE_ORDER; }
}
```

这里有一个疑惑，在 String 中已经有了一个 compareTo 的方法，为什么还要有一个 CaseInsensitiveComparator 的内部静态类呢？

其实这一切都是为了代码复用。

首先看一下这个类就会发现，其实这个比较和 compareTo 方法也是有差别的，这个方法在比较时是忽略大小写的。而且这是一个单例，可以简单得用它来比较两个 String，因为 String 类提供一个变量：CASE_INSENSITIVE_ORDER 来持有这个内部类，这样当要比较两个 String 时可以通过这个变量来调用。

其次，可以看到 String 类中提供的 compareToIgnoreCase 方法其实就是调用这个内部类里面的方法实现的。这就是代码复用的一个例子。

4.方法

1.substring（）：返回字符串中一个子串，看最后一行可以发现，其实就是指定头尾，然后构造一个新的字符串。

2. concat（）：将 str 拼接到当前字符串后面，通过代码也可以看出其实就是建一个新的字符串。

3. indexOf（）：实现找到某个子串在当前字符串的起始位置，若没找到，则返回-1。

4. startsWith（）：判断当前字符串是否以某一段其他字符串开始的，和其他字符串比较方法一样，其实就是通过一个 while 来循环比较。

5. regionMatches（）：比较该字符串和其他一个字符串从分别指定地点开始的 n 个字符是否相等。看代码可知道，其原理还是通过一个 while 去循环对应的比较区域进行判断，但在比较之前会做判定，判定给定参数是否越界。

6. public boolean contentEquals(CharSequence cs)：这个主要是用来比较 String 和 StringBuffer 或者 StringBuild 的内容是否一样。可以看到传入参数是 CharSequence ，这也说明了 StringBuffer 和 StringBuild 同样是实现了 CharSequence。源码中先判断参数是从哪一个类实例化来的，再根据不同的情况采用不同的方案，不过其实大体都是采用上面那个 for 循环的方式来进行判断两字符串是否内容相同。

7. hashCode()和 equals()两个方法比较重要且有所关系就放一起了，equals()是 string 能成为广泛用于 Map[key,value]中 key 的关键所在。

8. 知道了 String 其实内部是通过 char[]实现的，那么就不难发现 length()，isEmpty()，charAt()这些方法其实就是在内部调用数组的方法。

设计模式

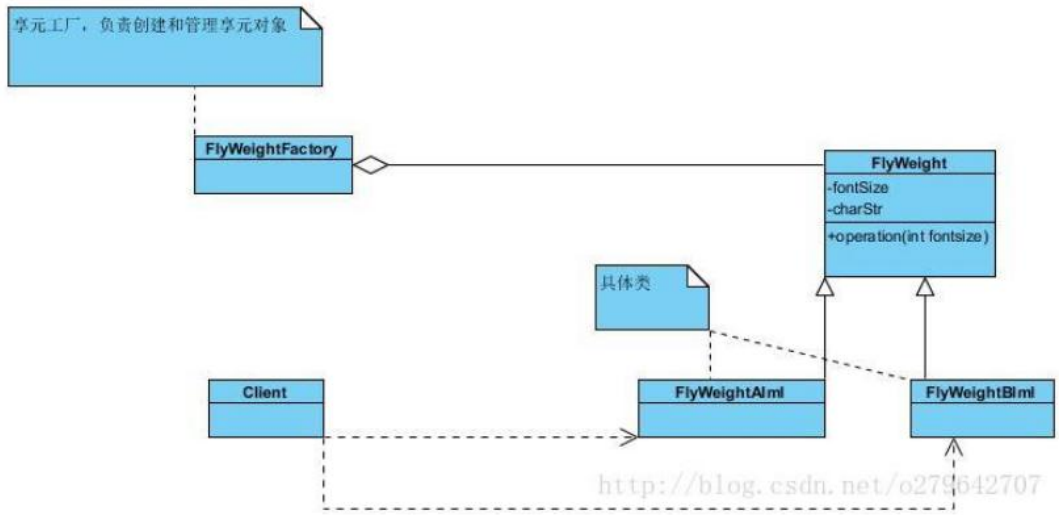
16. 享元模式

享元模式（Flyweight Pattern）：以共享的方式高效的支持大量的细粒度对象。通过复用内

存中已存在的对象，降低系统创建对象实例的性能消耗。

在面向对象中，大量细粒度对象的创建、销毁及存储所造成的资源和性能上的损耗，可能会在系统运行时形成瓶颈。那么该如何避免产生大量的细粒度对象，同时又不影响系统使用面向对象的方式进行操作呢？享元模式提供了一个比较好的解决方案。

uml类图：



享元模式的角色

抽象享元类（Flyweight）

它是所有具体享元类的超类。为这些类规定出需要实现的公共接口,那些需要外蕴状态(Exte的操作可以通过方法的参数传入。抽象享元的接口使得享元变得可能，但是并不强制子类实行共享，因此并非所有的享元对象都是可以共享的。

具体享元类(FlyWeightAIml，FlyWeightBIml)

具体享元类实现了抽象享元类所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。有时候具体享元类又称为单纯具体享元类，因为复合享元类是由单纯具体享元角色通过复合而成的。

享元工厂类(FlyweightFactoiy)

享元工厂类负责创建和管理享元对象。当一个客户端对象请求一个享元对象的时候，享元工厂需要检查系统中是否已经有一个符合要求的享元对象，如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有适当的享元对象的话，享元工厂角色就应当创建一个新的合适的享元对象。

客户类(Client)

客户类需要自行存储所有享元对象的外蕴状态。

适用场景：

当系统中某个对象类型的实例较多的时候；

当系统设计时候，对象实例真正有区别的分类很少，例如对于拼音，如果对每个字母都初始化一个对象实例的话，这样实例就太多了。使用享元模式只需要提前初始化基本拼音，就可以任意进行组装成不同的拼音。

引用个例子：

享元模式在一般的项目开发中并不常用，而是常常应用于系统底层的开发，以便解决系统的性能问题。

Java 和.Net 中的 String 类型就是使用了享元模式。如果在 Java 或者.NET 中已经创建了一个字符串对象 s1，那么下次再创建相同的字符串 s2 的时候，系统只是把 s2 的引用指向 s1 所引用的具体对象，这就实现了相同字符串在内存中的共享。如果每次执行 s1=“abc”操作的时候，都创建一个新的字符串对象的话，那么内存的开销会很大。