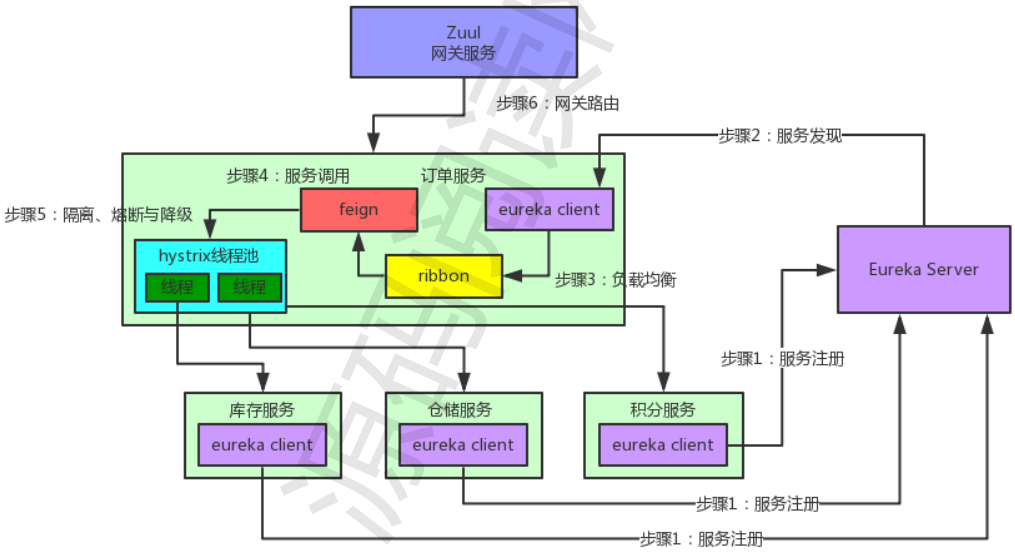
> 银盛支付

1. 说说springcloud的工作原理



springcloud由以下几个核心组件构成：

Eureka：各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里

Ribbon：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台

Feign：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求

Hystrix：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题

Zuul：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

2. 用什么组件发请求

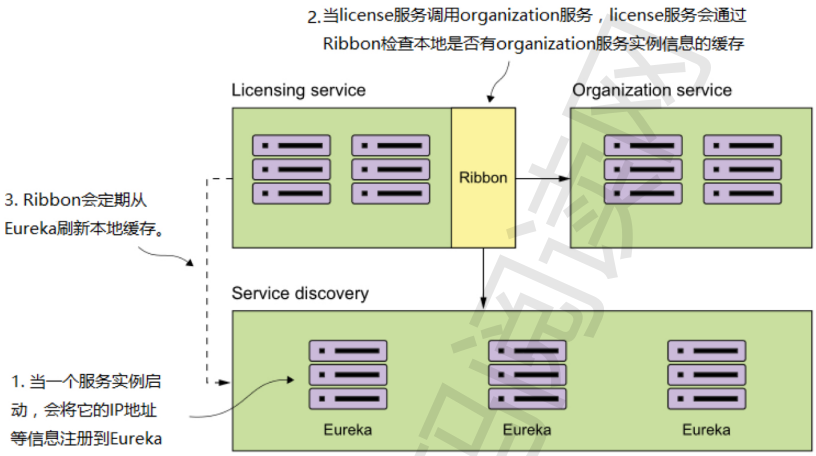
在Spring Cloud中使用Feign, 我们可以做到使用HTTP请求远程服务时能与调用本地方法一样的编码体验，开发者完全感知不到这是远程方法，更感知不到这是个HTTP请求。

3. 注册中心心跳是几秒

1、Eureka的客户端默认每隔30s会向eureka服务端更新实例，注册中心也会定时进行检查，发现某个实例默认90s内没有再收到心跳，会注销此实例，这些时间间隔是可配置的。

2、不过注册中心还有一个保护模式（服务端在短时间内丢失过多客户端的时候，就会进入保护模式），在这个保护模式下，他会认为是网络问题，不会注销任何过期的实例。

4. 消费者是如何发现服务提供者的



a. 当一个服务实例启动，会将它的ip地址等信息注册到eureka；

b. 当a服务调用b服务，a服务会通过Ribbon检查本地是否有b服务实例信息的缓存；

c. Ribbon会定期从eureka刷新本地缓存。

5. 多个消费者调用同一接口，eruka默认的分配方式是什么

a. RoundRobinRule:轮询策略，Ribbon以轮询的方式选择服务器，这个是默认值。所以示例中所启动的两个服务会被循环访问;

b. RandomRule:随机选择，也就是说Ribbon会随机从服务器列表中选择一个进行访问;

c. BestAvailableRule:最大可用策略，即先过滤出故障服务器后，选择一个当前并发请求数最小的;

d. WeightedResponseTimeRule:带有加权的轮询策略，对各个服务器响应时间进行加权处理，然后在采用轮询的方式来获取相应的服务器;

e. AvailabilityFilteringRule:可用过滤策略，先过滤出故障的或并发请求大于阈值一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个;

f. ZoneAvoidanceRule:区域感知策略，先使用主过滤条件（区域负载器，选择最优区域）对所有实例过滤并返回过滤后的实例清单，依次使用次过滤条件列表中的过滤条件对主过滤条件的结果进行过滤，判断最小过滤数（默认1）和最小过滤百分比（默认0），最后对满足条件的服务器则使用RoundRobinRule(轮询方式)选择一个服务器实例。

6. 说说常用的springboot注解，及其实现?

a. @Bean：注册Bean

i. 默认使用方法名作为id，可以在后面定义id如@Bean("xx")；

ii. 默认为单例。

iii. 可以指定init方法和destroy方法：

1. 对象创建和赋值完成，调用初始化方法；
2. 单实例bean在容器销毁的时候执行destroy方法；
3. 多实例bean，容器关闭是不会调用destroy方法。

b. @Scope：Bean作用域

i. 默认为singleton；

ii. 类型：

1. singleton单实例（默认值）：ioc容器启动时会调用方法创建对象放到ioc容器中，以后每次获取就是直接从容器中拿实例；
2. prototype多实例：ioc容器启动不会创建对象，每次获取时才会调用方法创建实例；
3. request同一次请求创建一个实例；
4. session同一个session创建一个实例。

c. @Value：给变量赋值

i. 代码：

```
import org.springframework.beans.factory.annotation.Value;

public class Person extends BaseEntity{
    @Value("xuan")
    private String name;
    @Value("27")
```

```
    private int age;
    @Value("#{20-7}")
    private int count;
    @Value("${person.nickName}")
    private String nickName;
}
```

i. 方式：

1. 基本数字
2. 可以写SpEL（Spring EL表达式）：#{}
3. 可以写${}，取出配置文件中的值（在运行环境变量里面的值）

d. @Autowired：自动装配

i. 默认优先按照类型去容器中找对应的组件：BookService bookService = applicationContext.getBean(BookService.class);

ii. 默认一定要找到，如果没有找到则报错。可以使用@Autowired(required = false)标记bean为非必须的。

iii. 如果找到多个相同类型的组件，再根据属性名称去容器中查找。

iv. @Qualifier("bookDao2")明确的指定要装配的bean。

v. @Primary：让spring默认装配首选的bean，也可以使用@Qualifier()指定要装配的bean。

e. @Profile：环境标识

i. Spring为我们提供的可以根据当前环境，动态的激活和切换一系列组件的功能；

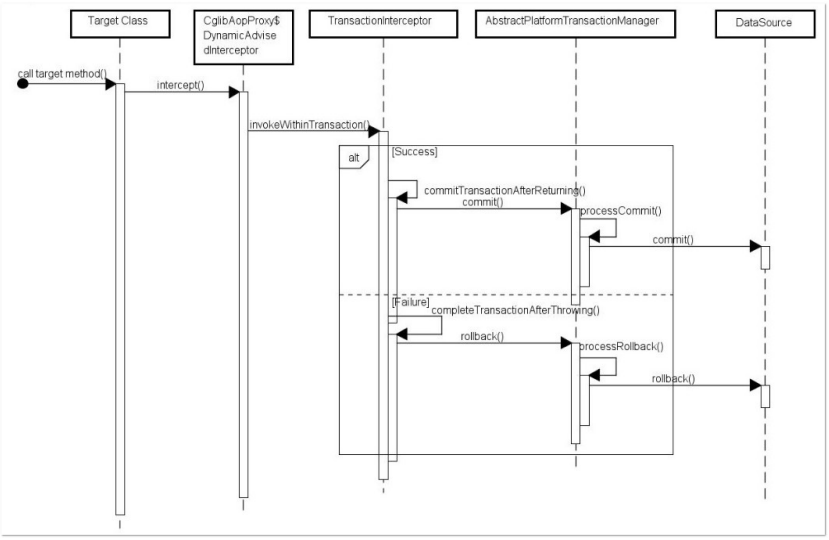
ii. @Profile指定组件在哪个环境才能被注册到容器中，默认为"default"@Profile("default")。

iii. 激活方式：

1. 运行时添加虚拟机参数：-Dspring.profiles.active=test
2. 代码方式：

7. spring的事务注解是什么？什么情况下事物才会回滚

a. spring事务实现机制：



b. 事务注解@transactional；

c. 默认情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常）或者 Error，则 Spring 将回滚事务。

d. @Transactional 只能应用到 public 方法才有效：只有@Transactional 注解应用到 public 方法，才能进行事务管理。这是因为在使用 Spring AOP 代理时，Spring 在调用在图 1 中的 TransactionInterceptor 在目标方法执行前后进行拦截之前，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的的 intercept 方法或 JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource（Spring 通过这个类获取表 1. @Transactional 注解的事务属性配置属性信息）的 computeTransactionAttribute 方法。

8. 说说spring事物的传播性和隔离级别

a. 七个事传播属性：

1. PROPAGATION_REQUIRED -- 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
2. PROPAGATION_SUPPORTS -- 支持当前事务，如果当前没有事务，就以非事务方式执行。
3. PROPAGATION_MANDATORY -- 支持当前事务，如果当前没有事务，就抛出异常。
4. PROPAGATIONREQUIRESNEW -- 新建事务，如果当前存在事务，把当前事务挂起。

5. PROPAGATIONNOTSUPPORTED -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
6. PROPAGATION_NEVER -- 以非事务方式执行，如果当前存在事务，则抛出异常。
7. PROPAGATIONNESTED -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATIONREQUIRED类似的操作。

b. 五个隔离级别：

1. ISOLATION_DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.
2. 另外四个与JDBC的隔离级别相对应：
3. ISOLATION*READ*UNCOMMITTED 这是事务最低的隔离级别，它充许别外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻读。
4. ISOLATION*READ*COMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻读。
5. ISOLATION*REPEATABLE*READ 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免不可重复读。
6. ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻读。

c. 关键词：

隔离级别	脏读（Dirty Read）	不可重复读（NonRepeatable Read）	幻读（Phantom Read）
未提交读（Read uncommitted）	可能	可能	可能
已提交读（Read committed）	不可能	可能	可能
可重复读（Repeatable read）	不可能	不可能	可能
可串行化（Serializable ）	不可能	不可能	不可能

1、脏读（新增或删除）：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据；

```
1.Mary的原工资为1000，财务人员将Mary的工资改为了8000(但未提交事务)
2.Mary读取自己的工资 ,发现自己的工资变为了8000，欢天喜地!
3.而财务发现操作有误，回滚了事务,Mary的工资又变为了1000
  像这样,Mary记取的工资数8000是一个脏数据。
```

2、不可重复读（修改）：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。（解决：只有在修改事务完全提交之后才可以读取数据，则可以避免该问题）；

```
1.在事务1中，Mary 读取了自己的工资为1000,操作并没有完成
2.在事务2中，这时财务人员修改了Mary的工资为2000,并提交了事务.
3.在事务1中，Mary 再次读取自己的工资时，工资变为了2000
```

3、幻读（新增或删除）：是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样（解决：如果在操作事务完成数据处理之前，任何其他事务都不可以添加新数据，则可避免该问题）。

```
目前工资为1000的员工有10人。
1.事务1,读取所有工资为1000的员工。
2.这时事务2向employee表插入了一条员工记录，工资也为1000
3.事务1再次读取所有工资为1000的员工 共读取到了11条记录。
```

9. mysql的引擎有什么？他们的区别

a. InnoDB：

1. 支持事务处理
2. 支持外键
3. 支持行锁
4. 不支持FULLTEXT类型的索引（在Mysql5.6已引入）
5. 不保存表的具体行数，扫描表来计算有多少行
6. 对于AUTO_INCREMENT类型的字段，必须包含只有该字段的索引
7. DELETE 表时，是一行一行的删除
8. InnoDB 把数据和索引存放在表空间里面

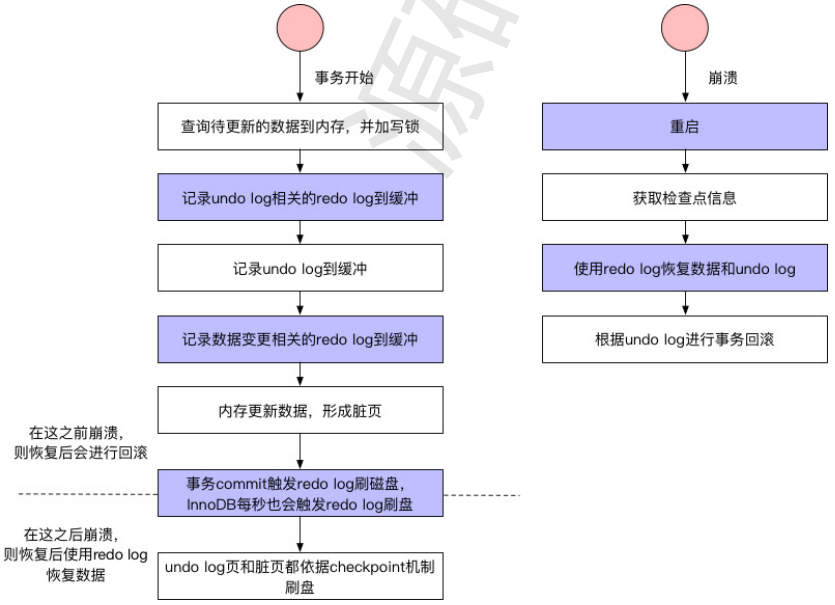
9. 跨平台可直接拷贝使用
10. 表格很难被压缩

b. MyISAM：

1. 不支持事务，回滚将造成不完全回滚，不具有原子性
2. 不支持外键
3. 支持全文搜索
4. 保存表的具体行数,不带where时，直接返回保存的行数
5. DELETE 表时，先drop表，然后重建表
6. MyISAM 表被存放在三个文件 。frm 文件存放表格定义。 数据文件是MYD (MYData) 。 索引文件是MYI (MYIndex)引伸
7. 跨平台很难直接拷贝
8. AUTO_INCREMENT类型字段可以和其他字段一起建立联合索引
9. 表格可以被压缩

c. 选择：因为MyISAM相对简单所以在效率上要优于InnoDB.如果系统读多，写少。对原子性要求低。那么MyISAM最好的选择。且MyISAM恢复速度快。可直接用备份覆盖恢复。如果系统读少，写多的时候，尤其是并发写入高的时候。InnoDB就是首选了。两种类型都有自己优缺点，选择那个完全要看自己的实际类弄。

10. innodb如何实现mysql的事务

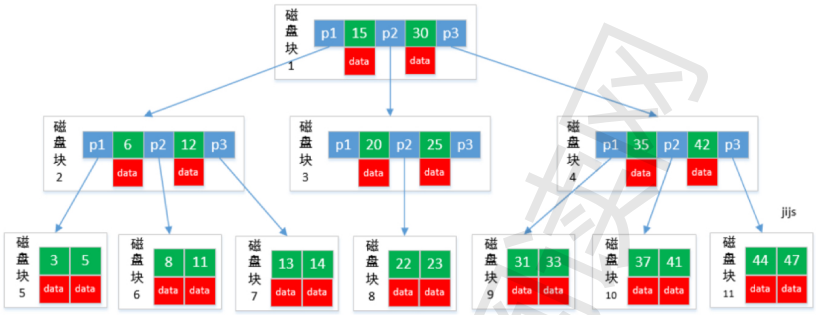


事务进行过程中，每次sql语句执行，都会记录undo log和redo log，然后更新数据形成脏页，然后redo log按照时间或者空间等条件进行落盘，undo log和脏页按照checkpoint进行落盘，落盘后相应的redo log就可以删除了。此时，事务还未COMMIT，如果发生崩溃，则首先检查checkpoint记录，使用相应的redo log进行数据和undo log的恢复，然后查看undo log的状态发现事务尚未提交，然后就使用undo log进行事务回滚。事务执行COMMIT操作时，会将本事务相关的所有redo log都进行落盘，只有所有redo log落盘成功，才算COMMIT成功。然后内存中的数据脏页继续按照checkpoint进行落盘。如果此时发生了崩溃，则只使用redo log恢复数据。

11. mysql索引谈一谈

12. 说说b+树的原理

a. B-tree：

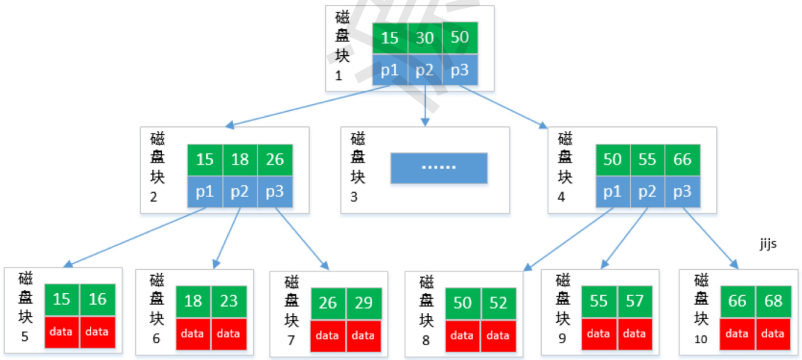


B-tree 利用了磁盘块的特性进行构建的树。每个磁盘块一个节点，每个节点包含了很关键字。把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B-tree巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页（每页为4K），这样每个节点只需要一次I/O就可以完全载入。

B-tree 的数据可以存在任何节点中。

1. B+tree：



B+tree 是 B-tree 的变种，B+tree 数据只存储在叶子节点中。这样在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定；

13. 让你设计一个索引，你会怎么设计

a. mysql默认存储引擎innodb只显式支持B树索引，对于频繁访问的表，innodb会透明建立自适应hash索引，即在B树索引基础上建立hash索引，可以显著提高查找效率，对于客户端是透明的，不可控制的，隐式的。

14. 还问了git和svn的区别

1、Git是分布式的，而Svn不是；

2、GIT把内容按元数据方式存储，而SVN是按文件

3、分支不同：git分支切换很方便；svn分支就是版本库的另外一个目录；

4、GIT没有一个全局的版本号，而svn有，SVN的版本号实际是任何一个相应时间的源代码快照。

5、GIT的内容完整性要优于SVN(GIT的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。)

15. git命令的一些底层原理以及流程

https://www.jianshu.com/p/2b47a3078a46

a. git init：使用git init初始化一个新的目录时，会生成一个.git的目录，该目录即为本地仓库。一个新初始化的本地仓库是这样的：

```
├── HEAD
├── branches
├── config
├── description
├── hooks
├── objects
|   ├── info
|   └── pack
```

```
└── refs
    ├── heads
    └── tags
```

- description用于GitWeb程序
- config配置特定于该仓库的设置（还记得git config的三个配置级别么）
- hooks放置客户端或服务端的hook脚本
- HEAD传说中的**HEAD指针**，指明当前处于哪个分支
- objectsGit对象存储目录
- refsGit引用存储目录
- branches放置分支引用的目录

其中description、config和hooks这些不在讨论中，后文会直接忽略。

b. git add：Git commit之前先要通过git add添加文件：

```
├── HEAD
├── branches
├── index
├── objects
│   ├── 9f
│   │   └── 4d96d5b00d98959ea9960f069585ce42b1349a
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

可以看到，多了一个index文件。并且在objects目录下多了一个9f的目录，其中多了一个4d96d5b00d98959ea9960f069585ce42b1349a文件。

其实9f4d96d5b00d98959ea9960f069585ce42b1349a就是一个Git对象，称为blob对象。

c. git commit：

```
├── HEAD
├── branches
├── index
├── logs
│   ├── HEAD
│   └── refs
│       └── heads
│           └── master
├── objects
│   ├── 88
│   │   └── 23efd7fa394844ef4af3c649823fa4aedefec5
│   ├── 91
│   │   └── 0fc16f5cc5a91e6712c33aed4aad2cfffccb73
│   ├── 9f
│   │   └── 4d96d5b00d98959ea9960f069585ce42b1349a
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   └── master
    └── tags
```